Todo list

TDK-dolgozat

Horváth Dávid

horvathdown@student.elte.hu

HoloDB: Relációs demóadatok on-the-fly generálása deklaratív konfigurációból

EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

INFORMÁCIÓS RENDSZEREK TANSZÉK

Szerző:

Horváth Dávid

horvathdown@student.elte.hu

Programtervező Informatikus BSc

Témavezető:

Dr. Vincellér Zoltán

Mesteroktató

Budapest, 2024

Tartalomjegyzék

Kivonat

A modern szoftverfejlesztési módszertanok fő fókuszai közé tartozik a gyors prototípusgyártás és a (kézi, illetve automatizált) integrált tesztelés. Mindkét esetben tesztadatokat kell biztosítanunk az alkalmazás működtetéséhez. Ugyanakkor általában demóadatokra számos egyéb helyzetben is szükségünk lehet (például mockolás, füst-teszt, szoftverbemutató, edukáció). Nem csak a demóadatok előállítása (jellemzően generálás vagy anonimizálás) jelent folyamatos terhelést a fejlesztési infrastruktúrára, de azok indítási, működtetési és memóriaigénye is jelentős, hiszen hús-vér adatbázisokról van szó. A pályamunkában egy jelentősen újszerű megközelítést mutatok be, mely szükségtelenné teszi az adatok előzetes előállítását. A HoloDB egy (felhasználói szemszögből szokványos) relációs adatbázis, mely minimális indulási idővel és memóriaigénnyel futtatható egy deklaratív konfigurációs fájlból, mely tömören leírja az adatbázis szerkezetét és az adataival szembeni követelményeket. Az on-the-fly generálás miatt maguk az adatok nem foglalnak memóriát, ugyanakkor a szerver egy lekérések között is koherens adathalmazt szolgáltat. Számos célmegoldás (köztük kriptográfiai módszerek) együttes alkalmazásával koherens és gyors működést biztosító adatelérési technikák váltak lehetővé (például a virtuális indexek), így a HoloDB (alkalmas skálázás mellett) runtime performanciában sem marad alul a hagyományos adatbázisokkal szemben. A relációs modell a megfelelő adapterekkel könnyen kiterjeszthető más adatmodellekre is (például NoSQL, GraphQL). Az új megoldás reményeim szerint további előnyök mellett ötvözi a korábbi megoldások erősségeit, míg kikerüli azok fő hátrányait. Valamint olyan új lehetőségek nyílnak meg, mint a „dev mode” (az adatbázis azonnali frissülése a konfigurációs fájl módosításakor) vagy a serverless adatbázis-szolgáltatás.

1. fejezet

Bevezetés

1.1. Problémafelvetés

A szoftverfejlesztési és -release-elési folyamatok lényeges előfeltevése, hogy az alkalmazás mint absztrakt entitás snapshot jellegű, azaz a mindenkori forráskód *aktuális állapotának* derivátuma, nem függ annak történetiségétől. Ezt az átlátható képet bolygatja meg a szoftverhez tartozó produkciós adatbázis, mely önálló életet él, a felhasználói világgal való interakcióban folyamatosan változik; szerkezete, szemantikája azonban szorosan az alkalmazás mindenkori állapotához kötődik[1, 2, 3].

Az *életnek kitett* produkciós adatbázist természetesen kénytelenek vagyunk minden erőforrásigényével életben tartani, és minden release-zel migrálni[4, 5, 6]. Ám az *absztrakt értelemben vett* adatbázisra számos egyéb helyen is szükségünk van, ahol ezek a folyamatos költségek igencsak nemkívánatosak. Ilyenek esetek jellemzően az integrációs és egyéb tesztelés, gyors prototípusgyártás, kísérletezés draft adatbázisokkal, rövid demonstrációk és prezentációk.

Mivel más opció fel sem merül, ezekben az esetekben is ugyanahhoz az infrastruktúrához nyúlunk, mint a produkciós adatbázis esetében. Így például az integrációs tesztek is jelentős erőforrást fognak igénybe venni (generálási vagy anonimizálási időt, indítási és üzemeltetési költséget, memória- és tárhelyfoglalást stb.), mindezt egy rövidéletű adathalmaz biztosítása miatt, amelynek ráadásul csak töredéke lesz érintve egy-egy teszt során.

Gyakran tehát szükségünk van egy működő adatbázisra, miközben valójában nincs szükségünk igazi adatokra. Meglehetősen paradox ez a helyzet. Ráadásul van valami abszurd abban, amikor ilyen nagy mennyiségű adatot gondosan lematerializálunk, hogy aztán nagyrészt használatlanul eldobjuk. Még

ha olykor fel is merül a kérdés, hogy vajon elkerülhető-e mindez, más egyértelmű megoldás híján továbbra is inkább a meglévő módszerek tökéletesítésére törekszünk. Ördögi kör.

Amikor egy szoftver kezdeti fázisában az adatbázis szerkezete is gyakran változik, a próbálgatáshoz, teszteléshez fenntartott lokális adatbázist folyton újra létre kell hozni, vagy pedig igazítani a megváltozott adatszerkezethez. Mindkét megközelítés nehézségekkel és erőforráshasználattal jár. Sokkal egyszerűbb lenne az adatbázis szerkezetét egyetlen deklaratív konfigurációs fájltól (vagy akár közvetlenül az alkalmazásban definiált adatosztályok szerkezetétől) függővé tenni.

A dolgozatban elsősorban ezekre a problémákra kínálok egy újszerű megoldást. Mivel a bemutatott szoftver egyúttal egy általános keretrendszer virtuális adattárakhoz, számos további alkalmazása is adódhat.

1.2. Tanulmány

1.2.1. A probléma státusza

Érdemes tisztázni, hogy nem ismerek még csak próbálkozást sem, mely a fenti problémafelvetésemre kívánt volna célzott megoldást találni, és főképpen nem tudok olyanról, amely ilyesmit már nyújtana. Természetesen valaminek a hiányát nehéz bizonyítani, és az állítást (egyelőre) pusztán egy sokéves fejlesztői tapasztalat során szerzett erőteljes benyomás támasztja alá.

Számomra meglepő, hogy egy ilyen nehézségre, mely napi szinten érinti fejlesztők, tesztelők és mások munkáját, produktivitását, egyelőre csak a legszofisztikáltabb *kerülőmegoldások* születtek. Hamarosan megkísérlem megtippelni ennek az okát. Előbb azonban röviden áttekintem azokat a tipikus megoldásokat, melyek valamennyire kapcsolódnak ehhez a problémához, némileg hasonlót akarnak megoldani.

1.2.2. Kérések mockolása on-the-fly

Főként webes API-k által szolgáltatott adathalmazok mockolásakor találkozunk olyan megoldással, mely a lekérés (és persze az előre beállított szabályok) alapján generálja a válaszként visszaadandó adatstruktúrát (például egy rekordot vagy dokumentumot JSON formátumban). Az adatok véletlenszerűen kerülnek kitöltésre a válaszhoz tartozó séma keretébe[7, 8, 9, 10].

A módszer SQL-lekérdezések közvetlen mockolásához is adaptálható, a lekérdezés (és ha ismert, akkor a séma) alapján meg kell állapítani a visszaadandó eredménytábla szerkezetét, adattípusait,

az egy-egy oszlopba kerülő adatok jellegét, majd random jelleggel elő kell állítani bizonyos számú, ezeknek a követelményeknek megfelelő eredményrekordot[11].

Ebbe a kategóriába sorolhatók az úgynevezett faker segédkönyvtárak is. Ezek olyan véletlenadat-generálók, melyek bizonyos gyakori entitástípusok adatait (pl. személynevek, e-mail, dátum) szolgáltatják[12, 13].

Ez a megközelítés természetesen nem is törekszik arra, hogy koherens eredményt adjon több lekérés esetén, tehát csak olyankor használható, ahol a lekérések nem épülnek egymásra. Cserébe erőforrásigénye csekély, nincs szükség az adatok tényleges tárolására.

1.2.3. Seed-alapú virtuális világok

Játékokban és egyszerűbb szimulációkban már igen régóta generálnak kisebb világokat, pályákat procedurálisan valamilyen pszeudovéletlen algoritmus segítségével. Ennek azonban komoly méretbeli korlátai vannak. Felmerül a gondolat, hogy lehetséges-e óriás méretű világokat konzisztensen a játékosok rendelkezésére bocsátani, anélkül, hogy le kellene generálni az egészet.

Univerzumot, nyílt világokat és egyéb óriás rendszereket felvonultató játékokban a probléma már régen felmerült, és természetesen megoldások is születtek. Egy ilyen módszer, ha a világteret (sorosan vagy hierarchikusan) szeletekre osztjuk, és biztosítunk egy függvényt, mely minden szelethez rendel egy saját seed értéket. Legegyszerűbb esetben a seed az adott szelet sorszámából generált hash. Amikor szükségünk van valamely szelet tartalmára, a szelet seedjéből kiindulva, az adott szabályrendszer szerint on-demand legeneráljuk az adott szelet tartalmát. A sűrűn benépesített világok szomszédos szeleteinek összeillesztése néha nem triviális. Felületszerű, folytonos struktúrák generálása esetén gyakran használnak valamilyen eleve interpolált eljárást, így biztosítva a töretlen átmenetet a szomszédos szeletek határvonalainál.

Egy közismert korai példa seed-alapú módszert alkalmazó szoftverre az *Elite* nevű 1984-ben kiadott űrhajós videójáték[14, 15]. A játékban meglátogatható bolygók kapnak egy-egy seedet, mely már teljesen determinálja a felépítésüket, így elég a bolygó meglátogatásakor legenerálni a tartalmát. A módszert aztán átvették és továbbfejlesztették az újabb és újabb óriástérképes játékok. A seed-alapú módszernél valójában nincs határa, mekkora virtuális világot definiálhatunk. Az Elite sikere után hamar megjelent a kifejezetten végtelen nyílt világ koncepciója, mely meghatározó eleme olyan közismert játékoknak, mint a *Starflight* (1986)[16], a *Minecraft* (2011)[17], a *StarMade* (2012)[18], a *No Man's Sky*[19], a *Starfield* (2023)[20] stb.

Ha elvonatkoztatunk a játékoktól és szimulációktól, ez a generálási módszer közel tetszőleges jellegű virtuális adathalmaz böngészését képes biztosítani. De azonnal szembeötlik a legfőbb hátrány is: a virtuális világ-adathalmaz nem kereshető. Nem tudjuk például hatékonyan lekérdezni, mely világszeletekben lelhető föl egy adott típusú objektum.

1.2.4. Relációs adatok generálása

Amikor adatokat generálunk egy relációs modellbe, azt legegyszerűbb megközelítésben fölfoghatjuk úgy is, mint az előbb bemutatott virtuális világ materializációját egy relációs adatbázisszerveren. Milyen előnyöket nyerünk ezzel? Lássunk néhányat:

- utófeldolgozás: utólag holisztikus módosításokat eszközölhetünk az adathalmazon
- egynemű módosíthatóság: minden eleve tárolva van, nem kell külön kezelni a változásokat
- kereshetőség: az adatbázis-kezelő biztosítja az indexeket

A fő hátrány egyértelmű: a nagy tárhelyigény. Vegyük észre, hogy még a kereshetőség is az indexek által foglalt további tárhely által valósul meg (az adatok eleve nagy tárhelyfoglalása mellett tűnhet esetleg elhanyagolhatónak). Ha pedig nem materializált adatok mellé gyártunk indexeket, akkor ki kell dolgoznunk valamilyen módszert a módosulások detektálására, ami általában nem triviális.

Egyes generátorok a séma és az adatok jellegének deklaratív leírását is támogatják. Ez a leírás tárolható egy verziókezelhető, dokumentálható, könnyen szerkeszthető konfigurációs fájlban[21, 22].

1.2.5. Relációs adatok anonimizálása

Ha már rendelkezésre áll egy éles adatbázis, kézenfekvőnek látszik, hogy ennek replikáit használjuk a teszt- vagy mockadatbázis kiindulópontjaként[23, 24]. Így rögtön egy valószerű adathalmazzal indulunk, az érzékeny adatok miatt azonban anonimizálásra van szükség. Tehát a generáláshoz szükséges számítási igényt összességében felcseréljük a másolás és anonimizáció költségeivel.

Meglévő adatokból kiindulni néha az anonimizálással együtt is kezelhetőbb, mint egy megfelelő generálási folyamatot kiépíteni[25]. Egy köztes megoldás lehet a séma és az adatok nagyjából jellegének letapogatása, és az ez alapján történő újragenerálás.

Azonban a bevezetőben felsorolt szituációk egy részében nem is áll rendelkezésre egy létező produkciós adatbázis. És ha rendelkezésre is áll, további probléma, hogy az adatok megszerzéséhez az

éles adatbázist kell felkeresni, ami egyfelől az éles környezet nemkívánatos terhelését jelenti, másrészt az oda való beauthentikálás (a konkrét megoldástól függően) biztonsági kockázatot rejthet.

Egy általános megoldás tehát aligha épülhet anonimizálásos módszerre.

1.2.6. Nagy teljesítményű megoldások

Számos nagyteljesítményű szolgáltatás érhető el a piacon szintetikus adatok generálásához. A *MOSTLY AI* megoldása az akkurátus adatokra helyezi a hangsúlyt. Az adatok statisztikai jellemzői és mintázatának lényeges elemei jól illeszkednek a valós adatoknál látható mintázatokra[26].

Egyes platformok általános megoldást nyújtanak az adatgenerálásra, anonimizációra és CI/CD-integrációra. Ilyen például a *Tonic*[27], a *GenRocket*[28] és a *Delphix*[29].

Ezek a rendszerek erősen optimalizálják és párhuzamosítják a tesztadatok előállításának egyes lépéseit, így a fapados megoldásokhoz képest költséghatékonyak. Csodát viszont nem lehet várni: a számításigény aszimptotikusan, a tárhelyigény pedig teljes egészében ugyanaz marad.

1.2.7. Miért csak kerülő megoldások vannak?

A rohanó világban, különösen a lokális optimumra törekvő fejlesztők világában, az egyértelmű alternatív megoldás hiánya szükségszerűen a bevett kerékvonalak továbbmélyítését eredményezi. Úgy látszik, hogy egy újszerű megoldással való kísérletezés egyszerűen kívül esik a megszokott paradigmán, és nem fér bele az időbe.

Néhány ezzel kapcsolatos tényezőt tényszerűen megállapíthatunk.

- **Érdekhiány**: a releváns területen működő nagyobb vendorok elsősorban erőforrásokat adnak bérbe, és az általuk fejlesztett architektúrák (történetileg érthetően) elsősorban az ezeken futó tényleges adatbázisok kezelését kell megoldják; nincs tehát nyomás jelentősen más típusú termékkel való kísérletezésre.
- **Bejáratott alternatívák**: az on-the-fly mockolás lehetősége, a NoSQL adatbázisok és felhőmegoldások fejlődése stb. sztenderdképző kerülőmegoldásokként elfedték a problémát.
- **Megfelelő SQL plannerek eddigi hiánya**: az Apache Calcite például csak az utóbbi néhány évben vált népszerűvé, és inkább elsősorban adatintegrátorként.
- **Kontraintuitivitás**: az új megoldás működésmódja nem triviális, így nem egyszerű átlátni a koncepciót, és megérteni hogy „hol is vannak az adatok”.

Jelenleg csak tippelni tudok, mennyire okozói ezek (illetve ezek összjátéka) a vizsgált helyzetnek. Érdemes lenne ezt a kérdést a későbbiekben külön tanulmányban alaposabban is megvizsgálni, nagyban építve egy megfelelően megtervezett, fejlesztőkre és technikai vezetőkre is kiterjedő kérdőíves kutatásra.

1.3. Az új megközelítés

A következőkben lényegileg egy sajátos relációsadatbázis-szervert fogok bemutatni. Ennek a már működő és kipróbálható szoftvernek a HoloDB nevet adtam. Kezdetben ugyanis úgy gondoltam rá, mint a hologram-ábrák megfelelőjére az adatbázisok világában, mint ami illuzórikusan kerül kivetítésre egy kis méretű definícióból. Ebben az adatbázisban ugyanis nincsenek tárolva adatok (legfeljebb adatok változásai, ha az írhatóság be van kapcsolva).

A deklaratív konfiguráció alapján a lekérdezések eredménye on-the-fly kerül kiszámításra, úgy, hogy a lekérdezések közötti konzisztencia biztosított, vagyis az egymás utáni lekérdezések egy koherens adatbázis képét adják. Ezt pusztán adatmezőnként determinisztikusan generált értékekkel is meg lehetne oldani, ekkor a kereséshez a teljes tábla szkennelése lenne szükséges (vagyis visszajutnánk ugyanahhoz a problémához, mint amit a seed-alapú világokban való kereshetőség kapcsán említettem). A virtuális indexeknek köszönhetően azonban nincs szükség ilyen költséges műveletekre. Az értéklekérés és keresés oda-vissza egybecsengő működésének biztosításához a kriptográfiából ismert módszereket is felhasználtam.

A végeredmény egy skálázható és testreszabható nyílt-forráskódú adatbázis-mockoló eszköz, számos kiegészítővel (REPL; networking; JDBC és JPA támogatása, Docker, haladó értékkészletek stb.).

Mivel nincsenek adatok, a memóriafogyasztás minimális, különösen, ha nem érkezik egyszerre sok lekérés. A nem létező adatok természetesen a háttértáron sem foglalnak helyet. Mivel a szerver elindulása pusztán a konfiguráció betöltéséből áll, az indulási idő is minimális. Az egyetlen előzetes költség, amit a felhasználónak be kell fektetnie, az a konfigurációs fájl megírása, bár példafájlokból kiindulva ez szinte néhány másolással is megoldható, illetve akár generálható is.

Az architektúra tárgyalása során kifejezetten a relációs storage API implementációja lesz a fókuszban. Ez az a szűk keresztmetszet, amely a megoldás újszerűségéhez leginkább hozzájárul, és ami a prototípusban is a legkidolgozottabb rész. A forráskód az itt bemutatottakon túl is sok érdekességet rejt, ezért jó szívvel ajánlom a szoftver Git-tárolóját, melyet a dolgozat legvégén linkeltem is.

A szoftver **Java környezetben készült**, a továbbiakban ez implicite odaértendő akkor is, ha külön nem hívom fel rá a figyelmet.

1.4. Alkalmazhatóság

A fentiek fényében az alábbi területeken biztosan alkalmazható a szoftver:

- **Tesztelés:** Integrációs és egyéb tesztek, kipróbálás.
- **Fejlesztés:** Kezdeti fázisban, prototípusoknál, gyakran változó adatszerkezet esetében.
- **Demonstrációk:** Példaadatok szolgáltatásához bemutatókhoz, koncepciótervekhez.
- **Oktatás:** Ad hoc adathalmaz biztosításához konkrét példákhoz és feladatokhoz.

Mivel az új megoldás erőforrásigénybeli paraméterei radikálisan eltérnek a tényleges adatbázisokétól, számos olyan helyen is felmerülhet az alkalmazása, ahol hagyományosan rossz gyakorlatnak számít adatbázisszervert futtatni. Ilyen esetek például a füsttesztek, unit tesztek, serverless; vagy a főleg frontend-fejlesztésből ismerős *dev mode* (vagy live mode), illetve annak extrém változata, a *zero mode* (részletesebben lásd lentebb).

1.5. Célkitűzések

A következőket fogjuk elvárni a megoldástól:

1. Támogatja a relációs adatmodellt.
2. Determinisztikus, koherens.
3. Az adatok könnyen variálhatóak.
4. Óriás adatmennyiséget is képes szimulálni.
5. Gyors, indexelt keresést nyújt, a lekérések időigénye elfogadható.
6. Azonnal elindul, nincs számottevő preparálási folyamat.
7. Kevés memóriát foglal, az adatokat on-the-fly számítja.
8. Könnyen és rugalmasan konfigurálható.
9. Skálázható, finomhangolható.
10. Egyedi működéssel könnyen bővíthető.
11. Opcionálisan írható.

1.6. Megjegyzés a dolgozat koncepciójához

A dolgozat egy újszerű megoldást mutat be a fentebb megfogalmazott problémára. Fő célja az elképzelés működőképességének bizonyítása, tehát annak a láncnak a bemutatása, mely a különféle

részmegoldásokat egybefűzi, hogy létrehozza a relációs adatbázis illúzióját, miközben megfelel ez imént megfogalmazott célkitűzéseknek. Mind a fejlesztés, mind a dolgozat írása során a szoftvertechnológiai és architekturális szempontokra koncentráltam. A későbbiekben természetesen egyre fontosabbá válik majd az implementációs lehetőségek részletes analízise, nagy mennyiségű mérési adat felhalmozása és összehasonlítása. Ha azonban kezdettől ezekre az aspektusokra koncentráltam volna, lehetetlen lett volna egy ilyen átfogó koncepció megvalósítása és bemutatása. Így a felvonultatott mérési eredmények is kifejezetten azt a célt szolgálják, hogy az átfogó működőképességet integrált tesztekből származó számszerű adatokkal alátámasszák. Figyelembe kell venni azt is, hogy a projekt nem egy már bejáratott nagyobb kutatási program része, egyelőre nincsenek további dedikált erőforrásai a részterületek mérési adatainak feldolgozására.

2. fejezet

Architektúra

2.1. A nagy illúzió: adatok a semmiből

Olyan adatszolgáltatás felépítésére vállalkozunk tehát, mely egy **tetszőlegesen nagy méretű, koherens és kereshető** adattárat imitál, miközben mindebből semmi sem létezik fizikálisan. Hogy egy koherens adattároló látszatát elérjük, apró eszközök egész tárát kell összeverbuválni, akár olyan távol eső területekről mint az automataelmélet és a kriptográfia. Mivel a cél egy felhasználói szemszögből is áttekinthető és rugalmas eszköz biztosítása, választanunk kell egy kellően általános adathozzáférési modellt a virtuális adathalmaz prezentálásához. Ez az adatmodell a relációs adatmodell lesz, melyhez egy konkrét **relációs storage API**-t fogunk definiálni. Aligha érdemes más megközelítést választani, hiszen a relációs adatmodell nem csak tradicionálisan megkerülhetetlen és máig a legnépszerűbb adattárolási módszer[30], de egyúttal egy olyan általános adatelérési normalizációs szabvány, mely közös nevezőként tud szolgálni további modellhez (például NoSQL, gráfadatbázisok).

A legtöbb szoftver architektúrájában jól elkülöníthető valamilyen alapvető szerepet betöltő, fokális entitástípus (például egy könyvtári katalógusban a *könyv*). Célszerű lehet ilyen entitást esetünkben is meghatározni, hogy segítse az architektúrában való eligazodást. Úgy tűnik, a virtuális adattár legalapvetőbb objektuma az ***értéklista***. Az értéklista egyszerű esete valamely egyszerű *értékkészlet*, az API-ban pedig egy-egy *oszlop* értéklistájával fogunk találkozni.

Egy oszlopban azonos típusú adatok gyűjteménye szerepel. Ezeket az adatokat általában egy adott *értékkészlet*ből válogatjuk össze. Jó, ha az értékkészlet rendezett, kereshető; mint majd látjuk, ez az egyszerű megkötés már általában elégséges ahhoz, hogy maga az oszlop is kereshető legyen.

Néha esetleg ki kell lépnünk a szigorú oszlop-orientáltságból. Lehetnek például összefüggések

egy táblán belüli oszlopok között. Ezt azonban még mindig egyszerűbb az oszlopos megközelítés megfelelő kiterjesztésével elérni, mintsem áttérni valamiféle rekordonkénti generálásra. Úgy is mondhatjuk, hogy az oszlopok alatti mezők jó jelöltek arra, hogy „világszeletek” legyenek, míg egész rekordok nem.

Lehetnek továbbá megkötések táblák között is. Ezek kezelése is könnyen elvégezhető az oszlopos alapvetésre építve. Az egyoszlopos idegen kulcsoknál például csak annyit kell biztosítani, hogy a hivatkozó oszlop értékkészletét a hivatkozott oszlop tartalma adja.

A mindenkori módszert, amivel egy oszlop értéklistáját biztosítjuk, *értékkiosztás*nak fogom nevezni. A virtuális értékkiosztások olyan függvényt valósítanak meg, mellyel egy adott mező értéke determinisztikusan előállítható. De az igazán lényeges kihívást az jelenti, hogy szeretnénk, ha az értéklistában keresni is tudnánk, méghozzá hatékonyan, hasonlóan ahhoz, mint amikor tényleges adatbázisok esetén az oszlophoz már generálva van egy index. A virtuális adattár esetén is lehetséges lenne indexeket generálni, de ez az adatok materializálásával járna, a virtuális adattár mégse lenne többé virtuális. Ideális esetben tehát az indexek is virtuálisak (ennek oda-vissza koherensnek kell lennie; itt jön majd képbe a kriptográfia). Elsősorban olyan megoldásokat fogok bemutatni, amelyek teljesítik a kereshetőség erősebb feltételét is, azaz **kereshető virtuális értékkiosztások**.

A legtöbb oszlop értékkészlete jól behatárolható. Az oszlopban tételesen szereplő értékek egy efölötti tetszőlegesen sorrendezett multihalmazt (tömböt) alkotnak, néha ehhez statisztikai megkötések is tartoznak (*NULL*-ok száma, értékgyakoriság stb.). A tábla rekordjait sorrendezettnek fogom tekinteni: azonos táblabeli valamely két oszlop értéklistájából kiválasztott egy-egy mező azonos rekordhoz tartozik pontosan akkor, ha a sorrendbeli pozíciójuk (sorindexük) megegyezik.

Vegyük észre, hogy több független, külön-külön is implementálható problémával szembesülünk:

1. az értékkészlet biztosítása
2. az értékkészlet monoton rendezett multihalmazának képzése
3. az értékek összekeverése

A multihalmaz itt egy monoton növekvő tömböt jelent, ezt fogjuk még egy további lépésben összekeverni egy permutáló függvény alkalmazásával. Ennek a többlépéses módszernek az átláthatóságon kívül az is nagy előnye, hogy a lépések egymástól függetlenül skálázhatók.

A lépéseket úgy kell megvalósítani, hogy a végső értéklista adott pozíciójú eleme is gyorsan lekérhető legyen, valamint egy adott érték vagy értéksáv előfordulási pozíciói is (keresés). A permutáció esetén ez annyit jelent, hogy legyen effektíve invertálható.

Az értékkészletből történő **kétlépéses értékkiosztásnak** ezzel az ötletével egy egyszerűbb adatbázis szimulációjának kívánalmait már tulajdonképpen le is fedtük, de néhány további hasznos értékkiosztási módszert is be fogok majd mutatni.

Ha oszlopokat elő tudunk állítani, akkor a táblák és egyéb objektumok köréépítése nem jelent kihívást. Ha pedig a relációs adatmodell ezen objektumait sikeresen implementáltuk, a relációs műveletek és **az SQL lekérdezések megvalósítása már a hagyományos módon történhet**. Éppen ezért a továbbiakban a lekérdezések problémakörével csak érintőlegesen foglalkozom. Ma már elérhető néhány nyílt forrású jól felépített SQL query planner framework is, melyek megkönnyítik egy teljes körű SQL-szerver kiépítését. Ilyen például az általam használt Apache Calcite. Ugyanakkor egy saját SQL-futtatót is implementáltam, mely sok esetben gyorsabb fut, cserébe csak limitált SQL-t támogat.

2.2. A prototípusról

A virtuális adatbázist megvalósító szoftvert HoloDB néven tettem elérhetővé. Az elnevezés a hologramokra utal: a konfigurációból leképzett, „kivetített” virtuális adatbázis hasonlóan illuzórikus, mint a hologramból előhívott térbeli látvány.

A HoloDB sok tekintetben már kinőtte a prototípus státuszt, hiszen egy nagyívű rétegelt architektúráról van szó, mely kellően megszilárdult alapvonalakkal rendelkezik a hosszútávú fejlesztéshez. Ugyanakkor stabil verzióról még nem beszélhetünk; az implementációs részleteket, valós használati eseteket, más eszközökkel való integrációt illetően még számos felmérendő, kikísérletezendő dolog adódik.

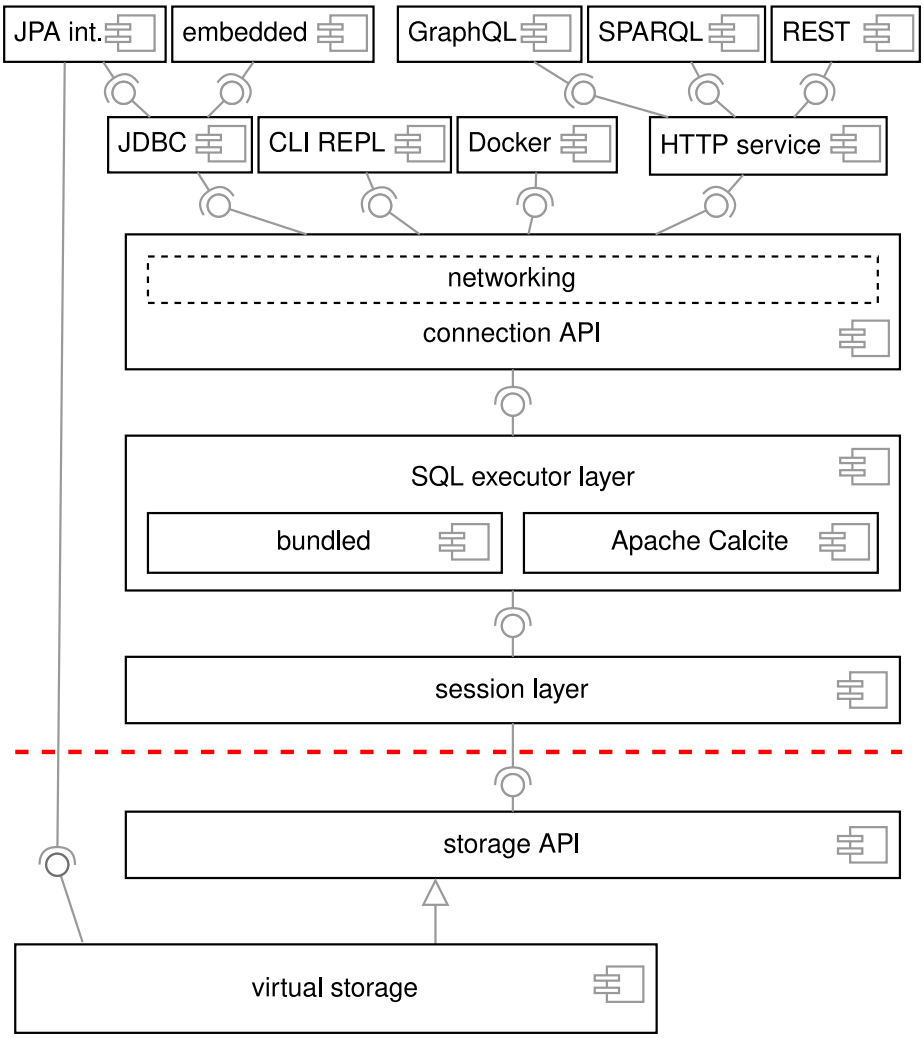
A szerver Docker konténerként is futtatható, amihez a Docker Hubon biztosítok egy testreszabott képet. A felhasználói Dockerfile mellé csak a konfigurációs fájlt kell elhelyezni az indításhoz. A testreszabott kép a kis erőforrásigény miatt serverless végpontként is jól használható.

A HoloDB a MiniBase projektre épül. Utóbbi egy relációsadatbázis-keretrendszer, mely definiálja a storage API-t, és annak konkrét implementációjához agnosztikusan viszonyulva egy komplett adatbázisszervert épít fölé; beleértve a lekérdezések fordítását és futtatását, a felhasználói munkamenetek, tranzakciók és változók kezelését, a runtime sémamódosításokat, illetve új táblákat és más felelősségi köröket. A HoloDB ehhez a storage API virtuális implementációját, a konfiguráció kezelését és a megfelelő futtatási környezetet adja hozzá.

A MiniBase közvetlenül biztosít egy SQL-futtató felületet, a MiniConnect API megvalósításával. A MiniConnect testvérprojekt célja egy egyszerű és egyértelmű adatbázis-elérési API definiálása,

melyet a JDBC nehézkessége motivált. A MiniConnecthez különféle middleware változatok is elérhetők, például hálózaton keresztül is használható, illetve letisztultsága miatt könnyen írhatók hozzá dekorátorok, proxyk. Egy (deb csomagként is elérhető) REPL is tartozik hozzá, mellyel parancssorból tudunk hálózaton keresztül csatlakozni a MiniConnectet támogató tetszőleges adatbázishoz. Ez egyúttal a HoloDB kipróbálásának legkényelmesebb módja, számos kényelmi funkciót támogat (szép kimenet, szintaxiskiemelés, automatikus kiegészítés, parancstörténet stb.).

Az architektúra köré sokféle eszközt fejlesztettem, melyek a robusztusság eltérő fokain állnak. Ilyenek például: automatikus REST API, GraphQL adapter, authentikációs proxy, Apache Calcite SQL driver stb.



2.1. ábra. A virtuális adatbázis vázlatos architektúrája

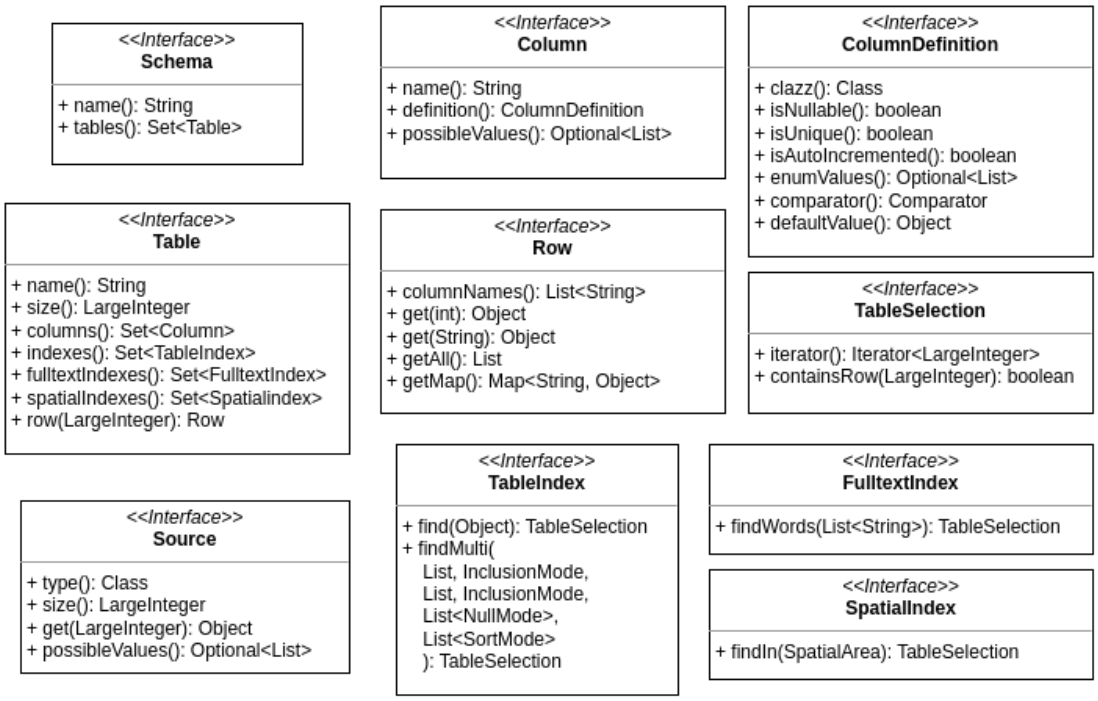
2.3. A relációs storage API

Ha már körvonalaztuk a relációs adathozzáférés fő entitásait, ezeket rendre implementálva egy funkcionálisan teljes körű relációs adattárat kapunk. Az entitások meghatározásakor elsőként a *hozzáférés* szempontjait érdemes átgondolni. Az írhatóság problémakörével később foglalkozom.

Nyilvánvalóan gondoskodni kell az alapvető hierarchiáról, erre szolgálnak a `Schema`, `Table` és `Column` típusok. Egy-egy oszlop elvont tulajdonságait a `ColumnDefinition` típus fogja össze. Bár az egyes mezők értékeit általában oszloporientáltan állítjuk elő (jellemzően az értéklistákat megadó `Source` objektumok használatával), a lekérések szemszögéből jobbnak mutatkozik a rekordorientált megközelítés. A rekordot a Row típus reprezentálja, ezen keresztül kérhetőek le a konkrét mezőértékek az oszlopnév vagy oszloppozíció megadásával.

Az adatok hatékony keresését virtuális indexek fogják segíteni. Három ilyen indexfajtával fogok foglalkozni: a normál `TableIndex`, a `FulltextIndex` és a `SpatialIndex` típusokkal. Ezek keresőmetódusai egy `TableSelection` példánnyal térnek vissza, melyen keresztül a találati listához férünk hozzá. A `TableIndex` a szokásos rendezett indextípus, mely egyúttal teljes kontrollal rendelkezik a találati lista rendezése és a *NULL* értékek kezelése fölött. Ha többoszlopos, akkor az oszloplista bármely prefixére is indexként működik (leftmost prefix lookup). A másik két típus találatainak rendezése definiálatlan. Egyik sem ad vissza *NULL* mezőket.

A következő osztálydiagramon a vázlatosan összefoglaltam az interfészeket:



2.2. ábra. A storage API fő interfészei (egy relációs adattár alapentitásai)

2.4. Az írhatósági réteg

A virtuális adattár egy-egy példánya várhatóan viszonylag rövid élettartamú lesz. Feltehetjük, hogy az írási műveletek összességében kevés módosítást eredményeznek. A tipikus szcenáriókban főként egyes konkrét rekordok lesznek módosítva, hozzáadva, törölve. A tömeges módosítások kérdésével itt nem foglalkozom, de nem látszik akadálya az ezek optimalizálását célzó továbbfejlesztésnek.

Az írhatósági réteget arra is használhatjuk, hogy egy-egy tesztesethez egyéni rekordokat adjunk hozzá, ami megkötéseken keresztül nehézkesebb lenne.

A módosítások támogatásához először is a `Table` interfészt magát ki kell egészíteni a megfelelő műveletekkel (pl. `isWritable()`, `applyPatch(TablePatch)`, `sequence()`). A csak-írható implementáció persze az írási műveleteket nem fogja támogatni.

A `DiffTable` egy tetszőleges `Table` implementáció fölé elhelyezhető dekorátor. Teljesen áttetszően működik: amíg nincsenek módosítások, minden műveletet egyszerűen továbbít a dekorált tábla felé. Módosítás esetén a különbségeket (módosított értékek, beszúrt és törölt sorok) a memóriában tárolja. A további lekérdezésekkor figyelembe veszi a különbségréteget a továbbhíváskor, illetve a szükséges mértékben összefésüli az eredményt a különbségekkel.

Olyan táblák is dekorálhatók így, amelyek valós adatbázistáblákat használnak backendként. Ezzel a módszerrel írhatóan használhatunk egy létező adatbázist, az eredeti adatok módosítása nélkül.

A tranzakciók kezelésénél megmaradjunk a réteges felépítésnél, semmiképp nem toljuk a táblákra a tranzakciókezelés terhét.[1]

Egy teljesen általános tranzakciókezelési módszer, amikor a szerver két állapotát különböztetjük meg, a `READ` és a `WRITE` státuszokat. Feltételezzük, hogy jóval több olvasási kérés érkezik be, mint írási.

Az alapértelmezett `READ` státuszban feltétel nélkül, konkurensen szolgáljuk ki a beérkező olvasási lekérdezéseket. Ha azonban egy módosító lekérdezés, illetve tranzakció érkezik, elhelyezzük a módosító műveleteknek fenntartott queue-ba, és átváltunk `WRITE` módba.

`WRITE` módban a beérkező olvasási, illetve írási műveletek a számukra fenntartott két dedikált queue-ba kerülnek. Először kiszolgáljuk a még folyamatban lévő olvasási műveleteket, ezután sorra szekvenciálisan kivesszük és végrehajtjuk a módosítási queue elemeit, amíg van ilyen. Ha már nincs elem a módosítási queue-ban, akkor az olvasási queue összes elemét felszabadítjuk, és továbbítjuk párhuzamos végrehajtásra, miközben visszaállítjuk a `READ` státuszt.

[1] Mint több adatbáziskezelőnél láthatjuk (a *H2* például tipikusan ilyen). A munkamenetek kezelésének a táblainterfész alá tolásával különféle architekturális nehézségek jelentkeznek.

Új módosító művelet érkezésekor a ciklus újrakezdődik.

A `DiffTable` dekorátor minden további nélkül elhelyezhető más `DiffTable` objektumok fölé is, ami lehetővé teszi, hogy egymásra épülő snapshotokat tároljunk. Így könnyen megvalósítható a *multiversion concurrency control* (MVCC), amikor a kiinduló állapot snapshotja a módosító műveletek közben is rendelkezésre áll, azaz az olvasási műveletek ekkor is zavartalanul futtathatók. Az írási tranzakció végeztével az érintett `DiffTable` példányok beküldik a saját módosítási rétegüket a dekorált táblába, így az belefésülésre kerül a megosztott adattérbe.

A snapshotok alkalmazásából a tranzakciós savepointok támogatása is természetesen következik. Egy savepointra való visszaugrás egyszerűen a megfelelő szintre való visszatérést jelenti a `DiffTable` dekorátorok egymásra épülő hierarchiájában.

Mivel a snapshotkezelés táblánként függetlenül működtethető, párhuzamosan is futtathatunk írási tranzakciókat, amíg azok különböző táblákon futnak.

2.5. Konfiguráció

A felhasználónak mindössze a konfigurációs fájlt kell szerkesztenie, hogy nulláról indulva egy teljes körű adatbázist tudjon futtatni. A fájlban megadhatja az adatbázis-séma elemeit, megkötéseit. Az oszlopok értékkészlete, keverési módja és egyéb paraméterei kényelmesen állíthatók, például előre adott értéklistákkal vagy akár reguláris kifejezéssel. A virtuális adatbázis kész Docker konténerrel is indítható, ez esetben tényleg csak a konfigurációs fájlra van szükség.

A konfigurációs fájl alapértelmezetten YAML formátumú, de más adatleíró nyelv is használható a megfelelő Jackson mappingen keresztül. A Jackson adatszerializációs keretrendszer Java objektumokat, adatstruktúrákat, különböző adatleíró nyelveken írt fájlokat, karakterláncokat és más adatreprezentációkat konvertál egymásba[31].

A Jackson mappinggel ellátott konfigurációs osztályokhoz formális specifikáció is tartozik, így a fájl strukturális és szemantikai helyessége statikusan ellenőrizhető. A formális specifikáció használatához csak egy minimalisztikus dedikált csomag függőségként való beemelése szükséges. A konfigurációs adatstruktúra a konfigurációs osztályok közvetlen használatával programozottan is felépíthető.

```yaml
seed: 425364
schemas:
  - name: shop
    tables:
      - name: customers
        size: 5
        columns:
          - name: id
            mode: COUNTER
          - name: firstname
            valuesBundle: forenames
          - name: lastname
            valuesBundle: surnames
          - name: birth
            valuesRange: [1950, 2000]
      - name: orders
        size: 12
        columns:
          - name: id
            mode: COUNTER
          - name: cid
            valuesForeignColumn:
              [customers, id]
          - name: product
            valuesBundle: fruits
          - name: quantity
            valuesRange: [1, 10]
```

⇒

customers

id	firstname	lastname	birth
1	Howard	Anderson	1968
2	Rebecca	Ferguson	1959
3	Jeremy	Moore	2000
4	Julie	Ellis	1951
5	Kathleen	Cook	1971

orders

id	cid	product	quantity
1	5	date	10
2	2	orange	1
3	2	sloe	2
4	3	melon	7
5	5	guava	9
6	3	orange	6
7	2	plantain	3
8	4	pear	7
9	4	papaya	9
10	5	lime	9
11	3	sloe	4
12	1	strawberry	1

2.3. ábra. Egy minimalisztikus konfiguráció és eredménye

Java JPA projekt esetén JPA-entitásokra helyezett annotációkkal is leírható a teljes konfiguráció, így beágyazott adatbázissal, külső eszközök nélkül is futtathatunk teszteket. Ehhez elegendő a megfelelő JDBC connection sztringet megadni, a többit (keretrendszertől függően[2]) automatikusan elvégzi az inicializáló szolgáltatás.

Az egyéni működéssel való kiegészítés több szinten támogatott. A beépített értékkiosztási módok esetén is lehetőség van egyéni adatfájl betöltésére, a permutációk és rányújtások egyéni implementációjának használatára. De maga az értékkiosztási mechanizmus is lecserélhető egyedi implementációra, ehhez meg kell adnunk egy olyan osztály nevét, mely implementálja a `SourceFactory` interfészt (illetve biztosítani kell, hogy az egyedi osztályok láthatóak legyenek a classpath alatt). Az egyéni működések egy része közvetlenül a konfigurációból is hivatkozható.

[2] A *Spring*, *Micronaut* és *Quarkus* beépítetten támogatott, de elméletileg bármilyen rendszerhez igazítható a megoldás.

A YAML formátum a humán kezelhetőségen kívül azzal az előnnyel is jár, hogy könnyen előfeldolgozható. Például egy megfelelő sablonkezelő használatával ugyanazt a vázat használva különféle tesztesetekhez finomhangolhatjuk az adatbázist.[3]

A konfiguráció gépileg is generálható. A prototípus tartalmaz néhány ilyen segédeszközt, ezek közül a legfontosabb a konfigurációt a meglévő adatbázis letapogatásával, különféle heurisztikák bevetésével legeneráló, Python nyelven írt szkriptfájl. Az eredmény (és persze maga a szkriptfájl is) könnyen testreszabható.

A konfiguráció szofisztikáltabb generálásának egy lehetséges módja, ha a konfigurációt optimalizáló mesterséges intelligenciára épülő módszert vetünk be. Ennek kutatása egy izgalmas jövőbeli projektnek ígérkezik. Érdekes probléma például, hogyan érdemes számítani a bemeneti adatbázis és az előállított konfiguráció közötti megfelelőségi mutatót[32]. Azonban a generálás egyszerű szóbeli megfogalmazás alapján is történhet, erre még az általános ChatGPT is képes.

A konfigurációs fájl az adatbázis nagyjából definíciójának is felfogható. Így adódik az az ígéretes továbbfejlesztési lehetőség, hogy kiegészítsük a struktúrát olyan opcionális elemekkel, melyek egyrészt teljesen leírják az éles adatbázis szerkezetét, másrészt pedig lehetővé teszik az adatmigrációknak pusztán a konfigurációs fájlok különbségén alapuló automatizálását. Természetesen ez egyáltalán nem triviális, hiszen néha meglévő adatok átstrukturálására van szükség, illetve lehetnek a két konfigurációs állapot közötti történetben elbújó lényeges módosítások. Úgy vélem, mindkét probléma kezelhető, jövőbeli terveim közé tartozik egy ilyen migrációs keretrendszer kialakítása.

[3] Ilyen megközelítést használ például a *helm* nevű deployment eszköz.

3. fejezet

A virtuális adattár építőkövei

3.1. Alapfeltevések

Bármilyen implementáció legyen is mögötte, a storage API egy relációs adatbázist ír le. Tehát ahhoz, hogy a virtuális adatok funkcionálisan egy valódi (csak-olvasható) relációs adathalmaz képét adják, nem kell más, mint hogy megfelelő viselkedéssel elérhetők legyenek a storage API-n keresztül. A megfelelőség ez esetben két dolgot jelent:

1. **Lekérdezések közötti konzisztencia:** felépíthető egy olyan tényleges immutábilis relációs adatbázis (M), hogy minden lehetséges relációs lekérdezés esetében, amely a közös sémára (S) értelmes, a virtuális és a tényleges adatbázis esetében visszaadott eredménytábla megegyezik.
2. **Megkötések kielégítése:** az S séma teljesíti a virtuális adatbázis felhasználói konfigurációját (C), valamint az M-ben szereplő adatok tulajdonságai illeszkednek a C-ben leírt megkötésekre.

A virtuális adatokhoz a legalsó szinten a storage API megfelelő hívásaival férünk hozzá. Ezek a speciális hívások szűk keresztmetszetet képeznek, hiszen ezek szimulálják például a közvetlen adatelérést. Ha ezen hívások performanciája nagyságrendileg (de legalább aszimptotikusan) összemérhető a tényleges adatbázisokéval, akkor az erre épülő lekérdezésfuttató és egyéb rétegek már a tényleges adatbázisoknál megszokott módon és nagyságrendi teljesítménnyel tudnak működni.

A lekérdező műveletek esetében két különösen fontos hozzáférési módot kell kiemelni:

1. rekordok véletlen elérése (random access)
2. adott érték előfordulásainak keresése egy oszlopban (reverse index)

Ha e két hozzáférési mód hatékony, akkor már a lekérdezések jelentős részénél elérhető a tényleges adatbázisokéval összemérhető performancia.

A storage API-ban úgy definiáltuk a `TableIndex` interfészt, hogy a keresésen kívül még néhány további funkciót is megköveteljen, például a rendezést és a *NULL* értékek kezelését. Ahol szükséges, külön kitérek majd az ezzel kapcsolatos problémákra.

3.2. Segédtípusok

3.2.1. Hatékony számítások

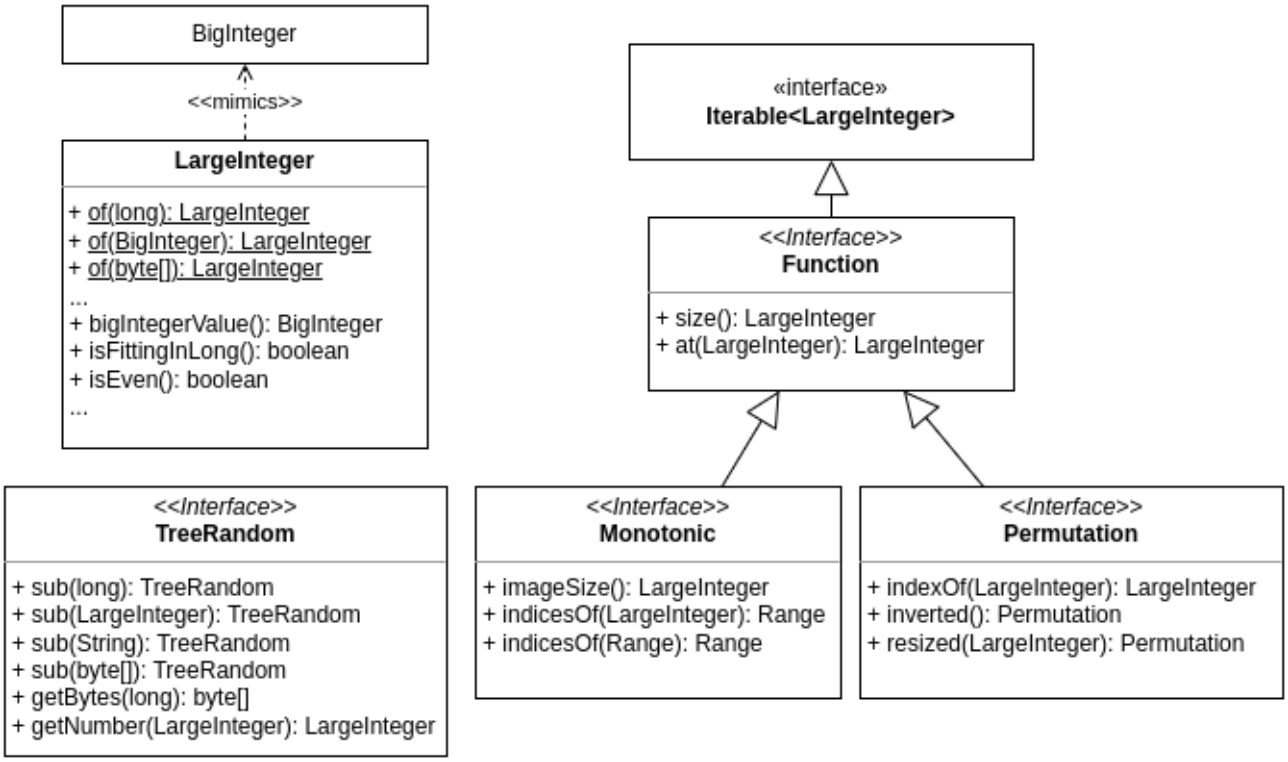
A hagyományos, tényleges adatbázisokban az egyik szűk keresztmetszetet az adatok fizikai elérése jelenti. Esetünkben ezt a virtuális értékkiosztások fenti függvényei helyettesítik. A blokkbetöltés, keresési index-struktúrák, materializált nézetek frissítése és hasonló problémák helyett a virtuális adattárban a visszaadandó adatok hatékony kiszámítását kell megoldani, ami általában az oszlopok értékeinek on-the-fly produkálását és keresését jelenti.

Az on-the-fly értékszámítás során gyakran fordulnak elő átmeneti óriás számok, ám nehezen jósolható előre, hogy pontosan mikor. Például eleve egy értékkészlet is lehet óriási (gondoljunk például a `\w{30}` reguláris kifejezés alapján kitöltött oszlop lehetséges értékeinek számára), miközben szükségünk van bármely pozíció közvetlen hivatkozására. Az adatbázis méretére vonatkozóan nem akartam megkötéseket tenni. Nem lett volna célszerű explicit ellenőrzések tömegét beépíteni a problémás számítások, potenciális túlcsordulások földerítésére. Továbbá az egész számokat egységesen szerettem volna kezelni, függetlenül azok méretétől. Így a beépített `BigInteger` típust egy idő után elvetettem, mivel az kisebb számokra a `long` típushoz képest rendkívül rossz performanciájú.

3.2.2. A fő segédtípusok áttekintése

Mielőtt rátérnék a storage API interfészeinek tárgyalására, röviden áttekintem azokat a segédinterfészeket, amelyek a későbbiekben nélkülözhetetlenek lesznek majd. Természetesen ezek az esszenciális típusok apró szeletét adják csak a teljes architektúrának.

A következő osztálydiagram összefoglalja ezeket a kiemelt interfészeket:



3.1. ábra. Néhány segédinterfész, melyek alapvető szerepet töltenek be az architektúrában

3.2.3. A `LargeInteger` típus

A hatékony on-the-fly számításokhoz szükségesnek mutatkozott egy olyan egész típus megalkotása, mely kis és óriás számokra egyaránt működik, de egyúttal a kis számokon nagyon hatékony (ellentétben a beépített `BigInteger` típussal).

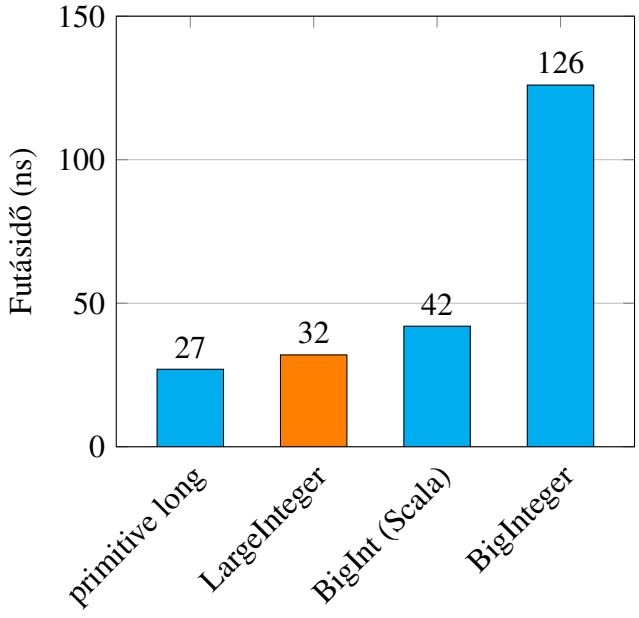
Az ecélból készített (egyébként teljesen általános) `LargeInteger` típus valójában egy absztrakt osztály két privát leszármazottal: az `ImplSmall` a `Long.MIN_VALUE`-tól a `Long.MAX_VALUE`-ig terjedő intervallumra (kis számok) optimalizálva, az `ImplBig` pedig az ezen kívül eső értékekre (nagy számok) specializálva.

Az `ImplSmall` egy primitív `long` értéket tárol. Bizonyos műveleteknél olcsó ellenőrzéssel előre megállapítja, hogy biztos-e, hogy nem lesz túlcsordulás; ha biztosan nem lesz, akkor a primitív műveletet végzi el, különben a `BigInteger` segítségével számol. Más műveletek esetében először végrehajtja a primitív műveletet, majd ez alapján ellenőrzi, hogy történt-e túlcsordulás. Például nem zéróval történő szorzás után az eredményt visszaoszthatjuk a szorzóval, és ha nem egyezik a szorzandóval, akkor tudjuk, hogy túlcsordulás volt. `BigInteger` eredmény esetén ellenőrzésre kerül, hogy a `long` értéksávjában van-e, ez esetben is egy `ImplSmall` kerül visszaadásra.

Az `ImplBig` mindig a `BigInteger`-en keresztül végzi a műveleteket. Hasonlóan a másik implementációhoz, szintén utóellenőrzi, hogy az eredmény kis szám-e. Egy `LargeInteger` szám akkor és csak akkor csomagolódik `ImplBig`-ként, ha kívül esik a `long` értékkészletén.

A `LargeInteger` a `BigInteger` összes metódusát replikálja, illetve számos további kényelmi metódust ad hozzá. A `LargeInteger` konstruktora kívülről nem elérhető, ehelyett a statikus `of()` metódusokon keresztül lehet példányokhoz jutni. Feltesszük, hogy a zérusközeli értékek a leggyakrabban használtak (gondoljunk például olyan triviális számokra, mint 0, 1, 2, 10), az ezekhez tartozó példányok statikusan tárazva vannak a duplikációk elkerülése céljából. Ez nem csak a memóriahasználatot csökkenti, de esetenként a performanciát is növelheti (például kihasználhatjuk, hogy egyenlő értékek egyúttal egyenlő példányokat jelentenek).

A Scala standard könyvtárának `BigInt` típusa szintén a `long` és `BigInteger` aritmetika váltogatásával oldja meg, hogy az óriás számok támogatása mellett kis számokon viszonylag gyorsan fusson[33]. A két megoldás performanciában hasonló, a számunkra releváns esetek egy részében (kisebb számok, sokféle művelet) a `LargeInteger` valamivel gyorsabb. A Scala `BigInt` (főként kompatibilitási okokból) nem használ két különböző típust, hanem egyetlen adatstruktúrába sűríti a két esetet, így enyhén több memóriát is használ. ❌❌❌



3.2. ábra. A `LargeInteger` teljesítményének összehasonlítása egy kis számokon végzett összetett számítás átlagos futási ideje alapján

3.2.4. Hierarchikus véletlengenerátorok

Az adatbázist egy hierarchikus képződményként is fölfoghatjuk, ahol a sémákon belül táblák, azokon belül oszlopok, azokon belül pedig sorindexekkel megjelölt mezők foglalnak helyet. Végül a mezők értéke is lehet strukturált adat, melyben a rétegződés folytatódik. Azt szeretnénk, hogy az értékek a hierarchiában bárhol látszólag nagyjából *véletlenszerűen* álljanak elő, ugyanakkor determinisztikusan, egy globális kiinduló seed alapján. Ezért szükségünk van egy olyan véletlengenerátor típusra, mely teljesíti a következő követelményeket:

- tetszőleges számú bitet képes generálni, mindig ugyanazt a (végtelen) bitsorozatot adva vissza
- egy kulcs bájtsorozat megadásával visszaad egy alpéldányt (eggyel alacsonyabb szint)
- ugyanazzal a kulcs bájtsorozattal egyenlő példányt ad vissza

Valamint lehetőleg az alábbi gyenge követelményeket:

- véletlenszerű kimenetet ad
- eltérő példányoknál jellemzően független, különböző kimenetet ad

Az ezeknek való megfelelés a `TreeRandom` interfészen keresztül történik.

Legtöbbször számokat akarunk előállítani, ezért az interfészhez adtam a `getNumber()` default metódust, mely a paraméterül kapott számnál kisebb nem negatív egészet ad vissza. Ehhez annyi generált bitet használ, amennyi a szám elfogulatlan kiválasztásához szükséges (rejection sampling).[1]

A fő implementáció hashelésen alapul, és magának a hashelésnek a minőségével skálázható. Minden ilyen tároló tartalmaz egy kulcs-bájtsorozatot. A root példánynál ez a root seed alapján áll elő. Alpéldány előállításakor az eredeti objektum kulcs-bájtsorozatához egy szeparátorbájt majd a kapott kulcs alapján generált bájtsorozat kerül hozzáfűzésre, így áll elő az alpéldány kulcs-bájtsorozata.

Amikor az adott példányból bitek kerülnek lekérdezésre, az első néhány bájt a kulcs-bájtsorozat hashelésével áll elő. A további bitek ennek derivátumai, aminek legegyszerűbb algoritmusa, ha a hash alapján előállított seeddel inicializált `Random` példánnyal rendre visszaadott bájtokat használjuk.

3.2.5. Monoton méretigazítás

Gyakran egy adott értékkészlet elemeiből akarunk képezni egy N hosszúságú rendezett listát. Az eloszlás (beleértve, hogy valamely érték egyáltalán előfordul-e) teljesen implementációfüggő lehet.

[1] Egy alapértelmezett limittel, hogy implementációtól függetlenül ne alakulhasson ki végtelen ciklus.

Mivel az értékkészlet maga is egy n hosszúságú lista, a további leképezés-kompozíciók lehetővé tétele érdekében rugalmasabb, ha inkább az értékindexeket, azaz a $[0..n)$ egész-intervallumot használjuk.

Követelmény, hogy le tudjuk kérdezni az alábbiakat:

- egy adott pozícióra mely értékindex került
- adott értékindex mely sávba került

A rendezettség miatt a második kérdésre mindig egy összefüggő sáv a válasz. Ha az értékindex nem szerepel, akkor ez egy zéróhosszú sáv azon a helyen, ahová a rendezés szerint beszúrható lenne.

Értelemszerűen értéksávokra is tudunk keresni, hiszen az értéksáv szélső értékeinek értékindexeire keresve megkapjuk a képzett sáv széleinek sorindexeit is.

Ezeket az elvárásokat formalizálja a `Monotonic` interfész.

A `Monotonic` interfész példánya egy függvényt reprezentál, amely az első n nemnegatív egész számot értékkészletként nem monoton növekvő módon az első N nemnegatív egész számhoz rendeli. Ez arra használható, hogy egy n elemű, rendezetten tárolt értékkészletet egy N méretű adatbázistábla oszlopához igazítsunk méretben. Tehát a példány az $M(i) = m$ függvényt valósítja meg, ahol $0 \leq i < N$, illetve $0 \leq m < n$, továbbá $i_1 < i_2 \implies M(i_1) \leq M(i_2)$. Ugyanakkor indexelt elérést is biztosít, azaz lekérdezhető, hogy egy adott érték mely indexsávhoz rendelődik, azaz megvalósítja az $M^{-1}(m) = [i, j)$ függvényt is, ahol $0 \leq m < n$, illetve $0 \leq i \leq j \leq N$, és teljesülnek az alábbiak:

- $x < i \implies M(x) < M(i)$
- $i \leq x < j \implies M(i) = M(x).$
- $i < j \leq x \implies M(i) < M(x)$

Az indexelt elérés az `indicesOf(value)` metóduson keresztül biztosított, amely egy `Range` objektumot ad vissza. A metódus egy variánsa egy `Range` példányt vár, értelemszerűen általánosítva az előbbit.

A legegyszerűbb implementáció a diszkrét lineáris vetítés. Az n hosszúságú értékkészletet ekkor úgy vetítjük az N listahosszra, hogy az i-edik helyre a $\lfloor \frac{n \cdot i}{N} \rfloor$ pozíciójú érték kerül.

A költségesebb, jobb minőségű implementációk pszeudovéletlen mintavételezésen alapulnak. Ehhez rekurzívan osztjuk föl az értékkészletet, miközben a felosztás fája determinisztikus, mindig ugyanaz. A fa minden elágazásához tartozik egy mintavételezés, mely eldönti, hogy az értékkészlet szeletei a céllista mely szeleteire vetüljenek. A mintavételezéshez jelenleg az Apache Commons Maths

könyvtár `BinomialDistribution` osztályát használom, melyet a `TreeRandom` példányból generált seeddel hívva determinisztikus eredményt tudok előállítani[34].

A mintavételezéses eljárás többféleképpen is továbbszofisztikálható. Egy ilyen lehetőség a szürjektív eredmény biztosítása, amikor szeretnénk, ha minden érték előfordulna (a zajosan monoton értékkiosztásoknál ennek lesz egy további alkalmazása). Illetve az is lehetséges, hogy konkrét táblázatunk van, mely minden értékkészletbeli értékre tartalmazza annak várható előfordulási gyakoriságát. Ezeket az előre definiált gyakoriságokat használhatjuk a mintavételezésben, de akár közvetlenül is alkalmazhatjuk egy diszkrét lineáris vetítéssel.

3.2.6. Permutációk

A `Permutation` interfész megvalósításához olyan eljárást keresünk, amellyel változó (és akár óriás) tartományon értelmezett, seed szerint variálható permutációfüggvény és annak inverze előáll, ahol mindkét függvény hatékony. Az inverz függvényhívást az `indexOf(value)` metódus deklarálja.

Triviális implementációk

Megjegyzem, hogy triviális implementációként használhatjuk egyszerűen az azonosságfüggvényt is. Ekkor nem keverednek az adatok, cserébe viszont performanciában verhetetlen.

Gyorsan számolható, de ránézésre már jó keveredést adó permutációt nyerünk, ha kihasználjuk, hogy az $ai \equiv p \pmod{n}$ lineáris kongruencia i-re és p-re nézve egy permutációt ad, ha a és n relatív prímek:

$$\sigma_L^{n(a,b)}(i) = (ai + b) \bmod n$$

Az a és b számot előre, a seed alapján határozzuk meg. A b eltolás értéke tetszőleges lehet. Az a szám relatív prím kell legyen n-re nézve. A megkereséséhez először választunk egy jelöltet, majd megvizsgáljuk (például az euklideszi algoritmussal), hogy relatív prím-e n-re. Ha igen, kész vagyunk. Ha nem, a számot tároló `LargeInteger` példány `nextProbablePrime()` metódusát hívva előállítjuk a következő jelöltet. Ez a háttérben a `BigInteger` osztály ugyanilyen nevű metódusát hívja, amely a Java beépített heurisztikus megoldása a következő valószínű prím megkeresésére.

Feistel-hálózatok alkalmazása

Az invertálható, seed alapján újrakeverhető, változtatható méretű permutációk könnyen analógiába állíthatók kriptográfiai titkosítófüggvényekkel. Ezek dekódolhatók, kulcs alapján újrakeverhetők,

illetve általában változtatható blokkméretűek. Felmerül a gondolat, hogy közvetlenül egy ilyen kriptográfiai módszert alkalmazzunk a permutáció megvalósításához. Olyan megoldásra van szükség, amelynél a blokkméret tetszőlegesen megadható, és kellően flexibilis, skálázható.

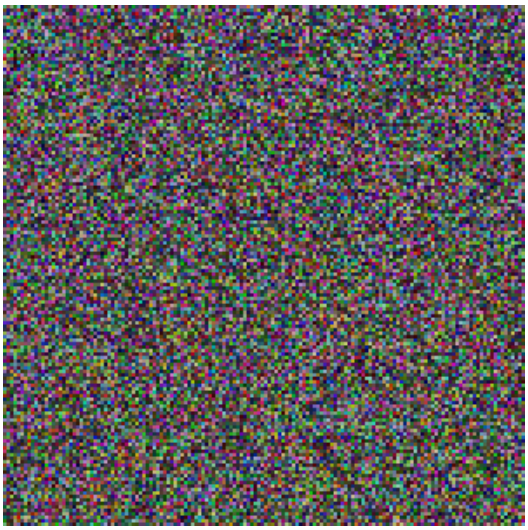
A Feistel-hálózatok ilyenek, így összetett titkosító eljárások keretrendszereként igen népszerűek[35]. A Feistel-hálózat többkörös eljárás, ahol minden körben az adat egyik felének bitjeit módosítjuk. Ez alapesetben megköveteli a páros blokkméretet, de amint látni fogjuk, ezen lehet enyhíteni. Az egyes körökben felváltva a bitsorozat egyik (bal vagy jobb) felét változtatjuk, nevezzük ezt célhelynek, a másik fél tartalmát pedig forrásbiteknek. Egy-egy körben a forrásbitekre meghívunk egy az adott körhöz rendelt hossztartó hash-függvényt, és az így kapott eredményt XOR-ral ráfésüljük a célhelyre. Az XOR tulajdonságai miatt ez reverzibilis (bármi legyen is a hash-függvény), újraalkalmazva az eredeti bitsorozatot kapjuk vissza (a forrásbiteket ezen a ponton még nem írtuk felül). Tehát a Feistel-hálózattal kapott kód visszafejtése úgy történik, hogy a lépéseket visszafelé újra végrehajtjuk. Általában nem szükséges, hogy az egyes körökhöz különböző hash-függvényt használjunk, bőven elég, ha a páros és páratlan körökhöz eltérő függvényt alkalmazunk (szigorúan még ez sem lenne szükséges).

A Feistel-hálózatokkal való kísérletezés során észrevettem, hogy a páros blokkméret követelménye enyhíthető a következőképpen. Páratlan blokkméret esetén vegyünk még egy zéró bitet az adat végéhez, amivel biztosítjuk, hogy a jobb oldali bitmező is a megfelelő méretű legyen. Minden olyan kör végén, ahol a jobb oldali biteket írtuk felül, az utolsó bitet ezután nullázzuk. A legvégén pedig csak az utolsó bit nélküli részt adjuk vissza. Az utolsó bit ekkor minden művelet előtt és után is fixen zéró, ami természetesen visszafelé haladva ugyanúgy teljesül, tehát a dekódolás is determinisztikus marad. Ha egyúttal páros számú kört követelünk meg, akkor az utolsó kör végére garantáltan ugyanabba a fázisba kerülünk, mint az első kör kezdete előtt; vagyis a dekódolás folyamata teljesen egyezni fog az elkódolással, ami egyszerűsíti az eljárást. (→ Algoritmus: Feistel-hálózat last-zero változat)

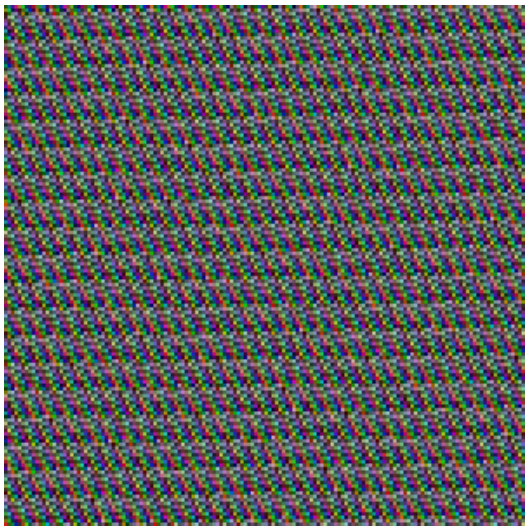
Persze, még ha a páros blokkméret követelményét így sikerült is eliminálni, továbbra is adott a probléma, hogy az n blokkmérethez mindig 2^n méretű permutáció tartozik. Vagyis a nem 2-hatvány méretű permutációkhoz méretigazítás szükséges. Ezt rejection sampling alkalmazásával érjük el. Az adott n mérethez vesszük azt a legkisebb N 2-hatványt, amely n-nél nem kisebb. Amikor egy adott i értékre a Feistel-hálózattól az intervallumon kívül eső p-t kapunk, azaz amelyre $n \leq p\ (< N)$, akkor újra meghívjuk a permutáló függvényt p-re. Addig ismételjük a hívást, míg végre n-nél kisebb szám nem születik. A páratlan blokkméret lehetősége a hatékonyság szempontjából is fontos, mivel jelentősen csökkenti a Feistel-hálózatra történő újrahívások számát.

A rejection sampling nem minden implementációnál hatékony, de pontosan ezért vezettük be a `resized(LargeInteger)` metódust magán a `Permutation` interfészen. Így minden implementáció maga döntheti el, hogyan kell átméretezni. Egy extrém eset például, amikor a $\sigma_{+1}^{N}(i) = (i+1) \bmod N$ permutációt n-re kell átméreteznünk. Rejection sampling módszerrel $i = n$ esetén $N-n$ számú extra iterációt kellene végrehajtani, ami nagy méret esetén rendkívül költséges. Helyette viszont egyszerűen áttérhetünk modulo n-re: $\sigma_{+1}^{n}(i) = (i+1) \bmod n$.

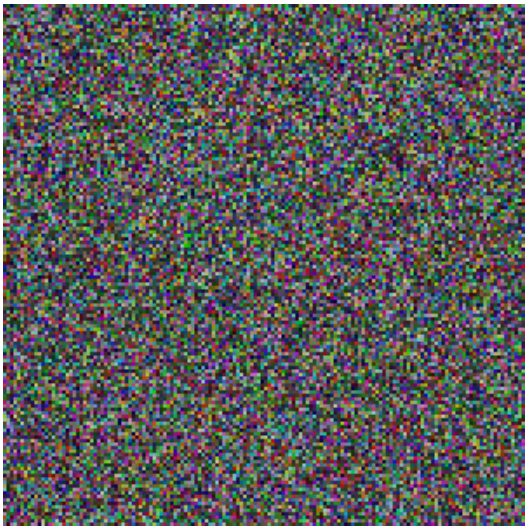
Az alábbi ábrán vizuálisan összehasonlítottam néhány permutáció-típus értékeloszlását. A pixel koordináta az indexből, a színárnyalat és a fényerő értékek az értékből vannak kalkulálva. Értelemszerűen a nagyobb összevisszaság erősebb véletlenszerűséget jelent. Felül a bal oldalon egy erős véletlenszerűséget adó FPE (Format Preserving Encryption) eljárás, a Rank-then-encipher[36], a jobb oldalon pedig egy egyszerű lineáris kongruencián alapuló permutáció kimenete látható. Alul két Feistel-hálózat kimenetét látjuk, melyek a fenti kettőhöz mérhető skálázódást mutatnak pusztán paraméterezés által; a bal oldalon egy kétkörös, erős SHA256 hashelést használóét, a jobb oldalon pedig egy egykörös, gyors XOR ellenőrzőösszeggel hashelőét. Ráadásul az XOR hash eredménye jobb, mint a lineáris kongruencián alapuló permutációé. (Forráskódért ld.: Függelék C)



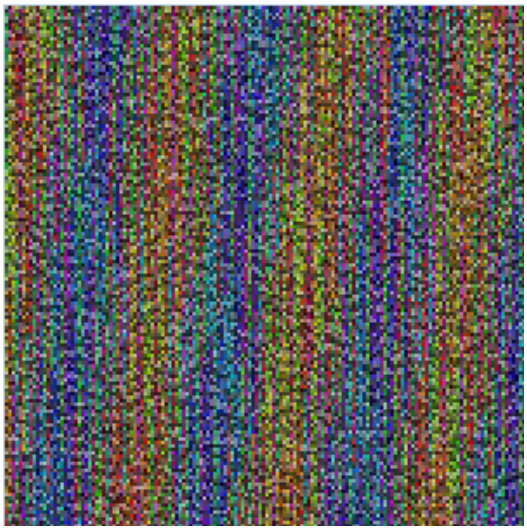
Rank-then-encipher FPE[36]



Modulo permutáció



Feistel-hálózat (2-round, SHA256)



Feistel-hálózat (1-round, fast hash)

3.3. ábra. A `Permutation` különféle implementációinak kimenetei

3.3. Egyoszlopos kereshető értékkiosztások

3.3.1. Általános megfontolások

Olyan értékkiosztási módszereket veszek most végig, amelyek biztosítják az alábbiakat:

1. hatékony pozíció-alapú elérés (random access)
2. hatékony kereshetőség valamilyen formában
3. hatékony rendezés
4. lekérdezések közötti konzisztencia

Az egyes esetekben megvizsgálom majd, hogyan teljesíthetők ezek az elvárások.

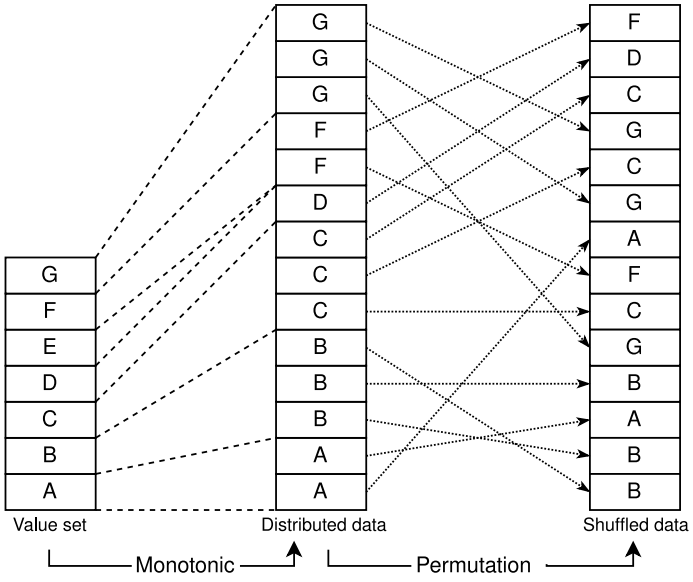
3.3.2. Keretrendszer az értékkiosztásokhoz

Alapesetben az adatokat oszlop-orientáltan fogjuk előállítani. Minden oszlophoz tartozik majd egy virtuális adatlista, melynek hossza egyenlő az adott oszlopot tartalmazó tábla hosszával. Opcionálisan szerepelhetnek benne *NULL* értékek, a többi érték típusának pedig kompatibilisnak kell lennie az oszlophoz megadott típussal.

Bár ez a megközelítés oszloponként független adatlistákra van szabva, valójában más jellegű értékkiosztási módszer is lehet mögötte, amennyiben az egy-egy oszlophoz tartozó listanézetek biztosítottak. Majd a több oszlopot érintő megkötések esetében ez lesz a helyzet. De lássuk először az egyoszlopos értékkiosztások lehetséges módszereit.

3.3.3. Általános kétlépéses értékkiosztás

Talán a leggyakoribb eset, hogy előre ismert véges értékkészlet elemeit szeretnénk viszontlátni az oszlopban, mégpedig összevissza. Fentebb már utaltam rá, hogy ez a legtöbb esetben megoldható két független lépéssel. Először egy `Monotonic` példány segítségével a kívánt célhosszra értelmezett függvényt állítunk elő, mely az értékkészlet pozícióira mutat (ezt a továbbiakban *rányújtás*nak nevezem majd). Majd az így előállt (esetlegesen *NULL* értékekkel kiegészített) listát egy `Permutation` példány segítségével elkeverjük.

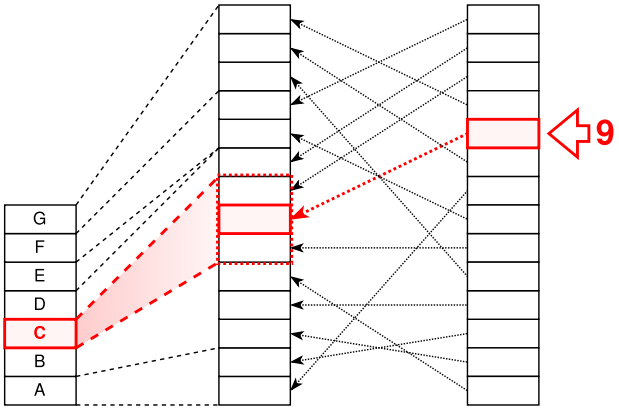


3.4. ábra. A kétlépéses értékkiosztás alapelve: egymás után végrehajtott visszafejthető disztribúció és permutáció

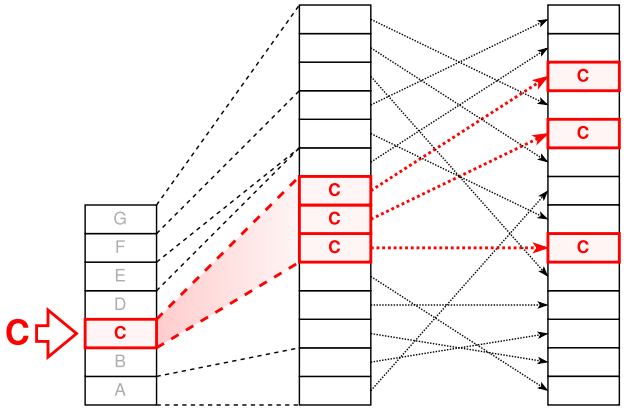
Láttuk, hogy mindkét függvénytípus interfésze úgy van definiálva, hogy relációként megfordítható. A `Monotonic` `indicesOf()` metódusaival egy értékindexnek, illetve -sávnak a céllistabeli sávjára tudunk rákérdezni. A `Permutation` pedig az `indexOf()` metódust biztosítja hasonló célra (illetve inverze maga is permutáció). Ezek használatával egy reverse index könnyen implementálható.

A még keveretlen rányújtott adathalmazbeli találati sáv határai egyúttal azonnal megadják a találatok számosságát, ami egyszerűbb `COUNT` lekérdezéseknél igen előnyös.

A következő két ábra szemlélteti az adatlekérés, illetve értékkeresés folyamatát:



3.5. ábra. Adatlekérés a kétlépéses értékkiosztásból



3.6. ábra. Keresés a kétlépéses értékkiosztásban: a rendezett értékkészlet és a leképezések megfordíthatósága biztosítja a hatékony keresést a virtuális listában

Kétlépéses értékkiosztás gyakoriságtáblázattal

Egyes esetekben előre ismert az értékek eloszlása, amit szeretnénk nagyjából vagy pontosan viszontlátni a céloszlopban. Ekkor egyszerűen a `Monotonic` fentebb tárgyalt konkrét eloszlásokat használó implementációnak egyikét kell használnunk.

A felhasználó számára biztosítani kell a gyakoriságok könnyű konfigurálhatóságát, ehhez érték-gyakoriság párokat kérünk be. A második érték egymáshoz képesti relatív gyakoriságot jelöl.

Zajosan monoton értékkiosztások

A rányújtásnak az előzőleg monoton függvény előállításához használt elve másféle értékkiosztáshoz is felhasználható, nevezetesen olyan (szigorúan vagy nem szigorúan) monoton adatsor előállításához, melynek egyes értékei összességében egy adott sűrűség szerint növekednek (vagy csökkennek; az

egyszerűség kedvéért most csak a növekedő esetről lesz szó), de lokálisan nagy az egyenetlenség. Ezt is két lépésben fogjuk megvalósítani.[2]

Az első lépés alapelve tehát hasonló az előbbi megoldás első fázisához. Ám itt nem lehetséges értékeket vetítünk ki a tábla hosszára, hanem a táblahossznyi alaphalmazt vetítjük majd ki egy diszkrét lehetséges értékkészletre. Minden sorindexhez hozzárendelődik egy (a szigorú monotonitás elvárása esetén nemüres) dedikált sáv az értékkészletből.

A második lépés választ egy értéket a sávból. Nem szigorúan monoton értéksor esetén megengedjük az üres sávot is, és ilyen esetekben mindig a rákövetkező elemet választjuk.[3]

Az érték elérése ekkor úgy történik, hogy először lekérjük az értéksávot a rányújtó függvénytől, majd meghívjuk az értékválasztó függvényt, melynek paraméterei az értéksáv, a sorindex és a `TreeRandom`-ból vett seed lesznek. Egy kézenfekvő megvalósítás, hogy a seed alapján inicializált véletlengenerátorral generáltatunk egy véletlen értéket, ami a sávba esik.

Értéksávra való kereséskor vesszük a minimális és a maximális keresett értéket, és az inverz rányújtást használva megkeressük a megfelelő sorindexeket, amelyek sávjához az érté tartozik. Ezen sorindexek feszítik majd ki a találati sorindexsávot. Hogy az alsó érték beletartozik-e, annak eldöntéséhez le kell generálni az alsó sorindexhez a konkrét értéket, és ellenőrizni, hogy nagyobbegyenlő-e, mint a keresett alsó érték. A felső érték esetében hasonlóan kell eljárni.

A rendezés triviális, mivel az értékek eleve rendezettek.

Ezzel az eljárással nem csak zajossá tudtuk tenni az eloszlást, de a monotonitás garantálása mellett megengedtük, hogy az értékek esetlegesen csomósodhassanak, illetve elméletben tetszőlegesen eltávolodhassanak attól a helytől, amit egy szigorúan egyenletes kiosztás esetén vettek volna föl.

Ez az értékkiosztás különösen alkalmas időbélyegek szimulálására, amikor az események általános sűrűsége adott, de véletlenszerű, zajos kimenetet szeretnénk látni.

[2] A két lépés általánosítható egy általános sűrűségfüggvény és egy zajfüggvény összegévé, ahol a sűrűségfüggvény meredeksége és a zajfüggvény kilengése közötti megfelelő viszonyt kell biztosítani, hogy az értékek ne ugorják át egymást. Itt most nem foglalkozom ezzel az általánosabb kerettel.

[3] Ebben az esetben explicite ki kell zárni, hogy az utolsó sáv üres legyen. Ez legtermészetesebben egy logikai paraméter beiktatásával érhető el, amelyet a rekurzió során mindig csak a felsőbb sávra küldünk tovább `true` értékkel (a többire `false` értékkel), alapértelmezett értéke `true`. Ha tehát `true` értéket kaptunk, biztosítani kell, hogy az aktuális felsőbb sáv ne legyen üres.

3.3.4. *NULL* értékek kezelése

Általánosan, bármely értékkiosztási módszerhez könnyen hozzáilleszthető a *NULL* értékek támogatása. Ekkor a rányújtást a tábla méreténél kisebb intervallumra végezzük, a maradék helyeket pedig egyszerűen *NULL*-nak tekintjük. A permutáció alkalmazása után a *NULL* értékek is szétszórtan szerepelnek majd.

Nem szükséges ez a plusz kompozíció, ha a fentiek magába az eredeti értékkiosztásba is könnyen beépíthetők. Akár beépítetten, akár plusz kompozícióval van megvalósítva a *NULL* értékek kezelése, természetesen figyelni kell a speciális kezelésre a keresés-rendezés során.

A fentiek fényében az egyes értékkiosztások tárgyalásánál a *NULL* értékek kezelésével nem foglalkozom.

3.3.5. Egyszerű értékkiosztások

Tényleges adatbázisokban némely esetben meglehetősen következetes módon szerepelnek az értékek az oszlop értéklistájában. Természetesen az ilyen esetek szimulálása a legegyszerűbb.

Triviális eset, ha egy oszlop mezői egy közös konstans értéket tartalmaznak.

A szekvenciális oszlopoknál az érték megegyezik a rekord 1-től indított sorszámával, de sok egyéb oszlop is generálható szekvenciális alapon (feltehetjük, hogy nem történt még releváns módosítás, ez aztán az írási réteg használatával korrigálható). Az ilyen esetekben az n-edik érték lekérése egy egyszerű lineáris függvénnyel számolható, és az érték keresése is hasonlóan történik.

Időbélyegeket is generálhatunk hasonló módon, amennyiben megengedhető, hogy a szimulált időadatok között egyenletes időközök legyenek.

3.3.6. Unique értékkiosztás

Elsődleges vagy más egyedi kulcsok szimulálásakor fontos, hogy ugyanaz az érték ne forduljon elő kétszer az értéklistában. Ennek természetesen előfeltétele, hogy a lehetséges értékkészlet nagyobb legyen, mint az oszlop hossza (*NULL* értékek nélkül).

Egyszerűen megvalósítható módszer, amikor egy permutációt alkalmazunk az értékkészlet fölött, és a permutált lista oszlophosszúságú prefixét vesszük értéklistaként. Ekkor a random elérés triviális: csak a permutációt kell visszafejteni. A keresés a rendezett értékkészletre alapozható, a találatokat a permutációval megforgatva kell listázni, kihagyva a prefixen kívül eső értékeket.

Ha azonban az értékkészlet mérete lényegesen nagyobb, mint az oszlophossz, jellemzően nagyon sok értéket kell átugranunk, amely a prefixen kívül esik. Ilyenkor sokkal hatékonyabb, ha az általános kétlépéses értékkiosztást használjuk egy megfelelő `Monotonic` implementációval, akár egyszerű diszkrét lineáris vetítéssel.

3.3.7. Reguláris kifejezésből képzett értékkészletek

Reguláris kifejezésből generált szó-fák

Karakterláncokból álló értékkészletnek egy reguláris kifejezés (minta) alapján történő generálása talán a legáltalánosabb és legrugalmasabb módja az értékek definiálásának. Emiatt ezzel a résztémával fogok a legrészletesebben foglalkozni.

Véletlenszerű karakterláncok reguláris kifejezés alapján történő generálására számos könyvtár elérhető a Java környezethez (például a *xeger*[37] és a *generex*[38]). Ezek olyan automaták, melyek az állapotgráfot véletlenszerű választásokkal járják be. Ez a megközelítés önmagában nem alkalmas arra, hogy az összes illeszkedő karakterláncból álló virtuális értékkészletet a számunkra megfelelő módon implementálni lehessen vele. Nem tudjuk ugyanis lekérni a rendezett értékkészlet *n*-edik elemét, és nem tudjuk kikeresni, hogy adott érték mely pozíción található az értéklistában.

Lássuk, hogyan tudnánk valami módon mégis egy rendezett értékkészletet implementálni. Egy olyan memóriagazdaságos adatstruktúrát kell előre legenerálni, melyet futásidőben a gyors lekéréshez és kereséshez használhatunk majd. Első megközelítésben a reguláris kifejezéssel kapcsolatban az alábbi megszorításokat tesszük:

- a minta csak konstans karakterek, karakterosztályok, csoportok és alternációk szerepelnek
- a rendezés karakterenkénti, azaz nincs másodlagos, harmadlagos stb. összehasonlítási szempont

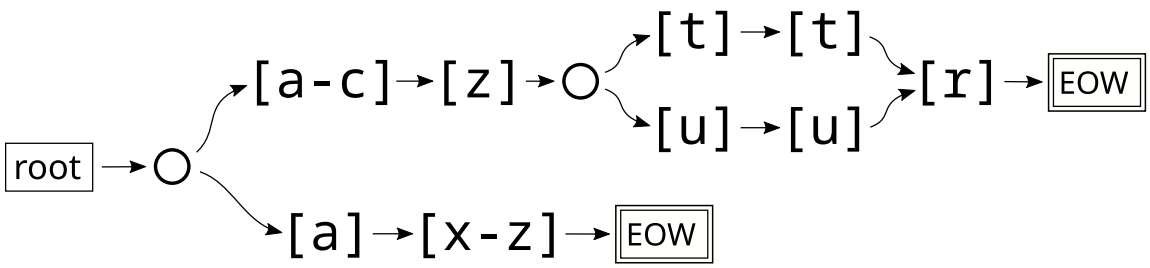
Ilyen megkötések mellett a reguláris kifejezéshez problémamentesen rendelhető egy irányított körmentes szógráf (DAWG), melyben egy érték nélküli kiinduló elemből érjük el a karaktereket reprezentáló csomópontokat, és kizárólag a szóvég csomópontokon (EOW) terminálhatunk, melyekből már nem érhető el további csomópont.[39]

A gráf lényegileg egy prefix fa, ahol egyes azonos részfákat újrahasznosítunk. Ezért a továbbiakban az egyszerűség kedvéért néha faként fogok a gráfra hivatkozni. A gyökértől az adott részfáig vezető útvonalra néha az adott részfa *prefixeként* fogok hivatkozni. Az újrahasznosítás miatt a fa tárhelyigénye a reguláris kifejezéshez képest lineáris. A szóvég csomópontok lesznek a fa levelei. A

prefix fa könnyen felépíthető a reguláris kifejezés szintaxisfájából (AST). A gráf tömöríthető úgy, hogy konstans karakterek helyett karakterosztályokat engedünk meg.

A felépítés két lépcsőben történik. Először egy nyers fát állítunk elő, ami közvetlenül reprezentálja a reguláris kifejezést. (→ Algoritmus: Nyers szófa összeállítása)

Például a `([a-c]z(tt|uu)r|a[x-z])` reguláris kifejezés nyers szófája így néz ki:



3.7. ábra. Egyszerű reguláris kifejezés nyers szófája

A nyers fa nagyon hasonlít egy szintaxisfához, a fő különbség, hogy az alternációk vége tovább van csatlakoztatva a külső szekvencia következő eleméhez tartozó csomóponthoz (a példában a `[t]` és az `[u]` csatlakozik így az `[r]` -hez). Ebből világos, hogy a nyers fa mérete és előállításának költsége a reguláris kifejezés méretével egyenesen arányos.

Látható, hogy a szekvenciákból láncolás lett, az alternációkból pedig egy több elágazással (gyermekelemmel) rendelkező üres csomópont. Az is látható, hogy néhány rákövetkező karakter duplikáltan szerepel egyes elágazásokban (például az ’a’ betű rögtön az elején).

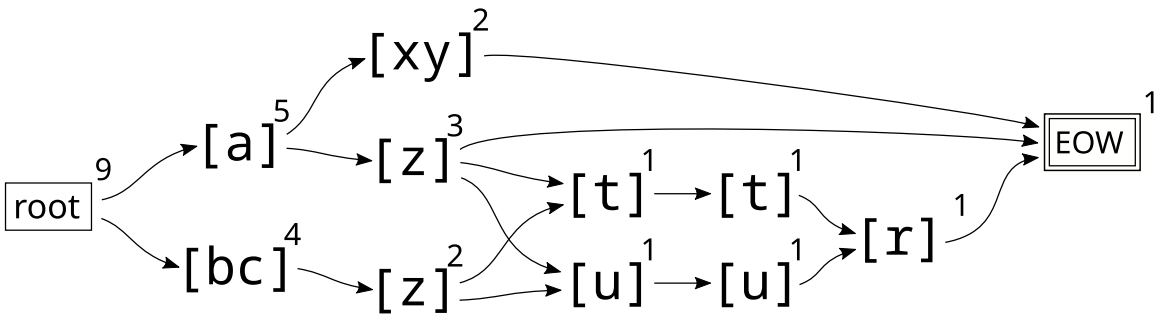
A közbeékelődő üres csomópontok és a duplikációk miatt a szófa alkalmatlan arra, hogy keressünk benne, illetve hogy rendezetten iteráljunk a csomópontokon. A probléma kiküszöbölésére második lépésben egy normalizálási eljárást hajtunk végre, mely után a gráfban adott csomópontból elérhető csomópontok listájára az alábbiak lesznek érvényesek:

- **explicit terminálás:** a lista opcionális eleme a szóvég csomópont, ami kötelező, ha egyébként nincs más a listában; kivételt képez maga a szóvég csomópont, melynek soha nincs gyermeke
- **egyértelműség:** a lista elemei ezen kívül diszjunkt karakterosztályok
- **rendezettség:** bármely karakterosztály listaelem minden tartalmazott karaktere későbbi, mint bármely megelőző karakterosztály listaelem bármely tartalmazott karaktere; a szóvég csomópont mindig első elem

Ezek a szabályok biztosítják, hogy a gráf bejárása rendezett sztringeket ad, valamint egy trie-szerű adatstruktúrát kényszerítenek ki, amelyben hatékonyan tudunk keresni.

Az átalakítás rekurzív módon történik. Ahol egymásnak alternatíváját jelentő két karakterosztály között átfedés vagy átugrás van, ott mindkét karakterosztályt szét kell bontani úgy, hogy a kapott karakterosztályokból képzett halmazra a fentiek érvényesek legyenek. Például ha az `[a-c]`, `[cf]` és `[c-e]` jelennek meg alternatívaként, akkor ezek szétbomlanak az `[ab]`, `[c]`, `[de]` és `[f]` karakterosztályokra, és átveszik a korábbiakból elérhető csomópontokat. A `[c]` karakterosztály így egy hármas elágazást vesz át a három eredeti csomópontból. A fát egyúttal célszerű úgy továbbalakítani, hogy megszüntetjük az alternációk miatti üres értékű, pusztán az elágazás kedvéért felvett csomópontokat, ez természetesen az egyenlő csomópontok részfáinak összefésülését vonja maga után. Nevezzük az így nyert gráfot a reguláris kifejezés *kompakt szófájának*. (→ Algoritmus: Nyers szófa kompakttá alakítása)

A `([a-c]z(tt|uu)r|a[x-z])` reguláris kifejezés fenti nyers szófájából az alábbi kompakt szófát nyerjük:



3.8. ábra. Egyszerű reguláris kifejezés kompakt szófája

Minden csomóponthoz hozzárendeltük a hozzá tartozó részfa leveleinek effektív számát, ami rekurzióval könnyen számítható. Ez az információ már elegendő ahhoz, hogy kikeressük az n-edik elemet, illetve hogy megkeressük egy adott karakterlánc (megtalált vagy beillesztési) pozícióját.

A kompakt szófában megtalálható különböző részfák száma a nyers szófa különböző részfáinak számához képest exponenciálisan megnőhet. Induljunk ki ugyanis egy n elemű C karakterhalmazból, és képezzük rendre a $C_1, C_2, \cdots, C_{n-1}, C_n$ karakterosztályokat úgy, hogy C_i-t C-ből az i-edik karakter kihagyásával nyerjük. Képezzük azt az n ágú alternációt, ahol az i-edik ág $C_i\{n\}$ alakú. Világos, hogy a reguláris kifejezés által megengedett összes $n-1$ hosszú prefixekben szereplő karakterek lehetséges halmazai kiadják C összes legfeljebb $n-1$ elemű halmazait, ami exponenciálisan növekszik n növelésével. Minden ilyen halmaz az alternáló ágak egyedi kombinációját azonosítja (ebben a konkrét

esetben ez azt jelenti, hogy különböző lesz a lehetséges utolsó betűk halmaza), tehát egyúttal legalább ennyi különböző részfa is van.

Az előbbi reguláris kifejezés lényegileg azt fejezi ki, hogy a karakterláncban nem szerepelhet C összes eleme, ezen kívül bármit megengedünk. Az ehhez hasonló konfigurációk szerencsére meglehetősen életszerűtlennek tűnnek. Az ilyen szempontból problémás alternációkat akár előre is detektálhatjuk, és túl nagy becsült növekedés esetén expliciten hibát dobhatunk.

Most pedig nézzük hogyan tudunk engedni a fenti megszorításokon.

A repetíciós operátorok explicite kifejtés által kerülnek visszavezetésre a fentiekre. A kifejtés természetesen egyes esetekben lényegesen megnövelheti a fa méretét a bemenethez képest. Ennek kioptimalizálásával most nem foglalkozom.

Először a korlátlan repetíciót (*, + stb.) eliminálju rendre az alábbi szabályokkal:

1. $A\texttt{+} \longrightarrow A\texttt{\{1,\}}$
2. $A\texttt{\{}n\texttt{,\}} \longrightarrow A\{n\}A*$
3. $A* \longrightarrow A\texttt{\{}c\texttt{\}}$ (ahol c előre definiált konstans)

A fix repetíció egyszerű konkatenációvá alakul, az opcionális repetíció kifejtése pedig a következő rekurzív módon történik:

1. $R_1 = A\texttt{\{,1\}} \longrightarrow (A\,|\,)$
2. $R_n = A\texttt{\{,}n\texttt{\}} \longrightarrow (AR_{n-1}\,|\,)$ (ahol $n > 1$)

Az opcionális előfordulás (`?`) az R_1 esettel egyenlő.

A támogatott konstrukciókat könnyen tovább bővíthetjük az egyszerűbb horgonyokkal, mint például szövegvég (`$`) vagy szóhatár (`\b`). Ezeket a kezdeti feldolgozáskor úgy tekintjük, mint az átmeneti üres csomópontokat, tehát eliminálju és megjegyezzük őket, de utána sorban kikényszerítjük a teljesülésüket. A horgonynak ellentmondó gyermekelemeket eltávolítjuk, ha ezután nem marad gyermekelem, a teljesíthetetlen ágat jelző kivételt dobunk, a kivételt elkapó szint el fogja dobni az ágat mint lehetetlent, ami újabb kivételdobást eredményezhet, ha az ottani gyermekelemek emiatt elfogytak, és így tovább.

Unicode karakterláncok és többszintű rendezés reguláris kifejezésekkel

Az igazi nehézségek akkor kezdődnek, amikor elengedjük a karakterenkénti rendezés megszorítását (ami természetesen további többletköltségeket fog eredményezni, ezért érdemes opcionálissá, konfigurálhatóvá tenni). Erre például ékezetes karakterek vagy kis- és nagybetűk keveredése esetén lehet szükségünk. Ekkor ugyanis esetleg csak egy későbbi eltérés dönti el, hogy egy korábbi ékezet vagy nagybetűsség számít-e. Vagyis az egyszerű, karakterenkénti prefix-fa sem elég a rendezettség biztosításához.

Ha egy ilyen szofisztikáltabb rendezési elvet akarunk megvalósítani, az *Unicode Collation Algorithm* (UCA) szabványból érdemes kiindulni[40]. A UCA azonban rendkívül összetett, így egyelőre csak az alapvető szempontok implementálását fogom bemutatni. Ki-kitérek majd arra, hogyan lehet ezeket a későbbiekben bővíteni, továbbfejleszteni, hogy még részletesebben lefedje a szabvány által tárgyalt lehetőségeket.

A UCA mindenekelőtt definiál egy általános keretrendszert, mely egy sémát ad arra, hogyan kell többszintű rendezéseket megvalósítani Unicode karakterláncokon. A sémát implementáló konkrét algoritmusokat *kollációknak* nevezzük. Egy Unicode karakterlánc lényegileg Unicode codepoint-ok tömbjét jelenti (Java környezetben a `char` adattípus ilyen codepointot tárol). Egy kolláció egy-egy ilyen tömbhöz rendezési szintenként hozzárendel egy-egy kollációselem-listát (*Collation Element Array*). A kollációs elemek lényegileg codepointok, de a kolláció által definiált súlyozás szerint átszámozva. Tehát úgy is felfoghatjuk, hogy a kolláció a Unicode karakterlánchoz szintenként egy másik Unicode karakterlánc nézetet rendel, ahol a codepointok között egyúttal egyedi sorrendezést definiál. A UCA öt alapértelmezett rendezési szintet különböztet meg:

- **L1**: alapkarakterek (Base characters)
- **L2**: diakritikus jelek (Accents)
- **L3**: kis-/nagybetűsség (Case/Variants)
- **L4**: írásjelek (Punctuation)
- **Ln**: kódolási különbségek (Identical)

Például a normál magyar kollációban a ’`Kör`’ karakterlánc L1-es alakja ’`kor`’ ahol a diakritikus jel és nagybetűsség stb. ignorálásra (eltávolításra) került. Ellenben például a svéd nyelvben az ’`ö`’ betű az ABC végén teljesen különálló betűként szerepel, így a normál svéd kollációban a fenti karakterlánc L1-es alakja ’`kör`’, és a rendezésben is egészen máshogy fog viselkedni[41].

A UCA szerint két karakterlánc összehasonlításakor sorban vizsgálandók az egyes szintek, míg az egyik szerint különbséget nem találunk. Ekkor egyszerűen a kollációselem-listák elemenkénti összehasonlítása szerint kerül megállapításra a két karakterlánc egymáshoz képesti sorrendje.

Az *International Components for Unicode* (ICU) a Unicode Consortium hivatalos referenciaimplementációja általában a Unicode szabványhoz, beleértve számos konkrét kollációt a világ nyelveihez[42]. Konkrét szabályok kialakításához ebből érdemes kiindulni. Az ICU általában (beleértve a standard, európai és magyar kollációkat is) teljesen az L1 szinten kezeli az írásjeleket is (az egymás utáni több írásjel is külön-külön elsődleges karakternek számít). Ez logikusabb, mint az írásjeleket egyáltalán nem figyelembe venni a betűk vizsgálatakor, hiszen akkor a szavak tagolása sem történik meg (például az '`ab:cd`' előrébb kerül, mint az '`a:cd`'). Ezt is figyelembe véve, mi most csak az első három szinttel foglalkozunk, legalábbis kizárólag pozíciótartó összehasonlításokkal.

A kompakt szófa fentebbi implementációját a következő ötlettel tudjuk kibővíteni a többszintű rendezés támogatásához. Minden rendezési szinthez külön szófát készítünk, melyeket az eredeti szófa egyszerűsítésével nyerünk (az életszerűség csorbítása nélkül feltehetjük, hogy a reguláris kifejezés olyan, hogy ez problémamentesen megtehető). Ekkor az egyszerűsített fákban egy-egy maximális útvonalhoz 1-nél több eredeti karakterlánc tartozhat (az egyszerűsítés éppen ezt az összevonást jelenti az adott szinten egyenlőnek számító karakterláncok között). Ezeket a szófákat aztán (virtuális értelemben) egymás alá fűzzük, azaz effektíve minden előző szint EOW csomópontja után konkatenáljuk a következő szint fáját. Az összevont (például egybeolvadó karakterekhez tartozó) részfák többszörös számossággal számítanak, amit jelezni kell.

Az n-edik karakterlánc lekérése úgy történik, hogy az egymás alá fűzött fák rendszerében (ami maga is egy nagy faként értelmezhető) navigálunk. Az első fában normál módon keresünk, azzal a kiegészítéssel, hogy a szorzó csomópontok egyszerűen olyan részfák listájaként értelmezendők, melyek a szorzó csomópont gyermekelemeihez tartozó részfákkal egyenlők, azonban minden méretükben a szorzóval növelve. A levélhez érve a szorzók miatt egy 1-nél nagyobb számosságot kaphatunk. Ezt a számosságot a további szintek részfái fogják tovább-bontani. Az utolsó szint kivételével mindegyik fában megengedett (sőt, a dolog lényegéhez tartozik) a többszörös számosság.

Világos, hogy (a szorzókat figyelembe véve) egy-egy szint fája a teljes eredeti értékkészlet számosságával bír. Amikor azonban már a többedik fában kezdünk továbbmenni, valójában a fa olyan szűkített nézetét kell vennünk, amely kompatibilis az összes már előbbi szint fájában bejárt útvonallal. Például (magyarral kompatibilis kollációs szabályokat feltéve), ha az első szinten a

’tarol’ karakterláncot kaptuk, a második szint fája megengedheti a ’tarol’, ‘tárol’, ’táról’ stb. karakterláncokhoz tartozó útvonalakat, de mondjuk a (tegyük föl, eredetileg megengedett) ’tartó’ útvonalát már nem.

Ez a szűkítés a bemenettől függ. Vagyis nem tudjuk előre generálni, a futási idejű performanciát fogja befolyásolni. A előbb bemutatott adatszerkezet alkalmas arra, hogy ezt hatékonnyá tegye, de ehhez szükség van egy lényeges megkötésre, nevezetesen, hogy az egyszerűsítés pozíciótartó legyen, azaz egy adott karakterlánc összes szintbeli nézetének hossza egyezzen az eredeti hosszal, és bármely karaktere a vele azonos pozíción található eredeti karakter derivátuma legyen. Mivel csak pozciótartó összehasonlításokra szorítkoztunk, ez a megkötés eleve biztosított.

Karakterlánc keresésekor is a fenti szempontokkal egészítjük ki a többszintű rendezést nem támogató kompakt fánál látott algoritmus-sémát. A fő eltérés, hogy a kereséskor nem az addigi útvonalakkal, hanem a keresett eredeti karakterlánccal való kompatibilitás számít, nyilván már az első szinttől kezdve.

3.3.8. Full-text indexelt értékkiosztás

Ez az értékkiosztási mód folyószövegeket állít elő, melyekhez full-text indexet tesz elérhetővé. Itt most csak a full-text index legegyszerűbb fajtájával foglalkozunk, melynek bemenete egy néhány szóból álló halmaz, eredménye pedig az ezen szavak mindegyikét tartalmazó mezők sorindexeinek halmaza. Rendezéssel ez esetben nem foglalkozunk, a full-text indexek ezt általában nem is biztosítják. Látni fogjuk, hogy a bemutatott megoldás könnyen továbbfejleszthető.

Első körben induljunk ki abból, hogy adott egy szólista, a rendre mellékelt f_i előfordulási gyakoriságokkal. Az f_i előfordulási gyakoriság szimulálható is a Zipf-törvényre alapozva. A Zipf-törvény egy számos különböző területen fennálló statisztikai jelenség, miszerint a diszkrét gyakoriságok általában hatványtörvény szerint rendeződnek (Zipf-eloszlás). Vagyis egy (némileg paraméterezhető) törvényszerűséget ad meg arra vonatkozóan, hogy a leggyakoribb, második leggyakoribb stb. elem milyen gyakorisággal fordul elő a teljes mintában. A törvényszerűség meglehetősen egyetemesen fennáll az organikusan alakuló rendszerekben, ezt követi például a jövedelmek, városok, állatok stb. méret szerinti gyakorisága. Esetünkben azért érdekes, mert a legtöbb természetes nyelvben a szavak gyakorisága is így oszlik el[43].

A szavakat egyszerűen betűkarakterek sorozataiként fogjuk föl, előzetesen függetlenítve a központozás, szövegfelépítés problematikájától.

Lesz emellett egy a szövegekre vonatkozó várható szószám is, jelöljük ezt c-vel. A gyakoriság és a hossz együtt meghatározza az adott szó p_i szereplési valószínűségét valamely szövegben a következő egyszerű képlet szerint:

$$p_i = 1 - (1 - f_i)^c$$

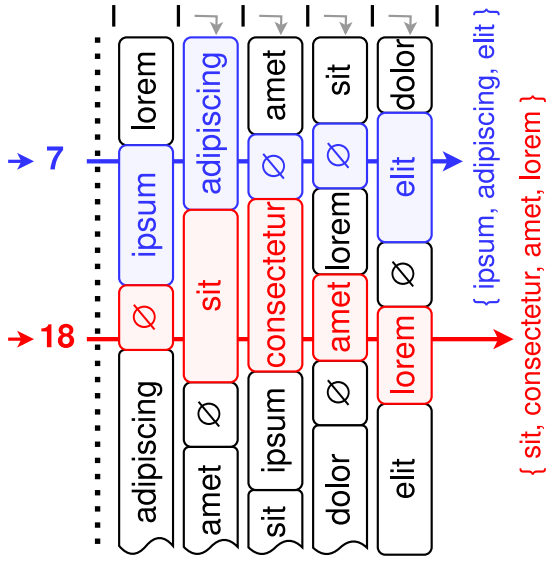
Legyen s az egy szövegben várhatóan előforduló szavak halmazának számossága ($s \leq c$). Az n táblahossz számú mezőre akarjuk képezni a szövegeket, amit egy $n \times s$ méretű virtuális mátrix segítségével fogunk elvégezni. A szavakat valamilyen sorrendben véve (a sorrend megválasztásának problémáit most mellőzöm), az ns hosszú listát feltöltjük úgy, hogy minden szó egy összefüggő sávban szerepel, a sávok sorrendje az említett szósorrend, a sávok egymáshoz képesti relatív hossza pedig a p_i szereplési valószínűségen alapul. Ezt a listát képezzük bele oszlopfolytonosan a mátrixba.

Egy adott sorindexhez tartozó mezőben szereplő szavakat úgy kapjuk, hogy egyszerűen vesszük a mátrix megfelelő sorát. Szavakra keresni pedig úgy tudunk, hogy a keresett szavakhoz tartozó (esetleg oszlopváltással megszakított) sávok metszetsávját vesszük, ha nincs metszet, nincs találat. A mátrixsorokat az életszerűség kedvéért érdemes permutálni a fentebb már ismertetett módszerrel.

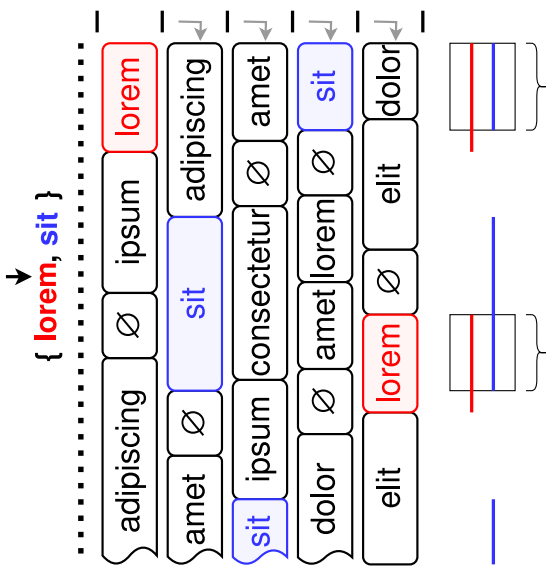
A mátrixos módszer egyik hátránya, hogy az egymás mellé került szavak eltartanak egymástól ugyanabban az oszlopban, így jellemzően nem tudnak majd bekerülni ugyanabba a szövegbe, illetve általában is viszonylag korlátolt, hogy melyek a lehetséges konstellációk. Ez viszonylag jól orvosolható azzal, ha egy-egy szó sávja több független sávba törik szét (jól ügyelve arra, hogy ezek a részsávok távoli sorokra essenek, de legalábbis ne legyen metszetük modulo n).

Egy másik probléma (hacsaknem épp ez a cél), hogy a szövegek túl uniformak, ugyanannyiféle szóból állnak. Ezen könnyen segíthetünk, ha a szósávok közé sok apró üres sávot helyezünk el.

A következő ábrák szemléltetik az eddig leírt szóösszeállítási, illetve keresési módszert:



3.9. ábra. Full-text szóösszeállítás



3.10. ábra. Full-text keresés

Egy harmadik egyszerű javítás adódik, ha előre ismert, mely néhány (ideálisan kb. 6-8) szó lesz gyakran keresve, ezeket teljesen külön kezelhetjük. Vesszük az összes lehetséges előfordulási kombinációt azok pontos gyakoriságával, az így kapott értékkészletet pedig lineárisan rányújtjuk a táblahosszra. Célszerű ezt a többi szótól teljesen függetlenül permutálni. Egy mező szóhalmazaként a kétféle kiosztás unióját kell venni, kereséskor pedig a kettő által adott találatok metszetét.

Megjegyzem, ha vannak kiemelt szavak, akkor a többi szó halmaza részben vagy egészben akár mesterségesen előállított szavakból is állhat. Ekkor egy minta alapján generálunk szavakat (on-the-fly, ld. a reguláris kifejezések kapcsán írottakat), majd például a Zipf-törvény alapján hozzájuk rendeljük a gyakoriságokat. Ha így járunk el, megspóroljuk a konkrét szavak tárolásának tárhelyigényét.

Egy-egy mezőértékhez még csak a szavak halmazát állítottuk elő. Amikor lekérjük a mező konkrét értékét, akkor egy tényleges szöveget szeretnénk kapni, mely pontosan ennek a halmaznak a szavait tartalmazza (sorrendtől és multiplicitástól függetlenül), és lehetőleg egy viszonylag valóságosnak tűnő szöveg, még ha nem is értelmes. Vegyük észre, hogy ebben teljes szabadságunk van, hiszen az érték és index konzisztenciája már biztosított.

Az aktuális `TreeRandom`, a sorindex és a kapott szóhalmaz függvényében a szöveg generálására tetszőleges módszert használhatunk, itt tehát egy széles skálázási lehetőség adódik. A fapados megoldás a halmaz szavainak egyszerű felsorolása szóközzel választva (abban a sorrendben, ahogy a kiemelt szavak fel voltak sorolva, illetve ahogyan a mátrixból nyertük őket). Egy nagyon szofisztikált módszer lehet, ha nyelvi modellt használunk a szöveg generálására.

Egy olcsó, ugyanakkor viszonylag jó eredményt adó módszer a következő:

1. Határozzuk meg a c_i konkrét szószámot.
2. Vegyük a szóhalmazt valamilyen konkrét sorrendben.
3. Helyezzünk ezek közé további szavakat, míg a c_i szószámot el nem érjük.
4. Határozzuk meg a mondat-, illetve részmondathatárok pozícióit.
5. Fűzzük össze a szöveget szóközökkel, adjuk hozzá a központozást.

Mindegyik lépés finomítható különféle heurisztikákkal, ezekkel most nem foglalkozom.

A fent tárgyalt full-text index természetes módon bővíthető. Könnyen integrálható például a kizárandó szavak kezelése. A szóhalmaz kiválasztásának és a szöveg generálásának összehangolásával a kifejezésekre való keresés bizonyos formái is megvalósíthatóvá válnak.

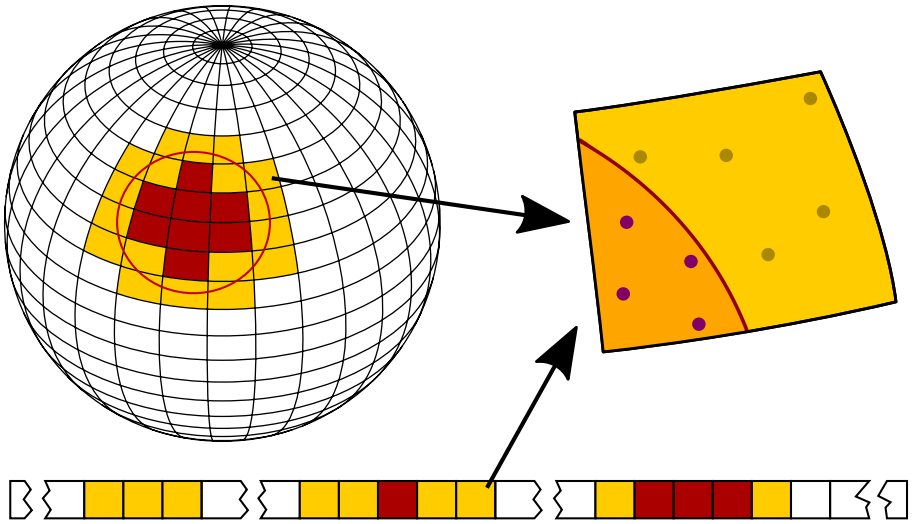
3.3.9. Földrajzi koordináták értékkiosztása

A legegyszerűbben közelítem meg a problémát, ami nagyobb dimenziószámra és lényegileg tetszőleges geometriára is átvihető. Csak a megoldás vázát ismertetem.

Az értékek sűrűségének, elhelyezkedésének valamint opcionálisan a várható lekérdezéseknek a figyelembevételével a felületet térszeletekre osztjuk. Ennek a felosztásnak nem feltétlenül kell egyenletesnek vagy egyáltalán szabályosnak lennie, de az egyszerűség kedvéért most maradjunk a szokványos hálónál: adott w földrajzi hosszúságonként, illetve adott h földrajzi szélességenként vágjuk föl a gömbfelszínt. A kapott háló (a sarkoknál háromszöggé fajuló) kis kvázi-négyszögeket határoz meg. Ezeket soronként véve egy globális sorrendezésüket kapjuk.

A szélesség-hosszúság párokat egy $2d$ bites ábrázolással kódoljuk. Ennek lehetséges értékkészletére nyújtjuk rá a sorindexeket a zajosan monoton értékkiosztásoknál ismertetett módszer valamely variánsával, de itt a nagyléptékű egyenletességet más szempontokra cserélhetjük. Az értékkészlet összefüggő sávjai a kvázi-négyszögeknek feleltethetők meg, tehát a kvázi-négyszögekbe tartozó értékekre ugyanúgy tudunk keresni, mint fentebb az időbélyegekre.

Kereséskor a kvázi-négyszögek két halmazát kell elkülöníteni: amelyek teljesen az alakzaton belülre esnek, illetve amelyeket az alakzat határvonala átmetsz. Az előbbiek értékei feltétel nélkül találatok lesznek. Utóbbiaknál a kvázi-négyszögön belülre kell hatolnunk, esetleg egyesével kell ellenőriznünk az értékek alakzaton belüliségét. Mivel ezek a határvonalhoz köthetők, számuk egyszerű alakzatok esetén a kerülettel egyenesen arányos, nem kell tehát területarányos mennyiségű kvázi-négyszöget megbontani, sőt, az egy sorban egymás után lévők összevontan is kezelhetők.



3.11. ábra. Geolokációk keresése alakzat alapján, rácsozott gömbfelületen.
Csak a határvonal által metszett mezőkbe eső pontokkal kell részletesebben foglalkozni.

3.4. Egyéb értékkiosztási módok

3.4.1. Nem-indexelt egyoszlopos értékkiosztások

A nem-indexelt oszlopok szimulálásakor a keresési-rendezési szempontokat egyáltalán nem kell figyelembe venni. Vagyis az elvárások a következőkre egyszerűsödnek:

1. hatékony elérés (random access)
2. pozíciótól függő érték (`TreeRandom` seed)

Így elég, ha a `TreeRandom` által a sorindexhez generált seed alapján előállítunk egy hagyományos `Random` példányt, ezt használva a tartalom előállításához már tetszőleges módszert alkalmazhatunk (az eredmény a seed által determinált lesz). Erre az előállításra kizárólag a konfigurációban megadott beállítások jelentenek megszorítást.

A következő listában a teljesség igénye nélkül felsoroltam az ilyen adattartalmak néhány jellegzetes, egyúttal könnyen szimulálható típusát:

- **Egyszerű szöveg:** a legegyszerűbb megoldásban egy szótár szavait véletlenszerűen helyezzük egymás után, közben feljavítva a szövegképet nagybetűs szavakkal és központozással (az eredmény tovább javítható kifejezésminták használatával, amikor időnként egy többszavas részletre előre adott sablont használunk; de ha a minőség az elsődleges, bevethetünk akár Markov-láncokra épülő egyszerűbb nyelvi modelleket[44])

- **Strukturált szöveg:** először előállítunk egy általános struktúrát (címek, bekezdések stb.), majd ennek elemeit feltöltjük szöveggel (lásd az előző pontot), végül a struktúrát a kívánt jelölőnyelven szolgáltatjuk (HTML, MarkDown, plain text stb.)
- **Strukturált adat:** generált vagy konfigurációban megadott adatséma alapján rekurzívan generáljuk le az adatstruktúrát, ekkor a sémanyelv (pl. JSON Schema) és a kimenet formátuma (JSON, YAML, TOML stb.) függetlenül konfigurálható
- **Egyszerűbb képek:** véletlen színű háttérre véletlen paraméterekkel rárajzolunk néhány egyszerű alakzatot, majd a kívánt vektoros vagy raszteres formátumban szolgáltatjuk
- **Jelszó-hash:** a tényleges a hash-algoritmust hívjuk meg valamely rendelkezésre álló adatra, pusztán formai elvárás esetén akár magára a `TreeRandom` seedre, demózáshoz pedig hasznos lehet a felhasználónevet használni jelszóként
- **Általános BLOB/CLOB:** lekéréskor a konfigurációban megadott mérethatárok közötti bájtsort/karaktersort kell visszaadni, ami könnyen ellátható random bájtok/karakterek generálásával (legegyszerűbb közvetlenül a `TreeRandom` által szolgáltatott biteket használni)

További indexeletlen eset, amikor az oszlop teljes tartalma explicite felsorolva van megadva, például egy listafájlból. Nagy táblahossznál érdemes lehet ezt indexeltté tenni egy tényleges adatstruktúrával (például B-Tree), mivel ilyenkor valószínűleg úgysem a memóriahasználat minimalizálása a cél.

3.4.2. Értékkiosztás oszlopok közötti összefüggéssel

Az oszlopok közötti összefüggésnek számtalan módja lehetséges. A legáltalánosabb eset képletszerű összefüggések felsorolása lenne, ami indexeletlen esetben meg is megvalósítható: a `TreeRandom`-ból nyert seeddel véletlenszerűsítve futtathatunk valamilyen constraint solvert, mely a követelményeket teljesítő értékeket determinisztikusan előállítja. Ha előfordulhat, hogy egyes seed értékek mellett a követelményrendszer nem kielégíthető, akkor meg kell engednünk a *NULL* értékeket az érintett oszlopokban, és a kielégíthetetlen esetekben mindegyikre *NULL*-t kell visszaadni (esetleg valamilyen egyéb alapértelmezett értéket).

Szintén elég általános eset, amikor mesteroszlopok alapján akarunk indexeletlen oszlopokat egymástól függetlenül legenerálni. Ekkor teljes szabadságunk van, értelemszerűen a mesteroszlopok értékei és a `TreeRandom`-ból nyert seed lesz a bemenet.

Életszerű azonban, hogy indexelt mesteroszlopokon túl a függő oszlopokra is szeretnénk indexelést. Ha egy nagy közös index elég, akkor elégséges a függő értékeket becsoportosítani a mesterértékek alá.

Bonyolult, de még mindig teljesen életszerű eset, amikor a függő értékek külön önálló indexeket is kapnak (ilyen eset például, ha az életkor adatot a végzettséghez igazítva generáljuk). Ha a mesteroszlop(ok) értékkészlete kicsi, akkor minden ilyen értéken belül elvégezhetjük a keresést, illetve rendezést a függő értékekre, és ezek összefésült eredményét adjuk vissza.

3.4.3. Táblák közötti kapcsolatok

Amíg megmaradunk a read-only szempontoknál, a táblák közötti idegenkulcs jellegű kapcsolatokra vonatkozó egyetlen kemény követelmény, hogy a hivatkozó tábla érintett oszlopa (vagy oszlop-*n*-ese) csak a hivatkozott tábla érintett oszlopához (vagy oszlop-*n*-eséhez) tartozó értékkészlet elemeit vehesse föl; vagyis (*NULL* értékektől eltekintve), a hivatkozó oszlop értékkiosztásakor az alaplista a hivatkozott oszlop tartalma lesz. Másként fogalmazva, az értéklistát indextartó módon kell átképezni a másik táblára. Ez a fentiek tükrében már egyszerűen biztosítható.

Egyes esetekben (bizonyos típusú 1:1 kapcsolatok) valójában egy tábla kerül szétbontásra több táblára. Ekkor a több táblára bontottság csak külsődleges, és az előző részben oszlopok közötti összefüggésekről írottak alkalmazhatók, még akkor is, ha a két tábla számossága nem egyezik teljesen (a hiányzó sorok a *NULL* értékek kezelésének fentebb ismertetett általános módszerével kerülhetnek kiválasztásra).

A legbonyolultabb esetben 1:n kapcsolat mellett különféle statisztikai vagy egyéb követelményeket biztosító indexelt oszlopokat szeretnénk előállítani. Ennek tárgyalása messzire vezet, de egy rövid megjegyzéssel érzékeltetném, hogy a dolog nem lehetetlen: egy lehetséges kiinduló megközelítés, hogy a két táblát egyként kezeljük, az egyesített táblában a kisebb tábla adatait pedig redundánsan szerepeltetjük (ez egy oszlopközi összefüggés a fenti értelemben); így az összefüggéseket már táblán belül tudjuk értelmezni.

4. fejezet

Gyakorlati eredmények

4.1. Néhány bővítési példaszcenárió

Eddig általános igényekkel foglalkoztunk, melyeket a megoldásnak out-of-the-box kell támogatnia. Alább bemutatok három speciálisabb szcenáriót, demonstrálva az architektúra bővíthetőségét és rugalmasságát.

4.1.1. Valós címadatok használata

Előfordulhat, hogy bizonyos adatok óriás értékkészletét külső adattárakból szeretnénk betölteni, amivel ugyan elvesztjük például a teljes virtualitás kis memóriaigényét, de még mindig igényt támaszthatunk az egyéb előnyökre. Most egy ilyen szcenáriót vizsgálok meg: valamely tábla rekordjaiban olyan magyarországi közigazgatási címeket szeretnénk viszontlátni, melyek valósak. Amennyire lehetséges, indexeket is szeretnénk biztosítani.

A cím elemei különálló mezőkre bonthatók. Első körben kiválasztjuk azokat, melyek egy egyértelmű hierarchiát definiálnak:

településnév $\rightarrow$ kerület $\rightarrow$ közterület neve $\rightarrow$ közterület jellege $\rightarrow$ házszám $\rightarrow$ egyéb

Az „egyéb” részt nem szofisztikáljuk tovább, egy szabad szöveges mezőnek tekintjük.

Néhány további mező nem tartozik a fő hierarchiába:

település típusa • vármegye • irányítószám

Utóbbi mezőkre külön-külön indexeket fogunk biztosítani. A fő hierarchiára pedig egy közös többoszlopos (leftmost prefix) indexet, pontosabban kettőt: lesz egy rövidített változat vidéki

címekhez, melyben a kerületet nem kell megadni, és implicite üres értékkel töltődik, amikor továbbhív a másik indexre.

A címadatbázist egy nagy fastruktúrába rendezzük, minden szintnek a fenti hierarchia egy-egy eleme felel meg. Az alsóbb szinteken spórolhatunk a tárolással, például egymásutáni megbontatlan házszámok tartományként tárolhatók. Egy prefix indexhívásnál a megfelelő szintig megyünk le a keresésben, a találatok pedig az az alatti levelek lesznek. A tényleges rekordokra való leképezést a kétlépéses értékkiosztásnál megismerthez hasonló módszerrel végezzük el.

A fő hierarchián kívüli mezők értékei és a fa közötti indexelést külön adatstruktúrába generáljuk. Változó lehet, hogy a tábla mely szintjén történik meg az irányítószámra való leképezés, a legtöbb helyen egy egész várost tudunk hozzárendelni.

4.1.2. Valószerű geográfiai értékek előállítása

A probléma: földrajzi koordinátákhoz akarunk valamilyen számszerű adatot kapcsolni úgy, hogy mindkettőhöz külön index tartozik, és a koordinátával ellátott adatok valamilyen szempontból valószerű elrendeződést mutatnak. Olyan megoldást kell találni, ahol tetszőleges számú ilyen kapcsolt adat akár függetlenül generálható (különben elég lenne a szélesség-hosszúság-adat háromdimenziós terében megfelelően alkalmazni a földrajzi koordináták fentebb ismertetett értékkiosztását).

Az alapötlet rendkívül egyszerű, és kellően rugalmas. Szükségünk lesz egy F függvényre, mely minden földrajzi koordinátához rendel egy kapcsolt értéket. A kapcsolt adat értékkészletét kis sávokra bontjuk föl (nevezzük ezeket kontextussávoknak), és minden sávhoz le kell tudnunk kérni azon alakzatok (véges) listáját, melyekben az F függvény értéke a sávba esik. Bármilyen függvény megfelel, ha ezeket tudjuk biztosítani hozzá. Használhatunk például teljesen virtuális háromszögelt felületet, ahol egy-egy háromszög mindig egy konkrét értéksávba esik, nem nyúlik át. Vagy akár betölthetünk tényleges domborzati adatokat.

Ha ezt sikerült megoldani, az értékkiosztás működése már triviális. Adott rekordhoz a földrajzi koordinátát a korábban ismertetett módszerrel generáljuk, a kapcsolt adatot ez alapján számítjuk az F függvény szerint. A kapcsolt adat valamely egyéni értéksávjára való kereséskor lekérjük a sávon teljesen belülre eső, illetve a kilógó kontextussávokat, lekérjük az ezekhez tartozó alakzatokat, és ezen alakzatokra keressük a földrajzi koordinátákat. A keresett sávon teljesen belülre eső kontextussávokhoz tartozó találatok feltétel nélkül megtarthatók. A kilógó kontextussávokban le kell kérni a konkrét kapcsolt értéket, és eldobni a találatot, ha az mégsem esik a keresett sávba.

4.1.3. Dev mode (live mode)

Főként frontend felületek (képernyők, komponensek) fejlesztése közben megszokott az úgynevezett *dev mode* (vagy *live mode*). Ilyenkor a forráskódban eszközölt módosítások életbe lépéséhez sem a frontend környezet újraindítására, sem a böngészőablak frissítésére nincs szükség. A dev mode-ban futó szerver figyeli a releváns változásokat a fájlrendszeren, így egy-egy fájl mentésekor azonnal értesíteni tudja a futó frontendet. A frontend pedig úgy van kialakítva, hogy ennek hatására gazdaságosan frissítse az éppen megjelenített nézetet.

A frontend újratöltése jellemzően nem jár komolyabb háttérszámításokkal, így instant újratöltése könnyen megvalósítható, sőt, elvárás is, hiszen a felhasználói élményt teljesen lerombolja, ha egy frontend felület maga lefagy. Ez hagyományosan az adatbázisok adattal való költséges feltöltésének az ellentéte. Az adatbázisok esetében a dev mode megvalósítása fel sem merül, hiszen a módosítás életbelépése esetenként igen költséges lekérdezések alkalmazását jelentené, rosszabb esetben egy generáló szkript teljes újrafuttatását. A HoloDB esetében azonban ezek a költségek nincsenek vagy minimálisak, így lehetővé válik, hogy a lokális tesztadatbázist dev mode-ban futtassuk.

4.1.4. Zero mode

A dev mode elképzelését tovább is vihetjük egy egészen radikális változatig, melyet *zero mode*-nak nevezek. Az elnevezés magyarázata, hogy nem pusztán az adatok tárolására nincs szükség, de konfigurációs fájlra sem. Ekkor ugyanis mindent a beérkező kérések alapján fogunk generálni. Kihasználjuk, hogy egy-egy lekérdezés implicite többé-kevésbé megmondja, milyen adatszerkezetre számít. Tegyük föl például, hogy a frissen indított zero mode szerverre elsőként a következő két SQL-lekérdezés érkezik:

```
SELECT email FROM customers WHERE vip=1 LIMIT 10;
SELECT COUNT(*) FROM customers WHERE phone IS NOT NULL;
```

Amikor az első lekérdezést értelmezzük, már tudjuk, hogy elvileg lennie kell egy `customers` nevű táblának, melyben van `email` és egy `vip` nevű mező, és az utóbbi típusa kompatibilis az `1` értékkel. Továbbá statisztikai tippet is nyerhetünk a `10`-es limitből, gondolhatjuk, hogy ennél több olyan rekord van, amelyek a `vip=1` feltételnek megfelelnek, legalábbis az biztosan érvényes helyzet, hogy ennél több van. A második lekérdezésben megtudjuk, hogy kell lennie egy `phone` nevű mezőnek is, amely `NULL` értéket is felvehet.

A konfigurációt a dev mode-nál megismert módon menet közben kell újratöltenünk, ha olyan lekérdezés érkezik, mely az előzőekhez képest többletinformációt tartalmaz. Ez azt is jelenti, hogy a

kérések ecélból történő előfeldolgozása egy külön szűk keresztmetszetet képez, mivel a konfiguráció inkrementális módosítása közben blokkolni kell a további lekéréseket. Egy alapértelmezett root seedet használhatunk, amit parancssori opcióval esetleg felül lehet írni.

Kézenfekvőenk tűnik a megoldás, hogy a konfigurációt mindig a megfelelő módon egyszerűen bővítsük, amikor szükséges. A helyzet azonban valamivel komplikáltabb. Előfordulhat például, hogy az addig (akár megfelelő részeltes információk hiányában) rögzített oszlopjellemzőknek ellentmondó implicit oszlopinformációt kapunk. Vagy eleve megpróbáljuk elkerülni ezeket az eseteket, vagy megadunk egy kezelési módot. Továbbá, vagy ragaszkodunk a koherenciához, vagy nem.

Az elkerülésre két fő stratégia adódik. A visszaadott eredménytáblák mindkettő esetében függetlenek lesznek a kérések beérkezési sorrendjétől. Az elsőt nevezhetjük *simple mode*-nak. Ekkor pusztán a tábla és az oszlopok neveiből indulunk ki (valamint egy alapértelmezett, de akár kapcsolóval felülírható root seed értékből). A tábla mérete és egyéb jellemző a táblanév alapján kerülnek meghatározásra, Az oszlop típusa és egyéb jellemzői pedig az oszlop neve alapján (mellékes, hogy pontosan milyen módszerrel). Természetesen érdemes bevetni olyan heurisztikákat, melyek a beszédes oszlopneveket felismerik (az előbbi `email` például ilyen). A másodikat hívhatjuk *ad hoc mode*-nak. Ekkor a lekérések függetlenek egymástól. Minden lekérdezést igyekszünk a legjobban igazodó adatszerkezettel kiszolgálni, cserébe teljesen elengedjük a kérések közötti koherenciát.

Ha nem próbáljuk elkerülni az oszlop jellemzőit illető ellentmondások kezelését, szintén két stratégia adódik. Az *incremental mode* esetén már felvett oszlopon nem változtatunk, az új oszlopokat természetesen az újonnan érkező információk alapján létrehozzuk. A *correction mode* esetén meglévő oszlopokat is módosíthatunk, amikor egy új lekérdezés nagyon eltérő oszlopjellemzőket sugall. A táblaméretet és néhány egyéb globális jellemzőt ekkor sem módosítunk.

Mind a dev mode, mind a zero mode működtethető írási réteggel együtt, akár úgy is, hogy a nem változó oszlopok módosításait megtartjuk.

4.2. Empirikus eredmények

4.2.1. Mérési módszertan

Mivel a HoloDB preparálási folyamat nélkül rendelkezésre áll, lényegileg azonnal elindul és minimális memóriát fogyaszt, ezekben a paraméterekben nem volt értelme összevetni a radikálisan eltérő opciókkal. Így alapvetően a lekérdezési sebességre koncentráltam, azt szűk keresztmetszetében és end-to-end jellegű integrált tesztekkel is megvizsgáltam. A performanciával kapcsolatban a

várakozás mindössze annyi volt, hogy legyen összemérhető a valódi adatbázisokéval. A Docker-képek összehasonlítását leszámítva minden esetben a HoloDB beágyazott verzióját használtam. Az összehasonlítást MySQL-lel (mint az egyik legelterjedtebb szerveradatbázissal) és H2-vel (mint a Java környezetben legelterjedtebb beágyazott adatbázissal) valósítottam meg. A lekérdezések közvetlen mérésénél az SQL parancsnak a JDBC API-n keresztüli beküldésétől az eredménytábla végigiterálásnak végéig eltelt időt mértem. Különböző bonyolultságú és válaszméretű lekéréseket vizsgáltam.

Az integrált tesztelést egy Micronaut keretrendszerben implementált alkalmazás REST API-n keresztüli hívásával futtattam. A Micronaut keretrendszer egy kis erőforrásigényű Java keretrendszer, amely minimális erőforrás-lábnyománál fogva a legalkalmasabbnak tűnt a mérésre.

A Spring és Quarkus rendszerekkel szemben a Micronaut már fordítási időben tárolja és optimalizálja az alkalmazás menedzselt objektumai közötti függőségeket, így általában gyorsabban is indul el, mint az előző kettő, így performancia-tesztekre is alkalmasabb.

Az alkalmazás futtatását egy POSIX shell szkript vezérli. Egy további Bash szkript az előbbi fájlt hívja többszörösen, közben átírja a konfigurációs fájlt, hogy különböző teszteket hajtson végre. A bash szkript a futtatási ciklusok között legalább 5 másodperc szünetet vár.

A lekérdezésszintű tesztekhez a H2-t a `QUERY_CACHE_SIZE=0` opcióval használtam, míg az integrált tesztnél ezt elhagytam az authentikusság kedvéért. (A teljes benchmarkért ld.: Függelék C).

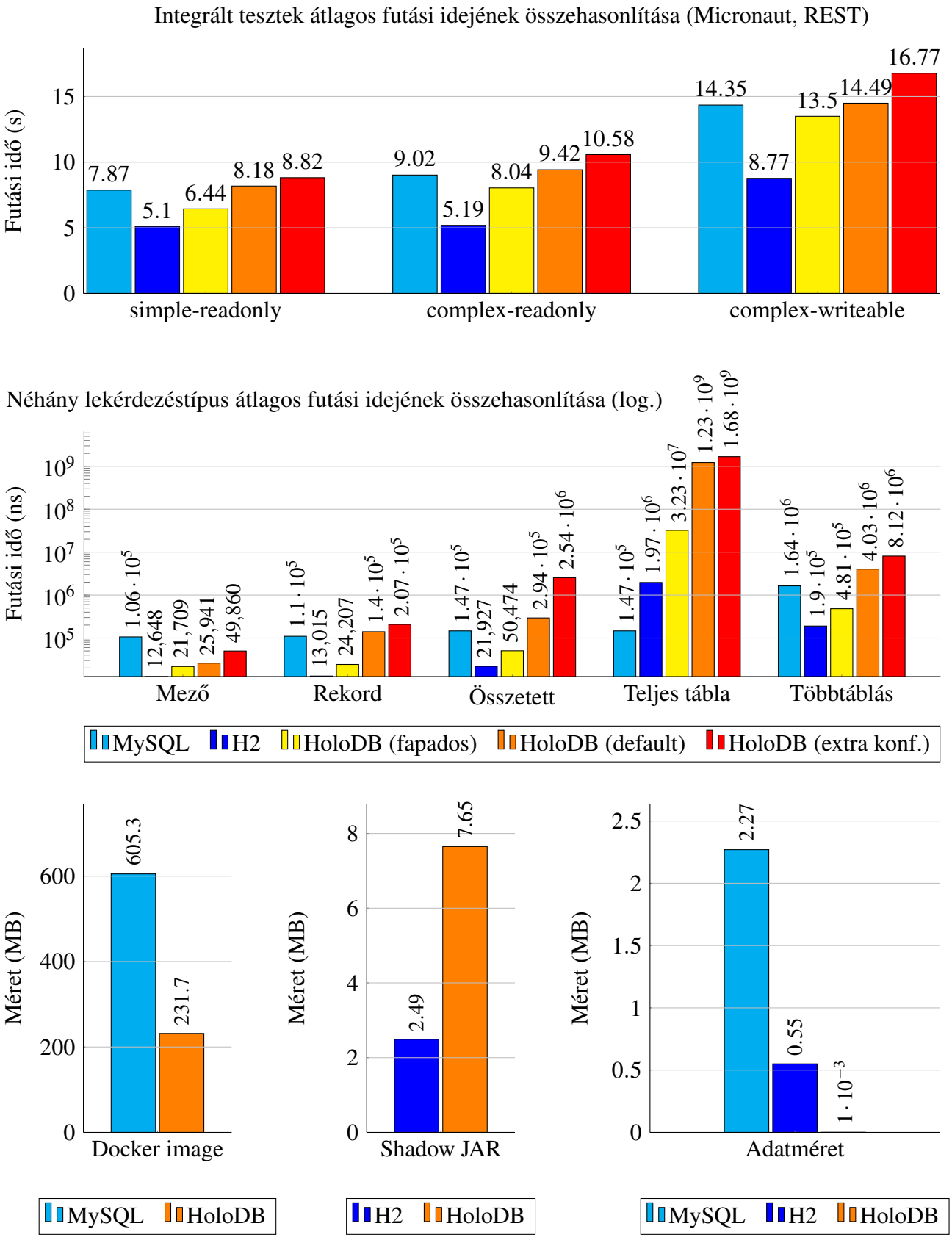
4.2.2. Az lekérdezési és integrált tesztek eredménye

A 4.1 ábra felső részén az integrált teszt három verziójának eredményei láthatók. A `simple-readonly` teszt főleg egyszerűbb lekérdezéseket tartalmazott, a `complex-readonly` teszt ezt néhány összetettebb lekérdezéssel egészítette ki. A `complex-writeable` tesztben pedig adatírások, majd arra épülő lekérdezések is történtek.

A lekérdezések esetében kevés adatnál a teljesítmény meglepően jó, és drasztikusan romlik nagy eredménytábláknál. Vagyis a korábban tárgyalt indexelési technikák jól teljesítenek, kulcsfontosságúak a megfelelő teljesítményhez. Ugyanakkor az adatelérés kevésbé optimális; de feltehetően mind az adatok számítása, mind az eredménytábla továbbítása jelentősen javítható még.

Az adatok tárhelyigényében a HoloDB természetesen drasztikus csökkenést mutat, hiszen eleve így lett tervezve. A szerveres kontextust nézve, a HoloDB Docker-képének mérete jóval kisebb, mint

a MySQL-é; de a base image változtatásával, az operációs rendszer és a JRE testreszabásával ez valószínűleg tovább javítható.



4.1. ábra. A különböző megközelítések teljesítményének és tárigényének összehasonlítása

5. fejezet

Összegzés

5.1. Az eredmények értékelése

A dolgozatban az adatbázismockolás egy újszerű megközelítését mutattam be. Láthattuk, hogy sok kicsi, többnyire egyszerű ötlet kombinálásával lehetséges egy funkcionálisan teljes értékű relációs adatbázist működtetni, anélkül, hogy azt ténylegesen tárolni kellene.

A prototípus a teszteredmények alapján valóban drasztikusan csökkenti az erőforráshasználatot, miközben tisztán deklaratív, snapshot jellegű entitássá teszi a nem-produkciós adatbázist. Megmutattam, hogy memóriahasználatban a HoloDB lényegesen kedvezőbb, mint a vizsgált alternatívák; valamint hogy runtime teljesítménye a legtöbb esetben összemérhető a valódi adatbázisokéval; ez pedig a generálási, anonimizálási és egyéb inicializáló lépések kikerülése miatt összességében sokkal gyorsabb tesztfuttatást tesz lehetővé.

Azzal, hogy nincs jelentős generálási és indítási költség, olyan új lehetőségek nyílnak meg, mint a teljes adatbázistartalommal történő füstteszt, vagy a teljesen izolált automatikus integrációs tesztek.

5.2. A célkitűzések teljesülése

1. **Támogatja a relációs adatmodellt.** Valóban: a megoldás megvalósítja a relációs storage API-t és támogatja az SQL lekérdezések futtatását.
2. **Determinisztikus, koherens.** Valóban: a hierarchikus véletlengenerátor biztosítja a determinisztikusságot, az értékkiosztások pedig a konzisztens működést adatelérés és index, illetve több lekérdezés között.

3. **Az adatok könnyen variálhatóak.** Valóban: minden adat függ a konfigurációban elhelyezett root seedtől, melynek módosításával a teljes adatbázis megváltozik.
4. **Óriás adatmennyiséget is képes szimulálni.** Valóban: mivel az adatok nincsenek tárolva, a mennyiség önmagában nem jelent problémát; az aritmetika a nagy számokkal történő számoláshoz lett igazítva.
5. **Gyors, indexelt keresést nyújt, a lekérések időigénye elfogadható.** Valóban: az értékkiosztásokban alkalmazott megoldások lehetővé teszik a virtuális indexeket; a lekérdezések sebessége az empirikus tesztek alapján is elfogadható.
6. **Azonnal elindul, nincs számottevő preparálási folyamat.** Valóban: alapértelmezetten pusztán a konfiguráció betöltése történik meg az indításkor.
7. **Kevés memóriát foglal, az adatokat on-the-fly számítja.** Valóban: maguk az adatok egyáltalán nincsenek tárolva, lekérdezéskor kerülnek számításra, néhány esetben kisebb adatstruktúrák generálódnak (pl. regex szófák), melyek mérete a táblamérettől független.
8. **Könnyen és rugalmasan konfigurálható.** Valóban: a deklaratív YAML konfigurációs fájl könnyen kezelhető és verziókezelhető, minden fontosabb paraméter beállítására alkalmas.
9. **Skálázható, finomhangolható.** Valóban: lényegileg minden komponens skálázható, illetve akár kicserélhető, különösen az értékkészletek, monoton méretigazítások és permutációk vannak hatással a teljesítményre és az eredmény minőségére.
10. **Egyedi működéssel könnyen bővíthető.** Valóban: lehetőség van az értékkiosztások egyedi Java osztályokkal való implementálására, ami nagyrészt a konfigurációban is megadható, emellett egy programozható API is rendelkezésre áll.
11. **Opcionálisan írható.** Valóban: a módosító SQL-ek is támogatottak, az írhatósági és az egyszerű tranzakciókezelési réteg lehetővé teszi az írási műveletek hatásainak megjegyzését.

5.3. Tervek a közeljövőre

A HoloDB egy alapvető, a szoftverfejlesztők többségét érintő probléma megoldását kínálja. Ha a szoftver, leküzdve gyermekbetegségeit, eljut egy igazán stabil verzióig, valamint sikerül a koncepció ideáját a fejlesztők szélesebb nyilvánosságával megismertetni, úgy természetes következményként nagy népszerűségre lehet számítani.

A további munkát több frontvonalra tudom bontani:

- az alapvető működés optimalizálása

- további kiegészítő funkciók, eszközök fejlesztése
- dokumentáció és kommunikáció

Az értékkiosztásokat és lekérdezéseket sok apró alkatrész teljesítményének finomhangolásával, elsősorban algoritmuselméleti kutatással lehet feljavítani. Ugyanilyen fontos viszont a programkörnyezet hatékonyabbá tétele is, különösen a Docker-képé. Optimalizáláson nem csak a performancia tuningolását, de az architekturális célszerűség növelését és a felhasználói élmény javítását is értem. Ebbe a logikus, könnyen érthető, lényegre törő konfiguráció éppúgy beletartozik, mint a kiegészítő eszközök ergonómiája. Az új eszközök fejlesztése elsősorban a különféle meglévő technológiákkal történő integrációt jelenti (tervezőeszközök, adatbázis-böngészők, programozási környezetek stb.). Másodsorban pedig a használat megkönnyítését szeretném megcélozni (generálás/materializálás, további REPL-funkciók, telepítőcsomagok, webes API-k stb.).

Az API-dokumentáción túl egy-egy alapvető kalauz megtalálható az egyes projektek `README.md` fájljaiban, valamint különféle instant elindítható példaprojektek is segítik az ismerkedést. Ezek bővítésén túl tutorial stílusú leírásokra is szükség lesz. A bevett online felületeken túl (*GitHub*, *Stack Overflow*, *BetaPage* stb.) mind az akadémiai, mind a versenyszféra-közegben tervezem, hogy workshopokon és kérdőíves formában is gyűjtsem a visszajelzéseket.

A bemutatott megoldás egy újszerű szoftver, melynek esszenciája a bevett terminológiákkal csak tekervényesen fogalmazható meg. Így egy megfelelő, jellegzetes és egyszerű megnevezés sokat segíthet majd szélesebb körű népszerűsítésben. Felmerült jelöltként például a *holografikus adatbázis*, a *fantomadatbázis*, és a *deklaratív adatbázis* kifejezés. Mindegyik mellett szólnak pro és kontra érvek.

Irodalomjegyzék

[1] Jan Ploski és tsai. „Introducing Version Control to Database-Centric Applications in a Small Enterprise”. *Software, IEEE* 24 (2007. febr.), 38–44. old. DOI: 10.1109/MS.2007.17.

[2] Jasmin Fluri, Fabrizio Fornari és Ela Pustulka. „Measuring the Benefits of CI/CD Practices for Database Application Development”. 2023. márc. DOI: 10.1109/ICSSP59042.2023.00015.

[3] Jonathan Edwards, Tomas Petricek és Tijs van der Storm. *Live Local Schema Change: Challenge Problems*. 2023. arXiv: 2309.11406 [cs.DB]. URL: https://arxiv.org/abs/2309.11406.

[4] Andres Sacco. „Versioning or Migrating Changes”. *Beginning Spring Data: Data Access and Persistence for Spring Framework 6 and Boot 3*. Berkeley, CA: Apress, 2023, 163–185. old. ISBN: 978-1-4842-8764-4. DOI: 10.1007/978-1-4842-8764-4_6. URL: https://doi.org/10.1007/978-1-4842-8764-4_6.

[5] Redgate Software. *Flyway*. v11.0.0. verzió. Accessed: 2024-12-01. URL: https://github.com/flyway/flyway.

[6] Nathan Voxland. *Liquibase*. v4.30.1. verzió. Accessed: 2024-12-01. URL: https://github.com/liquibase/liquibase.

[7] Abhijit Kane Abhinav Asthana Ankit Sobti. *Postman/Newman*. v6.2.1. verzió. Accessed: 2024-12-01. URL: https://github.com/postmanlabs/newman.

[8] Tom Akehurst. *WireMock*. v3.10.0. verzió. Accessed: 2024-12-01. URL: https://github.com/wiremock/wiremock.

[9] Stoplight.io. *Stoplight/Prism*. v5.12.1. verzió. Accessed: 2024-12-01. URL: https://github.com/stoplightio/prism.

[10] SmartBear Software. *SoapUI*. v5.7.2. verzió. Accessed: 2024-12-01. URL: https://github.com/SmartBear/soapui.

[11] Gediminas Morkevicius. *Go SQLMock*. v1.5.2. verzió. Accessed: 2024-12-01. URL: https://github.com/DATA-DOG/go-sqlmock.

[12] Benjamin Curtis. *Ruby Faker*. v3.5.1. verzió. Accessed: 2024-12-01. URL: https://github.com/faker-ruby/faker.

[13] DiUS Computing. *Java Faker*. v1.0.2. verzió. Accessed: 2024-12-01. URL: https://github.com/DiUS/java-faker.

[14] *Elite*. URL: http://www.iancgbell.clara.net/elite/.

[15] Ian Bell és David Braben. *Elite, 1984*. URL: https://elite.bbcelite.com/cassette/.

[16] Jimmy Maher. *Starflight*. 2014. URL: https://www.filfre.net/2014/10/starflight/ (elérés dátuma 2014. 10. 28.).

[17] *Minecraft Seed (level generation)*. URL: https://minecraft.fandom.com/wiki/Seed_(level_generation).

[18] *StarMade*. URL: https://www.star-made.org/.

[19] Raffi Khatchadourian. *World Without End*. The New Yorker. 2015. URL: https://www.newyorker.com/magazine/2015/05/18/world-without-end-raffi-khatchadourian (elérés dátuma 2015. 05. 11.).

[20] Mohsen Baqery. *Starfield: Are Planets Procedurally Generated? Answered*. 2023. URL: https://gamerant.com/starfield-are-planets-procedurally-generated-answered-how-algorithmic-generation-works/ (elérés dátuma 2023. 08. 15.).

[21] RapidDweller. *DATAMIMIC*. v1.1.0. verzió. Accessed: 2024-12-17. URL: https://github.com/rapiddweller/datamimic.

[22] Tirthajyoti Sarkar. *pydbgen*. v1.0.0. verzió. Accessed: 2024-12-01. URL: https://github.com/tirthajyoti/pydbgen.

[23] Fabian Prasser Florian Kohlmayer. *ARX*. v3.9.1. verzió. Accessed: 2024-12-01. URL: https://github.com/arx-deidentifier/arx.

[24] Matteo Giomi és Mikhail D. *Anonymeter*. v1.0.0. verzió. Accessed: 2024-12-01. URL: https://github.com/statice/anonymeter.

[25] Tim I Johann és tsai. „Anonymize or synthesize? Privacy-preserving methods for heart failure score analytics". *European Heart Journal - Digital Health* (2024. nov.), ztae083. ISSN: 2634-3916. DOI: 10.1093/ehjdh/ztae083. URL: https://doi.org/10.1093/ehjdh/ztae083.

[26] *MOSTLY AI*. Accessed: 2024-12-01. URL: https://github.com/mostly-ai/mostlyai.

[27] *Tonic*. URL: https://www.tonic.ai/ (elérés dátuma 2024. 12. 01.).

[28] *GenRocket*. URL: https://www.genrocket.com/ (elérés dátuma 2024. 12. 01.).

[29] *Delphix*. URL: https://www.delphix.com/ (elérés dátuma 2024. 12. 01.).

[30] DB-Engines. *DB-Engines Ranking per database model category*. Archived version: https://web.archive.org/web/20241001104444/https://db-engines.com/en/ranking_categories. Accessed: 2024-10-01. 2024. URL: https://db-engines.com/en/ranking_categories.

[31] FasterXML. *Jackson Databind*. v2.18.2. verzió. Accessed: 2024-12-01. URL: https://github.com/FasterXML/jackson-databind/tags.

[32] Mohammed Erritali és tsai. „An approach of semantic similarity measure between documents based on big data". 6 (2016. jan.), 2454–2461. old. DOI: 10.11591/ijece.v6i5.10853.

[33] Denis Rosset. *Optimizing BigInt for small integers*. Online forum post. Scala Contributors. 2019. URL: https://contributors.scala-lang.org/t/optimizing-bigint-for-small-integers/3412.

[34] Apache Commons. *Apache Commons Math*. v4.0-beta1 verzió. Accessed: 2024-12-01. URL: https://github.com/apache/commons-math.

[35] Chuck Easttom. „Feistel Networks". *Modern Cryptography: Applied Mathematics for Encryption and Information Security*. Cham: Springer International Publishing, 2022,

139–168. old. ISBN: 978-3-031-12304-7. DOI: 10.1007/978-3-031-12304-7_6. URL: https://doi.org/10.1007/978-3-031-12304-7_6.

[36] Mihir Bellare és tsai. „Format-Preserving Encryption”. *IACR Cryptol. ePrint Arch.* 2009 (2009), 251. old. URL: https://api.semanticscholar.org/CorpusID:1081553.

[37] Colm O’Connor. *Xeger*. v0.3.5. verzió. Accessed: 2024-12-01. URL: https://github.com/crdoconnor/xeger.

[38] Youssef Mifrah. *Generex*. v1.0.2. verzió. Accessed: 2024-12-01. URL: https://github.com/mifmif/Generex.

[39] Andrew W. Appel és Guy J. Jacobson. „The world’s fastest Scrabble program”. *Commun. ACM* 31.5 (1988. máj.), 572–578. ISSN: 0001-0782. DOI: 10.1145/42411.42420. URL: https://doi.org/10.1145/42411.42420.

[40] Ken Whistler és Markus Scherer. *Unicode Collation Algorithm*. Unicode Technical Standard UTS #10. 16.0.0. verzió. Accessed: 2024-12-01. The Unicode Consortium. URL: https://www.unicode.org/reports/tr10/.

[41] *Swedish Collation Tailoring*. Unicode Technical Standard CLDR v43.0. 43.0. verzió. Accessed: 2024-12-01. The Unicode Consortium. URL: https://www.unicode.org/cldr/charts/43/collation/sv.html.

[42] ICU Project. *ICU - International Components for Unicode*. Accessed: 2024-12-01. 2024. URL: https://icu.unicode.org/.

[43] George Kingsley Zipf. „The Unity of Nature, Least-Action, and Natural Social Science”. *Sociometry* 5.1 (1942), 48–62. old. ISSN: 00380431. URL: http://www.jstor.org/stable/2784953.

[44] @amrrs. *Meaningful Random Headlines by Markov Chain*. Kaggle. 2021. URL: https://www.kaggle.com/code/nulldata/meaningful-random-headlines-by-markov-chain/notebook (elérés dátuma 2023. 12. 02.).

A. függelék

Algoritmusok

1. Algoritmus `getFeistelLastZero(index)` – Feistel-hálózat last-zero változat

```
 1: ▷ blockSize: blokkméret, N: kör-párok száma, index: a lekért pozíció
 2: leftSize, rightSize : int
 3: allBits, leftBits, rightBits, tmpBits, resultBits : bit[]
 4: leftSize ← (blockSize + 1) / 2
 5: rightSize ← blockSize - leftSize
 6: allBits ← INDEXNUMBERTOBITS(index)
 7: leftBits ← allBits[0 .. leftSize-1]
 8: rightBits ← allBits[leftSize .. blockSize-1]
 9: for all i from 0 until N do
10:     rightHash ← HASH(rightBits, i, 0)
11:     tmpBits ← leftBits.xor(rightHash, leftSize)
12:     leftHash ← HASH(tmpBits, i, 1)
13:     rightBits ← rightBits.xor(leftHash, rightSize)
14:     leftBits ← tmpBits
15: resultBits ← rightBits.concat(leftBits)
16: return BITSTOINDEXNUMBER(resultBits)
```

2. Algoritmus `createRawTrie(regexAstSeq,suffixTree)` – Nyers szófa összeállítása

```
 1: ▷ regexAstSeq: regex AST szekvencia, suffixTree: suffix-fa, alapértelmezetten <EOW>
 2: astItem : RegexAstNode
 3: resultTree, branch : Trie
 4: branches : Queue<Trie>
 5: resultTree ← suffixTree
 6: for all i from regexAstSeq.length-1 down to 0 do
 7:     astItem ← regexAstSeq.items[i]
 8:     if astItem is QualifiedAstNode then
 9:         astItem ← QUALIFIEDASTNODETOALTERNATION(astItem)
10:     if astItem is SequenceAstNode then
11:         resultTree ← createRawTrie(astItem, suffixTree)
12:     else if astItem is AlternationAstNode then
13:         branches ← NEWEMPTYQUEUE
14:         for all branchSeq in astItem.branches do
15:             branch ← createRawTrie(astItem, suffixTree)
16:             branches.enqueue(branch)
17:         resultTree ← CREATETREEFROMROOTVALUEANDCHILDRENQUEUE(NEWEMPTYNODEVALUE,branches)
18:     else
19:         resultTree ← CREATETREEFROMROOTVALUEANDCHILDRENQUEUE(astItem,QUEUEOF(resultTree))
20: return resultTree
```

3. Algoritmus `convertRawTrieToCompact(parentNodeValue,children)` –
– Nyers szófa kompakttá alakítása

```
1:  ▷ parentNodeValue: regex AST érték, children: feldolgozandó gyermekelemek
2:  resultQueue, subQueue : Queue<Trie>
3:  nextInputs, subResultChildren : Array<Trie>
4:  normalizedNextInputs : SortedMap<CharClassAstNode|EowAstNode, Array<Trie>>
5:  noAnchor : Array<AnchorAstNode>
6:  noAnchor ← newEmptyArray()
7:  nextInputs ← collectNextInputs_Helper(parentNodeValue,noAnchor,children)
8:  normalizedNextInputs ← createEmptySortedMap
9:  for all nextInput in nextInputs do
10:     normalizedNextInputs ← mergeIntoNormalizedMap_OuterHelper(normalizedNextInputs,nextInput)
11: resultQueue ← newEmptyQueue
12: for all astNode, associatedChildren in normalizedNextInputs do
13:     subQueue ← newEmptyQueue
14:     for all associatedChild in associatedChildren do
15:         subQueue.enqueueAll(associatedChild.children)
16:     subResultChildren ← convertRawTrieToCompact(astNode,arrayFromQueue(subQueue))
17:     resultQueue.enqueue(createTreeFromRootValueAndChildrenQueue(astNode,subResultChildren))
18: arrayFromQueue(resultQueue)
```

3/a. Algoritmus `collectNextInputs_Helper(parentNodeValue,anchors,children)`

```
1:  ▷ parentNodeValue: regex AST érték, anchors: horgonyok eddigi szekvenciája, children: feldolgozandó gyermekelemek
2:  resultQueue : Queue<Trie>
3:  subAnchors : Array<AnchorAstNode>
4:  subChildren : Array<Trie>
5:  resultQueue ← newEmptyQueue
6:  for all childNode in children do
7:      if childNode.value is AnchorAstNode then
8:          subAnchors ← appendToArray(anchors,childNode.value)
9:          subChildren ← collectNextInputs_Helper(parentNodeValue, subAnchors, childNode.children)
10:         resultQueue.enqueueAll(subChildren)
11:     else if childNode.value is EmptyAstNode then
12:         subChildren ← collectNextInputs_Helper(parentNodeValue, anchors, childNode.children)
13:         resultQueue.enqueueAll(subChildren)
14:     else if checkAllAnchorsMatchBetween(parentNodeValue,childNode.value,anchors) then
15:         resultQueue.enqueue(childNode)
16: arrayFromQueue(resultQueue)
```

B. függelék

Ábrajegyzék

Ábrák jegyzéke

C. függelék

Projektlinkek

- GitHub organization:

  `https://github.com/miniconnect/`

- Használati példák:

  `https://github.com/miniconnect/general-docs/tree/main/examples`

- HoloDB projekt:

  `https://github.com/miniconnect/holodb`

- Docker Hub repository:

  `https://hub.docker.com/r/miniconnect/holodb/tags`

- `LargeInteger` benchmark:

  `https://github.com/miniconnect/miniconnect-api/tree/master/projects/lang/src/jmh/java/hu/webarticum/miniconnect/lang`

- Permutáció-összehasonlító:

  `https://github.com/miniconnect/holodb/tree/master/projects/core/src/lab/java/hu/webarticum/holodb/core/lab/permutation`

- HoloDB integrált benchmark:

  `https://github.com/davidsusu/holodb-tdk/tree/main/benchmark`