

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

Wilhelm-Schickard-Institut  
Algorithmen der Bioinformatik  
Prof. Dr. Daniel Huson

DIPLOMARBEIT

# Datenbankgestützte Analyse von Metagenomikdaten

Hans-Joachim Ruscheweyh  
2693910

Betreuer I: Prof. Dr. Daniel Huson  
Betreuer II: Prof. Dr. Torsten Grust

Tübingen, den 22. Juli 2010

## **Erklärung**

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ort, Datum

Unterschrift

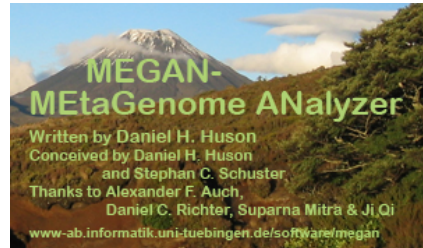
# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
<b>2</b>	<b>Read-Match Archive</b>	<b>9</b>
2.1	Header . . . . .	9
2.2	Data . . . . .	11
2.2.1	ReadBlock . . . . .	12
2.2.2	MatchBlock . . . . .	13
2.3	ClassificationBlock . . . . .	13
2.4	Auxiliary . . . . .	14
<b>3</b>	<b>MEGAN's mode of operation</b>	<b>16</b>
3.1	Opening a file . . . . .	16
3.2	Browsing - The Inspector . . . . .	17
3.3	Creating new datasets . . . . .	18
3.4	(Re)Calculating the classifications . . . . .	18
<b>4</b>	<b>Translation</b>	<b>21</b>
4.1	Entity relationship model . . . . .	22
4.2	From a RMA file format to a database design . . . . .	23
4.3	Translation of the Entity Relationship diagram . . . . .	26
4.3.1	Header and auxiliary . . . . .	27
4.3.2	Read and classread . . . . .	27
4.3.3	Classificationblock . . . . .	27
4.3.4	Match and blasthit . . . . .	27
<b>5</b>	<b>Database candidates</b>	<b>32</b>
5.1	Relational database management system . . . . .	32
5.1.1	Relational model . . . . .	33
5.1.2	Relational algebra . . . . .	33
5.1.3	Relational model versus RMA file . . . . .	34
5.2	Requirements . . . . .	36
5.3	Candidates . . . . .	37
5.3.1	PostgreSQL . . . . .	37
5.3.2	H2 . . . . .	39
<b>6</b>	<b>Implementation</b>	<b>41</b>
6.1	Making MEGAN independent from RMA . . . . .	41
6.1.1	Adaption of present implementation . . . . .	42
6.1.2	IConnector . . . . .	45

6.2	Preliminary database jobs . . . . .	50
6.2.1	Create MEGAN DB . . . . .	50
6.2.2	Injecting MEGAN's tables . . . . .	51
6.2.3	Additional indexes . . . . .	51
6.2.4	Functions . . . . .	53
6.2.5	Getting connected . . . . .	54
6.3	Communication with the database . . . . .	54
6.4	Database implementation of blocks . . . . .	56
6.4.1	ReadBlockDB . . . . .	57
6.4.2	MatchBlockDB . . . . .	59
6.5	DBConnector . . . . .	59
6.5.1	The class structure . . . . .	59
6.5.2	Open a dataset . . . . .	60
6.5.3	Find compatible reads . . . . .	60
6.5.4	Alter data . . . . .	63
6.5.5	Recalculation of classifications . . . . .	64
6.5.6	Creating new datasets . . . . .	65
<b>7</b>	<b>Prospects</b>	<b>67</b>
7.1	Calculation of the taxonomy classification in PostgreSQL . . . . .	68
7.1.1	Preliminaries . . . . .	68
7.1.2	Solving the LCA problem in PostgreSQL . . . . .	69
7.1.3	LCA in MEGAN . . . . .	70
7.1.4	Putting all together . . . . .	71
7.2	pg_bulkload . . . . .	71
7.2.1	COPY on an indexed table . . . . .	72
7.2.2	Deleting Index + COPY + Create Index . . . . .	72
7.2.3	pg_bulkload . . . . .	72
7.3	Supporting database characteristics . . . . .	73
7.4	Refinements on the databases tables . . . . .	73
7.5	Unlimited MEGAN . . . . .	73
7.6	IConnector/Connection handling . . . . .	74
<b>8</b>	<b>Runtime comparison RMA vs. PostgreSQL</b>	<b>76</b>
8.1	Tested datasets and system properties . . . . .	76
8.2	Create a new dataset . . . . .	77
8.3	Disk usage RMA vs. database . . . . .	81
8.4	Text searches . . . . .	81
8.5	Update classification . . . . .	82
<b>9</b>	<b>Conclusion</b>	<b>86</b>
	<b>Appendices</b>	<b>90</b>
.1	SQL code to create MEGAN's tables . . . . .	90
.2	PostgreSQL's function to assure a correct classificationblock table . . . . .	92
.3	Plpgsql implemenation of pre- and postorder ranks for table node . . . . .	93

# 1 Introduction

The Metagenome analyzer (MEGAN) is a Java programme designed to work on data representing metagenomic content. Since the size and the number of the datasets in which the data is arranged are growing, it is of a major interest to discover how MEGAN can profit from a technique well known to be capable of maintaining large sets of data - the database integration. It is also hoped to profit from the flexibility and additional functionality a database can offer.



Up to now the standard data container of MEGAN has been a binary Java file performing generally well, but is rather static when it comes to changes and offers not any additional functionality, which could be beneficial for MEGAN. However, in this thesis it will be shown why and under which circumstances a properly designed and configured database is superior to a Java file.

For the realisation of a database integrated into MEGAN, a couple of steps are required, and these are introduced in this thesis. To understand the structure of the data, the actual solution, the Read Match Archive (RMA), is analyzed in chapter 2. In addition, a short introduction into the semantics of data, which is crucial for a good database design, is given. The RMA file is a compressed Java binary specialised in providing all information queried by MEGAN in a fast manner. Therefore, the implementation of this file is the basis, or rather is taken as a template for the design of a database schema, as described in chapter 4. This is achieved by a technique utilized to translate informal requirements extracted from the RMA file and provided by the analysis of MEGAN's queries into a formal structure, the entity relationship model. The resulting diagram, based on the entity relationship model provides a high comfort to both the application programmer and the database designer by an increased level of the view on the data. Another reason why this representation is chosen is the fact that it includes clearly defined rules to map the diagram to the target structure of a database. Every database management system, that is a system containing and maintaining databases, is based on a data model. Since in this approach the relational data model is used, the target structure is a set of tables.

The relational data model introduced in chapter 5 is said to be superior to other data models [Cod70]. Even though this assumption cannot be proved in this thesis, it is shown that under certain circumstances do database management systems based on the relational model perform better than files in a file system. For example, data within RMA files are identified by a byte offset which makes updates and inserts expensive, whereas a database easily copes with this challenge since it provides a high level of data independence. The facts making the database system more efficient than a file system are pointed out in chapter 5. Additionally, two database management system

candidates are introduced. Both systems are based on the relational model, but have different approaches. H2 is a pure Java database system with a small footprint to be embedded in MEGAN's libraries and is designed to be a pure data storage. Hence is the usage of H2 limited, since only RMA functionality is emulated. However, H2 runs in the same Java virtual machine as MEGAN. Therefore, the overhead which usually appears between data object and application does not occur. The other database management system is PostgreSQL. PostgreSQL combines data storage and a large set of functions. This database system is for that reason far beyond a pure load/storage design. The idea is to outsource as many functions as possible from MEGAN's into the scope of a database. The expected gain is a considerable acceleration and a reduction of overhead, since the operations are executed within the database's memory and no transfer costs from and to MEGAN occur.

The main focus of this thesis is the implementation introduced in Chapter 6. Since MEGAN was based on the usage of RMA files, its source code contained much RMA specific information, such as pointer lists, at positions not suitable for integration of other data access methods, such as databases. Therefore, Prof. Dr. Huson installed an abstraction layer providing interfaces for the common data access, which excludes all RMA specific information. The entire communication from and to the data access object is directed through these interfaces, which also allows other access methods to be integrated later on.

In addition to Java interfaces, also the database has to be made capable. This can be done by creation of users, tables and functions. The tables are directly derived from the entity relationship diagram and refined to the requirements of MEGAN. The main function of MEGAN is to provide summaries of large sets of data. Of course a database can create these summaries on demand, but since the size of the datasets easily exceeds 1GB, it is computationally not feasible to create them in realtime. Therefore, the tables containing derived information are integrated where the functions are needed to keep the consistency between the tables. The last step to integrate a database is the connection between the Java interface and the database implemented.

Besides queries returning small results, such as summaries, MEGAN can also request results of an undefined size returned as an iterator. This is a difficult task for the database due to the expected result structure. MEGAN works mainly on blocks enclosing a set of related information. For a database which splits this information into several tables this means, that a block needs to be generated on demand. For queries requesting a large number of blocks as an iterator, generating complete blocks is not feasible in realtime. Finding a good solution to provide the iterator a block structure coping with the demands of MEGAN is one of the most interesting challenges solved in section 6.5.3.

Unfortunately, the database still works in the load/store mode. But in order to show the possibilities when using the workforce of a database, an example on how to outsource work to the database is provided in chapter 7. In addition, some prospects of not yet implemented ideas to speed up already existing operations are introduced. Since a database is more flexible than a RMA file, some new features for future versions of MEGAN are suggested, such as working on several datasets at the same time without needing to merge them physically.

In order to prove that a database accelerates the application remarkably, runtime comparisons are executed concurrently against RMA files and the database in chapter

8. The set of tested functions shows promising results with different kinds of datasets. It is however clear that only day-to-day usage will show how MEGAN performs and where possible bottlenecks are. The abstraction layer for data access also gives the database designer all possible freedom. Future changes can easily be applied as well.



## 2 Read-Match Archive

The first step relevant to the storage of MEGAN's data is that DNA sequences generated in a metagenomic sample are compared to a database e.g. NCBI<sup>1</sup>. The result is a text file produced by the alignment tool BLAST [AS90]. A BLAST file contains a structured list of the input sequences, its identifiers and the results of the comparison step. Hence every sequence has its own closed set of results, or, in other words, matches. MEGAN's scope begins with parsing the BLAST file and transferring its data into a binary file designed as a storage for this results and data derived from the matches. One information which is derived is the list of classifications. A classification assigns every read<sup>2</sup> a class derived from the information of the associated matches.

The described file is the Read Match Archive (RMA). For a good database design it is essential to understand the needs of MEGAN. Since the RMA file is the standard container for MEGAN's data, the database then targets to the design of RMA files, but deletes most of the structural data, which for example deals with the organization within the file. In order to understand MEGAN's needs at this step, the RMA file is described in detail.

The general structure of a RMA file is depicted in Fig. 2.1. The raw data, extracted from the BLAST file, is stored in the `Data` section. This section is surrounded by structural information, such as the `Header`, `ClassificationBlock` and `Auxiliary`. `Header` maintains global information valid for the whole file, whereas the `ClassificationBlock` represents an index pointing reads in the `Data` section.

### 2.1 Header

The `Header` provides global information. Listing 2.1 shows the fields of the `Header`. The fields `creationDate` and `modificationDate` define dates when the file was created and last changed. `fileSize` counts the size of the file measured in bytes. This is important, since the RMA file allows random access to file positions through a byte offset. `numberOfReads`, `numberOfMatches` and `numberOfClassifications` determine the number of reads, matches and classifications in the file. `beginData` and `endData` in combination with `beginAuxiliary` and `endAuxiliary` contain byte offsets pointing to the top byte and the last byte of the `Data` section, respectively `Auxiliary`. Since more than one classification can be assigned to a file, the `beginClassification` and `endClassification` fields serve as arrays, whose entries point to byte offsets of different `ClassificationBlocks`. Since for each classification is one `ClassificationBlock` created, the length of the array is dependent on the number of active classifications. The byte offsets are therefore used to determine positions of sections in the file in order

---

<sup>1</sup><http://blast.ncbi.nlm.nih.gov>

<sup>2</sup>Synonymly used for sequence and the data stored together with the sequence.

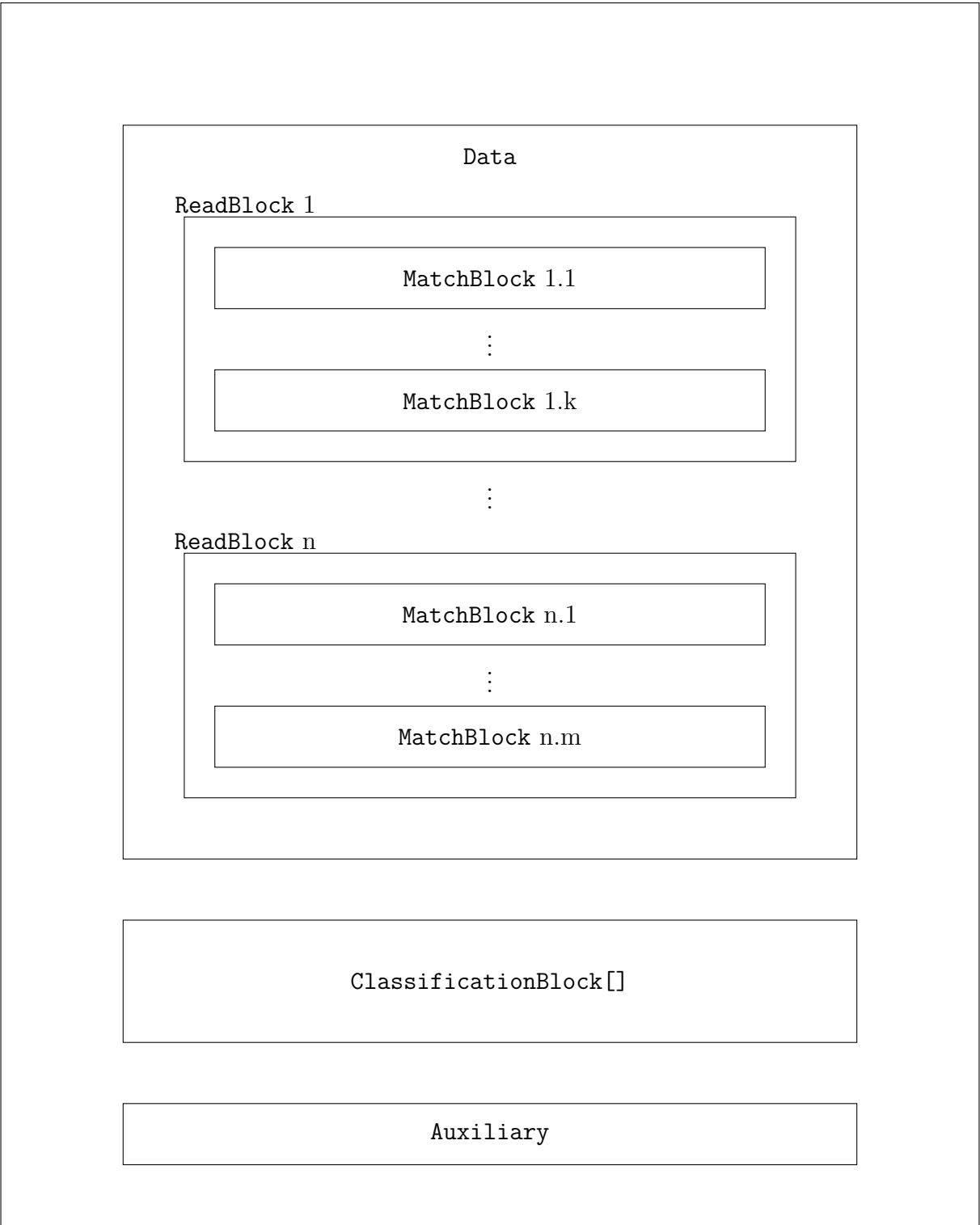


Figure 2.1: The structure of a typical RMA file

to allow random access. `readFormat` and `matchFormat` are Java data types providing information on how the data is stored in `ReadBlocks` and `MatchBlocks` representing the Data section.

Listing 2.1: Fields of a Header

```
1  int version;
2
3  String creator = "MEGAN";
4
5  long creationDate;
6
7  long modificationDate;
8
9  long fileSize;
10
11 int numberOfReads;
12
13 int numberOfMatches;
14
15 long beginData;
16
17 long endData;
18
19 int numberOfClassifications;
20
21 byte [] typeClassificationKey;
22
23 int [] sizeClassification;
24
25 long [] beginClassification;
26
27 long [] endClassification;
28
29 long beginAuxiliary;
30
31 long endAuxiliary;
32
33 ReadFormat readFormat;
34
35 MatchFormat matchFormat;
```

## 2.2 Data

The Data section contains all metagenomic information parsed from the BLAST file. Hence do all other sections carry derived information, such as `numberOfReads` in `Header` containing the number of all reads or structural information, such as byte offsets. The Data section of a typical RMA file, which encloses a set of `ReadBlocks`, contains around 100,000-500,000 reads. A `ReadBlock` itself encloses again a set of matches.

## 2.2.1 ReadBlock

Listing 2.2 depicts the fields<sup>3</sup> of one `ReadBlock`. A `ReadBlock` is the container for exactly one read and its information derived from the matches. The `uid` determines the byte offset, and since MEGAN works on single files, the `uid` is also unique. `readHeader` and `readSequence` contain the original data fields which were, as above, compared to a database<sup>4</sup>. The `readHeader` identifies a read non-uniquely but contains extra information on the reads. The `readSequence` is the DNA sequence gathered by a metagenomic sample. Its length is stored in the field `readLength`. The classifications are stored in the fields `taxonid`<sup>5</sup>, `cog`<sup>6</sup>, `goProcess`<sup>7</sup>, `goComponent`, `goFunction` and `seed`<sup>8</sup>; these are optional fields. Not all RMA files contain values for all classifications.

Listing 2.2: Fields of a `ReadBlock`

```
1 long uid;
2
3 String readHeader;
4
5 String readSequence;
6
7 long mateUIId;
8
9 int taxonId;
10
11 int readLength;
12
13 int cog;
14
15 int goProcess;
16
17 int goComponent;
18
19 int goFunction;
20
21 int seed;
22
23 int numberOfMatches;
24
25 MatchBlock [] matchBlock;
26
27 int [] nextInClass;
```

The `mateUIId` holds a `uid` of another `ReadBlock`, describing that both reads belong together. The array `nextInClass` deals with byte offsets for every classification. Hence

---

<sup>3</sup>The actual arrangement of the fields in `ReadBlock` and `MatchBlock` is an array of objects (`Object []`).

For simplicity are the fields listed as single variables. Also are all methods deleted which don't cope with pure data like methods to connect or read from a RMA file.

<sup>4</sup>e.g. NCBI reference database.

<sup>5</sup><http://www.ncbi.nlm.nih.gov/Taxonomy/taxonomyhome.html>

<sup>6</sup><http://www.ncbi.nlm.nih.gov/COG>

<sup>7</sup><http://www.geneontology.org>

<sup>8</sup>[http://www.theseed.org/wiki/index.php/Main\\_Page](http://www.theseed.org/wiki/index.php/Main_Page)

`nextInClass` points for every classification at the next `ReadBlock`, which belongs to the same class. A detailed description is provided in chapter 3. The last field of `ReadBlock` is an array of `MatchBlocks`. Its length is stored in `numberOfMatches`.

## 2.2.2 MatchBlock

The `readSequence` is compared to a reference database and results into a list of BLAST hits, as described in the previous chapter. The BLAST hits are parsed and their information is stored in `MatchBlocks`. Hence a `MatchBlock` always belongs to the `ReadBlock` containing the `readSequence`, which generated the BLAST hit now assigned to the `MatchBlock`. The typical number of `MatchBlocks` per read varies from 10 to 100, where the maximal number is defined by the user. Note that a `MatchBlock` represents static information. Besides, the `ignore` field values are usually not altered.

Listing 2.3: Fields of a `MatchBlock`

```
1 long uid;
2
3 float bitScore;
4
5 float eValue;
6
7 String refseqId;
8
9 boolean ignore;
10
11 int cog;
12
13 int taxonid;
14
15 String text;
```

Listing 2.3 depicts the fields of a `MatchBlock`. The `uid` determines the byte offset of the file position. Therefore it can be perceived as the unique identifier of a `MatchBlock`. `bitScore` and `eValue` determine the quality of the Match. Higher values correspond to the quality of the match. The classifications `cog` and `taxonid` are extracted from the BLAST hit and needed to calculate the classifications of the enclosing `ReadBlock`. The `ignore` field determines, whether a `MatchBlock` is to be ignored in calculations and not to be displayed when MEGAN produces an output, or not. In addition, the complete BLAST hit is stored in the field `text`. This fields can later be exported in order to create new RMA files or can be scanned by a text search.

## 2.3 ClassificationBlock

A `ClassificationBlock` is an index structure. For all classes of a classification present in a file, a pointer to the first `ReadBlock` assigned to this class is stored.

As described, MEGAN supports classification sets (`taxonid`, `cog`, `goProcess`, `goFunction`, `goComponent` and `seed`), each of them defines its own `ClassificationBlock`. The maximum number of `ClassificationBlocks` is six.

A classification divides reads according to given criteria into different classes. For example, a criterion can be the assumed functionality of the DNA sequence. A set of `ReadBlocks` is then assigned to one functionality and another set to another functionality. The result is that every `ReadBlock` is assigned for each classification to one class<sup>9</sup>. Since this is in MEGAN the main criterion, discriminating reads, the `ClassificationBlock` deals with making this information accessible in a fast manner.

For one classification and every class present in the file a pointer to the first `ReadBlock`, which belongs to the class, is stored in a `ClassificationBlock`. The `ReadBlock` itself contains the information where to find the next `ReadBlock`, which belongs to the same class stored in the `nextInClass` array. Hence the entries in `ClassificationBlock` represent the pointer to the first read of a chain of pointers spread over the whole file.

Listing 2.4: Fields of a `ClassificationBlock`

```
1 String name;
2
3 byte keyType;
4
5 Map<Object, Long> key2pos; //maps a class to the byte offset of the
   first read which belongs to this class;
6
7 Map<Object, Long> key2sum; //maps a class to the number of all reads
   which belong to this class;
```

A `ClassificationBlock` consists of a unique classification name and the `keyType` determining the Java datatype of this classification. In addition, two maps are provided. The map `key2pos` stores the byte offset of the first `ReadBlock` (pos) for every in the file present class (key) in the classification. The other map `key2sum` stores for each class (key) the sum of the `ReadBlocks` assigned to this class.

## 2.4 Auxiliary

The `Auxiliary` stores MEGAN's meta information. For example, `Auxiliary` provides the information on what is displayed when MEGAN loads a RMA file. Since MEGAN works on trees, `Auxiliary` defines which nodes are collapsed and which are uncollapsed.

---

<sup>9</sup>If no fitting class can be found the class *Not Assigned* is allowed.



# 3 MEGAN's mode of operation

What kind of data - and when - does MEGAN request? And how is it determined that a RMA file can cope with the requirements in a sufficient speed? A subset of interesting functions of MEGAN is introduced in the following paragraphs with the goal to answer these questions and to introduce MEGAN's requirements.

## 3.1 Opening a file

Once MEGAN is launched, it is possible to open RMA files and display their content in MEGAN's main window. The content shown is a summary of one classification and is now exemplarily described in the `taxonid` classification. This classification arranges classes in a tree with around 500,000 nodes and a maximal depth of 47<sup>1</sup>. A class is assigned to exactly one node in the tree. Hence the classes assigned to the reads can be perceived as nodes within the tree.

As a prerequisite (which is not discussed in this work), the complete tree representation is loaded into MEGAN's cache, where only the top three levels are displayed - as shown in Fig. 3.1.

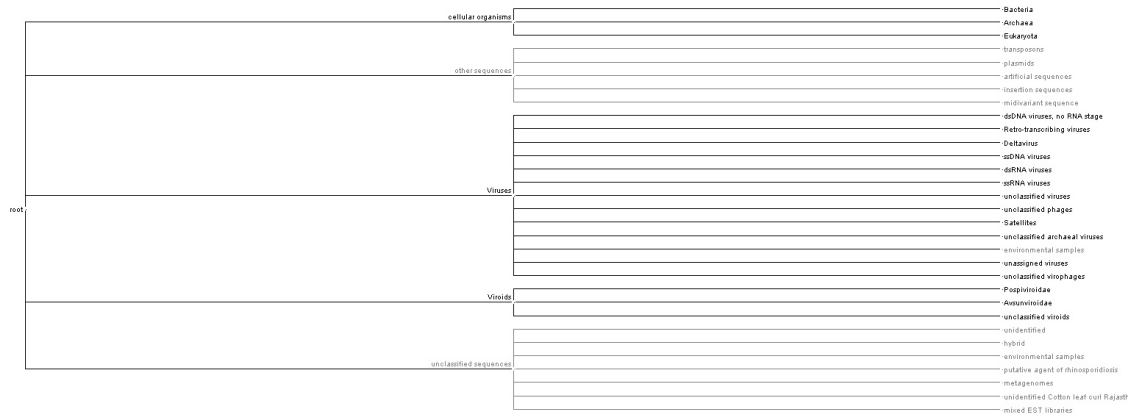


Figure 3.1: The initial screen of MEGAN after loading the `taxonid` tree

When a file is opened, MEGAN asks the Header where to find the `ClassificationBlock`, which belongs to the `taxonid` classification and loads the complete block into memory. This is the only MEGAN - RMA communication during the opening of a file when omitting the negligible `Auxiliary` interaction. Then the

<sup>1</sup>The files containing the tree structure can be found at <ftp.ncbi.nih.gov/pub/taxonomy/taxdump.tar.gz>. MEGAN uses a refined structure loaded into cache during the startup of MEGAN.

classes found in the `ClassificationBlock` are displayed within MEGAN's main window. Nodes not present in the `ClassificationBlock` are skipped if they are not ancestors of present nodes and therefore needed to construct the tree. All the information needed to display the classification tree, which are the classes and additionally the sums of assigned reads, is stored within the `ClassificationBlock`. Since a normal number of classes is around 1,000-20,000<sup>2</sup>, opening a RMA file is a swift operation.

The result of opening a file in MEGAN is depicted in Fig. 3.2<sup>3</sup>.

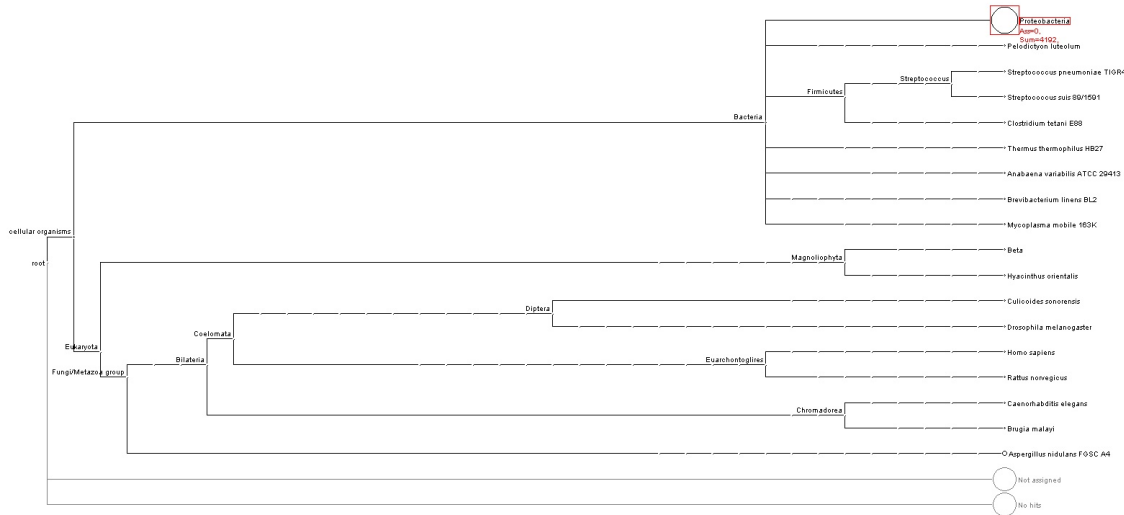


Figure 3.2: Open dataset *ecoli* displaying the summary of the `taxonid` classification

## 3.2 Browsing - The Inspector

Once a file is opened, it is possible to make a deeper analysis on the data. One approach is to use the Inspector. The Inspector is a simple tree based Java frame, which allows browsing through `ReadBlocks` and `MatchBlocks`.

Fig. 3.3 shows the Inspector on different levels. The tree with the lower depth shows two `ReadBlocks`, classified as *Pelodictyon luteolum*. The `ReadBlocks` are identified by the `readHeader` and also show the `numberOfMatches`. In this case, both `ReadBlocks` contain nine matches. On the data access level this is achieved in three queries. First, there is a lookup in the already cached `ClassificationBlock` for the key *Pelodictyon luteolum* respectively to its numeric representant. The key points to the first `ReadBlock` which belongs to this class and MEGAN loads the `ReadBlock` into its memory. `readHeader` and `numberOfMatches` are printed, and the pointer to the next `ReadBlock` in this class is looked up from the `nextInClass` array. After the two `ReadBlocks` have been evaluated, the Inspector displays the result.

The second window in Fig. 3.3 shows an example of browsing on `MatchBlocks`. After a click on a `readHeader` of a specific read, a list of assigned `Matchblocks` is shown in

<sup>2</sup>The size varies depending on how the parameters for the calculation of the classes are chosen.

<sup>3</sup>Consider that the chosen file, a test file with 2000 reads, *ecoli.rma* is uncommonly small. It is chosen to be able to display the complete classification.

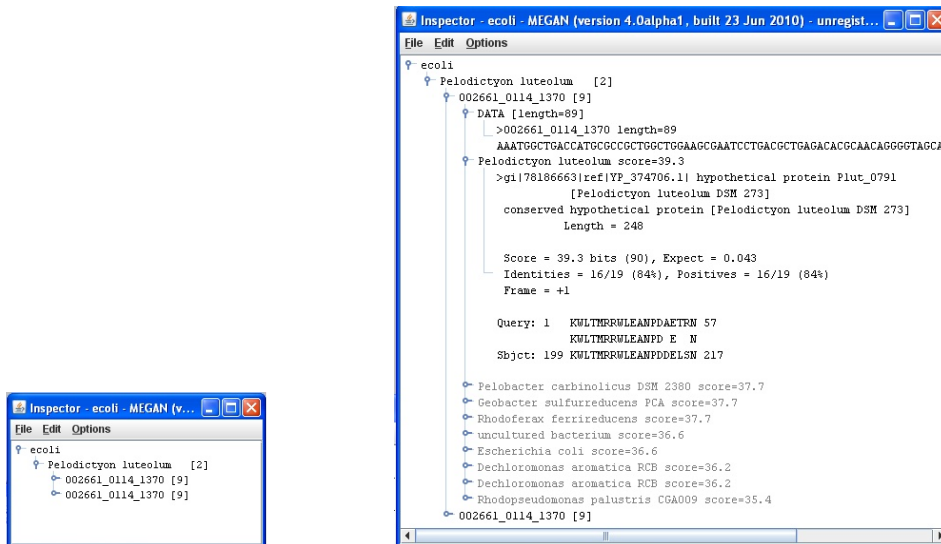


Figure 3.3: Inspector on read (left) and match (right) level

combination with an entry called *DATA*. *DATA* provides more information on the read, just like the `readSequence`.

Since the matches are also classified, their content is displayed as a concatenation of name (`taxonid` mapped to a name) and `bitScore`. The deepest level of information the Inspector provides is reached when one of the matches is chosen. Then the `text` field of the `MatchBlock` is shown. From the data access level, this is a simple task, because the matches are already loaded within the present `ReadBlock`, and if not, they can be requested later on.

### 3.3 Creating new datasets

Usually a new Dataset is created when a new metagenomic sample is produced, and as a result of a comparison step of these sequences, a BLAST file is generated. MEGAN parses BLAST files sequentially and produces `ReadBlocks` directly. The RMA file is created and then the `Read-` and `MatchFormats` are defined. Afterwards, `ReadBlocks` are sequentially written into the file. As the last step, structural data like `numberOfMatches`, `creationDate` are generated.

### 3.4 (Re)Calculating the classifications

Either during the creation of a dataset or as an independent operation, classifications can be (re)calculated. Hence in a file with six different classifications, six different calculations need to be applied. Since the calculation of the `taxonid` classification consumes the most time and is from the computational view the most interesting, its calculation process is described exemplarily.

Every `ReadBlock` has its own set of matches which are classified by a `taxonid`. The classification information in matches is provided by the BLAST file - therefore is static.

The class of a `ReadBlock` is then derived from the classes of its own matches. As mentioned before, `taxonid` describes nodes in a rooted tree. MEGAN applies the lowest common ancestor algorithm (LCA) on the classes of the matches<sup>4</sup> and stores the result in the enclosing `ReadBlock`. How this works in detail will be discussed later on, but it is important to see that all `ReadBlocks` and `Matchblocks` which are not filtered need to be loaded into MEGAN's memory. This is a time consuming and memory intensive operation. Chapter 7 will elaborate on this topic and show how a database can accelerate the calculation by outsourcing it from MEGAN's workload.

---

<sup>4</sup>There is a list of parameters determining if a match is included in the calculation.



## 4 Translation

A good database design is important for reaching the desired speed and saving space by avoiding redundancy. The first step in database design is to figure out the requirements of the application programme. In addition, what is in this case very interesting, because MEGAN already works on RMA files, its data structure and its semantic should be already known. Knowing the semantics is crucial for a good design in order to reach a clear separation between objects which do and which do not belong together. The requirement analysis was conducted in the previous chapters introducing MEGAN's standard data container, the RMA file.

Chapter 3 introduced a list of different scenarios. They are distinguished by the resource and computational demands.

- Browsing, just as loading a dataset or using the Inspector, needs data which is small but widespread over the whole dataset. Also, when comparing the fields of a `ReadBlock` and the information displayed in the Inspector (Fig. 3.3), it is clear that only a subset of the fields of the `ReadBlock` is actually needed. Since these tasks are realtime jobs, it is important that they retrieve a result the fastest. In order to reduce the overhead, only data which is needed will be loaded, and in order to minimize search efforts are the tables with a small set of columns desirable. To open a dataset and therefore returning the classification data, like in `ClassificationBlock`, the database needs to provide a summary of all `ReadBlocks`.
- BLAST files can easily exceed 5GB. Parsing this file and loading its information into a dataset consumes a lot of time. The runtime can be reduced by using tables with a larger set of columns, so that the indexing efforts on the data will stay at minimum<sup>1</sup>. But the creation is not a task which needs to be executed in realtime. Therefore it is desirable to optimize the database design for fast realtime job execution.
- Calculation of classifications is an expensive challenge. This is mainly due to the fact that all matches need to be read sequentially and later the classifications of all reads are altered. It would be then desirable that a set of tables would then maintain only classification information. With this alignment is all information not needed for the calculation skipped, which reduces the overhead.

As seen before, some points have to be balanced, but the realtime usability has priority over other aspects. In later chapters it will be shown that all three scenarios can be implemented and optimized in a database without blocking each other.

---

<sup>1</sup>Later in this thesis it will be shown that indexing is the bottleneck when it comes to the creation of new datasets in the database.

The second step is the conceptual database design. The information gathered in the requirement analysis is then used at a higher level design process, which includes user's and programmer's needs and options. For a conceptual design, a data model is needed. A data model, in our case the entity relationship data model, describes real world objects, such as reads or matches, as entities and their connection as relationships.

## 4.1 Entity relationship model

The entity relationship data model describes informal needs of the querying application in a formal way, that can be easily translated into a database schema.

The entity relationship model provides features some of them are introduced in the following list.

**Entity** An entity is a real world object, such as a dataset, read or match. Similar entities, such as a list of reads, are called entity sets.

**Relationship** A relationship connects entities. For example, a dataset encloses a set of reads, and a read belongs to a dataset. In this approach only relationships between two entities are needed.

**Attribute** Attributes describe entities and relationships on a certain level of detail. A dataset is for example defined by its name, the date of creation and the date of modification. The dataset can be specified in more detail, also by providing for example information about the size of the dataset. Hence an attribute extends the level of detail, in which the entity or relationship is described. It is possible to distinguish different kinds of attributes. The `readSequence` is an attribute in the normal sense, whereas the `readLength` stores the length of a `readSequence`. Therefore is an attribute like `readLength` defined as a derived attribute.

**Constraint** Constraints define requirements on entities. All entities in an entity set fulfill these requirements. For this approach, two constraints are important. The key constraint defines one or more attributes as an unique identifier of an entity. Consider the `uid` field shown in Listing 2.2. Every read can be distinguished by this field, hence a read with no `uid` field can not be part of this entity set. The second constraint is the participation constraint, defining how entities take part in relationships. For example, a dataset encloses a set of reads, whereas a read belongs to exactly one file. A read which does not belong to a file does not fulfill the participation constraint. Therefore it does not belong to the entity set. The one-to-many constraint is the only one needed in this approach<sup>2</sup>.

**Weak Entity** An entity in the normal sense, whose existence is dependent on another entity, is defined as a weak entity.

Figure 4.1 depicts the graphical representation of the entity relationship model structures needed in the next sections in order to design the database schema.

---

<sup>2</sup>For further information on the Entity Relationship Model read [RR03] chapter 2.

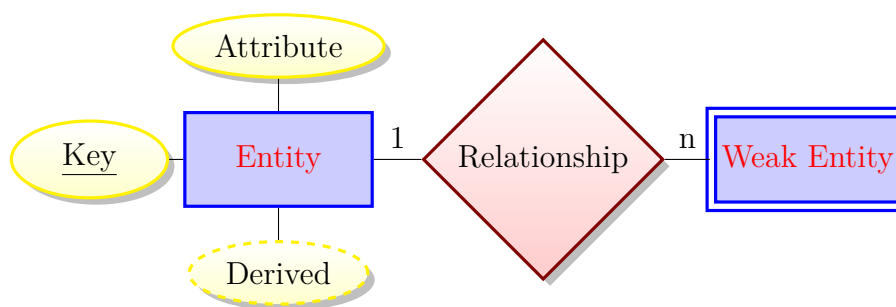


Figure 4.1: Structures of the Entity Relationship model

## 4.2 From a RMA file format to a database design

With the knowledge of basic structures of the ER model, the next step is to translate MEGAN's data into this model. Listings 2.1, 2.2, 2.3 and 2.4 depict the fields of a typical RMA file and are now used as a basic frame and a semantic guideline to determine connections within the data. Although it is a comfortable situation for the database designer to have a template of an already working data object, it should not be forgotten to analyze the requirements from scratch in order to not to be misled by existing (but for the database not optimal) arrangements. For example, the source data object might store information because it was not possible to compute them on demand. This might differ from the database's possibilities. The generation of the entity relationship model is a top-down approach with a range of refinements at later stages. Hence the design starts with the header level and is then subsequently extended up to the match level.

**Header** A dataset is an abstract container for reads. Global information on them is stored in the header. A set of reads is always assigned to a dataset. But every read can only be assigned to one dataset. If there is more than one dataset in the database, it is then distinguished either by the **File Name** or by the **File Key**. In addition, the header entity provides two relationships. One of them is maintained together with the auxiliary entity and the other one with the read entity. The relation 'Encloses' contains two attributes describing dates of creation and last modification of the dataset.

**Auxiliary** The auxiliary entity is a weak entity. Hence its existence is dependent on the existence of a dataset entity. The only attribute copes with the need to store a byte array split into single fields. Hence the participation constraint is one-to-many.

**Read** A set of reads is assigned to exactly one dataset, so that the relationship to a dataset is an one-to-many participation. A read is uniquely distinguishable from other reads by the **Read Key**. The fields **Header**, **Sequence** and **Length** represent the other attributes. Two relationships are maintained by this entity. First, a connection to classread is provided, an entity designed to cope with classification data of reads. The second relation connects the read with the match entity.

**Match** A set of matches is assigned to one read. In fact are the matches the result of the comparison step of the **Sequence** and a reference database. As a consequence, it is possible to define match as a weak entity, since its existence is dependent on

the enclosing read. However, it was decided to keep match as a non-weak entity. Therefore matches can also be extracted without the knowledge of the enclosing read. This can be beneficial when search algorithms are being executed on matches. Entries in match are distinguished by the **Match Key** field as a synonym to the **uid** field in a RMA file. The other attributes describe a match at a certain level of detail.

**Classread & Classmatch** Both entities maintain the classifications for either read or match. They contain two attributes, **Taxonomy** and **Value**. **Taxonomy** differentiates between classifications like **taxonid** or **cog** and **Value** stores classes.

With the design of these entities, a question arises how to differ between entities and attributes. Instead of creating two entities it is also possible to add a list of attributes to the owning entities read and match. In the case of match, this could be more efficient, since the number of classifications is limited and their values are static. On the other hand, it would be a good idea to provide the entity classread. At the moment there are six classifications provided, some of which are optional. Consider datasets with a small number of classifications, they would still have six attributes in the entity read (for every classification one attribute) but only entries of present classifications in the entity classread. With these entities, the schema is more flexible and hopefully speeds up the calculation of classifications. Also, redundancy is to be kept at a minimum since entities with **null** values in not present classifications do not appear. Classifications in classread or classmatch only appear if they exist in the surrounding dataset. In addition, classread remains at a desired size so that read has a smaller set of attributes. This can have a major impact on search algorithms. Without going into details of the process, it can be stated that the reason for the acceleration is a change in the physical storage, so that a larger set of entities<sup>3</sup> fits into one disc page. Consequently searching on 'thin' entities faster than searching on 'wide' entities, due to the smaller number of pages which need to be loaded from disc to memory.

---

<sup>3</sup>respectively rows

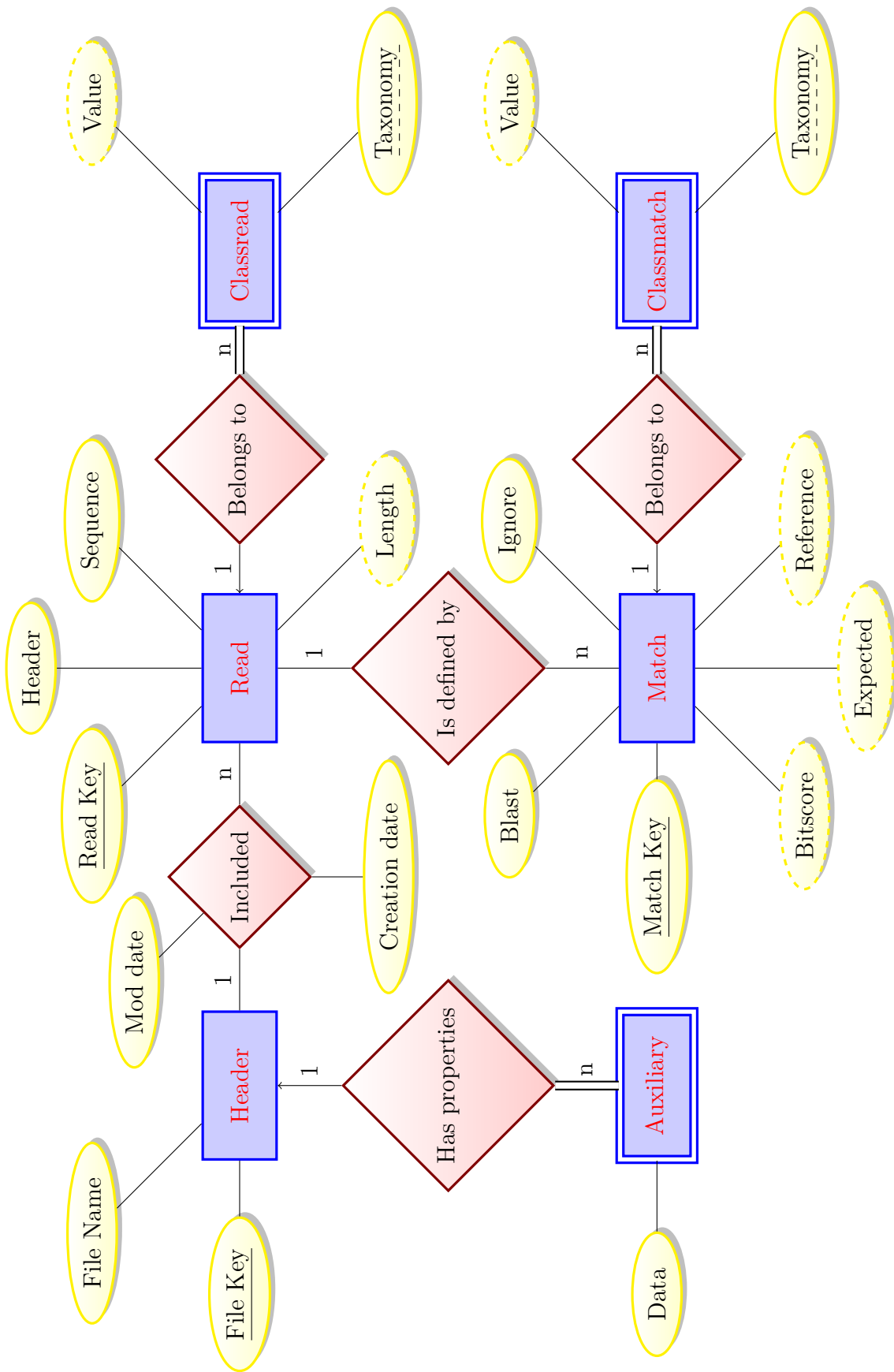


Figure 4.2: Entity Relationship model for MEGAN's data

## 4.3 Translation of the Entity Relationship diagram

The database designed in this work maintains data in a structure of loosely coupled tables. The reasons to work on tables, as well as what advantages they provide, will be introduced in the next chapter. For this section it is sufficient to know that tables are identified by their name, describe columns and contain rows. The columns force rows present in a table to fulfill criteria, such as certain data types. For example, a table *NAME* can consist of columns *First Name* and *Surname* can cope with entries like (name, name), and not with rows, such as (name, age).

The translation itself is strictly bound to rules, which can be found in [RR03] chapter 3.5. and are now applied to the entity relationship diagram in Fig. 4.2. However, this is not the final database design. If bottlenecks appear, the design of the tables can be changed at any later point of MEGAN's development.

### General translation rules

An entity results into a table. The columns of this table are defined by the attributes of the entity. Attributes of relationships are transferred to one of the connected entities. The participation constraint of the entities decides where the attributes are located. In the case of one-to-many, the relationship attributes are transferred to the entity participating once, in order to avoid redundancy. If the attributes are transferred to the entity participating n-times, duplicate information would be stored. Every table contains a column or a set of columns, which are defined to follow the key constraint. Potential rows are then distinguished by the key columns.

Entities and attributes now exist unrelated. Hence the relationships are conducted by a concept called foreign key. Since in the model in Fig. 4.2 there are only one-to-many participations, a foreign key is conducted by a new column in the table on the n-times side of the relationship. This column carries the key column of the one-time side of the relationship. For example, a set of matches always belongs to a read. Table match therefore gets a new column *Read Key*, identifying for every match the read to which it has been assigned.

Initially, weak entities have no key attributes, and they inherit their key column from the entity they belong to. In order to achieve this, the same system as for the foreign keys is applied. When the weak entity is connected via an one-to-many relation, the key stems from the added foreign key column plus a set of columns which originally belonged to this table.

The tables derived from the entity relationship model shown in Fig. 4.2 are depicted in Fig. 4.1, 4.2, 4.4, 4.5, 4.6, 4.7 and 4.3. Although all necessary data can be stored in a structure of tables strictly derived from the introduced model and all requirements of MEGAN are fulfilled, some refinements during implementation change the structure at some positions took place and resulted into adding new tables. This is conducted due to the size of the input data and the concurrent need of getting summaries in realtime.

A detailed description of the tables is provided in the next section.

### 4.3.1 Header and auxiliary

The entity header is translated into the table header with seven columns. As defined, the two attributes of the entity, respectively the two attributes of the relationship 'Encloses', are added. Additionally, three new columns are provided. One of MEGAN's functions is to return the number of classifications, reads and matches in a dataset. Since most datasets consist of 100,000-500,000 reads and around hundred times more matches, it is not possible to provide a calculation in realtime. Same applies to the third field. To answer the query about the number of classifications correctly, a sequential scan over all reads must be executed. This is not doable in an adequate span of time.

The auxiliary is translated following the rules exactly. As a weak entity it inherits the **File Key** column and keeps the attribute **Data**. Both columns form an unique key.

### 4.3.2 Read and classread

The columns **Read Key**, **Header**, **Sequence** and **Length** create the basic framework for the table read derived from the read entity. Since read maintains a relationship with the entity header, a foreign key column **File Key** is added. Another two columns **No. Matches** and **Maximal Score** are added in order to support certain queries. Their utility will be introduced in chapter 6. **No. Matches** stores the number of matches assigned to this read and **Maximal Score** column stores the highest **BitScore** of these matches.

Classread as a weak entity inherits the **Read Key** column from read and translates its attributes into columns.

### 4.3.3 Classificationblock

When MEGAN loads a dataset, the first query gets a summary of a classification. The content of the summary is divided into two parts. The first part is to find all distinct classes assigned to the classification and present them in the dataset. The second part is to calculate how many reads belong to each class found in the previous part. In order to calculate this information, a sequential scan over a subset of the rows in table classread is needed. Due to the size of datasets can this query be answered in realtime only if preprocessed. The table classificationblock consists of four columns. One column identifies the dataset (**File Key**), another one the classification (**Taxonomy**) and the third one the class (**Value**). The result of the query which gathers the sums of all classes is stored in column **Sum**. Columns **File Key**, **Taxonomy** and **Value** form the unique identifier for this table.

As mentioned before, one of the goals of database design is to avoid redundancy. The table classificationblock is redundant, since all its data is derived. But the gain in speed is enormous and required to provide the fast execution of an *open dataset* command. Since the fields are derived, maintenance and updating the table are another challenges, solution strategies to which are introduced in the following chapter.

### 4.3.4 Match and blasthit

The entities match and classmatch are merged and define the tables match and blasthit. Since both the number of classifications and their entries are static, it seems to be

beneficial to merge `classmatch` with the owning entity. The table `match` now consists of the columns derived from the attributes in Fig. 4.2 plus two extra columns `taxonid` and `COG`, forming the classification part. Additionally, `match` and `read` are connected by a foreign key `Read Key`, inserted to match the last column.

90% of the size of a dataset results from the `text` field in `match`. That field contains a long plain text extracted from the initial BLAST file. Another table is created for coping with the demand to store large objects like plain text. This is done by the table `blasthit`. At a later stage of this thesis it will be shown, why it is a good decision to outsource this field.

### HEADER

<u>File Key</u>	File Name	Cre Date	Mod Date	No. Classes	No. Matches	No. Reads
-----------------	-----------	----------	----------	-------------	-------------	-----------

Table 4.1: Table header

### READ

<u>File Key</u>	<u>Read Key</u>	Header	Sequence	Length	No. Matches	Maximal Score
-----------------	-----------------	--------	----------	--------	-------------	---------------

Table 4.2: Table read

### AUXILIARY

<u>File Key</u>	<u>Data</u>
-----------------	-------------

Table 4.3: Table auxiliary

### MATCH

<u>Read Key</u>	<u>Match Key</u>	Ignore	Reference	EValue	Bitscore	Taxonid	COG
-----------------	------------------	--------	-----------	--------	----------	---------	-----

Table 4.4: Table match

### CLASSREAD

<u>Read Key</u>	<u>Taxonomy</u>	Value
-----------------	-----------------	-------

Table 4.5: Table classread

### BLASTHIT

<u>Match Key</u>	text
------------------	------

Table 4.6: Table blasthit

### CLASSIFICATIONBLOCK

<u>File Key</u>	<u>Taxonomy</u>	Value	Sum
-----------------	-----------------	-------	-----

Table 4.7: Table classificationblock



# 5 Database candidates

In this chapter, database candidates able to cope with MEGAN's requirements are introduced. The database itself belongs to a database management system (DBMS). The workload of a DBMS is to create, maintain and control databases. Simplified, a DBMS can be compared to a file system and the files to databases, whereas databases come with many advantages highlighted later on.

Two different database management systems are presented: PostgreSQL and H2. Both are able to cope with MEGAN's requirements and build the database derived from the entity relationship model (development described in chapter 4).

The two DBMSs follow different approaches. On one hand there is PostgreSQL<sup>1</sup>, a large server-based system which provides many built-in functions and routines. It gives the user many options to optimize the database. It can be fitted exactly to any given requirements. On the other hand H2<sup>2</sup> is a pure Java DBMS with a small footprint of 1MB. Its big advantages are the platform independence and that no installation is needed. The executable jar file can come within MEGAN's library.

Every DBMS is based on a data model describing the manner in which the data is stored and handled. Since this approach uses the relational data model based on relational algebra, the idea of relational database management systems will be introduced in this chapter. Relational DBMS work on data which is organized in tables and queried by operators, which are provided by the relational algebra. The requirements of a DBMS and possible bottlenecks influencing the selection of DBMS are discussed afterwards. Finally, both candidates are introduced in detail and their strengths and weaknesses are elaborated upon.

## 5.1 Relational database management system

A database management system is a piece of software designed to maintain databases. It is comparable to a file system, but in many ways superior in its functionality and flexibility. In addition to creating, modifying, deleting and random access a DBMS provides advanced search algorithms and structures for database access. Other advantages are the guaranteed consistency on the data, multiuser control and crash recovery - to provide a short list of some of the possible features. However, these advantages come at a price. DBMSs are more complex and need a certain knowledge in optimization issues to gain the desired acceleration compared to file systems.

Every DBMS is based on a specific data model describing on a high level how the database is constructed. This work focuses on the relational data model. This model was introduced in 1970 by E.D. Codd [Cod70]. Codd describes the model as superior to

---

<sup>1</sup><http://www.postgresql.org/>

<sup>2</sup><http://www.h2database.com/html/main.html>

READ			CLASSIFICATION		
Key	Header	Sequence	Key	Taxonomy	Value
1	78_0077	GGGTTGAG	1	taxonId	543
2	85_0253	GGGGCAAA	1	COGs	-1
3	86_0789	GGGGATAG	1	SEED	1367
			2	taxonId	131567
			2	SEED	4553

Table 5.1: Tables read and classification

**READ**(Key:integer, Header:string, Sequence:string)  
**CLASSIFICATION**(Key:integer, Taxonomy:string, Value:integer)

Figure 5.1: Schema for the tables read and classification

other models. This assumption is established by comparing the relational model and its strength to a RMA file, introduced in chapter 2. Firstly, a basic idea of the relational model will be introduced, and then compared to the file model, e.g. RMA files.

### 5.1.1 Relational model

The relational model uses relations, or in other words tables, as the central construct. Greatly simplified, a set of tables define the database. A table consists of two components; the first component is the schema. The schema specifies the name of the relation, the name of each field and the type of each field. The second component is the instance of the table. The instance is a set of tuples conforming the constraints defined by the schema ([RR03], chapter 3). Hence the ordering of the rows in a table is arbitrary.

In this sense is the word relation ambiguous. It also describes how tables are connected. Tables are related to each other in a loose manner, this leads to a high level of data independence which comes with certain advantages and is one of the main benefits of storing data in a relational database. Section 5.1.3 elaborates on this fact by comparing the RMA file structure to the possibilities of the relational model.

Fig. 5.1 depicts two tables. The schema of table read in Fig. 5.1 defines a table enclosing three columns specified by name and type. Only tuples fulfilling this constraints are part of the instance, hence form the rows in the tables. In this case there are three tuples assigned to the table read. Considering both tables read and classification, the concept of loose coupling becomes clear. Both tables are identified by the same Key which allows the tables to be combined. The tuple with key value 1 has three different classifications assigned, whereas the tuple with key value 3 has none. Hence the relation between both tables is defined by the Key column.

### 5.1.2 Relational algebra

Relational algebra describes the language of a relational database. It allows a set of tables to be connected by predefined conditions and returns a new table with only

tuples fulfilling the conditions. Possible conditions are both projection to reduce the number of columns, but also selection deciding which rows are present in the result. Additionally, functions like sorting and aggregation are supported. In the daily usage of a relational database, the language SQL is used and is translated into the operators of relational algebra. A good introduction into the language can be found at the w3schools website<sup>3</sup>.

### 5.1.3 Relational model versus RMA file

The main goal of every data management is to provide the invoking application with the queried data correctly and as fast as possible, but also not to waste space - MEGAN's data can namely easily exceed the size of several GBs. As shown in the previous chapters, there are different strategies for storing data. Each strategy has different strengths and weaknesses. For example, maintaining data in a file is less costly than in a database, but a database enables more flexibility and if properly designed also more speed for certain queries.

When comparing a RMA file to a relational database, two points have priority: redundancy and speed. By designing a database on the relational model, the data redundancy is minimized in a normalization step<sup>4</sup>. Fig. 5.1 lists two tables related to each other. Two combinations of both tables (Tab. 5.2 and 5.3) contain same information but are harder to maintain due to the redundancy of the data. For example, an update on the `Header` column in 5.2 needs to change three rows instead of one. Additionally, redundancy wastes space. Consider that the `sequence`, a field which easily exceeds several hundred characters, is stored multiple times, as seen in Tab. 5.2. For datasets with a large number of reads would this have a major impact on the size of the resulting database. Hence the concept of loose coupling allows to minimize redundancy. Only information which has a semantic connection is stored in one table. In the table `read` (Tab. 5.1) a header always belongs to a sequence, but a header can belong to zero or more classifications. A classification does not belong to a header.

As seen before, the redundancy of the storage of data can be minimized. But indexes whose main task is to find desired data very fast (although tables contain a large number of rows) are from informational standpoint redundant. They contain no information which has not been stored in a table before. But since they are critical for search and ordering algorithms, this redundancy is a good tradeoff when the speed gained by using the right index is considered. The RMA file also provides an index. As seen, the `ClassificationBlock` (Listing 2.4) more or less<sup>5</sup> contains no data relevant to the application, but points at certain offsets inside the file in order to accelerate queries which need realtime execution.

According to [Cod70] is the main reason for the superiority of the relational model the data independence, divided into ordering dependence, indexing dependence and access path dependence. To make data independent from the application and therefore facilitate finding the systems optimal storage, all representational information must be

---

<sup>3</sup>[http://www.w3schools.com/SQL/sql\\_intro.asp](http://www.w3schools.com/SQL/sql_intro.asp)

<sup>4</sup>For further reading consider [RR03] chapter 19

<sup>5</sup>The `ClassificationBlock` of a RMA file combines both: The first of a chain of pointers representing the index but also the length of the pointer list which is displayed when MEGAN loads a file.

<u>Key</u>	<u>Header</u>	<u>Sequence</u>	<u>Taxonomy</u>	<u>Value</u>
1	78_0077	GGGTTGAG	taxonId	543
1	78_0077	GGGTTGAG	COGs	-1
1	78_0077	GGGTTGAG	SEED	1367
2	85_0253	GGGGCAAA	taxonId	131567
2	85_0253	GGGGCAAA	SEED	4553
3	86_0789	GGGGATAG	null	null

Table 5.2: Combination I: Tables read and classification

<u>Key</u>	<u>Header</u>	<u>Sequence</u>	<u>Taxonid</u>	<u>COG</u>	<u>Seed</u>
1	78_0077	GGGTTGAG	543	-1	1367
2	85_0253	GGGGCAAA	131567	null	4553
3	86_0789	GGGGATAG	null	null	null

Table 5.3: Combination II: Tables ead and classification

deleted.

**ordering dependence** In many cases the application requests data arranged in a predefined order. A data management is ordering dependent if it follows this order and stores the data in the same order. This is an easy way of avoiding the sorting step, but certain problems can also arise. What if the data has to be sorted differently to the initial ordering, or what to do if an insert or delete has to be executed? In these cases major rewriting of data is needed, which then leads to poor performance. The relational model allows storage of data in tables without forcing a certain order upon the data. If sorting is required, it is explicitly executed on the result of a query. Hence a relational database is independent allowing fast inserts, deletes and modifications without of the restrictions imposed by any orders. However, the RMA file has a certain order defined by offsets. This makes inserts and deletes expensive. Hence MEGAN allows no inserts or deletes in a naive way. A database easily copes with this challenge since the ordering is independent.

**indexing dependence** MEGAN works with data of the size of several gigabytes. Hence structures supporting the search for reads with special attributes are needed - the solution is to use indexes. The index structure in an RMA file is stored widely spread over the whole file and is conducted by pointer to offsets. However, the starting point for every query searching for reads determined by classification and class (comparable with `Taxonomy`, `Value` (Tab. 5.1)) is the `ClassificationBlock`. For every pair of classification and class, an offset is stored referring to the first read which belongs to this pair. Every read itself refers to the next read which also belongs to that pair. Hence a chain of pointers spread over the whole file is generated. This is a sufficient approach for static files. But in the case of deletes the pointer lists of all subsequent reads need to be changed. A database on the other hand maintains the index structure differently. This structures work in a way similar to pointers but are organized outside the data. For example, deletes do

not destroy the index structure. The creation and deletion of indexes will therefore not affect the storage, only seek times change.

**access path dependence** A DBMS allows many users to work on one database at the same time. If the database is designed properly, an update on the data, e.g. (re)calculation of the classifications or deletes, will not affect the other users' work. This is due to the fact that a database contains no representational information and is completely independent from the application. In addition, a DBMS provides functionality which guarantees the consistency of the data, such as locking or multiversion protocols. However, if the RMA file is modified, the whole chain of pointers must be recalculated. The offset, or in other words the access path, of other user working simultaneously on the same file is then invalid. The application is dependent on the access path.

With these dependencies an RMA file limits the options of MEGAN. An RMA file is just multiuser compatible when it is 'read-only' and, because of the ordering dependence, is also static. A database allows to delete unused reads and matches without heavy recalculation and therefore saves space and still maintains performance.

With a global database many users could consistently work on the same data without the need to download GB sized files. If MEGAN is 'read only' only little fragments of dataset are needed.

## 5.2 Requirements

The previous section summarized optional preconditions facilitating the storage of data and abstracting the communication between two layers, application and data management. As shown, a relational database can cope with these preconditions. However, there are non-optional requirements on a database. Most of the requirements are covered by standard features of DBMSs, and listed below. The candidate DBMS needs to

- provide a bridge to communicate with Java, e.g. JDBC<sup>6</sup>.
- fulfill the ACID<sup>7</sup> criteria.
- handle indexes.
- provide triggers([RR03] chapter 5).
- cope with large datasets.
- support multiuser functionality and therefore support locking protocols.
- compress large objects, such as the BLAST result, in order to save space.

---

<sup>6</sup><http://java.sun.com/javase/6/docs/technotes/guides/jdbc>

<sup>7</sup>Atomicity, Consistency, Isolation, Durability[RR03]

Furthermore, there are properties of DBMSs which are important for MEGAN but not standard for DBMSs. In the best case a DBMS has a small footprint and no installation is needed. A pure Java DBMS is the best solution due to its uncomplicated use as a local database. However, a rich set of built-in functions is welcome. Several calculations which are computed in MEGAN now can then be outsourced to the database's scope. This reduces the overhead which usually occurs in data transfers between MEGAN and the database. A calculation in the memory assigned to the DBMS also allows very selective conduction of queries and leaves the optimizer the largest possible freedom in finding the best plans. This is due to the fact that a large dataset is calculated differently to a small dataset. Hence a different execution of the operations might accelerate the whole calculation. Furthermore, it is known that only one database, MEGAN's database, will run on the DBMS. Hence MEGAN's performance can be optimized by adjusting certain parameters, such as the assigned memory size.

## 5.3 Candidates

Following the comparisons described above, it was decided to use PostgreSQL and H2. Neither can fulfill all conditions explained in the previous chapter. Although PostgreSQL is a DBMS which is not implemented in Java, it provides a rich set of built in functions. PostgreSQL is a mature DBMS giving the user the biggest possible freedom in optimizing the system to MEGAN's requirements. But PostgreSQL must be installed and cannot work in an embedded mode. However, H2 is a stable non-installation pure Java DBMS which can easily run in embedded mode. Unfortunately, H2 does not allow much optimization and the set of functions is rather small compared to PostgreSQL. Its big plus is the compatibility<sup>8</sup> mode emulating PostgreSQL's behavior<sup>9</sup>. For the implementation of MEGAN is this a big advantage, since the source code requires only minimal changes in order to work on either PostgreSQL or H2. To recapitulate, strengths and weaknesses of both candidates can be utilized by MEGAN. H2 is a good solution for the user who wants a minimal solution without an installation. For advanced users or users who need MEGAN on everyday basis is PostgreSQL a better option. After adjusting the parameters and adapting them to the users system, PostgreSQL can easily beat H2 in performance.

### 5.3.1 PostgreSQL

PostgreSQL is an old and therefore mature database management system. From 1995 on an SQL interpreter has been included in PostgreSQL. Due to the development of the World Wide Web and the raising need of large and fast databases, PostgreSQL became quite popular. The fact that it is an open source project and is still being developed and extended, makes PostgreSQL attractive also to sophisticated users who need special func-



<sup>8</sup><http://www.h2database.com/html/features.html#compatibility>

<sup>9</sup>This for exception handling in Java important.

tionality<sup>10</sup>. The library of builds to be included after the installation is large and is still being extending by a growing community. The main website containing information about builds and PostgreSQL related projects is the pgfoundry<sup>11</sup>.

PostgreSQL runs on all important operating systems and provides all functionality needed for MEGAN. However, since most requirements of MEGAN are rather standard, it is more interesting to show how to use and optimize PostgreSQL.

Two ways of installation are possible, either installing a binary or compiling the source code. Both packages and tutorials can be found online<sup>12</sup>.

The next step is to optimize PostgreSQL to MEGAN's requirements and to the system on which PostgreSQL will run. The optimization itself can be achieved by manipulating the parameters in *postgresql.conf* text file<sup>13</sup>.

**shared\_buffers** define the memory allocation of cached data in PostgreSQL. In the memory the data is cached. Pages of the most used indexes or results of queries executed shortly before remain in the cache still other pages replace them. The initial value, 32MB, is too small for MEGAN. It is recommended to set the value to around one quarter of the size of the main memory.

**effective\_cache\_size** determines how much OS cache (including external disk space) is expected to be available. Since MEGAN DB runs always in embedded mode, the value should be rather low; approximately 50 per cent of the main memory.

**default\_statistics\_target** When queries are executed, the database needs to find an optimal plan, respectively a plan with the most promising runtime. For example, it is sometimes more efficient to make a sequential scan then using an index. The query optimizer involves factors such as ordering of rows and sizes of tuples to find the best plan for the execution. For this reason, the optimizer needs to know what the structure and entries of the tables are. These are gathered by execution of the **analyze** command. Since it is too costly to analyze all tuples, a subset of these is chosen. The higher the value in **default\_statistics\_target** is, the more precise queries can be planned. It is recommended to set the value to around 1,000.

**work\_mem** If for example a query requires a sorting step, either explicit by using **ORDER BY** or implicit if the optimizer uses a join which needs a sorted result, reduces the correct dimension of this parameter the runtime of the query remarkably. Sorting is a task which can only be solved by knowing all entries which are to be sorted. In the best case the complete sorting step is executed within the memory without the need of external disk space. The memory a user has allocated for the execution of a query is defined by the **work\_mem** parameter. This memory should be set to a value around 20MB, although 20MB is more then required for MEGAN. But

---

<sup>10</sup>For a list of advantages consider <http://www.postgresql.org/about/advantages>

<sup>11</sup><http://pgfoundry.org/>

<sup>12</sup><http://www.postgresql.org/download/>

<sup>13</sup>All optimizations are derived from a tutorial which can be found on [http://wiki.postgresql.org/wiki/Tuning\\_Your\\_PostgreSQL\\_Server](http://wiki.postgresql.org/wiki/Tuning_Your_PostgreSQL_Server), respectively from a large set of evaluated test-queries on the database.

since one user only works on the database, there is no disadvantage to setting this parameter slightly too high.

**maintenance\_work\_mem** PostgreSQL provides diverse maintenance functions. **VACUUM** deletes unreferenced rows and **ANALYZE** gathers information about the tables. This operations require memory. A value of around 256MB is recommended to accelerate these operations.

**sequential\_page\_cost & random\_page\_cost** Every time a page is loaded to the memory since it contains tuples needed for the execution of a query, the optimizer needs to know the costs of loading the page from the disk to memory, or respectively the time to find the page on the disk. Two different types of disk access are distinguished. When pages are read during a sequential scan, a load of pages is subsequently read from the disk. The second type appears during an index scan. The index picks one page from a (random) position on the disk. Of course the sequential scan performs better when looking at the costs for single pages, since no disk seek time is needed, but the standard parameter settings in PostgreSQL describe the gap between random and sequential page cost as too large. Hence the optimizer might decide too often for an sequential scan, although an index scan might perform better. The value of random page cost should be lowered to 3.0, whereas the value of sequential page cost remains at 2.0.

### 5.3.2 H2

H2 is a database management system written entirely in Java. It was developed by Thomas Müller who also worked on HSQLDB. This DBMS is interesting for this approach for several reasons. It is an open source project which is still in the development but the performance tests are promising<sup>14</sup>. It fulfills all needs of MEGAN, except for the compression of large objects<sup>15</sup>.



A new approach in connecting Java to databases without the need of SQL, JaQu<sup>16</sup> has been introduced. This might be interesting for the future development of MEGAN. However, H2 is a young project and provides fewer function libraries than PostgreSQL. Optimization is also not an easy task. One of the only parameters similar to PostgreSQL is the `cache_size`, which is defined when a connection to the database is established.

---

<sup>14</sup><http://www.h2database.com/html/performance.html?highlight=performance&search=perform#firstFound>

<sup>15</sup>RMA files are usually compressed. So TOAST of PostgreSQL also compresses the text fields like in blasthit. H2 also provides a compression mode but the size of the results are not very promising

<sup>16</sup><http://www.h2database.com/html/jaqu.html>



# 6 Implementation

In this section, the details of the database implementation<sup>1</sup> are described. For the implementation, three programming languages are used: Java, SQL and plpgsql. Extension of MEGAN's functionality, the main part of the implementation, is achieved by using Java as the programming language. SQL queries, utilized to work on data in the database, are implemented in Java and executed against a database. Since SQL is a simple programming language, it lacks the structures of higher programming languages. Therefore, the language plpgsql is utilized for programming issues inside the database.

The functionality is split into three tasks, which are implemented separately. First, MEGAN requires capability for database communication. So far, the standard data container for MEGAN has been the RMA file. Therefore was the program optimized for communication with this file format. The current structure is not suitable, hence an abstraction layer for general data access is inserted. This layer separates the application from the raw data access. The second step is to do preliminary work on the DBMS, such as physically creating the database and implementing a class handling connections to the database. Internal database functions are also included in order to either outsource workload from MEGAN or to provide consistency on derived tables like classificationblock. The abstraction layer, mainly defined by the interface `IConnector`, is implemented as the last step, which provides MEGAN full database support.

## 6.1 Making MEGAN independent from RMA

Prior to enabling MEGAN to receive its data from a database, some modifications on the original programme architecture need to be performed. In detail it means to delete all fields carrying RMA-specific information, such as positions of the file in the program code which belongs to the application scope and not to the data access scope. Therefore, clear lines between these two scopes must be drawn. This is achieved by inserting interfaces to encapsulate data access.

Most methods in MEGAN dealing with receiving the data from RMA files use the block structure, such as `ReadBlock`, `MatchBlock` or `ClassificationBlock`, introduced in chapter 2. The use of blocks is so deeply integrated into MEGAN's source code so that it is not alterable. Therefore it is expected that future versions of MEGAN will be using the `ReadBlock` structure as well. For RMA files it is convenient to provide complete blocks since they physically exist in the file, but for a database is this task more complicated. The structure of loosely coupled tables spreads the data from a block into different tables. For example, for building a `ReadBlock`, data from the tables `read`, `classread`, `match` and `blasthit` are needed. Hence, the block structure needs to be remodeled in order to fit the databases demands, too.

---

<sup>1</sup>Consider that Exception handling is for simplicity skipped in all Java Listings in this thesis.

The second part of the adaption process is to bundle all communication routes between the application and the data access object into one interface, which implemented meets all requirements of MEGAN.

### 6.1.1 Adaption of present implementation

The present code uses in nearly all function calls blocks as data object, e.g. `ReadBlock`, `MatchBlock` or `ClassificationBlock`. Since it is performance-wise not feasible to retrieve complete blocks from a database - one of the strengths of a database is to be able to be highly selective when it comes to choosing the desired data - the block structure needs to be remodeled. The blocks are replaced by interfaces providing similar functionality, since all RMA specific functions are deleted. But they pass the implementation process on other instances, either to the RMA or to the database. Hence a database can cope with the challenge to build a block with its own possibilities.

#### **IReadBlock**

Listing 6.1<sup>2</sup> shows methods of the interface `IReadBlock`. Compared to Listing 2.2, many methods have been changed. The `nextInClass` array is deleted since it deals with RMA-specific information. Also, the Listing 2.2 does not provide the complete information about the RMA `ReadBlock`. As a simplification, the fields in Listing 2.2 are listed as single variables, although the actual arrangement stores the fields in an array (`object []`). The information where to find the single fields in the array is provided by the `ReadFormat`, stored in the `Header` of the RMA file. This array is deleted and replaced by a list of setters and getters now handling the data management. Other methods of the RMA `ReadBlock` are deleted, which delegates the communication with the RMA file. The result is a minimalistic interface that provides a list of setters and getters to represent the data. It leaves all possible freedom to the implementing instance, either RMA or DB.

#### **IMatchBlock**

The `IMatchBlock` (Listing 6.2) follows the same principle as the `IReadBlock`. Only the fields designed for pure data access are kept, the methods dealing with RMA-specific behavior are deleted. The interface provides a set of getters and setters and delegates the rest to the implementing instance.

#### **IClassificationBlock**

In order to implement the `IClassificationBlock` (Listing 6.3), the original `ClassificationBlock` (Listing 2.4) is changed completely. All RMA-specific information, such as the `key2pos` map pointing to file offsets, is deleted and replaced by a list of getters and setters to determine the name of the classification, together with a list of present classes in the dataset.

---

<sup>2</sup>In order to keep the clarity, some getters and setters are skipped. List of skipped setters/getters: `GoProcess`, `GoComponent`, `GoFunction`, `SEED`, `numberOfAvailableMatches`.

Listing 6.1: IReadBlock

```
1 public interface IReadBlock {
2
3     public long getUId();
4
5     public void setUId(long uid);
6
7     public String getReadName();
8
9     public String getReadHeader();
10
11    public void setReadHeader(String readHeader);
12
13    public String getReadSequence();
14
15    public void setReadSequence(String readSequence);
16
17    public long getMateUId();
18
19    public void setMateUId(long mateReadUId);
20
21    public void setTaxonId(int taxId);
22
23    public int getTaxonId();
24
25    public void setReadLength(int readLength);
26
27    public int getReadLength();
28
29    public void setCOG(int cog);
30
31    public int getCOG();
32
33    public int getNumberOfMatches();
34
35    public void setNumberOfMatches(int numberOfMatches);
36
37    public IMatchBlock[] getMatchBlocks();
38
39    public void setMatchBlocks(IMatchBlock[] matchBlocks);
40
41 }
```

In addition, the `IClassificationBlock` provides the central name mapping for the classifications. Consequently, static strings determining the names of all possible classifications are used to query the classification data. This is important since the classifications are stored in the database, so that they don't have own columns and are identified by strings stored in the `Taxonomy` column in the table `classread` (Tab. 4.5). Hence it is important to know the correct name of the classification when querying the `Taxonomy` column.

Listing 6.2: IMatchBlock

```
1 public interface IMatchBlock {
2
3     public long getUId ();
4
5     public void setUId (long uid);
6
7     public int getTaxonId();
8
9     public void setTaxonId(int taxonId) ;
10
11     public float getBitScore() ;
12
13     public void setBitScore(float bitScore);
14
15     public int getCOG();
16
17     public void setCOG(int cogId);
18
19     public String getRefSeqId();
20
21     public void setRefSeqId(String refSeqId);
22
23     public void setExpected(float expected);
24
25     public float getExpected();
26
27     public void setLength(int length);
28
29     public int getLength();
30
31     public boolean isIgnore();
32
33     public void setIgnore(boolean ignore);
34
35     public String getText();
36
37     public void setText(String text);
38
39 }
```

### Auxiliary

The Auxiliary remains unchanged and is a byte array (`byte[]`). No additional RMA specific information is stored within it.

### IReadBlockIterator

In addition to the query for single reads, MEGAN also provides a method to return a list of reads with certain common parameters. This allows queries like 'get all reads which belong to a certain classification and have the predefined class'. Therefore, the

Listing 6.3: IClassificationBlock

```

1 public interface IClassificationBlock{
2
3     public static String taxonid = "taxonId";
4
5     public static String COG = "COGs";
6
7     public static String gopro = "GoProcess";
8
9     public static String gofun = "GoFunction";
10
11    public static String gocmp = "GOCompoment";
12
13    public static String seed = "SEED";
14
15
16
17    public int getSum(Object key);
18
19    public void setSum(Object key, int num);
20
21    public void incrementSum(Object key);
22
23    public String getName();
24
25    public void setName(String name);
26
27    public String toString();
28
29    public int computeTotalSum();
30
31    Set<Object> getKeySet(); //All classes of a classification present in
32    the dataset
33 }

```

`IReadBlockIterator`<sup>3</sup> can be perceived as an access method to process results - for the case when a list of reads is returned from a query. In addition to the standard iterator methods, such as `hasNext()`, `next()` and `remove()` methods are added and they either display the progress of the operation or allow closing the connection to the data storage.

### 6.1.2 IConnector

In order to apply the abstraction layer to its full, not only containers for several kinds of data need to be provided. An interface bundling the entire communication between the application and the data object needs to be created. For this task the `IConnector` is developed. The `IConnector` is a single interface, which not only serves as a gateway from MEGAN to miscellaneous data access objects, but also provides a clear border between

<sup>3</sup>Actually, the `IReadBlockIterator` extends two other interfaces. As a simplification, all method declarations are now added to the `IReadBlockIterator`.

Listing 6.4: IReadBlockIterator

```

1 public interface IReadBlockIterator{
2
3     public String getStats();
4
5     public boolean hasNext();
6
7     public IReadBlock next();
8
9     public void remove();
10
11    public void close();
12
13    public int getMaximumProgress();
14
15    public int getProgress();
16
17 }

```

the both layers. Advantages of a single interface handling all communication are obvious. For the instance implementing the interface, a set of methods has to be implemented, whereas the rest of the programming code of the application is not considered. From the applications point of view it is obvious which methods the data object provides and, if needed, where to request new methods.

As described before, the `IConnector` manages the entire communication between MEGAN and the data object. Since the number of functions is too large, example scenarios of usage of the `IConnector` are discussed. The first scenario is to find compatible reads. In many cases, only a set of reads fulfilling given criteria should be returned. The most important criterion is to filter reads by classification and class. The second scenario concerns the general content, namely how to obtain summaries, such as the information provided by the `IClassificationBlock`. The third scenario is how to alter data, such as sending an updated `IReadBlock` back to the data object where the changes should be applied persistently. The last scenario, which will be described further, is creating a new dataset.

### Finding compatible reads

Finding a set of reads fulfilling given criteria is one of the main tasks of the implementation. On one hand, a large set of data needs to be looked up, on the other hand, MEGAN expects in many cases a realtime execution. Hence it is important to focus on this task and find an efficient solution.

Listing 6.5 shows five different ways to receive reads, either as an iterator or as a single block. First, the parameters are introduced, and as a second step, the methods are presented.

**dataSelection** A `IReadBlock` contains a list of data fields. Most of them are only needed for certain methods in MEGAN. In order to reduce overhead in the transfer and gain speed by using the selectivity of a database, MEGAN makes an assump-

tion on which fields will be needed. For this reason, the datatype `DataSelection` contains a boolean for every possible field in an `IReadBlock` or `IMatchBlock`. However, MEGAN needs to be able to use other fields than the ones listed in the `DataSelection`.

**findSelection & regex** MEGAN provides a text search on fields, e.g. the `text` in `IMatchBlock` or the `header` in `IReadBlock`. For each field that can be scanned, a boolean is stored in `FindSelection`, informing the executing instance whether a field should be included in the search or not. The pattern to be searched for is provided as a regular expression in the `regex` field.

**minScore & topPercent** The variable `minScore` in combination with `topPercent` defines which matches will be included in the result, even though it does not influence choice of reads. The bitscore determines the quality of a match. Matches where bitscore is lower than the `minScore` are ignored and will not appear in the results. If there are any matches left, besides the `minScore` criterion, the `topPercent` border is defined and applied. The `topPercent` and the highest bitScore of the matches of each read define the next border. In order to clearly set the border, `minScore` is a global variable applied to all matches, whereas the `topPercent` border is applied read for read. For example, if the value of `topPercent` = 10 and the best bitscore is 100, only matches with bitscore higher then 90 will appear in the results.

**classification & classid** One option is to find all reads that can be identified by the tuple `classification` and `classid`<sup>4</sup>.

**readUid** If the `uid` of an `IReadBlock` is already known, this block can be loaded by providing the parameter `readUid`.

To address a query that results into the desired reads, a combination of these parameters is used.

The first method `getAllReadsIterator` shown in Listing 6.5 provides no parameters influencing the selection of reads. Consequently, all reads are returned. A common size of the result is 100,000 - 500,000 reads. The matches are filtered by the `minScore` and `topPercent` tuple.

The second method, `getReadsIterator`, extends the functionality of the first method by adding the `classification`<sup>5</sup> tuple. The size of the result shrinks from 100,000-500,000 reads to 10-10,000 reads<sup>6</sup>. Since this is one of the crucial and most important operations, it will be described in detail shortly.

The query for one read where `readUid` is known can be executed by the `getReadBlock` method. To filter the matches, the `minScore` and `topPercent` borders are included.

The `getFindAllReadsIterator` and `getFindReadsIterator` methods provide the same functionality as the first two methods, but acquire a search field parameter. Since the size of one dataset can easily exceed one gigabyte, searching cannot be executed in realtime. Especially a search on the `text` field in match is a time consuming task.

---

<sup>4</sup>In the notation of the database, these paramaters are synonyms to `Taxonomy` and `Value`.

<sup>5</sup>This is a typical query of the kind the Inspector utilizes.

<sup>6</sup>The values can vary from dataset to dataset and depend on the classification.

Listing 6.5: Finding compatible reads

```

1 public IReadBlockIterator getAllReadsIterator(float minScore, float
   topPercent, DataSelection dataSelection);
2
3 public IReadBlockIterator getReadsIterator(String classification, int
   classId, float minScore, float topPercent, DataSelection
   dataSelection);
4
5 public IReadBlock getReadBlock(Long readUid, float minScore, float
   topPercent, DataSelection dataSelection);
6
7 public IReadBlockIterator getFindAllReadsIterator(String regex,
   FindSelection findSelection, DataSelection dataSelection);
8
9 public IReadBlockIterator getFindReadsIterator(String regex, String
   classification, int classId, FindSelection findSelection,
   DataSelection dataSelection);

```

### Get summaries

In many cases is MEGAN's workload reduced in order to display summaries of large datasets. The most important summary and also a computational bottleneck in MEGAN is the `IClassificationBlock`. The content of the `IClassificationBlock` lists all classes of a chosen classification found in the dataset and their frequency. In order to return correct values, a sequential scan over all reads has to be performed. Since MEGAN expects the result in realtime, the calculation on demand is not possible for large datasets. Hence the RMA file gathers aggregated data in the `ClassificationBlock`. The same can be applied to the database which maintains its own table in order to provide this information.

The main summary methods are shown in Listing 6.6.

The `IClassificationBlock` is received by determining the classification name and using the `getClassificationBlock` method. Since the set of classifications can vary from dataset to dataset, it is important to know which classifications are provided. The function `getAllClassificationsName` returns a list of all present classifications.

In addition to the information on the classifications, a little subset of general queries returning summaries is shown. Functions to determine the number of all matches and reads are specified. They are intended to provide the user with some meta-information on the currently open dataset.

### Alter data

If some changes in reads or matches are made, a data object has to be able to update the data and apply persistent changes. The `IConnector` lists three methods<sup>7</sup> in order to alter existing data shown in Listing 6.7.

In order to store single `IReadBlocks` or `IMatchBlocks`<sup>8</sup>, the `IConnector` offers the methods `putReadBlock` and `putMatchBlock`. Both methods identify the block, which

<sup>7</sup>Methods specialised to alter fields like `setNumberOfMatches(int)` are skipped

<sup>8</sup>To store a list of `IReadBlocks` the `createNewDataSet` method is used.

Listing 6.6: Summaries

```

1 String[] getAllClassificationNames();
2
3 int getClassificationSize(String classificationName);
4
5 int getClassSize(String classificationName, int classId);
6
7 IClassificationBlock getClassificationBlock(String classificationName);
8
9 int getNumberOfReads();
10
11 int getNumberOfMatches();

```

Listing 6.7: Methods to alter existing data

```

1 void putReadBlock(Long readUid, IReadBlock readBlock, DataSelection
   dataSelection);
2
3 void putMatchBlock(Long matchUid, IMatchBlock matchBlock, DataSelection
   dataSelection);
4
5 void updateClassifications(final String[] classificationNames, final
   List<UpdateItem> updateItems, ProgressListener progressListener);

```

should be updated, by its `uid` and replace the existing block with the provided one. The `dataSelection` determines the fields which should be updated.

The recalculation of the classifications is a special case of updating data. Only the classifications for the reads are updated. For this reason, the `updateClassifications` method is used. Providing a special function to this task is necessary when considering that only classifications yet classifications for all reads are changed. Same effect can be achieved by using the `putReadBlock`  $n$  times for  $n$  reads. This extra method, however, leaves the implementing instance with all possible freedom in coping with the task.

The structure of `updateClassifications` is shown in Listing 6.7<sup>9</sup>. A list of names of the classifications to be updated is provided in `classificationName`, whereas the data to be updated is presented in the list `updateItems`. One `updateItem` is uniquely assigned to a read determined by its `readUid` field. The classes of this read, which are to be updated, are therefore stored at the same `updateItem`. Therefore, the `updateClassifications` method alters classifications and classes of a whole dataset by a single method invocation.

### Create new datasets

Prior to the actual work, a dataset has to be created. This is performed by the `createNewDataSet` method, which is depicted in Listing 6.8. A new dataset can either be created from source files like a parsed BLAST file, or it displays reads extracted from an existing dataset. For the `IConnector`, the data source is not important.

<sup>9</sup>The `progressListener` field is ignored in the explanation since it has not effect on the execution.

Listing 6.8: Method to create a new dataset

```
1 void createNewDataset(DataSelection dataSelection, IReadBlockIterator  
    readBlockIterator, ProgressListener progressListener);
```

A new dataset is created by providing an `IReadBlockIterator` over the reads that should be part of the new dataset and the `DataSelection` which decides, which fields will be added to the new dataset.

## 6.2 Preliminary database jobs

Besides the adaption of MEGAN and implementing the abstraction layer, another important issue is to configure the database. So far the DBMS has been installed and optimized, but the database is not has been set up.

Once the database is generated, the next step is to create the schema. This implies creating tables derived from the structure described in Fig. 4.2<sup>10</sup>. Additionally, certain functions need to be inserted in order to keep the table classificationblock updated, since its entries are directly calculated from the values in classread.

In the last step a connection to the database has to be established. Both DBMS provide a bridge to Java<sup>11</sup>.

### 6.2.1 Create MEGAN DB

After installation and optimization of the DBMS a database within the DBMS has to be created. Although the general commands for creation are similar for both DBMSs, some important differences occur. These will be discussed in the following paragraphs.

#### Creating a database in Postgresql

The general command for creation is `CREATE DATABASE Megan`; it is, however, prerequisite to be logged on as superuser. In addition, PostgreSQL provides a script which allows the creation from the operation system console. The creation is then executed by the command

```
$ createdb Megan user
```

The field `Megan` defines the name of the database, the field `user` defines the user name. During the creation, the user becomes the owner of the database. It is also possible to skip the `user`, then PostgreSQL defines the user who has created the database as the owner.

---

<sup>10</sup>What has been done in the previous chapter.

<sup>11</sup>JDBC -<http://java.sun.com/javase/technologies/database/index.jsp>

## Creating a database with the H2 database

The H2 Database follows a different principle in database creation. The database is implicitly created when it is opened the first time. The user who invokes the 'open database' command automatically receives administrator rights for the database. Listing 6.9 shows how a connection to the database is opened.

Listing 6.9: Opening a connection to H2 via JDBC

```
1 Connection con = DriverManager.getConnection("jdbc:h2:~/name", "user",  
    "password");
```

### 6.2.2 Injecting MEGAN's tables

Once the database has been established, the next step is to create the tables. The table structure is described in Fig. 4.1, 4.2, 4.5, 4.3, 4.4 and 4.6 and defines a set of `CREATE TABLE` commands.

Appendix .1 shows the SQL creation commands for the tables. Exemplarily, the SQL commands are described for the tables header and read.

The table header consists of seven columns (`file_key`, `file_name`, ...), where to each column a datatype is assigned. The datatypes are derived from the Java datatypes<sup>12</sup> of corresponding getters/setters in the blocks, e.g. `IReadBlock`. The Java type `long`, for example, is the basis for the SQL datatype named `BIGINT`. A string with a maximal length is defined as `VARCHAR(x)`, where `x` is a number defining the maximal number of allowed literals. SQL also provides a datatype for strings of 'infinite' length: the `text` type. In addition to the columns, some constraints are defined. The table header has a unique identifier, or in SQL grammar: a `PRIMARY KEY`, the column `file_key`. For the table read another previously introduced constraint is defined, the `FOREIGN KEY`. This constraint is identified by the keyword `REFERENCES`. The `FOREIGN KEY` concept describes the connection between tables. For the tables read and header this means, that only those values for the column `file_key` are allowed for the table read, which are also present in the `file_key` column in header. The `ON DELETE CASCADE` triggers a certain behavior of a `FOREIGN KEY`. If the referenced value, here `file_key` in the table header, is deleted, all reads containing the same `file_key` value are deleted as well.

### 6.2.3 Additional indexes

Indexes allow fast searches even if the database contains large sets of data. This is mainly due to the structure of the index, e.g. a B+ tree, which reserves extra disk pages exclusively for the indexed columns. In this pages only indexed fields are stored. Hence all unindexed fields are not present and the density of tuples per disk page grows. The acceleration is therefore based on the following fact. An index scan needs to load a smaller subset of disk pages where information - usually pointers to the pages containing the rows - is found, compared to the costs of a sequential scan requiring a larger set of

<sup>12</sup>For the complete mapping consider <http://java.sun.com/j2se/1.3/docs/guide/jdbc/spec/jdbc-spec.frame8.html>

pages. For tables with a large set of rows, the usage of an indexes leads to a remarkable acceleration. But with an index is extra complexity implied and other operations, such as inserts and updates, loose speed since indexes need to be kept updated. Consequently, the fields to be indexed should be chosen wisely.

In both PostgreSQL and H2 are indexes implicitly created for all columns, which are parts of a `PRIMARY KEY`. Since MEGAN's database consists of a set of seven tables, seven indexes are created directly together with the creation of the tables.

Because MEGAN's requirements are known, the index structure is now optimized. For some queries, where additional indexes are needed, these are described in the next paragraphs<sup>13</sup>.

#### **Additional index on table read: `file_key,read_key`**

A typical query searches for all reads assigned to a dataset. Therefore, the query joins the two tables, header and read. From header, one row only - or in other words, one dataset only - is selected. Hence the actual task is to find all reads which have the same value in the `file_key` column as the row from header. For this task, an index on the table read on the column `file_key` would be sufficient. But since another query is to find all reads belonging to a dataset and assigned to a classification and a class, the `read_key` column is also included in the index. Hence a query on the tables header, read and the table dealing with the classifications classread can be evaluated solely by using indexes. Since other queries use this index to bypass a scan of the table read as well, this index should always be defined.

#### **Additional index on table read: `head`**

MEGAN supports text searches. One of the columns which can be scanned is the `head` column of the table read. Since the datatype of this column limits the maximal length to 100 characters, an index is still able to speed up this query while requiring only a limited amount of space. Other fields which can be scanned during a text search contain on the other hand datatypes - these are not suitable to keep a potential index at a reasonable size.

#### **Additional index on table match: `read_key,match_key`**

This index follows the same concept as the first index introduced. The related task is to find all matches belonging to a read. This is an interesting task since the implementation of the database loads the matches only if explicitly requested outside the `DataSelection`.

#### **Additional index on table classread: `read_key,taxonomy, value`**

A classification is always defined by a tuple of two values: the name of the classification and the class. For this reason, in order to combine this tuple with a read, the `read_key` field is also included in the index. This index allows fast access to the desired rows of the table classread.

---

<sup>13</sup>If in the future development of MEGAN bottlenecks appear, new indexes can be created or deleted anytime without changes in MEGAN's source code.

## 6.2.4 Functions

Besides tables and indexes, a database supports another structure: the functions. They provide additional features and extend the range of work which can be done within the database scope. Therefore, they reduce overhead and gain speed by using the internal cache and memory. In particular PostgreSQL provides a large set of advanced functions; some of them will be introduced in chapter 7.

Functions can be written in several programming languages. The native language of PostgreSQL is plpgsql, which extends the functionality of SQL to a higher level and includes loops and conditions. However, there are scenarios when other programming languages like C++ or Java are needed. The functions needed in these scenarios can also be loaded into PostgreSQL. The advantage of doing so is that these functions work in the cache of the database. Heavy swapping into different memory sections is not required.

The focus in the following section is on self written functions ensuring consistency of the data. The table `classificationblock` contains data directly derived from `classread`. Therefore, a change in this table affects the table `classificationblock`<sup>14</sup>, too. Three different scenarios of possible changes in `classread` have to be distinguished:

- The combination of `Read_key` and `Taxonomy` doesn't exist in `classread`, therefore a new row into the table `classread` is inserted. This can happen if the calculation is performed the first time.
- A row is deleted since the classification is deleted.
- A present row in `classread` is updated. Only the entry of the `value` column changes.

In the table `classificationblock` this results into three different scenarios:

- If the tuple relevant to the summary of this dataset, `file_key`, `Taxonomy` and `Value` exists already, then `sum` column is incremented by one. If it doesn't exist, then a new row with the `sum` one is inserted.
- The value of the column `sum` in the relevant row is decremented. If, consequently, the entry in the `sum` column is zero, delete the row.
- The principles described in the first two items are now followed.

PostgreSQL implements this by using the function shown in the appendix .2. For H2, Java methods within MEGAN are implemented in order to assure the correct functionality. Since H2 runs in the same memory section as MEGAN, the operation remains performant.

---

<sup>14</sup>This is a task usually handled by a Trigger. But since the queries on the database are limited and the database is embedded, an 'almighty' function is used.

## 6.2.5 Getting connected

For accessing external storages Java provides miscellaneous techniques. Communication with databases is enabled by `java.sql`, a part of the Java distribution. The counterpart to this on the side of the database is the JDBC bridge. Both parts are connected by the `java.sql.DriverManager`, which chooses the appropriate JDBC driver implementing the methods of `java.sql`.

One of these methods is the interface `java.sql.Connection`, which is in charge of opening, maintaining and closing a connection. Through this connection is all data bidirectly transferred. Hence it is important to keep the connection open as long as there is a stream between Java and the database. But since MEGAN works on an embedded database, it is desirable to open a connection once and use it until MEGAN is closed. This is due to the fact that opening and closing a connection is much more time consuming than maintaining it. Also it is important to close unused connections, since some databases allow only a certain number of connections. Because there is exactly one connection, this task can be implemented easily.

The technique of establishing<sup>15</sup> and closing a connection is for PostgreSQL exemplarily shown in Listing 6.10. To receive a connection, first, the JDBC driver must be known and included. To receive the connection, the `java.sql.DriverManager` finds an appropriate driver and receives the connection parameters, such as database name, username and password, and establishes the connection to the database. For closing the connection, the interface `Connection` provides the `connection.close()` command. Listing 6.10 shows the technique of opening and closing a connection to a PostgreSQL database.

It has to be emphasized that the connection is stable. Even if the connection is timed-out (which can happen if MEGAN requests no data for some time), the connection will automatically be re-established.

## 6.3 Communication with the database

The Java package `sql` enables communication with a database. Two structures essential for the implementation, which are used for nearly every query, the `Statement` and the `ResultSet`, are introduced in this section.

A `Statement` is a structure able to receive SQL queries and in combination with an open `Connection` also execute them against a database. Its existence is associated with a `Connection` object shown in Listing 6.11. The execution of the query is invoked by `statement.executeQuery("SQL code")`<sup>16</sup> returning a `ResultSet`. The main function of a `ResultSet` is that it controls the cursor which points to the result, where it remains within the database's scope. The method `set.next()` moves the cursor to the next row in the result. To retrieve the data from this row, methods like `set.getX(position)` are used, where `X` determines the type of the result and `position` the position in the `SELECT` list. In Listing 6.11 `set.getLong(1)` determines that a `long` is expected at position 1 in the `SELECT` list, which is the column `file_key`.

---

<sup>15</sup>Since PostgreSQL is server-based, the server needs to be started before a connection can be opened.

H2 does not need any extra starting or stopping of the server due to the support of embedded mode

<sup>16</sup>This is a special case where a result is expected. There are other methods to handle updates, deletes or inserts.

Listing 6.10: Opening a Connection to a PostgreSQL server via JDBC bridge

```

1 Connection con = null;
2
3 String url = "jdbc:postgresql://localhost:5432/Megan";
4
5 String user = "user";
6
7 String password = "password";
8
9
10
11 Connection getConnection() {
12
13     Class.forName("org.postgresql.Driver");
14
15     con = DriverManager.getConnection(url, user, password);
16
17     return con;
18 }
19
20
21 void closeConnection() {
22
23     con.close();
24
25 }

```

This way of retrieving data seems complicated, but since a result can contain a large set of rows and the size is in some cases not known beforehand, this prevents Java from running out of memory and also speeds up the streaming of data. This is due to the fact that it allows fetching data, even though the result in the database is not completely generated yet. In many cases<sup>17</sup>, the first rows of a result are already present and applicable by a `ResultSet`, although the database is still searching for more results.

It often occurs, that a similar query is executed many times where just selection parameters change, e.g. 'get all reads for a class in a classification' is widely used. Only the classification name and the class id change. In order to accelerate similar queries, Java introduces the `PreparedStatement` (Listing 6.12). The query is defined with a question mark for all parameters which should be included later on. The idea is to maintain a precompiled `Statement` where only parameters need to be inserted, whereas the normal `Statement` needs to be compiled each time when it is executed. When using a query regularly, the use of a `PreparedStatement` is recommended<sup>18</sup>.

Additionally, it has to be emphasized that both structures need to be closed when they are not needed anymore, for both of them memory within the database is allocated. If too many `ResultSets` remain open, this can have a major impact on the performance.

<sup>17</sup>In order to explain how a query is evaluated and which operations might affect this behavior, a deeper analysis would be needed. This is not included in this work. But it needs to be mentioned, that most queries in this work support this behaviour and allow streaming

<sup>18</sup><http://java.sun.com/docs/books/tutorial/jdbc/basics/prepared.html>

Listing 6.11: Usage example of Statement and ResultSet

```

1 public void useStatement(Connection con){
2
3     Statement statement = null;
4
5     ResultSet set = null;
6
7     try {
8
9         statement = con.createStatement();
10
11        set = statement.executeQuery("SELECT file_key, creation_date FROM
12            header WHERE file_name = 'sample'");
13
14        while(set.next()){
15
16            set.getLong(1);
17
18        }
19    }finally{
20
21        set.close();
22
23        statement.close();
24
25    }
26
27 }

```

## 6.4 Database implementation of blocks

As mentioned before, one of the strengths of a database is that it is possible to extract single values from a table uniquely defined by column and row. In order to realize this functionality in a RMA file, MEGAN needs to load a complete block<sup>19</sup> into its own memory and then to pick the desired field. Often just a small subset of the fields is needed. The count of transferred blocks, however, might be large. In the worst case the overhead is be enormous and slows down the execution. With the integration of the database a way of specifying the desired fields is installed by the `DataSelection` class introduced in section 6.1.2.

The counterpart on the database side of the implementation is the `SelectionAnalyzer`, which is also in charge of evaluating the fields. This class translates the fields in `DataSelection` into column names<sup>20</sup>. However, if fields are requested later on without being listed in the initial `DataSelection` the database is able to load them on demand, this is possible as the database sided implementation of the `IReadBlock` (or `IMatchBlock`) provides methods to reload missing information.

Since in later sections knowledge about the detailed implementation of these classes

<sup>19</sup>It was allowed to determine if matches are needed or not.

<sup>20</sup>All table and column names are listed in the class `PostGRESTableNamesAndData`

Listing 6.12: Usage example of PreparedStatement and ResultSet

```

1 public void usePreparedStatement(Connection con){
2
3     PreparedStatement statement = null;
4
5     ResultSet set = null;
6
7     try {
8
9         statement = con.prepareStatement("SELECT file_key, creation_date
10            FROM header WHERE file_name = ?");
11
12         statement.setString(1, "sample");
13
14         set = statement.executeQuery();
15
16         while(set.next()){
17             set.getLong(1);
18         }
19     }finally{
20
21         set.close();
22
23         statement.close();
24
25     }
26 }
27
28
29

```

is essential the concept of 'loading on demand' is presented in the following.

### 6.4.1 ReadBlockDB

The ReadBlockDB implements the IReadBlock, which mainly consists of a list of getters and setters. In order to create a ReadBlockDB(Listing 6.13)<sup>21</sup>, three parameters are needed. Firstly, a database object - a set of Java classes providing database logic for connections and query execution - in charge of reloading data is required. Secondly, each block in MEGAN is identified by a uid. In this case it is the `readuid`. The third parameter, the `dataSelection`, is optional and is only included since matches are loaded on demand only.

Listing 6.13 shows one getter and one setter. Both of them are in charge of either getting or setting the sequence of a read. For getting the field, two ways are possible. Either the sequence field is present and stored in the variable `sequence`, or it needs to be loaded. Through the method `getReadItem("columnName", readuid)` of the database

<sup>21</sup>The Listing shows only a subset of all getters and setters. Since all of them work similarly, one setter and one getter only are introduced.

Listing 6.13: A subset of the ReadBlockDB class

```

1 public class ReadBlockDB implements IReadBlock{
2
3     IDB db = null;
4
5     DataSelection selection = null;
6
7     Long readuid = null;
8
9     String sequence = null;
10
11    public ReadBlockDB(IDB db, long readuid, DataSelection selection) {
12
13        this.db = db;
14
15        this.readuid = readuid;
16
17        this.selection = selection;
18
19    }
20
21    public String getReadSequence() {
22
23        if (seq != null) {
24
25            return sequence;
26
27        } else {
28
29            sequence = (String) db.getReadItem(PostGresTableNamesAndData.
30                READ_SEQ, readuid);
31
32            return sequence;
33
34        }
35    }
36
37    public void setReadSequence(String readSequence) {
38
39        this.sequence = readSequence;
40
41    }
42
43 }

```

object, the sequence is received from the database. The setter method changes the variable locally but has no access to the database.

Note that the list of matches assigned to a read is never loaded until explicitly requested by method invocation.

## 6.4.2 MatchBlockDB

A `MatchBlockDB` is the equivalent to the `ReadBlockDB`, with a difference that it contains a match. The functionality is similar to the `ReadBlockDB`.

## 6.5 DBConnector

The `DBConnector` enables the adaption of MEGAN to the databases. This is achieved by implementating the `IConnector` introduced earlier. Additionally, the functionality is extended to some operations which a database provides but which are not listed in the `IConnector`, such as receiving a list of datasets present in the database.

Listing 6.14: The constructor of `DBConnector`

```
1 IDB db;
2
3 DBConnector() {
4
5     if(Document.getDb() == Document.postgres){
6
7         db = new PostGres();
8
9     }
10 }
11 }
```

First, the structure of the `DBConnector` is analyzed and methods specific for the `DBConnector` are introduced. Then scenarios of MEGAN's workload which have been presented before are described in detail as operations executed on the database.

### 6.5.1 The class structure

The structure of `DBConnector` and the architecture below is simple. `DBConnector` delegates the method invocations to suitable classes which then implement the functionality. Since MEGAN should not be limited to the usage of just one database, the constructor of `DBConnector` (Listing 6.14) allows choosing between databases<sup>22</sup>.

`DBConnector` delegates the workload to five different classes. All of them receive besides the given parameters a database object providing raw data access. In this case, it is always the `Postgres` object which is responsible for creating and executing SQL queries.

**ReadBlockIteratorDB** Every time an `IReadBlock` or an iterator over `IReadBlocks` is requested, `DBConnector` creates a new instance of this class. The constructor distinguishes between different kinds of queries by using a list of parameters (section 6.1.2).

---

<sup>22</sup>In this work, PostgreSQL and H2 are supported only. H2 has no own connector and uses the same queries as PostgreSQL.

**ClassificationBlockOperationsDB** All tasks related to classifications are completed in this class. This includes updating the classifications as well as returning all kinds of parameters in connection with classifications, e.g. classification sizes or lists of available classifications.

**StandardDBOperations** All tasks which are not fitting into the first two classes are implemented here. This includes load and store operations on the auxiliary table and getting general parameters, such as number of reads or matches in a dataset.

**MatchBlockIteratorDB** There is currently only one method in `DBConnector` which requests a single match directly. Usually the `IMatchBlocks` are requested implicitly using an `IReadBlock`. In order to connect both functions, explicit querying for an `IMatchBlock` through the `DBConnector` and implicit querying using an `IReadBlock`, the `MatchBlockIteratorDB` is implemented.

**Create new Dataset** Creating a new dataset is directly forwarded to the implementing DB instance.

The database object (Here always `PostGRES`) forks into different classes executing database access. For updates or inserts the `PostGRESUpdate` class and for loading `PostGRESQuery` are used. Same applies to classes dealing with copying csv files or implementing special H2 behaviour.

## 6.5.2 Open a dataset

Focusing on the data access level, loading a dataset is split into three operations where information is transferred from and to MEGAN. First, the user receives a list of all possible datasets in the database. The name, respectively the identifier of the dataset, is sent back to the database in order to retrieve the data - suitable entries of the classificationblock table determined by `file_key` and `Taxonomy` - in order to display the initial screen.

The data size is limited to some kilobytes<sup>23</sup>. This is important, as two different strategies for handling the database's result are used in this programme. The first strategy is to fetch the complete result into a Java datatype and then return it from the data access object to MEGAN's scope. Hence the `ResultSet` pointing to the result of the query is completely iterated and closed before the method returns the result.

## 6.5.3 Find compatible reads

Reads are distinguished by a list of parameters section 6.1.2 from other reads and are, if queried, returned within an `IReadBlockIterator` as `IReadBlocks`, or as their database implementation `ReadBlockDB` respectively.

Fetching the data within an iterator follows another strategy than loading the complete result into MEGAN's memory, as introduced in section 6.5.2. Here the result is retrieved on demand. Hence the row where the cursor of the `ResultSet` is located is read, wrapped

---

<sup>23</sup>Consider 1,000 different classes identified by an integer plus an integer to store the `sum` column a size of 8KB is to be transferred from the database to MEGAN's memory.

into a Java datatype and returned. The `ResultSet` remains open in order to enable fetching more rows on demand until the user closes it or the cursor reaches the end of the result.

As described earlier in this chapter, the `DBConnector` bundles all kind of queries searching for reads into the `ReadBlockIteratorDB`. The `ReadBlockIteratorDB` implements the `IReadBlockIterator` introduced in section 6.4. The focus of the following section will be on the implementation of the `ReadBlockIteratorDB`.

In order to implement the `IReadBlockIterator`, the `ReadBlockIteratorDB` has to accomplish four tasks.

- An SQL query which is sent to the database must be generated in accordance with the requirements of the result. The factors influencing the generation of the SQL code are first the `dataSelection` defining the entries in the `SELECT` list, and second the other possible parameters describing the `FROM` and `WHERE` parts of the SQL query. It has to be emphasized, that the `SELECT` list is generated as conservative as possible. This means that maximal the fields determined by `dataSelection` are loaded into the `ReadBlockDB`. Since the generation of a `ResultSet` which contains reads and matches is not feasible when the result contains a large number of rows, the matches are only loaded on demand. As the result of the query to the database, the `ReadBlockIteratorDB` expects to get a `ResultSet`. This task is executed within the constructor of the `ReadBlockIteratorDB`, which informs the database object about the requirements and expects a `ResultSet` in return (Listing 6.15).
- A `ResultSet` is not an iterator. Hence the functionality has to be emulated by inserting the `hasNext()` method shown in Listing 6.16. This is implemented by using two methods of the `ResultSet`. Initially it is checked whether the set is empty (`set.isBeforeFirst()`). This is achieved as a side effect of the method as long as the cursor is on the initial position. This is due to the fact that the second method (`set.isLast()`) fails its own test if the set is empty. The second method then checks for every non-empty set by verifying whether the cursor is at the last position or not.
- If an `IReadBlock` is requested, the `ReadBlockIteratorDB` must be able to build this block from the `ResultSet`'s data. The method `buildReadBlock()` aims to complete this task. By a given parameter (see Listing 6.16) the method gets information about the position of the fields and their datatypes in the `ResultSet`. This is carried out in order to build a `ReadBlockDB` and return it.
- In order to save the databases resources, the `Statement` and the `ResultSet`<sup>24</sup> must be closed when the cursor reaches the last row of the result. This is achieved either explicitly by using the `close()` method or implicitly when the last row of the `ResultSet` is reached.

---

<sup>24</sup>A `ResultSet` is existently connected with the `Statement`. Hence `Statement` must not be closed as long as a `ResultSet` is read.

Listing 6.15: Simplified constructor and database query method in ReadBlockIteratorDB

```
1 public class ReadBlockIteratorDB implements IReadBlockIterator {
2
3     private IDB db; //database object
4
5     private float minScore;
6
7     private float topPercent;
8
9     private DataSelection dataSelection;
10
11    private ResultSet set;
12
13    private Statement statement;
14
15    private boolean isEmpty = true;
16
17    private HashMap<String, Integer> columnOrderRead = null;
18
19    public ReadBlockIteratorDB(IDB db, float minScore, float topPercent,
20        DataSelection dataSelection){
21
22        this.db = db;
23
24        this.minScore = minScore;
25
26        this.topPercent = topPercent;
27
28        this.dataSelection = dataSelection;
29
30        getResults();
31    }
32
33    private void getResults(){
34
35        columnOrderRead = SelectionAnalyzer.analyzeDataSelectionForReadData
36            (dataSelection).getSecond(); //retrieves the ordering of rows
37            in the ResultSet
38
39        Pair<ResultSet, Statement> pair = db.getReadData(dataSelection);
40
41        this.set = pair.getFirst();
42
43        isEmpty = !this.set.isBeforeFirst(); //is the resultset is empty?
44
45        this.statement = pair.getSecond();
46
47    }
```

Listing 6.16: Standard iterator functions implemented in ReadBlockIteratorDB

```

1 public boolean hasNext() {
2
3     if (!isEmpty) {
4
5         if (!set.isLast()) {
6
7             return true;
8
9         } else {
10
11             return false;
12
13         }
14     } else {
15
16         return false;
17
18     }
19 }
20
21 }
22
23 public IReadBlock next() {
24
25     if (hasNext()) {
26
27         set.next();
28
29         return buildReadBlock(); //build the ReadBlockDB from the actual
30             row of the ResultSet
31     } else {
32
33         close();
34
35         return null;
36
37     }
38 }
39 }
40
41 private IReadBlock buildReadBlock(){
42
43     return BlockBuilder.buildReadBlock(set, db, dataSelection,
44         columnOrderRead, minScore, topPercent);
45 }

```

#### 6.5.4 Alter data

If changes on the data occur, it is necessary to ensure that the changes can be persistently written to the database. Examples of this are changing the number of reads or matches

or updating fields in `IReadBlock`. However, this does not include altering classification details, because in this case a special treatment is provided.

Besides methods specially developed to alter global data, such as the number of reads for which a static query is provided (which are not introduced in this work), there are interesting methods for writing dynamic content, such as fields from `IReadBlocks` and `IMatchBlockss`. For both kind of blocks three parameters are required: the `uid` determines the associated row in the database, the `dataSelection` is used to filter fields in order to be updated and the block which should be written. Not the complete block is written into the database, but only the fields defined in `dataSelection` are altered. In accordance with `dataSelection`, fields are picked from the block and updated in the final step. On the SQL level is this implemented by using one `UPDATE` statement per table which is included in the update. Hence a maximum of four update statements is executed: `read`, `classread`, `match`, `blasthit`.

Although it is possible to update many blocks by invoking this method several times, it might not be too efficient, when compared to the speed a database reaches if accurate techniques are used. For this task Java provides batch updates. They allow to bundle several similar SQL commands into one batch, which is only executed once. Without going into details (but since a batch execution leaves the database more freedom in applying the changes), the operation performance would profit from a batch execution. For updating the `IMatchBlocks` is this already conducted, since with a `read` a set of matches is updated.

### 6.5.5 Recalculation of classifications

As long as the algorithm for recalculation of the classifications happens within MEGAN's scope, the database's work is reduced to loading the data and later on storing the recalculated information back to the database. The loading is conducted by getting all reads and matches, the classification data only respectively, which can be done by using the methods returning a `ReadBlockIteratorDB`. The second step is to store the recalculated classifications of the reads back in the database.

In order to review the database design, there is a table `classread` designed precisely for coping with classification data only. Therefore, the storage of the recalculated data will only alter rows in this table. For performance issues this is not a negligible factor.

The instance recalculating the classifications returns two components (described in section 6.1.2). The first component is a list of items where every item contains the `readUid` of a read and an array with the freshly recalculated values of all available classifications. The second component is an array containing the names of the recalculated classifications. This array is necessary for mapping the classifications against the values inside the array.

The `DBConnector` splits up the list into single items and transfers them one by one to the database object. Hence for every read the database object is called once. In the database object, the item is translated into `UPDATE` commands; for every classification which is to be updated, one `UPDATE` command is generated. The resulting SQL commands are batched and executed against the database. Since the changes in `classread` should also affect the table `classificationblock`, the function shown in appendix .2 keeps this table updated.

## 6.5.6 Creating new datasets

Creating a dataset (Listing 6.8) consists of two steps. First, the `IReadBlockIterator` determining the reads to be stored to the database are analyzed and written to the disk. Fields chosen by the `dataSelection` are written into temporary csv files - one csv file per table. The second step is to copy these csv files to the database.

As described earlier, single `INSERT` statements do not perform when a large number of rows has to be inserted. To insert row by row with single SQL statements might perform up to several hundred statements, but not for the number of statements which have to be inserted when creating a standard-sized dataset<sup>25</sup>. Therefore, a DBMS provides tools for loading csv files<sup>26</sup> containing many rows with one query into the database. For PostgreSQL, this is the `COPY` command and for H2 is the `MERGE INTO` command utilized. In addition, PostgreSQL provides a special Java library in order to accelerate the insertion of a large number of rows. The package `org.postgresql.copy.CopyManager` is used for copying the rows from a csv file to the desired table by using a Java `InputStream`. This is important since PostgreSQL is otherwise not allowed to load csv files from temporary folders.

It is also important that the insertion to the tables is carried out without altering or deleting constraints or indexes. Chapter 8 will show that this leads into a rather poor performance. Alternative ways to copy csv files to the database which are already implemented but not included into MEGAN's scope will also be introduced. This is due to the fact, that the way to store datasets in the database has not been decided yet. As the current solution, storing datasets into one set of tables has been proposed. Another idea is to create one schema per dataset with its own closed set of tables.

The creation of a dataset is from the computational view the largest task for a database, since the size of the data can easily exceed several gigabytes. But it is not just the size which makes this operation time-critical.

- The rows have to be inserted into the tables.
- For every row the indexes have to be updated. This is the critical operation, because its complexity grows faster than the mere insertion time (which is nearly linear).
- For columns, such as `text`, in the `blasthit` table the value needs to be compressed<sup>27</sup>.
- Some additional calculations need to be done after the insertion, such as creating the `classificationblock` data from `classread`.
- Maintenance of the database, such as `analyze`, should be executed.

---

<sup>25</sup>10-20 million rows

<sup>26</sup>Other file types are supported, too. But in this approach, only csv files are used.

<sup>27</sup><http://www.postgresql.org/docs/8.0/interactive/storage-toast.html>



## 7 Prospects

So far MEGAN uses the database to load and store data. But a DBMS provides a wide variety of opportunities even in calculation matters. The improvements which will hopefully be achieved when functions of MEGAN are outsourced to the database are based on two principles. The first principle is that a database has its own memory. In this memory section, calculations can be executed instead of sending large blocks of data to MEGAN's scope and later re-calculated back. Hence the database can benefit from its own property and work on a large number of rows at the same time instead of emulating other structures, such as iterators for Java. The database also gathers information about the data which is then used to determine the best execution strategy depending on the data structure. In addition, the traffic from and to the application is minimized. In this chapter an example of the use of the database's workforce is shown by presenting algorithms for the calculation of the **Taxonomy**<sup>1</sup> classification.

In addition to the idea of outsourcing calculations to the database's scope, some other points have not yet been implemented satisfactorily. Loading a large number of rows into the database with built-in bulk loading functions, such as `COPY`, works fine in PostgreSQL only as long as the tables are empty or not too many rows are present. However, not the insertion of rows but the update of the index structure seems<sup>2</sup> to be the critical operation for bulk loading. For this purpose a package for PostgreSQL - `pg_bulkload` - is introduced.

The third field where optimizations can take place is MEGAN's manner of communication with the database. As described, a database does not perform very well when taking its full potential into account when many queries returning single rows are executed. It is more efficient to execute one query returning a set of rows containing the same information in order to reduce transfer costs and support the databases ability to provide and store data.

Finally, two points describe the ideas how to extend MEGAN's functionality by using the database's flexibility. For example, it is possible to merge datasets for an display output with no physical changes. Inserting reads into a dataset is no problem either for a database, but is not supported for RMA files. Finally, the idea about the usage of the `DBConnector` is introduced, since the current solution is not satisfactory.

---

<sup>1</sup>The calculation is just provided for PostgreSQL. To avoid misunderstandings, the **Taxonomy** classification is in RMA called `taxonid`.

<sup>2</sup>Examples can be found in 8

## 7.1 Calculation of the taxonomy classification in PostgreSQL

The calculation of the taxonomy classification is based on the lowest common ancestor problem. For a set of nodes in a rooted tree exactly one lowest node can be found, which contains all nodes of the set as descendants. In the case of MEGAN the taxonomy values of the match define the lowest common ancestor node then assigned to the enclosing read. There are different approaches to solve this problem. In this chapter, the presented approach is based on the preorder and postorder ranks of nodes within a tree. Since this tree is stored within the database, the whole calculation can be performed in the database's memory. The tree structure where the algorithm will be calculated on is provided by National Center for Biotechnology Information and can be found online<sup>3</sup>.

First, the given tree structure needs to be refined and the pre- and postorder ranks plus a tree datatype for PostgreSQL the ltree need to be calculated. Consequently, the algorithm to calculate the LCA is developed.

### 7.1.1 Preliminaries

The file *nodes.dmp* in the taxdmp contains the needed tree structure<sup>4</sup> provided by node ids and their parents. After deletion of redundant information, the entries in the file can be loaded to the database into the newly created table node.

Listing 7.1: Create Table and Index for Nodes

```
1 CREATE TABLE node(  
2 id BIGINT,  
3 parentid BIGINT,  
4 pre BIGINT,  
5 post BIGINT,  
6 tree LTREE,  
7 PRIMARY KEY(id)  
8 );  
9  
10 CREATE INDEX node_pre_idx ON node(pre);  
11 CREATE INDEX node_post_idx ON node(post);  
12 CREATE INDEX node_ltree_idx ON node(tree);
```

### Preorder, postorder

Preorder and postorder assign to all nodes in a rooted tree a unique rank during a depth first traversal. The preorder algorithm assigns to every node a rank when visited first, whereas the postorder assigns the rank when it is seen the last time during traversal (Fig. 7.1). The calculation of the values is executed in the database by using the recursive functions written in plpgsql (appendix .3).

An example on the pre and post ranks is given in Fig. 7.1.

<sup>3</sup><ftp://ftp.ncbi.nih.gov/pub/taxonomy/taxdmp.zip>

<sup>4</sup>The file is a csv file with around 600000 rows and 14 columns. For the calculation only a subset of the columns is needed.

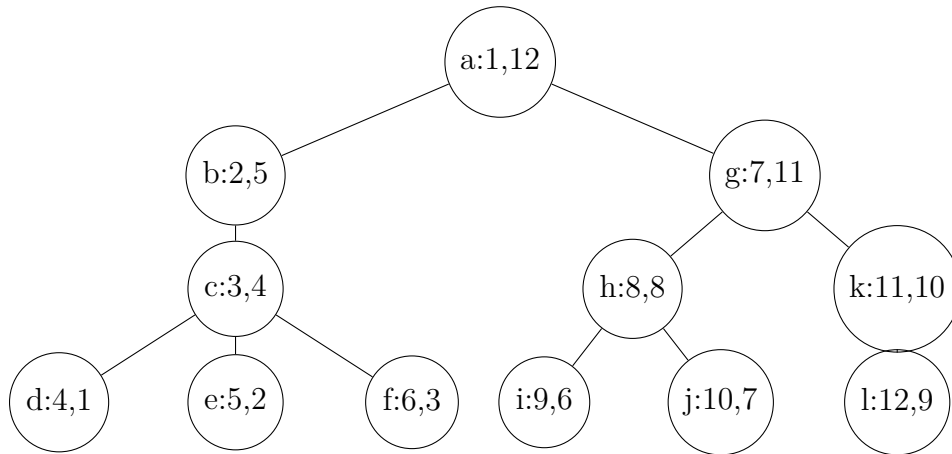


Figure 7.1: Rooted Tree with Pre and Postorder Ranks (label:pre,post)

## Ltree

In addition to the pre- and postorder ranks, PostgreSQL needs a special tree representation of the nodes of the freshly installed table. This is due to the fact that the built-in LCA algorithm works on this datatype. The `ltree`<sup>5</sup> assigns to every node the path to the root defined by a field of all ancestor nodes separated by a dot and is added to the table as the column `tree`.

## Resulting table

Once all parameters have been calculated, the table node contains all information needed to calculate the LCA<sup>6</sup> within the database. As an example of a node, Figure 7.2 displays the table derived from Fig. 7.1.

## 7.1.2 Solving the LCA problem in PostgreSQL

The LCA problem of a set of nodes can be reduced to a problem on maximal two nodes. For the remaining two nodes PostgreSQL offers a fast LCA algorithm. The first task is to find these two nodes, then the algorithm itself is presented.

### Finding the two crucial nodes

PostgreSQL's LCA algorithm always returns the correct result for two nodes, if one of them is the left-most node and the other one the right-most node of a set of nodes in a tree. This is where the pre- and postorder ranks come into play. The left-most node has the lowest pre value whereas the right-most node has the highest post value. This includes that if one node is found containing largest post and the lowest pre value, this node is already the LCA, since all other nodes are in its subtree.

<sup>5</sup>The datatype can be found in `./share/contrib/ltree.sql`. <http://www.postgresql.org/docs/current/static/ltree.html>

<sup>6</sup>The LCA algorithm needs to be installed first; it can be found in `./share/contrib/tablefunc.sql`.

ID	ParentID	Pre	Post	tree
a	0	1	12	a
b	a	2	5	a.b
c	b	3	4	a.b.c
c	b	3	4	a.b.c
d	c	4	1	a.b.c.d
e	c	5	2	a.b.c.e
f	c	6	3	a.b.c.f
g	a	7	11	a.g
h	g	8	8	a.g.h
i	h	9	6	a.g.h.i
j	h	10	7	a.g.h.j
k	g	11	10	a.g.k
l	k	12	9	a.g.k.l

Figure 7.2: Tree Structure from Fig. 7.1 as a Table

## PostgreSQL's LCA

The built-in function `lca(node1, node2)`, which consumes the `tree` column of two nodes, returns the LCA for these. Since they are chosen as described in the previous paragraph, the result is valid for all nodes of the initial set. The algorithm itself is simple since it consumes the paths of the two chosen nodes from the root to the nodes. The paths are compared and once they diverge, the LCA is found.

### 7.1.3 LCA in MEGAN

The table `node` already exists and the indexes are created. The indexes are essential for finding the correct rows identified by an id as fast as possible. A solution with a non-indexed table would make no sense since a repeatedly executed sequential scan through `node` would lead to slow execution. Now the queries on MEGAN's DB are introduced.

Every read contains a set of matches. The number depends on the parameters defined during the creation of the dataset and previous filtering. Most of the matches have a taxonomy value assigned, presenting nodes in table `node`. So what the algorithm should do is to calculate the LCA for all reads and then representing the taxonomy of the read derived from the taxonomy values of their matches. But not all matches are included in the calculation. MEGAN provides filtering by the bitscore value. A low bitscore determines a match with low quality which would influence the whole calculation and lead to undesirable results, e.g. a LCA in a higher level of the tree. The filtering is divided into two sections. First, there is a border of the minimal bitscore. All matches with lower bitscore are ignored. The second border is defined by the match with the best bitscore - for every read there is a match with the best bitscore - and a percentage. Matches having a higher bitscore then a predefined percentage of the best match and also pass the first border only are included in the calculation.

For all remaining sets of matches the nodes with the highest post- and the lowest

pre-value are shown in Listing 7.2.

Listing 7.2: Find min(pre) and max(pos) for all reads

```
1 SELECT min(pre) AS min_pre,max(post) AS max_post,match.read_key AS
   read_key
2 FROM (SELECT r.read_key AS read_key,m.match_key AS match_key,m.
   taxonid AS tax
3         FROM read r, match m
4         WHERE r.file_key = 'DEFINE FILE'
5         AND r.read_key = m.read_key
6         AND taxonid <> 0
7         AND bitscore > CAST('MINIMAL SCORE' AS REAL)
8         AND bitscore > r.max_score*'PERCENTAGE'
9         ) AS match,node
10 WHERE match.tax = node.id
11 GROUP BY read_key
```

The second step of the calculation is to feed the resulting nodes into the LCA function which returns lowest common ancestor (Listing 7.3).

Listing 7.3: Execute the LCA function.

```
1 INSERT INTO classread VALUES
2 (SELECT read_key, n3.id, VALUES('Taxonomy')
3 FROM node as n1,
4      node as n2,
5      node as n3,
6      'result from first query'
7 WHERE n1.pre = min_pre
8 AND n2.post = max_post
9 AND n3.tree = lca(n1.tree, n2.tree)
10 )
```

### 7.1.4 Putting all together

In order to perform a complete and consistent operation, the whole calculation needs to be done within a transaction. In the transaction is first all data of the recalculated dataset from the taxonomy classification in classread and table classificationblock deleted. Then the recalculation is executed. After completion the classificationblock is updated.

Since performance tests show a remarkable speedup, if constraints on the table classread are deleted before the execution and created after the recalculation, this will be considered in future versions of MEGAN.

## 7.2 pg\_bulkload

A database is designed to provide very fast and specific data. But since the main structure behind the fast data access, the index, is often based on a tree, the B+-tree, which is expensive to alter, the insertion of a large number of rows is a task consuming more time then simply storing data into an RMA file. This is a particularly big problem since the time to update the index grows faster then the insertion time for rows which is

nearly linear - no matter how many rows already exist. Hence it is valid, that the more rows are present in a database, the more time-consuming will the update be. Therefore either changes in the database design or advanced algorithms are required in order to lower the time consumption to another level. Here the algorithms are introduced.

### 7.2.1 COPY on an indexed table

A 'normal' PostgreSQL COPY function on an indexed table will perform poorly. The incremental index update is an expensive task and should only be utilized if a small number of rows is to be inserted.

### 7.2.2 Deleting Index + COPY + Create Index

Since the incremental update of indexes performs poorly, another idea is to delete the index and recreate it after insertion. This will speed up the insertion of a large number of rows since it is much easier for the database to build an index from the scratch then to update it without knowing the number of forthcoming rows. But there are two problems which make this approach unsuitable. A database can be used always, even though one operation loads a new dataset, another operation is still allowed to query data. If an index crucial for fast data access is deleted for recreation, this lowers the speed of the application, because instead of an index scan, for example a sequential scan must be performed. In order to create an index, also all entries of a table which should be part of the index must be loaded from the disk to the memory. This produces an overhead which might not be negligible for tables with a large number of non-indexed rows <sup>7</sup>. However, this approach performs better then the COPY on indexed tables.

### 7.2.3 pg\_bulkload

Pg\_bulkload is a tool for PostgreSQL introduced by Toru Shimogaki<sup>8</sup> in 2006 dealing with the problem described above.

First, the rows from a csv file are inserted into the table, but instead of directly updating the index is this operation bypassed. When writing rows to the disk they appear for a short time in the memory. This time span is used to extract the fields which are to be indexed, sort and keep them in memory while the rows are stored on the disk. The original index is loaded - typically parts of the index are already present in the memory - and merged with the new sorted values and create a new index. Since the leaf entries of an index, respectively the new values, are already sorted, is this a cheap operation compared to the two other approaches.

The problem with pg\_bulkload is that it is not that easy to install and also seems not to be completely stable on the Windows platform. Still, the performance described in <http://pgbulkload.projects.postgresql.org/> is remarkable.

---

<sup>7</sup>There is of course a long list of parameters also influencing this operation not introduced in this thesis.

<sup>8</sup>[http://pgfoundry.org/docman/view.php/1000261/456/20060709\\_pg\\_bulkload.pdf](http://pgfoundry.org/docman/view.php/1000261/456/20060709_pg_bulkload.pdf)

## 7.3 Supporting database characteristics

A database is optimized to work on a large number of rows at the same time. Even though the retrieval of single rows is fast when performed through index access, the performance suffers when a large number of rows is queried one by one. This is mainly due to the higher organisational effort in the database and in MEGAN. In some cases MEGAN is for example able to determine during iteration that in the foreseeable future more data of the same kind will be needed. However, most of the operations in MEGAN are based on simple algorithms, hence the retrieval of the data is the bottleneck. On the other hand it is faster for MEGAN to retrieve one row then to wait until a set of rows - even though most operations in MEGAN are non-blocking (no sorting, no aggregate) - is generated. It would be interesting to discover how large the set of rows has to be to beat the performance of the row-by-row retrieval and in which situations could MEGAN profit from this idea.

## 7.4 Refinements on the databases tables

As the database design has not been tested in the daily use yet, possible bottlenecks have not yet been identified. There are some ideas for refinements within the tables.

Under some circumstances - for example under regular recalculation of the classifications - a table classmatch (Fig. 4.2) could speed up the retrieval of the classifications of the matches. The retrieval of the classification data is then cheaper since disk pages only contain classifications. Hence the number of disk pages to be loaded is smaller.

Another idea comes with the number of rows in the table. As described before, the number of present rows in a table has a major influence on the speed of insertion, respectively the update of the index. But if a table is empty, the insertion works fast even with built-in insertion tools. Maybe another approach in storing the datasets helps. So far, all tables in MEGAN's DB are present in the *public* schema<sup>9</sup>. Another solution would be to create a new schema for every dataset with the schema name as identifier and all tables besides header. Then the public schema would be used to store the information of the header table and delegate function calls to the datasets via schema names.

## 7.5 Unlimited MEGAN

The versions of MEGAN using RMA as the only data object are limited by the static manner of the RMA files. Since MEGAN is multithreaded, several datasets can be opened at the same time, but the open datasets are separated not only physically, but also on the screen.

A database is more flexible in this. Reads can be extracted, inserted or deleted without any physical changes. Merging whole datasets is also possible. Another idea is to add and delete classifications on the fly. To add a new classification is a cheap task for the database since the data dependence is kept at a minimum. Same applies to all changes which include deletes or insertions to a dataset. These might be interesting features for future versions of MEGAN.

---

<sup>9</sup>A schema is a subsection of a database. Every table is assigned to one schema.

## 7.6 IConnector/Connection handling

For a DBMS, an important task is to maintain open connections. Since MEGAN uses the database in an embedded mode, only one connection is needed. H2 for example supports only one connection when started in the embedded mode. Consider that the user opens more than one dataset stored in the same database. The same connection should be used even though different instances of the `DBConnector`, respectively the `IConnector`, are created. Therefore a decoupling between the `DBConnector` and the `Connection` is required. One idea is to store open connections as an array<sup>10</sup> into a class which is persistent during the complete runtime of MEGAN. If then a connection to a database is needed, a suitable connection can be reused. The other idea is to keep `IConnector` instances open as long as a dataset is open. This can be achieved by storing their instances in an array of a persistent class. If then a `IConnector` of the same type like `DBConnector` to PostgreSQL is needed, the connection of the other `DBConnector` can be extracted and reused.

Both approaches have certain advantages.

- A `IConnector` for a dataset is created once and the reused until the dataset is closed.
- Connections are easier to create, maintain and close.

Currently, a new `IConnector` is instantiated every time the data is queried from a dataset. The context, or in other words the knowledge of which dataset from which data object should be used, is provided by an integer determining the type of the data object and the name of the dataset. For an RMA file is this a suitable solution, since for every data access the connection to the file is opened and directly closed again. But for a database is the process of opening and closing a connection a very resource-intensive. Hence solutions which allow to maintain database connections or `IConnector` objects would be desirable.

---

<sup>10</sup>Connections to several database can be open at the same time.



# 8 Runtime comparison RMA vs. PostgreSQL

The main topic of this thesis is to compare the performance of RMA files to the in the previous chapters developed database. It is hoped that a mature database system optimized to the needs of maintaining large sets of data and known to be able to provide stored information in a fast manner will enhance the potential of MEGAN. Another contribution of a database is the advanced flexibility which cannot be measured, possibilities of which were introduced in the previous chapter.

First, the time to create datasets is measured. Since the index creation for databases is costly and `pg_bulkload` is not included yet, it is expected that the creation of an RMA file performs better. It is also interesting to compare the size of the resulting datasets, since RMA files as well as PostgreSQL provide compression.

One function of MEGAN is to search through fields containing plain text for patterns. Tests on the fields `header` and `text` (of a match) are evaluated. The - from the database programmer's point of view - most interesting test is the comparison of the calculation of the classifications. Besides the standard implementation (calculating the classification within MEGAN's memory), pure PostgreSQL algorithm for the taxonomy classification (introduced in the previous chapter) is tested as well, leading to promising results.

Although two DBMS are introduced in this thesis, only PostgreSQL is included in the tests.

## 8.1 Tested datasets and system properties

The focus of this chapter is the comparison of RMA files and datasets stored in the database. For this task three different datasets<sup>1</sup> are tested. All of them are provided as RMA files. The runtime of the tests differs from system to system. Hence a detailed view to the testsystem is provided, too.

### Datasets

**ecoli** Reads: 2,000; matches: 42,374;

**mouse** Reads: 10,931; matches: 822,516;

**red** Reads: 334,386; matches: 5,382,776;

---

<sup>1</sup>The content and the source of the datasets is not important. Besides `ecoli` which is a test-dataset, `mouse` and `red` can be found as RMA files on MEGAN's project homepage <http://www-ab.informatik.uni-tuebingen.de/software/megan/welcome.html>.

## Hardware, software

**CPU** AMD Athlon 64 X2 Dual Core 5200+

**Main Memory** 2.00GB DDR2 667Mhz

**Hard Disk** Seagate SATA 320GB 7200rpm

**Operation System** Microsoft Windows XP Professional, Service Pack 3

**MEGAN** 4.0alpha1 (July 13th, 2010); standard memory allocation

**PostgreSQL** Version 8.4.2; `shared_buffers = 512MB`; `work_mem = 20MB`; `maintenance_work_mem = 256MB`; `seq_page_cost = 2.0`; `random_page_cost = 3.0`;

**Java** 1.5<sup>2</sup>

## Measurement methods

The runtime of the tests is measured in seconds by using a stopwatch implemented in Java (`System.currentTimeMillis()`). For pure database operations the `date` function of PostgreSQL is used.

When possible, MEGAN's GUI is bypassed and substituted by a single class directly communicating with either `DBConnector` or the RMA implementation, the `RMAConnector`.

## 8.2 Create a new dataset

Creating a dataset is the most time consuming operation since it includes sequence comparison and parsing of BLAST files. To measure the pure speed of the insertion into either RMA or PostgreSQL, the datasets from already created RMA files are used in order to provide the `IReadBlockIterator` required for the creation. Four different scenarios are evaluated. Apart from the creation of an RMA file, a list of database scenarios is tested. On one hand the insertion into an empty PostgreSQL database is executed. On the other hand, and in order to show the impact of index update, the datasets are loaded into a database with a present *red* dataset. Additionally, a feature for fast copying (which however has not been integrated yet) is introduced - to copy a dataset into a database with no indexes and integrity constraints. These are then created after termination of the copy-to-database command.

The creation is implemented by using the `createNewDataset` method from either `RMAConnector` or `DBConnector`. The required `IReadBlockIterator` is provided by the `getAllReadsIterator` function call on test datasets stored as RMA files. The `DataSelection` is implemented so that the largest set of possible fields in `IReadBlocks` and `IMatchBlocks` are included in the creation.

Fig. 8.1 and table 8.1 depict the results. As expected, the RMA file leads with a large advance.

The performance of creating a dataset in the database suffers mainly from one fact: the reads to be copied to the database provided as an `IReadBlockIterator` have to

---

<sup>2</sup>Java 1.5 is the platform MEGAN uses.

be transformed into csv files and copied to disk. This operation is already as costly as the whole creation of an RMA file, but at the same time inevitable since bulk load algorithms of DBMSs require these files.

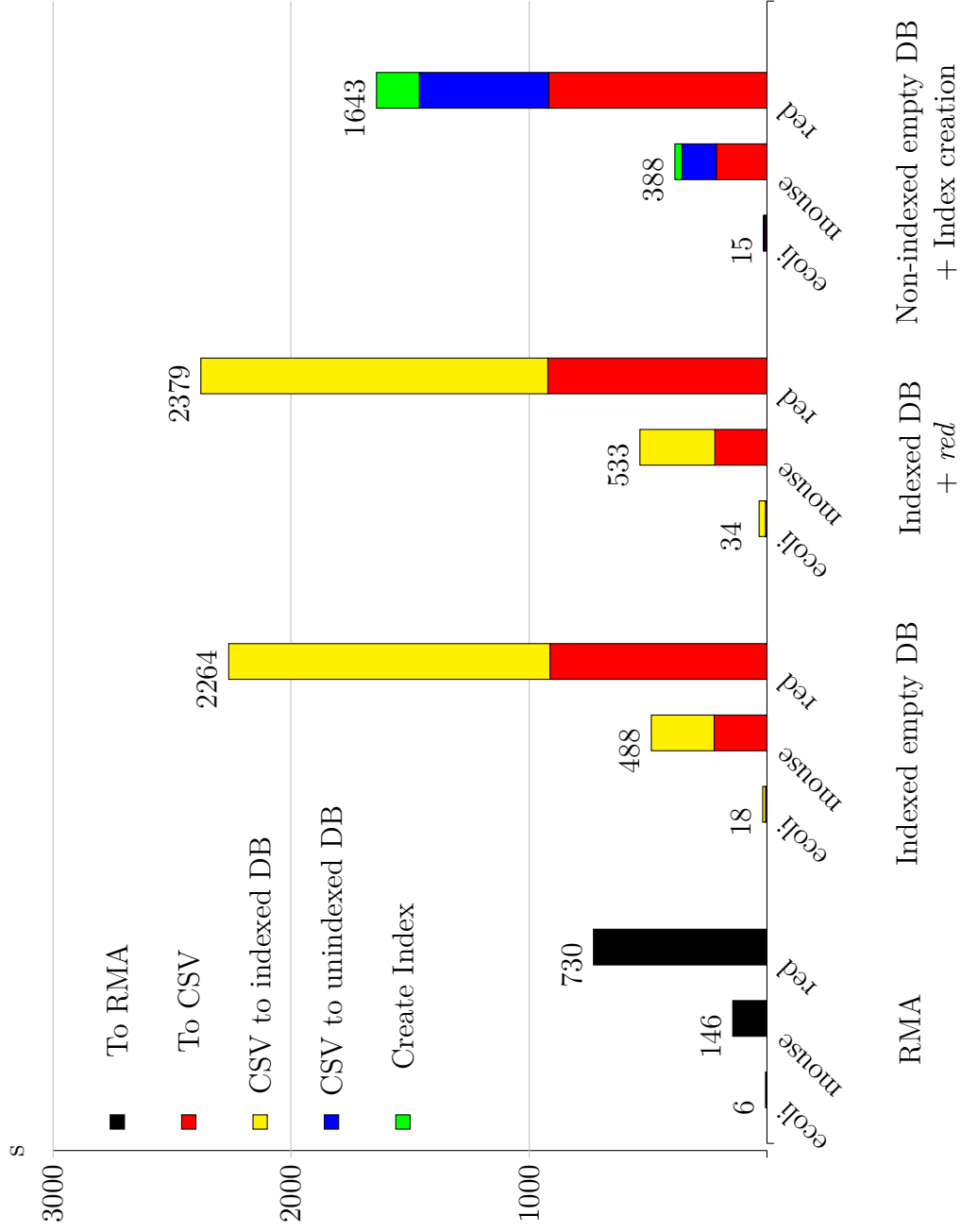


Figure 8.1: Copy catasets to RMA, to empty DB, to with *red* dataset loaded DB and to empty, non-indexed DB with index creation

	Dataset	To CSV	To DB	INDEX
Indexed empty DB	mouse	222	265	
	red	912	1352	
Indexed DB + red	mouse	219	314	
	red	921	1458	
Non-indexed empty DB + Index creation	mouse	213	145	30
	red	918	545	180

Table 8.1: Creation Times for Database Datasets

The next step in the creation of a dataset is to copy these csv files to the database. Here the execution time mainly depends on the number of present rows in the tables and whether the tables are indexed and whether the constraints are present. Fig.8.1 shows three different scenarios. Loading datasets into an indexed but empty database is compared to the runtime of the insertion of the same datasets when the database is not empty. While the creation of a list of RMA files always needs the same runtime, no matter if other RMA files have been created before, the insertion time to a database raises when other datasets are present. Here the performance of the insertion of mouse and red increased by 50 respectively 100 seconds when a large dataset is already present in the database.

The third approach (not included in MEGAN yet) is to delete all constraints and indexes prior to the insertion of a new dataset and recreate them after termination. The runtime for mouse dropped from 488s to 388s and for red from 2,264s to 1,643s. However, the disadvantages of this approach might be serious. During the insertion no indexes are present, therefore the performance of other open datasets present in the same tables suffers. Hence future versions of MEGAN should use different solutions like integrating `pg_bulkload` combining the advantages of an always indexed database and a fast insertion time. Another idea is to create a new set of tables for every dataset and therefore keep datasets physically detached.

Although the optimizations presented above lead to a fast insertion of present csv files, this does not speed up the creation of csv files. They need, mainly due the blocking character of their creation, different solutions. So far, once all csv files containing the complete dataset are created, they are then sent to the database. One solution to this problem might be to unblock this operation and instead of creating 'complete' csv files, a possible solution could be to create a list of files with a predefined rowcount which then can already be written to the database while other csv files are still in the creation. For example, creating csv files with 10,000 rows each and send them to the database subsequently.

However, the creation of a dataset is a process executed once and since the parsing of the BLAST file, not included in this measurements, is the most time intensive operation. Hence the timespan when a database is slower then a RMA file - when it comes to creation - is a non-crucial factor.

	ecoli	mouse	red
RMA	13	471	1630
PostgreSQL	29	864	3745

Table 8.2: Size requirements for storage in postgresQL and RMA in MB

### 8.3 Disk usage RMA vs. database

The BLAST file derived from a comparison step with a reference database can easily exceed some gigabytes. MEGAN parses this file and stores its information into an RMA file or into a database. In order to save space, RMA files can be compressed, which is due to the fact that the text field in match is a large plain text very effective. A DBMS can provide functions to compress single columns and complete databases<sup>3</sup>, too. In PostgreSQL this is achieved through the TOAST (The Oversized-Attribute Storage Technique) technique which allows to store large objects, first compressed and second stored in 'out of the line' tables. But although the TOAST was tested with several parameters<sup>4</sup>, the compression failed. Hence, so far the datasets are stored uncompressed, where 90 per cent of the size result from the BLAST field.

Table 8.2 shows the resulting size after insertion of the test datasets in either RMA or PostgreSQL. Since PostgreSQL provides compressing techniques, there must be a solution to shrink the BLAST field to a desired size. For the case the compression cannot be achieved in the database, it is also possible to use Java for compressing the BLAST field and store the result as a compressed object to the database. Testing compressing the BLAST field using the GZIP algorithm showed a reduction of the dataset size to the RMA level.

### 8.4 Text searches

When MEGAN initially parses a BLAST file in order to create a new dataset, the BLAST text contains much more information than actually extracted. Only structural information, such as the taxonomy or the cog values or the reference ids and scores, is parsed and stored as single fields. These fields are necessary for the daily usage of MEGAN as seen in the preceding chapters.

Since the BLAST result is stored as well, its information is still accessible. MEGAN allows to search through the field containing the BLAST result, for example by using regular expressions. This is interesting for the user who wants to extract the additional information, but also for the calculation of the seed classification, which under certain circumstances needs to perform a text search as well. In addition to the BLAST field, the search can also be executed on other fields with a string as datatype, such as the header field of a read.

For the comparison of RMA files to datasets stored in the database two tests are executed. In the field header (in MEGAN's GUI this field is called readname) of the dataset red, a regular expression (`?*645.*`, 1964 hits for the whole dataset) search is

<sup>3</sup>H2 can maintain read-only database in .zip archives.

<sup>4</sup>SET STORAGE [MAIN, EXTENDED, EXTERNAL]

	RMA	DB
Search on field header in dataset red	3.35 min	6 s
Search in BLAST text in dataset mouse	31.12 min	3.01 min

Table 8.3: Regular expression seektime RMA vs. DB

performed. The second test executes a search on the field containing the BLAST result of the mouse dataset using a regular expression, too (`.*Transposase.*`, 613 hits in the whole dataset). The result is shown in table 8.3.

The result shows that the database performs the search on the `header` field around 30 times faster than the RMA file. This is mainly due to two facts. The database performs the search in its own memory, whereas parts of the RMA file need to be loaded into MEGAN's scope in order to be scanned there. With the number of reads the overhead grows, which slows down the execution. The second aspect is that for the column `header` an index in the database was created. Hence the database is able to perform a pure index scan.

The search in the `text` field of match performed in the database is around 10 times faster than the same test in the RMA file. Similarly to the first scenario, the scan is performed within the database's memory. Hence overhead is reduced. But it is in this case risky to draw conclusions since the database provides no compression yet which speeds up the execution of the scan.

## 8.5 Update classification

Unlike the creation of a dataset, the update of classifications - the second operation requiring large computational efforts - is performed more than once. The classification which consumes the most time when it comes to its calculation is the taxonomy classification. Besides the two known parameters `minScore` and `topPercent` determines the third parameter `minSupport` the number of reads which have to be assigned to a class. If the number is lower than the `minSupport` the reads are assigned to the *Not Assigned* class. Hence this test compares the speed of the calculation of this classification for three different approaches. In the first scenario the calculation within the database scope introduced in section 7.1 is executed. Hence MEGAN's workload is reduced to execute the initial command. The other two approaches use the RMA file, respectively the database, for data loading and storing. The calculation is executed within MEGAN's scope.

Fig.8.2 depicts the runtimes for the three different techniques tested for the three datasets, *ecoli*, *mouse* and *red*. The calculation purely evaluated in the database shows the best performance for every dataset. Also the execution which uses the database in load/store mode shows promising results for the *mouse* and *ecoli* dataset. But for the *red* dataset the performance is rather poor. This seems to be mainly due to the size of the dataset and the function to keep the table classificationblock up to date (appendix .2). For a large number of changes in `classread` this function needs to be bypassed. Since the classification data for a complete dataset is changed and therefore all rows of the table `classread` which belong to that dataset are altered, a possible solution for

accelerating the insertion time<sup>5</sup> is to delete the rows of table classread which belong to the dataset and taxonomy classification. Then the altered classification data is written into a csv file and subsequently copied to the classread table. As the last step, the table classificationblock is updated. This comes with another large advantage. So far, the process of recalculation, respectively the process of updating the dataset, cannot be cancelled since the consistency on the data would be lost then. When combining the steps of deletion, insertion and updating into a database transaction, this operation can be cancelled anytime without losing the consistency on the data.

With these changes applied, the calculation of the taxonomy classification performs for all datasets better. The mouse dataset needs only 25 seconds instead of 48 seconds. The calculation of the classification costs 20 seconds. The copying to disk and the loading to the database costs 4 seconds. The final calculation of the classificationblock which is executed within the database only consumes less than a second.

The recalculation of the red dataset shows a different and for the author not understandable behavior. The calculation of the classification which also includes the retrieval of all matches costs 421 seconds. But copying the resulting information to the disk consumes with 309 seconds far too much time<sup>6</sup>. The final insertion to the database with recalculation of the classificationblock costs further 21 seconds. Hence even including the unreasonable time consumption for the csv creation is the calculation with 741 seconds currently around three minutes faster than the RMA based recalculation.

However, future versions of MEGAN should rather outsource this calculation to the databases scope. Fig.8.2 shows the potential of database management system when simple calculations on large sets of data are outsourced to the database's scope.

---

<sup>5</sup>MEGAN's algorithm works in the way that the complete set of calculations is finished and a datatype contains all data which has to be updated. Hence the optimization can start once this data is passed to the executing `DBConnector`

<sup>6</sup>The resulting file contains around 330,000 rows of the kind 'read\_key classification class'. A reasonable time consumption would be around 30 seconds.

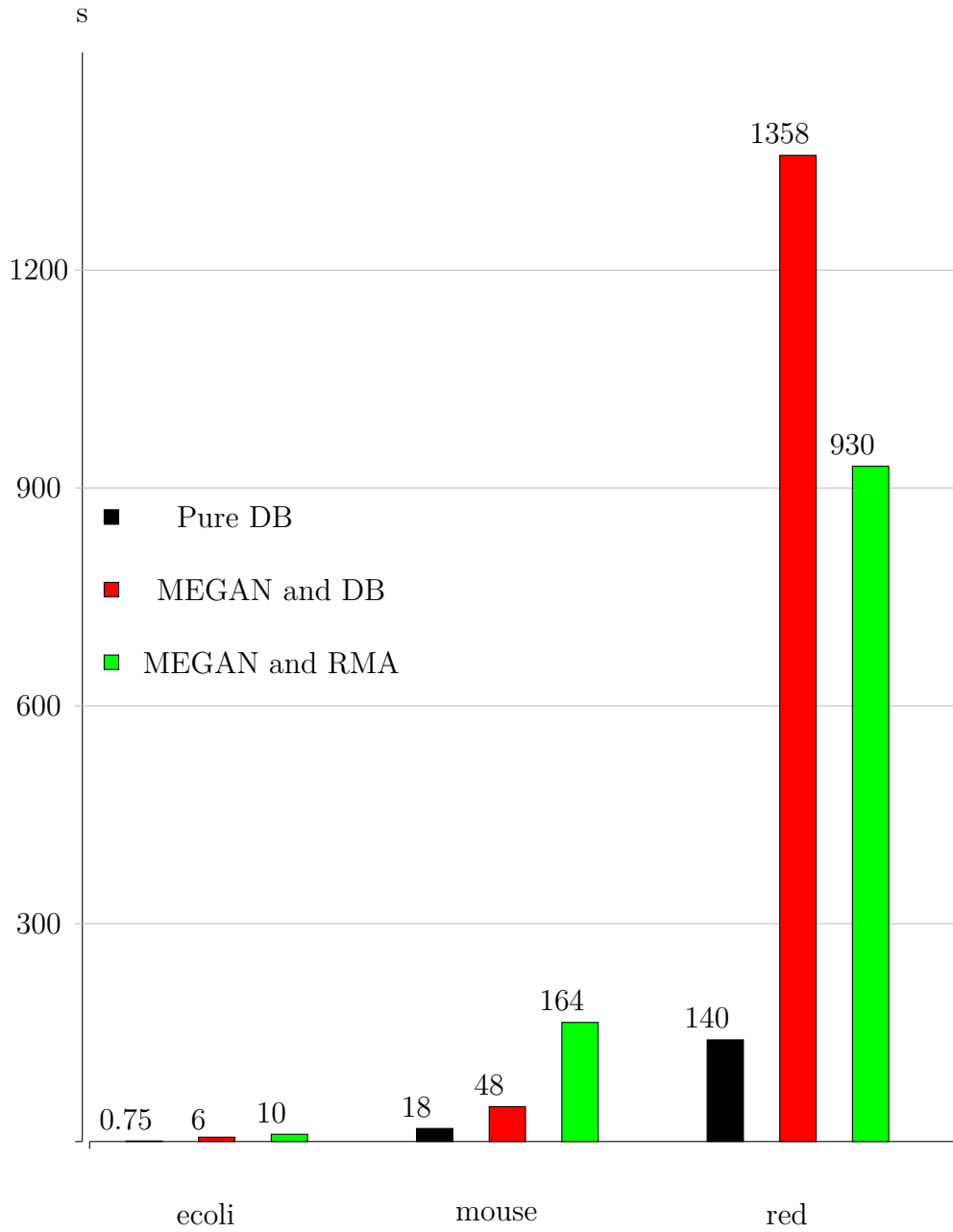


Figure 8.2: Update taxonomy classification via MEGAN - RMA, MEGAN - DB and solely DB. Parameters are minscore = 50, toppercnt = 10, minsupport = 1



## 9 Conclusion

The subject matter of this thesis was to introduce a new approach of storing MEGAN's datasets into a database. Additionally, this database was intended to provide functionality for outsourcing workload from MEGAN's to the database's scope and therefore avoid bottlenecks in calculation matters in order to save time and memory.

So far, a database has been developed which fulfills most of the initial requirements. In order to achieve the not yet implemented features of missing functionality, such as the insufficient compression of the BLAST field and to find a suitable solution to create new datasets, new ideas with promising performance gains were introduced aiming to bridge this gap.

In addition, when comparing the flexible characteristic of a database to the rather static behavior of an RMA file, new ideas come to mind. Future functions of MEGAN are not longer reduced to the possibilities of the RMA file, they can focus on dealing with the question 'How can I achieve this within the database?'. This includes not solely for example working on data of several datasets at the same time or possibilities to delete and insert reads or matches without costly recalculation, but also updating single classifications while still working on the same dataset. Additionally, with the large function library of PostgreSQL still being extended, new possibilities arise when it comes to calculation or parsing issues. Ideas not implemented due to the unreasonable runtime could now be implemented directly within the databases' scope.

Chapter 8 shows that both the calculation and the parsing beat the implementation of the same operations within MEGAN's scope easily - not only in time, but also in memory issues. One example of future options could due to the convincing runtimes achieved in the text search test be the 'database based creation of a dataset'. Instead of using MEGAN respectively Java to parse a large BLAST file, this workload can be outsourced to the database by using a suitable parsing function provided by PostgreSQL.

However, the results of this thesis should be seen as a stable adaption of a database in the load/store mode to MEGAN. The additional functions, such as calculation of the taxonomy classification, are implemented as 'ready-to-use' but are not integrated to be executed from MEGAN yet.

To put all together, this thesis proves that the use of databases enhances MEGAN's potential.

# List of Figures

2.1	The structure of a typical RMA file . . . . .	10
3.1	The initial screen of MEGAN after loading the <code>taxonid</code> tree . . . . .	16
3.2	Open dataset <i>ecoli</i> displaying the summary of the <code>taxonid</code> classification	17
3.3	Inspector on read (left) and match (right) level . . . . .	18
4.1	Structures of the Entity Relationship model . . . . .	23
4.2	Entity Relationship model for MEGAN's data . . . . .	25
5.1	Schema for the tables read and classification . . . . .	33
7.1	Rooted Tree with Pre and Postorder Ranks (label:pre,post) . . . . .	69
7.2	Tree Structure from Fig. 7.1 as a Table . . . . .	70
8.1	Copy catasets to RMA, to empty DB, to with <i>red</i> dataset loaded DB and to empty, non-indexed DB with index creation . . . . .	79
8.2	Update taxonomy classification via MEGAN - RMA, MEGAN - DB and solely DB. Parameters are <code>minscore = 50</code> , <code>toppercent = 10</code> , <code>minsupport</code> <code>= 1</code> . . . . .	84

# List of Tables

4.1	Table header . . . . .	29
4.2	Table read . . . . .	29
4.3	Table auxiliary . . . . .	29
4.4	Table match . . . . .	30
4.5	Table classread . . . . .	30
4.6	Table blasthit . . . . .	30
4.7	Table classificationblock . . . . .	30
5.1	Tables read and classification . . . . .	33
5.2	Combination I: Tables read and classification . . . . .	35
5.3	Combination II: Tables ead and classification . . . . .	35
8.1	Creation Times for Database Datasets . . . . .	80
8.2	Size requirements for storage in postgresQL and RMA in MB . . . . .	81
8.3	Regular expression seektime RMA vs. DB . . . . .	82

# Bibliography

- [AS90] ALTSCHUL SF, GISH W, MILLER W MYERS EW LIPMAN DJ: *Basic local alignment search tool*. J Mol Biol, 215(3):493–10, October 1990.
- [Cod70] CODD, E. F.: *A relational model of data for large shared data banks*. Commun. ACM, 13(6):377–387, 1970.
- [Cod90] CODD, E. F.: *The relational model for database management: version 2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.
- [DHH07] DANIEL H. HUSON, ALEXANDER F. AUCH, JI QI ET AL.: *MEGAN analysis of metagenomic data*. Genome Research, 17(3):377–386, March 2007.
- [HSA<sup>+</sup>08] HUSON, DANIEL H., STEPHAN C. SCHUSTER, CONTRIBUTIONS FROM ALEX, ER F. AUCH, DANIEL C. RICHTER, SUPARNA MITRA QI JI: *Contents User Manual for MEGAN V2beta9*, 2008.
- [pgb03] *PostgreSQL - Das offizielle Handbuch*. mitp-Verlag, 2003.
- [RR03] RAGHU RAMAKRISHNAN, JOHANNES GEHRKE: *Database Management Systems*. MCGRAW HILL, 2003.

## .1 SQL code to create MEGAN's tables

```
1 CREATE TABLE header (
2 file_key BIGINT,
3 file_name VARCHAR(200),
4 creation_date BIGINT,
5 modification_date BIGINT,
6 number_of_classes INTEGER,
7 number_of_reads INTEGER,
8 number_of_matches INTEGER,
9 PRIMARY KEY (file_key)
10 );
11
12 CREATE TABLE read(
13 file_key BIGINT REFERENCES header(file_key) ON DELETE CASCADE,
14 read_key BIGINT,
15 head VARCHAR(100),
16 seq TEXT,
17 number_of_matches INTEGER,
18 length INTEGER,
19 max_score REAL,
20 mate BIGINT,
21 PRIMARY KEY (read_key)
22 );
23
24 CREATE TABLE classread(
25 read_key BIGINT REFERENCES read(read_key) ON DELETE CASCADE,
26 taxonomy VARCHAR(100),
27 value INTEGER,
28 PRIMARY KEY (read_key, taxonomy)
29 );
30
31 CREATE TABLE match(
32 read_key BIGINT REFERENCES read(read_key) ON DELETE CASCADE,
33 match_key BIGINT,
34 bitScore REAL,
35 eValue REAL,
36 ref VARCHAR(100),
37 length INTEGER,
38 ignore SMALLINT,
39 taxonid INTEGER,
40 COG INTEGER,
41 PRIMARY KEY (match_key)
42 );
```

```
1 CREATE TABLE blasthit(  
2 match_key BIGINT REFERENCES match(match_key) ON DELETE CASCADE,  
3 text TEXT,  
4 PRIMARY KEY (match_key)  
5 );  
6  
7 CREATE TABLE aux(  
8 file_key BIGINT REFERENCES header(file_key) ON DELETE CASCADE,  
9 data SMALLINT  
10 );  
11  
12 CREATE TABLE classificationblock(  
13 file_key BIGINT REFERENCES header(file_key) ON DELETE CASCADE,  
14 taxonomy VARCHAR(100),  
15 value INTEGER,  
16 SUM INTEGER,  
17 PRIMARY KEY (file_key, taxonomy, value)  
18 );
```

## .2 PostgreSQL's function to assure a correct classificationblock table

```
1 CREATE OR REPLACE FUNCTION merge_classread(key BIGINT, tax VARCHAR(100)
2 , val INTEGER) RETURNS BOOLEAN AS
3 $$
4 DECLARE
5 fk BIGINT = (SELECT file_key FROM read WHERE read_key = key);
6 value_old INTEGER = (SELECT value FROM classread WHERE read_key = key
7 AND taxonomy = tax);
8 BEGIN
9     IF EXISTS(SELECT value FROM classread WHERE read_key = key AND
10 taxonomy = tax)
11 THEN
12     UPDATE classread SET value = val WHERE read_key = key AND
13 taxonomy = tax;
14 IF((SELECT sum FROM classificationBlock WHERE file_key = fk AND
15 taxonomy = tax AND value = value_old)= 1)
16 THEN
17     DELETE FROM classificationblock WHERE file_key = fk AND
18 taxonomy = tax AND value = value_old;
19 ELSE
20     UPDATE classificationblock SET sum = (sum - 1) WHERE file_key =
21 fk AND taxonomy = tax AND value = value_old;
22 END IF;
23 ELSE
24     INSERT INTO classread VALUES (key, tax, val);
25 END IF;
26
27 IF EXISTS(SELECT sum FROM classificationblock WHERE file_key = fk
28 AND taxonomy = tax AND value = val)
29 THEN
30     UPDATE classificationblock SET sum = (sum + 1) WHERE file_key =
31 fk AND taxonomy = tax AND value = val;
32 ELSE
33     INSERT INTO classificationblock VALUES (fk, tax, val, 1);
34 END IF;
35
36 RETURN false;
37
38 END;
39 $$
40 LANGUAGE plpgsql;
```

### .3 Plpgsql implementation of pre- and postorder ranks for table node

```
1 CREATE OR REPLACE FUNCTION pre(int, int) returns int AS '  
2 DECLARE  
3     child integer;  
4     preval2 int;  
5 BEGIN  
6     UPDATE node SET pre = $1 WHERE id = $2;  
7     preval2 = $1 +1;  
8     FOR child IN SELECT id FROM node WHERE parentid = $2 LOOP  
9         preval2 = pre(preval2,child);  
10    END LOOP;  
11    RETURN preval2;  
12 END;  
13 ' LANGUAGE plpgsql;  
14  
15  
16 CREATE OR REPLACE FUNCTION post(int, int) returns int AS '  
17 DECLARE  
18     child integer;  
19     postval2 int;  
20 BEGIN  
21     postval2 = $1;  
22     FOR child IN SELECT id FROM node WHERE parentid = $2 LOOP  
23         postval2 = post(postval2,child);  
24     END LOOP;  
25     UPDATE node SET post = postval2 WHERE id = $2;  
26     RETURN postval2 +1;  
27 END;  
28 ' LANGUAGE plpgsql;
```