Budapesti Műszaki- és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Elektronikai Technológia Tanszék

Összesítőtáblák generálása és optimalizálása ad-hoc adatpiaci lekérdezésekhez Oracle 11g környezetben

TDK dolgozat

Készítette: Ableda Péter
Konzulens: dr. Tilly Károly,
dr. Martinek Péter

Tartalomjegyzék

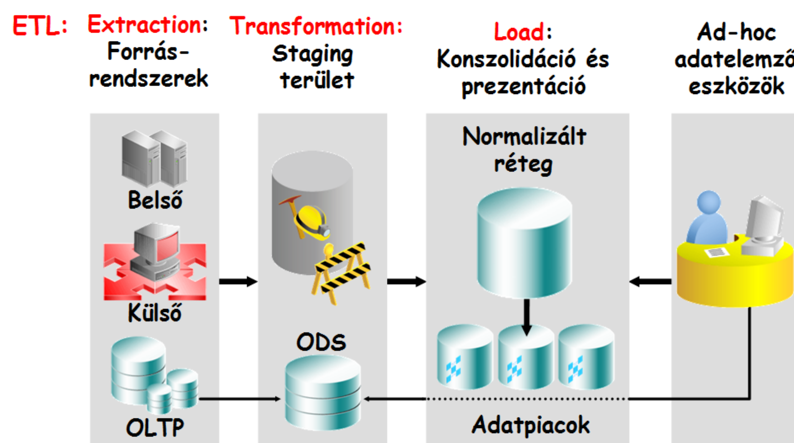
Tartalmi összefoglaló 3
1. Bevezetés 4
1.1 Adattárházak alapfogalmai 6
1.2 Architektúra 7
1.3 Séma 8
1.3.1 Összesítő táblák 8
2. Előzmények 10
3. Oracle megoldások 12
3.1 Az SQL Access Advisor 12
4. Összesítőtáblák az Oracle 11g adatbáziskezelőben 16
4.1 Materializált nézetek 16
4.2 Tervezési megfontolások a Query Rewrite képességeinek növelése érdekében 19
4.2.1 Megkötések 19
4.2.2 Dimenziók 19
4.2.3 Külső összekapcsolások (Outer Joins) 19
4.2.4 Szöveges illesztés 20
4.2.5 Aggregátumok 20
4.2.6 Csoportosítási logikák 20
5. Megvalósítás 21
5.1 A keresőfa felépítése és bejárása 21
5.2 Megfontolások materializált nézetek összevonhatóságára 23
5.2.1 Koncepció 24
5.2.2 A közös szűrőfeltételek meghatározása 27
5.2.3 A közös csoportosítások meghatározása 28
5.2.4 A rendezés (ORDER BY klauzula) feldolgozása az összevonás során 32
5.3 Lehetőségek Materializált nézetek szétbontására: 32
5.4 Materializált nézetek elhagyása 33
5.5 Az elkészült alkalmazás 33
6. A megoldás értékelése 36
7. Elért eredmények 39
8. Továbbfejlesztési lehetőségek 40
Felhasznált irodalom 41
Függelék 42

Tartalmi összefoglaló

A korszerű adattárházak rendkívül nagy mennyiségű (tipikusan több terabyte-nyi) adatot tárolnak, igen nagyszámú, bonyolult objektum formájában. A lekérdezési idők minimalizálása érdekében az adattárházakban tárolt adatokat az üzleti felhasználók számára általában speciális, lekérdezésekre optimalizált, csillag vagy hópihe szerkezetű adatpiaci rétegen keresztül teszik hozzáférhetővé.
Az óriási adatmennyiség, a tipikusnak mondható, nagyobb (pl. havi, régiónkénti, termékkategóriánkénti) adatcsoportokra vonatkozó aggregált lekérdezések és a végfelhasználók által elvárt, általában legfeljebb néhány perces lekérdezési idők miatt az adattárházak építésének egyik legnehezebb része az adatpiacok optimalizálása.
Az optimalizálás igen hatékony, ugyanakkor bonyolult és ellentmondásos módja az összesítőtáblák alkalmazása. Az összesítőtáblák gyakran előforduló aggregált lekérdezések (pl. napi jelentések) eredményeit tárolják, és tartalmukat rendszeresen (pl. naponta) frissítik. A fejlett adatbáziskezelő rendszerekben az SQL motor képes az alaptáblákra hivatkozó lekérdezéseket automatikusan összesítőtáblák fölötti lekérdezésekké átírni. Mivel az összesítőtáblák mérete általában több nagyságrenddel kisebb az alaptábláknál, illetve nem csak aggregátumok, hanem komplex összekapcsolások eredményeit is tartalmazhatják, a lekérdezésátírás drámai sebességnövekedést eredményezhet.
A hatékonyságnövekedésért azonban komoly árat kell fizetni, főként ad-hoc lekérdezéseket megengedő adatpiaci rendszerekben, ahol a lekérdezések tervezés időben nem láthatók pontosan előre. A legfőbb problémát ilyenkor az egyszerű módszerekkel előállítható összesítőtáblák nagy száma, nagy összmérete és karbantartásuk rendkívüli erőforrásigénye okozza.
Ebben a dolgozatban bemutatom az összesítőtáblákkal és lekérdezésátírással kapcsolatos általános alapfogalmakat és alapvető technikákat, elemezve a módszer előnyeit és buktatóit.
A rendelkezésre álló dokumentáció alapján áttekintem az Oracle 11g adatbázis szerver által összesítőtáblák kezelésére kínált eszközöket és módszereket. Megvizsgálom ezen eszközök előnyeit és korlátait.
Munkám során olyan optimalizálási technikákat tanulmányozok és dolgozok ki, amelyekkel adott lekérdezéshalmazra alkalmazható összesítőtáblák száma és mérete minimalizálható a várható végrehajtási sebesség maximalizálása és az összesítőtáblák karbantartási időinek minimalizálása mellett. Az elkészült megoldásomat tesztelem egy adott lekérdezéshalmazra, az eredményemet összevetetem az Oracle vizsgált eszközének az eredményével.

1. Bevezetés

Az adattárházak építésekor alkalmazott alapelveket Ralph Kimball és Bill Inmon fogalmazták meg, akik egymástól eltérően vélekednek az adattárházak feladatairól.
Kimball definíciója szerint ("The conglomeration of an organization's data warehouse staging and presentation areas, where operational data is specifically structured for query and analysis performance and ease-of-use." [1]) az adattárház riportok és nagyteljesítményű, egyszerűen kezelhető elemzések kiszolgálására készített adatbázis, ahol egy adott szervezet adatait a célnak megfelelően átstrukturálva tárolják.
Míg Kimball az adattárházakat speciális struktúrájú adatbázisnak tekinti, Inmon definíciója szerint ("A data warehouse is a subject oriented, integrated, nonvolatile, and time variant collection of data in support of management's decisions." [2]) az adattárházakat alapvetően nem a struktúrájuk, hanem a tartalmuk és felhasználásuk módja különbözteti meg más adatbázis alkalmazásoktól.



1. ábra. Adattárházak általános architekturális elemei

A korszerű adattárházak az 1. ábrán látható háromrétegű architektúrába szerveződnek, amelynek legalsó rétege a *staging* (átmeneti tároló) terület, középső rétege a *normalizált*-, és legfelső rétege az ú.n. *adatpiaci* réteg. Bár az inmoni és kimballi megközelítés első pillantásra ellentmondónak tűnik, a gyakorlatban kiegészítik egymást, ugyanis a normalizált réteg inmoni, az adatpiacok pedig kimballi elvek alapján szerveződnek.

A forrásrendszerekből származó sokféle formátumú és eltérő szerkezetű adatot a staging rétegbe gyűjtik össze, majd onnan a normalizált rétegbe töltik, amely az adatokat egységes, konszolidált szerkezetben tárolja. Így például, ha egy cég ügyfeleit több különböző forrásrendszerben kezelik, az ezekből származó ügyféladatokat a normalizált rétegben egyetlen közös struktúrába vonják össze, ahol lehetőség szerint minden ügyfél adatai csak egyszer szerepelnek, és az összes lényeges ügyfélattribútum megjelenik.

Az adatpiaci réteg tartalmát a normalizált rétegből állítják elő. Az adatpiaci réteg elsődlegesen az üzleti elemzők és egyéb adatmegjelenítési feladatok céljait szolgálja, ezért lekérdezésekre optimalizált adatstruktúrákat tartalmaz.

Míg a normalizált réteg entitásai közötti kapcsolatok topológiája gráf, az adatpiaci réteg objektumai fa struktúrába szerveződnek, melyet az elfogadott szakzsargon csillag illetve hópihe struktúrának nevez. A fa struktúrát felhasználói oldalról az elemzésekben alkalmazott lekérdezések függvényszerű volta indokolja. A tipikus elemzésekben ugyanis adott kritériumok (pl. időpont, földrajzi hely, termékkategória) függvényében akarjuk meghatározni

bizonyos értékek változását (pl. eladott darabszám, vételár vagy forgalom). Az adatpiacokban a független változókat *dimenzióknak*, a függvényértékeket pedig *tényeknek* nevezik, és ezeket egymástól elkülönítve tárolják. Egy adatpiaci csillag struktúra ezek alapján egy tény táblából és a hozzá kapcsolódó tetszőleges számú dimenzió táblából áll. Mivel a dimenzió táblák tipikusan diszkrét kategóriákat tartalmaznak (pl. terméktípus, országok, megyék vagy ügyfélkategóriák), ezek számossága viszonylag kicsi (leggyakrabban nem több, mint néhány ezer rekord). Ezzel szemben a ténytáblák konkrét eseményekre, tranzakciókra és fizikai egyedekre vonatkozó információkat tárolnak, általában évekre visszamenően, így ezek mérete óriási. Manapság már hazánkban sem ritkák a több milliárd soros ténytáblák.

A lekérdezések optimalizálásának szempontjából a csillag struktúrájú adatsémák rendkívül előnyösek, ugyanis a függvényszerű lekérdezésekben ritka kivételektől eltekintve dimenzió táblákat kapcsolunk a hozzájuk tartozó ténytáblához, amelyek között közvetlen kapcsolat létezik, továbbá, a fa struktúra miatt a séma bármely két táblája között egy és csakis egy kapcsolási út van. Mindez jelentősen megkönnyíti az SQL motor számára a lekérdezésekhez tartozó optimális végrehajtási tervek automatikus generálását.

A csillag struktúra előállítása a normalizált réteg objektumai alapján az egyik legbonyolultabb adatmodellezési feladat, amely gyakran megkívánja, hogy a dimenziókat 2NF alakban tároljuk a normalizált réteg legalább 3NF alakja helyett. Bár ily módon elérhető a csillag struktúra, a 2NF denormalizáció miatt egyes dimenziótáblák mérete kellemetlenül megnőhet (több százezer vagy akár több millió sorosra). Ilyen esetekben megengedett a túl nagy méretű dimenziók egyes, a lekérdezésekben ritkábban hivatkozott attribútumainak 3NF alakúra hozása. Ezzel kialakulhatnak többszintű dimenziók, amelyek a hópihe struktúrát eredményezik.

Az 1. ábrának megfelelően az adatpiacok tartalmát az informatikai szakképzettséggel nem rendelkező végfelhasználók - tipikusan üzleti elemzők - speciális eszközök segítségével kérdezhetik le, amely lehetővé teszi számukra az adatpiacban szereplő adatelemek fölött tetszőleges (ú.n. ad-hoc) lekérdezések megfogalmazását, és az eredmények könnyen értelmezhető, általában táblázatok, kereszttáblázatok és grafikonok formájában való megjelenítését.

Az adatpiacok fölötti ad-hoc lekérdezések elsődleges célja naprakész, gyors, pontos információk előállítása, illetve az elemző igényeinek megfelelően, a végrehajtott korábbi elemzések eredményei alapján újabb lekérdezések megfogalmazása és végrehajtása. Egyrészt ez a fajta működési mód nem engedi meg, hogy a lekérdezések végrehajtásának átlagos válaszideje néhány percnél hosszabb legyen, másrészt a lekérdezések gyakran a ténytáblák rekordjainak jelentős hányadát, nem ritkán több száz millió sort érintenek. Az ilyen – sok sort felölelő, ú.n. aggregált - lekérdezések tipikus példái a hosszabb időszakokra (pl. hónapokra, évekre) visszanyúló összesítéseket (számosságokat, összegeket, átlagokat) tartalmazó kimutatások.

Az összesítéseket tartalmazó aggregált lekérdezések gyors végrehajtása azért okoz problémát, mert ezeknél csődöt mondanak az OLTP rendszerekben hagyományosan jól bevált indexek. Ha például meg akarjuk tudni, hogy egy telekommunikációs cég ügyfelei egy évre visszamenőleg átlagosan milyen hosszú telefonbeszélgetéseket folytattak, ehhez be kell olvasni az adott évben folytatott beszélgetések adatait tartalmazó összes rekordot, majd átlagolni kell a hívások hosszát. Ez biztosan több milliárd rekord beolvasását igényli, ami legalábbis órákat vesz igénybe. Ha indexeléssel próbálkozunk, a helyzet csak romlani fog, hiszen ez esetben nem elég a ténytáblánk sorait végigolvasni, hanem még az indexből is sorban ki kell olvasni a megfelelő sorokra hivatkozó bejegyzéseket.

A megoldás ilyenkor az összesítőtáblák alkalmazása. Az összesítőtáblák aggregált lekérdezések eredményeit tartalmazzák (a beszélgetések átlagos hosszára például évenként egyetlen sort az évi több milliárd sor helyett). Ezek után már csak annyit kell tennünk, hogy a lekérdezést nem a többmilliárd soros ténytáblánk, hanem a néhány soros összesítő tábla fölött hajtjuk végre. Ezzel a lekérdezési idő több nagyságrenddel csökkenthető. Bizonyos feltételek mellett a korszerű relációs adatbáziskezelő rendszerek (pl. az Oracle 11g) adott lekérdezések végrehajtásakor képesek fölismerni, hogy bizonyos alaptáblák fölött megfogalmazott lekérdezések eredményével ekvivalens adatok találhatók megfelelő összesítőtáblákban. Ilyenkor automatikusan átírják az eredeti lekérdezés szövegét úgy, hogy az alaptáblák helyett az összesítő táblákra hivatkozzon, és a lekérdezést ebben az átírt formában hajtják végre.

Ezt a módszert *lekérdezésátírásnak* (query rewrite) nevezik. A lekérdezésátírás nagy előnye többek között az elérhető drámai sebességnövekedés, illetve, hogy működő alkalmazások, már létező kimutatások és lekérdezések optimalizálására is használható anélkül, hogy az eredeti lekérdezés szövegén változtatni kellene.

A lekérdezésátírásnak azonban ára van, mégpedig az összesítőtáblák tárolásának és karbantartásának a terhe. Az összesítőtáblák tartalmát le kell generálni, majd a tartalmukat az adatpiacok frissítésekor – tipikusan naponta – frissíteni kell. A frissítésnek bele kell férnie az adattárház és adatpiacok töltésére rendelkezésre álló napi időkeretbe, az ú.n. batch window-ba, amely általában nem több, mint 4-8 óra. Másrészt az összesítőtáblák nem lehetnek túl specifikusak, mert így csökken a használatuk várható gyakorisága, de nem lehetnek túl általánosak sem, mert ezzel általában jelentősen csökken az általuk elérhető sebességnövekedés. Itt érezhetően egy igen összetett és izgalmas optimalizálási problémával állunk szemben, amelynek megoldása nem triviális ugyan, de jelentős előnyökkel kecsegtet.

A dolgozatom célja tehát olyan optimalizálási módszerek kidolgozása, amelyek adott lekérdezéshalmazokhoz képesek meghatározni minél kevesebb és lehetőség szerint minél kisebb méretű összesítőtáblát, amelyekre az adott lekérdezések automatikusan átírhatók, és ezáltal a lehető legnagyobb sebességnövekedést érhetjük el. Az optimalizálási eljárásokat Oracle 11g környezetben valósítottam és próbáltam ki konkrét mintapéldákon.

Az adattárházakkal kapcsolatos alapfogalmak ismertetését követően, a 2. fejezetben összesítőtáblák generálását és optimalizálását megvalósító megoldásokat mutatok be, melyeket összevetek az általam kidolgozott megoldással. A 3. fejezetben bemutatom az Oracle tuning eszközeit, melyek segítségével az adatbázisok és adattárházak hatékonysága növelhető, ismertetem ezen eszközök korlátait és hátrányait. Ezek után a 4. fejezetben az összesítőtáblákkal kapcsolatos Oracle specifikus megoldásokat mutatok be. Az 5. fejezetben részletesen ismertetem az általam kidolgozott megoldást materializált nézetek generálására, valamint azokat az optimalizálási metódusokat, melyek segítségével előállítható egy optimális materializált nézet halmaz. Az elkészült alkalmazásomat teszteltem, a részletes eredményeimet a 6. fejezetben az Oracle eszközei javaslatával hasonlítottam össze. Ezt követően a 7. fejezetben összefoglalom az elért eredményeimet, valamint továbbfejlesztési lehetőségeket mutatok be a 8. fejezetben.

1.1 Adattárházak alapfogalmai

Az adattárházak megjelenésével újszerű rendszer jött létre, amely a hagyományos Online Transaction Processing (OLTP) rendszerektől eltérő követelményeknek kell megfeleljen. A rendszer optimalizálása során ezen követelményeknek való minél jobb megfelelés a cél, ami új megoldások alkalmazásával érhető el. Az adatttárházak és OLTP rendszerek közötti eltérő követelmények főként a következő területeken mutatkoznak:

- **Terhelés:** Az adattárházakat ad-hoc (vagyis nem előre definiált) lekérdezések és adatelemzések feldolgozására tervezik. E mellett a lekérdezések által érintett adatmennyiség általában rendkívül nagy, hiszen tipikusak az aggregált lekérdezések (pl. hogyan alakultak egy adott termékcsoport eladásai az elmúlt három évben). Ez azt jelenti, hogy tervezési időben nem becsülhetők pontosan a jövőbeni terhelésekkel kapcsolatos adatok. Ezért az adattárházakat úgy kell optimalizálni, hogy a sokféle lehetséges lekérdezést és elemzési feladatot a lehető legjobban ki tudják szolgálni. Ezzel szemben az OLTP rendszerek tipikusan előre definiált feladatokat látnak el, és a lekérdezésekben érintett adatmennyiség általában kicsi (pl. egy-egy utas beszállókártyát kér egy adott repülőgépjáratra). Ezért az alkalmazásokat elég úgy hangolni, vagy tervezni, hogy az adott lekérdezésekre és specifikus szűrőfeltételekre (pl. jegy azonosító, járatszám) hatékonyan működjenek.

- **Adatmódosítások:** Az adattárházak adatait speciális adatbetöltő (ún. ETL) folyamatokon keresztül frissítik, általában éjszakánként vagy hétvégenként. Míg adattárházakban a végfelhasználók nem módosítják az adatokat, OLTP rendszerekben a végfelhasználók rendszeresen módosítják az adatokat. Ebből az is következik, hogy OLTP rendszerekben minden adat naprakész, és az üzleti tranzakciók legfrissebb állapotát tükrözi.

- **Általános utasítások:** Egy általános adattárház több ezer, esetenként több millió sort dolgoz fel utasításonként, ezzel szemben az OLTP rendszerek sokkal kezelhetőbb adatmennyiséggel dolgoznak. Ennek megfelelően az adattárházak lekérdezései sokkal erőforrásigényesebbek mint az OLTP rendszereké.

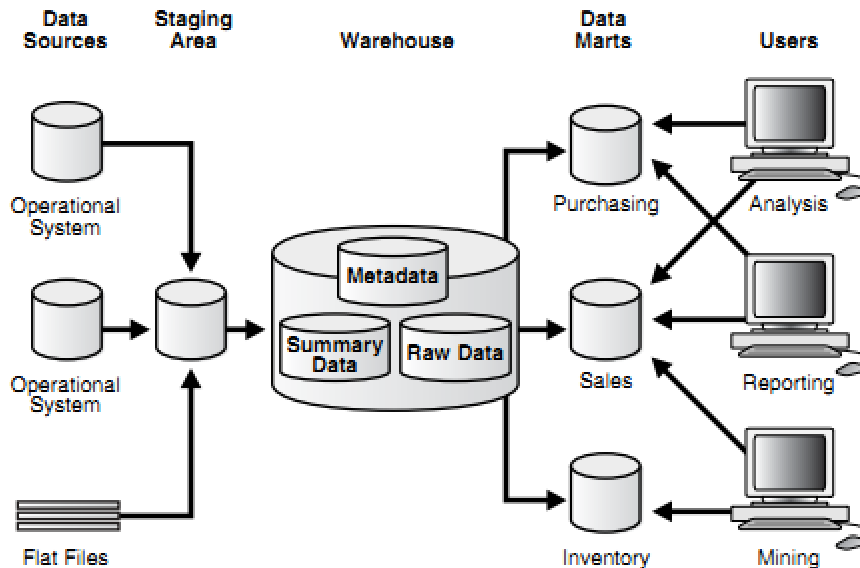
1.2 Architektúra

Ebben a dolgozatban adatpiacok optimalizálásával foglalkozom, amelyek egy vállalat kisebb csoportjai számára készült, konkrét feladatot ellátó, adattároló és elemző egységek, amelyek önmagukban is adattárház funkciókat láthatnak el. Többféle adattárház architektúra létezik, melyek közül csak egyet mutatok be. További koncepciók találhatók [1]-ben és [2]-ben. A következőkben bemutatott architektúra egy adatpiacokat kiszolgáló rendszert valósít meg.
A más rendszerektől származó adatokat meg kell tisztítani és fel kell dolgozni, mielőtt, betöltenénk az adattárházba. Ezt a folyamatot az esetek többségében egy átmeneti adatbázis segítségével valósítják meg, ahol a szükséges adatokat átalakítják, a feleslegeseket eldobják, valamint ellenőrzik a konzisztenciát.
Az 2. ábrán az OLTP rendszerekben is megjelenő metaadatok és nyers adatok mellet megtalálhatóak az összesítő adatok. Az összesítések nagyon értékesek az adattárházakban, ezek segítségével hosszú feladatok eredményeinek eltárolásával jelentősen csökkenthető a rendszer válaszideje – lásd részletesen később az 1.4.1. fejezetben.
Az esetek többségében érdemes a különböző végfelhasználók igényeire szabni az adattárházat. Ezt egy új absztrakciós szint létrehozásával lehet megtenni. Ezt a réteget adatpiaci rétegnek nevezik. Egy adatpiaci réteg létrehozása sokkal kisebb költséggel jár, mint egy teljes adattárház implementálása, mégis képes közös nézetet definiálni bizonyos felhasználói csoportoknak. Az adatpiacok tipikusan különböző üzletágak számára tervezett adathozzáférési felületek. Segítségükkel egyszerűbben és gyorsabban lehet elérni a gyakran használt adatokat.
Inmon [3] szerint a független adatpiacok egy logikai (view) vagy fizikai (extract) részhalmazát valósítják meg egy nagy adattárháznak. A különválasztás oka lehet:

- Egy speciális séma vagy adatmodell számára frissítési felületként szolgáljon, például átstrukturálja az adatokat OLAP kockára.

- Teljesítmény: a nagyobb hatékonyság érdekében külön szerverekre lehet tenni az egyes részeket, így tehermentesíthetjük a központi adattárházat.

- Biztonság: a különböző jogosultságok szerint szétválaszthatjuk az adathalmazokat.
- Célszerűség: elkerülhető, hogy a központi adattárháznak hozzáférési és adatirányítási képességekkel keljen rendelkeznie.



2. ábra. Adattárház architektúra [4]

1.3 Séma

Az adattárházak és adatpiacok optimalizálásának többféle módja van. Egyik az adatpiacok lekérdezéseinek módosítása: ezzel csökkenteni lehet a lekérdezések válaszidejét, így a rendszer hatékonysága nőni fog. A másik megoldás a rendszer sémájának módosítása: ekkor olyan objektumokat hozunk létre, amelyek használatával a lekérdezési időket csökkenteni tudjuk. Ad hoc lekérdezések esetén nemcsak hogy nem tudjuk módosítani a rendszeren futó lekérdezéseket, ezeket tervezési időben nem is látjuk előre, így csak a második megoldást lehet alkalmazni.

1.3.1 Összesítő táblák

Az egyik technika az adattárházak teljesítményének növelésére az összesítőtáblák alkalmazása. Az összesítőtáblák gyakran előforduló aggregált lekérdezések (pl. napi jelentések) eredményeit tárolják, és tartalmukat rendszeresen (pl. naponta) frissítik. A fejlett adatbázis-kezelő rendszerekben az SQL motor képes az alaptáblákra hivatkozó lekérdezéseket automatikusan összesítő táblák fölötti lekérdezésekké átírni. Mivel az összesítőtáblák mérete általában több nagyságrenddel kisebb az alaptábláknál, illetve nem csak aggregátumok, hanem komplex összekapcsolások eredményeit is tartalmazhatják, a lekérdezésátírás drámai sebességnövekedést eredményezhet.

Megfontolások összesítőtáblák alkalmazására

A hatékonyságnövekedésért komoly árat kell fizetni, főként ad-hoc lekérdezéseket megengedő adatpiaci rendszerekben, ahol a lekérdezések tervezés időben nem láthatók pontosan előre.
Ha ad-hoc lekérdezések teljesítményét szeretnénk növelni, akkor első pillantásra úgy tűnik, hogy minél általánosabb összesítőtábla alkalmazására kell törekedni, hiszen minél

általánosabb egy összesítőtábla annál szélesebb körben használható lekérdezésátírásra. Ez azt jelenti, hogy az összesítőtábla mérete az általánosítás során jelentősen nőni fog, ami meglehetősen rossz megközelítés, helyette érdemesebb több egyszerűbb összesítőtáblát létrehozni. Az így létrehozott táblákon belüli függőségek kisebb redundanciát okoznak, mint a közös összesítőtábla esetén, tehát csökken a táblák összmérete. Az is probléma, ha túl sok összesítőtáblát használunk, ekkor ugyanis az összesítőtáblák összekapcsolása jelentős terhelést jelent a rendszernek, ami a lekérdezési idők növekedésével jár, továbbá az összesítőtábláknak fenntartási költségeik is vannak. Az összesítőtáblákat adott rendszerességgel frissíteni kell. Ezt tipikusan éjszaka vagy hétvégén végzik el, attól függően, hogy az elemzések során mennyire követelik meg a legfrissebb adatok használatát.
Az összesítőtáblák megtervezésekor tehát törekedni kell a minél szélesebb körben való használhatóságra, de figyelembe kell venni a karbantartási és tárolási korlátokat is, tehát cél a végrehajtási sebesség maximalizálása az összesítőtáblák karbantartási időinek és tárigényének minimalizálása mellett.

2. Előzmények

Az összesítőtáblák kiválasztási problémája jelentős figyelmet kapott a szakirodalomban. A kutatások az alábbi pontokban térnek el egymástól [9]:

- Hogyan választják ki a lehetséges összesítőtáblákat?
- A keretrendszer használható-e a lekérdezések közötti kapcsolatok észlelésére?
- Matematikai költségfüggvényt használnak-e vagy a lekérdezés optimalizálót?
- A nézetek kiválasztása relációs vagy multidimenziós kontextusban történik-e?
- Többszörös vagy egyszerű-e a lekérdezés optimalizálás?
- Elméleti vagy technológiai a megoldás?

Az általam kidolgozott megoldás terhelés alapú, azaz adott lekérdezések végrehajtása alapján állítja elő a lehetséges összesítőtáblákat. Ezeket a nézeteket csoportokba rendezi, és egy matematikai költségfüggvény segítségével, mohó algoritmussal választja ki az összesítőtáblák optimális halmazát.

A megoldásommal szemben a klasszikus tanulmányok az összesítőtáblák kiválasztására rácsos szerkezetű keretrendszert javasolnak, amellyel a függőségek az aggregált nézetek mentén, többdimenziós kontextusban modellezhetők és rögzíthetők.

Harinarayan, Rajaraman, és Ullman [5] mohó algoritmusa, egy költségmodell segítségével képes döntéstámogató rendszerek (DSS) teljesítményének növelésére. Az algoritmusuk kihasználja, hogy DSS esetében a felhasználók tipikusan többdimenziós adatkockába szervezve kérdezik le az adatokat. Az algoritmus kiválasztja azon összesítőtáblákat, amelyekkel a lekérdezések kiértékelési költsége optimalizálhatók. Ők az én megoldásommal ellentétben a nézetek karbantartási költségeivel és a tárolási korlátokkal nem foglalkoznak.

Ezek a kezdeti megoldások arra a speciális esetre fókuszálnak, amikor minden aggregátum egy kockából számítható ki (a csillag séma esetén egyetlen ténytábla található). Ez nem szerencsés, hiszen a legtöbb valós alkalmazás több ténytáblára való aggregátumot követel meg. Ezt a problémát már Shukla, Deshpande és Naughton [10] is megfogalmazták. Rámutattak arra is, hogy milyen egyéb megfontolások és nehézségek vannak összesítőtáblák kiválasztása során több kockából álló (multi-cube) rendszerek esetében.

Egy másik elméleti keretrendszer az „AND-OR” nézet gráfokat használ a nézetek közötti kapcsolatok rögzítéséhez. Gupta [7] továbbfejlesztette Harinarayan, Rajaraman és Ullman [5] mohó algoritmusát. Az ő módszere már figyelembe veszi a karbantartási költséget és tárolási korlátot az összesítőtáblák kiválasztása során. „AND-OR” nézet gráfok segítségével képes megjeleníteni az összes lehetőséget az összesítőtáblák létrehozására.

Karlapalem és Li [6] egy heurisztikus algoritmust fejlesztettek ki, amely több nézetet tartalmazó végrehajtási terv (Multiple View Processing Plan - MVPP) segítségével választja ki az összesítőtáblák egy optimális halmazát. Tapasztalataik szerint ezzel a halmazzal jó teljesítmény és alacsony karbantartási költségek érhetők el. A nézetek tárolási költségeit azonban az ő algoritmusuk sem veszi figyelembe.

Chan, Li és Feng [8] esettanulmányukban az összesítőtáblák tervezésében és kiválasztásában szerzett tapasztalataikat ismertetik. Egy valós vállalat csillag és hópihe sémájú adattárház rendszerét használják. Átvették és módosították Gupta mohó algoritmusát [7]. Új költségmodellt dolgoztak ki, hogy értékelni tudják az összesítőtáblák költségeit, és hasznukat. A költségmodelljük figyelembe veszi a karbantartási költségeket, a tárolási költségeket, és az elérhető sebességnövekedést is, de felhasznál olyan adatokat is, amelyek tervezési időben nem mindig állnak rendelkezésre. A kiszámított költségeik alapján választják ki az összesítőtáblák azon halmazát, amellyel a legjobb teljesítményt lehet elérni a legkisebb befektetés mellett.

Az eddig felsorolt megközelítések többsége pusztán elméleti, és nem nyújtanak skálázható megoldást a problémára. Másrészt az optimalizálandó rendszer sémáját dolgozzák fel, és

ennek alapján próbálják meghatározni a szükséges összesítőtáblákat, de nem veszik figyelembe a ténylegesen végrehajtott lekérdezéseket. Ennek a megközelítésnek az a hátránya, hogy semmi nem garantálja, hogy valaha keletkeznek olyan lekérdezések, amelyeket az optimalizáló ezen összesítőtáblák fölötti lekérdezésekké tudna alakítani.

A wavelet keretrendszer adaptívan reprezentálja a többdimenziós adatkockákat [11]. Ez a módszer wavelet elemek indexelt hierarchiájára bontja fel az adatkockákat, amelyek megfelelnek az adatkockák részleges és reziduális aggregációinak. Egy mohó algoritmus választja ki azon alacsony költségű wavelet nézet elemek halmazait, amely minimalizálják az adatkockára definiált lekérdezések átlagos feldolgozási idejét. Ebben a szellemben, Kotidis a Dwarf (törpe) struktúrát javasolja [12], amely megoldást ad az adatkockák tömörítésére. A tömörítés a redundanciák kiküszöbölésével valósul meg. A tömörítés csökkenti a karbantartási és lekérdezési költségeket is. Ezek a megközelítések főként a fizikai adattárolási modell módosításával próbálnak teljesítménynövekedést elérni. Ez az út azonban legtöbbször nem járható, hiszen a fizikai modell módosítására csak ritkán van lehetőség.

A legújabb megközelítések terhelésvezéreltek, és ezt az utat követi az én megoldásom is. Ezek a módszerek szintaktikai úton elemzik a terhelést, és ennek alapján előállítják a lehetséges összesítőtáblákat. A lekérdezésoptimalizáló segítségével, mohó algoritmussal építik fel a legmegfelelőbb összesítőtáblák konfigurációját.

A terhelés valóban jó kiindulási pont lehet, mert ez tartalmazza a végrehajtott lekérdezéseket, vagy szintaktikailag hasonlóak lesznek az előző terhelésekhez. Mivel a terhelés alapján állítjuk elő a lehetséges nézeteket, így biztosítható, hogy a létrehozott összesítőtáblákat használni fogják lekérdezések, és nem fölöslegesen keletkeznek és tartjuk őket karban.

Aouiche, Jouve, és Darmont [9] keretrendszere a megfelelő összesítőtáblákat adatbányászati módszerekkel, klaszterező eljárás segítségével választja ki. A megoldásukban egy adott terhelésminta alapján állítják elő a lekérdezési klasztereket (hasonlóságok és különbözőségek bevonásával). Ezen klaszterezés segítségével csoportokat lehet képezni a hasonló lekérdezésekből, végül ezek a csoportok szolgálnak a mohó algoritmusuk alapjául. Ez a megközelítés a klaszterezés alacsony komplexitásának köszönhetően jól skálázható (az attribútumok számát tekintve lineáris, a lekérdezések számát tekintve pedig logaritmikus komplexitású). A megoldásuk korlátozott tárhely esetén is jól skálázható. Megoldásukban egy hatékony módszert mutatnak be, ami képes kiválasztani a létrehozandó összesítőtáblákat.

Ezzel szemben az általam javasolt megoldás a lehető legtöbb lekérdezést kiszolgáló összesítőtáblát hoz létre, amit a lekérdezések összevonásával érek el. Egy alkalmas költségfüggvény segítségével hasonlítom össze az egyes lehetséges megoldásokat, így optimális összesítőtábla halmaz áll elő.

3. Oracle megoldások

Számos eszköz található az Oracle adatbázisok kezelésére és hangolására. Ezeket az eszközöket *tanácsadóknak* (advisor) hívják. A *tanácsadók* konkrét megoldásokkal szolgálnak az adatbáziskezelés legjellegzetesebb kihívásaira, a tár- és teljesítménykezeléstől kezdve az undo kezelésig. A tanácsadók segítenek az adatbázis teljesítményének növelésében. Három csoportjuk van: Automatic Database Diagnostic Monitor (ADDM), SQL Advisorok, és Memória Advisorok.

Az SQL Advisorok egy vagy több SQL utasítást vizsgálnak, és tanácsokkal szolgálnak, miként lehetne növelni a hatékonyságukat. Az Advisorok sokféle típusú tanáccsal szolgálhatnak, többek között SQL profilok készítése (információk halmaza, amely segítségével a lekérdezés optimalizáló egy optimális végrehajtási tervet képes készíteni), SQL utasítások átszervezése, új hozzáférési struktúrák készítése, az optimalizáló statisztikáinak a frissítése. Az Oracle Enterprise Manager Database Control (Database Control) segítségével ezeket a tanácsokat elfogadhatjuk, és a módosításokat végre is hajthatjuk. Az egyik Advisor az SQL Tuning Advisor, amely az SQL utasítások módosításával próbál teljesítménynövekedést elérni.

Az SQL Access Advisor az adatbázis sémájának módosítására tesz javaslatokat. Ez az eszköz hozzáférési struktúrákat ajánl, azaz materializált nézetekkel (összesítőtáblákkal), materializált nézet logokkal, partíciókkal és indexekkel segít elérni a kívánt hatékonyságnövekedést.

Az SQL Access Advisor index javaslatai kiterjednek a bitmap indexekre, függvény alapú indexekre és B-fa indexekre is. A bitmap indexek használatával csökkenhet az ad hoc lekérdezések válaszideje, és más indexelési technikákhoz viszonyítva is kisebb a tárhely igénye. B-fa indexeket adattárházak esetén csak egyedi, vagy közel egyedi oszlopokra érdemes létrehozni.

3.1 Az SQL Access Advisor

Az SQL Access Advisor [3] gyors (fast refreshable) és teljes (full refreshable) frissítésű materializált nézeteket is ajánl, melyek képesek általános (general rewrite) és szövegegyezéses (exact text match rewrite) lekérdezések átírására.

Az eszköz a DBMS_ADVISOR PL/SQL csomag TUNE_MVIEW procedúrája használatával meglévő materializált nézetek módosítására is készít javaslatokat, hogy a nézeteink gyorsan frissíthetőek (fast refreshable) legyenek, valamint képesek legyenek általános lekérdezésátírásra. A materializált nézetek fast refresh opciójának a használatával a nézeteket nem kell minden frissítéskor újraépíteni, a nézetek inkrementális módszerrel frissítődnek.

E mellett az SQL Access Advisor partíciókat is ajánl (az Oracle 11g Release 2-től kezdve) nem partícionált alaptáblák teljesítményének növelésére. Sőt képes partícionált indexek és materializált nézetek ajánlására is. A partícionált indexek és materializált nézetek létrehozása nem tér el a nem partícionált esettől, azonban a partícionálást nagyon meg kell fontolni. Különösen akkor, ha már indexeket, kényszereket, vagy triggereket definiáltunk a táblára.

Az Access Advisor az Oracle Enterprise Manager-ből futtatható az SQL Access Advisor Wizard használatával. A varázsló a DBMS_ADVISOR csomagot használja, amely elemző és tanácsadó függvényeket és procedúrákat tartalmaz.

Ezen kívül lehetőségünk van az SQL Access Advisor API használatára is.

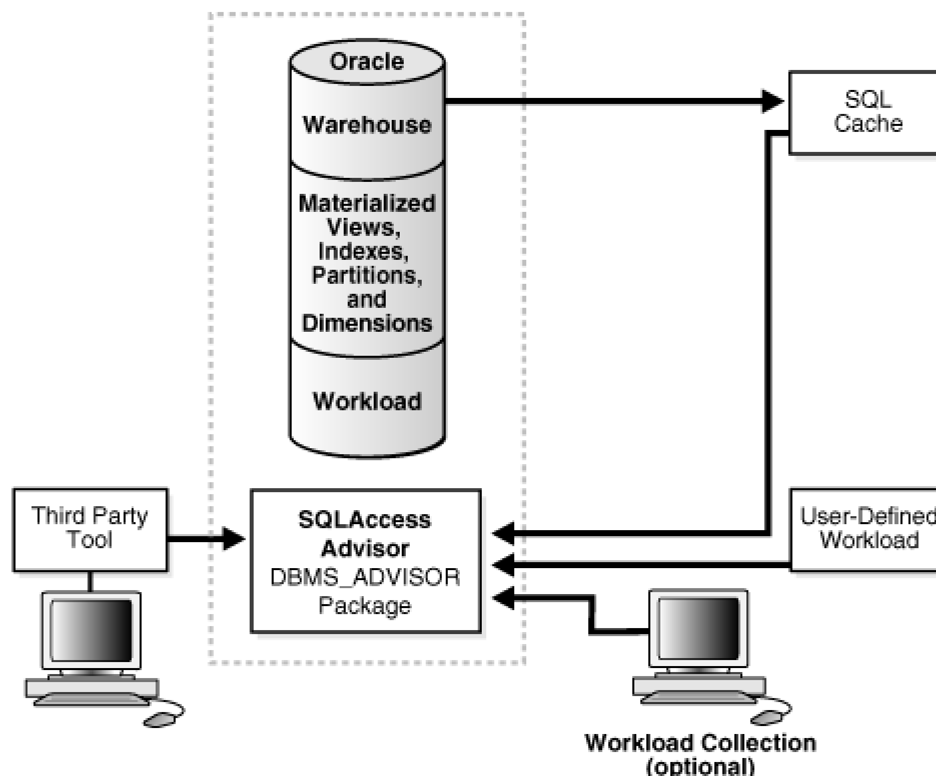
A 3. ábrán látható, hogy az Advisor a felhasználó által definiált, cache-ből kinyert terhelés alapján javasolja a hozzáférési struktúrákat. Ha nem áll rendelkezésünkre terhelés, akkor az eszköz képes hipotetikus terhelést generálni a felhasználó sémája alapján.

Ha az Oracle Enterprise Manageren (OEM) vagy az API-n keresztül használjuk az SQL Access Advisort, akkor az alábbi funkcionalitásokat érhetjük el:

- Felhasználó által definiált, cache-ből kinyert, vagy hipotetikus terhelés alapján tesz javaslatokat materializált nézetek és indexek létrehozására.
- Javaslatokat tesz táblák, indexek és materializált nézetek partícionálására.
- A meglévő hozzáférési struktúrák megtartására, módosítására vagy eldobására tesz javaslatokat.

Ha az API-n keresztül használjuk az SQL Access Advisort, akkor az OEM-hez képest további funkciókat érhetünk el:

- Gyors ajánlást tesz egy egyszerű SQL utasítás alapján.
- Megmutatja, hogy egy materializált nézetet hogyan tegyünk gyorsan frissíthetővé.
- Megmutatja, hogy egy materializált nézetet hogyan módosítsunk, hogy használható legyen általános lekérdezésátírásra (ekkor a lekérdezés átírásához nem szükséges szövegegyezés, ezzel több lekérdezés átírása valósulhat meg).



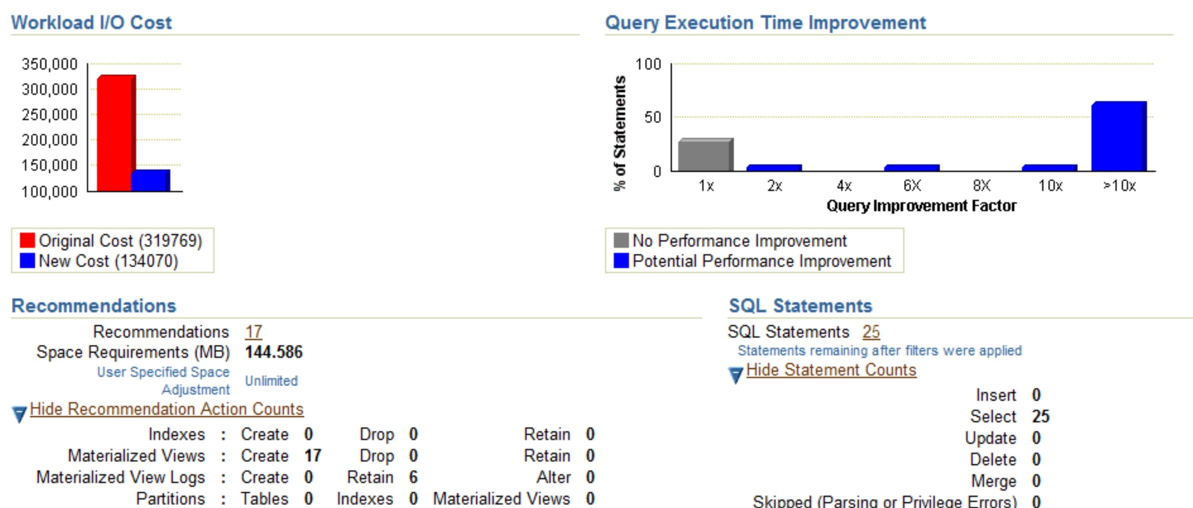
3. ábra. SQL Acces Advisor működése

Az SQL Access Advisor az javaslatok készítésekor, statisztikákra támaszkodik: a kapcsoló kulcsoszlopoknak, a ténytábla kulcsoszlopainak és a dimenzió szint oszlopainak számosságára. Ezeket a statisztikákat a DBMS_STATS PL/SQL csomag segítségével lehet pontosan mérni, vagy becsülni. Mivel a statisztikák gyűjtése időigényes, és a teljes statisztikai pontosság nem követelmény, ezért általában jobb a becslés használata. Ha egy adott táblához nem rendelkezünk a kellő statisztikákkal, akkor az erre a táblára vonatkozó lekérdezések a terhelésben érvénytelenné válnak, ennek eredményeképpen nem kapunk javaslatokat az adott lekérdezésekhez. Továbbá minden létező indexet és materializált nézetet analizálni kell, a további ajánlások készítéséhez.
Működés az Oracle Enterprise Manager-en keresztül:

A Varázsló végigvezet az SQL Access Advisor különböző beállításain.

- Meg kell adni, hogy milyen terhelés alapján szeretnénk dolgozni:
    - A terhelést kiolvassuk a cache-ből.
    - Egy hipotetikus terhelést használunk. Ehhez meg kell adni egy sémát, amelyre a lekérdezések hivatkoznak.
    - Egy SQL Tuning Set-et használunk.
- Milyen ajánlásokra van szükségünk:
    - Hozzáférési struktúrák (indexek, materializált nézetek, partíciók).
    - Érvényességi tartomány (minden lekérdezést, vagy csak a magas költségűeket vegye figyelembe).

A varázsló végén elindul az Advisor, majd lefutása után megtekinthetjük az általa tett javaslatokat (4. ábra). Egy összesítő lapot mutat, ahol megnézhetjük, hogy az javaslatok implementálásával milyen teljesítménynövekedést érhetünk el.



4. ábra. Az SQL Access Advisor javaslata

A további füleken további statisztikák érhetőek el, megnézhetjük, hogy a terhelésükben milyen lekérdezések voltak, és azokhoz milyen hozzáférési struktúrákat generált az advisor. Kilistázhatjuk, az advisor által generált hozzáférési struktúrákat, valamint be is ütemezhetjük ezek implementálását.

Az SQL Access Advisort segítségével materializált nézeteket generáltam egy adott SQL halmaz lekérdezési idejének növelésére. Az OEM felületen keresztül ehhez először létre kellett hoznom egy SQL Tuning Set-et, és abba kellett betennem a lekérdezéseimet. Az eszköz még nem teljesen kiforrott, több nehézségbe is ütköztem a végrehajtás során.

Mivel az Oracle nem publikálta az SQL Access Advisor algoritmusait, ezért csak a tapasztalataim alapján tudom az eszközt értékelni.

Saját tesztjeim alapján úgy látom, hogy az Access Advisor nem próbálja minimalizálni a materializált nézetek számát és méretét, az adott terhelésben szereplő lekérdezésekből csak a legegyszerűbb függőségek szerint vonja össze a materializált nézeteket (az azonos

alaptáblákra hivatkozó, azonos csoportosítási logikát tartalmazó materializált nézeteket vonja össze), így nagyszámú és nagyméretű materializált nézetet generál. Az advisor a lekérdezésekben szereplő szűréseket eldobja. Ennek hátránya, hogy a materializált nézetek mérete jelentősen nagyobb, mintha az eredeti szűrő feltételek is szerepelnének a nézetekben. A megoldás előnye, hogy általánosabb, több lekérdezést kiszolgáló nézetek jönnek létre.
Az SQL Access Advisor javaslatai a gyakorlatban általában nem használhatóak, a materializált nézetek mérete és száma miatt ugyanis kezelhetetlenné válna az adatbázis.

4. Összesítőtáblák az Oracle 11g adatbáziskezelőben

Az összesítőtáblákat az Oracle adatbázisban materializált nézeteknek (Materialized View) hívják.

4.1 Materializált nézetek

A materializált nézetek létrehozásakor a SELECT utasításrész definiálja azt az adathalmazt, amit a nézetnek tárolnia kell. Csak néhány megkötés korlátozza, hogy a SELECT utasítás mit tartalmazhat. Akárhány tábla összekapcsolható, és a táblák mellett más elemek, azaz nézetek, inline nézetek, allekérdezések és materializált nézetek is összekapcsolhatóak, vagy szerepelhetnek a SELECT utasításban.

Általános megkötések:

- A materializált nézetet definiáló lekérdezés nem tartalmazhat nem-ismétlődő kifejezéseket (ROWNUM, SYSDATE).
- A lekérdezés nem tartalmazhat referenciát RAW, LONG RAW adattípusokra.

Az adattárházak teljesítményének növelése érdekében hozunk létre materializált nézeteket, ezért be kell tartani bizonyos követelményeket, hogy a query rewrite használni tudja az általunk létrehozott materializált nézeteket.

Általános megkötések Query Rewrite használatához:

- Se a részletező táblák, se a materializált nézetek nem lehetnek a SYS felhasználó tulajdonában.
- Ha egy attribútum vagy kifejezés megjelenik a materializált nézet GROUP BY utasításrészében, akkor a SELECT listában is meg kell jelenniük.
- Az aggregáló függvények csak a kifejezések legkülső részeként jelenhetnek meg. Például az AVG(AVG(x)) vagy az AVG(x)+AVG(x) kifejezések nem megengedettek.
- Nem tartalmazhat a lekérdezés CONNECT BY utasításrészt.

A Materializált nézeteknek 3 fajtájuk van, aszerint, hogy mit tartalmaznak:

- Aggregátumokat tartalmazó materializált nézet.
- Csak kapcsolófeltételeket (joinokat) tartalmazó materializált nézet.
- Egymásba ágyazott materializált nézet (Nested materialized view).

A materializált nézetek hatékonyságának szempontjából fontos, hogy milyen karbantartási igényekkel jár a használatuk. A karbantartási igény alatt jelen esetben a materializált nézetek frissítését értjük. A frissítés költsége szempontjából két paramétert kell figyelembe venni:

1. Mikor frissítünk: ez a **Refresh Mode**, amelynek értéke lehet *On Commit* vagy *On Demand*. Az előbbi azt jelenti, hogy frissítés történik mindig, amikor egy tranzakció módosítja a Materializált nézet által hivatkozott egyik táblát és commitált (ez csak Gyors frissítés esetén alkalmazható). Utóbbi esetben a felhasználónak kell manuálisan frissítenie a nézetet.
2. Milyen módszerrel frissítsük a materializált nézetet: ezt a **Refresh Options** paraméter határozza meg, melynek értéke lehet *Complete*, *Fast*, *Force* vagy *Never*. Míg a

Complete esetében az egész materializált nézetet újraszámítja a rendszer, a *Fast* opció esetében csak az inkrementális változásokat kell figyelembe vennie, így jelentősen csökken a frissítés ideje. A *Force* opció esetén *Fast* opcióval próbálja frissíteni a materializált nézetet, és ha ez nem lehetséges, akkor *Complete*-tel. A *Never* opció használatakor nincs frissítés.

Bizonyos megkötések vonatkoznak a *Fase Refresh* opció használatára. Mivel optimális materializált nézet halmaz létrehozása a célom, aminek feltétele a kis karbantartási igény, ezért csak olyan materializált nézeteket hozok létre, amelyek teljesíteni fogják a *Gyors frissítés* mechanizmus alkalmazhatóságának a követelményeit.
Gyors frissítés használatakor a materializált nézet frissítése optimalizáltan hajtódik végre az elérhető elsődleges- és idegen kulcsok segítségével. Ezzel az elsődleges/idegen kulcs optimalizálással jelentősen csökken a frissítés költsége. Például, ha a materializált nézet egy joint tartalmaz egy ténytábla és egy dimenziótábla között, és csak a dimenziótáblába veszünk fel új sorokat, a ténytáblát viszont változatlanul hagyjuk, ilyenkor a materializált nézetet nem kell frissítenünk.
Ennek az az oka, hogy mivel a dimenziótábla kapcsolótáblája rendelkezik elsődleges kulcs kényszerrel, a ténytábla kapcsolótáblája pedig rendelkezik idegen kulcs kényszerrel, ezért ha a dimenziótáblába veszünk fel új sorokat, az új sorok nem fognak kapcsolódni egy ténytáblabeli sorhoz sem, így nincs semmi, amit frissíteni kellene.
A fenti megkötéseken kívül további egyszerűsítéseket vezettem be, amelyek tervezési megfontolásokból erednek:

- Nem használok egymásba ágyazott lekérdezéseket (ezekkel kapcsolatban hatékonysági problémák lépnek fel, nem használható a fast refresh opció a frissítésük során).
- Csak helyi adatbázissal dolgozom, távoli táblákat nem használok.

Általános megkötések Gyors frissítés használatához:

- A SELECT utasításrész nem tartalmaz:
  - allekérdezéseket;
  - analitikus függvényt (például, RANK);
  - MODEL utasításrészt;
  - egymásba ágyazott lekérdezéseket, amelyek az ANY, ALL, EXSIST feltétellel kapcsolódnak;
  - START WITH … CONNECT BY utasításrészt;
- HAVING utasításrész nem tartalmaz beágyazott lekérdezést.

Megkötések csak Joinokat tartalmazó Materializált Nézetek esetén:

- *Az általános megkötések Gyors frissítés használatához*.
- Minden tábla azonosítójának szerepelnie kell a SELECT listában.
- A materializált nézet log-ok elérhetőnek kell legyenek, és tartalmazniuk kell a FROM listában szereplő összes alaptábla azonosítóját.

A frissítési módszer nem lesz hatékony, ha:

- A definiáló lekérdezés outer joint használ, amelynek működése megegyezik az inner join-éval (ekvivalensek, például, elsődleges/idegen kulcs szerint kapcsolódnak). Ebben az esetben javasolt az inner join használata.
- A SELECT listában több táblára hivatkozó kifejezések vannak.

Megkötések aggregátumokat tartalmazó Materializált Nézetek esetén:

- *A megkötések csak Joinokat tartalmazó Materializált Nézetek esetén*
- Csak a SUM, COUNT, AVG, STDDEV, VARIANCE, MIN, és MAX aggregátumok szerepelhetnek benne.
- A COUNT(*)-ot specifikálni kell.
- Az aggregáló függvények csak a kifejezések legkülső részeként jelenhetnek meg. Például az AVG(AVG(x)) vagy az AVG(x)+AVG(x) kifejezések nem megengedettek.
- Egyes aggregátumok mellé fel kell venni a materializált nézetbe a vele megegyező más csoportosításokat. Ezeket a 5. ábra tartalmazza: ha az X oszlopbeli aggregátumot szeretnénk tárolni, akkor az Y oszlopbeli aggregátum kötelező, a Z oszlopbeli pedig választható.

X	Y	Z
COUNT(expr)	-	-
MIN(expr)		
MAX(expr)		
SUM(expr)	COUNT(expr)	-
SUM(col), col has NOT NULL constraint	-	
AVG(expr)	COUNT(expr)	SUM(expr)
STDDEV(expr)	COUNT(expr) SUM(expr)	SUM(expr * expr)
VARIANCE(expr)	COUNT(expr) SUM(expr)	SUM(expr * expr)

5. ábra. Kötelező aggregátumok [4]

Meg kell jegyezni, hogy a COUNT(*)-nak mindig meg kell jelennie a gyors frissítés használatához, különben csak az insert típusú változások után lehet alkalmazni ezt a frissítési mechanizmust. Ajánlott az Y oszlopbeli aggregátumot is felvenni a materializált nézetbe, hogy elérhessük a leghatékonyabb, és legpontosabb frissítést.

- A SELECT listában meg kell jelennie minden, a GROUP BY listában szereplő oszlopnak.
- A materializált nézet minden táblájának rendelkeznie kell Materialized View Log-gal,

és a Log-nak teljesíteniük kell, a következő kritériumokat:

- Minden oszlopot tartalmaz, amelyre a materializált nézetben hivatkoznak;
- ROWID és INCLUDING NEW VALUES opciókkal kell létrehozni őket;
- SEQUENCE opcióval kell létrehozni, ha várhatóan nem csak beszúrni fogunk a táblákba (direct load, delete, update);

- A CUBE, ROLLUP és GROUPING SETS használata esetén az alábbi megkötéseknek eleget kell tenni:
  - A materializált nézetnek tartalmaznia kell egy csoportosítási azonosítót, azaz egy GROUP_ID függvényt a GROUP BY minden kiterjesztése esetén. Például a GROUP BY kitétele a lekérdezésnek GROUP BY CUBE(a, b), akkor a projekcióban szerepelnie kell a GROUPING_ID(a, b)-nek.
  - A materializált nézet GROUP BY utasításrésze nem tartalmazhat többször egy csoportosítást. Például a GROUP BY a, ROLLUP(a, b) kitételt tartalmazó materializált nézet nem frissíthető a gyors frissítés módszerrel, mert az eredményhalmaz az alábbi csoportosításokat tartalmazza: (a), (a,b), (a)

4.2 Tervezési megfontolások a Query Rewrite képességeinek növelése érdekében

4.2.1 Megkötések

Referenciális integritást kell biztosítani a materializált nézetek által kapcsolt táblákhoz, hogy a lekérdezésátírás minden esetben képes legyen átirányítani a lekérdezéseket. Más szóval a kapcsolt tábláknak elsődleges/idegen kulcs kapcsolatban kell állniuk, valamint az idegen kulcsot NOT NULL kényszerrel kell ellátni. Ebben az esetben az inner join veszteségmentes és nem többszöröző lesz. Ez esetünkben nem probléma, hiszen adatpiacok esetén feltételezhető a csillag sémájú adatmodell, amelyben csak tény és dimenziótáblák kapcsolata jelenik meg.

4.2.2 Dimenziók

Hierarchikus kapcsolatokat és funkcionális függéseket lehet létrehozni dimenzió táblák alkalmazásával. A dimenziók képesek olyan táblán belüli kapcsolatokat kifejezni, amelyeket a kényszerek nem képesek. E kapcsolatokat a lekérdezésátirányító fel tudja használni a lekérdezések átírására.

4.2.3 Külső összekapcsolások (Outer Joins)

Ha a materializált nézetekben outer joinokat használunk, ezekkel kiválthatjuk a kényszereket. Ugyanis a lekérdezésátírás ki tudja nyerni a belső összekapcsolást ezekből a lekérdezésekből is, azaz (A.a = B.b) levezethető az (A.a = B.b(+)) összekapcsolásból, ha a materializált nézet tartalmazza a B ROWID-jét vagy a B.b attribútumot. Az outer joinok használata mégsem javasolt, mert a gyors frissítés módszer hatékonyságát csökkenti.
Ezek alapján úgy döntöttem, hogy indokolt esetben outer joint használok. Például: ha két materializált nézetet szeretnék összevonni, az egyikben inner a másikban outer joinnal

kapcsolódnak össze ugyanazok a táblák, akkor a lekérdezésátírás minél szélesebb körű használhatósága érdekében a közös táblában outer joinnal kapcsolom az adott táblákat.

4.2.4 Szöveges illesztés

Ha egy nagyon komplex, hosszú futásidejű lekérdezést kell felgyorsítanunk, akkor a materializált nézetet a lekérdezés pontos másával érdemes létrehozni. A létrehozás után a materializált nézet tartalmazni fogja a lekérdezés eredményét, így megtakaríthatjuk azt az időt, amely a táblák összekapcsolásához, és az összes adat feldolgozásához kellene, amelyekkel megkaphatjuk a szükséges információkat. Ad-hoc lekérdezések esetén a szövegegyezéses lekérdezésátírás nem járható út, hiszen nem tudhatjuk előre, hogy melyek lesznek a nagy terhelésű lekérdezések. Ehelyett olyan általános materializált nézeteket érdemes létrehozni, amelyek több lekérdezés átírására is használhatók.

4.2.5 Aggregátumok

A lekérdezésátirányítás hatékonyságának érdekében a materializált nézeteknek tartalmazniuk kell minden olyan aggregátumot, amelyek szükségesek lesznek a kiválasztott lekérdezések kiszámításához. Például az AVG(x) használata helyett a COUNT(x) és a SUM(x) aggregátumokat hoztam létre a materializált nézetekben. Ezen aggregátumok használata esetén a materializált nézetek szélesebb körben alkalmazhatók lekérdezésátírásra, hiszen segítségükkel előállítható a felsorolt aggregátumok mindegyike.

4.2.6 Csoportosítási logikák

Az adatok alacsonyabb szinten való aggregálása jobb, mint a magasabb szinten lévő aggregálás, mert az alacsonyabb szinttel több lekérdezés irányítható át. Ennek viszont az a hátránya, hogy a materializált nézet több helyet foglalni, és a lekérdezések is lassabban futnak, mert adott esetben aggregálási műveleteket kell végrehajtsanak az összesítőtáblák fölött.

Ahelyett, hogy több olyan materializált nézetet készítenénk, amelyek hierarchikusan összefüggő, vagy egymást átfedő oszlopokat csoportosítanak, érdemes inkább egy közös materializált nézetet készíteni, amely tartalmazza az összes attribútum szerinti csoportosítást. A csoportosítások létrehozásának több módja is van, ezeket az 5.1.3. fejezetben részletezem.

5. Megvalósítás

Az általam kidolgozott megoldás adott terhelés alapján képes összesítőtáblák ajánlására, ennek megfelelően bemenetként meg kell adni egy lekérdezés halmazt, amelyeket szeretnénk felgyorsítani materializált nézetek segítségével (ez a halmaz származhat a cache-ből, statisztikák és mérések segítségével választható ki, vagy a rendszerből kinyerhető adatbányászati módszerekkel). Ezt a lekérdezéshalmazt tekintem az ad-hoc lekérdezések alapjául, hiszen feltételezhető, hogy az azonos adatpiaci rétegen dolgozó üzleti elemzők hasonló lekérdezéseket fognak készíteni.
A lekérdezések feldolgozása után előállítjuk az első materializált nézet halmazt, majd ezen a halmazon végzett műveletek segítségével egy fát építünk. A fa minden csomópontja materializált nézetek egy halmazát tartalmazza, a fa bejárása során kell eldöntenünk, hogy mely halmazt választjuk.
A materializált nézetekkel való optimalizálás költségeken alapszik, ezért mindig egy költségalapú választással kell eldönteni, hogy mely halmazokat érdemes megtartani, és melyeket nem.

A költségek értékelési szempontjai a következőek:

- Tárhelyigény
- Karbantartási igény
- Elérhető sebességnövekedés

Első lépésként a feldolgozott lekérdezésekből készítek egy materializált nézet halmazt, amelyek szövegillesztéses alapon tudnak együttműködni a lekérdezésekkel. Ezután a materializált nézet halmazt próbálom meg minimalizálni, az elemek összevonásaival, elemek elhagyásával.

5.1 A keresőfa felépítése és bejárása

Ebben a fejezetben bemutatom, hogy a bemenetként kapott terhelésből hogyan építhető fel a keresőfa, valamint a fa segítségével hogyan lehet megtalálni a materializált nézetek azon optimális halmazát, amely hatékonyan ki tudja szolgálni az adott terhelést.

A fa gyökér csomópontjába vesszük fel azokat a materializált nézeteket, amelyeket a kiszolgálandó lekérdezésekből generáltunk. Az algoritmus a gyökérből indul, innen kezdi el bejárni a fát a mohó algoritmus.
Egy n darab materializált nézetet tartalmazó csomópont kibontása (n-1) elemű csomópontokat hoz létre (ha a materializált nézetek között nincsenek olyanok, amelyek azonos alaptáblákra hivatkoznak), az alábbi módon:

- Páronként összevonja (uniós összevonás) a materializált nézeteket (a materializált nézetek összevonási lehetőségeit, az összevonások értékelését az 5.2 fejezetben ismertetem), minden összevonás során új csomópontot hozva létre. A csomópontba az új materializált nézetet veszi fel, és azokat, amelyek nem voltak az összevonás alanyai.
- Páronként összevonja (metszetes összevonás) a materializált nézeteket, minden összevonás során új csomópontot hozva létre. A csomópontba az új materializált nézetet veszi fel, és azokat, amelyek nem voltak az összevonás alanyai.
- Létrehozza azokat a csomópontokat, amelyek 1-1 materializált nézet kihagyásával keletkeznek.

Az így kibontott csomópontnak $\binom{n}{2} \cdot 2 + n$ gyereke lesz.
Ha a gyökér csomópont n elemű (azaz a lekérdezések száma n), akkor az így felépített fa mélysége n.

A fa csomópontjait egy költségfüggvénnyel értékelem, az i. csomópont (g_i) a költségei az alábbiak:

- Tárhely

$$C_{storage}(MV_j) = \text{Size}(\ MV_j)$$

$$Total_{storage} = \sum_{k=1}^{|g_i|} C_{storage}(MV_k)$$

- Karbantartási

$$Total_{maint} = |g_i|$$

- Lekérdezési idő

$$C_{time}(Query_j) = \text{Time}(\ Query_j)$$

$$Total_{time} = \sum_{k=1}^{n} C_{time}(Query_k)$$

A költségfüggvény:

$$Total_{costs} = s_s \cdot Total_{storage} + s_m \cdot Total_{maint} + s_t \cdot Total_{time}$$

Az s_s a tárolási költség súlya, az s_m a karbantartási költség súly, az s_t a lekérdezési költség súlya. Ennek az összegnek a minimalizálása a cél, azaz egy minimális költségű, minimális karbantartási igényű materializált nézet halmaz létrehozása a feladat, ami mellett a lekérdezési idők is minimálisak lesznek.

A fa bejárásához egy mohó algoritmust készítettem, amely a helyi optimumok kiválasztásával keres egy jó materializált nézet halmazt. Ez a megoldás nem garantálja az abszolút optimum megtalálását, de ennek elérése esetünkben nem lehet cél, hiszen az összesítőtáblák optimalizálása NP-n kívül esik (vagyis nem létezik polinom időben kiértékelhető kritérium az optimum minősítésére).

Részletes algoritmus (6. ábra):

1. A gyökércsomópont létrehozása: a gyökércsomópontot a kiszolgálandó lekérdezésekből generáljuk, a lekérdezések szövegét át kell alakítani úgy, hogy az általános lekérdezésátírás követelményeinek megfeleljen. Ha a materializált nézeteket a Fast Refresh opcióval szeretnénk frissíteni, akkor ennek a követelményeinek is eleget tevő materializált nézeteket generálunk.

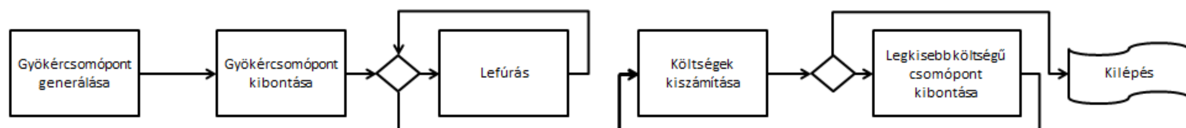
2. A gyökércsomópont kibontása: létre kell hozni azokat a csoportokat, amelyek a gyökércsomópont 1-1 elemének összevonásával vagy kihagyásával keletkeztek. A generálás után nem kell a csoportokat értékelni, egy irányított választás segítségével fogunk a fában haladni.

3. Irányított választás segítségével, mélységi keresés a fában: azokat a materializált nézeteket vonja össze, amelyek egyazon alaptáblákat választják ki. A gyakorlati

tapasztalatok azt mutatják, hogy a keresés során jó materializált nézet halmazok jönnek létre, ezeket a lépéseket a mohó algoritmus is megtette volna. Az irányítás segítségével jelentősen gyorsítható az alkalmazás futásideje, hiszen nem szükséges a csomópontok értékelése az új kiválasztásához.

4. Ha nincs az adott feltételnek megfelelő csomópont, azaz nincs több olyan materializált nézet, melyek azonos alaptáblákra hivatkoznának, akkor a mohó algoritmust kell használni a fa bejárásához. Ehhez a költségfüggvény segítségével kell értékelnünk a legalacsonyabb szinten lévő csomópontokat.

5. Az értékelt csomópontok közül kiválasztjuk a legkisebb költségűt. Ha ennek értéke kisebb, mint a szülő csomópontjának a költsége, akkor kibontjuk a csomópontot, és visszalépünk a 4. lépésre. Ha nagyobb, akkor megtaláltunk egy jó csomópontot. A megtalált csomópont egy jó materializált nézet halmaz lesz, amely hatékonyan ki tudja szolgálni a rendszerbe érkező ad-hoc lekérdezéseket.



6. ábra. Részletes algoritmus a fa építésére és bejárására

5.2 Megfontolások materializált nézetek összevonhatóságára

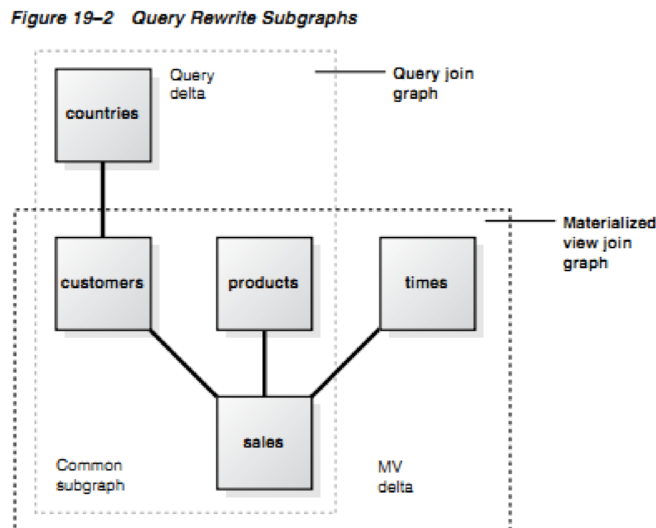
Ebben a fejezetben két típusú összesítőtábla összevonhatóságát vizsgálom, a csak joinokat tartalmazó és az aggregátumokat is tartalmazó tábláké t. Csak azonos típusú összesítőtáblák összevonására van lehetőség, nem lehet aggregátumokat is tartalmazó nézeteket csak join típusúakkal összevonni. A továbbiakban az összevonási lehetőségeket a kapcsolt táblák szerint vizsgálom.

Különbségek lehetnek a materializált nézetek között:

- az attribútumoknak nem ugyanazon részhalmazai vannak kiválasztva (projekció);
- nem ugyanazokat az attribútumokat aggregálják;
- az attribútumokat nem ugyanúgy aggregálják (AVG, COUNT, SUM…);
- különböző szűrések vannak beállítva (WHERE, HAVING);
- nem ugyanazokat a táblákat kapcsolják össze;

Az összekapcsolt táblák szerinti Query Rewrine-ra 3 lehetőség van (7. ábra):

- **Common joins:** a lekérdezések ugyanazokat a táblákat kapcsolják össze.
- **Query Delta joins:** a materializált nézet a lekérdezés által igényelt tábláknak csak egy részhalmazát tartalmazza.
- **Materialized View Delta joins:** a materializált nézet több táblát kapcsol össze, mint amennyire a lekérdezésnek szüksége lenne.



7. ábra. Összekapcsolási lehetőségek [4]

5.2.1 Koncepció

Ebben a fejezetben a materializált nézetek összevonhatóságát, valamint különböző összevonási módjait mutatom be részletesen.

Legyen Aq és Bq 2 lekérdezés, amelyek összevonhatóságát fogom vizsgálni. Az első lekérdezés által kapcsolt táblák halmaza A, a második lekérdezés által kapcsolt táblák halmaza B.

1, Ha a lekérdezések ugyanazokat a táblákat kapcsolják össze, azaz A = B, akkor a materializált nézeteik összevonhatóak.

1.1 Csak Joinokat tartalmazó materializált nézetek esetén (tehát nem tartalmaz aggregátumot)

Az összevonás módszere (1. Függelék):

SELECT (az eredeti projekciók uniója + **azok az attribútumok, amelyekre feltételeket fogalmaztunk meg**)
FROM (a hivatkozott táblák – ezek megegyeznek)
WHERE (a hivatkozott táblák kapcsolásai, és a közösen meghatározott szűrő feltételek)

Értékelés:

Ez a megoldás nagyobb tárigénnyel jár, mint az egyes materializált nézetek, de mivel köztük redundancia fordulhat elő, ezért kevesebb tárhelyet várunk, mint az egyes materializált nézetek helyfoglalásának összege. Mindemellett nem jelentkezik számottevő sebességcsökkenés. A karbantartási költségek viszont felére csökkennek, hiszen ugyanazokat a táblákat kell összekapcsolni, valamint a kapcsolás során kiválasztani a megfelelő oszlopokat, amelyekre szükségünk lesz.
Az összevonás javasolt.

1.2 Aggregátumokat is tartalmazó materializált nézet esetén

Az összevonás módszere (2. Függelék):

SELECT (az eredeti projekciók uniója + **azok az attribútumok, amikre feltételeket fogalmaztunk meg**)
FROM (a hivatkozott táblák – ezek megegyeznek)
WHERE (a hivatkozott táblák kapcsolásai, és a közösen meghatározott szűrő feltételek)

GROUP BY (az eredeti csoportosítások összevontja+ **azok az attribútumok, amikre feltételeket fogalmaztunk meg**)
HAVING (a közösen meghatározott szűrő feltételek)

Értékelés:
Ez a megoldás jelentősen nagyobb tárigénnyel jár, mint az egyes materializált nézetek.
Megoldás lehet a Grouping Set-ek használata, ebben az esetben a tárhely nőni fog, de nem haladja meg jelentősen az eredeti materializált nézetek méreteinek az összegét. A karbantartásigény a felére csökken, míg a hatékonyság pedig nem csökken számottevően.

2, Ha A–nak és B-nek van közös része, akkor az összevonásra két lehetőség van:

1. *Egy bővebb materializált nézetet készítünk a 2 materializált nézet uniójával.*
2. *Egy szűkebb materializált nézetet készítünk a 2 materializált nézet metszetével.*

2.1. Eset:
Az összevonás feltétele:
Az első lépés, amit figyelembe kell venni, a join veszteségmentessége.
Egy Join veszteségmentes (**Lossless**), ha

- Elsődleges/idegen kulcs párost joinolunk (és az idegen kulcs NOT NULL).
- a kapcsolt tábla megfelelő attribútuma NOT NULL kényszerrel van ellátva.
- Outer Joinról van szó.

A veszteségmentesség azonban nem elég. Azt is biztosítanunk kell, hogy minden rekordhoz a kapcsolt tábla egy sorát rendeljünk hozzá.
Egy join nem többszöröző (**Non-duplicating**), ha

- Elsődleges/idegen kulcs páros-t joinolunk.
- a kapcsolt tábla megfelelő attribútuma egyedi kulcs kényszerrel van ellátva.

A és B lekérdezés esetén is **Materialized View Delta join** típusú lekérdezésátírással fogjuk megkapni az eredményt. Azaz a lekérdezésekből csak egy a materializált nézetre irányuló SELECT keletkezik.

2.1.1 Csak Joinokat tartalmazó materializált nézetek esetén (tehát nem tartalmaz aggregátumot)
Az összevonás módszere (3. Függelék):

SELECT (az eredeti projekciók uniója + **azok az attribútumok, amikre feltételeket fogalmaztunk meg**)
FROM (az **A** U **B** táblák)
WHERE (az **A** U **B** táblák kapcsolásai, és a közösen meghatározott szűrő feltételek)

Értékelés:
Ez a megoldás nagyobb tárigénnyel jár, mint az egyes materializált nézetek, de mivel köztük redundancia fordulhat elő, ezért kevesebb tárhelyet várunk, mint az egyes materializált nézetek helyfoglalásának összege. Mindemellett nem jelentkezik számottevő sebességcsökkenés. A karbantartási költségek viszont jelentősen csökkenhetnek, hiszen részben ugyanazokat a táblákat kell összekapcsolni, valamint a kapcsolás során kiválasztani a megfelelő oszlopokat, amelyekre szükségünk lesz.

2.1.2 Aggregátumokat is tartalmazó materializált nézet esetében

Az összevonás módszere (4. Függelék):

SELECT (az eredeti projekciók uniója + **azok az attribútumok, amikre feltételeket fogalmaztunk meg**)
FROM (az **A** U **B** táblák)
WHERE (az **A** U **B** táblák kapcsolásai, és a közösen meghatározott szűrő feltételek)
GROUP BY (a csoportosítások uniója + **azok az attribútumok, amikre feltételeket fogalmaztunk meg**)
HAVING (a közösen meghatározott szűrő feltételek)

Értékelés:
Ez a megoldás jelentősen nagyobb tárigénnyel jár, mint az egyes materializált nézetek helyfoglalásának összege.
Megoldás lehet a GROUPING SET-ek használata. Ebben az esetben a tárhely nőni fog, de nem haladja meg jelentősen az eredeti materializált nézetek méreteinek az összegét. A karbantartás viszont a felére esik., a hatékonyság pedig nem csökken számottevően.

2.2. Eset:
A és B lekérdezés esetén is **Query Delta Joins** lekérdezésátírással fogjuk megkapni az eredményt. Azaz a lekérdezések a materializált nézetet fogják összekapcsolni a hiányzó táblákkal.

2.2.1 Csak Joinokat tartalmazó materializált nézetek esetében (tehát nem tartalmaz aggregátumot)
Az összevonás módszere:

SELECT (az eredeti projekciók uniója, de csak azokkal az attribútumokkal amelyeket, az **A ∩ B** halmaz táblái tartalmaznak **+ a kimaradt táblák kapcsolókulcsai (Foreign key-ek) + azok az attribútumok, amikre feltételeket fogalmaztunk meg**)
FROM (az **A ∩ B** halmaz táblái)
WHERE (az **A ∩ B** által hivatkozott táblák kapcsolásai, és a közösen meghatározott szűrő feltételek)

Értékelés:
Ez a megoldás jelentősen csökkenti a tárigényt, az eredeti materializált nézeteknél is kevesebb helyet foglal, viszont jelentős sebességcsökkenéssel jár az eredeti materializált nézetekhez képest. A karbantartási igény jelentősen csökken, több mint felére esik.

2.2.2 Aggregátumokat is tartalmazó materializált nézetek esetén
Az összevonás módszere (Függelék 2.2.2):

SELECT (az eredeti projekciók uniója, de csak azokkal az attribútumokkal amelyeket, az **A ∩ B** halmaz táblái tartalmaznak **+ a kimaradt táblák kapcsolókulcsai (Foreign key-ek) + azok az attribútumok, amikre feltételeket fogalmaztunk meg**)
FROM (az **A ∩ B** halmaz táblái)
WHERE (az **A ∩ B** által hivatkozott táblák kapcsolásai, és a közösen meghatározott szűrő feltételek)
GROUP BY (a csoportosítások uniója, de csak azokkal az attribútumokkal amelyeket, az **A ∩ B** halmaz táblái tartalmaznak **+ a kimaradt táblák kapcsolókulcsai (Foreign key-ek) + azok az attribútumok, amikre feltételeket fogalmaztunk meg**)
HAVING (a közösen meghatározott szűrő feltételek)

Értékelés:
Ez a megoldás jelentős sebességcsökkenéssel jár az eredeti materializált nézetekhez képest. A karbantartási igény jelentősen csökken, több mint felére esik. A tárigény tekintetében nem tudni, hogyan fog változni, de biztosan kevesebb területre lesz szükség, mintha megtartottuk volna az eredeti táblákat.

3, Ha A –nak és B-nek nincs közös táblájuk, akkor nincs értelme a táblák összekapcsolásának.

5.2.2 A közös szűrőfeltételek meghatározása

Materializált nézetek összevonásánál figyelembe kell venni, hogy az egyes materializált nézetek definíciójában milyen szűrő feltételek találhatók.
Szűrőfeltételek fogalmazhatók meg az oszlopokra vonatkozóan (WHERE) valamint az aggregátumokra vonatkozóan (HAVING). A szűrőfeltételeket az alábbi relációs operátorok (relop) segítségével fejezhetjük ki: =, <, <=, >, >=, !=, [NOT] BETWEEN | IN| LIKE |NULL. Ezek a lekérdezésekben <bal_oldali_kifejezés> *relop* <jobb_oldali_kifejezés> alakban jelennek meg, ahol a bal oldalon általában egy oszlop neve, a jobb oldalon pedig egy érték áll. Például a color = ’green’ feltételben a color a bal_oldali_kifejezés (oszlop), a ’green’ a jobb_oldali_kifejezés (érték), az (=) pedig a relációs operátor.
A kiválasztásokat különböző kategóriákba sorolhatjuk

- Simple

  Az egyszerű szűrések az alábbi alakúak: <kifejezés> relop <konstans>

- Complex

  A komplex szűrések az alábbi alakúak: <kifejezés> relop <kifejezés>

- Range

  A tartományra való szűrések az alábbi alakúak: cust_last_name BETWEEN ’abacrombe’ AND ’anakin’

  De ezen kívül az <, <=, >, >= operátorokkal megadott predikátumokat is ide soroljuk.

- IN-lists

  Egy- vagy többoszlopos listák, pl: prod_id IN (102, 103, 104, 105)

  Ezen kívül az alábbi típusú szűrések is ide tartoznak: column1 = ’c1’, column1 = ’c2’

- IS [NOT] NULL

- [NOT] LIKE

- Egyéb.

  Ide soroljuk azokat a szűréseket, amelyeknek nem lehet meghatározni a határait.
  Például: ALL, ANY EXISTS…

A WHERE utasításrészben megfogalmazott feltételeket két részre kell bontani. Egyik az általános szűrőfeltételek, amikor az eredményhalmazunk csak egy feltétel szerinti részhalmazát szeretnénk kiválasztani. A másik az illesztések, ami a táblákat valamilyen kulcs szerint kapcsolja össze.
A materializált nézetek összevonásakor meg kell vizsgálni, hogy milyen lehetőségeink vannak az egyes materializált nézetek WHERE utasításrészében megfogalmazott feltételek összevonására.

A legegyszerűbb megközelítés szerint az általános szűrőfeltételeket elhagyjuk, így sokkal általánosabb materializált nézeteket hozhatnánk létre, de a szűrő feltételek elhagyásával a tárhely szükséglet jelentősen megnőhet. Ad-hoc lekérdezéseket kiszolgáló rendszerek esetén ez a jó megközelítés. A megvalósítás során a szűrő feltételeket elhagytam, a gyakorlat azt mutatja, hogy a lekérdezésekre általában egymástól eltérő szűrőfeltételeket fogalmaznak meg, ezért nem érdemes ezeket meghagyni. Az üzleti elemzők más-más időszakokra, területi egységekre, vásárolói csoportokra, termékekre, termékkategóriákra vonatkozó összesítésekre kíváncsiak, ezeket az igényeket a szűrő feltételek megtartásával nem lehetne kielégíteni.
Az illesztéseket az összevonás típusa szerint (uniós, metszetes) kell kezelni, a közös materializált nézetbe felvett táblákra vonatkozó kapcsolásokat meg kell tartani.

5.2.3 A közös csoportosítások meghatározása

Materializált nézetek összevonásánál meg kell fontolni, hogy az összevonás során előállt materializált nézet, milyen csoportosítási logikát alkalmazzon. A GROUP BY kiterjesztéseinek hatékonyságnövelő tulajdonságai miatt ezeket lehetőség szerint érdemes megtartani.
A GROUP BY GROUPING SET, CUBE, ROLLUP kiterjesztései segítségével a lekérdezés és riportkészítést lehet egyszerűbbé tenni. Ezek egy egyszerű eredményhalmazt adnak vissza, amely egyenértékű a különbözőképpen csoportosított (GROUP BY) sorok uniójával (UNION ALL). A ROLLUP kiszámítja az aggregátumokat (SUM, COUNT, MAX, MIN, AVG) az aggregálás különböző szintjein, a legrészletesebbtől a teljes összegig. A CUBE hasonló, mint a ROLLUP, de ez a kiterjesztés elkészíti az adott oszloplista összes lehetséges kombinációja szerinti csoportosítást. A GROUPING SETS segítségével kiválasztással specifikáljuk a csoportosítást a GROUP BY kifejezésen belül.
Ez a 3 kiterjesztés segítségével hatékony többdimenziós lekérdezéseket tudunk készíteni anélkül, hogy kockába kellene rendezni az adatainkat. Az adatkockákkal való számítás nagy terhelést jelent a relációs adatbázisok számára, így GROUPING SET-ek segítségével jelentősen növelhetjük a rendszerünk teljesítményét.
A teljesítmény növelése érdekében a CUBE, ROLLUP, GROUPING SETS párhuzamosan végrehajthatók: egyidejűleg több folyamat tudja ezeket az utasításokat végrehajtani. Ez a képesség az aggregátum számítást hatékonyabbá teszi, ezzel növelve az adatbázis teljesítményét és skálázhatóságát [4].
Például:

```
(…)
GROUP BY (p.prod_subcategory, t.calendar_month_desc)
(…)
GROUP BY (c.cust_city, p.prod_subcategory)
```

A base grouping halmaz a csoportosítás egyedi elemeit tartalmazza. A jelen esetben a base grouping a (p.prod_subcategory, t.calendar_month_desc, c.cust_city) oszlopok halmaza.

Csoportosítások összevonási lehetőségei:

Legegyszerűbb esetben eldobhatjuk a GROUP BY–ok kiterjesztéseit és a base grouping halmazok unióját alkalmazzuk csoportosításként. Ez jó eredményt fog adni számunkra, a lekérdezésátírás elő tudja állítani a base groupig csoportosításból bármely összevont csoportosítási típust, ha azok a base grouping halmaz elemeit tartalmazzák. Ezzel a megoldással viszont elveszítjük a kiterjesztések előnyeit, a párhuzamos végrehajtást, ráadásul jelentős többlet tárigénnyel járhat.
A másik lehetőség, hogy meghagyjuk az egyes GROUP BY kiterjesztéseket, és a közös materializált nézet ezek mindegyikét tartalmazni fogja. Hogy a lekérdezésátírás egy GROUP

BY kiterjesztést tartalmazó materializált nézetet használni tudjon, két feltételnek teljesülnie kell:

- A materializált nézetnek tartalmaznia kell egy csoportosítási azonosítót, azaz egy GROUP_ID függvényt a GROUP BY minden kiterjesztése esetén. Ha például a a lekérdezésben GROUP BY CUBE(a, b, c, d) szerepel, akkor a projekcióban szerepelnie kell a GROUPING_ID(a, b, c, d)-nek.

- A materializált nézet GROUP BY része nem tartalmazhat többször egy csoportosítást. Például a GROUP BY GROUPING SET((a, b), (a, b)) részt tartalmazó materializált nézetet nem tudná használni a lekérdezésátírás.

Az összevonás során a 2. résszel vannak a nagyobb problémák. Például a GROUP BY CUBE(a, b) és a GROUP BY ROLLUP(a, b, c) között redundancia van, ami sérti ezt a szabályt. Ez ráadásul általában nem is vehető észre ránézésre, ezért a CUBE és ROLLUP kiterjesztéseket mindig át kell alakítani GROUPING SET alakra, majd megvizsgálni a köztük lévő egyezőségeket. Ha ilyet nem találunk, akkor az eredeti alakban visszaírhatók a közös materializált nézetbe. Ha találunk egyezőséget, akkor azt meg kell szüntetni, és a felbontott alakban kell beírni a közös materializált nézetbe.

Megoldások összehasonlítása

A materializált nézetek által használt csoportosítások:

(A1, A2, ..., An, C1, C2, ..., Cn)
(B1, B2, ..., Bn, C1, C2, ..., Cn)

Első megoldás:

Az összevont alakjuk: *(A1, A2, ..., An, B1, B2, ..., Bn, C1, C2, ..., Cn)*
A sorok hossza: *(A1, A2, ..., An, B1, B2, ..., Bn, C1, C2, ..., Cn)*
A sorok száma: *(|A1|*|A2|*...*|An|*|B1|*|B2|*...*|Bn|*|C1|*|C2|*...*|Cn|)*

Második megoldás:

Az összevont alakjuk: *((A1, A2, ..., An, C1, C2, ..., Cn), (B1, B2, ..., Bn, C1, C2, ..., Cn))*
A sorok hossza: *(A1, A2, ..., An, B1, B2, ..., Bn, C1, C2, ..., Cn)*, de sok NULL elem lesz, amelyek csak minimális tárhelyet igényelnek.
A sorok száma: *(|A1|*|A2|*...*|An|*|C1|*|C2|*...*|Cn|)* + *(|B1|*|B2|*...*|Bn|*|C1|*|C2|*...*|Cn|)*

A fentiekből látszik, hogy általánosságban a második megoldás a jobb, hiszen ebben az esetben kevesebb sort fog tartalmazni a materializált nézetünk.

Teszt egy konkrét példán:

A **2. Függelék**-ben lévő lekérdezéseket vizsgáltam meg.
Az összevonás során a base grouping csoportosítási logikát alkalmazva:
Sorok száma: 179279.
Materializált nézet mérete: 4062 Kb.

Az összevonás során a szétválasztós csoportosítási logikát alkalmazva:
Sorok száma: 177297, ami pontosan a két eredeti materializált nézet sorainak összege.
Materializált nézet mérete: 2514 Kb.

A mérésből az látszik, hogy bár a „base grouping” esetén a sorok száma nem sokkal nagyobb, mint a GROUPING SET-es megoldás esetében, a mérete viszont jelentősen nagyobb.

Lehetőség a tárigény csökkentésére

A NULL általában egy érték hiányát jelzi az oszlopokban. A NULL azt mutatja, hogy hiányzik vagy nem ismert egy adat. Ezzel szemben a GROUP BY kiterjesztés által visszaadott NULL, nem a hagyományos értelemben vett null-t jelenti. A NULL ebben az esetben azt jelzi, hogy itt egy részösszegről van szó. De ezekhez a részösszegekhez tartozó NULL elemeket is tárolni kell.
A NULL elemeket csak akkor tárolja az adatbázis, ha olyan oszlopok közé esik, amelyek értékeket tárolnak. Ebben az esetben 1 byte helyet foglal minden NULL elem. A sorok végén megjelenő NULL-ok nem foglalnak helyet, hiszen a következő sor fejléce követheti az utolsó nem NULL értéket, így az adatbázis tárhelyet takarít meg.
Ha a csoportosító oszlopokat a projekció végére tesszük, ezzel jelentősen csökkenthetjük a tárigényt, mert a sok NULL végű sor-t levágja a rendszer.
A **2. Függelék** szerinti materializált nézetekben, ha a sorok elején vannak a csoportosító attribútumok, akkor az alábbi tárhely igények merülnek fel (A mérést a **6. Függelék** segítségével végeztem):

```
TABLE_NAME                Foglalt Kb
----------------------------- ----------------------
COMMON_MV                 2514
A_MV                      2100
B_MV                      126
```

Ha a sorok végén vannak a csoportosító attribútumok:

```
TABLE_NAME                Foglalt Kb
----------------------------- ----------------------
COMMON_MV                 2374
A_MV                      2100
B_MV                      126
```

Ez természetesen nem minden esetben fogja csökkenteni a méretet, erre nem lehet pontos szabályt megfogalmazni (pl, ha a két grouping set meg lenne cserélve, akkor már kisebb lenne a helymegtakarítás), de általános esetben csökkentheti az igényeket.

A CUBE és ROLLUP transzformálása:

A GROUP BY CUBE (p.prod_category, p.prod_subcategory, c.cust_city) csoportosítás az alábbi 8 csoportot foglalja magába:

(), (p.prod_category), (p.prod_subcategory), (c.cust_city), (p.prod_category, p.prod_subcategory), (p.prod_category, c.cust_city), (p.prod_subcategory, c.cust_city), (p.prod_category, p.prod_subcategory, c.cust_city)

A GROUP BY RULLUP (p.prod_category, p.prod_subcategory, c.cust_city) csoportosítás az alábbi 4 csoportot foglalja magába:

(p.prod_category, p.prod_subcategory, c.cust_city), (p.prod_category, p.prod_subcategory), (p.prod_category), ()

Mint az a példában is látszik, ki lehet dolgozni olyan szabályokat, amelyek alapján a különböző típusú csoportosítások összevonhatóvá válnak.

Általános leírás a Cube és Rollup szétbontására:

A Rollup csoportosítás szétbontása:

```
GROUP BY ROLLUP (C1, C2, …, Cn-1, Cn)
GROUP BY GROUPING SETS ( (C1, C2, …, Cn-1, Cn)
  ,(C1, C2, ..., Cn-1)
  ...
  ,(C1, C2)
  ,(C1)
  ,() )
```

A Cube csoportosítás szétbontása:

```
GROUP BY CUBE (C1, C2, C3, ..., Cn-2, Cn-1, Cn)
GROUP BY GROUPING SETS (
   (C1, C2, C3, ..., Cn-2, Cn-1, Cn) -- Minden dimenziót tartalmazó csoport
  ,( C2, C3, ..., Cn-2, Cn-1, Cn) -- n-1 dimenziót tartalmazó csoport
  ,(C1, C3, ..., Cn-2, Cn-1, Cn)
  …
  ,(C1, C2, C3, ..., Cn-2, Cn-1,)
  ,(C3, ..., Cn-2, Cn-1, Cn) -- n-2 dimenziót tartalmazó csoport
  ,(C1  ..., Cn-2, Cn-1, Cn)
  …
  ,(C1, C2) -- 2 dimenziót tartalmazó csoport
  ,…
  ,(C1, Cn)
  ,…
  ,(Cn-1, Cn)
  ,…
  ,(C1) -- 1 dimenziót tartalmazó csoport
  ,(C2)
  ,…
  ,(Cn-1)
  ,(Cn)
  ,() ) – Teljes összeg, 0 dimenziót tartalmazó csoport
```

A Rollup csoportosítás szétbontása összetett oszloplisták esetén

```
ROLLUP(A, (C1, C2, ..., Cn) )
GROUPING SETS ( (A, C1, C2, ..., Cn), (A), () )
```

A Cube csoportosítás szétbontása összetett oszloplisták esetén

```
CUBE(A, (C1, C2, ..., Cn) )
GROUPING SETS ( (), (A), (C1, C2, ..., Cn), (A, C1, C2, ..., Cn) )
```

Ha egyéb módon vannak megadva a csoportosítások, akkor azokat az algoritmusok futtatása előtt át kell alakítani olyan alakra, amit az algoritmus értelmezni tud.
A második megoldást alkalmaztam a csoportosítási halmazok összevonására, azaz a közös materializált nézetbe felvettem az összes eredeti csoportosítást. Az azonos csoportosításokat megszüntettem, és a SELECT listába elkészítettem a GROUPING_ID oszlopot, amely a csoportosítások base grouping halmazát tartalmazta.
A gyors frissítést támogató materializált nézet SELECT részébe fel kellett venni a táblák elsődleges kulcsait. Ennek következtében minden csoportba fel kellett venni a kulcsokat (ha még nem szerepelnek a csoportosításban), így jelentősen nőtt a materializált nézet által foglalt tárterület, hiszen a kulcsok egyedisége miatt jelentősen megnőtt a sorok száma is.

5.2.4 A rendezés (ORDER BY klauzula) feldolgozása az összevonás során

Hiába alkalmazunk ORDER BY megkötést materializált nézetek definiálásakor, a rendszer ezeket csak a Materializált nézet felépítésékor veszi figyelembe, a frissítési metódusok (Fast/Full refresh) során már nem [4].
A nagy adathalmazokon futó lekérdezések teljesítményének növelése érdekében, a sorokat az ORDER BY megkötés szerinti sorrendben tárolhatjuk. Az inicializálás során, a rendezés következtében az adatokat fizikailag csoportosítjuk. Ha indexeket építünk azokra az oszlopokra, amik szerint a materializált nézetekben rendezünk, akkor a fizikai csoportosításnak köszönhetően a materializált nézet által elérni kívánt sorok átlagos I/O ideje (hozzáférési idő) csökken.
A materializált nézet inicializálását követően a rendszer nem veszi figyelembe a definícióban szereplő ORDER BY részeket, melynek eredményeképpen, nincs ráhatása arra, hogy mely típusú materializált nézetként észleli az Oracle adatbázis (pl: materializált nézet, amely nem tartalmaz aggregátumot). Ugyanezen okok miatt, a lekérdezésátírást sem befolyásolja az ORDER BY rész.
A fentiekből következik, hogy ha az összevonandó materializált nézetek valamelyike tartalmaz ORDER BY részt, akkor ezek a részek átemelhetők a közös materializált nézetbe, de ez nem fogja befolyásolni lekérdezésátírást. Tehát ha egy lekérdezés során rendezést is szeretnénk beállítani (és szöveges illesztés kapcsolatban van a lekérdezés és a materializált nézet), akkor a query rewrite a lekérdezés eredményét csak a materializált nézet rendezése után adja vissza.
A fenti problémák miatt az ORDER BY megkötést nem vettem figyelembe az összevonások vizsgálatakor.

5.3 Lehetőségek Materializált nézetek szétbontására:

Megvizsgáltam, milyen előnyökkel járna, ha szétbontanánk egy materializált nézetet. Egy materializált nézet szétbontása többletterhet jelent a nézetek frissítése során, valamint a köztük lévő redundancia miatt több tárhelyet is igényel. Emellett, a lekérdezés sebessége is csökkenni fog, hiszen a lekérdezésátírásnak újra össze kell kapcsolnia a szétbontott materializált nézeteket.
A szétbontott materializált nézetek újbóli összekapcsolása ráadásul plusz nehézségekbe ütközik, ugyanis a két résztábla közötti közös kapocs a ténytábla, ami általában nem rendelkezik elsődleges kulccsal.

Egyik lehetőség, ha a tábla elsődleges kulcsaként egy összetett kulcsot használ, amely a tábla összes idegen kulcsát tartalmazza, de az e szerinti kapcsoláshoz, ezt az összetett kulcsot mindkét résztáblában tárolni kéne, ez pedig jelentős többletterületet igényel.
A másik lehetőség az, hogy generálunk a ténytáblákhoz egy egy-oszlopos azonosítót.
A fentiekkel ellentétben mégis érdemes megvizsgálni a szétbontás lehetőségét, mert a szétbontás során az eredetitől általánosabb, több lekérdezést kiszolgáló materializált nézeteket kapunk. Az optimalizálás során a szétbontás utáni táblák átvehetik más nézetek helyét, kiváltva azok szükségességét, így javítva a rendszer hatékonyságát.

5.4 Materializált nézetek elhagyása

A materializált nézetek létrehozásánál mindig mérlegelni kell, hogy milyen befektetések milyen eredményekkel járnak. Előfordulhat olyan eset, hogy olyan nagy a materializált nézet fenntartási költsége, hogy nem éri meg megtartani. Valamint előfordulhat az is, hogy a materializált nézet megléte nem növeli a rendszer hatékonyságát, nem gyorsítja a lekérdezések lefutási idejét. Ez akkor fordulhat elő, ha a lekérdezésoptimalizáló úgy dönt, hogy hatékonyabb az alaptáblákra hivatkozó lekérdezést futtatni, mint átirányítani az összesítőtáblákra.
Ekkor a materializált nézetet elhagyhatjuk.

5.5 Az elkészült alkalmazás

A gyakorlatban a lekérdezések és a materializált nézetek nem sokban térnek el egymástól, a materializált nézetek tulajdonképpen lekérdezések, amelyeknek az eredményét tároljuk az adatbázisban. A megvalósítás során ezért nem is teszek különbséget lekérdezés és materializált nézet között, ugyanazon táblában tárolom őket, csak a tábla „description" oszlopának segítségével lehet megkülönböztetni őket.
A programkódot három PL/SQL csomagba szerveztem, itt a csak a csomagok legfontosabb függvényeit, procedúráit tüntetem fel:

- QUERY_ADAPTER csomag olyan procedúrákat és függvényeket tartalmaz, amelyek feladata a kapott string értelmezése, és adatbázis-táblákba töltése.
    - A QUERY_SEPARATOR procedúra két stringet kap bemenetként, az egyik tartalmazza a feldolgozandó lekérdezést, a másik pedig a lekérdezés azonosítójául szolgál. A procedúra meghívásával indul meg a lekérdezés feldolgozása, ő hívja meg a feldolgozáshoz szükséges további függvényeket és procedúrákat.
- AZ MV_GENERATOR csomag az egyes materializált nézetek létrehozásáért felelős elemeket tartalmazza:
    - QUERY_BUILDER: Függvény, amely visszatérési értéke a bemenetére megadott azonosítójú lekérdezés.
    - M_CONCENTRATION Procedúra két materializált nézetet metszetes módszerrel von össze. Bemeneti paraméterként meg kell adni az összevonandó materializált nézetek azonosítóját, egy azonosítót (ami az összevonás során előállt materializált nézetet fogja azonosítani), valamint egy karaktert, ami meghatározza az összevonás típusát. A procedúra 'f' paraméterrel való meghívása esetén egy 'FAST REFRESHABLE' materializált nézetet hoz létre.
    - U_CONCENTRATION Procedúra két materializált nézetet metszetes módszerrel von össze. Működése hasonló a M_CONCENTRATION procedúrával.

- MV_TREE csomag azokat a procedúrákat és függvényeket tartalmazza, amelyek segítségével felépíthető és bejárható a fa.
  - NODE_EXPOSE procedúra egy fa csomópont gyerekeit generálja le és tölti be az adatbázisba. Bemenetként megadható egy paraméter, amellyel engedélyezhető, illetve letiltható, hogy a generálás során elemek kihagyásával is létrejöjjenek új csomópontok.
  - COMPUTE_COSTS procedúra a fa csomópontjainak költségeit számítja ki. Ehhez létrehozza a csomópontban lévő materializált nézeteket. A materializált nézetek összméretét, a lekérdezések idejét és a csomópontban lévő materializált nézetek számát tárolja el.
  - COMPUTE_HX procedúra a COMPUTE_COSTS által kiszámolt költségek súlyozott összegét számítja ki, és menti el. Ezt a funkciót külön procedúrába szerveztem, hogy lehetőség legyen az összköltség megváltoztatására (a költségfüggvény súlyozásának átállítása esetén) anélkül, hogy a csoportok egyes költségelemeit újra kellene számolni.

Az implementáció során törekedtem a FAST REFRESHABLE materializált nézetek minél szélesebb körű támogatására. Az általános követelményeket teljesítik az általam generált materializált nézetek. Sok olyan feltétele van az opció használatának, amelyek a gyakorlati felhasználás során egyáltalán nem, vagy csak nagyon ritkán fordulnak elő. Ezek a feltételek a lekérdezések megadása előtt egyszerű mechanizmusok segítségével vizsgálhatók.

A csomópontok értékelése:
A csomópontok értékeléséhez létre kell hozni a csomópontban szereplő materializált nézeteket. Ez jelentős terhelést jelent az adatbázis számára, ezt ki kéne kerülni.
A materializált nézetek méretének meghatározására nem találtam hatékonyabb módszert, mint a nézetek fizikai létrehozása. A DBMS_MVIEW.ESTIMATE_MVIEW_SIZE procedúra segítségével meg lehet becsülni egy létrehozandó materializált nézet méretét, de ezzel a megoldással több probléma is van. Ez a procedúra nem tudja feldolgozni a SELECT klauzulában szereplő GROUPING_ID függvényt, de ennek kihagyása még nem rontaná el jelentősen az eredményt, hiszen itt csak egy becslésről van szó. A másik probléma, hogy a procedúra nem veszi figyelembe (nem is lenne képes) azt, hogy a materializált nézetben hány NULL elem szerepel. A megoldásom egyik alapköve, hogy a materializált nézetek összevonásakor a csoportosítási logikákat is összevonom (tehát nem „base grouping” csoportosítást használok), ezáltal kihasználom, hogy egy NULL elem sokkal kisebb helyet foglal, mint ha az adott oszlop ki lenne töltve. Ezeknek az a következménye, hogy a procedúra jelentősen túlbecsüli a létrehozandó materializált nézeteket, így lehetetlenné teszi a csomópontok közötti jó megoldás megtalálását. A materializált nézetek létrehozása után a nézetek méretét már könnyű meghatározni (a táblák által lefoglalt nem üres blokkjainak a számát meg kell szorozni a blokkmérettel).
A lekérdezési idők is egyszerűen mérhetőek, meg kell mérni, hogy mennyi idő alatt fut le az összes lekérdezés. A lekérdezések hatékonyságnövekedését egyéb módszerekkel is meg lehetne határozni, például a végrehajtási tervek elemzésével. A végrehajtási eredmények elemzése nehéz feladat, mivel az Oracle optimalizálója minden futtatáskor más és más végrehajtási tervet készít (ennek megfelelően a lekérdezési idők se állandóak, néhány futtatás átlagát érdemes figyelembe venni). Az egyszerűbb módszert választottam, a lekérdezési idők mérésével pontosabb, könnyebben feldolgozható eredményt kaptam.

A karbantartási költségek meghatározása a legnehezebb feladat, a költségek minimalizálása érdekében a FAST REFRESHABLE materializált nézetek létrehozását is támogatom, így inkrementális módon frissíthetőek a materializált nézetek (nem szükséges a materializált nézetek újbóli létrehozása). Ennek megfelelően, a karbantartási költségek tervezési időben nem határozhatóak meg (ha mégis tudhatóak, akkor megfelelő súlyozás segítségével a karbantartási költségek is jól becsülhetőek), ezek attól fognak függni, hogy milyen sűrűn frissítik az alaptáblákat, továbbá a frissítés során az alap adatok mekkora részét módosítják, mennyi adatot töltenek be a rendszerbe. A karbantartási költségekre ezért egy egyszerű módszert találtam ki, a materializált nézetek számával arányosnak tekintem a költségeket, ezért a karbantartási költséget a fa csomópontjában lévő materializált nézetek számával becsülöm.
Különbséget kell tenni a sima és a fast refreshable materializált nézetek között. A különbséget a súlyozás megfelelő beállításával lehet elérni, ha fast refreshable opcióval fogjuk a materializált nézeteket frissíteni, akkor nagyságrendileg kisebb súlyt érdemes rendelni a karbantartási költséghez, mintha teljes frissítést alkalmaznánk.

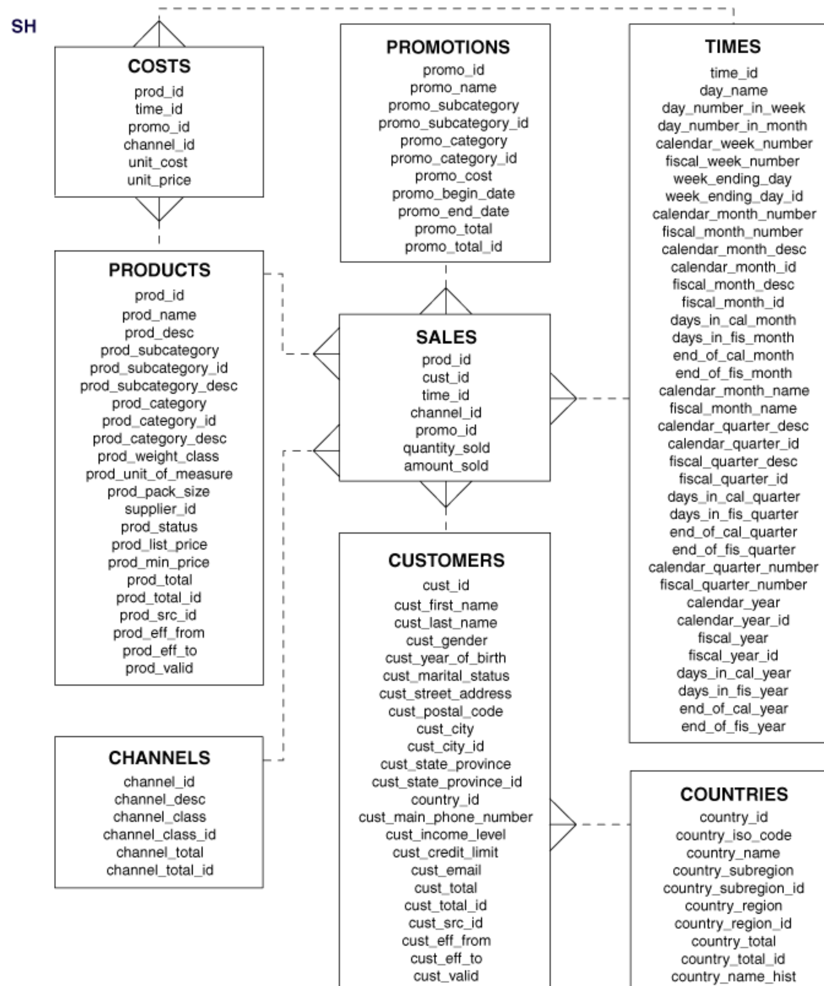
Az optimum megtalálása nem teljesen automatizálható feladat. A materializált nézetek kiválasztása során figyelembe kell venni az üzleti adatokat is, amelyek nem mindegyike nyerhetőek ki csak a rendszerből:

- Tárigény
- Karbantartás
    - ideje
    - **sűrűsége**
- **Milyen változásokkal kell számolni**
- Elérhető sebességnövekedés

Például a karbantartás sűrűségén spórolni lehet, ha megengedhető, hogy nem mindig a legpontosabb információk álljanak a rendelkezésünkre, azaz az adattárház feltöltése után (ETL) után nem egyből akarjuk frissíteni a materializált nézetet.
Ezen adatok figyelembevétellel kell meghatározni a materializált nézetek halmazait értékelő költségfüggvény súlyozását.

6. A megoldás értékelése

Az alkalmazásomat az értékesítési történet (Sales History, SH) sémán teszteltem le, ez az Oracle adattárházak és adatpiacok tesztelésére és oktatására készített rendszere. A tesztadatbázisnak köszönhetően egy fiktív cég adatpiacán dolgozhattam, amiben nagy adathalmaz állt rendelkezésre. Ezt a sémát módosítottam, a táblákon kívül minden objektumot töröltem. Erre azért volt szükség, hogy egy rosszul működő, nem hatékony rendszeren dolgozhassak, egy jól működő, hatékony rendszeren már nincs mit optimalizálni.



8. ábra. Az SH séma relációs diagramja

Az alkalmazásomat egy 25 lekérdezésből álló terhelésre teszteltem (7. Függelék). A költségfüggvényt minden esetben az alábbi súlyozással számoltam, a teljes méretet 0,1-es, a lekérdezési időket 1000-es, a karbantartási költséget 200-as súllyal vettem figyelembe. A futási eredmények a 9. ábrán láthatóak.
A 0. esetben a módosított adatbázison, materializált nézetek létrehozása nélkül mértem meg a lekérdezési időket.

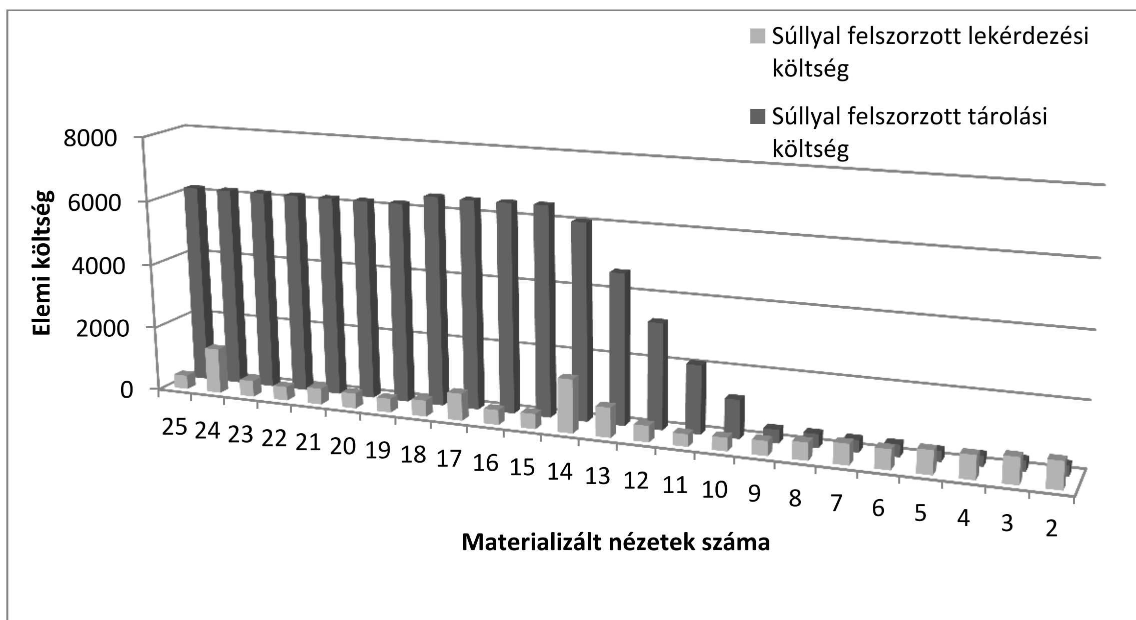
Ezután háromszor futtattam a tesztet, az alkalmazásom különböző paraméterezésével (FAST REFRESHABLE materializált nézetek támogatása, materializált nézetek kihagyása).
Az 1. esetben nem engedélyeztem materializált nézetek elhagyását, és nem támogattam a FAST REFRESHABLE frissítést sem. Ezekkel a beállításokkal érhetem el a legalacsonyabb lekérdezési időt, de a materializált nézetek mérete, és karbantartási költsége nagyobb lett, mint a többi esetben (8. Függelék).

A 2. futtatás alkalmával engedélyeztem a materializált nézetek elhagyását, így nőtt a lekérdezési idő, de jelentős karbantartási és tárolási költségcsökkenést értem el az 1. esethez képest (9. Függelék).

	Teljes méret [Kb]	Lekérdezési idő [s]	Karbantartási költség	Teljes súlyozott költség
0.	0	9,13	0	9130
1.	25888	0,53	3	3718,8
2.	2804	0,75	2	1430,4
3.	3240	0,84	2	1564
SAA.	18988	5,27	17	10568,8

9. ábra. A részletes futási eredmények

Az a tapasztalatom, hogy hatékony rendszer kiépítésénél engedélyezni kell a materializált nézetek elhagyását, valamint a FAST REFRESHABLE materializált nézetek ajánlását. A 3. futtatásom során is ezeket a paramétereket állítottam be, így a 2. futtatáshoz képest némi méretbeli és lekérdezési időbeni növekedés tapasztalható, viszont a karbantartási költségek a FAST REFRESH opció használatával jelentősen csökkennek (a futtatások könnyebb összehasonlítása érdekében a költségfüggvény súlyozását nem módosítottam, ennek megfelelően a 2. és 3. futtatás karbantartási költségei megegyeznek) (10. Függelék).
Az SAA. esetben az SQL Access Advisor javaslatait követve hoztam létre a materializált nézeteket. Az eszköz nem tudta feldolgozni a nagy terhelést jelentő lekérdezéseimet (amik összetett csoportosítási logikát alkalmaznak), ezért nem is tudta jelentősen csökkenteni a lekérdezési időket. Emellett nagyon sok nagy méretű materializált nézetet ajánlott, csak a legalapvetőbb összevonási módszereket alkalmazta, és nem is mérlegelte, hogy egy materializált nézet létrehozása milyen költségekkel jár. Előnye viszont, hogy az eszköz támogatja a FAST REFRESHABLE materializált nézetek létrehozását, valamint materializált nézet logokat is ajánl, amik szükségesek a materializált nézetek gyors frissítéséhez.

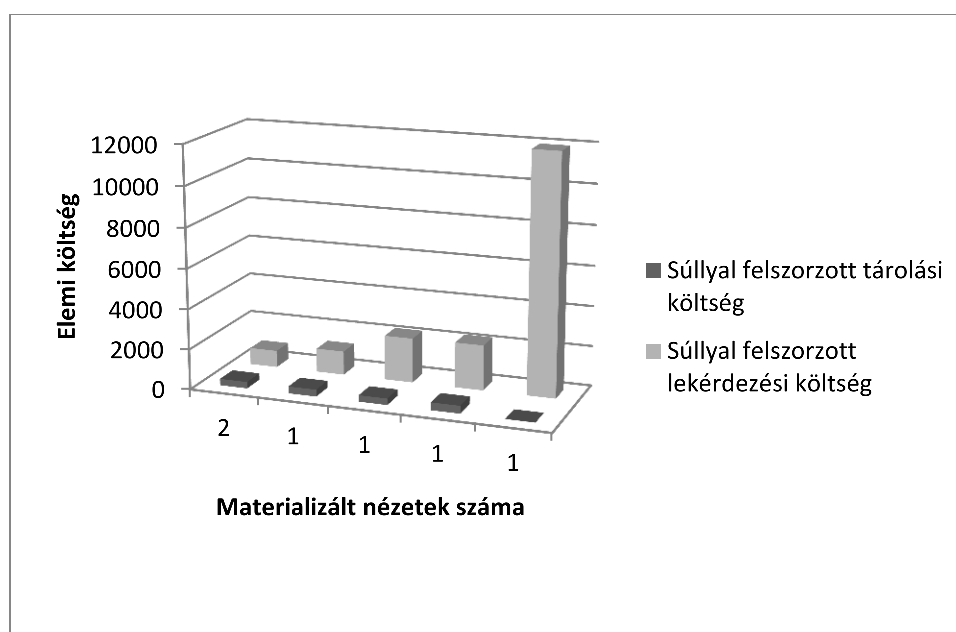


10. ábra. Az alkalmazásom futásának részeredményei

A 3. futtatás részletes eredményei a 10. ábrán láthatóak, azokat a csomópontokat jelenítettem meg az ábrán, amelyeket a fa bejárása során kiválasztott a mohó algoritmus. Az ábrán látható,

hogyan változtak a lekérdezési idők és a materializált nézetek összmérete az összevonások során. A 25 materializált nézetet tartalmazó csomópont a fa gyökér csomópontja. Ekkor a materializált nézetek összmérete 61842 Kb, a lekérdezési idő pedig 0,43 másodperc.
A 24-13 materializált nézetet tartalmazó csomópontok azok a csomópontok, amelyeket az irányított algoritmus választott ki, azaz amikor az azonos alaptáblákra vonatkozó materializált nézeteket vontuk össze. A bejárás során a materializált nézetek száma folyamatosan csökken (azaz csökken a karbantartási költség), emellett a materializált nézetek összmérete kezdetben kis mértékben nő, majd jelentősen lecsökken. Az összes azonos alaptáblára vonatkozó lekérdezés összevonása után a materializált nézetek összmérete 46458 Kb, a lekérdezési idő pedig 0,92 másodperc. A 12-2 materializált nézetet tartalmazó csomópontok azok a csomópontok, amelyeket a mohó algoritmus járt be, azaz a költségfüggvény segítségével választotta ki a megfelelő csomópontokat.

A mohó algoritmus a két materializált nézetet tartalmazó csomópontot adta eredményül, az egy materializált nézetet tartalmazó csomópontok a megadott költségfüggvény szerint kevésbé voltak hatékonyak, ezek az eredmények a 11. ábrán láthatóak.



11. ábra. A mohó algoritmus által megtalált csomópont

A grafikonokon is látszik, hogy a költségfüggvény súlyozása mennyire befolyásolhatja az eredményt. Például, ha egy olyan költségfüggvényt készítünk, amely a karbantartási költségeket kisebb súllyal veszi figyelembe, és a lekérdezési időket és a materializált nézetek méretét nagy súllyal számítja, akkor a 9 materializált nézetet tartalmazó csoporthoz hasonlót fog ajánlani (ez a csoport 4108 Kb tárhelyet foglal és 0,45 másodperc alatt szolgálják ki a lekérdezéseket).

7. Elért eredmények

A dolgozatomban adatpiacok optimalizálásával foglalkoztam, ahol a lekérdezések végrehajtási ideje kritikus lehet az adott vállalat számára. Az optimalizálás egyik eszköze az összesítőtáblák alkalmazása, ezek gyakran előforduló aggregált lekérdezések eredményeit tárolják. Az alaptáblákra hivatkozó lekérdezések lefutási idejét jelentősen csökkenteni lehet, ha ezeket összesítő táblák fölötti lekérdezésekké írjuk át. Ezt a módszert alkalmazva értem el jelentős eredményeket az adatpiacok optimalizálása terén.

Arra vállalkoztam, hogy ad-hoc lekérdezéseket kiszolgáló rendszerek optimalizálásával foglalkozzam. Az ad-hoc lekérdezések hatékony kiszolgálása különösen nehéz feladat, mivel a lekérdezések tervezés időben nem láthatók pontosan előre. Minél általánosabb materializált nézetek létrehozása volt a célom, hiszen minél általánosabb egy összesítőtábla annál szélesebb körben alkalmazható lekérdezésátírásra.

Munkám során olyan optimalizálási technikákat tanulmányoztam és dolgoztam ki, amelyekkel adott lekérdezéshalmazra alkalmazható összesítőtáblák száma és mérete minimalizálható a várható végrehajtási sebesség maximalizálása és az összesítőtáblák karbantartási időinek minimalizálása mellett.

A megoldást a materializált nézetek összevonásában találtam meg, ehhez egy teljes körű összevonási mechanizmust dolgoztam ki. A materializált nézetek összevonására többféle módszert fejlesztettem ki, melyeknek a rendszerre gyakorolt hatásait vizsgáltam. A materializált nézetek összevonásával a materializált nézetek száma csökken (így a karbantartási költségek is csökkennek), de az összevonás következtében a materializált nézetek összmérete, és az általuk kiszolgált lekérdezések lefutási ideje esetenként nőhet.

Annak eldöntésére, hogy mely materializált nézeteket érdemes összevonni, és melyeket nem, egy olyan fát építettem, amely segítségével a materializált nézetek összes lehetséges összevonásait megvizsgálhattam. Az összevonások értékelését egy matematikai költségfüggvénnyel végeztem, amely az adott rendszer igényeire szabva képes meghatározni az összevonás során előállt materializált nézetek költségeit.

Az így felépített fa bejárásával sikerült megtalálnom az optimális materializált nézeteket.

Munkám során egy olyan alkalmazást készítettem, amely képes egy megadott terhelés alapján előállítani egy jó materializált nézet halmazt. Az alkalmazásomat egy 25 lekérdezésből álló halmazra teszteltem, az eredményeimet összevetettem az Oracle eszközének javaslatával. Az SQL Access Advisorhoz képest sokkal jobb javaslatokat készítettem, az általam készített materializált nézetek száma, összmérete, és az elérhető sebességnövekedés terén is egy nagyságrenddel jobb eredményeket értem el.

8. Továbbfejlesztési lehetőségek

Mivel a feladat bonyolult optimalizálási probléma, amelyre nem létezik polinom idejű, algoritmus, a jelenlegi megoldás finomítására számos további lehetőség kínálkozik.
Az elkészített alkalmazás hatékonysága tovább fokozható, ha a csomópontok értékeléséhez, a csomópontok összehasonlításához nem hozzuk létre fizikailag az összesítőtáblákat. Ennek megoldása további vizsgálatokat igényel. Jó megközelítés lehet az összesítőtáblák méretének meghatározásához, ha az alaptáblák összekapcsolásainak és projektált oszlopainak elemzésével, az Oracle által képzett adatbázis statisztikák alapján becsüljük a szükséges tárhelyet.
Az elérhető sebességnövekedés mértékét és a karbantartási költségek becslését a lekérdezésoptimalizáló által generált végrehajtási tervek alapján lehet becsülni.
Bár a jelenlegi megoldás még nem képes az összes lehetséges lekérdezés feldolgozására, de a gyakorlatban előforduló tipikus lekérdezések kezelésére igen. Ezen a téren az esetleges továbbfejlesztési igényeket a gyakorlatban fölmerülő lekérdezések pontosabb vizsgálatával kell majd meghatározni.
A fast refreshable materializált nézetek generálásához az általános követelmények teljesítésén túl teljeskörű ellenőrzési mechanizmus implementálása is lehetne a cél, ezzel azonban csak a fejlesztői beavatkozások száma lenne tovább csökkenthető, amely a feladat jellegéből következően nem alapvetően fontos, hiszen a rendszert jólképzett informatikusok fogják használni.

Felhasznált irodalom

[1] Kimball, Ralph, Ross, Margy: „The Data Warehouse Toolkit – The complete guide to dimensional modeling", Wiley & Sons, 2002.

[2] Inmon, William Harvey: „Building the Data Warehouse", Wiley & Sons, New York, 2002.

[3] Immanuel Chan, Lance Ashdown, at al „Oracle® Database Performance Tuning Guide 11g Release 2 (11.2)", 2011
http://download.oracle.com/docs/cd/E11882_01/server.112/e16638.pdf

[4] Paul Lane, et al „Oracle Data Warehousing Guide 11g Release 2 (11.2)", 2010,
http://download.oracle.com/docs/cd/E11882_01/server.112/e25554.pdf

[5] V. Harinarayan, A. Rajaraman, és J. Ullman „Implementing data cubes efficiently", Proceedings of ACM SIGMOD 1996 International Conference on Management of Data, Montreal, Canada, 1996.

[6] J.Yang, K. Karlapalem, és Q. Li. „A framework for designing materialized views in data warehousing environment", Technical Report HKUST-cs96-35, 1996. IEEE Int' l conference on Distributed Computing Systems (ICDCS ' 97), Maryland, U.S.A., 1997.

[7] H. Gupta, „Selection of Views to Materialize in a Data Warehouse". Proceedings of 23rd VLDB Conference, Athens, Greece pp 42-47, 1997.

[8]G. K. Y. Chan, Q. Li, és L. Feng. „Design and selection of materialized views in a data warehousing environment: a case study", In 2nd ACM international workshop on Data warehousing and OLAP (DOLAP 1999), Kansas City, USA, pages 42-47,1999.

[9] K. Aouiche, P.-E. Jouve, és J. Darmont. „Clustering-based materialized view selection in data warehouses", Advances in Databasesmand Information Systems 2006 (ADBIS'06), vol. 4152, LNCS, pp. 81–95, 2006.

[10] A. Shukla, P. Deshpande, és J. F. Naughton. „Materialized view selection for multi-cube data models", 7th International Conference on Extending DataBase Technology (EDBT 2000), Konstanz, Germany, pp. 269-284, 2000.

[11] J. R. Smith, C.-S. Li, és A. Jhingran. „A wavelet framework for adapting data cube views for OLAP", IEEE Transactions on Knowledge and Data Engineering, pp. 552-565, 2004.

[12] Y. Sismanis, A. Deligiannakis, N. Roussopoulos, és Y. Kotidis. „Dwarf: shrinking the petacube", In ACM SIGMOD International Conference on Management of Data (SIGMOD 2002), Madison, USA, pages 464-475, 2002.

Függelék

A függelék példákat tartalmaz materializált nézetek különböző típusú összevonására. Továbbá tartalmazza az alkalmazásom teszteléséhez használt lekérdezéseket, valamint az alkalmazásom és az SQL Access Advisor futásának eredményeit.

1. Függelék - Csak Joinokat tartalmazó materializált nézetek összevonása, azonos alaptáblák esetén

A materializált nézet :

```
CREATE MATERIALIZED VIEW A_MV
ENABLE QUERY REWRITE AS
SELECT p.prod_id, p.prod_name, t.time_id, t.week_ending_day,
     s.channel_id, s.promo_id, s.cust_id, s.amount_sold
FROM   sales s, products p, times t
WHERE  s.time_id=t.time_id
AND s.prod_id = p.prod_id
```

B materializált nézet :

```
CREATE MATERIALIZED VIEW B_MV
ENABLE QUERY REWRITE AS
SELECT p.prod_subcategory, t.calendar_month_desc,
   s.amount_sold,
FROM sales s, products p, times t
WHERE s.time_id = t.time_id
AND s.prod_id = p.prod_id
```

Az összevont materializált nézet :

```
CREATE MATERIALIZED VIEW COMMON_MV
ENABLE QUERY REWRITE AS
SELECT p.prod_id, p.prod_name, p.prod_subcategory, t.time_id, t.week_ending_day,
t.calendar_month_desc, s.channel_id, s.promo_id, s.cust_id, s.amount_sold
FROM sales s, products p, times t
WHERE s.time_id = t.time_id
AND s.prod_id = p.prod_id
```

2. Függelék - Aggregátumokat is tartalmazó materializált nézetek összevonása, azonos alaptáblák esetén

A materializált nézet :

```
CREATE MATERIALIZED VIEW A_MV
ENABLE QUERY REWRITE AS
SELECT p.prod_subcategory, t.calendar_month_desc, c.cust_city,
   COUNT(s.amount_sold) AS count_amount_sold,
   SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, times t, customers c
WHERE s.time_id=t.time_id
   AND s.prod_id=p.prod_id
   AND s.cust_id=c.cust_id
GROUP BY p.prod_subcategory, t.calendar_month_desc, c.cust_city;
```

B materializált nézet :

```
CREATE MATERIALIZED VIEW B_MV
ENABLE QUERY REWRITE AS
SELECT p.prod_category, t.calendar_quarter_desc, c.cust_state_province,
   COUNT(s.amount_sold) AS count_amount_sold,
   SUM(s.amount_sold) AS sum_amount_sold
```

```
FROM sales s, products p, times t, customers c
WHERE s.time_id=t.time_id
   AND s.prod_id=p.prod_id
   AND s.cust_id=c.cust_id
GROUP BY p.prod_category, t.calendar_quarter_desc, c.cust_state_province;
```

Az összevont materializált nézet :

```
CREATE MATERIALIZED VIEW COMMON_MV
ENABLE QUERY REWRITE AS
SELECT p.prod_subcategory, t.calendar_month_desc, c.cust_city, p.prod_category,
t.calendar_quarter_desc, c.cust_state_province,
   COUNT(s.amount_sold) AS count_amount_sold,
   SUM(s.amount_sold) AS sum_amount_sold,
   GROUPING_ID(p.prod_subcategory, t.calendar_month_desc, c.cust_city, p.prod_category,
t.calendar_quarter_desc, c.cust_state_province)
FROM sales s, products p, times t, customers c
WHERE s.time_id=t.time_id
   AND s.prod_id=p.prod_id
   AND s.cust_id=c.cust_id
GROUP BY GROUPING SETS((p.prod_subcategory, t.calendar_month_desc, c.cust_city),(
 p.prod_category, t.calendar_quarter_desc, c.cust_state_province));
```

3. Függelék - Csak Joinokat tartalmazó materializált nézetek összevonása UNIÓS módszerrel

A materializált nézet :

```
CREATE MATERIALIZED VIEW A_MV
ENABLE QUERY REWRITE AS
SELECT p.prod_id, p.prod_name, t.time_id, t.week_ending_day,
     s.channel_id, s.promo_id, s.cust_id, s.amount_sold
FROM   sales s, products p, times t
WHERE  s.time_id=t.time_id
AND s.prod_id = p.prod_id;
```

B materializált nézet :

```
CREATE MATERIALIZED VIEW B_MV
ENABLE QUERY REWRITE AS
SELECT p.prod_subcategory, t.calendar_month_desc,
   s.amount_sold, c.cust_city
FROM sales s, products p, times t, customers c
WHERE s.time_id = t.time_id
AND s.prod_id = p.prod_id
AND s.cust_id = c.cust_id;
```

Az összevont materializált nézet :

```
CREATE MATERIALIZED VIEW COMMON_MV
ENABLE QUERY REWRITE AS
SELECT p.prod_id, p.prod_name, p.prod_subcategory, t.time_id, t.week_ending_day,
t.calendar_month_desc, c.cust_city, s.channel_id, s.promo_id, s.cust_id, s.amount_sold
FROM sales s, products p, times t, customers c
WHERE s.time_id = t.time_id
AND s.prod_id = p.prod_id
AND s.cust_id = c.cust_id;
```

4. Függelék - Aggregátumokat is tartalmazó materializált nézetek összevonása UNIÓS módszerrel

A materializált nézet :

```
CREATE MATERIALIZED VIEW A_mv
ENABLE QUERY REWRITE AS
SELECT p.prod_subcategory, t.calendar_month_desc, c.cust_city,
   COUNT(s.amount_sold),
   SUM(s.amount_sold)
FROM sales s, products p, times t, customers c
WHERE s.time_id=t.time_id
   AND s.prod_id=p.prod_id
   AND s.cust_id=c.cust_id
GROUP BY p.prod_subcategory, t.calendar_month_desc, c.cust_city;
```

B materializált nézet :

```
CREATE MATERIALIZED VIEW B_mv
ENABLE QUERY REWRITE AS
SELECT p.prod_subcategory, t.week_ending_day,
      SUM(s.amount_sold)
FROM   sales s, products p, times t
WHERE  s.time_id=t.time_id  AND s.prod_id=p.prod_id
GROUP BY p.prod_subcategory, t.week_ending_day;
```

Az összevont materializált nézet:

```
CREATE MATERIALIZED VIEW COMMON_MV
ENABLE QUERY REWRITE AS
SELECT GROUPING_ID(p.prod_subcategory, t.calendar_month_desc, c.cust_city,
t.week_ending_day),
   t.week_ending_day, p.prod_subcategory, t.calendar_month_desc, c.cust_city,
   COUNT(s.amount_sold), SUM(s.amount_sold)
FROM sales s, products p, times t, customers c
WHERE s.time_id=t.time_id
   AND s.prod_id=p.prod_id
   AND s.cust_id=c.cust_id
GROUP BY GROUPING SETS((p.prod_subcategory, t.calendar_month_desc, c.cust_city),(
 p.prod_subcategory, t.week_ending_day));
```

5. Függelék - Aggregátumokat is tartalmazó materializált nézetek összevonása METSZETES módszerrel

A materializált nézet :

```
CREATE MATERIALIZED VIEW A_MV
ENABLE QUERY REWRITE AS
SELECT p.prod_subcategory, t.calendar_month_desc,
   COUNT(s.amount_sold) AS count_amount_sold,
   SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, times t
WHERE s.time_id=t.time_id
   AND s.prod_id  =p.prod_id
GROUP BY p.prod_subcategory, t.calendar_month_desc;
```

B materializált nézet :

```
CREATE MATERIALIZED VIEW B_MV
ENABLE QUERY REWRITE AS
SELECT p.prod_subcategory, t.calendar_month_desc, c.cust_city,
   COUNT(s.amount_sold) AS count_amount_sold,
   SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, times t, customers c
WHERE s.time_id=t.time_id
```

```
   AND s.prod_id=p.prod_id
   AND s.cust_id=c.cust_id
GROUP BY p.prod_subcategory, t.calendar_month_desc, c.cust_city;
```

Az összevont materializált nézet:

```
CREATE MATERIALIZED VIEW COMMON_MV
ENABLE QUERY REWRITE AS
SELECT p.prod_subcategory, t.calendar_month_desc,
   COUNT(s.amount_sold) AS count_amount_sold,
   SUM(s.amount_sold) AS sum_amount_sold,
   s.cust_id
FROM sales s, products p, times t
WHERE s.time_id=t.time_id
   AND s.prod_id  =p.prod_id
GROUP BY p.prod_subcategory, t.calendar_month_desc, s.cust_id;
```

6. Függelék - A materializált nézetek méretének meghatározása

```
select count(*) from common_mv;
select count(*) from a_mv;
select count(*) from b_mv;

ANALYZE TABLE SH.A_MV ESTIMATE STATISTICS;
ANALYZE TABLE SH.B_MV ESTIMATE STATISTICS;
ANALYZE TABLE SH.COMMON_MV ESTIMATE STATISTICS;

SELECT t.TABLE_NAME, (s.BLOCKS-t.EMPTY_BLOCKS)*2048/1024 "Foglalt Kb"
FROM   DBA_SEGMENTS s, DBA_TABLES t
WHERE  ((s.OWNER=UPPER('SH') AND s.SEGMENT_NAME = UPPER('common_mv'))
AND (t.OWNER=UPPER('SH') AND t.TABLE_NAME = UPPER('common_mv')))
OR ((s.OWNER=UPPER('SH') AND s.SEGMENT_NAME = UPPER('a_mv'))
AND (t.OWNER=UPPER('SH') AND t.TABLE_NAME = UPPER('a_mv')))
OR ((s.OWNER=UPPER('SH') AND s.SEGMENT_NAME = UPPER('b_mv'))
AND (t.OWNER=UPPER('SH') AND t.TABLE_NAME = UPPER('b_mv')))
```

7. Függelék - Ezekre a lekérdezésekre futtattam az alkalmazásomat, ez a terhelés

```
SELECT s.prod_id,
  SUM(amount_sold)   AS dollar_sales,
  SUM(quantity_sold) AS unit_sales
FROM sales s
GROUP BY s.prod_id;

SELECT s.prod_id,
  s.time_id,
  SUM(s.amount_sold)     AS sum_dollar_sales,
  COUNT(s.amount_sold)   AS count_dollar_sales,
  SUM(s.quantity_sold)   AS sum_quantity_sales,
  COUNT(s.quantity_sold) AS count_quantity_sales
FROM sales s
GROUP BY s.prod_id,
  s.time_id;

SELECT t.calendar_month_desc,
  SUM(s.amount_sold) AS dollars
FROM sales s,
  times t
```

```
WHERE s.time_id = t.time_id
GROUP BY t.calendar_month_desc;

SELECT t.week_ending_day,
  SUM(s.amount_sold)
FROM sales s,
  times t
WHERE s.time_id = t.time_id
AND t.week_ending_day BETWEEN TO_DATE ('01-08-1999', 'DD-MM-YYYY') AND
TO_DATE('10-09-1999', 'DD-MM-YYYY')
GROUP BY week_ending_day;

SELECT p.prod_name,
  SUM(s.amount_sold) AS dollar_sales
FROM products p,
  sales s
WHERE p.prod_id = s.prod_id
GROUP BY prod_name
HAVING SUM(s.amount_sold) BETWEEN 5000 AND 50000;

SELECT p.promo_name,
  SUM(s.amount_sold) AS sum_amount_sold
FROM promotions p,
  sales s
WHERE s.promo_id  = p.promo_id
AND p.promo_name IN ('coupon', 'premium', 'giveaway')
GROUP BY promo_name;

SELECT c.cust_last_name,
  SUM(amount_sold) AS sum_amount_sold
FROM customers c,
  sales s
WHERE s.cust_id = c.cust_id
GROUP BY c.cust_last_name;

SELECT s.time_id,
  p.prod_id,
  p.prod_name,
  SUM(s.quantity_sold),
  SUM(s.amount_sold),
  COUNT(s.quantity_sold),
  COUNT(s.amount_sold)
FROM sales s,
  products p,
  times t
WHERE s.time_id = t.time_id
AND s.prod_id   = p.prod_id
GROUP BY s.time_id,
  p.prod_id,
  p.prod_name;

SELECT t.calendar_month_name,
  p.prod_id,
  p.prod_name,
  SUM(s.quantity_sold),
```

```
  SUM(s.amount_sold),
  COUNT(s.quantity_sold),
  COUNT(s.amount_sold)
FROM sales s,
  products p,
  times t
WHERE s.time_id = t.time_id
AND s.prod_id   = p.prod_id
GROUP BY t.calendar_month_name,
  p.prod_id,
  p.prod_name;

SELECT p.prod_subcategory_desc,
  t.week_ending_day,
  SUM(s.amount_sold)
FROM sales s,
  products p,
  times t
WHERE s.time_id=t.time_id
AND s.prod_id  =p.prod_id
AND p.prod_subcategory_desc LIKE '%Games%'
GROUP BY p.prod_subcategory_desc,
  t.week_ending_day;

SELECT p.prod_category,
  t.week_ending_day,
  SUM(s.amount_sold) AS sum_amount
FROM sales s,
  products p,
  times t
WHERE s.time_id=t.time_id
AND s.prod_id  =p.prod_id
GROUP BY p.prod_category,
  t.week_ending_day;

SELECT p.prod_name,
  t.week_ending_day,
  SUM(s.amount_sold)
FROM sales s,
  products p,
  times t
WHERE s.time_id=t.time_id
AND s.prod_id  =p.prod_id
GROUP BY p.prod_name,
  t.week_ending_day;

SELECT p.prod_id,
  t.week_ending_day,
  s.cust_id,
  SUM(s.amount_sold) AS sum_amount_sold
FROM sales s,
  products p,
  times t
WHERE s.time_id=t.time_id
AND s.prod_id  =p.prod_id
```

```
GROUP BY p.prod_id,
  t.week_ending_day,
  s.cust_id;

SELECT p.prod_name,
  t.week_ending_day,
  SUM(s.amount_sold)
FROM sales s,
  products p,
  times t
WHERE s.time_id=t.time_id
AND s.prod_id  = p.prod_id
AND t.week_ending_day BETWEEN TO_DATE('01-01-1999', 'DD-MM-YYYY') AND
TO_DATE('01-02-1999', 'DD-MM-YYYY')
GROUP BY p.prod_name,
  t.week_ending_day;

SELECT p.prod_subcategory,
  t.calendar_month_desc,
  c.cust_city,
  AVG(s.amount_sold)
FROM sales s,
  products p,
  times t,
  customers c
WHERE s.time_id=t.time_id
AND s.prod_id  =p.prod_id
AND s.cust_id  =c.cust_id
GROUP BY p.prod_subcategory,
  t.calendar_month_desc,
  c.cust_city;

SELECT country_name country,
  prod_name prod,
  calendar_year YEAR,
  SUM(amount_sold) sale,
  COUNT(amount_sold) cnt
FROM sales,
  times,
  customers,
  countries,
  products
WHERE sales.time_id      = times.time_id
AND sales.prod_id        = products.prod_id
AND sales.cust_id        = customers.cust_id
AND customers.country_id = countries.country_id
GROUP BY country_name,
  prod_name,
  calendar_year;

SELECT p.prod_subcategory,
  t.calendar_month_desc,
  c.cust_city,
  SUM(s.amount_sold) AS sum_amount_sold
FROM sales s,
```

```
  customers c,
  products p,
  times t
WHERE s.time_id=t.time_id
AND s.prod_id  = p.prod_id
AND s.cust_id  = c.cust_id
GROUP BY GROUPING SETS ((p.prod_subcategory, t.calendar_month_desc), (c.cust_city,
p.prod_subcategory));

SELECT p.prod_category,
  p.prod_subcategory,
  c.cust_state_province,
  c.cust_city,
  GROUPING_ID(p.prod_category,p.prod_subcategory, c.cust_state_province,c.cust_city) AS
gid,
  SUM(s.amount_sold)                                      AS sum_amount_sold
FROM sales s,
  products p,
  customers c
WHERE s.prod_id = p.prod_id
AND s.cust_id   = c.cust_id
GROUP BY GROUPING SETS ((p.prod_category, p.prod_subcategory, c.cust_city),
(p.prod_category, p.prod_subcategory, c.cust_state_province, c.cust_city), (p.prod_category,
p.prod_subcategory));

SELECT p.prod_category,
  p.prod_subcategory,
  c.cust_state_province,
  t.calendar_month_desc,
  SUM(s.amount_sold) AS sum_amount_sold
FROM sales s,
  products p,
  customers c,
  times t
WHERE s.prod_id = p.prod_id
AND s.cust_id   = c.cust_id
AND s.time_id   = t.time_id
GROUP BY GROUPING SETS ((p.prod_subcategory, t.calendar_month_desc),
(t.calendar_month_desc), (p.prod_category, p.prod_subcategory, c.cust_state_province),
(p.prod_category, p.prod_subcategory));

SELECT c.cust_id,
  c.cust_last_name,
  s.amount_sold,
  s.quantity_sold,
  s.time_id
FROM sales s,
  times t,
  customers c
WHERE s.cust_id = c.cust_id(+)
AND s.time_id   = t.time_id(+);

SELECT c.cust_id,
  c.cust_last_name,
  s.amount_sold,
```

```
  t.time_id,
  t.day_number_in_week
FROM sales s,
  customers c,
  times t
WHERE s.time_id = t.time_id
AND s.cust_id   = c.cust_id;

SELECT p.prod_id,
  p.prod_name,
  t.time_id,
  t.week_ending_day,
  s.channel_id,
  s.promo_id,
  s.cust_id,
  s.amount_sold
FROM sales s,
  products p,
  times t
WHERE s.time_id=t.time_id
AND s.prod_id  = p.prod_id;

SELECT p.prod_id,
  p.prod_name,
  t.time_id,
  t.week_ending_day,
  s.channel_id,
  s.promo_id,
  s.cust_id,
  s.amount_sold
FROM sales s,
  products p,
  times t
WHERE s.time_id=t.time_id
AND s.prod_id  =p.prod_id(+);

SELECT t.calendar_year,
  t.calendar_month_number,
  t.day_number_in_month,
  c1.country_name,
  s.prod_id,
  s.quantity_sold,
  s.amount_sold
FROM times t,
  countries c1,
  sales s,
  customers c2
WHERE s.time_id      = t.time_id
AND s.cust_id        = c2.cust_id
AND c2.country_id    = c1.country_id
AND c1.country_name IN ('United States of America', 'Argentina', 'Japan', 'India', 'France',
'Spain', 'Ireland');

SELECT cust_last_name,
  cust_first_name,
```

```
  cust_credit_limit,
  cust_year_of_birth
FROM customers
WHERE cust_credit_limit BETWEEN 1000 AND 5000
AND cust_year_of_birth  > 1950
AND cust_year_of_birth <= 1955;
```

8. Függelék - Az alkalmazásom első futtatása során előállított materializált nézetek

```
CREATE MATERIALIZED VIEW "SH"."MV1" REFRESH FORCE
WITH ROWID ENABLE QUERY REWRITE AS
SELECT sales.cust_id,
  sales.time_id,
  times.day_number_in_week,
  sales.channel_id,
  times.day_number_in_month,
  sales.amount_sold,
  times.calendar_year,
  times.calendar_month_number,
  times.week_ending_day,
  sales.promo_id,
  sales.quantity_sold,
  sales.prod_id
FROM sales,
  times
WHERE sales.time_id = times.time_id (+)

CREATE MATERIALIZED VIEW "SH"."MV2" REFRESH FORCE
WITH ROWID ENABLE QUERY REWRITE AS
SELECT COUNT(sales.amount_sold),
  customers.cust_city,
  SUM(sales.amount_sold),
  products.prod_category,
  products.prod_subcategory,
  customers.cust_state_province,
  times.calendar_month_desc,
  customers.cust_last_name,
  GROUPING_ID(customers.cust_city , products.prod_category , products.prod_subcategory ,
customers.cust_last_name , customers.cust_state_province , times.calendar_month_desc)
FROM customers,
  products,
  sales,
  times
WHERE sales.cust_id = customers.cust_id
AND sales.prod_id   = products.prod_id
AND sales.time_id   = times.time_id
GROUP BY GROUPING SETS((customers.cust_last_name) , (customers.cust_city ,
products.prod_category , products.prod_subcategory) , (customers.cust_city ,
customers.cust_state_province , products.prod_category , products.prod_subcategory) ,
(times.calendar_month_desc) , (customers.cust_state_province , products.prod_category ,
products.prod_subcategory) , (products.prod_category , products.prod_subcategory) ,
(customers.cust_city , products.prod_subcategory , times.calendar_month_desc) ,
(products.prod_subcategory , times.calendar_month_desc) , (customers.cust_city ,
products.prod_subcategory))

CREATE MATERIALIZED VIEW "SH"."MV3" REFRESH FORCE
```

```
WITH ROWID ENABLE QUERY REWRITE AS
SELECT COUNT(sales.amount_sold),
  sales.time_id,
  promotions.promo_name,
  SUM(sales.amount_sold),
  SUM(sales.quantity_sold),
  products.prod_name prod,
  sales.cust_id,
  countries.country_name country,
  times.calendar_year YEAR,
  sales.prod_id,
  COUNT(sales.quantity_sold),
  GROUPING_ID(promotions.promo_name , sales.time_id , times.calendar_year ,
countries.country_name , sales.cust_id , products.prod_name , sales.prod_id)
FROM customers,
  countries,
  products,
  sales,
  times,
  promotions
WHERE sales.time_id = times.time_id
AND sales.prod_id = products.prod_id
AND sales.cust_id = customers.cust_id
AND customers.country_id = countries.country_id
AND sales.promo_id  = promotions.promo_id
GROUP BY GROUPING SETS((countries.country_name , products.prod_name ,
times.calendar_year) , (promotions.promo_name) , (sales.prod_id , sales.time_id) ,
(sales.cust_id , sales.prod_id , sales.time_id))
```

9. Függelék - Az alkalmazásom második futtatása során előállított materializált nézetek

```
CREATE MATERIALIZED VIEW "SH"."MV1" REFRESH FORCE
WITH ROWID ENABLE QUERY REWRITE AS
SELECT countries.country_name,
  products.prod_name,
  times.calendar_year,
  SUM(sales.amount_sold),
  COUNT(sales.amount_sold)
FROM sales,
  times,
  customers,
  countries,
  products
WHERE sales.time_id  = times.time_id
AND sales.prod_id   = products.prod_id
AND sales.cust_id       = customers.cust_id
AND customers.country_id = countries.country_id
GROUP BY countries.country_name ,
  products.prod_name ,
  times.calendar_year

CREATE MATERIALIZED VIEW "SH"."MV2" REFRESH FORCE
WITH ROWID ENABLE QUERY REWRITE AS
SELECT COUNT(sales.amount_sold),
  SUM(sales.amount_sold),
  customers.cust_city,
```

```
  products.prod_category,
  products.prod_subcategory,
  times.week_ending_day,
  products.prod_name,
  customers.cust_last_name,
  customers.cust_state_province,
  times.calendar_month_desc,
  GROUPING_ID(customers.cust_city , times.week_ending_day , products.prod_category ,
products.prod_subcategory , products.prod_name , customers.cust_last_name ,
times.calendar_month_desc , customers.cust_state_province)
FROM customers,
  products,
  sales,
  times
WHERE sales.time_id  = times.time_id
AND products.prod_id = sales.prod_id
AND sales.cust_id    = customers.cust_id
GROUP BY GROUPING SETS((times.week_ending_day) , (products.prod_name) ,
(customers.cust_last_name) , (times.calendar_month_desc) , (customers.cust_state_province ,
products.prod_category , products.prod_subcategory) , (products.prod_category ,
products.prod_subcategory) , (customers.cust_city , products.prod_subcategory ,
times.calendar_month_desc) , (products.prod_subcategory , times.calendar_month_desc) ,
(customers.cust_city , products.prod_subcategory))
```

10. Függelék - Az alkalmazásom harmadik futtatása során előállított materializált nézetek

```
CREATE MATERIALIZED VIEW "SH"."MV1" REFRESH FORCE
WITH ROWID ENABLE QUERY REWRITE AS
SELECT COUNT(*),
  COUNT(sales.amount_sold),
  customers.cust_city,
  customers.cust_state_province,
  GROUPING_ID(customers.cust_city , times.week_ending_day , products.prod_category ,
products.prod_subcategory , products.prod_name , customers.cust_state_province ,
times.calendar_month_desc),
  products.prod_category,
  products.prod_name,
  products.prod_subcategory,
  SUM(sales.amount_sold),
  times.calendar_month_desc,
  times.week_ending_day
FROM customers,
  products,
  sales,
  times
WHERE sales.time_id = times.time_id
AND sales.prod_id   = products.prod_id
AND sales.cust_id   = customers.cust_id
GROUP BY GROUPING SETS((times.week_ending_day) , (customers.cust_city ,
products.prod_category , products.prod_subcategory) , (customers.cust_city ,
customers.cust_state_province , products.prod_category , products.prod_subcategory) ,
(products.prod_name) , (times.calendar_month_desc) , (customers.cust_state_province ,
products.prod_category , products.prod_subcategory) , (products.prod_category ,
products.prod_subcategory) , (customers.cust_city , products.prod_subcategory ,
times.calendar_month_desc) , (products.prod_subcategory , times.calendar_month_desc) ,
(customers.cust_city , products.prod_subcategory))
```

```
CREATE MATERIALIZED VIEW "SH"."MV2" REFRESH FORCE
WITH ROWID ENABLE QUERY REWRITE AS
SELECT COUNT(*),
  countries.country_name,
  COUNT(sales.amount_sold),
  products.prod_name prod,
  SUM(sales.amount_sold),
  times.calendar_year
FROM customers,
  countries,
  products,
  sales,
  times
WHERE sales.time_id      = times.time_id
AND sales.prod_id        = products.prod_id
AND sales.cust_id        = customers.cust_id
AND customers.country_id = countries.country_id
GROUP BY countries.country_name ,
  products.prod_name ,
  times.calendar_year
```