Benchmark Analysis Tool

Richard Cornelissen, Rens van der Heijden, Matthijs Hofstra,
Edwin Keijl, Ties de Kock, Remco Swenker

July 13, 2010

Contents

Chapter 1

Problem Statement

The Formal Methods and Tools research group has developed a toolset, called Minimization and Instantiation of Labelled Transition Systems (ltsmin), which can be used to analyze labeled transition systems through various tools. Ltsmin also supports this analysis on a cluster of machines, which is managed by a PBS scheduler. The toolkit produces a lot of data, for which no analysis tool is available. This is the problem that BeAT attempts to solve.

The data includes run times, the time the tool was started, the amount of memory it required (both virtual and Resident Set Size (RSS)), whether the tool successfully completed its task, as well as the number of states and the amount of transitions [1]. Other data that BeAT collects is the cluster nodes and provided options.

Currently, without BeAT, all this data is manually inserted into a database. This process has been automated. BeAT should also be future proof, so that new tools and modifications to the other tools and their output can be made without modifying the BeAT code.

To provide interaction with the user, BeAT will have a web interface. It has been decided at the start of the project that the software would be written using Django, which is a web framework for Python. Javascript is used for client-side manipulation of the data, to allow the use of dynamic webpages. The interface allows the user to examine the data at will in table form, as well as allowing the generation of two types of graphs. These graphs are generated by the python module MatPlotLib, which also provides the necessary export capabilities.

[1] There is an exception for the amount of transitions: it is not recorded in some space exploration tools.

Chapter 2

Requirements

The requirements have been made using the description given in chapter one. they have been divided into functional requirements and non functional requirements.

	Functional Requirements
R1	The system will store the output that is produced in a database. This should include benchmark names, their results and similar information.
R2	The user interface will support filtering of the results stored in the database.
R3	The system will allow the user to examine the results in the form of a table.
R4	The system will allow the user to examine the results in the form of a chart.
R5	The system will allow the user to export data, tables and charts to a form that allows the user to easily incorporate this in a scientific paper.
R6	The system will allow the user to export data, tables and charts to a form that is machine-readable (preferably CSV).
R7	The system will read and store results from the cluster as soon as possible after it completes a batch job.
R8	The system will support user management (validation, authorization...).
R9	The system will offer full support of the input format that LTSmin uses.
R10	The system will have at least basic support for the input format of other benchmark toolkits.
R11	The system will have a build environment in which it will shape input into a batch job that can be provided to the PBS.
R12	The system will support personal results.
R13	The system will allow the user to provided generated results to the public (including users without accounts).

	Non Functional Requirements
R13	The system will be written in a dynamic programming language using a maintained, stable web framework.
R14	The system will have a user-interface in the form of a website (web interface).
R15	The system must be easy to deploy and be portable, preferably in a virtual machine.
R16	The system shall respond to 90% of the http requests within 250 ms.
R17	The system shall work with Firefox 3.5+, Chrome 4.0+, Safari 4+.
R18	The system shall conform to web standards (HTML 4+, CSS).
R19	The system shall be protected against SQL Injection and XSS attacks.

Chapter 3

Design Decisions

In this chapter important decisions we made during the development process will be discussed. Most of the sections here are also discussed in more detail in chapter 4, where implementation issues are discussed.

3.1 Database

3.1.1 Initial Design

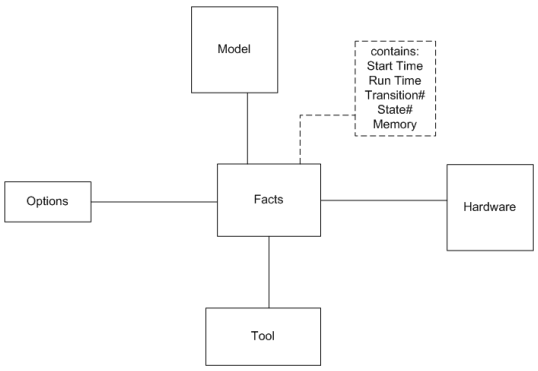

Figure 3.1: The initial, higher level database design, based on the data warehouse concept.

Our initial idea was to use a data warehouse, a type of database that allows easy aggregation and filtering. For our database, the fact (the data warehousing term for the value(s) on which computation occurs) is a few fields in the Benchmark table. This idea, illustrated in figure 3.1, still exists within the database design, but it has been altered to suit our needs. The data warehouse approach focuses too much on a single fact, while our database contains several pieces of

data for each combination of variables (eg, model, tool and so on). This means we cannot use SQL queries that were designed for a data warehouse; however, these queries are not available in django, so they would force us to use custom queries, which breaks django's cross-database nature.

3.1.2 Current Design

Below, the tables in the current database design are outlined and the relations between tables are explained. The design itself is outlined in figure 3.2, excluding those tables that are not related to the BeAT core system (these tables relate to the functionality of the web interface). Please note that certain terms used in this diagram and in the text below are different from those used within ltsmin. See the glossary (appendix A) for more information.

A row in the Benchmark table corresponds to a single benchmark result. It is connected with Model and AlgorithmTool, which together with relations to the Hardware and OptionValue tables (see below) specify the environment in which the benchmark was executed. Most of the columns are the statistics that are at the base of BeAT (time the benchmark was started, execution times, memory values, exit status and the amount of states/transitions). The `logfile` column contains a URI to the original log file, which is recorded for further examination by the user. The BenchmarkHardware table represents a ManyToMany relation between Benchmark and Hardware.

The Model, Algorithm and Tool tables simply represent what their names imply, as defined in the glossary (see appendix A). An AlgorithmTool row represents a combination of an Algorithm and a Tool, together with a version identifier, date of the git revision and a regular expression (a row in the Regex table) that can be used to parse logs produced by this particular AlgorithmTool. A row in the Regex table represents a regular expression, which is currently used to parse a log file. Since options may provide additional output of interest, a ValidOption row (see below) may also point to a Regex.

Options are handled through the following tables:

Table	**explanation**
Option	names of options.
ValidOption	connection between option and AlgorithmTool, also contains a connection to regex.
RegisteredShortcut	representation of a short option, relates to an AlgorithmTool and an option.
OptionValue	connects an option with an assigned value.

The Option table is used to reduce the size of the database, storing the name of an option only once (similar to Model, Algorithm and Tool). ValidOption is used to indicate an option is valid for a certain AlgorithmTool. It also links to a regular expression, which may be used to parse additional information from a log file. RegisteredShortcut is used to allow short options to relate to different Options for different AlgorithmTools. For example, in a certain AlgorithmTool, `-c` could be short for `--cache`, in another it may be short

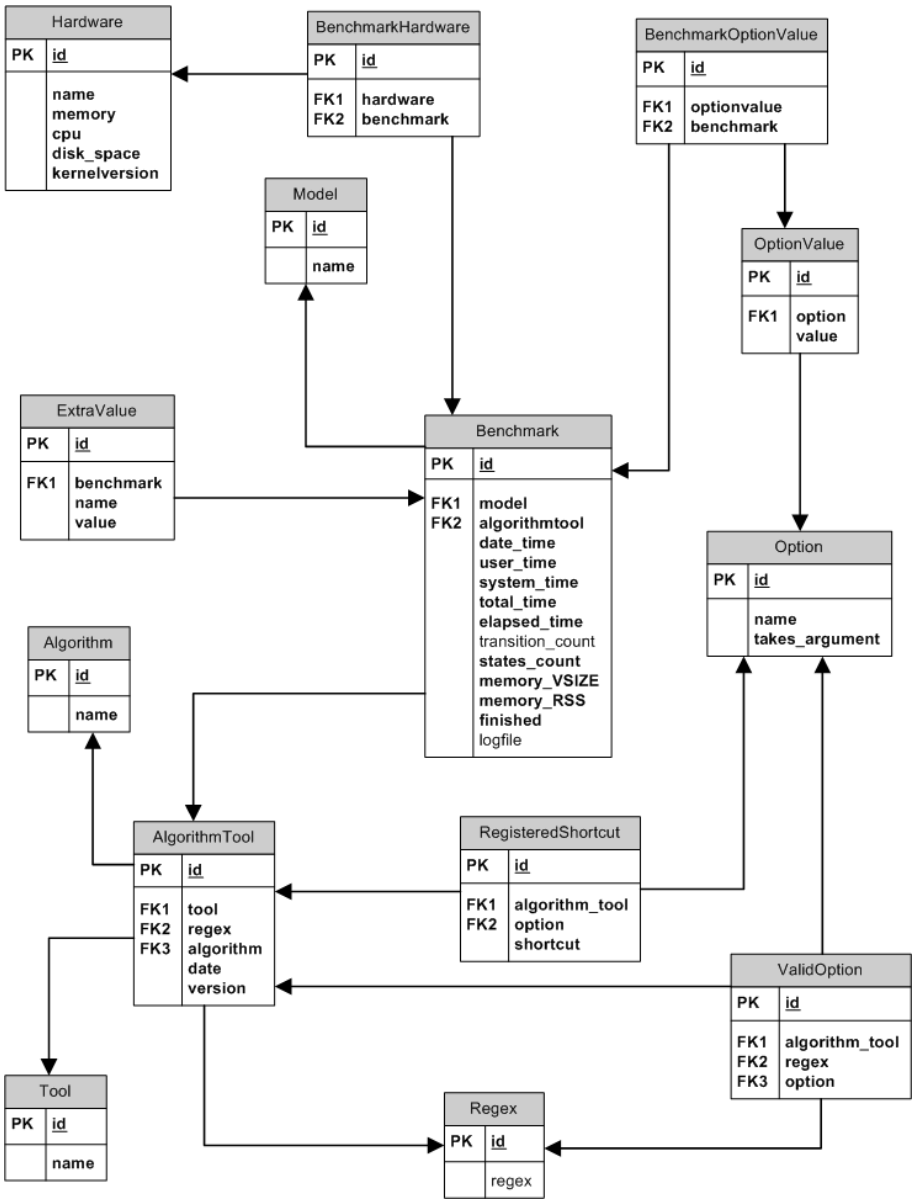

Figure 3.2: Our current database design. Excluded here are tables that are part of the other BeAT modules.

for `--confidence=`. While this is desirable, the said structure introduces a problem, because it always relates a RegisteredShortcut to an Option, while this is not strictly true in reality. To solve this, Option objects whose name starts with a space character `(' ')` are ignored as long options. This works because spaces can never be part of an option passed in the command line. OptionValue objects are used to indicate a certain option was set (value = `'True'`), or was passed with some argument (value=value_of_argument). A set of zero or more OptionValue objects is connected to each Benchmark. For example, `--cache`, `--state=vset` and `--vset=tree` could be provided to execute a Benchmark, resulting in three OptionValue objects (represented here as tuples of name and value): (`--cache`,`'True'`), (`--state=`,`'vset'`) and (`--vset`,`'tree'`).

The ExtraValue table is used to associate additional information with a Benchmark. The current scheme is quite fragile, since an ExtraValue with the same name does not always indicate the same value, but it is a simple and fairly flexible approach. The names are derived from the user-provided regular expressions, while the values are whatever the regular (sub-)expression matches on. To be more exact, python regular expressions provide a way of naming a group as follows: ...`(?P<group_name> expression)`..., which creates an ExtraValue row with `name=group_name` and `value=match`, where match is that which expression matches to. For more information about the parsing process, see sections 3.2.2 and 4.2.2.

3.2 Database Population

The BeAT database needs to be filled with data in some way. Log files generated by running ltsmin need to be parsed and placed into a database. However, since the output format of ltsmin is not static, the use of regular expressions was proposed. To allow the user to configure different ways of parsing the logs for each version of ltsmin, the database contains a Regex table, in which the regular expressions are saved. These are added through the web interface; see section 3.2.3 for more information.

Routine population of the database (ie. importing data from log files) consists of two steps:

1. Moving logs to the server
2. Parsing the logs and inserting them into the database

These two steps are described below, followed by the way tools are added to the database. Parsing logs includes parsing a header, which is automatically generated for jobs that are created through our web-interface. For documentation on this interface, see section 3.6. For documentation of the header, see appendix B.

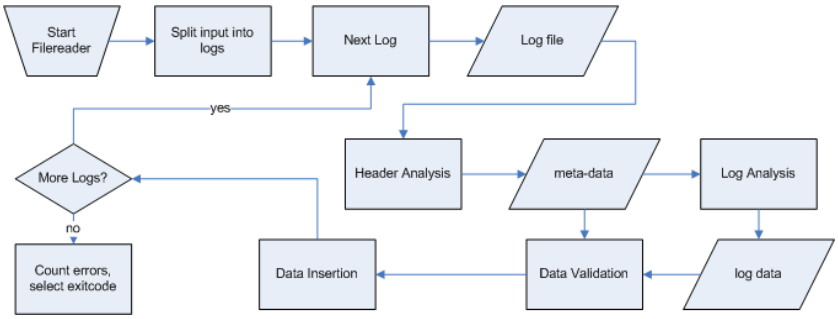

Figure 3.3: High level flowchart for filereader script.

3.2.1 Moving Logs

Initially, the way logs were pushed to the server that BeAT runs on was unclear. We considered three options: automatic movement, user uploaded (through a form on the website) and user-controlled copying (where the user moves the files with a tool like scp). Automatic movement incurs a relatively large security risk, while its advantage over the others is limited. We assumed both of the other options would be implemented.

Later, it became clear that an upload mechanism is not an optimal solution, since the amount and total size of a set of logs is usually too great to submit through a website. After discussing with our client, we assumed the intended user was familiar with unix tools such as `scp`. In the current implemenation, the user uploads a set of logs to a specific file on the BeAT server, which is then automatically inserted (using the process described below). The user can also manually execute the insertion script.

3.2.2 Parsing & Insertion

Because of the scheme we use for reading data from logs, and the way we allow different tools to be used, we assume the regular expressions and the information about the tools are already in the database. As mentioned, we have created a page to do this; see section 3.2.3 for more information.

For historical reasons, we allow several logs to be placed in one file. This is also useful when inserting a large amount of logs without access to bash. The process is illustrated in the flowchart in figure 3.3. Available options for the filereader are documented in appendix B. For each log, the analysis process consists of two parts; of the header and of the log itself. The header is used to find the appropriate AlgorithmTool, which in turn references the regular expression to analyze the log. The provided options are also included in the header; the regular expressions associated with those are queried in a similar manner. Other information (not required for the log analysis) is also retrieved from the header; see the full description of the header in appendix B. The

regular expressions are sequentially applied to the log file; the one related to the AlgorithmTool is first and is required to succeed. The others are applied in the order they are fetched from the database; no guarantees are made about this order. Should certain group names overlap, the result from the regular expression applied last is used. It is up to the user to ensure this does not happen, which can be achieved by using sensible names.

After successfully collecting all this information, it is passed through a checking function (which can be extended in later versions of BeAT to allow for more specific checks). If this check succeeds, the data is passed to the submission function that inserts the data. This function gets or creates database items as needed (for example, the hardware item might already be in the database) and returns the benchmark object as well as its created status, allowing the main script to create the appropriate message for the user. The log file is saved as a plain text file, either on the file system or in a git using the python module dulwich (if the `--dulwich` switch is provided). The advantage of the latter is that this allows BeAT to take advantage of git's superior compression methods, while the disadvantage is the user cannot directly access the files (though the repository can be used as a normal git repository). See section 4.7 for more information on dulwich.

3.2.3 Tool Uploader

The tool uploader will allow the following data to be sent to the database:

Tool Name: The name of the tool for example: "lps"

Algorithm Name: The name of the back-end algorithm for example: "-reach"

Version: The version output of the new tool for example: "ltsmin-1.5-20-g6d5d0c". This version number is used to link the date when the tool was made in git to the uploaded logs. This makes it possible to plot performances of tools over time.

Regular Expression: The regular expression that should be used to parse output generated by this tool. Here is an example of a regex:

```
lps-reach: .*(\r\n|\n)((?P<kill>Killed|.*?error:.*)|lps-reach:
reachability took.*(\r\n|\n)state space has (?P<scount>\d+)
states.*; (?P<peakNodes>\d+) peak nodes.*(\r\n|\n)Exit) \[[0-9]+\]
(\r\n|\n)(?P<utime>[0-9.]+) user, (?P<stime>[0-9.]+) system,
(?P<etime>[0-9.]+) elapsed --( Max | )VSize = (?P<vsize>\d+)KB,
( Max | )RSS = (?P<rss>\d+)KB
```

to make sure the regex works there is a field to place a test log and a button to run the regex over the log. This allows you to see if the regex works as its supposed to.

Options: this will allow the addition of tool options. Short options (eg. "-c" for "–cache") can be entered as follows: "–cache:c" or ":x". If the option takes an argument, an "=" sign should precede the colon. Some of these options may also cause a change in the way the log should be parsed these extra regex rules.

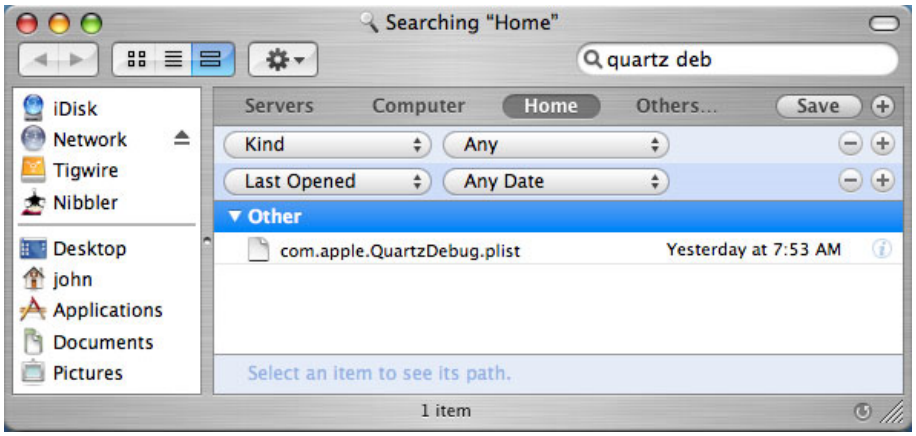

Figure 3.4: Mac OS X Finder, taken from [18]

.

3.3 Benchmark Table

The benchmark table was designed so that the user can view benchmarks, filter the view on certain properties, sort the view, and select which data is shown for every benchmark. It is created with the use of JavaScript, AJAX, jQuery [7], JSON [2], and some jQuery plugins to form an intuitive user interface, like hoverIntent [1].

3.3.1 Initial Design

Initially, we planned to build the benchmark table so that it looks like the search feature of Mac OS Xs Finder. With this application, you can add and remove certain filters (like last opened date and file type) to view the filtered results, as in figure 3.4. The idea of our benchmark table is similar, using the data in our database. For the benchmark table, we want the user to be able to add and remove filters, specify the type of the filter (e.g. a date filter), specify a filtering style (e.g. before the given date), and specify a filtering value.

The filter types we originally planned to implement are:

- Model name;
- Algorithm name;
- Tool name;
- Memory (RSS);
- Runtime (= user time + system time);
- Number of states;

- Number of transitions;
- Date;
- Options.

In these filter types, we recognized certain generalizations, namely:

- Selecting a name (model name, algorithm name and tool name);
- Specifying an integer and whether the value in the database is less than, greater than or equal to the provided value (memory (RSS), runtime, number of states, number of transitions);
- Specifying a date and whether the benchmark was executed on, before or after the specified date (date);
- Selecting certain options from a list and, if necessary, specify a value for the chosen option (options).

Furthermore, we want the user to be able to select some benchmarks from the table and compare them by clicking a compare-button.

3.3.2 Detailed Design

During the course of the project, some features of the benchmark table have changed, some features were added, and some were discarded. These decisions are explained here.

Change of features during development

During our meetings, we made the decision to add some extra features to the benchmark table, namely:

- Selecting which columns are displayed;
- Sorting on (multiple) columns;
- Adding filters for:
  - Version of algorithmtool;
  - Memory (VSIZE);
  - Finished;
  - Processor;
  - Physical Memory size;
  - Computer name;
  - Kernel version;
  - Disk space.

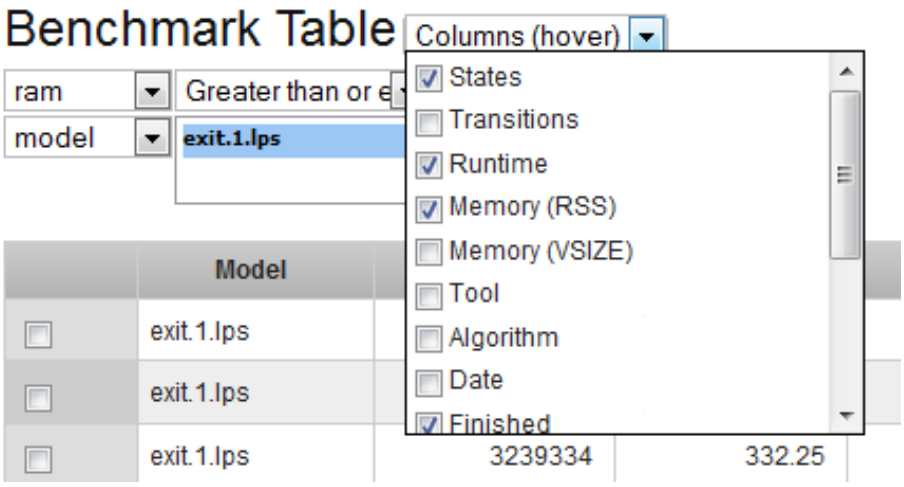

Figure 3.5: Screenshot of column selection

- Paging the results;
- Selecting all benchmarks and inverting the selection;
- Exporting selected benchmarks as a CSV-file;
- Filtering the queryset on the selected benchmarks;
- Instead of selecting one value for the filters model name, algorithm name, tool name, and later on, algorithmtool version, processor, computer name, and kernel version, it should be possible to select a list of values.

Furthermore, the feature to select and compare benchmarks was removed. We did this because we could not find a suitable interpretation for any combination of selected benchmarks.

Column selection

The user is able to select the columns of data he wants to see, this is done by checking and unchecking in a list of column names that pops up when he hovers his mouse over the input-element called "Columns (hover)", as seen in Figure 3.5. There are columns that are available for every benchmark; the standard columns, and columns that are not available for every benchmark run, and thus can only be selected when there is at least one benchmark run in the QuerySet the user is viewing that contains that value; the extra columns.

Standard columns

The columns which the user should be able to select by default are:

- Model name;
- States;
- Transitions;
- Runtime;
- Memory (RSS);
- Memory (VSIZE);
- Tool name;
- Algorithm name;
- Version;
- Date;
- Finished;
- Computername;
- Processor;
- Amount of physical memory;
- Kernel version;
- Amount of hard disk space;
- Options used.

By default, the values of model name, number of states, runtime, memory (RSS), and finished are shown in the benchmark table.

Extra values

The user should also be able to select which extra values he wants to see. The extra values are not in the database by default, but might be found in some log files because of certain options specified or certain tool and algorithm combinations used. Only the available extra value names that are present in the QuerySet the user is currently viewing are shown in the column list.

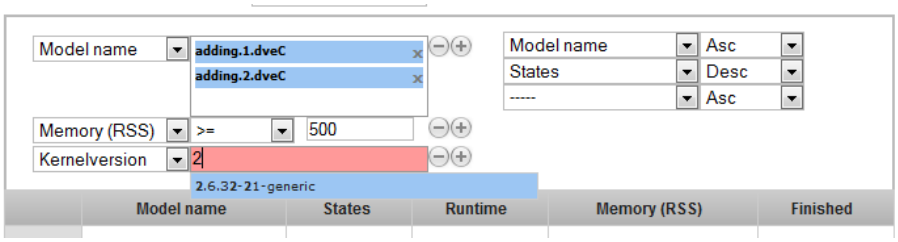

Figure 3.6: Screenshot of filter table and sorting

Sorting

The user is able to sort the QuerySet he is viewing by one or more of the columns he has selected. To do so, a table has been added above the benchmark table, where the user can specify a column and an order (ascending or descending), as seen in Figure 3.6. After specifying a column to sort on, an extra row in the table will appear so the user can specify the second column he wishes to sort on. Notice that the user can select the same column multiple times, though the database will only evaluate the first appearance of a column. The user can also sort the QuerySet on an extra value column, bat as these values are stored as text in the database, the sorting will be done alphabetical.

Filter table

The part of the benchmark table that regulates the filtering system is called the filter table. This table consists of a list of filters the user specified, as in Figure 3.6. On the right side of each filter are an add button and a remove button. The remove button deletes the filter in that row (or sets it to an empty filter if it is the only filter in the filtertable) and the add button adds an empty filter one row under the row of the button. Furthermore, when a filter contains an error (e.g. no value has been selected for a states-greater than filter), it is marked red, so the user can easily fix it.

Empty filter

The empty filter is a filter that does nothing, its only purpose is to give the user the option to select a filter type.

Model name, algorithm name, tool name, processor, computer name, algorithmtool version, and kernel version filters

The model name, algorithm name, tool name, processor, computer name, algorithmtool version, and kernel version filters all have the requirement that the user should be able to select one or multiple values. To achieve this, we were inspired by the way users can select to whom a message is sent in Facebook, like

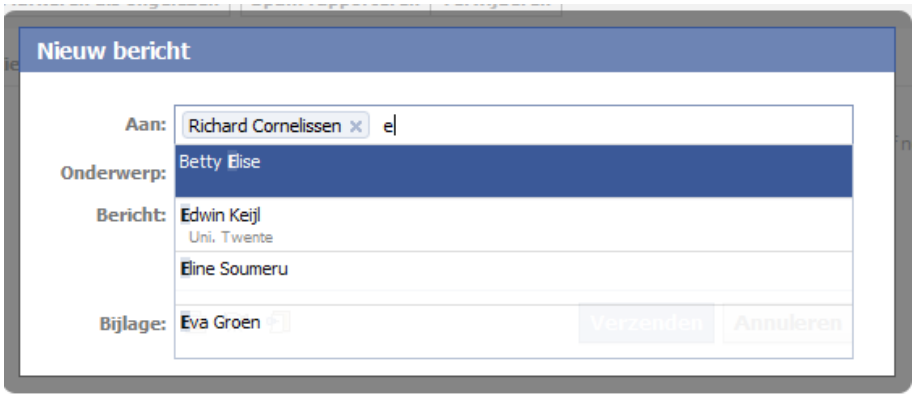

Figure 3.7: Screenshot of Facebook's receiver selection

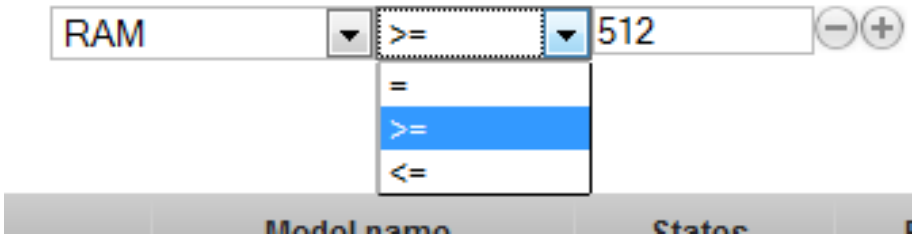

Figure 3.8: Screenshot of RAM filter

Figure 3.7. Here, the user starts typing and results pop up in the list below the input element. Then the user can use the arrow keys to select a receiver and press enter to add him or her to the list of receivers. The filter does the same: it knows a list of possible values and searches for possible values while the user is typing.

Memory (RSS), memory (VSIZE), runtime, states, transitions, RAM, and disk space filters

These filters require the user to specify a number and specify whether he wants results in which the value is equal to, less than or greater than that number, as in Figure 3.8.

Date filter

This filter requires the user to specify a date to filter on, and whether the benchmark runs have been executed before, after or on this date, an example of this filter is shown in Figure 3.9. The format for the date is yyyy-mm-dd.

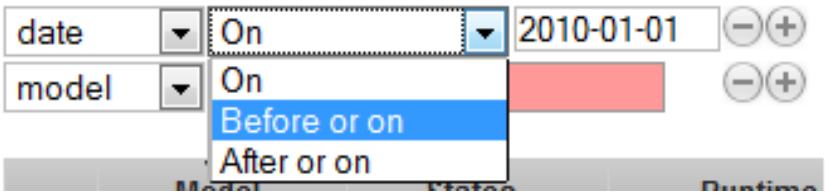

Figure 3.9: Screenshot of date filter

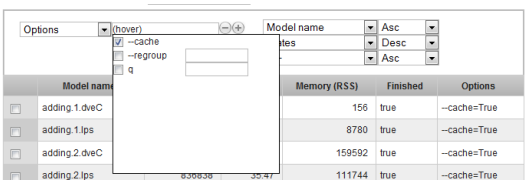

Figure 3.10: Screenshot of options filter

Options filter

This filter consists of a list of options that have been registered in the database, like in Figure 3.10. Only the options that are possible in the QuerySet the user is viewing are displayed. The user is able to check which options are used for benchmark runs and, if necessary, what value that option has.

Finished filter

The finished filter requires the user to select True or False, depending on whether he wants to see the benchmark runs that have finished correctly or that havent finished correctly. An example is shown in Figure 3.11.

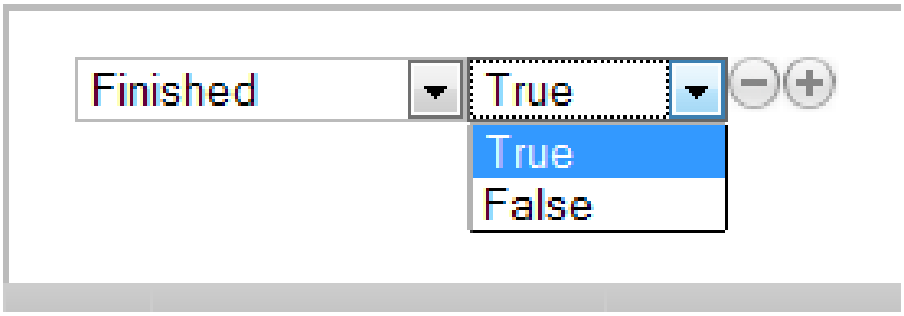

Figure 3.11: Screenshot of finished filter

Figure 3.12: Screenshot of paging features and buttons

Paging

Though we first thought paging would not be necessary, we noticed Firefox did not work well with a large set of benchmark runs in the benchmark table. To fix this, we added paging, so the results the user sees are limited. The user is also able to specify the page size, which is 25 by default, and is able to jump to the first, last, next and previous page through the interface shown in Figure 3.12.

Select all and invert selection

For better usability, the user is able to select all benchmark runs in the QuerySet he is viewing by clicking the "All"-button or inverting his selection with the "Invert"-button. These buttons work on all benchmark runs in the QuerySet, not just on the page the user is viewing at that moment. After the "All"-button has been clicked, it changes into a "None"-button, meaning that clicking it clears the selection.

Exporting

The user is able to export the selected benchmark runs as a CSV-file by selecting benchmark runs in the benchmark table and clicking the "Export"-button.

Filter on selected benchmark runs

The user is able to limit the benchmark runs he wants to see in the benchmark table by selecting these benchmark runs and pressing the "Filter Checked"-button.

3.4 Comparisons

BeAT allows the user to analyze the data in two different ways. First of all, the user can look up the raw data as it would appear in the database on the benchmarks page, see section 4.3. Second, the user can produce two types of graphs by selecting a set of properties on a page containing a form. We will refer to these graphs as comparisons. Both graph types (Comparison and ModelComparison) are discussed below and in section 4.4. Both types are recorded in the

database for future re-generation; the database model used is also discussed, in section 3.4.3.

3.4.1 Comparison

On the "compare tools" page, the user can select two sets of benchmarks by specifying a algorithm, tool, version and the options for each set. We intersect these two sets on their models, so that all models that ran on only one of the two combinations selected by the user are ignored. We then aggregate the set, so the result set consists of a model associated with data from two different benchmark settings. The current pieces of data are run time and virtual memory size, both of which are plotted in seperate scatterplot. The axes of this plot represent the combinations the user selected.

The graphs containing this data are shown on the result page, including the specified filtering properties. The data is also represented as a table below the graphs. The Flot JavaScript library is used to produce the graphic representation. Flot was chosen because it enables us to create interactive graphs, in particular that the user can hover over datapoints to see more information about that specific item. Each graph can be exported to pdf, ps, eps or svg format using the MatPlotLib python framework. MatPlotLib makes it easy to export anti-aliased graphs.

3.4.2 ModelComparison

The "compare models" page works similar to the "compare tools" page. Here the user selects a tool, algorithm, option combination and data type. BeAT redirects the user to a similar result page, producing a graph on which each line represents one model. The y axis represents the selected data type, while the x axis shows the revision date of the AlgorithmTool on which the benchmarks was performed.

3.4.3 Database Model

The full database model is shown in figure 3.13. The core tables are Comparison and ModelComparison. They both contain `name`, `date_time` (last modified), `hash` (the auth key) and a foreign key `user` to the User table. The auth key is a hash used to allow users without access to view the particular graph, see section 3.5 for more information.

The `ModelComparison` table has a foreign key to the `Tool` and `Algorithm` tables, representing which combination was selected by the user. A Many-To-Many relation called `optionvalue` with the `OptionValue` table is used to store all the options selected by the user. The type of data to be plotted is stored in the `type` attribute.

The `Comparison` table has two foreign keys to the `AlgorithmTool` table, one for each selected combination, instead of the keys to `Tool` and `Algorithm`.

Each of these can have user-selected options; those options are represented by two Many-To-Many relations with the `OptionValue` table.

The difference between the two tables is caused by the fact that the `AlgorithmTool` table contains a version string; this is required to identify a unique entry in the case of `Comparison`, while in `ModelComparison`, an abstract representation is required (as this graph iterates over the different versions). Possible future types of graphs can be in a manner similar to these two.

3.5 User management

Not all pages in BeAT should be publicly accessible. For example, visitors should be allowed to view the frontpage and other general website information. Only logged in users should have permissions to access the data, generate graphs and create job scripts. The data added to the database is available to all users that have login credentials, while generated jobs and graphs are assigned privately to the user that generated them. If needed, a superuser has the power to view and modify this information.

Sharing results publicly is possible, through a provided permalink on each result page. The data that can be shared currently only includes both types of graphs. The permalink is a simple URL, which provides an argument to the HTTP request the unknown user will make, allowing this user access. Note that anyone with access to the URL may access the data: this mechanism should therefore not be considered a security measure.

To manage users, the standard Django user authentication system is used. Through django's default administration interface (accesable through a link on any BeAT page), users can be added, if the user is logged in as a superuser.

3.6 PBS script generation

Not all the data relevant to a benchmark is present in the default output of the ltsmin tools. Data such as the name of the node on which it was executed, the operating system running there, the amount of memory, the version of the tool being used, the option given to the tool, and the date on which the execution took place are examples of that data. To make sure this data was available in the log files of future runs it was decided that a header containing that data should be available. An easy way to do this was to make sure the scripts that are executed by the scheduler collect and output this information before running the benchmark itself.

With the above information in mind a python class was designed that would generate PBS scripts that could be submitted to the scheduler. The generated PBS files are based on the files generated by a shell script that was already is use, and also contains a few lines that collect and output the data we required. During the project the scripts had to be changed a couple of times and we came to the conclusion that it was easier to switch to a template-based system, which

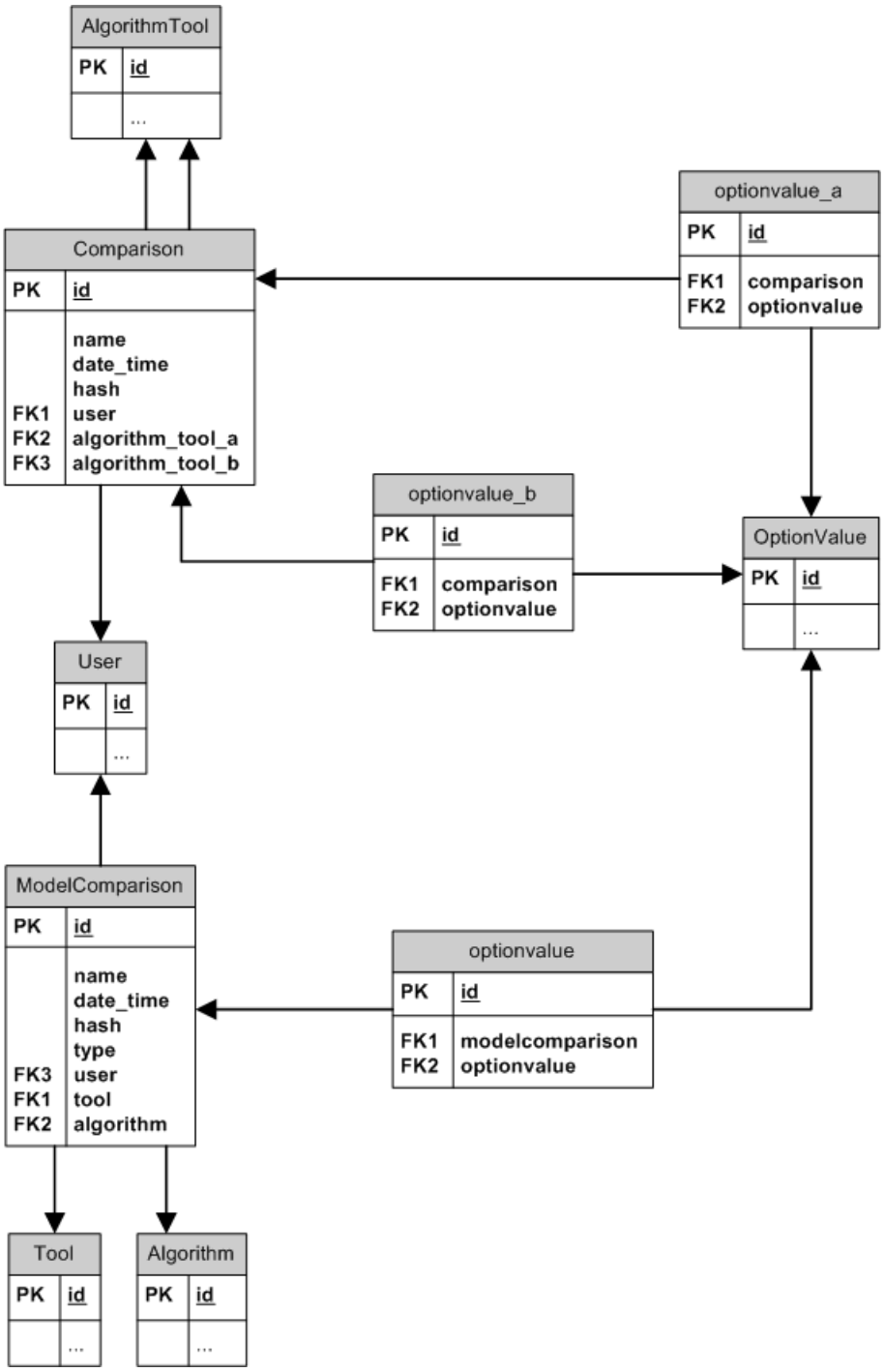

Figure 3.13: The database model for comparisons.

is what we included in the final product. Because the required templates were very simple, we could use Django's built-in template support, which supports everything we need, even though it was originally designed to generate webpages.

To make script generation available to the end user we chose to create a webpage that allows the user to perform the generation of a single job with some set of user-supplied options, as well as the generation of a complete test suite that contains all relevant permutations of options for a single model. The permutations were provided by our client and also contain some types that are not yet supported, concerning jobs that should run on multiple nodes. We have chosen to also generate these, as we intend for BeAT to be extended to support these jobs too, in time. At the request of our client we also added support for saving specific jobs. When a user creates such a job and supplies a name, the options supplied by the user are saved under that name, and he or she can reload that information at a later time and produce the same job again, or alter this job.

Chapter 4

Implementation

The following chapter will discuss the issues we faced and choices we made during the implementation of BeAT. We start with a brief overview of Django, followed by an explanation of administrative backend of the system. Afterwards we discuss the user-facing website.

4.1 Basic Django Information

This section contains some basic information about the filestructure that Django uses. Anyone already familiar with Django can probably skip it. Almost all of our python code uses Django in some way, and is installed in Django as an application ("app"). A noteable exception is the filereader, which is not directly accessable through the website and for that reason has been placed somewhere else. Because all the apps have a few things in common, we will cover that information here in order not to repeat the same information when discussing the implementation details of each app.

All of our apps are located in the BeAT root folder. Each app has its own folder, for example, `beat/benchmarks` and `beat/graphs`. In general apps do not depend on each other with the exception of `beat/benchmarks`, on which most apps depend. This is because it is the main application and contains the classes required to interact with the benchmark data in the database.

There are a few python files that are are present in most or all of the apps. The most common ones are listed below, along with some information concerning their contents and purpose. When the description of the implementation of some part of the website does not specify where the code for one of the things mentioned below is located, it will be in the default file. For a more in-depth explanation on Django we recommend a visit to the Django documentation [8]. If appropriate, sections of this documentation are referenced. Unless otherwise noted, the documentation for version 1.2 is intended.

Filename	Contents
`__init__.py`	This file is required to tell Python that the directory is a Python module and can be imported (and imported from).
`admin.py`	The Django admin interface can be tailored to the user's needs by creating subclasses of the `django.contrib.admin.ModelAdmin` class in this file.
`forms.py`	Contains a description of one or more forms used on the website. Each form is described as a python class.
`manage.py`	A thin wrapper around `django-admin.py` that takes care of two things for you before delegating to `django-admin.py`: It puts your projects package on `sys.path` and it sets the `DJANGO_SETTINGS_MODULE` environment variable so that it points to your projects `settings.py` file. `django-admin.py` is Django's command-line utility for administrative tasks.
`models.py`	Contains one or more classes, each of which is a representation of the data held in a particular table in the database.
`settings.py`	The automatically generated Django settings module, including settings for the database, caching, authentication, template directory and media directory.
`urls.py`	A mapping of URL patterns (in the form of regular expressions) to functions in `views.py`.
`views.py`	This file contains code that presents data to the user for one specific application on the website.

4.1.1 Template structure

Django provides a template system with tags which function similar to some programming constructs. Though the template system is documented in the Django documentation [9], we explain the basics here. A template contains variables, which get replaced by values when the template system is evaluated and tags, which control the logic of the template. Context variables passed to a template can be accessed using `{{ variable.attribute }}`. Tags are defined as `{% tag %}` and can be used for more complex purposes as if-statements, for-loops and template inheritence (using the `block` tag).

The HTML structure is defined through this template inheritence. `base.html` defines the basic HTML structure that is used for every page and inclusion of CSS and JavaScript files. More importantly, five different blocks are defined that can be extended in a specific page as shown in figure 4.1.

head can be used to add extra things to the header, such as JavaScript files.

title overrides the title in the `<title>` tag.

structure defines the basic layout of the page, this is overridden in `base_site.html`.

sidebar allows you to add extra items to the navigation bar on the right side of the page, if that may be needed.

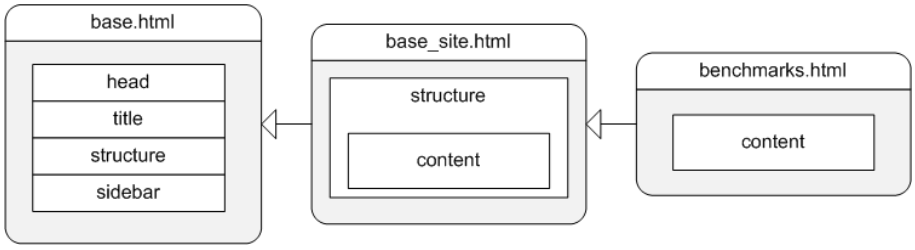

Figure 4.1: The basic template structure that is used for BeAT. Here `base_site.html` only extends the content block.

Nearly every page that uses a 'standard' one column layout that extends `base_site.html`. This template only overrides the `structure` block, defines the one column layout and defines a new block called `content` inside of that column. This makes it possible to use a different structure as on the frontpage, if this is desired. As described above, all other blocks can be overridden as well if there is a need.

So to make a new template, create an HTML document in the preferred application directory of the templates directory. It needs to extend the `base_site.html` template and override the content block to be able to contain any new content. Furthermore, it can optionally override any of the blocks mentioned above.

Navigation

The `sidebar` block contains the navigation of the websites. All links are defined as list items with an anchor tag that uses the Django `{% url %}` tag to output absolute URLs directly to a specific view. This prevents hard coding the URLs, so they can easily be changed in `urls.py`.

4.2 Database Population

In this section, some implementation details for the various database population scripts are discussed. Insight for future development is also provided.

4.2.1 Log movement

This section will give a short description about the log movement process and the issues we considered while implementing it. Part of this feature has been discussed in section 3.2.1.

Processs

Before logs can be processed they need to be available on the server. The first step in the regular flow when importing logs is moving them to the location where they can be proccessed. A few options were discussed in section 3.2.1. Since our users are quite technical and used to `ssh` and a ssh key infrastructure, we decided to process the files on the server and use `scp` to copy the logs. This removes the need to install python with the proper libraries on the client PC, keeps the SQL server secure and does not waste bandwith, because the logs need to be stored on the server anyway.

Implementation

Implementation of the log movement was straight forward. The first part involved setting up a usergroup on the server and creating a directory, which was periodically checked by the filereader script. The output from the filereader is not saved at the moment (since all regular logs should be parseable), but this warrants a look in the future.

4.2.2 Filreader

In this section, the filereader script, which imports data from provided logs, will be discussed in detail. Some information will be provided on how to improve the script; for those sections of text, familiarity with python is assumed. The flowchart in figure 3.3 is used for the structure of this section.

Call & Parse options

The filereader is intended to be called from a shell. At some point in the project, support for using it from another python script existsed; however, currently, this functionality is considered depricated. It is possible to include this functionality again, but the `Filereader.main()` method should be updated to reflect changes in the available options. From the command line, a number of options can be provided, followed by any number of paths to a file. Assuming the use of bash, one should use xargs to allow any amount of arguments to be passed. The disadvantage is that the counting of errors no longer works (since xargs causes the filereader to be called on each file individually instead of passing a large set of arguments). This is not needed if the code runs on a linux kernel version after 2.6.23, from which onward an arbitrary amount of arguments can be passed [3]. Options for the filereader are parsed using the optparse python module, which makes adding additional options (almost) trivial. See the python documentation of optparse [6] for more information.

Splitting into logs

This code is pretty straight-forward, simply iterating over the lines in a file. One notable thing is that the current scheme allows data to be placed in between

the ending of one log and the beginning of the header of another. During our tests, we have encountered some issues with loading ltsmin modules; generated error messages in between logs are ignored. It should be possible to parse this information and display it to the user.

Header Analysis

The header analysis parses the header, as documented in B. A regular expression extracts the relevant data from the file, elements of which are processed. The most relevant are first, the tool version, which looks like `ltsmin-1.5-20-gdd431`, where the last six characters are the initial six of the git revision from which the tool was compiled (for the ltsmin tools, this string can be retrieved through `<executable> --version`). Second, the start time/date is converted into a datetime type (the type used by django's `DateTimeField`, the field this data needs to be inserted in), from the python module datetime. Finally, the call (the string used to get the tool to execute) is analyzed to find the tool name and algorithm name (through a regular expression), as well as applied options and the used model (using the gnu_getopt method provided by python's getopt module). Note this is slightly different from what is used in ltsmin; there, the C module popt is used. However, gnu_getopt provides the needed basic features, without the need to package a C module in BeAT. If needed, however, it should be possible to use python's ctypes module to load C code and use it to parse options. In theory, it should even be possible to allow the user to specify which module should be used, by adding an option_module field to the AlgorithmTool table. Because of the complexity of adding these features and the lack of a significant advantage over gnu_getopt, we have chosen to leave this as future work, however.

The specification of the used hardware and details about the kernel is also in the header; however, currently, we only use the nodename and the kernel-release fields, which are those that were considered most relevant. More information (for example, installed library versions) , as well as the data that not currently used, can be added to the header and the Hardware table. To collect this information, the regular expression should be altered. Support for multiple machines is currently not provided; adding this feature is a non-trivial task to perform. Options would include using a delimiter to separate the information for different machines (eg, assuming a delimiter `|`, Nodename could contain `node1|node2|node3`), which should be relatively easy to split into different hardware items. Assuming these items are placed in a list, the filereader can already insert multiple items into the database (ie. an alteration of the database structure is not required).

Finally, relevant regular expressions to parse the log are extracted from the database and returned with the retrieved data for futher processing.

Log Analysis

The first regular expression, which was retrieved based on tool, algorithm and version, is used to get the basic data out of the log: finish status, state and possibly transition counts, memory VSize and RSS, and user, system and elapsed time. These are retrieved through group names, which are, respectively: `kill, scount, tcount, vsize, rss, utime, stime` and `etime`. The group `kill` should match to the empty string if execution was successful. The first regular expression may contain additional information, through the use of additional groups as specified in section 3.1.2. As mentioned in that section, the other regular expressions, belonging to the options, are considered in a similar manner. Again, it should be noted that the regular expressions are in-order and data will be overwritten if the same group name is used twice. All the collected information, including that of the header, is placed in a python dictionary type and returned. This dictionary is provided to the database writing logic, which inserts the data after performing a basic validity check.

Data Validation

Currently, data validation occurs through a simple method checks whether all the fields are properly filled. It can mark data as invalid, or modify it to correct errors. An example of the latter is that when transition count is not in the dictionary (for example, because the `-reach` algorithm is used, which cannot collect this data), it is set to zero (as opposed to leaving it on the value None). Due to lack of knowledge on our part, not much interesting happens here. However, a programmer with experience with ltsmin should be capable to build a useful extension that can detect potentially invalid, or perhaps even better, interesting data, before it is entered into the database. Since the filereader is build in an object-oriented manner, the function can simply be overwritten by a sub-class.

Data Insertion

Inserting data into the database consists mostly of simple `get_or_create()` queries, which create items as needed. If the `--override` switch is provided, the Benchmark row is deleted if the `get_or_create()` query returns that it already exists. Should the `get_or_create()` query result in an exception that indicates multiple objects match the criteria, an error will be printed. This type of error implies the database contains invalid data, so the user is informed to take action, restoring the integrity of the database.

4.2.3 Tool Upload

In this section, the Tool Uploader will be discussed. Some information will be provided on how to improve the script; for those sections of text, familiarity with Python and JavaScript is assumed.

Webpage

The tool upload is preformed form the website with the page located under Tool Upload. On this page there are several options fields in the form of an `django.forms.CharField`. The page contains the five main fields as described in tooluploadand the two fields for the checking of the file. The cheking is done by JavaScript.

Tool analysis

When the Send button is pressed the form is sent to `views.py`. The first thing that is done is getting the gitrevision to check if the tool exists in the gitrepository where all the tools should be stored. If not, the form is returned to the user, displaying an error that the git version does not exist.

Tool Storing

If the given git version does exist the date is pulled from this git commit. It is then parsed to a datetime object and together with all the pieces of data it is then parsed into the database. Inserting data into the database consists mostly of simple `get_or_create()` queries. Before the data of the options field is placed into the database it first needs some adjustments. The options string is pulled apart by using `split('\n')` so that every line is a different string. Each line is then read out and placed into the database. In the final part of the Tool Upload the user will have to go to the Django admin page to add regexes so that any changes that the options bring to the log can be parsed.

4.3 Benchmark Table

The benchmark table is implemented in a client server strategy. The server side receives a request, creates a response and sends it to the client. The client creates requests and handles responses to show them on screen.

4.3.1 Server side

The server side receives requests from the server. This request contains a JSON [2] object with data in it, which can be decoded with the python json library [5]. This object contains information to create a specific QuerySet of the Benchmark table, namely:

- Filters to be applied on the initial QuerySet `Benchmark.objects.all()`;
- Columns to be sent to the client;
- What the QuerySet should be sorted on.

We will now describe how the server side works.

Filters

?? After receiving a request, the server starts making a QuerySet with the data the user has sent. The first step is applying the filters. At first, the selection filter is run, which limits the QuerySet to all the id's of the benchmark runs the user has selected before pressing "Filter Selection" by evaluating:

```
if len(data['subset']) != 0:
    qs = qs.filter(id__in = subset)
```

Where `qs` is the QuerySet and `data` is a dictionary containing an array of integers called `subset`. After applying the selection filter, each filter in the filters array is applied:

- If the filter is one of the list filters (see below), every possible value of this field is stored in an array, so that the backend knows what values are available for the filter;
- The filter is applied by `qs = filter.apply(qs)`.

BeAT recognizes the following filter types:

- Model name;
- Algorithm name;
- Tool name;
- Algorithm-tool version;
- Memory (RSS);
- Memory (VSIZE);
- Runtime;
- Number of states;
- Number of transitions;
- Date;
- Options;
- Finished;
- Computer name;
- Processor;
- RAM;
- Kernel version;

- Disk space;
- Selection.

Except for the selection filter, all filters have been ordered in classes specified in `beat.benchmarks.filter`:

- `ListFilter` for the filters of type model name, algorithm name, tool name, algorithm-tool version, processor, kernel version, and computer name.
- `ValueFilter` for the filters of type memory (RSS), memory (VSIZE), runtime, number of states, number of transitions, RAM, and disk space.
- `OptionsFilter` for the options filter;
- `DateFilter` for the date filter;
- `FinishedFilter` for the finished filter.

These classes all use the QuerySet.filter function [11].

List filter The `ListFilter` class gets a list of strings containing names for either models, tools, algorithms, computers, processors, algorithm-tool versions, or kernel versions. When calling it's `apply` function with a QuerySet `qs`, it checks what filter type is given and filters `qs` so that only those benchmarks with corresponding values are included.

Value filter The `ValueFilter` takes a certain integer or real value and a style of filtering like 'greater than or equal' or 'less than or equal'. When calling the `apply` function, the corresponding field in the database is filtered with that value and style.

Options filter The `OptionsFilter` contains a list of tuples of keys of options in the database and values for that option. Its `apply` function finds the corresponding `OptionValue` object for this key `k` and value `v` for every option and value by calling `ov = OptionValue.objects.get(option=k,value=v)`. If the `OptionValue` does not exist, an empty QuerySet is returned, else, the QuerySet `qs` is filtered by:

```
qs = qs.filter(optionvalue__in = [ov.id])
```

Date filter The `DateFilter` splits the received date in three parts: year, month, and date. These values are used to filter the QuerySet on benchmark runs that have been run before, after, or on a certain date by applying one of the following commands:

- `qs = qs.filter(date_time__gte = datetime(year, month, day, 0, 0, 0), date_time__lte = datetime(year, month, day, 23, 59, 59))`, if the benchmark runs should be on the specified date.
- `qs = qs.filter(date_time__gte = datetime(year, month, day, 0, 0, 0))`, if the benchmark runs should be on or after the specified date.
- `qs = qs.filter(date_time__lte = datetime(year, month, day, 23, 59, 59))`, if the benchmark runs should be on or before the specified date.

Finished filter The `FinishedFilter` takes a Boolean value which it applies on the QuerySet `qs` by calling:

```
qs = qs.filter(finished__exact=self.finished)
```

Column Selection

The server determines which columns the client can choose from. These columns are divided into two categories: standard columns and extra columns. The standard columns are the same as the filters described above in section **??**, minus the selection filter.

The extra columns are only available when one or more of the benchmark runs in the used QuerySet contain it. The values for these columns are kept in the table specified by the `ExtraValue` Model in `beat.benchmarks.models` and are mainly generated by specific choices for options and/or algorithm and tool combinations.

All available columns for a QuerySet (which are the standard columns combined with the possible extra columns) are sent to the server in a `JSON`-object array. Each of the objects in this array contains the attributes 'name' and 'dbname', which stand for the name to be displayed to the user and the name to be send to the server respectively. As an example, the object for the model name column is:

```
{
    'name'     : 'Model name'
    'dbname'   : 'model__name'
}
```

The objects for the extra columns have both the `name` and `dbname` attribute set to the name specified in the `ExtraValue` table.

Besides determining and sending all available columns, the server must also apply the columns the user has chosen to the QuerySet. This is handled by the `addColumns` function, which takes a QuerySet and an array of column names.

This function checks every column name c in columns and sees the following cases:

- The column name is equal to the options column - in which case it needs to execute a rather difficult query including concatenating strings and rows;
- The column name is in the `HARDWARECOLUMNSARRAY` (computername, processor, ram, kernel version, disk space) - in which case it should get the values from the `Hardware` table;
- The column name is not in the `STANDARDCOLUMNSARRAY` (containing the names of all standard columns) - in which case it should get the values from the `ExtraValue` table.

All remaining column names are automatically added when the command

```
apply(qs.values, columns + ['id'])
```

is evaluated [13], which also adds the identifier-column that is always needed in the response. This command completes the column selection.

Sorting

The client sends an array containing arrays of column names and either 'ascending' or 'descending' to the server, specifying on which columns it should sort. For example, the array

```
[
    ['model__name', 'ASC']
    ['total_time',  'DESC']
]
```

specifies that the server should first sort the QuerySet on the model name (ascending) and then on the runtime (descending).

The `sortQuerySet` function sorts a QuerySet. If the array the client sent is empty, the function sorts the QuerySet on the identifier and returns it. If the array is not empty, the function goes over each tuple of column name and order, creating an array of keywords: the column name itself if the order is ascending, the column name preceded by a '-' if the order is descending. The example above would lead to the array `['model__name','-total_time']`.

After all tuples are handled, the QuerySet is sorted by `apply(qs.order_by, array)` [12].

Context

Every `ListFilter` has a certain context: this is the list of all possibilities the filter can have as values. These possibilities are calculated either before a filter of the corresponding type is applied or when all specified filters have been applied. For example, when first applying the filter runtime $\geq$ 12.5, then getting the

possible model names and then applying the filter model name on ['leader.b'], the user sees all possible model names after the runtime filter is applied. This way, the user can't be surprised by an empty QuerySet when a model is chosen that is unavailable after applying the runtime filter.

The function used to get the context of `ListFilter` filters is `getContext`, which calls the corresponding get-function.

Response

The response the server sends to the client is structured as follows:

```
{
    'benchmarks'        : [],
    'columns'           : {},
    'benchmark_ids'     : [],
    MODEL               : [],
    ALGORITHM           : [],
    TOOL                : [],
    OPTIONS             : [],
    CPU                 : [],
    COMPUTERNAME        : [],
    VESION              : [],
    KERNELVERSION       : []
}
```

The values in the dictionary contain the following data:

- `benchmarks` is the array of benchmarks to be shown in the benchmark table;
- `columns` is the list of possible columns the user can select from;
- `benchmark_ids` is the list of all the benchmark identifiers in the QuerySet before paging (needed for certain client-side functionality like selecting all benchmarks);
- `MODEL`, `ALGORITHM`, `TOOL`, `OPTIONS`, `CPU`, `COMPUTERNAME`, `VERSION`, and `KERNELVERSION` all contain context for `ListFilters`.

This python dictionary will be transformed into a `JSON`-String and send to the client.

Flow Chart

The server handles requests from the client as specified in the flow chart in figure 4.2.

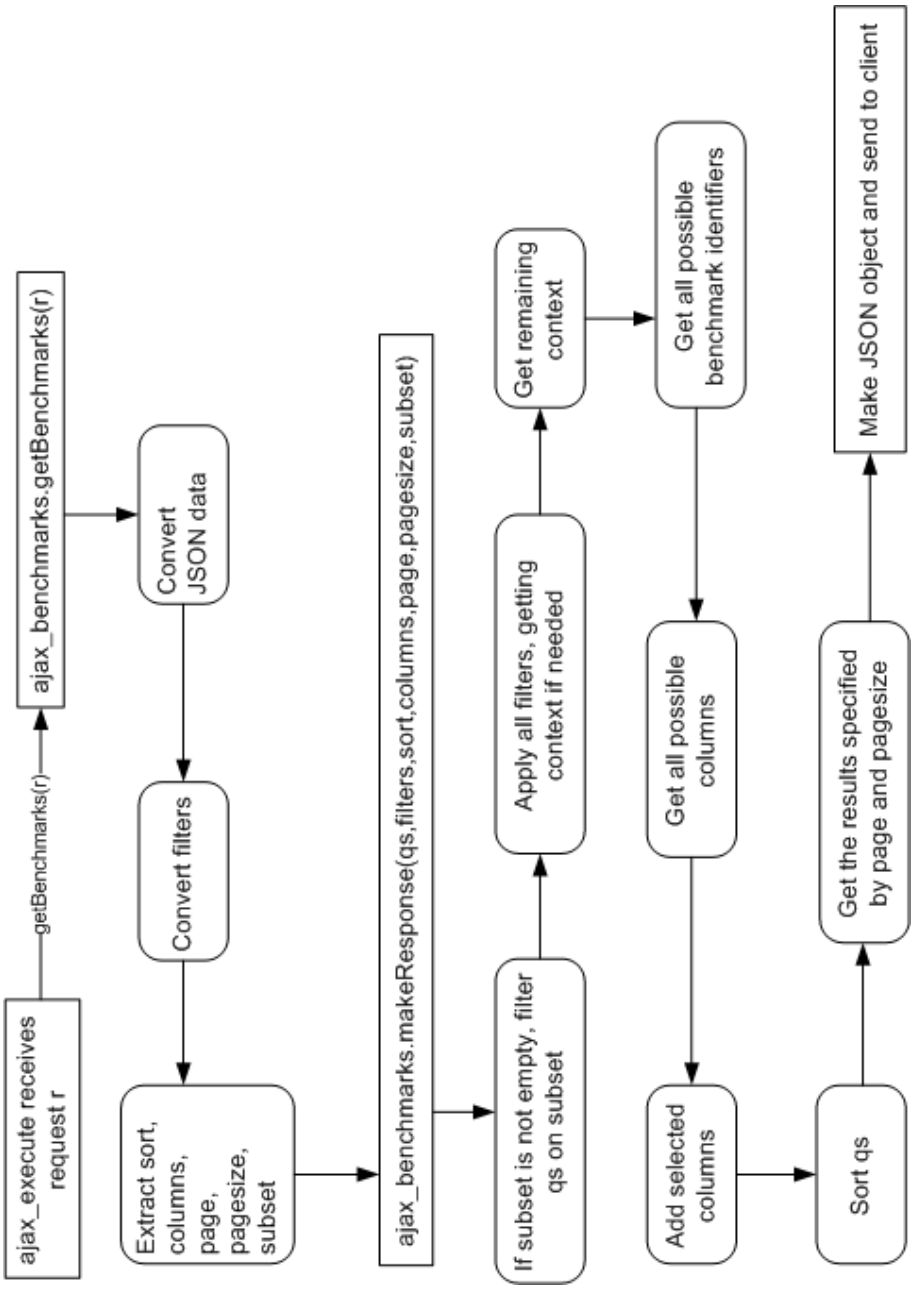

Figure 4.2: Flow chart of the server side of the benchmark table

4.3.2 Client side

The client side collects data and automatically sends a new request when there is a change in a filter, in the columns selected, in the sorting, when the pagesize is changed, or when another page is requested. It also keeps a list of identifiers of the selected benchmarks, a list of all benchmark identifiers, and a list of all possible columns.

Filter Table

The filter table, which can be found in `beat/site_media/js/filtertable.js`, is written as a jQuery [7] plugin, meaning that it can transform an HTML-element, in this case a TABLE-element, into a filter table. The filter table handles all the possible filters (except for the selection filter), meaning it makes sure the user can select a filter type, select a filter style (if necessary), type, select, add, or remove values, add or remove filters, etc.

For every filter in the filter table, an object is stored. This object is structured as:

```
{
    type (String),
    row (integer),
    style (String),
    value (String/integer/array),
    error (Boolean)
}
```

In this structure, `type` specifies the type of the filter, `row` specifies the row in which the filter is shown in the filter table, `style` indicates the filtering style the user choose (which may be empty), `value` indicates the value the user has specified, and `error` indicates if there is an error in the filter (e.g. when `type` is set to "empty" or `value` is empty).

In the filter table, the filters have been generalized into list filters, value filters, a date filter, an option filter and a finished filter, which we will look at now.

List filters The list filters include the filters for model name, tool name, algorithm name, computer name, processor, algorithm-tool version, and kernel version.

For the user to be able to specify which values he wants in the QuerySet, he needs to be able to select one or more from a list of possibilities. As said before, we have chosen for a user interface similar to that of Facebook, using an edited jQuery plugin called tokeninput [17]. Originally, this plugin made an AJAX request to the server to ask for possible values are, but because we already receive these possibilities when doing a request, we modified it to search in a locally stored array. The possibilities in this array have been filtered on the server, so that the user won't get an empty QuerySet. Furthermore, when

a request is made and a response has been received, all selected values in the list filter that are not available anymore are removed.

In the list filter object, the `style` is left empty and `value` contains an array of integers pointing to identifiers in the database. The `error` value is set to True if the value contains no identifiers.

Value filters The value filters include the filters for memory (RSS), memory (VSIZE), runtime, number of states, number of transitions, amount of RAM, and amount of disk space.

The user is able to specify an integer or real value and a style on which the filter should check, for example, it is possible to specify the QuerySet should only contain those benchmark runs of which the measured runtime is greater or equal to 10.3 seconds.

To be able to do this, the value of `style` in the filter object contains either "equal", "greaterthan", or "lessthan" and `value` contains the integer or real the user wants to check on. `error` is set to True when the textbox of the value is empty or does not contain an integer/real.

Date filter The date filter works the same as a value filter, except that its `style` can be equal to either "on", "before", or "after" and its `value` should contain a date in the form of yyyy-mm-dd. The `error` value in the filter object is set to True if the textbox of the filter does not contain a string in the form of yyyy-mm-dd.

Options filter The option filter is shown as a list of possible options in what is called a mega dropdown menu; when the user hovers over a specific element, a small menu appears. In this mega dropdown menu, a list of checkboxes with option names and, if the option requires an argument, a textbox for the user to specify a value for that argument.

The `value` of an option filter, which consists of a list of selected options and a list of values for those options, is stored in the filter object as an array consisting of two arrays: the first containing the identifiers of the selected options, the second containing the values specified by the user for those options. If the option takes no argument, it's value is specified as "True" in the value of the filter object. The `style` in the filter object is always empty.

The `error` value in the filter object is set to True if no option is selected or when an option that requires an argument is selected, but no value has been specified.

Finished filter The finished filter's `value` can only be True or False, which is stored in the value of the corresponding filter object. The `style` in the filter object is always empty and the `error` value of the filter object is always False, as the finished filter is displayed as a selection menu, with True being the default value.

Communication Filter table and Benchmark table

The benchmark table, which uses the filter table, can get the filters by calling filters on the filter table element. When doing this, the filter table returns all the filter objects that do not contain an error, so that the benchmark table can send a request to the server.

After receiving a request from the server, the context of the possible options, models, etc. might have changed, so the benchmark table will call `updateContext` on the filter table, after which the filter table will apply the new context to every filter.

When a filter is changed, the filter table automatically sends a trigger with the value specified in the constant `TRIGGERCODE`. When the benchmark table receives this trigger, it will use the filters function to make a new request to the server.

Benchmark Table

The benchmark table, specified in `beat/site_media/js/benchmarktable.js`, handles the following tasks:

- Selection of columns;
- Sorting;
- Creating the filter table;
- Paging;
- Filter on selection;
- Selecting all/invert selection;
- Exporting selected benchmark runs;
- Making requests to the server and handling them.

The benchmark table contains some variables to store data, namely:

- `data`, contains all data to be sent to the server;
- `ASCENDING`, contains the constant value for ascended sorting;
- `DESCENDING`, contains the constant value for descended sorting;
- `DELAY`, contains the constant value indicating the delay between changing some of the data and sending a request to the server;
- `checked_benchmarks`, contains the identifiers of all selected benchmark runs;
- `updating`, Boolean value specifying whether a request to the server is still being handled;

- `update_again`, Boolean value specifying whether some data has changed during an update;
- `benchmarks`, contains all received benchmarks from a response;
- `benchmark_ids`, contains all identifiers in the QuerySet;
- `columns`, contains all possible columns;
- `table`, keeps the filter table;
- `timeout`, keeps the value resulting from a setTimeout function call;
- `previousRequest`, contains the `JSON.stringify` value of `data` while updating, will overwrite `previousSucceededRequest` if a request completes successfully;
- `previousSucceededRequest`, contains the `JSON.stringify` of the latest succeeded request, if the `data` of a new request equals this value, the request will not be sent to the server.

Column selection The selected columns are stored in `data.columns` as an array of strings. All possible columns are displayed in a mega drop down menu as a list of checkboxes. Each of these checkboxes has a change function specified, that update `data.columns`. Everytime a request is made, the function `updateColumns` is called, which renews the mega drop down menu.

Sorting The variable `data` also contains the sorting the user specified. This is structured as an array keeping tuples (in JavaScript arrays with two values) of strings: the column name and ascending/descending. This means the user can sort on any column he has selected in the column selection.

The table next to the filter table contains the mechanism for sorting. The user can first specify which column name he wants to sort on and whether he wants to sort ascending or descending. After specifying a column name, a new row is added to the table to give the user the ability to sort on multiple columns.

As an example, if the user first wants to sort on model name (ascending) and then on runtime (descending), `data.sort` contains:

```
[
    ['model__name',  'ASC']
    ['total_time',   'DESC']
]
```

Creating filter table When the benchmark table is loaded, it creates a filter table by calling:

```
$("#filters").filtertable([], {});
```

Paging The current page and pagesize are kept in `data.page` and `data.pagesize` respectively. When the filters or sorting has changed, the `page` is set to 0 and the benchmark table is updated. When clicking the first, previous, next, or last button, the first, previous, next, or last page respectively is loaded by setting the `page` and doing an update.

The `pagesize` is set in a form, with a default of 25 results per page. When typing in another value and pressing enter/return, the `page` is set to 0, the `pagesize` is set to the new value and the benchmark table is updated.

Selection filter When pressing "Filter Checked", `data.subset` is set to the identifiers currently in `checked_benchmarks`, the `page` is set to 0 and the benchmark table is updated. After doing so, the QuerySet is filtered to only contain these identifiers, before the filters, sorting, column selection, etc. are handled.

This way, the user can specify it only wants to do something with these specific benchmark runs.

Select all and invert selection When the user presses "All", `checked_benchmarks` is set to a copy of `benchmark_ids` - the identifiers of all benchmarks in the QuerySet, meaning that it is cross-page instead of only the current page. After this, the "All"-button changes into the "None"-button, which, when pressed, sets `checked_benchmarks` to an empty list.

The "Invert"-button inverts the selection: all identifiers that are in `benchmark_ids`, but not in `checked_benchmarks`, will form the new `checked_benchmarks`. Again, this works cross-page.

When clicking the "All"-, "None"-, or "Invert"-button, `updateCheckboxes` is called, which updates all the checkboxes in the benchmark table.

Export to CSV When clicking "Export CSV", the user can download a CSV-file containing all the information of the benchmark runs in `checked_benchmarks`.

Communication handling The `update` function is used to update the benchmark table. This function sets `data.filters` to the array of all filterobjects in the filter table that do not have an error and sets a timeout to call `makeRequest` if it's argument direct is not set. If direct is set, it does not set a timeout, but directly calls `makeRequest`.

`makeRequest` then sends an AJAX-request to the server, specifying that it should call `handleResponse` when it finishes successfully.

`handleResponse` will then update local variables, update the filter table's context, update the sort table, columns and `checked_benchmarks` and show the new results in the benchmark table.

Request

A request is stored in the `data` variable in the benchmark table. This variable is structured as follows:

```
{
    sort      : array,
    columns   : array,
    page      : integer,
    pagesize  : integer,
    filters   : array,
    subset    : array
}
```

Where `sort` contains the sorting data, `columns` an array of column names the user wants to see, `page` an integer of the page the user requests, `pagesize` the size of that page, `filters` an array of filterobjects from the filter table, and `subset` the subset of benchmark identifiers the user wants to see. As a side note, if `subset` is empty it is not processed on the server side.

4.3.3 Adding/Changing Functionality

Adding a filter

Adding a filter table means changing both the server and client side. On the server side, a new constant must be added in `beat.benchmarks.filter` to name the filter, for example, `COMPUTERNAME` contains the name for the Computername filter.

Then the following cases are recognized:

- The filter is a list filter, in which case the constant must be added to `LISTFILTERS` and `CONTEXTFILTERS`. Furthermore, a case for the new filter must be added to the `ListFilter` class;
- The filter is a value filter, in which case the constant must be added to VALUEFILTERS and a case must be added to the ValueFilter class to support the new filter.
- The filter is neither a list filter nor a value filter, in which case a new filter class must be added to `beat.benchmarks.filter`, extending the Filter class. This class must have an `apply` function which takes a QuerySet and returns a new QuerySet. Furthermore, a case for the new filter type should be added to `convertfilters` function.

If the new filter also requires certain context, but isn't a list filter (just like the OptionsFilter), the constant should be added to `CONTEXTFILTERS` in `beat-.benchmarks.filter` and to the variable result in the function `makeResponse` in `beat.benchmarks.ajax_benchmarks`. A corresponding get-function should be added in `beat.benchmarks.ajax_benchmarks`, which, in turn, needs to be added to `getContext()`.

On the client side, the constant should be added to `beat/site_media/js-/filtertable.js`, which in turn should be added to `ALLFILTERS` and to the `filtername`-function. If the user should only be able to use the filter once, the constant should also be added to `UNIQUEFILTERS`.

If the filter is a list filter or value filter, it should be added to `LISTFILTERS` or `VALUEFILTERS` respectively. After doing that, you are done adding the filter.

If, however, the filter is not a list or value filter, the following functions should be edited:

- `$.fn.updateContext`; if the filter does have context, it should be added here in the same way `OPTIONS` is handled, checking whether the `filterobj` is of the new filter type and if so, edit the `filterobj` and call `rewriteRow`.
- `makeRowContents`; here, the row for the filter is made. A case must be added for the new filter type.

With this, the new filter has been added.

Adding a column

The columns are completely server side, meaning no JavaScript has to be edited to add a new column.

To add a column, it should be added to `STANDARDCOLUMNS` and `STANDARDCOLUMNSARRAY` in `beat.benchmarks.ajax_benchmarks` if it is directly available in the `Benchmark` table or through one of its foreign keys (like `model` goes to `model__name`), or in `HARDWARECOLUMNS` and `HARDWARECOLUMNSARRAY` if the value is available for every benchmark and the value resides in the `Hardware` table.

Changing default columns

The default columns are defined in the initial `data` variable in `beat/site_media/js/filtertable.js`. To change the default columns, the user must change the value of `data.columns` to the default columns he wants. Keep in mind that the values in this array equal the `dbname` values in `STANDARDCOLUMNS` in `beat.benchmarks.ajax_benchmarks` (note that these constant values are defined in `beat.benchmarks.filter`) and that the order of these default columns must still follow the order at which they are defined in `STANDARDCOLUMNS`.

Changing column order

To change the order of the standard columns, the order of the values in the constants `STANDARDCOLUMNS` and `STANDARDCOLUMNSARRAY` in `beat.benchmarks.ajax_benchmarks` must be changed. The order of the extra columns cannot be changed, as they are queried from the database.

Again, keep in mind that the order of the default columns defined in `data.sort` in `beat/site_media/js/filtertable.js` must be equal to the order at which they are defined in `STANDARDCOLUMNS` in `beat.benchmarks.ajax_benchmarks`.

Changing default pagesize

The default pagesize can be found in `data.pagesize` in `beat/site_media/js/filtertable.js`. Here, you can edit the initial (and thus, default) pagesize.

When changing this pagesize, the benchmarks template must also be edited to show this new initial pagesize. This template can be found in `beat/templates/benchmarks.html` in a form called `pagesizeform`. Here, the value of the input-element must be changed to the new default pagesize.

Changing list filter result size

The size of the number of results shown when typing in a list filter can be found in `beat/site_media/js/filtertable.js` and is stored in the constant `LISTFILTERSIZE`. Changing this value will change the number of results shown when typing, but keep in mind that the higher the number, the longer the JavaScript will have to look for results.

4.3.4 Notes

Notes on database access and differences

Though we tried to use the Object Relationship Mapping of Django as much as often as possible, it was not always sufficient for the data we had to get from the database. To get the hardware columns, extra columns, and options column (computername, processor, etc.), we had to use manually written SELECT queries, as specified in the `addColumns` function of `beat.benchmarks.ajax_benchmarks`. Furthermore, one of these SELECT queries cannot be used over all SQL-servers and uses custom queries; the query that gets the Options column from the database. This query does concatenates all options used for a benchmark run, meaning it concatenates over multiple rows. The queries have been written for both SQlite3 and postgresql.

4.4 Comparisons

In the next two sections, the interactions and data flows for creating graphs will be discussed. Afterwards we will discuss our implementation issues with flot.

4.4.1 Scatterplot

The user starts at a form where he can select the options to be used for the comparison. The form fields are defined in `comparisons.forms.py` and the data is processed in the `compare_scatterplot()` method of `comparisons.views.py`. The options are discussed in section 3.4. The standard Django pattern for forms processing is used [16]. If the form is not submitted, then an empty form will be shown to the user. Otherwise, submitted form data will be handled: An entry in the `Comparison` table is added by calling the `create()` method. Next, the selected options are created in the many-to-many table. This many-to-many relation is needed because for one comparison, multiple options can be selected. Finally, the hash value for this comparison is calculated based on the ID. The user is then redirected to a page where the graph is displayed.

The `comparisons.views.compare_detail()` function first filters all data from the `Benchmark` (and related) tables in the database. Specifically the modelnames, memory and time data (for both benchmark settings) are zipped into a list of tuples. All this data is then sent to the `compare.html` template as context variables. Before the `HttpResponse` is returned, the user is checked for his authorization key. If the user has no permission to view this comparison, he is redirected to a `403 permission denied` page.

The template contains a static link to an image called `scatterplot.png`. In `urls.py`, this image is mapped to the method that produces the graph as a PNG file. The comparison ID is read from the URL, so the view function knows what data to plot.

The `comparisons.views.scatterplot()` function is responsible for converting the data of one comparison to a scatterplot graph. The MatPlotLib Python library is used to plot both types of graphs.The `Benchmark` objects are filtered from the database, as defined by the comparison. The two sets of `Benchmark` objects are intersected on the modelnames they contain.

If the resulting data set is empty, then an empty graph is shown to the user. Othewise, the data is converted a bit so it can be plotted in a graph. Two separate graphs (one for runtime, the other for memory usage) are plotted into one figure by using the `add_subplot()` method. For each graph, the following actions are performed:

- The `total_time` or `memory_RSS` values are selected from the benchmark sets.
- The datapoints are masked either red or blue.
- A linear function is plotted to show where the values for both series are equal.
- The data is plotted using the `scatter()` function.
- The axes are labeled and colored.

The same list is executed for the `memory_RSS` plot. Finally, the figure is exported in PNG format.

4.4.2 ModelComparison

The actions performed to create a graph for comparison of models are nearly the same. The `comparisons.views.graph_model()` method will be explained here:

- Data is extracted from the database, specified by the `ModelComparison` object.
- All models in the database are aggregated by name.

- For each model in the dataset, a line is plotted of the selected data type over all available dates.
- Finally, the graph legend is printed and the plots are labeled.

4.5 User Management

Django comes with a user authentication system to handle user accounts, groups and permissions. Only the first feature is used for BeAT. The standard Django model `django.contrib.auth.models.User` is used to model a user in the database. No additional data about a users is stored.

In the `base.html` template, the {`% if user-is_authenticated %`} block is used to check whether a user is authorized to view links to personal pages. The `@login_required` decorator is used in views to check whether a user is authorized to access a specific view. This used for all of the comparison and jobs pages and the benchmark table page. If a user is not authorized to view a certain page, he will be redirected to the login page. In `settings.py`, the location of the login page is defined as `LOGIN_URL = '/login/'`. This URL is mapped in `urls.py` to a predefined Django view called `django.contrib.auth.views.login` that uses the custom template `login.html`. After a succesful login, the user is redirected back to the page he tried to acces. More information about django's authorization mechanisms can be found in the django doucmentation [15].

4.6 PBS script generation

The code that handles the generation of PBS scripts and the saving of a the options of a specific job specified on the website is all located in the app `beat/jobs`. This app defines a single table in the database that describes the arguments that can be passed to the script generation code. The scripts are generated using Django templates, located in `beat/jobs/templates`. When a script is to be generated the template `main.tpl` is loaded from this folder and supplied with the values of all relevant variables, and Django will then use its builtin template handling to produce the script. The template language is quite simple: `{% include "somefile.tpl" %}` will insert the contents of `somefile.tpl` at that point, and `{{ variable_name }}` will be replaced with the value of the variable with that name. Django supports more instructions, and can handle quite complex templates, but none of those are in use in the PBS script templates at this time. For that reason they will not be discussed at this point, for more information on Django-templates visit the Django website. After Django has processed the templates it produces a single long string containing the resulting text. When a single pbs script is requested, the app will write that string to a virtual file and then serve it to the user. When the user requested a suite, all the resulting strings are written to virtual files, which are then added to a (virtual) gzipped tar-file, which is then served to the user. By using virtual files the server never needs to write to the harddrive to generate these scripts and

they are small enough to fit in memory (a few kilobytes per file, which should be freed by the garbage collector after the download is complete). The code that produces a suite and the code to create a single script are both in `jobs.py`. The code that creates a virtual tar-file is located in `jobs_fileserv.py`.

4.7 Tools Module

The Tools package contains some helpful modules with mostly generic methods that are explained below.

export_csv - contains a function `export()` that takes a Django queryset and outputs a comma-separated values (csv) document with column names and all rows of that queryset. Optionally, a title for the document and a list of column names that should be excluded from the document can be given as arguments. We use this for exporting data from the benchmark table. This module uses the python CSV library [4], as documented in the django docoumentation [10].

feeds - Django provides a high-level syndication-feed-generating framework that makes creating RSS and AtomFeed easy [14]. This module can be extended by creating additional subclasses of `django.contrib.syndication.views.Feed`. Each class needs the `title`, `link` and `description` attributes and the methods `items()`, `item_title()` and `item_description()`. Furthermore, each `Feed` class needs to be mapped in `urls.py`.

graph - the `export()` method in this module is used for exporting graphs as a `HttpResponse`. The user can indicate the file format to which the graph is exported (default is `png`, other options are `pdf`, `ps`, `eps` and `svg`). This method requires a MatPlotLib Canvas object as an argument.

hash - a simple method to create a SHA1 hash out of a string. This is used to create the authorization key for sharing comparisons.

intersect - the `intersect()` method takes two lists and returns the intersection of this list, by converting the lists to sets, applying the `and` operator and finally converting it back to a list. In BeAT, this is used to find the correct set of benchmarks for a comparison.

logsave - the code used to interact with a local git repository. This code has been seperated from the filereader code so that it may be completely optional, preventing issues for machines where dulwich is not available. This file provides a function to write and read logs, as well as initialization code. A function to get the latest tree in the git repository containing the logs is also provided. The most important advantage of this code is that it does not generate raw text files, but instead commits directly to the git repository. This avoids two issues: file size and slowness of a filesystem for folders that contain a large amount of files.

regex_tester - This script provides a simple function to test a regular expression as would be done in the filereader code. It is used in conjunction with some javascript on the tool upload page (see section 3.2.3).

git interface - The GitInterface is a small shell around the gitpython module, allowing BeAT to interact with git. It contains two functions that will be discussed: `match_from_tag(given_tag)` and `get_matching_item(hash)`. `match_from_tag(given_tag)` will find the revision related to the given tag. It takes the `TagReference.list_items(repo)` from gitpython, it will iterate over this list and return the requested commit. `get_matching_item(hash)` does the same but it uses `iter_commits()` instead of `TagReference.list_items(repo)`, searching for the revision based on hash instead of tag name. The returned item from these functions can be used to get the commit date.

Chapter 5

Future work

We have several recommendations for features that can be added to the system. These are features that could not be added during the project because of time constraints. We will give a short description of each feature, an explanation of the feature and our estimate of the implementation difficulty. These suggestions are sorted by their priority. The overview of these features is in table 5.1 and continues in table 5.2.

Table 5.1: Future work

Name	Difficulty	Explanation
FLOT	medium	At the moment our graphs are generated using Mathplotlib. Graphs that are generated using this library are static and do not allow for interactions. In a scatterplot it is not possible to add labels because these would take up too much space. Instead, a solution with `flot` and a mouseover when hovering over a point was proposed. Part of this feature was implemented but it was not realised because there were some issues with client-side transformations of (time-based) data.
Easier addition of algorithm-tools	medium	AlgorithTools need to be added to the database in order to recognise the logs. A good solution should be created that does never add invalid data, but is not too unfriendly for the users. The current solution is safe, but might be a nuisance depending on the amount ltsmin revisions used to create logs.
Filter on git revision	medium	The current benchmark table only allows the user to filter on the date of a tool's creation. It does not enable the user to filter on the hash of the revision of the used ltsmin version.
Filter on extra values	medium	Filter benchmark instances by the value of their extra values.
Multiple hardware support	medium/hard	BeAT does not use all the information it has with regards to the hardware used to create runs. The impact of differences in cpuś (for example: more cache, new instructions) might have a large impact on runtime. At the moment BeAT is not able to generate graphs using this data.
Options column	medium	The benchmark table does not show the options used to create each benchmark instance. This is confusing since multiple instances which only vary in options look exactly alike.

Table 5.2: Future work (continued)

Name	Difficulty	Explanation
Better user management	medium	Currently all benchmarks are available for registered users. Visitors can only see benchmarks if they have the permalink. Complete groups support could be added. This could hide private branches their benchmark results or make all results for published versions public.
Better CSV export	easy	The current CSV export exports all columns. This is easy to fix with console tools, or by ignoring the extra data. Selecting the columns you wish to export (and saving this setting) would be a nice addition.
Typed extra vlues	hard	Currently the extra values are stored as a string. Support for typing would add some validation, make processing a bit easier and is considered to be a cleaner solution.
Tools from other repositories	easy/hard	Enable BeAT to import logs created by other tools. Depending on their log structure and structure (frontend + backend, or more parts) adding these tools is relatively easy or could get very hard.

Appendix A

Glossary

A.1 BeAT Glossary

Name	Description
Algorithm	The state space exploration method used to analyze the models.
AlgorithmTool	The combination of an algorithm and a tool, which is represented by an ltsmin executable.
Benchmark	The data that results from running an AlgorithmTool once (usually on specified hardware, with certain options and a particular model).
Comparison	The section of the BeAT application that concerns comparing two AlgorithmTools, resulting in a scatterplot of results.
Filters	The section of the BeAT application that allows filtering within the table. Does not refer to django filters (the table filtering is performed in javascript).
Log	The output generated by the AlgorithmTool. Note that this refers to an abstract level, not to a particular file.
Model	A description of a state space in a certain language.
ModelComparison	The section of the BeAT application that allows the user to see information about multiple models over time for a particular AlgorithmTool.
Run	An execution of a Benchmark. May also refer to the results of such, as a synonym to Log
Tool	The appropriate language module to read models.
(AlgorithmTool) Version	The exact version string of the AlgorithmTool, containing the first 6 characters of the has of the git revision it was built from

A.2 Django Glossary

Term	Definition
admin	The Django administration is an automatically-generated admin site which runs directly on top of the database. It can be used to browse and modify specific data. For BeAT, the admin is used to create new users. Views for editing or browsing data in the admin can be manually adjusted in admin.py.
application	All the modules of a Django website are called applications. BeAT contains three applications: benchmarks (stores all result data), comparisons (handles graphing) and jobs (to generate job scripts). Within BeAT, Tools is a module that only contains some helpful methods, such as a simple hash function and data export functions. Tools does not need models.py or views.py.
decorator	Special Django methods to do a preliminary check before a view function is executed. For BeAT, we use decorators in for login authentication and caching.
field	attribute on a model; a given field usually maps directly to a single database column.
model	Models store your application's data. They represent the database structure of a project. Each model is represented by a class that subclasses django.db.models.Model. Each model has a number of class variables, each of which represents a database field in the model.
project	A Python package i.e. a directory of code that contains all the settings for an instance of Django. This would include database configuration, Django-specific options and application-specific settings.
queryset	An object representing some set of rows to be fetched from the database. No database activity actually occurs until you do something to evaluate the queryset.
template	A chunk of text that acts as formatting for representing data. A template helps to abstract the presentation of data from the data itself. It contains the HTML code to which Django can pass context data.
url	The file urls.py defines how URLs are mapped to views. Django runs through each pattern and stops at the first item matching the requested URL. Each pattern contains a regular expression in which named groups can be used, to capture specific bits of the URL.
view	A function responsible for rendering a page. In the Model-View-Controller pattern, a Django view is about equivalent to the controller.

A.3 Software

Name	Description
Dulwich	A python module that allows the programmer full access to any (locally stored) git repository.
Django	A framework for webdevelopment in python.
Gitpython	A python module that provides basic operations on git repositories.
Matplotlib	A graphing module for python.

Appendix B

Header

The header looks as follows:
BEGIN OF HEADER
Nodename: hostname of the executing node
Hardware-name: the name of the hardware platform (eg x86_64)
OS: the OS name (eg GNU/Linux)
Kernel-name: the name of the kernel (eg Linux)
Kernel-release: the release name of the kernel (eg 2.6.26-2-amd64)
Kernel-version: the version of the release (eg #1 SMP Thu Feb 11 00:59:32 UTC 2010)
Hardware-platform: name of the hardware platform
Processor: name of the processor
Memory-total: the amount of memory available, in kilobytes 1028804
DateTime: the time the run was started, in the format "year month day hour minute second microsecond"
ToolVersion: the version of ltsmin used to create this log (eg ltsmin-1.5-20-g6d5d0c)
Call: the way the tool was called to generate this log. This is usually bash. For example:
memtime lpo2lts-grey -c ./ltsmin-models/mucrl/f42.tbf ./ltsmin-models/mucrl/t42.dir
memtime tracks time and memory use. lpo2lts-grey is the AlgorithmTool executable, followed by options (here -c) and
additional arguments (it is assumed that the first of these is the model).
END OF HEADER

Appendix C

Filereader Options

This information can also be obtained by using `python filereader.py --help`.
Usage: filereader.py [options]
Options:

`-h, --help`	show this help message and exit
`--silent`	Print only dangerous errors (like database integrity warnings).
`-q, --quiet`	Print default amount of messages; one per item under normal circumstances.
`-v, --verbose`	Print additional (helpful) information, such as a summary of added data.
`--noisy`	Print as much as possible. Useful for debugging this script, not intended for other use.
`--override`	Override existing data.
`--dulwich`	Use this switch if logs are to be saved to a local git repository.

Bibliography

[1] Brian Cherne. hoverintent jquery plug-in. http://cherne.net/brian/resources/jquery.hoverIntent.html.

[2] D. Crockford. The application/json media type for JavaScript Object Notation (JSON). RFC 4627 (informational). Published by the Internet Engineering Task Force.

[3] Fee Software Foundation. GNU core utilities frequently asked questions. http://www.gnu.org/software/coreutils/faq/coreutils-faq.html#Argument-list-too-long.

[4] Python Software Foundation. csv - CSV file reading and writing. http://docs.python.org/release/2.6/library/csv.html.

[5] Python Software Foundation. json - JSON encoder and decoder. http://docs.python.org/release/2.6/library/json.html.

[6] Python Software Foundation. optparse - more powerful command line option parser. http://docs.python.org/release/2.6/library/optparse.html.

[7] The jQuery Project. jQuery: the write less, do more, JavaScript library. http://www.jquery.com.

[8] Django Project. Django documentation. http://docs.djangoproject.com/en/1.2/.

[9] Django Project. The django template language. http://docs.djangoproject.com/en/1.2/topics/templates/.

[10] Django Project. Outputting CSV with django. http://docs.djangoproject.com/en/1.2/howto/outputting-csv/.

[11] Django Project. Queryset api reference - filter keyword arguments. http://docs.djangoproject.com/en/1.2/ref/models/querysets/#filter-kwargs.

[12] Django Project. Queryset api reference - order by. http://docs.djangoproject.com/en/1.2/ref/models/querysets/#order-by-fields.

[13] Django Project. Queryset api reference - values fields. http://docs.djangoproject.com/en/1.2/ref/models/querysets/#values-fields.

[14] Django Project. The syndication feed framework. http://docs.djangoproject.com/en/1.2/ref/contrib/syndication/.

[15] Django Project. User authentication in django. http://docs.djangoproject.com/en/1.2/topics/auth/.

[16] Django Project. Working with forms. http://docs.djangoproject.com/en/-1.2/topics/forms/#using-a-form-in-a-view.

[17] James Smith. Blog of james smith. http://loopj.com.

[18] Ars Technica. Image of OS X finder. http://media.arstechnica.com/-images/tiger/spotlight-find-metal.jpg.