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

Gráfadatbázisok teljesítményének vizsgálata valóélet-beli tudásbázison és lekérdezési mintákkal

TDK dolgozat

Készítette:

Kovács Tibor

Konzulens:

Simon Gábor

2019

Tartalomjegyzék

Kivonat **i**

Abstract **ii**

1. Bevezetés **1**

2. Gráfadatbázisok **3**
- 2.1. Adatmodell . . . 3
  - 2.1.1. Történet . . . 3
  - 2.1.2. Tulajdonsággráf alapú modellek . . . 4
  - 2.1.3. RDF modell . . . 6
- 2.2. Implementációk . . . 8
  - 2.2.1. Neo4j . . . 8
  - 2.2.2. Blazegraph . . . 9
  - 2.2.3. JanusGraph . . . 10
  - 2.2.4. Azure Cosmos DB . . . 11
  - 2.2.5. Orient DB . . . 12
  - 2.2.6. TigerGraph . . . 14
- 2.3. Benchmarkok . . . 16

3. Metaadatok modellezése **17**
- 3.1. Problémafelvetés . . . 17
- 3.2. Modellezési megoldások . . . 18
  - 3.2.1. Éltulajdonság modell . . . 18
  - 3.2.2. Négyeseken (quad) alapuló modell . . . 19
  - 3.2.3. RDR-alapú modellezés . . . 21
  - 3.2.4. Standard modell . . . 22
  - 3.2.5. Általánosított standard modell . . . 23
  - 3.2.6. N-Ary modell . . . 25
  - 3.2.7. Singleton tulajdonság modell . . . 26
  - 3.2.8. További lehetőségek . . . 28

4. Mérési paraméterek **29**
- 4.1. Adathalmaz . . . 29
- 4.2. Lekérdezések . . . 33
- 4.3. Mérési környezet . . . 37

5. Mérési eredmények **43**
- 5.1. Adatbetöltés . . . 45
- 5.2. Tárhely . . . 48
- 5.3. Teljesítmény, válaszidő . . . 51

5.4. Skálázhatóság 58
5.5. Skálázódás 61

6. Összefoglalás, konklúzió **70**

7. Továbbfejlesztési lehetőségek **72**

Irodalomjegyzék **73**

Kivonat

Megnézve az elmúlt évek „hype ciklusait”, azt tapasztalhatjuk, hogy a meglehetősen nagy fluktuáció ellenére vannak olyan elemek, melyek valamilyen formában folyamatosan jelen vannak az aktuálisan divatos, nagy potenciállal rendelkező technológiák között. Ilyen elem például a nagyméretű szemantikus hálók koncepciója, ami néha tudásgráfként vagy tudásbázisként, máskor szakértői rendszerként, megint máskor természetes nyelvű kérdés megválaszolásként hosszú évek óta fel-felbukkan a vezető előrejelzésekben, ami jól mutatja a téma fontosságát és aktualitását.

Manapság már természetes gondolatként adódik, hogy a hatékony kezelés érdekében ezen nagyméretű adathalmazokat adatbázis-kezelő rendszerekben tároljuk, azonban a kitűzött nemfunkcionális célok elérését szem előtt tartva a megfelelő implementáció kiválasztása már közel sem ennyire magától értetődően egyszerű feladat, ugyanis az elmúlt évek NoSQL forradalmának eredményeként újabb és újabb adatbázis technológiák jelentek meg vagy éledtek újjá. Ezen alternatív technológiák egyik fő irányát a gráfadatbázisok alkotják, amik jellegükből adódóan nyilvánvaló választásnak tűnhetnek tudásgráfok tárolására.

A megfelelő implementáció mellett a másik döntő faktor, ami egy rendszer valamennyi nemfunkcionális paraméterét lényegesen befolyásolja, a legmegfelelőbb logikai modell kiválasztása. Tudásgráfok esetében ez elsőre magától értetődőnek tűnik, ám, ha reifikációt is támogat a tudásbázisunk, a modellezés kulcskérdésévé a metaállítások tárolásának miképtje válik, azaz a reifikációs módszer megválasztása. A legtöbb modellezési problémához hasonlóan a reifikációra is számos lényegileg különböző megoldás ismert, melyek spektrumát tovább tágítják az egyes NoSQL adatbázis-kezelők adatmodelljei között rendszerint megtalálható különbségekből adódó lehetőségek.

Dolgozatomban több különböző szempont szerint összehasonlítom a jelenleg népszerű, rendszerint koncepcionálisan eltérő gráfadatbázisokat valós szemantikus hálókon, valódi felhasználók által megfogalmazott lekérdezési minták alapján. Az összehasonlítás kiterjed az adatbázis-kezelők válasz idejére, betöltési idejére, tárhelyigényére és skálázódására, illetve skálázhatóságára eltérő logikai reprezentációs modellek mellett. Az eredmények alapján láthatóvá válnak az egyes rendszerek erősségei és gyengeségei, amik alapján meghatározható, hogy egy konkrét cél elérése érdekében, mint például leggyorsabb válaszidő, legkisebb tárhely stb., melyik implementáció-modell páros az optimális választás.

Abstract

Regarding the „hype cycles” of the previous years, one may find that even it has significant fluctuations, there are elements that are constantly resurfacing among the actual trendsetting technologies, that are believed to have a great future potential. Large-scale semantic networks are such a domain, they are masquerading in the leading forecasts for long years, sometimes as knowledge graphs or knowledge bases, sometimes as advisory systems and other times as natural language question answering technologies, which clearly shows the importance and actuality of the topic.

Nowadays it is a natural idea to store this kind of large-scale datasets in database management systems for the sake of efficiency, but selecting the right implementation, while satisfying the proper non-functional requirements, is clearly not so trivial, as newer and newer database technologies were born or reborn, in the course of the NoSQL revolution. One of the main branches of these alternative technologies is the graph database approach, which seems an obvious choice by its nature to store knowledge graphs.

Beside the proper implementation, the other main factor that affects nearly all non-functional parameter is finding the optimal logical representation. As for knowledge graphs, this seems a trivial task at first, but the support for reification makes the storage approach of the reification the key modelling issue, i.e. picking the right reification technique. Similarly to almost every modelling challenge, a couple of significantly different approaches are known to tackle this problem, and the spectrum is even wider considering the possibilities from the varying data modelling capabilities of the NoSQL database manage systems.

In my paper, I compare the currently popular, conceptionally different graph databases from several aspects using real-life datasets and query patterns based on real users’ queries. The comparison includes response time, data loading time, storage size, scaling and scalability of the database manage systems on different logical representation models. Analysing these results, I determine the strengths and weaknesses of each of the database implementations, in order to decide which implementation-model pair is the optimal choice to achieve a particular purpose, like fastest response time, smallest storage size, etc..

1. fejezet

Bevezetés

A technológia folyamatos fejlődésének következményeként, ahogy a digitalizáció az életük újabb és újabb területeire tört be, soha nem látott mennyiségű részben strukturált adat kerül a különböző informatikai rendszerekbe. Ez az információs robbanás paradigmaváltást hozott az adatkezelés területén is, ahol a NoSQL forradalom révén új technológiák nyertek teret – például a gráfadatbázisok, megtörve a relációs modell korábbi egyeduralkodását. Ezen technológiák egyre kifinomultabbá válása lehetőséget adott arra is, hogy olyan helyeken is jól használható információs rendszerek kezdenek megjelenni, ahol korábban – részben a technológia, részben a relációs rendszerek korlátai miatt – elképzelhetetlenek voltak.

Az ilyen területek egy jó része a szemantikus hálók koncepcióját veszi alapul, és ugyan rendszerint más-más néven, de évről évre jelen vannak az éppen divatos technológiák [1] között, ami alátámasztja a témával kapcsolatos munkák relevanciáját. A korábban említett paradigmaváltás azonban nem csak a technológiát érintette, de az adatok jellegére vonatkozó aspektusa is volt, ugyanis a korábban domináns adatcentrikusság helyett egyre elterjedtebbé válnak a minél gazdagabb metaadat tárolást lehetővé tevő funkciók – elegendő például a hashtag-re vagy a fényképekhez kapcsolt hely adatokra gondolni. Ezen kiegészítő adatok hatékony tárolása új kihívások elé állítja a rendszereket, melyek megoldásának alapeszköze a megfelelő logikai, pontosabban reifikációs modell kiválasztása.

Körülnézve a gráfadatbázisok világában azt láthatjuk, hogy gyors változásokra reagálva gyakran jelennek meg új technológiák, melyek rendszerint testreszabott benchmarkokkal a saját rendszerük erősségeit kiemelve igyekeznek a konkurenciánál jobbnak feltüntetni magukat. Csupán az eredményeket megvizsgálva meglehetősen nehéz egy valós projekthez kiválasztani a megfelelő adatbázis-kezelőt, hiszen az egyes mérések eltérő körülmények között és eltérő munkafolyamatok szempontjából jellemzik a bevont rendszereket, így rendszerint ellentmondanak egymásnak. Ráadásul a publikusan elérhető, különböző gráfadatbázisokat vizsgáló független benchmarkok szinte kivétel nélkül figyelmen kívül hagyják a logikai modell fontosságát. Munkámmal ennek a hiánynak a pótlását tűztem ki célul.

A dolgozat elkészítése során tehát a feladatom az volt, hogy részletesen megvizsgáljam az egyes gráfadatbáziskezelők teljesítményét valós (és azonos) körülmények között. Jelen dolgozat egy természetes evolúció következő lépésének tekinthető, ugyanis a korábbi TDK dolgozatom, illetve szakdolgozatom szintén gráfadatbázisok teljesítményelemzéséről szólt, azonban jelenleginél lényegesen egyszerűbb megközelítést alkalmazva. Ezen munkáimban felvetett továbbfejlesztési lehetőségek mentén tovább haladva lépésről-lépésre lecseréltem a mérési konfiguráció valamennyi elemét. Először a vizsgálathoz kiválasztottam egy valódi tudásbázist – a Wikidata-t, majd valóélet-beli lekérdezések alapján meghatároztam egy mérési lekérdezés készletet, melyeket felhasználva viszonylag részletes teljesítményprofilt

tudtam előállítani az egyes adatbázis kezelőkhöz. További fejlesztésként bővítésre került mind a megvizsgált logikai modellek, mind a fizikai implementációk halmaza, illetve a mérési eszközkészlet is általánosabbá vált.

Dolgozatom két nagy részből épül fel: az első részben ismertetem, illetve bevezetem a mérések során használt technológiákat és fogalmat, majd a második részben részletekbe menően leírom a mérési környezet különböző paramétereit és magukat az eredményeket.

Az elméleti bevezetést a gráfadatbázisok fogalmának bevezetésével kezdem. Röviden összefoglalom a kialakulásuk történetét és motivációját, kitérve néhány manapság elterjedt alkalmazási területükre. A történeti kontextust az adatmodell fogalom tisztázása követi, aminek elsődleges feladata annak tisztázása, hogy mit is nevezünk gráfadatbázisnak. A fogalom bevezetését követően bemutatom a két széles körben elterjedt adatmodell koncepciót, a tulajdonsággráf modellt és az RDF megközelítést, továbbá megemlítek néhány további, kevesek által implementált koncepciót is.

Az adatmodellek ismertetését az általam megvizsgált adatbázis technológiák (implementációk) rövid bemutatása követi, kiemelve egy-egy adatbázisspecifikus aspektust. Elsőként bemutatom a Neo4j-t, kitérve az általa használt Cypher nyelvre. Ezt a Wikidata által is használt Blazegraph és SPARQL mintaillesztő nyelv rövid ismertetése követi, majd JanusGraph rendszert, és a megvalósított TinkerPop gráf stack-et mutatom be röviden. Ezután következik az Azure Cosmos DB és a felhő-alapú technológiákban rejlő lehetőségek leírása. Utolsó előttiként a TigerGraph, és az előfordított, natív végrehajtású megközelítés következik. Végül bemutatom az OrientDB-t, és a multimodális adatbáziskezelők koncepcióját.

A gráfadatbázisokról szóló rész zárásaként ismertetek néhány, gráfadatbázisokhoz kapcsolható benchmark eredményt, külön kitérve mindegyiknél arra, hogy miben tér el az általam készített méréstől.

A gráfadatbázisok után felvetem a reifikáció problémáját, majd bemutatom az elterjedt modellezési megoldásokat – az éltulajdonság, a standard, az általánosított standard, az n-ary és a singleton tulajdonság modelleket, illetve a négyes-alapú és RDR kódolásokat, kitérve mindegyiknél az előnyökre és a hátrányokra is.

Az elméleti részt követően ismertetem a mérési környezetet annak fontosabb paraméterei mellett. Először bemutatom a Wikidata-t, azaz a mérések alapjául szolgáló adathalmazt. Ehhez kapcsolódóan ismertetem egy példán keresztül a kiindulási adatmodellt, és azt a transzformációt, ami ebből az adathalmazból a különböző méretű gráfokat előállítja.

Az adathalmazt a mérési lekérdezések szintetizálási folyamatának, és maguknak a lekérdezéseknek a bemutatása követi. Ennek részeként részletesen leírom a lekérdezések előállításának folyamatát, és megindokolom, hogy miért lehet ezeket valós lekérdéseknek tekinteni. A szintetizáló folyamaton túl az eredmény lekérdezési mintákat is bemutatom, illetve szemléltetem.

A mérési paraméterek zárásaként ismertetem a mérési infrastruktúrát, aminek részként bemutatom a mérési rendszer topológiáját, az egyes implementációk fizikai é szoftveres környezetét, valamint a lényeges alkalmazott konfigurációs beállításokra is kitérek.

Ezt a tényleges mérési eredményeim ismertetése követi. Ennek során bemutatom, hogy az egyes vizsgált implementációk hogyan teljesítenek skálázhatóság, skálázódás, adatbetöltési idő, tárhely idény és teljesítmény szempontjából, minden szempont esetén egyesével kitérve az egyes adatbázis-kezelők sajátosságaira.

A dolgozat zárásaként összefoglalom az elvégzett munkámat és a kapott mérési eredményeket, majd felvetek néhány lehetőséget arra, hogy hogyan lehetne továbbfejleszteni a mostani munkámat.

2. fejezet

Gráfadatbázisok

2.1. Adatmodell

2.1.1. Történet

A matematika történet a gráfelmélet születését a XVIII. századhoz, pontosabban Euler königsbergi hidak problémájának megoldásával kapcsolatos munkájához köti [43]. A matematikusok hamar felismerték, hogy gráfok jól illeszkednek az emberi gondolkodáshoz, a valóélet-beli problémák egy jelentős részét intuitív módon lehet modellezni, illetve formalizálni a segítségükkel. Ennek következtében meglehetősen mély fogalmi rendszert és gazdag eszközkészletet építettek a gráfok koncepciója köré, melynek fejlődése napjainkban is folyamatban van.

A gráfelmélet fejlődése szempontjából óriási lökést jelentett a XX. században megjelent számítógép. A gráfok már kis erőforrással rendelkező környezetben is numerikusan pontosan használhatók – például az analízissel ellentétben –, így használatuk hamar megjelent és elterjedt a digitális környezetben. A számítógépes rendszerek elterjedése magával vonta az algoritmuselmélet robbanásszerű fejlődését is, ami új gráfalgoritmusok megalkotásával szintén hozzájárult a gráfok még intenzívebb használatához.

Mindezek ellenére az adatbázis technológiák területén csak a '80-as évek végén jelentek meg olyan adatbázis-kezelő rendszerek (DBMS), amik gráf-alapú adatmodellre épültek. Az akkor már szinte kizárólagosan használt relációs rendszerek különböző gyengeségi a '90-es években kezdtek megjelenni: ahogy az információs rendszerek egyre inkább elkezdtek elterjedni, újabb és újabb, lényegesen különböző követelményeket kezdtek támasztani az adatbázisokkal szemben – például a sémamentesség, manapság pedig a skálázhatóság –, felismerhetővé vált, hogy egy adatmodell nem tudja mindezen igényeket hatékonyan kielégíteni, ami az úgynevezett NoSQL technológiák egész sorának születését és elterjedését eredményezte, köztük a gráfadabázisokét is.

Ezen technológiák rendszerint nem olyan általános célú eszközök, mint a relációs rendszerek, hanem egy-egy speciális területen tudnak csak igazán hatékonyak lenni, például a dokumentum adatbázisok félig strukturált adatok kezelésében jeleskednek, a gráfadatbázisok pedig kusza, szűrű kapcsolati hálóval rendelkező adatstruktúrák esetén igazán hatékonyak. Ezt azáltal tudják elérni, hogy általában nincs szükség a relációs világban megszokott szigorú sémadefiniálásra. Ezen rendszerek további általános jellemzője, hogy horizontálisan jól skálázhatók – ami kimondottan előnyös például egy felhő-alapú környezetben, cserébe gyengébb konzisztencia szintet tudnak csak biztosítani a CAP-tétel következményeként [79].

Ahogy az a NoSQL világról általában elmondható, nagyon sok különböző megközelítésű és kiforrottságú technológia van ugyanarra a problémára, melyek rendszerint semmilyen formában nem kompatibilisek egymással. Ez részben a NoSQL fiatalságának, részben az

egyes adatbázis-kezelő családokból hiányzó olyan egyeduralkodó, szabványos technológiáknak köszönhető, mint amilyen az SQL a relációs világban. Ez a sokszínűség a gráfadatbázisok területén is jelentkezett, az elmúlt években folyamatosan jelentek meg újabb és újabb technológiák, és teszik ezt manapság is.

Ezen diverzitás miatt fontos tisztázni, hogy mely rendszereket tekintünk egyáltalán gráfadatbázis-kezelőnek. Magas szinten nézve azt mondhatjuk, hogy azt a DBMS-t nevezzük gráfadatbázisnak, ami gráfok koncepcióját használja fel az adatmanipuláció logikai alapelemeként, a fizikai megvalósítástól függetlenül. A fizikai és logikai modellek szétválasztásának legfőbb oka a manapság egyre elterjedtebb, úgynevezett multimodális rendszerek koncepciója, melyek egy rendszeren belül több különböző adatmodell szerint is kezelni tudják az adatokat – azaz például ugyanazt az adathalmazt képesek gráfként, és dokumentumként vagy relációként manipulálni.

Ennek a Codd-i megközelítésnél [10] általánosabb definíciónak több fontos következménye is van. Az egyik, hogy nincs meghatározva a pontos adatstruktúra – matematikai értelemben vett gráf elemekkel, vagy például rendezett hármasokkal dolgozik a rendszer –, csak annyi megkötés van, hogy a gráfok koncepciójára leképezhető legyen. A másik pedig, hogy elegendő gráf-alapú interfészt nyújtani a rendszernek, annak belső végrehajtása történhet másik adatmodell szerint is. Emiatt az olyan rendszerek is gráfadatbázisnak tekinthetők, mint az Azure CosmosDB a Gremlin API-t, vagy a Microsoft SQL Server 2017 az SQL gráf kiterjesztéseit használva, nem csak az olyan natív gráfadatbázisok, mint a Neo4j vagy a TigerGraph.

A gráfok koncepcióját alapul véve számos különböző megközelítés és modell bővítés lehetséges [72], melyek közül a két legelterjedtebb a tulajdonsággráf és az RDF adatmodell.

2.1.2. Tulajdonsággráf alapú modellek

A manapság legelterjedtebb gráfadatbázis-kezelők az úgynevezett tulajdonsággráfokat használják az adatmodelljük alapjául. A koncepció a matematikai gráfok definíciójából indul ki:

Definíció 1. (Irányított) Gráf

Legyen V halmaz: $V \neq \emptyset$ és legyen $E \subseteq V \times V$. Ekkor a $G(V, E)$ rendezett párt gráfnak nevezzük, aminek a V-beli elemek a csúcsai, az E-beli elemek pedig az élei. ▪

Ezen gráf koncepció segítségével már leírhatók a modellezett domain objektumai – ezek lesznek a gráf csúcsai, illetve a köztük levő kapcsolatok – ezekből pedig az élek lesznek. Fontos, hogy az élhalmaz definíciójában szereplő Descartes-szorzat tulajdonságai miatt az élek egyirányú kapcsolatot jelentenek, melyek iránya általában modellezési kérdési, illetve az élek halmaz jellegéből adódik, hogy nem engedélyezettek a két adott csúcs között futó párhuzamos élek.

Mivel ez a kifejezőerő általában még nem elégendő, számos bővítést és módosítást szükséges bevezetni. A gráf modell általánosítására számos lehetőség adódik ezek közül a tulajdonsággráfok az ezt követően ismertetetteket támogatják általában.

Definíció 2. Csúcs címke

Legyen $G(V, E)$ gráf és jelölje S a szövegek halmazát. Ekkor az $f_{label} : V \to S^n$ függvény által a csúcsokhoz rendelt értékeket a csúcs címkéinek nevezzük. ▪

Látható, hogy a csúcs címkék csak szöveges adatok lehetnek, illetve, hogy egy csúcshoz több címke is tartozhat. Ezek a kiegészítő szövegek lehetővé teszik (többek között) típusinformációk hozzárendelését a csúcsokhoz. Fontos kiemelni, hogy ugyan általában típusra vonatkozó adatok tárolására használják, általános célú bővítés, így hozzáadása nem

jelent semmilyen sémakényszert a csúcspontra nézve – tehát lényegesen lazább jelentése van, mint a relációs modellben egy rekordnak egy relációhoz tartozása.

Definíció 3. Él címke
Legyen $G(V, E)$ gráf és jelölje S a szövegek halmazát. Ekkor az $f_{rtype} : E \rightarrow S$ függvény által az élekhez rendelt értéket az él címkéjének (vagy típusának) nevezzük. ▪

A csúcsokhoz hasonlóan az élekhez is rendelhetünk kiegészítő szöveges adatokat, fontos különbség a csúcsokhoz képest azonban, hogy egy élhez egy ilyen információ csatolható. Ezen okból szinte kizárólag az élek típusát, szemantikáját szokás itt megadni. A relációs modell analógiáját tekintve ez a kapcsolótábla, vagy az idegen kulcs által reprezentált kapcsolat típusa.

Definíció 4. Csúcs tulajdonságok
Legyen $G(V, E)$ gráf és jelölje S a szövegek, T az eltárolható egyszerű objektumok halmazát. Ekkor az $f_{prop} : V \rightarrow (S \times T^n)^m$ függvény által a csúcsokhoz rendelt párokat a csúcs tulajdonságainak nevezzük. ▪

A gráf csúcspontjai csak az egyes entitásokat reprezentálják, azok attribútumait nem tartalmazzák. Ezt a hiányosságot hivatott kiküszöbölni a csúcsokhoz rendelt tulajdonságok bővítés, illetve innen kapta a nevét az adatmodell is. A definíció alapján látható, hogy minden csúcshoz kulcs-érték párokat lehet rendelni, ahol a kulcs szöveges adat lehet, míg az érték típusa a konkrét adatbázis-kezelő implementációtól függ. A legtöbb rendszer támogatja a szöveges, különböző numerikus és logikai értékek egyedi és kollekció-alapú tárolását – azaz egy kulcshoz több azonos típusú egyszerű érték is tartozik, de ezeken felül is sokszor lehetőség van dátum, lokáció tárolására vagy más komplexebb adat tárolására is [87, 15].

Definíció 5. Él tulajdonságok (minősítők)
Legyen $G(V, E)$ gráf és jelölje S a szövegek, T az eltárolható egyszerű objektumok halmazát. Ekkor az $f_{qual} : E \rightarrow (S \times T^n)^m$ függvény által az élekhez rendelt párokat az él tulajdonságainak, vagy a kapcsolat minősítőinek nevezzük. ▪

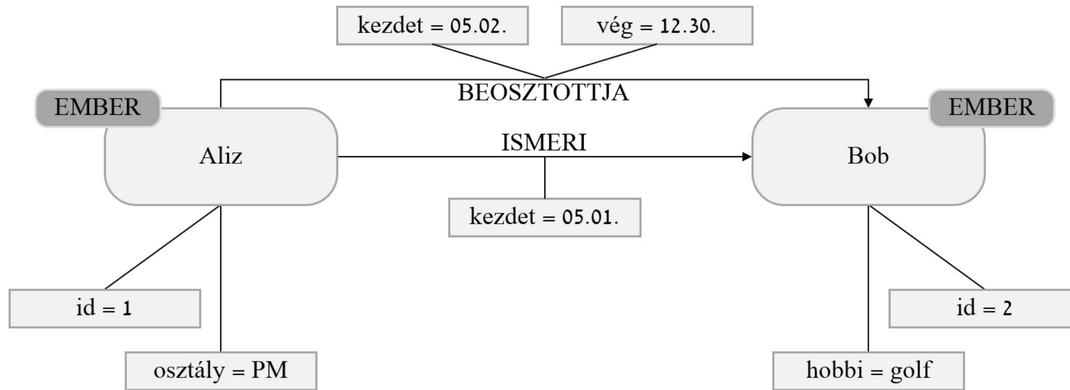
A modell egyik érdekes bővítése, hogy – a csúcsokhoz hasonlóan – az élekhez is kulcs-érték párok formájában tulajdonságok rendelhetők, melyek szemantikailag magát a kapcsolatot jellemzik. A modellezés igazán intuitív kifejezőerejét például itt észrevehető, különösen, ha a relációs modellel hasonlítjuk össze, ahol ugyanez csak kapcsolótáblával és idegen kulcsok bevezetésével érhető el.

A valóélet-beli modellezési szituációkban gyakran előfordul az az eset, hogy két csúcs által reprezentált entitás között több különböző kapcsolat is van – például Aliz nem csak ismert Bobot, hanem a felettese is. Ilyen esetek kezelése miatt az adatbázis-kezelők a gráfok helyett általában multigráfokkal dolgoznak.

Definíció 6. Multigráf
Legyen V halmaz: $V \neq \emptyset$ és legyen $E \subseteq V \times V$ *multihalmaz* [50]. Ekkor a $G(V, E)$ rendezett párt multigráfnak nevezzük, aminek a V-beli elemek a csúcsai, az E-beli elemek pedig az élei. ▪

Az ezen bővítések bevezetését követően eredményül egy általános, mégis intuitív adatmodellt kapunk, melyet a 2.1 ábra mutat be. Az adatmodell igazi előnyei a kapcsolatok kezelésében van, belátható ugyanis, hogy – a relációs modellel ellentétben – nincs szükség idegen kulcsok vagy kapcsolótáblák definiálására többes kapcsolatok esetén, hanem azok az emberi gondolkodás szerint képződnek le.

Az általam megvizsgált összes adatbázis-kezelő rendszer valamilyen formában támogatja ezt a gráfmodellt.



2.1. ábra. Egy példaállítás, ami valamennyi modell bővítést tartalmazza, továbbá a sémamentességet is bemutatja.

2.1.3. RDF modell

A web elterjedésével robbanásszerűen nőni kezdett az elérhető adat mennyisége, mely életre hívta a különálló szolgáltatásra épülő egyre komplexebb, információkat integráló rendszereket. Ennek megvalósításához olyan technológiára volt szükség, ami általános, a strukturált és félig strukturált adatok kezelését jól támogatja, és képes a nagyon sűrű, komplex kapcsolatok hatékony kezelésére, így a gráfadatbázisok használata kézenfekvőnek tűnt. Tekintve, hogy az imént bemutatott tulajdonsággráf modell a sok bővítése által a gépek számára viszonylag bonyolult és nehezen értelmezhető, így kidolgoztak és szabványosítottak egy sokkal egyszerűbb gráfmodellt, a Resource Description Framework-öt (RDF-et) [63].

Az RDF adatmodell alapgondolata arra a megfigyelésre vezethető vissza, hogy a tulajdonsággráfok szinte minden bővítését vissza lehet vezetni egyszerű kapcsolatokra (élekre), ugyanis a legtöbb esetben két gráf elem kapcsolódik össze valamilyen értelemben – például a csúcs címke leírható úgy is, mint az eredeti csúcs és a címkéhez újonnan bevezetett csúcs között futó „címkéje” típusú él. Ebből kifolyólag a kiindulási gráf koncepciót mindössze a multigráf és a kötelezően használandó él típus kiterjesztésekkel bővíti. Az egyedüli problémásan leképezhető bővítés az él tulajdonságok használata, azonban ezek leképzését a Metaadatok modelletése (3.) fejezet részletesen tartalmazza.

Az így kapott gráf modellben mindössze a gráf csúcsai, és a köztük futó tipizált élek találhatók. Ha a csúcsokat megvizsgáljuk, akkor az adódik, hogy két okból lehet egy csúcs a gráfban: vagy egy eredetileg is modellezett entitást jelképez, vagy a tulajdonsággráf bővített elemeinek leképzésekor került bevezetésre. Ebből kifolyólag az RDF adatmodellben kétféle „csúcs” reprezentálása lehetséges:

1. Lehet a modellezett domain-ből származó entitás, ebben az esetben valamilyen egyedi azonosítóval kell rendelkeznie. Ezeket az egyedeket az azonosítójukból képzett egyedi azonosító erőforrással (URI-val) lehet reprezentálni.

2. Lehet a bővítések leképzéseként előálló csúcs, ebben az esetben biztosan egyszerű tartalommal rendelkező adatról van szól – például szöveges vagy numerikus értékről.

Egy meglehetősen logikus további korlátozás bevezetésével az ezen a ponton még három lehetséges adat fajta (erőforrás, egyszerű adat, él típus) kettőre csökkenthető, mindössze azt kell kikötni, hogy az él típusa is egyedileg azonosított legyen, azaz URI-val rendelkezzen. Az RDF mellett más adatmodellekbe történő átalakításokat is jól bemutatja a [72] 3. ábrája.

Ha az eddig ismertetett koncepciót használva el szeretnénk tárolni a gráfot, akkor ezt legegyszerűbben éllistás reprezentációban tehetjük meg, amiben az alábbi három adat megadására van szükség: a forrás csúcs URI-ja, az él típusának URI-ja és cél csúcs adatai – utóbbi esetén vagy az URI-ja, vagy a tartalma. Ezen hármas sorrendjének lekötésével és szemantikus általánosításával jutunk el az állítás definíciójához:

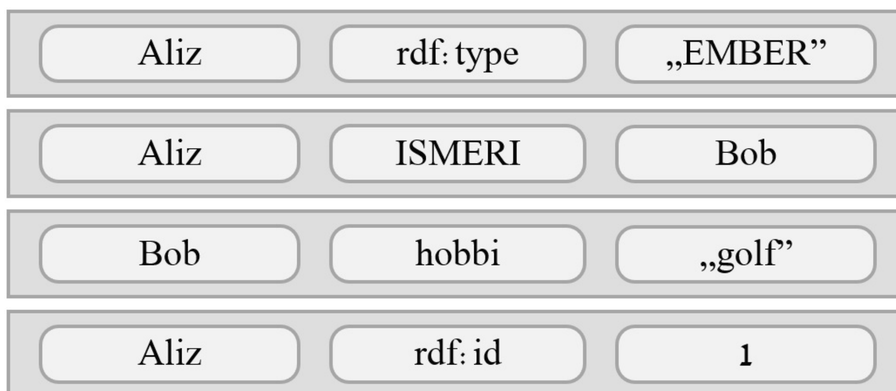
Definíció 7. RDF állítás

Jelölje URI az erőforrások halmazát, T az eltárolható egyszerű objektumok halmazát. Ekkor $S \in URI, P \in URI, O \in (URI \cup T)$ esetén az (S, P, O) rendezett hármast (RDF) állításnak nevezzük, melyben S-t az állítás alanyának, P-t az állítás állítmányának és O-t az állítás tárgyának nevezzük. ▪

Az RDF, vagy az állítások jellege miatt gyakran „triple store”-nak nevezett adatbázisok olyan speciális gráfadatbázisok, amik nagy mennyiségű állítást (azaz RDF hármast) tudnak nagyon hatékonyan kezelni. Az adatmodell tehát végső soron nem csúcsokra és élekre, hanem állításokra épül - ahogy a 2.2 ábrán is látható, amiből több minden is következik:

- Mivel csak állításokat lehet az adatbázisba felvenni, erőforrást önmagában nem, így 0-ad fokú csúcsok nem reprezentálhatók RDF-ben.
- Tekintve, hogy a csúcsok és az élek is erőforrásokra képződnek le, az RDF modellben nincs különbség a kettő között. Ebből az következik, hogy az RDF szintaktika le tud írni olyan gráfszerű struktúrát is (a tulajdonsággráf modellben gondolkodva), amiben egy él egy másik élből indul ki, de ezt az RDF szabvány nem engedi.
- Egységes URI-használati konvenciók mellett nagyon egyszerű különböző helyekről érkező adatok integrálása.
- Abból kifolyólag, hogy mind a struktúra (RDF hármas), mind a szemantika (alany, állítmány, tárgy) kötött, a gépek számára is könnyen feldolgozható és értelmezhető.

Habár az RDF közvetlenül nem használja a gráfokhoz köthető fogalomrendszert, a leírt gondolatmenetet visszafelé eljátszva könnyen belátható, hogy tulajdonsággráf modell és az RDF között létezik transzformáció mindkét irányba. Ennek, az erre épülő általánosan elterjedt állítás reprezentációnak – az alany és a tárgy két csúcs, amik között az állítmány típusának megfelelő irányított él fut –, valamint az erre épülő SPARQL gráfminta-illesztő nyelvnek köszönhetően tekinthető az RDF speciális gráfadatbázisnak.



2.2. ábra. A tulajdonsággráfnál bemutatott példa RDF modellre leképzett reprezentációjának egy részlete.

A méréseimben résztvevő rendszerek közül egyedül a Blazegraph használ ilyen „triple store” adatmodellt.

2.2. Implementációk

2.2.1. Neo4j

A gráfadatbázis-kezelők között egyértelműen kiemelt helyzetben van a Neo4j, ami a DB-Engines gráfadatbázisokra vonatkozó népszerűségi listáját annak 2013-as megjelenése óta folyamatosan magasan vezeti, ugyan egyre kisebb előnnyel [16].

A rendszer a korábban bemutatott tulajdonsággráf koncepcióra épül, a dolgozatban bemutatott összes gráf modell bővítési elemet támogatja. A Neo4j a saját adatmodelljét „labeled property graph”-nak hívja, ami csúcsokat (*nodes*), kapcsolatokat (*relationships*), tulajdonságokat (*properties*) és címkéket (*labels*) tartalmaz. A csúcsok és a kapcsolatok tulajdonságokkal rendelkezhetnek, melyek olyan kulcs-érték párok lehetnek, amik a kulcsai szövegek, értékei pedig a Java nyelv primitív típusú értékei, ugyanis a rendszer maga erre a futtatókörnyezetre épül. Minden csúcs rendelkezhet címkékkel, a kapcsolatok pedig szigorúan irányítottak, és mindig rendelkeznek egy típussal is. Könnyen belátható, hogy ezek a fogalmak közvetlenül megfeleltethetők a tulajdonsággráf modellnél bemutatott kiegészítésekkel [71].

A legtöbb másik NoSQL adatbázistól megkülönbözteti a Neo4j-t, hogy támogatja a hagyományos értelemben vett tranzakciókat, azaz a NoSQL világban megszokott „eventually consistent” megközelítésen túl az erős konzisztencia biztosítására is képes [71].

Alapvetően egy on-premise adatbázis-kezelőnek indult a rendszer (saját gépre feltelepítve működik), azonban a felhő technológiák forradalmához csatlakozva mára bevezettek olyan fejlesztéseket, ami jól skálázhatóvá tette az architektúráját, melynek eredményeként valamennyi elterjedt cloud platformon lehetőség van több klaszteres (előtelepített virtuális gépként igénybevehető) adatbázisként is igénybe venni [51].

A Neo4j natív gráfadatbázisnak [44] tekinthető abban az értelemben, hogy csak gráfalapú adatmodell szerint működik, és mind a logikai, mind a fizikai szinten gráf struktúrákkal dolgozik. A tárolási szinten az egyes gráf elemek külön fájlokba kerülnek, minden elem fix struktúrájú, amik lényegileg fizikai mutatókból álló egy-, illetve kétirányú láncolt listák [71]. Ez a direkt összekapcsoltság okozza, hogy a kapcsolatok menti navigációk nagyon gyorsak, hiszen nem indexeken keresztüli keresés történik.

A natív jelleg másik aspektusa a natív végrehajtás - azaz, hogy egy lekérdezés leírása és kiértékelése azonos adatmodell szerint történik, melynek leírására a deklaratív jellegű Cypher nyelv használható. Ugyan a rendszer megalkotásakor már léteztek szabványosított gráfminta-illesztő nyelvek – például a Balzegraph-nál bemutatott SPARQL, ezeket rendszerint más adatmodellhez dolgozták ki (például RDF-hez), emiatt volt szükség új lekérdezőnyelv bevezetésére. A nyelv viszonylag tömör, a gráfminták leírását ASCII karakterek segítségével meglehetősen intuitív módon lehet leírni, ez a 2.1 kódrészleten is látható, ami *Bob ismerőseinek kedvenc hobbijait kérdezi le*:

```
MATCH (bob:Person)-[:KNOWS]->(:Person)-[:FAVORITE_HOBBY]->(hobby:Activity)
WHERE bob.name = "Bob"
RETURN hobby.name;
```

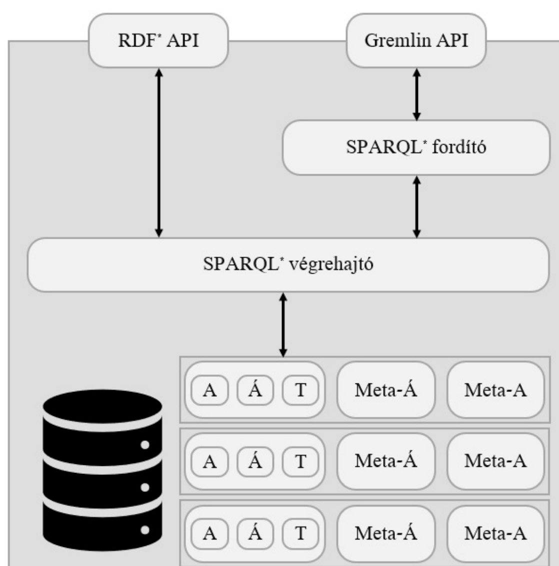
2.1. kódrészlet. Példaállítás Cypher nyelven.

A megvizsgált rendszerek közül az egyik legkiforrottabb programozási interfésszel rendelkezik – bár a relációs világhoz képest még ez is nagyon kezdetleges állapotú –, továbbá a kiváló dokumentáció és a sok elérhető példa miatt talán a legkönnyebben használható gráfadatbázis-kezelő.

2.2.2. Blazegraph

A mérések alapjául szolgáló Wikidata hivatalos lekérdező szolgáltatása [94] a Blazegraph-ot – vagy korábbi nevén BigData-t – használja adatbázis-kezelőként, ezért is került be a mérési implementációk csoportjába. A Neo4j-hez hasonlóan ez a rendszer is Java futtató-környezetre van megírva, azonban jórészt nyílt forráskódú [6].

A rendszer jellegét tekintve mind a tulajdonsággráf modellt, mind az RDF adatmodellt megvalósítja, emiatt részben multimodálisnak tekinthető (erről részletesebben az OrientDB-nél írok). Habár mindkét adatmodellt képes használni [8], mégis elsősorban „triple store” adatbázisnak tekinthető, ugyanis a fizikai adattárolás RDF* [65] modell szerint történik, továbbá a tulajdonsággráf modellre megfogalmazott Gremlin nyelvű lekérdezések (amiről pedig a JanusGraph-nál lesz részletesebb szó) is először SPARQL* nyelvre fordulnak le, és azok kerülnek ténylegesen végrehajtásra [86]. Ennek az architektúráját mutatja be a 2.3 ábra:



2.3. ábra. A Blazegraph magasszintű architektúrája.

Ezzel kapcsolatban fontos megjegyezni, hogy a Blazegraph valódi multimodalitást valósít meg, azaz lehetőség van a két adatmodell közötti teljes átjárásra. Ez azt jelenti, hogy az egyik utasítás beküldhető imperatív Gremlin nyelven, a következő pedig a deklaratív szemléletű SPARQL nyelven – legyen szó akár lekérdezésről, akár módosításról vagy adatfelvételről.

A Blazegraph tehát tárolási modellként az RDF-et, pontosabban annak egy nem szabványos bővítését, az RDF*-ot használja, ami a reifikáció natív támogatásával kapcsolatos bővítéseket tartalmaz (lásd. 3 fejezet), illetve kulcsszerepe van a tulajdonsággráf – RDF leképezésben is (az éltulajdonságok leképzésekor) [8]. A RDF modellben tárolt adatokat – a legtöbb triple store adatbázishoz hasonlóan – a már létező, szabványos SPARQL [77] lekérdezőnyelv segítségével lehet manipulálni, ami egy daklaratív gráfminta-illesztő nyelv. Ez lehetőséget ad – akár változókat is tartalmazó hármasok formájában – komplex részgráfok leírására, aminek minden lehetséges egyezését megkeresi a rendszer. A 2.1 kódrészlet SPARQL megfelelője a 2.2 kódrészleten látható:

```
SELECT ?hobbyName
WHERE {
    ?bob rdf:name "Bob" .
    ?bob example:knows ?somebody .
    ?somebody example:favoriteHobby ?hobby .
    ?hobby rdf:name ?hobbyName? .
}
```

2.2. kódrészlet. Példaállítás SPARQL nyelven.

Abból kifolyólag azonban, hogy RDF* szerint tárolt adatokat kell lekérdezni, szükség volt a SPARQL nyelv bővítésére is, amit a Blazegraph SPARQL*-nak nevezett el. Ez lehetővé teszi teljes hármasok alanyként viselkedését, ezáltal lehetőséget adva a reifikált adathalmazok intuitív lekérdezésére és módosítására.

A Blazegraph-hoz viszonylag jó, stabil könyvtárak érhetők el Java környezetben, más platformon viszont jelenleg még csak nagyok kezdetleges csomagok vannak. Ezek használatát sajnos még tovább nehezíti, hogy a rendszerhez tartozó online dokumentáció borzasztóan elavult és hiányos, pusztán az itt elérhető adatok alapján még az adatbázis-kezelő elindítása is problémás lehet.

2.2.3. JanusGraph

A megvizsgált gráfadatbázis-kezelők közül az egyik legérdekesebb a Linux Foundation keretei között fejlesztett, nyílt forráskódú JanusGraph, korábbi nevén Titan DB. Az eddig említett két rendszerhez hasonlóan ez is JVM futtatókörnyezetre épül, azonban a használatának fő nyelve nem a Java, hanem a Groovy [21] – bár természetesen előbbihez is léteznek API-k.

Saját leírása alapján a JanusGraph egy „elosztott, nyílt forrású, masszívan skálázódó gráfadatbázis” [40], aminek valamennyi aspektusát a nagyon gazdag integrációs lehetőségeinek köszönheti. A rendszer a klasszikusan az adatbázis-kezelő rendszerek felelősségének tekintett feladatok egy viszonylag nagy részét integrációs szolgáltatások formájában kidelegálja, így a ténylegesen implementálandó funkciók köre lényegileg a séma- illetve tranzakció menedzsmentet és a lekérdezések végrehajtását foglalja magába.

Nem definiál saját adatmodellt és lekérdező nyelvet sem, hanem „natívan integrálódik” [40] az Apache TinkerPop gráf stack-kel. Ahogy NoSQL népszerűbbé válása miatt egyre több különböző gráfadatbázis jelent meg – természetesen mindegyik saját fogalomrendszerrel és lekérdező nyelvvel, felmerült az igény egy olyan egységes és domináns gráf stack-re, mint amilyen az SQL a relációs világban, ennek a manapság legelterjedtebb megvalósítása a TinkerPop gráf stack. Ennek a két legfontosabb komponense a Gremlin és Gremlin Structure API (régebbi nevén Blueprints API).

A Gremlin Structure API gyakorlatilag a gráf modellre kialakított egységes adatmodellt és rendszer menedzsmentet definiálja, a relációs modellbeli megfelelője az ADO.NET és a JDBC. Ennek a központi elemei a tulajdonsággráf elemeinek megfelelő gráf elem típusok (például a Vertex és az Edge osztályok), és az adatbázis menedzsmenthez tartozó interfészek (például a Transaction és GraphManager interfészek). Fontos megjegyezni, hogy a tranzakcionális végrehajtás a Neo4j-hez hasonlóan támogatott, mind az eventually, mind az erős konzisztenciát biztosítani tudja.

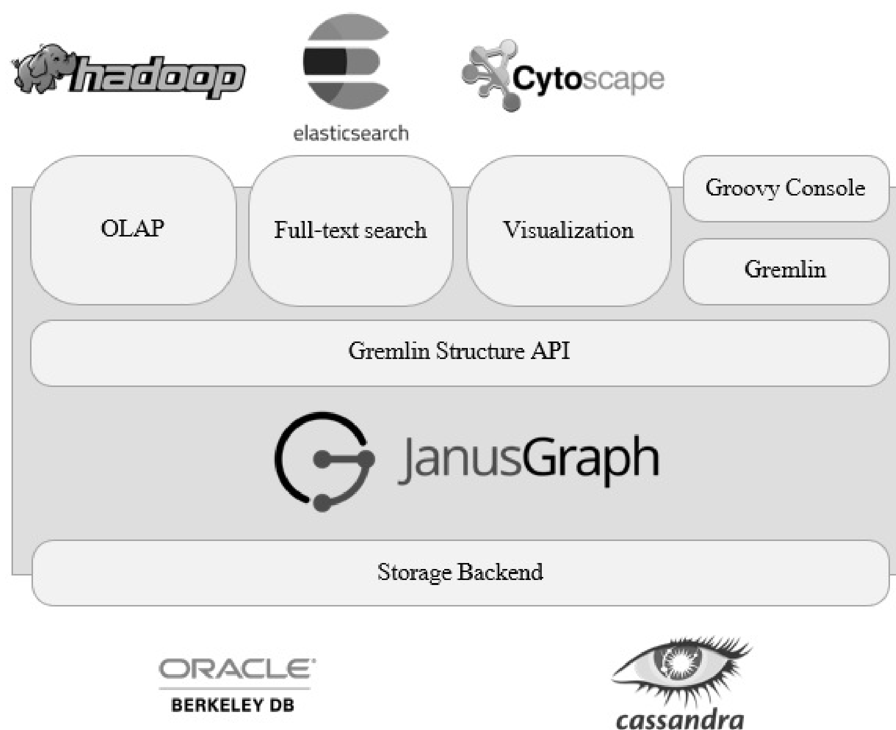
A Gremlin pedig egy erre az API-ra épülő nagyon gazdag, imperatív jellegű gráf lekérdező nyelv. Az ilyen típusú nyelvek egyik következménye, hogy a JanusGraph lekérdezés optimalizálója meglehetősen egyszerű, hiszen minimális mozgástere van csak – de ez a Gremlin legutóbbi verziójának deklaratív bővítései miatt a jövőben változhat. A korábban használt minta lekérdezés Gremlin nyelven a 2.3 kódrészlettel írható le:

```
g.V().has("name", "Bob").outE("knows").inV.outE("favoriteHobby").inV.values("name")
```

2.3. kódrészlet. Példaállítás Gremlin nyelven.

Az adatok fizikai tárolását sem a JanusGraph végzi, hanem kidelegálja a feladatot storage backend szolgáltatásoknak. Ilyen lehet például a Oracle BerkeleyDB, vagy az Apache Cassandra. Azáltal, hogy ez is funkcionalitás is kiszervezett, lehetőség van a fizikai tárolás módját a használati mód alapján kiválasztani és testreszabni – például a BerkeleyDB egy, a Cassandra több klaszteres környezetben igazán jó.

Ezeken túl több más integrációs lehetőség is adódik. A komplexebb, gráf analitikai jellegű lekérdezések (OLAP) kiszolgálását például Apache Hadoop és Giraph segítségével lehet megtenni, a full-text search funkciókat Elasticsearch-ön, vagy Apache Lucene-en keresztül is megtehető, illetve különböző adatvizualizációs technológiák – például Cytoscape – integrációja is támogatott. Ezek alapján a JanusGraph magas szintű architektúrája a 2.4 ábrán látható.



2.4. ábra. A JanusGraph magasszintű architektúrája.

A mérésben résztvevő rendszerek közül JanusGraph-on kívül az Azure Cosmos DB, az Orient DB és a Blazegraph is hivatalosan támogatja a TinkerPop gráf stack-et, illetve a Neo4j is rendelkezik nem hivatalos illesztővel.

2.2.4. Azure Cosmos DB

A vizsgált adatbázis-kezelők közül az egyetlen igazi felhő-alapú megoldás az Azure Cosmos DB. Ez azt jelenti, hogy a Microsoft 2017-es Build konferenciáján bemutatott rendszer csak a Microsoft Azure felhőszolgáltatás keretein belül érhető el (letölthető emulátor létezik hozzá, elsősorban tesztelési célokból), cserébe viszont a felhő adta valamennyi lehetőséget kínálja, mint például a globális elosztottságot, a hatékony, automatikus partícionálást és replikációt, valamint a lényegileg korlátlan skálázhatóságot. A rendszer népszerűségét

jól mutatja, hogy a fiatalsága ellenére is jelenleg a második helyen van a DB-Engines gráfadatbázisokkal kapcsolatos listáján.

Adatmodellt tekintve az egyik legsokoldalúbb adatbázis-kezelő, ugyanis multimodális jellegéből adódóan (amiről részletesebben az OrientDB-nél írok) a gráfon kívül támogatja többek között a relációs, a dokumentum, a kulcs-érték és az oszlopcsalád NoSQL megközelítésekbe sorolható API-kat is. A mérések szempontjából a gráf modell játszik kulcsszerepet, de érintőlegesen – a JanusGraph teljesítményének vizsgálatakor – az oszlopcsalád típusba sorolható Cassandra API-t is használtam. A Cosmos DB által megvalósított gráf adatmodell a TinkerPop gráf stack-et valósítja meg, tehát tulajdonsággráf megközelítést használ, és Gremlin lekérdezőnyelvvel lehet megszólítani. Azonban fontos különbség a korábbi rendszerekhez képest, hogy a fizikailag nem gráfokon, hanem úgynevezett atom-record-sequence-eken (ARS-eken) dolgozik [80], ami egy általános tárolási mód, ezáltal például a gráf adatok elérhetők dokumentum API-n keresztül is, és fordítva.

A felhőszolgáltatások első említett fontos tulajdonsága a globális elosztottság volt. Az Azure Cosmos DB világszerte jelenleg 54 régióban érhető el [66], amihez garantált 0,01 másodperces maximum késleltetés, és 99,999%-os rendelkezésre állás társul [4], ráadásul ez nagyon egyszerűen (grafikus felhasználói portálon), akár futás közben is megváltoztatható.

Az elosztottság általában magával vonja az adat elosztásának kérdéskörét is, emiatt a Cosmos DB-nek egyik legalapvetőbb része gráf partícionálás, ugyanis minden 10 GiB-nál nagyobb adathalmazt tárolása [5] esetén már az adatbázis létrehozásakor kötelező partícionáló kulcs, és a partíciók számának megadása. Ezt követően az adatok partícionált tárolásának megvalósítása transzparens, csupán a partícionáló kulcs érték megadása szükséges – melyet egyébként a keresési tér szűkítésével lekérdezés kiértékelések gyorsítására is lehet használni.

Az eddig ismertetett rendelkezésre állás és partícionálás mellett az elosztott adattárolással kapcsolatban a harmadik kulcs aspektus az adatok konzisztenciájának biztosítása. Az Azure Cosmos DB ezen a téren is konfigurációs lehetőségekben gazdag, hiszen a konkrét használati módtól függően 5 különböző konzisztencia szint is választható, az erős konzisztenciától egyészen az eventually consistent modellig. Ezek – jellegükből adódóan – minél gyengébbek, annál nagyobb elérhetőséget és átviteli teljesítményt, valamint annál kisebb késleltetést eredményeznek [14]. Fontos, hogy az adatok replikációja a háttérben, teljesen transzparensen zajlik.

Mindezek ellenére a rendszer igazi előnye az integrációs lehetőségekben és a skálázhatóságban van. A cloud-first megközelítésnek köszönhetően nagyon könnyen és gyakorlatilag korlátlanul integrálható az Azure más szolgáltatásaival, például az Azure Search full-text search szolgáltatással, vagy az Azure Data Lake Analytics ETL és adatelemző eszközzel.

A rendszer teljesítményét alapvetően befolyásoló tényezők – mint például a rendelkezésre álló feldolgozási sebesség és tárhely – a rendszer működése közben is bármikor, egy gomb megnyomásával az aktuális igények szerint növelhetők vagy csökkenthetők. Ezt az óriási rugalmasságot egyik másik vizsgált rendszer sem képes nyújtani.

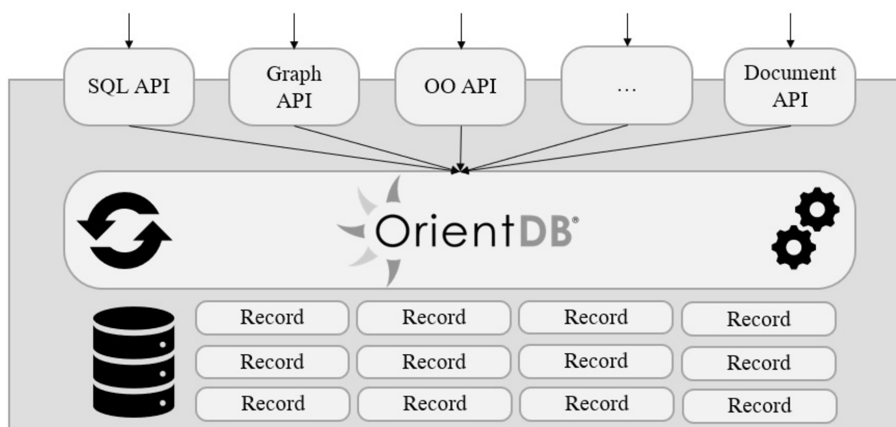
2.2.5. Orient DB

Az Orient DB szintét kitűntetett szerepű rendszer, ugyanis ez volt az első nyílt forrású, valódi multimodális adatbázis-kezelő. A vizsgált rendszerek legtöbbjéhez hasonlóan Java futtatókörnyezetre épül. Talán elsőségének is köszönhető, hogy a több különböző API egy rendszerbe integrálását – azaz a multimodalitást – az összes vizsgált konkurensnél mélyebben és átjárhatóbban teszi meg.

Ahogy az információs rendszerek egyre bonyolultabbá váltak, egyre több lényegesen különböző jellegű adaton kezdtek el dolgozni egy rendszeren belül. Ezek hatékony keze-

lését csak úgy lehetett megoldani, hogy a különböző jellegű adatokat különböző típusú rendszerek menedzselték, amit az úgynevezett polyglot [38] tárolás valósított meg.

Ennek a gondolatnak egy magasabb szintű megközelítésének tekinthetők a multimodális rendszerek: egy adathalmaznak az adott feladathoz leginkább illeszkedő modell szerinti kezelését teszik lehetővé, a tárolási struktúráról függetlenül. Ebből következik, hogy a multimodális rendszerek több eltérő adatmodell API-val rendelkeznek, általában a relációs, a dokumentum, a kulcs-érték pár, az oszlopcsalád és a gráf API-k kerülnek megvalósításra. Fontos, hogy ezek mindegyike ugyanannnak az adathalmaznak a kezelésére, egy alkalmazáson belül akár igény szerint keverve használható. Ennek a megközelítésnek az előnye, hogy az adatot kezelése az API választás szabadsága miatt minden esetben kifejező lesz, cserébe viszont a teljesítmény nem lesz optimális. Ezt a koncepciót mutatja be a 2.5 ábra.



2.5. ábra. A multimodális rendszerek magasszintű felépítése az Orient DB példáján.

A multimodalitásnak is több szintje lehetséges. Az Azure Cosmos DB egy viszonylag gyenge multimodalitást valósít meg, ugyanis habár a rendszer több modellt is támogat, az adatbázis létrehozásakor meg kell határozni a később használni kívánt API-t [32] – bár néhány speciális esetben van lehetőség a modellek közötti átjárásra [78]. Az Orient DB ezzel szemben valódi multimodális implementációval rendelkezik, azaz minden adat minden API-n keresztül elérhető – pontosabban, ahol ez értelmezhető [49], ráadásul ezek teljesen egyenrangúak.

Maga az Orient DB többek között relációs, gráf, dokumentum, objektum-orientált és geolokációs API-kat is támogat. A gráf API a TinkerPop gráf stack-re épül [91], így a Gremlin nyelv segítségével érhető el, és a tulajdonsággráf modellben dolgozik. Az egyes API-k közötti átjárhatóságot jól mutatja, hogy a különböző API-k koncepciói átszivárogtak egymás lekérdezési nyelveibe is, így például az eddig használt mintaállítás a Gremlin mellett gráf kiterjesztésekkel ellátott SQL API-n keresztül is elérhető, ahogy az a 2.4 kódrészlet mutatja.

```
SELECT hobby.name AS hobbyName
FROM (MATCH {class: Person, where: (name = 'Bob')}.out("Knows"){class: Person}.out("FavoriteHobby"){
    class: Hobby, as: hobby}
     RETURN hobby)
```

2.4. kódrészlet. Példaállítás az Orient DB gráfokkal kiterjesztett SQL nyelvén.

A legtöbb multimodális rendszerhez hasonlóan a tárolás alapegysége az Orient DB esetén is egy absztrakt elem, a rekordok koncepciója. Ezeket a működése során képes az adatbázis-kezelő on-the-fly, transzparens módon tömöríteni, ami ebben az esetben egy kompromisszumot jelent a tárhely és a sebesség között, hiszen minden olvasásnál kitö-

mörítés, írásnál pedig betömörítés történik [55]. A rekordok közötti kapcsolatok – vagy Orient DB terminológia szerint LINK-ek – a rekordokhoz a rendszer által automatikusan társított egyedi azonosítókat felhasználva, mutatóként kerülnek kialakításra.

Érdekesség az rendszer sémakezelésével kapcsolatban, hogy három különböző módot is definiál a rendszer az adatséma használatához: lehetőség van explicit séma definiálására, teljesen sémamentes használatra és ezek keverésére is [91]. Erre építve van ACID tranzakciós támogatás, azonban a két definiált izolációs szint (Read Committed és Repeatable Reads [55]) nem nyújt teljes védelmet tranzakciós egymásra hatások [79] ellen.

2.2.6. TigerGraph

Habár napjaink adatbázis trendje azt mutatja, hogy a multimodális adatbázisoké a jövő – elegendő az SQL Server-re [28], Azure Cosmos DB-re vagy az Orient DB-re gondolni, ezek pont a több modell támogatása miatt általában feláldozzák a natív adatmodell használatából adódó teljesítmény optimalizáció lehetőségeit – legalábbis a „másodlagos" adatmodellek esetén. Ezen iránnyal teljesen ellentétes megközelítésének tekinthető a TigerGraph, ami a Neo4j-hez hasonlóan egy natív gráfadatbázis, és ezt a kizárólagosságot extrém módon ki is használja.

A TigerGraph is a tulajdonsággráf modellt valósítja meg, fogalmi rendszere gyakorlatilag megegyezik a Gremlin Structure API típusaival – például Vertex, Edge, Property stb. Fontos különbség azonban TinkerPop gráf stack-hez képest, hogy erős sémadefinícióra van szükség, ugyanis amíg előbbi esetén elegendő a séma elemek definiálása – például, hogy milyen típusú csúcsok lesznek, milyen tulajdonságok lehetségesek, addig a TigerGraph esetén az ezek közötti kapcsolatot is meg kell adni – azaz, hogy például melyik csúcs típusnak pontosan milyen tulajdonságai lehetnek, az SQL-hez hasonlóan.

Ezt, a NoSQL világban viszonylag szigorú típusosságot, mind a végrehajtó, mind a fizikai tároló komponens erősen ki is használja. A fizikai tárolás a gráf elemek szintjén történik – azaz ténylegesen csúcsokat, éleket és tulajdonságokat tárol a rendszer, amivel a multimodális rendszerekhez képest az alábbi két okból is teljesítményelőnybe tud kerülni a rendszer:

1. A logikai és a fizikai modell azonossága miatt nincs szükség mapping-re a két réteg között, ami azt is eredményezi, hogy a lekérdezésekben megfogalmazott operátorokat nagyon egyszerű közvetlenül fizikai műveletekre lefordítani. Multimodális rendszerek esetén a konkrét API-n megfogalmazott lekérdezést először át kell fordítani a tárolási modell-en értelmezett lekérdezésre, és azt tovább a tényleges végrehajtandó tárolási műveletekre.

2. A tényleges tárolási struktúrát optimalizálni lehet a gráf műveletekre. Multimodális rendszerek esetén is lehetőség van egy kitűntetett modellre optimalizálni a tárolást, azonban ez a többi modell sebességét negatívan befolyásolhatja.

A TigerGraph tárolásának van egy további nagyon érdekes tulajdonsága, ami erősen kihasználja a konkrét gráf modellt. Fizikai szinten a tárolás tömörített kódolással történik, melynek hatékonysága a TigerGraph saját adatai szerint 2-10-szeres méret csökkenést okoz. A ki- és betömörítés a műveletek végzése során teljesen transzparens, az így elért nagyobb adatlokalitás – a cache hatékonyabb kihasználása miatt – gyorsítja a lekérdezések végrehajtását. Az igazi érdekesség ezzel kapcsolatban azonban az, hogy olyan tömörítési kódolást használnak, ami a gráf műveletekre nézve homomorf, azaz a lekérdezések kiértékeléséhez nincs szükség a kitömörítésre, csak az eredmények megjelenítéséhez [18], ezáltal elhanyagolható hatása van csak a futtatási időre.

A hatékony tárolásra egy natív, C++ nyelvű adatbázis-kezelő épül. A TigerGraph főként a memóriában dolgozik, a gráf lehető legnagyobb részét felolvassa a minél gyorsabb használathoz. Tovább gyorsítja a végrehajtást, hogy a lekérdezések végrehajtásához szükséges gráf operációkat erősen párhuzamos algoritmusokkal implementálták.

Lekérdezések megfogalmazására a saját, deklaratív jellegű GSQL nyelve használható, melyre a korábbiakban is használt állítást felhasználva a 2.5 kódrészlet mutat példát:

```
SELECT hobby.name
FROM Person: bob -(Knows: e1)- Person: somebody -(FavoriteHobby: e2)- Hobby: hobby
WHERE bob.name = 'Bob'
```

2.5. kódrészlet. Példaállítás a TigerGraph GSQL nyelvén.

A lekérdezések futtatására alapvetően két lehetőség adódik: előfordított lekérdezésként és interpretált lekérdezésként. Minden fordítás szintaktikai, szemantikai és típus ellenőrzéssel kezdődik – utóbbit az erősebb típusosság miatt tudja megtenni a rendszer, melyek sikeres lefutását követően a lekérdezés optimalizáló előállítja az optimális lekérdezési tervet. Ezt követően – a natív végrehajtást mintegy megkoronázva – a rendszer ezt az optimális tervet gépi kódra fordítja (DLL-t fordít belőle), amit követően a GSQL shell-ből egyszerű függvényhívásként lefuttatható, ami a natív alkalmazások sebességével fog végrehajtódni.

A rendszer legnagyobb gyengesége az integrációs lehetőségekben van, üzleti alkalmazással lényegileg csak REST API-n keresztül lehet integrálni, az interpretált módú végrehajtás miatt többszörösen lassabb lekérdezés kiértékelést eredményez.

2.3. Benchmarkok

Munkám alapötlete a [30] cikkből származik, amiben a szerzők a Wikidata adathalmazt használva, különböző reifikációs modelleket is kipróbálva összehasonlították 4 adatbázis-kezelő rendszer teljesítményét, melyek között volt relációs-, triple store és gráfadatbázis is. Az összehasonlítást két általuk definiált és kiválasztott lekérdezés család segítségével végezték. A cikk eredményei alapján a vizsgált lekérdezések esetén a Neo4j és a Blazegraph messze a leglassabb volt, az egyszerűbb lekérdezése esetén a relációs PostgreSQL, a bonyolultabbaknál viszont a triple store Virtuoso volt a leggyorsabb.

Ez alapján a [81]- ban kidolgoztam egy benchmark eszközkészletet, amivel több különböző gráfadatbázis többféle reifikációs modellen mért teljesítményét hasonlítottam össze a [30]-ben is használt atomic lookups lekérdezés család segítségével egy generált adathalmazon. Az itt kapott eredmények azt mutatták, hogy az Azure Cosmos DB még a Neo4j-hez képest is sokkal lassabb volt, továbbá a Titan bizonyos típusú lekérdezések esetén nyújtott nagyon jó teljesítményére is rávilágítottak. Az elkészített eszközkészletre építve, a [83]-ben ugyanilyen típusú méréseket végeztem a Wikidata adathalmazt felhasználva. Az ebben kapott eredmények többnyire megerősítették a [30]-ben és [81]-ban tapasztalt jelenségeket. Mindezek folytatásának, és jelen munkám közvetlen előzményének, a [82] tekinthető, melyben a felhő-alapú és az on-premise gráfadatbázis-kezelők teljesítményét hasonlítottam össze az Azure Cosmos DB és a Neo4j példáján keresztül, a Wikidata adathalmazt és a dolgozatban is használt valóélet-beli lekérdezéseket használva. Ezen mérések alapján arra jutottam, hogy bizonyos esetekben a cloud-first megoldások már felveszik a versenyt a hagyományos megközelítések teljesítményével, mindezek mellett pedig a lekérdezések végrehajtását lényegesen stabilabban – kisebb szórással – tudják elvégezni.

Tekintve, hogy a legtöbb rendszer esetén fontos szempont a végrehajtás teljesítménye, illetve gyorsasága, számos benchmarkot, illetve benchmark keretrendszert dolgoztak már ki a gráfadatbázisokat illetően is. Ezek eredményeivel kapcsolatban fontos megjegyezni, hogy a [22] és [48] benchmarkokkal kapcsolatos munkák megmutatták, hogy a mesterséges (generált) és a valós adathalmazokon mért eredmények lényegesen eltérők lehetnek.

A létező rengeteg gráfadatbázis benchmark közül kiemelkedik a Linked Data Benchmark Council [89] nevű független szolgáltató által készített, folyamatosan bővülő teszt készlet [39, 23, 42]. Erre építve a [58]-ban arra a következtetésre jutottak, hogy a rendszerek teljesítményét sokkal jobban befolyásolja a technológiai kiforrottság és optimalizáltság, mint a használt adatmodell, míg a [59] eredményei azt mutatták, hogy jelenleg különböző mérési munkafolyamatok (workload-ok) esetén más-más triple store adatbázis-kezelő tudja a legjobb teljesítményt nyújtani.

A benchmarking területének gazdagsága miatt tehát az összes született eredmény ismertetése meghaladná e dolgozat kereteit. Sok adatbázis-kezelő rendszer készít promóciós célból speciális méréseket [57, 27] , melyek saját rendszerük gyorsaságát emelik ki a konkurenciához képest. Fontos megjegyezni azonban, hogy ezek a benchmarkok rendszerint eltérő munkafolyamaton mérik a rendszerek teljesítményét – így bizonyos esetekben még akár ellentmondás is felléphet közöttük –, emiatt csak nehezen, vagy egyáltalán nem összehasonlíthatók az eredmények.

3. fejezet

Metaadatok modellezése

3.1. Problémafelvetés

Az internet globális elterjedésével együtt járt számos új, digitális szolgáltatás megjelenése is. Ahogy a technológiai fejlődés következtében ezen szolgáltatások egyre komplexebbé váltak, úgy kezdett el a működésük során robbanásszerűen növekvő online információ a strukturált irányból a kevésbé strukturált, illetve strukturálatlan irányba eltolódni. Ez az eltolódás együtt járt a metaadatok szerepének növekedésével, azaz az adatok kontextusba helyezésének – más néven reifikálásának – egyre fontosabbá válásával.

Manapság már szinte minden rendszer tárol metaadatokat a normál működéséhez. Legyen szó egy vállalati rendszerről, aminek a napi működése során csak az éppen aktuális érvényességi adatokkal ellátott adatokon szabad dolgozni, tudásbázisról, aminek meg kell tudnia mondani egy visszaadott állításról annak forrását, illetve lektoráltságát, vagy akár egy logikai alapú következtető rendszerről – például valamilyen szakértői rendszerről, ami az egyes következtetési lépésekhez azok modalitását vagy valószínűségét is figyelembe veszi, a metaadatok tárolása és aktív felhasználása mára az információs rendszerek nyújtotta szolgáltatások szerves részévé vált. Mindezen példák ellenére azonban ezen „extra" információk intenzív használata legtipikusabban mégis a strukturálatlan adatokra (például képekre, videókra, szabad szövegekre) épülő alkalmazásokra jellemző, gondolva itt például a Facebook hashtag-ekre, vagy a fotókhoz csatolt lokációs adatokra.

Maga a reifikáció eredetileg a tudásgráfok állításainak kontextusának – például érvényességének – reprezentációs problémáját jelentette, mára azonban általában véve a metaadat modellezés szinonimájaként is használják. A szemantikus tudáshálók általános jellemzője, hogy historikusan tárolják az adatokat, komplex adatmodellel rendelkeznek és a benne tárolt információk gyorsan bővülhetnek. Ezen szempontokat figyelembe véve adódik, hogy a metaadatok kezelését csak a logikai modell megfelelő kialakításával lehet hatékonyan elvégezni.

A korábban bemutatott két alapvető gráf koncepcióra reflektálva az adódik, hogy a tulajdonsággráf és az RDF megközelítés másként képes a reifikált, vagy „magasabb rendű" állításokat reprezentálni. Ezek fogalmi rendszerét használva a reifikáció általános modelljeit a 3.1 és 3.2 ábrák mutatják be, amik alapján látható, hogy egyik koncepció sem rendelkezik explicit reifikációt kifejező eszközkészlettel, hiszen:

- A tulajdonsággráf modell esetén a reifikációs (meta) adatrész úgy kapcsolódik a tényleges adatokhoz, mintha az ezek között futó élekből (3.1 bal ábra), vagy esetleg részgráfokból (3.1 jobb ábra) indulnának további élek a metaadatokhoz, ami viszont a gráfok definícióján kívül esik (csak két csúcs között futhat él).

- Az RDF modellre a reifikáció úgy képezhető le, mintha egy állítás (RDF hármas) önmagában egy másik állítás alanyává válna (3.2 ábra). Ez azonban az RDF azon megkötése miatt, hogy az állítások alanya csak erőforrás lehet, közvetlenül nem leképezhető.

3.1. ábra. A reifikáció általános sémái tulajdonsággráf modellnél.

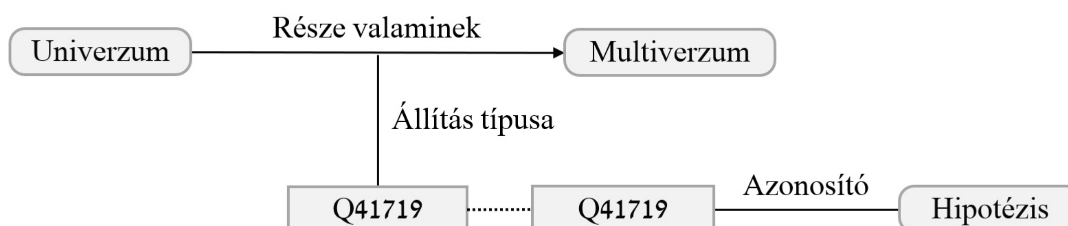
3.2. ábra. A reifikáció általános sémája RDF modellnél.

Habár egyik megközelítés sem képes direkt módon leképezni a reifikált állításokat, mégis létezik több modellezési megoldás is, ami vagy csak az egyik, vagy mindkét adatmodell esetén is használható. A megoldásokban közös, hogy az imént ismertetett, gráf modellből adódó problémákat vagy a reifikáció korlátozásával, vagy az absztrakciós szintek keverésével érik el. Az egyes bemutatott modelleket – a könnyebb összehasonlíthatóság érdekében – egy közös mintaállítást felhasználva mutatok be és jellemzek, mely állítás a későbbiekben is előforduló *„Az, hogy az univerzum a multiverzum része, még csak hipotézis."* mondat lesz.

3.2. Modellezési megoldások

3.2.1. Éltulajdonság modell

A korábbiaknak megfelelően a jelenleg legnépszerűbb gráfadatbázisok a tulajdonsággráfok koncepciójára épülnek, így elsőként a kizárólag ezen adatmodellen implementálható éltulajdonság reifikációs modellt vizsgáltam. Ennek a modellnek az alapkoncepcióját mutatja be a 3.3 ábra a mintaállítás segítségével:



3.3. ábra. A példaállítás éltulajdonság modellben reprezentálva.

A példa alapján is látható, hogy egy reifikált állítás leírása két részre bontható fel. Az egyik az alapállítás, mely a ténylegesen kapcsolatban részvevő két objektumot reprezentáló

csúcspont, és a köztük futó, a valós kapcsolat típusának megfelelő típusú élből áll. Ehhez az élhez éltulajdonságok formájában kapcsolódnak a minősítők oly módon, hogy a minősítők típusaiból (a meta-állítmányokból) lesznek az éltulajdonságok nevei vagy kulcsai, míg a minősítő adatok (metaadatok) azonosítói az ezen kulcsokhoz tartozó értékek.

A másik rész a metaadat leírása. Mivel valamennyi vizsgált rendszer csak primitív tulajdonság értékek tárolását teszi lehetővé, komplex objektumokét nem, illetve mert él csak csúcsból indulhat, tulajdonságból nem, így a metaadat objektumokat csak külön csúcspontokként lehet modellezni. A kapcsolatot a metaadat csúcs és az éltulajdonság érték között egy közös primitív tulajdonság (a példában egy szöveges azonosító) hozza létre.

Előnyök

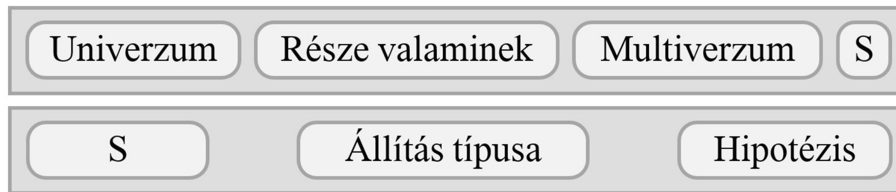
- Az alapállítás modellezése intuitív, az emberi gondolkodást jól követi. A tulajdonsággráf adatmodell keretei között talán ez a legkifejezőbb reifikációs modell, a 3.1 ábrán látható sémához is ez áll a legközelebb.
- Egyszerű reifikáció esetében – ha a metaadatok primitív típusúak csak, és nem vesznek részt más kapcsolatban – a két rész összevonható, és a metaadatok közvetlen éltulajdonság értékként tárolhatók. Ebben az esetben a modellezés még az előzőnél is kifejezőbb.
- Mivel általában amúgy is minden elemet egyedi azonosítóval látunk el, így csak a metaadat objektumoknak kell új azonosítót bevezetni, ami a többi modellhez képest viszonylag kis tárolási overhead-et jelent.

Hátrányok

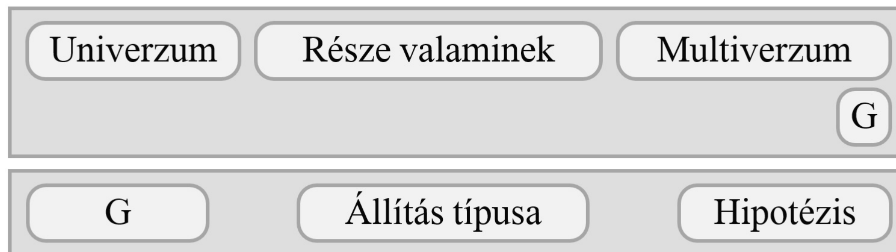
- Amennyiben több azonos típusú metaadat is tartozik egy állításhoz (például több forrás), akkor az állítás csak akkor leképezhető ebbe a modellbe, ha az adatbázis-kezelő rendszer a kollekció típusú tulajdonságokat is támogatja – primitív értékek listáját vagy halmazát.
- Az éltulajdonság értéke és a metaadat csúcspont között csak logikai kapcsolat van, így a konzisztencia biztosítása a felhasználó feladata.
- A lekérdezések a két rész miatt meglehetősen bonyolulttá válnak, főleg, ha kollekció éltulajdonságok is előfordulhatnak.
- A többszintű reifikáció nem megvalósítható. Mivel a metaadat és az állítás közötti kapcsolat tulajdonságra képződik le, így ahhoz már további tulajdonság nem társítható – a vizsgált rendszer verziók tulajdonságnak már nem lehet tovább tulajdonsága.
- Csak tulajdonsággráf modellben használható, hiszen az RDF modellben élek sincsenek.

3.2.2. Négyeseken (quad) alapuló modell

Az RDF korábban ismertetett szigorú hármasokra (triples) épített megközelítése nem ad lehetőséget a reifikáció igazán intuitív modellezési megoldására. Ennek egy lehetséges megoldási lehetőségeként adódik a hipergráf irányú általánosítás, azaz a négyesek tárolásának engedélyezése is. A mintaállítás négyes alapú reprezentációját a 3.4 és 3.5 ábrák mutatják be két eltérő megközelítésben:



3.4. ábra. A példaállítás rendezett négyesekként reprezentálva.



3.5. ábra. A példaállítás négyesekként reprezentálva, részgráf hivatkozásokat használva.

A 3.4 ábra az RDF alany-állítmány-tárgy hármasát egy általános célú negyedik elemmel egészíti ki (az „S” elemmel), ami az állítást reprezentáló egyedi erőforrás, majd a továbbiakban ezt az erőforrást – tehát magát az állítást – használja a klasszikus RDF struktúra szerint további hármasok (esetleg négyesek) leírására. Habár a negyedik elem ilyen általános célú használata hatékonyan megoldja a reifikáció problémáját, a webes szabványok nem ezt a definíciót használják, így ez jelenleg csak, mint elméleti lehetőség adódik, hiszen például lekérdező nyelv sincs definiálva hozzá.

A négyeseken alapuló tárolás webes szabványa [64] a negyedik paramétert az állítást tartalmazó gráf azonosítására használja, mint ahogy azt a 3.5 ábra is mutatja. A megközelítés alapja, hogy minden állítást egy egyedileg azonosított részgráfhoz lehet hozzáadni – például az alapállítást az ábrán a G részgráfhoz), és mivel az így definiált részgráfok is rendelkeznek azonosítóval, tetszőlegesen felhasználhatók további állításokban – például alanyként, ezáltal metaadatokat kapcsolva az egész részgráfhoz. Emiatt a 3.5 ábra a 3.1 ábra jobb oldali megközelítésének RDF leképezésének tekinthető négyesek felhasználásával.

Előnyök

- Viszonylag egyszerű, jól átlátható adatmodellt eredményez, melyben a kapcsolatok döntő része a domain modellből származik.
- Nagyon általános megközelítés, a reifikáció minden strukturális szintre leképezhető. Ez azt jelenti, hogy nem csak egyes elemek (állítások), hanem tetszőleges részgráfok is reifikálhatók.
- Támogatja a többszintű reifikációt is, azaz a meta-állításokra további meta-állításokat lehet leírni vele.

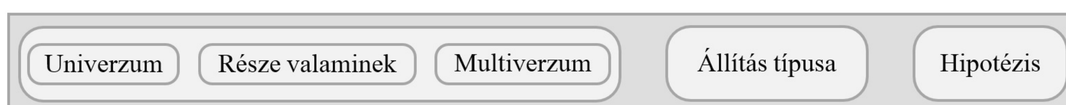
Hátrányok

- Az első megközelítést lényegileg egy jelenleg elterjedt gráfadatbázis-kezelő sem támogatja.

- Az RDF halmazok lekérdezéséhez használt SPARQL nyelv nem támogatja a négyesek használatát [33], a lekérdezéseket GRAPH clause-okkal kell bővíteni, ami lekérdezéseket lényegesen bonyolultabbá teszi, főként, ha több részgráfot érintőket.
- Habár több rendszer [88, 36] is támogatja a második koncepciót, kevésbé kiforrott technológiának számít, mint a hármas-alapú tárolás, emiatt rendszerint sem eszközkészletet, sem teljesítményt tekintve nem jobb annál – például a Blazegraph több száz oldalas dokumentációjából mindössze 2 sor és egy minta konfigurációs állomány tartozik a négyeseken alapuló tároláshoz [12].
- Csak RDF modellben használható, hiszen a tulajdonsággráf sem hipergráfokat (első megközelítés), sem olyan struktúrákat nem támogat, amit a második megközelítés használ (olyan gráf általánosítás, hogy tetszőleges részgráfok maguk is csúcsként jelen tudnak lenni gráfban).

3.2.3. RDR-alapú modellezés

Az előző modell a reifikációt az RDF állítás elemeinek számának bővítésével oldotta meg, azonban – több más irány mellett – ez a hármasokat megtartva is lehetséges, ha az egyes elemekre vonatkozó korlátok egy részét hagyjuk el. A Blazegraph a reifikáció natív támogatására egy saját, nem szabványos RDF bővítést vezetett be, az RDF*-ot, melyet a szintén nem szabványos, a rendszer saját SPARQL* nyelvével lehet lekérdezni. A RDF* és SPARQL* konfiguráció az RDR nevet kapta [67], és legfőbb újdonsága az RDF-hez képest, hogy állítás alanya legyet egy egész állítás maga is. Ezt a mintaállításon keresztül a 3.6 ábra mutatja be:



3.6. ábra. A példaállítás RDR modellben reprezentálva.

Ahogy a 3.6 ábrán is látható, az alapállítás egy RDF hármasra van leképezve, majd ez a hármas egészében megjelenik a meta-állítás alanyaként, de maga a meta-állítás is az RDF alany-állítmány-tárgy felépítést követi.

Előnyök

- Az emberi gondolkodáshoz talán legközelebb álló megközelítés. Az adatmodell egyszerű, könnyen érthető és kifejező.
- Nincs modellezésből származó tárolási overhead, hiszen az eltárolt összes elem a domain-ből származik.
- Bonyolultabb, metaadatokat is használó lekérdezéseket viszonylag egyszerű megfogalmazni SPARQL* segítségével.
- A metaadatok kezelés transzparens, azaz ezen adatok tárolása a domain adatokra semmilyen hatással nincs (sem strukturálisan, sem más szempontból), független tőlük.

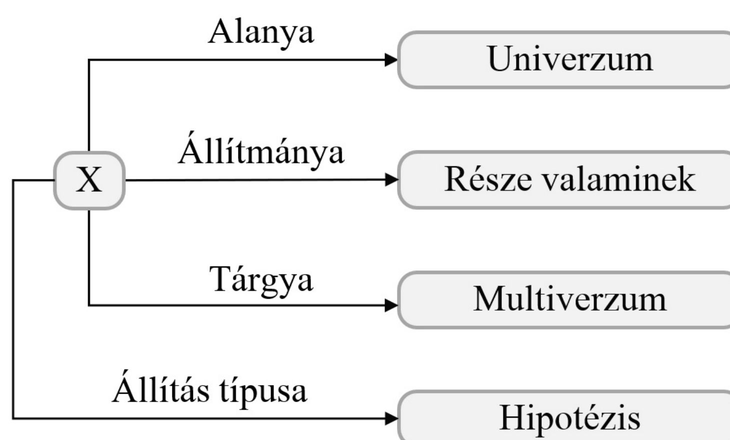
Hátrányok

- Nem szabványos, csak a Blazegraph rendszer esetén használható.

- Saját adatformátuma van, aminek következtében az ezzel való munkához mindenképp saját, egyedi komponenseket kell készíteni.
- Bár a lekérdezések egyszerűbbek lehetnek, a Blazegraph amúgy is elavult, hiányos dokumentációja, illetve az elérhető kevés példa alapján a bonyolultabb lekérdezések esetleges hibajavítása nehézkes lehet.
- Kis számú metaadat tárolására van optimalizálva (<5 metaadat állításonként) [67], ahogy a metaadatok száma növekszik, úgy romlik a teljesítménye a többi modellhez képest.
- Nem támogatja a több szintű reifikációt. A dokumentációban nem találtam erre vonatkozó korlátozást, azonban ennek az aspektusnak a tesztelése során több különböző esetet próbálva is csak hibaüzeneteket kaptam vissza.

3.2.4. Standard modell

Már a szemantikus web koncepció megszületésekor – ami életre hívta magát az RDF technológiát is – felkészültek a szabványok kidolgozásánál a metaadatok kezelésére, azaz a reifikációra. Az általuk kidolgozott reifikációs modellre a vonatkozó irodalom standard modellként hivatkozik, mert a magasabb rendű állítások leírására szabványos azonosítókat (URI-kat) használ [68], ahogy az a 3.7 és 3.8 ábrákon is látható a mintaállítás példáján keresztül, tulajdonsággráf és RDF reprezentációban is.



3.7. ábra. A példaállítás standard reifikációs modellt használva, tulajdonsággráfként reprezentálva.

A modell alapgondolata, hogy – a négyeseken alapuló első megközelítéshez hasonlóan – minden állításhoz bevezet egy dedikált csúcspontot (a példában az „X” csúcspont), ami magát az állítást jelképezi. Az eredeti koncepció szerint ennek az új csúcsnak a típusa (az `rdf:type` kapcsolat végpontja) az `rdf:Statement`, ezt azonban a méréseknél nem vezettem be, mert információs értéket nem hordozott. Az alapállítás részei ehhez a csúcsponthoz kapcsolódnak az `rdf:subject` (alanya), `rdf:predicate` (állítmánya) és `rdf:object` (tárgya) típusú élekkel. Fontos, hogy minden „`rdf`” névtérbe tartozó említett elem szabványosított jelentéssel rendelkezik. Az alapállításon túl a minősítők is az állítást reprezentáló csúcshoz kapcsolódnak, a kapcsolatok típusát a meta-állítmányok határozzák meg, míg az élek másik végén a metaadatok csúcspontjai vannak.

Előnyök

- Szabványosított elemek segítségével írja le a reifikált állításokat.

3.8. ábra. A példaállítás standard reifikációs modellt használva, RDF-ben reprezentálva.

- Az ábrák alapján látható, hogy mind RDF, mind tulajdonsággráf modellre leképezhető.
- Támogatja a többszintű reifikációt, hiszen az állítást jelképező csúcspont alanyként és tárgyként is részt tud venni további kapcsolatokban.
- Ugyan a négyeseken alapuló modellnél lényegesen bonyolultabb módon, de a reifikáció ebben a modellben is a struktúra minden szintjén megvalósítható, tehát megoldható részgráfok felvétele csúcspontként a gráfba – speciálisan a domain modell alapállításához tartozó részgráf az állítás csúcspontként jelenik meg, de ezek közötti ugyanilyen kapcsolatokkal magasabb struktúra szintek is reifikálhatók.

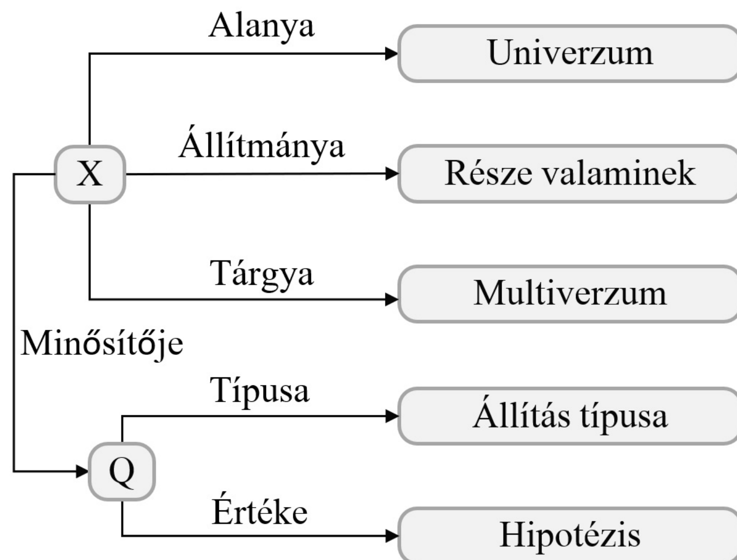
Hátrányok

- Az egyszerű adatmodell nagyon bonyolult struktúrát eredményez, mely nehezen áttekinthető, hiszen a minősítőket nem számítva összesen három (vagy négy) különböző, magas szintű kapcsolattípussal ír le mindent. A modell használatával a gráfadatbázisok egyik legnagyobb előnyét, a kifejezőerejét veszíti el a logikai reprezentáció.
- Nagy tárolási overhead-et visz a rendszerbe, hiszen egy állítás leírásához minimum három (a típussal együtt négy) állítás tárolása szükséges. Ez a gráf méreteit többszörösére növeli az éltulajdonság modellhez viszonyítva is, aminek a lekérdezésekre is észrevehető hatása lehet.
- A lekérdezések bonyolult gráfminták lesznek sok csillag-jellegű elágazással.
- A domain-modell elveszti a strukturáló szerepet, az adatmodellt szinte kizárólag a modellezési szempont határozza meg, „absztrakciós szivárgás” lép fel.
- Az OGM (object-graph mapping) keretrendszerek használatát lényegileg lehetetlenné teszi, hiszen a kapcsolatok végpontja gyakorlatilag tetszőleges típusúak lehetnek.

3.2.5. Általánosított standard modell

A [30], [83], [82] mérések eredményei azt mutatták meg, hogy bizonyos gráfadatbázis-kezelő rendszerek válaszideje lényegesen megnő, ha a lekérdezés az élekre vonatkozóan is tartalmaz ismeretlen információt – például, hogy két adott elem között milyen kapcsolat

van. A standard modellen megfogalmazott lekérdezések ezeket ki is küszöbölik, leszámítva a minősítőkhöz tartozó kapcsolatokat. A [30]-ban felvetett általánosított standard modell ezeket a meta-állításokat generalizálja ugyanazon logika mentén, mint ahogy azt a standard modell az alapállítás esetén teszi. Ezt mutatják a 3.9 és 3.10 ábrák.



3.9. ábra. A példaállítás általánosított standard reifikációs modellt használva, tulajdonsággráfként reprezentálva.

3.10. ábra. A példaállítás általánosított standard reifikációs modellt használva, RDF-ben reprezentálva.

Az ábra alapján jól látható, hogy az alapállítás kezelése azonos a standard modellnél bemutatottal, hiszen az volt a kiindulási állapot, a különbség a metaadatok kapcsolataiban van csak. Az általánosított standard modell bevezet minden állításhoz minden minősítő típushoz egy csúcspontot (a példában a „Q” csúcs), ami az `rdf:type` (típusa) típusú éllel

kapcsolódik a minősítő típusához, míg egyes metaadatok az `rdf:value` (értéke) szemantikájú kapcsolatokkal teszik meg ugyanezt.

Előnyök

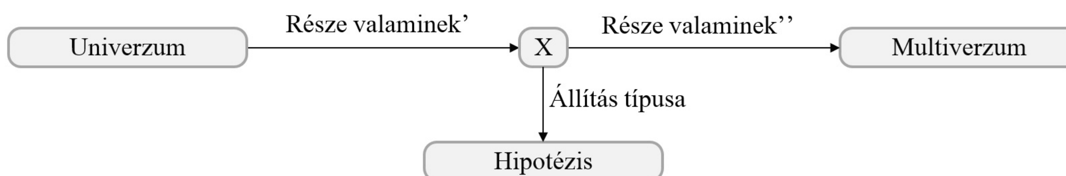
- Ugyanazok, mint a standard modell esetén, de az adatséma még szabványosabb, hiszen egyedül az állítás és a minősítő csúcspontok közötti kapcsolat típusa nem szerepel az RDF szabványban, de az is előre definiálható – például `rdf:qualfier` néven.

Hátrányok

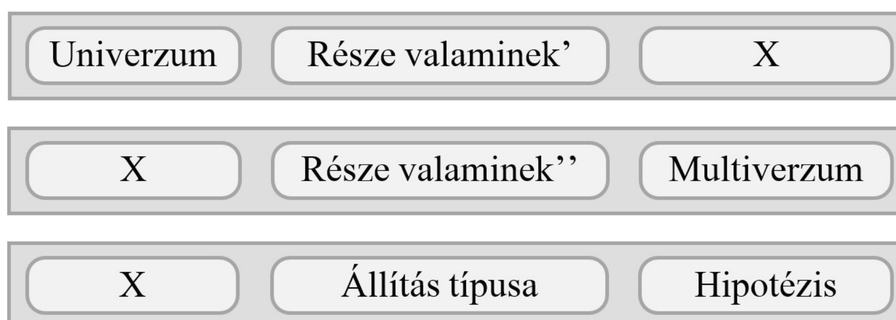
- Ugyanazok, mint a standard modell esetén, de az adatstruktúra, illetve a metaadatokra is vonatkozó lekérdezések még bonyolultabbá válnak, a domain modell már a metaadatok szintjén is háttérbe kerül, illetve a tárhely overhead még a standard modellnek is többszöröse lehet.

3.2.6. N-Ary modell

Ha tulajdonsággráf adatmodell eddig ismertetett reifikációs lehetőségeit megnézzük, akkor az látszik, két lényegileg eltérő koncepció alakult ki, melyek teljesen ellentétesek egymással. Az éltulajdonság modell egyszerű, kifejező, kis tárhely overhead-del rendelkezik, viszont a reifikációs képessége korlátozott, míg a standard modellek bonyolultak, az emberi gondolkodástól idegenek, nagy tárhely overhead-del járnak, cserébe a reifikációt a struktúra minden szintjén korlátok nélkül megvalósíthatóvá teszik. A két modell kombinálásával született meg az n-ary modell, mely a mintaállítás példáján keresztül a 3.11 és 3.12 ábrákon látható.



3.11. ábra. A példaállítás n-ary reifikációs modellt használva, tulajdonsággráfként reprezentálva.



3.12. ábra. A példaállítás n-ary reifikációs modellt használva, RDF-ben reprezentálva.

Az ábra alapján jól láthatók a koncepció azon elemei, amiket az éltulajdonság modelltől, és azok is, amiket a standard modelltől lettek átvéve. Legyen a kiindulás az állítás

éltulajdonság modellben vett reprezentációja (3.3 ábra). Az alapállítás állítmányát (a része valaminek típusú élt) két különböző élre kell bontani, melyek típusai kapcsolatban vannak egymással – például én a mérésekben azonos típusúra definiáltam őket. A szétbontott élek közé be kell ékelni egy új csúcspontot, ami – a standard modell filozófiáját tükrözve – magát az állítást fogja jelképezni, majd az éltulajdonságok eldobását követően a metaadat csúcspontokat a megfelelő meta-állítmány típussal ehhez a csúcshoz kell kötni. Az így kapott logikai modellt megvizsgálva azt kapjuk, hogy kombinálja a két „ősmodell" legfontosabb előnyeit, miközben a hátrányok legtöbbje kioltja egymást.

Előnyök

- Az adatmodell intuitív, viszonylag könnyen érthető – csupán két él távolságban kell minden kapcsolatot értelmezni, nagyjából a két „ősmodell" közé tehető, inkább a tulajdonsággráfhoz közelebb, hiszen a struktúrát szinte kizárólag a domain modell határozza meg.
- A standard modellekhez képest kisebb tárolási overhead-et eredményez. Ebből a szempontból is a két „ősmodell" közé tehető, de inkább a standard modellhez közelebb.
- A lekérdezések mindkét „ősmodellnél" egyszerűbben fogalmazhatók meg, hiszen sem csillag elágazások, sem tulajdonság-alapú `JOIN`-ok leírására nincs szükség legtöbbször.
- Implementálható mind RDF, mind tulajdonsággráf modell felett.
- A standard modellből örökli a többszintű, és a struktúra minden szintjén elérhető reifikáció képességét.
- Nincs szükség kollekció tulajdonságok támogatására az adatbázis-kezelő részéről.
- A metaadatok konzisztenciáját a rendszer biztosítja.
- Részben lehetővé teszi az OGM keretrendszerek használatát – a kettős, kapcsolódó élpárok kezelése okozhat problémát.

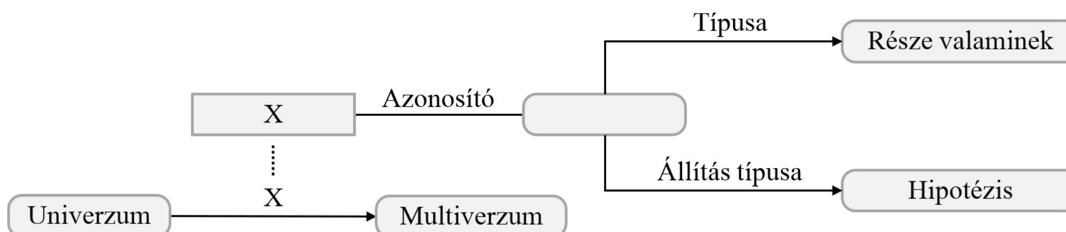
Hátrányok

- Az éltulajdonság modell kompaktabb, az emberi gondolkodáshoz közelebb áll.
- A kettős élek továbbra is bonyolítják a lekérdezéseket, a gráfbejárásokat pedig kifejezetten lassíthatják.

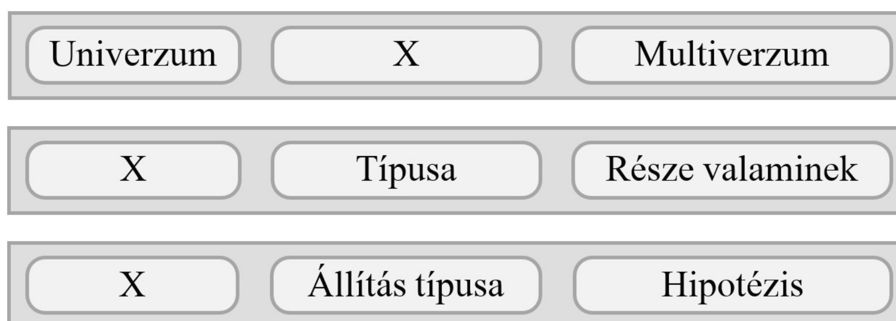
3.2.7. Singleton tulajdonság modell

A korábbiak alapján már látható, hogy az adatmodell specifikus és a standard megközelítések között olyan logikai modellek alkothatók meg, amik az „ősmodellek" előnyeit egyesítik, míg a hátrányaikat bizonyos szintig eltűntetik. Az arany középút valahol az n-ary modellnél van, az RDF világban azonban ennek alternatívájaként jelent meg a singleton tulajdonság modell. Tekintve, hogy a reifikáció során állításokról teszünk további állításokat, logikus gondolatnak tűnhet magát az állítmányt egyedileg azonosítani, majd ehhez kapcsolni a típus- és metaadatokat, ahogy az a ábrákon is látható.

A modell leírását és értékelését célszerű szétválasztani a tulajdonsággráf és az RDF modellekre, ugyanis jelentősen eltérően eredményt kapunk a két adatmodellen. Az RDF



3.13. ábra. A példaállítás singleton tulajdonság reifikációs modellt használva, tulajdonsággráfként reprezentálva.



3.14. ábra. A példaállítás singleton tulajdonság reifikációs modellt használva, RDF-ben reprezentálva.

implementáció kulcsa az eddig ki nem használt középső tag, az állítmány. Minden domain-beli állítás kap egy egyedi erőforrás azonosítót (a példában „X"), amit állítmányként használunk az alapállítás leírásához, emellett ehhez az erőforráshoz az eredeti állítmányt az `rdf:type` (típusa) típussal, a metaadatokat a hozzájuk tartozó meta-állítmány típussal kapcsoljuk.

A tulajdonsággráf implementáció ennél lényegesen bonyolultabb. Itt is minden domain-beli él egyedi típussal rendelkezik (az analógiát megtartva itt is „X" jelöli), azonban ezek mindegyikéhez tartozik egy csúcspont is, aminek ez az érték lesz az előre megadott tulajdonsága (a példában az azonosító tulajdonsága), ezzel logikailag összekötve az élt a csúcsponttal. Az RDF-fel azonos módon az él típusát és a metaadatokat ehhez új csúcsponthoz kötjük.

Előnyök

- RDF esetén ugyanannyi hármassal tárolja a magasabb rendű állításokat, mint az n-ary modell, azaz a standard modelleknél lényegesen kisebb tárolási overhead-et visz a rendszerbe.
- Viszonylag egyszerű lekérdezéseket eredményez, a meta- és éladatokat nem használó lekérdezések nagyon gyorsak lehetnek.
- A többszintű reifikációt talán a legintuitívebben lehet leírni vele RDF-ben.
- A standard modellekhez hasonlóan a tetszőleges struktúra szinten megvalósítható a reifikáció, bár ugyanannyira körülményesen.
- Mind RDF, mint tulajdonsággráf modellre leképezhető.

Hátrányok

- Nem jól olvasható, ugyanis nem lehet egyből megállapítani a kapcsolatok konkrét típusát, ahhoz meg kell nézni az élhez tartozó erőforrás (vagy csúcspont) kapcsolódó elemeit is.
- Az adatséma áttekinthetetlen lesz, OGM keretrendszerek használatát nem teszi lehetővé, hiszen minden él egyedi típussal rendelkezik.
- Tulajdonsággráf esetén az él és a hozzá tartozó csúcspont között csak logikai kapcsolat van, a konzisztenciát itt is a felhasználó felelőssége biztosítani.
- Az adatséma nagyon rosszul skálázódik. Amennyiben szükség van az éltípusok explicit definiálására egy adatbázis-kezelő esetén (több általam vizsgált rendszer is ilyen), úgy minden él beszúrása sémamódosítással jár.
- Az éltulajdonság modellhez hasonlóan itt is minden reifikált állítás két részgráfra esik szét, emiatt a lekérdezések nagyon bonyolultak lesznek, ugyanis olyan `JOIN`-okat kell leírni a gráfmintában, amik egy élt és csúcsot kapcsolnak össze az él típusa és a csúcs egy tulajdonságának értéke alapján.

3.2.8. További lehetőségek

Az eddig bemutatott megoldásokon túl – a modellezés természetéből adódóan – számos más logikai modell hozható létre a reifikáció megoldására, például a már ismertetett modellek kombinálásával, vagy az adatmodellek további általánosításával. A vizsgálat mindezek ellenére a gráfok körében közel teljesnek nevezhető, ugyanis az általam viszonylag széles körben megvizsgált, reifikáció témaköréhez kapcsolódó munkák és megoldások kizárólag ezeket a reprezentációkat használták.

4. fejezet

Mérési paraméterek

4.1. Adathalmaz

A mérések készítésének egyik fő motivációja az volt, hogy képet kapjunk az egyes adatbázis-kezelők teljesítményéről valós használati körülmények között. Ezen körülmények biztosításának első lépése egy valódi adathalmaz kiválasztása volt, ugyanis a mesterséges és a valós adatok mind a benchmarkokban, mind a valós terhelésekben lényegesen eltérően viselkedhetnek [22]. Az interneten számos ingyenesen is elérhető, nagyobb méretű tudásbázis is elérhető, azonban a korábbi munkáim és a meglehetősen könnyen érthető és átlátható adatmodellje miatt a Wikidata-ra [95] esett a választásom.

A Wikidata egy ingyenes, folyamatos fejlesztés alatt álló, bárki által elérhető és szerkeszthető tudásbázis, mely a Wikimedia-hoz tartozó weboldalak (pl. Wikipedia) forrásadatainak egy jelentős részét strukturált formában tartalmazza [95]. A legtöbb hasonló adatbázistól megkülönbözteti a Wikidata-t, hogy viszonylag egyszerű, jól dokumentált módon tárolja az információkat, valamint a tényleges adatokon túl – a reifikáció problémáját előhívva – az ezekhez tartozó metaadatokat is tartalmazza [95]. A szolgáltatás szellemiségének töretlen népszerűségét jól mutatja, hogy a felhasználók (és különböző bot-ok) mára már több, mint 61 millió elemről 770 milliónál is több állítást rögzítettek – legnagyobb részüket az elmúlt 2 évben, ráadásul átlagosan minden ötödik állítást metaadatokkal is ellátták [92].

A mérések során a Wikidata-t, annak is a 2016. januári JSON verzióját használtam kiindulásként, mely több, mint 67 millió reifikált állítást tartalmaz. A választás elsősorban a korábbi munkáim [83, 82] során kialakított infrastruktúrával való kompatibilitás miatt esett a viszonylag régebbi dump verzióra. A 4.1 kódrészleten egy reifikált állítás példa látható, ami lényegi tartalommal azt írja le, hogy *az univerzum a multiverzum része, és hogy ez az állítás egyelőre még csak hipotézis.*

A mintaállítás legegyszerűbben az RDF fogalomrendszerét használva érthető meg. A legfelső szintű `id` tulajdonság definiálja az állítás alanyát (Q1 - *univerzum*), a `claims` objektum kulcsai határozzák meg az állítmányok típusait (P361 – *része valaminek kapcsolat*), míg a `mainsnak` objektumok leírják a tárgyakat (értékkel rendelkező elem, ezen belül 3327819 azonosítójú, entitás típusú elem – *multiverzum*). A `qualifiers` és a példán nem látható `references` részek a metainformációkat tartalmazzák az állításról, a kulcs értékek a minősítők típusait (P31 – *típusa kapcsolat*) határozzák meg, míg az objektum értékek a minősítő értékeket (41719 azonosítójú, entitás típusú, értékkel rendelkező elem – *hipotézis*).

```
{
   "id":"Q1",
   "type":"item",
   ...
   "claims":{
      "P361":[
         {
            "id":"q1$21f31f42-4f4d-79b0-0380-92039776e884",
            "mainsnak":{
               "snaktype":"value",
               "property":"P361",
               "datatype":"wikibase-item",
               "datavalue":{
                  "value":{
                     "entity-type":"item",
                     "numeric-id":3327819
                  },
                  "type":"wikibase-entityid"
               }
            },
            "qualifiers":{
               "P31":[
                  {
                     "hash":"94962579945ddcb356b701e18b46a8ca04361fac",
                     "snaktype":"value",
                     "property":"P31",
                     "datatype":"wikibase-item",
                     "datavalue":{
                        "value":{
                           "entity-type":"item",
                           "numeric-id":41719
                        },
                        "type":"wikibase-entityid"
                     }
                  }
               ]
            },
            "qualifiers-order":[
               "P31"
            ],
            "type":"statement",
            "rank":"deprecated"
         }
      ],
      ...
   }
}
```

4.1. kódrészlet. Minta reifikált állítás részlet a Wikidata JSON dump-ból.

A példaállítás is jól mutatja, hogy az adathalmaz sok olyan információt tartalmaz, amik a felhasználói lekérdezések szempontjából nem relevánsak – például az adatmodell és belső adatmenedzsment információk –, vagy amik kezelésének feladata manapság tipikusan nem egy gráfadatbáziskezelő feladata. A minta entitásból már el lettek hagyva a `label`, `alias`, `description` és `sitelink` adatok, melyek hatékony lekérdezése az ún. full-text search adatbázisok területe, és a Wikidata rendszere maga is külső szolgáltatások segítségével használja őket a lekérdezésekben.

Látható továbbá, hogy a tárolási egység az entitás, azaz nem gráfként, hanem objektumokként – dokumentumokként, az elterjedt NoSQL terminológiát használva – jelennek meg a dump-ban az adatok. Az objektum szemléletű tárolásból következik, hogy a gráfalapú használathoz egy transzformáció definiálására volt szükség, ami a Wikidata struktúrát leképezi egy olyan közös, köztes szintű gráfra, ami tovább transzformálható konkrét logikai modell alapján olyan formára, amit az egyes vizsgált rendszerek képesek eltárolni. A transzformációt a 4.2 kódrészleten látható algoritmussal valósítottam meg:

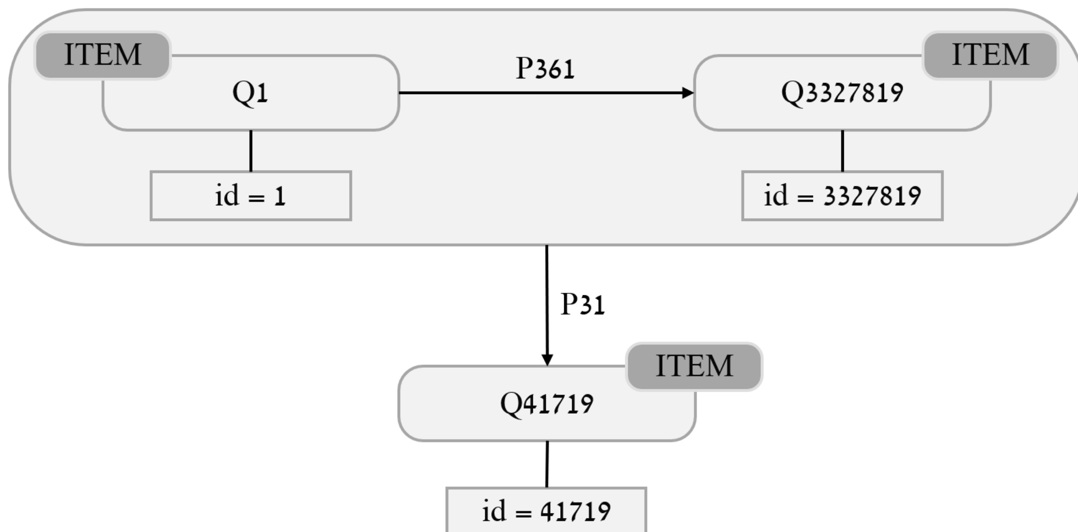
```
AddEntityToGraph(Entity entity, Graph graph)
  subjectVertex = new Vertex(entity.id);
  IF (NOT graph.Contains(subjectVertex)) THEN
    graph.AddVertex(subjectVertex);

  FOR EACH claim IN entity.claims
    objectVertex = new Vertex(claim.mainsnak);
    IF (NOT graph.Contains(objectVertex)) THEN
      graph.AddVertex(objectVertex);

    edge = new Edge(from: subjectVertex, to: objectVertex, type: claim.Key);
    FOR EACH qualifier IN (claim.qualifiers UNION claim.references)
      qualifierVertex = new Vertex(qualifier.Value)
      IF (NOT graph.Contains(qualifierVertex)) THEN
        graph.AddVertex(qualifierVertex);
      edge.AddQualifier(type: qualifier.Key, value: qualifierVertex);
    graph.AddEdge(edge);
```

4.2. kódrészlet. A Wikidata adathalmaz egy entitását gráf modellre leképező transzformáció magas szintű algoritmusa.

Az `id`, a `mainsnak`, a `qualifiers` és a `references` értékei alapján új csúcspontok kerülnek hozzáadásra a gráfhoz, amennyiben az még nem tartalmazza az adott entitást, vagy adatelemet. Az egyes adatelemek mezői csomópont tulajdonságokra képződnek le. A gráf csomópontok közötti kapcsolatokat a `claims`, az egyes kapcsolatokhoz tartozó minősítőket pedig a `qualifiers` és `references` kulcsai alapján alakítottam ki. Ezek alapján a 4.1 kódrészlet az alábbi köztes részstruktúrára képződött le (4.1 ábra):



4.1. ábra. A minta reifikált állítás köztes struktúrára leképezve.

Ezt a transzformációt felhasználva a letöltött Wikidata JSON dump adattartalma átalakítható a később bemutatott köztes reprezentációra, amire a tényleges mérési lépések építenek. Ezen felül azonban ez a transzformáció – kis kibővítéssel – felhasználható a skálázódás vizsgálatához szükséges további, eltérő méretű adathalmazok előállítására is. A módosítás előkészítéseként a Wikidata Query Service-t felhasználva manuálisan összegyűjtöttem azoknak a gráf elemeknek a halmazát, amik az 50 mérő lekérdezés eredményét bármilyen módon befolyásolják: vagy konkrétan szerepelnek az eredményben, vagy mint köztes elem vannak jelen, például egy út közepén. Ezt követően úgy módosítottam a 4.1 kódrészleten ismertetett transzformációt, hogy új gráf elem beszúrása előtt bevezettem két további ellenőrzést: ha az előre összegyűjtött, befolyásoló halmaz tartalmazza az új elemet, akkor biztosan beszúrásra kerül, különben pedig csak egy adott valószínűséggel.

Amennyiben egy csúcspont eltávolításra kerül, akkor a tulajdonságai és a rá illeszkedő élek sem kerülnek a gráfba, míg ugyanez az élek esetén a hozzá tartozó minősítők törlését eredményezi. Az így módosított transzformációt három különböző valószínűség paraméterrel is lefuttattam, így három különböző méretű adathalmaz keletkezett: a *Wikidata 100* az 1 (teljes), a *Wikidata 60* a 0,6, míg a *Wikidata 30* a 0,3 valószínűséghez tartozó adathalmazok.

4.2. Lekérdezések

Mint minden benchmark, az általam elvégzett mérések is a vizsgált rendszerek teljesítményét csak a lefuttatott lekérdezések által reprezentált terheléstípusra (workload-ra) írják le jól. Ebből kifolyólag a megfelelő mérő lekérdezéshalmaz kiválasztása kritikus feladat, hiszen alapjaiban határozza meg a mérési eredmények (fel)használhatóságát.

Ugyan jelenleg is számos gráfokat, illetve gráfadatbázisokat megcélzó benchmark készlet létezik, az általam végrehajtott teljesítménymérés azonban ezek legtöbbjétől lényegesen eltér a használt lekérdezések jellege miatt. A kapcsolódó munkák (lsd. 2.3 fejezet) legnagyobb része domain-független adathalmazokon, „mesterségesnek” tekinthető lekérdezéseket futtat le, általában olyan gráfspecifikus mintákat tartalmazva, mint a BFS/DFS alapú, illetve legrövidebb út keresések, k-szomszédságok meghatározása, vagy a page rank kiszámítása. Ezekkel szemben az én méréseim a Wikidata-át használják adatként, a lekérdezési mintákat pedig az ehhez tartozó több évnyi, valódi felhasználók által beküldött, és gyakran előforduló kérdések alapján határoztam meg, emiatt a kapott eredmények jobban leírják a rendszerek viselkedését valós felhasználási környezetben.

A mérési lekérdezések meghatározásának első lépése a kiinduló, „nyers” Wikidata Query Service-en [94] elérhető lekérdezések begyűjtése volt a későbbi offline feldolgozáshoz. Tekintve, hogy több, mint 400 gyakran felmerült kérdés típust tartalmaznak a példák [93], ezek letöltését és egységesítését egy saját készítésű segédprogrammal végeztem. A fennálló erőforrás és technológiai korlátok miatt az összes lekérdezés futtatására nem volt lehetőségem, így ki kellett választanom egy lehetőleg minél szélesebb spektrumú, de korlátozott elemszámú részhalmazt a mérésekhez. A szűkítést két alapgondolat mentént végeztem el:

1. Először eltávolítottam azokat a lekérdezéseket, amik olyan SPARQL feature-t használnak, ami nem érhető el minden vizsgált implementációnál. Ezen lépés során elsősorban olyan elemek kerültek eltávolításra, amik komplexebb külső szolgáltatásokat, például geolokációs szolgáltatásokat használtak. Ezt a lépést szintén egy általam készített segédprogram végezte el egyszerű szöveges mintakeresés alapján.

2. Ezt követően a megmaradt lekérdezéseket csoportosítottam „hasonlóság” alapján, majd minden csoportból kiválasztottam véletlenszerűen 1 elemet, ami a csoportot reprezentálja a mérésekben. Ezt azért tehettem meg, mert az egy csoportba került elemek „hasonló” gráf mintákat írnak le. A lekérdezések „hasonlóságának” eldöntése azonban az előzőnél lényegesen komplexebb feladat volt, a letöltött elemek vizsgálatához több különböző módszert is kidolgoztam, illetve kipróbáltam.

Először formális nyelvi szintű elemzést próbáltam végezni rajtuk az ANTLR nyelvi elemző keretrendszer segítségével [2]. Az eszközt be is üzemeltem, azonban csak a legegyszerűbb letöltött lekérdezéseket tudtam elemzésnek alávetni, mert az interneten nem találtam teljeskörű SPARQL nyelvi szintaxis definíciót, például az elérhető nyelvtanok egyike sem támogatta a beágyazott lekérdezéseket. Tekintve, hogy egy megfelelő nyelvtan elkészítése a dolgozat témájához csak marginálisan kapcsolódott volna, így ezt az módszert végül elvetettem.

Második módszerként egy egyszerűbb, szöveges mintákra épülő statisztika módszert dolgoztam ki. A módszer alapelve az volt, hogy a nyelvi, és a példákban található konvencionális elnevezéseket kihasználva egyszerű szöveges mintaillesztésekkel és keresésekkel kategorizáljam a hasonló lekérdezéseket. Habár az elemzést elvégeztem, a módszer egyszerűségéből adódóan a statisztikai adatok alapján csak olyan alacsony felbontású csoportosítást tudtam végezni, ami a további felhasználásra nem volt alkalmas.

Végül a lekérdezések csoportosítására egy BorderFlow nevű külső eszközt használtam, melyet egy másik hasonló benchmarkhoz fejlesztettek ki [17]. Az eszköz a tartalmazott SPARQL nyelvi feature-ök és a leírt gráfminta alapján képes SPARQL lekérdezések klaszterezésére. Miután megadtam a letöltött lekérdezéseket, az alkalmazás 17 klaszterbe be is sorolta őket.

Ahhoz, hogy a korábbi munkáimban [81, 83, 82] kidolgozott mérési metodikát – minden változót tartalmazó lekérdezés több különböző behelyettesítéssel is le legyen futtatva – alkalmazni tudjam, csak olyan elemet választhattam az egyes klaszterek reprezentálására, amik fix elemeit változónak tekintve az adathalmaz tartalmaz legalább 5 lényegileg különböző – legalább a csúcspontokban eltérő – illeszkedést. Ezt a további szűrést elvégezve 10 lekérdezés maradt, azaz volt 7 olyan klaszter, aminek minden lekérdezésének minden fix SPARQL elemét változóra cserélve is 5-nél kisebb számosságú volt az eredményhalmaza [82]:

1. A `PTN1` minta megkeresi az adott konkrét tulajdonsággal rendelkező elemeket, és az ezekhez valamely adott módon kapcsolódó másik elemeket. Egy lehetséges példa kérdés, ami erre a mintára illeszkedik: *Melyek azok a zeneszámok, amik 1984-ben jelentek meg, és kik ezeknek a dalszerzői?*

2. A `PTN2` minta megkeresi egy adott elemhez közvetlenül kapcsolódó adott tulajdonságú elemeit. Példa: *Melyek azok az általam kedvencnek jelölt zeneszámok, amik a rock műfajba tartoznak és 2008-ban jelentek meg?*

3. A `PTN3` minta megszámolja, hogy hány adott tulajdonságú elem van. Példa: *Hány olyan kedvencnek jelölt zeneszám van, ami 3 percnél rövidebb?*

4. A `PTN4` minta meghatározza azokat az elemeket vissza irányú élszemantika mentén, amik egy adott elemtől maximum 2 távolságra vannak. Példa: *Kik azok az előadók, akik hatással voltak az ACDC együttes Highway To Hell című dalának szerzőire?*

5. A `PTN5` minta megszámolja, hogy adott összetett csoportosításban melyik csoport mennyi elemet tartalmaz. Példa: *Évenkénti és műfajonkénti bontásban mennyi dal jelent meg az elmúlt 50 évben?*

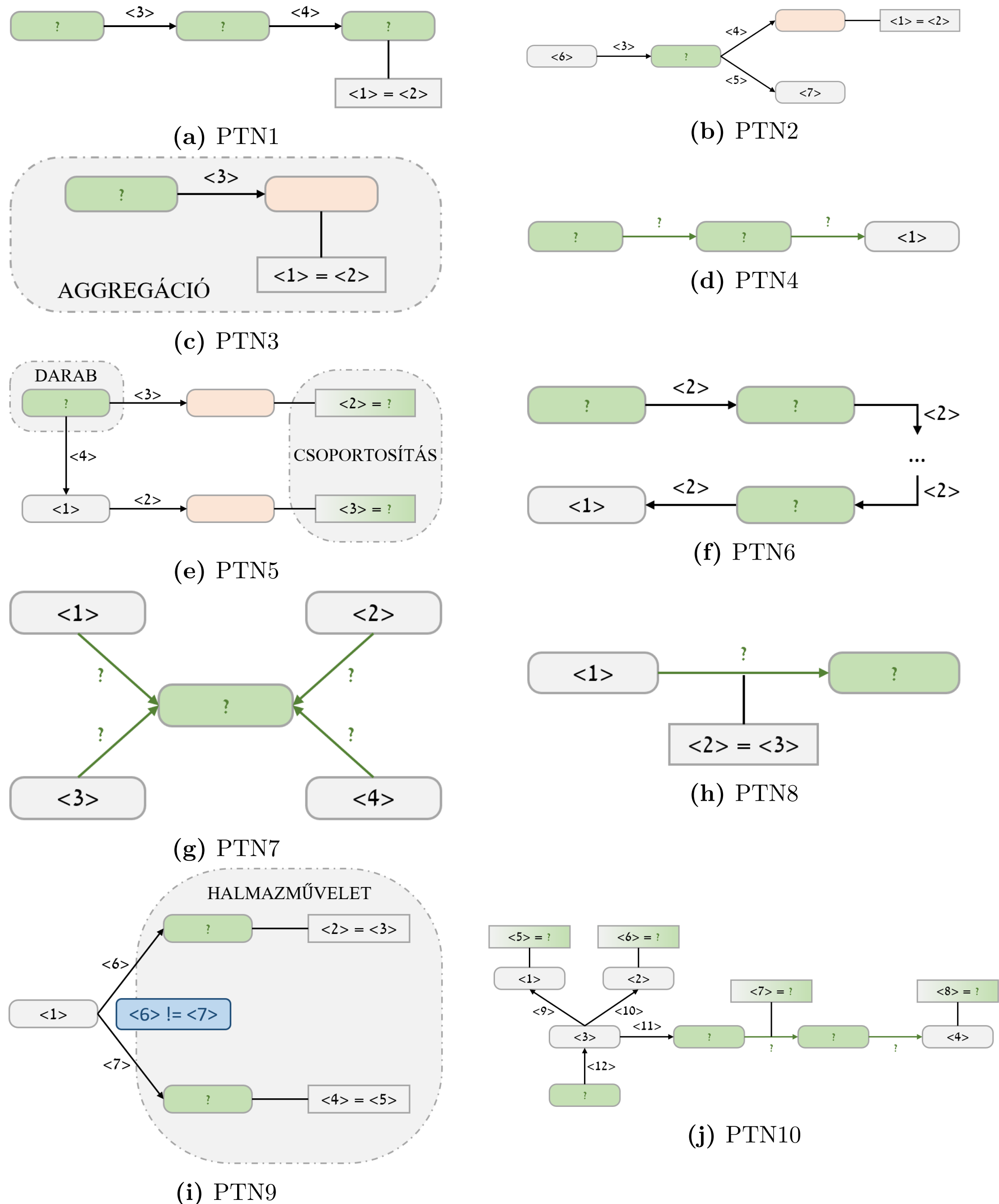
6. A `PTN6` egy adott elem vissza irányú, adott szemantika menti tranzitív lezártját határozza meg. Példa: *Kik azok az előadók, akik hatással voltak a Queen-re, vagy olyan előadóra, aki hatással volt a Queen-re, és így tovább?*

7. A `PTN7` minta megkeresi a közös pontokat 4 különböző elem között, és hogy ezek milyen értelemben kapcsolódnak össze. Példa: *Mi a közös a Queen-ben, Hadrianus császár falában, Wimbledon-ban és a Harry Potter-ben, és ez a közös elem milyen értelemben kapcsolódik az előbbi 4 dologhoz?*

8. A `PTN8` minta egy metainformációval ellátott egyszerű kérdést válaszol meg. Példa: *Mi az összes elérhető, 1983-ra vonatkozó, lektorált információ az Iron Maiden együttesről?*

9. A `PTN9` egy logikai VAGY típusú kérdést ír le. Példa: *Kik azok, akik vagy énekesei, vagy dobosai voltak az AC/DC együttesnek?*

10. A `PTN10` egy általános, sok ismeretlent (ismeretlen adatot, elemet és kapcsolatot is) tartalmazó komplex kérdés.



4.2. ábra. A mérések során használt lekérdezési minták.

A csupa változót tartalmazó lekérdezések (generalizált lekérdezések) meghatározásával együtt ki tudtam választani a mérésekben hozzájuk tartozó 5-5 véletlenszerűen választott változó behelyettesítést is, hiszen a generalizált lekérdezések lefuttatásának eredményhalmaza a számosság mellett az összes lehetséges behelyettesítést is tartalmazta. Ezt a kiválasztást manuálisan végeztem el.

A 4.2 ábrán láthatók a mérés során futtatott lekérdezés minták tulajdonsággráf modellben reprezentálva. A lekerekített sarkú téglalapok csúcspontokat jelölnek, a többi pedig vagy csúcshoz, vagy élhez tartozó tulajdonságot (metaadatot). Az ábrán zöld szín és kérdőjel jelöli az adott lekérdezésben azokat az ismeretleneket, amik a lekérdezés végeredményében szerepelnek, míg a narancssárga szín az ismeretlen, de a kimenetben meg nem

jelenő csúcsokat jelenti. A szürke szín, és a relációs jelek közé írt számok reprezentálják az illesztendő minta azon elemeit, amik a konkrét behelyettesítésből származnak. A PTN9-nél megjelenik még továbbá a kék szín is, ami további megszorítást tartalmaz a mintára vonatkozóan.

A mérésekben ténylegesen használható lekérdezések előállításának utolsó lépése az előző lépések eredményeként előálló 50 lekérdezés átalakítása a különböző reifikációs modelleknek megfelelően az összes szükséges lekérdező nyelvre. Az összesen 1250 darab lekérdezés transzformációt és az így előálló lekérdezések helyességének ellenőrzését szintén manuálisan végeztem el.

4.3. Mérési környezet

Ahhoz, hogy egy végrehajtott benchmarkban a különböző rendszerek és lekérdezések eredményei értelmesen összehasonlíthatók legyenek, alapvető követelmény, hogy ezek mérési körülményei a lehetőségek szerint minél azonosabbak legyenek. Ennek hiányában az egyes mérések különböző környezeti hatások zavarait fogják tartalmazni, ami az eredmények torzításához, végsősoron pedig a valóságnak nem megfelelő teljesítmény viszonyokhoz és következtetésekhez vezet. A mérési folyamat jellegéből adódóan emiatt alapvetően az alábbi aspektusok azonosságát kell biztosítani:

- Futtatási környezet
- Mérési adathalmaz tartalma
- Mérő lekérdezések
- Mérő szoftver rendszer és folyamat torzításai

A négy aspektus közül talán az adathalmaz azonosságának biztosítása a legegyszerűbb feladat. A kiinduló adathalmaz a Wikidata 2016. januári JSON dump-ja volt. Az adathalmaz kezelésének első lépése ennek átalakítása a később bemutatott mérő szoftverrendszer komponensei által elvárt köztes formátumra (IF-re). Ez a köztes réteg még a [81]-ban került bevezetésre a rendszerbe, elsősorban a konkrét adatforrás könnyű lecserélhetősége miatt, azonban a valós adathalmazok kezelése szempontjából is van egy előnyös következménye, abból kiindulva, hogy a valóélet-beli nagyméretű adatsokaságok rendszerint tartalmaznak akár szintaktikai, akár szemantikai hibákat. Tekintve, hogy a legtöbb rendszer adatbetöltő komponense más és más bemeneti formátumot vár el az importáláshoz, ezekhez külön konverziós programokat kell készíteni, melyekben az említett hibák kezelését azonos módon kellene implementálni, és ezek azonosságát külön-külön tesztelni is kellene. Mivel az IF réteg bevezetésével a kiinduló adathalmaz csak egy konverzióban vesz részt – ami a forrást IF-re transzformálja, így a hibakezelő logikát elég egy helyen tesztelni, a közös konvertáló lépés garantálja az azonos bemenetet a további transzformációkhoz.

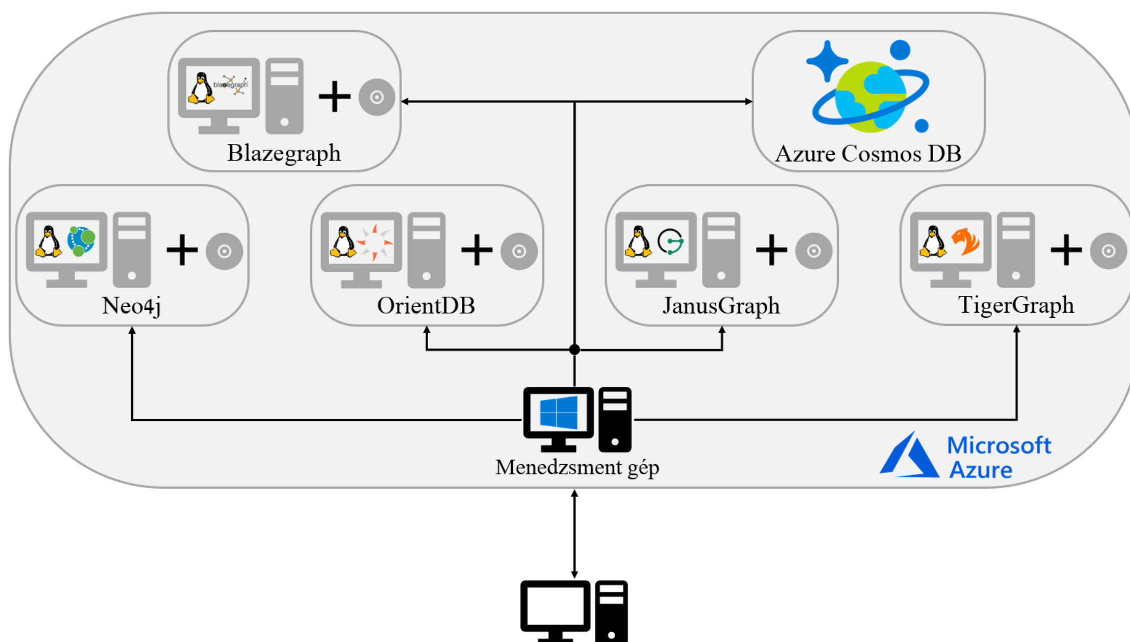
Az így előálló köztes formátumú adathalmaz vagy további transzformáción esett át, vagy közvetlenül betöltésre került a konkrét adatbázis-kezelőbe annak betöltési eszköztámogatásától függően. Ezek a komponensek külön-külön is tesztelve lettek, illetve a betöltést követően 20 előre definiált, egyszerű, de élekre és csúcsokra is kiterjedő aggregációs lekérdezés eredményét is összehasonlítottam a betöltött adathalmaz azonosságának biztosítása érdekében.

Ahogy az a vizsgált adatbázis-kezelők bemutatásánál is észrevehető volt, a legtöbb rendszer saját – vagy legalábbis eltérő – lekérdező nyelvvel rendelkezik, így a mérő lekérdezéseket nem csak különböző reifikációs modellekre, hanem különböző nyelvekre is át kell fordítani. Mivel ezek egyike sem volt automatizálható – bizonyos lekérdezőnyelvek között ugyan van transzformációs program [45], de ezek eredménye se nem teljesen megbízható, se nem optimális –, így a manuális fordítások helyességét is szükséges volt tesztelni. Ehhez minden modell-nyelv kombináció 1-1 lekérdezését egy külön erre a célra definiált kisméretű teszt gráfon ellenőriztem, amivel a futtatott lekérdezések azonosságát is biztosítottam.

Az eddigiek alapján látható, hogy a négy aspektusból kettő esetén sikerült biztosítani az azonosságot, azonban a fennmaradó két körülmény egyezőségét a rendszerek sajátosságai miatt gyakorlatilag lehetetlen biztosítani. Ennek legegyszerűbb példája, hogy amíg az Azure Cosmos DB csak felhős erőforrásként érhető el, azaz a futtatókörnyezetről semmi információnk nincs, addig például a JanusGraph virtuális gépen fut, aminek nem csak minden fontos hardver paraméterét, de a pontos szoftveres környezetét – például a Java heap mérete – is ismerjük, átállíthatjuk.

Ugyan az azonos körülmények biztosítása az előző okokból kifolyólag nem volt megvalósítható, igyekeztem a lehető legkisebb különbséget elérni a mérési környezetek között. A tényleges mérések teljes egészében a Microsoft Azure publikus felhőjében lettek elvégezve. A 4.3 ábrán látható mérési topológián is látható, hogy a Neo4j, a Blazegraph, a JanusGraph, az Orient DB és a TigerGraph is külön virtuális gépben futott, melyek azonos, Standard E4s v3 [47] specifikációval rendelkeztek, kiegészítve egy adattároló lemezzel. Az egyes VM-ek pontos specifikációja az alábbi volt:

- Intel® Xeon® E5-2673 v4 processzor, 4 virtuális CPU maggal, 2,3-3,5 GHz magonkénti órajel frekvenciával
- 32 GiB RAM memória
- 30 GiB prémium SSD tárhely az operációs rendszer és az adatbázis-kezelők számára
- 64 GiB SSD tárhely ideiglenes adattárolásra – ez nem volt használva a mérésekben
- Ubuntu 18.04 LTS Server operációs rendszer
- 256 GiB extra, prémium SSD tárhely az betöltés előkészítéséhez, és a betöltött adatok tárolására



4.3. ábra. A mérési infrastruktúra topológiája.

Maga a felhő-alapú környezet is behoz némi bizonytalanságot a mérésekbe a megosztott erőforrások miatt, azonban ez egyrészt elhanyagolható nagyságú, másrészt ez valós használat között is jelentkezne, így – némileg furcsa módon, pont emiatt – jobb képet ad a rendszerek teljesítményéről valóélet-beli esetekben. Ezen rendszerek esetén a mérő és a mért rendszer egy gépre volt telepítve.

Ezekkel szemben – az adatbázis-kezelő jellegéből adódóan – az Azure Cosmos DB felhős erőforrásként vettem igénybe. Ebből adódóan a mérő rendszer a menedzsment gépen volt, onnan lettek beküldve a lekérdezések és az utasítások, ami miatt ezek az eredmények a hálózati késleltetést is tartalmazzák. Ez VM-ekhez hasonlóan két okból nem jelent

problémát: egy valós alkalmazás is így tud csak az adatokhoz hozzáférni, emiatt a torzítás valójában pontosabbá teszi a mérést, továbbá a biztosított SLA következtében ez összes garantáltan 100ms alatt marad, ami nem befolyásolja érdemileg az eredményeket. Mindezek mellett valamennyi rendszer mérése az alábbi folyamat alapján lett elvégezve:

1. KÖZÖS LÉPÉS: Wikidata JSON adathalmaz átalakítása köztes reprezentációra. Ezt összesen egyszer tettem meg mindhárom adathalmaz esetén a fejlesztői és menedzsment gépen.
2. A mérések elvégzéséhez szükséges szoftveres környezet kialakítása a virtuális gépeken. Ezen lépés keretében telepítésre kerültek a konkrét adatbázis-kezelő rendszerek, azok minden függőségeivel együtt – például a Java futtatókörnyezet megfelelő verziójával. A mérő keretrendszer komponensei .NET Core-ban – kivéve a JanusGraph komponenseket, amik Java-ban – lettek implementálva, így a .NET Core SDK 2.2 verziója is mindenhova telepítésre került. Ez minden rendszer esetén egyszer történt meg.
3. OPCIONÁLIS: Az adatbázis-kezelő rendszer felkonfigurálása, amennyiben van rá lehetőség.
4. A köztes formátumú adathalmaz letöltése az erre a célra kialakított közös BLOB tárolóból, majd ennek kicsomagolása.
5. OPCIONÁLIS: A köztes formátumú adatok átkonvertálására olyan reprezentációra, amit az adott rendszer importáló eszköze támogat.
6. Az adathalmaz betöltése az adatbázisba, majd az esetlegesen keletkezett ideiglenes állományok törlése.
7. Az adatbázis-kezelő elindítása, a lekérdezések futtatása és mérése, a kapott eredmények mentése egy előre definiált formátumban.
8. Az eredmény fájlok felmásolása a menedzsment gépre, az erre a célra készített komponens segítségével az eredmények betöltése egy SQLite adatbázisba az egyszerűbb adatelemzés céljából.

Adatbázis-kezelő konfigurációk

Ahogy arra a [58] is rámutatott, az egyes adatbázis-kezelők teljesítménye szempontjából kulcsfontosságú tényező lehet annak kiforrottsága, ami miatt ezek a rendszerek tipikusan verzióról verzióra gyorsabbá válnak. Ugyanezen okokból nem elhanyagolható szempont a konfigurációs beállítás sem – annak ellenére sem, hogy a gráfadatbázisok a relációs világ érettebb rendszereihez képes a teljesítménnyel kapcsolatosan nagyon kevés beállítási lehetőséggel rendelkeznek, így fontosnak tartom ezen körülmények rendszerenkénti pontos leírását is. Minden itt nem említett beállítás esetén az adott rendszer alapértelmezéseit használtam.

Neo4j

A Neo4j-t az ingyenes Community Edition 3.4.16-os verziójával mértem meg, a JRE 11.04-es verziójú futtatókörnyezete mellett. A korábbi munkáim miatt a Neo4j méréseket több adatbázis és Java környezet verzióval is elvégeztem , melyeknek a teljesítményre gyakorolt hatását a válaszidővel kapcsolatos mérési eredmények részben ismertetem részletesen.

A konfigurációs lehetőségek egyike a Java heap méretének beállítása. Az Neo4j újabb verziói tartalmaznak egy optimalizációs eszközt, ami többek között képes a futtató rendszer hardveres erőforrásai alapján megállapítani az ideális heap méretet. Ennek a javaslata alapján a mérésekhez meg is növeltem 24 GiB-ra az alapértelmezett méret korlátot.

A másik általános lehetőség a teljesítmény javítására az indexek definiálása. A Neo4j lehetőséget ad B*-fa alapú [74] indexek definiálására. A mérések során kizárólag az alapértelmezetten bekapcsolt ID tulajdonság alapú indexelés volt használva.

A Neo4j egy rendszer specifikus teljesítmény optimalizációs lehetősége az úgynevezett „planner hint”- ek (lekérdezés tervezési tippek) nyelvi támogatása [60]. Ez gyakorlatilag azt jelenti, hogy a lekérdezések megfogalmazásakor lehetőségünk van USING záradékban leírást adni a lekérdezés optimalizáló számára, hogy milyen indexeket, illetve kiértékelési stratégiát használjon. A módszert csak egyedileg, a konkrét adathalmaz statisztikái alapján lehet optimalizációra használni, ellenkező esetben lassítja a lekérdezést. Néhány mérési lekérdezés esetén kipróbáltam mind a helyes, mind a helytelen USING záradékok hatását a lekérdezések teljesítményére, ezeket szintét a végrehajtási időkkel kapcsolatos mérési eredményéknél ismertetem.

Blazegraph

A Blazegraph rendszer esetén a 2.1.5-ös verziót használtam a mérésekhez. A szükséges futtatókörnyezetként a viszonylag régi 8.0-ás verziójú Open JDK-t telepítettem, mert – több más emberhez hasonlóan [53] – problémába ütköztem minden más verziójú Java használata esetén, ugyanis NullPointerException hibával már az adatbázis elindítása is sikertelen volt.

A Blazegraph meglehetősen sok beállítási lehetőséggel rendelkezik, azonban ezek döntő része nem dokumentált, a forráskód és a wiki oldalakon elérhető minta konfigurációs állományok [13] adnak csak némi iránymutatást – ezek használhatóságát azonban jól jellemzi, hogy egyik letöltött konfiguráció sem használható egy az egyben.

A konfigurációs állomány segítségével testreszabható többek között a fájl zárolási mód, a tárolási struktúra, a klaszter kezelés, a használandó automatikus következtetési logika, a tárolási és naplózási méretek, indexelés stb. A kiindulást az RDF Only és a Fast Load konfigurációk együttes használata jelentette, ami a gyors futtatásra van optimalizálva, így a lehető legtöbb „okos” funkció, mint a következtetés, kikapcsolásra került.

A Blazegraph esetén olyan modelleket is használtam, amik az RDF* adatmodell képességeit teljesen kihasználják, így a már bemutatott konfigurációból két további változatot is csináltam, az egyik a négyes-alapú tárolást, a másik az RDR mód használatát kapcsolta be.

Ezen rendszer esetén is van lehetőség a Java heap méretének beállítására, ezt indításkor parancssori paraméterként kell átadni. A dokumentáció szerint érdemes kis méreten tartani a Blazegraph számára elérhető heap méretet (4 GiB maximum), hogy a szabad memóriát az operációs rendszer lapozó cache-ként tudja használni [7]. A mérések egy részét elvégeztem több heap méret mellett is, ennek hatását a lekérdezések teljesítményére a rendszer válaszidő analízisénél ismertetem.

JanusGraph

Abból kifolyólag, hogy a JanusGraph valamennyi funkcióját integráció révén valósítja meg, és a lekérdező nyelve is imperatív, így meglehetősen kevés lehetőség van magának a rendszernek a konfigurálására – ez leginkább kimerül a megfelelő integrációs szolgáltatások kiválasztásában.

Tekintve, hogy a mérés célja az egyes rendszerek OLTP jellegű teljesítményének mérése volt, sem vizualizációs, sem analitikai, sem keresési szolgáltatás nem lett a rendszerhez kapcsolva. Tárolási szolgáltatásként a mérésekhez az Oracle BerkeleyDB-t használtam. Azért erre esett a választásom, mert ez a JanusGraph-fal egy JVM-ben fut, és így a milliárd nagyságrendű csúcsszám alatt ez nyújtja a legjobb teljesítményt [62]. Kísérleti szándékkal megpróbáltam az Azure Cosmos DB-t is a Cassandra API-ján keresztül beállítani storage backend-nek, azonban ezt nem sikerült működésre bírnom.

A benchmarkhoz a JanusGraph 0.4.0-ás verzióját használtam, válaszidejeit egyedül a JRE 11.04-es verziója felett mértem meg. Abból kiindulva, hogy a Neo4j automatikus indexelést használ az ID tulajdonságon, a betöltés során beállításra került az azonosító alapú indexelés a JanusGraph esetén is.

Azure CosmosDB

Tekintve, hogy az Azure Cosmos DB szolgáltatásként érhető el, így az eddigiekhez hasonló alacsony szintű finomhangolásra nincs lehetőség. A teljesítmény befolyásolására alapvetően négy lehetőség adódik: az átviteli teljesítmény beállítása (minél nagyobb, annál gyorsabb a rendszer, cserébe annál drágább), a megfelelő konzisztencia szint kiválasztása, a munkafolyamathoz leginkább illő partíciók számának meghatározása, és az ehhez illő partícionáló kulcs megadása.

A mérések során az Azure Cosmos DB ingyenes próbaverzióját használtam, ami teljes funkcionalitást és teljesítményt biztosít, csak idő-, illetve erőforrás használati korlát van definiálva. Az adatok betöltése alatt 1000 RU/s átviteli korlát volt beállítva, míg a lekérdezések 10000 RU/s-os korlát mellett futottak, így egy logikai modell mérése pont belefért az ingyenes verzió korlátozásaiba.

A partíciók számának meghatározásakor azt kell figyelembe venni, hogy egy partíció nem tartalmazhat 10 GiB-nál több adatot. Az tárhelyre vonatkozó mérési adatoknál részletezett okokból a mérésekhez 20 partíció lett kialakítva. A partícionáló kulcsként az egyes csúcspontok azonosító száma modulo 20 lett definiálva, ami következtében a kapott eredmények felső korlátnak tekinthetők, a konkrét munkafolyamatra optimalizált partícionálással ennél jobb eredmények is elérhetők.

Tekintve, hogy csak lekérdezéseket tartalmazott a mérő lekérdezés halmaz, melynek elemei egymást követően lettek lefuttatva, így a mérésekhez a leggyengébb konzisztencia szintet állítottam be. A JanusGraph-nál ismertetett okok miatt a Cosmos DB is úgy lett konfigurálva, hogy a betöltést követően az azonosító tulajdonság értéke alapján indexelje fel a csúcsokat.

OrientDB

Az Orient DB Community Edition3.0.21-es verzióját használtam a mérések során, futtató platformként pedig az Open JDK 1.8.0.222 verziója lett feltelepítve. Konfiguráció szempontjából az Orient DB esetén két fontos beállítási lehetőséget találtam: a tárolási mechanizmus módját, és az indexelést.

A rendszer lehetőséget biztosít arra, hogy négy lehetőségből ki, milyen módon szeretnénk tárolni az adatainkat. Ugyan van lehetőség memória adatbázis létrehozására, de ehhez nem kapcsolható perzisztens adattárolás, így csak addig élnek a betöltött adatok, ameddig a JVM is fut [46]. Emiatt a mérések során a plocal lehetőséget használtam, ami ugyan lemezre perzisztálja az adatokat, de a használat alapján, egy intelligens lapozási algoritmus segítségével a memóriában cache-eli őket [61], így nagy rendelkezésre álló memória esetén minimális lemezművelet végzésére van szükség.

Az Orient DB egyedi megközelítéssel rendelkezik az indexekkel kapcsolatban, ugyanis a használatukat explicit jelezni kell a lekérdezésekben [35], emiatt sokkal inkább az relációs világból ismert materializált nézetéhez hasonlítható. Mivel ez is egy workload specifikus, és nem általános optimalizációs eszköz, így nem lettek indexek definiálva a lekérdezésekhez. A többi ilyen specifikus elemhez hasonlóan ennek a teljesítményre vett vonzatát is a végrehajtási idő fejezetben ismertetem.

TigerGraph

A mérésekben részt vett még a TigerGraph Developer Edition 2.4-es verziójú változata, melyhez nem találtam érdemi konfigurációs lehetőséget, illetve nem is volt szükség ilyesmire. A teljesítményt leginkább befolyásoló tényezők közül mindössze az előfordított és az interpretált mód között kellett választanom. Tekintve, hogy az előfordított mód minden lekérdezés esetén általánosan alkalmazható, így a méréseket elsődlegesen így végeztem el. Összehasonlítás céljából lemértem néhány lekérdezés sebességét interpretált módban is, ennek a lekérdezésekre gyakorolt hatásáról a végrehajtási sebesség fejezetben írok bővebben.

Tekintve, hogy natív kódú végrehajtás történik (C++), így nem lehetséges heap méret megadása, illetve a memóriaorientált, erősen párhuzamos, statisztikák alapján kioptimalizált végrehajtási terv miatt még az indexelés definiálására sincs igazán szükség.

5. fejezet

Mérési eredmények

Habár a korábbi fejezetekben 6 gráfadatbázis implementációt, és 7 reifikációs modellezési lehetőséget mutattam be részletesen, a tényleges méréseket csak 29 különböző, ezekből képezhető rendezett páron végeztem el a lehetséges 42 helyett. A fennmaradó 13 páros azért hiányzik, mert az érintett adatbázis-kezelőn technológiai okokból nem implementálható az adott logikai modell, vagy a rendelkezésre álló erőforrás korlátokon belül nem sikerült megvalósítani. A mérésekben résztvevő párosokat (konfigurációkat) az 5.1 ábra mutatja be.

	Neo4j	Blazegraph	Azure Cosmos DB	JanusGraph	OrientDB	TigerGraph
Éltulajdonság	✓	✗	✗	✓	✓	✓
Standard	✓	✓	✓	✓	✓	✓
N-Ary	✓	✓	✓	✓	✓	✓
Általánosított standard	✓	✓	✓	✓	✓	✓
Singleton tulajdonság	✓	✓	✓	✓	✓	✗
Quad	✗	✓	✗	✗	✗	✗
RDR	✗	✓	✗	✗	✗	✗

5.1. ábra. A mérésekben résztvevő implementáció-modell párosok.

A hiányzó 13 párosból 10 a négyes-alapú és az RDR modellekhez tartozik, ugyanis ezt a két modellt csak a Blazegraph rendszer esetén tudtam megvizsgálni, ami a két megoldás direkt hipergráf-alapú megközelítésére vezethető vissza. A többi rendszer a klasszikus gráf koncepciót használja adatmodellje alapjaként, a Blazegraph azonban tartalmaz – nem szabványos – hipergráf kiegészítéseket, ami miatt közvetlenül is implementálni tudja az erre épülő modelleket. A további hiányzó esetek egyedi okok miatt maradtak ki. Az éltulajdonság modell sem az Azure CosmosDB, sem a Blazegraph esetén nem került megmérésre, azonban két teljesen különböző problémából kifolyólag. A Blazegraph esetén – lévén, hogy RDF alapú rendszer – nem értelmezett a tulajdonság, így az éltulajdonság fogalma, tehát a probléma alapvetően adatmodell korlátokból fakad. Az Azure CosmosDB estén a van

lehetőség éltulajdonságok definiálására, azonban ezek jelenleg még nem támogatják a kollekciókat (azaz éltulajdonság értéke csak egy skalár lehet), amire az adathalmaz tartalma alapján viszont szükség lenne. Ebben az esetben a probléma egy implementációs korlátozásra vezethető vissza, ugyanis a csúcsokhoz rendelt tulajdonságok lehetnek kollekciók.

Az utolsó meg nem vizsgált páros a singleton tulajdonság modell használata a TigerGraph rendszerrel, mely alapvetően két ok egyidejű fennállásara vezethető vissza. Az egyik a singleton tulajdonság modell jellegéből fakad – miszerint minden él saját egyedileg azonosított típussal rendelkezik –, ami miatt minden él beszúrása sémamódosítást vált ki a gráfból. A másik ok, hogy a TigerGraph esetén két lehetőséget találtam az adatbázis séma módosítására: az egyik az adatok betöltése előtt DDL utasításokkal statikus séma kialakítása, vagy webes felületen manuális sémamódosítás. Előbbi a kisebb adathalmazok generálásánál alkalmazott nem-determinizmus miatt nem használható, utóbbi pedig azt eredményezné, hogy minden él beszúrása előtt manuálisan kellene sémát módosítani, ami egy akkora adathalmaz tömeges betöltése esetén praktikusan kivitelezhetetlen. A páros tehát azért került kihagyásra a mérésekből, mert nem sikerült erre a problémára hatékony megoldást találnom.

Fontosnak tartom megemlíteni továbbá, hogy az eredmények értelmezésekor a betöltési idő, tárhely idény és a teljesítmény alfejezetekben bemutatott értékek és azok analízise a *Wikidata 100* adathalmazon végzett mérések eredményein alapulnak, az egyes rendszerek különböző aspektusú teljesítményeinek mérettől függő változásait a skálázódásra vonatkozó részben ismertetem.

5.1. Adatbetöltés

Ugyan egy adatbázis-kezelő működési teljesítményére nézve kevésbé lényeges az adatbetöltési folyamat hatékonysága, de az egyes rendszerek használata, illetve beüzemelése kapcsán kikerülhetetlen a kezdeti, ősfeltöltési adatok bevitele, így érdemes megvizsgálni a kiválasztott gráfadatbázisokat ebből a szempontból is. A tömeges betöltés kapcsán alapvetően négy kérdés merülhet fel, így az egyes vizsgált rendszereket is ezek mentén fogom jellemezni:

1. *Szükséges-e explicit módon adatsémát definiálni?* A NoSQL rendszerek egyik széles körben elterjedt jellemzője, hogy sémamentesek, azonban nem minden adatbázis-kezelőre igaz ez. Ezen okból fontos lehet, hogy szükség van-e a séma explicit definiálására, és ha igen, akkor ezt az adatbetöltés előtt meg kell-e tenni, vagy lehetőség van a betöltés során on-the-fly is módosítani a sémán.

2. *Rendelkezik-e a rendszer tömeges betöltő eszközzel?* Ha egy alkalmazás feltételezi a forrás adatok konzisztenciáját, és ismeri a rendszer belső tárolási mechanizmusát, az adatbetöltést akár nagyságrendekkel is gyorsabban el tudja végezni az egyenként beszúráshoz képest. Emellett természetesen azért sem elhanyagolható, mert nemleges válasz esetén ezt a funkcionalitást a felhasználónak külön implementálnia kell.

3. *Mennyi időbe kerül betölteni az adatokat a rendszerbe?* A rendszer beüzemelése szempontjából fontos lehet, hogy az adatbáziskezelő indítása, és az adatokkal feltöltött futásra kész állapotba lépése között mennyi idő telik el.

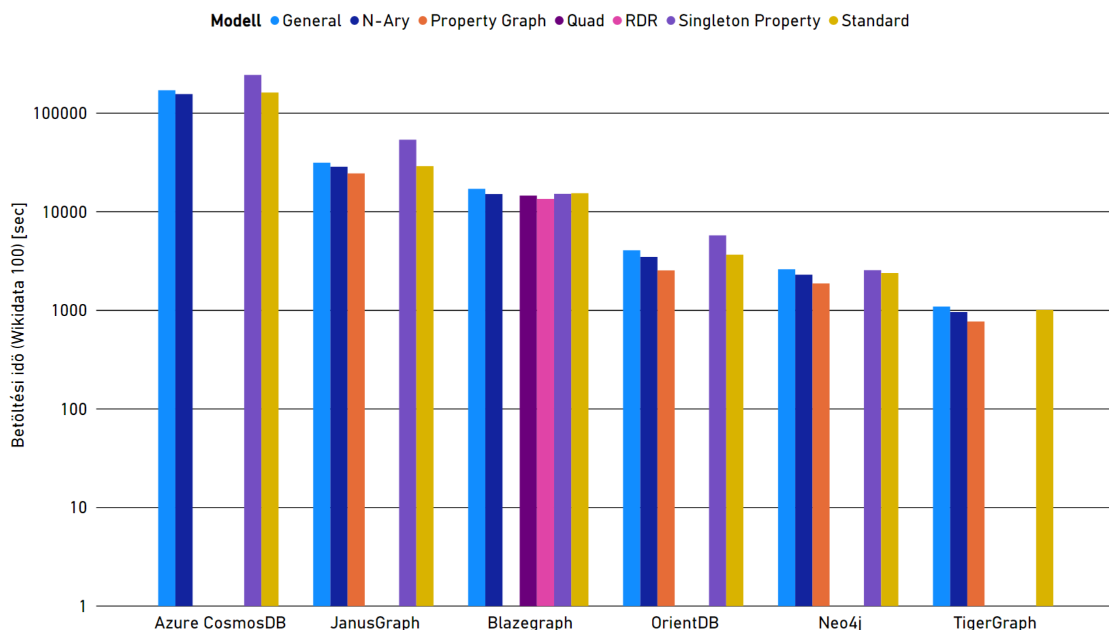
4. *Hogyan skálázódik a betöltési idő?* Az előző kérdéshez kapcsolódva az sem elhanyagolható, hogy a betöltési idő az adatmennyiség növekedésével hogyan változik. Az ezen szempont szerint eredményeimet a skálázódással kapcsolatos 5.5 fejezetben ismertetem.

Az 5.2 ábra logaritmikus skálán mutatja az egyes rendszerek betöltési idejeit különböző reifikációs modelleken a *Wikidata 100* adathalmaz esetén. A feltüntetett értékek másodpercben értendők, és minél kisebb az érték, annál gyorsabban betöltötte a rendszer az adatokat.

A grafikonok alapján több dolog is szembetűnő. Egyrészt látható, hogy óriási különbségek voltak az egyes rendszerek betöltési idejei között, a két szélsőérték között 317-szeres szorzó volt, de a leglassabb és a leggyorsabb rendszer átlagideje között is 191,286-szoros különbség adódott.

Másrész az is látszik, hogy a reifikációs modell választásának lényegesen kisebb hatása van a betöltésre, a legnagyobb mért hatás 2,268-szoros, a legkisebb 1,264-szeres változás volt. Ez lényegesen kisebb, mint a rendszerek betöltési ideje közötti átlagos 3,135-szörös különbség.

Harmadrészt megfigyelhető, hogy a Blazegraph kivételével valamennyi rendszer esetén ugyanaz a mintázat látható: az éltulajdonság modell töltődött be a leggyorsabban, ezt követi sorrendben az n-ary, a standard és az általános standard modell, a leglassabb pedig a singleton tulajdonság modell volt (a Neo4j esetén az utóbbi kettő fordított sorrendet mutat, de közel azonos eredménnyel). Ez az általánosan megfigyelhető sorrend alapvetően a modellek jellegére vezethető vissza. A legkevesebb gráf elemet az éltulajdonság modell tartalmazza, míg a legtöbbet az általánosított standard modell, köztük helyezkedik el az n-ary és a standard a betöltési idővel egyező sorrendben. A singleton tulajdonság azért speciális, mert ugyan kevesebb elemet tartalmaz, mint az általánosított standard modell, azonban az adatsémája minden él hozzáadásakor bővül, ami a betöltési sebességet átlagosan több, mint másfélszeresére növelte a vizsgált rendszereknél.



5.2. ábra. Az egyes gráfadatbázisok betöltési idejei a különböző reifikációs modellek esetén.

A mérések alapján a leglassabban az Azure CosmosDB-be töltődtek be az adatok. Explicit séma definiálásra sem adatbetöltés előtt, sem közben nincs szükség, a rendszer automatikusan módosítja a sémát, egyedül a korábban említett partícionáló kulcs megadása kötelező. Tekintve, hogy a Gremlin API meglehetősen fiatal, tömeges betöltő eszköz nincs még hozzá, csak egy tömeges betöltést valamennyire támogató, előzetes állapotban levő .NET-es driver könyvtár, így ezt felhasználva egy saját importáló komponenst készítettem a betöltéshez. Ezen felül a betöltés kiugró lassúsága mögött több más okot is találtam:

- A felhő-alapú tulajdonságból adódóan a betöltést végző, és az adatokat fogadó komponens különböző gépeken helyezked(het)nek el, így az adatokat ténylegesen át kell küldeni hálózaton, amihez tartozó kommunikációs időt a mérési eredmények tartalmazzák.
- Az előfizetés idő- és adathasználat-alapú, ami miatt a betöltéshez viszonylag alacsony átviteli sebesség korlátozást kellett beállítanom, hogy az ingyenes próbaverzió keretein belül lényegileg korlátozás nélkül le tudjam futtatni később a lekérdezéseket (maradjon rájuk az előfizetési időből). Lényegesen magasabb átviteli korlát mellett vélhetően a betöltési idők biztosan kisebbek lennének.

A JanusGraph az adatbetöltés szempontjából speciális helyzetben van, ugyanis architektúrájából adódóan nem ő felel a tényleges adattárolásért, hanem a kiválasztott storage backend. Más, elsősorban a jelenleginél lényegesen nagyobb adatmennyiségre, és elosztott tárolási környezetre tervezett tároló szolgáltatásokhoz (pl. Cassandra) létezik hatékony tömeges betöltő eszköz, azonban a vizsgált adathalmazhoz ideálisnak számító BerkeleyDB-hez nem, így szintén saját komponenst kellett készítenem. A betöltés szempontjából fontos, hogy az adatséma elemeit – a csúcsok és élek típusait és tulajdonságaikat – explicit definiálni szükséges, azonban erre az adatbetöltés előtt és közben is van lehetőség. A rendszerhez tartozó mérési eredményeket a korábbi megállapítások jól magyarázzák, a singleton tulajdonság modell kiemelkedő lassúságát az on-the-fly sémamódosítások okozzák.

Habár az OrientDB rendelkezik saját adatbetöltő eszközzel [24], ennek használata során több problémába is belefutottam – megfelelő konfigurációs fájlok elkészítése, kollekció tulajdonságok importálás –, amiket hosszas próbálkozás után sem tudtam megoldani, így ehhez a rendszerhez is saját komponenst készítettem. Habár támogatott a teljesen sémamentes használat, illetve az automatikus séma generálás is [73], én – elsősorban az esetleges betöltési hibák kiszűrése céljából – a séma explicit megadását választottam. Az OrientDB és a JanusGraph különböző modelleken tapasztalt eredményei arányaikat tekintve hasonlók, azonban közel egy nagyságrendi eltérés van köztük, amit alapvetően az élek betöltési mechanizmusára vezettem vissza. A JanusGraph esetén objektum szinten szükség van a forrás és cél csúcspontokra él beszúrásakor – tehát meg kell keresni őket, míg az OrientDB esetén elegendő az azonosító megadása, így nincs szükség a csúcspontok (explicit) keresésére, ami gyorsítja a betöltést.

A TigerGraph a gyors adat betöltéshez háttér job-ok definiálására ad lehetőséget, a méréseket is ezek segítségével végeztem el. Ezen job-ok explicit tartalmazzák a séma leírását, ami egyrészt azt eredményezi, hogy a betöltést nagyon gyorsan el tudja végezni a rendszer, másrészt, hogy a sémát az adatok betöltése előtt „manuálisan” definiálni szükséges. Ez az előre definiáltság a natív végrehajtással párosítva a leggyorsabb importálást eredményezi a vizsgált rendszerek között, cserébe meglehetősen rugalmatlan.

A Neo4j szintén rendelkezik saját betöltő komponenssel, azonban a TigerGraph-tól eltérően nem a rendszerbe épített folyamatokkal, hanem egy külön erre a célra dedikált alkalmazással lehet létrehozni a forrás adatokból a bináris adatbázis fájlokat, amiket konfigurációs fájl átírással lehet aktiválni. A kiinduló állományok struktúrája kötött, így a betöltő eszköz viszonylag gyorsan működik, illetve emiatt explicit séma definiálásra sincs szükség, mert azt az input adatok alapján a rendszer automatikusan kialakítja. A rendszer tényleges sémamentessége, és a fizikai tárolási mechanizmusa miatt az új „séma elemek” felvitelét nagyon hatékonyan tudja elvégezni a rendszer, ez az oka annak, hogy a singleton tulajdonság modell betöltési ideje nem kiugróan magasabb a korábban bemutatott rendszerhez hasonlóan, hanem – a betöltött gráf elem számának megfelelően – a standard és az általánosított standard modellek eredményei közé esik.

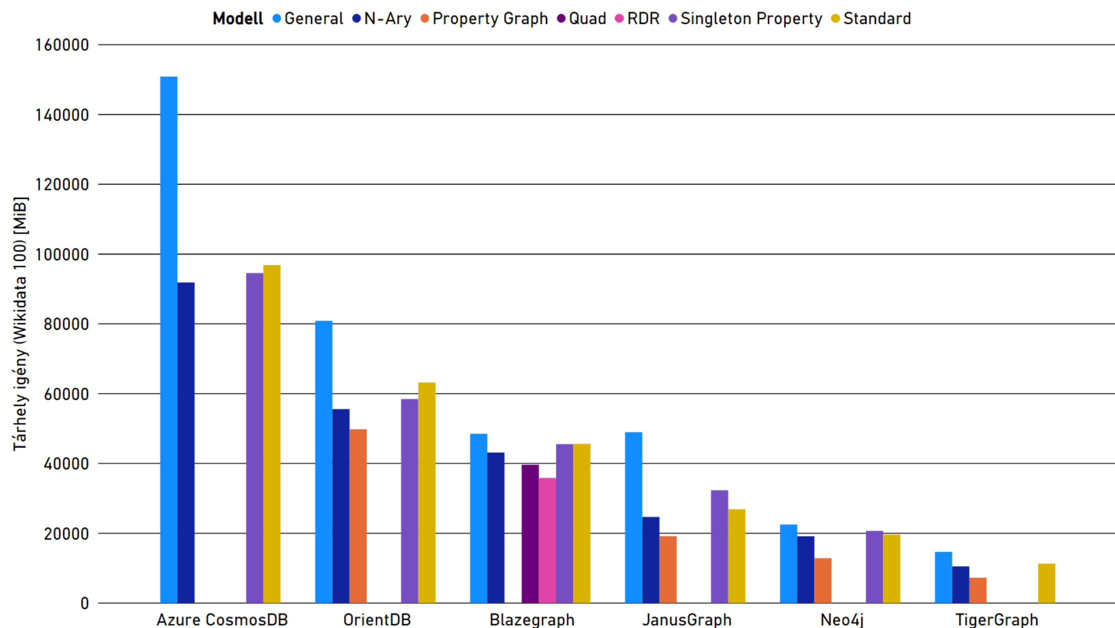
Az eltérő gráfkoncepció miatt a Blazegraph rendszer esetén a betöltési idők lényegesen eltérő mintázatot mutatnak, ugyanis az RDF jellege miatt nem a beszúrt gráf elemek, hanem a beszúrt állítások száma határozza meg betöltés sebességét. Mivel az egyes modellek hasonló mennyiségű állítást használnak, így – legalábbis az arányokat tekintve – kisebb a különbség az egyes modellek betöltési ideje között. Az RDF jellegből adódik továbbá az is, hogy az erőforrások beszúrása nem függ az állításon belül elfoglalt szerepüktől (gráfokra lefordítva, hogy csúcsról vagy élről van szó), így a singleton tulajdonságok nem okoznak komplex, állandóan változó sémát, így extra lassulást sem. A rendszer a tömeges betöltés támogatására egy harmadik koncepciót használ, egy REST API végpont van dedikálva erre, amit a betöltendő fájl elérési útvonalakkal kell paraméterezni. Ezen támogatás ellenére a Blazegraph betöltési ideje meglehetősen lassú – például az OrientDB a gráfelemek egyenként beszúrásával is átlagosan 3,876-szor gyorsabb –, aminek az oka a betöltési mechanizmus belső működésében keresendő. A lassúság fő oka, hogy – a tárhelyekre vonatkozó fejezetben részletesen kifejtett – betöltési működés során először egy átmeneti fájlba kerülnek az adatok, majd innen kerülnek átmentése a tényleges adatbázis fájlba, ami rengeteg plusz I/O műveletet eredményez, ezzel növelve betöltési időt.

5.2. Tárhely

Az adatbázis-kezelők rendszerint nagy mennyiségű adat hatékony kezelésére vannak kitalálva. Ugyan a memóriatechnológia fejlődésnek köszönhetően ma már egyre kevésbé jelent problémát a megfelelő mennyiségű memória és tárhely biztosítása, a rendszerek kapacitástervezésekor fontos figyelembe venni az egyes rendszerek tárolási igényeit és sajátosságait.

A kapacitástervezés szempontjából két fontos kérdés merülhet fel: Mekkora tárhelyre van szükség adott adat tárolásához? Hogyan skálázódik a szükséges tárhely az adathalmaz méretének növekedésével? Utóbbi kérdést a vizsgált rendszerek esetében a skálázódásról szóló fejezetben válaszolom meg. Mindezek mellett ide vonatkozóan érdekes lehet még, hogy az egyes adatbázis-kezelők támogatják-e az adatok tömörített tárolását.

Az 5.3 ábrán az látható, hogy a Wikidata 100 adathalmazt betöltve a vizsgált rendszerekbe, különböző reifikációs modelleket használva, hogyan alakul a szükséges tárhely igény az egyes esetekben. Az egyes értékeket MiB mértékegységben tüntettem fel, így minél kisebb az érték, annál kisebb tárhelyre volt szükség az adatok adott implementáció-modell párosban történő eltárolására. Referenciaként, a nyers Wikidata JSON állomány több, mint 64 GiB tárhelyet foglal, a köztes reprezentációban azonban már 7 GiB is elegendő a tárolásra.



5.3. ábra. Az egyes gráfadatbázisoknak a Wikidata 100 adathalmaz tárolásához szükséges tárhely mérete különböző reifikációs modellek esetén.

A betöltési időkhöz hasonlóan a szükséges tárhely igény esetén is hatalmas különbségek voltak az egyes adatbázis implementációk között, hiszen a legkompaktabb és a legnagyobb igényű konfiguráció között 20,75-szeres különbség volt, továbbá az egyes rendszereken az összes reprezentáció átlagát nézve is, a szélsőértékek között 9,919-es tárhely faktort mértem. Arányaikat tekintve tehát tárhely igényt tekintve közelebb vannak egymáshoz a rendszerek.

A betöltési idővel szemben a reifikációs modell különbségei dominánsabban jelennek meg a tárhelyben, ugyanis rendszertől függően átlagosan 1,35-től egészen 2,55-szörös különbséget eredményezett csupán a logikai modell megváltoztatása. Ez nem csak abszolútértékben nagyobb, de már a rendszerek közötti különbségekkel is összemérhető. Ezt az

ábra alapján is látni lehet, amin lényegesen több esetben van olyan, hogy egy rendszer két modellen mért értéke közé esik a másik rendszer egy értéke – például a Blazegraph és JanusGraph esetén.

Ezzel szemben hasonlóság a betöltési időhöz képest, hogy megfigyelhető néhány mintázat a reprezentációk sorrendjét illetően. Némiképp nyilvánvaló, hogy kapcsolat van az adathalmaz reprezentálásához szükséges gráfelemek száma és a szükséges tárhely között, amit jól látni lehet az ábra alapján is. A legkevesebb gráfelemre az éltulajdonság modellben van szükség, ennek megfelelően minden kapcsolódó rendszer esetén ez is a legkompaktabb modell. Az általánosított standard modell használja egyértelműen a legtöbb elemet, ami láthatóan a legnagyobb tárhely igénnyel is jár.

Ehhez kapcsolódóan érdekes megvizsgálni, hogy a multimodális rendszerek esetén a singleton tulajdonság megközelítés tömörebb, mint a standard modell, míg a natív gráfadatbázisok esetén fordított a sorrend. Ezt leginkább azzal lehet magyarázni, hogy a multimodális adatbázis-kezelők kevésbé hatékonyan tudják a kapcsolatokat (éleket) leképezni, hiszen több adatmodell szükségleteit is figyelembe kell venniük a tároláskor.

Az egyes rendszereket az átlagos tárhely igény szerint sorba rendezve szintén az tapasztalható, hogy a multimodális rendszerek használják a legtöbb területet. Ez szintén visszavezethető a multimodalitással járó általános tárolási struktúrákra (ARS és record), ahol több kiegészítő információt is tárolnia kell a rendszernek a tényleges adatokon felül – például az Orient DB és az Azure Cosmos DB is saját belső azonosító mezőt generál és tárol minden csúcshoz, Orient DB esetén ehhez még hozzájönnek a séma információk.

A Blazegraph ezen szempont esetén is érdekes kivételt jelent. A grafikon alapján látható, hogy meglehetősen többi rendszerhez képest viszonylag kicsi a különbség az egyes reprezentációk között. Ez alapvetően két okra vezethető vissza:

1. A többi rendszer esetén – leszámítva az Azure Cosmos DB-t – az éltulajdonság alapú modell volt a leggyorsabb, és az általánosított standard a leglassabb. Mivel a Blazegraph esetén előbbi nem került megmérésre, így ennek hiánya a szélsőértékek közeledését eredményezte.

2. A Blazegraph RDF* adatmodell fölött dolgozik, ami tárolási elemként egyedül az állítások fogalmát ismeri, így a tárhely igény gyakorlatilag csak ezek számától függ. Abból kiindulva, hogy az általánosított standard és az n-ary modellek állításainak száma között többszörös szorzó van, de a tárhely igényük alig különbözik, arra lehet következtetni, hogy a Blazegraph rendelkezik egy viszonylag nagy alap overhead-del, ami fölött egy meglehetősen hatékony tárolási mechanizmus van implementálva. Ez magyarázatot ad mind a nagy tárhely igényre, mind a kis különbségekre.

A Blazegraph esetén további érdekesség a betöltéskor tapasztalható tárhelyigény-robbanás. A betöltés során a betöltendő adatokból a Blazegraph egy ideiglenes segédstruktúrát épít fel a napló fájlban, majd ebből végzi el a pontos betöltést. Ez a segédstruktúra az egy tranzakcióban betöltendő adatok méretétől függ, és meglehetősen nagy méretű. Ennek következtében, ha a Wikidata 100 adathalmazt egy tranzakcióban akartam betölteni, akkor ugyan a végeredmény minden esetben 50 GiB alatt maradt, a betöltés során az említett naplófájl mérete bizonyos esetekben meghaladta a 180 GiB-ot is, ami a közvetlenül a betöltés után törlésre került. Ezen okból kifolyólag, a mért adatbetöltéseket a Blazegraph esetén több kisebb tranzakcióval végeztem.

A TigerGraph esetén korábban említésre került, hogy nagyon hatékony tömörítési mechanizmust használ a tároláshoz, ami miatt ez a rendszer képes a legkompaktabb módon tárolni az adathalmazt. A TigerGraph mellett azonban a Neo4j [11] és az Orient DB [25] is rendelkezik hasonló lehetőséggel, de kifinomultságban messze elmaradnak a TigerGraphtól. Mindkét esetben az éltulajdonság és a standard modell esetén vizsgáltam meg ezen

módok hatását. Előbbi esetén a Neo4j 2,18%-kal, az Orient DB 1,47%-kal foglalt el kisebb helyet, míg utóbbi a Neo4j-nél 3,99%-kal, az Orient DB-nél 6,35%-kal volt kompaktabb.

Az adatok tárolására a Azure Cosmos DB-nek volt messze a legtöbb tárhelyre szüksége. Ez a korábban említett multimodális adattárolás mellett egy másik okra is visszavezethető, az magas rendelkezésreállás biztosítására. A rendszer 99,99%-os rendelkezésre állást biztosít egy régió esetén, amit négyszeres elosztott redundanciával biztosít, azaz minden adat régiónként legalább négy példányban szerepel, lehetőség szerint más partíciókban [31]. Ezen redundancia nélkül az egyik legkompaktabban tároló rendszer lenne, még úgy is, hogy multimodális, és nem alkalmaz tárolási tömörítést.

Észrevehető, hogy az Azure Cosmos DB, az Orient DB és a JanusGraph esetén az általánosított standard modell kiugróan sok helyet foglal. A pontos okát nem sikerült kideríteni, azonban abból kiindulva, hogy ez a modell az élek számában kicsi, a csúcsok számában viszont nagy többlettel rendelkezik a standard modellhez képest, arra a következtetésre jutottam, hogy ezen rendszerek esetén az élek és a csúcsok drasztikusan különböző hatékonysággal tárolhatók, pontosabban az élek valamilyen okból sokkal kompaktabban tárolhatók, mint a csúcsok.

5.3. Teljesítmény, válaszidő

Habár a többi vizsgált paraméter sem elhanyagolható, az adatbázis-kezelők szempontjából a legfontosabb nemfunkcionális mérték alapvetően a rendszer teljesítménye. A vizsgált gráfadatbázisok sebességét a korábban ismertetett lekérdezési minták segítségével mértem meg. Minden ilyen minta 5 különböző változó behelyettesítéssel (lekötéssel) lett lefuttatva. Annak érdekében a működés közben fellépő tranziens jelenségeket hatását kiküszöböljem – például, hogy az indítás utáni első legkérdezés általában lassabban fut le – minden behelyettesítést 5-ször futtattam le. Ezek alapján mindegyik lekérdezési mintához konfigurációnként összesen 25 mérési eredmény tartozik (5.4 ábra). Mivel ennyi lekérdezés limitáció nélküli futtatása irreálisan sok időt igényelt volna, így minden rendszer esetén egységesen 3 perc futtatási időlimitet (timeout-ot) állítottam be.

Az 5.4 ábrán az egyes gráfadatbázisok átlagos válaszidejei láthatók lekérdezési mintánként, különböző reifikációs modellek használata mellett. A feltüntetett értékek az adott mintához és konfigurációhoz tartozó 25 lekérdezés átlagos válaszidejei milliszekundumban feltűntetve, ebből kifolyólag a kisebb érték gyorsabb működést jelent. A beállított timeout miatt a lehető legnagyobb érték 180000, így az időkorlátot túllépő lekérdezéseket 180001 értékkel számoltam az átlagba.

Az eredményekre globálisan ránézve az látható, hogy a rendszerek két csoportba sorolhatók. Az egyikben található a Neo4j, az Azure Cosmos DB, a JanusGraph és a Blazegraph, melyek a lekérdezés mintától függően hol nagyon gyorsak, hol nagyon lassúak voltak – akár tízszeres válaszidő különbséget tapasztalva eltérő minták esetén. A másik csoportba az OrientDB és a TigerGraph tartozik, amik teljesítményét – legalábbis abszolútértéket tekintve – csak kis mértékben befolyásolta a lekérdezés típusa, konstans gyors válaszidőket nyújtottak. Mindezek mellett a grafikonokról az is leolvasható, hogy a lekérdezések teljesítményét ugyan alapvetően az adatbázis kezelő határozza meg, sok esetben a reifikációs modell megválasztásának is óriási – akár több, mint kétszeres válaszidő növekedés – teljesítmény vonzata lehet.

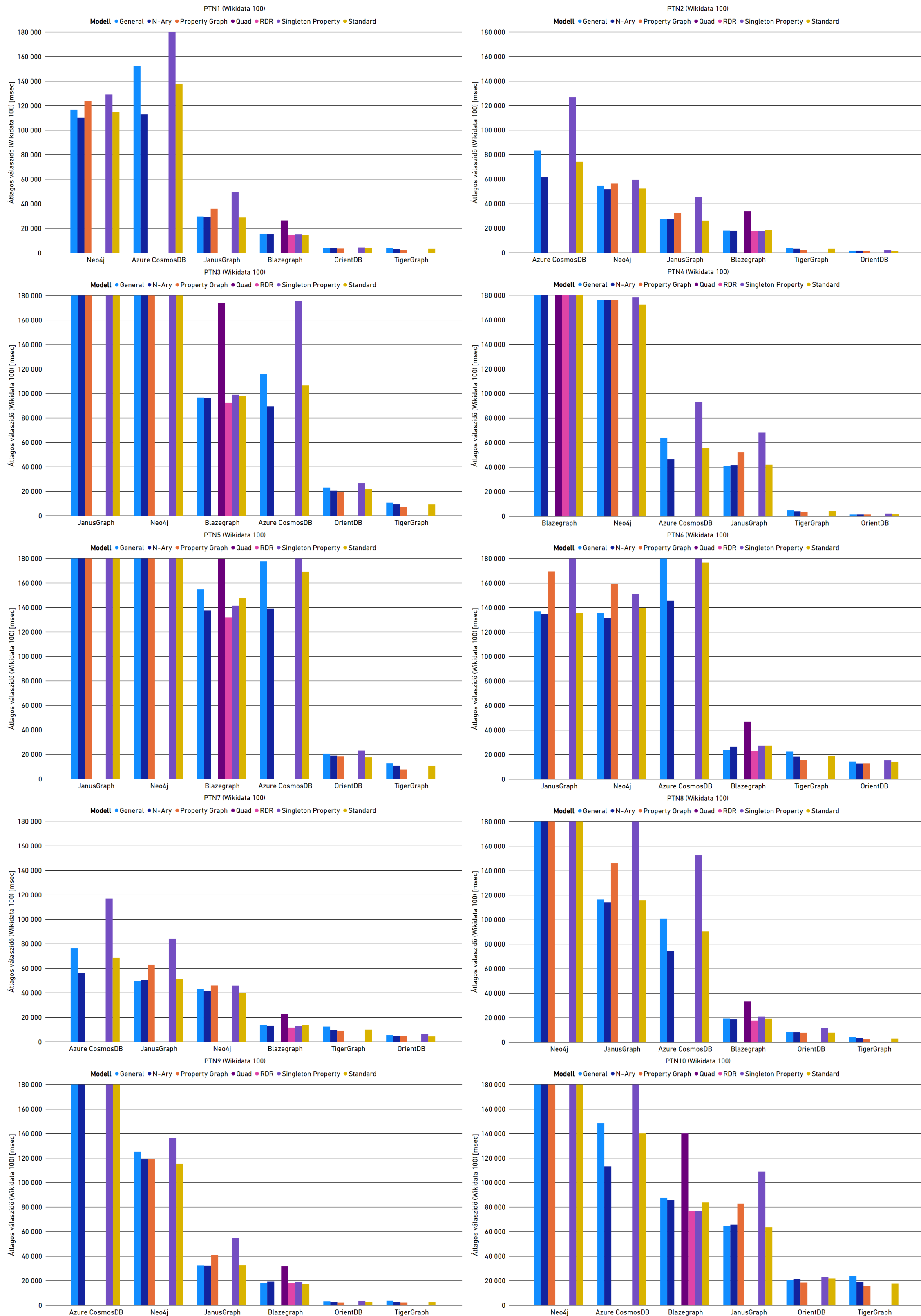
Neo4j

A megvizsgált rendszerek közül – a teljes futásidőt tekintve – a Neo4j volt messze a leglassabb, ami a grafikonok alapján is látható, ugyanis 8 lekérdezési minta esetén a leglassabb kettő rendszer egyike volt, míg a maradék 2 esetén is csak a harmadik leglassabb teljesítményt nyújtotta. Mindezt tovább erősíti, hogy a Neo4j-nél tapasztaltam a legtöbb timeout-ot is, pontosan 656-ot – a második legtöbb a JanusGraph-nál volt, 300.

A kapott eredményekre mintánként ránézve, a rendszer teljesítményének több alapvető jellemzője és gyengesége is megállapítható. Habár a PTN1 és a PTN2 hasonlít egymásra, egy fontos elemben különböznek: az ismert elemektől a legtávolabbi változó a PTN1 esetén kettő távolságra van, míg a PTN2 esetén csupán egy. Ha figyelembe vesszük, hogy a PTN1 esetén az átlagos idők közel kétszeresei a PTN2 idejeinek – valamennyi logikai reprezentáció esetén –, akkor arra a következtetésre juthatunk, hogy a Neo4j csak szélességi keresést (BFS-t) használ a mintaillesztés elvégzésekor.

A PTN3 és a PTN5 eredményei egyértelműen azt mutatják, hogy a Neo4j sem a csoportosítást, sem az erre épülő aggregációs számításokat nem tudja hatékonyan elvégezni, ugyanis az összes ezekhez tartozó lekérdezés túllépte a beállított időkorlátot.

Egy másik fontos, közös lekérdezés jellemző található a PTN4, PTN8 és a PTN10 minták esetén: ezen minták tartalmaznak ismeretlen éltípusra vonatkozó részt is. A korábbi megfigyeléseimmel összhangban, amennyiben egy lekérdezés nem specifikálja az összes él típusát, akkor ezt a Neo4j továbbra sem képes hatékonyan végrehajtani, ahogy az a diag-



5.4. ábra. A vizsgált gráfadatbázisok átlagos válaszidejei az egyes lekérdezések esetén, különböző reifikációs modellekben.

rammokon is látható. Ez alól érdekes kivételt jelent a PTN7 esete, azonban nem sikerült megállapítanom, hogy ebben az esetben mi áll a viszonylag gyors végrehajtás mögött.

A Neo4j további gyengeségeire mutatnak rá a PTN6, PTN8 és a PTN9 lekérdezési mintákhoz tartozó eredmények. A PTN6 egy nem definiált hosszúságú út megkeresését írja le a gráfban, így némileg meglepő módon a Neo4j még az ilyen tipikus gráf műveletek megvalósításában sem igazán hatékony – ahogy a JanusGraph és az Azure Cosmos DB sem. Ugyan nem feltétlen elvárás egy gráfadatbázistól, hogy hatékonyan tudjon halmazokat kezelni, azonban a Neo4j ezen a téren is csak az Azure Cosmos DB-t tudta megelőzni.

Mindezek ellenére mégis a PTN8 – és részben a PTN10 – mutatnak rá a dolgozat szempontjából leginkább releváns és legsúlyosabb problémára: amint olyan lekérdezést fogalmazunk meg, ami tartalmaz metaadatra vonatkozó részt is – akár kötött, akár változó formájában –, a Neo4j nem képes ennek a hatékony végrehajtására, a kiértékeléskor a teljes élhalmazt összes elemét ellenőrzi, emiatt drasztikusan megnő a rendszer válaszideje.

Érdemes megnézni a Neo4j teljesítményét a különböző reifikációs modelleken. A grafikonok alapján az látszik, hogy az éltulajdonság és a singleton tulajdonság egy kivételével minden esetben a két leglassabb volt – és a kivételes esetben sem voltak a leggyorsabbak. Az intuitivitással és kompaktsággal szemben az éltulajdonság modell rendelkezik egy óriási hátránnyal ezen rendszer esetén: alkalmatlan metaadatokra vonatkozó lekérdezések kiszolgálására. A PTN8 minta egy lekérdezését időkorlát beállítása nélkül is lefuttattam valamennyi reifikációs modellen, és az éltulajdonság modell volt az egyetlen, aminek a végrehajtása 4 óra után sem fejeződött be – ezt követően manuálisan megszakítottam a futtatást. A Neo4j-n vizsgált másik három modell nagyon hasonló teljesítményt nyújtott, a legtöbb esetben – ugyan nagyon kicsivel – a standard modell bizonyult a leggyorsabbnak.

Tekintve, hogy ennél a rendszernél lehetőség volt a lekérdezési tervek mentésére és későbbi elemzésére is, így a Neo4j teljesítményét tudtam a legmélyrehatóbban elemezni. Ezen lekérdezési tervek elemzésekor arra az érdekes eredményre jutottam, hogy egy konkrét lekérdezés többszöri futtatása esetén – az adathalmaz megváltoztatása nélkül – a legtöbbször teljesen eltérő lekérdezési terv készül, például a mintaillesztések másik csomópontokból indulnak. Ez alapján arra a következtetésre jutottam, hogy a Neo4j lekérdezés optimalizálója nem determinisztikusan működik.

A lekérdezési tervek további elemzéséből azt is kiderítettem, hogy a lekérdezés optimalizáló más szempontból sem működik teljesen jól. Bizonyos esetekben (közel) optimális módon végezte el a mintaillesztést, azonban a legtöbb esetben rossz sorrendet választott. Utóbbira legjobb példa a PTN8 esete, ahol valamilyen okból minden esetben a teljes csúcshalmaz minden eleméből keresést indított a bejövő élek mentén, ahelyett, hogy az ismert csúcsból indulna ki. A korábbi [81] munkámban tapasztaltakhoz képest érezhetően fejlődött ez a része a rendszernek, de továbbra sem működik hatékonyan nagy méretek esetén.

A korábbiaknak megfelelően, a rendszer lehetőséget az tippek megadására a lekérdezés kiértékeléséhez query hint-ek formájában. Ezek használatát kipróbáltam 4 lekérdezési minta 1-1 lekötésével. A kapott eredmények alapján pusztán ezen tippek hozzáadása átlagosan 16,33%-ot gyorsított a lekérdezések végrehajtását, figyelemre méltó azonban, hogy a PTN4 esetén 48,32%-os javulást tapasztaltam. További érdekesség ezekkel kapcsolatban, hogy a szándékosan rosszul megadott tippek észrevehetően nem befolyásolták a végrehajtást, ugyanis ezen esetekben 0,7%-nál is kisebb átlagos eltérést tapasztaltam csak.

További érdekesség a rendszer teljesítményével kapcsolatban, hogy meglehetősen nagy hatással van rá a JRE futtatókörnyezet verziója. Minden minta 1-1 lekötéséhez tartozó konkrét lekérdezését lefuttattam 8-as és 11-es JRE verzió fölött is. Ezen mérések során azt tapasztaltam, hogy pusztán a Java verzió 8-ról 11-re növelése átlagosan 5,2%-os gyorsulást eredményezett. A mérési eredmények már ennek a gyorsabb konfigurációnak az eredményeit tartalmazzák.

Összegezve a Neo4j teljesítményét, az látható, hogy konkurens rendszerekhez hasonló teljesítményt a vizsgált esetek közül egyedül a PTN7-esetén tudott nyújtani, minden

más esetben található több, ennél lényegesen gyorsabb alternatíva. Mindezek ismeretében némiképp meglepő a népszerűségi listán elfoglalt első helye.

Blazegraph

A Neo4j-vel összehasonlítva a Blazegraph lényegesen jobb eredményeket produkált. Az ide vonatkozó grafikonok alapján könnyen észrevehető, hogy az egyes lekérdezési minták a végrehajtási sebesség szempontjából három csoportba sorolhatók be: lassú végrehajtás, biztos timeout és nagyon gyors végrehajtás.

Megvizsgálva a PTN3 és a PTN5 futási eredményeit, az látható, hogy ezen lekérdezések esetén a Blazegraph is lényegesen lelassul, de messze nem olyan mértékben, mint a Neo4j. Ez tehát azt jelenti, hogy a Blazegraph sem képes az aggregációt vagy csoportosítást tartalmazó lekérdezések igazán hatékony futtatására, hiszen 5-7-szeres teljesítmény romlás tapasztalható a PTN2-höz viszonyítva, bár a megadott időkorláton belül teljesíteni képes azokat – a quad módot leszámítva, ami a PTN5 esetén folyamatosan elérte a timeout-ot. A két lassulást okozó lekérdezés elem közül, az ábra alapján is láthatóan, az aggregációk végrehajtása gyorsabb a csoportosításnál.

Mivel a Blazegraph is lehetőséget ad a lekérdezés kiértékelési terv megtekintésére, így lehetőségem volt alaposabban megvizsgálni, hogy mi okozza a PTN5 lassúságát. A lementett tervből azt lehetett kiolvasni, hogy a lekérdés feldolgozási idejének 84%-át a csoportosítás művelete tette ki, ami még inkább megerősíti azt, hogy a Blazegraph meglehetősen lassan képes csak GROUP BY műveleteket elvégezni.

Mindezeken túl a rendszer igazi gyengesége a PTN4 segítségével azonosítható. Az eredmények alapján valamennyi lekérdezés timeout-ra futott az ezen mintához tartozó lekérdezések esetén, így a Blazegraph gyenge pontját az jelenti, ha ismeretlen típusú élek mentén kell visszafelé navigálnia. Az ide tartozó lekérdezések kiértékelési terveit szintén megvizsgáltam, de a lassúság okát nem sikerült azonosítani – valami okból a máshol gyorsan működő illesztés itt csak nagyon lassan hajtódik végre.

A többi minta esetén a Blazegraph nagyon jól teljesített, szinte minden esetben az Orient DB és a TigerGraph után alig lemaradva a 3. legjobb válaszidőket mértem. Mivel több esetben a részben memória adatbázisnak tekinthető TigerGraph-fal is felvette a versenyt teljesítményben, arra lehet következtetni, hogy a rendszer nagyon hatékony I/O kezeléssel rendelkezik, a lekérdezések kiszolgálásához szükséges adatokat szinte kizárólag a memóriából veszi, ahova előzetesen felolvasta. Ez a hatékonyság részben megerősíti a választás helyességét a Wikidata lekérdező szolgáltatása esetén.

Ha a reifikációs modellenként vizsgáljuk az eredményeket, alapvetően három jellegzetességet lehet észrevenni. Az egyik, hogy arányait tekintve a quad mód nagyon lassú volt, az összes lekérdezési minta esetén ez teljesített a legrosszabbul, emiatt ezt a modell nem érdemes használni Blazegraph-fal. A másik, hogy a singleton tulajdonság, a standard, az általánosított standard és az n-ary modelleke nagyon hasonló teljesítményt nyújtottak. Ezzel kapcsolatban további érdekesség, hogy ennél a rendszernél a singleton tulajdonság jó teljesítménnyel használható, általában mind a standard, mind az általánosított standard modellnél gyorsabb, azonban az n-ary megközelítésnél lassabb. A harmadik, hogy a Blazegraph esetén egyértelműen az RDR modell a legjobb választás, ugyanis az összes lekérdezési minta esetén ez a modell érte el a legkisebb válaszidőket. Ez azonban némiképp el is várható volt, tekintve, hogy egy kimondottan erre a célra kifejlesztett szabvány bővítést vezet be és használ.

Ahogy azt a Blazegraph bemutatásánál is írtam, a rendszer támogatja a TinkerPop gráf stack-et, így a Gremlin lekérdező nyelvet is. Ebből kifolyólag néhány lekérdezés esetén megvizsgáltam, hogy a Gremlin használatával jobb teljesítmény érhető-e el. Attól kifolyólag azonban, hogy első lépésben ez átfordításra kerül SPARQL-re – mindez néhány

milliszekundim alatt, a végrehajtási sebességben se gyorsulást, se lassulást nem tapasztaltam. Mivel a két nyelv teljesítmény szempontból ekvivalens, így ennek csupán annyi előnyös következménye van, hogy minden lekérdezést azon a nyelven lehet megírni, amin az a legkifejezőbb, vagy legkönnyebben leírható.

Mindezek mellett azt is megvizsgáltam, hogy a különböző Java heap méret beállításának milyen hatása van a teljesítményre. Ennek érdekében több lekérdezést lefuttattam 4 GiB és 24 GiB beállított Java heap méret esetén is, de – némiképp meglepő módon – mérhető különbséget nem tapasztaltam.

JanusGraph

Korábbi munkám során [81] már foglalkoztam a JanusGraph elődjének tekinthető Titan DB-vel, melynek teljesítményének mérését csak részben tudtam elvégezni egy vélhetően felfedezett Gremlin API implementációs hiba miatt. Ehhez viszonyítva a JanusGraph már előrelépést jelent, ugyanis a korábban tapasztalt API hiba nem jelentkezett, valamennyi Gremlin lépés az elvárások szerint működött.

Tekintve, hogy a Gremlin imperatív jellegű nyelv, a lekérdezés optimalizálónak csak minimális hatása van tényleges végrehajtási tervre, hiszen lényegileg az maga a Gremlin lépések sorozataként kiadott utasítás. Ebből kifolyólag a teljesítmény elemzése ennél a rendszernél – és majd az Azure Cosmos DB-nél is – a grafikonokról leolvasható értékek elemzését és értelmezését jelenti. A korábbiakhoz hasonlóan, ezt a lekérdezési minták, és a reifikációs modellek mentén tettem meg.

Az egyes logikai modellek teljesítményét megnézve a Neo4j-hez hasonló mintázat látható: a singleton tulajdonság és az éltulajdonság modell minden lekérdezési minta esetén a két legrosszabb válaszidőt produkálták. A másik három modell közül viszont nem lehet egy egyértelműen leggyorsabbat kiemelni – ahogy ezt a Neo4j-nél sem lehetett megtenni, ugyanis mindháromhoz található olyan lekérdezési minta, amelyiknél az adott modell volt a leggyorsabb: a PTN4 esetén az általánosított standard, a PTN2 esetén a standard és a PTN8 esetén az n-ary megközelítés. Ha mindenképp választani kellene egyet, akkor a többi paramétert is figyelembe véve – elsősorban a tárhely igényt –, az n-ary modell tűnik a legjobb választásnak.

A rendszer teljesítménye az egyes lekérdezési mintákon sokkal változatosabb, mint az eddig vizsgált rendszerek esetén. A PTN3-on és PTN5-ön nyújtott teljesítménye alapján látható, hogy ez a rendszer sem képes az aggregációk és csoportosító műveletek hatékony kezelésére, hiszen valamennyi ide kapcsolódó lekérdezés végrehajtása túllépte a megengedett 3 percet.

Ezeken túl az is látható, hogy a PTN6 és a PTN8 minták esetén is szignifikáns válaszidő növekedést tapasztaltam, ami a JanusGraph két kevésbé optimalizált területét azonosítja. A nagyobb lassulást a PTN6 mintánál mértem, ami alapján arra lehet következtetni, hogy a rendszer nem képes hatékonyan változó hosszúságú utakat keresni. A dolgozat szempontjából relevánsabb viszont a PTN8 esete, ugyanis ennek a lassú kiszolgálása azt jelenti, hogy a JanusGraph nem tudja igazán hatékonyan kiszolgálni a metaadatokra is vonatkozó lekérdezéseket.

A fennmaradó esetekben viszonylag jól teljesített a rendszer, általában az (Azure Cosmos DB, Neo4j) és az (Orient DB, TigerGraph) csoportok közé esik a teljesítménye – nagyjából középre, viszont a Blazegraph gyorsabb nála rendszerint, így leginkább csak 4. helyre sorokható.

Általánosságban az mondható el a rendszer teljesítményéről még a PTN1, PTN7 és PTN 10 alapján, hogy amennyiben egy lekérdezés nem hasonlítható az eddig részletezett mintákhoz, úgy a lekérdezés végrehajtásához szükséges idő nagyjából arányos a lekérdezés által leírt gráf minta bonyolultságával.

Azure Cosmos DB

Az Azure Cosmos DB az egyedüli olyan rendszer, amelynek a mérési eredményeit bizonyos szempontból érdemes összehasonlítani az [81]-ben tapasztaltakkal. A korábbi mérésben 5 perces timeout értéket használtam, és az összes mérő lekérdezés futtatása túllépte ezt az időkorlátot. Ehhez képest figyelemre méltó, hogy a mostani mérésben egy lényegesen nagyobb gráfon, 3 perces timeout értéket használva, több hasonló lekérdezés esetén is sikeres lefutásokat tapasztaltam, ami látványosan mutatja a rendszer teljesítményének fejlődését.

A lekérdezési minták szerint nézve az eredményeket az látható, hogy a PTN3 és a PTN5 minták esetén Blazegraph-fal közel egy szinten van a teljesítménye, azaz aggregációkat és csoportosításokat az Azure Cosmos DB sem tud igazán hatékonyan számolni. Mindezek mellett a PTN6 eredményei azt mutatják, hogy a JanusGraph-hoz hasonlóan a változó hosszúságú utak keresésével is nehezen boldogul a rendszer. A Cosmos DB számára a legnagyobb nehézséget azonban egy algoritmikusan lényegesen egyszerűbb feladat, a halmazműveletek megvalósítása - azaz a PTN9 minta lekérdezéseinek futtatása - jelenti.

Ezekkel szemben, a PTN2 és a PTN4 viszonylag jó eredményei alapján arra lehet következtetni, hogy a rendszer valamilyen okból meglehetősen hatékonyan képes – legalábbis relatív teljesítményt nézve – az élek irányításával ellentétes irányú navigációra, amennyiben a kiindulási csúcs ismert. Mindezek mellett a közös elemek keresésében sem rossz a rendszer, ugyanis a PTN7 mintán mért teljesítménye – ugyan összesítésben a legrosszabb – nagyon közel van a többi, vele egy csoportban levő rendszeréhez. A többi minta esetén a rendszer viszonylag egyenletes teljesítményt nyújt.

Az összes vizsgált rendszer közül az Azure Cosmos DB teljesítménye volt a legérzékenyebb a használt reifikációs modellre, a teljesítményét alapvetően befolyásolta, ugyanis meglehetősen nagy különbségeket tapasztaltam a leggyorsabb és a leglassabb modell között. A JanusGraph-hoz hasonlóan a singleton tulajdonság modell messze a legrosszabb eredményeket hozta, így arra következtetésre jutottam, hogy ez a modell nem igazán használható a Gremlin lekérdező nyelvvel. A többi modell minden lekérdezési minta esetén az n-ary, standard, általánosított standard modell sorrendben teljesített – a standard modell rendszerint a két modell átlaga körül mozgott, így a Cosmos DB-t az n-ary megközelítéssel érdemes használni.

A korábban említett stabilitás magasabb szinten is jellemzi a rendszert, ugyanis – az Orient DB-t és a TigerGraph-ot nem számítva – lekérdezési konfigurációnként (lekérdezési minta, reifikációs modell páronként) ennél a rendszernél mértem a legkisebb szórást a válaszidőkben.

Orient DB és TigerGraph

A mérések során az Orient DB és a TigerGraph kiemelkedtek a többi gráfadatbázis közül, a teljesítményük messze felülmúlta azokét. Mindkét rendszer nagy gyors, nagyon hatékony a lekérdezések kiszolgálását illetően, a válaszidejeik a legtöbb esetben nagyon közel esnek egymáshoz.

Abból kifolyólag, hogy a multimodális Orient DB lemezes használati mód mellett is közel azonos, sokszor pedig kisebb válaszidőket produkált, mint a tisztán memóriában dolgozó, natív gráfadatbázis a TigerGraph, arra lehet következtetni, hogy az Orient DB elképesztően jól I/O optimalizált, így a lekérdezéseket ő is memóriabeli adatokon futtatja – más különben a TigerGraph eredményét meg se közelíthetné.

Ugyan a rendszerek minden lekérdezési minta esetén nagyon gyors válaszidőket adtak, a különböző minták eredményei között akár négyszeres különbség is lehet – ami azonban még így is jó időnek számít, és messze a másik csoport eredményei előtt van. A legnagyobb

nehézséget a TigerGraph számára a PTN6, vagyis a változó hosszúságú út keresése okozta, míg az Orient DB a PTN3 és PTN5, azaz a csoportosítás és aggregáció műveleteket végezte el viszonylag lassan, az átlagosnál kétszer lassabban – bár abszolútértékben még így nagyon gyorsan. Ebből kifolyólag, az utóbbi két esetben tapasztalható a legnagyobb különbség a két rendszer teljesítménye között.

Az egyes reifikációs modellek teljesítménye között csak kis különbségek vannak, ennek ellenére az szinte összes lekérdezési minta esetén megfigyelhető, hogy az éltulajdonság modell a leggyorsabb, ezt követi az n-ary és a standard modell majdnem azonos teljesítménnyel, amiknél nem sokkal lassabb az általánosított standard megközelítés. Az Orient DB estén ezt követi leglassabb reprezentációként a singleton tulajdonság modell.

Mindkét rendszer esetén megvizsgáltam alternatív lekérdezés végrehajtási módok teljesítményét is. A TigerGraph esetén az interpretált módú futtatást vizsgáltam meg néhány lekérdezés futtatásán keresztül. A tapasztalt eredmények alapján pusztán a lekérdezések natív futtatása 2,79-szeresére gyorsítja a végrehajtást.

Az Orient DB esetén a multimodalitásból adódóan azt próbáltam ki, hogy más lekérdező nyelven keresztül gyorsul, vagy lassul a lekérdezések végrehajtása. Ehhez a Gremlin lekérdezések, és a gráfokkal kiegészített SQL utasítások eredményét hasonlítottam össze, de mérhető eltérést nem tapasztaltam. Ugyanezen lekérdezések estén az Orient DB-nél bemutatott indexelés teljesítményre vett hatását is lemértem, és ezek definiálásával megközelítőleg további 1,41%-os gyorsulást sikerült elérnem.

5.4. Skálázhatóság

Ahogy napjainkban a felhő-alapú technológiák egyre kifinomultabbá válnak, úgy válik egyre inkább fontos tényezővé az adatbázis-kezelők körében a skálázhatóság kérdésköre. Tekintve, hogy ennek a fogalomnak számos definíciója van, munkámban ez egységesen – a [90] alapján – az alábbi tulajdonságot jelenti:

Definíció 8. Skálázhatóság
A skálázhatóság az adatbázis-kezelő rendszerek azon tulajdonsága, hogy a felhasználói igények változása okán kiváltott futtatási környezetváltozás ellenére működőképes marad, továbbá a megváltozott körülményeket előnyösen ki tudja használni. ▪

Ezt a definíciót alapul véve, a környezetváltozás jellegét tekintve alapvetően két skálázási modellt lehet megállapítani. A vertikális skálázás az erőforrások méretének növekedését jelenti – például nagyobb memóriát, a horizontális skálázás pedig új erőforrás bevonását – például egy új, második példány indítását a rendszerből. A vertikális aspektusra egyedül az Azure Cosmos DB-nél térek ki, a fejezet fókusza a horizontális skálázhatóságon van. Utóbbi az adatbázis-kezelők körében manapság az alábbi fő kérdések megválaszolását jelenti:

- Támogatja-e a rendszer a több klaszteres futtatást? Ha igen, milyen modell szerint?
- Milyen beépített replikációs lehetőségeket kínál?
- Van-e lehetőség sharding-ra, azaz az adatok elosztott tárolására? Ha igen, hogyan történik a gráf partícionálása?
- Elérhető-e jól skálázható felhő-alapú erőforrásként, vagy csak on-premise megoldásként?
- Van-e hivatalos konténer (például Docker) támogatása?

Az 5.5 ábrán egy, a most felvázolt szempontok szerinti összehasonlítása látható az egyes vizsgált rendszerek skálázhatósági képességeinek.

A Neo4j-nek csak az Enterprise változata rendelkezik multiklaszteres támogatással, ami a „Causal Clustering” modellt követi [37]. Ez azt jelenti, hogy egy klaszter két csoportba tartozhat: vagy a Core szerverek közé, amiket írni és olvasni is lehet, továbbá közöttük biztosítva van az erős konzisztencia, vagy a Replica szerverek közé, amiket csak olvasni lehet, és automatikus replikáció segítségével frissülnek a Core változásai alapján, de eventually consistent modell alapján. Ezek alapján látható, hogy a rendszer a sharding-ot nem támogatja, így csak olyan gráfok kezelésére képes, amik egy gépen tárolva elférnek. Nemrégiben elérhetővé vált a Google Cloud-on belül natív felhőszolgáltatásként [52], de előtelepített virtuális gépként mindhárom nagy felhőszolgáltatónál elérhető [51]. Mindezek mellett hivatalos Docker konténerbe zárt változata is van [34].

A Blazegraph is támogatja az több klaszteres futtatást, ráadásul két lehetőséget is kínál erre: létrehozható replikációs klaszter (HAJournalServer), ami a magas rendelkezésre állást biztosítja, és shard klaszter is (BlazegraphFederation), ami pedig az adatok elosztott tárolására és feldolgozására használható – akár HDFS fájlrendszerben tárolva az adatokat [76]. Sem natív felhőszolgáltatást, sem hivatalos konténert nem találtam a rendszerhez.

A JanusGraph szintén telepíthető multiklaszteres környezetbe, ami a Gremlin utasítások elosztott futtatását teszi lehetővé, az adatok elosztott tárolásáért a kiválasztott storage backend a felelős. Ezek közül például az Apache Cassandra-t választva van lehetőség replikációra és token gyűrű alapú tárolás miatt sharding-ra is [19]. Felhőszolgáltatásként nem, de Docker konténerként elérhető [41].

	Neo4j	Blazegraph	Azure Cosmos DB	JanusGraph	OrientDB	TigerGraph
Több klaszteres mód	✓	✓	✓	✓	✓	✓
Replikáció	Centrális	Centrális	Leader, Follower, Forwarder	Token	Elosztott	?
Sharding	✗	✓	✓	✓	✓	✓
Cloud-native támogatás	✓	✗	✓	✗	✗	✓
Docker támogatás	✓	✗	✗	✓	✓	✓

5.5. ábra. Az egyes rendszerek skálázhatósággal kapcsolatos jellemzőinek összefoglalása.

A vizsgált adatbázisok közül a legjobban skálázható rendszer az Azure Cosmos DB, ami felhőszolgáltatásként jól kihasználja ennek az architekturális előnyeit, ezáltal gyakorlatilag korlátlanul skálázható. A rendszer minden 10 GiB-nál nagyobb adathalmaz esetén megköveteli az adatok fizikailag elosztott, partícionált tárolását még egy régión belül is. Az egyes adatok partíciók közötti elosztását az adatbázis létrehozásakor kötelezően megadandó partícionáló kulcs szerint végzi a rendszer, így lényegileg teljes mértékben kontrollálni lehet. A redundáns adattárolás miatt az egyes adatokhoz tartozó replikák halmazt alkotnak, melynek tagjai három szerepkörbe sorolhatók [26]:

- *Leader*: pontosan egy ilyen van minden replikációs halmazban, rajta keresztül történik az írás, illetve az ő felelőssége a változtatások replikálása a többiek számára a beállított konzisztencia modellnek megfelelően.
- *Follower*: több ilyen is lehet egy replikációs halmazban, rajtuk keresztül lehet lekérdezni az adatokat.
- *Forwarder*: pontosan egy kitüntetett follower, aminek az a feladata, hogy a leadertől beérkező változásokat továbbítsa a más régióban levő, azonos adatokat kezelő replikációs halmaz leader-ének.

Mindezek mellett az Azure Cosmos DB még vertikálisan is nagyon könnyen és jól skálázható. Az átviteli teljesítményt egy absztrakt RU/s egységben lehet megadni, és a futás közben bármikor megváltoztatni 400 és 1000000 RU/s közötti értékre. Maga az RU (request unit) egy összetett egység, ami 1 KiB adat olvasásának teljes CPU, RAM és IO költségének feleltethető meg [70].

Habár több különböző felhőszolgáltató piacteréről is elérhető az Orient DB, mégsem nevezhető felhős erőforrásnak, mert ezek valójában egy előre telepített virtuális gépet hoznak csak létre [3, 56], így vertikálisan kevésbé skálázható. A jó horizontális skálázhatóság eléréséhez ez a rendszer is támogatja a több klaszteres környezetet, igaz csak az Enterprise változatában. A replikáció megvalósításának érdekessége, hogy minden klaszter írható és olvasható is, így az írási tranzakciók esetén sem alakul ki szűk keresztmetszet [69]. Korlátozottan támogatja a sharding-ot is, az adatok elosztása osztály szinten történik [75] –

Orient DB esetén az osztály nagyjából megfeleltethető a relációs világ tábla fogalmának. Docker konténerbe zárva is elérhető [54].

A Neo4j-hez és az Orient DB-hez hasonlóan a TigerGraph esetén is csak az Enterprise kiadás rendelkezik a jó skálázhatósághoz szükséges képességekkel. A rendszer ezen változta támogatja a multiklaszteres működést, azonban van néhány limitáció erre vonatkozólag. Az egyik, hogy egy klaszterhez legalább három gépnek tartoznia kell [29], a másik, hogy a rendszert a tényleges használat előtt fel kell készíteni a multiklaszteres használatra, ugyanis az adatbázis indítása után ez már csak a teljes adathalmaz és séma definíció törlése után tehető meg jelenleg [9]. Mind a magas rendelkezésre álláshoz szükséges automatikus replikációt, mind az elosztott adattárolás támogatja. A rendszer egyik különlegessége, hogy a használat alapján képes automatikusan partícionálni a gráfot [20]. A másik különlegessége, hogy saját felhőszolgáltatással rendelkezik [84], ahol jól skálázódó SaaS-ként lehet igénybe venni. Mindezek mellett a három nagy felhőszolgáltatónál is elérhető, az Orient DB-hez hasonlóan előtelepített virtuális gép formájában [85]. Regisztrációt követően elérhető a rendszerhez tartozó hivatalos Docker image is.

5.5. Skálázódás

Ahogy az információs rendszerekben egyre több és több adat került eltárolása, úgy változnak a nemfunkcionális paraméterei is – mint a válaszidő és a tárhely. Maga a skálázódás, habár névleg nagyon hasonlít a korábban bemutatott skálázhatóságra, egy egyészen más szempontból jellemzi a rendszert:

Definíció 9. Skálázódás
A skálázódás egy adatbázis-kezelő azon tulajdonsága, hogy bizonyos kiválasztott nemfunkcionális paraméterei – például válaszidő vagy tárhely – hogyan változnak az kezelendő adat mennyiségének változásával, tipikusan növekedésével. ▪

Tekintve, hogy az eddig bemutatott mérési eredményeit is paraméterenként mutattam be, az egyes rendszerek skálázódását is a nemfunkcionális jellemzők mentén ismertetem.

Betöltési idő

Az 5.6 ábra az egyes rendszerek és reifikációs modellek relatív betöltési idő növekedését mutatják. Fontos, hogy ugyan a Wikidata30 és a Wikidata60 mérete között arányait tekintve nagyobb, de abszolút változást tekintve lényegesen kisebb változás van, mint a Wikidata60 és Wikidata100 között. A felső ábra megmutatja, hogy a Wikidata60 betöltési ideje hányszorosa volt a Wikidata30 importálási idejének, míg az alsó ugyanezt ábrázolja a Wikidata100 és a Wikidata60 esetén. Mindezekből kifolyólag a felső grafikon a „kisebb" gráf méretek esetén jellemzik a rendszerek skálázódását, míg az alsó a ténylegesen nagy adathalmazok esetén. A mérték jellegéből adódik, hogy minél kisebb az érték, annál jobban skálázódik a rendszer az adott méret tartományban.

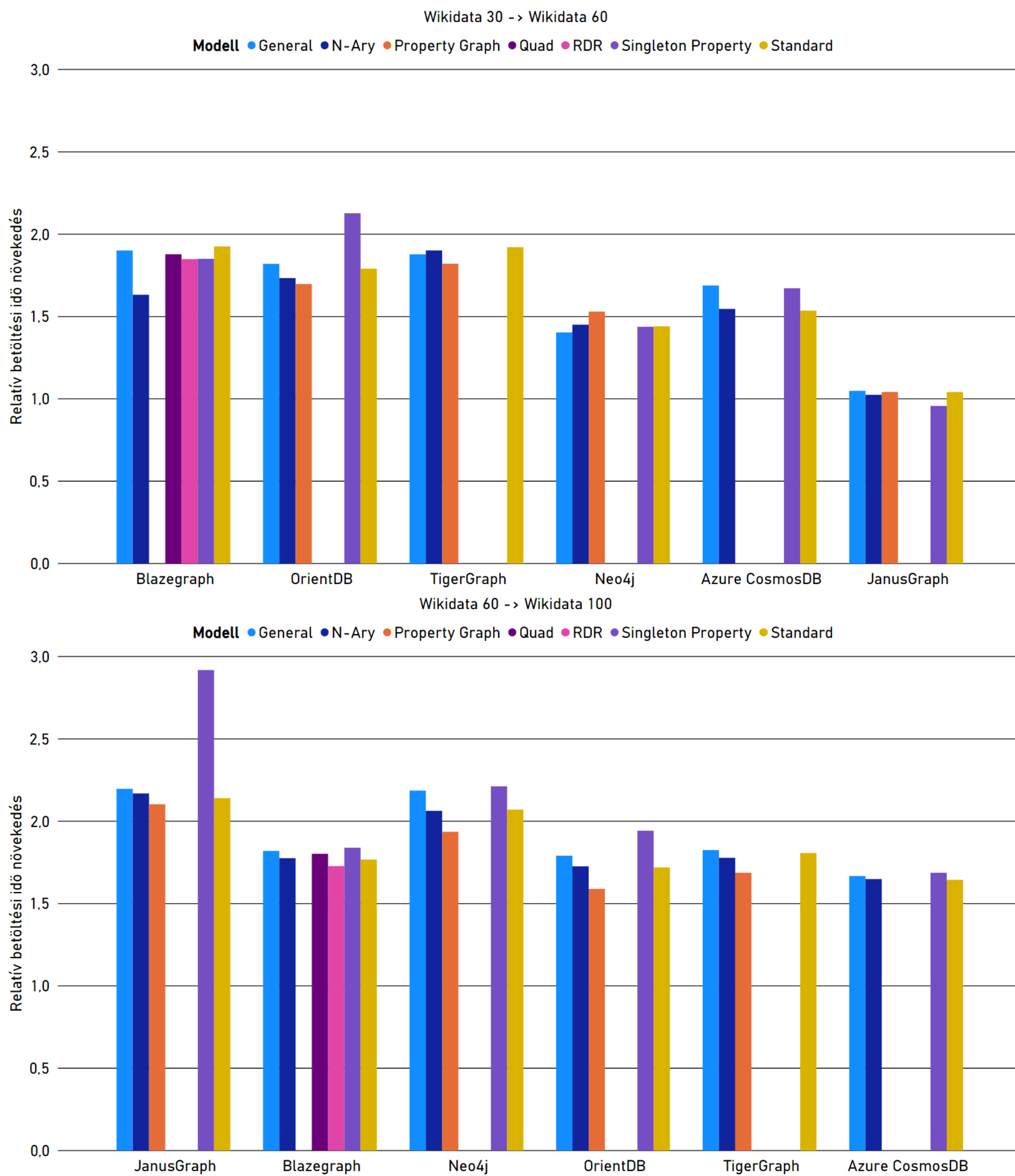
Az egyes rendszerek skázálódási képességét ennek megfelelően a két méret alapján külön-külön is érdemes megvizsgálni. A kisebb tartományban a Blazegraph-nál átlagosan 1,84-szeres betöltési idő növekedést mértem. Az n-ary modell kimagaslóan a leghatékonyabb volt a maga 1,63-szoros növekedésével. A többi modell nagyon hasonló eredményt ért el, az 1,88 érték körül helyezkedtek el 0,04-nél kisebb eltéréssel.

Ennél minimálisan hatékonyabb volt az Orient DB, amin mért betöltési idők átlagosan csak a 1,83-szorosukra nőttek. Az összes konfiguráció közül az Orient DB és singleton tulajdonság modell párosa skálázódott a legrosszabbul ezen a gráf méreten, 2,13-szorosára nőtt a betöltési idő. A többi modell skálázási faktora között kisebb, de észrevehető különbségek vannak, ezek az 1,7 és az 1,82 közötti tartományban körülbelül egyenletesen helyezkednek el.

A tartomány legrosszabbul skálázódó rendszere – ezen paraméter szempontjából – a TigerGraph volt, ugyanis átlagosan 1,88-szorosára nőttek a betöltési idők. Ezen rendszer esetén az éltulajdonság volt a legjobb 1,82-es értékével, a többi modell 1,9 körül szinte azonos – 0,025-nél nem nagyobb eltérésű – eredményt ért el.

A Neo4j összességében a második legjobban skálázódik ebben a tartományban, hiszen átlagosan csak 1,45-szörösére nőttek a betöltési idők. Az eredmények alapján az látható, hogy a singleton tulajdonság, a standard és az általánosított standard modellek szinte azonos módon skálázódnak, az n-ary ezeknél némileg, az éltulajdonság modell pedig lényegesen jobban teljesített.

Ezen méretek esetén az Azure Cosmos DB skálázódása a Neo4j és az első három bemutatott rendszer eredményei közé esik a maga 1,61-es átlagos skálázási faktorával. A grafikon alapján látható, hogy az (n-ary, standard) és az (általánosított standard, singleton tulajdonság) modell párosok lényegileg azonosan teljesítettek, de a két pár között meglehetősen nagy, 0,15 különbség volt.



5.6. ábra. A vizsgált gráfadatbázisok betöltési idejeinek relatív növekedése több gráf méret esetén, különböző reifikációs modellekben.

A kisebb gráfméret esetén egyértelműen a JanusGraph skálázódott a legjobban, és a legérdekesebben is. Valamennyi reifikációs modell közel azonos, 1 körüli skálázási faktorral rendelkezik, azaz a gráf méretének lényeges növekedése ellenére a betöltési sebesség érdemileg nem növekedett. A legérdekesebb eredményt egyértelműen a singleton tulajdonság modell adta, ami esetén a Wikidata60 adathalmaz valamivel gyorsabban betöltődött, mint a Wikidata30. A nagyobb gráfméret vizsgálata esetén az egyes rendszerek közötti különbségek csökkentek, viszont a reifikációs modellből fakadó eltérések nagyobbá váltak. Tekintve, hogy több rendszer nagyon hasonló eredményeket ért el a két különböző méreten, így csak a tapasztalt különbséget emeltem ki.

Az Orient DB esetén mind a singleton tulajdonság, mind az éltulajdonság modell teljesítménye sokat javult, így utóbbi már nem a legrosszabban skálázódó reprezentáció. Mindezek eredményeként, nagyobb méret esetén az átlagos értékeket tekintve az Orient DB a második legjobban skálázódik.

A Blazegraph esetén azt tapasztaltam, hogy a korábban legjobb eredményt elérő n-ary modell előnye eltűnt, ezen méret esetén már az RDR skálázódik a többi modellnél valamivel jobban. Hasonlóan kis változás volt a TigerGraph esetén is, ahol az általánosított standard modell eredménye romlott le. Mindkét rendszer esetén az átlagos skálázási faktor is javult kis mértékben.

Két rendszer esetén is drasztikus skálázási romlást, és átrendeződést tapasztaltam, hiszen a Neo4j átlagosan 0,5-öt, a JanusGraph pedig majdnem 1,5-öt rontott a kis méreten mért eredményein. A Neo4j esetén teljes átrendeződés volt, nagy méret esetén már az éltulajdonság modell töltődött be leggyorsabban. Hasonló volt tapasztalható a JanusGraph esetén is, ahol a korábban nagyon hatékony singleton tulajdonság modell a teljes mérés legrosszabb skála faktorát produkálta.

Legkevésbé az Azure Cosmos DB skálázása változott a méret függvényében – bár a modellek közötti különbségek itt is csökkentek, ami azt eredményezte, ennek a rendszernek a betöltése skálázódott a legjobban nagyobb méret esetén, habár messze a leglassabb volt.

Tárhely

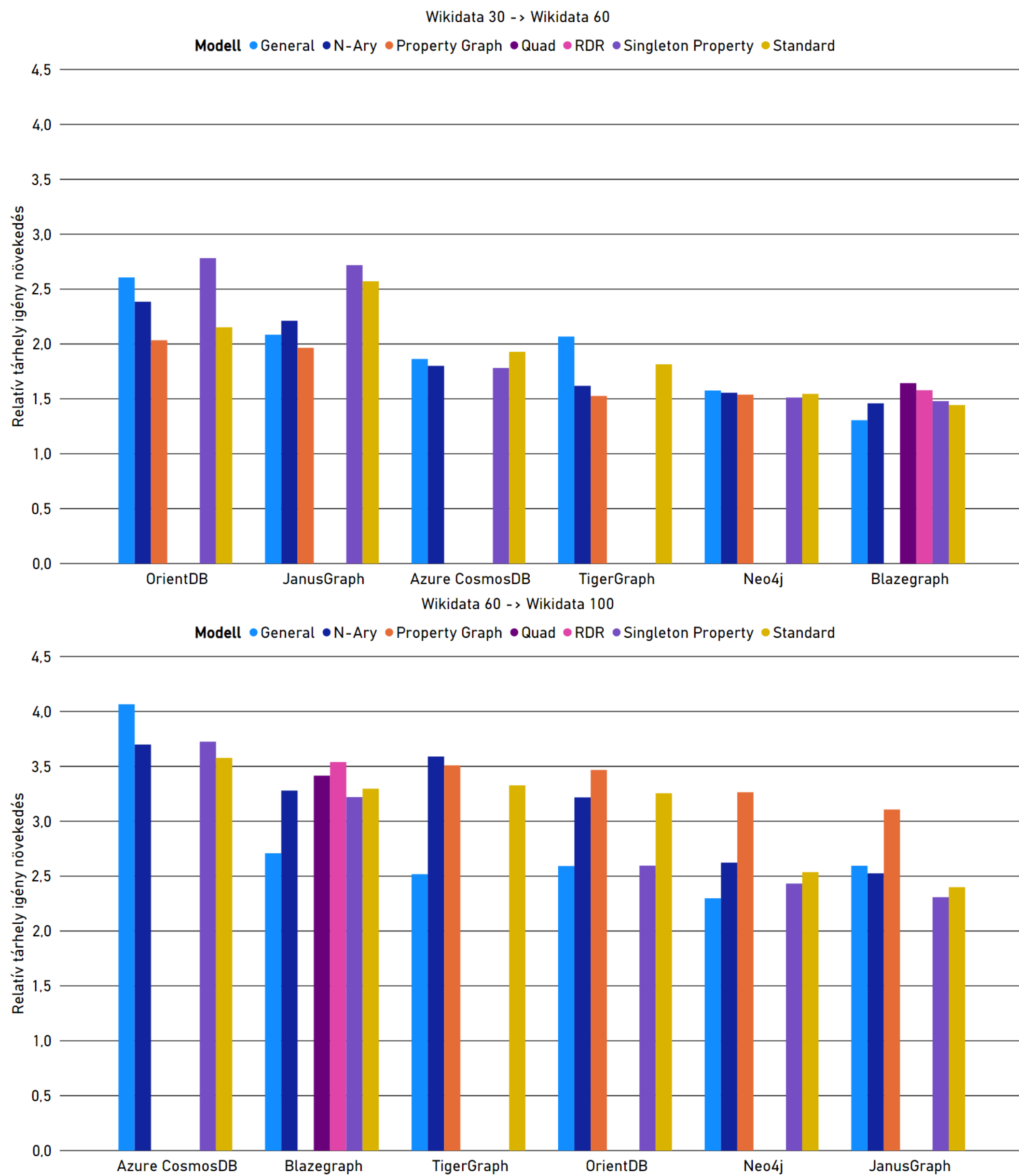
Az 5.7 ábra az előbbivel azonos logika szerint lett kialakítva, csak a feltüntetett értékek a betöltés idő helyett a szükséges tárhely igény relatív növekedését ábrázoltam. További hasonlóság az előbbi ábrával, hogy itt is a kisebb érték jelenti a jobb skálázódást.

Ezt ábrát megnézve jó látható, hogy a kisebb és nagyobb méretű gráfok esetén teljesen eltérően viselkednek az adatbázis-kezelők és a reifikációs modellek is, így ezen esetben is indokolt a két méretet külön értékelni.

A kisebb gráf méret esetén a legrosszabb átlagos eredményeket az Orient DB-nél mértem, amely esetben átlagosan 2,39-szeres tárhely igény volt tapasztalható. Meglehetősen nagy különbségek voltak az egyes reifikációs modellek skálázhatósága között. A legjobb skálázódást ennél a rendszernél az éltulajdonság modellnél mértem – 2,05-ös értéket, aminél csak kevéssel volt rosszabb a standard modell a maga 2,15-ös skála faktorával. A többi modell ezeknél lényegesen rosszabb értékeket produkált, olyannyira, hogy a singleton tulajdonság esetén volt mérhető ezen a méreten a legrosszabb skálázódás: 2,78.

A második legrosszabb átlagot a JanusGraph érte el – körülbelül 2,31-et, csak minimálisan jobban teljesítve az Orient DB-nél. Ennél a rendszernél is nagy különbségek voltak tapasztalhatók a reprezentációk között. A singleton tulajdonság és standard modellek kimagaslóan rossz értéket értek el, a többi megközelítés a 2 skála faktor körül helyezkedik el nagyjából egyenletesen. Ennél a rendszernél is az éltulajdonság modell adta a legjobb skálázási értéket.

Az Azure Cosmos DB a maga 1,84-es átlagos skálázási tényezője alapján az előző két rendszernél lényegesen jobban skálázódik. Viszonylag kis különbségek vannak csak



5.7. ábra. A vizsgált gráfadatbázisok tárhely igényének relatív növekedése több gráf méret esetén, különböző reifikációs modellekben.

az egyes modellek között, egyedül a standard modell lóg csak ki egy kicsit a rosszabb irányba. Érdekesség, hogy ennél a rendszernél az eddig nagyon rosszul teljesítő singleton tulajdonság modell volt a legjobb. A TigerGraph esetén szintén nagy különbségek voltak az egyes reprezentációs modellek skálázódásai között, ugyanis az általánosított standard modell kiugróan rosszul teljesített, illetve a standard modell sem ért el túl jó eredményt, ezzel szemben az itt is legjobb éltulajdonság modell az összes eredményt megvizsgálva is az egyik legjobb skála eredménnyel rendelkezik.

A Neo4j „kisebb” gráfokon mért skálázódási értékei elapján az látható, hogy a reifikációs modell választásától függetlenül nagyjából ugyanolyan arányban fog változni a szükséges tárhely igény, a különböző modellek között csak minimális különbség van. Mindezek mellett a Neo4j a maga 1,55-ös átlag értékével a második legjobban skálázódó rendszer lett ezen a méreten.

A legjobb átlagos skála eredményt a Blazegraph érte el: 1,49-et. Ezt annak ellenére is sikerült a rendszernek elérnie, hogy viszonylag nagyok a különbségek a modellek értékei között, hiszen például az összesített legjobb eredményt elérő általánosított standard 1,31-es értékével szemben ott van a quad modell 1,64-es értéke. A két szélsőséges modellel szemben a további négy modell az 1,49-es érték körül kisebb távolságra (maximum 0,08-ra) helyezkedik el. A tárhely paraméter érdekessége, hogy a „kisebb” és a nagy adathalmazok esetén nagyon máshogy viselkednek a különböző konfigurációk, a reprezentációs modellek eltérősége sokkal dominánsabban megjelennek. Ez utóbbi abból látható – akár a grafikon alapján –, hogy óriási skálázási különbségek mérhetők egy rendszer esetén eltérő reifikációs modellek esetén.

Nagyobb gráfok esetén tárhely szempontjából az Azure Cosmos DB skálázódik a legrosszabbul. Ez nem csak annak köszönhető, hogy általánosított standard modell eredménye kiugróan rossz – az egyedüli 4 fölötti érték, hanem annak is, hogy a legjobban skálázódó standard modell is 3,58-as skála faktorral rendelkezik, ami a rendszer egyetlen olyan értéke, aminél a többi rendszer esetén mértem rosszabbat.

A „kisebb” gráfokkal ellentétben nagy adathalmaznál a Blazegraph már nem teljesít olyan jól: 3,24-es átlag skála faktorával a második legrosszabb rendszer. A „kisebb” méret esetén tapasztalt nagyobb különbségek lényegesen nagyobbakká nőttek, nem csak a szélsőértékek, hanem az egyes reprezentációk teljesítménye között is. Továbbra is az általánosított standard modell skálázódik Blazegraph-on a legjobban, tovább nagyobb méretek esetén az RDR megközelítés a legrosszabbak közé esik vissza a 3,54-es skála faktorával.

Az egyes reprezentációk között a legnagyobb különbséget a TigerGraph esetén mértem. Érdekesség, hogy az ennél a rendszernél „kisebb” méreteknél legrosszabban szereplő általánosított standard modell nagy méretekben messze a legjobb eredményt érte el. Ezen kívül azonban az n-ary modell összesített eredményt tekintve is az egyik legrosszabb, de az éltulajdonság és a standard modellek is meglehetősen rosszul skálázódnak.

Az Orient DB a TigerGraph-fal nagyon hasonló módon skálázódik, tehát ennél a rendszernél is nagy különbségek vannak a különböző modellek skálázódásai között. A legjobb eredményt az Orient DB-nél (is) az általánosított standard modell, valamint az ezzel közel azonos eredményt elérő singleton tulajdonság modell adta 2,6 körüli értékkel. Ennél a rendszernél is volt egy nagyobb helycsere, ugyanis a „kisebb” méreten legjobb éltulajdonság modell nagyobb halmazon már a legrosszabbul teljesít.

Nagyobb méretek esetén a 2,63-as mért skála faktorával a Neo4j lett a térhely igényt tekintve második legjobban skálázódó gráfadatbázis, habár ezen rendszer esetén is nagy különbségek tapasztalhatók az egyes modellek között. Ez ennél a rendszernél is legjobb eredménnyel rendelkező általánosított standard modell a teljes mérést tekintve a legjobb eredménnyel rendelkezik, ennél nem sokkal rosszabb a singleton tulajdonság és a standard modell skálázódása sem. Az n-ary modell már lényegesen rosszabb módon teljesített, de a legrosszabb skála faktor itt is az éltulajdonság modellhez tartozik: 3,26.

A legjobb átlagos eredményt a JanusGraph érte el – 2,59-es skálázási tényezőt. Az éltulajdonság modell ennél a rendszernél is kiugróan gyenge. A nagyon hasonló teljesítményt nyújtó (általánosított standard, n-ary) és (singleton tulajdonság, standatd) modell párosok teljesítménye között csak kisebb különbség van, de az előbbi páros elemei a jobban skálázódók.

Teljesítmény

Ez eddig bemutatott skálázódási paraméterek elsősorban a kapacitás tervezéshez és az üzemeltetéshez nyújtanak segítséget, a felhasználás szempontjából azonban valószínűleg a legfontosabb tényező, hogy válaszidő, azaz a rendszer kifelé mutatott teljesítménye hogyan változik a gráf méretének növekedésével. A betöltési időhöz és a tárhelyigényhez hasonlóan az átlagos válaszidő relatív változását is két különböző méret változás esetén vizsgáltam, az 5.8 ábra a rendszerek skálázódását kisebb gráfokon írja le, míg az 5.9 ábra nagy méretű adathalmazok esetén jellemzi a vizsgált adatbázis-kezelőket.

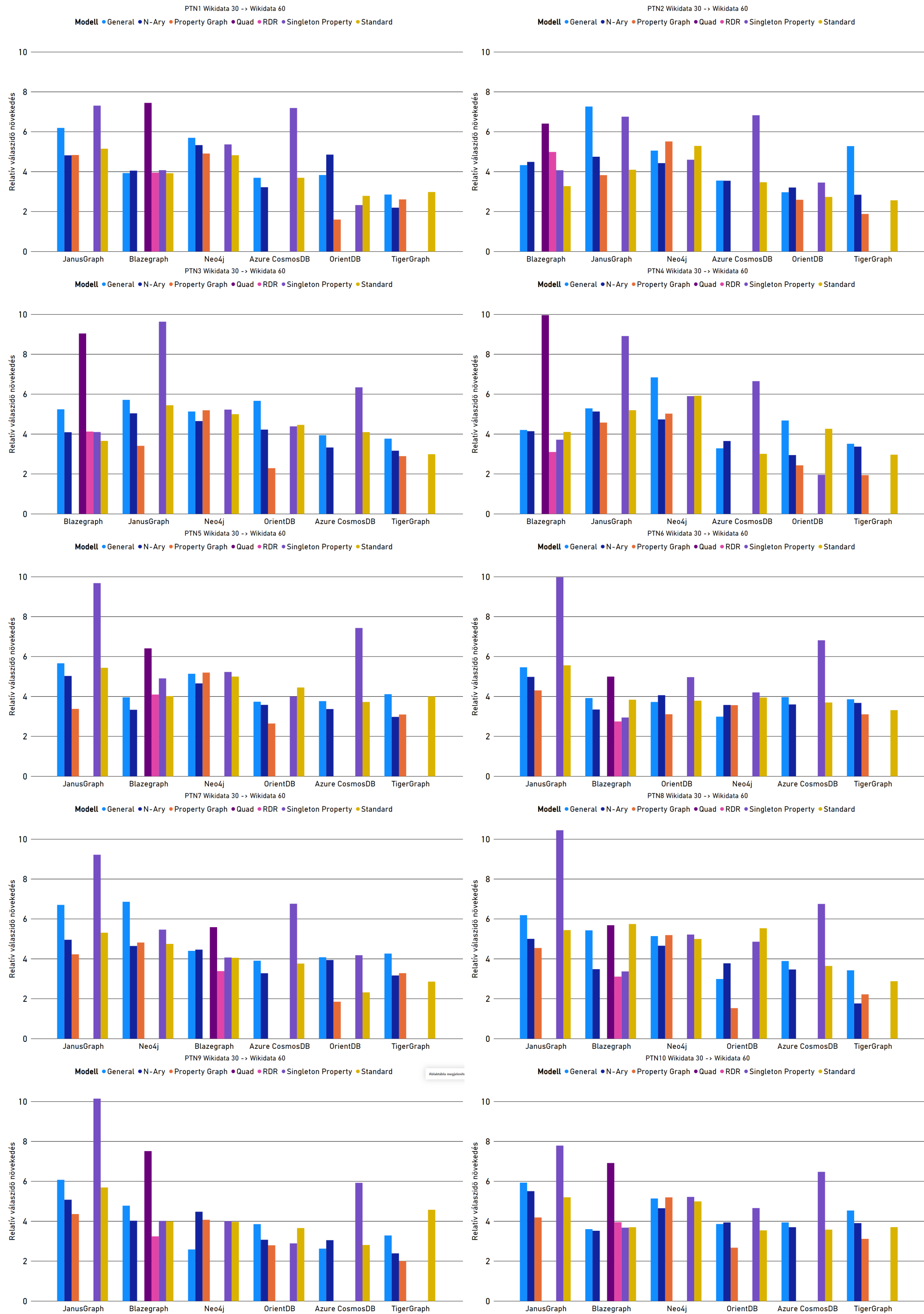
Az 5.8 és az 5.9 ábra is a rendszerek relatív válaszidő növekedését írja le, azonban amíg előbbi a növekedést a Wikidata60 és a Wikidata30 adathalmazokon mért teljesítmény alapján mutatja be, utóbbi a Wikidata100-on mért válaszidők arányát mutatja a Wikidata60-on mértekhez képest. A relatív jellegből adódóan a kisebb érték, jobb teljesítmény skálázódást jelent. A gráf mérete mindkét esetben túlságosan is sokat nőtt ahhoz, hogy a lekérdezések kiszolgálási ideje ne változzon lényegesen, így az 1-hez viszonylag közeli értékek akkor fordultak csak elő, ha a rendszer a viszonyított mindkét méreten timeout-okra futott.

A két ábrán látható grafikonokat összehasonlítva több alapvető fontosságú jellegzetesség is látható. Az egyik, hogy a reifikációs modell megválasztásának óriási hatása lehet a rendszer skálázódására – szélsőséges esetben háromszoros különbséget is tapasztaltam ugyanazon rendszer legjobban, és legrosszabban skálázódó reprezentációja esetén. A másik, hogy az egyes logikai modellek eltérő módon teljesítenek a különböző lekérdezési mintákon, azaz az adatbázis-kezelők teljesítményének skálázási faktora nem egy skalár érték, hanem sokkal inkább egy felhasználási munkafolyamattól (is) függő függvény. A harmadik, hogy ugyan van néhány lényeges eltérés a két méreten tapasztalt eredmény között, mégis alapvetően hasonló teljesítmény mutatnak a rendszerek – nem számítva a timeout miatti torzításokat.

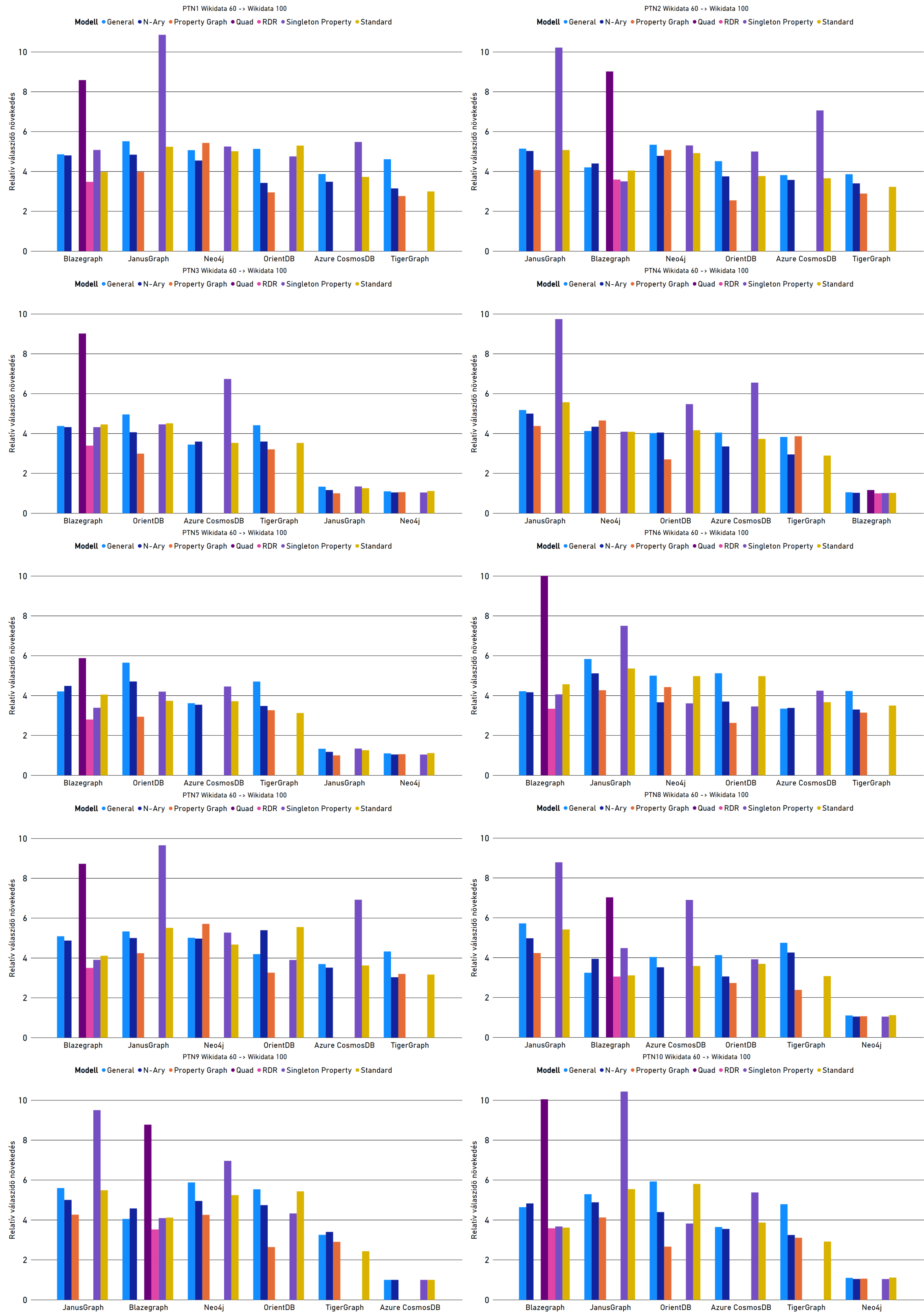
Mindkét méret esetén, szinte az összes lekérdezési minta eredménye alapján, a rendszereket skálázódási képességük alapján két csoportba lehet sorolni. Az egyik csoportban a JanusGraph és a Neo4j van, ezek voltak a rosszabban skálázódó rendszerek. A másikban a jobban skálázódó Azure Cosmos DB, Blazegraph, Orient DB és TigerGraph adatbázisok vannak.

Ugyan az ábrák alapján könnyen látható, hogy szinte az összes lekérdezési minta esetén a JanusGraph teljesített átlagosan a legrosszabban, a fennmaradó esetekben pedig a második legrosszabb volt. Ez a kiugróan rossz teljesítmény a singleton tulajdonság modellnek köszönhető, ami mindkét méret majdnem minden lekérdezése esetén kétszer nagyobb teljesítmény romlást mutatott. Ha azonban a lényegesen kiugró eredményeket nem vesszük figyelembe, akkor az összesített eredmények alapján már „csak” a második legrosszabban skálázódó rendszer.

Habár a Neo4j esetén nem tapasztaltam kiugróan rossz eredményt egyik reprezentációnál sem, továbbá egyik logikai modell használata mellett sem teljesített a legrosszabbul a Neo4j – igaz a legjobban sem, összességében mégis ez a rendszer skálázódik a legrosszabban, ha a többi rendszer kiugróan rossz eredményeitől eltekintünk – bár az Azure Cosmos DB még azzal együtt is gyorsabb az átlagokat nézve. Bár kis különbség van a reifikációs modellek skálázódása között, mindegyik a legrosszabbak közé sorolható. A rossz teljesít-



5.8. ábra. A vizsgált gráfadatbázisok relatív válaszidő növekedése a Wikidata30 - Wikidata60 közötti gráfmérettartományban, különböző reifikációs modellekben.



5.9. ábra. A vizsgált gráfadatbázisok relatív válaszidő növekedése nagy gráf méret tartományban, különböző reifikációs modellekben.

mény alól érdekes kivételt jelent a PTN6 esete, ahol nem csak viszonylag gyors a rendszer, de mindkét mért gráf méret esetén skálázódásban is a legjobbak között van.

A Blazegraph esetén is van egy mindkét gráf méreten, valamennyi lekérdezési mintán a többinél lényegesen rosszabban skálázódó reprezentáció, a quad modell. Ha ettől a modelltől eltekintünk – hiszen mind skálázódásban, mind teljesítményeben a legrosszabb –, a Blazegraph a többi rendszer eredményét tekintve átlagosan skálázódik. A rendszer legjobban az RDR reifikációs modellt használva skálázódik „kisebb" és nagy gráf méret esetén is. A modell eredményei nem csak a Blazegraph-ot nézve, hanem globálisan is legjobbak között vannak, bizonyos lekérdezési minta-gráf méret párosok .

Az utolsó kiugróan rossz eredményt elérő konfiguráció az Azure Cosmos DB és singleton tulajdonság modell párosa volt. A lekérdezési mintától függően volt, hogy kis különbséggel, de volt olyan is, hogy közel kétszeres eredménnyel maradt el második legrosszabb modell mögött. A legtöbb esetben az n-ary modell volt a legjobb, de viszonylag kis különbségek mérhetők az egyes modellek teljesítményének skálázódása között. A rendszer érdekessége, hogy abszolút teljesítményt nézve az egyik leglassabb volt, a teljesítmény skálázódását tekintve azonban a második legjobb, ha a singleton tulajdonság modell eredményeit nem vesszük figyelembe.

Az átlagos eredményeket tekintve a második legjobban skálázódó rendszer az Orient DB – a kiugró értékeket figyelmen kívül hagyva is a harmadik. A rendszer érdekessége, hogy az éltulajdonság modell esetén nagyon jól skálázódik, míg az összes többi reprezentáció csak átlagosan. Az előbbi hatékonyságát jól mutatja, hogy mindkét gráf méret szinte minden lekérdezés mintáján ez érte el a legjobban eredményt, nem csak az Orient DB-t, hanem az összes rendszert nézve is.

Pusztán az átlagot tekintve a TigerGraph meglehetősen nagy különbséggel a legjobban skálázódó rendszer minden gráf méret minden lekérdezési mintája esetén, tehát nem csak abszolút teljesítményt nézve a leggyorsabb rendszer. Részben abból kifolyólag, hogy nincsenek olyan nagy különbségek az egyes modellek között, mint az Orient DB esetén, nincs egy dominánsan legjobb reprezentáció, ugyanis az esetek nagyobb részében az éltulajdonság modell érte el a legjobb eredményt, de a standard modell is többször a legjobb értékkel rendelkezett.

Összességében tehát az látható, hogy az legtöbb rendszer esetén lényegesen befolyásolhatja a skálázódást a választott reprezentációs modell. Ezen mérési eredmények alapján azt lehet mondani, hogy nagy méretű gráfok esetén a vizsgált konfigurációk között az Orient DB és az éltulajdonság modell kombinációja skálázódik a legjobban jelenleg a teljesítmény alapján, ami ráadásul pusztán a válaszidők nagysága alapján is a legjobbak között van.

6. fejezet

Összefoglalás, konklúzió

A dolgozat elkészítése során továbbfejlesztettem, illetve lényegesen kibővítettem az előzmény munkámban létrehozott mérési keretrendszert. Ezen folyamat során általánosításra került a köztes formátum definíciója, az erre épülő meglévő komponenseket ennek megfelelően módosítottam. Továbbá lecseréltem a korábbi mesterséges adathalmaz generátor réteget, hogy a mérések alapjául most már a Wikidata szolgál egy Barabási-Albert-modell alapján szintetizált gráf helyett.

Az egyes rendszerek valóélet-beli teljesítmények vizsgálatához több száz valós lekérdezés alapján összeállítottam egy mérési lekérdezés készletet. Ehhez többféle módszert is kipróbáltam, végül egy gráfminta alapján klaszterező eljárás során kapott eredményeket használtam a konkrét lekérdezések megállapításához. Ezáltal a korábbi rendszer mesterséges lekérdezés generátor komponensét valóélet-beli forgatókönyvre alakítottam át.

A mérési eszközkészlet ezeken kívül lényeges kibővítésre is került. A korábban vizsgált Titan DB és komponensei helyét átvette JanusGraph és hozzá kapcsolódóan implementált új modulok. Valamennyi már létező rendszer esetén is új logikai reprezentációk kerültek bevezetésre, például a singleton tulajdonság modell és általánosított standard modell minden korábbi rendszer esetén, míg a quad és az RDR a Blazegraph esetén. Ezeken felül kettő eddig nem vizsgált rendszer is bevonásra került, a TigerGraph és az Orient DB.

Minden rendszer és reprezentáció esetén megvizsgáltam az adatok betöltéséhez szükséges időt. Az itt kapott eredmények azt mutatták, hogy a reifikációs modell megválasztásának viszonylag kis hatása van a betöltés időre, ez inkább rendszer szintű jellemző. Az adatbázisok betöltési idejei között több nagyságrendű különbségeket is tapasztaltam, a legjobb eredményt ezen a téren Neo4j és TigerGraph érte el, míg kimagaslóan a legrosszabb az Azure Cosmos DB volt.

A betöltési időn túl megvizsgáltam az adatok tárolásához szükséges tárhely igényt is. Ezen szempont esetén már nagyobb különbségek voltak az egyes reifikációs modellek között egy rendszer esetén is, de továbbra is inkább rendszer jellemzőnek lehet tekinteni. Az eredmények alapján az adódott, hogy a multimodális rendszereknek majdnem kétszer nagyobb tárhelyre van szüksége ugyanazon adatok ugyanolyan logikai reprezentációjú tárolására. A kapott adatok jól megmutatták a magas rendelkezésre állás tárhely költségeit az Azure Cosmos DB esetén, illetve a natív gráf tárolási eljárások előnyeit az ebből a szempontból is legjobb eredményt elérő Neo4j és TigerGraph esetén.

Ezt követően bemutattam a vizsgált adatbázis-kezelők tényleges teljesítményét az egyes lekérdezési mintákon. Ez alapján azt lehet mondani, hogy a válaszidőre meglehetősen nagy hatással van a logikai modell választás, és a lekérdezés különböző jellemzői is, hiszen például a TigerGraph-ot leszítva az összes rendszer lényegesen lelassult, ha aggregációt, vagy csoportosító műveletet kellett végeznie. Teljesítményt tekintve a JanusGraph, az Azure Cosmos DB és a Neo4j átlagosan messze a legrosszabb eredményt érték el. Ezeknél

ugyan a legtöbb esetben jóval gyorsabb volt a quad modellt leszámítva a Blazegraph, de két leggyorsabb rendszer, az Orient DB és a TigerGraph még ehhez lépest is lényegesen jobb válaszidőket adott.

A rendszerek összehasonlítását a skálázhatóság és a skálázódási képességek ismertetésével fejeztem be. A skálázhatóság vizsgálata során bemutattam, hogy melyik rendszer esetén milyen lehetőségek állnak rendelkezésre a horizontális skálázásra, illetve, hogy ehhez milyen támogatást nyújtanak, kitérve a replikáció és a sharding kérdéskörére.

A skálázási tulajdonság mértékének megállapításához az egyes méréseket három különböző méreten is elvégeztem, így egy viszonylag átfogó képet kaptam a rendszerekről a vizsgált nemfunkcionális paraméterek esetén. Ennek során láthatóvá vált, hogy a használt logikai modell kulcsszerepet játszik a jellemzők skálázódása szempontjából, illetve az is, hogy eltérő méret tartományban másként skálázódnak a rendszerek. A dolgozat eredményei arra a fontos következtetésre is elvezettek, hogy az adatbázis-kezelő rendszerek teljesítményének skálázódása erősen függ a lekérdezések jellegétől is.

Valamennyi eredményt figyelembe véve az látszik, hogy az Orient DB és a TigerGraph a teljesítmény területen kiemelkedik a többi rendszer közül. Mind az adatbetöltés, mind a tárhely igény, mind a teljesítmény szempontjából a TigerGraph teljesített a legjobban, azonban ezt csak meglehetősen rugalmatlan adatkezeléssel párosítva tudta elérni. A kapott adatok alapján azt lehet mondani, hogy amennyiben a vizsgálandó gráf elfér a memóriában, és nem várható változás az adatsémát illetően, akkor a TigerGraph tűnik jelenleg a legjobb ingyenes választásnak éltulajdonság modell használatával. Ha azonban memória méret feletti gráfot kell kezelni, vagy várhatóan szükség lesz adatséma változásra, vagy ha felmerül az igény az adatok elérésére külső alkalmazásból, akkor a nagyon jó skálázódás, a minimálisan rosszabb összteljesítmény és a könnyű integrálhatóság miatt az Orient DB az éltulajdonság modellel jobb választás lehet.

7. fejezet

Továbbfejlesztési lehetőségek

Habár a dolgozat a korábbi munkáim továbbfejlesztésének tekinthető, ez nem jelenti, hogy a bemutatott eredmények ne nyújthatnának alapot további kutatások kiindulásához. A számos továbbgondolási lehetőség közül a legegyszerűbb, ha valamelyik mérési paraméter bővítésre kerül.

Jelen kutatásomban 6 különböző, általam választott gráfadatbázis implementáció teljesítményét vizsgáltam meg, azonban ezek köre tetszőlegesen bővíthető. Ez lehetőséget biztosít arra, hogy akár további gráf-alapú rendszerek – például a Dgraph, az Amazon Neptune vagy a HyperGraphDB –, illetve multimodális rendszerek – például az ArangoDB vagy a Microsoft SQL Server 2019 – kerüljenek bevonásra. Ennek eredményeként az új adatbázisok új, alternatív reifikációs modellezési lehetőségek vizsgálatát is lehetővé teszik – például hipergráf-alapú reprezentációkét.

A mérések alapjául szolgáló két további paraméter, az adathalmaz és a lekérdezések változtatása is egy lehetséges fejlesztési lehetőség. A Wikidata lecserélése más valódi adathalmazra, vagy egy újabb, még a jelenleginél is lényegesen több adatot tartalmazó verzióra megnyitja a lehetőséget az eredmények általánosítására, illetve a skálázódás szélesebb körül vizsgálatára. A lekérdezések halmazának módosítása vagy bővítése pedig a rendszerek más aspektusú teljesítmény jellemzőire is fényt deríthetnek.

Dolgozatomban a skálázhatóság vizsgálatakor szinte kizárólag a horizontális skálázódással foglalkoztam. Az üzleti életben azonban gyakran felmerülő kérdés, hogy az egyszerűbb vertikális skálázás, vagy bonyolultabb, tipikusan a rá épülő alkalmazásokat is érintő horizontális skálázódás az ideálisabb. Ezen kérdés megválaszolása megnyitja a lehetőséget a mérések eltérő hardver konfiguráción történő elvégzése előtt. Ennek relevanciáját jól bizonyítja például az, hogy a Blazegraph 4 GiB és 24 GiB memória korlát mellett is lényegileg azonos teljesítményt nyújtott.

Jóval magasabb szintű kutatás ennek eredményeire építve egy olyan rendszert elkészíteni, ami lekérdezéseket fogad, és azok jellegét és gráf méretét is figyelembe véve a megfelelő rendszer számára továbbítja azt, vagy komplexebb esetben csak annak egy allekérdezését.

Irodalomjegyzék

[1] 5 trends appear on the gartner hype cycle for emerging technologies, 2019. https://www.gartner.com/smarterwithgartner/5-trends-appear-on-the-gartner-hype-cycle-for-emerging-technologies-2019/. Hozzáférés: 2019-10-23.

[2] Antlr. https://www.antlr.org/. Hozzáférés: 2019-10-23.

[3] Aws marketplace: Orientdb community edition 2.2.31. https://aws.amazon.com/marketplace/pp/Orient-Technologies-LTD-OrientDB-Community-Edition/B00VKPTLU0. Hozzáférés: 2019-10-23.

[4] Azure cosmos db. https://azure.microsoft.com/hu-hu/services/cosmos-db/. Hozzáférés: 2019-10-23.

[5] Azure cosmos db service quotas. https://docs.microsoft.com/en-us/azure/cosmos-db/concepts-limits. Hozzáférés: 2019-10-23.

[6] Blazegraph. https://dbdb.io/db/blazegraph. Hozzáférés: 2019-10-23.

[7] Blazegraph parameter tuning. https://wiki.blazegraph.com/wiki/index.php/IOOptimization#File_System_Cache. Hozzáférés: 2019-10-23.

[8] Blazegraph tinkerpop3 implementation. https://github.com/blazegraph/tinkerpop3. Hozzáférés: 2019-10-23.

[9] Cluster scale-out. https://en-doc.graphtiger.com/admin/admin-guide/installation-and-configuration/offline-node-expansion. Hozzáférés: 2019-10-23.

[10] E. F. Codd: Data models in database management. 1981. 74. sz., 112–114. p.

[11] Compressed storage. https://neo4j.com/docs/operations-manual/3.5/performance/property-compression/. Hozzáférés: 2019-10-23.

[12] Configuring blazegraph. https://wiki.blazegraph.com/wiki/index.php/Configuring_Blazegraph#Quads_Mode. Hozzáférés: 2019-10-23.

[13] Configuring blazegraph. https://wiki.blazegraph.com/wiki/index.php/Configuring_Blazegraph. Hozzáférés: 2019-10-23.

[14] Consistency levels in azure cosmos db. https://docs.microsoft.com/en-us/azure/cosmos-db/consistency-levels. Hozzáférés: 2019-10-23.

[15] Data types. https://en-doc.graphtiger.com/v/2.3/dev/gsql-ref/querying/data-types. Hozzáférés: 2019-10-23.

[16] Db-engines ranking - trend of graph dbms popularity. https://db-engines.com/en/ranking_trend/graph+dbms. Hozzáférés: 2019-10-23.

[17] Dbpedia sparql benchmark. http://aksw.org/Projects/DBPSB.html. Hozzáférés: 2019-10-23.

[18] Alin Deutsch – Yu Xu – Mingxi Wu – Victor Lee: Tigergraph: A native mpp graph database, 2019.

[19] Does cassandra support sharding? https://stackoverflow.com/questions/16428008/does-cassandra-support-sharding. Hozzáférés: 2019-10-23.

[20] Download tigergraph. https://www.tigergraph.com/download/. Hozzáférés: 2019-10-23.

[21] Downloading janusgraph and running the gremlin console. https://docs.janusgraph.org/#downloading-janusgraph-and-running-the-gremlin-console. Hozzáférés: 2019-10-23.

[22] Songyun Duan – Anastasios Kementsietsidis – Kavitha Srinivas – Octavian Udrea: Apples and oranges: A comparison of RDF benchmarks and real RDF datasets. 2011. 01, 145–156. p.

[23] Orri Erling – Alex Averbuch – Josep Larriba-Pey – Hassan Chafi – Andrey Gubichev – Arnau Prat – Minh-Duc Pham – Peter Boncz: The LDBC social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15 konferenciasorozat. New York, NY, USA, 2015, ACM, 619–630. p. ISBN 978-1-4503-2758-9.
URL http://doi.acm.org/10.1145/2723372.2742786. 12 p.

[24] Etl. https://orientdb.org/docs/3.0.x/etl/ETL-Introduction.html. Hozzáférés: 2019-10-23.

[25] Explanation of storage compression options in orientdb needed. https://stackoverflow.com/questions/25108243/explanation-of-storage-compression-options-in-orientdb-needed. Hozzáférés: 2019-10-23.

[26] Global data distribution with azure cosmos db - under the hood. https://docs.microsoft.com/en-us/azure/cosmos-db/global-dist-under-the-hood. Hozzáférés: 2019-10-23.

[27] Graph database benchmark. https://www.tigergraph.com/benchmark/. Hozzáférés: 2019-10-23.

[28] Graph database features in sql server 2019 – part 1. https://www.sqlshack.com/graph-database-features-in-sql-server-2019-part-1/. Hozzáférés: 2019-10-23.

[29] Ha cluster configuration. https://en-doc.graphtiger.com/admin/admin-guide/installation-and-configuration/ha-cluster. Hozzáférés: 2019-10-23.

[30] Daniel Hernández: Querying Wikidata: Comparing SPARQL, Relational and Graph Databases — Complementary Documentation. 2016. 5.
URL https://figshare.com/articles/Querying_Wikidata_Comparing_SPARQL_Relational_and_Graph_Databases/3219217.

[31] High availability with azure cosmos db. https://docs.microsoft.com/en-us/azure/cosmos-db/high-availability. Hozzáférés: 2019-10-23.

[32] How do i change api in cosmosdb? https://stackoverflow.com/questions/57142665/how-do-i-change-api-in-cosmosdb. Hozzáférés: 2019-10-23.

[33] How to import and query 4-column n-quad quadruples into blazegraph? https://stackoverflow.com/questions/38112473/how-to-import-and-query-4-column-n-quad-quadruples-into-blazegraph. Hozzáférés: 2019-10-23.

[34] How-to: Run neo4j in docker. https://neo4j.com/developer/docker-run-neo4j/. Hozzáférés: 2019-10-23.

[35] Indexes. http://www.orientdb.com/docs/last/Indexes.html. Hozzáférés: 2019-10-23.

[36] Installing rdf4j server and rdf4j workbench. https://rdf4j.eclipse.org/documentation/server-workbench-console/. Hozzáférés: 2019-10-23.

[37] Introduction. https://neo4j.com/docs/operations-manual/current/clustering/introduction/#causal-clustering-introduction-operational. Hozzáférés: 2019-10-23.

[38] Introduction to polyglot persistence: Using different data storage technologies for varying data storage needs. http://www.informit.com/articles/article.aspx?p=1930511. Hozzáférés: 2019-10-23.

[39] Alexandru Iosup – Tim Hegeman – Wing Lung Ngai – Stijn Heldens – Arnau Prat-Pérez – Thomas Manhardto – Hassan Chafio – Mihai Capotă – Narayanan Sundaram – Michael Anderson – Ilie Gabriel Tănase – Yinglong Xia – Lifeng Nai – Peter Boncz: LDBC graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms. *Proc. VLDB Endow.*, 9. évf. (2016. szeptember) 13. sz., 1317–1328. p. ISSN 2150-8097.
URL https://doi.org/10.14778/3007263.3007270. 12 p.

[40] Janusgraph. https://janusgraph.org/. Hozzáférés: 2019-10-23.

[41] janusgraph/janusgraph - docker hub. https://hub.docker.com/r/janusgraph/janusgraph. Hozzáférés: 2019-10-23.

[42] Venelin Kotsev – Nikos Minadakis – Vassilis Papakonstantinou – Orri Erling – Irini Fundulaki – Atanas Kiryakov: Benchmarking RDF query engines: The LDBC semantic publishing benchmark. In *BLINK@ ISWC* (konferenciaanyag). 2016.

[43] Königsbergi hidak problémája. https://hu.wikipedia.org/wiki/Königsbergi_hidak_problémája. Hozzáférés: 2019-10-23.

[44] A letter regarding native graph databases. https://www.datastax.com/blog/2013/11/letter-regarding-native-graph-databases. Hozzáférés: 2019-10-23.

[45] Litmus-benchmark-suite/sparql-to-gremlin. https://github.com/LITMUS-Benchmark-Suite/sparql-to-gremlin. Hozzáférés: 2019-10-23.

[46] Memory storage. http://www.orientdb.com/docs/last/Memory-storage.html. Hozzáférés: 2019-10-23.

[47] Memória-optimalizált virtuális gépek méretei. https://docs.microsoft.com/hu-hu/azure/virtual-machines/windows/sizes-memory#esv3-series. Hozzáférés: 2019-10-23.

[48] Mohamed Morsey – Jens Lehmann – Sören Auer – Axel-Cyrille Ngonga Ngomo: Dbpedia SPARQL benchmark – performance assessment with real queries on real data. In Lora Aroyo – Chris Welty – Harith Alani – Jamie Taylor – Abraham Bernstein – Lalana Kagal – Natasha Noy – Eva Blomqvist (szerk.): *The Semantic Web – ISWC 2011* (konferenciaanyag). Berlin, Heidelberg, 2011, Springer Berlin Heidelberg, 454–469. p. ISBN 978-3-642-25073-6.

[49] Multi-model. https://orientdb.com/docs/last/Tutorial-Document-and-graph-model.html. Hozzáférés: 2019-10-23.

[50] Multiset. https://en.wikipedia.org/wiki/Multiset. Hozzáférés: 2019-10-23.

[51] Neo4j cloud services. https://neo4j.com/neo4j-cloud-services/. Hozzáférés: 2019-10-23.

[52] Neo4j for google cloud. https://neo4j.com/neo4j-for-google-cloud/. Hozzáférés: 2019-10-23.

[53] Nullpointerexception when launching blazegraph server. https://github.com/blazegraph/database/issues/89. Hozzáférés: 2019-10-23.

[54] orientdb - docker hub. https://hub.docker.com/_/orientdb. Hozzáférés: 2019-10-23.

[55] Orientdb. https://dbdb.io/db/orientdb. Hozzáférés: 2019-10-23.

[56] Orientdb + microsoft azure = simplified scale and zero-configuration. https://orientdb.com/cloud/orientdb-microsoft-azure/. Hozzáférés: 2019-10-23.

[57] Orientdb vs neo4j. https://orientdb.com/orientdb-vs-neo4j/. Hozzáférés: 2019-10-23.

[58] Anil Pacaci – Alice Zhou – Jimmy Lin – M. Tamer Özsu: Do we need specialized graph databases?: Benchmarking real-time social networking applications. 2017. 05, 1–7. p.

[59] Zhengyu Pan – Tao Zhu – Hong Liu – Huansheng Ning: A survey of RDF management technologies and benchmark datasets. *Journal of Ambient Intelligence and Humanized Computing*, 9. évf. (2018. Oct) 5. sz., 1693–1704. p. ISSN 1868-5145. URL https://doi.org/10.1007/s12652-018-0876-2.

[60] Planner hints and the using keyword. https://neo4j.com/docs/cypher-manual/current/query-tuning/using/. Hozzáférés: 2019-10-23.

[61] Plocal storage. http://www.orientdb.com/docs/last/Paginated-Local-Storage.html. Hozzáférés: 2019-10-23.

[62] Powers of ten – part i. https://www.datastax.com/blog/2014/05/powers-ten-part-i. Hozzáférés: 2019-10-23.

[63] Rdf. https://www.w3.org/RDF/. Hozzáférés: 2019-10-23.

[64] Rdf 1.1 n-quads. https://www.w3.org/TR/n-quads/. Hozzáférés: 2019-10-23.

[65] Rdf* and sparql*. http://www.snee.com/bobdc.blog/2018/05/rdf-and-sparql.html. Hozzáférés: 2019-10-23.

[66] Regional presence with azure cosmos db. https://docs.microsoft.com/en-us/azure/cosmos-db/regional-presence. Hozzáférés: 2019-10-23.

[67] Reification done right. https://wiki.blazegraph.com/wiki/index.php/Reification_Done_Right. Hozzáférés: 2019-10-23.

[68] Reification vocabulary. https://www.w3.org/TR/rdf-schema/#ch_reificationvocab. Hozzáférés: 2019-10-23.

[69] Replication. https://orientdb.com/docs/2.2.x/Replication.html. Hozzáférés: 2019-10-23.

[70] Request units in azure cosmos db. https://docs.microsoft.com/en-us/azure/cosmos-db/request-units. Hozzáférés: 2019-10-23.

[71] Ian Robinson – Jim Webber – Emil Eifrem: *Graph Databases*. 2015, O'Reilly & Associates.

[72] Marko A. Rodriguez: Constructions from dots and lines. 2010. 6. sz., 35–41. p.

[73] Schema. https://orientdb.org/docs/3.0.x/java/Object-DB-Schema.html. Hozzáférés: 2019-10-23.

[74] Schema indexes. https://neo4j.com/docs/operations-manual/3.5/performance/index-configuration/schema-indexes/. Hozzáférés: 2019-10-23.

[75] Sharding. https://orientdb.com/docs/2.2.x/Distributed-Sharding.html. Hozzáférés: 2019-10-23.

[76] Leslie Sikos: *Mastering Structured Data on the Semantic Web: From HTML5 Microdata to Linked Open Data*. 2015, Apress.

[77] Sparql query language for rdf. https://www.w3.org/TR/rdf-sparql-query/. Hozzáférés: 2019-10-23.

[78] Storing documents and graph in the same collection of documentdb. https://stackoverflow.com/questions/46647373/storing-documents-and-graph-in-the-same-collection-of-documentdb. Hozzáférés: 2019-10-23.

[79] Gajdos Sándor: *Adatbázisok*. 2015, A-SzínVonal. ISBN 9789633131954.

[80] A technical overview of azure cosmos db. https://azure.microsoft.com/hu-hu/blog/a-technical-overview-of-azure-cosmos-db/. Hozzáférés: 2019-10-23.

[81] Kovács Tibor: Eszközkészlet szemantikus adathalmazok külféle tárolási megoldásainak összehasonlítására, 2017.

[82] Kovács Tibor – Simon Gábor: Storing large-scale knowledge bases in cloud-native graph databases: Are they up for the task? *Proceedings of the Automation and Applied Computer Science Workshop 2019*, 2019.

[83] Kovács Tibor – Simon Gábor – Mezei Gergely: Benchmarking graph database backends—what works well with wikidata? *Acta Cybernetica*, 2019. ISSN HUISSN: 0324-721X.

[84] Tigergraph cloud. https://www.tigergraph.com/cloud/. Hozzáférés: 2019-10-23.

[85] Tigergraph in cloud marketplaces. https://www.tigergraph.com/cloud-marketplaces/. Hozzáférés: 2019-10-23.

[86] tinkerpop3/src/main/java/com/blazegraph/gremlin/structure/blazegraph.java. https://github.com/blazegraph/tinkerpop3/blob/master/src/main/java/com/blazegraph/gremlin/structure/BlazeGraph.java. Hozzáférés: 2019-10-23.

[87] Values and types. https://neo4j.com/docs/cypher-manual/current/syntax/values/. Hozzáférés: 2019-10-23.

[88] Virtuosomanager class. http://www.dotnetrdf.org/api/html/T_VDS_RDF_Storage_VirtuosoManager.htm. Hozzáférés: 2019-10-23.

[89] Welcome to ldbc website. http://ldbcouncil.org/. Hozzáférés: 2019-10-23.

[90] What is scalability? - definition from whatis.com. https://searchdatacenter.techtarget.com/definition/scalability. Hozzáférés: 2019-10-23.

[91] Why a multi-model database? https://orientdb.com/multi-model-database/. Hozzáférés: 2019-10-23.

[92] Wikidata dashboards / wikidata datamodel statements. https://grafana.wikimedia.org/d/000000175/wikidata-datamodel-statements?refresh=30m&orgId=1https://grafana.wikimedia.org/d/000000167/wikidata-datamodel?refresh=30m&orgId=1. Hozzáférés: 2019-10-23.

[93] Wikidata query service. https://query.wikidata.org/. Hozzáférés: 2019-10-23.

[94] Wikidata query service/implementation. https://www.mediawiki.org/wiki/Wikidata_Query_Service/Implementation. Hozzáférés: 2019-10-23.

[95] Wikidata:introduction. https://www.wikidata.org/wiki/Wikidata:Introduction. Hozzáférés: 2019-10-23.