Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

Kovács Tibor

NAGYMÉRETŰ SZEMANTIKUS ADATHALMAZOK TÁROLÁSI MEGOLDÁSAINAK TELJESÍTMÉNYKÖZPONTÚ ÖSSZEHASONLÍTÁSA

KONZULENS

Simon Gábor

BUDAPEST, 2017

Tartalomjegyzék

Összefoglaló

Napjainkban a különböző NoSQL alapú adatbázis-kezelők fénykorukat élik, egyre gyakrabban jelennek meg új, a korábbiaktól koncepcionálisan eltérő megoldások. Ez a fajta sokszínűség természetesen a gráfadatbázisokat területét sem kerülte el. A manapság egyre nagyobb népszerűségre szert tevő multimodális adatbázis-kezelők megjelenésével pedig a gráf alapú megoldások olyan klasszikus területeken is megjelentek, mint a relációs adatbázis-kezelők, hiszen például a Microsoft SQL Server 2017 egyik nagy újdonsága a gráf jellegű lekérdezések támogatása volt.

Mindezzel párhuzamosan, ahogy a globalizáció is kezd kiteljesedni, soha nem látott mértékben növekszik az információ mennyisége és fontossága is, így természetszerűen felvetődik az információ érvényességének, vagy éppen hitelességének kérdése, ennek számítógépes ábrázolásának mikéntje, ami tovább vezet bennünket a reifikáció általános problémájához, azaz ahhoz, hogy hogyan tárolunk olyan információt, ami maga is információt jellemez, ír le.

Tekintve, hogy ma egy rendszer használhatósága szempontjából a teljesítmény fontossága összemérhető a megvalósított funkcionalitás hasznosságával, így kritikus, hogy az ilyen jellegű információkat a lehető legnagyobb hatékonysággal tudjuk felhasználni, aminek legelső eszköze a megfelelő reprezentációs modell és fizikai tárolási mód kiválasztása.

Dolgozatomban első sorban teljesítmény szempontból hasonlítom össze a reifikációra adott klasszikus modellezési megoldásokat a jelenleg elérhető, gráf jellegű adatfeldolgozást lehetővé tevő adatbázis-kezelőket felhasználva. Az eredmények alapján megválaszolhatóvá válik az a kérdés, hogy egy adott jellegű reifikált adathalmazt hogyan kell ábrázolni, és milyen adatbázis-kezelőt kell használni ahhoz, hogy egy előre definiált jellegű családba tartozó lekérdezések a lehető leggyorsabban végrehajthatók legyenek rajta.

Abstract

Nowadays as the different NoSQL based database managers are living their golden age, new solutions appear more frequently using completely different concepts. This kind of versatility does not avoid the graph databases as well. With the appearance and growing popularity of the multimodal database managers the graph-based solutions are beginning to appear in such classic fields as the relation database managers as one of the biggest new feature of the Microsoft SQL Server 2017 is the support of the graph-like queries.

At the same time, as the globalization is slowly reaching its totality, the quantity and the importance of information is growing in an unprecedented way, so it's quite natural that the question of validity or reliability arises, and how we represent it in computers, which lead us straight towards the general problem of reification i.e. how we store information about information.

Regarding the usability of a system today, its performance is almost as important as the realized functionality, so it's critical to use this kind of information in the most efficient way possible. The first step is to select the proper representation model and physical storage mode.

In my paper, I compare the classic modeling solutions to the problem of reification on the available database managers which offer graph-based data processing. My focus is on the performance of the different models on different storages. The measurement results will help to answer what database manager and data model should be used to gain the best performance on a reified dataset with predefined characteristic used by queries belonging to a predefined kind of query group.

1 Bevezetés

A big data, a gépi tanulás és az adatbányászat soha nem látott népszerűséget ért el, amik során a nagyméretű adathalmazok komplex analízise került előtérbe. Ez a fajta komplex analízis az adatok közötti összefüggésekre összpontosít, ami a relációs adatmodellben csak nehézkesen valósítható meg. [7]

A gráfadatbázis használata számos előnnyel bír a manapság elterjedt relációs adatbáziskezelőkkel szemben, melyek közül az egyik legfontosabb, hogy segítségükkel az adatok közötti kapcsolatok jóval sokrétűbben, és az emberi gondolkodás számára sokkal természetesebben kifejezhetők, így ideális választások lehetnek komplex, sűrű kapcsolatrendszerrel rendelkező adathalmazok számára. [1] Ez a tulajdonság teszi kimondottan alkalmassá szemantikus hálók kezelésére.

A napjainkban is zajló információs robbanás következménye, hogy az információ érvényessége és hitelessége korábban nem felmerülő problémákat vet fel, melyek közül az egyik legalapvetőbb a reifikáció problémája. A reifikáció megoldására a szemantikus adathalmazokat olyan módon kell reprezentálni, hogy azok a képesek legyenek reifikált állításokat leírni, ugyanakkor ezt a lehető leghatékonyabban tegyék meg. Mivel manapság a rendszerek sebességbéli teljesítménye – a funkcionalitás mellett – a legfontosabb metrikája, így munkám középpontjában ezen probléma modellezési lehetőségek teljesítménye van.

Dolgozatom elkészítése során azt a célt is kitűztem, hogy megállapítsam, a reifikációt megvalósító modellek teljesítménye hogyan viszonyul egymáshoz, melyiknek a legnagyobb a teljesítménye, egy előre definiált lekérdezéscsaládon mérve.

A gráfadatbázisok területe ma is aktívan kutatott terület [20][21][22][23], számos cikk és publikáció jelenik meg ezzel kapcsolatban. A reifikáció ezzel szemben egy lényegesen kevésbé népszerű téma. A gráfadatbázisokkal kapcsolatos tudományos írások nagyrésze vagy nem foglalkozik egyáltalán a reifikáció kérdésével, vagy puszta elméleti úton elemzi annak lehetőségeit [24], nem található olyan, amely a különböző reifikált modellezési eljárások teljesítményét gráfadatbázis specifikusan számszerűen összehasonlítja (más adatmodelleket nézve már vannak ilyen publikációk), így munkámmal ez a hiányt szeretném pótolni.

Munkám első részében bevezető jelleggel bemutatom és jellemzem a gráfadatbázisokat általánosságban, ami a vizsgált adatbázis-kezelő rendszerek specifikus bemutatása követ. Ebben a részben minden rendszert sorra véve bemutatok, kiemelve az egyes rendszerek eltéréseit az általános gráfadatbázis koncepciótól, illetve a többi rendszertől.

A vizsgált rendszerek bemutatása után következik a reifikáció problémájának felvetése – gráfadatbázis illetve szemantikus adathalmaz vonatkozásában – és klasszikus megoldásinak ismertetése, röviden bemutatva az ismertetett modelleket, kihangsúlyozva a köztük levő szemléletbéli különbségeket.

Az elméleti ismertetés után az elkészített szoftverrendszer leírása következik, melyben bemutatom a mérési környezetet, a mérést végző rendszer architektúráját és munkafolyamatát. Itt ismertetem, hogy működik az elkészített rendszer, mely komponensei miért felelősek.

Az elkészített rendszer pontos ismertetése után következik a mérési eredmények elemzése. Ebben a részben mutatom be a konkrét lekérdezéscsaládot, amire a mérések történtek, illetve ebben a részben mutatom be a kapott eredményeket, a hozzá tartozó elemzéseket, amik során megválaszolom azokat a kérdéseket, melyeket a dolgozat készítésekor célul tűztem ki.

Zárásként ismertetek néhány tovább gondolási lehetőséget, melyeken elindulva érdemes lehet további kutatásokat kezdeni, majd összefoglalom a kapott eredményeket.

2 Gráfadatbázisok

2.1 Történetük

Annak ellenére, hogy a gráfok fogalmát és koncepcióját már Leonhard Euler is ismerte a XVIII. század végén, és a matematikusok – kihasználva, hogy a valóságban felmerülő problémák óriási részének megoldásához kiváló eszköznek bizonyultak – a XIX. és XX. század során gazdag és digitális környezetben is jól használható eszköztárt építettek köré, az adatbázis-kezelés területén történő megjelenésük egészen a ’80-as évekig végéig váratott magára, és ekkor is főként csak elméleti alternatívaként állt az akkor már egyeduralkodónak számító relációs modellel. [1]

A ’90-es években a web és az informatika általános elterjedésével egyidőben fokozatosan kezdtek mutatkoztak annak a jelei, hogy egy adatmodell nem képes az összes, egyre sokoldalúbb adatkezelést igénylő feladat hatékony megoldására [2] (amely sejtést némileg megerősít az, hogy ilyen egyetemes adatmodellt azóta sem sikerült senkinek találnia). Ennek a felismerésnek a kapcsán indult el a NoSQL mozgalom [3], mely manapság minden korábbinál nagyobb népszerűségnek örvend.

A NoSQL családba tartozó adatbázis-kezelők, így a gráfadatbázis-kezelők is, gyakran csak bizonyos típusú feladatok megoldására képesek hatékonyan, de cserében általában nagyon jó horizontális skálázhatóságot kínálnak, ami a manapság divatosnak számító felhő-alapú digitális infrastruktúrák esetén egy nagyon előnyös tulajdonság [1]. Emellett a relációs modell számos kötöttségét is feloldják (például a kapcsolótábla kényszert több-több kapcsolat esetén, az előre definiált adatsémákat stb.), így például nagyon komplex adatmodell esetén ideális alternatívaként állnak a relációs rendszerekkel szemben.

2.2 Elméleti háttér

Mint a történeti áttekintésből látszik, a gráfadatbázisok (és általában a NoSQL is) területe – még informatikai léptékkel nézve is – egy viszonylag fiatal terület, így még nincs egy olyan egységes közös nevező, mint a relációs világban az SQL. Egy ilyen közös nevező nélkül pedig rendszeresen tűnnek fel újabb és újabb megoldások, melyek akár csak szintaktikailag, akár koncepcionálisan lényegesen eltérnek a korábbi rendszerektől.

Gráfadatbázis alatt általánosságban tehát olyan rendszert értünk, ami gráf adatmodellt használ fel adatbázis-kezelő funkciók megvalósítására. Az adatmodell fogalmi megközelítése azonban manapság szintén alapvető változásokon megy át. A Codd által definiált adatmodell fogalom [1] megköveteli az adaton végezhető műveletek definiálásán túl – többek között – az adatstruktúra típusok definiálását is. Manapság ez utóbbi feltételt már nem feltétlenül tekintjük a definíció szükséges részévé, aminek két alapvető oka van:

- Az adat logikai kezelésének és fizikai kezelésének teljes szétválasztása. Ez azt jelenti, hogy az adatkezelő műveletek interfészként működnek a ténylegesen megvalósított műveletek felett, így elrejtve annak részleteit (tehát előfordulhat olyan, hogy gráf alapú lekérdést küldünk egy adatbázis-kezelő rendszernek, például Cypher nyelven, ami először leképződik rendszeren belül relációs lekérdezésre, majd megtörténik a kiértékelése a relációs algebra szabályai szerint, végük az eredményhalmaz olyan formában jön vissza, mintha az eredeti gráf lekérdezés futott volna le).
- A multimodális adatbázis-kezelő rendszerek megjelenése. Az elmúlt években megjelentek, és egyre nagyobb népszerűségre tettek szert az ún. multimodális rendszerek, melyek egy rendszeren belül több különböző adatmodellt is támogatni tudnak. Az ilyen rendszerek megvalósítása maga után vonja az előző pontban leírtakat, hiszen másként nem értelmezhető, hogy hogyan lehet ugyanazt a rendszert gráf-alapú és relációs lekérdezéssel is egyaránt megszólítani.

A definíció ilyen jellegű egyszerűsítéséből azonban több dolog is következik. Egyrészt, hogy akkor tekint egy rendszert gráfadatbázis-kezelőnek, ha rendelkezik gráf-alapú adatkezelő interfésszel (például lekérdező nyelvvel). Másrészt, hogy annak eldöntése, hogy egy rendszer gráfadatbázis-e, eldönthető egy Turing-tesztszerű vizsgálattal [4] is (azaz akkor tekintünk egy rendszert gráfadatbázis-kezelőnek, ha ugyanarra a lekérdezés inputra ugyanaz az output érkezik, mint egy elfogadottan gráfadatbázis-kezelő rendszernek, feltételezve egy közös lekérdező nyelvet). Harmadrészt pedig, hogy olyan rendszereket is gráfadatbázis-kezelőnek tekint (vagy emiatt multimodális rendszernek), amiket a korábbi megközelítés nem. Ilyen például a Microsoft SQL Server 2017, aminek az egyik nagy újdonsága a Cypher-szerű [5] gráf

illesztések támogatása lekérdezésekben [6], miközben a háttérben továbbra is dominánsan relációs technológia szerint működik.

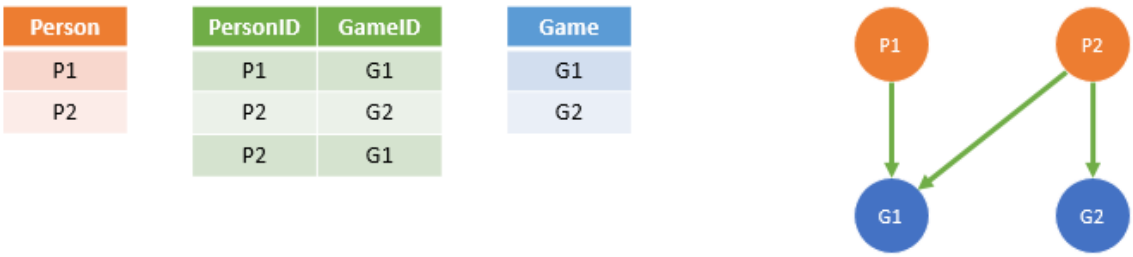
Dolgozatomban ezt az újabb, lényegesen megengedőbb megközelítést tekintem irányadónak, hiszen ez az adatmodell fogalom egy jóval sokoldalúbb és modernebb megközelítésének tekinthető.

2.3 Adatmodellek

A gráf-alapú adatmodellek – sokszínűségük ellenére – általánosan elfogadott közös pontja, hogy az adatkezelés (adatmanipuláció és adat lekérdezés) központjában a matematikai gráfok valamilyen (kiegészített) interpretációja áll.

A matematikában gráfnak a $G(V, E)$ rendezett párokat nevezzük, ahol V egy nem üres halmaz (a gráf csúcsai), míg E egy V-beli elemekből képzett – nem feltétlenül egyedi – párokból álló halmaz (a gráf élei).

Ez az egzakt, általános definíció tökéletesen alkalmassá teszi a gráfokat egy modellezett világ objektumainak (V halmaz) és a köztük levő kapcsolatok (E halmaz) leírására, azonban az adatbázisokban megszokott más adatok leírására (például az objektumok típusára és tulajdonságaira, a kapcsolatok szemantikájára) közvetlenül nem alkalmas [2], de különböző bővítésekkel ezek az információk is könnyen leírhatóvá válnak. Az ilyen jellegű bővítési lehetőségek száma szinte korlátlan, így teljeskörű felsorolás nem is adható róluk. A vizsgált rendszerek szempontjából fontos bővítési lehetőségeket a tulajdonsággráf modellnél ismertetem.



1. ábra: A relációs és a gráf alapú modellezés összehasonlítása

Az ismertetett adatmodell és gráf definíciókból következik, hogy a gráfadatbázisok kapcsán az elvégezhető műveleteket a matematika területéről átvett gráf-operátorok jelentik. [1] A legáltalánosabb körben megvalósított műveleteket tehát a gráfbejárás, a szomszédságok és más globális gráfmetrikák adják (mint például az általános fokszám).

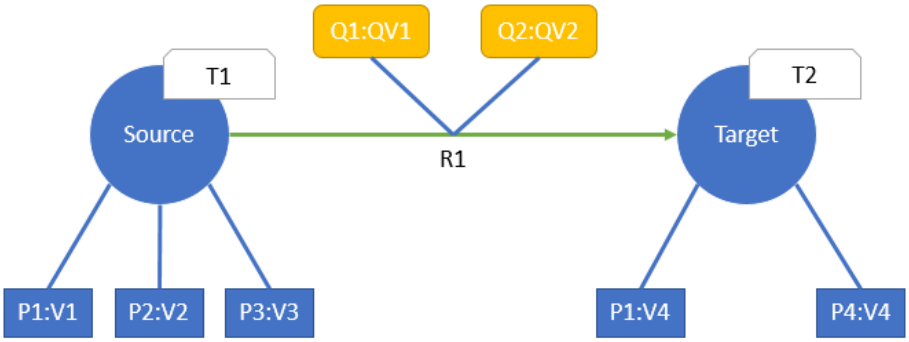
2.3.1 Tulajdonsággráf modell

A konkrét gráfadatmodell interpretációk ismertetése között nem csak azért van az első helyen a tulajdonsággráf modell [2] [7], mert ezt valósítja meg a jelenlegi legnépszerűbb tisztán gráfadatbázis-kezelő rendszer [8], a Neo4j, hanem mert ez operál a legtöbb bővítési lehetőséggel, a későbbiekben vizsgált rendszerek interpretációi mind megkaphatók ennek a modellnek az egyszerűsítéseként.

Az első alkalmazott bővítés a csúcs és él címkék bevezetése. A csúcs címke formálisan $f_{Nlabel}: V \rightarrow T^k$, az él címke $f_{Elabel}: E \rightarrow R$ alakban írható fel, ahol T a gráf csúcsai által reprezentált objektumok lehetséges típusainak halmaza, míg R a lehetséges kapcsolattípusok halmaza. Látható tehát, hogy ezen bővítés lehetővé teszi, hogy az élekhez pontosan egy, míg a csúcsokhoz több típust rendeljünk. Fontos azonban, hogy ez a hozzárendelés semmilyen séma kényszert nem volt maga után, ellentétben a relációs adatmodell sémadefinícióival szemben.

A gráfok matematikai definíciója nem teszi lehetővé az objektumok és kapcsolataik tulajdonságainak leírását, ezt a problémát oldja meg a tulajdonságok bevezetése. A tulajdonságok a modellben kétféle módon jelennek meg. Az egyik megjelenés formálisan $f_{property}: V \rightarrow P_k \times P_v$, a másik $f_{qualifier}: E \rightarrow Q_k \times Q_v$ alakban definiálható, azaz minden csúcshoz és élhez hozzárendelhetünk kulcs-érték párok formájában tulajdonságokat, melyek csak az adott csúcsot vagy élt jellemzik.

Mivel a kapcsolatok szemantikáját rendszerint úgy választjuk meg, hogy az egyirányú kapcsolatot határoz meg, így valamennyi gráf-alapú adatmodell irányított gráfokat használ. Az irányítás egy olyan $f_{direction}: E \rightarrow \{0; 1\}$ függvény, ami minden élhez hozzárendel egy irányt, és az élt meghatározó két csúcs között – más élt nem feltételezve – csak az egyik irányban van szomszédsági viszony. Például egy település és egy ember között a „születési hely” kapcsolatnak csak egy irányban van szemantikailag értelme. Ezzel kapcsolatban azonban fontos megjegyezni, hogy ennek az iránynak a megválasztása modellezési kérdés, hiszen előfordulhat, hogy ugyanazon adathalmazt használva, az alkalmazás szempontjából előnyösebb a „született itt” kapcsolatot felvenni, ami az ilyen szemantikájú összeköttetések irányítását megfordítja.



2. ábra: A tulajdonsággráf modell bővítései

A megvizsgált rendszerek közül ezt az adatmodellt valósítja meg a már említett Neo4j, illetve a TinkerPop gráf stack-et implementáló Titan DB is, melyeket azonban sajátosságaik miatt külön részben ismertetek.

2.3.2 RDF modell és a SPARQL

A ’90-es években megszülető szemantikus web koncepció alapja a nagyméretű, félig strukturált adatokból álló, sűrű kapcsolati rendszerrel rendelkező adathalmaz. A óriás méretű adathalmaz, szigorúan véve a sémamentesség és a nagyon sűrű és szövevényes kapcsolati rendszer reprezentálására ideálisnak tűnt a gráf adatmodell. Annak érdekében azonban, hogy a gráf a gépek számára is egyszerűen értelmezhető legyen, a tulajdonsággráf modellnél lényegesebb egyszerűbb reprezentációs modellre volt szükség, amit a szabványosított Resource Description Framework (RDF) valósított meg [14].

Az RDF adatmodell [10] lényegesen egyszerűbb a tulajdonsággráf modellnél, hiszen az ott bevezetett bővítések közül csak az irányított gráf jelleget és az él címkék használatát alkalmazza. Az adatmodell különlegessége, hogy bevezeti az erőforrás és az állítás fogalmát, és oszthatatlan egységévé is azt teszi, aminek egyik közvetlen következménye, hogy csúcsot önmagában (kapcsolódó él nélkül) nem lehet hozzáadni a gráfhoz (csak állítás részeként).

Egy RDF adatbázisban kétféle csúcs található meg:

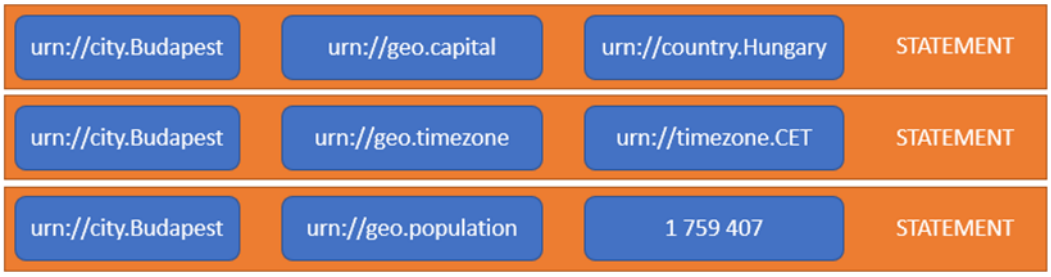
- Egyszerű nyers adat, azaz egyszerű szöveges, numerikus vagy akár logikai érték. Ezek az értékek felelősek a tulajdonságok értékének tárolására.
- Erőforrás, azaz olyan modellezett objektum, mely globális egyedi azonosítóval (URI) rendelkezik.

Ezek után már bevezethető az állítás fogalma, ami egyfajta megszorításként értelmezhető a gráf éleire nézve. Az irányított él és az azt meghatározó két csúcspont alkotja az állítás három komponensét. A kiinduló csúcspontot alanynak (subject), a cél csúcspontot tárgynak (object), míg a köztük húzódó irányított él típusát állítmánynak (predicate) nevezzük. A korlátozás arra vonatkozik, hogy egy állítás alanya és állítmánya csak erőforrás lehet.

3. ábra: Az RDF állítás modellje

Az RDF ilyen módon a gráf éllistás reprezentációjára hasonlít, melyben minden mellett tárolva van az állítás típusa is. Ez a reprezentáció lehetővé teszi nagyméretű adathalmazokban objektumok egyszerű, rugalmas összekötését az URI-kon keresztül, ugyanakkor a gépek számára is egyértelmű interpretációs lehetőséget nyújt, hiszen minden állítás adott szemantika és struktúra szerint épül fel.

Az úgynevezett „triple store” vagy RDF adatbázis-kezelő rendszereket [43] speciálisan az ilyen RDF hármasok hatékony tárolására és lekérdezésére fejlesztették ki, emiatt a gráfadatbázisok egy speciális esetének tekinthetők. Ilyen rendszer például a [13]-ban is vizsgált Virtuoso [25].



4. ábra: Részlet egy RDF adatbázisból

Az általam vizsgált adatbázis-kezelő rendszerek közül az RDF modellt a Wikidata [15] által is használt Blazegraph valósítja meg a szabványos SPARQL [16] lekérdezőnyelv segítségével.

A SPARQL egy szabványosított, deklaratív jellegű mintaillesztő nyelv [16]. A segítségével lehetőség van leírni egy változókat is tartalmazó gráf mintát, amit az összes

lehetséges illeszkedés esetén behelyettesít a rendszer, így lehetővé téve a gráf sokoldalú lekérdezését. Erre a példa a következő lekérdezés, ami Bob barátainak nevét kérdezi le az adatbázisból:

```
SELECT ?friend_name
WHERE
{
  <http://example.edu/people/Bob> <http://example.edu/friend> ?friend .
  ?friend <http://example.edu/name> ?friend_name
}
```

2.4 Adatbázis-kezelő rendszerek

2.4.1 TinkerPop gráf stack és a Titan DB

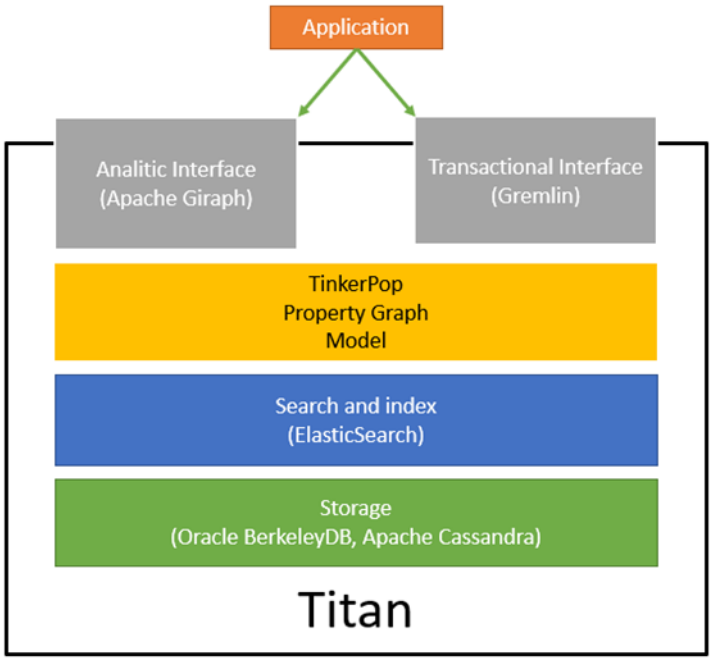
Ahogy a gráfadatbázis-kezelők elkezdtek népszerűvé válni, folyamatosan jelentek meg újabb és újabb rendszerek, melyek egyáltalán nem voltak kompatibilisek egymással (tekintve, hogy az adatmodellen túl a lekérdező nyelv is a legtöbb esetben alapjában eltérő volt), ami megnehezítette elterjedésüket az igazán komoly üzleti alkalmazások körében. Ezt a problémát volt hivatott megoldani a TinkerPop gráf stack, ami a relációs Java Database Connectivity (JDBC) adatbázis elérés egy gráfos [26] megfelelőjének tekinthető.

A TinkerPop gráf stack egy egységesített adatkezelési metodikát definiál a gráfadatbázisok számára. [9] A dolgozat szempontjából két elemét fontos kiemelni:

1. Gremlin Structure API (korábbi verziókban Blueprints), ami – a JDBC-hez hasonlóan – interfészeket és típusokat definiál, amelyek segítségével a gráfok kezelése egységes módon történhet a megvalósító konkrét rendszertől függetlenül. [27]
2. Gremlin, ami a gráf stack-hez igazodó lekérdező nyelv [17]. Fontos kiemelni, hogy a többi vizsgált lekérdező nyelvtől eltérően a Gremlin imperatív, és nem deklaratív jellegű lekérdezések megfogalmazását teszi lehetővé.

Ennek a gráf stack-nek az egyik megvalósítása a Titan DB adatbázis-kezelő. A Titan DB nagyméretű adathalmazok elosztott tárolására és feldolgozására lett kifejlesztve. A Titan DB igazi ereje azonban az integrációs lehetőségeiben rejlik, ugyanis szinte minden adatbázis-kezelő funkcionalitást integráció révén éri el [28]:

- Nem definiál saját adattárolási megoldást, hanem lehetőséget ad arra, hogy külső tárolási háttérszolgáltatást válasszon a felhasználó, ami az igényeihez a legjobban illik (ilyen tárolási szolgáltatás például az Oracle BerkeleyDB, vagy az Apache Cassandra)
- A gráf analitikai és keresési funkcionalitásokat is külső szolgáltatások (például ElasticSearch, Apache Giraph) integrációjával éri el
- Ezen kívül nem definiál saját lekérdező nyelvet és adatmodellt sem, hiszen a TinkerPop gráf stack megvalósítása magával vonja a Gremlin és a tulajdonsággráf modell integrációját



5. ábra: A Titan adatbázis-kezelő architektúrája és integrációs lehetőségei

A vizsgált rendszerek közül a TinkerPop gráf stack-et megvalósítja a Titan-on kívül még a Microsoft elosztott, felhő központú multimodális rendszere, a CosmosDB (annak is a gráf-alapú része), valamint a Blazegraph is.

2.4.2 Blazegraph

A Blazegraph maga egy multimodális rendszer, ami mind az általános gráfadatbázisok, mind az RDF adatbázisok használatát támogatja, így kategorizáláskor a kettő közé tehető.

A korábban említetteknek megfelelően támogatja a TinkerPop gráf stack-et, és így a tulajdonsággráf modellt, amihez Gremlin nyelven lehet utasításokat küldeni.

Az ezután bemutatott reifikációval kapcsolatban érdekes tulajdonsága, hogy nem csak az RDF-et, hanem annak egy kiterjesztett változatát, a Reification Done Right-ot (RDR) is támogatja [29], ami – a tulajdonsággráf modellhez hasonlóan – azzal terjeszti ki a RDF-et, hogy az állítások önmagukban (azaz hármasként) is szerepelhetnek állítás alanyaként, ami élekhez rendelt tulajdonságokra képződik le.

Ezen kívül lehetőség van RDF adatbázisként (triple store-ként) is használni, ami a szabványos SPARQL nyelv, és az RDR bővítéseknek köszönhetően a SPARQL* nyelv támogatását is magával vonja.

Fontos azonban kiemelni ezzel kapcsolatban, hogy ez a két eltérő koncepció nem zárja ki az egymást, lehetőség van arra, hogy ugyanazt az adatbázist egyik feladathoz Gremlin-en, másik feladathoz SPARQL-en keresztül szólítjuk meg [30].

A mérések során két másik TinkerPop alapú adatbázis-kezelő rendszert is megvizsgáltam, így a Blazegraph esetén csak a szabványos (tehát nem RDR) RDF adatbázis-kezelő funkcionalitását vizsgáltam a rendszernek – azaz csak a SPARQL-es API-t.

2.4.3 Neo4j

A Neo4j a jelenleg a legnépszerűbb elérhető gráfadatbázis [8]. A tulajdonsággráf modell valamennyi kiterjesztését támogatja. A rendszer különlegessége, hogy az adatok tárolása fizikai szinten is gráfként történik, láncolt listák struktúrájaként. [7]

További különlegessége, hogy saját deklaratív lekérdezőnyelvet definiál, a Cypher-t, ami jelenleg a SPARQL egyetlen lényeges deklaratív konkurensének tekinthető [31]. Ezt a nyelvet a tulajdonsággráf modellhez, és így az általános gráfadatbázis műveletekhez fejlesztették ki. A nyelv a gráfillesztést jól olvasható szintaxissal írja le, a csúcsok közötti kapcsolatokat ASCII karakterekből kirajzolt nyilak formájában fejezhető ki. Erre példa a korábban SPARQL-ben megfogalmazott lekérdezés Cypher-es megfelelője:

```
MATCH (bob:Person)-[:FRIEND]->(friend:Person)
WHERE bob.name = ’Bob’
RETURN friend.name;
```

Nagyon jó dokumentációval rendelkezik, emiatt könnyű beüzemelni, és elkezdeni vele a munkát, viszont a relációs világban megszokott részletes teljesítmény hangolási megoldásokkal nem rendelkezik.

2.4.4 CosmosDB

A CosmosDB felhőközpontú multimodális adattárolási megoldása, ami a régió szintű elosztott tárolást is lehetővé teszi. Korábban a dokumentum és a kulcs-érték pár alapú adattárolást támogatta csak, azonban a közelmúltban elérhető vált gráf adatmodell támogatása is (és a közeljövőben további modellek támogatása is várható). [32]

A bevezetett Gráf API a TinkerPop gráf stack-jére épül, így a Gremlin lekérdező nyelvvel lehet lekérdezéseket intézni felé. A rendszer automatikus csúcspont indexeléssel és több konzisztenciaszinttel is rendelkezik, ami nagyfokú konfigurálhatóságot eredményez. [32]

Abból kifolyólag, hogy a rendszer felhő szolgáltatás keretében érhető el, mind a rendszer teljesítménye, mind az elérhető tárhely a futás során dinamikusan változtatható az igények szerint, amire semelyik másik rendszer nem képes.

3 Reifikáció

3.1 Problémafelvetés

A ’90-es években meginduló információs robbanás következtében az interneten elérhető tudásanyag mennyisége óriási mértékben megnőtt, ez a növekedés manapság is tart (gyorsul). A nagy információmennyiség hatására még az olyan egyszerűnek tekinthető problémák is, mint a keresés, hatalmas idő ráfordítást igényelnének gépi segítség nélkül, ami magával vonta a szemantikus web koncepciójának megszületését. Ezekkel összefüggésben számos olyan konkrét probléma felmerült, melyek egy közös alapproblémára vezethetők vissza.

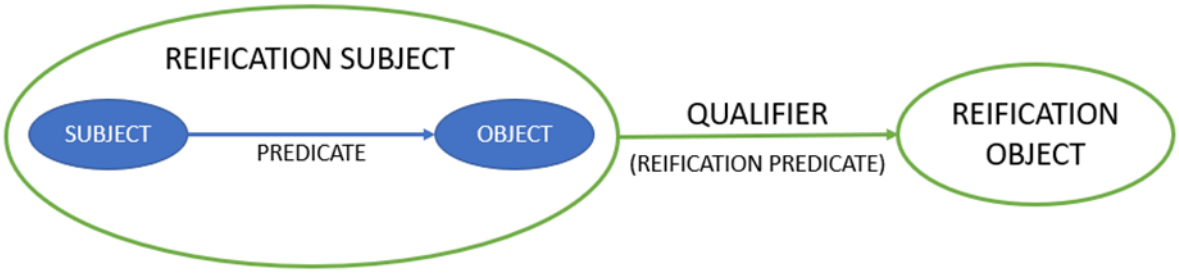
Az egyik ilyen problémaként az információ érvényességet emelném ki. Egy ilyen gyorsan változó információs rendszer természetes velejárója, hogy a benne tárolt adatok valamilyen okból érvényüket vesztik. Például tekintsük azt az állítást, hogy „Barack Obama az Amerikai Egyesült Államok elnöke.” Ez az állítás 2009.01.20. és 2017.01.20. között igaz volt, a dolgozat írásakor azonban már nem. Annak érdekében, hogy az ezt az állítást tartalmazó rendszer konzisztens maradhasson, az új elnök beiktatását követően változtatni kell az adatbázison, amire több lehetőség is van:

- Töröljük az állítást az adatbázisból, ami kizárja annak lehetőségét, hogy a későbbiekben az elnök személyével kapcsolatos historikus információkat kérdezzünk le. Mivel a legtöbb vállalat múltbéli adatok alapján hozza döntéseit, így ez a lehetőség leginkább csak elméleti szinten merülhet fel.
- Módosítjuk az adatbázist amint az új elnök beiktatása megtörtént úgy, hogy Barack Obama kapcsolatát az Amerikai Egyesült Államokkal exelnök kapcsolatra állítjuk. Ennek a módszernek az alkalmazása azonban komplex kapcsolati háló esetén reménytelen, ha a gráf szintű konzisztencia megtartása fontos (ami az adatbázisok talán legfontosabb tulajdonsága)
- A rendszer kialakításakor már eleve olyan modell szerint történik az adatok tárolása, ami támogatja kiegészítő információk tárolását is. Ekkor mindössze egy hozzáadás történik az adatbázisban, ami komplex kapcsolatok esetén is könnyen elvégezhető, és információveszteség sem lép fel.

A másik problémát az információ hitelessége vagy megbízhatósága jelenti. Az információmennyiség növekedésével elkerülhetetlenné válik, hogy ugyanarról a dologról több, akár egymásnak ellentmondó állítást is tartalmaz. Például az egyik forrás szerint Barack Obama, a másik forrás szerint Donald Trump az Amerikai Egyesült Államok elnöke. Ilyen helyzetben dönteni kell, hogy mely forrásból származó információt fogadjuk el hitelesnek. Ezt nehezíti, hogy egy állításnak nem része, hogy mely forrásból származik. Ebben az esetben is ésszerű gondolatnak tűnik, hogy kiegészítő információkat adjunk az állításokhoz, általánosabban kifejezve állításokat tegyünk állításokról, azaz reifikáljuk.

3.2 Modellezés

A reifikáció, vagy „magasabb rendű állítás” [10] fogalma tehát – ha az RDF modell fogalmait vesszük alapul – olyan modellezési szituáció, amikor egy állítás egésze egy másik állítás alanyává válik.



6. ábra: A reifikáció általános sémája

A korábban bemutatott modellekre nézve ennek két fontos következménye is van. Az egyik, hogy a tulajdonsággráf modell rendelkezik azzal az eszközkészlettel, ami a reifikációt közvetlenül támogatja, hiszen az élekhez hozzárendelhető tulajdonságok (minősítők) segítségével tetszőleges élhez, azaz állításhoz, kulcs-érték párok formájában további állítások rendelhetők, mely – lévén, hogy élhez tartoznak – nyilvánvalóan magát a teljes állítást jellemzi.

A másik, jóval meglepőbb következmény, hogy a szemantikus web egyeduralkodó modellezési technológiája, az RDF közvetlenül nem rendelkezik ilyen kifejező képességgel. A később bemutatott standard és proxy modellek ugyan csak az RDF szabvány által is használt eszközkészletet használják, azonban szemantikailag

némileg mást írnak le: az adatbázis csak az állításról tett állítást tartalmazza, magát a reifikált állítást nem. [10]

Az imént bemutatott két problémán kívül még számtalan eset vezethető vissza arra, hogy az RDF modell szerinti állítás túl egyszerű ahhoz, hogy a modellezés alapvető megváltoztatása nélkül metainformációkat rendelhessünk állításokhoz. Ez az RDF korábban bemutatott azon sajátosságára vezethető vissza, hogy állítás alanya csak erőforrás lehet.

Ennek ellenére mégsem igaz az, hogy az RDF fogalomrendszerével a reifikáció lényegileg nem megvalósítható. A viszonylag szerény szakirodalom ellenére számos megoldás született a probléma megoldására. Ezen modellezési minták használatakor azonban két dolgot érdemes figyelembe venni:

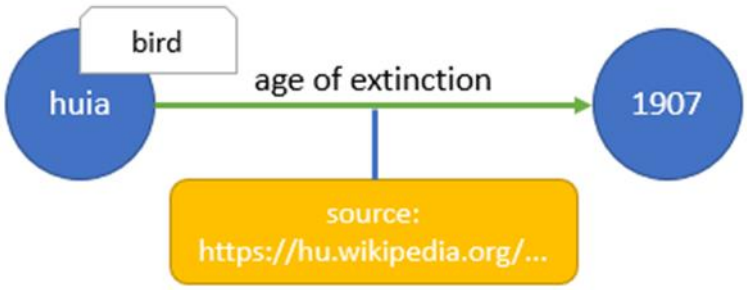
- Mivel a legtöbb megoldás az adatbázis struktúrájának alapvető megváltoztatását jelenti, így akkor lehet hatékonyan alkalmazni őket, ha már az adatbázis séma tervezésekor figyelembe vesszük azok jellemzőit. Ez különösen akkor jelenthet óriási problémát, hogy az adathalmazhoz – a szemantikus web koncepcióján tovább haladva – ontológia [10] is tartozik, hiszen az átalakítás sokszor a teljes ontológia újra írását jelentené, míg néhány modell esetén a veszteségmentes átalakítás sem lenne megoldható.
- Valamennyi megoldás több absztrakciós szintet egyszerre használ fel a modellezéshez. Ez azt jelenti, hogy nem csak a modellezett világ fogalmai (például ember, ismeri kapcsolat stb.) jelennek meg, hanem az adott modell elemei is (például állítás), amivel pont a gráfadatbázisok azon – talán legnagyobb – előnyeit veszítjük el, hogy a kapcsolatok modellezése egyszerű, az emberi gondolkodást követi.

Annak érdekében, hogy az egyes modellek könnyen összehasonlíthatók legyenek, minden modellt egy közös példán keresztül mutatok be, a következő állítás lesz:

„A https://hu.wikipedia.org/wiki/Huja_(mad%C3%A1r) oldal szerint a huja madár 1907-ben kihalt.” állítás lesz.

3.2.1 Éltulajdonság alapú reifikációs modell

Az éltulajdonság modell [13] áll talán a legközelebb az emberi gondolkodáshoz, ezt támogatja a tulajdonsággráf modell is. A korábbiaknak megfelelően ez megközelítés nem képezhető le RDF-re.



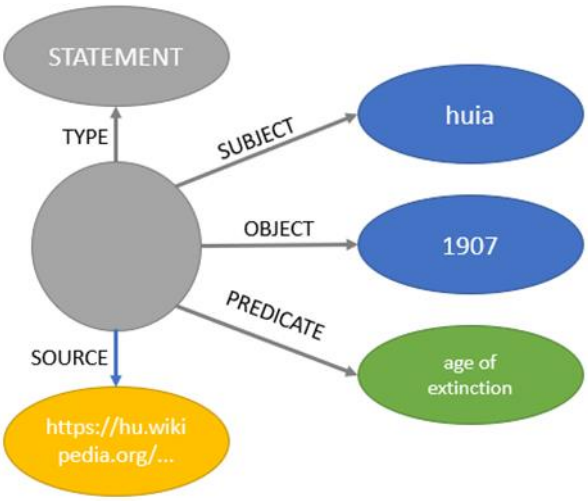
7. ábra: A példaállítás éltulajdonság modellben reprezentálva

A mellékelt képen is látható, hogy az állítást reprezentáló él továbbra is azon két objektum között húzódik, amiket a reifikáció nélkül is összekötne, a reifikációs állítások ezen él tulajdonságainak formájában jelennek meg. Annak ellenére, hogy ez a megközelítés a legegyszerűbb, egy nagyon komoly hátránya:

A reifikált állítások kulcs-érték párok formájában, tulajdonságként jelennek meg, nem csúcspontként, így nem lehet őket további állítások alanyaként felhasználni. Ez a példában azt jelenti, hogy sem a „forrás” szemantikájú kapcsolatról, sem a Wikipedia oldalról nem tudunk további állítást felvenni. Természetesen fel lehet venni olyan csomópontot is, ami az oldalt reprezentálja, ez viszont adatbázis szinten semmilyen kapcsolatban nincs a reifikált tulajdonsággal. Emiatt ez a modell nem alkalmas olyan komplex hálók leírására, ahol a kapcsolatokban a reifikált tulajdonságok is részt vesznek. Ennek egy közvetlen következménye, hogy ez a modell nem támogatja a többszintű reifikációt sem.

3.2.2 Standard reifikációs modell

A reifikáció kiemelt fontosságát jelzi, hogy már az RDF megalkotásakor beépítettek olyan szabványos megoldásokat, amik erre jelentenek megoldást [10]. A standard reifikációs modell [13] a RDF szabványban rögzített osztályokat és kapcsolatokat használja fel.



8. ábra: A példaállítás standard modellben reprezentálva

A kép alapján lászik, hogy lényegesen bonyolultabb állatás modell, mint az éltulajdonság modell. Minden állítást külön erőforrásként azonosítunk, amihez a szabványos alanya, állítmánya és tárgya éllel kapcsolódnak az állítás megfelelő részei.

A modellnek vannak előnyei és hátrányai is. Könnyen belátható, hogy az előző modell mindkét problémáját megoldja: mind a reifikált tulajdonságok, mind a állítás objektumai (alany, predikátum, tárgy) gráf csomópontok, így további kapcsolatokban vehetnek részt, valamint azáltal, hogy az állítás azonosított erőforrásként van reprezentálva, alanya lehet más állításnak is, amivel a többszintű reifikáció problémáját is megoldja. Összességében ez tekinthető a reifikáció legáltalánosabb modelljének.

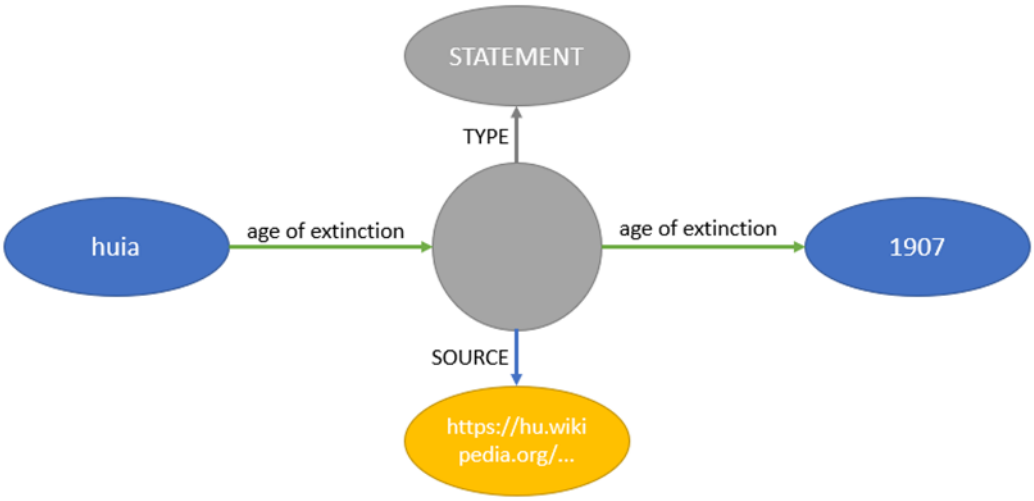
A modell ezen általánosság elérése érdekében számos más modelleknél meglévő jó tulajdonságot feláldoz, melyek közül a legfontosabbak:

- Az állítások és csúcspontok száma az adatbázisban többszörösére növekedik. Mivel egy konkrét állítást legalább 4 állítással lehet reprezentálni, így mind az állítások száma, mind a szükséges tárhely mérete ilyen mértékű növekedést szenved el.
- A modellezett világ absztrakciós szintje alárendelt szerepbe kerül a reprezentációs modell absztrakciójával szemben, azaz a domináns strukturáló elem nem a modellből, hanem a modellezésből származik, ezáltal a modell kevésbé kifejezővé válik.
- Ontológiák bevezetését gyakorlatilag lehetetlenné teszi. Abból kifolyólag, hogy minden állítást ugyanaz a kapcsolathalmaz ír le, azokra érdemi

ontológiai jellemzőket, megszorításokat nem lehet definiálni, hiszen például az „alanya” kapcsolat végpontja tipikusan bármi lehet.

3.2.3 Proxy reifikációs modell

Az előző két modell a reifikációs spektrum két végpontjának tekinthető. Az éltulajdonság modell kis komplexitású, könnyen érthető, de nem elég általános, míg a standard modell nagyobb komplexitással rendelkezik, kevésbé kifejező, viszonyt a lehető legáltalánosabb modell. A proxy (n-ed rendű) [13] modell ennek a két modellnek az erősségeire építve a középútnak tekinthető.



9. ábra: A példaállítás proxy modellben reprezentálva

A kép alapján könnyen észrevehetők az előbbi modellek jellegzetességei. Az állítások ebben a modellben is – a standardhoz hasonlóan – erőforrásként szerepelnek, ezáltal maguk is részt tudnak venni kapcsolatokban, ami egyben a többszintű reifikáció problémáját is megoldja. Azonban az éltulajdonság modell kifejező erejét is nagyrészt megtartja, hiszen a kapcsolatok továbbra is könnyen értelmezhetők, ha nem egy, hanem két mélységben interpretáljuk őket. Az állítások és csúcspontok száma is az előző két modell között található, továbbá ez a modell már lehetővé teszi, hogy olyan ontológiát rendeljünk hozzá, mely modell megszorításait jól le tudja írni.

3.2.4 További modellezési megoldások

Az előzőekben bemutatott három reifikációs reprezentáció csak három általam kiválasztott és megmért modell, ezeken kívül is számos lehetséges modellezési és technológia megoldás van a problémára.

Az egyik lehetséges megoldás az úgynevezett „singleton tulajdonságok” [13][18] bevezetése. A singleton tulajdonság modell azt használja ki, hogy az RDF-ben az predikátumnak is erőforrásnak kell lennie. Az eddigi modellekben kapcsolat típusokat lehetett definiálni, aminek egy tulajdonságainak értékei tértek csak el. Ebben a modellben minden kapcsolat példányt egyedi erőforrás azonosít (tehát két „kihalás éve” kapcsolatnak is más lesz az azonosítója), így lehetővé teszi az egyedi állítások reifikációját.

Egy másik lehetséges megoldás, az adatmodell általánosítása. Az RDF modell szigorú hármasságát négyesekre cserélve egy kifejező, és nagyon általános modellt kapunk. Az így kapott négyesek első három eleme ugyanaz, mint az RDF esetében, míg a negyedik elem egy erőforrás, ami magát az állítást jellemzi. Ez azonban azt is jelenti, hogy ez a modell az adatokat nem gráfként, hanem hipergráf formában írja le, ami nagymértékben növelheti az ilyen rendszerek komplexitását.

Egy harmadik lehetséges megoldás magának az RDF-nek és a hozzá kapcsolódó SPARQL nyelvnek a bővítése. A Blazegraph esetén lehetőség van egy RDF* mód használatára [33], ami úgy bővíti az szabványos RDF formátumokat és lekérdező nyelvet, hogy állítás alanyaként állítás önmagában is szerepelhet.

4 Mérési környezet

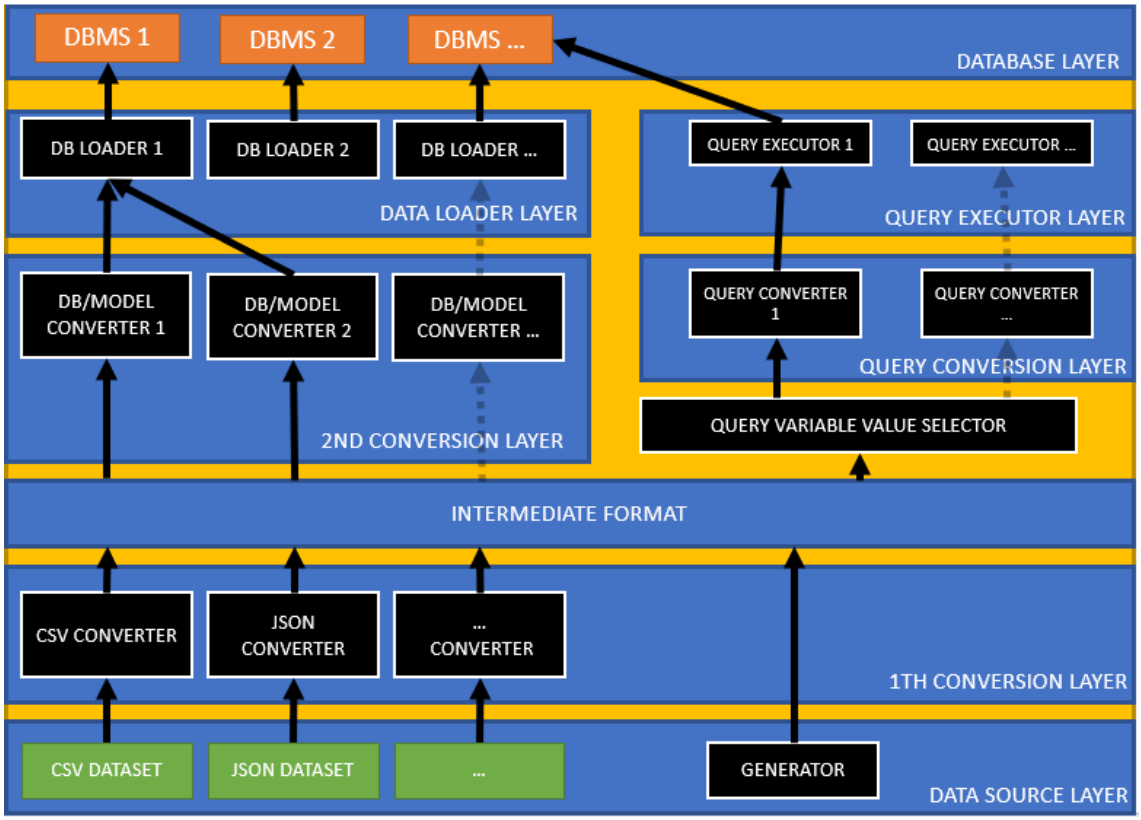
4.1 Mérő szoftverrendszer

A gráf-alapú adatbázisok teljesítményének mérésére már létezik számos elérhető megoldás, azonban ezek mindegyike a gráfadatbázisok egy speciális aspektusát méri csak, hiszen az ilyen jellegű benchmarkoknál számos szempontot figyelembe kell venni. Ilyen szempont lehet a gráf típusa, a modell, ami alapján a gráf felépül vagy a lefuttatott lekérdezések jellege [11].

A reifikáció témakörének szerény kutatottsága miatt nagyon kevés az interneten elérhető benchmark, amely a reifikált gráfok teljesítményét vizsgálná, így elkészítettem egy saját teljesítmény-mérő rendszert, mely az általam vizsgálni kívánt gráf- és lekérdezés típus teljesítményének mérésére alkalmas. Annak érdekében, hogy ezt a célt elérjem, az elkészített rendszer a benchmarking folyamat minden fontos lépését megvalósítja a gráf generálásától a lekérdezések idejének összegyűjtéséig.

Az előzők szerint minden (gráf) benchmark és eredményeik csak adott feltételek mellett használható, így fontosnak tartom az elkészített rendszert jellemezni ebből a szempontból. A rendszer gráfadatbázisok teljesítményének elemzésére készült. A vizsgált gráf a bemutatott tulajdonsággráf modell valamennyi bővítésével rendelkezik: az egyes objektumok tulajdonságokkal és típussal rendelkeznek, a kapcsolatoknak irányuk van, és hozzájuk szemantika rendelhető. Ezeken kívül a legfontosabb tulajdonság, hogy reifikált adathalmazt vizsgáltam.

A vizsgált lekérdezések lokális lekérdezések, így az egyes gráf-alapú rendszerek tranzakcionális teljesítményének egy aspektusát jellemzik csak, más jellegű lekérdezések esetén – elsősorban analitikai lekérdezések esetén – teljesen eltérő eredmények is adódhatnak.



10. ábra: Az elkészített rendszer architektúrája

Mint a mellékelt architektúra is mutatja, az elkészült rendszer több szintből épül fel, amelyből valamennyi lecserélhető, módosítható.

A legalsó szint az opcionális adatgenerátor réteg. Ahhoz, hogy mérhető lekérdezéseket tudjunk megfogalmazni, szükség van valamilyen lekérdezhető adathalmazra. Mivel a generált adathalmaz jellege döntően befolyásolhatja a mérési eredményeket, így nem lényegtelen, hogy az adatgenerálás folyamata milyen modell szerint történik. Annak érdekében, hogy a generált adathalmazon elvégzett mérések ne álljanak nagyon távol egy valós szituációban felmerülő modellen mértektől, a generálás a Barabási-Albert-modell szerint történik, mert ez a modell számos fontos hálózattípus (például szociális hálók) leírásának jó közelítésének tekinthető [12].

A sokszínű adatmodell és adatbázis-kezelő rendszer egyik velejáró hátránya, hogy valamennyi rendszer saját, egyedi megoldást kínál az adatok betöltésére is. Ezek a megoldások nem csak az egyedi betöltési mechanizmusban (például dedikált betöltő program) térnek el, hanem a betöltendő adathalmaz formátumában is. Mind az adatok nyers reprezentálására, mind az adatok betöltésére számos formátum adódik lehetőségként, melyek rendszerint egymással eltérő prioritás szerint reprezentálják az

adatokat. A nyers adatok tárolási formátumánál az egyik elsődleges szempont, hogy az adattárolás kompakt módon történjen, hiszen tipikusan nagy méretű adathalmazokat tárolnak. Az adatok betöltésekor viszont az az előnyös, ha a formátum a lehető legegyszerűbben átalakítható az adott rendszer belső reprezentációjára, ahol viszont nem (feltétlenül) a tárhely, hanem a lekérdezés teljesítménye a meghatározó szempont.

Ez azt eredményezi, hogy ha van N bemeneti formátum, amit S különböző rendszerbe szeretnénk betölteni, és az i. rendszer M_i különböző reifikációs modellt is támogat, melyekben a betöltendő adathalmaz eltérő struktúrában szerepel, akkor szélsőséges esetben $N * \sum_{i=1}^{S} M_i$ konverziós programot kellene készíteni. Annak érdekében, hogy ne kelljen minden újonnan felvett rendszer vagy adatformátum miatt rengeteg új konverziós programot készíteni, definiáltam egy közös köztes reprezentációs formátumot [34]. Ez lehetővé teszi, hogy új adatreprezentációs formátum használatakor elegendő egy új konvertáló programot készíteni, ami az adatokat átalakítja a közös formátumra, míg új rendszer hozzáadásakor is elegendő egy új konverziós programot elkészíteni, ami a közös formátumot átalakítja az új rendszer számára betölthető formátummá. Ezt a megoldást használva a kiindulási $N * \sum_{i=1}^{S} M_i$ konvertáló helyett elegendő $N + \sum_{i=1}^{S} M_i$ átalakító programot implementálni. Az elkészített gráf generátor is ilyen, közös formátumú adathalmazt állít elő.

Azután, hogy megtörtént a nyers adathalmaz átalakítása köztes formátumra, majd onnan az adott rendszer számára betölthető reprezentációra – ami néhány rendszer esetén megegyezhet a köztes formátummal –, megtörténik az adatok rendszerspecifikus betöltése.

A teljesítménymérés fókusza lehet azon, hogy a rendszerek teljesítményét olyan helyzetben mérjük, amikor az egyes lekérdezések eredményhalmaza (elvártan) üres. Mivel az ilyen jellegű lekérdezések a rendszerek teljesítményét egy nagyon speciális aspektusból jellemzik, így az általam végzett mérések esetén fontos szempont volt, hogy ne ilyen lekérdezések kerüljenek megfogalmazásra, hiszen a vegyes lekérdezések (volt találat vagy nem volt találat) torzítanák az eredményt. Annak érdekében, hogy minden lekérdezésnek biztosított legyen a nem üres eredményhalmaza, a lekérdezés változóit a köztes formátumból, egyenletes eloszlás szerint választja ki egy szelekciós program.

A lekérdezés változóinak értékének kiválasztása után megtörténik az adott rendszer- és reifikációs modellspecifikus lekérdezések generálása, amelyeket ezután egy

futtató program elküld a megfelelő adatbázis-kezelőnek, és megméri a válasz idejét, majd az így kapott időeredményeket ezután elmenti a későbbi elemzéshez.

4.2 Mérési környezet

A mérési eredmények kiértékelése szempontjából alapvető jelentőségű, hogy a különböző mérések egymástól függetlenül, a lehető legazonosabb körülmények között legyenek végrehajtva, hiszen a kapott lekérdezési futásidők csak ilyen esetben hasonlíthatók össze értelmesen. Ezt számos módon próbáltam biztosítani.

A mérési eredmények akkor hordoznak releváns, összehasonlítható információt, ha minden rendszer minden reprezentációjában azonos nyers adathalmazon ugyanazokat lekérdezéseket mérik (eltekintve a nyelvi szintaxistól). Annak érdekében, hogy ezt biztosítsam, mind a nyers adathalmaz, mind a lekérdezések változóinak értékei előre le lettek generálva, az összes mérés ennek az állománynak a feldolgozásából indul ki. A szelekciós program a teljes adathalmazból, egyenletes valószínűség szerint választotta ki a konkrét változó értékeket. Az egyes betöltések, konvertálások mindig újonnan letöltött adathalmazról történtek, így nem fordulhatott elő olyan, hogy az előző mérés esetleges változtatásai befolyásolták volna a későbbi méréseket.

Minden különböző reprezentációs modellel végzett mérés előtt az aktuális adatbázis-kezelő rendszer legfrissebb változata újra lett telepítve – vagy az adatbázis ki lett törölve, ahol ez nem volt lehetséges –, így a korábbi mérések semmilyen formában sem befolyásolhatták a rendszer állapotát a későbbiekben. Abból kifolyólag, hogy a mérések rövid időn belül lezajlottak, az egyazon rendszeren elvégzett, de más reifikációs modellt vizsgáló mérések között adatbázis-kezelő frissítés nem történt.

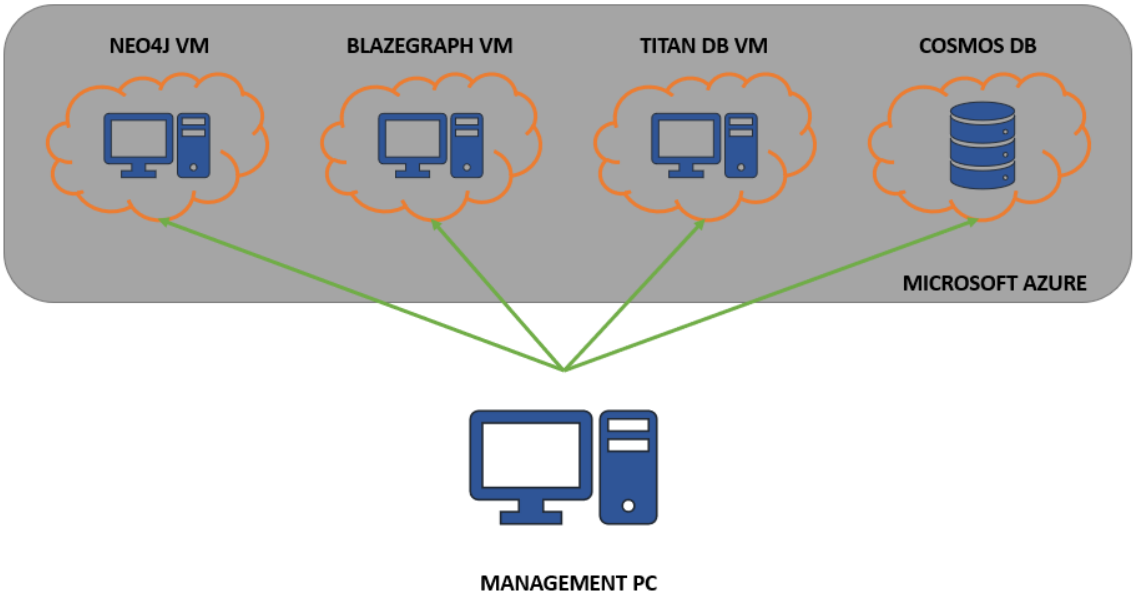
A méréseket a Microsoft Azure felhőszolgáltatáson keresztül elérhető virtuális gépeken, illetve adatbázis-kezelőkön végeztem el. Abból kifolyólag, hogy a mérések a felhőben zajlottak, a mások által használt szolgáltatásoknak (kismértékben) befolyása lehetett a mérési környezetre, hiszen a különböző virtuális gépek megosztott erőforrásokat használnak, így előfordulhatott, hogy más szolgáltatás terheltsége miatt a mérést végző rendszer teljesítménye is csökkent, ami a válaszidők növekedését okozhatta. Ezt a mérések elvégzése, az eredmények elemzése során két okból hagytam figyelmen kívül:

1. A megosztott erőforrások miatti kölcsönhatások az egyes virtuális gépek között kicsinek tekinthető, így a mérést minimális mértékben befolyásolhatta csak.

2. A manapság bevezetésre kerülő új szolgáltatások nagyrésze már felhő-alapú héttérrendszerrel rendelkezik. Ezeknek az „éles” rendszereknek a környezete ezáltal azonos a mérést végző rendszer környezetével, tehát az előbbiekben bemutatott egymásra hatás miatti torzítás valójában a valós helyzethez közelebbi környezetet biztosítja a méréseknek.

A méréseket lényegében ugyanolyan konfigurációjú, azonos operációs rendszerrel rendelkező virtuális gépeken végeztem, melyek jelenleg aktuális Microsoft Azure Standard E4s v3 specifikációja szerint a következő erőforrásokkal rendelkezik [19]:

- Intel XEON E5-2673 v4 processzor, 4 virtuális mag, magonként 2,3 GHz 3,5 GHz órajel frekvenciával
- 32 GB RAM memória
- 64 GB SSD háttértár az operációs rendszernek, igény szerinti méretű további SSD Premium tároló a rendszerek és adataik számára
- Ubuntu 16.04 LTS Server operációs rendszer



11. ábra: A mérési környezet topológiája

A kép alapján látható, hogy a Neo4j, a Blazegraph és a Titan is dedikált virtuális gépen futott, így a rendszerek bekonfigurálását és magukat a méréseket is el tudtam végezni a virtualizált környezetben. Ezzel szemben a CosmosDB-t – mivel az csak a Microsoft Azure keretein belül, külön erőforrásként elérhető, így virtuális gépre nem telepíthető – közvetlenül a management számítógépről vezéreltem és mértem.

A mérések folyamata minden esetben a következő forgatókönyv szerint zajlott:

1. A mérés előkészítéséhez szükséges szoftverek telepítése a virtuális gépekre. Valamennyi vizsgált adatbázis-kezelő rendszer Java alapú, így szükséges volt a Java feltelepítése a gépekre. Ezeken kívül a .NET Core keretrendszer is minden esetben telepítésre került, mert az elkészített mérőrendszer ebben íródott.
2. A megfelelő adatbázis-kezelő szoftver telepítése, bekonfigurálása.
3. Teszt adathalmaz letöltése, átkonvertálása a rendszer mérendő modell szerinti importálható formátumára. Ezzel egyidejűleg a lekérdezések generálása az előre generált változó helyettesítéseket felhasználva.
4. A szükségtelen átmeneti fájlok törlése, hogy az eltérő betöltési folyamat során keletkező átmeneti állományok nem befolyásolják a mérési eredményeket.
5. Az adatbázis-kezelő rendszer elindítása, majd a lekérdezések futási idejének megmérése és összegyűjtése az erre a célra készített futtató programmal.

5 Mérési paraméterek

5.1 Adathalmaz

Abból kifolyólag, hogy nincs konkrét alkalmazási eset, ami alapján a rendszerek teljesítményét mérni lehetne, egy generátor által készített nagyméretű reifikált adathalmazon történtek a mérések.

A generált adathalmaz 100 000 csomópontot tartalmazott, melyek 500 csomópont típus valamelyikéhez tartoztak. A csomópont típusok típusinformációként szolgálnak az egyes csúcspontokról, és az általuk definiált tulajdonság halmazok a gráf generálásánál vannak felhasználva. Minden csomópont rendelkezik minimum egy, maximum 50 tulajdonsággal. Mivel a mérés elsődlegesen a reifikált kapcsolatokra fókuszál, így ennél több tulajdonság használata csak erőforrás pazarlás lett volna, hiszen a mért lekérdezésekben nincs rájuk hivatkozás. Összességében tehát a nyers adathalmaz 500 típushoz tartozó, 3 678 739 tulajdonsággal (kulcs érték párral) jellemzett 100 000 csomópontot tartalmaz.

A csomópontok között sűrű, többszörösen reifikált kapcsolatok vannak. A 100 000 csomópont között összesen 16 046 133 irányított összeköttetés található, melyek mindegyike az 1 000 generált éltípus valamelyikéhez tartozik. Az éltípusok – analóg módon a csomópont típusokkal – típusinformációként vannak jelen, de nem csak a generálásnál, hanem a lekérdezések esetében is fontos szerepük van. Minden egyes állítás legalább egy, de maximum 100 minősítővel rendelkezhet. Mivel ez a mennyiség már alkalmas arra, hogy az adatbázis-kezelő rendszerek reifikált tulajdonságokkal kapcsolatos teljesítményét mérni tudjam, így ennek a számosságnak a lényeges növelése már csak felesleges erőforráshasználatot eredményezett volna. Összességében tehát a nyers adathalmaz 1 000 típushoz tartozó, 459 387 189 éltulajdonsággal reifikált 16 046 133 élt tartalmaz.

Gráf elem	Mennyiség
csomópont	100 000 db
tulajdonság	3 678 739 kulcs-érték pár
él	16 046 133 db
minősítő	459 387 189 kulcs-érték pár
csomópont típus	500 db
tulajdonság típus	50 db
él típus	1 000 db
minősítő típus	100 db

1. táblázat: A méréshez generált gráf paraméterei

Annak érdekében, hogy a generált adathalmaz a nagy mérete ellenére is viszonylag kis tárhelyet foglaljon, mind a generált nevek, mind a tulajdonság értékek a lehető legrövidebbeknek lettek választva.

`{id: „123”, type: „N3”, P1: „K”, P2: „T”, P5: „E”, …, P50: „V”}`
`{sourceId: „3”, targetId: „123”, type: „E34”, Q1: „U”, Q3: „M”, …, Q98: „A”}`

2. táblázat: Részlet egy generált csúcspontról (felső) és egy élről (alsó)

5.2 Lekérdezések

Mint azt korábban említettem, az adatbázis-kezelők teljesítményének elemzésekor a futtatott lekérdezések jellegének megállapítása kulcsfontosságú, hiszen a rendszer teljesítményét mindig csak szigorúan ebből az aspektusból jellemzi. Emiatt fontos kitérnem, hogy az ezután ismertetett mérési eredmények milyen jellegű lekérdezések futásából születtek, hiszen más felhasználói esetben akár teljesen eltérő eredmények is születhetnek.

A futtatott és mért utasítások mind lekérdezések voltak, azaz az adatbázis tartalmát nem változtatták meg semmilyen módon. Az általam vizsgált lekérdezéscsalád a [13]-ban is ismertetett, lokális reifikált gráfillesztés sebességét hivatott megmérni. Ennek a lényegét a következő példa mutatja be:

Adott a következő rendezett ötös: „11, 54, E32, Q12, L”. Észre vehető, hogy ez egy konkrét reifikált állítást ír le, mégpedig a 11-es azonosítójú csomópontból az 54-es azonosítójú csomópontba mutató, E32 típusú élt Q12 szemantika szerint L értékkel jellemző reifikált állítást. Egy ilyen (s, o, p, q, v) ötöshöz pontosan 32 különböző lekérdezés fogalmazható meg, attól függően, hogy az ötös mely tagjait tekintjük kitöltött értéknek. Ezáltal ezzel a lekérdezés mintával lehetőség van a teljes adatbázis lekérdezésére – ha egyik változót sem kötjük le –, ugyanakkor meg tudunk keresni egy konkrét reifikált állítást is – ha az összes változót lekötjük.

5.3 Adatbázis-kezelő rendszerek

Az adatokon és a lekérdezéseken túl a teljesítményt erősen befolyásolhatja az adatbázis-kezelő belső konfigurációja, emiatt a mérési eredmények mellett ezeket is fontos közölni.

Az egyik vizsgált rendszer a Neo4j 3.2.6 Community Edition volt, Java 8-as futtatókörnyezet mellett. A rendszer dokumentációja alapján nagyon kevés lehetőség van a teljesítmény befolyásolására. Az egyik lehetőség a Java halom (heap) méretének növelése, amit meg is tettem, a mérések 20 GB-os mérettel zajlottak. A másik lehetőség az ún. „lekérdezés tippek” használata [42], amellyel jelezni lehet a végrehajtó rendszer számára, hogy szerintünk hogyan kellene a gráfillesztést elvégezni. Mivel ez nem általánosan alkalmazható optimalizálási megoldás, így a mérések során ezt a funkciót nem használtam. Az alapértelmezett beállítások között kiemelendő, hogy a Neo4j alapértelmezés szerint indexeli a csúcspontokat, ami a teljesítményt jelentős mértékben növelheti [13].

Egy másik vizsgált rendszer a Blazegraph 2.1.4 volt, szintén Java 8-as futtatókörnyezet mellett. A Blazegraph-nak nagyon sok konfigurációs lehetősége van, emiatt a dokumentációban elérhető számos előre elkészített konfigurációs állomány is [35]. A méréseknél használt konfigurációs állomány az RDF Only és a Fast Load módok beállításait együttesen tartalmazta, azaz a következtetéshez használatos információk letiltásra kerültek, ugyanakkor a gyors betöltéshez és lekérdezés kiértékeléshez használatos funkciók bekapcsolásra kerültek. Mindezek mellett a dokumentáció által javasolt maximális Java halom méret is (6 GB) be lett állítva [13].

A CosmosDB Gráf API esetén két konfigurációs lehetőséget találtam. Lehet változtatni az előfizetéshez tartozó átviteli sebességet [36], ami az utasítások végrehajtási

sebességét nem, csak a rendszer áteresztő képességét befolyásolja, tehát azt, hogy mennyi lekérdezést lehet lefuttatni másodpercenként. Az méréseket 1 000 RU/s (kérési egység másodpercenként) teljesítményű gépen végeztem. A másik lehetőség a konzisztenciaszint beállítása [37]. A mérések során az alapértelmezett munkamenet konzisztenciaszintet használtam, ami a méréshez szükséges műveleteket sorrendhelyesen végzi el. A rendszer ezen kívül alapértelmezés szerint minden tulajdonság szerint indexet épít fel. [36]

Az utolsó vizsgált rendszer a Titan DB volt, Java 8-as futtatókörnyezet mellett. Ezen rendszer esetén több teljesítménybefolyásoló lehetőség is van. Az egyik alapvető tényező a megfelelő tárolási háttérszolgáltatás kiválasztása. A mérések az Oracle BerkeleyDB háttérszolgáltatást használták. A választás azért erre esett, mert ez nem elosztott, a Titan-nal egy JVM-ben futó tárolást valósít meg, ami néhányszor 100 000 000 elemű gráf esetén a legnagyobb teljesítményt biztosítja [38]. Ezen kívül indexet hoztam létre a csúcspontok azonosítójára.

Minden más beállítás esetén az adott rendszerek alapértelmezett beállításait használtam, hiszen munkámnak nem volt célja az összes adatbázis-kezelő teljesítménnyel kapcsolatos valamennyi beállításának megismerése. A 3. táblázat első két oszlopában látható, hogy mely adatbázis-kezelő rendszerek esetén mely reifikációs modellek kerültek megmérésre.

6 Mérési eredmények

6.1 Betöltés

Rendszer	Modell	Betöltési idő (perc)
Neo4j	Tulajdonsággráf	16,8
	Standard	23,2
	Proxy	21,2
Blazegraph	Standard	46,4
	Proxy	43,4
Titan DB	Tulajdonsággráf	153,8
	Standard	413,2
	Proxy	346,2
CosmosDB	Tulajdonsággráf	3 931,2

3. táblázat: A különböző rendszerek és modellek betöltési ideje

Annak ellenére, hogy a dolgozatom központjában az adat lekérdező műveletek vannak, a betöltési sebességek is a rendszer teljesítményét jellemzik bizonyos szempontból. A mérések során az egyes rendszerek betöltési sebességei között alapvető különbségeket tapasztaltam.

A betöltés esetén a Neo4j és a Blazegraph rendelkezik saját importáló eszközzel, amiket fel is használtam az adatok tömeges betöltésekor. A Titan DB is rendelkezik több ilyen eszközzel [39], azonban ezek egyikét sem sikerült összekapcsolni a Titan-nal úgy, hogy a betöltés után az adatokat vissza is tudjam olvasni, lekérdezést tudjak megfogalmazni rájuk. Vélhetően a fiatalsága miatt, a CosmosDB Gráf API nem rendelkezik ilyen tömeges betöltő eszközzel. Ezen okokból az utóbbi két rendszer esetén a betöltés a csúcspontok, majd az élek egymás utáni hozzáadásával történik.

Ezek tükrében már értelmezhető, hogy miért van olyan nagy szakadás, az első és a második két rendszer betöltési eredményei között. A CosmosDB esetén beállított átviteli teljesítmény, illetve a hálózati átvitel tovább lassította a betöltést, ezzel magyarázható, hogy még a Titan DB betöltési idejének is többszörösét produkálta.

6.2 Lokális lekérdezések

A fent bemutatott lekérdezéscsalád volt a mérés fókusza. Az adatok generálása után egyből megtörtént a lekérdezési változók értékeinek véletlenszerű kiválasztása is. A korábbiak alapján értelmezhető a kiválasztott tíz ötös:

SUBJECT	OBJECT	PREDICATE	QUALIFIER	QUAL. VALUE
51	9426	E129	Q52	J
4878	17432	E612	Q92	H
54	33618	E282	Q5	K
23	41145	E595	Q56	W
21	43683	E843	Q49	R
17875	49976	E693	Q31	S
36	60922	E478	Q97	G
30	62950	E525	Q58	N
162	95376	E915	Q61	D
264	99221	E611	Q51	R

4. táblázat: A lefuttatott változó behelyettesítések

A mérések úgy történtek, hogy a 32 lehetséges lekérdezés mintára mind a 10 lekérdezést lefuttattam az összes vizsgált adatbázis-kezelő rendszer, reifikációs modell kombinációra. A Neo4j és a Blazegraph esetén a lekérdezések futtatása REST API-n keresztül történt, a CosmosDB-hez készült .NET Core alapú kliensoldali SDK, így a szerverrel folytatott kommunikáció ezen keresztül zajlott, a Titan DB esetén pedig – mivel egyazon JVM-ben fut a mérő alkalmazás és az adatbázis is – a hivatalos, Maven-en keresztül elérhető könyvtár segítségével hajtottam végre a méréseket.

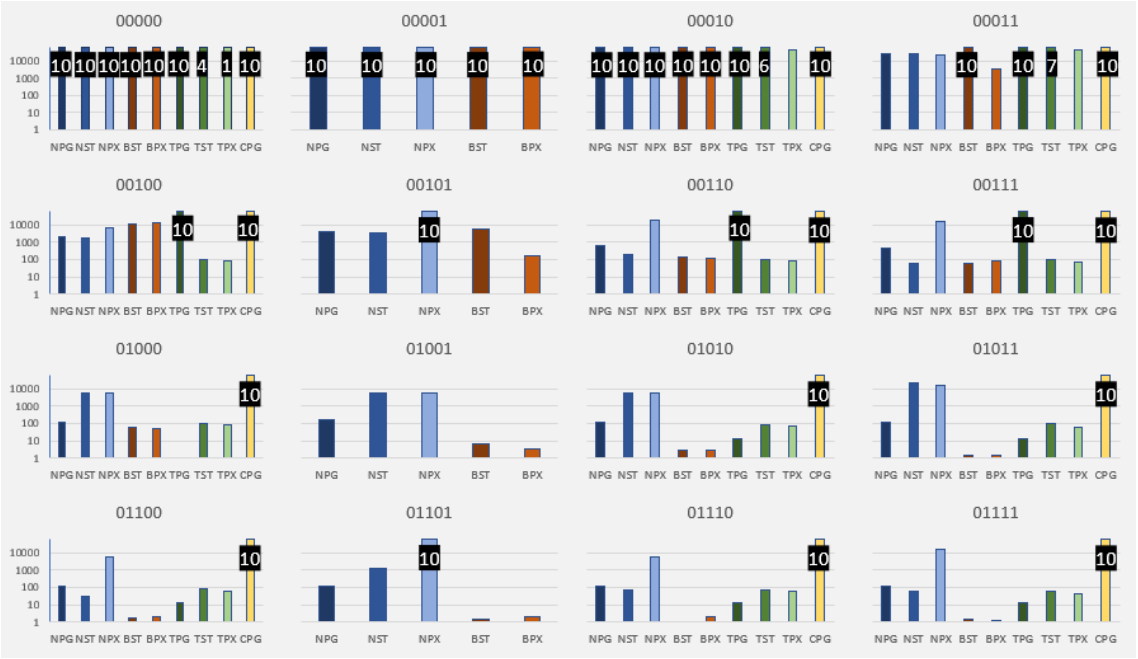
A lekérdezési minták során észrevehető, hogy vannak olyan lekérdezések, amik annyira általánosak, hogy ekkora adathalmazon várhatóan óriási futási idővel, és eredményhalmaz mérettel rendelkeznének. Az idő problémájára található viszonylag általános megoldás: valamennyi rendszer támogatja a lekérdezés időkorlát megadását,

amit minden esetben egy percre állítottam. Az előállított eredményhalmaz nagy méretének kezelése viszont egyedileg történt meg.

A Neo4j és a Blazegraph esetén REST API-n keresztül lehetőség van arra, hogy ne az eredményeket, hanem a kiértékelési statisztikát küldje vissza a rendszer, amiben megtalálhatók a lekérdezés kiértékelésekor előálló statisztikus adatok, mint például a lekérdezési terv, a végrehajtás ideje stb. A későbbiekben ismertetett végrehajtási idők esetén ennél a két rendszernél ezeket az időket tüntettem fel.

A CosmosDB esetén az SDK szintén lehetőséget ad ilyen információk lekérésére, azonban a mérés során ez a statisztikai objektum mindig null értéket kapott vissza a szervertől, így más módot választottam. A lekérdezéseket átalakítottam úgy, hogy az eredményhalmaz számosságát adja vissza, így a méret problémát ez megoldotta. A lefuttatott lekérdezések jellegét ugyan ez megváltoztatta, ez azonban a lekérdezések futási idejét nem befolyásolta olyan mértékben, hogy ez időkorlát túllépését okozná. Ezt úgy ellenőriztem, hogy valamennyi lekérdezés típus 2-2 lekötött változójú elemét a számosság aggregáció nélkül is lefuttattam, limitálva az visszaadandó eredményhalmaz számosságát 10 000-re, azonban így sem volt olyan lekérdezés, ami az időkorláton belül lefutott volna.

A Titan DB esetén a gráfbejárásnak van egy iterate metódusa, ami végrehajtja a megadott lépésekből álló bejárást. Ennek segítségével lehetőség van a Java beépített időmérési lehetőségeit felhasználva a lekérdezés teljesítményének mérésére.



12. ábra: Az első 16 minta mérési eredménye

13. ábra: A második 16 minta mérési eredményei

A diagramm az egyes adatbázis-kezelő, reifikációs modell párosok átlagos futási idejét mutatja a különböző lekérdezések esetén. Minden rész diagramm egy-egy lekérdezési minta esetén mutatja a páros tíz lekérdezéséből származó átlagos futási időt. Az x-tengely mentén a párosok nevei találhatók, ahol a N a Neo4j-t, B a Blazegraph-ot, C a CosmosDB-t és T a Titan DB-t jelenti, míg az indexek esetén a PG a tulajdonsággráf, az ST a standard és a PX a proxy modellt jelenti. Az y tengelyek az átlagos futási időt jelképezik milliszekundum mértékegységben, logaritmikus skálán. Az átlag számításánál

– annak érdekében, hogy a mérési eredmények a valós teljesítményt mérjenek, és ne csak a befejezett lekérdezésekét, hiszen az torzítaná az eredményeket – az időlimit túllépés miatti lekérdezések egy perccel lettek számítva. [13] Az oszlopokon jelzett számok az esetleges időtúllépések számát jelentik.

6.2.1 CosmosDB

Az eredmények alapján nyilvánvalóan látszik, hogy a CosmosDB Gráf API-ja jelenleg messze a leggyengébb a vizsgált rendszerek közül az ilyen jellegű lekérdezések esetén. A betöltési idő a legdrágább előfizetés esetén lecsökkenthető a Titan DB szintjére. A lekérdezések futási ideje azonban az összes lekérdezés esetén túllépte az időlimitet, ami megerősíti azt a mások által is tapasztalt eredményt [40], hogy a rendszer jelenlegi állapotában nem igazán alkalmas nagyméretű gráfok tranzakcionális jellegű lekérdezéseire.

A CosmosDB jelenleg a csak a TinkerPop gráf stack egy részhalmazát valósítja meg [41], ami következtében bizonyos minták esetén (01 végű minták) nem tudtam lekérdezést megfogalmazni, ez az oka annak, hogy néhány grafikon esetén nincs feltüntetve a CosmosDB eredménye.

6.2.2 Neo4j

A Neo4j-t három különböző modellel is teszteltem, azonban az eredmények alapján nem lehet egy egyértelműen legjobb reifikációs módot megállapítani. Az eredmények alapján 32-ből 19 esetben a proxy modell teljesítménye volt a leggyengébb, ami alapján arra lehet következtetni, hogy a Neo4j szélességi logika szerint járja be a gráfot, hiszen az eggyel hosszabb kapcsolatok eszerint a megközelítés szerint növelik a bejárandó csúcspontok számát az eredmény előállításakor.

Látható, hogy a legáltalánosabb lekérdezésekre – amikor vagy egyáltalán nincs változó lekötve, vagy csak a reifikáció szemantikája vagy értéke ismert – egyik modell sem tudott időlimiten belül válaszolni.

Egy másik megfigyelhető minta, hogy a tulajdonsággráf modell a legtöbb esetben akkor bizonyult a leggyorsabbnak, amikor az él típusa nem volt lekötve. Ilyen esetekben akár egy nagyságrenddel is jobban teljesít a másik két modellnél, míg az élek típusának lekötésekor a teljesítmény olyan mértékben romlik le, hogy a standard modell átlagosan 2-3-szor gyorsabb lesz nála. Ez alapján arra lehet következtetni, hogy a rendszer nincs

megfelelően optimalizálva az éltípusok és az éltulajdonságok egyidejű használatára. Ezt erősíti meg az is, hogy legtöbb esetben az reifikációs tulajdonságok specializálásával – annak ellenére, hogy a lekérdezés specifikusabbá válik – a lekérdezések futásideje növekszik.

Ez utóbbi egy általánosan megfigyelhető jelenség, ugyanis, ha például a 01010 és az 11010 minták eredményeit megnézzük, akkor azt látjuk, hogy ugyan a lekérdezés konkrétabb lett, a futásidő valamennyi modell esetén növekedett, a tulajdonsággráf modell esetében ráadásul majdnem 4-szeresére. Ez pedig azt jelentheti, hogy a Neo4j a gráfillesztést nem a legoptimálisabb sorrendben végzi el, hiszen más különben a 11010 mintának a szűkítő feltétele nem növelhette volna a lekérdezés futási idejét, főleg, hogy figyelembe vesszük, hogy a csúcsok és az azokhoz tartozó szomszédsági lista indexelve, a memóriában van [13].

Ezen utóbbi két megfigyelés is összhangban van mások által tapasztaltakkal: a [13]-ban leírt kísérleti eredményekben mind az éltulajdonságokkal kapcsolatos lassulás, mind a helytelen lekérdezési terv kiválasztása megtalálható.

6.2.3 Blazegraph

A Blazegraph esetén két ismertetett modellt is megvizsgáltam. Az eredmények alapján az látszik, hogy a Neo4j-hez hasonlóan, az első három – az eredmények tükrében legnehezebb – lekérdezést ez a rendszer sem tudja a megadott időkorláton belül végrehajtani. Az előbbi rendszerrel összehasonlítva azonban azt találjuk, hogy a többi lekérdezés esetén – modelltől függően – a legtöbb esetben egy vagy két nagyságrenddel jobb a Blazegraph-on mért futási idő. Annak ellenére, hogy a mérő rendszerek háttértára gyors SSD technológiájú volt, ilyen rövid futási időket valószínűleg csak úgy tud a rendszer elérni, hogy a háttértárhoz a nem, vagy csak minimális mértékben fordul, a műveletek elvégzéséhez szükséges adatokat (gráf, indexek) a memóriában gyorsítótárazza. Ezen összehasonlításból az következhet, hogy az I/O műveletek végzésének szempontjából a Blazegraph valószínűleg jobban optimalizált, mint a Neo4j.

A modellek teljesítményének összehasonlításakor – a Neo4j-vel ellentétben – azt kapjuk, hogy a 00100 mintát leszámítva a proxy reifikációs modell az általános, kevés lekötött változójú lekérdezések esetén akár egy nagyságrenddel is jobban teljesít, míg a specifikusabbak esetén lényegileg azonos teljesítményt nyújtanak (néhány milliszekundumos eltéréssel). 00100 minta esetén tapasztalt anomália könnyen

magyarázható azzal, hogy a standard modell esetén az éltípusok egy kis elemű csúcshalmazként vannak reprezentálva, míg a proxy modell esetén kétszeres számosságú élhalmazként – hiszen egy kapcsolatot két azonos típusú él reprezentál ebben a modellben –, ami az előbbi esetén lényegesen felgyorsíthatja a gráfillesztést, ahogy azt az ehhez a lekérdezés típushoz tartozó lekérdezési tervek is mutatják.

A mérési eredmények kiértékelésekor az az érdekes jelenség volt tapasztalható, hogy a Neo4j és a Blazegraph esetén is a második és a hatodik lekérdezés ideje töredéke volt a többi lekérdezés idejének valamennyi az időkorlátot nem túllépő lekérdezés minta esetén. A végrehajtási tervek elemzésekor azt találtam, hogy a kérdéses lekérdezésekben szereplő csomópontok meglehetősen kis fokszámúak – a második lekérdezésben szereplő 4 878-as csúcs fokszáma 63, addig a harmadikban alanyként szereplő 54-es csúcsnak 94 662 –, amit a lekérdezés optimalizálásakor ki is használnak ezek a rendszerek. Abból kifolyólag, hogy a Titan DB által használt Gremlin lekérdezőnyelv imperatív jellegű, annál a rendszernél ilyen jellegű optimalizáció a mérési eredményeken nem tapasztalható, lekérdezési terv pedig nem állt rendelkezésemre.

A Blazegraph-fal kapott mérési eredmények több helyen is megegyeznek a mások által kapott eredményekkel, azonban tapasztaltam lényegi eltérést is. A [13]-ban ismertetett eredmények szerint is a Blazegraph rendszerint gyorsabb, mint a Neo4j, és a mért teljesítménybéli különbségek is hasonlóak. További hasonlóság, hogy aszerint a mérés szerint is igaz, hogy a Blazegraph-on vizsgált két modell az esetek több, mint felében körülbelül azonosan teljesített. Azonban azzal a méréssel ellentétben eltérő eredményt kaptam a hatékonyabb modell kiválasztásakor. Az ottani eredményekkel ellentéteben – az általam végzett mérések szerint – egyértelműen a proxy modell tűnik a hatékonyabb választásnak.

6.2.4 Titan DB

A Titan DB esetén is három reifikációs modell teljesítményét vizsgáltam. A mérési eredményeket tartalmazó grafikonon azért nincs feltűntetve néhány esetben a Titan DB, mert azoknál a lekérdezési mintáknál a rendszer helytelen eredményeket szolgáltatott, így a kapott időeredmények sem irányadók. A mérések elvégzése előtt valamennyi lekérdezés helyességét egy kisebb mintahalmazon ellenőriztem, ami során kiderült, hogy a hivatalos, Maven-en keresztül is elérhető modul esetén a TinkerPop

Gremlin által bevezetett hasValue lépés hozzáadása a gráfbejáráshoz minden esetben eltávolítja az összes elemet az eredményhalmazból.

Ennek a megállapítása azért volt egyszerű, mert a lekérdezések ellenőrzésekor a hibás eredményeket és a lekérdezési mintákat összevetve az adódott, hogy minden esetben a 01 végű mintákban volt a hiba, amelyek kizárólagosan tartalmazták az előbb említett lépést. A lekérdezések fokozatos bővítése során pedig kiderült, hogy minden esetben ezután a lépés után vált üressé az eredményhalmaz.

Abból kifolyólag, hogy a lekérdezés mind szintaktikailag, mind szemantikailag helyes, és mégsem ad vissza eredményt, arra a következtetésre jutottam, hogy a használt hivatalos függőségben van a hiba. Mivel más módot nem találtam az ilyen jellegű lekérdezések megfogalmazására, így ezen mintákat a Titan DB esetén nem mértem meg.

A három modell mérési eredményei összességében vegyes képet nyújtanak a rendszerről. A grafikonokról is jól látszik, hogy a standard modell szinte minden esetben lassabb, vagy közel azonos teljesítményt nyújtott, mint a proxy modell, ami alapján azt lehet mondani, hogy ezen rendszer esetén – ha csak ezt a két modellt tekintjük – akkor proxy modell tűnik a jobb választásnak.

A tulajdonsággráf modell esetén az az érdekes jelenség volt megfigyelhető, hogy amennyiben a rendszer képes volt indexből származó információt felhasználni – tehát volt csúcspontra vonatkozó információ –, úgy a válaszidő minden esetben stabilan a 0 26 milliszekundum közötti intervallumba esett, ellenkező esetben mindig túllépte az időkorlátot. Ez alapján arra következtetek, hogy – mivel ez az eset nem állt fenn a másik két modell esetén, ahol az állításokat csúcspont is reprezentálja – vagy a gráfot alkotó élhalmazból induló lekérdezéseket nem tudja a rendszer a csúcsokéhoz hasonló hatékonysággal kezelni, vagy az indexek nélküli lekérdezések sebességcsökkenése a rendszer kevésbé hatékony I/O kezelésének indikátora.

A rendszer két leggyorsabb modelljének teljesítményének összehasonlításakor tehát az eredmények azt támasztják alá, hogy abban az esetben, ha a lekérdezésben legalább az egyik érintett csúcspont le van kötve, akkor tulajdonsággráf modell a leggyorsabbnak bizonyul, más esetben viszont túllépi az időkorlátot, ilyenkor a proxy modell bizonyul a leghatékonyabbnak.

6.3 Konklúzió

A mérési eredmények tükrében nem lehet egy abszolút értelemben vett legjobb adatbázis-kezelő rendszer és reifikációs modell párost kijelölni. Jól mutatja a rendszerek és modellek sokszínűségét, hogy egy ilyen viszonylag korlátozott spektrumú lekérdezéscsalád esetén is nagy különbségek tapasztalhatók modell és modell, de rendszer és rendszer között is.

Az CosmosDB Gráf API-ja nagyon fiatal, ami meg is látszik még a teljesítményén, ezt tekintve a rendszernek még nagyon sokat kell fejlődnie a rendszernek, hogy sebességben utolérje a konkurenseit. A Neo4j nagyon jól használható dokumentációval és kisegítő eszközkézlettel rendelkezik, a vizsgált rendszerek közül messze ezt volt a legkényelmesebb használni. A kényelmesség és a népszerűség ellenére – a CosmosDB-t nem számítva – általgosan ez a rendszer érte el a leggyengébb eredményt, köszönhetően elsősorban a közel sem optimálisan működő lekérdezés tervezője miatt. A Blazegraph esetén némileg fordított a helyzet. A rendszer teljesítménye még ekkora gráf esetén is kifogástalan, azonban a néhol elavult és nem teljes dokumentáció hiányában a kezdő konfigurációs lépések sokkal tovább tarthatnak. A Titan DB teljesítménye meglehetősen szélsőséges volt, hiszen a néhány tíz milliszekundumos válaszidő és az időkorlát túllépése között nem nagyon volt köztes mérési eredmény. Ennél a rendszernek az egyik legnagyobb hátránya, hogy csak imperatív jellegű lekérdezéseket lehet megfogalmazni, ami a futás közbeni optimalizáció lehetőségét elveszi a rendszertől.

A mérési eredményekből az következik tehát, hogy ilyen jellegű reifikált gráfok lokális, tranzakcionális lekérdezéseinek kiszolgálásához a Blazegraph rendszer és a proxy reifikációs modell párosa nyújtja jelenleg a legnagyobb teljesítményt.

6.4 További kutatási lehetőségek

A mérést számos módon tovább lehet fejleszteni, ami további kutatások alapja lehet. Ennek az egyik nyilvánvaló módja a mérési paraméterek változtatása. Ez történhet akár további adatbázis-kezelő rendszerek bevonásával (például a bevezetőben említett Microsoft SQL Server 2017), akár a megemlített további reifikációs lehetőségek vizsgálatával, vagy a megmért lekérdezések jellegének megváltoztatásával.

A felhő technológia lehetővé teszi, hogy manapság az alkalmazásainkat a felhasználás mértéke szerint skálázhassuk. Ez felvet egy olyan kutatás lehetőségét, ami

akár az általam elkészített méréseket több, eltérő méretű gráfon is végrehajtja, majd a kapott adatokat összevetve jellemzi az egyes rendszereket az adat skálázódás szempontjából.

Egy másik fontos tovább gondolási lehetőség, ami a gyakorlati alkalmazás szempontjából is különösen fontos eredményeket szolgáltathat, ha a méréseket nem generált, hanem valamilyen valós gráfon, például a Wikidatá-n végezzük el, ami hozzájárulhat a szemantikus web egy hatékonyabbá tételéhez, így hosszútávon elősegítheti ennek a koncepciónak minél szélesebb elterjedését.

7 Irodalomjegyzék

[1] R. Angles és C. Gutierrez, „Survey of Graph Database Models," *ACM Comput. Surv.*, pp. 1:1.-1:39., 2008..

[2] R. Marko A. és P. Neubauer, „Constructions from Dots and Lines," *arXiv:1006.2361 [cs]*, 2010.

[3] F. Martin, „bliki: NosqlDefinition," 2017. [Online]. Available: https://martinfowler.com/bliki/NosqlDefinition.html. [Hozzáférés dátuma: 23 10 2017].

[4] D. S. Barrasa, „Debunking some RDF-vs-Property Graph Alternative Facts," Neo4j, 2017. [Online]. Available: https://www.youtube.com/watch?v=t1Mn178sEYg. [Hozzáférés dátuma: 23 10 2017].

[5] „Neo4j's Graph Query Language: An Introduction to Cypher," Neo4j, 2017. [Online]. Available: https://neo4j.com/developer/cypher-query-language/. [Hozzáférés dátuma: 23 10 2017].

[6] „Graph Data Processing with SQL Server 2017 and Azure SQL Database," Microsoft, 2017. [Online]. Available: https://blogs.technet.microsoft.com/dataplatforminsider/2017/04/20/graph-data-processing-with-sql-server-2017/. [Hozzáférés dátuma: 23 10 2017].

[7] R. Ian, W. Jim és E. Emil, Graph Databases, 2 szerk., O'Reilly Media Inc., 2015, pp. 1-170, 193-211.

[8] „DB-Engines Ranking - popularity ranking of graph DBMS," Db-engines.com, 2017. [Online]. Available: https://db-engines.com/en/ranking/graph+dbms. [Hozzáférés dátuma: 23 10 2017].

[9] K. Vojtěch, S. Martin és H. Irena, „Experimental Comparison of Graph Databases," *Proceedings of International Conference on Information Integration and Web-based Applications & Services*, p. 115:115–115:124, 2013.

[10] P. Szeredi, G. Lukácsy és T. Benkő, in *A szemantikus világháló elmélete és gyakorlata*, Typotex, 2005, pp. 61-116.

[11] D. Dominguez-Sal, N. Martinez-Bazan, V. Muntes-Mulero, P. Baleta és J. L. Larriba-Pey, „A discussion on the design of graph database benchmarks," in *Performance Evaluation, Measurement and Characterization of Complex Systems*, Springer, 2011, pp. 25-40.

[12] A.-L. Barabási, M. Pósfai, I. Seres és E. Kirchner, in *A hálozatok tudománya*, Budapest, Libri, 2016.

[13] H. Daniel, H. Aidan, R. Cristian, R. Carlos és Z. Enzo, „Querying Wikidata: Comparing SPARQL, Relational and Graph Databases," in *Lecture Notes in Computer Science*, 2016, pp. 88-103.

[14] „Az RDF alapfogalmai és absztrakt szintaxisa," W3c.hu, 2017. [Online]. Available: http://w3c.hu/forditasok/RDF/REC-rdf-concepts-20040210.html. [Hozzáférés dátuma: 23 10 2017].

[15] K. Markus és V. Denny, „Wikidata," *Communications of the ACM*, pp. 78-85, 2014.

[16] „SPARQL 1.1 Overview," W3.org, 2017. [Online]. Available: http://www.w3.org/TR/sparql11-overview/. [Hozzáférés dátuma: 23 10 2017].

[17] R. Marko A., „The Gremlin graph traversal machine and language (invited talk)," *Proceeding DBPL 2015 Proceedings of the 15th Symposium on Database Programming Languages*, pp. 1-10, 2015.

[18] Vinh Nguyen, B. Olivier és S. Amit, „Don't like RDF Reification? Making Statements about Statements Using Singleton Property," 2014. [Online]. Available: https://www.slideshare.net/ntkimvinh7/www2014-singleton-propertyfinal. [Hozzáférés dátuma: 23 10 2017].

[19] „Linux Azure VM sizes - Memory," Microsoft, 2017. [Online]. Available: https://docs.microsoft.com/hu-hu/azure/virtual-machines/linux/sizes-memory#esv3-series. [Hozzáférés dátuma: 23 10 2017].

[20] R. Marko, „Quantum Processes in Graph Computing,” 2016. [Online]. Available: https://www.slideshare.net/slidarko/quantum-processes-in-graph-computing. [Hozzáférés dátuma: 23 10 2017].

[21] „Graph processing with SQL Server and Azure SQL Database,” Microsoft, 2017. [Online]. Available: https://docs.microsoft.com/en-us/sql/relational-databases/graphs/sql-graph-overview. [Hozzáférés dátuma: 23 10 2017].

[22] M. Andrew, „Causal Consistency for Large Neo4j Clusters,” 2017. [Online]. Available: https://www.infoq.com/news/2017/04/causal-consistency-neo4j. [Hozzáférés dátuma: 23 10 2017].

[23] M. Charles, „Graph Processing Using Big Data Technologies,” 2014. [Online]. Available: https://www.infoq.com/news/2014/03/graph-bigdata-tapad. [Hozzáférés dátuma: 23 10 2017].

[24] H. Olaf és T. Bryan, „Foundations of an Alternative Approach to Reification in RDF,” 2014. [Online]. Available: https://arxiv.org/pdf/1406.3399.pdf. [Hozzáférés dátuma: 23 10 2017].

[25] „OpenLink Virtuoso,” OpenLink Software, [Online]. Available: https://virtuoso.openlinksw.com/. [Hozzáférés dátuma: 23 10 2017].

[26] R. Marko, „Gremlin: A Graph-Based Programming Language,” 2010. [Online]. Available: https://www.slideshare.net/slidarko/gremlin-a-graphbased-programming-language-3876581. [Hozzáférés dátuma: 23 10 2017].

[27] „TinkerPop3 Documentation,” Apache, 2017. [Online]. Available: http://tinkerpop.apache.org/docs/3.3.0/reference/. [Hozzáférés dátuma: 23 10 2017].

[28] „TITAN: Distributed Graph Database,” [Online]. Available: http://titan.thinkaurelius.com/. [Hozzáférés dátuma: 23 10 2017].

[29] „Concepts - Blazegraph,” Blazegraph, 2015. [Online]. Available: https://wiki.blazegraph.com/wiki/index.php/Concepts. [Hozzáférés dátuma: 23 10 2017].

[30] „blazegraph/tinkerpop3,” [Online]. Available: https://github.com/blazegraph/tinkerpop3. [Hozzáférés dátuma: 23 10 2017].

[31] M. Alaa, „No more joins: An overview of Graph database query languages,” IBM, 2017. [Online]. Available: https://developer.ibm.com/dwblog/2017/overview-graph-database-query-languages/. [Hozzáférés dátuma: 23 10 2017].

[32] „Introduction to Azure Cosmos DB,” Microsoft, 2017. [Online]. Available: https://docs.microsoft.com/en-us/azure/cosmos-db/introduction. [Hozzáférés dátuma: 23 10 2017].

[33] „Reification Done Right - Blazegraph,” Blazegraph, 2015. [Online]. Available: https://wiki.blazegraph.com/wiki/index.php/Reification_Done_Right. [Hozzáférés dátuma: 23 10 2017].

[34] W. Prof. David , „IntermediateRepresentation,” Princeton, [Online]. Available: http://www.cs.princeton.edu/courses/archive/spr03/cs320/notes/IR-trans1.pdf. [Hozzáférés dátuma: 23 10 2017].

[35] „Configuring Blazegraph,” Blazegraph, 2016. [Online]. Available: https://wiki.blazegraph.com/wiki/index.php/Configuring_Blazegraph. [Hozzáférés dátuma: 23 10 2071].

[36] „Request Units in Azure Cosmos DB,” Microsoft, 2017. [Online]. Available: https://docs.microsoft.com/hu-hu/azure/cosmos-db/request-units. [Hozzáférés dátuma: 23 10 2017].

[37] „Tunable data consistency levels in Azure Cosmos DB,” Microsoft, 2017. [Online]. Available: https://docs.microsoft.com/hu-hu/azure/cosmos-db/consistency-levels. [Hozzáférés dátuma: 23 10 2017].

[38] „BerkekeyDB,” Titan, 2015. [Online]. Available: http://s3.thinkaurelius.com/docs/titan/1.0.0/bdb.html. [Hozzáférés dátuma: 23 10 2017].

[39] „Bulk Loading,” Titan, 2014. [Online]. Available: http://s3.thinkaurelius.com/docs/titan/0.5.4/bulk-loading.html. [Hozzáférés dátuma: 23 10 2017].

[40] „Azure CosmosDB Graph traversal performance issue,” 2017. [Online]. Available: https://social.msdn.microsoft.com/Forums/azure/en-US/f1150594-259d-40d5-a283-e5c4e4e85480/azure-cosmosdb-graph-traversal-performance-issue?forum=AzureDocumentDB. [Hozzáférés dátuma: 23 10 2017].

[41] „Azure Cosmos DB Gremlin graph support,” Microsoft, 2017. [Online]. Available: https://docs.microsoft.com/en-us/azure/cosmos-db/gremlin-support. [Hozzáférés dátuma: 23 10 2017].

[42] „Planner hints and the USING keyword,” Neo4j, [Online]. Available: https://neo4j.com/docs/developer-manual/current/cypher/query-tuning/using/. [Hozzáférés dátuma: 23 10 2017].

[43] R. Jack, „Rhetorical Device: Triple Store,” w3.org, [Online]. Available: http://www.w3.org/2001/sw/Europe/events/20031113-storage/positions/rusher.html. [Hozzáférés dátuma: 23 10 2017].