Installation of Samtools

- Download the latest version of Samtools from (link).
- Create a folder that will contain your Samtools installation. We will refer to this folder as <samtools>.
- Unzip the main directory of the download into the folder <samtools>, thereby creating the folder <samtools>/samtools-x.x, where x.x denotes the version number of Samtools.
- Create two symbolic links that point to <samtools>/samtools-x.x: <samtools>/include and <samtools>/lib
- Now open the makefile that is shipped with Samtools (<samtools>/samtools-x.x/Makefile) with your favorite editor and add the flag -fPIC to the CFLAGS variable. This allows MonetDB to use the Samtools libraries during the installation of MonetDB.
- Now you can 'cd' into the <samtools>/samtools-x.x directory and run the make command, without any arguments.

Installation of MonetDB with the BAM library

If you want to install MonetDB with the BAM library, the installation has to know where it can find Samtools. This is achieved by specifying the additional option --with-samtools=<samtools> to the configuration step, where <samtools> is the path to the Samtools directory as described in the 'Installation of Samtools' section.

BAM library

The BAM library contains the following features to enable working with BAM files:

- Loading feature that enables loading a single BAM file, a user specified selection of BAM files or all BAM files in a user specified directory into the database.
- Removal feature that enables removing all data from a BAM file from the database.
- A BAM SQL library, that contains some handy SQL functions for doing analysis on BAM data that is loaded into the database.
- A SAM formatter, that renders a database result set into the SAM format.

These features are addressed in the following sections. All of these features follow the same database design. This database design defines tables that can be used to store the information contained in BAM files. First of all, the database design contains a 'files' table that stores which BAM files are loaded into the database. Every file tuple in this table contains a unique file id (handed out by the BAM loader during the loading process), the location of the BAM file when it was loaded, the value of the dbschema argument (explained in the next paragraph) and the information that is contained within the HD tag of the BAM file header. The database design defines additional tables to store the remainder of the BAM file header, where a really straightforward approach was used (e.g. table 'sq' for SQ and table 'rg' for RG header records). The definition of the files table and the other header tables can be seen in the next figure.

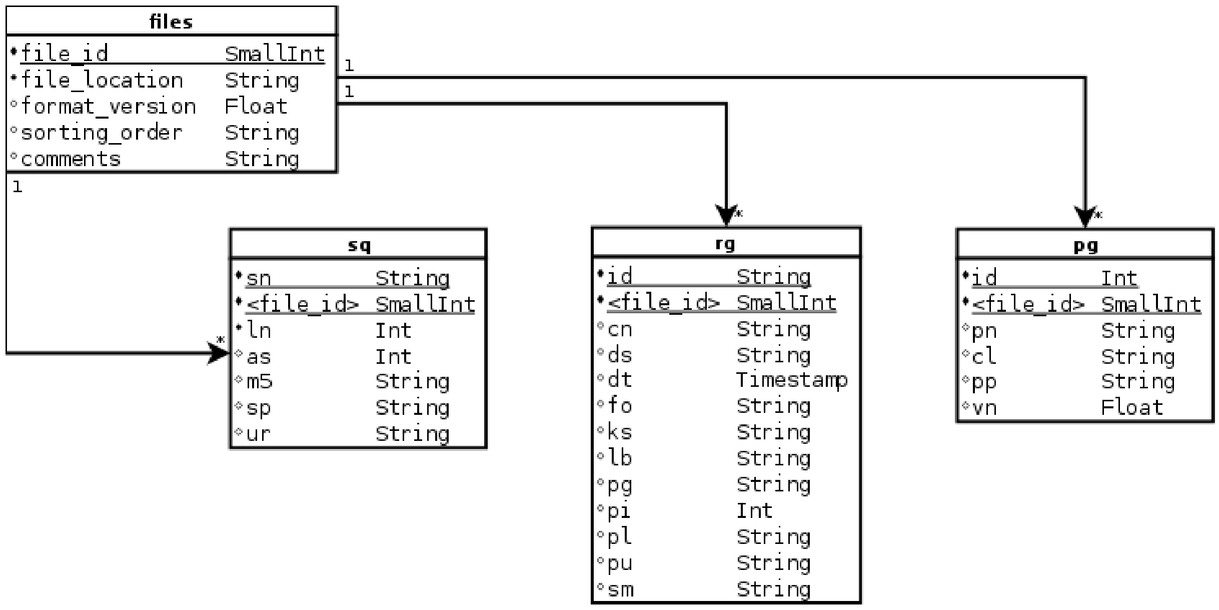


Figure 1: *Database tables that store header information*

The BAM database design defines a separate set of database tables for every BAM file that is loaded into the database. This design choice was made to speed up analyses on only one or two BAM files, since this often occurs in practice. This speed up is realized by not having to filter out alignment data of specific BAM files as a first analysis step.

When loading BAM alignment data into the database, you can choose to store these data in one of two sets of predefined database tables, referenced to as the straightforward table set and the pairwise table set. These table sets are presented in the next figures, for a BAM file with file id 'i'.

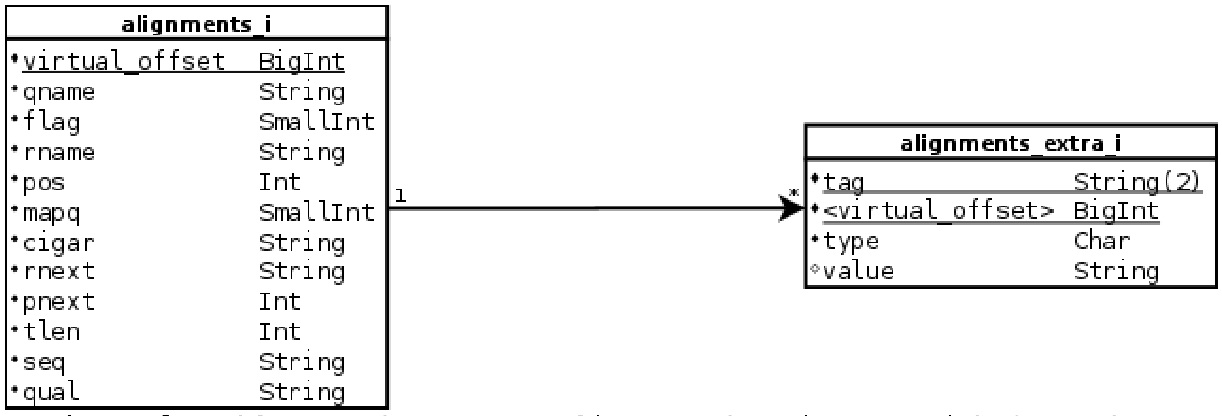
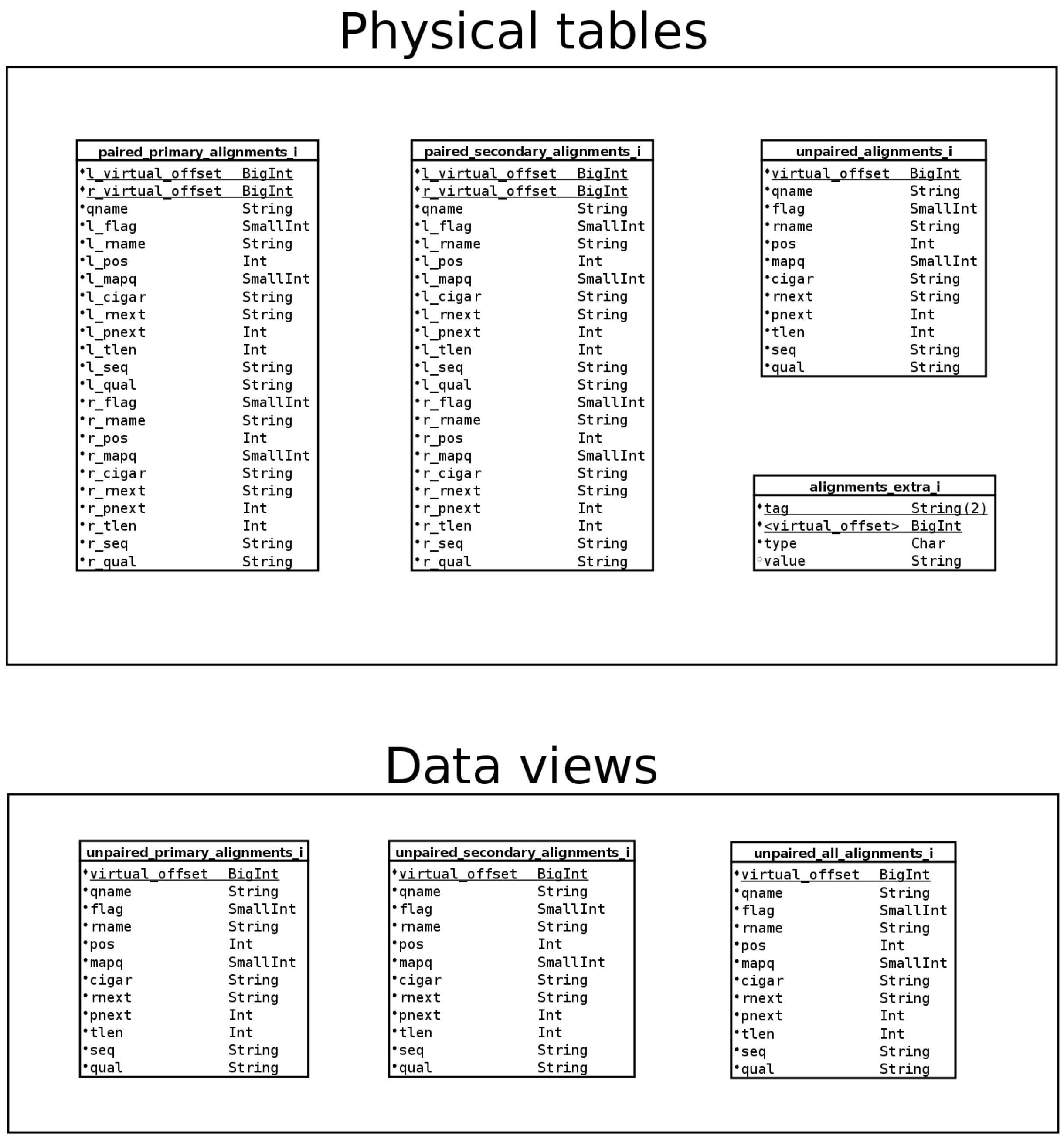


Figure 2: *Table set that stores alignment data in a straightforward way*



Physical tables

paired_primary_alignments_i	
•l_virtual_offset	BigInt
•r_virtual_offset	BigInt
•qname	String
•l_flag	SmallInt
•l_rname	String
•l_pos	Int
•l_mapq	SmallInt
•l_cigar	String
•l_rnext	String
•l_pnext	Int
•l_tlen	Int
•l_seq	String
•l_qual	String
•r_flag	SmallInt
•r_rname	String
•r_pos	Int
•r_mapq	SmallInt
•r_cigar	String
•r_rnext	String
•r_pnext	Int
•r_tlen	Int
•r_seq	String
•r_qual	String

paired_secondary_alignments_i	
•l_virtual_offset	BigInt
•r_virtual_offset	BigInt
•qname	String
•l_flag	SmallInt
•l_rname	String
•l_pos	Int
•l_mapq	SmallInt
•l_cigar	String
•l_rnext	String
•l_pnext	Int
•l_tlen	Int
•l_seq	String
•l_qual	String
•r_flag	SmallInt
•r_rname	String
•r_pos	Int
•r_mapq	SmallInt
•r_cigar	String
•r_rnext	String
•r_pnext	Int
•r_tlen	Int
•r_seq	String
•r_qual	String

unpaired_alignments_i	
•virtual_offset	BigInt
•qname	String
•flag	SmallInt
•rname	String
•pos	Int
•mapq	SmallInt
•cigar	String
•rnext	String
•pnext	Int
•tlen	Int
•seq	String
•qual	String

alignments_extra_i	
•tag	String(2)
•<virtual_offset>	BigInt
•type	Char
◦value	String

Data views

unpaired_primary_alignments_i	
•virtual_offset	BigInt
•qname	String
•flag	SmallInt
•rname	String
•pos	Int
•mapq	SmallInt
•cigar	String
•rnext	String
•pnext	Int
•tlen	Int
•seq	String
•qual	String

unpaired_secondary_alignments_i	
•virtual_offset	BigInt
•qname	String
•flag	SmallInt
•rname	String
•pos	Int
•mapq	SmallInt
•cigar	String
•rnext	String
•pnext	Int
•tlen	Int
•seq	String
•qual	String

unpaired_all_alignments_i	
•virtual_offset	BigInt
•qname	String
•flag	SmallInt
•rname	String
•pos	Int
•mapq	SmallInt
•cigar	String
•rnext	String
•pnext	Int
•tlen	Int
•seq	String
•qual	String

Figure 3: *Table set that stores alignment data in a paired way*

The database design uses the virtual offset of every alignment record as a primary key. This way, specific alignments can easily be found back in the original BAM file using e.g. Samtools.

The straightforward table set is aimed towards a straightforward mapping from the alignment data as it occurs in a BAM file. Every alignment field gets stored in a database column. Furthermore, the extra information that is contained in alignment data is parsed and stored in a separate table (alignments_extra_i).

The pairwise table set aims at reducing the performance overhead when many operations are done on paired alignment data. During the loading process, both primary and secondary alignment pairs are automatically recognized and stored accordingly in the appropriate database tables. The columns in the paired tables have either an 'l' or an 'r' prefix (except for the 'qname', since two alignment records in the same pair always have the same qname). Columns with an 'l' prefix store data from alignment records that have their 'first segment' flag set to 1 and columns with an 'r' prefix do this for alignments with their 'last segment' flag set to 1. All alignments that can not be paired are stored in the unpaired alignments table, which has the same structure as the regular alignments table in the straightforward table set. In addition to physical tables, the pairwise table set also defines some views over the data, which are also automatically created. These views offer ways to access the data is if it was stored in an unpaired fashion. E.g., any query that is aimed towards the alignments table from the straightforward table set can be fired at the unpaired_all_alignments view.

Note: currently, if you want to load a BAM file into the pairwise storage schema, the BAM file has to be sorted on queryname. If this is not the case yet, please sort it first using e.g. Samtools before trying to load it into the database.

Loading BAM files into the database

After succesful installation of the BAM library, loading BAM files into the database is really simple. You do not have to worry about creating the appropriate database schemas or tables, as all of this is taken care of by the BAM loader. When given a load command, the BAM loader executes the following routine:

- It checks whether or not the database schema 'bam' already exists, since all data that will be stored by the BAM loader will be stored in this schema. If it does not exist, it is automatically created.
- It checks for every header table (Figure 1) whether or not this table exists already. If it does not exist, it is automatically created.
- For every BAM file to be loaded:
    - A unique file id 'k' is calculated for this BAM file
    - The appropriate table set is created, with all i suffixes replaced by 'k'
    - Load all header data and alignment data into binary files
    - Use the MonetDB native COPY INTO statement to transfer all binary files into the database tables

The following SQL functions are available to load BAM files into the database:

bam_loader_file(file_location, dbschema)

Loads a single BAM file into the database.

file_location: Absolute path to the file

dbschema: 0 if you want to load the data from the BAM file into the straightforward table set
          1 if you want to load the data from the BAM file into the pairwise table set

bam_loader_files(list_location, dbschema, nr_threads)

Loads a list of BAM files into the database.

list_location: Absolute path to a text file containing the paths to all the BAM files that have to be loaded, separated by a newline character

dbschema: 0 if you want to load the data from the BAM file into the straightforward table set
          1 if you want to load the data from the BAM file into the pairwise table set

nr_threads: Enables you to specify how many threads may be used to write the data from the BAM files to binary files

bam_loader_repos(repos_location, dbschema, nr_threads)

Loads all BAM files in a directory (non-recurisve) into the database.

repos_location: Absolute path to a directory containing BAM files.

dbschema: 0 if you want to load the data from the BAM file into the straightforward table set
          1 if you want to load the data from the BAM file into the pairwise table set

nr_threads: Enables you to specify how many threads may be used to write the data from the BAM files to binary files

For example, loading a single BAM file /path/to/bam/example.bam into the straightforward table set could be done from the SQL context by executing the following SQL:
```
CALL bam_loader_file('/path/to/bam/example.bam', 0);
```

After loading BAM files into the database, their data will be stored in a table set that will be created during the loading process. You will not know beforehand what the file id of the loaded files will be and therefore, you do not know beforehand what tables will be created. Therefore, you can consult the files table to see what file id was assigned to your file:
```
SELECT * FROM bam.files;
```

Result:
```
+---------+-------------------------+----------+----------------+---------------+----------+
| file_id | file_location           | dbschema | format_version | sorting_order | comments |
+=========+=========================+==========+================+===============+==========+
|     33  | /path/to/bam/example.bam |      0   | 1.0            | unsorted      | null     |
+---------+------------------------+---------+--------------+-------------+---------+
```

In this case, the alignment data will be stored in the tables bam.alignments_33 and bam.alignments_extra_33.

Removing a BAM file from the database

The BAM library enables you to remove all traces from a BAM file from the database, by providing you with the following SQL function:

bam_drop_file(file_id, dbschema)
Removes all entries of a BAM file in the header tables and removes the table set that was created for this BAM file.

file_id: The file id from the BAM file that has to be removed (can be found in the bam.files table)

```
dbschema: 0 if data was loaded into a straightforward table set
          1 if data was loaded into a pairwise table set
```

Using the BAM SQL library

There are some operations that are often done on BAM data that are hard, or even impossible, to express using SQL. Therefore, we added some SQL functions that hopefully make your life easier.

bam_flag(flag, name)

Returns a boolean value, indicating whether or not the bit with the given name was set to 1 in the given flag.

flag: Any integer value

name: The name of the bit that has to be checked. See Table 1 for the names that we have given to the several bit positions.

Bit	Name
0x1	mult_segm
0x2	prop_alig
0x4	segm_unma
0x8	next_unma
0x10	segm_reve
0x20	next_reve
0x40	firs_segm
0x80	last_segm
0x100	seco_alig
0x200	qual_cont
0x400	opti_dupl
0x800	supp_alig

Table 1: *Mapping of bit position to names*

For example, if we want to select all primary alignment records from table bam.alignments_1, we could use the **bam_flag** function to construct an easy to read SQL query:

```
SELECT *
FROM bam.alignments_1
WHERE bam_flag(flag, 'seco_alig') = False;
```

reverse_seq(seq)

Compute the reverse complement of a sequence string.

seq: Sequence string containing only the characters A,T,C,G,R,Y,S,W,K,M,H,D,V,B,N

reverse_qual(qual)
Compute the reverse quality string.

qual: Quality string.

seq_length(cigar)

Use the cigar string of an alignment to calculate the actual length of its sequence string (the length of the area of the reference string the alignment got mapped to).

cigar: Cigar string

Rendering query results to a SAM format

If you are using mclient to communicate with MonetDB, you can use the built-in SAM formatter to export query results to a SAM format. In order for this to work, the output columns must have the appropriate names, i.e. the renderer looks for the column names as defined by the alignments_i table defined in Figure 1. If there are columns that cannot be mapped, you will receive a notification about this and the renderer will simply not do anything with the data in these columns.

First of all, select the SAM formatter in mclient by typing:
```
\f sam
```

Now, if you execute a query within mclient, the result will be written to the selected output channel in SAM format. In case you want to write the output to a SAM file, you can type the following command in mclient:
```
\>/path/to/your/new/samfile.sam
```

After that command, query results will be written to your SAM file. In case you want to generate multiple SAM files, don’t forget to use the \> command every time, to steer your output in the right direction. When you do not do this, query results will be simply appended to the already selected file.