Eötvös Loránd Tudományegyetem
Informatikai Kar
Programozási Nyelvek és Fordítóprogramok Tanszék

Statikus elemzést támogató hatékony adattárolási módszerek

Témavezetők:
Tóth Melinda, Bozó István
Tudományos főmunkatárs, Tudományos munkatárs

Szerző:
Harmaci Marcell
Programtervező informatikus, MSc

Budapest, 2024

Kivonat

Az Erlang az Ericsson által kifejlesztett, majd utólag nyílt forráskódúvá tett, általános érvényű, dinamikusan típusos funkcionális programozási nyelv. Kiemelt funkciói közé tartozik a jól skálázható és párhuzamosítható, kisebb folyamatokban történő problémamegoldás, a magas hibatűrés, valamint a magas rendelkezésre állás.

A statikus elemzés a szoftvertesztelés egy olyan megközelítése, amely a kód vizsgálatát annak végrehajtása nélkül végzi el. A statikus elemző szoftverek jellemzően már fejlesztés közben is alkalmazhatóak. A kódminőség és biztonság javítása mellett monetáris előnyökkel is járhat az ilyen eszközök használata, mivel a segítségükkel a hibák már a fejlesztési folyamat korai fázisában észlelhetővé válnak. A RefactorErl egy ilyen statikus elemző szoftver Erlang forráskódok elemzésére és sérülékenységeinek felderítésére amit, az Eötvös Loránd Tudományegyetem, Programozási Nyelvek és Fordítóprogramok Tanszékén fejlesztettek ki. Maga az eszköz is Erlang nyelven íródott és a programkód elemzése alatt összegyűjtött információ alapján egy szemantikus programgráfot épít fel. A programgráf tárolására és azon való különböző lekérdezések végrehajtására egy belső adatbázist használ. A RefactorErl már rendelkezik több bekapcsolható adatbáziskezelő réteggel, viszont a szoftver fejlesztőkörnyezetekbe történő integrációja során felmerült az igény egy "könyűsúlyú" adatbázis backend meglétére is.

A diplomamunkám célja egy olyan hatékony adattárolást biztosító megoldás megvalósítása volt, ami lehetővé teszi a RefactorErl által generált attribútumgráf memóriában való tárolását és azon lekérdezések gyors végrehajtását. Az in-memory adatbázisok célja a hagyományos diszk alapú megoldásoknál gyorsabb műveleti sebességek lehetővé tétele, amit az adattárolásra használt gyorsabb fizikai médium, a RAM használatával érnek el. A felmerülő feladat megoldására egy ilyen in-memory adatbázisréteget valósítottam meg egy Erlang nyelvben megtalálható kulcs-érték tároló használatának segítségével. Emellett a RefactorErl szemantikus programgráfjának eltárolására egy saját adatmodellt dolgoztam ki, ami alkalmas attribútumgráfok reprezentálására az adatbázisban. A dolgozatomban az elkészült adatbáziskezelő réteg teljesítményét összehasonlítom a RefactorErl többi adatrétegével, különböző méretű adathalmazok használata esetén.

Tartalomjegyzék

1. fejezet

Bevezetés

A statikus elemzés a programkód futtatás nélkül történő elemzése, mellyel felfedhetőek különféle hibák, biztonsági rések és teljesítményproblémák. Az elemzők jellemzően képesek különféle transzformációkat javasolni ezen problémák megoldására. A RefactorErl egy ilyen statikus elemző és transzformáló keretrendszer Erlang nyelven írt programokhoz, amely a megadott kódbázis statikus elemzése során egy szemantikus programgráfot épít fel. Az elemzés eredményét egy adatbázisban tárolja, amelyből egy szemantikus lekérdezőnyelv segítségével sokféle információ nyerhető ki. Az eszköz használata szempontjából nagyon fontos, hogy az elemzések eredményét mennyi ideig tart eltárolni és, hogy egy-egy lekérdezésre mennyi időn belül kapunk választ a szoftvertől. Emiatt már a jelenlegi RefactorErl is több adatbázisréteget tartalmaz, melyekkel az adatelérési időt próbálták optimalizálni a fejlesztők. Felmerült azonban az igény egy "könnyűsúlyú" RefactorErl adatréteg meglétére is, amely lehetővé tehetné a fejlesztői környezetekbe való hatékony integrációt is.

A dolgozatom célja ennek elérése gyanánt egy Erlang alapú gráftároló adatbázisréteg megvalósítása és a RefactorErl-be történő integrációja, amely képes a szoftver által generált szemantikus programgráfot teljes mértékben a memóriában tárolni és azon hatékonyan lekérdezéseket végrehajtani. Diplomamunkám részeként előzetes méréseket végeztem, melyek alapján az Erlang nyelv map adatstruktúráját választottam a belső tároló feladatának betöltésére. Az in-memory adatbázisréteg megvalósításához egy saját adatmodellt építettem fel, amely alkalmas a szemantikus programgráfban tárolt információk elraktározására. Végezetül pedig tesztek segítségével ellenőriztem az új adatbázis helyes működését és az adatréteg megfelelő integrációját a RefactorErl meglévő rendszereibe, majd összehasonlítottam az eredményt a meglévő adatbázisrétegek teljesítményével.

1.1. Erlang

Az Erlang [1, 2, 3] egy erősen típusos, dinamikus típusrendszerrel rendelkező, deklaratív funkcionális programozási nyelv. Eredetileg 1986-ban az Ericsson cégen belül telekommunikációs környezetben történő alkalmazásra fejlesztette ki Joe Armstrong, Robert Virding és Mike Wil-

liams. Soft real-time rendszerekre tervezték, ahol a válaszidők milliszekundumokban mérhetők. Először Prolog alapokon, így annak erős ráhatása volt az Erlang szintaxis kialakításában, majd 1998-ban nyílt forráskódú szoftverré vált. A nyelv a nevét Agner Krarup Erlang dán matematikus tiszteletére kapta, illetve az Ericsson Language rövidítéseként is szoktak hivatkozni rá. Jó skálázhatóság, beépített párhuzamosíthatóság és magas hibatűrés jellemzi, ami miatt alkalmas megbízható, magas rendelkezésre állású elosztott rendszerek megvalósítására. Előszeretettel használják például telekommunikációs, banki és pénzügyi, továbbá e-kereskedelemi területeken.

Az Erlang kód futtatásához bytekódra történő fordítás szükséges. Az így kapott kód a BEAM virtuális gépen futó Erlang Runtime System-en (ERTS) futtatható. Ez a mechanizmus biztosítja az Erlang platformfüggetlen működését, valamint ezek a rendszerek felelősek a párhuzamos folyamatok ütemezéséért és a dinamikus memóriakezelésért is, amely valósidejű szemétgyűjtéssel történik.

1.1.1. Megbízhatóság és hibatűrés

A nyelv megalkotásakor fontos szempont volt a mellékhatásmentesség elérése. Erlangban a változók értékük szerint lehetnek szabad vagy kötött változók. Az "egyszeri értékadás" tulajdonság érvényes rájuk, ami ezt jelenti, hogy ha egy szabad változónak értéket adunk, onnantól azt kötött válozónak tekintjük, aminek már nem adható újabb érték. Ennek a tulajdonságnak köszönhetően a nem állhat elő a versenyhelyzet hibaforrás (race condition), amikor egy változó értéke és típusa attól függ, hogy különböző folyamatok milyen sorrendben hajtódnak végre. A függvények definíciója mintaillesztéssel történik, ezért a függvények túlterhelhetőek. A különböző számú vagy típusú paraméterekkel történő futtatásra eltérő viselkedést valósítható meg.

A párhuzamos folyamatok az aktor modell alapján működnek, tehát nem rendelkeznek közös memóriaterülettel, így a köztük történő adatátvitel egyetlen módja a közvetlen üzenetküldés. Az üzenetek küldése aszinkron módon történik, így a küldő folyamat az üzenet elküldését követően folytatja a működését. Az üzenetet fogadó folyamatok várakozhatnak beérkező üzenetekre. A várakozásnak megadható maximális időlimit, melynek eltelte után lekezelhető ha nem érkezett üzenet. Az Erlang folyamatok emellett rendkívül könnyű súlyúak, azaz alacsony erőforrásigénnyel rendelkeznek. Emiatt alkalmazható a "let it crash" paradigma, melynek megfelelően az Erlang a hibák elkerülése helyett azok helyes lekezelésére helyezi a hangsúlyt. Ha egy folyamat valamilyen hiba folytán leáll, a rendszer a folyamatok monitozozásával észleli a leállást és egyszerűen indít helyette egy újabb példányt, azzal helyettesítve az eredetit.

1.1.2. Erlang/OTP (Open Telecom Platform)

Az Erlang/OTP [4, 5] összetett Erlang programok megvalósítására használatosalkalmas tervezési mintákat és segédkönyvtárak sokassága. (Pl.: `gen_server`) Többek között magába foglal elosztott adatbáziskezelőt (Erlang Term Storage, röviden ETS), interfészeket különböző programozási nyelvek Erlanggal történő kommunikációjára és hibakeresési eszközöket egyaránt.

1.2. Statikus elemzés

A statikus elemzés a programok forráskódjának vizsgálatát jelenti, annak futtatása nélkül. Ez egy white-box módszer, mivel a forráskód pontos ismerete szükséges hozzá. A elemzést általában valamilyen automatizmus keretében lefuttatott statikus elemző szoftverek végzik, így az integrálható CI/CD (folyamatos integráció / folyamatos szállítás) rendszerekbe is.

A statikus elemzés célja olyan problémák észlelése a programkódban, amik annak futása során hibákhoz vezethetnek, biztonsági kockázatot okozhatnak vagy egyszerűen nem felelnek meg az adott nyelv vagy vállalat által megkövetelt programozási konvencióknak. Ilyenek lehetnek például nemkívánatos mellékhatások, kódduplikáció, nem megfelelően elnevezett változók vagy elavult interfészek használata. Bizonyos problémák feltárása manuális kódellenőrzéssel is lehetséges volna, viszont a megadott szabályok és szabványok alapján ezt az elemző szoftverek sokkal gyorsabban és hatékonyabban képesek elvégezni. Sok elemző eszköz különböző fejlesztőkörnyezetekbe is integrálható, így a sérülékenységek már a fejlesztési fázisban észlelhetőek, valamint jellemzően ezek az eszközök refaktorálási javaslatokat is képesek tenni a feltárt problémák javítására. A szoftverfejlesztési folyamat során minél korábban kerülnek detektálásra a hibák, annál kisebb költséggel jár azok javítása, így sok esetben pénzügyileg is kedvező lehet ilyen eszközök vállalati környezetben történő alkalmazása.

Mindehhez szükség van a kódbázis feldolgozására, szintaktikai és szemantikai értelmezésére, hogy az elemzéshez ismertek legyenek a programot alkotó programkonstrukciók, a köztük fennálló kapcsolatok és jelentésük. A megszerzett információkat az elemzőszoftverek működéséhez szükséges valamilyen adatbázisban eltárolni. Ez óriási adatmennyiség tárolásával jár és értelemszerűen az adatbázis mérete a vizsgálat alatt álló kódbázis méretével arányosan nő. A kapcsolatok felállítása, azaz az adatbázis felépítése után az elemzés a megszerzett adatok olvasásával és értelmezésével történik. Így az elemzőszoftver által használt adatbázis kiválasztásakor fontos kritérium mind az írás, mind az olvasás műveleti sebessége egyaránt.

1.3. RefactorErl

A RefactorErl [6, 7, 8, 9], az Eötvös Loránd Tudományegyetemen fejlesztett statikus kódelemző és refaktoráló eszköz Erlang programokhoz. Maga a RefactorErl eszköz forráskódja is Erlang nyelven íródott. A nagy kódbázisokra fordítandó növekvő karbantartási munkálatok és költségek mennyiségét kívánja csökkenteni. A RefactorErl szoftver többek között lehetővé teszi a megadott programkód vizualizációját, mellékhatásainak és függőségeinek vizsgálatát, az adatáramlás és vezérlési folyam analízisét, a kódbázisban előforduló duplikációk és biztonsági kockázatok feltárását, továbbá hasonló viselkedésbeli tulajdonságokkal rendelkező kódrészletek klaszterekbe gyűjtését egyaránt. Mindemellett lehetőséget biztosít a kód jelentéstartó átalakítására 24 különböző refaktorálási metódus segítségével. Többféle adatbáziskezelő réteggel használható: Erlangos Mnesia [2], C++ gráf és Kyoto Cabinet [10] alapú implementációkkal. A szoftver használatára különféle interfészeken keresztül nyílik lehetőség:

- Interaktív Erlang shell (`ri`)

  Ez a komplex interfész az Erlang shellből az `ri` modul függvényhívásaival használható.

Rengeteg funkció elérhető a modulon keresztül. Végrehajthatunk adatbázis műveleteket, szemantikus lekérdezéseket futtathatunk, biztonsági mentéseket keszíthetünk, refaktorálási transzformációkat végezhetünk el és további interfészeket is elindíthatunk a segítségével.

- Szkriptelhető Erlang shell (`ris`)

  Nagyon hasonlít az interaktív `ri` modulhoz, viszont bizonyos eltérésekkel megkönnyíti a RefactorErl Erlang szkriptekben való felhasználását. Többek között például az `ris` hívások visszatérnek a saját eredményükkel, illetve a függvények paraméterei általánosabbak és tartalmazhatnak szemantikus lekérdezéseket is.

- Webes interfész

  Ez a RefactorErl legfelhasználóbarátabb interfésze, ami egy böngészőben használható grafikus felületen keresztül tesz elérhetővé rengeteg funkciót. A webes interfész az Erlang alapú Nitrogen keretrendszerre [11] és Yaws [12] webszerverre épült. Az interfész használatához szükséges a Yaws telepítése. A webszerver telepítése és a RefactorErl megfelelő konfigurációja után az interfész egyszerűen indítható, a parancssorból a `bin/referl -web2`, erlang shellből padig az `ri:start_web()` parancsok futtatásával.

- Linux parancssori interfész prototípus (`RefactorErl`)

  A RefactorErl-en belüli `module:fun(arg1, arg2)` függvényhívás a következő parancssori utasítás használatával hívható meg: `RefactorErl module fun arg1 arg2`. Az interfész használatát úgynevezett "szintaktikus cukorkák" egyszerűsítik a felhasználók számára. Az interaktív shell `ri` moduljának paraméter nélküli függvényhívasaikor a modulnév és a paraméterlista is elhagyható, például az adatbázist alaphelyzetve állítású `ri:reset()` parancs megfelelője a `RefactorErl reset`.

- Többféle fejlesztőkörnyezetbe beágyazva:
  - Emacs integráció
  - Vim plugin
  - Wx interfész
  - QT interfész
  - Eclipse plugin prototípus
  - Visual Studio Code plugin prototípus

1.4. A diplomamunka és az eredmények összefoglalása

A dolgozat további részében kitérek arra, hogy a RefactorErl hogyan kezeli a programkódról gyűjtött információt, majd részletesen bemutatom az in-memory tároló megvalósításának lépéseit, ismertetem az elkészült eredmény alkotóelemeit és azok integrációját a szoftverbe. Végül bemutatom, hogy milyen mérésekkel értékeltem ki az új adatréteget és összehasonlítom, hogy az hogyan viszonyul a korábbi megoldások által elért teljesítményhez.

A 2. fejezetben bemutatom, hogy a RefactorErl a feldolgozott programkódot hogyan képezi le a szemantikus programgráfjában, valamint, hogy milyen adatbáziskezelő megoldások léteznek már a szoftverben ezen információk tárolására. Arra is kitérek, hogy miért lehet hasznos egy tisztán

memóriában tároló adatréteg megvalósítása és milyen limitációkkal rendelkeznek az ilyen jellegű adatbáziskezelők.

Az in-memory adatbázis megvalósításához szükség volt egy adattároló struktúrára, ezért ismertetem, hogy milyen lehetőségeket kínál jelenleg az Erlang nyelv és, hogy milyen előzetes méréseket végeztem el, hogy ezek közül kiválasszam a célnak megfelelő adatszerkezetet. A 3. fejezetben ezen kívül részletes bemutatom a RefactorErl szemantikus programgráfjának sémáját, majd, hogy az alapján hogyan határoztam meg az új adatréteg által használt, attribútumgráfok tárolására alkalmas adatmodellt. Ez után bottom-up megközelítésben bemutatom az in-memory adatbáziskezelő komponens rétegmodelljét, valamint az azt felépítő modulok feladatait és működését. Ezen belül részletesebben kitérek a path lekérdezések megvalósítására, mivel ez az algoritmus biztosítja a szemantikus lekérdezések megvalósítását és adatelérését.

A 4. fejezetben egy kiértékelést adok az elkészült megoldásról és összehasonlítom azt a meglévő adatrétegek működésével. Ehhez ismertetem az elvégzett méréseket, melyek során Erlang alkalmazások elemzésével és az így keletkezett adatbázisokon futtatot szemantikus lekérdezések kiértékelésével gyűjtöttem adatokat.

Mindezeket követően egy irodalmi áttekintés olvasható az 5. fejezetben, ahol további adatbáziskezelők és statikus elemzők kerülnek bemutatásra, majd a dolgozatot a 6. fejezet zárja, ahol összefoglalom az általam elvégzett munkát és a dolgozat tartalmát.

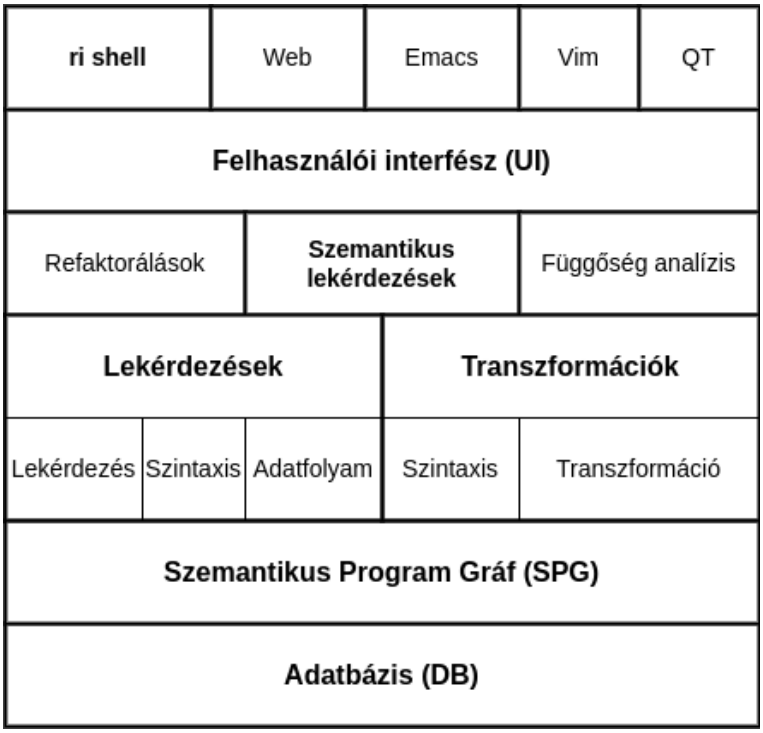
2. fejezet

Kapcsolódó háttéranyagok ismertetése

Diplomamunkám során azzal foglalkoztam, hogy a RefactorErl-hez, mint statikus elemzőhöz adjak egy olyan in-memory tárolási modellt, mellyel hatékony adatelérést biztosítok a szoftver által használt adatbázishoz. Ebben a fejetben a kapcsolódó munkákat mutatom be, amire építettem a dolgozat során. Ehhez ismertetem a RefactorErl jelenlegi programgráfját, valamint bemutatom hogyan vizsgáltam meg, milyen tulajdonságoknak kellene megfelelnie a tárolási modellnek és milyenek a szoftverben megtalálható jelenlegi megoldások.

2.1. A RefactorErl Szemnatikus Program Gráfja

A RefactorErl felépítése a szétválasztási elvnek (separation of concerns) megfelelően, felelősségi körök alapján szétválasztott rétegből épül fel. A szoftver által biztosított refaktorálás, szemantikus lekérdezések és függőség analízis funkciók egy réteget képeznek, melyhez sokféle különböző felhasználói interfészen keresztük biztosított a hozzáférés. A rendszer funkciói az üzleti logika megvalósítását tartalmazó rétegen keresztül tudják végrehajtani a lekérdezéseket és a transzformációkat, amelyekhez a szintaxis és adatfolyamok kezelésének ismerete is szükséges. Az üzleti logika a szemantikus programgráfban reprezentálja a feldolgozott programkódot. Ez egy olyan attribútumgráf, aminek csúcsai és azok tulajdonságai az Erlang kód alkotóelemeiről, az élei pedig a köztük lévő kapcsolatokaról hordoznak információt. A szemantikus programgráfot a szoftver a nagy adatmennyiség optimális kezelése és a perzisztencia megvalósítása érdekében egy adatbázisban tárolja.

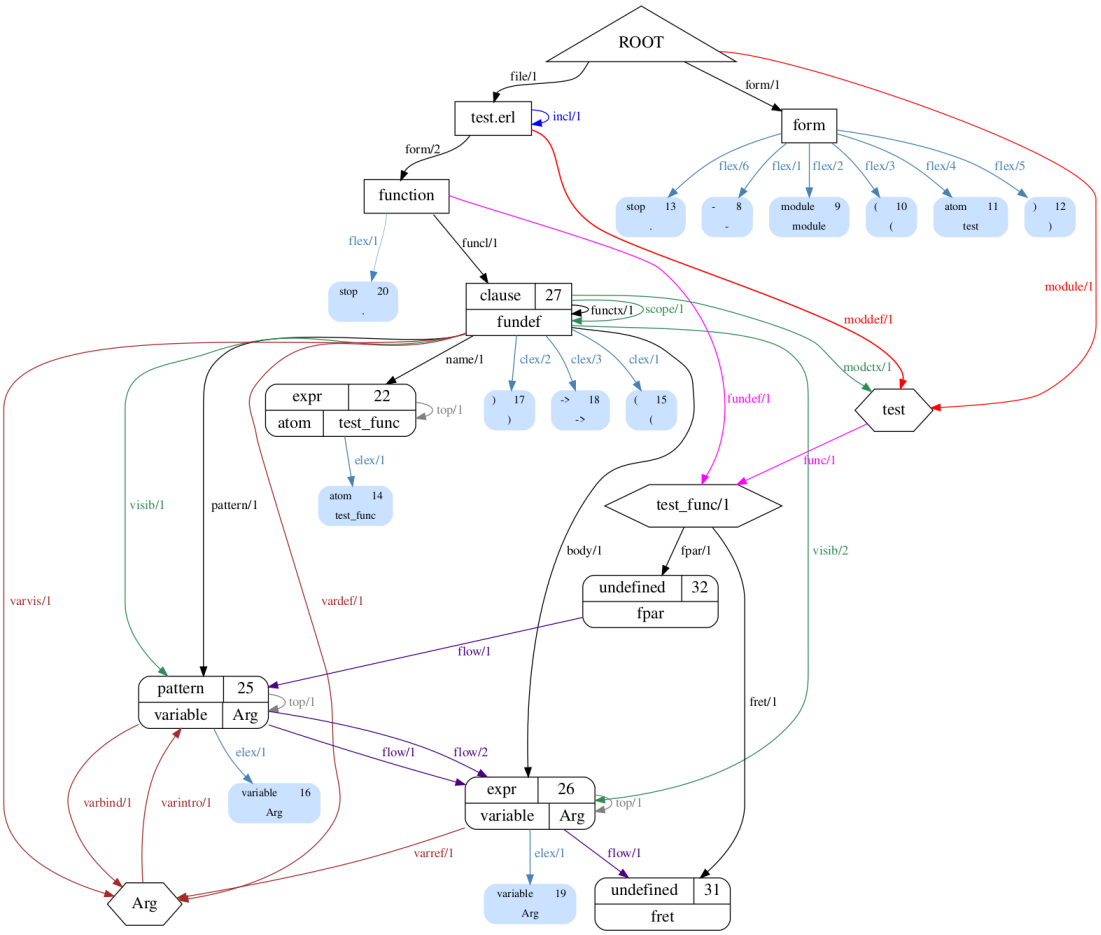


2.1. ábra. A RefactorErl-t felépítő komponensek rétegmodellje

A RefactorErl shell interfész használatakor az `ri:add/1` paranccsal tudunk az adatbázishoz fájlokat hozzáadni. Ilyenkor a hozzáadott fájlokat a szoftver feldolgozza és az elemzés alapján felépíti a szemantikus programgráfot, amit az adatbázisban tárol el. Az `ri:q/1,2,3,4` függvényekkel tudunk szemantikus lekérdezéseket futtatni a hozzáadott fájlokon. A RefactorErl ezeket a lekérdezéseket feldolgozza, hogy a programgráfból ki tudja nyerni a megfelelő információt, majd azt az adatbázishoz eljuttatva összegyűjti az adatokat.

A rendszer az adatbázishoz hozzáadott fájlokból alkotott gráfot a képes kirajzolni a szoftver az `ri:svg/0` parancs futtatásakor a Graphviz [13] gráfrajzoló eszköz használatával. A következő egyszerű test.erl fájból, ami csupán egy modul definíciót és 1 db egy paraméteres függvényt tartalmaz a 2.2. ábrán látható attribútumgráfot építi fel a RefactorErl.

```
-module(test).
test_func(Arg) -> Arg.
```



2.2. ábra. A RefactorErl által felépített absztrakt szintaxisfa egy egyszerű modul esetén

2.2. In-memory adatbáziskezelés

Ahogy a számítástechnológia világában oly sokszor, a hagyományos és in-memory adatbázisok közötti választás is a kompromisszumokról szól. Egy adatbáziskezelő rendszertől jellemzően elvárjuk az adatok biztonságos és hosszú távú tárolását és az azokhoz való gyors hozzáférés biztosítását egyaránt. Általános esetben viszont ezek közül az egyik tulajdonság optimalizálása csak a másik rovására valósítható meg. A megfelelő kompromisszum kiválasztása mindig az adott felhasználási eset által elvárt optimális tulajdonságoktól függ. Ebben a fejezetben az egyes megközelítések előnyeit és hátrányait mutatom be, valamit azt hogy szükség esetén, memóriatárolók használata mellett hogyan valósítható meg mégis a tartósság biztosítása.

2.2.1. In-memory és hagyományos adatbázisok összehasonlítása

A hagyományos adatbázisok az adatokat a számítógép háttértárján (jellemzően SSD vagy merevlemez, azaz HDD) tárolják el, ami az eszköz kikapcsolása és áramtalanítása után is megőrzi az adatokat. Az in-memory adatbázisok ezzel szemben a számítógépek memóriáját használják adattárolásra. Ennek előnye, hogy a memóriában tárolt adatokon végzett műveletek ideje töredéke a háttértárhoz hasonlítva, így az olvasási sebességen, az adatbázis áteresztőképességén nagyságrendi növekedés érhető el. A memórián történő adattárolás által elért teljesítménynövekedésnek viszont ára van, ugyanis a memória egy nem maradandó tároló, tehát a tartalmát a tápfeszültség megszűnésével elveszti.

Az in-memory adatbázisok gyakran célspecifikusak, azaz az adatbáziskezelő rendszer (DBMS) megalkotásakor az eltárolni kívánt adatok és azok felhasználása, az adatokon várható gyakori műveletek határozzák meg, hogy milyen megvalósításra van szükség a legjobb teljesítmény eléréséhez, az adott felhasználási esetben.

2.2.2. ACID tulajdonságok

Az ACID tulajdonságok [14] adatbáziskezelő rendszerek tranzakciókezelésére vonatkozó elvárások, amelyek az adatok integritását hivatottak biztosítani. A betűszót 4 tulajdonság együttese alkotja, atomiság (Atomicity), konzisztencia (Consistency), izoláció (Isolation), és tartósság (Durability).

Ezek jelentése a következő:

- Atomiság (Atomicity): A tranzakciók állhatnak több adatbázis műveletből, így előfordulhat, hogy valamelyik művelet végrehajtásakor hiba áll elő. Az atomiság tulajdonság garantálja az adatok helyességét ilyen esemény fennáltakor is. Egy tranzakciónak vagy minden művelete hatásos, vagy belőle semmi sem.
- Konzisztencia (Consistency): A tranzakción kívülről szemlélve, a tranzakció egyes műveleteinek végrehajtása által létrejött köztes állapotok nem elérhetők. A tranzakciók az adatbázist egyik konzisztens állapotból egy másikba viszik át.
- Izoláció (Isolation): Mivel egyszerre több tranzakció is írhatja és olvashatja ugyanazt az adatbázist, ezért az adathibák elkerülése végett fontos a párhuzamos műveletek elszigetelése. Az izoláció tulajdonság garantálja, hogy minden tranzakció úgy fut le (egy konkurrens környezetben is), mintha közben más tranzakció nem futna.
- Tartósság (Durability): A tartósság tranzakciók esetén azt jelenti, hogy ha egy tranzakció már sikeresen lefutott, akkor annak hatása "nem veszhet el". Ezt természetesen az adatbázis felhasználási területétől függően szigorúan kell megvalósítani, ami szélsőséges esetekben jelentős költségekkel is járhat. A tartósság tulajdonság teljesítéséhez általában elvárás, hogy a tranzakció hatása rögzítésre kerüljön valamilyen fizikai adattároló médiumon, ami tápellátás nélkül is megőrzi a rajta tárolt adatokat.

2.2.3. In-memory adatbázisok és a tartósság

Mivel az in-memory adatbázisok az adatokat a memóriában tárolják, így alapesetben azok nem felelnek meg az ACID tulajdonságok egyikének, a tartósságnak, viszont a másik három tulajdonsággal általában rendelkeznek. Ennek a hiányosságnak az orvosolására két megoldás létezik.

Az egyik, hogy a tranzakciók után az adatbázis állapotáról egy másolatot készítünk egy perzisztens adattárolóra, amely garantálja az adatok tartósságát. Ez megoldható pillanatképek készítésével, amelyek az egész adatbázis állapotát tartalmazzák, viszont a pillanatkép létrehozásának a magas műveletigénye miatt általában csak adott időközönként szokás ezt elvégezni. Mivel ez a köztes állapotok tartósságáról nem gondoskodik, így csak részlegesen teljesíti a tulajdonságot. Ez a megoldást a tranzakciók logolásával kiegészítve viszont már kielégíti a feltételt. A logok tartalmazzák a módosításokat, így a pillanatképekből kiindulva és a logolt tranzakciók újbóli alkalmazásával helyreállítható az adatbázis legfrissebb konzisztens állapota.

A másik megoldás, az adatbázist futtató számítógépben a szokásos DRAM (Dynamic Random-Access Memory) helyett valamilyen speciális memória hardverek használatával megvalósítható amik saját tápellátással rendelkeznek, így újraindítás után is folytatható az adatbázis működése a korábban mentett adatokkal. Ilyenek például az NVDIMM (non-volatile dual in-line memory module) ami a normál működéshez gyors, nem perzisztens memóriát használ viszont a tápellátás megszűnésekor kiírja annak tartalmát a saját perzisztens tárára, és az EEPROM (Electrically Erasable Programmable Read-Only Memory), ami egy speciális írható és törölhető ROM. Ezeknek a hardveres megoldásoknak sajnos megvan a hátulütőjük is a DRAM-hoz képest, például a megnövekedett költségek, lassabb írási és olvasási sebességek vagy rövidebb élettartam.

2.3. A RefactorErl adatrétegei

A RefactorErl statikus elemző szoftverbe már többféle adatbáziskezelő rendszert alkalmazó adatréteg is intergrálásra került. Ebben a fejezetben ezeket a lehetőségeket fogom bemutatni, illetve az újabb adatrétegek bevezetésére tett korábbi próbálkozásokat.

2.3.1. Mnesia

A Mnesia [15] egy az ETS (Erlang term storage) [16] és a DETS (disk Erlang term storage) [17] modulokon alapuló strukturált Erlang adatbáziskezelő modul. Az ETS és DETS Erlang kifejezések tárolására alkalmas modulok, amelyek az adatokat táblákba szervezve tárolják. Az ETS modulban minden tábláért egy Erlang folyamat felelős, ami a memóriában tárolja az adatokat, míg a DETS modul a számítógép háttértárát veszi igénybe az adatok rögzítésére. A Mnesia működéséhez mindkét modult felhasználva, ki tudja használni azok sajátosságait és egyesíteni az előnyeiket, így az ETS gyors keresési idejével és a DETS perzisztenciájával egyaránt rendelkezhet a megfelelő konfigurációval. Az adatbázis elfedi a tárolási mechanizmust, így annak használatakor a kliensnek nem szükséges tudnia, hogy a memória vagy a háttértár van használatban. Ennek ellenére ez a viselkedés táblák létrehozásakor Erlang node-onként külön konfigurálható ha szükséges a következő beállítási lehetőségekkel:

- `ram_copies` - Minden adat kizárólag ETS táblákban, a memórián kerül tárolásra, perzisztencia nélkül, viszont gyors adatelérési sebességgel. Ez az alapértelmezett működés.
- `disc_only_copies` - Az adatok a DETS modul használatával kerülnek tárolásra, így biztosított a perzisztencia, de lassabb lesz az olvasás mint ram_copies módban. Ilyenkor a DETS modul 2GB-os limitje korlátozza a tábla méretét.
- `disc_copies` - Az adatok tárolásra kerülnek a memóriában és a háttértáron egyaránt. Ilyenkor nem érvényes a DETS használatából adódó 2GB-os limit. A Mnesia pillanatképek és a tranzakciók logolásának együttesével szinkronizálja a perzisztens adatbázist a memória állapotával.

Az ETS-hez és DETS-hez hasonlóan a Mnesia is táblákban tárolja az adatokat. Az Erlang atomokkal nevesített táblák sorait Erlang rekordok alkotják. Az adatbázis táblák sémája futásidőben is módosítható a rendszer újraindítása nélkül. A Mnesia támogatja a tranzakciókezelést és támogatja az ACID tulajdonságokat, tehát konkurrens környezetben is jól alkalmazható, de ha az adott felhasználási esetben a gyors működés fontosabb mint a konzisztencia, akkor használható piszkos (dirty) módon is. A Mnesia mindemellett saját lekérdezőnyelvvel is rendelkezik. A lekérdezéseket írhatjuk magában az Erlang nyelvben, illetve a QLC (query list comprehension) modul segítségével, melyben list comprehension műveletek segítségével összetett lekérdezések kombinálhatók. Mindez nem azt jelenti, hogy alkalmas lenne egy teljes SQL alapú adatbáziskezelő rendszer helyettesítésére. Leginkább akkor alkalmazható jól, ha előre ismert az adatbázissal kommunikáló Erlang node-ok száma és nem túl nagy a táblákban tárolt adatmennyiség, mivel bizonyos módokban a táblák mérete korlátozva van.

A RefactorErl a következő paranccsal indítható el úgy, hogy a mnesia-s adatbázis réteget [18] használja:

```
bin/referl -db mnesia
```

Habár a `-db mnesia` kapcsoló nélkül is ugyanezt a működést tapasztalnánk, mivel a Mnesia az alapértelmezettként használt adatbázis. Ennek az adatbázis rétegnek a fejlesztése és karbantartása kapta a legtöbb figyelmet a támogatott megoldások közül, ezért ez is a legstabilabb módja a RefactorErl futtatásának. Az implementáció a táblákat `disc_copies` módban használja, annak korábban bemutatott előnyei miatt.

2.3.2. NIF

Az Erlang nyelv képes más programozási nyelveken megírt kódokkal is kommunikálni. Ez port driver-eken keresztül megoldható, azonban ez a kontextusváltások miatt magas teljesítménybeli költségekkel jár. Erre nyújtanak megoldást Erlangban a NIF-ek (Native Implemented Function) [19], amelyek lehetővé teszik hogy natív C-ben megvalósított függvényeket hívjunk meg Erlangból erőforrásigényes kontextusváltás nélkül. Az Erlang modulokban használt NIF-eket a modul betöltésekor kell betölteni. Minden natív függvénynek rendelkeznie kell egy Erlang megvalósítással is, amik lehetnek egyszerű hibaüzenetek, de akár teljes Erlang implementációk is arra az esetre ha nem sikerült volna betölteni az adott NIF-et. A NIF-ek használatával viszont

az Erlang veszít a stabilitásából, mert a natív kódban bekövetkező futásidejű hibák és eldobott kivételek az azt hívó Erlang node-ot is leállíthatják, ezért javasolt mellékhatásmentes, egyszerű számítások szinkron elvégzésére használni a natív implementációt.

A RefactorErl NIF-es adatrétege egy natív C++ könyvtárat használ a háttérben az adatok tárolására. Mivel a C++ maga is egy gyorsabb és alacsonyabb szintű programozási nyelv mint az Erlang, ezért ennek az adatrétegnek a használatával 8-10-szer gyorsabb működés érhető el a Mnesiához képest, viszont magas memóriahasználat figyelhető meg. A RefactorErl a következő paranccsal indítható el NIF-es adatbázis módban [20]:

```
bin/referl -db nif
```

2.3.3. Kyoto Cabinet

A Kyoto Cabinet [10] a Tokyo Cabinet utódja, amik magasabb teljesítményű DBM implementációk. A DBM (DataBase Manager) egy gyors NoSQL adatbáziskezelő és fájlformátum, amit az AT&T fejlesztett ki. A Kyoto Cabinet C++ nyelven íródott nyílt forráskódú szoftver, GNU Általános Nyilvános Licenc (GNU GPL) alatt. Egy egyszerű adatfájlt használ az adatok tárolására, amin a ZLIB Deflate algoritmussal történő tömörítést alkalmaz. Az adatokat kulcs-érték párokban tárolja változó hosszúságú bájtsorozatként. Mivel NoSQL adatbázisról beszélünk, ezért itt nem értelmezett az adatbázis táblák és adattípusok fogalma. A kulcsok és értékek is lehetnek bájtsorozatok vagy karakterstringek is, a rekordok tárolása pedig hash táblákban vagy B+ fákban történik. Minden rekordot egy az egész adatbázis szintjén egyedi kulcs azonosít. Az adatbázis támogatja a kulcs alapú keresés, törlés és rekord beszúrás műveleteket, illetve a kulcs alapú bejárást.

A hash adatbázis esetén minden művelet komplexitása $O(1)$, így a teljesítmény elméletileg az adatbázis méretétől függetlenül konstans, de gyakorlatban a memória méretétől, illetve a memória és a háttértár gyorsaságától függ. Az adatbázis teoretikus maximum mérete 8 exabájt. Természetesen mikor az adatbázis mérete meghaladja a rendelkezésre álló memóriaterületet, az operációs rendszer lapozással bevonja a háttértárat is és romlik az adatbázis teljesítménye. Ez könnyen bekövetkezhet akár a RefactorErl használatakor is, mert nagyobb kódbázis elemzésekor rengeteg adat keletkezik.

A B+ fa alapú adatbázis implementáció esetén az egy rekordot érintő műveletek komplexitása $O(\log n)$, tehát az adatbázis méretének növekedésével logaritmikusan csökken a teljesítménye. Bár a B+ fa használatakor a véletlenszerű adatelérés lassabb, mint a hash esetén, de a B+ fa adatbázis támogatja kulcsok alapján a szekvenciális olvasást, így ilyen esetben sokkal jobb teljesítménny tapasztalható.

A RefactorErl Kyoto Cabinetes adatbázis módban [21] csak Linux és OS X operációs rendszereken használható. A következő paranccsal indítható el ebben a módban a szoftver:

```
bin/referl -db kcmini
```

ahol a `kcmini` kapcsoló a "Kyoto Cabinet Minimal", minimális C implementáció használatát jelzi.

A Kyoto Cabinet többféle belső adatbázis implementációt támogat, melyek közül a RefactorErl a PolyDB-t használja. A PolyDB egy polimorf adatbázis, ami futhat fájl és memória módban is a konfigurációtól függően. A RefactorErl esetében fájlban tárolja a teljes adatbázist a háttértáron és csak az indexek vannak memóriában tárolva használat közben.

2.3.4. Korábbi próbálkozások

Riak

A Riak [22] egy elosztott, kulcs-érték NoSQL adatbázis. Erlangban íródott, 2017 óta nyílt forráskódú és több programozási nyelvhez is elérhető hozzá kliensoldani interfész, köztük a Java-hoz, C#-hoz, PHP-hoz és az Erlanghoz is. Az Erlang nyelv használatából eredően a nyelvhez hasonló előnyös tulajdonságokkal bír, mint például a skálázhatóság, megbízhatóság és magas hibatűrés. További előnye, hogy egy hash algoritmussal képes elosztottan, több szerveren úgynevezett vödrökben tárolni az adatbázist. Az adatbázis szerverrel egy REST API-n keresztül kommunikálva érhetők el a CRUD (Create, Read, Update, Delete) műveletek. A Mnesiához hasonlóan a Riak is támogatja az adatok tárolását a memóriában, a háttértáron vagy mindkét helyen is és konfigurálható hogy milyen szigorú konzisztencia kritériumokkal kívánjuk működtetni az adatbázist.

A RefactorErl-be integrált Riak adatréteg a Mnesiával ellentétben egy rekordban tárolja a szoftver által generált gráf csúcsait annak éleivel együtt, így jóval kevesebb rekordra van szükség a gráf eltárolásához. Az integrációt követő mérésekből viszont az derült ki, hogy nem sikerült vele nagyobb teljesítményt elérni mint a Mnesiával. Ezt feltehetően a RefactorErl és a Riak közötti rengeteg szinkronizációs művelet okozza, ami lelassítja a lekérdezéseket. Ennek következtében a RefactorErl kiadott verziói nem támogatják a Riak adatréteg használatát és nem is áll tervben továbbfejleszteni azt.

Neo4j

A Neo4j [23] egy modern és jól elterjedt, Java nyelven implementált natív gráfadatbázis kezelő rendszer, melyet 2000-ben kezdtek fejleszteni, majd 2010-ben készült el az 1.0 verzió. Az adatbázis futtatására lehetséges központilag a felhőben vagy akár saját számítógépünkön vagy szerverünkön is. A Neo4j közösségi (community) verziója ingyenesen használható, úgynevezett source-available szoftver és elérhető GitHub-on a forráskódja, de nem nevezhető nyílt forráskódú szoftvernek, de elérhető egy zárt, vállalati (enterprise) verziója is. A hagyományos SQL adatbázisokkal ellentétben táblák helyett gráfokban tárolja az adatokat. A gráfokban 3 fajta entitást különböztet meg: csúcsokat, éleket és attribútumokat, melyek csúcsokhoz és élekhez is hozzárendelhetők. Az adatbázis támogatja a tranzakciókezelést és rendelkezik az ACID tulajdonságokkal. A Neo4j Bloom szoftver segítségével vizuálisan böngészhető az eltárolt gráf és lekérdezések futtathatók azon. HTTP és Bolt protokollokon keresztül képesek más programnyelvek is kommunikálni az adatbázissal. Hivatalosan csak a Go, Java, JavaScript, .NET és Python támogatottak, de a közösségi driverekkel sok más nyelv, köztük az Erlang is interaktálhat a Neo4j-vel. A gráfadatbázison a lekérdezéseket a saját fejlesztésű Cypher [24] lekérdezőnyelvben fogalmazhatjuk meg a `MATCH`, `WHERE` és `RETURN` kulcsszavak és szemléletes mintaillesztések segítségével.

Annak ellenére, hogy a modern gráfadatbázisok jobb teljesítményt ígérnek, mint a hagyományos

relációs adatbázisok ez nem minden esetben van így. Kimutatható [25], hogy a relációs adatbáziskezelők analitikus lekérdezések esetén (például csúcsok fokszámának számításakor) továbbra is jobban teljesítenek a Neo4j-nél. Emellett a Neo4j használata alatt jóval magasabb memóriahasználat mérhető azonos adathalmazon, mint a MySQL relációs adatbáziskezelő használata mellett [26]. Olyan útkereső lekérdezések esetében viszont amik nem szűrnek specifikusan él típusokra a gráfban vagy a gráf kört tartalmazott, már a Neo4j ért el sokkal jobb teljesítményt mint a relációs adatbázisok.

A korábban elvégzett tesztek alapján viszont az figyelhető meg, hogy a RefactorErl-en belül egy kisebb adatbázisokon a Mnesiás adatréteg sokkal jobban teljesít mint a Neo4j-s adatréteg. Általánosan 10-20-szor lassabb a Neo4j, de bizonyos esetekben akár 40-szer is. Ez annak volt köszönhető, hogy az adatok integrált lekérdezése sok kis részletben történt, így a Neo4j-nek nem volt lehetősége jól kihasználni a gráfadatbázis előnyeit. Egy másik mérésben azt vizsgálták, hogy ugyanehhez a Mnesiás adatréteghez képest hogy viszonyul ha a Neo4j adatbázisból részletek helyett egyben, közvetlenül történik a lekérdezés. Ekkor már sikerült jelentősen jobb eredményeket elérnie a Neo4j-nek, viszont míg a Mnesia esetén komplex feldolgozás zajlott több absztrakciós rétegen keresztül, addig a Neo4j esetében a mérés nem így történt. A RefactorErl rendszerébe integrálva ez a megoldás is veszíthet a gyorsaságából, de lehetséges hogy így is meghaladná a Mnesia sebességét.

3. fejezet

Statikus elemzést támogató adattárolási módszerek

Ebben a fejezetben az in-memory adatbázis megvalósítását mutatom be. Ismertetem milyen struktúrákat kínál az Erlang nyelv a tároló feladatának betöltésére, illetve az ezeken végzett műveleti időkre és memóriahasználatra vonatkozó előzetes méréseket. Bemutatom a mérések alapján választott adatstruktúrát hasznosító megoldás adatmodelljét és annak belső struktúráját bottom-up megközelítésből. Emellett ismertetem a szemantikus programgráf felépítését, valamint hogy az abból való lekérdezéseket hogyan dolgozza fel és hajtja végre az adatréteg.

3.1. Megfelelő adatstruktúra kiválasztása

Azok az Erlang fájlok, amiken használni szeretnénk a RefactorErl funkcióit az Erlang shell `ri:add` függvényével adhatók hozzá a szoftver adatbázisához. Ezen parancs futtatásakor a RefactorErl az Erlang fájlokat leképezi egy többrétegű saját belső reprezentációjára, hogy abból kinyerhetők legyenek a funkciók működéshez szükséges lexikai, szintaktikus és szemantikus információk. Ennek a leképezésnek az eredménye a szemantikus programgráf, amely az Erlang kód alkotóelemeit reprezentáló csúcsokból és az azok között fennálló kapcsolatokat reprezentáló élekből áll. A RefactorErl szemantikus programgráfját a 3.2.1. fejezetben mutatom be részletesen. A forráskódok elemzésekor rengeteg adat keletkezik, amit az Erlang Mnesia 4.23-as verziójának elemzése is alátámaszt. Ez a könyvtár 28,156 sor kódból áll, míg az `ri:add` paranccsal a RefactorErl-hez hozzáadva egy 394,515 csúcsból és 890,089 élből álló gráf keletkezik, azaz egy több mint 1 millió rekordot tartalmazó adatbázis jön létre.

Az Erlangban implementált in-memory adatbázis megvalósításakor a kódelemzéskor keletkező nagy adatmennyiségből adódóan kulcsfontosságú szempont volt az adatbázis gyorsasága, ezért megvizsgáltam a nyelvben megtalálható adatstruktúrákat. Az összehasonlításukra előzetes teszt méréseket végeztem, hogy megállapítsam hogyan viszonyul egymáshoz azok teljesítménye. A műveleti sebességek vizsgálatakor a beszúrás, keresés és a törlés műveleket mértem meg, illetve

az adatstruktúra kiválasztása során figyelembe vettem a memóriahasználatot is. A mérések során törekedtem a RefactorErl felhasználási esetéhez hasonló felépítésű és méretű teszt adatokat felhasználni. Mivel a RefactorErl szemantikus programgráfjának csúcsai és élei változó hosszúságú Erlang rekordok formájában vannak reprezentálva amikor eljutnak az adatréteghez, ezért a mérések során is azokhoz hasonló rekordokkal dolgoztam. A méréseimet a teszt adathalmaz legenerálásával kezdtem, majd az összes megvizsgált adatstruktúra teljesítményét megegyező adatokkal hasonlítottam össze. A keresés és törlés mérése során szintén véletlenszerűen előre kiválasztott elemekkel dolgoztam az azonos mérési feltételek érdekében. A teszteket az adott teszteset által indokolt és a rendelkezésemre álló számítási kapacitás által megengedett alkalommal és nagyságú adathalmazzal futtattam, majd az így összegyűlt mérések átlagát vettem eredményül. A teszteredmények egy 10 magos, 16 szálas (Intel 12600K) processzorral, 48 GB 3200 MHz-es DDR4 memóriával és NVMe SSD tárhellyel ellátot rendszeren születtek.

3.1.1. List

Elsőként az egyik legegyszerűbb adatstruktúrát, a listákat vizsgáltam meg, hogy egy jó viszonyítási alapot kapjak. A listák különböző típusú adatokat képesek szekvenciálisan tárolni. Mint sok más nyelvben, Erlangban is szögletes zárójelekkel hozhatunk létre listákat a következőképpen: `["string", 123, [1, 2, 3], {type, atom}]`. A listák struktúrájukat tekintve rekurzív adatszerkezetek. Ezek legegyszerűbb esetben üresek (`[]`) vagy két részből állnak, a fejből, ami a lista első eleme és törzsből, ami a lista hátralevő elemeit tartalmazó lista (`[head|tail]`). Az új elem hozzáfűzésére két lehetőség van, melyek között nagy teljesítménykülönbség figyelhető meg a listák szerkezetéből adódóan. Amikor a lista elejére szúrjuk be az új elemet, akkor ki tudjuk használni az adatstruktúra rekurzív tulajdonságát, így újra felhasználható a meglévő listánk (`[uj_elem|lista]`). Amennyiben viszont a `++` operátor segítségével a lista végére kerül a új elem, akkor rekurzív módon vissza kell bontani a listát a legbelső elemére, azon elvégezni a beszúrást, majd visszaépíteni a meglővő elemeit az eredeti listának. Az utóbbi megoldásnak természetesen jóval magasabb műveletigénye van. Feltételezhetően az in-memory adatréteg megvalósítása során is célszerű lenne az optimális teljesítmény elérésére törekedni, ezért a teszteset implementálása során a gyorsabb beszúrást választottam. A keresés és törlés megvalósításakor lineáris keresést alkalmaztam.

3.1.2. Dictionary

Az Erlang többféle kulcs-érték tároló struktúrával is rendelkezik, amik alkalmasak lehetnek a RefactorErl gráfjának eltárolására. Ilyen egyszerűbb struktúrák a proplist, orddict, dictionary és a gb_tree [2]. A proplist egyszerű kulcs-érték párokat tartalmazó tuple-ök listája (`[{Key|Value}]`), emiatt ettől nem várható jobb teljesítmény mint magától a listáktól. Az orddict (ordered dictionary, azaz rendezett könyvtár) szintén párok listájában tárolja az adatokat, viszont a listát kulcsok szerint rendezve tartja nyilván. Ezzel kissé megnövelve a struktúra karbantartási költségét, de gyorsabb keresés érhető el. Az orddict-ek hátránya, hogy kisebb méretű adathalmazok kezelésére alkalmas. A 75 elemszámot meghaladó adathalmazok esetében gyorsabb működés érhető el a bonyolultabb adatstruktúrák használatával, mint a dictionary és a gb_tree. A dictionary az orddict-tel megegyező interfészt használ. A modul által használt belső struktúrára az Erlang nyelv dokumentációja szándékosan nem tér ki, feltehetően abból adódóan,

hogy az implementáció az interfésztől függetlenül változhat. A gb_tree (general balanced tree) egy általános kiegyensúlyozott fastruktúra. Az adatmódosító műveleteket jellenmzően a gb_tree gyorsabban kezeli mint a dictionary, míg az olvasásban a dictionary kerekedik felül. Mivel a RefactorErl funkcióinak használata közben többnyire a kódbázisról összegyűjtött információk olvasása szükséges, ezért az előzetes tesztekhez a dictionary vizsgálatát választottam. A mérések során a `dict:store/3`, `dict:find/2` és `dict:erase/3` műveleteket használtam.

3.1.3. Array

Az array adatstruktúrában egész számokat használhatunk kulcsként a kulcs-érték párok tárolására. Az indexelés nullától kezdődik, az array mérete pedig a létrehozástól függően lehet előre rögzített vagy szükség esetén automatikusan növekvő méretű is. Az Erlang array-ek nem követelik meg, hogy szekvenciálisan egymást követő kulcsok azonosítsák az eltárolt rekordokat. A köztes kihasználatlan területek reprezentálására megadható egy alapértelmezett érték, ami alapesetben az `undefined` atom. Mindez az `array:new` metódussal konfigurálható. Az array adatstruktúrán belül egy 5 elemű Erlang tuple felelős az adatok tárolására. Az első 4 eleme az array konfigurációját tartalmazza, illetve az 5. elemben találhatóak az adatok, amit szintén egy tuple reprezentál. Ennek köszönhető az array gyors működése. Továbbá a következő példát megvizsgálva látható, hogy az egymást követő üres elemek tárolása is optimalizált azok száma alapján egy egész számmá történő egyszerűsítéssel.

```
> A = array:new(100).
{array,100,0,undefined,100}

> A0 = array:set(0, zero, A).
> A3 = array:set(3, three, A0).
> A99 = array:set(99, ninety_nine, A3).
{array,100,0,undefined,
   {{zero,undefined,undefined,three,undefined,undefined,
          undefined,undefined,undefined,undefined},
    10,10,10,10,10,10,10,10,
    {undefined,undefined,undefined,undefined,undefined,
               undefined,undefined,undefined,undefined,ninety_nine},
    10}}
```

A példában létrehoztam egy 100 elemű array-t, aminek beállítottam a 0, 3 és 99-es indexű elemeit. Létrehozás után egy teljesen üres struktúrát kapunk, így a kimeneten látható, hogy csak az elemszám reprezentálja az üres adathalmazt. A beszúrások utáni eredményen látható, hogy az első és utolsó 10 elemet 1-1 tuple-ként tárolja az array, amikben az üres helyeken az alapértelmezett `undefined` atom található. A 10-es és 89-es indexek között nem találhatóak értékek, ezért az a 80 elem ebben az esetben tízesével összevonva lett tárolva.

A méréseim megvalósításában az `array:new/0` függvényt használtam az array létrehozására. Ez egy kezdetben 10 elemű, de automatikusan bővülő tárolót hoz létre. Mivel az in-memory adatbázis esetén nem ismert előre a tárolni kívánt adatmennyiség, ezért ez egy elengedhetetlen követelmény. Kereséshez az `array:get/2`, törléshez pedig az `array:reset/2` műveleteket

alkalmaztam.

3.1.4. Map

Az Erlangban megtalálható adatstruktúrák között egy újabb megoldásnak tekinthető a map. Az OTP 17.0 verziójában lett bevezetve 2014-ben, mint kísérleti funkció. Igény volt egy dictionary-nél és gb_tree-nél gyorsabb kulcs-érték tárolóra, amivel egyszerűbben lehet meglévő adatokat módosítani, mint az Erlang rekordok esetében (amik valójában a háttérben Erlang tuple-ök), így ezzel a motivációval kerültek bevezetésre a nyelvbe a map-ek.

A maps modul a dict modulhoz hasonló műveleteket biztosít az adatstruktúrák kezelésére, de az Erlang nyelv felhasználói számára a map-ek egyik fontos előnye annak a nyelvbe integrált könnyű használata. A maps modul bizonyos függvényeire egyszerűsített beépített szintaxis is elérhető. A `#{}` kifejezéssel új map hozható létre a `maps:new()` függvény használata nélkül. Az ily módon létrehozott map-ek elemei előre megadhatók (`#{ Key => Value }`), sőt kulcsként és értékként változók is használhatók. Ez ugyanígy igaz a beszúrásra is a következő szintaxissal `Map#{ NewKey => NewValue }`, illetve a map-ben már megtalálható kulcshoz tartozó érték módosítására is következőképpen `Map#{ Key := NewValue }`, ahol a `Map` egy már létező map. Az elemek törlésére használatos `maps:remove(Key, Map)` függvénynek nincs megfelelője.

létrehozás	`maps:new()`	`#{}`, `#{ Key => Value }`
beszúrás	`maps:put(NewKey, NewValue, Map)`	`Map#{ NewKey => NewValue }`
felülírás	`maps:put(Key, NewValue, Map)`	`Map#{ Key := NewValue }`
törlés	`maps:remove(Key, Map)`	nincs megfelelője

3.1. táblázat. A maps modul függvényeinek egyszerűsített szintaxisa.

A map-ek mintaillesztésekben is használhatók, ahol a `:=` operátor áll a kulcsok és értékek között. Például a `#{ K := V } = M` kifejezés egy olyan mintaillesztés, ahol `M` egy létező map, `K` egy létező kulcs `M`-en belül (ez lehet egy kötött változó is), `V` pedig a `K`-hoz tartozó érték. Amennyiben `V` egy nem kötött változó, akkor a mintaillesztés köti és értékül kapja a `K`-hoz tartozó értéket. Ezzel a módszerrel egyszerre több kulcs-érték párra is készíthetünk mintaillesztéseket.

A listák generálásához használatos list comprehension szintaxisnak is létezik hasonló, map-eken értelmezett párja, a map comprehension. A `#{K => 2*K || K <- [1,2,3]}` map generátor eredménye a következő map: `#{1 => 2, 2 => 4, 3 => 6}`.

A map adatstruktúra belső működése eltérő kis (maximum 32 elemszámú) és nagy (32 elemszámot meghaladó) map-ek esetén. A kis map-ek (flat map), erlang implementációban a kulcsok egy tuple-ben vannak eltárolva, így a hozzájuk tartozó értékek frissítésekor az így keletkező map ugyanazokat a kulcsokat tartalmazza. A kis map-ek szintén újra tudja használni a kulcsokat egymás között, ha a a programunkban egy függvénnyel hozzuk őket létre azonos kulcsokkal, így kisebb memóriahasználat érhető el. A nagy elemszámú map-ek implementációja egy HAMT (Hash Array Mapped Trie) struktúrában tárolja az adatokat [4], ami nagyobb memóriahasználatot eredményez, mivel nem lehet olyan jól újrahasználni a létező struktúrát a frissétesek során, viszont továbbra is hatékonyan kereshető és módosítható a map tartalma annak méretétől függetlenül.

A RefactorErl használata során 32-nél biztosan több adatot szükséges eltárolni az elemzés alatt álló forráskódról és a teszt adathalmaz mérete is minden esetben meghaladja ezt a mennyiséget, ezért a nagy map-ek tulajdonságai érvényesülnek a tesztelés során. A forráskódok feldolgozása során az adatréteg felé egyesével érkeznek az adatbázist érintő kérések, emiatt a tesztek megvalósításában is egyesévek szúrtam be az adatokat a `Map#{Id => Data}` szintaxissal. A keresések során az maps modul `maps:get(Id, Map, error)` függvényhívását használtam, aminek hibakezelés gyanánt alapértelmezett értéket is beállítottam. Alapértelmezés használatával lassabb keresési eredmények érhetőek el, mint nélküle, de a meg nem talált rekordokat az adatrétegnek is tudnia kell kezelnie használat közben és jeleznie a hívó felé. Adatok törlésére a `maps:remove(Id, Map)` műveletet alkalmaztam.

3.1.5. ETS táblák

Az ETS táblák alkalmasak nagy mennyiségű Erlang adathalmazok tárolására. Az adatokat táblákba szervezve tárolják, ahol a rekordokat Erlang tuple-ök reprezentálják. Az ETS jelentése Erlang Term Storage, tehát bármilyen Erlang kifejezést lehetőségünk van eltárolni, amit egy tuple-ben is el tudunk. A tuple-ök egyik eleme elsődleges kulcsként fog viselkedni, ami azonosítja az adott sort, és ez alapján sorbarendezhetőek az adatok. Az adatok tárolásáért egy dedikált Erlang folyamat felelős. Az adattábla addig elérhető, amíg az azért felelős process is aktív. Az ETS táblák tartalma kiírható a fájlrendszerbe dump fájlokba, illetve ezekből a fájlokból újból létrehozhatók a táblák az ets modul `tab2file` és `file2tab` függvényeivel, így az adatstruktúra lehetőséget biztosít a perzisztencia megvalósítására. Az ETS megvalósítása során konstans adatelérési idő elérése volt a cél, azonban valós használat során inkább az adatmennyiséggel logaritmikusan arányos olvasási sebesség érhető el. Az Erlang adattárolási mechanizmusok között továbbra is gyorsnak tekinthető. Az adattábla létrehozásakor megadható többféle konfigurációs beállítás. A tábla típusára 4 beállítási lehetőségünk van:

- `set`: Az ilyen típusú táblák kulcsai egyedi azonosítói a tábla rekordjainak. Alkalmas általános kulcs-érték tárolók megvalósítására, konstans adateléséri sebességgel. Ez az alapértelmezett táblatípus.
- `ordered_set`: Ebbe a típusba tartozó táblák továbbra is egyedi kulcsokkal rendelkeznek, viszont a rekordokat a kulcsok alapján rendezve tárolják. Ez a tulajdonság olyan esetekben hasznosítható, amikor a rekordok olvasásánál gyakran a kulcsok szakaszába tartozó minden elemére szükség van. Ilyen esetekben gyorsabbak mint a `set` típusú táblák, viszont az egyszeri adatelérések esetén történő random olvasás komplexitása $O(\log N)$-re növekszik, ahol N a tárolt adatok mennyisége.
- `bag`: Az ilyen táblákban egy kulcs többször is előfordulhat, amennyiben a rekordot alkotó tuple többi tagja között van eltérés.
- `duplicate_bag`: Ez a típus, a `bag` típusnál is megengedőbb. Ilyen típusú táblákban megengedett, hogy teljesen azonos rekordok többször is előforduljanak az adathalmazban.

A tesztek megvalósításakor a set tábla típust használtam, mivel van a RefactorErl gráf csúcsai és élei rendelkeznek egyedi azonosítóval, amit kulcsként lehet használni. Várhatóan nem származna előny az ordered_set gyors szakaszos adateléréséből, mivel a gráfot annak saját sztruktúrája

alapján szükséges bejárni az információ kinyeréséhez. A tároló feltöltésekor a beszúrásokat a 3.1.4. fejezetben leírtak okok miatt itt is egyesével végeztem el az `ets:insert/2` művelettel. Keresésre a `ets:lookup/2`, törlésre pedig a `ets:delete/2` függvényeket használtam.

3.1.6. Digraph

A digraph a directed graph rövitítése. Ez az adatstruktúra alkalmas irányított gráfok tárolására. Megengedett benne több él létrehozása két csúcs között, üres csúcsok létrehozása, továbbá a csúcsokat és éleket címkékkel lehet annotálni további infromáció tárolására. Két Erlang modul kapcsolódik a megvalósításához, a digraph és digraph_utils. A digraph modul használható az adatstruktúra felépítésére és manipulálására, a digraph_utils modul pedig hasznos DFS-en (Depth-First Search) alapuló algoritmusokhoz biztosít segédfüggvényeket. A digraph az adatok tárolásához ETS táblákat használ, ami miatt hasonló teljesítmény várható ettől az adatstrutúrától is. A mérések megvalósítása során a `digraph:new/0`, `digraph:add_vertex/3`, `digraph:vertex/2` és `digraph:del_vertex/2` függvényeket használtam.

3.1.7. Műveletidők mérése

Minden előzőleg bemutatott adatstruktúra műveleti sebességét megmértem, hogy meghatározzam melyik volna a legalkalmasabb az adatok memóriában való tárolására. A teszt adatok generálásához meg kellett határoznom, hogy milyen teszt adatokon történjenek a mérések. Ehhez megvizsgálzam a RefactorErl szemantikus programgráfjának sémáját, amiben 1-től 9-it terjedő elemszámú Erlang rekordok formájában van reprezentálva a programkódról gyűjtött információ. A rekordok jellemzően atomokat, számokat és karaktersorozatokat tartalmaznak. Ebből adódóan a mérések egyszerűsége kedvéért és, hogy átlagos méretű adatokkal mérjek, egy 5 elemű rekordot definiáltam, ami képes volna tárolni egy egyedi azonosítót, a rekord osztályát, 2 atomot és egy véletlen számot a következőképpen:

```
-record(data, {id, class, atom1, atom2, num}).
```

Az adatgenerálást követően létrejött adathalmaz birtokában többször futtattam a méréseket a vizsgált adatstruktúrákon, minden esetben ugyanazokkal az adatokal. Az összes teszteset a következő felépítést és lépéseket követte:

1. Tároló feltöltése az összes előkészített adattal.
2. Egy véletlen kulcs alapján egy rekord megkeresése.
3. A véletlen kiválasztott rekord törlése.

Az eltelt idő méréséhez `timer:tc` függvényhívásokat használtam. A timer modul az általam használt rendszeren alapértelmezett módon mikroszekundumban mért időkkel dolgozik, amennyiben nem adunk meg várt mértékegységet, így a következő mérési adatok is mikroszekundumban értendők.

Az első méréshez egy 1 millió rekordból álló mintát vettem alapul, mert ehhez hasonló méretű adatmennyiséget generált az Erlang Mnesia könyvtár 4.23-as verziójának RefactorErl-lel történő elemzése is (lásd 3.1. fejezet). A 3.2. táblázaton megtalálhatóak a mérés eredményei. Ez alapján látható, hogy a dictionary adatstruktúra adatokkal feltöltése nagyságrendekkel több időt igényel, mint amire a többi struktúrának szüksége van. Amiatt a további műveleti sebesség méréseket a dictionary-vel már nem végeztem el, hiszen biztosan van annál alkalmasabb adastruktúra az in-memory tároló feladatának betöltésére. Emellett a mérések futtatását is lassította volna a szükségtelen adatgyűjtés.

	Műveletidő (μs)		
Adatstruktúra	Beszúrás	Olvasás	Törlés
list	22 841	89	476
dict	29 225 960	2	31
array	328 473	1	0
map	306 536	1	1
ets	297 023	1	1
digraph	306 613	1	2

3.2. táblázat. 1. mérés: műveletidők 1 millió adatrekorddal, 1 mintából

A második mérést, az első méréshez hasonlóan 1 millió rekorddal futtattam, azzal a különbséggel, hogy immár 100 mintát vettem és a mérések átlagát vettem eredményül, egyetlen minta helyett. Ezzel csökkenthető az operációs rendszeren futó egyéb folyamatok hatására keletkező kiugró mérési adatok hatása a végeredményre nézve. Az eredmények a 3.3. táblázaton olvashatók.

	Műveletidő (μs)		
Adatstruktúra	Beszúrás	Olvasás	Törlés
list	6 704.17	1 704.09	10 104.53
array	240 896.96	1.02	0.13
map	500 190.29	1.04	1.30
ets	297 820.80	0.94	0.47
digraph	311 190.20	0.96	1.18

3.3. táblázat. 2. mérés: műveletidők 1 millió adatrekorddal, 100 minta átlagából

A második mérés eredményein látható, hogy a list adatstruktúra esetén a beszúrás szembetűnően alacsony, az olvasás és törlés pedig kimagaslóan sok időt vett igénybe a többi méréshez képest. Ezek az eredmények a mérések megvalósításakor is várhatóak voltak, hiszen a lista felépítésekor az elejére szúrtam be az új elemeket, ami a tároló rekurzív struktúráját kihasználva nagyon gyors művelet. Az olvasáshoz és törléshez viszont először meg kell határozni a keresett adatrekord helyét, amire a hagyományos lista esetében egyedül a lineáris keresés áll a rendelkezésünkre. További kiemelkedő érték az array esetén a törlés időtartama, ami valószínűleg az array felépítésének, az egymásba ágyazott tuple-öknek köszönhető. Array-ből való törléskor elegendő egy kisebb méretű belső tuple elemét visszaállítani az array üres értékre, így a módosítás előtti struktúra nagy része újrahasznosítható.

A 3.3. táblázat további mérési eredményein látható, hogy az adott műveletekhez szükséges

műveletidők közel azonos, ugyanolyan nagyságrendű értékeket vesznek fel. A tároló 1 millió rekorddal való feltöltésében az array és az ets tábla bizonyult a leggyorsabbnak, amit a digraph és a map követett. A digraph az előzetes várakozásoknak megfelelően mindenben kicsit lassabb volt, mint az ets tábla. Mivel a digraph ets táblákat használ az adatok tárolására, így valamennyi extra költséggel jár az ets táblák önmagában való használatához képest.

Mivel ilyen közeli teljesítményt ért el a legtöbb megvizsgált adatstruktúra, ezért azok skálázhatóságának vizsgálatához egy harmadik tesztet is elvégeztem a korábbinál több, 10 millió adatrekorddal. A második méréssel megegyezően a mérési eredményeket ismét 100 db minta átlagából határoztam meg. Az eredmények a 3.4. táblázaton olvashatók.

	Műveletidő (μs)		
Adatstruktúra	Beszúrás	Olvasás	Törlés
list	70 758.54	17 929.69	104 996.42
array	2 877 262.21	1.45	0.15
map	7 156 210.20	1.08	1.27
ets	4 439 614.51	1.34	1.16
digraph	4 579 545.26	1.40	1.64

3.4. táblázat. 3. mérés: műveletidők 10 millió adatrekorddal, 100 minta átlagából

A 2. és 3. mérés eredményeit a 3.5. táblázatban hasonlítom össze. Ezen a rekordok számának megtízszerezése után bekövetkező műveleti idők változása olvasható, azaz hogy a 3. mérés eredményei hányszorosai a 2. mérésben kapott adatoknak.

	Változás a 2. és 3. mérés között		
Adatstruktúra	Beszúrás	Olvasás	Törlés
list	10.55	10.52	10.39
array	11.94	1.42	1.15
map	14.31	1.04	0.98
ets	14.91	1.43	2.47
digraph	14.72	1.46	1.39

3.5. táblázat. 3. mérés műveletideinek változása a 2. méréshez viszonyítva

A beszúrási sebességben 10 millió rekord esetén is a második méréssel azonos sorrendben teljesítettek a tárolók, illetve minden esetben legalább tízszeresére nőtt a művelethez szükséges idő. Az olvasási teljesítményen viszont látható, hogy a map jobban skálázódik, mint a többi adastruktúra. 10-szeres adatmennyiség esetén is, 1.04-szeresére, csupán 0.04 mikroszekundummal nőtt az átlagos olvasási idő a map esetében. A map átlagos törlési időejében pedig 0.03 μs csökkenést mértem, ami vélhetőleg mérési hiba miatt adódott, de szintén jól szemlélteti, hogy közel azonos maradt a map teljesítménye az adatmennyiség növelése ellenére is.

Műveletidő méréseinek kiértékelése

Összességében az előzetes mérések alapján kiderült, hogy a listák esetén a lineáris bejárás nagyon lelassítja a keresést igénylő műveleteket, az ets tábláknál és a digraph-nál pedig ebben az adott felhasználási esetben hasonlóan gyorsnak, viszont jobban skálázhatónak bizonyult az array használata. A map lassabban végezte el a beszúrásokat, mint az array, viszont gyors keresést biztosít nagy adatmennyiségek esetén is, amit fontosabb tényezőnek tekintettem, mivel a RefactorErl kódelemzésekhez az absztrakt szintaxisfa gyors bejárása szükséges. Ezek alapján tehát a map és az array tűntek a legalkalmasabb adatstruktúrának az in-memory tároló megvalósítására.

Az Erlang dokumentáció alapján [27] általában a map-ek bizonyulnak a leggyorsabb kulcs-érték tárolónak amennyiben teljesül a következő 3 feltétel:

1. A kulcsok fordítási időben még nem ismert értékű változók
2. Nem tudni előre, hogy mennyi elem tárolására van szükség
3. Olvasáskor általában csak egy elemet kell megkeresni

Ez alól kivételt képezhet, ha kulcsokként kizárólag pozitív egészeket használunk, amikor az arrayek is jól teljesítenek. A méréseim során egyszerűsített adatokkal és pozitív egész kulcsokkal dolgoztam, de valójában a RefactorErl programgráfjának éleit azok kiindulási csúcsa, illetve az él típusa és indexe azonosítja. Emiatt és az előzetes mérések során kapott ígéretes eredmények miatt a map adatstruktúrát választottam az adatbázis réteg megvalósítására.

3.1.8. Memóriahasználat mérése

Ebben a fejezetben a különböző adatstruktúrák memóriahasználatát mutatom be. Ennek a pontos mérése sokkal bonyolultabb problémának bizonyult, mint a előzőekben a műveletidők meghatározása. Az Erlang dinamikusan kezeli a memóriát és a programok futása alatt a beépített szemétgyűjtő algoritmus felszabadítja azokat a memóriaterületeket, amik már nincsenek használatban.

Az első próbálkozásom során az `erlang:memory/1` függvényhívás segítségével próbáltam méréseket készíteni egy futó Erlang shell környezeten belülről. Az `erlang:memory/1` az Erlang emulátor által dinamikusan foglalt memóriaterületekről ad információt. A függvény egy memória típus (`memory_type`) paramétert vár a következő lehetőségek közül, amivel meghatározhatjuk, hogy milyen memóriaterületről szeretnénk információt lekérdezni:

```
memory_type() = total | processes | processes_used | system |
                atom | atom_used | binary | code | ets
```

Ezek közül a következő négy típust érdemes megemlíteni, amellyekkel a méréseket is végeztem:

- `total`: A teljes lefoglalt memóriaterület, ami magába foglalja az Erlang rendszer és a folyamatok memóriaterületét.
  $total = system + processes$
- `system`: Az Erlang emulátor számára lefoglalt memória, ami nem a folyamatok számára van allokálva.
  $system = atom + binary + code + ets + OtherSystem$
- `processes`: Az összes Erlang folyamat számára foglalt memória.
  $processes = processes\_used + ProcessesNotUsed$
- `processes_used`: Az épp használatban levő memóriaterület, Erlang folyamatok számára foglalva.

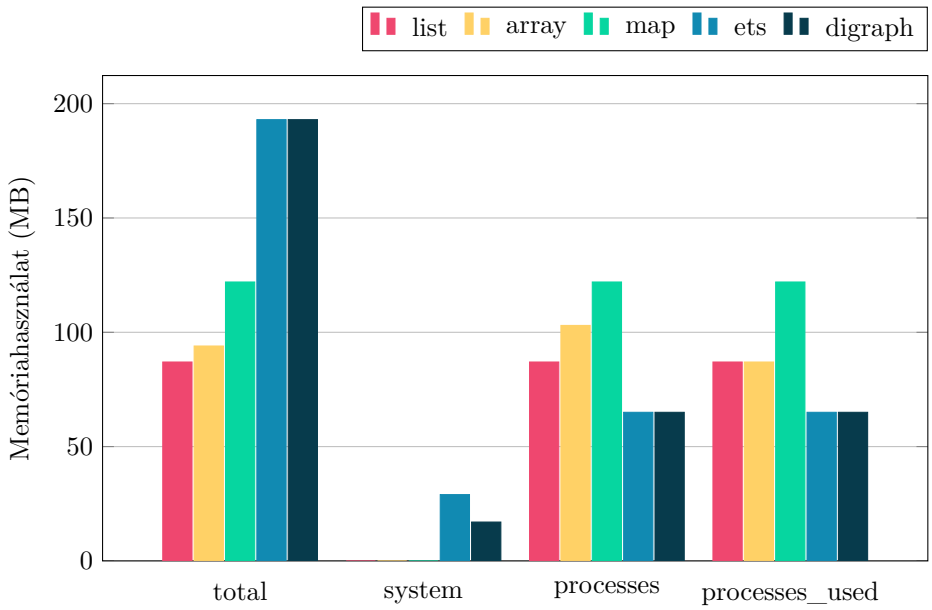
A méréseket a következő mintára végeztem el minden adatstruktúra esetén:

```
measure_data_type_memory(Parent, Data) ->
    MemBefore = measure_memory(),
    ...
    // adatstruktúra feltöltése
    ...
    MemAfter = measure_memory(),
    Mem = MemAfter - MemBefore,
    if
        Mem < 0 -> mem_func(Parent, Data);
        true    -> Parent ! {self(), Mem}
    end.
```

Ahol a `measure_memory()` hívás az adott mérésnek megfelően beállított paraméterrel hívta meg az `erlang:memory/1` függvényt. Minden esetben megmértem a memóriahasználatot az adatstruktúra feltöltése előtt és után, majd kiszámoltam az így keletkezett különbséget. Az eredménnyel való visszatérés előtt még beiktattam egy ellenőrzést, hogy nullánál nagyobb értéket mértem-e eredménynek, ezzel kiszűrve azokat a méréseket, amik alatt a szemétgyűjtő a méréstől függetlenül nagyobb memóriaterületet szabadított fel a háttérben. Természetesen így is előfordulhat, hogy vannak olyan memóriaterületek, amik a mérés közben szabadultak fel. Ennek a hatásnak a minimalizálására is a műveleti sebességek méréséhez hasonlóan több mérést végeztem egymás után, aminek az átlagát jegyeztem fel végső eredményként. A mérések izolációja érdekében a fenti mintára implementált függvényeket egy külön erre a célra létrehozott Erlang folyamaton belül futtattam. Az `erlang:memory` függvény bájtban ad információt a memóriahasználatról. Az bájtban mért pontos eredmények a 3.6. táblázaton olvashatók, továbbá az eredményeket megabájtra átváltva és egészre kerekítve a következő oszlopdiagramon hasonlíthatjuk össze (3.1. ábra).

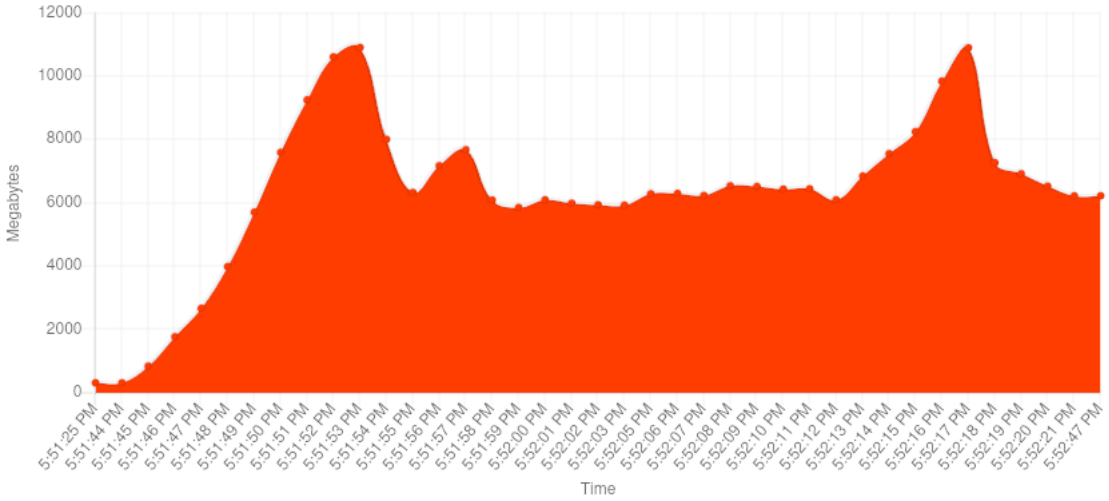
	erlang:memory/1 paraméter			
Adatstruktúra	total	system	processes	processes_used
list	86862638	43	86860665	86861794
array	94231851	55	103087289	87359235
map	121978029	105	121921104	121929696
ets	193016327	29014080	64990762	64988878
digraph	193026051	17273623	64993172	64991322

3.6. táblázat. 4. mérés: Adatstruktúrák memóriahasználata (bájtban mérve)



3.1. ábra. Adatstruktúrák memóriahasználata az erlang:memory/1 függvénnyel mérve

Az eddig bemutatott memóriahasználat méréses sajnos rosszul reprodukálhatóak. Fluktuáló értékeket kaptam a tesztek többszöri ismétlésekor. Emellett tovább nehezíti az eredményeket összehasonlítását, hogy a tárolók implementációjából adódóan különböző memóriatípusokat használnak. Ennek következtében más megközelítést kerestem a memóriahasználat pontosabb mérésére. A következőekben az Erlang shell-en kívülről, az operációs rendszerben futó folyamatok erőforrás-használatának monitorozásával próbálkoztam, amivel konzisztensebb méréseket sikerült elérni. A futó Erlang virtuális gépek, linux operációs rendszeren egy-egy "beam.smp" nevű folyamaton belül futnak (SMP = Symmetric Multi-Processing). Ennek a folyamatnak a memóriahasználatát vizsgáltam és jegyeztem fel. Azt figyeltem meg a mérések futtatása során, hogy a memóriahasználat az adatstruktúra feltöltése alatt megnövekedik, majd egy idő után visszaesik és végül elér egy stabil, a kezdetinél magasabb szintet. Példaként a parikls/mem_usage_ui szoftver [28] segítségével vizualizáltam egy map 20 millió adatrekorddal való feltöltését a 3.2. ábrán.



3.2. ábra. Erlang folyamat memóriahasználata map adatstruktúra feltöltésekor 20 millió adatrekorddal

Feltehetőleg ez az azért adódik így, mivel Erlangban a kötött változók megváltoztathatatlanok, így a példán a map-be új elemek beszúrásakor újabb map-ek jönnek létre. A régi map-ekre ezután már nincs szükség ezért az Erlang szemétgyűjtője felszabadítja őket és ezért csak idővel áll be a stabil memóriaszint, ami már nem tartalmaz felszabadításra váró adatokat. Emiatt a érdemes megvizsgálni a maximális és a végső stabil memóriahasználatot is a különböző adatstruktúrák esetén. Ezt két mérés kereteiben tettem meg, először 1 millió rekorddal és 10 db mintavétellel, majd 10 millió rekorddal és 3 db mintavétellel. A mérések eredményeit a 3.7. és 3.8. táblázatokon ábrázoltam és összehasonlítottam a különböző adatstruktúrákat a map memóriahasználatával.

	Memóriahasználat (MiB)			
Adatstruktúra	max	max/map arány	stabil	stabil/map arány
list	39	76.39%	16	76.92%
dict	70	138.10%	24	117.31%
array	51	101.59%	22	105.29%
map	50	100.00%	21	100.00%
ets	17	33.13%	13	62.98%
digraph	17	33.13%	13	62.98%

3.7. táblázat. 5. mérés: Memóriahasználat vizsgálata az Erlang shell-en kívülről (1 millió rekord, 10 mérés átlaga)

	Memóriahasználat (MiB)			
Adatstruktúra	max	max/map arány	stabil	stabil/map arány
list	4 259	88.26%	1 726	77.54%
dict	2 026	41.98%	1 426	64.06%
array	2 893	59.94%	1 593	71.55%
map	4 826	100.00%	2 226	100.00%
ets	1 693	35.07%	1 426	64.06%
digraph	1 959	40.60%	1 426	64.06%

3.8. táblázat. 6. mérés: Memóriahasználat vizsgálata az Erlang shell-en kívülről (10 millió rekord, 3 mérés átlaga)

Memóriahasználat méréseinek kiértékelése

A 3.7. táblázaton látható, hogy a legtöbb adatstruktúra esetén a beszúrások során elért maximális memóriahasználat meghaladja a stabil állapot kétszeresét, kivéve az ets és digraph esetén, amik a korábbi tapasztalatokkal megegyezően ismét hasonló eredményeket értek el. A dict memóriahasználata bizonyult a legmagasabbnak. A map és az array hasonlóan teljesítettek, viszont 1 millió adatrekord esetén az array 1.59%-kal magassabb max és 5.29%-kal magasabb stabil memóriahasználatot ért el a map-hez képest. Ilyen adatmennyiség kelezésekor az ets és a digraph használta a legkevesebb memóriát a stabil állapot beálltakor, csupán 62.98%-át a map által igénybe vett mennyiségnek.

Az adatmennyiség növelésével érdekes változások figyelhetőek meg az eredményekben a 3.8. táblázaton. Az adatstruktúrák többségénél a tízszeres adatmennyiség növelésével közel egyenesen arányos, százszoros memóriahasználat növekedés figyelhető meg, kivéve a dict és array esetén, amik ennél jobban skálázódtak. Kifejezetten érdekes, hogy a dictionary utolsó helyről első helyre ugrott a stabil és harmadik helyre a maximális memóriahasználat tekintetében. Az eredmények azt mutatják, hogy array ilyen nagy adatmennyiséget kisebb memóriahasználat mellett képes kezelni, mint a map. Az ets és digraph map-hez viszonyított eredményeit vizsgálva viszont az is megfigyelhető, hogy a map kissé jobban skálázódik, mint az ets tábla alapú tárolók, hiszen azok vesztettek a map-hez viszonyított előnyükből.

3.1.9. Konklúzió

A műveletidők vizsgálata során megmutatkozott, hogy a map és az array alkalmas adatstruktúrák lehetnek az in-memory adatbázisréteg működésének megvalósítására. Ugyan az array gyorsabban végzi a beszúrásokat, az olvasásban a map több mint 25%-kal gyorsabbnak bizonyult nagyobb adatmennyiségek kezelésekor. További előnye a map-nek hogy képes összetett kulcsokat is kezelni, míg az array csak egész számokkal indexelhető. A memóriahasználat mérésekor kisebb adathalmazon mindkét adatstruktúra hasonlóam teljesített, viszont az adathalmaz növelésével a map magasabb memóriahasználatot produkált, mint az array. Mivel a gyors műveletidő, kifejezetten az olvasás sebesség fontosabb kritérium volt az adatstruktúra kiválasztása során, ezért ennek ellenére is a mep-re esett a végső választás az adatréteg megvalósításához.

3.2. Gráf reprezentáció

Ebben a fejezetben ismertetem részletesebben a RefactorErl szemantikus programgráfjának felépítését és az azt felépítő három réteg sémáját.

3.2.1. Erlang kód gráf leképezése

A RefactorErl által feldolgozott fájlokról szerzett információt egy adatbázisban tárolja a szoftver, amelyben a kódot egy szemantikus programgráfként reprezentálja. [29, 30] A szemantikus programgráf az Erlang programkódról információt reprezentáló csúcsokból és a köztük lévő kapcsolatokat leíró élekből áll. A gráf felépítése a 3.1. fejezetben említett három egymásra épülő réteget foglalja magába.

Lexikai réteg

Az Erlang forráskód szöveges reprezentációját tartalmaza beleértve a szóközöket és a kommenteket is. Ebben a rétegben tárolt információ alapján teljes mértékben visszaállítható az eredeti fájl tartalma. A forráskód token-ekre van bontva, melyek atomi alkotóelemei a forráskódnak. Ilyenek például az operátorok, kulcsszavak és a literálok, mint a számok és az atomok.

Szintaktikus réteg

A RefactorErl szintaktikus rétege az Erlang forráskódok absztrakt szintaxisfájára épül (`erl_syntax` modul [31]) pár módosítással, hogy a szemantikai reprezentációt megkönnyítse.

Szemantikus réteg

Az Erlang forráskód szemantikáját szintaktikus csúcsokat összekötő szemantikus élek és különböző szemantikus csúcs típusok reprezentálják a gráfban. A szemantikus információk közé tartoznak például a függvények, modulok, rekord definíciók és változók is.

Attribútumok tárolása a gráfban

A RefactorErl szemantikus programgráfja egy attribútum gráf. Ez azt jelenti, hogy a gráf csúcsaihoz és éleihez bizonyos attribútumok vannak rendelve. Az adott entitáshoz hozzárendelt attribútumok az entitás típusától függnek. A RefactorErl programon belül ez rekordok formájában van leképezve. Példaként a következő kódrészleten olvasható pár csúcshoz tartozó rekord, köztük szintaktikai (`file`, `expr`), lexikai (`lex`, `token`) és szemantikus (`module`, `func`, `variable`) csúcsok attribútumainak definíciója egyaránt.

```
-record(file,       {type, path, eol, lastmod, hash}).
-record(expr,       {type, role, value, pp=none}).
-record(lex,        {type, data}).
-record(token,      {type, text, prews="", postws="", scalar, linecol}).
-record(module,     {name}).
-record(func,       {name             :: atom(),
                     arity            :: integer(),
                     dirty  = int     :: no | int | ext,
                     type   = regular :: regular | anonymous,
                     opaque = false   :: false | module | name | arity}).
-record(variable,   {name}).
% további rekord definíciók ...
```

Látható, hogy a fájl csúcsokhoz öt attribútum is tartozik, a fájl típusa, elérési útvonala a fájlrendszeren, a fájban használt sor vége karakter, az utolsó módosítás ideje és a fájl hashelt átírása, míg a modulokhoz egyedül egy név attribútum rendelhető.

A gráf rétegeinek sémája

A RefactorErl gráfjának éleihez nem tartoznak típusonként eltérő attribútumok. Egyedül a csúcsok közt fennálló kapcsolatok típusának tárolása és azok sorrendjének megőrzése a feladatuk. A szemantikus programgráf sémája határozza meg hogy a gráfban milyen csúcsokat milyen típusú élek köthetnek össze egymással. A lexikai és szintaktikai séma a szoftverben makrókban van definiálva. A lexikális réteg sémáját a `LEXICAL_SCHEMA`, valamint a szintaktikai réteg sémáját a `SYNTAX_SCHEMA` makró határozza meg. Ebből például kiolvasható, hogy a lexikai rétegben `lex` csúcsokat `elex` élek kapcsolhatnak az `expr` kifejezésekhez. A szintaktikai réteg sémája továbbá meghatározza, hogy a kifejezés típusú csúcsokhoz (`expr`) kapcsolódhat egy másik kifejezés `esub` típusú élen keresztül, vagy egy klóz típusú csúcs (`clause`) a `catchcl`, `exprcl`, `aftercl` vagy `headcl` élek valamelyikén keresztül.

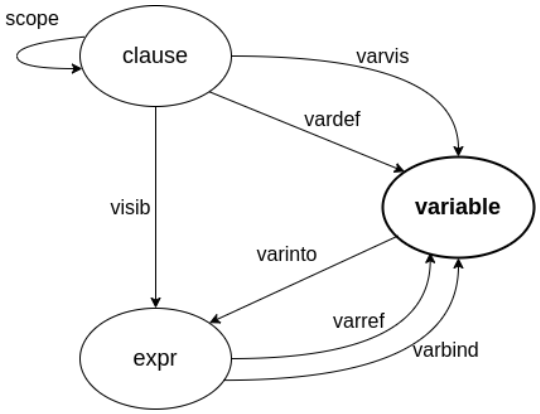
```
-define(LEXICAL_SCHEMA,
    [
     {file,   [{incl, file}]},
     {expr,   [{elex, lex}]},
     {lex,
      record_info(fields, lex),
      [{mref, form}, {orig, lex}, {llex, lex}]},
     % további szerkezet definíciók...
    ]).

-define(SYNTAX_SCHEMA,
    [{root, [], [{file, file}]},
     {file, record_info(fields, file), [{form, form}]},
     {expr,
      record_info(fields, expr),
      [{aftercl, clause},
```

```
      {catchcl, clause},
      {esub, expr},
      {exprcl, clause},
      {headcl, clause}]},
    % további szerkezet definíciók ...
    ]).
```

A szemantikus séma definíciója a programgráf szemantikai rétegét felépítő elemzőkben található. A RefactorErl-ben az Erlang kód különböző szemantikai elemeiért külön elemző modulok felelősek. A teljes szemantikus séma az összes elemző sémájának együtteséből áll elő. Ezek a `refcore_anal` viselkedést megvalósító callback modulok, melyeknek a `schema/0` függvénye írja le a szemantikai séma adott modulra vonatkozó részletét. Például a `refanal_var`, változókkal kapcsolatos elemzésekért felelős modul sémájában megtalálható a `variable` szemantikus csúcs és az ahhoz kapcsolódó élek definíciója, amelyek a 3.3. ábrán látható sémát írják le.

```
schema() ->
    [{variable, record_info(fields, variable), [{varintro, expr}]},
     {clause, [{scope, clause}, {visib, expr},
               {vardef, variable}, {varvis, variable}]},
     {expr,   [{varref, variable}, {varbind, variable}]}
    ].
```



3.3. ábra. Változók sémája a gráf szemantikus rétegében

3.3. Map alapú gráftároló modell

A map alapú in-memory tároló megvalósításához egy az előzőekben bemutatott felépítésű attribútumgráf tárolására alkalmas modell felépítése volt szükséges. Az Erlang nyelv jövőbeli bővülését és a RefactorErl szoftverben várható fejlesztéseket is figyelembe kellett vennem, hogy az általam meghatározott modell képes legyen eltárolni a szükséges adatokat a típusok, attribútumok és sémák bővülése esetén is. Ezeket a szempontokat figyelembe véve a következő modellt építettem fel három rekord defincíójával.

```
-record(state, {
    nodes   = #{} :: map(),
    edges   = #{} :: map(),
    node_id = 0   :: integer(),
    edge_id = 1   :: integer()
}).
```

A state rekord az adatbázisréteg állapotának, azaz az adatbázisnak a tárolására lett kialakítva. Külön map-ekben tárolja a gráf csúcsokat és éleit, melyeknek alapértelmezett értékei egy-egy üres map. Ezeken kívül még két elemet tartalmaz a rekord, a map-ekben soron következő azonosítók eltárolására. Amikor egy új elemet szükséges beszúrni, akkor a jelenlegi állapotból kiolvasható milyen index-szel kell bekerülnie az adatbázisba, majd beszúrás során megnöveljük az state-ben tárolt indexet. Fontos, hogy ezt atomi módon tegyük meg, hogy megőrizzük az azonosítók egyediségét. (Erről gondoskodik a gen_server tervezési minta, amit részletesebben tárgyal a 3.4.1. fejezet.) A csúcsok esetén 0 az azonosító alapértelmezett kezdőértéke, mivel a gráf gyökereként mindig beszúrásra kerül egy root típusú kitüntetett és egyedi csúcs. Az élek esetén ilyenre nincs szükség, ezért azok indexelése 1-től kezdődik.

```
-record(node, {
    id              :: integer(),
    class           :: atom(),
    data            :: nodeData(),
    edges_fwd  = [] :: list(integer()),
    edges_back = [] :: list(integer())
}).
```

A node rekord modellezi a gráf csúcsait és tartalmazza a csúcs azonosítóját, amit annak beszúrás során a state rekord `node_id` tagjából nyertünk ki. A `class` és `data` elemekben tároljuk a csúcs típusát és attribútumait, amik a beszúrási kéréskor egy `nodeData()` típusú adat rekordban érkeznek be az adatbázisréteghez. A `nodeData()` típus specifikációt úgy definiáltam, hogy tartalmazzon minden csúcs típust, ami az attribútumgráf definíciójában is megtalálható. A modellem bármilyen csúcshoz tartozó attribútum halmazt képes eltárolni. Emiatt ha a jövőben a RefactorErl-be bekerül egy újfajta csúcs, akkor elegendő a `nodeData()` típusát kibővíteni vele. Emellett a `edges_fwd` és `edges_back` tagokban pedig rögzíthetők az adott csúcsból kivetező és abba beérkező élek azonosítói, amik egy-egy listában vannak tárolva. Mivel a programgráf egy irányított gráf, ezért az éleket külön listában szükséges nyilvántartanunk.

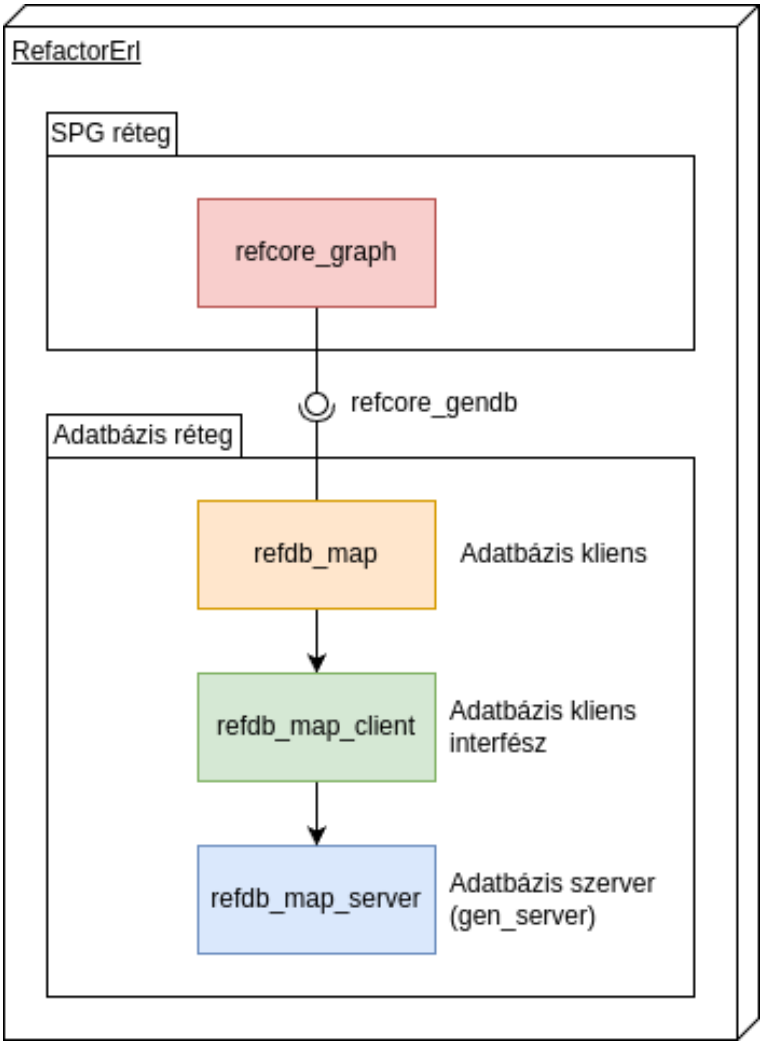
```
-record(edge, {
    id   :: integer(),
    from :: integer(),
    tag  :: atom(),
    idx  :: integer(),
    to   :: integer()
}).
```

Az adatbázis modellben az edge rekord reprezentálja a gráf éleit. Itt is jelen van egy egyedi

azonosító, ami a node-hoz hasonlóan a state rekord `edge_id` tagjával áll összefüggésben. Irányított élek lévén a `from` és `to` tagok tartalmazzák az él kiindulási és végpontjaiban megtalálható csúcsok azonosítóit. Az él címkéjét a `tag`-ben tudjuk tárolni, ami a típusát határozza meg. Továbbá a rekord rendelkezik egy `idx` taggal, ami egy (az egyedi azonosítótól független) index eltárolására alkalmas. A gráf egyik csúcspontjából több azonos típusú él is eredhet. Mivel az élek egyedi azonosítója egy az adatbázis rétegen belüli belső azonosító, ezért ez az index egyrészt azt a célt szolgálja, hogy az adatbázison kívül az megegyező csúcsból eredő és címkéjű éleket elkülönítsük egymástól. Másrészt az index további szerepe, hogy az élek sorrendje konzisztens maradjon. Ez kulcsfontosságú tulajdonság annak érdekében, hogy a forráskódot alkotóelemeit a kódnak megfelelő sorrendben tudja értelmezni és elemezni a RefactorErl szoftver.

3.4. Architektúra

A következőkben alulról felfelé haladva bemutatom az általam készített adatbázisréteg struktúráját és az azt felépítő modulokat, kezdve a adatbáziskezelő rendszert megvalósító szerver (`refdb_map_server`) modullal, majd ismertetem a szerver interfészét elrejtő (`refdb_map_client`), végül pedig az adatbázis klienst megvalósító (`refdb_map`) modulokat. Az architektúra megtervezésénél törekedtem a felelősségek szétválasztására, ezért az in-memory adatbáziskezelő maga is réteges felépítésű. Így amennyiben módosítás következne be a RefactorErl oldaláról az adatrétegekkel történő kommuikáció megvalósításáért felelős `refcore_gendb` modulban, elegendő volna az adatbázis kliensét módosítani, az adatok kezeléséért felelős réteget viszont nem érintené a változás.



3.4. ábra. In-memory adatbázisréteg felépítése

3.4.1. Állapot megőrzése, Üzleti logika (refdb_map_server)

Az adatbázis réteg legfontosabb feladata elsősorban az adatok tárolása. Mivel az Erlang egy funkcionális nyelv, ezért nem olyan triviális feladat az adatbázis állapotának számontartása, mint más objektumorientált nyelvekben, ahol egy singleton osztály belső tagjaként egyszerűen tárolható volna az állapot. Szerencsére az Erlang fejlesztői gondoltak erre és az Erlang OTP-ben megvalósított tervezési minták között erre a problémára is megoldást nyújtottak a `gen_server` viselkedési mintával. A `gen_server` célja, hogy egy kliens-szerver mintához nyújtson általános implementációt. A `gen_server` alkalmas állapotmegőrzésre, és több kliens konkurrens kéréseinek kezelésére. Ezt a viselkedési mintát valósítottam meg a `refdb_map_server` callback modulban, amelyben az állapotot a 3.3. fejezetben bemutatott state rekordban tároltam. A szerver modul kizárólag a 3.3. fejezetben bemutatott map alapú adatmodellel dolgozik, így elkülöníthetőek a

különböző rétegek közötti felelősségek. Továbbá ez a modul tartalmazza az "üzleti logikát", azaz a map alapú tárolón végzett beszúrás, törlés, rekord elérés, path lekérdezés alapú útvonalkeresés és az összes további adatbáziskezelő művelet részletes megvalósítását.

Ehhez a `gen_server` mintának a következő callback függvényeit volt szükséges megvalósítani.

- `init/1`: Az a függvény mindig meghívásra kerül, amikor a szerver folyamatot elindítjuk a `start/3,4`, `start_monitor/3,4` vagy `start_link/3,4` függvényeinek egyikével. Ez a függvény felelős a szerver kezdeti állapotának inicializációjáért. A mi esetünkben egy üres state rekordot állít be, amibe a gráf gyökeréül szolgáló kitüntetett root típusú csúcsot szúrja be.
- `handle_call/3`: Ez a függvény értékelődik ki, amikor a kliens egy szinkron hívást küld a szervernek a `call/2,3` vagy `multi_call/2,3,4` függvények használatával. A kliens várakozik, amíg nem érkezik válasz a szervertől, vagy nem következik be időtúllépés (timeout) esemény. A metódus egy tuple-lel tér vissza, ami tartalmazza a kliensnek visszakülött választ, illetve a szerver új állapotát, azaz módosítás esetén a frissített state rekordott.
- `handle_cast/2`: Ez a függvény értékelődik ki, amikor a kliens egy aszinkron hívást küld a szervernek a `cast/2` vagy `abcast/2,3` függvények használatával, melyek egyből visszatérnek egy `ok` üzenettel a szerver állapotától függetlenül. A metódusnak a `handle_call` hívással megegyező felépítésű tuple a visszatérési értéke.

A viselkedési minta következő opcionális callback metódusait csak naplózással kezeltem le, mivel ezek nem várt események bekövetkeztét jelzik. Fejlesztés során előnyt jelenthetnek bizonyos problémák felderítésére.

- `handle_info/2`: Bármi egyéb üzenet érkezésekor, ami nem szinkron vagy aszinkron kérés, amit lekezel a `handle_call/3` vagy `handle_cast/2` callback az a `handle_info/2` függvénybe érkezik be.
- `terminate/2`: A szerver a saját terminálása előtt futtatja ezt a függvényt. Ez az `init/1` függvény ellentettje, tehát a feladata a szerver által használt erőforrások lezárása. Mivel inicializáció során a szerver egyedül a kezdőállapotot állítja be, ezért itt nincs teendőnk. Egyedül naplózás történik, hogy amennyiben szükséges értesüljünk a terminálás okáról.
- `code_change/3`: Az Erlang környezetekben lehetséges futás közben frissíteni a program forráskódját (hot code upgrade). Ilyen esetekben előfordulhat, hogy verzióváltáskor megváltozik a `gen_server`-t megvalósító callback modul viselkedése is és a korábbitól eltérő struktúrájú belső állapotot vár a működéshez. Ilyenkor az állapot migrációját ez a függvény felelős elvégezni. Paraméterül megkapja a kiindulási és cél verziókat, illetve a legutóbbi ismert állapotot. Ha létezik az adott verziók között definiált migráció, akkor a szerver elvégzi az állapotának átalakítását, hogy megszakítás nélkül tovább működhessen a szerverért felelős Erlang folyamat. Mivel ez az adatréteget egyelőre csak a koncepció működőképességének bizonyítására készült, és nincs éles verziókban bevezetve, ezért ilyen migrációkre nincs szükség. Ettől függetlenül érdemes naplózni az ilyen események bekövetkeztét is.

Mivel a `gen_server` működése is az Erlang nyelv üzenetküldésein alapul, ezért az adatbázis elérés során a konkorrencia sem okoz problémát. A szervernek küldött kérések bekerülnek a szerverért

felelős folyamat által számontartott beérkező üzenetek sorába. A kérések beérkezési sorrendben kerülnek feldolgozásra egyetlen szerver folyamat által. A szerveren belüli műveletek az üzenet beérkezése után kiváltott `handle_call/3` vagy `handle_cast/2` hívástól kezdve az adott függvény visszatértéséig egy tranzakció részeinek tekinthetők. A 3.3. fejezetben említett azonosító kiosztás problémának megoldásáról is az gondoskodik, hogy egyszerre csak egy kérést dolgoz fel a szerver, így új adat beszúrásakor a következő felhasználható azonosító kiolvasása és az új rekord beszúrása atomi műveletnek tekinthető a kliens oldaláról szemlélve.

3.4.2. Szerver absztrakció, Kliens interfész (refdb_map_client)

A `gen_server` modul nem csak a callback modulok által megvalósítandó generikus függvényeket tartalmazza, hanem a callback modul funkcióinak elérése is ezen a modulon keresztül történik. Az előző fejezetben említett függvények közül a `gen_server:start_link/4` hívásával indítottam el a szervert, majd a szerver adatbáziskezelő funkcióit a `gen_server:call/2,3` és `gen_server:cast/2` függvények megfelelő paraméterezésével értem el. Ezeket a hívásokat a szervert használó kliens elől érdemes elrejteni, mert ez olvashatóbb, könnyebben használható és karbantartható kódot eredményez. Az adatbázis szerver eléréséhez szükséges `gen_server` hívások absztrahálására a `refdb_map_client` kliens interfész modult hoztam létre.

A RefactorErl konvenciójának megfelelően a projekt `core.hrl` header fájljába felvittem az adatbázis szerver folyamat azonosítóját egy makróba a következőképpen:

```
-define(MAP_SERVER, refdb_map_server).
```

Majd a kliens interfészen belül a következőképpen fedtem el a szerver funkciók elérését:

```
start_link() ->
    gen_server:start_link({local, ?MAP_SERVER}, ?MAP_SERVER, [], []).

path(Id, Path) when is_list(Path) ->
    gen_server:call(?MAP_SERVER, {path, Id, Path}).
```

3.4.3. Adatbázis kliens (refdb_map)

Az adatbázis klienst a `refdb_map` modulban implementáltam. Ez az általam megvalósított adatbáziskezelő legkülső rétege, amivel a RefactorErl létező részei közvetlenül kommunikálhatnak, tehát ide érkeznek be először a kérések a szoftver felől. A `refdb_map` modul a `refdb_map_client` használata által elérhető kliens interfészen keresztül fér hozzá az adatbázishoz.

A RefactorErl tartalmazza a kompatibilis adatbázis kliensek interfészének specifikációját a saját `refcore_gendb` viselkedési mintájában. Ezt a viselkedést valósítja meg az összes már létező és a szoftverbe integrált adatbáziskezelő réteg a kapcsolat kialakítására. Ez a viselkedés szabja meg azt az adatmodellt amit az adatréteggel történő kommunikáció során használ a RefactorErl.

Ez a modell eltér a 3.3. fejezetben definiált, adatbázisban használt modelltől, így a `refdb_map` moldul feladata a modellek közötti átírások kezelése is. A kliens modellben a gráf csúcsai egy három elemű tuple-ként a következő módon vannak reprezentálva:

```
-define(NODETAG, '$gn').
-type class()     :: atom().
-type id()        :: integer().
-type gnode()     :: {?NODETAG, Class::class(), ID::id()}.
```

A `NODETAG`-nek csupán annyi szerepe van, hogy jelezze, hogy az adott tuple a gráf egy csúcsát tartalmazza. Ezen kívül a `gnode()` tuple-ben csak a csúcs típusa és azonosítója van jelen. A gráf attribútumait tároló rekordot (lásd 3.2.1. fejezet) külön adatként kezeli a kliens. Az éleknek nincsenek attribútumai, így elegendőek a szomszédos csúcsok és az él címkéje az élek reprezentálására.

A `refcore_gendb` viselkedési minta emellett azt is meghatározza, hogy milyen adatelérési, módosító és egyéb műveletekkel kell rendelkeznie az adatrétegnek. A következőekben bemutatom a fontosabb callback függvényeket amelyeket a viselkedés megkövetel.

- Az adatbázisréteg inicializálása:

  ```
  init(Args::any()) ->
      {ok, State::any()} |
      {stop, Reason::any()}.
  ```

  Ez a függvény indítja el az adatbáziskezelő folyamatot. A függvény egy argumentumot kap, ami felhasználható az adatbázis konfigurálására. Egy tuple a visszatérési értéke, ami tartalmazza az adatbázis kezdeti állapotát, vagy sikertelen inicializáció esetén a hiba okát. A `gen_server:init/1` függvényének a megfelelője.

- Az adatbázisréteg leállítása:

  ```
  terminate(Reason::any(), State::any()) ->
      any().
  ```

  A leállítással kapcsolatos adminisztratív feladatok elvégzése. A `gen_server:terminate/2` függvényének a megfelelője.

- Csúcs vagy él tároló létrehozása:

  ```
  create_class(Class::atom()) ->
      any().
  ```

  A függvény feladata, hogy létrehozza a gráf építőelemei számára az adattároló objektumokat. Például a Mnesia alapú adatrétegben típusonként külön táblák tartják számon a gráf csúcsait és éleit, így fontos meggyőződni a táblák létezéséről és szükség esetén létrehozni azokat, mielőtt az adatbázis készen állna az adatok tárolására.

  A map alapú, in-memory adatréteg egy-egy adatstruktúrában tárolja a gráf összes csúcsát és élét, így nem szükséges csúcs- és él-osztályok alapján külön tárolókat létrehozni. Az `init` függvény elvégez minden szükséges előkészítést a `state` rekord inicializálásával.

- Gráf csúcsokkal kapcsolatos műveletek
  - A gráf gyökerének lekérdezése:

    ```
    root() ->
        {ok, Root::gnode()}.
    ```

    A szemantikus programgráf kiindulópontjaként szolgáló kitültetett root csúcsot adja vissza.
  - Csúcs beszúrása:

    ```
    create(data()) ->
        {ok, Node::gnode()}.
    ```

    Beszúr egy új csúcsot a gráfba a megadott `data()` rekordban kapott attribútumokkal. Visszatér az `{ok, {?NODETAG, Class, Id}}` tuple-lel, ahol a `Class` értékét az attribútum rekord első eleme határozza meg az osztályt, az `Id` pedig az adatbázis által az új rekordnak kiosztott azonosítót tartalmazza.
  - Csúcs frissítése:

    ```
    update(Node::gnode(), Data::data()) ->
        {ok, Node::gnode()} |
        {error, {bad_class, Class::class()}}.
    ```

    Felülírja egy megadott, meglévő csúcshoz tartozó adatokat a megadott attribútumokkal. Sikeres művelet esetén visszatér.
  - Csúcs törlése:

    ```
    delete(Node::gnode()) ->
        ok.
    ```

    Eltávolítja a gráfból a megadott csúcsot és az ahhoz kapcsolódó éleket, majd visszatér az `ok` atommal.
  - Csúcs adatainak lekérdezése:

    ```
    data(Node::gnode()) ->
        {ok, Data::data()} |
        {error, bad_node}.
    ```

    Amennyiben megtalálható az adatbázisban az adott csúcs, akkor visszaadja a csúcshoz tartozó attribútum rekordot, egyébként pedig a `bed_node` üzenetet.
- Gráf élekkel kapcsolatos műveletek
  - Él beszúrása:

    ```
    mklink(From::gnode(), Tag::tag(), To::gnode()) ->
        ok |
        {error, {bad_node, Node::gnode()}} |
        {error, {bad_nodes, From::gnode(), To::gnode()}} |
        {error, bad_link} |
        {error, {bad_link, From::gnode(), Tag::tag(), To::gnode()}}.
    ```

A gráf megadott csúcsai közé hozzáad egy új élet a megadott `Tag::tag()`) alapján, ahol `-type tag() :: atom() | {atom(), integer()}`. Ennek megfelelően a beszúrás lehetséges előre megadott index-szel, amennyiben a kiindulási csúcs még nem rendelkezik a megadott címkéjű és indexű éllel, illetve lehetséges egyedül a címke megadásával, amikor viszont az adatbázis automatikusan osztja ki a következő szabad indexet az élnek. Sikeres művelete esetén egy `ok` atommal, míg hiba esetén egy `{error, Error}` tuple-lal tér vissza, ahol `Error` tartalmazza a hiba okát.

– Él törlése:

```
rmlink(From::gnode(), Tag::tag(), To::gnode()) ->
    ok |
    {error, not_exists} |
    {error, bad_link} |
    {error, {bad_link, From::gnode(), Tag::tag(), To::gnode()}}.
```

Törli a gráf adott csúcsai közti, megadott címkével rendelkező élet. Az előző függvényekkel megegyezően a visszatérési értéke `ok` vagy `{error, Error}`.

– Él indexének lekérdezése:

```
index(From::gnode(), Tag::tag(), To::gnode()) ->
    {ok, none} |
    {ok, Id::id()} |
    {error, {bad_node, From::gnode()}} |
    {error, {bad_nodes, From::gnode(), To::gnode()}} |
    {error, bad_link} |
    {error, {bad_link, From::gnode(), Tag::tag(), To::gnode()}}.
```

A megadott `From` kiincsulási csúcs és `To` cél csúcs közötti, `Tag` címkével rendelkező él indexének lekérdezése. Amennyiben az adatbázisban megtalálhatóak a megadott csúcsok, de nem létezik ilyen él közöttük, akkor a válasz `{ok, none}`, ha viszont létezik ilyen él, akkor az `{ok, Id::id()}` tuple.

◦ Összetett műveletek a gráfon

– Csúcs éleinek lekérdezése:

```
links(Node::gnode()) ->
    {ok, [{Tag::tag(), Node::gnode()}]}.
```

Lekérdezhetőek az adott csúcsból kiinduló élek és az általuk elérhető szomszédos csúcsok listája. A válaszban adott lista elemeit az élek címkéje alapján csoportosítva `{Tag::tag(), Node::gnode()}` formában adja vissza, ahol az elemek sorrendjét a hozzájuk vezető élek indexe határozza meg.

Például egy `refdb_map:links({'$gn',root,0})` hívás eredményének részlete:

```
[{env,{'$gn',env,1}},
 {env,{'$gn',env,2}},
 {env,{'$gn',env,4}},
 ...,
 {file,{'$gn',file,7}},
 ...]
```

– Path lekérdezések:

```
path(Node::gnode(), Path::pathexpr()) ->
    {ok, [Node::gnode()]} |
    {ok, []} |
    {error, Reason::any()}.
```

A `Node::gnode()` kiinduló csúcsból, a `Path::pathexpr()` mintára illeszkedő útvonalakon elérhető csúcsok lekérdezése. A `pathexpr()` a RefactorErl path nyelvén leírt kifejezés, amely a gráfban egy különböző lépésekből álló útvonalat ír le. A path nyelv felépítése a következő:

```
-type pathexpr() :: [pathelem()].
-type pathelem() :: step() | {step(), filter()}.
-type step()     :: tag() | {tag(), 'back'}.
-type filter()   ::
    'last' |
    id() |
    {id(), 'last'} |
    {id(), id()} |
    {'not', filter()} |
    {filter(), 'and', filter()} |
    {filter(), 'or', filter()} |
    {attribute(), operator(), any()}.
-type operator() :: '>' | '>=' | '<' | '=<' | '==' | '/='.
```

Az útvonal `pathelem()` elemekből áll, amik a gráf bejárása során végrehajtandó lépéseket írják le. A csúcsokból induló lépések `step()` része határozza meg, hogy milyen címkék mentén haladjunk, illetve, hogy az irányított gráfban az adott csúcsból kimenő élek felé, vagy az abba beérkező élek irányában visszafelé lépjünk. A lépés opcionális `filter()` tagja további szűréseket érvényesíthet a lépéshez igénybe vehető éleken, szűkítve az adott lépés eredményét.

A `filter()` által használható szűrési feltételek a következők:

* `'last'`: Index alapján az utolsó él kiválasztása.
* `id()`: Pontosan a megadott indexű él kiválasztása.
* `{id(), 'last'}`: Az `id()` által meghatározott és annál nagyobb indexű él halmaz kiválasztása.
* `{id(), id()}`: A megadott indexek által meghatározott zárt intervallumba eső élek kiválasztása.
* `{'not', filter()}`: A szűrő által meghatározott élhalmaz komplementere.
* `{filter(), 'and', filter()}`: Két szűrő metszete.
* `{filter(), 'or', filter()}`: Két szűrő uniója.
* `{attribute(), operator(), any()}`: A lépés által elérhető csúcsok attribútumain értelmezett szűrés. Olyan csúcsok lesznek a lépés eredményében, amik rendelkeznek a szükséges attribútummal és az általa felvett értékre teljesül a megadátt operátor és érték alapján meghatározott kritérium.

3.5. Path algoritmus

A path lekérdezéseket feldolgozó algoritmust a `refdb_map_server` modulban valósítottam meg a `path(State :: state(), Id :: nodeid(), Path :: pathexpr())` függvényében. Az adatbázis jelenlegi állapota a `State` pareméteren keresztül elérhető, a lekérdezés kiindulópontja pedig az `Id` által azonosított csúcs. Ezt a csúcsot a `path` függvényben kérdezem le és ellenőrzöm a létezését, majd ha jelen van az adatbázisban, akkor a `do_path` függvényben rekurzív módon dolgozom fel a `Path` által leírt lépéseket. A `do_path` rekurzió megállási feltétele az ha elfogynak a `Path`-ból a további lépések, így a bejárás mélységét a `Path` lista elemszáma határozza meg. Az soron következő `patheelem()` által leírt lépés kiértékelését a `path_step` metódus végzi el, amely visszaadja egy megadott csúcs összes olyan szomszédját, ahova egy adott lépés vezet. A `path_step` lépések összegyűjtése során a csúcsok sorrendjére nem szükséges figyelni, mivel a `refdb_map` kliens modul a szükséges sorba rendezi az eredményeket. Emiatt a részlisták összefűzése `do_path(State, NextNode, Rest) ++ Acc` módra történik, mivel általában a `do_path` hívás eredménye egy rövidebb lista, mint az egyre bővülő `Acc` akkumulátor. Ezzel a listák konkatenálásához kevesebb műveletre van szükség.

1. Algorithm Path lekérdezés feldolgozása

Funct path(State :: state(), Id :: nodeid(), Path :: pathexpr())

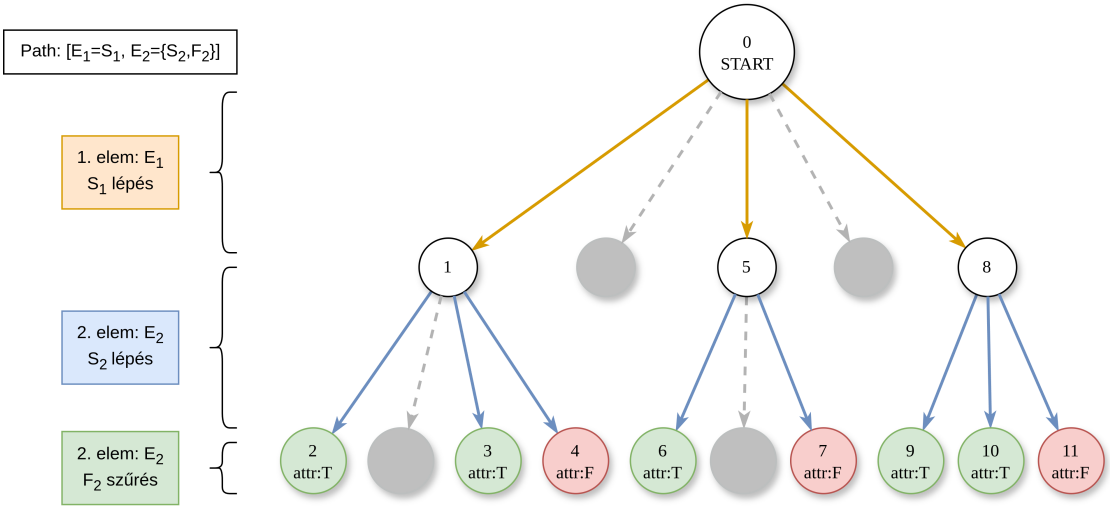
```
 1: Node ← find_node_by_id(Id)
 2:
 3: if Node = not_found then
        return [ ]
 4: end if
 5:
 6: return do_path(State, Node, Path)
 7:
```

Funct do_path(State :: state(), Node :: node(), Path :: pathexpr())

```
 1: if Path = [ ] then
        return [Node]
 2: end if
 3:
 4: [PathElem|Rest] ← Path
 5: StepResult ← path_step(State, Node, PathElem)
 6:
 7: Acc ← [ ]
 8: for all NextNode in path_step(State, Node, PathElem) do
 9:     Acc ← do_path(State, NextNode, Rest) ++ Acc
10: end for
```

A lépések feldolgozása a szélességi (BFS=Breadth-First Search) és a mélységi (DFS=Depth-First Search) bejárások tulajdonságait ötvözi. Először a BFS-hez hasonlóan az adott csúcsnak ahol az algoritmus tart, megkeresi az összes szomszédját, amit az adott lépés eredményezhet. Ezek után a DFS-hez hasonlóan, az első szomszédot kiválaszva mélységi irányba folytatja a bejárást és rekurzívan megismétli a fennmaradó lépések alkalmazását. Szemléltetésképpen a 3.5. ábrán látható gráfon a $[E_1 = S_1, E_2 = (S_2, F_2)]$ útvonal lekérdezése, ahol $\{E_1, E_2\} \in pathelem()$, $\{S_1, S_2\} \in step()$ és $F_2 \in filter()$, a $[2, 3, 6, 9, 10]$ csúcsokat eredményezi. A bejárás először megkeresi a 0-ás kiindulási csúcs $E_1 = S_1$ lépésnek megfelelő szomszédait, azaz a $[1, 5, 8]$ csúcsokat.

Az adott rétegben szürkével jelölt csúcsok nem érhetők el afeldolgozott lépésen kereszül. Ez után a fennmaradó E_2 elemet először az 1-es csúcson vizsgálja meg. A S_2 lépés alapján megkapjuk a $[2, 3, 4]$ csúcsokat, majd ezen a F_2 szűrést alkalmazva csak a $[2, 3]$ csúcsok maradnak az eredményben. Az F_2 a 3.5. ábra esetében $attr = T$, ami csúcsok egy attribútumának értéke alapján szűri az eredményt. Ezt követően, ha még lennének további elemei a `Path` listának, akkor a 2-es csúcs szomszédjait vizsgálná tovább az algoritmus. Mivel elfogytak a lépések, ezért az algoritmus visszalép az előző szintre és az 5-ös, majd a 8-as csúcsokon ugyanígy alkalmazza az E_2 által leírt útvonalat. Az így összegyűjtött végső eredmény a $[2, 3, 6, 9, 10]$ csúcslista lesz, melyek a `Path` által leírt útvonalon megegyező távolságra vannak a 0-ás kiindulási csúcstól.



3.5. ábra. Path algoritmus szemléltetése

4. fejezet

Eredmények kiértékelése

A RefactorErl program indítása során lehetőséget kell biztosítani a felhasználóknak az általam készített adatbázis használatához. Ebben a fejezetben bemutatom, hogy hogyan tettem elérhetővé a map alapú in-memory adatréteget a RefactorErl-ben. Az integráció meglétéhez az adatbázis hibamentes és helyes működése is elengedhetetlen, ezért azt is ismertetem, hogy ezt hogyan igazoltam az új adatbáziskezelő esetében. Végül megvizsgáltam az elkészült adatbázis teljesítményét valós felhasználási esetekben. Az így kapott eredmények kiértékelése során összahasonlítom azokat a RefactorErl-be korábban integrált adatbázisok gyorsaságával, hogy megállapítsam, melyik megoldás nyújtja a legjobb teljesítményt.

4.1. Integráció a RefactorErl-be

Az in-memory adatbázis komponens a 3.4.3. fejezetben leírtaknak megfelelően, a `refcore_gendb` viselkedés megvalósításával kapcsoltam hozzá a RefactorErl szoftverhez. Ahhoz hogy a rendszert az új map alapú adatbázist felhasználó módban is el lehessen indítani a `bin/referl` szkript paraméterei között kibővítettem a `-db` kapcsoló által elfogadott értékeket a `map` opcióval a következőképpen:

```
if [ "$1" = -db ]
then
    if [ "$2" = map ]
    then
        DBMOD="-dbmod refdb_map"
        DBARGS="-dbargs []"
```

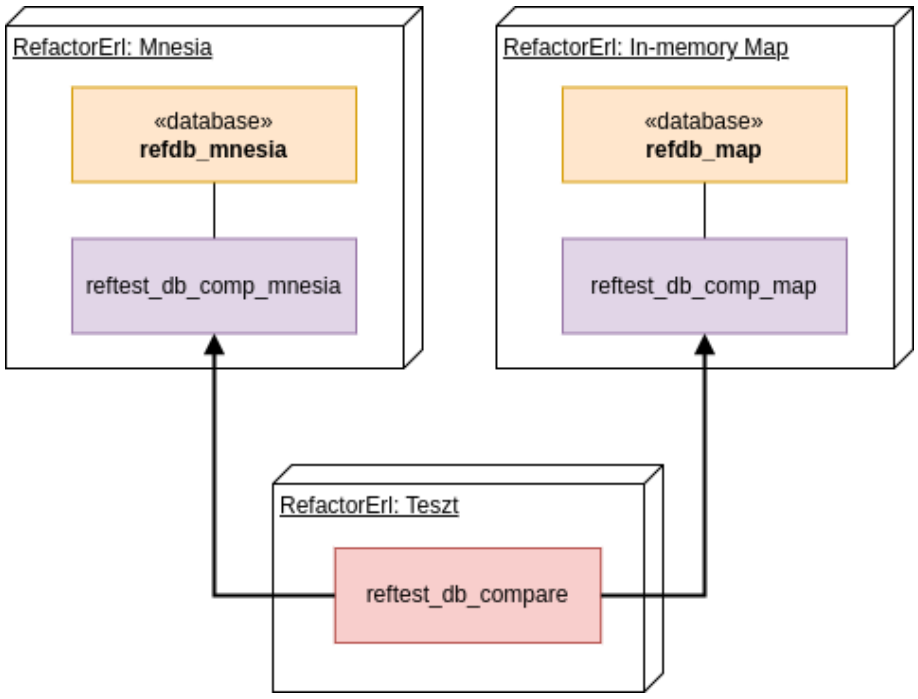
Ennek köszönhetően a RefactorErl egy egyszerű parancs kiadásával elindítható map-es adatbázis módban a projekt gyökérkönyvtárából:

```
bin/referl -db map
```

4.1.1. A modell helyességének vizsgálata

Az általam előállított modell helyességét azzal izagoltam, hogy megvizsgáltam, hogy egy kódbázis RefactorErlbe való betöltése ugyanazt az adatbázist eredményezi-e, mint amit a meglévő Mnesia alapú adatréteg használata esetén kapunk. Emellett a beszúrás során létrejött adatbázisokon lekérdezések segítségvel az adatelérést biztosító funkciók helyessége is igazolható. Ehhez a `refcore_gendb` viselkedést megvalósító callback modulok különböző műveletek esetén keletkező kimenetének az összehasonlítása szükséges.

A RefactorErl különböző, `reftest` prefixű modulokat tartalmaz a saját komponenseinek tesztelésére. Ezek között megtalálható a `reftest_db_compare` modul, ami alkalmas a szoftverhez kapcsolt adatrétegek integrációjának és működésük helyességének ellenőrzésére. Ez a modul meghatároz különböző adatbázis műveleteket és ezen műveleteknek egy elvégzési sorrendjét. Ezeket a műveleteket két különböző adatréteget használó RefactorErl node-on a megadott sorrendben végrehajtja, majd a részeredményeket és a végeredményt összehasonlítva megállapítható, hogy a két adatréteg működése megegyezik-e. Mivel a RefactorErl alapértelmezett módon a Mnesia alapú adatréteget használja annak megbízhatósága miatt, ezért azt választottam összehasonlítási alapként az új in-memory adatréteg tesztelése során. A `reftest_db_compare` összehasonlításért felelős modul a `reftest_db_comp_mnesia` modulon keresztül kommunikál a `refdb_mnesia` modulban megvalósított Mnesiás adatbázis klienssel. Ennek mintájára én is egy `reftest_db_comp_map` nevű modulban valósítottam meg a `refdb_map` modulnak küldött kéréseket. Így összesen 3 RefactorErl node-ra volt szükség a teszt elvégzéséhez. Egyre a Mnesia alapú adatbázis, egy másikra a Map alapú adatbázis futtatásáért, és a harmadikra, ami a két másik node-dal kommunikálva a tesztek futtatásáért felelt.



4.1. ábra. Validációs környezet felépítése

A `reftest_db_comp_map` modulban megvalósított funkciók a következő műveleteket foglalják magukba:

- `file(Node)`: A gráf gyökeréből `file` éleken keresztül elérhető csúcsok attribútum rekordjai.
- `file_id(Node,File)`: A gráf gyökeréből `file` éleken keresztül elérhető csúcsok attribútumrekordjai, amelyek megegyeznek a megadott `File` rekorddal.
- `file_lnk(Node,FNode)`: A megadott `FNode` csúcs élei.
- `form(Node,FNode)`: A megadott `FNode` csúcs attribútum rekordja.
- `form_index(Node, FNode)`: A megadott `FNode` csúcsból induló `form` élek indexei.
- `clauses(Node,Form)`: A megadott `Form` csúcsból a `funcl` éleken keresztül elérhető csúcsok.
- `clause_index(Node, Forms, ClauseNode)`: A megadott `Forms` csúcsok és a `ClauseNode` csúcs közötti, `funcl` címkéjű élek indexei.
- `expr_index(Node, Clauses, ExpNode)`: A megadott `Clauses` csúcsokból a `ExpNode` által meghatározott címkével és végponttal rendelkező élek indexei, ahol `ExpNode :: {tag(), gnode()}`.
- `collect_expr_ind(Node, Exprs, ExpNode)`: A megadott `Exprs` csúcsokból a `ExpNode` által meghatározott címkével és végponttal rendelkező élek indexei, ahol `ExpNode :: {tag(), gnode()}`.
- `expr(Node,Clause)`: A `Clause` csúccsal szomszédos olyan él-csúcs párok, ahol a szomszéd csúcs típusa `expr`. A 2. paraméter típusa `Clause :: gnode() | {tag(), gnode()}`.
- `collect_expr_id(Node,Expr)`: A megadott `Expr` csúcs által meghatározott csúccsal szomszédos, `esub` élen keresztül elérhető `expr` típusú csúcsba vezető él-csúcs párok, ahol `Expr :: {tag(), gnode()}`.
- `expr_data(Node,Expr)`: A megadott `Expr` csúcs attribútum rekordja, ahol `Expr :: {tag(), gnode()}`.
- `collect_expr_cls_id(Node,ClNode)`: A megadott `ClNode` csúcs által meghatározott csúcscsal szomszédos, `exprcl` vagy `headcl` élen keresztül elérhető, `clause` típusú csúcsba vezető él-csúcs párok listája, ahol `ClNode :: {tag(), gnode()}`.
- `collect_expr_cls_ind(Node, Exprs, ClauseNode)`: A megadott `Exprs` lista által meghatározott csúcsokból a `ClauseNode` csúcsba vezető `headcl` vagy `exprcl` címkéjű élek indexei.

Az teszt által felfedett hibákat javítottam az in-memory tárolóban, míg végül az összehasonlítást lefuttatva a következő sikeres eredményt kaptam:

```
START
Filecompare: true
FormIndexCompare: true
FormCompare: true
```

```
ClausesCompare: true
ExpIndex: true
ExpData: true
ExpressionCompare: true
Recursive_Clause_Exp_Indexes_Compare: true
Recursive_Clause_Exp_Compare: true
RecExp compare: true
```

Az új in-memory adatréteg helyességét alátámasztja az összes integrációs teszt sikeres teljesítése.

4.2. Az új adatréteg teljesítményének vizsgálata

Ebben a fejezetben bemutatom a Mnesia, NIF és Kyoto Cabinet alapú tárolók teljesítményét valós használatnak megfelelő, komplex írás és olvasás műveletek mérésével, majd összehasonlítom azokat az új map alapú in-memory adatréteg által elért teljesítménnyel. A méréseket a következő technikai paraméterekkel rendelkező számítógépen végeztem el:

```
Processzor: Intel i5-12600K (6P+4E mag, 16 szál)
Memória:    48GB DDR4 3200MHz CL17
Háttértár:  500GB Samsung 970 EVO Plus NVMe
```

4.2.1. Beszúrások vizsgálata

A beszúrások gyorsaságát az Erlang Mnesia kódbázisának[1] RefactorErl-hez való hozzáadásához szükséges idő mérésével vizsgáltam meg. A hozzáadáshoz az `ri:add(erlang, mnesia)` parancsot használtam, a szükséges időt pedit a `timer:tc(ri, add, [erlang, mnesia])` hívással mértem meg, amivel az én rendszeremen mikorszekundumban mérve kaptam meg az eltelt időt. A mért idők tartalmazzák a 28,156 soros kódbázis adatbázisba történő eltárolásához szükséges időt, amely egy 394,515 csúcsból és 890,089 élből álló gráf felépítését eredményezi, illetve a beszúrás után a RefactorErl által automatikusan lefutatott kezdeti szemantikai elemzéseket. A teszt során 5 mérést végeztem, majd kiszámoltam ezek átlagát és a Mnesiához viszonyított értéküket. Ezt a kisebb mintát elegendőnek tekintettem, mivel az eredmények között nem találhatók szignifikánsan kiugró adatok. Az eredmények a 4.1. táblázaton láthatók.

[1] Az Mnesia alkalmazás forráskódjának általam használt verziója a következő címen érhető el: https://github.com/erlang/otp/tree/maint-26/lib/mnesia

	Eltelt idő (μs)			
Mérés	**Mnesia**	**NIF**	**Kyoto**	**Map**
1	105,638,870	26,798,840	39,484,630	45,437,248
2	115,016,860	27,228,527	39,903,820	46,329,952
3	105,594,620	27,151,260	39,944,281	45,687,768
4	105,933,276	27,102,363	40,643,878	45,709,218
5	106,330,406	27,546,824	40,261,429	45,433,036
Átlag	**107,702,806.4**	**27,165,562.8**	**40,047,607.6**	**45,719,444.4**
Mnesiához viszonyítva	**100%**	**25.2%**	**37.1%**	**42.4%**

4.1. táblázat. Mnesia adatbázisba töltéséhez szükséges idők összehasonlítása

A Mnesia adatréteg átlagosan 107.7 másodperc alatt végzi el a kódbázis feldolgozását, aminél jobban teljesít a Map, aminek 45.7 mp-re volt szüksége. Ennél valamivel jobban teljesít a Kyoto, aminek 40mp-be, és még jobban a NIF, aminek 27 mp-be telt az adatbázis feltöltése. A Mnesiához viszonítva az összes többi adatréteg egy nagyságrenddel gyorsabban végezte el a műveletet. A Map rétegnek ehhez az idő 42.4%-ára, a Kyotonak 37.1%-ára, a NIF-nek pedig csupán a 25.2%-ára volt szüksége.

Az eredmények alapján kijelenthetjük, hogy a C++ nyelven megvalósított adatrétegek, az Erlang és C++ közötti natív interakciónak köszönhetően, a RefactorErl szoftverbe beágyazott beszúrások esetében gyorsabbak, mint a tisztán Erlang alapú adatrétegek. Ennek ellenére a Map alapú adatbázis figyelemre méltó, 57.6%-os növekedést ért el a Mnesiához képest, valamint a Kyoto Cabinethez képest csupán 14%-kal több időt igényelt ugyanazon művelet elvégzéséhez.

4.2.2. Lekérdezések vizsgálata

A különböző adatrétegek olvasási sebességének vizsgálata során is izolált olvasások helyett, komplex használati esetekben szerettem volna méréseket végezni, hogy az eredmények jól tükrözzék az adatbázisok valós működését. Ehhez a RefactorErl funkciói közül a szemantikus lekérdezéseket választottam mérendő műveletnek. A felhasználói felületeken futtatot lekérdezéseket a RefactorErl eljuttatja az üzleti logikai réteghez[2], ami a kapott szemantikus lekérdezést lefordítja a 3.4.3. fejezetben bemutatott path nyelvre. A path nyelvet az adatrétegek már képesek feldolgozni és ez alapján az adatbázison végrehajtani a lekérdezést, majd kiolvasni és összegyűjteni az eredményt felépítő rekordokat.

A lekérdezések futtatását a RefactorErl Erlang shell interfészén keresztül, az `ri:q(QueryString)` függvényhívással valósítottam meg. A műveletidő meghatározásához ebben az esetben is a `timer:tc/2` metódust használtam. A különböző lekérdezések futtattását többször ismételtem, majd ezek átlagát vettem eredményül. Egyes lekérdezéseket esetén eltérő mennyiségű mintát vettem a végeredmény meghatározásához. Erre azért volt szükség, mert a bonyolultabb lekérdezések több időt vesznek igénybe, de minden esetben arra törekedtem, hogy a lehető legtöbb mérés alapján határozzam meg a végső eredményt.

[2] A RefactorErl felépítésének bemutatását lásd a 2.1. fejezetben és a rétegmodellt ábrázoló 2.1. ábráján.

13 különböző lekérdezést vizsgáltam meg, először egy kisebb, majd egy nagyobb méretű adatbázison. Az S_i, ahol $i \in [1; 13]$ lekérdezések elvégzésekor a Mnesia adatbáziskezelő forráskódja volt betöltve a RefactorErl-be. Az L_i lekérdezések végrehajtása előtt a Mnesián kívül, az Erlang SSH[3] (Secure Shell klienst megvalósító) és Edoc[4] (Erlang program dokumentáció generálására alkalmas) programok forrását is hozzáadtam az egyes adatrétegekkel futó RefactorErl példányok adatbázisához, hogy egy megnövekedett adatmennyiséggel is megvizsgáljam a végrehajtásukhoz szükséges időt. Ahogy a 4.2.1. fejezetben is olvasható, az S_i mérésekben használt Mnesia forráskódja 28,156 sor kódból áll, valamint egy 394,515 csúcsból és 890,089 élből álló programgráfot alkot. A L_i mérések alatt betöltött Mnesia, SSH és Edoc alkalmazások együttesen 54,330 sorból állnak. A RefactorErl-hez való hozzáadásuk egy 957,163 csúcsos és 2,110,396 éles gráfot eredményez. Így az összesen 26 fajta mérést a 4.2. és 4.3. táblázatokon olvasható módon határoztam meg. A táblázatok megadják a lekérdezéseket és az alkalmazott mintavételek számát is. A kisebb adatbázison végrehajtott S_i mérések eredményei a 4.4., míg a nagyobb adatbázist vizsgáló L_i mérések eredményei a 4.5. táblázaton találhatók meg.

Mérés	**Lekérdezés**	**Minták száma**
S_1	mods.loc:sum	100
S_2	mods.records[name=mnesia_select].refs	10 ezer
S_3	mods.records[name=mnesia_select].field[name=orig].refs	10 ezer
S_4	mods.funs[max_length_of_line>80]	10
S_5	mods.funs.exprs.sub[type=integer, not .macro]	10
S_6	mods.funs[.calls[module=lists]]	100
S_7	mods.funs[max_depth_of_cases>3]	10
S_8	mods[name=mnesia_log].funs[name=open_log] .refs[.param[index=1]]	10 ezer
S_9	mods[name=mnesia_log].funs[name=open_log] .refs[.param[index=1].origin[type=atom, value=decision_log]]	10 ezer
S_{10}	mods[name=mnesia_log].funs.exprs .sub[type=tuple,[.sub[index=1]]]	1000
S_{11}	mods[name=mnesia_log].funs.exprs.sub[type=tuple, [.sub[index=1].origin[type=atom,value=backup_args]]]	100
S_{12}	mods.funs[arity=5].refs[.param[index=1]]	100
S_{13}	mods.funs[arity=5].refs[.param[index=1].origin[type=atom]]	10

4.2. táblázat. Adatbázis: Mnesia; 28,156 sor kód; 394,515 csúcs és 890,089 él

[3]Az SSH alkalmazás forráskódjának általam használt verziója a következő címen érhető el: https://github.com/erlang/otp/tree/maint-26/lib/ssh

[4]Az Edoc alkalmazás forráskódjának általam használt verziója a következő címen érhető el: https://github.com/erlang/otp/tree/maint-26/lib/edoc

Mérés	Lekérdezés	Minták száma
L_1	mods.loc:sum	100
L_2	mods.records[name=mnesia_select].refs	1000
L_3	mods.records[name=mnesia_select].field[name=orig].refs	1000
L_4	mods.funs[max_length_of_line>80]	10
L_5	mods.funs.exprs.sub[type=integer, not .macro]	10
L_6	mods.funs[.calls[module=lists]]	100
L_7	mods.funs[max_depth_of_cases>3]	10
L_8	mods[name=mnesia_log].funs[name=open_log] .refs[.param[index=1]]	1000
L_9	mods[name=mnesia_log].funs[name=open_log] .refs[.param[index=1].origin[type=atom, value=decision_log]]	100
L_{10}	mods[name=mnesia_log].funs.exprs .sub[type=tuple,[.sub[index=1]]]	1000
L_{11}	mods[name=mnesia_log].funs.exprs.sub[type=tuple, [.sub[index=1].origin[type=atom,value=backup_args]]]	10
L_{12}	mods.funs[arity=5].refs[.param[index=1]]	100
L_{13}	mods.funs[arity=5].refs[.param[index=1].origin[type=atom]]	10

4.3. táblázat. Adatbázis: Mnesia, SSH, Edoc; 54,330 sor kód; 957,163 csúcs és 2,110,396 él

	Műveletidő (μs)				**Arányok**		
Mérés	**Mnesia**	**NIF**	**Kyoto**	**Map**	$\frac{NIF}{Mnesia}$	$\frac{Kyoto}{Mnesia}$	$\frac{Map}{Mnesia}$
S_1	4,341,414	1,313,283	1,893,560	3,240,063	30.3%	43.6%	**74.6%**
S_2	10,515	2,713	4,331	7,404	25.8%	41.2%	**70.4%**
S_3	2,582	1,087	2,313	3,140	42.1%	89.6%	**121.6%**
S_4	4,179,300	1,257,600	1,821,460	2,887,861	30.1%	43.6%	**69.1%**
S_5	1,423,675	843,211	1,274,261	1,308,156	59.2%	89.5%	**91.9%**
S_6	160,665	71,004	134,434	280,878	44.2%	83.7%	**174.8%**
S_7	2,064,304	619,237	799,504	1,069,551	30.0%	38.7%	**51.8%**
S_8	22,444	6,823	8,119	13,078	30.4%	36.2%	**58.3%**
S_9	29,652	8,045	9,960	15,506	27.1%	33.6%	**52.3%**
S_{10}	124,774	61,547	85,635	95,873	49.3%	68.6%	**76.8%**
S_{11}	926,845	335,401	473,578	373,077	36.2%	51.1%	**40.3%**
S_{12}	114,464	56,464	75,540	97,061	49.3%	66.0%	**84.8%**
S_{13}	4,840,435	1,855,058	2,384,807	2,116,360	38.3%	49.3%	**43.7%**

4.4. táblázat. Lekérdezésekhez szükséges idők összehasonlítása

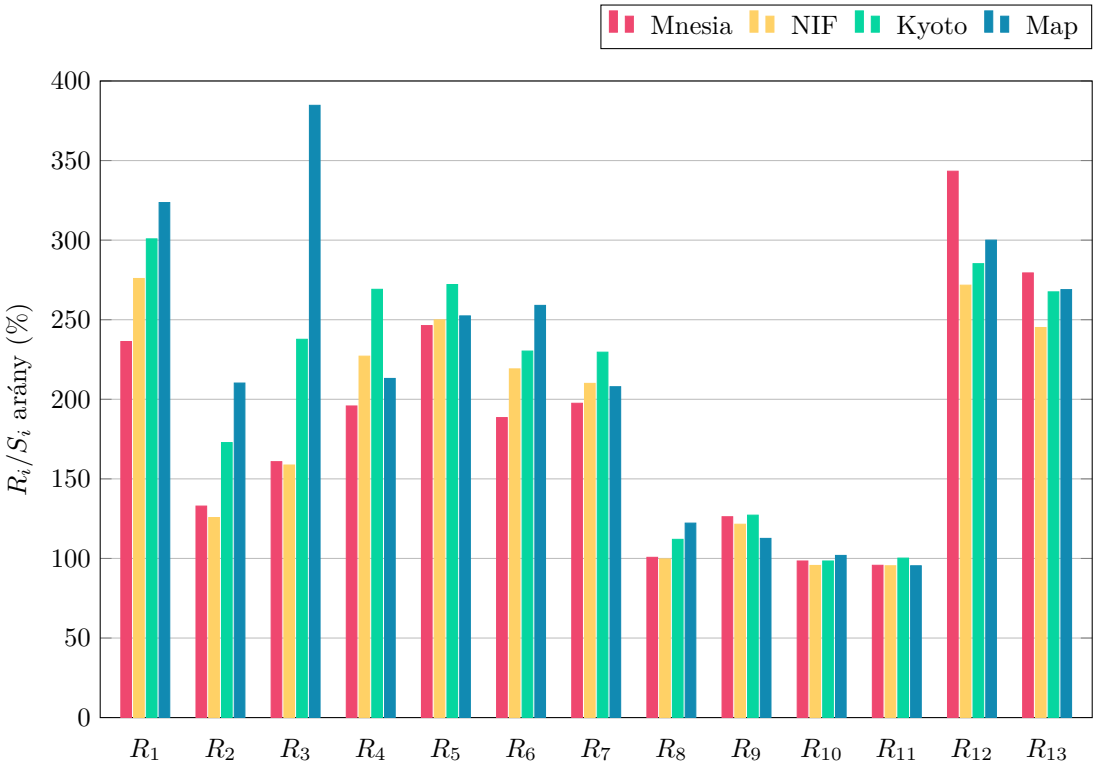
	Műveletidő (μs)				**Arányok**		
Mérés	**Mnesia**	**NIF**	**Kyoto**	**Map**	$\frac{NIF}{Mnesia}$	$\frac{Kyoto}{Mnesia}$	$\frac{Map}{Mnesia}$
L_1	10,259,955	3,622,934	5,696,092	10,484,160	35.3%	55.5%	**102.2%**
L_2	13,978	3,410	7,484	15,567	24.4%	53.5%	**111.4%**
L_3	4,153	1,724	5,499	12,082	41.5%	132.4%	**290.9%**
L_4	8,182,683	2,855,932	4,902,308	6,153,943	34.9%	59.9%	**75.2%**
L_5	3,506,434	2,107,053	3,467,040	3,302,239	60.1%	98.9%	**94.2%**
L_6	302,822	155,566	309,564	727,567	51.4%	102.2%	**240.3%**
L_7	4,076,420	1,300,128	1,835,874	2,223,075	31.9%	45.0%	**54.5%**
L_8	22,596	6,801	9,092	15,978	30.1%	40.2%	**70.7%**
L_9	37,411	9,778	12,668	17,465	26.1%	33.9%	**46.7%**
L_{10}	122,799	58,819	84,224	97,651	47.9%	68.6%	**79.5%**
L_{11}	887,300	319,822	474,735	355,971	36.0%	53.5%	**40.1%**
L_{12}	392,919	153,394	215,416	291,179	39.0%	54.8%	**74.1%**
L_{13}	13,523,725	4,546,092	6,378,456	5,690,800	33.6%	47.2%	**42.1%**

4.5. táblázat. Lekérdezésekhez szükséges idők összehasonlítása

A kizárólag a Mnesiát tartalmazó adathalmazon az S_3 és S_6 lekérdezések kivételével a Map alapú adatbázis gyorsabb volt mint a Mnesia. A Kyoto Cabinetet a S_{11} és S_{13} esetében előzte meg a Map, és az S_5 esetében is csupán 0.034 másodperc különbség adódott köztük. Minden esetben a NIF adatbázis érte el a leggyorsabb futásidőt. A nagyobb, három alkalmazást magába foglaló adatbázison is átlagosan gyorsabb működést biztosított a Map, mint a Mnesia. Az 4.4. táblázat eredményeihez képest immár az L_1 és L_2 eredményeiben a Map elmaradt a Mnesiához képest, de zeken a méréseken a NIF és a Kyoto is, ugyan gyorsabb eredményt érnek el, mint a Mnesia, de az S_1 és S_2-ben mért előnyükből veszítettek. Ebből arra következtethetünk, hogy bizonyos esetekben a Mnesia jobban skálázódik az adatmennyiség tekintetében. Ennek a vizsgálatához kiszámoltam, hogy az egyes adatbázisok esetén a nagy és kis adatbázisokon mért eredmények hogyan viszonyulnak egymáshoz, amelyre az R_i jelölést vezettem be.

$$R_i = \frac{L_i}{S_i}$$

Az így kapott értékeket a 4.2. ábrán illusztráltam.



4.2. ábra. Lekérdezések elvégzési idejének változása adatbáziskezelőnként, az adatbázis több mint kétszeres méretnövelésének hatására

Sok esetben a Mnesia lekérdezési ideje kevésbé növekszik meg az adatmennyiség növelésének következtében, mint a többi adatbázisnak, így kijelenthezjük, hogy az a lekérdezések futtatásakor kevésbé érzékeny az adatbázis méretére. A Map alapú adatbázis lekérdezési ideje, a NIF-hez és Kyoto Cabinethez hasonló mértékben növekszik meg az adatbázisméretének növelése után. Ennek feltehetően a Mnesiás adatréteg lekérdezésfeldolgozási mechanizmusa és maga a Mnesia adatbázis belső lekérdezésoptimalizációja lehet a hátterében. A Mnesiás adatréteg, ugyanis a path lekérdezések eredményét ha lehetséges, a táblák kapcsolásával, egy lekérdezésben kérdezi le az adatbázisból. Az adatrétegében felépített összetett lekérdezések optimalizációját a Mnesia adatbáziskezelő motorja végzi el.

Az (S_8, S_9), (S_{10}, S_{11}), (S_{12}, S_{13}), illetve az (L_8, L_9), (L_{10}, L_{11}), (L_{12}, L_{13}) lekérdezések páronként hasonlóak egymáshoz, azzal a különbséggel, hogy a pár egyik tagja tartalmaz a program adatfolyamára vonatkozó szűrési feltételt is. Ezeknek az eredményeit vizsgálva megfigyelhető, hogy az `origin` kulcsszót tartalmazó lekérdezéseket szignifikánsan gyorsabban végzi el a Map alapú adatbázis, mint a Mnesia. Az `origin` segítségével lekérdezhetjük, hogy milyen értéket vehet fel egy kifejezés vagy annak értéke alapján szűrhetjük az eredménybe kerülő kifejezéseket. Az ezt a kulcsszót alkalmazó lekérdezéseket a Mnesia nem tudja egyetlen lekérdezéssé fordítva eljuttatni a Mnesia adatbáziskezelőjéhez. Ehelyett (a Map alapú adatbázisréteg lekérdezési mechanizmusával megegyező módon) sok kis lekérdezést végez el az adatfolyamok kiszámításához, majd ezekből a részeredményekből építi fel a végső eredményt. A 8-11 számú lekérdezések a Mnesia alkalmazás forráskódjának elemeire szűrnek, mint például a `mnesia_select` rekordra,

vagy a `mnesia_log` modulra, így azok a programgráfban a Mnesia alkalmazás kódját leképező részgráfra lokalizáltak. Emiatt az R_8-R_{11} oszlopokon egyik adatbázis esetén sem figyelhető meg nagy változás a lekérdezések feldolgozási idejében a nagy és kis adatbázisok között. A 12 és 13 számú lekérdezések ezzel ellentétben nem tartalmaznak szűrést modulnév alapján, így a gráf egészét meg kell vizsgálni olyan 5 paraméteres függvények után kutatva, amelynek az első paramétere lehet egy atom. Ez a művelet sokkal több részlekérdezéssel jár a Mnesia adatréteg esetén, ami miatt mind a Map, mind a NIF és Kyoto Cabinet adatbázisok sokkal gyorsabbak ilyen esetben.

4.3. Továbbfejlesztési lehetőségek

A 4.2.1. fejezet eredményein látható, hogy sok esetekben sikerült gyorsabb működést elérni, mint amit a Mnesia adatréteg használatával, de továbbra is a natív C++ alapú adatbázisokat felhasználó adatrétegek nyújtják a leggyorsabb működést. Emiatt érdemes lehet további optimalizálási lehetőségeket keresni a Map alapú in-memory tároló továbfejlesztése végett.

Érdemes lehet egy másik map struktúra kipróbálása és gyorsaságának összehasonlítása a meglévő modellel. Jelenleg egy map tároló felelős a szemantikus programgráf összes csúcsának, és egy másik map annak összes élének tárolásáért. A csúcsokat és éleket azok típusa alapján külön-külön map-ekben tárolva lehetséges, hogy rövidebb keresési idők is elérhetők lennének, amennyiben a részstruktúrák karbantartása nem okozna túlzott teljesítményvesztést.

További megközelítés lehet a map helyett az array adatstruktúra alkalmazása is. A diplomamunkám során, a fejlesztés kezdetekor fontos szempontnak tartottam, hogy az adatbázis rekordjait akár összetett kulcsokkal is azonosítani lehessen. A végső implementáció az adatbázis ezen tulajdonságát nem használja ki, ehelyett egy olyan adatmodellt alkottam meg, amiben azonosítóknak kizárólag egész számok vannak használatban. Ilyen esetben az előzetes mérések alapján array-ek használatával is igen gyors adatelérési idő érhető el. Mivel az in-memory tároló megvalósítása során alkalmaztam a "separation of concerns" szoftvertervezési elvet, így maga az adatréteg is réteges felépítéssel rendelkezik. Ebből adódóan ha a jövőben lesz erre próbálkozás, amíg a kliens interfésszel való kompatibilitás teljesül, elegendő lesz a `refdb_map_server` komponens belső működését (és adott esetben annak elnevezését) átalakítani.

Ezeken túl a tartósság megvalósítása is releváns továbbfejlesztési célkitűzés lehet. Ez nem volna bonyolult feladat, mivel kizárólag Erlang alapú adatstruktúrákat használ az elkészült adatbázis, ami miatt a state rekordot bináris adattá alakítva a következő két egyszerű függvény ellátná a perzisztens tárolás feladatát.

```
write(FileName, State) ->
    file:write_file(FileName, erlang:term_to_binary(State)).

read(FileName) ->
    {ok,Binary} = file:read_file(FileName),
    erlang:binary_to_term(Binary).
```

Ezek használatával a Mnesia alkalmazás programgráfja 0.2-0.3 s alatt kiírható egy fájlba, majd

0.5-0.6 s alatt onnan kiolvasható és dekódolható belőle ismét az adatbázis eredeti állapota. Ilyen formán lehetséges volna egy háttérfolyamatként megvalósítani az adatbázis perzisztens adattárolását is, viszont ennek ellenére ez nem képezi részét az adatréteg funkcionalitásának, mert kifejezetten egy "könnyúsúlyú" megoldás megvalósítása volt az eredeti célkitűzés.

5. fejezet

Kapcsolódó irodalom

A RefactorErl szemantikus programgráfjának tárolására a diplomamunkám részeként egy célspecifikus, Erlang alapú, in-memory adatbázist valósítottam meg. A dolgozatom 3.1. fejezetében már ismertettem az Erlang nyújtotta adattároló struktúrákat, illetve a 2.3. fejezetben a RefactorErl által használható egyéb adatrétegeket. Ezek mellett érdemes kitekinteni, hogy milyen további adatbáziskezelők volnának alkalmasak a RefactorErl programgráfjának gyors kezelésére, illetve, hogy más statikus kódelemző szoftverek milyen adatbáziskezelőket alkalmaznak hasonló feladatra.

A modern adatbáziskezelők világában a legrégebbi és legkiforrottabb megközelítés a strukturált, relációs adatbáziskezelés, ezen belül is a táblás szervezésű, SQL lekérdezőnyelvet használó adatbázisok terjedtek el a leginkább. Ezek elterjedése után egyre nagyobb teret kezdtek hódítotani maguknak a nem strukturált és NoSQL adatbáziskezelők is, amik mára már hasonló teljesítményt képesek elérni, sőt bizonyos alkalmazási területeken meg is haladják az SQL alapú adatbáziskezelő rendszerekét [25, 32, 33]. A NoSQL adatbáziskezelők nagyobb szabadságot engednek a felhasználónak, mivel nem követelik meg egy séma szigorú alkalmazását és a gyors konkurrens műveletek megvalósítására törekednek, de előfordulhat, hogy ez az ACID tulajdonságok megszegését vonja maga után [34].

Az egyik legelterjedtebb gráfadatbázis, aminek a RefactorErl-be való integrálására már volt is korábbi próbálkozás a Neo4j. A Neo4j egy Java alapú, attribútumgráfok tárolására alkalmas gráfadatbáziskezelő, melybeől a Cypher lekérdezőnyelv segítségével kérdezhetők le a tárolt adatok. Ugyan rohamosan fejlődik, egyelőre csak bizonyos esetekben biztosít jobb teljesítményt a hagyományos SQL adatbázisokhoz képest és a használata magasabb memóriahasználatot is eredményez [35].

Az Apache Cassandra [36] és a ScyllaDB [37] nyílt forráskódú NoSQL adatbázisok, melyek a manapság legmagasabb áteresztőképességű és legalacsonyabb késleltetéssel rendelkező adatbáziskezelők közé tartoznak [38]. A Cassandra a Java nyelvre épült, míg a ScyllaDB egy C++ alapú megoldás. Jellemzően big data alkalmazásokban vannak használatban, tehát magas szintű konkurrencia kezelésére és óriási, akár terabájtokban mérhető adatmennyiségre vannak optimalizálva. Ebből adódóan elosztott környezetben, például egy felhőben létrehozott klaszterben érhető

el optimális teljesítmény a Cassandra és a ScyllaDB használata mellett. [34] Ezzel szemben a RefactorErl elemzéseit lokálisan a felhasználók saját hardverén való futtatásra tervezték. Ezek az adatbázisok ilyen lokalizált környezetben sokat vesztenek a teljesítményükből, így nem volnának alkalmasak a RefactorErl adatbáziskezelő feladatának betöltésére.

A Sonar statikus elemző a felhőben (SonarCloud [39]) és lokálisan (SonarQube [40]) is futtatható és több mint 30 programnyelv elemzésére képes. A forráskód védelmére való tekintettel kevés adatot tárol el a programok kódbázisáról. A RefactorErl-hez hasonlóan a SonarQube is használ adatbázist, amibe az elemzések eredményein kívül egy a forráskódról készített pillanatképet tárol el. A saját magunknak futtatott változat konfigurációjánál fontos szempont lehet, hogy a kódbázis csak annyira van biztonságban, amennyire az adatbázis is biztonságos. Ez az elemző kizárólag relációs adatbáziskezelőket támogat, köztük a PostgreSQL-t, a Microsoft SQL szervert és az Oracle adatbázist.

A Codacy egy több mint 40 nyelv elemzését, koztük a Java-t, Kotlin-t és SQL-t támogató, valamint nyílt forrású kódelemző eszköz [41]. Ez az eszköz még a SonarQube-nél is kevesebb adatbázist, egyedül PostgreSQL használatát támogatja [42]

A Klocwork [43] statikus elemző és SAST (Static Application Security Testing) eszköz a C, C++, C#, Java, JavaScript, Python, és Kotlin nyelvek analízisét támogatja. Az adattárolásra ez az eszköz MariaDB-t használ [44] ami a MySQL-nek egy változata. A lekérdezések egy részéhez, ahol a szöveg alapú keresés dominál, Apache Lucene [45] keresőmotort használ.

6. fejezet

Összefoglalás

A diplomamunkám célja az Erlang nyelv elemzésére és refaktorálására kifejlesztett, RefactorErl alkalmazáshoz egy "könnyűsúlyú" és hatékony, Erlang alapú, in-memory adatbáziskezelő backend elkészítése volt.

Az adatok tárolására használt Erlang adatstruktúra kiválasztásához megvizsgáltam a listák, dictionary-k, array-ek, map-ek, ETS táblák és digraph irányított gráftárolók írási, olvasási és törlési műveleti sebességét. A méréseket olyan adathalmazokon végeztem, amelyek rekordjai hasonló felépítésűek voltak, a RefactorErl által épített szemantikus programgráfban megtalálható csúcsok struktúrájához.

Először egy 1 millió Erlang rekordból álló adathalmazon végeztem méréseket. A dictionary adatstruktúra 1 millió rekorddal való feltöltése több mint 29 szekundumba telt, ami nagyságrendekkel tovább tartott, mint ugyanez a művelet bármelyik másik adatstruktúra esetén. A struktúrák feltöltésében a méréseim alapján a lista volt a leggyorsabb, amit az okozott, hogy a rekurzív struktúra elejére történt a beszúrás. A többi struktúra feltöltéséhez szükséges idők mérésekor azonos nagyságrendbe tartozó értékeket kaptam, melyek közül az array bizonyult a leggyorsabbnak 0.24 szekundumos mért idővel, míg a map-nek 0.5 s-re volt szüksége. A lista struktúra olvasási és törlési sebessége szignifikánsan lassabb a többi struktúránál. Egy rekord olvasása minden egyéb esetben körülbelül 1 μs-ba telt, míg a törlési idők között nagyobb eltéréseket tapasztaltam. Ismét az array volt a leggyorsabb, ami átlagosan 0.13 μs alatt törölt egy rekordot, míg a map-nek 1.3 μs-be telt ugyanez.

Ez után egy nagyobb adathalmazon, 10 millió rekorddal is megismételtem a méréseket, hogy az eredményekből az adatstruktúrák magasabb számosság melletti működéséről és azok skálázhatóságáról is információt gyűjtsek. Az adatbázis feltöltéséhez szükséges időt tekintve ebben az esetben is az array (2.9 s), ets (4.4 s), digraph (4.5 s), map (7.2 s) sorrend állt fenn. Olvasási sebességben ezzel szemben a map kimagaslóan gyors volt a tízszeres adatmennyiség kezelésében, csupán 1.04 μs műveleti idővel, míg a többi esetben ez az 1.4 μs-ot közelítő értéknek adódott. A skálázhatóságot tekintve ugyanolyan mértékben növekedtek a beszúrási idők, az olvasási és törlési idők változását vizsgálva viszont a map kiemelkedően jónak bizonyult. A map olvasási ideje 4%-os növekedést, míg törlési ideje 2%-os csökkenést mutatott. Ezzel szemben az array,

ets és digraph olvasási sebességei több, mint 40%-kal lettek magasabbak. Az array 15%-kal, a digraph 39% és az ets 147%-kal lassabban végezte el a törléseket.

Az előzetes tesztek részeként a korábban megviszgált adatstruktúrák memóriahasználatát is megvizsgáltam. Ezt többféle megközelítésből is megtettem, mivel az Erlang dinamikus memóriakezelése miatt nehéz informatív adatokat mérni ezzel a kérdéskörrel kapcsolatban. A mérések alapján az ETS alapú tárolók vették igénybe a legkevesebb memóriát. 1 millió rekordos adathalmaz esetén az array és a map hasonlóan viselkedett, de az array struktúra jó skálázhatóságának köszönhetően 10 millió rekord esetén már a map memóriahasználatának csupán 71.55%-ára volt szüksége.

A jól skálázódó, nagy adatmennyiség esetén is gyors olvasási és törlési idők, valamint az összetett adatstruktúrák kulcsként alkalmazhatósága miatt a map tároló használatával valósítottam meg az in-memory adatbázisréteget. A RefactorErl szemantikus programgráf struktúrájának tanulmányozása után megalkottam egy attribútumgráfok tárolására alkalmas adatmodellt. Ennek a modellnek a segítségével az új adatbázisréteget egy három réteges szoftverkomponensként valósítottam meg, ami a egy adatbáziskezelést megvalósító szerver modulból, egy adatbázis szerver interfészt absztraháló kliens interfész modulból és a modellek közti átalakításokat is megvalósító adatbázis kliens modulból áll. Az in-memory tároló implementációja után teszteléssel igazoltam annak helyes működését és megfelelő integrációját a RefactorErl meglévő komponenseihez.

Az elkészült adatréteg teljesítményét valós felhasználási esetek mérésével hasonlítottam össze a Mnesia, NIF és Kyoto Cabinet alapú adatbázisrétegekkel. A beszúrások vizsgálatához egy több mint 28 ezer soros kódbázist adtam az adatbázishoz, ami közel 400 ezer csúcs és 900 ezer él beszúrásával járt. A mérési erredmények azt mutatják, hogy ekkora kódbázis feldolgozásában a Mnesia-hoz képest sikerült jelentős, 57.6%-os teljesítménynövekedést elérni az új adatbázissal. A C++ alapú tárolók viszont a map-nél is gyorsabbak, a Kyoto Cabinet 12.4%-kal, a NIF alapú pedig 40.6%-kal. Az egyes adatrétegek olvasási sebességét szemantikus lekérdezések végrehajtásához szükséges idő mérésével valósítottam meg. 13 különböző lekérdezéssel végeztem méréseket 2 adatbázison. Az első tesztelés alatt álló adatbázisba az Erlang Mnesia, a másodikba pedig a Mnesia, SSH és Edoc alkalmazások voltak betöltve. Mind a kisebb, mind a nagyobb adatbázis esetén, a Mnesiánál gyorsabban végrehajtotta a lekérdezések többségét a map alapú adatbázis, bár bizonyos esetekben továbbra is a Mnesia volt gyorsabb. Ezekben az esetekben a Kyoto Cabinet is a megszokottnál lassabban hajtotta végre a lekérdezést. A map alapú adatbázis kevés esetben tudta megelőzni a Kyoto Cabinetet, a NIF pedig minden lekérdezés esetében a leggyorsabb adatbáziskezelő volt. A két eltérő méretű adatázis között mért eredmények azt mutatták, hogy a 4 adatbáziskezelő közül a Mnesia skálázódik a legjobban. Ez alól kivételt képeztek azok az esetek, ahol a gráf adatfolyamának vizsgálatára is szükség volt. Ezt az okozta, hogy általában a NIF, Kyoto és Map alapú adatbázisokkal ellentétben a Mnesiás adatréteg nem sok kis lekérdezés eredményéből építi fel a végeredményt, hanem ezeket egy lekérdezéssé fordítva egyszerre hajtja végre, a Mnesia adatbáziskezelővel. Mivel az adatfolyamot érintő szemantikus lekérdezéseket ilyen formán nem lehet optimalizálni, ezért a tiszta adatelérési idők határozzák meg az egyes adatbázisrétegek válaszidejét ilyen lekérdezések végrehajtásakor.

Összességében sikerült egy a RefactorErl alapértelmezett adatbáziskezelőjének, a Mnesiának teljesítményét a legtöbb esetben felülmúló in-memory adatbázisréteget megvalósítani és beépíteni az alkalmazásba. Ez az Erlang alapú megoldás a C++ nyelven implementált adatbázisok teljesítményét nem tudta meghaladni, de bizonyos esetekben megközelítette azt.

Irodalomjegyzék

[1] Ericsson AB. Erlang programozási nyelv dokumentációja. `https://www.erlang.org`. [Hozzárférés dátuma: 2024.05.29].

[2] Fred Hebert. *Learn You Some Erlang for Great Good! A Beginner's Guide*. No Starch Press, USA, 2013. [ISBN: 9781593274351].

[3] Joe Armstrong. The development of erlang. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, ICFP '97, page 196–203, New York, NY, USA, 1997. Association for Computing Machinery.

[4] Ericsson AB. Erlang/OTP System Documentation. `https://www.erlang.org/docs/24/pdf/otp-system-documentation.pdf`, 2024. [Hozzárférés dátuma: 2024.05.29].

[5] Simon St. Laurent. *Introducing Erlang*. O'Reilly Media, Inc, 2nd edition, 2017. [ISBN: 9781491973370].

[6] A RefactorErl hivatalos oldala. `https://plc.inf.elte.hu/erlang/`. [Hozzárférés dátuma: 2024.05.29].

[7] RefactorErl dokumentáció. `http://pnyf.inf.elte.hu/trac/refactorerl/wiki`. [Hozzárférés dátuma: 2024.05.29].

[8] Melinda Tóth and István Bozó. *Static Analysis of Complex Software Systems Implemented in Erlang*, pages 440–498. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[9] I. Bozó, D. Horpácsi, Z. Horváth, R. Kitlei, J. Köszegi, Tejfel. M., and M. Tóth. Refactorerl - source code analysis and refactoring in erlang. In *Proceedings of the 12th Symposium on Programming Languages and Software Tools, ISBN 978-9949-23-178-2*, pages 138–148, Tallin, Estonia, October 2011.

[10] Kyoto Cabinet dokumentáció. `https://dbmx.net/kyotocabinet/api/`. [Hozzárférés dátuma: 2024.05.29].

[11] A Nitrogen keretrendszer hivatalos oldala. `https://nitrogenproject.com/`. [Hozzárférés dátuma: 2024.05.29].

[12] YAWS - Yet Another Web Server. `https://erlyaws.github.io/`. [Hozzárférés dátuma: 2024.05.29].

[13] AT&T Labs Research. A graphviz hivatalos oldala. `https://graphviz.org/`. [Hozzárférés dátuma: 2024.05.29].

[14] Gajdos Sándor. *Adatbázisok*. A-SzínVonal 2000 Nyomdaipari Kft., 2019. [ISBN: 978-963-313-195-4].

[15] Ericsson AB. Erlang Mnesia dokumentáció. `https://www.erlang.org/doc/apps/mnesia/mnesia`. [Hozzárférés dátuma: 2024.05.29].

[16] Ericsson AB. Erlang ETS dokumentáció. `https://www.erlang.org/doc/apps/stdlib/ets`. [Hozzárférés dátuma: 2024.05.29].

[17] Ericsson AB. Erlang DETS dokumentáció. `https://www.erlang.org/doc/apps/stdlib/dets.html`. [Hozzárférés dátuma: 2024.05.29].

[18] RefactorErl Mnesia adatbázis dokumentáció. `http://pnyf.inf.elte.hu/trac/refactorerl/wiki/MnesiaDB`. [Hozzárférés dátuma: 2024.05.29].

[19] Ericsson AB. Erlang NIF-ek dokumentációja. `https://www.erlang.org/doc/system/nif`. [Hozzárférés dátuma: 2024.05.29].

[20] RefactorErl NIF adatbázis dokumentáció. `http://pnyf.inf.elte.hu/trac/refactorerl/wiki/NifDB`. [Hozzárférés dátuma: 2024.05.29].

[21] RefactorErl Kyoto Cabinet adatbázis dokumentáció. `http://pnyf.inf.elte.hu/trac/refactorerl/wiki/kcmini`. [Hozzárférés dátuma: 2024.05.29].

[22] Basho Technologies. A Riak KV hivatalos oldala. `https://docs.riak.com/riak/kv/latest/index.html`. [Hozzárférés dátuma: 2024.05.29].

[23] Neo4j Inc. A Neo4j gráfadatbázis hivatalos oldala. `https://neo4j.com/product/`. [Accessed: 2024.04.07].

[24] Neo4j Inc. A Cypher lekérdezőnyelv dokumentációja. `https://neo4j.com/docs/cypher-manual/current/introduction/`. [Hozzárférés dátuma: 2024.05.29].

[25] Jürgen Hölsch, Tobias Schmidt, and Michael Grossniklaus. On the performance of analytical and pattern matching graph queries in neo4j and a relational database. In Yannis Ioannidis, editor, *Proceedings of the Workshops of the EDBT/ICDT 2017 Joint Conference*, number 1810 in CEUR workshop proceedings, Aachen, 2017. CEUR-WS.org.

[26] Rahmatian Jayanty Sholichah, Mahmud Imrona, and Andry Alamsyah. Performance analysis of neo4j and mysql databases using public policies decision making data. In *2020 7th International Conference on Information Technology, Computer, and Electrical Engineering (ICITACEE)*, pages 152–157, 2020.

[27] Ericsson AB. Erlang map dokumentációja. `https://www.erlang.org/doc/system/maps`. [Hozzárférés dátuma: 2024.05.29].

[28] Dmytro Smyk. Measuring and graphing memory usage of local processes. `https://github.com/parikls/mem_usage_ui`. [Hozzárférés dátuma: 2024.05.29].

[29] L. Lövei, Z. Horváth, Z. Csörnyei T. Kozsik, R. Király, R. Kitlei, I. Bozó, Cs. Hoch, M. Tóth, D. Horpácsi, D. Drienyovszky, K. Horváth. *Implementation of Erlang Refactoring*. Eötvös Loránd Tudományegyetem, Programozási Nyelvek és Fordítóprogramok Tanszék, 10 2008.

[30] Zoltán Horváth, László Lövei, Tamás Kozsik, Róbert Kitlei, Anikó Víg, Thomas Nagy, Melinda Tóth, and Roland Király. Modeling semantic knowledge in erlang for refactoring. *Studia Universitatis Informatica*, 54:7–16, 01 2009.

[31] Ericsson AB. Erlang absztrakt szintaxisfák dokumentációja. `https://www.erlang.org/doc/apps/syntax_tools/erl_syntax.html`. [Hozzárférés dátuma: 2024.05.29].

[32] Petr Filip and Lukáš Čegan. Comparison of mysql and mongodb with focus on performance. In *2020 International Conference on Informatics, Multimedia, Cyber and Information System (ICIMCIS)*, pages 184–187, 2020.

[33] Wisal Khan, Teerath Kumar, Cheng Zhang, Kislay Raj, Arunabha M. Roy, and Bin Luo. Sql and nosql database software architecture performance analysis and assessments—a systematic literature review. *Big Data and Cognitive Computing*, 7(2), 2023.

[34] Didar Yedilkhan, Assel Mukasheva, Dariya Bissengaliyeva, and Yerulan Suynullayev. Performance analysis of scaling nosql vs sql: A comparative study of mongodb, cassandra, and postgresql. In *2023 IEEE International Conference on Smart Information Systems and Technologies (SIST)*, pages 479–483, 2023.

[35] Cajetan Rodrigues, Mit Ramesh Jain, and Ashish Khanchandani. Performance Comparison of Graph Database and Relational Database. *ResearchGate*, 05 2023.

[36] Apache Software Foundation. Apache Cassandra hivatalos oldala. `https://cassandra.apache.org/`. [Hozzárférés dátuma: 2024.05.27].

[37] ScyllaDB Inc. ScyllaDB hivatalos oldala. `https://www.scylladb.com/product/`. [Hozzárférés dátuma: 2024.05.27].

[38] ScyllaDB Inc. Apache Cassandra 4.0 Performance Benchmark: Comparing Cassandra 4.0, Cassandra 3.11 and Scylla Open Source 4.4. `https://www.scylladb.com/wp-content/uploads/wp-apache-cassandra-4-performance-benchmark-3.pdf`. [Hozzárférés dátuma: 2024.05.27].

[39] SonarSource. SonarCloud dokumentáció. `https://docs.sonarsource.com/sonarcloud/`. [Hozzárférés dátuma: 2024.05.28].

[40] SonarSource. SonarQube dokumentáció. `https://docs.sonarsource.com/sonarqube/latest/`. [Hozzárférés dátuma: 2024.05.28].

[41] Codacy. A Codacy statikus elemző dokumentációja. `https://docs.codacy.com/getting-started/supported-languages-and-tools/`. [Hozzárférés dátuma: 2024.05.29].

[42] Codacy. A Codacy statikus elemző adatbázisa. `https://docs.codacy.com/chart/requirements/#postgresql-server-setup`. [Hozzárférés dátuma: 2024.05.29].

[43] Perforce. A Klocwork elemző hivatalos oldala. `https://www.perforce.com/products/klocwork`. [Hozzárférés dátuma: 2024.05.28].

[44] MariaDB Foundation. A MariaDB adatbáziskezelő hivatalos oldala. `https://www.perforce.com/products/klocwork`. [Hozzárférés dátuma: 2024.05.28].

[45] Apache Software Foundation. Az Apache Lucene hivatalos oldala. `https://lucene.apache.org/`. [Hozzárférés dátuma: 2024.05.28].