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

Hatékony gráflekérdezési technikák

Diplomaterv

Készítette
Antal János Benjamin

Konzulens
Szárnyas Gábor

2018. december 16.

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott *Antal János Benjamin*, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2018. december 16.

Antal János Benjamin
hallgató

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 gráfadatok hatékony feldolgozásához.

A gráfadatbázisok számos felhasználási esetében – ajánlórendszerek, pénzügyi csalások felderítése – a lekérdezések össszetettek, és ezek ismételt, gyors végrehajtása kritikus eleme a munkafolyamatoknak, továbbá a gráfadatbázisokat egyre többször használják szoftverek modelljeinek validációjára és forráskód-elemzésére. Az ilyen felhasználási módok esetében a hagyományos gráfadatbázis-kezelőknél jobb teljesítményt tudna nyújtani egy inkrementális lekérdezéseket támogató gráfadbázis-kezelő rendszer, amely támogatja nézetek létrehozását és azokat automatikusan karbantartja az adatbázis változása során.

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 teljesítménymérési keretrendszer létezik a gráflekérdezések teljesítménymérésére. Azért, hogy segítsem egy szabványos teljesítménymérési keretrendszer létrejöttét, bekapcsolódtam az LDBC (Linked Data Benchmark Council) Social Network Benchmark fejlesztésébe, amelynek keretében frissítettem és jelentősen fejlesztettem a meglévő implementációkat, továbbá elkészítettem a SPARQL nyelvű implementációt. Ezek felhasználásával megvizsgáltam és részletesen elemeztem az adatbázis kezelőket különböző adatmodellek (relációs, gráf és szemantikus) felhasználásával.

Az inkrementális gráflekérdezésekhez kapcsolódóan megismertem az időalapú és differenciális adatfolyamok programozási paradigmákat, és a TTC (Transformation Tool Contest) verseny 2018-as feladatát megvalósítottam egy differenciális adatfolyamok létrehozását támogató szoftverkönyvtár, a Naiad felhasználásával. Ennek teljesítményét összehasonlítottam a versenyre érkezett más megoldások teljesítményével.

Abstract

In the last decade, numerous database management systems were developed under the umbrella of NoSQL techniques. One group of these systems is the family of graph databases, which allows 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 at all.

In many use cases of graph databases – financial fraud detection, recommendation engines – the graph queries are complex, and their repeated, efficient executions are crucial for these use cases. Furthermore, graph databases are increasingly used for software model validation and source code analysis. These application scenarios could greatly benefit from an incremental graph query engine that allows users to register views and maintain their state upon on changes.

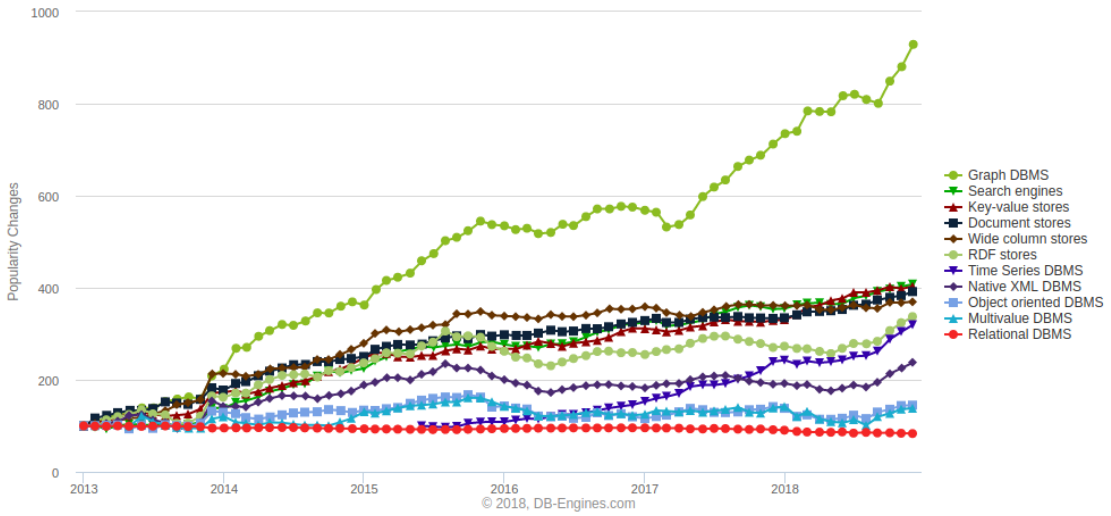
Comparing the performance of database systems requires standard benchmarks. For relational databases, this has been fulfilled by the benchmarks of the Transaction Processing Performance Council for more than two decades. Due to the relative immaturity of graph databases, there is only a limited number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

Related to incremental graph query processing, I studied the timely and differential dataflow programming paradigms. I demonstrated their applicability by implementing a solution for the 2018 Transformation Tool Contest's live challenge using the Naiad differential dataflow software library and compared its performance to other solutions.

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.



1.1. ábra. Adatbázisokban alkalmazott adatmodellek népszerűségének változása a `https://db-engines.com` oldal rangsorolása alapján

Mi lehet a gráfadatbázisok 1.1. ábrán is látható sikerének oka? Ezek a rendszerek több előnnyel is rendelkeznek a megszokott relációs adatbázisokhoz képest:

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

- **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.
- **Gyors prototipizálás**: A gráfadatbázisok gyenge sémájának köszönhetően egyszerűbben létrehozhatók a prototípusok mint a relációs adatbázisoknál. Természetesen ennek következtében a lekérdezések optimalizálása bonyolultabbá válik a relációs lekérdezésekhez képest [34]. Az egyik legnagyobb nehézséget a kapcsolatok kardinalitásának megbecslése jelenti körkörös illesztéseket tartalmazó lekérdezések esetében, melyre egy éltípuson történő ismételt navigálás megvalósításához van szükség.

A fentieket figyelembe véve elmondható, hogy sok esetben érthetőbb adathalmazt, valamint tömörebb lekérdezéseket eredményez a gráfadatbázisok használata. Azonban mivel ezek a rendszerek az informatika és a számítástudomány világában annyira újnak számítanak, hogy számos nyitott kérdés van velük kapcsolatban. Az egyik ilyen kérdés a gráfalapú lekérdezések és lekérdezőnyelvekkel kapcsolatos. Mivel a relációs adatbázisokban a – legtöbbször SQL-ben megfogalmazott – lekérdezések optimalizációjára igen hatékony megoldások léteznek, ezek bizonyos esetekben nem működnek jól a gráfadatbázisokban. Ezeknek az eseteknek az azonosítása az egyik legnagyobb nyitott kérdés. További fontos kérdés, hogy az ezeket a problémákat megoldó, illetve a már meglévő megoldások – pl. a relációs adatbázisokban megszokott kétoperandusú illesztés (join) műveletek – mennyire hatékonyak. Az alapvető problémák azonosításán és megoldásán túl fontos kérdés, hogy egyéb, a relációs adatbázisokban elterjed módszerek (pl. inkrementális nézetkarbantartás) hogyan ültethetőek át a gráfadatbázisokba.

Dolgozatomban megvizsgáltam, hogy mennyire hatékonyak a klasszikus relációs, és a modern gráfadatbázis-kezelők a gráfjellegű lekérdezések számítására. Az ehhez szükséges elméleti háttér (2. fejezet) megismerése után a kiértékeléshez kiválasztottam egy széleskörben elfogadott, sok nyelvi elemet és teljesítmény-aspektust lefedő teljesítménymérési keretrendszert, az LDBC Social Network Benchmarkot. A benchmark keretrendszerében javítottam a specifikáció hiányosságait és hibáit, ezek alapján frissítettem a benne található lekérdezéseket és szoftvermodulokat

(3. fejezet). A klasszikus adatbázis-kezelőkön kívül megvizsgáltam a differenciális adatfolyamot, mint alacsony válaszidővel rendelkező, elosztott, iteratív és inkrementális számításokat együttesen támogató számítási modellt. Az inkrementális lekérdezésekben nyújtott teljesítményének és a megközelítés használhatóságának tanulmányozásához a Transformation Tool Contest gráftranszformációs verseny (4. fejezet) egyik feladatát valósítottam meg. Az így rendelkezésre álló implementációk segítségével elkészítettem a különböző adatmodellt, nyelvet és megközelítést alkalmazó rendszerek összehasonlító teljesítménymérését (5. fejezet). Végezetül áttekintettem a kapcsolódó kutatási eredményeket (6. fejezet), amelyek alapján javaslatot tettem a tanszéken fejlesztett *ingraph* rendszer továbbfejlesztésére. Végezetül pedig meghatároztam 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 mutatom be. A példa egy közösségi háló lehetséges adatbázisának egy részlete. A 2.1. ábrán látható a példa gráfos ábrázolása, melyről leolvashatóak az alábbi adatok:

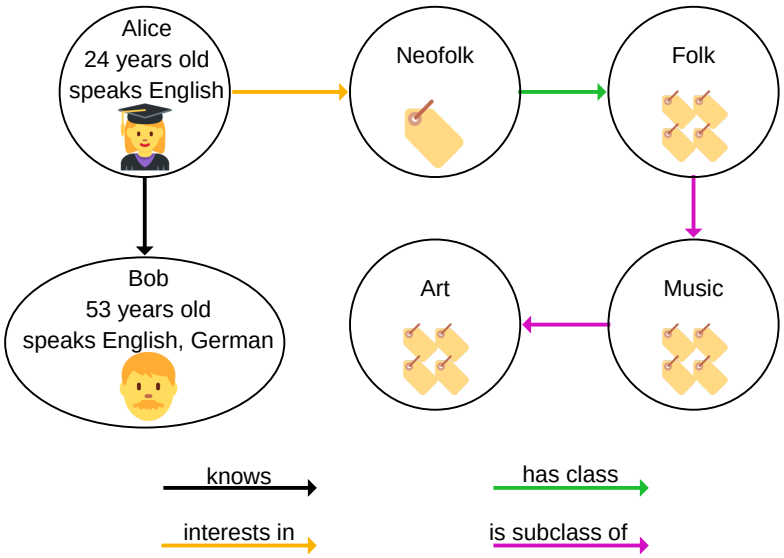
- **Bob**: 53 éves, angolul és németül beszél.
- **Alice**: 24 éves, angolul beszél, *érdeklődik* a **Neofolk** iránt. **Alice** *ismeri* **Bob**ot.
- A **Neofolk** *egy* **Népzene**i műfaj.
- A **Népzene** *része* a **Zené**nek, amely *része* a **Művészet**nek.

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

2.2. 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



2.1. ábra. A példa gráf

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 ezt 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 [25]. 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 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.
- 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.

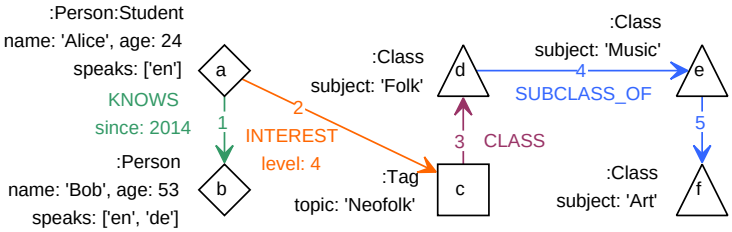
Legyen r reláció n-esek egy *multihalmaza* (bag). Ekkor r egy gráf reláció a G tulajdonsággráfon, ha igaz az alábbi:

$$\forall A \in \mathrm{sch}\,(r) : \mathrm{dom}(A) \subseteq V \cup E \cup D,$$

ahol $sch(r)$ jelöli r sémáját, azaz r-ben lévő attribútumok neveinek listáját, $dom(A)$ az A attribútum doménjét, továbbá V és E a definíció szerint a gráf csúcsait és éleit jelölik.

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

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



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

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 emeltem ki a specifikációból.

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

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

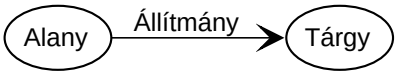
Az IRI[3] (Internationalized Resource Identifier) egy UNICODE karakterlánc, amely egyértelműen azonosítja a gráf elemeit. Egy RDF gráfban az üres csomópontokat egy végtelen halmazból nyerjük. Az üres csomópontoknak ez a halmaza, továbbá az összes IRI hivatkozás halmaza, valamint az összes literál halmaza 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 .
```

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

```
_: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.

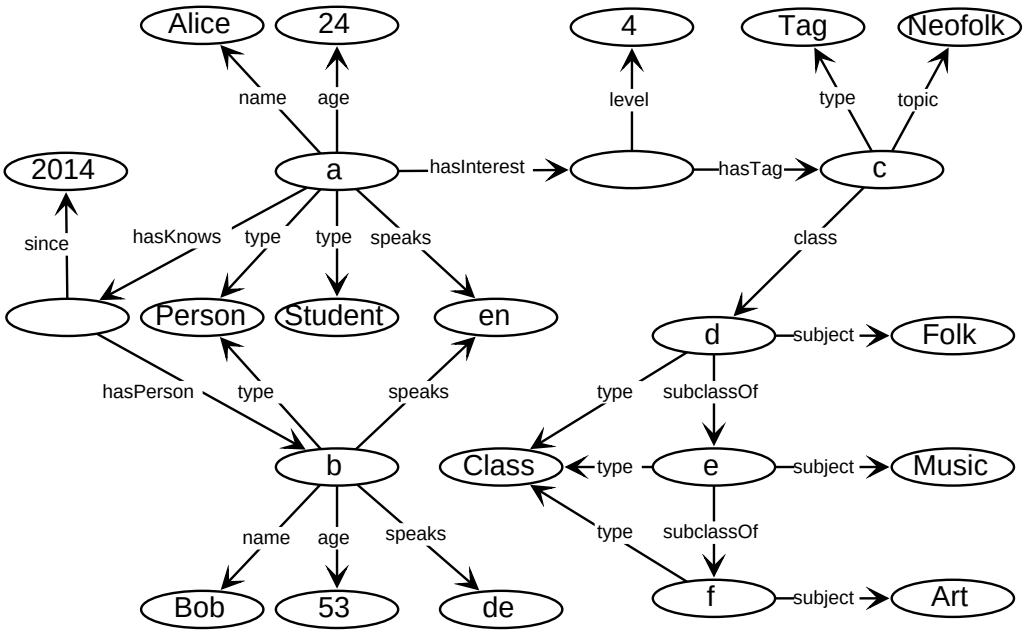
2.2.3. Relációs adatmodell

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 [6] ismertetem. 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 az i. oszlop az i. domén értékeit tartalmazza. Az oszlopokat szokás a hozzájuk tartozó domén nevével felcímkézni, ezzel 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* 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



2.4. ábra. A példa RDF gráf modellje

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).

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

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 bemutatok három új lekérdezőnyelvet és természetesen a relációs adatbázisok (relational databases, RDB) lekérdezőnyelvét, az SQL-t is bemutatom. A lekérdezőnyelvek bemutatása során a dolgozat megértéséhez szükséges részletekre fókuszálok, mivel a nyelvek részletes bemutatása meghaladja a dolgozat kereteit. A bemutatott nyelveken egy egyszerű lekérdezést is megfogalmazok 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 [35, 13, 36]. Az alábbiakban röviden áttekintem a legtöbbet használt relációalgebrai operátorokat.

2.3.1.1. 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 relációt 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.

2.3.1.2. Bináris operátorok

Illesztés jellegű műveletek Az *illesztés* (join) jellegű műveletek két reláció összekapcsolását végzik el, általában növelve az attribútumok számát. Az illesztés jellegű 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:

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

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, $\overset{\frown}{\bowtie}$) [36] 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 operátort 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 [33], Redis Graph[5]) implementálták. A kereskedelmi termékek mellett több kutatási projektben (ingraph [24], Cytosm [37]) 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 felhívom 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

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

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`, 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 megfogalmazására 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 mutatom 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 [32] 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

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

- 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 a 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 függvényparaméterekkel pedig a bejárások egymásba ágyazása is lehetséges, például az $a(b \circ c) \circ d$ kifejezés `a(b().c()).d()` alakban írható le.

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 [31] elterjedt és az 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 ismertetett 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 lekérdezés 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 relációalgebrai 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 használjuk 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 kötetlen 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 hiányosságát 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 [15], 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 bemutatom a dolgozat szempontjából legfontosabb alkalmazásokat.

Formátum	Alkamazás	Lekérdezőnyelv	Mem.	Impl. nyelve
PG	JanusGraph	Gremlin	○	Java
	Neo4j	Cypher	○	Java
	Sparksee	C++, Java, C#, Python API	○	C++
	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 alacsony szintű Java API-n keresztül, amellyel primitív gráfműveleteket hajthatunk végre, illetve deklaratív módon a Cypher nyelven megfogalmazott lekérdezésekkel. Az egyik hátránya, hogy nem támogatja a tisztán memóriában történő tárolást, csak a merevlemez alapú perzisztens tárolást. Támogatja azonban a fürtök létrehozását.

Sparksee Egy nagy teljesítményű, C++-ban írt kereskedelmi forgalomban kapható tulajdonsággráf adatbázis[14]. Több nyelvhez készítettek programozói interfészt, köztük JAVA-hoz, C#-hoz és Pythonhoz is. Támogatja az elosztott működést is.

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

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.

Virtuoso A Virtuoso [9] 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[15] egy nagy teljesítményű, perzisztens adattárolást megvalósító kereskedelmi forgalomban kapható szemantikus gráfalapú adatbázis-kezelő.

[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`
[14] `http://www.sparsity-technologies.com/`
[15] `https://allegrograph.com/`

Lekérdező nyelve a SPARQL, de számos programozási nyelven megírt interfésszel rendelkezik. Beépített támogatást nyújt a szemantikus következtetésre RDFS[16], SPIN[17] vagy Prolog nyelven megfogalmazott szabályok alapján.

4store A 4store [18] 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, egyszerveres konfigurációtól kezdve a hatalmas adattárházakig mindenhol megtalálható. Lekérdezőnyelve az SQL szabvány jelentős részét lefedi, beleértve a `WITH RECURSIVE` kifejezést[18], amely rekurzív lekérdezések megfogalmazását teszi lehetővé, így teremtve meg a lehetőséget az útkereső lekérdezéseknek.

MySQL A 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[19] óta szintén támogatja a `WITH RECURSIVE` kifejezést, így lehetőséget teremtve az útkereső lekérdezések futtatására.

2.5. Inkrementális nézetkarbantartás

Nézetek definiálása során különböző lekérdezéseket fogalmazunk meg az adatbázis-kezelő rendszerekben, amelyek eredményét az első kiértékelés után frissítjük az adatok változása esetén. A triviális megoldás alapján a nézet frissítése a definiálásához használt lekérdezés újbóli kiértékelésével történik. Egy hatékonyabb, ám komplexebb megoldás lehet az, ha az adatok változása alapján csupán a nézet változásait határozzuk meg a lekérdezés teljes kiértékelése nélkül. Ezt a módszert *inkrementális nézetkarbantartás*nak (INK, incremental view maintenance) nevezzük. Az ilyen, inkrementálisan karbantartott nézetek definiálása akkor fontos, amikor egy lekérdezést sűrűn, rövid válaszidővel szeretnénk futtatni, esetleg folyamatosan értesülni szeretnénk a legfrissebb eredményről. Ezek a lekérdezések gyakran igen összetettek (pl. sok illesztés és aggregáció műveletet tartalmaznak), ezért kiértékelésük sokáig tart. Így sokszor célszerű megoldás az inkrementális nézetkarbantartás, ahol az előző eredmények és az adatbázis változásából – az újbóli kiértékelésnél jelentősen hatékonyabb módon – meghatározzuk a lekérdezés eredményét a frissített adathalmazon. Az INK két csoportját különböztetjük meg: az *algebrai* és a *procedurális* megközelítéseket. Az algebrai

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

megközelítések a lekérdezésekből újabb lekérdezéseket állítanak, amelyek meghatározzák a nézetek változásait. A procedurális megközelítések pedig nevükből adódóan procedurális algoritmusokat használnak a nézetek karbantartásához.

2.5.1. Történeti áttekintés

Az egyik első, több mint három évtizedes megközelítés [5] egy hibrid megoldást alkalmaz: egyszerre használ algebrai és procedurális megoldásokat. Az egyik legnagyobb megkötés a módszer használhatóságával kapcsolatban, hogy csak illesztéseket, szelekciókat és projekciókat támogat, és azokat is csak a hagyományos halmaz szemantika mentén. A következő jelentős mérföldkő egy teljesen procedurális megközelítés [16], ami már a multihalmaz szemantika felett képes INK-t végezni. A procedurális megoldás után készült egy teljesen algebrai módszereket használó megoldás [7] is, ami támogatja a multihalmaz szemantikát. Az említett megoldások közül egy sem támogatja a relációs adatbázis-kezelőkben széles körben használ NULL értékeket. Az első NULL értékeket is támogató algebrai megoldást [14] nagyjából két évtizede alkották meg. A megoldás jelentősége, hogy így már a külső illesztések (outer join) támogatása is lehetséges.

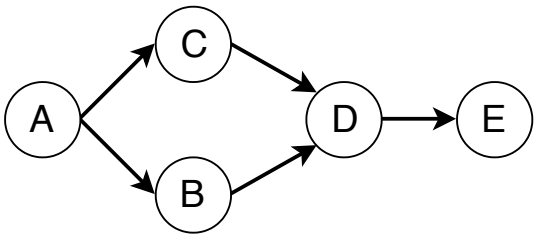
A DBToaster [20] egy nyílt forráskódú[20] akadémia prototípus eszköz, ami egy egyedi technikát használ az INK-hoz: a viewlet transzformációt. A megoldás lényege, hogy nem külön multihalmazokban tárolja az újonnan hozzáadott és kitörölt elemeket, hanem *általánosított multihalmaz relációt* (ÁMR, generalized multiset relation) használ, amiben lehetséges negatív, sőt racionális (nem egész számú) multiplicitás definiálása. Az ÁMR használata lehetőséget teremt a delta lekérdezések kompakt megvalósítására (a delta lekérdezés kevesebb illesztést tartalmaz, mint az eredeti lekérdezés), amely utat nyit a lekérdezések rekurzív deriválásának (a delta lekérdezéseik meghatározásának). A DBToaster az SQL szabvány jelentős részét támogatja, köztük beágyazott (nested) lekérdezéseket is.

2.5.2. Differenciális adatfolyamok

A gráfokon történő lekérdezések futtatásának egy, az említettektől eltérő módja a lekérdezések *adatfolyamként* (dataflow) történő megfogalmazása. Az adatfolyamok olyan irányított gráfok, ahol a csomópontok jelentik az adatokon való műveleteket, a csomópontok között irányított élek pedig meghatározzák, hogy a csomópontok kimenetét mely csomópontok bemenete felé kell továbbítani. Egy adatfolyamra látunk példát a 2.7. ábrán: az A kimenete továbbításra kerül a B és C csomópontokhoz, amelyek kimenetét D csomópont dolgozza fel. Végül a D kimenetét az E csomópont dolgozza fel. Az adatfolyamban minden csomópontnak opcionálisan lehet belső állapota (pl. count típusú műveleteknél az elemszámok nyilvántartása).

Az adatfolyamok áteresztőképességét triviálisan tudjuk növelni, ha csomópontokat valamilyen módszer szerint elosztjuk különböző feldolgozóegységek között. Ekkor azonban

[20] `https://dbtoaster.github.io/`

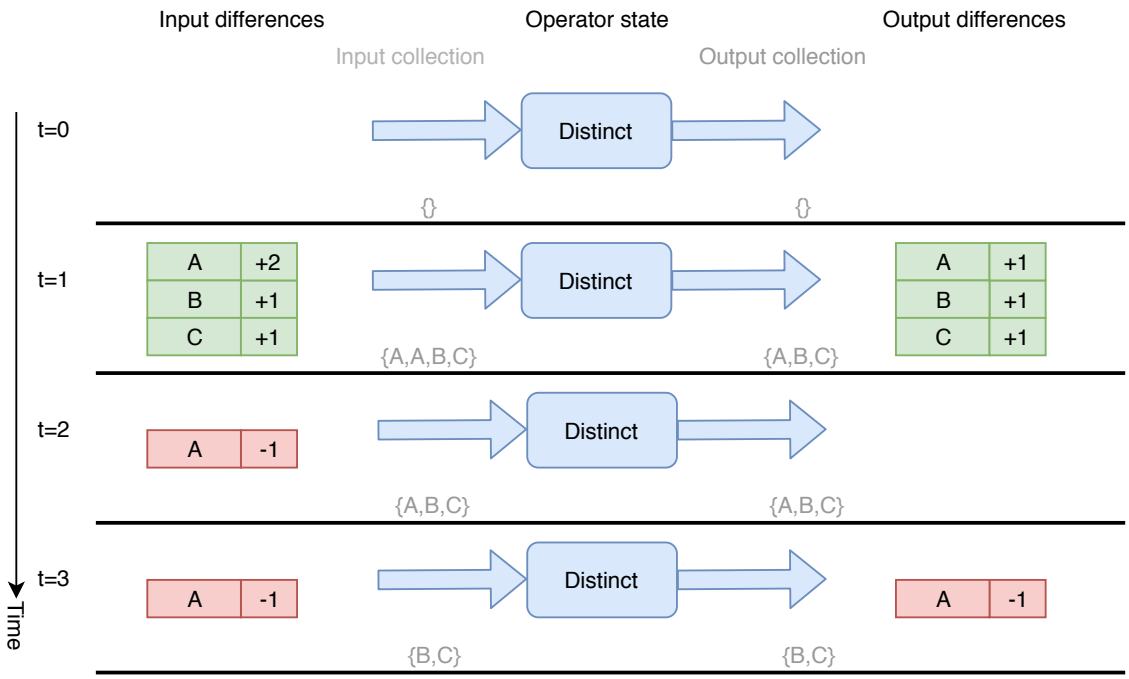


2.7. ábra. Egy példa adatfolyam

a konzisztens eredmények miatt szükséges valamilyen koordináció a csomópontok között. Továbbá ahhoz, hogy összetettebb műveleteket tudjuk elvégezni az adatfolyamokkal, szükség lehet iteratív algoritmusok végrehajtására, amikhez elengedhetetlen elemek a ciklusok. Az ilyen nagy áteresztőképességű, elosztott, iteratív elemeket tartalmazó, rövid válaszidővel rendelkező adatfolyamokban a csomópontok közötti koordináció kulcsfontosságú a rendszer hatékonyságához.

Erre ad egy megoldást az *időzített adatfolyam* (IAF, timely dataflow) [27] számítási modell. Az IAF egy olyan batch-es számításokat támogató számítási modell, ahol egy ciklikus adatfolyamokban a csomópontok között közlekedő minden adatrekordhoz egy virtuális időbélyeget (egész számok egy listáját) rendelünk. Az időbélyegek használatával lehetőség nyílik aszinkron adatfolyamok létrehozására úgy, hogy az esetlegesen szükséges szinkronizáció biztosítása is megoldható marad. Az IAF esetében lehetőség van arra, hogy a csomópontok értesüljenek arról, hogy már megkapták egy adott időbélyeghez tartozó összes adatrekordot. A cikkben [27] leírt elosztott mechanizmus használatával garantálni lehet azt, hogy az értesítés valóban az adott időbélyeghez tartozó összes üzenet megérkezése után történjen. A ciklikus adatfolyamok támogatása pedig a hierarchikus időbélyegekkel lehetséges: a ciklusok kezdetén mindig egy újabb időbélyeget fűzünk hozzá a meglévő időbélyegekhez, amely a ciklus iterációnak számát jelöli. Tehát az ilyen hierarchikus időbélyegek első száma azt mutatja meg, hogy az adott rekord a bementi adatok közül melyik batch-be tartozik, az összes többi szám pedig a beágyazott ciklusok iterációinak számát jelölik.

Az időzített adatfolyam felett definiált inkrementális számítási modell a *differenciális adatfolyam* (DAF, differential dataflow). A DAF célja adatpárhuzamos számítások inkrementális karbantartásának hatékony megvalósítása. Mivel a DAF-ok az adatrekordok mentén párhuzamosítanak, így a bemeneti adatok változása esetén minden csomópontnak elég a változott adatrekordokhoz tartozó eredményeket felülvizsgálni, így hatékonyan meg lehet határozni a kimenet változását is. Mivel az adatfolyamban csak a változásokat kommunikálják a csomópontok, így a csomópontok az őket követő csomópontoknak az adathalmaz méretéhez képest csak kevés adatrekordot továbbítanak. Egy DAF-ban használt *distinct* operátor működése látható a 2.8. ábrán. Figyeljük meg, hogy *t=2* időpontban a bemenet változása nem vonja maga után a kimenet változását, így az operátor utáni csomópontoknak semmilyen számítást nem kell végezniük, garantáltan nem



2.8. ábra. A *distinct* operátor működése DAF esetében

fog változni a kimenetük a DAF bemenetében történt változás hatására. Ezzel a módszerrel a változások okozta számítások minimalizálhatóak.

2.5.3. Technológiák

2.5.3.1. Eclipse Modeling Framework (EMF)

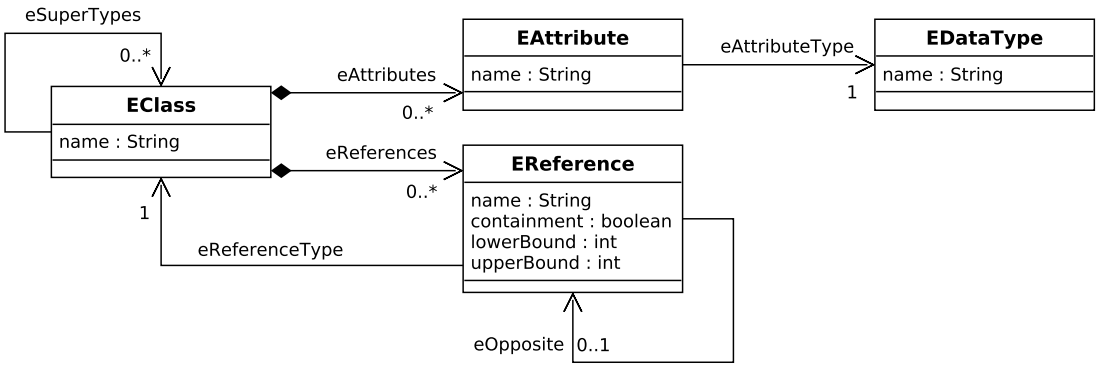
Az *Eclipse Modeling Framework* egy keretrendszer, melyet modellvezérelt alkalmazásfejlesztésre terveztek. A keretrendszer része az Ecore metamodellező nyelv, melyet főleg doménspecifikus nyelvek és azok szerkesztőinek készítésére használják. Az Ecore a nyelv fejlesztői által definiált metamodell alapján számos hasznos eszköz (parser, serializer vagy akár triviális grafikus szerkesztő undo-redo támogatással) forráskódját képes legenerálni. A 2.9. ábrán látható az Ecore metamodellje, én csak a három legfontosabb elemét emelném ki:

- Az *EClass* példányai jelentik a modellezési nyelv osztályait.
- Az *EAttribute* példányai jelentik a modellezési nyelv osztályainak attribútumait, az objektumok tulajdonságait.
- Az *EReference* példányai pedig irányított kapcsolatot teremtenek a modellezési nyelv objektumai között.

Az EMF-en alapszik a modell alapú világ egy inkrementális lekérdezéseket és transzformációkat nyújtó keretrendszere, a VIATRA [41].

2.5.3.2. .NET Modeling Framework (NMF)

Az EMF-hez hasonló modellezési keretrendszer a *.NET Modeling Framework (NMF)* [19]. Ahogy a neve is sugallja, .NET-ben (C# nyelven) fejlesztett keretrendszer, azonban a .NET Core[21] futtatókörnyezetnek köszönhetően cross-platform keretrendszer.



2.9. ábra. Az Ecore metamodellje

2.5.3.3. Naiad

A Naiad [26] a Microsoft Research Silicon Valley Lab[22] által fejlesztett IAF-t implementáló szoftverkönyvtár. A könyvtár által kínált szolgáltatások erősen beágyazottak a C# által nyújtott infrastruktúrába: a felhasználók a LINQ[23]-hoz hasonló módon definiálhatják az adatfolyam gráfot. Természetesen igény esetény egyedi csomópontokat is definiálhatunk. A Naiad szoftverkönyvtár a DAF-t implementáló keretrendszeren kívül tartalmaz kifejezetten gráf specifikus IAF operátorokat megvalósító keretrendszert is. A laboratórium bezárása, azaz 2014 ősze óta a Naiad nincs aktív fejlesztés alatt.

2.5.3.4. Timely Dataflow és Differential Dataflow

A IAF és DAF kidolgozásában és a Naiad implementációjában részt vevő egyik kutató saját projektjeként elkészítette az IAF[24] és DAF[25] implementációját Rust nyelven[26]. Fontos előnye a Naiaddal szemben, hogy jelenleg is aktívan folyik a fejlesztése.

[21] https://dotnet.microsoft.com
[22] https://www.microsoft.com/en-us/research/project/naiad/
[23] https://msdn.microsoft.com/en-us/library/bb308959.aspx
[24] https://github.com/frankmcsherry/timely-dataflow
[25] https://github.com/frankmcsherry/differential-dataflow
[26] https://www.rust-lang.org/

3. 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 alapján a Linked Data Benchmark Council (LDBC)[1] szervezet Social Network Benchmark (SNB) keretrendszerét választottam, 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).

3.1. 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.1.1. ábrán látható. A keretrendszer kétféle terhelési profilt tartalmaz.

Az úgynevezett *Business Intelligence* [8] (BI) terhelési profil olyan analitikus lekérdezéseket fogalmaz meg, amelyeknek a megválaszolásához az adathalmaz átfogó vizsgálata szükséges, például minden felhasználó üzeneteinek megszámolása. Ilyen

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

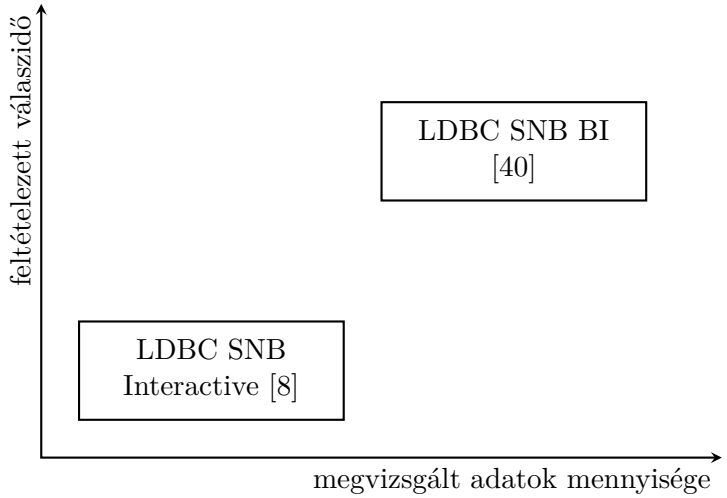
[2] Az aktuális partnercégek listája megtekinthető a szervezet honlapján: `http://ldbcouncil.org/industry/members`

lekérdezés például a legaktívabb, vagy éppen a legkevésbé aktív felhasználók megkeresése, vagy a baráti háromszögek megkeresése. A terhelési profil tervezése során gondosan figyeltek az olyan különböző kihívásokra, amelyeket az adatbázis-kezelő rendszereknek hatékonyan kell kezelni ahhoz, hogy nagyméretű adathalmazokon is megfelelő teljesítményt nyújtsanak. Ezek a kihívások a funkcionalitás széles köréből gyűjtötték össze: a relációs adatbázis-kezelőknél megfigyelt nehézségeken túl többek között a lekérdezőnyelvek kifejezőerejét, vagy a gráf specifikus lekérdezések komplexitását is vizsgálták ezen szempontrendszer felépítéséhez. A két teljesítménymérési profil közül a BI jött létre később, véglegesítése még jelenleg is folyamatban van.

Az *Interactive* [40] profil lekérdezései pedig inkább egy, vagy néhány csomóponthoz kapcsolódó adat alapos vizsgálatát követelik meg. Az *Interactive* profil lekérdezései három további csoportba sorolhatóak:

- 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.

A két terhelési profil két fontos tulajdonságának összehasonlítását mutatja a 3.1. ábra: a feltételezett válaszidőt és a lekérdezés során megvizsgált adatmennyiséget.



3.1. ábra. A teljesítménymérési profilok karakterisztikája.

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[3] formátumban tárolt adathalmaz gigabájtban

[3]Comma-Separated Values

számolt mérete adja. A keretrendszer által előre konfigurált skálázási együtthatók a következők: 0.1, 0.3, 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 [22].

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

3.1.1. A teljesítménymérés munkafolyamata

A keretrendszer munkafolyamatát a 3.2. 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 és *(4)* generált adat. Az összetevőket továbbá két csoportba sorolhatjuk az alapján, hogy az elkészítésük vagy végrehajtásuk a keretrendszer fejlesztőinek (LDBC SNB munkacsoport) vagy felhasználóinak (Adatbázis-kezelők fejlesztői, felhasználói) a feladata.

A keretrendszer fejlesztőinek feladatai:

- A *lekérdezések specifikációnak*[4] elkészítése és karbantartása [22]
- Az adathalmazokat és lekérdezések paramétereit *generáló alkalmazás* (DATAGEN)[5] elkészítése
- A megvalósítások ellenőrzése és a teljesítménymérést végző *alkalmazás keretrendszer* (DRIVER)[6] 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 Sparksee implementációt tekintettem referenciaimplementációnak.

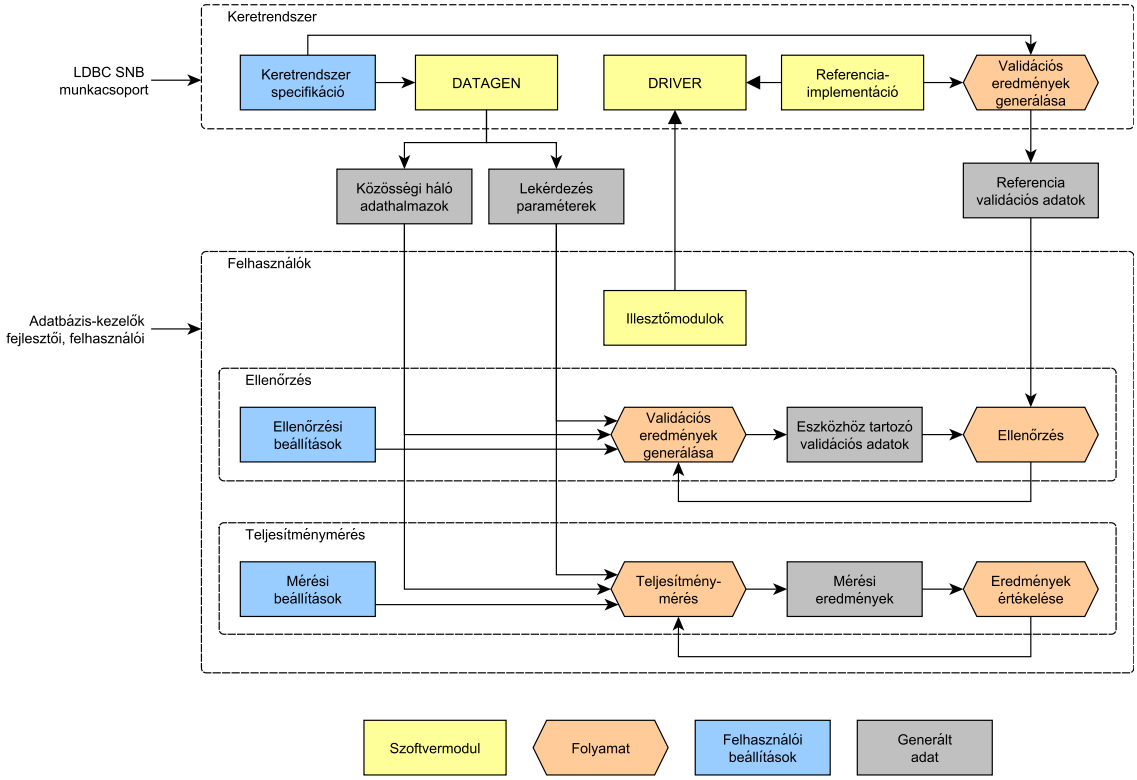
A felhasználók által végzett munkát további két csoportba lehet osztani: az implementáció ellenőrzése és a teljesítménymérés. Mindkettőhöz szükséges az adott adatbázis-kezelőhöz tartozó szoftvermodulok és lekérdezések implementációja. A szoftvermodulok közé tartoznak az adatok betöltését, a lekérdezések futtatását végző és az eredményeket a DRIVER számára feldolgozható formátumra konvertáló modulok. Az ellenőrzés lépései ezek után az alábbiak:

[4] `https://github.com/ldbc/ldbc_snb_docs`
[5] `https://github.com/ldbc/ldbc_snb_datagen`
[6] `https://github.com/ldbc/ldbc_snb_driver`

- 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 referencia 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..



3.2. 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)

3.2. Teljesítménymérési keretrendszer bővítése

A dolgozat készítése során az LDBC SNB-t bővítettem a *Business Intelligence* terhelési profil SPARQL, és az *Interactive* terhelési profil SPARQL és Cypher implementációjával, ezzel bővítve ki a keretrendszerrel lemérhető adatbázis-kezelők halmazát. Ezen felül implementáltam egy Gremlin-t támogató adatbázis-kezelőkhöz használható adatbetöltő alkalmazást.

3.2.1. Business Intelligence terhelési profil

A Business Intelligence terhelési profil lekérdezéseit SPARQL nyelven készítettem el. A lekérdezéseknek nem létezett SPARQL megvalósítása. A lekérdezések komplexitásából adódóan az implementálás során sok esetben a nyelv kifejezőerejét is próbára tették. A 25-ös lekérdezést nem is lehet szabványos SPARQL-ben megvalósítani, mert nem támogatja a legrövidebb utak keresését. A következőkben bemutatom az általam legérdekesebb, vagy legnagyobb kihívásokat, amelyekkel az implementáció készítése során szembesültem.

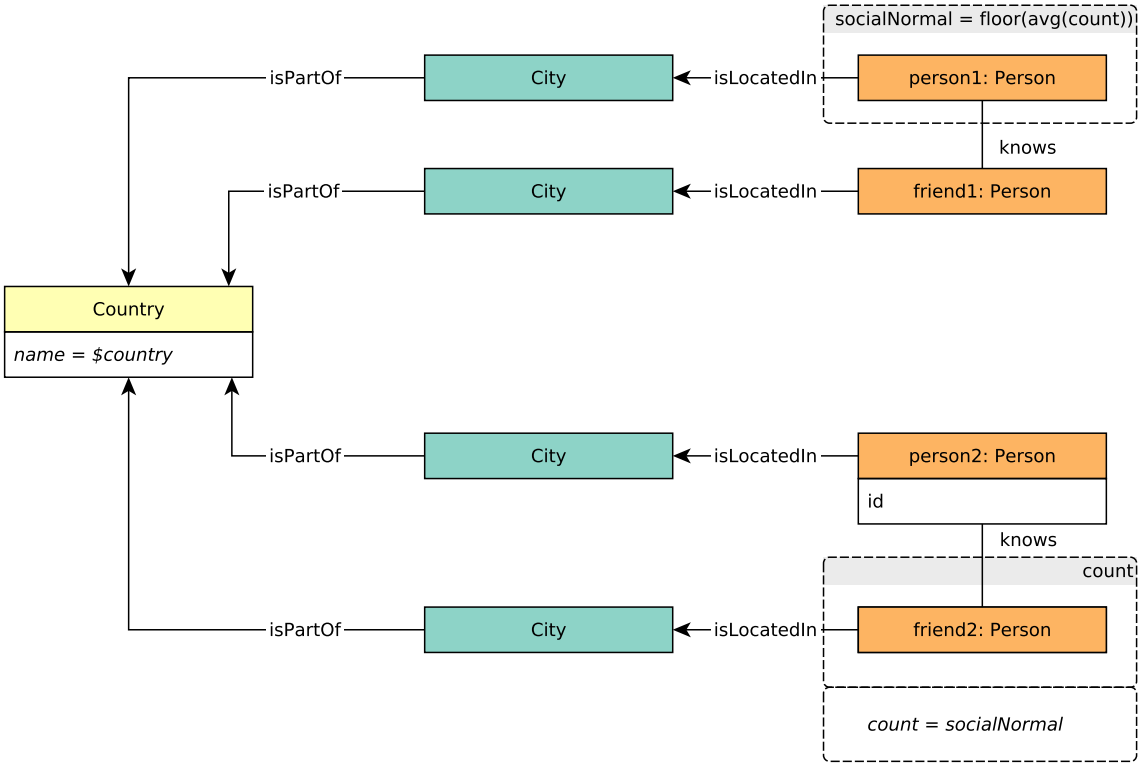
NULL érték kezelése A megszokott lekérdezési nyelvekkel ellentétben a SPARQL-ben nincsenek a megszokott értelemben vett NULL értékek: a változók lehetnek kötöttek, vagy kötetlenek, azaz vagy tartozik hozzájuk egy érték vagy nem. Az egyszerű gráfminták csak akkor illeszkednek egy adott részgráfra, ha minden változó kötött, azaz mindegyikhez tartozik egy érték. Ez alól az egyetlen kivétel a `OPTIONAL` kulcsszóval megadott opcionális gráfminták, mert az azokban megadott változók lehetnek kötetlenek. Fontos megjegyezni, hogy egy opcionális gráfmintában megadott változókat tekintve vagy mind kötött, vagy egyik sem, azaz az ilyen minták esetében is igaz az, hogy csak akkor illeszkednek, ha minden változójuk kötött. Például a 3.1. kódrészletben szereplő SPARQL lekérdezés eredményében az *?a*, *?b* és *?c* változók mindig kötöttek lesznek, míg a *?i* és *?j* közül vagy mindkettő kötött, vagy mindkettő kötetlen lesz az eredmény egy sorában.

```
SELECT
  ?a, ?b, ?c, ?i, ?j
WHERE
{
  ?a predicate1 ?b .
  ?b predicate2 ?c .
  OPTIONAL {
    ?a predicate3 ?i .
    ?b predicate4 ?j
  }
}
```

3.1. kódrészlet. Példa az OPTIONAL kulcsszóra

Nevesített gráfminták hiánya A SPARQL nem támogatja nevesített gráfminták létrehozását, így nem lehet többször felhasználható gráfmintákat létrehozni. Ez

kifejezetten kényelmetlenné teszi a 15-öshöz hasonló lekérdezések leírását. A 15-ös lekérdezés grafikus specifikációja látható a 3.3. ábrán. Jól látható, hogy az azonos országból származó barátok megszámolására kétszer is szükség van, azonban mivel a gráfmintát nem lehet elnevezni, így a lekérdezésben is kétszer kellett leírni ugyanazt a gráfmintát. Összetett gráfmintáknál a több példány szinkronban tartása kényelmetlen és sok hibalehetőséget hordoz.

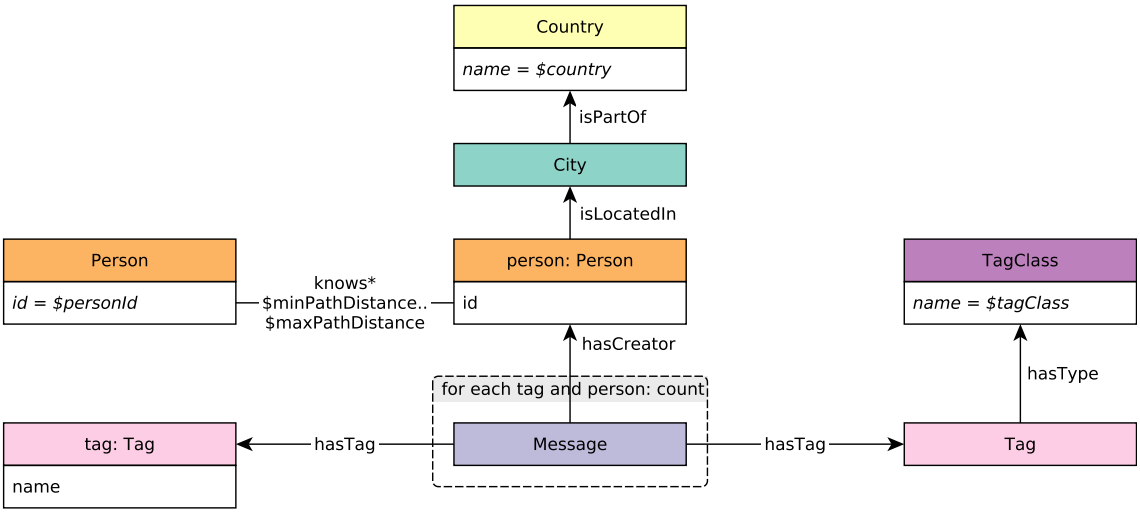


3.3. ábra. A BI15-ös lekérdezés

Deklarációk hiánya Mivel SPARQL nyelvben a változók deklarálása a változóra történő első hivatkozás, így a hosszabb lekérdezések esetében az esetleges elgépelésekkel újabb, természetesen nem kívánt változókat deklarálhatunk. Ezen a nyelv megváltoztatása nélkül egy megfelelő SPARQL szerkesztővel lehet segíteni, azonban a dolgozat készítése közben nem sikerült ilyen eszközt találni, a SPARQL szintaxis kiemelését is csak néhány egyszerű szöveges szerkesztő támogatja. A deklarációk hiánya egyébként több másik lekérdezőnyelvben is problémát jelent (pl. Datalog, Cypher, VQL).

Utak hosszának definiálása Bár SPARQL-ben van lehetőség változó hosszú útvonal-kifejezések leírására, azonban a Cypher `x..y` alakú kifejezésével megegyező nyelvi konstrukciót nem tartalmaz a SPARQL. Ez a nyelvi konstrukció azonban szükséges a 3.4. ábrán látható 16-os lekérdezés megvalósításához, mivel a benne lévő út hossza a lekérdezése paramétere. Ennek ellenére a 16-os lekérdezéshez készítettem egy olyan implementációt, amelyben az útvonal-kifejezést a keretrendszer szoftvermodulja generálja

le és illeszti be a SPARQL lekérdezésbe, így megteremtve a lehetőségét a nem triviális útvonal-kifejezéseket tartalmazó lekérdezések összehasonlítására.



3.4. ábra. A BI16-os lekérdezés

3.2.2. Interactive terhelési profil

3.2.2.1. SPARQL implementációk

Az interaktív terhelési profil CR és SR lekérdezéseinek két implementációja (az egyik az LDBC SNB implementációja, a másik pedig egy szabadon elérhető implementáció[7] [30]) 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 döntöttem, nem próbáltam meg a meglévő lekérdezéseket átalakítani. A munka során a specifikáció alapján implementáltam[8] 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 SPARQL szabvány nem tartalmazza, így ezeket a lekérdezéseket nem

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

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

implementáltam. Az implementáció elkészítése során *(1)* implementáltam a 27 darab lekérdezést SPARQL nyelven, *(2)* implementáltam a lekérdezések végrehajtását és a lekérdezések eredményének konvertálását végző szoftvermodulokat Java nyelven, *(3)* a keretrendszert használva sikeresen végrehajtottam 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 frissítettem, kiegészítettem, így megkönnyítve más felhasználóknak a saját lekérdezéseik és moduljaik elkészítését.

3.2.2.2. Cypher implementációk

A terhelési profilhoz nem volt elkészítve a Cypher implementáció, azonban egy másik nyílt forráskódú projektben[9] 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 tudtam használni a végleges implementáció[10] elkészítéséhez. A SPARQL implementációhoz hasonló módon itt is három részre 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.

3.2.2.3. Gremlin adatbetöltő

A dolgozat készítése során megpróbálkoztam a Cypher for Gremlin[11] 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 nagy méretűek, 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. 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().
```

[9] A Stanford Egyetemen fejlesztett implementáció: `https://github.com/PlatformLab/ldbc-snb-impls`
[10] Cypher implementáció: `https://github.com/ldbc/ldbc_snb_implementations/pull/57`
[11] `https://github.com/opencypher/cypher-for-gremlin`

```
    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.

4. fejezet

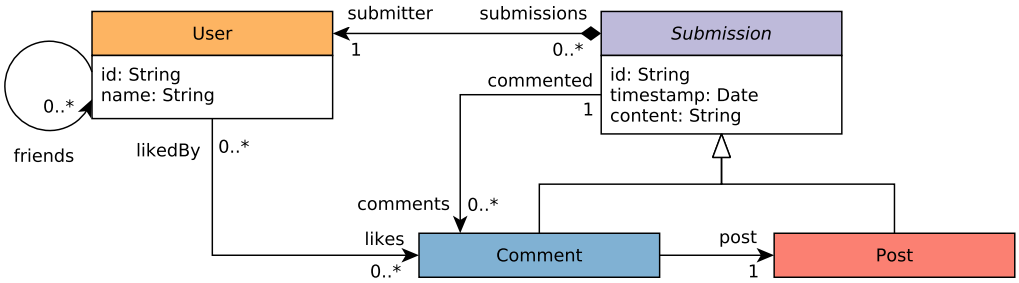
Transformation Tool Contest (TTC)

Az adatok transzformációja az alkalmazások egy széles rétegében központi szerepet tölt be. Ezen alkalmazások erősen függnek az elérhető transzformációs eszközök által nyújtott szolgáltatásokon és azok hatékonyságán. Az alkalmazások fejlesztőinek a számukra legalkalmasabb eszköz kiválasztása nehéz feladat, mivel számos modell- és gráftranszformációs eszköz érhető el manapság. Még a legtapasztaltabb szakemberek is általában egy vagy két eszközt ismernek alaposan, a többiről pedig csak kevés ismeretük van. Az évente megrendezésre kerülő Transformation Tool Contest (TTC)[1] célja, hogy összehasonlítsa a különböző transzformációs eszközök kifejezőerejét, használhatóságát és teljesítményét. 2007 óta minden évben különböző, a valós felhasználások inspirálta feladatokat tesznek közzé, amelyekre bármilyen eszközzel lehet megoldást készíteni. A feladatok a transzformációk felhasználásának széles köréből merítenek ötleteket, többek között:

- modell szinkronizáció és egyesítés (merge)
- modellek és transzformációk validációja
- szemantikus keresés
- programok manipulációja és transzformációja

Az utóbbi két évben pedig mindig volt egy feladat az *inkrementális nézetkarbantartás* témakörében is. A 2018-as feladatok egyike az LDBC SNB egy egyszerűsített sémája feletti két nézet inkrementális karbantartása volt. A következőkben ezt a feladatot ismertetjük részletesen.

[1] A TTC hivatalos weboldala: `https://www.transformation-tool-contest.eu/`



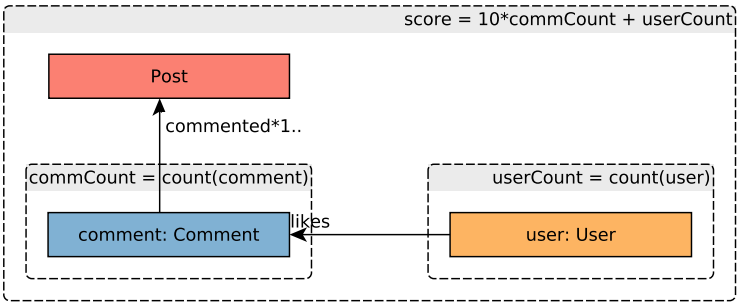
4.1. ábra. A TTC 2018 Közösségi háló feladatának adatsémája

4.1. TTC Közösségi háló feladat

Az TTC 2018 *Közösségi háló* feladatának (Social Media case) egyszerűsített LDBC SNB adatsémáját mutatja be a 4.1. ábra. A modellben tehát lehetnek felhasználók (User), akik különböző üzeneteket (Submission) írhatnak (submitter és submissions kapcsolatok): bejegyzéseket (Post) és hozzászólásokat (Comment). A felhasználók továbbá kedvelhetik (likedBy és likes kapcsolatok) a hozzászólásokat. A felhasználók között lehetőség van barátságok kifejezésére (friends kapcsolatok) is. A hozzászólásokat a felhasználók egy már meglévő üzenethez fűzhetik hozzá (comments és a commented kapcsolatok), tehát a hozzászólásokat reprezentálhatjuk olyan fákkal, ahol egy hozzászólás őse mindig az az üzenet, amelyikre reagáltak az adott hozzászólással. Ekkor belátható, hogy minden ilyen fában pontosan egy bejegyzés lesz, ami a fa gyökere, mivel hozzászólásokat csak meglévő üzenetekhez lehet fűzni. A feladatban a modellek frissítései csak bővítő jellegűek, azaz már meglévő objektumok és kapcsolatok nem törlődnek a modellből, csak újak adódnak hozzá.

Mivel a feladat kiinduló adatmodellje az NMF-fel (2.5.3.2. szakasz) egyszerűen beolvasható, ezért adott volt a lehetőség a feladatok Naiaddal történő megoldására, mivel a modellek beolvasásához használhattam az NMF-et. Ennek következtében a feladat kiváló lehetőségnek bizonyult a DAF számítási modell (2.5.3.4. szakasz) és a Naiad szoftverkönytár (2.5.3.3. szakasz) használhatóságának tesztelésére és teljesítménymérésére.

4.1.1. A Q1 lekérdezés



4.2. ábra. A Q1 lekérdezés grafikus ábrázolása

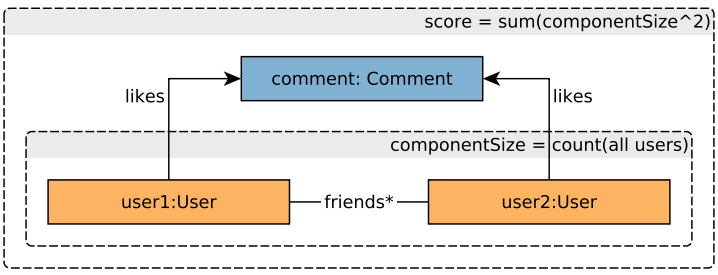
Ebben a lekérdezésben a bejegyzéseket pontozzuk a hozzájuk tartozó hozzászólások és kedvelések alapján. A bejegyzések minden hozzájuk tartozó hozzászólás után 10, a bejegyzéshez és hozzászólásokhoz tartozó kedvelések után pedig további 1 pontot kapnak. Egy felhasználó kedvelései több pontot is érhetnek, ha azok különböző üzenetekhez tartoznak. Az így pontozott bejegyzések közül keressük a három legnagyobb pontszámmal rendelkező bejegyzést. Egyenlő pontszámok esetén a korábban létrehozott bejegyzés élvez elsőbbséget. A lekérdezés grafikus szemléltetése látható a 4.2. ábrán.

A lekérdezés egy konkrét eredményét láthatjuk a 4.4. ábrán. Az ábra felső részén látható a kiindulási állapot `p1` bejegyzéssel, `c1` és `c2` hozzászolásokkal és `u1`, `u2`, `u3` és `u4` felhasználókkal. A `p1` bejegyzéshez tartozó pontszám az alábbi részpontszámokból adódik össze:

- 20 pont `c1` és `c2` hozzászólások miatt
- 2 pont a `c1` hozzászólásra adott kedvelések (`u2` és `u3` felhasználó) miatt
- 3 pont a `c2` hozzászólásra adott kedvelések (`u1`, `u2` és `u3` felhasználó) miatt

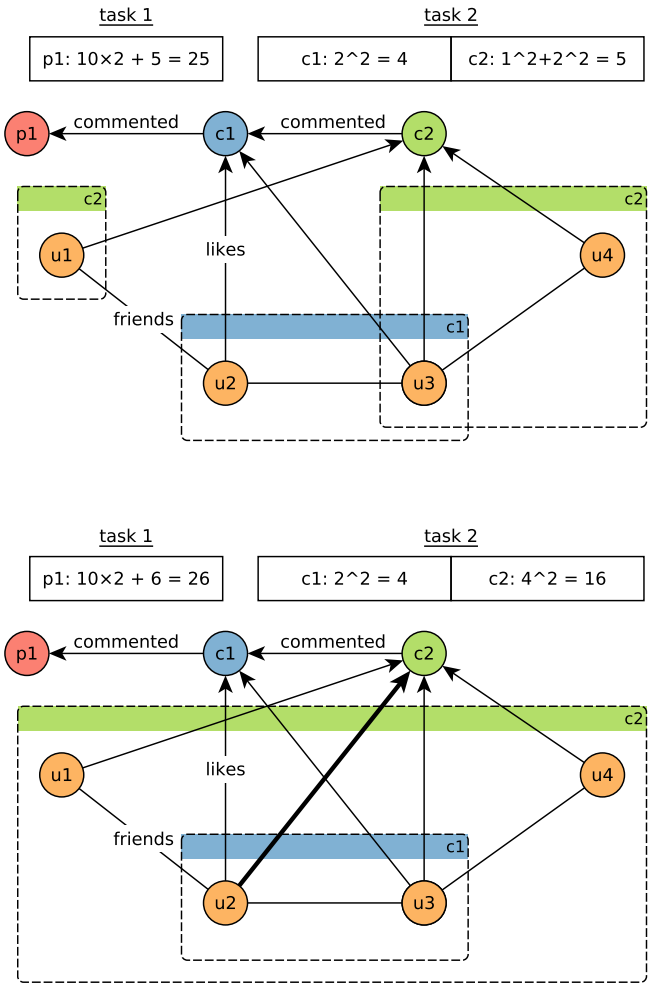
Összesen tehát 25 pontot rendelünk ebben az esetben a `p1` bejegyzéshez. Az ábra alsó felében a vastagon szedett nyíl jelenti a frissítés által hozzáadott új kedvelést. Ez a kedvelés a `c2` hozzászóláshoz tartozó kedvelés pontjait növeli meg 3-ról 4-re, így a `p1` bejegyzés pontszáma 26-ra módosul.

4.1.2. A Q2 lekérdezés



4.3. ábra. A Q2 lekérdezés grafikus ábrázolása

Ebben a lekérdezésben azokat a hozzászólásokat keressük, amelyek a felhasználók legnagyobb csoportjai által kedveltek. A felhasználó csoportokat a barátság kapcsolaton keresztül azonosítjuk: egy adott hozzászólást kedvelő felhasználók közül egy csoportba tartoznak azok a felhasználók, akik a barátság kapcsolatokon keresztül összefüggő komponenseket (strongly connected components, SCC) alkotnak. Tehát egy olyan gráfban keresünk összefüggő komponenseket, ahol a hozzászólást kedvelő felhasználók a csúcsok, és a közöttük lévő barátság kapcsolatok pedig az élek. Az összefüggő komponensek azonosítása után összegezzük a komponensek méretének négyzetösszegét. A lekérdezés grafikus szemléltetése látható a 4.3. ábrán.



4.4. ábra. A lekérdezések eredményének grafikus ábrázolása. A második modellben a vastag él az újonnan hozzáadott élet jelöli.

A 4.4. ábrán szintén láthatjuk a lekérdezés egy konkrét eredményét. A `p1` bejegyzés pontszáma a következő részpontszámokból adódik össze:

- $2 \times 2 = 4$ pont a `c1` hozzászólást kedvelő felhasználók miatt, mivel ők egy 2 elemű komponenst alkotnak
- $2 \times 2 + 1 \times 1 = 5$ pont a `c2` hozzászólást kedvelő felhasználók miatt, mivel egy 1 elemű (`u1`) és egy 2 elemű (`u3` és `u4`) komponenst alkotnak

Összesen tehát a `p1` bejegyzéshez 9 pontot rendelünk. Az ábra alsó felében, a frissítés után látható, hogy a `c2` hozzászólást kedvelő felhasználók immár egy 4 elemű komponenst alkotnak, tehát a `c2`-höz tartozó részpontszám 16-ra (4×4) változik, így a `p1` bejegyzés pontszáma pedig 20 pontra változik.

4.1.3. Megoldás ismertetése

A megoldás során a Naiad könyvtárban rendelkezésre álló DAF keretrendszert használtam. A keretrendszer a relációs lekérdezőnyelvekben használatos operátorok (`SELECT`, `JOIN`, `DISTINCT`, `COUNT`, `SUM` stb.) DAF alapú megvalósításán túl további operátorokat is definiál. A megoldásaim három olyan operátort használnak, amelyek túlmutatnak vagy nem megszokottak a relációs lekérdezőnyelvek világán:

- `FixedPoint`: A `FixedPoint` operátor használatával definiálhatjuk a DAF-ban említett ciklusokat. A ciklusok létrehozásakor meg kell adni azt a DAF-ot, amit a ciklus belsejében szeretnénk végrehajtani, valamint a DAF bemenetének számító adatforrást. Az operátor nevével összhangban egy iterációban addig hajtja végre a beágyazott DAF-ot, amíg annak kimenete két iteráció között megváltozik. A beágyazott DAF bemenetének típusa természetesen meg kell egyezzen a kimenetének típusával, így lehet cikluskénti végrehajtani. Az operátor által létrehozott ciklust az úgynevezett `LoopContext` objektum azonosítja. Az operátor használatára a Q1 esetében a bejegyzésekhez tartozó hozzászólások (tranzitív lezártak) keresésekor, a Q2 esetében pedig az összefüggő komponensek keresésekor volt szükség.

- `EnterLoop`: A `FixedPoint` operátor alapértelmezetten csak egy adatforrást támogat, azonban lehetőség van a ciklusokon belül több adatforrást is használni az `EnterLoop` operátor használatával. Bemenete egy adatforrást reprezentáló objektum és egy `LoopContext` objektum, kimenete pedig egy olyan adatforrást reprezentáló objektum, ami már használható a beágyazott DAF-ban is. Használatát a `FixedLoop` használatának szükségessége indokolta.

- `CoGroupBy`: A Naiad által nyújtott DAF keretrendszer nem definiál külső illesztéseket végrehajtó operátorokat, azonban a `CoGroupBy` operátor nagyon hasonló szemantikával rendelkezik, használatával megvalósíthatóak a külső illesztések. Az operátor bemenete két adatforrás és a hozzájuk tartozó kulcsderiváló függvények. Az operátor a kulcsok szerint csoportosítja a két adatforrásból érkező adatrekordokat, így <key, list1, list2> hármasokat állít elő belőlük, ahol a key a közös kulcs, list1 és list2 pedig rendre az első és a második adatforrásból a kulcshoz tartozó adatrekordok listája. A `CoGroupBy` operátort a hozzászólásokhoz tartozó kedvelések megszámolására használtam.

A Q1 lekérdezés differenciális adatfolyamának szemléltetése az F.2.1. ábrán látható, amelyeken megfigyelhető a `FixedPoint` és a `EnterLoop` operátorok kapcsolata.

5. fejezet

Kiértékelés

5.1. Adatbázis-kezelő rendszerek teljesítményének mérése az LDBC SNB keretrendszerrel

5.1.1. Motiváció

Bár a relációs adatbázis-kezelők a mai napig a legelterjedtebb adatbázis-kezelők[1], az újgenerációs 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. Ezért a tudományban már bizonyított módon kísérletek, mérések alapján próbálom összehasonlítani őket.

A dolgozatban összehasonlított technológiák kiválasztásában leghangsúlyosabb szempont a lekérdezőnyelveik voltak:

- SQL: Régóta használt, a legelterjedtebb relációs adatbázis-kezelők szabványos nyelve.
- SPARQL: A dolgozatban bemutatott gráf alapú lekérdezőnyelvek közül a legteljesebb matematikai háttérrel rendelkező, szemantikailag legtisztább lekérdezőnyelv.
- Cypher: A gráf alapú lekérdezőnyelvek között az egyik legnépszerűbb lekérdezőnyelv, véleményem szerint az egyik legintuitívabb és legkifejezőbb nyelv.

A Cypher kifejezőerejét és tömörségét az 5.1. táblázat is alátámasztja. A lekérdezések SQL-ben 37%, SPARQL-ben pedig 132%-kal több karaktert tartalmaznak, mint a Cypherben írt lekérdezések.

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

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

Nyelv	BI karakterszám	Int. karakterszám	Összesen
Cypher	13 657	19 021	32 678
SPARQL	32 417	43 548	75 965
SQL	30 417	14 563	44 980

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

Annak érdekében, hogy átfogó képet kapjak az elérhető gráf információs rendszerek teljesítményéről, egy összetett mérési sorozatot terveztem és futtattam.

5.1.2. Business Intelligence terhelési profil

A mérés során három implementációval végeztem méréseket a BI terhelési profil felhasználásával:

1. PostgreSQL: a PostgreSQL relációs adatbázis-kezelő rendszer.
2. Sparksee: a Sparksee tulajdonsággráf adatbázis-kezelő rendszer.
3. SDB2: egy szemantikus adatbázis.

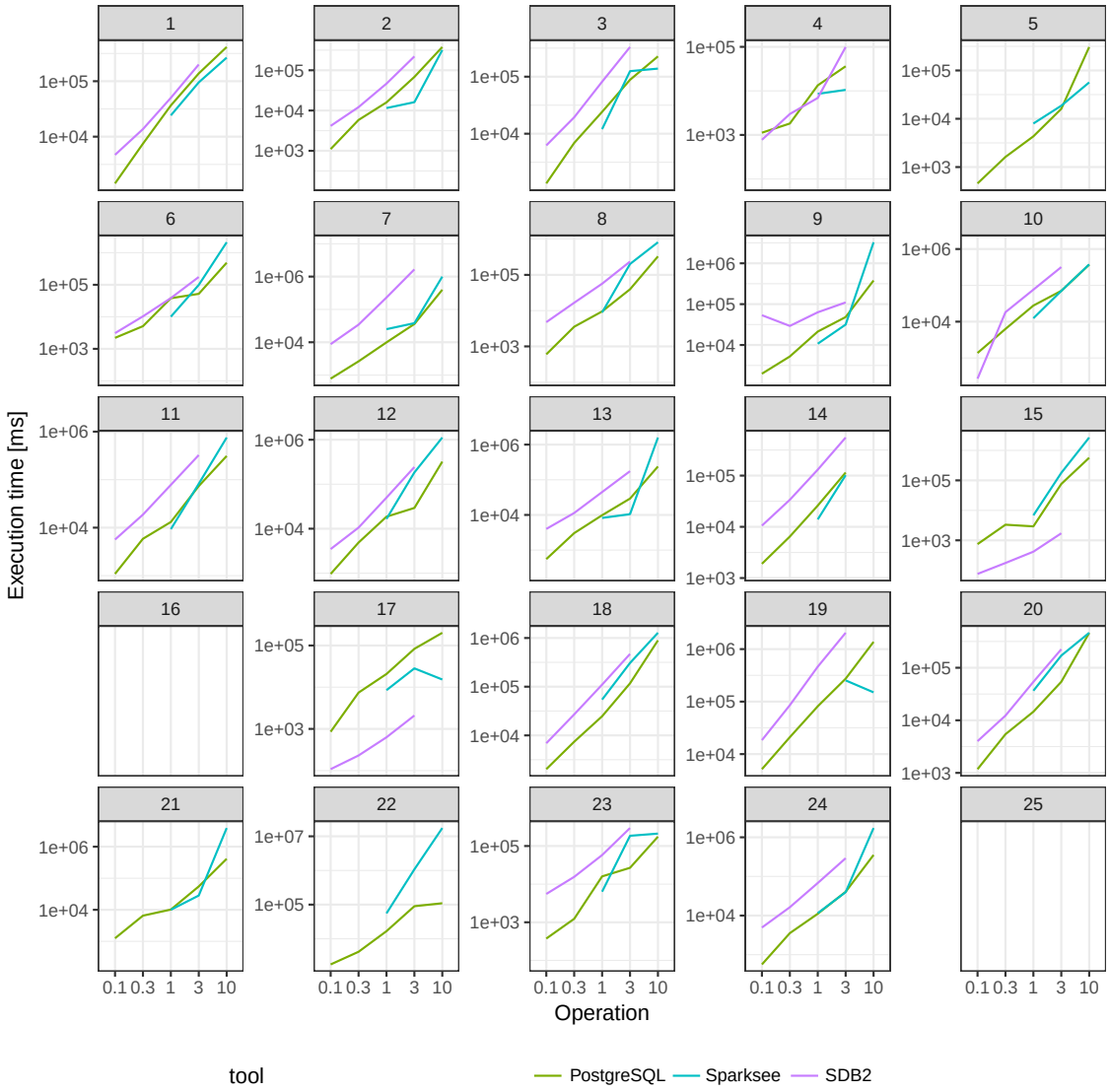
A meg nem nevezett adatbázis-kezelő rendszer eredményeit anonimizált módon adom közre. Ennek oka, hogy ugyan a lekérdezések minden esetben *validáltak* (azaz helyes eredményeket biztosítanak), de nem *auditáltak* (azaz a rendszerek gyártói nem vizsgálták meg az implementációt, így nem garantált, hogy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

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

5.1.2.1. 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 8 fizikai processzormagot (Intel(R) Xeon(R) E5-2673), és 256 GB memóriát tartalmaz. A háttértár egy SCSI interfészen kapcsolt 128GB-os SSD lemez. Az operációs rendszere 64 bites Ubuntu 16.04.

A PostgreSQL 10.5-ös verzióját használtam az SQL mérésére. A meg nem nevezett adatbázis-kezelő méréséhez az utóbbi 6 hónapban kiadott verziót használtam. A driver és opcionálisan a meg nem nevezett adatbázis-kezelő rendszer az OpenJDK 1.8.0_181-es verzióját használták. Az adatbázis-kezelő konfigurációját a dokumentációja alapján próbáltam optimalizálni, azonban az anonimitás miatt a pontos beállításokat nem közölhetem.



5.1. ábra. A lekérdezések összesített mérési eredményei

A lekérdezések a 3. fejezetben leírt módon kerültek ellenőrzésre az SF1-es adathalmazokon. A teljesítménymérés során a rendszereken először futtattam 100 darab bemelegítő (warmup) lekérdezést, majd 250 lekérdezés válaszidejét mértem meg. A lekérdezések véletlenszerűen választottak és végrehajtásuk szekvenciálisan, átlapolódás nélkül történt.

5.1.2.2. Eredmények értékelése

A mérési eredményeket az 5.1. ábra tartalmazza. A profil bonyolultságát jól szemlélteti, hogy több olyan lekérdezést (4-es és 14-es) is tartalmaz, amelyeket egyik adatbázis-kezelő sem tudott végrehajtani a legnagyobb adathalmazon, továbbá számos lekérdezés válaszideje elérte a több tíz másodpercet mindhárom rendszer esetében már az SF3-as adathalmazon is (1-es, 3-as, 11-es, 14-es, 15-ös, 18-as, 19-es, 22-es lekérdezés).

Egyértelműen látszik, hogy nehezebben megfogalmazható és összetettebb lekérdezéseket tartalmaz, mint az Interactive profil.

Érdemes megfigyelni, hogy a Sparksee két lekérdezés (17-es és 19-es) esetében is jobb válaszidőt produkált az SF10-es adathalmazon, mint az SF3-as adathalmazon.

5.1.3. Interactive terhelési profil

A mérés során négy implementációval végeztem méréseket az Interactive terhelési profil komplex lekérdezéseinek felhasználásával:

1. PostgreSQL: a PostgreSQL relációs adatbázis-kezelő rendszer.
2. Sparksee: a Sparksee tulajdonsággráf adatbázis-kezelő rendszer.
3. SDB1: egy szemantikus adatbázis.
4. SDB2: egy szemantikus adatbázis.

A két meg nem nevezett adatbázis-kezelő rendszer eredményeit a BI terhelési profilhoz hasonlóan anonimizált módon adom közre a már említett ok miatt: a lekérdezések *validáltak*, de nem *auditáltak*.

A teljesítménymérés során az egyes implementációk válaszidejét és skálázhatóságát mértem. 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.

5.1.3.1. 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 használtam az SQL implementációk mérésére.

A meg nem nevezett adatbázis-kezelők méréséhez (a BI profil méréséhez hasonlóan) az utóbbi 6 hónapban kiadott verziókat használtam. 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 próbáltam optimalizálni, azonban az anonimitás miatt a pontos beállításokat nem közölhetem.

A lekérdezések a rövid lekérdezésekkel és a frissítésekkel együtt a 3. fejezetben leírt módon kerültek ellenőrzésre az SF1-es adathalmazokon, több mint 13 ezer lekérdezés eredményének összehasonlításával.

[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
Sparksee	1000	1000	1000
SDB1	1000	1000	1000
SDB2	1000	750	40

5.2. táblázat. A implementációk mérése során futtatott lekérdezések száma

A mérés során a lekérdezéseket különböző behelyettesítési paraméterekkel futtattam az 5.2. táblázat szerinti darabszámban. Annak érdekében, hogy a mérések összideje ne legyen túl nagy, több korlátozást is kellett tennem. A SDB2 mérése során már SF3 esetében is csökkenteni kellett a lekérdezések számát, illetve az SF10-es adathalmazon egyáltalán nem mértem le az SDB2-t, mert nem futott le egy lekérdezés sem.

Mindegyik eszköz mérésnél a különböző méretű adathalmazokon legalább 20-szor futott egy lekérdezés (különböző paraméterekkel).

5.1.3.2. Eredmények értékelése

A mérési eredményeket az 5.2. ábra tartalmazza.

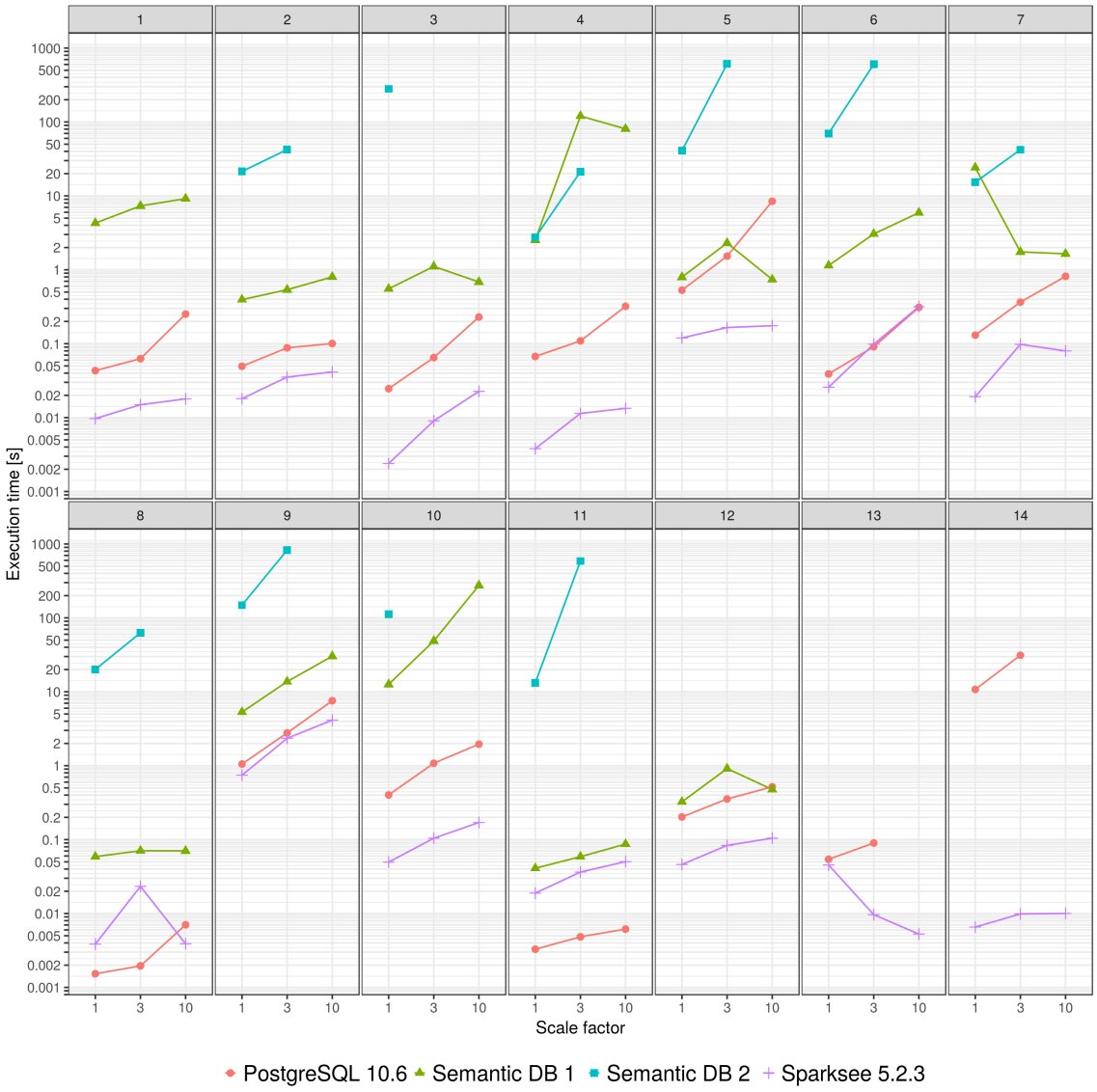
Az egyik legfontosabb eredmény, hogy a lekérdezések többségénél a legjobb eredményt a Sparksee érte el. Hozzá hasonlóan jól teljesített a PostgreSQL, ezzel bizonyítva, hogy a relációs adatbázis-kezelők a közel 50 éves fejlődésnek köszönhetően manapság is a leghatékonyabb adatbázis-kezelők közé tartoznak.

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 [22] 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 teljesítménynövekedés. A jelenségre nem sikerült konkrét hipotézist felállítani, de az említett komplex aggregációnak köze lehet a jelenséghez.

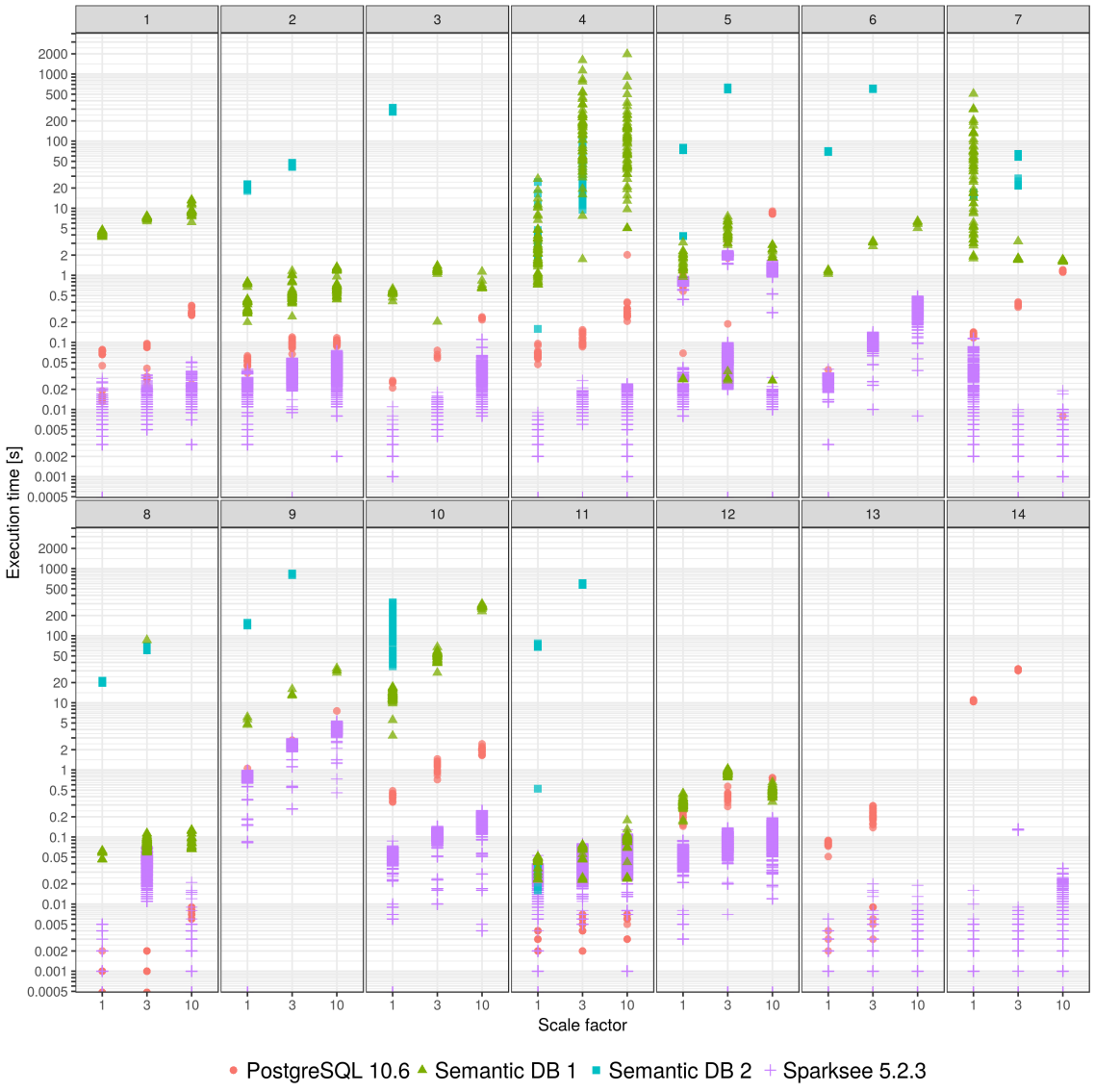
A 7-es lekérdezésnél szintén megfigyelhető a teljesítmény javulása a nagyobb adathalmazokon. Abban azonban más jellegű ez a javulás, mert az SF1 adathalmaz után, az SF3-as adathalmaznál figyelhető meg jelentős javulás, míg az SF3 adathalmaz után az SF10-es adathalmaznál csak minimális mértékű.

A lekérdezések részletesebb eredményeit ábrázolja az 5.3. ábra. Megfigyelhető, hogy az eszközök egy adott lekérdezés esetében is nagyságrendekkel eltérő válaszidővel rendelkezhetnek a lekérdezés paramétereitől függően. Kifejezetten látványos az SDB1 viselkedése a 4-es lekérdezés esetében, ahol a legjobb és a legrosszabb eredmény között több, mint két nagyságrendnyi az eltérés.

Hasonlóan érdemes megvizsgálni a Sparksee viselkedését az 5-ös lekérdezés esetében. A mérési eredmények mindhárom méretű adathalmazon két jól elkülöníthető csoportba sorolhatóak: *(1)* az 1 tizedmásodperc alatti és az *(2)* egy másodperc körüli válaszidők



5.2. ábra. A lekérdezések összesített mérési eredményei



5.3. ábra. A lekérdezések részletes mérési eredményei

csoportjába. Előbbiekre lehetséges magyarázat lehetne az, hogy a lekérdezés ebben az esetben üres eredményhalmazt ad vissza, de tapasztalataim szerint ez nem igaz.

Az említett jelenségek többségére sajnos hipotéziseket sem tudtam felállítani. Ennek egyik oka, hogy a rendszerek belső működéséről nincs információm, ezért nem tudok következtetni a lehetséges okokra.

5.2. Differenciális adatfolyamok teljesítménymérése TTC-vel

5.2.1. Motiváció

A DAF számítási modell pozitív tulajdonságai miatt (elosztott, nagy teljesítményű, iteratív és inkrementális számítások támogatása) a hatékony INK egyik lehetséges megoldása lehet. A gráfadatbázisokban való felhasználásához azonban fontos tisztában lenni azzal, hogy a gráf alapú adathalmazokon milyen teljesítményt nyújt. Ez a cél tökéletesen illeszkedik a TTC 2018-as versenyfeladatához, ahol a cél az, hogy minél jobb teljesítményű (skálázódás és válaszidő szempontjából is) megoldást készítsünk. Ezért elkészítettem a feladat megoldását a Naiad szoftverkönytár által nyújtott DAF keretrendszerrel, teljesítményét pedig összehasonlítottam a többi megoldás teljesítményével. Így a mérés során az alábbi eszközök teljesítményét hasonlítottam össze:

- DAF implementáció a Naiad könyvtár használatával, 1, 2, 4, 6 és 8 szálon történő feldolgozással
- NMF alapú inkrementális megoldás
- VIATRA alapú inkrementális megoldás
- Xtend nyelven[4] manuálisan megvalósított inkrementális megoldás

Az Xtend egy Java forráskódra forduló programozási nyelv. Szintaxisa nagyon hasonló a Java nyelvhez, azonban számos szintaktikai édesítőszerrel egészíti ki annak szintaxisát.

5.2.2. Mérési elrendezés

A méréseket egy számítógépen végeztem el a TTC által nyújtott keretrendszer használatával. A számítógép 6 fizikai processzormagot (Intel Core i7-8700K @ 3.70GHz) és 32 GB memóriát tartalmaz. A háttértár SATA2 interfészen csatolt 512GB-os SSD lemez. Az operációs rendszer 64 bites Windows 10.

A Java környezetből az Oracle Java 8u191-es, a .NET Core 2.2-es és a .NET Framework 4.7.2-es verzióját használtam a mérés során. Az eszközök mérése során a különböző adathalmazokon 3-szor futtattam minden eszközt, az eredményeket aggregáltam az értékek mértani közepét (geometric mean) véve, [11] javaslatának megfelelően. Egy futtatás alkalmával a TTC által az adott modellhez tartozó összes (20 darab) frissítést

[4] `https://www.eclipse.org/xtend/`

végrehajtottam és a modell betöltése és minden frissítés után lefuttattam a mért lekérdezést. A két lekérdezés mérése szekvenciálisan, átlapolódás nélkül történt történt.

A mérés során használt modelleket egy számmal azonosítjuk. A modellek számozása megegyezik a 2 hatványaival, mivel a legkisebb modellt önkényesen 1-gyel jelölték és a modellek mérete nagyjából duplázódik. A modellek legfontosabb jellemzőit az 5.3. táblázat mutatja be.

Modell	Felhasználók	Bejegyzések	Hozzászólások	Méret (MB)
1	80	554	638	0.154
2	118	889	1 059	0.252
4	190	1 845	2 305	0.538
8	204	2 270	5 027	0.984
16	394	5 518	9 136	2.1
32	595	10 929	18 689	4.3
64	781	18 083	38 753	8.3
128	1 158	37 228	75 605	18
256	1 678	74 668	145 821	34
512	2 606	167 299	267 160	68
1 024	3 699	314 510	526 006	136

5.3. táblázat. Az egyes modellekben szereplő modellelemek darabszáma (db) és a kezdeti modell mérete MB-ban

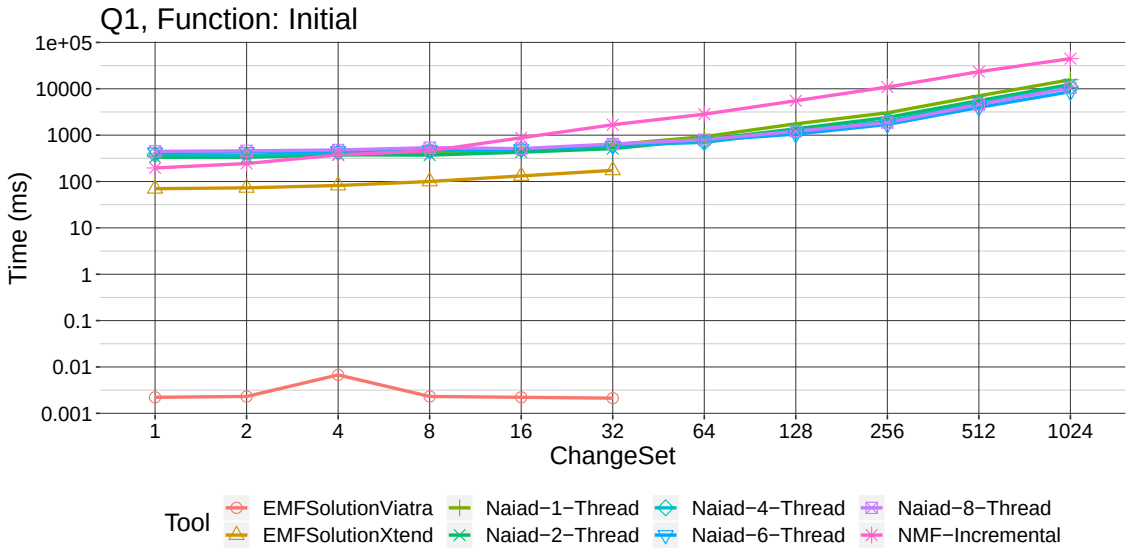
A VIATRA és Xtend alapú megoldásokat nem mértem a nagyobb adathalmazokon, mivel a teljesítménymérés túl hosszú ideig tartott volna.

5.2.2.1. Eredmények értékelése

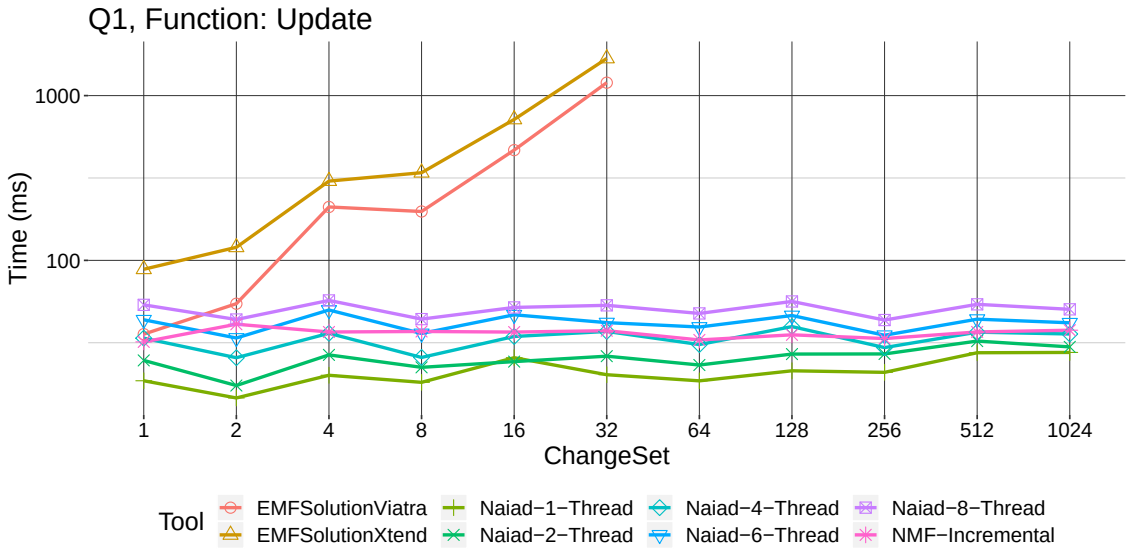
Az eredmények értékelésénél fontos kiemelni, hogy a mért lekérdezések validáltak, de nem auditáltak (azaz az eszközök fejlesztői által nem vizsgáltak, így nem garantált, hogy optimális teljesítményt nyújtanak).

Modell	Naiad-1	Naiad-2	Naiad-4	Naiad-6	Naiad-8	NMF
1	68	93	141	194	241	48
2	70	101	152	208	251	54
4	74	107	166	216	269	70
8	83	113	176	233	287	84
16	96	128	190	254	316	155
32	113	147	209	274	334	274
64	151	189	255	316	385	445
128	221	257	324	395	454	853
256	399	398	471	528	602	1 408
512	658	668	729	815	865	3 346
1 024	1 308	1 285	1 300	1 351	1 418	6 424

5.4. táblázat. A Q1 lekérdezés utolsó kiértékelése utáni memóriahasználat a különböző méretű modellek és eszközök esetében MB-ban



5.4. ábra. A Q1 lekérdezés első kiértékeléseinek válaszidői

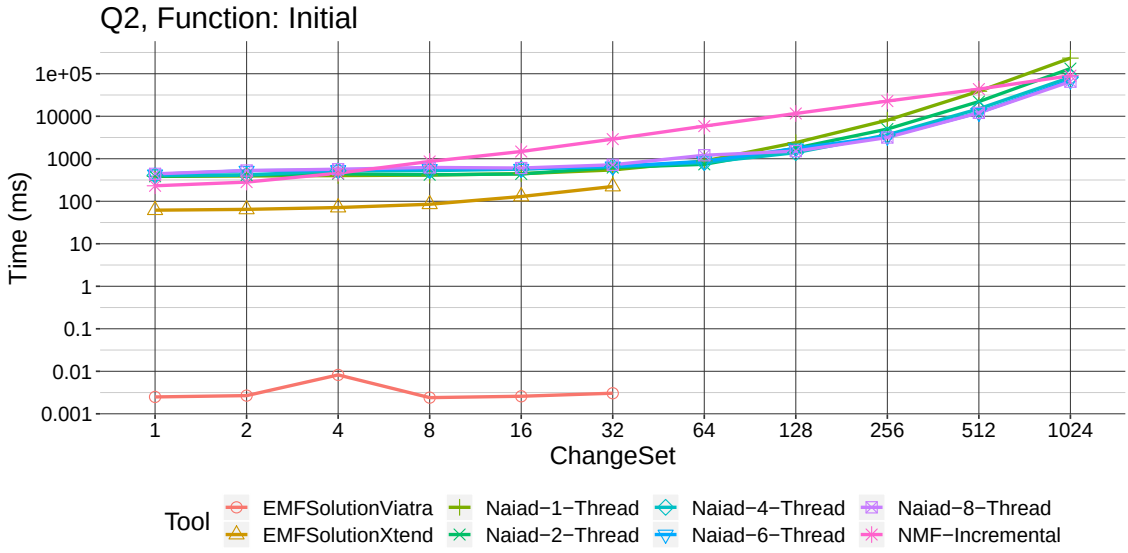


5.5. ábra. A Q1 lekérdezés frissítések utáni kiértékeléseinek válaszidői

Q1 lekérdezés Az 5.4. ábra mutatja a Q1 lekérdezés első kiértékeléseinek válaszidejét a modell méretének függvényében. A legszembetűnőbb eredmény a VIATRA rendkívül alacsony, a többi eszköz válaszidejénél nagyjából 4 nagyságrenddel kisebb válaszideje. Ezt az okozza, hogy a VIATRA már a betöltés során elkezdi a kiértékelést, és az első lekérdezésnél valószínűsíthetően már rendelkezésre álló eredményeket adja vissza. A többi eszköz közül az Xtend-es megoldás valamivel gyorsabb az Naiad és NMF alapú megoldásoknál. Az 5.5. ábrán azonban jól látszik, hogy a VIATRA és az Xtend alapú megoldás is jelentősen rosszabbul teljesít, mint a Naiad és NMF alapú megoldások. Jól látható, hogy a Naiad alapú megoldások esetében a válaszidő arányos a feldolgozásra használt szálak számával. Ennek több magyarázata is lehet. Egyik lehetséges magyarázat az, hogy a számítás olyan rövid ideig tart (<100ms), hogy a többszálú feldolgozás akkora

plusz komplexitást (overheadet) jelent, hogy ekkora méretű modellek esetében egyszerűen nem éri meg. Egy másik lehetséges magyarázat az, hogy egyszerűen rosszul működik a számítás elosztásáért felelős logika. Ennek eldöntéséhez nagyobb modellekre lenne szükség, ahol a számítások ideje több nagyságrenddel több időt vesz igénybe.

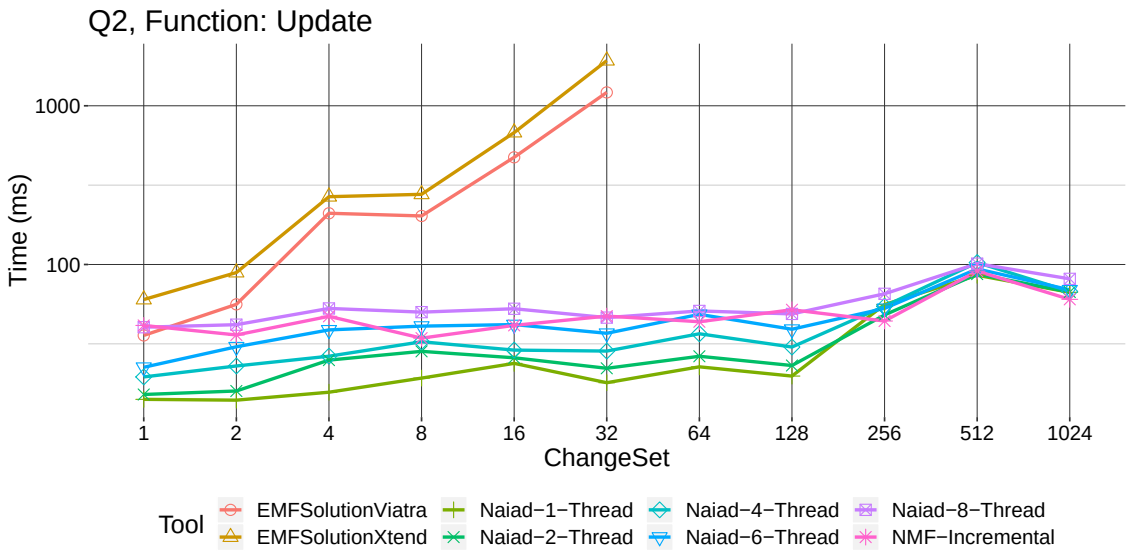
A Naiad teljesítményéről egyébként elmondható, hogy a vizsgált modellek esetében jól skálázódik az NMF alapú megoldással együtt, gyakorlatilag a legkisebb és legnagyobb modellen is azonos válaszidővel rendelkeznek. Érdemes megfigyelni az 5.4. táblázaton, hogy a Naiad alapú megoldások az NMF alapú megoldáshoz képest a kisebb modellek esetén több memóriát használnak, azonban ez nagyjából a 32-es modell környékén megváltozik. A legnagyobb modell esetén az NMF alapú megoldás nagyjából ötször több memóriát használ mint a Naiad alapú megoldások. Ahogy az 5.2.2. fejezetben már kitértem rá, a VIATRA és az Xtend alapú megoldások nagyjából lineárisan skálázódnak a modell méretéhez viszonyítva.



5.6. ábra. A Q2 lekérdezés első kiértékeléseinek válaszidői

Modell	Naiad-1	Naiad-2	Naiad-4	Naiad-6	Naiad-8	NMF
1	60	71	88	114	123	47
2	77	113	140	197	208	55
4	84	127	186	252	300	74
8	95	137	210	290	351	120
16	107	155	243	335	414	199
32	124	181	275	370	464	344
64	168	224	330	444	555	681
128	251	318	444	555	679	1301
256	433	527	639	777	905	2537
512	929	997	1 148	1 251	1 410	4 882
1 024	2 281	2 398	2 546	2 679	2 836	9 864

5.5. táblázat. A Q2 lekérdezés utolsó kiértékelése utáni memóriahasználat a különböző méretű modellek és eszközök esetében MB-ban



5.7. ábra. A Q2 lekérdezés frissítések utáni kiértékeléseinek válaszidői

Q2 lekérdezés Az 5.6. ábrán látható a Q2 lekérdezés első kiértékeléseinek válaszidői a modell méretének függvényében. Az eredmények trendje nagyjából megegyezik a Q1 lekérdezés eredményével, azonban fontos észrevenni, hogy a legnagyobb modellen már egy nagyságrenddel több időt vett igénybe az első kiértékelés a Naiad és az NMF alapú megoldások esetében is. A VIATRA kiugróan alacsony válaszidejének oka a Q1 lekérdezésnél leírt betöltési mechanizmus. Az 5.7. ábrán pedig a frissítések utáni kiértékelések válaszidejét tekinthetjük meg, valamint az 5.5. táblázatban a használt memória mennyiségét. A válaszidőkről és a memóriahasználatról is nagyjából hasonló megállapítások mondhatóak el, mint a Q1 lekérdezés esetében.

Az egyetlen különböző és egyben érdekes dolog, ami a Naiad és az NMF megoldásra is jellemző a 256-os megoldástól kezdve: a válaszidők elkezdenek konvergálni, azonban nem kezd el jelentősen nőni a válaszidő. Mivel mindkét megoldás C# alapú, ezért egy lehetséges magyarázat az, hogy a *just-in-time* fordító (JIT) úgy értékeli a program futtatását, hogy érdemes az egyik gyakran használt kódrészletét alaposabban optimalizálni, ezért azt a kódrészletet újrafordítja, ami összemérhető nagyságrendű az egyébként nagyon alacsony válaszidővel.

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 a Spark adatfolyam-orientált (streaming) programozási modellje miatt nem alkalmas módosítás és törlés műveletek kezelésére.

G-CORE A G-CORE [1] 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 [2].

VIATRA Query Language A gráfmintaillesztést széles körben használják a modellvezérelt technológiákban, például a VIATRA keretrendszerben [41]. A keretrendszer saját lekérdezőnyelve egy Dataloghoz hasonló nyelv, a VIATRA Query Language (VQL), amely támogatja a gráfminták komponálását, a rekurzív minták megfogalmazását és az aggregációt. A VIATRA keretrendszer a Rete algoritmust használja a gráfmodellek hatékony ellenőrzésre és transzformációjához. Az INCQUERY-D egy elosztott inkrementális gráflekérdező [38], amely lekérdezőnyelve a VQL-en alapul és RDF gráfok lekérdezését teszi lehetővé.

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

A VIATRA egy korábbi verziójában készült egy prototípus alkalmazás [4], ami inkrementális gráftranszformációt végez relációs adatbázisokon. A prototípus gráfmintákból SQL lekérdezéseket készít a kezdeti illeszkedési halmazok meghatározásához. 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 tanulmányoztam, amelyek lehetővé teszik gráf információs rendszerekben futtatott lekérdezések teljesítményének vizsgálatát. Az általam fontosnak vélt keretrendszerek legmeghatározóbb tulajdonságait mutatom be ebben a fejezetben.

LDBC Semantic Publishing Benchmark Az LDBC egy másik teljesítményérési keretrendszere a Semantic Publishing Benchmark [21] (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.

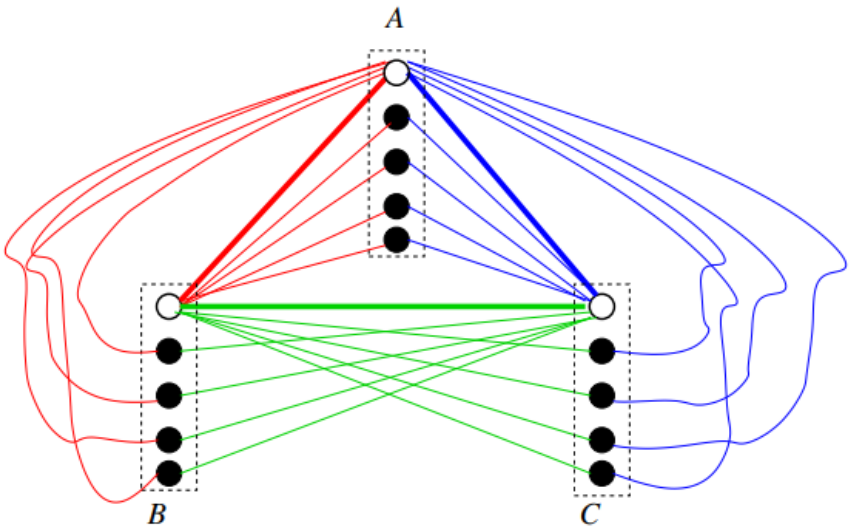
Azon túl, hogy csak RDF adathalmaz generálására alkalmas, az LDBC SPB a teljesítménymérés során olyan csak RDF-ben elérhető funkciókat használ, mint a következtetés (*inferencing*), azaz a meglévő kapcsolatok és előre definiált szabályok alapján új kapcsolatok létrehozása. A keretrendszer ezen tulajdonsága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

Train Benchmark A Train Benchmark [39] a modellvezérelt technológiák által inspirált teljesítménymérési keretrendszer, amely adatmodellje a vasúthálózat és a hozzá kapcsolódó érzékelő és biztosító berendezések hálózatát 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, útvonal-kifejezések vagy éleken definiált tulajdonságok teljesítménymérését.

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

6.3. Illesztések optimalizációja gráfadatbázis-kezelőkben

A relációs adatbázis kezelők elterjedésének köszönhetően az elmúlt évtizedekben egyre hatékonyabb illesztési algoritmusokat dolgoztak ki. A lekérdezések végrehajtási tervének optimalizálása a végrehajtási terv elemei közötti egymásra épülésének (kb. végrehajtási sorrend, amolyan termelő-fogyasztó viszony) – a szelekciós tényező becsült értékén alapuló – ekvivalens átalakításain alapszik. A szelekciós tényező egy illesztés esetén az eredményként előálló reláció sorainak számának aránya az alaprelációk Descartes-szorzatának sorainak számához viszonyítva. Formálisan $s = \frac{|R_1 \bowtie R_2|}{|R_1 \times R_2|}$, ahol s a szelekciós tényező, R_1 és R_2 pedig az illesztés két alaprelációja. Az illesztési sorrend nagyon fontos a hagyományos relációs lekérdezésekben, így egy jó illesztési sorrenddel sokat lehet nyerni, a hiányában bukni. Annak ellenére, hogy a relációs adatbázis kezelőkben az ezen alapuló algoritmusok és módszerek hatékonynak bizonyultak, a gráfadatbázis kezelők tekintetében sok esetben nem bizonyultak megfelelőnek. A problémát a gráfos lekérdezések tulajdonságai, a tipikusan használt gráfminták okozzák, amelyek általában nincsenek jelen a relációs lekérdezésekben, pl. a körkörös gráfminták, vagy az adatok szokatlan eloszlása.

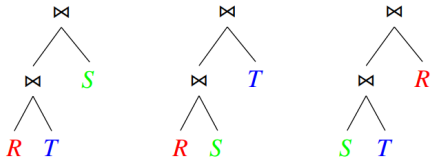


6.1. ábra. ∘ királyok, • alattvalók [29]

6.3.1. Körkörös lekérdezés

A relációs adatbázissal szemben a gráfadatbázisok esetében gyakoriak a köröket tartalmazó lekérdezések. Mivel a relációs adatbázis-kezelőkben a bináris (két alaprelációt tartalmazó) illesztések beváltak, így számos gráfadatbázis is bináris illesztéseket használ az illesztések elvégzésére, köztük a tanszéken fejlesztett ingraph [24] inkrementális gráfadatbázis-kezelő rendszer is. A bináris illesztések rossz teljesítményét a 6.1. ábrán látható gráfon mutatom be: tegyük fel, hogy a három, szaggatott vonallal jelölt királyságban egyik király (üres körök) sem ismeri a saját alattvalóit (teli körök), de minden más királyság összes

alattvalóját és a királyukat is ismeri. Ha ebben a gráfban a pontosan 3 darab, páronként különböző színű élt tartalmazó köröket keressük, akkor bináris illesztéseket használva a 6.2. ábrán látható végrehajtási tervek egyikét kell végrehajtanunk. Az ábrán R, S és T az azonos színű élhalmazokat jelentik. Ekkor a második (a végrehajtási terveben legfelső) illesztés alaprelációinak 25 (az első illesztés eredménye) és 9 (a harmadik színű élek) darab sora lesz, az illesztés eredménye pedig 13 darab (3×4 darab pontosan kettő királyt tartalmazó kör és 1 darab mindhárom királyt tartalmazó kör) sorból fog állni. Ezek alapján a szelekciós tényező $s = \frac{13}{25 \times 9} \approx 0.058$. Az alacsony szelekciós tényező miatt ebben az esetben a bináris illesztésnél létezik hatékonyabb módszer is [29], a WCOJ (worst-case optimal join) algoritmusok [28]. Ezek az algoritmusok egyszerre nem kettő, hanem több, akár tetszőleges számú reláció illesztését végzik el egyszerre. Egy ilyen algoritmus a Leapfrog Triejoin [42], ami olyan attribútumok illesztését képes hatékonyan elvégezni, amelyeken értelmezett a rendezés művelet. A WCOJ algoritmusok kifejezetten az ilyen lekérdezések hatékony végrehajtását segítik. Véleményem szerint *az ingraph teljesítményét jelentősen növelné*, ha a bináris illesztések mellett WCOJ algoritmusokat is megvalósítana.



6.2. ábra. A lehetséges végrehajtási tervek bináris illesztések esetén [29]

6.3.2. Eredmények számosságának becslése

Mivel az illesztések szelekciós tényezőjének pontos kiszámítása túl költséges művelet, ezért az adatbázis-kezelő rendszerekben gyakran használnak különböző becsléseket (*cardinality estimation*) a szelekciós tényező meghatározására, hogy a lekérdezések végrehajtási tervét ennek figyelembevételével készítsék el. Az ilyen becslések két nagy csoportba sorolhatóak [23]: az *(1)* aggregált adatokon (hisztogramokon) alapuló és a *(2)* mintavételezésen alapuló becslések.

Az *(1)* esetben az adatbázis-kezelő rendszereknek fel kell építeniük és karban kell tartaniuk az alaprelációkról a szükséges statisztikákat. Ahhoz, hogy az aggregált adatokból megfelelő pontossággal lehessen megbecsülni a szelekciós tényezőt, sok esetben pontosan ismerni kell az adatok struktúráját, esetleg részletesebb statisztikákat kell tárolni és frissíteni. Extrém nagy adathalmazok esetén az is előfordulhat, hogy már a statisztikákat is becslésekkel állítják elő, így növelve a becslés pontatlanságát.

A *(2)* esetben a bemeneti relációk néhány rekordját kiválasztjuk, majd ezek illesztése után az eredeti relációk illesztésének szelekciós tényezőjét becsüljük a kiválasztott elemek szelekciós tényezőjével. A mintavételezésen alapuló eljárások nem követelik meg az adatbázisra vonatkozó statisztikák tárolását és karbantartását, valamint korrelált és nem egységes adatok esetében robusztusak [17]. Továbbá a mintavétel alapú eljárások lehetővé

teszik a becslési hibák értékelését és ellenőrzését. A mintavételezés történhet több lépésben is: minden lépésben kiválasztunk $n \geq 1$ rekordot mindkét relációból, ezeket illesztjük majd a mintavételezési lépések után azok eredményeit kombináljuk és következtetünk az eredeti illesztés szelekciós tényezőjére. Az egyes lépések után a mintavételezett rekordokat a későbbi lépésekben felhasználhatjuk, vagy akár el is dobhatjuk. A mintavételezési lépések száma lehet egy előre rögzített szám, vagy a mintavételezés folytatódhat addig, amíg egy bizonyos valószínűséget el nem ér a becslés pontossága (azaz amíg nem vagyunk elég biztosak a becslés helyességében).

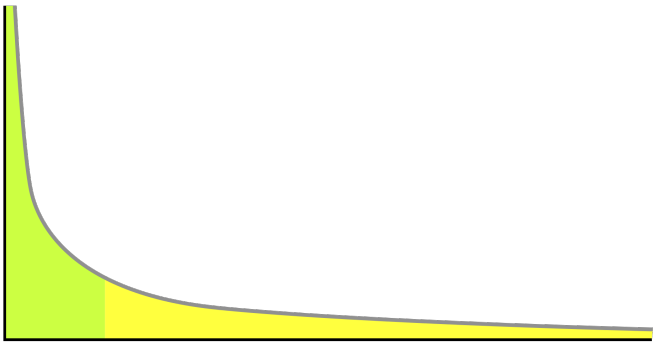
A gráfadatbázisok egy fontos tulajdonsága, hogy a csúcsok fokszámeloszlása erősen eltolt (skewed). Míg relációs adathalmazok gyakran normál vagy teljesen véletlenszerű eloszlást követnek, addig előbbiek általában a hatványtörvény-eloszlást (power law distribution) követik [3]. A hatványtörvény-eloszlás egy olyan valószínűség-eloszlás, melynek sűrűségfüggvénye

$$P(X > x) \sim L(x)x^{-(\alpha+1)}$$

alakú, ahol $\alpha > 0$, és $L(x)$ egy lassan változó függvény, amire minden pozitív r esetében igaz, hogy:

$$\lim_{x\to\infty} \frac{L(r \cdot x)}{L(x)} = 1.$$

Egy példa hatványtörvény-eloszlás a 6.3. ábrán[3] látható.



6.3. ábra. Egy példa határtörvény-eloszlás, sárgával van jelölve az eloszlás farka (*tail*), zölddel pedig a domináns fej (*head*) rész – a jól ismert 80-20 szabályhoz hasonlóan.

Javaslat az ingraph rendszer továbbfejlesztésére Ezen ismeretek alapján az ingraph rendszerben szerintem a mintavételezésen alapuló becslési módszert érdemes implementálni, figyelembe véve az *end-biased* [10] módszert. Ennek a módszernek a lényege, hogy az egyes elemek gyakoriságát nyilvántartja, és az alapján választja ki a mintavételezés során az elemeket. Egy adott T gyakoriság felett minden elemet belevesz a mintába, alatta pedig a gyakoriságával arányos valószínűséggel választja bele a mintába az egyes elemeket. Ez a módszer biztosíthatja, hogy a legmeghatározóbb (leggyakoribb)

[3]Forrás: Wikipédia, `https://en.wikipedia.org/wiki/Power_law`

elemek kiválasztásra kerüljenek, azonban a kevésbé gyakori, azonban összességében az adathalmaz jelentős részét alkotó – a 6.3. ábrán sárgával jelölt – elemek közül is megfelelő mennyiségű a mintába kerüljön, így növelve a becslés pontosságát. A módszer lehetőséget biztosít arra, hogy megfelelő hash függvények választásával növeljük annak a valószínűségét, hogy egy adott kulcshoz tartozó elem vagy minden alaprelációban kiválasztásra kerüljön, vagy egyikben sem.

7. fejezet

Összefoglalás és jövőbeli tervek

Dolgozatomban megvizsgáltam azt a kérdést, hogy a manapság elérhető gráf információs rendszerek milyen kifejezővel, teljesítménnyel rendelkeznek, és vajon a hagyományos relációs adatbázisok megfelelő alternatívát nyújthatnak-e gráflekérdezések kiértékelésére [30]. Ezen kívül megvizsgáltam az illesztési algoritmusok legnagyobb kihívásait a gráfadatbázisokban, és ezek alapján javaslatokat tettem az ingraph rendszer fejlesztésére. Az inkrementális nézetkarbantartás témakörében megvizsgáltam a differenciális adatfolyamokat, mint lehetséges inkrementális számítási modellt.

7.1. Kontribúciók

Teljesítménymérés A dolgozat készítése során elkészítettem az LDBC Social Network Benchmark *Business Intelligence* terhelési profiljának lekérdezéseihez a SPARQL nyelvű implementációkat és az *Interactive* terhelési profiljának lekérdezéseihez a Cypher és a SPARQL nyelvű implementációkat, illetve implementáltam a keretrendszerhez szükséges szoftvermodulokat három gráf alapú adatbázis-kezelő eszközhöz. Implementáltam továbbá a Gremlin nyelvet támogató eszközökhöz az adatok betöltését elvégző alkalmazást. Az elkészített implementációkkal lemértem több eszköz teljesítményét és értékeltem a kapott eredményeket. Az eredmények rávilágítottak arra, hogy a hagyományos relációs adatbázisok gráf jellegű terhelési profilok esetén is versenyképesek.

Inkrementális nézetkarbantartás Az inkrementális nézetkarbantárssal kapcsolatban megvizsgáltam a lehetséges megközelítéseket, és implementáltam a Transformation Tool Contest verseny 2018 feladatai közül a *Közösségi háló* feladat két lekérdezését differenciális adatfolyamokat felhasználva. Az elkészült implementációt összemértem a versenyre készített megoldások közül hárommal. Az eredmények alapján a differenciális adatfolyamok jól skálázódnak a válaszidőt és a memóriahasználatot tekintve is.

7.2. Jövőbeli tervek

Céljaim között szerepel a teljesítménymérési keretrendszer bővítése, azaz további eszközökhöz szükséges szoftvermodulok implementálása. Kiemelt célom a lekérdezések teljesítményének mérése a *Cypher for Spark*, valamint a *Cypher for Gremlin* eszközökkel. A *Business Intelligence* terhelési profil kapcsán az LDBC SNB munkacsoport célja a profil bővítése kötegelt (batch) frissítésekkel, amelyek az additív frissítéseken túl törlő jellegű frissítéseket is tartalmaznak. Ezen bővítések kidolgozásában és implementálásában is tervezek részt venni. Terveim között szerepel továbbá az *Interactive* terhelési profil komplex lekérdezésein túl a rövid lekérdezések (SR1, . . . , SR7) és a frissítések (U1, . . . U8) teljesítménymérése és publikálása is.

A differenciális adatfolyamhoz kapcsolódóan tervezem implementálni a dolgozatomban bemutatott TTC feladatot a differenciális adatfolyamokat támogató Differental Dataflow nevű, Rust nyelvű keretrendszerben. Az implementáláson túl természetesen a teljesítményét is tervezem összehasonlítani a meglévő megoldások teljesítményével.

Irodalomjegyzék

[1] 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* (konferenciaanyag). 2018, ACM, 1421–1432. p. URL http://doi.acm.org/10.1145/3183713.3190654.

[2] 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* (konferenciaanyag). 2015, ACM, 1371–1382. p. URL https://doi.org/10.1145/2723372.2742796.

[3] Albert-László Barabási – Eric Bonabeau: Information technology: Scale-free networks. *Scientific American*, 288. évf. (2003. május) 5. sz., 60–69. p. ISSN 0036-8733 (print), 1946-7087 (electronic). URL http://www.nature.com/scientificamerican/journal/v288/n5/pdf/scientificamerican0503-60.pdf.

[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 – Per-Åke Larson – Frank Wm. Tompa: Efficiently updating materialized views. In *SIGMOD* (konferenciaanyag). 1986, ACM Press, 61–71. p. URL https://doi.org/10.1145/16894.16861.

[6] E. F. Codd: A relational model of data for large shared data banks. *Commun. ACM*, 13. évf. (1970) 6. sz., 377–387. p.
URL http://doi.acm.org/10.1145/362384.362685.

[7] Latha S. Colby – Timothy Griffin – Leonid Libkin – Inderpal Singh Mumick – Howard Trickey: Algorithms for deferred view maintenance. In *SIGMOD* (konferenciaanyag). 1996, ACM Press, 469–480. p. URL https://doi.org/10.1145/233269.233364.

[8] Orri Erling – Alex Averbuch – Josep-Lluis Larriba-Pey – Hassan Chafi – Andrey Gubichev – Arnau Prat-Pérez – Minh-Duc Pham – Peter A. Boncz: The LDBC Social

Network Benchmark: Interactive workload. In *SIGMOD* (konferenciaanyag). 2015, 619–630. p. URL http://doi.acm.org/10.1145/2723372.2742786.

[9] Orri Erling – Ivan Mikhailov: Virtuoso: RDF support in a native RDBMS. In Roberto De Virgilio – Fausto Giunchiglia – Letizia Tanca (szerk.): *Semantic Web Information Management - A Model-Based Perspective*. 2009, Springer, 501–519. p. URL https://doi.org/10.1007/978-3-642-04329-1_21.

[10] Cristian Estan – Jeffrey F. Naughton: End-biased samples for join cardinality estimation. In *ICDE* (konferenciaanyag). 2006, IEEE Computer Society, 20. p. URL https://doi.org/10.1109/ICDE.2006.61.

[11] Philip J. Fleming – John J. Wallace: How not to lie with statistics: The correct way to summarize benchmark results. *Commun. ACM*, 29. évf. (1986) 3. sz., 218–221. p. URL http://doi.acm.org/10.1145/5666.5673.

[12] Nadime Francis – Alastair Green – Paolo Guagliardo – Leonid Libkin – Tobias Lindaaker – Victor Marsault – Stefan Plantikow – Mats Rydberg – Petra Selmer – Andrés Taylor: Cypher: An evolving query language for property graphs. In *SIGMOD* (konferenciaanyag). 2018, ACM, 1433–1445. p.
URL http://doi.acm.org/10.1145/3183713.3190657.

[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] Timothy Griffin – Bharat Kumar: Algebraic change propagation for semijoin and outerjoin queries. *SIGMOD Record*, 27. évf. (1998) 3. sz., 22–27. p.
URL https://doi.org/10.1145/290593.290597.

[15] Paolo Guagliardo – Leonid Libkin: A formal semantics of SQL queries, its validation, and applications. *PVLDB*, 11. évf. (2017) 1. sz., 27–39. p.
URL http://www.vldb.org/pvldb/vol11/p27-guagliardo.pdf.

[16] Ashish Gupta – Inderpal Singh Mumick – V. S. Subrahmanian: Maintaining views incrementally. In *SIGMOD* (konferenciaanyag). 1993, ACM Press, 157–166. p. URL https://doi.org/10.1145/170035.170066.

[17] Peter J. Haas – Jeffrey F. Naughton – S. Seshadri – Arun N. Swami: Selectivity and cost estimation for joins based on random sampling. *J. Comput. Syst. Sci.*, 52. évf. (1996) 3. sz., 550–569. p. URL https://doi.org/10.1006/jcss.1996.0041.

[18] S. Harris – N. Lamb – N. Shadbolt: 4store: The design and implementation of a clustered RDF store. In *SSWS (International Workshop on Scalable Semantic Web Knowledge Base Systems)*, CEUR Workshop Proceedings konferenciasorozat, 517. köt. 2009, CEUR-WS.org.
URL http://ceur-ws.org/Vol-517/ssws09-paper7.pdf.

[19] Georg Hinkel: NMF: a multi-platform modeling framework. In *ICMT* (konferenciaanyag). 2018, 184–194. p.
URL https://doi.org/10.1007/978-3-319-93317-7_10.

[20] Christoph Koch – Yanif Ahmad – Oliver Kennedy – Milos Nikolic – Andres Nötzli – Daniel Lupei – Amir Shaikhha: DBToaster: higher-order delta processing for dynamic, frequently fresh views. *VLDB J.*, 23. évf. (2014) 2. sz., 253–278. p. URL https://doi.org/10.1007/s00778-013-0348-4.

[21] Venelin Kotsev – Nikos Minadakis – Vassilis Papakonstantinou – Orri Erling – Irini Fundulaki – Atanas Kiryakov: Benchmarking RDF query engines: The LDBC Semantic Publishing Benchmark. In *BLINK at ISWC* (konferenciaanyag). 2016. URL http://ceur-ws.org/Vol-1700/paper-01.pdf.

[22] 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.

[23] Viktor Leis – Bernhard Radke – Andrey Gubichev – Atanas Mirchev – Peter A. Boncz – Alfons Kemper – Thomas Neumann: Query optimization through the looking glass, and what we found running the Join Order Benchmark. *VLDB J.*, 27. évf. (2018) 5. sz., 643–668. p. URL https://doi.org/10.1007/s00778-017-0480-7.

[24] József Marton – Gábor Szárnyas – Márton Búr: Model-driven engineering of an openCypher engine: Using graph queries to compile graph queries. In *SDL Forum* (konferenciaanyag). 2017, 80–98. p.
URL https://doi.org/10.1007/978-3-319-68015-6_6.

[25] 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.

[26] Derek Gordon Murray – Frank McSherry – Rebecca Isaacs – Michael Isard – Paul Barham – Martín Abadi: Naiad: a timely dataflow system. In *SIGOPS* (konferenciaanyag). 2013, ACM, 439–455. p.
URL https://doi.org/10.1145/2517349.2522738.

[27] Derek Gordon Murray – Frank McSherry – Michael Isard – Rebecca Isaacs – Paul Barham – Martín Abadi: Incremental, iterative data processing with timely dataflow. *Commun. ACM*, 59. évf. (2016) 10. sz., 75–83. p.
URL https://doi.org/10.1145/2983551.

[28] Hung Q. Ngo – Ely Porat – Christopher Ré – Atri Rudra: Worst-case optimal join algorithms. *J. ACM*, 65. évf. (2018) 3. sz., 16:1–16:40. p.
URL https://doi.org/10.1145/3180143.

[29] Hung Q. Ngo – Christopher Ré – Atri Rudra: Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Record*, 42. évf. (2013) 4. sz., 5–16. p. URL https://doi.org/10.1145/2590989.2590991.

[30] Anil Pacaci – Alice Zhou – Jimmy Lin – M. Tamer Özsu: Do we need specialized graph databases? Benchmarking real-time social networking applications. In *GRADES at SIGMOD* (konferenciaanyag). 2017, 12:1–12:7. p.
URL http://doi.acm.org/10.1145/3078447.3078459.

[31] 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. URL https://doi.org/10.1145/1567274.1567278.

[32] Marko A. Rodriguez: The Gremlin graph traversal machine and language (invited talk). In *DBPL* (konferenciaanyag). 2015, 1–10. p.
URL http://doi.acm.org/10.1145/2815072.2815073.

[33] Michael Rudolf – Marcus Paradies – Christof Bornhövd – Wolfgang Lehner: The graph story of the SAP HANA database. In *BTW* (konferenciaanyag). 2013, 403–420. p. URL http://www.btw-2013.de/proceedings/The%20Graph%20Story%20of%20the%20SAP%20HANA%20Database.pdf.

[34] Sherif Sakr – Sameh Elnikety – Yuxiong He: G-SPARQL: a hybrid engine for querying large attributed graphs. In *CIKM* (konferenciaanyag). 2012, ACM, 335–344. p. URL https://doi.org/10.1145/2396761.2396806.

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

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

[37] 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 at SIGMOD* (konferenciaanyag). 2017, 4:1–4:6. p. URL http://doi.acm.org/10.1145/3078447.3078451.

[38] 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* (konferenciaanyag). 2014, 653–669. p. URL http://dx.doi.org/10.1007/978-3-319-11653-2_40.

[39] Gábor Szárnyas – Benedek Izsó – István Ráth – Dániel Varró: The Train Benchmark: cross-technology performance evaluation of continuous model queries. *Softw. Syst. Model.*, 17. évf. (2018) 4. sz., 1365–1393. p.
URL https://doi.org/10.1007/s10270-016-0571-8.

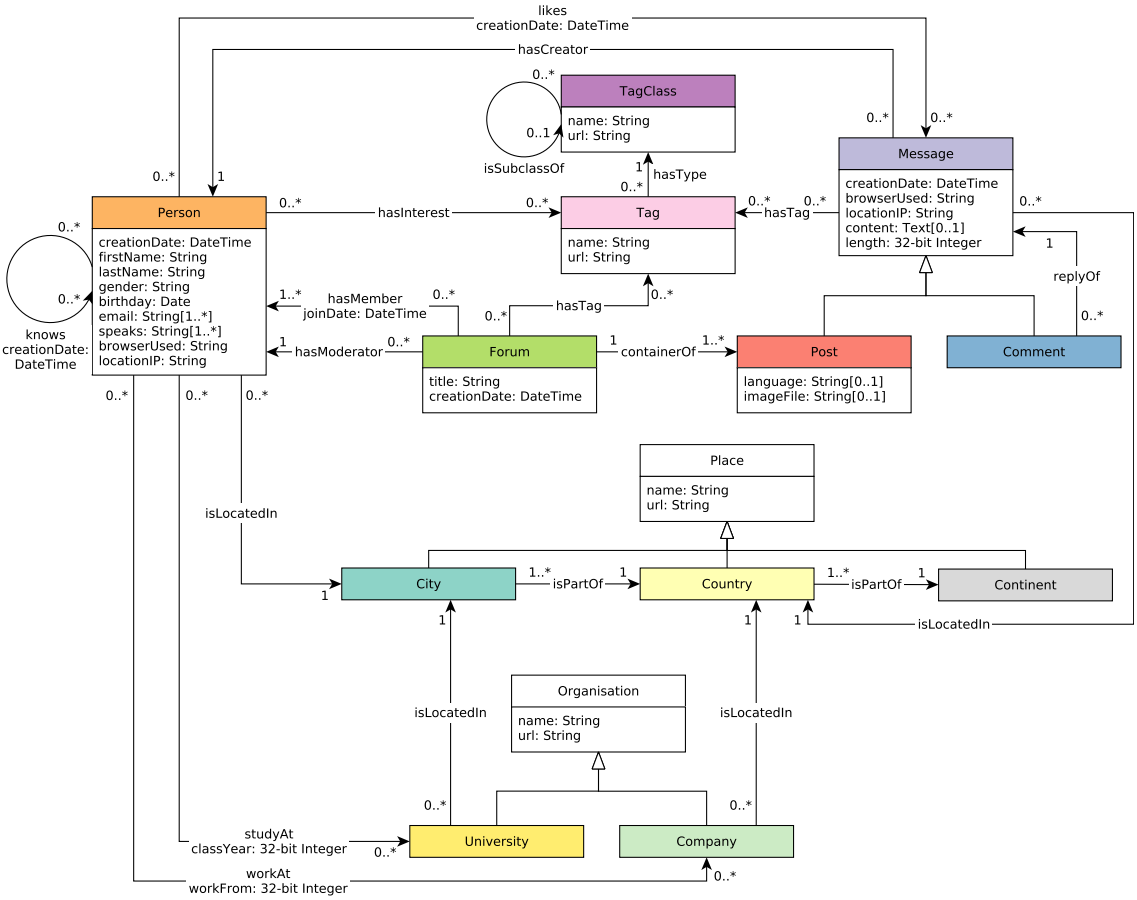
[40] Gábor Szárnyas – Arnau Prat-Pérez – Alex Averbuch – József Marton – Marcus Paradies – Moritz Kaufmann – Orri Erling – Peter A. Boncz – Vlad Haprian – János Benjamin Antal: An early look at the LDBC Social Network Benchmark's Business Intelligence workload. In *GRADES-NDA at SIGMOD* (konferenciaanyag). 2018, ACM, 9:1–9:11. p.
URL http://doi.acm.org/10.1145/3210259.3210268.

[41] 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. *Softw. Syst. Model.*, 15. évf. (2016) 3. sz., 609–629. p. URL http://dx.doi.org/10.1007/s10270-016-0530-4.

[42] Todd L. Veldhuizen: Triejoin: A simple, worst-case optimal join algorithm. In *ICDT* (konferenciaanyag). 2014, 96–106. p.
URL https://doi.org/10.5441/002/icdt.2014.13.

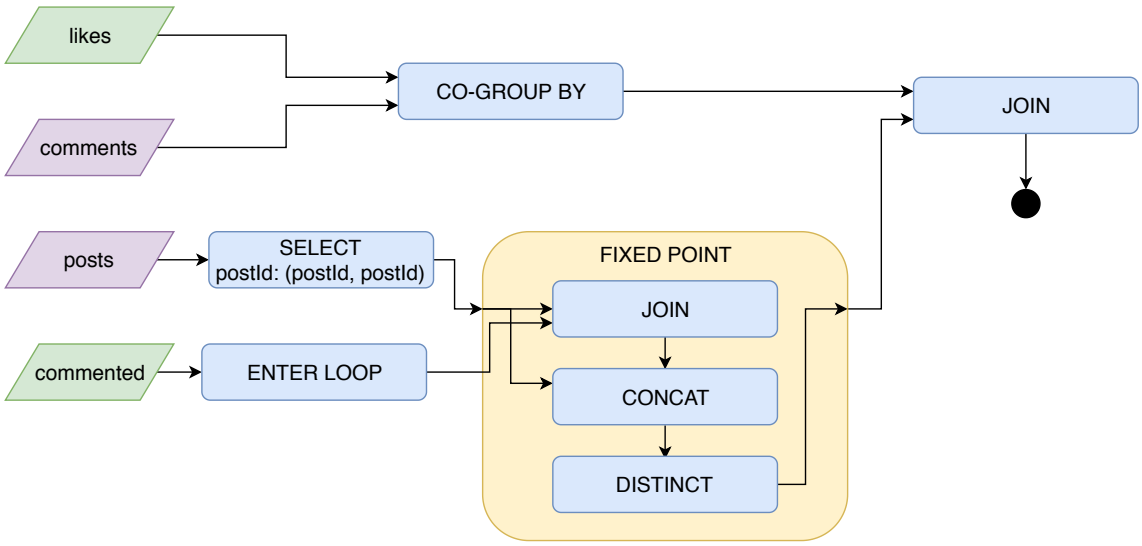
Függelék

F.1. LDBC SNB adatséma



F.1.1. ábra. Az LDBC Social Network Benchmark gráf sémája

F.2. Q1 lekérdezés differenciális adatfolyama



F.2.1. ábra. A TTC Social Network feladat Q1 lekérdezésének differenciális adatfolyama