Maastricht University

Work conducted at the European Bioinformatics Institute

Bachelor Thesis

The Development and Optimisation of a Healthcheck System for the Ensembl Databases

Emma den Brok

External supervisors
Magali Ruffier & Andrew Yates

Internal supervisor
Dr. Christopher Pawley

June 6, 2016

Contents

Contents **1**

Abstract **2**

1 Introduction **2**

1.1 Bioinformatics . . . 2
1.2 Ensembl . . . 2
1.3 Relational Databases and Integrity Constraints . . . 3
1.4 Introduction to First Order Logic . . . 4
1.5 Logic for Integrity Checking . . . 5
1.6 Research statement . . . 7

2 Development **8**

2.1 Perl . . . 8
2.2 MySQL . . . 8
2.3 Other Specifications . . . 9
2.4 The Old HealthCheckSystem System . . . 9
2.5 Requirements for the New System . . . 10
2.6 Development of the New System . . . 11
2.7 The Change Detection System . . . 12
2.8 Integrity Simplification . . . 14

3 The System in Action **16**

3.1 Anatomy of a Healthcheck . . . 16
3.2 Runtime . . . 16
3.3 Results Integrity Simplification . . . 19

4 Discussion **20**

4.1 The New Healthcheck System . . . 20
4.2 Investigating Simplification of Integrity Constraints . . . 24
4.3 Shortcomings of the New System . . . 28
4.4 Further Development . . . 29
4.5 Other Possibilities . . . 30

5 Conclusion **31**

6 Reflection **31**

References **33**

A Appendix: Patch update **36**

B Appendix: Full table update **43**

Abstract

The Ensembl projects hosts genomic data for a large number of species. The current healthcheck system is outdated and cannot scale with the growth of Ensembl. This thesis will discuss the development of a new healthcheck system, written in Perl. Modules have been written to produce a general structure so that new healthchecks can easily be added. Healthchecks can be run both individually and in batch, and a system has been developed that runs healthchecks in response to changes to the database. This makes the healthcheck system much more efficient and therefore also more scalable. Furthermore, an investigation has been made into the benefits and limitations of integrity simplification. It has been shown that execution times are greatly reduced if knowledge about the changes made is exploited. This includes both limiting checks to the data that has been added, and increasing specificity of the change detection system. These possibilities show great promise of improving the healthcheck system in further development.

1 Introduction

1.1 Bioinformatics

Bioinformatics is a relatively new discipline within the natural sciences aimed at understanding and organizing biological information, often focussed on macromolecules such as DNA and proteins. The field emerged in the 1980s and 90s, when the first full genome sequencing projects started, in order to deal with the vast amounts of data obtained [1] [2]. Its origins can be traced back to the 1960s, when the first program that could deduce a sequence of amino-acids from fragmented protein molecules was created [3]. Since then bioinformatics has evolved to not only concern itself with the interpretation of biological data, but also the presentation and accessibility of this data [1]. For this purpose, biological databases have been created for a wide range of topics, ranging from marine biogeography [4], fMRI and dMRI [5], and proteomics [6] data. As one of the major institutes in the field, the European Bioinformatics Institute, part of the European Molecular Biology Laboratory (EBI-EMBL), provides data in a range of biological sciences on an open access and open source basis [7]. One of its data services is Ensembl.

1.2 Ensembl

The Ensembl project is a system that provides access to genome data of chordates and important model species [8]. It creates genome annotation, analysis, and storage through the processing of experimental evidence from sources such as the Genome Reference Consortium (GRC), UnitProt, and ENCODE. As such, Ensembl is a valuable information source and tool for researchers in fields such as biomedical research and conservational genomics. Recent research shows that users of EBI's dataservices rate its impact on their work to be worth £47 million annually [9], which is six times the operational cost of the EBI. In the same survey, it was shown that Ensembl and Ensembl Genomes were the fourth and sixth most accessed resources provided by the EBI. Taken together they were the most frequently accessed resource [9]. The main Ensembl project contains four databases: core, compara, variation, and regulation [10]. Ensembl Genomes extends the project for non-vertebrates with interfaces for bacteria, protists, fungi, plants, and (invertebrate) metazoa [11]. The Ensembl resources for each species are based on its genome assembly. A genome assembly is a set of combined DNA sequences that represents an organism's genome [12]. Ensembl does not produce genome assemblies itself, but downloads them from the International Nucleotide Sequence Database Collaboration (INSDC) [13]. It then performs automatic annotation through the Ensembl pipelines [14]. Annotation is the process wherein information is extracted from the genome. The pipelines compute data such as gene location, regulatory information (non-genic elements of the genome), variation in the genome over members of the same species, and comparisons between genomes of different species [8]. The genome and the annotation data are stored in the appropriate database (i.e. gene annotation in the core database, comparative annotation in the compara database, etc.). All Ensembl databases are in MySQL Relational Database Management Systems [15].

1.3 Relational Databases and Integrity Constraints

Relational databases are currently the most commonly used type of database [16]. First introduced by Codd [17] in 1970, the relational database model allows for data independence (i.e. application programs are no longer affected by changes and additions to the underlying database). This is in contrast with previous approaches, where access often depended on knowing the exact file path to and structure of the desired data. Based in mathematical set theory [16], the relational model can be interpreted as storing data in tables, or relations. If you have sets $S_1, S_2, \ldots, S_n$ that represent domains of various types of data, then the Cartesian product of these domains

$$S_1 \times S_2 \times ... \times S_n \tag{1}$$

represents the set of n-tuples $t_1, t_2, \ldots, t_n$ where for all i

$$t_i \in S_i \tag{2}$$

A relation R of degree n is then defined if it is a subset of this Cartesian product[17] [18]. Instead of ordering the set S_i as $S_1, S_2, \ldots, S_n$, an unordered set may be used if each tuple is not only associated with its domain, but also with a distinct index called an attribute. A tuple t in relation R is then a set of attributes defined on their corresponding domains that represents an instance of the data described by the relation.

For example, the **gene** relation in the Ensembl core database has 17 attributes, such as gene_id, source, description, and biotype. For each gene that is stored in the database, the information corresponding to that particular gene is stored as values of the appropriate attributes.

Set theory demands that all elements in a set must be distinct. As a result, all tuples in a relation must also be distinct [16]. This means that each relation must have a set of one or more attributes that uniquely identifies every tuple in that relation [17]. Such a subset of attributes is called a key, and a key is nonredundant if it consists of only one attribute or if the removal of any attributes from the set means uniqueness is no longer guaranteed. It is possible that a relation contains multiple nonredundant keys, and if this is the case one of the keys is arbitrarily chosen to be the primary key.

Codd [17] also demonstrated how the concept of primary keys can be used to display a relationship between two relations. A subset of attributes in a relation R is a foreign key if it has the same value as the primary key of the relation B it is referring to. Following the earlier example of the **gene** relation, gene_id is the primary key because it uniquely identifies each gene stored in the database. The foreign key canonical_transcript_id, also in the **gene** table, references to the **transcript** table's primary key transcript_id. This foreign key reference describes the relationship between a gene and its transcript, i.e. the RNA transcription of a gene (see Figure 1).

To ensure that a relation (and therefore the database) is valid, two constraints arise from the previous discussion [18]:

- Entity integrity: The primary key, or parts thereof, cannot have the value NULL.
- Referential integrity: A foreign key must refer to the primary key of a tuple of the referenced table, or else have the value NULL.

In addition, another type of constraint is the user-defined integrity constraint, which is a constraint a developer implements to reflect real-world limitations. Furthermore, it is important that the data for each attribute in a tuple is sensible, that is, realistic. This is known as data sanity. Enforcing and maintaining data sanity and data integrity is an important part of quality control in databases.

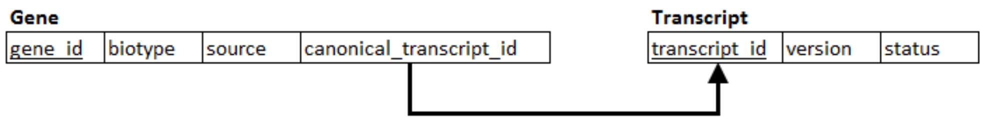


Figure 1: The foreign key relationship between the **gene** table and the **transcript** table

1.4 Introduction to First Order Logic

First order logic, also called predicate logic or quantified logic, is a type of logic that uses predicates and quantifiers to construct logic sentences [19]. An n-ary predicate is a sentence that contains n "gaps" that need to be filled by terms (variables or constants). In higher order logic, a predicate can also have other predicates or functions as arguments. Since this is not the case in first order logic, a predicate is also automatically an *atomic formula*. Consider the predicate P "x is a b", where x is a variable and b is a constant. Once these terms have been assigned values, the predicate can be evaluated to a *truth-value* (true or false). I.e. if the constant b has the value 'dog' and we assign 'Lassie' to x, the predicate "Lassie is a dog" evaluates to be true. However, if we assign the value 'Nemo' to x, the predicate "Nemo is a dog" is false (because Nemo is a fish). The value that a constant refers to is also known as the *referent* of that constant. The set of objects that is considered in a logic sentence is called the *universe of discourse* or the UD (in the example, the UD could be 'Animals in films'). The *extension* of a predicate is the set of values in the universe of discourse for which the predicate is true. A *model* M is the set containing the UD, the referents for each constant, and the extensions of all the predicates in the sentence(s) considered.

There are two types of quantifiers. The first is the *universal quantifier* denoted by the symbol $\forall$. If we take our previous example with the predicate Px, $\forall xPx$ is equivalent to "for all x, x is a dog". The *existential quantifier* is denoted by the symbol $\exists$. Writing $\exists xPx$ is equivalent to "there exists an x such that x is a dog". The variable x refers to an object in the UD.

Apart from predicate symbols (denoted with capital letters with subscripts as needed (i.e. P, P_1, A_{24})), variables, constants, and quantifiers, there are also connective symbols ($\wedge$ for AND (conjunction), $\vee$ for OR (disjunction), $\neg$ for NOT (negation), $\rightarrow$ for IF THEN (conditional), and $\leftrightarrow$ for IF AND ONLY IF (biconditional)), and parentheses to specify order of evaluation. A sentence of first order logic is a combination of atomic formula with connectives, quantifiers, and parentheses. It is important to note that sentences containing quantifiers do not have a truth-value, because they have open variables (i.e. x) that depend on the UD. Instead, we say that they are (not) *satisfied* in a certain model. For example, if our UD in model M is 'Animals in films', and the predicate P is "x is a dog", $\exists xPx$ is satisfied in M because there are animals in films that are dogs (i.e. Lassie). However, $\forall xPx$ is not satisfied in M because not all animals in films are dogs (i.e. Nemo is a fish).

All meaningful sentences are *well-formed formulas* (*wffs*) and only wffs will be discussed in this thesis. For a full definition of wffs see [19, p. 69].

1.5 Logic for Integrity Checking

The first publications on the use of first order logic for the simplification of integrity constraints were by Nicolas [20], and Bernstein and Blaustein [21]. Both published independently from one another in 1982. Bernstein and Blaustein's work is directly applicable to relational databases. They consider join clauses (comparing two indexed tuple variables) that can be rewritten to selection clauses (comparing an indexed tuple variable to a constant). In their method, a clause containing one or more joins is rewritten to one or more separate selection clauses. The constant in each selection clause has the value of the relevant column from the update. These selection clauses are cheaper to evaluate than join clauses, so checking constraints becomes more efficient. The constraint is evaluated with a set emptiness test (i.e. if a selection clause returns an empty set, the constraint is satisfied).

Nicolas [20] takes a similar approach, but his paper explicitly stresses one of the core ideas in integrity simplification: obtaining a simplified integrity constraint that can replace the initial constraint for a given update, provided that the database state satisfied all initial constraints before the update. Also, whereas Bernstein and Blaustein develop an algorithm to produce a simplified constraint for a database state, Nicolas turns to first order logic. He views a relational database state as a first order model: an interpretation of a set of logical statements that describe the various constraints. He describes certain special forms wffs can take that are important for logic use in this context. A wff in the *prenex form* has all its quantifiers at the front of the formula. Then, if the remainder of the wff is rewritten to consist of a conjunction of disjunctions, atoms, and negated atoms, the wff is in *prenex conjunctive normal form*. Integrity constraints are defined as wffs in prenex conjunctive normal form that satisfy the *range-restricted property*:

- Every universally quantified variable in the wff is in at least one negated atomic formula for every disjunction in which the variable exists.
- For every existentially quantified variable in a negated atomic formula, there is a disjunction containing only non-negated atomic formulas which all have this variable as an argument.

The properties of a range-restricted wff cause its variables to only take values that can exist in the database state. Nicolas then goes on to show that insertions can only cause constraint violations if the insertion is made into a table that occurs in a negated atom in the constraint. Similarly, only deletions from tables that occur in non-negated atoms in the statement can violate the constraint. By substituting the values of the inserted or deleted tuple into the constraint, it can be evaluated to see the effect of the change. Nicolas' way of seeing a database state as an interpretation of logical statements would evolve into the concept of deductive databases, on which he published a defining paper in 1984 together with Gallaire and Minker [22].

Deductive databases lend themselves well to integrity simplification because of their close relation to first order logic (they are defined by a declaration of rules, not by a set of relations [16]). In a 2006 article, Christiansen and Martinenghi [23] expand on previous research including [20] and their own work [24], setting out a framework for integrity simplification for deductive databases. They stress that this should be done during the design of the database, and propose methods to produce parametric constraints which will be evaluated before each update is applied to the database. They call this the *weakest precondition*; a constraint that can be checked in the present database but using properties of the future state (which are known if the update is known). They go on to show how to specialize and optimize the weakest precondition. Checking constraints before the update is applied will greatly reduce the cost of integrity maintenance, as incorrect data is never introduced into the database and therefore expensive rollback operations are never needed.

This type of integrity simplification assumes the update, or the update pattern, is known during simplification [25]. Generally speaking, simplification methods work either by making the evaluation of the constraints more effective such as in [21], or as in [20] and [23] by limiting the number of constraints that need to be evaluated through exploitation of update knowledge and the assumption that before the update, all constraints were satisfied [26]. Still, these methods rely on the same principle: exploit what is known, both of the database state and the update, to make the evaluation of constraints as specialized and efficient as possible. This means that practical implementation does not need to be limited to one method, but can combine several as is suitable. However, no works that demonstrate application of these methods to real-life databases have been found, and examples given in the aforementioned papers are simple and consider databases of only four or five tables. The only work known to the author on the implementation of simplified integrity constraints is a 1981 master thesis [27], which was published before the works of Bernstein and Blaustein and Nicolas. A copy of this thesis could not be obtained.

1.6 Research statement

Ensembl releases new data four to five times a year. A release can contain new or updated data for species that were already in Ensembl, and databases for new species. During the release cycle that takes place before a release, hundreds of databases are created, updated and moved between servers. Pipelines that compute additional information or analyse data rely on the results of earlier processes. Because of the dependencies between pipelines, faulty data can cause problems downstream. If this happens, several weeks worth of processing may need to be redone. The quality control system currently in place to prevent this was implemented in Java more than a decade ago. The system as it is now is the result of years of build-up; if faulty data was not picked up by existing healthchecks, a new check would be created. As a result the current system contains a large amount of healthchecks for which the purpose is not always clear. At the moment, the entire healthcheck system is run every night regardless of whether changes are made to the databases or not. This consumes a lot of time and resources, and every time a new species is added to Ensembl the runtime increases linearly. The problem that arises is that the system will not scale with planned future developments. Moreover, Ensembl has become strongly Perl oriented, and as a result there is not enough Java expertise to extend or alter the current system as necessary. This means that a new system is needed.

This thesis will describe the development and optimisation of a new healthcheck system for Ensembl. The new system will be written in Perl and should provide a framework in which future development can take place. The new healthcheck system should address all the aforementioned issues in order to evolve into a system that can scale with the future growth of Ensembl, but also allows for easy addition of desired checks. If the development is successful it will free up time and computational resources, allowing the Ensembl team to focus on development and innovation rather than trivial maintenance. Moreover, it should be easier to identify, develop, and run necessary healthchecks, which means that safeguarding the correctness of data will be easier. In the end, this will help Ensembl provide its data services to geneticists and other researchers all over the world.

Moreover, the research project will serve as an investigation in the practical application of integrity simplification. There are no works known to the author on the practical application of integrity constraints save for the master thesis by Homeier [27], produced before the major works in this field were published. Christiansen and Martinenghi[23] have touched on the speed-up that their methods produces, but their example considers a very simple database. As a result, not much is known about the practical possibilities and pitfalls of integrity simplification. This project will provide insight into this area as well.

The structure of the thesis is as follows: in the next section, the development of the new healthcheck system will be laid out. This includes specifications on languages and software used, a discussion on the old healthcheck system, more specified requirements for the new system, and the process of its development. It also contains a section in which constraints from the new system are simplified following [20]. The section 'The System in Action' describes the system in its current state, and also contains the results of the integrity simplification. After this, the discussion section evaluates both the new system and the merits of integrity simplification. Moreover, it sets out possible directions for future development, as well as other possibilities for the healthcheck system that were out of the scope of the research. The thesis finishes with a conclusion on the findings, and a reflection on the work carried out during the bachelor thesis period.

2 Development

2.1 Perl

Perl is a dynamic programming language [28] first developed by Larry Wall in 1987 [29]. The language in its current form is known as Perl 5 [29]. Perl is widely used in bioinformatics and is the main language of the Ensembl project [15]. The new healthcheck system is compatible with Perl version 5.14.2. The system utilizes modules from the Ensembl Perl API. The API provides a level of abstraction between the databases and the healthchecks, through object-oriented methods that can readily be used to connect to, and query the database. Other modules used are: Moose [30], Getopt::Long [31], File::Spec [32], and Data::Dumper [33]. All these modules are included in standard distributions of Perl, except for Moose, which due to its widespread use is still considered standard [28]. For performance testing, the Devel::NYTProf module [34] was used.

2.2 MySQL

The Ensembl generic databases all share the same schema definition of 72 tables. Testing of the system focussed on the 'core' database in this group. Full details of the schema can be found at `http://www.ensembl.org/info/docs/api/core/core_schema.html`. The database engine is MyISAM. Later on in the development triggers were used. A trigger is an object which is attached to a table, that performs a task when it is activated by a MySQL command on the table [35]. For example, a trigger can be activated by the trigger event of an insertion into a table. Triggers can be set to act before or after the trigger event is commited to the database [35]. Both MyISAM and the underlying MySQL engine make use of indexing to speed up querying, even when query caching is switched off. To prevent this from affecting results, all relevant tables were deleted (not dropped) and then repopulated before a query was timed during benchmarking. Unless stated otherwise, all healthchecks were run on a local copy of the homo_sapiens_core_84_38 database.

2.3 Other Specifications

The development of the healthchecks and overarching infrastructure was done on two Linux virtual machines: the Ensembl VM (version 82) which is a 64-bit Ubuntu 12.04.1 with 1024MB base memory, and another 64-bit Ubuntu VM, with 1024MB base memory and Ubuntu version 14.04. Git was used for code management, and the repository containing all work produced during the course of the research can be found at `https://github.com/edenbrok/Ensembl_thesis`.

2.4 The Old HealthCheckSystem System

The old healthcheck system is contained in a github repository (`https://github.com/Ensembl/ensj-healthcheck`) consisting of various directories containing the healthchecks for several database types, helper functions, and configuration files. Initial development of the healthcheck system will, per request, focus on the 'generic' directory, which contains checks for the Core, EST, EST-Gene, Vega, CDNA, Otherfeatures, Sangervega, Rnaseq, and Presite databases. As these databases all share the schema definition a lot of the healthchecks are shared between them. Still, some types have different user-defined integrity constraints than others, which means that not every healthcheck in the directory applies to all the generic databases. The old healthchecks are written in Java and can be run individually or in batch. Currently, the whole repository of healthchecks is run on the Ensembl server farm every night.

Since there are more than one hundred healthchecks in the generic directory, it was clear that only a subset of these could be adapted for the new system given the time frame of the research. In order to provide a good proof of concept it was important that this subset covered a variety of healthchecks that were representative of the whole set. Therefore, an index was made for all the healthchecks, where each check was assessed on function, correctness, and clarity. This meant that for each healthcheck, it was checked that its functionality matched its description. Moreover, an assessment on the possible redunancy of checks or the possibility of the combination or generalization of several healthchecks was made. Finally, the healthchecks were grouped into five different categories based on the type of check they performed:

1. 'Hard' integrity - healthchecks that check entity or referential integrity.
2. 'Soft' integrity - healthchecks that check user-defined integrity.
3. Schema-related sanity - healthchecks that check data sanity as dictated by schema declaration, i.e. uniqueness of rows, and allowance of blank or NULL values of cells.
4. Data-related sanity - healthchecks that check the sanity of values based on outside constraints, i.e. realistic values, or correct format of values.
5. Database comparisons - healthchecks that check if tables or declarations between databases match as expected.

Some healthchecks fit more than one group, in which case the category was determined by the dominating property. In some cases the distinction between categories two and four was hard to make - in these cases more complex healthchecks were placed in category two and the simpler ones in four. Still, the grouping is not absolute and in some cases arguments for placement can be made for several categories.

The subset for development was then chosen, making sure it contained healthchecks from each category, and also some variety within each category. The selection contained the following healthchecks:

1. Hard integrity:
   - CoreForeignKeys
   - AncestralSequencesExtraChecks
2. Soft integrity:
   - AssemblyMapping
   - AssemblyMultipleOverlap
   - FeatureCoords
   - LRG
   - Meta
   - ProjectedXrefs
   - SeqRegionCoordSystem
   - VariationDensity
   - XrefTypes
3. Schema-related sanity:
   - AutoIncrement
   - BlanksInsteadOfNulls
   - SchemaType
   - StableID
4. Data-related sanity:
   - AssemblyNameLength
   - DataFiles
   - GeneCount
   - NonGTACNSequence
   - XrefPrefixes
5. Database comparisons:
   - ComparePreviousDatabases
   - CoordSystemAcrossSpecies
   - MySQLStorageEngine
   - ProductionMeta
   - SeqRegionAcrossSpecies

All healthchecks are named as they appear in the old system, and can be found in their old form at `https://github.com/Ensembl/ensj-healthcheck/tree/release/84/src/org/ensembl/healthcheck/testcase/generic`.

2.5 Requirements for the New System

The Ensembl core and production teams laid out key requirements for the project. They asked that the system be largely independent from external software or code, and to only use standard Perl modules (i.e. those installed with Perl, or ones so widely used they can be considered standard (i.e. Moose)). This ensures the project does not depend on (continued) support by sources that are out of Ensembl's control. As a result, the healthchecks will either query the databases directly or use the Perl Ensembl API.

Another requirement was that the new healthchecks should be able to run both independently and in groups, and in a number of different contexts. This ensures the healthchecks can be used in a flexible way - a user can choose to run a single healthcheck, decide to run them all in batch, or do checks at certain points during a pipeline that populates (part of) the database. Moreover, there should be an infrastructure that runs healthchecks in response to changes in the database.

The new system should also give output that is both human and machine readable with regards to success or failure of healthchecks, plus additional information where necessary. For certain healthchecks (those involved in data sanity), failing a check can mean there is a problem, but this problem is not necessarily critical. For these cases there should be a mechanism in the infrastructure that allows a user to make a note of the failure and set the infrastructure to ignore future failure of these checks for the particular database.

2.6 Development of the New System

In the first phase of development, a set of healthchecks were adapted into Perl. During this process, helper functions that were needed for the individual healthcheck but showed potential for wider use were written as modules in the DBUtils namespace. All these modules can be found at `https://github.com/edenbrok/Ensembl_thesis/tree/master/lib/DBUtils`. Queries to the database were made with the Bio::EnsEMBL::Utils::SqlHelper functionalities. At this stage, information on success or failure of the healthchecks was printed out to the user using Perl's `print` function. During this stage it became clear that the initial set of healthchecks chosen for development was too ambitious. As the goal was to give a proof of concept, and the infrastructure would be an important part of this, the decision was made to not spend extra time developing the full set that was previously chosen, but to keep to the planning to allow enough time for infrastructure development. In order to ensure that the healthchecks that were adapted were diverse enough, it was made sure one or more healthchecks were adapted from each group as described in section 2.4. The healthchecks that have been adapted into Perl are: CoreForeignKeys (1), AssemblyMapping (2), LRG (2), ProjectedXrefs (2), SeqRegionCoordSystem (2), SequenceLevel (2), XrefTypes (2), AutoIncrement (3), Meta (3), AssemblyNameLength (4), DataFiles (4), and CoordSystemAcrossSpecies (5). In the repository, the script of each healthcheck can be found in the folder of the group they belong to. The focus was laid on integrity healthchecks, as they were more interesting for integrity simplification. It should also be noted that AncestralSequencesExtraChecks was merged with CoreForeignKeys.

Once the healthchecks had been adapted into Perl in a basic matter, it was investigated where they could be improved. First, user input from the command line was added as a feature. This was done with the Getopt::Long module. For each healthcheck, users could now specify the species and database type (core, rnaseq, etc) they wanted to run the healthcheck on. The relevant database connection would then be retrieved using the Bio::EnsEMBL::Registry which retrieves all database adaptors from the Ensembl server. All healthchecks shared the same required step, where they need to retrieve a Bio::EnsEMBL::DBSQL::DBAdapter and establish connection to the database in order to create a Helper instance to query the database. The fetching of a DBAdapter was generalized in the DBUtils::Connect module. Moreover, the Connect module also allows for user input to be read from a configuration file, so that the user does not need to specify the desired database for every individual healthcheck. This makes running healthchecks in batch easier. The user can choose to retrieve a database adapter from the registry or from a specified (local) connection. If the database adapter is taken from the registry, command line input on species and type will override the configuration file. The user can also specify the name and location of the configuration file through the command line. In order for the command line arguments to be handled properly the pass_through option of Getopt::Long had to be set to true for all healthchecks. Otherwise all the command line arguments are removed from the `@ARGV` array on the first lookup (in this case when the healthcheck calls DBUtils::Connect with the location of the configuration file), and no information on species or database type is left for the Connect module itself. The default for the configuration file is a file named 'config' in the parent directory of the working directory.

Another effort to achieve uniformity was made with the Logger module (see `https://github.com/edenbrok/Ensembl_thesis/blob/master/lib/Logger.pm`). This allows for informative and uniform output on the process of each healthcheck. Each healthcheck creates a Logger object (in Moose) that has the healthcheck name, the species, and database type the healthcheck is currently being run on, as attributes. Informative messages are then sent by calling the message method on the logger object. It will print the message, prefixed by the healthcheck name, species, and database type. This will allow the user to clearly see what database the healthcheck applies to, if the checks are run in batch. To inform the user of the success or failure of a healthcheck, the logger object can call the result method with a boolean value. If the boolean is true, the user is informed the healthcheck passed without finding any problems. Otherwise, it will inform the user that the healthcheck has failed.

Another important step was to remove large amounts of hardcoded values from the healthchecks. This concerned the CoreForeignKey and AutoIncrement healthchecks, which, at this point, both contained large amounts of hardcoded values on what tables to query within the checks. Taking these out of the healthcheck script would improve readability, and would also allow users to make changes or additions to the set of tables that need checking without touching the actual script, thus minimizing the possibility of errors being introduced during this process. The files containing the input of these two healthchecks were stored as modules in the Input namespace, at `https://github.com/edenbrok/Ensembl_thesis/tree/master/lib/Input`. In the case of AutoIncrement, all the table-column pairs that should be set to autoincrement are stored in an array. For CoreForeignKeys, the foreign-key pairs were stored in a multidimensional hash reference which also specifies if the constraint should hold in both directions or if any further constraints should apply. Because the queries are all made through the same format, the healthcheck script can simply iterate over the input file to run the queries.

2.7 The Change Detection System

A fairly simple but very effective change detection system that uses metadata from the database was set up. Accessing the metadata database information_schema through the query

```
SELECT TABLE_NAME, UPDATE_TIME FROM
    information_schema.tables
        WHERE table_schema = DATABASE();
```

returns the table names from the current database, and the date and time when they were last updated. The change detection system is based on comparing the result of this query with an earlier result of the same query, which was saved the last time the health check suite was run on the database.

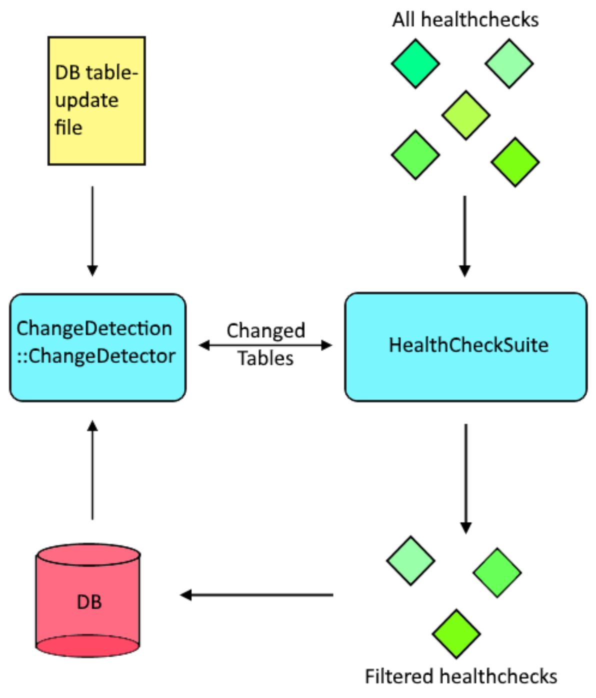


Figure 2: The interactions of the HealthCheckSuite with other parts of the health check system.

When HealthCheckSuite.pl is run, it retrieves an EnsEMBL::DBAdaptor object that provides a connection to a database (local or on an a external server), as specified in the configuration file ('config'). It passes this database adaptor to the ChangeDetector module. The ChangeDetector module runs the above query on the database using the SqlHelper `execute_into_hash` function. This returns a hash reference where the table names are the keys. Then, it retrieves the database name from the adaptor, and looks for a file of the same name. If such a file is not found, it is assumed that it is the first time the healthcheck suite is run for this particular database. A file with the same name as the database is created and the hash reference is printed to it using Data::Dumper. As it is the first time the health check suite is run, all tables need checking, so the ChangeDetector module returns an array containing all the tables in the database to HealthCheckSuite.

If a file matching the database name is found, the ChangeDetector module parses this file (using Perl's `do` method). For each table name the old update time is taken from the file and compared to the newly retrieved update time. If the update time is found to have changed it means that the table has changed since the last time the healthcheck suite was run. If this is the case, the table name is pushed into an array containing all the changed tables. Once update times have been compared for all tables, the file containing the table-update pairs is rewritten with the new results. Then, the array containing the changed tables is returned to HealthCheckSuite.

The next step in HealthCheckSuite is the initialization of the healthcheck objects. Each object describes a healthcheck: its name, the database types it applies to, the tables it uses, and in which type of healthcheck it is (1-integrity, 3-sanity, etc.). Turning each healthcheck into an object means they can be treated more intuitively, and extracting the necessary properties from them is easier and cleaner, code wise. All the created healthcheck objects are stored in an array.

Iterating over this array, healthchecks that do not apply to the type of the database that will be checked are removed from the array (i.e. the DataFiles healthcheck is only relevant for 'rnaseq' databases – so it will be removed if a 'core' database is being checked). Then, each element in the array containing all the changed tables is compared to the tables that are used by each healthcheck object. If a healthcheck uses the changed table, its 'applicable' property is set to 1 (true). If the table is not used by the healthcheck, the 'applicable' property keeps it value (the default is 0 (false)).

At last, the healthcheck objects are iterated over a final time. If the applicable property is set to true for a healthcheck, it calls the function `run_healthcheck` on itself. This function determines the location of the healthcheck based on its category, and then proceeds to run it. Once the healthcheck has finished, the program returns to HealthCheckSuite which can then move to the next healthcheck object. A schematic representation of the change detection system can be seen in Figure 2. The code involved can be found at `https://github.com/edenbrok/Ensembl_thesis/tree/master/lib/ChangeDetection`

2.8 Integrity Simplification

We will now apply the simplification method in [20] to several constraints that apply to the Ensembl core databases to demonstrate its uses and limitations. The first concerns the foreign key constraint, which is defined as follows: A foreign key must refer to the primary key of a tuple of the referenced table, or else have the value null. This can be expressed in first order logic:

$$\forall x(TABLE1(x) \rightarrow (TABLE2(x) \lor x = NULL)) \tag{3}$$

were **TABLE1** is the referencing table, **TABLE2** is the referenced table, and x is the value of a primary key of **TABLE1**. This statement is already in prenex form. To transform it into prenex conjunctive form we must rewrite it without the conditional:

$$\forall x(\neg TABLE1(x) \lor TABLE2(x) \lor x = NULL) \tag{4}$$

These statements are logically equivalent, that is, their truth tables match. This transformation is fairly trivial, but neatly illustrates one of Nicolas' core ideas: a constraint can only be violated by an insertion into a relation that occurs in a negated atom in the constraint, or by a deletion from a relation that occurs in a non-negated atom. In the foreign key constraint, a newly inserted tuple in **TABLE1** may refer to a non-existent tuple in **TABLE2** and thus violate the constraint. But as the dependency is one-directional, no insertions into **TABLE2** could violate the result. It follows that deletions from **TABLE2** could violate the constraint, but not deletions from **TABLE1**.

Next, constraints that are checked in the SequenceLevel healthcheck will be investigated. The first constraint is fairly simple, and states that if a tuple in the **coord_system** table has the value 'contig' for the attribute *name*, then the value of the *version* attribute may not be NULL for that tuple:

$$\forall x \forall y((coord\_system(x,y) \wedge x = \text{`}contig\text{'} \rightarrow y \neq NULL) \tag{5}$$

Where x is the value of the attribute *name* and y is the value of *version.* This statement is in prenex form. If we then rewrite it to prenex conjunctive form

$$\forall x \forall y(\neg(coord\_system(x,y) \wedge x = \text{`}contig\text{'}) \vee y \neq NULL) \tag{6}$$

$$\forall x \forall y((\neg coord\_system(x,y) \wedge \neg x = \text{`}contig\text{'}) \vee y \neq NULL) \tag{7}$$

$$\forall x \forall y((\neg coord\_system(x,y) \vee y \neq NULL) \wedge (\neg x = \text{`}contig\text{'} \vee y \neq NULL)) \tag{8}$$

In this case, this does not help narrow down the constraint as only one table is involved. We can generalize this and say that if a constraint only involves one table, the simplification method as proposed in [20] does not work.

Now a more complex constraint, also from the SequenceLevel healthcheck, will be considered. This constraint states that if a tuple in the **seq_region** table is referenced by a tuple in the **dna** table, then the **coord_system** table that the **seq_region** tuple refers to should have the value 'sequence_level' for the attribute attrib_type. This can be written in logic form as:

$$\forall x \exists y(seq\_region(x,y) \wedge dna(x) \rightarrow coord\_system(y, \text{`}sequence\_level\text{'})) \tag{9}$$

Where x is the primary key of the seq_region table and y is the primary key of the coord_system table. If we then transform the statement into prenex conjunctive form:

$$\forall x \forall y(\neg seq\_region(x,y) \vee \neg dna(x) \vee coord\_system(y, \text{`}sequence\_level\text{'})) \tag{10}$$

We can see that the statement satisfies all the conditions for a range-restricted wff. The relations **seq_region** and **dna** occur in negated atoms, so insertions into these relations can potentially violate the constraint. Furthermore, only deletions from **coord_system** can potentially violate the constraint. This can be specified even further: only deletions from **coord_system** where the attribute attrib_type has the value 'sequence_level' can potentially violate this constraint.

This method cannot be used for all constraints. For example, take the constraint from the ProjectedXrefs healthcheck. This states that there must be at least one entry in the **gene** table such that gene.display_xref_id refers to an entry in the **xref** table that has its info_type attribute set to 'PROJECTION':

$$\exists x(gene(x) \wedge xref(x, \text{`}PROJECTION\text{'})) \tag{11}$$

where x is the primary key of **xref**. This statement meets all the conditions for a range-restricted wff. It tells us that insertions into the **gene** or **xref** cannot cause constraint violation, but that deletions from either of these tables can. However, if we substitute the values of a deleted tuple into the statement and it is satisfied, we can still not infer if the relation from which it was deleted now violates the constraint. This is because all variables in non-negated atoms in this constraint are existentially quantified. One instance that satisfies the constraint may be lost, but we cannot know if there are still other instances that do. In contrast, consider the constraint in equation (10). If we delete a tuple from the **coord_system** table with the value 'sequence _level' for attrib_type we can check if this will violate the constraint by substituting the deleted tuple into the constraint. If the constraint is satisfied, we know that after the deletion the constraint will be violated, as the tuple from **seq_region** now has a foreign key y that refers to a non-existent tuple and the universal quantifiers dictate that the constraint should be met by all instances.

3 The System in Action

3.1 Anatomy of a Healthcheck

One of the challenges in the design of the new healthcheck system was its extensibility. An important requirement was the easy adaptation and addition of existing and new healthchecks to the system. To allow this, several modules were created, and existing modules from the Ensembl API were exploited. As a result, every healthcheck has the same 'skeleton':

- The DBAdaptor Ensembl object: using the DBUtils::Connect module, a database adaptor is retrieved according to the specifications of the configuration file. The location and name of this configuration file can be specified as command line arguments of the healthcheck.
- The Logger object: from the DBAdaptor, the species and type of the database are retrieved. Along with the name of the healthcheck, this information is used to create a Logger object, which will print information about the healthcheck to the command line window (or a user-specified file) while it is being run.
- The SqlHelper object: through the Ensembl API, a SqlHelper object is created using the DBAdaptor. The SqlHelper takes care of the preparation and execution of SQL statements on the database.

All that needs to be added to this framework are the SQL statements specific to the healthcheck, and the handling of the results of these statements. The logger object can be used to inform a user of problems found in the data. It also prints the final result (success or failure) of the healthcheck to the user. As an example, part of the LRG healthchecks that shows these elements can be found in Codesnippet 1 (on the next page).

3.2 Runtime

First, to give the reader an idea of the runtime of the healthchecks, each of them was run individually and timed using the NYTProf module. The results can

```
my $config_file;

GetOptions('config_file:s' => \$config_file);

my $dba = DBUtils::Connect::get_db_adaptor($config_file);

my $species = DBUtils::Connect::get_db_species($dba);

my $database_type = $dba->group();

my $log = Logger->new(
        healthcheck => 'LRG',
        species => $species,
        type => $database_type,
);

if(lc($database_type) ne 'core'){
        $log->message("WARNING: this healthcheck only
            applies to core databases. Problems in
            execution will likely arise");
}

my $helper = Bio::EnsEMBL::Utils::SqlHelper->new(
        -DB_CONNECTION => $dba->dbc()
);

my $result = 1;

if(assert_lrgs($helper)){
        $result &= assert_lrg_annotations($helper, 'gene'
            );
        $result &= assert_lrg_annotations($helper, '
            transcript');
}
else{
        $log->message("SKIPPING: No LRG seq_regions found
             for $species; skipping the test");
}

$log->result($result);
```

Codesnippet 1: The LRG healthcheck, displaying the common elements of every healthcheck.

be found in Table 1. Healthchecks were run on a local human core database (ensembl version 84, assembly 38), exceptions noted.

Healthcheck	Runtime (s)	Comments
CoreForeignKeys	1624	
AssemblyMapping	1.29	
LRG	2.51	
ProjectedXrefs	4.58	On local cavia porcellus core database
SeqRegionCoordSystem	16.9	
SequenceLevel	1.35	
XrefTypes	188	
AutoIncrement	0.92	
Meta	1.02	
AssemblyNameLength	0.64	
DataFiles	2.27	On human Rnaseq database through registry
CoordSystemAcrossSpecies	2.70	On registry

Table 1: Runtimes of the different healthchecks

Run	Time (s)
Healthchecks in batch	1758
HealthCheckSuite	1781
HealthChecksuite with mock update	240

Table 2: Execution times for running all the healthchecks in batch, running the full healthcheck suite, and running the healthcheck suite with an update and change detection.

The ProjectedXrefs healthcheck does not apply to the human database and is therefore run on a local guinea pig database. DataFiles only applies to Rnaseq databases, so therefore a connection to the human rnaseq database was retrieved from the registry (the local guinea pig database had been installed during orientation, but complete databases take up a significant amount of memory so the decision was made not to download the human Rnaseq database as well). The CoordSystemAcrossSpecies healthcheck compares several databases of one species, so here a connection to the registry was needed as well. It should be noted that connecting to and retrieving the registry may take up to two to three seconds, and is therefore the dominating time factor for both these healthchecks.

As the change detection system does not actually speed up individual healthchecks (save for the CoreForeignKey check), execution times should not change considerably depending on whether the HealthCheckSuite is run for the first time on a database or if all the healthchecks are run in batch. How much time is saved by the HealthCheckSuite depends entirely on how many tables have been changed, and speed-up can range from 0% (when all tables have changed) to 100% (no tables have changed). Still, to illustrate the usefulness of the system we will provide an example. First the HealthChekcSuite will be run, followed by a run of all the healthchecks in batch, in order to compare execution times. These should not differ significantly. Then, a patch update will be mocked. A patch update represents a corrected version of a piece of DNA. No actual update will take place, but the database file will be altered to make the suite think the **assembly**, **assembly_exception**, **dna**, **seq_region**, and **seq_region_attrib** tables have changed, and the HealthCheckSuite will be run again. The results of this experiment are shown in Table 2.

3.3 Results Integrity Simplification

Benchmarking tests with three different SQL queries were run to investigate the simplification proposed by [21]. They suggest join clauses are rewritten as select clauses, which should be computationally less costly and therefore quicker. This means that the content of the update must be known. For this purpose, audit tables were created for all tables involved in a patch update. Using MySQL triggers which acted after each insertion, the primary key of each inserted tuple was added to the respective audit table after an insertion was made into the table. To test the effectiveness of this simplification, a patch update was then mocked by inserting all rows in the **assembly**, **assembly_exception**, **dna**, **seq_region**, and **seq_region_attrib** tables from release 82 that where present in release 84 but not in release 82 (as patch updates occur every other release). The foreign key constraints applicable to these tables were then checked with three different queries:
Method 1:

```sql
SELECT COUNT(*) FROM assembly LEFT JOIN
    seq_region
        ON assembly.asm_seq_region_id =
            seq_region.seq_region_id
        WHERE seq_region.seq_region_id IS NULL;
```

Method 2:

```sql
SELECT COUNT(*) FROM assembly_audit LEFT JOIN
    seq_region
        ON assembly_audit.asm_seq_region_id =
            seq_region.seq_region_id
        WHERE seq_region.seq_region_id IS NULL;
```

Method 3:

```sql
SELECT COUNT(*) FROM assembly_audit
        WHERE asm_seq_region_id NOT IN
         (SELECT seq_region_id FROM seq_region);
```

Where method 1 is no different from the normal check done by CoreForeignKeys on these tables, and serves as a control. Method 2 also uses the same query, but makes use of the audit tables and therefore only checks rows that have changed. Method 3 represents the constraint rewritten to not contain any joins as suggested in [21]. The scripts used in these tests can be found on the integrity-simplification branch of the repository, in the 'integrity-simplification' folder. To see how these different queries scaled, method 2 and 3 were also run on audit tables containing all the primary keys from the relevant tables in release 84 (this simulates an update where a large number of rows have been added to a previously empty table). For both updates every method was run ten times, where each time the query times were retrieved using the NYTProf module and then summed to show the total time spent querying the database. The average over ten runs plus standard deviation for each method are shown in Table 3.

Update	Method 1 (time in μs)	Method 2 (time in μs)	Method 3 (time in μs)
On patch update	22334910 ± 8079329	217447 ± 64313	208426 ± 45961
On complete table update	not applicable	4450478 ± 788958	5144512 ± 1412043

Table 3: The execution times of foreign key constraints checked with three different methods, on two different updates

4 Discussion

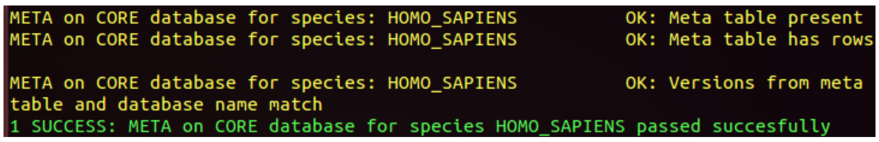
4.1 The New Healthcheck System

The new healthcheck system in its current state provides a good proof of concept, both on the level of writing individual healthchecks in Perl, and on how an infrastructure that can detect change will greatly speed up the healthcheck process.

To ensure correct working of each healthcheck, faulty entries were created in a test database for all of them. It was made sure that the healthchecks picked up on a variety of constraint violations or wrong data cases as necessary. While doing this it was also made sure that these problems were correctly reported to the user.

In section 3.1, it has been demonstrated that the majority of healthchecks that have been adapted show the same structure. This structure has been further generalized through the DBUtils::Connect and Logger modules. As a result, new healthchecks can easily be developed. All that needs to be done is to write an SQL statement that can find the problem in the data, and some code to parse the result. The connection to the database and reporting to users is handled by the existing modules. Moreover, new healthchecks can easily be added to the infrastructure. This will be discussed in more depth later. This general structure does, however, not work for all healthchecks. The connect module only returns one DBAdaptor object: The one specified in the configuration file. Since the DButils::Connect module cannot be used twice within the same healthcheck to retrieve different database adaptors, healthchecks that involve multiple databases will still need code that connects to and retrieves adaptors for those databases.

The Logger object produces messages that are always prefixed by the healthcheck name, and species and database type on which the check was run. This makes it easy for users to identify where issues occur when healthchecks are run in batch (see Figure 3).



```
META on CORE database for species: HOMO_SAPIENS          OK: Meta table present
META on CORE database for species: HOMO_SAPIENS          OK: Meta table has rows

META on CORE database for species: HOMO_SAPIENS          OK: Versions from meta
table and database name match
1 SUCCESS: META on CORE database for species HOMO_SAPIENS passed succesfully
```

Figure 3: Example output to the command line from the Logger module.

Moreover, the messages are colour-coded (yellow for general messages, green for success, and red for failure) so the user can easily spot failures. No work has been done on the computer parsing of these messages, but this should be possible in the current format. The result messages all follow the format: “0/1 SUCCESS/FAIL: (healthcheck name) etc.”, and this could easily be parsed using regular expressions. Information printed with the message function all share the same prefix of “(HEALTHCHECK NAME) on (TYPE) database for species: (SPECIES)”, but there is no structure imposed on the message itself. Still, during development the following standard was followed: Messages informing the users a certain part of the healthcheck had passed succesfully started with “OK:” followed by the relevant information. Messages that informed users of faulty data started with “PROBLEM:” followed by a description of the mistake found. If this standard is followed in future development then machine parsing of these messages should also be possible. This could mean, for example, that a program parses all the messages and only keeps the result messages, but in case of failure also stores the informative messages from that particular healthcheck. This would keep the output free from noise and make it easier to find and identify mistakes in a database.

Though the runtime of healthchecks can vary significantly depending on whether they are run on a local or external server, how busy the server is etc., running times will still be proportional between healthchecks run in the same conditions. Therefore, the results in Table 1 can be used to make some observations. Firstly, it is clear that the CoreForeignKeys healthcheck is by far the most time consuming healthcheck, taking 1624 seconds (about 27 minutes) in this case. This is not surprising: it contains 58 foreign key requirements that are checked by joins on complete tables, that can have anywhere between 7 and 26676923 rows (in the core human database), plus some additional checks. Join statements are expensive because they require a full index scan on the referring table, and then loop over these indexes to do a unique key lookup on the referenced table. It is slightly better than a brute force algorithm, because in the inner loop MySQL uses the keys to look up pairs, rather than to test every row in the referred table against one foreign key from the referring table [16]. Still, the MySQL engine has to loop over large amounts of data.

On the other side of the spectrum are AutoIncrement and AssemblyNameLength. Both take less than a second to run, which is expected as they use simple, specified queries. AssemblyNameLength queries the **meta** table for existence of the value 'assembly.name' in the meta_key column, and checks if the corresponding meta_value is no longer than 16 characters. AutoIncrement makes use of the SHOW COLUMNS syntax which accesses metadata of a table, specifying which column it wants results for. This is a very well specified query that returns only one row. The check simply looks if the auto_increment flag is present in this database. Both these healthchecks are fast because they have very specific and simple queries, and as a result MySQL does not have to shift through a large amount of rows to produce a result.

Most other healthchecks take slightly longer, but take only a few seconds (from 1.02s for Meta to 4.58 for ProjectedXrefs). Outliers (apart from CoreForeignKeys) are SeqRegionCoordSystem (16.9s) and XrefTypes (188s). Analysing SeqRegionCoordSystem with NYTProf shows that the subroutine check_lengths is the main culprit, and is responsible for almost 70% of the runtime, even though it only runs one query on single-species databases such as the human core. Looking at the query, it quickly becomes clear why it takes a long time:

```
SELECT COUNT(*) FROM seq_region s1, seq_region s2
    , coord_system c1, coord_system c2
        WHERE s1.name = s2.name
        AND s1.coord_system_id != s2.
            coord_system_id
        AND c1.coord_system_id = s1.
            coord_system_id
        AND c2.coord_system_id = s2.
            coord_system_id
        AND s1.length != s2.length
        AND c1.species_id = 1
        AND c2.species_id = 1;
```

It is essentially performing a join on four tables. The same can be seen in the single query from XrefTypes:

```
SELECT x.external_db_id, ox.ensembl_object_type,
    COUNT(*), e.db_name
        FROM object_xref ox, external_db e, xref
            x
        LEFT JOIN transcript t ON t.
            display_xref_id = x.xref_id
                WHERE x.xref_id = ox.xref_id
                AND e.external_db_id = x.
                    external_db_id
                AND isnull(transcript_id)
GROUP BY x.external_db_id,ox.ensembl_object_type;
```

Looking at the MySQL query explainer (see Figures 4 & 5), this is confirmed. Further on in section 4.2, we will discuss the possibilities that integrity simplification provide for both these queries. Due to time constraints it was not possible to look at query optimization on the queries themselves. However, it is believed that the speed up obtained by implementing effective change detection is far more effective and faster to implement than what can be gained by optimization of individual queries.

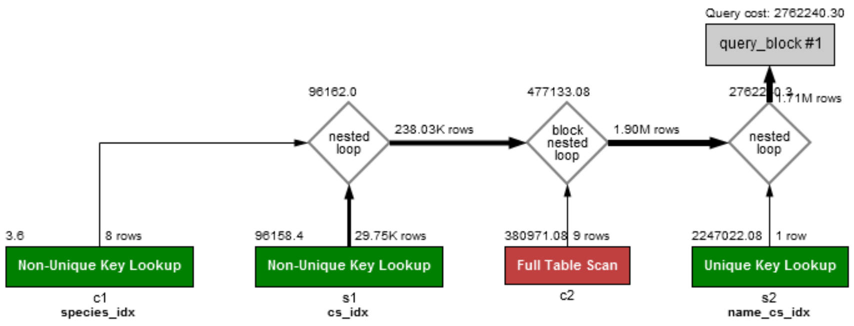


Figure 4: Output of the MySQL query explainer for the query in the SeqRegionCoordSystem check_lengths subroutine. The three tables joined to c1 (**coord_system**) are, from left to right, s1 (**seq_region**) with a non-unique key lookup, c2 (**coord_system**) with a full table scan, and s2 (**seq_region**) with a unique key lookup.

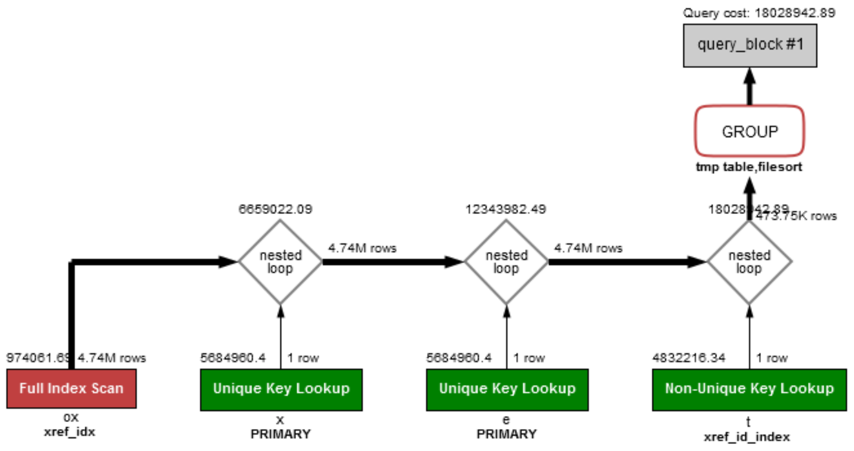


Figure 5: Output of the MySQL query explainer for the query from XrefTypes. The tables that are joined to ox (**objext_xref**) are x (**xref**), e (**external_db**) and t (**transcript**). After the joins a group operation is performed.

How big the gain of change detection can be is illustrated by the results in Table 2. Running the HealthCheckSuite and running all healthchecks in batch shows that they do not differ significantly in runtime (both take about 29 minutes, where differences can be attributed to background processes on both the VM and the MySQL server). When the mock patch update is applied, the Healthcheck suite runs more than 7 times faster, because it only runs healthchecks that have been affected by the update. Most importantly, it filters out the expensive constraints in CoreForeignKey in order to only check those that involve the affected tables. On the other hand, the healthchecks run in batch would take the same amount of time because what has changed in the database is not taken into consideration. It is clear that using change detection, even in this basic form, can greatly increase the efficiency of the healthcheck system. It should be noted that the healthchecks that have been developed in Perl at this moment cover but a fraction of the complete set of checks needed. Still, the same principle applies and the HealthCheckSuite has been developed to include new healthchecks with ease. This is because the information that the suite uses to determine which healthchecks should be run is contained in a separate module in the Input namespace. If a new healthcheck is developed, all that needs to be done in order to include it in the HealthCheckSuite is to add the relevant information (i.e. the name, what tables are involved, etc.) to this Input::HealthChecks. There is, however, no mechanism in place that checks if a healthcheck has not been run before (the only thing that is checked is if the suite in its entirety has been run before or not). This means that if a new healthcheck is added during the testing of a certain database, this healthchecks needs to be run separately to ensure it is satisfied. Else, if none of the involved tables change in the rest of the process, the new healthcheck may not be run at all.

4.2 Investigating Simplification of Integrity Constraints

The reader may have noticed that simplification of integrity constraints as discussed by Bernstein [21] and by Nicolas [20] have been applied to constraints of the healthchecks, but that the application of Christiansen and Martinenghi's method [24] [23] has not been discussed. The reason is as follows: because their work mainly focusses on deductive databases, the integrity constraints are an integral part of the definition of the database. A large part of their work considers the extraction of separated integrity constraints from this definition to produce the more specialised weakest precondition. In relational databases like those of Ensembl, the (integrity) constraints are defined separately from the database (though, in fact some constraints can be implemented in the definition; this will be discussed in section 4.5). Because of this, checking the satisfaction of these constraints is naturally separated in several steps (these steps are the various healthchecks). With the implementation of the change detection system, the vast amount of specialization of constraints for the weakest precondition is already done. Further specialisation of constraints in order to obtain the *optimal weakest precondition* can be done with the methods described in [20]. It should be noted that if we compare the integrity simplification done through the method of [20] in our relational databases, we should speak of the *(optimal) weakest postcondition*, which is also discussed in [24], because in the current system, integrity constraints are checked after an update is applied, not before as Christiansen and Martinenghi suggest. The possibility of checking constraints before an update is applied will also be discussed in the section 4.5.

Both Nicolas [20] and Bernstein and Blaustein [21] consider relational databases and separated constraints applied to relational databases. As described earlier, Nicolas' method works on limiting the number of constraints that are checked for a given update. In turn, Bernstein and Blaustein optimize the actual checking of the constraint by rewriting it to a selection clause involving only the data that has been inserted. First, the results of applying Bernstein and Blaustein's method will be discussed.

After a mock patch update was applied to the database, three methods were used to check integrity constraints as described in section 3.3. Results from method 1 show how long this would take in the current system, which includes the change detection. Method 2 serves as a control and uses the same query structure as method 1, but only checks the foreign keys that have been inserted by using the audit table instead. Method 3 is the constraint simplified following the method in [21]. However, the results are not as expected: method 2 and 3 do not differ in time significantly, and lie well within one another's standard deviation. Both methods are about one hundred times faster than the traditional method 1, so it is clear that exploiting update information can greatly improve the healthcheck system. But why do these two different methods give such similar results?

The first explanation could be that the MySQL optimizer has recognized that one of the two queries (or a third, different one) is logically equivalent but much faster, and therefore chooses this optimal execution plan for both of the queries. However, making use of the MySQL query explainer we can quickly rule out this possibility. It shows that both of the queries have completely different execution paths (see Figures 6 & 7). As expected, method 2 performs a join by performing a full table scan on **assembly_audit** and then looks up the corresponding key in **seq_region**. The number of entries in **assembly_audit** that do not correspond to any key in **seq_region** are then returned to the user. The most expensive parts of this query are the full table scan on the audit table, and, if the audit table is very large, the many iterations over **seq_region** to find the corresponding key. However, for this update all the audit tables contain fewer than 400 rows, so this should not be the deciding factor.

In method 3, the subquery looks up all the keys in **seq_region** using the index and joins it with the **assembly_audit** table. This last selection is identical for both queries, where a full table scan is performed on the **assembly_audit** joined with the results of the previous steps. Now we can also see that even though the query for method 3 contains no explicit joins, a join is still used by the MySQL engine. In their work, Blaustein and Bernstein consider insertions of only one row. In this case, a selection clause is suitable and undeniably faster than a join because only one row (the one that was inserted) needs to be looked up. However, in reality, many rows can be added at once, and that makes joins inevitable if verifying the constraint with one query is desired. In fact, using joins in this context is likely to be faster than to send multiple queries with selection clauses to the server.

Still, the execution paths of the two methods are different yet yield almost identical execution times. It is possible that in this case the execution time is dominated by the time it takes to load the tables or indexes from disk to memory. Since the audit tables are small, it could be that differences in execution

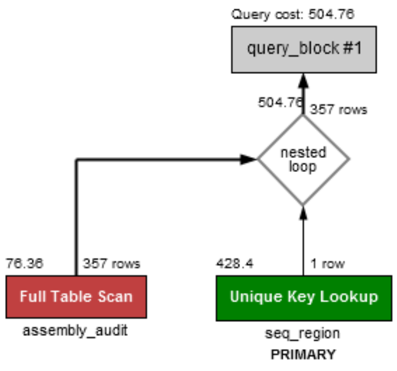


Figure 6: MySQL execution path of method 2

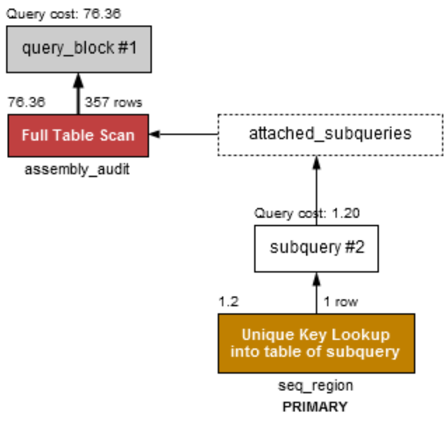


Figure 7: MySQL execution path of method 3

paths are negligible compared to the time it takes to load the data. Because of this, it was decided to run methods 2 and 3 again, but on an update that inserted all the data in an empty table. In table 3 it can be seen that the average execution times lie further apart, but fluctuated considerably, resulting in a large standard deviation. As a result, no conclusive statement on which query is faster can be made yet. However, we can attempt to make some observations based on the average execution times of individual queries. In the patch update, differences in runtime between methods on individual queries could all be accounted for with the standard deviation in all cases (see appendix A). In the case of the complete table update, four queries showed a significant difference between methods (see appendix B). Two queries were significantly faster in method 2; these were the **dna**.seq_region_id → **seq _region**.seq_region_id and **seq_region_attrib**.seq_region_id → **seq_region**.seq_region_id foreign key constraints. The queries to check the **seq_region**.coord_system_id → **coord_system** .coord_system_id and the **assembly**.cmp_seq_region_id → **seq_region**.seq_region_id constraints were faster for method 3.

However, these results still do not give us any conclusive answers. It might be hypothesised that method 3 is slowed down on the one hand because it loads the complete table in the subquery, and on the other hand saves time by then using indexes on this table. In this case it would be expected to see it perform well if the referred table is small, because then loading it in the subquery takes less time. This seems to be supported for the **seq_region**.coord_system_id → **coord_system**.coord_system_id constraint, which is evaluated faster with method 3. The **coord_system** table only contains 9 rows and is the smallest table involved in the constraints that have been tested. However, method 3 is also almost twice as fast as method 2 for the constraint **assembly**.cmp_seq_region_id → **seq_region**.seq_region_id. The **seq_region** table contains 267783 rows and is the largest referred table used in the test, which seems to refute the hypothesis. Curiously, the constraint **assembly**.asm_seq_region_id → **seq_region**.seq_region_id, which is evaluated on the same tables, has very similar execution times for both methods. Neither asm_seq_region_id or cmp_seq_region_id are unique keys by themselves, and the former has 133949 unique values and the latter has 134181, so this cannot account for the difference. In fact, there seems to be no consistency in possible reasons why one method is faster than the others in some cases, and not in others. There are many factors that can influence execution time, including the presence of indexes, table size, and the number of primary keys, and all of these may interact differently on different tables, even with the same query. Furthermore, the MySQL query optimiser is a highly complex piece of software and considers many different execution plans [36]. Even so, the cost estimates it uses to determine the execution plan are not always a correct prediction for the actual cost of the query. Furthermore, the query with the lowest computational cost is not always the fastest as well [37]. This is illustrated by the diagrams MySQL produced for method 2 and method 3 (see figure 6 & 7). It estimated that method 2 is over five times more expensive than method 3, yet the actual execution times are largely similar.

It can be concluded that utilizing update knowledge as proposed by Bernstein and Blaustein [21] provides a very high speed up. However, their suggestions on rewriting queries are not very valuable when considering insertions of multiple rows. The MySQL optimizer has become very sophisticated and as a result it has been demonstrated that rewriting the query does not produce significantly different execution times.

Currently, no infrastructure is in place that enables the HealthCheckSuite to exploit the results obtained in section 2.8. However, it can be stated with confidence that this would increase efficiency in a similar fashion as the current change detection system does, as it essentially acts as a further specialisation of this mechanism. If the change detection system, with this extension, is then combined with optimisation through audit tables, performance could be improved even further. It should be noted that there is one significant disadvantage to the use of audit tables: they increase the size of the database. Considering the number of databases used in production this could have a serious effect on scalability. One way to curb this growth would be to delete the contents of an audit table using a trigger immediately after a change has been successfully committed.

In response to the obtained runtimes of the healthchecks, one might be interested in applying the simplification of Nicolas to the queries from SeqRegion-

CoordSystem and XrefTypes, as they are both very time consuming. However, here the limitations of this method are encountered. In the case of SeqRegion-CoordSystem, the constraint is evaluated on a comparison of two values from the same table. This means that all tables are involved in the antecedent of the conditional, and as a result all occur as negated occurrences when the constraint is rewritten:

$$\begin{aligned}(s1(x, c_1, n_1, l_1) \wedge s2(y, c_2, n_2, l_2) \wedge c_1 \neq c_2 \wedge n_1 = n_2 \\ \wedge c1(c_1, s) \wedge c2(c_2, s)) \rightarrow (l_1 = l_2)\end{aligned} \tag{12}$$

$$\begin{aligned}\neg s1(x, c_1, n_1, l_1 \vee \neg s2(y, c_2, n_2, l_2) \vee c_1 = c_2 \vee n_1 \neq n_2 \vee \\ \neg c1(c_1, s) \vee \neg c2(c_2, s) \vee l_1 = l_2\end{aligned} \tag{13}$$

Where c_i is the (foreign) key for **coord_system**, n_i is the name attribute and l_i the length attribute in **seq_region**, and x and y are primary keys for **seq_region**. Note that all variables are universally quantified, but that the quantifiers have been omitted for readability. Thus, we can see that deletions cannot violate the constraint, but insertions into either tables involved can. This is useful in case deletions take place, but it does not help with regards to insertions.

In some cases applying the method is not even possible. The query of XrefTypes does not evaluate to a true/false value (i.e. empty set means the constraint is satisfied). Instead, it returns a set of values that need to be parsed further. As a result, we cannot apply the method. The method is also not applicable to checks on schema definition such as AutoIncrement, or for database comparisons like CoordSystemAcrossSpecies.

The method as proposed by Nicolas [20] is very promising as it can further specialise the change detection system to increase efficiency. However, it cannot be applied in all healthchecks, which means that some time consuming constraints still need to be evaluated every time one of the involved tables has changed. Still, by extending the change detection system with this technique considerable time can be saved when running the health check suite. Like [21], Nicolas also suggests exploiting the update information in order to check only what has changed. This can be applied through audit tables, as shown before for [21].

4.3 Shortcomings of the New System

The current system lacks the infrastructure to apply integrity simplification with the method of [20]. This means possible time saving methods are not utilized. Because the healthcheck system will be much larger once completed, and the amount of databases that need checking will only grow, making the system as efficient as possible is very important.

Though the Connect module is already in place to make running healthchecks easier, it is still a somewhat limited system. As discussed before, the format is not suitable for all types of healthchecks. Moreover, its flexibility from the command line is limited to specifying species and database type for retrievals from the registry. The user needs to go into the configuration file in order to change hosts or ports, for example. This can be tedious if the user wants to run checks on databases that are on different servers.

Another limitation to the current system is that there is no computer parsing of the output. This is manageable for now because of the limited number healthchecks. However, in the future, when the number of healthchecks has increased, this would make investegating the results from the HealthCheckSuite very difficult, as a user would need to scroll through the output to see whether individual checks have failed or not.

Due to time constraints the codebase has no test suite that checks if the modules behave as they should. This is identified as the most severe shortcoming of the project in its current state, as there is now no efficient, reliable way of checking continued functionality when changes are made to the codebase. As a result, errors that are a 'side effect' of code changes may go unnoticed. This risk can become particularly problematic when the codebase is handed over, because future developers and users have not been involved in project so far and therefore may not have a complete understanding of the project. To account for this problem, it has been made sure that each module and script has extensive documentation describing its behaviour. This information should be enough to allow future developers to create a test suite.

4.4 Further Development

One of the most apparent and interesting extensions in line with the research would be to develop the change detection infrastructure further so that it can also involve the simplification by Nicolas' method as described above. The information_tables that are currently used to keep track of changes are not suitable for this because they do not contain information on the nature of the update (whether it was an insertion, deletion, or alteration). This problem could be resolved by adding triggers and an audit table to hold this information. Moreover, the healthcheck objects would need to be extended in order to distinguish between the consequences of an insertion or deletion of a table.

Another way in which the healthcheck object could be extended is to add an attribute that acts as a flag for importance. For example, it could be that if this property is set to true, it indicates that this healthcheck must pass successfully for a database to be valid. Healthchecks that have the flag set to false may indicate problems in the database, but these are not critical. This could be further extended in the infrastructure in several ways. For example, an option could be added to the HealthCheckSuite to tell it to only run critical healthchecks. Or the property could be used to filter out noise output from the Logger object, in order to isolate the critical results.

To provide an outlook in terms of practical further development, Staines [38] has reviewed the codebase in its current state. Some suggested improvements and extensions include further modularisation (i.e. making checks into modules) in order to allow different inputs (from a database connection to a species name) on a scripting level, and to make maintainability easier. It is also suggested that the messages the logger now prints out are saved in the object so at the end of testing this object can be returned to the user, which should make long term storage (i.e. in a database) of results easier. Another suggestion is that TAP (Test Anything Protocol, see `https://testanything.org/`), which is often used for code testing, is used as a format for healthchecks as well.

These suggestions are quite specific to the project in its current state. However, how the project will evolve from here is very hard to predict. Ultimately, it will be an extensive system applied at multiple stages of an Ensembl release, and it is impossible to anticipate at this stage, and as a relative outsider to the Ensembl project, what future requirements will appear.

4.5 Other Possibilities

One important option that has not been considered is the possibility to check constraints using triggers. This could eliminate the need of a separate test suite as the constraints are then enforced by the database. Because of the strict definition of triggers (i.e. they need to be declared separately for insertions, updates, and deletions) simplification through the method of Nicolas can be implemented with ease. Because triggers only act when the table is changed, change detection is present by definition. Moreover, by setting triggers to evaluate before the change is actually applied, the weakest precondition as suggested by Christiansen and Martinenghi [24] [23] is implemented, and data that violates constraints is never introduced in the database.

However, triggers also have several disadvantages. Because triggers need to be defined for every different type of change, it is important that this is done correctly. Adding new constraints will be complicated. Additionally, only one trigger per change type per moment (before/after an update) is permitted. As a result, all constraints that apply to a table in the case of an insertion need to be defined in one trigger. Because of this it will be hard to maintain oversight. It also means that no distinction can be made between critical and non-critical checks.

Moreover, using triggers to check constraints before the update is applied, as suggested above, will lead to problems for circular constraints. For example, in Ensembl these occur as foreign key constraints that need to hold in both directions. This may produce deadlocks where, for instance, **TABLE1** cannot be updated because the referenced key in **TABLE2** is not present, but **TABLE2** cannot be updated for this key because the referenced key in **TABLE1** is not present.

Additionally, not all constraints will be suitable for checking through triggers. For example, XrefTypes requires further parsing of results, healthchecks that compare two or more databases cannot be replaced with triggers, and checks on schema definition like AutoIncrement would need to respond to changes in the database definition, not tables.

Another possibility would be to use table definitions to enforce foreign key constraints. The database engine Ensembl currently employs is MyISAM, which does not allow for this. The InnoDB engine does [39], but switching database engines would be a big change. Additionally, problems with circular constraints and inflexibility to changes would also occur.

5 Conclusion

The research project has provided a good proof of concept for the new healthcheck system. It has met almost all of the requirements that were laid out for it: the healthchecks are independent from external software save for the Perl

Moose module, which can be considered standard. With a set of new modules, the addition of new healthchecks in the future has been made easy. The health-checks can be run both independently and in batch. Moreover, when they are run with the HealthCheckSuite, the change detection makes sure only relevant healthchecks are run, which results in a faster and more efficient system. There are two requirements that it does not meet: there is currently no infrastructure in place that parses the output in order to provide more oversight for the user, and there is no mechanism yet to distinguish between tests that absolutely need to be satisfied, and those that may be ignored in certain cases.

Furthermore, research into the application of integrity simplification has shown what measures are practical. Using audit tables to keep track of changed data has proven to be a relatively easy and efficient way to speed up simple queries such as integrity constraints. This can be extended to include the method of Nicolas [20] so that checks are only run when tables have changed in such a way that the constraint that is being tested is possibly violated. It has been shown that the method of rewriting constraints as proposed by Bernstein and Blaustein [21] does not actually speed up integrity checking when multiple rows are inserted. It has also been pointed out that using triggers to test constraints before updates as proposed by Christiansen and Martinenghi [24] [23] is a very promising alternative for checking certain constraints, but that due to the limited flexibility of triggers it may also become a mechanism that is very hard to oversee. In the end, the research has provided an investigation into the suitability of the various simplification methods and has thus created an important stepping stone for future development. Implementing these methods in the new healthcheck system may greatly improve efficiency and thus allow the system to scale gracefully with the growth of Ensembl's data.

6 Reflection

I have learnt a lot during my time at the EBI, both in the field of bioinformatics, but also in terms of soft skills. What I really appreciate is the chance I had to look 'behind the scenes' at a project like Ensembl. For me this has really highlighted the collaborative nature of scientific research.

In terms of the research, there are a few things that, in hindsight, I could have done better. It was quite ambitious to decide to both do the development of the new system and the integrity simplification. As a result, the investigation in either subjects was not as in depth as it could have been. On the other hand, I think the work I have done on integrity simplification can be very valuable in the future development of the system. Additionally, I think I could have worked more efficiently if I had used the expertise of the people around me more. In the balance of researching independently and asking questions, I think I relied too much on the former. Because of this I have learnt a lot, but it also meant I have pursued ideas that turned out to be dead ends, and as a result lost time to work on my project. I believed this could have been prevented by asking questions sooner rather than later. Still, I think the MSC has prepared me well to work independently for my thesis research. I think that project periods in particular have been a very good preparation in this regard.

In the end, I think my time at Ensembl has taught me how high level research and development at a non-academic institution works. I have learnt important skills and practices in applied computer science, such as best practices for maintainability, scalability and benchmarking. Moreover I have learnt a programming language that was completely new to me at the start of the project. I think the overall experience I had here will be very valuable in my future career and life.

References

[1] Fenstermacher D. Introduction to Bioinformatics. Journal of the American Society for Information Science and Technology. 2005;56:440–446.

[2] Scheibye-Alsing K, Hoffman S, Frankel A, Jensen P, Stadler PF, Mang Y, et al. Sequence Assembly. Computational Biology and Chemistry. 2005;33:121–136.

[3] Hagen JB. The Origins of Bioinformatics. Nature Reviews Genetics. 2000;1:231–236.

[4] Vandepitte L, Bosch S, Tyberghein L, Waumans F, Vanhoorne B, Hernandez F, et al. Fishing for Data and Sorting the Catch: Assessing the Data Quality, Completeness and Fitness for Use of Data in Marine Biogeographic Databases. Database. 2015;2015:1–14.

[5] Marcus DS, Harms MP, Snyder AZ, Jenkinson M, Wilson JA, Glasser MF, et al. Human Connectome Project Informatics: Quality Control, Database Services, and Data Visualization. NeuroImage. 2013;80:202–219.

[6] Csordas A, Ovelleiro D, Wang R, Foster JM, Rios D, Vizcaino JA, et al. PRIDE: Quality Control in a Proteomics Data Repository. Database. 2012;2012.

[7] Cook CE, Bergman MT, Finn RD, Cochrane G, Birney E, Apweiler R. The European Bioinformatics Institute in 2016: Data Growth and Integration. Nucleic Acids Research. 2016;44:20–26.

[8] Yates A, Akanni W, Amode MR, Barrell D, Billis K, Carvalho-Silva D, et al. Ensembl 2016. Nucleic Acids Res. 2016;44:710–716.

[9] Beagrie N, Houghton J. The Value and Impact of the European Bioinformatics Institute [Report]; 2016. Available from: `http://www.beagrie.com/static/resource/EBI-impact-report.pdf`.

[10] EMBL-EBI. Perl API Documentation [Online Documentation]; 2015 [cited 2016 Feb 16]. Available from: `http://www.ensembl.org/info/docs/api/index.html`.

[11] Kersey PJ, Allen JE, Armean I, Boddu S, Bolt BJ, Carvalho-Silva D, et al. Ensembl Genomes 2016: More Genomes, More Complexity. Nucleic Acids Research. 2016;44:574–580.

[12] Church DM, Schneider VA, Graves T, Auger K, Cunningham F, Bouk N, et al. Modernizing Reference Genome Assemblies. PLoS Biology. 2011;9.

[13] EMBL-EBI. What is a Genome Assembly? [Online Documentation]; 2016 [cited 2016 Feb 17]. Available from: `http://www.ensembl.org/Help/Faq?id=216`.

[14] Potter SC, Clarke L, Curwen V, Keenan S, Mongin E, Searle SMJ, et al. The Ensembl Analysis Pipeline. Genome Research. 2004;14:934–941.

[15] Stabenau A, McVicker G, Melsopp C, Proctor G, Clamp M, Birney E. The Ensembl Core Software Libraries. Genome Research. 2004;14:929–933.

[16] Elmasri R, Navathe SB. Fundamentals of Database Systems. 6th ed. Boston, MA: Pearson (Addison-Wesley); 2010.

[17] Codd EF. A Relational Model of Data for Large Shared Data Banks. Communications of the ACM. 1970;13:377–387.

[18] Codd EF. Extending the Database Relational Model to Capture More Meaning. ACM Transactions on Database Systems. 1979;4:397–434.

[19] Magnus PD. forall x - An Introduction to Formal Logic [eBook]; c2005 - 2014 [cited 2016 Apr 27]. Available from: `http://www.fecundity.com/logic`.

[20] Nicolas JM. Logic for Improving Integrity Checking in Relational Data Bases. Acta Informatica. 1982;18:227–253.

[21] Bernstein PA, Blaustein BT. Fast Methods for Testing Quantified Relational Calculus Assertions. In: Proceedings of the 1982 ACM SIGMOD international conference on Management of Data. New York: ACM; 1982. p. 39–50.

[22] Gallaire H, Minker J, Nicolas JM. Logic and Databases: A Deductive Approach. Computing Surveys. 1984;16:153–185.

[23] Christiansen H, Martinenghi D. On Simplification of Database Integrity Constraints. Fundamenta Informaticae. 2006;71:371–417.

[24] Christiansen H, Martinenghi D. Simplification of Integrity Constraints for Data Integration. In: Seipel D, Turull-Torres JM, editors. Foundations of Information and Knowledge Systems: Third International Symposium, FoIKS 2004 Wilheminenburg Castle, Austria, February 17-20, 2004 Proceedings. Berlin, Heidelberg: Springer Berlin Heidelberg; 2004. p. 31–48. Available from: `http://dx.doi.org/10.1007/978-3-540-24627-5_4`.

[25] Martinenghi D, Christiansen H, Decker H. Integrity Checking and Maintenance in Relational and Deductive Databases and Beyond. In: Ma Z, editor. Intelligent Databases: Technologies and Applications. Hershey, PA: Idea Group Piblishing; 2007. p. 238–285.

[26] Orman LV. Differential Relational Calculus for Integrity Maintenance. IEEE Transactions on Knowledge and Data Engineering. 1998;10:328–341.

[27] Homeier PV. Simplifying Integrity Constraints in a Relational Database: an Implementation [Master thesis]. University of California. Los Angeles, CA; 1981.

[28] Poe CO. Beginning Perl. Indianapolis, IN: John Wiley and Sons; 2012.

[29] chromatic. Modern Perl. 4th ed. Swaine M, editor. Dallas, TX: The Pragmatic Bookshelf; 2015.

[30] Little S, Rolsky D, Luehrs J, Moore SM, Kogman Y, Etheridge K, et al.. Perl Moose Module [Software module]; 2006. Available from: `http://search.cpan.org/~ether/Moose-2.1802/lib/Moose.pm`.

[31] Vromans J. Perl Getopt::Long Module [Software module]; 2015. Available from: `http://search.cpan.org/~jv/Getopt-Long-2.48/lib/Getopt/Long.pm`.

[32] Williams K, Albanowski K, Dougherty A, Köning A, Bunce T. Perl File::Spec Module [Software module]; 2013. Available from: `http://search.cpan.org/~rjbs/PathTools-3.62/lib/File/Spec.pm`.

[33] Sarathy G. Perl Data::Dumper Module [Software module]; 2014. Available from: `http://search.cpan.org/~smueller/Data-Dumper-2.154/Dumper.pm`.

[34] Bunce T. Perl Devel::NYTProf Module [Software module]; 2016. Available from: `http://search.cpan.org/~timb/Devel-NYTProf-6.03/lib/Devel/NYTProf.pm`.

[35] Oracle. Using Triggers [Online Documentation]; 2016 [cited 2016 Jun 4]. Available from: `http://dev.mysql.com/doc/refman/5.7/en/triggers.html`.

[36] Oracle. Understanding the Query Execution Plan [Online Documentation]; 2016 [cited 2016 Jun 2]. Available from: `https://dev.mysql.com/doc/refman/5.7/en/execution-plan-information.html`.

[37] Baron Schwartz VT Peter Zaitsev. Query Performance Optimization. In: Oram A, editor. High Performance MySQL. 3rd ed. Sebastopol: O'Reilly; 2012. p. 201–264.

[38] Staines D. edenbrok code review [Internal Documentation]; 2016 [cited 2016 Jun 4]. Available from: `https://www.ebi.ac.uk/seqdb/confluence/display/GTI/edenbrok+code+review`.

[39] Oracle. InnoDB and FOREIGN KEY Constraints [Online Documentation]; 2016 [cited 2016 Jun 2]. Available from: `http://dev.mysql.com/doc/refman/5.7/en/innodb-foreign-key-constraints.html`.

A Appendix: Patch update

Method 1 - Patch update
Total runtime (s)
31
29.3
40.7
33.5
22.8
15.55
17.8
14.9
24.5
23.3
25.335

Total query time (µs)	SD
28403086	3.68E+13
25947775	1.31E+13
37813683	2.40E+14
30829805	7.22E+13
20174734	4.67E+12
12510474	9.65E+13
13144720	8.45E+13
12231810	1.02E+14
21685840	4.21E+11
20607169	2.99E+12
22334909.6	8079329.9

all in µs

assembly.asm_seq_region_id	SD	assembly.cmp_seq_region_id	SD
12350038	1.62E+13	15100034	2.91E+12
6220036	4.43E+12	18800033	2.92E+13
18100034	9.56E+13	18800017	2.92E+13
19500037	1.25E+14	10400032	8.96E+12
12000031	1.35E+13	7230042	3.80E+13
3120038	2.71E+13	9000031	1.93E+13
2970037	2.87E+13	9880030	1.23E+13
2910039	2.93E+13	9030040	1.90E+13
3020037	2.81E+13	18400032	2.51E+13
3050038	2.78E+13	17300033	1.53E+13
8324036.5	6289891.2	**13394032.4**	4464700.8

assembly_exception.seq_region_id	SD	assembly_exception.exc_seq_regior	SD
18642	1.06E+08	2069	3.85E+05
3865	2.00E+07	1220	5.22E+04
3120	2.72E+07	1207	5.83E+04
3056	2.79E+07	1116	1.11E+05
19937	1.35E+08	2712	1.60E+06
2028	3.98E+07	956	2.43E+05
15033	4.49E+07	2166	5.15E+05
14834	4.22E+07	1279	2.87E+04
1418	4.78E+07	849	3.59E+05
1408	4.80E+07	911	2.89E+05
8334.1	7338.9	**1448.5**	603.1

dna.seq_region_id	SD	seq_region.coord_system_id	SD
786054	9.87E+10	115054	1.26E+06
796030	1.05E+11	114031	9.53E+03
782030	9.62E+10	114031	9.53E+03
797029	1.06E+11	114031	9.53E+03
780035	9.49E+10	113032	8.13E+05
158026	9.85E+10	178032	4.11E+09
154016	1.01E+11	96630	2.99E+08
155017	1.00E+11	98831	2.28E+08
159025	9.79E+10	98330	2.43E+08
152025	1.02E+11	97332	2.76E+08
471928.7	316351.4	**113933.4**	22709.7

seq_region_attrib.seq_region_id	SD	seq_region_attrib.attrib_type_id	SD
29134	2.22E+08	2061	2.41E+07
9611	2.13E+07	2949	1.61E+07
9121	2.61E+07	4123	8.08E+06
9131	2.60E+07	5373	2.54E+06
25432	1.25E+08	3513	1.19E+07
32031	3.17E+08	19332	1.53E+08
3755	1.10E+08	23053	2.59E+08
17950	1.38E+07	3820	9.90E+06
3224	1.21E+08	2925	1.63E+07
2914	1.28E+08	2508	1.99E+07
14230.3	10538.9	**6965.7**	7214.9

Method 2 - Patch update
Total runtime (s)
3.3
2.93
2.96
2.99
4.13
4.18
2.86
4.07
3.32
3.02
3.376

Total query time (µs)	**SD**
205291	1.48E+08
204248	1.74E+08
189811	7.64E+08
144116	5.38E+09
338920	1.48E+10
292640	5.65E+09
137576	6.38E+09
150422	4.49E+09
274255	3.23E+09
237190	3.90E+08
217446.9	64312.9

all in µs

assembly.asm_seq_region_id	**SD**	**assembly.cmp_seq_region_id**	**SD**
189036	1.93E+08	1404	2.29E+04
189048	1.93E+08	1350	9.47E+03
177039	6.71E+08	1360	1.15E+04
131037	5.17E+09	2150	8.05E+05
307034	1.08E+10	1124	1.66E+04
267038	4.11E+09	1036	4.70E+04
130035	5.31E+09	925	1.07E+05
142037	3.71E+09	1116	1.87E+04
266036	3.98E+09	1055	3.91E+04
231037	7.90E+08	1007	6.04E+04
202937.7	59132.2	**1252.7**	337.4

assembly_exception.seq_region_id	**SD**	**assembly_exception.exc_seq_regior**	**SD**
1212	1.35E+03	821	2.87E+03
1460	8.11E+04	1675	6.41E+05
1458	7.99E+04	876	1.96E+00
1358	3.34E+04	820	2.98E+03
1162	1.77E+02	730	2.09E+04
1005	2.90E+04	758	1.36E+04
832	1.18E+05	807	4.57E+03
1372	3.87E+04	739	1.84E+04
1101	5.52E+03	822	2.77E+03
793	1.46E+05	698	3.12E+04
1175.3	230.9	**874.6**	271.6

dna.seq_region_id	SD	seq_region.coord_system_id	SD
1168	9.76E+03	1309	2.92E+06
1276	4.28E+04	1526	2.23E+06
1184	1.32E+04	1321	2.88E+06
1034	1.24E+03	1399	2.62E+06
942	1.62E+04	18523	2.40E+08
1017	2.72E+03	1227	3.21E+06
1091	4.75E+02	1279	3.03E+06
1006	3.99E+03	1417	2.57E+06
1210	1.98E+04	1370	2.72E+06
764	9.31E+04	818	4.84E+06
1069.2	142.6	**3018.9**	5171.1

seq_region_attrib.seq_region_id	SD	seq_region_attrib.attrib_type_id	SD
7679	2.68E+07	2662	3.82E+06
4458	3.83E+06	3455	1.35E+06
2280	4.88E+04	4293	1.05E+05
2598	9.41E+03	3720	8.06E+05
1736	5.85E+05	7669	9.31E+06
1227	1.62E+06	19332	2.17E+08
1298	1.45E+06	1309	1.09E+07
1318	1.40E+06	1417	1.02E+07
1369	1.28E+06	1292	1.11E+07
1047	2.11E+06	1026	1.29E+07
2501	1978.6	**4617.5**	5263.7

Method 3 - Patch Update
Total runtime (s)
7.38
3.04
4.66
4.44
2.94
3.22
2.93
3.02
3.74
3.55
3.892

Total query time (µs)	**SD**
398009357.6	1.46E+18
164355467	2.08E+18
3231458265	2.64E+18
3488351150	3.54E+18
1631177282	6.32E+14
2385640315	6.08E+17
411888403.8	1.43E+18
3486561899	3.54E+18
144951609.2	2.13E+18
717978568.8	7.89E+17
1606037232	**1.3497E+09**

all in µs

assembly.asm_seq_region_id	**SD**	**assembly.cmp_seq_region_id**	**SD**
172021	3.46E+08	4180	3.72E+06
201020	1.08E+08	2756	2.56E+05
247020	3.18E+09	2938	4.73E+05
132019	3.43E+09	2767	2.67E+05
218019	7.51E+08	1036	1.47E+06
231018	1.63E+09	1957	8.61E+04
209020	3.39E+08	750	2.25E+06
132018	3.43E+09	1167	1.17E+06
199021	7.06E+07	1166	1.18E+06
165020	6.55E+08	3787	2.36E+06
190619.6	37350.4	**2250.4**	1150.7

assembly_exception.seq_region_id	**SD**	**assembly_exception.exc_seq_regior**	**SD**
1405	7.73E+04	792	1.15E+05
801	7.78E+05	704	1.82E+05
1093	3.48E+05	835	8.74E+04
2966	1.65E+06	771	1.29E+05
1021	4.38E+05	645	2.36E+05
1074	3.71E+05	444	4.72E+05
6269	2.10E+07	610	2.71E+05
806	7.69E+05	362	5.91E+05
838	7.14E+05	5591	1.99E+07
558	1.27E+06	553	3.34E+05
1683.1	1656.5	**1130.7**	1493.7

dna.seq_region_id	**SD**	**seq_region.coord_system_id**	**SD**
1293	1.41E+05	1433	4.56E+07
778	1.94E+04	813	5.43E+07
1072	2.40E+04	1222	4.85E+07
1022	1.10E+04	1011	5.15E+07
920	7.84E+00	37812	8.78E+08
898	3.69E+02	35612	7.52E+08
897	4.08E+02	1171	4.92E+07
806	1.24E+04	1171	4.92E+07
922	2.30E+01	985	5.18E+07
564	1.25E+05	609	5.74E+07
917.2	182.6	**8183.9**	14274.1

seq_region_attrib.seq_region_id	**SD**	**seq_region_attrib.attrib_type_id**	**SD**
3562	2.28E+06	1432	2.52E+04
2692	4.11E+05	969	3.86E+05
2942	7.94E+05	1142	2.01E+05
2871	6.73E+05	941	4.22E+05
1593	2.09E+05	6261	2.18E+07
1725	1.06E+05	1102	2.39E+05
1452	3.58E+05	1091	2.50E+05
1310	5.49E+05	1008	3.39E+05
1310	5.49E+05	1040	3.03E+05
1050	1.00E+06	920	4.50E+05
2050.7	832.8	**1590.6**	1562.9

B Appendix: Full table update

Method 2 - Full Table Update
Total runtime (s)
5.66
8.24
13.5
8.31
7.57
8.78
7.57
10
7.62
9.59
8.684

Total query time (µs)	**SD**
2724748	2.98E+12
4062422	1.51E+11
5513071	1.13E+12
4317827	1.76E+10
4086652	1.32E+11
4384153	4.40E+09
4402782	2.27E+09
5045930	3.55E+11
4317608	1.77E+10
5649584	1.44E+12
4450477.7	788958.0

all in µs

assembly.asm_seq_region_id	**SD**	**assembly.cmp_seq_region_id**	**SD**
1270036	4.51E+12	1160034	1.19E+11
3050037	1.18E+11	776033	1.47E+09
4450038	1.12E+12	799035	2.37E+08
3310035	6.89E+09	771030	1.88E+09
3080048	9.80E+10	773036	1.71E+09
3380035	1.69E+08	770031	1.97E+09
3390037	9.01E+06	775032	1.55E+09
4050038	4.32E+11	766033	2.34E+09
3310042	6.89E+09	775031	1.55E+09
4640038	1.56E+12	779032	1.25E+09
3393038.4	885472.4	**814432.7**	115508.2

assembly_exception.seq_regi	**SD**	**assembly_exception.exc_seq_region_id**	**SD**
1911	6.43E+04	963	9.18E+02
1863	4.23E+04	961	8.01E+02
1702	1.99E+03	896	1.35E+03
1692	1.20E+03	878	2.99E+03
1640	3.03E+02	918	2.16E+02
1779	1.48E+04	918	2.16E+02
1963	9.34E+04	1112	3.21E+04
1298	1.29E+05	925	5.93E+01
1267	1.52E+05	894	1.50E+03
1459	3.94E+04	862	5.00E+03
1657.4	232.2	**932.7**	67.2

dna.seq_region_id	SD	seq_region.coord_system_id	SD
147026	9.99E+05	119030	1.65E+09
147029	1.00E+06	74131	1.88E+07
147027	1.00E+06	73936	2.05E+07
146024	7.29E+00	74430	1.63E+07
145026	1.00E+06	74031	1.97E+07
145025	1.00E+06	73831	2.15E+07
148032	4.02E+06	73853	2.13E+07
144027	4.00E+06	73732	2.24E+07
146025	2.89E+00	74631	1.47E+07
145026	1.00E+06	73039	2.94E+07
146026.7	1184.5	78464.4	13527.9

seq_region_attrib.seq_region_	SD	seq_region_attrib.attrib_type_id	SD
5320	1.37E+03	20428	9.58E+07
5488	4.20E+04	6880	1.42E+07
5288	2.50E+01	35149	6.01E+08
5447	2.69E+04	8291	5.53E+06
5297	1.96E+02	6656	1.59E+07
5258	6.25E+02	7276	1.13E+07
5737	2.06E+05	7016	1.32E+07
4895	1.51E+05	4982	3.20E+07
4876	1.66E+05	4842	3.36E+07
5224	3.48E+03	4904	3.29E+07
5283	244.3	10642.4	9246.6

Method 3 - Full Table Update
Total runtime (s)
6.86
6.92
11.9
7.72
11.2
9.94
8.32
13.6
7.83
8.55
9.284

Total query time (µs)	SD
3819777	1.75E+12
2887733	5.09E+12
8434491	1.08E+13
4515970	3.95E+11
6158353	1.03E+12
4873943	7.32E+10
5011338	1.77E+10
5785169	4.10E+11
4599871	2.97E+11
5358470	4.58E+10
5144511.5	1412043.6

all in µs

assembly.asm_seq_region_id	SD	assembly.cmp_seq_region_id	SD
3090020	1.76E+12	499028	1.79E+08
2610020	3.27E+12	50227	1.90E+11
7560027	9.88E+12	597018	1.24E+10
3750020	4.45E+11	527022	1.71E+09
5360013	8.89E+11	558004	5.24E+09
4140018	7.67E+10	505019	3.76E+08
4210019	4.28E+10	554019	4.68E+09
5030011	3.76E+11	518027	1.05E+09
3850022	3.21E+11	511018	6.44E+08
4570021	2.34E+10	537020	2.64E+09
4417019.1	1306867.6	**485640.2**	147818.1

assembly_exception.seq_regi	SD	assembly_exception.exc_seq_region_id	SD
1677	2.84E+04	1552	2.62E+02
1639	4.27E+04	932	4.05E+05
1626	4.82E+04	1012	3.09E+05
1797	2.36E+03	1073	2.45E+05
1670	3.08E+04	1033	2.86E+05
1637	4.35E+04	998	3.25E+05
1689	2.45E+04	1012	3.09E+05
1297	3.01E+05	1104	2.15E+05
4076	4.97E+06	5872	1.85E+07
1348	2.48E+05	1094	2.25E+05
1845.6	757.9	**1568.2**	1443.7

dna.seq_region_id	SD	seq_region.coord_system_id	SD
186015	9.60E+07	30920	1.08E+06
183012	1.64E+08	31120	7.05E+05
223014	7.40E+08	40420	7.16E+07
195013	6.40E+05	30718	1.54E+06
196014	4.03E+04	31317	4.13E+05
184012	1.39E+08	31030	8.65E+05
203013	5.18E+07	30918	1.09E+06
193014	7.84E+06	31218	5.50E+05
188012	6.09E+07	30518	2.08E+06
207014	1.25E+08	31419	2.92E+05
195813.3	11771.6	**31959.8**	2831.8

seq_region_attrib.seq_region_	SD	seq_region_attrib.attrib_type_id	SD
7552	8.88E+02	3013	1.69E+04
7558	1.28E+03	3225	6.71E+03
8017	2.45E+05	3357	4.58E+04
7555	1.08E+03	2772	1.38E+05
7127	1.56E+05	3175	1.02E+03
7516	3.84E+01	3713	3.25E+05
7695	2.99E+04	2973	2.89E+04
7296	5.12E+04	3202	3.47E+03
7368	2.38E+04	2985	2.50E+04
7538	2.50E+02	3016	1.62E+04
7522.2	225.7	**3143.1**	246.3