Eötvös Loránd Tudományegyetem

Informatikai Kar

Információs Rendszerek Tanszék

Bioinformatikai adatbázisok

dr. habil. Kiss Attila Elemér
Tanszékvezető, habilitált docens

Lehotay-Kéry Péter
Programtervező Informatikus MSc

Budapest, 2018

Tartalomjegyzék

1 Genetikai alapok

Deoxyribonucleic acid (DNA) egy genetikai információt hordozó komplex molekula. Ezeket nukleotidok építik fel. Minden nukleotid 3 komponensből áll:

- nucleobázisok (adenin - A, guanin - G, citozin - C, timin - T),
- deoxiribóz
- foszforsav

A bioinformatikában, a DNS szekvenciákat karakterkláncokként tároljuk, melyeket a 4 nukleobázishoz tartozó 4 karakterből komponáljuk: 'A', 'C', 'G' and 'T'. Hasonlóan tárolhatjuk az RNA és fehérjeszekvenciákat is, RNS esetében 'A', 'C', 'G', 'U', fehérje esetében 'A', 'R', 'N', 'D', 'C', 'E', 'Q', 'G', 'H', 'I', 'L', 'K', 'M', 'F', 'P', 'S', 'T', 'W', 'Y', 'V' lesznek a karakterláncok lehetséges elemei.

Minden ember génállománya körülbelül 3 milliárd bázispárból áll, de általában csak azoknak az analízise érdekel minket, melyek különböznek az egyes egyénekben. Ezek a különböző szekvenciák a variánsok és kb 100 millió létezése ismert.

Egy egyénhez tartozó szekvenciából kinyerve a variánst kapjuk a mintát. Egy egyén genotípusa meghatározható például úgy, ha szekvenciába állítjuk, majd lehívjuk a variánsait a teljes génállományának. Másik módszer, ha genotípizáljuk (egyed szekvenciájának összevetése referenciaszekvenciával) a választott variánsokat microarray segítségével.

Allele a bázispár, vagy bázispárok szekvenciája, melyet az egyén bitrokol az adott variánson. A legtöbb variánshelyen, az emberek két allelét hordoznak, 1-et 1-et az egyes homológ kromoszómákon (apai és anyai).

2 Motiváció és Kapcsolódó munkák

Születtek már munkák bioinformatikai adatbázisok témakörében.

[1] egy hatékony sémát prezentál PostgreSQL-ben és összehasonlítja más sémákkal és a file alapú tárolással.

[2] méréseket mutat a PostgreSQL hatékonyságáról Cassandra NOSQL adatbázissal összehasonlítva.

[3] sémákat és lekérdezéseket prezentál genetikai adatok feldolgozására MySQL-ben.

[4] és [5] genetikai adatok tárolását vizsgálja RDF és SPARQL segítségével.

Előbbiben az EBI által fejlesztett RDF platform használatával a felhasználók több összekapcsolt erőforrás felhasználásával tehetnek fel kérdéseket. Ezen erőforrások közös azonosítóval és a közös RDF formátummal és SPARQL lekérdező interfésszel rendelkeznek.

Utóbbiban 2 fő ötlet jelenik meg: létező adatbázisok konverziója RDF formátumba és a létező szemantikus web szoftverek használata összefuttatásra, lekérdezésre és adatvizualizációra.

De ezek egyike se mond semmit ezen érzékeny adatok védelméről, vagy a felhasználói hierarchiákról.

Tehát a fő cél tárolt eljárásokból álló könyvtárak építése volt PostgreSQL-ben, Oracle DB-ben és MS SQL-ben, melyek lehetőséget adnak genomok titkosítására és biztonságos, titkosított feldolgozására, valamint felhasználói hierarchiák használatára.

Áttekintjük továbbá a tömörítési lehetőségeket is a vizsgált adatbáziskezelő rendszerekben, mivel fontos szempont mekkora helyet igényelnek ezek a 3 milliárd karakter hosszú, fájlként 3 GB-on tárolt adatok.

Nyilván veszítünk némi időt a titkosítással és feloldásával, így hatékonysági méréseket végeztem, hogy még mindig ésszerű időben fussanak az eljárások, és ellenőrizve mennyibe kerül a védelem. Tulajdonképpen ezen vizsgálatok elvégzésének segítése lenne a dolgozat kapcsán az elkészült könyvtár fő célja.

Végül cél volt feltérképezni a lehetőségeket, hogy mindezekhez milyen eszközöket kínálnak az említett adatbáziskezelők.

A mérésekre használt és tesztelt bioinformatikai algoritmusok a [6] könyvből származnak.

3 Genom tárolása fájlokban

[1] több speciális-célú file formátumról ír génadatok kezelésére. Ezen formátumok használata elterjedt tárolásra, lekérdezésre, analízisre.

- **gzip és shell scripting**: Gzip-tömörítésű flat fájlok tárolása és lekérdezése shell használatával egy általános szokás. Tömörítünk egy minták szerint sorokba rendezett, variánsok szerint oszlopokba rendezett, tabulátorokkal határolt flat fájl gzip -6 paranccsal. Ezután mintaillesztéseket végezhetünk head, tail, sed, cut és grep parancsokkal.

- **Tabix** eszközzel indexelhetők és lekérdezhetők tabulátorral határolt adatok. A céladatot először rendezni kell genom pozíció szerint és aztán tömöríteni bgzip eszközzel. Az eredmény fájl ezután indexelhető variáns lekérdezések támogatásához, a minták szerinti szűrés cut paranccsal történik. Több formátumot támogat, SQL exportokat is beleértve.

- **PLINK** többdéle statisztikai analízist képes elvégezni, formátum konverziókat, támogatja az alap minta és variáns lekérdezéseket. Minden hívás beolvassa az összes adatot a memóriába a lemezről, mielőtt a kért műveleteket elvégzi és fájlokba írja az eredményt.

4 Genom tárolása relációs adatbázisban

4.1 Adatbázis kategóriák

[7]Adatbázisok általában elsődleges és másodlagos kategóriákba sorolhatóak.

Elsődlegesben kísérletből származtatott adatokat tárolnak, mint nucleotide szekvencia, protein szekvencia, micromolecular struktúra. Ezek direkt kísérleti eredmények, melyeket kutatók töltenek fel.

Másodlagos adatbázisok ezzel ellentétben az elsődlegedes adatbázisok analizálásának eredményeképp nyerik az adataikat.

Bioinformatikai adatbázisok általában alábbi kategóriákba sorolhatóak:

- Nukleinsav szekvenciák és struktúrák
- Fehérjeszekvenciák és struktúrák
- Fehérje-fehérje interakciók
- Vírusok, baktériumok, gombák genomjai
- Emberi és model organizmusok genomjai
- Emberi betegségek és drogok
- Növények

	id integer	seq text
1	1	TTGAATGCTGAAATCAGCAGGTAATATATGATAATAGAGAAAGCTATCCCGAAGGTGC
2	2	TAACTCAGTAGATCCTAAAAGAAAGCAATTTTTGCTGCTAACCTAACATTTCACAATGTC
3	3	ACAACCTATGGCAGTTACTGGCATCTACTAGAGGTCAGAGATGCTGGTAAATACTCTGT
4	4	AGCAAATAAATACCCAGCCCAAGATGGCAATAGTGCCCAGATTGAGAAACTTCACCTTA
5	5	TCTGAAGAAAGACACAAACATAACTAAAGAAAGATGATTACCAGAAGAGATATTCATAA
6	6	AAACACAAGGCAATGTTTTCAGTGTCCAGGCAATTATCTTCCTGGGAAAAGCTAGCCTA
7	7	ACCTTGCTGGCAACCATTCTACTCTTCTGAAGAAGGAGACATCATTTGGACTCTAAAATC
8	8	CATCAAGAAATCTATCCATTTGGCTTAGTTTGTAGCTTATGCTGAAAAACGTGACTTGAC
9	9	GAGATTGCTTAATGTAGATTGACATTCTCAACATTTGGACAATAGTGGGATCAATTATCT
10	10	TATACCTCTGGGCAACAGTCAAATTACCAAGGTAAATGCTTAGTTGTAGTCAGCATGGC
11	11	CATTTTTTAAAGAGATATAGGGCTTTTCAGGTTCTCTTTTTCTTCTTGAGTGAGCTTAAG

Figure 1: Genom táblában tárolása

4.2 Metaadat

[7]Hogy hasznos legyen, az adatot kontextusba kell helyezni. Egy módja ennek, ha metaadattal asszociáljuk őket. Például, a BioSamples adatbázis metaadatot tartalmaz mintákról, melyeket az ENA, PRIDE és ArrazExpress adatainak generálására használtak. Metaadat tárolásával biztosítható, hogy adott minta konzisztensen azonosítható több adatforrásban.

4.3 Minimum információs sztenderd

[7]Minimum információs sztenderd irányvonalak és formátumok gyűjteménye annak biztosítására, hogy a generált adat könnyen analizálható, interpretálható legyen.

Tipikusan két részből állnak. Először is követelmények halmaza általában táblaként vagy listaként prezentálva. Másodszor az adat formátuma.

BioSharing.org minimum információs standardek listáját tartalmazza.

4.4 Tárolás PostgreSQL-ben

[8]PostgreSQL-ben alapvetően 3 féle típusban tárolhatjuk karakterláncainkat:

- varchar(n) n karaktert képes tárolni, mérete maximum n, de lehet kisebb is.
- char(n) n karaktert képes tárolni, elemei space-ekkel határolva, tehát mérete akkor is n, ha rövidebb karakterláncot töltünk bele.
- text bármilyen hosszú karakterláncot képes eltárolni, nem SQL sztenderd.

126 byte-ig a szöveg tárolási mérete 1 byte plusz a szöveg, char esetén beleértve a space-eket. Hosszabb karakterek tárolása 1 helyett 4 byte-ba kerül, ezeket azonban a rendszer automatikusan tömöríti. A leghosszabb karakterlánc körülbelül 1 GB lehet maximum.

Mi esetünkben a text lesz a legmegfelelőbb.

Genom-ot tároló tábla PostgreSQL-ben például:

```
CREATE TABLE genom (
id NUMBER,
    seq TEXT
)
```

4.5 Tárolás Oracle DB-ben

[9] Oracle DB-ben több lehetőségünk van karakterláncok tárolására:

- char(n): n itt byte és karakter is lehet. Ez a típus maximum 2000 byte tárolására képes. Mérete fixen n, akkor is ha kisebb szöveget töltünk bele, ekkor space-ekkel kitölti a maradék helyet.
- varchar2 és varchar n karaktert (vagy byte-ot) képes tárolni, de maximum 4000 byte-ot. Mérete maximum n, rövidebb szöveg esetén kisebb.
- nchar(n) char Unicode megfelelője.
- nvarchar(n) varchar Unicode megfelelője.
- CLOB LOB adattípus karakterek számára, 8 terrabytenyi karaktert is képes eltárolni.
- Long 2 GB-ig képes tárolni. Az Oracle már nem támogatja.

Mi esetünkben a CLOB lesz a legmegfelelőbb.

Genom-ot tároló tábla Oracle DB-ben például:

```
CREATE TABLE genom (
id NUMBER,
    seq CLOB
)
```

4.6 Tárolás MS SQL-ben

[10] MS SQL-ben a tárolási lehetőségek részben a PostgreSQL-ben és részben az Oracle DB-ben látottakhoz nagyon hasonlóak:

- char(n) fixen n hosszú karakterlánc tárolására alkalmas, ahol n 1 és 8000 közötti. A tárolási méret n byte.
- varchar(n — max) változó hosszúságú karakterlánc tárolására képes, maximum n karaktert, mely itt is 1 és 8000 közötti érték. Max használatával a maximális tárolási méret 2 GB.
- nchar(n) char Unicode megfelelője, n azonban már csak 1 és 4000 közti lehet. A tárolási méret n*2 byte.
- nvarchar(n — max) varchar Unicode megfelelője, szintén n csak 1 és 4000 közti lehet, a maximum tárolási méret viszont itt is 2 GB.
- text változó hosszúságú karakterlánc tárolására alkalmas, maximum hossza 2,147,483,647 lehet, 2,147,483,647 byteon.
- ntext előbbinek a Unicode megfelelője, 1,073,741,823 maximális hosszal, 1,073,741,823 byteon.

Mi esetünkben itt is a text lenne a legmegfelelőbb. Azonban valamiért ki akarják vonni ezt a típust a forgalomból, így valószínűleg a varchar-ral kell majd megelégednünk.

Genom-ot tároló tábla Microsoft SQL-ben például:

```
CREATE TABLE genom (
id NUMBER,
    seq varchar(max)
)
```

5 DNS feldolgozás

5.1 Mintaillesztés

Tipikus eljárás ezeken az adathalmazokon például a mintaillesztés, amikor azokat a pozíciókat keressük, ahol egy minta előfordul a szöveg részstringjeként. Az esetünkben nukleobázisok sorozata a minta és azon pozíciókat keressük a genomban, ahol ez a minta előfordul.

Tehetjük ezt naivan, amikor egyszerűen illesztjük a mintát minden lehetséges pozícióban és sorban végigpróbáljuk a minta karaktereit a szemben levő genom karaktereivel, hogy egyeznek-e.

Gyorsíthatjuk ezt az eljárást, ha feldolgozzuk előre a mintát, keresőtáblákat építve, melyek segítségével a szövegben kihagyhatunk pozíciókat, melyekről tudjuk, hogy nem lesz illeszkedés. Ez a Boyer-Moore algoritmus.

5.2 Indexelés

Feldolgozhatjuk előre a genomot is, indexek használatával. Indexelhetjük például a szöveget részstring indexekkel, például egy hash táblán. A hash táblában eltároljuk a szöveg minden k-asát (k karakterből álló részstringjét) rendezetten az adott részstring előfordulási pozícióival.

Így ezen az adatszerkezeten már elég lesz egy gyors bináris keresést végezni és csak az eredmény pozíciókban vizsgálni a szöveget a karakterek összehasonlításával, hogy megbizonyosodjunk arról, hogy a k karakteren túl is egyezés van.

	kmer text	offs integer[]
1	AAAA	799185,799334,799427,799459,799460,799522,799523,799524,799525,799
2	AAAC	795440,795492,795624,795647,795699,795886,795926,796125,796169,796
3	AAAG	798037,798146,798477,798514,798539,798760,798824,798882,799039,799
4	AAAT	798566,798571,798591,798616,798652,798697,798736,798755,798924,799
5	AACA	795946,795979,795984,796126,796423,796708,796923,797010,797288,797
6	AACC	790710,791489,791631,792082,792141,792213,792423,792612,793128,793
7	AACG	751717,752070,752599,754055,754261,758032,758602,763831,766929,767
8	AACT	795441,795625,795634,795700,796170,796397,796465,796982,797403,797
9	AAGA	798478,798494,798522,799064,799070,799163,799176,799179,799336,799
10	AAGC	795482,795610,796114,796136,796200,796364,796782,797036,797113,797
11	AAGG	795239,795336,795411,795466,795509,795559,795592,795629,795814,795

Figure 2: Genom indexelése hash táblában

Adatbázisban tárolás esetén nem ilyen egyszerű a helyzet, nagy k esetén nagy row overheadbe ütközhetünk, mely esetben az indexelt keresés még a naiv illesztésnél is lassabb lehet, főleg titkosított genom tárolás esetén.

Ez megoldható, ha egy cellában tárolt genom index egészét egyetlen cellában tároljuk. Konvertáljuk json stringbe az indexet és ha akarjuk ezt a cellát titkosítjuk.

Amikor lekérdezzük az indexet, visszakonvertálunk, feloldjuk a titkosítást és feldolgozzunk a szokott módon a cellához tartozó indexet.

Más indexelő struktúrák és algoritmusok is használhatóak: prefix fák, szuffix fák, szuffix tömbök, FM indexek.

5.3 Közelítés és összehangolás

Szekvenálási hibák és a természetes variációk miatt fontos, hogy mikor genomokkal dolgozunk, képesek legyünk megengedni egy bizonyos számú eltérést a mintaillesztés során. Ez a közelítő illesztés

Néha ez nem elég, így szekvencia összehangolást használunk, mely egy módja annak, hogy egyeztessük a szekvenciákat, így hasonló régiókat azonosítsunk be, melyek jelezhetnek funkcionális, strukturális, vagy evolucionális kapcsolatot a szekvenciák között.

Ekkor nem csak eltéréseket engedünk meg, hanem hézagokat is, azaz, például azt, hogy a minta egy szakaszon illeszkedjen, aztán csak kihagyás után legyen újabb illeszkedés. Ennek megoldására dinamikus programozást használunk.

5.4 Összeszerelés

Összeszerelés, mikor összehangoljuk és összefuttatjuk a töredékeit egy hosszabb DNS szekvenciának, hogy rekonstruáljuk az eredeti szekvenciát. Erre azért van szükség, mert a DNS szekvenáló technológia nem képes teljes genomokat beolvasni egyszerre, ehelyett kisebb, 20 és 30000 közti hosszúságú darabokat olvas, a használt technológia függvényében.

Erre használhatjuk a legrövidebb közös részstring megkeresésének algoritmusát, átfedési gráfokat használó algoritmusokat, vagy De Bruijn gráfot.[6]

5.5 Feldolgozás PostgreSQL-ben

[8] PostgreSQL-ben a genomok feldolgozását megkönnyítő eszköz a pl/python, mely segítségével pythonban írhatunk tárolt eljárásokat.

Ehhez először is hozzáadjuk a plpython kiegészítőt és nyelvet, ezután már írhatjuk is a python eljárásokat. Ennek megkönnyítésére saját típusokat is létrehozhatunk.

Példa típus és python függvény:

```
CREATE EXTENSION plpython3u;
CREATE LANGUAGE plpython3u;

CREATE TYPE Sequences AS(
id integer,
    seq text
)

CREATE OR REPLACE FUNCTION GenomReader(filename varchar(100))
RETURNS SETOF Sequences
AS $$
...
$$ LANGUAGE plpython3u;
```

Ezt már a megszokott módon használhatjuk.

5.6 Feldolgozás Oracle DB-ben

[9] Oracle DB-ben a genomok feldolgozását az teszi könnyűvé, hogy lehetőségünk van java kódokat futtatni tárolt eljárásként.

Itt helyben létrehozhatunk java forráskódokat, ezután létrehozzuk hozzá a tárolt eljárást vagy függvényt

Java kód létrehozása és tárolt eljárásként használata például:

```
CREATE OR REPLACE AND COMPILE JAVA SOURCE NAMED "GenomReader" AS
public class GenomReader
{

    public static void read(String url) throws Exception
    {
    ...

CREATE OR REPLACE PROCEDURE readGenome(url in VARCHAR2)
AS LANGUAGE JAVA NAME 'GenomReader.read(java.lang.String)';
```

Ezt pedig már a szokásos módon használhatjuk.

5.7 Feldolgozás MS SQL-ben

[10] Microsoft SQL-ben a CLR szolgáltatás segíti a genomok feldolgozását. Ezzel lehetőségünk van C#, C kódokat futtatni tárolt eljárásként.

Ehhez először engedélyeznünk kell a külső scriptek futtatását és mivel url-el keresztül fogunk genomokat tartalmazó fájlokat beolvasni, ki kell kapcsolnunk a clr strict securityt és megbízhatóvá kell tennünk az adatbázisunkat.

CLR bekonfigurálásának első lépései:

```
ALTER DATABASE genom_db
SET TRUSTWORTHY ON;
EXEC sp_configure  'external scripts enabled',1;
EXEC sp_configure 'clr strict security', 0;
RECONFIGURE WITH OVERRIDE;
```

Ezután egyszerűen be kell töltenünk a dll kiterjesztésű assemblyt. Ezt is külső eszközök elérésének engedélyezésével tegyük. Ezután létrehozunk egy típust, mely az osztályt fogja tartalmazni amit használni akarunk. Végül létrehozhatjuk a tárolt eljárást, vagy függvényt, amit már C#-ban, vagy C-ban megírtunk.

Assembly, típus és függvény létrehozása például:

```
CREATE ASSEMBLY ReadGenom from 'C:\ReadGenom.dll'
WITH PERMISSION_SET = EXTERNAL_ACCESS

CREATE TYPE GenomReader
EXTERNAL NAME ReadGenom.GenomReader;

CREATE FUNCTION GetGenom(
@url VARCHAR(100)
)
RETURNS TABLE(
id INTEGER
seq VARCHAR(MAX)
)
AS EXTERNAL NAME [ReadGenom].[GenomReader].[GetGenom];
```

Ez pedig már a szokott módon futtatható.

6 Titkositás

6.1 Szimmetrikus és publikus kulcsú titkosítás

A publikus kulcsú titkosítás kulcspárokat használ: a publikus kulcs közismert lehet, míg a privát kulcsot csak a tulajdonos ismeri. Ez 2 funkcióval jár:

Autentikáció, amikor a publikus kulcs igazolja, hogy a privát kulcs tulajdonosa küldte az üzenetet.

Titkosítás, amikor csak a privát kulcs tulajdonosa tudja dekódolni az üzenetet, melyet a publikus kulccsal titkosítottak.

Szimmetrikus kulcsú titkosítók ezzel szemben ugyan azt a kulcsot használják a titkosításra, mint a titkosítás feloldására. A mi esetünkben a felhasználó ezzel a kulccsal fér hozzá a genomjához.

6.2 Folyam titkosítók

A folyam titkosító egy szimmetrikus kulcsú titkosító, mely a titkosítatlan szöveg számjegyeit egy pseudovéletlen titkosító számjegy folyammal kombinálja. A folyam titkosító minden titkosítatlan számjegyet egyesével titkosít a soron következő kulcs folyam számjegyével. Gyakorlatban a számjegyek a bitek, a kombináló pedig XOR.

A pszeudovéletlen titkosító számjegy folyam általában egy vélelten mag értékből generálódik. A mag érték szolgál a titkosító kulcsként a titkosított szövegfolyam feloldására.

Általában gyorsabbak a blokk titkosítóknál, viszont nagyobb körültekintéssel kell használni, sokféle sikeres támadás létezik ellene.

Genomok titkosítására az RC4 folyam titkosítót vizsgáljuk meg. [11] RC4 egyszerű és gyors, de több sebezhetőségét is felfedezték már. [12]

6.3 Blokk titkosítók

Blokk titkosítók egy transzformációval dolgoznak, mely szimmetrikus kulccsal és egy blokkal, azaz egy fix-hosszú bitcsoporttal paraméterezhető.

Blokk titkosítók 2 párba állított algoritmusból állnak, egyik a tiktosításért, másik a titkosítás feloldásáért felel, mely a titkosító algoritmus inverze. Mindkettő két inputot fogad: egy bemenő n bit hosszú blokkot és egy k bit méretű kulcsot. Mindkettő n bitből álló transzformált blokkot ad vissza.

6.4 Rijndael

Blokk titkosítók közül például az AES-t fogjuk megvizsgálni. AES a Rijndael titkosításon alapszik, néhány nemzetközi szervezet használja, mint bankok. Blokk méret 128-bit, kulcs méret 128-bit AES128 esetén, 192-bit AES192 esetén és 256-bit AES256 esetén.

AES 10 kulcsot generál az eredetiből, ezeket és a titkositandó szöveget eltárolja 4X4 táblákban, a titkositandó szöveget 128 bites csonkokban. A 128 bites titkositandó szöveg darabjait 10 körös procedúra dolgozza fel. Ha a kulcs mérete 192 bit, akkor ez 12 körre, ha 256 bit, akkor 14 körre változik.

Rindjael titkositásban ehhez képest annyi a különbség, hogy a blokk és kulcsméret is 32 bit tetszőleges többszöröse lehet, de minimum 128-bit, maximum 256 bit.[13]

Vegyük számításba, hogy AES-el szembeni támadás során sikeresen megszerezhető a kulcs. [14]

6.5 Feistel titkosítók

Feistel titkosítók több körben dolgozzák fel a titkosítandó szöveget. Minden kör egy helyettesítő és egy permutáció lépésből áll.

Az algoritmus minden kör eslő részében felbontja a titkosítandó szöveget 2 egyenlő méretű részre. A jobb oldali változatlanul megy a következő körbe. A bal oldalira egy titkosító eljárást alkalmaz melynek 2 bemenete a jobb oldal és a kulcs. Az eredményt XOR-olja az eredetivel.

A második lépésben a 2 felet felcseréli.

Több kör használata nagyobb biztonsághoz vezet, kevesebb kör gyorsabb algoritmust ad.[15]

- Blowfish (bf) egy 16-körös Feistel hálózat 64-bites blokk mérettel és változó kulcs hosszal 32 bittől 448 bitig.[16]
- A DES 16-körös Feistel struktúrát használ, mindössze 56 bites kulccsal 64 bites blokkokon. A kulcs mérete miatt nagyon sebezhető.[17]
- Bár mint majd később kiderül, Triple DES (3DES) lassabb a többinél, jó védelmet nyújt, használják az elektonikus fizetés iparában is. Ez nem egy teljesen új blokk titkosító, egyszerűen 3 56 bites DES kulcsot használ 64 bites blokkokon. 3 féleképpen használhatunk hozzá kulcsot: lehet mind három kulcs független, lehet 2 független és 2 identikus, végül lehet mind három identikus.[18]
- Cast5 egy 12, vagy 16 körös Feistel hálózat 64 bites blokk mérettel és 40 és 128 bit közötti kulcs mérettel. Ez az alapértelmezett titkosító a GPG bizonyos verzióiban és PGP-ben.[19]
- Bár az RC4 már folyam titkosító, az RC2 egy 64-bites, 18 körös Feistel blokk titkosító.[20]

Vegyük figyelembe, hogy 64-bites, vagy kisebb blokkokkal dolgozó titkosítók sebezhetőek a születésnap támadással a kis blokkméret miatt. [21]

6.6 Titkosítás PostgreSQL-ben

[8] PostgreSQL-ben a pgcrypto kiegészítőt használhatjuk genomjaink titkosított tárolására. Ez a modul criptográfiai funkciókat nyújt, mint digest() és encode(), melyek kiszámítják a bináris hash-ét egy adott adatnak, vagy hmac, mely csak kulccsal működik. Jelszavak hash-elésére a crypt() és gen_salt() függvényeket használhatjuk.

Ez a modul támogatja a szimmetrikus kulcsú és publikus kulcsú titkosítást is. Szimmetrikus titkosítást a pgp_sym_encrypt() függvénnyel, feloldását a pgp_sym_decrypt() függvénnyel végezzük.

A titkosítás során választhatunk titkosító algoritmust és a 'cipher-algo' opcióval állíthatjuk be, mely a bf. aes128, aes192, aes256, 3des és cast5 titkosításokat nyújtja.

Titkosítás PostgreSQL-ben:

```
CREATE TABLE encrypted_lambda_virus AS(
SELECT id, pgp_sym_encrypt(seq, 'password', 'cipher-algo=aes128')
    FROM lambda_virus;
);
```

Titkosítás feloldása PostgreSQL-ben:

```
CREATE TABLE lambda_virus AS(
SELECT id, pgp_sym_decrypt(seq, 'password')
    FROM encrypted_lambda_virus;
);
```

	id integer	seq bytea
1	1	[binary data]
2	2	[binary data]
3	3	[binary data]
4	4	[binary data]
5	5	[binary data]
6	6	[binary data]
7	7	[binary data]
8	8	[binary data]
9	9	[binary data]
10	10	[binary data]
11	11	[binary data]

Figure 3: Titkosított genom PostgreSQL-ben

6.7 Titkosítás Oracle DB-ben

[9] Az oracle adatbázisrendszer úgy nevezett transzparens adat titkositást (Transparent Data Encryption) használ. Ez az AES256, AES192, AES128 és 3DES168 adattitkosításokat támogatja.

Ez egy speciális esete a szimmetrikus kulcsú titkosításnak. TDE az egész adatbázist titkosítja az adatbázis titkosító kulccsal. Ezt a kulcsot más kulcsok védik, vagy az adatbázis mester kulcs.

Titkositott genom létrehozása TDE használatával:

```
CREATE TABLE encrypted_lambda_Virus (
     id NUMBER,
     seq LONG VARCHAR ENCRYPT USING '3DES168'
);
```

TDE azonban nem használható nagy adattipusokra, mint a BLOB-ra és CLOB-ra. Ennek megoldására haszálhatjuk a DBMS_CRYPTO csomag ENCRYPT és DECRYPT függvényeit, melyek már támogatják az előzőek mellett a DES és RC4 algoritmusokat is.

Titkosítás DBMS_CRYPTO használatával:

```
CREATE TABLE encrypted_lambda_virus AS(
SELECT id, DBMS_CRYPTO.encrypt(
UTL_RAW.CAST_TO_RAW (seq),
     4353 /* = dbms_crypto.DES_CBC_PKCS5 */,
     'A1A2A3A4A5A6CAFE')
FROM lambda_virus
);
```

6.8 Titkosítás Microsoft SQL-ben

[10] Microsoft SQL-ben is elérhető a transzparens adat titkositást (Transparent Data Encryption). Ez a DES, Triple DES, TRIPLE_DES_3KEY, RC2, RC4, 128-bit RC4, 192-bit 3DES, 128-bit AES, 192-bit AES, and 256-bit AES adattitkosításokat támogatja. SQL Server 2016-tól azonban csak az ASES_128, AES_192 és AES_256 támogatott.

A TDE használatának megkezdéséhez létre kell hoznunk egy mester jelszót, tanúsítványokat kell létrehoznunk, felhasználónként egyet, be kell állítani a szerver szintű titkosítást és engedélyezni kell a titkosítást a szerveren.

TDE bekonfigurálása egy mester jelszó létrehozásával:

```
CREATE MASTER KEY ENCRYPTION BY PASSWORD = 'password';
```

Tanúsítványok létrehozása:

```
CREATE CERTIFICATE Cert1 WITH SUBJECT = 'Cert1';
CREATE CERTIFICATE Cert2 WITH SUBJECT = 'Cert2';
CREATE CERTIFICATE Cert3 WITH SUBJECT = 'Cert3';
```

Server szintű titkosítás beállítása:

```
CREATE DATABASE ENCRYPTION KEY
WITH ALGORITHM = AES_128
ENCRYPTION BY SERVER CERTIFICATE Cert1;
```

Titkosítás engedélyezése a továbbiakban:

```
ALTER DATABASE adatb
SET ENCRYPTION ON;
```

Több módszert is használhatunk a felhasználó azonosítására: például tanúsítványt, jelszót. Először megnyitja a felhasználó a kulcsát, tevékenykedhet, majd bezárja. Mint egy bejelentkezés és kijelentkezés.

Jelszóval, tanúsítvánnyal védett szimmetrikus kulcs hassnálata:

```
CREATE SYMMETRIC KEY key1
    WITH ALGORITHM = AES_192
ENCRYPTION BY PASSWORD = 'password'

CREATE SYMMETRIC KEY key2
    WITH ALGORITHM = AES_256
ENCRYPTION BY CERTIFICATE Cert2;
```

Felhasználó úgymond belépése, innentől ezzel a kulccsal titkosítunk:

```
OPEN SYMMETRIC KEY key1
   DECRYPTION BY PASSWORD = 'password'
```

Titkosított genom létrehozása:

```
SELECT id, EncryptByKey(Key_GUID('key_name'), seq) as enc_seq
INTO enc_lambda_virus
    FROM lambda_virus
```

Ami viszont nehézség, hogy az itteni titkosító VARBINARY(8000) típussal tér vissza. Ez azt jelenti, hogy túl nagy cellák titkosítására képtelen. Egy bizonyos szintig megoldást jelenthet a titkosítás előtti tömörítés.

Titkosított genom létrehozása:

```
SELECT id, EncryptByKey(Key_GUID('key_name'), COMPRESS(seq)) as enc_seq
INTO enc_lambda_virus
    FROM lambda_virus
```

Másik dolog, amire figyelni kell: a titkosítás feloldása varbinary typussal tér vissza, még konvertálni kell.

Titkosítás feloldása:

```
SELECT id, CONVERT(varchar, DecryptByKey('key_name')) as seq
    FROM lambda_virus;
```

Tömörített és titkosított genom feloldása:

```
SELECT id, CAST(DECOMPRESS(DecryptByKey(seq))  AS VARCHAR(MAX) ) as seq
    FROM chr1_100r_aes256
```

Ne feledjük bezárni a kulcsot, miután végeztünk. Innentől ez a kulcs már nem használható titkosításra

Felhasználó úgymond kilépése:

```
CLOSE SYMMETRIC KEY key1
   DECRYPTION BY PASSWORD = 'password'
```

7 Tömörítés

7.1 Tömörítő algoritmusok

A zip, gzip és zlib is meg fog jelenni, mint használt algoritmus, ezek pedig mind a deflate algoritmuson alapulnak, mely kombinálja a LZ77 és Huffman-kódolást.[22]

Duplikált stringeket keres a bejövő adatban. A második előfordulása egy stringnek lecserélődik egy pointerre az előző stringre egy pár formájában: távolság, illeszkedés hossza. Távolságok 32 ezer bytera limitálva, hosszak 258 bytera. Ha egy string sehol nem fordul elő az előző 32 ezer byteon, akkor nem cserélődik le.

A szöveg darabok és illeszkedés hosszok egy Huffman fába kódolódnak, az illeszkedési távolságok pedig egy másik fában. A fák kompakt formában tárolódnak az egyes blokkok elején.

7.2 Tömörítés PostgreSQL-ben

[8] PostgreSQL-nek van egy alapvető tömörítő mechanizmusa: a TOAST (The Oversized-Attribute Storage Technique). PostgreSQL fix lapméretet használ: 8kb. Nagy mezők értékeit tömöríti és/vagy felbontja több fizikai sorba.

Csak változó hosszúságú adattípusokat (varchar, text) támogatja a TOAST, ahol az első 32-bites szava bármilyen tárolt értéknek tartalmazza a teljes méretét byteban. Más megkötés nincs.

Ez egy egyszerű és gyors tagja a tömörítő technikák LZ családjának.

A TOAST kód csak akkor aktiválódik, amikor egy tárolandó sor értéke a táblában szélesebb a TOAST_TUPLE_THRESHOLD byte-nál: 2kb-nál. A TOAST kód ekkor tömöríti és/vagy elmozgatja a soron kívüli mező értékeket, amíg a sor értéke kisebb nem lesz TOAST_TUPLE_TARGET byte-nál: szintén 2kb.

Több stratégiát is képes alkalmazni a TOAST:

- **PLAIN** a nem TOAST-olható típusok stratégiája.
- **EXTENDED** tömörítést és soronon kívüli tárolást is alkalmaz. Ez az alapértelmezet stratégia.
- **EXTERNAL** engedi a soron kívüli tárolást, de a tömörítést nem. Ezzel a text-ek részstring műveletei gyorsabbak lesznek.

- **MAIN** engedi a tömörítést, de nem engedi a soron kívüli tárolást.

Tömörítési stratégia meghatározása PostgreSQL-ben:

```
ALTER TABLE lambda_genom
ALTER COLUMN seq SET STORAGE EXTERNAL
```

A pgcrypto is lehetővé tesz tömörítést a 'compress-algo' opcióval, mely nyújtja a ZIP és ZLIB tömörítéseket. Beállíthatjuk a tömörítési szintet is a 'compress-level' opcióval.

Tömörítés PostgreSQL-ben:

```
CREATE TABLE compressed_lambda_virus AS(
SELECT id,
    pgp_sym_encrypt(seq, 'password',
     'compress-algo=ZIP, compress-level=9')
    FROM lambda_virus;
);
```

Kicsomagolás PostgreSQL-ben:

```
CREATE TABLE lambda_virus AS(
SELECT id, pgp_sym_decrypt(seq, 'password')
    FROM encrypted_lambda_virus;
);
```

7.3 Tömörítés Oracle DB-ben

[9] Oracle 3 szintű tömörítést tesz lehetővé LOB-k, nagy objektumok számára. Ide tartozik az általunk használt CLOB is.: HIGH, MEDIUM, LOW. A szint megadása nélkül MEDIUM az alapértelmezett.

Nyilván a szint növelésével több helyet spórolhatunk, viszont a nagyobb tömörítéssel és kicsomagolással CPU időt veszítünk.

Tömörítés Oracle DB-ben:

```
CREATE TABLE compressed_lambda_virus(
  id number,
  seq CLOB
)
LOB(seq) (
COMPRESS
)
```

Tömörítés módosítása Oracle DB-ben:

```
ALTER TABLE compressed_lambda_virus MODIFY
    LOB(a) (
         COMPRESS HIGH
    );
```

Oracle DB-ben lekérdezéskor automatikus a kicsomagolás, ezzel nem kell külön foglalkozni.

Használható tábla szintű tömörítést is.

Tábla szintű tömörítés Oracle DB-ben:

```
CREATE TABLE compressed_lambda_virus(
  id number,
  seq CLOB
)
Compress for all operations
```

Az all operations itt azt jelenti, hogy minden DML művelet (INSERT, UPDATE, stb) esetén aktiválódik a tömörítés.

Továbbá lehetőség van itt is meghatározni tömörítési szintet is, valamint 2 féle megközelítést, aszerint, hogy milyen gyakorisággal valószínűsíthető az adat lekérdezése.

Tábla szintű tömörítés Oracle DB-ben:

```
create table compressed_lambda_virus
nologging compress for archive high
as select * from lambda_virus;
```

Az előbbi példában a compressed for mögött szerepelhetne query is, ezzel jelezve, hogy ez egy gyakran lekérdezett adat, ezzel szemben az archive ennek ellentétét jelzi. Ez után pedig megadható a szint, ami itt például high.

Ez a tömörítés annyit tesz, hogy egy cellába von össze egy oszlopban ismétlődő adatokat.

7.4 Tömörítés Microsoft SQL-ben

[10] Microsoft SQL-ben egyszerűek a tömörítési lehetőségek, a compress és decompress függvények használhatók erre a célra, melyek GZIP tömörítési algoritmust használnak.

Tömörítés MS SQL-ben:

```
SELECT id, COMPRESS(seq) cmp_seq
into compressed_lambda_virus
    FROM lambda_virus;
```

Mindkettő függvény varbinary(max) típust ad vissza, így kicsomagolásnál még át kell alakítani az eredményt.

Kicsomagolás MS SQL-ben:

```
SELECT id, CAST(DECOMPRESS(seq) AS VARCHAR(MAX)) seq
into lambda_virus
    FROM compressed_lambda_virus;
```

Itt is használható tábla szintű tömörítés is. Ekkor meg kell adnunk, hogy row-store, vagy columnstore szeretnénk tömöríteni. Ez esetben nem kell külön kicsomagolni.

Tábla szintű tömörítés MS SQL-ben:

```
ALTER TABLE lambda_virus
REBUILD PARTITION = ALL
WITH (DATA_COMPRESSION = ROW);
```

8 Jogosultság kezelés

8.1 Jogosultság kezelés PostgreSQL-ben

[8] PostgreSQL-ben szerepeket használunk a hozzáférési jogosultságok kezelésére. Szerepek lehetnek felhasználók, csoportok, vagy mindkettő. Szerepek birtokolhatnak adatbázis objektumokat és kezelhetik a hozzáférési jogokat ezeken.

Szerepek készítése, törlése és lekérdezése a pg_roles katalógusból:

```
CREATE ROLE name
DROP ROLE name
SELECT rolname FROM pg_roles
```

Egy friss rendszer mindig tartalmaz egy előre definiált szerepet, mely mindig a szuperfelhasználó és ugyan az a neve, mint az operációs rendszer felhasználójának, aki inicializálta a klasztert. Legyünk azonban óvatosak, egy szuperfelhasználó bármilyen jogosultsági ellenőrzésen át tud jutni. Szuperfelhasználóként, több szuperfelhasználót is létre lehet hozni. Szuperfelhasználónak lenni egy szerep tulajdonság.

Példa tulajdonságok: szuperfelhasználó, engedély a belépésre, adatbázis létrehozására, szerepek létrehozására és a jelszó beállítása:

```
CREATE ROLE name SUPERUSER
CREATE ROLE name LOGIN
CREATE ROLE name CREATEDB
CREATE ROLE name CREATEROLE
CREATE ROLE name PASSWORD 'string'
```

Ezek a tulajdonságok módosíthatók a szerep létrehozása után:

```
ALTER ROLE name CREATEROLE CREATEDB;
```

Minden objektumnak van egy tulajdonosa. Normális esetben a tulajdonos az a szerep, mely létrehozta az objektumot. Alapértelmezés szerint, csak a tulajdonos, vagy a szuperfelhasználó tud bármit tenni az objektummal. Engedélyek állíthatók ennek megváltoztatására. Sok féle engedély létezik: SELECT, INSERT,

UPDATE, DELETE, TRUNCATE, REFERENCES, TRIGGER, CREATE, CONNECT, TEMPORARY, EXECUTE, USAGE és ALL az összes engedélyhez.

A GRANT parancs használható engedélyek beállítására. PUBLIC kulcsszó használható arra, hogy mindenki megkapha az engedélyt a rendszerben. Engedélyek visszavonhatók a REVOKE paranccsal.

Engedély megadása és visszavonása:

```
GRANT privilege ON object TO role
REVOKE privilege ON object FROM role
GRANT privilege ON object TO PUBLIC
```

Ha csoportokat akarunk használni, először egy szerepet kell létrehoznunk. Ezek általában nem rendelkeznek a LOGIN tulajdonsággal. Minden tagja a csoportnak használhatja a SET ROLE parancsot, hogy ideiglenesen használja az engedélyeit a csoportnak és az objektumok amiket ekkor létrehoz a csoport lesz a tulajdonosa. Ezen kívül az INHERIT tulajdonsággal rendelkező szerepek automatikusan használják a csoportjuk engedélyeit.

Szerepek hozzáadása és elvétele egy csoporttól, valamint csoport jogainak használata:

```
GRANT group_role TO role1, ...
REVOKE group_role FROM role1, ...
SET ROLE group_role
```

Esetünkben, szuperfelhasználók lennének az adatbázis üzemeltetői, és a felhasználók genomjai titkosítva kerülnének tárolásra, hogy a szuperfelhasználó ne legyen képes olvasni őket a jelszó nélkül. A felhasználók a saját genomjaik tulajdonosai lennének, de például orvosaiknak megadhatnák az lekérdezési jogot, ha szeretnék. Valamint az orvosok csoportokba rendeződhetnének.

8.2 Jogosultság kezelés Oracle DB-ben

[9] Oracle DB-ben egy felhasználói jog lehet bizonyos típusú SQL parancsok futtatása, hozzáférés más objektumához, PL/SQL csomag futtatása, és így tovább.

A szerepeket itt felhasználók (általában adminisztrátorok) hozzák létre, hogy csoportosítsák az engedélyeket, vagy más szerepeket. Több engedély kategóriát különböztetünk meg.

Rendszer engedélyek: ezek a kedvezményezettnek engedélyezik, hogy sztenderd adminisztrátori feladatokat végezzenek az adatbázisban. Ez a jog egy bizonyos akció elvégzésére egy bizonyos típusú bármilyen séma objektumon.

Példa táblaterek létrehozásának engedélyének megadása és visszavonása egy felhasználónak:

```
GRANT ANY CREATE TABLE SPACE TO PUBLIC;
REVOKE ANY CREATE TABLE SPACE FROM PUBLIC;
```

Itt is használható a PUBLIC kulcsszó, mely egy szerep, melyet minden felhasználó megkap a létrejöttekor. Az ANY kulcsszóval pedig egész objektumkategóriákkal kapcsolatos engedélyek kezelhetők. Az ADMIN OPTION kulcsszavak kiadása esetén, a kedvezményezett szintén képes lesz továbbadni a kapott jogokat.

Szükséges engedélyek rendszer engedélyek kezeléséhez:

```
GRANT ANY PRIVILEGE TO user;
GRANT SYSTEM PRIVILEGE TO user ADMIN OPTION;
```

Felhasználói engedélyek: Szerepek engedélyezhetők és megvonhatók felhasználóktól, engedélyezni kell egy szerepet egy felhasználónak, mielőtt használni tudná.

Példa szerep engedélyeinek megadása és visszavonása egy felhasználónak:

```
GRANT role TO user;
REVOKE role FROM user;
```

Jelszóval védett szerep létrehozása és használata:

```
CREATE ROLE role IDENTIFIED BY password;
SET ROLE role IDENTIFIED BY password;
```

Objektum engedélyek: Minden objektumhoz engedélyek vannak asszociálva.

Példa lekérdezés engedélyezése és minden engedély visszavonása egy táblán egy felhasználónak:

```
GRANT SELECT ON table TO user;
REVOKE ALL ON table FROM user;
```

Tehát két féleképp lehet egy felhasználónak engedélyt adni: explicit, vagy az engedély megadásával egy szerepnek, majd a szerep engedélyezésével a felhasználóknak.

8.3 Jogosultság kezelés Microsoft SQL-ben

[10] Jogosultságkezelés Microsoft SQL-ben nagyon hasonló a jogosultságkezeléshez Oracle DB-ben. Itt is szerepekhez rendelhetjük a felhasználókat és szerep szinten menedzselhetjük a hozzáféréseket.

Sysadmin szerephez tartoznak a rendszer adminisztrátorok.

Példa felhasználó hozzáadása szerephez, szerep átnevezése, szerep elvétele felhasználótól:

```
ALTER ROLE szerep ADD MEMBER felhasználó;
ALTER ROLE szerep WITH NAME = újszerep;
ALTER ROLE újszerep DROP MEMBER felhasználó;
```

MS SQL 3 féle jogosultságkezelő állítást használ:

- GRANT engedély megadása.
- REVOKE engedély megvonása. Ez az alapállapot. Azonban a felhasználó, vagy szerep melytől megvonásra került az engedély, továbbra is megörökölheti azt más szerepeken keresztül.
- DENY megvonja az engedélyt úgy, hogy megörökölhető se legyen. Ez azonban nem vonatkozhat a sysadmin szerep tagjaira.

Példa lekérdezés jogának megadása tábla táblán felhasználó számára, majd a jog megvonása:

```
GRANT SELECT ON OBJECT::tábla TO felhasználó;
REVOKE SELECT ON OBJECT::tábla TO felhasználó;
```

9 Eredmények

Elkészült egy-egy kisebb könyvtár genomok tárolására, feldolgozására és titkosítására. A feldolgozó függvényekkel lehetőség van mintaillesztés, közelítés és indexelt illesztés végzésére genomokon és titkosított genomokon.

Jogosultságok beállíthatóak az adathozzáférések felügyelésére és felhasználói hierarchiák is létrehozhatók.

A titkosított genomokhoz természetesen csak a felhasználó hitelesítése után lehet hozzáférni.

Következzen néhány mérés a futási időre vonatkozóan különböző titkosítások, tömörítések használatával, valamint a foglalt terület ellenőrzése.

10 Eredmények PostgreSQL-ben

10.1 Genom eltárolása

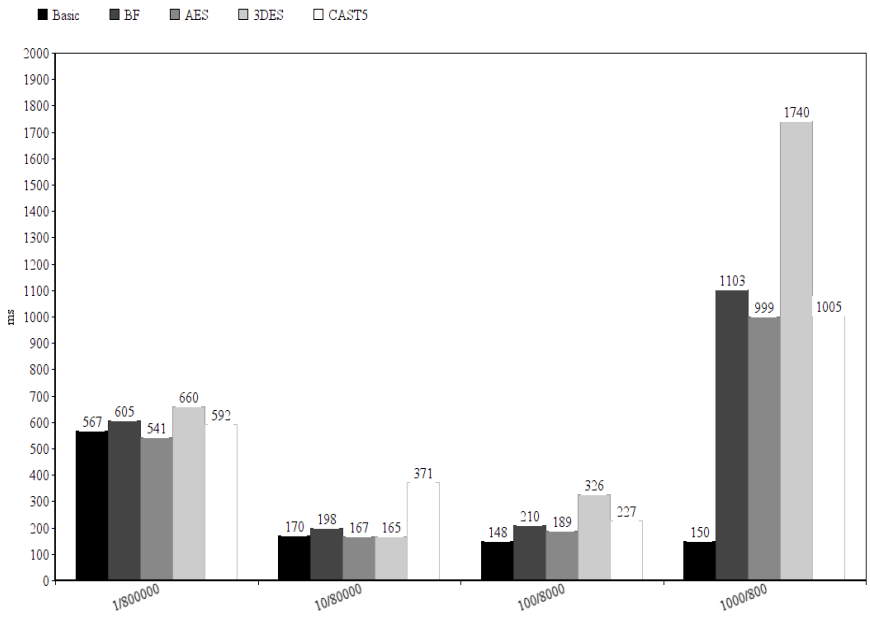

Figure 4: Egy 792 KB genom beolvasása különböző titkosításokkal és sorokba osztással: x: sorok száma/karakterek száma soronként, y: idő ms-ban

Első ábránkon genomok PostgreSQL-be olvasását mértem. Ennek ideje 1 sorra hasonló a különböző titkosításokkal. Ami érdekes, hogy 10 és 100 sorra, kisebb részek tárolásával még csökkent az idő, végül 1000 sorra a titkosítások használatával nagyon megugrik, főleg a titkosítatlan beillesztéshez képest.

Titkosítások közt egyébként az látszik, hogy 3des a leglassabb, majd Cast5. AES teljesített a legjobban.

10.2 Genom feldolgozása

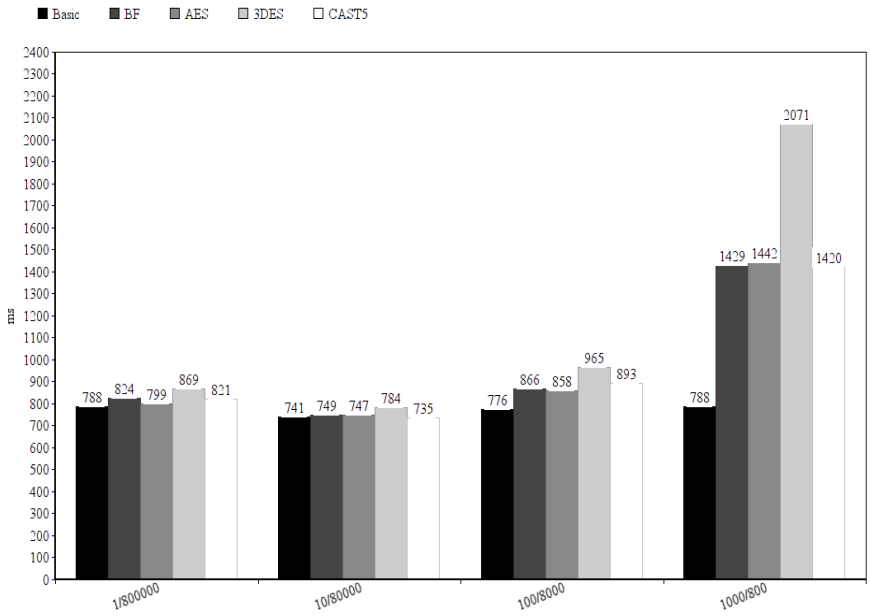

Figure 5: Naive mintaillesztés 792 KB titkosított genomon különböző sorokba osztással: x: sorok száma/karakterek száma, y: idő ms-ben

Titkosított genomokon végzett naive mintaillesztés futási idejét méréve azt mondhatjuk, általában lassabb algoritmusaink lesznek 3des-el titkosítva. A többi hasonlóan szerepelt.

Továbbá azt látjuk még, hogy a sorok számának növekedésével a távolság a titkosítatlan és titkosított genomon futó algoritmus közt nő futási időben. De egyetlen soron a titkosított verzió nem sokkal lassabb a titkosítatlannál.

Bár PostgreSQL text típusának használatával az egy cellában tárolható adat méretének elméleti maximuma 1 GB, a memóriával spórolás miatt érdemes lehet mégis darabokra bontani a 3 GB-os genomot. Valamint ha régiókra bontva tároljuk a genomot és tudjuk hogy körülbelül hol akarunk keresni, nem kell annyi szöveget áttekinteni felbontás esetén.

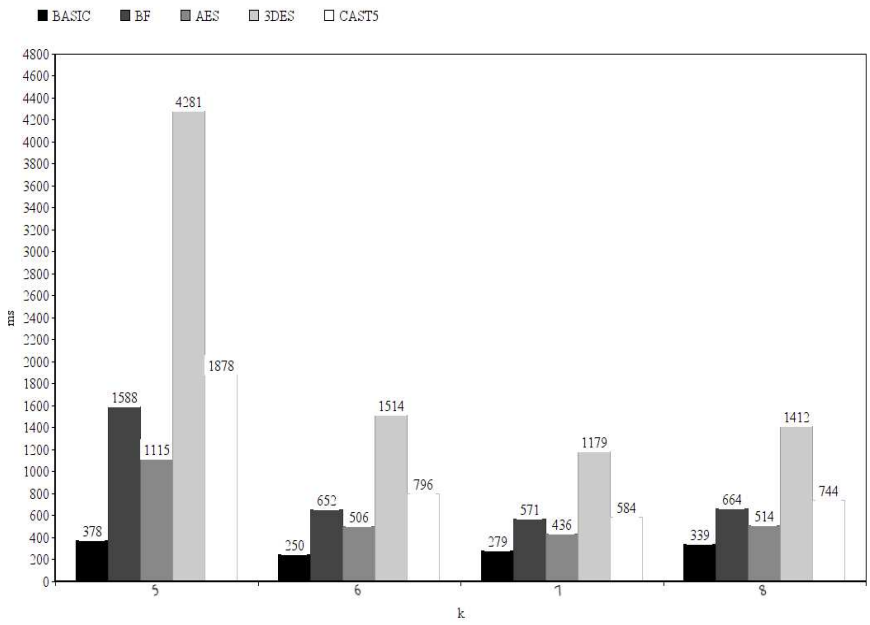

Figure 6: Egy 792 KB titkosított genom lekérdezése különböző méretű k-asok tárolásával: x: k-as hossza, y: idő ms-ban

Index lekérdezése esetén azt láthatjuk, hogy az eredményeink erősen függnek attól, hogyan választjuk meg k-t. Titkosítás nélkül k=6 a legjobb, de titkosítással k=7 a legjobb.

Itt is 3Des a leglassabb, azonban jobban kijönnek a különbségek a többi titkosítás közt is. Most már tisztán láthatjuk, hogy AES a leggyorsabb, BF a második és CAST5 a harmadik a titkosítás feloldásában.

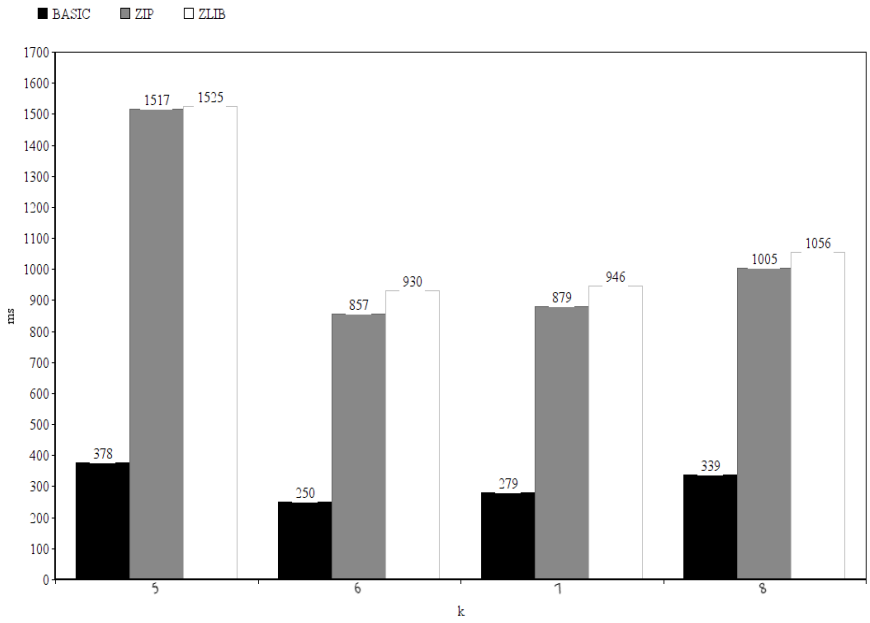

Figure 7: Egy 792 KB tömörített genom index lekérdezése különböző méretű k-asok tárolásával: x: k-as hossza, y: idő ms-ban

Ha tömörítjük az indexet, azt láthatjuk, hogy tömörítéssel is k=6 a legjobb választás és azt is láthatjuk, hogy a ZIP gyorsabb a ZLIB-nél.

10.3 Terület igény

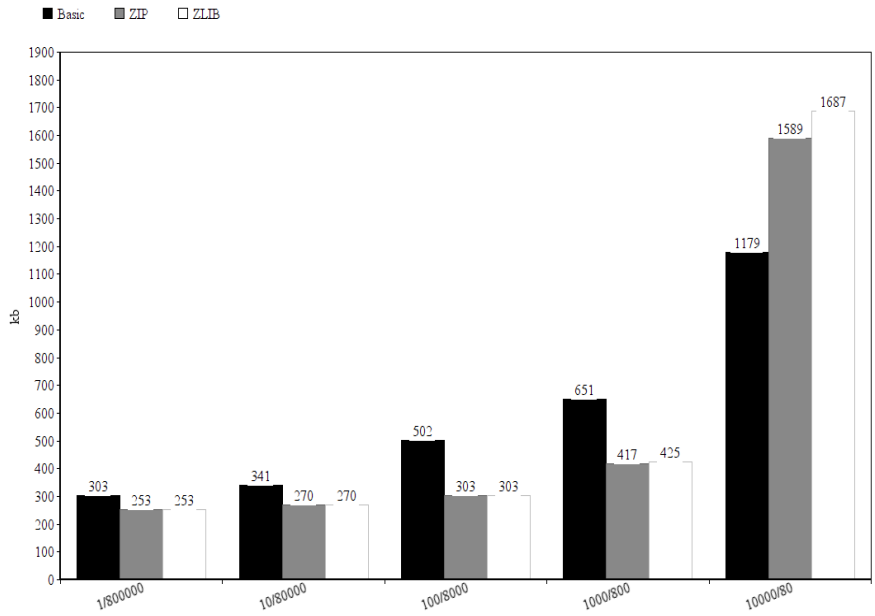

Figure 8: 792 KB genom tömörítése különböző sorokba osztással: x: sorok száma/karakterek száma, y: teljes méret KB-ban

Amit itt láthatunk, hogy megpróbáltuk tárolni a 792 KB-os kivonatát az emberi genomnak többféleképp. Amikor az egészet egyetlen cellában tároltuk, a tömörítés működött, de ahogy több és több cellába osztottuk a genomot, kisebb és kisebb részeket téve a cellákba, a teljes méret nőni kezdett és végül nagyobb lett a forrás file-nál.

Azt is láthatjuk, hogy ZIP-el kicsit több helyet spórolhatunk a kisebb darabokon, mint ZLIB-el.

Az alap tömörítő mechanizmussal a 4,300 KB-os volt az index tábla, de csökkenteni tudtuk a méretet zip-el és zlib-el 2,850 KB-ra.

Az alap PostgreSQL tömörítéssel a teljes 3,205,606 KB emberi genom 1,181,802 KB-on tárolódott. ZLIB-bel tovább csökkent 939,851 KB-ra, zip-el pedig még tovább, 939,835 KB-ra.

11 Eredmények Oracle DB-ben

11.1 Genom eltárolása

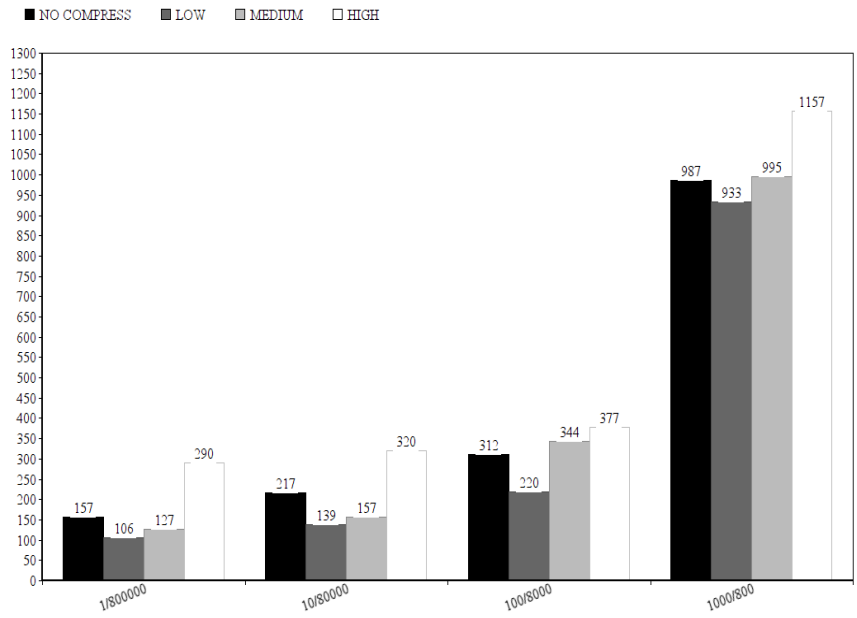

Figure 9: Egy 792 KB genom beolvasása különböző tömörítésekkel és sorokba osztással: x: sorok száma/karakterek száma soronként, y: idő ms-ban

Oracle DB-ben a beolvasás tartogat egy kis meglepetést. Mégpedig azt, hogy tömörítés nélkül több idő beolvasni a genomot, mint alacsony szintű tömörítéssel. Még a közepes tömörítés is képes versenybe szállni a tömörítés nélkülivel. Egyedül a magas tömörítésű beolvasás marad el a tömörítés nélkülitől.

Ennek az oka, az I/O műveletek. Az Oracle DB tömörítési eljárásai úgy látszik olyan gyorsak, hogy behozza a tömörítési időt a kisebb adatok merevlemezre helyezésének ideje.

A kisebb darabokra bontott szöveg több sorba helyezése viszont itt is sokkal lassabb a fordított esetnél.

11.2 Genom feldolgozása

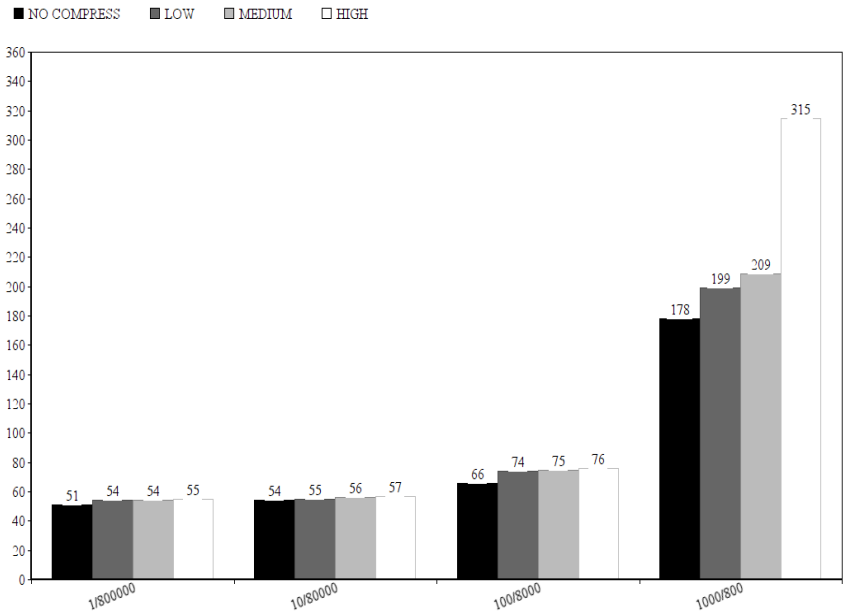

Figure 10: Naive mintaillesztés 792 KB genom különböző sorokba osztásával: x: sorok száma/karakterek száma, y: idő ms-ban

Tömörített genomokon végzett naiv mintaillesztés futási idején megfigyelhetjük amit a PostgreSQL-ben végzett mérések után már sejthettünk is: egyrészt a tömörítettséggel nő a futási idő, ez nyilvánvaló volt eddig is, másrészt a sorok számának növelésével és azzal, hogy egyre kisebb és kisebb darabokat helyezünk a genomból az egyes cellákba egyre nő és nő a futási idő.

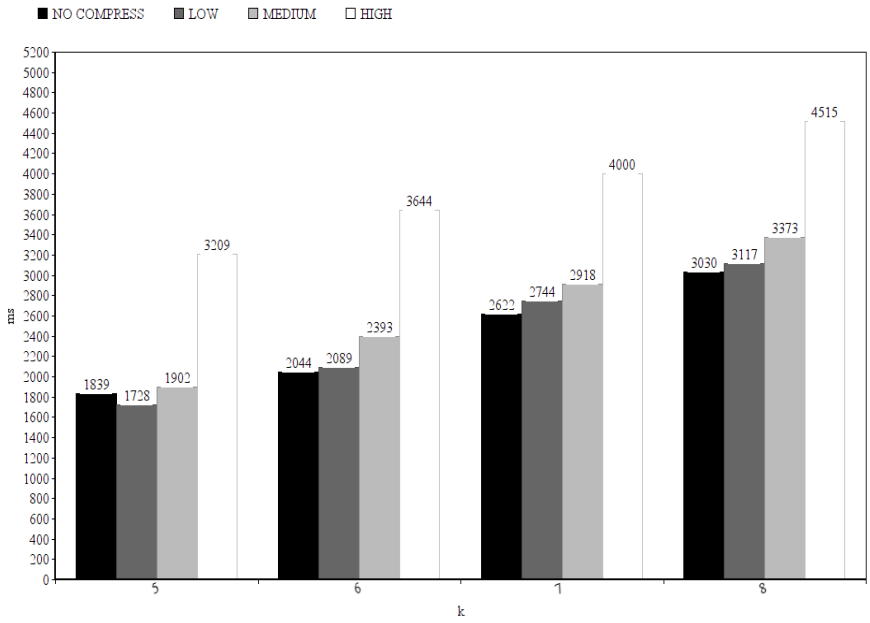

Figure 11: Egy 792 KB genomból index készítése különböző tömörítésekkel és k-as értékekkel: x: k, y: idő ms-ban

Index készítésekor különböző tömörítésekkel már nem ér akkora meglepetés, k=5-re még hasonló a tömörítetlen, az alacsony tömörítésű és közepes tömörítésű index elkészülésének ideje, nagyobb k-ra nőnek a távolságok, magas tömörítéssel pedig mindegyik esetben lényegesen több időbe kerül.

11.3 Terület igény

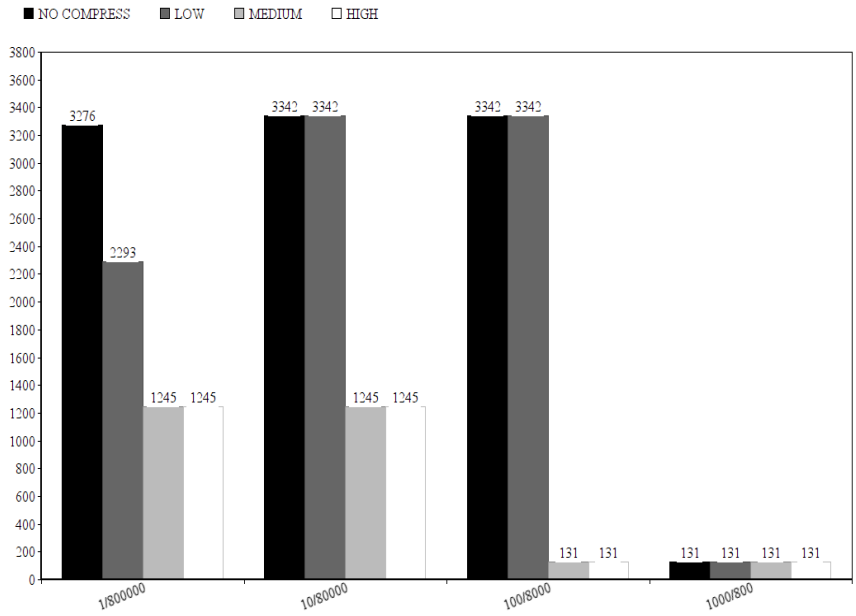

Figure 12: 792 KB genom tömörítése különböző sorokba osztással: x: sorok száma/karakterek száma, y: teljes méret KB-ban

Tárolás tekintetében viszont érdekes módon a PostgreSQL-el ellentétes eredményeket kapunk: Ha egyetlen cellában tároljuk az adatokat, az több helyet emészt fel, mintha szétdaraboljuk őket kisebb részekre.

Ráadásul az itt látható legapróbb részeknél már nem is érvényesül a tömörítés, mégis itt kaptuk a legkisebb méretet, jóval kisebbet mint a forrás fájl mérete.

Érdekes azonban továbbá, hogy nagyobb szeletek esetén rengeteg helyet spórolhatunk meg, viszont általában az összméret nagyobb lesz a forrásfájl méreténél.

12 Eredmények MS SQL-ben

12.1 Genom eltárolása

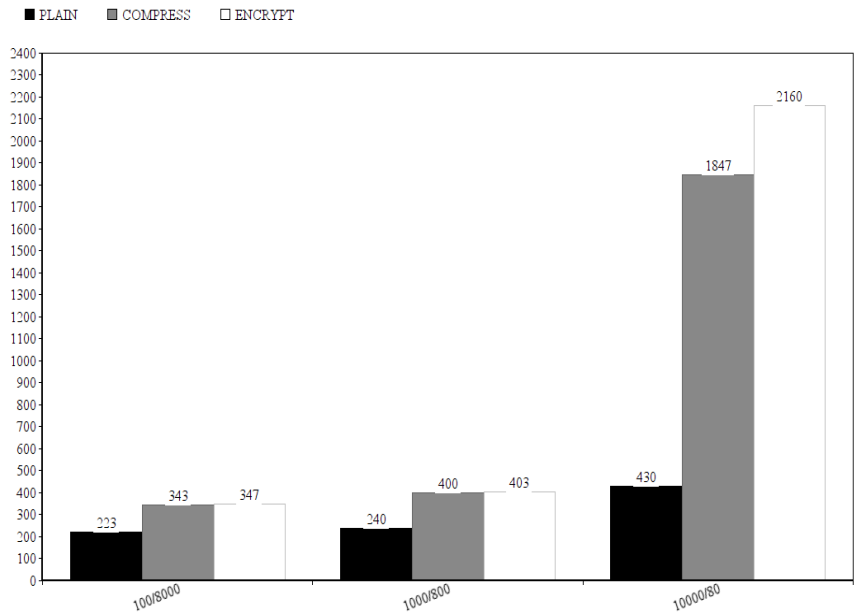

Figure 13: Egy 792 KB genom beolvasása tömörítéssel és titkosítással különböző sorokba osztással: x: sorok száma/karakterek száma soronként, y: idő ms-ban

Microsoft SQL-ben a genom beolvasása hasonló az eddig látottakhoz: nagyobb szeletek beolvasása kevesebb sorba hamarabb megy, mint kisebb szeletek beolvasása több sorba.

Ahogy azt már korábban írtam, MS SQL-ben tömörítés után titkosítok. Ennek tudatában az ábrából az derül ki, hogy a tömörítés után a titkosítás már nem kerül sok időbe, a tömörítés viszont annál inkább.

12.2 Genom feldolgozása

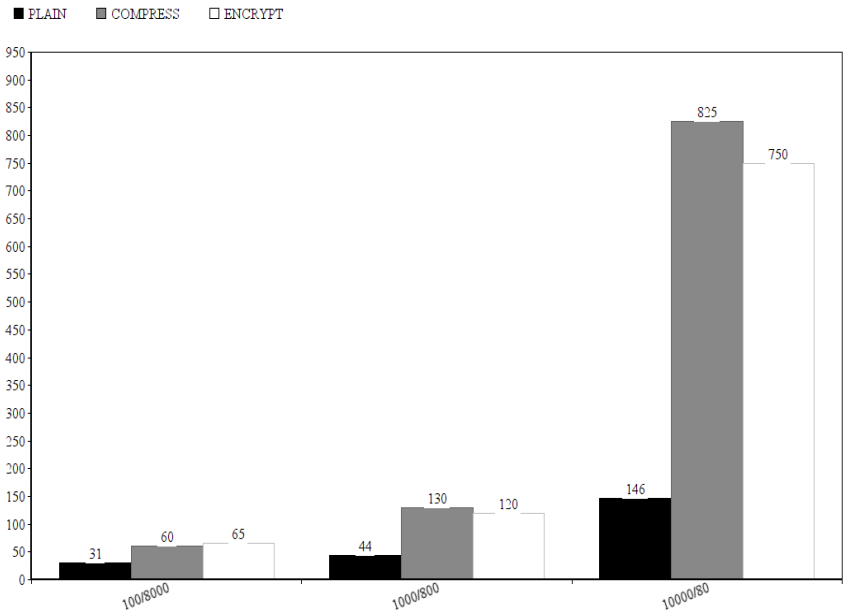

Figure 14: Naive mintaillesztés 792 KB genom különböző sorokba osztásával: x: sorok száma/karakterek száma, y: idő ms-ban

Naiv illesztés során a beolvasási sebességhez hasonló arányokat kapunk, azonban a titkosított adatok itt legyőzik a tömörített adatokon való munkát kisebb, de több cellán.

12.3 Terület igény

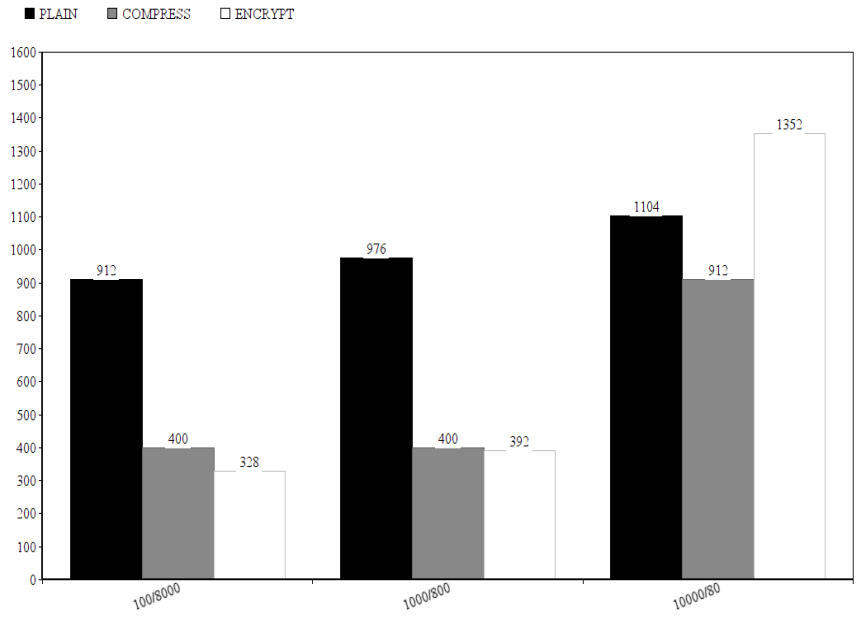

Figure 15: 792 KB genom tömörítése és titkosítása különböző sorokba osztással: x: sorok száma/karakterek száma, y: teljes méret KB-ban

Itt a PostgreSQL-ben látottakhoz hasonlóan, nagy szeletek esetén hatékonyabb tömörítést érhetünk el.

Érdekes továbbá, hogy nagy szeletek esetén a titkosítás kisebbé teszi a tábla össz méretét tömörített táblánál is. kis szeletek esetén viszont túl nő az eredeti tábla méretén is.

Továbbá alapvetően azt látjuk, hogy az eredeti file méreténél nagyobb lesz a tábla tárolási mérete tömörítés nélkül, azonban tömörítéssel is csak nagyobb szeletekkel és kevesebb sorral érhetünk el hatékonyságot.

13 Összefoglalás

Létrejöttek a könyvtárak a vizsgált rendszerekben, melyek segítségével a felhasználó képes lehet jelszóval védett, titkosított és tömörített genomok letárolására, valamint biztonságos feldolgozására.

PostgreSQL-el kapcsolatban elmondható, hogy a vizsgált titkosítások közül az AES tűnik a leghatékonyabbnak és legbiztonságosabbnak is. A tömörítések közül a ZIP bizonyult hatékonyabbnak.

Oracle DB-ben érdemes lehet a leghatékonyabb tömörítési eljárást használni, mivel valószinüleg amúgy se kell minden nap lekérdezni ezeket az adatokat, letárolni is csak egyszer kell őket.

MS SQL-nek nagy hátránya, hogy a titkosított adatokat csak kisebb darabokban tudjuk tárolni, ennek ellenére nem szerepelt rosszul a mérések során. Mivel már csak az AES lesz támogatott, ezt lesz érdemes használni.

13.1 Genom eltárolása

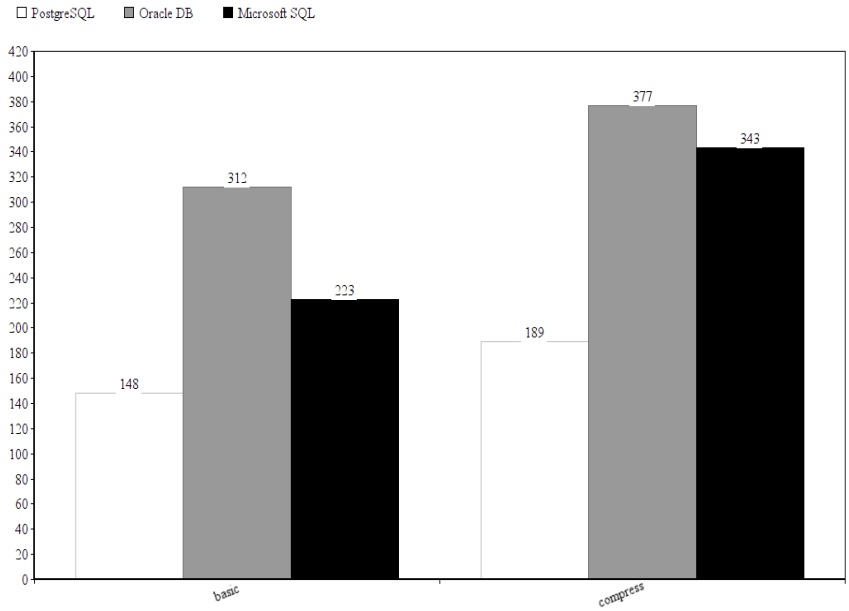

Figure 16: Egy 792 KB genom beolvasása 100 sorba, tömöritéssel: y: idő ms-ban

Összegezve az eredményeket genomok beolvasására, nagy darabok eltárolásában kevés cellába az Oracle DB volt a leggyorsabb, közepes mezőnyben a PostgreSQL, egyedül apró darabok több sorba tárolásában jeleskedett az MS SQL. Az ábrán a közepes darabok esete látható.

13.2 Genom feldolgozása

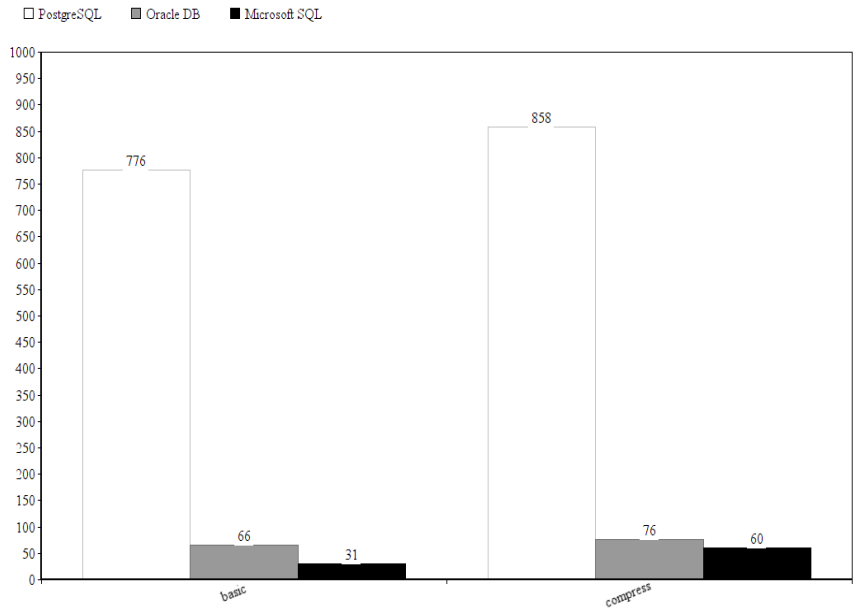

Figure 17: Naive mintaillesztés 792 KB genom különböző sorokba osztásával, y: idő ms-ban

Mintaillesztés során az mondható, hogy bár MS SQL-ben nem tudjuk titkosítani a nagyobb adatszeleteket, kisebb adatszeletekkel, több cellában gyorsabbnak mutatkozott a PostgreSQL-nél és Oracle DB-nél. Oracle DB nem sokkal maradt le.

13.3 Terület igény

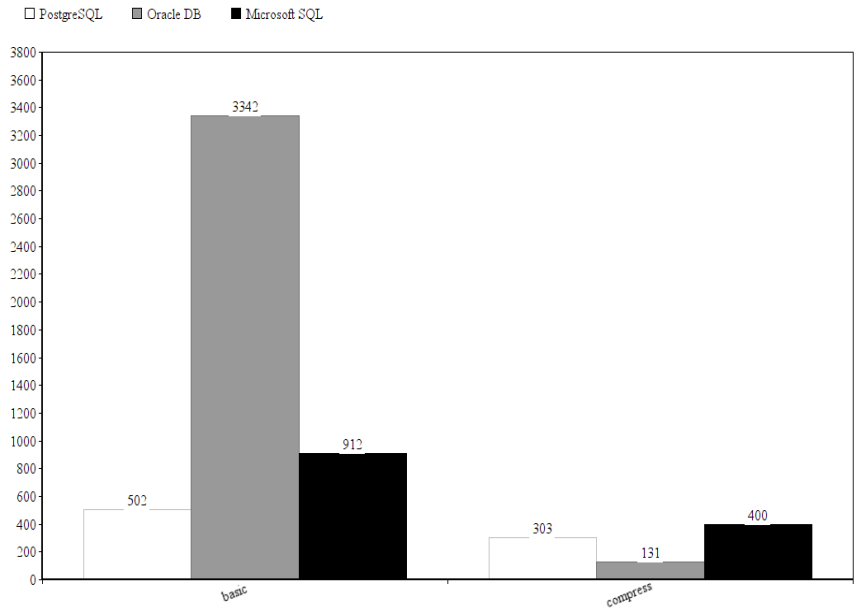

Figure 18: 792 KB genom tömörítése és titkosítása 100 sorba osztva, y: teljes méret KB-ban

Az viszont jobban látszik, hogy a legtöbb felosztásban a PostgreSQL használ kisebb területet a tárolásra a használt technikákkal, legalábbis nagy adatszeletek esetén. Ellenkező esetben az Oracle DB vezet, viszont a mérésekből az derül ki, hogy nem érdemes kis cellákat használni.

14 Jövőbeli munkák

Egyrészt a táblák mérete tovább csökkenthető lenne, ha a genomok 4 karakte 2 biten tárolódna. Ekkor nyilván külön típus kéne a genomoknak és külön típus az aminósav szekvenciáknak, melyeket pedig kb 5 biten lehetne tárolni.

Másrészt érdemes lenne megvizsgálni a bockchain technológia használatát bioinformatikai adatok tárolására. [23], [24]

Fontos lenne még további méréseket végezni többféle hardveren az eredményekről való teljesebb kép alkotásának érdekében.

Végül érdemes lenne feltérképezni bioinformatikai adatok tárolásának lehetőségeit geoinformatikai eszközökkel, térbeli molekulastruktúrák használatával. [6]

15 Köszönetnyilvánítás

A projekt az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósul meg (EFOP-3.6.3-VEKOP-16-2017-00002).

16 Referenciák

[1] Ryan N Lichtenwalter, Katerina Zorina-Lichtenwalter, and Luda Diatchenko. "Genotypic Data in Relational Databases: Efficient Storage and Rapid Retrieval". In: *Advances in Databases and Information Systems*. Springer. 2017, pp. 408–421.

[2] Rodrigo Aniceto et al. "Evaluating the cassandra nosql database approach for genomic data persistency". In: *International journal of genomics* 2015 (2015).

[3] Michael Rice, William Gladstone, and Michael Weir. "Relational databases: a transparent framework for encouraging biology students to think informatically". In: *Cell Biology Education* 3.4 (2004), pp. 241–252.

[4] Simon Jupp et al. "The EBI RDF platform: linked open data for the life sciences". In: *Bioinformatics* 30.9 (2014), pp. 1338–1339.

[5] François Belleau et al. "Bio2RDF: towards a mashup to build bioinformatics knowledge systems". In: *Journal of biomedical informatics* 41.5 (2008), pp. 706–716.

[6] H-J Bockenhauer and Dirk Bongartz. *Algorithmic aspects of bioinformatics*. Springer, 2007.

[7] *EMBL-EBI Bioinformatics for the terrified*. URL: `https://www.ebi.ac.uk/training/online/course/bioinformatics-terrified-0`.

[8] *PostgreSQL documentation*. URL: `https://www.postgresql.org/docs/`.

[9] *Oracle DB documentation*. URL: `https://docs.oracle.com/en/database/`.

[10] *Microsoft SQL documentation*. URL: `https://docs.microsoft.com/en-us/sql/?view=sql-server-2017`.

[11] Goutam Paul and Subhamoy Maitra. *RC4 stream cipher and its variants*. CRC press, 2011.

[12] Andrey Popov. "Prohibiting RC4 cipher suites". In: *Computer Science* 2355 (2015), pp. 152–164.

[13] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.

[14] Andrey Bogdanov, Dmitry Khovratovich, and Christian Rechberger. "Biclique cryptanalysis of the full AES". In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2011, pp. 344–371.

[15] Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography*. CRC press, 2014.

[16] Bruce Schneier. "Description of a new variable-length key, 64-bit block cipher (Blowfish)". In: *International Workshop on Fast Software Encryption*. Springer. 1993, pp. 191–204.

[17] Data Encryption Standard. "Data encryption standard". In: *Federal Information Processing Standards Publication* (1999).

[18] Elaine Barker. "SP 800-67 Rev. 2, Recommendation for Triple Data Encryption Algorithm (TDEA) Block Cipher". In: *NIST special publication* 800 (2017), p. 67.

[19] Carlisle Adams. "The CAST-128 encryption algorithm". In: (1997).

[20] Lars R Knudsen et al. "On the design and security of RC2". In: *International Workshop on Fast Software Encryption*. Springer. 1998, pp. 206–221.

[21] Karthikeyan Bhargavan and Gaëtan Leurent. "On the practical (in-) security of 64-bit block ciphers: Collision attacks on HTTP over TLS and OpenVPN". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2016, pp. 456–467.

[22] Jacob Ziv and Abraham Lempel. "A universal algorithm for sequential data compression". In: *IEEE Transactions on information theory* 23.3 (1977), pp. 337–343.

[23] P Mytis-Gkometh et al. "Notarization of Knowledge Retrieval from Biomedical Repositories Using Blockchain Technology". In: *Precision Medicine Powered by pHealth and Connected Health*. Springer, 2018, pp. 69–73.

[24] Zonyin Shae and Jeffrey JP Tsai. “On the Design of a Blockchain Platform for Clinical Trial and Precision Medicine”. In: *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*. IEEE. 2017, pp. 1972–1980.