Korszerű adatkezelési eljárások

Önálló laboratórium 1 (BMEVIMIM815) beszámoló

Készítette: Csikós Donát (B6ZRF5)
Konzulens: Horváth Ákos, Ráth István
BME-VIK, Méréstechnika és Információs Rendszerek Tanszék
Szolgáltatásbiztos rendszertervezés szakirány, 2009/2010. II. félév

Tartalomjegyzék

1 Bevezetés

Napjainkban a szoftverfejlesztésben a relációs adatbázis-kezelő rendszereket széles körben használják. Ennek oka elsősorban az, hogy feladatukat hatékonyan és biztonságosan képesek végrehajtani. Azonban kialakulóban vannak olyan új technológiák, melyek újszerű módszerekkel képesek a relációs adatbázis-kezelésre reális alternatívát nyújtani. Ilyenek az objektum-orientált adatbázisok és a gráfadatbázisok. Ezek az eszközök viszont tipikusan nem olyan kiforrottak, ebből adódóan pedig nem garantálható, hogy alkalmasak éles többfelhasználós, tranzakciós rendszerekben az adattárolás megvalósítására.

Az önálló labor 1 (BMEVIMIM815) tárgy keretében ezeket az újszerű alkalmazásokat ismertem meg, illetve egy olyan tesztelő programot készítettem el, melynek segítségével vizsgálni lehet, hogy az egyes adatbázis-kezelők milyen hatékonysággal képesek tranzakciós műveleteket végrehajtani. A félév során végzett munkám részleteit ebben a dokumentumban foglaltam össze.

2 Adatbázis-kezelő rendszerek sajátosságai

A félév első felét azzal töltöttem, hogy megismerkedtem az objektum-orientált és gráfadatbázisok alaptulajdonságaival, valamint áttanulmányoztam és kipróbáltam két-két konkrét megvalósítást. Vizsgálódásaim közben külön figyelmet fordítottam arra, hogy az egyes eszközök a tranzakciós műveleteket és a zárolási mechanizmusokat hogyan valósítanak meg. Erre azért volt szükség, mert a később elkészített tesztprogramba ezen tulajdonságok segítségével lehetett integrálni őket, illetve a tesztkörnyezet tervezésekor is figyelembe kellett venni, hogy pontosan milyen esetekre is kell felkészülni.

2.1 Relációs adatbázis-kezelők

A relációs adatbázis-kezelés legnagyobb előnye a kiforrottsága. A 60-as évek óta használják és fejlesztik őket, valamint olyan informatikai nagyvállalatok szállítanak megoldásokat, mint az Oracle, vagy a Microsoft.

Ezen felül egy nagyon erős matematikai leírás, a relációalgebra is adott az adatok ábrázolására. Erre építve lehet egységesíteni az adatbázis tervezést, illetve a hozzá tartozó lekérdező és adatmanipulációs nyelvet, melynek neve: Structured Query Language, vagy röviden SQL. Ebből több szabvány is létezik, a termékek legnagyobb körében megvalósított verziója az SQL:1999.

Az erős matematikai alapok nagymértékű optimalizációra is lehetőséget adnak. Többfelhasználós környezetben az adatok zárolási szintjeit az adott alkalmazásnak megfelelően lehet finomhangolni.

Sok egyéb mellett ezen tulajdonságai miatt elterjedtek a relációs adatbázis-kezelők, melyekre széles körben épülnek eszközök, IT megoldások. Azonban figyelembe kell venni azt a tényt is, hogy vannak olyan területek, melyeken a relációs adatbázisok használata korlátozott, nehézkes. Emellett a mai napig nem sikerült valóban egységes felületet biztosítani ezen adatbázisokhoz, az egyes gyártók külön-külön verziót szállítanak a saját SQL megvalósításukból. Ez különösen igaz a szerveroldali programozásra.

Egy másik példa az objektum-orientált programozási paradigmakör. Egy perzisztencia-képes objektum adatainak elmentéséhez egy leképezést kell megvalósítani az OO világ objektumai és az adatbázis táblái között. Erre számos eszköz készült el, de figyelembe kell venni, hogy külön

fejlesztési lépést jelent egy alkalmazás elkészítésekor, amibe időt és energiát kell fektetni. Szintén említésre méltó, hogy vannak olyan adatstruktúrák, melyeken szintén nem hatékony ilyen modellben dolgozni. Ilyen struktúrák például a gráfok, illetve nagyméretű hálók, hálózatok.

Ezek az ellenpéldák inspirálták azokat a fejlesztőket, akik az alább taglalt Objektum-orientált adatbázisokat és gráfadatbázisokat jelenleg is folyamatosan fejlesztik.

2.2 Objektum-orientált adatbázisok

Az Objektum-orientált adatbázisok alapkoncepciója, hogy segítségükkel a memóriabeli objektumokat közvetlenül, leképezés nélkül lehet perzisztens tárba menteni. Ennek megfelelően az ilyen típusú rendszerek alapstruktúrája objektumokból és referenciákból áll. Ez alapján közvetlenül adódik az az előny, hogy nincs szükség köztes rétegben történő adatleképezésre, amivel például a fejlesztési idő is csökkenhet. Emellett, mivel az adatok között referenciák mentén lehet haladni, ezért a bejárás alapú keresés jóval hatékonyabb lesz, mint a relációs megvalósítás esetén. Ezt az ODMG (Object Data Management Group) által bejegyzett OQL (Object Query Language) lekérdezési szabvány nyelv támogatja[1].

Ezen adatkezelők elterjedésének fő korlátja az, hogy teljesítményük a mai napig elmarad a relációs adatbázisokétól. Jól mutatja ezt, hogy például a bennük megvalósított zárolási stratégiák objektumszinten zajlanak, ennek nyilvántartása pedig nagy számítási többletköltséggel jár. Emellett tipikusan a többfelhasználós hozzáférések kezelése is lassabb, mint az elvárt.

db4objects

Ennek [1] az adatbázis-kezelőnek fő tulajdonsága az egyszerűsége. Ez úgy nyilvánul meg, hogy az egyes szolgáltatások megvalósításához csak felületet biztosít, de konkrét implementációt vagy külső csomagokkal lehet hozzá telepíteni, vagy a programozó feladata, hogy azokat megvalósítsa.

Maga a rendszer Java-val, vagy .NET környezettel képes együttműködni. Az objektumokhoz való hozzáférésre a rendszer több felületet is kínál (OQL, JDOQL, SODA), a dokumentáció azonban az úgynevezett Native Queries nyelvet ajánlja, melyben az implementációs nyelv objektumaiból kell felépíteni az egyes lekérdezéseket, ahogyan az alábbi Java nyelvű példa is mutatja:

```
List <Pilot> pilots = db.query(new Predicate<Pilot>() {
    public boolean match(Pilot pilot) {
        return pilot.getPoints() == 100;
    }
});
```

Ezt felhasználva a rendszer képes nagymértékű optimalizációt is végrehajtani a lekérdezések, ezzel is javítva a teljesítményt. A legfrissebb verziók tesztjei [2] már meglepően jó teljesítményt mutatnak.

1 Bővebben az objektum-orientált adatbázisokról és a hozzájuk kapcsolódó szabványokról: http://www.odbms.org

A rendszer, sok egyéb objektum-orientált adatbázis-kezelővel ellentétben, képes szerver-kliens környezetben működni az eredeti beágyazott működés mellett. Azonban fontos megjegyezni, hogy alapértelmezésben úgynevezett „overly optimistic locking”-ot, azaz túlzottan optimista zárolást alkalmaz. Ennek lényege, hogy bár az objektumokon elhelyezésre kerülnek a kizáró, illetve megosztott zárak, azonban semmilyen ütközés detektálás nem történik, a tényleges kezelésről itt is a programozónak kell gondoskodnia (vagy külső csomagból már létező megvalósítást kell betöltenie). Ezt db4o szemafor osztályával, illetve az egyes tranzakciós eseményeket elérhetővé tevő callback mechanizmusával lehet megvalósítani.

Perst

A Perst [3] az előző eszköztől eltérően elsősorban arra koncentrál, hogy minél gyorsabb legyen, feladva a párhuzamos hozzáféréseket hatékonyan kezelésének lehetőségét. Ebből adódóan alapvetően beágyazott, egyfelhasználós üzemmódban használatos, akár igen nagy adathalmazok kezelésére is.

A Perst is képes több nyelven kommunikálni, a Java, a .NET és a Mono szerepel a használható környezetek listáján. A tárolás implementálásánál fontos tényező, hogy interfészt kell megvalósítania a perzisztencia-képes objektumoknak. Ezen keresztül egy úgynevezett shadow object mechanizmussal van megvalósítva az adattárolás. Ez azt jelenti, hogy egy adott adott folyamat(process) nem közvetlenül a tárolt objektumot éri el, hanem annak egy másolatát. A sikeres tranzakció végén ez a másolat kerül az eredeti objektum helyére. Mivel az írás atomian van megvalósítva, ezért az adatbázis az egyik konzisztens állapotból közvetlenül kerül át egy másik konzisztens állapotba. Így nincs szükség naplózásra, és a helyreállítás is nagyon gyors tud lenni, de a több felhasználó kezelése valóban nehézkessé válik.

Ha mégis több folyamat éri el az adatbázist, a következő izolációs szintek állnak rendelkezésre:

- *Synhronized access to the database*: Egyetlen szál éri el az adatbázist.
- *Cooperative transactions*: A program szálai megosztoznak a tranzakción, és nyelvi elemekkel készülnek fel a versenyhelyzetekre
- *Exclusive per-thread transactions*: Az adatbázishoz adott időben csak egyetlen szál fér hozzá.
- *Serializable per-thread transactions*: Minden tranzakció úgy fut a rendszerben mintha egyedül ő futna csak. A megvalósításban a zárak elhelyezése a programozó feladata.

Az egyes zárolási mechanizmusokra előre elkészített implementációk érhetőek el, mint az alaprendszer kiegészítői (erre egy példa a verziókezelő megvalósítás).

2.3 Gráfadatbázisok

Egy másik egészen új megoldás, hogy az adatbázisok alapstruktúrájának a gráfokat választják. Ebben a megközelítésben értelemszerűen az adatokat és kapcsolataikat csúcsokkal és élekkel reprezentálják, erős matematikai alapokra helyezve ezzel a perzisztencia megvalósítását.

Az elképzelés frissességét mutatja, hogy jelen pillanatban csak kevés konkrét implementáció létezik, melyeket folyamatosan változtatnak, fejlesztenek. Tipikusan igaz az, hogy az egyes eszközökben a lekérdezések teljesen egyéni módon vannak megvalósítva.

Fontos megjegyezni, hogy ha ilyen adatbázisra építjük a programunkat, akkor új szemlélet kell a fejlesztéshez, mivel az egyes objektumok, és a mögöttük álló hálózat közötti kapcsolatot más szemantika alapján kell karban tartani.

Az ok, amiért érdemes használni ezt a megközelítést, az hasonló, mint az objektum-orientált esetben: nagyméretű, szemi-strukturált, esetleg gyorsan változó adatok esetén jóval hatékonyabbak az adattárolás műveletei. Ilyen esetekre sok, életszerű példa van. Néhány példát említve ilyen a manapság divatos közösségi oldalak kapcsolati hálózata, a bioinformatikai kutatások, szemantikus hálók leírása, illetve egyes hálózatelméleti és számítás-intenzív feladatok megoldása.

HyperGraphDB

Ez az eszköz [4] matematikai alapként a hipergráfokat[2], a gráfok egy általánosabb reprezentációját használja fel. A hipergráf fő tulajdonsága, hogy az élek nem csak két, hanem tetszőleges számú csúcs között húzódhatnak.

Fontos, hogy alacsony szinten egy speciális relációs adatbázis-kezelőt, a BekerleyDB-t használja, ami kulcs-érték párosokat tud hatékonyan tárolni.

Sajnos igaz a rendszerre, hogy rendkívül rosszul dokumentált, a finomabb részleteket csak komoly utánanézéssel lehet kideríteni. Ettől függetlenül azonban meglepően nagy tudású, például képest az adatokat elosztott módon peer-to-peer protokoll felett működni, és lehetősége ad szerver-kliens architektúrában való üzemre a szokásos beágyazott megvalósítás mellett.

Implementációs részlet, hogy az egyes elemekhez az adatbázisból ún. leírókon keresztül lehet hozzáférni, valamint a lekérdezésekkor is ilyeneket kapunk vissza. Egy tipikus lekérdezés figyelhető meg a következő példában:

2 Hipergráfok matematikai leírása: http://en.wikipedia.org/wiki/Hypergraph

```
HGQueryCondition condition = new And(
              new AtomTypeCondition(Book.class),
              new AtomPartCondition(new String[]{"author"},
"H. P. Lovecraft", ComparisonOperator.EQ));
HGSearchResult rs = graph.find(condition);
try
{
    while (rs.hasNext())
    {
        HGHandle current = rs.next();
        Book book = graph.get(current);
        System.out.println(book.geTitle());
    }
}
finally
{
    rs.close();
}
```

Az eszköz zárolási rendszere úgy van megvalósítva, hogy felüldefiniálja a java.util.concurrent.locks.ReadWriteLock osztályt, melyen keresztül közvetve lehet a BekerleyDb elemeit lockolni.

Neo4j

Ez az eszköz [5] egy tipikus beágyazott alkalmazás, mely sokkal inkább az alkalmazásfejlesztésre koncentrál, mint az előző, melynek dokumentációjában is utalnak arra, hogy a kialakítása elsősorban tudományos felhasználási körre fókuszál.

Szerencsére a neo4j sokkal jobban dokumentált, részletesen kidolgozott példaprogramokon lehet elsajátítani a használatát. Alapvetően a Java nyelv alaptípusait lehet vele eltároltatni. Ezt úgy lehet megtenni, hogy mind a csúcsokhoz, mind pedig az élekhez kulcs-érték párokat rendelve lehet összeállítani a tárolási gráfot.

Sok kiegészítő elérhető hozzá, melyek adott esetben kifejezetten hasznosak tudnak lenni. Ilyenek például:

- Full-text-search motor.
- Tárolt adatok metamodelljének lekérdezése.
- Adminisztrációs céllal használható távoli elérés.

A tranzakciók megvalósításához saját indítás és commit tartozik. Ezt az alábbi példa szemlélteti:

```
Transaction tx = graphDb.beginTx(); // begin transaction
try{
    Node node1 = graphDb.createNode();
    Node node2 = graphDb.createNode()
    tx.success(); // flag success
}
catch ( SomeException e ){
    // Exception handling
}
finally{
    tx.finish(); // rollback if not success, else commit
}
```

Az írás és olvasás zárolását alapértelmezésben a rendszer saját maga elintézi (read committed izolációs szinten), de ez könnyedén felüldefiniálható. Ha az így kialakított program véletlenül holtpontra juttatja a feldolgozást, akkor az egyik – véletlenszerűen kiválasztott – tranzakció DeadLockException-nel száll el.

Összességében ez az eszköz bizonyult a legkényelmesebbnek kis mérete, könnyű integrálhatósága és Eclipse pluginként[3] megjelenő vizualizációs eszköze miatt.

3 Neoclipse: http://wiki.neo4j.org/content/Neoclipse_Guide

3 Az elkészült program leírása

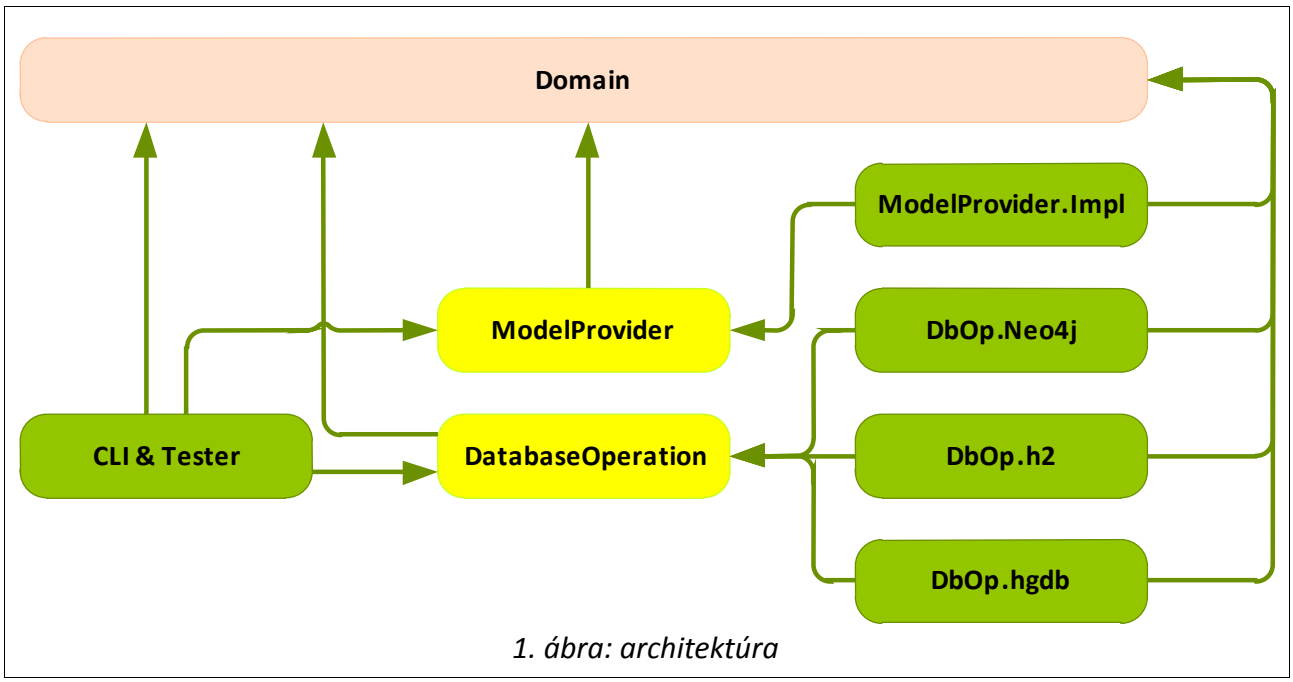
A félév második felében elkészítettem egy tesztelő programot, melynek célja, hogy segítségével mérni lehessen az abba beintegrált adatbázis-kezelők olyan tranzakciós tulajdonságait, mint például a holtpontok száma, vagy a tranzakciók futásához szükséges idő.

Ehhez olyan modellt és végrehajtó modult dolgoztam ki mely egyrészt képes jól megfogható tranzakciókat generálni és lefuttatni, másrészt pedig egységesen kezeli az egyes adatbázis-kezelők zárolásának és tranzakcióinak műveleteit. Az így elkészült környezetbe betöltött rendszerek tulajdonságai összevethetővé váltak.

3.1 Architektúra

A program elkészítésekor követelmény volt, hogy kényelmesen kiegészíthető legyen külső modulokkal, melyeket egymástól jól elválasztva kezelendőek. Emiatt, és személyes előismereteimből adódóan egyértelmű volt, hogy a választás az Equinox[4] keretrendszerben való megvalósításra fog esni. A megvalósításra kerülő modulok (bundle-ök), és a köztük megjelenő függőségek az alábbi ábrán láthatóak:

4 Az Equinox egy OSGi framework implementáció, mely az Eclipse platform alapját alkotja. Bővebben: http://www.eclipse.org/equinox/



1. ábra: architektúra

Az egyes bundle-ök részletes leírása a következő pontban található.

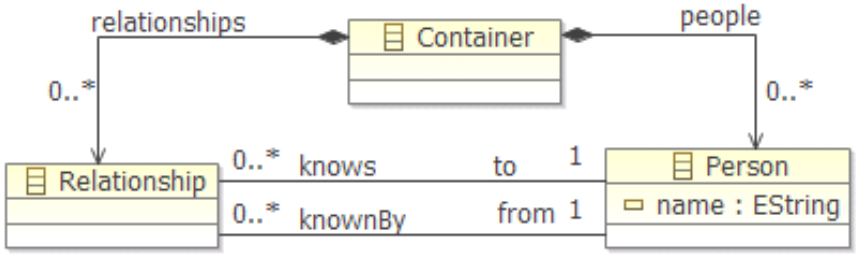
3.2 Az egyes modulok leírása

Domain

Ebben a bundle-ben található meg a rendszer által használt adatmodell, mely tartalmazza egyrészt a modellezett rendszer alapelemeit, illetve a tranzakciók modelljét is. Erre a bundle-re minden más bundle, mint függőség hivatkozik, mivel minden funkció a benne található osztályokkal dolgozik.

Az alapmodell EMF segítségével ecore modellként dolgoztam ki, amiből aztán generálva lett a tényleges java kód. Az EMF használata azért volt indokolt, mivel a generált kódhoz sok kényelmi szoltáltatás külön implementáció nélkül elérhető. Ilyen a legenerált modellpéldányok lemezre mentése és betöltése.

Az alapmodell gyakorlatilag egy kapcsolati hálózatot [6] ír le. Ebben felhasználók vannak, akik képesek regisztrálni a rendszerbe,majd kereséseket hajthatnak végre, illetve megjelölhetik, hogy kit ismernek. Ez a modell jól leképezhető gráfokra, ezért illeszkedik a kijelölt tesztelési feladathoz. A konkrét modell a következő ábrán látható:

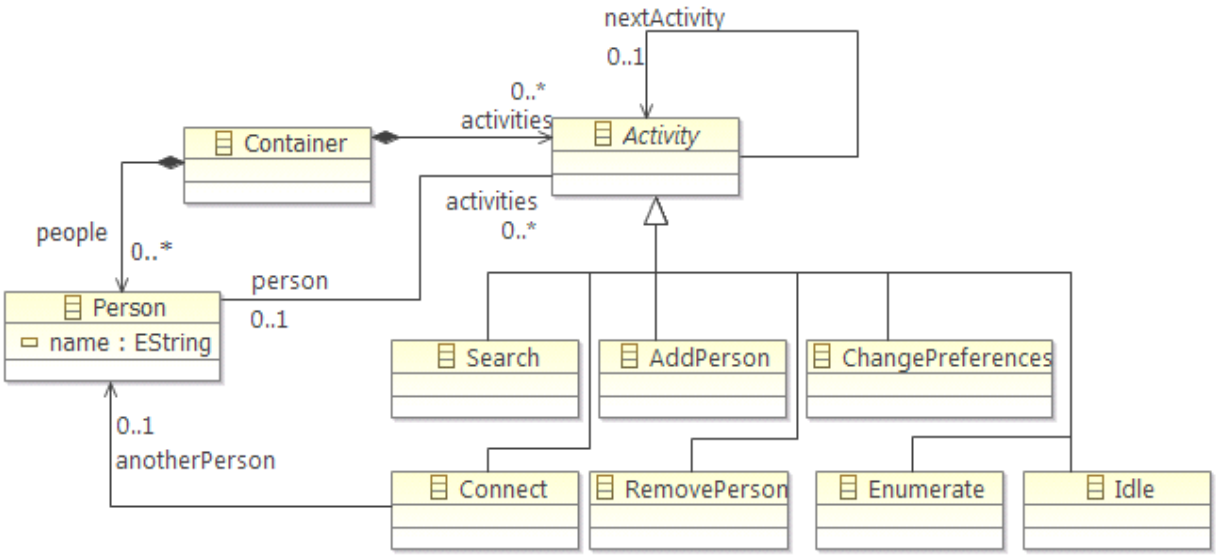


2. ábra: alapmodell

A Container, a modell gyökerét alkotó objektum. Ez tartalmazza a felhasználókat reprezentáló Person és a kapcsolatokat jelző Relationship objektumokat. A könnyebb navigáció miatt kétirányú kapcsolatok vannak felvéve.

Fontos megjegyezni, hogy ez az alapmodell példány kerül tárolásra a tesztelendő adatbázisokban, az alább tárgyalt viselkedés modellezés pedig az alapmodellel együtt egy fájlba kerül generálásra, ahonnan később fel lehet azt olvasni. Így lehet ugyanazt a tesztet újra és újra,de eltérő adatbázisokon végrehajtani.

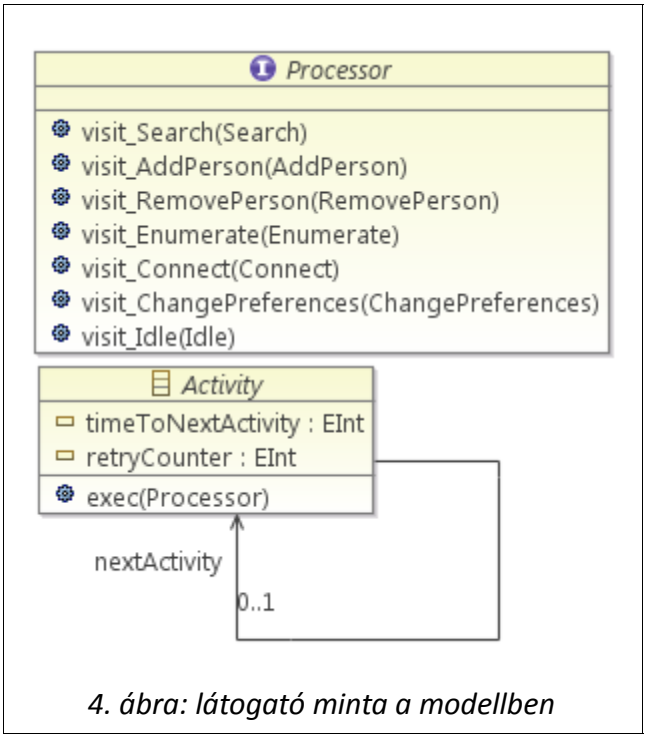
Az alapmodellhez tehát hozzátartozik felhasználók viselkedésének modellezése, általuk indított műveletek kezelése, generálása. Ennek megvalósítására kibővítettem az alapmodellt, melyet a következő ábra mutat:



3. ábra: viselkedés modellezése

Itt látható, hogy az egyes tranzakcióknak az aktivitások felelnek meg. Minden aktivitáshoz tartozik egy következő aktivitás, illetve ha nem, akkor az adott felhasználó nem indít újabb tranzakciót. Az egyes konkrét aktivitásoknak lehet extra paramétere, amiknek felhasználása a végrehajtás közben történik meg. Ilyen például a Connect, ahol az anotherPerson referencia által mutatott felhasználó lesz behúzva ismerősként az adatbázisban.

Az aktivitások adatainak elérését a modellben a látogató (visitor) minta alapján valósítottam meg a következő módon:



4. ábra: látogató minta a modellben

Az ábrán látható az absztrakt exec() művelet, melynek minden megvalósításában az adott objektum meghívja magára a paraméterül kapott processor objektum megfelelő függvényét. Ezt a hívást a generált kódhoz kézzel adtam hozzá. Ezen technika segítségével egységesen lehetett kezelni az összes Activity-t mint tranzakciót külső implementációval.

ModelProvider

A modelprovider név két bundle-re utal, egy interfész és egy implementációs bundle-re. Az interfészben egyetlenegy függvény van, melynek alakja a következő:

```
Container getModel(Dictionary<String, Object> params);
```

Ezzel az interféssszel lehet generáltatni új, modellt, illetve létező modellt betölteni. A generálás és a betöltés a bemenő paraméteren múlik. Ha létező azonosító van benne megadva, akkor a modell betöltése történik meg, ha pedig nem, akkor az új modell generálódik aszerint, hogy milyen más opciókkal van feltöltve a params objektum. Megadható többek között a felhasználók szám, a kapcsolatok száma és az aktivitások száma is. A függvény a modell gyökérobjektumával tér vissza.

Ezek megvalósítása az implementációs bundle-ben történik meg, ahol OSGi szolgáltatásként adódik hozzá a rendszerhez. A generáláskor nem teljesen véletlenszerűen generálódnak az Activity objektumok, azonban ennek precíz kidolgozása további munkálatokat igényel.

DatabaseOperation

Ebben a bundle-ben definiáltam az egyes tranzakciókban szereplő műveletek interfészét, melyben olyan elemek találhatóak mint például a tranzakció indítása, egy objektum zárolása, vagy egy kapcsolat felvétele két felhasználó közé. Az interfész neve: DatabaseOperation. Az egyes konkrét megvalósítások külön bundle-ökben vannak leírva, azokat az OperationFactory#createOp(String ServiceId) metódusán keresztül lehet példányosítani. Természetesen a regisztrált megvalósítások azonosítóit is le lehet kérdezni ugyanezen osztályból.

Fontos megemlíteni, hogy itt van definiálva egy DatabaseOperationException nevű hibaosztály, melynek leszármazott osztályain keresztül lehet egységesen kezelni az olyan jelenségeket, mint a holtpontok, vagy adatbázis rendellenes viselkedése (DeadlockDBOE és UnknownDBOE hiba dobódik, melyeket külön-külön le lehet kezelni).

A DatabaseOperation-t jelenleg 3 bundle valósítja meg, melyeken a tesztelést elvégeztem. Kettő ezek közül a már ismertetett HyperGraphDB, illetve a neo4j, de az összehasonlítás kedvéért egy H2 nevű klasszikus relációs adatbázis-kezelőt is beépítettem.

Cli & tester

Ebben a bundle-ben a nevéből adódóan több dolog is helyet kapott, nevezetesen a felhasználói inputok kezelése és a tesztelést megvalósító kód is. Ennek szétválasztása a további munka részét fogja képezni.

A felhasználói input fogadása az Activator osztályon keresztül került megvalósításra, pontosabban az OSGi console-on keresztül kell kiadni a test parancsot a megfelelő paraméterekkel (azonosító, modell mérete, tranzakciók száma, stb.). Ez először a ModelProvider modulon keresztül betölti, vagy legenerálja a modellt attól függően, hogy létezett-e már ilyen azonosítóval mentett modell. Ezután a modul elkéri az összes elérhető DatabaseOperation megvalósítást, és végrehajtja a teszteket. Ebben a következő lépések kerülnek végrehajtásra:

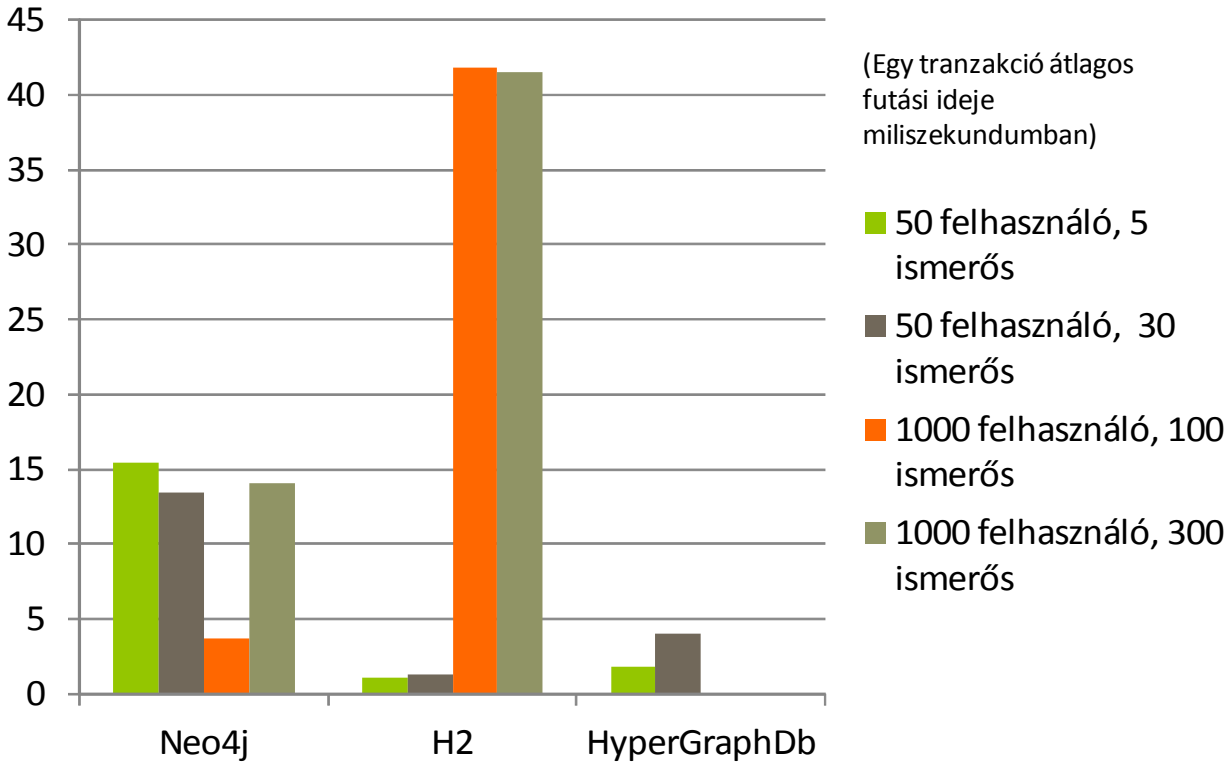
- A DatabaseOperation#initDb() függvénye az alapmodell kiinduló alapállapotát betölti az adott konkrét adatbázis-kezelőbe.
- A Runner osztály minden egyes felhasználóhoz külön szálat indít, és az egyes hivatkozott Activity objektumok által reprezentált műveleteket végrehajtja.
- A teszt lefutása végén meghívódik a dispose() művelet, mely minden, már nem szükséges erőforrást elenged.

Az egyes aktivitások az adatmodell definiált Processor interfész implementálásával kerülnek feldolgozásra. Az egyes visit_XXX() függvényekben a DatabaseOperation interfész elemein keresztül vannak egységesen kezelve a tranzakció elemi lépései, illetve, hogy mi történjen például holtpont esetén.

A tesztek adatainak méréséről a Stat osztály gondoskodik, melyhez adatait a singleton minta alapján az INSTANCE() statikus metóduson keresztül lehet elérni.

3.3 Tesztelési eredmények

A tesztekhez először különböző méretű és kitöltöttségű kiinduló modelleket generáltam. A mérésekben az előzetes tervekkel ellentétben csak az egy tranzakcióra jutó átlagos időt tudtam mérni, mivel a három vizsgált eszközből kettőben nem sikerült holtpontra figyelmeztető hibát dobatni. Ez természetesen később pótlásra fog kerülni. A teszteredmények az alábbi ábrán láthatóak:



5. ábra: mérési eredmény – egy tranzakció átlagos futási ideje

A tesztek egy átlagos laptopon lettek futtatva, melynek paraméterei a következőek:

- Processzor: Core2Duo T7300, 2.0GHz
- Memória: 3Gb
- Operációs rendszer: Windows 7 Enterprise edition, 32bit

Minden teszt többször futott le, a táblázatban ezek átlaga szerepel. Az átlagolásból kivételre kerültek a kiugróan eltérő értékek. Ettől függetlenül további vizsgálatokra szorul az, hogy ezek az adatok mennyire hitelesek, nem-e véletlenül valamilyen implementációs eltérés miatt alakultak ki a fenti viszonyok.

Amennyiben mégis hitelesek az értékek, a következő megállapításokat lehet tenni. A H2 relációs adatbázis-kezelő kis adatmodellre jóval gyorsabb volt, mint a másik kettő, azonban az adatok mennyiségének növekedésével a teljesítménye nagyon leromlik. A neo4j ezzel szemben bár alapvetően lassabb, de ugyanolyan sebességgel képes kiszolgálni a nagyobb méretű és több párhuzamos kérést. Végül a HyperGraphDb közel olyan gyorsnak bizonyult kis modellre, mint a H2, azonban nagy adatmodellt nem volt képes kezelni. Ennek oka egyelőre ismeretlen.

4 Összefoglalás

A félév során megismerkedtem olyan modern adatkezelési eljárásokkal, amelyek sok tekintetben lehetnek a jövőben reális vetélytársai a mostani széles körben elterjedt relációs adatbázis-kezelésnek. Ezen felül elkészítettem egy tesztelő környezet első verzióját mellyel több nézőpontból is össze lehet hasonlítani ezen eszközök tranzakciós képességeit.

Mindezek mellett – mivel folytatni fogom a témát a következő félévben is – nagyon sok tennivaló és továbbfejlesztés vár rám. Első körben a generált modellt kell finomhangolni, hogy a valósághoz sokkal jobban közelítő viselkedést modellezzen, és többféle paraméter mentén lehessen a generálást finomhangolni. Ezen felül a mérendő mennyiségek számát is ki kell terjeszteni, illetve azok feldolgozására is nagyobb fokú automatizmust kell kidolgozni. Még extra teendőként, hogy újabb adatbázisok (pl.: Objectivity) kerüljenek be a teszteltek közé.

Az igazán érdekes részletek a hosszú távú tervekben rejlenek. Cél az, hogy nem csak egy, hanem többféle zárolási stratégiát valósítson meg a rendszer, melyeknek teljesítményét különböző típusú tranzakciókon kellene mérni. Ezután ki kellene dolgozni egy olyan automatizmust, mely képes az adott tranzakciók tulajdonságainak megfigyelésével eldönteni, hogy mely stratégiával fog valamilyen paraméter mentén hatékonyabban működni az adott adatbázis-kezelő.

5 Irodalomjegyzék

[1] “db4objects online dokumentáció,” *http://developer.db4o.com/Documentation.aspx*.
[2] “db4objects teljesítménytesztek,” *http://polepos.sourceforge.net/results/html/index.html*.
[3] “Perst wweboldal és dokumentáció,” *http://www.mcobject.com/*.
[4] “HyperGraphDB dokumentáció és bemutató,” *http://www.kobrix.com/hgdb.jsp*.
[5] “Neo4j weboldal,” *http://neo4j.org/*.
[6] “Social network,” *http://en.wikipedia.org/wiki/Social_network*.