Assignment report, data integration and interaction

MSc Applied Bioinformatics,

Marie Schmit

Database

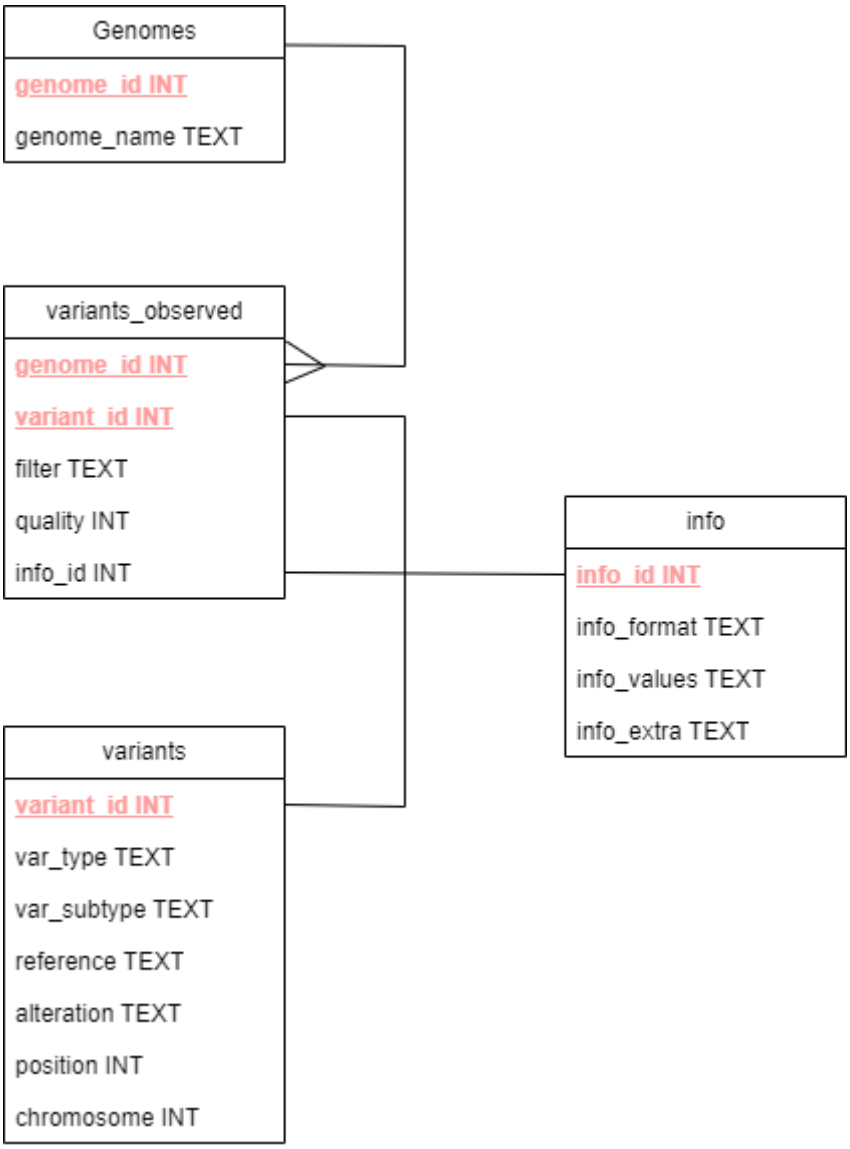


Figure 1 Database schema. Primary keys are in pink and underlined. Tables connect on foreign keys.

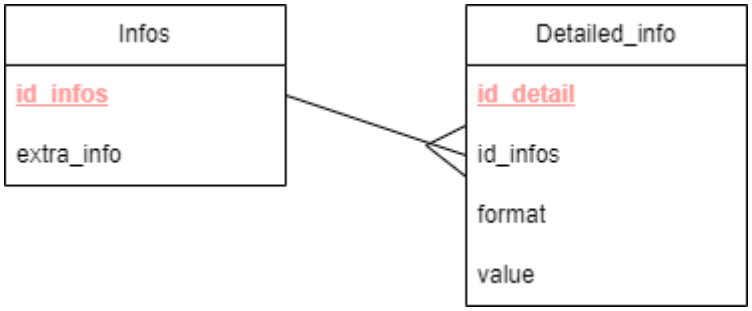
Table genomes contains the genomes id and names. It is connected (one to multiple) to variants_observed thanks to the foreign key genome_id. The primary key of variants_observed is the couple genome identifier, variant identifier. Variant identifier is a foreign key to connect the table to variants table, which contain information about the variant. The table info has the primary key info_id which is a foreign key to variant_observed. Infos contains additional information provided to the variants.

Variants_observed and variants are separated because variant observed contains information that are quite general about both variants and their genome, while variants contain more specific information about the variant. The primary key of variant_observed (genome id and variant id) was meant to allow the assignment of variant to two genomes, in case two genomes had the exact same variant. However, this never happened, and I realised later during the project that the separation of variants in two tables was not necessarily. I could simply have merged both tables.

The information are saved in a separate table because the information field varies across different datafiles. Here, the value of information was always the depth, which is not always the case. So, the

number of elements stored in this table could also change. Also, the format and values were not split because the number and type of element in those fields can change, depending on the vcf information. Here, the format was “GT:GQ:DP” across all the provided documents, which is not always the case. It was not possible to create columns “GT”, “GQ” and “DP” since the format could be for instance “GT:GQ” or “DB:NS:AC:AF:AN”.

A table detailed_infos could have been created, to store individually each format (eg. “GT”) and the corresponding value.



Again, this choice of divided the information in a different table instead of keeping them with the variants table was not necessarily for the rest of the project. The idea was motivated by the possibility of having changing fields of information.

Here, variants of only one specy are considered. In case of a database dealing with many species, a table “ species” could have been added, and link to variant_observed.

Database population

The database is populated with java. When the program is launched, a frame opens to select the vcf file containing data to save in the database. When a file is selected, queries are sent to the database (*java.sql.Connection, java.sql.DriverManager, java.sql.Statement, java.sql.ResultSet*) to get the primary keys of every tables. The are saved in HashSets (*java.util.HashSet*) containing every primary key for each table.

The file is then opened and each line is stored in a buffer (*java.io.BufferedReader*). If the line is a row header, the identifier of the genome is extracted. Lines containing data are parsed: they are split on tabulation. The genome identifier and the data corresponding to each element of the resulting list are sent to the database. For each table, the set of primary keys is checked so only the rows which keys do not exist in the table are added to it. It guarantees the unicity of sql primary keys. For the tables variants and infos, keys are automatically generated by sql, and those generated keys are saved in the appropriate set.

For table variants, the type (either SNP or InDel) is determined by a comparison of the length of reference and alteration values. SNPs have indeed the same length for reference and alteration. The subtype of InDels is “Insertion” if the alteration is longer than the reference. If it is shorter, the subtype is “Deletion”.

Exceptions are managed for each connection with the database (*java.sql.SQLException*), and for the file reading (*java.io.FileReader, java.io.FileNotFound, java.io.IOException*).

Server and first router

The rest API is created with the module “express” of node.js. It listens to port 3000. Two routers are attached to the API, to make those files lighter and easier to read. The first one, router, is deployed on /api and corresponds to router.js file. The second one, routerDensity, is deployed on /apiDens and written in the file routerDensity.js.

For every url, a query is written, as well as a list of parameters. If type and subtype that are optional parameters are given in url, a condition is added in the query for each. They are also added in parameters. Then, the query is sent to the database and its result is displayed.

To return a list of variants located in a specific region of a specific chromosome (url /variants/region/:chromosome/:startPosition/:endPosition/:type?/:subtype?), the information format and value are displayed as a json nested in the json containing all results. Both format and value are split, then each element is added in a json information, with format as key of value. This is intending for a better visualisation.

For the request of variants of minimal depth (url /variants/depth/:genome/:chromosome/:depth/:type?/:subtype?), the value of the depth is extracted and compared in the sql query. This value is saved in table infos, element extra_info as “DP=value”. A substring of the value that follow the string “DP=” is extracted. This way, the result only gives a result for DP information, even if the field in extra_info is not DP. Then, this substring is cast to a number and compared to the requested depth.

Second router, density calculation and plot

An error message is sent if the window size given by the user is empty. A query requesting the maximal position of variants from the given genomes and chromosome is sent to the database. This value is used in a function **variantDensity** to calculate the maximal number of windows of the given size into which the positions can be divided. Then, the density is calculated for each of those windows: a loop goes through the number of windows, the start and end positions of the window are calculated and the function **calculateDensity** returns the density for this window. This density is pushed into an array, that is saved in a json with **jsonResult**. The resulting json contains the genome, chromosome, window size, type and subtype, as well as another json windows into which the density, the end and start position of the window are saved. This is the displayed json.

The function **calculateDensity** writes a query request asking for the count of the variants whose position is between the start and end of the windows. This request is sent to the database in the function **getCounts**, returning the number of variants for the given window.

The functions **calculateDensity**, **getCounts** and **variantDensity** are asynchronous. When they are called, the functions "**await**" forces the system to wait for their results before executing next instructions. It allows to assign their results to variables. Without it, the system does not wait for the result of the callback function of "**Database**.**all**". This result is assigned to a value before the response of the database: it is thus empty. "**Promise**" is used in **getCounts** to send the result of the database query out of the callback function of **db.get**. "Promise" is also used to get the result of **calculateDensity** in the router get function. This "get" function is not asynchronous: the use of "then" makes it wait for the result of **calculateDensity** before doing the rest of the script.

I have chosen to use JavaScript to calculate the density of the variants for two reasons. The first one is to get better at this new language. The second is to optimise the code. I thought that sending multiple little requests to the database to calculate directly the count of variants in a given window, with sql, was lighter. No data must be copied from the database, transferred to another script in R or Python, and gone through. Indeed, another solution would have been to request all positions to the database, to use them as an array. Then, for each window, to go through this array and update a counter for the positions falling in the given window. In that case, the array should have been scanned for each window, increasing the complexity.

I do not know the complexity of a sql request, but for the javascript method, the actions requiring complexity are:

- One sql query
- One foor loop on windows with one sql query each: O(n) complexity for n windows (plus sql complexity)

For Python or R script:

- Sql query to pass all the positions from the database to another script
- One loop on windows with sql: O(n) complexity for n windows
- Inside, one loop for positions: O(n*m) complexity for n windows, m positions, knowing that the number of positions is very big.

This approach is less interesting if the complexity of sql requests to the database is very large. I assumed that it was not.

Plumber in R

In R, the library plumber creates another API on port 3001, that gets for a given url the arrays of density and of windows start positions calculated in the router **routerDensity**. They are stored in a dataframe and plotted on the browser as image (the serializer is png) thanks to the library ggplot2.