FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Databázové systémy – IDS

Dokumentácia projektu databázy pre informačný systém futbalového klubu

Tomáš Hrúz (xhruzt00)
Lukaš Tkáč (xtkacl00)

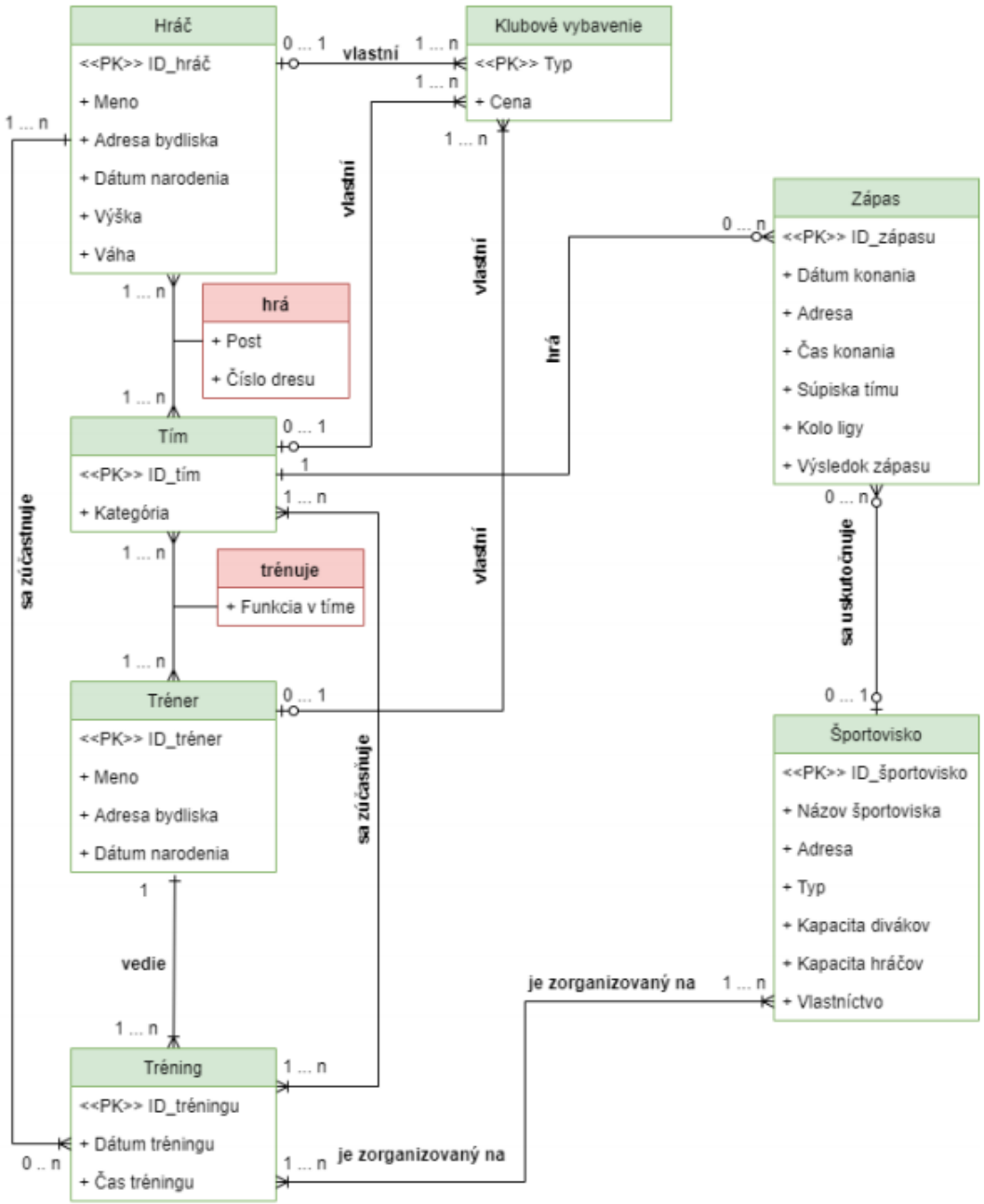
28.4.2021

Úvod

Pre projekt do predmetu IDS sme si vybrali ako základ Model informačného systému pre Futbalový klub vypracovaný členom tímu Lukášom Tkáčom. V pôvodnom riešení sme museli spraviť menšie úpravy aby bolo možné implementovať všetky požiadavky databázy, ale inak sa projekt zaobišiel bez komplikácii. Pracovali sme na ňom v priebehu celého semestra postupne ako sme zbierali informácie o databázach a myslíme si, že jeho rozdelenie a náročnosť boli primerané našim schopnostiam a zároveň nás naučil základnú prácu s databázami.

Prvá časť projektu

Pôvodný návrh z predmetu IUS:



Druhá časť projektu

V druhej časti bolo našou úlohou vytvoriť SQL skript, ktorý na základe ER diagramu vytvára tabuľky a naplní ich vzorovými dátami. Vytvorené tabuľky mali zmeny oproti pôvodnému návrhu. Jednou z týchto zmien bolo napríklad zmena samostatných atribútov "čas" a "dátum" na jeden spoločný s typom `TIMESTAMP`, ktorý zahŕňal obidve informácie. Ďalej boli pridané väzobné tabulky: "HracHraRelation", "TeamHralRelation", "TeamTreningRelation", TreningSportoviskoRelation, Trenuje. Následne sme pridávali primárne a cudzie kľúče. Primárne kľúče boli vo väčšine prípadov ID a v prípade väzobných tabuliek to bolo zloženie dvoch ID. Tieto ID sa používali ako cudzie kľúče pre spájané tabuľky.
Napríklad:

Cudzie kľúče v tabuľke "Trenuje" odkazujúce na PK v tabuľke "Team" a PK v tabuľke "Trener"

```
ALTER TABLE Trenuje
ADD CONSTRAINT FK_id_team_rel FOREIGN KEY (id_team_rel) REFERENCES Team ON
DELETE CASCADE;

ALTER TABLE Trenuje
ADD CONSTRAINT FK_id_trener_rel FOREIGN KEY (id_trener_rel) REFERENCES Trener
ON DELETE CASCADE;
```

Poslednou úlohou v druhej časti bolo naplniť tabuľky vzorovými dátami. Tu sme názorne ukázali, že niektoré atribúty nie sú povinné, takže sme ich vkladať nemuseli.
V prípade klubového vybavenia to bol napríklad atribút "cena", ktorý nebol povinný ale typ povinný bol.

```
INSERT INTO Klubove_vybavenie (typ, cena) VALUES ('Lopta adeedas', 250);
INSERT INTO Klubove_vybavenie (typ)       VALUES ('Lopta pooma');
```

Tretia časť projektu

V tretej časti bolo potrebné vytvoriť dotazy SELECT s rôznymi požiadavkami.

1. **Select pre spojenie informácii z dvoch tabuliek**
   Zistíme, ktorí hráči hrajú na poste obrancu.

   ```
   SELECT Hrac.meno, HracHraRelation.post
   FROM Hrac
   JOIN HracHraRelation ON Hrac.id_hrac = HracHraRelation.id_hrac_rel
   WHERE HracHraRelation.post='obrana';
   ```

   Zistíme, aké zápasy sa hrali na štadióne SuperStadium na adrese Cervinkova 66.

   ```
   SELECT Zapas.datum_a_cas_konania, Zapas.adresa, Zapas.kolo_ligy,
   Zapas.vysledok_zapasu, Zapas.superiaci_tim,
   Sportovisko.nazov_sportoviska
   FROM Zapas JOIN Sportovisko ON Zapas.adresa = Sportovisko.adresa
   WHERE Sportovisko.adresa='Cervinkova 66';
   ```

2. **Select pre spojenie informácii z troch tabuliek**

Zistíme, ktorý tréner trénuje dorasteneckú kategóriu a akú trénerskú funkciu v tíme zastáva.

```
SELECT Trener.meno, Trenuje.funkcia_v_time, Team.kategoria
FROM Trener
      JOIN Trenuje ON Trener.id_trener = Trenuje.id_trener_rel
JOIN Team ON id_team_rel = id_tim
WHERE Team.kategoria='Dorast';
```

3. **Select s klauzulou GROUP BY a agregačnou funkciou.**

Zistíme, koľko hráčov hrá v jednotlivých kategóriách tímov.

```
SELECT Team.kategoria, COUNT(*) AS POCET_HRACOV
FROM Hrac
JOIN HracHraRelation ON Hrac.id_hrac = HracHraRelation.id_hrac_rel
JOIN Team ON id_tim_rel = id_tim
GROUP BY Team.kategoria;
```

Zistíme, koľko zápasov bolo odohraných v jednotlivých kategóriách tímov.

```
SELECT Team.kategoria, COUNT(*) as POCET_ZAPASOV_V_KATEGORII
FROM Team
JOIN TeamHralRelation ON Team.id_tim =
TeamHralRelation.id_team_hral_zapas
JOIN Zapas ON TeamHralRelation.id_zapas_hral_team = Zapas.id_zapasu
GROUP BY Team.kategoria;
```

4. **Select s predikátom EXISTS**

Zistíme, či existuje hráč s číslom dresu 99.

```
SELECT Hrac.meno
FROM Hrac
WHERE EXISTS (SELECT HracHraRelation.id_hrac_rel
      FROM HracHraRelation
      WHERE Hrac.id_hrac = HracHraRelation.id_hrac_rel AND cislo_dresu
= 99);
```

5. **Select s predikátom IN a vnoreným SELECT**

Zistíme, ktorí hráči boli narodení medzi rokmi 1998 a 2000.

```
SELECT Hrac.meno, Hrac.datum_narodenia
FROM Hrac
WHERE Hrac.datum_narodenia IN
      (SELECT datum_narodenia FROM Hrac
      WHERE datum_narodenia BETWEEN '01-JAN-1998' AND '31-DEC-2000');
```

Štvrtá časť projektu

Triggery

1. **Trigger pre automatické generovanie ID zápasu.**
   Pre generovanie ID zápasu využívame sekvenciu, ktorá začína na čísle jedna (`START WITH 1`) a inkrementuje o 1(`INCREMENT BY 1`). Trigger následne pri vkladaní alebo úprave tabuľky Zápas vloží na miesto ID inkrementovanú predchádzajúcu poslednú hodnotu zápisom (`:NEW.id_zapasu := zapas_seq.NEXTVAL;`).

2. **Trigger kontrolujúci správnosť pri vkladaní zápasov**
   V prípade, že sa má zápas ešte len uskutočniť, tak nemôže mať už vyplnený výsledok. Napríklad, ak bude mať dátum konania o mesiac a užívateľ bude chcieť vyplniť výsledok hodí chybu, pretože tento zápas ešte nemôže mať výsledok.
   Sú tu použité dve podmienky:
   ```
   IF (:NEW.vysledok_zapasu IS NOT NULL) THEN
   IF (:NEW.datum_a_cas_konania > CURRENT_TIMESTAMP) THEN
   ```

3. **Trigger pre automaticky generované kľúče u hráčov**
   Pre generovanie ID hráča využívame sekvenciu, ktorá začína na čísle jedna (`START WITH 1`) a inkrementuje o 1(`INCREMENT BY 1`). Trigger následne pri vkladaní alebo úprave tabuľky Hráč vloží na miesto ID inkrementovanú predchádzajúcu poslednú hodnotu zápisom (`:NEW.id_hrac := hrac_seq.NEXTVAL;`).

4. **Trigger pre automaticky generované kľúče väzobnej tabuľky viazanej identifikátormi na tabuľku Hrac. (*v tomto prípade)**
   Generovanie prebieha rovnakým spôsobom ako v bodoch 1 a 3, len so zmenou toho, že sa jedná o väzobnú tabuľku “HracHraRelation” a trigger pri vkladaní alebo úprave tejto tabuľky vloží na miesto ID inkrementovanú predchádzajúcu poslednú hodnotu zápisom `(:NEW.id_hrac_rel := hrac_rel_seq.NEXTVAL)`.
   **Upresnenie: Relačné identifikátory viazané na tabuľku Team negenerujeme iteratívne s krokom k=1. Hráči sú totiž explicitne priraďovaní do kategórií tímov.*

5. **Trigger pre stráženie spodnej vekovej hranice hráčov.**
   V prípade, že hráč je mladší ako 6 rokov (+/- rok tolerancia kvôli nádejným talentom mladším trocha ako 6 rokov), tak je vyvolaná výnimka poukazujúca na existenciu tejto udalosti počas vykonávania triggeru. Je vyvolaná s chybovým kódom “20000”. Predvedenie tejto časti vypadá nasledovne:
   ```
   IF (EXTRACT(YEAR FROM CURRENT_DATE()) - EXTRACT(YEAR FROM
   :NEW.datum_narodenia) < 6) THEN
        RAISE_APPLICATION_ERROR(-20000, 'Hráč je príliš mladý');
   END IF;
   ```

Procedúry

Najskôr je potrebné použiť `SET SERVEROUTPUT ON;` To nám umožní robiť výpisy vstavanou funkciou dbms_output.put_line().

1. **Procedúra, ktorá nám umožní zistiť všetky štadióny, s minimálnou kapacitou a typom, ktoré zadá užívateľ.**

   Vstupy sú "kapacita" typu `INTEGER` a "typ_stadionu" typu `VARCHAR`.
   Na prechádzanie riadkov používame kurzor ukazujúci na tabuľku "Sportovisko".
   Pre riadok zistíme typ pomocou `%ROWTYPE`. Vo vnútri procedúry potom otvoríme kurzor pomocou `OPEN` a prechádzame každý riadok a pokiaľ kapacita športoviska je väčšia ako zadaná hodnota a typ športoviska zodpovedá požadovanému typu potom vypíše informácie o tomto športovisku. Tieto kontroly sa vykonávajú príkazom IF. Na konci kurzor uzavrieme pomocou `CLOSE`. Pri prechádzaní riadkov je ešte dôležitý príkaz `FETCH`, ktorý nahráva obsah riadku do premennej. Na konci procedúry sa nachádza `EXCEPTION` blok a ten v prípade chyby spraví chybový výpis.
2. **Procedúra, ktorá nám umožní zistiť priemernú výšku hráčov pre jednotlivú tímovú kategóriu.**

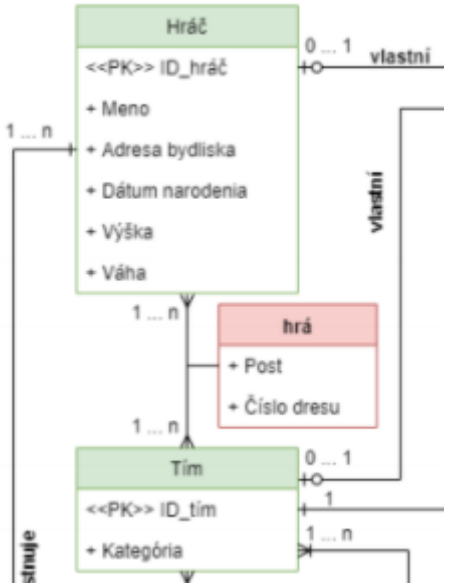
   Vstup je "nazov_kategorie" typu `VARCHAR`, kde je očakávaná jedna z tímových kategórií ako vstupný parameter od užívateľa. Procedúra začína deklaráciou premenných a definíciou kurzora(cursor_vyska_hracov). Pri deklarácii sú uvedené premenné na výpočet daného priemeru a explicitne deklarovaná výnimka programátorom "exception_ziadny_hraci", pre prípad, že v tímovej kategórií sa žiadny hráči nenachádzajú. Takto deklarovaná výnimka je použitá z ukážkových a demonštračných dôvodov (Je možnosť použitia preddefinovanej výnimky `ZERO_DIVIDE`). Predpokladáme, pre túto situáciu, že nechceme používať preddefinované chybové výstupy pri každej procedúre. Pri definícii kurzora sme si určili, čo bude našou aktívnou sadou dát, ktorú budeme používať pri získavaní potrebného údaju a to výšky. V kurzory používame prepojenie troch tabuliek. Ďalej sme si deklarovali jeden riadok záznamu (ktorý bude pri cykle, pri každej iterácií iný) "zaznam_vyska_hraca". Typ spomenutej premennej bude mať typ riadku, na ktorý kurzor ukazuje a to vďaka atribútu `%ROWTYPE`. Pokračujeme kontrolou, či kurzor nie je otvorený a keď nie, tak sme nami vytvorený kurzor otvorili pomocou `OPEN`. Následne vynulovali premenné a začali načítať z aktívnej sady dát riadok po riadku do premennej z pozície, kde práve kurzor ukazoval a počítať medzivýpočty pre danú požadovanú informáciu. K tomu nám pomohlo použitie `LOOP` a `FETCH`. Prechádzanie záznamov tabuľky končíme, keď kurzor neukazuje už na žiadny riadok, k tomu sme použili túto konštrukciu s atribútom `%NOTFOUND (EXIT WHEN cursor_vyska_hracov%NOTFOUND)`. Pokračujeme kontrolou, či v tímovej kategórií zadanú užívateľom sa nachádza nejaký hráč. Ak nie, tak vyvoláme na to definovanú výnimku (odkazujúc sa do `EXCEPTION` bloku) a ak áno, tak pokračujeme výpočtom priemernej výšky a následným výpisom na výstup terminálu klienta pomocou `DBMS_OUTPUT.put_line(...)`. V bloku `EXCEPTION`
   máme už spomínanú výnimku, ktorá vráti klientovi nulový výpis výsledku a vyvolá sa výnimka s chybovým kódom 20002. V ostatných prípadoch sa volá `ROLLBACK`, pre navrátenie vykonávaných zmien do stavu pred volaním tejto transakcie.

Explicitné vytvorenie indexu a výpis plánu EXPLAIN PLAN

Tieto dve operácie sme sa rozhodli aplikovať na túto požiadavku:

```
SELECT Team.kategoria, AVG(Hrac.vaha) AS PRIEMERNA_VAHA_TIMOVEJ_KATEGORIE
FROM Hrac JOIN HracHraRelation ON Hrac.id_hrac=HracHraRelation.id_hrac_rel
          JOIN Team ON id_tim_rel = id_tim
GROUP BY Team.kategoria;
```

jedná sa o komunikáciu medzi tabuľkami “Hrac”, “Team” a “HracHraRelation”, kde v ER diagrame sú to tieto tabuľky (v PL/SQL zdrojovom kóde reprezentácia mierne odlišná od reprezentácie na ER diagrame):



po zavolaní operácie `EXPLAIN PLAN` pre túto požiadavku sme dostali nasledovné:

```
----------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name                | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                     |    15 |  1380 |     2  (50)| 00:00:01 |
|   1 |  HASH GROUP BY                |                     |    15 |  1380 |     2  (50)| 00:00:01 |
|   2 |   NESTED LOOPS                |                     |    15 |  1380 |     1   (0)| 00:00:01 |
|   3 |    NESTED LOOPS               |                     |    15 |  1380 |     1   (0)| 00:00:01 |
|   4 |     NESTED LOOPS              |                     |    15 |   990 |     1   (0)| 00:00:01 |
|   5 |      INDEX FULL SCAN          | PK_HRACHRARELATION  |    15 |   390 |     1   (0)| 00:00:01 |
|   6 |      TABLE ACCESS BY INDEX ROWID| TEAM              |     1 |    40 |     0   (0)| 00:00:01 |
|*  7 |       INDEX UNIQUE SCAN       | PK_TIM              |     1 |       |     0   (0)| 00:00:01 |
|*  8 |     INDEX UNIQUE SCAN         | PK_HRAC             |     1 |       |     0   (0)| 00:00:01 |
|   9 |    TABLE ACCESS BY INDEX ROWID| HRAC                |     1 |    26 |     0   (0)| 00:00:01 |
----------------------------------------------------------------------------------------------------
```

Po decentnom naštudovaní témy zlepšovania výkonnosti databáz (z prednášok, demo cvičenia prednášáného RNDr. Marekom Rychlým Ph.D. a voľne dostupných zdrojov na internete) vieme povedať, že sme schopný ovplyvniť typ JOIN CONDITION a akým spôsobom sa uplatní. Ako je vidieť na obrázku vyššie, tak máme 3-krát NESTED LOOPS JOIN, ktorý funguje tak, že každý riadok tabuľky sa porovnáva so všetkými riadkami druhej tabuľky. Táto metóda spojenia má *časovú zložitosť O(N*M)*, čo je veľmi *pomalá metóda.*

A teda sme sa rozhodli navrhnúť optimalizáciu pre efektívnejšie spracovanie. Ako prvé sme skúsili podľa zadania napísať *explicitne vlastný index*, lenže sme narazili na "problém", ktorý sa nám *nepodarilo odstrániť*. A to taký, že po vytvorení rôznych kombinácií explicitne napísaných indexov *a po opätovnom zavolaní operácie EXPLAIN PLAN nenastala žiadna zmena a tabuľka vykonania plánu s operáciami bola totožná*. Takže od tejto možnosti riešenia sme nakoniec upustili (ale i napriek tomu nechávame v kóde zakomentované explicitné vytváranie indexu), kde predpokladáme, že toto spôsobilo nadefinovanie primárnych kľúčov a previazanie cudzími kľúčmi, že zaindexovalo spomínané 3 tabuľky dostatočne dobre a preto sa neudiali žiadne zmeny.

Našli sme teda iné riešenie a to pomocou použitia nápovedy pre optimalizér takzvaného **HINT-u** (/*+ HINT */) Zistili sme, že by sa nám hodilo nahradiť tie **NESTED LOOP JOINS** za **HASH JOINS** alebo **SORT MERGE JOINS** obe varianty pracujú rýchlejšie ako **NESTED LOOP JOIN**. Ale ktorú variantu si vybrať ? Snažili sme sa to zistiť.
O tabuľkách, ktorých sa optimalizácia týka vieme povedať:

1. Identifikátory u tabuľky hráčov majú usporiadané indexy vzostupne sú teda utriedené. Jednotlivé riadky ID sú zhodné s identifikátormi vo väzobnej tabuľke "HracHraRelation".
2. V tabuľke "Hrac" je očakávaných veľa záznamov, kde na druhú stranu v tabuľke "Team" záznamy budú pribúdať len výnimočne (ak by sa pridávala nejaká nová tímová kategória).

Bod 1 je priaznivý pre použitie možnosti **SORT MERGE JOIN**. Pretože sort merge join proces sa skladá z operácií merge a sort ako je uvedené v názve. A keďže tabuľky sú zoradené podľa identifikátorov, tak **SORT** operácia nebude vykonávaná tak dlho ako pri nezoradených tabuľkách. Dokonca výkonnostne by mala **SORT MERGE JOIN** predbehnúť **HASH_JOIN** v prípade zoradených tabuliek**.**
Bod 2 je priaznivý skôr pre možnosť použitia **HASH_JOIN** keďže v prvej spomenutej tabuľke sa očakáva veľmi veľa dát a v tej druhej veľmi málo.
Obe metódy sú efektívne pri veľkom počte dát oproti **NESTED LOOP JOIN**. Kde najväčší výkon má **HASH_JOIN** práve vtedy, keď sú dáta v tabuľke veľké, usporiadané a neindexované no napriek tomu je stále výkonnejší **SORT MERGE JOIN**, ak bavíme hlavne o spracovaní veľkých tabuliek. Kde ten je rýchlejší a spotrebuje menej pamäte ako **HASH_JOIN**.
Keďže sme zistili, že argumenty jednotlivých metód optimalizácie sa nám "bijú", tak sme sa rozhodli pre experiment a vyskúšame oba prístupy.

Začneme s **HASH JOIN,** kde takto poradíme optimalizátoru:

```
EXPLAIN PLAN FOR
SELECT /*+ USE_HASH(Team Hrac HracHraRelation) */ …
```

Dostali sme takýto výsledok, kde môžeme vidieť, že sme sa už minimálne zbavili **NESTED LOOP JOINS.**

```
-----------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name                | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                     |    15 |  1770 |     7  (29)| 00:00:01 |
|   1 |  HASH GROUP BY                |                     |    15 |  1770 |     7  (29)| 00:00:01 |
|*  2 |   HASH JOIN                   |                     |    15 |  1770 |     6  (17)| 00:00:01 |
|   3 |    MERGE JOIN                 |                     |    15 |  1170 |     3  (34)| 00:00:01 |
|   4 |     TABLE ACCESS BY INDEX ROWID| HRAC                |    16 |   416 |     1   (0)| 00:00:01 |
|   5 |      INDEX FULL SCAN          | PK_HRAC             |    16 |       |     1   (0)| 00:00:01 |
|*  6 |     SORT JOIN                 |                     |    15 |   780 |     2  (50)| 00:00:01 |
|   7 |      VIEW                     | VW_GBF_10           |    15 |   780 |     1   (0)| 00:00:01 |
|   8 |       HASH GROUP BY           |                     |    15 |   390 |     1   (0)| 00:00:01 |
|   9 |        INDEX FULL SCAN        | PK_HRACHRARELATION  |    15 |   390 |     1   (0)| 00:00:01 |
|  10 |    TABLE ACCESS FULL          | TEAM                |     3 |   120 |     3   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------
```

Pokračujeme so **SORT MERGE JOIN,** kde takto poradíme optimalizátoru:

```
EXPLAIN PLAN FOR
SELECT /*+ USE_MERGE(Team Hrac HracHraRelation) */ …
```

```
-----------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name                | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                     |    15 |  1380 |     7  (58)| 00:00:01 |
|   1 |  HASH GROUP BY                |                     |    15 |  1380 |     7  (58)| 00:00:01 |
|   2 |   MERGE JOIN                  |                     |    15 |  1380 |     6  (50)| 00:00:01 |
|   3 |    TABLE ACCESS BY INDEX ROWID| HRAC                |    16 |   416 |     1   (0)| 00:00:01 |
|   4 |     INDEX FULL SCAN           | PK_HRAC             |    16 |       |     1   (0)| 00:00:01 |
|*  5 |    SORT JOIN                  |                     |    15 |   990 |     5  (60)| 00:00:01 |
|   6 |     VIEW                      | VW_GBF_10           |    15 |   990 |     4  (50)| 00:00:01 |
|   7 |      HASH GROUP BY            |                     |    15 |   990 |     4  (50)| 00:00:01 |
|   8 |       MERGE JOIN              |                     |    15 |   990 |     3  (34)| 00:00:01 |
|   9 |        TABLE ACCESS BY INDEX ROWID| TEAM            |     3 |   120 |     1   (0)| 00:00:01 |
|  10 |         INDEX FULL SCAN       | PK_TIM              |     3 |       |     1   (0)| 00:00:01 |
|* 11 |        SORT JOIN              |                     |    15 |   390 |     2  (50)| 00:00:01 |
|  12 |         INDEX FULL SCAN       | PK_HRACHRARELATION  |    15 |   390 |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------
```

Čo môžeme na základe získaných dát povedať ? Zhrnuli sme to v tejto tabuľke:

Aplikácia nápovedy pre optimalizátor	Počet krokov operácií	Počet spracovaných riadkov	Použitá pamäť	CPU spotreba (cena)
BEZ NÁPOVEDY	10	94	6966 B	8
HASH JOIN	11	133	9356 B	33
SORT MERGE JOIN	13	173	9416 B	43

Na základe tohto experimentu sme zistili, že použitie nápovedy nedosiahlo popisovanú a očakávanú efektivitu (teda aspoň nie pri menšom počte dát v tabuľke). Je vidieť, že náš návrh jednotlivých tabuliek a obsahujúcich kľúčov bol mierne efektívnejší ako tieto optimalizačné varianty (viď tento scenár).

Prístupové práva

V tejto časti bola naša motivácia, taká že druhý člen, ktorému budú udeľované práva reprezentuje trénera s menšími právomocami ako hlavný tréner. To znamená, že má celkový prístup k tabuľkám hráčov, vybavenia, tímov a tréningov. Ale v prípade športovísk, zápasov a trénerov nemá sprístupnenú možnosť editácie. Celkový prístup je vytvorený príkazom `GRANT ALL ON` nazov_tabulky `TO` meno užívateľa. Obmedzený prístup sprístupňuje len selecty týmto spôsobom: `GRANT SELECT ON` nazov_tabulky `TO` meno užívateľa.

Materializovaný pohľad

Materializovaný pohľad ukazuje hráčov v útoku. Vytvára sa príkazom `CREATE MATERIALIZED VIEW`. Príkaz `BUILD IMMEDIATE` určuje, že pohľad sa naplní hneď po vytvorení. Aktualizácia pohľadu sa vykonáva po commite príkazom `REFRESH ON COMMIT AS`. Potom už nasleduje iba potrebný `SELECT`.