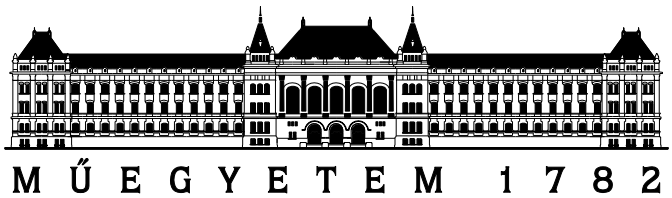
BUDAPESTI MŰSZAKI ÉS GAZDASÁGTUDOMÁNYI EGYETEM



VILLAMOSMÉRNÖKI ÉS INFORMATIKAI KAR

Biztonságos keresés szemantikus webalkalmazás tárolókban

TDK dolgozat

Készítette:

Bodó Gábor (FSVGK1)

Konzulens:

Simon Balázs

IRÁNYÍTÁSTECHNIKA ÉS INFORMATIKA TANSZÉK

2011.

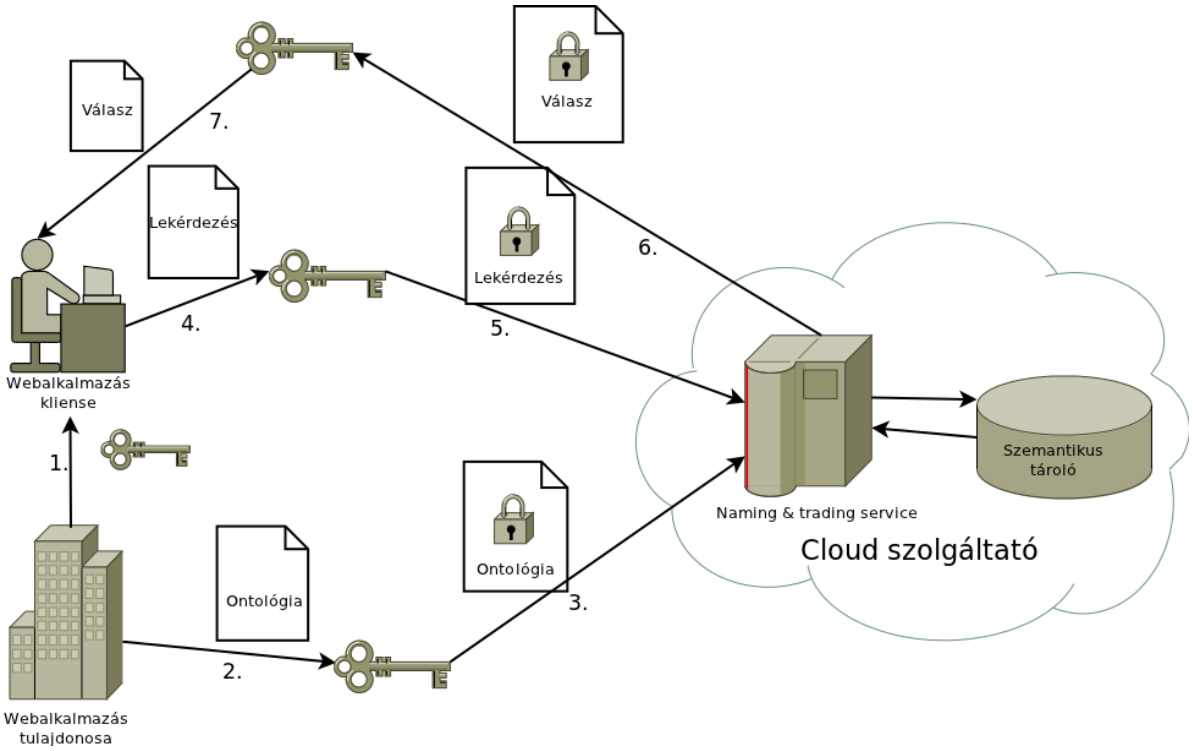
TARTALOMJEGYZÉK

1. BEVEZETÉS

Számítási felhőnek (cloud) olyan, jellemzően külső szolgáltató által nyújtott informatikai megoldást vagy környezetet nevezünk, amelynek építőelemei, azaz a hardverek, hálózati eszközök, szoftverplatformok és az őket kezelő folyamatok lehetővé teszik szolgáltatásaink Interneten vagy helyi hálózaton történő elérését [1]. A számítási felhő tehát absztrakciós szintet képez a hardver és/vagy valamilyen szoftverplatform felett, megkönnyítve az infrastruktúra telepítésével és felügyeletével kapcsolatos feladatokat. Felhasználási terület szerint megkülönböztetünk **SaaS** (Software-as-a-Service), **PaaS** (Platform-as-a-Service), illetve **IaaS** (Infrastructure-as-a-Service) felhőket. SaaS felhőkben a kliens valamilyen cloud infrastruktúrán futó webalkalmazást használ – ilyen pl. a Gmail -, PaaS rendszerekben a szolgáltató által biztosított eszközökkel, speciális platformra fejlesztett webalkalmazást futtat, míg IaaS szolgáltatásról akkor beszélünk, ha a kliens virtualizált hardver- és hálózati erőforrásokhoz férhet hozzá. Ismert PaaS szolgáltatás például a Google App Engine, az Amazon Elastic Compute Cloud (EC2) pedig az IaaS architektúra talán legtisztább mintapéldája. A rendszer elhelyezkedése szerint megkülönböztetünk **nyilvános**- (azaz egy külön vállalat által működtetett, valamilyen díjazás ellenében bárki által igénybe vehető), illetve szervezeten belüli, **privát** felhőket (utóbbit a szolgáltatás tulajdonosa birtokolja és kezeli). A nyilvános és privát komponenseket egyaránt tartalmazó infrastruktúrákat **hibrid** felhőnek nevezzük [1].

A felhőalapú szolgáltatások megjelenése eddig még soha nem látott lehetőséget adott a vállalati informatikai infrastruktúrák költségének csökkentésére, ugyanakkor olyan biztonsági problémákat is felvetett, amellyel a hagyományos architektúrák szakértői korábban nem találkoztak. Mindezek miatt a cloud szolgáltatások elterjedését jelentősen gátolja a félelem az adatok elvesztésétől, illetve illetéktelen kézbe kerülésétől. Szükséges tehát olyan algoritmusok és protokollok bevezetése, amelyek képesek a felhő klienseinek adatait megvédeni úgy, hogy azokat továbbra is elfogadható teljesítménnyel lehessen olvasni és módosítani a biztonsági szabályok megsértése nélkül. Dolgozatomban egy olyan megoldást mutatok be, amely alkalmas PaaS felhőkben telepített, szemantikus webalkalmazás tárolókban található adatok rejtjelezésére és az így biztosított információkon történő lekérdezések futtatására. Ezen megoldás lehetővé teszi, hogy egy vállalat vagy állami intézmény az általa futtatott webalkalmazásokhoz egységes, magas rendelkezésre állású, alacsony költségű keresőszolgáltatást (naming-, illetve trading service-t) béreljen anélkül, hogy adatszi-

várgás veszélye fenyegetné. Az egyes webalkalmazások továbbra is saját infrastruktúrán futnak, de az erőforrás-igényes, költséges keresőrendszer hosztolása kiszervezhető egy megfelelően auditált cloud szolgáltatóhoz.



1. ábra: A rendszer működési elve

A fenti ábrán láthatjuk a rendszer működésének alapelveit: A webalkalmazások publikálói és kliensei megosztanak egymással egy kulcsot, amit a cloud szolgáltató nem ismerhet meg. Ezt követően a webalkalmazás tulajdonosa szemantikus ontológiákkal leírja szolgáltatása(i) működését, ezt a leírást rejtjelezi, majd elküldi a cloud szolgáltatónak. A kliens, ha keresni szeretne a rendelkezésre álló webalkalmazások között, összeállít egy lekérdezést, amelyben a szolgáltatások szemantikájára kérdez rá (pl. „Milyen, legalább háromkilences rendelkezésre állású szolgáltatással tudok push üzeneteket fogadni?”). Ezt a lekérdezést a már említett közös kulccsal transzformálja és elküldi a cloud szolgáltató keresőrendszerének, ami a rejtjelezett ontológiákon kiértékeli anélkül, hogy az adatok valódi tartalmát megismerné. A visszakapott rekordokat a kliens dekódolja és feldolgozza, azaz megkapja a referenciákat az általa használni kívánt webalkalmazásokra.

A webszolgáltatások tulajdonosai tehát képessé válnak rejtjelezett metaadatokkal ellátni szolgáltatásaik interfészleíróit, amelyek között a kliensek a megfelelő jogosultságok birtokában képesek hatékonyan, alacsony számítási többletköltséggel keresni, majd a kapott információkat dekódolni. A rendszer működésével szemben fontos elvárás, hogy a webalkalmazás keresőszolgáltatásának tulajdonosa (általában cloud üzemeltető) ne ismerje meg az általa tárolt paramétereket nyílt szöveg formában.

Dolgozatomban először elemzem a cloud rendszerek biztonsági kérdéseit, majd bemutatok néhány korszerű megoldást rejtjelezett adatokon végzett keresések és transzformációk futtatására, részletezve az általam implementált rendszerben használt eljárásokat [2][3]. Ezt követően röviden ismertetem a **szemantikus web** alapfogalmait, valamint a szemantikus információkkal annotált webalkalmazások jellegzetességeit és előnyeiket a hagyományos SOAP architektúrával szemben. A kiindulópontként használt infrastruktúra bemutatása után részletezem a rendszeren végrehajtott változtatásokat, az implementáció és integráció kérdéseit, beleértve a webalkalmazások kereséséhez használt lekérdező nyelven elvégzett módosításokat. A dolgozat végén ismertetem a rendszer tesztelése és profilozása során felmerült kérdéseket és tanulságokat.

2. A PROBLÉMA ELEMZÉSE

2.1. Cloud rendszerek biztonsági kérdései

2.1.1. Algoritmusok és protokollok cloud rendszerek biztosítására

Külső szervezet, vagy más nem megbízható partner által üzemeltett informatikai rendszerek adatrétegének biztosítására számos algoritmikus módszer létezik, többé-kevésbé eltérő módszerekkel és szűk értelemben vett felhasználási területekkel. Az általam használt megoldások részletes bemutatása előtt áttekintem a témával kapcsolatos fő kutatási irányvonalakat, előnyeikkel és hátrányaikkal együtt.

A talán legismertebb megoldások az ún. **predikátum** rejtjelezések [4]. Egy ilyen architektúrában az adatokat rejtjelezve tároljuk, a mesterkulcs tulajdonosa pedig képes a rendszer használatára - a mi esetünkben: lekérdezések futtatására - jogosult felhasználóknak biztonsági tokeneket kiosztani, amihez predikátumokat rendelünk. Egy predikátum valamilyen feltétel kiértékelése a rejtjelezett adat felett; adatbázis lekérdezéseknél ilyen pl. az egyenlőség numerikus attribútumra. Formálisan: ha x a rejtjelezett adat, a Tk_f tokennel kiértékelhető rajta az $f(x)$ predikátum, és a kiértékelés eredménye boolean érték lesz. A megoldás előnye, hogy számos biztonságos algoritmust implementáltak a témában [4], hátránya, hogy a mesterkulcs tulajdonosának - aki általában az adatok gazdája - aktív részvétele szükséges hozzá.

Bizalmas adatok lekérésére szolgálnak a különféle **PIR** (Private Information Retrieval) protokollok is [5]. A módszer lényege, hogy a felhasználó úgy tudjon adataihoz hozzájutni, hogy a szerver ne dönthesse el, melyik rekordo(ka)t kérte le. Mivel az információk nyílt szövegként vannak tárolva, ez a megoldás csak olyan esetekben alkalmazható, ha nem adataink elrejtése, hanem klienseink privát szférájának védelme a cél. Egyes PIR protokollok egy szerveren működnek, de számos megköveteli az egymással nem kommunikáló adatbázis-másolatok jelenlétét [5], ami szintén csökkenti az eljárás gyakorlati alkalmazhatóságát.

Elviekben a kriptográfia "Szent Gráljának" tartott **homomorfikus rejtjelezés** [6] is alkalmas titkosított adatokon való keresésre. Homomorfikus rejtjelezést használ egy program, ha a bemenete és a kimenete is titkosított, és a bemeneten *tetszőleges* operáció helyesen elvégezhető a nyílt szöveg felfedése nélkül. Az eljárás előnye, hogy bármilyen, számítógép által értelmezhető problémát képes megoldani rejtjelezett adatokon. Hátránya, hogy a jelenlegi hardverinfrastruktúra teljesítménye kevés ahhoz, hogy bármely ismert

homomorfikus algoritmust hatékonyan futtasson. A [6] tanulmányban ismertetett implementáció például 17 MB és 2.25 GB közötti kulcsméreteket használt, a kulcsgenerálás pedig a legnagyobb méret esetén több mint két óráig tartott egy nagy teljesítményűnek mondható IBM System x3500 számítógépen.

Az **adatstruktúra-specifikus** protokollok az adattároló jellegzetességeit kihasználva teszik lehetővé a rejtjelezett információkon végzett lekérdezések futtatását. Számos ilyen, gyakorlatban is alkalmazható algoritmus ismert, amiknek legtöbbje relációs adatbázist használ. Noha ezek a megoldások az adatbázisrendszerektől elvárt teljesítményparaméterek megtartása mellett nem nyújtanak - nem is nyújthatnak [7] - formálisan bizonyítható módon teljes biztonságot, megfelelően megválasztott paraméterekkel kockázatcsökkentő hatásuk van, és magas valószínűséggel megvédik adatainkat. A 2.1.2. és 2.1.3. részben két ilyen, rendszeremben is alkalmazott megoldást fogok bemutatni.

2.1.2. Numerikus adatok biztonságos lekérdezése

Relációs (SQL) adatbázisokban numerikus, vagy numerikus értékre leképezett egyéb adatok biztonságos, cloud-kompatibilis tárolására és keresésére alkalmas a [2] tanulmányban bemutatott eljárás. Ezen algoritmus előnye a teljes rejtjelezett adatbázis elküldésével szemben, hogy kisebb a hálózati overhead, ugyanis először az adatbázis szerver elvégzi a rejtjelezett adatokon a lekérdezést, majd a lehetséges illeszkedő értékek közül a kliens kiszűri a hamis pozitívokat. Az alábbiakban bemutatom azt a megoldást, ami lehetővé teszi a rejtjelezett adatokon végzett lekérdezést.

Cseréljünk le minden $R(X_1, X_2, \ldots, X_k, \ldots, X_n)$ relációt egy rejtjelezett $R^S(etuple, X_1^S, X_2^S, \ldots, X_k^S, \ldots, X_n^S)$ relációra, ahol $etuple$ az aktuális rekord összes attribútuma $(X_1, X_2, \ldots, X_k, \ldots, X_n)$ rejtjelezve tetszőleges szimmetrikus kódolóval, X_i^S pedig X_i **indexe** $\forall\ 1 \leq i \leq n$ értékre. Az $etuple$ dekódolása után jut hozzá a kliens a kívánt információkhoz. Ahhoz pedig, hogy az index mező feladatát értelmezni tudjuk, be kell vezetnünk a **particionáló függvény** fogalmát.

Legyen R reláció X_i attribútumának particionálása, ahol a $partition(R.X_i) = \{p_1, p_2, \ldots, p_k\}$ felosztásra igaz, hogy $D_i = \bigcup_{j=1}^{k} p_j$ és $p_j \cap p_l = \emptyset\ \forall\ j \neq l$ értékre, ahol D_i az X_i attribútum értékkészlete. Más szavakkal: osszuk fel az attribútum értékkészletét k darab diszjunkt halmazra (ún. **ládákra** vagy **kosarakra**). A felosztás ismerős lehet azoknak, akik jártásak az adatbányászatban használatos kosarazásban. Valóban, bármilyen ott alkalmazott hisztogram függvény (pl. MaxDiff) [8] megfelel a particionálás céljára, de fon-

tos ügyelnünk arra, hogy a felosztás milyensége befolyásolni fogja a majdani lekérdezések során kapott hamis pozitívok és valós értékek arányát.

A particionáló függvényen kívül szükségünk van egy olyan $ident_{R.X_i}(p_j)$ leképezésre, amely minden partícióhoz hozzárendel egy egyedi azonosítót. Az azonosító előállítása történhet például egy ütközésmentes hash eljárással. Ezt az azonosítót fogja használni a $Map_{R.X_i}(v)$ **leképező függvény**, amely tetszőleges $v \in D_i$ értékre teljesíti a $Map_{R.X_i}(v) = ident_{R.X_i}(p_j)$ összefüggést, amennyiben $v \in p_j$. A függvény lehet sorrendet megtartó vagy véletlenszerű attól függően, hogy nagyobb értékhez nagyobb azonosítót rendel-e, vagy sem. Előbbi könnyebben kezelhető a lekérdezések során, utóbbi biztonságosabb. A leképező függvény adja a már említett index mező értékét.

A lekérdezések során a feltételeket a kliens transzformálja olyan formába, hogy a szerver képes legyen őket úgy kiértékelni, hogy a bizalmasság sérülése nélkül visszaadhassa a kívánt értékeket minél kevesebb hamis pozitívval. A transzformációs szabályok a következők (véletlenszerű leképező függvényt feltételezve; sorrendet megtartó esetben is működnek, de ott lehetőség van egyszerűbb megoldásokra is):

a, $X_k = value$ (numerikus egyenlőség) → $X_k^S = Map_{R.X_k}(value)$

b, $X_k < value \rightarrow X_k^S \in Map\_less_{R.X_k}(value)$, ahol Map_less visszaad minden olyan partíciót, amire az egyenlőtlenség igaz lehet.

c, $X_k > value \rightarrow X_k^S \in Map\_more_{R.X_k}(value)$, ahol Map_more visszaad minden olyan partíciót, amire az egyenlőtlenség igaz lehet.

d, $X_i = X_j$, $j \neq i$ (pl. join művelet során kerülhet elő) → Vegyük fel a $\varphi = (p_k \in partition(X_i) \wedge p_l \in partition(X_j) \wedge p_l \cap p_k \neq \emptyset)$ feltételt, majd legyen $\bigvee_{\varphi}(X_i = ident_{X_i}(p_k) \wedge X_j = ident_{X_j}(p_l))$, azaz válasszuk ki páronként X_i és X_j minden olyan partícióját, melyek értékkészlete nem diszjunkt, majd írjuk fel őket a logikai OR operátorral.

e, $X_i < X_j$, $j \neq i$ (ezzel is join művelet során találkozhatunk) → Vegyük fel a $\varphi = (p_k \in partition(X_i) \wedge p_l \in partition(X_j) \wedge \mathrm{Max}(p_l) \geq Min(p_k))$ feltételt, majd legyen $\bigvee_{\varphi}(X_i = ident_{R.X_i}(p_k) \wedge X_j = ident_{R.X_j}(p_l))$. Látható, hogy a megoldás az előzőhöz hasonló, csupán a feltételt kell megváltoztatni az egyenlőtlenségnek megfelelően.

A különféle feltételek logikai operátorokkal össze is köthetjük; ehhez a következő szabályokat kell betartanunk:

f, $Condition_1 \wedge \text{Condition}_2$ (logikai AND) → $Map(Condition_1) \wedge Map(Condition_2)$

g, $Condition_1 \vee \text{Condition}_2$ (logikai OR) → $Map(Condition_1) \vee Map(Condition_2)$

Az itt bemutatott módszer megfelelő szimmetrikus kódoló, erős hash algoritmus és véletlenszerű leképező függvény együttes alkalmazása esetén képes numerikus adataink védelmére és lehetővé teszi a rejtjelezett rekordokon lekérdezések futtatását transzparens módon.

2.1.3. Szövegek biztonságos lekérdezése

A 2.1.2 részben bemutatott algoritmus hiányossága, hogy csak korlátozott mértékben képes szöveges adatokat kezelni. Lehetséges ugyan karakterfüzérekhez numerikus értéket rendelni indexként, és valamilyen szabály szerint (pl. ABC sorrend) particionálni az értéktartományt, de az így rejtjelezett adatokon nem végezhetünk lekérdezéseket szövegrészletekre (SQL LIKE kulcsszó). Ezt a hiányosságot igyekeztem kiküszöbölni a [3] tanulmányban bemutatott eljárás implementációjával és alkalmazásával.

A fent említett algoritmus egy új rekord beszúrása során először egy tetszőleges szimmetrikus kódolóval rejtjelezi a szöveges adatokat (jelölése a továbbiakban: $E(s)$), majd minden ilyen attribútumhoz rendel egy indexmezőt. Az index kiszámításához egy ún. Pairs Coding Function (PC) leképezést használ, amelynek működési mechanizmusa a következő: Legyen PC egy $s_1 \rightarrow s_2$ leképezés, ahol s_1 egy $c_1, c_2, \dots, c_n$ karakterfüzér (az eredeti, nyílt szövegű adat), s_2 pedig egy $b_1, b_2, \dots, b_{m-1}$ bitsorozat (az index mező), ahol $n < m$. Ismert továbbá a H hash algoritmus, ami s_1 minden c_i, c_{i+1} karakterpárját leképezi egy $0 \leq k \leq m-1$ egészre. Az index bitjeinek értéke a következő szabály szerint alakul:

$$b_i = 1, \text{ha } H(c_j, c_{j+1}) = i \text{ valamely } 1 \leq j \leq n-1 \text{ egészre.}$$

$$\text{Egyébként } b_i = 0$$

A PC leképezés tehát minden $R(X_1, X_2, \dots, X_k, \dots, X_n)$ relációs sémát lecserél egy $R^E(X_1, X_2, \dots, X_k^E, \dots, X_n, X_k^S)$ rejtjelezett sémára, ahol minden X_k bizalmas attribútum helyére az $X_k^E = E(X_k)$ szimmetrikus kódolóval rejtjelezett érték kerül, valamint bizalmas mezőnként felveszünk egy $X_k^S = PC(X_k)$ indexet is. A lekérdezés során első körben az adatbázis szerver megpróbál az indexre illeszteni, a behelyettesíthető értékek visszaadása után a kliens oldalon történik egy újabb lekérdezés a dekódolt adatokon, a hamis pozitív értékek kizárására. A rendszer működési elvét arra alapozzuk, hogy megfelelően megválasztott m paraméterrel és H algoritmussal a leképezések során elég hash ütközés történik

ahhoz, hogy az adatbázis üzemeltetője kis eséllyel tudjon következtetni az adatmezők tartalmára, de nem történik olyan sok, hogy az első lekérdezés során nagy mennyiségű hibás adat jusson el a felhasználóhoz. Az egyes lekérdezések kliens oldali transzformációja az alábbi szabályok szerint zajlik (a szerver már a transzformált lekérdezés-adatokkal dolgozik):

a, $X_k = "value"$ feltétel (pontos szövegre való keresés) → $X_k^S = PC("value")$

b, $X_k\ LIKE\ c_1, c_2, \ldots, c_{n-1}, c_n$ (szövegrészletre való keresés) → $X_k^S\ \forall\ X_{ki}^S$ indexű bitjére és $\forall\ c_j, c_{j+1}$ karakterpárra $(1 \leq j \leq n-1)$ igaz, hogy ha $H(c_j, c_{j+1}) = i$, akkor $X_{ki}^S = 1$.

c, $X_k\ NOT\ LIKE\ c_1, c_2, \ldots, c_{n-1}, c_n$ (szövegrészletre való keresés különbözőség szerint) → $X_k^S\ \forall\ X_{ki}^S$ indexű bitjére és $\forall\ c_j, c_{j+1}$ karakterpárra $(1 \leq j \leq n-1)$ igaz, hogy ha $H(c_j, c_{j+1}) = i$, akkor $X_{ki}^S = 0$.

A rendszer természetesen képes a különféle feltételek logikai operátorokkal való összekapcsolására is; ebben az esetben a transzformációt az egyes feltételekre külön-külön elvégezhetjük az operátorok átrendezése nélkül. Például az $X_k = "xx"\ AND\ X_m = "yy"$ lekérdezés az $X_k^S = PC("xx")\ AND\ X_m^S = PC("yy")$ kifejezésre transzformálódik. Ez az összefüggés a többi logikai operátorra (NOT, OR) is igaz. A karakterpár-kódoló algoritmus működési mechanizmusaiból adódóan az is triviálisan látható, hogy a tartalmazó lekérdezések nem működnek két karakternél rövidebb LIKE feltételekre. Mivel a gyakorlatban az egy karakterre való keresések eredményei kevés információt hordoznak magukban, és az ilyen lekérdezések igen ritkák, ezt a kikötést nem tartom jelentős problémának.

A [3] tanulmány szerzői megemlítik, hogy a fenti algoritmus statisztikai módszerekkel támadható. Minél nagyobb ugyanis az m index-bithossz paraméter értéke, annál kisebb az ütközések valószínűsége, ha pedig a támadó ismeri az adott nyelvben előforduló karakterpárok gyakoriságát, kikövetkeztetheti a bizalmas mezők tartalmát. A rendszer nyílt szöveg alapú támadásokkal is sebezhető, mert ha a támadó ismeri valamely X_k paraméter és X_k^S index ismeretében kikövetkeztetheti, hogy egy másik X_k^S indexű X_l^E attribútum értéke is X_k. Ennek az esélye kisebb bithossznál, azaz több ütközés esetén természetesen alacsonyabb. A bithossz és az első körben visszakapott hamis pozitív rekordok száma viszont fordított arányban áll egymással, hiszen minél több ütközés történik, annál több hamis érték kerül át a felhasználóhoz, különösen a rövid szövegrészletek egyezésére rákérdező kereséseknél. Az ütközések számának becslésére és a bithossz paraméter kiválasztására használjuk az alábbi összefüggést:

$$\frac{x^2}{m} = avg(COLL),$$

ahol $COLL$ az ütközések száma, x a karakterkészlet elemszáma, m pedig a bithossz.

Mivel a [3] tanulmány nem tesz kikötéseket a hash függvény pontos implementációjára, a biztonsági problémák kiküszöbölésére az eredetinél erősebb feltételeket szabtam az indexelő algoritmus tulajdonságaira, az eredeti megoldást kibővítve. A 3.2. fejezetben részletesen is kifejtem, hogy az $E(s)$ szimmetrikus kódoláson felül milyen rejtjelező operációkat kell elvégeznünk az indexelés előtt, hogy a kiszervezett adatbázis üzemeltetői nehezebben tudják kikövetkeztetni a tárolt adatok értékeit.

2.2. A szemantikus web alapjai

2.2.1. Röviden a szemantikus webről

A szemantikus technológiák fő célja egyszerű, absztrakt modellt adni tudásreprezentációs megoldások számára, lehetővé téve az alkalmazás és az általa használt adatok különválasztását [9]. Jelentéstartalmak (ontológiák) formális leírása segítségével gép által feldolgozhatóvá tesznek olyan - elsősorban webes - dokumentumokat, amelyet emberek számára készítettek. A legtöbb szemantikus eljárás azon az elven alapul, hogy számos egyszerű kijelentés nyelvtanilag leírható az alany – állítmány - tárgy hármassal. Ha ezt az alapelvet felhasználjuk adatformátumok megadása és információink tárolása során, olyan sémákat kapunk, amelyekben az adat implicite magában hordozza a strukturális metainformációkat, ezért jóval rugalmasabbak, kifejezőbbek, bővíthetőbbek és gyakran hatékonyabban elemezhetőek számítógéppel, mint a hagyományos (pl. relációs) adatmodellek, de az emberi kognitív tulajdonságokhoz is közel állnak annyira, hogy könnyen értelmezhessük őket.

Az előbbiekben említett **önleíró** adatok reprezentációja irányított gráfokkal történik, ahol az élekhez a hármasokból az állítmányokat (predikátum) rendeljük, a csúcsokhoz pedig az alanyokat és a tárgyakat úgy, hogy élek csak alanyból tárgyba mutassanak [9]. Fontos megjegyeznünk, hogy az állítmány szófaja nem feltétlenül ige, valamint a hármasok jobb híján tárgynak (object) nevezett eleme akár határozószó is lehet. A fent bemutatott gráfokra példaként lássunk egy olyan kitalált alkalmazást, ami irodalmi művekkel kapcsolatos földrajzi információkat képes elemezni. Lássuk a következő állításokat:

a, Az Iskola a határon-t Ottlik Géza írta.

b, Ottlik Géza Budapesten született.

c, Budapest Magyarországon található.

d, Az Iskola a határon Kőszegen játszódik.

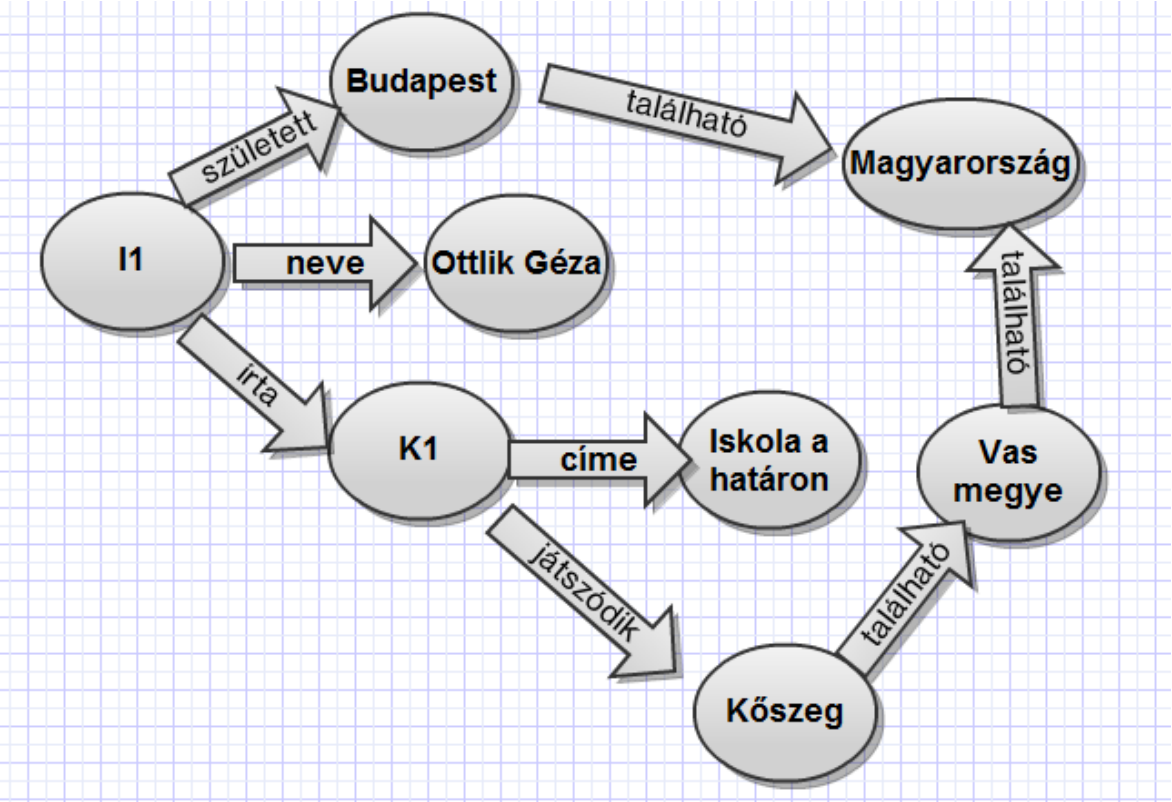
e, Kőszeg Vas megyében található.

f, Vas megye Magyarországon található.

Az a-f állításokat kibővíthetjük még kettővel, ha elképzelhető, hogy több Ottlik Géza nevű író és több Iskola a határon című regény létezik. Ebben az esetben legyen egy I1 nevű írónk, és egy K1 nevű könyvünk, amikre igaz, hogy

g, I1 neve Ottlik Géza.

h, K1 címe Iskola a határon.



2. ábra: Az a-h szabályrendszer gráfja

A fenti ábrán láthatjuk állításaink vizuális reprezentációját egy irányított gráffal. A szemantikus elemző alkalmazások is hasonló gráfok formájában tárolják és kezelik adataikat. Az ilyen szemantikus információk leírására az **RDF** szabvány használatos [9], amely URI-kat alkalmaz objektumaink egyedi azonosítójaként, és háromféleképpen képes sorosítani adatainkat (a negyedik, **RDFa** nevű adatformátum XHTML weboldalak RDF-annotációkkal való ellátására szolgál). A legegyszerűbb szerializálási módszer az **N-Triple**, amely voltaképp alany-állítmány-tárgy hármasok ponttal elválasztott sorozata. A b, g, és a h, szabályt például a következő módon írhatjuk le:

```
<http://arts.hu/authors/I1>
<http://xmlns.com/literature/name> "Ottlik Géza".
<http://arts.hu/authors/I1>
<http://xmlns.com/literature/place_of_birth>
<http://places.hu/cities/Budapest>.
<http://arts.hu/books/K1> <http://xmlns.com/literature/title>
"Iskola a határon".
```

Mivel az N-Triple meglehetősen nehezen olvasható, definiálták az **N3** szabványt, ami egyrészt bevezette a névtér-prefixumok használatát, másrészt az egy alanyhoz több tárgy felsorolásának lehetőségét pontosvesszővel elválasztva. A fenti három szabály a következőképpen írható át N3-ba:

```
@prefix author: <http://arts.hu/authors/>.
@prefix book: <http://arts.hu/books/>.
@prefix literature: <http://xmlns.com/literature/>.
@prefix city: <http://places.hu/cities/>.

author:I1 literature:name "Ottlik Géza";
          literature:place_of_birth city:Budapest.
book:K1 literature:title "Iskola a határon".
```

Mivel az N3 és az N-Triple nem gép által könnyen feldolgozható formátumok, már az eredeti W3C RDF ajánlás bevezette az RDF/XML szabványt, ami voltaképp utakat ír le a gráfban, csúcs-él-csúcs formában. Lássuk a már ismert három szabályt RDF/XML-ként!

```
<rdf:RDF  xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-
ns#'
          xmlns:author='http://arts.hu/authors/'
          xmlns:book='http://arts.hu/book/'
          xmlns:literature='http://xmlns.com/literature/'
          xmlns:city='http://places.hu/cities/'>
 <author:I1>
     <literature:name>Ottlik Géza</literature:name>
     <literature:place_of_birth>
          <city:Budapest/>
     </literature:place_of_birth>
 </author:I1>
 <book:K1>
     <literature:title>Iskola a határon</literature:title>
 </book:K1>
</rdf:RDF>
```

Az alapelvek ismeretében a továbbiakban be fogom mutatni, hogyan nyerhetünk ki aggregált információkat RDF adatstruktúrákból, és milyen használati esetei vannak a szemantikus megoldásoknak a különféle (SOAP és RESTful) webalklamazások kontextusában.

2.2.2. A SPARQL lekérdező nyelv

A SPARQL az SQL-hez hasonló lekérdező nyelv, amely RDF gráfokban történő keresést tesz lehetővé. Készült hozzá egy kiegészítés is, a SPARQL/Update; ez az adatok módosítását és új elemek felvételét, törlését is támogatja [10]. Az alábbiakban röviden ismertetem a SPARQL nyelv alapvető szintaxisát [9].

A lekérdezésekre a **SELECT** parancs szolgál, amelyhez opcionálisan névtérrövidítéseket is megadhatunk a **PREFIX** kulcsszóval, illetve – amennyiben a lekérdezésben relatív URI-kat kívánunk használni – egy bázis-URI értéket a **BASE** kulcsszóval. Ha minden értéket csak egyszer szeretnénk megkapni, a **SELECT DISTINCT** parancsot kell használnunk. Ezt követően a lekérdezendő paramétereket és a **WHERE** feltételhármasokat kell megadnunk (ponttal elválasztva). A WHERE feltételekben szereplő paraméterek nevei olyan alfanumerikus értékek, amelyek dollár- vagy kérdőjellel kezdődnek. A WHERE feltételben rákérdezhetünk olyan értékre is, aminek nem feltétlenül kell szerepelnie a gráfunkban: ezt az **OPTIONAL** kulcsszóval jelöljük. A visszakapott adatok közötti szűrésre (pl. reguláris kifejezés alapján) a **FILTER** szolgál. Ha több különböző gráf formátumra szeretnénk illeszteni a lekérdezést, az egyes feltételcsoportokat kapcsos zárójellel kell elválasztani. Utóbbi a feltételcsoportok közé helyezett **UNION** kulcsszóval az illesztéseket külön-külön végzi el, majd egyesíti a kapott értékeket. A SPARQL használatát szemlélteti az alábbi lekérdezés, amely egy képzeletbeli, egyetemi nyilvántartásra használt gráf adatbázisból visszaadja mindazon személyek Neptun kódját (és ha lehetséges, email címét), akik 1984 után születtek, és mind oktatóként, mind hallgatóként szerepelnek benne. Vegyük észre, hogy a UNION kulcsszó hiánya miatt a keresett értéknek mindkét mintára illeszthetőnek kell lennie, azaz a két gráf formátum ÉS kapcsolatban van.

```
PREFIX iit: <http://iit.bme.hu/ns/>
SELECT ?code
WHERE {
     {
     ?tcourse iit:course.course.taught_by ?teacher .
     ?teacher iit:type.person.neptun_code ?code .
     ?teacher iit:type.person.year_of_birth ?birth .
     OPTIONAL { ?teacher iit:type.person.email ?email . }
     FILTER (?birth > 1984)
     }
     {
     ?acourse iit:course.course.taken_by ?student .
     ?student iit:type.person.neptun_code ?code
     }
}
```

Amennyiben a lekérdezés lefutása után kötött értékekből valamilyen szabály szerint hármasokat szeretnénk készíteni és gráfunkhoz hozzáadni őket, alkalmazzuk a **CONSTRUCT** kulcsszót. A CONSTRUCT szintaxisa a SELECT parancséhoz hasonló, azzal a különbséggel, hogy a lekérdezésben értéket kapó paramétereken felül szerepelnie kell a belőlük kialakított, újonnan beszúrandó hármasnak is. Az alábbi példa a CONSTRUCT használatát szemlélteti. Láthatjuk, hogy példánk az előző lekérdezésen alapul: felveszi a tanárként és diákként is szereplő személyeket PhD hallgatóként a ?department paraméterben visszakapott tanszékkel.

```
PREFIX iit: <http://iit.bme.hu/ns/>
CONSTRUCT {
?teacher <http://phd.info/department> ?department
}
WHERE {
     {
     ?tcourse iit:course.course.taught_by ?teacher .
     ?tcourse iit:course.course.department ?department .
     ?teacher iit:type.person.neptun_code ?code .
```

```
    ?teacher iit:type.person.year_of_birth ?birth .
    FILTER (?birth > 1984)
    }
    {
    ?acourse iit:course.course.taken_by ?student .
    ?student iit:type.person.neptun_code ?code
    }
}
```

A jelentősebb SPARQL parancsok közül érdemes még megemlíteni az **ASK** kulcsszót, amely eldönti és egy Boolean értékkel jelzi, hogy egy megadott formátum illeszthető-e a gráfra. Az alábbi példa igazként értékelődik ki, ha a két megadott hallgató ugyanazt az előadást látogatja.

```
PREFIX iit: <http://iit.bme.hu/ns/>
ASK {
?course iit:course.course.taken_by iit:person.bitje_imre .
?course iit:course.course.taken_by iit:person.gaz_geza .
}
```

Az alábbiakban az adatmódosító (SPARQL/Update) nyelvi bővítmények közül mutatom be a jelentősebbeket. Az **INSERT** kulcsszó segítségével szúrhatunk be új hármasokat, pontosvesszővel elválasztva egy alanyhoz akár többet is. Az alábbi példa egy új PhD hallgató felvételét szemlélteti névvel és Neptun kóddal.

```
PREFIX iit: <http://iit.bme.hu/ns/>
INSERT
{ <http://phd/student42>      iit:info.code "QWERT1" ;
                              iit:info.name "Gáz Géza" .
}
```

Adatok törlésére a **DELETE** kulcsszó használható; több különböző módosító operációt a **MODIFY GRAPH** paranccsal foghatunk össze. Egy hallgató nevét például a következő módon változtathatjuk meg:

```
PREFIX iit: <http://iit.bme.hu/ns/>
MODIFY GRAPH <http://phd/students>
DELETE
{ <http://phd/student42> iit:info.name "Gáz Géza" }
INSERT
{ <http://phd/student42> iit:info.name "Gőz Géza" }
```

A DELETE és INSERT utasításnál a lekérdezéseknél már ismertetett módon alkalmazhatjuk a WHERE feltételt (szűrőkkel együtt). Gráfjainkra nevekkel is hivatkozhatunk, ebben az esetben az **INSERT INTO GRAPH**, illetve **DELETE FROM GRAPH**. A következő példa szemlélteti, hogyan helyezhetünk át ezen nyelvi elemek segítségével egy hallgatót MSc-ről PhD képzésre. A lekérdezés először beszúr, majd törli az eredeti gráfból a hármasokat. A ?p és ?v paraméter "predicate" és "value" jelentésű; azt jelezzük velük, hogy minden olyan hármast szeretnénk beszúrni, ami az adott alanyra illeszkedik. Az "str(param)" beépített függvény típuskonverziót végez, ezért string típusú objektumok összehasonlításakor különösen hasznos.

```
PREFIX iit: <http://iit.bme.hu/ns/>
INSERT INTO GRAPH <http://phd/students>
{ ?student ?p ?v }WHERE
{ GRAPH <http://msc/students>
     {    ?student iit:info.name ?name .
          FILTER (str(?name) = "Gőz Géza")
     }
}
DELETE FROM GRAPH <http://msc/students>
{ ?student ?p ?v }
WHERE
{ GRAPH <http://msc/students>
     {    ?student iit:info.name ?name .
          FILTER (str(?name) = "Gőz Géza")
     }
}
```

A fentieken túl még számos nyelvi elem tartozik a SPARQL szintaxisba, például a gráfok létrehozására és törlésére használható **DROP** és **CREATE GRAPH**. A teljes SPARQL nyelv részletes specifikációja elolvasható a [10][11] forrásokban. A 3.1 részben bemutatom, hogy a fentieken felül a SPARQL nyelvet milyen új elemekkel volt szükséges bővíteni a biztonsági szolgáltatások használatához.

2.2.3. Szemantikus webalkalmazások és az SAWSDL

Szemantikus webalkalmazás alatt (SWS, Semantic Web Service) olyan szolgáltatásorientált (SOA) architektúrában futó rendszert értünk, melynek bemeneteit, kimeneteit, pre- és posztkondícióit tudásreprezentációs nyelveken és adattípusokkal írjuk le [12][13]. A szemantikus technológiák előnye a hagyományos megoldásokkal szemben, hogy automatizálhatóvá teszik a szolgáltatások felderítését, meghívását és dinamikus kompozícióját. Ezek az eljárások a felhasználó szemszögéből lehetővé teszik, hogy üzleti folyamatok formális leírása alapján válasszanak ki vagy állítsanak össze webalkalmazásokat, olyan kritikus felhasználási területeken növelve az interoperabilitást, mint például az egészségügy [13].

Az SWS eljárások fő feladata, hogy webalkalmazások interfészét ontológiákhoz kössék. Ontológián valamely koncepció formális, gép által feldolgozható, megosztott, egy adott területre szűkített értelmezésű reprezentációját értjük [13]. Általában logikai formalizmusokra épülnek, és formális nyelvekkel írjuk le őket; ilyen például az RDF/XML alapú OWL (Web Ontology Language). Modellezési szemantikája kissé hasonló az objektumorientált elvekhez: megkülönböztet példányokat, osztályokat (igaz, egy példány több osztályba is tartozhat vagy akár egybe sem), öröklési hierarchiát és attribútumokat. Az alábbi példa a „macska” osztályt írja le OWL szemantikával. Az **owl:disjointWith** azt jelzi, hogy a két osztály példányai diszjunkt halmazt alkotnak (az OWL részletes bemutatása túlmutat jelen dolgozat keretein, de a teljes specifikációt az érdeklődők megtalálják a [14] weboldalon).

```
<owl:Class rdf:about="Cat">
     <rdfs:subClassOf rdf:resource="#Animal"/>
     <owl:disjointWith rdf:resource="#Dog"/>
</owl:Class>
```

Az ontológia- és interfészleírás megléte önmagában nem elég szemantikus webalkalmazások fejlesztéséhez, hiszen szükséges ezeket összekötni, egymásra leképezni. Míg a korai szemantikus annotációs megoldások – OWL-S, WSMO - top-down elven működtek, azaz az ontológia megadása a konkrét implementációtól függetlenül zajlott, majd bekerült az interfészleíróba, az újabb eljárások már bottom-up módszert követnek, vagyis a kész WSDL fájlba modulárisan kerülnek be szemantikus adatokra vonatkozó *referenciák* [12]. Ilyen megoldás az **SAWSDL**, amelyet a **WSMO-Lite** nevű keretrendszer segítségével köthetünk szemantikákhoz. A WSMO-Lite ugyan használható REST webalkalmazásokkal is, de mivel azok ritkán alkalmaznak WSDL, vagy akár a számukra kifejlesztett WADL formátumú interfészleírásokat, ezért készült hozzájuk egy MicroWSMO nevű megoldás. Utóbbi egy speciális mikroformátummal (hRESTS) géppel feldolgozható tett, eredetileg ember számára írt XHTML oldalakat – általában az adott RESTful alkalmazás formális fejlesztői dokumentációját – fogad és szemantikus adatokhoz köti őket [12]. Mivel munkám elsősorban SOAP alapú webalkalmazásokra fókuszál, a továbbiakban az SAWSDL-t mutatom be.

Az SAWSDL a WSDL szabvány kiterjesztése. Alapvetően két új fogalmat vezet be: a **modellreferenciát** és a **sémaleképezést**. Előbbi egy olyan attribútum, amelyet bármely WSDL séma elemhez (interfész, operáció, üzenet) hozzáadva egy vagy több szemantikus koncepcióhoz köthetjük azt. Sémaleképezésből kétféle létezik: **lifting** és **lowering**. Előbbi egy SOAP üzenet paramétereit szemantikához (pl. RDF gráfhoz) köti, utóbbi a szemantikus modellből generál üzenetet. Jellegzetes hívási konvenció lehet például a kérés elkészítése lowering leképezéssel, vagy lifting használata a válasz szemantikus feldolgozásához. Az alábbi példa eredetije a hivatalos WSDL 1.1 specifikációból való [15]; a félkövéren szedett kiegészítéseket az SAWSDL szolgáltatásinak szemléltetése céljából adtam hozzá. Láthatjuk, hogy a kérésnél megjelenik a szemantikus modellreferencia és a lowering sémaleképezés, amely az RDF adatok alapján összeállítja az üzenetet.

```
<xs:element name="listFlightsRequest"
sawsdl:modelReference="http://www.w3.org/2002/ws/sawsdl/spec/
ontology/flightlist#ListFlightsRequest"

sawsdl:loweringSchemaMapping="http://www.w3.org/2002/ws/sawsd
l/spec/mapping/RDFOnt2Request.xml">

      <xs:complexType name="tListFlights">
            <xs:sequence>
```

```
            <xs:element name="travelDate" type="xs:date"/>
            <xs:element name="startCity" type="xs:string"/>
            <xs:element name="endCity" type="xs:string"/>
            </xs:sequence>
      </xs:complexType>
</xs:element>
<xs:element name="listFlightsResponse"
```

sawsdl:modelReference="http://www.w3.org/2002/ws/sawsdl/spec/ontology/flightlist#ListFlightsResponse">

```
      <xs:complexType name="tFlightsResponse">
            <xs:sequence>
            <xs:element name="flightNumber"
                  type="xs:integer" minOccurs="0"
                  maxOccurs="unbounded"/>
           </xs:sequence>
      </xs:complexType>
</xs:element>
```

A konkrét szemantika leírása általában WSMO-Lite segítségével történik, ami négy szemantikát különböztet meg: az **információs modell** az üzenetekhez, a **funkcionális** a webalkalmazás által nyújtott szolgáltatásokhoz kötődik, a **viselkedési** a kliens által követendő protokollokat írja le (pl. pre- és posztkondíciók), a **nem funkcionális** pedig olyan szerver oldali paramétereket, mint például a QoS [12]. Az alábbi példák az egyes szemantikákat hivatottak szemléltetni (RDF formátumban, a prefix importokat elhagyva). Az első példa egy oktatási szolgáltatás információs modelljét ír le, ahol a tanár által tanított kurzus és annak kreditszáma, mint attribútum jelenik meg.

```
<> a wsl:Ontology.

iit:Teacher a rdfs:Class .
iit:teachesCourse a rdf:Property ;
     rdfs:domain iit:Teacher;
     rdfs:range iit:CourseService .
iit:CourseService a rdfs:Class .
iit:hasCourseData a rdf:Property ;
     rdfs:domain iit:CourseService ;
     rdfs:range iit:CourseData .
iit:CourseData a rdfs:Class .
     iit:creditNumber a rdf:Property ;
     rdfs:domain iit:CourseData ;
     rdfs:range xs:integer .
```

A második RDF dokumentum a funkcionális szemantikát mutatja be a típus-taxonómia („leszármaztatás”) példáján keresztül. A **wsl:FunctionalClassificationRoot** azért szerepel tárgyként az **iit:CourseService** alanyhoz, hogy jelezze, az a taxonómia gyökere („ősosztály”).

```
iit:CourseService a wsl:FunctionalClassificationRoot .
iit:ExamCourseService rdfs:subClassOf iit:CourseService .
```

Mint már említettem, nem funkcionális szemantikán olyan kontextuális információkat értünk, mint a QoS vagy egy webszolgáltatás ára. Az alábbi példa megmutatja, egy adott tárgy és az azt oktató tanár milyen OHV-értékelést kapott.

```
iit:OhvValue rdfs:subClassOf wsl:NonFunctionalParameter .
iit:CORBACourseValue a iit:OhvValue;
iit:teacherGrade "5"^^iit:ohvGrade;
iit:subjectGrade "4"^^iit:ohvGrade .
```

Az utolsó példa a viselkedési szemantika; láthatjuk, hogy a tárgy indításának előfeltétele a pozitív kreditszám, a posztkondíció (effect) pedig a kurzus megjelenése a tanár oldalán.

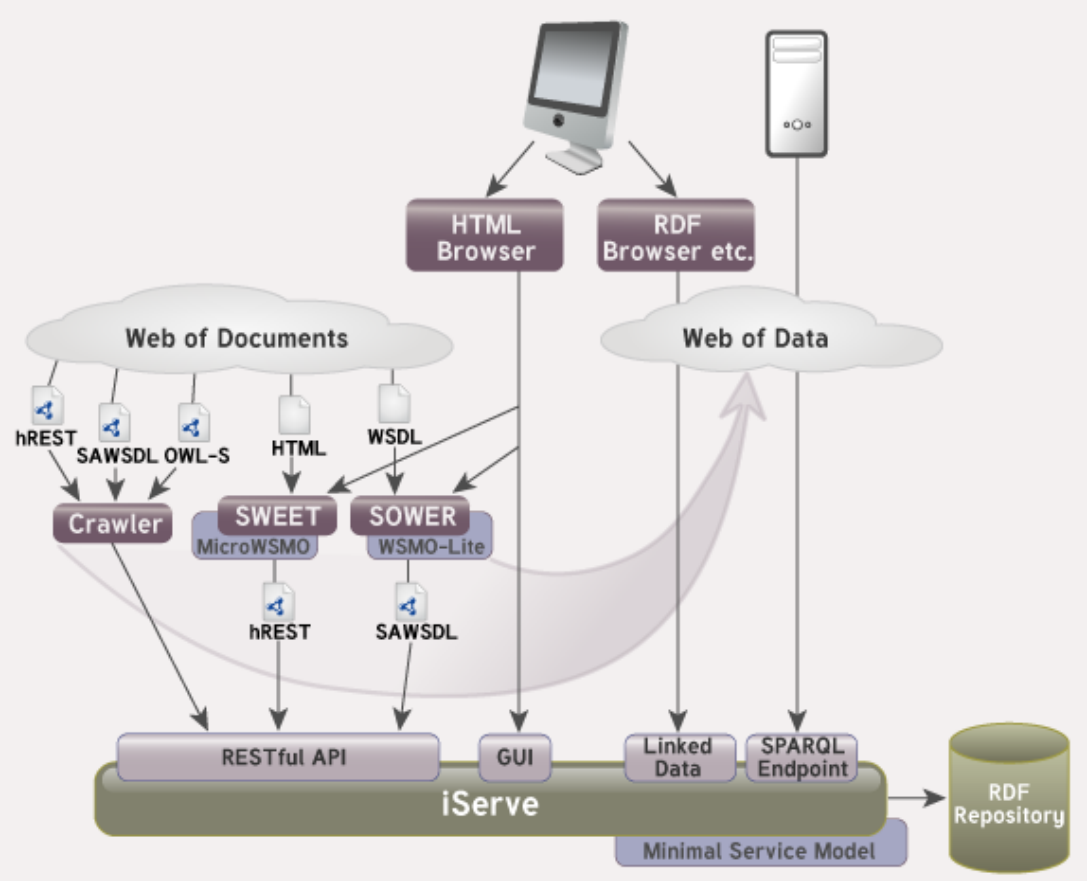
```
iit:CourseStartPrecondition a wsl:Condition ;
     rdf:value "'"
     ?coursedata[iit#creditNumber hasValue ?cr]
     memberOf iit#CourseData and
     ?cr > 0
     "'"^^wsml:AxiomLiteral .

iit:CourseStartEffect a wsl:Effect ;
     rdf:value "'"
     ?teacher[iit#teachesCourse hasValue ?course]
     memberOf iit#Teacher
     "'"^^wsml:AxiomLiteral .

wsml:AxiomLiteral a rdfs:Datatype .
```

2.3. A rendszerarchitektúra építőelemei

2.3.1. Az iServe webalkalmazás repository

Nem a nyílt forráskódú iServe [16] az egyetlen olyan webalkalamzás tároló, amely képes szemantikus annotációkat kezelni, de mind kezelhetősége, mind bőséges szolgáltatásai miatt rá esett a választásom. Az egyik első, UDDI alapú szemantikus repository a METE-OR-S volt [17], amely kezdetben egy korai, szemantikus metaadatokkal bővített WSDL-re épülő interfészleíró szabványt használt. A más filozófiára építő iServe megjelenését az UDDI publikus webes környezetben elszenvedett kudarca indukálta.

Az iServe fejlesztése a SOA4ALL EU projekt keretében történik, a brit Open University kutatcsoportja által. Célja, hogy lightweight, gép és ember által egyaránt egyszerűen használható szemantikus technológiák segítségével a szolgáltatások leírását kifejezőbbé tegye és felderítésüket megkönnyítse.



3. ábra: Az iServe rendszer felépítése (forrás: http://iserve.kmi.open.ac.uk/)

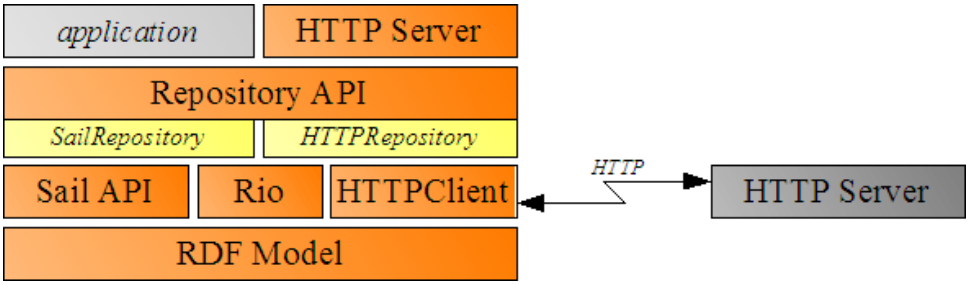
A fenti ábra mutatja az iServe architektúra-szerkezetét. Az adatok tárolása egy távolról elért RDF repository-ban történik; munkám során erre a feladatra a 2.3.2 részben bemutatott Sesame-ot használtam. Az információk között egy RDF böngészővel, vagy az iServe

által kiajánlott SPARQL endpoint segítségével kereshetünk, ami a repository felé tovább delegálja a lekérdezést. Ez a SPARQL interfész a GWT alapú webes kezelőfelületről is elérhető. A webszolgáltatások regisztrálására és automatizált felderítésére az iServe egy Restlet segítségével írt RESTful API-t nyújt, amit egyrészt az alkalmazásokat összegyűjtő web crawler, másrészt a SWEET és a SOWER használ. Utóbbiak olyan webalkalmazások, amiknek segítségével könnyen annotálhatunk már meglévő interfészleírásokat szemantikus információkkal. A SWEET olyan ember által is olvasható HTML oldalakat alakít hREST dokumentummá MicroWSMO segítségével, amelyek egy webalkalmazás interfészét írják le (lásd 2.2.3 rész). A SOWER WSDL fájlokat vár, amikből WSMO-Lite ontológiákkal SAWSDL-t gyárt. Mivel ezeken felül az iServe kezeli még az OWL-S és WSMO szabványt, valamint az SA-REST nevű, kevésbé ismert formátumot, szükséges volt egy olyan RDF ontológiát bevezetni, amely lehetővé teszi köztük az átjárhatóságot. Ez a Minimal Service Model, avagy MSM.

A továbbiakban bemutatom, hogyan történik rendszeremben a regisztrált webalkalmazásokból ontológiáiból kinyert RDF adatok tárolása és módosítása.

2.3.2. A Sesame szemantikus adattároló

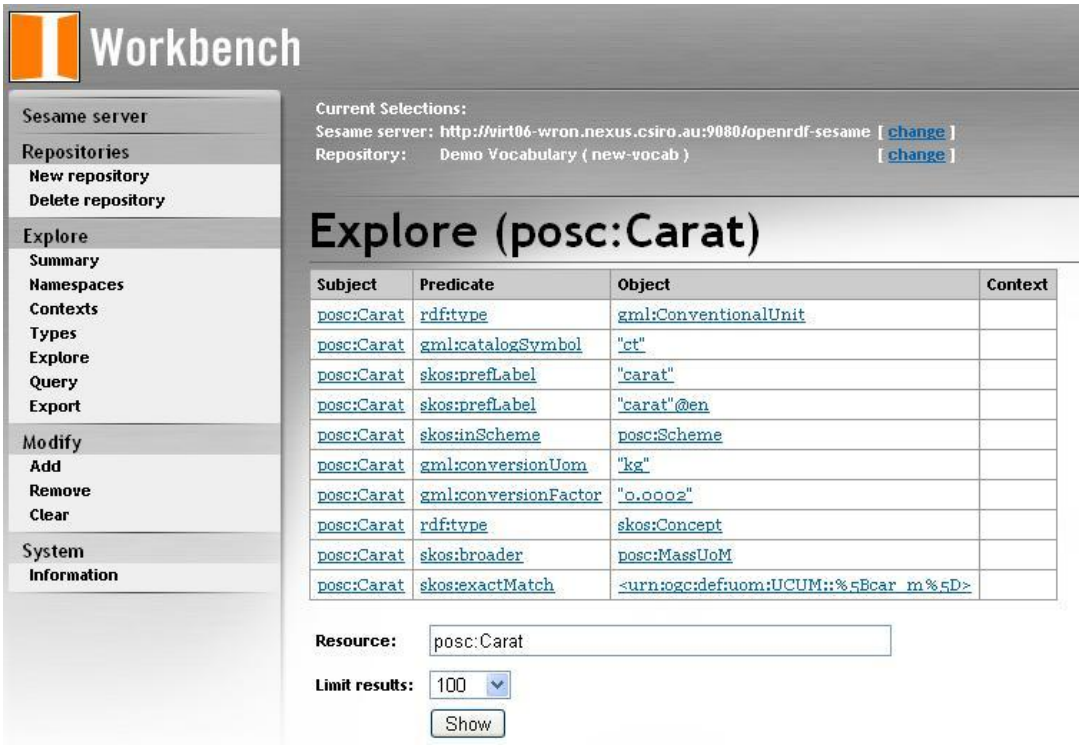
Az iServe bármilyen, RDF2Go szabványnak megfelelő interfészt nyújtó szemantikus adattárolóval képes együttműködni [18]; választásom ezekből a Sesame nevű [19], nyílt forráskódú, Java nyelven íródott rendszer 2.5.1-es verziójára esett, elsősorban jól dokumentáltsága és könnyen bővíthető API-ja miatt.



4. ábra: A Sesame rendszer rétegei

A Sesame két API-t nyújt az RDF adatok elérésére és lekérdezésére: az alacsonyabb szintű Sail egy absztrakciós réteg a fizikai adattároló felett, míg a magasabb szintű Repository fájlok és más szemantikus adatok exportálását, importálást és feldolgozását teszi lehetővé. Lekérdezéshez használhatjuk a SPARQL (lásd: 2.2.2.), illetve a saját fejlesztésű SeRQL nyelveket. I/O műveletekre a Rio RDF parser programkönyvtár használható, ha pedig HTTP protokollon keresztül szeretnénk elérni tárolónkat, a Sesame Java

Servleteket kínál erre a célra (az ábrán: HTTP Server néven szerepelnek). Rendszerünket konzolból vagy az openRDF Workbench nevű webes kezelőfelületről is elérhetjük (utóbbihoz, és a tároló telepítéséhez is szükséges egy Servlet container).



5. ábra: Az openRDF Workbench felhasználói felülete (forrás: http://tiny.cc/qbdv5)

Az információk tárolása többféle adatbázistípusban történhet. Lehetőségünk van memóriában tárolt (opcionális perzisztenciával), illetve háttértárra író (indexelhető) natív RDF adatbázis használatára, ami képes implicit összefüggéseket is kinyerni adatainkból (inferencing). Beállíthatunk relációs adatbázist is tároló közegként, amit a Sesame JDBC driveren keresztül ér el; a jelenleg támogatott MySQL és PostgreSQL közül én az utóbbit választottam. (A natív tárolás nem lehetett alternatíva, hiszen a 2.1.2 és 2.1.3 részben bemutatott algoritmusok relációs adatbázisokhoz készültek). Szemantikus-relációs leképezésnél megadhatjuk azt is, hogy a Sesame egy táblát használjon, vagy predikátumonként készítsen újat; ennek az opciónak a teljesítmény finomhangolása során van jelentősége.

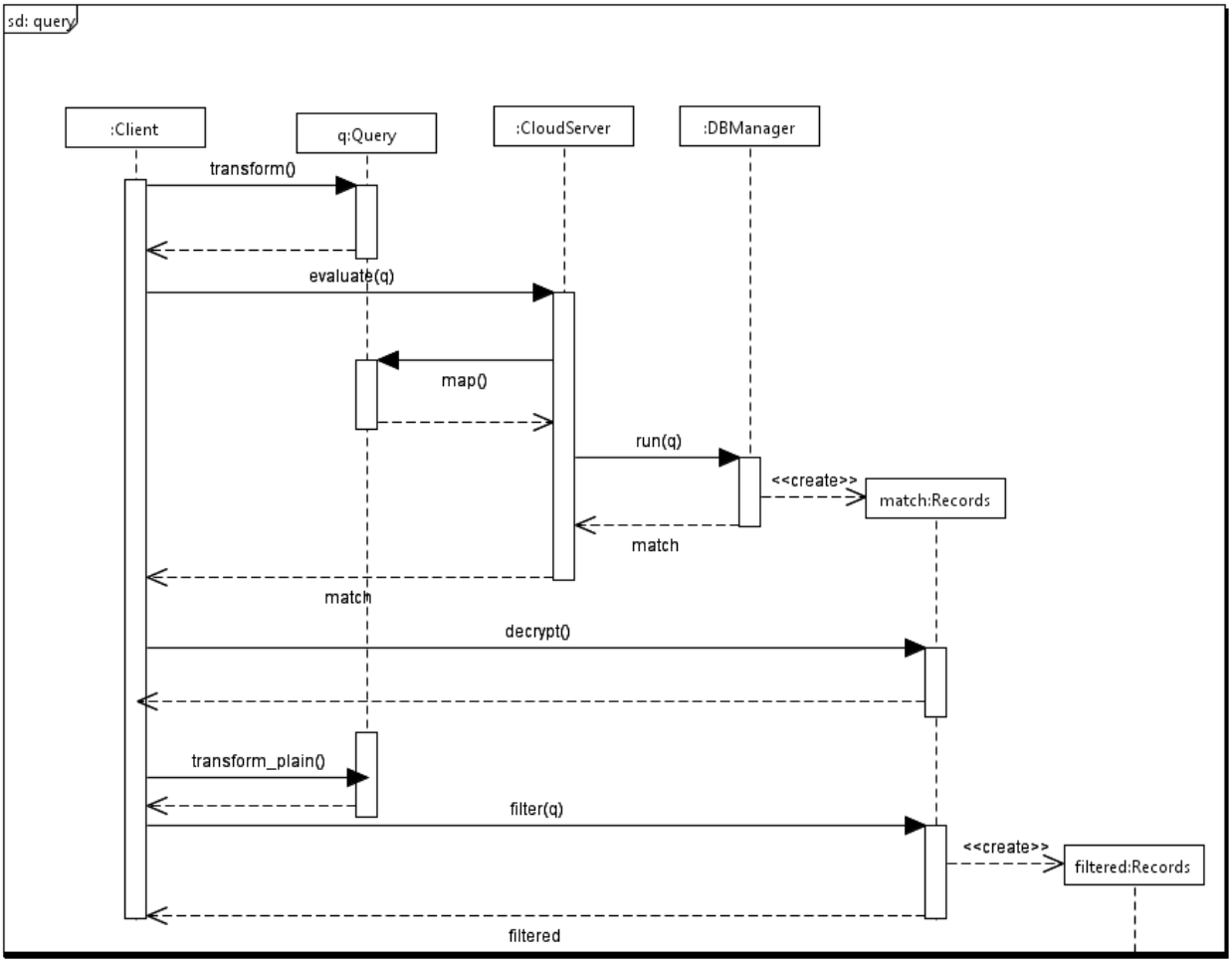
A Sesame framework sikerét és fontosságát mutatja, hogy nagy teljesítményű, ipari színvonalú és hátterű továbbfejlesztése, az OWLIM felhasználói között olyan neveket találunk, mint a BBC, a Raytheon biztonságtechnikai vállalat, vagy a kanadai kormány [20]. Láthatjuk, hogy a szemantikus web mostanra dinamikusan fejlődő, gyakorlati haszonnal bíró technológiává lépett elő, épp ezért tartom fontosnak biztonsági szolgáltatásaik bővítését és a cloud megoldásokkal való integrációjukat.

3. RÉSZLETES RENDSZERTERVEK

3.1. A SPARQL nyelv kibővítése

A kriptográfiai funkciók biztosításához szükséges volt a SPARQL szintaxist új kulcsszavakkal bővíteni. A SPARQL specifikáció [11] tartalmaz néhány, az SQL tárolt eljárásokhoz hasonló konvencióval hívható ún. beépített függvényt – ilyen például a 2.2.2. részben már említett **str**, vagy a reguláris kifejezéseket kiértékelő **regex.** A rejtjelezett adatok felvitele, valamint a biztonságos lekérdezések transzformációja a szerverre küldés előtt szükségessé tette, hogy új beépített függvények bevezetésével létrehozzak egy új, SPARQL alapú lekérdező nyelvet. Ez a nyelv, ami a Chispa nevet kapta – spanyol, jelentése magyarul „szikra", akárcsak az angol „sparkle" szónak, aminek a SPARQL többé-kevésbé fonetikus átírása – elfogad minden, a 2008.01.15-ös SPARQL [11], valamint a 2008.07.15-ös SPARQL/Update [10] specifikációnak megfelelő lekérdezést, de kezeli a 2.1.2 és 2.1.3 részben bemutatott algoritmusok kissé módosított és megerősített változatát. A lekérdezőnyelv interpretálásáért felelős logikát Java nyelven készítettem el.

A rejtjelezett szemantikus adatokon folytatott lekérdezések biztosítására öt műveletet szükséges definiálni. Az első a **transzformáció**, amely a felhasználó által hívott kriptográfiai funkciókat kiértékeli és a nem megbízható szerver számára értelmezhető funkciókkal és paraméterekkel helyettesíti. Ezt követi a **leképezés,** amely során a szerver kiértékeli a beépített függvényeket, és a fizikai adatstruktúrának megfelelő módon végrehajtja a rejtjelezett lekérdezéseket – relációs adatbázis használatánál például beépített eljárásokat használ az indexek szűrésére. Az eredményül kapott rekordokat a kliens **dekódolja,** majd elvégez egy második **transzformációt** az eredeti lekérdezésen, hogy olyat kapjon belőle, amely a kapott nyílt szövegű adatokból ki tudja **szűrni** a hamis pozitívokat. Az adatbeszúrás, törlés vagy módosítás során a transzformáció a fentiekhez teljesen hasonló módon zajlik; de a leképezés ebben az esetben mindössze annyiban tér el a közönséges SPARQL/Update SQL mapping-től, hogy egyes mezőkbe rejtjelezett érték kerül, és ezekre a szerver beszúrja az index attribútumot is. Mivel a kliens adatmódosító műveletek során nem vár rekordokat válaszként, a dekódolás, a második lekérdezés-transzformáció és a szűrés ilyenkor nem szükséges. Az alábbi (sematikus, a konkrét implementációt és objektumtípusokat leegyszerűsítő) UML szekvencia-diagram a biztonságos lekérdezés működésének egyes fázisait hivatott szemléltetni.



6. ábra: A biztonságos lekérdezés menete

Láthatjuk, hogy a kliens átalakítja a lekérdezést (transform), elküldi a nem megbízható cloud szolgáltatónak, aki az adattípusra való leképezés (map) után kiértékeli és visszaküldi a lekérdezésre illeszkedő rejtjelezett eredmény-rekordokat (match). A kliens ezt dekódolja (decrypt), majd összeállítja a hagyományos SPARQL query-t (transform_plain), amivel kiszűri (filter) az indexelő algoritmus(ok) jellegéből adódó, overhead-ként jelentkező fölös rekordokat.

3.1.1. A lekérdező szintaxis új elemei

Rejtjelezett paraméterek három helyen jelenhetnek meg SPARQL lekérdezéseinkben: a szűrőfeltételekben (FILTER, lásd 2.2.2. rész), gráf-mintaillesztésnél literálok helyett, illetve a SELECT paraméter-deklarációiban. A következőkben táblázatos formában bemutatom a Chispa új, kriptográfiai célú kulcsszavainak és beépített függvényeinek működését és hívási konvencióik formátumát és transzformációs szabályait. Ahol külön nem jelölöm, a bővített Backus-Naur formájú (EBNF) leírásban szereplő, nem általam bevezetett

SPARQL tokenek (pl. Var a változók jelölésére) definíciója a [11][10] forrásokból származik. Az itt felsorolt új nyelvi elemek nem kisbetű-nagybetű érzékenyek.

3.1.1.1. Rejtjelezett paraméterek jelölése

Token neve: SecureParam

EBNF leírása: *SecureParam* ::= ('SECURE_STR:' | 'SECURE_FLOAT:' | 'SECURE_INT:')Var . ; ahol Var szabványos SPARQL változó.

Leírása: Egy string, egész-, vagy lebegőpontos szám típusú változót rejtjelezettként deklarál. Segítségével a Chispa interpreter kliens oldalon követni tudja, mely visszakapott paraméterek értékeit kell dekódolnia. Szintaxisa a tényleges rejtjelező algoritmustól független; az általam elkészített mintaimplementáció 128 bites AES blokk-kódolót használ.

Transzformációs szabályai: ('SECURE_STR:' | 'SECURE_FLOAT:' | 'SECURE_INT:')Var => Var, azaz a szerverre már szabványos SPARQL változóként érkezik meg a deklarált paraméter.

Működés szemléltetése: Legyen az eredeti lekérdezés az alábbi:

```
SELECT SECURE_STR:?title
WHERE
{
     <http://example.org/book/book1>
     <http://purl.org/dc/elements/1.1/title> ?title .
}
```

Láthatjuk, a lekérdezés szűrés nélkül illeszt minden olyan hármasra, amely egy könyvet reprezentál, és visszaadja az adott rekord cím (title) mezőjét. A SELECT kulcsszó után következő SecureParam azt jelzi a kliensnek, hogy a szerver az adott paraméter értékeit rejtjelezetten fogja visszaadni. Ennek feldolgozása után az alábbi szabványos SPARQL lekérdezés fog eljutni a szerverre, a kliens pedig megjegyzi, hogy az eredményt dekódolni kell. A szervernek ebben az esetben mindegy, rejtjelezett-e az eredmény, mert csak illeszt a gráfra, de nem szűr a ?title paraméter értéke szerint.

```
SELECT ?title
WHERE {
     <http://example.org/book/book1>
     <http://purl.org/dc/elements/1.1/title> ?title . }
```

3.1.1.2. Beépített hash függvény szövegek kezelésére

Token neve: StringHash

EBNF leírása: *StringHash* ::= 'HASH_STR' '(' Var ', ' LiteralExpr+') ' .

LiteralExpr ::= (BracketStringExpr ('||' BracketStringExpr)* ('&&' BracketStringExpr)* .

BracketStringExpr ::= (StringExpr | '(' StringExpr')') .

StringExpr ::= (STRING | 'LIKE' STRING | 'NOT LIKE' STRING)+ .; ahol STRING SPARQL szöveg típus.

Leírása: Egy lekérdezés szűrő feltételében jelzi, hogy az adott paramétert a szerver rejtjelezetten tárolja. A LIKE, illetve a NOT LIKE kitétel szövegrészletekre való egyezést jelent. Egy paraméterhez több kitétel is tartozhat, ezeket logikai operátorokkal kell elválasztani. A paraméter indexelése a 2.1.3. részben taglalt algoritmus megerősített (lásd 3.2. rész) változatával történik. Az indexelés elvégzése után a lekérdezést elküldhetjük a szerverre.

Transzformációs szabályai: 'HASH_STR' '(' Var ', ' LiteralExpr+) => 'INDEX_STR' '(' Var ', ' LiteralIndexExpr+); a LiteralIndexExpr token feloldását lásd a 3.1.1.3. részben.

Működés szemléltetése: Vegyük az alábbi lekérdezést:

```
PREFIX  dc:  <http://purl.org/dc/elements/1.1/>
SELECT  SECURE_STR:?title SECURE_STR:?author
WHERE
{
          ?x dc:title ?title
          ?x dc:author ?author
          FILTER HASH_STR(?author, "John Smith")
          FILTER  HASH_STR(?title,  LIKE  "the"  &&  NOT  LIKE
          "\"hello world\"")
}
```

A lekérdezés jelentése a következő: A ?title (cím) és ?author (szerző) rejtjelezett paramétereket illesszük a WHERE feltételben megadott hármasokra, majd szűrjünk azokra, ahol a

szerző neve „John Smith”, a címben pedig szerepel a „the” szöveg, de nem szerepel a „hello world”, mint literál (azaz a szöveg idézőjelek között; a Chispa kezeli a whitespace és escape karaktereket). A transzformáció lezajlása után a szerverre eljutnak az index értékek, amelyekre szűrhet. Fontos megjegyezni, hogy ugyan a szemléltetés kedvéért rejtjelezett paramétereket deklaráltam a SELECT kifejezésben, ebben az esetben ezt akár el is hagyhatjuk, hiszen a szűrő feltételből a Chispa ki tudja következtetni, hogy kódolt értékeket fog kapni. A szerverre a következő kifejezés jut el (az indexelő függvény definícióját a 3.1.1.3. alfejezet részletezi):

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
SELECT ?title ?author
WHERE     {
               ?x dc:title ?title ?x dc:author ?author
               FILTER INDEX_STR(?author,
               "00000000010000010001101000100010")
               FILTER INDEX_STR(?title,LIKE
               "00000000010000000000000000000010" && NOT LIKE
               "00001010010110010000001110000010" )
          }
```

3.1.1.3. Szerver oldali beépített indexelő függvény szövegek kezelésére

Token neve: StringIndex

EBNF leírása: *StringIndex* ::= 'INDEX_STR' '(' Var ', ' LiteralIndexExpr+) ' .

LiteralIndexExpr ::= (BracketIndexExpr ('||' BracketIndexExpr)* ('&&' BracketIndexExpr)* .

BracketIndexExpr ::= (LiteralIndex | '(' LiteralIndex')') .

LiteralIndex ::= (BinaryStream | 'LIKE' BinaryStream | 'NOT LIKE' BinaryStream)+ .

BinaryStream ::= '"""' (0|1)+ '"""' .

Leírása: A kliens a 2.1.3. részben bemutatott algoritmus segítségével a szűrőfeltételekben

szereplő összes karakterpárhoz hozzárendel egy egész számot, majd a kezdetben csupa 0 index megfelelő elemét 1-be állítja. Például a 3.1.1.2. részben látható LIKE „the” kifejezésben két párt („th”, „he”) vizsgál, majd a nekik megfelelő indexeket írja át (esetünkben a 10. és 31. elemet). A szerver ebből az adattípusnak megfelelő lekérdezést tud gyártani, azaz megnézi, az adott paraméter indexe pontosan ugyanaz, mint a klienstől kapott, illeszkedik az 1-es értékekre (LIKE), vagy nem illeszkedik az 1-es értékekre (NOT LIKE).

Transzformációs szabályai: N/A (lásd 3.1.2. rész)

Működés szemléltetése: Lásd 3.1.1.2. rész.

3.1.1.4. Beépített hash függvény számértékek kezelésére

Token neve: NumHash

EBNF leírása: *NumHash* ::= ('HASH_INT' | 'HASH_FLOAT') '(' LogicalNumericalExpression+ ') ' .

LogicalNumericalExpression ::= (BracketedRelationalExpression ('||'BracketedRelationalExpression)*('&&'BracketedRelationalExpression)* .

BracketedRelationalExpression ::= (SimpleRelationalExpression | '('SimpleRelationalExpression')' | VarRelationalExpression | '('VarRelationalExpression')') .

SimpleRelationalExpression ::= (NumericLiteral ('==' Var | '!=' Var | '<' Var | '>' Var | '<=' Var | '>=' Var)) | (Var ('=' NumericLiteral | '!=' NumericLiteral | '<' NumericLiteral | '>' NumericLiteral | '<=' NumericLiteral | '>=' NumericLiteral)) . ; ahol NumericLiteral szabványos SPARQL számérték (egész vagy lebegőpontos).

VarRelationalExpression ::= (Var ('==' Var | '!=' Var | '<' Var | '>' Var | '<=' Var | '>=' Var)) .

Leírása: Szűrőfeltételekben szereplő, logikai operátorokkal összekötött relációs kifejezé-

seket értékel ki rejtjelezett numerikus (egész vagy lebegőpontos) adatokon a 2.1.2. részben ismertetett indexelő hash algoritmus segítségével, a konkrét implementáció meghatározása nélkül. Az opcionálisan zárójelezett kifejezésekben szám-paraméter, illetve paraméter-paraméter közötti egyenlőtlenségeket is kiértékelhetünk.

Transzformációs szabályai: 'HASH_INT' '(' LogicalNumericalExpression ') ' => 'INDEX_INT' '('LogicalBinExpression')' 'HASH_FLOAT '(' LogicalNumericalExpression ') ' => 'INDEX_FLOAT' '('LogicalBinExpression')'; a LogicalBinExpression token feloldását lásd a 3.1.1.5. részben.

Működés szemléltetése:

Nézzük az alábbi lekérdezést, amely olyan 1000 egységnél olcsóbb termékek jelenlegi és régi árát adja vissza, amelyek nem drágultak (a régi ár nagyobb vagy egyenlő az újjal). Figyeljük meg, hogy mivel a HASH_INT beépített függvény már jelzi, hogy a ?price és ?oprice paramétereket rejtjelezetten kell kezelni, a SELECT utasításban ezt nem szükséges külön deklarálni (de nem is tilos).

```
PREFIX dc:  <http://purl.org/dc/elements/1.1/>
SELECT ?price ?oprice
WHERE
{
     ?x dc:price ?price
     ?y dc:oldprice ?oprice
     FILTER HASH_INT(?price < 1000 && ?oprice >= ?price )
}
```

A transzformáció elvégzése után a következő oldalon látható köztes kód kerül a szerverre. Az INDEX_INT (vagy INDEX_FLOAT) beépített eljárás jelzi, hogy ládaindex szerint kell keresni. A logikai operátorokkal tagolt indexelő kifejezéseknek két típusát különböztetjük meg: ha az eredeti reláció paraméter és érték között volt értelmezve, a paraméter neve után következnek kapcsos zárójelek között vesszővel elválasztva azon ládák azonosítói, amikbe a paraméter az egyenlőtlenség alapján tartozhat. Paraméter-paraméter relációban a két pa-

raméter neve jelenik meg kettősponttal elválasztva, majd kapcsos zárójelek között felsorolva azok az indexpárok (paraméter1_index:paraméter2_index alakban), ahol a hozzárendelt ládapárokra a 2.1.2. részben bemutatott módon kiértékelhetjük az egyenlőtlenséget. Látható, hogy példánkban a ládaazonosítók kiosztása véletlenszerű volt. Azt, hogy az értékeket milyen algoritmus szerint osztjuk el, hány ládát használunk, és milyen függvény rendeli hozzájuk az azonosítót, a konkrét implementáció határozza meg.

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
SELECT ?price ?oprice
WHERE
{
     ?x dc:price ?price ?y dc:oldprice ?oprice
     FILTER   (   INDEX_INT(?price{508,   7313,   7500}   &&
     ?oprice:?price{4604:508,      4604:7313,      4604:7500,
     4604:8049, 4604:4501, 4604:6678, 4604:11394, 4604:11770,
     4604:9862, 4604:5720, 4604:1483, 1154:10048, 4604:10048,
     5265:10048,    332:10048,    6001:10048,    5525:10048,
     2582:10048} ))
}
```

Az általam megírt tesztrendszerben a névtérrel és a gráf-hármasok élével azonosítunk minden numerikus értéket (tehát pl. a dc:price esetében a dc prefixum névtere és a price együttesen azonosítja a típust). A cloud szerver olyan párokat kap, ami az előbb említett azonosítóból, valamint egy, az adat feltöltőjének és használóinak közös kulcsával rejtjelezett, az indexelő algoritmus adott típusra vonatkozó paramétereit leíró hármasból áll. A rejtjelezett paraméterek a (típus domain-je ismeretében becsült, nem kötelező érvényű) minimum és maximum érték, valamint a ládák száma (ténylegesen ennél kettővel több ládát használunk, fel kell ugyanis vennünk egyet a minimum alatti, egyet a maximum feletti értékeknek (az értékkészlet felosztása egyébként egyenletes). Ha például egy könyvet modellezünk, a rejtjelezett lekérdezések gyorsaságának biztosítására jó választásnak tűnhet 2000 és 6000 forint közötti értékeket felvenni 4 darab (1000 forintos) ládába, de nem szabad elfeledkeznünk arról, hogy ritkán nagyobb vagy kisebb értékek is megjelenhetnek. Természetesen előfordulhat olyan eset, hogy egy küszöbérték kötelező érvényű (pl. életkor pozitív), de ennek kezelése nem a szerver feladata, hiszen ő nem is ismerheti ezeket az

értékeket. Hogy a transzformációt elvégezhesse, a kliensnek el kell kérnie a cloud-tól az adott paraméterhez tartozó küszöbértékeket és ládaszámot, majd dekódolnia kell azokat.

3.1.1.5. Szerver oldali beépített indexelő függvény számértékek kezelésére

Token neve: NumIndex

EBNF leírása: *NumIndex* ::= **(**'INDEX_INT' | 'INDEX_FLOAT') '('LogicalBinExpression')' .

LogicalBinExpression ::= (BracketedBinExpression ('||' BracketedBinExpression)* ('&&' BracketedBinExpression)* .

BracketedBinExpression ::= (BinExpression|'('BinExpression')') .

BinExpression ::= (UnaryBinExpr | BinaryBinExpr) .

UnaryBinExpr ::= Var'{' (INTEGER ', ')* INTEGER'}' .

BinaryBinExpr ::= Var':'Var'{' (INTEGER ':' INTEGER ', ')* INTEGER ':' INTEGER '}' .

Leírása: A konkrét adattárolási megoldás függvényében illeszt rejtjelezett numerikus rekordokat szűrőfeltételre a szerveren. Fontos megjegyezni, hogy a kliensről a lekérdezést végző személy hibájából átkerülhetnek egymásnak ellentmondó logikai feltételek (pl. egy paraméter szerepel egy ládában ÉS nem szerepel benne). Az indexelő algoritmus jellegéből adódóan pedig felesleges feltételek mehetnek át (pl. a 3.1.1.4. részben mutatott lekérdezésnél a ?price{508, 7313, 7500} kitétel miatt a logikai ÉS után szereplő feltételből eleve kiesnek azon párok, amelyek második paramétere nem ezek közül való). Mivel ezen adatok csak minimális hálózati overhead-et generálnak, de a terheléselosztás szempontjából fontos, hogy ne csak a kliens végezzen számításigényes műveleteket, az ilyen problémák és redundanciák feloldása a szerver feladata.

Transzformációs szabályai: N/A (lásd 3.1.2. rész)

Működés szemléltetése: Lásd 3.1.1.4 rész.

3.1.1.6. Rejtjelező beépített függvény szövegek kezelésére

Token neve: StringEnc

EBNF leírása: *StringEnc* ::= 'ENC_STR' '(' STRING ')' .

Leírása: WHERE feltételben megjelenő, rejtjelezetten tárolt literál illesztése, illetve rejtjelezett szöveges paraméter beszúrása esetén a kliens elvégzi a kódolást és az indexelést.

Transzformációs szabályai: 'ENC_STR' '(' STRING ')' => 'DEC_STR' '(' EncHexaStr ',' LiteralIndex ')'; EncHexaStr feloldását lásd a 3.1.1.7. részben.

Működés szemléltetése: Lássuk a következő Chispa szintaxissal bővített SPARQL/Update kifejezést, amely a „bookStore" gráfba szúr be egy olyan könyvcímet, melyet rejtjelezetten szeretnénk megadni:

```
PREFIX dc:  <http://purl.org/dc/elements/1.1/>

INSERT DATA INTO <http://example/bookStore>
{
     <http://example/book3> dc:title
     ENC_STR("Fundamentals of Compiler Design")
}
```

A kliens oldali interpreter kódolja a szöveget (esetünkben AES 128-al), a kapott bájtfolyamból pedig egy számsorozatot generál, amely az eredeti string helyett kerül be a gráfba, valamint létrehozza az adott szöveg indexét is:

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
INSERT DATA INTO <http://example/bookStore>
{
     <http://example/book3> dc:title
     DEC_STR("6437c30888aa16f0ba6a298b79e92df91aeb6c468fb7dbe
     df61ade778c8fbece","11011011111110111110011011101111" )
}
```

3.1.1.7. Szerver oldali beépített függvény szövegek biztonságos kezelésére

Token neve: StringDec

EBNF leírása: *StringDec* ::= 'DEC_STR' '(' EncHexaStr ',' LiteralIndex ')' .

EncHexaStr ::= ' " " " ' ('0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F')+ ' " " " ' .

Leírása: A 3.1.1.6. részben említett módon transzformált lekérdezést vagy módosító utasítást a szerver úgy hajtja végre, hogy beszúrja, visszaadja, avagy törli az adott számsorozatra illeszkedő rekordokat (adatfelvétel vagy törlés esetén a paraméter indexe is bekerül a gráfba, ill. eltűnik onnan).

Transzformációs szabályai: N/A (lásd 3.1.2. rész)

Működés szemléltetése: Lásd 3.1.1.6 rész.

3.1.1.8. Rejtjelező beépített függvény számértékek kezelésére

Token neve: NumEnc

EBNF leírása: *NumEnc* ::= ('ENC_INT'|'ENC_FLOAT) '(' NumericLiteral ')' .

Leírása: WHERE feltételben megjelenő, rejtjelezetten tárolt számérték illesztése, illetve rejtjelezett numerikus értékű paraméter beszúrása esetén a kliens elvégzi a kódolást és az indexelést.

Transzformációs szabályai: 'ENC_INT' '(' NumericLiteral ')' => 'DEC_INT' '('

EncHexaStr',' INTEGER ')'

'ENC_FLOAT '(' NumericLiteral ')' => 'DEC_FLOAT' '(' EncHexaStr',' INTEGER ')'

Működés szemléltetése:

Az alábbi utasítás beszúr egy új, valamely könyv árát reprezentáló csúcsot a „bookStore” gráfba. Láthatjuk, hogy az ENC_INT beépített függvénnyel utasítottuk a kliens oldali interpretert, hogy a paramétert rejtjelezve küldje át.

```
PREFIX dc:  <http://purl.org/dc/elements/1.1/>
INSERT DATA INTO <http://example/bookStore>
{ <http://example/book3>  dc:price  ENC_INT(2500) }
```

A szerverre az alant látható formában jut el az utasítás. A DEC_INT szerver oldalon kiértékelt függvény első paramétere a rejtjelezett érték, a második a hozzá tartozó láda indexe.

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
INSERT DATA INTO <http://example/bookStore>
{
  <http://example/book3> dc:price
  DEC_INT( "1704fa589cfd756c6aee3d6a73cd88b7" , 6678 )

}
```

3.1.1.9. Szerver oldali beépített függvény számértékek biztonságos kezelésére

Token neve: NumDec

EBNF leírása: *NumDec* ::= ('DEC_INT' | 'DEC_FLOAT') '(' EncHexaStr',' INTEGER')' .

Leírása: A 3.1.1.8. részben említett módon transzformált lekérdezést vagy módosító utasítást a szerver úgy hajtja végre, hogy beszúrja, visszaadja, avagy törli az adott számsorozatra illeszkedő rekordokat (adatfelvétel vagy törlés esetén a paraméter indexe is bekerül a gráfba, ill. eltűnik onnan).

Transzformációs szabályai: N/A (lásd 3.1.2. rész)

Működés szemléltetése: Lásd 3.1.1.8. rész.

3.1.2. SPARQL-SQL leképezés

Mint már említettem, a Sesame tároló az alkalmazott rejtjelező algoritmusok tulajdonságaiból adódóan relációs adatbázist (PostgreSQL 9.0) használ. Az indexek szerint való kereséshez és adatmódosításhoz tehát szükséges tárolt eljárásokat definiálni. A PostgreSQL számos procedurális nyelvet támogat, ezek közül a PL/Python-ra esett a választásom [21].

A Python alapú scriptnyelv segítségével könnyen definiálhatunk tárolt eljárásokat az alábbi formában:

```
CREATE FUNCTION funcname (argument-list)
  RETURNS return-type
AS $$
  # PL/Python function body
$$ LANGUAGE plpythonu;
```

Érdemes megjegyezni, hogy noha a Python dinamikus tipizálású nyelv, adatbázisban futtatva a bemeneti paramétereknél és visszatérési értéknél explicit típusmegadás kell. Az alábbi példa, amellyel a rejtjelezett szöveges értékek indexét ellenőrzöm, jól szemlélteti ezt a szabályt:

```
CREATE OR REPLACE FUNCTION check_index("indexes" integer[],hash text):
RETURNS BOOLEAN AS
$$
    for index in indexes:
        retval=hash[index]
        if(retval!='1'):
            return False
    return True;
$$ LANGUAGE plpythonu;
```

Az egyes szerver oldali feladatok, és a hozzájuk tartozó SQL operációk ismertetése előtt szeretném röviden bemutatni, hogyan képezi le a Sesame az RDF gráfokat adatbázis táblákra (ez csak az egyik lehetőség, közös táblát is használhat a predikátumoknak, de az általam használt rendszer úgy konfiguráltam, hogy külön relációkat hozzon létre). Minden adattípushoz hozzárendel egy külön táblát (pl. **numeric_values** számokra, **label_values**

szövegekre, **uri_values** névterekre). Ezen táblák két attribútumból állnak: az elsődleges kulcs egy generált azonosító (ID, típusa integer), a másik pedig az adat konkrét értéke (típusa táblafüggő, például numeric_values esetében double, a label_values táblában pedig legfeljebb 255 hosszú varchar). Újonnan beszúrt adat esetén létrejön vagy módosul az a tábla, ami az adott hármas állítmányának (azaz az irányított gráf két csúcsa között futó élnek) felel meg. Lássuk az alábbi SPARQL/Update utasítás végrehajtása után bekövetkező változásokat az adatbázisban!

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
INSERT DATA
{ <http://example/book3> dc:title    "Minden megvan" ;
                         dc:author  "Ottlik Géza" .
}
```

A „szerző” (author) és „cím” (title) kapcsolatok leírására létrejön az **author_3** (felső) és a **title_2** (alsó) tábla az alábbi tartalommal:

ctx integer	**subj integer**	**obj [PK] integer**	**expl [PK] boolean**
0	1	805306370	TRUE

ctx integer	**subj integer**	**obj [PK] integer**	**expl [PK] boolean**
0	1	805306369	TRUE

Az utasításban szereplő névterek feloldása az **uri_values** táblába kerül. Figyeljük meg, hogy az azonosító (id) értéke megjelenik a kapcsolatokat leíró táblák nevében [kapcsolatnév]_[id] formában (pl. a title azonosítója 2, ezért nevezte el title_2-nek a rendszer a táblát). Észrevehetjük azt is, hogy a kapcsolatleíró táblák alany (subj) attribútuma valóban a „book3” névterének azonosítója.

id [PK] integer	**value character varying(255)**
1	http://example/book3
2	http://purl.org/dc/elements/1.1/title
3	http://purl.org/dc/elements/1.1/author

A **hash_values** táblába bekerülnek az egyes azonosítók – az uri_values id, a kapcsolatleíró táblák obj attribútuma - hash értékei, a **label_values** pedig eltárolja a beszúrt literálokat a kapcsolat tárgy (obj) attribútumával, mint elsődleges kulccsal.

id [PK] integer	**value character varying(255)**
805306369	Ottlik Géza
805306370	Minden megvan

A fenti leképezéseket ki kell bővíteni néhány új lépéssel, hogy a paraméterek rejtjelezett lekérdezése megoldott legyen. A titkosított érték szöveg és numerikus paraméter esetén is beszúrható a label_values tábla value oszlopába, hiszen rejtjelezve a számok is hexadecimális string formájában kerülnek eltárolása. Az index tárolására két megoldás adódik: vagy felveszünk egy-egy új táblát az azonosítóknak és a numerikus (típusa integer), illetve a szöveges indexeknek (típusa 0 és 1 karakterekből álló string), vagy a label_values-t bővítjük két új oszloppal (vagy eggyel, ha a numerikus indexeket is szövegként tároljuk). Előbbi megoldás előnye, hogy nem módosítjuk a Sesame által definiált táblát, az utóbbié pedig az, hogy alkalmazásával kevesebb SQL join művelet szükséges egy SPARQL lekérdezéshez.

Titkosított szövegek szűrésére a következő megoldást kell alkalmaznunk: Vegyük az alábbi, kliens által transzformált lekérdezést:

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
SELECT ?title ?author
WHERE     {
               ?x dc:title ?title ?x dc:author ?author
               FILTER INDEX_STR(?author,
               "00000000010000010001101000100010")
               FILTER INDEX_STR(?title,LIKE
               "00000000010000000000000000000010" && NOT LIKE
               "00001010010110010000001110000010" )
          }
```

A rendszer az uri_values táblából kiolvassa, hogy a dc prefixummal megadott névtér author és title elemeihez milyen kapcsolatleíró táblák tartoznak (azaz mintaillesztést végez

a gráfon). Ezt követően az adott táblák values oszlopából azokat a (rejtjelezett) sorokat adja vissza, ahol a kapcsoló index értéke megfelelő. Részleges egyezés (LIKE) vizsgálatára a már említett **check_index** tárolt eljárást definiáltam, ami egy integer tömbben megkapja, az index hányadik karaktereinek kell 1-nek lenni. Kizáró egyezés (NOT LIKE) esetén a 0 értékű indexeket ellenőrizzük a **check_index_not** eljárással, teljes egyezésnél pedig **check_index_all** azt is megnézi, hogy a kapott sorszámú indexek 1, a többi 0 legyen.

Numerikus adatok szűrése is hasonló módon történik, de itt nincs szükség tárolt eljárásra. Paraméter-érték relációk esetén olyan rekordokat adunk vissza, amikhez a kapcsos zárójelek közötti indexek valamelyike tartozik. Paraméter-paraméter relációknál a két kapcsolat tábláját (esetünkben: price_[id1], oprice_[id2]) szűrjük az indexpárok megfelelő felével, és join-oljuk a subj attribútum egyezésére.

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
SELECT ?price ?oprice
WHERE
{
     ?x dc:price ?price ?y dc:oldprice ?oprice
     FILTER   (   INDEX_INT(?price{508,   7313,   7500}   &&
     ?oprice:?price{4604:508,      4604:7313,      4604:7500,
     4604:8049, 4604:4501, 4604:6678, 4604:11394, 4604:11770,
     4604:9862, 4604:5720, 4604:1483, 1154:10048, 4604:10048,
     5265:10048,    332:10048,    6001:10048,    5525:10048,
     2582:10048} ))
  }
```

Mind számok, mind szövegek rejtjelezett beszúrása, törlése és WHERE feltételben való illesztésének módja triviálisan következik a fentiekből, hiszen a DEC_STR, DEC_INT és DEC_FLOAT Chispa beépített eljárások (lásd 3.1.1.7 és 3.1.1.9) egyaránt egy titkosított értéket és egy indexet küldenek a szervernek.

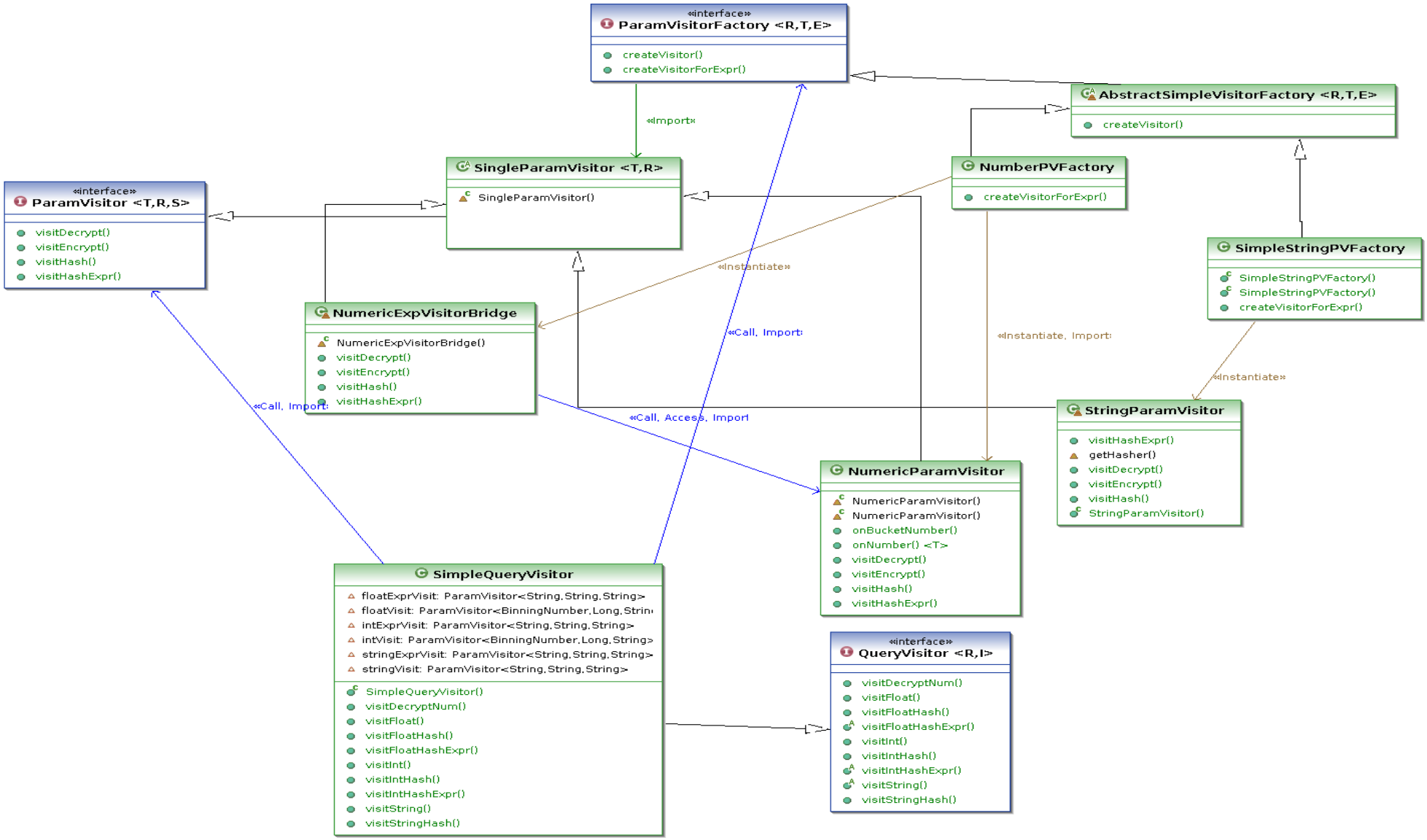
3.2. A kliens oldali logika

3.2.1. Az interpreter modul

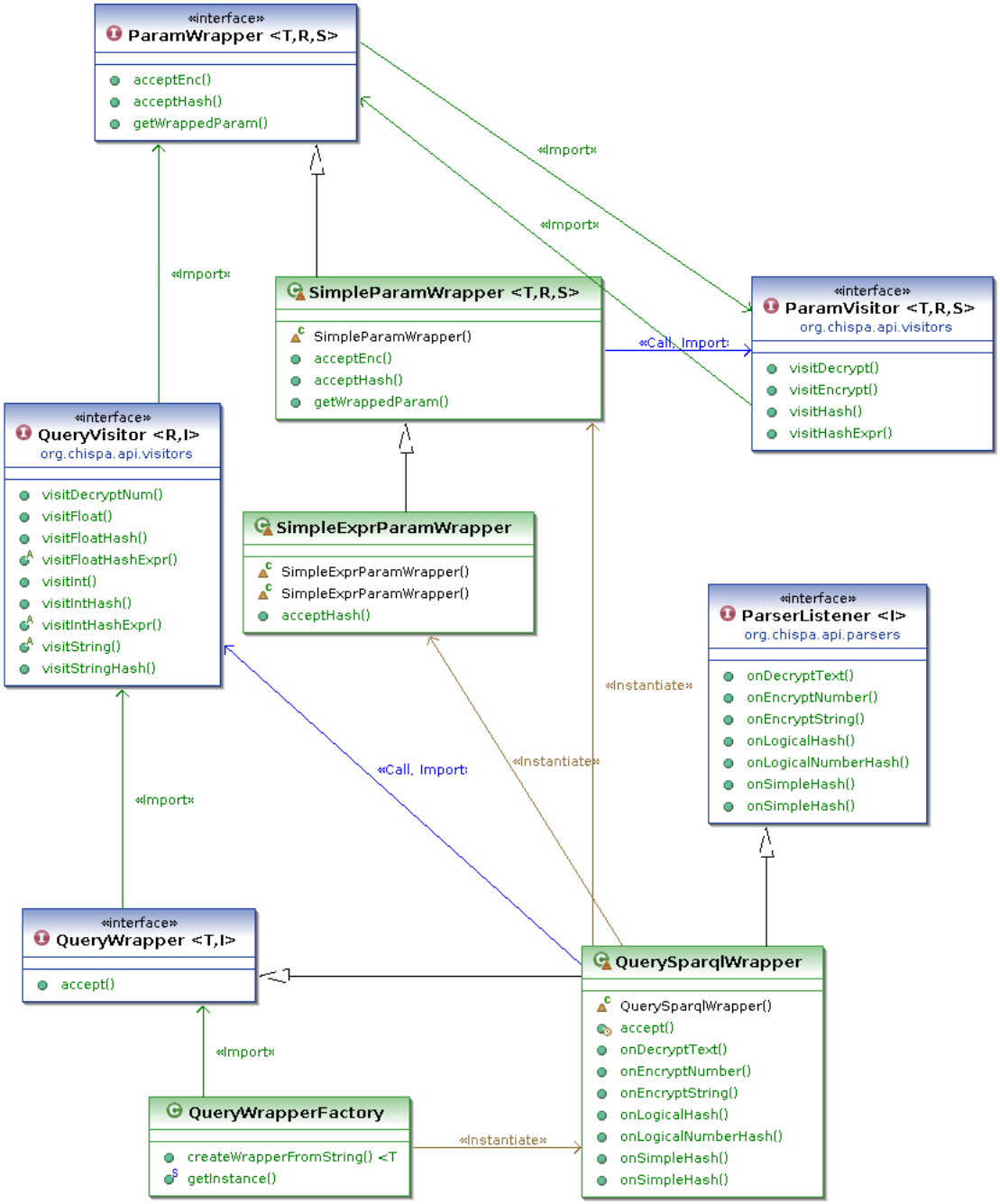
A kliens oldali logika teljes egészében Java nyelven íródott. A biztonságos lekérdezések értelmezéséhez és transzformációjához visitor mintát alkalmazok, azaz az adatstruktúrát (lekérdezés) elválasztom a rajta végzett műveletektől. A 6. ábra (lásd a következő oldalon) mutatja be az **org.chispa.api.visitors** package osztályait és a köztük lévő kapcsolatokat.

A **QueryVisitor** interfészt megvalósító osztályok tetszőleges lekérdezést képesek feldolgozni. Olyan metódusokkal rendelkezik, amelyek az egyes típusú paraméterek rejtjelezésére (pl. **visitInt()**), paraméterek és kifejezések indexelésére (pl. **visitStringHash()**, **visitFloatHashExpr()**), illetve dekódolására (pl. **visitDecryptNum()**) használhatók. Az egyes metódusok visszatérnek a transzformált paraméterrel, de annak behelyettesítése az új lekérdezésbe nem az ő feladatuk. A QueryVisitor implementációja a **SimpleQueryVisitor**, amely a konkrét operációkat továbbdelegálja a megfelelő **ParamVisitor** interfészű objektumoknak. Ezekre a konstruktorában átadott **ParamVisitorFactory** objektumoktól kap referenciát. Minden paramétertípushoz (szám, szöveg, numerikus kifejezés) külön ParamVisitor-t kér, ezeknek példányosításáról az **AbstractSimpleVisitorFactory** absztrakt ősosztály leszármazottai (**NumberPVFactory**, **SimpleStringPVFactory**) gondoskodnak.

Szövegeket a **StringParamVisitor** kezel; a rejtjelezéshez szükséges információkat (kulcs, alkalmazott algoritmus) a factory-től kap, és az indexelésért felelős objektumot (lásd: 9. ábra) is az injektálja bele. Az egyszerű numerikus értékeket a **NumericParamVisitor** kezeli; logikai relációs kifejezések transzformációjához a **NumericExprVisitorBridge** csomagoló osztályon keresztül hívjuk.

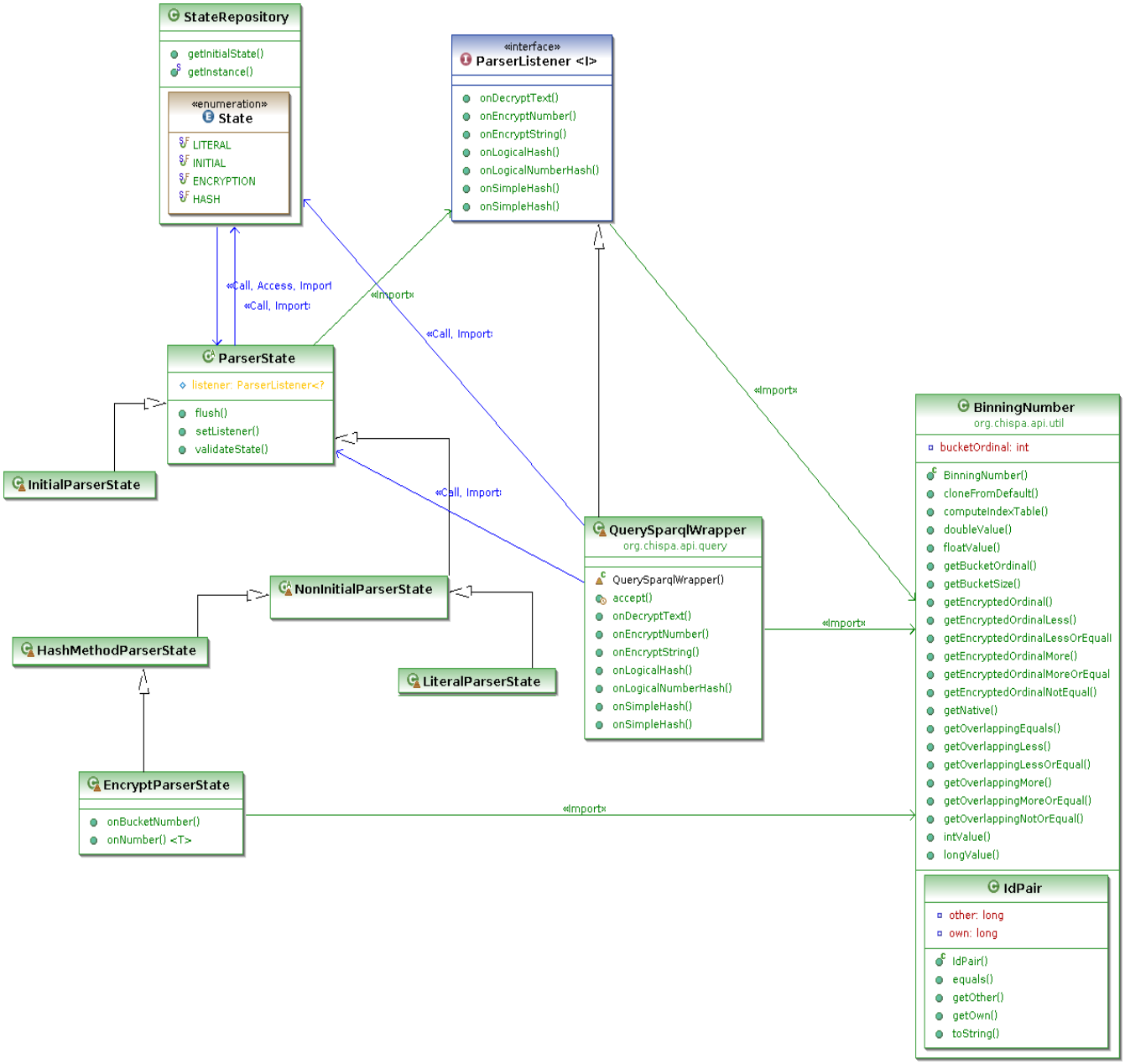


7. ábra: A visitor részkomponens osztálydiagramja



8. ábra: A struktúra-leíró részkomponens osztálydiagramja

A fenti ábrán a lekérdezések, paraméterek és kifejezések adatstruktúráját csomagoló osztályokat („visitable”) látjuk. A **QueryWrapper** interfészű objektumok egy teljes lekérdezést csomagolnak be, tehát képesek fogadni QueryVisitor-okat. A **ParamWrapper** típusúak paramétereket (**SimpleParamWrapper**) és kifejezéseket (**SimpleExprParamWrapper**) kezelnek, és ParamVisitor-okat fogadnak. A **QueryWrapperFactory** példányosítja a **QuerySparqlWrapper** alapértelmezett lekérdezés-csomagolót.



9. ábra: A feldolgozó részkomponens osztálydiagramja

Láthatjuk, hogy a QuerySparqlWrapper implementálja a **ParserListener** interfészt (lásd: 8. ábra). Erre azért van szükség, mert a csomagolóosztályok egy állapotgép segítségével járják be a lekérdezést. A **StateRepository** határozza meg az állapotok egymásutániságát, és a flyweight mintának megfelelően menedzseli, hogy egy lekérdezés kontextusában ne keletkezzenek felesleges példányok az állapotokról. Az egyes állapotok a **ParserState** absztrakt ősosztályból származnak. Mindegyik állapot tartalmaz egy szövegbuffert, amiben a lekérdezés eddig feldolgozott elemei szerepelnek. Ha egy állapot helyben marad, rekurzió történik és tovább olvas, állapotváltásnál pedig elkéri a következő állapot referenciáját a StateRepository-tól. Ha olyan állapotba kerülnek, ahol egy visitor-

nak át kell adni a paramétereket (pl. rejtjelezésre, indexelésre), ezt jelzik a ParserListener objektumnak. A feldolgozás végén a QuerySparqlWrapper a **flush()** meghívásával kapja vissza a transzformált lekérdezést.

Kezdetben, némi normalizálás után (lásd: 3.2.2. fejezet) az állapotgép megkapja a lekérdezést, és elkezdi szavanként feldolgozni (**InitialParserState**). Ha lezáratlan literállal találkozik (ez olyan idézőjelek közötti szöveg lehet, amiben whitespace karakter is van), átmegy a **LiteralParserState** állapotba. Itt addig marad, amíg a literál le nem záródik (az itt talált nyelvi elemekkel – pl. ENC_INT, HASH_STR – természetesen nem foglalkozik, csak hozzáfűzi őket a bufferhez), majd visszalép az előző állapotba. Indexelő vagy rejtjelező beépített függvény nevére rendre a **HashMethodParserState** és **EncryptParserState** állapotba lép, ami szintaktikailag ellenőrzi a kifejezést, összegyűjti a paramétereket, majd jelez a listener-nek és hozzáfűzi a bufferhez a tőle visszakapott, rejtjelezett vagy indexelt szöveget. A rejtjelező állapot által kezelt **BinningNumber** a java.lang.Number osztályból származik, és egy domain-paraméterekkel ellátott numerikus változót reprezentál. Metódusai a 2.1.2. részben definiált egyenlőtlenségek kiszámítását teszik lehetővé, a **getEncryptedOrdinalLess()** a hívott objektumnál kisebb értéket tároló ládák indexét adja vissza a megfelelő titkos kulccsal hívva, míg a **getOverLappingMore()** egy másik BinningNumber-t is kap paraméterként, és megmondja, milyen index-párokra lehet az átadott objektum értéke nagyobb a hívotténál. Utóbbi esetben természetesen a két paraméter indexelése számít, nem az, hogy éppen milyen értékkel vannak inicializálva. Tehát ha egy „?price < ?oprice” relációval dolgozunk, teljesen mindegy, milyen értéket tárolnak az egyes paraméterekhez tartozó BinningNumber példányok, a getOverLapping* formájú metódusok azt adják vissza, mely indexeknél **lehet** igaz az adott paramétertípusokra az egyenlőtlenség.

Már említettem, hogy az eredeti szövegeket indexelő algoritmus könnyen támadható. Emlékeztetőül: s_1 egy $c_1, c_2, \dots, c_n$ karakterfüzér (nyílt szövegű adat), s_2 egy $b_1, b_2, \dots, b_{m-1}$ bitsorozat (index), $n < m$. H hash algoritmus, ami s_1 minden c_i, c_{i+1} karakterpárját leképezi egy $0 \leq k \leq m-1$ egészre. Az index bitjeinek értéke:

$$b_i = 1, \text{ha } H(c_j, c_{j+1}) = i \text{ valamely } 1 \leq j \leq n-1 \text{ egészre.}$$

$$\text{Egyébként } b_i = 0$$

Mint látjuk, a hash algoritmus az eredeti karakterpárokon dolgozik, és mivel az index bithosszának viszonylag rövidnek kell lennie, az eredeti karakterpárok könnyen visszafejthetők az indexből. Szükséges ezért egy szimmetrikus rejtjelezést vagy kulcsolt hash függ-

vényt definiálni az adattulajdonosok és a klienseik által megosztott – a rejtjelezésre használttól eltérő - kulccsal, amellyel elrejtjük a karakterpárjainkat. Implementációm kulcsolt hash-t alkalmaz, mert így a kulcs megszerzésével a támadónak még fel kell építenie egy hash táblát, nem tudja rögtön visszanyerni az indexekből a nyílt szöveget, mint szimmetrikus titkosítás esetén. Formálisan:

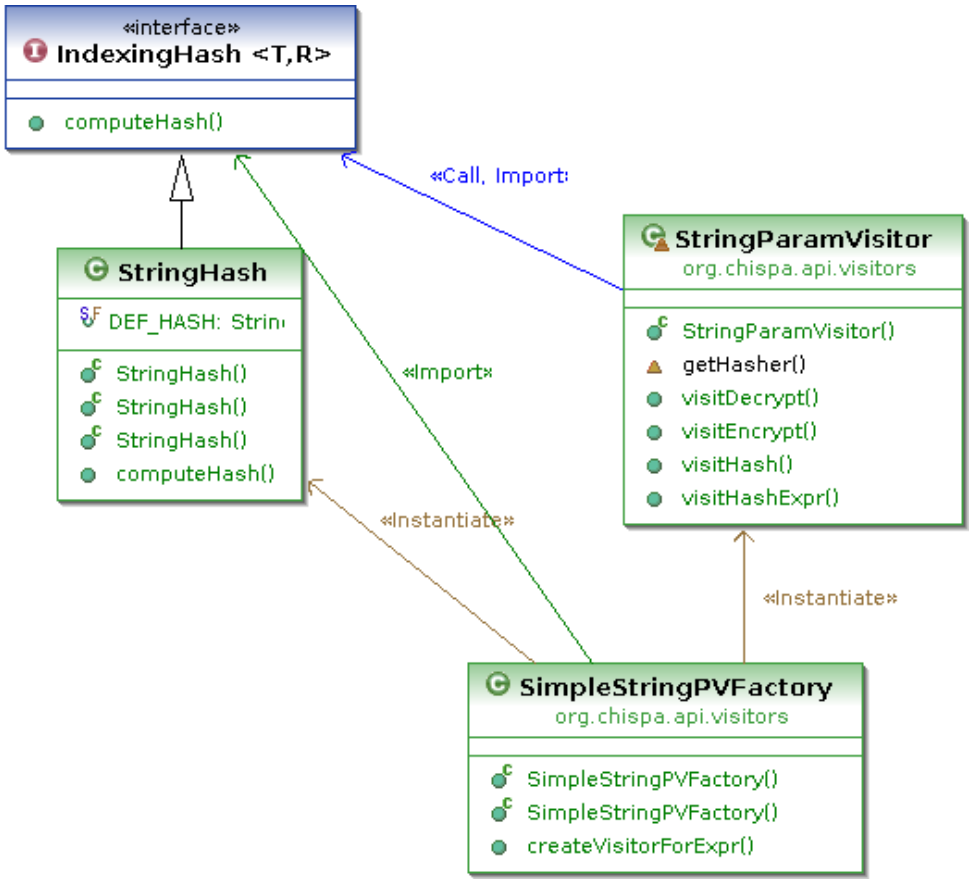
K kulcsolt hash, amelynek bemenete két karakter (c_1, c_2), kimenete string (s_e).

A módosított H' algoritmussal az index bitjeinek értéke a következő:

$b_i = 1$, ha $H'\big(K(c_j, c_{j+1})\big) = i$ valamely $1 \leq j \leq n-1$ egészre.

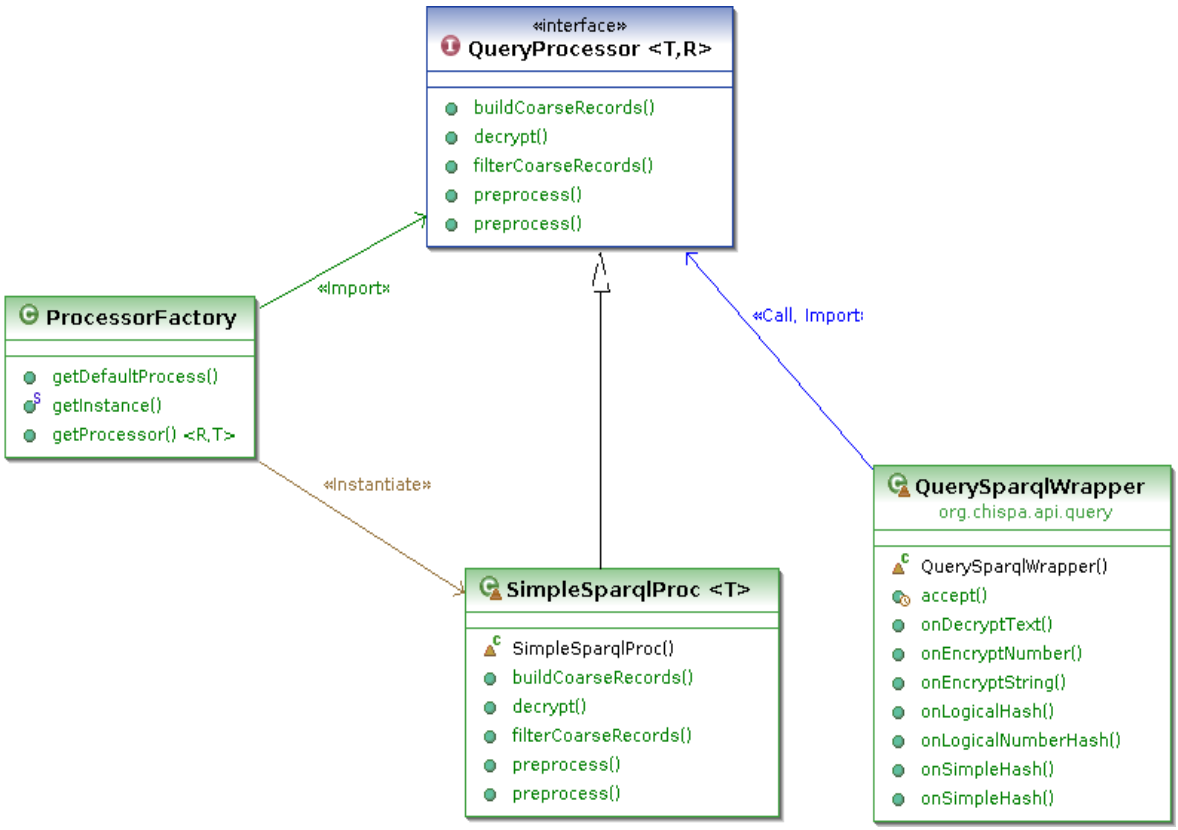
Egyébként $b_i = 0$

A fent látható algoritmust a StringParamVisitor példányoktól kapott kulccsal az **IndexingHash** interfészű objektumok (pl. **StringHash**) végzik el, ahogy az a 9. ábrán is látszik.



10. ábra: A szövegeket kezelő részkomponens osztálydiagramja

3.2.2. A processzor modul



11. ábra: A processzor modul osztálydiagramja

A lekérdezés előző fejezetben leírt transzformációja előtt és a lekérdezés után szükséges bizonyos műveleteket elvégezni ahhoz, hogy a felhasználó a kívánt értékeket kapja vissza. Ahogy a fenti ábrán is látható, a **ProcessorFactory** felel a QuerySparqlWrapper csomagoló osztály által hívott **QueryProcessor** interfészű feldolgozó objektum (**SimpleSparqlProc**) példányosításáért. Ez az osztály felel a lekérdezés elő-, és a kapott rekordok utófeldolgozásáért.

A **preprocess()** operáció során reguláris kifejezések segítségével normalizáljuk a lekérdezést, hogy az állapotgép által feldolgozható legyen. Első lépésként kiszedjük és eltároljuk a literálokat, hogy a normalizálás őket ne érintse, és egy helyőrző tokent teszünk a helyükre. Ezt követően eltüntetjük a felesleges whitespace karaktereket, összegyűjtjük és eltároljuk a biztonságosan kezelendő paramétereket és a névtér-prefix összerendeléseket, majd visszahelyezzük a literálokat és visszatérünk az előkészített lekérdezéssel.

A visszakapott rekordok dekódolását a **decrypt()** delegálja a megfelelő visitor-nak, majd a **buildCoarseRecords()** memóriában tárolt gráfot épít belőlük (itt még lehetnek hamis pozitív értékek). A **filterCoarseRecords()** az eredeti lekérdezésből nyílt szövegen

futtathatót generál, majd szűri vele a szervertől kapott értékeket. A leképezések a következő szabályok szerint alakulnak:

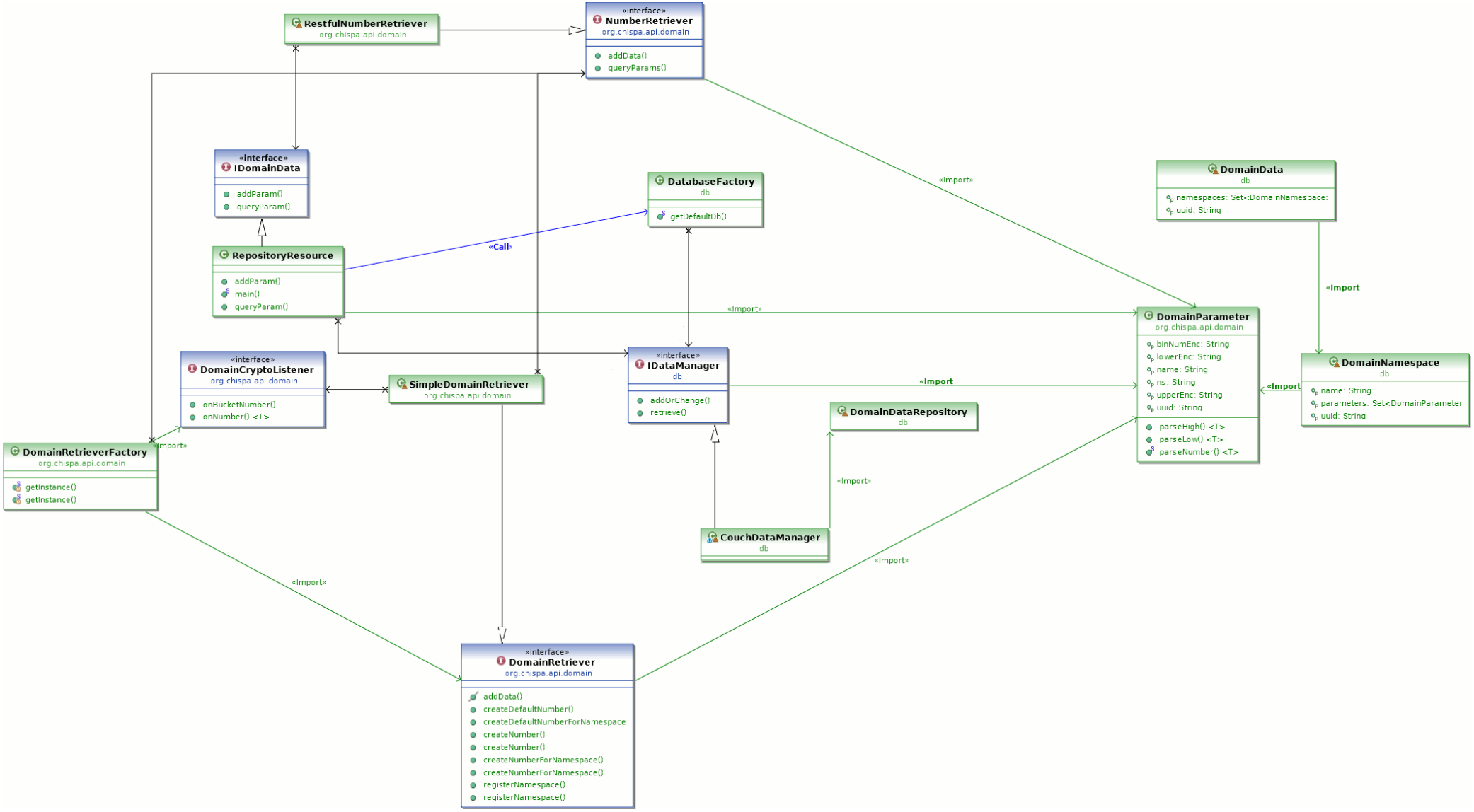
```
HASH_STR(?author, "John Smith") => regex(?author, "John Smith")
HASH_STR(?author, LIKE "Smith") => regex(?author, ".*Smith.*")
HASH_STR(?author, NOT LIKE "John") => regex(?author, "^((?!John).)*")
```

Numerikus értékekre a transzformáció triviális, mindössze a HASH_INT vagy HASH_FLOAT kifejezést kell levágni.

3.3. A szerver oldali logika

3.3.1. A domain-paramétereket kezelő modul

Ahhoz, hogy az adatok tulajdonosai publikálni tudják numerikus paramétereik domain-értékeit (ládaszám, alsó korlát, felső korlát), a felhasználók pedig lekérdezhessék ezeket, a cloud szervernek egy szabványos interfészt kell nyújtania ezen információk kezelésére. Mivel ezek ismeretében a cloud szolgáltató könnyen támadást tud intézni a szemantikus adatok bizalmassága ellen, szükséges a paramétereket rejtjelezve elküldeni és tárolni, valamint megosztani egy kulcsot az adat gazdája és kliensei között. Hogy a paramétereket kezelő modul a szemantikus adattároló konkrét implementációjától független legyen, külön adatelérési réteget és Java nyelvű alkalmazást definiáltam hozzá, melynek UML2 osztálydiagramja a következő oldalon szerepel. Igyekeztem az alkalmazást bővíthetővé és újrakonfigurálhatóvá tenni, azaz az egyes részfeladatokat ellátó komponensek elrejtik a konkrét implementációt, hogy könnyen lecserélhessük bármelyik, például az adatelérési vagy távoli eljáráshívásért felelős réteget.



12. ábra: A paraméterkezelő komponens osztálydiagramja

Numerikus adatra hivatkozó lekérdezés elemzése során a kliens először a **DomainRetrieverFactory**-n keresztül kér egy **DomainRetriever** interfészű objektumon, amely új domain információk (**DomainParameter**) hozzáadására és lekérdezésükre használható. Ezen keresztülregisztrálja a prefixum-névtér összerendeléseket (hiszen az egyes paraméterek nevében a rövid formában hivatkozik a névtérre), és ha numerikus értékű paraméterrel találkozik, **BinningNumber** csomagoló objektumot (az ábrán nem szerepel) gyártat belőle a megfelelő „create*" kezdetű metódus meghívásával. A DomainRetrieveFactory-nek opcionálisan átadhatunk egy DomainCryptoListener interfészt megvalósító objektumot, aminek a DomainRetriever majd delegálja a visszakapott paraméterek dekódolásának feladatát az **onBucketNumber(String, T extends Number)** és **onNumber(String)** metódusok meghívásán keresztül. Amennyiben ilyet nem adunk át neki, feltételezzük, hogy a domain paraméterek plain text formában kerültek eltárolásra egy, a cloud szervertől eltérő, megbízható félnél, és tőle kérdezünk le.

A factory által visszaadott konkrét implementáció egy **SimpleDomainRetriever**, amely elvégzi a kapott paraméterek transzformációját, a DomainCryptoListener példányt visszahívja, és (amennyiben a konfigurációs property fájlban engedélyeztük) a már lekérdezett domain információkat gyorsítótárba helyezi, hogy kevesebb legyen a hálózati kommunikáció. A konkrét lekérdezést viszont egy **NumberRetriever** objektumon keresztül végzi, amelyet a factory injektál bele. Ez a jelenlegi implementációban egy **RestfulNumberRetriever** példány, amely a nyílt forráskódú Restlet keretrendszer [22] segítségével, JSON kérésekkel hívja a domain-paraméterek tárolóját (természetesen ez igény szerint lecserélhető más, pl. CORBA megoldásra a SimpleDomainRetriever módosítása nélkül).

A fent említett REST paradigmát követő kliens egy **IDomainData** interfészű proxy-n keresztül hívja a tárolót, melynek URL-jét a property fájlból olvassa ki. A cloud szolgáltató egy ilyen interfészt megvalósító **RepositoryResource** nevű Restlet szervert futtat, ami a hívásokat kezeli. A domain adatok konkrét tárolásáról ennek az osztálynak nincs információja, hanem a **DatabaseFactory**-től kapott **IDataManager** implementációhoz fordul keresés vagy beszúrás esetén. Mintaimplementációm dokumentum-orientált CouchDB [23] adatbázist használ (**CouchDataManager**); ez JSON formátumban tárolja az információkat, ami a relációsnál jobban illeszkedik a domain-paraméterek tárolásának feladatára. Az adatbázispéldányt a **DomainDataRepository**-n keresztül, Ektorp API [24] segítségével érjük el. Az itt tárolt JSON adatok formátuma a következő: Minden adattulajdonos rendel-

kezik egy egyéni azonosítóval (UUID), amit a kliensek is ismernek, ez a dokumentumok gyökere (**DomainData**). Minden egyéni azonosítóhoz rendelünk nulla vagy több névteret (**DomainNamespace,** a DomainData nyilvántart róluk egy referenciát); ezek teljesen megfeleltethetők a SPARQL operációk során alkalmazott névterekkel. Minden névtér tartalmazhat paramétereket, amelyeket a nevük jellemez (pl. ha beszúrtunk http://purl.org/dc/elements/1.1/ névtérrel „price" címkéjű élet a gráfba, amely rejtjelezett numerikus értékre mutat, ebben a névtérben biztosan lesz pontosan egy ilyen nevű paraméter). A paraméterekhez megadjuk az alsó- és felső korlátot, valamint a ládaszámot (mindezt rejtjelezve), amit a kliens lekérdezhet, és segítségével elvégezheti az indexelést.

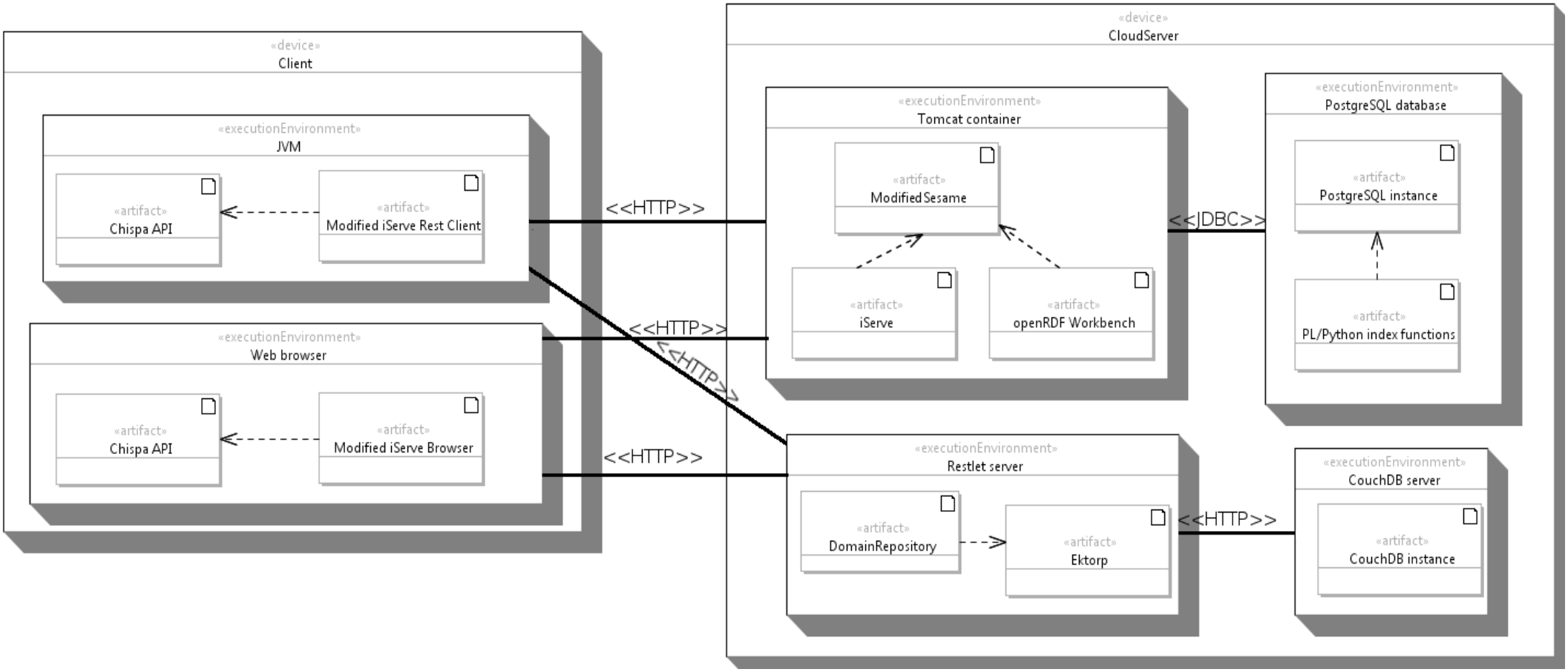
3.4. Az új komponensek integrációja

A teljes, integrált rendszert a következő oldalon található UML2 deployment diagram szemlélteti.

Az iServe két módon képes SPARQL lekérdezést végrehajtani. Egyrészt a GWT-alapú iServe Browser webes felületen keresztül, másrészt egy különálló REST kliens segítségével (egyszerű Java alkalmazás). A kliensgép tehát ezek valamelyikén keresztül lép kapcsolatba a cloud szerverrel. A rejtjelezett adatkezeléshez szükséges, hogy mindkét alkalmazás használja a Chispa API-t.

A szerver egy szervlet konténerben futtatja az iServe alkalmazást, azt általa hívott, módosított (lásd: 3.4.1.) Sesame tárolót, valamint utóbbi menedzsment felületét (openRDF Workbench). Az adatelérés JDBC driver segítségével történik; a PostgreSQL adatbázis tartalmazza azokat a PL/Python eljárásokat, amiket az indexelés során használ a rendszer (lásd: 3.1.2. fejezet).

A numerikus értékek domain-paramétereinek (lásd: 3.3.1. rész) lekérdezése is történhet mindkét kliens oldali alkalmazásból. Az ezt kiszolgáló komponenst (az ábrán **DomainRepository** névvel szerepel) a Restlet keretrendszer beágyazott HTTP szervere futtatja. A DomainRepository alkalmazás az Ektorp adatkezelő API-n keresztül, szintén HTTP protokollon éri el a CouchDB szervert, ahol a domain-adatok vannak. (Fontos megjegyezni, hogy a PostgreSQL adatbázis, a Sesame szervletek, az iServe, a CouchDB és a DomainRepository nem szükséges, hogy ugyanazon a fizikai hoszton vagy virtuális gépen fussanak, de természetesen a property fájlokban egyértelműen meg kell adni, milyen címen érik el egymást).



13. ábra: A teljes rendszer deployment diagramja

3.4.1. Integráció a Sesame tárolóval

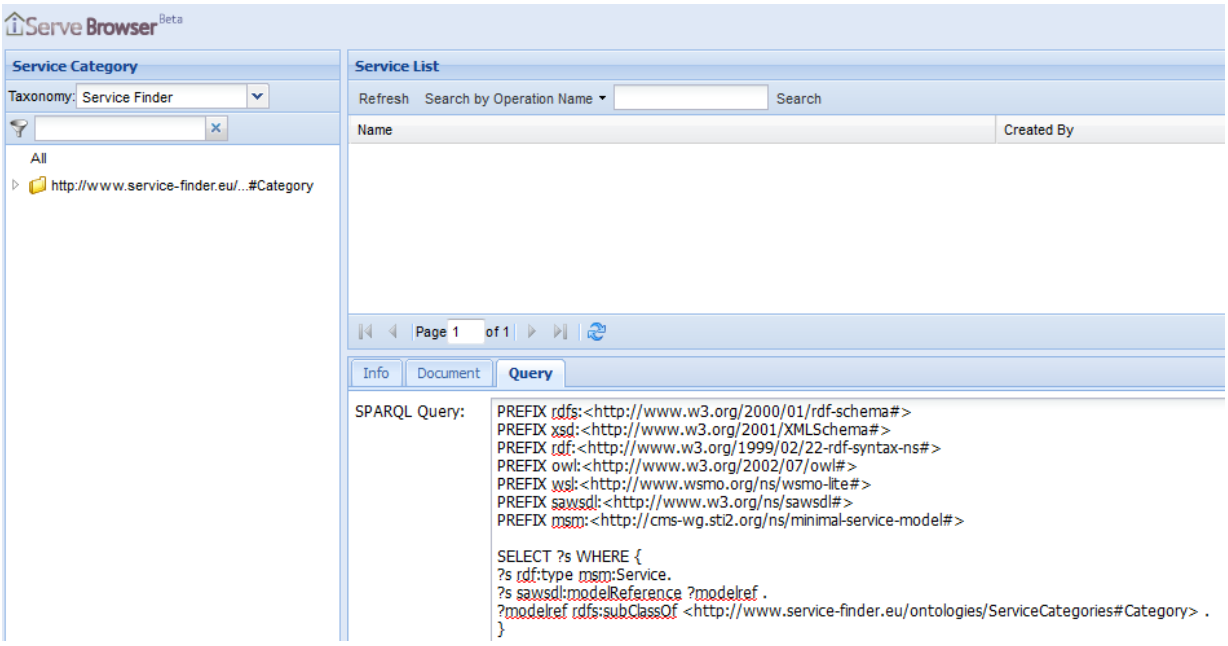
Az indexre való keresést segítő tárolt eljárások definícióján kívül a Sesame tároló szemantikus-relációs leképező logikáját is át kell alakítani ahhoz, hogy a rejtjelezett lekérdezések működjenek, hiszen a 3.1.1. fejezetben bemutatott új szerver oldali SPARQL függvényeket és nyelvi elemeket nem támogatja.

A Sesame-ban a relációs adatbázis kezelése a Sail API feladata. A SPARQL kifejezéseket először kiértékeli, egy lekérdező algebra-modellt generál belőlük, majd ebből készít egy vagy több SQL utasítást. Az SQL lekérdezések összeállítása visitor minta segítségével történik. Az integrációhoz be kell vezetni a lekérdező algebra modelljébe a numerikus és szöveges indexeket és a rejtjelezett paramétereket reprezentáló osztályokat, valamint az SQL lekérdezéseket összeállító komponensben a relációs lekérdezés modelljének új elemeit (tárolt eljárások). Szükséges volt még az SQL modellt kiértékelő komponensben építő (builder) osztályt definiálni az új elemeket tartalmazó lekérdezések összeállításához. Az integráció során feltételezzük, hogy a Sesame-ot úgy konfiguráltuk, hogy a 3.1.2. részben látható módon külön táblába szúrja be az egyes predikátumokat (ez az alapbeállítás).

3.4.2. Integráció az iServe alkalmazással

A sikeres integráció előfeltétele, hogy az iServe kapcsolatba léphessen egy olyan szemantikus tárolóval, amely képes rejtjelezett adatokon műveleteket végezni. Ha ez a feltétel biztosítva van (pl. az előző fejezetben említett Sesame integráció útján), az interpreter modul transzformációit futtatni kell a felhasználó által megadott SPARQL lekérdezésen, mielőtt azt a szerver megkapja.

Mint már említettem, az iServe egy GWT és egy egyszerű REST alapú klienst kínál. A rejtjelezett adatok kezeléséhez tehát első lépésben a GWT widget által beolvasott, illetve a REST kliens paramétereként átadott lekérdezést át kell alakítani köztes formátumra, azaz a Chispa API-val el kell végezni a paraméterek rejtjelezését és az indexek kiszámítását (lásd 3.1.1. rész). A transzformált adatokat a módosított Sesame tároló helyesen lekezeli, és visszaadja a relációs adatbázisból kiolvasott, illeszthető rekordokat, amik között lehet hamis pozitív is. Az eredményeket tehát dekódolni kell, majd el kell végezni rajtuk egy finomító lekérdezést, mielőtt azt a felhasználónak megjelenítjük (GWT kliens) vagy visszatérünk velük string formában (REST kliens).



14. ábra: Az iServe Browser lekérdező felülete

A SPARQL lekérdezések szemantikus webalkalmazások kontextusában általában valamilyen WSMO-Lite paraméterre keresnek rá vagy módosítják azt (lásd: 2.2.3. rész). Tipikus felhasználási eset lehet, ha a webalkalmazás publikálója valamilyen zárt felhasználói kör számára elérhető szolgáltatást publikál, és nem szeretné, ha annak kontextuális információt leíró attribútumai kiszivárognának. Ekkor nem funkcionális WSMO-Lite szemantikát kell alkalmaznia rejtjelezéssel kombinálva. Mivel a nem funkcionális szemantika gyakran ír le QoS metrikákat, itt nagy szerepet kap a numerikus értékek rejtjelezése. Az is elképzelhető, hogy az alkalmazás funkcionalitását, vagy működési részleteit szeretné elrejteni a szolgáltató szerver elől; ebben az esetben a viselkedési szemantikát kell rejtjelezetten kezelni (jellemzően szöveg-kódolással). Lássunk két példát numerikus értékekkel leírt nem funkcionális szemantika biztonságos keresésére és viselkedési jellemzők (prekondíciók) rejtjelezett beszúrására!

Az első példában a nem funkcionális paraméterek leírására adott RDF-definíciót látjuk a 2.2.3. fejezetből. A biztonságos lekérdezés pedig visszaadja az összes olyan (rejtjelezetten tárolt) kurzusnevet, ahol az adott előadás (mint szolgáltatás) oktatója jó OHV értékelést kapott.

```
iit:OhvValue rdfs:subClassOf wsl:NonFunctionalParameter .
iit:CORBACourseValue a iit:OhvValue;
iit:teacherGrade "5"^^iit:ohvGrade;
iit:subjectGrade "4"^^iit:ohvGrade .
```

```
PREFIX iit:  <http://iit.bme.hu/>
SELECT SECURE_STR:?cname
WHERE
{
     ?x iit:courseName ?cname
     ?x iit:OhValue ?ohv
     ?ohv iit:teacherGrade ?tgrade
     FILTER HASH_INT(?tgrade >= 4 )
}
```

A második példában a dolgozatban ismertetett szövegrejtjelező algoritmussal szúrunk be viselkedési szemantikát.

```
PREFIX iit:  <http://iit.bme.hu/>

INSERT DATA INTO <http://teaching/courses>
{
     <http://teaching/course42> iit:CourseStartPrecondition
     ENC_STR("?coursedata[iit#creditNumber hasValue ?cr]
               memberOf iit#CourseData and ?cr > 0")
}
```

4. AZ IMPLEMENTÁCIÓ ÉRTÉKELÉSE

4.1. A tesztkörnyezet bemutatása

A rendszer tesztelésére és profilozására használt számítógép paramétereit az alábbi táblázat mutatja:

Komponens megnevezése	Komponens leírása
Operációs rendszer	Ubuntu 10.10 64bit
CPU	AMD Phenom II X4 955
CPU magok száma	4
RAM	8 GB (2x 4GB) 1333 MHz DDR3
Szervlet konténer	Apache Tomcat 6.0.28
Adatbázis	PostgreSQL 9.0
Szemantikus tároló	Sesame 2.5.1 (módosított)
Szemantikus web service repository	iServe 1.0 (kliens módosított)

4.2. Verifikáció és validáció során alkalmazott eljárások

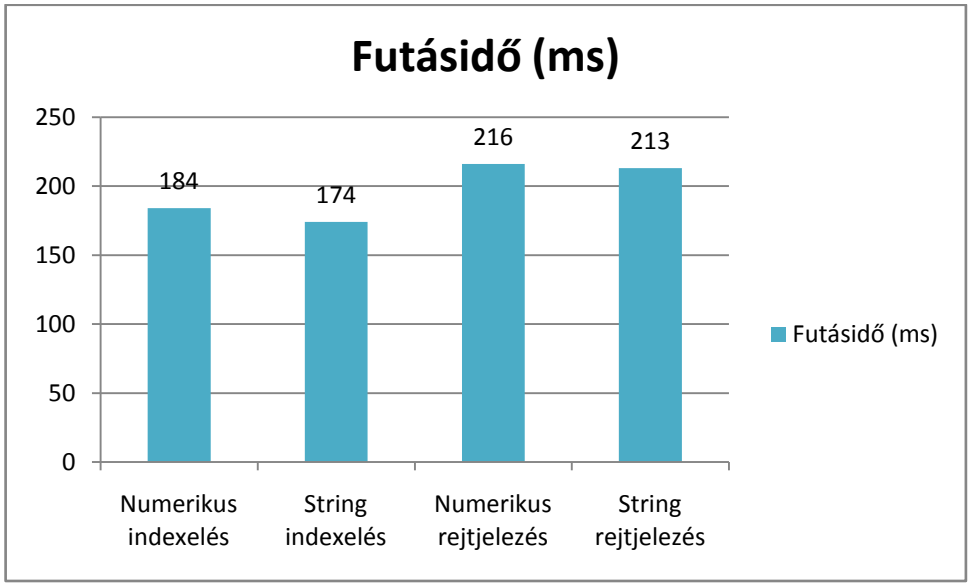
A rendszer fejlesztése során test-driven development módszerekkel dolgoztam; az egyes új funkciókat megvalósító komponensek integrálását mindig egységtesztek előzték meg. A rendszer helyességi vizsgálatához JUnit 4.8.1 keretrendszert, teljesítményének profilozására (CPU terhelés, memóriafogyasztás, processzoridő-allokáció) pedig a VisualVM 1.3.3. alkalmazást használtam.

4.3. A rendszer érettségének és teljesítményének vizsgálata

Egy nem megbízható környezetben futó alkalmazásnál elengedhetetlen a biztonságos működés, ami természetesen a legtöbb esetben teljesítménygyengüléssel jár, hiszen a kriptográfiai műveletek számításigényesek. Valóban, a rendszer tesztelése közben észleltem, hogy a teljes transzformációs overhead mintegy 20-25 %-át a rejtjelezés okozza. Amennyiben ez a teljesítményromlás a felhasználó részéről is érezhető, a rendszer kiszervezéséből adódó előnyök jelentősen csökkennek. A profilozás során azt tapasztaltam, hogy az általam írt rendszer gyakorlati használhatósága nagyban függ a lekérdezés pontosságától,

kezelt adattömeg méretétől, (numerikus értékek esetén) a használt kosarazó algoritmus és paraméter-felosztás milyenségétől, valamint attól, mekkora hasonlóság van az egyes értékek között (tehát mennyi rekord kerül átlagosan egy ládába). Ezen változók közvetlenül befolyásolják a hálózati overhead nagyságát; noha a jelen dolgozatban szereplő use case (biztonságos webalkalmazás tároló) viszonylag kevés, de egymástól nagyban eltérő adattal dolgozik, más alkalmazási területeken (pl. döntéstámogató rendszerek) a hamis pozitív rekordok aránya igen magas lehet (szövegek titkosításánál, 32 bites indexet használva akár az összes adat 20 százaléka! [3]). A hamis pozitív rekordok aránya természetesen a futási időt is növeli, hiszen a második, kliens oldali szűrőlekérdezés előtt be kell szúrni őket a lokális gráfba. Megállapíthatjuk tehát, hogy biztonságos szemantikus tároló megoldásom gyakorlati hasznosságának feltétele szöveges értékek esetén a kezelhető méretű adattömeg, numerikus értékek esetén pedig a jól megválasztott index-felosztás. Utóbbi esetben sokat segít az is, ha az érték-domain felosztása jól ismert – esetleg valamilyen létező adattömeg cloud környezetbe való migrációja esetén, ahol az egyes attribútumok átlagát, mediánját és egyéb származtatott értékeit kiszámolhatjuk és heurisztikát adhatunk a kosarazás módjára.

Adattípustól független, viszonylag kismértékű számítási többletköltségek is jelentkeznek, ezeket a biztonságos lekérdezés jellegéből adódó kriptográfiai és query-feldolgozó műveletek okozzák. Az alábbi táblázat bemutatja ezek mértékét (hálózaton egyszer lekérdezett és dekódolt, majd lokálisan cache-elt változó-kosár összerendeléseket feltételezve):



15. ábra: Transzformációk futásideje

A teljesítmény mellett fontos a rendszert adatbiztonsági szemszögből is vizsgálni. A kulcsolt hash bevezetése csökkentette ugyan a szöveges rekordok támadhatóságát, de na-

gyobb indexméret, azaz kevés ütközés – és alacsony hálózati overhead - esetén a karakterpárokat ugyan nem lehet úgy visszafejteni, mint az algoritmus eredeti változatában, de statisztikai módszerekkel (az adott nyelv gyakori karakterpárjainak megkeresésével) támadható a rendszer. Ennek kivédésére a 4.4. részben határozok meg egy, a továbbfejlesztés során implementálandó algoritmust.

A numerikus értékeket kezelő algoritmus véletlenszerű ládaindexelés esetén nem szivárogtat adatok a rekordok sorrendjéről, de eloszlásukról igen. Ez olyan paraméterek esetében jelenthet gondot, ahol az eloszlás az adott rekord típusából, szemantikus jelentéstartalmából kikövetkeztethető. Ilyen adat például a (webalkalmazás-leírások kontextusában ritkán előkerülő) életkor.

Összességében megállapíthatjuk, hogy a jelen dolgozatban definiált prototípus kisebb változtatásokkal alapot adhat egy nem kritikus – pl. orvosi, pénzügyi -, de bizalmasan kezelendő szemantikus adatok biztosításával kapcsolatos rendszer számára. A dolgozatban szereplő felhasználási terület, a szemantikus webalkalmazás tárolók rejtjelezett adatainak felderítése számára a rendszer kielégítő biztonságot nyújt.

4.4. Továbbfejlesztési lehetőségek

Az alábbi táblázat röviden összefoglalja azokat a teendőket és továbbfejlesztési lehetőségeket, amiknek elvégzése a téma folytatása során érdemes vagy hasznos lehet.

PROBLÉMA	TÍPUS	FONTOSSÁG
Minimal Service Model rejtjelezett tárolása	Szolgáltatás	Közepes
Aritmetikai operációk kiértékelése	Fejlesztés	Közepes
Szöveges rekordok magasabb biztonsága	Fejlesztés	Magas
Szerep alapú hozzáférésvezérlés domain értékekre	Szolgáltatás	Magas

A 2.3.2. részben már említettem, hogy az iServe platformon a különféle szemantikus webalkalmazás-formátumok közös kezelésére kidolgoztak egy ún. Minimal Service

Model-t (MSM). Ez a modell az SAWSDL leírásokat is transzformálja RDF gráffá, azaz nem csak a szemantikus modellreferenciákat van lehetőségünk rejtjelezni, hanem az adott szolgáltatás interfészleírójának bizonyos részeit is. A megoldáshoz szükséges lehet egy, az XML elemek titkosítását leíró szabvány alkalmazására [25].

A biztonságos numerikus lekérdezésekben a szám-konstansok behelyettesítése mellett érdemes lehet aritmetikai operációk kiértékelését is lehetővé tenni. Ez nem kritikus feladat, de tanulságos folytatása lehet a projektnek.

Fontos viszont a szöveges rekordok kezelésének javítása biztonsági szempontból. A statisztikai támadások eredményességének csökkentésére szükséges keresni egy olyan rejtjelezése módot, amely csökkenti a sikeres statisztikai támadás valószínűségét. Jó megoldás lehet az indexek predikátum alapú titkosítása és kiértékelése (esetleg az indexeket több részre felbontva, különböző kulcsokkal rejtjelezve, hogy szövegrészlet-egyezésre is kereshessünk).

Hasznos továbbfejlesztés lehet még a numerikus adatok rejtjelezett kosarazó paramétereinek (alsó korlát, felső korlát, ládaszám) tárolásáért felelős komponens biztonsági tulajdonságainak javítása szerep alapú hozzáférés-vezérlés bevezetésével. Ez a megoldás biztosítaná, hogy az adatok felhasználóinak és publikálóinak olvasási és módosítási jogait formálisan is definiálhassuk.

4.5. Összefoglaló

Zárásként röviden össze szeretném foglalni, hogy milyen problémákkal találkoztam a dolgozat elkészítése során, hogyan oldottam meg őket, és hogy ezek milyen tanulságokkal és tapasztalatokkal szolgáltak a szakmai fejlődésemben.

Megismertem a biztonságos (cloud) adattárolás jellemző megoldásait, protokolljait és algoritmusait, és összehasonlítottam őket gyakorlati alkalmazhatóság és biztonság szerint. Bemutattam két olyan eljárást, amikkel a jelenlegi informatikai infrasturktúrán is hatékonyan végezhetünk kereséseket rejtjelezetten tárolt adatok között. A szöveges adatokat kezelő algoritmust átalakítottam és kibővítettem, hogy nehezebben támadható legyen.

Megismerkedtem a szemantikus web alapjaival, elsősorban az adatstruktúrákat definiáló és lekérdező megoldásokkal (RDF, SPARQL), és alkalmazásokkal (Sesame). Megvizsgáltam a szemantikus webalkalmazások leírására szánt legújabb szabvány-ajánlásokat (pl. SAWSDL, WSMO-család), valamint az ezeket kezelő, jelenleg elérhető legkorszerűbb rendszereket (iServe).

A SPARQL lekérdező nyelv definícióját kibővítettem, hogy nyelvi szinten képes legyen lekérdezni és módosítani a rejtjelezetten tárolt adatokat. Ez a bővítés két nagy részből állt: egyrészt a felhasználó által kezelhető szintaktikájú kliens oldali komponensből, másrészt abból a köztes kódból, amelyet ebből generáltam, és amit a szerver kiértékelhet a rejtjelezett adatok felett. A bővítéseket (EBNF segítségével) formálisan leírtam, és példákkal is illusztráltam. Ezen feladat elvégzése során sok hasznos tapasztalattal gazdagodtam a formális nyelvek és reguláris kifejezések témájában.

Elkészítettem egy Java nyelvű mintaimplementációt, amely kliens oldalon feldolgozza a lekérdezést, rejtjelez, köztes kódot generál, amit elküld a szervernek, a visszakapott rekordokat pedig dekódolja és kiszűri közülük az algoritmus jellegéből adódó hamis pozitívokat. Ezen kívül képes lekérni és feldolgozni olyan metaadatokat, amelyek a numerikus paraméterek biztonságos kezeléséhez szükségesek.

A (cloud) szerver oldalán integráltam implementációmat a Java nyelvű Sesame szemantikus adattárolóval, lehetővé téve az általam kibővített SPARQL-szintaxis kezelését. A Sesame adattároló rétegében PL/Python nyelven eljárásokat készítettem a rejtjelezett keresés támogatására. Ezen felül a már említett, numerikus paraméterekhez tartozó metaadatok kezelésére szolgáló alkalmazás szerver oldali párját is elkészítettem, szintén Java nyelven.

A kliens oldali megoldást integráltam az iServe webalkalmazás tárolóval, hogy a szolgáltatások felderítésénél a szemantikus megoldások segítségével kifejezőbb (de továbbra is rejtjelezett adatok feletti) lekérdezéseket lehessen összeállítani.

Végül teszteltem és profiloztam a rendszert, majd meghatároztam azokat az aspektusokat, amikben továbbfejleszthető vagy kibővíthető.

5. IRODALOMJEGYZÉK ÉS HIVATKOZÁSOK

1. **Winkler, V.** *Securing the Cloud: Cloud Computer Security Techniques and Tactics.* USA : Syngress, 2009.

2. *Executing SQL Over Encrypted Data in the Database-service-provider Model.* **Hacigümüs, H., et al.** Madison, Wisconsin, USA : ACM SIGMOD Conference, 2002. Proceedings of the 2002 ACM SIGMOD international conference on Management of data.

3. *Fast Query Over Encrypted Character Data in Database.* **Wang, Z., et al.** s.l. : Springer, 2005., Lecture Notes In Computer Science, Vol. 3314/2005., pp. 1027-1033.

4. *Predicate Privacy in Encryption Systems.* **Shen, E., Shi, E. and Waters, B.** s.l. : Springer, 2009., Lecture Notes in Computer Science, Vol. 5444/2009., pp. 457-473.

5. **Dr. Buttyán, L.** Privacy Enhancing Techniques. *Adatbiztonság (VIHIM102) előadásjegyzet.* 2011..

6. *Implementing Gentry's Fully-Homomorphic Encryption Scheme.* **Gentry, C. and Halevi, S.** s.l. : Springer, 2011., Lecture Notes in Computer Science, Vol. 6632/2011., pp. 129-148.

7. *Security Issues in Querying Encrypted Data.* **Kantarcioglu, M. and Clifton, C.** s.l. : Springer, 2005., Lecture Notes In Computer Science,, Vol. 3654/2005.

8. **Han, J. and Kamber, M.** *Adatbányászat Koncepciók és technikák.* Budapest : Panem Könyvkiadó, 2004.

9. **Segaran, T., Evans, C. and Taylor, J.** *Programming the Semantic Web.* Sebastopol : O'Reilly Media, 2009.

10. **Seaborne, A. and Manjunath, G.** SPARQL/Update: A language for updating RDF graphs. *W3C Member Submission* . 2008.

11. **W3C.** SPARQL Query Language for RDF. *World Wide Web Consortium (W3C).* [Online] 2008. 01. 15. [Hivatkozva: 2011. 09. 28.] http://www.w3.org/TR/rdf-sparql-query/.

12. **Fensel, D., et al.** *Semantic Web Services.* Berlin Heidelberg : Springer, 2011.

13. **Studer, R., Grimm, S. and Abecker, A.** *Semantic Web Services: Concepts, Technologies, and Applications.* Berlin Heidelberg : Springer, 2007.

14. **W3C.** OWL Web Ontology Language. *World Wide Web Consortium (W3C).* [Online] 2004. 02. 10. [Hivatkozva: 2011. 10. 04.] http://www.w3.org/TR/owl-ref.

15. —. WSDL 1.1 Element Identifiers. *World Wide Web Consortium (W3C).* [Online] 2007. 07. 20. [Hivatkozva: 2011. 10. 04.] http://www.w3.org/TR/wsdl11elementidentifiers/.

16. *iServe: a linked services publishing platform.* **Pedrinaci, C., et al.** Hersonissos : ORES 2010, 2010. Ontology Repositories and Editors for the Semantic Web.

17. *METEOR-S Web Service Annotation Framework.* **Patil, A., et al.** New York : WWW 2004, 2004. Proceedings of the 13th international conference on World Wide Web.

18. **Pedrinaci, C.** Developers Setup - GitHub. *kmi/iserve GitHub.* [Online] GitHub Inc., 2011. 06. 29. [Hivatkozva: 2011. 09. 28.] https://github.com/kmi/iserve/wiki/Developers-Setup.

19. **Aduna.** openRDF.org. *openRDF.org: Documentation.* [Online] Aduna, 2011. 09. 23. [Hivatkozva: 2011. 09. 28.] http://www.openrdf.org/documentation.jsp.

20. **Ontotext.** Clients by Industry. *Ontotext Semantic Technology Developer.* [Online] Ontotext. [Hivatkozva: 2011. 09. 30.] http://www.ontotext.com/clients.

21. **The PostgreSQL Global Development Group.** PostgreSQL. *PostgreSQL: Documentation: Manuals: PostgreSQL 9.0: PL/Python - Python Procedural Language.* [Online] 2010. [Hivatkozva: 2011. 10. 19.] http://www.postgresql.org/docs/9.0/static/plpython.html.

22. **Noelios Technologies.** Restlet - RESTful web framework for Java. *Restlet - RESTful web framework for Java.* [Online] Noelios Technologies, 2011. 10. 07. [Hivatkozva: 2011. 10. 19.] http://www.restlet.org/.

23. **Apache.** Apache CouchDB: The Apache CouchDB Project. *Apache CouchDB: The Apache CouchDB Project.* [Online] Apache. [Hivatkozva: 2011. 10. 19.] http://couchdb.apache.org/.

24. **Lundgren, Henrik.** helun/Ektorp - GitHub. *GitHub.* [Online] GitHub Inc., 2011. 10. 22. [Hivatkozva: 2011. 10. 23.] https://github.com/helun/Ektorp.

25. **W3C.** XML Encryption Syntax and Processing. *World Wide Web Consortium (W3C).* [Online] 2002. 12. 10. [Hivatkozva: 2011. 10. 18.] http://www.w3.org/TR/xmlenc-core/.

A. FÜGGELÉK: RÖVIDÍTÉSEK JEGYZÉKE

EBNF: Extended Backus-Naur Form; Környezetfüggetlen nyelvtanok leírására használatos metaszintaxis.

hRESTS: HTML for RESTful Service Descriptions; RESTful szolgáltatásokat deklaráló HTML oldalakon használatos interfészleíró mikroformátum.

IaaS: Infrastructure-as-a-Service; virtualizált hardverplatformot nyújtó felhőszolgáltatói modell.

MicroWSMO: hRESTS dokumentumok attribútumait RDFa annotációk segítségével szemantikus információkhoz (pl. WSMO-Lite ontológiához) kötő protokoll.

OWL: Web Ontology Language; ontológiák létrehozására használatos tudásreprezentációs nyelvcsalád.

OWL-S: OWL alapú ontológia szemantikus webalkalmazások automatikus kezelésére.

PaaS: Platform-as-a-Service; speciális fejlesztői eszközökkel készült, egyedi szoftverplatformra írt webalkalmazások futtatását lehetővé tevő felhőszolgáltatói modell.

PIR: Private information retrieval; olyan kriptográfiai protokoll, ahol egy adatbázisból úgy kérdezhetünk le egy elemet, hogy nem kell felfednünk, melyik volt a kívánt rekord.

RDF: Resource Description Framework; adatmodellezési specifikáció webes erőforrások és gráfadatbázisok leírására.

RDFa: RDF-in-attributes; XHTML kiterjesztés weboldalak RDF metaadatokkal való ellátásához.

REST, RESTful: Representational State Transfer, „lightweight SOA”; olyan webalkalmazás, ahol a rendszer egésze állapotmentes, mindössze az egyes erőforrások rendelkeznek állapottal.

SaaS: Software-as-a-Service; virtualizált környezetben futtatott webalkalmazást kínáló felhőszolgáltatói modell.

SAWSDL: Semantic Annotations for WSDL and XML Schema; WSDL-kiterjesztés szemantikus webalkalmazások ontológiákhoz kötéséhez.

SOA: Service-Oriented Architecture; szolgáltatásorientált architektúra lazán csatolt elosztott rendszerkomponensek számára.

SOAP: Simple Object Access Protocol; strukturált adatok cseréjére szolgáló, SOA rendszerekben használt XML alapú üzenetformátum.

SPARQL: SPARQL/Simple Protocol and RDF Query Language; lekérdező nyelv RDF adatgráfokban történő keresésre. Kiegészítése, a SPARQL/Update adatmódosító operációkat tesz lehetővé.

SWS: Semantic Web Service; számítógép által feldolgozható, szemantikus modelleket használó SOAP és REST alkalmazások.

UDDI: Universal Description Discovery and Integration; webalkalmazások regisztrálására és felderítésére alkalmas XML-tároló szabvány. Nem terjedt el széles körben.

WADL: Web Application Description Language; a WSDL specifikációnál egyszerűbb, XML formátum (általában) RESTful, HTTP alapú webalkalmazások funkcionalitásának leírásához.

WSDL: Web Service Description Language; SOAP (ritkábban REST) architektúrájú webalkalmazások funkcionalítását leíró XML formátum.

WSMO: Web Service Modeling Ontology; szemantikus webalkalmazások modellezésére használatos keretrendszer és formális nyelv. Lásd még: MicroWSMO és WSMO-Lite.

WSMO-Lite: SOAP és RESTful webalkalmazások ontológiáinak megadására használt leíró formátum.

B. FÜGGELÉK: ÁBRÁK JEGYZÉKE