

TDK-dolgozat

Horváth Dávid

`horvathdown@student.elte.hu`

HoloDB: Relációs demóadatok konzisztens 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

1. Bevezetés	1
1.1. Problémafelvetés	1
1.2. Tanulmány	2
1.2.1. A probléma státusza	2
1.2.2. Kérések mockolása on-the-fly	2
1.2.3. Seed-alapú virtuális világok	3
1.2.4. Relációs adatok generálása	4
1.2.5. Relációs adatok anonimizálása	4
1.2.6. Nagy teljesítményű megoldások	5
1.2.7. Miért csak kerülő megoldások vannak?	5
1.3. Az új megközelítés	6
1.4. Alkalmazhatóság	7
1.5. Célkitűzések	7
1.6. Megjegyzés a pályázat koncepciójához	8
1.7. JMH mérési környezet	8
2. Architektúra	9
2.1. A nagy illúzió: adatok a semmiből	9
2.2. A prototípusról	11
2.3. A relációs storage API	13
2.4. Az írhatósági réteg	14
2.5. Konfiguráció	15
3. A virtuális adattár építőkövei	18
3.1. Alapfeltevések	18
3.2. Segédtypusok	19
3.2.1. A fő segédtypusok áttekintése	19
3.2.2. A LargeInteger típus	20
3.2.3. Hierarchikus véletlengenerátorok	22
3.2.4. Monoton méretigazítás	23
3.2.5. Permutációk	24
3.3. Egyoszlopos kereshető értékkiosztások	29
3.3.1. Általános megfontolások	29
3.3.2. Keretrendszer az értékkiosztásokhoz	29

3.3.3.	Általános kétlépéses értékkiosztás	29
3.3.4.	<i>NULL</i> értékek kezelése	32
3.3.5.	Egyszerű értékkiosztások	32
3.3.6.	Unique értékkiosztás	33
3.3.7.	Reguláris kifejezésből képzett értékkészletek	33
3.3.8.	Full-text indexelt értékkiosztás	40
3.3.9.	Földrajzi koordináták értékkiosztása	42
3.4.	Egyéb értékkiosztási módok	43
3.4.1.	Nem-indexelt egyoszlopos értékkiosztások	43
3.4.2.	Értékkiosztás oszlopok közötti összefüggéssel	44
3.4.3.	Táblák közötti kapcsolatok	45
4.	Empirikus tapasztalatok, összehasonlítás	46
4.1.	Mérési környezet	46
4.2.	Általános futtatási tapasztalatok	46
4.3.	Lekérdezések sebessége	48
4.4.	Integrált tesztek eredménye	50
4.5.	Adatgenerálás, indítás és tárigény	51
4.6.	Terheléses teszt	52
5.	Speciális használati példák	54
5.1.	Dev mode (live mode)	54
5.2.	Zero mode	54
5.3.	Valós címadatok használata	56
5.4.	Valószerű geográfiai értékek előállítása	57
6.	Összegzés	58
6.1.	Az eredmények értékelése	58
6.2.	A célkitűzések teljesülése	58
6.3.	Tervek a közeljövőre	60
	Hivatkozások	61
A.	Algoritmusok	64
	Feistel-hálózat last-zero változat	64
	Nyers szófa összeállítása	64
	Nyers szófa kompakttá alakítása	65
B.	Táblázatjegyzék	66
C.	Ábrajegyzék	67
D.	Projektlinkek	68

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). Nem csak a demóadatok előállítása (jellemzően generálás vagy anonimizálás) jelent folyamatos terhelést a fejlesztési infrastruktúrára, de azok indítási, működtetési és memóriaigénye is jelentős.

A pályamunkában egy újszerűnek szánt megközelítést mutatok be, mely szükségtelessé teszi az adatok előzetes generálását. 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. Számos részmegoldás (például egyedi egész-aritmetika, kompakt szófák, kriptográfiai módszerek) együttes alkalmazásával koherens és gyors működést biztosító adatelérési technikák váltak lehetővé (például a virtuális indexek), így a lekérdezési eredmények hatékonyan számíthatók on-the-fly.

Az on-the-fly generálás miatt maguk az adatok nem foglalnak memóriát, ugyanakkor a szerver egy lekérések között is koherens adathalmazt szolgáltat. Így a HoloDB, alkalmas skálázás mellett, runtime performanciában is összemérhető a hagyományos adatbázisokkal. A relációs modell a megfelelő adapterekkel könnyen kiterjeszthető más adatmodellekre is.

Az új megoldás további előnyök mellett ötvözi a korábbi megoldások erősségeit, míg kikerüli a főbb hátrányokat, különösen a generálási költséget. A HoloDB potenciálisan széleskörűen alkalmazható adatbázismockoló eszközként, és a meglévő folyamatokba is könnyen integrálható. Továbbá olyan új lehetőségek is megnyílnak, mint a „dev mode” (az adatbázis azonnali frissülése a konfigurációs fájl módosításakor) vagy a serverless adatbázis-szolgáltatás.

ACM: Classification:

1. *Information systems / DBMS engine architectures*
2. *Software and its engineering / Development frameworks and environments*
3. *Software and its engineering / Software prototyping*

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. Paradox helyzet, melynek megoldása is abszurd: nagy mennyiségű adatot gondosan lematerializálunk, hogy aztán javarészt 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 fájlról (vagy akár közvetlenül az alkalmazásban definiált adatosztályok szerkezetéről) függővé tenni.

A dolgozatban elsősorban ezekre a problémákra kínálok egy újszerű megoldást. Mivel a bemutatott szoftver egyúttal egy általános keretrendszer virtuális adattárházhoz, további alkalmazások is adódhatnak.

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 ide kapcsolódnak, némileg hasonlókat 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, 11, 12].

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[13].

Ebbe a kategóriába sorolhatók az úgynevezett faker segédkönyvtárak is. Ezek olyan véletlenadat-generálók, melyek bizonyos gyakori entitástípusok adatmezőihez (pl. személynevek, e-mail, dátum) generálnak véletlenszerű értékeket[14, 15].

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

1.2.3. Seed-alapú virtuális világok

Játékokban és egyszerűbb szimulációkban már igen régóta generálnak kisebb világokat, pályákat procedurálisan valamilyen pszeudóvéletlen algoritmus segítségével[16]. Ennek azonban komoly méretbeli korlátai vannak. Felmerül a gondolat, hogy lehetséges-e óriás méretű világokat koherensen 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 valamilyen 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[17, 18, 19].

Egy közismert korai példa seed-alapú módszert alkalmazó szoftverre az *Elite* nevű 1984-ben kiadott űrhajós videójáték[20, 21]. A játékban meglátogatható bolygók kapnak egy-egy származtatott 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 például a *Starflight* (1986)[22], a *Minecraft* (2011)[23], a *StarMade* (2012)[24], a *No Man's Sky* (2016)[25] vagy a *Starfield* (2023)[26].

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, ami tárolható egy verziókezelhető, dokumentálható, könnyen szerkeszthető konfigurációs fájlban[27, 28, 29].

1.2.5. Relációs adatok anonimizálása

Ha már rendelkezésre áll egy éles adatbázis, kézenfekvőnek látszik, hogy ennek replikáit használjuk a teszt- vagy mockadatbázis kiindulópontjaként, így rögtön egy valószerű adathalmazzal indulunk. Az érzékeny adatok miatt azonban anonimizálásra lehet szükség. tehát a generáláshoz szükséges számítási igényt összességében felcseréljük a másolás és anonimizáció költségeivel[30, 31].

Meglévő adatokból kiindulni néha az anonimizálással együtt is kezelhetőbb, mint egy megfelelő generálási folyamatot kiépíteni[32]. 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 autentikáció (a konkrét megoldástól függően) biztonsági kockázatot rejthet.

Egy általános megoldás tehát aligha épülhet anonimizálós 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[33].

Egyes platformok általános megoldást nyújtanak az adatgenerálásra, anonimizációra és CI/CD-integrációra. Ilyen például a *Tonic*[34], a *GenRocket*[35] és a *Delphix*[36].

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 körében) 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 persze nem fér bele a fejlesztők idejébe sem.

Néhány ezzel kapcsolatos tényezőt tényyszerűen megállapíthatunk.

- **Érdeklhiány:** 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 sejtés a részemről, mennyire közvetlen okozói ezek (illetve ezek összjátéka) a vizsgált kínálati résznek. Érdemes lenne ezt a kérdést a későbbiekben külön tanulmányban alaposabban is megvizsgálni, nagyban építve egy 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, amely 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 koherencia biztosított, vagyis az egymás utáni lekérdezések egy ép, kerek adatbázis képét adják. Önmagában ezt 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 memóriefogyasztás 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 linkelek.

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, olyan helyeken 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 pályázat koncepciójához

A dolgozat egy újszerűnek szánt megoldást mutat be a fentebb megfogalmazott problémára. Fő célja az elképzelés alapvető működőképességének bizonyítása. Ez elsősorban annak a láncnak az elméleti bemutatását jelenti, mely a különféle részmegoldásokat egybefűzi, hogy ezáltal létrehozza a relációs adatbázis illúzióját (miközben megfelel ez imént megfogalmazott célkitűzéseknek). Mind a fejlesztés, mind a dolgozat írása során elsődlegesen a szoftvertechnológiai és architektúrais szempontokra koncentráltam. A projekt nem egy már bejáratott nagyobb kutatási program része, így az egyes részproblémákra eddig nem volt lehetséges külön-külön humán erőforrást dedikálni. Az implementációk konkrét algoritmusainak optimalizációjára, kiterjedtebb mérési eredmények produkálására és analízisére, illetve a szélesebb összehasonlító elemzésre majd csak egy későbbi fázisban kerülhet sor.

1.7. JMH mérési környezet

A dolgozatban szereplő, mikroperformanciára vonatkozó méréseket JMH (*Java Microbenchmark Harness*) keretrendszerben végeztem, (jelenleg ez számít standardnak Java környezetben). Ezeket méréseket a megfelelő Gradle taszkon keresztül futtattam. A mérések a forráskód részei, projektenként a `jmh` forrásmappában találhatók (link a forráskódhoz a dolgozat végén, a Projektlinkek függelékben található). Minden benchmark osztályt az alábbi annotációkkal paramétereztem:

```
1 @State(Scope.Benchmark)
2 @Fork(value = 1)
3 @Warmup(iterations = 1, time = 1, timeUnit = TimeUnit.SECONDS)
4 @Measurement(iterations = 1, time = 1, timeUnit = TimeUnit.SECONDS)
5 @BenchmarkMode(Mode.AverageTime)
6 @OutputTimeUnit(TimeUnit.NANOSECONDS)
```

Futtatókörnyezet:

- Gradle 8.10.2, `me.champeau.jmh` 0.7.2, JMH 1.37
- Debian 12 GNU/Linux, OpenJDK 17.0.13
- Intel Core i5-13500H, 16GB RAM

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 táráat 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[37], de egyúttal egy olyan általános adatelérési normalizációs szabvány, mely közös nevezőként tud szolgálni más modellekhez is (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 lépünk 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, *értékkiosztásnak* 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 keresetőség 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 sok felmérendő, kikísérletezendő dolog adódik.

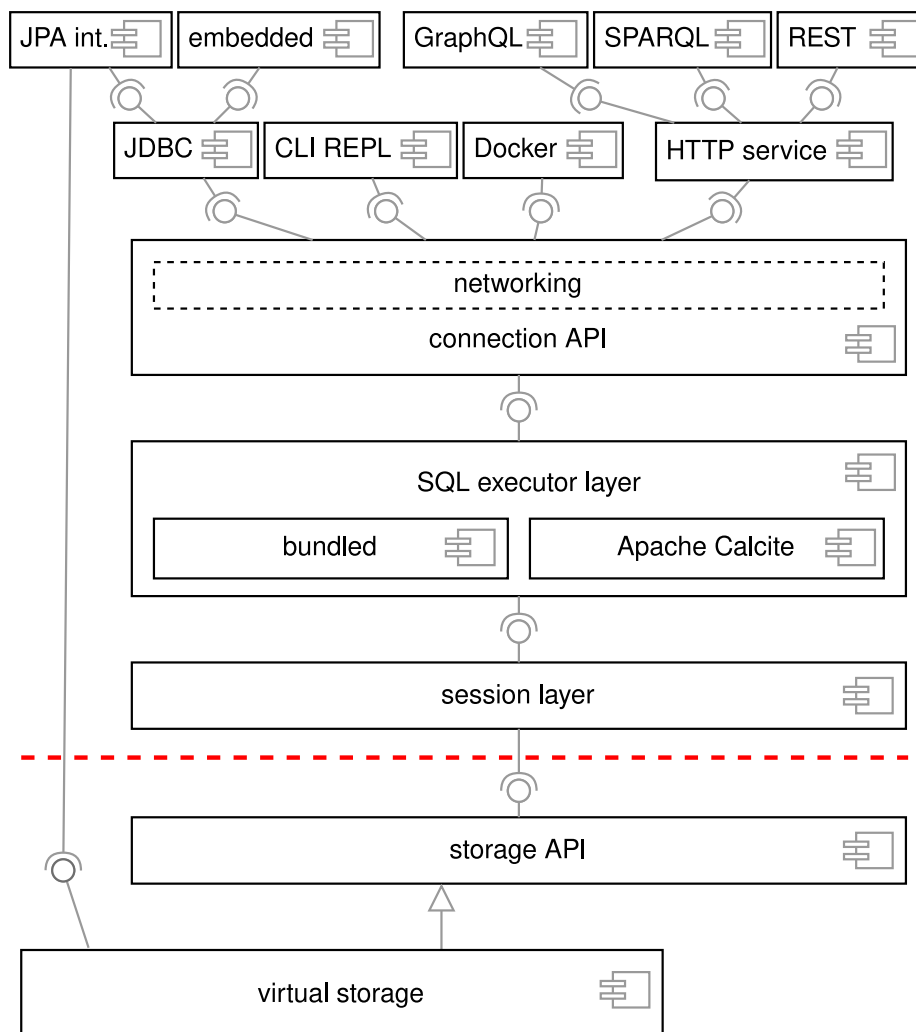
A szerver Docker konténerként is futtatható, amihez a Docker Hubon biztosítok egy testreszabott képet. A felhasználói Dockerfile mellé csak a konfigurációs fájlt kell elhelyezni az indításhoz. A testreszabott kép a kis erőforrásigény miatt serverless végpontként is jól használható.

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 nehézsége motivált. A MiniConnecthez middleware változatok is elérhetők, például hálózaton keresztül is használható, illetve letisztultsága miatt könnyen írhatók hozzá dekorátorok, proxyk. Debian csomagként is elérhető egy interaktív parancsfuttató (REPL), mellyel parancssorból tudunk hálózaton keresztül csatlakozni a MiniConnectet támogató tetszőleges adatbázishoz. Ez egyúttal a HoloDB kipróbálásának legkényelmesebb módja, különféle kényelmi funkciókkal (szép kimenet, szintaxiskiemelés, automatikus kiegészítés, parancstörténet stb.).

Az architektúra köré sokféle eszközt fejlesztettem, melyek a robusztusság eltérő fokain állnak. Ilyenek például: automatikus REST API, NoSQL adapterek (MongoDB, SPARQL, GraphQL), autentikációs proxy, Apache Calcite[38, 39] SQL driver stb. Megjegyzem, hogy a HoloDB nem támogatja natívan dokumentumok vagy szemantikus adatok tárolását, hanem a NoSQL API-k háttértáraként használható. Az ehhez készült adapterek még meglehetősen kísérleti fázisban vannak, egyelőre elsősorban az elméleti lehetőségekre igyekeztem rámutatni.



2.1. ábra. A virtuális adatbázis vázlatos architektúrája

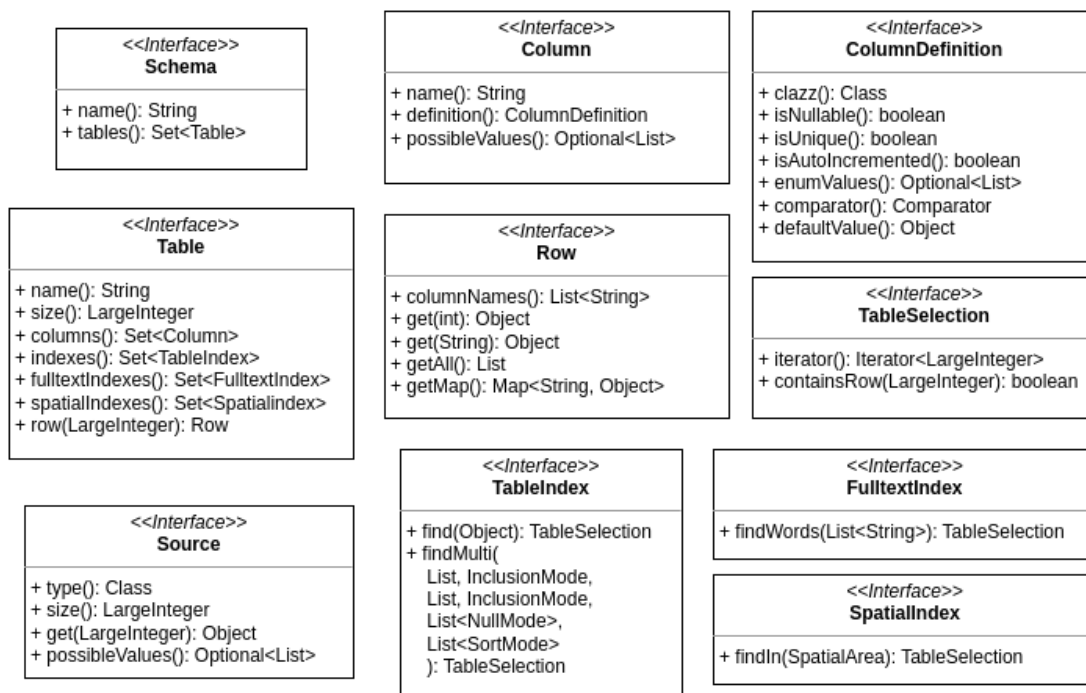
2.3. A relációs storage API

Ha már körvonalaztuk a relációs adathozzáférés fő entitásait, ezeket rendre implementálva egy funkcionálisan teljes körű relációs adattárat kapunk. Az entítások 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 egy rekordorientált nézet. 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[40]). A másik két típus találatinak 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 scenáriókban főként egyes konkrét rekordok lesznek módosítva, hozzáadva, törölve. A tömeges módosítások kérdésével itt nem foglalkozom, de nem látszik akadály a ezek optimalizálását célzó továbbfejlesztésnek.

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.

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

A tranzakciók kezelésénél megmaradunk 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.

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

Új módosító művelet érkezésekor a ciklus újratezdődik.

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 savepontra 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 szerver kész Docker konténerrel is indítható, tehát dockeres környezetben a konfigurációs fájl mellett pusztán a megfelelő *docker* parancs kiadására van szükség az indításhoz.

A konfigurációs fájl alapértelmezetten YAML formátumú, de más adatleíró nyelv is használható a megfelelő Jackson mappingen keresztül. A Jackson adatserializációs keretrendszer Java objektumokat, adatstruktúrákat, különböző adatleíró nyelveken írt fájlokat, karakterláncokat és más adatrepresentációkat konvertál egymásba[41].

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 (`config` projekt). A konfigurációs adatstruktúra a konfigurációs osztályok közvetlen használatával programozottan is felépíthető.

```

1 seed: 425364
2 schemas:
3   - name: shop
4     tables:
5       - name: customers
6         size: 5
7         columns:
8           - name: id
9             mode: COUNTER
10          - name: firstname
11            valuesBundle: forenames
12          - name: lastname
13            valuesBundle: surnames
14          - name: birth
15            valuesRange: [1950, 2000]
16      - name: orders
17        size: 12
18        columns:
19          - name: id
20            mode: COUNTER
21          - name: cid
22            valuesForeignColumn:
23              [customers, id]
24          - name: product
25            valuesBundle: fruits
26          - name: quantity
27            valuesRange: [1, 10]

```

⇒

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

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[42] projekt esetén JPA-entitásokra helyezett annotációkkal is leírható a teljes konfiguráció, így beágyazott adatbázissal, külső eszközök nélkül is futtathatunk teszteket. Az annotációk eléréséhez egy minimalisztikus csomagot kell beemelni (jpa-annotations projekt). A mockolást használó projektben a megfelelő függőség beemelésén (jpa projekt) túl elegendő a megfelelő JDBC connection sztringet megadni, a többi (keretrendszeről függően²) automatikusan elvégzi az inicializáló szolgáltatás.

Az egyéni működéssel való kiegészítés több szinten támogatott. A beépített értékiosztási módok esetén is lehetőség van egyéni adatfájl betöltésére, a permutációk és rányújtások egyéni implementációjának használatára. De maga az értékiosztási mechanizmus is lecserélhető egyedi implementációra, ehhez meg kell adnunk egy olyan osztály nevét, mely implementálja a

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

SourceFactory interfészt (illetve biztosítani kell, hogy az egyedi osztályok láthatóak legyenek a classpath alatt). Az egyéni működések egy része közvetlenül a konfigurációból is hivatkozható:

```
1 - name: some_column
2   sourceFactory: org.example.myorg.SomeCustomDataFactory
3   sourceFactoryData:
4     additionalParameter1: 10
5     additionalParameter2: 'lorem ipsum'
6 - name: some_column
7   valuesResource: custom-data.txt
```

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 megfeleltetési mutatót[43]. 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.

³Ilyen megközelítést használ például a *Helm* nevű *Kubernetes* platformhoz elérhető 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 elérhető legyenek a storage API-n keresztül. A megfeleltetés ez esetben két dolgot jelent:

1. **Lekérdezések közötti koherencia:** 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 ez a két hozzáférési mód hatékony, akkor nincs elvi akadály, hogy a lekérdezések jelentős részénél a tényleges adatbázisokéval összemérhető performanciát érjünk el.

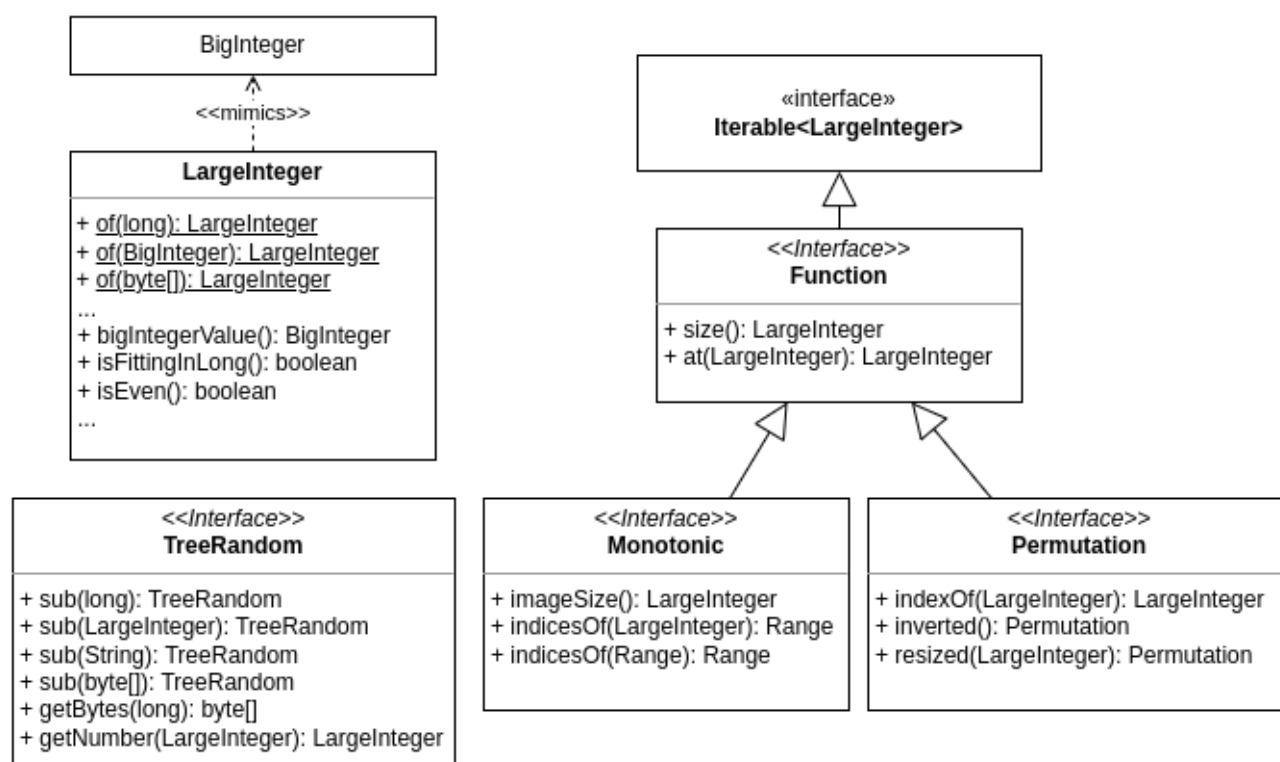
A storage API-ban úgy definiáltuk a `TableIndex` interfészt, hogy a keresésen kívül még néhány további funkciót is megköveteljen, például a rendezést és a `NULL` értékek kezelését. Ahol szükséges, külön kitérek majd az ezzel kapcsolatos problémákra.

3.2. Segédtypusok

3.2.1. A fő segédtypusok 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

3.2.2. A `LargeInteger` típus

A hagyományos, tényleges adatbázisokban az egyik szűk keresztmetszetet az adatok fizikai elérése jelenti, esetünkben azonban a számításokra helyeződik át a hangsúly. A hatékony on-the-fly számításokhoz pedig szükségesnek mutatkozott egy olyan egész típus megalkotása, mely az alábbi feltételeket teljesíti:

1. Kis számokon is nagyon hatékony (ellentétben a beépített `BigInteger` típussal).
2. Tetszőleges méretű egész számokat támogat, elfogadható hatékonysággal.

Az ecélből 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 túlszordulás; 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 túlszordulás. Például nem zéróval történő szorzás után az eredményt visszaoszthatjuk a szorzóval, és ha nem egyezik a szorzandóval, akkor tudjuk, hogy túlszordulás volt. `BigInteger` számítás esetén ellenőrzésre kerül, hogy az eredmény 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` konstruktora kívülről nem elérhető, ehelyett a statikus `of()` metódusokon keresztül lehet példányokhoz jutni. Feltehető, hogy a zérusközelie értékek a leggyakrabban használtak (gondoljunk például olyan triviális számokra, mint 0, 1, 2, 10), az ezekhez tartozó példányok statikusan tározva 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).

Amikor a `LargeInteger` megfelelő implementációjával kísérletezni kezdtem (a 2019-2021 időszakban), párhuzamosan a Scala nyelv standard könyvtárába is bevezetésre kerül egy új működési elv a beépített `BigInt` osztályhoz[44]. Az eredeti `BigInt` egy egyszerű csomagoló osztály volt a Java `BigInteger` fölött, az új megoldás pedig a `LargeInteger`-hez hasonlóan primitív `long` értékeket

használ a kis számokhoz (nagy részt a scalás Spire matematikai keretrendszer `extttSafeLong` osztályát alapul véve[45]). A `BigInt` végül is csak egyetlen osztályt definiál, melynek van egy `long` és egy `BigInteger` típusú mezője is, melyek közül mindig az egyik van használva. A JVM általában 12 bájtos fejléctet tárol az objektumokhoz, és 4 bájtos tömörített címezést használ a referenciákhoz, ha a heap mérete 32GB alatti, a `long` pedig 8 bájtot foglal. Így a `BigInt` memóiafoglalása kis számokon összesen 24 bájt. A `LargeInteger` kis számokra ugyan csak 20 bájtot foglal, a legtöbb JVM azonban 8 bájtos egységeket kezel, és padding bájtokat ad hozzá, így végül az is 24 bájtra jön ki. Ilyen szempontból a `BigInt` osztályszerkezete szerencsésen kezelődik. Nagy számok esetén a `LargeInteger` vékony helyfoglalása csak 16 bájt, bár ekkor már kevésbé számít.

Abban a döntésben, hogy miért nem tértem át a `BigInt` vagy a `SafeLong` típus használatára. főként az alábbi megfontolások játszottak szerepet:

- A scalás osztályok javás használata nem idiomatikus. Például beszédes metódusnevek helyett az operátortúlterhelés kódolt nevet kell használnunk (`n.shiftLeft(m)` helyett `n.$less$less(m)` stb.).
- Nemkívánatos külső függőséget jelentene, mely az implementációs szabadságot és a könyvtár rekurzív méretét is növelné.
- A `LargeInteger` teljesen replikálja a beépített `BigInteger` metódusait. Így ismerősebb lehet azok számára, akik az utóbbit már használták. Emellett a refaktorálás is egyszerű az új típusra.
- A `LargeInteger` kényelmi metódusokat is ad (`n.isGreaterThan(m)`, `n.increment()` stb.).
- Kis számokon a legrelevánsabb műveletekre a `LargeInteger` gyorsabb. (Igaz, vannak helyzetek, amikor éppen a `BigInt` gyorsabb, de minden bizonnyal a `LargeInteger` is bőven optimalizálható még).

Néhány további létező implementációval is kísérleteztem, bár ezek meglehetősen gyenge teljesítményt mutattak. Az egyetlen ezek közül, amit érdemes megemlíteni, az `Apint` a típus C++-ból Java nyelvre portolt `Apfloat` könyvtárból. Azonban ennek teljesítménye még a beépített `BigInteger`-ét is messze alulmúlja¹, illetve sok műveletet közvetlenül egyáltalán nem (például bitszintű műveletek), vagy csak az `ApfloatMath` segédosztályon keresztül támogat.

JMH-val lemértem az említett implementációk hatékonyságát. Az eredményeket a 3.1 táblázat tartalmazza. Látható, hogy a `LargeInteger` jól teljesíti a fentebb megfogalmazott elvárásokat.

¹ Valószínűleg azért, mert az `Apfloat` könyvtár elsősorban lebegőpontos számításokra lett kifejlesztve. Maga az `Apint` is csak egy `Apfloat` példány csomagolója.

	SMALL				LARGE	MIXED		
	SADD (ns)	SSIMP (ns)	SNORM (ns)	SPOW (ns)	LADD (ns)	MBTL (ns)	MADD (ns)	MCALC (ns)
long	1,1	2,6	5,3	16,7	N/A	N/A	N/A	N/A
BigInteger	36,8	20,9	79,5	92,3	39,2	11,6	136,6	109,8
LargeInteger	5,2	5,4	8,1	8,1²	53,8	17,3	222,7	148,2
BigInt	6,0	6,9	22,3	35,6	46,0	13,7	192,8	139,1
SafeLong	12,0	7,3	15,0	24,6	52,3	16,6	1002,1	1613,2
Apint	558,6	973,5	2405,7	2736,6	471,4	85,6	1768,8	2274,4

3.1. táblázat. Óriás egész szám implementációk jellemzői. **A sorok jelentése:** long: a Java primitív long típusa; BigInteger: a Java beépített BigInteger típusa; LargeInteger: az általam implementált LargeInteger; BigInt: a Scala beépített BigInt típusa; SafeLong: a scalás Spire matematikai könyvtár SafeLong típusa; Apint: a javás Afloat matematikai könyvtár Apint típusa. **Az oszlopok jelentése:** SMALL: műveletek a long értéktartományán belül (overflow nélkül); LARGE: műveletek a long értéktartományán kívül; MIXED: műveletek vegyes értéktartományon (overflow is előfordul). A második sor jelentése: SADD: $\sum_{i=1}^7 a_i$; SSIMP: $\frac{(a+b)c}{d}$; SNORM: összetett számítás; SPOW: összetett számítás, hatványozás is; LADD: $\sum_{i=1}^5 X_i$; MBTL: $a + X$; MADD: összetett vegyes összeadás overflow-val; MCALC: összetett vegyes számítás overflow-val. A képletekben a kisbetűk long értéktartományán belüli, a nagybetűk pedig azon kívüli megfelelő véletlenszámokat jelentenek.

3.2.3. Hierarchikus véletlengenerátorok

Az adatbázist egy hierarchikus képződményként is fölfoghatjuk: a sémákon belül táblák, azokon belül oszlopok, azokon belül pedig adatmezők foglalnak helyet. Végül a mezők értéke is lehet strukturált adat, melyben a rétegződés folytatódik. Szeretnénk, hogy az értékek a hierarchiában bárhol nagyjából *véletlenszerűen*, de a globális kiinduló seed alapján determinisztikusan álljanak elő, Olyan véletlengenerátorra van szükség, mely teljesíti a következő követelményeket:

- tetszőleges számú bitet képes generálni, mindig ugyanazt a (végtelen) bitsorozatot adva vissza
- egy kulcs bájtsorozat megadásával visszaad egy példányt (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.

²Ha a pozitív kitevő nem túl nagy, akkor a LargeInteger optimális szorzást végez. A long tesztjében a beépített Math.pow függvényt használtam, amelyben nincs ilyen optimalizáció.

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).³

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 algoritmus, ha a hash alapján előállított seeddel inicializált Random példánnyal rendre visszaadott bájtokat használjuk.

3.2.4. 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. 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 beszűrhető 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 példány 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

³Egy alapértelmezett limittel, hogy implementációtól függetlenül ne alakulhasson ki végtelen ciklus.

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 előbbi.

A legegyszerűbb implementáció a diszkrét lineáris vetítés. Az n hosszúságú értékkészletet ekkor úgy vetítjük az N listahosszra, hogy az i -edik helyre az $\lfloor \frac{n \cdot i}{N} \rfloor$ 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[46].

A mintavételezéses eljárás többféleképpen is továbbsofizisztikálható. Egy ilyen lehetőség a szűrjektív eredmény biztosítása, amikor szeretnénk, ha minden érték előfordulna (az egyenetlen 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.

3.2.5. 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.

Permutációk méretigazítása

A permutáció-implementációk egyes típusai nem támogatnak tetszőleges hosszt. Nem támogatott hossz esetén egy megfelelő dekorátor objektum alkalmazása szükséges, mely valamilyen

méretigazítást valósít meg. Általában a *rejection sampling* technikát érdemes bevetni. Az adott n mérethez vesszük azt a legkisebb N támogatott méretet, amely n -nél nem kisebb. Amikor egy adott i értékre a dekorált permutációtól az intervallumon kívül eső p -t kapunk, azaz amelyre $n \leq p (< N)$, akkor újra meghívjuk a dekorált permutációt p -re. Addig ismételjük a hívást, míg végre n -nél kisebb szám nem születik.

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: $\sigma_{+1}^n(i) = (i+1) \bmod n$. Természetesen számos egyéb módszer is felmerülhet a méretigazításra.

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 újramezhető, változtatható méretű permutációk könnyen analógiába állíthatók kriptográfiai titkosítófüggvényekkel. Ezek dekódolhatók, kulcs alapján újramezhetők, 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.

Az összetett titkosító eljárások keretrendszereként igen népszerű Feistel-hálózatok épp ilyenek[47]. 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-al ráfésüljük a célhelyre. Az XOR tulajdonságai miatt ez reverzibilis (bármilyen 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 a minden körhöz más hashelést 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)

A páratlan blokkméret lehetősége a hatékonyság szempontjából is fontos, mivel jelentősen csökkenti a Feistel-hálózatra történő újrachívások számát. 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.

FE1 sémájú FPE

Tetszőleges méretű permutáció állítható elő az FE1 sémájú FPE (*Format Preserving Encryption*) használatával[48, 49]. Ez is egy többkörös, hashelésen alapuló eljárás. Ennek egy létező Java-portjából kiindulva[50] implementáltam egy tetszőleges hosszúságot támogató permutációt. Bár ez a megoldás nagyon jó keveredést ad, az alternatívákhoz képest meglehetősen lassúnak tűnik.

Egyszerű heurisztikák

A triviális implementációk gyenge keveredés mellett nagyon jó sebességet biztosítanak és memóriatakarékosak, az erős kriptográfiai megoldások pedig erős keveredés mellett nagyobb erőforrásigényűek. A Feistel-hálózatok egyik előnye éppen az volt, hogy a hasheléssel, a körök számával, és egyéb paraméterek állításával viszonylag jól skálázhatók. Azonban még a legegyszerűbb Feistel-hálózat esetében is van egy padlója az erőforráshasználatnak. Felmerül tehát a kérdés, hogy milyen skálázási lehetőségek adódnak a triviális implementációk és a Feistel-hálózatok között.

A legegyszerűbb módszerek kombinációinak próbálgatásával igyekeztem olyan megoldást találni, amely ránézésre (ha nem is kriptográfiai mércével) már erős keveredést adnak. Úgy gondoltam, a lineáris kongruencián alapuló permutációt (MP1) kombinálni lehetne más gyors eljárásokkal. Két jó jelölt kínálkozott: az egyszerű bitszintű XOR (BXP) és a bitek permutálása (BSP). Mindkettőnek hátránya, hogy bit-alapú, tehát nem kettő-hatvány hosszúság esetén méretigazítás szükséges, de ettől most eltekintek. Úgy találtam, hogy a $X \circ M_2 \circ S \circ M_1$ (FAV) kompozíció (ahol X egy BXP, S egy BSP, illetve M_1 és M_2 különböző lineáris kongruenciák) megfelel a céloknak: egyrészt gyorsabb, mint egy egykörpáros, gyors hasht használó Feistel-hálózat, másrészt nagyon jó keveredést ad.

A permutációk mért jellemzői

A 3.2 táblázatban tíz implementációt vettem össze egy 65 536 hosszúságú alaphalmazon.

A rekurzív memóriahasználatot a Java Instrumentation API segítségével mértem. Egyes implementációk a beépített Mac osztályt használják, jellemzően a HmacSHA256 beépített algoritmussal. Ezek a példányok jellemzően megosztottak, ezért lemértem a memóriát ennek kihagyásával is. A teljes permutációt egy tömbben tároló implementáció (MEM) saját adatai miatt foglal sok memóriát (int-értékenként 4 bájt, ami már önmagában is 262 144 bájt).

Eleve nem a kriptográfiai szintű véletlenszerűség volt a cél, így a keveredés méréséhez statisztikai eszközök helyett tömörítési eljárások hatékonyságát mértem. A 65 536 érték 16 bites, tehát 2 bájtos reprezentációkkal kiírható egy-egy 131 072 bájt méretű fájlba. Ezt a fájlt tömörítettem két erős tömörítővel, a bzip programmal, illetve a paq8 programmal (4-es erősséggel).

A példánylétrehozási és hívási időket JMH-val mértem. Az értékek lekérését és keresését az első 30 pozíción futtattam, így egyfelől nagyobb lett a minta, másfelől az identikus permutáció (ID) is kibújt az 1 ns érték alól. A FAV esetében a létrehozási idő nagy azért magas, mert a második MP1-hez új kulcsot generál. Ez valószínűleg elkerülhető, egyelőre ezzel nem foglalkoztam.

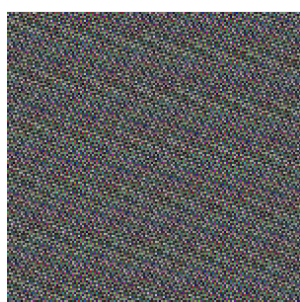
A 3.2 ábrán az első négy permutáció-típus értékeloszlását vizuálisan is összehasonlítottam. Látszik, hogy a FAV ránézésre a FEI-S-2 kevertiségi szintjét hozza. Minden pixelpozíció egy-egy index, és a pixel színe kódolja az értéket. A teljes minta egy 100×100-as részlete lett kiemelve (ami jól reprezentálja a teljes mintát). A színek a HSB komponensenként lettek kalkulálva: az árnyalat a legfelső bájt, a telítettség a legalsó bájt, a fényerő pedig a bájtok átlaga alapján.

	mem (bytes)	mem-ex (bytes)	paq8/4 (%)	bzip2 (%)	create (ns)	at×30 (ns)	iof×30 (ns)
FAV	648	648	31,54	99,77	968	2 040	2 007
MP1	152	152	0,19	22,13	267	161	188
FEI-F-4	608	608	19,56	98,19	660	8 800	9 185
FEI-S-2	212 312	336	97,08	99,95	434	32 221	32 536
FEI-F-1	240	240	2,66	90,58	347	3 234	3 137
ID	40	40	0,04	19,49	2	6	6
FPE1	211 352	160	97,07	99,94	540	67 155	68 192
BXP	72	72	0,06	39,93	136	82	82
BSP	208	208	0,09	34,55	108	904	904
MEM	262 208	262 208	97,09	99,93	686 358	82	148 220

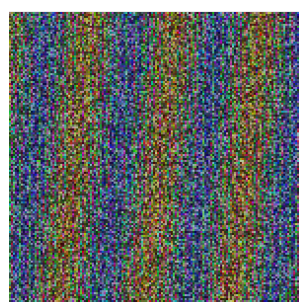
3.2. táblázat. Permutáció-implementációk jellemzői. **A sorok jelentése**⁴: FAV: az általam favorizált heurisztika (részletek a főszövegben); MP1: lineáris kongruencia; FEI-F-4: Feistel-hálózat (4 körpár, FastHasher); FEI-S-2: Feistel-hálózat (2 körpár, SHA-256); FEI-F-1: Feistel-hálózat (1 körpár, FastHasher); ID: identikus permutáció; FPE1: FE1 sémájú FPE; BXP: bit XOR (BitSet-tel); BSP: bit-permutáció (int-tömbbel); MEM: random kevert in-memory int-tömb használata. **Az oszlopok jelentése** mem: teljes rekurzív memóriaigény példányonként; mem-ex: u. a., de a megosztható Mac példány nélkül; paq8/4: méretváltozás 4-es erősségű paq8 tömörítéssel; bzip: méretváltozás bzip tömörítéssel; create: példány elkészítési ideje; at×30: első 30 index értéklekérésének ideje összesen; iof×30: első 30 érték indexkeresésének ideje összesen.



FAV



MP1



FEI-F-4



FEI-S-2

3.2. ábra. A Permutation különféle implementációinak kimenetei

⁴A sorokat ár-érték arány szerint csökkenően igyekeztem rendezni, figyelembe véve az értékeket és a vizuális eredményeket. Az összbnyomás mérésére nem sikerült egységes képletet találnom, így mondhatjuk, hogy a sorrend részben szubjektív.

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 koherencia

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

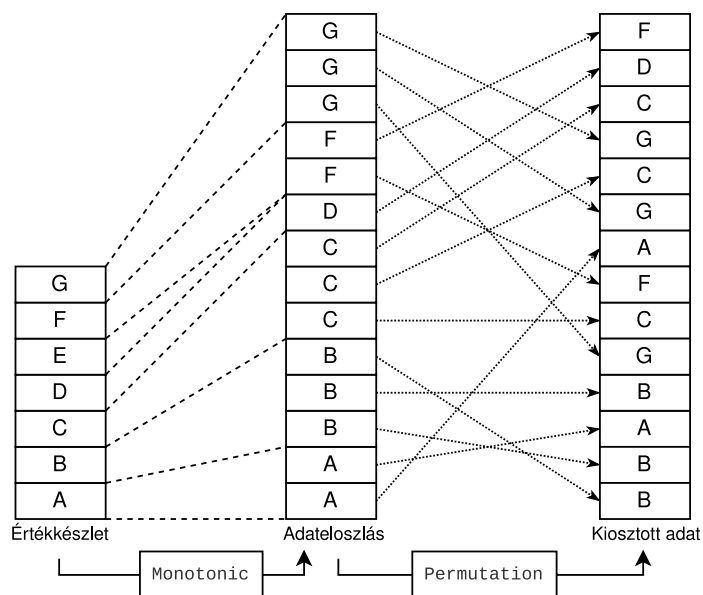
3.3.2. Keretrendszer az értékkiosztásokhoz

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 kompatibilisnek 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

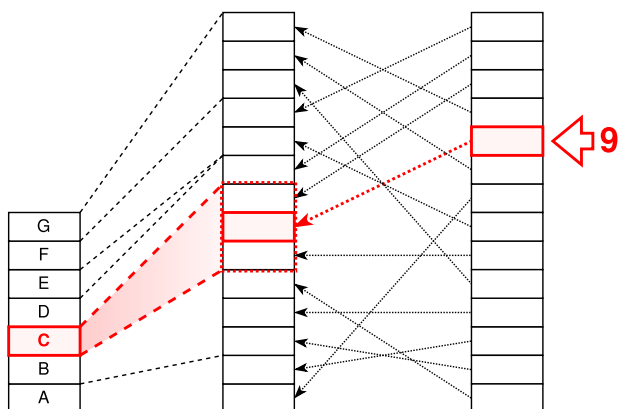
Talán a leggyakoribb eset, hogy előre ismert véges értékkészlet elemeit szeretnénk vizsgálni az oszlopban, mégpedig összevissza. Fentebb már utaltam rá, hogy ez a legtöbb esetben megoldható két független lépéssel. Először egy Monotonic példány segítségével a kívánt célhosszra értelmezett függvényt állítunk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



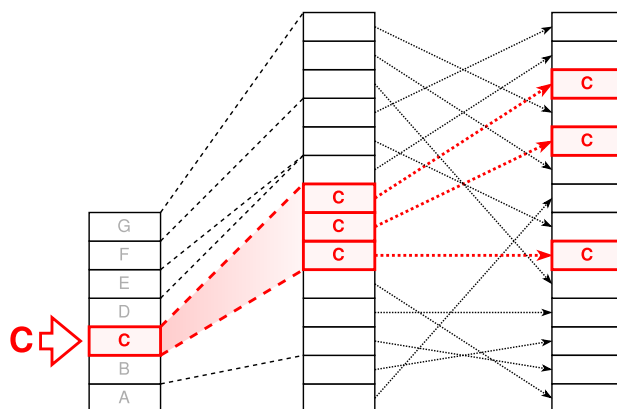
3.3. ábra. A kétlépéses értékkiosztás alapelve: egymás után végrehajtott visszafejthető disztribúció és permutáció

Láttuk, hogy mindkét függvény típus interfésze úgy van definiálva, hogy relációként megfordítható. A Monotonic interfész `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:



3.4. ábra. Adatlekérés a kétlépéses értékkiosztásból



3.5. 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 vizsgálni 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.

Egyenetlen monoton értékkiosztások

A rányújtási elv, amit előzőleg monoton függvények előállításához használtunk, 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.⁵

Az első lépés alapelve tehát hasonló az előbbi megoldás első fázisához. Ám itt nem a lehetséges értékeket nyújtjuk rá 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.⁶

Az érték elérése ekkor úgy történik, hogy először lekérjük az értéksávot a rányújtó függvényről, 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 érték tartozik. Ezen sorindexek feszítik majd ki a találati sorindexsávot. Hogy az alsó érték beletartozik-e,

⁵A 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.

⁶Ebben az esetben explicite ki kell zárni, hogy az utolsó sáv üres legyen. Ez legtermészetesebben egy logikai paraméter beiktatásával érhető el, amelyet a rekúzió 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.

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 egyenletlenné tudtuk tenni a kiosztást, 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, egyenetlen kimenetet szeretnénk látni.

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 hozzáadása, természetesen figyelni kell a speciális kezelésre a keresés-rendezés során.

A fentiek fényében az egyes értékkiosztások tárgyalásánál a *NULL* értékek kezelésével külön 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 sorszámaival, 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*[51] és a *generex*[52]). 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:

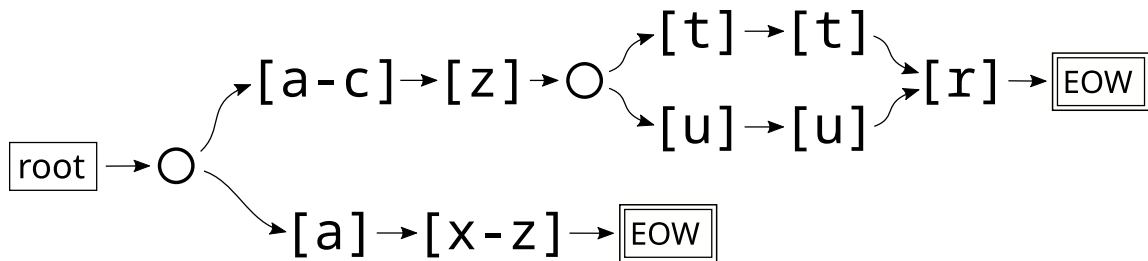
- a minta csak konstans karakterek, karakterosztályok, csoportok és alternációk szerepelnek
- a rendezés karakterenkénti, azaz nincs másodlagos, harmadlagos stb. összehasonlítási szempont

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[53].

A gráf lényegileg egy prefix fa, ahol egyes azonos részfákat újrahasznosítunk. Ezért a továbbiakban az egyszerűség kedvéért többnyire faként fogok a gráfra hivatkozni. A gyökértől az adott részfaig vezető útvonalra 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:



3.6. 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 csomópontozhoz (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 (gyermekkelemmel) 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

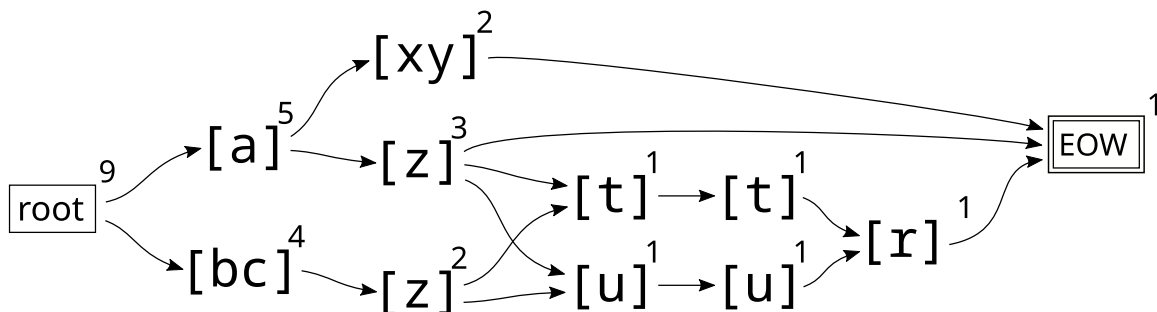
lépésben egy normalizálási eljárást hajtunk végre, mely után a gráfban adott csomópontból elérhető csomópontok listájára az alábbiak lesznek érvényesek:

- **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 értelemszerűen átveszik a korábbiakból elérhető csomópontokat. A `[c]` karakterosztály így mind a három eredeti csomópont gyerekeit átveszi (majd ezeket is rekurzívan tovább kell bontani). 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:



3.7. ábra. Egyszerű reguláris kifejezés kompakt szófája

Minden csomóponthoz hozzárendeltük a hozzá tartozó részfa leveleinek effektív számát, ami rekúzióval 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 $C_1, C_2, \dots, C_{n-1}, C_n$ 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 $C_i \setminus \{n\}$ 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.) elimináljuk rendre az alábbi szabályokkal:

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

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

1. $R_1 = A\{, 1\} \longrightarrow (A \mid)$
2. $R_n = A\{, n\} \longrightarrow (AR_{n-1} \mid)$ (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 elimináljuk 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[54]. 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 nagybetűség 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[55].

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[56]. 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.

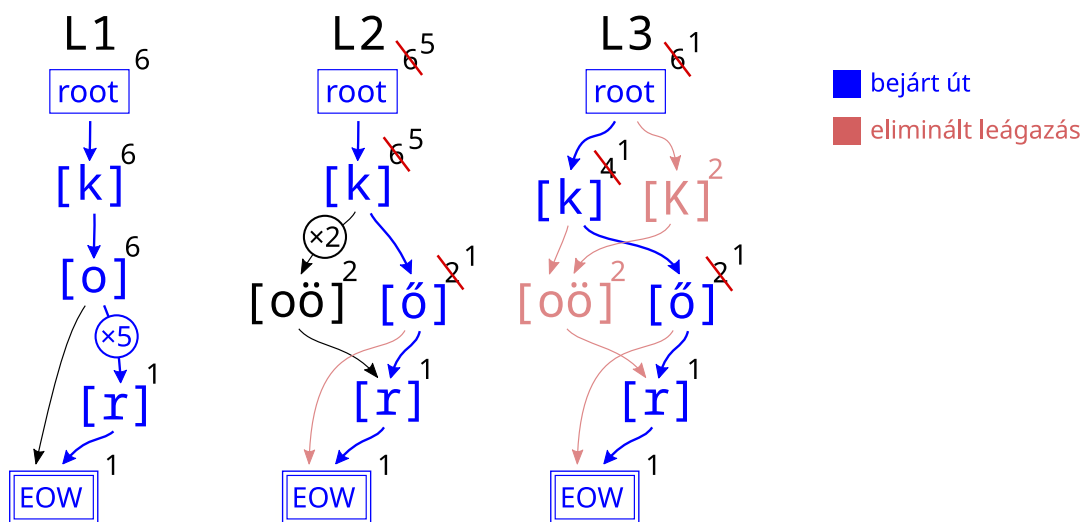
Az n-edik karakterlánc lekérése úgy történik, hogy az egymás alá fűzött fák rendszerében (ami maga is egy nagy faként értelmezhető) navigálunk. Az első fában normál módon keresünk, azzal a kiegészítéssel, hogy *szorzó csomópontokat* iktatunk be, melyek olyan részfák listájaként értelmezendők, melyek 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 hatékonyá 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 pozíciótartó összehasonlításokra szorítkoztunk, ez a megkötés eleve biztosított.

A következő ábra a `(([kK][oö]r|kõr?))` reguláris kifejezés többszintű kompakt szófájának viselkedését mutatja a `kõr` bemenet esetén:



3.8. ábra. Többszintű kompakt szófa viselkedése

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 azt is, hogy a 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 (némielg 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[57].

A szavakat egyszerűen betűkarakterek sorozataiként fogjuk föl, előzetesen függetlenül a központosítás, 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 szórend, a sávok egymáshoz képesti relatív hossza pedig a p_i szereplési valószínűségen alapul. Ezt a listát képezzük bele oszlopfolytonosan a mátrixba.

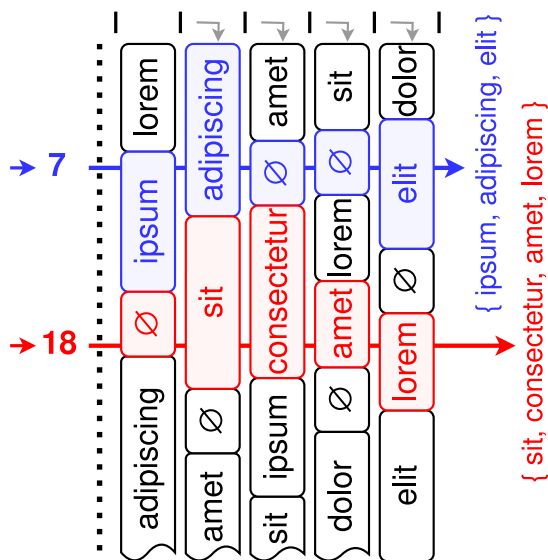
Egy adott sorindexhez tartozó mezőben szereplő szavakat úgy kapjuk, hogy egyszerűen vesszük a mátrix megfelelő sorát. Szavakra keresni pedig úgy tudunk, hogy a keresett szavakhoz tartozó

(esetleg oszlopváltással megszakított) sávok metszetsávját vesszük, ha nincs metszet, nincs találat. A mátrixsorokat az életszerűség kedvéért érdemes permutálni a fentebb már ismertetett módszerrel.

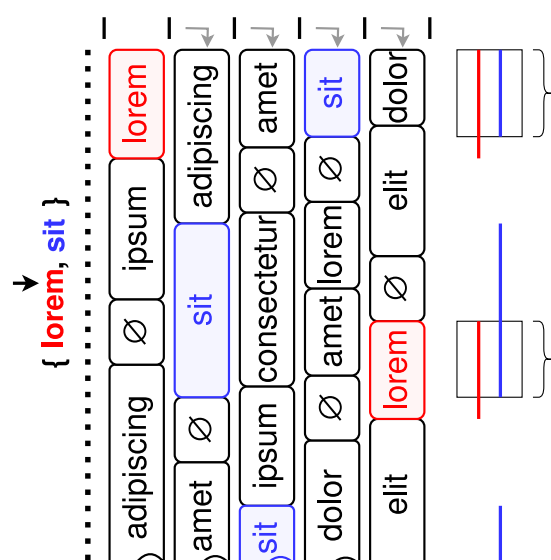
A mátrixos módszer egyik hátránya, hogy az egymás mellé került szavak eltartanak egymástól ugyanabban az oszlopban, így jellemzően nem tudnak majd bekerülni ugyanabba a szövegbe, illetve általában is viszonylag korlátolt, hogy melyek a lehetséges konstellációk. Ez viszonylag jól orvosolható azzal, ha 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, lásd 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 közötti koherencia 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 a halmazból, 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 központosítást.

Mindegyik lépés finomítható különféle heurisztikákkal, ezekkel most nem foglalkozom.

Könnyen integrálható továbbá a kizárandó szavak kezelése. A szóhalmaz kiválasztásának és a szöveg generálásának összehangolásával a kifejezésekre való keresés bizonyos formái is megvalósíthatóvá válnak.

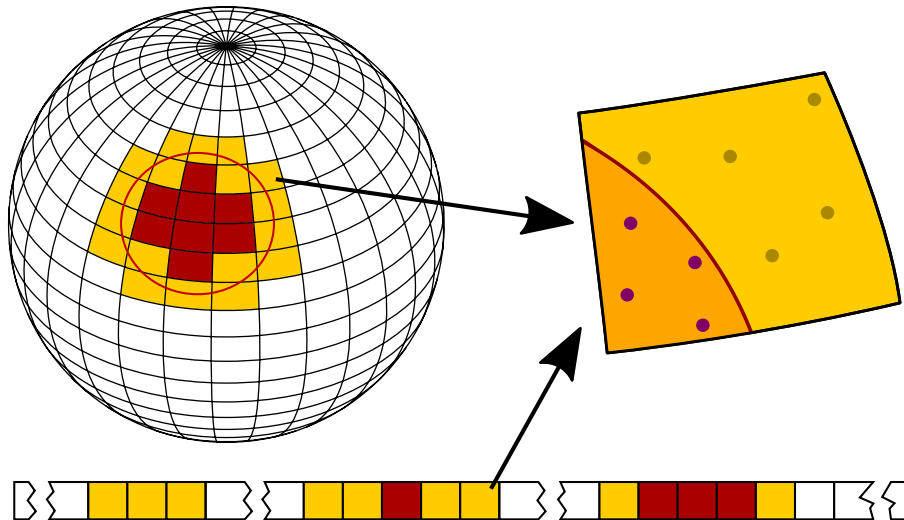
3.3.9. Földrajzi koordináták értékkiosztása

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 hosszúságunként, illetve adott h földrajzi szélességünként 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 belülségét. 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 szövegméretet nagybetűs szavakkal és központosítással (az eredmény tovább javítható előre adott kifejezésminták használatával, de bevethetünk akár Markov-láncokra épülő egyszerűbb nyelvi modelleket is[58])
- **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/karakter sort kell visszaadni, ami könnyen ellátható random bájtok/karakterek generálásával (legegyszerűbb közvetlenül a `TreeRandom` által szolgáltatott biteket használni)

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 a függő értékeket becsoportosíthatók 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 értéklístát indextartó módon kell átképezni a másik táblára. Ez a fentiek tükrében már egyszerűen biztosítható.

Egyes esetekben (néhány 1:1 kapcsolatnál) valójában egyetlen tábla kerül szétbontásra több táblára. Ekkor a több táblára bontottság csak külsődleges, és az előző részben az oszlopok közötti összefüggésekről írottak alkalmazhatók, még akkor is, ha a két tábla számossága nem egyezik teljesen (a hiányzó sorok a *NULL* értékek kiosztásához hasonlóan kerülhetnek kiválasztásra).

A legbonyolultabb esetben 1:n kapcsolat mellett statisztikai vagy egyéb követelményeket szeretnénk biztosítani indexeltség mellett. Ez messzire vezet, csak egy rövid megjegyzéssel érzéltetném, 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

Empirikus tapasztalatok, összehasonlítás

4.1. Mérési környezet

Az ebben a fejezetben leírt mérések nem JMH alapokon történtek, hiszen főként end-to-end vagy integrált jellegűek, illetve több helyen más adatbázisokkal vettem össze az eredményeket. Minden mérést ugyanazon a Debian 12 GNU/Linux rendszerű gépen futtattam (az egyéb paramétereket lásd az 1.7 alfejezetet). A táblázatoknál a konkrét mérési módszert a táblázat alatti leírásban írtam le. Az összehasonlításoknál a következő szoftververziókat használtam (az alapértelmezett kiegészítő eszközökkel együtt értve):

	Standalone	Docker
HoloDB	0.5.1-SNAPSHOT	miniconnect/holodb:0.5.1-SNAPSHOT
H2	2.2.224	mertcingoz/h2:2.3.232
MariaDB	10.11.6 (MySQL 15.1)	mariadb:10.11.6-jammy (official)
PostgreSQL	15.10	N/A
Laravel	11.36.1	N/A

4.1. táblázat. Az összehasonlító méréseknél használt egyéb rendszerek verziói

4.2. Általános futtatási tapasztalatok

Felhasználói oldalról a HoloDB használata főként három dolgot jelent: a konfigurációt, a futtatást, illetve a tesztadatbázishoz való csatlakozást. A konfigurációról már volt szó, a következőkben a futtatásról és csatlakozásról lesz szó.

A HoloDB szerver használható Docker konténerként vagy önálló alkalmazásként.

	Docker image (MiB)	Shadow JAR (MiB)
HoloDB	237,3	23,09
HoloDB-opt	53,6	14,37
H2	45,4	2,49
MariaDB	116,7	N/A

4.2. táblázat. JAR fájl és Docker-kép mérete összehasonlításban. A **HoloDB-opt** sor egy *kísérleti* disztribúciót mutat, ahol a JAR fájlban a Proguard eszközzel kirostáltam a függőségek nem használt részeit, illetve egy minimális alpine-alapú Docker-alapképet használtam minimális Java-telepítéssel.

A dockeres indítás akár egyetlen paranccsal is történhet, például:

```
1 docker run --rm -v sample-config.yaml:/app/config.yaml -p 3430:3430 -it
   ↪ miniconnect/holodb:latest
```

A `sample-config.yaml` helyén megadhatjuk a saját konfigurációs fájlunkat, az kettőspont bal oldalán lévő 3430 helyett pedig egy egyedi portot, amelyen az adatbázist el kívánjuk érni.

Önálló alkalmazásként forráskódból a megfelelő `gradle run` taszkkal vagy a standalone JAR fájl indításával tudjuk futtatni.

A szerver indításához körülbelül minimum 8MB heap-méretre van szükségünk, de a használat körülbelül 10MB-tól mondható biztonságosnak. Ha nem töltünk be óriás értékkészleteket és egyedi működések, akkor egyetlen munkamenettel csatlakozva ez a 10MB már elégséges bonyolult lekérdezések futtatásához is akár óriás adatbázison (például a `-Xms5M -Xmx10M -XX:+UseG1GC` JVM kapcsolókkal).

Javás programkörnyezetből a MiniConnect API-n keresztül vagy pedig JDBC-vel tudunk csatlakozni az adatbázishoz. Java projekt esetén beágyazott adatbázis is használható, ami akár JPA entitásokra helyezett annotációkkal is indítható. (A részleteket lásd a `holodb project README.md` fájljában, illetve a példaprojektekben.)

Interaktívan csatlakozni a `micl` klienssel lehet (telepítéshez lásd a `miniconnect-client project README.md` fájlját). Ez a REPL kliens azonnal csatlakozik a MiniConnect API-val kompatibilis adatbázishoz a megadott hoszton és porton (alapértelmezetten `localhost:3430`). Gépeléskor szintaxiskiemelést, automatikus kiegészítést és parancstörténetet kapunk, az eredménytáblák pedig szépen formázottan jelennek meg (ugyanakkor egy fapados ASCII üzemmód is elérhető).

```
Welcome in miniConnect SQL REPL! - localhost:3430

SQL > USE economy;

Query was successfully executed!

SQL > SELECT * FROM employees WHERE id=1;

Query was successfully executed!
```

id	company_id	firstname	lastname	birth_year	birth_country
1	5	Jennifer	Morris	1953	Laos

```
SQL > exit

Bye-bye!
```

4.1. ábra. A MiniConnect REPL használat közben HoloDB szerverrel

4.3. Lekérdezések sebessége

A lekérdezéseket egy nem túl nagy adatbázison futtattam, melyben három tábla található: `subjects` (3 oszlop, 3 rekord), `courses` (4 oszlop, 30 rekord), `students` (4 oszlop, 10 000 rekord) és `course_student` (kapcsolótábla, 3000 rekord). (A konkrét konfigurációhoz lásd: Függelék D)

A lekérdezések közvetlen mérésénél az SQL parancsnak a JDBC API-n keresztüli beküldésétől az eredménytábla végigiterálásnak végéig eltelt időt mértem egyszerűen `System.nanoTime()` hívások visszaadott értékeinek különbségeit nézve. Öt különböző bonyolultságú és válaszméretű fix lekérést mértem, alapesetben 10 000-szerfuttatva, a teljes tábla kivételével, ahol 100-szor.

Az első tesztben (Q-FLD) egyetlen mező lekérését nézem. Egy adott táblában keresem az `id` oszlop egy adott értékének pozícióját, majd egy másik oszlop értékét a talált pozíción. A konkrét SQL:

```
1 SELECT firstname FROM students WHERE id = 30
```

A második teszt (Q-REC) hasonló, de az összes oszlopból keresem ki az értékeket, azaz egy teljes rekordot kérek le:

```
1 SELECT * FROM students WHERE id = 30
```

A harmadik tesztben (Q-CMPX) a kulcsmezőtől eltérő mezőben keresek, ami így több találatot eredményez. A találatok számát 20-ra limitálom:

```
1 SELECT s.lastname FROM students s WHERE s.firstname='Ian' LIMIT 20
```

A negyedik tesztben (Q-TFUL) lekérem a teljes táblát:

```
1 SELECT * FROM students
```

Az ötödik tesztben (Q-MULT) több táblából gyűjtöm az adatokat, melyeket a kulcsmezőkkel fűztem össze (LEFT JOIN). Egyúttal dupla keresési feltételt is megadok két különböző táblára:

```
1 SELECT s.lastname FROM students s
2     LEFT JOIN course_student cs ON cs.student_id = s.id
3     LEFT JOIN courses c ON c.id = cs.course_id
4     LEFT JOIN subjects su ON su.id = c.subject_id
5     WHERE s.firstname='Ian' AND su.id=2
```

A lekérdezéseket a beágyazott HoloDB három különböző skálázási szintjén mértem. A MIN esetben mindenből a leggyorsabb, legfapadosabb implementációt használtam (például egyszerű lineáris extrapoláció). A DEF esetben mindent az alapértelmezett beállításokon hagytam. Az EXT esetben pedig a legjobb kimeneti minőségű implementációkat állítottam be, ami természetesen a performancia rovására megy, különösen a teljes tábla lekérésekor, mivel az érték kiszámítását minden egyes visszaadott mezőre el kell végezni¹. Ugyanakkor kevés adatnál a teljesítmény meglepően jó, vagyis a korábban tárgyalt indexelési technikák jól teljesítenek.

Az összehasonlítást beágyazott H2-vel és (nem dockeres) MariaDB szerverrel végeztem, ezekhez az EXT eset adatbázistartalmát materializáltam le.

A 4.3 táblázat mutatja a futási időket mikromásodpercben.

¹Ez bőven optimalizálható még. Jelenleg például ugyanannak az értékkészlet-pozíciónak, sőt ugyanannak a mezőnek az értékét is minden hozzáférés alkalmával kiszámítjuk. A lekérdezésfuttató később továbbfejleszthető úgy, hogy az eredménytábla közös értékre vagy mezőre mutató hozzáférési pontjai közös lazy-computing referenciát használjanak.

	Q-FLD (μ s)	Q-REC (μ s)	Q-CMPX (μ s)	Q-TFUL (μ s)	Q-MULT (μ s)
HoloDB/EXT	43,00	180,35	1 981,98	1 514 177	6 818
HoloDB/DEF	24,66	130,08	225,17	1 143 721	3 592
HoloDB/MIN	22,82	20,22	44,58	25 909	365
H2	12,77	9,73	17,60	1 965	177
MariaDB	47,87	46,27	73,02	3 393	1 429

4.3. táblázat. Lekérdezési tesztek eredménye különböző adatbázisokkal.

A sorok jelentése (pontos verziók fentebb a főszövegben): **HoloDB/EXT**: beágyazott HoloDB legjobb minőségű paraméterekkel; **HoloDB/DEF**: beágyazott HoloDB alapértelmezett paraméterekkel; **HoloDB/MIN**: beágyazott HoloDB minimális paraméterekkel; **H2**: beágyazott H2; **MariaDB**: MySQL/MariaDB szerver. **A sorok jelentése**: Lásd a főszöveget.

4.4. Integrált tesztek eredménye

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.

A mérési apparátus egy szerver és egy kliens alkalmazásból áll. A szerver tartja a kapcsolatot az adatbázissal JDBC-n keresztül, és REST API-t biztosít hozzá. A szintén Micronaut keretrendszerben írt kliens ezt a REST API-t hívogatja tömegesen.

Pontosan ugyanazt az induló adatbázistartalmat használtam, mint az előző fejezet lekérdezési tesztjeinél. Az adatgenerálás maga nem része a mérésnek.

Az alkalmazások futtatását egy POSIX shell szkript vezérli. Egy további Bash szkript az előbbi fájlt hívja ciklusban, közben átírja a konfigurációs fájlt az egyes tesztekhez igazítva. A bash szkript a futtatási ciklusok között legalább 5 másodperc szünetet vár. (A teljes benchmarkért lásd: Függelék D).

A **RO-SIMP** tesztben csak egyszerű entitás-adatlekérdezési összesen (10 203 hívás történik). HoloDB esetén minden tábla csak-olvasható módban van (vagyis a `TableDiff` dekorátor nélkül).

A **RO-CMPX** teszt is csak-olvasható. A **RO-SIMP** hívásain túl egy külön végpont is meghívásra kerül, mely néhány egyszerű statisztikai lekérdezést hajt végre.

A WR-CMPX tesztben írási műveletek is végrehajtásra kerülnek. A RO-SIMP hívásain túl 50 entitás-módosítási kérés is megy a REST API-ra. HoloDB esetén ekkor minden tábla írható módban van (vagyis a TableDiff dekorátort használja).

A futási idők a 4.4 táblázatban láthatók.

	RO-SIMP (s)	RO-CMPX (s)	WR-CMPX (s)
HoloDB/EXT	6,79	7,10	12,16
HoloDB/DEF	6,36	6,56	11,09
HoloDB/MIN	4,60	4,76	7,54
H2	3,92	4,02	6,07
MariaDB	5,10	5,43	8,38

4.4. táblázat. Integrált tesztek eredménye különböző adatbázisokkal.

A sorok jelentése: Lásd a 4.3 táblázatot.

Az oszlopok jelentése: Lásd a főszöveget.

4.5. Adatgenerálás, indítás és tárigény

A HoloDB esetében nincs szükség adatgenerálásra, ám konfigurációjának méretét, indítási idejét² és tárhelyigényét összemérhetjük más adatbázisokéval. Természetesen ezek azok az aspektusok, ahol a HoloDB diadalmenetét várjuk, ez adja az egész projekt értelmét.

Három táblát használtam: companies (4 oszlop), employees (7 oszlop), coupons (3 oszlop). Az adatokat két méretbeli változatban generáltam: az első, kisméretű esetben 200, 2 000, illetve 5 000; az második, nagyméretű esetben pedig 2 000, 1 000 000, illetve 10 000 000 rekordszámmal. A HoloDB alapértelmezett beállításait használtam.

Úgy gondoltam, legcélszerűbb, ha egy nagyon gyors, natív megoldást, illetve egy népszerű keretrendszer megoldását veszem bele az összehasonlításba.

Gyors generáló környezetként a PostgreSQL-re esett a választásom, egyszerűen azért, mert performancia és testreszabhatóság tekintetében kiemelkedik a nyílt forrású adatbázis-szerverek közül. A generálást egy SQL-szkripttel végeztem. Az egyes speciális értékek előállításához (nevek, városok stb.) bevett extensionöket is igénybe vettem (postgresql_anonymizer[59], postgresql_faker[60]).

²Jelenleg indításkor a HoloDB konfigurációjában megadott összes oszlop szükséges adatstruktúrái legenerálódnak. Ez természetesen erősen optimalizálható. Ha ezeket a generálások csak később, on-demand jelleggel futnának, az indítási időt szinte a JVM indulási idejére lehetne redukálni. Fejlesztői gépeken még ezt is ki lehetne küszöbölni egy daemon futtatásával.

Népszerű megoldásként a PHP nyelven írt Laravel keretrendszert választottam[61]. A Laravel beépített megoldást tartalmaz az adatmodell leírásához és tesztadatok generálásához. A generáláshoz a beépített artisan szkript által biztosított standard migráló eszközt használtam. Sajnos a nagyméretű esetben a Laravel megoldása órákig való futás után sem végzett, és egyre lassabban futott; így kénytelen voltam kerülőmegoldásokat beépíteni, azaz a generálás egy részét közvetlen SQL-lekérdezésekkel futtatni (még így is rengeteg időt vett igénybe, maga a MySQL/MariaDB is gyengébb teljesítményt nyújt, mint a PostgreSQL).

Az eredményeket 10 futtatás átlagából számoltam. Kivétel a MariaDB+Laravel a nagyméretű esetben, a jelentős időigény miatt ezt csak 3-szor futtattam le, és a leggyorsabb esetet vettem.

(A teljes generáló környezetért lásd: Függelék D).

A kisméretű adathalmaz esetében a következő eredmények születtek:

	Generátor	Generálás (s)	Indítás (s)	Konfiguráció (B)	Tárigény (MiB)
HoloDB	N/A	0,00	0,28	1 303	0.16
MariaDB	Laravel seeder	7,08	0,28	7 944	0,45
Postgres	SQL-szkript	0,20	2,25	2 005	1,40

4.5. táblázat. Kisméretű adathalmaz (7 200 rekord) generálása különböző megoldásokkal.

A nagyméretű adathalmaz esetében pedig az eredmények a következők:

	Generátor	Generálás (s)	Indítás (s)	Konfiguráció (B)	Tárigény (MiB)
HoloDB	N/A	0	0,28	1 311	0,16
MariaDB	Laravel seeder	15 734	0,30	8 008	1 101,47
Postgres	SQL-szkript	95	2,26	2 017	1 409,57

4.6. táblázat. Nagyméretű adathalmaz (11 002 000 rekord) generálása különböző megoldásokkal.

4.6. Terheléses teszt

Több szálon futó klienssel teszteltem, hogyan reagál a HoloDB szerver tömegesen beérkező változó lekérdezésekre. A konfiguráció ugyanaz volt, mint az előbbi nagyméretű generálás esetében.

A szervert önálló alkalmazásként futtattam a `-Xms8M -Xmx50M -XX:+UseG1GC` JVM argumentumokkal. A kliens egy egyszerű Java program volt, mely 45 szálon 10 másodpercig

küldött kéréseket. Minden szál 0,2 másodperc szünetet tartott két kérés között. A szálak háromféle SQL-t küldtek (mindegyiket 15 szál):

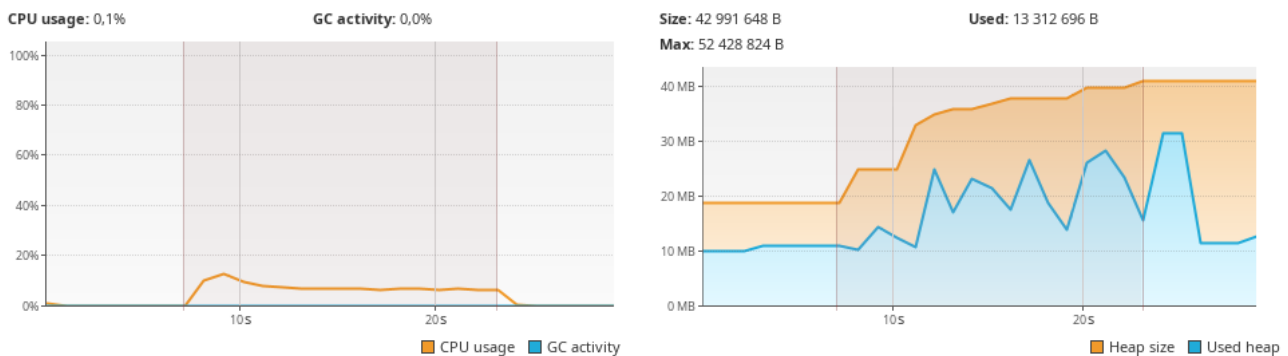
```

1 SELECT * FROM employees WHERE id = ?
2 SELECT e.*, c.identifier FROM coupons c LEFT JOIN employees e ON
   ↪ c.employee_id=e.id WHERE c.id = ?
3 SELECT * FROM employees ORDER BY lastname ASC, firstname DESC LIMIT 20

```

A kérdőjelek helyére kérésenként véletlenszámok kerültek behelyettesítésre.

A szerver a kéréseket jól fogadta, bár a hálózati átvitel egyelőre meglehetősen lassú. Egy-egy kérés end-to-end feldolgozási ideje az első két SQL esetében kb. 0,25 másodperc volt, a harmadik kérés esetében akár 4-5 másodpercre is megnyúlt. A hálózati átvitelen tehát még bőven van fejleszteni való.



4.2. ábra. Terheléses teszt eredménye. Az árnyékolt terület mutatja a kliens futási ablakát.
(A grafikon a VisualVM eszközről mentett képek alapján készült.)

5. fejezet

Speciális használati példák

5.1. 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.

5.2. 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:

```
1 SELECT email FROM customers WHERE vip=1 LIMIT 10;  
2 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 egy 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 ecélből 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.

Kézenfekvőnek 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 részletes 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 első nevezhetjük *simple mode*-nak. Ekkor pusztán a tábla és az oszlopok neveiből indulunk ki (valamint a 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 könnyen felismerhető). 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, ha egy új lekérdezés nagyon eltérő oszlopjellemzőket sugall. A olyan globális jellemzőket, mint például a táblaméret, ekkor sem módosítjuk.

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

5.3. Valós cím adatok 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 scenáriót vizsgálok meg: valamely tábla rekordjaiban olyan magyarországi közigazgatási címeket szeretnénk vizsgálni, melyek valósak. Amennyire lehetséges, indexeket is szeretnénk biztosítani.

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

településnév → kerület → közterület neve → közterület jellege → házszám → 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 cím adatbázist 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.

5.4. 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 függvényvel. A kapcsolt adat valamely egyéni értéksávjára való kereséskor lekérjük a sávon teljesen belülrre eső, illetve a kilógó kontextussávokat, lekérjük az ezekhez tartozó alakzatokat, és ezen alakzatokra keressük a földrajzi koordinátákat. A keresett sávon teljesen belülrre 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.

6. fejezet

Összegzés

6.1. 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 tárhelyhasználatban 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 az adatbázisos füstteszt, a teljesen izolált automatikus integrációs tesztek, vagy a dev mode.

6.2. A célkitűzések teljesülése

1. **Támogatja a relációs adatmodellt.** Valóban; a relációs adatmodell entitásait a relációs storage API-val definiáltuk, a bemutatott megoldás pedig megvalósítja ezt az API-t és támogatja az SQL lekérdezések futtatását.
2. **Determinisztikus, koherens.** Valóban; a TreeRandom mint hierarchikus véletlengenerátor biztosítja a determinisztikusságot, az értékkiosztások pedig a koherenciát adatelérés és index,

illetve több lekérdezés között.

3. **Az adatok könnyen variálhatóak.** Valóban; minden számítás véletlenfaktora a fő TreeRandom példányból származik. Ez a konfigurációban elhelyezett root seed alapján példányosodik, így módosított root seed esetén egy egészében megváltozott adatbázist kapunk.
4. **Óriás adatmennyiséget is képes szimulálni.** Valóban; mivel az adatok nincsenek tárolva, a mennyiség önmagában nem jelent problémát. Az értékkiosztások, SQL-futtatók és más számítások pedig a nagy számokat is támogató beépített aritmetikát használják. Láttuk, hogy a HoloDB a gyakorlatban is jól kezeli az óriás táblákat.
5. **Gyors, indexelt keresést nyújt, a lekérések időigénye elfogadható.** Valóban; az értékkiosztásokban alkalmazott megoldások lehetővé teszik a virtuális indexeket. Az óriás számokat is támogató beépített aritmetika kis számokon nagyon hatékony. Végeredményben, az empirikus tesztek alapján a lekérdezések sebessége elfogadható.
6. **Azonnal elindul, nincs számottevő preparálási folyamat.** Valóban; alapértelmezetten pusztán a konfiguráció betöltése történik meg az indításkor. Ez esetenként bizonyos adatstruktúrák legenerálását is jelenti, de láttuk, hogy az indítási idő ekkor is rövid.
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, ezek azonban elenyészőek a szimulált adatmennyiséghez képest.
8. **Könnyen és rugalmasan konfigurálható.** Valóban; a deklaratív YAML konfigurációs fájl kényelmesen szerkeszthető és verziókezelhető, minden fontosabb paraméter beállítására alkalmas. Emellett JPA entitások annotálásán keresztüli beállítás is lehetséges. Megadható a teljes séma, a táblajellemzők, az oszlopokban használt értékkészletek és adateloszlási paraméterek, bizonyos megkötések, skálázási paraméterek és sok más.
9. **Skálázható, finomhangolható.** Valóban; lényegileg minden komponens skálázható, illetve akár egyenként cserélhető. 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 konfigurációból is hivatkozható. Emellett egy programozható API is rendelkezésre áll.
11. **Opcionálisan írható.** Valóban; a Table interfész támogatja a módosító műveleteket, melyeket a DiffTable dekorációval valósít meg, csak a módosítások számontartásával. Az SQL-futtató réteg egy egyszerű tranzakciókezelést és módosító SQL-eket is támogat.

6.3. 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 legalábbis pozitív fogadtatásra, de akár nagy népszerűsége számíthat.

A további munkát több frontvonalra tudom bontani:

- az alapvető működés optimalizálása
- további kiegészítő funkciók, eszközök fejlesztése
- dokumentáció
- kommunikáció és terepmunka

Az értékkiosztásokat és lekérdezéseket sok apró alkatrész teljesítményének finomhangolásával, elsősorban algoritmuselméleti kutatással lehet továbbfejleszteni. Ugyanilyen fontos viszont a programkörnyezet hatékonyabbá tétele is, különösen a Docker-képé. Optimalizáláson nem csak a performancia tuningolását, de az architektúrális célszerűség növelését és a felhasználói élmény javítását is értem. Ebbe a logikus, könnyen érthető, lényegre törő konfiguráció éppúgy beletartozik, mint a kiegészítő eszközök ergonómiája. Kiemelendő még a hálózati átvitel átviteli sebességének javítása, ami minden alkalmazást érint, amely nem beágyazottan futtatja a tesztadatbázist.

Az új eszközök fejlesztése elsősorban a különféle meglévő technológiákkal történő integrációt jelenti (tervezőeszközök, adatbázis-böngészők, programozási környezetek stb.). Másodszorban pedig a használat megkönnyítését szeretném megcélózni (további REPL-funkciók, generálás/materializálás, 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 részletesebb tutorial stílusú leírásokra is szükség lesz.

A valós gyakorlattal való összhangba kerülés érdekében mind az akadémiai, mind a versenyszféra-közegben szükséges lesz a visszajelzések intenzívebb gyűjtése. A HoloDB által megoldani kívánt probléma természetét, illetve a felmerülő elméleti és gyakorlati igényeket kérdőíves formában és aktív terepmunka keretében is alaposabban fel kell mérni, melybe fejlesztőket, technikai vezetőket és oktatókat egyaránt érdemes lenne bevonni.

Hivatkozások

- [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] Amid Golmohammadi, Man Zhang és Andrea Arcuri. „Testing RESTful APIs: A Survey”. *ACM Trans. Softw. Eng. Methodol.* 33.1 (2023. nov.). ISSN: 1049-331X. DOI: 10.1145/3617175. URL: <https://doi.org/10.1145/3617175>.
- [8] Hesam Samimi és tsai. „Declarative mocking”. *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ISSTA 2013. Lugano, Switzerland: Association for Computing Machinery, 2013, 246–256. ISBN: 9781450321594. DOI: 10.1145/2483760.2483790. URL: <https://doi.org/10.1145/2483760.2483790>.
- [9] Abhijit Kane Abhinav Asthana Ankit Sobti. *Postman/Newman*. v6.2.1. verzió. Accessed: 2024-12-01. URL: <https://github.com/postmanlabs/newman>.
- [10] Tom Akehurst. *WireMock*. v3.10.0. verzió. Accessed: 2024-12-01. URL: <https://github.com/wiremock/wiremock>.
- [11] Stoplight.io. *Stoplight/Prism*. v5.12.1. verzió. Accessed: 2024-12-01. URL: <https://github.com/stoplightio/prism>.
- [12] SmartBear Software. *SoapUI*. v5.7.2. verzió. Accessed: 2024-12-01. URL: <https://github.com/SmartBear/soapui>.
- [13] Gediminas Morkevicius. *Go SQLMock*. v1.5.2. verzió. Accessed: 2024-12-01. URL: <https://github.com/DATA-DOG/go-sqlmock>.
- [14] Benjamin Curtis. *Ruby Faker*. v3.5.1. verzió. Accessed: 2024-12-01. URL: <https://github.com/faker-ruby/faker>.
- [15] DiUS Computing. *Java Faker*. v1.0.2. verzió. Accessed: 2024-12-01. URL: <https://github.com/DiUS/java-faker>.
- [16] Gillian Smith. „An Analog History of Procedural Content Generation.” *FDG*. Boston, MA. 2015, 1–6. old.
- [17] Ken Perlin. „An image synthesizer”. *ACM Siggraph Computer Graphics* 19.3 (1985), 287–296. old.

- [18] Ian Parberry. „Designer worlds: Procedural generation of infinite terrain from real-world elevation data”. *Journal of Computer Graphics Techniques* 3.1 (2014).
- [19] Lars Krecklau és Leif Kobbelt. „Procedural modeling of interconnected structures”. *Computer Graphics Forum*. 30. köt. 2. Wiley Online Library. 2011, 335–344. old.
- [20] *Elite*. URL: <http://www.iancgbell.clara.net/elite/> (elérés dátuma 2024. 12. 01.).
- [21] Ian Bell és David Braben. *Elite, 1984*. URL: <https://elite.bbcelite.com/cassette/>.
- [22] Jimmy Maher. *Starflight*. 2014. URL: <https://www.filfre.net/2014/10/starflight/> (elérés dátuma 2014. 10. 28.).
- [23] *Minecraft Seed (level generation)*. URL: [https://minecraft.fandom.com/wiki/Seed_\(level_generation\)](https://minecraft.fandom.com/wiki/Seed_(level_generation)) (elérés dátuma 2024. 12. 01.).
- [24] *StarMade*. URL: <https://www.star-made.org/> (elérés dátuma 2024. 12. 01.).
- [25] 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.).
- [26] 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.).
- [27] Chongyang Gao és tsai. „FakeDB: Generating Fake Synthetic Databases”. *IEEE Transactions on Dependable and Secure Computing* 21.6 (2024), 5553–5564. old. DOI: 10.1109/TDSC.2024.3379434.
- [28] RapidDweller. *DATAMIMIC*. v1.1.0. verzió. Accessed: 2024-12-17. URL: <https://github.com/rapiddweller/datamimic>.
- [29] Tirthajyoti Sarkar. *pydbgen*. v1.0.0. verzió. Accessed: 2024-12-01. URL: <https://github.com/tirthajyoti/pydbgen>.
- [30] Fabian Prasser Florian Kohlmayer. *ARX*. v3.9.1. verzió. Accessed: 2024-12-01. URL: <https://github.com/arx-deidentifier/arx>.
- [31] Matteo Giomi és Mikhail D. *Anonymeter*. v1.0.0. verzió. Accessed: 2024-12-01. URL: <https://github.com/statice/anonymeter>.
- [32] 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>.
- [33] *MOSTLY AI*. Accessed: 2024-12-01. URL: <https://github.com/mostly-ai/mostlyai>.
- [34] *Tonic*. URL: <https://www.tonic.ai/> (elérés dátuma 2024. 12. 01.).
- [35] *GenRocket*. URL: <https://www.genrocket.com/> (elérés dátuma 2024. 12. 01.).
- [36] *Delphix*. URL: <https://www.delphix.com/> (elérés dátuma 2024. 12. 01.).
- [37] 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.
- [38] Edmon Begoli és tsai. „Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources”. *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. Houston, TX, USA: Association for Computing Machinery, 2018, 221–230. ISBN: 9781450347037. DOI: 10.1145/3183713.3190662. URL: <https://doi.org/10.1145/3183713.3190662>.
- [39] Julian Hyde. *Apache Calcite*. 1.38.0. verzió. Accessed: 2024-12-01. URL: <https://github.com/apache/calcite>.
- [40] Oracle Corporation. *Multiple-Column Indexes (MySQL)*. URL: <https://dev.mysql.com/doc/refman/8.4/en/multiple-column-indexes.html> (elérés dátuma 2024. 12. 01.).
- [41] FasterXML. *Jackson Databind*. v2.18.2. verzió. Accessed: 2024-12-01. URL: <https://github.com/FasterXML/jackson-databind/tags>.

- [42] Oracle Corporation. *Java Persistence API, Version 2.2*. Final Release Specification. 2017. URL: https://download.oracle.com/otn-pub/jcp/persistence-2_2-mrel-spec/JavaPersistence.pdf (elérés dátuma 2025. 01. 01.).
- [43] Mohammed Erritali és tsai. „An Approach of Semantic Similarity Measure between Documents Based on Big Data”. *International Journal of Electrical and Computer Engineering* 6 (2016), 2454–2461. old. URL: <https://api.semanticscholar.org/CorpusID:54714656>.
- [44] 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>.
- [45] Erik Osheim. *Typelevel Spire*. 0.18.0. verzió. Accessed: 2024-12-01. URL: <https://github.com/typelevel/spire>.
- [46] Apache Commons. *Apache Commons Math*. v4.0-beta1 verzió. Accessed: 2024-12-01. URL: <https://github.com/apache/commons-math>.
- [47] 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.
- [48] Mihir Bellare és tsai. „Format-Preserving Encryption”. *IACR Cryptol. ePrint Arch.* 2009 (2009), 251. old. URL: <https://api.semanticscholar.org/CorpusID:1081553>.
- [49] Jack Lloyd. *FPE for Java*. URL: https://botan.randombit.net/handbook/api_ref/fpe.html (elérés dátuma 2024. 12. 01.).
- [50] Rob Shepherd. *FPE for Java*. URL: <https://github.com/robshep/JavaFPE> (elérés dátuma 2024. 12. 01.).
- [51] Colm O’Connor. *Xeger*. v0.3.5. verzió. Accessed: 2024-12-01. URL: <https://github.com/crdoconnor/xeger>.
- [52] Youssef Mifrah. *Generex*. v1.0.2. verzió. Accessed: 2024-12-01. URL: <https://github.com/mifmif/Generex>.
- [53] 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>.
- [54] 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/>.
- [55] *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>.
- [56] ICU Project. *ICU - International Components for Unicode*. Accessed: 2024-12-01. 2024. URL: <https://icu.unicode.org/>.
- [57] 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>.
- [58] @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.).
- [59] Damien Clochard. *PostgreSQL Anonymizer*. v1.1.0. verzió. Accessed: 2024-12-01. URL: https://github.com/neondatabase/postgresql_anonymizer.
- [60] Damien Clochard. *DALIBO PostgreSQL Faker*. v0.5.3. verzió. Accessed: 2024-12-01. URL: https://gitlab.com/dalibo/postgresql_faker.
- [61] Taylor Otwell. *The Laravel Framework*. v11.36.1. verzió. Accessed: 2024-12-01. URL: <https://github.com/laravel/framework>.

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_EXTERNALHELPER(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 ← CONVERTRAWTRIE TO COMPACT(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

Táblázatjegyzék

Táblázatok jegyzéke

3.1. Óriás egész szám implementációk jellemzői	22
3.2. Permutáció-implementációk jellemzői	28
4.1. Összehasonlító mérések verziói	46
4.2. JAR fájl és Docker-kép mérete	47
4.3. Lekérdezési tesztek eredménye	50
4.4. Integrált tesztek eredménye	51
4.5. Kisméretű generálás összehasonlítása	52
4.6. Nagyméretű generálás összehasonlítása	52

C. függelék

Ábrajegyzék

Ábrák jegyzéke

2.1. A virtuális adatbázis vázlatos architektúrája	12
2.2. A storage API fő interfészei (egy relációs adattár alapentitásai)	13
2.3. Egy minimalisztikus konfiguráció és eredménye	16
3.1. Alapvető segédinterfészek	19
3.2. Permutációk kimeneteinek összehasonlítása	28
3.3. A kétlépéses értékkiosztás alapelve	30
3.4. Adatlekérés a kétlépéses értékkiosztásból	30
3.5. Keresés a kétlépéses értékkiosztásban	30
3.6. Egyszerű reguláris kifejezés nyers szófája	34
3.7. Egyszerű reguláris kifejezés kompakt szófája	35
3.8. Többszintű kompakt szófa viselkedése	39
3.9. Full-text szóösszeállítás	41
3.10. Full-text keresés	41
3.11. Geolokációk keresése alakzat alapján	43
4.1. A MiniConnect REPL használat közben HoloDB szerverrel	48
4.2. Terheléses teszt eredménye	53

D. 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>