Technical Documentation

Obiba GenoByte

Génome Québec

Version 1.0 - 2007-09-26

## Table of Contents

# Introduction

GenoByte is a generic API for manipulating large collections of genotypes. Typically, it is used as the basis of a genotyping application. It started as an internal project at the McGill University and Génome Québec Innovation Center to counter the limitations imposed by the use of traditional RDBMS (Relation Database Management System) such as MySQL.

As genotyping technologies evolve, experiments allow to test a growing number of variants on a growing number of patients, thus generating exponentially bigger datasets. Using MySQL as an RDBMS back-end, our genotyping application quickly became overwhelmed by the requirements of simple statistic calculations such as the MAF and Hardy-Weinberg values. It became even worse with more complex calculations such as case-control studies. A replacement solution was therefore necessary.

A column-oriented database solution has proven to be the desired method. Short of finding an acceptable free software solution, we developed one ourselves, adapted to the needs of genotyping applications. The solution has proven to significantly improve the efficiency of our in-house database system. Therefore, we decided to release the GenoByte API as open-source software under the umbrella of Génome Québec's open-source initiative aimed at biobanks, Obiba.

While fitting the needs of the genotyping portion of biobanks software applications, GenoByte can go over that realm, assisting in the development of any Java application manipulating large amounts of genotypes.

This document is aimed at describing the behaviour of the GenoByte API and its components. We will first define some general concepts in use. Then, we will explore the “nuts and bolts” of the system, explaining how it works and giving cues on how it can be extended. Finally, we will explore the functionality of the main application modules.

# General concepts definition

## *Row-Oriented VS Column-Oriented Databases*

This section introduces the data organization strategy involved in the GenoByte API, and compares it with conventional methods.

In traditional RDBMS, data is organized in records. Database systems using this methodology are known as "row-oriented" databases. The organization in row means that all information relevant to one instance (one entry) is regrouped, and called a record. For example, imagine a "user" table in an RDBMS, with three fields to be stored: id, name and email. In this case, all data related to a single user is regrouped. We have a triple (id, name, email) for the first user, another triple for the second user, etc.

| | |
|---|---|
| **Record 1** | 4, "John Smith", "smith.john@obiba.org" |
| **Record 2** | 10, "Bill Wong", "wong.bill@obiba.org" |
| **Record 3** | 12, "Anna Ban", "ban.anna@obiba.org" |

**Table 1**: Data organized in a row-oriented fashion

This makes it easy to append new information because everything that is relevant to one record is regrouped. Adding a new record simply requires adding a new triple.

Another type of databases which has gained visibility in the recent years is the so-called "column-oriented" database. The basic idea is to regroup data by field, rather than by record. A field holds one information type for all records. Column-oriented databases is not a new concept, being described in papers such as TAXIR[1] in 1969.

We could imagine a field as being a huge vector, containing values of its type for all database records. Another field of the same table contains values of its type for all records. Data pertaining to the same instance is always located at the same index in each Field.

1 G.F. Estabrook and R.C. Brill, The Theory of the TAXIR Accessioner, Mathematical Biosciences 5 (1969), 327-340

Taking our previous “user” table example, adapted to the column-oriented concept, the id information for all users becomes regrouped in one field. So are the users name and the email. That's where the term “column-oriented” comes from, because the data is organized by columns (or if you prefer, by information type). Because field information is stored in consecutive order, column-oriented databases are efficient at executing queries on the dataset.

| | |
|---|---|
| **id** | 4, 10, 12 |
| **name** | “John Smith”, “Bill Wong”, “Anna Ban” |
| **email** | “smith.john@obiba.org”, “wong.bill@obiba.org”, “ban.anna@obiba.org” |

**Table 2**: Data organized in a column-oriented fashion

The end result is that column-oriented databases are optimized for reading, by running querying on a per-field basis. On the other hand, row-oriented databases are optimized for writing, when appending new records or modifying existing records. This explains why column-oriented databases are more appropriate in the case of genotype statistical analysis. As post-experiment genotyping datasets are generated only once, they will not likely be modified after being loaded in the database. They will however be queried many times, either to read data or to run statistical analysis with changing parameters. Therefore, organizing data in a column-oriented way represents an interesting optimization.

For a more complete definition of column-oriented, see “http://en.wikipedia.org/wiki/Column-oriented_database”

## *Berkeley DB*

Berkeley DB (commonly called BDB), is a database library with an API in various programming languages. It is used to store and access data manipulated inside an application, using a local hard disk files to store data. It is multi threaded and supports database transactions. Moreover, implementations exist in multiple operating systems, and there is a specific implementation written purely in Java.

We opted to use BDB in GenoByte because we needed an efficient storage system that could easily be embedded in our Java application. Traditional Relational Database Systems such as MySQL were not answering our needs, being too slow to store/read huge columns of data from and to the GenoByte API code. The plain text file approach wasn't appropriate either, because we wanted the API to make use of database transactions. With such an approach, we would have had to reimplement those concepts ourselves.

For more information on BDB, please visit the Oracle Berkeley DB Website at “http://www.oracle.com/technology/products/berkeley-db/index.html”

# Nuts and bolts of GenoByte

The following section will explain the functionality of the GenoByte API, by exploring its components and the interaction between them.

## *What is the Bitwise?*

The Bitwise is our implementation of a column-oriented database table, in the shape of a Java API. All data manipulation, including CRUD operations (create-read-update-delete), are done through this API. As such, the Bitwise API is the foundation for GenoByte, managing all data storage functionalities. By default, it uses BDB as a support for data persistence, but it is possible to provide another implementation of this portion of the API to use other kinds of data persistence.

The Bitwise uses “stores” to organize data, a concept similar to RDBMS tables. A store is usually mapped on a Java class, whose relevant properties become fields. All data stored in the Bitwise is first transformed into bits to increase query efficiency on large datasets. The transformation of original object instances or primitive values into a series of bits, and from bits back to an original value is done using dictionaries.

### Dictionary

Dictionaries can be seen as the interpreters in the Bitwise, as they are the ones who know how to convert original values into vectors of bits (encoding with the `lookup()` method), and vectors of bits back to original values (decoding with the `reverseLookup()` method). There is a set of default dictionaries provided with the Bitwise API for all primitive types and a variety of classes. These are designed to encode values with the smallest possible number of bits.

On the other hand, it is possible to extend the API with custom dictionaries to handle value classes which are not supported by default. To do so, it is only required that the new dictionary implements the `Dictionary` interface, which defines the basic set of `lookup`/`reverseLookup` methods.

Some dictionary implementations are parametrizable, to allow the use of that implementation in different contexts. For example, the `IntegerDictionary` provided with the API can be parametrized to specify value boundaries and an increment, allowing maximum compression of the data. In this case, if a programmer already knows that an attribute's integer value will always be even numbers between 0 and 20, he can set a lower bound of 0, a higher bound of 20 andan increment of 2.

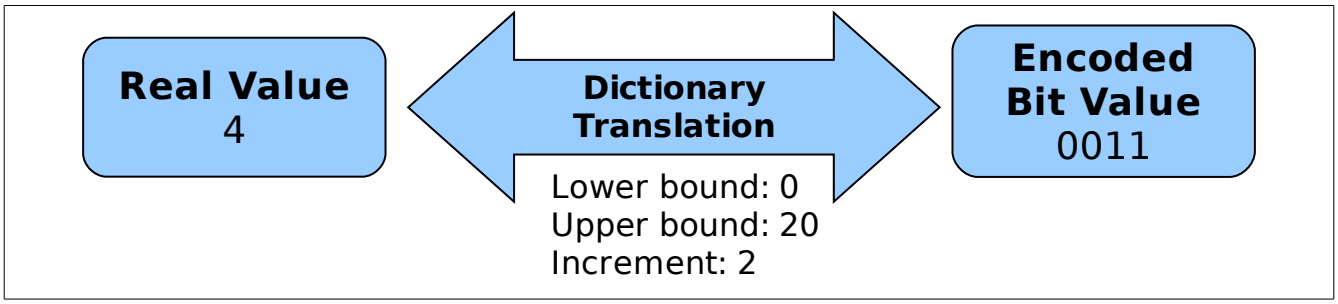


**Figure 1**: An integer is encoded into a value in bits, and back to the original value using a Dictionary.

Figure 1 shows illustrates this example. An integer dictionary is being parametrized with a lower bound of 0, a higher bound of 20 and an increment of 2. There is therefore a total of 11 possible values, which can be encoded into 4 bits. The number "4" is the the third possible value, starting from zero, hence it's bit value of "011".

Most dictionaries will encode values using a fixed number of bits. There is a another type of dictionary which uses a End-of-Bit-String character. The unused bits are all set to zeros. The default dictionary for `Strings` uses such a dictionary. The encoded bit strings are generated using an Huffman code algorithm, making the resulting number of bits dependant on which characters are used in a value.

The string of bits which is the result of an encoding operation is stored in a `BitVector`, the next concept that we will explain.

### BitVector

A `BitVector` is a vector of bits (zeros and ones), the basic unit of information with which data is manipulated in the Bitwise. It is very close to the implementation of the standard Java class "`java.util.BitSet`", with some extra operations useful to this API.

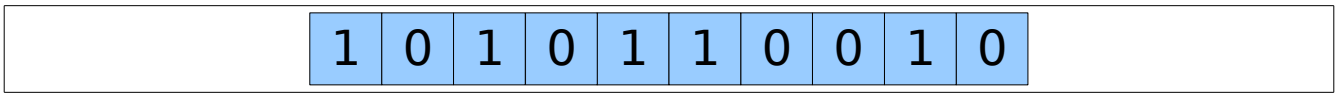


**Figure 2**: A BitVector of 10 cells.

### Field

A `Field` represents an information to be stored in a Bitwise store. In the Java world, it is a class attribute (also called a class field), represented by a single class/primitive (e.g. an `int`, an `Integer`, a `char`, a `String`, etc.). As we are organizing data in a column-oriented fashion, a `Field` instance will hold the values of all records for that attribute. It can be seen as a single column in an RDBMS table, only that the value's class is not limited to a specific set of supported types (as in MySQL). Instead, it can be

used to hold any Java class/primitive value. For attribute classes not provided by default with the API, a programmer simply needs to implement a dictionary, and plug it in the Bitwise. This dictionary will tell the Bitwise how to encode and decode data between an original value and a bit vector value.

A `Field` consists of multiple `BitVectors`. Each `BitVector` holds one bit of data for all store's records. A Field value for a record at index i is made up of the bits found at index `i` in all `BitVectors` belonging to the field. The number of bits required to store all values is called a *dimension*. Note than in a `Field`, a record for which all `BitVector` values are 0 is always considered as being set to the null value.

As an example, in Figure 2, the `Field` values are held on four bits, and there are ten records in the store. The `Field` value, in bits, of the fourth record is 0100.

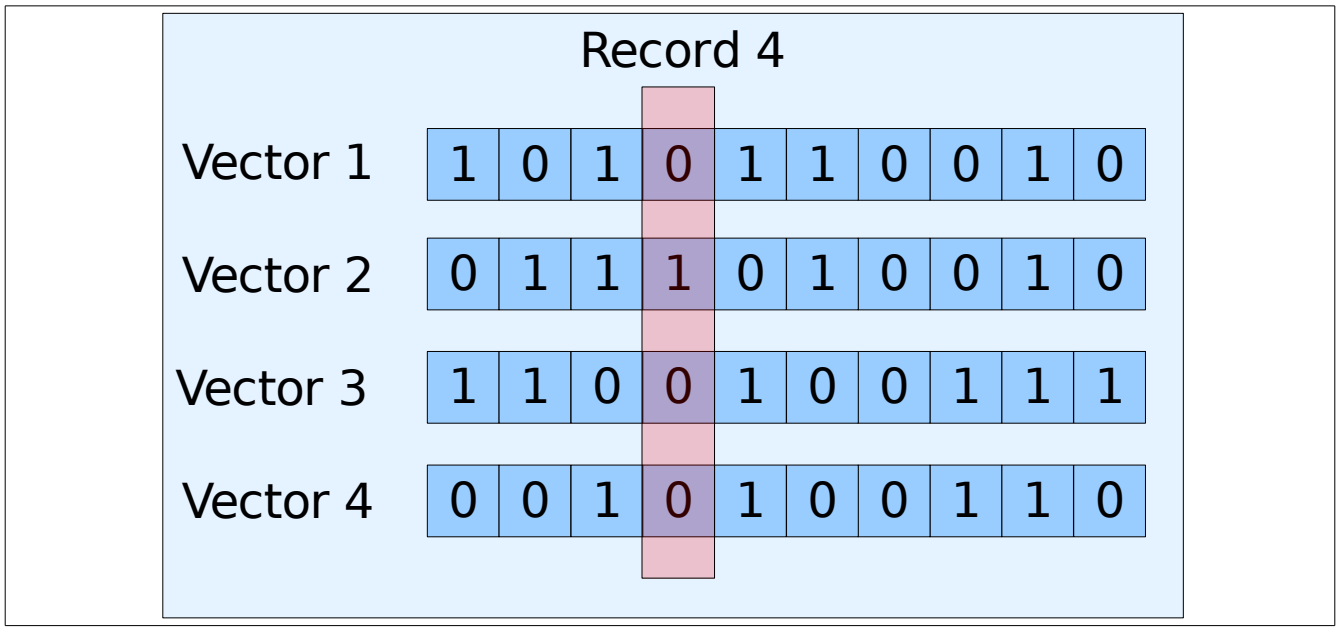


**Figure 3**: A `Field`. The purple region identifies record #4's value for this field.

As we have already seen, the data stored into `Fields` is first transformed into vectors of bits. These values transformation allows to organize the data in a convenient way inside the store. For example, a certain data type could be encoded to fit in the smallest possible vector of bits for memory efficiency purposes. On the other hand, another data type could be encoded to optimize querying on that `Field`.

Each `Field` is assigned a `Dictionary` implementation, to encode/decode data to/from the `Field` in a stable way. The mapping between a `Field` and a `Dictionary` can be defined in two ways: with an XML schema and using Java Annotations. In both methods, the `Dictionary-Field` binding will also provide parameters required by a dictionary implementation, if needed.

### Record

A record keeps the same meaning as in any RDBMS, being a set of values for one specific database entry. In the Bitwise, a record is always positioned at the same index value in all `BitVectors` of all `Fields`. This is the way by which the connection is made between all `Fields` and `BitVectors` for a given record. Accessing a record in the bitwise actually means reading or writing on Fields by column.

### BitwiseStore

A `BitwiseStore` can be seen as a table in a traditional RDBMS. It regroups all fields belonging to the table. In each store, there is always exactly one field that acts as the primary key, allowing to uniquely identify each record. This means that there shall be no two records with the same value for that field.

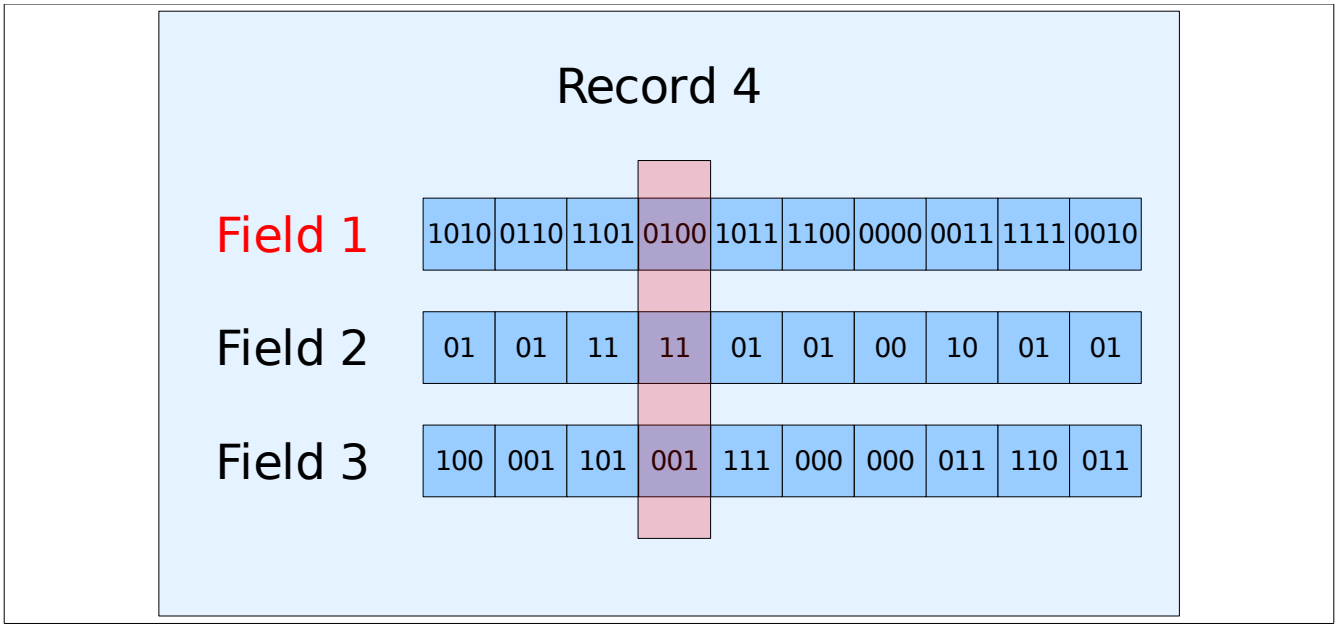


**Figure 4**: A `BitwiseStore` with 3 fields. Field 1 is the unique key to the store. The purple region identifies the record #4.

A `BitwiseStore` is usually mapped on a Java class, whose attributes define the fields of the store. This mapping is doing using a set of special Java annotations for GenoByte. Figure 5 shows a `BitwiseStore` built on a User class made of three attributes: “id”, “name” and “email”. To get the three values that belong to the sixth record, we shall get the values at index #5 (vectors are zero-based) from each Field.

The `BitwiseRecordManager` becomes useful to manipulate a record as a whole using the `BitwiseStore` defining class.

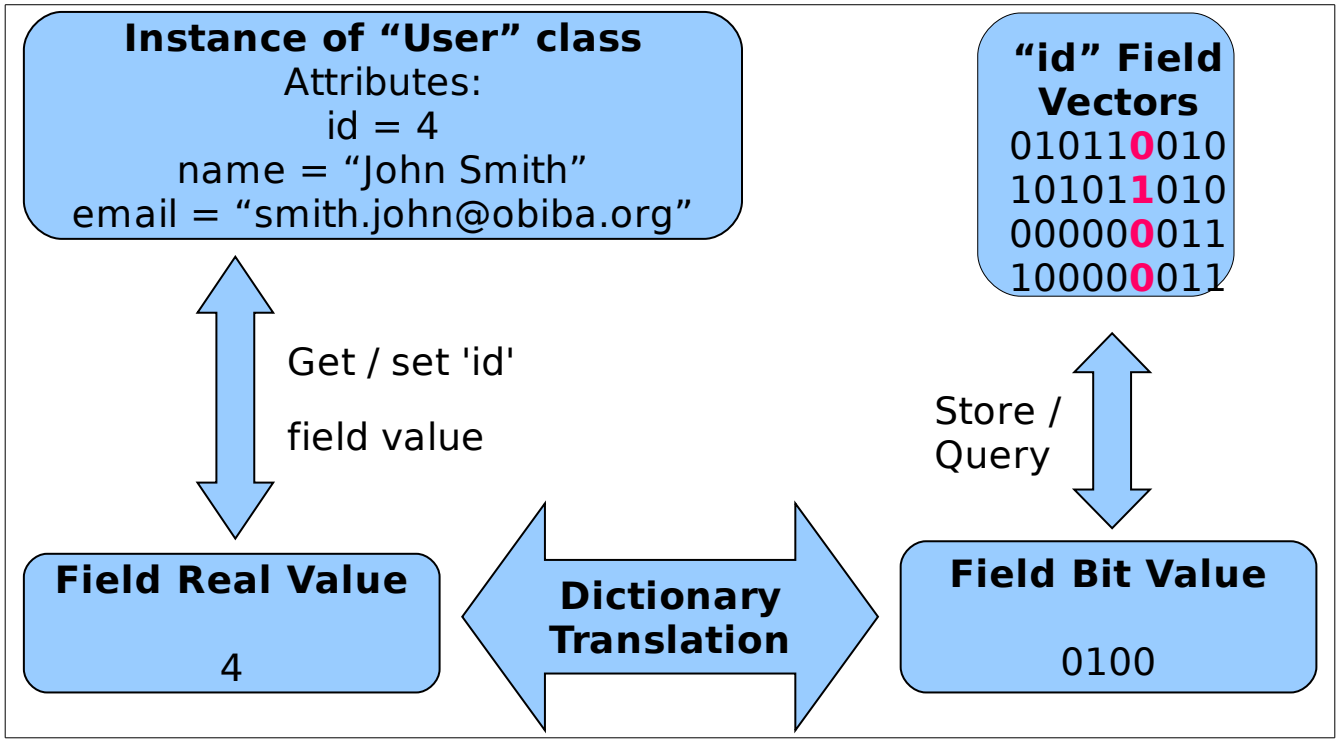


**Figure 5**: A `BitwiseStore` uses a `Dictionary` to transfer data from an object's attribute value (here an `Integer`) into a `BitVector`, and then back to an `Integer`.

### BitwiseRecordManager

The `BitwiseRecordManager` provides the necessary methods to handle records as a whole in a store, using column-based access. It allows such things as getting the unique key for a record from its index and vice versa, and records create / update / delete operations. Using the `BitwiseRecordManager` is optional, but makes it easier to handle a `BitwiseStore` as a store of Java objects.

### Last word on the Bitwise

As you might have already realized, the Bitwise portion of the GenoByte API could be considered an API by itself, that could be used to store any kind of Java class and not only genotype-related data. It is possible in the future to see the Bitwise portion separated as a standalone API, would we need to use it elsewhere in the Obiba project.

## *What is GenoByte?*

GenoByte is an extension of the Bitwise API aimed at developing genotyping applications. The main component is the `GenotypingStore`, which is made to hold genotypes for a matrix of assays/samples. It uses two `BitwiseStores` to store its information in a column-oriented fashion for both assays and samples. It is important to understand that **both** BitwiseStores contain the genotypes.

As a result the dataset shouldn't be seen as two separated matrices but rather as a single big matrix, on which we either operate on lines of samples and columns of assays or vice versa.

### GenotypingStore

A genotyping store is composed of two `BitwiseStores`, one for assays and one for samples. The genotypes are contained in those two stores. The reason for that is to allow quick computation of statistics, by assay and by sample. To keep the querying efficiency, coming from the data being organized by columns, the double-matrix system became a necessary evil.

Figure 5 shows a simplified version of a `GenotypingStore`. The assay store holds the assay-relevant `Fields`, such as the assay id, and one more `Field` per sample genotyped in the experiment. It is used to run genotype queries using assay criteria. On the other side, the samples store possesses one `Field` per assay. The assay store genotypes for record #1 (Assay 1) corresponds to the genotypes in the sample store for the `Field` (row) entitled "Assay 1".

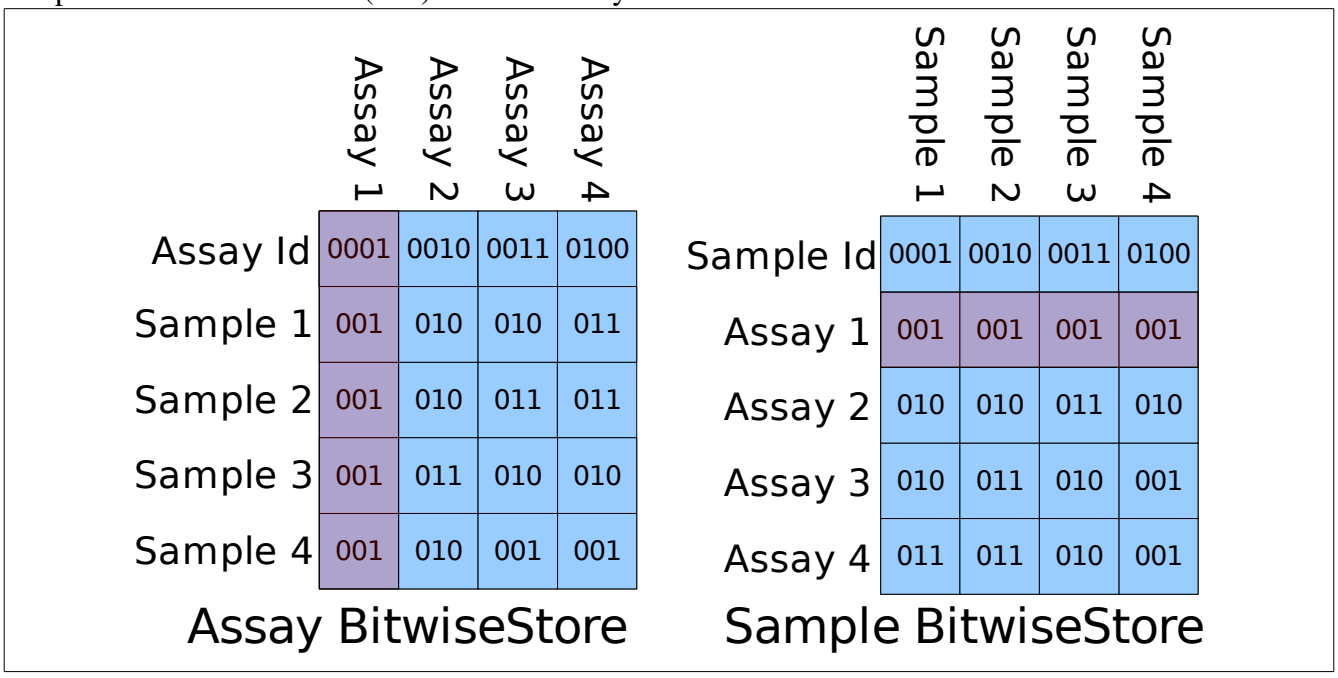


**Figure 6**: The two matrices composing a `GenotypingStore`. The purple area in the sample store represents the genotypes for Assay 1.

# GenoByte modules

## *Queries*

As it has been mentioned previously, queries in a column-oriented database are quicker than in their row-oriented RDBMS counterpart because when querying a given field, all that field's data is already regrouped. This allows to run a field query on all records at the same time with binary operators, which are very fast.

The following example will give you an idea of how this works in practice. Imagine a genotyping experiment for which we test a sample on five different assays. In the `GenotypingStore`, we have a field called "genotype" for which we have four possible values beside null: "A" if the individual is an homozygote of the first possible allele, "B" if the individual is an homozygote of the second allele, "H" if he is an heterozygote and "U" if the genotype is unknown. Encoded into bits, we get the following scheme:

| Genotype Call | Value in bits |
|---|---|
| Null value | 000 |
| Homozygote for allele A | 001 |
| Homozygote for allele B | 010 |
| Heterozygote | 011 |
| Unknown genotype | 100 |

**Table 3**: Hypothetical bit values for genotype calls.

All genotype values are being held on three bits. This means that the `Field` containing the genotype calls can have a dimension of 3 (using three `BitVectors`). Now imagine that our tested sample gives the following genotypes:

| Record | Genotype Call | Value in bits |
|---|---|---|
| Variant 1 | Homozygote for allele A | 001 |
| Variant 2 | Heterozygote | 011 |
| Variant 3 | Homozygote for allele B | 010 |
| Variant 4 | Heterozygote | 011 |
| Variant 5 | Unknown genotype | 100 |

**Table 4**: A set of five genotypes obtained from five assays tested on one sample.

Organizing the information in a column-oriented fashion, we split the genotype calls into the three `BitVectors` composing the "genotype" `Field`, and obtain the following matrix:

| | Assay 1 | Assay 2 | Assay 3 | Assay 4 | Assay 5 |
|---|---|---|---|---|---|
| Call BitVector 1 | 0 | 0 | 0 | 0 | 1 |
| Call BitVector 2 | 0 | 1 | 1 | 1 | 0 |
| Call BitVector 3 | 1 | 1 | 0 | 1 | 0 |

**Table 5**: Genotype calls divided in three `BitVectors`.

Now suppose that we want to know which records are the variants for which the sample is heterozygote. As we can see in Table 3, the `BitVector` value of heterozygote genotypes is "011". Having a total of 3 bit vectors to evaluate, we need to execute 3 binary operations. We can express our "011" query with the following logical expression, where BV is a `BitVector` of the queried field:

$$\neg(BV1) \wedge BV2 \wedge BV3$$

Queries generate a `QueryResult` instance, which is essentially a wrapped `BitVector` with the size of the number of records in the store. This means that the result set `BitVector` will be as long as any `BitVector` composing a `Field`. That vector will indicate which records are matching the query.

Table 6 shows that using bitwise operations, the `QueryResult` vector narrows down to the records matching our query:

1. By looking at the first bit of our "heterozygote" query, we can see that the first `BitVector` for the genotype call `Field` should be a zero. It's already possible to realize that "Assay 5" does not match our query. Consequently, the first binary operation, which consists on retaining all records with a "0" bit, eliminates the fifth record.

2. We can then eliminate the "Assay 1" record by looking at the second `Field`'s `BitVector`, knowing that our heterozygote value holds a one as the second bit.

3. The last bit comparison eliminates the record "Assay 3".

| Term | Bitwise Operation | Vector | **Assay 1** | **Assay 2** | **Assay 3** | **Assay 4** | **Assay 5** |
|---|---|---|---|---|---|---|---|
| $\neg(BV1)$ | NOT | Call BitVector 1 | 1 | 1 | 1 | 1 | **0** |
| $\wedge BV2$ | AND | Call BitVector 2 | **0** | 1 | 1 | 1 | 0 |
| $\wedge BV3$ | AND | Call BitVector 3 | 1 | 1 | **0** | 1 | 0 |
| | | **Result Vector** | **0** | **1** | **0** | **1** | **0** |

**Table 6**: Querying the Field for all five records at a time using binary operators.

In our example, the resulting `BitVector` "01010" indicates that both the second and the fourth record are matching the "heterozygote" criterion on the assay table.

Note that with traditional row-based RDBMS, this query would have needed to be ran on each record separately, or at least require the creation of indexes for optimization.

In short, this example shows the underlying process of querying a single field for a single value. Real-world queries can be much more complex, involving multiple fields and ranges of values for each field. Expressing such complex queries simply becomes a matter of combining single-field queries, using single values or ranges, with boolean operators, such as AND, OR, NOT, etc. Any query executed on a `BitwiseStore` is always executed on every record simultaneously using bitwise operators.

## *Statistics*

GenoByte includes a statistical calculation module that can be used on a `GenotypingStore` dataset. A statistic can be any kind of computation done with the data available in a store. It is also possible to use the result of statistic calculation as an input to the calculation of other statistics. The Statistics module works with the following four components:

- **Statistic interface**: An implementation of the Statistic interface knows how to use data from a `GenotypingStore`'s fields and data generated by other Statistic implementation, to calculate a statistical result. It can do any kind of data processing to produce an output. Typical basic examples that are implemented in GenoByte are the allelic frequencies, the heterozygosity, the minor allele frequency, and the Hardy-Weinberg coefficient. There are two Statistic subinterfaces: `FieldStatistic` to run a calculation on a whole Field at the same time, and `RecordStatistic`, working on a single record of the `GenotypingStore`.

- **StatsRunDefinition**: A class of this type allows to regroup statistics to be computed together in batches. It is a kind of "grocery list" of statistics that should be computed when needed. The

statistics module will take that list to determine everything that needs to be calculated.

- **StatsPool**: This is the engine of the statistics module, as it coordinates the calculation of everything listed in the `StatsRunDefinition` and remembers the results between each step.

- **StatsDigester**: A digester is class that knows what to do with results once they have been computed. It is external to the statistic calculation process. Once all statistics which are part of a `StatsRunDefinition` have been calculated, the digester will take the result set from the `StatsPool` and execute a series of actions. For example, a certain digester can be used to persist data in a database or BitwiseStore, to generate reports, etc.

It is possible to run series of calculation on a whole `GenotypingStore` dataset, or on a subset. By defining the proper record masks, it becomes possible to eliminate unwanted assays and/or samples, therefore defining the subset. This fact can be observed in the GenoByte Infinium example, where running queries on either the samples or the assays matrix will restrain the dataset used to generate reports.

### How inconsistencies statistics are handled in the system

The inconsistencies detection has its own module in GenoByte. The package "`org.obiba.genobyte.inconsistency`" contains all the necessary classes to run the inconsistencies discovery methods. Out of the box, GenoByte provides the ability to identify three types of inconsistencies: DNA reproducibility errors, SNP reproducibility errors and Mendelian errors. The DNA and SNP reproducibility use the same logical engine, but act on a different dataset (samples or assays respectively). Although GenoByte does not impose a data structure in the sample and assay stores, producing inconsistencies requires some knowledge. For example, to compute Mendelian errors, GenoByte needs to know the parents-children relationships between the record(s). To describe those to GenoByte, it is necessary to implement a few interfaces:

- **ComparableRecordProvider**: This interface tells GenoByte which records may be compared with one-another to detect reproducibility errors. The method `getComparableReferenceRecords()` should provide a vector of all records that may be compared with replicates. For each of these records, the method `getComparableRecords(int reference)` should return a vector of all records that should be compared with the specified reference.

- **MendelianRecordTrioProvider**: This interface describes the data structure to provide child-father-mother trios or child-parent duos that may be used to produce Mendelian errors. The method `getChildRecords()` should provide a vector of all records that may be considered as a child in one computation instance. The methods `getMotherRecords()` and `getFatherRecords()` should provide vectors of records which are considered as the parents

of a given child record.

- **MendelianErrorCalculator and ReproducibilityErrorCalculator**: These classes implement the calculation engine. They use the two interfaces described above to determine which records to use during the computation and produce objects that extend from Inconsistencies.

- **InconsistencyCountingStrategy**: The Inconsistencies objects generated by the calculators are passed down to instances of this interface which may handle them as they are being calculated. For example, an implementation could produce reports or persist the inconsistencies in an RDBMS.

## *How to extend GenoByte to use it in another program*

Provided with the main GenoByte distribution file is an example called "GenoByte Infinium". This example allows to see GenoByte in action, and also acts as a programming example showing the usage of the GenoByte API. We invite you to try executing this example, and then consult its source code (also included in the main distribution file). It will certainly give you a good idea of how to proceed.

In the future, tutorials offering a step-by-step guide of how to extend certain parts of GenoByte, such as dictionaries and statistics, will follow.

# Conclusion

We have presented you the basics of the internal behaviour of GenoByte. First, we explored a few general concepts on which the API is built. Then we introduced you to all the major components in use. Finally, we described the queries and statistics mechanisms, while mentioning that GenoByte has many extension points on which a programmer can add functionalities. We hope this has given you a good idea of the way things work in this application. Being in its first release version, GenoByte comprises some functional limitations, as listed on the project's website. Those limitations will be addressed in the future releases, depending on several things such as the general community's interest in the project. We encourage everybody to share their thoughts with us on GenoByte and Obiba, and we are very opened to suggested patches to our projects. We sincerely hope that this project will be helpful in your application.

# Copyright Notice