Human Genome Data Protection Using PostgreSQL DBMS

Péter Lehotay-Kéry[1](✉) and Attila Kiss[1,2]

[1] Department of Information Systems, Faculty of Informatics, ELTE Eötvös Loránd University, Budapest, Hungary
`lkp@caesar.elte.hu`

[2] J. Selye University, Komárno, Slovakia
`kissae@ujs.sk`

Abstract. There can be a data boom in the near future, due to cheaper methods make possible for everyone to keep their own DNA on their own device or on a central medical cloud. These are sensitive data. There are a lot of cases, when genomes are contained in text files. The size of these can even be 3 GB on every user. Secured data management is not solved in these files.

By using database managers, the levels of permissions can be managed, security and encryption is not the task of the user, because these are integrated into the database manager systems. In this paper, we would like to demonstrate, with the use of an open-source database manager system and with some typical bioinformatical algorithms, that bioinformatical methods can be solved with integrating them into the database manager systems. With efficiency measurements we would like to present, that the use of database manager systems can be efficient in more complex environments.

Keywords: Bioinformatics · Biology · Cryptography · Databases Genetics

1 Storing and Working with DNA

Deoxyribonucleic acid (DNA) is a complex molecule which contains the genetic information. These are built up by nucleotides. Each nucleotide is built up by 3 components: nucleobases (adenin - A, guanin - G, citozin - C, timin - T), a sugar called deoxyribose and a phosphate group.

In bioinformatics, we store DNA sequences as strings, which are composed by the 4 characters for the 4 nucleobases: 'C', 'G', 'A' and 'T'. Similarly, we can store RNA and protein sequence data too.

Typical function on these datasets is string matching, when we are looking for places, where a pattern occurs as a substring of a text, in our case a pattern of nucleotides in a DNA sequence. For example the Boyer-Moore algorithm, when we preprocess our pattern is a good choice on this task. But we can preprocess the genome too, using indexes: we can index with substrings, on suffix trees, on suffix arrays or FM indexes.

Due to sequencing errors and natural variations it's important when we work with DNAs to be able to let a certain number of mismatches when we do matching. For this case we must have approximate matching too. Sometimes it is not enough, so we use sequence alignment, which is a way of arranging the sequences of DNA, RNA, or protein to identify regions of similarity that may be a consequence of functional, structural, or evolutionary relationships between the sequences. For this, we can use dynamic programming.

Other typical function is the sequence assembly, when we align and merge fragments from a longer DNA sequence to reconstruct the original sequence. This is needed as DNA sequencing technology cannot read whole genomes in one go, but rather reads small pieces of between 20 and 30000 bases, depending on the technology used. We can use the shortest common superstring, overlap layout consensus assembly or De Bruijn Graphs for this task [1].

In this paper, we made measurements on naive matching and Hash table indexes.

2 Block Cyphers

PostgreSQL supports some block cypher algorithms. Block cyphers are working with a transformation specified with a symmetric key on fixed-length groups of bits, called block.

A block cipher consists of two paired algorithms, one for encryption E, the other for decryption D, which is the inverse of E. Both will accept 2 inputs: an input block of size n bits and a key of size k bits. Both will give a size n bit output block.

Feistel ciphers split the block of plain text to be encrypted into two equal-sized parts. The round function is applied to one half, then the output is XOR-ed with the other half. After that the two halves are swapped. From the examined ciphers, Blowfish, Triple DES and Cast-5 uses this method [2].

3 Related Works

There are some works already born on the topic of databases for bioinformatics. In [3] an efficient scheme have been presented in PostgreSQL, and compared with other scheme and with file based storage. In [4] measurements have been made on the effectiveness of PostgreSQL compared to Cassandra NoSQL database. In [5] where the authors used MySQL, presented scheme and queries. But none of these say anything about data protection or user hierarchies.

The used and tested bioinformatical algorithms are taken from [1].

When encrypting genomes, We must consider that cyphers with 64-bit blocks are vulnerable to birthday attacks because of the small block sizes [6]. However when attacking the AES, witch uses 128-bit blocks, the key can be recovered [7]. So perhaps there's no algorithm which gives 100% protection, but we should not just store these sensitive information without any encryption. For genome protection we also examined blockchain [8].

4 Results

We implemented the a library for bioinformatics, with processing methods working on encrypted genomes. It can easily be added and used in databases and queries and it can be used for matching, aligning nucleotides and indexing.

Permissions can be set to data accesses and to module accesses and also user hierarchy can be created.

We can crypt our genomes with chosen cipher algorithm.

5 Permissions

In PostgreSQL, we can use roles to manage access privileges. Roles can act as users, groups or both. Roles can own database objects and can manage the permissions on these objects.

Creating, dropping roles and can get the existing roles from the pg_roles catalog:

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

A fresh system always contains a predefined role, which is always superuser and has the same name as the OS user, who initialized the cluster. Be careful, a superuser can bypass any permission checks. As a superuser, we can create more superusers. Being a superuser is a role attribute. The superuser has all these privileges.

Example attributes: superuser, permission to log in, to create databases, roles and setting the password:

```
CREATE ROLE name SUPERUSER
CREATE ROLE name LOGIN
CREATE ROLE name CREATEDB
CREATE ROLE name CREATEROLE
CREATE ROLE name PASSWORD ‘string’
```

These attributes can be modified after the creation of the role using ALTER ROLE. Every object has an owner. Normally the owner is the role that has created the object. By default, only the owner or superuser can do anything with the object. Permissions must be set to change this. There are a lot of privileges: SELECT, INSERT, UPDATE, DELETE, TRUNCATE, REFERENCES, TRIGGER, CREATE, CONNECT, TEMPORARY, EXECUTE, USAGE and ALL for all privileges. Now we won’t discuss all of these. The GRANT command can be used to set permissions. PUBLIC keyword can be used to give the permission to everyone on the system. We can revoke privileges using REVOKE.

Grant and revoke privileges:

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

If we want to use groups, we must first create a role. These usually don't have LOGIN role. Every member of the group can use SET ROLE to temporarily use the privileges of the group and objects created by the user role are considered to be owned by the group. Moreover roles that have the INHERIT attribute, automatically can use the privileges of their groups.

Add and remove roles to the group:

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

In our cases, superusers would be database engineers, and the genome of the users would be stored encrypted, to not let the superuser read them without the symmetric key: the password. The users would be the owners of their own genomes.

6 Genome Crypting

In PostgreSQL, we can use the pgcrypto extension to encrypt genomes. This module provides cryptographic functions, like digest() and encode() computing the binary hash of the given data, or hmac, which works only with a key. For password hashing we can use crypt() and gen_salt() (Fig. 1).

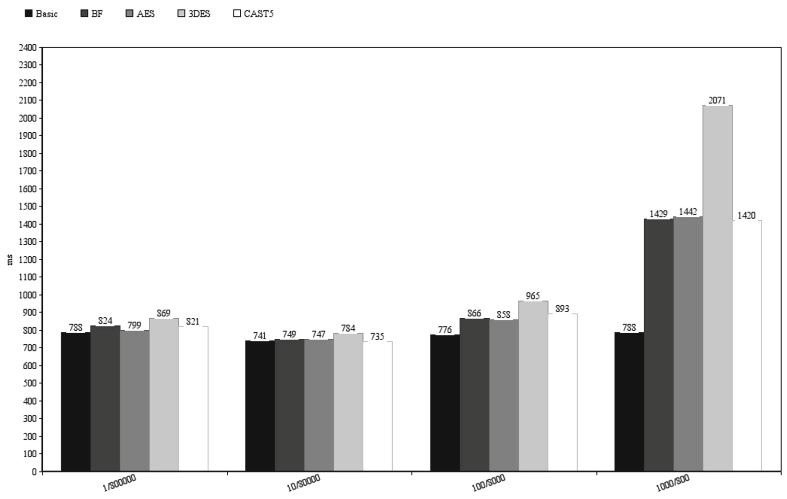

Fig. 1. Naive matching on 792 KB encrypted genome with different divisions into rows: x: number of rows/number of characters in each, y: time in ms

This module supports both symmetric-key and public-key encryption. We are using symmetric encryption, where the user will have a password and the user will store one's genome encrypted with that password. This encryption will be done by pgp_sym_encrypt(), decryption will be done by pgp_sym_decrypt().

When encrypting, we chose cipher algorithm, set it with 'cipher-algo' option, which provides bf, aes128, aes192, aes256, 3des, cast5. Based on the measurements, overall we can say, we will generally have slower algorithms with 3des. The others were similar. Moreover, based on the measurements, with the growing number of rows to decrypt, the distance between the basic and encrypted algorithms will grow. But on one row the encrypted version is not much slower. Usually we won't use much rows.

Blowfish (bf) is a 16-round Feistel network with a 64-bit block size and a variable key length from 32 bits up to 448 bits [9].

AES is based on the Rijndael encryption, used by some international organization like banks. Block size is 128-bit, key size 128-bit for AES128, 192 for AES192 and 256 for AES256 [10].

Although Triple DES (3des) is slower than the others, it gives good protection, used in the electronic payment industry too. It is not a completely new block cypher algorithm, it simple uses three DES keys, K1, K2 and K3, each of 56 bits on 64-bit blocks [11] (Fig. 2).

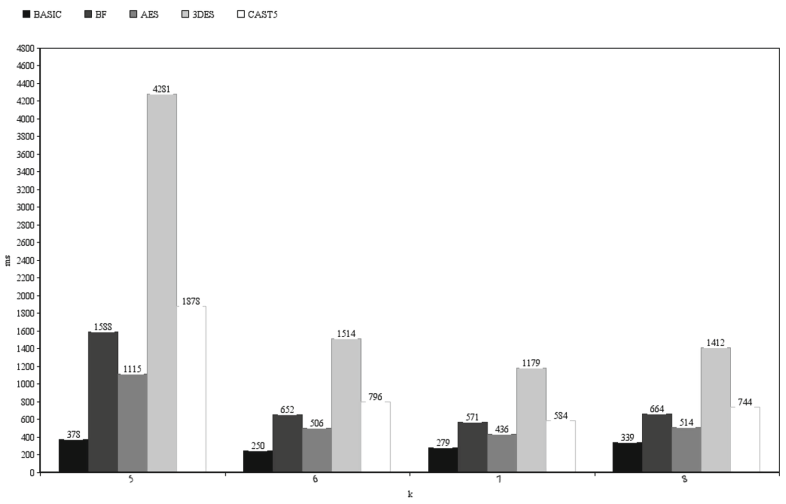

Fig. 2. Querying the index of a 792 KB encrypted genome with different stored kmer lengths: x: kmer, y: time in ms

Cast5 is a 12- or 16-round Feistel network with a 64-bit block size and a key size of between 40 and 128 bits. It is the default cipher in some versions of GPG and PGP [12].

Bad news about encrypted genomes that we tried some measurements using encrypted hash indexes too, in classical hash table style. But it was slower than the naive match because of the lot of rows and the lot of decryption.

We can solve this issue, if we put an index of a whole cell into a single cell. We convert the index into a json string and we encrypt that whole index into one cell.

In the hash index we store every kmer of the genome in sorted order and for every kmer the offsets.

Then when querying the index, we convert back, decrypt and process the index of the cell.

We can see that, our results are depend very much on how we chose k. Without encryption k = 6 is the best, but with encryption k = 7 is the best. Here 3Des is the slowest, too. But differences comes out much more on the cipher algorithms. Now we can clearly see that AES is the fastest, BF is the second, CAST5 is the third.

7 Genome Compression

We can use compression algorithms too, we can set the compress algorithm with 'compress-algo' option, which provides ZIP or ZLIP compressions. They usually have really similar compression result. We can also set the compression level with 'compress-level' option. PostgreSQL has some basic compression mechanisms (Fig. 3).

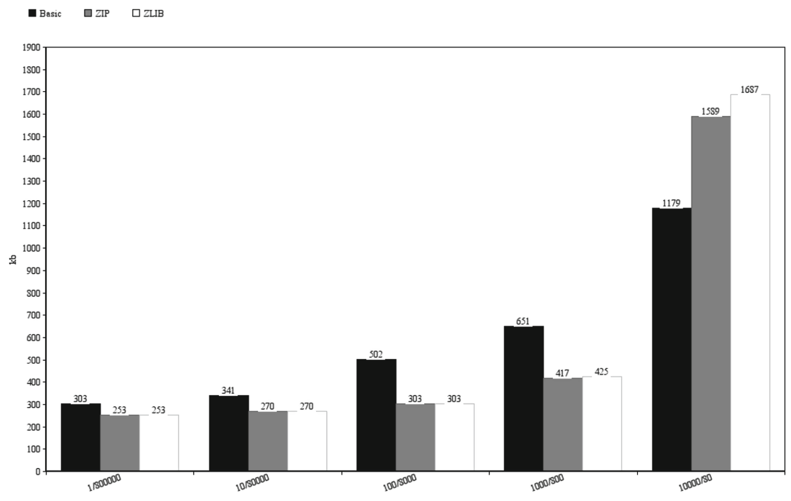

Fig. 3. Compression of 792 KB genome with different divisions into rows: x: number of rows/number of characters in each, y: overall table size in KB

What we can see here, we tried to store a 792 KB excerpt from the human genome in multiple ways. When we stored the whole in a single cell, the compression worked, but as we have divided the genome into more cells, and put smaller parts into the cells, the size grew, and in the end, became bigger than the source file. We can see that too, ZIP worked a little better on small parts, but ZLIB was a little faster in decompression.

With the basic compression mechanism we had the index table on 4,300 KB, but we could make it smaller with zip and zlib to 2,850 KB.

With the basic postresql compression, the whole 3,205,606 KB human genome has been stored on 1,181,802 KB, with ZLIB it has been decreased to 939,851 KB, with zip, it has further decreased to 939,835 KB.

8 Future Work

We are planning to extend the library with further functions, like faster up indexes, and will make it available it for free usage on more platforms. Moreover, we will compare the features of this PostgreSQL solution with solutions in other database manager systems. We will develop an online test user interface too. We will include the usage of blockchain on bioinformatical data.

Acknowledgment. The project was supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002).

References

1. Bockenhauer, H.-J., Bongartz, D.: Algorithmic Aspects of Bioinformatics. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71913-7
2. Katz, J., Lindell, Y.: Introduction to Modern Cryptography. CRC Press, Boca Raton (2014)
3. Lichtenwalter, R.N., Zorina-Lichtenwalter, K., Diatchenko, L.: Genotypic data in relational databases: efficient storage and rapid retrieval. In: Kirikova, M., Nørvåg, K., Papadopoulos, G.A. (eds.) ADBIS 2017. LNCS, vol. 10509, pp. 408–421. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66917-5_27
4. Aniceto, R., Xavier, R., Guimarães, V., Hondo, F., Holanda, M., Walter, M.E., Lifschitz, S.: Evaluating the Cassandra NoSQL database approach for genomic data persistency. Int. J. Genomics **2015** (2015)
5. Rice, M., Gladstone, W., Weir, M.: Relational databases: a transparent framework for encouraging biology students to think informatically. Cell Biol. Educ. **3**(4), 241–252 (2004)
6. Bhargavan, K., Leurent, G.: 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, pp. 456–467. ACM (2016)
7. Bogdanov, A., Khovratovich, D., Rechberger, C.: Biclique cryptanalysis of the full AES. In: Lee, D.H., Wang, X. (eds.) ASIACRYPT 2011. LNCS, vol. 7073, pp. 344–371. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25385-0_19

8. Mytis-Gkometh, P., Drosatos, G., Efraimidis, P.S., Kaldoudi, E.: Notarization of knowledge retrieval from biomedical repositories using blockchain technology. In: Maglaveras, N., Chouvarda, I., de Carvalho, P. (eds.) Precision Medicine Powered by pHealth and Connected Health. IP, vol. 66, pp. 69–73. Springer, Singapore (2018). https://doi.org/10.1007/978-981-10-7419-6_12
9. Schneier, B.: Description of a new variable-length key, 64-bit block cipher (Blowfish). In: Anderson, R. (ed.) FSE 1993. LNCS, vol. 809, pp. 191–204. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58108-1_24
10. Daemen, J., Rijmen, V.: The Design of Rijndael: AES-the Advanced Encryption Standard. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-662-04722-4
11. Barker, E.: SP 800–67 Rev. 2, Recommendation for Triple Data Encryption Algorithm (TDEA) Block Cipher. NIST Special Publication 800:67 (2017)
12. Adams, C.: The cast-128 encryption algorithm (1997)