Problematika relačních databází a rozdíl mezi SQL a NoSQL databázemi

Relační databáze

Relační databáze je typ databáze, který ukládá a poskytuje přístup k datům organizovaným do předem definovaných tabulek (relací). Každá tabulka se skládá ze sloupců (atributů) a řádků (záznamů). Hlavní výhody relačních databází jsou:

1. **Strukturovanost a konzistence dat**: Data jsou uložena v tabulkách s pevně definovanými strukturami, což usnadňuje zajištění konzistence a integrity dat. To znamená, že data v tabulce musí splňovat určitá pravidla (například data v určitém sloupci musí být unikátní nebo nemohou být prázdná).

2. **Referenční integrita**: Relační databáze umožňuje definovat vztahy mezi tabulkami pomocí primárních a cizích klíčů. To zajišťuje, že vztahy mezi daty jsou platné a konzistentní, což je důležité pro udržení integrity dat.

3. **Transakční zpracování**: Relační databáze podporují transakce, které zajišťují, že více operací na databázi bude provedeno jako celek nebo vůbec. To je důležité pro zajištění integrity dat při složitých operacích, například při převodech peněz.

4. **Dotazování pomocí SQL**: Relační databáze používají jazyk SQL (Structured Query Language) pro dotazování a manipulaci s daty. SQL je standardizovaný a umožňuje složité dotazy na data.

SQL databáze vs. NoSQL databáze

SQL databáze

- **Struktura**: Data jsou ukládána do tabulek s pevnou strukturou (sloupce a řádky).
- **Schéma**: Vyžaduje předem definované schéma, které určuje strukturu dat (typy sloupců, povinná pole, apod.).
- **Transakce**: Podporuje ACID (Atomicity, Consistency, Isolation, Durability) vlastnosti, které zajišťují bezpečné a konzistentní zpracování transakcí.
- **Vhodnost**: Ideální pro aplikace, kde je důležitá konzistence a integrita dat, například bankovní systémy, účetní systémy, apod.
- **Příklady SQL databází**:
  - **MySQL**: Open-source relační databáze, široce používaná pro webové aplikace.
  - **PostgreSQL**: Pokročilá open-source relační databáze, známá pro svou robustní podporu datových typů a komplexních dotazů.
  - **Microsoft SQL Server**: Komerční relační databáze vyvinutá společností Microsoft, často používaná v podnikovém prostředí.
  - **Oracle Database**: Velmi rozšířená komerční databáze s podporou velkého množství funkcí a optimalizací pro různé typy aplikací.

NoSQL databáze

- **Struktura**: Data mohou být ukládána v různých formátech, například dokumenty (JSON, XML), klíč-hodnota páry, grafy nebo sloupce. To umožňuje větší flexibilitu v ukládání různorodých dat.
- **Schéma**: Nevyžaduje pevně definované schéma, což umožňuje dynamicky měnit strukturu dat. To je užitečné pro aplikace s rychle se měnícími datovými potřebami.
- **Transakce**: Podporuje BASE (Basically Available, Soft state, Eventual consistency) model, který je méně striktní než ACID. To může být výhodné pro aplikace, které potřebují vysokou dostupnost a škálovatelnost.
- **Vhodnost**: Vhodné pro aplikace, které vyžadují rychlé čtení a zápis velkých objemů dat, například sociální sítě, analytické aplikace, IoT systémy, apod.
- **Příklady NoSQL databází**:
  - **MongoDB**: Dokumentová databáze, která ukládá data ve formátu JSON-like dokumentů, často používaná pro flexibilní datové struktury.
  - **Cassandra**: Distribuovaná sloupcová databáze, navržená pro škálovatelnost a vysokou dostupnost bez centrálního bodu selhání.
  - **Redis**: In-memory klíč-hodnota databáze, často používaná pro caching a real-time analytické aplikace.
  - **Neo4j**: Grafová databáze, která ukládá data ve formě uzlů a hran, což umožňuje efektivní reprezentaci a dotazování na složité vztahy mezi daty.

Úvod do SQL Serveru a SQL

Úvod do SQL Serveru

SQL Server je relační databázový systém (RDBMS) vyvinutý společností Microsoft. Slouží k ukládání, správě a načítání dat, která jsou organizována do tabulek s řádky a sloupci. SQL Server podporuje širokou škálu funkcí včetně transakčního zpracování, business intelligence a analýzy dat. Je vhodný pro různé typy aplikací, od malých aplikací až po rozsáhlé enterprise systémy. SQL Server nabízí různé edice, například Express, Standard a Enterprise, které se liší funkcemi a kapacitou.

Základní informace o jazyku SQL a historie

SQL (Structured Query Language) je standardizovaný jazyk používaný pro správu relačních databází. Byl vyvinut v 70. letech minulého století v laboratořích IBM a jeho první komerční implementace byla realizována společností Oracle v roce 1979. SQL umožňuje uživatelům definovat strukturu databáze (příkazy DDL), manipulovat s daty (příkazy DML), řídit přístup k datům (příkazy DCL) a spravovat transakce (příkazy TCL). Díky své jednoduchosti a standardizaci se stal SQL nejpoužívanějším jazykem pro databázové systémy.

Standardy jazyka

SQL je definován různými standardy, které zajišťují kompatibilitu a jednotnost v používání jazyka napříč různými databázovými systémy. Mezi nejdůležitější standardy patří:

- **SQL-86**: První standard, který zavedl základní syntaxi a konstrukce.
- **SQL-92**: Přinesl významná vylepšení včetně přidání subdotazů a integritních omezení.
- **SQL:1999**: Zavedl objektově-relační rozšíření, triggery a SQLJ pro integraci s Javou.
- **SQL:2003**: Přinesl podporu pro XML data, okna a analytické funkce.
- **SQL:2011**: Přidání standardizovaných temporálních datových typů a operací.

Rozdělení příkazů

SQL příkazy lze rozdělit do několika kategorií:

- **DDL (Data Definition Language)**: Příkazy pro definici struktury databáze.
  - `CREATE`: Vytvoření nové databáze, tabulky nebo jiného objektu.
  - `ALTER`: Změna struktury existující tabulky nebo objektu.
  - `DROP`: Odstranění databáze, tabulky nebo objektu.
  - `TRUNCATE`: Odstranění všech řádků z tabulky, ale zachování její struktury.
- **DML (Data Manipulation Language)**: Příkazy pro manipulaci s daty.
  - `SELECT`: Výběr dat z jedné nebo více tabulek.
  - `INSERT`: Vložení nových dat do tabulky.
  - `UPDATE`: Aktualizace existujících dat v tabulce.
  - `DELETE`: Odstranění dat z tabulky.

- **DCL (Data Control Language)**: Příkazy pro řízení přístupu k datům.
  - `GRANT`: Udělení práv na databázové objekty uživatelům.
  - `REVOKE`: Odebrání dříve udělených práv.
- **TCL (Transaction Control Language)**: Příkazy pro správu transakcí.
  - `COMMIT`: Potvrzení všech změn provedených v transakci.
  - `ROLLBACK`: Vrácení změn provedených v transakci zpět do stavu před zahájením transakce.
  - `SAVEPOINT`: Nastavení bodu v transakci, ke kterému se lze vrátit.

Primární a cizí klíče

- **Primární klíč (Primary Key)**: Unikátní identifikátor pro záznam v tabulce. Primární klíč musí být jedinečný pro každý záznam a nemůže obsahovat `NULL` hodnoty. Používá se k zajištění, že každý řádek v tabulce je jednoznačně identifikovatelný. Například tabulka `Employees` může mít sloupec `EmployeeID` jako primární klíč.

- **Cizí klíč (Foreign Key)**: Sloupec nebo soubor sloupců, které odkazují na primární klíč v jiné tabulce. Cizí klíče slouží k vytvoření vazby mezi tabulkami a zajišťují, že data mezi nimi zůstanou konzistentní. Například, pokud máme tabulky `Orders` a `Customers`, sloupec `CustomerID` v tabulce `Orders` může být cizím klíčem odkazujícím na `CustomerID` v tabulce `Customers`.

Relace, Relační vazby, referenční integrita

- **Relace**: V SQL databázích se data ukládají do tabulek, které jsou propojeny relacemi. Relace mezi tabulkami mohou mít různé typy:
  - **1:1 (jedna k jedné)**: Jeden řádek v tabulce A odpovídá jednomu řádku v tabulce B.
  - **1:N (jedna k mnoha)**: Jeden řádek v tabulce A může odpovídat více řádkům v tabulce B.
  - **N:M (mnoho k mnoha)**: Více řádků v tabulce A může odpovídat více řádkům v tabulce B, což obvykle vyžaduje třetí, spojovací tabulku.
- **Referenční integrita**: Tento koncept zajišťuje, že vztahy mezi tabulkami jsou platné a konzistentní. Například referenční integrita zajišťuje, že nelze vložit záznam do tabulky `Orders` s neexistujícím `CustomerID`, pokud je `CustomerID` cizím klíčem odkazujícím na tabulku `Customers`.

Pravidla referenční integrity v SQL Serveru

Referenční integrita v SQL Serveru je klíčovým konceptem pro zajištění konzistence a správnosti dat v databázích. Umožňuje správnou vazbu mezi tabulkami a zajišťuje, že vztahy mezi nimi jsou udržovány podle definovaných pravidel. Níže jsou uvedena hlavní pravidla referenční integrity:

Cizí klíč (Foreign Key)
Cizí klíč je sloupec nebo skupina sloupců v tabulce, které odkazují na primární klíč (Primary Key) v jiné tabulce. Tento vztah zajišťuje, že hodnoty ve sloupci cizího klíče odpovídají existujícím hodnotám v tabulce, na kterou se odkazují.

Primární klíč (Primary Key)
Primární klíč je sloupec nebo skupina sloupců, které jednoznačně identifikují každý řádek v tabulce. V tabulce může být pouze jeden primární klíč a žádná jeho hodnota nesmí být NULL ani duplicitní.

Akce při aktualizaci a mazání (ON UPDATE/ON DELETE)
Tato pravidla určují, co se stane s hodnotami cizího klíče, když se změní nebo smaže odpovídající hodnota v primárním klíči:

- **ON UPDATE CASCADE:** Automaticky aktualizuje cizí klíče, když se změní primární klíč.
- **ON DELETE CASCADE:** Automaticky smaže řádky obsahující cizí klíče, když se smaže odpovídající řádek v tabulce s primárním klíčem.
- **ON DELETE SET NULL:** Nastaví cizí klíče na NULL, pokud je odpovídající řádek s primárním klíčem smazán.
- **ON DELETE SET DEFAULT:** Nastaví cizí klíče na výchozí hodnotu, pokud je odpovídající řádek s primárním klíčem smazán.

5. **Kontrola existence (Referential Integrity Check)**
   SQL Server zajišťuje, že žádný řádek nemůže být vložen do tabulky s cizím klíčem, pokud odpovídající hodnota neexistuje v tabulce s primárním klíčem. Tato kontrola zajišťuje, že všechny odkazy mezi tabulkami jsou platné.

Datové typy sloupců v SQL Serveru

Datové typy sloupců v SQL Serveru určují, jaký typ dat může sloupec uchovávat. Výběr správného datového typu je klíčový pro optimalizaci výkonu a efektivitu úložiště. SQL Server nabízí širokou škálu datových typů, které lze rozdělit do několika kategorií:

Numerické datové typy

- **INT**: Celá čísla v rozsahu -2,147,483,648 až 2,147,483,647.
- **BIGINT**: Větší celá čísla, v rozsahu -9,223,372,036,854,775,808 až 9,223,372,036,854,775,807.
- **SMALLINT**: Menší celá čísla v rozsahu -32,768 až 32,767.
- **TINYINT**: Nepodepsaná celá čísla v rozsahu 0 až 255.
- **DECIMAL(p, s)** nebo **NUMERIC(p, s)**: Čísla s pevnou přesností a měřítkem, kde p určuje počet číslic a s počet číslic za desetinnou čárkou.

- **FLOAT** a **REAL**: Čísla s plovoucí desetinnou čárkou, používaná pro hodnoty, kde je požadována vysoká přesnost.
- **MONEY**: Používá se pro ukládání peněžních hodnot s pevnou přesností na čtyři desetinná místa. Rozsah hodnot je od -922,337,203,685,477.5808 do 922,337,203,685,477.5807.
- **SMALLMONEY**: Menší verze datového typu `MONEY`, ukládá peněžní hodnoty s pevnou přesností na čtyři desetinná místa a s rozsahem od -214,748.3648 do 214,748.3647.

Řetězcové datové typy

CHAR (Fixed-length character)

- **Pevná délka:** Datový typ `CHAR` má pevnou délku, což znamená, že pokud je definován jako `CHAR(10)`, každý záznam v tomto sloupci bude mít délku 10 znaků. Pokud je vložený řetězec kratší než 10 znaků, SQL Server automaticky doplní chybějící znaky prázdnými místy (spaces).
- **Použití:** `CHAR` je vhodný pro ukládání textových dat, která mají přesně stanovenou délku, jako jsou kódy, identifikační čísla nebo zkratky.

VARCHAR (Variable-length character)

- **Proměnná délka:** Datový typ `VARCHAR` má proměnnou délku, což znamená, že je možné uložit řetězce různé délky, až do maximální délky specifikované při definici (např. `VARCHAR(50)`). SQL Server ukládá pouze skutečnou délku vloženého řetězce, což může ušetřit místo v případě, že jsou uloženy kratší řetězce.
- **Použití:** `VARCHAR` je vhodný pro ukládání textových dat, jejichž délka se může výrazně lišit, jako jsou jména, adresy nebo popisy.

Datumové a časové datové typy

- **DATE**: Ukládá datum (rok, měsíc, den).
- **TIME**: Ukládá čas (hodiny, minuty, sekundy).
- **DATETIME**: Ukládá datum a čas, s přesností na sekundy.
- **SMALLDATETIME**: Ukládá datum a čas, s přesností na minuty.
- **DATETIME2**: Ukládá datum a čas s vyšší přesností než `DATETIME`.
- **DATETIMEOFFSET**: Ukládá datum a čas včetně časového pásma.

Binární datové typy

- **BINARY(n)**: Pevná délka binárních dat.
- **VARBINARY(n)**: Proměnná délka binárních dat, kde n určuje maximální počet bajtů.
- **IMAGE**: Používal se pro velmi velká binární data, nyní je doporučeno používat `VARBINARY(MAX)`.

Používání uvozovek, jednoduchých uvozovek, zpětných apostrofů, závorek a komentářů v SQL Serveru

Uvozovky (Double Quotes)

V SQL Serveru se uvozovky používají k označení názvů objektů, jako jsou tabulky nebo sloupce, které obsahují speciální znaky nebo jsou klíčovými slovy. Uvozovky mohou také sloužit k zachování malých a velkých písmen v názvech objektů, pokud je to vyžadováno, ačkoliv SQL Server obvykle nerozlišuje malá a velká písmena, pokud není specifikována case-sensitive kolace.

Jednoduché uvozovky (Single Quotes)

Jednoduché uvozovky se používají k označení textových literálů. Všechny textové hodnoty musí být obklopeny jednoduchými uvozovkami. Pokud je nutné vložit jednoduchou uvozovku do textového řetězce, používají se dvojité jednoduché uvozovky uvnitř řetězce.

Zpětné apostrofy (Backticks)

V SQL Serveru se zpětné apostrofy běžně nepoužívají. Místo toho se k označení názvů objektů, které obsahují speciální znaky nebo mezery, používají hranaté závorky nebo dvojité uvozovky. Použití zpětných apostrofů může vést k syntaktické chybě, protože nejsou v T-SQL standardem.

Malá a velká písmena v příkazech

SQL Server je obvykle case-insensitive, což znamená, že nerozlišuje mezi malými a velkými písmeny v příkazech a názvech objektů. Klíčová slova SQL, jako `SELECT`, `FROM` a `WHERE`, mohou být psána v jakékoliv kombinaci malých a velkých písmen. Pokud je však použita kolace, která rozlišuje malá a velká písmena, může SQL Server rozlišovat písmena i v názvech objektů a hodnotách.

Práce se závorkami

- **Kulaté závorky (Parentheses)**: Používají se k určení pořadí operací, vytváření poddotazů, definování argumentů funkcí a při práci s výrazovými seznamy. Kulaté závorky jsou základním prvkem syntaxe pro správnou strukturu a čitelnost složitých dotazů.

- **Hranaté závorky (Square Brackets)**: Slouží k označení názvů objektů, jako jsou tabulky a sloupce, které obsahují mezery, speciální znaky nebo jsou rezervovanými slovy v SQL. Pomáhají také zachovat malá a velká písmena v názvech objektů, pokud je to vyžadováno.

- **Složené závorky (Curly Braces)**: V T-SQL se složené závorky běžně nepoužívají jako součást syntaxe SQL. Mohou se však objevit v některých nástrojích pro správu databází nebo při použití rozšířených funkcí, jako jsou CTE (Common Table Expressions) nebo při tvorbě dynamických SQL dotazů.

Komentáře

Komentáře jsou v SQL Serveru používány pro přidávání poznámek k SQL kódu, které nejsou vykonávány databázovým systémem. Existují dva hlavní způsoby, jak psát komentáře:

1. **Jednořádkové komentáře**: Označují se dvěma pomlčkami (`--`). Všechny znaky po těchto pomlčkách až do konce řádku jsou považovány za komentář.

2. **Víceřádkové komentáře**: Označují se začátkem `/*` a koncem `*/`. Všechny znaky mezi těmito symboly jsou považovány za komentář, což umožňuje komentovat bloky kódu přes více řádků.

Zajištění datové integrity za pomocí SSMS

Datová integrita v SQL Serveru zajišťuje, že data v databázi jsou přesná a konzistentní. SQL Server Management Studio (SSMS) nabízí nástroje a funkce, které pomáhají správcům databází implementovat a udržovat datovou integritu. Mezi tyto nástroje patří definice omezení (constraints), pravidel referenční integrity a nastavení výchozích hodnot pro sloupce.

Základní orientace v SQL Server Management Studio (SSMS)

SQL Server Management Studio (SSMS) je integrované prostředí pro správu SQL Serveru. Poskytuje nástroje pro konfiguraci, správu a administraci všech komponent SQL Serveru. Základní prvky SSMS zahrnují:

- **Object Explorer**: Umožňuje prohlížet a spravovat objekty databáze jako jsou tabulky, zobrazení, procedury a funkce.
- **Query Editor**: Umožňuje psát a spouštět SQL dotazy.
- **Solution Explorer**: Umožňuje organizovat SQL soubory a projekty.

Omezení na vkládané hodnoty - NULL a NOT NULL, CONSTRAINTS

V SQL Serveru lze omezit vkládané hodnoty do sloupců pomocí následujících mechanismů:

- **NULL a NOT NULL**: Určují, zda sloupec může obsahovat hodnoty `NULL`. `NOT NULL` zajišťuje, že každý řádek musí mít v daném sloupci hodnotu.
- **CONSTRAINTS**: Omezení, která se aplikují na data v tabulce, aby se zajistila jejich správnost. Typy omezení zahrnují:
  - **PRIMARY KEY**: Zajišťuje jedinečnost a nepřítomnost `NULL` hodnot v jednom nebo více sloupcích.
  - **FOREIGN KEY**: Zajišťuje, že hodnoty v jednom nebo více sloupcích odpovídají hodnotám v primárním klíči jiné tabulky.
  - **UNIQUE**: Zajišťuje jedinečnost hodnot v jednom nebo více sloupcích.
  - **CHECK**: Omezuje hodnoty, které mohou být do sloupce vloženy na základě logického výrazu.

Výchozí hodnoty

Výchozí hodnoty v SQL Serveru jsou přednastavené hodnoty, které jsou automaticky přiřazeny sloupcům v tabulce, pokud uživatel explicitně nezadá hodnotu při vložení nového záznamu. To se provádí pomocí klíčového slova `DEFAULT`. Například:

```
CREATE TABLE data_nova (
    stanice NVARCHAR(50) PRIMARY KEY,
    lokalita NVARCHAR(50) NOT NULL,
    srazky FLOAT,
    datum DATE DEFAULT GETDATE()
);
```

Unikátní hodnoty

Omezení `UNIQUE` zajišťuje, že všechny hodnoty v určitém sloupci nebo kombinaci sloupců jsou jedinečné v rámci tabulky. Toto omezení se používá k zabránění duplicitám. Příklad použití:

```
CREATE TABLE dim_lokality (
    stanice NVARCHAR(50) PRIMARY KEY,
    zeme NVARCHAR(50),
    UNIQUE (stanice, zeme)
);
```

Tímto se zajistí, že každá hodnota v sloupci `Email` bude jedinečná, což znamená, že žádní dva zaměstnanci nebudou mít stejnou e-mailovou adresu.

Identita a její automatické generování

SQL Server umožňuje automatické generování hodnot v primárním klíči pomocí klíčového slova `IDENTITY`. Tento mechanismus je často využíván pro číselné identifikátory. Například:

```
CREATE TABLE data_nova (
    zaznam_id INT IDENTITY(1,1) PRIMARY KEY,
    lokalita NVARCHAR(50) NOT NULL,
    srazky FLOAT,
    datum DATE DEFAULT GETDATE()
);
```

V tomto příkladu se `zaznam_id` automaticky inkrementuje o 1 pro každý nový záznam, počínaje hodnotou 1.

Referenční integrita - primární a cizí klíče

Referenční integrita zajišťuje, že vztahy mezi tabulkami jsou konzistentní. To se provádí pomocí:

- **Primárního klíče (PRIMARY KEY)**: Zajišťuje jedinečnost každého záznamu v tabulce.

- **Cizího klíče (FOREIGN KEY)**: Odkazuje na primární klíč v jiné tabulce, zajišťuje, že data mezi tabulkami zůstávají propojená a konzistentní.

Příklad:

```
CREATE TABLE dim_oblasti (
    stanice NVARCHAR(50) PRIMARY KEY,
    zeme NVARCHAR(100)
);

CREATE TABLE data_nova (
    lokalita NVARCHAR(50),
    datum DATE DEFAULT GETDATE(),
    srazky FLOAT,
    FOREIGN KEY (lokalita) REFERENCES dim_oblasti(stanice)
);
```

V tomto příkladu cizí klíč CustomerID v tabulce Orders odkazuje na primární klíč CustomerID v tabulce Customers, čímž se zajišťuje, že každá objednávka je spojena s existujícím zákazníkem.

Logika INDEXŮ

Indexy v SQL Serveru slouží k zrychlení přístupu k datům v tabulce. Fungují podobně jako index v knize, kde umožňují rychle najít konkrétní záznamy bez nutnosti procházet celou tabulku. Existují dva hlavní typy indexů:

Clustered Index

- **Clustered Index** určuje fyzické pořadí záznamů v tabulce. To znamená, že data v tabulce jsou fyzicky uspořádána podle hodnoty v clustered indexu.
- Každá tabulka může mít pouze jeden clustered index, protože data mohou být uspořádána pouze jedním způsobem.
- Clustered indexy jsou často vytvořeny na sloupcích, které jsou často používány k třídění a filtrování dat, jako je primární klíč.

Příklad vytvoření clustered indexu:

```
CREATE TABLE dim_oblasti (
    stanice NVARCHAR(50) PRIMARY KEY,
    zeme NVARCHAR(50)
);
```

V tomto příkladu OrderID automaticky vytvoří clustered index, protože je to primární klíč.

Non-Clustered Index

Non-Clustered Index vytváří oddělenou strukturu, která obsahuje kopie klíčových hodnot a ukazatele na skutečné řádky dat v tabulce. Tabulka může mít více non-clustered indexů. Non-clustered indexy jsou užitečné pro rychlé vyhledávání, třídění a filtrování dat na základě sloupců, které nejsou součástí clustered indexu.

Příklad vytvoření non-clustered indexu:

```
CREATE INDEX data_nova_lokalita_datum ON data_nova (lokalita,datum);
```

V tomto příkladu index na CustomerID umožní rychlé vyhledávání objednávek podle identifikátoru zákazníka.

Použití a optimalizace indexů

- **Optimalizace dotazů**: Indexy zrychlují dotazy, které často třídí, filtrují nebo agregují data. Je důležité analyzovat, které dotazy jsou nejčastější, a podle toho vytvářet indexy.
- **Náklady na údržbu**: Indexy přidávají náklady na údržbu, zejména při vkládání, aktualizaci a mazání záznamů, protože indexy musí být aktualizovány při každé změně dat.
- **Pokrytí indexů**: Někdy je výhodné vytvořit indexy, které pokryjí všechny sloupce požadované dotazem, což může eliminovat potřebu přístupu k základním datům tabulky.

Rozdíly mezi clustered a non-clustered indexy

Aspekt	Clustered Index	Non-Clustered Index
Fyzické uspořádání dat	Určuje fyzické uspořádání záznamů v tabulce.	Vytváří samostatnou strukturu oddělenou od skutečných dat.
Počet indexů na tabulku	Pouze jeden na tabulku.	Může být více než jeden na tabulku.
Klíčové hodnoty a řádky	Klíčové hodnoty přímo odpovídají fyzickému umístění řádků.	Obsahuje kopie klíčových hodnot a ukazatele na fyzické umístění dat.
Přístup k datům	Rychlý přístup k datům, pokud je dotaz zaměřen na klíčové sloupce.	Může být méně efektivní při přímém přístupu k datům, ale je flexibilní pro různé typy dotazů.
Použití u primárního klíče	Často se používá u primárního klíče.	Není typicky použit u primárního klíče, ale může být použit pro jedinečné omezení.
Vliv na INSERT/UPDATE/DELETE operace	Může výrazně ovlivnit výkon při vkládání, aktualizaci a mazání záznamů.	Může také ovlivnit výkon při změnách dat, ale méně výrazně než clustered index.
Podpora u více sloupců	Podporuje vícesloupcové klíče.	Podporuje vícesloupcové klíče.
Způsob použití	Optimální pro dotazy, které vrací rozsahy dat.	Vhodný pro různé dotazy zahrnující třídění, filtrování a vyhledávání podle klíčových hodnot.

Použití příkazu SELECT

Příkaz SELECT je jedním z nejpoužívanějších příkazů v SQL a slouží k dotazování a načítání dat z databázových tabulek. Jeho základní syntaxe umožňuje vybrat konkrétní sloupce a řádky z jedné nebo více tabulek.

Základní syntaxe a použití

Základní syntaxe příkazu SELECT je následující:

Základní syntaxe příkazu `SELECT` je následující:

```
SELECT YEAR(datum) AS rok, lokalita, SUM(srazky) AS rocni_srazky
FROM data_nova
LEFT JOIN dim_oblasti ON data_nova.lokalita = dim_oblasti.stanice
WHERE dim_oblasti.zeme = 'Cesko'
GROUP BY YEAR(datum), lokalita
HAVING SUM(srazky) > 1000
ORDER BY rocni_srazky DESC
```

- **SELECT**: Klíčové slovo, které určuje, že se jedná o dotaz pro načtení dat.
- **sloupec1, sloupec2, ...**: Seznam sloupců, které chcete vybrat. Můžete použít hvězdičku (*), abyste vybrali všechny sloupce z tabulky.
- **FROM**: Klíčové slovo, které určuje tabulku, ze které chcete data načíst.
- **tabulka**: Název tabulky, ze které se data vybírají.
- **JOIN**: (Volitelné) Klauzule pro kombinování řádků z více tabulek na základě podmínky slučování.
- **ON podmínka_slučování**: Podmínka, která určuje, jak se mají tabulky slučovat na základě společných sloupců.
- **WHERE**: (Volitelné) Klauzule, která specifikuje podmínky pro filtrování řádků. Pouze řádky, které splňují tuto podmínku, budou vráceny.
- **GROUP BY**: (Volitelné) Klauzule, která seskupuje řádky podle jednoho nebo více sloupců. Používá se často s agregačními funkcemi.
- **HAVING**: (Volitelné) Klauzule pro filtrování seskupených řádků podle podmínek, které nelze použít ve `WHERE`.
- **ORDER BY**: (Volitelné) Klauzule pro třídění výsledků podle jednoho nebo více sloupců v určitém pořadí (vzestupně `ASC` nebo sestupně `DESC`).

Pořadí exekuce příkazů

- **FROM**: Určuje tabulku nebo tabulky, ze kterých se data vybírají.
- **JOIN**: (Volitelné) Kombinování řádků z více tabulek na základě podmínky slučování.
- **WHERE**: (Volitelné) Specifikuje podmínky pro filtrování řádků. Pouze řádky, které splňují tuto podmínku, budou vráceny.
- **GROUP BY**: (Volitelné) Seskupuje řádky podle jednoho nebo více sloupců. Často se používá s agregačními funkcemi (např. COUNT, SUM).
- **HAVING**: (Volitelné) Filtrování seskupených řádků na základě podmínek, které nelze použít ve `WHERE`.
- **SELECT**: Určuje sloupce, které chcete vybrat. Můžete použít hvězdičku (*), abyste vybrali všechny sloupce z tabulky.
- **ORDER BY**: (Volitelné) Třídění výsledků podle jednoho nebo více sloupců v určitém pořadí (vzestupně `ASC` nebo sestupně `DESC`).

Používání aliasů v SQL Serveru

Alias je alternativní název, který lze přiřadit tabulce nebo sloupci v SQL dotazu. Alias usnadňuje čtení a psaní dotazů, zejména když pracujete s více tabulkami nebo složitými výpočty. Alias se definuje pomocí klíčového slova `AS` (které je volitelné) a je platný pouze po dobu trvání dotazu.

Alias pro sloupce

Alias pro sloupec umožňuje přiřadit nový název pro výstupní sloupec. To je užitečné, pokud chcete zobrazit srozumitelnější názvy nebo pokud provádíte výpočty, jejichž výsledky chcete pojmenovat. Syntaxe je následující:

```
SELECT lokalita AS mesto, srazky
FROM data_nova;
```

Alias pro tabulky

Alias pro tabulku je zkratka, která usnadňuje odkazování na tabulky v dotazu. To je obzvláště užitečné při práci s více tabulkami a při používání klauzule JOIN. Syntaxe je následující:

```
SELECT dn.datum, dn.lokalita, dn.srazky,do.zeme
FROM data_nova AS dn
JOIN dim_oblasti AS do ON dn.lokalita = do.stanice;;
```

Kombinace aliasů pro tabulky a sloupce

Je možné kombinovat aliasy pro tabulky i sloupce v jednom dotazu:

```
SELECT dn.datum, dn.lokalita as mesto, dn.srazky,do.zeme
FROM data_nova AS dn
JOIN dim_oblasti AS do ON dn.lokalita = do.stanice;
```

Práce se sloupci za pomocí textových, matematických a datumových funkcí

Textové funkce

- **`LEN()`**: Vrací délku řetězce (počet znaků).
- **`UPPER()` a `LOWER()`**: Převádí všechny znaky v řetězci na velká nebo malá písmena.
- **`SUBSTRING()`**: Extrahuje podřetězec z řetězce, počínaje na specifikované pozici.
- **`REPLACE()`**: Nahrazuje výskyty určitého textu jiným textem v řetězci.
- **`TRIM()`**: Odstraňuje mezery z obou stran řetězce.

Matematické funkce

- **`ABS()`**: Vrací absolutní hodnotu čísla.
- **`ROUND()`**: Zaokrouhluje číslo na daný počet desetinných míst.
- **`CEILING()` a `FLOOR()`**: Zaokrouhluje číslo na nejbližší celé číslo směrem nahoru nebo dolů.
- **`POWER()`**: Vrací hodnotu čísla umocněného na zadanou mocninu.
- **`SQRT()`**: Vypočítá druhou odmocninu čísla.

Datumové funkce

- **`GETDATE()`**: Vrací aktuální datum a čas.
- **`DATEADD()`**: Přidává k datu určitý počet specifikovaných časových jednotek.
- **`DATEDIFF()`**: Vrací rozdíl mezi dvěma daty ve specifikovaných časových jednotkách.
- **`FORMAT()`**: Formátuje datum nebo čas podle specifikovaného formátu.
- **`CONVERT()`**: Konvertuje datum na jiný formát.

Extrakce konkrétních částí data

- **`YEAR()`**: Extrahuje rok z datového typu.
- **`MONTH()`**: Extrahuje měsíc z datového typu.
- **`DAY()`**: Extrahuje den z datového typu.
- **`DATEPART(quarter, date)`**: Vrací čtvrtletí (1-4) pro dané datum.

Příklad:

```
SELECT FORMAT(dn.datum,'d. MMMM yyyy', 'cs-CZ'), LOWER(dn.lokalita) as mest
o,dn.srazky,LOWER(do.zeme)
FROM data_nova AS dn
JOIN dim_oblasti AS do ON dn.lokalita = do.stanice;
```

Řazení výsledků pomocí ORDER BY

V SQL Serveru se klauzule `ORDER BY` používá k řazení výsledků dotazu podle jednoho nebo více sloupců. Můžete specifikovat, zda chcete výsledky řadit vzestupně (`ASC`) nebo sestupně (`DESC`). Pokud není specifikován žádný směr, je výchozí řazení vzestupné.

Základní použití ORDER BY

Klauzuli `ORDER BY` umístíme na konec SQL dotazu, po všech dalších klauzulích jako `SELECT`, `FROM`, `WHERE` a `GROUP BY`. Můžeme řadit podle jednoho nebo více sloupců, a to jak vzestupně, tak sestupně.

Příklady

Řazení podle jednoho sloupce vzestupně:

Pokud chcete řadit výsledky podle jednoho sloupce, například `datum`, jednoduše použijte:

```
ORDER BY datum ASC;
```

Řazení podle jednoho sloupce sestupně:

Pro řazení sestupně použijte klíčové slovo `DESC`. Tímto způsobem můžete například řadit výsledky podle sloupce `datum` v sestupném pořadí.

```
ORDER BY datum DESC;
```

Řazení podle více sloupců

Pokud chcete řadit podle více sloupců, oddělte je čárkou. Například, můžete nejprve seřadit podle `datum` a poté podle `lokalita`. Toto pořadí určí, že pokud budou dva záznamy mít stejný `datum`, budou se dále řadit podle `lokalita`.

```
ORDER BY datum ASC, lokalita ASC;
```

Řazení podle odvozených sloupců a aliasů

Je možné řadit i podle odvozených sloupců nebo aliasů, které jsou vytvořeny v dotazu. Například můžete vytvořit odvozený sloupec `AnnualSalary` jako alias pro roční plat a řadit výsledky podle tohoto aliasu v sestupném pořadí.

```
SELECT datum, lokalita as mesto, srazky
FROM data_nova
ORDER BY mesto DESC;
```

Filtrování dat pomocí WHERE

Klauzule `WHERE` se v SQL používá k filtrování řádků v tabulce na základě specifických podmínek. Použitím `WHERE` můžete vrátit pouze ty řádky, které splňují určité kritérium, čímž zefektivníte dotazy a snížíte množství vrácených dat. Klauzule `WHERE` se obvykle používá s příkazy `SELECT`, `UPDATE`, `DELETE` a `INSERT`.

Základní syntaxe WHERE

```
SELECT datum, srazky
FROM data_nova
WHERE lokalita = 'RUZYNE';
```

Používání výrazů a operátorů

V SQL se výrazy a operátory používají k manipulaci a vyhodnocování dat. Operátory mohou být aritmetické, relační, logické a další. Některé běžné operátory zahrnují:

Aritmetické operátory

- + (sčítání)
- - (odečítání)
- * (násobení)
- / (dělení)
- % (modulo - zbytek po dělení)

Relační operátory

- = (rovno)
- != nebo <> (nerovno)
- > (větší než)
- < (menší než)
- >= (větší nebo rovno)
- <= (menší nebo rovno)

Efektivní vyhledávání záznamů - ignorování velkých, malých písmen a diakritiky

V SQL můžeme efektivně vyhledávat záznamy bez ohledu na velikost písmen a diakritiku pomocí následujících technik:

- **Použití funkce `LOWER()` nebo `UPPER()`**: Tímto způsobem můžeme transformovat text na malé nebo velké písmo a poté porovnat. Například:

```
SELECT * FROM data_nova WHERE LOWER(lokalita) = 'Ruzyně';
```

- **Ignorování diakritiky:** SQL Server podporuje kolace, které mohou ignorovat diakritiku. Například:

```
  SELECT * FROM data_nova WHERE lokalita COLLATE SQL_Latin1_General_CP1_CI_
AI = 'Ruzyně';
```

SQL_Latin1_General_CP1_CI_AI znamená case-insensitive (CI) a accent-insensitive (AI) kolaci, což znamená, že porovnávání bude ignorovat velikost písmen i diakritiku.

Vyhledávání NULL hodnot a zobrazení výchozích hodnot místo NULL

NULL hodnoty v SQL Serveru představují neznámé nebo chybějící hodnoty. Pro vyhledávání NULL hodnot můžeme použít operátor `IS NULL` nebo `IS NOT NULL`.

Příklad vyhledávání NULL hodnot:

```
SELECT datum,lokalita FROM data_nova WHERE srazky IS NULL;
```

Příklad použití ISNULL():

```
SELECT lokalita, ISNULL(CAST(srazky AS NVARCHAR(50)), 'nevyplneno') AS sraz
ky
FROM data_nova;
```

Tento příkaz zobrazí sloupec PhoneNumber s výchozí hodnotou 'N/A' tam, kde je NULL.

Příklad použití COALESCE():

```
SELECT lokalita, COALESCE(prumerna_teplota,minimalni_teplota,maximalni_tepl
ota) AS teplota FROM data_nova;
```

Funkce COALESCE() může zpracovávat více argumentů a vrací první ne-NULL hodnotu ze seznamu.

Seskupování dat pomocí GROUP BY

Příkaz `GROUP BY` se používá v SQL k seskupení řádků, které mají stejné hodnoty v jedné nebo více sloupcích. Tento příkaz je často používán s agregačními funkcemi, jako jsou `COUNT()`, `SUM()`, `AVG()`, `MAX()`, a `MIN()`, k provedení výpočtů na skupinách dat.

Příklad použití `GROUP BY`:

```
SELECT lokalita, AVG(srazky) AS prumerne_srazky
FROM data_nova
GROUP BY lokalita;
```

Tento příklad vrátí počet zaměstnanců v každém oddělení. Všechny záznamy jsou seskupeny podle oddělení (Department).

GROUP BY vs. Alias sloupců

Při použití aliasů sloupců v SQL dotazu je důležité vědět, že aliasy nelze přímo použít ve výrazu `GROUP BY`. Alias je vytvořen ve fázi `SELECT`, což je pozdější fáze než `GROUP BY`. Proto musíme použít původní název sloupce nebo vyjádření ve výrazu `GROUP BY`.

Příklad aliasu a `GROUP BY`:

```
SELECT lokalita as mesto, AVG(srazky) AS prumerne_srazky
FROM data_nova
GROUP BY lokalita
```

V tomto příkladu používáme alias Dept ve výstupu, ale ve výrazu GROUP BY musíme stále používat původní název sloupce Department.

Základní agregační funkce

Agregační funkce provádějí výpočty na více hodnotách a vracejí jednu hodnotu. Běžné agregační funkce zahrnují:

- **COUNT()**: Počet řádků. Počítá počet záznamů, včetně těch s hodnotou NULL, pokud je použito `COUNT(*)`, nebo pouze ne-NULL hodnoty při použití `COUNT(column_name)`.
- **SUM()**: Součet hodnot ve sloupci. Používá se pro sloupce s číselnými hodnotami a sčítá všechny hodnoty, ignoruje NULL hodnoty.
- **AVG()**: Průměr hodnot ve sloupci. Vypočítá průměr číselných hodnot, ignoruje NULL hodnoty.
- **MAX()**: Maximální hodnota ve sloupci. Najde největší hodnotu ve sloupci, ignoruje NULL hodnoty.
- **MIN()**: Minimální hodnota ve sloupci. Najde nejmenší hodnotu ve sloupci, ignoruje NULL hodnoty.
- **STDEV()**: Vypočítá směrodatnou odchylku pro daný sloupec.

- **VAR()**: Vypočítá rozptyl pro daný sloupec.

Příklad použití agregačních funkcí:

```
SELECT lokalita ,
       SUM(srazky) AS suma_srazek,
       AVG(srazky) AS prumerne_srazky,
       MAX(srazky) AS maximalni_srazky
FROM data_nova
GROUP BY lokalita;
```

Agregační funkce a zpracovávání NULL hodnot

NULL hodnoty v SQL Serveru představují neznámé nebo chybějící hodnoty. Při práci s NULL hodnotami je důležité vědět, jak se chovají různé agregační funkce, protože NULL hodnoty mohou ovlivnit výsledky výpočtů.

Chování jednotlivých agregačních funkcí s NULL hodnotami:

- **COUNT(column_name)**: Počítá pouze ne-NULL hodnoty ve sloupci. NULL hodnoty jsou ignorovány.
- **COUNT(*)**: Počítá všechny řádky, včetně těch s NULL hodnotami.
- **SUM()**: Sčítá všechny ne-NULL hodnoty ve sloupci. NULL hodnoty jsou ignorovány, což znamená, že nepřidávají žádnou hodnotu do celkového součtu.
- **AVG()**: Vypočítává průměr pouze z ne-NULL hodnot. NULL hodnoty jsou ignorovány, a proto neovlivňují průměr.
- **MAX()** a **MIN()**: Tyto funkce vrací maximální nebo minimální hodnotu ve sloupci, ignorujíc NULL hodnoty.

Filtrování seskupených dat pomocí HAVING

Klauzule HAVING se používá v SQL k filtrování výsledků, které byly seskupeny pomocí GROUP BY. Na rozdíl od klauzule WHERE, která filtruje řádky před seskupením, HAVING filtruje skupiny po aplikaci agregačních funkcí.

Kdy použít HAVING

- **Po použití GROUP BY**: HAVING je často používán v kombinaci s GROUP BY, aby se omezily výsledky na základě agregačních funkcí, jako jsou COUNT(), SUM(), AVG(), MAX(), a MIN().
- **K filtrování na základě agregačních výsledků**: Například, pokud chcete zobrazit pouze ty skupiny, kde je počet položek v určité skupině větší než určitá hodnota.

Příklad použití HAVING

Představte si, že chcete zjistit, která oddělení ve firmě mají více než 10 zaměstnanců. Použijete GROUP BY k seskupení zaměstnanců podle oddělení a HAVING k filtrování těch oddělení, která mají více než 10 zaměstnanců.

Klauzule HAVING je užitečná, když potřebujete aplikovat podmínky na souhrnné výsledky, což nelze provést pomocí WHERE, protože WHERE nemůže pracovat s agregačními funkcemi.

```
SELECT lokalita ,
       SUM(srazky) AS suma_srazek,
       AVG(srazky) AS prumerne_srazky,
       MAX(srazky) AS maximalni_srazky
FROM data_nova
GROUP BY lokalita
HAVING AVG(srazky) > 5
```

Omezování množství vrácených záznamů pomocí TOP

Klauzule `TOP` v SQL Serveru se používá k omezení počtu vrácených záznamů na základě specifikovaného počtu nebo procenta. Tato klauzule je velmi užitečná, když potřebujete pracovat pouze s omezeným množstvím dat, například při analýze nebo prezentaci nejnovějších nebo nejvýznamnějších záznamů.

Použití klauzule TOP

- **Omezení na pevný počet záznamů**: Klauzule `TOP` umožňuje vrátit pouze první N záznamů z výsledné sady. Tento přístup je často používán, když potřebujete pracovat s konkrétním počtem největších nebo nejmenších hodnot v tabulce.
- **Omezení na procento záznamů**: `TOP` může být také použito k vrácení určitého procenta záznamů z celkové výsledné sady. To je užitečné například při analýze vzorku dat.

Kombinace s klauzulí ORDER BY

Klauzule `TOP` je často používána v kombinaci s `ORDER BY` k určení pořadí záznamů, které budou vráceny. Například, pokud chcete získat nejvyšší nebo nejnižší hodnoty, `ORDER BY` určí pořadí, a `TOP` omezí počet vrácených záznamů.

Příklad použití TOP

Pokud máte tabulku se záznamy o prodejích a chcete zobrazit pouze 5 největších prodejů, můžete použít klauzuli `TOP` spolu s `ORDER BY` k seřazení záznamů podle výše prodeje v sestupném pořadí a vrátit pouze prvních 5 záznamů.

Výhody použití TOP

- **Zlepšení výkonu**: Omezení počtu vrácených záznamů může zlepšit výkon dotazu, zejména u velkých datových sad. SQL Server nemusí zpracovávat všechny záznamy, což snižuje čas a zdroje potřebné k provedení dotazu.
- **Efektivní zpracování dat**: Klauzule `TOP` umožňuje rychlé získání relevantních informací z velkých tabulek, což je užitečné například při vytváření přehledů nebo analýze dat.

Použití klauzule `TOP` je efektivní způsob, jak omezit množství vrácených dat a zlepšit výkon dotazů v SQL Serveru.

```sql
SELECT TOP 5 lokalita, SUM(srazky) as suma_srazek
FROM data_nova
GROUP BY lokalita
ORDER BY suma_srazek DESC;
```

Dopad dotazů na výkon SQL Serveru

Efektivita dotazů má zásadní vliv na výkon SQL Serveru. Níže jsou uvedeny klíčové aspekty týkající se dopadu klauzulí `SELECT`, `WHERE`, `GROUP BY`, `ORDER BY` a `TOP` na výkon dotazů.

SELECT

- **Výběr sloupců**: Použití `SELECT *` načítá všechny sloupce, což může vést ke zbytečnému přenosu dat a zpomalení dotazů. Je vhodné specifikovat pouze potřebné sloupce.
- **Odvozené sloupce a výpočty**: Výpočty v `SELECT` klauzuli mohou zvyšovat zátěž, zejména pokud se provádějí na velkém množství dat. Optimalizace může zahrnovat předpočítání hodnot nebo použití efektivnějších algoritmů.

WHERE

- **Efektivní filtrování**: Klauzule `WHERE` je klíčová pro omezování datových sad, což zvyšuje efektivitu dotazů. Použití filtrů co nejdříve v dotazu může snížit množství dat zpracovávaných v následných operacích.
- **Indexy**: Indexování sloupců používaných ve `WHERE` podmínkách může výrazně zlepšit výkon dotazů, zejména při velkých datových sadách.

GROUP BY

- **Seskupování dat**: `GROUP BY` může být náročné na výkon, pokud se používá na velkých datových sadách. Doporučuje se použít filtraci dat před seskupením.
- **Agregační funkce**: Funkce jako `COUNT()`, `SUM()`, `AVG()` mohou být náročné na výkon. Optimalizace zahrnuje použití správných indexů a omezení počtu zpracovávaných řádků.

ORDER BY

- **Řazení dat**: Klauzule `ORDER BY` může vyžadovat velké množství paměti a CPU, zejména pokud není optimalizována nebo pokud je použita na velké datové sady. Doporučuje se minimalizovat počet řádků před řazením.
- **Indexy a řazení**: Použití indexů může zlepšit výkon při řazení dat, zejména pokud jsou řazena podle sloupců, které jsou indexované.

TOP

- **Omezení počtu výsledků**: Klauzule `TOP` se používá k omezení počtu vrácených řádků, což může zlepšit výkon dotazů tím, že zmenší množství dat přenášených a zpracovávaných. Použití `TOP` spolu s `ORDER BY` může být obzvláště užitečné pro získání největších nebo nejmenších hodnot v datové sadě.

Spojování dat z více tabulek

Spojování dat z více tabulek je klíčovým aspektem práce s relačními databázemi. Pomocí klauzule `JOIN` můžeme kombinovat řádky z dvou nebo více tabulek na základě vzájemně propojených sloupců, což umožňuje získat komplexní pohled na data.

Základní rozdělení klauzulí JOIN

SQL poskytuje několik typů `JOIN` klauzulí, které určují, jak budou záznamy z tabulek kombinovány:

3. **INNER JOIN**: Vrací pouze ty řádky, které mají odpovídající záznamy v obou tabulkách.
4. **LEFT JOIN (LEFT OUTER JOIN)**: Vrací všechny řádky z levé tabulky a odpovídající řádky z pravé tabulky. Pokud neexistuje odpovídající řádek, jsou hodnoty z pravé tabulky nahrazeny NULL.
5. **RIGHT JOIN (RIGHT OUTER JOIN)**: Vrací všechny řádky z pravé tabulky a odpovídající řádky z levé tabulky. Pokud neexistuje odpovídající řádek, jsou hodnoty z levé tabulky nahrazeny NULL.
6. **FULL JOIN (FULL OUTER JOIN)**: Vrací všechny řádky, když existuje shoda v jedné z tabulek. Pokud neexistuje shoda, vrátí NULL pro sloupce, kde neexistuje odpovídající záznam.

```
SELECT zeme,lokalita,AVG(srazky) prumerne_srazky
FROM data_nova
LEFT JOIN dim_oblasti ON data_nova.lokalita = dim_oblasti.stanice
WHERE zeme = 'Cesko'
GROUP by zeme,lokalita
```

Na co si dát pozor při použití JOIN

- **Podmínky spojení (Join Conditions)**: Ujistěte se, že jsou správně definovány podmínky spojení, aby se zabránilo chybnému spojování dat. Použití nesprávných nebo neúplných podmínek spojení může vést k nesprávným výsledkům.
- **Výkon dotazů**: Velké datové sady mohou zpomalit výkon dotazů, zejména pokud nejsou správně indexovány. Použití filtrů v `WHERE` klauzuli a indexování sloupců použitých v podmínkách spojení může zlepšit výkon.
- **Kardinální chyby**: Spojení tabulek na sloupcích s vysokou kardinálností (velký počet jedinečných hodnot) může vést k neočekávaným výsledkům, jako je násobení řádků. Je důležité zajistit, že podmínky spojení jsou vhodně definovány.

Spojování tabulek sami se sebou

Tento typ spojení se nazývá **self-join** a používá se, když potřebujeme porovnat záznamy v tabulce se sebou samými. Typicky se používá aliasování tabulky, aby bylo možné rozlišit mezi různými instancemi tabulky v dotazu.

UNION, UNION ALL, EXCEPT, INTERSECT

Tyto klauzule se používají pro kombinování výsledků z více dotazů.

- **UNION**: Kombinuje výsledky dvou dotazů a odstraní duplikáty. Pozor na výkon, zejména u velkých datových sad, protože odstranění duplikátů může být náročné.

```
SELECT * FROM data_nova
UNION
SELECT * FROM data_stara
```

- **UNION ALL**: Kombinuje výsledky dvou dotazů, včetně duplikátů. Je rychlejší než `UNION`, protože neodstraňuje duplikáty.

```
SELECT * FROM data_nova
UNION ALL
SELECT * FROM
```

- **EXCEPT**: Vrací rozdíl mezi dvěma dotazy. Vrací řádky z prvního dotazu, které nejsou ve druhém dotazu.

```
SELECT lokalita FROM data_stara
EXCEPT
SELECT lokalita FROM data_nova
```

- **INTERSECT**: Vrací průnik mezi dvěma dotazy, tedy pouze ty řádky, které jsou v obou dotazech.

```
SELECT lokalita FROM data_stara
INTERSECT
SELECT lokalita FROM data_nova
```

Na co si dát pozor při použití UNION, UNION ALL, EXCEPT, INTERSECT

- **Datové typy a struktura sloupců**: Ujistěte se, že všechny dotazy kombinované pomocí `UNION`, `UNION ALL`, `EXCEPT` nebo `INTERSECT` mají stejný počet sloupců a odpovídající datové typy, jinak dojde k chybě.
- **Pořadí výsledků**: Pokud je důležité pořadí výsledků, použijte `ORDER BY` na konci kombinovaného dotazu. Pamatujte, že `ORDER BY` se vztahuje na celý výsledek kombinovaného dotazu, nikoli na jednotlivé dotazy.

Výběr jedinečných dat z dvou tabulek

Kombinace klauzulí `UNION` a `SELECT DISTINCT` se často používá k získání jedinečných záznamů z dvou tabulek. `UNION` automaticky odstraní duplikáty, zatímco `UNION ALL` zobrazí všechny výsledky včetně duplikátů.

Pohledy (VIEW)

Úvod do VIEW

Pohledy (anglicky "views") jsou virtuální tabulky v SQL, které poskytují způsob, jak uložit složité dotazy nebo specifické pohledy na data v databázi. Pohled neobsahuje fyzicky data, ale slouží jako uložený dotaz, který lze použít stejně jako normální tabulku. Použití pohledů může zjednodušit složité dotazy, zlepšit bezpečnost a zvýšit opakovatelnost kódu.

Rozdíly mezi pohledy a klasickými tabulkami

- **Fyzické ukládání dat**:
  - **Tabulky**: Klasické tabulky v databázi fyzicky ukládají data. Každá tabulka má definovanou strukturu sloupců a obsahuje skutečná data, která jsou uložena na disku.
  - **Pohledy**: Pohledy neukládají data fyzicky. Místo toho ukládají pouze definici dotazu, který se provádí při každém přístupu k pohledu. Výsledky jsou generovány dynamicky na základě aktuálních dat v podkladových tabulkách.
- **Aktualizace dat**:
  - **Tabulky**: Data v tabulkách lze běžně aktualizovat, mazat nebo přidávat.
  - **Pohledy**: Aktualizace dat prostřednictvím pohledu závisí na struktuře pohledu a použitých funkcích. Některé pohledy, zejména ty s agregačními funkcemi nebo `JOIN` klauzulemi, mohou být jen pro čtení a nemusí podporovat operace `INSERT`, `UPDATE` nebo `DELETE`.
- **Bezpečnost a přístupová práva**:
  - **Tabulky**: Přístupová práva se nastavují na úrovni tabulky, což může znamenat, že uživatelé mají přístup ke všem datům v tabulce.
  - **Pohledy**: Pohledy mohou být použity k omezení přístupu na určité sloupce nebo řádky. Například, můžete vytvořit pohled, který zahrnuje pouze sloupce, ke kterým má uživatel přístup, a skrýt citlivá data.
- **Opakovatelnost a údržba kódu**:
  - **Tabulky**: Složité dotazy nebo výpočty musí být opakovány pokaždé, když jsou potřeba, což může vést k duplikaci kódu.
  - **Pohledy**: Složité dotazy lze uložit jako pohledy. Jakmile je pohled vytvořen, může být znovu použit bez nutnosti opakování složitých SQL dotazů, což usnadňuje údržbu.

Základy vytvoření a použití VIEW

Pohledy jsou vytvořeny pomocí příkazu `CREATE VIEW`. Struktura příkazu je následující:

```
CREATE VIEW data_nova_rocni_srazky AS
SELECT lokalita, zeme, YEAR(datum) AS rok, SUM(srazky) AS rocni_srazky
FROM data_nova
LEFT JOIN dim_oblasti ON data_nova.lokalita = dim_oblasti.zeme
GROUP BY lokalita,zeme, YEAR(datum)
```

Použití VIEW

Pohledy lze použít téměř stejným způsobem jako tabulky. Můžete je zahrnout do dotazů, použít je v JOIN klauzulích, nebo aplikovat další filtry.

```
SELECT * FROM data_nova_rocni_srazky
```

Úvod do pod dotazů

Pod dotazy, někdy nazývané vnořené dotazy, jsou dotazy vložené uvnitř jiného SQL dotazu. Slouží k provádění složitějších operací, které by nebylo možné snadno dosáhnout jedním dotazem. Poddotazy mohou být použity v různých částech SQL příkazu, jako jsou klauzule `SELECT`, `WHERE`, `FROM` nebo `HAVING`.

Typy pod dotazů

7. **Jednoduché poddotazy (Scalar Subqueries)**: Vrací jednu hodnotu, která může být použita jako výraz v jiném dotazu.
8. **Řádkové poddotazy (Row Subqueries)**: Vrací jeden nebo více řádků, ale vždy se stejným počtem sloupců, který může být porovnán s jiným řádkem v dotazu.
9. **Tabulkové poddotazy (Table Subqueries)**: Vrací celou tabulku dat, která může být použita ve `FROM` klauzuli jako zdroj dat pro vnější dotaz.

Použití poddotazů za pomocí odvozené tabulky

Odvozená tabulka je poddotaz uvedený v klauzuli `FROM` hlavního dotazu, který se chová jako dočasná tabulka. Tento poddotaz musí být vždy pojmenován (přidělen jí alias) a může být použit k provedení dalších operací v hlavním dotazu.

Charakteristiky:

- Pod dotaz je definován v klauzuli `FROM` hlavního dotazu.
- Výsledek pod dotazu se považuje za dočasnou tabulku.
- Dočasná tabulka musí mít alias.
- Používá se pro složitější dotazy, kde jsou výsledky pod dotazu zpracovány dále v hlavním dotazu.

```
SELECT dt.lokalita, dt.rok, dt.prumerne_srazky
FROM (
    SELECT lokalita, YEAR(datum) AS rok, AVG(srazky) AS prumerne_srazky
    FROM data_nova
    GROUP BY lokalita, YEAR(datum)
) AS dt
WHERE dt.prumerne_srazky > 10;
```

Použití poddotazů za pomocí příkazu `WITH` (CTE - Common Table Expressions)

`WITH` klauzule, také známá jako CTE (Common Table Expressions), poskytuje způsob, jak definovat dočasné výsledky, které mohou být referencovány v hlavním dotazu. CTE je deklarována před hlavním dotazem a může být použita v celém dotazu, což zlepšuje čitelnost a strukturu komplexních dotazů.

Charakteristiky:

- `WITH` klauzule je uvedena na začátku dotazu.
- Definuje dočasnou výsledkovou sadu, která může být použita v celém dotazu.
- CTE může zlepšit čitelnost a strukturu složitých dotazů.
- Umožňuje použití stejné dočasné výsledkové sady vícekrát v dotazu.

```
WITH tbl_prumerne_srazky AS (
    SELECT lokalita, YEAR(datum) AS rok, AVG(srazky) AS prumerne_srazky
    FROM data_nova
    GROUP BY lokalita, YEAR(datum)
)
SELECT lokalita, rok, prumerne_srazky
FROM tbl_prumerne_srazky
WHERE prumerne_srazky > 10;
```

Použití poddotazů

- **V klauzuli SELECT**: Poddotaz může být použit k výpočtu hodnoty pro každý řádek ve výsledné sadě.

```
SELECT lokalita, srazky,
       (SELECT AVG(srazky) FROM data_nova) AS prumerne_srazky_celkem
FROM data_nova;
```

- **V klauzuli WHERE**: Použití poddotazu v `WHERE` klauzuli umožňuje filtrovat data na základě podmínky, která závisí na výsledku poddotazu. Tímto způsobem lze například vrátit všechny zaměstnance, kteří mají vyšší plat než je průměrný plat všech zaměstnanců.

```
SELECT lokalita, srazky
FROM data_nova
WHERE srazky >
    (SELECT AVG(srazky)
     FROM data_nova);
```

- **V klauzuli FROM**: Poddotazy mohou být použity k vytvoření virtuálních tabulek, které jsou dále zpracovávány vnějšími dotazy.

```
SELECT dt.lokalita, dt.rok, dt.prumerne_srazky, d.zeme
FROM (
    SELECT lokalita, YEAR(datum) AS rok, AVG(srazky) AS prumerne_srazky
    FROM data_nova
    GROUP BY lokalita, YEAR(datum)
) AS dt
JOIN dim_oblasti AS d ON dt.lokalita = d.stanice
ORDER BY dt.rok, dt.prumerne_srazky DESC;
```

- **V klauzuli HAVING**: Poddotazy mohou být použity k filtrování agregovaných výsledků.

```
SELECT lokalita, YEAR(datum) AS rok, AVG(srazky) AS prumerne_srazky
FROM data_nova
GROUP BY lokalita, YEAR(datum)
HAVING AVG(srazky) > (
    SELECT AVG(srazky)
    FROM data_nova
);
```

Využití a výhody poddotazů

- **Modularita**: Poddotazy umožňují rozdělit složité dotazy na menší, snadno spravovatelné části.
- **Flexibilita**: Umožňují použití výsledků z jednoho dotazu v jiných částech dotazu, což může být velmi užitečné při provádění komplexních výpočtů nebo filtrování dat.
- **Zjednodušení kódu**: Poddotazy mohou snížit potřebu opakování podobné logiky v různých částech dotazu, což zjednodušuje údržbu a čitelnost kódu.

Poddotazy jsou mocným nástrojem v SQL, který umožňuje vytvářet složité dotazy a provádět pokročilé operace nad daty.

Použití operátorů IN, ALL, ANY a EXISTS

V SQL se operátory `IN`, `ALL`, `ANY` a `EXISTS` používají k filtrování dat na základě podmínek v poddotazech nebo sadách hodnot. Tyto operátory umožňují provádět srovnání a vyhledávání nad datovými sadami, což poskytuje flexibilitu při vytváření komplexních dotazů.

Operátor IN

Operátor `IN` se používá k porovnání hodnoty s množinou hodnot nebo výsledky poddotazu. Vrací `TRUE`, pokud hodnota odpovídá kterékoli hodnotě v seznamu nebo množině. Je užitečný pro kontrolu, zda hodnota existuje v předdefinované množině, čímž zjednodušuje psaní podmínek ve srovnání s použitím více podmínek s operátorem `OR`.

```
SELECT YEAR(datum) rok, lokalita, AVG(srazky) AS prumerne_srazky
FROM data_nova
WHERE YEAR(datum) IN (
    SELECT YEAR(datum)
    FROM data_nova
    WHERE srazky < (SELECT AVG(srazky) FROM data_nova)
    GROUP BY YEAR(datum)
)
GROUP BY lokalita, YEAR(datum)
ORDER BY prumerne_srazky DESC;
```

Operátor ALL

Operátor `ALL` se používá k porovnání hodnoty s každou hodnotou v množině nebo poddotazu. Vrací `TRUE`, pokud podmínka platí pro všechny hodnoty v množině. To znamená, že je splněna pouze tehdy, pokud je daná podmínka pravdivá pro všechny hodnoty ve výsledné množině.

```
SELECT lokalita, srazky
FROM data_nova
WHERE srazky > ALL (
    SELECT srazky
    FROM data_nova
    WHERE YEAR(datum) = 2020
);
```

Operátor ANY

Operátor `ANY` se používá k porovnání hodnoty s jakoukoli hodnotou v množině nebo poddotazu. Vrací `TRUE`, pokud podmínka platí pro alespoň jednu hodnotu v množině. Je užitečný, když chcete zjistit, zda alespoň jedna hodnota v množině splňuje určitou podmínku.

```
SELECT lokalita, srazky
FROM data_nova
WHERE srazky > ANY (
    SELECT srazky
    FROM data_nova
    WHERE lokalita <> data_nova.lokalita
);
```

Operátor EXISTS

Operátor `EXISTS` se používá k ověření existence řádků vrácených poddotazem. Vrací `TRUE`, pokud poddotaz vrátí alespoň jeden řádek, což je užitečné pro kontrolu existence dat bez nutnosti vracení samotných dat. `EXISTS` se často používá v poddotazech v kombinaci s klauzulí `WHERE` k filtrování výsledků na základě přítomnosti určitých dat v jiné tabulce.

```
SELECT lokalita, srazky
FROM data_nova dn
WHERE EXISTS (
    SELECT 1
    FROM dim_oblasti
    WHERE dim_oblasti.stanice = dn.lokalita AND dim_oblasti.zeme = 'Cesko'
);
```

Korelované vs. Nekorelované poddotazy a jejich zatížení serveru

Poddotazy v SQL lze rozdělit na korelované a nekorelované poddotazy. Tyto dvě kategorie se liší svým způsobem vykonávání a mohou mít různý dopad na výkon serveru.

Nekorelované poddotazy

Nekorelované poddotazy jsou poddotazy, které jsou nezávislé na hlavním dotazu. To znamená, že poddotaz může být vykonán jednou a jeho výsledek může být použit pro celý hlavní dotaz. Nekorelované poddotazy jsou jednodušší a obvykle mají menší dopad na výkon serveru.

```
SELECT lokalita, srazky
FROM data_nova
WHERE srazky > (SELECT AVG(srazky) FROM data_nova);
```

Vlastnosti nekorelovaných poddotazů:

- Nezávislé na hlavním dotazu.
- Může být vykonán jednou a výsledek použit v hlavním dotazu.
- Obvykle rychlejší a méně náročné na výkon.

Korelované poddotazy

Korelované poddotazy jsou poddotazy, které závisí na každém řádku hlavního dotazu. To znamená, že poddotaz musí být vykonán pro každý řádek zvlášť, což může výrazně zvýšit zatížení serveru.

```
SELECT lokalita, datum, srazky
FROM data_nova AS dn1
WHERE srazky > (
    SELECT AVG(srazky)
    FROM data_nova AS dn2
    WHERE dn2.lokalita = dn1.lokalita
);
```

Vlastnosti korelovaných poddotazů:

- Závislé na hlavním dotazu.
- Musí být vykonány pro každý řádek hlavního dotazu zvlášť.
- Obvykle pomalejší a náročnější na výkon.

Příklad rozdílu v zatížení serveru

Nekorelované poddotazy vykonají poddotaz jednou a použijí jeho výsledek pro celý hlavní dotaz. To znamená, že množství vykonání poddotazu je konstantní, nezávisle na počtu řádků v hlavním dotazu.

Korelované poddotazy musí vykonat poddotaz pro každý jednotlivý řádek hlavního dotazu. To znamená, že množství vykonání poddotazu roste s počtem řádků v hlavním dotazu, což může výrazně zvýšit zatížení serveru, zejména u velkých datových sad.

Optimalizace výkonu

- **Použití nekorelovaných poddotazů**: Kdykoli je to možné, používejte nekorelované poddotazy, protože jsou obvykle rychlejší a méně náročné na výkon.
- **Indexování**: Zajistěte, že sloupce používané v podmínkách poddotazů jsou správně indexovány, což může zlepšit výkon dotazu.
- **Zvažování alternativních přístupů**: Někdy lze korelované poddotazy nahradit jinými metodami, jako jsou spojení (`JOIN`), která mohou být efektivnější.

Příkazy CREATE, ALTER, DROP

V SQL se příkazy `CREATE`, `ALTER` a `DROP` používají pro správu struktury databáze, zejména tabulek, indexů, klíčů a dalších objektů. Tyto příkazy umožňují vytvářet, měnit a odstraňovat databázové objekty.

Příkaz CREATE

Příkaz `CREATE` se používá k vytváření nových databázových objektů, jako jsou tabulky, pohledy, indexy a další.

```
CREATE TABLE prumerne_srazky_cesko (
    lokalita NVARCHAR(50) PRIMARY KEY,
    prumerne_srazky FLOAT
);
```

Základní vytváření tabulek

Při vytváření tabulky je nutné definovat název tabulky a jednotlivé sloupce včetně jejich datových typů a případných omezení, jako jsou primární klíče nebo unikátní omezení. Každý sloupec musí být definován s názvem a typem dat, který určuje, jaký typ hodnot může obsahovat. Mohou být také nastavena další omezení, jako je `NOT NULL`, což znamená, že sloupec nesmí obsahovat žádné prázdné hodnoty.

Příkaz ALTER

Příkaz `ALTER` se používá k modifikaci existujících databázových objektů. Tento příkaz umožňuje přidávat nové sloupce, měnit datové typy sloupců, přidávat nebo odstraňovat omezení a provádět další změny ve struktuře tabulek.

Přidání sloupce

Přidání nového sloupce do existující tabulky je běžnou operací, kdy je potřeba rozšířit strukturu tabulky o další informace. Nový sloupec může mít specifikovaný datový typ a další omezení, jako například výchozí hodnotu nebo `NOT NULL` omezení, které zajišťuje, že sloupec nemůže obsahovat prázdné hodnoty.

Změna datového typu sloupce

Pokud je nutné změnit datový typ stávajícího sloupce, lze použít příkaz `ALTER` k aktualizaci definice sloupce. Tato operace může být nezbytná například při změně

obchodních požadavků nebo při optimalizaci výkonu. Při změně datového typu je důležité zajistit, že nové datové hodnoty jsou kompatibilní s novým typem.

Přidání nebo odstranění omezení

Příkaz ALTER umožňuje také přidávat nebo odstraňovat omezení (constraints) na tabulky. To zahrnuje:

- **Primární klíče**: Zajišťují jedinečnost záznamů v tabulce.
- **Unikátní klíče**: Zajišťují, že hodnoty v jednom nebo více sloupcích jsou jedinečné v rámci celé tabulky.
- **Cizí klíče**: Udržují referenční integritu mezi tabulkami, což zajišťuje, že hodnoty v cizím klíči odpovídají hodnotám v primárním klíči jiné tabulky.
- **Kontrolní omezení (CHECK)**: Omezují hodnoty, které mohou být do sloupce vloženy, na základě definované podmínky.

```
ALTER TABLE prumerne_srazky_cesko
ADD okres NVARCHAR(50);
```

Příkaz DROP

Příkaz DROP se používá k odstranění databázových objektů, jako jsou tabulky, pohledy nebo indexy. Tento příkaz je nevratný, což znamená, že všechny data a struktury budou trvale odstraněny.

Odstranění tabulky

Odstranění celé tabulky z databáze je operace, která zcela smaže strukturu tabulky a všechna data, která obsahuje. To je užitečné, pokud tabulka již není potřebná nebo je nutné provést zásadní změny struktury, které nelze provést prostřednictvím příkazu ALTER.

```
DROP TABLE prumerne_srazky_cesko
```

Odstranění sloupce

Pokud určitý sloupec v tabulce již není potřebný, může být odstraněn pomocí příkazu ALTER a následného odstranění sloupce. Tato operace může být nevratná, pokud neexistují záložní kopie dat, a proto by měla být provedena s opatrností.

```
ALTER TABLE prumerne_srazky_cesko
DROP COLUMN source;
```

Příkazy INSERT, UPDATE, DELETE a TRUNCATE

V SQL se příkazy `INSERT`, `UPDATE`, `DELETE` a `TRUNCATE` používají k manipulaci s daty v tabulkách. Tyto příkazy umožňují přidávat, měnit a odstraňovat záznamy v databázi, což je zásadní pro správu a údržbu dat.

Příkaz INSERT

Příkaz `INSERT` se používá k přidávání nových záznamů do tabulky.

Základní syntaxe

Základní syntaxe `INSERT` zahrnuje specifikaci tabulky a hodnot pro vložení do odpovídajících sloupců. Můžete vložit jeden nebo více záznamů najednou, přičemž každý záznam musí obsahovat hodnoty pro všechny sloupce, které nemají nastavenou výchozí hodnotu nebo povolenou hodnotu NULL.

```
INSERT INTO data_nova_cesko (lokalita,datum-srazky) VALUES('Praha','2022-08
-01',12.5)
```

Příkaz UPDATE

Příkaz `UPDATE` se používá k úpravě existujících záznamů v tabulce. Tento příkaz umožňuje aktualizovat hodnoty jednoho nebo více sloupců pro záznamy, které splňují určité podmínky.

Základní syntaxe

Základní syntaxe `UPDATE` zahrnuje specifikaci tabulky, sloupců k aktualizaci a nové hodnoty pro tyto sloupce. Klauzule `WHERE` se používá k určení, které záznamy mají být aktualizovány.

```
UPDATE data_nova_cesko
SET srazky = 14.0
WHERE lokalita = 'Praha' AND datum = '2022-08-01';
```

Příkaz DELETE

Příkaz `DELETE` se používá k odstranění záznamů z tabulky.

Základní syntaxe

Základní syntaxe `DELETE` zahrnuje specifikaci tabulky a klauzule `WHERE`, která určuje, které záznamy mají být odstraněny. Bez klauzule `WHERE` by došlo k odstranění všech záznamů v tabulce, což je obvykle nechtěný výsledek.

```
DELETE FROM data_nova_cesko
WHERE lokalita IN (
    SELECT stanice
    FROM dim_oblasti
    WHERE zeme = 'Cesko'
);
```

Příkaz TRUNCATE

Příkaz `TRUNCATE` se používá k rychlému odstranění všech záznamů z tabulky, ale na rozdíl od `DELETE` neumožňuje použití klauzule `WHERE`. Tento příkaz je efektivnější než `DELETE` bez `WHERE` klauzule, protože neprovádí jednotlivé operace mazání pro každý řádek.

Základní syntaxe

`TRUNCATE` se používá k rychlému vyprázdnění tabulky, což resetuje identifikátory a rychleji uvolňuje paměť. Je však důležité si uvědomit, že `TRUNCATE` je nevratný a nelze jej použít, pokud tabulka obsahuje cizí klíče odkazující na jiné tabulky.

```
TRUNCATE TABLE data_nova_cesko;
```

Hromadné zpracování dat jedním dotazem

Hromadné zpracování dat je klíčové pro efektivní manipulaci s velkým množstvím dat. SQL umožňuje provádět hromadné operace, jako je vložení, aktualizace nebo odstranění více záznamů najednou. To lze dosáhnout pomocí:

- **Vícenásobného vkládání**: Vložením více záznamů jedním příkazem `INSERT`.
- **Hromadná aktualizace**: Aktualizací více záznamů jedním příkazem `UPDATE` s použitím klauzule `WHERE` nebo pomocí složitých podmínek.
- **Hromadné odstranění**: Odstraněním více záznamů jedním příkazem `DELETE` s klauzulí `WHERE`.

Použití těchto technik zvyšuje výkon a efektivitu databázových operací, protože snižuje počet potřebných transakcí a s tím související režii.