Vysoké učení technické v Brně
Fakulta informačních technologií

Databázové systémy

2021/2022

Projektová dokumentace

Projekt č. 6 - Pekárna

Vladyslav Kovalets (xkoval21)

Evgeniya Taipova (xtaipo00)

Brno, 1. květen 2022

Zadání

Navrhněte jednoduchý informační systém pekárny, který bude spravovat informace o nabízených druzích pečiva jak z hlediska výroby (suroviny, náklady at.), tak odbytu (objednávky). Pekárna zajišťuje rozvoz vlastními auty nebo si odběratelé odvoz zajišťují sami. Systém musí umožnit vedení pekárny plánovat produkci v závislosti na objednávkách, poskytovat informace pro rozvoz apod.

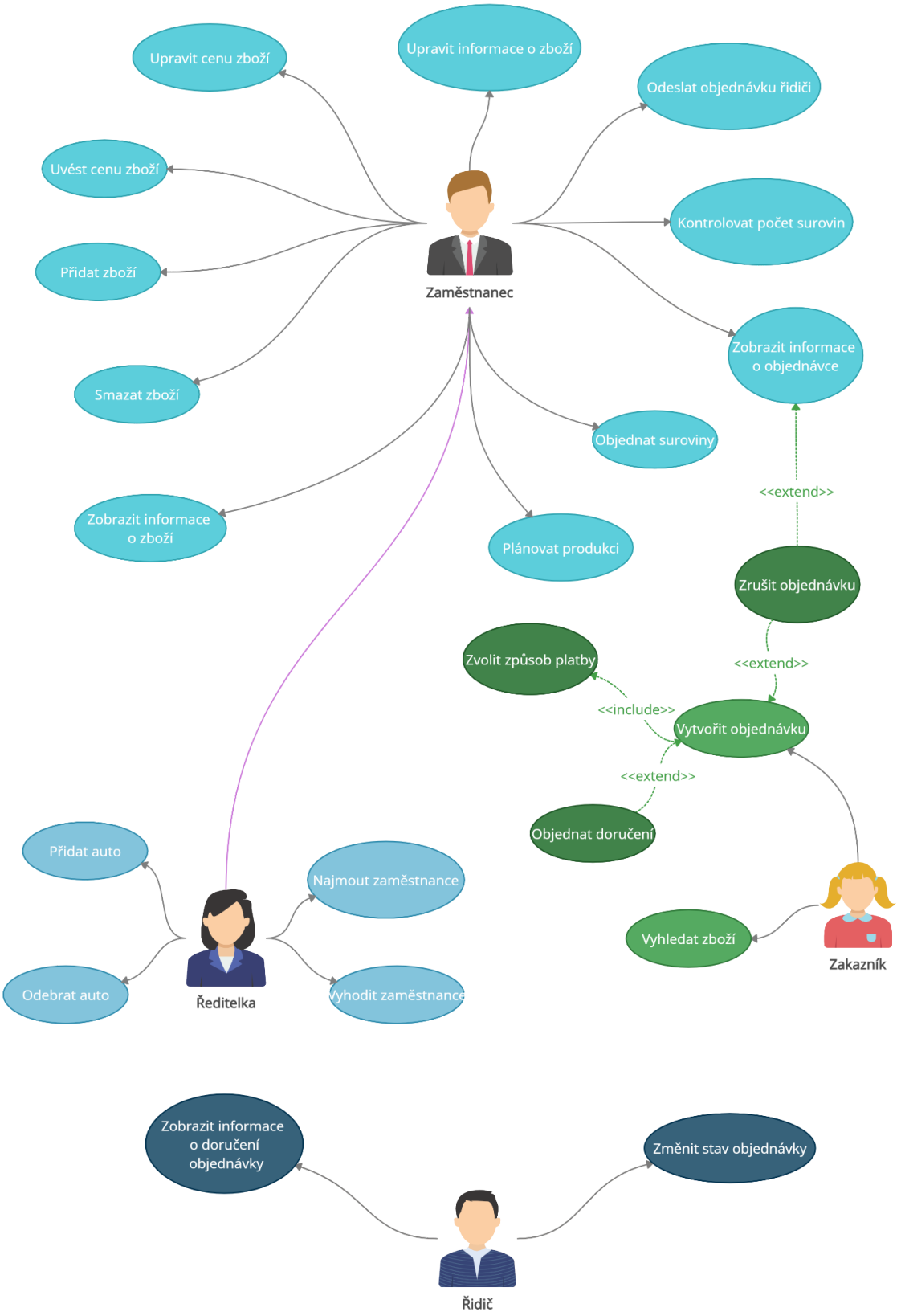
Úvod

Projekt jsme zahájili vytvořením modelu případů užití a ER diagramu. V diagramu jsme udělali malou chybu, kterou jsme později opravili. Každá fáze projektů byla vypracována společně. Všechny vzniklé problémy byly řešeny pomocí demonstračních cvičení.

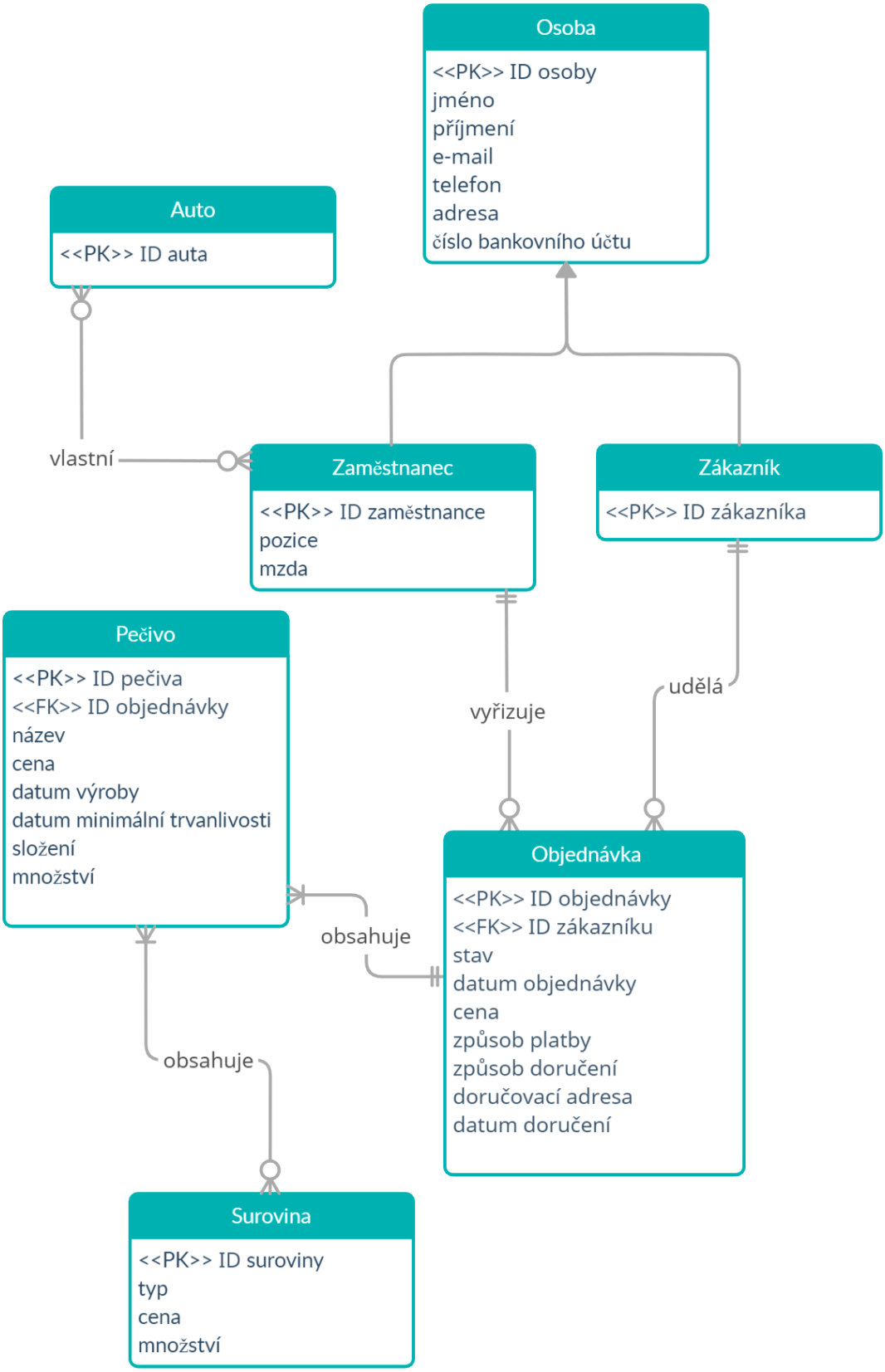
První část

Popis datového modelu

ER diagram se skládá z několika entit. Rozhodli jsme se použít generalizaci/specializaci, proto máme dva typy, které se dědí z tabulky `Osoba`: `Zakaznik`, který vytváří objednávky a `Zamestnanec`, který objednávky zpracovává. Entita `Objednavka` obsahuje informace o zákazníkovi a pečivu, které si objednal, jakož i o způsobu dodání. Také jsme vyrobili entitu `Auto`, kterou může vlastnit `Zamestnanec` pro doručování objednávek. Entita `Pecivo` se používá k popisu produktu, který bude použit pro objednávku. Má takové základní atributy jako je název, cena, datum výroby a další. A každé `Pecivo` obsahuje entitu `Surovina`, která hlavně určuje typ a množství.



Model případů užití


Osoba
<<PK>> ID osoby
jméno
příjmení
e-mail
telefon
adresa
číslo bankovního účtu
Auto
<<PK>> ID auta
vlastní
Zaměstnanec
<<PK>> ID zaměstnance
pozice
mzda
Zákazník
<<PK>> ID zákazníka
Pečivo
<<PK>> ID pečiva
<<FK>> ID objednávky
název
cena
datum výroby
datum minimální trvanlivosti
složení
množství
vyřizuje
udělá
obsahuje
obsahuje
Objednávka
<<PK>> ID objednávky
<<FK>> ID zákazníku
stav
datum objednávky
cena
způsob platby
způsob doručení
doručovací adresa
datum doručení
Surovina
<<PK>> ID suroviny
typ
cena
množství

ER diagram

Druhá část

Podle ER diagramu jsme pomocí funkce `CREATE TABLE` vytvořili několik tabulek: `Osoba`, `Zamestnanec`, `Zakaznik`, `Auto`, `Surovina`, `Pecivo`, `Objednavka`. V tabulce `Osoba` máme vytvořené ověření e-mailu a čísla bankovního účtu pomocí `CHECK(REGEXP_LIKE)`. Dále jsme přidali primární a cizí klíče a naplnili tabulky daty, které jsme použili později.

Třetí část

V této části jsme vytvořili SELECT dotazy. Příkaz `SELECT` se používá k získání výběru bez podmínek nebo s podmínkami z databázových tabulek.

Spojení dvou tabulek

Použili jsme *`NATURAL JOIN`* ke spojení některých datových tabulek. Spojovací podmínka realizována automaticky dle přítomnosti referenčních vazeb nebo shodnosti názvů sloupců a datových typů ve spojovaných tabulkách.

Vypíše údaje kurýrů:

```
SELECT * FROM Osoba NATURAL JOIN Zamestnanec
WHERE Zamestnanec.pozice = 'Kurýr';
```

Spojení tří tabulek

Dále jsme se spojili pomocí podmínek *`WHERE`*.

Vypíše objednávku a jméno, příjmení zákazníka:

```
SELECT Objednavka.id_objednavky, Osoba.jmeno, Osoba.prijmeni
FROM Objednavka, Zakaznik, Osoba
WHERE Objednavka.id_zakaznika = Zakaznik.id_zakaznika
AND Zakaznik.id_osoby = Osoba.id_osoby;
```

Agregační funkce a GROUP BY

Agregační funkce nám umožnily seskupit vybrané řádky dotazu a provádět na nich výpočty. Používají s klauzulí `GROUP BY`.

Vypíše id zákazníka a počet jeho objednávek:

```
SELECT Zakaznik.id_zakaznika, COUNT(*) pocet_objednavek
FROM Zakaznik, Objednavka
WHERE Objednavka.id_zakaznika = Zakaznik.id_zakaznika
GROUP BY Zakaznik.id_zakaznika;
```

Obsahující predikát EXISTS

Řekněme tedy, že potřebujeme zjistit, kteří zákazníci si objednali konkrétní produkt. Musíme přidat podmínku pomocí predikátu `EXISTS`, že daný zákazník vůbec nějakou objednávku učinil. Poté už získáme opravdu zákazníka, jehož všechny objednávky splňují danou podmínku.

Vypíše zákazníky a objednávky, které obsahují rohlik:

```
SELECT Objednavka.id_objednavky, Objednavka.id_zakaznika
FROM Objednavka WHERE EXISTS(SELECT * FROM Pecivo
WHERE id_objednavky = Objednavka.id_objednavky
AND nazev = 'Rohlik');
```

Čtvrtá část

TRIGGER

V této části projektu jsme vytvořili dva triggery. Trigger se spustí automaticky při pokusu o změnu dat v tabulce, ke které je přidružen.

První trigger `zkontroluj_data_peciva` kontroluje, zda jsou zadaná data výroby a data minimální trvanlivosti správná, a také datum spotřeby pro konkrétní pečiva. Jako příklad jsme použili `Rohlík`, u kterého by měl být rozdíl mezi datem výroby nebo datem spotřeby dva dny. Datum výroby musí být před datem spotřeby. Pokud není při přidávání data splněna podmínka, dojde k chybě. Pro testování jsme použili

```
INSERT INTO Pecivo (id_objednavky, nazev, cena,
datum_vyroby, datum_minimalni_trvanlivosti, slozeni,
mnozstvi)
VALUES (3, 'Bageta malá světlá', 3.90, '28-03-2023',
'30-03-2022', 'Mouka, voda, olej, droždí, sůl', 4);
```

V tomto příkladu dojde k chybě, protože datum výroby je pozdější než datum spotřeby.

Druhý trigger `informace_mzda` zobrazuje informace o mzdě zaměstnance. Volá se kdykoliv, když dojde ke změně dat. Při odebrání zaměstnance se vydá stará mzda.

```
WHEN DELETING THEN
dbms_output.put_line('Stará mzda: ' || :OLD.mzda);
```

Při přidání nového zaměstnance se vydá nová mzda.

```
WHEN INSERTING THEN
dbms_output.put_line('Nová mzda: ' || :NEW.mzda);
```

Při změně mzdy zaměstnance se zobrazí stará mzda, nová mzda a rozdíl (a také procentuální) mezi nimi.

```
WHEN UPDATING THEN
rozdil := :NEW.mzda - :OLD.mzda;
procentualni_rozdil := (:NEW.mzda / :OLD.mzda -1) * 100;
dbms_output.put_line('Stará mzda: ' || :OLD.mzda);
dbms_output.put_line('Nová mzda: ' || :NEW.mzda);
dbms_output.put_line('Rozdil: ' || rozdil || ' (' ||
ROUND(procentualni_rozdil, 2) || '%)');
```

PROCEDURE

Dále skript obsahuje dvě procedury : `kupni_sila_zakaznika` a `pozadovane_pecivo`.

Procedura `kupni_sila_zakaznika` obdrží jako vstup ID zákazníka a vypíše informace o něm a jeho dokončených objednávkách. K vytvoření proměnných jsme použili `%TYPE`, který nám umožňuje deklarovat proměnnou, aby měla stejný datový typ jako dříve deklarovaná proměnná nebo sloupec.

Také v této proceduře používáme kurzor, který je vytvořen pomocí této struktury

```
CURSOR cursor_name IS select_statement;
```

V našem případě je `cursor_name` `kursor` a `select_statement` je

```
SELECT Objednavka.cena, Zakaznik.id_zakaznika, Osoba.jmeno,
Osoba.prijmeni, Osoba.id_osoby, Zakaznik.id_osoby,
Osoba.telefon, Osoba.email
FROM Objednavka, Zakaznik, Osoba
WHERE zakaznik_id = Zakaznik.id_zakaznika
AND Osoba.id_osoby = Zakaznik.id_osoby
AND Zakaznik.id_zakaznika = Objednavka.id_zakaznika
AND Objednavka.stav IN('Doručeno');
```

Načítání kurzoru zahrnuje přístup k jednomu řádku najednou. Načítáme řádky z kurzoru následovně

```
FETCH kursor INTO cena_objednavky, id_zakaznika, jmeno,
prijmeni, id_osoby, id_osoby_zak, telefon, email;
```

Pokud má zákazník nulové objednávky, pak přejde na výjimku.

Procedura `pozadovane_pecivo` vypíše název a množství pečiva, které je potřeba připravit pro objednávky (berou se v úvahu pouze objednávky ve stavu 'Zpracovává se'). Pro test zavoláme proceduru s `CALL pozadovane_pecivo`. Očekávaný výstup:

```
Potřebujeme připravit 1 Brioška máslová
Potřebujeme připravit 4 Rohlik
```

Dále k objednávce číslo tři přidáme `'Bageta malá světlá'` a pak očekávaný výstup bude:

```
Potřebujeme připravit 1 Brioška máslová
Potřebujeme připravit 4 Rohlik
Potřebujeme připravit 10 Bageta malá světlá
```

Přístupové práva

Přístupová práva jsou udělena příkazem GRANT. Pro přístup k tabulce nebo proceduře potřebuje uživatel nebo objekt práva SELECT, INSERT, UPDATE, DELETE nebo REFERENCES. Všechna práva lze udělit pomocí možnosti ALL, kterou jsme použili. V našem případě dáváme práva `XTAIPO00`. Příklad plného přístupu k tabulce `Objednavka` :

```
GRANT ALL ON Objednavka TO XTAIPO00;
```

Pro procedury používáme `GRANT EXECUTE ON nazev_procedury TO XTAIPO00;`

Materializovaný pohled

Dalším úkolem bylo vytvořit materializovaný pohled, který umožňuje provést SQL dotaz v určitém okamžiku a uložit výsledek do tabulky. Pomocí našeho pohledu můžeme vidět informace o zaměstnanci (jeho jméno a příjmení, ID, pozici a mzdu). A poté se nám ukáže pomocí `SELECT * FROM zamestnanec_info`, jak pohled funguje před a po přidání nových dat (nebylo nic přidáno).

EXPLAIN PLAN

Příkaz `EXPLAIN PLAN` je velmi užitečný. Když provádíme dotaz, SQL se snaží navrhnout nejlepší plán provedení pro tento dotaz. A toto nám pomáhá pochopit, jak jsou požadavky zpracovávány a jak je lze optimalizovat.

Tento příkaz vypíše id zákazníka, jeho jméno a příjmení, počet objednávek a celkovou cenu těchto objednávek:

```
SELECT Zakaznik.id_zakaznika, Osoba.jmeno, Osoba.prijmeni,
COUNT(*) pocet_objednavek, SUM(Objednavka.cena) celkova_cena
FROM Zakaznik, Osoba, Objednavka
WHERE Objednavka.id_zakaznika = Zakaznik.id_zakaznika
AND Osoba.id_osoby = Zakaznik.id_osoby
GROUP BY Zakaznik.id_zakaznika, Osoba.jmeno, Osoba.prijmeni;
```

Lze použít pro propagační akce. Například: po každých utracených 5000 korunách dostanete kupón na 500 korun.

Pokud na to použijeme příkaz `EXPLAIN PLAN FOR` a pomocí `SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY)` to vypíšeme, uvidíme tabulku s posloupností spustitelných operací a jejich výkonnostní cenou.

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		3	192	10 (20)	00:00:01
1	NESTED LOOPS		3	192	10 (20)	00:00:01
2	NESTED LOOPS		3	192	10 (20)	00:00:01
3	MERGE JOIN		3	135	7 (29)	00:00:01
4	TABLE ACCESS BY INDEX ROWID	ZAKAZNIK	4	24	2 (0)	00:00:01
5	INDEX FULL SCAN	PK_ZAKAZNIK	4		1 (0)	00:00:01
* 6	SORT JOIN		3	117	5 (40)	00:00:01
7	VIEW	VW_GBC_10	3	117	4 (25)	00:00:01
8	HASH GROUP BY		3	21	4 (25)	00:00:01
9	TABLE ACCESS FULL	OBJEDNAVKA	5	35	3 (0)	00:00:01
* 10	INDEX UNIQUE SCAN	PK_OSOBY	1		0 (0)	00:00:01
11	TABLE ACCESS BY INDEX ROWID	OSOBA	1	19	1 (0)	00:00:01

INDEX

Používání indexů může být užitečné, pokud potřebujete často prohledávat tabulku. Vyžadují však další paměť, a pokud tabulku často opravujete, bude to trvat déle, protože indexy bude nutné pokaždé aktualizovat.

Protože často používáme tabulky `Osoba` a `Objednavka`, bylo rozhodnuto vytvořit pro ně indexy takto:

```
CREATE INDEX osoba_info ON Osoba (id_osoby, jmeno,
prijmeni);
CREATE INDEX objednavka_info ON Objednavka(id_zakaznika,
cena);
```

Aplikujme `EXPLAIN FOR` na stejný dotaz jako předtím.

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		3	192	5 (20)	00:00:01
* 1	HASH JOIN		3	192	5 (20)	00:00:01
2	MERGE JOIN		3	135	4 (25)	00:00:01
3	TABLE ACCESS BY INDEX ROWID	ZAKAZNIK	4	24	2 (0)	00:00:01
4	INDEX FULL SCAN	PK_ZAKAZNIK	4		1 (0)	00:00:01
* 5	SORT JOIN		3	117	2 (50)	00:00:01
6	VIEW	VW_GBC_10	3	117	1 (0)	00:00:01
7	HASH GROUP BY		3	21	1 (0)	00:00:01
8	INDEX FULL SCAN	OBJEDNAVKA_INFO	5	35	1 (0)	00:00:01
9	INDEX FULL SCAN	OSOBA_INFO	8	152	1 (0)	00:00:01

Po přidání indexů byl SQL schopen najít odpovídající záznamy ve všech tabulkách pomocí indexů a neuchýlil se k úplnému prohledání tabulky. Výsledkem je snížení režie CPU, což vede k výraznému zvýšení výkonu, a proto operace byla dokončena rychleji.

Můžeme také optimalizovat dotazy pomocí jednoduchých pravidel:

1. Vyhněte se `SELECT DISTINCT`. To je praktický způsob, jak odstranit duplikáty z dotazu. K dosažení tohoto cíle je však zapotřebí velké množství výpočetního výkonu. Chcete-li se vyhnout použití `SELECT DISTINCT`, vyberte více polí a vytvořte jedinečné výsledky.
2. Používejte pole SELECT místo použití SELECT *.

Závěr

Jak bylo uvedeno na začátku, veškerou práci jsme dělali společně a bylo to skvělé rozhodnutí, protože oba rozumíme tomu, proč a jak používat příkazy. Všechny vzniklé problémy byly řešeny pomocí demonstračních cvičení a prostřednictvím přednáškových prezentací. S nedostatkem času na projekt jsme neměli problémy.