Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

Gráf információs rendszerek összehasonlító teljesítménymérése

Készítette
Antal János Benjamin
Elekes Márton

Konzulens
Marton József Ernő
Szárnyas Gábor

2018.

Tartalomjegyzék

Kivonat

Az elmúlt évtizedben sokféle különböző NoSQL technikát használó adatbázis-kezelő készült. Ezek egyik csoportja a gráfadatbázisoké, melyek lehetővé teszik az adatok gráf formában történő tárolását és lekérdezését. Ez az adatmodell gyakran jobban illeszkedik a sok összefüggést tartalmazó adatok tárolására, mint a relációs modell, és a tömörsége miatt gyakran képes jobb teljesítményt nyújtani. Mindezek ellenére, mivel a relációs adatbázisokat majdnem 50 éve fejlesztik és optimalizálják, jelenleg is nyitott kérdés, hogy szükség van-e specializált gráfadatbázisokra a gráf adatok hatékony feldolgozásához.

Gráflekérdezések megfogalmazására új lekérdező nyelvek jelentek meg, mint például az openCypher. Ezeken a nyelveken gyakran kényelmesebben fogalmazhatunk meg gráflekérdezéseket, mint az SQL-alapú nyelveken. Az üzleti adatok jelentős része azonban jelenleg is hagyományos relációs adatbázisban van tárolva, emiatt ezen adatokat át kell tölteni gráfadatbázisokba, amely éles adatbázisok esetén általában nem megoldható. Célunk, hogy egy olyan megoldást készítsünk, amelyben lehetséges a magas kifejezőerővel rendelkező gráflekérdező nyelveken megfogalmazott lekérdezéseket hatékony relációs lekérdezőmotorokon futtatni anélkül, hogy szükség lenne egyik rendszerből a másikba áttölteni az adatokat. Ennek érdekében olyan fordítót (transpilert) készítettünk, ami képes openCypher lekérdezéseket SQL-re fordítani.

Különböző adatbázis-kezelő rendszerek összehasonlításához elengedhetetlenek a teljesítménymérési specifikációk (benchmarkok). Relációs adatbázisok esetében ezt a szerepet a Transaction Processing Performance Council benchmarkjai töltik be. A gráfadatbázisok relatív kiforratlansága miatt jelenleg kevés benchmark létezik a gráflekérdezések teljesítménymérésére. Mi az LDBC (Linked Data Benchmark Council) Social Network Benchmark fejlesztésébe kapcsolódtunk be, amelynek keretében frissítettük és jelentősen fejlesztettük a meglévő implementációkat, továbbá elkészítettük a SPARQL nyelvű implementációt. Ezek felhasználásával alaposan megvizsgáltuk és részletesen elemeztük az adatbázis kezelőket különböző adatmodellek (relációs, gráf és szemantikus) felhasználásával.

Abstract

In the last decade, a lot of database management systems were developed with different NoSQL techniques. One group of these systems are graph databases, which allow users to store and query their data as graphs. This data model is often a better fit to represent strongly interlinked data sets than the traditional relational model, and its conciseness can lead to better performance. That said, relational databases have been developed and optimized for almost 50 years, and it is an open question whether efficient processing of graph data requires specialized databases.

New query languages, such as openCypher, were developed for querying and processing graph data. These languages usually offer a more intuitive way to express graph queries than SQL-like languages. However, most enterprises still store their data in traditional relational databases, which necessitates loading their data to graph databases. This is often impractical or infeasible for production databases. Our goal is to allow using expressive graph query languages and leverage the performance of existing relational databases while avoiding the overhead of transferring the data between different systems. To this end, we developed a transpiler which can transform openCypher graph queries into SQL.

Comparing the performance of database systems requires standard benchmarks. For relational databases, this is fulfilled by the benchmarks of the Transaction Processing Performance Council. Due to the relative immaturity of graph databases, there is only a limited number of benchmarks available for graph query workloads. We joined the development of the LDBC (Linked Data Benchmark Council) Social Network Benchmark. We reworked and significantly improved existing implementations of the benchmark, and also implemented the queries in the SPARQL language for semantic databases. We performed a thorough evaluation and detailed analysis of database systems using various data models (relational, graph, and semantic).

1. fejezet

Bevezetés

Az informatikában az adatbázis-kezelés területét az elmúlt közel 50 évben a relációs adatmodell dominálta. Mostanra azonban többen felismerték, hogy számos olyan alkalmazási terület van – pl. közösségi hálók, ajánlórendszerek, pénzügyi csalások felderítése – ahol az adatok gráf jellegű tárolása és feldolgozása előnyös lehet. A gráfadatbázisokban használt tulajdonsággráf (property graph) adatmodell hasznos eszköznek bizonyult sokféle probléma modellezésére, így az ezt használó eszközök az elmúlt évtizedben egyre népszerűbbek lettek, pl. a Neo4j gráfadatbázis-kezelő rendszer.[1] Az elmúlt években indult openCypher kezdeményezés célja pedig, hogy – a relációs adatbázisokban alkalmazott SQL nyelv mintájára – szabványos gráflekérdező nyelvet definiáljon.

Mi lehet a gráfadatbázisok sikerének oka? Ezek a rendszerek több előnnyel is rendelkeznek a megszokott relációs adatbázisokhoz képest:

- **Intuitív adatmodell**: Az emberek szeretik úgy modellezni a világot, mint különböző entitások (csúcsok) és közöttük lévő kapcsolatok (élek) sokasága. Mind az entitások, mind a közöttük lévő kapcsolatok rendelkezhetnek különböző tulajdonságokkal, amelyek a tulajdonsággráfokkal egyszerűen kifejezhetőek.
- **Olvashatóság**: Ha megnézünk egy relációs adatbázison futtatott SQL lekérdezést a séma ismerete nélkül, akkor nem magától értetődő, hogy egy attribútum egy tulajdonságot vagy kapcsolatot (idegen kulcs) jelent. A több-több kapcsolatoknál a kapcsolótáblák ezt egyértelműen meghatározzák, azonban a lekérdezés kevésbé olvasható lesz tőlük.
- **Tömörség**: Adott éltípusok mentén történő útkereső lekérdezéseket kifejezetten nehéz leírni SQL-ben és nem is minden SQL-dialektusban támogatottak. Még nehezebb a legrövidebb utat kereső lekérdezések megfogalmazása.

A fentieket figyelembe véve elmondható, hogy sok esetben érthetőbb adathalmazt valamint tömörebb lekérdezéseket eredményez. Az elmúlt években a gráf központú adatmodellezés és a gráflekérdezőnyelvek olyan népszerűvé váltak, hogy egyre gyakrabban merül fel az igény arra, hogy örökölt (legacy) relációs adatbázis-kezelőkben tárolt adatokat gráfként kezeljenek. Jelenleg azonban nem érhető el olyan megoldás, ami ezt az igényt a gazdag gráflekérdező nyelveket támogatva kielégítené.

Dolgozatunkban megvizsgáltuk, hogy mennyire alkalmasak a klasszikus relációs adatbázis-kezelők a gráf jellegű lekérdezések számítására. Az ehhez a szükséges elméleti háttér (2. fejezet) megismerése után kidolgoztunk egy módszert az openCypher nyelven

[1] `https://neo4j.com/`

írt lekérdezések SQL-re történő automatizált leképzésére (3. fejezet). A kiértékeléshez kiválasztottunk egy széleskörben elfogadott, sok nyelvi elemet és teljesítmény-aspektust lefedő teljesítménymérési keretrendszert, az LDBC Social Network Benchmarkot. A benchmark keretrendszerében javítottuk a specifikáció hiányosságait és hibáit, ezek alapján frissítettük a benne található lekérdezéseket és szoftvermodulokat (4. fejezet). Az így rendelkezésre álló implementációk segítségével elkészítettük a különböző adatmodellt, nyelvet és megközelítést alkalmazó rendszerek összehasonlító teljesítménymérését (5. fejezet). Végezetül áttekintettük a kapcsolódó kutatási eredményeket (6. fejezet) és meghatároztuk a további kutatási irányokat (7. fejezet).

2. fejezet

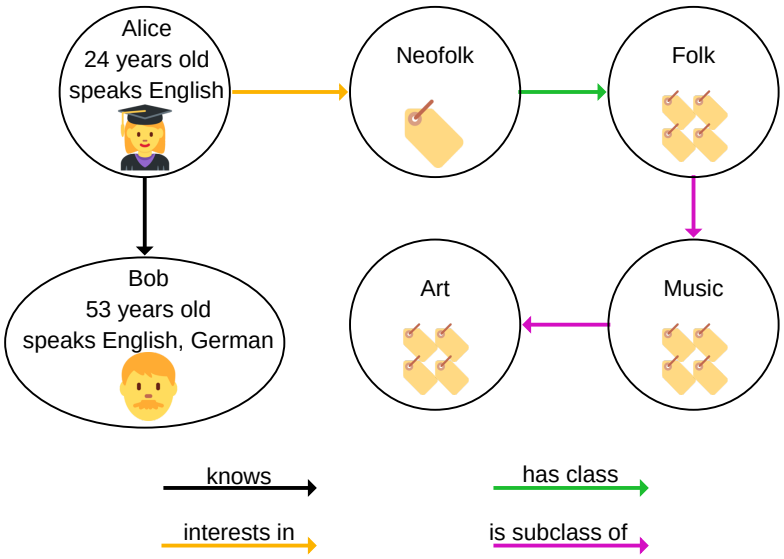
Háttérismeretek

2.1. Esettanulmány

Az ebben a fejezetben bemutatásra kerülő különböző adatmodellek és lekérdezőnyelvek összehasonlítása egy közös, gráf alapú adathalmazon fog történni, így először ezt mutatjuk be. A példa egy közösségi háló lehetséges adatbázisának egy részlete. A 2.1. ábrán látható a példa gráfos ábrázolása, melyről leolvahatóak az alábbi adatok:

- **Bob**: 53 éves, angolul és németül beszél.
- **Alice**: 24 éves, angolul beszél, *érdeklődik* a **Neofolk** iránt. **Alice** *ismeri* **Bob**ot.
- A **Neofolk** *egy* **Népzene**i műfaj.
- A **Népzene** *része* a **Zené**nek, amely *része* a **Művészet**nek.

A példa tehát **csúcsok**ból (csomópontokból), azok tulajdonságaiból és a csúcsok között lévő kapcsolatokból, azaz *élek*ből áll.



2.1. ábra. A példa gráf

2.2. Adatmodellek

Az adatmodellek a példában megismert alkotórészek (csúcsok, élek és tulajdonságok) lehetséges használatának, jellemzőinek szempontjából igen különbözőek. Ezeket a különbségeket mutatja be a 2.1. táblázat. Jól látható, hogy a spektrum egyik vége a matematikában is megszokott irányított gráfok, a másik vége pedig az egyik legújabb adatmodell, a tulajdonsággráf. Az irányított gráfokkal ellentétben a címkézett gráfok csúcsainak már lehetnek típusai, amelyeket a megfelelő címkék csúcshoz rendelésével lehet meghatározni. Ha egy gráf csúcsain kívül az éleit is címkékkel látjuk el, akkor szemantikus gráfot kapunk. A tulajdonsággráf esetében a gráf csúcsainak és éleinek nem csak típusuk, hanem tulajdonságaik is lehetnek. Egy tulajdonság az egyszerű numerikus vagy szöveges adattípusokon túl akár komplexebb típus is lehet, például lista vagy halmaz.

A táblázatban továbbá látható az objektum-orientált modell, amely nem egyértelmű, hogyan kapcsolódik a gráf alapú adatmodellekhez: ha egy címkézett gráf csúcsainak lehetnek tulajdonságai, akkor az megfeleltethető a széles körben használt objektum-orientált modellnek. Fontos megjegyezni, hogy elméletben az objektum-orientált modellben is lehet az éleknek tulajdonsága, de a gyakorlati megvalósítások közül elhanyagolható számú támogatja csupán. A továbbiakban három adatmodell kerül részletes bemutatásra a táblázatból: a szemantikus gráfok, a tulajdonsággráfok és a relációs adatmodell.

adatmodell	adatmodell tulajdonság			
	csúcsok		élek	
	típus	tulajdonság	típus	tulajdonság
irányított gráf	∘	∘	∘	∘
címkézett gráf	•	∘	∘	∘
szemantikus gráf	•	*	•	∘
objektum-orientált modell	•	•	•	∘
tulajdonsággráf	•	•	•	•

2.1. táblázat. Az adatmodellek tulajdonságainak összehasonlítása

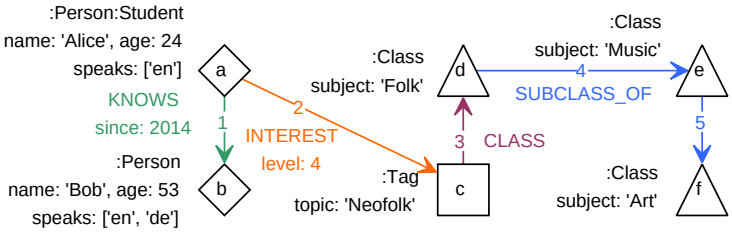
2.2.1. Tulajdonsággráf

A *tulajdonsággráf* (property graph, PG) egy $G = (V, E, st, L, T, lbl, typ, P_v, P_e)$ struktúrával írható le, ahol V a csúcsok (csomópontok) halmaza, E az élek halmaza és $st : E \to V \times V$ függvény határozza meg az élek kiindulási és cél csúcsát [19]. Formálisan a csúcsok típusát címkének nevezzük, az élek esetében pedig formálisan is típusról beszélünk:

- L a címkék halmaza, a $lbl : V \to 2^L$ függvény pedig *a címkék egy halmazát* rendeli a csúcsokhoz.
- T a típusok halmaza, a $typ : E \to T$ függvény pedig *pontosan egy típust* rendel minden minden élhez.

A tulajdonságok definiálásához legyen $D = \bigcup_i D_i$ a különböző D_i elemi domének uniója, és legyen NULL a `NULL` érték.

- P_v a csúcsok tulajdonságainak halmaza. A $p_i \in P_v$ csúcstulajdonság egy függvény $p_i : V \to D_i \cup \{\mathsf{NULL}\}$, amely egy csúcstulajdonság értéket rendel a $D_i \in D$ doménből a $v \in V$ csúcshoz, ha v rendelkezik a p_i csúcstulajdonsággal, egyébként $p_i(v)$ értéke NULL.



2.2. ábra. A példa tulajdonsággráf modellje. ◇ Person, □ Tag, △ TagClass

- P_e az élek tulajdonságainak halmaza.A $p_j \in P_e$ éltulajdonság egy függvény $p_j : E \to D_j \cup \{\mathsf{NULL}\}$, amely egy éltulajdonságot rendel a $D_j \in D$ doménből az $e \in E$ élhez, ha e rendelkezik a p_j éltulajdonsággal, egyébként $p_j(e)$ értéke NULL.

A továbbiakban a csúcs- és éltulajdonságokat is tulajdonságnak hívjuk, csak ott különböztetjük meg őket ahol a megértés könnyebbsége miatt szükséges.

A 2.2. ábrán látható a példa gráf tulajdonsággráf vizuális modellje, a tulajdonsággráf formális modellje pedig az alábbi:

$$
\begin{aligned}
L &= \{\mathsf{Person}, \mathsf{Student}, \mathsf{Class}, \ldots\} \\
T &= \{\mathsf{KNOWS}, \mathsf{INTEREST}, \mathsf{CLASS}, \mathsf{SUBCLASS\_OF}\} \\
P_v &= \{\mathsf{name}, \mathsf{speaks}, \mathsf{age}, \ldots\} \\
P_e &= \{\mathsf{since}\} \\
V &= \{a, b, c, d, e, f\} \\
E &= \{1, 2, 3, 4, 5\} \\
st: &\ 1 \to \langle a, c\rangle, 2 \to \langle b, c\rangle, \ldots \\
lbl: &\ b \to \{\mathsf{Person}\}, a \to \{\mathsf{Person}, \mathsf{Student}\}, \ldots \\
typ: &\ 1 \to \mathsf{KNOWS}, 3 \to \mathsf{CLASS}, \ldots \\
\mathsf{name}: &\ a \to \text{„}\mathsf{Alice}\text{”}, b \to \text{„}\mathsf{Bob}\text{”}, e \to \mathsf{NULL}, \ldots \\
\mathsf{since}: &\ 1 \to 2014, 2 \to \mathsf{NULL}, 3 \to \mathsf{NULL}, \ldots \\
&\ldots
\end{aligned}
$$

2.2.2. Szemantikus gráf

A szemantikus gráf olyan speciális adatmodell, amelyben egy gráfban találhatóak a metamodell és a példánymodell elemei. A szemantikus gráfnak számos reprezentációja létezik, ezek közül az egyik leggyakoribb a World Wide Web Consortium (W3C)[1] gondozásában lévő Resource Description Framework (RDF)[2]. Mivel az RDF pontos definíciója meghaladja e dolgozat kereteit, ezért csak a legfontosabb részeket emeltük ki a specifikációból.

Egy RDF gráf hármasok egy halmaza. Egy hármas elemei az alábbiak:

- alany, amely egy IRI hivatkozás vagy egy üres csomópont

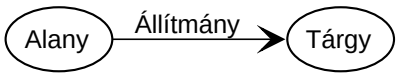
[1] https://www.w3.org/
[2] https://www.w3.org/standards/techs/rdf

- állítmány, amely egy IRI hivatkozás
- tárgy, amely egy IRI hivatkozás, egy literál vagy üres csomópont

Az IRI[3] (Internationalized Resource Identifier) egy UNICODE karakterlánc, amely egyértelműen azonosítja a gráf elemeit. Egy RDF gráfban az üres csomópontokat egy végtelen halmazból nyerjük. Az üres csomópontoknak ez a halmaza, továbbá az összes IRI hivatkozás halmaza, valamint az összes literál halmaza páronként diszjunkt halmazokat alkotnak. Formálisan tehát legyen I az IRI-k halmaza, B az üres csomópontok halmaza, L pedig a literálok halmaza, akkor az $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$ egy RDF hármas, ahol s az alany, p az állítmány és o a tárgy. A példa egy részletének megfelelő RDF szöveges reprezentációja[4]:

```
a type Person .
a type Student .
a name "Alice" .
a age 24 .
a knows _:tmp1 .
_:tmp1 hasPerson b .
b name "Bob" .
```

Ebben példában az IRI-k halmaza $\{a, b, Person, Student, type, name, age, knows, hasPerson\}$, a literálok halmaza $\{„Alice", „Bob", 24\}$ és az üres csomópontok halmaza pedig $\{_:tmp1\}$.



2.3. ábra. Egy RDF hármas grafikus ábrázolása

Az RDF gráf csomópontjainak halmazát a gráf hármasainak alanyai és tárgyai alkotják, az élek halmazát pedig a gráf hármasainak állítmányai alkotják. Egy hármas grafikusan ábrázolható két csomóponttal és a közöttük lévő irányított éllel, ahogy a 2.3. ábrán látható. Ilyen módon megkaphatjuk a példa RDF gráf modelljének a 2.4. ábrán látható grafikus vizualizációját. Az ábrán látható, hogy az RDF gráfok csúcs- és élcímkézett irányított gráfok.

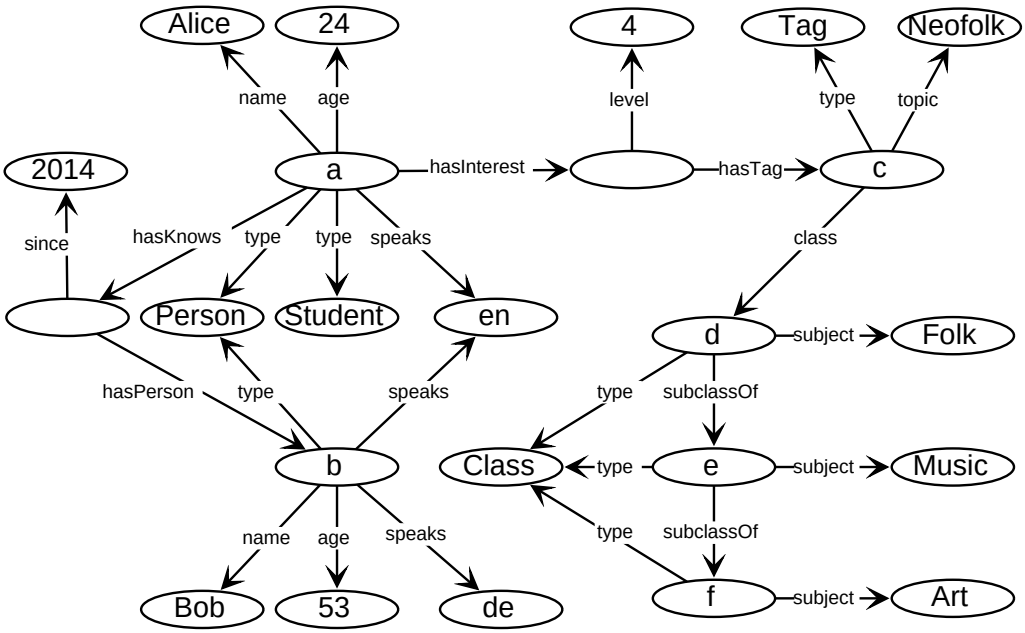
A 2.1 táblázat szemantikus gráfhoz tartozó sorában látható $*$ azt jelenti, hogy bár egy RDF gráfban a csúcsoknak nem lehet tulajdonsága, de a csúcsok tulajdonságait egyszerűen leképezhetjük hármasokká, ahogy a példa gráf RDF gráfján látszik: az eredeti gráf *name* tulajdonságát az RDF gráfban az azonos nevű állítmánnyal rendelkező hármas jelenti.

2.2.3. Relációs adatmodell

A dolgozatban megemlített adatmodellek közül a relációs adatmodell a legrégebb óta használt és kutatott modell. Ebben a fejezetben az E. F. Codd által leírt definíciót [7] ismertetjük. Adottak S_1, S_2, ..., S_n nem feltétlenül különböző halmazok, ekkor r egy reláció ezen az n halmazon, ha $r \subseteq S_1 \times S_2 \times \ldots \times S_n$. Ilyenkor az S_i halmaz az r i. doménje.

Gyakran a relációkat táblázatokként ábrázoljuk. Az r relációt ábrázoló táblázat minden sora megfeleltethető pontosan egy elemnek r-ből, így a reláció elemeit a reláció sorainak is nevezzük. Az oszlopok sorrendje követi az S_1, S_2, ..., S_n sorrendet, azaz

[3]Formális definíció:https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/#section-IRIs
[4]N-Triples formátumról bővebben: https://www.w3.org/TR/n-triples/



2.4. ábra. A példa RDF gráf modellje

az i. oszlop az i. domén értékeit tartalmazza. Az oszlopokat szokás a hozzájuk tartozó domén nevével felcímkézni, így egyértelműsíteni a jelentésüket. Esetenként az oszlopokra egyszerűen a nevükkel hivatkozunk és nem vesszük figyelembe az attribútumok sorrendjét.

A relációk lehetnek *halmaz* (set) szemantikájúak, ebben az esetben az elemeik egyediek, valamint *multihalmaz* (bag) szemantikájúak, amikor az elemeiek között előfordulhatnak ismétlődések. A reláció elemeinek sorrendje egyik esetben sem számít.

A példa relációs modellje a 2.4. ábrán látható. Megfigyelhető, hogy a példában minden csúcstípushoz (Persons, Tags, TagClasses relációk), minden többes kardinalitású csúcs tulajdonságához (Speaks) és minden éltípushoz (Interests, SubclassOf, HasClass) külön reláció tartozik: ez gyakori leképzése a gráfoknak a relációs adatmodellre. További fontos megfigyelés, hogy az éleket jelentő relációknak (Knows, Interests, SubclassOf, HasClass) nincsen azonosítója, mert ezeket az él kezdő- és végpontja azonosítja. Ez a séma feltételezi, hogy két csúcs között nem vezet egynél több azonos él (ugyanolyan típusú él vezethet, ha legalább egy éltulajdonságban eltérnek), továbbá hogy tanulók csak személyek lehetnek (Students reláció).

2.3. Lekérdezőnyelvek

Az utóbbi évtizedben a gráf alapú adatbázisok elterjedtek mind az ipari, mind az akadémiai területeken. Az adatbázisok ezen új generációja által nyújtott új szolgáltatások (gráfminták megfogalmazása, gráf algoritmusok használata, részgráf illesztész) használatához szükségessé vált új lekérdezőnyelvek megalkotása. Ebben a fejezetben bemutatunk három új lekérdezőnyelvet és természetesen a relációs adatbázisok (relational databases, RDB) lekérdezőnyelvét, az SQL-t is bemutatjuk. A lekérdezőnyelvek bemutatása során a dolgozat megértéséhez szükséges részletekre fókuszálunk, mivel a nyelvek részletes bemutatása meghaladja a dolgozat kereteit. A bemutatott nyelveken egy

id	name	age
a	Alice	24
b	Bob	53

(a) Persons

personId
a

(b) Students

id	topic
c	Neofolk

(c) Tags

id	subject
d	Folk
e	Music
f	Art

(d) TagClasses

tag	class
c	d

(e) HasClass

src	trg
d	e
e	f

(f) SubclassOf

src	trg	since
a	b	2014

(g) Knows

person	tag	level
a	d	4

(h) Interests

personId	lang
a	en
b	de
b	en

(i) Speaks

2.5. ábra. A példa relációs modellje

egyszerű lekérdezést is megfogalmazunk a nyelvek szemléltetése miatt: „adjuk vissza az emberek nevét és ismerőseik számát!”

2.3.1. Relációalgebra

Az adatbázis-kezelés egyik legismertebb formális nyelve a relációalgebra, mely a relációs adatmodell (2.2.3. szakasz) feletti különböző halmazműveleteket definiál [25, 13, 26]. Az alábbiakban röviden áttekintjük a dolgozatban alkalmazott relációalgebrai operátorokat.

Unáris operátorok

A *projekció* operátor $\pi_A(r)$ a bemeneti r relációt olyan relációra alakítja, ami csak az A halmazban szereplő attribútumokat tartalmazza. A projekció operátor képes továbbá attribútumok átnevezésére is az alábbi jelöléssel: $\pi_{x1/y1,x2/y2,\ldots}(r)$. A *szelekció* operátor $\sigma_\theta(r)$ a bemeneti r reláció azon sorait tartja meg, melyek kielégítik a θ logikai kifejezést.

Multihalmazok felett értelmezett műveletek A *duplikátumszűrés* operátor (δ) a bemeneti relációból olyan reláció készít, amiben nincsenek azonos sorok. Más szavakkal a bemeneti multihalmaz szemantikájú relációt halmaz szemantikájúra alakítja. A *csoportosítás* operátor (γ) a reláció sorait egy adott attribútumhalmaz szerint csoportosítja, majd a fennmaradó attribútumokat aggregálja. A $\gamma^{c_1,c_2,\ldots}_{e_1,e_2,\ldots}(r)$ kifejezés az r reláció sorait a $c_1, c_2, \ldots$ kritériumattribútumok mentén csoportosítja, majd minden csoportra kiszámítja a $\langle e_1, e_2, \ldots \rangle$ sorok értékét.

Bináris operátorok

Illesztés jellegű műveletek Az *illesztés* (join) jellegű műveletek két reláció összekapcsolását végzik el, általában növelve az attribútumok számát. Az illesztés műveletek alapja a *Descartes-szorzás* (Cartesian product). A $r \times s$ kifejezés egy n sorból álló r és egy m sorból álló s relációból egy $n \cdot m$ sorú relációt állít elő, mely tartalmazza r és s minden attribútumát, valamint azok sorait minden lehetséges kombinációban.

A *természetes illesztés* ($\bowtie$) művelet az azonos nevű attribútumok mentén kapcsolja össze a bemeneti relációkat. Formálisan:

$$r \bowtie s = \pi_{R \cup S}\left(\sigma_{r.A_1=s.A_1 \wedge \ldots \wedge r.A_n=s.A_n}(r \times s)\right)$$

Az *antijoin operátor* ($\overline{\rtimes}$) a bal oldali bemeneti reláció azon sorait tartja meg, amikhez nincs illeszkedő sor a jobb oldali bemeneti relációban. Formálisan:

$$r \,\overline{\rtimes}\, s = r - \pi_R \left(r \bowtie s \right)$$

A *bal oldali külső illesztés* operátor (left outer join, $\overset{\frown}{\bowtie}$) [26] előállítja a két bemeneti reláció természetes illesztését, majd hozzáfűzi ehhez a baloldali reláció azon elemeit, amik nem kerültek be az illesztésbe, NULL elemekkel a megfelelő szélességűre kiegészítve. Formálisan, $r \overset{\frown}{\bowtie} s \equiv (r \bowtie s) \cup \left((r \,\overline{\rtimes}\, s) \times \langle \mathsf{NULL} \rangle_{(S-R)} \right)$, ahol a $\langle \mathsf{NULL} \rangle_k$ jelentése egy k szélességű, NULL elemekből álló sor.

Unió műveletek Az unió művelet ($\cup$) két relációt fűz össze az attribútumok mentén. Formálisan $t \in r \cup s \iff (t \in r) \vee (t \in s)$. A multihalmazok felett definiált relációalgebra esetén gyakran megkülönböztetik az unió operátor két típusát: a *halmaz unió* ($\cup$) operátor duplikátumszűrést végez, míg a *multihalmaz unió* operátor ($\uplus$) esetén lehetnek duplikátumok a kimenetben. A kétféle unió operátor közötti összefüggés a $r \cup s = \delta(r \uplus s)$ kifejezéssel írható le. A dolgozatban a továbbiakban az unió operátor alatt a multihalmaz unió operatort értjük.

Példa A példa lekérdezés relációalgebrában az alábbi módon fogalmazható meg:

$$\gamma^{p.\mathsf{name}}_{p.\mathsf{name},\mathtt{count}(f)} \left[\pi_{p,p.\mathsf{name}}(\mathsf{Persons}) \overset{\frown}{\bowtie} \left(\pi_{p,f}(\mathsf{Knows}) \bowtie \pi_f(\mathsf{Persons}) \right) \right]$$

2.3.2. Cypher

A Cypher nyelv [12] az egyik legelterjedtebb a nyelv tulajdonsággráfok lekérdezésére és módosítására. A nyelv a Neo4j gráfadatbázis-kezelőben jelent meg először, majd később számos másik termékben (SAP HANA Graph [24], Redis Graph[5]) implementálták. A kereskedelmi termékek mellett több kutatási projektekben (ingraph [18], Cytosm [27]) is használni kezdték a Cyphert. Az openCypher[6] projekt 2015-ös elindulásával létrejött egy olyan platform, amelynek célja a Cypher nyelv új nyelvi elemeinek kollaboratív kidolgozása a különböző Cyphert használó termékek és projektek fejlesztőinek bevonásával. A projekt célja az, hogy a Cypher váljon az ipari szabvánnyá a tulajdonsággráfok lekérdezésére és módosítására. Fontos megjegyezni, hogy a projekttel együtt létrejött az openCypher nyelv is, amely a Cypher nyelv egy valódi részhalmaza, azonban nem tartalmazza az összes Cypherben lévő nyelvi konstrukciót (pl. `shortestPath`, `reduce` stb.). A dolgozat további részében mindig felhívjuk a figyelmet a két nyelv közötti különbségekre, ha azok nélkülözhetetlenek a megértés szempontjából.

Linearitás. Egy Cyper lekérdezés bemenete egy tulajdonsággráf, kimenete pedig egy táblázat, amely tartalmazza a gráfból kinyert információt. A lekérdezések struktúrája lineáris: az állítások egymás után következnek a lekérdezésben. Az állítások tekinthetőek függvényeknek, amelyek bemenete és kimenete is egy táblázat. Az állítások módosíthatják az oszlopok számát, sorokat szűrhetnek ki és adhatnak hozzá a táblázathoz. Egy lekérdezés tehát ilyen függvények sorozata. Fontos azonban, hogy az állítások sorrendje szigorúan deklaratív jellegű, azaz a konkrét implementációk felcserélhetik két állítás végrehajtását, ha az nem változtatja meg az eredményt. A deklaratív jellegnek köszönhetően nem kell az SQL `SELECT`-hez hasonlóan rögtön a lekérdezés elején leírni a projekciót, hanem elég a lekérdezés végén a `RETURN` kulcsszóval. Az egyes állításokban szintén lehetséges projekciót végezni a `WITH` kulcsszóval. A `WITH` ugyanazokat a projekciókat engedi meg, mint a `RETURN`,

[5] `https://oss.redislabs.com/redisgraph/`
[6] `https://www.opencypher.org/`

ideértve az aggregációt. Továbbá, a `WITH` támogatja az egyes mezők értéke szerinti szűrést. A lineáris komponálhatóságon túl a Cypher támogatja az olyan beágyazott lekérdezéseket, mint például az `UNION` lekérdezések.

Mintaillesztés. A Cypher központi eleme a gráfminták megfogalmazása. A gráfminták Cypherben `(a)-[r]->(b)` alakúak, ahol `a` és `b` a gráf két csúcsát, `r` pedig az őket összekötő él típusát jelenti. Van lehetőség egy éltípusból alkotott utak megofgalmazására is `(a)-[r*x..y]->(b)` formában, ahol `x` és `y` tetszőleges egész számok az `x <= y` feltétellel. Opcionálisan `x` és `y` is elhagyható. A `MATCH` kulcsszó az ilyen módon megfogalmazott gráfminta lehetséges értékeit adja eredményül.

A nyelv mélyebb ismerete nem szükséges a dolgozat megértéséhez, így a további részleteket (gráfok módosítása, többes attribútumok és listák kezelése) nem mutatjuk be. A példa lekérdezés Cypher nyelven az alábbi:

```
MATCH (p:Person)
OPTIONAL MATCH (p)-[:KNOWS]->(f:Person)
RETURN p.name, count(f)
```

2.1. kódrészlet. Cypher példakód

Ahogy látható, a `OPTIONAL MATCH` kulcsszó használható egy gráfminta opcionális illesztésére. Fontos részlet, hogy az SQL-lel szemben Cypherben nem kell `GROUP BY` kulcsszóval megadni a nem aggregált tulajdonságokat.

2.3.3. Gremlin

A Gremlin [23] az Apache TinkerPop[7] projekt által tervezett, fejlesztett és terjesztett gráfbejáró automata és nyelv. A Cypherrel ellentétben a Gremlin nem csak gráf minták illesztését tudja elvégezni, hanem iteratív lekérdezéseket is megfogalmazhatunk használatával. A Gremlin továbbá elősegíti, hogy a felhasználó:

- a saját programozási nyelvébe beágyazza,
- kiterjessze domén specifikus kifejezésekkel,
- a kiterjeszthető, fordítási idejű újraírási szabályokkal optimalizálja
- és egy számítógép-klaszteren futassa

a Gremlin lekérdezéseket.

A gráfbejáró automata három részből áll: egy G gráfból (adat), egy Ψ bejárási szekvenciából (instrukciók) és bejárók egy T halmazából (olvasási/írási referenciák). Az automata magas szintű működése: a bejárók T halmaza mozog a G gráfon a Ψ-ben megfogalmazott instrukciók szerint. A számítás akkor ér véget, amikor már vagy nem létezik bejáró T-ben, vagy az összes létező bejáró megállt, azaz nem végez több instrukciót Ψ-ből. Az első esetben az eredmény egy üres halmaz, utóbbi esetben pedig a bejárók által hivatkozott G-beli helyek multihalmaz uniója.

A Gremlin lekérdezőnyelv egy funkcionális programozási nyelv. Célja annak biztosítása, hogy a felhasználók az emberek számára érthető kifejezésekkel, egyszerűen definiálhassák bejárási szekvenciát, azaz egyszerűen hozzájussanak a számukra fontos információhoz. A nyelv építő elemei a függvénykompozíciók és a functor típusú objektumok. Például az $a \circ b \circ c$ kifejezés leírható `a().b().c()` alakban. A függvény paraméterekkel pedig a bejárások egymásba ágyazása is lehetséges, például az $a(b \circ c) \circ d$ kifejezés `a(b().c()).d()` alakban írható le.

[7] `http://tinkerpop.apache.org/`

Példa A példa lekérdezés Gremlin nyelven:

```
g.V().hasLabel('person').as('person').property('name').as('pName')
    .select('person').optional(outE('KNOWS').inV()).count()
    .select('pName', 'count')
```

2.2. kódrészlet. Gremlin példakód

2.3.4. SPARQL

Az RDF 1998-as megjelenésével együtt megjelent az igény az RDF gráfok lekérdezését, módosítását lehetővé tevő nyelvekre. 2004-ben a W3C RDF Data Access Working Group szervezete kiadta a Simple Protocol and RDF Query Language első publikus tervezetét. Azóta a SPARQL [22] elterjedt és a RDF gráfok szabványos lekérdezőnyelvévé vált.

Mivel az RDF egy irányított címkézett gráf adatmodell, ezért a SPARQL esszenciális része a gráfminták megfogalmazása. A SPARQL-ben az RDF gráfoknál ismeretett I, B és L halmazokon kívül létezik egy, velük diszjunkt V halmaz, amely a változók egy végtelen halmaza. Egy SPARQL lekérdezést tekinthetünk $H \longleftarrow M$ alakúnak, ahol M a lekérdezése törzse, egy komplex RDF gráfminta változókkal, opcionális részekkel, metszetekkel, különbségekkel és a változókra vonatkozó kényszerekkel, H pedig a lekérdezés feje, egy kifejezés, amely megmondja, hogy hogyan állítsuk össze a lekérdezés eredményét. H különböző módosítókat tartalmazhat, például sorrendet, limitet definiálhat, de tartalmazhat klasszikus reláció algebrai kifejezéseket is. Egy SPARQL lekérdezés kimenete változatos formátumú lehet: igen/nem válasz, egy táblázat vagy egy új RDF gráf. Egy Q lekérdezés kiértékelése a D RDF gráfon két lépésben történik: Q törzsét illesztjük D-re, hogy a törzsben lévő változók lekötéseinek egy halmazát kapjuk, amelyet a fej kiértékelésekor használjunk fel. A SPARQL gráfminta rekurzív definíciója az alábbi:

1. Egy hármas a $(I \cup B) \times I \times (I \cup B \cup L)$ halmazból gráfminta.
2. Ha P_1 és P_2 gráfminták, akkor $(P_1 \ . \ P_2)$, $(P_1$ `OPTIONAL` $P_2)$ és $(P_1$ `UNION` $P_2)$ is gráfminták.
3. Ha P egy gráfminta és R egy SPARQL feltétel, akkor P `FILTER` R is gráfminta.

A pont szimbólum a minták összefűzését jelenti, tehát P_1-nek és P_2-nek is illeszkednie kell. Az `OPTIONAL` kulcsszó esetén pedig a P_1-re illeszkedő részgráfokon lekötésre kerülnek P_2 változói, ha P_2 illeszkedik a részgráfra. Egyéb esetben P_2 változói lekötetlen változók lesznek.

Példa A példa lekérdezés SPARQL nyelven az alábbi:

```
SELECT
    ?personName,
    (COUNT(friend) AS ?friendCount)
WHERE
{
    ?person a Person .
    ?person foaf:name ?personName .
    OPTIONAL {
        ?person knows/hasPerson ?friend .
        ?friend a Person
    }
}
GROUP BY ?personName
```

2.3. kódrészlet. SPARQL példakód

A példán látható, hogy SPARQL-ben a változókat a ? prefixszel jelöljük.

2.3.5. SQL

Az SQL-t 1979-ben, az Oracle V2 megjelenésekor kezdték el használni relációs adatbázisok lekérdezésére, módosítására. Azóta iparági szabvánnyá vált, a relációs adatbázisok túlnyomó részében elérhető. Először 1986-ban szabványosították, azóta többször frissítették a szabványt, utoljára 2016-ban. A megjelenése óta számos más nyelv alapjául szolgált. A népszerűsége ellenére azonban az egyik legjelentősebb hiányosságot még manapság sem sikerült megjavítani: az SQL szabvány természetes nyelven írt, ebből adódóan sok részletet nem tud kellő pontossággal specifikálni. Az SQL-t többször próbálták már formalizálni [14], de általánosan elfogadott megoldást eddig nem sikerült alkotni a szabvány kiterjedtsége és a `NULL` értékek kezelésével járó kihívások miatt.

Közismert, hogy az SQL a sikerét nagyban a deklaratív jellegének köszönheti. Így nem kell pontosan tudnunk, hogyan kell hatékonyan előállítani a számunkra fontos információt, csupán elég megfogalmazni azt, hogy milyen információra van szükségünk. Az adatbázis-kezelő feladata, hogy a lekérdezést lefordítsa, optimalizálja és végrehajtsa.

Az SQL számos nyelvi elemet tartalmaz, köztük a következőket:

- Klózok (`SELECT`, `WHERE`, `LIMIT`, `JOIN` stb.), amelyek kompozíciója alkotja a lekérdezéseket és állításokat.
- Kifejezések, amelyek eredménye lehet egy skalár vagy egy sorokból és oszlopokból álló táblázat.
- Predikátumok, amelyek olyan feltételeket határoznak meg, amelyek az SQL háromértékű logikájával (igaz/hamis/ismeretlen) is kiértékelhetőek, és használatukkal limitálható, módosítható az állítások és lekérdezések hatása, működése.
- Állítások, amelyekkel perzisztens módosításokat végezhetünk az adatokon.

Példa A példa lekérdezés SQL nyelven (a 2.5. ábrán látható táblákat feltételezve):

```
SELECT p.name, COUNT(f.id)
FROM persons AS p
LEFT JOIN knows ON p.id = knows.src
JOIN persons AS f ON knows.trg = f.id;
```

2.4. kódrészlet. SQL példakód

2.4. Technológiák

A bemutatott adatmodelleket és lekérdezőnyelveket több adatbázis-kezelő implementálta. Ebben a fejezetben bemutatjuk a dolgozat szempontjából legfontosabb alkalmazásokat.

Formátum	Alkamazás	Lekérdezőnyelv	Mem.	Impl. nyelve
PG	JanusGraph	Gremlin	∘	Java
	Neo4j	Cypher	∘	Java
	TinkerGraph	Gremlin	•	Java
RDF	4store	SPARQL	∘	C
	AllegroGraph	SPARQL	∘	Lisp
	Stardog	SPARQL	∘	Java
	Virtuoso	SPARQL, SQL	∘	C, C++
RDB	MySQL	SQL	∘	C, C++
	PostgreSQL	SQL	∘	C, C++

2.6. ábra. Adatbázisok összefoglalása. A *Mem.* oszlop a memória alapú (in-memory) működést jelöli

2.4.1. Tulajdonsággráf alapú adatbázisok

JanusGraph A JanusGraph[8] egy nyílt forráskódú elosztott gráfadatbázis, amely több tárolási technológiát is támogat: Apache Cassandra[9], Apache HBase[10], Google Cloud Bigtable[11], Oracle BerkeleyDB[12]. Natív támogatást nyújt az Apache TinkerPop termékcsaládhoz, lekérdezőnyelve a Gremlin.

Neo4j Jelenleg, 2018-ban az egyik leggyakrabban használt gráfadatbázis a Neo4j[13]. Az adatok lekérdezése történhet egy alacsonyszintű Java API-n keresztül, amellyel primitív gráfműveleteket hajthatunk végre, illetve deklaratív módon a Cypher nyelven megfogalmazott lekérdezésekkel. Az egyik hátránya, hogy nem támogatja a csak memóriában történő tárolást, csak a merevlemez alapú perzisztens tárolást. Támogatja azonban a fürtök létrehozását.

TinkerGraph A TinkerGraph egy memóriaalapú referencia implementáció a TinkerPop interfészhez. Jellegéből adódóan nem ipari felhasználásra van tervezve, így nem teljesítmény-orientált.

2.4.2. Szemantikus adatbázisok

Stardog A Stardog támogatja az RDF és tulajdonsággráf adatmodellt, lekérdezéshez használható SPARQL és Gremlin is. A Stardog új verzióiban a SPARQL lekérdezésekhez a végrehajtásra vonatkozó plusz információkat adhatunk (például milyen algoritmust használjon egy `JOIN` művelet), ezzel gyorsítva a végrehajtást.

[8] `http://janusgraph.org/`
[9] `http://cassandra.apache.org/`
[10] `https://hbase.apache.org/`
[11] `https://cloud.google.com/bigtable/`
[12] `https://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html`
[13] `https://db-engines.com/en/ranking/graph+dbms`

Virtuoso A Virtuoso [11] egy többféle adatmodellt támogató adatbázis-kezelő. támogatja a relációs és RDF adatmodellt is. Az adathozzáférés ennek is köszönhetően két nyelven is lehetséges: SPARQL és SQL nyelven. Fontos tulajdonsága, hogy a többi gráf alapú adatbázis-kezelővel ellentétben nem Java, hanem C és C++ nyelvű az implementáció.

AllegroGraph Az AllegroGraph[14] egy nagyteljesítményű, perzisztens adattárolást megvalósító kereskedelmi forgalomban kapható szemantikus gráfalapú adatbázis-kezelő. Lekérdező nyelve a SPARQL, de számos programozási nyelven megírt interfésszel rendelkezik. Beépített támogatást nyújt a szemantikus következtetésre RDFS[15], SPIN[16] vagy Prolog nyelven megfogalmazott szabályok alapján.

4store A 4store [15] egy nyílt forráskódú, RDF alapú adatbázis-kezelő, amely támogatja a fürtök létrehozását. Lekérdező nyelve a SPARQL. A fürtözéssel kapcsolatban fontos megjegyezni, hogy a számítások mellett az adattárolás is elosztott módon történik.

2.4.3. Relációs adatbázisok

PostgreSQL A PostgreSQL az egyik leginnovatívabb nyílt forráskódú relációs adatbázis-kezelő. Felhasználási területe igen széles spektrumú: az egyszerű, egy szerveres konfigurációtól kezdve a hatalmas adattárházakig mindenhol megtalálható. Lekérdezőnyelve az SQL szabvány jelentős részét lefedi, beleértve a `WITH RECURSIVE` kifejezést[17], amely rekurzív lekérdezések megfogalmazását teszi lehetővé, így teremtve meg a lehetőséget az útkereső lekérdezéseknek.

MySQL Az MySQL az Oracle tulajdonában lévő nyílt forráskódú relációs adatbázis-kezelő, azonban létezik többletfunkcionalitást nyújtó licenszelt verziója is. Lekérdezőnyelve egy saját elemekkel kiegészített SQL dialektus. A 2018 áprilisban megjelent 8.0-as verzió[18] óta szintén támogatja a `WITH RECURSIVE` kifejezést, így szintén lehetőséget teremtve az útkereső lekérdezések futtatására.

[14] https://allegrograph.com/
[15] RDF Schema: https://www.w3.org/TR/rdf-schema/
[16] SPARQL Inferencing Notation: https://www.w3.org/Submission/spin-overview/
[17] Szintaktika és szemantika: https://www.postgresql.org/docs/9.1/static/queries-with.html
[18] MySQL 8.0 dokumentáció: https://dev.mysql.com/doc/refman/8.0/en/with.html

3. fejezet

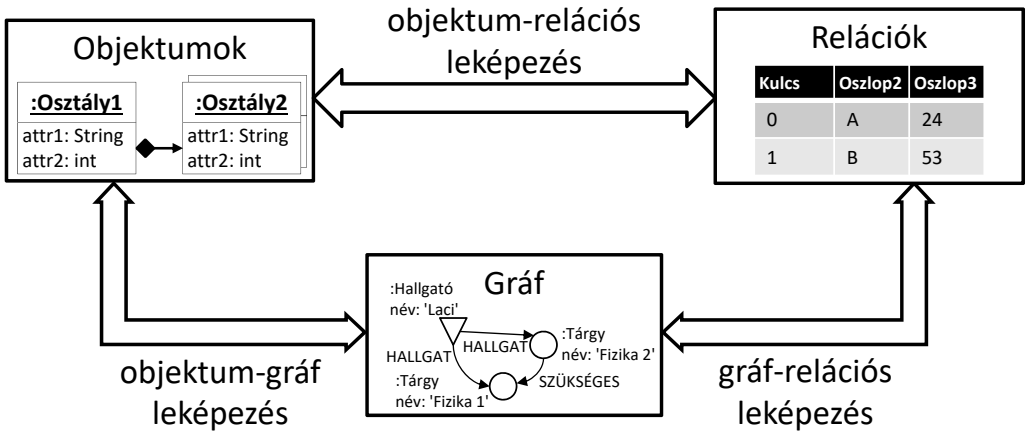
Gráflekérdezések leképzése relációs adatbázisokra

Az alábbi fejezetben bemutatjuk a gráflekérdezések relációs adatbázisra való leképzésünk módszerét. A 3.1. szakasz a relációs és a gráfos adatmodellek közötti különbségeket, a 3.2. szakasz a gráflekérdezések SQL lekérdezésre való fordításának menetét mutatja be. A 3.3. szakaszban kiterjesztjük a leképezést, hogy tetszőleges, előre definiált sémát követő relációs adatbázison alkalmazható legyen. Az elkészült szoftver lehetséges felhasználási módjait a 3.4. szakasz ismerteti.

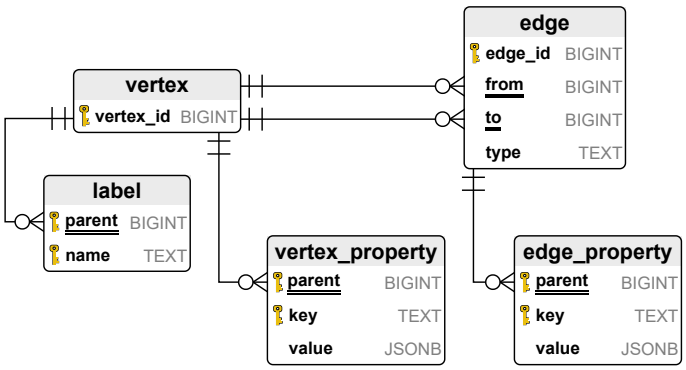
3.1. Relációs és gráfadatbázisok fogalmainak megfeleltetése

A relációs modell (RM) és tulajdonsággráf (PG) adatmodellek közti különbségek vizsgálatához ezeket vessük össze egy további gyakran használt adatmodellel, az objektumorientált (OO) adatmodellel. A 3.1. ábra bemutat egy-egy példát ezen adatmodellekre, továbbá a köztük lévő leképezéseket.

Objektum-gráf leképezés A három modell közül az objektumorientált és a tulajdonsággráf modellek közötti különbségek a legkisebbek. Közöttük a következő leképezés definiálható: Az egyek objektumok csomópontoknak feleltethetők meg, az attribútumait a csúcs tulajdonságaiban tárolhatjuk, az osztályok adják a csúcsok címkéit, amelyek a típust jelzik. Az öröklési kapcsolatok további címkékkel jelölhetőek: például a `Student` címkéjű csúcsok rendelkeznek a `Person` címkével is, amely jelzi az öröklést.



3.1. ábra. Objektumorientált, relációs és gráf adatmodellek közti leképzések



3.2. ábra. Tulajdonsággráfot reprezentáló relációs séma
A kulcs oszlopokat, az idegen kulcsokat dupla aláhúzás,
a több-egy kapcsolatokat jelöli.

A PG modellben ez a hierarchia csak a példánygráfban tud megjelenni, mivel a tulajdonsággráfokhoz nem tartozik séma. Az objektumok közötti kapcsolatok (asszociáció, aggregáció stb.) éleket jelentenek a gráfban. A ritkán használt UML *Association Class* feleltethető meg az élek tulajdonságainak.

Gráf-relációs és objektum-relációs leképezés A gráf-relációs leképezésre léteznek ajánlások[1][2], azonban ezek főként a relációs rendszerekről gráfadatbázisra történő áttérést szolgálják. Mivel csekély különbség mutatkozik az objektumok és a tulajdonsággráfok között, érdemes a gyakran alkalmazott objektum-relációs leképezést (Object-Relational Mapping – ORM) vizsgálnunk. Erre a célra különböző szoftverek léteznek (pl.: Hibernate [9, 20], Entity Framework [5], Java Persistence API [8]), amelyek relációs adatbázisok OO nyelvekben való használatát könnyítik meg.

3.1.1. Relációs séma definiálása tulajdonsággráf reprezentálására

A tulajdonsággráfok reprezentálására általunk használt relációs séma a 3.2. ábrán, a példa relációk a 3.3. ábrán láthatóak. A gráf csúcsainak azonosítóit a `vertex` tábla tárolja. Mivel a csúcsok tetszőlegesen sok címkével rendelkezhetnek, ezeket külön tábla, a `label` tábla tárolja. Az élek azonosítóit, idegen kulccsal a hozzájuk tartozó forrás- és célcsúcsokat, valamint az él típusát az `edge` tábla tárolja. A csúcsokhoz és élekhez tartozó tulajdonságok nevét és értékét a `vertex_property` és az `edge_property` tábla tárolja. A *generikus adatséma*[3] alkalmazását a gráfadatbázisok séma nélküli tulajdonsága [1, 6] indokolja. A séma nélküliség azt jelenti, hogy egy gráfadatbázisba bármikor új típusú csúcsokat és éleket lehet beszúrni, nincs szükség a séma előzetes módosítására. Emiatt előre nem ismert, hogy milyen típusú csúcsok és élek lesznek az adatbázisban, így a relációs sémának tetszőleges hierarchia tárolására alkalmasnak kell lennie.

3.1.2. A leképzéshez szükséges további követelmények

A Cypher nyelv több olyan konstrukcióval is rendelkezik, amelyet kevés SQL-dialektus támogat, így ezeket a használandó relációs adatbázis kiválasztása során figyelembe kellett vennünk. Ezek alapján a vizsgált rendszerek a következők: MySQL, SQLite, PostgreSQL.

[1] `https://neo4j.com/resources-old/rdbms-developer-graph-white-paper/`
[2] `https://dzone.com/refcardz/from-relational-to-graph-a-developers-guide`
[3] `http://www.agiledata.org/essays/mappingObjects.html`

vertex_id
a
b
c
d
e
f

(a) vertex reláció

parent	name
a	Person
a	Student
b	Person
c	Tag
d	Class
e	Class
f	Class

(b) label reláció

parent	key	value
a	name	"Alice"
a	age	24
a	speaks	["en"]
b	name	"Bob"
b	age	53
b	speaks	["en", "de"]
...		

(c) vertex_property reláció

edge_id	from	to	type
1	a	b	KNOWS
2	a	c	INTEREST
3	c	d	CLASS
4	d	e	SUBCLASS_OF
5	e	f	SUBCLASS_OF

(d) edge reláció

parent	key	value
1	since	2014
2	level	4

(e) edge_property reláció

3.3. ábra. A példa tulajdonsággráfot reprezentáló relációk

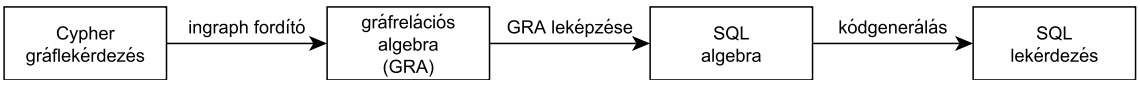
Tömbkezelés A Cypher nyelv támogatja a tömbök használatát. A tulajdonságok értékei mellett az egyes lekérdezések eredményét is tömbbe lehet gyűjteni a `collect` függvény használatával (3.2.5. szakasz). Az SQL99 szabvány korlátozottan, SQL2003 teljeskörűen támogatja a tömbök használatát, ennek ellenére a felsoroltak közül csak a PostgreSQL támogatja a tömböket, ill. az azokkal végezhető műveleteket.[4][5][6]

Dinamikus típus A PG adatmodellben különböző tulajdonságok értékei eltérő típusúak lehetnek, továbbá egyazon tulajdonság értékeinek típusa is eltérő lehet különböző csúcsok vagy élek esetében. Ezt SQLite adatbázisban a `property` táblákban `BLOB` (más rendszerekben `VARIANT`) típus használatával lehet megoldani, ilyenkor a bemenetnek megfelelően tárolja az adott értéket. MySQL és PostgreSQL adatbázisokban erre nincs lehetőség. Megoldás lehet felvenni az összes lehetséges típusnak egy-egy oszlopot, és mindig csak az adott értéknek megfelelőt kitölteni. MySQL adatbázisban ez egy működőképes megoldás. PostgreSQL-ben ez tárolás közben megvalósítható, azonban a lekérdezésekben is szerepelhet ugyanabban az oszlopban két eltérő típusú érték, amelyet a PostgreSQL nem támogat. (Minden oszlop megtöbbszörözése az eltérő típusok érdekében feleslegesen növelné a lekérdezések komplexitását.)

MySQL és PostgreSQL adatbázisokban is támogatott JSON típusú oszlopok használata. Ennek segítségével tetszőleges (JSON-ban kifejezhető) adattípust lehet ábrázolni, amely elégséges a gráfadatbázisok leképzéséhez. Ezek mellett mindkét rendszer segédfüggvényekkel támogatja ezen adatok létrehozását, módosítását, vizsgálatát.

Rekurzív lekérdezések Mivel a Cypher nyelv támogat tetszőleges hosszú útvonalakat tartalmazó lekérdezéseket is (pl.: `MATCH (p:Person)<-[:KNOWS*]->(foaf)`), ezért ezek kifejezéséhez szükséges az SQL99 szabvány óta a nyelv részét képező rekurzív lekérdezések

[4] `https://www.postgresql.org/docs/10/static/arrays.html`
[5] `https://dev.mysql.com/worklog/task/?id=2081`
[6] `https://www.sqlite.org/datatype3.html`



3.4. ábra. Cypher lekérdezések SQL nyelvre fordításának lépései

(`WITH RECURSIVE`) támogatása. A PostgreSQL 2009, a SQLite 2014, a MySQL 2018 óta támogatja ezt a kifejezést.

A felsorolt kritériumokat legnagyobb részben a PostgreSQL adatbázis-kezelő teljesítette, így ezt a rendszert választottuk az implementációnk alapjául, illetve a továbbiakban ezen a rendszeren futtatható lekérdezések kerülnek bemutatásra. A tulajdonságok értékeit és ezek típusát JSON oszlopban tároljuk, ezért a betöltésnél, a kiolvasásnál és a műveletvégzéseknél konverziókra van szükség.

3.2. Gráflekérdezések leképzése SQL nyelvre (C2S)

Az alábbiakban ismertetjük a lekérdezések leképzésének módszerét, lépéseit és az egyes elemi lépéseket bemutató példákat. A 3.2.1. szakasz a C2S (*Cypher-to-SQL*) fordító felépítését, a 3.2.2.–3.2.8. szakaszok az olvasás műveletek fordításának elemi lépéseit, a 3.2.9. szakasz a létrehozás művelet lépéseit mutatja be.

3.2.1. A leképzés lépései

A Cypher nyelvű gráflekérdezések SQL nyelvre történő fordításának lépéseit a 3.4. ábra mutatja be. A Cypher lekérdezések közvetlen feldolgozását a BME MIT és TMIT tanszékein fejlesztett *ingraph* eszköz [30] végzi. Az eszköz egy gráfspecifikus operátorokkal kiterjesztett relációs algebrai kifejezésre (GRA) fordítja a lekérdezést. Ezekre a kifejezésekre a következő tulajdonságok jellemzőek:

- gráfrelációs algebrai kifejezésekből alkotott fa,
- minden művelet eredménye a tulajdonsággráf doménjén értelmezett reláció, amely multihalmaz szemantikával rendelkezik (azaz a duplikátumok megengedettek),
- minden csomópont a gyerekeinek eredményéből állítja elő a saját eredményét,
- a tulajdonsággráfból értékek a relációba csak a levél csomópontokban kerülhetnek, a többi csomópont a gyerekeinek az adatait dolgozza fel.

Továbbiakban a [30] munkában bemutatott algebrát és jelölésrendszert használjuk.

Az általunk készített fordító a gráfrelációs algebra kifejezéseit SQL algebrává fordítja. Ennek során előállítja a GRA-műveletek SQL algebrabeli megfelelőit az operátorokban használt metaadatok összegyűjtésével (címkék, típusok, tulajdonságok, változók, predikátumok stb.), illetve több műveletre való szétbontással, ha szükséges. A következő fázisban az SQL algebra műveleteiből SQL kódot állítunk elő kódgenerálás segítségével. Minden műveletből egy névvel ellátott lekérdezést (Common Table Expression) készítünk, amely hivatkozik a gyermek lekérdezésekre. Ezek a lekérdezések az SQL lekérdezés `WITH` szakaszába kerülnek (lásd 3.2.4. szakasz).

A következő szakaszokban bemutatjuk az egyes GRA műveletek SQL-beli megfelelőit.

3.2.2. Csúcsok kigyűjtése (get-vertices)

A következő lekérdezés kigyűjti az összes diák azonosítóját (`p`) és nevét (`p.name`) a gráfból. A felső sorban a Cypher nyelvű lekérdezés és a lekérdezés eredménye, középen az ingraph eszköz által előállított ekvivalens GRA kifejezés, alul a GRA kifejezésből általunk előállított SQL lekérdezés látható. Az alábbi nulláris GRA művelet a get-vertices, amely a csúcsok azonosítóit és szükséges tulajdonságait gyűjti egy relációba.

Példa. *A diákok azonosítójának és nevének kigyűjtése*

```
MATCH (p:Person:Student)
RETURN p, p.name
```

p	p.name
a	Alice

$$\left(\bigcirc^{\text{Person,Student}}_{p,p.\text{name}}\right)$$

```
1 SELECT vertex_id AS "p",
2       (SELECT "value" FROM vertex_property
3        WHERE parent = vertex_id AND key = 'name') AS "p.name"
4 FROM vertex
5 WHERE NOT EXISTS(VALUES ('Person'), ('Student')
6                  EXCEPT ALL
7                  SELECT name FROM label WHERE parent = vertex_id)
```

A minta elején és a GRA kifejezésben a csomópontot a későbbiekben azonosító változónév (`p`) látható. A gráfban a csúcsok rendelkezhetnek címkékkel, amelyekkel a csúcsok kategorizálhatóak. A mintában, illetve a GRA kifejezésben felsorolt címkék a csúcsra vonatkozó feltételeket írják le, ha vannak, azaz a kívánt csúcsoknak a felsorolt összes címkével rendelkeznie kell. Ezeket a feltételeket SQL-ben az 5-7. sorok írják le: a `VALUES` kulcsszó egy az elvárt címkékből álló egyoszlopos relációt képez, amelyből az `EXCEPT ALL` kivonja a 7. sorban lévő allekérdezés által előállított, az éppen vizsgált csúcs címkéit tartalmazó relációt. Így az eredményül kapott reláció valóban akkor lesz üres, ha az összes címkének mint feltételnek megfelel az adott csúcs. (Üres címkehalmaz esetén az összes csúcsot tartalmazza az eredmény.) A csúcsok tulajdonságokkal rendelkezhetnek, amelyet `változó.tulajdonság` alakban lehet elérni. A szükséges tulajdonságok a GRA-ban alsó indexben láthatóak. SQL-ben minden tulajdonságra egy-egy allekérdezést fogalmazunk meg, amely a csúcs azonosítója és tulajdonság neve alapján visszaadja a tulajdonság értékét. (2-3. sor)

3.2.3. Élek kigyűjtése (get-edges)

A következő lekérdezés kigyűjti az összes olyan diákot (`s`), aki érdeklődik valamilyen téma (`t`) iránt. A reláció felsorolja az összes érdeklődő diákot az összes érdeklődése mellett, az érdeklődést jelentő él azonosítóját (`i`), az érdeklődés szintjét (`i.level`), a téma csúcs azonosítóját (`t`) és a téma nevét (`t.topic`). A GRA-ban a felső sorban rendre az él forrás csúcsának elvárt címkéi, az él lehetséges típusai és a cél csúcs elvárt címkéi szerepelnek. Az alsó sor a hozzájuk tartozó változóneveket és tulajdonságokat tartalmazza. Az éltípusok halmaza a lehetséges típusokat jelöli, azaz az összes él szerepel az eredményben, amelynek a típusa a felsoroltak között van. Üres típushalmaz esetén nincs megkötés az élek típusára.

Példa. *A diákok, érdeklődési köreik és az érdeklődés szintjének kigyűjtése*

```
MATCH (s:Student)-[i:INTEREST]->(t)
RETURN s, i, i.level, t, t.topic
```

s	i	i.level	t	t.topic
a	2	4	c	Neofolk

$$\left[\text{Student}_{s} \bigcirc\!\!\xrightarrow[i,i.\text{level}]{\text{INTEREST}} \bigcirc_{t,t.\text{topic}}\right]$$

```
1  SELECT "from" AS "s", edge_id AS "i", "to" AS "t",
2         (SELECT "value" FROM edge_property
3          WHERE parent = edge_id AND key = 'level') AS "i.level",
4         (SELECT "value" FROM vertex_property
5          WHERE parent = "to"    AND key = 'topic') AS "t.topic"
6  FROM edge
7  WHERE type IN ('INTEREST') AND
8        NOT EXISTS(VALUES ('Student')
9                   EXCEPT ALL
10                  SELECT name FROM label WHERE parent = "from")
```

SQL-ben az 1. sor kiválasztja az edge táblából az él és a csúcsainak az azonosítóit. A 2-5. sor a szükséges él- vagy csúcstulajdonságok értékét adja vissza. A 7. sor vizsgálja, hogy az aktuális él típusa szerepel-e az elvárt típushalmazban. Üres típushalmaz esetén itt nincs feltétel. A 8-10. sor a forrás csúcs címkéit vizsgálja, hogy az összes elvárt címkével rendelkezik-e.

A get-edges műveletnek létezik irányítatlan változata is, amely az él irányát figyelmen kívül hagyja. A művelet az alábbi átírási szabállyal két irányított művelet uniójára bontható. [30] Az SQL algebrában csak az irányított get-edges szerepel, így az irányítatlan változat két irányított műveletből áll össze.

$$\left[\text{L1}_{v} \bigcirc\!\!\xleftrightarrow[e]{\text{T}} \bigcirc_{w}\text{L2}\right] \equiv \left[\text{L1}_{v} \bigcirc\!\!\xrightarrow[e]{\text{T}} \bigcirc_{w}\text{L2}\right] \cup \left[\text{L2}_{w} \bigcirc\!\!\xrightarrow[e]{\text{T}} \bigcirc_{v}\text{L1}\right]$$

3.2.4. Vetítés és szűrés

Az alábbi lekérdezés a 30 év alatti személyek neveit adja vissza és kiszűri az ismétlődéseket. A get-vertices visszaadja a személyeket és a további műveletekhez szükséges tulajdonságaikat. A szelekció (selection) csak a feltételnek megfelelő sorokat tartja meg. A projekció (projection) csak a felsorolt oszlopokat tartja meg és opcionálisan átnevezi ezeket. A duplikátumszűrés (duplicate-elimination) a relációban előforduló ismétlődő sorokból csak egyet-egyet tart meg.

Példa. *A 30 év alatti személyek nevei (ismétlődések nélkül)*

```
MATCH (p:Person)
WHERE p.age < 30
RETURN DISTINCT p.name AS name
```

name
Alice

$$\delta\Big(\underbrace{\pi_{p.\mathsf{name}/name}}_{\mathsf{q2}}\ \underbrace{\sigma_{p.\mathsf{age}<30}}_{\mathsf{q1}}\ \underbrace{\left(\bigcirc^{\mathsf{Person}}_{p,p.\mathsf{age},p.\mathsf{name}}\right)}_{\mathsf{q0}}\Big)$$

```
1  WITH q0 AS ( -- GetVertices
2    SELECT vertex_id AS "p",
3           (SELECT value FROM vertex_property
4            WHERE parent = vertex_id AND key = 'age') AS "p.age",
5           (SELECT value FROM vertex_property
6            WHERE parent = vertex_id AND key = 'name') AS "p.name"
7    FROM vertex
8    WHERE NOT EXISTS(VALUES ('Person')
9                     EXCEPT ALL SELECT name FROM label WHERE parent = vertex_id)),
10 q1 AS ( -- Selection
11   SELECT * FROM q0 WHERE "p.age" < 30),
12 q2 AS ( -- Projection
13   SELECT "p.name" AS "name" FROM q1)
14 -- DuplicateElimination
15 SELECT DISTINCT * FROM q2
```

Több GRA művelet összefűzésére SQL-ben a `WITH` konstrukciót alkalmazzuk, amely névvel ellátott lekérdezéseket (CTE) hoz létre. Minden SQL algebrabeli művelet egy CTE lekérdezésre képződik le, amely hivatkozik a gyerek műveletek eredményeit előállító korábbi CTE-kre. A lekérdezések összefűzésénél megszorítást jelent, hogy az SQL `INSERT` művelet támogatásához, amennyiben ezután további művelet is előfordul, kizárólag `WITH`-tel írt lekérdezéseket lehet használni. Az 1-9. sor a már bemutatott get-vertices műveletet, a 10-11. sor a szelekciót, a 12-13. sor a projekciót tartalmazza. Az ismétlődő sorok kiszűrését a 15. sor valósítja meg.

A projekció és szelekció műveletek során előfordulhatnak olyan függvények, konstrukciók, amelyeknek nincs SQL-beli megfelelője vagy eltérő szintaxissal rendelkezik. Ezeket a függvényeket a megfelelőikre való cserével, illetve tárolt eljárások segítségével valósítottam meg. Ezek a függvények, konstrukciók az alábbiak: `CASE`, `length`, `exists`, `collect`, `labels`, `startNode`, `endNode`, `type`, típuskonverziók.

3.2.5. Csoportosítás és kibontás

A következő lekérdezés az emberek által beszélt nyelveket sorolja fel és mellettük tömbben listázza a nyelvet beszélők neveit. A get-vertices visszaadja az embereket, a projekció átnevezést végez.

ω jelöli a kibontás műveletét (unwind). A speaks tulajdonság tárolja az egyes emberek által beszélt nyelvek listáját. Az unwind művelet minden sort annyiszor másol le, ahány eleme van a kibontandó listának (p.speaks), és kiegészíti ezeket a sorokat *lang* nevű oszloppal, ahol a lista egy-egy elemét, azaz az adott ember által beszélt nyelveket sorolja fel.

γ jelöli a csoportosítás műveletét (grouping). A művelet felső indexben szereplő egy vagy több oszlop szerint csoportosítja a sorokat, majd minden sorhoz az alsó indexben jelölt értékeket adja eredményül. A példában a lang oszlop (nyelv) szerint csoportosítja a sorokat és minden csoporthoz a nyelvet és a name oszlopban szereplő csoportbeli értékek listáját állítja elő a `collect` függvény segítségével.

Példa. *A beszélt nyelvek és a beszélőik nevei*

```
MATCH (p:Person)
WITH p, p.name AS name
UNWIND p.speaks AS lang
RETURN lang, collect(name) AS speakers
```

lang	*speakers*
en	[Alice, Bob]
de	[Bob]

$$\gamma^{lang}_{lang,\texttt{collect}(name)/speakers}\Big(\underbrace{\omega_{p.\mathsf{speaks}\Rightarrow lang}}_{\mathsf{q2}}\Big(\underbrace{\pi_{p,p.\mathsf{speaks},p.\mathsf{name}/name}}_{\mathsf{q1}}\Big(\underbrace{\circ^{\mathsf{Person}}_{p,p.\mathsf{speaks},p.\mathsf{name}}}_{\mathsf{q0}}\Big)\Big)\Big)$$

```
1 WITH q0 AS ( /* GetVertices: (p:Person) | attributes: p.speaks, p.name */ ),
2 q1 AS ( /* Projection: p, p.speaks, p.name AS name */ ),
3 q2 AS ( -- Unwind
4   SELECT "p", "name", unnest("p.speaks") AS "lang"
5   FROM q1)
6 -- Grouping
7 SELECT "lang", array_agg("name") AS "speakers"
8 FROM q2
9 GROUP BY "lang"
```

Az SQL lekérdezésben az 1-2. sor a csúcsokat adja vissza és elvégzi az átnevezéseket. A 4. sorban szereplő unwind művelet megfelel a PostgreSQL `unnest` függvényének, amely képes tömbök kibontására. Azonban a tulajdonságok értékeit JSON típusú oszlopban tároljuk, ezért tárolt eljárásra van szükség, amely egy JSON-beli tömbön elvégzi az unwind műveletet. A grouping műveletet a 7-9. sor végzi. A `SELECT` kulcsszó után soroljuk fel a csoportonként előállítandó értéket, a `GROUP BY` rész tartalmazza a csoportosítási feltételeket. A Cypherbeli `collect` függvénynek a PostgreSQL `array_agg` függvénye felel meg, amely a csoport értékeit tömbben aggregálja.

3.2.6. Természetes illesztés

Az alábbi lekérdezés Alice barátainak a barátait sorolja fel (Alice-t nem ideértve). A bal oldali get-edges művelet visszaadja a p `Person` címkéjű csúcsból `KNOWS` élen (az él irányát figyelmen kívül hagyva) elérhető csúcsokat. A szelekció szűrést végez, így csak az Alice nevű csúcsból induló vagy oda érkező élek maradnak a relációban. A második get-edges felsorolja az összes `KNOWS` típusú élt irányítatlan módon, azaz egyszer az f változóba kerül a forrás csúcs és a *foaf* változóba a cél csúcs, másodszor fordítva.

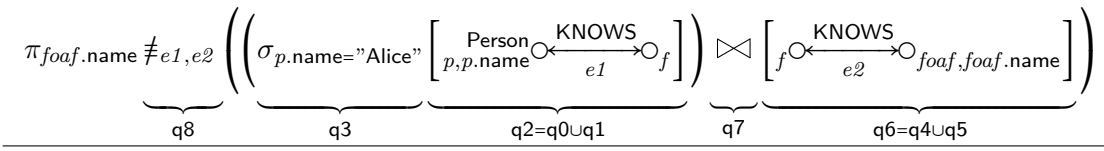
A $\bowtie$ jellel jelölt természetes illesztés (natural join) az azonos nevű oszlopok mentén kapcsolja össze a két relációt, így csak az f-ben érintkező élek maradnak, tehát csak a kiinduló p csúcsból két egymásutáni élen elérhető *foaf* csúcsokat tartalmazó sorok maradnak a relációban.

Ekkor még lehetséges, hogy Alice is az eredményhalmaz része, ha egy ismerősébe menő élen, majd ugyanezen az élen vissza eljutunk hozzá. Az egy `MATCH` blokkon belüli élekre a Cypher nyelv szemantikája az élek egyediségét követeli meg, így ezt az esetet ki kell szűrni. A $\not\equiv$ jellel jelölt all-different művelet vizsgálja a felsorolt élek egyediségét.

Példa. *Alice barátainak barátai*

```
MATCH (p:Person {name: 'Alice'})<-[e1:KNOWS]->(f)<-[e2:KNOWS]->(foaf)
RETURN foaf.name
```

foaf.name
∅




```
1  WITH q0 AS ( -- GetEdges: (p:Person)-[e1:KNOWS]->(f) | attributes: p.name
2    SELECT "from" AS "p", edge_id AS "e1", "to" AS "f",··· AS "p.name" FROM edge ···),
3  q1 AS ( -- GetEdges:      (p:Person)<-[e1:KNOWS]-(f) | attributes: p.name
4    SELECT "from" AS "f", edge_id AS "e1", "to" AS "p",··· AS "p.name" FROM edge ···),
5  q2 AS ( -- UnionAll: q0 ∪ q1
6    SELECT ··· FROM q0 UNION ALL SELECT ··· FROM q1),
7  q3 AS ( -- Selection: p.name = 'Alice'
8    SELECT * FROM q2 WHERE ("p.name" = 'Alice')),
9  q4 AS ( -- GetEdges: (f)-[e2:KNOWS]->(foaf) | attributes: foaf.name
10   SELECT "from" AS    "f", edge_id AS "e2", "to" AS "foaf",··· FROM edge ···),
11 q5 AS ( -- GetEdges: (f)<-[e2:KNOWS]-(foaf) | attributes: foaf.name
12   SELECT "from" AS "foaf", edge_id AS "e2", "to" AS    "f",··· FROM edge ···),
13 q6 AS ( -- UnionAll: q4 ∪ q5
14   SELECT ··· FROM q4 UNION ALL SELECT ··· FROM q5),
15 q7 AS ( -- Join
16   SELECT "left"."p", "left"."p.name", "left"."e1", "left"."f",
17     "right"."e2", "right"."foaf", "right"."foaf.name"
18   FROM q3 AS "left" INNER JOIN q6 AS "right" ON "left"."f" = "right"."f"),
19 q8 AS ( -- AllDifferent
20   SELECT * FROM q7 WHERE is_unique(ARRAY["e1", "e2"]))
21 -- Projection
22 SELECT "foaf.name" FROM q8
```

Az SQL kód 1-14. sora végzi az élek irányítatlan előállítását (unió művelettel) és a szelekciót. A 16-17. sor az illesztés kimenetét adja meg. A 18. sorban szerepel az illesztési feltétel az egyező oszlopnevek alapján. A 20. sor végzi az élek egyediségének vizsgálatát egy saját tárolt eljárás segítségével. Eldobja azokat a sorokat, amelyekben a halmaz ismétlődő élazonosítót tartalmaz.

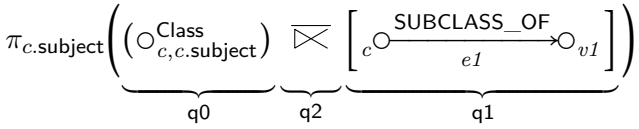
3.2.7. Antijoin

Az alábbi lekérdezés a szülőkategória nélküli kategóriákat sorolja fel, azaz azokat a kategóriákat (`Class` címkéjű csúcsok), amelyekből nem megy ki `SUBCLASS_OF` típusú él. Ehhez a get-vertices művelet előállítja az összes kategóriát, illetve a get-edges művelet az összes `SUBCLASS_OF` típusú élt. A $\overline{\bowtie}$ jellel jelölt antijoin művelet eldobja az összes bal relációból származó csúcsot, amelyhez létezik illeszkedés a jobb relációban (létezik él), és csak azokat tartja meg az eredményben, amelyek nem illeszkednek. Ezzel a művelettel lehetséges negatív mintákat felírása.

Példa. *Kategóriák szülő nélkül*

```
MATCH (c:Class)
WHERE NOT (c)-[:SUBCLASS_OF]->()
RETURN c.subject
```

c.subject
Art




```
1 WITH q0 AS ( /* GetVertices: (c:Class) | attributes: c.subject */ ),
2 q1 AS ( /* GetEdges: (c)-[e1:SUBCLASS_OF]->(v1) */ ),
3 q2 AS ( -- AntiJoin
4   SELECT * FROM q0 AS "left"
5   WHERE NOT EXISTS(
6     SELECT 1 FROM q1 AS "right"
7     WHERE "left"."c" = "right"."c"))
8 -- Projection
9 SELECT "c.subject" FROM q2
```

Az SQL kód 4. sora választja ki az antijoin reláció oszlopait, azaz a bal oldali reláció összes oszlopát. A 6-7. sorban szereplő allekérdezés annyi sort állít elő, ahány jobb oldali sor illeszkedik a vizsgált bal oldali sorhoz. Ha ilyen nem létezik, akkor a sor az eredmény része.

3.2.8. Tranzitív illesztés

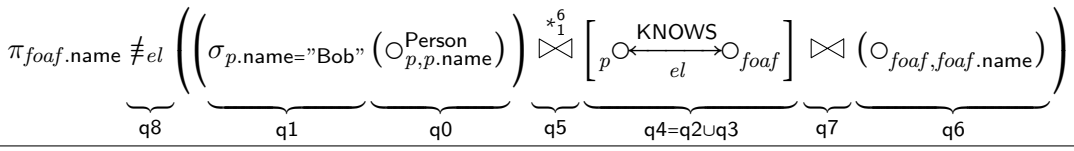
A következő lekérdezés Bobtól kiindulva, a Karinthy-féle hat lépés távolsággal elérhető embereket sorolja fel. Azaz azokat az embereket, akik elérhetők Bobtól kiindulva legalább 1 és legfeljebb 6 db. `KNOWS` típusú, kapcsolódó él irányítatlan (ismétlés nélküli) felhasználásával.

A get-vertices és a szelekció Bob csúcsát adja vissza. A $*^{\text{up}}_{\text{low}}\bowtie$ jellel jelölt tranzitív illesztés (transitive join) művelet segítségével lehet az útvonal-kifejezéseket leírni. [30] A transitive join művelet a bal oldali reláció elemeiből indul ki, ezeket az egyező változónév (p) mentén egészíti ki illeszkedő élekkel, majd ezen élek végpontjait kiindulásként felhasználva készít útvonalakat. Az élekre vonatkozó egyediség feltétel miatt egy él legfeljebb egyszer szerepelhet egy ilyen útvonalban. Az útvonalak hosszára alsó és felső korlát adható meg. Így kapjuk meg a legalább 1, legfeljebb 6 lépésben elérhető embereket (*foaf*), illetve a felhasznált éleket tartalmazó listát (*el*). Az útvonal végéhez illesztjük a jobb get-vertices művelet eredményét, hogy megkapjuk a *foaf*.name tulajdonság értékét. Az él egyediséget biztosító all-different művelet nem változtatja a relációt, mivel már a transitive join is biztosította az egyediséget.

Példa. *Bobtól legfeljebb 6 lépésben elérhető emberek*

```
MATCH (p:Person {name: 'Bob'})<-[el:KNOWS*1..6]->(foaf)
RETURN foaf.name
```

foaf.name
Alice




```
1  WITH q0 AS ( /* GetVertices: (p:Person) | attributes: p.name */ ),
2  q1 AS ( /* Selection: p.name = 'Bob' */ ),
3  ...
4  q4 AS ( /* GetEdges: (current_from)<-[current_edge:KNOWS]->(current_to) */ ),
5  q5 AS ( -- TransitiveJoin
6    WITH RECURSIVE recursive_table AS (
7      SELECT "p"               AS start_vertex,
8             ARRAY[]::BIGINT[] AS edge_list,
9             "p"               AS end_vertex,
10            "p.name"
11     FROM q1
12     UNION ALL
13     SELECT start_vertex,
14            (edge_list || current_edge),
15            current_to AS end_vertex,
16            "p.name"
17     FROM q4 INNER JOIN recursive_table
18              ON "current_edge" <> ALL (edge_list) AND
19                 end_vertex = current_from         AND
20                 array_length(edge_list) < 6)
21   SELECT start_vertex AS "p",
22          edge_list    AS "el",
23          end_vertex   AS "foaf",
24          "p.name"
25   FROM recursive_table
26   WHERE array_length(edge_list) >= 1),
27 q6 AS ( /* GetVertices: (foaf) | attributes: foaf.name */ ),
28 q7 AS ( /* Join: q5 ⋈ q6 */ ),
29 q8 AS ( -- AllDifferent
30   SELECT * FROM q7)
31 -- Projection
32 SELECT "foaf.name" FROM q8
```

Az SQL kód 1-4. sora végzi Bob csúcsának, illetve a KNOWS élek irányítatlan előállítását. A transitive join műveletet rekurzív SQL lekérdezésre képezzük le. A 7-10. sorok végzik a 0 hosszú útvonalból álló kezdeti reláció előállítását. Ebben az esetben a kiindulási és a cél csúcs megegyezik a p csúccsal, és üres tömb alkotja az éllistát. Ezt a relációt a 13-20. sorban bővítjük egy éllel a q4 relációból, ha az él még nem szerepelt az aktuális útvonalban (18. sor), az útvonal végpontjára illeszkedik (19. sor) és az útvonal nem hosszabb a felső korlátnál (20. sor). Ekkor az él másik csúcsa lesz az útvonal végpontja (15. sor). Ezt a bővítést folytatjuk, amíg fixpontig nem jutunk. Ekkor elvégezzük a szükséges átnevezéseket, és kiszűrjük a már felesleges, túl rövid útvonalakat (26. sor).

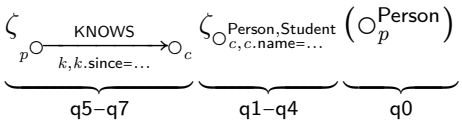
Külső illesztés leképzésére az F.1. szakasz ad példát.

3.2.9. Létrehozás művelet (create)

A olvasás műveletek mellett a fordító támogatja a létrehozás művelet leképzését is. Az alábbi lekérdezés felsorolja az összes személyt, majd mindegyikhez létrehoz egy új ismerőst (azonos névvel). ζ jelöli a létrehozás műveletét (create). A jobb oldali create minden sorhoz a q0 relációból létrehoz egy új csúcsot, és beállítja a tulajdonságait, illetve kiegészíti az eredményt az új csúcsok azonosítóit tartalmazó oszloppal. A bal oldali create létrehoz egy új élt a megadott csúcsok között, beállítja a tulajdonságait és az azonosítókat továbbadja.

Példa. *Új személyek és ismeretségek létrehozása*

```
MATCH (p:Person)
CREATE (p)-[k:KNOWS {since: 2018}]->(c:Person:Student {name: 'Carol'})
```

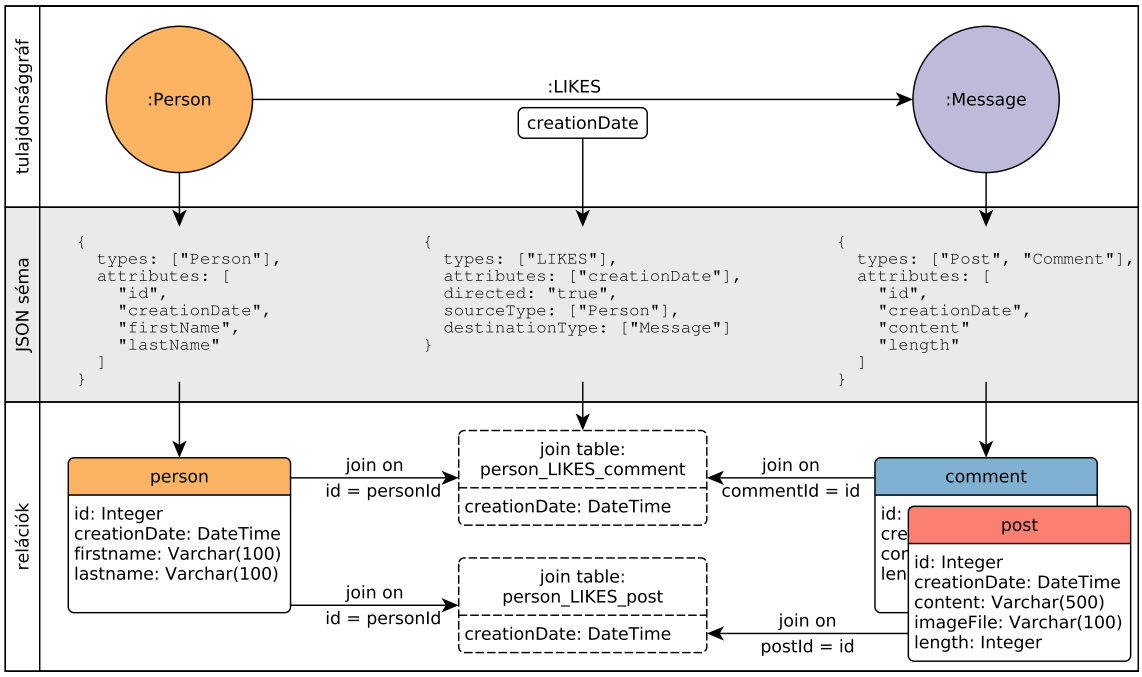



```
1  WITH q0 AS ( /* GetVertices: (p:Person) */ ),
2  q1 AS ( -- GenerateId
3    SELECT *, nextval('vertex_seq') AS "c" FROM q0),
4  q2 AS ( -- InsertVertex
5    INSERT INTO vertex SELECT "c" AS vertex_id FROM q1),
6  q3 AS ( -- InsertLabels
7    INSERT INTO label SELECT q1."c" AS parent, labels.l AS name
8                      FROM q1, (VALUES ('Person'), ('Student')) AS labels(l)),
9  q4 AS ( -- InsertVertexProperty
10   INSERT INTO vertex_property
11     SELECT "c" AS parent, 'name' AS key, 'Carol' AS value FROM q1),
12 q5 AS ( -- GenerateId
13   SELECT *, nextval('edge_seq') AS "k" FROM q1),
14 q6 AS ( -- InsertEdge
15   INSERT INTO edge
16     SELECT "k" AS edge_id, "p" AS "from", "c" AS "to", 'KNOWS' AS type FROM q5),
17 q7 AS ( -- InsertEdgeProperty
18   INSERT INTO edge_property
19     SELECT "k" AS parent, 'since' AS key, 2018 AS value FROM q5)
20 SELECT "p", "k", "c" FROM q5
```

SQL-ben az új csúcsok és élek azonosítóinak előállításához számszekvenciákat használunk (3., 13. sor), amelyek növekvő azonosítókkal látják el az új csúcsokat és éleket. A create művelet egy új oszlopot eredményez a relációban, amely az új elemek azonosítóját tartalmazza. A 5. sor az új csúcsazonosítókat, majd 7-8. sor a csúcs címkéit tárolja el. Az élek esetében az él eltárolását a 15-16. sor mutatja be. A csúcs- és éltulajdonságokat a 10-11. és a 18-19. sorokban rögzítjük.

3.3. Leképzés kiterjesztése adott séma használatára

Ahhoz, hogy a fordító támogassa gráflekérdezések meglévő relációs adatbázisokon való futtatását, a fordítót kiegészítettük a Cytosm (Cypher-to-SQL mapping) projekt gTop sémaleírójának [27] támogatásával. A gTop (graph topology) definiálja a tulajdonsággráf



3.5. ábra. Példa a gTop sémaleíróra ([27] alapján)

sémáját, az előforduló csúcsokat, éleket és ezek tulajdonságait, valamint azt, hogy ezek milyen relációs adatbázisbeli tábláknak felelnek meg.

A 3.5. ábra felső részében látható a tulajdonsággráf sémája ábrázolva, amelyet a középső részben leírt JSON fájl ír le. A gTop séma leírja az egyes csúcsoknak megfelelő táblákat, illetve az éleknek megfelelő kapcsolótáblákat. Emellett több-egy kapcsolatok esetén támogatja, hogy az egyik csúcsnak megfelelő tábla tartalmazzon idegen kulcsot egy másik táblára, ami a gráfban élként jelenik meg: például szülőkategória tárolása a kategóriák táblájában. Továbbá lehetséges oszlopokra vonatkozó megszorításokat adni: például típust tartalmazó cella alapján szűrni, hogy milyen címke kerül az adott csúcsra.

3.3.1. Nulláris műveletek tetszőleges séma felett

A gráfos adatmodellben a csúcsok és az élek (külön-külön) egyedi azonosítóval rendelkeznek, ami a műveletvégzések alapját képezi. Ez relációs adatbázisokban rendszerint hiányzik, csak az egyes táblákon belül rendelkeznek egyedi azonosítóval a sorok. Ennek érdekében a leképezés során bevezetünk egy csúcstípust, ami a táblabeli azonosítót kiegészíti a tábla nevével, így képezve globálisan egyedi azonosítót. Az élek esetében az SQL-beli azonosítót a forrás és cél csúcs azonosítója jelenti. Mivel különböző típusú élek is mehetnek két csúcs között, ezért az él típusával egészítjük ki az azonosítót. Mivel az SQL algebra esetén csak a nulláris, levél műveletek hoznak be adatot, ezért ezek módosításával valósítható meg a gTop séma támogatása.

A get-vertices művelet során lehetséges, hogy a csúcsok összegyűjtését különböző táblák alapján kell végezni, így a műveletet ezeken a táblákon külön-külön dolgozó get-vertices műveletek uniójával lehet megvalósítani. Az alábbi példán a `Message` címkéjű csúcsok lehetnek a `post` és a `comment` táblában is, így ebből a két táblából kell összegyűjteni a sorokat, amelyeket el kell látni globálisan egyedi azonosítóval.

Példa. *Message címkéjű csúcsok content tulajdonsága*

```
MATCH (m:Message)
RETURN m.content
```

$$\pi_{m.\text{content}} \underbrace{\left(\circ^{\text{Message}}_{m,m.\text{content}}\right)}_{\text{q0–q2}}$$

```
1  WITH q0 AS ( -- GetVertices
2    SELECT make_vertex_id('post', "id") AS "m",
3           "content" AS "m.content"
4    FROM "post"),
5  q1 AS ( -- GetVertices
6    SELECT ROW('comment', "id")::vertex_type AS "m",
7           "content" AS "m.content"
8    FROM "comment"),
9  q2 AS ( -- UnionAll
10   SELECT "m", "m.content" FROM q0
11   UNION ALL
12   SELECT "m", "m.content" FROM q1)
13 -- Projection
14 SELECT "m.content" FROM q2
```

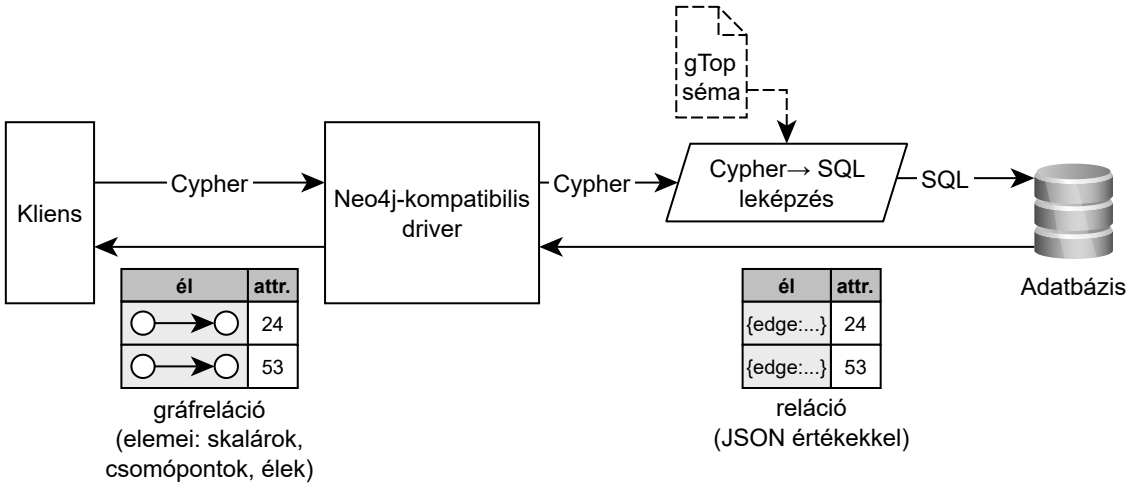
Az alábbi lekérdezés a felhasználók által kedvelt üzenetek tartalmát és az él keletkezési időpontját adja meg. A gTop sémában előforduló összes olyan tábla párra, amelyet egy KNOWS típusú kapcsolótábla köt össze végrehajtjuk a get-edges műveletet. Az élek globális azonosítóját a végpontjaik és az él típusa adják (4. sor). Az élhez végpontjaihoz tartozó tulajdonságok eléréséhez szükséges az adott csúcsot tároló tábla illesztése az él kapcsolótáblájához (9. sor).

Példa. *LIKES típusú élek és végpontjaik*

```
MATCH (p)-[l:LIKES]->(m:Message)
RETURN m.content, l.creationDate
```

$$\pi_{m.\text{content},l.\text{creationDate}} \underbrace{\left[p \circ \xrightarrow[l,l.\text{creationDate}]{\text{LIKES}} \circ^{\text{Message}}_{m,m.\text{content}}\right]}_{\text{q0–q2}}$$

```
1  WITH q0 AS ( -- GetEdges
2    SELECT
3      make_vertex_id('person', edgeTable."personId") AS "p",
4      make_edge_id('LIKES', edgeTable."personId", edgeTable."postId") AS "l",
5      make_vertex_id('post', edgeTable."postId") AS "m",
6      toTable."content" AS "m.content",
7      edgeTable."creationDate" AS "l.creationDate"
8    FROM "person_LIKES_post" edgeTable
9           JOIN "post" toTable ON (edgeTable."postId" = toTable."id")),
10 q1 AS ( /* GetEdges: (p:Person)-[l:LIKES]->(m:Comment)
11               attributes: m.content, l.creationDate */ ),
12 q2 AS ( /* UnionAll: q0 ∪ q1 */ )
13 -- Projection
14 SELECT "m.content", "l.creationDate" FROM q2
```

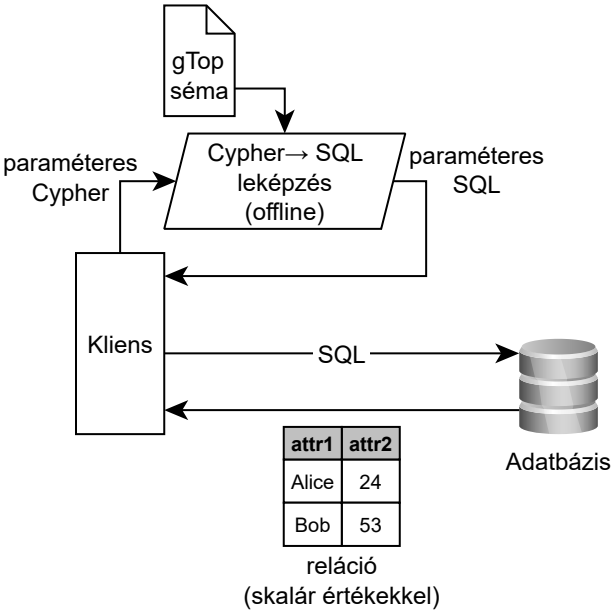


3.6. ábra. A fordító használata Neo4j-kompatibilis driveren keresztül

3.4. A fordító lehetséges felhasználási lehetőségei

Az elkészült fordítót több más rendszerekkel való használata több módon is lehetséges. A fordítóhoz készült egy – a Neo4j gráfadatbázissal kompatibilis – driver, amely segítségével az eszköz gráfadatbázisként használható (3.6. ábra). Ennek során az adatok tárolását a bemutatott generikus sémában vagy – gTop sémaleíró esetén – adott sémában egy PostgreSQL adatbázis valósítja meg. Ennek során ki lehet használni a driver képességeit, azaz nemcsak skalár eredményeket adhatnak a lekérdezések, hanem gráfos adatokat is. Lekérdezhetők a csomópontok, címkéik, tulajdonságaik, valamint az élek, típusok és tulajdonságaik. Ez a felhasználási mód (gTop nélkül) a prototipizálás során hasznos, amikor még nem szükséges a teljesítményszempontokat is figyelembe vevő relációs séma használata.

Meglévő relációs adatbázis mellé a fordító alkalmazásához szükséges a sémát és az azon értelmezett tulajdonsággráfot leíró gTop fájl. Gráflekérdezések futtatásához elegendő, ha a lekérdezések írása után azokat offine módon SQL-re fordítjuk, majd a meglévő rendszerben ezeket az SQL lekérdezéseket használjuk (3.7. ábra). A Cypher lekérdezésekben alkalmazhatók paraméterek, amelyek később, generált SQL lekérdezés felhasználásakor futásidőben lesznek behelyettesítve.



3.7. ábra. A fordító használata meglévő relációs adatbázis esetén

4. fejezet

Teljesítménymérési keretrendszer

A gráfalapú és a relációs adatbázis-kezelők teljesítményének és a használhatóságának összehasonlítása az adatbázis-kezelők fejleszőinek és a felhasználóinak is fontos információk forrása lehet. A fejlesztők megtudhatják, hogy a rendszerük mennyire hatékony a többi adatbázis-kezelőhöz képes, a felhasználók pedig a mérési eredmények révén több információ alapján választhatják ki a számukra megfelelő adatbázis-kezelőt.

A különböző adatbázisok és nyelvek összehasonlítása során szükség volt egy teljesítménymérési keretrendszerre, amely fókuszában a gráfadatbázisok állnak. Kiemelt szempont volt, hogy a keretrendszer lekérdezései általános formában legyenek definiálva, ne pedig egy adott lekérdezési nyelven megírt lekérdezésekkel. Ezen feltétel és a korábbi kontribúcióink alapján a Linked Data Benchmark Council (LDBC)[1] szervezet Social Network Benchmark (SNB) keretrendszerét választottuk, mert ez az elérhető legátfogóbb teljesítménymérési keretrendszer gráf információs rendszerek összehasonlító teljesítménymérésére.

Az LDBC egy európai uniós projekt keretében létrejött szervezet, amelynek célja teljesítménymérési keretrendszerek, munkafolyamatok létrehozása gráf vagy RDF alapú adatbázis-kezelő rendszerekhez, valamint az auditált mérési eredmények publikálása. Jelenleg több nagy cég támogatja a szervezetet[2], mint például az Oracle, IBM, Intel, Neo4j és OpenLink software (a Virtuoso gyártója).

4.1. LDBC Social Network Benchmark

Az LDBC SNB célja a gráf típusú adatbázis-kezelő rendszerek funkciójának széles körű tesztelése. Ehhez a keretrendszer egy szintetikus közösségi háló gráf alapú adatbázisát használja. Az adathalmaz sémája az F.2.1. ábrán látható. A keretrendszer kétféle terhelési profilt tartalmaz. Az úgynevezett *Business Intelligence* [10][3] profil olyan analitikus lekérdezéseket fogalmaz meg, amelyeknek a megválaszolásához az adathalmaz átfogó vizsgálata szükséges, például minden felhasználó üzeneteinek megszámolása. Ilyen lekérdezés például a legaktívabb, vagy éppen a legkevésbé aktív felhasználók megkeresése, vagy a baráti háromszögek megkeresése. Az *Interactive* [31] profil lekérdezései pedig inkább egy, vagy néhány csomóponthoz kapcsolódó adat alapos vizsgálatát követelik meg. Az *Interactive* profil lekérdezései három további csoportba sorolhatóak:

[1] A szervezet honlapja: `http://ldbcouncil.org`

[2] Az aktuális partnercégek listája megtekinthető a szervezet honlapján: `http://ldbcouncil.org/industry/members`

[3] A terhelési profil kidolgozásában részt vett jelen dolgozat egyik szerzője és mindkét konzulense is.

- Komplex lekérdezések (Complex reads, CR), 14 darab: Az adathalmaz számos pontját, általában egy ember ismerőseit és azok ismerőseit, valamint az ő aktivitásukat érintő lekérdezések.
- Rövid lekérdezések (Short reads, SR), 7 darab: Egyszerű, általában öt csomópontnál nem többet érintő lekérdezések.
- Frissítések (Updates, U), 8 darab: Legfejlebb egy csomópont és néhány él beszúrása az adathalmazba.

Ahhoz, hogy a különböző rendszerek teljesítményét átfogóan lehessen elemezni, a keretrendszer támogatja az adathalmaz skálázását is. Az adathalmaz skálázási együtthatóját (scale factor, SF) a CSV[4] formátumban tárolt adathalmaz gigabájtban számolt mérete adja. A keretrendszer által előre konfigurált skálázási együtthatók a következők: 1, 3, 10, 30, 100, 300, 1000. A különböző skálázási együtthatójú adathalmazok méretének szabályozása az adathalmazban szereplő emberek számának megadásával történik, ennek megfelelően generálódik a hálózat többi része. A mérésekhez használt SF1-es adathalmazban 11 ezer, az SF3-asban 27 ezer, az SF10-esben 73 ezer ember szerepel [17].

Az LDBC SNB lekérdezéseinek specifikációja tartalmaz egy szemléltető ábrát, és a lekérdezés szabadszöveges megfogalmazását is. Ennek köszönhetően nem csak az adatbáziskezelő rendszerek, de a lekérdezési nyelvek összehasonlítására is alkalmas a keretrendszer.

4.1.1. A teljesítménymérés munkafolyamata

A keretrendszerhez munkafolyamatát a 4.1. ábra mutatja be. A munkafolyamatban négy típusú összetevő van: *(1)* a keretrendszerhez kapcsolódó szoftvermodulok és a lekérdezések forráskódja, *(2)* eljárás, *(3)* emberek által definiált adat, *(4)* generált adat. Az összetevőket továbbá két csoportba sorolhatjuk az alapján, hogy az elkészítésük vagy végrehajtásuk a keretrendszer fejlesztőinek (LDBC SNB munkacsoport) vagy felhasználóinak (Adatbáziskezelők fejlesztői, felhasználói) a feladata.

A keretrendszer fejlesztőinek feladatai:

- A *lekérdezések specifikációnak*[5] elkészítése és karbantartása [17]
- Az adathalmazokat és lekérdezések paramétereit *generáló alkalmazás* (DATAGEN)[6] elkészítése
- A megvalósítások ellenőrzését és a teljesetménymérést végző *alkalmazás keretrendszer* (DRIVER)[7] elkészítése
- A *referenciaimplementáció* elkészítése
- Az egyes lekérdezések elvárt eredményét tartalmazó adathalmaz, azaz a *referencia validációs adathalmaz* elkészítése

Ezen részfolyamatok elkészülte után a felhasználók elkezdhetik megvalósítani a saját összetevőket. A DATAGEN modullal a felhasználók generálhatnak adott méretű adathalmazt, illetve a hozzá tartozó lekérdezés paramétereket. A dolgozat készítése során a PostgreSQL implementációt tekintettük referencia implementációnak.

A felhasználók által végzett munkát további két csoportba lehet osztani: az implementáció ellenőrzése és a teljesítménymérés. Mindkettőhöz szükséges az

[4] Comma-Separated Values

[5] `https://github.com/ldbc/ldbc_snb_docs`

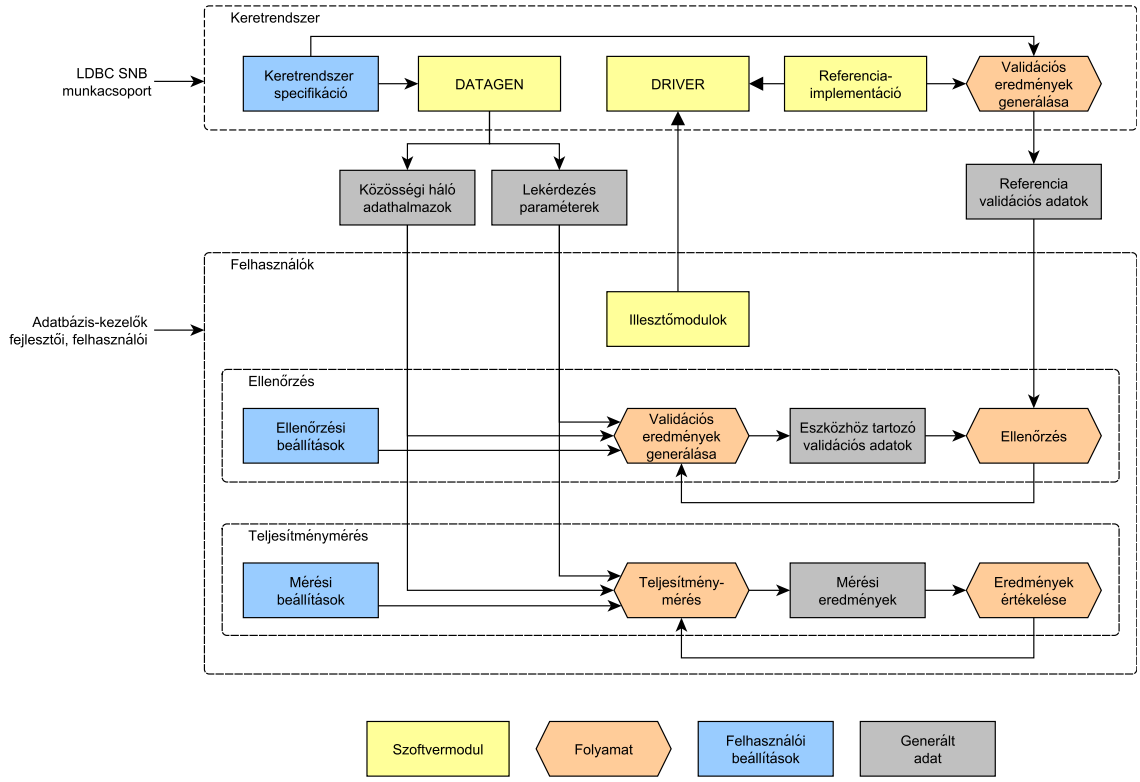
[6] `https://github.com/ldbc/ldbc_snb_datagen`

[7] `https://github.com/ldbc/ldbc_snb_driver`

adott adatbázis-kezelőhöz tartozó szoftvermodulok és lekérdezések implementációja. A szoftvermodulok közé tartoznak az adatok betöltését, a lekérdezések futtatását végző és az eredményeket a DRIVER számára feldolgozható formátumra konvertáló modulok. Az ellenőrzés lépései ezek után az alábbiak:

- Az *ellenőrzési beállítások* (például mennyi lekérdezést futtason a keretrendszer) és a *lekérdezés paraméterek* alapján az *eszközhöz tartozó validációs adatok* (a lekérdezések eredményeinek) előállítása
- Az eredmények összevetése a referncia validációs adathalmazzal: ha az eredmény megegyezik a referencia validációs adathalmazzal, akkor következhet a teljesítménymérés, ellenkező esetben a hibákat javítani kell és újra ellenőrizni az implementációt

A felhasználói beállítások sok finomhangolási lehetőséget biztosítanak, például mennyi végrehajtási szálon fusson egyidőben az ellenőrzés, összesen mennyi lekérdezést végezzen el, illetve az egyes lekérdezések ellenőrzését egyenként lehet engedélyezni vagy tiltani, stb..



4.1. ábra. A teljesítménymérési keretrendszer munkafolyamata

Ha az implementáció átment az ellenőrzésen, akkor a teljesítménymérés a következő lépés. A teljesítménymérés lépései hasonlóak az ellenőrzés lépéseihez:

- A *mérési beállítások* és a lekérdezés paraméterek alapján a teljesítményérés futtatásával a *mérési eredmények* elkészítése
- Az eredmények értékelése és esetleges új mérés indítása (például nagyobb méretű adathalmazon)

4.2. Teljesítménymérési keretrendszer bővítése

A dolgozat készítése során az LDBC SNB-t bővítettük az *Interactive* terhelési profil SPARQL és Cypher implementációjával, ezzel bővítve ki a keretrendszerrel lemérhető adatbázis-kezelők halmazát. Ezen felül implementáltunk egy Gremlin-t támogató adatbázis-kezelőkhöz használható adatbetöltő alkalmazást. A fejezet további részében ha nem jelöljük külön, akkor a terhelési profil alatt az *Interactive* terhelési profilt értjük.

4.2.1. SPARQL implementációk

A terhelési profil CR és SR lekérdezéseinek két implementációja (az egyik az LDBC SNB implementációja, a másik pedig egy szabadon elérhető implementáció[8] [21]) is létezett, de több probléma is volt velük:

- A lekérdezések csak a Virtuoso adatbázis saját SPARQL dialekutásban voltak elérhetők. A Virtuoso SPARQL dialektusa több olyan konstrukciót tartalmaz, ami az eredeti SPARQL nyelvhez képest tömörebb és kifejezőbb lekérdezések írását teszi lehetővé. Ezek átírása szabványos SPARQL nyelvre nem-triviális lépéseket igényel, és nem is lehetséges minden esetben.
- A lekérdezések specifikációi az implementáció elkészülte óta pontosítva, javítva lettek, így néhány lekérdezés hibássá vált.
- Néhány lekérdezés nem a leghatékonyabb struktúrájú volt.

Ezen okok miatt a lekérdezések teljes újraimplementálása mellett döntöttünk, nem próbáltuk meg a meglévő lekérdezéseket átalakítani. A munka során a specifikáció alapján implementáltunk[9] 27 darab (12 db CR, 7 db SR, 8 db U) lekérdezést SPARQL nyelven.

A CR13 lekérdezéshez szükséges egy legrövidebb út megtalálása, a CR14 lekérdezéshez az összes legrövidebb út megtalálása és súlyozása az érintett csomópontok alapján. Ezeket a funkciókat a szabvány SPARQL nem támogatja, így ezeket a lekérdezéseket nem implementáltuk. Az implementáció elkészítése során *(1)* implementáltuk a 27 darab lekérdezést SPARQL nyelven, *(2)* implementáltuk a lekérdezések végrehajtását és a lekérdezések eredményének konvertálását végző szoftvermodulokat Java nyelven, *(3)* a keretrendszert használva sikeresen végrehajtottuk az implementáció ellenőrzését.

A munkát nehezítette, hogy a lekérdezések specifikációi sok esetben pontosításra, vagy kiegészítésre szorultak, vagy az adathalmaz régebbi változtatásai miatt frissíteni kellett azokat. Tehát a terhelési profil lekérdezéseinek implementálása közben a terhelési profilhoz kapcsolódó dokumentációkat is frissítettük, kiegészítettük, így megkönnyítve más felhasználóknak a saját lekérdezéseik és moduljaik elkészítését.

4.2.2. Cypher implementációk

A terhelési profilhoz nem volt elkészítve a Cypher implementáció, azonban egy másik nyílt forráskódú projektben[10] a lekérdezések jelentős része implementálásra került. Bár ezek a lekérdezések sem voltak hibátlanok, minőségük jelentősen jobb volt, mint a SPARQL-ben létező lekérdezéseké, így ezeket fel tudtuk használni a végleges implementáció[11] elkészítéséhez. A SPARQL implementációhoz hasonló módon itt is három részre

[8] A Waterloo-i Egyetemen fejlesztett implementáció: `https://github.com/anilpacaci/ldbc_snb_implementations`

[9] SPARQL implementáció: `https://github.com/ldbc/ldbc_snb_implementations/pull/51`

[10] A Stanford Egyetemen fejlesztett implementáció: `https://github.com/PlatformLab/ldbc-snb-impls`

[11] Cypher implementáció: `https://github.com/ldbc/ldbc_snb_implementations/pull/57`

lehet osztani az elvégzett munkát: *(1)* a meglévő lekérdezések implementációjának ellenőrzése és javítása, illetve a hiányzók implementálása Cypher nyelven, *(2)* a szükséges szoftvermodulok implementációja, *(3)* az implementáció ellenőrzése. Fontos megjegyezni, hogy a jelenleg szabványosítás alatt álló openCypher nyelv nem tartalmazza a legrövidebb utak kifejezésére szolgáló nyelvi konstrukciókat (pl. `shortestPath` és `allShortestPath`), így CR13 és CR14 lekérdezések csak Cypher (és nem openCypher) nyelven értelmezhetőek.

4.2.3. Gremlin adatbetöltő

A dolgozat készítése során megpróbálkoztunk a Cypher for Gremlin[12] projekt felhasználásával a keretrendszert kiterjeszteni a Gremlin-t támogató adatbázis-kezelőkre is, azonban nem sikerült minden technikai problémát megoldani a dolgozat elkészültéig. Még a triviális Cypher lekérdezésekből generált Gremlin lekérdezések is igen nagyméretűek, a generált jellegből fakadóan az elnevezések nem intuitívak, így a lekérdezések futtatása, a hibakeresés vagy éppen az eredmények megfelelő formátumra való konvertálása sem triviális feladat. kódrészlet 2.1. kódrészletben szereplő Cypher lekérdezésből az alábbi Gremlin lekérdezést állítja elő a projekt:

```
g.V().as('p').hasLabel('Person').as('p').choose(__.select('p').is(neq('  cypher.
    null')).outE('KNOWS').inV().as('f').hasLabel('Person'), __.select('p').is(neq
    ('  cypher.null')).outE('KNOWS').inV().as('f').hasLabel('Person'), __.
    constant('  cypher.null').as('f')).select('p', 'f').group().by(__.select('p')
    .choose(neq('  cypher.null'), __.choose(__.values('name'), __.values('name'),
     __.constant('  cypher.null')), __.constant('  cypher.null'))).by(__.fold().
    project('p.name', 'count(f)').by(__.unfold().select('p').choose(neq('  cypher
    .null'), __.choose(__.values('name'), __.values('name'), __.constant('
    cypher.null')), __.constant('  cypher.null'))).by(__.unfold().select('f').is(
    neq('  cypher.null')).count())).unfold().select(values)
```

Az implementáció nem teljes körű, de az adatok betöltését végző szoftvermodul funkcionalitását tekintve elkészült, az adatok betöltésére alkalmas.

[12] `https://github.com/opencypher/cypher-for-gremlin`

5. fejezet

Kiértékelés

Annak érdekében, hogy átfogó képet kapjunk az elérhető gráf információs rendszerek és a Cypherről SQL-re leképzett lekérdezések teljesítményéről, egy összetett mérési sorozatot terveztünk és futtattunk. A mérés során öt implementációval végeztünk méréseket:

1. PostgreSQL: a PostgreSQL relációs adatbázis-kezelő rendszer.
2. PGDB: egy tulajdonsággráf alapú gráfadatbázis-kezelő.
3. SDB1: egy szemantikus adatbázis.
4. SDB2: egy szemantikus adatbázis.
5. C2S: a Cypher lekérdezéseket SQL-re fordító eszköz, PostgreSQL-en futtatva.

A három meg nem nevezett adatbázis-kezelő rendszer eredményeit anonimizált módon adjuk közre. Ennek oka, hogy ugyan a lekérdezések minden esetben *validáltak* (azaz helyes eredményeket biztosítanak), nem *auditáltak* (azaz a rendszerek gyártói nem vizsgálták meg az implementációt, így nem garantált, hogy az optimális teljesítményt biztosít).

A dolgozatban csak 12 darab komplex lekérdezés mérésének eredményét ismertetjük, a rövid lekérdezések és a frissítések teljesítménymérése közeljövőbeli terveink között szerepel. A C13-as és C14-es lekérdezések méréseinek eredményét az anonimitás megőrzése miatt nem közöljük, ezeket ugyanis csak néhány rendszer képes kiértékelni.

A teljesítménymérés során az egyes implementációk válaszidejét és skálázhatóságát mértük. Válaszidő alatt a lekérdezés végrehajtásának elkezdésétől a lekérdezésre adott válasz teljes megérkezéséig eltelt időt értjük. A skálázhatóság alatt a válaszidő adathalmaz méretétől függő változását értjük.

5.1. Motiváció

Bár a relációs adatbázis-kezelők a mai napig a legelterjedtebb adatbázis-kezelők[1], az újfajta adatbázis-kezelők változatos funkcionalitásaikkal és lekérdezőnyelveikkel újra és újra megpróbálják felvenni a versenyt velük. A gráf alapú adatbázis-kezelők az utóbbi években jelentős fejlődésen mentek keresztül, ezzel jelentősen növelve népszerűségüket[2]. Ahhoz, hogy eldöntsük, fel tudják-e venni a versenyt a relációs adatbázis-kezelőkkel, szükséges a teljesítményük összehasonlítása is.

A dolgozatban összehasonlított alkalmazások összetettsége bőven meghaladja azt a szintet, hogy a teljesítményüket a forráskódok elemzése alapján össze lehetne hasonlítani.

[1] `https://db-engines.com/en/ranking`

[2] `https://db-engines.com/en/ranking_categories`

Ezért a tudományban már bizonyított módon kísérletek, mérések alapján próbáljuk összehasonlítani őket. Az LDBC fejlesztőinek célja pedig az, hogy a gráf alapú adatbázis-kezelők összehasonlításában az LDBC meghatározó szerepet töltsön be.

A dolgozatban összehasonlított technológiák kiválasztásában leghangsúlyosabb szempont a lekérdezőnyelveik voltak:

- SQL: régóta használt, a legelterjedtebb adatbázis-kezelők nyelve.
- SPARQL: A gráf alapú lekérdezési nyelvek közül a legteljesebb matematikai háttérrel rendelkező, szemantikailag legtisztább lekérdezőnyelv.
- Cypher: A gráf alapú lekérdezési nyelvek között az egyik legnépszerűbb lekérdezési nyelv, véleményünk szerint az egyik legintuitívabb és legkifejezőbb nyelv.

Nyelv	Karakterszám
SQL	10219
SPARQL	18616
Cypher	7992

5.1. táblázat. A különböző nyelveken megírt lekérdezések karakterszáma fehér szóközök (szóközök, tabulátor és újsor karakterek) nélkül

A Cypher kifejezőerejét és tömörségét az 5.1. táblázat is alátámasztja. Az általunk megírt lekérdezések SQL-ben 28%, SPARQL-ben pedig 132%-kal több karaktert tartalmaznak mint a Cypherben írt lekérdezések. A C2S implementáció az új lekérdezőnyelvet próbálja ötvözni a relációs adatbázis-kezelők hatékonyságával.

5.2. Mérési elrendezés

A dolgozat készítése során végzett mérések egy számítógépen történtek az LDBC SNB keretrendszer DRIVER szoftvermoduljának 0.3.1-es verziójának felhasználásával. A számítógép 16 fizikai processzormagot (Intel(R) Xeon(R) Platinum 8167M CPU @ 2.00GHz), és 236 GB memóriát tartalmaz. A háttértár egy SCSI interfészen kapcsolt 128GB-os SSD lemez. Az operációs rendszere 64 bites Ubuntu 18.04.

A PostgreSQL 10.5-ös[3] verzióját használtuk az SQL és C2S implementációk mérésére. A C2S implementációt a PostgreSQL-lel való összevethetőség érdekében az eredeti sémához hasonló sémán futtattuk. Az eltérések a következők:

1. Az irányítatlan `KNOWS` élek eredetileg mindkét irányban szerepeltek az adathalmazban, míg a C2S implementációban csak az egyik irányban. Itt a lekérdezés fejezte ki, hogy az él irányítatlan.
2. A dátum és idő értékeket az openCypher nyelv limitációi miatt számértékekként kezeljük.

A meg nem nevezett adatbázis-kezelők méréséhez az utóbbi 6 hónapban kiadott verziókat használtunk. A DRIVER és a meg nem nevezett adatbázis-kezelők közül a Java nyelvű rendszerek az OpenJDK 1.8.0_181-es verzióján futottak. Az adatbázis-kezelők konfigurációját a dokumentációjuk alapján próbáltuk optimalizálni, azonban az anonimitás miatt a pontos beállításokat nem közölhetjük.

[3](PostgreSQL 10.5 (Ubuntu 10.5-0ubuntu0.18.04) on x86_64-pc-linux-gnu, compiled by gcc (Ubuntu 7.3.0-16ubuntu3) 7.3.0, 64-bit)

Skálázási tényező	SF1	SF3	SF10
PostgreSQL	1000	1000	1000
PGDB	1000	1000	1000
SDB1	1000	1000	1000
SDB2	1000	750	40
C2S	100	100	100

5.2. táblázat. A implementációk mérése során futtatott lekérdezések száma

A lekérdezések a rövid lekérdezésekkel és a frissítésekkel együtt a 4. fejezetben leírt módon kerültek ellenőrzésre az SF1-es adathalmazokon, több mint 13 ezer lekérdezés eredményének összehasonlításával.

A mérés során a lekérdezéseket különböző behelyettesítési paraméterekkel futtatuk az 5.2. táblázat szerinti darabszámban. Annak érdekében, hogy a mérések összideje ne legyen túl nagy, több korlátozást is kellett tennünk. A SDB2 mérése során már SF3 esetében is csökkenteni kellett a lekérdezések számát, illetve az SF10-es adathalmazon egyáltalán nem mértük le az SDB2-t, mert nem futott le egy lekérdezés sem. A C2S esetében pedig mindhárom méretű adathalmazon csak 100 lekérdezést mértünk.

Mindegyik eszköz mérésnél a különböző méretű adathalmazokon legalább 20-szor futott egy lekérdezés (különböző paraméterekkel). A legkevesebbszer a C2S esetében futottak a lekérdezések, de mivel ott csak 5 lekérdezést mértünk a többi implementációval szemben (ahol maximum 14-et), így itt is legalább 20-szor futott egy lekérdezés.

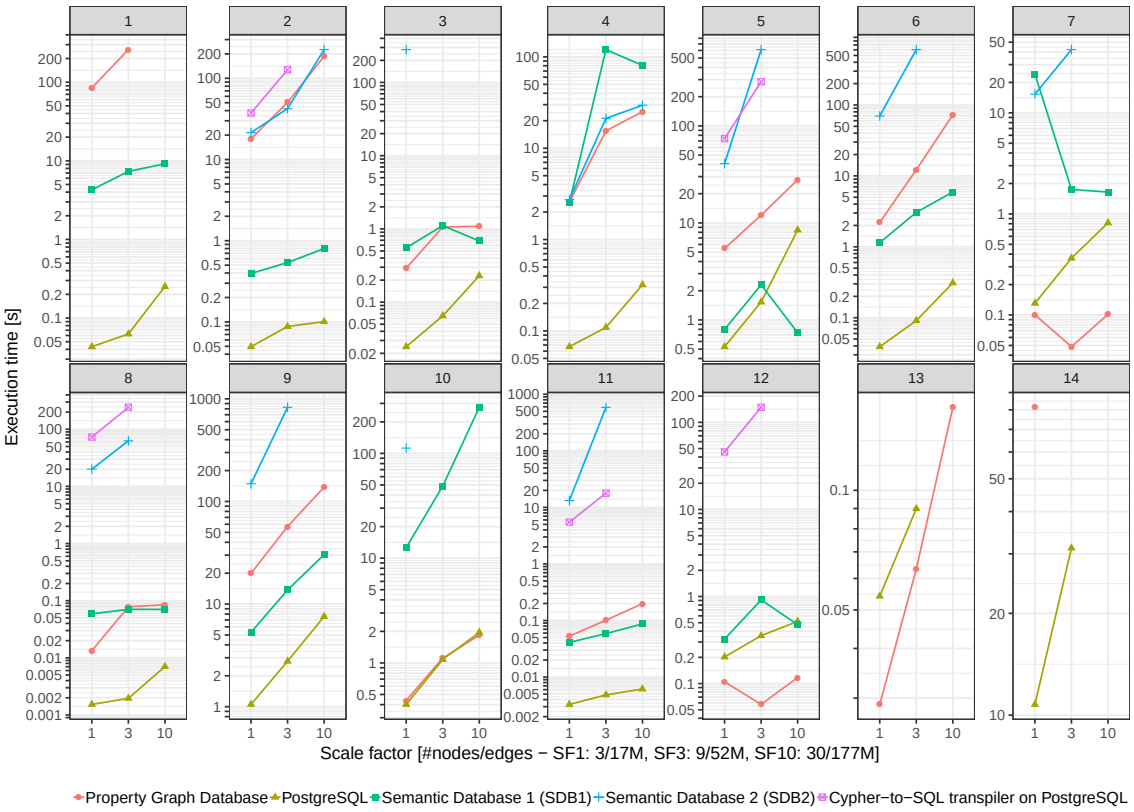
5.3. Eredmények értékelése

A mérési eredményeket az 5.1. ábra tartalmazza. Amint látható, a C2S implementáció 5 lekérdezés leképzését tudta megvalósítani. Ez jobb eredmény, mint a hasonló leképzést végző Cytosm [27] projekt 2 lekérdezése. Az ábrán ugyan nem látható, de a C2S implementáció a CR4-es lekérdezést is át tudta alakítani, azonban lemérni nem tudtuk. A probléma az volt, hogy a PostgreSQL a feltételezéseink szerint a lemezen kezdte tárolni a lekérdezés közben létrehozott köztes relációkat, azonban a lemezen elfogyott a szabad terület. A vizsgált lekérdezési tervek alapján a `WITH` kulcsszóval bevezetett lekérdezések rossz hatással voltak a teljesítményre, mivel az adatbázis a lekérdezések eredményeit materializálta. A PostgreSQL adatbázis a `WITH`-ben szereplő lekérdezések között nem tud optimalizálni. [4] Ennek ellenére néhány esetben egyes adatbázis-kezelőknél jobb teljesítményre volt képes. Az eredmények kis szórása annak tudható be, hogy a materializálás miatt a műveletek időigénye csak kis mértékben függött a lekérdezés paramétereitől.

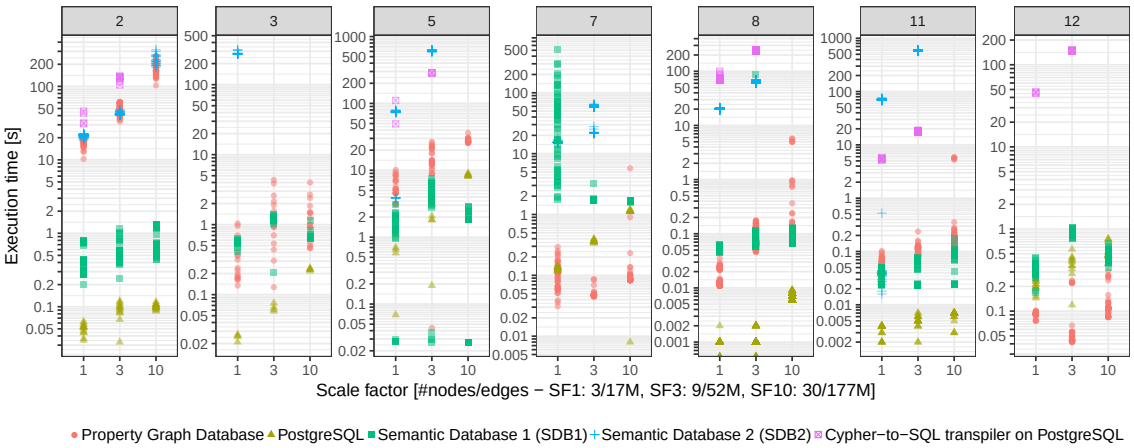
Az egyik legfontosabb eredmény, hogy a lekérdezések többségénél a legjobb eredményt a PostgreSQL érte el, ezzel bizonyítva, hogy a relációs adatbázis-kezelők a közel 50 éves fejlődésnek köszönhetően manapság is a leghatékonyabb adatbázis-kezelők közé tartoznak.

Megfigyelhető, hogy a 3-as, 4-es, 5-ös és 12-es lekérdezéseknél az SDB1 eszköz kisebb átlagos válaszidővel rendelkezik az SF10-es adathalmazon, mint az SF3-as adathalmazon. A lekérdezések specifikációjából [17] kiderül, hogy az említett lekérdezésekben szerepel összetett aggregáció. Az említett lekérdezéseken kívül összetett aggregáció csak a 6-os lekérdezésben van, amelynél azonban nem figyelhető meg ilyen fajta teljesítmény-növekedés. A jelenségre nem sikerült konkrét hipotézist felállítanunk, de az említett komplex aggregációnak köze lehet a jelenséghez.

[4] `https://blog.2ndquadrant.com/postgresql-ctes-are-optimization-fences/`



5.1. ábra. A CR1-CR12-es lekérdezések összesített mérési eredményei



5.2. ábra. A CR2, CR3, CR5, CR7, CR8, CR11 és CR12-es lekérdezések részletes mérési eredményei

A 7-es lekérdezésnél szintén megfigyelhető a teljesítmény javulása a nagyobb adathalmazokon. Abban azonban más jellegű ez a javulás, mert az SF1 adathalmaz után, az SF3-as adathalmaznál figyelhető meg jelentős javulás, míg az SF3 adathalmaz után az SF10-es adathalmaznál csak minimális mértékű.

Néhány kiválasztott lekérdezés részletesebb eredményeit ábrázolja az 5.2. ábra. Megfigyelhető, hogy az eszközök egy adott lekérdezés esetében is nagyságrendekkel eltérő válaszidővel rendelkezhetnek a lekérdezés paramétereitől függően. Kifejezetten látványos a PGDB viselkedése a 7-es lekérdezés esetében, ahol az SF1-es adathalmazon a legjobb és a legrosszabb eredmény között több mint két nagyságrendnyi az eltérés.

Hasonlóan érdemes megvizsgálni a PGDB viselkedését az 5-ös lekérdezés esetében. A mérési eredmények mindhárom méretű adathalmazon két jól elkülöníthető csoportba sorolhatóak: *(1)* az 5 századmásodperc alatti és a *(2)* néhány másodperces válaszidők csoportjába. Előbbiekre lehetséges magyarázat lehetne az, hogy a lekérdezés ebben az esetben üres eredményhalmazt ad vissza, de tapasztalataink szerint ez nem igaz.

Az említett jelenségek többségére sajnos hipotéziseket sem tudtunk felállítani. Ennek egyik oka, hogy a rendszerek belső működéséről nincs információnk, ezért nem tudunk következtetni a lehetséges okokra. A C2S implementációval kapcsolatos jelenségre viszont egy erős hipotézist tudtunk felállítani, mivel annak működését ismerjük.

6. fejezet

Kapcsolódó munkák

6.1. Gráf adathalmazok lekérdezése

Cypher A Cypher nyelvhez elérhető a *Cypher for Apache Spark*[1] projekt, ami a népszerű Spark keretrendszeren futtatható programokra képez le openCypher lekérdezéseket. A Cypher for Apache Spark projekt megoldása hasonlít a dolgozatban bemutatott C2S leképzéshez, azonban a Spark adatfolyam-orientált (streaming) programozási modellje miatt nem alkalmas módosítás és törlés műveletek kezelésére.

G-CORE A G-CORE [2] az LDBC Graph Query Language munkacsoportjának és több cég közreműködésével tervezett lekérdezési nyelv a tárolt utakkal rendelkező tulajdonsággráfokhoz (path property graph). Ebben az adatmodellben az éleken túl lehetőség van utak és azok tulajdonságainak tárolására is. A G-CORE lehetőséget nyújt az utak és tulajdonságaik lekérdezésére, módosítására. A G-CORE tervezésére befolyással volt a Cypher nyelv, így a szemantikai hasonlóságok mellett sok konstrukció azonos szintaxissal definiálható.

Datalog A Datalog egy deklaratív programozási nyelv, amely szintaktikailag a Prolog nyelv részhalmaza. Datalogban sor- és oszlopkalkulushoz hasonló lekérdezéseket lehet megfogalmazni. Számos kiterjesztése létezik, amelyeket különböző adatbázis-kezelőkben használnak, például LogiQL nevű kiterjesztését a LogicBox adatbázis-kezelőben [3].

GraphGen A GraphGen [33] eszköz a dolgozatban bemutatott C2S megoldásunkhoz hasonlóan relációs adatbázis-kezelők feletti gráflekérdezések futtatását biztosítja. A dolgozatban használt megközelítéstől eltérően azonban a GraphGen két Datalog alapú nyelvet definiál: (1) a GraphGenDL határozza meg az RDB feletti gráf nézetet, míg (2) a GraphGenQL fogalmazhatók meg a gráf nézetek feletti lekérdezések. A GraphGenDL tehát azt a szerepet tölt be, mint a gTop sémaleíró, míg a GraphGenQL az openCypher nyelv szerepéhez hasonló feladatot lát el.

VIATRA Query Language A gráfmintaillesztést széles körben használják a modellvezérelt technológiákban, például a VIATRA keretrendszerben [32]. A keretrendszer saját lekérdezőnyelve egy Dataloghoz hasonló nyelv, a VIATRA Query Language (VQL), amely támogatja a gráfminták komponálását, rekurzív minták megfogalmazását és aggregációt. A VIATRA keretrendszer a Rete algoritmust használja a gráfmodellek hatékony ellenőrzésre és transzformációjához. Az INCQUERY-D egy elosztott inkrementális

[1] https://github.com/opencypher/cypher-for-apache-spark

gráflekérdező [28], amely lekérdezőnyelve a VQL-en alapul és RDF gráfok lekérdezését teszi lehetővé.

A VIATRA egy korábbi verziójában készült egy prototípus alkalmazás [4], ami inkrementális gráftranszformációt végez relációs adatbázisokon. A prototípus a C2S implementációhoz hasonlóan gráfmintákból SQL lekérdezéseket készít a kezdeti illeszkedési halmazok meghatározásához. A mi megoldásunktól eltérően azonban egy mintát egy több illesztést tartalmazó, lapos SQL lekérdezéssé képez le. Ezután az illeszkedési halmazokat az adatbázis triggereinek (adott esemény hatására lefutó SQL kifejezések, például sorok beszúrása, törlése vagy módosítása) használatával tartja karban.

6.2. Teljesítménymérési keretrendszerek gráflekérdezésekre

A dolgozat készítése során több teljesítménymérési keretrendszert tanulmányoztunk, amelyek lehetővé teszik gráf információs rendszerekben futtatott lekérdezések teljesítményének vizsgálatát. Az általunk fontosnak vélt keretrendszerek legmeghatározóbb tulajdonságait mutatjuk be ebben a fejezetben.

LDBC Semantic Publishing Benchmark Az LDBC egy másik teljesítményérési keretrendszere a Semantic Publishing Benchmark [16] (SPB), amely a British Broadcasting Corporation[2] (BBC) Dynamic Semantic Publishing rendszeréhez hasonló RDF adathalmazt használ. Az adathalmazban különböző kreatív munkák (cikkek, fényképek és videók) érhetőek el különböző szempontok szerint csoportosítva. A teljesítménymérés során párhuzamosan futnak szerkesztői módosítások, beszúrások és a felhasználói lekérdezések az állandó terhelést szimulálva. A mérés során az LDBC SNB-hez hasonlóan az adatbázis-kezelők hatékonyságát leíró jellemzőket rögzít.

Azon túl, hogy csak RDF adathalmaz generálására alkalmas, az LDBC SPB a teljesítménymérés során olyan csak RDF-ben elérhető funkciókat használ, mint a következtetés (*inferencing*), azaz a meglévő kapcsolatok és előre definiált szabályok alapján új kapcsolatok létrehozása. A keretrendszer ezen tulajdonsága miatt nem alkalmas tulajdonsággráf alapú vagy relációs adatbázis-kezelők teljesítménymérésére.

Train Benchmark A Train Benchmark [29] a modellvezérelt technológiák által inspirált teljesítménymérési keretrendszer, amely adatmodellje a vasúthálózat és a hozzá kapcsolódó érzékelő és biztosító berendezések hálozatát tartalmazza. Többféle terhelési profilt definiál, több adatmodellt és lekérdezőnyelvet is támogat, köztük az ebben a dolgozatban használtakat is. Az LDBC SNB keretrendszernél azonban kevesebb nyelvi elemet fed le, például nem teszi lehetővé aggregációk, útvonalkifejezések vagy éleken definiált tulajdonságok teljesítménymérését.

[2] `http://www.bbc.com/`

7. fejezet

Összefoglalás és jövőbeli tervek

Dolgozatunkban azt a kérdést vizsgáltuk, hogy a manapság elérhető gráf információs rendszerek milyen kifejezővel, teljesítménnyel rendelkeznek, és vajon a hagyományos relációs adatbázisok megfelelő alternatívát nyújthatnak-e gráflekérdezések kiértékelésére [21].

7.1. Kontribúciók

Lekérdezések leképzése Dolgozatunkban bemutattuk, hogy openCypher nyelven megfogalmazott gráflekérdezések hogyan fordíthatók le SQL nyelvre. Az elkészült leképzés a felhasznált benchmark keretrendszer komplex lekérdezései közül képes volt az openCypher által támogatott 12 lekérdezésből 10 helyes fordítására, melyek közül 5-nek a teljesítményelemzését is elvégeztük. Ez komoly előrelépés a korábbi megoldásokhoz képest [27], melyek ugyanebből a lekérdezéshalmazból legfeljebb 2 lekérdezést tudtak helyesen lefordítani.

Teljesítménymérés A dolgozat készítése során elkészítettük az LDBC Social Network Benchmark *Interactive* terhelési profiljának lekérdezéseihez a Cypher és a SPARQL nyelvű implementációkat, illetve implementáltuk a keretrendszerhez szükséges szoftvermodulokat három gráf alapú adatbázis-kezelő eszközhöz. Implementáltuk továbbá a Gremlin nyelvet támogató eszközökhöz az adatok betöltését elvégző alkalmazást. Az elkészített implementációkkal lemértük 5 eszköz teljesítményét és értékeltük a kapott eredményeket. Az eredmények rávilágítottak arra, hogy a hagyományos relációs adatbázisok gráf jellegű terhelési profilok esetén is versenyképesek, azonban gráflekérdezések SQL-re leképzése nemtriviális feladat és külön optimalizáció nélkül gyakran rossz teljesítményhez vezet.

7.2. Jövőbeli tervek

A kutatási következő fázisaként a közeljövőben megvizsgáljuk, hogy milyen optimalizálási lehetőségek alkalmazhatók a C2S leképzés során előállított lekérdezések teljesítményének javításához. Erre egy lehetséges megközelítés a CTE (Common Table Expression) kifejezések részleges cseréje allekérdezésekre, továbbá a több táblából álló lekérdezések összevonása materializált nézetek segítségével.

A teljesítménymérési keretrendszer fejlesztésében szerepel céljaink között további eszközökhöz szükséges szoftvermodulok implementálása. Kiemelt célunk a lekérdezések teljesítményének mérése a *Cypher for Spark*, valamint a *Cypher for Gremlin* eszközökkel, melyek megközelítésükben hasonlítanak a dolgozatban bemutatott C2S (*Cypher-to-SQL*) rendszerre.

Köszönetnyilvánítás

Az Emberi Erőforrások Minisztériuma ÚNKP-18-2 kódszámú Új Nemzeti Kiválóság Programjának támogatásával készült.

Irodalomjegyzék

[1] Barabás Ákos – Szárnyas Gábor – Gajdos Sándor: NoSQL adatbázis-kezelők. In *Adatbázisok*. 4. kiad. 2012.

[2] Renzo Angles – Marcelo Arenas – Pablo Barceló – Peter A. Boncz – George H. L. Fletcher – Claudio Gutierrez – Tobias Lindaaker – Marcus Paradies – Stefan Plantikow – Juan F. Sequeda – Oskar van Rest – Hannes Voigt: G-CORE: A core for future graph query languages. In *SIGMOD Conference* (konferenciaanyag). 2018, ACM, 1421–1432. p.

[3] Molham Aref – Balder ten Cate – Todd J. Green – Benny Kimelfeld – Dan Olteanu – Emir Pasalic – Todd L. Veldhuizen – Geoffrey Washburn: Design and implementation of the LogicBlox system. In *SIGMOD Conference* (konferenciaanyag). 2015, ACM, 1371–1382. p.

[4] Gábor Bergmann – Dóra Horváth – Ákos Horváth: Applying incremental graph transformation to existing models in relational databases. In *ICGT*, Lecture Notes in Computer Science konferenciasorozat, 7562. köt. 2012, Springer, 371–385. p. URL `https://doi.org/10.1007/978-3-642-33654-6_25`.

[5] José A. Blakeley – David Campbell – S. Muralidhar – Anil Nori: The ADO.NET entity framework: making the conceptual level real. *SIGMOD Record*, 35. évf. (2006) 4. sz., 32–39. p. URL `http://doi.acm.org/10.1145/1228268.1228275`.

[6] Katrin Braunschweig – Maik Thiele – Wolfgang Lehner: A flexible graph-based data model supporting incremental schema design and evolution. In *ICWE*, Lecture Notes in Computer Science konferenciasorozat, 7059. köt. 2011, Springer, 302–306. p. URL `https://doi.org/10.1007/978-3-642-27997-3_29`.

[7] E. F. Codd: A relational model of data for large shared data banks. *Commun. ACM*, 13. évf. (1970) 6. sz., 377–387. p.

[8] Stefan Edlich – Daniel Oltmanns: db4o im spiegel von JPA/EJB und hibernate. *Datenbank-Spektrum*, 7. évf. (2007) 22. sz., 20–24. p.

[9] James Elliott: *Hibernate - a developer's notebook*. 2004, O'Reilly. ISBN 978-0-596-00696-9.

[10] Orri Erling – Alex Averbuch – Josep-Lluís Larriba-Pey – Hassan Chafi – Andrey Gubichev – Arnau Prat-Pérez – Minh-Duc Pham – Peter A. Boncz: The LDBC Social Network Benchmark: Interactive workload. In *SIGMOD Conference* (konferenciaanyag). 2015, ACM, 619–630. p.

[11] Orri Erling – Ivan Mikhailov: Virtuoso: RDF support in a native RDBMS. In *Semantic Web Information Management*. 2009, Springer, 501–519. p.

[12] Nadime Francis és mások: Cypher: An evolving query language for property graphs. In *SIGMOD Conference* (konferenciaanyag). 2018, ACM, 1433–1445. p.

[13] Hector Garcia-Molina – Jeffrey D. Ullman – Jennifer Widom: *Database systems - the complete book (2. ed.)*. 2009, Pearson Education. ISBN 978-0-13-187325-4.

[14] Paolo Guagliardo – Leonid Libkin: A formal semantics of SQL queries, its validation, and applications. *PVLDB*, 11. évf. (2017) 1. sz., 27–39. p.

[15] S. Harris – N. Lamb – N. Shadbolt: 4store: The design and implementation of a clustered RDF store. In *International Workshop on Scalable Semantic Web Knowledge Base Systems* (konferenciaanyag). 2009.

[16] Venelin Kotsev – Nikos Minadakis – Vassilis Papakonstantinou – Orri Erling – Irini Fundulaki – Atanas Kiryakov: Benchmarking RDF query engines: The LDBC Semantic Publishing Benchmark. In *BLINK@ISWC*, CEUR Workshop Proceedings konferenciasorozat, 1700. köt. 2016, CEUR-WS.org.

[17] LDBC Social Network Benchmark task force: LDBC Social Network Benchmark (SNB). Jelentés, 2018, Linked Data Benchmark Council. `http://https://ldbc.github.io/ldbc_snb_docs/ldbc-snb-specification.pdf`.

[18] József Marton – Gábor Szárnyas – Márton Búr: Model-driven engineering of an openCypher engine: Using graph queries to compile graph queries. In *SDL Forum*, Lecture Notes in Computer Science konferenciasorozat, 10567. köt. 2017, Springer, 80–98. p.

[19] József Marton – Gábor Szárnyas – Dániel Varró: Formalising opencypher graph queries in relational algebra. In *ADBIS*, Lecture Notes in Computer Science konferenciasorozat, 10509. köt. 2017, Springer, 182–196. p.
URL `https://doi.org/10.1007/978-3-319-66917-5_13`.

[20] Elizabeth J. O'Neil: Object/Relational Mapping 2008: Hibernate and the Entity Data Model (EDM). In *SIGMOD* (konferenciaanyag). 2008, ACM, 1351–1356. p.
URL `http://doi.acm.org/10.1145/1376616.1376773`.

[21] Anil Pacaci – Alice Zhou – Jimmy Lin – M. Tamer Özsu: Do we need specialized graph databases?: Benchmarking real-time social networking applications. In *GRADES@SIGMOD/PODS* (konferenciaanyag). 2017, ACM, 12:1–12:7. p.

[22] Jorge Pérez – Marcelo Arenas – Claudio Gutiérrez: Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34. évf. (2009) 3. sz., 16:1–16:45. p.

[23] Marko A. Rodriguez: The Gremlin graph traversal machine and language (invited talk). In *DBPL* (konferenciaanyag). 2015, ACM, 1–10. p.

[24] Michael Rudolf – Marcus Paradies – Christof Bornhövd – Wolfgang Lehner: The graph story of the SAP HANA database. In *BTW*, LNI konferenciasorozat, 214. köt. 2013, GI, 403–420. p.

[25] Gajdos Sándor: *Adatbázisok*. 4. kiad. 2012.

[26] Abraham Silberschatz – Henry F. Korth – S. Sudarshan: *Database System Concepts, 5th Edition*. 2005, McGraw-Hill Book Company. ISBN 978-0-07-295886-7.

[27] Benjamin A. Steer – Alhamza Alnaimi – Marco A. B. F. G. Lotz – Félix Cuadrado – Luis M. Vaquero – Joan Varvenne: Cytosm: Declarative property graph queries without data migration. In *GRADES@SIGMOD/PODS* (konferenciaanyag). 2017, ACM, 4:1–4:6. p.

[28] Gábor Szárnyas – Benedek Izsó – István Ráth – Dénes Harmath – Gábor Bergmann – Dániel Varró: Incquery-d: A distributed incremental model query framework in the cloud. In *MoDELS*, Lecture Notes in Computer Science konferenciasorozat, 8767. köt. 2014, Springer, 653–669. p.

[29] Gábor Szárnyas – Benedek Izsó – István Ráth – Dániel Varró: The Train Benchmark: cross-technology performance evaluation of continuous model queries. *Software and System Modeling*, 17. évf. (2018) 4. sz., 1365–1393. p.

[30] Gábor Szárnyas – József Marton – János Maginecz – Dániel Varró: Reducing property graph queries to relational algebra for incremental view maintenance. *CoRR*, abs/1806.07344. évf. (2018). URL `http://arxiv.org/abs/1806.07344`.

[31] Gábor Szárnyas – Arnau Prat-Pérez – Alex Averbuch – József Marton – Marcus Paradies – Moritz Kaufmann – Orri Erling – Peter A. Boncz – Vlad Haprian – János Benjamin Antal: An early look at the LDBC Social Network Benchmark's Business Intelligence workload. In *GRADES/NDA@SIGMOD/PODS* (konferenciaanyag). 2018, ACM, 9:1–9:11. p.

[32] Dániel Varró – Gábor Bergmann – Ábel Hegedüs – Ákos Horváth – István Ráth – Zoltán Ujhelyi: Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework. *Software and System Modeling*, 15. évf. (2016) 3. sz., 609–629. p.

[33] Konstantinos Xirogiannopoulos – Virinchi Srinivas – Amol Deshpande: GraphGen: Adaptive graph processing using relational databases. In *GRADES@SIGMOD/PODS* (konferenciaanyag). 2017, ACM, 9:1–9:7. p.

Függelék

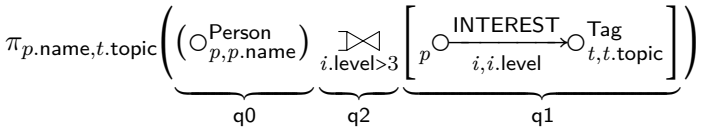
F.1. Opcionális gráfminták leképzése

Az alábbi példa bemutatja, hogy hogyan képezzük le az opcionális gráfmintákat baloldali külső illesztésre.

Példa. *A személyek és a 3-nál magasabb szintű érdeklődéseik felsorolása (ha vannak)*

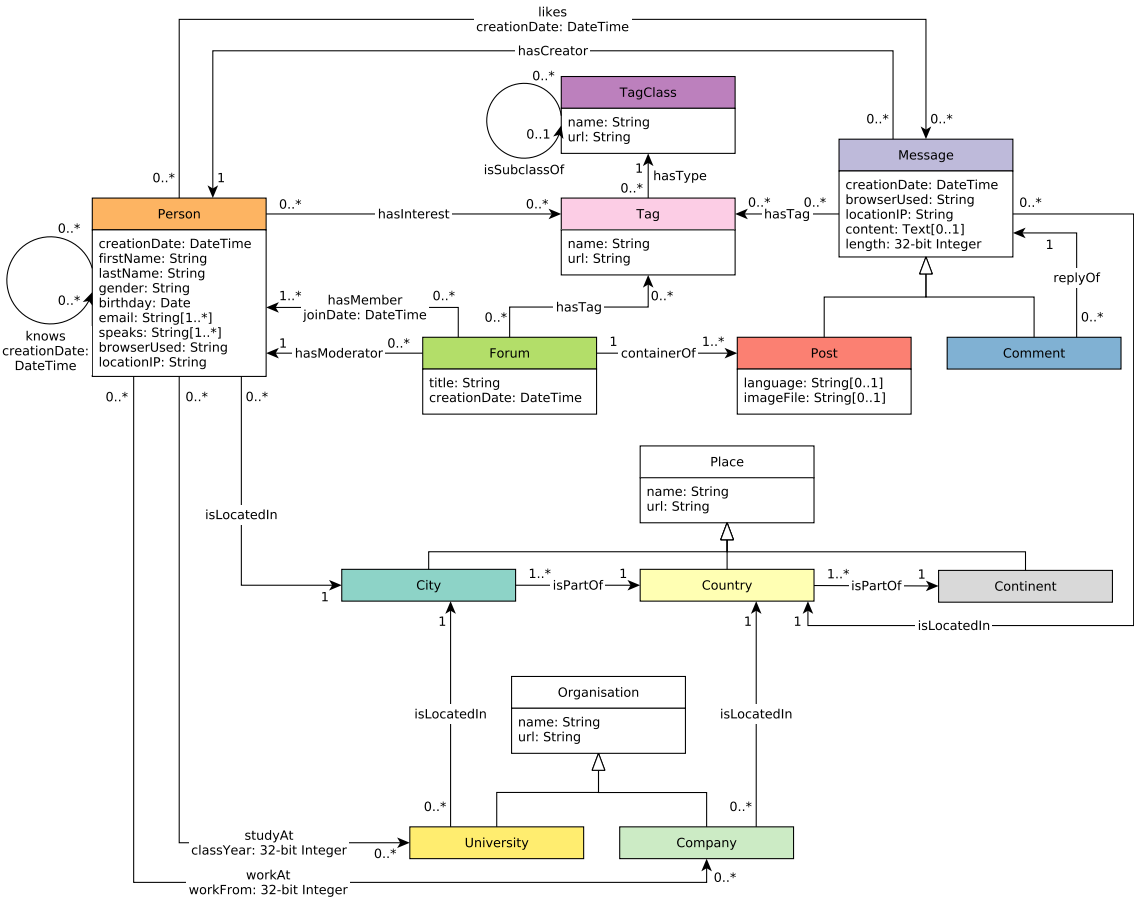
```
MATCH (p:Person)
OPTIONAL MATCH (p)-[i:INTEREST]->(t:Tag)
WHERE i.level > 3
RETURN p.name, t.topic
```

p.name	t.topic
Alice	Neofolk
Bob	NULL




```
1 WITH q0 AS (/* GetVertices: (p:Person) | attributes: p.name */),
2 q1 AS (/* GetEdges: (p)-[i:INTEREST]->(t:Tag) | attributes: i.level, t.topic */),
3 q2 AS (-- ThetaLeftOuterJoin
4   SELECT "left"."p", "left"."p.name",
5     "right"."i.level", "right"."i", "right"."t", "right"."t.topic"
6   FROM q0 AS "left" LEFT OUTER JOIN q1 AS "right"
7   ON "left"."p" = "right"."p" AND "i.level" > 3)
8 -- Projection
9 SELECT "p.name", "t.topic" FROM q2;
```

F.2. LDBC SNB adatséma



F.2.1. ábra. Az LDBC Social Network Benchmark gráf sémája