МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
МИКОЛАЇВСЬКИЙ НАЦІОНАЛЬНИЙ АГРАРНИЙ УНІВЕРСИТЕТ
Факультет менеджменту
Кафедра економічної кібернетики, комп'ютерних наук та інформаційних технологій

БАЗИ ДАНИХ

Конспект лекцій
для здобувачів першого (бакалаврського) рівня вищої освіти
ОПП «Комп'ютерні науки» спеціальності 122 «Комп'ютерні науки» денної форми здобуття вищої освіти

МИКОЛАЇВ
2025

УДК 004.65

Б17

Друкується за рішенням науково-методичної комісії факультету менеджменту Миколаївського національного аграрного університету від 27 березня 2025 року, протокол № 7.

Укладачі:

Рецензенти:

С. І. Тищенко — к.п.н., доцент, доцент кафедри економічної кібернетики, комп'ютерних наук та інформаційних технологій Миколаївського національного аграрного університету

О. Ю. Пархоменко — к.ф.-м.н., доцент, доцент кафедри економічної кібернетики, комп'ютерних наук та інформаційних технологій Миколаївського національного аграрного університету

О. О. Жебко — асистент кафедри економічної кібернетики, комп'ютерних наук та інформаційних технологій, Миколаївського національного аграрного університету

Ю. В. Волосюк — к.т.н., доцент, заступник сільського голови з питань відбудови та цифрової трансформації громади Шевченківської сілької ради Миколаївської області

О. С. Садовий — к.т.н., доцент, завідувач кафедри агроінженерії Миколаївського національного аграрного університету

Б17 **Бази** даних: конспект лекцій для здобувачів першого (бакалаврського) рівня вищої освіти ОПП «Комп'ютерні науки» спец. 122 «Комп'ютерні науки» денної форми здобуття вищої освіти / уклад. : С. І. Тищенко, О. Ю. Пархоменко, О. О. Жебко. Миколаїв : МНАУ, 2025. 242 с.

Конспект лекцій призначений для вивчення теоретичних основ і практичних методів проектування баз даних, ознайомлення з принципами побудови інформаційних систем, моделями даних та мовами запитів. Містить навчальні матеріали з основних тем курсу «Бази даних», що передбачені освітньо-професійною програмою підготовки здобувачів першого (бакалаврського) рівня вищої освіти спеціальності 122 «Комп'ютерні науки», галузі знань 12 «Інформаційні технології».

ЗМІСТ

ПЕРЕДМОВА

Курс дисципліни «Бази даних» призначений для формування у здобувачів вищої освіти спеціальності 122 «Комп'ютерні науки» цілісного уявлення про принципи побудови, функціонування та адміністрування сучасних систем зберігання та обробки даних, що працюють у реляційних та нереляційних середовищах.

Предметом вивчення дисципліни є закономірності організації структурованих даних, моделі даних, методи проектування логічних і фізичних схем, а також мови маніпулювання даними.

Об'єктом вивчення є системи керування базами даних (СКБД), реляційні та об'єктно-орієнтовані моделі, транзакційні процеси та архітектури інформаційних сховищ.

Метою викладання дисципліни є підготовка висококваліфікованих фахівців, здатних ефективно проектувати, аналізувати та розробляти архітектури баз даних, розуміти принципи їхньої оптимізації, забезпечувати цілісність, безпеку та швидкий доступ до інформації в інформаційних системах різного масштабу.

Основними завданнями, що мають бути вирішені в процесі викладання дисципліни, є:

– ознайомлення студентів із базовими поняттями теорії баз даних, життєвим циклом розробки ПЗ та сучасними моделями представлення знань;

– формування знань щодо мови SQL (DDL, DML, DCL, TCL), механізмів нормалізації, індексування та управління транзакціями;

– навчання практичним методам розробки баз даних та взаємодії з ними через програмні інтерфейси (на прикладі PostgreSQL, MySQL або інших сучасних СКБД);

– формування навичок аналізу та оптимізації складних запитів, моніторингу продуктивності систем та забезпечення коректності конкурентного доступу до даних;

– розвиток умінь застосовувати набуті знання у моделюванні (ER-діаграми), тестуванні та побудові масштабованих архітектур, включаючи NoSQL та хмарні рішення.

Конспект лекцій складено з урахуванням вимог чинної освітньо-професійної програми підготовки фахівців зі спеціальності «Комп'ютерні науки», а також на основі сучасних підручників, навчальних посібників, стандартів SQL та нормативних документів, що регламентують галузь управління даними.

ЗМІСТОВИЙ МОДУЛЬ 1. МОДЕЛІ ДАНИХ ТА ЗАПИТИ ДО РЕЛЯЦІЙНОЇ МОДЕЛІ ДАНИХ. ПРОЕКТУВАННЯ БАЗ ДАНИХ У MS ACCESS

ТЕМА 1. ІНФОРМАЦІЙНІ СИСТЕМИ ТА СИСТЕМИ УПРАВЛІННЯ БАЗАМИ ДАНИХ

1. Поняття інформації та інформаційної системи.
2. Класифікація інформаційних систем.
3. Історія розвитку баз даних.
4. Логічна та фізична структура баз даних.
5. Архітектура інформаційної системи.
6. Банки даних.
7. Бази даних та системи управління базами даних.
8. Архітектура СУБД.
9. Функції СУБД.
10. Мовні засоби СУБД: мова структурованих запитів та її підмови

1. ПОНЯТТЯ ІНФОРМАЦІЇ ТА ІНФОРМАЦІЙНОЇ СИСТЕМИ

Інформація є одним із фундаментальних понять сучасної нау бою відомості про об'єкти, явища, процеси та події навколишнього світу, що зменшують наявну невизначеність або збагачують наші уявлення про них.

У контексті інформаційних систем інформацію розглядають як сукупність ки та технології. У широкому розумінні інформація представляє со

даних, які організовані певним чином і мають значення для користувача. Дані являють собою факти, числа, текст чи інші символи у формі, придатній для обробки, зберігання та передавання. Перетворення даних у корисну інформацію відбувається через процеси обробки, аналізу та інтерпретації відповідно до конкретних потреб користувача.

Властивості інформації визначають її цінність для прийняття рішень та функціонування організацій:

Достовірність означає відповідність інформації реальному стану справ. Недостовірна інформація може призвести до прийняття неправильних рішень і завдати значних збитків організації.

Повнота характеризує достатність інформації для прийняття обґрунтованих рішень. Неповна інформація може призвести до неправильної оцінки ситуації, тоді як надлишкова інформація ускладнює процес обробки та аналізу.

Актуальність відображає відповідність інформації поточному моменту часу. У динамічному бізнес-середовищі застаріла інформація може виявитися марною або навіть шкідливою.

Доступність визначає можливість отримання інформації користувачем у потрібний момент часу. Навіть найціннішу інформацію марно мати, якщо вона недоступна тоді, коли вона потрібна.

Релевантність показує відповідність інформації запиту або проблемі, яку необхідно вирішити. Нерелевантна інформація лише відволікає увагу та знижує ефективність роботи.

Інформаційна система являє собою організовану сукупність технічних, програмних, людських та інформаційних ресурсів, призначених для збирання, зберігання, обробки та поширення інформації з метою підтримки прийняття рішень та управління в організації.

Основними компонентами інформаційної системи виступають технічне забезпечення (комп'ютери, сервери, мережеве обладнання), програмне забезпечення (операційні системи, системи управління базами даних, прикладні програми), інформаційне забезпечення (бази даних, файли, документи), організаційне забезпечення (процедури, регламенти, інструкції) та персонал (користувачі, адміністратори, розробники).

Інформаційні системи виконують такі основні функції як введення інформації з різних джерел, зберігання інформації у структурованому вигляді, обробку інформації для перетворення даних у корисну форму, передачу інформації між компонентами системи та користувачами, а також представлення інформації у зручному для користувача вигляді.

2. КЛАСИФІКАЦІЯ ІНФОРМАЦІЙНИХ СИСТЕМ

Інформаційні системи можна класифікувати за різними ознаками, що дозволяє краще розуміти їх призначення, можливості та особливості застосування.

За рівнем управління інформаційні системи поділяються на системи операційного рівня, які обслуговують оперативний персонал та забезпечують реєстрацію та обробку поточних операцій організації. Системи тактичного рівня призначені для менеджерів середньої ланки та забезпечують моніторинг, контроль і прийняття тактичних рішень. Системи стратегічного рівня обслуговують вище керівництво організації та підтримують прийняття стратегічних рішень, довгострокове планування.

За функціональним призначенням виділяють інформаційно-пошукові системи, що забезпечують зберігання, пошук та видачу інформації за запитом користувача. Інформаційно-довідкові системи надають довідкову інформацію у зручній формі. Системи обробки даних виконують регулярну обробку великих обсягів інформації. Системи підтримки прийняття рішень допомагають аналізувати ситуацію та вибирати оптимальні рішення. Експертні системи використовують знання експертів для вирішення складних задач. Системи підтримки керівника надають інформацію у формі, зручній для топ-менеджменту.

За ступенем автоматизації розрізняють ручні інформаційні системи, де всі операції виконуються людиною без використання технічних засобів. Автоматизовані системи передбачають використання комп'ютерної техніки разом із ручною обробкою. Автоматичні системи виконують всі операції без участі людини.

За характером обробки інформації системи можуть бути інформаційно-пошуковими, розрахунковими, що виконують математичні розрахунки та моделювання, та інтелектуальними, здатними до навчання та адаптації.

За масштабом виділяють персональні системи для індивідуального використання, групові системи для робочих груп та відділів, корпоративні системи масштабу всієї організації та глобальні системи, що охоплюють декілька організацій або регіонів.

За сферою застосування інформаційні системи можуть бути економічними, що обслуговують економічну діяльність підприємств, медичними для лікувальних закладів, освітніми для навчальних закладів, науковими для наукових досліджень, військовими для оборонних потреб тощо.

3. ІСТОРІЯ РОЗВИТКУ БАЗ ДАНИХ

Історія розвитку баз даних тісно пов'язана з еволюцією комп'ютерних технологій та зростаючими потребами в ефективному управлінні даними.

Перший етап (1960-ті роки) характеризувався появою перших комерційних систем управління базами даних. У цей період панували файлові системи, де дані зберігалися у послідовних файлах. Основною проблемою була надмірна залежність програм від структури даних. Зміна формату даних вимагала перероблення всіх програм, що з ними працювали. Крім того, існувала значна надмірність даних, оскільки різні програми зберігали копії однієї й тієї ж інформації. Виникли перші ієрархічні системи управління базами даних, такі як

IMS (Information Management System) компанії IBM, розроблена для програми Apollo.

Другий етап (1970-ті роки) ознаменувався революційною роботою Едгара Кодда, який у 1970 році опублікував статтю "A Relational Model of Data for Large Shared Data Banks", що заклала теоретичні основи реляційних баз даних. Реляційна модель представляла дані у вигляді таблиць, що значно спрощувало роботу з інформацією. У цей період також розвивалися мережні моделі даних, представлені стандартом CODASYL. Були створені перші прототипи реляційних СУБД, такі як System R в IBM та INGRES в Каліфорнійському університеті в Берклі.

Третій етап (1980-ті роки) характеризувався комерціалізацією реляційних СУБД. Компанія Oracle випустила першу комерційну реляційну СУБД у 1979 році. IBM представила DB2 у 1983 році. З'явилися настільні СУБД для персональних комп'ютерів, такі як dBase, FoxPro та Microsoft Access. Була стандартизована мова SQL (Structured Query Language), що стала універсальним засобом роботи з реляційними базами даних. Реляційні СУБД почали витісняти ієрархічні та мережні системи завдяки простоті використання та гнучкості.

Четвертий етап (1990-ті роки) відзначився появою об'єктно-орієнтованих та об'єктно-реляційних СУБД, що поєднували переваги реляційного підходу з можливостями об'єктно-орієнтованого програмування. Розвивалися розподілені бази даних, що дозволяли розміщувати дані на різних серверах. З'явилися сховища даних (Data Warehouses) для аналітичної обробки великих обсягів історичних даних. Інтернет-технології призвели до необхідності інтеграції СУБД з веб-додатками.

П'ятий етап (2000-ті роки) характеризувався появою XML-баз даних для роботи з напівструктурованими даними. Почали розвиватися NoSQL системи для обробки величезних обсягів неструктурованих даних. З'явилися хмарні бази даних, що надають послуги зберігання та обробки даних через Інтернет. Розвивалися технології Big Data для роботи з надвеликими масивами інформації.

Сучасний етап (2010-ті - 2020-ті роки) відзначається появою NewSQL систем, що поєднують масштабованість NoSQL з підтримкою транзакцій та гарантіями узгодженості реляційних СУБД. Активно розвиваються графові бази даних для аналізу зв'язків між об'єктами. Штучний інтелект та машинне навчання інтегруються в СУБД для автоматичної оптимізації запитів та виявлення аномалій. Блокчейн-технології знаходять застосування в розподілених базах даних. Квантові обчислення відкривають нові можливості для роботи з даними.

4. ЛОГІЧНА ТА ФІЗИЧНА СТРУКТУРА БАЗ ДАНИХ

Розуміння різниці між логічною та фізичною структурами баз даних є критично важливим для ефективного проектування та використання інформаційних систем.

Логічна структура бази даних описує, як дані організовані з точки зору користувача та прикладних програм. Вона визначає типи даних, зв'язки між ними, обмеження цілісності та бізнес-правила незалежно від способу фізичного зберігання. Логічна структура представляє концептуальне бачення даних, абстрагуючись від деталей реалізації.

Основними елементами логічної структури виступають сутності, що представляють об'єкти реального світу (клієнти, замовлення, продукти). Атрибути описують властивості сутностей (ім'я клієнта, дата замовлення, ціна продукту). Зв'язки визначають асоціації між сутностями (клієнт робить замовлення, замовлення містить продукти). Обмеження цілісності забезпечують коректність та узгодженість даних (унікальність ідентифікаторів, обов'язковість певних полів, діапазони допустимих значень).

У реляційних базах даних логічна структура представлена таблицями (відношеннями), де рядки відповідають записам (кортежам), а стовпці - атрибутам (полям). Кожна таблиця має первинний ключ, що однозначно ідентифікує кожен запис. Зв'язки між таблицями реалізуються через зовнішні ключі.

Фізична структура бази даних визначає, як дані фактично зберігаються на носіях інформації. Вона включає організацію файлів, методи доступу до даних, індексацію, розподіл пам'яті та інші деталі реалізації, оптимізовані для швидкодії та ефективного використання ресурсів.

Елементами фізичної структури є файли бази даних, що зберігають таблиці та індекси на диску. Сторінки даних представляють мінімальні одиниці читання/запису, зазвичай розміром 4-16 Кб. Індекси прискорюють пошук даних, створюючи впорядковані структури для швидкого доступу. Журнали транзакцій зберігають інформацію про зміни для забезпечення відновлення даних. Буфери пам'яті кешують часто використовувані дані для зменшення звернень до диска.

Незалежність даних є ключовою концепцією, що забезпечує можливість зміни однієї структури без впливу на іншу. Логічна незалежність дозволяє змінювати логічну структуру (додавати нові таблиці або поля) без зміни існуючих додатків. Фізична незалежність дозволяє змінювати спосіб зберігання

даних (переносити файли на інші диски, змінювати методи індексації) без впливу на логічну структуру та додатки.

Переваги розділення логічної та фізичної структур включають спрощення проектування, оскільки розробники можуть зосередитися на бізнес-логіці без занурення в технічні деталі зберігання. Це забезпечує гнучкість, дозволяючи оптимізувати продуктивність через зміни фізичної структури без переробки додатків. Підтримується переносимість додатків між різними платформами та СУБД. Спрощується супроводження системи завдяки можливості незалежного вдосконалення окремих рівнів.

5. АРХІТЕКТУРА ІНФОРМАЦІЙНОЇ СИСТЕМИ

Архітектура інформаційної системи визначає її структурну організацію, компоненти та взаємодію між ними. Правильна архітектура забезпечує масштабованість, надійність, продуктивність та легкість супроводження системи.

Трирівнева архітектура є класичною моделлю побудови інформаційних систем, запропонованою American National Standards Institute (ANSI) та Міжнародною організацією зі стандартизації (ISO). Вона складається з трьох незалежних рівнів, кожен з яких виконує специфічні функції.

Рівень представлення (Presentation Layer) відповідає за взаємодію з користувачем. Цей рівень включає користувацький інтерфейс, форми введення даних, звіти, графіки та інші елементи візуалізації інформації. Основними функціями є отримання введення від користувача, перевірка коректності введених даних на початковому рівні, форматування та відображення даних у зручному вигляді, обробка подій інтерфейсу. Цей рівень може бути реалізований як настільний додаток, веб-інтерфейс або мобільний додаток.

Рівень бізнес-логіки (Business Logic Layer) містить основну функціональність системи. Він обробляє запити від рівня представлення, виконує необхідні обчислення та маніпуляції з даними, застосовує бізнес-правила та обмеження, координує роботу різних компонентів системи, керує транзакціями та забезпечує цілісність операцій. Цей рівень інкапсулює логіку додатка, роблячи його незалежним від способу представлення та зберігання даних.

Рівень даних (Data Layer) відповідає за зберігання та управління даними. Він включає системи управління базами даних, файлові системи, сховища даних. Основними функціями є безпечне та надійне зберігання інформації,

забезпечення цілісності та узгодженості даних, ефективний доступ до даних, резервне копіювання та відновлення, управління транзакціями на рівні СУБД.

Переваги трирівневої архітектури включають модульність, що дозволяє розробляти та тестувати кожен рівень незалежно. Забезпечується масштабованість через можливість розподілу рівнів на різні сервери та збільшення потужності кожного рівня окремо. Підтримується можливість повторного використання компонентів у різних додатках. Спрощується супроводження, оскільки зміни на одному рівні не вимагають переробки інших. Покращується безпека завдяки можливості контролювати доступ на кожному рівні.

Клієнт-серверна архітектура є практичною реалізацією багаторівневого підходу. У дворівневій архітектурі клієнтський додаток містить інтерфейс та бізнес-логіку, а сервер забезпечує зберігання даних. У трирівневій архітектурі рівень представлення розміщується на клієнті, бізнес-логіка - на сервері додатків, а дані - на сервері баз даних. У багаторівневій архітектурі функціональність розподіляється між більшою кількістю серверів для досягнення максимальної гнучкості та продуктивності.

Сучасні архітектурні підходи включають сервіс-орієнтовану архітектуру (SOA), де функціональність надається у вигляді незалежних сервісів, доступних через стандартизовані інтерфейси. Мікросервісна архітектура розбиває додаток на невеликі незалежні сервіси, кожен з яких може розроблятися, розгортатися та масштабуватися окремо. Хмарна архітектура використовує віддалені ресурси та послуги, доступні через Інтернет, забезпечуючи еластичність та економічну ефективність.

6. БАНКИ ДАНИХ

Банк даних являє собою комплексну систему, що включає не лише самі дані, але й програмні, технічні та організаційні засоби для їх ефективного збирання, зберігання, обробки та надання користувачам.

Структура банку даних включає декілька взаємопов'язаних компонентів. База даних містить структуровану сукупність даних предметної області. Система управління базами даних забезпечує програмні засоби для роботи з даними. Словник даних (метадані) описує структуру та характеристики даних, що зберігаються в базі. Адміністратор бази даних відповідає за проектування, підтримку та забезпечення безпеки. Прикладні програми реалізують специфічну функціональність для різних груп користувачів. Кінцеві користувачі працюють з даними через інтерфейси прикладних програм.

Словник даних відіграє критично важливу роль у функціонуванні банку даних. Він містить інформацію про структуру таблиць та зв'язки між ними, описує типи даних, допустимі значення та формати, зберігає правила цілісності та обмеження, документує права доступу різних користувачів до об'єктів бази даних, фіксує інформацію про індекси та способи оптимізації запитів.

Функції банку даних охоплюють централізоване зберігання даних, що забезпечує їх доступність для багатьох користувачів та додатків. Контроль цілісності гарантує коректність та узгодженість інформації. Управління доступом забезпечує безпеку через розмежування прав користувачів. Підтримка одночасної роботи багатьох користувачів досягається через механізми транзакцій та блокувань. Резервне копіювання та відновлення захищають від втрати даних. Оптимізація продуктивності досягається через індексацію, кешування та інші методи.

Переваги використання банків даних включають зменшення надмірності даних, оскільки інформація зберігається в одному місці без дублювання. Забезпечується узгодженість даних через централізоване управління. Покращується доступність інформації для авторизованих користувачів. Спрощується супроводження завдяки централізованому управлінню змінами. Підвищується безпека через комплексну систему захисту. Збільшується продуктивність за рахунок оптимізованих методів доступу.

Розподілені банки даних набувають все більшого значення в сучасних інформаційних системах. Дані розміщуються на різних серверах, можливо, в різних географічних локаціях. Забезпечується прозорість для користувачів, які працюють з даними, не знаючи про їх фізичне розташування. Підвищується надійність через реплікацію даних на різних вузлах. Покращується продуктивність завдяки розподіленню навантаження. Виникають складності із забезпеченням узгодженості даних та управлінням розподіленими транзакціями.

7. БАЗИ ДАНИХ ТА СИСТЕМИ УПРАВЛІННЯ БАЗАМИ ДАНИХ

База даних та система управління базами даних є взаємопов'язаними, але різними концепціями, розуміння яких критично важливе для ефективної роботи з інформаційними системами.

База даних (БД) являє собою організовану сукупність структурованих даних, що зберігаються в комп'ютерній системі. База даних представляє конкретні дані предметної області, організовані відповідно до певної моделі даних. Вона включає не лише самі дані, але й зв'язки між ними, обмеження цілісності та метадані, що описують структуру даних.

Характеристиками бази даних є структурованість, що означає організацію даних згідно з певною моделлю. Інтегрованість передбачає об'єднання даних з різних джерел у єдину систему. Багатоцільове використання дозволяє обслуговувати потреби різних користувачів та додатків. Спільне використання забезпечує одночасний доступ багатьох користувачів. Мінімальна надмірність означає відсутність непотрібного дублювання даних.

Система управління базами даних (СУБД) є програмним забезпеченням, що надає засоби для створення, підтримки та використання баз даних. СУБД виступає посередником між користувачами, прикладними програмами та даними, забезпечуючи ефективний та безпечний доступ до інформації.

Основні функції СУБД включають визначення даних, що дозволяє описувати структуру бази даних, типи даних, зв'язки та обмеження. Маніпулювання даними надає операції додавання, видалення, зміни та вибірки інформації. Управління транзакціями забезпечує атомарність, узгодженість, ізольованість та довговічність операцій. Контроль цілісності перевіряє дотримання обмежень та бізнес-правил. Управління доступом контролює права користувачів на читання, зміну та видалення даних. Відновлення після збоїв забезпечує повернення бази даних до узгодженого стану після апаратних або програмних помилок. Оптимізація запитів автоматично вибирає найефективніший план виконання операцій.

Компоненти СУБД включають компілятор мови визначення даних (DDL), що обробляє команди створення та зміни структури бази даних. Інтерпретатор мови маніпулювання даними (DML) виконує запити на вибірку та модифікацію даних. Оптимізатор запитів аналізує запити та вибирає оптимальний план виконання. Менеджер транзакцій координує виконання транзакцій та забезпечує їх властивості. Менеджер буферів управляє кешем даних в оперативній пам'яті. Менеджер файлів здійснює фізичний доступ до даних на диску. Менеджер блокувань координує одночасний доступ користувачів до даних. Менеджер відновлення забезпечує відновлення після збоїв.

Класифікація СУБД може здійснюватися за різними критеріями. За моделлю даних розрізняють ієрархічні, мережні, реляційні, об'єктно-орієнтовані, об'єктно-реляційні, документо-орієнтовані, ключ-значення, графові СУБД. За архітектурою виділяють централізовані системи на одному комп'ютері, клієнт-серверні системи з розподілом функцій між клієнтом та сервером, розподілені системи з даними на декількох серверах. За кількістю користувачів існують однокористувацькі та багатокористувацькі системи. За способом доступу

розрізняють файл-серверні, де клієнт обробляє дані, отримані з файлового сервера, та клієнт-серверні системи, де обробка відбувається на сервері.

Популярні СУБД на ринку включають Oracle Database - потужну комерційну СУБД для великих підприємств. Microsoft SQL Server інтегрується з продуктами Microsoft та широко використовується в корпоративному середовищі. MySQL є найпопулярнішою відкритою реляційною СУБД. PostgreSQL відзначається розширеними можливостями та дотриманням стандартів SQL. MongoDB представляє документо-орієнтовані NoSQL бази даних. SQLite є вбудованою СУБД для мобільних та настільних додатків. Microsoft Access призначена для невеликих баз даних на персональних комп'ютерах.

8. АРХІТЕКТУРА СУБД

Архітектура СУБД визначає внутрішню організацію системи управління базами даних та взаємодію її компонентів. Розуміння архітектури допомагає ефективно проектувати, налаштовувати та використовувати СУБД.

Трирівнева архітектура ANSI-SPARC є фундаментальною моделлю організації СУБД, запропонованою комітетом ANSI Standards Planning and Requirements Committee. Вона забезпечує незалежність даних через розділення уявлень про дані на три рівні.

Зовнішній рівень (External Level) представляє уявлення окремих користувачів або груп користувачів про базу даних. Кожен користувач бачить лише ті дані та в такому форматі, які необхідні для його роботи. На цьому рівні визначаються зовнішні схеми (представлення, views), що описують структуру даних для конкретних додатків або користувачів. Один користувач може бачити дані у вигляді таблиць, інший - у вигляді форм або звітів. Різні користувачі можуть мати різні представлення одних і тих же даних залежно від їхніх потреб та прав доступу.

Концептуальний рівень (Conceptual Level) описує структуру всієї бази даних для спільноти користувачів. Концептуальна схема визначає всі сутності, їх атрибути, зв'язки між сутностями, обмеження цілісності, правила безпеки та інші логічні аспекти бази даних. Цей рівень абстрагується від деталей фізичного зберігання та особливостей окремих користувацьких уявлень. Концептуальна схема є єдиною для всієї бази даних і описує, які дані зберігаються та як вони пов'язані між собою.

Внутрішній рівень (Internal Level) визначає фізичну організацію даних на носіях інформації. Внутрішня схема описує структури зберігання даних,

методи доступу, індекси, організацію файлів, розміщення даних на диску, методи стиснення та шифрування. Цей рівень максимально наближений до фізичного зберігання даних і оптимізується для забезпечення продуктивності.

Відображення між рівнями забезпечує незалежність даних. Зовнішньо-концептуальне відображення перетворює запити від зовнішнього рівня до концептуального, дозволяючи створювати різні представлення одних і тих же даних. Концептуально-внутрішнє відображення перетворює концептуальні структури у внутрішні, дозволяючи змінювати фізичну організацію без впливу на логічну структуру.

Незалежність даних є ключовою перевагою трирівневої архітектури. Логічна незалежність означає, що зміни на концептуальному рівні (додавання нових сутностей або атрибутів) не вимагають зміни зовнішніх схем та додатків, які не використовують ці нові елементи. Фізична незалежність означає, що зміни на внутрішньому рівні (зміна методів доступу, створення індексів, переміщення файлів) не впливають на концептуальний та зовнішній рівні. Це дозволяє адміністраторам оптимізувати продуктивність системи без переробки додатків.

Модульна архітектура СУБД складається з взаємодіючих модулів, кожен з яких виконує специфічні функції. Інтерфейс користувача забезпечує взаємодію з користувачами через графічний або текстовий інтерфейс. Компілятор DDL обробляє команди визначення структури даних та оновлює словник даних. Інтерпретатор DML обробляє запити на маніпулювання даними. Процесор запитів аналізує та оптимізує запити користувачів. Планувальник запитів вибирає оптимальний план виконання. Менеджер транзакцій координує виконання транзакцій та забезпечує їх атомарність. Менеджер блокувань контролює одночасний доступ до даних. Менеджер буферів управляє кешем в оперативній пам'яті. Менеджер файлів здійснює фізичний доступ до даних на диску. Менеджер журналу веде запис всіх операцій для відновлення. Менеджер відновлення забезпечує повернення до узгодженого стану після збоїв.

Процес виконання запиту в СУБД проходить декілька етапів. Користувач або додаток подає запит до СУБД. Парсер аналізує синтаксис запиту та перевіряє правильність. Оптимізатор генерує різні плани виконання та вибирає найефективніший на основі статистики. Виконавчий механізм реалізує обраний план, звертаючись до менеджера буферів. Менеджер буферів перевіряє наявність потрібних даних в кеші або запитує їх у менеджера файлів. Менеджер файлів читає дані з диска або записує зміни. Результати повертаються користувачу.

9. ФУНКЦІЇ СУБД

Системи управління базами даних виконують широкий спектр функцій, що забезпечують ефективну роботу з даними. Розуміння цих функцій допомагає повноцінно використовувати можливості СУБД.

Функція визначення даних дозволяє описувати структуру бази даних. СУБД надає мову визначення даних (DDL - Data Definition Language) для створення таблиць, визначення типів даних для кожного поля, встановлення обмежень цілісності, створення індексів для прискорення пошуку, визначення представлень (views), створення процедур та тригерів. Усі метадані зберігаються в словнику даних, що дозволяє СУБД керувати структурою бази.

Функція маніпулювання даними забезпечує операції з даними. Мова маніпулювання даними (DML - Data Manipulation Language) включає команди для вставки нових записів, оновлення існуючих даних, видалення непотрібних записів, вибірки даних за заданими критеріями. СУБД оптимізує виконання цих операцій, використовуючи індекси та інші механізми для підвищення продуктивності.

Функція забезпечення цілісності даних гарантує коректність та узгодженість інформації. СУБД підтримує різні типи обмежень: обмеження домену визначають допустимі значення для атрибутів, обмеження сутності забезпечують унікальність первинних ключів, обмеження посилальної цілісності контролюють зв'язки між таблицями через зовнішні ключі, семантичні обмеження реалізують бізнес-правила. Тригери дозволяють автоматично виконувати дії при певних подіях для підтримки складних правил цілісності.

Функція управління транзакціями забезпечує коректне виконання послідовностей операцій. Транзакція є логічною одиницею роботи, яка повинна виконатися повністю або не виконатися взагалі. СУБД гарантує властивості ACID для транзакцій. Атомарність (Atomicity) означає, що всі операції транзакції виконуються як єдине ціле. Узгодженість (Consistency) забезпечує перехід бази даних з одного узгодженого стану в інший. Ізольованість (Isolation) означає, що одночасні транзакції не впливають одна на одну. Довговічність (Durability) гарантує, що результати завершеної транзакції зберігаються навіть при збоях системи.

Функція контролю одночасного доступу координує роботу багатьох користувачів. Механізми блокування запобігають конфліктам при одночасній зміні даних. Пессимістичне блокування захоплює ресурси перед їх використанням. Оптимістичне блокування дозволяє виконувати операції без блокування та перевіряє конфлікти перед завершенням транзакції. Рівні ізоляції

визначають ступінь ізольованості транзакцій: Read Uncommitted дозволяє читати незавершені зміни інших транзакцій, Read Committed дозволяє читати лише завершені зміни, Repeatable Read гарантує незмінність прочитаних даних протягом транзакції, Serializable забезпечує повну ізоляцію транзакцій.

Функція забезпечення безпеки захищає дані від несанкціонованого доступу. Система аутентифікації перевіряє особу користувача. Система авторизації визначає права доступу до об'єктів бази даних. Механізми шифрування захищають конфіденційні дані. Аудит реєструє всі дії користувачів для виявлення порушень безпеки.

Функція резервного копіювання та відновлення захищає від втрати даних. СУБД підтримує різні стратегії резервного копіювання: повне копіювання зберігає всі дані бази, інкрементне копіювання зберігає лише зміни з моменту останнього копіювання, диференційне копіювання зберігає зміни з моменту останнього повного копіювання. Журналізація транзакцій записує всі зміни для можливості відновлення до певного моменту часу. Механізми відновлення дозволяють повернути базу даних до узгодженого стану після збоїв.

Функція оптимізації продуктивності забезпечує ефективну роботу з даними. Оптимізатор запитів аналізує запити та вибирає найефективніший план виконання на основі статистики про дані. Індексація прискорює пошук даних, створюючи додаткові структури для швидкого доступу. Кешування зберігає часто використовувані дані в оперативній пам'яті. Секціонування розподіляє великі таблиці на менші частини для паралельної обробки.

Функція адміністрування надає засоби для управління СУБД. Моніторинг продуктивності відстежує використання ресурсів та виявляє вузькі місця. Налаштування параметрів дозволяє оптимізувати роботу системи. Управління користувачами включає створення облікових записів та призначення прав. Обслуговування включає реорганізацію даних, оновлення статистики, перебудову індексів.

10. МОВНІ ЗАСОБИ СУБД: МОВА СТРУКТУРОВАНИХ ЗАПИТІВ ТА ЇЇ ПІДМОВИ

Мова структурованих запитів SQL (Structured Query Language) є стандартною мовою для роботи з реляційними базами даних. SQL надає уніфікований інтерфейс для визначення структури даних, маніпулювання даними та управління доступом.

Історія SQL починається з 1970-х років, коли компанія IBM розробила мову SEQUEL (Structured English Query Language) для системи System R. Пізніше

назву змінили на SQL. У 1986 році SQL став стандартом ANSI (American National Standards Institute), а в 1987 - ISO (International Organization for Standardization). Стандарт регулярно оновлюється, включаючи нові можливості: SQL-92, SQL:1999, SQL:2003, SQL:2008, SQL:2011, SQL:2016.

Характеристики SQL включають декларативність - користувач описує, які дані потрібно отримати, а не як їх отримати. SQL працює з множинами даних, а не з окремими записами. Мова є непроцедурною, звільняючи програміста від деталей реалізації операцій. SQL максимально наближена до природної мови, що спрощує її вивчення.

Підмови SQL поділяють функціональність на логічні групи команд.

Мова визначення даних (DDL - Data Definition Language) призначена для створення та зміни структури бази даних. Команда CREATE створює нові об'єкти бази даних: таблиці, індекси, представлення, процедури. Команда ALTER змінює структуру існуючих об'єктів, дозволяючи додавати, змінювати або видаляти стовпці таблиць. Команда DROP видаляє об'єкти бази даних назавжди. Команда TRUNCATE видаляє всі дані з таблиці, зберігаючи її структуру.

Приклад створення таблиці:

```
CREATE TABLE Students (
    StudentID INT PRIMARY KEY,
    FirstName VARCHAR(50) NOT NULL,
    LastName VARCHAR(50) NOT NULL,
    BirthDate DATE,
    Email VARCHAR(100) UNIQUE,
    EnrollmentDate DATE DEFAULT CURRENT_DATE
);
```

Мова маніпулювання даними (DML - Data Manipulation Language) забезпечує операції з даними. Команда SELECT вибирає дані з таблиць за заданими критеріями, підтримуючи складні умови, сортування, групування та об'єднання таблиць. Команда INSERT додає нові записи в таблиці. Команда UPDATE змінює значення в існуючих записах. Команда DELETE видаляє записи з таблиць.

Приклад вибірки даних:

```
SELECT FirstName, LastName, Email
FROM Students
WHERE EnrollmentDate > '2023-01-01'
ORDER BY LastName;
```

Приклад вставки даних:

```
INSERT INTO Students (StudentID, FirstName, LastName, BirthDate, Email)
VALUES (1001, 'Іван', 'Петренко', '2002-05-15', 'ivan.petrenko@example.com');
```

Мова керування даними (DCL - Data Control Language) управляє правами доступу до даних. Команда GRANT надає користувачам права на виконання операцій з об'єктами бази даних. Команда REVOKE відкликає надані права. Ці команди забезпечують безпеку бази даних через контроль доступу.

Приклад надання прав:

```
GRANT SELECT, INSERT, UPDATE ON Students TO User1;
```

Мова керування транзакціями (TCL - Transaction Control Language) управляє транзакціями. Команда BEGIN TRANSACTION або START TRANSACTION розпочинає нову транзакцію. Команда COMMIT завершує транзакцію, зберігаючи всі зміни назавжди. Команда ROLLBACK скасовує всі зміни, зроблені в рамках поточної транзакції. Команда SAVEPOINT створює проміжну точку збереження в межах транзакції для часткового відкату.

Приклад використання транзакцій:

```
BEGIN TRANSACTION;
  UPDATE Accounts SET Balance = Balance - 1000 WHERE AccountID = 101;
  UPDATE Accounts SET Balance = Balance + 1000 WHERE AccountID = 102;
COMMIT;
```

Основні компоненти SELECT запиту включають різні клаузули для точного формулювання вибірки. Клаузула SELECT визначає, які стовпці повертати. Клаузула FROM вказує таблиці, з яких вибирати дані. Клаузула WHERE задає умови фільтрації записів. Клаузула GROUP BY групує записи за значеннями стовпців. Клаузула HAVING фільтрує групи після групування. Клаузула ORDER BY визначає порядок сортування результатів. Клаузули LIMIT та OFFSET обмежують кількість повертаємих записів.

Оператори SQL дозволяють будувати складні умови. Оператори порівняння включають =, <>, <, >, <=, >=. Логічні оператори AND, OR, NOT поєднують умови. Оператор BETWEEN перевіряє діапазон значень. Оператор IN перевіряє наявність значення в списку. Оператор LIKE виконує пошук за шаблоном. Оператор IS NULL перевіряє відсутність значення.

Функції SQL надають додаткові можливості обробки даних. Агрегатні функції обчислюють сумарні значення: COUNT підраховує кількість записів, SUM обчислює суму, AVG знаходить середнє значення, MIN та MAX визначають мінімальне та максимальне значення. Строкові функції обробляють текст: CONCAT об'єднує рядки, SUBSTRING витягує частину рядка, UPPER та LOWER змінюють регістр. Функції дати та часу працюють з темпоральними даними: CURRENT_DATE, DATE_ADD, DATEDIFF. Математичні функції виконують обчислення: ROUND, FLOOR, CEILING, ABS.

Об'єднання таблиць (JOIN) дозволяє комбінувати дані з декількох таблиць. INNER JOIN повертає лише записи, що мають відповідність в обох таблицях. LEFT JOIN повертає всі записи з лівої таблиці та відповідні з правої. RIGHT JOIN повертає всі записи з правої таблиці та відповідні з лівої. FULL JOIN повертає всі записи з обох таблиць. CROSS JOIN створює декартовий добуток таблиць.

Приклад об'єднання:

```
SELECT Students.FirstName, Students.LastName, Courses.CourseName
FROM Students
INNER JOIN Enrollments ON Students.StudentID = Enrollments.StudentID
INNER JOIN Courses ON Enrollments.CourseID = Courses.CourseID;
```

Підзапити дозволяють використовувати результати одного запиту в іншому. Скалярні підзапити повертають одне значення. Табличні підзапити повертають набір записів. Корельовані підзапити посилаються на зовнішній запит.

Переваги SQL включають стандартизацію, що забезпечує переносимість між різними СУБД. Простота вивчення завдяки декларативному синтаксису. Потужність для виконання складних операцій однією командою. Оптимізація, оскільки СУБД автоматично вибирає ефективний план виконання. Масштабованість для роботи з великими обсягами даних.

Обмеження SQL включають складність виконання деяких операцій, що природніше реалізувати процедурними мовами. Відмінності в реалізації різних СУБД створюють проблеми переносимості. Продуктивність може погіршуватися для надзвичайно складних запитів.

ТЕМА 2. МОДЕЛІ ДАНИХ. РЕЛЯЦІЙНА МОДЕЛЬ ДАНИХ

1. Ієрархічна та мережна моделі даних. Переваги та недоліки

2. Проблеми маніпулювання даними та обмеження цілісності даних
3. Реляційна модель та її характеристики
4. Структура реляційних даних
5. Домени
6. Схема баз даних
7. Таблиці баз даних
8. Первинні та зовнішні ключі
9. Індекси
10. Методи та способи доступу до даних
11. Цілісність реляційних даних
12. Зв'язки між таблицями
13. Поняття транзакції
14. Механізм транзакцій
15. Нормалізація реляційних баз даних
16. Перша, друга та третя нормальні форми

1. ІЄРАРХІЧНА ТА МЕРЕЖНА МОДЕЛІ ДАНИХ. ПЕРЕВАГИ ТА НЕДОЛІКИ

Перш ніж реляційна модель даних стала домінуючою, у 1960-х та 1970-х роках широко використовувалися ієрархічна та мережна моделі. Розуміння цих моделей, їх переваг та обмежень допомагає оцінити революційність реляційного підходу.

Ієрархічна модель даних організує дані у деревоподібну структуру, де кожен запис має одного батька (за винятком кореневого запису) та може мати багато дочірніх записів. Ця модель нагадує організаційну структуру підприємства або файлову систему комп'ютера.

Основними характеристиками ієрархічної моделі є деревоподібна структура, де дані організовані у вигляді дерева з кореневим вузлом на вершині. Відношення батько-нащадок означає, що кожен дочірній запис має рівно одного батьківського запису. Шлях доступу до даних є єдиним - від кореня вниз до листків. Фізична залежність означає, що логічні зв'язки тісно пов'язані з фізичним зберіганням.

Класичним прикладом ієрархічної СУБД є IMS (Information Management System), розроблена компанією IBM у 1966 році для програми Apollo. Ця система успішно використовувалася десятиліттями, особливо у банківському секторі.

Приклад ієрархічної структури організації підприємства:

Компанія

Мережна модель даних

Мережна модель даних виникла як еволюційний крок у розвитку систем управління базами даних, спрямований на подолання фундаментальних обмежень ієрархічної моделі. Розроблена у 1960-х роках групою CODASYL (Conference on Data Systems Languages), ця модель запропонувала більш гнучкий підхід до організації та представлення даних, дозволяючи моделювати складніші взаємозв'язки між сутностями.

На відміну від жорсткої деревоподібної структури ієрархічної моделі, мережна модель організує дані у вигляді орієнтованого графа. У цій структурі вузли графа представляють записи даних, а ребра відображають зв'язки між ними. Принципова відмінність полягає в тому, що один запис може бути пов'язаний з декількома батьківськими записами, що дозволяє природно моделювати відношення типу "багато-до-багатьох" без необхідності дублювання інформації.

Основним механізмом для представлення зв'язків між типами записів у мережній моделі є концепція "набору" (set). Набір визначає зв'язок типу "один-до-багатьох" між типом запису-власника (owner) та типом запису-члена (member). При цьому один тип запису може бути членом у декількох різних наборах, що і забезпечує можливість моделювання складних мережних структур. Кожен конкретний екземпляр набору складається з одного запису-власника та нуля або більше записів-членів.

Доступ до даних у мережній моделі здійснюється через навігаційний підхід. Програми отримують дані шляхом явної навігації по зв'язках від одного запису до іншого, використовуючи операції типу "знайти власника", "знайти наступного члена набору" тощо. Цей підхід вимагає від програміста детального розуміння структури бази даних та зв'язків між типами записів.

Стандарт CODASYL визначив не лише концептуальну модель даних, але й мову визначення даних (DDL) та мову маніпулювання даними (DML). Ці мови були інтегровані з процедурними мовами програмування, такими як COBOL, що відображало технологічні реалії того часу. Типовими представниками мережних СУБД є системи IDMS (Integrated Database Management System) компанії Cullinet та IDS (Integrated Data Store), розроблена Чарльзом Бахманом.

Приклад мережної структури

Розглянемо базу даних університету для ілюстрації можливостей мережної моделі. У цій предметній області студент може бути зарахований на декілька навчальних курсів, при цьому кожен курс відвідують багато студентів. Крім того, кожен курс викладає певний викладач, а викладач може вести декілька курсів. У мережній моделі це представляється через систему взаємопов'язаних наборів.

Студент "Іван Петренко" може бути одночасно членом набору "Зарахування" для курсу "Бази даних", набору "Зарахування" для курсу "Алгоритми" та набору "Зарахування" для курсу "Операційні системи". Кожен з цих курсів, у свою чергу, є членом відповідного набору "Викладання", де власником виступає запис викладача. Наприклад, курс "Бази даних" може бути пов'язаний з викладачем "Професор Коваленко", який також веде курс "Розподілені системи".

Така структура дозволяє ефективно відповідати на запити різних типів. Можна легко знайти всі курси, на які зарахований конкретний студент, всіх студентів певного курсу, або всі курси, які веде певний викладач. При цьому інформація про студентів, курси та викладачів зберігається без дублювання, що є суттєвою перевагою порівняно з ієрархічною моделлю.

Переваги мережної моделі:

Мережна модель успадкувала багато переваг ієрархічної моделі, зокрема високу продуктивність операцій читання завдяки фізичній близькості пов'язаних даних та заздалегідь визначеним шляхам доступу. Водночас вона усунула одне з найсуттєвіших обмежень ієрархічної моделі, надавши можливість природно моделювати відношення "багато-до-багатьох" без необхідності дублювання даних.

Гнучкість у моделюванні складних зв'язків між сутностями робить мережну модель придатною для широкого спектру предметних областей. Можливість створювати довільні мережні структури дозволяє точніше відображати реальні бізнес-процеси та взаємозв'язки в організаціях. Система наборів забезпечує явне та ефективне представлення асоціацій між різними типами даних.

Мережна модель зберігає переваги щодо забезпечення цілісності даних через механізми підтримки зв'язків на рівні СУБД. Система автоматично контролює коректність посилань між записами та може забезпечувати каскадні операції видалення або оновлення. Це знижує ймовірність виникнення неузгодженостей у базі даних.

Недоліки мережної моделі:

Попри значне покращення можливостей моделювання порівняно з ієрархічною моделлю, мережна модель зберігає багато концептуальних та практичних проблем. Головним недоліком є надзвичайна складність як для розробників, так і для користувачів. Навігаційний підхід до доступу даних вимагає від програмістів детального знання фізичної структури бази даних та всіх зв'язків між типами записів. Створення навіть простих запитів потребує написання складного процедурного коду, що описує покрокову навігацію по мережі записів.

Мережна модель залишається тісно пов'язаною з фізичною організацією даних. Зміни у структурі бази даних, додавання нових наборів або типів записів часто вимагають модифікації існуючих прикладних програм. Це порушує принцип незалежності даних, який є критично важливим для довгострокового супроводу інформаційних систем. Адміністратори не можуть вільно оптимізувати фізичну структуру без ризику порушення працездатності існуючих додатків.

Відсутність стандартної декларативної мови запитів робить роботу з мережними базами даних трудомісткою та схильною до помилок. На відміну від сучасних СУБД, де користувач може просто описати, які дані йому потрібні, мережна модель вимагає явного опису процедури отримання цих даних. Це значно збільшує обсяг програмного коду, ускладнює його розуміння та супровід, а також підвищує ймовірність логічних помилок.

Складність навігаційних програм призводить до високих витрат на розробку та супровід інформаційних систем. Навіть досвідчені програмісти потребують значного часу для розуміння структури великих мережних баз даних. Внесення змін до існуючих систем стає дедалі складнішим у міру зростання бази даних та кількості взаємозв'язків. Це обмежує здатність організацій швидко адаптуватися до нових бізнес-вимог.

Крім того, мережна модель зберігає проблему жорсткості схеми даних. Хоча вона і більш гнучка за ієрархічну модель у представленні зв'язків, структурні зміни все одно залишаються складними та ресурсомісткими.

Реорганізація мережі наборів або додавання нових типів зв'язків може вимагати значних зусиль і призводити до тимчасової недоступності системи.

Аналіз ієрархічної та мережної моделей виявляє фундаментальні проблеми, які властиві навігаційному підходу до організації та маніпулювання даними. Ці проблеми стали каталізатором для розробки принципово нової, реляційної моделі даних.

Однією з центральних проблем є складність маніпулювання даними через процедурний характер операцій. У навігаційних моделях програміст повинен явно описувати не лише те, які дані потрібно отримати, але й детальну процедуру їх отримання. Це вимагає глибокого розуміння внутрішньої структури бази даних, включаючи всі шляхи доступу, індекси та фізичну організацію записів. Така залежність від фізичної реалізації робить програми крихкими та складними у супроводі.

Підтримка цілісності даних у навігаційних моделях покладається переважно на прикладні програми. Хоча СУБД забезпечують базові механізми підтримки зв'язків між записами, складніші обмеження цілісності, такі як бізнес-правила або складні залежності між атрибутами, повинні реалізовуватися у коді кожної програми. Це призводить до дублювання логіки перевірки цілісності у різних додатках та підвищує ризик неузгодженостей.

Проблема забезпечення транзакційної цілісності також стоїть гостро. У багатокористувацькому середовищі необхідно гарантувати, що одночасні операції різних користувачів не призведуть до порушення узгодженості даних. Навігаційні моделі мають обмежені можливості для декларативного визначення транзакційних обмежень, що знову покладає значну відповідальність на розробників прикладних програм.

Відсутність високорівневих засобів опису обмежень цілісності означає, що кожна програма повинна самостійно реалізовувати перевірки коректності даних. Наприклад, обмеження на діапазон значень атрибута, унікальність певних комбінацій атрибутів або референційну цілісність між різними типами записів доводиться програмувати вручну. Це не лише збільшує обсяг коду, але й створює ризик пропуску перевірок або їх неоднакової реалізації у різних частинах системи.

3. РЕЛЯЦІЙНА МОДЕЛЬ ТА ЇЇ ХАРАКТЕРИСТИКИ

Революційний перелом у розвитку систем управління базами даних відбувся у 1970 році, коли Едгар Кодд, співробітник дослідницького центру IBM, опублікував статтю "A Relational Model of Data for Large Shared Data Banks". Ця

робота заклала теоретичні основи реляційної моделі даних, яка згодом стала домінуючою у галузі баз даних і залишається такою дотепер.

Реляційна модель базується на строгому математичному апараті теорії множин та реляційної алгебри. На відміну від навігаційних моделей, які оперують записами та явними зв'язками-покажчиками між ними, реляційна модель представляє всі дані у вигляді відношень (relations), які зручно візуалізувати як таблиці. Це забезпечує високий рівень абстракції та незалежності даних від способів їх фізичного зберігання.

Фундаментальною характеристикою реляційної моделі є повна незалежність логічного представлення даних від їх фізичної організації. Користувачі та прикладні програми взаємодіють з даними на логічному рівні, не турбуючись про те, як саме ці дані зберігаються на диску, які індекси використовуються, або яким чином оптимізуються запити. Адміністратор бази даних може змінювати фізичну структуру для підвищення продуктивності без необхідності модифікації прикладних програм, що є критично важливим для довгострокового супроводу інформаційних систем.

Реляційна модель запроваджує декларативний підхід до маніпулювання даними. Замість того, щоб описувати покрокову процедуру отримання даних, користувач просто вказує, які дані йому потрібні, використовуючи мову запитів високого рівня, такою як SQL (Structured Query Language). Система сама визначає оптимальний спосіб виконання запиту, вибираючи відповідні шляхи доступу, індекси та алгоритми з'єднання таблиць. Це радикально спрощує розробку додатків та робить їх більш стійкими до змін у структурі даних.

Важливою характеристикою реляційної моделі є вбудована підтримка обмежень цілісності на рівні схеми бази даних. Реляційні СУБД дозволяють декларативно визначати різноманітні обмеження, такі як первинні та зовнішні ключі, унікальність значень, обмеження на діапазони значень, складні бізнес-правила. Система автоматично контролює виконання цих обмежень при кожній операції модифікації даних, гарантуючи узгодженість інформації без необхідності дублювання логіки перевірки у кожній прикладній програмі.

Реляційна модель забезпечує високий рівень гнучкості у формулюванні запитів. Оскільки зв'язки між даними визначаються не фізичними покажчиками, а логічними значеннями атрибутів, користувачі можуть формулювати довільні запити, навіть ті, які не були передбачені при проектуванні бази даних. Це різко контрастує з навігаційними моделями, де можливості запитів обмежені заздалегідь визначеними шляхами доступу.

Математична основа реляційної моделі забезпечує можливість формальної оптимізації запитів. Реляційна алгебра та реляційне числення надають строгий апарат для перетворення запитів користувача у еквівалентні, але більш ефективні форми. Оптимізатор запитів може використовувати правила еквівалентних перетворень для пошуку найкращого плану виконання, враховуючи статистику розподілу даних, доступні індекси та інші фактори.

4. СТРУКТУРА РЕЛЯЦІЙНИХ ДАНИХ

Центральним поняттям реляційної моделі є відношення (relation), яке можна розглядати з двох взаємодоповнюючих перспектив: математичної та практичної. З математичної точки зору, відношення є підмножиною декартового добутку доменів. З практичної точки зору, відношення зручно представляти у вигляді таблиці, де кожен рядок відповідає одному кортежу (tuple), а кожен стовпець відповідає атрибуту.

Таблична форма представлення є лише зручною візуалізацією відношення і не слід плутати її з фізичною організацією даних. У реляційній моделі порядок рядків у таблиці є несуттєвим, оскільки відношення є множиною кортежів, а множини за означенням є невпорядкованими. Аналогічно, хоча стовпці зазвичай мають визначений порядок для зручності представлення, концептуально вони ідентифікуються своїми іменами, а не позиціями.

Кожен кортеж у відношенні представляє один факт або один екземпляр сутності предметної області. Наприклад, у таблиці "Студенти" кожен рядок містить інформацію про одного конкретного студента. Важливою властивістю відношення є те, що всі кортежі повинні бути унікальними - не може існувати двох абсолютно ідентичних рядків. Це випливає з визначення відношення як множини, елементи якої не повторюються.

Кожен атрибут відношення має ім'я, яке повинно бути унікальним у межах цього відношення, та асоційований домен, який визначає множину допустимих значень для цього атрибута. Значення атрибута у конкретному кортежі повинно належати відповідному домену або бути спеціальним значенням NULL, яке позначає відсутність або невідомість інформації.

Ступінь відношення (degree) визначається кількістю його атрибутів. Наприклад, відношення "Студент" з атрибутами (StudentID, Прізвище, Ім'я, ДатаНародження, Група) має ступінь п'ять. Потужність відношення (cardinality) визначається кількістю кортежів у ньому. Потужність є динамічною характеристикою, яка змінюється при додаванні або видаленні записів, тоді як ступінь зазвичай залишається сталим протягом життєвого циклу бази даних.

Реляційна модель накладає певні обмеження на структуру відношень, які забезпечують їх математичну коректність та практичну зручність. По-перше, всі значення атрибутів повинні бути атомарними (неподільними). Це означає, що не допускається зберігання складних структур, таких як масиви або вкладені таблиці, у якості значень атрибутів. Ця вимога є основою першої нормальної форми і спрощує операції маніпулювання даними.

По-друге, усі кортежі одного відношення мають однакову структуру - вони визначаються однією й тією ж множиною атрибутів. Не може бути ситуації, коли один рядок таблиці має певний набір стовпців, а інший рядок - інший набір. Це забезпечує однорідність даних та спрощує їх обробку.

По-третє, порядок атрибутів та кортежів не несе семантичного навантаження. Хоча в практичних реалізаціях таблиці зазвичай відображаються з певним порядком рядків та стовпців, логічно цей порядок є несуттєвим. Це дозволяє СУБД вільно реорганізовувати дані для оптимізації продуктивності без впливу на логічний зміст інформації.

5. ДОМЕНИ

Домен у реляційній моделі даних є фундаментальним поняттям, яке визначає множину допустимих значень для одного або декількох атрибутів. Концепція домену є значно ширшою, ніж просто визначення типу даних, оскільки вона також включає семантичне значення та допустимі операції над значеннями.

З формальної точки зору, домен є іменованою множиною скалярних значень одного типу. Наприклад, домен "Вік_людини" може бути визначений як множина цілих чисел від 0 до 150. Хоча технічно цей домен має той самий базовий тип даних (ціле число), що й домен "Кількість_товару", семантично вони є різними, і операції порівняння між атрибутами цих різних доменів можуть не мати змісту.

Визначення домену включає декілька компонентів. Базовий тип даних специфікує фундаментальний тип значень, такий як ціле число, дійсне число, рядок символів, дата тощо. Формат представлення визначає, як значення відображаються та вводяться користувачем, наприклад, формат дати може бути "ДД.ММ.РРРР" або "РРРР-ММ-ДД". Діапазон допустимих значень накладає обмеження на можливі значення атрибута, як-от мінімум та максимум для числових типів або максимальна довжина для рядків.

Семантичне значення домену є критично важливим для правильного розуміння та використання даних. Два атрибути можуть мати однаковий базовий

тип даних та діапазон значень, але різні семантичні значення. Наприклад, атрибут "Ціна_покупки" та атрибут "Ціна_продажу" обидва можуть бути визначені як десяткові числа з двома знаками після коми, але їх семантика є різною, і операція порівняння між ними має специфічне бізнес-значення.

Домени відіграють важливу роль у забезпеченні цілісності даних. Визначаючи домен для атрибута, ми створюємо перший рівень обмежень, які СУБД автоматично контролює при введенні або модифікації даних. Якщо користувач намагається ввести значення, яке не належить відповідному домену, система відхиляє таку операцію, запобігаючи внесенню некоректних даних.

Використання доменів також сприяє узгодженості схеми бази даних. Якщо декілька атрибутів у різних таблицях представляють однотипну інформацію, вони повинні бути визначені на одному домені. Наприклад, атрибут "Код_студента" у таблиці "Студенти" та атрибут "Код_студента" у таблиці "Оцінки" повинні використовувати один і той самий домен. Це гарантує, що зв'язки між таблицями будуть коректними та що операції порівняння матимуть семантичний зміст.

У практичних реляційних СУБД підтримка доменів реалізована з різним ступенем повноти. Деякі системи дозволяють створювати іменовані домени з усіма обмеженнями та правилами перевірки, які автоматично застосовуються до всіх атрибутів цього домену. Інші системи обмежуються визначенням типів даних та обмежень безпосередньо для кожного атрибута, що є менш елегантним, але функціонально еквівалентним підходом.

6. СХЕМА БАЗ ДАНИХ

Схема бази даних є формальним описом структури бази даних, який визначає організацію даних, відношення між ними, обмеження цілісності та інші метадані. Схема представляє собою статичний опис бази даних, який змінюється рідко, на відміну від екземпляра бази даних - фактичних даних, які динамічно змінюються в процесі експлуатації системи.

У реляційній моделі схема бази даних складається з множини схем відношень. Кожна схема відношення визначає структуру однієї таблиці, включаючи її ім'я, множину атрибутів з їх доменами, первинний ключ, зовнішні ключі та інші обмеження цілісності. Наприклад, схема відношення "Студент" може бути записана як: Студент(Код_студента: INTEGER, Прізвище: VARCHAR(50), Ім'я: VARCHAR(50), Група: VARCHAR(10), PRIMARY KEY(Код_студента)).

Схема бази даних також включає визначення зв'язків між відношеннями через механізм зовнішніх ключів. Ці зв'язки визначають референційну цілісність - правила, які гарантують, що посилання з одного відношення на записи в іншому відношенні завжди вказують на існуючі записи. Наприклад, якщо таблиця "Оцінки" містить атрибут "Код_студента", який посилається на таблицю "Студенти", схема повинна явно визначати цей зв'язок та правила підтримки цілісності.

Розрізняють три рівні представлення схеми бази даних відповідно до трирівневої архітектури ANSI/SPARC. Концептуальна схема описує логічну структуру всієї бази даних незалежно від її фізичної реалізації. Вона відображає інформаційні потреби організації та зв'язки між різними типами даних. Це той рівень, на якому проектувальники баз даних визначають таблиці, їх атрибути та зв'язки.

Зовнішні схеми (також звані підсхемами або представленнями) визначають різні погляди на дані для різних груп користувачів або додатків. Кожна зовнішня схема є підмножиною концептуальної схеми, яка включає лише ті частини бази даних, які є релевантними для конкретної групи користувачів. Наприклад, бухгалтерія може мати доступ до фінансової інформації, але не до персональних даних працівників, тоді як відділ кадрів має протилежний набір доступів.

Внутрішня схема описує фізичну організацію даних на носіях інформації, включаючи структури зберігання, індекси, методи доступу та інші деталі реалізації. Цей рівень є прозорим для більшості користувачів та прикладних програм, що забезпечує незалежність даних. Адміністратор бази даних може оптимізувати внутрішню схему для підвищення продуктивності без впливу на концептуальну та зовнішні схеми.

Визначення схеми бази даних здійснюється за допомогою мови визначення даних (Data Definition Language, DDL), яка є частиною SQL. DDL надає оператори для створення, модифікації та видалення об'єктів бази даних, таких як таблиці, індекси, представлення та обмеження цілісності. Наприклад, оператор CREATE TABLE визначає нову таблицю, ALTER TABLE модифікує існуючу структуру, а DROP TABLE видаляє таблицю.

Схема бази даних зберігається у системному каталозі (data dictionary або system catalog), який сам є базою даних спеціальних системних таблиць. Системний каталог містить метадані про всі об'єкти бази даних, включаючи інформацію про таблиці, стовпці, індекси, обмеження, користувачів та їх привілеї. СУБД використовує системний каталог для перевірки коректності запитів, контролю прав доступу та оптимізації виконання операцій.

7. ТАБЛИЦІ БАЗ ДАНИХ

Таблиця є основною структурою зберігання даних у реляційній базі даних та практичною реалізацією концепції відношення. Кожна таблиця представляє певний тип сутності предметної області, такої як студенти, курси, замовлення, товари тощо, і складається з рядків та стовпців.

Структурні компоненти таблиці

Стовпці (колонки) представляють атрибути сутності та визначають, які характеристики описуються для кожного екземпляра. Кожен стовпець має:

- Унікальне ім'я в межах таблиці.
- Тип даних, який визначає характер значень.
- Необов'язкові обмеження (наприклад, NOT NULL, UNIQUE).
- Домен допустимих значень.

Рядки (записи) представляють окремі екземпляри сутності. Кожен рядок містить конкретні значення для всіх атрибутів, визначених стовпцями таблиці. Наприклад, у таблиці "Студенти" кожен рядок містить інформацію про одного конкретного студента.

Властивості реляційних таблиць

Реляційні таблиці мають специфічні властивості, які відрізняють їх від звичайних електронних таблиць:

1. **Атомарність значень** - кожна комірка таблиці містить лише одне неподільне значення. Не допускається зберігання списків, масивів або складних структур у одній комірці.

2. **Унікальність рядків** - у таблиці не може бути двох абсолютно ідентичних рядків. Ця властивість забезпечується визначенням первинного ключа.

3. **Неупорядкованість рядків** - логічно рядки таблиці не мають визначеного порядку, хоча фізично вони можуть зберігатися в певній послідовності для оптимізації.

4. **Однорідність стовпців** - всі значення в одному стовпці мають один і той самий тип даних та домен.

Приклад структури таблиці

Розглянемо таблицю "Студенти" з наступною структурою:

Таблиця: СТУДЕНТИ

Код_студента (INTEGER, PRIMARY KEY)

Прізвище (VARCHAR(50), NOT NULL)

Ім'я (VARCHAR(50), NOT NULL)

По_батькові (VARCHAR(50))
Дата_народження (DATE)
Група (VARCHAR(10), NOT NULL)
Електронна_пошта (VARCHAR(100), UNIQUE)
Телефон (VARCHAR(15))

Ця таблиця описує студентів освітнього закладу. Атрибут "Код_студента" є первинним ключем, що гарантує унікальну ідентифікацію кожного студента. Атрибути "Прізвище", "Ім'я" та "Група" визначені як NOT NULL, оскільки ця інформація є обов'язковою. Електронна пошта має обмеження UNIQUE, що запобігає реєстрації двох студентів з однаковою адресою.

Операції над таблицями

З таблицями можна виконувати різноманітні операції маніпулювання даними:

Вставка даних (INSERT) - додавання нових рядків до таблиці. При цьому СУБД автоматично перевіряє всі визначені обмеження цілісності.

Оновлення даних (UPDATE) - модифікація існуючих значень у таблиці. Можна оновлювати один або декілька рядків одночасно, а також один або декілька атрибутів.

Видалення даних (DELETE) - вилучення рядків з таблиці. СУБД перевіряє референційну цілісність перед видаленням, щоб запобігти видаленню записів, на які посилаються інші таблиці.

Вибірка даних (SELECT) - отримання даних з таблиці згідно з заданими критеріями. Це найпоширеніша операція, яка може включати фільтрацію, сортування, групування та об'єднання даних з декількох таблиць.

Метадані таблиць

Для кожної таблиці СУБД зберігає метадані, які включають:

- Назву таблиці та схеми, до якої вона належить.
- Список усіх стовпців з їх типами даних.
- Визначення первинного ключа.
- Список зовнішніх ключів та посилань.
- Індекси, створені для таблиці.
- Обмеження цілісності.
- Права доступу для різних користувачів.
- Статистичну інформацію про розподіл даних.

8. ПЕРВИННІ ТА ЗОВНІШНІ КЛЮЧІ

Ключі є фундаментальним механізмом у реляційних базах даних, який забезпечує унікальну ідентифікацію записів та встановлення зв'язків між таблицями.

Первинний ключ (Primary Key)

Первинний ключ - це атрибут або мінімальна комбінація атрибутів, яка однозначно ідентифікує кожен рядок у таблиці. Первинний ключ є основним засобом забезпечення унікальності записів.

Властивості первинного ключа:

1. **Унікальність** - значення первинного ключа повинно бути унікальним для кожного рядка таблиці. Не може існувати двох рядків з однаковим значенням первинного ключа.

2. **Обов'язковість (NOT NULL)** - первинний ключ не може містити NULL значення, оскільки NULL означає невідомість, а невідоме значення не може однозначно ідентифікувати запис.

3. **Незмінність** - бажано, щоб значення первинного ключа не змінювалося протягом життєвого циклу запису, оскільки зміна може порушити посилання з інших таблиць.

4. **Мінімальність** - якщо первинний ключ складається з декількох атрибутів (складений ключ), жоден з цих атрибутів не може бути вилучений без втрати властивості унікальності.

Типи первинних ключів:

– **Простий ключ** складається з одного атрибута. Наприклад, код студента, ідентифікаційний номер товару.

– **Складений (композитний) ключ** складається з двох або більше атрибутів. Наприклад, комбінація (Код_студента, Код_курсу) може бути первинним ключем у таблиці, яка реєструє зарахування студентів на курси.

Приклад визначення первинного ключа:

```
CREATE TABLE Студенти (
    Код_студента INTEGER PRIMARY KEY,
    Прізвище VARCHAR(50) NOT NULL,
    Ім'я VARCHAR(50) NOT NULL,
    Група VARCHAR(10)
);
```

або для складеного ключа:

```
CREATE TABLE Зарахування (
    Код_студента INTEGER,
    Код_курсу INTEGER,
```

```
    Дата_зарахування DATE,
    PRIMARY KEY (Код_студента, Код_курсу)
);
```

Потенційні ключі та альтернативні ключі

Таблиця може мати декілька комбінацій атрибутів, які задовольняють вимогам унікальності. Такі комбінації називаються **потенційними ключами** (candidate keys). З множини потенційних ключів один обирається як первинний, а решта стають **альтернативними ключами**.

Наприклад, у таблиці "Студенти" потенційними ключами можуть бути:

- Код_студента
- Номер_студентського_квитка
- Електронна_пошта (якщо вона обов'язкова та унікальна)

Якщо "Код_студента" обрано як первинний ключ, то "Номер_студентського_квитка" та "Електронна_пошта" стають альтернативними ключами, для яких також потрібно забезпечити унікальність через обмеження UNIQUE.

Зовнішній ключ (Foreign Key)

Зовнішній ключ - це атрибут або комбінація атрибутів в одній таблиці, значення яких посилаються на первинний ключ іншої (або тієї ж самої) таблиці. Зовнішні ключі є основним механізмом встановлення зв'язків між таблицями у реляційній моделі.

Призначення зовнішніх ключів:

1. **Встановлення зв'язків** між таблицями для моделювання відношень між сутностями предметної області.

2. **Забезпечення референційної цілісності** - гарантування, що посилання з однієї таблиці завжди вказують на існуючі записи в іншій таблиці.

3. **Реалізація бізнес-правил** через каскадні операції оновлення та видалення.

Властивості зовнішнього ключа:

- Значення зовнішнього ключа повинно або збігатися зі значенням первинного ключа у зв'язаній таблиці, або бути NULL (якщо дозволено).
- Зовнішній ключ може посилатися на первинний ключ у тій самій таблиці (рекурсивний зв'язок).
- Декілька зовнішніх ключів можуть посилатися на одну й ту саму батьківську таблицю.

Приклад визначення зовнішнього ключа:

CREATE TABLE Оцінки (

```
    Код_оцінки INTEGER PRIMARY KEY,
    Код_студента INTEGER NOT NULL,
    Код_курсу INTEGER NOT NULL,
    Оцінка INTEGER CHECK (Оцінка BETWEEN 0 AND 100),
    Дата_виставлення DATE,
    FOREIGN KEY (Код_студента) REFERENCES Студенти(Код_студента),
    FOREIGN KEY (Код_курсу) REFERENCES Курси(Код_курсу)
);
```

У цьому прикладі таблиця "Оцінки" має два зовнішніх ключі: один посилається на таблицю "Студенти", інший - на таблицю "Курси".

Правила підтримки референційної цілісності

При визначенні зовнішнього ключа можна вказати правила, що визначають поведінку системи при спробі видалення або оновлення запису в батьківській таблиці:

ON DELETE CASCADE - при видаленні запису в батьківській таблиці автоматично видаляються всі пов'язані записи в дочірній таблиці.

ON DELETE SET NULL - при видаленні запису в батьківській таблиці значення зовнішнього ключа в дочірній таблиці встановлюється в NULL.

ON DELETE RESTRICT або **NO ACTION** - забороняє видалення запису в батьківській таблиці, якщо існують пов'язані записи в дочірній таблиці.

ON UPDATE CASCADE - при зміні значення первинного ключа в батьківській таблиці автоматично оновлюються відповідні значення зовнішнього ключа в дочірній таблиці.

Приклад з правилами підтримки цілісності:

```
CREATE TABLE Оцінки (
    Код_оцінки INTEGER PRIMARY KEY,
    Код_студента INTEGER NOT NULL,
    Код_курсу INTEGER NOT NULL,
    Оцінка INTEGER,
    FOREIGN KEY (Код_студента)
        REFERENCES Студенти(Код_студента)
        ON DELETE CASCADE
        ON UPDATE CASCADE,
    FOREIGN KEY (Код_курсу)
        REFERENCES Курси(Код_курсу)
        ON DELETE RESTRICT
```

);

9. ІНДЕКСИ

Індекси є критично важливим механізмом оптимізації продуктивності реляційних баз даних. Вони створюють додаткові структури даних, які дозволяють швидко знаходити записи без необхідності сканування всієї таблиці.

Концепція індексу

Індекс - це допоміжна структура даних, яка містить відсортовані значення одного або декількох стовпців таблиці разом з покажчиками на відповідні рядки. Індекс працює за принципом, подібним до алфавітного покажчика в кінці книги, який дозволяє швидко знайти потрібну сторінку без перегляду всього тексту.

Без індексу СУБД змушена виконувати **повне сканування таблиці** (full table scan) - послідовний перегляд кожного рядка для знаходження потрібних даних. Для великих таблиць це може бути надзвичайно повільною операцією. Індекс дозволяє значно скоротити час пошуку, особливо для великих обсягів даних.

Типи індексів

1. Первинний індекс (Primary Index)

Автоматично створюється для первинного ключа таблиці. Він забезпечує унікальність значень та швидкий доступ до записів за значенням первинного ключа. У багатьох СУБД дані фізично організовані відповідно до первинного індексу (кластеризований індекс).

2. Унікальний індекс (Unique Index)

Гарантує унікальність значень у індексованих стовпцях. Використовується для реалізації альтернативних ключів та забезпечення бізнес-правил, які вимагають унікальності певних комбінацій атрибутів.

```
CREATE UNIQUE INDEX idx_email ON Студенти(Електронна_пошта);
```

3. Неунікальний індекс (Non-unique Index)

Найпоширеніший тип індексу, який створюється для прискорення пошуку за стовпцями, що часто використовуються в умовах WHERE, JOIN або ORDER BY. Допускає дублювання значень.

```
CREATE INDEX idx_прізвище ON Студенти(Прізвище);
```

4. Композитний (складений) індекс

Будується на основі декількох стовпців. Ефективний для запитів, які фільтрують або сортують за комбінацією цих стовпців. Порядок стовпців в індексі є важливим.

CREATE INDEX idx_група_прізвище ON Студенти(Група, Прізвище);

Такий індекс буде ефективним для запитів типу:

- WHERE Група = 'ІПЗ-21' AND Прізвище = 'Іваненко'
- WHERE Група = 'ІПЗ-21' (використовується лівий префікс індексу)

Але менш ефективним для:

- WHERE Прізвище = 'Іваненко' (не використовує лівий префікс)

5. Кластеризований індекс (Clustered Index)

Визначає фізичний порядок зберігання даних у таблиці. Таблиця може мати лише один кластеризований індекс, оскільки дані можуть бути фізично впорядковані лише одним способом. Зазвичай кластеризованим є індекс первинного ключа.

6. Некластеризований індекс (Non-clustered Index)

Зберігає відсортовані значення індексованих стовпців окремо від фактичних даних таблиці. Таблиця може мати багато некластеризованих індексів.

Структури даних для індексів

B-дерево (B-tree) - найпоширеніша структура для індексів у реляційних СУБД. B-дерево є збалансованим деревом пошуку, яке забезпечує ефективний пошук, вставку та видалення за логарифмічний час O(log n). Ця структура особливо ефективна для операцій порівняння (=, <, >, BETWEEN) та пошуку за діапазоном.

Хеш-індекс - використовує хеш-функцію для швидкого доступу до даних. Ефективний для операцій точної відповідності (=), але не підтримує операції порівняння або пошук за діапазоном.

Bitmap-індекс - використовується для стовпців з невеликою кількістю унікальних значень (низька кардинальність). Ефективний для складних запитів з багатьма умовами AND/OR.

Переваги використання індексів

1. **Прискорення пошуку даних** - драматичне зменшення часу виконання SELECT запитів, особливо для великих таблиць.

2. **Оптимізація сортування** - якщо запит вимагає сортування (ORDER BY) за індексованими стовпцями, СУБД може використати індекс замість виконання окремої операції сортування.

3. **Прискорення операцій JOIN** - індекси на стовпцях, які використовуються для з'єднання таблиць, значно покращують продуктивність.

4. **Забезпечення унікальності** - унікальні індекси гарантують, що не буде дублювання значень у відповідних стовпцях.

Недоліки та витрати на індекси

Попри очевидні переваги, індекси мають певні недоліки:

1. **Додаткове дискове простір** - кожен індекс займає місце на диску, іноді порівнянне з розміром самої таблиці.

2. **Уповільнення операцій модифікації** - при INSERT, UPDATE або DELETE операціях необхідно оновлювати не лише саму таблицю, але й всі її індекси, що збільшує час виконання цих операцій.

3. **Витрати на супровід** - СУБД повинна підтримувати індекси в актуальному стані, що створює додаткове навантаження.

4. **Складність вибору оптимальних індексів** - надмірна кількість індексів може призвести до зниження продуктивності операцій модифікації без значного покращення швидкості запитів.

Стратегії створення індексів

Індексуйте стовпці, які часто використовуються в:

– Умовах WHERE для фільтрації даних

– Операціях JOIN для з'єднання таблиць

– Сортуванні ORDER BY

– Групуванні GROUP BY

Уникайте надмірного індексування:

– Не створюйте індекси на стовпцях, які рідко використовуються в запитах

– Уникайте індексів на стовпцях з низькою селективністю (багато дублювання значень)

– Обмежте кількість індексів на таблицях з частими операціями модифікації

Регулярно аналізуйте використання індексів:

– Моніторте, які індекси фактично використовуються запитами

– Видаляйте невикористовувані індекси

– Перебудовуйте фрагментовані індекси для підтримки оптимальної продуктивності

10. МЕТОДИ ТА СПОСОБИ ДОСТУПУ ДО ДАНИХ

Ефективний доступ до даних є критично важливим для продуктивності системи управління базами даних. СУБД використовує різноманітні методи та способи доступу для оптимізації операцій читання та модифікації даних.

Основні методи доступу

1. Послідовне сканування (Sequential Scan / Full Table Scan)

Це найпростіший метод доступу, при якому СУБД послідовно читає кожен рядок таблиці від початку до кінця. Цей метод використовується, коли:

– у таблиці немає відповідних індексів;

– потрібно отримати більшість рядків таблиці;

– вартість використання індексу перевищує вартість повного сканування.

Переваги: простота реалізації, ефективність при читанні великої частини таблиці.

Недоліки: неефективність для великих таблиць, коли потрібна невелика підмножина рядків.

2. Доступ за індексом (Index Scan)

СУБД використовує індекс для швидкого знаходження потрібних рядків. Процес включає:

– пошук у структурі індексу (зазвичай B-дереві);

– отримання покажчиків на рядки таблиці;

– читання фактичних даних з таблиці.

Цей метод ефективний, коли запит вибирає невелику частину рядків таблиці.

3. Пошук за індексом (Index Seek)

Прямий пошук конкретного значення або діапазону значень в індексі без сканування всього індексу. Це найбільш ефективний метод для точкових запитів.

4. Кластеризований індексний скан

Якщо таблиця має кластеризований індекс, дані вже впорядковані фізично, що робить послідовне читання більш ефективним для запитів з сортуванням або діапазонних запитів.

Способи з'єднання таблиць (JOIN методи)

При виконанні операцій JOIN між таблицями СУБД може використовувати різні алгоритми:

1. Nested Loop Join (Вкладений цикл)

Для кожного рядка з першої таблиці (зовнішньої) система переглядає всі відповідні рядки другої таблиці (внутрішньої). Це найпростіший алгоритм, ефективний коли:

– одна з таблиць невелика;

– на внутрішній таблиці є індекс за стовпцем з'єднання.

Складність: $O(n \times m)$, де n та m - кількість рядків у таблицях.

2. Hash Join (Хеш-з'єднання)

СУБД створює хеш-таблицю для однієї з таблиць, потім для кожного рядка другої таблиці виконує пошук у хеш-таблиці. Ефективний для великих таблиць без відповідних індексів та для операцій рівності (equi-joins).

3. Merge Join (З'єднання злиттям)

Обидві таблиці спочатку сортуються за стовпцями з'єднання, потім виконується одночасний послідовний обхід обох таблиць. Дуже ефективний для великих відсортованих наборів даних або таблиць з відповідними індексами.

Оптимізація запитів

Оптимізатор запитів - це компонент СУБД, який аналізує SQL запит та вибирає найбільш ефективний план виконання. Процес оптимізації включає:

1. Парсинг та перевірка синтаксису Запит аналізується на предмет синтаксичних помилок та перевіряється коректність посилань на об'єкти бази даних.

2. Генерація можливих планів виконання Оптимізатор генерує множину можливих способів виконання запиту, враховуючи доступні індекси, різні методи доступу та порядок з'єднання таблиць.

3. Оцінка вартості кожного плану Для кожного плану розраховується оцінка вартості на основі:

– статистики розподілу даних у таблицях;

– кількості рядків у таблицях (кардинальність);

– селективності умов WHERE;

– доступності індексів;

– вартості операцій вводу-виводу.

4. Вибір оптимального плану Обирається план з найменшою оцінкою вартості, який буде використаний для виконання запиту.

Кешування та буферизація

Буферний пул (Buffer Pool) - це область оперативної пам'яті, де СУБД зберігає часто використовувані сторінки даних та індексів. Це значно зменшує кількість операцій вводу-виводу, оскільки читання з пам'яті на порядки швидше, ніж з диска.

Механізми управління буферним пулом включають:

– алгоритми заміщення сторінок (LRU - Least Recently Used);

– попереднє завантаження даних (prefetching);

– відкладений запис змінених сторінок на диск.

Кеш планів виконання зберігає скомпільовані плани запитів для повторного використання, що економить час на оптимізацію при повторному виконанні подібних запитів.

11. ЦІЛІСНІСТЬ РЕЛЯЦІЙНИХ ДАНИХ

Цілісність даних є фундаментальною вимогою до будь-якої системи управління базами даних. Вона гарантує точність, узгодженість та надійність інформації, що зберігається в базі даних.

Типи обмежень цілісності

1. Доменна цілісність (Domain Integrity)

Забезпечує коректність значень окремих атрибутів через обмеження на домени. Включає:

– **Типи даних** - визначають базовий формат значень (INTEGER, VARCHAR, DATE тощо).

– **NOT NULL** - забороняє NULL значення для обов'язкових атрибутів.

– **DEFAULT** - визначає значення за замовчуванням.

– **CHECK** - задає умови, яким повинні відповідати значення.

Приклад:

```
CREATE TABLE Оцінки (
    Код_оцінки INTEGER PRIMARY KEY,
    Оцінка INTEGER CHECK (Оцінка BETWEEN 0 AND 100),
    Дата DATE NOT NULL DEFAULT CURRENT_DATE,
    Статус VARCHAR(20) CHECK (Статус IN ('Зараховано', 'Перескладання', 'Незараховано'))
);
```

2. Сутнісна цілісність (Entity Integrity)

Гарантує, що кожен рядок у таблиці може бути однозначно ідентифікований. Основні правила:

– Кожна таблиця повинна мати первинний ключ.

– Значення первинного ключа повинно бути унікальним для кожного рядка.

– Первинний ключ не може містити NULL значення.

Це забезпечує, що кожна сутність в базі даних є чітко ідентифікованою та доступною для посилань з інших таблиць.

3. Референційна цілісність (Referential Integrity)

Забезпечує коректність зв'язків між таблицями через механізм зовнішніх ключів. Основні правила:

1. Значення зовнішнього ключа повинно або збігатися зі значенням первинного ключа в батьківській таблиці, або бути NULL (якщо дозволено).

2. Не можна вставити рядок з значенням зовнішнього ключа, яке не існує в батьківській таблиці.

3. Не можна видалити рядок з батьківської таблиці, якщо на нього посилаються рядки в дочірній таблиці (без відповідних правил каскадування).

Приклад:

```
CREATE TABLE Оцінки (
   Код_оцінки INTEGER PRIMARY KEY,
   Код_студента INTEGER NOT NULL,
   Код_курсу INTEGER NOT NULL,
   Оцінка INTEGER,
   FOREIGN KEY (Код_студента) REFERENCES
Студенти(Код_студента) ON DELETE RESTRICT ON UPDATE CASCADE, FOREIGN KEY (Код_курсу) REFERENCES Курси(Код_курсу) ON DELETE CASCADE );
```

4. Користувацька (бізнес-логічна) цілісність.

Включає специфічні правила та обмеження, що відображають бізнес-логіку предметної області. Це можуть бути:

- Складні перевірки, що включають декілька атрибутів або таблиць
- Тригери, що автоматично виконуються при певних подіях
- Збережені процедури, що інкапсулюють бізнес-правила

Приклад: студент не може бути зарахований на більше ніж 8 курсів одночасно, або дата завершення проекту не може бути раніше дати початку.

Механізми підтримки цілісності

Декларативні обмеження

Визначаються безпосередньо в схемі бази даних через DDL оператори. СУБД автоматично перевіряє ці обмеження при кожній операції модифікації даних:

```sql
CREATE TABLE Курси (
   Код_курсу INTEGER PRIMARY KEY,
   Назва VARCHAR(100) NOT NULL,
   Кредити INTEGER CHECK (Кредити > 0 AND Кредити <= 10),
   Семестр INTEGER CHECK (Семестр BETWEEN 1 AND 8),
   UNIQUE (Назва, Семестр)
);
```

Тригери (Triggers)

Це спеціальні збережені процедури, які автоматично виконуються у відповідь на певні події в базі даних (INSERT, UPDATE, DELETE). Тригери використовуються для реалізації складної бізнес-логіки, яку неможливо виразити через прості обмеження:

```
CREATE TRIGGER перевірка_максимум_курсів
BEFORE INSERT ON Зарахування
FOR EACH ROW
BEGIN
   DECLARE кількість INTEGER;
   SELECT COUNT(*) INTO кількість
   FROM Зарахування
   WHERE Код_студента = NEW.Код_студента;

   IF кількість >= 8 THEN
      SIGNAL SQLSTATE '45000'
      SET MESSAGE_TEXT = 'Студент не може бути зарахований більше ніж на 8 курсів';
   END IF;
END;
```

Збережені процедури

Дозволяють інкапсулювати складну логіку перевірки та модифікації даних на стороні сервера, забезпечуючи централізоване управління бізнес-правилами.

Переваги централізованої підтримки цілісності

1. **Консистентність** - правила цілісності визначаються один раз у схемі БД і автоматично застосовуються для всіх операцій.

2. **Зменшення дублювання коду** - не потрібно реалізовувати перевірки в кожній прикладній програмі.

3. **Підвищення надійності** - СУБД гарантує перевірку всіх обмежень незалежно від того, яка програма звертається до даних.

4. **Спрощення розробки** - розробники можуть покладатися на те, що СУБД автоматично підтримує коректність даних.

5. **Полегшення супроводу** - зміни в бізнес-правилах вносяться у одному місці (схема БД), а не у багатьох програмах.

12. ЗВ'ЯЗКИ МІЖ ТАБЛИЦЯМИ

Зв'язки між таблицями є фундаментальним механізмом моделювання відношень між сутностями у реляційних базах даних. Правильне визначення

зв'язків є критично важливим для створення ефективної та логічно коректної структури бази даних.

Типи зв'язків

1. Зв'язок "один-до-одного" (One-to-One, 1:1)

У цьому типі зв'язку один запис в таблиці А пов'язаний максимум з одним записом у таблиці В, і навпаки. Такі зв'язки зустрічаються рідше за інші типи.

Приклади використання:

– Студент - Студентський квиток (кожен студент має один квиток, кожен квиток належить одному студенту).

– Користувач - Профіль користувача (розділення базової та додаткової інформації).

– Працівник - Деталі заробітної плати (конфіденційна інформація в окремій таблиці).

Реалізація:

```
CREATE TABLE Студенти (
    Код_студента INTEGER PRIMARY KEY,
    Прізвище VARCHAR(50),
    Ім'я VARCHAR(50)
);
CREATE TABLE Студентські_квитки (
    Номер_квитка VARCHAR(20) PRIMARY KEY,
    Код_студента INTEGER UNIQUE NOT NULL,
    Дата_видачі DATE,
    Дата_закінчення DATE,
    FOREIGN KEY (Код_студента) REFERENCES Студенти(Код_студента)
);
```

Ключовою особливістю є обмеження UNIQUE на зовнішньому ключі, що гарантує унікальність зв'язку.

2. Зв'язок "один-до-багатьох" (One-to-Many, 1:N)

Це найпоширеніший тип зв'язку, де один запис у батьківській таблиці може бути пов'язаний з багатьма записами в дочірній таблиці, але кожен запис у дочірній таблиці пов'язаний лише з одним записом у батьківській таблиці.

Приклади:

– Факультет - Студенти (один факультет має багато студентів, кожен студент належить одному факультету).

– Викладач - Курси (один викладач веде багато курсів, кожен курс веде один викладач).

– Замовлення - Рядки замовлення (одне замовлення містить багато позицій).

Реалізація:

```
CREATE TABLE Факультети (
    Код_факультету INTEGER PRIMARY KEY,
    Назва VARCHAR(100),
    Декан VARCHAR(100)
);
CREATE TABLE Студенти (
    Код_студента INTEGER PRIMARY KEY,
    Прізвище VARCHAR(50),
    Ім'я VARCHAR(50),
    Код_факультету INTEGER NOT NULL,
    FOREIGN KEY (Код_факультету) REFERENCES Факультети(Код_факультету)
);
```

Зовнішній ключ розміщується в таблиці "багато" (Студенти) і посилається на первинний ключ таблиці "один" (Факультети).

3. Зв'язок "багато-до-багатьох" (Many-to-Many, M:N)

У цьому типі зв'язку багато записів у таблиці А можуть бути пов'язані з багатьма записами у таблиці В, і навпаки. Реляційна модель не підтримує прямої реалізації таких зв'язків, тому використовується проміжна таблиця (таблиця зв'язків, асоціативна таблиця).

Приклади:

– Студенти - Курси (студент відвідує багато курсів, курс відвідують багато студентів).

– Автори - Книги (автор пише багато книг, книга може мати багато авторів).

– Актори - Фільми (актор знімається у багатьох фільмах, у фільмі знімаються багато акторів).

Реалізація через проміжну таблицю:

```
CREATE TABLE Студенти (
    Код_студента INTEGER PRIMARY KEY,
    Прізвище VARCHAR(50),
    Ім'я VARCHAR(50)
```

```
);

CREATE TABLE Курси (
    Код_курсу INTEGER PRIMARY KEY,
    Назва VARCHAR(100),
    Кредити INTEGER
);

CREATE TABLE Зарахування (
    Код_студента INTEGER,
    Код_курсу INTEGER,
    Семестр INTEGER,
    Рік INTEGER,
    Оцінка INTEGER,
    PRIMARY KEY (Код_студента, Код_курсу),
    FOREIGN KEY (Код_студента) REFERENCES Студенти(Код_студента),
    FOREIGN KEY (Код_курсу) REFERENCES Курси(Код_курсу)
);
```

Проміжна таблиця "Зарахування" містить зовнішні ключі до обох таблиць, і її первинний ключ зазвичай є композитним, що складається з обох зовнішніх ключів. Додатково ця таблиця може містити атрибути, що характеризують сам зв'язок (наприклад, оцінка, дата зарахування).

Кардинальність та модальність зв'язків

Кардинальність (cardinality) визначає максимальну кількість екземплярів однієї сутності, які можуть бути пов'язані з екземпляром іншої сутності. Це те, що ми обговорювали як "один" або "багато" у типах зв'язків.

Модальність (modality або optionality) визначає мінімальну кількість екземплярів, тобто чи є участь у зв'язку обов'язковою чи необов'язковою:

- **Обов'язкова участь** (NOT NULL) - кожен запис повинен бути пов'язаний
- **Необов'язкова участь** (NULL дозволено) - запис може існувати без зв'язку

Приклад:

```
-- Студент ПОВИНЕН належати факультету (обов'язкова участь)
CREATE TABLE Студенти (
    Код_студента INTEGER PRIMARY KEY,
    Прізвище VARCHAR(50),
```

```
    Код_факультету INTEGER NOT NULL,
    FOREIGN KEY (Код_факультету) REFERENCES Факультети(Код_факультету)
);

-- Студент може мати або не мати наукового керівника (необов'язкова участь)
CREATE TABLE Студенти (
    Код_студента INTEGER PRIMARY KEY,
    Прізвище VARCHAR(50),
    Код_керівника INTEGER NULL,
    FOREIGN KEY (Код_керівника) REFERENCES Викладачі(Код_викладача)
);
```

Рекурсивні зв'язки

Рекурсивний зв'язок виникає, коли таблиця пов'язана сама з собою. Це корисно для моделювання ієрархічних структур.

Приклади:

– Працівники та їх керівники (керівник також є працівником)

– Категорії товарів та підкатегорії

– Організаційна структура (підрозділ може мати батьківський підрозділ)

Реалізація:

```
CREATE TABLE Працівники (
    Код_працівника INTEGER PRIMARY KEY,
    Прізвище VARCHAR(50),
    Ім'я VARCHAR(50),
    Посада VARCHAR(50),
    Код_керівника INTEGER NULL,
    FOREIGN KEY (Код_керівника) REFERENCES Працівники(Код_працівника)
);
```

У цьому прикладі атрибут "Код_керівника" посилається на первинний ключ тієї ж таблиці, дозволяючи представити ієрархію керівників та підлеглих.

Проектування зв'язків - практичні рекомендації

1. **Визначайте природні зв'язки** - зв'язки повинні відображати реальні відношення в предметній області, а не бути штучними конструкціями.

2. **Використовуйте змістовні імена** - імена зовнішніх ключів повинні чітко вказувати на зв'язок (наприклад, "Код_факультету" замість просто "Факультет_ID").

3. **Завжди визначайте правила підтримки цілісності** - явно вказуйте ON DELETE та ON UPDATE правила залежно від бізнес-логіки.

4. **Документуйте семантику зв'язків** - пояснюйте значення кожного зв'язку, особливо для складних структур даних.

5. **Уникайте зайвих зв'язків** - не створюйте зв'язків, які можна вивести з існуючих (транзитивні зв'язки).

13. ПОНЯТТЯ ТРАНЗАКЦІЇ

Транзакція є одним з найважливіших понять у теорії баз даних і критично важливою для забезпечення надійності та узгодженості даних у багатокористувацьких системах.

Визначення транзакції

Транзакція - це логічна одиниця роботи, що складається з однієї або більше операцій з базою даних, які виконуються як єдине ціле. Транзакція переводить базу даних з одного узгодженого стану в інший узгоджений стан.

Ключова характеристика транзакції полягає в тому, що всі операції в її складі повинні бути виконані повністю або не виконані взагалі. Не допускається часткове виконання транзакції, коли деякі операції виконалися, а інші - ні.

Простий приклад: Розглянемо банківську операцію переказу грошей з рахунку А на рахунок Б:

1. Зняти 1000 грн з рахунку А
2. Додати 1000 грн на рахунок Б

Ці дві операції повинні виконуватися як транзакція. Якщо перша операція виконається, а друга - ні (наприклад, через збій системи), гроші "зникнуть", що є неприпустимим. Транзакція гарантує, що або обидві операції виконаються успішно, або база даних залишиться в початковому стані.

Властивості ACID

Транзакції характеризуються чотирма фундаментальними властивостями, відомими як ACID:

1. Атомарність (Atomicity)

Транзакція є неподільною одиницею роботи. Всі операції в транзакції або виконуються повністю, або не виконуються взагалі. Якщо будь-яка операція в транзакції не може бути завершена, всі попередньо виконані операції повинні бути скасовані (відкочені).

Атомарність забезпечує принцип "все або нічого". У випадку збою системи, помилки виконання або явного скасування транзакції база даних повертається до стану, який був до початку транзакції.

2. Узгодженість (Consistency)

Транзакція переводить базу даних з одного узгодженого стану в інший узгоджений стан. Це означає, що після завершення транзакції всі обмеження цілісності, визначені в базі даних, повинні залишатися виконаними.

Якщо виконання транзакції призведе до порушення будь-якого обмеження цілісності, транзакція має бути скасована. Наприклад, якщо визначено обмеження на додатні залишки на рахунках, транзакція, яка призведе до від'ємного залишку, не може бути завершена.

3. Ізольованість (Isolation)

Одночасне виконання декількох транзакцій не повинно призводити до неузгодженості даних. Кожна транзакція має виконуватися так, ніби вона є єдиною в системі, навіть якщо насправді багато транзакцій виконуються одночасно.

Ізольованість запобігає різним аномаліям, що можуть виникнути при паралельному виконанні транзакцій, таким як:

- **Брудне читання** (dirty read) - читання незафіксованих змін іншої транзакції.
- **Неповторюване читання** (non-repeatable read) - отримання різних значень при повторному читанні тих самих даних у межах однієї транзакції.
- **Фантомне читання** (phantom read) - поява або зникнення рядків при повторному виконанні запиту.

4. Довговічність (Durability)

Після успішного завершення (фіксації) транзакції її зміни є постійними і зберігаються навіть у разі збою системи. Зафіксовані дані записуються на постійний носій інформації і не можуть бути втрачені.

СУБД використовує різні механізми для забезпечення довговічності, включаючи журналювання транзакцій, контрольні точки та резервне копіювання.

Оператори управління транзакціями

SQL надає спеціальні оператори для управління транзакціями:

BEGIN TRANSACTION (або START TRANSACTION) - позначає початок нової транзакції.

COMMIT - успішно завершує транзакцію, роблячи всі зміни постійними.

ROLLBACK - скасовує транзакцію, відміняючи всі зміни, зроблені з моменту початку транзакції.

SAVEPOINT - створює точку збереження в транзакції, до якої можна відкотити зміни без скасування всієї транзакції.

Приклад використання:

```
BEGIN TRANSACTION;

UPDATE Рахунки
SET Баланс = Баланс - 1000
WHERE Номер_рахунку = 'A123';

UPDATE Рахунки
SET Баланс = Баланс + 1000
WHERE Номер_рахунку = 'B456';

-- Перевірка коректності балансів
IF (SELECT MIN(Баланс) FROM Рахунки WHERE Номер_рахунку IN ('A123', 'B456')) < 0
THEN
    ROLLBACK; -- Скасувати транзакцію
ELSE
    COMMIT; -- Зафіксувати зміни
END IF;
```

Рівні ізоляції транзакцій

SQL стандарт визначає чотири рівні ізоляції транзакцій, які балансують між узгодженістю даних та продуктивністю:

1. READ UNCOMMITTED (Читання незафіксованих даних) Найнижчий рівень ізоляції. Транзакція може читати незафіксовані зміни інших транзакцій. Дозволяє всі типи аномалій, але забезпечує максимальну паралельність.

2. READ COMMITTED (Читання зафіксованих даних) Транзакція може читати лише зафіксовані дані. Запобігає брудному читанню, але допускає неповторюване та фантомне читання.

3. REPEATABLE READ (Повторюване читання) Гарантує, що якщо транзакція прочитала значення, воно залишиться незмінним протягом життя транзакції. Запобігає брудному та неповторюваному читанню, але допускає фантомне читання.

4. SERIALIZABLE (Серіалізоване виконання) Найвищий рівень ізоляції. Забезпечує повну ізоляцію, ніби транзакції виконуються послідовно одна за одною. Запобігає всім аномаліям, але може знижувати продуктивність через блокування ресурсів.

Важливість транзакцій у практиці

Транзакції є критично важливими для:

– **Фінансових систем** - забезпечують коректність грошових операцій.

– **Систем бронювання** - запобігають подвійному бронюванню.

– **Системи управління запасами** - гарантують точність обліку товарів.

– **Будь-яких систем**, де потрібна гарантія узгодженості даних при складних операціях, що включають багато кроків.

Розуміння концепції транзакцій та правильне їх використання є необхідною компетенцією для розробників систем управління базами даних та додатків, що працюють з критичними даними.

14. МЕХАНІЗМ ТРАНЗАКЦІЙ

Транзакція є фундаментальним поняттям у системах управління базами даних, що забезпечує надійність та узгодженість даних навіть у разі збоїв системи або одночасного доступу багатьох користувачів. Механізм транзакцій становить основу для побудови надійних інформаційних систем, що функціонують у реальних умовах з множинними користувачами та можливими проблемами.

Поняття транзакції визначається як логічна одиниця роботи з базою даних, що складається з однієї або більше операцій, які повинні виконуватися як єдине ціле. Транзакція переводить базу даних з одного узгодженого стану в інший узгоджений стан. Критично важливим є те, що всі операції транзакції повинні виконатися успішно, або жодна з них не повинна вплинути на базу даних.

Класичним прикладом транзакції є операція переказу грошей між банківськими рахунками. Ця операція складається з двох кроків: зняття певної суми з одного рахунку та зарахування цієї суми на інший рахунок. Якщо перша операція виконається успішно, але друга з якихось причин не відбудеться, виникне неприпустима ситуація - гроші зникнуть. Транзакція гарантує, що або виконаються обидві операції, або жодна з них не змінить стан бази даних.

Властивості ACID визначають фундаментальні характеристики транзакцій, що забезпечують надійність роботи з даними. Абревіатура ACID

походить від англійських термінів Atomicity, Consistency, Isolation, Durability, кожен з яких описує критично важливу властивість.

Атомарність (Atomicity) означає неподільність транзакції як єдиної операції. Транзакція виконується повністю і успішно, переводячи базу даних з одного узгодженого стану в інший, або не виконується взагалі, залишаючи базу даних в початковому стані. Не існує проміжних станів - транзакція не може бути виконана частково. Якщо на будь-якому етапі виконання транзакції відбувається помилка або збій системи, всі зміни, що вже були зроблені, повинні бути скасовані. Цей процес називається відкатом транзакції (rollback).

Механізм забезпечення атомарності базується на веденні журналу транзакцій, де реєструються всі операції зміни даних. Перед внесенням змін у базу даних система записує в журнал інформацію про старі значення даних. У разі необхідності відкату ця інформація використовується для повернення бази даних до стану, що існував до початку транзакції. Журнал транзакцій являє собою послідовний файл, куди записи додаються в хронологічному порядку, що забезпечує високу швидкість запису навіть при інтенсивному навантаженні.

Узгодженість (Consistency) гарантує, що транзакція переводить базу даних з одного узгодженого стану в інший узгоджений стан, зберігаючи всі визначені правила цілісності. Узгоджений стан означає, що всі дані відповідають встановленим обмеженням, бізнес-правилам та взаємозв'язкам. Якщо транзакція порушує будь-яке правило цілісності, вона має бути відкинута, а всі її зміни - скасовані.

Правила цілісності включають обмеження домену, що визначають допустимі значення для атрибутів, обмеження сутності, що забезпечують унікальність первинних ключів, обмеження посилальної цілісності, що контролюють зв'язки між таблицями, та семантичні обмеження, що реалізують специфічні бізнес-правила. СУБД автоматично перевіряє дотримання всіх цих обмежень під час виконання транзакції.

Відповідальність за забезпечення узгодженості розподіляється між СУБД та розробником додатка. СУБД автоматично перевіряє формальні обмеження цілісності, такі як типи даних, унікальність ключів, посилальна цілісність. Розробник додатка повинен забезпечити, щоб логіка транзакції відповідала бізнес-правилам та семантичній коректності операцій.

Ізольованість (Isolation) означає, що одночасне виконання транзакцій дає такий самий результат, як їх послідовне виконання. Кожна транзакція виконується так, ніби вона єдина в системі, не спостерігаючи проміжних станів

інших транзакцій. Ізольованість захищає транзакції від взаємного впливу та запобігає виникненню аномалій при одночасному доступі до даних.

Проблеми одночасного доступу виникають, коли декілька транзакцій намагаються працювати з одними й тими ж даними. Проблема втраченого оновлення (Lost Update) виникає, коли дві транзакції читають одне й те саме значення, змінюють його та записують назад, внаслідок чого оновлення однієї транзакції втрачається. Проблема брудного читання (Dirty Read) виникає, коли транзакція читає дані, змінені іншою транзакцією, яка потім відкочується, і прочитані дані виявляються некоректними. Проблема неповторюваного читання (Non-repeatable Read) виникає, коли транзакція двічі читає одні й ті ж дані, але між читаннями інша транзакція змінює ці дані. Проблема фантомного читання (Phantom Read) виникає, коли транзакція двічі виконує запит з однаковими умовами, але між запитами інша транзакція додає або видаляє записи, що задовольняють умовам.

Рівні ізольованості визначають, які з цих проблем допускаються для досягнення балансу між узгодженістю даних та продуктивністю системи. Стандарт SQL визначає чотири рівні ізольованості транзакцій.

Read Uncommitted (читання незавершених даних) є найнижчим рівнем ізольованості, де транзакція може читати дані, змінені іншими незавершеними транзакціями. Цей рівень дозволяє всі три типи аномалій: брудне читання, неповторюване читання та фантомне читання. Використовується рідко, лише для операцій, де точність даних не критична, але потрібна максимальна продуктивність.

Read Committed (читання завершених даних) дозволяє транзакції читати лише дані, що були збережені завершеними транзакціями. Цей рівень запобігає брудному читанню, але дозволяє неповторюване та фантомне читання. Є типовим рівнем за замовчуванням у багатьох СУБД, включаючи Oracle та SQL Server.

Repeatable Read (повторюване читання) гарантує, що якщо транзакція прочитала деякі дані, повторне читання цих самих даних поверне ті ж значення, навіть якщо інші транзакції намагаються їх змінити. Цей рівень запобігає брудному та неповторюваному читанню, але дозволяє фантомне читання. Використовується MySQL за замовчуванням на рівні InnoDB.

Serializable (серіалізований) є найвищим рівнем ізольованості, що гарантує повну ізоляцію транзакцій. Транзакції виконуються так, ніби вони йдуть послідовно одна за одною. Цей рівень запобігає всім трьом типам

аномалій, але може значно знизити продуктивність через необхідність більш жорстких блокувань.

Механізми забезпечення ізольованості включають різні методи контролю одночасного доступу. Блокування є основним механізмом, де транзакція захоплює ресурси перед їх використанням. Спільне блокування (Shared Lock) дозволяє декільком транзакціям одночасно читати дані, але не змінювати їх. Монопольне блокування (Exclusive Lock) надає транзакції ексклюзивний доступ до даних для зміни, блокуючи інші транзакції від читання та запису.

Багатоверсійний контроль одночасності (MVCC - Multiversion Concurrency Control) є альтернативним підходом, де система підтримує декілька версій кожного запису. Кожна транзакція бачить знімок даних на момент свого початку, що дозволяє транзакціям читання працювати без блокувань. Цей підхід використовується в PostgreSQL, Oracle та інших сучасних СУБД для підвищення продуктивності.

Довговічність (Durability) гарантує, що після успішного завершення транзакції її результати збережуться в базі даних назавжди, навіть у разі збою системи. Зміни, зроблені завершеною транзакцією, не можуть бути втрачені. Ця властивість критично важлива для надійності системи, особливо у фінансових та інших критичних додатках.

Механізми забезпечення довговічності базуються на використанні журналу транзакцій з технікою випереджувального запису в журнал (Write-Ahead Logging, WAL). Перед тим як зміни фізично записуються в файли бази даних, інформація про ці зміни записується в журнал транзакцій. Журнал зберігається на надійному носії, часто з дублюванням. Після запису в журнал транзакція може бути зафіксована, і користувач отримує підтвердження успішного завершення. Фактичний запис змін у файли бази даних може відбутися пізніше.

У разі збою системи процес відновлення використовує журнал транзакцій для повернення бази даних до узгодженого стану. Транзакції, що були завершені, але зміни яких не встигли записатися на диск, повторно застосовуються з журналу (операція redo). Транзакції, що не були завершені на момент збою, відкочуються (операція undo).

Команди управління транзакціями дозволяють програмісту явно контролювати межі транзакцій. Команда BEGIN TRANSACTION або START TRANSACTION позначає початок нової транзакції. З цього моменту всі операції з базою даних виконуються в контексті цієї транзакції. Команда COMMIT

завершує транзакцію успішно, зберігаючи всі зміни назавжди. Після виконання COMMIT зміни стають видимими іншим транзакціям та не можуть бути скасовані. Команда ROLLBACK скасовує транзакцію, відкочуючи всі зміни та повертаючи базу даних до стану, що існував до початку транзакції. ROLLBACK може виконуватися явно програмою або автоматично системою при виникненні помилок.

Команда SAVEPOINT створює іменовану точку збереження всередині транзакції, до якої можна виконати частковий відкат без скасування всієї транзакції. Це корисно для довгих транзакцій, де потрібно скасувати лише частину операцій. Команда ROLLBACK TO SAVEPOINT відкочує транзакцію до вказаної точки збереження, скасовуючи операції, виконані після неї, але зберігаючи попередні зміни.

Режими завершення транзакцій визначають, як СУБД обробляє транзакції. У режимі автоматичного завершення (Autocommit) кожна окрема SQL-команда розглядається як окрема транзакція і автоматично фіксується після виконання. Цей режим зручний для інтерактивної роботи, але не підходить для операцій, що вимагають атомарності декількох команд. У режимі явних транзакцій програміст явно позначає початок та кінець кожної транзакції за допомогою команд BEGIN, COMMIT та ROLLBACK. Цей режим надає повний контроль над межами транзакцій.

Точки блокування (Deadlock) виникають, коли дві або більше транзакцій взаємно блокують одна одну, очікуючи ресурси, захоплені іншими транзакціями. Класичний приклад: транзакція A захопила ресурс X і чекає на ресурс Y, а транзакція B захопила ресурс Y і чекає на ресурс X. Обидві транзакції чекатимуть нескінченно, якщо система не втрутиться.

СУБД використовують різні стратегії для запобігання та виявлення deadlock. Виявлення deadlock здійснюється через періодичну перевірку графа очікування транзакцій. При виявленні циклу в графі система вибирає жертву - одну з транзакцій для відкату. Критерії вибору можуть включати вік транзакції, кількість виконаної роботи, пріоритет. Запобігання deadlock реалізується через впорядкування захоплення ресурсів або використання таймаутів очікування.

Розподілені транзакції охоплюють операції на декількох незалежних базах даних або серверах. Забезпечення властивостей ACID для розподілених транзакцій значно складніше, ніж для локальних. Використовується протокол двофазного завершення (Two-Phase Commit, 2PC), де координатор транзакції керує процесом фіксації на всіх вузлах.

На першій фазі (фаза підготовки) координатор запитує кожен вузол, чи готовий він зафіксувати транзакцію. Кожен вузол виконує всі операції, записує зміни в журнал, але не фіксує остаточно, відповідаючи "готовий" або "не готовий". Якщо всі вузли відповіли "готовий", координатор починає другу фазу. На другій фазі (фаза фіксації) координатор відправляє команду COMMIT всім вузлам, якщо всі були готові, або команду ROLLBACK, якщо хоча б один вузол відмовився. Кожен вузол виконує відповідну команду та підтверджує її виконання.

Оптимізація продуктивності транзакцій є критично важливою для високонавантажених систем. Тривалість транзакцій має бути мінімальною - чим довше виконується транзакція, тим більше ресурсів вона блокує та тим вища ймовірність конфліктів. Блокування має бути настільки жорстким, наскільки необхідно для забезпечення коректності, але не більше. Використання відповідних рівнів ізольованості дозволяє знайти баланс між узгодженістю та продуктивністю. Пакетна обробка операцій в одній транзакції замість багатьох маленьких транзакцій зменшує накладні витрати на управління транзакціями.

15. НОРМАЛІЗАЦІЯ РЕЛЯЦІЙНИХ БАЗ ДАНИХ

Нормалізація є систематичним процесом організації даних у реляційній базі даних з метою зменшення надмірності інформації та запобігання аномаліям при виконанні операцій вставки, оновлення та видалення. Процес нормалізації розкладає складні таблиці на простіші за визначеними правилами, створюючи структуру, що забезпечує цілісність та ефективність роботи з даними.

Цілі нормалізації включають усунення надмірності даних, що означає зберігання кожного факту лише один раз, що економить місце та спрощує підтримку узгодженості. Запобігання аномаліям оновлення досягається через правильну структуру, яка не дозволяє виникнення неузгодженостей при зміні даних. Забезпечення логічної структури даних робить базу даних зрозумілою та простою для використання. Спрощення підтримки цілісності даних досягається через чітке визначення залежностей між атрибутами.

Проблеми ненормалізованих даних можна проілюструвати на прикладі погано спроектованої таблиці, що містить інформацію про замовлення:

```
Таблиця: Замовлення
ЗамовленняID | КлієнтID | ІмяКлієнта | АдресаКлієнта | ТелефонКлієнта | ПродуктID | НазваПродукту | ЦінаПродукту | Кількість
```

```
1 | 101 | Іван Петренко | вул. Шевченка, 10 | 0501234567 | 201 | Ноутбук | 15000 | 2
1 | 101 | Іван Петренко | вул. Шевченка, 10 | 0501234567 | 202 | Миша | 300 | 1
2 | 102 | Марія Коваль | пр. Перемоги, 25 | 0509876543 | 201 | Ноутбук | 15000 | 1
```

У цій таблиці виникають численні проблеми. **Надмірність даних** проявляється в тому, що інформація про клієнта (ім'я, адреса, телефон) повторюється для кожного продукту в замовленні. Інформація про продукт повторюється в кожному замовленні, що його містить. Ця надмірність призводить до марнування місця та ускладнює підтримку даних.

Аномалія вставки виникає, коли неможливо додати інформацію без наявності інших даних. Наприклад, неможливо додати інформацію про нового клієнта, якщо він ще не зробив жодного замовлення, оскільки немає значень для атрибутів, пов'язаних із замовленням. Неможливо додати інформацію про новий продукт без прив'язки до конкретного замовлення.

Аномалія оновлення виникає, коли зміна одного факту вимагає оновлення багатьох записів. Якщо клієнт змінює адресу, потрібно оновити всі записи його замовлень. Пропуск хоча б одного запису призводить до неузгодженості даних. Якщо змінюється ціна продукту, потрібно оновити всі замовлення, що містять цей продукт.

Аномалія видалення виникає, коли видалення інформації призводить до непередбаченої втрати інших даних. Якщо видалити останнє замовлення клієнта, втрачається вся інформація про цього клієнта. Якщо видалити всі замовлення, що містять певний продукт, втрачається інформація про існування цього продукту.

Функціональна залежність є фундаментальним поняттям теорії нормалізації. Атрибут B функціонально залежить від атрибута A, якщо кожному значенню A відповідає рівно одне значення B. Це записується як A → B, що читається "A визначає B" або "B функціонально залежить від A".

У наведеному прикладі існують такі функціональні залежності: КлієнтID → ІмяКлієнта, АдресаКлієнта, ТелефонКлієнта (ідентифікатор клієнта визначає всю інформацію про клієнта), ПродуктID → НазваПродукту, ЦінаПродукту (ідентифікатор продукту визначає його назву та ціну), (ЗамовленняID, ПродуктID) → Кількість (комбінація замовлення та продукту визначає кількість цього продукту в замовленні).

Процес нормалізації здійснюється поетапно через послідовне приведення таблиць до різних нормальних форм. Кожна наступна нормальна форма є більш жорсткою і усуває певний тип проблем. Традиційно виділяють шість нормальних форм, але на практиці найчастіше використовуються перші три.

Ненормалізована форма (UNF - Unnormalized Form) являє собою таблицю, що може містити повторювані групи атрибутів або багатозначні атрибути. Це відправна точка процесу нормалізації, де дані можуть бути організовані довільним чином без дотримання будь-яких правил.

16. ПЕРША, ДРУГА ТА ТРЕТЯ НОРМАЛЬНІ ФОРМИ

Процес нормалізації здійснюється послідовно через приведення таблиць до нормальних форм зростаючої строгості. Кожна нормальна форма усуває певні типи залежностей та аномалій, створюючи все більш оптимальну структуру бази даних.

Перша нормальна форма (1NF - First Normal Form) є базовою вимогою для реляційних таблиць. Таблиця знаходиться в першій нормальній формі, якщо всі її атрибути містять лише атомарні (неподільні) значення, кожен атрибут містить значення лише одного типу, кожен атрибут має унікальну назву, порядок рядків не має значення, кожен рядок унікально ідентифікується первинним ключем, немає повторюваних груп атрибутів.

Розглянемо приклад таблиці, що порушує 1NF:

```
Таблиця: Студенти_Курси (порушення 1NF)
СтудентID | ІмяСтудента | Курси
1 | Іван Петренко | Математика, Фізика, Хімія
2 | Марія Коваль | Література, Історія
3 | Петро Сидоренко | Математика, Інформатика
```

Ця таблиця порушує 1NF, оскільки атрибут "Курси" містить множинні значення (список курсів), що є неатомарним значенням. Для приведення до 1NF необхідно розкласти кожне множинне значення на окремі рядки:

```
Таблиця: Студенти_Курси (відповідає 1NF)
СтудентID | ІмяСтудента | Курс
1 | Іван Петренко | Математика
1 | Іван Петренко | Фізика
1 | Іван Петренко | Хімія
2 | Марія Коваль | Література
2 | Марія Коваль | Історія
3 | Петро Сидоренко | Математика
3 | Петро Сидоренко | Інформатика
```

Тепер кожен атрибут містить атомарне значення, і таблиця відповідає 1NF. Однак виникає нова проблема - надмірність інформації про студентів, що призводить до аномалій оновлення.

Інший приклад порушення 1NF - використання повторюваних груп атрибутів:

```
Таблиця: Замовлення (порушення 1NF)
ЗамовленняID | КлієнтID | Продукт1 | Кількість1 | Продукт2 | Кількість2 | Продукт3 | Кількість3
1 | 101 | Ноутбук | 2 | Миша | 1 | NULL | NULL
2 | 102 | Клавіатура | 1 | NULL | NULL | NULL | NULL
```

Ця структура неефективна та обмежує кількість продуктів у замовленні. Для приведення до 1NF створюється окремий рядок для кожного продукту в замовленні, аналогічно попередньому прикладу.

Друга нормальна форма (2NF - Second Normal Form) усуває часткові залежності від ключа. Таблиця знаходиться в другій нормальній формі, якщо вона відповідає 1NF та кожен неключовий атрибут повністю функціонально залежить від всього первинного ключа, а не від його частини. Ця вимога застосовується лише до таблиць із складеним (багатоатрибутним) первинним ключем.

Розглянемо таблицю, що відповідає 1NF, але порушує 2NF:

```
Таблиця: Замовлення_Деталі (відповідає 1NF, порушує 2NF)
ЗамовленняID | ПродуктID | НазваПродукту | ЦінаПродукту | Кількість | ДатаЗамовлення | КлієнтID
1 | 201 | Ноутбук | 15000 | 2 | 2024-01-15 | 101
1 | 202 | Миша | 300 | 1 | 2024-01-15 | 101
2 | 201 | Ноутбук | 15000 | 1 | 2024-01-16 | 102
3 | 203 | Клавіатура | 500 | 3 | 2024-01-17 | 101
```

Первинний ключ цієї таблиці складається з (ЗамовленняID, ПродуктID). Проаналізуємо функціональні залежності:

- (ЗамовленняID, ПродуктID) → Кількість (кількість залежить від обох компонентів ключа - повна залежність)
- ПродуктID → НазваПродукту, ЦінаПродукту (назва та ціна залежать лише від продукту - часткова залежність)
- ЗамовленняID → ДатаЗамовлення, КлієнтID (дата та клієнт залежать лише від замовлення - часткова залежність)

Часткові залежності призводять до проблем: інформація про продукт дублюється в кожному замовленні, зміна ціни продукту вимагає оновлення

багатьох рядків, інформація про замовлення дублюється для кожного продукту в ньому.

Для приведення до 2NF необхідно розкласти таблицю на три таблиці, кожна з яких містить атрибути, що повністю залежать від свого ключа:

```
Таблиця: Замовлення (відповідає 2NF)
ЗамовленняID | ДатаЗамовлення | КлієнтID
1 | 2024-01-15 | 101
2 | 2024-01-16 | 102
3 | 2024-01-17 | 101
Таблиця: Продукти (відповідає 2NF)
ПродуктID | НазваПродукту | ЦінаП
```

ТЕМА 3. МОВИ ЗАПИТІВ ДО РЕЛЯЦІЙНИХ БАЗ ДАНИХ

1. Основні поняття мови SQL.
2. Запити на читання даних.
3. Склеювання таблиць.
4. Умови відбору рядків таблиць.
5. Агрегатні функції.
6. Запити з групуванням.
7. Складні запити.
8. Запити на оновлення даних.

1. ОСНОВНІ ПОНЯТТЯ МОВИ SQL.

Реляційні бази даних стали фундаментом сучасних інформаційних систем завдяки їхній здатності ефективно зберігати та обробляти структуровані дані. Однак сама наявність даних не має практичної цінності без можливості їх витягування, аналізу та модифікації. Саме для цього було розроблено спеціалізовані мови запитів, серед яких мова SQL (Structured Query Language) посідає домінуюче становище як індустріальний стандарт.

SQL виник у 1970-х роках у дослідницьких лабораторіях IBM на основі реляційної моделі даних, запропонованої Едгаром Коддом. З того часу ця мова пройшла шлях від експериментального інструменту до універсального засобу роботи з даними, який підтримується практично всіма сучасними системами управління базами даних. Розуміння принципів роботи SQL є необхідною компетенцією для фахівців у галузі інформаційних технологій, аналітики даних та розробки програмного забезпечення.

SQL являє собою декларативну мову програмування, що відрізняється від процедурних мов своїм підходом до виконання завдань. Замість того, щоб описувати покроковий алгоритм обробки даних, користувач формулює запит, який описує бажаний результат, а система управління базою даних самостійно визначає оптимальний спосіб його отримання.

Мова SQL традиційно поділяється на кілька підмов, кожна з яких відповідає за певний аспект роботи з базою даних. DDL (Data Definition Language) призначена для визначення структури даних і включає оператори CREATE, ALTER та DROP, які дозволяють створювати, модифікувати та видаляти об'єкти бази даних. DML (Data Manipulation Language) охоплює операції маніпулювання даними через оператори SELECT, INSERT, UPDATE та DELETE. DCL (Data Control Language) керує правами доступу за допомогою операторів GRANT та REVOKE. Існує також TCL (Transaction Control Language) для управління транзакціями з операторами COMMIT, ROLLBACK та SAVEPOINT.

Базовою одиницею зберігання даних у реляційній моделі виступає таблиця, яка складається з рядків та стовпців. Кожен стовпець має визначений тип даних, що забезпечує цілісність інформації. Типи даних у SQL включають числові типи (INTEGER, DECIMAL, FLOAT), символьні типи (CHAR, VARCHAR, TEXT), типи дати та часу (DATE, TIME, TIMESTAMP), а також спеціалізовані типи для роботи з бінарними даними, JSON-структурами та іншими форматами.

Ключовим концептом реляційних баз даних є первинний ключ – атрибут або набір атрибутів, які однозначно ідентифікують кожен рядок таблиці. Зовнішні ключі забезпечують зв'язки між таблицями, посилаючись на первинні ключі інших таблиць і підтримуючи референційну цілісність даних. Це дозволяє уникнути надмірності інформації та забезпечує нормалізацію бази даних.

2. ЗАПИТИ НА ЧИТАННЯ ДАНИХ

Оператор SELECT є найбільш часто використовуваним у SQL і призначений для вилучення інформації з бази даних. У своїй найпростішій формі він дозволяє вибрати всі стовпці з таблиці або лише певні з них. Синтаксис базового запиту включає ключове слово SELECT, після якого перераховуються потрібні стовпці або символ зірочки для вибору всіх стовпців, та ключове слово FROM з вказанням імені таблиці.

Наприклад, запит для вибору всіх даних із таблиці студентів виглядатиме так: SELECT * FROM Students. Якщо потрібно отримати лише імена та прізвища

студентів, запит набуває вигляду: SELECT FirstName, LastName FROM Students. Можливість вибірки конкретних стовпців є важливою для оптимізації продуктивності, оскільки зменшує обсяг переданих даних.

SQL надає потужні механізми для трансформації даних безпосередньо в запиті. Можна створювати обчислювані поля, застосовуючи арифметичні операції до числових стовпців. Наприклад, запит SELECT ProductName, Price, Price * 1.2 AS PriceWithTax FROM Products обчислює ціну з податком для кожного товару. Ключове слово AS дозволяє надавати псевдоніми стовпцям, що покращує читабельність результатів.

Для видалення дублікатів із результуючої вибірки використовується ключове слово DISTINCT. Запит SELECT DISTINCT City FROM Customers поверне список унікальних міст, у яких проживають клієнти, без повторень. Це особливо корисно при аналізі категоріальних даних та побудові довідників.

Сортування результатів здійснюється за допомогою конструкції ORDER BY, яка може сортувати дані за одним або кількома стовпцями у зростаючому (ASC) або спадному (DESC) порядку. Запит SELECT * FROM Employees ORDER BY LastName ASC, FirstName ASC впорядковує співробітників спочатку за прізвищем, а потім за іменем. Сортування відбувається після вибірки даних і може істотно впливати на час виконання запиту при великих обсягах інформації.

3. СКЛЕЮВАННЯ ТАБЛИЦЬ

Однією з найпотужніших можливостей реляційних баз даних є здатність поєднувати дані з кількох таблиць в одному запиті. Цей процес називається з'єднанням або склеюванням таблиць і реалізується через оператор JOIN. Розуміння різних типів з'єднань є критичним для ефективної роботи з нормалізованими базами даних.

Внутрішнє з'єднання (INNER JOIN) є найпоширенішим типом і повертає лише ті рядки, для яких знайдено відповідність в обох таблицях згідно з умовою з'єднання. Розглянемо приклад бази даних, яка містить таблиці Students (StudentID, Name) та Enrollments (EnrollmentID, StudentID, CourseID). Запит SELECT Students.Name, Enrollments.CourseID FROM Students INNER JOIN Enrollments ON Students.StudentID = Enrollments.StudentID поверне список студентів разом з курсами, на які вони записані, виключаючи студентів без жодного запису.

Ліве зовнішнє з'єднання (LEFT JOIN або LEFT OUTER JOIN) включає всі рядки з лівої таблиці навіть якщо для них немає відповідності в правій таблиці. У такому випадку для стовпців правої таблиці встановлюються значення NULL.

Це корисно, наприклад, для отримання списку всіх студентів, включаючи тих, хто ще не записався на жоден курс.

Праве зовнішнє з'єднання (RIGHT JOIN) працює аналогічно лівому, але зберігає всі рядки з правої таблиці. Повне зовнішнє з'єднання (FULL OUTER JOIN) комбінує результати лівого та правого з'єднання, включаючи всі рядки з обох таблиць і заповнюючи NULL там, де немає відповідності.

Перехресне з'єднання (CROSS JOIN) створює декартів добуток двох таблиць, поєднуючи кожен рядок першої таблиці з кожним рядком другої. Хоча цей тип з'єднання рідко використовується безпосередньо, він може бути корисним для генерації комбінацій даних або тестування.

Самоз'єднання (self-join) – це техніка, коли таблиця з'єднується сама з собою. Це використовується, наприклад, для знаходження пар співробітників з одного відділу або для роботи з ієрархічними структурами, де рядки посилаються на інші рядки тієї ж таблиці через зовнішній ключ.

4. УМОВИ ВІДБОРУ РЯДКІВ ТАБЛИЦЬ

Конструкція WHERE є фундаментальним механізмом фільтрації даних у SQL, дозволяючи вибирати лише ті рядки, які відповідають заданим критеріям. Умови можуть бути простими або складними, використовуючи різноманітні оператори порівняння та логічні з'єднання.

Базові оператори порівняння включають знак рівності для перевірки точної відповідності, нерівності для виключення значень, а також оператори більше, менше та їхні комбінації з рівністю. Запит SELECT * FROM Products WHERE Price > 100 вибере всі товари, ціна яких перевищує 100 грошових одиниць. Для роботи з текстовими даними особливо корисним є оператор LIKE, який підтримує шаблони з використанням спеціальних символів: відсоток замінює будь-яку послідовність символів, а підкреслення – один символ.

Оператор BETWEEN спрощує перевірку належності значення до діапазону. Замість запису Price >= 50 AND Price <= 150 можна написати компактніше: Price BETWEEN 50 AND 150. Оператор IN дозволяє перевіряти наявність значення в заданому списку, наприклад, City IN ('Київ', 'Львів', 'Одеса') вибере записи для трьох зазначених міст.

Логічні оператори AND, OR та NOT надають гнучкість у формулюванні складних умов. Оператор AND вимагає виконання всіх перелічених умов одночасно, OR задовольняється виконанням принаймні однієї з умов, а NOT інвертує логічне значення умови. Важливо враховувати пріоритет виконання логічних операцій та використовувати дужки для явного визначення порядку обчислення складних виразів.

Робота з NULL-значеннями потребує особливої уваги, оскільки NULL представляє відсутність значення, а не конкретне значення. Порівняння з NULL за допомогою звичайних операторів завжди повертає невизначений результат, тому для перевірки використовуються спеціальні оператори IS NULL та IS NOT NULL. Запит SELECT * FROM Employees WHERE MiddleName IS NULL знайде співробітників, у яких не вказано по батькові.

5. АГРЕГАТНІ ФУНКЦІЇ

Агрегатні функції дозволяють виконувати обчислення над набором рядків та повертати єдине значення, що робить їх незамінними інструментами для аналізу даних. SQL надає стандартний набір таких функцій, які покривають найпоширеніші аналітичні потреби.

Функція COUNT підраховує кількість рядків або ненульових значень у стовпці. COUNT(*) повертає загальну кількість рядків у таблиці або результуючій вибірці, тоді як COUNT(ColumnName) підраховує лише рядки з ненульовими значеннями у вказаному стовпці. Функція SUM обчислює суму числових значень, що корисно для знаходження загальних показників, таких як сумарний дохід або кількість товарів на складі.

Функції AVG, MIN та MAX відповідно обчислюють середнє значення, мінімальне та максимальне значення у наборі даних. Наприклад, запит SELECT AVG(Salary) FROM Employees визначить середню заробітну плату співробітників. Важливо розуміти, що більшість агрегатних функцій ігнорують NULL-значення при обчисленнях, що може впливати на результати аналізу.

Агрегатні функції можуть комбінуватися з іншими елементами SQL для створення складніших аналітичних запитів. Наприклад, можна обчислити відсоток максимального значення: SELECT ProductName, Price, (Price / MAX(Price) OVER ()) * 100 AS PercentOfMax FROM Products. Використання віконних функцій значно розширює аналітичні можливості SQL.

При використанні агрегатних функцій важливо пам'ятати, що всі неагреговані стовпці у SELECT-секції повинні бути включені до GROUP BY конструкції, інакше запит призведе до помилки або непередбачуваних результатів. Це правило забезпечує логічну коректність агрегації даних.

6. ЗАПИТИ З ГРУПУВАННЯМ

Конструкція GROUP BY є потужним механізмом для групування рядків за спільними значеннями одного або кількох стовпців з метою застосування

агрегатних функцій до кожної групи окремо. Це дозволяє переходити від аналізу окремих записів до аналізу категорій та агрегованих показників.

Базовий синтаксис групування передбачає вказівку стовпців групування після ключових слів GROUP BY. Запит SELECT Department, COUNT(*) AS EmployeeCount FROM Employees GROUP BY Department підрахує кількість співробітників у кожному відділі. Результат міститиме по одному рядку для кожного унікального відділу з відповідною кількістю працівників.

Групування можна здійснювати за кількома стовпцями, що дозволяє створювати ієрархічні категорії. Наприклад, SELECT Department, Position, AVG(Salary) FROM Employees GROUP BY Department, Position обчислить середню заробітну плату для кожної комбінації відділу та посади. Порядок стовпців у GROUP BY визначає ієрархію групування.

Конструкція HAVING використовується для фільтрації груп після агрегації, на відміну від WHERE, яка фільтрує окремі рядки до групування. Це дозволяє накладати умови на результати агрегатних функцій. Запит SELECT Department, AVG(Salary) AS AvgSalary FROM Employees GROUP BY Department HAVING AVG(Salary) > 50000 поверне лише ті відділи, де середня зарплата перевищує вказану суму.

Важливо розуміти порядок виконання різних частин SQL-запиту: спочатку виконується WHERE для фільтрації вхідних даних, потім GROUP BY для групування, після чого обчислюються агрегатні функції, і нарешті застосовується HAVING для фільтрації груп. ORDER BY виконується останнім для сортування фінального результату. Це розуміння допомагає правильно будувати складні аналітичні запити.

7. СКЛАДНІ ЗАПИТИ

Підзапити або вкладені запити дозволяють використовувати результати одного запиту як частину іншого запиту, що значно розширює виразні можливості SQL. Підзапит може розташовуватися в різних частинах головного запиту: у списку стовпців SELECT, у конструкції FROM як похідна таблиця, або в умові WHERE для динамічної фільтрації.

Скалярні підзапити повертають єдине значення і можуть використовуватися скрізь, де допускається звичайне значення. Наприклад, SELECT Name, Salary, (SELECT AVG(Salary) FROM Employees) AS AvgSalary FROM Employees порівнює зарплату кожного співробітника із середньою по компанії. Таблична форма підзапиту повертає множину рядків і може використовуватися з операторами IN, EXISTS, ANY або ALL.

Корельовані підзапити встановлюють зв'язок із зовнішнім запитом, посилаючись на його стовпці. Такий підзапит виконується один раз для кожного рядка зовнішнього запиту, що може впливати на продуктивність, але надає гнучкість у формулюванні логіки. Класичним прикладом є знаходження співробітників із заробітною платою вище середньої у їхньому відділі.

Оператор EXISTS перевіряє наявність рядків у результаті підзапиту, повертаючи істину, якщо підзапит повертає принаймні один рядок. Це ефективний спосіб перевірки існування зв'язаних даних. NOT EXISTS, відповідно, перевіряє відсутність відповідних рядків.

Конструкції UNION, INTERSECT та EXCEPT дозволяють комбінувати результати кількох запитів. UNION об'єднує результати, автоматично видаляючи дублікати, UNION ALL зберігає всі рядки включно з дублікатами. INTERSECT повертає лише спільні рядки з обох запитів, а EXCEPT виключає з першого запиту рядки, що присутні в другому. Усі запити, що об'єднуються, повинні мати однакову кількість стовпців з сумісними типами даних.

Загальні табличні вирази (Common Table Expressions, CTE) надають елегантний спосіб організації складних запитів через створення іменованих тимчасових результуючих наборів. Синтаксис WITH дозволяє визначити один або кілька CTE, які потім можуть використовуватися в головному запиті. CTE покращують читабельність коду та дозволяють будувати рекурсивні запити для роботи з ієрархічними структурами.

8. ЗАПИТИ НА ОНОВЛЕННЯ ДАНИХ

Оператор INSERT додає нові рядки до таблиці і може використовуватися в кількох формах. Найпростіша форма явно перераховує значення для всіх стовпців: INSERT INTO Students (StudentID, Name, Age) VALUES (1, 'Іван Петренко', 20). Можна вставити кілька рядків одним запитом, перераховуючи їх у дужках через кому. Альтернативно, INSERT може використовувати результат SELECT-запиту для масового копіювання даних.

Оператор UPDATE модифікує існуючі рядки в таблиці. Базовий синтаксис включає ключове слово UPDATE з ім'ям таблиці, SET зі списком стовпців та їхніх нових значень, та опціональну конструкцію WHERE для визначення рядків, що підлягають оновленню. Відсутність WHERE призведе до оновлення всіх рядків таблиці, що часто є небажаним наслідком помилки програміста. Запит UPDATE Employees SET Salary = Salary * 1.1 WHERE Department = 'Engineering' підвищить зарплату всім співробітникам інженерного відділу на десять відсотків.

Оператор DELETE видаляє рядки з таблиці згідно з заданими критеріями. Синтаксис DELETE FROM TableName WHERE Condition визначає, які саме рядки будуть видалені. Як і у випадку з UPDATE, відсутність умови WHERE призведе до видалення всіх рядків таблиці. Важливо розуміти різницю між DELETE, який видаляє рядки, але зберігає структуру таблиці, та TRUNCATE, який швидко очищає таблицю, але не дозволяє задавати умови та не може бути відкликаний у деяких системах.

При виконанні операцій модифікації даних критично важливим є дотримання цілісності бази даних. Обмеження цілісності, такі як первинні ключі, унікальність, зовнішні ключі та перевірочні обмеження, запобігають введенню некоректних даних. Спроба вставити дублікат первинного ключа або порушити обмеження зовнішнього ключа призведе до відхилення операції.

Транзакції забезпечують атомарність операцій, гарантуючи, що послідовність команд або виконується повністю, або не виконується взагалі. Це особливо важливо для операцій, що включають кілька взаємопов'язаних модифікацій даних. Команди BEGIN TRANSACTION, COMMIT та ROLLBACK керують межами транзакцій та дозволяють відміняти зміни у разі виявлення проблем.

Мова SQL є незамінним інструментом для роботи з реляційними базами даних, забезпечуючи потужні та гнучкі засоби для маніпулювання даними. Оволодіння SQL-запитами вимагає не лише знання синтаксису, але й розуміння принципів реляційної моделі, оптимізації продуктивності та найкращих практик проектування баз даних.

Початок роботи з SQL передбачає освоєння базових конструкцій SELECT, WHERE та JOIN, які складають основу більшості операцій читання даних. Поступово додаються знання про агрегатні функції та групування, що відкриває можливості аналітичної обробки інформації. Вміння будувати складні запити з підзапитами та табличними виразами дозволяє розв'язувати нетривіальні задачі обробки даних.

Операції модифікації даних через INSERT, UPDATE та DELETE є невід'ємною частиною роботи з базами даних і вимагають особливої уваги до питань цілісності інформації та безпеки. Розуміння механізмів транзакцій та обмежень цілісності є необхідним для створення надійних систем.

Важливо зазначити, що різні системи управління базами даних можуть мати певні відмінності в реалізації SQL-стандарту та надавати додаткові можливості. Тому практична робота з конкретною СУБД передбачає вивчення її документації та специфічних особливостей. Однак базові принципи та

конструкції SQL залишаються універсальними та застосовними в усіх сучасних реляційних системах.

Подальший розвиток навичок роботи з SQL передбачає освоєння таких тем, як оптимізація запитів, робота з індексами, процедурне розширення SQL через збережені процедури та тригери, а також інтеграція SQL-запитів у програмні додатки через різні інтерфейси доступу до даних.

ЗМІСТОВИЙ МОДУЛЬ 2
ЛОГІЧНЕ ТА ФІЗИЧНЕ ПРОЕКТУВАННЯ БАЗ ДАНИХ У РЕЛЯЦІЙНІЙ СУБД SQLITE

ТЕМА 4. ПРОЕКТУВАННЯ БАЗ ДАНИХ

1. Проект реляційної бази даних.
2. Етапи проектування.
3. Системний аналіз предметної області.
4. Даталогічне проектування.
5. Логічне проектування БД.
6. Інфологічна модель БД.
7. Правила перетворення ER-моделі на реляційну.
8. Алгоритм приведення семантичної моделі до 5-ї нормальної форми.

1. ПРОЕКТ РЕЛЯЦІЙНОЇ БАЗИ ДАНИХ.

Проектування баз даних є критично важливим етапом у створенні інформаційних систем, від якості якого залежить ефективність, надійність та масштабованість майбутнього програмного продукту. Неправильно спроектована база даних може призвести до численних проблем: від надмірності даних та аномалій при оновленні до неможливості ефективного виконання запитів та складності підтримки системи. Саме тому методології проектування баз даних розроблялися протягом десятиліть і продовжують вдосконалюватися.

Процес проектування бази даних не є лінійним, він включає кілька взаємопов'язаних етапів, кожен з яких має свої специфічні цілі, методи та результати. Від правильного розуміння предметної області до формальних математичних перетворень моделей даних – кожен крок вимагає певної компетентності та уваги до деталей. Сучасні системи управління базами даних, зокрема SQLite, надають потужні інструменти для реалізації спроектованих структур, однак якість самого проекту залишається визначальним фактором успіху.

Проект реляційної бази даних являє собою комплексний документ, що містить повний опис структури даних, правил їх організації, взаємозв'язків між сутностями та обмежень цілісності. Цей документ служить основою для фізичної реалізації бази даних у конкретній системі управління базами даних та є важливим артефактом для подальшої підтримки та розвитку інформаційної системи.

Основною метою проектування є створення такої структури даних, яка б оптимально відображала предметну область, забезпечувала мінімальну надмірність інформації, гарантувала цілісність даних та забезпечувала ефективне виконання типових операцій. При цьому необхідно досягти балансу між нормалізацією структури, яка мінімізує аномалії, та продуктивністю системи, яка може вимагати певної денормалізації для прискорення виконання складних запитів.

Якісний проект бази даних має відповідати низці вимог. По-перше, він повинен бути повним, тобто охоплювати всі необхідні аспекти предметної області та забезпечувати можливість зберігання всієї релевантної інформації. По-друге, проект має бути несуперечливим, не допускаючи конфліктуючих правил або обмежень. По-третє, важливою є гнучкість проекту, його здатність адаптуватися до змін у вимогах без кардинального переробляння структури. По-четверте, проект повинен забезпечувати ефективність виконання типових операцій над даними.

Реляційна модель даних, запропонована Едгаром Коддом у 1970 році, базується на математичній теорії множин та логіці першого порядку. Вона передбачає представлення даних у вигляді таблиць (відношень), де кожен рядок є кортежем, а стовпці визначають атрибути. Реляційна модель має строгу теоретичну основу, що дозволяє формально обґрунтовувати правильність проектних рішень та застосовувати математичні методи для оптимізації структури даних.

Важливим аспектом проектування є визначення ключів та зв'язків між таблицями. Первинний ключ однозначно ідентифікує кожен запис у таблиці та гарантує унікальність рядків. Зовнішні ключі встановлюють зв'язки між таблицями, забезпечуючи референційну цілісність даних. Правильне визначення ключів є фундаментом для забезпечення якості даних та ефективності їх обробки.

2. ЕТАПИ ПРОЕКТУВАННЯ БАЗИ ДАНИХ

Проектування бази даних традиційно поділяється на кілька послідовних етапів, кожен з яких має специфічні завдання та використовує відповідні методології. Хоча в реальних проектах ці етапи можуть частково перекриватися або виконуватися ітеративно, розуміння їх логічної послідовності є важливим для структурованого підходу до проектування.

Концептуальне проектування є першим та найбільш абстрактним етапом. На цьому етапі створюється високорівнева модель предметної області, незалежна від особливостей конкретної системи управління базами даних. Основним результатом концептуального проектування є інфологічна модель, яка описує сутності предметної області, їх атрибути та взаємозв'язки. Найпоширенішим інструментом для концептуального моделювання є ER-діаграми (Entity-Relationship), які забезпечують наочне графічне представлення структури даних.

Логічне проектування перетворює концептуальну модель у логічну модель даних, орієнтовану на конкретну модель даних (реляційну, документоорієнтовану, графову тощо), але все ще незалежну від конкретної СУБД. Для реляційної моделі цей етап включає перетворення ER-моделі в набір таблиць, визначення первинних та зовнішніх ключів, а також застосування правил нормалізації для усунення надмірності даних та запобігання аномаліям.

Фізичне проектування адаптує логічну модель до специфіки обраної СУБД. На цьому етапі визначаються конкретні типи даних для атрибутів з урахуванням можливостей СУБД, створюються індекси для оптимізації

виконання запитів, розробляються представлення (views), збережені процедури та тригери. Для SQLite, наприклад, необхідно враховувати обмеження на типи даних, особливості роботи з транзакціями та специфіку оптимізації запитів.

Імплементація та тестування завершують процес проектування. На цьому етапі створюється фізична база даних за допомогою DDL-команд, завантажуються початкові дані, розробляються та тестуються типові запити. Важливою частиною є тестування продуктивності та перевірка коректності роботи обмежень цілісності.

Кожен етап проектування супроводжується створенням відповідної документації. Концептуальна модель документується у вигляді ER-діаграм з описом сутностей та зв'язків. Логічна модель представляється схемою реляційних таблиць з визначенням ключів та обмежень. Фізична модель включає DDL-скрипти створення бази даних та супровідну документацію щодо індексів та оптимізації.

3. СИСТЕМНИЙ АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

Системний аналіз предметної області є фундаментальним початковим етапом проектування, від якості якого залежить адекватність всієї майбутньої системи. Цей етап передбачає глибоке вивчення бізнес-процесів, виявлення інформаційних потреб користувачів та формулювання вимог до майбутньої бази даних.

Процес системного аналізу починається зі збору інформації про предметну область через різні канали. Інтерв'ювання стейкхолдерів дозволяє зрозуміти їхні потреби, робочі процеси та очікування від системи. Аналіз існуючої документації, регламентів та звітів виявляє формальні вимоги та обмеження. Спостереження за роботою користувачів розкриває реальні робочі процеси, які можуть відрізнятися від задекларованих процедур. Вивчення аналогічних систем надає корисні інсайти щодо можливих підходів до вирішення типових задач.

Важливим результатом системного аналізу є ідентифікація сутностей предметної області. Сутність являє собою об'єкт реального світу, інформацію про який необхідно зберігати в базі даних. Це можуть бути матеріальні об'єкти (товари, співробітники, обладнання), концептуальні поняття (замовлення, проекти, курси) або події (транзакції, візити, виклики). Кожна сутність характеризується набором атрибутів – властивостей, що описують її стан та характеристики.

Виявлення зв'язків між сутностями є не менш важливим завданням. Зв'язки відображають взаємодії та залежності між об'єктами предметної області. Класифікація зв'язків за кардинальністю (один-до-одного, один-до-багатьох, багато-до-багатьох) та обов'язковістю (обов'язкові чи опціональні) допомагає точно моделювати бізнес-правила та обмеження. Наприклад, зв'язок між студентом та університетом є зв'язком багато-до-одного, оскільки кожен студент навчається в одному університеті, але університет має багато студентів.

Формулювання функціональних вимог визначає операції, які система повинна підтримувати. Це включає типові запити на читання даних (звіти, пошук, аналітика), операції модифікації (додавання, оновлення, видалення), вимоги до продуктивності та обсягів даних. Розуміння частоти та складності операцій критично важливе для прийняття рішень щодо структури даних та індексації на наступних етапах проектування.

Нефункціональні вимоги охоплюють аспекти, пов'язані з безпекою, надійністю, масштабованістю та сумісністю системи. Для SQLite особливо важливими є вимоги щодо обсягів даних, оскільки ця СУБД оптимізована для невеликих та середніх баз даних, а також вимоги до паралельного доступу, оскільки SQLite має певні обмеження на одночасну роботу кількох процесів.

4. ДАТАЛОГІЧНЕ ПРОЕКТУВАННЯ

Даталогічне проектування являє собою процес створення формальної моделі даних, яка описує структуру інформації у термінах, незалежних від конкретної СУБД, але орієнтованих на реляційну модель даних. Цей етап є містком між абстрактним концептуальним моделюванням та конкретною фізичною реалізацією.

Основним завданням даталогічного проектування є перетворення інфологічної моделі в набір реляційних схем. Кожна сутність, як правило, відображається в окрему таблицю, атрибути сутності стають стовпцями таблиці, а екземпляри сутності представляються рядками. Однак це базове правило має численні нюанси, особливо при роботі зі складними зв'язками та спеціальними типами сутностей.

Визначення первинних ключів є критичним кроком даталогічного проектування. Первинний ключ може бути природним (складеним з атрибутів, що природно ідентифікують сутність) або штучним (суррогатним ключем, створеним спеціально для ідентифікації). У SQLite часто використовуються автоінкрементні цілочисельні ідентифікатори як суррогатні ключі, що

забезпечує простоту та ефективність. Вибір між природними та суррогатними ключами залежить від специфіки предметної області та вимог до стабільності ідентифікаторів.

Моделювання зв'язків між таблицями здійснюється через зовнішні ключі. Зв'язок один-до-багатьох реалізується розміщенням зовнішнього ключа в таблиці на стороні "багато", що посилається на первинний ключ таблиці на стороні "один". Зв'язок багато-до-багатьох вимагає створення додаткової таблиці-зв'язки (асоціативної таблиці), яка містить зовнішні ключі до обох пов'язаних таблиць. Зв'язок один-до-одного може бути реалізований як через зовнішній ключ з обмеженням унікальності, так і через об'єднання сутностей в одну таблицю.

Визначення доменів атрибутів передбачає специфікацію типів даних, допустимих діапазонів значень та обмежень. Хоча SQLite має гнучку систему типів із динамічною типізацією, явне визначення типів у схемі таблиці покращує цілісність даних та продуктивність. Окрім типу даних, важливо визначити, чи може атрибут містити NULL-значення, які за замовчуванням значення повинні використовуватися, та які додаткові обмеження (CHECK constraints) мають застосовуватися.

Документування даталогічної моделі зазвичай здійснюється у вигляді схем реляцій з позначенням ключів та зв'язків. Для кожної таблиці створюється опис, що включає назву, призначення, перелік атрибутів з їх типами та обмеженнями, первинний ключ та зовнішні ключі. Така документація є важливим артефактом для команди розробників та адміністраторів бази даних.

5. ЛОГІЧНЕ ПРОЕКТУВАННЯ БД

Логічне проектування бази даних поглиблює даталогічну модель, застосовуючи формальні методи оптимізації структури даних та забезпечення їх якості. Центральним процесом логічного проектування є нормалізація – систематичний підхід до організації таблиць, який мінімізує надмірність даних та запобігає аномаліям при виконанні операцій вставки, оновлення та видалення.

Теорія нормалізації базується на концепції функціональних залежностей між атрибутами. Функціональна залежність A → B означає, що значення атрибута A однозначно визначає значення атрибута B. Наприклад, у контексті бази даних університету, ідентифікатор студента функціонально визначає його ім'я, дату народження та інші персональні дані. Аналіз функціональних залежностей дозволяє виявити проблемні конструкції в схемі даних та застосувати відповідні нормальні форми.

Перша нормальна форма (1НФ) вимагає атомарності значень атрибутів, тобто кожен атрибут повинен містити лише неподільні значення, а не списки чи множини. Таблиця не повинна містити повторюваних груп атрибутів. Наприклад, замість стовпців Телефон1, Телефон2, Телефон3 слід створити окрему таблицю телефонів із зовнішнім ключем до основної таблиці.

Друга нормальна форма (2НФ) стосується таблиць із складеними первинними ключами та вимагає, щоб кожен неключовий атрибут функціонально залежав від усього первинного ключа, а не від його частини. Якщо виявляється часткова залежність, відповідні атрибути виділяються в окрему таблицю. Це усуває надмірність, пов'язану з повторенням даних для різних комбінацій складеного ключа.

Третя нормальна форма (3НФ) усуває транзитивні залежності, коли неключовий атрибут залежить від іншого неключового атрибута. Наприклад, якщо таблиця студентів містить код відділення та назву відділення, виникає транзитивна залежність: студент → код відділення → назва відділення. Для досягнення 3НФ необхідно винести інформацію про відділення в окрему таблицю.

Нормальна форма Бойса-Кодда (НФБК) є посиленою версією 3НФ і вимагає, щоб кожна функціональна залежність мала лівою частиною потенційний ключ. Хоча більшість таблиць у 3НФ автоматично задовольняють НФБК, існують специфічні випадки, коли необхідне додаткове розбиття.

Четверта нормальна форма (4НФ) стосується багатозначних залежностей та вимагає, щоб таблиця не містила незалежних багатозначних залежностей. П'ята нормальна форма (5НФ) або проекційно-з'єднувальна нормальна форма є найвищим ступенем нормалізації та стосується залежностей з'єднання. На практиці досягнення 3НФ або НФБК зазвичай є достатнім для більшості застосунків.

Важливо розуміти, що нормалізація не є самоціллю. У деяких випадках контрольована денормалізація може бути виправданою для підвищення продуктивності читання даних за рахунок деякої надмірності. Рішення про денормалізацію повинні прийматися свідомо, з розумінням компромісів між нормалізацією та продуктивністю.

6. ІНФОЛОГІЧНА МОДЕЛЬ БД

Інфологічна модель являє собою концептуальне уявлення предметної області на високому рівні абстракції, незалежне від особливостей конкретної моделі даних чи системи управління базами даних. Ця модель служить засобом

комунікації між аналітиками, які розуміють предметну область, та проектувальниками баз даних, які володіють технічними знаннями.

Основними компонентами інфологічної моделі є сутності, атрибути та зв'язки. Сутність представляє клас об'єктів реального світу, які мають загальні властивості та поведінку. Назви сутностей зазвичай є іменниками в однині (Студент, Курс, Викладач). Кожна сутність характеризується набором атрибутів, які описують її властивості. Атрибути можуть бути простими або складеними, однозначними або багатозначними, зберігаються або обчислюваними.

Зв'язки відображають взаємодії між сутностями та мають важливі характеристики. Кардинальність зв'язку визначає, скільки екземплярів однієї сутності може бути пов'язано з екземпляром іншої сутності. Зв'язок один-до-одного (1:1) означає, що кожен екземпляр першої сутності пов'язаний рівно з одним екземпляром другої сутності. Зв'язок один-до-багатьох (1:N) є найпоширенішим та означає, що один екземпляр першої сутності може бути пов'язаний з багатьма екземплярами другої сутності. Зв'язок багато-до-багатьох (M:N) дозволяє багатьом екземплярам обох сутностей бути взаємопов'язаними.

Обов'язковість участі в зв'язку визначає, чи повинен кожен екземпляр сутності обов'язково брати участь у зв'язку. Це позначається мінімальною кардинальністю: 0 для опціональної участі та 1 для обов'язкової. Наприклад, у зв'язку між студентом та гуртожитком студент може опціонально проживати в гуртожитку (мінімальна кардинальність 0), тоді як гуртожиток обов'язково має мати принаймні одного мешканця для того, щоб фіксуватися в системі.

Слабкі сутності є особливим типом сутностей, які не можуть існувати незалежно від іншої (сильної) сутності. Класичним прикладом є залежні члени сім'ї співробітника – вони не мають сенсу без співробітника, до якого вони прив'язані. Слабкі сутності зазвичай не мають власного повного первинного ключа, а ідентифікуються через комбінацію свого часткового ключа та ключа сильної сутності.

Спеціалізація та узагальнення дозволяють моделювати ієрархічні відношення між сутностями. Спеціалізація передбачає виділення підтипів загальної сутності з додатковими специфічними атрибутами. Наприклад, сутність Особа може мати підтипи Студент, Викладач та Адміністратор, кожен з яких має специфічні атрибути на додаток до загальних атрибутів особи. Узагальнення є протилежним процесом – об'єднання кількох подібних сутностей у більш загальну.

Агрегація дозволяє розглядати зв'язок разом із пов'язаними сутностями як сутність вищого рівня. Це корисно, коли потрібно встановити зв'язок із самим зв'язком. Наприклад, проект може залучати співробітників у певних ролях (зв'язок між Проектом та Співробітником через Роль), і до цього зв'язку можна прив'язати оцінку виконання роботи.

7. ПРАВИЛА ПЕРЕТВОРЕННЯ ER-МОДЕЛІ НА РЕЛЯЦІЙНУ

Перетворення ER-моделі в реляційну схему є формалізованим процесом, що забезпечує збереження семантики концептуальної моделі при переході до логічного рівня проектування. Існує набір систематичних правил, застосування яких дозволяє отримати коректну реляційну схему.

Правило перетворення сильних сутностей є найпростішим: кожна сильна сутність перетворюється в окрему таблицю. Атрибути сутності стають стовпцями таблиці, а ідентифікатор сутності стає первинним ключем таблиці. Наприклад, сутність Студент з атрибутами StudentID, Name, BirthDate, Email перетворюється в таблицю Students з відповідними стовпцями та первинним ключем StudentID.

Слабкі сутності також перетворюються в окремі таблиці, але їх первинний ключ складається з комбінації власного часткового ключа та зовнішнього ключа, що посилається на ідентифікатор власника (сильної сутності). Наприклад, сутність Залежний член сім'ї, пов'язана зі Співробітником, перетворюється в таблицю Dependents з первинним ключем (EmployeeID, DependentName), де EmployeeID є зовнішнім ключем до таблиці Employees.

Перетворення зв'язків один-до-багатьох здійснюється розміщенням зовнішнього ключа в таблиці на стороні "багато". Цей зовнішній ключ посилається на первинний ключ таблиці на стороні "один". Наприклад, зв'язок "Відділ має багато співробітників" реалізується додаванням стовпця DepartmentID до таблиці Employees, який є зовнішнім ключем до таблиці Departments. Якщо зв'язок має власні атрибути, вони також додаються до таблиці на стороні "багато".

Зв'язки багато-до-багатьох вимагають створення окремої таблиці-зв'язки (junction table або associative table). Ця таблиця містить зовнішні ключі до обох пов'язаних таблиць, і комбінація цих ключів зазвичай формує складений первинний ключ. Якщо зв'язок має власні атрибути, вони стають додатковими стовпцями таблиці-зв'язки. Наприклад, зв'язок "Студенти записуються на курси" з атрибутом Grade перетворюється в таблицю Enrollments з стовпцями StudentID, CourseID та Grade.

Зв'язки один-до-одного можуть бути реалізовані кількома способами залежно від обов'язковості участі. Якщо обидві сторони обов'язкові, сутності можна об'єднати в одну таблицю. Якщо одна сторона опціональна, зовнішній ключ розміщується в таблиці опціональної сторони з обмеженням унікальності. Якщо обидві сторони опціональні, може бути створена окрема таблиця-зв'язок, подібно до зв'язку багато-до-багатьох, але з обмеженнями унікальності на обидва зовнішні ключі.

Багатозначні атрибути перетворюються в окремі таблиці. Ця таблиця містить зовнішній ключ до основної таблиці та стовпець для зберігання значень багатозначного атрибута. Первинний ключ складається з комбінації зовнішнього ключа та значення атрибута. Наприклад, багатозначний атрибут PhoneNumbers сутності Person перетворюється в таблицю PersonPhones з стовпцями PersonID та PhoneNumber.

Складені атрибути можуть оброблятися двома способами: або розбиватися на прості атрибути (наприклад, Address розбивається на Street, City, PostalCode), або зберігатися як єдиний атрибут складного типу, якщо СУБД це підтримує. У SQLite зазвичай використовується перший підхід через відсутність складних типів даних.

Ієрархії спеціалізації/узагальнення можуть перетворюватися трьома основними способами. Перший спосіб – єдина таблиця для всієї ієрархії з додатковим дискримінаторним стовпцем для ідентифікації типу та NULL-значеннями для неактуальних атрибутів. Другий спосіб – окрема таблиця для кожного підтипу, що містить як загальні, так і специфічні атрибути. Третій спосіб – таблиця для базового типу з загальними атрибутами та окремі таблиці для підтипів зі специфічними атрибутами, пов'язані через зовнішні ключі. Вибір методу залежить від характеру використання даних та вимог до продуктивності.

8. АЛГОРИТМ ПРИВЕДЕННЯ СЕМАНТИЧНОЇ МОДЕЛІ ДО П'ЯТОЇ НОРМАЛЬНОЇ ФОРМИ

П'ята нормальна форма (5NF), також відома як проекційно-з'єднувальна нормальна форма (PJ/NF), представляє собою найвищий рівень нормалізації реляційних баз даних, спрямований на усунення аномалій, пов'язаних із залежностями з'єднання. Приведення семантичної моделі до 5NF передбачає послідовне застосування правил декомпозиції відношень з метою забезпечення відсутності надлишковості інформації та аномалій оновлення.

Процес нормалізації до п'ятої нормальної форми розпочинається з перевірки виконання умов попередніх нормальних форм. Семантична модель

повинна відповідати вимогам четвертої нормальної форми, що означає відсутність нетривіальних багатозначних залежностей. Після підтвердження відповідності 4NF здійснюється аналіз відношень на наявність залежностей з'єднання.

Ключовим поняттям при приведенні до 5NF є залежність з'єднання, яка виникає тоді, коли відношення може бути декомпоновано на кілька проекцій, а потім відновлено шляхом природного з'єднання цих проекцій без втрати або появи додаткової інформації. Відношення знаходиться в п'ятій нормальній формі, якщо воно не може бути декомпоновано на менші відношення без втрат, або якщо всі залежності з'єднання в ньому випливають із потенційних ключів.

Алгоритм приведення до 5NF включає наступні етапи. Спершу виконується ідентифікація всіх залежностей з'єднання у відношеннях семантичної моделі. Для кожного відношення аналізуються можливі варіанти його декомпозиції на підмножини атрибутів таким чином, щоб з'єднання цих підмножин повністю відновлювало вихідне відношення. Особливу увагу приділяють виявленню циклічних залежностей між атрибутами, які не можуть бути виражені через функціональні або багатозначні залежності.

Після виявлення залежностей з'єднання здійснюється перевірка їх тривіальності. Залежність з'єднання вважається тривіальною, якщо одна з проекцій співпадає з вихідним відношенням. Нетривіальні залежності з'єднання вказують на необхідність декомпозиції відношення. Декомпозиція виконується шляхом розбиття відношення на кілька менших відношень відповідно до виявленої залежності з'єднання, при цьому кожне нове відношення містить підмножину атрибутів вихідного відношення.

Критично важливим аспектом алгоритму є забезпечення властивості безвтратності декомпозиції. Для цього перевіряється, чи може вихідне відношення бути точно відновлено шляхом природного з'єднання декомпонованих відношень. Використовуються методи теорії залежностей, зокрема побудова таблиць Чейза або застосування теореми Хіта для верифікації коректності декомпозиції.

Особливістю п'ятої нормальної форми є те, що вона застосовується у випадках складних багатосторонніх зв'язків, які не можуть бути адекватно представлені через бінарні відношення. Типовим прикладом є ситуації, коли між трьома або більше сутностями існують незалежні зв'язки, які при зберіганні в одному відношенні призводять до надмірної надлишковості даних.

Після виконання декомпозиції проводиться верифікація отриманої моделі. Перевіряється збереження всіх семантичних обмежень вихідної моделі,

відсутність аномалій вставки, видалення та оновлення, а також можливість виконання всіх необхідних запитів до бази даних. У випадку виявлення проблем здійснюється коригування структури відношень або перегляд стратегії декомпозиції.

Завершальним етапом алгоритму є документування отриманої схеми бази даних у п'ятій нормальній формі, включаючи опис всіх відношень, їх атрибутів, потенційних ключів, зовнішніх ключів та семантичних обмежень цілісності. Створюється відображення між вихідною семантичною моделлю та нормалізованою схемою для забезпечення трасування змін та підтримки цілісності проектних рішень.

ТЕМА 5. ЦІЛІСНІСТЬ ДАНИХ. ЗАХИСТ БАЗ ДАНИХ

1. Методи захисту баз даних
2. Безпека даних та її функції
3. Адміністрування баз даних
4. Управління доступом
5. Шифрування даних
6. Засоби підтримки безпеки в SQL та SQLite

1. МЕТОДИ ЗАХИСТУ БАЗ ДАНИХ

Захист і цілісність даних є критично важливими аспектами функціонування сучасних інформаційних систем. У епоху цифрової трансформації, коли організації зберігають величезні обсяги конфіденційної інформації, забезпечення безпеки баз даних стає не просто технічним завданням, а стратегічним пріоритетом. Втрата, несанкціоноване розкриття або пошкодження даних можуть призвести до фінансових збитків, репутаційних ризиків та юридичних наслідків. Комплексний підхід до захисту баз даних включає організаційні, технічні та програмні заходи, що забезпечують конфіденційність, цілісність та доступність інформації.

Методи захисту баз даних – це сукупність технічних, організаційних та програмних заходів, спрямованих на запобігання несанкціонованому доступу, модифікації, розкриттю або знищенню інформації, що зберігається в базі даних. Ефективна система захисту базується на багаторівневому підході, який враховує різноманітні загрози та вразливості.

Фізичний захист – це комплекс заходів щодо забезпечення безпеки апаратного обладнання та носіїв інформації від фізичного доступу несанкціонованих осіб, природних катастроф, пожеж, крадіжок та інших

фізичних загроз. Фізичний захист включає контроль доступу до серверних приміщень, використання систем відеоспостереження, організацію резервного живлення, застосування систем пожежогасіння та клімат-контролю. Належний фізичний захист є фундаментом безпеки, оскільки жодні програмні засоби не захистять дані, якщо зловмисник отримає прямий доступ до серверного обладнання.

Логічний захист – це система програмних та алгоритмічних механізмів, що контролюють доступ до даних на рівні операційної системи, СУБД та прикладних програм. Логічний захист реалізується через механізми аутентифікації, авторизації, розмежування прав доступу, аудиту та шифрування. На відміну від фізичного захисту, логічний захист оперує з електронними ідентифікаторами, паролями, цифровими сертифікатами та правами доступу користувачів.

Криптографічний захист – це застосування математичних алгоритмів шифрування для перетворення інформації в нечитабельний вигляд, що унеможливлює її розуміння без знання відповідного ключа дешифрування. Криптографічні методи включають симетричне шифрування, асиметричне шифрування, хешування, електронний цифровий підпис та інфраструктуру відкритих ключів. Шифрування застосовується як для захисту даних при зберіганні, так і при передачі через мережі зв'язку.

Організаційний захист – це комплекс адміністративних заходів, процедур та політик, що регламентують роботу з базами даних, визначають відповідальність персоналу, порядок надання та відкликання прав доступу, процедури резервного копіювання та відновлення даних. Організаційний захист включає розробку політики безпеки, навчання персоналу, проведення аудитів безпеки, класифікацію інформації за рівнями конфіденційності.

Основні методи захисту можна систематизувати за рівнями реалізації:

1. Захист на рівні мережі – використання міжмережевих екранів, систем виявлення вторгнень, віртуальних приватних мереж

2. Захист на рівні операційної системи – налаштування прав доступу до файлів, застосування політик безпеки ОС

3. Захист на рівні СУБД – механізми аутентифікації користувачів, розмежування прав, аудит операцій

4. Захист на рівні додатків – валідація вхідних даних, захист від SQL-ін'єкцій, безпечне кодування

5. Захист на рівні даних – шифрування окремих полів або всієї бази даних, маскування чутливої інформації

Резервне копіювання – це процес створення та зберігання копій бази даних у безпечному місці для можливості відновлення інформації у разі її втрати, пошкодження або знищення внаслідок апаратних збоїв, програмних помилок, зловмисних дій або природних катастроф. Стратегія резервного копіювання визначає частоту створення копій, типи резервних копій (повні, інкрементні, диференціальні), місця зберігання та процедури відновлення.

2. БЕЗПЕКА ДАНИХ ТА ЇЇ ФУНКЦІЇ

Безпека даних – це стан захищеності інформації в базі даних від несанкціонованого доступу, використання, розкриття, руйнування, модифікації або порушення доступності, що забезпечується комплексом технічних, організаційних та правових заходів. Безпека даних є багатогранним поняттям, що охоплює різні аспекти інформаційної безпеки.

Конфіденційність – це властивість інформації бути доступною лише для авторизованих користувачів та процесів, що мають законне право на доступ до неї. Порушення конфіденційності відбувається, коли інформація розкривається особам, які не мають відповідних прав. Забезпечення конфіденційності особливо критично для персональних даних, комерційної таємниці, фінансової та медичної інформації.

Цілісність даних – це властивість інформації залишатися повною, точною та незміненою протягом усього життєвого циклу, за винятком авторизованих модифікацій. Цілісність забезпечує, що дані не були змінені несанкціоновано або випадково під час зберігання, обробки чи передачі. Порушення цілісності може призвести до прийняття неправильних рішень на основі спотворених даних.

Доступність – це властивість інформаційної системи забезпечувати своєчасний та надійний доступ до даних для авторизованих користувачів у потрібний момент часу. Доступність передбачає, що система працює стабільно, швидко реагує на запити та здатна витримувати навантаження. Порушення доступності (атаки типу відмова в обслуговуванні) можуть паралізувати роботу організації навіть без втрати або розкриття даних.

Функції безпеки даних включають наступні ключові напрямки діяльності:

1. Ідентифікація користувачів та процесів, що звертаються до бази даних
2. Аутентифікація для підтвердження заявленої ідентичності
3. Авторизація для визначення дозволених операцій для кожного користувача

4. Аудит для реєстрації всіх значущих подій у системі

5. Забезпечення відмовостійкості та відновлюваності системи

6. Захист від зловмисного програмного забезпечення

7. Моніторинг безпеки та виявлення аномальної активності

Аутентифікація – це процес перевірки та підтвердження ідентичності користувача, процесу або пристрою, що намагається отримати доступ до системи, шляхом порівняння наданих облікових даних з еталонною інформацією, збереженою в системі. Аутентифікація може базуватися на знанні (пароль, PIN-код), володінні (смарт-карта, токен) або біометричних характеристиках (відбиток пальця, розпізнавання обличчя).

Авторизація – це процес надання або відмови у доступі до ресурсів системи після успішної аутентифікації користувача на основі визначених прав та політик безпеки. Авторизація визначає, які саме операції (читання, запис, видалення, виконання) дозволені конкретному користувачу щодо конкретних об'єктів бази даних.

Аудит безпеки – це систематичний процес реєстрації, аналізу та моніторингу подій, пов'язаних з доступом до даних та виконанням операцій у базі даних, з метою виявлення порушень політики безпеки, несанкціонованих дій та аномальної поведінки користувачів. Журнали аудиту зберігають інформацію про час події, ідентифікатор користувача, тип операції та об'єкти, до яких здійснювався доступ.

3. АДМІНІСТРУВАННЯ БАЗ ДАНИХ

Адміністрування баз даних – це комплекс організаційних, технічних та управлінських заходів, спрямованих на забезпечення ефективного функціонування, безпеки, цілісності та доступності бази даних протягом усього її життєвого циклу. Адміністрування охоплює планування, проектування, впровадження, підтримку, оптимізацію та захист баз даних.

Адміністратор бази даних – це фахівець, відповідальний за проектування, впровадження, обслуговування та захист бази даних, який володіє глибокими знаннями про СУБД, розуміє бізнес-процеси організації та здатен забезпечити надійне функціонування інформаційної системи. Адміністратор БД має найвищі привілеї в системі та несе відповідальність за збереження даних.

Основні обов'язки адміністратора бази даних включають:

1. Проектування логічної та фізичної структури бази даних

2. Створення та підтримка схеми бази даних, таблиць, індексів, представлень

3. Управління обліковими записами користувачів та їхніми правами доступу

4. Налаштування параметрів продуктивності та оптимізація запитів

5. Планування та виконання резервного копіювання даних

6. Відновлення даних після збоїв або втрат

7. Моніторинг роботи системи та усунення проблем

8. Забезпечення безпеки та цілісності даних

9. Планування розвитку та масштабування системи

10. Документування структури бази даних та процедур обслуговування

Політика безпеки бази даних – це формалізований документ, що визначає цілі, принципи, правила та процедури забезпечення безпеки інформації в базі даних, розподіл відповідальності між співробітниками, вимоги до захисту даних різних категорій конфіденційності. Політика безпеки є основою для впровадження конкретних технічних та організаційних заходів захисту.

Ефективне адміністрування передбачає створення процедур для типових операцій, автоматизацію рутинних завдань, ведення детальної документації, регулярне тестування процедур відновлення та постійне підвищення кваліфікації адміністраторів у відповідь на еволюцію загроз та технологій.

4. УПРАВЛІННЯ ДОСТУПОМ

Управління доступом – це система механізмів та політик, що контролюють та обмежують можливість користувачів та процесів виконувати операції з об'єктами бази даних на основі визначених правил та привілеїв. Управління доступом є ключовим елементом забезпечення безпеки та реалізує принцип найменших привілеїв.

Принцип найменших привілеїв – це концепція безпеки, згідно з якою кожен користувач, програма або процес повинні мати доступ лише до тієї інформації та ресурсів, які абсолютно необхідні для виконання їхніх легітимних функцій, і не більше. Цей принцип мінімізує потенційну шкоду від помилок користувачів або компрометації облікових записів.

Дискреційне управління доступом – це модель контролю доступу, при якій власник об'єкта має повноваження визначати, які користувачі можуть отримати доступ до цього об'єкта та які операції їм дозволені. У реляційних СУБД дискреційне управління реалізується через команди GRANT та REVOKE, які дозволяють надавати та відкликати привілеї.

Обов'язкове управління доступом – це модель контролю доступу, при якій доступ до об'єктів визначається системними правилами на основі міток

конфіденційності, присвоєних користувачам та об'єктам, незалежно від бажання власника об'єкта. Ця модель типова для систем з високими вимогами до безпеки, таких як військові або урядові організації.

Рольова модель управління доступом – це підхід до організації прав доступу, при якому привілеї призначаються не окремим користувачам, а ролям, які представляють певні посади або функції в організації, а користувачі отримують доступ шляхом призначення їм відповідних ролей. Рольова модель спрощує адміністрування, особливо в великих організаціях з частою зміною персоналу.

Переваги рольової моделі включають:

1. Спрощення управління правами доступу великої кількості користувачів;
2. Забезпечення узгодженості прав користувачів з однаковими функціональними обов'язками
3. Полегшення процесів призначення та відкликання прав при зміні персоналу
4. Покращення аудиту та дотримання нормативних вимог
5. Зменшення ймовірності помилок при налаштуванні прав доступу

Привілеї користувачів – це конкретні дозволи на виконання певних операцій з об'єктами бази даних, такі як вибірка даних (SELECT), вставка (INSERT), оновлення (UPDATE), видалення (DELETE), створення об'єктів (CREATE), зміна структури (ALTER) та інші адміністративні привілеї. Привілеї можуть надаватися на різних рівнях деталізації: на всю базу даних, окремі таблиці, конкретні стовпці або навіть окремі рядки через механізми представлень.

Представлення як засіб безпеки – це віртуальні таблиці, що створюються на основі запитів до базових таблиць і можуть використовуватися для обмеження доступу користувачів до підмножини даних або стовпців, приховуючи чутливу інформацію. Користувачам надаються права лише на представлення, а не на базові таблиці, що реалізує детальний контроль доступу.

5. ШИФРУВАННЯ ДАНИХ

Шифрування даних – це процес перетворення інформації з читабельного вигляду (відкритого тексту) в нечитабельну форму (шифротекст) за допомогою криптографічних алгоритмів та ключів, що робить дані незрозумілими для несанкціонованих осіб навіть у разі їх перехоплення або викрадення. Шифрування є одним з найбільш надійних методів захисту конфіденційності даних.

Симетричне шифрування – це метод криптографічного перетворення, при якому один і той же ключ використовується як для шифрування, так і для дешифрування інформації. Симетричні алгоритми характеризуються високою швидкістю обробки даних, що робить їх ефективними для шифрування великих обсягів інформації. Найпоширеніші симетричні алгоритми включають AES (Advanced Encryption Standard), DES (Data Encryption Standard), та Blowfish.

Асиметричне шифрування – це криптографічний метод, що використовує пару математично пов'язаних ключів: відкритий ключ для шифрування та приватний ключ для дешифрування, причому знання відкритого ключа не дозволяє обчислити приватний ключ за прийнятний час. Асиметричне шифрування вирішує проблему безпечного обміну ключами, але працює значно повільніше симетричного. Типовими алгоритмами є RSA, ECC (Elliptic Curve Cryptography).

Хешування – це процес перетворення вхідних даних довільної довжини в рядок фіксованої довжини (хеш-значення або дайджест) за допомогою односпрямованої функції, яка робить практично неможливим відновлення вихідних даних з хеш-значення. Хешування використовується для зберігання паролів, перевірки цілісності даних, створення цифрових підписів. Популярні хеш-функції включають SHA-256, SHA-3, bcrypt.

Основні варіанти застосування шифрування в базах даних:

1. Шифрування на рівні стовпців – окремі поля, що містять чутливу інформацію, шифруються індивідуально.
2. Шифрування на рівні таблиць – вся таблиця шифрується цілком.
3. Шифрування на рівні бази даних – вся база даних зберігається в зашифрованому вигляді.
4. Прозоре шифрування даних – СУБД автоматично шифрує дані при записі та дешифрує при читанні без змін у додатках.
5. Шифрування резервних копій – копії бази даних зберігаються в зашифрованому форматі.
6. Шифрування каналів зв'язку – дані шифруються при передачі між клієнтом та сервером через протоколи TLS/SSL.

Управління ключами – це комплекс процедур генерації, розповсюдження, зберігання, ротації та знищення криптографічних ключів протягом їхнього життєвого циклу. Належне управління ключами критично важливе для безпеки, оскільки компрометація ключів робить шифрування безглузким. Ключі повинні зберігатися окремо від зашифрованих даних, регулярно змінюватися та захищатися від несанкціонованого доступу.

Важливо розуміти, що шифрування захищає конфіденційність даних, але не вирішує проблем контролю доступу для авторизованих користувачів та не гарантує цілісності без додаткових механізмів, таких як цифрові підписи або коди автентифікації повідомлень.

6. ЗАСОБИ ПІДТРИМКИ БЕЗПЕКИ В SQL ТА SQLITE

SQL (Structured Query Language) включає вбудовані механізми управління доступом та безпекою, що дозволяють адміністраторам баз даних контролювати права користувачів на виконання операцій з об'єктами бази даних. Стандарт SQL визначає команди для створення користувачів, надання та відкликання привілеїв, створення ролей.

Команда GRANT – це SQL-інструкція для надання привілеїв користувачам або ролям на виконання певних операцій з об'єктами бази даних. Синтаксис команди дозволяє гнучко визначати, які саме операції дозволені та на які об'єкти. Приклад використання: GRANT SELECT, INSERT ON table_name TO user_name надає користувачу права на вибірку та вставку даних у конкретну таблицю.

Команда REVOKE – це SQL-інструкція для відкликання раніше наданих привілеїв у користувачів або ролей. Відкликання прав є важливою частиною управління доступом, особливо при звільненні працівників або зміні їхніх функціональних обов'язків. Приклад: REVOKE INSERT ON table_name FROM user_name відбирає право на вставку даних.

Типові привілеї в SQL включають:

1. SELECT – право на вибірку даних з таблиці або представлення
2. INSERT – право на вставку нових рядків у таблицю
3. UPDATE – право на зміну існуючих даних
4. DELETE – право на видалення рядків з таблиці
5. REFERENCES – право на створення зовнішніх ключів, що посилаються на таблицю
6. TRIGGER – право на створення тригерів для таблиці
7. EXECUTE – право на виконання збережених процедур та функцій
8. ALL PRIVILEGES – надання всіх доступних привілеїв

SQLite як вбудована СУБД має значні особливості в контексті безпеки, які відрізняють її від клієнт-серверних систем управління базами даних. SQLite не має вбудованої системи управління користувачами та привілеями, оскільки база даних зберігається в одному файлі, а всі операції виконуються в контексті процесу додатку.

Особливості безпеки SQLite включають наступні аспекти:

1. Відсутність вбудованої аутентифікації користувачів та ролей

2. Безпека забезпечується на рівні операційної системи через права доступу до файла бази даних

3. Немає мережевого інтерфейсу, що усуває цілий клас вразливостей, пов'язаних з мережевими атаками

4. Підтримка шифрування всієї бази даних через розширення SQLite Encryption Extension (SEE) або відкриті альтернативи як SQLCipher

5. Захист від SQL-ін'єкцій через використання параметризованих запитів

SQLCipher – це розширення SQLite з відкритим вихідним кодом, що додає прозоре шифрування бази даних з використанням алгоритму AES-256, забезпечуючи захист даних у стані спокою без змін у структурі додатку. SQLCipher шифрує кожну сторінку бази даних перед записом на диск та дешифрує при читанні, роблячи файл бази даних повністю нечитабельним без правильного ключа шифрування.

Параметризовані запити – це техніка написання SQL-запитів, при якій змінні частини запиту передаються як окремі параметри, а не вбудовуються безпосередньо в текст запиту через конкатенацію рядків. Параметризовані запити є найефективнішим засобом захисту від SQL-ін'єкцій, оскільки СУБД чітко розрізняє код запиту та дані користувача.

SQL-ін'єкція – це тип атаки на додатки, що використовують бази даних, при якій зловмисник вставляє або "вприскує" шкідливий SQL-код у вхідні дані додатку, що призводить до виконання несанкціонованих команд у базі даних. SQL-ін'єкції можуть дозволити зловмисникам отримати несанкціонований доступ до даних, модифікувати або видалити інформацію, виконати адміністративні операції.

Приклад вразливого коду (Python з SQLite):

query = "SELECT * FROM users WHERE username = '" + user_input + "'"

Якщо user_input містить значення "admin' OR '1'='1", запит стане: SELECT * FROM users WHERE username = 'admin' OR '1'='1', що поверне всі записи з таблиці.

Безпечний варіант з параметризованим запитом:

query = "SELECT * FROM users WHERE username = ?" cursor.execute(query, (user_input,))

Додаткові заходи безпеки для SQLite включають:

1. Налаштування прав доступу до файла бази даних на рівні операційної системи

2. Розміщення файлу бази даних поза веб-доступною директорією

3. Валідація та санітизація всіх вхідних даних від користувачів

4. Використання ORM (Object-Relational Mapping) фреймворків, які автоматично застосовують параметризовані запити

5. Регулярне резервне копіювання бази даних

6. Обмеження розміру бази даних для запобігання атакам типу відмова в обслуговуванні

7. Ведення журналу операцій для аудита та виявлення аномальної активності

Обмеження цілісності – це правила, визначені на рівні схеми бази даних, що автоматично забезпечують коректність та узгодженість даних шляхом накладення обмежень на допустимі значення атрибутів та взаємозв'язки між таблицями. SQLite підтримує основні типи обмежень цілісності: PRIMARY KEY, FOREIGN KEY, UNIQUE, NOT NULL, CHECK, DEFAULT.

Обмеження зовнішніх ключів у SQLite за замовчуванням вимкнені для зворотної сумісності та повинні бути активовані командою PRAGMA foreign_keys = ON для кожного з'єднання з базою даних. Це обмеження забезпечує референційну цілісність, запобігаючи створенню "осиротілих" записів.

Тригери – це спеціальні збережені процедури, які автоматично виконуються при настанні певних подій у базі даних, таких як вставка, оновлення або видалення даних. Тригери можуть використовуватися для додаткового контролю цілісності даних, ведення журналів змін, автоматичної модифікації пов'язаних даних, реалізації складних бізнес-правил. У контексті безпеки тригери можуть забезпечувати аудит операцій або блокувати операції, що порушують політику безпеки.

Захист і цілісність баз даних є комплексним завданням, що вимагає поєднання технічних, організаційних та процедурних заходів на всіх рівнях інформаційної системи. Ефективна система безпеки базується на принципах багаторівневого захисту, найменших привілеїв та постійного моніторингу, що забезпечують конфіденційність, цілісність та доступність даних навіть в умовах еволюції загроз.

Управління доступом через механізми аутентифікації, авторизації та аудиту дозволяє контролювати дії користувачів та запобігати несанкціонованим операціям. Шифрування даних захищає інформацію від розкриття навіть у разі компрометації фізичних носіїв або мережевих каналів. Адміністрування баз

даних забезпечує належне налаштування систем безпеки, регулярне резервне копіювання та швидке відновлення після інцидентів.

Для SQLite, незважаючи на відсутність вбудованих механізмів управління користувачами, безпека може бути забезпечена через комбінацію засобів операційної системи, шифрування файлів бази даних, використання параметризованих запитів та дотримання принципів безпечного програмування. Розуміння особливостей безпеки конкретної СУБД та застосування відповідних найкращих практик є запорукою створення надійних інформаційних систем, здатних протистояти сучасним загрозам кібербезпеки.

ТЕМА 6. КЛАСИФІКАЦІЯ БАЗ ДАНИХ. ОГЛЯД КЛІЄНТ-СЕРВЕРНИХ ТЕХНОЛОГІЙ

1. «Локальна» архітектура баз даних
2. Архітектура «файл-сервер»
3. Експорт та імпорт таблиць баз даних
4. Архітектура клієнт-серверних СУБД
5. Концепція відкритих систем
6. Відкритий зв’язок з базою даних
7. Технології доступу: ADO, ADO.Net, ODBC, JDBC
8. Транзакції
9. Адміністрування
10. Виконання транзакцій
11. Журналізація
12. Відтік (log shipping / replication context — залежно від курсу)
13. Властивості ACID
14. Проблеми паралелізму
15. Блокування
16. Рівні ізоляції транзакцій
17. Управління транзакціями в мовах програмування
18. Розробка баз даних у середовищі СУБД SQLite
19. Створення таблиць
20. Зміна складу полів
21. Обчислювальні поля
22. Зв’язки між таблицями
23. Використання спеціальних функцій
24. Інструкція SELECT

1. ЛОКАЛЬНА АРХІТЕКТУРА БАЗ ДАНИХ

Еволюція архітектур баз даних відображає розвиток інформаційних технологій від однокористувацьких локальних систем до розподілених багатокористувацьких комплексів. Вибір архітектури бази даних є стратегічним рішенням, що визначає масштабованість, продуктивність, надійність та вартість інформаційної системи. Сучасні організації використовують різноманітні архітектурні підходи залежно від специфіки бізнес-процесів, кількості користувачів, обсягів даних та вимог до доступності. Розуміння переваг та обмежень кожної архітектури, механізмів забезпечення цілісності даних у багатокористувацькому середовищі та технологій доступу до баз даних є необхідним для проектування ефективних інформаційних систем.

Локальна архітектура баз даних – це найпростіша організація системи управління базами даних, при якій база даних, СУБД та прикладна програма розміщуються на одному комп'ютері та обслуговують одного користувача без мережевого доступу. Вся обробка даних відбувається локально на робочій станції користувача, що забезпечує максимальну швидкість доступу за відсутності мережевих затримок.

Характерні особливості локальної архітектури включають:

1. Відсутність мережевого з'єднання та розподілених компонентів системи
2. Пряме звернення додатку до файлу бази даних через файлову систему
3. Виключно однокористувацький режим роботи
4. Максимальна простота розгортання та налаштування
5. Мінімальні апаратні та програмні вимоги
6. Відсутність необхідності в спеціалізованому серверному обладнанні

Переваги локальної архітектури полягають у простоті реалізації, відсутності витрат на мережеву інфраструктуру, високій швидкості доступу до даних, легкості резервного копіювання шляхом простого копіювання файлів, автономності роботи без залежності від мережевого з'єднання. Локальні бази даних ідеально підходять для персональних додатків, невеликих офісних програм, мобільних застосунків, вбудованих систем.

Недоліки локальної архітектури включають неможливість одночасної роботи декількох користувачів, складність централізованого управління даними, проблеми з консолідацією інформації з різних робочих місць, відсутність централізованого резервного копіювання, ризики втрати даних при виході з ладу робочої станції, обмежену масштабованість системи. Ці обмеження роблять локальну архітектуру непридатною для корпоративних систем з множинними користувачами.

SQLite є типовим представником локальних СУБД, що зберігає всю базу даних у одному файлі та працює як вбудована бібліотека в додатку без необхідності у серверному процесі. SQLite широко використовується в мобільних додатках Android та iOS, веб-браузерах, операційних системах та інших програмних продуктах, де потрібна легка вбудована база даних.

2. АРХІТЕКТУРА ФАЙЛ-СЕРВЕР

Архітектура файл-сервер – це організація роботи з базою даних, при якій файл бази розміщується на файловому сервері в локальній мережі, а обробка даних здійснюється на робочих станціях клієнтів, які отримують доступ до файлу через мережеві протоколи обміну файлами. Файловий сервер виконує лише функції зберігання файлів та управління доступом до них, не здійснюючи обробки даних.

Принцип роботи архітектури файл-сервер полягає в тому, що кожна клієнтська робоча станція має встановлену копію СУБД, яка відкриває файл бази даних по мережі та виконує всі операції з даними локально. Коли користувач виконує запит до бази даних, весь необхідний блок даних передається через мережу на робочу станцію, де відбувається його обробка, після чого результат може бути записаний назад у файл на сервері.

Характеристики архітектури файл-сервер:

1. Централізоване зберігання даних на файловому сервері
2. Розподілена обробка даних на клієнтських робочих станціях
3. Можливість одночасної роботи декількох користувачів
4. Високе навантаження на локальну мережу через передачу великих обсягів даних
5. Необхідність механізмів блокування для координації доступу користувачів
6. Складність забезпечення цілісності даних при одночасному доступі

Переваги архітектури файл-сервер включають відносну простоту реалізації порівняно з клієнт-серверною архітектурою, низькі вимоги до серверного обладнання, можливість використання звичайних файлових серверів, централізоване зберігання даних, що спрощує резервне копіювання, підтримку багатокористувацького доступу без необхідності в спеціалізованій серверній СУБД.

Недоліки архітектури файл-сервер є значними та обмежують її застосування в сучасних інформаційних системах. Основні проблеми включають високе навантаження на локальну мережу, оскільки великі обсяги даних

передаються між сервером та клієнтами навіть для виконання простих запитів. Низька продуктивність при збільшенні кількості користувачів через конкуренцію за мережеві ресурси та файлові блокування. Складність забезпечення цілісності даних при паралельних змінах. Обмежені можливості централізованого управління безпекою та аудиту. Підвищений ризик пошкодження даних при збоях мережі або клієнтських станцій. Ці обмеження роблять архітектуру файл-сервер прийнятною лише для невеликих робочих груп з обмеженою інтенсивністю роботи з даними.

Microsoft Access, **FoxPro** та інші настільні СУБД часто використовуються в архітектурі файл-сервер, хоча їхня ефективність швидко знижується при збільшенні кількості одночасних користувачів понад десять осіб.

3. ЕКСПОРТ ТА ІМПОРТ ТАБЛИЦЬ БАЗ ДАНИХ

Експорт даних – це процес вилучення інформації з бази даних та збереження її у файлі певного формату для подальшого використання в інших системах, передачі між додатками, створення резервних копій або обміну даними між різними СУБД. Експорт дозволяє перетворити структуровані дані бази у переносний формат.

Імпорт даних – це процес завантаження інформації із зовнішніх файлів у таблиці бази даних з перетворенням форматів, валідацією даних та встановленням відповідності між структурою джерела та цільової бази. Імпорт є зворотною операцією до експорту та дозволяє інтегрувати дані з різноманітних джерел.

Основні формати обміну даними між базами:

1. CSV (Comma-Separated Values) – текстовий формат, де значення розділені комами або іншими роздільниками, простий у обробці але не підтримує складні типи даних
2. JSON (JavaScript Object Notation) – текстовий формат для структурованих даних, зручний для веб-додатків та API
3. XML (eXtensible Markup Language) – розширюваний формат розмітки, що підтримує ієрархічні структури та метадані
4. SQL-скрипти – набір SQL-команд INSERT для відтворення даних у будь-якій сумісній СУБД
5. Бінарні формати – спеціалізовані формати конкретних СУБД, що забезпечують швидкий обмін з повним збереженням типів даних
6. Excel-файли – популярний формат для обміну табличними даними між СУБД та електронними таблицями

Процес експорту та імпорту вимагає уваги до наступних аспектів:

1. Відповідність типів даних між джерелом та призначенням
2. Обробка спеціальних символів та роздільників у текстових форматах
3. Кодування символів для підтримки багатомовних даних
4. Валідація даних при імпорті для запобігання порушенням цілісності
5. Обробка помилок та конфліктів при імпорті дублікатів
6. Збереження зв'язків між таблицями при експорті реляційних даних

У SQLite експорт та імпорт даних можуть здійснюватися через команди командного рядка. mode csv та. import для CSV-файлів, або програмно через API мов програмування з використанням бібліотек для обробки різних форматів даних.

4. АРХІТЕКТУРА КЛІЄНТ-СЕРВЕРНИХ СУБД

Архітектура клієнт-сервер – це організація роботи інформаційної системи, при якій функції розподілені між клієнтськими додатками, що формують запити та представляють результати користувачам, та серверною СУБД, що виконує обробку запитів, управління даними та забезпечення цілісності. Клієнт-серверна архітектура є домінуючою моделлю для корпоративних систем управління базами даних.

Сервер бази даних – це спеціалізоване програмне забезпечення, що виконується на серверному обладнанні, постійно очікує з'єднань від клієнтів, приймає запити на мові SQL, виконує обробку даних, забезпечує транзакційну цілісність та повертає результати клієнтам. Сервер централізовано керує доступом до даних, оптимізує виконання запитів та забезпечує безпеку.

Клієнт бази даних – це прикладна програма або засіб доступу, що встановлює з'єднання з сервером, формує SQL-запити, передає їх серверу та отримує результати для представлення користувачу або подальшої обробки. Клієнт не має прямого доступу до файлів бази даних та взаємодіє з даними виключно через інтерфейс сервера.

Принципи функціонування клієнт-серверної архітектури:

1. Чітке розмежування функцій між клієнтом та сервером
2. Клієнт відповідає за інтерфейс користувача та бізнес-логіку додатку
3. Сервер відповідає за зберігання даних, обробку запитів та забезпечення цілісності
4. Взаємодія здійснюється через стандартизовані протоколи передачі SQL-запитів
5. Сервер обробляє запити від багатьох клієнтів одночасно

6. Мережею передаються лише запити та результати, а не великі обсяги сирих даних

Переваги клієнт-серверної архітектури включають високу продуктивність за рахунок централізованої обробки на потужному серверному обладнанні, ефективне використання мережі через передачу компактних запитів та результатів замість великих блоків даних, можливість підтримки сотень та тисяч одночасних користувачів, централізоване управління безпекою та правами доступу, забезпечення транзакційної цілісності даних у багатокористувацькому середовищі, централізоване резервне копіювання та адміністрування, масштабованість системи через оновлення серверного обладнання, незалежність розробки клієнтських додатків від деталей зберігання даних.

Недоліки клієнт-серверної архітектури включають вищу складність розгортання та налаштування порівняно з файл-серверною архітектурою, необхідність у спеціалізованому серверному обладнанні та ліцензіях на серверну СУБД, потребу в кваліфікованих адміністраторах баз даних, єдину точку відмови у вигляді сервера бази даних, залежність від мережевого з'єднання для доступу до даних, вищі початкові витрати на впровадження.

MySQL, **PostgreSQL**, **Microsoft SQL Server**, **Oracle Database** є типовими представниками клієнт-серверних СУБД, що забезпечують надійну роботу корпоративних інформаційних систем з великою кількістю користувачів та високими вимогами до продуктивності.

5. КОНЦЕПЦІЯ ВІДКРИТИХ СИСТЕМ ТА ТЕХНОЛОГІЇ ДОСТУПУ ДО БАЗ ДАНИХ

Концепція відкритих систем – це підхід до проектування інформаційних систем, заснований на використанні відкритих стандартів, інтерфейсів та протоколів, що забезпечують сумісність, переносність та взаємодію компонентів від різних виробників без прив'язки до конкретних технологічних платформ. Відкриті системи дозволяють уникнути залежності від одного постачальника та забезпечують гнучкість у виборі технологій.

Переносність додатків – це здатність програмного забезпечення функціонувати на різних апаратних платформах, операційних системах та з різними СУБД без значних змін у коді, що досягається через використання стандартизованих API та абстрагування від специфіки конкретних реалізацій.

Відкритий зв'язок з базою даних (Open Database Connectivity) – це концепція уніфікованого інтерфейсу доступу до різноманітних СУБД, що дозволяє прикладним програмам взаємодіяти з базами даних через

стандартизовані API незалежно від типу та виробника СУБД. Це забезпечує можливість заміни СУБД без переписування додатків.

ODBC (Open Database Connectivity) – це стандартний інтерфейс прикладного програмування для доступу до баз даних, розроблений Microsoft на основі специфікації Call Level Interface консорціуму SQL Access Group, який використовує SQL як мову запитів та забезпечує уніфікований доступ до різних СУБД через драйвери. ODBC став де-факто стандартом для доступу до баз даних у Windows-середовищі.

Архітектура ODBC включає чотири основні компоненти:

1. Додаток – програма, що викликає функції ODBC для виконання SQL-операцій

2. Менеджер драйверів – бібліотека, що завантажує відповідні драйвери та маршрутизує виклики

3. Драйвер СУБД – бібліотека, специфічна для конкретної СУБД, що транслює ODBC-виклики у власний протокол СУБД

4. Джерело даних – конкретна база даних та асоційована з нею конфігурація з'єднання

JDBC (Java Database Connectivity) – це стандартний API для доступу до баз даних із програм на мові Java, що забезпечує платформо-незалежний інтерфейс для виконання SQL-запитів та обробки результатів через драйвери для різних СУБД. JDBC є аналогом ODBC для Java-платформи та широко використовується у корпоративних Java-додатках.

ADO (ActiveX Data Objects) – це об'єктна модель доступу до даних від Microsoft, що надає високорівневий інтерфейс для роботи з різними джерелами даних через технологію OLE DB, спрощуючи розробку додатків баз даних у середовищі Windows та підтримуючи роботу з реляційними та нереляційними джерелами даних.

ADO.NET – це еволюція ADO для платформи .NET Framework, що надає набір класів для роботи з даними в .NET-додатках, підтримує з'єднаний та роз'єднаний режими роботи, оптимізована для веб-застосунків та забезпечує ефективну роботу з XML-даними. ADO.NET використовує провайдери даних для взаємодії з конкретними СУБД.

Основні переваги використання стандартизованих технологій доступу:

1. Незалежність додатків від конкретної СУБД та можливість їх заміни

2. Зменшення складності розробки через уніфіковані API

3. Можливість використання однієї кодової бази для роботи з різними СУБД

4. Широка підтримка інструментів розробки та бібліотек
5. Ефективне використання ресурсів через пули з'єднань
6. Стандартизована обробка помилок та метаданих

6. ТРАНЗАКЦІЇ ТА ВЛАСТИВОСТІ ACID

Транзакція – це логічна одиниця роботи з базою даних, що складається з однієї або декількох операцій, які повинні бути виконані повністю або не виконані взагалі, забезпечуючи перехід бази даних з одного узгодженого стану в інший узгоджений стан. Транзакції є фундаментальним механізмом забезпечення цілісності даних у багатокористувацьких системах.

Властивості ACID – це набір чотирьох фундаментальних характеристик транзакцій, що гарантують надійність обробки даних у СУБД навіть у разі збоїв, помилок та паралельного виконання операцій. Абревіатура ACID утворена від перших літер англійських термінів: Atomicity, Consistency, Isolation, Durability.

Атомарність (Atomicity) – це властивість транзакції бути неподільною одиницею роботи, при якій або всі операції транзакції виконуються успішно та їх результати фіксуються в базі даних, або жодна з операцій не має ефекту, і база даних залишається в стані до початку транзакції. Атомарність унеможливлює часткове виконання транзакції, що могло б призвести до неузгодженого стану даних.

Узгодженість (Consistency) – це властивість транзакції переводити базу даних з одного валідного стану в інший валідний стан з дотриманням усіх визначених обмежень цілісності, бізнес-правил та тригерів. Транзакція не може залишити базу даних у стані, що порушує будь-які обмеження цілісності, навіть якщо проміжні стани під час виконання транзакції тимчасово порушували ці обмеження.

Ізольованість (Isolation) – це властивість, що гарантує ізоляцію паралельно виконуваних транзакцій одна від одної таким чином, що результат їх виконання не залежить від порядку або паралельності виконання, а еквівалентний послідовному виконанню транзакцій. Ізольованість запобігає появі аномалій, коли одна транзакція бачить проміжні незафіксовані зміни іншої транзакції.

Довговічність (Durability) – це властивість, що гарантує збереження результатів зафіксованої транзакції в базі даних навіть у разі подальших збоїв системи, відключень живлення або інших катастрофічних подій. Після успішного завершення транзакції (commit) її зміни повинні бути записані на енергонезалежний носій та не можуть бути втрачені.

Виконання транзакцій у СУБД здійснюється через стандартні SQL-команди управління транзакціями. Транзакція починається командою BEGIN TRANSACTION або неявно з першого SQL-оператора модифікації даних. Завершення транзакції відбувається або командою COMMIT для збереження всіх змін, або командою ROLLBACK для скасування всіх змін та повернення бази даних до стану до початку транзакції.

Приклад транзакції переказу коштів між рахунками:

```
BEGIN TRANSACTION;
UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;
UPDATE accounts SET balance = balance + 100 WHERE account_id = 2;
COMMIT;
```

Якщо будь-яка з операцій UPDATE завершиться помилкою, транзакція буде скасована командою ROLLBACK, що запобіжить неузгодженості, коли гроші зняті з одного рахунку але не додані до іншого.

7. ЖУРНАЛІЗАЦІЯ ТА ВІДНОВЛЕННЯ

Журналізація – це процес реєстрації всіх змін у базі даних в спеціальному журналі транзакцій (transaction log) перед фактичним застосуванням цих змін до основних файлів даних, що забезпечує можливість відновлення бази даних після збоїв та підтримку властивостей атомарності та довговічності транзакцій.

Журнал транзакцій – це послідовний файл, що містить записи про всі операції модифікації даних, включаючи старі та нові значення змінених даних, ідентифікатори транзакцій, часові мітки операцій. Журнал використовується для відновлення даних та скасування незавершених транзакцій після збоїв. Журнал зазвичай зберігається на окремому фізичному диску для підвищення продуктивності та надійності.

Випередження запису в журнал (Write-Ahead Logging, WAL) – це протокол, згідно з яким запис про зміну даних повинен бути записаний у журнал транзакцій перед тим, як сама зміна буде застосована до основних файлів бази даних. Цей принцип гарантує можливість скасування незавершених транзакцій при відновленні після збою.

Контрольна точка (Checkpoint) – це операція періодичного збереження узгодженого стану бази даних, при якій всі зміни з буферів пам'яті записуються на диск, а в журналі фіксується мітка, до якої всі транзакції завершені, що

прискорює відновлення після збою, оскільки не потрібно обробляти весь журнал з самого початку.

Процес відновлення після збою включає дві фази:

1. Фаза повтору (Redo) – відтворення всіх зафіксованих транзакцій з журналу для відновлення змін, які могли не встигнути бути записані на диск до збою

2. Фаза скасування (Undo) – відкат всіх незавершених транзакцій, які почалися але не були зафіксовані до моменту збою, для забезпечення атомарності

SQLite використовує режим журналювання для забезпечення властивостей ACID. За замовчуванням застосовується режим rollback journal, при якому перед зміною даних оригінальні значення зберігаються у файлі журналу, що дозволяє відновити попередній стан при необхідності. Альтернативний режим WAL (Write-Ahead Logging) забезпечує кращу продуктивність та можливість одночасного читання під час запису.

Реплікація – це процес копіювання та синхронізації даних між кількома серверами баз даних для забезпечення відмовостійкості, балансування навантаження та географічного розподілу даних. Реплікація може бути синхронною, коли зміни застосовуються до всіх реплік одночасно, або асинхронною, коли зміни поширюються з затримкою.

9. ДОСТАВКА ЖУРНАЛІВ ТА РЕПЛІКАЦІЯ

Доставка журналів (Log Shipping) – це технологія забезпечення відмовостійкості та створення резервних копій бази даних у реальному часі шляхом періодичного копіювання файлів журналів транзакцій з основного сервера на один або декілька резервних серверів з наступним застосуванням цих журналів до копій бази даних. Доставка журналів створює затримані копії бази даних, які можуть бути активовані у разі відмови основного сервера.

Принцип роботи доставки журналів включає наступні етапи. На основному сервері періодично створюються резервні копії журналів транзакцій, які містять всі зміни даних з моменту попереднього копіювання. Ці файли журналів автоматично копіюються через мережу на резервні сервери. На резервних серверах журнали автоматично застосовуються до копій бази даних, що підтримує їх у майже актуальному стані з певною затримкою. У разі відмови основного сервера резервна база даних може бути переведена в робочий режим з мінімальною втратою даних.

Переваги доставки журналів включають простоту реалізації та налаштування порівняно з складнішими технологіями реплікації, низьке навантаження на основний сервер через періодичне, а не постійне копіювання, можливість підтримки кількох резервних серверів для різних цілей (відмовостійкість, звітність, географічне розподілення), гнучкість у налаштуванні періодичності доставки журналів, можливість затримки застосування журналів на резервних серверах для захисту від логічних помилок або шкідливих дій.

Недоліки доставки журналів включають затримку між основним та резервними серверами, яка може становити від хвилин до годин залежно від налаштувань, відсутність автоматичного перемикання на резервний сервер при відмові основного (потрібне ручне втручання або додаткові інструменти), неможливість використання резервних баз даних для обробки запитів користувачів під час застосування журналів, обмежена деталізація управління порівняно з транзакційною реплікацією.

Реплікація – це технологія автоматичного копіювання та синхронізації даних між кількома серверами баз даних для забезпечення відмовостійкості, підвищення продуктивності через розподілення навантаження, географічного розподілення даних ближче до користувачів та створення аналітичних копій без навантаження на операційні системи. Реплікація забезпечує більш тісну синхронізацію порівняно з доставкою журналів.

Типи реплікації за методом синхронізації включають синхронну та асинхронну реплікацію. При синхронній реплікації транзакція вважається завершеною лише після підтвердження запису на всіх репліках, що гарантує ідентичність даних на всіх серверах але може знижувати продуктивність через очікування підтверджень. При асинхронній реплікації транзакція завершується на основному сервері негайно, а зміни поширюються на репліки з певною затримкою, що забезпечує кращу продуктивність але допускає тимчасову розбіжність даних.

Топології реплікації визначають напрямки потоків даних між серверами:

1. Реплікація master-slave (основний-підлеглий) – один основний сервер приймає зміни, які реплікуються на один або декілька підлеглих серверів лише для читання

2. Реплікація master-master (основний-основний) – кілька серверів можуть приймати зміни, які реплікуються між усіма серверами, що підвищує доступність але ускладнює вирішення конфліктів

3. Каскадна реплікація – зміни поширюються через ланцюг серверів, що зменшує навантаження на основний сервер

4. Кільцева топологія – кожен сервер реплікує зміни на наступний сервер у кільці

Рівні реплікації визначають деталізацію об'єктів, що реплікуються. Реплікація на рівні бази даних копіює всю базу даних цілком. Реплікація на рівні таблиць дозволяє вибірково реплікувати окремі таблиці. Реплікація на рівні рядків або стовпців забезпечує найбільш детальний контроль над тим, які дані реплікуються, що корисно для дотримання вимог безпеки та зменшення обсягів переданих даних.

Методи реплікації включають різні технічні підходи до копіювання даних:

1. Реплікація на основі журналів транзакцій – аналізує журнал транзакцій основного сервера та відтворює зміни на репліках, забезпечуючи точну синхронізацію

2. Реплікація на основі тригерів – використовує тригери бази даних для перехоплення змін та їх відправлення на репліки

3. Реплікація знімків – періодично копіює повні знімки таблиць на репліки, придатна для даних, що рідко змінюються

4. Об'єднана реплікація – комбінує різні методи залежно від характеристик таблиць

Конфлікти реплікації виникають у топологіях master-master, коли різні сервери одночасно модифікують одні й ті самі дані. Стратегії вирішення конфліктів включають пріоритет часової мітки (перемагає остання зміна), пріоритет сервера (зміни з певного сервера мають перевагу), користувацькі бізнес-правила для прийняття рішень, ручне вирішення конфліктів адміністратором. Правильний дизайн додатку може мінімізувати ймовірність конфліктів через розподілення даних таким чином, щоб різні сервери відповідали за різні підмножини даних.

Моніторинг реплікації є критично важливим для забезпечення актуальності даних на всіх репліках. Адміністратори відстежують затримку реплікації (час між зміною на основному сервері та її появою на репліці), помилки реплікації, які можуть зупинити синхронізацію, стан з'єднань між серверами реплікації, обсяг даних, що очікують на реплікацію, споживання мережевої пропускної здатності процесами реплікації.

Застосування реплікації у практичних сценаріях включає створення реплік для звітності та аналітики, що дозволяє виконувати складні запити без

впливу на продуктивність операційної бази даних. Географічне розподілення даних розміщує репліки ближче до регіональних користувачів для зменшення затримок доступу. Забезпечення високої доступності через автоматичне перемикання на репліку при відмові основного сервера. Балансування навантаження розподіляє запити на читання між кількома репліками для підвищення загальної пропускної здатності системи.

SQLite та реплікація – оскільки SQLite є вбудованою СУБД без серверного компонента, традиційні механізми реплікації не застосовні. Проте існують альтернативні підходи для синхронізації SQLite баз даних між пристроями, включаючи використання сторонніх інструментів для синхронізації файлів баз даних, розробку власних механізмів відстеження змін та їх поширення через API, використання хмарних сервісів для синхронізації мобільних додатків, застосування SQLite у комбінації з клієнт-серверною СУБД, де SQLite служить локальним кешем з періодичною синхронізацією з центральним сервером.

Гібридні архітектури поєднують різні підходи до забезпечення доступності даних. Наприклад, система може використовувати синхронну реплікацію між серверами в одному дата-центрі для забезпечення негайної відмовостійкості та асинхронну реплікацію на географічно віддалені сервери для захисту від регіональних катастроф. Комбінація доставки журналів для створення резервних копій та реплікації для розподілення навантаження забезпечує комплексну стратегію захисту даних та підвищення продуктивності.

8. АДМІНІСТРУВАННЯ КЛІЄНТ-СЕРВЕРНИХ БАЗ ДАНИХ

Адміністрування клієнт-серверних баз даних – це комплекс діяльності з управління, налаштування, моніторингу та оптимізації роботи серверів баз даних у багатокористувацькому середовищі, що включає забезпечення безперервної роботи системи, планування ресурсів, управління користувачами та підтримку продуктивності на належному рівні. На відміну від адміністрування локальних баз даних, робота з клієнт-серверними системами вимагає глибокого розуміння мережевих технологій, механізмів паралельної обробки запитів та розподілу навантаження.

Ролі та відповідальність адміністратора у клієнт-серверному середовищі значно ширші порівняно з локальними системами. Адміністратор клієнт-серверної СУБД відповідає не лише за структуру даних, але й за налаштування серверного програмного забезпечення, розподіл системних ресурсів між конкуруючими процесами, управління пулами з'єднань клієнтів,

налаштування кешування та буферизації, конфігурацію мережевих параметрів для оптимальної взаємодії клієнтів та сервера.

Основні завдання адміністрування клієнт-серверних БД:

1. Установка та конфігурація серверного програмного забезпечення СУБД
2. Планування та створення структури бази даних відповідно до вимог додатків
3. Управління обліковими записами користувачів та їхніми привілеями на сервері
4. Моніторинг продуктивності сервера та виявлення вузьких місць
5. Оптимізація конфігураційних параметрів сервера для максимальної ефективності
6. Планування та виконання стратегії резервного копіювання та відновлення
7. Управління дисковим простором та файлами бази даних
8. Налаштування реплікації та відмовостійких конфігурацій
9. Застосування оновлень безпеки та патчів СУБД
10. Аудит безпеки та аналіз журналів доступу

Моніторинг продуктивності є критично важливим аспектом адміністрування, що дозволяє виявляти проблеми до того, як вони вплинуть на користувачів. Адміністратори відстежують ключові метрики продуктивності, включаючи завантаження процесора сервера, використання оперативної пам'яті, швидкість операцій вводу-виводу на дисках, кількість активних з'єднань клієнтів, час виконання запитів, розмір черг на обробку запитів, частоту виникнення блокувань та взаємних блокувань, розмір кешів та коефіцієнт їх ефективності.

Налаштування параметрів сервера включає конфігурацію численних параметрів, що впливають на продуктивність та поведінку СУБД. Ключові параметри включають розмір буферного пулу пам'яті для кешування даних та індексів, кількість робочих потоків для обробки запитів клієнтів, максимальну кількість одночасних з'єднань, таймаути з'єднань та запитів, параметри журналювання транзакцій, налаштування механізму блокування, розміри сегментів пам'яті для сортування та хешування, конфігурацію планувальника запитів.

Управління з'єднаннями у клієнт-серверній архітектурі вимагає балансування між забезпеченням достатньої кількості з'єднань для всіх клієнтів та обмеженням надмірного споживання ресурсів. Кожне з'єднання споживає пам'ять сервера та системні ресурси, тому необхідно встановлювати оптимальні

ліміти. Багато додатків використовують пули з'єднань, де заздалегідь створені з'єднання використовуються повторно замість створення нових з'єднань для кожного запиту, що значно підвищує продуктивність.

Оптимізація запитів є спільною відповідальністю адміністраторів баз даних та розробників додатків. Адміністратори аналізують плани виконання повільних запитів, виявляють відсутність необхідних індексів, перевіряють актуальність статистики таблиць, що використовується оптимізатором запитів, налаштовують параметри оптимізатора, створюють та підтримують індекси, що прискорюють доступ до даних. Інструменти профілювання запитів дозволяють виявити найбільш ресурсомісткі операції та сконцентрувати зусилля на їх оптимізації.

Управління індексами включає створення індексів на стовпцях, що часто використовуються в умовах WHERE, JOIN та ORDER BY, періодичну перебудову фрагментованих індексів для підтримки їх ефективності, видалення надлишкових індексів, що уповільнюють операції модифікації даних без суттєвого прискорення читання, балансування між кількістю індексів та швидкістю операцій INSERT, UPDATE, DELETE.

Планування обслуговування передбачає створення регламентних процедур для підтримки здоров'я бази даних, включаючи оновлення статистики таблиць для оптимізатора запитів, перебудову або реорганізацію фрагментованих індексів, стиснення таблиць для вивільнення простору, перевірку цілісності бази даних для виявлення пошкоджень, очищення журналів транзакцій та тимчасових файлів, архівування старих даних. Ці завдання зазвичай плануються на періоди низького навантаження системи.

Масштабування клієнт-серверних систем може здійснюватися вертикально (збільшення потужності сервера) або горизонтально (додавання додаткових серверів). Вертикальне масштабування включає додавання процесорів, пам'яті, швидших дисків до існуючого сервера. Горизонтальне масштабування використовує розподілення даних між кількома серверами (шардинг), реплікацію для розподілення навантаження читання, кластеризацію для забезпечення відмовостійкості та балансування навантаження.

9. ПРОБЛЕМИ ПАРАЛЕЛІЗМУ ТА БЛОКУВАННЯ

Проблеми паралелізму – це аномалії та неузгодженості даних, що можуть виникати при одночасному виконанні декількох транзакцій, які звертаються до одних і тих же даних, коли транзакції не повністю ізольовані одна від одної.

Правильне управління паралельним доступом критично важливе для забезпечення цілісності даних у багатокористувацьких системах.

Втрачене оновлення (Lost Update) – це ситуація, коли дві транзакції читають одне й те саме значення, потім обидві модифікують його та записують назад, в результаті чого зміни однієї транзакції втрачаються, будучи перезаписаними іншою транзакцією. Приклад: два оператори одночасно резервують останнє вільне місце в готелі, обидва бачать статус "вільно", обидва змінюють на "зайнято", але одне бронювання губиться.

Брудне читання (Dirty Read) – це ситуація, коли транзакція читає дані, змінені іншою незавершеною транзакцією, яка згодом може бути скасована, в результаті чого перша транзакція працює з даними, які ніколи не існували в узгодженому стані бази даних. Приклад: транзакція А збільшує баланс рахунку, транзакція Б читає новий баланс, потім транзакція А скасовується, але Б вже використала неіснуючі дані.

Неповторюване читання (Non-Repeatable Read) – це ситуація, коли транзакція двічі читає одні й ті самі дані в межах однієї транзакції та отримує різні значення, оскільки між читаннями інша транзакція змінила ці дані та зафіксувала зміни. Приклад: банківський звіт читає баланс рахунку на початку та в кінці дня, але між читаннями відбулися операції, що змінили баланс.

Фантомне читання (Phantom Read) – це ситуація, коли транзакція виконує запит з певним критерієм відбору двічі та отримує різну кількість рядків, оскільки інша транзакція вставила або видалила рядки, що задовольняють критерію, між виконаннями запитів. Приклад: запит "скільки клієнтів з Києва" виконується двічі в одній транзакції та дає різні результати.

Блокування – це механізм координації доступу декількох транзакцій до спільних даних шляхом тимчасового надання одній транзакції ексклюзивного або спільного права доступу до об'єкта, запобігаючи конфліктуючим операціям інших транзакцій до завершення першої транзакції.

Спільне блокування (Shared Lock, S-Lock) – це тип блокування, що дозволяє декільком транзакціям одночасно читати об'єкт, але забороняє будь-якій транзакції модифікувати його, доки існує хо ча б одне спільне блокування. Спільні блокування сумісні один з одним але несумісні з ексклюзивними блокуваннями.

Ексклюзивне блокування (Exclusive Lock, X-Lock) – це тип блокування, що надає транзакції виключне право на читання та модифікацію об'єкта, блокуючи доступ усіх інших транзакцій до цього об'єкта до зняття блокування. Ексклюзивні блокування несумісні з будь-якими іншими блокуваннями.

Взаємне блокування (Deadlock) – це ситуація, коли дві або більше транзакцій взаємно очікують звільнення ресурсів, заблокованих одна одною, утворюючи циклічну залежність, яка робить неможливим продовження виконання жодної з транзакцій без втручання СУБД. Приклад: транзакція А блокує ресурс X і очікує ресурс Y, а транзакція Б блокує ресурс Y і очікує ресурс X.

СУБД виявляє взаємні блокування та розв'язує їх шляхом примусового скасування однієї з транзакцій, що дозволяє іншим продовжити виконання. Скасована транзакція зазвичай автоматично перезапускається додатком.

10. РІВНІ ІЗОЛЯЦІЇ ТРАНЗАКЦІЙ

Рівні ізоляції транзакцій – це набір стандартизованих режимів, що визначають ступінь ізоляції транзакцій одна від одної, балансуючи між забезпеченням узгодженості даних та продуктивністю системи шляхом дозволу певних аномалій паралелізму в обмін на зменшення блокувань. SQL-стандарт визначає чотири рівні ізоляції.

READ UNCOMMITTED – найнижчий рівень ізоляції, при якому транзакція може читати незафіксовані зміни інших транзакцій, що дозволяє всі типи аномалій паралелізму включно з брудним читанням, неповторюваним читанням та фантомами, але забезпечує максимальну продуктивність через мінімальні блокування. Цей рівень рідко використовується через високі ризики неузгодженості даних.

READ COMMITTED – рівень ізоляції, при якому транзакція бачить лише зафіксовані зміни інших транзакцій, що запобігає брудному читанню, але дозволяє неповторюване читання та фантоми. Це рівень за замовчуванням у багатьох СУБД, що забезпечує прийнятний баланс між узгодженістю та продуктивністю. Спільні блокування утримуються лише на час читання рядка.

REPEATABLE READ – рівень ізоляції, при якому гарантується, що якщо транзакція прочитала значення, повторне читання того самого значення в межах цієї транзакції поверне той самий результат, що запобігає неповторюваному читанню, але дозволяє фантомне читання. Спільні блокування утримуються до кінця транзакції.

SERIALIZABLE – найвищий рівень ізоляції, при якому транзакції виконуються з повною ізоляцією, еквівалентною послідовному виконанню, що запобігає всім аномаліям паралелізму включно з фантомами, але може значно знижувати продуктивність через інтенсивне блокування та збільшення ймовірності взаємних блокувань.

Співвідношення рівнів ізоляції та дозволених аномалій:

1. READ UNCOMMITTED: дозволяє брудне читання, неповторюване читання, фантоми

2. READ COMMITTED: запобігає брудному читанню; дозволяє неповторюване читання, фантоми

3. REPEATABLE READ: запобігає брудному та неповторюваному читанню; дозволяє фантоми

4. SERIALIZABLE: запобігає всім аномаліям

Управління транзакціями в мовах програмування здійснюється через API відповідних бібліотек доступу до баз даних. Типовий сценарій включає отримання з'єднання з базою даних, встановлення рівня ізоляції, початок транзакції, виконання операцій, фіксацію або відкат транзакції, обробку помилок.

Приклад управління транзакціями в Python з SQLite:

```
import sqlite3
conn = sqlite3.connect('database.db')
try:
    conn.execute('BEGIN TRANSACTION')
    conn.execute('UPDATE accounts SET balance = balance -
100 WHERE id = 1')
    conn.execute('UPDATE accounts SET balance = balance +
100 WHERE id = 2')
    conn.commit()
except Exception as e:
    conn.rollback()
    print(f"Transaction failed: {e}")
finally:
    conn.close()
```

SQLite підтримує транзакції з рівнями ізоляції, хоча його модель блокувань відрізняється від традиційних клієнт-серверних СУБД. SQLite використовує блокування на рівні бази даних, а не окремих рядків чи таблиць, що спрощує реалізацію але обмежує паралелізм. У режимі WAL підтримується одночасне читання декількома транзакціями під час запису.

11. РОЗРОБКА БАЗ ДАНИХ У СЕРЕДОВИЩІ СУБД SQLITE

Створення таблиць у SQLite здійснюється командою CREATE TABLE з визначенням імені таблиці, переліку стовпців, їхніх типів даних та обмежень. SQLite має динамічну систему типів, де тип стовпця є рекомендацією, а не

жорстким обмеженням, що надає гнучкість але вимагає уваги до коректності даних.

Приклад створення таблиці:

```
CREATE TABLE employees (
    employee_id INTEGER PRIMARY KEY AUTOINCREMENT,
    first_name TEXT NOT NULL,
    last_name TEXT NOT NULL,
    email TEXT UNIQUE,
    hire_date TEXT DEFAULT CURRENT_DATE,
    salary REAL CHECK(salary > 0),
    department_id INTEGER,
    FOREIGN KEY (department_id) REFERENCES
departments(department_id)
);
```

Класи зберігання SQLite включають п'ять типів: NULL для відсутності значення, INTEGER для цілих чисел, REAL для чисел з плаваючою точкою, TEXT для текстових рядків у кодуванні UTF-8 або UTF-16, BLOB для бінарних даних. Будь-яке значення будь-якого типу може зберігатися в будь-якому стовпці незалежно від оголошеного типу, хоча рекомендується дотримуватися типізації.

Зміна складу полів у SQLite має обмеження через відсутність повної підтримки команди ALTER TABLE. Можливо лише додавання нових стовпців командою ALTER TABLE ADD COLUMN та перейменування таблиці. Для інших змін структури необхідно створити нову таблицю з бажаною структурою, скопіювати дані зі старої таблиці, видалити стару таблицю та перейменувати нову.

Приклад додавання стовпця:

```
ALTER TABLE employees ADD COLUMN phone TEXT;
```

Обчислювальні поля – це стовпці, значення яких обчислюються динамічно на основі інших стовпців або виразів без фізичного зберігання. У SQLite обчислювальні поля можуть бути реалізовані через представлення (views) або, починаючи з версії 3.31.0, через згенеровані стовпці (generated columns), які можуть бути віртуальними або збереженими.

Приклад створення представлення з обчислювальним полем:

```
CREATE VIEW employee_info AS
SELECT employee_id, first_name, last_name,
       first_name || ' ' || last_name AS full_name,
       salary * 12 AS annual_salary
```

```
FROM employees;
```

Зв'язки між таблицями у реляційних базах даних реалізуються через первинні та зовнішні ключі. SQLite підтримує обмеження зовнішніх ключів, але вони за замовчуванням вимкнені та повинні активуватися командою PRAGMA foreign_keys = ON для кожного з'єднання. Зовнішні ключі забезпечують референційну цілісність, запобігаючи створенню записів з посиланнями на неіснуючі рядки.

Типи зв'язків:

1. Один-до-одного – реалізується через зовнішній ключ з обмеженням UNIQUE

2. Один-до-багатьох – реалізується через зовнішній ключ у таблиці на боці "багато"

3. Багато-до-багатьох – реалізується через проміжну таблицю з двома зовнішніми ключами

Використання спеціальних функцій у SQLite розширює можливості обробки даних. SQLite підтримує численні вбудовані функції для роботи з рядками, числами, датами, агрегації даних. Функції можуть використовуватися в SELECT-запитах, умовах WHERE, обчисленнях.

Приклади корисних функцій SQLite:

1. LENGTH(string) – повертає довжину рядка

2. UPPER(string), LOWER(string) – перетворення регістру

3. SUBSTR(string, start, length) – витяг підрядка

4. ROUND(number, decimals) – округлення числа

5. DATE('now'), DATETIME('now') – поточні дата та час

6. COALESCE(value1, value2, ...) – повертає перше не-NULL значення

7. IFNULL(value, default) – повертає default якщо value є NULL

Інструкція SELECT є найважливішою командою SQL для вибірки даних з бази. SELECT дозволяє визначати стовпці для вибірки, таблиці-джерела, умови фільтрації, порядок сортування, групування та агрегацію даних.

Основний синтаксис SELECT:

```
SELECT column1, column2, ...
FROM table_name
WHERE condition
GROUP BY column
HAVING aggregate_condition
ORDER BY column ASC|DESC
LIMIT number OFFSET number;
```

Приклад складного запиту з об'єднанням таблиць та агрегацією:

```
    SELECT d.department_name, COUNT(e.employee_id) AS
employee_count,
           AVG(e.salary) AS average_salary
    FROM departments d
    LEFT JOIN employees e ON d.department_id =
e.department_id
    WHERE e.hire_date > '2020-01-01'
    GROUP BY d.department_name
    HAVING COUNT(e.employee_id) > 5
    ORDER BY average_salary DESC;
```

Об'єднання таблиць (JOIN) дозволяє комбінувати рядки з декількох таблиць на основі пов'язаних стовпців. SQLite підтримує різні типи об'єднань: INNER JOIN (тільки рядки з співпадіннями в обох таблицях), LEFT JOIN (всі рядки з лівої таблиці плюс співпадіння), CROSS JOIN (декартовий добуток).

Адміністрування клієнт-серверних баз даних є комплексною діяльністю, що вимагає глибоких знань не тільки про структуру даних, але й про архітектуру серверів, мережеві технології, механізми оптимізації продуктивності та стратегії забезпечення відмовостійкості. Ефективне адміністрування базується на проактивному моніторингу, регулярному обслуговуванні, оптимізації конфігурацій та швидкому реагуванні на проблеми до того, як вони вплинуть на користувачів системи.

Технології доставки журналів та реплікації є ключовими інструментами для забезпечення високої доступності даних, захисту від втрат інформації та підвищення продуктивності систем через розподілення навантаження. Вибір між доставкою журналів та реплікацією, або їх комбінацією, залежить від вимог до затримки синхронізації, складності інфраструктури, бюджету на обладнання та ліцензії, географічного розподілення користувачів та критичності даних для бізнесу організації.

Розуміння архітектурних підходів від простих локальних систем до складних розподілених клієнт-серверних конфігурацій з реплікацією дозволяє проектувати інформаційні системи, що відповідають специфічним потребам організації, забезпечуючи баланс між продуктивністю, надійністю, масштабованістю та вартістю володіння. Правильний вибір архітектури та технологій на етапі проектування визначає успішність інформаційної системи протягом усього її життєвого циклу.

ТЕМА 7. ЛОГІЧНЕ ТА ФІЗИЧНЕ ПРОЕКТУВАННЯ БАЗ ДАНИХ

1. Рівні моделювання предметної області
2. Поняття предметної області
3. Архітектура ANSI/SPARC
4. Зовнішній рівень архітектури
5. Концептуальний рівень архітектури
6. Внутрішній рівень архітектури
7. Властивості кожного архітектурного рівня
8. Концептуальна модель даних
9. Фізична модель даних
10. Перша нормальна форма (1НФ)
11. Друга нормальна форма (2НФ)
12. Третя нормальна форма (3НФ)
13. Четверта нормальна форма (4НФ)
14. П’ята нормальна форма (5НФ)
15. Проектування бази даних методом «сутність—зв’язок» (ER-метод)
16. Основні поняття про середовище графічної мови ERWin
17. Команди та інструменти ERWin
18. Проектування баз даних засобами ERWin
19. Проектування бази даних у СУБД SQLite

1. РІВНІ МОДЕЛЮВАННЯ ПРЕДМЕТНОЇ ОБЛАСТІ

Моделювання предметної області являє собою процес створення формалізованого опису реального світу або його частини з метою подальшої автоматизації. У процесі проектування баз даних виділяють три основні рівні моделювання, які відображають різні аспекти представлення даних.

Концептуальний рівень моделювання є найбільш абстрактним і орієнтованим на користувача. На цьому рівні створюється **концептуальна модель**, яка описує структуру предметної області в термінах, зрозумілих експертам та користувачам, без прив'язки до конкретної СУБД.

Логічний рівень моделювання передбачає трансформацію концептуальної моделі в **логічну модель даних**, яка враховує особливості обраної моделі даних (реляційної, об'єктної, мережевої тощо), але ще не залежить від конкретної СУБД.

Фізичний рівень моделювання описує, як дані фізично зберігаються в конкретній СУБД, включаючи структури файлів, індекси, методи доступу та інші аспекти реалізації.

2. ПОНЯТТЯ ПРЕДМЕТНОЇ ОБЛАСТІ

Предметна область (domain area) — це частина реального світу, яка підлягає вивченню та автоматизації. Це сукупність об'єктів, процесів, явищ та зв'язків між ними, інформація про які повинна зберігатися та оброблятися в базі даних.

Об'єкти предметної області — це матеріальні чи нематеріальні сутності, які мають значення для функціонування системи. Наприклад, у предметній області університету об'єктами можуть бути студенти, викладачі, дисципліни, аудиторії.

Атрибути об'єктів — це властивості чи характеристики об'єктів, які описують їх стан. Кожен атрибут має **домен** — множину допустимих значень.

Зв'язки між об'єктами відображають взаємодії та залежності між сутностями предметної області. Розрізняють зв'язки типу "один до одного", "один до багатьох" та "багато до багатьох".

Коректне визначення предметної області є критично важливим для успішного проектування бази даних, оскільки помилки на цьому етапі призводять до неадекватної моделі та неефективної роботи системи.

3. АРХІТЕКТУРА ANSI/SPARC

Архітектура ANSI/SPARC (American National Standards Institute / Standards Planning and Requirements Committee) є стандартною трирівневою архітектурою систем баз даних, запропонованою в 1975 році. Ця архітектура забезпечує **незалежність даних** та відокремлення різних аспектів роботи з базою даних.

Трирівнева архітектура складається з:

- **Зовнішнього рівня** (external level) — рівень користувацьких представлень
- **Концептуального рівня** (conceptual level) — рівень логічної структури всієї бази даних
- **Внутрішнього рівня** (internal level) — рівень фізичного зберігання даних

Основною перевагою архітектури ANSI/SPARC є **незалежність даних**, яка поділяється на два типи:

Логічна незалежність даних означає, що зміни на концептуальному рівні не впливають на зовнішні схеми та прикладні програми. Можна додавати нові типи даних, змінювати структуру без перепрограмування додатків.

Фізична незалежність даних означає, що зміни способу фізичного зберігання даних не впливають на концептуальну та зовнішні схеми. Можна змінювати структури файлів, методи індексації без модифікації логічної структури.

Між рівнями існують **відображення** (mappings), які визначають, як дані одного рівня відповідають даним іншого рівня. **Зовнішньо-концептуальне відображення** встановлює відповідність між зовнішніми представленнями та концептуальною схемою, а **концептуально-внутрішнє відображення** — між концептуальною схемою та фізичною структурою.

4. ЗОВНІШНІЙ РІВЕНЬ АРХІТЕКТУРИ

Зовнішній рівень (external level) представляє **користувацькі представлення** даних. На цьому рівні кожен користувач або група користувачів бачить лише ті дані, які їм необхідні для виконання їхніх функцій.

Зовнішня схема (external schema) або **підсхема** (subschema) — це опис частини бази даних з точки зору конкретного користувача або прикладної програми. Одна база даних може мати багато зовнішніх схем.

Представлення (view) — це віртуальна таблиця, яка формується на основі даних з базових таблиць концептуального рівня. Представлення не містять власних даних, а генеруються динамічно при зверненні.

Переваги зовнішнього рівня:

- **Безпека даних** — користувачі не мають доступу до даних, які їм не потрібні
- **Спрощення сприйняття** — користувачі працюють з простішими структурами
- **Незалежність додатків** — зміни в базі даних не впливають на програми, якщо зовнішня схема залишається незмінною

На зовнішньому рівні можуть виконуватися операції обмеження доступу, перейменування атрибутів, агрегування даних з декількох таблиць, обчислення похідних значень.

5. КОНЦЕПТУАЛЬНИЙ РІВЕНЬ АРХІТЕКТУРИ

Концептуальний рівень (conceptual level) описує **глобальну логічну структуру** всієї бази даних для всього співтовариства користувачів. Це центральний рівень архітектури, який забезпечує єдиний узгоджений погляд на дані.

Концептуальна схема (conceptual schema) визначає:

1. Всі типи **сутностей** (entities) та їх **атрибути.**

2. **Зв'язки** між сутностями та їх **кардинальність.**

3. **Обмеження цілісності**, які повинні виконуватися для даних.

4. Семантичну інформацію про дані.

5. **Правила безпеки** та авторизації.

Концептуальна схема не містить деталей про фізичне зберігання даних — це чисто логічний опис. Вона повинна бути стабільною та не змінюватися при модифікації зовнішніх схем або внутрішньої реалізації.

Концептуальне моделювання використовує різні **моделі даних**:

- **реляційна модель** — дані представлені у вигляді таблиць (відношень);
- **об'єктна модель** — дані організовані як об'єкти з методами;
- **об'єктно-реляційна модель** — комбінація обох підходів.

На концептуальному рівні визначаються **обмеження цілісності**:

- **Цілісність сутностей** — кожна сутність повинна мати унікальний ідентифікатор.
- **Посилальна цілісність** — зовнішні ключі повинні відповідати первинним ключам.
- **Доменна цілісність** — значення атрибутів повинні належати визначеним доменам.

6. ВНУТРІШНІЙ РІВЕНЬ АРХІТЕКТУРИ

Внутрішній рівень (internal level) описує, як дані фізично зберігаються в системі. Це найнижчий рівень абстракції, найближчий до фізичного зберігання даних.

Внутрішня схема (internal schema) визначає:

- **структури файлів** для зберігання даних;
- **методи організації даних** (послідовні файли, індексовані файли, хеш-таблиці);
- **типи індексів** (B-дерева, хеш-індекси, bitmap-індекси);
- **розміщення записів** на фізичних носіях;
- **стратегії буферизації** та кешування;
- **методи стиснення даних.**

Фізичні структури даних включають:

- **файли даних** (data files) — містять власне дані таблиць**;**
- **індексні файли** (index files) — прискорюють пошук даних;
- **журнали транзакцій** (transaction logs) — забезпечують відновлення після збоїв;

– **системні каталоги** (system catalogs) — зберігають метадані.

Методи доступу визначають, як СУБД знаходить та отримує дані:

– **послідовний доступ** — читання записів один за одним;

– **прямий доступ** — звернення до запису за його адресою;

– **індексний доступ** — використання індексів для швидкого пошуку.

Оптимізація на внутрішньому рівні критично важлива для продуктивності системи, але повністю прихована від користувачів та розробників додатків завдяки незалежності даних.

7. ВЛАСТИВОСТІ КОЖНОГО АРХІТЕКТУРНОГО РІВНЯ

Кожен рівень архітектури ANSI/SPARC має специфічні властивості та характеристики, які визначають його роль у загальній системі.

Властивості зовнішнього рівня:

– **множинність** — може існувати багато зовнішніх схем для однієї бази даних;

– **персоналізація** — кожна схема адаптована до потреб конкретної групи користувачів;

– **динамічність** — представлення формуються під час виконання запитів;

– **віртуальність** — зовнішні схеми не містять реальних даних;

– **обмеженість** — користувач бачить лише дозволену частину даних.

Властивості концептуального рівня:

– **єдність** — існує лише одна концептуальна схема для всієї бази даних;

– **повнота** — описує всі дані, що зберігаються в системі;

– **інтеграція** — забезпечує узгоджене представлення даних;

– **незалежність від реалізації** — не містить деталей фізичного зберігання;

– **стабільність** — рідко змінюється порівняно з іншими рівнями.

Властивості внутрішнього рівня:

– **конкретність** — описує реальне фізичне зберігання;

– **СУБД-специфічність** — залежить від конкретної системи управління БД;

– **оптимізація** — орієнтований на ефективність роботи;

– **динамічна адаптація** — може змінюватися для покращення продуктивності;

– **прихованість** — повністю прихований від кінцевих користувачів.

Відображення між рівнями забезпечують трансформацію даних та запитів:

– **зовнішньо-концептуальне відображення** перетворює запити користувача до концептуальної схеми;

– **концептуально-внутрішнє відображення** трансформує логічні операції у фізичні операції доступу до даних.

8. КОНЦЕПТУАЛЬНА МОДЕЛЬ ДАНИХ

Концептуальна модель даних (conceptual data model) — це абстрактне, незалежне від реалізації представлення структури даних предметної області. Вона описує, які дані повинні зберігатися в системі та як вони пов'язані між собою.

Основні елементи концептуальної моделі:

- сутність (entity) — це об'єкт реального світу, який має самостійне існування та про який необхідно зберігати інформацію. Сутності можуть бути матеріальними (студент, книга) або абстрактними (курс, замовлення);

- атрибут (attribute) — це властивість або характеристика сутності. Атрибути поділяються на:

– **прості атрибути** — неподільні (ім'я, вік);

– **складені атрибути** — можуть бути розділені на компоненти (адреса = вулиця + місто + індекс);

– **однозначні атрибути** — мають одне значення для екземпляра сутності;

– **багатозначні атрибути** — можуть мати декілька значень (телефони);

– **похідні атрибути** — обчислюються з інших атрибутів (вік з дати народження).

Ключ (key) — це атрибут або комбінація атрибутів, які унікально ідентифікують екземпляр сутності:

– **Первинний ключ** (primary key) — обраний для унікальної ідентифікації.

– **Альтернативний ключ** (alternate key) — інші можливі унікальні ідентифікатори.

– **Складений ключ** (composite key) — складається з декількох атрибутів.

Зв'язки (relationships) визначають асоціації між сутностями:

– **один до одного** (1:1) — один екземпляр сутності A пов'язаний з одним екземпляром сутності B;

– **один до багатьох** (1:N) — один екземпляр сутності A пов'язаний з багатьма екземплярами сутності B;

– **багато до багатьох** (M:N) — багато екземплярів сутності A пов'язані з багатьма екземплярами сутності B.

Кардинальність зв'язку (cardinality) визначає максимальну кількість екземплярів, які можуть брати участь у зв'язку. **Модальність** (optionality) визначає обов'язковість участі у зв'язку.

Концептуальна модель повинна бути зрозумілою для експертів предметної області та служити основою для створення логічної моделі.

9. ФІЗИЧНА МОДЕЛЬ ДАНИХ

Фізична модель даних (physical data model) описує, як концептуальна та логічна моделі реалізуються в конкретній СУБД з урахуванням особливостей фізичного зберігання та продуктивності.

Компоненти фізичної моделі:

Фізичні таблиці включають:

- Визначення **типів даних** для кожної колонки (INTEGER, VARCHAR, DATE тощо);
- Розміри полів та точність числових значень;
- Правила допустимості NULL-значень;
- **Значення за замовчуванням** (default values).

Первинні ключі (primary keys) реалізуються як **унікальні індекси**, які забезпечують швидкий доступ та унікальність значень.

Зовнішні ключі (foreign keys) реалізують **посилальну цілісність** та визначають **правила каскадних операцій**:

- CASCADE — автоматичне видалення або оновлення пов'язаних записів;
- SET NULL — встановлення NULL у зовнішньому ключі;
- RESTRICT — заборона операції при наявності пов'язаних записів.

Індекси (indexes) — це структури даних, які прискорюють пошук:

- **B-дерево індекси** (B-tree) — універсальні, підтримують діапазонні запити;
- **Хеш-індекси** (hash) — дуже швидкі для точного пошуку;
- **Bitmap-індекси** — ефективні для колонок з низькою кардинальністю;
- **Повнотекстові індекси** (full-text) — для пошуку в текстових полях.

Секціонування таблиць (partitioning) — розділення великих таблиць на менші частини для покращення продуктивності:

- **горизонтальне секціонування** — розділення за рядками (наприклад, по датах);
- **вертикальне секціонування** — розділення за колонками.

Кластеризація (clustering) — групування записів, які часто використовуються разом, для зменшення операцій введення-виведення.

Денормалізація — свідоме порушення нормальних форм для підвищення продуктивності читання даних шляхом зменшення кількості з'єднань таблиць.

Стратегії зберігання:

– розміщення файлів на різних фізичних дисках;

– використання SSD для критичних індексів;

– налаштування розмірів буферів та кешів.

10. ПЕРША НОРМАЛЬНА ФОРМА (1НФ)

Нормалізація (normalization) — це процес організації даних у базі даних для зменшення надмірності та покращення цілісності даних. Процес нормалізації включає послідовне приведення таблиць до **нормальних форм**.

Перша нормальна форма (1НФ) (First Normal Form, 1NF) вимагає, щоб всі атрибути таблиці містили лише **атомарні** (неподільні) значення, і щоб не було повторюваних груп атрибутів.

Вимоги 1НФ:

– кожен атрибут повинен містити лише **атомарне значення** — неподільну одиницю даних;

– у таблиці не повинно бути **повторюваних груп** атрибутів;

– кожен рядок повинен бути **унікальним;**

– порядок рядків та колонок не має значення.

Приклад порушення 1НФ:

Таблиця "Студенти" з колонкою "Телефони", яка містить декілька номерів телефонів через кому:

СтудентID	Ім'я	Телефони
1	Іван	050-123-45-67, 067-890-12-34

Це порушує 1НФ, оскільки атрибут "Телефони" містить множинні значення.

Приведення до 1НФ:

Варіант 1: Створення окремих рядків для кожного телефону:

СтудентID	Ім'я	Телефон
1	Іван	050-123-45-67
1	Іван	067-890-12-34

Варіант 2: Створення окремої таблиці "Телефони":

Таблиця "Студенти"

СтудентID	Ім'я
1	Іван

Таблиця "Телефони"

СтудентID	Телефон
1	050-123-45-67
1	067-890-12-34

Переваги 1НФ:

1. Спрощує обробку даних
2. Забезпечує коректну роботу реляційних операцій
3. Усуває неоднозначність інтерпретації даних

Приведення до 1НФ є обов'язковим першим кроком нормалізації та базовою вимогою для реляційної моделі даних.

11. ДРУГА НОРМАЛЬНА ФОРМА (2НФ)

Друга нормальна форма (2НФ) (Second Normal Form, 2NF) вимагає, щоб таблиця відповідала 1НФ та щоб кожен **неключовий атрибут** був **повністю функціонально залежним** від усього первинного ключа, а не від його частини.

Функціональна залежність — це зв'язок між атрибутами, за якого значення одного атрибута (детермінанта) однозначно визначає значення іншого атрибута. Позначається: A → B (B функціонально залежить від A).

Повна функціональна залежність означає, що атрибут залежить від усього складеного ключа, а не від його частини. **Часткова функціональна залежність** виникає, коли атрибут залежить лише від частини складеного ключа.

Вимоги 2НФ:

– Таблиця повинна бути в 1НФ

– Якщо первинний ключ складається з одного атрибута, таблиця автоматично в 2НФ

– Якщо первинний ключ **складений**, всі неключові атрибути повинні залежати від усього ключа

Приклад порушення 2НФ:

Таблиця "Замовлення_Товари" з первинним ключем (ЗамовленняID, ТоварID):

ЗамовленняID	ТоварID	Кількість	ДатаЗамовлення	НазваТовару	Ціна
1	101	5	2024-01-15	Ноутбук	25000
1	102	2	2024-01-15	Миша	500

Порушення: "ДатаЗамовлення" залежить лише від "ЗамовленняID", а "НазваТовару" та "Ціна" залежать лише від "ТоварID" — це часткові залежності.

Приведення до 2НФ:

Розділяємо на три таблиці:

Таблиця "Замовлення":

ЗамовленняID	ДатаЗамовлення
1	2024-01-15

Таблиця "Товари":

ТоварID	НазваТовару	Ціна
101	Ноутбук	25000
102	Миша	500

Таблиця "Замовлення_Товари":

ЗамовленняID	ТоварID	Кількість
1	101	5
1	102	2

Переваги 2НФ:

– усунення **аномалій оновлення** — зміна назви товару не вимагає оновлення багатьох рядків;

– усунення **аномалій видалення** — видалення останнього замовлення товару не призводить до втрати інформації про товар;

– усунення **аномалій вставки** — можна додати товар без створення замовлення;

– зменшення надмірності даних.

12. ТРЕТЯ НОРМАЛЬНА ФОРМА (3НФ)

Третя нормальна форма (3НФ) (Third Normal Form, 3NF) вимагає, щоб таблиця відповідала 2НФ та щоб не було **транзитивних функціональних залежностей** неключових атрибутів від первинного ключа.

Транзитивна залежність виникає, коли атрибут A визначає атрибут B, а B визначає атрибут C, тоді C транзитивно залежить від A (A → B → C).

Вимоги 3НФ:

– Таблиця повинна бути в 2НФ.

– Всі неключові атрибути повинні залежати **безпосередньо** від первинного ключа.

– Не повинно бути залежностей неключових атрибутів один від одного.

Приклад порушення 3НФ:

Таблиця "Співробітники"

СпівробітникID	Ім'я	ВідділID	НазваВідділу	МенеджерВідділу
1	Петро	10	IT	Олена
2	Марія	10	IT	Олена
3	Іван	20	HR	Андрій

Порушення: "НазваВідділу" та "МенеджерВідділу" залежать від "ВідділID", а не безпосередньо від "СпівробітникID". Маємо транзитивну залежність: СпівробітникID → ВідділID → НазваВідділу.

Приведення до 3НФ:

Розділяємо на дві таблиці:

Таблиця "Співробітники":

СпівробітникID	Ім'я	ВідділID
1	Петро	10
2	Марія	10
3	Іван	20

Таблиця "Відділи":

ВідділID	НазваВідділу	МенеджерВідділу
10	IT	Олена
20	HR	Андрій

Переваги 3НФ:

– Подальше зменшення надмірності даних.

– Усунення аномалій: зміна менеджера відділу вимагає оновлення лише одного рядка.

– Можна додати новий відділ без наявності співробітників.

– Видалення всіх співробітників відділу не призводить до втрати інформації про відділ.

Нормальна форма Бойса-Кодда (НФБК) є посиленою версією 3НФ, яка вимагає, щоб кожен **детермінант** був **кандидатом на ключ**. НФБК усуває деякі аномалії, які можуть залишатися в 3НФ у випадку множинних перекриваючих кандидатів на ключ.

13. ЧЕТВЕРТА НОРМАЛЬНА ФОРМА (4НФ)

Четверта нормальна форма (4НФ) (Fourth Normal Form, 4NF) вирішує проблеми, пов'язані з **багатозначними залежностями** (multivalued dependencies). Таблиця відповідає 4НФ, якщо вона в НФБК та не містить нетривіальних багатозначних залежностей.

Багатозначна залежність виникає, коли в таблиці один атрибут визначає множину значень іншого атрибута, незалежно від інших атрибутів таблиці. Позначається: A →→ B (B багатозначно залежить від A).

Приклад порушення 4НФ:

Таблиця "Викладачі_Курси_Підручники"

ВикладачID	Курс	Підручник
1	Бази даних	Дейт "SQL"
1	Бази даних	Коннолі "Databases"
1	Програмування	Страуструп "C++"
1	Програмування	Дейт "SQL"
1	Програмування	Коннолі "Databases"

Порушення: викладач може викладати декілька курсів та рекомендувати декілька підручників, але ці множини незалежні одна від одної. Виникають **незалежні багатозначні залежності**: ВикладачID →→ Курс та ВикладачID →→ Підручник.

Це призводить до надмірності: якщо викладач рекомендує 3 підручники та викладає 4 курси, буде 12 рядків замість 7 (3+4).

Приведення до 4НФ:

Розділяємо на дві таблиці:

Таблиця 1. Викладачі_Курси

ВикладачID	Курс
1	Бази даних
1	Програмування

Таблиця 2. Викладачі_Підручники

ВикладачID	Підручник
1	Дейт "SQL"
1	Коннолі "Databases"
1	Страуструп "C++"

Переваги 4НФ:

- Усунення надмірності, спричиненої багатозначними залежностями
- Усунення аномалій оновлення при зміні незалежних множин
- Зменшення кількості рядків у таблицях
- Спрощення підтримки цілісності даних

Важливо: 4НФ застосовується в ситуаціях, коли є множинні незалежні зв'язки типу "багато-до-багатьох" від одного атрибута. У більшості практичних випадків достатньо 3НФ або НФБК.

14. П'ЯТА НОРМАЛЬНА ФОРМА (5НФ)

П'ята нормальна форма (5НФ) або проекційно-з'єднувальна нормальна форма (Project-Join Normal Form, PJNF) є найвищим рівнем нормалізації у класичній теорії нормалізації. Таблиця відповідає 5НФ, якщо вона в 4НФ та не

містить залежностей з'єднання (join dependencies), які не є наслідком кандидатів на ключ.

Залежність з'єднання виникає, коли таблиця може бути розкладена на декілька таблиць без втрати інформації, але не може бути відтворена з'єднанням будь-якої пари цих таблиць — потрібне з'єднання всіх таблиць одночасно.

Приклад порушення 5НФ:

Розглянемо таблицю "Постачальники_Товари_Проекти", яка описує, які постачальники можуть надавати які товари для яких проектів:

Постачальник	Товар	Проект
Компанія_А	Цемент	Проект_1
Компанія_А	Цемент	Проект_2
Компанія_В	Цегла	Проект_1
Компанія_А	Цегла	Проект_1

Чому це порушує 5НФ:

– таблиця містить **комбінації трьох сутностей** (Постачальник, Товар, Проект).

– неможливо розкласти її на менші таблиці (наприклад, “Постачальник-Товар”, “Постачальник-Проект”, “Товар-Проект”) **без втрати інформації** про точні комбінації.

– виникає **надлишковість даних**: один постачальник і товар повторюються для кількох проектів.

Можуть існувати специфічні бізнес-правила:

- постачальник може постачати товар;

- товар може використовуватися в проекті;

- постачальник може працювати з проектом.

Але комбінація (Постачальник, Товар, Проект) дійсна лише якщо виконуються всі три умови.

Приведення до 5НФ:

Розділяємо на три таблиці:

Таблиця 1. Постачальники_Товари

Постачальник	Товар
Компанія_А	Цемент
Компанія_А	Цегла
Компанія_В	Цегла

Таблиця 2. Товари_Проекти

Товар	Проект
Цемент	Проект_1

Цемент	Проект_2
Цегла	Проект_1

Таблиця 3. Постачальники_Проекти

Постачальник	Проект
Компанія_А	Проект_1
Компанія_А	Проект_2
Компанія_В	Проект_1

Оригінальна таблиця може бути відтворена тільки природним з'єднанням всіх трьох таблиць одночасно.

Характеристики 5НФ:

- Є теоретичною границею нормалізації.
- Рідко застосовується на практиці через складність.
- Потрібна для уникнення дуже специфічних аномалій.
- Може призвести до надмірного ускладнення структури бази даних.

Практичні рекомендації:

У більшості реальних проектів достатньо нормалізації до 3НФ або НФБК. 4НФ та 5НФ застосовуються лише у виняткових випадках, коли бізнес-логіка вимагає представлення складних багатосторонніх зв'язків. Часто на практиці виконують денормалізацію для підвищення продуктивності, свідомо повертаючись до нижчих нормальних форм.

15. ПРОЕКТУВАННЯ БАЗИ ДАНИХ МЕТОДОМ «СУТНІСТЬ—ЗВ'ЯЗОК» (ER-МЕТОД)

Метод «сутність-зв'язок» (Entity-Relationship method, ER-метод) є одним із найбільш поширених та визнаних підходів до концептуального моделювання структури баз даних. Цей метод був розроблений та вперше представлений американським вченим Пітером Ченом (Peter Chen) у 1976 році в його фундаментальній роботі, присвяченій моделюванню даних для систем управління базами даних. З моменту свого створення ER-метод став стандартом де-факто в індустрії розробки програмного забезпечення та використовується практично в усіх серйозних проектах, пов'язаних з проектуванням інформаційних систем.

Основна ідея методу полягає у використанні графічної нотації для представлення структури даних, що робить модель інтуїтивно зрозумілою як для технічних фахівців, так і для експертів предметної області, які не мають глибоких знань у галузі інформаційних технологій. Ця особливість методу є критично важливою, оскільки дозволяє забезпечити ефективну комунікацію між

різними учасниками проекту та гарантує, що створювана модель адекватно відображає реальні бізнес-процеси та вимоги організації.

Фундаментальні компоненти ER-моделі

ER-модель базується на декількох основних концепціях, які у своїй сукупності дозволяють повністю описати структуру даних предметної області.

Сутності (entities) представляють центральну концепцію ER-моделювання. Сутність являє собою абстракцію об'єкта реального світу, який має самостійне існування та про який необхідно зберігати інформацію в базі даних. Графічно сутності зображуються у вигляді прямокутників, що містять назву сутності. Важливо розуміти, що сутність описує не конкретний екземпляр об'єкта, а клас або тип об'єктів зі спільними характеристиками. Наприклад, сутність "Студент" описує всіх студентів як клас, а не конкретну особу.

Сутності класифікуються за декількома критеріями. З точки зору природи об'єктів розрізняють матеріальні сутності, які відповідають фізичним об'єктам реального світу (наприклад, "Співробітник", "Товар", "Будівля"), та абстрактні сутності, які представляють нематеріальні концепції (наприклад, "Курс", "Замовлення", "Договір"). З точки зору залежності існування виділяють сильні сутності (strong entities), які мають власний унікальний ідентифікатор та можуть існувати незалежно, і слабкі сутності (weak entities), які не мають власного повного ідентифікатора та залежать від існування іншої сутності. Слабкі сутності часто представляють деталі або компоненти інших об'єктів і не мають змісту поза контекстом породжуючої їх сутності.

Атрибути (attributes) описують властивості або характеристики сутностей. Кожен атрибут має назву та домен — множину допустимих значень. У класичній нотації Чена атрибути зображуються у вигляді овалів, з'єднаних лініями з відповідною сутністю. Атрибути класифікуються за декількома ознаками, що відображають їх структуру та семантику.

Прості атрибути (simple attributes) є атомарними, тобто неподільними одиницями даних, які не можуть бути декомпозовані на більш дрібні компоненти, що мають самостійне значення. Приклади простих атрибутів включають ім'я, вік, дату народження. **Складені атрибути** (composite attributes) можуть бути розділені на більш прості складові частини. Класичним прикладом є атрибут "Адреса", який може бути декомпозований на компоненти "Вулиця", "Місто", "Область", "Поштовий індекс", "Країна". Графічно складені атрибути представляються основним овалом з дочірніми овалами для кожного компонента.

Однозначні атрибути (single-valued attributes) можуть мати лише одне значення для конкретного екземпляра сутності в певний момент часу. Наприклад, атрибут "Дата народження" є однозначним, оскільки особа може мати лише одну дату народження. **Багатозначні атрибути** (multi-valued attributes) можуть одночасно мати декілька значень для одного екземпляра сутності. Прикладом може бути атрибут "Телефонний номер", оскільки особа може мати декілька телефонів. У нотації Чена багатозначні атрибути позначаються подвійним овалом.

Похідні атрибути (derived attributes) є такими, значення яких можуть бути обчислені або виведені з інших атрибутів у базі даних. Наприклад, атрибут "Вік" може бути обчислений з атрибута "Дата народження" та поточної дати. Похідні атрибути позначаються пунктирним овалом. Важливо відзначити, що в реляційній базі даних похідні атрибути зазвичай не зберігаються фізично, а обчислюються динамічно при потребі, що забезпечує консистентність даних та економію пам'яті.

Ключові атрибути мають особливе значення в моделюванні даних. **Ключ** (key) — це атрибут або мінімальна комбінація атрибутів, які унікально ідентифікують кожен екземпляр сутності. У графічній нотації ключові атрибути підкреслюються. Розрізняють декілька типів ключів. **Первинний ключ** (primary key) — це ключ, обраний проектувальником як основний ідентифікатор сутності. **Альтернативні ключі** (alternate keys або candidate keys) — це інші можливі унікальні ідентифікатори, які не були обрані як первинні. **Складений ключ** (composite key) складається з декількох атрибутів, жоден з яких окремо не є унікальним, але їх комбінація забезпечує унікальність.

Зв'язки та їх характеристики

Зв'язки (relationships) визначають асоціації або взаємодії між сутностями. Зв'язки описують, як екземпляри однієї сутності пов'язані з екземплярами іншої сутності. У нотації Чена зв'язки зображуються у вигляді ромбів, з'єднаних лініями з відповідними сутностями.

Зв'язки класифікуються за кількістю сутностей, що беруть участь у зв'язку. **Унарний зв'язок** (unary relationship), також званий рекурсивним зв'язком, встановлюється між екземплярами однієї й тієї ж сутності. Класичним прикладом є зв'язок "керує" в сутності "Співробітник", де один співробітник може керувати іншими співробітниками, при цьому обидві ролі у зв'язку заповнюються екземплярами однієї сутності. **Бінарний зв'язок** (binary relationship) встановлюється між екземплярами двох різних сутностей. Це найбільш поширений тип зв'язку в ER-моделях. Наприклад, зв'язок "працює в"

між сутностями "Співробітник" та "Відділ". **Тернарний зв'язок** (ternary relationship) залучає три різні сутності одночасно. Приклад: "Постачальник постачає Товар для Проекту" — цей зв'язок не може бути адекватно представлений комбінацією бінарних зв'язків, оскільки описує тристоронню взаємодію. **N-арні зв'язки** можуть залучати чотири і більше сутностей, хоча на практиці такі зв'язки зустрічаються рідко.

Кардинальність зв'язку (cardinality) визначає кількісні характеристики участі сутностей у зв'язку. Кардинальність відповідає на питання: скільки екземплярів сутності B може бути пов'язано з одним екземпляром сутності A, і навпаки. Розрізняють три основні типи кардинальності бінарних зв'язків.

Зв'язок один-до-одного (one-to-one, 1:1) означає, що один екземпляр сутності A пов'язаний максимум з одним екземпляром сутності B, і навпаки. Приклад: зв'язок між сутностями "Країна" та "Столиця" — кожна країна має одну столицю, і кожне місто є столицею максимум однієї країни (у більшості випадків). Зв'язки типу 1:1 є найменш поширеними та часто свідчать про необхідність перегляду моделі, оскільки дві сутності з таким зв'язком потенційно можуть бути об'єднані в одну.

Зв'язок один-до-багатьох (one-to-many, 1:N) означає, що один екземпляр сутності A може бути пов'язаний з багатьма екземплярами сутності B, але кожен екземпляр сутності B пов'язаний максимум з одним екземпляром сутності A. Приклад: зв'язок "має" між сутностями "Відділ" та "Співробітник" — один відділ може мати багато співробітників, але кожен співробітник належить до одного відділу. Це найбільш поширений тип зв'язку в реляційних базах даних.

Зв'язок багато-до-багатьох (many-to-many, M:N) означає, що багато екземплярів сутності A можуть бути пов'язані з багатьма екземплярами сутності B. Приклад: зв'язок "вивчає" між сутностями "Студент" та "Курс" — один студент може вивчати багато курсів, і один курс може вивчатися багатьма студентами. Зв'язки типу M:N не можуть бути безпосередньо реалізовані в реляційній моделі та вимагають створення проміжної таблиці зв'язку.

Модальність (optionality або participation) визначає обов'язковість участі екземплярів сутності у зв'язку. Розрізняють обов'язкову участь (mandatory participation), коли кожен екземпляр сутності повинен брати участь у зв'язку, та необов'язкову участь (optional participation), коли участь є факультативною. Модальність часто позначається на діаграмі: обов'язкова участь може бути представлена подвійною лінією або вертикальним штрихом, а необов'язкова — одинарною лінією або колом.

Атрибути зв'язків

Важливою особливістю ER-методу є те, що зв'язки можуть мати власні атрибути. **Атрибути зв'язку** (relationship attributes) описують властивості самого зв'язку, а не сутностей, що беруть у ньому участь. Ці атрибути семантично належать до взаємодії між сутностями, а не до окремих сутностей. Наприклад, у зв'язку "працює над" між сутностями "Співробітник" та "Проект" атрибут "Кількість годин" описує не співробітника і не проект окремо, а саме факт роботи конкретного співробітника над конкретним проектом. Аналогічно, у зв'язку між "Студент" та "Курс" атрибут "Оцінка" належить саме до зв'язку, оскільки оцінка характеризує навчання конкретного студента на конкретному курсі.

Етапи проектування за ER-методом

Процес проектування бази даних з використанням ER-методу є структурованим та включає послідовність взаємопов'язаних етапів, кожен з яких має специфічні цілі та результати.

Етап аналізу предметної області є початковим та критично важливим для успіху всього проекту. На цьому етапі виконується збір вимог до майбутньої системи через інтерв'ювання користувачів, експертів предметної області, аналіз існуючих документів, форм, звітів, спостереження за бізнес-процесами. Результатом цього етапу є детальний опис предметної області, список функціональних та нефункціональних вимог до системи, глосарій термінів предметної області. Якість виконання цього етапу безпосередньо впливає на адекватність та повноту майбутньої моделі.

Етап ідентифікації сутностей передбачає виявлення основних об'єктів предметної області, про які необхідно зберігати інформацію. Критерієм виділення сутності є наявність множини екземплярів об'єктів одного типу з однаковими характеристиками та наявність інформації, яку необхідно зберігати про ці об'єкти. Важливо відрізняти сутності від атрибутів — загальним правилом є те, що якщо про об'єкт необхідно зберігати лише одне значення (наприклад, назву), це швидше атрибут, а якщо множину характеристик — це сутність. Також важливо визначити відповідні імена сутностей, які повинні бути іменниками в однині, змістовними та зрозумілими для експертів предметної області.

Етап визначення атрибутів включає ідентифікацію властивостей кожної сутності. Для кожного атрибута визначаються його назва, тип даних, домен допустимих значень, обов'язковість заповнення, значення за замовчуванням. На цьому етапі також виконується аналіз атрибутів на предмет їх атомарності — складені атрибути при необхідності декомпозуються на простіші компоненти. Визначаються багатозначні атрибути, які потенційно можуть бути винесені в окремі сутності для підтримки першої нормальної форми. Ідентифікуються

похідні атрибути та приймається рішення про доцільність їх фізичного зберігання або динамічного обчислення.

Етап встановлення зв'язків передбачає аналіз взаємодій між сутностями та визначення типів цих взаємодій. Для кожної пари сутностей аналізується, чи існує між ними семантичний зв'язок, що повинен бути відображений у моделі. Для кожного виявленого зв'язку визначається його змістовна назва, яка зазвичай представлена дієсловом або дієслівною фразою, що описує характер взаємодії. Важливо уникати надмірного моделювання зв'язків — включати лише ті, які дійсно необхідні для підтримки функціональності системи.

Етап визначення кардинальності є критичним для коректної реалізації моделі. Для кожного зв'язку необхідно точно визначити кардинальність в обох напрямках та модальність участі кожної сутності. Це вимагає глибокого розуміння бізнес-правил предметної області. Помилки у визначенні кардинальності можуть призвести до неможливості представлення певних ситуацій або, навпаки, до представлення невалідних станів системи.

Етап створення ER-діаграми полягає у графічному представленні всіх елементів моделі згідно з обраною нотацією. Діаграма повинна бути візуально зрозумілою, з логічним розташуванням елементів, мінімальною кількістю перетинів ліній зв'язків. Для великих моделей доцільно створювати декілька діаграм, що представляють різні тематичні області або рівні деталізації — від діаграми високого рівня з основними сутностями до детальних діаграм окремих підсистем.

Етап валідації моделі включає перевірку моделі на повноту, коректність та відповідність вимогам. Валідація виконується шляхом проходження типових сценаріїв використання системи та перевірки, чи може модель підтримати всі необхідні операції. Модель переглядається разом з експертами предметної області для підтвердження коректності представлення бізнес-правил. На цьому етапі також виконується перевірка на відсутність надмірності, наявність всіх необхідних ключів, коректність типів зв'язків.

Етап трансформації в реляційну модель є завершальним етапом концептуального проектування. ER-модель перетворюється в набір таблиць реляційної бази даних відповідно до формальних правил трансформації.

Правила трансформації ER-моделі в реляційну модель

Перетворення ER-моделі в реляційну схему виконується згідно з чітко визначеними правилами, які забезпечують збереження семантики вихідної моделі.

Правило трансформації сильних сутностей: кожна сильна сутність перетворюється в окрему таблицю (відношення). Назва таблиці відповідає назві сутності, зазвичай в множині. Кожен атрибут сутності стає колонкою таблиці. Ключовий атрибут або комбінація атрибутів стає первинним ключем таблиці. Для кожної колонки визначається відповідний тип даних СУБД, обмеження NOT NULL для обов'язкових атрибутів, значення за замовчуванням та інші обмеження цілісності.

Правило трансформації слабких сутностей: слабка сутність також перетворюється в таблицю, але її первинний ключ є складеним і включає як власний часткові ключ слабкої сутності, так і первинний ключ сутності-власника (сильної сутності, від якої залежить слабка). Це забезпечує унікальність екземплярів слабкої сутності в контексті конкретного екземпляра сутності-власника. Автоматично створюється зовнішній ключ, що посилається на таблицю сутності-власника.

Правило трансформації зв'язків один-до-багатьох: зв'язок 1:N реалізується шляхом міграції первинного ключа сутності на боці "один" до таблиці сутності на боці "багато" як зовнішнього ключа. Цей зовнішній ключ посилається на первинний ключ батьківської таблиці. Обов'язковість зв'язку визначає, чи може зовнішній ключ містити NULL-значення. Якщо зв'язок має власні атрибути, вони також додаються до таблиці на боці "багато".

Правило трансформації зв'язків один-до-одного: зв'язок 1:1 може бути реалізований декількома способами. Перший варіант — об'єднання обох сутностей в одну таблицю, що доцільно, якщо зв'язок є обов'язковим з обох боків. Другий варіант — додавання зовнішнього ключа до однієї з таблиць, що посилається на первинний ключ іншої таблиці. Вибір таблиці для розміщення зовнішнього ключа зазвичай базується на семантиці зв'язку або на боці необов'язкової участі. Третій варіант — створення окремої таблиці зв'язку з двома зовнішніми ключами, що може бути корисним, якщо зв'язок має значну кількість власних атрибутів.

Правило трансформації зв'язків багато-до-багатьох: зв'язок M:N завжди реалізується через створення додаткової **таблиці зв'язку** (junction table, associative table або linking table). Ця таблиця містить як мінімум два зовнішні ключі, що посилаються на первинні ключі обох сутностей, що беруть участь у зв'язку. Комбінація цих двох зовнішніх ключів зазвичай формує складений первинний ключ таблиці зв'язку. Якщо зв'язок має власні атрибути, вони додаються як додаткові колонки таблиці зв'язку. Наприклад, зв'язок M:N між "Студент" та "Курс" перетворюється в таблицю "Реєстрації" з зовнішніми

ключами StudentID та CourseID та додатковими атрибутами, такими як дата реєстрації та оцінка.

Правило трансформації багатозначних атрибутів: багатозначний атрибут перетворюється в окрему таблицю з двома колонками — зовнішнім ключем, що посилається на первинний ключ батьківської таблиці, та колонкою для зберігання значення багатозначного атрибута. Первинним ключем нової таблиці є комбінація обох колонок. Наприклад, багатозначний атрибут "Телефон" сутності "Особа" перетворюється в таблицю "Телефони" з колонками PersonID (зовнішній ключ) та PhoneNumber.

Правильне застосування цих правил трансформації забезпечує створення нормалізованої реляційної схеми, яка адекватно відображає семантику концептуальної ER-моделі та підтримує цілісність даних на структурному рівні.

16. ОСНОВНІ ПОНЯТТЯ ПРО СЕРЕДОВИЩЕ ГРАФІЧНОЇ МОВИ ERWIN

ERWin (Entity Relationship for Windows) являє собою професійний CASE-інструмент (Computer-Aided Software Engineering) для проектування баз даних, розроблений компанією Computer Associates, яка згодом трансформувалася у CA Technologies. Цей програмний продукт підтримує повний життєвий цикл проектування баз даних, починаючи від етапу концептуального моделювання та завершуючи генерацією фізичної схеми бази даних для конкретної СУБД.

Архітектура та рівні моделювання в ERWin

Середовище ERWin реалізує багаторівневий підхід до моделювання даних, що відповідає загальноприйнятим стандартам проектування інформаційних систем. Програмний продукт підтримує два основні рівні представлення моделі даних, кожен з яких має специфічне призначення та характеристики.

Логічна модель (Logical Model) представляє дані на абстрактному рівні, незалежному від особливостей конкретної системи управління базами даних. На цьому рівні проектувальник оперує поняттями сутностей, атрибутів та зв'язків у термінах, які відображають предметну область без прив'язки до технічних деталей реалізації. Логічна модель є платформо-незалежним представленням структури даних, що дозволяє зосередитися на бізнес-логіці та взаємозв'язках між об'єктами предметної області. Важливою характеристикою логічної моделі є те, що вона може бути трансформована в фізичні моделі для різних СУБД без зміни базової структури.

Фізична модель (Physical Model) описує конкретну реалізацію логічної моделі в обраній системі управління базами даних. Цей рівень моделювання

враховує специфічні особливості цільової СУБД, включаючи типи даних, обмеження, індекси, секціонування та інші аспекти фізичної організації даних. Фізична модель безпосередньо відображає структуру таблиць, колонок та взаємозв'язків, які будуть створені в базі даних. ERWin забезпечує автоматичну трансформацію логічної моделі у фізичну з урахуванням обраної СУБД, що значно прискорює процес проектування.

Система типів даних в ERWin

Середовище ERWin оперує двома категоріями типів даних, які відповідають різним рівням моделювання та забезпечують гнучкість у проектуванні.

Логічні типи даних є універсальними конструкціями, які не залежать від специфіки конкретної СУБД. Ці типи включають такі базові категорії, як String для представлення символьних рядків, Integer для цілих чисел, Decimal для чисел з фіксованою точністю, Date для дат, Time для часу, DateTime для комбінованих значень дати та часу, Boolean для логічних значень. Використання логічних типів на етапі концептуального моделювання забезпечує портативність моделі та можливість її адаптації до різних СУБД без необхідності переробки базової структури.

Фізичні типи даних є специфічними для конкретної системи управління базами даних та відображають реальні типи, які підтримуються цільовою СУБД. Наприклад, для Oracle Server логічний тип String може бути трансформований у VARCHAR2 або CHAR, для Microsoft SQL Server — у NVARCHAR або VARCHAR, для PostgreSQL — у TEXT або CHARACTER VARYING. Кожна СУБД має власний набір підтримуваних типів даних з різними характеристиками продуктивності, обмеженнями розміру та семантикою використання. ERWin автоматично виконує відображення логічних типів у відповідні фізичні типи обраної СУБД, хоча проектувальник завжди має можливість вручну скоригувати це відображення відповідно до специфічних вимог проекту.

Структурні елементи моделі в ERWin

Графічне представлення моделі даних в ERWin базується на розширеній нотації Entity-Relationship діаграм з додатковими елементами для відображення фізичних характеристик.

Сутності (Entities) візуалізуються у формі прямокутників, структурованих на декілька логічних секцій. Верхня частина прямокутника містить заголовок з назвою сутності, яка може відображатися як у логічній, так і у фізичній формі залежно від поточного режиму перегляду. Безпосередньо під заголовком розташована секція атрибутів первинного ключа, які виділяються візуально для

підкреслення їх особливої ролі в ідентифікації екземплярів сутності. Нижня секція містить неключові атрибути, що описують властивості сутності. Така структурована візуалізація забезпечує швидке розуміння структури сутності та дозволяє відразу ідентифікувати ключові поля.

Атрибути в ERWin характеризуються комплексом властивостей, які визначають їх поведінку та характеристики. Кожен атрибут має логічну назву, яка використовується на концептуальному рівні моделювання, та фізичну назву, яка використовується для генерації SQL-коду та безпосередньо стане іменем колонки в таблиці бази даних. Атрибут асоціюється з певним типом даних, причому на логічному рівні використовується логічний тип, а на фізичному — відповідний тип даних цільової СУБД. Властивість обов'язковості (NOT NULL) визначає, чи може атрибут містити невизначене значення. Можливість встановлення значення за замовчуванням дозволяє автоматично заповнювати атрибут при створенні нового запису. Валідаційні правила визначають обмеження на допустимі значення атрибута, забезпечуючи цілісність даних на рівні структури бази даних.

Зв'язки між сутностями відображаються лініями, що з'єднують прямокутники сутностей, з додатковою семантичною інформацією про характер взаємозв'язку. Кожен зв'язок характеризується напрямком, кардинальністю та типом. Батьківська сутність (parent entity) розташовується на боці відношення типу "один", тоді як дочірня сутність (child entity) знаходиться на боці "багато". ERWin розрізняє два фундаментальні типи зв'язків, які мають різну семантику та по-різному реалізуються в фізичній базі даних.

Ідентифікуючий зв'язок (identifying relationship) позначається суцільною лінією та має особливу семантику: первинний ключ батьківської сутності не лише мігрує до дочірньої сутності як зовнішній ключ, але й стає частиною складеного первинного ключа дочірньої сутності. Це означає, що екземпляр дочірньої сутності не може існувати без відповідного екземпляра батьківської сутності, і його ідентичність частково визначається через зв'язок з батьківською сутністю. Ідентифікуючі зв'язки використовуються для моделювання слабких сутностей та залежних об'єктів, які не мають самостійного існування.

Неідентифікуючий зв'язок (non-identifying relationship) позначається пунктирною лінією та означає, що первинний ключ батьківської сутності мігрує до дочірньої сутності виключно як зовнішній ключ, не входячи до складу її первинного ключа. Дочірня сутність має власний незалежний первинний ключ і може існувати автономно, хоча й може посилатися на батьківську сутність через

зовнішній ключ. Неідентифікуючі зв'язки є найбільш поширеним типом взаємозв'язків у реляційних базах даних.

Домени в ERWin

Концепція **доменів** (Domains) в ERWin реалізує механізм повторного використання визначень типів даних з асоційованими валідаційними правилами та обмеженнями. Домен являє собою іменований тип даних зі специфікованими характеристиками, який може бути багаторазово застосований до різних атрибутів у моделі. Наприклад, домен "EmailAddress" може бути визначений як символьний рядок з максимальною довжиною 255 символів та регулярним виразом для валідації формату електронної адреси. Цей домен може бути призначений атрибутам "Email" у різних сутностях моделі, забезпечуючи консистентність визначень та централізоване управління правилами валідації. Використання доменів підвищує якість моделі, зменшує вірогідність помилок та спрощує внесення змін, оскільки модифікація домену автоматично поширюється на всі атрибути, які його використовують.

Робоче середовище та інтерфейс ERWin

Користувацький інтерфейс ERWin організований за принципом багатовіконного середовища з декількома спеціалізованими робочими областями, кожна з яких призначена для виконання специфічних завдань проектування.

Model Explorer представляє собою ієрархічне дерево навігації, яке відображає всі об'єкти моделі в структурованій формі. Цей компонент дозволяє швидко переміщатися між різними елементами моделі, знаходити специфічні сутності, атрибути чи зв'язки, а також виконувати базові операції редагування безпосередньо в навігаційному дереві. Model Explorer особливо корисний при роботі з великими моделями, що містять десятки або сотні сутностей, коли візуальна навігація по діаграмі стає утрудненою.

Diagram Window являє собою основну робочу область для візуального проектування моделі даних. У цьому вікні відображається графічне представлення сутностей, атрибутів та зв'язків у формі ER-діаграми. Проектувальник може безпосередньо маніпулювати елементами діаграми, створювати нові об'єкти, встановлювати зв'язки, змінювати розташування елементів для покращення читабельності. Diagram Window підтримує масштабування, панорамування та різні режими відображення для забезпечення оптимальної візуалізації моделі. Середовище дозволяє створювати декілька

діаграм у межах однієї моделі, що особливо корисно для великих проектів, де різні діаграми можуть відображати різні **тематичні області** (subject areas) системи.

Properties Window відображає детальні властивості вибраного об'єкта моделі та надає засоби для їх редагування. Це вікно динамічно змінює свій вміст залежно від типу вибраного елемента — для сутності відображаються одні властивості, для атрибута — інші, для зв'язку — треті. Properties Window організоване у формі структурованих вкладок або секцій, що групують пов'язані властивості для зручності навігації та редагування.

Toolbox являє собою панель інструментів, яка містить засоби для створення та маніпуляції об'єктами моделі. Інструменти організовані в логічні групи відповідно до типів операцій, які вони виконують. Toolbox забезпечує швидкий доступ до найбільш часто використовуваних функцій проектування, дозволяючи проектувальнику ефективно створювати та модифікувати структуру бази даних.

Інтеграція цих робочих областей створює потужне та гнучке середовище для проектування баз даних, яке підтримує як візуальний, так і текстовий підходи до моделювання, забезпечуючи ефективність роботи проектувальників з різним досвідом та уподобаннями.

17. КОМАНДИ ТА ІНСТРУМЕНТИ ERWIN

Середовище ERWin надає розширений набір команд, інструментів та функціональних можливостей для всебічної підтримки процесу проектування баз даних. Ці засоби організовані в логічні групи відповідно до етапів та аспектів проектування, забезпечуючи ефективний робочий процес від початкового моделювання до генерації фізичної схеми та документації.

Інструменти створення та редагування об'єктів моделі

Entity Tool є основним інструментом для створення нових сутностей у моделі. Активація цього інструменту переводить систему в режим створення сутності, після чого одинарний клік мишею на робочій області діаграми створює нову сутність у вказаній позиції. Новостворена сутність отримує тимчасову назву за замовчуванням, яку проектувальник може негайно змінити. Подвійний клік на існуючій сутності відкриває діалогове вікно **Entity Editor**, яке надає доступ до всіх властивостей сутності через систему структурованих вкладок. Вкладка **General** містить основні властивості, такі як логічна та фізична назви, опис сутності, тип сутності. Вкладка **Attributes** надає можливість додавання,

редагування та видалення атрибутів, визначення їх порядку відображення. Вкладка **Key Groups** дозволяє визначати первинні та альтернативні ключі. Вкладка **Indexes** призначена для створення та налаштування індексів. Додаткові вкладки надають доступ до тригерів, валідаційних правил, дозволів безпеки та інших аспектів визначення сутності.

Relationship Tool забезпечує створення різноманітних типів зв'язків між сутностями. Інструмент має декілька варіантів, кожен з яких призначений для створення специфічного типу зв'язку. **Identifying Relationship Tool** створює ідентифікуючий зв'язок, при якому первинний ключ батьківської сутності мігрує до дочірньої сутності та стає частиною її складеного первинного ключа. На діаграмі такий зв'язок відображається суцільною лінією, що візуально відрізняє його від інших типів зв'язків. **Non-Identifying Relationship Tool** створює неідентифікуючий зв'язок, де первинний ключ батьківської сутності мігрує як звичайний зовнішній ключ, не входячи до складу первинного ключа дочірньої сутності. Такий зв'язок зображується пунктирною лінією. **Many-to-Many Relationship Tool** створює зв'язок типу багато-до-багатьох, який на логічному рівні моделювання відображається як прямий зв'язок між двома сутностями, а при переході до фізичного рівня автоматично трансформується в проміжну таблицю зв'язку з відповідними зовнішніми ключами.

Процес створення зв'язку є інтуїтивним: після вибору відповідного інструменту зв'язку проектувальник клікає на батьківську сутність та, утримуючи кнопку миші, перетягує курсор до дочірньої сутності, після чого відпускає кнопку. Система автоматично створює зв'язок, мігрує необхідні атрибути та встановлює відповідні обмеження. Подвійний клік на лінії зв'язку відкриває **Relationship Editor**, який дозволяє налаштувати детальні властивості зв'язку, включаючи кардинальність, модальність, правила каскадних операцій, словесні фрази для опису зв'язку в обох напрямках.

Attribute Tool надає можливість швидкого додавання атрибутів безпосередньо на діаграмі без необхідності відкривати окремі діалогові вікна. Після активації інструменту та клікання на сутності з'являється inline-редактор, в якому можна ввести назву нового атрибута. Цей підхід особливо ефективний на початкових етапах моделювання, коли необхідно швидко створити базову структуру моделі. Детальне налаштування властивостей атрибутів може бути виконане пізніше через Entity Editor.

Команди меню та їх функціональність

Система меню ERWin організована за функціональним принципом, де кожне меню групує команди, пов'язані з певним аспектом роботи з моделлю.

Model Menu містить команди для роботи з моделлю в цілому. Команда **Model Properties** відкриває діалогове вікно з глобальними властивостями моделі, де визначаються назва моделі, ціль моделювання, цільова СУБД, стандарти іменування, налаштування відображення та інші параметри, що впливають на всю модель. Команда **Logical/Physical** дозволяє перемикатися між логічним та фізичним представленням моделі. При переключенні з логічного на фізичний рівень ERWin автоматично виконує трансформацію логічних типів даних у відповідні фізичні типи обраної СУБД, застосовує правила іменування для створення фізичних назв таблиць та колонок, генерує DDL-конструкції для реалізації зв'язків через зовнішні ключі. Команда **Domains** відкриває редактор доменів, де можна створювати, модифікувати та видаляти домени — іменовані типи даних з асоційованими валідаційними правилами. Команда **Subject Areas** дозволяє організувати сутності моделі в тематичні групи, що особливо корисно для великих моделей, де сотні сутностей можуть бути логічно згруповані за функціональними областями системи.

Entities Menu об'єднує команди для роботи з сутностями. Команда **New Entity** створює нову сутність без необхідності використання графічного інструменту, що зручно при роботі через Model Explorer. Команда **Entity Editor** відкриває детальний редактор властивостей вибраної сутності. Команда **Key Groups** надає спеціалізований інтерфейс для визначення та управління групами ключів, включаючи первинні ключі, альтернативні ключі (candidate keys) та інверсні ключі. Команда **Indexes** відкриває редактор індексів, де можна створювати різні типи індексів — B-tree для загальних пошукових операцій, унікальні індекси для забезпечення унікальності значень, bitmap-індекси для колонок з низькою кардинальністю, повнотекстові індекси для текстового пошуку. Для кожного індексу визначається список колонок з їх порядком (висхідним або спадним), фактор заповнення, налаштування табличного простору та інші специфічні для СУБД параметри.

Relationships Menu містить команди для управління зв'язками між сутностями. Команда **Relationship Editor** відкриває детальний редактор властивостей вибраного зв'язку. У цьому редакторі можна змінити тип зв'язку (ідентифікуючий або неідентифікуючий), налаштувати **правила каскадних операцій** для операцій видалення та оновлення, визначити індексацію зовнішніх ключів, встановити додаткові обмеження. Команда **Cardinality** дозволяє змінити кардинальність існуючого зв'язку — перетворити зв'язок 1:N у 1:1 або M:N, що автоматично виконує необхідні структурні зміни в моделі. Команда **Verb Phrase** надає можливість визначити словесні описи зв'язку в обох напрямках, що

покращує читабельність моделі. Наприклад, для зв'язку між "Студент" та "Курс" можна визначити фрази "записується на" (від студента до курсу) та "має записаних" (від курсу до студента). Ці фрази відображаються на діаграмі та використовуються при генерації документації.

Tools Menu містить потужні інструменти для взаємодії моделі з реальними базами даних та генерації різноманітних артефактів. Ці інструменти реалізують критично важливі функції, які забезпечують практичне застосування створених моделей.

Forward Engineer є інструментом для генерації SQL-скриптів на основі моделі. Команда **Schema Generation** відкриває майстер прямого інжинірингу, який проводить проектувальника через процес налаштування параметрів генерації та створення SQL-коду. Майстер дозволяє вибрати, які об'єкти бази даних повинні бути включені в скрипт: таблиці, індекси, обмеження цілісності, тригери, збережені процедури, представлення. Можна налаштувати порядок генерації операторів, формат коду, включення коментарів. Система генерує не лише оператори CREATE для створення нових об'єктів, але й може створювати оператори DROP для видалення існуючих об'єктів, що корисно при повному перестворенні бази даних. Згенерований SQL-код можна переглянути, відредагувати при необхідності, зберегти у файл або безпосередньо виконати на цільовому сервері бази даних, якщо налаштовано відповідне підключення.

Reverse Engineer реалізує зворотний інжиніринг — імпорт структури існуючої бази даних в ERWin модель. Цей інструмент підключається до працюючої бази даних, зчитує її метадані та створює відповідну ER-модель. Процес зворотного інжинірингу включає декілька етапів. Спочатку встановлюється підключення до бази даних через відповідний драйвер. Потім система зчитує список доступних схем, таблиць, представлень та інших об'єктів. Проектувальник вибирає, які об'єкти повинні бути імпортовані. ERWin аналізує структуру таблиць, первинні та зовнішні ключі, індекси, обмеження, тригери та створює відповідні сутності, атрибути та зв'язки в моделі. Зворотний інжиніринг особливо корисний при роботі з legacy-системами, документування існуючих баз даних, міграції між СУБД або створенні базової моделі для подальшого рефакторингу структури даних.

Complete Compare є інструментом для порівняння та синхронізації моделі з реальною базою даних. Ця функція критично важлива в процесі ітеративної розробки, коли модель еволюціонує, а база даних повинна бути оновлена відповідно до змін у моделі. Complete Compare виконує двосторонній аналіз, виявляючи відмінності між моделлю та базою даних: таблиці, що існують в

моделі, але відсутні в базі даних; таблиці, що існують в базі даних, але відсутні в моделі; відмінності в структурі існуючих таблиць, включаючи додані, видалені або змінені колонки; відмінності в індексах, обмеженнях, тригерах. Після аналізу система генерує ALTER-скрипти, які містять SQL-оператори для приведення бази даних у відповідність з моделлю. Проектувальник може вибірково застосувати ці зміни, виключивши певні операції, якщо це необхідно. Також можливий зворотний напрямок синхронізації — оновлення моделі відповідно до змін, внесених безпосередньо в базу даних.

Generate Database дозволяє створити фізичну базу даних безпосередньо з моделі без проміжного етапу генерації та виконання SQL-скриптів. Ця команда встановлює підключення до серверу СУБД та виконує всі необхідні операції для створення повної структури бази даних відповідно до моделі, включаючи створення схем, табличних просторів, таблиць, індексів, обмежень, тригерів та інших об'єктів.

Функції валідації та контролю якості моделі

Якість моделі даних безпосередньо впливає на якість фізичної бази даних та ефективність її функціонування. ERWin надає розвинені засоби для валідації моделі та виявлення потенційних проблем.

Model Validation є комплексним інструментом перевірки моделі на відповідність різноманітним правилам та стандартам проектування. Валідація виконує перевірки декількох категорій. **Структурна валідація** перевіряє базову коректність моделі: наявність первинних ключів у всіх сутностей, коректність визначення зовнішніх ключів, відсутність циклічних залежностей в ідентифікуючих зв'язках, коректність визначення доменів та типів даних. **Семантична валідація** перевіряє логічну консистентність моделі: відсутність дубльованих сутностей або атрибутів, коректність кардинальності зв'язків, відповідність типів даних зовнішніх ключів первинним ключам, на які вони посилаються. **Валідація стандартів іменування** перевіряє відповідність назв об'єктів встановленим конвенціям: формат назв таблиць, колонок, індексів, використання префіксів або суфіксів, відсутність зарезервованих слів СУБД в назвах. **Валідація повноти** перевіряє, чи всі необхідні метадані визначені: наявність описів сутностей та атрибутів, визначення обов'язковості атрибутів, наявність індексів для зовнішніх ключів. Результати валідації представляються у структурованому звіті з категоризацією проблем за рівнем критичності: помилки, попередження, інформаційні повідомлення. Для кожної виявленої проблеми надається детальний опис та рекомендації щодо виправлення.

Report Designer надає гнучкі можливості для створення різноманітної документації на основі моделі. Документація є критично важливою для комунікації з зацікавленими сторонами, передачі знань в команді, архівування проектних рішень. ERWin підтримує декілька типів стандартних звітів та надає можливість створення користувацьких звітів.

Entity Report генерує детальну документацію про сутності моделі, включаючи логічні та фізичні назви, описи, список всіх атрибутів з їх характеристиками, визначення первинних та альтернативних ключів, список зв'язків, в яких бере участь сутність, індекси, обмеження. **Attribute Report** надає детальну інформацію про атрибути, включаючи назви, типи даних, обов'язковість, значення за замовчуванням, домени, валідаційні правила, використання в ключах та індексах. **Relationship Report** документує всі зв'язки між сутностями з описом типу зв'язку, кардинальності, модальності, міграції атрибутів, каскадних правил. **Subject Area Report** надає високорівневий огляд моделі з групуванням сутностей за функціональними областями. **Data Dictionary** генерує повний словник даних, який містить всю інформацію про всі об'єкти моделі в структурованому форматі, придатному для використання як довідковий матеріал.

Система також підтримує створення **Custom Reports** — користувацьких звітів з повністю контрольованим форматом та змістом. Проектувальник може визначити, які об'єкти та які їх властивості повинні бути включені в звіт, задати формат представлення, групування, сортування. Звіти можуть бути згенеровані в різних форматах: HTML для веб-публікації, PDF для друку та розповсюдження, RTF для редагування в текстових процесорах, XML для подальшої обробки.

Налаштування відображення та візуалізації

Ефективна візуалізація моделі критично важлива для її розуміння та роботи з нею, особливо для великих та складних моделей.

Display Options надає детальний контроль над тим, які елементи моделі відображаються на діаграмі. Можна вибірково показувати або приховувати: типи даних атрибутів, домени, обов'язковість атрибутів, значення за замовчуванням, коментарі, фізичні назви об'єктів, кардинальність зв'язків, словесні фрази зв'язків. Ці налаштування дозволяють адаптувати рівень деталізації діаграми до конкретних потреб: для обговорення з бізнес-аналітиками корисно приховати технічні деталі та показувати логічні назви; для розробників доцільно показувати всю технічну інформацію, включаючи типи даних та обмеження.

Layout Options надає засоби для автоматичного розміщення об'єктів на діаграмі. Алгоритми автоматичного розміщення аналізують структуру моделі та

організовують сутності таким чином, щоб мінімізувати перетини ліній зв'язків, забезпечити збалансоване використання простору діаграми, групувати пов'язані сутності. Підтримуються різні стилі розміщення: ієрархічний (батьківські сутності зверху, дочірні знизу), ортогональний (зв'язки під прямими кутами), circular (сутності по колу). Проектувальник може застосувати автоматичне розміщення до всієї діаграми або до вибраної групи сутностей, а потім вручну скоригувати позиції окремих елементів.

Color Schemes дозволяє налаштувати кольорове кодування елементів моделі для покращення візуального сприйняття. Різні типи сутностей можуть бути відображені різними кольорами: довідкові таблиці одним кольором, транзакційні таблиці іншим, таблиці зв'язку третім. Можна встановити колір для окремих сутностей, що дозволяє виділити критично важливі або проблемні елементи. Зв'язки також можуть бути кольорово диференційовані за типом або іншими характеристиками. Правильне використання кольорів значно покращує читабельність складних діаграм.

Font Settings надає контроль над типографікою діаграми, дозволяючи налаштувати шрифти, їх розміри та стилі для різних типів текстових елементів: назв сутностей, атрибутів, зв'язків, коментарів. Уніфікований та продуманий типографічний стиль підвищує професійність вигляду документації та полегшує її читання.

18. ПРОЕКТУВАННЯ БАЗ ДАНИХ ЗАСОБАМИ ERWIN

Практичний процес проектування бази даних засобами ERWin являє собою структуровану послідовність етапів, кожен з яких використовує специфічні можливості інструменту та має чітко визначені результати. Розглянемо детально всі аспекти цього процесу на прикладі проектування бази даних для типової предметної області.

Етап 1: Ініціалізація проекту та створення логічної моделі

Початок роботи над новим проектом бази даних в ERWin починається з запуску **New Model Wizard** — майстра створення нової моделі. Цей інтерактивний помічник проводить проектувальника через процес початкового налаштування проекту, пропонуючи зробити декілька фундаментальних виборів, які визначатимуть характер подальшої роботи.

На першому кроці майстра необхідно обрати **тип початкової моделі**. ERWin пропонує два основні варіанти: розпочати з логічної моделі (Logical Model) або одразу з фізичної моделі (Physical Model). Вибір логічної моделі є рекомендованим підходом для більшості проектів, оскільки дозволяє

зосередитися на бізнес-аспектах структури даних без відволікання на технічні деталі конкретної СУБД. Логічна модель є платформо-незалежною та може бути згодом трансформована в фізичні моделі для різних СУБД. Проте, якщо проект має жорстко визначену цільову платформу і проектувальник має глибоке знання специфіки цієї СУБД, можливий прямий початок з фізичної моделі.

Якщо обрано створення фізичної моделі або планується подальша трансформація логічної моделі у фізичну, майстер пропонує обрати **цільову СУБД** зі списку підтримуваних систем. ERWin підтримує широкий спектр комерційних та відкритих СУБД, включаючи Oracle Database, Microsoft SQL Server, IBM DB2, MySQL, PostgreSQL, SQLite та багато інших. Вибір цільової СУБД впливає на доступні типи даних, синтаксис генерованого SQL-коду, підтримувані можливості індексації та інші аспекти фізичної реалізації.

Наступним кроком є налаштування **базових параметрів моделі**. Визначається назва моделі, яка повинна бути описовою та відображати призначення бази даних. Встановлюються правила іменування (naming conventions), які визначають, як логічні назви трансформуються у фізичні назви: використання верхнього чи нижнього регістра, формат розділення слів (підкреслення, CamelCase), префікси для різних типів об'єктів. Визначається автор моделі, організація, дата створення та інша метаінформація, яка буде включена в документацію.

Створення та налаштування сутностей

Після завершення початкового налаштування проектувальник переходить до безпосереднього моделювання структури даних. Створення сутностей виконується за допомогою **Entity Tool** з панелі інструментів. Активація інструменту та клік на робочій області діаграми створює нову сутність з тимчасовою назвою за замовчуванням, наприклад "Entity_1". Проектувальник негайно вводить змістовну назву сутності, яка повинна бути іменником в однині та відображати тип об'єктів, що моделюються.

Для детального налаштування властивостей сутності виконується подвійний клік на створеній сутності, що відкриває **Entity Editor**. На вкладці **General** визначаються основні характеристики сутності. Поле **Entity Name** містить логічну назву сутності, яка використовується на концептуальному рівні та повинна бути зрозумілою для бізнес-користувачів. Поле **Physical Name** містить фізичну назву, яка буде використана як ім'я таблиці в базі даних. За замовчуванням ERWin автоматично генерує фізичну назву на основі логічної відповідно до встановлених правил іменування, але проектувальник може вручну скоригувати її при необхідності. Поле **Definition** призначене для

введення детального текстового опису призначення сутності, що є важливою частиною документації моделі.

Додавання атрибутів до сутності виконується на вкладці **Attributes**. Інтерфейс представляє табличне подання з рядком для кожного атрибута та колонками для різних властивостей атрибута. Для додавання нового атрибута використовується кнопка **New** або контекстне меню. Для кожного атрибута визначаються наступні ключові властивості.

Attribute Name — логічна назва атрибута, яка описує зберігану інформацію в термінах предметної області. Назва повинна бути описовою та однозначною в контексті сутності. **Physical Name** — фізична назва, яка стане іменем колонки в таблиці бази даних. **Datatype** — тип даних атрибута. На логічному рівні вибирається логічний тип даних (String, Integer, Decimal, Date, DateTime, Boolean), а при переході до фізичного рівня ERWin автоматично трансформує його у відповідний тип даних цільової СУБД з можливістю ручного коригування.

Domain — опціонально атрибут може бути асоційований з попередньо визначеним доменом, що забезпечує консистентність визначень однотипних атрибутів у різних сутностях. При виборі домена тип даних та валідаційні правила успадковуються від домену. **Null Option** — властивість, що визначає, чи може атрибут містити невизначене (NULL) значення. Для обов'язкових атрибутів встановлюється **NOT NULL**, що гарантує, що кожен запис матиме значення для цього атрибута.

Default Value — значення, яке автоматично присвоюється атрибуту при створенні нового запису, якщо явно не вказано інше значення. Значення за замовчуванням може бути константою (число, рядок, дата) або виразом (наприклад, CURRENT_DATE для автоматичного встановлення поточної дати). **Definition** — текстовий опис призначення та семантики атрибута, який включається в документацію та допомагає розробникам розуміти призначення даних.

Після визначення всіх атрибутів сутності необхідно визначити **первинний ключ**. Первинний ключ може складатися з одного або декількох атрибутів, які в комбінації унікально ідентифікують кожен запис. Для визначення первинного ключа використовується вкладка **Key Groups** в Entity Editor. Створюється нова ключова група типу **Primary Key** та до неї додаються атрибути, що формують первинний ключ. Якщо первинний ключ складається з одного атрибута (простий ключ), зазвичай це суррогатний ключ типу автоінкрементного цілого числа, що не несе бізнес-семантики, але забезпечує стабільну та ефективну ідентифікацію

записів. Якщо первинний ключ є складеним (composite key), він може включати декілька атрибутів, які разом забезпечують унікальність.

Встановлення зв'язків між сутностями

Після створення базового набору сутностей з їх атрибутами та ключами наступним етапом є визначення зв'язків між сутностями, які відображають взаємодії та асоціації між об'єктами предметної області.

Процес створення зв'язку починається з вибору відповідного інструменту зв'язку з панелі інструментів. Вибір між **Identifying Relationship Tool** та **Non-Identifying Relationship Tool** базується на семантиці зв'язку. Ідентифікуючий зв'язок використовується, коли дочірня сутність є залежною від батьківської, і її екземпляри не мають незалежного існування. Класичним прикладом є зв'язок між "Замовлення" та "Рядок замовлення" — рядок замовлення не має сенсу без контексту конкретного замовлення, тому ID замовлення входить до складу первинного ключа рядка замовлення. Неідентифікуючий зв'язок використовується, коли дочірня сутність має власний незалежний ідентифікатор, але посилається на батьківську сутність. Наприклад, зв'язок між "Відділ" та "Співробітник" — співробітник має власний унікальний ID, але також має зовнішній ключ, що вказує на відділ, до якого він належить.

Після вибору інструменту проектувальник клікає на батьківську сутність (на боці "один" зв'язку) та, утримуючи кнопку миші, перетягує курсор до дочірньої сутності (на боці "багато" зв'язку). При відпусканні кнопки миші ERWin автоматично створює зв'язок та виконує **міграцію атрибутів** — первинний ключ батьківської сутності копіюється до дочірньої сутності як зовнішній ключ. У випадку ідентифікуючого зв'язку мігровані атрибути автоматично додаються до складу первинного ключа дочірньої сутності. У випадку неідентифікуючого зв'язку мігровані атрибути залишаються звичайними неключовими атрибутами з встановленим обмеженням зовнішнього ключа.

Детальне налаштування властивостей зв'язку виконується через **Relationship Editor**, який відкривається подвійним кліком на лінії зв'язку. На вкладці **General** визначаються основні характеристики зв'язку. **Relationship Name** — назва зв'язку, яка часто формується автоматично на основі назв сутностей, що беруть участь. **Verb Phrase** — словесні фрази, що описують зв'язок в обох напрямках. Від батьківської сутності до дочірньої фраза зазвичай має форму дієслова або дієслівної фрази (наприклад, "містить", "має", "володіє").

У зворотному напрямку фраза описує належність (наприклад, "належить до", "є частиною"). Ці фрази покращують читабельність діаграми та автоматично використовуються в генерованій документації.

Cardinality визначає кількісні характеристики зв'язку. ERWin використовує нотацію, де для кожної сутності визначається як максимальна кардинальність (один або багато), так і модальність (обов'язкова або опціональна участь). Типові комбінації включають: **One-to-Many** (1:N) — найбільш поширений тип, де один екземпляр батьківської сутності може бути пов'язаний з багатьма екземплярами дочірньої; **One-to-One** (1:1) — рідкісний тип, де зв'язок є взаємно унікальним; **Zero-or-One to Many** — один екземпляр може бути пов'язаний з багатьма, але участь в зв'язку не є обов'язковою; **One to Zero-or-Many** — обов'язкова участь батьківської сутності, але дочірня сутність може існувати без зв'язку.

Вкладка **RI Actions** (Referential Integrity Actions) визначає **правила каскадних операцій**, які автоматично підтримують посилальну цілісність при операціях оновлення та видалення записів. Для операції **DELETE** (видалення запису з батьківської таблиці) доступні наступні опції: **Restrict** або **No Action** — заборонити видалення батьківського запису, якщо існують пов'язані дочірні записи; **Cascade** — автоматично видалити всі пов'язані дочірні записи; **Set Null** — встановити NULL в зовнішньому ключі дочірніх записів (доступно лише якщо зовнішній ключ допускає NULL); **Set Default** — встановити значення за замовчуванням в зовнішньому ключі. Аналогічні опції доступні для операції **UPDATE** (зміна значення первинного ключа батьківської таблиці), хоча на практиці зміна первинних ключів є рідкісною операцією, особливо при використанні суррогатних ключів.

Для зв'язків типу **багато-до-багатьох** використовується спеціалізований **Many-to-Many Relationship Tool**. На логічному рівні моделі такий зв'язок відображається як пряма лінія між двома сутностями. При переході до фізичного рівня або при генерації SQL-скриптів ERWin автоматично трансформує зв'язок M:N у **асоціативну сутність** (associative entity) або **таблицю зв'язку** (junction table). Ця проміжна таблиця містить як мінімум два зовнішні ключі, що посилаються на обидві вихідні таблиці, і ці зовнішні ключі формують складений первинний ключ таблиці зв'язку. Якщо зв'язок має власні атрибути (наприклад, дата початку зв'язку, статус, кількість), вони додаються як додаткові колонки до таблиці зв'язку.

Етап 2: Трансформація в фізичну модель та оптимізація

Після завершення роботи над логічною моделлю та її валідації настає етап трансформації в фізичну модель, яка безпосередньо відображає структуру бази даних в обраній СУБД.

Перемикання на **Physical Model View** виконується через меню **Model → Logical/Physical** або натисканням відповідної кнопки на панелі інструментів. При першому переключенні ERWin виконує автоматичну трансформацію, застосовуючи наступні перетворення до моделі.

Трансформація типів даних: кожен логічний тип даних перетворюється у відповідний фізичний тип цільової СУБД згідно з налаштованими правилами відображення (mapping rules). Наприклад, логічний тип String може бути трансформований у VARCHAR(255) для MySQL, VARCHAR2 для Oracle, або NVARCHAR для SQL Server. Логічний тип Integer може стати INT, BIGINT або SMALLINT залежно від очікуваного діапазону значень. Проектувальник може переглянути та скоригувати ці відображення, вибравши більш підходящі типи з урахуванням специфіки даних та вимог до продуктивності.

Генерація фізичних назв: логічні назви сутностей та атрибутів трансформуються у фізичні назви таблиць та колонок згідно з встановленими правилами іменування. Наприклад, логічна назва сутності "Customer Order" може бути трансформована у "CUSTOMER_ORDERS" (верхній регістр з підкресленнями) або "customer_orders" (нижній регістр) залежно від конвенцій. Правила можуть включати додавання префіксів до таблиць (наприклад, "tbl_") або до первинних ключів (наприклад, закінчення "_id").

Створення індексів: ERWin автоматично створює унікальні індекси для первинних ключів всіх таблиць. Також автоматично створюються неунікальні індекси для зовнішніх ключів, оскільки ці колонки часто використовуються в операціях JOIN та WHERE. Проектувальник може додати додаткові індекси для колонок, які часто використовуються в пошукових запитах або сортуванні.

Реалізація зв'язків: зв'язки між сутностями перетворюються у конструкції зовнішніх ключів з відповідними обмеженнями. Для кожного зв'язку генерується ім'я обмеження зовнішнього ключа (foreign key constraint name), яке зазвичай включає назви обох таблиць. Визначаються правила каскадних операцій у синтаксисі цільової СУБД.

Трансформація зв'язків M:N: зв'язки багато-до-багатьох перетворюються у проміжні таблиці зв'язку з відповідними зовнішніми ключами та складеним первинним ключем. ERWin автоматично генерує назву для таблиці зв'язку, зазвичай комбінуючи назви обох пов'язаних таблиць.

Після автоматичної трансформації проектувальник виконує детальне налаштування фізичних властивостей, які специфічні для обраної СУБД та критично впливають на продуктивність та функціональність бази даних.

Налаштування індексів: для кожної таблиці аналізуються типові запити та патерни доступу до даних. Створюються додаткові індекси для колонок, що часто використовуються в умовах WHERE, JOIN, ORDER BY. Для індексів визначається тип (B-tree, hash, bitmap), порядок колонок у композитних індексах (найбільш селективні колонки першими), опція унікальності, фактор заповнення (fill factor) для оптимізації співвідношення між швидкістю читання та модифікації. Для СУБД, що підтримують спеціалізовані типи індексів, можуть бути створені повнотекстові індекси для текстового пошуку, просторові індекси для геоданих, індекси на виразах.

Визначення обмежень цілісності: окрім автоматично створених обмежень первинних та зовнішніх ключів, додаються додаткові обмеження для забезпечення валідності даних. **Check constraints** визначають умови, яким повинні відповідати значення колонок, наприклад, вік повинен бути більше нуля, дата завершення повинна бути пізніше дати початку, статус повинен належати до визначеного набору значень. **Unique constraints** забезпечують унікальність комбінацій колонок, що не є первинним ключем, наприклад, унікальність комбінації email та домену. **Default constraints** визначають значення за замовчуванням для колонок, що спрощує вставку даних та забезпечує консистентність.

Створення тригерів: для реалізації складної бізнес-логіки, яка не може бути виражена через декларативні обмеження, створюються тригери. Тригери можуть виконувати валідацію даних з перевіркою умов, що залучають множинні таблиці, автоматичне обчислення та оновлення похідних значень, ведення журналів аудиту для відслідковування змін у критичних таблицях, забезпечення специфічних правил посилальної цілісності, які не підтримуються стандартними зовнішніми ключами. ERWin дозволяє визначати тригери на рівні моделі з прив'язкою до конкретних таблиць та подій (INSERT, UPDATE, DELETE).

Оптимізація структури таблиць: аналізується структура таблиць з точки зору продуктивності. Для великих таблиць може бути визначене **секціонування** (partitioning) — розділення таблиці на менші фізичні частини за певним критерієм (діапазон дат, список значень, хеш-функція). Секціонування покращує продуктивність запитів, які звертаються лише до підмножини даних, та спрощує адміністрування великих таблиць. Для окремих випадків може бути виконана контрольована **денормалізація** — свідоме порушення нормальних форм для

підвищення продуктивності читання даних шляхом зменшення кількості JOIN-операцій. Денормалізація може включати дублювання атрибутів з батьківських таблиць у дочірні, створення агрегованих полів з попередньо обчисленими значеннями, об'єднання таблиць з зв'язком 1:1.

Етап 3: Генерація SQL-скриптів та створення бази даних

Завершальним етапом проектування є генерація SQL-коду для створення фізичної бази даних та виконання цього коду на цільовому сервері СУБД.

Процес генерації SQL розпочинається з виклику **Forward Engineer → Schema Generation**. Відкривається майстер прямого інжинірингу, який проводить проектувальника через процес налаштування параметрів генерації.

На першому кроці вибираються **об'єкти для генерації**. Можна вибрати всю модель або окремі схеми, таблиці, представлення. Визначається, які типи об'єктів повинні бути включені: таблиці, індекси, первинні ключі, зовнішні ключі, check constraints, default constraints, тригери, збережені процедури, представлення. Для кожного типу об'єктів можна окремо визначити, чи повинні генеруватися оператори CREATE (створення нових об'єктів) та/або DROP (видалення існуючих об'єктів перед створенням нових).

На наступному кроці налаштовуються **параметри генерації SQL**. Визначається порядок генерації операторів — зазвичай спочатку створюються таблиці, потім індекси, потім обмеження зовнішніх ключів (оскільки вони вимагають існування обох таблиць). Налаштовується формат коду: відступи, перенесення рядків, використання великих або малих літер для SQL-ключових слів. Визначається, чи потрібно включати коментарі в згенерований код: коментарі з описами таблиць та колонок з моделі, роздільні коментарі між секціями скрипта. Встановлюється, чи потрібно генерувати команди для контролю транзакцій (BEGIN TRANSACTION, COMMIT) та обробки помилок.

Система генерує SQL-скрипт, який можна переглянути в інтегрованому редакторі. Проектувальник може вручну відредагувати згенерований код при необхідності, хоча це рідко потрібно при правильному налаштуванні моделі. Скрипт можна зберегти у файл для подальшого використання або безпосередньо виконати на сервері СУБД, якщо налаштовано підключення.

Альтернативно до генерації та виконання SQL-скриптів, ERWin підтримує **пряме створення бази даних** через команду **Generate Database**. Ця функція встановлює підключення до сервера СУБД, використовуючи налаштовані параметри аутентифікації, та безпосередньо виконує всі необхідні операції для створення структури бази даних без проміжного збереження SQL-коду у файл.

Етап 4: Синхронізація моделі з базою даних

У процесі ітеративної розробки модель еволюціонує, і необхідно періодично синхронізувати структуру реальної бази даних зі зміненою моделлю.

Complete Compare виконує детальне порівняння моделі з базою даних, виявляючи всі структурні відмінності. Система генерує звіт, що включає таблиці, які існують в моделі, але відсутні в базі даних (потрібно створити), таблиці, які існують в базі даних, але відсутні в моделі (можливо, потрібно видалити або додати в модель), для існуючих таблиць — додані, видалені або змінені колонки, відмінності в індексах, обмеженнях, тригерах.

На основі виявлених відмінностей ERWin генерує **ALTER-скрипти**, які містять SQL-оператори для приведення бази даних у відповідність з моделлю. Ці скрипти є інтелектуальними та враховують залежності між об'єктами — наприклад, перед видаленням колонки спочатку видаляються обмеження та індекси, що на неї посилаються. Проектувальник може переглянути згенеровані ALTER-оператори, вибірково виключити певні зміни, відредагувати окремі оператори, додати додаткові команди для міграції існуючих даних. Після затвердження змін скрипти можуть бути виконані на сервері СУБД.

Етап 5: Документування проекту

Завершальним аспектом проектування є створення комплексної документації, яка описує структуру бази даних та проектні рішення.

ERWin надає **Report Designer** для генерації різноманітної документації. Створюється **Entity Relationship Diagram Report** — набір діаграм з різним рівнем деталізації, від високорівневого огляду основних сутностей до детальних діаграм окремих підсистем. Генерується **Complete Data Dictionary** — повний словник даних з детальним описом всіх таблиць, колонок, індексів, обмежень, включаючи типи даних, обов'язковість, значення за замовчуванням, валідаційні правила.

Створюється **Schema Documentation** з описом структури схем бази даних, організації таблиць по функціональних областях, опису ключових зв'язків між таблицями. Додається **Business Rules Documentation**, яка описує бізнес-правила, реалізовані через обмеження, тригери та збережені процедури. Формується **Change Log** — історія змін моделі з датами, авторами та описом модифікацій.

Документація може бути згенерована в форматі HTML для публікації на внутрішньому веб-сайті команди, PDF для друку та розповсюдження серед зацікавлених сторін, RTF/DOCX для редагування та включення в проектну документацію, XML для подальшої автоматизованої обробки або інтеграції з іншими інструментами.

Правильно підготовлена документація є критично важливою для підтримки бази даних, навчання нових членів команди, передачі знань, аудиту та сертифікації системи.

19. ПРОЕКТУВАННЯ БАЗИ ДАНИХ У СУБД SQLITE

SQLite представляє собою унікальну реалізацію концепції реляційної системи управління базами даних, яка відрізняється від традиційних клієнт-серверних СУБД своєю архітектурою та філософією використання. Розроблена Річардом Хіппом (D. Richard Hipp) у 2000 році, SQLite є вбудованою бібліотекою, яка надає повнофункціональний SQL-движок безпосередньо в додаток без необхідності окремого серверного процесу.

Архітектурні характеристики та філософія SQLite

Serverless архітектура є визначальною характеристикою SQLite. На відміну від традиційних СУБД, таких як PostgreSQL, MySQL або Oracle, які працюють як окремі серверні процеси, до яких клієнтські додатки підключаються через мережу або локальні сокети, SQLite працює безпосередньо в процесі додатка як бібліотека. База даних читається та записується безпосередньо з дискових файлів без проміжного серверного процесу. Ця архітектура усуває складність встановлення, налаштування та адміністрування окремого сервера бази даних, роблячи SQLite ідеальним вибором для вбудованих систем, мобільних додатків, настільних програм та інших сценаріїв, де складність клієнт-серверної архітектури є надмірною.

Zero-configuration принцип означає, що SQLite не потребує жодного налаштування або адміністрування. Немає конфігураційних файлів для редагування, немає серверних процесів для запуску, немає користувачів та дозволів для налаштування (на рівні СУБД, хоча файлова система може контролювати доступ до файлу бази даних). Додаток просто відкриває файл бази даних та починає працювати з ним. Якщо файл не існує, він створюється автоматично.

Single file база даних зберігає всю структуру та дані в одному звичайному файлі операційної системи. Це кардинально спрощує резервне копіювання (просто скопіювати файл), переміщення бази даних між системами, додавання бази даних до системи контролю версій. Формат файлу є кросплатформенним — база даних, створена на Windows, може бути прочитана на Linux або macOS без конвертації.

Cross-platform сумісність забезпечує роботу SQLite на всіх основних операційних системах та апаратних платформах: Windows, Linux, macOS, iOS,

Android, різноманітні вбудовані операційні системи. Бібліотека написана на C з мінімальними залежностями від зовнішніх бібліотек, що забезпечує високу портабельність.

ACID-compliant транзакції означають, що, незважаючи на простоту архітектури, SQLite повністю підтримує транзакції з гарантіями Atomicity (атомарність — транзакція виконується повністю або не виконується взагалі), Consistency (консистентність — база даних залишається в узгодженому стані), Isolation (ізольованість — паралельні транзакції не впливають одна на одну), Durability (довговічність — після підтвердження транзакції зміни є постійними навіть при збоях системи).

Система типів даних SQLite

SQLite використовує унікальну систему типів, яка відрізняється від традиційних реляційних СУБД. Замість жорсткої статичної типізації, SQLite використовує **динамічну типізацію**, де тип асоціюється не з колонкою, а з конкретним збереженим значенням.

SQLite визначає п'ять **класів зберігання** (storage classes), які описують формат зберігання значень на диску:

NULL представляє відсутність значення або невизначене значення. Це спеціальний клас зберігання для NULL-значень, які використовуються для позначення відсутніх або невідомих даних.

INTEGER зберігає цілі числа зі знаком. SQLite автоматично вибирає оптимальний розмір для зберігання залежно від величини числа: 0 байт для нуля, 1 байт для чисел від -128 до 127, 2 байти для чисел від -32768 до 32767, 3 байти, 4 байти, 6 байт або 8 байт для більших чисел. Це динамічне визначення розміру економить місце для малих чисел, забезпечуючи водночас підтримку великих значень до 2^63-1.

REAL зберігає числа з плаваючою точкою у форматі IEEE 754 з подвійною точністю (64 біти, 8 байт). Це забезпечує точність приблизно 15 десяткових цифр та діапазон від приблизно 10^-308 до 10^308.

TEXT зберігає текстові рядки в одному з підтримуваних кодувань: UTF-8, UTF-16BE (big-endian) або UTF-16LE (little-endian). Кодування визначається при створенні бази даних, за замовчуванням використовується UTF-8. SQLite підтримує рядки необмеженої довжини (обмежені лише доступною пам'яттю та налаштуваними лімітами).

BLOB (Binary Large Object) зберігає двійкові дані в точності в тому вигляді, в якому вони були введені. Це може бути що завгодно: зображення,

аудіо, відео, зашифровані дані, серіалізовані об'єкти. Як і TEXT, BLOB підтримує практично необмежені розміри.

Type affinity (спорідненість типів) є концепцією, яка забезпечує сумісність SQLite з традиційними SQL-СУБД. Кожна колонка має оголошений тип (наприклад, VARCHAR, INTEGER, DECIMAL), який визначає type affinity — рекомендований клас зберігання для значень у цій колонці. SQLite визначає п'ять type affinities: TEXT, NUMERIC, INTEGER, REAL, BLOB. Коли значення вставляється в колонку, SQLite намагається перетворити його у клас зберігання, відповідний type affinity колонки, але якщо перетворення неможливе без втрати інформації, значення зберігається в його оригінальному класі зберігання. Це забезпечує гнучкість, дозволяючи зберігати різні типи даних в одній колонці, зберігаючи водночас очікувану поведінку для стандартних SQL-операцій.

Процес створення бази даних та таблиць в SQLite

Початок роботи з SQLite надзвичайно простий завдяки принципу zero-configuration. База даних автоматично створюється при першому з'єднанні з неіснуючим файлом.

Створення та підключення до бази даних виконується через команду **sqlite3** з іменем файлу бази даних:

bash

```
sqlite3 university.db
```

Ця команда запускає інтерактивний командний інтерпретатор SQLite. Якщо файл `university.db` не існує, він буде створений автоматично. Якщо файл існує, буде відкрито існуючу базу даних.

Створення таблиць виконується стандартною SQL-командою CREATE TABLE з визначенням структури таблиці. Розглянемо детальний приклад створення таблиці для зберігання інформації про студентів:

sql

```
CREATE TABLE Students (
    StudentID INTEGER PRIMARY KEY AUTOINCREMENT,
    FirstName TEXT NOT NULL,
    LastName TEXT NOT NULL,
    DateOfBirth DATE,
    Email TEXT UNIQUE,
    PhoneNumber TEXT,
    EnrollmentDate DATE DEFAULT CURRENT_DATE,
    GPA REAL CHECK(GPA >= 0.0 AND GPA <= 4.0),
    IsActive INTEGER DEFAULT 1
```

);

Аналізуючи це визначення, ми бачимо декілька ключових елементів. **StudentID** визначений як INTEGER PRIMARY KEY AUTOINCREMENT, що робить його первинним ключем з автоматичною генерацією унікальних значень. Кожен новий запис отримує значення на одиницю більше за максимальне існуюче. **FirstName** та **LastName** визначені як TEXT NOT NULL, що означає, що ці поля обов'язкові для заповнення та не можуть містити NULL-значення. **Email** має обмеження UNIQUE, що гарантує, що два студенти не можуть мати однакову адресу електронної пошти. **EnrollmentDate** має значення за замовчуванням CURRENT_DATE, яке автоматично встановлює поточну дату при створенні запису, якщо явно не вказано інше значення. **GPA** (Grade Point Average) має CHECK constraint, який забезпечує, що значення знаходиться в валідному діапазоні від 0.0 до 4.0. **IsActive** використовує INTEGER для зберігання булевого значення (SQLite не має окремого типу BOOLEAN, тому зазвичай використовується INTEGER з значеннями 0 для false та 1 для true).

Створення таблиці для курсів демонструє використання зовнішніх ключів:

```
sql
CREATE TABLE Courses (
    CourseID INTEGER PRIMARY KEY AUTOINCREMENT,
    CourseCode TEXT NOT NULL UNIQUE,
    CourseName TEXT NOT NULL,
    Credits INTEGER NOT NULL CHECK(Credits > 0 AND Credits <= 6),
    DepartmentID INTEGER,
    Description TEXT,
    FOREIGN KEY (DepartmentID) REFERENCES Departments(DepartmentID)
        ON DELETE SET NULL
        ON UPDATE CASCADE
);
```

У цьому визначенні **CourseCode** має як обмеження NOT NULL, так і UNIQUE, забезпечуючи, що кожен курс має унікальний код. **Credits** має CHECK constraint, який обмежує кількість кредитів розумним діапазоном. **FOREIGN KEY** встановлює посилання на таблицю Departments, з правилами каскадних операцій: ON DELETE SET NULL означає, що при видаленні департаменту значення DepartmentID в пов'язаних курсах буде встановлено в NULL замість видалення курсів; ON UPDATE CASCADE означає, що при зміні DepartmentID в

таблиці Departments відповідні значення автоматично оновляться в таблиці Courses.

Важливе зауваження: підтримка зовнішніх ключів за замовчуванням **вимкнена** в SQLite з метою забезпечення зворотної сумісності. Для активації підтримки зовнішніх ключів необхідно виконати команду PRAGMA при кожному підключенні до бази даних:

```sql
PRAGMA foreign_keys = ON;
```

Таблиця зв'язку для реалізації зв'язку багато-до-багатьох між студентами та курсами демонструє використання складеного первинного ключа та множинних зовнішніх ключів:

```sql
CREATE TABLE Enrollments (
    EnrollmentID INTEGER PRIMARY KEY AUTOINCREMENT,
    StudentID INTEGER NOT NULL,
    CourseID INTEGER NOT NULL,
    EnrollmentDate DATE DEFAULT CURRENT_DATE,
    Grade TEXT CHECK(Grade IN ('A', 'B', 'C', 'D', 'F', 'W', 'I')),
    Status TEXT DEFAULT 'Active' CHECK(Status IN ('Active', 'Completed', 'Dropped')),
    FOREIGN KEY (StudentID) REFERENCES Students(StudentID)
        ON DELETE CASCADE
        ON UPDATE CASCADE,
    FOREIGN KEY (CourseID) REFERENCES Courses(CourseID)
        ON DELETE CASCADE
        ON UPDATE CASCADE,
    UNIQUE(StudentID, CourseID, EnrollmentDate)
);
```

Тут використовується UNIQUE constraint на комбінацію StudentID, CourseID та EnrollmentDate, що дозволяє студенту записатися на той самий курс в різні семестри, але запобігає дубльованим реєстраціям в один день. **Grade** має CHECK constraint з списком допустимих значень оцінок. **Status** також обмежений визначеним набором значень. Обидва зовнішні ключі налаштовані з ON DELETE CASCADE, що означає автоматичне видалення записів про реєстрацію при видаленні студента або курсу, що є логічною поведінкою для цього типу зв'язку.

Створення та управління індексами

Індекси є критично важливими для продуктивності запитів, особливо в таблицях з великою кількістю записів. SQLite автоматично створює унікальний індекс для кожного первинного ключа та для кожного UNIQUE constraint, але додаткові індекси повинні створюватися явно.

Створення простого індексу для прискорення пошуку студентів за прізвищем:

sql

```
CREATE INDEX idx_students_lastname
ON Students(LastName);
```

Композитний індекс для оптимізації запитів, які фільтрують або сортують за кількома колонками:

sql

```
CREATE INDEX idx_enrollments_student_course
ON Enrollments(StudentID, CourseID);
```

Порядок колонок в композитному індексі є важливим. Індекс ефективний для запитів, які використовують перші колонки індексу. Вказаний індекс оптимізує запити по StudentID та по комбінації StudentID+CourseID, але не оптимізує запити лише по CourseID.

Унікальний індекс забезпечує як прискорення пошуку, так і унікальність значень:

sql

```
CREATE UNIQUE INDEX idx_students_email
ON Students(Email);
```

Індекс з частковою фільтрацією (partial index) включає лише записи, що відповідають певній умові, що робить індекс меншим та ефективнішим для специфічних запитів:

sql

```
CREATE INDEX idx_active_students
ON Students(LastName)
WHERE IsActive = 1;
```

Цей індекс включ

ТЕМА 8. ФІЗИЧНЕ ПРОЕКТУВАННЯ БАЗ ДАНИХ

1. Апаратні та програмні складові
2. Особливості OLTP, DSS та OLAP систем
3. Зберігання даних

4. Індексація
5. Кластеризація
6. Розподіл
7. Методи доступу

1. АПАРАТНІ ТА ПРОГРАМНІ СКЛАДОВІ

Фізичне проектування баз даних становить завершальний етап проектування системи управління базами даних і передбачає перетворення логічної моделі даних у фізичну структуру, оптимізовану для конкретної апаратно-програмної платформи. На відміну від логічного проектування, яке зосереджується на структурі даних незалежно від технічних деталей реалізації, фізичне проектування враховує специфіку конкретної СУБД, характеристики апаратного забезпечення, особливості навантаження системи та вимоги до продуктивності. Метою фізичного проектування є досягнення оптимального балансу між швидкістю обробки запитів, ефективністю використання пам'яті, надійністю зберігання даних та складністю адміністрування системи.

1.1. Апаратна інфраструктура

Апаратне забезпечення становить фундамент будь-якої системи управління базами даних і безпосередньо впливає на її продуктивність. Центральний процесор визначає швидкість обробки запитів та виконання складних обчислень, тому для систем з інтенсивною обробкою транзакцій рекомендується використовувати багатоядерні процесори з високою тактовою частотою. Оперативна пам'ять відіграє критичну роль у продуктивності бази даних, оскільки сучасні СУБД активно використовують кешування даних та індексів у пам'яті для мінімізації звернень до дискових систем. Для великих баз даних рекомендується обсяг оперативної пам'яті не менше п'ятнадцяти-двадцяти відсотків від загального обсягу бази даних.

Системи зберігання даних поділяються на кілька категорій за швидкістю доступу та вартістю. Традиційні жорсткі диски забезпечують великий обсяг зберігання за помірну ціну, але характеризуються повільним часом доступу через механічну природу зчитування. Твердотільні накопичувачі надають значно вищу швидкість читання та запису завдяки відсутності рухомих частин, що робить їх оптимальним вибором для критичних до продуктивності додатків. Гібридні системи зберігання поєднують переваги обох технологій, розміщуючи часто використовувані дані на швидких SSD, а архівні дані на ємних HDD.

Мережева інфраструктура набуває особливого значення у розподілених системах баз даних. Пропускна здатність мережі, затримки передачі даних та

надійність з'єднання безпосередньо впливають на продуктивність розподілених запитів та реплікації даних. Для критичних систем застосовуються високошвидкісні мережі з пропускною здатністю десять гігабіт на секунду та вище, резервовані канали зв'язку та спеціалізовані мережеві протоколи для зменшення затримок.

1.2. Програмне забезпечення

Вибір системи управління базами даних становить одне з найважливіших рішень на етапі фізичного проектування. Сучасні СУБД поділяються на реляційні, документо-орієнтовані, графові, колоночні та інші типи, кожен з яких оптимізований для певних типів навантаження. Реляційні СУБД, такі як PostgreSQL, Oracle Database та Microsoft SQL Server, залишаються домінуючим вибором для транзакційних систем завдяки підтримці ACID властивостей, розвиненим засобам забезпечення цілісності даних та зрілим інструментам адміністрування.

Операційна система, на якій функціонує СУБД, впливає на продуктивність через механізми управління пам'яттю, планування процесів та роботи з файловою системою. Unix-подібні системи традиційно вважаються оптимальним вибором для серверів баз даних завдяки стабільності, ефективному управлінню ресурсами та розвиненим засобам моніторингу. Вибір файлової системи також має значення: сучасні журналовані файлові системи забезпечують кращу надійність та швидкодію порівняно зі старими альтернативами.

Додаткове програмне забезпечення включає системи моніторингу продуктивності, інструменти резервного копіювання та відновлення, засоби автоматизації адміністративних завдань. Системи моніторингу дозволяють відстежувати використання ресурсів, виявляти вузькі місця та прогнозувати необхідність масштабування. Інструменти резервного копіювання повинні забезпечувати як повне, так і інкрементне копіювання з мінімальним впливом на роботу продуктивної системи.

1.3. Архітектура ANSI/SPARC

Трирівнева архітектура ANSI/SPARC становить концептуальну основу для розуміння взаємозв'язку між різними рівнями абстракції в системах баз даних. Зовнішній рівень представляє уявлення даних для кінцевих користувачів та додатків, концептуальний рівень описує логічну структуру всієї бази даних, а внутрішній або фізичний рівень визначає фактичне зберігання даних на носіях.

Фізичний рівень оперує поняттями файлів, сторінок, записів та методів доступу. Саме на цьому рівні визначається організація даних на диску, структури індексів, методи стиснення даних та інші технічні деталі реалізації. Незалежність

між рівнями дозволяє змінювати фізичну організацію даних без впливу на логічну схему та прикладні програми, що забезпечує гнучкість системи та спрощує її еволюцію.

2. ОСОБЛИВОСТІ OLTP, DSS ТА OLAP СИСТЕМ

2.1. OLTP системи

Системи оперативної обробки транзакцій характеризуються великою кількістю коротких транзакцій, які виконують операції вставки, оновлення та видалення окремих записів. Типові приклади включають системи банківських операцій, бронювання квитків, обробки замовлень у електронній комерції. Ключовими вимогами до OLTP систем є висока швидкість відгуку на окремі запити, підтримка одночасної роботи великої кількості користувачів, забезпечення цілісності даних та надійне відновлення після збоїв.

Фізичне проектування OLTP систем орієнтується на оптимізацію операцій запису та читання невеликих обсягів даних. Нормалізація таблиць до третьої або вищої нормальної форми зменшує надмірність даних та спрощує підтримку цілісності при оновленнях. Індекси створюються на колонках, які часто використовуються у предикатах WHERE та умовах з'єднання таблиць, але надмірна індексація уникається через негативний вплив на швидкість операцій вставки та оновлення.

Управління транзакціями в OLTP системах базується на механізмах блокування або багатоверсійного контролю одночасності. Короткі транзакції мінімізують час утримання блокувань і зменшують ймовірність виникнення тупикових ситуацій. Журналювання транзакцій забезпечує можливість відновлення бази даних до консистентного стану після апаратних або програмних збоїв. Оптимізація продуктивності OLTP систем часто включає розподіл навантаження між кількома серверами, використання реплікації для читання та регулярне обслуговування індексів.

2.2. DSS системи

Системи підтримки прийняття рішень призначені для аналізу великих обсягів історичних даних з метою виявлення тенденцій, закономірностей та підготовки аналітичних звітів. На відміну від OLTP систем, DSS характеризуються відносно невеликою кількістю одночасних користувачів, але складними запитами, які обробляють мільйони записів та виконують агрегування, групування та з'єднання багатьох таблиць.

Фізичне проектування DSS систем передбачає денормалізацію схеми для зменшення кількості з'єднань таблиць при виконанні аналітичних запитів.

Матеріалізовані представлення зберігають попередньо обчислені результати складних запитів, що суттєво прискорює отримання аналітичних звітів. Використання компресії даних дозволяє зменшити обсяг зберігання та прискорити читання великих обсягів інформації з диска.

Партиціонування таблиць за часовими або географічними ознаками дозволяє обмежити обсяг даних, які обробляються типовими аналітичними запитами. Наприклад, таблиця продажів може бути розділена за місяцями або роками, що дозволяє швидко отримувати звіти за конкретні періоди без сканування всієї таблиці. Спеціалізовані індекси для аналітичних запитів включають bitmap індекси для колонок з невеликою кардинальністю та колоночні індекси для агрегуючих запитів.

2.3. OLAP системи

Аналітичні системи онлайн обробки базуються на багатовимірній моделі даних, яка представляє інформацію у вигляді гіперкубів з вимірами та мірами. Виміри представляють перспективи аналізу, такі як час, географія, продукти, клієнти, тоді як міри містять числові показники, які аналізуються, наприклад обсяги продажів, прибуток, кількість операцій.

Фізична реалізація OLAP систем може використовувати реляційну базу даних у формі схеми зірки або сніжинки, або спеціалізовані багатовимірні СУБД. Схема зірки включає центральну таблицю фактів, яка містить числові міри та зовнішні ключі до таблиць вимірів. Таблиці вимірів містять описову інформацію та ієрархії для агрегування даних на різних рівнях деталізації. Схема сніжинки нормалізує таблиці вимірів, розділяючи їх на кілька пов'язаних таблиць для зменшення надмірності.

OLAP операції включають зрізи для вибору підмножини даних за фіксованим значенням виміру, обертання для зміни перспективи аналізу, деталізацію для переходу на нижчий рівень ієрархії виміру та узагальнення для переходу на вищий рівень. Попереднє обчислення агрегатів на різних рівнях деталізації прискорює виконання аналітичних запитів, але вимагає додаткового простору для зберігання та часу на оновлення при завантаженні нових даних.

Гібридні OLAP системи поєднують переваги реляційного зберігання для детальних даних та багатовимірного зберігання для агрегатів. Така архітектура дозволяє досягти балансу між гнучкістю, продуктивністю та ефективністю використання ресурсів. Оптимізація продуктивності OLAP систем включає інтелектуальне вибіркове обчислення агрегатів, компресію багатовимірних структур та кешування результатів запитів.

3. ЗБЕРІГАННЯ ДАНИХ

3.1. Організація файлів

Фізичне зберігання даних базується на структурі файлів, які поділяються на сторінки або блоки фіксованого розміру. Типовий розмір сторінки становить чотири, вісім або шістнадцять кілобайт, вибір якого залежить від характеристик навантаження системи. Менші сторінки ефективніші для транзакційних систем з частими оновленнями окремих записів, тоді як більші сторінки краще підходять для аналітичних систем з послідовним читанням великих обсягів даних.

Записи в межах сторінки можуть організовуватися у неупорядкованому вигляді, коли нові записи додаються у перший доступний простір, або в упорядкованому вигляді згідно з ключем сортування. Неупорядковане зберігання забезпечує швидку вставку нових записів, але вимагає повного сканування сторінки для пошуку конкретного запису. Упорядковане зберігання уповільнює вставку через необхідність підтримання порядку, але прискорює пошук та дозволяє ефективне виконання запитів з діапазонними умовами.

Структури переповнення дозволяють динамічно розширювати таблиці при вставці нових записів. Коли основна сторінка заповнюється, додаткові записи розміщуються в окремих сторінках переповнення, пов'язаних з основною сторінкою через вказівники. Періодична реорганізація таблиць об'єднує основні сторінки та сторінки переповнення, усуваючи фрагментацію та відновлюючи оптимальну продуктивність доступу до даних.

3.2. Методи розміщення записів

Запис фіксованої довжини спрощує управління простором на сторінці, оскільки розмір кожного запису відомий заздалегідь. Це дозволяє використовувати прості математичні обчислення для знаходження позиції запису на сторінці за його порядковим номером. Однак для таблиць з колонками змінної довжини, такими як текстові поля, використання записів фіксованої довжини призводить до неефективного використання простору через необхідність резервування максимально можливого розміру для кожного поля.

Записи змінної довжини дозволяють ефективніше використовувати дисковий простір для таблиць з текстовими полями або необов'язковими атрибутами. Кожен запис містить заголовок зі інформацією про розмір запису та зміщення окремих полів. Управління простором на сторінці ускладнюється через фрагментацію, яка виникає при видаленні або оновленні записів. Каталог слотів

на кожній сторінці відстежує позиції активних записів та вільні простори, дозволяючи ефективно повторно використовувати простір після видалення записів.

Стиснення даних зменшує обсяг дискового простору та може прискорити операції читання за рахунок зменшення обсягу даних, які передаються з диска в пам'ять. Алгоритми стиснення варіюються від простого словникового кодування повторюваних значень до складних статистичних методів. Вибір методу стиснення залежить від характеристик даних, співвідношення між швидкістю стиснення та розпакування, ступенем компресії. Для аналітичних систем колоночне зберігання в поєднанні зі стисненням забезпечує високу ефективність для запитів, які обробляють окремі атрибути великої кількості записів.

3.3. Управління буферним пулом

Буферний пул становить область оперативної пам'яті, яка використовується для кешування сторінок бази даних. Розмір буферного пулу критично впливає на продуктивність системи, оскільки доступ до даних у пам'яті на кілька порядків швидший за доступ до диска. Оптимальний розмір буферного пулу залежить від загального обсягу бази даних, робочого набору даних, які активно використовуються, та доступної оперативної пам'яті на сервері.

Стратегії заміщення сторінок у буферному пулі визначають, які сторінки видаляються з кешу при необхідності завантаження нових сторінок. Алгоритм найменш нещодавно використаної сторінки видаляє сторінку, до якої найдовше не було звернень, виходячи з припущення, що сторінки, які не використовувалися довго, навряд чи будуть потрібні найближчим часом. Модифікований алгоритм враховує частоту використання сторінки, надаючи перевагу сторінкам, до яких часто звертаються.

Попереднє завантаження сторінок дозволяє покращити продуктивність для предбачуваних патернів доступу. При послідовному скануванні таблиці система може завантажувати наступні сторінки в буферний пул ще до того, як вони знадобляться, ховаючи латентність дискових операцій. Аналогічно, при виконанні запитів з індексним доступом можна попередньо завантажувати листкові сторінки індексу, покращуючи продуктивність операцій пошуку.

4. ІНДЕКСАЦІЯ

4.1. Призначення та принципи індексів

Індекси становлять допоміжні структури даних, які прискорюють пошук записів за значеннями певних атрибутів без необхідності повного сканування таблиці. Концептуально індекс схожий на предметний покажчик у книзі, який

дозволяє швидко знайти сторінки з потрібною інформацією. Індекс містить відсортовані значення індексованих колонок разом з вказівниками на відповідні записи у таблиці, що дозволяє ефективно локалізувати необхідні дані.

Переваги використання індексів включають драматичне прискорення запитів з умовами пошуку за індексованими колонками, покращення продуктивності операцій сортування та з'єднання таблиць, забезпечення унікальності значень. Однак індекси також мають недоліки: вони займають додатковий дисковий простір, уповільнюють операції вставки, оновлення та видалення через необхідність підтримки індексних структур, потребують періодичного обслуговування для запобігання деградації продуктивності через фрагментацію.

Селективність індексу визначає ефективність його використання для конкретного запиту. Високоселективний індекс на колонці з великою кількістю унікальних значень дозволяє швидко відфільтрувати переважну більшість записів. Низькоселективний індекс на колонці з малою кількістю різних значень може виявитися менш ефективним за повне сканування таблиці, особливо коли запит вибирає значну частину записів. Оптимізатор запитів аналізує селективність індексів та статистику розподілу даних для вибору оптимального плану виконання запиту.

4.2. B-дерева та їх варіанти

B-дерево становить найпоширенішу індексну структуру в реляційних СУБД завдяки збалансованості, ефективності для діапазонних запитів та хорошим характеристикам при операціях вставки та видалення. B-дерево організоване як багаторівневе дерево, де кожен вузол містить кілька ключів та вказівників на дочірні вузли. Коренева та внутрішні вершини містять ключі для навігації по дереву, тоді як листкові вершини містять фактичні значення індексу та вказівники на записи таблиці.

Збалансованість B-дерева гарантує, що всі шляхи від кореня до листків мають однакову довжину, що забезпечує предбачувану продуктивність операцій пошуку незалежно від конкретних значень ключів. Кількість дискових звернень для пошуку запису пропорційна висоті дерева, яка зростає логарифмічно відносно кількості записів. Типове B-дерево для таблиці з мільйонами записів має висоту три або чотири рівні, що дозволяє знаходити будь-який запис за декілька дискових операцій.

B+-дерево, варіант базової структури, зберігає всі значення індексу лише в листкових вершинах, а внутрішні вершини містять тільки ключі для навігації. Листкові вершини пов'язані між собою послідовними вказівниками, утворюючи

впорядкований список всіх значень індексу. Така організація особливо ефективна для діапазонних запитів, оскільки після знаходження початкового значення діапазону система може послідовно проходити листкові вершини до кінця діапазону без повернення до внутрішніх вершин дерева.

4.3. Хеш-індекси та інші структури

Хеш-індекси використовують хеш-функцію для відображення значень індексованої колонки в позиції у хеш-таблиці, де зберігаються вказівники на відповідні записи. Ідеальна хеш-функція рівномірно розподіляє значення по всіх позиціях хеш-таблиці, мінімізуючи колізії. Основна перевага хеш-індексів полягає в константній часовій складності операцій пошуку за точним співпадінням значення. Однак хеш-індекси неефективні для діапазонних запитів, оскільки близькі значення відображаються в непередбачувані позиції хеш-таблиці.

Bitmap індекси ефективні для колонок з невеликою кількістю різних значень, таких як стать, тип продукту, статус замовлення. Bitmap індекс створює окремий бітовий вектор для кожного можливого значення колонки, де кожен біт відповідає одному запису таблиці і встановлюється в одиницю, якщо запис має відповідне значення. Bitmap індекси дозволяють ефективно виконувати логічні операції над умовами пошуку за допомогою побітових операцій AND, OR та NOT, що особливо корисно для аналітичних запитів з багатьма умовами фільтрації.

Повнотекстові індекси призначені для пошуку слів та фраз у текстових полях. Інвертований індекс відображає кожне унікальне слово в списку документів, де воно зустрічається, разом з позиціями слова в документі. Повнотекстові індекси підтримують складні операції пошуку, включаючи пошук за формою слова, синоніми, близькість слів, релевантне ранжування результатів. Морфологічний аналіз та лематизація дозволяють знаходити документи, які містять різні форми шуканого слова.

4.4. Композитні та покриваючі індекси

Композитний індекс створюється на кількох колонках і дозволяє ефективно обробляти запити з умовами на всіх або перших колонках індексу. Порядок колонок у композитному індексі критично впливає на його ефективність для різних запитів. Індекс на колонках прізвище, ім'я ефективний для пошуку за прізвищем або за прізвищем і ім'ям одночасно, але неефективний для пошуку тільки за ім'ям. Правило визначення оптимального порядку колонок полягає в розміщенні найбільш селективних колонок на перших позиціях.

Покриваючий індекс містить всі колонки, які необхідні для виконання запиту, що дозволяє отримати результат без звернення до самої таблиці. Для запиту, який вибирає ім'я та прізвище клієнтів з певного міста, індекс на колонках місто, прізвище, ім'я виявиться покриваючим. Використання покриваючих індексів суттєво зменшує кількість дискових операцій та значно прискорює виконання запитів. Однак покриваючі індекси займають більше місця та уповільнюють операції оновлення, тому їх слід створювати вибірково для найбільш критичних запитів.

Частковий індекс створюється тільки для підмножини записів таблиці, які задовольняють певну умову. Наприклад, індекс на активних замовленнях виключає завершені замовлення, зменшуючи розмір індексу та покращуючи продуктивність для запитів, які працюють тільки з активними замовленнями. Часткові індекси особливо корисні для великих таблиць, де типові запити працюють з невеликою частиною даних, яка може бути чітко визначена умовою фільтрації.

5. КЛАСТЕРИЗАЦІЯ

Кластеризація даних передбачає фізичне розміщення записів з пов'язаних таблиць або пов'язаних записів однієї таблиці поруч на диску для мінімізації кількості дискових операцій при виконанні запитів, які обробляють пов'язані дані. Основна ідея полягає в тому, що записи, які часто обробляються разом, повинні зберігатися на одній дисковій сторінці або на близьких сторінках, щоб їх можна було прочитати за мінімальну кількість операцій вводу-виводу.

Кластеризований індекс визначає фізичний порядок зберігання записів у таблиці. На відміну від некластеризованих індексів, які містять вказівники на фізичні позиції записів, кластеризований індекс організує самі дані таблиці в порядку зростання або спадання ключа індексу. Таблиця може мати тільки один кластеризований індекс, оскільки фізичний порядок зберігання записів може бути визначений лише одним способом. Вибір колонки для кластеризованого індексу значно впливає на продуктивність системи і повинен базуватися на аналізі найбільш критичних запитів.

Міжтабличні кластери об'єднують записи з різних таблиць, які часто обробляються разом у запитах з операціями з'єднання. Наприклад, записи з таблиць замовлень та деталей замовлень можуть зберігатися в одному кластері, де кожна група записів містить запис замовлення разом з усіма пов'язаними деталями. Такий підхід дозволяє отримати всю необхідну інформацію про

замовлення за одну дискову операцію замість виконання окремих читань для кожної таблиці.

Основна перевага кластеризації полягає в значному прискоренні запитів, які обробляють пов'язані дані. Діапазонні запити за ключем кластеризованого індексу виконуються особливо ефективно, оскільки записи з послідовними значеннями ключа розміщені фізично поруч і можуть бути прочитані послідовним скануванням. Операції сортування за ключем кластеризації не потребують додаткової обробки, оскільки дані вже впорядковані належним чином. З'єднання таблиць за ключем кластеризації виконуються швидше завдяки локальності даних.

Недоліки кластеризації включають уповільнення операцій вставки та оновлення, оскільки система повинна підтримувати фізичний порядок записів. Вставка нового запису може вимагати переміщення існуючих записів для звільнення місця в потрібній позиції. Оновлення ключа кластеризації може призвести до фізичного переміщення запису на нову позицію. Фрагментація кластеру виникає з часом через операції вставки та видалення, що призводить до розсіювання логічно послідовних записів по різних сторінках та погіршення продуктивності.

Вибір стратегії кластеризації повинен базуватися на детальному аналізі характеристик навантаження системи. Для таблиць з переважно операціями читання та рідкісними оновленнями кластеризація надає суттєві переваги. Для таблиць з інтенсивними операціями вставки некластеризоване зберігання може виявитися більш ефективним. Періодична реорганізація кластерів необхідна для усунення фрагментації та відновлення оптимальної продуктивності.

Вибір ключа кластеризації становить критичне рішення, яке впливає на продуктивність системи протягом усього життєвого циклу бази даних. Ідеальний ключ кластеризації повинен бути стабільним, тобто його значення рідко змінюються після створення запису. Ключ повинен відповідати найбільш частим патернам доступу до даних, особливо діапазонним запитам та операціям сортування. Монотонно зростаючі ключі, такі як послідовні ідентифікатори або часові мітки, мінімізують фрагментацію при вставці нових записів.

Фактор заповнення визначає, скільки вільного простору залишається на кожній сторінці кластеру при його створенні або реорганізації. Нижчий фактор заповнення, наприклад сімдесят відсотків, залишає більше місця для майбутніх вставок без необхідності розділення сторінок, але призводить до менш ефективного використання дискового простору. Вищий фактор заповнення,

наприклад дев'яносто п'ять відсотків, максимізує щільність даних, але збільшує ймовірність розділення сторінок при вставці нових записів.

Моніторинг стану кластеризації дозволяє своєчасно виявляти деградацію продуктивності через фрагментацію. Метрики, такі як середня кількість сторінок на діапазонне сканування, відсоток логічної фрагментації, середня щільність сторінок, допомагають оцінити необхідність реорганізації. Автоматизовані процеси обслуговування можуть періодично аналізувати ці метрики та виконувати реорганізацію кластерів у періоди низького навантаження системи.

6. РОЗПОДІЛ

Розподілені бази даних передбачають зберігання даних на кількох фізично незалежних вузлах, з'єднаних комп'ютерною мережею. Основні мотивації для розподілу включають покращення продуктивності через паралельну обробку запитів, підвищення доступності через надмірність даних, можливість масштабування системи додаванням нових вузлів, розміщення даних близько до користувачів для зменшення мережевих затримок. Розподілені системи вимагають складніших механізмів управління транзакціями, забезпечення узгодженості даних та обробки збоїв окремих вузлів.

Прозорість розподілу визначає ступінь, до якого користувачі та додатки можуть працювати з розподіленою базою даних так, ніби вона є єдиною централізованою системою. Прозорість розташування дозволяє звертатися до даних без знання їх фізичного розміщення. Прозорість фрагментації приховує той факт, що таблиці розділені на частини, розподілені між вузлами. Прозорість реплікації дозволяє системі автоматично вибирати найбільш доступну або швидкодоступну копію даних без явного вказування користувачем.

Архітектури розподілених баз даних варіюються від тісно пов'язаних систем з централізованим управлінням до слабко пов'язаних федеративних систем з автономними вузлами. Системи з розподіленим однорідним середовищем використовують однаковий тип СУБД на всіх вузлах, що спрощує реалізацію глобальних запитів та транзакцій. Гетерогенні розподілені системи інтегрують різні типи СУБД, що вимагає складних протоколів перетворення даних та координації транзакцій між різними платформами.

6.2. Горизонтальна та вертикальна фрагментація

Горизонтальна фрагментація розділяє таблицю на підмножини записів, кожна з яких містить всі колонки, але різні набори рядків. Критерії розділення

можуть базуватися на діапазонах значень певної колонки, хеш-функції від ключа, географічному розташуванні або інших бізнес-правилах. Наприклад, таблиця клієнтів може бути горизонтально розділена за регіонами, де кожен фрагмент містить клієнтів з певної географічної області. Горизонтальна фрагментація ефективна, коли різні користувачі або додатки працюють з різними підмножинами даних.

Вертикальна фрагментація розділяє таблицю на підмножини колонок, де кожен фрагмент містить первинний ключ разом з певним набором атрибутів. Вертикальна фрагментація корисна, коли різні додатки або запити використовують різні набори атрибутів однієї таблиці. Наприклад, таблиця співробітників може бути вертикально розділена на фрагмент з особистими даними та фрагмент з професійною інформацією. Запити, які потребують даних з кількох вертикальних фрагментів, вимагають операцій з'єднання для реконструкції повних записів.

Гібридна фрагментація поєднує горизонтальне та вертикальне розділення для досягнення більш точного відповідності розподілу даних патернам їх використання. Спочатку таблиця може бути вертикально розділена на логічні групи атрибутів, а потім кожен вертикальний фрагмент горизонтально розділяється за певним критерієм. Оптимальна стратегія фрагментації визначається шляхом аналізу частоти та характеру типових запитів, мережевої топології, вимог до продуктивності та доступності.

6.3. Реплікація даних

Реплікація передбачає створення та підтримку кількох копій одних і тих самих даних на різних вузлах розподіленої системи. Основні переваги реплікації включають підвищення доступності даних при відмові окремих вузлів, покращення продуктивності операцій читання через розподіл навантаження між репліками, зменшення мережевих затримок через розміщення реплік близько до користувачів. Однак реплікація ускладнює операції оновлення, оскільки зміни повинні поширюватися на всі репліки для підтримання узгодженості.

Синхронна реплікація гарантує, що всі репліки оновлюються в межах однієї транзакції перед її фіксацією. Це забезпечує сувору узгодженість даних між усіма репліками, але значно збільшує час відгуку на операції запису через необхідність чекати підтвердження від усіх вузлів. Синхронна реплікація чутлива до мережевих затримок та відмов окремих вузлів, які можуть заблокувати виконання транзакцій.

Асинхронна реплікація дозволяє транзакції фіксуватися на первинному вузлі до того, як зміни будуть поширені на вторинні репліки. Це забезпечує

кращу продуктивність операцій запису та меншу чутливість до мережевих проблем, але призводить до тимчасової неузгодженості між репліками. Період затримки реплікації може становити від мілісекунд до хвилин залежно від навантаження системи та мережевих умов. Додатки повинні враховувати можливість читання застарілих даних з вторинних реплік.

6.4. Розподілена обробка запитів

Обробка запитів у розподіленій базі даних включає декомпозицію глобального запиту на підзапити, які виконуються на окремих вузлах, передачу проміжних результатів між вузлами та об'єднання часткових результатів у фінальну відповідь. Оптимізація розподілених запитів повинна враховувати не тільки локальну вартість обробки на кожному вузлі, але й вартість передачі даних по мережі, яка часто домінує в загальному часі виконання.

Стратегії виконання розподілених з'єднань включають відправлення однієї таблиці до вузла з іншою таблицею, розділення обох таблиць за ключем з'єднання та виконання локальних з'єднань на кожному вузлі, використання напівз'єднання для передачі тільки ключів замість повних записів. Вибір оптимальної стратегії залежить від розмірів таблиць, селективності предикатів фільтрації, пропускної здатності мережі та можливості паралельної обробки.

Управління розподіленими транзакціями використовує протоколи дворівневої або трирівневої фіксації для забезпечення атомарності транзакцій, які оновлюють дані на кількох вузлах. Координатор транзакції взаємодіє з учасниками для досягнення консенсусу щодо фіксації або відкату транзакції. Обробка збоїв у розподілених транзакціях вимагає складних протоколів відновлення для забезпечення узгодженості глобального стану бази даних після відмови окремих вузлів або мережевих розділень.

7. МЕТОДИ ДОСТУПУ

Послідовне сканування таблиці читає всі сторінки таблиці від початку до кінця, перевіряючи кожен запис на відповідність умовам запиту. Це найпростіший метод доступу, який не вимагає індексів або спеціальних структур даних. Послідовне сканування ефективне для запитів, які обробляють значну частину або всі записи таблиці, оскільки накладні витрати на навігацію індексом будуть перевищувати вартість простого читання всіх даних.

Оптимізації послідовного сканування включають попереднє завантаження сторінок для приховування латентності дискових операцій, паралельне сканування з розділенням таблиці між кількома потоками обробки, використання векторизованого виконання для обробки багатьох записів одночасно. Сучасні

процесори з SIMD інструкціями дозволяють ефективно застосовувати предикати фільтрації до багатьох записів паралельно, значно прискорюючи послідовне сканування великих таблиць.

Вибір між послідовним скануванням та індексним доступом залежить від селективності запиту, наявності відповідних індексів, розміру таблиці, кешування даних у пам'яті. Порогове значення селективності, після якого послідовне сканування стає ефективнішим за індексний доступ, зазвичай становить від п'яти до двадцяти відсотків записів таблиці, залежно від структури індексу та характеристик апаратного забезпечення.

Індексний пошук використовує індекс для швидкої локалізації записів, які задовольняють умови запиту, без необхідності сканування всієї таблиці. Процес включає навігацію по структурі індексу до листкових вершин, які містять вказівники на відповідні записи таблиці, та читання цих записів для отримання повної інформації. Індексний доступ ефективний для високоселективних запитів, які вибирають невелику частину записів таблиці.

Індексне сканування діапазону використовується для запитів з умовами на інтервал значень індексованої колонки. Після знаходження початкового значення діапазону система послідовно проходить листкові вершини індексу до кінцевого значення, збираючи вказівники на відповідні записи. Впорядкованість листкових вершин у B+-дереві робить індексне сканування діапазону особливо ефективним, оскільки воно вимагає тільки одного проходу по індексу.

Індексне сканування може використовуватися для отримання відсортованих результатів без явної операції сортування, якщо порядок індексу відповідає потрібному порядку сортування. Це особливо корисно для запитів з обмеженням на кількість результатів, оскільки система може зупинити сканування після знаходження необхідної кількості записів. Покриваючі індекси дозволяють уникнути доступу до таблиці взагалі, читаючи всю необхідну інформацію безпосередньо з індексу.

Метод вкладених циклів виконує з'єднання шляхом ітерації по кожному запису зовнішньої таблиці та пошуку відповідних записів у внутрішній таблиці. Цей метод простий у реалізації та ефективний, коли зовнішня таблиця невелика, а внутрішня таблиця має індекс на колонці з'єднання. Вартість виконання пропорційна добутку кількості записів у зовнішній таблиці на вартість пошуку в внутрішній таблиці. Модифікований метод вкладених циклів з блоковим читанням зменшує кількість дискових операцій, обробляючи кілька записів зовнішньої таблиці одночасно.

Метод злиття сортування виконує з'єднання двох попередньо відсортованих наборів записів за ключем з'єднання. Алгоритм одночасно проходить обидва відсортовані набори, порівнюючи ключі та об'єднуючи відповідні записи. Якщо таблиці ще не відсортовані, необхідна попередня фаза сортування, яка може бути дорогою для великих таблиць. Однак метод злиття сортування має лінійну складність відносно сумарного розміру вхідних наборів і добре масштабується для великих таблиць однакового розміру.

Метод хешування будує хеш-таблицю для меншої з таблиць, що з'єднуються, використовуючи ключ з'єднання як хеш-ключ. Потім система сканує більшу таблицю, для кожного запису обчислює хеш-значення ключа з'єднання та шукає відповідні записи в хеш-таблиці. Цей метод ефективний для з'єднань рівності великих таблиць, особливо коли немає відповідних індексів. Гібридне хешування використовує рекурсивне розділення даних на партиції, які помістяться в пам'яті, для обробки з'єднань, які не вміщуються цілком у доступній оперативній пам'яті.

Міжзапитна паралельність дозволяє одночасне виконання кількох незалежних запитів від різних користувачів або додатків. Це базовий рівень паралелізму, який забезпечується багатопоточною архітектурою СУБД та механізмами контролю одночасності. Ефективність міжзапитної паралельності обмежується конфліктами доступу до спільних ресурсів, таких як блокування на таблицях та записах, конкуренція за процесорний час та дисковий ввід-вивід.

Внутрішньозапитна паралельність розділяє виконання одного запиту між кількома процесорами або ядрами для прискорення його обробки. Паралелізм даних розділяє вхідні дані на партиції, які обробляються незалежно різними виконавцями. Наприклад, послідовне сканування великої таблиці може бути розділене між десять потоків, кожен з яких обробляє одну десяту таблиці. Паралелізм конвеєра розділяє операції запиту на етапи, які виконуються одночасно, передаючи результати від одного етапу до наступного без очікування завершення попереднього етапу.

Ефективність паралельної обробки залежить від можливості рівномірного розподілу роботи між виконавцями, мінімізації синхронізації та комунікації між паралельними потоками, балансування навантаження для уникнення простою окремих виконавців. Накладні витрати на координацію паралельного виконання можуть перевищити переваги паралелізму для невеликих запитів. Оптимальний ступінь паралелізму залежить від кількості доступних процесорних ядер, обсягу даних, характеру обчислень та можливості паралелізації конкретних операцій запиту.

Фізичне проектування баз даних становить комплексний процес, який вимагає глибокого розуміння характеристик апаратного та програмного забезпечення, специфіки навантаження системи, компромісів між різними аспектами продуктивності. Успішне фізичне проектування досягається через ітеративний процес проектування, реалізації, тестування та оптимізації, який враховує реальні патерни використання системи та еволюцію вимог з часом.

Вибір між різними стратегіями зберігання, індексації, кластеризації та розподілу даних повинен базуватися на емпіричному вимірюванні продуктивності та систематичному аналізі альтернатив. Інструменти моніторингу та профілювання допомагають ідентифікувати вузькі місця та направляють зусилля з оптимізації на найбільш критичні аспекти системи. Документування рішень з фізичного проектування та обґрунтувань за ними полегшує майбутню еволюцію та обслуговування системи.

Сучасні тенденції у фізичному проектуванні баз даних включають використання автоматизованих систем налаштування, які аналізують робоче навантаження та рекомендують оптимальні конфігурації індексів та інших фізичних структур. Інтеграція машинного навчання у компоненти СУБД дозволяє адаптивну оптимізацію запитів та динамічне управління ресурсами на основі спостережуваних патернів. Гетерогенні системи зберігання, які поєднують різні типи носіїв та організацій даних в єдиній архітектурі, надають нові можливості для оптимізації співвідношення ціни та продуктивності.

ЗМІСТОВИЙ МОДУЛЬ 3
СИСТЕМА УПРАВЛІННЯ БАЗАМИ ДАНИХ MYSQL

ТЕМА 9. ОСОБЛИВОСТІ ПІДКЛЮЧЕННЯ ДО БАЗИ ДАНИХ ПРИКЛАДНИХ ПРОГРАМ

1. Схема підключення до бази даних та драйвера
2. Складання схеми бази даних
3. Діаграми відношень об'єктів
4. Реалізація моделі
5. Аналіз функціональної моделі предметної області бази даних
6. Визначення функцій
7. Відображення функцій в модулі
8. Системні модулі
9. Розміщення логіки обробки
10. Загальні принципи розробки специфікацій модулів
11. Проектування процесу тестування модулів застосувань

12. Тестування
13. Планування життєвого циклу

1. СХЕМА ПІДКЛЮЧЕННЯ ДО БАЗИ ДАНИХ ТА ДРАЙВЕРА

Процес взаємодії прикладної програми з **базою даних** реалізується через складну ієрархічну архітектуру, де ключову роль відіграють спеціалізовані програмні компоненти. **Драйвер бази даних** являє собою програмний модуль, який забезпечує стандартизований інтерфейс між прикладною програмою та конкретною системою керування базами даних.

Архітектура підключення базується на декількох рівнях абстракції. На верхньому рівні розташовується прикладна програма, яка формує запити до бази даних. Ці запити передаються через **програмний інтерфейс додатків** (API), який забезпечує уніфікований спосіб взаємодії незалежно від типу бази даних. Серед найпоширеніших API слід відзначити **ODBC** (Open Database Connectivity), **JDBC** (Java Database Connectivity), **ADO.NET** для платформи .NET, а також **ORM-фреймворки** (Object-Relational Mapping), такі як Entity Framework, Hibernate або SQLAlchemy.

Драйвер виконує критичну функцію трансляції стандартизованих викликів API у специфічні команди, які розуміє конкретна СУБД. Це дозволяє розробникам писати код, який є переносимим між різними системами баз даних з мінімальними модифікаціями. Драйвер також відповідає за встановлення **мережевого з'єднання** з сервером бази даних, управління **транзакціями**, обробку помилок та оптимізацію передачі даних.

Процес підключення включає кілька етапів. Спочатку відбувається **ініціалізація драйвера**, під час якої завантажуються необхідні бібліотеки. Далі формується **рядок підключення** (connection string), який містить критичну інформацію: адресу сервера бази даних, номер порту, ім'я бази даних, облікові дані користувача та додаткові параметри. Після успішної **автентифікації** встановлюється активне з'єднання, через яке здійснюється обмін даними.

Важливим аспектом є **управління з'єднаннями**. Оскільки встановлення нового з'єднання є ресурсоємною операцією, використовуються **пули з'єднань** (connection pooling), які дозволяють повторно використовувати існуючі з'єднання, значно підвищуючи продуктивність системи.

2. СКЛАДАННЯ СХЕМИ БАЗИ ДАНИХ

Схема бази даних представляє собою формалізовану структуру, яка визначає організацію даних, їхні взаємозв'язки та обмеження цілісності. Процес

розробки схеми є фундаментальним етапом проектування інформаційної системи.

Розробка схеми починається з **концептуального моделювання**, де визначаються основні **сутності** (entities) предметної області та їхні **атрибути**. Сутність являє собою об'єкт реального світу, який має самостійне значення та про який необхідно зберігати інформацію. Наприклад, у системі управління університетом сутностями можуть бути студенти, викладачі, курси та аудиторії.

Кожна сутність характеризується набором **атрибутів** – властивостей, які описують її характеристики. Серед атрибутів виділяють **первинний ключ** (primary key) – унікальний ідентифікатор, який однозначно визначає кожен екземпляр сутності. Також визначаються **зовнішні ключі** (foreign keys), які забезпечують зв'язки між різними таблицями.

Нормалізація є критичним процесом оптимізації схеми бази даних. Вона передбачає послідовне застосування **нормальних форм** для усунення надмірності даних та аномалій при виконанні операцій вставки, оновлення та видалення. **Перша нормальна форма** (1NF) вимагає атомарності значень атрибутів. **Друга нормальна форма** (2NF) усуває часткову функціональну залежність неключових атрибутів від складеного первинного ключа. **Третя нормальна форма** (3NF) виключає транзитивні залежності між неключовими атрибутами.

У деяких випадках застосовується **денормалізація** – свідоме відхилення від принципів нормалізації з метою підвищення продуктивності читання даних. Це рішення приймається на основі аналізу конкретних вимог до швидкодії системи.

Індекси є спеціалізованими структурами даних, які прискорюють пошук інформації. Правильне проектування індексів значно впливає на продуктивність запитів, особливо при роботі з великими обсягами даних. Однак надмірна кількість індексів може уповільнити операції модифікації даних.

3. ДІАГРАМИ ВІДНОШЕНЬ ОБ'ЄКТІВ

Діаграма відношень сутностей (Entity-Relationship Diagram, ERD) є потужним інструментом візуального представлення структури бази даних. Вона дозволяє наочно продемонструвати сутності, їхні атрибути та взаємозв'язки між ними.

На діаграмі сутності зображуються у вигляді прямокутників, атрибути – овалів, а відношення – ромбів. **Типи зв'язків** між сутностями класифікуються за кардинальністю. **Зв'язок "один-до-одного"** (1:1) означає, що один екземпляр

першої сутності пов'язаний рівно з одним екземпляром другої сутності. **Зв'язок "один-до-багатьох"** (1:N) є найпоширенішим та вказує, що один екземпляр однієї сутності може бути пов'язаний з багатьма екземплярами іншої. **Зв'язок "багато-до-багатьох"** (M:N) реалізується через введення проміжної сутності – **асоціативної таблиці**.

Діаграми також відображають **атрибути зв'язків** – характеристики, які належать не окремим сутностям, а їхнім відношенням. Наприклад, у зв'язку між студентом та курсом атрибутом може бути оцінка або дата реєстрації.

Нотація Чена та **нотація "курячі лапки"** (crow's foot notation) є двома основними стандартами створення ERD. Друга нотація є більш інтуїтивною завдяки графічному зображенню кардинальності зв'язків.

При проектуванні складних систем використовуються **спеціалізація** та **узагальнення** – механізми, що дозволяють моделювати ієрархічні відношення між сутностями. Спеціалізація передбачає виділення підтипів сутності з додатковими специфічними атрибутами, тоді як узагальнення об'єднує спільні характеристики різних сутностей.

4. РЕАЛІЗАЦІЯ МОДЕЛІ

Перехід від концептуальної моделі до **фізичної реалізації** бази даних потребує перетворення абстрактних конструкцій у конкретні структури, які підтримуються обраною СУБД. Цей процес називається **фізичним проектуванням**.

Відображення сутностей здійснюється шляхом створення таблиць у базі даних. Кожна сутність концептуальної моделі трансформується в окрему таблицю, а атрибути стають стовпцями цієї таблиці. При цьому необхідно визначити **типи даних** для кожного стовпця: числові типи (INTEGER, DECIMAL), символьні (VARCHAR, TEXT), дати та часу (DATE, TIMESTAMP), булеві значення та інші спеціалізовані типи.

Обмеження цілісності впроваджуються через механізми СУБД. **Обмеження первинного ключа** (PRIMARY KEY constraint) гарантує унікальність та відсутність NULL-значень. **Обмеження зовнішнього ключа** (FOREIGN KEY constraint) забезпечує **референційну цілісність**, не дозволяючи посилатися на неіснуючі записи. **Обмеження унікальності** (UNIQUE constraint) запобігає дублюванню значень у певних стовпцях. **Перевірочні обмеження** (CHECK constraint) дозволяють визначити довільні умови, яким повинні відповідати дані.

Тригери (triggers) є спеціальними процедурами, які автоматично виконуються у відповідь на певні події в базі даних: вставку, оновлення або видалення записів. Вони використовуються для підтримки складної бізнес-логіки, аудиту змін, каскадного оновлення залежних даних.

Збережені процедури та **функції** дозволяють інкапсулювати складну логіку обробки даних безпосередньо в базі даних. Це зменшує мережевий трафік, підвищує безпеку через обмеження прямого доступу до таблиць та полегшує повторне використання коду.

Представлення (views) є віртуальними таблицями, які формуються на основі запитів до реальних таблиць. Вони спрощують доступ до даних, приховують складність структури бази даних та забезпечують додатковий рівень безпеки.

5. АНАЛІЗ ФУНКЦІОНАЛЬНОЇ МОДЕЛІ ПРЕДМЕТНОЇ ОБЛАСТІ БАЗИ ДАНИХ

Функціональна модель описує процеси обробки даних у системі, визначаючи операції, які можуть виконуватися над інформацією. Цей аналіз є критичним для розуміння вимог до прикладного програмного забезпечення.

Методологія **IDEF0** (Integration Definition for Function Modeling) широко використовується для створення функціональних моделей. Вона базується на **декомпозиції функцій** – послідовному розбитті складних процесів на простіші підпроцеси. Кожна функція на діаграмі зображується у вигляді прямокутника з чотирма типами зв'язків: **вхідні дані**, **вихідні дані**, **механізми виконання** (ресурси) та **керуючі впливи** (правила, політики).

Діаграми потоків даних (Data Flow Diagrams, DFD) є альтернативним інструментом моделювання. Вони фокусуються на русі інформації через систему, визначаючи **процеси** обробки, **зовнішні сутності** (джерела та приймачі даних), **потоки даних** та **сховища даних**.

Аналіз функціональної моделі дозволяє ідентифікувати **транзакції** – логічно пов'язані послідовності операцій, які повинні виконуватися атомарно. Розуміння транзакційних вимог є ключовим для забезпечення **властивостей ACID**: атомарності (Atomicity), узгодженості (Consistency), ізоляції (Isolation) та довговічності (Durability).

Сценарії використання (use cases) деталізують взаємодію користувачів з системою, описуючи послідовності дій для досягнення конкретних цілей. Вони служать основою для визначення функціональних вимог до програмного забезпечення.

6. ВИЗНАЧЕННЯ ФУНКЦІЙ

На основі функціональної моделі відбувається **специфікація програмних функцій** – детальний опис операцій, які повинна виконувати прикладна програма. Кожна функція характеризується **сигнатурою** (назва, параметри, тип результату), **передумовами** (умови, які повинні виконуватися перед викликом) та **постумовами** (гарантований стан після виконання).

CRUD-операції (Create, Read, Update, Delete) є базовими функціями для маніпулювання даними. Операція **створення** (Create) додає нові записи до бази даних. Операція **читання** (Read) отримує інформацію без її модифікації. Операція **оновлення** (Update) змінює існуючі дані. Операція **видалення** (Delete) вилучає записи з бази даних.

Крім базових операцій, визначаються **бізнес-функції**, які реалізують специфічну логіку предметної області. Наприклад, у банківській системі це можуть бути функції переказу коштів, нарахування відсотків, формування звітності.

Функції пошуку та фільтрації дозволяють користувачам ефективно знаходити потрібну інформацію серед великих обсягів даних. Вони включають **простий пошук** за одним критерієм, **складний пошук** з комбінацією умов, **повнотекстовий пошук** та **фасетну навігацію**.

Функції агрегації та аналітики обчислюють узагальнені показники: суми, середні значення, максимуми, мінімуми, кількості записів. Вони є основою для створення звітів та дашбордів.

Функції авторизації та аутентифікації забезпечують безпеку системи, контролюючи доступ користувачів до даних та операцій на основі їхніх ролей та прав.

7. ВІДОБРАЖЕННЯ ФУНКЦІЙ В МОДУЛІ

Модульна архітектура передбачає розбиття програмної системи на логічно пов'язані компоненти з чітко визначеними інтерфейсами. Кожна функція або група пов'язаних функцій інкапсулюється в окремий **модуль**.

Принцип єдиної відповідальності (Single Responsibility Principle) стверджує, що кожен модуль повинен мати одну причину для зміни. Це означає, що модуль повинен бути відповідальним за одну чітко визначену частину функціональності системи.

Шари архітектури визначають вертикальну організацію модулів. **Шар представлення** (presentation layer) відповідає за взаємодію з користувачем, відображення інформації та обробку введення. **Шар бізнес-логіки** (business logic layer) містить правила та процеси предметної області. **Шар доступу до даних** (data access layer) інкапсулює взаємодію з базою даних, приховуючи деталі SQL-запитів та специфіки СУБД.

Патерни проектування надають перевірені рішення для типових проблем. **Патерн Repository** абстрагує логіку доступу до даних, надаючи об'єктно-орієнтований інтерфейс для роботи з колекціями сутностей. **Патерн Unit of Work** координує зміни множини об'єктів, забезпечуючи їх атомарне збереження в рамках однієї транзакції. **Патерн Data Mapper** відокремлює об'єкти предметної області від деталей їх зберігання в базі даних.

Інверсія залежностей (Dependency Inversion Principle) та **впровадження залежностей** (Dependency Injection) є механізмами, які підвищують гнучкість та тестованість коду. Замість того щоб модулі безпосередньо створювали екземпляри залежних компонентів, вони отримують їх зовні через конструктор або методи.

8. СИСТЕМНІ МОДУЛІ

Системні модулі забезпечують інфраструктурну функціональність, необхідну для роботи прикладної системи. Вони не реалізують бізнес-логіку безпосередньо, але надають критичні сервіси іншим компонентам.

Модуль логування (logging module) реєструє події, що відбуваються в системі. Він фіксує інформаційні повідомлення, попередження та помилки, які використовуються для моніторингу, діагностики проблем та аудиту. Логи структуруються за **рівнями важливості**: DEBUG, INFO, WARNING, ERROR, CRITICAL. Сучасні системи логування підтримують **структуроване логування** у форматах JSON або XML, що полегшує автоматичний аналіз.

Модуль обробки помилок (error handling module) забезпечує централізовану обробку винятків. Він перехоплює помилки, форматує їх у зрозумілі повідомлення, визначає стратегію відновлення та при необхідності відкочує транзакції для збереження цілісності даних.

Модуль кешування (caching module) зберігає результати дорогих операцій у швидкій пам'яті для повторного використання. Це значно підвищує продуктивність системи, зменшуючи навантаження на базу даних. Використовуються різні **стратегії кешування**: кеш сторінок, кеш запитів, кеш об'єктів. **Алгоритми витіснення** (eviction algorithms), такі як LRU (Least

Recently Used) або LFU (Least Frequently Used), визначають, які дані видаляти при заповненні кешу.

Модуль валідації (validation module) перевіряє коректність вхідних даних перед їх обробкою. Він реалізує правила валідації на рівні типів даних, форматів, діапазонів значень, а також бізнес-правила предметної області.

Модуль конфігурації (configuration module) керує налаштуваннями системи, дозволяючи змінювати параметри без перекомпіляції коду. Конфігурація може зберігатися у файлах (XML, JSON, YAML), змінних середовища або спеціалізованих сервісах.

Модуль безпеки (security module) реалізує механізми аутентифікації, авторизації, шифрування даних та захисту від типових атак (SQL-ін'єкцій, XSS, CSRF).

9. РОЗМІЩЕННЯ ЛОГІКИ ОБРОБКИ

Вибір місця розташування **логіки обробки даних** є стратегічним архітектурним рішенням, яке впливає на продуктивність, масштабованість та підтримуваність системи.

Обробка на стороні клієнта виконується у браузері користувача або настільному додатку. Вона забезпечує швидкий відгук інтерфейсу, зменшує навантаження на сервер та дозволяє працювати в режимі офлайн. Однак клієнтська логіка не може повністю довіряти, оскільки зловмисники можуть її модифікувати.

Обробка на рівні прикладного сервера є найпоширенішим підходом. Бізнес-логіка виконується в контрольованому середовищі, де забезпечується безпека та цілісність обробки. Прикладний сервер може масштабуватися горизонтально шляхом додавання нових екземплярів.

Обробка на рівні бази даних через збережені процедури, функції та тригери має свої переваги: мінімізація мережевого трафіку, використання оптимізацій СУБД, забезпечення атомарності складних операцій. Проте надмірне використання логіки на рівні БД ускладнює міграцію між різними СУБД та тестування.

Гібридний підхід комбінує різні рівні обробки. Проста валідація може виконуватися на клієнті для швидкого відгуку, критична бізнес-логіка – на прикладному сервері, а операції, що вимагають транзакційної цілісності множини таблиць, – у збережених процедурах.

Мікросервісна архітектура розподіляє логіку між множиною незалежних сервісів, кожен з яких має власну базу даних. Це забезпечує високу

масштабованість та гнучкість, але потребує складної координації та обробки розподілених транзакцій.

10. ЗАГАЛЬНІ ПРИНЦИПИ РОЗРОБКИ СПЕЦИФІКАЦІЙ МОДУЛІВ

Специфікація модуля є формальним описом його функціональності, інтерфейсів та поведінки. Якісна специфікація є основою для імплементації, тестування та документування.

Інтерфейс модуля визначає його публічні методи – операції, доступні зовнішнім клієнтам. Кожен метод специфікується через **сигнатуру**: ім'я, список параметрів з їх типами та тип значення, що повертається. Важливо дотримуватися принципу **мінімального інтерфейсу** – експонувати тільки ті методи, які дійсно необхідні зовнішнім споживачам.

Передумови (preconditions) описують умови, які повинні виконуватися перед викликом методу. Це можуть бути обмеження на значення параметрів, вимоги до стану об'єкта або системи. Відповідальність за виконання передумов лежить на клієнті модуля.

Постумови (postconditions) гарантують стан системи після успішного виконання методу. Вони описують зміни в об'єктах, повернені значення та побічні ефекти. Відповідальність за забезпечення постумов лежить на самому модулі.

Інваріанти – це властивості, які залишаються істинними протягом усього життєвого циклу об'єкта. Вони визначають внутрішню узгодженість стану модуля.

Документація виключень описує всі можливі виняткові ситуації, які можуть виникнути під час виконання методу, та умови їх появи. Це дозволяє клієнтам коректно обробляти помилки.

Специфікація складності вказує на **часову** та **просторову складність** операцій, що допомагає у прийнятті рішень про використання модуля в критичних за продуктивністю місцях.

Контракти (contracts) формалізують угоду між модулем та його клієнтами, чітко визначаючи права та обов'язки кожної сторони. Це концепція **програмування за контрактом** (Design by Contract).

11. ПРОЕКТУВАННЯ ПРОЦЕСУ ТЕСТУВАННЯ МОДУЛІВ ЗАСТОСУВАНЬ

Тестування є систематичним процесом верифікації та валідації програмного забезпечення. **Верифікація** відповідає на питання "чи правильно

ми створюємо продукт?" (відповідність специфікаціям), тоді як **валідація** – "чи створюємо ми правильний продукт?" (відповідність потребам користувачів).

Пірамід тестування визначає оптимальне співвідношення різних типів тестів. В її основі знаходяться **модульні тести** (unit tests) – найбільш численні та швидкі тести, які перевіряють окремі компоненти в ізоляції. Вище розташовуються **інтеграційні тести** (integration tests), що верифікують взаємодію між модулями. На вершині піраміди – **системні** та **приймальні тести** (acceptance tests), які перевіряють систему в цілому.

Модульне тестування фокусується на найменших програмних одиницях – функціях, методах, класах. Для ізоляції модуля від залежностей використовуються **тестові дублери**: **заглушки** (stubs) повертають заздалегідь визначені значення, **макети** (mocks) перевіряють правильність взаємодії, **фальшивки** (fakes) є спрощеними робочими реалізаціями.

Методологія TDD (Test-Driven Development) передбачає написання тестів до реалізації функціональності. Цикл розробки складається з трьох етапів: "червоний" (написання тесту, який не проходить), "зелений" (мінімальна реалізація для проходження тесту), "рефакторинг" (покращення коду без зміни поведінки).

Покриття коду (code coverage) вимірює відсоток коду, що виконується під час тестування. Розрізняють покриття **рядків**, **гілок** (розгалужень), **шляхів виконання** та **умов**. Високе покриття не гарантує відсутність помилок, але низьке покриття певно вказує на недостатність тестування.

Тестування баз даних включає перевірку коректності SQL-запитів, обмежень цілісності, тригерів та збережених процедур. Використовуються **тестові бази даних** з відомими наборами даних, які скидаються до початкового стану перед кожним тестом.

12. ТЕСТУВАННЯ

Комплексне **тестування програмного забезпечення** включає множину типів та рівнів перевірок, кожен з яких має специфічні цілі та методології.

Функціональне тестування верифікує, що система виконує заявлені функції відповідно до вимог. Воно включає **тестування еквівалентних класів**, де простір вхідних значень розбивається на класи, які очікувано обробляються однаково. **Аналіз граничних значень** фокусується на перевірці крайніх випадків: мінімальних, максимальних та граничних значень параметрів.

Нефункціональне тестування оцінює якісні характеристики системи. **Тестування продуктивності** вимірює швидкодію, пропускну здатність, час

відгуку під різним навантаженням. **Навантажувальне тестування** (load testing) перевіряє поведінку системи при нормальному та піковому навантаженні. **Стрес-тестування** визначає точку відмови системи при екстремальних умовах.

Тестування безпеки шукає вразливості: можливості SQL-ін'єкцій, міжсайтового скриптингу (XSS), підробки міжсайтових запитів (CSRF), вито ку конфіденційних даних. Використовуються інструменти автоматичного сканування та **пентестинг** (penetration testing) – імітація атак зловмисників.

Тестування сумісності перевіряє коректну роботу з різними версіями операційних систем, браузерів, СУБД та іншого програмного оточення. **Тестування зручності використання** (usability testing) оцінює, наскільки інтуїтивним та зручним є інтерфейс для кінцевих користувачів.

Регресійне тестування забезпечує, що нові зміни не порушили раніше працюючу функціональність. Це особливо важливо при **рефакторингу** коду та додаванні нових можливостей.

Автоматизація тестування використовує спеціалізовані фреймворки та інструменти для автоматичного виконання тестів. **Continuous Integration** (CI) системи автоматично запускають тести при кожному комміті коду, забезпечуючи швидке виявлення дефектів.

13. ПЛАНУВАННЯ ЖИТТЄВОГО ЦИКЛУ

Життєвий цикл програмного забезпечення (Software Development Life Cycle, SDLC) охоплює всі етапи від початкової концепції до виведення системи з експлуатації. Планування життєвого циклу є критичним для успішного завершення проекту.

Каскадна модель (Waterfall) передбачає послідовне виконання фаз: збір вимог, проектування, імплементація, тестування, впровадження, супровід. Кожна фаза завершується перед початком наступної. Ця модель підходить для проектів зі стабільними, добре визначеними вимогами.

Ітеративні моделі розбивають розробку на цикли, кожен з яких проходить усі фази та виробляє робочу версію системи. **Спіральна модель** комбінує ітеративність з аналізом ризиків, приділяючи особливу увагу виявленню та мітигації загроз на ранніх етапах.

Agile-методології фокусуються на гнучкості, швидкій адаптації до змін та безперервній доставці цінності клієнту. **Scrum** організує роботу в **спринти** – короткі ітерації тривалістю 1-4 тижні. **Kanban** використовує візуалізацію робочого процесу та обмеження кількості одночасних задач для оптимізації потоку.

Фаза аналізу вимог включає збір, документування та узгодження функціональних та нефункціональних вимог зі зацікавленими сторонами. Створюються **документи специфікації вимог** (Software Requirements Specification, SRS).

Фаза проектування розробляє архітектуру системи, проектує бази даних, визначає інтерфейси компонентів. Створюються **діаграми UML**: діаграми класів, послідовностей, діяльності, розгортання.

Фаза імплементації передбачає написання коду згідно з проектними рішеннями, дотримуючись **стандартів кодування** та використовуючи **системи контролю версій** (Git, SVN).

Фаза тестування виконує всі типи перевірок, документує дефекти в **системах відстеження помилок** (issue tracking systems) та верифікує їх виправлення.

Фаза впровадження включає встановлення системи в продуктивному середовищі, міграцію даних, навчання користувачів та поступовий перехід від старої системи.

Фаза супроводу забезпечує виправлення дефектів, додавання нових можливостей, оптимізацію продуктивності та адаптацію до змін зовнішнього середовища. Розрізняють **коригувальне** (виправлення помилок), **адаптивне** (пристосування до нових умов), **перфективне** (покращення характеристик) та **превентивне** (запобігання майбутнім проблемам) супроводження.

Управління конфігурацією відстежує версії артефактів, контролює зміни та забезпечує можливість відтворення будь-якої версії системи. **Управління релізами** координує процес доставки нових версій до користувачів.

Сьогодні ми розглянули комплексний процес підключення прикладних програм до баз даних, що включає архітектурні рішення, проектування структури даних, реалізацію функціональності та забезпечення якості через тестування. Розуміння цих аспектів є фундаментальним для створення надійних, ефективних та масштабованих інформаційних систем. Успішна реалізація проекту бази даних потребує інтеграції знань з різних областей: теорії баз даних, програмної інженерії, управління проектами. Систематичний підхід до кожного етапу життєвого циклу, використання перевірених практик та постійне вдосконалення процесів забезпечують створення якісних програмних продуктів, які відповідають потребам бізнесу та користувачів.

ТЕМА 10. СИСТЕМА УПРАВЛІННЯ БАЗАМИ ДАНИХ MYSQL. ВСТАНОВЛЕННЯ СЕРВЕРА. ПІДКЛЮЧЕННЯ ДО MYSQL

1. Основні команди СУБД MySQL.
2. Функції, типи даних, робота з таблицями.
3. Створення ключів та індексів.
4. Зовнішні ключі. Зв’язування таблиць.
5. Захист даних в MySQL. Адміністрування. Привілеї.
6. Організація транзакцій.
7. Проектування баз даних за допомогою інструментальних засобів програми HeidiSQL

Вступ до MySQL та основні поняття

Система управління базами даних MySQL є однією з найпопулярніших реляційних СУБД у світі, яка знайшла широке застосування як у невеликих веб-проектах, так і у великих корпоративних системах. MySQL розроблена шведською компанією MySQL AB у 1995 році, а нині належить корпорації Oracle. Назва MySQL походить від імені дочки одного з розробників - My, та абревіатури SQL (Structured Query Language).

Популярність MySQL обумовлена кількома ключовими факторами. По-перше, це відкрите програмне забезпечення, яке розповсюджується під ліцензією GNU GPL, що робить його безкоштовним для більшості застосувань. По-друге, MySQL демонструє високу продуктивність та надійність, що підтверджується її використанням такими гігантами як Facebook, Twitter, YouTube та Wikipedia. По-третє, MySQL підтримує багатоплатформеність, працюючи на операційних системах Windows, Linux, macOS та багатьох інших.

Архітектура MySQL побудована за клієнт-серверною моделлю. Серверна частина відповідає за зберігання даних, обробку запитів та управління доступом, тоді як клієнтські програми встановлюють з'єднання з сервером для виконання операцій над базами даних. Така архітектура дозволяє одночасну роботу багатьох користувачів з однією базою даних, забезпечуючи при цьому ізоляцію та безпеку даних.

Основною перевагою MySQL є її швидкість виконання операцій читання, що робить її особливо ефективною для веб-додатків, де переважають запити вибірки даних. MySQL підтримує різні механізми зберігання даних, найпопулярнішими з яких є InnoDB та MyISAM. InnoDB забезпечує підтримку транзакцій, зовнішніх ключів та блокування на рівні рядків, що робить його стандартним вибором для більшості сучасних додатків. MyISAM, хоча і не

підтримує транзакції, пропонує вищу швидкість для операцій читання у простих сценаріях використання.

MySQL реалізує стандарт SQL, проте має деякі власні розширення та особливості. Система підтримує складні запити з об'єднаннями таблиць, підзапитами, групуванням та агрегатними функціями. Вбудовані функції MySQL охоплюють роботу з рядками, числами, датами, умовні вирази та багато іншого, надаючи розробникам потужний інструментарій для маніпуляції даними безпосередньо на рівні бази даних.

Встановлення сервера MySQL та підключення

Процес встановлення MySQL сервера залежить від операційної системи, проте загальні принципи залишаються схожими. Для операційної системи Windows найзручнішим способом є використання MySQL Installer - спеціального інсталяційного пакету, який включає всі необхідні компоненти. Після завантаження інсталятора з офіційного сайту mysql.com, користувач запускає його та обирає тип встановлення. Варіант Developer Default включає MySQL Server, MySQL Workbench, MySQL Shell та інші інструменти розробника, тоді як Server only встановлює лише серверну частину без додаткових утиліт.

Під час встановлення система пропонує налаштувати основні параметри сервера. Найважливішим є вибір типу конфігурації, де Development Machine оптимізує використання ресурсів для локальної розробки, Server Machine налаштовує сервер для використання як один з багатьох сервісів, а Dedicated Machine виділяє максимум ресурсів для MySQL. Також необхідно встановити порт підключення, за замовчуванням використовується порт 3306, та обрати метод аутентифікації користувачів.

Особливу увагу слід приділити створенню пароля для root-користувача, який має повний доступ до всіх баз даних та налаштувань системи. Пароль повинен бути достатньо складним, містити великі та малі літери, цифри та спеціальні символи. MySQL також пропонує створити додаткові облікові записи користувачів безпосередньо під час встановлення, що є хорошою практикою для розділення адміністративного доступу та робочих облікових записів.

В операційних системах сімейства Linux встановлення MySQL зазвичай виконується через менеджер пакетів. Для дистрибутивів на базі Debian та Ubuntu команда sudo apt install mysql-server завантажує та встановлює MySQL з репозиторіїв. Після встановлення рекомендується запустити скрипт mysql_secure_installation, який виконує базове налаштування безпеки, зокрема встановлення пароля root, видалення анонімних користувачів, заборону віддаленого входу для root та видалення тестової бази даних.

Підключення до MySQL сервера може здійснюватися різними способами залежно від потреб користувача. Найпростішим методом є використання консольного клієнта mysql, який встановлюється разом з сервером. Команда mysql з параметром мінус u вказує ім'я користувача, а параметр мінус p сигналізує про необхідність введення пароля. Після успішної аутентифікації користувач отримує командний рядок MySQL, де можна виконувати SQL-запити.

Для підключення до віддаленого сервера MySQL додається параметр мінус h з вказанням хосту або IP-адреси сервера. При роботі з віддаленими серверами важливо переконатися, що фаєрвол дозволяє підключення до порту MySQL, та що користувач має дозвіл на віддалений доступ. За замовчуванням MySQL дозволяє підключення тільки з localhost з міркувань безпеки.

Перевірка успішності встановлення та підключення виконується за допомогою простих SQL-команд. Команда SELECT VERSION показує версію встановленого MySQL, що корисно для діагностики та перевірки сумісності. Команда STATUS виводить детальну інформацію про поточне з'єднання, включаючи версію сервера, час роботи, кількість активних з'єднань та інші параметри. Команда SHOW DATABASES відображає список всіх баз даних, до яких користувач має доступ.

1. ОСНОВНІ КОМАНДИ СУБД MYSQL.

Робота з базами даних починається зі створення нової бази або підключення до існуючої. Команда CREATE DATABASE створює нову базу даних з вказаною назвою. Рекомендується використовувати варіант з перевіркою існування IF NOT EXISTS, який запобігає помилці, якщо база з такою назвою вже існує. При створенні бази даних важливо правильно налаштувати кодування символів, оскільки це впливає на можливість зберігання текстів різними мовами. Кодування utf8mb4 з collation utf8mb4_unicode_ci забезпечує повну підтримку Unicode, включаючи емодзі та спеціальні символи.

Перегляд існуючих баз даних здійснюється командою SHOW DATABASES, яка виводить список всіх баз, до яких поточний користувач має принаймні якийсь рівень доступу. Для початку роботи з конкретною базою даних використовується команда USE з назвою бази. Після виконання цієї команди всі наступні операції будуть виконуватися в контексті обраної бази даних, що дозволяє не вказувати назву бази перед кожною таблицею в запитах.

Видалення бази даних виконується командою DROP DATABASE, яка безповоротно знищує базу та всі її об'єкти, включаючи таблиці, індекси,

процедури та дані. Оскільки ця операція є критичною, рекомендується використовувати варіант з перевіркою IF EXISTS, який не генерує помилку, якщо база вже не існує. Перед видаленням продакшн бази даних завжди необхідно створювати резервну копію.

SQL-команди в MySQL поділяються на кілька категорій відповідно до їх призначення. Мова визначення даних DDL включає команди для створення, зміни та видалення структур бази даних. Команда CREATE застосовується для створення баз даних, таблиць, індексів, представлень та інших об'єктів. Команда ALTER дозволяє змінювати існуючі структури, додаючи або видаляючи стовпці, змінюючи типи даних, додаючи обмеження. Команда DROP видаляє об'єкти бази даних, а TRUNCATE швидко очищає таблицю від всіх даних без можливості відновлення.

Мова маніпулювання даними DML призначена для роботи безпосередньо з даними в таблицях. Команда SELECT виконує вибірку даних за заданими критеріями, підтримуючи складні умови, об'єднання таблиць, групування та сортування результатів. Команда INSERT додає нові записи в таблицю, дозволяючи вставити як один рядок, так і множину рядків за одну операцію. Команда UPDATE змінює значення полів в існуючих записах відповідно до заданих умов. Команда DELETE видаляє записи з таблиці, причому без вказівки умови WHERE будуть видалені всі записи.

Мова управління даними DCL забезпечує контроль доступу до об'єктів бази даних. Команда GRANT надає привілеї користувачам на виконання певних операцій з базами даних та таблицями. Команда REVOKE відкликає раніше надані привілеї. Ці команди критично важливі для забезпечення безпеки системи, дозволяючи реалізувати принцип мінімальних привілеїв, коли кожен користувач має лише ті права, які необхідні для виконання його робочих завдань.

Мова управління транзакціями TCL контролює виконання транзакцій в базі даних. Команда START TRANSACTION або BEGIN розпочинає нову транзакцію, після чого всі зміни даних виконуються в межах цієї транзакції. Команда COMMIT фіксує всі зміни, зроблені в межах транзакції, роблячи їх постійними та видимими для інших користувачів. Команда ROLLBACK скасовує всі зміни транзакції, повертаючи дані до стану перед початком транзакції. Команда SAVEPOINT дозволяє створити точку збереження всередині транзакції, до якої можна виконати частковий відкат.

2. ФУНКЦІЇ, ТИПИ ДАНИХ, РОБОТА З ТАБЛИЦЯМИ.

Система типів даних MySQL розроблена для ефективного зберігання різних видів інформації при оптимальному використанні дискового простору та пам'яті. Вибір правильного типу даних для кожного стовпця є важливим аспектом проектування бази даних, який впливає на продуктивність, цілісність даних та витрати ресурсів.

Числові типи даних представлені цілими та дробовими числами різної розрядності. Тип TINYINT займає один байт і зберігає цілі числа від мінус ста двадцяти восьми до ста двадцяти семи, або від нуля до двохсот п'ятдесяти п'яти для беззнакового варіанту UNSIGNED. Тип SMALLINT використовує два байти і підходить для більшого діапазону значень. MEDIUMINT займає три байти, INT або INTEGER використовує чотири байти і є найбільш уживаним типом для ідентифікаторів та лічильників. BIGINT резервує вісім байтів і здатний зберігати дуже великі цілі числа.

Дробові числа можуть зберігатися у двох форматах. Тип DECIMAL забезпечує точне зберігання десяткових дробів, що критично важливо для фінансових розрахунків де помилки округлення неприпустимі. При оголошенні DECIMAL вказуються два параметри - загальна кількість цифр та кількість цифр після десяткової крапки. Типи FLOAT та DOUBLE зберігають числа з плаваючою крапкою, займаючи відповідно чотири та вісім байтів, проте можуть мати невеликі похибки округлення через особливості двійкового представлення дробових чисел.

Рядкові типи даних призначені для зберігання текстової інформації. Тип CHAR зберігає рядки фіксованої довжини, доповнюючи коротші рядки пробілами до вказаної довжини. Це забезпечує передбачуваність розміру записів, проте може призводити до марнотратства простору для коротких рядків. Тип VARCHAR зберігає рядки змінної довжини, використовуючи тільки необхідну кількість байтів плюс один або два байти для зберігання довжини. VARCHAR є оптимальним вибором для більшості текстових полів змінної довжини.

Типи TEXT призначені для зберігання великих текстових даних. TINYTEXT зберігає до двохсот п'ятдесяти п'яти символів, TEXT вміщує до шістдесяти п'яти тисяч символів, MEDIUMTEXT може містити тексти розміром до шістнадцяти мегабайтів, а LONGTEXT підтримує тексти розміром до чотирьох гігабайтів. Варто врахувати, що TEXT-поля зберігаються окремо від основного запису таблиці, що може вплинути на продуктивність запитів.

Типи даних для роботи з датою та часом дозволяють зберігати темпоральну інформацію в різних форматах. Тип DATE зберігає календарну дату у форматі рік-місяць-день без інформації про час доби. Тип TIME зберігає час у

форматі година-хвилина-секунда без прив'язки до конкретної дати. Тип DATETIME комбінує дату та час, зберігаючи повну мітку моменту часу. Тип TIMESTAMP також зберігає дату та час, проте має особливість автоматичного оновлення при зміні запису, що корисно для відстеження часу модифікації даних. Тип YEAR зберігає лише рік у чотиризначному форматі.

Бінарні типи BLOB схожі на текстові типи TEXT, проте призначені для зберігання двійкових даних, таких як зображення, аудіо, відео або інші файли. Вони мають аналогічну градацію розмірів від TINYBLOB до LONGBLOB. Проте зберігання великих файлів безпосередньо в базі даних не завжди є оптимальним рішенням - часто краще зберігати файли в файловій системі, а в базі даних тримати лише шляхи до них.

Тип ENUM дозволяє визначити список допустимих текстових значень для поля, при цьому фактично зберігаючи числовий індекс обраного значення. Це економить простір та забезпечує цілісність даних, обмежуючи можливі значення заздалегідь визначеним набором. Тип SET схожий на ENUM, проте дозволяє вибрати кілька значень одночасно зі списку.

MySQL надає багатий набір вбудованих функцій для обробки даних різних типів. Рядкові функції дозволяють маніпулювати текстом безпосередньо в запитах. Функція CONCAT об'єднує декілька рядків в один, що корисно для формування повних імен з окремих полів імені та прізвища. Функція LENGTH повертає довжину рядка в байтах, тоді як CHAR_LENGTH рахує кількість символів, що важливо для багатобайтових кодувань. Функції UPPER та LOWER перетворюють регістр символів, що часто потрібно для регістронезалежного порівняння.

Функція SUBSTRING витягує підрядок із заданої позиції та довжини, що дозволяє працювати з частинами рядків. Функція REPLACE замінює всі входження одного підрядка на інший у вихідному рядку. Функції TRIM, LTRIM та RTRIM видаляють зайві пробіли відповідно з обох кінців, початку або кінця рядка. Функція CONCAT_WS об'єднує рядки з заданим роздільником, автоматично пропускаючи NULL значення.

Числові функції забезпечують математичні обчислення та обробку числових даних. Функція ABS повертає абсолютне значення числа. Функція ROUND округлює число до вказаної кількості десяткових знаків, CEILING округлює вгору до найближчого цілого, а FLOOR округлює вниз. Функція POW або POWER піднімає число до заданого степеня, SQRT обчислює квадратний корінь. Функція MOD повертає остачу від ділення одного числа на інше.

Агрегатні функції виконують обчислення над множиною рядків та повертають одне результуюче значення. Функція COUNT підраховує кількість записів або ненульових значень у стовпці. Функція SUM обчислює суму числових значень, AVG розраховує середнє арифметичне. Функції MAX та MIN знаходять відповідно максимальне та мінімальне значення в наборі даних. Ці функції зазвичай використовуються разом з групуванням даних за допомогою GROUP BY.

Функції для роботи з датою та часом надають потужні можливості для темпоральних обчислень. Функція NOW повертає поточну дату та час сервера, CURDATE повертає поточну дату, а CURTIME поточний час. Функції YEAR, MONTH, DAY, HOUR, MINUTE та SECOND витягують відповідні компоненти з значення дати-часу. Функція DATEDIFF обчислює різницю між двома датами у днях.

Функції DATE_ADD та DATE_SUB дозволяють додавати або віднімати часові інтервали до дати. Інтервали можуть бути виражені в різних одиницях - днях, місяцях, роках, годинах тощо. Функція DATE_FORMAT форматує дату відповідно до заданого шаблону, що дозволяє отримати текстове представлення дати у будь-якому потрібному форматі.

Умовні функції дозволяють реалізувати логіку прийняття рішень безпосередньо в SQL-запитах. Функція IF приймає умову та повертає одне значення якщо умова істинна, інше якщо хибна. Конструкція CASE надає більш гнучкі можливості для реалізації складної умовної логіки з множиною варіантів. Функція IFNULL повертає альтернативне значення якщо перший аргумент є NULL. Функція COALESCE повертає перше ненульове значення зі списку аргументів.

Робота з таблицями

Створення таблиць є фундаментальною операцією при побудові структури бази даних. Команда CREATE TABLE визначає назву нової таблиці та специфікацію всіх її стовпців разом з їх характеристиками. Кожен стовпець описується своєю назвою, типом даних та набором необов'язкових атрибутів та обмежень. Грамотне проектування таблиць визначає ефективність всієї системи та легкість подальшої роботи з даними.

При визначенні стовпців важливо правильно підібрати тип даних та встановити відповідні обмеження. Атрибут NOT NULL вказує що поле є обов'язковим і не може містити невизначене значення, що забезпечує цілісність критично важливих даних. Атрибут UNIQUE гарантує унікальність значень у стовпці, запобігаючи дублюванню, наприклад електронних адрес або номерів

документів. Атрибут DEFAULT встановлює значення за замовчуванням, яке буде автоматично присвоєно при вставці запису без явного вказівки значення цього поля.

Атрибут AUTO_INCREMENT автоматично генерує послідовне числове значення для кожного нового запису, що особливо корисно для створення унікальних ідентифікаторів. Зазвичай AUTO_INCREMENT використовується для первинного ключа таблиці. При вставці запису можна не вказувати значення для такого поля або вказати NULL, і система сама присвоїть наступне доступне число.

Обмеження CHECK дозволяє визначити умову яку повинні задовольняти значення в стовпці або комбінація значень кількох стовпців. Це забезпечує додатковий рівень валідації даних безпосередньо на рівні бази даних, наприклад можна обмежити діапазон числових значень або встановити залежності між полями.

Вставка даних у таблицю виконується командою INSERT INTO з вказанням назви таблиці, списку стовпців та значень для вставки. Якщо надаються значення для всіх стовпців у порядку їх визначення, список стовпців можна опустити. Проте явне вказування стовпців робить запит більш читабельним та захищеним від помилок при зміні структури таблиці. Можна вставити кілька рядків за одну команду, перерахувавши кортежі значень через кому, що є більш ефективним ніж виконання окремої команди для кожного рядка.

Вибірка даних здійснюється командою SELECT яка є найбільш використовуваною командою в SQL. У найпростішому вигляді SELECT зірочка FROM вибирає всі стовпці з таблиці. Для отримання лише певних стовпців замість зірочки перераховуються їх назви. Умови відбору задаються в секції WHERE з використанням операторів порівняння, логічних операторів AND OR NOT, оператора LIKE для шаблонного пошуку, оператора IN для перевірки належності до множини значень, операторів BETWEEN для діапазонів та IS NULL для перевірки на невизначеність.

Сортування результатів вибірки виконується секцією ORDER BY зі вказанням стовпця або виразу для сортування та напрямку ASC для зростання або DESC для спадання. Можна вказати кілька критеріїв сортування, і вони будуть застосовані послідовно. Обмеження кількості повернутих рядків задається секцією LIMIT з числом рядків або парою чисел для пропуску певної кількості рядків та повернення наступної порції, що корисно для реалізації посторінкової навігації.

Групування даних виконується секцією GROUP BY яка об'єднує рядки з однаковими значеннями вказаних стовпців у групи, для яких можна обчислити агрегатні функції. Умови фільтрації груп задаються секцією HAVING яка на відміну від WHERE застосовується після групування та може використовувати агрегатні функції в умовах. Це дозволяє наприклад знайти категорії товарів де середня ціна перевищує певне значення або відділи де працює більше певної кількості співробітників.

Оновлення існуючих даних виконується командою UPDATE з вказанням таблиці, стовпців що змінюються та їх нових значень, а також умови WHERE що визначає які саме рядки будуть оновлені. Без умови WHERE будуть оновлені всі рядки таблиці, тому необхідно бути дуже уважним при формуванні запитів на оновлення. В секції SET можна використовувати вирази та функції для обчислення нових значень, включаючи посилання на поточні значення стовпців.

Видалення даних з таблиці виконується командою DELETE FROM з умовою WHERE що вказує які рядки слід видалити. Як і з UPDATE без умови WHERE будуть видалені всі рядки, що зазвичай не є бажаним результатом. Команда TRUNCATE TABLE видаляє всі дані з таблиці значно швидше ніж DELETE оскільки не генерує записів про видалення для кожного рядка, проте не дозволяє використати умови відбору та не може бути відкочена в транзакції для деяких механізмів зберігання.

Зміна структури існуючої таблиці виконується командою ALTER TABLE яка підтримує широкий спектр операцій. Команда ALTER TABLE ADD COLUMN додає новий стовпець з вказаним типом та атрибутами, можна також вказати позицію вставки стовпця відносно інших. Команда DROP COLUMN видаляє стовпець разом з усіма його даними безповоротно. Команда MODIFY COLUMN змінює визначення стовпця включаючи тип даних та атрибути зберігаючи його назву. Команда CHANGE COLUMN дозволяє одночасно перейменувати стовпець та змінити його визначення.

Команда ALTER TABLE також дозволяє додавати та видаляти обмеження цілісності, індекси, зовнішні ключі та інші елементи структури таблиці. Можна перейменувати таблицю командою RENAME TO або змінити механізм зберігання командою ENGINE дорівнює. При виконанні ALTER TABLE на великих таблицях слід враховувати що деякі операції можуть вимагати перебудови всієї таблиці та займати тривалий час, блокуючи доступ до даних.

Видалення таблиці виконується командою DROP TABLE яка безповоротно знищує таблицю разом з усіма даними та залежними об'єктами такими як індекси. Рекомендується використовувати варіант з перевіркою IF

EXISTS щоб уникнути помилки якщо таблиця вже не існує. Перед видаленням важливих таблиць необхідно переконатися в наявності актуальної резервної копії.

3. СТВОРЕННЯ КЛЮЧІВ ТА ІНДЕКСІВ.

Первинний ключ є фундаментальним поняттям реляційних баз даних, що забезпечує унікальну ідентифікацію кожного запису в таблиці. Первинний ключ може складатися з одного стовпця або комбінації кількох стовпців, головною вимогою є унікальність значень та відсутність NULL. MySQL автоматично створює унікальний індекс для первинного ключа, що забезпечує швидкий доступ до записів за ключем.

При проектуванні таблиць зазвичай створюється штучний первинний ключ на основі AUTO_INCREMENT стовпця типу INT або BIGINT. Такий підхід має кілька переваг - простота, незмінність ключа незалежно від змін бізнес-даних, компактність та ефективність індексування. Альтернативно можна використовувати природний ключ з бізнес-даних, наприклад номер паспорта або код товару, проте це має свої недоліки через можливість зміни таких даних.

Складений первинний ключ утворюється з комбінації кількох стовпців і використовується коли жоден окремий стовпець не гарантує унікальності або для реалізації зв'язку багато-до-багатьох через проміжну таблицю. Наприклад у таблиці реєстрації студентів на курси первинним ключем може бути пара ідентифікатор студента та ідентифікатор курсу.

Індекси є структурами даних що прискорюють пошук та вибірку даних з таблиць за рахунок створення впорядкованих копій певних стовпців з посиланнями на відповідні рядки. Індекси критично важливі для продуктивності запитів оскільки без них база даних змушена виконувати повне сканування таблиці для знаходження потрібних записів. Проте індекси мають свою ціну - вони займають додатковий дисковий простір та сповільнюють операції вставки, оновлення та видалення оскільки потребують супровідної підтримки.

Звичайний індекс створюється командою CREATE INDEX і може включати один або кілька стовпців. При виборі стовпців для індексування слід орієнтуватися на ті що часто використовуються в умовах WHERE, JOIN, ORDER BY або GROUP BY. Індекси особливо ефективні на стовпцях з високою селективністю де значення добре розподілені і більшість значень унікальні або майже унікальні.

Унікальний індекс створюється з атрибутом UNIQUE і на додачу до прискорення пошуку забезпечує унікальність значень у проіндексованих

стовпцях. Це дозволяє реалізувати бізнес-правила на рівні бази даних, наприклад гарантувати що електронна адреса або номер телефону не повторюються. Спроба вставити або оновити запис з дублюючим значенням призведе до помилки.

Повнотекстові індекси типу FULLTEXT призначені для пошуку слів та фраз у текстових полях. Вони створюють спеціальні структури для швидкого пошуку за природномовними запитами, підтримуючи булеві оператори та ранжування результатів за релевантністю. Повнотекстовий пошук значно ефективніший за LIKE для великих текстів проте вимагає більше ресурсів для підтримки індексу.

Складені або композитні індекси включають кілька стовпців і найбільш ефективні коли запити фільтрують або сортують за комбінацією цих стовпців. Важливим є порядок стовпців в індексі - більш селективні стовпці зазвичай розміщують першими. Індекс може використовуватися для запитів які використовують префікс стовпців індексу починаючи з першого.

Для перегляду індексів таблиці використовується команда SHOW INDEX яка виводить детальну інформацію про всі індекси включаючи їх назви, стовпці, унікальність та інші характеристики. Видалення індексу виконується командою DROP INDEX або ALTER TABLE DROP INDEX коли індекс більше не потрібен або заважає продуктивності операцій модифікації даних.

Для аналізу використання індексів та оптимізації запитів MySQL надає команду EXPLAIN яка показує план виконання запиту. Вона відображає які таблиці та індекси будуть використані, скільки рядків буде просканово, який тип з'єднання застосовується та іншу корисну інформацію для оцінки ефективності запиту. Аналіз виводу EXPLAIN допомагає виявити проблемні місця та прийняти рішення про створення нових індексів або переформулювання запиту.

При проектуванні індексної стратегії важливо дотримуватися балансу. Недостатня кількість індексів призводить до повільних запитів особливо на великих таблицях. Надмірна кількість індексів марнотратно використовує дисковий простір та пам'ять, сповільнює модифікацію даних та може навіть погіршити продуктивність вибірки якщо оптимізатор обере неоптимальний індекс. Рекомендується створювати індекси на основі аналізу реальних запитів додатку та моніторингу продуктивності.

4. ЗОВНІШНІ КЛЮЧІ. ЗВ’ЯЗУВАННЯ ТАБЛИЦЬ.

Зовнішні ключі є механізмом забезпечення ссилочної цілісності даних між пов'язаними таблицями. Зовнішній ключ в одній таблиці посилається на

первинний ключ або унікальний ключ іншої таблиці, встановлюючи зв'язок батько-нащадок між ними. MySQL перевіряє що значення зовнішнього ключа завжди відповідає існуючому значенню в батьківській таблиці або є NULL якщо це дозволено, запобігаючи появі сирітських записів.

Створення зовнішнього ключа виконується при визначенні таблиці або додається до існуючої таблиці командою ALTER TABLE. В описі зовнішнього ключа вказується його назва, стовпець або стовпці що утворюють ключ, батьківська таблиця та стовпець на який здійснюється посилання. Рекомендується давати зовнішнім ключам описові назви що відображають зв'язок між таблицями для полегшення подальшого супроводу схеми бази даних.

Опції каскадування визначають поведінку системи при оновленні або видаленні запису в батьківській таблиці на який посилаються дочірні записи. Опція CASCADE означає що зміни автоматично поширюються на пов'язані записи - при видаленні батьківського запису видаляються всі дочірні, при оновленні ключа оновлюються всі посилання. Опція SET NULL встановлює зовнішній ключ у NULL при видаленні або зміні батьківського запису, що дозволяє зберегти дочірні записи але розірвати зв'язок.

Опція RESTRICT забороняє видалення або зміну батьківського запису якщо на нього посилаються дочірні записи, генеруючи помилку при спробі такої операції. Це найбільш суворий варіант що гарантує збереження цілісності шляхом запобігання небезпечним операціям. Опція NO ACTION працює аналогічно RESTRICT та є значенням за замовчуванням. Вибір правильної стратегії каскадування залежить від бізнес-логіки та вимог додатку.

Типи зв'язків між таблицями визначають кардинальність відношення. Зв'язок один-до-багатьох є найпоширенішим типом де один запис з батьківської таблиці може бути пов'язаний з багатьма записами дочірньої таблиці, проте кожен дочірній запис пов'язаний лише з одним батьківським. Наприклад один факультет може мати багато студентів, але кожен студент належить до одного факультету. Реалізується шляхом розміщення зовнішнього ключа в дочірній таблиці що посилається на первинний ключ батьківської таблиці.

Зв'язок багато-до-багатьох означає що записи обох таблиць можуть бути пов'язані з багатьма записами іншої таблиці. Наприклад студенти можуть бути зареєстровані на кілька курсів і кожен курс має багато студентів. Такий зв'язок не може бути безпосередньо реалізований в реляційній базі даних і вимагає створення проміжної асоціативної таблиці що містить зовнішні ключі до обох пов'язаних таблиць. Ці два зовнішні ключі зазвичай разом утворюють складений

первинний ключ проміжної таблиці, також в ній можуть зберігатися додаткові атрибути самого зв'язку.

Зв'язок один-до-одного означає що один запис однієї таблиці пов'язаний максимум з одним записом іншої таблиці і навпаки. Використовується для розділення таблиці на дві частини, наприклад для винесення рідко використовуваних або конфіденційних даних в окрему таблицю. Реалізується шляхом створення зовнішнього ключа з атрибутом UNIQUE в одній з таблиць що посилається на первинний ключ іншої.

Операції об'єднання таблиць дозволяють комбінувати дані з кількох пов'язаних таблиць в одному запиті результату. INNER JOIN повертає тільки ті записи де є співпадіння в обох таблицях за умовою об'єднання. Це найбільш поширений тип JOIN що використовується для отримання пов'язаних даних. Наприклад при об'єднанні таблиць студентів та факультетів внутрішнє з'єднання поверне тільки студентів які мають призначений факультет.

LEFT JOIN або LEFT OUTER JOIN повертає всі записи з лівої таблиці і співпадаючі записи з правої, для записів лівої таблиці без співпадінь в правій таблиці стовпці правої заповнюються NULL. Це корисно коли потрібно отримати всі записи головної таблиці навіть якщо для деяких відсутні пов'язані дані. Наприклад можна вибрати всіх студентів включно з тими хто ще не призначений на факультет.

RIGHT JOIN працює дзеркально до LEFT JOIN повертаючи всі записи правої таблиці і співпадаючі ліво таблиці. На практиці RIGHT JOIN використовується рідше оскільки його завжди можна переписати як LEFT JOIN змінивши порядок таблиць. FULL OUTER JOIN який би повертав всі записи обох таблиць незалежно від співпадінь безпосередньо не підтримується в MySQL проте може бути емульований через об'єднання LEFT та RIGHT JOIN.

CROSS JOIN генерує декартовий добуток таблиць повертаючи всі можливі комбінації записів з обох таблиць. Це рідко потрібна операція яка може створити дуже великий результуючий набір, проте іноді використовується для генерації комбінацій або при роботі з невеликими довідковими таблицями.

При написанні складних запитів з множинними JOIN важливо розуміти порядок виконання та продуктивність. Рекомендується використовувати аліаси для таблиць щоб зробити запит більш компактним та читабельним. Умови об'єднання вказуються в секції ON, тоді як додаткові умови фільтрації розміщуються в WHERE. Правильне індексування стовпців що використовуються в JOIN критично важливе для продуктивності складних запитів.

5. ЗАХИСТ ДАНИХ В MYSQL. АДМІНІСТРУВАННЯ. ПРИВІЛЕЇ.

Система безпеки MySQL побудована на багаторівневій архітектурі контролю доступу що забезпечує захист даних від несанкціонованого доступу та модифікації. Центральним елементом системи безпеки є облікові записи користувачів які ідентифікуються комбінацією імені користувача та хосту з якого дозволено підключення. Така двокомпонентна ідентифікація дозволяє створювати різні облікові записи з однаковим іменем для різних джерел підключення з різними правами доступу.

Створення нового користувача виконується командою CREATE USER з вказанням імені користувача хосту та паролю. Хост може бути вказаний як localhost для локального підключення, як конкретна IP-адреса для підключення з певної машини, або як знак відсотка що дозволяє підключення з будь-якого хосту. Останній варіант слід використовувати обережно з міркувань безпеки, дозволяючи віддалений доступ тільки коли це дійсно необхідно.

При створенні користувача необхідно встановити надійний пароль що відповідає політиці безпеки організації. MySQL підтримує різні плагіни аутентифікації з яких найпоширенішим є mysql_native_password для сумісності зі старішими клієнтами та caching_sha2_password що надає кращу безпеку і є стандартом в новіших версіях. Паролі зберігаються в хешованому вигляді що запобігає їх компрометації навіть при отриманні доступу до системних таблиць.

Зміна паролю користувача виконується командою ALTER USER що дозволяє оновити пароль як для власного облікового запису так і для інших користувачів за наявності відповідних привілеїв. Регулярна зміна паролів адміністративних облікових записів є важливою практикою безпеки. Також можна встановити термін дії паролю після якого користувач буде змушений його змінити.

Видалення користувача виконується командою DROP USER коли обліковий запис більше не потрібен. При цьому автоматично відкликаються всі привілеї цього користувача. Рекомендується регулярно переглядати список користувачів та видаляти застарілі облікові записи для мінімізації поверхні атаки.

Система привілеїв MySQL забезпечує детальний контроль над тим які операції може виконувати користувач з якими об'єктами бази даних. Привілеї організовані ієрархічно на глобальному рівні, рівні бази даних, рівні таблиці та навіть рівні окремих стовпців. Така гранулярність дозволяє реалізувати принцип

мінімальних необхідних привілеїв де кожен користувач має тільки ті права які потрібні для виконання його робочих функцій.

Глобальні привілеї застосовуються до всього сервера MySQL і включають адміністративні операції такі як створення користувачів перезавантаження конфігурації зупинка сервера та інші системні функції. Такі привілеї повинні надаватися дуже обмеженому колу адміністраторів оскільки вони надають широкі повноваження що можуть вплинути на роботу всієї системи.

Привілеї рівня бази даних контролюють доступ до конкретної бази даних та всіх її об'єктів. Користувач може мати різні набори привілеїв для різних баз даних що дозволяє ізолювати доступ до даних різних додатків або відділів. Найпоширеніші привілеї включають CREATE для створення нових об'єктів DROP для їх видалення ALTER для зміни структури та привілеї маніпулювання даними.

Привілеї маніпулювання даними включають SELECT для читання даних INSERT для вставки нових записів UPDATE для зміни існуючих та DELETE для видалення записів. Ці привілеї можуть надаватися на рівні таблиці або навіть окремих стовпців для максимального контролю доступу до чутливих даних. Наприклад користувач може мати доступ на читання всіх стовпців таблиці працівників крім стовпця зарплати.

Надання привілеїв виконується командою GRANT з вказанням привілеїв об'єктів до яких вони застосовуються та користувача який їх отримує. Опція WITH GRANT OPTION дозволяє користувачу передавати свої привілеї іншим користувачам, проте таку можливість слід надавати обережно. Після надання нових привілеїв рекомендується виконати команду FLUSH PRIVILEGES щоб переконатися що зміни негайно набули чинності хоча в більшості випадків це відбувається автоматично.

Відкликання привілеїв виконується командою REVOKE коли користувач більше не повинен мати певний рівень доступу. Це може бути потрібно при зміні ролі співробітника звільненні або виявленні зловживань. Відкликання привілеїв відбувається негайно і користувач втрачає можливість виконувати відповідні операції навіть в активних сесіях.

Перегляд привілеїв користувача виконується командою SHOW GRANTS що відображає всі надані привілеї у вигляді команд GRANT які можна виконати для відтворення цих привілеїв. Це корисно для аудиту безпеки документування конфігурації та перенесення налаштувань між серверами. Інформація про користувачів та їх привілеї зберігається в системній базі даних mysql в таблицях user db tables_priv та columns_priv.

Практика безпеки вимагає створення різних облікових записів для різних ролей в системі. Адміністратор бази даних має повний доступ до всіх баз даних та можливість створювати користувачів і керувати сервером. Користувач додатку має обмежений набір привілеїв достатній для роботи додатку включаючи читання запис та модифікацію даних у призначених таблицях. Аналітик або користувач звітності має тільки привілеї SELECT для читання даних без можливості їх зміни. Резервний користувач має привілеї необхідні для створення резервних копій включаючи SELECT та LOCK TABLES.

Додаткові механізми захисту включають шифрування з'єднань через SSL/TLS що запобігає перехопленню паролів та даних при передачі мережею. MySQL підтримує вимогу використання SSL для певних користувачів через атрибут REQUIRE SSL при створенні або зміні користувача. Для критичних систем рекомендується обов'язкове використання шифрованих з'єднань.

Обмеження ресурсів дозволяє контролювати скільки запитів оновлень та з'єднань може виконати користувач за певний період часу. Це запобігає випадковому або навмисному перевантаженню сервера одним користувачем. Обмеження встановлюються при створенні користувача або додаються пізніше командою ALTER USER.

Резервне копіювання є критично важливим аспектом захисту даних від втрати через апаратні збої помилки програмного забезпечення людські помилки або зловмисні дії. MySQL надає утиліту mysqldump для створення логічних резервних копій які являють собою текстові файли з SQL-командами для відтворення структури та даних. Така копія може включати одну або кілька баз даних тільки структуру тільки дані або все разом.

Процес створення резервної копії виконується запуском mysqldump з командного рядка з вказанням параметрів підключення до сервера та назв баз даних для експорту. Результат може бути перенаправлений у файл який потім зберігається в безпечному місці бажано на іншому фізичному носії або в хмарному сховищі. Для великих баз даних може знадобитися стиснення файлу резервної копії для економії місця.

Відновлення з резервної копії виконується шляхом виконання SQL-команд з файлу копії через клієнт mysql. Перед відновленням рекомендується створити нову порожню базу даних якщо це повне відновлення або переконатися що цільова база в очікуваному стані. Процес відновлення може зайняти значний час для великих баз даних тому важливо мати актуальні копії та регулярно тестувати процедуру відновлення.

Стратегія резервного копіювання повинна включати регулярні повні копії та інкрементальні копії змін між повними копіями. Частота створення копій залежить від критичності даних та частоти їх зміни. Для високонавантажених систем може знадобитися щогодинне або навіть безперервне копіювання транзакційного журналу. Важливо також документувати процедури та регулярно тестувати відновлення щоб переконатися що копії дійсно придатні для використання в разі потреби.

6. ОРГАНІЗАЦІЯ ТРАНЗАКЦІЙ.

Транзакція представляє собою логічну одиницю роботи що складається з однієї або кількох операцій з базою даних які повинні бути виконані атомарно тобто або всі разом або жодна. Концепція транзакцій є фундаментальною для забезпечення цілісності та надійності даних особливо в багатокористувацьких системах де одночасно виконується багато операцій.

Властивості ACID визначають вимоги до транзакційних систем та гарантії які вони надають. Атомарність означає що транзакція є неподільною одиницею роботи всі операції в ній виконуються повністю або не виконується жодна навіть якщо виникне збій в середині виконання. Узгодженість гарантує що транзакція переводить базу даних з одного цілісного стану в інший не порушуючи жодних обмежень цілісності правил та бізнес-логіки.

Ізольованість означає що проміжні результати незавершеної транзакції не видимі іншим транзакціям що виконуються одночасно, кожна транзакція працює так ніби вона виконується одна в системі. Довговічність гарантує що після успішного завершення транзакції її результати зберігаються назавжди навіть у разі наступного збою системи.

Початок транзакції явно позначається командою START TRANSACTION або її синонімом BEGIN. Після цього всі наступні команди модифікації даних виконуються в контексті цієї транзакції і їх ефект не стає постійним поки транзакція не буде зафіксована. Це дозволяє виконати кілька пов'язаних операцій і переконатися що всі вони успішні перед фіксацією змін.

Фіксація транзакції виконується командою COMMIT яка робить всі зміни постійними та видимими для інших користувачів. Після COMMIT неможливо відкотити зміни зроблені в транзакції. Фіксація також звільняє блокування утримувані транзакцією дозволяючи іншим транзакціям отримати доступ до змінених даних.

Відкат транзакції виконується командою ROLLBACK яка скасовує всі зміни зроблені після початку транзакції повертаючи дані до стану перед START

TRANSACTION. Відкат зазвичай виконується при виявленні помилки або порушення бізнес-правил яке робить неможливим успішне завершення всієї логічної операції. Це забезпечує що база даних не залишається в частково оновленому некоректному стані.

Практичним прикладом використання транзакцій є переказ коштів між банківськими рахунками. Операція складається з двох дій зменшення балансу одного рахунку та збільшення балансу іншого. Якщо після зменшення першого рахунку виникне помилка і не вдасться збільшити другий кошти будуть втрачені. Транзакція гарантує що або обидві операції виконаються або жодна і гроші не зникнуть та не здублюються.

Точки збереження SAVEPOINT дозволяють створити проміжну точку всередині транзакції до якої можна відкотити зміни не скасовуючи всю транзакцію. Це корисно для складних довгих транзакцій де можливі часткові помилки та бажано зберегти вже виконану роботу. Після створення точки збереження командою SAVEPOINT з іменем можна виконати ROLLBACK TO з цим іменем для відкату до цієї точки зберігаючи всі зміни зроблені до неї.

Рівні ізоляції транзакцій визначають як транзакції взаємодіють між собою і які аномалії можливі при паралельному виконанні. READ UNCOMMITTED є найнижчим рівнем де транзакція може бачити незафіксовані зміни інших транзакцій що може призвести до брудного читання коли читаються дані які потім будуть відкочені. Цей рівень використовується рідко переважно для деяких видів звітності де точність не критична.

READ COMMITTED гарантує що транзакція бачить тільки зафіксовані зміни інших транзакцій запобігаючи брудному читанню. Проте можливе неповторюване читання коли повторна вибірка тих самих даних всередині транзакції повертає різні результати якщо інша транзакція змінила ці дані між двома читаннями. Цей рівень є компромісом між узгодженістю та продуктивністю.

REPEATABLE READ гарантує що якщо транзакція прочитала певні дані то при повторному читанні вона побачить ті ж самі дані незалежно від того що інші транзакції могли їх змінити. Це досягається утримуванням блокувань на прочитаних даних або використанням багатоверсійності. Проте можливі фантомні читання коли повторний запит повертає інші рядки через вставку або видалення іншою транзакцією.

SERIALIZABLE є найвищим рівнем ізоляції що забезпечує повну ізоляцію транзакцій так ніби вони виконуються послідовно одна за одною. Це запобігає всім аномаліям включаючи фантомні читання проте ціною зниження

паралелізму та потенційно нижчої продуктивності через збільшення блокувань та очікувань. Використовується для критичних операцій де важлива абсолютна узгодженість даних.

Автоматичний режим фіксації autocommit є режимом за замовчуванням в MySQL коли кожна окрема SQL-команда автоматично виконується в окремій транзакції яка негайно фіксується. Це зручно для інтерактивної роботи та простих операцій проте не підходить для логічних операцій що вимагають кількох кроків. Autocommit можна вимкнути встановивши відповідну змінну сесії після чого потрібно явно використовувати COMMIT або ROLLBACK.

Блокування записів використовується для запобігання конфліктам при одночасному доступі кількох транзакцій до тих самих даних. SELECT з опцією FOR UPDATE встановлює ексклюзивне блокування на вибрані рядки запобігаючи іншим транзакціям читати або змінювати ці дані поки блокування не буде звільнено. Опція LOCK IN SHARE MODE встановлює спільне блокування що дозволяє іншим транзакціям читати дані але не змінювати їх.

При роботі з транзакціями важливо мінімізувати час їх виконання щоб не затримувати інші транзакції які можуть очікувати доступу до заблокованих даних. Довгі транзакції збільшують ймовірність конфліктів та взаємних блокувань deadlock коли дві або більше транзакції чекають одна одну створюючи циклічну залежність. MySQL автоматично виявляє взаємні блокування та відкочує одну з транзакцій для розв'язання ситуації.

7. ПРОЕКТУВАННЯ БАЗ ДАНИХ ЗА ДОПОМОГОЮ ІНСТРУМЕНТАЛЬНИХ ЗАСОБІВ ПРОГРАМИ HEIDISQL

HeidiSQL є безкоштовним графічним інструментом що надає зручний інтерфейс для роботи з MySQL та іншими системами управління базами даних. Програма розроблена спеціально для Windows проте може працювати на інших платформах через Wine. HeidiSQL об'єднує функціональність клієнта бази даних редактора запитів інструменту адміністрування та засобу проектування в одному додатку.

Встановлення HeidiSQL є простим процесом завантаження інсталятора з офіційного сайту та його запуску. Програма не вимагає складного налаштування і готова до роботи одразу після встановлення. Розмір програми невеликий і вона працює швидко навіть на скромному обладнанні. HeidiSQL підтримує портативний режим коли програма може працювати з USB-накопичувача без встановлення в систему.

Підключення до сервера MySQL починається з створення нової сесії в менеджері сесій який з'являється при запуску програми. В діалозі налаштування сесії вказується тип мережі який для MySQL зазвичай TCP/IP хост який може бути localhost для локального сервера або IP-адреса або доменне ім'я віддаленого сервера. Також вказується порт за замовчуванням 3306 ім'я користувача та пароль.

Після заповнення параметрів з'єднання можна протестувати підключення натиснувши відповідну кнопку перед збереженням сесії. Це допомагає виявити помилки в налаштуваннях такі як неправильний пароль недоступний хост або закритий порт. Успішно налаштовані сесії зберігаються для швидкого підключення в майбутньому. HeidiSQL дозволяє організувати сесії в папки для зручності коли потрібно працювати з багатьма серверами.

Інтерфейс HeidiSQL розділений на три основні області. Ліва панель відображає дерево об'єктів бази даних включаючи бази даних таблиці представлення процедури функції та тригери. Центральна робоча область містить вкладки для різних режимів роботи з обраним об'єктом. Нижня панель показує лог виконаних запитів повідомлення про помилки та іншу діагностичну інформацію що корисно для відлагодження та моніторингу.

Створення нової бази даних виконується через контекстне меню правий клік на з'єднанні в дереві об'єктів та вибір опції створення. В діалозі створення бази даних вказується її назва обирається кодування символів де utf8mb4 рекомендується для повної підтримки Unicode та collation правила порівняння та сортування рядків. HeidiSQL відразу генерує відповідну SQL-команду яку можна переглянути та при потребі відредагувати перед виконанням.

Створення таблиць в HeidiSQL може виконуватися як через графічний інтерфейс так і за допомогою SQL-команд. Графічний спосіб передбачає використання редактора таблиць де на вкладці Columns додаються стовпці з вказанням їх назв типів даних довжини та різних атрибутів. Інтерфейс дозволяє перетягувати стовпці для зміни їх порядку встановлювати прапорці для NOT NULL AUTO_INCREMENT та інших властивостей вибирати значення за замовчуванням.

Вкладка Indexes дозволяє управляти індексами таблиці. Можна додати новий індекс вказавши його назву тип звичайний унікальний або повнотекстовий та стовпці які він включає. Для складених індексів стовпці додаються у потрібному порядку що важливо для оптимальної роботи індексу. HeidiSQL відображає існуючі індекси включаючи автоматично створені для первинних та зовнішніх ключів.

Вкладка Foreign keys призначена для визначення зв'язків між таблицями. При додаванні зовнішнього ключа вказується його назва стовпець в поточній таблиці батьківська таблиця та відповідний стовпець в ній. Також обираються опції каскадування для дій оновлення та видалення. HeidiSQL допомагає уникнути помилок перевіряючи що типи даних пов'язаних стовпців співпадають та що існує індекс на стовпці зовнішнього ключа.

Робота з даними в HeidiSQL дуже зручна завдяки табличному інтерфейсу на вкладці Data. Дані відображаються в сітці схожій на електронну таблицю де можна безпосередньо редагувати значення подвійним кліком на комірці. Зміни підсвічуються кольором і не застосовуються негайно - спочатку можна переглянути всі зміни і потім зберегти їх одночасно натиснувши кнопку збереження або відкинути скасувавши зміни.

Додавання нових рядків виконується кнопкою Insert або клавішею вставки після чого з'являється новий порожній рядок для заповнення. Видалення рядків виконується вибором їх та натисканням Delete. Для великих таблиць HeidiSQL підтримує посторінкову навігацію та можливість фільтрації даних для показу тільки потрібних записів. Також можна швидко відсортувати дані клікнувши на заголовок стовпця.

Редактор SQL-запитів на вкладці Query надає повнофункціональне середовище для написання та виконання SQL-команд. Редактор підтримує підсвічування синтаксису автодоповнення назв таблиць і стовпців форматування коду та збереження запитів у файли для подальшого використання. Можна виконати весь текст запиту або тільки виділену частину що зручно при роботі з кількома запитами в одному вікні.

Результати виконання запитів SELECT відображаються у вигляді таблиці нижче редактора з можливістю експорту в різні формати. Для запитів модифікації даних показується кількість змінених рядків та час виконання. У разі помилки виводиться детальне повідомлення що допомагає знайти та виправити проблему. Історія виконаних запитів зберігається і до попередніх запитів можна швидко повернутися.

Функція експорту даних дозволяє вивантажити структуру та дані бази у різних форматах. SQL-експорт створює текстовий файл з командами для відтворення бази даних який можна виконати на іншому сервері або використати для резервного копіювання. Можна вибрати що експортувати тільки структуру тільки дані або обидва варіанти а також налаштувати різні опції формату та включення службової інформації.

Експорт у CSV корисний для передачі даних в інші програми такі як Excel або для подальшої обробки скриптами. Можна налаштувати роздільник полів символ обрамлення текстових полів та інші параметри формату. XML та JSON експорт надають структуровані формати придатні для обміну даними між різними системами та мовами програмування.

Імпорт даних з файлів підтримує зворотні операції завантаження даних у таблиці. При імпорті CSV HeidiSQL дозволяє налаштувати відповідність між стовпцями файлу та таблиці вказати яким чином обробляти заголовки та порожні значення. Імпорт SQL-файлів виконує команди з файлу що дозволяє відновити базу даних з резервної копії або виконати підготовлений набір команд.

Додаткові інструменти HeidiSQL включають перегляд процесів що працюють на сервері де можна побачити активні запити та при необхідності завершити довгі або проблемні процеси. Перегляд змінних сервера показує поточні налаштування MySQL багато з яких можна змінити на льоту без перезапуску сервера. Інструмент обслуговування таблиць дозволяє виконувати операції оптимізації аналізу та перевірки цілісності таблиць.

HeidiSQL підтримує синхронізацію даних між базами даних що корисно для розгортання змін зі середовища розробки на продакшн або для створення тестових копій баз даних. Інструмент порівняння структур виявляє відмінності в схемах двох баз даних та може згенерувати SQL-скрипт для приведення однієї бази до стану іншої.

Налаштування програми дозволяють персоналізувати інтерфейс змінити шрифти та кольорову схему встановити параметри автозбереження та резервного копіювання запитів налаштувати формати експорту за замовчуванням. HeidiSQL також підтримує розширення функціональності через плагіни та можливість виконання зовнішніх інструментів інтегрованих у меню програми.

При роботі з HeidiSQL важливо розуміти що програма генерує та виконує SQL-команди за кулісами графічного інтерфейсу. Перегляд згенерованого SQL в логу допомагає краще зрозуміти як працюють різні операції та навчитися писати власні запити. Це робить HeidiSQL не тільки інструментом продуктивності але й засобом навчання SQL для початківців.

Використання HeidiSQL особливо ефективне при проектуванні нових баз даних коли потрібно швидко створювати та модифікувати структури експериментувати з різними підходами та відразу бачити результати. Візуальне представлення таблиць зв'язків та даних допомагає краще зрозуміти структуру бази та виявити потенційні проблеми проектування на ранніх етапах розробки.

ТЕМА 11. ВВЕДЕННЯ В MYSQL. ОСНОВНІ ОПЕРАЦІЇ З ДАНИМИ

1. Запити на створення та оновлення схеми баз даних, таблиць та представлень.
2. Поняття індексації даних.
3. Способи організації індексів.
4. Внутрішня мова програмування СУБД.
5. Збережені процедури сервера та тригери.
6. Призначення та переваги. Безпека баз даних.
7. Управління користувачами. Привілеї.

1. ЗАПИТИ НА СТВОРЕННЯ ТА ОНОВЛЕННЯ СХЕМИ БАЗ ДАНИХ, ТАБЛИЦЬ ТА ПРЕДСТАВЛЕНЬ.

Робота з MySQL починається з правильного проектування та створення структури бази даних, що включає визначення схеми, таблиць та допоміжних об'єктів. Схема бази даних являє собою логічну організацію всіх об'єктів та їх взаємозв'язків, що визначає архітектуру всієї системи зберігання даних. Правильне проектування схеми є критично важливим етапом, оскільки помилки на цьому рівні можуть призвести до проблем з продуктивністю, складності підтримки та обмежень масштабованості в майбутньому.

Створення бази даних є першим кроком у побудові системи. Команда CREATE DATABASE ініціює створення нового простору для зберігання таблиць та інших об'єктів. При створенні бази даних критично важливо правильно вказати кодування символів, оскільки воно визначає які символи можуть зберігатися та як вони будуть сортуватися. Кодування utf8mb4 стало стандартом де-факто для нових проектів, оскільки забезпечує повну підтримку Unicode включаючи емодзі та рідкісні символи з різних мов світу. Правило порівняння collation визначає як порівнюються та сортуються рядки, наприклад utf8mb4_unicode_ci забезпечує коректне сортування для більшості мов з урахуванням національних особливостей.

Модифікація існуючої бази даних може знадобитися для зміни її характеристик, найчастіше це стосується кодування. Команда ALTER DATABASE дозволяє змінити параметри бази даних, проте слід бути обережним оскільки зміна кодування існуючої бази з даними може призвести до пошкодження інформації якщо не виконати попереднє перетворення. Видалення бази даних командою DROP DATABASE є необоротною операцією що знищує

всі таблиці, дані та об'єкти в базі, тому завжди потрібно мати актуальну резервну копію перед виконанням таких критичних дій.

Створення таблиць потребує ретельного планування структури даних. Кожна таблиця представляє певну сутність предметної області, наприклад студентів, курси або замовлення. При визначенні таблиці необхідно продумати всі атрибути сутності та вибрати відповідні типи даних для їх зберігання. Вибір типу даних впливає не лише на те які значення можна зберігати, але й на обсяг дискового простору що займає таблиця, швидкість виконання запитів та можливості індексування.

Обмеження цілісності визначені при створенні таблиці забезпечують автоматичну валідацію даних на рівні бази даних. Обмеження NOT NULL гарантує що поле завжди матиме значення, що критично важливо для обов'язкових атрибутів таких як ім'я клієнта або дата замовлення. Обмеження UNIQUE забезпечує унікальність значень, що потрібно для ідентифікаторів, електронних адрес, номерів документів. Обмеження CHECK дозволяє визначити складні умови валідації, наприклад що вік повинен бути додатнім або дата народження не може бути в майбутньому.

Первинний ключ є обов'язковим елементом правильно спроектованої таблиці, що забезпечує унікальну ідентифікацію кожного запису. Найпростішим підходом є використання суррогатного ключа - штучного числового ідентифікатора з автоматичним інкрементом, який не несе бізнес-значення але гарантує простоту та стабільність. Альтернативою є природний ключ складений з реальних атрибутів сутності, проте такі ключі можуть бути громіздкими та нестабільними при зміні бізнес-правил.

Зовнішні ключі встановлюють зв'язки між таблицями та забезпечують посилочну цілісність. При визначенні зовнішнього ключа важливо правильно налаштувати правила каскадування які визначають що відбувається при зміні або видаленні пов'язаного запису. Каскадне видалення CASCADE автоматично видаляє залежні записи що підходить для зв'язків де дочірні сутності не мають сенсу без батьківської. Встановлення NULL підходить для необов'язкових зв'язків де дочірня сутність може існувати самостійно. Заборона RESTRICT запобігає видаленню записів на які посилаються інші, що найбезпечніше але потребує ручного управління залежностями.

Модифікація структури таблиць після їх створення є звичайною практикою в процесі розвитку системи. Команда ALTER TABLE надає широкі можливості для зміни структури без втрати даних. Додавання нових стовпців розширює схему для підтримки нових вимог, при цьому існуючі дані

залишаються недоторканими а нові стовпці заповнюються значеннями за замовчуванням або NULL. Видалення стовпців дозволяє очистити схему від застарілих полів, проте це незворотна операція що знищує всі дані в цих стовпцях.

Зміна типу даних стовпця може знадобитися коли початковий вибір виявився недостатнім, наприклад коли поле для коротких кодів потрібно розширити для довших значень. MySQL намагається автоматично конвертувати існуючі дані в новий тип, проте не всі перетворення можливі і деякі можуть призвести до втрати точності або обрізання даних. Тому перед зміною типу важливо перевірити чи всі існуючі значення коректно конвертуються.

Перейменування стовпців та таблиць дозволяє привести схему у відповідність до змінених стандартів іменування або кращого розуміння предметної області. При перейменуванні об'єктів важливо врахувати що це вплине на весь код додатку що посилається на ці об'єкти, тому такі зміни потребують координації з розробниками та оновлення всіх залежних компонентів системи.

Представлення або VIEW є віртуальними таблицями що визначаються запитом SELECT і не зберігають дані фізично. Представлення створюються для спрощення доступу до даних, приховування складності об'єднань, забезпечення безпеки через обмеження доступу до певних стовпців або рядків, та для створення різних перспектив даних для різних користувачів. Коли користувач звертається до представлення, MySQL виконує його визначальний запит та повертає результат так ніби це звичайна таблиця.

Представлення особливо корисні для реалізації рівня абстракції між структурою бази даних та додатком. Якщо структура таблиць змінюється, можна оновити визначення представлення щоб воно повертало дані в очікуваному форматі, і додаток продовжить працювати без змін. Це знижує зв'язаність між рівнями системи та полегшує еволюцію схеми бази даних.

Матеріалізовані представлення на відміну від звичайних фізично зберігають результати запиту і оновлюються періодично або за подією. MySQL не має вбудованої підтримки матеріалізованих представлень, проте їх можна емулювати через звичайні таблиці з періодичним перезавантаженням даних за допомогою тригерів або запланованих завдань. Матеріалізовані представлення значно прискорюють складні аналітичні запити оскільки дані вже попередньо обчислені та збережені.

Видалення представлень виконується командою DROP VIEW коли вони більше не потрібні або для їх перестворення зі зміненим визначенням. Видалення

представлення не впливає на базові таблиці та їх дані, знищується лише визначення запиту. Проте якщо на представлення посилаються інші об'єкти такі як інші представлення або процедури, їх потрібно буде оновити або також видалити.

2. ПОНЯТТЯ ІНДЕКСАЦІЇ ДАНИХ.

Індексація є одним з найпотужніших механізмів оптимізації продуктивності баз даних, що дозволяє різко прискорити операції пошуку та вибірки даних. Суть індексації полягає у створенні додаткових структур даних, що містять впорядковані копії певних стовпців з посиланнями на відповідні рядки в таблиці. Це аналогічно предметному покажчику в книзі - замість перегортання всіх сторінок для пошуку терміну, можна швидко знайти потрібні сторінки в покажчику.

Без індексів база даних змушена виконувати повне сканування таблиці для знаходження потрібних записів, що означає послідовне читання кожного рядка та перевірку чи відповідає він умові запиту. Для невеликих таблиць з сотнями або кількома тисячами записів це може бути прийнятно, проте для великих таблиць з мільйонами записів повне сканування стає неприпустимо повільним. Індекс перетворює лінійний пошук зі складністю O(n) в логарифмічний пошук зі складністю O(log n), що дає драматичне прискорення на великих об'ємах даних.

Принцип роботи індексу базується на впорядкованому зберіганні значень індексованих стовпців. Коли виконується запит з умовою за індексованим стовпцем, MySQL може використати бінарний пошук або подібні алгоритми для швидкого знаходження потрібних значень в індексі. Індекс містить не самі дані а посилання на рядки таблиці де знаходяться повні записи, тому після знаходження значення в індексі виконується додатковий доступ до таблиці для отримання решти полів.

Вартість індексування полягає в тому що індекси займають додатковий дисковий простір та пам'ять. Кожен індекс є окремою структурою яка дублює частину даних з таблиці в впорядкованому вигляді. Для великих таблиць індекси можуть займати значний обсяг, іноді порівнянний з розміром самої таблиці або навіть більший якщо індексовано багато стовпців чи є складені індекси з багатьма полями.

Операції модифікації даних сповільнюються при наявності індексів, оскільки при вставці, оновленні або видаленні запису потрібно також оновити всі індекси що включають змінені стовпці. Чим більше індексів має таблиця тим повільніше виконуються операції запису. Це створює фундаментальний

компроміс між швидкістю читання та швидкістю запису, що потребує збалансованої стратегії індексування відповідно до характеру навантаження системи.

Вибір стовпців для індексування повинен базуватися на аналізі реальних запитів додатку. Найкращими кандидатами для індексування є стовпці що часто використовуються в умовах WHERE оскільки саме там індекси найбільш ефективні для фільтрації даних. Стовпці що використовуються в JOIN також критично потребують індексів, особливо зовнішні ключі, оскільки об'єднання таблиць без індексів може призвести до катастрофічного погіршення продуктивності через декартові добутки.

Стовпці для ORDER BY та GROUP BY теж виграють від індексування, оскільки індекс вже містить дані в впорядкованому вигляді і сортування може виконатися простим проходом по індексу замість дорогої операції сортування всього набору результатів. Проте індекс буде корисним для сортування тільки якщо порядок стовпців в індексі співпадає з порядком в ORDER BY.

Селективність індексу визначає наскільки ефективно він відфільтровує дані. Висока селективність означає що більшість значень унікальні і індекс дозволяє швидко звузити пошук до невеликого набору рядків. Низька селективність означає багато дублюючих значень і індекс буде менш корисним. Наприклад індекс на стовпці статі з лише двома можливими значеннями має низьку селективність і рідко буде використаний оптимізатором, тоді як індекс на електронній адресі має високу селективність та дуже ефективний.

Унікальні індекси поєднують функції оптимізації та забезпечення цілісності даних. Крім прискорення пошуку вони гарантують що не може бути двох записів з однаковим значенням індексованого стовпця. Це корисно для реалізації бізнес-правил на рівні бази даних, наприклад забезпечення унікальності електронних адрес, номерів документів або логінів користувачів. Спроба вставити або оновити запис з дублюючим значенням призведе до помилки що запобігає порушенню цілісності.

Первинний ключ автоматично отримує унікальний індекс що забезпечує швидкий доступ до записів за їх ідентифікатором. Це найважливіший індекс в таблиці оскільки доступ за первинним ключем є найпоширенішою операцією і повинен бути максимально швидким. В InnoDB первинний ключ має особливе значення оскільки дані фізично організовані за його значенням в кластерному індексі.

Покриваючі індекси є особливою оптимізацією коли індекс містить всі стовпці необхідні для виконання запиту. В такому випадку MySQL може

отримати всі потрібні дані безпосередньо з індексу не звертаючись до таблиці, що значно швидше особливо для великих таблиць. Створення покриваючих індексів потребує включення в індекс не лише стовпців з умов але й стовпців що вибираються в SELECT.

Частковий індекс індексує не повне значення стовпця а лише його початкову частину заданої довжини. Це корисно для текстових полів великого розміру де індексування повного значення було б неефективним через великий розмір індексу. Частковий індекс займає менше місця та швидше обробляється, проте може бути менш селективним якщо значення відрізняються лише в кінці рядка.

Функціональні індекси дозволяють індексувати не саме значення стовпця а результат застосування до нього функції або виразу. Наприклад можна створити індекс на нижньому регістрі текстового поля для регістронезалежного пошуку або на році з поля дати для швидкого фільтрування за роком. Підтримка функціональних індексів з'явилася в пізніших версіях MySQL та значно розширює можливості оптимізації.

3. СПОСОБИ ОРГАНІЗАЦІЇ ІНДЕКСІВ.

Структура даних яка використовується для реалізації індексу критично впливає на його характеристики та область застосування. MySQL підтримує кілька типів індексів кожен з яких оптимізований для певних сценаріїв використання. Розуміння внутрішньої організації індексів допомагає приймати правильні рішення при проектуванні схеми бази даних та оптимізації запитів.

B-Tree індекси є найпоширенішим типом індексів в MySQL та використовуються за замовчуванням для більшості випадків. Назва походить від збалансованого дерева balanced tree, хоча фактична реалізація зазвичай використовує варіант B+Tree де всі дані зберігаються в листових вузлах а внутрішні вузли містять лише ключі для навігації. Дерево автоматично балансується при вставці та видаленні, забезпечуючи що глибина всіх гілок приблизно однакова і пошук завжди виконується за логарифмічний час.

Структура B-Tree особливо ефективна для діапазонних запитів оскільки листові вузли зв'язані в послідовний список і можна швидко прочитати послідовність значень після знаходження початку діапазону. Це робить B-Tree ідеальним для умов з операторами порівняння менше більше BETWEEN та для сортування. B-Tree також підтримує префіксний пошук для текстових полів дозволяючи ефективно виконувати запити з LIKE що починаються з постійного префікса.

Кластерний індекс в механізмі зберігання InnoDB організує фізичне розміщення даних таблиці відповідно до значень первинного ключа. На відміну від вторинних індексів які містять лише копію індексованих стовпців та посилання на рядки, кластерний індекс містить повні дані всіх стовпців таблиці впорядковані за первинним ключем. Це означає що доступ за первинним ключем не потребує додаткового пошуку в таблиці - дані знаходяться безпосередньо в індексі.

Кластерна організація має важливі наслідки для продуктивності. Запити за первинним ключем надзвичайно швидкі оскільки потребують лише одного пошуку в індексі. Діапазонні запити за первинним ключем також дуже ефективні через послідовне розміщення даних. Проте вставка нових записів може бути повільнішою якщо первинний ключ не монотонно зростає, оскільки потрібне переміщення даних для підтримки впорядкованості.

Вторинні індекси в InnoDB містять індексовані значення разом зі значенням первинного ключа замість фізичного адресу рядка. Це дозволяє підтримувати узгодженість індексів при реорганізації даних, проте означає що доступ через вторинний індекс потребує двох пошуків - спочатку в вторинному індексі для отримання первинного ключа, потім в кластерному індексі для отримання повних даних рядка. Тому важливо щоб первинний ключ був компактним інакше вторинні індекси стануть надто великими.

Хеш-індекси використовують хеш-таблицю для відображення значень у позиції рядків. Вони надзвичайно швидкі для точного пошуку за рівністю оскільки хеш-функція дозволяє знайти потрібний запис за один крок без проходження дерева. Проте хеш-індекси не підтримують діапазонні запити або сортування оскільки хеш-функція розподіляє значення випадково без збереження порядку. В MySQL хеш-індекси підтримуються механізмом MEMORY та створюються автоматично механізмом InnoDB для часто запитуваних індексованих значень в адаптивному хеш-індексі.

Повнотекстові індекси FULLTEXT призначені для швидкого пошуку слів та фраз у великих текстових полях. Вони використовують інвертований індекс який для кожного слова зберігає список документів де воно зустрічається разом з позиціями входжень. Це дозволяє швидко знаходити документи що містять задані слова виконувати булеві запити з операторами AND OR NOT, та ранжувати результати за релевантністю на основі частоти зустрічання термінів.

Повнотекстовий пошук значно потужніший за оператор LIKE для складних текстових запитів. LIKE з шаблоном що містить знаки підстановки всередині або на початку вимагає повного сканування таблиці і дуже повільний

на великих даних. Повнотекстовий індекс дозволяє знайти документи за секунди навіть у таблицях з мільйонами записів. Також підтримується природномовний режим який враховує стоп-слова, стемінг та інші лінгвістичні особливості для більш релевантних результатів.

Просторові індекси підтримуються для геопросторових типів даних та використовують спеціальні структури даних такі як R-Tree оптимізовані для багатовимірних даних. Вони дозволяють ефективно виконувати геометричні запити такі як знаходження всіх точок у заданому прямокутнику або полігоні, пошук найближчих об'єктів, перевірка перетину геометричних фігур. Це критично важливо для геоінформаційних систем додатків з картами та локаційних сервісів.

Складені або композитні індекси включають кілька стовпців в один індекс і є потужним інструментом оптимізації складних запитів. Порядок стовпців у складеному індексі має критичне значення для його ефективності. Індекс можна використати для запитів які фільтрують за префіксом стовпців індексу - якщо індекс створено на стовпцях A B C, він буде використаний для умов на A, A і B, A і B і C, але не на B, C або B і C окремо.

Стратегія побудови складених індексів полягає в розміщенні більш селективних стовпців першими щоб максимально звузити пошук на ранніх етапах. Проте якщо запити часто фільтрують за певним стовпцем окремо, він повинен бути першим навіть якщо має нижчу селективність. Іноді доводиться створювати кілька індексів з різним порядком стовпців для підтримки різних варіантів запитів.

Індекси сортування дозволяють уникнути дорогої операції сортування якщо результати вже впорядковані в індексі. Для цього порядок стовпців в ORDER BY повинен відповідати порядку в індексі включаючи напрямок сортування зростання чи спадання. Можна створювати індекси з різним напрямком для окремих стовпців щоб підтримати складні сортування, проте це збільшує загальну кількість індексів та відповідно вартість їх підтримки.

Стратегії оптимізації індексів включають регулярний моніторинг використання індексів для виявлення недовикористовуваних які можна видалити для зменшення навантаження на операції запису. Інструменти аналізу запитів показують які індекси використовуються а які ігноруються оптимізатором. Іноді індекс не використовується через неоптимальне формулювання запиту і переписування запиту може дозволити використати існуючий індекс замість створення нового.

4. ВНУТРІШНЯ МОВА ПРОГРАМУВАННЯ СУБД.

MySQL надає потужну внутрішню мову програмування що дозволяє реалізувати складну бізнес-логіку безпосередньо в базі даних. На відміну від звичайних SQL-запитів які виконують окремі операції з даними, процедурні розширення дозволяють писати програми з умовами, циклами, обробкою помилок та іншими конструкціями повноцінної мови програмування. Це відкриває можливості для створення складних алгоритмів обробки даних що виконуються безпосередньо на сервері бази даних.

Збережені процедури є іменованими блоками коду які можна викликати з додатку передаючи параметри та отримуючи результати. Процедура створюється один раз і зберігається в базі даних, після чого може виконуватися багаторазово. Це забезпечує інкапсуляцію логіки та повторне використання коду - замість дублювання однієї і тієї ж послідовності операцій у різних місцях додатку, логіка централізується в процедурі яку викликають за потреби.

Переваги збережених процедур включають зменшення мережевого трафіку оскільки замість передачі багатьох окремих SQL-команд від клієнта до сервера передається лише один виклик процедури з параметрами. Це особливо важливо для складних операцій що потребують багатьох кроків та можуть виконуватися десятки разів швидше коли логіка виконується на сервері. Також поліпшується безпека оскільки можна надати користувачам право виконання процедури не даючи прямого доступу до таблиць.

Параметри процедур можуть бути вхідними IN що передають значення в процедуру, вихідними OUT що повертають значення з процедури, або двонапрямленими INOUT що дозволяють і передавати і отримувати значення. Це забезпечує гнучкий механізм обміну даними між додатком та процедурою. Типи параметрів повинні бути явно вказані при оголошенні процедури що забезпечує контроль типів та запобігає помилкам.

Змінні в процедурах оголошуються командою DECLARE з вказанням імені типу та необов'язкового початкового значення. Змінні існують лише в межах блоку в якому оголошені та автоматично знищуються при виході з блоку. Присвоєння значень змінним виконується командою SET або через SELECT INTO який дозволяє присвоїти результат запиту змінним.

Умовні конструкції IF дозволяють виконувати різні дії залежно від умов. Синтаксис включає умову блок команд для виконання якщо умова істинна необов'язкові секції ELSEIF для перевірки додаткових умов та ELSE для дій за замовчуванням. Умови можуть бути довільно складними з використанням логічних операторів порівнянь та викликів функцій.

Конструкція CASE надає більш компактний синтаксис для вибору між багатьма альтернативами. Існує два варіанти - простий CASE який порівнює вираз з набором значень та пошуковий CASE який оцінює список умов. CASE може використовуватися як оператор в процедурному коді або як вираз в SELECT та інших командах SQL.

Цикли дозволяють повторювати блок команд багато разів що необхідно для обробки наборів даних або виконання ітераційних алгоритмів. Цикл LOOP виконується безкінечно поки не зустріне команду LEAVE для виходу. Цикл WHILE виконується поки умова істинна перевіряючи її перед кожною ітерацією. Цикл REPEAT виконується поки умова хибна перевіряючи її після кожної ітерації що гарантує хоча б одне виконання тіла циклу.

Курсори надають механізм для послідовної обробки результатів запиту по одному рядку за раз. Курсор оголошується для певного SELECT запиту, потім відкривається для виконання запиту, і рядки витягуються по черзі в змінні для обробки. Після обробки всіх рядків курсор закривається для звільнення ресурсів. Курсори корисні коли потрібна складна логіка обробки кожного рядка яку неможливо виразити одним SQL-запитом.

Обробка помилок реалізується через оголошення обробників HANDLER які визначають дії при виникненні певних умов. Обробник може перехоплювати специфічні коди помилок, класи помилок або загальні умови такі як NOT FOUND для кінця даних у курсорі. Обробники можуть продовжити виконання CONTINUE після обробки помилки або вийти з блоку EXIT. Правильна обробка помилок критично важлива для надійності процедур особливо при роботі з транзакціями.

Функції схожі на процедури але повертають одне значення і можуть використовуватися в виразах SQL. На відміну від процедур які викликаються окремою командою CALL, функції викликаються як частина виразу в SELECT WHERE або інших місцях де очікується значення. Функції повинні бути детерміністичними тобто завжди повертати однаковий результат для однакових параметрів або бути позначені як NONDETERMINISTIC.

Функції корисні для інкапсуляції складних обчислень або бізнес-правил які потрібно застосовувати в багатьох місцях. Наприклад функція для розрахунку знижки з урахуванням різних факторів може викликатися в запитах замість дублювання складної формули. Функції також забезпечують абстракцію дозволяючи змінити логіку обчислення в одному місці без модифікації всіх запитів які її використовують.

5. ЗБЕРЕЖЕНІ ПРОЦЕДУРИ СЕРВЕРА ТА ТРИГЕРИ.

Збережені процедури представляють один з найпотужніших механізмів для реалізації складної бізнес-логіки безпосередньо в базі даних. Процедура являє собою іменований набір SQL-команд та процедурних конструкцій який компілюється один раз при створенні і потім може виконуватися багаторазово з різними параметрами. Компіляція та оптимізація процедури при створенні означає що подальші виклики виконуються швидше ніж еквівалентна послідовність окремих команд надісланих з клієнта.

Створення процедури виконується командою CREATE PROCEDURE з вказанням імені списку параметрів та тіла процедури. Тіло обмежується ключовими словами BEGIN та END і може містити довільно складну логіку з локальними змінними умовами циклами курсорами та викликами інших процедур. Важливим аспектом є вибір роздільника оскільки тіло процедури містить крапки з комою для розділення команд і потрібен спосіб позначити кінець всього визначення процедури.

Параметри процедури визначають інтерфейс взаємодії з зовнішнім кодом. Вхідні параметри IN дозволяють передати дані в процедуру для обробки. Всередині процедури такі параметри поводяться як константи і їх не можна змінювати. Вихідні параметри OUT призначені для повернення результатів з процедури - при виклику для них передаються змінні які будуть заповнені значеннями при завершенні процедури. Параметри INOUT поєднують обидві функції дозволяючи і отримати вхідне значення і повернути результат через той самий параметр.

Виклик процедури з додатку виконується командою CALL з передачею фактичних значень або змінних для параметрів. Після виконання процедури змінні передані для OUT та INOUT параметрів містять повернуті значення які можна прочитати. Процедура також може повертати набори результатів виконуючи SELECT запити без присвоєння результату змінним - такі набори передаються клієнту як результат виклику.

Модифікація існуючих процедур не підтримується напряму - для зміни логіки потрібно спочатку видалити процедуру командою DROP PROCEDURE і створити її заново з оновленим кодом. Це може бути проблематичним для продакшн систем де процедури активно використовуються, тому зміни потрібно координувати з періодами низького навантаження. Також важливо враховувати що видалення процедури на яку посилаються інші об'єкти може призвести до помилок.

Збережені процедури часто використовуються для реалізації складних бізнес-операцій які потребують кількох кроків та перевірок. Наприклад процедура оформлення замовлення може перевіряти наявність товару резервувати його зменшувати залишки створювати записи замовлення та позицій обчислювати загальну суму застосовувати знижки та виконувати інші необхідні дії - все це в одній транзакції з правильною обробкою помилок та відкатом у разі проблем.

Тригери є особливим типом збережених процедур які виконуються автоматично у відповідь на певні події модифікації даних в таблиці. На відміну від звичайних процедур які викликаються явно, тригери спрацьовують неявно коли відбувається подія для якої вони визначені. Це дозволяє автоматизувати виконання дій при зміні даних забезпечуючи узгодженість додаткову валідацію аудит або каскадні оновлення пов'язаних даних.

Тригер пов'язується з конкретною таблицею та одною з трьох операцій - INSERT UPDATE або DELETE. Також вказується час спрацювання - BEFORE перед виконанням операції або AFTER після її виконання. Це дає шість можливих комбінацій для кожної таблиці проте на одну комбінацію може бути визначений тільки один тригер в старих версіях MySQL, новіші версії підтримують множинні тригери з визначеним порядком виконання.

Тригери BEFORE виконуються перед тим як дані фактично змінюються в таблиці і можуть модифікувати значення які будуть записані або навіть заборонити операцію генеруючи помилку. Це корисно для додаткової валідації даних яку складно або неможливо виразити через обмеження CHECK, для автоматичного обчислення значень похідних полів, або для стандартизації даних наприклад приведення тексту до нижнього регістру.

Тригери AFTER виконуються після успішної зміни даних і використовуються для дій які повинні відбутися як наслідок операції але не впливають на самі дані що змінюються. Типові застосування включають логування змін в таблицю аудиту оновлення агрегатних значень в інших таблицях відправку повідомлень або подій для зовнішніх систем. Тригер AFTER не може змінити значення які вже записані але може модифікувати інші таблиці.

Всередині тригера доступні спеціальні псевдо-записи OLD та NEW які представляють стан рядка до та після операції. Для INSERT доступний тільки NEW оскільки немає попереднього стану. Для DELETE доступний тільки OLD оскільки немає нового стану. Для UPDATE доступні обидва дозволяючи порівняти старі та нові значення і діяти відповідно до того які саме поля змінилися.

Тригери мають важливі обмеження які потрібно враховувати. Вони не можуть повертати результати клієнту або використовувати команди які змінюють схему бази даних. Тригер не може модифікувати ту саму таблицю до якої він прив'язаний оскільки це призвело б до безкінечної рекурсії. Помилка в тригері призводить до відкату всієї операції яка його викликала тому критично важлива ретельна обробка помилок та тестування.

Складність відлагодження тригерів пов'язана з їх неявною природою - вони виконуються автоматично і можуть бути не очевидні для розробників які модифікують дані. Це може призвести до непередбачуваної поведінки коли проста операція INSERT викликає каскад тригерів які виконують багато додаткових дій. Тому важлива детальна документація всіх тригерів та обережне планування логіки їх взаємодії.

Продуктивність може страждати від надмірного використання тригерів особливо якщо вони виконують складні операції або модифікують багато пов'язаних таблиць. Кожна операція вставки оновлення або видалення стає повільнішою через виконання тригерів що може бути неприйнятним для високонавантажених систем. Іноді логіку краще реалізувати на рівні додатку де є більше контролю над виконанням та легше оптимізувати продуктивність.

6. ПРИЗНАЧЕННЯ ТА ПЕРЕВАГИ. БЕЗПЕКА БАЗ ДАНИХ.

Використання збережених процедур та тригерів надає значні переваги в архітектурі додатків що працюють з базами даних. Централізація бізнес-логіки в базі даних забезпечує єдину точку визначення правил які застосовуються незалежно від того який додаток або компонент системи працює з даними. Це критично важливо для забезпечення узгодженості в середовищах де кілька різних додатків написаних різними мовами програмування отримують доступ до однієї бази даних.

Зменшення мережевого трафіку є значною перевагою особливо для операцій які потребують багатьох окремих запитів. Замість передачі десятків SQL-команд від клієнта до сервера та назад, процедура викликається один раз з параметрами виконує всю необхідну роботу на сервері і повертає лише кінцевий результат. Для додатків що працюють через повільні або нестабільні мережеві з'єднання це може дати драматичне поліпшення продуктивності та надійності.

Поліпшення продуктивності досягається через попередню компіляцію та оптимізацію процедур. Коли процедура створюється MySQL аналізує її код будує план виконання та зберігає його в оптимізованій формі. При кожному наступному виклиці використовується готовий план що швидше ніж парсинг та

оптимізація кожного запиту знову. Також процедури виконуються безпосередньо на сервері близько до даних що усуває накладні витрати на передачу даних туди і назад.

Безпека посилюється через можливість надання користувачам права виконання процедур без надання прямого доступу до базових таблиць. Процедура може виконуватися з правами її власника який має необхідні привілеї тоді як користувач що викликає процедуру має тільки право EXECUTE на неї. Це дозволяє контролювати які саме операції можуть виконуватися з даними запобігаючи несанкціонованій модифікації або витоку інформації.

Інкапсуляція складності дозволяє приховати деталі реалізації від додатку надаючи простий інтерфейс для складних операцій. Додаток викликає процедуру з бізнес-назвою та параметрами не знаючи і не потребуючи знати деталей того як дані зберігаються в таблицях та які саме запити виконуються. Це знижує зв'язаність між рівнями системи та полегшує модифікацію реалізації без зміни інтерфейсу.

Повторне використання коду забезпечується тим що одна процедура може викликатися з багатьох місць в додатку або навіть з різних додатків. Замість дублювання однієї і тієї ж логіки у кожному місці де вона потрібна код пишеться один раз в процедурі і викликається за потреби. Це спрощує підтримку оскільки зміни потрібно вносити лише в одному місці та зменшує ймовірність помилок через неузгоджені копії логіки.

Забезпечення цілісності даних на рівні бази даних через тригери гарантує що правила виконуються завжди незалежно від того як саме змінюються дані. Навіть якщо додаток має помилку або користувач виконує прямі команди SQL через адміністративний інтерфейс тригери спрацюють і забезпечать узгодженість. Це надійніше ніж покладатися виключно на логіку в коді додатку яка може бути обійдена.

Автоматизація рутинних завдань через тригери економить час розробників та зменшує ймовірність помилок. Такі задачі як заповнення службових полів типу дата створення або модифікації автоматичне обчислення похідних значень підтримка денормалізованих даних для продуктивності можуть бути повністю автоматизовані через тригери і розробники можуть не думати про них при написанні коду додатку.

Аудит та логування змін реалізуються елегантно через тригери які записують в окрему таблицю інформацію про всі зміни даних включаючи старі та нові значення час зміни та користувача. Це забезпечує повну історію змін для відповідності регуляторним вимогам відстеження проблем та відновлення даних.

Оскільки логування відбувається на рівні бази даних воно працює незалежно від додатку і не може бути обійдене.

Проте є і недоліки які потрібно враховувати. Відлагодження процедур та тригерів складніше ніж звичайного коду додатку оскільки інструменти відлагодження менш розвинені. Тестування також ускладнюється необхідністю встановлення процедур та тригерів в тестовій базі даних та координації версій між кодом бази даних та кодом додатку. Контроль версій для процедур потребує спеціального підходу оскільки вони зберігаються в базі даних а не у файлах вихідного коду.

Перенесення між різними СУБД ускладнюється оскільки синтаксис процедурних розширень відрізняється між MySQL PostgreSQL Oracle та іншими системами. Якщо є ймовірність міграції на іншу платформу варто обмежити використання специфічних для MySQL можливостей або підготуватися до значних зусиль з переписування коду. Деякі команди віддають перевагу тримати всю логіку в додатку щоб залишатися більш незалежними від конкретної СУБД.

Безпека баз даних є критично важливим аспектом будь-якої інформаційної системи оскільки бази даних зберігають найціннішу інформацію організації яка може включати персональні дані клієнтів фінансову інформацію комерційні таємниці та іншу конфіденційну інформацію. Компрометація бази даних може призвести до серйозних наслідків включаючи фінансові втрати репутаційні збитки юридичні проблеми та порушення регуляторних вимог таких як GDPR.

Багаторівнева система безпеки MySQL забезпечує захист на всіх рівнях від мережевого доступу до індивідуальних стовпців таблиць. Першим рівнем захисту є мережева безпека яка контролює хто може підключитися до сервера. За замовчуванням MySQL слухає лише на localhost запобігаючи віддаленим підключенням поки це явно не налаштовано. Фаєрвол повинен бути налаштований для обмеження доступу до порту MySQL лише з довірених IP-адрес.

Аутентифікація перевіряє ідентичність користувача перед наданням доступу до системи. MySQL підтримує різні методи аутентифікації від традиційних паролів до сучасніших підходів з плагінами аутентифікації. Найбільш поширеним є парольна аутентифікація де користувач надає своє ім'я та пароль які перевіряються проти збережених в системі. Паролі зберігаються в хешованому вигляді використовуючи криптографічні хеш-функції що робить їх непридатними для прямого використання навіть при компрометації таблиці користувачів.

Політика паролів визначає вимоги до складності паролів їх мінімальної довжини обов'язкового використання різних типів символів та термінів дії. MySQL підтримує налаштування політики паролів через плагін validate_password який може відхиляти слабкі паролі примушувати користувачів періодично змінювати паролі та запобігати повторному використанню старих паролів. Сильна політика паролів є першою лінією захисту проти несанкціонованого доступу.

Дворівнева ідентифікація в MySQL базується на комбінації імені користувача та хоста з якого здійснюється підключення. Це дозволяє створювати різні облікові записи з однаковим іменем але різними правами залежно від джерела підключення. Наприклад користувач application з localhost може мати одні привілеї тоді як той самий користувач з певної IP-адреси може мати обмеженіший доступ. Це надає гнучкість в налаштуванні безпеки відповідно до топології мережі.

Шифрування з'єднань через SSL/TLS захищає дані під час передачі між клієнтом та сервером від перехоплення та підміни. Без шифрування паролі та дані передаються відкритим текстом який може бути прочитаний будь-ким хто має доступ до мережевого трафіку. Сучасні версії MySQL підтримують вимагання SSL для певних користувачів або глобально для всіх з'єднань. Також можна використовувати сертифікати для додаткової перевірки ідентичності клієнта.

Шифрування даних в спокої захищає інформацію збережену на диску від несанкціонованого доступу через викрадення носіїв резервних копій або прямого доступу до файлів бази даних обійшовши MySQL. InnoDB підтримує прозоре шифрування на рівні таблиць або цілої бази даних де дані автоматично шифруються при записі на диск та розшифровуються при читанні. Ключі шифрування зберігаються окремо та можуть управлятися через спеціалізовані системи управління ключами.

Резервні копії також потребують захисту оскільки вони містять повну копію всіх даних і можуть бути використані для несанкціонованого доступу якщо потраплять в неправильні руки. Копії повинні зберігатися в безпечних місцях з обмеженим доступом бажано в зашифрованому вигляді. При передачі копій через мережу слід використовувати захищені канали. Старі копії повинні надійно знищуватися коли вони більше не потрібні.

Аудит безпеки включає логування та моніторинг всіх спроб доступу до системи особливо невдалих аутентифікацій підозрілих запитів та змін конфігурації безпеки. MySQL Enterprise Edition включає розширені можливості

аудиту проте навіть Community Edition надає базове логування яке можна аналізувати для виявлення аномалій. Регулярний перегляд логів допомагає виявити спроби атак або інсайдерські загрози на ранніх стадіях.

SQL-ін'єкція є однією з найпоширеніших вразливостей веб-додатків що працюють з базами даних. Вона виникає коли додаток некоректно обробляє вхідні дані користувача включаючи їх безпосередньо в SQL-запити без належної санітизації. Атакуючий може ввести спеціально сформовані дані які зміниться логіку запиту дозволяючи обійти аутентифікацію витягнути конфіденційні дані або навіть виконати довільні команди. Захист від SQL-ін'єкцій включає використання підготовлених запитів з параметрами валідацію та санітизацію всіх вхідних даних та застосування принципу мінімальних привілеїв.

Регулярне оновлення програмного забезпечення є критично важливим для безпеки оскільки нові вразливості виявляються постійно і публікуються оновлення безпеки. Слід підписатися на оголошення безпеки MySQL та оперативно встановлювати патчі особливо для критичних вразливостей. Тестування оновлень в непродакшн середовищі перед застосуванням в продакшні допомагає уникнути проблем сумісності.

Принцип мінімальних привілеїв стверджує що кожен користувач процес або програма повинні мати лише ті права які абсолютно необхідні для виконання їх функцій і не більше. Це обмежує потенційну шкоду від компрометованого облікового запису або помилки в коді. Застосування цього принципу вимагає ретельного аналізу вимог та дисципліни в наданні прав. Також важливо регулярно переглядати та відкликати непотрібні привілеї.

7. УПРАВЛІННЯ КОРИСТУВАЧАМИ. ПРИВІЛЕЇ.

Система управління користувачами в MySQL побудована на детальній моделі контролю доступу яка дозволяє точно визначити хто може робити що з якими даними. Кожен користувач ідентифікується унікальною комбінацією імені та хоста і має асоційований набір привілеїв що визначають дозволені операції. Ця система забезпечує гнучкість необхідну для підтримки складних організаційних структур з різними ролями та обов'язками.

Створення користувача є першим кроком в наданні доступу до бази даних. Команда CREATE USER визначає нового користувача з вказанням імені хоста з якого дозволено підключення та методу аутентифікації. Хост може бути точною адресою або використовувати підстановочні символи для групи адрес. Знак відсотка означає будь-який хост проте використання його слід обмежувати з

міркувань безпеки. Локальні підключення позначаються як localhost і відрізняються від підключень через TCP/IP навіть до тієї самої машини.

Вибір імені користувача повинен слідувати певним конвенціям для полегшення управління. Часто використовується підхід де ім'я відображає роль або функцію користувача наприклад app_reader app_writer admin backup. Це робить зрозумілим призначення облікового запису при перегляді списку користувачів. Для людських користувачів можна використовувати реальні імена проте це може створити проблеми конфіденційності та ускладнити управління при звільненні співробітників.

Встановлення паролю при створенні користувача або його зміна пізніше виконується через IDENTIFIED BY з паролем у відкритому вигляді або IDENTIFIED WITH для вибору конкретного плагіну аутентифікації. MySQL автоматично хешує пароль перед збереженням тому він ніколи не зберігається в читабельному вигляді. Адміністратори не можуть побачити паролі користувачів лише змінити їх. Користувачі повинні бути проінструктовані про важливість вибору сильних унікальних паролів та їх конфіденційного збереження.

Зміна паролів виконується користувачами для власних облікових записів або адміністраторами для будь-яких користувачів. Регулярна зміна паролів є хорошою практикою безпеки хоча сучасні рекомендації підкреслюють важливість складності паролю більше ніж частоти зміни. Примусова зміна пароля може бути налаштована для нових користувачів або після скидання паролю адміністратором щоб гарантувати що тільки сам користувач знає свій дійсний пароль.

Видалення користувачів виконується коли обліковий запис більше не потрібен наприклад після звільнення співробітника або виводу додатку з експлуатації. DROP USER негайно видаляє користувача та всі асоційовані з ним привілеї. Існуючі сесії користувача залишаються активними до їх завершення проте нові підключення стають неможливими. Важливо мати процес своєчасного видалення застарілих облікових записів для мінімізації ризиків безпеки.

Система привілеїв MySQL організована ієрархічно з чотирма основними рівнями. Глобальні привілеї застосовуються до всього сервера та всіх баз даних. Вони надаються рідко і зазвичай лише адміністраторам оскільки дають широкі повноваження. Привілеї рівня бази даних застосовуються до конкретної бази та всіх її об'єктів. Привілеї рівня таблиці обмежують доступ до певних таблиць в базі даних. Привілеї рівня стовпця є найбільш детальними дозволяючи контролювати доступ до окремих полів таблиці.

Адміністративні привілеї включають операції управління сервером такі як CREATE USER для створення користувачів RELOAD для перезавантаження привілеїв та кешів SHUTDOWN для зупинки сервера SUPER для виконання адміністративних команд. Ці привілеї повинні надаватися дуже обмеженому колу довірених адміністраторів оскільки вони дозволяють виконувати критичні операції що впливають на всю систему.

Привілеї на об'єкти схеми включають CREATE для створення баз даних та таблиць DROP для їх видалення ALTER для зміни структури INDEX для створення та видалення індексів CREATE VIEW для створення представлень CREATE ROUTINE для створення процедур та функцій TRIGGER для створення тригерів. Ці привілеї зазвичай надаються розробникам та адміністраторам баз даних які відповідають за проектування та еволюцію схеми.

Привілеї маніпулювання даними визначають які операції можна виконувати з даними в таблицях. SELECT дозволяє читати дані і є найбільш поширеним привілеєм необхідним практично всім користувачам. INSERT дозволяє додавати нові записи. UPDATE дозволяє змінювати існуючі дані. DELETE дозволяє видаляти записи. Ці чотири привілеї можуть надаватися незалежно дозволяючи налаштувати точний рівень доступу відповідно до потреб.

Надання привілеїв виконується командою GRANT яка вказує які привілеї надаються на які об'єкти якому користувачу. Синтаксис дозволяє надати кілька привілеїв в одній команді використовуючи ALL для надання всіх можливих привілеїв на даному рівні. Зірочка використовується як підстановочний знак для всіх баз даних або всіх таблиць. Команда WITH GRANT OPTION дозволяє користувачу передавати свої привілеї іншим що створює делегування повноважень проте потребує обережності.

Відкликання привілеїв командою REVOKE видаляє раніше надані права. Важливо розуміти що REVOKE видаляє лише безпосередньо надані привілеї але не впливає на привілеї отримані опосередковано через інші механізми. Після відкликання привілеїв існуючі сесії користувача можуть продовжувати використовувати старі привілеї поки не буде виконано FLUSH PRIVILEGES або поки сесія не завершиться і не почнеться нова.

Перегляд привілеїв критично важливий для аудиту безпеки та розуміння поточного стану дозволів. SHOW GRANTS виводить всі привілеї для вказаного користувача в форматі команд GRANT які можна виконати для відтворення цих привілеїв. Це корисно для документування конфігурації безпеки створення резервних копій налаштувань та діагностики проблем з доступом. Регулярний

перегляд та аудит привілеїв допомагає виявити надлишкові дозволи які слід відкликати.

Ролі є механізмом групування привілеїв який спрощує управління в системах з багатьма користувачами. Замість надання привілеїв кожному користувачу окремо створюється роль якій надаються необхідні привілеї а потім роль призначається користувачам. Зміна привілеїв ролі автоматично впливає на всіх користувачів які мають цю роль. MySQL підтримує ролі починаючи з версії вісім нуль що значно полегшує адміністрування в великих системах.

Типові ролі можуть включати readonly з привілеєм SELECT на всі таблиці для користувачів які тільки читають дані, dataentry з SELECT INSERT UPDATE для співробітників які вводять та оновлюють дані, analyst з більш широкими правами читання включно з доступом до чутливих даних, developer з правами на зміну схеми в середовищі розробки, dba з повними адміністративними правами. Організація доступу через ролі а не прямі привілеї значно спрощує управління та знижує ймовірність помилок конфігурації.

Аудит доступу та привілеїв повинен бути регулярною практикою для забезпечення що користувачі мають лише ті права які їм необхідні. Це включає перегляд списку всіх користувачів перевірку їх привілеїв ідентифікацію застарілих облікових записів та надлишкових дозволів. Автоматизація перевірок через скрипти може допомогти виявити порушення політик безпеки такі як користувачі з глобальними привілеями або паролі які не змінювалися тривалий час.

Документування політик безпеки та процедур управління користувачами є критично важливим для підтримки узгодженості особливо в командах де кілька адміністраторів управляють системою. Документація повинна описувати стандартні ролі та їх призначення процес створення нових користувачів вимоги до паролів процедури надання та відкликання привілеїв та графік періодичних аудитів. Це забезпечує що всі адміністратори дотримуються однакових практик та нові члени команди можуть швидко зрозуміти поточну конфігурацію безпеки.

ТЕМА 12. СУЧАСНІ ТЕНДЕНЦІЇ РОЗВИТКУ БАЗ ДАНИХ. СТВОРЕННЯ ФОРМ ДЛЯ РОБОТИ З БАЗОЮ ДАНИХ В MYSQL

1. Пост реляційні, об’єктно-орієнтовані та XML бази даних.
2. Технології інтелектуальної обробки даних.
3. Методи та засоби багатовимірного статистичного аналізу даних.
4. Технології для розробка частин програми на стороні сервера.
5. Технології для розробка частин програми на стороні клієнта

1. ПОСТ РЕЛЯЦІЙНІ, ОБ’ЄКТНО-ОРІЄНТОВАНІ ТА XML БАЗИ ДАНИХ.

Еволюція систем управління базами даних відображає зміни в потребах сучасних додатків та нові підходи до організації і обробки інформації. Традиційна реляційна модель, незважаючи на свою елегантність та математичну обґрунтованість, виявилася недостатньо гнучкою для деяких сучасних застосувань, що призвело до появи альтернативних підходів та розширень класичної реляційної моделі. Розуміння цих альтернатив допомагає обрати найбільш підходящу технологію для конкретних вимог проекту.

Пост реляційні бази даних представляють еволюцію реляційної моделі через додавання нових можливостей при збереженні основних принципів. Ці системи розширюють традиційний SQL підтримкою складних типів даних, об'єктних можливостей, процедурних розширень та інших функцій що виходять за межі класичної реляційної алгебри. Сучасні СУБД такі як PostgreSQL Oracle та сам MySQL в новіших версіях включають багато пост реляційних можливостей, що робить кордон між реляційними та пост реляційними системами все більш розмитим.

Підтримка складних типів даних дозволяє зберігати в базі даних не лише прості атомарні значення але й структуровані об'єкти масиви множини та користувацькі типи. Наприклад замість розбиття адреси на окремі поля вулиця місто індекс можна визначити складний тип Address і зберігати адресу як єдине значення з внутрішньою структурою. Це спрощує моделювання предметної області наближаючи структуру даних в базі до природного уявлення в коді додатку.

Масиви та колекції дозволяють зберігати множинні значення в одному полі порушуючи першу нормальну форму реляційної моделі але часто спрощуючи роботу з даними. Наприклад замість створення окремої таблиці для номерів телефонів контакту можна зберігати масив телефонів безпосередньо в записі контакту. Це зменшує кількість JOIN операцій та може поліпшити продуктивність для певних шаблонів доступу до даних хоча й ускладнює пошук та індексування окремих елементів масиву.

Підтримка JSON стала важливою можливістю сучасних СУБД що дозволяє зберігати напівструктуровані дані без необхідності визначати фіксовану схему. MySQL починаючи з версії п'ять сім підтримує нативний тип JSON з можливістю індексування окремих полів та виконання запитів всередині JSON документів.

Це поєднує гнучкість документо-орієнтованих баз даних з потужністю SQL та транзакційними гарантіями реляційних систем.

Об'єктно-орієнтовані бази даних виникли з потреби зберігати складні об'єкти з ієрархією успадкування інкапсуляцією та поліморфізмом безпосередньо в базі даних без необхідності відображення об'єктної моделі на реляційну. Така трансформація відома як object-relational impedance mismatch є джерелом значної складності в традиційних додатках і мотивувала розробку альтернативних підходів до збереження даних.

Об'єктні СУБД зберігають об'єкти з їх методами та станом безпосередньо підтримуючи концепції об'єктно-орієнтованого програмування такі як класи успадкування інкапсуляція та поліморфізм. Об'єкти можуть містити посилання на інші об'єкти формуючи складні графи замість простих таблиць з зовнішніми ключами. Мова запитів зазвичай розширює об'єктно-орієнтовану парадигму дозволяючи фільтрувати об'єкти за їх властивостями та викликати методи в запитах.

Переваги об'єктних баз даних включають природне відображення об'єктної моделі додатку без необхідності ORM фреймворків, підтримку складних зв'язків між об'єктами, збереження бізнес-логіки разом з даними через методи об'єктів. Проте об'єктні бази даних не отримали широкого поширення через відсутність стандартизації, меншу зрілість технології порівняно з реляційними системами, складність оптимізації запитів та обмежену підтримку інструментів та екосистеми.

Об'єктно-реляційні системи представляють компроміс що намагається поєднати переваги обох підходів. Вони зберігають основну реляційну модель та SQL але додають об'єктні можливості такі як користувацькі типи даних успадкування типів методи та складні типи. PostgreSQL є яскравим прикладом об'єктно-реляційної СУБД що підтримує визначення нових типів даних з їх операторами функціями та методами індексування.

XML бази даних виникли з популярності XML як універсального формату обміну даними між системами. Нативні XML бази даних зберігають дані в форматі XML підтримуючи повну структуру документа включно з порядком елементів атрибутами коментарями та інструкціями обробки. Мова запитів XQuery дозволяє навігувати та маніпулювати XML документами використовуючи шляхи XPath та функціональні конструкції.

Реляційні СУБД також додали підтримку XML через спеціальні типи даних та функції. MySQL підтримує зберігання XML в текстових полях та надає функції для парсингу та маніпуляції XML хоча без нативного типу XML. Більш

розвинені системи як Oracle та SQL Server мають повноцінні XML типи з можливістю валідації за схемою індексування та виконання XQuery запитів безпосередньо в SQL.

Гібридні бази даних що поєднують кілька моделей даних стають все більш поширеними. Сучасні СУБД підтримують одночасно реляційні таблиці JSON документи графові структури просторові дані та повнотекстовий пошук дозволяючи використовувати найбільш підходящу модель для кожної частини додатку. Це явище іноді називають multi-model databases і воно відображає розуміння що універсального рішення для всіх типів даних не існує.

NoSQL бази даних виникли як альтернатива реляційній моделі для специфічних сценаріїв де традиційні СУБД виявилися недостатньо ефективними. Документо-орієнтовані бази як MongoDB зберігають дані у вигляді JSON або BSON документів без фіксованої схеми що забезпечує гнучкість для еволюції структури даних. Колонкові сховища як Cassandra оптимізовані для записування великих обсягів даних та аналітичних запитів що читають небагато стовпців з багатьох рядків.

Графові бази даних як Neo4j спеціалізуються на зберіганні та обробці даних з багатьма складними зв'язками де запити часто потребують проходження графа на багато рівнів вглиб. Традиційні реляційні бази погано справляються з такими запитами оскільки кожен крок проходження потребує JOIN операції проте графові бази оптимізовані саме для такого типу навігації. Застосування включають соціальні мережі системи рекомендацій аналіз шахрайства та управління знаннями.

Key-value сховища є найпростішим типом NoSQL баз що зберігають дані як набір пар ключ-значення. Вони надзвичайно швидкі для простих операцій читання та запису за ключем і часто використовуються для кешування сесій та інших сценаріїв де потрібна висока швидкість та простота. Redis та Memcached є популярними прикладами key-value сховищ що підтримують також додаткові структури даних як списки множини та хеш-таблиці.

Вибір між різними типами баз даних повинен базуватися на конкретних вимогах додатку. Реляційні бази залишаються оптимальним вибором для транзакційних систем з складними зв'язками та потребою в строгій узгодженості даних. NoSQL системи краще підходять для дуже великих розподілених систем де важливіша доступність та партиціювання ніж строга узгодженість. Гібридні підходи що використовують кілька типів баз даних для різних частин системи стають все більш поширеними в сучасній архітектурі мікросервісів.

2. ТЕХНОЛОГІЇ ІНТЕЛЕКТУАЛЬНОЇ ОБРОБКИ ДАНИХ.

Інтелектуальна обробка даних або Data Mining являє собою процес виявлення раніше невідомих нетривіальних практично корисних та доступних для інтерпретації знань у великих масивах даних. На відміну від традиційного аналізу де людина формулює гіпотезу і перевіряє її на даних, Data Mining дозволяє системі автоматично виявляти закономірності та шаблони про існування яких аналітик міг не здогадуватися. Це особливо цінно в епоху великих даних коли обсяги інформації перевищують можливості людини для ручного аналізу.

Основні завдання інтелектуальної обробки даних включають класифікацію кластеризацію пошук асоціативних правил регресійний аналіз виявлення аномалій та прогнозування. Кожне з цих завдань вирішує специфічні бізнес-проблеми та використовує відповідні алгоритми та методи. Успішне застосування Data Mining потребує не лише технічних знань алгоритмів але й глибокого розуміння предметної області для правильної інтерпретації результатів.

Класифікація полягає в побудові моделі що дозволяє віднести нові об'єкти до одного з заздалегідь визначених класів на основі їх характеристик. Модель навчається на історичних даних де класи об'єктів відомі і потім застосовується для передбачення класів нових об'єктів. Типові застосування включають визначення кредитоспроможності клієнтів розпізнавання спаму виявлення шахрайських транзакцій медичну діагностику. Алгоритми класифікації включають дерева рішень нейронні мережі метод опорних векторів наївний баєсовський класифікатор.

Кластеризація або сегментація групує об'єкти в кластери так щоб об'єкти всередині кластера були схожі між собою а об'єкти з різних кластерів відрізнялися. На відміну від класифікації при кластеризації класи заздалегідь не відомі і система сама виявляє природні групування в даних. Це корисно для сегментації клієнтів виявлення типових шаблонів поведінки організації асортименту товарів. Популярні алгоритми включають k-means ієрархічну кластеризацію DBSCAN та алгоритми на основі щільності.

Пошук асоціативних правил виявляє закономірності спільної появи подій або об'єктів у великих базах даних. Класичним прикладом є аналіз ринкового кошика що виявляє які товари часто купуються разом дозволяючи оптимізувати розміщення товарів у магазині та формувати персоналізовані рекомендації. Алгоритм Apriori та його модифікації ефективно знаходять часті набори

елементів навіть у дуже великих датасетах через використання властивості що підмножини частих наборів також є частими.

Регресійний аналіз будує математичну модель залежності однієї змінної від інших дозволяючи передбачати числові значення на основі вхідних параметрів. Лінійна регресія моделює лінійну залежність тоді як поліноміальна та нелінійна регресія можуть вловлювати складніші відношення. Застосування включають прогнозування продажів оцінку вартості нерухомості аналіз факторів що впливають на результат. Сучасні підходи використовують ансамблі моделей та методи машинного навчання для підвищення точності прогнозів.

Виявлення аномалій або outlier detection ідентифікує об'єкти події або спостереження що суттєво відхиляються від очікуваної норми. Аномалії можуть вказувати на помилки в даних шахрайство несправності обладнання або інші важливі події що потребують уваги. Методи виявлення аномалій включають статистичні підходи на основі розподілів методи на базі відстаней кластеризацію з ідентифікацією віддалених точок та нейромережеві автоенкодери.

Прогнозування часових рядів аналізує дані зібрані послідовно в часі для виявлення трендів сезонності циклічності та випадкових коливань з метою передбачення майбутніх значень. Це критично важливо для планування запасів прогнозування попиту фінансового планування аналізу метрик систем. Методи включають традиційні статистичні підходи як ARIMA експоненційне згладжування та сучасні підходи на основі рекурентних нейронних мереж та LSTM.

Процес Knowledge Discovery in Databases складається з кількох етапів що формують методологію перетворення сирих даних на корисні знання. Вибір та інтеграція даних включає ідентифікацію релевантних джерел даних їх об'єднання та створення єдиного датасету для аналізу. Попередня обробка та очищення даних усуває помилки дублікати пропущені значення та неузгодженості що критично важливо оскільки якість результатів безпосередньо залежить від якості вхідних даних.

Трансформація даних перетворює дані в формат придатний для аналізу через нормалізацію агрегацію створення похідних атрибутів та зменшення розмірності. Вибір методу Data Mining залежить від типу завдання характеристик даних та бізнес-цілей. Інтерпретація та оцінка результатів потребує залучення експертів предметної області для перевірки чи виявлені закономірності мають сенс та чи є вони дійсно корисними а не артефактами даних.

Інструменти Data Mining включають спеціалізоване програмне забезпечення та бібліотеки що реалізують різні алгоритми. RapidMiner та KNIME надають графічні інтерфейси для побудови аналітичних пайплайнів без програмування. Python з бібліотеками scikit-learn pandas та TensorFlow став стандартом де-факто для машинного навчання та Data Mining. R залишається популярним в академічному середовищі та для статистичного аналізу. Комерційні рішення як SAS Enterprise Miner та IBM SPSS Modeler надають повнофункціональні середовища для корпоративного Data Mining.

Інтеграція Data Mining з базами даних дозволяє виконувати аналіз безпосередньо на даних в СУБД без необхідності експортування їх в окремі інструменти. Деякі СУБД включають вбудовані можливості аналітики як Oracle Data Mining що надає PL/SQL API для побудови та застосування моделей машинного навчання. Розширення PostgreSQL як MADlib реалізують алгоритми машинного навчання що виконуються паралельно на даних в базі використовуючи SQL інтерфейс.

Етичні аспекти Data Mining включають питання конфіденційності персональних даних можливість дискримінації через упередження в даних або алгоритмах та прозорість автоматизованих рішень. Регуляції як GDPR встановлюють вимоги до обробки персональних даних включно з правом на пояснення автоматизованих рішень. Відповідальне використання Data Mining потребує балансування між корисністю аналітики та захистом прав та інтересів людей чиї дані аналізуються.

3. МЕТОДИ ТА ЗАСОБИ БАГАТОВИМІРНОГО СТАТИСТИЧНОГО АНАЛІЗУ ДАНИХ.

Багатовимірний аналіз даних працює з даними що містять багато змінних або вимірів одночасно дозволяючи виявляти складні взаємозв'язки та структури що неможливо побачити при аналізі окремих змінних. На відміну від одновимірного аналізу що розглядає розподіл однієї змінної або двовимірного що досліджує зв'язок між парами змінних багатовимірні методи аналізують всі змінні разом враховуючи їх спільну варіацію та взаємодії.

OLAP або Online Analytical Processing являє собою підхід до організації та аналізу багатовимірних даних що дозволяє швидко виконувати складні аналітичні запити. Дані організовані у вигляді багатовимірного кубу де виміри представляють категоріальні атрибути як час продукт географія клієнт а міри або факти є числовими показниками як обсяг продажів прибуток кількість. Така

організація дозволяє інтуїтивно досліджувати дані змінюючи перспективу та рівень деталізації.

Операції над OLAP кубом включають slice що вибирає один шар куба фіксуючи значення одного виміру, dice що вибирає підкуб обмежуючи кілька вимірів, drill-down що переходить до більш детального рівня виміру наприклад від року до кварталу до місяця, drill-up що агрегує дані на вищий рівень виміру, та pivot що обертає куб змінюючи орієнтацію вимірів для іншої перспективи перегляду даних.

Архітектура OLAP систем може бути реалізована різними способами. MOLAP або Multidimensional OLAP зберігає дані безпосередньо в багатовимірних структурах забезпечуючи найшвидший доступ проте обмежений масштабованістю та потребує попереднього обчислення агрегатів. ROLAP або Relational OLAP зберігає дані в реляційних таблицях у формі зірки або сніжинки і виконує аналітичні запити через SQL забезпечуючи кращу масштабованість проте повільніший доступ. HOLAP поєднує обидва підходи зберігаючи детальні дані в реляційних таблицях а агрегати в багатовимірних структурах.

Схема зірки є найпростішим та найпоширенішим способом моделювання даних для аналітичних систем. Центральна таблиця фактів містить міри та зовнішні ключі до таблиць вимірів. Таблиці вимірів містять описові атрибути та зазвичай денормалізовані для спрощення запитів. Ця структура оптимізована для читання та агрегації даних хоча й дублює деяку інформацію порушуючи нормалізацію прийнятну для транзакційних систем але ефективну для аналітики.

Схема сніжинки нормалізує таблиці вимірів розбиваючи їх на пов'язані таблиці для усунення дублювання. Наприклад вимір Продукт може бути розбитий на таблиці Продукт Категорія Виробник кожна з яких нормалізована. Це економить простір та спрощує підтримку довідників проте ускладнює запити що потребують більше JOIN операцій і можуть працювати повільніше ніж зі схемою зірки.

Slowly Changing Dimensions або повільно змінювані виміри представляють виклик в аналітичних системах де атрибути вимірів змінюються з часом. Тип один просто перезаписує старі значення не зберігаючи історію що найпростіше але втрачає можливість аналізу історичних даних в контексті того як виміри виглядали тоді. Тип два додає нові версії записів вимірів зі строками валідності дозволяючи точно відтворити стан на будь-який момент часу проте ускладнює запити та збільшує розмір бази.

Тип три SCD додає окремі стовпці для зберігання попередніх значень обмежена кількість історичних версій що є компромісом між простотою та

глибиною історії. Гібридні підходи комбінують різні типи для різних атрибутів залежно від бізнес-вимог до історичності. Вибір стратегії SCD має значний вплив на складність ETL процесів розмір сховища та можливості аналізу.

Data Warehouse або сховище даних є центральним репозиторієм інтегрованих даних з різних джерел оптимізованим для аналітичних запитів та звітності. На відміну від операційних баз даних що оптимізовані для транзакційної обробки сховища даних організовані для складних запитів що сканують великі обсяги історичних даних. Вони зазвичай оновлюються періодично через ETL процеси а не в реальному часі і зберігають історію змін для аналізу трендів.

ETL процеси Extract Transform Load вилучають дані з різних джерел перетворюють їх до єдиного формату очищають від помилок та неузгодженостей інтегрують дані з різних систем та завантажують в сховище даних. Етап Extract підключається до різних джерел даних що можуть включати реляційні бази даних файли API зовнішніх систем та витягує необхідні дані. Етап Transform виконує очищення валідацію стандартизацію агрегацію та інші перетворення для підготовки даних до аналізу.

Етап Load завантажує підготовлені дані в цільове сховище використовуючи стратегії повного завантаження для початкового наповнення або інкрементального завантаження для регулярних оновлень тільки змінених даних. Сучасні підходи ELT Extract Load Transform завантажують дані в сховище з мінімальними перетвореннями і виконують складні трансформації безпосередньо в цільовій системі використовуючи її потужності обчислення що може бути ефективніше для великих обсягів даних.

Data Mart є підмножиною або секцією сховища даних сфокусованою на конкретній бізнес-області відділі або групі користувачів. Замість створення одного величезного монолітного сховища організації часто будують кілька Data Mart для різних потреб як продажі фінанси маркетинг кожен з власною оптимізованою структурою та набором даних. Це спрощує управління та підвищує продуктивність оскільки користувачі працюють лише з релевантними для них даними.

Візуалізація багатовимірних даних є критично важливою для інтерпретації результатів аналізу. Інструменти бізнес-аналітики як Tableau Power BI QlikView надають інтерактивні дашборди що дозволяють користувачам досліджувати дані через візуальні представлення без написання SQL запитів. Теплові карти деревоподібні діаграми паралельні координати та інші спеціалізовані візуалізації

допомагають виявляти шаблони та аномалії в багатовимірних даних що були б не очевидні в таблицях чисел.

Аналіз головних компонент PCA є методом зменшення розмірності що трансформує множину можливо корельованих змінних в меншу множину некорельованих змінних названих головними компонентами. Перша головна компонента захоплює максимум варіації в даних друга максимум з залишкової варіації і так далі. Це дозволяє візуалізувати багатовимірні дані в двох або трьох вимірах зберігаючи більшість інформації та виявити латентні фактори що впливають на дані.

Факторний аналіз схожий на PCA але фокусується на виявленні прихованих факторів що пояснюють кореляції між спостережуваними змінними. Він використовується для розуміння структури даних наприклад які групи питань в опитуванні вимірюють однаковий базовий конструкт або які економічні показники визначаються спільними факторами. Результати факторного аналізу допомагають спростити складні набори даних та побудувати узагальнені індекси або шкали.

Кластерний аналіз в багатовимірному просторі групує об'єкти на основі їх схожості за всіма змінними одночасно. Методи ієрархічної кластеризації будують дерево групувань що показує як об'єкти та кластери об'єднуються на різних рівнях схожості. Методи розділення як k-means поділяють дані на задану кількість кластерів мінімізуючи варіацію всередині кластерів. Результати кластеризації використовуються для сегментації клієнтів класифікації документів виявлення типових шаблонів та інших задач групування.

4. ТЕХНОЛОГІЇ ДЛЯ РОЗРОБКА ЧАСТИН ПРОГРАМИ НА СТОРОНІ СЕРВЕРА.

Серверна частина веб-додатків відповідає за бізнес-логіку обробку даних взаємодію з базою даних автентифікацію та авторизацію генерацію динамічного контенту та інші операції що повинні виконуватися в захищеному контрольованому середовищі. Вибір технологій для серверної розробки впливає на продуктивність масштабованість безпеку та швидкість розробки всього додатку. Розуміння різних підходів та їх компромісів допомагає приймати обґрунтовані архітектурні рішення.

PHP залишається однією з найпопулярніших мов для веб-розробки особливо для проектів що працюють з MySQL завдяки їх тісній інтеграції та широкому поширенню на більшості хостингів. PHP виконується на сервері генеруючи HTML який надсилається клієнту і має низький поріг входу що

робить його доступним для початківців. Сучасний PHP з версії сім та вище значно покращився в плані продуктивності типізації та об'єктно-орієнтованих можливостей стаючи зрілою платформою для серйозних проектів.

Підключення до MySQL з PHP виконується через різні розширення. Застаріле mysql розширення більше не підтримується і не повинно використовуватися. Розширення mysqli надає процедурний та об'єктно-орієнтований інтерфейс для роботи з MySQL підтримуючи підготовлені запити транзакції та інші сучасні можливості. PDO або PHP Data Objects надає універсальний об'єктно-орієнтований інтерфейс для роботи з різними базами даних що полегшує міграцію між СУБД і є рекомендованим підходом для нових проектів.

Підготовлені запити або prepared statements є критично важливими для безпеки оскільки запобігають SQL-ін'єкціям через відокремлення структури запиту від даних. Запит визначається з плейсхолдерами замість прямої підстановки значень а потім параметри прив'язуються окремо і автоматично екрануються. Це не тільки безпечніше але й може бути ефективніше для запитів що виконуються багато разів оскільки парсинг та оптимізація виконуються один раз а потім план запиту використовується повторно з різними параметрами.

Фреймворки PHP як Laravel Symfony CodeIgniter надають структуру та набір компонентів для швидкої розробки веб-додатків. Вони включають ORM для роботи з базами даних без написання SQL систему маршрутизації шаблонізатори для генерації HTML міграції для управління схемою бази даних та багато інших можливостей. Використання фреймворку прискорює розробку забезпечує дотримання best practices та полегшує підтримку коду в командах розробників.

Python з фреймворками Django та Flask став популярним вибором для веб-розробки завдяки чистому синтаксису потужним бібліотекам та сильній підтримці науки про дані та машинного навчання. Django є повнофункціональним фреймворком що включає ORM адміністративний інтерфейс систему аутентифікації та багато інших компонентів готових до використання. Flask є мікрофреймворком що надає мінімальну основу і дозволяє розробникам вибирати та інтегрувати лише потрібні компоненти забезпечуючи більшу гнучкість.

Підключення до MySQL з Python виконується через бібліотеку mysql-connector-python або PyMySQL що надають DB-API сумісний інтерфейс. SQLAlchemy є потужною ORM бібліотекою що дозволяє працювати з базою даних через об'єкти Python автоматично генеруючи SQL запити і підтримуючи

різні СУБД. Django ORM інтегрована в фреймворк і надає декларативний спосіб визначення моделей даних та виконання запитів через зручний Python API.

Node.js дозволяє використовувати JavaScript на сервері використовуючи подієвий неблокуючий ввід-вивід що робить його особливо ефективним для додатків з великою кількістю одночасних з'єднань та інтенсивним вводом-виводом. Фреймворк Express є мінімалістичним та гнучким забезпечуючи основу для веб-додатків та API. Пакет mysql2 надає швидкий та безпечний спосіб підключення до MySQL з підтримкою promises та async/await для зручної роботи з асинхронним кодом.

ORM або Object-Relational Mapping бібліотеки як Sequelize для Node.js Hibernate для Java Entity Framework для .NET дозволяють працювати з базою даних через об'єкти мови програмування без написання SQL. ORM автоматично перетворює операції над об'єктами в SQL запити і навпаки відображає результати запитів в об'єкти. Це спрощує розробку робить код більш переносимим між різними СУБД і знижує ймовірність SQL-ін'єкцій проте може створювати неоптимальні запити і потребує додаткового вивчення API ORM.

RESTful API дизайн став стандартом для побудови веб-сервісів що надають доступ до даних та функціональності через HTTP. REST використовує стандартні HTTP методи GET для читання POST для створення PUT або PATCH для оновлення DELETE для видалення ресурсів які ідентифікуються URL. Клієнти та сервери взаємодіють через обмін JSON або XML представленнями ресурсів що робить API простими для розуміння та використання з будь-яких платформ та мов програмування.

GraphQL є альтернативним підходом до API що дозволяє клієнтам точно вказувати які дані їм потрібні замість отримання фіксованих структур з REST ендпоінтів. Клієнт надсилає запит що описує бажану структуру даних включно з вкладеними зв'язками і сервер повертає точно ці дані без надлишку або недостачі. Це зменшує кількість запитів та обсяг переданих даних особливо для складних інтерфейсів проте потребує додаткової інфраструктури на сервері для обробки GraphQL запитів.

Автентифікація та авторизація є критичними аспектами серверної розробки. Сесії зберігають стан користувача між запитами традиційно через cookies з ідентифікатором сесії і серверним сховищем даних сесії. JWT або JSON Web Tokens є альтернативним підходом де токен що містить закодовану інформацію про користувача підписується сервером і надсилається клієнту який включає його в наступні запити. Сервер перевіряє підпис токену без необхідності

зберігати стан сесії що спрощує масштабування проте потребує уваги до безпеки та управління термінами дії токенів.

OAuth та OpenID Connect є стандартними протоколами для делегованої автентифікації та авторизації що дозволяють користувачам входити через сторонні сервіси як Google Facebook GitHub без створення окремих облікових записів. Це покращує користувацький досвід та безпеку оскільки додаток не зберігає паролі користувачів проте вимагає реалізації складного потоку взаємодії з провайдерами ідентичності та обробки різних сценаріїв помилок.

Кешування на серверній стороні значно покращує продуктивність зменшуючи навантаження на базу даних та час відповіді. Кешування результатів запитів в пам'яті через Redis або Memcached дозволяє миттєво повертати часто запитувані дані без звернення до бази. Кешування згенерованих сторінок або фрагментів HTML економить процесорний час на повторну генерацію однакового контенту. Важливо правильно налаштувати інвалідацію кешу щоб користувачі бачили актуальні дані коли вони змінюються.

Черги повідомлень та асинхронна обробка дозволяють виконувати довготривалі операції як відправка email генерація звітів обробка зображень у фоновому режимі не блокуючи відповідь користувачу. RabbitMQ Redis Celery є популярними рішеннями для організації черг задач. Запит від користувача швидко поміщає задачу в чергу і повертає відповідь тоді як воркери обробляють задачі незалежно і можуть масштабуватися горизонтально для обробки більшого навантаження.

Мікросервісна архітектура розбиває монолітний додаток на набір невеликих незалежних сервісів кожен з яких відповідає за певну бізнес-функцію і може розроблятися розгортатися та масштабуватися незалежно. Сервіси взаємодіють через легкі протоколи зазвичай HTTP REST або повідомлення. Це дозволяє командам працювати паралельно над різними сервісами вибирати оптимальні технології для кожного сервісу та масштабувати лише ті компоненти які потребують більше ресурсів проте додає складності в організації комунікації моніторингу та забезпеченні узгодженості даних.

5. ТЕХНОЛОГІЇ ДЛЯ РОЗРОБКА ЧАСТИН ПРОГРАМИ НА СТОРОНІ КЛІЄНТА.

Клієнтська частина сучасних веб-додатків відповідає за інтерфейс користувача презентацію даних та взаємодію з користувачем виконуючись безпосередньо в браузері. Еволюція клієнтських технологій перетворила веб-додатки з простих статичних сторінок в складні інтерактивні застосунки що

конкурують з нативними десктопними програмами за функціональністю та користувацьким досвідом. Розуміння сучасного стеку фронтенд технологій є критично важливим для створення ефективних та привабливих веб-інтерфейсів.

HTML п'ять залишається основою структури веб-сторінок надаючи семантичні елементи для позначення різних типів контенту. Семантичні теги як header nav main article section footer покращують доступність дозволяючи скрін-рідерам та пошуковим системам краще розуміти структуру та значення контенту. Нові елементи форм як input типу email date number включають вбудовану валідацію та спеціалізовані клавіатури на мобільних пристроях покращуючи користувацький досвід без необхідності JavaScript.

CSS три та новіші специфікації надають потужні можливості для створення привабливих адаптивних дизайнів. Flexbox та Grid дозволяють створювати складні макети що автоматично адаптуються до різних розмірів екрану без використання float або позиціювання. Media queries реалізують адаптивний дизайн змінюючи стилі залежно від характеристик пристрою як ширина екрану орієнтація роздільна здатність. CSS змінні або custom properties дозволяють визначати значення що використовуються повторно та можуть динамічно змінюватися через JavaScript.

Препроцесори CSS як Sass Less Stylus розширюють можливості CSS додаючи змінні вкладеність міксини функції та інші програмні конструкції. Це робить великі таблиці стилів більш організованими та підтримуваними через можливість повторного використання коду та модульної організації. Препроцесори компілюються в звичайний CSS перед розгортанням тому не вимагають підтримки браузерами та не впливають на продуктивність сайту.

JavaScript є мовою програмування клієнтської частини що дозволяє створювати інтерактивні елементи обробляти події користувача маніпулювати DOM виконувати асинхронні запити до сервера та реалізовувати складну логіку в браузері. Сучасний JavaScript або ECMAScript з версії ES6 додав безліч покращень як класи стрілкові функції деструктуризація promises async await модулі що зробили мову більш виразною та зручною для розробки великих додатків.

DOM або Document Object Model API дозволяє JavaScript маніпулювати структурою стилями та контентом веб-сторінки. Методи як getElementById querySelector дозволяють знаходити елементи методи createElement appendChild видаляти та додавати елементи властивості innerHTML textContent змінювати контент властивості style className змінювати стилі. Обробка подій через

addEventListener дозволяє реагувати на дії користувача як кліки введення тексту наведення миші прокрутка.

AJAX або Asynchronous JavaScript and XML дозволяє виконувати HTTP запити до сервера асинхронно без перезавантаження сторінки що є основою для створення динамічних Single Page Applications. Традиційний XMLHttpRequest API замінюється сучасним Fetch API що надає більш зручний інтерфейс на основі promises. Запити зазвичай обмінюються JSON даними які легко парсяться JavaScript і дозволяють оновлювати частини сторінки на основі отриманих даних.

Фреймворки та бібліотеки JavaScript значно спрощують розробку складних клієнтських додатків надаючи структуру компоненти та інструменти. React є бібліотекою від Facebook для побудови користувацьких інтерфейсів через компоненти що описують як UI повинен виглядати залежно від стану. Віртуальний DOM дозволяє ефективно оновлювати тільки змінені частини сторінки. Однонаправлений потік даних та концепція компонентів спрощують розуміння та відлагодження складних інтерфейсів.

Vue.js є прогресивним фреймворком що може використовуватися як проста бібліотека для додавання інтерактивності до окремих частин сторінки або як повноцінний фреймворк для побудови SPA. Vue має низький поріг входу завдяки простому синтаксису шаблонів та поступовому впровадженню складних концепцій. Реактивна система автоматично оновлює DOM коли змінюються дані що спрощує синхронізацію стану та відображення.

Angular є повнофункціональним фреймворком від Google що включає все необхідне для побудови великих корпоративних додатків включно з двонаправленим зв'язуванням даних dependency injection маршрутизацією HTTP клієнтом та інструментами розробки. Angular використовує TypeScript що додає статичну типізацію та покращує підтримку коду в великих командах. Архітектура на основі компонентів та модулів забезпечує чітку організацію коду.

TypeScript є надмножиною JavaScript що додає статичну типізацію інтерфейси класи та інші можливості що покращують якість коду та підтримку особливо для великих проектів. Компілятор TypeScript перевіряє типи на етапі розробки виявляючи багато помилок до запуску коду і надає кращу підтримку автодоповнення та рефакторингу в IDE. TypeScript компілюється в звичайний JavaScript що виконується в браузерах без необхідності спеціальної підтримки.

Бандлери модулів як Webpack Rollup Parcel об'єднують багато файлів JavaScript CSS зображень в оптимізовані бандли для продакшн розгортання. Вони виконують трансформації коду через лоадери та плагіни як компіляція

TypeScript транспіляція сучасного JavaScript для старих браузерів мінімізація коду оптимізація зображень. Code splitting дозволяє розбити додаток на частини що завантажуються по потребі зменшуючи початковий час завантаження.

State management бібліотеки як Redux MobX Vuex допомагають управляти складним станом додатку особливо коли багато компонентів потребують доступу до однакових даних. Централізоване сховище стану з чітко визначеними правилами його зміни робить потік даних передбачуваним та полегшує відлагодження. Redux використовує однонаправлений потік даних та іммутабельні оновлення стану через чисті функції редюсери що забезпечує передбачуваність та можливість time-travel debugging.

Прогресивні веб-додатки або PWA поєднують найкраще з веб та нативних додатків надаючи можливість працювати офлайн push-повідомлення встановлення на домашній екран та швидке завантаження. Service Workers є скриптами що виконуються у фоні окремо від веб-сторінки і можуть перехоплювати мережеві запити кешувати ресурси та синхронізувати дані у фоні. Web App Manifest описує додаток для браузера включно з іконками кольорами екраном запуску що дозволяє встановити PWA як звичайний додаток.

WebAssembly або WASM є низькорівневим байт-кодом що виконується в браузері з майже нативною швидкістю дозволяючи портувати код написаний на C C++ Rust та інших мовах в веб. Це відкриває можливості для створення продуктивних веб-додатків як відеоредактори ігри CAD системи наукові обчислення що раніше були можливі лише як нативні програми. WebAssembly доповнює JavaScript і обидві технології можуть взаємодіяти в одному додатку використовуючи сильні сторони кожної.

Тестування клієнтського коду включає юніт-тести для перевірки окремих функцій та компонентів інтеграційні тести для перевірки взаємодії компонентів та end-to-end тести що імітують дії користувача в реальному браузері. Jest є популярним фреймворком для тестування JavaScript та React додатків. Cypress та Selenium дозволяють писати автоматизовані тести що виконують дії в браузері та перевіряють результати симулюючи реальних користувачів.

Доступність або a11y забезпечує що веб-додатки можуть використовуватися людьми з різними обмеженнями включно з порушеннями зору слуху моторики когнітивними особливостями. Семантичний HTML ARIA атрибути клавіатурна навігація достатній контраст кольорів та підтримка скрін-рідерів є критично важливими для інклюзивного дизайну. Автоматизовані інструменти як Lighthouse axe можуть виявити багато проблем доступності проте

ручне тестування з асистивними технологіями залишається необхідним для забезпечення повної доступності.

Продуктивність фронтенду впливає на користувацький досвід та показники бізнесу. Оптимізації включають мінімізацію та стиснення файлів ліниве завантаження зображень та компонентів використання CDN для статичних ресурсів кешування критичний рендеринг шлях зменшення кількості запитів використання sprite для іконок оптимізацію шрифтів. Інструменти як Lighthouse WebPageTest дозволяють вимірювати продуктивність та ідентифікувати вузькі місця для поліпшення.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Берко А. Ю., Верес О. М., Пасічник В. В. Системи баз даних та знань. Кн. 1. Організація баз даних та знань : навч. посібник. Львів : Магнолія 2006, 2024. 456 с.
2. Берко А.Ю., Верес О.М., Пасічник В.В. Моделі баз даних та знань : підручник. Львів : ПП "Магнолія 2006", 2025. 463 с. (до замовлення, ціна -990 грн).
3. Верес О. М., Берко А. Ю., Пасічник В. В. Технології баз даних та знань : підручник. Львів : Магнолія 2006, 2024. 636 с.
4. Власюк А. Г. Основи використання SQL у серверних системах : навч. посібник. Київ: Компринт, 2017. 125 с.
5. Гайдаржи В. І., Ізварін І. В. Бази даних в інформаційних системах : підруч. Київ. 2018. 418 с.
6. Демиденко М. А. Введення в сучасні бази даних : навч. посіб. / НТУ «Дніпровська політехніка». Дніпро : Дніпровська політехніка, 2020. 38 с. URL: https://ir.nmu.org.ua/server/api/core/bitstreams/7d1d93bf-305f-4d4c-ba18-aa4eb2735c87/content
7. Доценко С. І. Організація та системи керування базами даних : навч. посібник. Харків : УкрДУЗТ, 2023. 117 с. URL: https://surli.cc/wgzyzk
8. Костенко О. Б. Організація баз даних та знань : конспект лекцій (для студентів денної та заочної форм навчання першого (бакалаврського) рівня вищої освіти за спеціальністю 126 – Інформаційні системи та технології). Харків. нац. ун-т міськ. госп-ва ім. О. М. Бекетова. Харків. 2021. 92 с.
9. Мікула М. П., Коцюк Ю. А., Мікула О. М. Організація баз даних та знань : навчальний посібник для студентів спеціальності «Комп’ютерні науки». Острог : Острозька академія, 2021. 194 с.
10. Організація баз даних та знань : конспект лекцій / уклад. : О. Б. Костенко, І. О. Гавриленко ; Харків. нац. ун-т міськ. госп-ва ім. О. М. Бекетова. Харків : ХНУМГ ім. О. М. Бекетова, 2021. 92 с. URL: https://eprints.kname.edu.ua/60505/1/2020%20%D0%BF%D0%B5%D1%87.%20134_%D0%9B.pdf
11. Офіційний сайт MySQL WorkBench. URL: http://www.mysql.com/
12. Павлиш В. А. Основи інформаційних технологій і систем : підручник. Львів : Львівська політехніка. 2018. 619 с.
13. Павловський В. І., Петрашенко А. В. Бази даних та засоби управління : підручник / ; КПІ ім. Ігоря Сікорського. Київ : КПІ ім. Ігоря Сікорського, 2024. 326 с. URL: https://ela.kpi.ua/server/api/core/bitstreams/9f26675d-c42f-485e-aab5-13e6e014b560/content
14. Проектування та використання баз даних-1. Комп’ютерний практикум : навчальний посібник / КПІ ім. Ігоря Сікорського ; уклад. І. В. Сегеда. Київ : КПІ ім. Ігоря Сікорського, 2021. 49 с. URL:

https://ela.kpi.ua/server/api/core/bitstreams/a20f3199-33a5-47d1-ba26-0a2ef8b73790/content
15. Рзаєва С. Л., Харченко О. А. Бази даних : посібник. Київ : Київський національний торговельно-економічний університет, 2021. 228 с.
16. Технології баз даних : навчально-практичний посібник / уклад. А. А. Гаврилова, С. С. Погасій, Р. В. Корольов, В. С. Хвостенко, Т. С. Мілевська ; за заг. ред. С. П. Євсеєва. Харків : НТУ «ХПІ» ; Львів : «Новий Світ-2000», 2025. 222 с.

Навчальне видання

БАЗИ ДАНИХ

Методичні рекомендації

Укладачі

Тищенко Світлана Іванівна

Пархоменко Олександр Юрійович

Жебко Олександр Олегович

Формат 60х84 1/16. Ум. друк. арк. 14,31

Тираж 20 пр. Зам. №__

Надруковано у видавничому відділі

Миколаївського національного аграрного університету

54020, м. Миколаїв, вул. Георгія Гонгадзе, 9

Свідоцтво суб'єкта видавничої справи ДК № 4490 від 20.02.2013 р.