DIPLOMAMUNKA

Tajti Ákos

Debrecen

2009.

Debreceni Egyetem
Informatikai Kar

Adattisztítás adatbányászati módszerekkel multidimenzionális adatbázisokban

Témavezető:
Dr. Juhász István
Egyetemi adjunktus

Készítette:
Tajti Ákos
PTM

Debrecen

2009.

„Úgy tűnik, a tökéletességet nem akkor közelítjük meg a legjobban, amikor már nem tudunk mit hozzáadni, hanem amikor már nem tudunk mit elvenni.”

(Antoine de Saint Exupéry)

Tartalomjegyzék

1 Bevezetés

Manapság az adatbázisok használata szinte minden területen elkerülhetetlen. Adatbázisokat használnak a különböző tudományágak kísérleti eredményeik tárolására, a hivatalok nyilvántartásaik vezetésére, az állami intézmények statisztikáik összegyűjtésére és a sort tovább folytathatnánk. Akármilyen alkalmazási területtel van is dolgunk a felhasználás módjában, és így az adatokkal kapcsolatban felmerülő problémákban rengeteg a közös.

Ahol az adatokat nem automatikusan tárolják el, szinte biztos, hogy az adatbázisban idővel hibák, inkonzisztenciák jelennek meg. A TDWI (The Data Warehouse Institute) 2002-es jelentése szerint az adatok rossz minősége évente 611 milliárd dollár plusz (főként postázási és javítási) költséget jelent az USA-beli vállalatoknak (Eckerson, 2002.). Válságos időben különösen fontos, hogy a cégek csökkenteni tudják kiadásaikat, így az adattisztítás egyre vonzóbb lehetőséggé válhat.

Ha az adatok „tisztasága” elsőrendű szempont, akkor valamilyen módon biztosítanunk kell azt. Ha rendelkezésünkre áll a megfelelő mennyiségű emberi erőforrás, akkor ezt megtehetjük manuálisan is. Ez viszont kétséges, főként akkor, ha az adatbázis mérete hatalmas (ami napjainkban nem ritka jelenség).

Az említett problémák miatt általában hatékonyabb (mind az idő, mind a költségek szempontjából) valamiféle automatikus adattisztítási módszert alkalmazni. Ezzel nem csak az emberi erőforrást takaríthatjuk meg, de elkerülhetjük a kézi javítás által bevezetett újabb hibák megjelenését is. Szerencsére a feladatra már léteznek automatikus módszerek, ezeket a következő szakaszban tekintem át. A diplomamunkám tárgya egy olyan új módszer kifejlesztése, amelyben az adatbázis aktuális tartalmából kikövetkeztetett tudást használjuk fel a hibák, ha nem is tökéletes, de elfogadható javítására. A tudás felfedezésére megalapozott adatbányászati módszereket használhatunk.

A következő szakaszokban röviden áttekintem, melyek azok a megoldások, amelyeket sikeresen felhasználhatunk az adattisztításhoz szükséges információk megszerzésére. Ezek közül részletesen tárgyalom a funkcionális függőségek felfedezését, hiszen a későbbi fejezetekben egy erre épülő algoritmust mutatok be. Ezután áttekintem, milyen módszereket alkalmaznak ma az adatok tisztítására, kiemelve ezek erősségeit és gyengeségeit.

2 Adattisztítás

Az adattisztítás feladata az adatokban található hibák felderítése és kijavítása. A cél az, hogy az adatokban minőségbeli javulás következzen be. Hibák bármilyen adatbázisban jelentkezhetnek, de még égetőbb a piszkos adatok problémája, amikor különböző adatforrások tartalmának integrációjára van szükség. Ez a helyzet akkor, amikor adattárházakat töltünk fel adatokkal. Az adattárházak általában több forrásból táplálkoznak. Ezekben a forrásokban gyakran megtalálhatók ugyanazok az adatok különböző reprezentációkban (hogy csak egy problémát említsek). Ha ezt nem vesszük észre az integráció során és nem egységesítjük az ábrázolást, akkor az adattárházban redundáns rekordok jelennek meg.

Az adattárházak nagyméretű adatbázisok folyamatosan frissülő tartalommal. Az adattárházakat főként döntéstámogatásra használják, így különösen fontos az adatok megbízhatósága. Az ETL (Extraction, Transformation and Loading) folyamat célja az adatok kinyerése a heterogén forrásokból, azok transzformációja valamilyen egységes formátumra és betöltése az adattárházba. Ebben a folyamatban, a sémaszintű megfontolások mellett, jelentős szerepet kap az adattisztítás is, hiszen az adattárházba bekerülő adatoknak az elérhető legjobb minőségűeknek kell lenniük.

A megfelelő adattisztítási módszernek az alábbi követelményeknek kell megfelelnie (Rahm & Do, 2000.):

- *Fedezze fel az adatokban a kritikus hibákat*, inkonzisztenciákat, valamint legyen alkalmas azok kijavítására! Fontos megjegyezni, hogy egy jó eljárás nem csak egyedi adatforrásokban képes erre, hanem több adatforrás esetén is, azok integrációja előtt.
- *Legyen támogatható automatikus eszközökkel!* Mivel az adatforrások mérete hatalmas lehet, a kézi munkával végzett adattisztítás kivitelezhetetlen. Hiába találja meg tehát egy módszer az összes hibát az adatokban, ha nem lehet automatikusan megvalósítani, akkor a gyakorlatban használhatatlan.
- Az adattisztítás mellett szükség van a *sémaszintű átalakítások*ra is, csak így érhetünk el megfelelő eredményt. Az adattisztítási módja mellett tehát meg kell fogalmaznunk azt is, hogy az integrált adatokat milyen sémában kívánjuk tárolni, illetve le kell írnunk, milyen transzformációk szükségesek ahhoz, hogy a források adatai a célséma formátumára átalakíthatók legyenek.

- Az adattisztítás *legyen tárgyterülettől független* (azaz ne csak egy specifikus terület adataira működjön helyesen) (Maletic & Marcus, Automated Identification of Errors in Data Sets, 2000).

Ha egy adattisztítási módszer rendelkezik a felsorolt tulajdonságokkal, akkor valószínűleg a gyakorlatban is jól alkalmazható. Mindazonáltal néha szükség lehet a folyamatos emberi visszajelzésre is (Raman & Hellerstein, 2001).

2.1 A hibák típusai

A lehetséges hibák típusait (Rahm & Do, 2000.) több szempont szerint csoportosítja. A hiba megjelenésének szintje szerint beszélhetünk séma- és példányszintű hibákról.

A sémaszintű hibák magában a sémában jelentkeznek, ezek körültekintő tervezéssel elkerülhetők (vagy javíthatók). A példányszintű hibák azok, amelyek a konkrét adatokban jelennek meg. A későbbiekben az ilyen típusú hibák felderítésére keresek megoldásokat.

A hibák felderítésénél ezen kívül fontos szempont, hogy egy vagy több forrással dolgozunk-e. Több forrás esetén az adatok integrációját meg kell előznie azok tisztításának.

2.1.1 Egy forrás esetén jelentkező hibák

Ha az adatok egyetlen forrásból származnak, akkor nem kell foglalkoznunk azok integrációjával.

A relációs adatbázis-kezelő rendszerek megszorításokat tesznek az adatokra. Ilyenek az integritási és az attribútumok értékkészletére vonatkozó megszorítások, amelyek a sémában fogalmazódnak meg. Emellett számos olyan megszorítás típus is létezik, amit nem lehet a séma szintjén megfogalmazni. Például nem, vagy csak körülményesen fogalmazható meg sémaszinten az, hogy egy attribútum csak bizonyos előre ismert értékeket vehet fel.

A sémaszintű hibáknak tehát általában két oka lehet: a rosszul megfogalmazott séma és az adatbázis-kezelő rendszer funkcionális korlátozottsága.

A sémaszintű hibák közé tartoznak a következők:

- *Nem megengedett attribútum értékek*. Például a 2009.13.40. érték szerepel egy dátum típusú oszlopban. Az érték nem része a dátum típus tartományának.
- *Attribútumok közötti függőségek megsértése*. Például egy személyi adatokat tároló adatbázis táblában a SZULETESI_DATUM attribútum értéke alapján számított kor és a KOR attribútum értéke nem egyezik meg.

- *Egyediségi megszorítás megsértése*. Például egy személyeket tartalmazó adatbázisban két különböző személyhez ugyanaz a személyi szám tartozik.
- *Hivatkozási integritás megsértése*. Az egyik tábla egy sora olyan elsődleges kulcs értékre hivatkozik egy másik táblában, ami ott nem szerepel.

Az adatokban (tehát nem sémaszinten) megjelenő hibák közül az alábbiakat emelhetjük ki:

- *Hiányzó értékek*. Az értékek hiányát vagy valamilyen alapértelmezett értékkel, vagy NULL értékkel jelzik.
- *Elgépelések*. Például „Budapest" helyett „Buapest". Az ilyen hibák implicit módon sérthetik a sémaszintű megszorításokat (például a városnévre vonatkozó egyediségi megszorítást), hiszen a helyes és az elgépelt érték egyszerre szerepelhet az adatbázisban, holott logikailag mindkettő ugyanazt az objektumot jelöli.
- *Rövidítések, kódolt értékek*. Ilyen hibával akkor találkozhatunk, amikor egy attribútum értékeit úgy állítják elő, hogy azok több tulajdonságra vonatkozó információt is tartalmazzanak. Például a VASARLO_TIPUSA mező értékei „F32" alakúak, ahol az első karakter jelzi, hogy mi a vásárló neme, a további karakterek pedig a korára utalnak.
- *Beágyazott értékek*. Egy attribútum több értéket is tartalmaz. Például a CIM oszlopban a következő érték áll: „4032, Debrecen, Lehel út 12". Ez a hiba ráadásul sérti az 1NF szabályát, miszerint az attribútum-értékeknek atominak kell lenniük.
- *Rossz oszlopba írt értékek*. Például a VAROS oszlopban a „USA" érték szerepel.
- *Attribútumok közötti függőségek megsértése*. Például a VAROS oszlopban „Debrecen" szerepel, az IRANYITO_SZAM oszlopban pedig „1034" (egy budapesti irányítószám). Ez a hibatípus abban különbözik a sémaszintű hibák közt felsorolt párjától, hogy amíg ott a séma szintjén meg tudjuk fogalmazni az attribútumok között fennálló összefüggéseket, addig itt csak azt tudjuk, hogy bizonyos oszlopok értékei meghatározzák egymást. Ezt a sémában nem tudjuk megfogalmazni.
- *Duplikált rekordok*. Ha a séma jól van megfogalmazva, akkor ez a hiba általában elkerülhető. Előfordulhat viszont, hogy ugyanaz az érték eltérő reprezentációkban szerepel az adatbázisban, így nehéz felismerni, hogy a két rekord azonos. Például a NEV oszlop értéke az egyik sorban „K. János", egy másik sorban pedig „Kovács

János”. Ez a típusú hiba főként akkor jelentkezhet, ha több forrásból vesszük az adatokat.

- *Ellentmondó rekordok*. Például az egyik rekordban Kovács János 10 éves, egy másikban pedig 90.
- *Hibás hivatkozások*. Egy rekord egy másik adatbázis tábla olyan elsődleges kulcsára hivatkozik, amely nem létezik.

A fent felsorolt hibákon kívül természetesen más típus is előfordulhat, de ezek a legáltalánosabbak.

A lehetséges hibák száma tovább nő, ha az adatok több forrásból származnak. Ebben az esetben fel kell készülnünk többek között az eltérő reprezentációkból és az eltérő sémákból adódó problémákra. Ráadásul a források maguk is tartalmazhatnak piszkos adatokat, ezért az integráció előtt már tisztításra van szükség.

Több forrás esetén az alábbi problémák nehezítik tovább az adattisztítást:

- A különböző forrásokban ugyanazok az attribútumok különböző reprezentációkban szerepelhetnek.
- Ugyanazok az attribútumok különböző szintig lehetnek összesítve. Például az egyik táblában az eladásokat telephelyenként összegzik, míg a másikban régiónként.
- A különböző források olyan összesített adatokat tartalmazhatnak, amelyek eltérő időpontokra vonatkoznak. Például eladások a múlt héten, illetve eladások egy adott napon.

Az adattisztítás több forrás esetén kiegészül néhány újabb feladattal:

- El kell döntenünk, hogy a különböző adatbázisok rekordjai közül melyek reprezentálják ugyanazt az objektumot. Ezt nevezzük az objektum-identitás problémájának (Hern'andez & Stolfo, 1995.).
- A duplikációkat meg kell szüntetnünk.
- A különböző forrásokból származó, egymást kiegészítő adatokat valamilyen formában egyesítenünk kell. Például ha az egyik adatbázisban szerepel egy vásárló neve és kora, egy másik adatbázisban pedig további adatokat találunk róla, akkor célszerű ezeket összevonnunk az integráció során.

A lehetséges hibák áttekintése után vizsgáljuk meg, milyen módszerek állnak rendelkezésünkre azok kijavításához!

2.2 Adattisztítási módszerek

Adattisztításra számos módszert használhatunk. Ezek között vannak olyanok, amelyek tetszőleges tárgyterület adatain jól működnek, másoknál viszont szükség van a doménre vonatkozó információk előzetes ismeretére.

A következő szakaszokban az adattisztítás fontosabb megközelítéseit mutatom be.

2.2.1 Parsing

Az adatok szintaktikai hibáinak felismerésére használhatjuk (Müller & Freytag, 2003). Ehhez meg kell adnunk egy nyelvtant, ami alapján az ellenőrzés történik. A módszer azt vizsgálja, hogy az adatbázis adatai levezethetők-e ebből a nyelvtanból. Amelyik adat levezethető, azt szintaktikailag helyesnek tekintjük.

Az adattisztítás szempontjából a szintaktikai helyességet vizsgálhatjuk rekord és attribútum szinten. Azaz megfogalmazhatunk nyelvtanokat az egyes attribútumok lehetséges értékeire (például megadunk egy reguláris kifejezést a dátum formátumára), illetve az egyes rekordok kinézetére. Relációs adatbázisok esetén a második lehetőséget ritkán használják, hiszen a sémák „kinézete” megfelelően leírható az adatbázis sémával. A gyakorlatban tehát a módszer abban segít, hogy megtaláljuk azokat az attribútum értékeket, amelyek nem illeszkednek az adott attribútum értéktartományába.

Ha a szintaktikailag helytelen adatokat felderítettük, akkor lehetőség van azok kijavítására és törlésére. A javításhoz valamilyen hasonlósági mértéket használhatunk. Ez sztringek általában valamilyen Levenshtein távolságfüggvény (Bar-Yossef, Jayram, Krauthgamer, & Ravi, 2004.), numerikus értékek esetén pedig tetszőleges távolsági mérték.

Az elemzéshez használt nyelvtan megkonstruálható automatikusan is, az adatbázis tartalma alapján, tanuló algoritmusokkal. Ennek hátránya, hogy a hibás adatok magas relatív előfordulása helytelen eredményhez vezethet.

2.2.2 Integritási megszorítások kikényszerítése

A módszer célja azon adatok módosítása, amelyek nem felelnek meg az integritási megszorításoknak. A módosítás jelenthet törlést, frissítést és beszúrást

A megközelítésnek van néhány hátránya. A helyes eredményhez szükséges, hogy el tudjuk dönteni, milyen adatbázis műveletek szükségesek ahhoz, hogy az adatbázis tartalma kielégítse

az integritási megszorításokat. Ez általában nem könnyű feladat. A módszer ezen kívül azt feltételezi, hogy az adatbázisban túlnyomórészt olyan adatok szerepelnek, amelyek megfelelnek az integritási megszorításoknak. Sokszor azonban ez nem teljesül, a hibás adatok száma igen magas is lehet. Ráadásul a megközelítés az adatok javítása helyett az adatbázis tartalmának megváltoztatására koncentrál, így hasznos adatok elvesztéséhez vezethet.
Az említett problémák miatt a módszert nem alkalmazzák egyedüliként az adattisztításban, főleg támogató szerepben jelenik meg (Müller & Freytag, 2003).

2.2.3 Duplikációk megszűntetése

Az ebbe a csoportba tartozó módszerek közös tulajdonsága, hogy az adatbázis összes sorát összehasonlítják a többivel, azaz viszonylag költségesek. Az egyes megoldások abban térnek el, ahogyan a hatékonyságot javítani próbálják. (Hern'andez & Stolfo, 1995.) például az adatok kulcs szerinti rendezésével éri el, hogy a szükséges hasonlítások száma csökkenjen.

2.2.4 Statisztikai megközelítések

Ezek a módszerek az adatok bizonyos statisztikai tulajdonságai alapján próbálják kiszűrni a hibás elemeket. (Maletic & Marcus, Data Cleansing: Beyond Integrity Analysis, 2000.) ide sorolja az adatbányászati módszereket is, viszont nem tesz említést a funkcionális függőségek felhasználási lehetőségeiről.
Az idézett szerzők meglátása szerint a hibás adatok megkeresése visszavezethető a kilógó adatok kiszűrésére. Ha az adatok jelentős többsége megfelel valamilyen kritériumnak, akkor joggal tekinthetjük a többi adatot hibásnak. Az adattisztítás feladata tehát az, hogy megállapítsa, mik azok a közös tulajdonságok, amelyek a „normális” értékeket jellemzik. Ha ezt sikerül feltárnunk, akkor a hibás értékek viszonylag könnyen kijavíthatók.
A korábban említett módszerek nem képesek az attribútumok közötti összefüggések felfedezésére, míg a statisztikai megközelítések igen. A feltárt összefüggések felhasználásával általában nagyobb javulást érhetünk el, mint a korábban említett eljárásokkal.
A következő szakaszokban bemutatom a kilógó értékek megtalálására kínálkozó lehetőségeket.

2.2.4.1 Statisztikai függvények

Bizonyos statisztikai függvények alkalmasak arra, hogy a kilógó értékeket felfedezésében használjuk őket. A célnak tökéletesen megfelelnek például a különböző középértékek, a

standard szórás és az értéktartomány elemzés. Ezen kívül felhasználhatjuk az attribútumok konfidencia intervallumairól megszerezhető tudást is.

2.2.4.2 Klaszterezés

A klaszterezés célja, hogy a vizsgált objektumokat (esetünkben az adatbázis sorait) jól elkülöníthető csoportokba sorolja. Ezekről a csoportokról kevés a priori tudással rendelkezünk: nem tudjuk, hogy az egyes csoportokba milyen tulajdonságú elemek tartoznak (Dr. Abonyi, 2006).

Egy jó klaszterező algoritmus olyan csoportokat alakít ki, amelyeken belül az objektumok nagyon hasonlítanak egymásra, viszont az eltérő csoportok objektumai eléggé különböznek. A hasonlóság mérésére több mértéket használhatunk. Az alkalmazandó mérték kiválasztása függ az objektumok típusától és a klaszterezési folyamat kulcsfontosságú része.

A klaszterező algoritmusok két nagy csoportját jelentik a hierarchikus és a particionáló algoritmusok.

A hierarchikus algoritmusok egyszerre több csoportosítást is eredményül adnak. Kezdetben egy csoportként kezelik a teljes objektumsokaságot, majd a választott hasonlósági mérték alapján ezt alcsoportokra osztják egészen addig, amíg el nem érik a megfelelő számú klasztert. A hierarchikus algoritmusok közül a legismertebbek a CURE, a ROCK és a CHAMELEON.

A partícionáló módszerek csak egy lehetséges csoportosítást adnak eredményül és azt is előre meg kell mondanunk, hogy hány csoportot állítsanak elő. Ezek az algoritmusok kezdetben a megadott számú csoportba sorolják az elemeket, majd a választott hasonlósági mértékek alapján a csoportok között mozgatják az objektumokat addig, amíg további mozgatás már a csak a csoportok „elromlásával” jár.

A hierarchikus módszerek mellett érdemes megemlíteni, hogy léteznek fuzzy logikát és neurális hálókat használó, illetve sűrűség alapú (Bodon, 2006.) algoritmusok is.

A klasztererezés eredményéből megállapítható, hogy melyek a kilógó értékek, hiszen ezek azok az objektumok, amelyek kiugróan távol vannak saját csoportjuk többi tagjától.

A felsorolt algoritmusok közös hátránya, hogy igen nagy a számítási igényük. Ez nagyméretű adatbázisok és sok attribútum esetén megakadályozhatja a módszer sikeres alkalmazását.

2.2.4.3 Asszociációs szabályok

Az asszociációs szabályok feltárásával az adatbázisban előforduló gyakori elemhalmazokra fogalmazhatunk meg állításokat. Olyan szabályokat keresünk, amelyek azt állítják, hogy

azokban a sorokban, ahol egy *I* érték előfordul, nagy valószínűséggel előfordul egy *J* érték is. Például azok a vásárlók, akik pelenkát vesznek, nagy valószínűséggel sörből is visznek néhány üveggel.

Olyan asszociációs szabályokat keresünk, amelyek kellő megalapozottsággal és valószínűséggel rendelkeznek. Egy szabály megalapozott, ha *I* és *J* egy előre megadott értéknél nagyobb százalékban fordul elő az adatbázis sorai között. Egy szabály valószínűsége nagy, ha az *I* nem fordul elő egymagában sokkal többször az adatbázisban, mint *I* és *J* együtt. Ha egy szabály megfelel mindkét kritériumnak, akkor érvényes, ha a valószínűsége 1, akkor pedig egzakt szabálynak nevezzük. Az egzakt szabályok funkcionális függőségeknek felelnek meg.

Az asszociációs szabályok feltárása során a célunk a legalább érvényes szabályok feltárása. Az előállított szabályok száma nagy lehet (rengeteg szabályt kapunk, ha a küszöbértékeket nem megfelelően választjuk meg). Emiatt minden algoritmusnál szükség van ún. érdekességi értékek kiszámítására, amelyek megmutatják, hogy érdemes-e egy adott szabállyal foglalkozni (Bodon, 2006., old.: 124-128.).

Az asszociációs szabályokat használhatjuk a kilógó értékek felderítésére, hiszen ha találunk egy érvényes asszociációs szabályt, akkor azokat a rekordokat, amelyekben az asszociációs szabály nem érvényes, tekinthetjük hibásnak. Ezen kívül a módszert sikeresen alkalmazták duplikációk kiszűrésére biológiai adatbázisokban(Koh, Lee, Khan, Tan, & Brusic, 2004.).

2.2.4.4 Funkcionális függőségek felderítése

A funkcionális függőségek felderítésével és azok adattisztítási felhasználásával egy külön fejezetben foglalkozom. Itt csak azt emelném ki, hogy léteznek olyan módszerek, amelyekkel a funkcionális függőségek teljesülését ki tudjuk kényszeríteni. Ha sikerül megtalálnunk ezeket a függéseket, akkor az említett módszereket felhasználva elérhetjük az adatok minőségének javulását, tisztulását.

3 Funkcionális függőségek feltárása

Ebben a fejezetben a funkcionális függőségek feltárásának módszereit, alapfogalmait és nehézségeit mutatom be. Áttekintem továbbá, hogyan lehet a feltárt függőségeket az adatok tisztításában alkalmazni. Fontos azonban megjegyezni, hogy a funkcionális függőségek feltárásának a relációs adatbázisok más részterületein is léteznek felhasználási lehetőségei. Az

adatbázisról ily módon megszerzett információk alkalmazhatók például lekérdezés-optimalizációra is.

3.1 Funkcionális függőségek

A funkcionális függőség a relációs adatbázisok alapvető fogalma. Eredetileg az adatbázistáblák normalizálására, a redundancia csökkentésére vezették be. A funkcionális függőségekkel olyan a priori tudást fogalmazhatunk meg, amelyekkel ezeket a célokat elérhetjük.

A funkcionális függőség fogalmát formálisan a következőképpen írhatjuk le (Maier, 1983., old.: 42-44.):

1. **definíció.** *Legyen adott egy* ***R*** *relációs séma. Legyen* ***r*** *egy reláció az* ***R*** *sémán és* ***X*** *és* ***Y*** *legyen* ***R*** *részhalmaza. Azt mondjuk, hogy az* ***r*** *reláció kielégíti az* $X \rightarrow Y$ *funkcionális függőséget, ha* ***X*** *tetszőleges* ***x*** *értékére a* $\pi_Y(\sigma_{X=x}(r))$ *eredmény relációja legfeljebb egy sort tartalmaz.*

Másképpen megfogalmazva, az $X \rightarrow Y$ funkcionális függőség akkor és csak akkor áll fenn **R**-en, ha **r** minden t_1 és t_2 sorára, amelyre $t_1[X] = t_2[X]$ teljesül, teljesül $t_1[Y] = t_2[Y]$ is. Azaz az egyes sorok **X** attribútum-halmazon felvett értékéből *következtethetünk* az **Y** attribútum-halmazon felvett értékére.

Fontos aláhúzni, hogy bár a funkcionális függőségeket mindig egy relációs sémán értelmezzük, könnyen előfordulhat, hogy ugyanazon séma különböző példányain más és más funkcionális függőségek teljesülnek.

A funkcionális függőségek speciális esete a szuperkulcs.

2. **definíció.** *Legyen* ***R*** *egy relációs séma és* ***X*** *egy valódi részhalmaza* ***R****-nek. Legyen továbbá* ***r*** *egy reláció az* ***R*** *séma felett. Ha* ***r*** *kielégíti az* $X \rightarrow R$ *funkcionális függőséget, akkor azt mondjuk, hogy* ***X*** *egy szuperkulcs. Ha* ***X****-nek nincs olyan valódi* ***Y*** *részhalmaza, amelyre* $Y \rightarrow R$ *teljesül, akkor* ***X*** *kulcs.*

A definícióból és a relációs adatbázisok tulajdonságaiból következik, hogy egy relációban nincs két olyan sor, amely megegyezik a szuperkulcs attribútumokon, azaz egy szuperkulcs egyértelműen azonosítja a reláció sorait.

3.1.1 Armstrong axiómák

Az Armstrong axiómák olyan levezetési szabályok, amelyekkel már ismert funkcionális függőségekből vezethetünk le újakat.

Az Armstrong-axiómák rendszere *helyes és teljes*. Helyes abban az értelemben, hogy ha a szabályok segítségével egy **X → Y** funkcionális függőség levezethető egy **F** funkcionális függőség halmazból, és **F** minden függősége teljesül egy **R** séma **r** relációjában, akkor **X → Y** is teljesül **r**-ben (ahol **X,Y ⊆ R**). Teljes abban az értelemben, hogy ha egy **X → Y** funkcionális függőség teljesül **r**-en, akkor az le is vezethető **F**-ből.

Az Armstrong axiómák a következők:

1. Reflexivitás: **X → X** mindig teljesül.
2. Augmentáció: ha **X → Y** teljesül, akkor **XZ → Y** is teljesül.
3. Additivitás: ha **X → Y** és **X → Z** teljesül, akkor teljesül **X → YZ** is.
4. Projektivitás: ha **X → YZ** teljesül, akkor **X → Y** is teljesül.
5. Tranzitivitás: ha **X → Y** és **Y → Z** teljesül, akkor **Y → Z** is teljesül.
6. Pszeudotranzitivitás: ha **X → Y** és **YW → Z** teljesül, akkor teljesül **XW → Z** is.

A fenti szabályokban **X, Y, Z, W ⊆ R.** A szabályokat **R** egy **r** relációjára értjük.

Az elmondottakból az következik, hogy egy bányászati algoritmusnak elég azokat a funkcionális függőséget megtalálnia, amelyből a többi levezethető. Így az algoritmus hatékonyabbá tehető. Az ilyen funkcionális függőség halmazok leírásához be kell vezetnünk a lezárt fogalmát.

3. **definíció.** *Legyen adott egy **R** séma, egy **r** reláció **R** felett és legyen **F** olyan funkcionális függőségek halmaza, amelyeket **r** kielégít. **F** lezártja, $\boldsymbol{F^+}$, az a legkisebb funkcionális függőség halmaz, amely tartalmazza **F**-et, és amely elemeire tetszőleges Armstrong-axiómát alkalmazva nem kaphatunk olyan funkcionális függőséget, amely nem eleme $\boldsymbol{F^+}$-nak.*

Egy funkcionális függőség halmaz lezártját kiszámíthatjuk az Armstrong-axiómák többszöri alkalmazásával. A szabályokat addig kell alkalmaznunk, amíg sikerül levezetnünk olyan funkcionális függőségeket, amelyek még nem szerepelnek $\mathbf{F^+}$-ban. Bizonyítható, hogy egyedül a reflexivitás, az augmentáció és a tranzitivitás alkalmazásával minden funkcionális

függőség előállítható, amely levezethető egy adott funkcionális függőség halmazból. Az Armstrong-axiómák ezen részhalmaza önmagában is helyes és teljes (Date, 2003., old.: 338-341.).
A függőségek feltárása szempontjából különösen fontosak az alábbi fogalmak.

4. **definíció.** *Legyen adott funkcionális függőségek két halmaza,* ***F*** *és* ***G****. Ha* $\boldsymbol{F^+ \subseteq G^+}$*, akkor azt mondjuk, hogy* ***G*** *az* ***F*** *fedése. Ha* $\boldsymbol{F^+ = G^+}$*, akkor azt mondjuk, hogy* ***F*** *és* ***G*** *ekvivalens (*$\boldsymbol{F \equiv G}$*), azaz pontosan ugyanazok a funkcionális függőségek vezethetők le mindkét halmazból.*

   Egy ***F*** *funkcionális függőség halmaz nem redundáns, ha nincs olyan* ***F'*** *valódi részhalmaza, amelyre* $\boldsymbol{F' \equiv F}$*.* ***F*** *egy nem redundáns fedése* ***G****-nek, ha* ***F*** *fedése* ***G****-nek és emellett* ***F*** *nem redundáns.*

Egy nem redundáns funkcionális függőség halmaz méretét tovább csökkenthetjük, ha a benne szereplő funkcionális függőségekből attribútumokat távolítunk el.

5. **definíció.** *Egy* ***F*** *funkcionális függőség halmazt redukáltnak nevezünk, ha:*
   - *minden funkcionális függőségének jobb oldalán egyetlen attribútum áll.*
   - *semelyik funkcionális függőségének bal oldaláról nem hagyhatunk el attribútumokat anélkül, hogy* $\boldsymbol{F^+}$ *megváltozna.*

6. **definíció.** *Egy* ***F*** *funkcionális függőség halmaz minimális, ha nincs olyan* $\boldsymbol{G \equiv F}$*, hogy* ***G*** *kevesebb funkcionális függőséget tartalmaz, mint* ***F****.*

Sajnos nincs olyan algoritmus, amivel hatékonyan meghatározhatnánk egy minimális fedést, viszont létezik ilyen algoritmus a nem redundáns fedésekre (Maier, 1983., old.: 79.).
A funkcionális függőségek feltárása során célunk az ilyen nem redundáns, redukált funkcionális függőség halmazok felderítése. Ezekből később levezethetjük a többi funkcionális függőséget, ha szükségünk van rájuk.

3.2 Közelítő funkcionális függőségek

A valós adatbázisokban gyakran funkcionális függőségek helyett közelítő funkcionális függőségekkel találkozunk. Ezek olyan függőségek, amelyek „közel állnak ahhoz", hogy funkcionális függőségek legyenek, csak a reláció kevés sora nem elégíti ki őket.

Ezt a közelséget különböző közelítő mértékekkel határozhatjuk meg. Egy közelítő mérték egy olyan φ függvény, ami két attribútum halmazt (egy lehetséges függőség bal és jobb oldalát) egy [0,1] intervallumba eső számra képez le és $\varphi(X \rightarrow Y) = 0$ akkor és csak akkor, ha a reláció kielégíti az $X \rightarrow Y$ funkcionális függőséget.

Sokféle közelítő mértéket ismerünk. Az egyik legismertebb a TANE algoritmus g_3 mértéke (Huhtala, Karkkainen, & Toivonen, 1999.). A g_3 értékét megkapjuk, ha vesszük azoknak a soroknak a számát, amelyeket el kellene távolítani a relációból ahhoz, hogy $X \rightarrow Y$ teljesüljön, és ezt elosztjuk a reláció sorainak számával.

A közelítő funkcionális függőségek felderítésének feladata, hogy megtaláljon minden olyan függőséget, amelyre a közelítő metrika egy előre megadott ε küszöbértéknél kisebb.

A közelítő funkcionális függőségeket felhasználhatjuk adattisztításra, viszont a küszöbértékek megfelelő megválasztására mindig oda kell figyelnünk, nehogy a közelítés a tisztítás pontosságának rovására menjen.

3.3 Problémák

A funkcionális függőségek feltárása kapcsán számos problémára kell felkészülnünk. Ezek egyrészt az adatok tulajdonságaiból, másrészt az algoritmusok jellemzőiből adódnak.

A feladatot megnehezíti, hogy az adatbázis táblák (különösen adattárházak esetén) hatalmas méretűek, akár több millió sorosak is lehetnek. Ha érvényes funkcionális függőségeket akarunk találni, akkor vagy minden sort, vagy azok jelentős részét fel kell használnunk a függőségek kinyeréséhez. Ez azt jelenti, hogy nem tudjuk elkerülni azt, hogy nagy adathalmazzal kelljen dolgoznunk. Emiatt az algoritmusoknak a lehető leghatékonyabban kell működniük – márpedig (ahogy később látni fogjuk) a hatékonyság javítása igen nehéz.

Egy másik probléma (ami kapcsolódik az előzőhöz), hogy az adatbázisok a feltárás alatt is használatban lehetnek (Bell, 1995.). Tehát az algoritmusokat úgy kell megtervezni és megvalósítani, hogy azok a lehető legkisebb mértékben foglalják le az adatbázis erőforrásait. Ez szintén nehéz feladat, mivel azt jelenti, hogy a funkcionális függőségek kinyerését minél

kevesebb adat felhasználásával kell megoldanunk, miközben a kinyert függőségeknek érvényesnek kell maradniuk.

Az előző problémákhoz kapcsolódóan felvetődik a pontosság kérdése is. Ahogy később látni fogjuk, a funkcionális függőségek felderítéséhez teljes adathalmaz helyett használhatjuk annak egy részhalmazát is, ezzel javítva a hatékonyságot. Ilyenkor el kell döntenünk, hogy mennyire bízunk az így megszerzett információkban. A közelítő funkcionális függőségekre ugyanez igaz. A pontosság meghatározása különösen fontos kérdés, amikor a funkcionális függőségeket adattisztításra használjuk, hiszen az adatok minőségének javulása csak megfelelő pontosság mellett érhető el.

A feltárt függőségek pontatlanok lehetnek akkor is, ha az adatok között túl nagy részben fordulnak elő hibásak. Ilyenkor az algoritmusok helytelen eredményeket adnak, amit nem érdemes felhasználni semmilyen célra. Ha ugyanis ilyen információkkal kezdenénk neki az adattisztításnak, azzal csak rontanánk az adatok minőségén. Sajnos az ilyen jellegű (a hibás sorok magas számából adódó) pontatlanságokat automatikusan nem lehet kiszűrni, ilyen esetekben csak a manuális beavatkozás segíthet.

Végezetül, az adatbázistáblák nagy mérete mellett problémát jelent az attribútumok magas száma. Mint látni fogjuk, a legtöbb algoritmus érzékeny erre a tulajdonsága. A problémára bizonyos attribútumok figyelmen kívül hagyása nyújthat megoldást. Ezt úgy valósíthatjuk meg, hogy valamilyen érdekességi mértéket kiszámítunk az egyes attribútumokra, és amelyeknél az érték nem ér el egy határt, azokat nem használjuk fel. Emellett maga az adattisztítás kezdeményezője is dönthet úgy, hogy csak bizonyos attribútumokkal kíván foglalkozni.

3.4 Lehetséges megközelítések

A funkcionális függőségek feltárására napjainkig számos algoritmus született. Ezeket célszerű valamilyen csoportosítás mentén áttekinteni. Ebben az alfejezetben erre teszek kísérletet. Az algoritmusokat a következőkben a jelöltek generálása, a jelöltek közötti keresés, a felhasznált adatok milyensége és mennyisége alapján csoportosítom.

3.4.1 Alulról-felfelé és felülről-lefelé haladó algoritmusok

Az algoritmusok a haladási irányuk szerint lehetnek felülről-lefelé és alulról-felfelé haladók (Lim & Harrison, 1997.).

Az *alulról-felfelé* haladó algoritmusok először megvizsgálják az adatbázisban található adatokat, összehasonlítanak minden sort az összes többivel, és elemzik, hogy melyek azok a függőségek, amelyeket a reláció kielégít. Az így megszerzett tudás alapján döntenek bizonyos függőségek elvetéséről és mások megtartásáról.

A megközelítés hátránya, hogy igen sok hasonlítást igényel, így erőforrás igényes.

A *felülről-lefelé* algoritmusok először funkcionális függőség jelölteket generálnak, majd ellenőrzik ezek érvényességét. Mivel a lehetséges jelöltek száma exponenciális, így az egyes algoritmusok különböző metsző eljárásokat alkalmaznak, amivel a jelöltek számát próbálják csökkenteni.

Az ellenőrzéshez az adatbázis tartalmát használjuk. Ez történhet a sorok páronkénti hasonlításával, illetve a sorok rendezésével. Ez utóbbi hatékonyága valamivel jobb, $O(n \log n)$ idejű, ahol n a sorok száma, míg az első esetén $O(n^2)$. A felülről-lefelé algoritmusok tehát a metsző eljárásokon kívül az érvényesség-ellenőrzésben is eltérnek. Mivel sok jelölt érvényességének vizsgálata mindenképpen időigényes, ezért különösen fontos a jelöltek számának csökkentése, a megfelelő metsző eljárások alkalmazása.

3.4.2 Partíció, differencia halmaz és funkcionális függetlenségek

Az alulról-felfelé haladó algoritmusok tehát először átvizsgálják az adatbázis tartalmát, kinyernek belőle bizonyos információkat, majd ezek alapján előállítják azokat a funkcionális függőségeket, amelyek az adatbázis aktuális tartalma alapján fennállnak. A naiv algoritmus az összes sort összehasonlítja és ez alapján következtet. Ez nagy adattömeg esetén nyilvánvalóan kivitelezhetetlen, így más megoldások után kell néznünk.

A hatékonyabb algoritmusok ehelyett az egyes attribútumokon megegyező vagy eltérő sorokról gyűjtenek össze információkat és ezek alapján végzik a függőségek feltárását. Ezen módszerek közül hármat emelek ki: az egyik az egyező attribútum-értékek segítségével partíciókra osztja az adatbázist, a másik a különböző attribútum-értékek mentén negatív fedéseket alakít ki, a harmadik megtalálja azokat a funkcionális függőségeket, amelyek biztosan nem teljesülnek és ezekből következtet. A három megközelítés között szoros összefüggés van.

3.4.2.1 Partíciók

A partícionálást alkalmazó algoritmusok közül a legfontosabb (és egyben a legnépszerűbb is) a TANE algoritmus (Huhtala, Karkkainen, & Toivonen, 1999.), de ezt a konstrukciót használja többek között az FD_MINE is (Hong, Hamilton, & Butz, 2002.) és az

FD_DISCOVER (Atoum, Bader, & Awajan, 2008.) is. A TANE algoritmus alkalmas funkcionális és közelítő funkcionális függőségek feltárására egyaránt. Működésében a partíciók tulajdonságait használja ki, megértéséhez tehát meg kell ismernünk a partíciók fogalmát.

Legyen adott egy reláció két sora, **t** és **u**, valamint egy **X** attribútum halmaz. $\boldsymbol{t}$ és $\boldsymbol{u}$ egyenlők $\boldsymbol{X}$-re nézve, ha $\boldsymbol{t}[\boldsymbol{A}] = \boldsymbol{u}[\boldsymbol{A}]$ minden $\boldsymbol{A} \in \boldsymbol{X}$ −re. Minden **X** attribútum halmaz ekvivalencia osztályokat generál a reláción. Pontosan azok a sorok tartoznak egy ekvivalencia osztályba, amelyek egyenlők **X**-re nézve. Egy **t** sor $\mathbf{X} \subseteq \mathbf{R}$ attribútum halmazra vonatkozó ekvivalencia osztályát (tehát azt az **X** által generált ekvivalencia osztályt, amelybe **t** tartozik) $\mathbf{[t]_X}$-szel jelöljük:

$$[t]_X = \{u \in r \mid t[A] = u[A] \; \forall \; A \in R \}.$$

7. **definíció.** *A* $\pi_X = \{[t]_X \mid t \in r \}$ *halmazt az* **r** *reláció* **X***-re vonatkozó partíciójának nevezzük.*

Egy reláció **X** attribútum halmazra vonatkozó partíciója tehát tartalmazza az összes **X** által generált ekvivalencia osztályt. Az ekvivalencia osztályok diszjunkt halmazok, és egy halmazon belül a sorok értéke **X**-en megegyezik, viszont két különböző ekvivalencia osztály tetszőleges sorai **X**-en különbözőek. Egy $\boldsymbol{\pi}$ partíció rangjának nevezzük a partíció halmazainak számát, és ezt $|\boldsymbol{\pi}|$-vel jelöljük.

(Huhtala, Karkkainen, & Toivonen, 1999.) bevezeti a partíciók finomításának fogalmát.

8. **definíció.** *Egy* $\boldsymbol{\pi'}$ *partíció finomítása egy* $\boldsymbol{\pi}$ *partíciónak, ha* $\boldsymbol{\pi'}$ *minden eleme részhalmaza* $\boldsymbol{\pi}$ *valamely elemének.*

A partíciók finomítása közvetlenül felhasználható a funkcionális függőségek felderítésére. Ha ugyanis egy $\boldsymbol{\pi_X}$ partíció finomítása egy $\boldsymbol{\pi_A}$ partíciónak, az azt jelenti, hogy minden olyan sor, amely megegyezik **X**-en, megegyezik **A**-n is, ez pedig nem más, mint a funkcionális függőségek definíciója.

(Huhtala, Karkkainen, & Toivonen, 1999.) bizonyítás nélkül közli az alábbi lemmákat.

1. **Lemma** *Egy X → A funkcionális függőség akkor és csak akkor érvényes, ha* $\boldsymbol{\pi_X}$ *finomítása* $\boldsymbol{\pi_{\{A\}}}$*-nak.*

Ez a fent elmondottak logikus következménye. Ez alapján a partíciók segítségével eldönthető, hogy egy funkcionális függőség érvényes-e, vagy sem. Viszont létezik még egy ennél egyszerűbb mód is ennek eldöntésére.

2. **Lemma** *Egy X → Y funkcionális függőség akkor és csak akkor érvényes, ha* $|\boldsymbol{\pi_X}| = |\boldsymbol{\pi_{\{A\} \cup X}}|$.

A funkcionális függőség definíciójából adódik, hogy ha egy **X** attribútumhoz hozzáadunk egy tőle függő **A** attribútumot, az így kapott halmaz nem generál újabb ekvivalencia osztályokat **X**-hez képest. Így a fenti egyszerű egyenlőség használható a funkcionális függőségek felderítésére.
A partícióknak ráadásul van egy igen előnyös tulajdonságuk: az $\boldsymbol{\pi_{X \cup \{A\}}}$ egyszerűen előállítható $\boldsymbol{\pi_X}$ és $\boldsymbol{\pi_A}$ partíciókból.
A partíciókat használó algoritmusok tehát legelőször az adatbázis tartalma alapján meghatározzák az egyes attribútumokhoz tartozó partíciókat, majd jelölteket generálnak, és a jelöltek érvényességét ellenőrzik a partíciók és azok metszetei rangjának felhasználásával, a 2. lemma segítségéve. Ez az alap működés, az egyes megoldások eltérhetnek

- a jelöltek generálásában.
- az ellenőrzött jelöltek számában (metszés).
- a partíciók előállításában.
- az optimalizációs lehetőségekben.

Összességében ezek az algoritmusok hatékonyak, ráadásul hatékonyságuk tovább javítható.

3.4.2.2 Differencia halmazok

A partíciók kiszámítása csak az egyik lehetőség. A FastFDs és a Dep-Miner algoritmusok a differencia halmazok fogalmát használják ugyanerre a célra (Wyss, Gianella, & Robertson, 2001.). A differencia halmazokkal tulajdonképpen a funkcionális függőségeket jellemezhetjük az eredeti definíciótól eltérő módon.

9. **definíció.** *Legyen adott egy **R** séma és egy **r** reláció **R** fölött. **r** két, **t** és **u** sorának differencia halmazát a következőképpen definiálhatjuk:*
   $D(t,u) = \{A \in R \mid t[A] \neq u[A]\}$

Azaz D(t,u) azokat az attribútumokat tartalmazza, amelyeken **t** és **u** eltér. Az **r** reláció differencia halmazait $\mathfrak{D}_r$-rel jelöljük, és a következőképpen definiáljuk:

$$\mathfrak{D}_r = \{D(t,u) \mid t,u \in r \text{ és } D(t,u) \neq \emptyset\}$$

Definiáljuk továbbá $\mathfrak{D}_r$ -t modulo **A** (**A∈ R**):

$$\mathfrak{D}_r^A = \{D - \{A\} \mid D \in \mathfrak{D}_r \text{ és } A \in D\}$$

$\mathfrak{D}_r^A$ tehát tartalmazza azokat az attribútum halmazokat, amelyeken legalább két olyan sor eltér, ami eltér az **A** attribútumon. Ez lehetőséget ad majd a funkcionális függőségek új karakterizációjára, azonban ehhez még szükség van az attribútum halmaz minimális fedésének definíciójára.

10. **definíció.** *Legyen $\mathcal{P}(R)$ az **R** hatványhalmaza és legyen $\mathcal{X} \subseteq \mathcal{P}(R)$. $X \subseteq R$ pontosan akkor fedése $\mathcal{X}$-nek, ha $\forall Y \in \mathcal{X}$-re $X \cap Y \neq 0$. X továbbá minimális fedése $\mathcal{X}$-nek, ha nincs olyan $Z \subset X$, hogy Z fedése $\mathcal{X}$-nek.*

Tegyük fel ezek után, hogy adott egy $X \subseteq R$ attribútum halmaz ($A \notin X$), ami fedése $\mathfrak{D}_r^A$-nak. Ekkor, a definíció szerint, $X \cap D \neq 0$ ($D \in \mathfrak{D}_r^A$). Ez azt jelenti, hogy azok a sorok, amelyek eltérnek az **A** attribútumon, eltérnek az **X** attribútum halmaz legalább egy attribútumán is. Azaz ha két sor megegyezik **X** minden attribútumán, akkor megegyezik **A**-n is. Ez pedig pontosan azt jelenti, hogy az $X \rightarrow A$ funkcionális függőség érvényes a reláció fölött. Ezt fejezi ki a következő tétel (Wyss, Gianella, & Robertson, 2001.).

1. **Tétel** *Legyen $X \subseteq R$, $A \in R / X$ és legyen r egy reláció R felett. $X \rightarrow A$ pontosan akkor érvényes minimális funkcionális függőség r felett, ha X minimális fedése $\mathfrak{D}_r^A$-nak.*

A differencia halmazokat használó algoritmusok tehát először kiszámítják $\mathfrak{D}_r^A$ –t minden **A** attribútumra, majd ezen halmazok alapján meghatározzák a minimális funkcionális függőségeket. Sajnos a differencia halmazok kiszámítása $O(n^2)$ időt igényel, mivel minden sort össze kell hasonlítani. Ráadásul a differencia halmazok kiszámítása közvetetten történik, ami szintén lassítja a folyamatot. Elmondhatjuk tehát, hogy a partíciókat használó algoritmusok hatékonyabbak (bár ez a módszer is optimalizálható).

3.4.2.3 Funkcionális függetlenségek

(Savnik & Flach, 1993.) az érvénytelen funkcionális függőségek fogalmát használja a funkcionális függőségek feltárására.

Egy **X → Y** funkcionális függőség érvénytelen az **r** reláció felett, ha létezik az adatbázisban két olyan sor, **t** és **u**, amelyekre igaz, hogy **t[X] = u[X]** és emellett **t[Y] ≠ u[Y]**. Ilyenkor azt mondjuk, hogy **Y** független **X**-től ((Bell, 1995.) egyébként a funkcionális függetlenség megnevezést használja ugyanerre a fogalomra).

Legyen **X** és **Y** egy attribútum halmaz (**X**,**Y** ⊆ **R**) úgy, hogy **X** ⊆ **Y** és **X → A** (**A** ∈ **R**). Ekkor azt mondjuk, hogy **X** általánosabb, mint **Y**, illetve **Y** specifikusabb, mint **X**. A relációk segítségével definiálhatjuk a minimális negatív fedés fogalmát.

Érvénytelen funkcionális függőségek egy *IF* halmaza minimális negatív fedése az **r** relációnak, ha

1. *IF* minden elemének jobb oldala egyetlen attribútumból áll,
2. minden olyan funkcionális függőséghez, amely érvénytelen **r** felett, létezik egy olyan érvénytelen függőség *IF*-ben, amely specifikusabb nála.

Az így definiált fedés azért minimális, mert ha egy **Y → A** funkcionális függőség érvénytelen, akkor minden olyan **X → A** függőség is érvénytelen, amelyet úgy képzünk, hogy **Y**-ból elhagyunk attribútumokat (azaz **Y → A** specifikusabb, mint **X → A**).

Egy minimális negatív fedésből tehát levezethető minden **r** feletti érvénytelen funkcionális függőség. Az érvénytelen funkcionális függőségek megtalálásához (Savnik & Flach, 1993.) (Savnik & Flach, 1993.) páronként összehasonlítja a reláció sorait és megvizsgálja, hogy az egyes párok milyen funkcionális függőségeket sértenek meg. Ez a gyakorlatban úgy történik, hogy az algoritmus két csoportba sorolja a reláció attribútumait: az első csoportba azok az attribútumok kerülnek, amelyeken a két sor megegyezik, a második csoportba pedig azok az

attribútumok, amelyeke a két sor eltér. A párokhoz tehát a következő két halmazt kell elkészíteni:

$$Z = \{A \in R \mid t[A] \neq u[A] \text{ és } t, u \in r\}$$

és

$$X = \{B \in R \mid t[B] = u[B] \text{ és } t, u \in r\}$$

A két halmaz alapján megmondhatjuk, melyek azok a funkcionális függőségek, amelyek a két sor alapján biztosan érvénytelenek:

$$FI_{t,u} = \{X \rightarrow A \mid A \in Z\}$$

Az érvénytelen funkcionális függőségeket tehát úgy képezhetjük, hogy **r** minden **t** és **u** sorára elkészítjük az **X** és **Z** halmazokat, majd minden sorpárra kiszámítjuk az $FI_{t,u}$ halmazt és ezeket összegezzük. Ezzel azonban nem a minimális negatív fedést kapjuk meg. Az minimális negatív fedés kiszámításához mielőtt egy funkcionális függőséget beletennénk *FI*-be, meg kell vizsgálnunk, hogy tartalmazza-e már attól speciálisabb függőséget. Ha tartalmaz, akkor ez a függőség már nem kerülhet bele a halmazba.

Ha a minimális negatív fedést kiszámítottuk, tovább nincs szükségünk az adatbázisra, *FI* alapján el tudjuk dönteni tetszőleges funkcionális függőség jelöltről, hogy az érvényes-e **r** felett vagy sem. Ugyanis a minimális negatív fedés definíciójából adódik, hogy ha egy funkcionális függőség érvénytelen, akkor *FI*-ben található nála specifikusabb érvénytelen funkcionális függés. Ha nem találunk ilyet, akkor a funkcionális függőség érvényes **r** felett. Az eljárás hatékonysága javítható megfelelő adatstruktúrák használatával (Savnik & Flach, 1993.) (Savnik & Flach, 1993.).

A funkcionális függetlenségeket alkalmazó módszer hátránya, hogy *nagyon érzékeny az adatbázisban rejlő hibákra*. Minden hibát, ami sért egy funkcionális függőséget, úgy értelmez, hogy az adott függőség érvénytelen az adatokon. Így az adattisztítási feladat megalapozására a módszer ebben a formában nem használható.

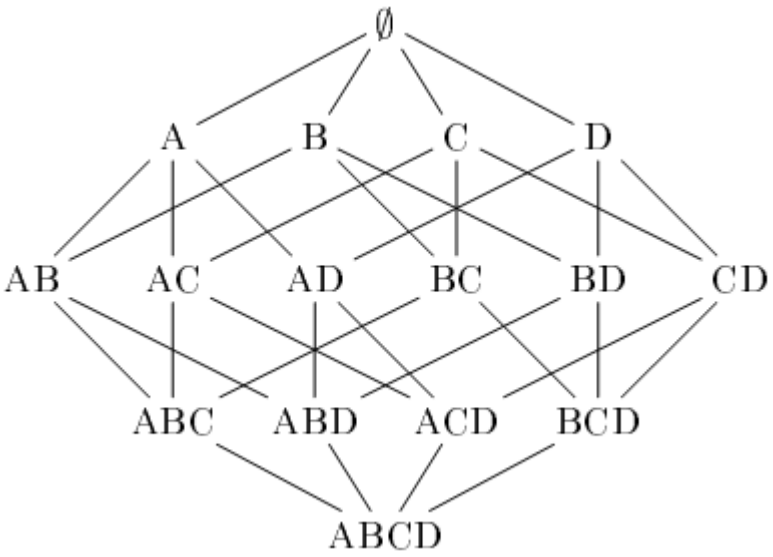
3.4.3 Keresés

Akármilyen is az algoritmus (alulról-felfelé vagy felülről lefelé haladó) mindenképpen alkalmaznia kell valamilyen keresési eljárást. Minden algoritmus előállít ugyanis jelölteket (a funkcionális függőségek valamelyik oldalára), amelyeket valamilyen sorrendben fel kell dolgoznia. A feldolgozást nevezzük keresésnek, vagy bejárásnak is, mivel a jelöltek előállítása rendszerint szisztematikusan történik. A jelölteket egy gráfban ábrázolhatjuk.

Egy ilyen gráfot mutat az 1. ábra. A gráf egyes szintjeinek csomópontjai rendre eggyel több attribútumot tartalmaznak, mint az előző. Látható tehát, hogy a keresési tér igen nagy lehet, így fontos valamilyen metsző eljárás alkalmazása.

A használt kereső módszerek alapvetően kétfélék lehetnek: szélességi és mélységi keresők.

A szélességi (vagy szintenkénti) keresők a gráf elemeit szintenként dolgozzák fel. Azaz



1. ábra A keresési gráf (Mannila & Toivonen, 1997.)

először ellenőrzik az egy elemű attribútum halmazokat, aztán a kételeműeket és így tovább. A szintenkénti keresőkről (Mannila & Toivonen, 1997.) átfogó elemzést ad. A szerzők általános nézőpontból közelítenek az algoritmushoz és összefoglalják, hogy a tudásfeltárás (KDD) területei közül mely részterületeken alkalmazzák. Találkozhatunk vele a funkcionális függőségek feltárása mellett az asszociációs szabályok bányászatánál is.

A keresők másik csoportja a mélységi keresők. Ezek ahelyett, hogy szintenként dolgoznák fel a gráfot, végighaladnak egy adott szinttől az utolsóig, és csak ezután dolgozzák fel az azonos szint következő elemét.

A szélességi keresők valamivel több optimalizációra adnak lehetőséget, mivel, ahogy látni fogjuk, a metszési eljárásokban felhasználhatjuk az előző szinten előállított jelölteket. Ha viszont mélységi keresővel dolgozunk, akkor egy elem vizsgálatánál még nem állnak rendelkezésünkre az előző szintről megfelelő optimalizációk, így ezt nem használhatjuk föl a metszéshez.

3.4.3.1 Metsző eljárások

Mivel a bejárandó gráf mérete magas attribútum szám mellett igen nagy lehet, célszerű az összes jelölt helyett csak egy részhalmazukkal foglalkoznunk. Azt a folyamatot, amikor kiválasztjuk ezt a részhalmazt, metszésnek nevezzük. A metszés eredménye a feldolgozandó jelöltek számának jelentős csökkenése, amivel a keresési idő is rövidül.

A metszés többféle lehet. Jelentheti egyrészt a funkcionális függőségek bal oldalának jelöltjei közötti válogatást, illetve a jobb oldali jelöltek szűrését.

A metszéshez a funkcionális függőségek tulajdonságait használhatjuk. (Hong, Hamilton, & Butz, 2002.) részletesen bemutatja, hogy melyek ezek a tulajdonságok. A szerzők erőteljesen támaszkodnak az attribútum halmaz lezártjának fogalmára.

Legyen **F** egy funkcionális függőség halmaz, **X** pedig egy attribútum halmaz **r** felett. **X**-nek **F**-re vonatkozó lezártját a következőképpen definiálhatjuk:

$$X^{+} = \{Y \in R |\ \text{X-ből levezethető Y az Armstrong axiómák segítségével}\}$$

X nem triviális lezártja **F**-re nézve az $X^{+'} = X^{+} \setminus \{X\}$ halmaz.

A metszéshez a következő tételeket használhatjuk fel.

2. **Tétel** *Legyen X és Y két attribútum halmaz és* $Z = X \cap Y$. *Ha* $Y \setminus Z \subseteq X^{+'}$ *és* $X \setminus Z \subseteq Y^{+'}$, *akkor* $X \equiv Y$.

A tétel haszna számunkra az, hogy ha egy ilyen ekvivalenciára bukkanunk, akkor az egyik halmazt elvethetjük, elég csak az egyiket tárolnunk a memóriában, és az Y jelölttel tovább nem kell foglalkoznunk. Ráadásul a következő tulajdonság miatt Y minden szuperhalmazát is figyelmen kívül hagyhatjuk.

Az ekvivalencia megállapításához az attribútum halmazok kiszámított lezártjait használhatjuk, így ezt a tulajdonságot csak akkor érdemes kihasználni, ha a lezártakat egyébként is kiszámítjuk.

3. **Lemma** *Legyen **X** és **Y** két attribútum halmaz és legyen XY^+' az XY attribútum halmaz nem triviális lezártja. Ekkor X^+'$\cup Y^+$' $\subseteq XY^+$'.*

A lemmából következik, hogy ha egy **X → Y** funkcionális függőség érvényes, akkor **XZ → Y** is érvényes tetszőleges **Z** attribútum halmazra. Azaz **X** minden szuperhalmazát elhagyhatjuk az **Y**-hoz tartozó bal oldali jelöltek közül.
Ezen kívül a tulajdonságból adódik az is, hogy tetszőleges **X** és **Y** attribútumokra az **XY → X^+' ∪ Y^+'** függőség érvényességét nem kell vizsgálnunk, mivel az mindig érvényes.

4. **Lemma** *Legyen **R** egy relációs séma és **X** $\subseteq$ **R**. Ha **X** $\cup$ X^+' = **R**, akkor **X** kulcs.*

Mivel a kulcsokra igaz, hogy funkcionálisan függ tőlük a séma összes többi attribútuma, így tetszőleges funkcionális függőség, amelynek a bal oldalán kulcs áll, érvényes. Ez azt jelenti, hogy egy kulcs egyetlen szuperhalmazát sem kell jelöltként számba vennünk, mivel azok is érvényes bal oldalak lesznek.
A felsorolt metszési szabályokkal jelentős hatékonyságnövekedést lehet elérni.

3.4.4 Felhasznált adatok mennyisége

Az egyes algoritmusokat csoportosíthatjuk aszerint is, hogy az adatbázisnak mekkora részét használják fel. A módszerek többsége az adatbázis teljes tartalma alapján próbálja feltárni a funkcionális függőségeket a lehető leghatékonyabban. Léteznek azonban olyan algoritmusok, amelyek csak az adatok egy reprezentatív részét, egy mintát használnak fel.
Vannak olyan mintavételező algoritmusok, amelyek a mintát csak kiindulási alapként alkalmazzák: segítségével állítanak elő funkcionális függőségeket, de ezeket az adatbázis teljes tartalmát felhasználva vizsgálják meg és döntik el, hogy érvényesek-e. Ez inkább egy kombinált módszernek tekinthető. Hatékonysága jobb, mintha a teljes adatbázis alapján derítene fel függőségeket és a megtalált függőségek biztosan érvényesek. Kérdés azonban, hogy minden függőség megtalálható-e így. A válasz valószínűleg nemleges.

3.4.5 Függőségek pontossága

Az algoritmusokat csoportosíthatjuk aszerint is, hogy alkalmasak-e közelítő funkcionális függőségek feltárására, vagy csak teljesen érvényes funkcionális függőségeket találhatunk meg vele.

4 Adattisztítás funkcionális függőségekkel

Az előző fejezetekben áttekintettem, hogyan lehet a rejtett funkcionális függőségeket az adatbázis aktuális tartalma alapján felderíteni. Szót ejtettem arról is, milyen hibák léphetnek fel az adatokban. Ebben a fejezetben azt vizsgálom meg, hogyan lehet a hibákat a feltárt funkcionális függőségek segítségével kijavítani.

4.1 Funkcionális függőségek kikényszerítésének módszerei

Az alábbiakban több módszert mutatok arra, hogyan lehet a feltárt funkcionális függőségek alapján az adatbázis tartalmát javítani. Előtte azonban fontos bevezetnünk néhány fogalmat.

Ha adott egy **R** séma és egy **r** reláció **R** fölött, azt mondjuk, hogy **r** konzisztens funkcionális függőségek egy **F** halmazára nézve, ha **r**-en érvényes az összes **F**-beli függőség. Ha ez nem teljesül, akkor **r** inkonzisztens. Az utóbbi esetben lehet szükség az adatok tisztítására. A tisztítást a reláció javításának fogalmával írhatjuk le formálisan.

Jelöljük $\Sigma(\mathbf{r})$-rel azoknak a funkcionális függőségeknek a halmazát, amelyek érvényesek **r**-ben. Legyen **F** funkcionális függőségek egy halmaza, **r** és **r'** relációk ugyanazon **R** séma felett. Ekkor a javítás fogalmát kétféleképpen definiálhatjuk:

- *Javítás törléssel:* Azt mondjuk, hogy **r'** egy javítása **r**-nek **F**-re nézve, ha $\Sigma(\mathbf{r'})$ maximális részhalmaza $\Sigma(\mathbf{r})$-nek és **r'**-ben érvényes minden $\mathbf{X} \rightarrow \mathbf{Y} \in \mathbf{F}$.
- *Javítás sorok beszúrásával:* Azt mondjuk, hogy **r'** egy javítása **r**-nek **F**-re nézve, ha $\Sigma(\mathbf{r'})$ minimális szuperhalmaza $\Sigma(\mathbf{r})$-nek és **r'**-ben érvényes minden $\mathbf{X} \rightarrow \mathbf{Y} \in \mathbf{F}$.

Ha a fenti definíciókhoz hozzávesszük azt is, hogy a javításban az eredeti sorok módosítva szerepelhetnek, akkor sejthető, hogy az adattisztítás nem triviális feladat.

Ahogy látható, az optimális javítás minden esetben a lehető legkisebb mértékben tér el az eredeti relációtól. Az optimális javítás megtalálása azonban nagyon költséges lehet, így gyakran megelégszünk egy elfogadható javítással.

Fontos megemlítenünk a lekérdezésekre adott konzisztens válaszok problémáját. Egy lekérdezésre adott válasz konzisztens, ha egy reláció minden javításában ugyanazt a választ kapjuk rá. Amikor egy adatbázist tisztítunk, ügyelnünk kell arra, hogy a lekérdezések konzisztenciáját megőrizzük. (Chomicki & Marcinkowski, 2005.) bizonyítja, hogy ez a legtöbb esetben igen nehéz feladat (a co-NP vagy az NP osztályba tartozik, a függőségek számától és a reláció méretétől függően).

Még egy dolgot számításba kell vennünk, amikor az adattisztítási feladat problémáit tekintjük át. Az adatbázisokban a hibás értékek mellett hiányzó értékek is lehetnek (azaz egyes mezők NULL értéket tartalmazhatnak). Ilyenkor azt mondjuk, hogy az adatbázis nem teljes. Ha teljes adatbázissal van dolgunk, akkor javításhoz csak a törlés jöhet szóba (Chomicki & Marcinkowski, 2005.). Ha nem teljes, akkor új sorokat is beszúrhatunk, viszont bonyolítja a helyzetet, hogy a NULL értéknek különböző értelmezései léteznek (Zimanyi & Pirotte, 1997.):

- Nemlétezést jelző NULL értékek: amikor egy attribútumnak nem lehet értéke az adott sorban. Például egy férfi leánykori neve.
- Univerzális NULL értékek: arra utalnak, hogy egy tulajdonság teljesül egy attribútum minden lehetséges értékére. Például egy oktató minden tanszéken tanít valamilyen kurzust.
- Alapértelmezett NULL értékek: azt jelzik, hogy egy adott attribútum az adott sorban valamilyen előre definiált értékkel rendelkezik.
- Információhiányra utaló NULL: amikor nem tudjuk, hogy egy adott attribútumnak az adott sorban lehet-e értéke, és ha igen mi.
- Ismeretlen értékeket helyettesítő NULL.

Az utolsó két esetben az értékek ténylegesen hiányoznak, az első három esetben viszont szándékosan hagyták el őket, így valóban nem teljes adatbázisról csak az utolsó két esetben beszélhetünk.

A későbbiekben látni fogjuk, illetve már ebben a szakaszban is utaltam rá, hogy a felhasznált funkcionális függőségek száma nagymértékben befolyásolja az adattisztítás bonyolultságát. Az előző fejezetben tárgyalt algoritmusok viszont a lehető legtöbb függőség feltárására törekszenek. Szükséges tehát olyan módszerek alkalmazása, amelyekkel egy funkcionális függőség halmazból kiválaszthatjuk azokat a függőségeket, amelyeket ténylegesen fel

akarunk használni. Ez legegyszerűbb esetben lehet az emberi elemzés, bonyolultabb esetben pedig valamilyen automatikus módszer. A (Kaewbuadee, Temtanapat, & Peachavanish, 2006.)-ban leírtak alapján például kiszámíthatunk egy szelektivitási értéket az egyes függőségekre és ez alapján választhatunk közülük.

Végezetül fontos megjegyeznünk az adattisztításnak erről a módszeréről, hogy ha a funkcionális függőségek feltárásánál már tudjuk, hogy a megtalált függőségeket adattisztításra fogjuk használni, akkor „előre gondolkodhatunk", azaz a feltáró algoritmust úgy módosíthatjuk, hogy eltároljon olyan információkat, amelyekkel az adattisztítási feladat leegyszerűsíthető. Például ha az algoritmus megtalálja azokat a sorokat, amelyek sértik a függőségeket, akkor célszerű ezekre a sorokra később „emlékezni".

4.1.1 Törlés

Azoknak az algoritmusoknak, amelyek a tisztítást sorok törlésével végzik, először meg kell találniuk a törlendő sorokat. A funkcionális függőségekre vonatkoztatva ez azt jelenti, hogy át kell vizsgálniuk az adatbázis teljes tartalmát, és azokat a sorokat, amelyek megsértik funkcionális függőségek egy halmazát, meg kell jelölnie törlésre. Ez azonban nem triviális feladat, hiszen ahhoz, hogy eldöntsük, egy sor megsért-e egy **X** → **A** függőséget, tudnunk kell, hogy azokban a sorokban, amelyekben **X** értéke megegyezik az adott soréval, milyen érték szerepel **A**-n.

Minden tisztítási módszernél fontos, hogy az eredeti adatbázistól a lehető legkevésbé eltérő új adatbázis kapjunk. A törlésnél ez azt jelenti, hogy a lehető legkevesebb sort kell törlésre megjelölnünk.

4.1.2 Értékmódosítás

A funkcionális függőségek felhasználhatók úgy is, hogy nem töröljük az azokat sértő sorokat, hanem módosítjuk őket. Megkeressük azokat a sorokat, amelyekben egy adott funkcionális függőség nem teljesül, megvizsgáljuk, hogy mely attribútumokon kell az értékeket megváltoztatnunk, hogy teljesüljön és végrehajtjuk a módosítást.

Az ötlet egyszerűnek tűnik, megvalósítása mégis bonyolult. Az adatok és a funkcionális függőségek magas száma mellett az értékmódosítás NP-teljes probléma (Bohannon, Fan, Flaster, & Rastogi, 2005.).

Az értékmódosítással történő adattisztítást több tényező is megnehezíti. Egyrészt el kell döntenünk, hogy mely sorok sértik a függőségeket. Másrészt meg kell határoznunk, hogy milyen értékre kell módosítanunk a régit. Ez azt jelenti, hogy ha adott egy **X** → **Y**

funkcionális függőség és egy olyan **t** sor, amelyen ez a függőség nem érvényes, akkor keresnünk kell egy olyan **u** sor, amelyre nem sérti a függőséget és vagy **t[X]** = **u[X]**, vagy **t[Y]** = **u[Y]**. Azaz el kell döntenünk, hogy a **t** sorban az **X** attribútum halmaz értékeit tekintjük helyesnek, vagy az **Y** attribútum halmazéit. Általában az előbbit.

A harmadik, talán legnehezebb probléma, hogy a módosításokat úgy kell végrehajtanunk, hogy az eredeti adatbázistól a lehető legkisebb mértékben eltérő javítást kapjuk meg. Ehhez elengedhetetlen a közelség vagy a változtatások költségének valamiféle mérése. (Bohannon, Fan, Flaster, & Rastogi, 2005.) beszámol egy költségalapú eljárásról és a költségekre épített heurisztikáról. A költségek kiszámítása sajnos már önmagában is bonyolult lehet (gondoljunk csak a sztringek hasonlóságának mérésére), amit tovább tetéznek az esetleges NULL értékek.

Más megközelítések, például (Bertossi, Bravo, Franconi, & Lopatenko, 2005.) leszűkítik az értékmódosítást csak numerikus attribútumokra. (Bertossi, Bravo, Franconi, & Lopatenko, 2005.) emellett tesz egy olyan megkötést is, hogy kulcsokat módosítani nem lehet. A javítások távolságának becslésére a legkisebb négyzetek módszerét használják. A szerzők bizonyítják, hogy annak eldöntése, hogy létezik-e optimális javítás, nehéz feladat.

Az értékmódosítás egyik legfőbb problémája a nehézsége. Ez kiküszöbölhető úgy, hogy az optimális javítást csak közelítjük. A másik probléma az, hogy egy attribútum érték módosításával olyan sort kaphatunk, ami egy újabb funkcionális függőséget sért, így ennél a módszernél körültekintően kell eljárnunk.

4.1.3 Beszúrás

Mint korábban említettem, a beszúrással történő javítást csak akkor szabad alkalmazni, ha az adatbázis nem teljes. A beszúrással történő javítás azt jelenti, hogy a fennálló függőségek segítségével, az adatbázis aktuális tartalma alapján eldöntjük, hogy mely sorok hiányoznak az adatbázisból. Ez azonban csak sorgeneráló függőségek esetén működik, amelyek arra fogalmaznak meg összefüggéseket, hogy ha bizonyos tulajdonságú sorok jelen vannak az adatbázisban, akkor milyen más soroknak kell még jelen lenniük. Sorgeneráló függőség például a többértékű függőség is.

5 Multidimenzionális adatbázisok

A kidolgozott algoritmus ismertetése előtt fontosnak tartom a multidimenzionális adatbázisok bemutatását, mivel diplomamunkám célja az ilyen, feltehetőleg nagyméretű adatbázisok adatainak hatékony tisztítása.

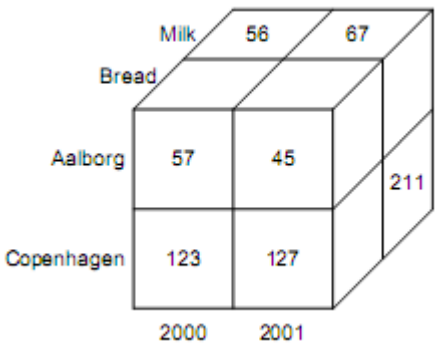
A multidimenzionális adatbázisokban az adatokat több dimenziós struktúrákban tárolhatjuk. Ezeket szokás kockáknak nevezni (háromnál több dimenzió esetén is). A dimenziók egyes tengelyein az adott dimenzió lehetséges értékei szerepelnek. Így egy adatkocka minden elemét azonosíthatjuk a tengelyeken hozzá tartozó értékekkel.

A *dimenziók* olyan fogalmak, amelyek szerint az adatokat csoportosítani szeretnénk (Pedersen, Pedersen, & Jensen, 2005.). Dimenzió lehet például egy áruház esetén az eladás helye, az eladás ideje és az eladott termék. Minden dimenzióhoz tartozik egy domén, ami a dimenzió lehetséges értékeit tartalmazza.

A dimenziók gyakran hierarchikus szervezésűek. Ez azt jelenti, hogy a dimenziókat különböző aggregációs szinten tudjuk kezelni. Az eladások számát például tekinthetjük egyszer egy adott évre összegezve, máskor pedig hónaponként. Ráadásul egyszerre több hierarchiát is értelmezhetünk. Az idő dimenzió két lehetséges hierarchiája például az év – hét – nap és az év – hónap – nap. Mivel a hetek vége nem esik mindig a hónapok határára, így ezeket nem tudjuk egy hierarchiában kezelni.

A cellák a kocka dimenzió értékek által meghatározott elemei. A nem üres cellákat *tényeknek* nevezzük. A tény a dimenziók bizonyos értékeihez tartozó információk. A fenti példában tény lehet például a Koppenhágában 2001-ben eladott kenyerek száma. A tényeket gyakran valamilyen aggregáló függvénnyel (például összegzés) számítják ki.

Hogy milyen értékek szerepelhetnek a cellákban, azt a *mértékek* határozzák meg. Tárolhatjuk például azt, hogy hány darabot adtunk el az adott helyen az adott időben egy termékből, vagy tárolhatjuk azt, hogy mekkora bevételre tettünk szert az eladásokból. A mértékek meghatározzák azt is, hogy a tények kiszámítása milyen aggregáló függvénnyel történik. Aggregációra akkor van szükség, amikor a dimenzió hierarchiában alacsonyabb szinten álló dimenzió helyett egy magasabb szintűt akarunk használni.



2. ábra Egy háromdimenziós adatkocka

A multidimenzionális adatbázisokat gyakran nem valódi adatkockákkal implementálják. Ehelyett a fenti fogalmakat leképező relációs sémákat alakítanak ki. A legegyszerűbb ezek közül a csillag séma. A csillag séma egy *ténytáblából* és több *dimenziótáblából* áll. A ténytáblában tároljuk a tényeket, azaz a különböző mértékekhez tartozó értékeket. A ténytáblához külső kulccsal kapcsolhatók a dimenziótáblák. Minden dimenzióhoz készítenünk kell egy ilyen táblát, amely az adott dimenzió értékeit tartalmazza.

A csillag sémánál bonyolultabbak is elképzelhetők. Ha például normalizáljuk a dimenziótáblákat, akkor hópehely sémát kapunk (Kimball & Ross, 2002.).

A multidimenzionális adatbázisok tipikusan több forrás adataiból épülnek föl. Emiatt kulcsfontosságú, hogy a források adatai megfelelően tiszták legyenek, hiszen csak így kaphatunk „hiteles" adattárházat. Ezen kívül az egyesített adatokon is szükség lehet a minőség javítására.

A multidimenzionális adatbázisok megkülönböztető tulajdonsága, hogy rengeteg adatot tartalmaznak. Ez nehézzé teszi az adattisztítási feladatot, különösen akkor, ha azt az adatokból kinyert szabályszerűségek (például funkcionális függőségek) alapján kívánjuk elvégezni. A továbbiakban egy olyan algoritmust mutatok be, amely képes hatékonyan kezelni a nagyméretű adathalmazokat és a funkcionális függőségek felderítése után alkalmas az abban rejlő hibák kijavítására.

6 A funkcionális függőségek felderítésének ötvözése az adattisztítási feladattal

A fentiekben bemutattam, hogy a diplomamunkám készítése során milyen ismereteket kellett áttekintenem, illetve milyen megközelítéseket kellett megvizsgálnom. Célom az volt, hogy ezen információk felhasználásával megalkossak egy olyan módszert, ami a korábbiaknál jobb hatékonysággal működik. Az első perctől szem előtt tartottam, ahogy erre korábban is utaltam, hogy a funkcionális függőségekkel történő adattisztítás egyik gyenge hatékonysága abból adódik, hogy a függőségek feltárását és a tisztítási folyamatot külön választjuk. Ezzel lemondunk rengeteg információról, amit a függőségek feltárása során „melléktermékként" megszerzünk és az adatok minőségének javításában felhasználhatnánk. Ha a két folyamatot együtt kezeljük, akkor ezek az információk felhasználhatóvá válnak és segítségükkel javíthatunk a hatékonyságon.

Célom tehát az volt, hogy a fenti funkcionális függőség feltáró algoritmusok valamelyikét képessé tegyem arra, hogy az adattisztítást minél nagyobb mértékben támogassa. Sajnos nem minden fent ismertetett algoritmus alkalmas erre az átalakításra, így sokszor falba ütköztem. Általában az jelentette a problémát, hogy a legtöbb algoritmus nem termel ki menet közben olyan információkat, amelyeket fel lehetne használni az adattisztításban, vagy ha vannak is ilyen információk, tárolásuk a hatékonyság rovására menne.

Szintén problémát jelentett az, hogy az algoritmusok nagy része igen érzékeny az adatokban jelentkező hibákra (így nem alkalmas azok kijavítására), ha pedig kevésbé érzékennyé akarnánk tenni, azzal ismét a hatékonyságot rontanánk. Jó példa erre a 3.4.2.3 fejezetben bemutatott algoritmus, amely a funkcionális függetlenségek megtalálására épít. Ez a módszer feltételezi, hogy az adatbázis mentes mindenféle hibától. Ha ezt a könnyítést megszüntetjük, akkor úgy kell módosítanunk az algoritmust, hogy csak akkor jelöljön meg egy funkcionális függetlenséget érvényesnek, ha az legalább bizonyos számú sorpárban érvényes. Ebben az esetben már nem elég csak a legspecifikusabb függetlenségeket tárolni, az általánosabbakat is meg kell jegyeznünk, hiszen előfordulhat, hogy egy általánosabb függetlenség érvényes a megfelelő számú sorban, míg egy nála specifikusabb nem. Ezzel elvesztettük az algoritmus szinte egyetlen optimalizálási lehetőségét.

Az algoritmusok vizsgálata után arra jutottam, hogy a TANE algoritmus alkalmas arra, hogy képessé tegyük az adattisztítási feladat támogatására, így a fejezet hátralévő részében csak ezzel foglalkozom. A tisztítás lehetséges módjai közül a törlést választottam.

A következőkben összefoglalom, milyen szempontok szerint módosítottam az algoritmust, a fejezet végén pedig írok az implementációról és az elért eredményekről.

6.1 A TANE algoritmus módosítása

A TANE algoritmus partíciókat használ és alkalmas funkcionális és közelítő függőségek viszonylag hatékony feltárására (lásd 3.4.2.1), viszont sok szempontból ez az algoritmus is lehetne jobb. Vizsgálataim során a következő tulajdonságain próbáltam javítani, illetve módosítani:

- Az *algoritmus hatékonysága* nagyon nagy adathalmazokon igen rossz. Ezen sajnos az algoritmus alapvető működési elve miatt nem lehet javítani, viszont a memória felhasználás csökkenthető. Ehhez a vizsgálandó adathalmazt részhalmazokra osztom és a partíciókat minden lépésben csak egy részhalmazon készítem el. A teljes partíció

inkrementálisan felépíthető a részhalmazokra kiszámított partíciók ekvivalencia-osztályainak megfelelő egyesítésével.

- A hatékonyság javításának másik módszereként számba vettem a *minta* használatának lehetőségét is. Ebben a megközelítésben a partíciókat nem a teljes adathalmazon, csak annak egy reprezentatív részén készítjük el. Ezen a részhalmazon feltárjuk a függőségeket, majd a teljes adathalmazon verifikáljuk azokat. A verifikáció során új függőségeket már nem keresünk, csak meglévőket vethetünk el. Ha feltételezzük, hogy legfeljebb azok a függőségek állnak a teljes adathalmazon, amik a mintán, akkor ezzel a „veszteséggel” együtt lehet élni.
- A TANE algoritmus alapértelmezetten alkalmas *közelítő funkcionális függőségek* feltárására. Mint látni fogjuk, a közelítő funkcionális függőségek erősen köthetők az adatbázis hibáihoz. Viszont ahhoz, hogy ezeket a hibákat meg tudjuk fogni és ki tudjuk javítani, nem elegendő közelítő függőségekről beszélnünk, hiszen ezzel csak arról nyerünk információkat, hogy léteznek hibás sorok. Ha magukat a hibás sorokat meg akarjuk találni, át kell fogalmaznunk az algoritmus bizonyos szakaszait.
- Végül megvizsgáltam, hogy a generált jelöltek feldolgozásának sorrendjére létezik-e valamilyen *heurisztika*, amellyel a hatékonyság javítható azáltal, hogy csökken a később feldolgozandó jelöltek száma.

A továbbiakban az egyes szempontokról írok bővebben különös tekintettel arra, ami egyben a fő témám is, hogy miként lehet a feltárási és a tisztítási feladatot ötvözni.

6.1.1 A TANE algoritmus és a hibás adatok

A TANE algoritmus érzékeny a hibás adatokra abban az értelemben, hogy ha egy $\mathbf{X \rightarrow A}$ funkcionális függőség jobb oldala sérül akár egy sorban is, akkor a $\boldsymbol{\pi}_{\mathbf{X \cup \{A\}}}$ partíció „elromlik”, azaz $|\boldsymbol{\pi}_{\mathbf{X}}| \neq |\boldsymbol{\pi}_{\mathbf{X \cup \{A\}}}|$. (Huhtala, Karkkainen, & Toivonen, 1999.) ugyan definiál egy mértéket (g_3), amivel az algoritmus képes a közelítő függőségek feltárására is, azonban ha a feltárás során információkat szeretnénk találni arról, hogy melyek azok a sorok, amelyek megsértik a függőséget, akkor ennél többre van szükségünk.

Első lépésként be kell vezetnünk egy küszöbértéket.

11. definíció. *Legyen* $\boldsymbol{R}$ *egy relációs séma és* $\boldsymbol{r}$ *egy reláció* $\boldsymbol{R}$ *felett. Egy* $\boldsymbol{X \rightarrow A}$ *funkcionális függőség megfelelő* $\boldsymbol{r}$ *felett, ha* $|\boldsymbol{r} \setminus \{\boldsymbol{t} \in \boldsymbol{r} \mid \boldsymbol{t[A]} \neq \boldsymbol{u[A]}, \boldsymbol{u} \in r \text{ és } \boldsymbol{t[X]} = \boldsymbol{u[X]}\}|/|\boldsymbol{r}| \leq \boldsymbol{\varepsilon}$*, ahol* $\boldsymbol{\varepsilon}$ *tetszőleges és* $0 \leq \boldsymbol{\varepsilon} < 1$*.*

Azaz csak azokat a funkcionális függőségeket fogadjuk el, amelyekre igaz az, hogy az őket megsértő sorok száma megfelelően kicsi a reláció összes sorához képest.

Ha a definícióban említett arány 0, akkor a funkcionális függőség minden sorban érvényes. Ebben az esetben $|\boldsymbol{\pi}_{\mathbf{X}}| = |\boldsymbol{\pi}_{\mathbf{X} \cup \{\mathbf{A}\}}|$. A másik lehetőség, hogy $|\boldsymbol{\pi}_{\mathbf{X}}| < |\boldsymbol{\pi}_{\mathbf{X} \cup \{\mathbf{A}\}}|$ (a partíciók definíciójából következik, hogy $|\boldsymbol{\pi}_{\mathbf{X}}| > |\boldsymbol{\pi}_{\mathbf{X} \cup \{\mathbf{A}\}}|$ soha nem teljesülhet, mivel $\boldsymbol{\pi}_{\mathbf{X} \cup \{\mathbf{A}\}}$ legalább annyi osztály tartalmaz, mint $\boldsymbol{\pi}_{\mathbf{X}}$). Ez utóbbi esetben $\boldsymbol{\pi}_{\mathbf{X} \cup \{\mathbf{A}\}}$ minden ekvivalencia osztálya vagy megegyezik $\boldsymbol{\pi}_{\mathbf{X}}$ valamely ekvivalencia osztályával, vagy részhalmaza annak. Azaz igaz a következő összefüggés:

$$\forall \mathbf{c} \in \boldsymbol{\pi}_{\mathbf{X}} \exists \mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_k \in \boldsymbol{\pi}_{\mathbf{X} \cup \{\mathbf{A}\}}, \text{hogy } \mathbf{c} = \mathbf{c}_1 \cup \mathbf{c}_2 \cup \cdots \mathbf{c}_k, (\mathbf{k} > 0)$$

Ebből adódik, hogy ha az $\mathbf{X} \rightarrow \mathbf{A}$ függőség érvényességét ki akarjuk kényszeríteni, akkor a $|\boldsymbol{\pi}_{\mathbf{X}}| < |\boldsymbol{\pi}_{\mathbf{X} \cup \{\mathbf{A}\}}|$ egyenlőtlenség két oldalát egyenlővé kell tennünk. Ehhez a $\mathbf{c_i}$ részhalmazok közül egy kivételével mindet javítanunk vagy törölnünk kell. Ezt nyilván úgy kell megtennünk, hogy a lehető legkevesebb módosítással járjon, azaz minden esetben a maximális méretű részhalmazt kell meghagynunk. Törekednünk kell ugyanis arra, hogy a módosított adatbázis a lehető legkevésbé térjen el az eredetitől.

A törlés felvet néhány problémát. Először is, ha bizonyos sorokat törlendőnek jelölünk meg, akkor el kell döntenünk, hogy a többi függőség vizsgálata előtt töröljük őket, vagy csak az összes függőség megtalálása után. Az előbbi esetben a későbbi függőség jelöltek vizsgálata során már nem kell figyelembe vennünk a törölt sorokat. Ennek az az előnye, hogy a későbbi felderítés már kisebb adathalmazon folyik, hátránya viszont, hogy a sorok törlése elronthat más függőségeket.

A maximális méretű részhalmazokat megkereső algoritmus bemutatása előtt fontos kitérnünk egy optimalizációs lehetőségre. Tegyük fel, hogy adott egy $\boldsymbol{\pi}_{\mathbf{X}}$ partíció. Ha létezik olyan $\mathbf{c} \in \boldsymbol{\pi}_{\mathbf{X}}$, hogy $|\mathbf{c}| = 1$, akkor a $\mathbf{c}$ ekvivalencia osztályt kivehetjük a partícióból, hiszen semmilyen információhoz nem jutunk felhasználásával. Jelöljük $\boldsymbol{\pi}_{\mathbf{X}}^{*}$-gal azt a partíciót, amit úgy kapunk, hogy $\boldsymbol{\pi}_{\mathbf{X}}$-ből elhagyjuk az egy elemű ekvivalencia osztályokat. A későbbiekben csak ilyen, ún.

kanonikus alakú partíciókkal dolgozunk (Huhtala, Karkkainen, & Toivonen, 1999.). A kanonikus partíciók és a hibás sorok összefüggése kapcsán két dologra kell odafigyelnünk.

Amikor egy $\mathbf{X} \rightarrow \mathbf{A}$ függőség két partícióját, a $\boldsymbol{\pi}_{\mathbf{X}}$ és $\boldsymbol{\pi}_{\mathbf{X} \cup \{A\}}$ partíciókat vizsgáljuk, akkor a hibás sorok miatt $\boldsymbol{\pi}_{\mathbf{X} \cup \{A\}}$-ban előfordulhatnak olyan egyelemű ekvivalencia osztályok, amelyek $\boldsymbol{\pi}_{\mathbf{X}}$-ben nem szerepeltek. Ezekről tudjuk, hogy hibás sorokhoz tartoznak, így ezeket nyilván kell tartanunk a törlendő sorok között.

Másrészt előfordulhat, hogy $\boldsymbol{\pi}_{\mathbf{X}}$ minden sora $\boldsymbol{\pi}_{\mathbf{X} \cup \{A\}}$ egy egyelemű ekvivalencia osztályában szerepel. Ekkor $\boldsymbol{\pi}_{\mathbf{X}}$ minden ekvivalencia osztályából egy tetszőleges sort meg kell hagynunk, a többit viszont fel kell vennünk a törlendő sorok közé. Az első esetet a HIBÁS-SOROK algoritmus 19. sora kezeli, a másodikat az algoritmus 11-15. sora.

Ezen kívül akkor is figyelembe kell vennünk az eredeti partíciókból eltávolított sorokat, amikor azt vizsgáljuk, hogy egy attribútum szuperkulcs-e vagy sem. Nem elég ugyanis azt ellenőriznünk, hogy a kanonikus partícióban pontosan annyi ekvivalencia osztály van-e, mint a sorok száma, hanem hozzá kell vennünk ehhez az eltávolított sorok számát is.

A kanonikus partíciók használata hatékonysági szempontból fontos. Később ugyanis látni fogjuk, hogy a hibás sorokat megkereső algoritmus érzékeny a partíciók méretére. Így ha a partíciók mérete csökken, a hatékonyság javul, amit a tapasztalat is igazol.

A maximális elemszámú halmazok megtalálására használható algoritmust ezek után a 3. ábrán látható pszeudokóddal írhatjuk le.

Az algoritmus használ egy **T** táblázatot. A táblázatban tároljuk $\boldsymbol{\pi}_{\mathbf{X} \cup \{A\}}^{*}$ ekvivalencia osztályainak méretét. Minden ekvivalencia osztályt egy tetszőleges sorával azonosítunk. Ha $\boldsymbol{\pi}_{\mathbf{X}}^{*}$ több sorához is tartozik érték **T**-ben, az azt jelenti, hogy **c**-nek vannak részhalmazai $\boldsymbol{\pi}_{\mathbf{X} \cup \{A\}}^{*}$-ban. Az algoritmus ekkor kiválasztja a részhalmazok közül az egyik legnagyobbat és minden más részhalmazt a törlendő ekvivalencia osztályok **L** listájához ad.

Ha **c**-nek egyetlen részhalmaza sincs $\boldsymbol{\pi}_{\mathbf{X} \cup \{A\}}^{*}$-ban, az azt jelenti, hogy a kanonikus partíció előállításakor ezeket töröltük. Ebben az esetben az algoritmus **c** egy tetszőleges sorát kiveszi a törlendő sorok közül. Végül minden olyan sort hozzáad **L**-hez, amit törölt $\boldsymbol{\pi}_{\mathbf{X} \cup \{A\}}$-ból, amikor kanonikus alakra hozta, de nem törölt $\boldsymbol{\pi}_{\mathbf{X}}$-ből.

Az algoritmus helyessége könnyen belátható. Tegyük fel, hogy $\mathbf{c} \in \boldsymbol{\pi}_{\mathbf{X}}^{*}$, $\mathbf{c_1}, \ldots, \mathbf{c_k} \in \boldsymbol{\pi}_{\mathbf{X} \cup \{A\}}^{*}$ és $\mathbf{c} = \mathbf{c_1} \cup \ldots \cup \mathbf{c_k}$. Legyenek $\mathbf{t_1} \in \mathbf{c_1}, \ldots, \mathbf{t_k} \in \mathbf{c_k}$ a 3. kódsorban kiválasztott tetszőleges sorok. Ekkor $\mathbf{T}[\mathbf{t_i}]$ ($\mathbf{i} = 1, \ldots, \mathbf{k}$) létezik és értéke az **i**-edik ekvivalencia osztály számossága $\boldsymbol{\pi}_{\mathbf{X} \cup \{A\}}^{*}$-ban. Legyen $\mathbf{t} \in \mathbf{c}$.

Ha $|\boldsymbol{\pi}_{\mathbf{X}}^{*}| = |\boldsymbol{\pi}_{\mathbf{X}\cup\{A\}}^{*}|$, akkor $\mathbf{T}[\mathbf{t}]$ definiált és pontosan egy ilyen **t** létezik. Ebben az esetben az algoritmus (helyesen) nem jelöl meg semmit törlésre. Ha több olyan **t** van **c**-ben, amelyre $\mathbf{T}[\mathbf{t}]$ definiált, az azt jelenti, hogy **c**-nek vannak részhalmazai $\boldsymbol{\pi}_{\mathbf{X}\cup\{A\}}^{*}$-ban, azaz a függőség sérül. Ebben az esetben az algoritmus minden olyan ekvivalencia osztályt törlésre jelöl $\boldsymbol{\pi}_{\mathbf{X}\cup\{A\}}^{*}$-ból, amelyben nem szerepel $\mathbf{t_m}$. Mivel pedig $\mathbf{t_m}$-et az algoritmus az egyik maximális elemszámú halmazból választja, így ez a működés megint csak helyes.

Ha nem létezik a fent leírt tulajdonságú **t** sor, az azt jelenti, hogy a kanonikus alakra hozáskor az algoritmus **c** minden sorát eltávolította, mivel $\boldsymbol{\pi}_{\mathbf{X}\cup\{A\}}^{*}$-ban minden részhalmaza egyelemű volt. Ebben az esetben a maximális részhalmaz mérete is 1, azaz mindegy, hogy melyik részhalmazt tartjuk meg, így az algoritmus tetszőlegesen választhat.

A HIBÁS-SOROK algoritmus a funkcionális függőségek érvényességének eldöntésére használható. Az ellenőrizendő függőségeket a TANE algoritmus keresi meg. A módosított változat pszeudokódja a 4. ábrán látható.

Az eredeti algoritmus kimenete a felderített funkcionális függőségek halmaza, míg a módosított algoritmus visszaadja azoknak a soroknak a listáját is, amelyek sértik a függőségeket. Az alábbiakban áttekintem, hogy milyen további módosításokat kell a TANE algoritmus egyes részein végrehajtanunk, hogy az új algoritmus is helyesen működjön.

A kódban $\mathbf{C}^{+}$ a jobb oldali jelöltek optimalizált halmaza. Ezekre a jelöltekre fogja FÜGGŐSÉGEK ellenőrizni, hogy egy adott bal oldallal érvényes funkcionális függőséget alkotnak-e. Az „optimalizált” jelző itt azt jelenti, hogy a jelöltek közül minden olyat elhagyunk, amelyeket már fölösleges ellenőrizni.

A KÖVETKEZŐ-SZINT előállítja az attribútumoknak azon ℓ+1 elemű részhalmazait, amelyeket a szintenkénti keresésnek a következő lépésben fel kell dolgoznia. $\mathbf{L}_{\ell+1}$ minden eleme olyan **Y** attribútum halmaz, amelyre teljesül, hogy tetszőleges $\mathbf{X} \subseteq \mathbf{Y}$, $|\mathbf{X}| = \ell$ esetén $\mathbf{X} \in \mathbf{L}_{\ell}$. Azaz $\mathbf{L}_{\ell+1}$ kiszámításához csak $\mathbf{L}_{\ell}$ elemeit kell felhasználnunk. A KÖVETKEZŐ-SZINT pszeudokódja megtalálható a függelékben.

ALGORITMUS: HIBÁS-SOROK
INPUT: π_X^* ÉS $\pi_{X\cup\{A\}}^*$
OUTPUT: L, A TÖRLENDŐ SOROK LISTÁJA

1. T üres táblázat, L = Ø
2. **for** c ∈ $\pi_{X\cup\{A\}}^*$ **do**
3. t ∈ c' tetszőleges
4. T[t] := | c' |
5. **for** c ∈ π_X^* **do**
6. max := 0
7. **for** t ∈ c **do**
8. **if** T[t] létezik és T[t] > max **then**
9. max = T[t]
10. t_m := t
11. **if** tm = NULL **then**
12. t ∈ c tetszőleges
13. L = L \ {t}
14. **continue**
15. **for** t ∈ c **do**
16. **for** c' ∈ $\pi_{X\cup\{A\}}^*$ **do**
17. **if** t ∈ c' és $t_m \notin c'$ **then**
18. c'-t minden sorát hozzáadjuk L-hez
19. L = L ∪ {t ∈ r| t ∈ $\pi_{X\cup\{A\}} \setminus \pi_{X\cup\{A\}}^*$, t ∉ $\pi_X \setminus \pi_X^*$ }

3. ábra. A törölhető ekvivalencia osztályok megkeresése

ALGORITMUS: TANE
INPUT: R SÉMA
OUTPUT: A MEGTALÁLT FUNKCIONÁLIS FÜGGŐSÉGEK

1. $L_0 := \{Ø\}$
2. $C^+(Ø) := R$
3. $L_1 := \{\{A\} \mid A \in R\}$
4. $\ell := 1$
5. **while** $L_\ell \neq Ø$ **do**
6. FÜGGŐSÉGEK(L_ℓ)
7. METSZÉS(L_ℓ)
8. $L_{\ell+1}$:= KÖVETKEZŐ-SZINT(L_ℓ)
9. $\ell := \ell + 1$

4. ábra. Az eredeti TANE algoritmus

A METSZÉS a korábban tárgyalt metszési lehetőségeket alkalmazza C^+-ra.

Az előző két algoritmusnál érdekesebb a FÜGGŐSÉGEK, hiszen az eredeti változatban csak a függőségeket számítja ki. Ha a törlendő sorokat is meg akarjuk találni vele, akkor az 5. ábrán látható módon kell megváltoztatnunk.

Az érvényesség vizsgálatát az 5. sorban a 11. definíciónak megfelelően végezzük a partíciók és a HIBÁS-SOROK algoritmus felhasználásával: kiszámítjuk a sorok listáját, amelyek sértik az éppen vizsgált függőséget, majd a lista számossága és a reláció számossága alapján döntünk.

Az algoritmus „mellékhatásként” minden $\mathbf{X} \in \mathbf{L_\ell}$-re kiszámítja a $\mathbf{C^+(X)}$ halmazt. Hogy ezt helyesen is teszi, azt (Huhtala, Karkkainen, & Toivonen, 1999.) függeléke bizonyítja. Ezeket a jelölthalmazokat a következő szinteken felhasználja az algoritmus.

$\mathbf{C}^+(\mathbf{X})$-et egy lemma segítségével számíthatjuk ki.

ALGORITMUS: FÜGGŐSÉGEK
INPUT: L_ℓ
OUTPUT: FUNKCIONÁLIS FÜGGŐSÉGEK ÉS AZ AZOKAT SÉRTŐ SOROK LISTÁJA

1. **for** $X \in L_\ell$ **do**
2. $C^+(X) := \bigcap_{A \in X} C^+(X \setminus \{A\})$
3. **for** $X \in L_\ell$ **do**
4. **for** $A \in X \cap C^+(X)$ **do**
5. **if** $X \setminus \{A\} \rightarrow A$ érvényes **then**
6. **output** $X \setminus \{A\} \rightarrow A$
7. $C^+(X) := C^+(X) \setminus \{A\}$
8. **output** hibás sorok listája
9. **if** $X \setminus \{A\} \rightarrow A$ egzakt **then**
10. $C^+(X) := C^+(X) \setminus \{B \mid B \in R \setminus X\}$

5. ábra. A függőségek és a hibás sorok megkeresése

5. **Lemma** *Legyen* $\boldsymbol{B \in X}$ *és legyen* $\boldsymbol{X \setminus \{B\} \rightarrow B}$ *érvényes függőség. Ekkor:*
   - *ha* $\boldsymbol{X \rightarrow A}$ *érvényes, akkor* $\boldsymbol{X \setminus \{B\} \rightarrow A}$ *is érvényes.*
   - *ha* $\boldsymbol{X}$ *szuperkulcs, akkor* $\boldsymbol{X \setminus \{B\}}$ *is szuperkulcs.*

A $\mathbf{C}^+(\mathbf{X})$ optimalizált jelölthalmaz (egy **X** bal oldalhoz tartozó lehetséges jelöltek optimalizált halmaza) három halmaz metszeteként áll elő. A kiindulási alap $\mathbf{C}(\mathbf{X})$, a jelöltek még nem optimalizált halmaza. Ebben minden jelölt benne van, azok is, amelyeket esetleg fölösleges tesztelni. A halmaz az alábbi szabály szerint állítható elő:

$$C(X) = \{A \in X \mid X \setminus \{A\} \rightarrow A \text{ nem érvényes}\} \cup R \setminus X$$

Tehát egy **X** attribútum halmazhoz tartozó jobb oldali jelölt lehet minden olyan **A** attribútum, amely nem függ **X**-nek egyetlen részhalmazától sem. Ezzel a jelölthalmaz minimalitását garantáljuk.

Az 5. lemma alapján azonban még ezt a jelölthalmazt is lehet csökkenteni. A lemma első fele alapján ha $\mathbf{X} \setminus \{\mathbf{B}\} \rightarrow \mathbf{B}$ érvényes valamely $\mathbf{B} \in \mathbf{X}$ esetén és **X** megjelenik egy függőség bal oldalán, akkor **B**-t eltávolíthatjuk **X**-ből, ennek semmilyen hatása nincs a függőség érvényességére. Így elegendő **C**(**X**)-ből csak azokkal a jelöltekkel foglalkoznunk, amelyek az alábbi **C'** (**X**) halmazban is megjelennek:

$$C'(X) = \{A \in R \setminus X \mid X \setminus \{B\} \rightarrow B \text{ nem érvényes egyetlen } B \in X \text{ esetén sem}\} \cup X$$

Hasonlóképpen, ha $\mathbf{Y} \setminus \{\mathbf{B}\} \rightarrow \mathbf{B}$ érvényes valamely $\mathbf{B} \in \mathbf{Y}$ esetén, akkor minden további nélkül eltávolíthatjuk az összes $\mathbf{A} \notin \mathbf{Y}$-t **C**(**X**)-ből, ahol **X** szuperhalmaza **Y**-nak. Így csak azokat a jelölteket kell figyelembe vennünk, amelyek az alább definiált **C'' ** halmaznak is elemei.

$$C'' = \{A \in X \mid X \setminus \{A,B\} \rightarrow B \text{ nem érvényes egyetlen } B \in X \setminus \{A\}\text{-ra sem}\} \cup R \setminus X$$

A $\mathbf{C}^+(\mathbf{X})$ optimalizált jelölthalmaz ezután a következőképpen számítható ki:

$$C^+(X) = C(X) \cap C'(X) \cap C''(X)$$

Ezután kimondhatjuk a következő lemmát.

6. **Lemma** *Legyen* $\boldsymbol{A \in X}$ *és legyen* $\boldsymbol{X \setminus \{A\} \rightarrow A}$ *egy érvényes funkcionális függőség.Az* $\boldsymbol{X \setminus \{A\} \rightarrow A}$ *akkor és csak akkor minimális, ha* $\boldsymbol{A \in C^+(X \setminus \{B\})}$ *minden* $\boldsymbol{B \in X}$*-re.*

A lemma azt biztosítja, hogy ne csak egyszerűen az érvényes függőségeket találjuk meg, hanem a minimalitást is ellenőrizni tudjuk, azaz ne pazaroljuk a futási időt olyan függőségek feltárására, amelyek már a többiből levezethetők. De *milyen hatással vannak a hibás sorok* $\boldsymbol{C^+(X)}$*-re*? A fölösleges jelöltek eltávolítása a FÜGGŐSÉGEK algoritmus 9. és a 10. soraiban történik. A 10. sor tulajdonképpen a **C**(**X**) és a $\mathbf{C}^+(\mathbf{X})$ közötti különbséget implementálja, ezzel a sorral érjük el a jelölthalmaz optimalitását. A 9. sor azt jelenti, hogy az optimális jelölthalmaz csak akkor állítható elő, ha a függőség egzakt, azaz minden sorban érvényes. Az eredeti algoritmusban ilyen feltételre nem volt szükség, hiszen ott feltételeztük a sorok

hibamentességét. A FÜGGŐSÉGEK algoritmus tehát az optimális jelölthalmaz előállításában különbözik az eredetitől.

Fontos megjegyezni, hogy ha HIBÁS-SOROK nem csak megkeresi, hanem azonnal törli is a hibás sorokat, akkor a 9. sorra nincs szükség, az optimalizáció feltétel nélkül elvégezhető. Ez azért lehetséges, mivel ebben az esetben a függőség egzakttá válik.

A TANE algoritmus többi részegységét (a metszést és a partíciók szorzatát) a függelékben mutatom be. Ezek a részek változatlanok maradnak akkor is, ha hibás sorokra is fel kell készülnünk.

6.1.1.1 Az algoritmusok bonyolultsága

A módosított TANE algoritmus hatékonyságát a HIBÁS-SOROK algoritmus futási ideje befolyásolja leginkább, mivel ez legösszetettebb részfeladat. Ebben a szakaszban ezért csak a HIBÁS-SOROK bonyolultságára adok becslést. A komplexitást a ciklus iterációk számával közelítem, az iterációkban végrehajtott lépéseket nagyvonalúan egységnyi idejűnek tekintem. Amikor az ekvivalencia osztályok méretét kell használnom a számításban, mindenhol az átlagos méretet alkalmazom. A bonyolultság becslésében figyelmen kívül hagyom a kanonikus alak nyújtotta optimalizációt.

Az algoritmus két ciklusból áll: a 2. sornál és az 5. sornál kezdődő **for** ciklusokból. Az első ciklusban feldolgozzuk $\boldsymbol{\pi}_{\mathbf{X\cup\{A\}}}{}^{*}$ minden ekvivalencia osztályát. Ez $|\boldsymbol{\pi}_{\mathbf{X\cup\{A\}}}{}^{*}|$ lépést jelent.

A második ciklus ennél bonyolultabb, hiszen abban több beágyazott ciklus is látható. A fő ciklus $\boldsymbol{\pi}_{\mathbf{X}}{}^{*}$ miden ekvivalencia osztályára lefut egyszer, így $|\boldsymbol{\pi}_{\mathbf{X}}{}^{*}|$ iterációval számolhatunk.

A 7. sor **for** ciklusa az adott ekvivalencia osztály minden sorát feldolgozza, ami $|\mathbf{c}|_{\text{átlag}}$ lépést jelent (ahol $|\mathbf{c}|_{\text{átlag}}$ a $\boldsymbol{\pi}_{\mathbf{X}}{}^{*}$ partíció ekvivalencia osztályainak átlagos mérete). A 15. sorban ismét végigmegyünk az aktuális osztály sorain, így itt szintén $|\mathbf{c}|_{\text{átlag}}$ iteráció történik. Végül a 16. sor ciklusa $\boldsymbol{\pi}_{\mathbf{X\cup\{A\}}}{}^{*}$ minden osztályát feldolgozva $|\boldsymbol{\pi}_{\mathbf{X\cup\{A\}}}{}^{*}|$ lépést tesz meg.

Az algoritmus lépéseinek összesített száma tehát a következő:

$$|\pi_{X\cup\{A\}}{}^{*}| + |\pi_{X}{}^{*}|\cdot(|c|_{\text{átlag}} + |c|_{\text{átlag}}\cdot|\pi_{X\cup\{A\}}{}^{*}|)$$

Ebből egyszerű átalakítással kapjuk a következőt:

$$|\pi_{X\cup\{A\}}{}^{*}| + n\cdot(1 + |\pi_{X\cup\{A\}}{}^{*}|)$$

A képletben **n** a reláció sorainak száma.

Ha a kanonikus alakra hozás hatásait is figyelembe vesszük, akkor a legrosszabb eset az, amikor $\pi_{X \cup \{A\}}^{*}$ minden ekvivalencia osztálya 2 elemű, hiszen ekkor a kanonikus partícióból nem távolítunk el sorokat. Ekkor a bonyolultság az alábbi:

$$\frac{n}{2} + n \cdot (1 + \frac{n}{2}) = \frac{3}{2} \cdot n + \frac{n^2}{2}$$

Azaz az algoritmus a legrosszabb esetben $O(n^2)$ iterációt hajt végre. Ez a reláció nagy számossága mellett nagyon sok lépést jelent, viszont fontos megjegyezni, hogy megfelelő implementációval és körültekintő optimalizációval a hatékonyság javítható. Továbbá a partíciók kanonikus alakja is drasztikusan csökkentheti a szükséges iterációk számát.

A többi algoritmus bonyolultsága kevésbé befolyásolja a futási időt, így azok kiszámítására nem térek ki.

6.1.2 A partíciók inkrementális előállítása

Nagyméretű adatbázisokon az algoritmus rosszul teljesíthet, mivel nem elegendő csak megszámolnunk az egyes partíciók méretét, nyilván kell tartanunk az ekvivalencia osztályokban szereplő sorokat is. Emiatt az adatbázison hosszadalmas lekérdezéseket kell lefuttatnunk, ami viszont ellent mond 2. fejezetben felsorolt kritériumok közül annak, hogy a jó adattisztítási módszernek a lehető legkisebb mértékben kell lefoglalnia az adatbázis erőforrásait. Ráadásul a rengeteg adatot nem tudjuk egyszerre a memóriában tartani. Emiatt a partíciókat érdemes inkrementálisan felépíteni. A módszer a következő.

Az adatbázistáblát részekre osztjuk és minden lépésben csak egy részrelációra számítjuk ki a partíciókat, a lekérdezett sorokat a memóriában feldolgozva. A részpartíciókat ezután, szintén a memóriában, egyesíthetjük. Ennek a megoldásnak az az előnye is megvan, hogy jól párhuzamosítható, az egyes részpartíciók kiszámítása külön gépekre bízható.

A partíciók kiszámítását csak az egyelemű attribútum halmazokra kell inkrementálisan végeznünk, a többi partíció ezek szorzataként áll elő. A partíciók előállítását és egyesítését az alábbi ábrákon szereplő algoritmusokkal végezhetjük.

A PARTÍCIÓK minden $\mathbf{A} \in \mathbf{R}$ attribútumhoz használ egy $\mathbf{T_A}$ táblázatot. A $\mathbf{T_A}$ táblázat annyi különböző sort tartalmaz, ahány különböző érték **r**-ben szerepel az **A** attribútumon. A sorokat

ezekkel az értékekkel címkézi meg az algoritmus. Az algoritmus végére $\mathbf{T_A}$ sorai az **A** attribútum ekvivalencia osztályait tartalmazzák, ezekből pedig előállíthatjuk $\boldsymbol{\pi_X}$-et.

A PARTÍCIÓK algoritmust használhatjuk a teljes reláción illetve részrelációkon is. Ha részrelációkon használjuk, akkor az egyes lépések partícióit egyesítenünk kell. Ezt az EGYESÍTÉS algoritmussal végezhetjük el (7. ábra).

Az EGYESÍTÉS algoritmus szintén egy táblázatot használ. Első lépésként végignézzük az első $\boldsymbol{\pi'_A}$ partíciót és a táblázatban eltároljuk, hogy az **i**-edik ekvivalencia osztályában a soroknak milyen értéke van **A**-n. Ehhez az implementációban célszerű valamilyen formában tárolni ezt az információt, hogy elkerüljük a fölösleges adatbázis műveleteket.

ALGORITMUS: PARTÍCIÓK
INPUT: R SÉMA ÉS r RELÁCIÓ R FELETT
OUTPUT: Π_A, MINDEN $A \in R$ ATTRIBÚTUMRA

1. **for** $A \in R$ **do** $\pi_A := \emptyset$
2. **for** $t \in r$ **do**
3. **for** $A \in R$ **do**
4. t-t hozzáadjuk $T_A[t[A]]$-hoz
5. **for** $A \in R$ **do**
6. **for** $i \in \{i \mid T_A[i]$ definiált$\}$ **do**
7. $\pi_A := \pi_A \cup T_A[i]$

6. ábra. A partíciók kiszámítása

ALGORITMUS: EGYESÍTÉS
INPUT: $\Pi'_A = \{C'_1, \ldots, C'_{|\Pi'_A|}\}$ ÉS $\Pi''_A = \{C''_1, \ldots, C''_{|\Pi''_A|}\}$
OUTPUT: Π_A, AMI Π'_A ÉS Π'_A EGYESÍTÉSE

1. **for** i := 1 to $|\pi'_A|$ **do**
2. legyen $t \in c'_i$ tetszőleges
3. T[t[A]] := i
4. $\pi_A := \pi'_A$
5. **for** i := 1 to $|\pi''_A|$ **do**
6. legyen $t \in c_i$ tetszőleges, $c_i \in \pi_A$
7. **if** T[t[A]] definiált **then**
8. $c_i := c_i \cup c''_i$
9. **else**
10. $\pi_A := \pi_A \cup c''_i$

7. ábra. Partíciók egyesítése

Ezután $\boldsymbol{\pi'_A}$-t átmásoljuk az eredménypartícióba, hiszen $\boldsymbol{\pi_A}$ tartalmazni fog minden ekvivalencia osztályt, ami $\boldsymbol{\pi'_A}$-ban szerepel, ezek legfeljebb bővülhetnek, vagy kiegészülhetnek újakkal.

Végül, az 5. sortól kezdődően, megvizsgáljuk $\boldsymbol{\pi''_A}$ ekvivalencia osztályait. Ha van olyan $\mathbf{c'_i}$ és $\mathbf{c''_j}$, hogy a két ekvivalencia osztály sorai ugyanazt az értéket tartalmazzák **A**-n, akkor a két ekvivalencia osztály egyesítjük, különben c''_j-t új ekvivalencia osztályként hozzáadjuk az eredménypartícióhoz.

Tegyük fel, hogy az **r** relációt **m** darab részre osztjuk. Jelöljük az **i**-edik részrelációt $\mathbf{r_i}$-vel (**i** = 1, …, **m**). Ekkor a PARTÍCIÓK algoritmust **m**-szer kell végrehajtanunk: minden $\mathbf{r_i}$-re egyszer. PARTÍCIÓK egyszeri végrehajtása $|\mathbf{r_i}|\cdot\mathbf{a}$-val arányos időt igényel, ahol ***a*** az attribútumok száma. Azaz a teljes reláció partícióinak részrelációnként történő kiszámítása $\boldsymbol{O(|r|\cdot a)}$ bonyolultságú.

Az EGYESÍTÉS algoritmust $\boldsymbol{(m-1)}$-szer kell végrehajtanunk minden attribútumra. Minden végrehajtás $|\boldsymbol{\pi'_A}|\cdot|\boldsymbol{\pi''_A}|$ lépést jelent. $\boldsymbol{\pi'_A}$–nak minden lépésben az előző egyesítés eredményét

használjuk, így ennek a paraméternek a mérete folyamatosan növekszik. Az algoritmus végrehajtási ideje tehát felülről becsülhető a $|\pi_A| \cdot h \cdot (m-1) \cdot a$ kifejezéssel, ahol $|\pi_A|$ a végleges partíció **A**-ra (tehát az utolsó egyesítés eredménye), **A** az az attribútum, amelyre a legnagyobb partíciót kaptuk eredményül, ***h*** pedig a részrelációkon számított partíciók átlagos mérete. Az EGYESÍTÉS algoritmus hatékonysága az implementációban a megfelelő adatstruktúrák alkalmazásával jelentősen javítható.

6.1.3 Minták alkalmazása

Ha a nagyméretű adatbázisok teljes átvizsgálása helyett azoknak csak egy részét használjuk fel a funkcionális függőségek feltárásában, azzal sokat javíthatunk a hatékonyságon. Azt vizsgáltam meg, létezik-e olyan módszer, amivel eldönthetjük, hogy az adatbázis mekkora részének átvizsgálása szükséges.

(Toivonen, 1996.) leírt egy olyan algoritmust asszociációs szabályok feltárására, amelynél a felhasználó megmondhatja, hogy mekkora mértékű tévedést képes elviselni és az algoritmus ez alapján számítja ki a mintaméretet.

A mintaméret kiszámításánál két dolgot kell figyelembe vennünk:

- itt a pontosság különösen fontos tényező, hiszen a kinyert információkat adattisztításra fogjuk használni.
- míg Toivonen algoritmusa általában csak a minta sorait dolgozza fel, addig itt a törlés miatt mindenképp szükséges egy második menet is, amelyben végig megyünk a teljes adatbázison; ezt a második menetet viszont felhasználhatjuk információink pontosítására.

A fentieket figyelembe véve próbáltam egy egyszerű összefüggést keresni a mintaméretre. A képletet igyekeztem úgy megválasztani, hogy az csak kis mértékben függjön az adatbázis méretétől és minden esetben a lehető legkisebb mintaméretet eredményezze.

Végül a (Kivinen & Mannila, 1993.) által leírt összefüggést mellett döntöttem. Ez a következő képletet használja a mintaméret kiszámításához (ha a függőség megítélésében a g_3 hibát vesszük alapul):

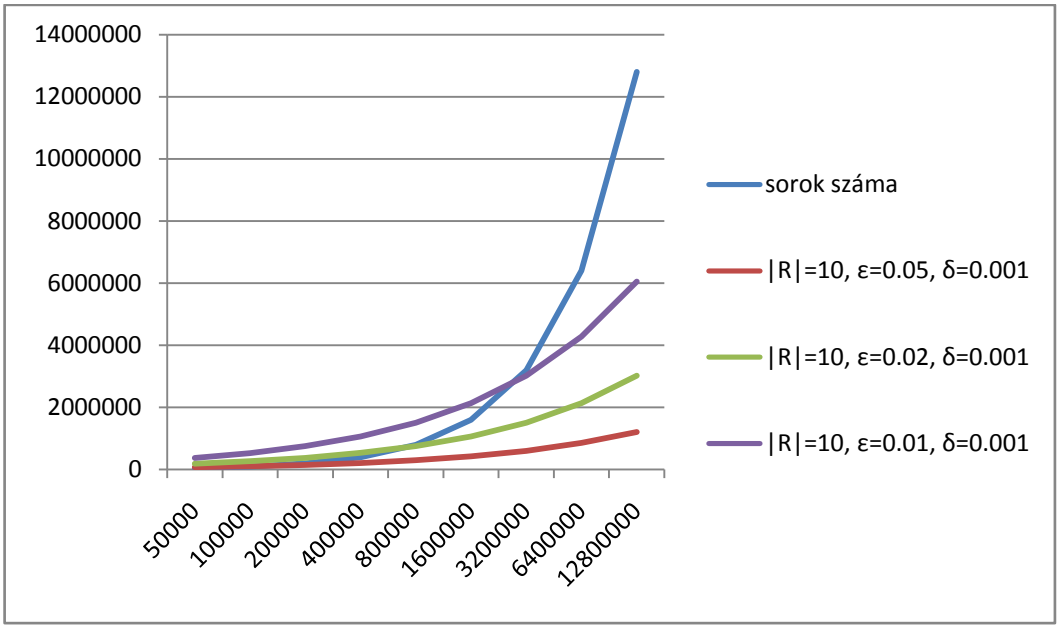
$$m = O\left(\left(\frac{|r|^{1/2}}{\varepsilon}\right)\left(\mathrm{n} + \ln\left(\frac{1}{\delta}\right)\right)\right)O($$

A képlettel azok a hibák kezelhetők, amelyekben egy függőségre a hibaérték a mintán kisebb, mint **ε**, a teljes reláción azonban nem érvényes. A hiba valószínűségét szeretnénk egy előre megadott **δ** érték alatt tartani. Ez a két paraméter és a reláció mérete befolyásolja tehát, hogy mekkora mintaméreten érdemes végrehajtanunk a vizsgálatot (**m**).

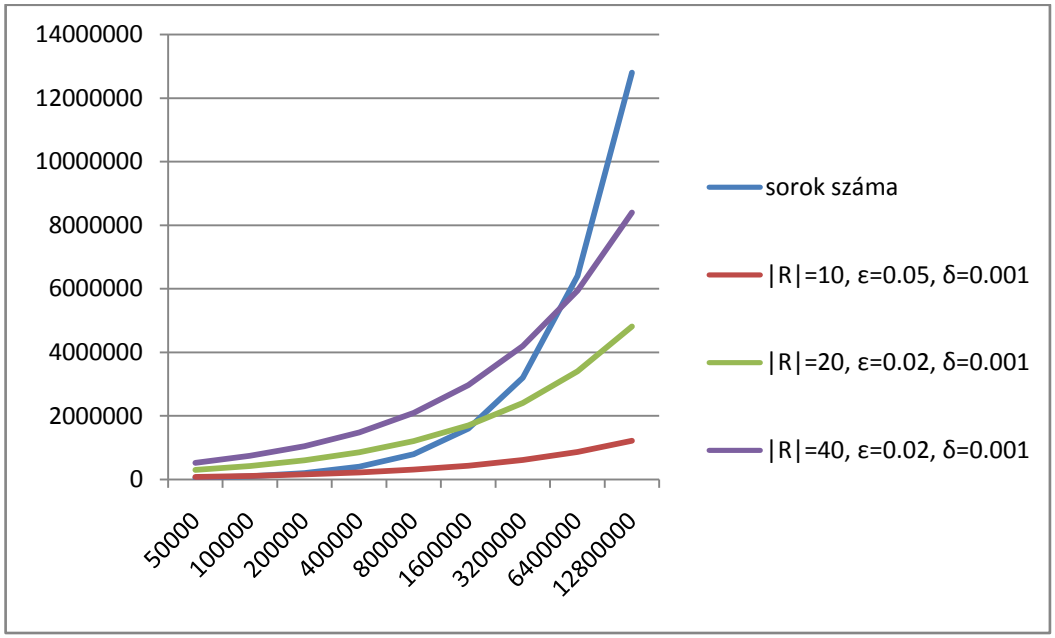
A képlettel kezelhető hibák azért fontosak számunkra, mert annak a valószínűségétől függ a minimális mintaméret, hogy elfogadunk egy olyan függőséget, ami a teljes mintán nem érvényes. Ez különösen veszélyes akkor, ha a függőségek alapján sorokat törlünk, hiszen egy érvénytelen függőség alapján hibás döntés miatt veszíthetünk el adatokat. Az implementációkban tehát úgy használhatjuk a mintákból kinyert információkat, hogy egy második körben végigmegyünk a teljes adatbázison és megkeressük a törlendő sorokat. Eközben azonban a minta alapján kinyert információinkat frissítjük: kiegészítjük a partíciókat és ez alapján érvénytelennek nyilváníthatunk korábban érvényesnek hitt függőségeket. Fontos, hogy a második körben új függőségeket nem keresünk, legfeljebb korábbiaktól szabadulunk meg.

A 7. és 8. ábrákon látható, hogyan változik a szükséges mintaméret nagysága **ε** illetve az attribútum szám növekedésével. Az ábrákból kiderül, hogy az attribútumok száma erősebben befolyásolja a mintaméretet, mint **ε**. Az is jól látszik, hogy bizonyos esetekben nem éri meg mintát alkalmazni, viszont nagy relációk esetén a mintaméret jelentősen kisebb lehet a reláció számosságánál.

A fent leírt képlet dinamikus mintakezelést tesz lehetővé, mivel arra való tekintettel készült, hogy milyen célra fogják használni. Tapasztalatok mutatják, hogy a dinamikus mintakezelés mindig jobb, mint a statikus (John & Langley, 1996.).



7. ábra. A mintaméret növekedése ε csökkenésével



8. ábra. A mintaméret növekedése az attribútumszám növekedésével

6.2 Implementáció és a tesztkörnyezet

Az algoritmusokat Java 1.6 programozási nyelven implementáltam. Az adatbázis eléréséhez JDBC-t használtam. A futtatási környezet JRE 1.6 volt. Ebben a verzióban, a HotSpot technológiának köszönhetően a hatékonyság sok esetben vetekszik a natív kódú alkalmazások hatékonyságával, ezért döntöttem a Java használata mellett.

Az implementáció négy osztályból áll:

- `EquivalenceClass`: Az ekvivalencia osztályok absztrakt reprezentációja. Minden olyan műveletet tartalmaz, amire az ekvivalencia osztályok kezelésénél szükség van.
- `Partition`: A partíciók absztrakt reprezentációja. Tartalmazza `EquivalenceClass` objektumok egy listáját. Ebben az osztályban implementáltam a partíciókhoz köthető műveleteket: két partíció szorzása; két, egyazon attribútumhoz tartozó partíció egyesítése; a törlendő sorok megkeresése stb.
- `Partitioner`: Az adatbázis sorainak feldolgozásáért, a kezdeti partíciók kiszámításáért felelős.
- `Cleaner`: A törlendő sorok kiszámítását végzi. Ez az alkalmazás lelke, itt implementáltam a módosított TANE algoritmust. Az osztály metódusai végzik a metszést, a jelöltek előállítását, a szintek kiszámítását, a generált függőségek ellenőrzését stb.

Az algoritmus megvalósításához sok helyen kollekciókat kellett használnom. Az első implementációban ez a memória elfogyásához vezetett közepes attribútumszám esetén is. Hogy ezt a hibát kijavítsam, megvizsgáltam, melyek a leghatékonyabb kollekció implementációk, és az algoritmusokat is átszerveztem bizonyos pontokon a hatékonyabb végrehajtás eléréséhez. Ezek olyan módosítások voltak, amelyekre más programozási nyelvek esetén talán nem lenne szükség, így az eredeti algoritmusokba nem vezettem át őket. Szintén a hatékonyság javítása miatt az implementációban minden szint átvizsgálása után „felszabadítom” azokat a partíciókat, amelyekre már nincs szükség a további partíciók kiszámításához. Ezzel jelentős javulást értem el a memóriahasználatban.

6.3 Eredmények

Az alkalmazást több különböző adathalmazon futtattam. Ezek között voltak valós adatok különböző tulajdonságokkal illetve mesterséges adatok. A tesztekkel több célom is volt. Egyrészt vizsgáltam az algoritmus hatékonyságát szélsőséges és általános esetekben, másrészt arra voltam kíváncsi, hogy az alkalmazás valóban a törlendő sorokat találja-e meg, harmadrészt azt akartam megtudni, hogy minta használatával vesztenek-e érvényességükből a függőségek.

6.3.1 A tesztadatok bemutatása

Az adatokat részben az UCI Machine Learning Repository-ból szereztem be (Asuction & Newman). Az adathalmazok megválasztásánál törekedtem arra, hogy azok különböző tulajdonságúak legyenek. A tesztekben használt adatok az alábbiak:

- **Solar Flare Data Set**: 1389 sort és 10 attribútumot tartalmaz. A sorokban nincsenek hiányzó értékek. Az egyes attribútumoknak szűk az értéktartománya, így kicsi a valószínűsége az egyelemű ekvivalencia osztályok előfordulásának. Ez egyrészt azt jelenti, hogy ezen az adathalmazon a HIBÁS-SOROK algoritmus nem tudja kihasználni a kanonikus partíciók nyújtotta optimalizációs lehetőségeket, másrészt az ekvivalencia osztályok viszonylag nagy mérete miatt az algoritmus futási ideje közelít a legrosszabb esetéhez. Az attribútumok kategorikus értékeket tartalmaznak.
- **Mushroom Data Set:** 8124 sort és 22 attribútumot tartalmaz. Tulajdonságai hasonló az előző adathalmazéhoz.
- **Mesterséges Adathalmaz1:** 8000 sort tartalmazó, 6 attribútumból álló reláció. Az attribútumok mind egész értékűek. Bizonyos attribútumok értékeit véletlenszerűen állítottam elő, mások értékét ezekből származtattam. Emellett néhány sorhoz véletlen hibát is adtam. Ezt az adathalmazt használtam annak tesztelésére, hogy az algoritmus a helyes sorokat jelöli-e meg törlésre. Emellett az adathalmaz megfelelő összeállításával elértem, hogy viszonylag nagy számban forduljanak elő egyelemű ekvivalencia osztályok, így az adathalmazzal ellenőrizhető az is, hogy ebben az esetben hogyan alakul az algoritmus futási ideje.
- **Mesterséges Adathalmaz2:** 30000 sort tartalmaz 6 attribútumon. A hibás sorok aránya 2%, ezek a sorok véletlen hibát tartalmaznak. Az adathalmazt azért generáltam, hogy a mintahasználat hatásait ellenőrizzem vele.
- **SPECT Heart Data Set**: 267 sort tartalmaz 22 attribútumon. Ennek a halmaznak fontos tulajdonsága, hogy sok a lehetséges funkcionális függőség, és ezek közül magas az érvényesek aránya is. Ez a halmaz a hatékonyság vizsgálatához megfelelő.

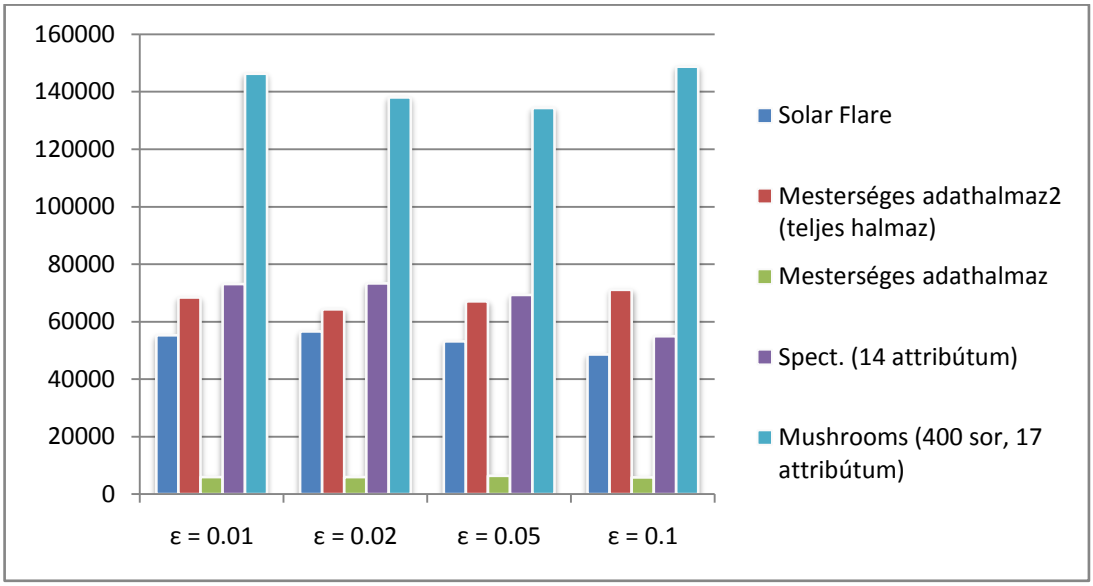
6.3.2 A hatékonyságról általában

Az algoritmus futási ideje az eredetihez képest jelentősen megnőtt. Ennek oka a HIBÁS-SOROK algoritmus egymásba ágyazott ciklusaiban keresendő. Az alkalmazás az idő legnagyobb részét

ebben a kódrészben tölti. A ciklus iterációk száma a partíciók ekvivalencia osztályainak számától függ. Minél több az osztály, annál több lépésre van szükség.

A partíciók kanonikus alakra hozása a legtöbb esetben javít a hatékonyságon, hiszen a törölt partíciókkal időt spórolhatunk meg. Vannak azonban olyan adathalmazok, amelyekre a kanonikus alakra hozásnak kevés hatása van. Ez például akkor fordulhat elő, ha az adott adathalmazban az egyes attribútumok kevés különböző értéket vehetnek fel. Ezekben az esetekben kicsi az esélye az egyelemű partícióknak, azaz a kanonikus alak kis valószínűséggel fog kevesebb ekvivalencia osztályt tartalmazni, mint az eredeti.

A tesztek eredményei pontosan ezt igazolták: amikor az algoritmus kevés sort hagyhatott figyelmen kívül, akkor a hatékonyság rosszabb volt a többi esethez képest. Az eredmények közül csak egyet emelek ki: a második adathalmazon futtattam az alkalmazást a kanonikus alakra hozás kikapcsolásával is. Ebben az esetben a végrehajtott ciklus iterációk száma körülbelül ezerszer több volt, mint amikor kanonikus partíciókat használtam. A javulás annak köszönhető, hogy a függőségek vizsgálatához kiszámított partíciókból az alkalmazás összesen több mint 650000-szer tudott ekvivalencia osztályt eltávolítani. Ez természetesen a futási időt is nagyságrendekkel csökkentette.



9. ábra. Az adatok feldolgozásának ideje (ezredmásodperc)

Az egyes halmazok feldolgozásának idejét a 9. ábra mutatja. Ahogyan az várható volt, az algoritmus futási idejét erősen befolyásolja az attribútumok száma. Látható, hogy bár a

második mesterséges adathalmaz számossága nagyobb volt a többinél, mivel az attribútumok száma kevés, az algoritmus a többi adathalmaz feldolgozási idejéhez képest kevesebb idő alatt dolgozta azt fel. Ez azt jelenti, hogy ha a függőségek felderítése előtt valamilyen módon szelektáljuk az attribútumokat (akár a felhasználó segítségével, akár valamilyen érdekességi metrika alkalmazásával), akkor a hatékonyság javítható.

Az eredményekből az is látszik, hogy **ε** növekedésével a futási idő csökken. Ez annak tudható be, hogy nagyobb **ε** esetén az algoritmus több funkcionális függőséget talál a feldolgozás elején, így a metszés több jelöltet távolít el, mint egyébként. Ennek köszönhetően kevesebb jelölt vizsgálatára van szükség és az algoritmus hamarabb véget érhet. Ezt igazolja az alábbi táblázat (és az algoritmus működéséből ez egyértelmű is).

	ε = 0.01	**ε = 0.02**	**ε = 0.05**	**ε = 0.1**
Solar Flare	10	20	18	32
Mesterséges adathalmaz2 (teljes halmaz)	12	12	16	16
Mesterséges adathalmaz	14	16	16	16
Spect. (14 attribútum)	0	3	366	2695
Mushrooms (17 attribútum)	269	373	620	511

A táblázat tartalmazza az egyes adathalmazokon a különböző **ε** értékek mellett megtalált függőségek számát. Az előző grafikonnal összevetve látható, hogy azokban az esetekben, amelyeken az algoritmus az attribútumok számához viszonyítva több függőséget talált, ott a hatékonyság kedvezőbb volt. Ezt az eredményt azonban nem tudjuk felhasználni az algoritmus hatékonyságának javítására, hiszen ehhez az ε értékének növelése szükséges, ezzel viszont nagyobb hibát engedünk meg, így a megtalált függőségek kevésbé lesznek megbízhatók.

A metszés minden esetben sokat javított a hatékonyságon. Például a Solar Flare adathalmazon **ε** = 0.1 esetén a metszés algoritmus a jelöltek 32%-át elvetette. Általánosságban is elmondható, hogy az eltávolított jelöltek száma az összes jelölt 20-30%-a volt, bár nehéz ilyen átlagot mondani, hiszen ez erősen függ az adathalmaz szerkezetétől.

6.3.3 A hibás sorok megtalálásának pontossága

Ezzel a teszttel azt vizsgáltam, hogy a HIBÁS-SOROK algoritmus megtalálja-e a hibás sorokat, valóban csak a hibás sorokat találja-e meg, és milyen mértékben függ **ε**-tól az eredmény pontossága. Ehhez az első mesterséges adathalmazt használtam.

A halmaz 6 attribútumot tartalmazott (ID, A, B, C, D, E). A tábla sorait úgy állítottam elő, hogy azokban teljesüljenek a következő funkcionális függőségek: A→B, AE → D, E → C. Emellet ID-től funkcionálisan függött az összes többi attribútum.

Az adatok előállítása során bizonyos sorokhoz véletlen hibát adtam, így ezek a sorok megsértették a felsorolt függőségeket. A hibák aránya az adathalmazon 1% volt.

A teszt futtatása után a következő eredményeket kaptam:

- Amikor **ε** közel van a hibás sorok tényleges arányához (ebben az esetben ha **ε** ≈ 0.01), akkor az algoritmus megtalálta a felsorolt függőségeket és megjelölte törlésre azokat a sorokat, amelyeket hibásan generáltam. Azaz ebben az esetben helyesen működött.
- **ε** növelésével, ahogyan az várható volt, megtalált olyan függőségeket is, amelyeket nem szándékosan alakítottam ki. Ezeknél a törlendő sorok száma magasabb volt, mint a három szándékos függőségnél.

Az eredmények rávilágítottak arra, milyen fontos **ε** helyes megválasztása, hiszen csak helyes hibaértéknél lehetünk eléggé biztosak abban, hogy tényleg csak a törlendő sorokat találja meg az algoritmus. **ε** értékét tehát hibás sorok arányának minél pontosabb előzetes becslésével kell beállítanunk. Ehhez támaszkodhatunk a hasonló adatok hibaarányáról publikált adatokra, vagy korábbi tapasztalatokra. Semmiképp nem szabad azonban túlzottan pesszimista becslést alkalmazni, hiszen az értékes sorok kitörléséhez vezethet.

6.3.4 Minta használata

A 6.1.3 fejezetben bemutattam egy lehetséges megközelítést a szükséges mintaméret meghatározására. Az ott leírt képlettel számított mintaméret gyakran nagyobb, mint a sorok száma a feldolgozandó táblában. Az algoritmust úgy implementáltam, hogy azokban az esetekben, amikor a mintaméret kisebb, mint a reláció számossága, csak a véletlen minta sorait dolgozza fel.

A mintahasználat tesztelésére készítettem el a második mesterséges adathalmazt. Ebben 10% a hibás sorok aránya (3000 a 30000-ből). A hibaarányt azért állítottam be ilyen magasra, mert

vélhetőleg több hibás sor esetén nagyobb annak a valószínűsége, hogy a mintán megtalált függőségek eltérnek a teljes adathalmazon megtaláltaktól.

A teszteket **δ** = 0.05 mellett **ε** = 0.05 és **ε** = 0.1 értékekre futtattam, ugyanis csak ezekben az esetekben volt a kiszámított mintaméret kisebb a reláció számosságánál. Az első esetben ez az érték 27699 (az sorok számának 92%-a), a másodikban 13850 (a sorok számának 46%-a).

A kevesebb feldolgozandó sornak köszönhetően a feltárás ideje jelentősen csökkent. Az egyes paraméterértékekhez tartozó időket az alábbi táblázat mutatja ezredmásodpercekben.

	δ = 0	**δ = 0.05**
ε = 0.05	67125	53000
ε = 0.1	67532	13219

A hatékonyság javulása nem volt nagy meglepetés, hiszen a minta használatának célja pontosan ez. A tesztben inkább arra voltam kíváncsi, hogy a mintán futtatva az algoritmus felfedezi-e ugyanazokat a függőségeket, amiket a teljes mintán felfedezett, illetve hogy fedez-e fel olyan függőségeket, amelyek a teljes mintán nem érvényesek.

A teszt eredménye azt mutatja, hogy az algoritmus pontosan ugyanazokat a függőségeket fedezi fel a mintán, mint a teljes adathalmazon (**ε** = 0.05 esetén 20, **ε** = 0.1 esetén 16 függőség). Minden esetben csak a valóban hibás sorok kerültek a törlendők közé, azaz az algoritmus ebben sem hibázott. Természetesen a mintából nem található meg az összes hibás sor. Ehhez egy újabb menetre van szükség, amelyben megkeressük a további hibás sorokat. Ez plusz költségeket jelent, viszont még így is gyorsabb a törlendő sorok megkeresése, mintha a teljes adathalmazon bányásznánk a funkcionális függőségeket.

A második körben is felhasználhatjuk az előállított partíciókat. Egy **X→A** érvényes funkcionális függőséghez tartozó $\pi_{XU\{A\}}$ partíció minden ekvivalencia osztályából kiválasztunk egy tetszőleges **t** sort. Ezután az adathalmaz minden u sorára ellenőrizzük, hogy **u[X]** megegyezik-e valamely választott **t**-re **t[X]**-szel. Ha igen, akkor azt kell megvizsgálnunk, hogy a **t[A]** = **u[A]** egyenlőség fennáll-e. Ha nem, akkor ez egy hibás sor. Ha az adatbázis újabb bejárása után az új hibás sorokkal együtt a hibaarány nagyobb, mint **ε**, akkor a függőséget elvetjük, ha nem, akkor megtartjuk. Ha olyan sort találunk, ami egyik korábbi ekvivalencia osztályhoz sem tartozik, akkor mindig felveszünk egy új osztályt és így folytatjuk a keresést.

Ha a verifikációt így implementáljuk, akkor elveszítjük azt a memória optimalizációs lehetőséget, hogy a fölösleges partíciókat felszabadítjuk, ugyanis minden érvényes **X→A** függőséghez meg kell őriznünk a $\pi_{X \cup \{A\}}$ partíciót.

A minta használatával kapcsolatban fontos megjegyezni, hogy a minta sorainak kiválasztásánál körültekintően kell eljárni. Ha ugyanis a mintát nem megfelelő módszerrel állítjuk össze, akkor előfordulhat, hogy a mintában nagyobb a hibaarány, mint a teljes adathalmazban. Ilyenkor elveszíthetünk funkcionális függőségeket.

6.3.5 Heurisztika

A tesztek során tapasztalati úton próbáltam bizonyítani, hogy az attribútumok valamilyen heurisztika szerinti rendezésével hatékonyság növekedés érhető el. Heurisztikának az attribútumhoz tartozó partíció rangját választottam. A rendezést csak az egyelemű attribútum halmazokra alkalmaztam. Teszteltem a növekvő és a csökkenő rendezést is.

A feltételezésem az volt, hogy a több ekvivalencia osztályt tartalmazó partíciókból nagyobb valószínűséggel távolít el sorokat a kanonikus alakra hozás, így ha azokat előrevesszük, akkor gyorsabban befejeződhet a keresés. A feltételezésem sajnos nem igazolták a tesztek, legfeljebb 2-3% javulást értem el a feldolgozás sebességében, de valószínűleg ez is csak a véletlennek köszönhető.

6.3.6 Partíciók inkrementális előállítása

A partíciók inkrementális előállítását és egyesítését a 6. ábrán leírtak szerint implementáltam. Csak a kezdeti partíciókat (azaz az egyelemű attribútum halmazokhoz tartozókat) állítottam elő ezzel a módszerrel, a későbbiek feldolgozását már teljes mértékben a memóriában végeztem, mivel a későbbi lépésekben már nincs szükség a teljes sorok tárolására, csak az elsődleges kulcs értékekre. A célom az volt, hogy a nagyméretű relációk memóriában kezelhetővé váljanak.

A módszert akkor kell alkalmaznunk, amikor a relációnak olyan sok sora van, hogy azok tárolására nem elég a rendelkezésre álló memória. Ilyenkor el kell döntenünk, hogy a relációt mekkora részekre osztjuk. A részrelációk természetesen legfeljebb akkora méretűek lehetnek, amekkorákat még a memóriában tudunk kezelni. Néha azonban a hatékonyságon is javíthatunk azáltal, hogy kisebb részreláció méretet használunk. Ha ugyanis az attribútumok száma kicsi, akkor a sorok száma erősebben befolyásolja az algoritmus futásának idejét, azaz a sorok számának csekély növekedése nagyobb növekedést okozhat a futási időben, mint sok attribútum esetén. A méretet minden esetben úgy kell megválasztani, hogy figyelembe

vesszük a lekérdezések idejét. Fontos azonban megjegyezni, hogy a kezdeti partíciók előállításának ideje töredéke a későbbi partíciók kiszámítására fordított időnek, így jelentős hatékonyságnövekedést nem érhetünk el.

A tesztek egy fontos dologra mutattak rá (az inkrementális előállítás helyességén kívül): memóriaproblémák nem csak akkor jelentkezhetnek, ha a reláció számossága nagy, hanem akkor is, ha sok az attribútum. Húsz attribútum fölött már néhány száz sor mellett is igen nagy az alkalmazás memóriafogyasztása. Ez a jelöltek magas számának és az emiatt előállított rengeteg partíciónak köszönhető. Ezt a problémát hosszas optimalizáció és a használt adatszerkezetek megváltoztatása után sem sikerült kiküszöbölnöm. A memóriafogyasztáshoz jelentősen hozzájárul a törlendő sorok tárolása és a rengeteg listamásolat (a HIBÁS-SOROK algoritmusban) is. Összességében tehát elmondható, hogy a kezdeti partíciók kiszámításánál a memória teljes elfogyasztása ugyan elkerülhető, viszont ha az attribútumok száma magas, akkor ez a későbbiekben valószínűleg elő fog jönni.

7 Összefoglalás

A diplomamunkám célja az volt, hogy kidolgozzam egy olyan algoritmust, amellyel megoldható a rejtett funkcionális függőségek kinyerése és az adatok tisztítása. Megpróbáltam a két részfeladatot ötvözni, mivel abban bíztam, hogy az adattisztítás így hatékonyabban megoldható.

A kitűzött célokat sikerült elérnem: elkészítettem egy alkalmazást, ami a kidolgozott algoritmust valósítja meg és a tesztek azt mutatják, hogy a feladat nehézségéhez képest viszonylag hatékonyan működik. Akadnak azonban hiányosságok is. Amit a legégetőbbnek érzek, az a HIBÁS-SOROK algoritmus optimalizálása. Mivel az alkalmazás a legtöbb időt ennek az algoritmusnak a végrehajtásával tölti el, ezért fontos, hogy javítsunk a hatékonyságán. Szintén fontosnak érzem a minta használat javítását. A későbbiekben, amennyiben időm engedi, szeretném optimalizálni a minta alapján feltárt függőségek verifikálását, esetleg egy új algoritmus implementálásával.

8 Irodalomjegyzék

Asuction, A., & Newman, D. J. (dátum nélk.). *UCI Machine Learning Repository*. (Irvine, CA: University of California, School of Information and Computer Science.) Forrás: http://www.ics.uci.edu/~mlearn/MLRepository.html

Atoum, J., Bader, D., & Awajan, A. (2008.). Mining Functional Dependency from Relational Databases Using Equivalent Classes and Minimal Cover. *Journal of Computer Science* , 421426.

Bar-Yossef, Z., Jayram, T. S., Krauthgamer, R., & Ravi, K. (2004.). Approximating edit distance efficiently. *FOCS '04: Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science* , 550-559.

Bell, S. (1995.). Discovery and Maintenance of Functional Dependencies by Independencies. In *Proceedings of KDD-95* (old.: 27-32.). AAAI Predss.

Bertossi, L., Bravo, L., Franconi, E., & Lopatenko, A. (2005.). Complexity and Approximation of Fixing Numerical Attributes in Databases Under Integrity Constraints. *International Workshop on Database Programming Languages* , 262-278.

Bodon, F. (2006.). *Adatbányászat.* Budapest: Bodon Ferenc.

Bohannon, P., Fan, W., Flaster, M., & Rastogi, R. (2005.). A cost-based model and effective heuristic for repairing constraints by value modification. *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data* , 143-154.

Bohannon, P., Fan, W., Geerts, F., Jia, X., & Kementsietsidis, A. (2007.). Conditional Functional Dependencies for Data Cleaning. *IEEE 23rd International Conference on Data Engineering* , 746-755.

Chomicki, J., & Marcinkowski, J. (2005.). Minimal-change integrity maintenance using tuple deletions. *Information and Computation* , 90-121.

Date, C. J. (2003.). *An Introduction to Database Systems, Eighth Edition.* Addison Wesley.

Dr. Abonyi, J. (2006). *Adatbányászat, a hatékonyság eszköze.* Budapest: ComputerBooks Kiadó Kft.

Eckerson, W. W. (2002.). *Excerpt from TDWI's Research Report - Data Quality and the Bottom Line*. Forrás: The Data Warehouse Institute: http://www.tdwi.org/research/display.aspx?ID=6589

Hern'andez, M. A., & Stolfo, S. J. (1995.). The merge/purge problem for large databases. *In Proceedings of the 1995 ACM SIGMOD* , 127-138.

Hong, Y., Hamilton, H. J., & Butz, C. J. (2002.). FD_Mine: Discovering Functional Dependencies in a Database Using Equivalences. *ICDM '02: Proceedings of the 2002 IEEE International Conference on Data Mining* , 729-732.

Huhtala, Y., Karkkainen, J., & Toivonen, H. (1999.). TANE: An Efficient Algorithm for Discovering Functional and Approximate Dependencies. *The Computer Journal , 42.*, 100-111.

Ilyas, I. F., Markl, V., Haas, P., Brown, P., & Aboulnaga, A. (2004.). CORDS: automatic discovery of correlations and soft functional dependencies. *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data* , 647-658.

John, G. H., & Langley, P. (1996.). Static versus Dynamic Sampling for Data Mining. *Proceedings of the Second International Conference on Knowledge Discovery in Databases and Data Mining* .

Kaewbuadee, K., Temtanapat, Y., & Peachavanish, R. (2006.). Data Cleaning Using FD from Data Mining Process. *IADIS International Journal Computer and Information Systems* , 117-131.

Kimball, R., & Ross, M. (2002.). *The Data Warehouse Toolkit* (Second Edition. kiad.). USA: John Wiley and Sons, Inc.

Kivinen, J., & Mannila, H. (1993.). *The power of sdampling for data mining.* Helsinki: University of Helsinki, Department of Computer Science.

Koh, J. L., Lee, M. L., Khan, A. M., Tan, P. T., & Brusic, V. (2004.). Duplicate Detection in Biological Data using Association Rule Mining. *2nd European Workshop on Data Mining and Text Mining for Bioinformatics* .

Lee, C.-C., Jiang, J.-H. R., Huang, C.-Y., & Mishchenko, A. (2007.). Scalable exploration of functional dependency by interpolation and incremental SAT solving. In *ICCAD '07* (old.: 227-233.). San Jose, California: IEEE Press.

Lim, W. M., & Harrison, J. (1997.). Discovery of Constraints from Data for Information System Reverse Engineering. In *In Proc. of Australian Software Engineering Conference* (old.: 39.). Washington, USA: IEEE Press.

Maier, D. (1983.). *Theory of Relational Databases.* Computer Science Press.

Maletic, J. I., & Marcus, A. (2000). *Automated Identification of Errors in Data Sets.* Memphis: The University of Memphis.

Maletic, J. I., & Marcus, A. (2000.). Data Cleansing: Beyond Integrity Analysis. *Proceedings of the Conference on Information Quality* .

Mannila, H., & Toivonen, H. (1997.). *Levelwise search and borders of theories in knowledge discovery.* Helsinki, Finnország: University of Helsinky, Department of Computer Science.

Müller, H., & Freytag, J.-C. (2003). *Problems, Methods, and Challenges in Comprehensive Data Cleaning.* Berlin: Humboldt-Universitat zu Berlin, Institut für Informatik.

Pedersen, T. B., Pedersen, C., & Jensen, C. S. (2005.). Multidimensional databases. In *The Industrial Information Technology Handbook* (old.: 1-13.).

Rahm, E., & Do, H. H. (2000.). Data Cleaning: Problems and Current Approaches. *IEEE Bulletin of the Technical Committee on Data Engineering* .

Raman, V., & Hellerstein, J. M. (2001). Potter's Wheel: An Interactive Data Cleaning System. *VLDB.* Róma.

Savnik, I., & Flach, P. A. (1993.). A Bottom-Up Induction of Functional Dependencies From Relations. *In Proceedings of the AAAI'93 Workshop on Knowledge Discovery in Databases* , 174-185.

Toivonen, H. (1996.). Sampling Large Databases for Association Rules. *roceedings of the 22th International Conference on Very Large Data Bases* , 134-145.

Wyss, C., Gianella, C., & Robertson, E. (2001.). FastFDs: A Heuristic-Driven, Depth-First Algorithm for Mining Functional Dependencies from Relation Instances. *Data Warehousing and Knowledge Discovery* , 101-110.

Zimanyi, E., & Pirotte, A. (1997.). Imperfect knowledge in databases.

Függelék

A. Függelék – algoritmusok

A.1. METSZÉS

ALGORITMUS: METSZÉS
INPUT: L_ℓ, az ℓ-EDIK SZINT ATTRIBÚTUM HALMAZAINAK LISTÁJA
OUTPUT: L HALMAZAI A FÖLÖSLEGESEK NÉLKÜL

1. **for** $X \in L_\ell$ **do**
2. **if** $C^+(X) = \emptyset$ **then**
3. X-et töröljük L_ℓ-ből
4. **if** X szuperkulcs **then**
5. **for** $A \in C^+(X) \setminus X$ **do**
6. **if** $A \in \bigcap_{B \in X} C^+(X \cup \{A\} \setminus \{B\})$ **then**
7. $X \rightarrow A$ érvényes
8. X-et töröljük L_ℓ-ből

Az algoritmus eltávolítja azokat az attribútum halmazokat a listából, amelyekkel fölösleges foglalkoznunk. Ilyenek azok, amelyekhez egyetlen jelölt sem tartozik, illetve azok, amelyek kulcsok vagy szuperkulcsok. Az utóbbi esetben a felderített funkcionális függőségek közé fel is veszi a szükségeseket.

Annak eldöntése, hogy egy attribútum halmaz szuperkulcs-e, a hozzá tartozó partíció alapján történik. Ha a partícióban (még a kanonikus alakra hozás előtt) pontosan annyi ekvivalencia osztály van, mint ahány sor, akkor az attribútum halmaz szuperkulcs.

A.2. SZORZÁS

Ez az algoritmus a $\boldsymbol{\pi_X}^*$ és a $\boldsymbol{\pi_A}^*$ partíciókból előállítja a $\boldsymbol{\pi_{X\cup\{A\}}}^*$ partíciót. A kanonikus alakot úgy érjük el, hogy egy ideiglenes táblázatban tároljuk $\boldsymbol{\pi_{X\cup\{A\}}}^*$ ekvialencia osztályait és ezeket csak akkor adjuk ténylegesen $\boldsymbol{\pi_{X\cup\{A\}}}^*$-hoz, ha 2, vagy több sort tartalmaznak.

ALGORITMUS: SZORZÁS
INPUT: π_X^* és π_A^*
OUTPUT: $\pi_{X\cup\{A\}}^*$

1. $\pi_{X\cup\{A\}}^* = \emptyset$
2. **for** i = 1 to $|\pi_X^*|$ **do**
3. **for** $t \in c'_i \in \pi_X^*$ **do**
4. T[t] = i
5. **for** $c'' \in \pi_A^*$ **do**
6. **for** $t \in c''$ **do**
7. **if** T[t] definiált **then**
8. $S[T[t]] = S[T[t]] \cup \{t\}$
9. **for** $t \in c''$ **do**
10. **if** $|S[T[t]]| \geq 2$ **then**
11. $\pi_{X\cup\{A\}}^* = \pi_{X\cup\{A\}}^* \cup \{S[T[t]]\}$

A.3. KÖVETKEZŐ-SZINT

A 2. sorban szereplő PREFIX-BLOKKOK algoritmus halmazok egy listáját állítja elő. Ezek a halmazok azokat az attribútumokat tartalmazzák, amelyekhez van két olyan elem $\mathbf{L_\ell}$-ben, amely csak az utolsó attribútumon tér el. Ekkor az a két utolsó attribútum egy halmazba kerül. A KÖVETKEZŐ-SZINT algoritmus ezek alapján állítja elő a $\mathbf{L_{\ell+1}}$–et. További információk az algoritmusról (Huhtala, Karkkainen, & Toivonen, 1999.) cikkében találhatók.

ALGORITMUS: KÖVETKEZŐ-SZINT
INPUT: L_ℓ, AZ ℓ-EDIK SZINT
OUTPUT: $L_{\ell+1}$, AZ ℓ+1-EDIK SZINT

1. $L_{\ell+1} = \emptyset$
2. **for** $K \in$ PREFIX-BLOKKOK(L_ℓ) **do**
3. **for** $\{Y, Z\} \subseteq K, Y \neq Z$ **do**
4. $X = Y \cup Z$
5. **if** $\forall A \in X, X \setminus \{A\} \in L_\ell$ **then**
6. $L_{\ell+1} = L_{\ell+1} \cup \{X\}$

Köszönetnyilvánítás

Köszönettel tartozom témavezetőmnek, Dr. Juhász István egyetemi adjunktusnak a hasznos tanácsokért, az egyetemi éveim alatt nyújtott állandó támogatásáért és folyamatos segítőkészségéért. A kezdetektől tartó szakmai irányítása nélkül, amellyel jelentősen hozzájárult informatikai szemléletem és érdeklődésem kialakulásához, dolgozatom nem készülhetett volna el.

Köszönöm Hannu Toivonennek, a Computer Science University of Helsinki professzorának, hogy rendelkezésemre bocsátotta cikke kibővített változatát, amely diplomamunkám alapjául szolgált.

Hálával tartozom a UCI Machine Learning Repository fenntartóinak, hogy tesztjeimhez felhasználhattam az általuk összegyűjtött adathalmazokat.

Köszönöm mindazoknak, akik hozzájárultak ahhoz, hogy a Debreceni Egyetemen folytathassam tanulmányaimat.

Végül, de nem utolsó sorban, köszönöm Sajtos Ágnesnek, hogy diplomamunkám készítése során mindenben támogatott.