Ruprecht-Karls Universität Heidelberg
Institute of Computer Science
Research Group Parallel and Distributed Systems

Internship Report

Community Platform for Parabench

Name: Dennis Runz, Christian Seyda
Betreuer: Olga Mordvinova, Julian M. Kunkel
Sommersemester 10
1. April - 1. September 2010

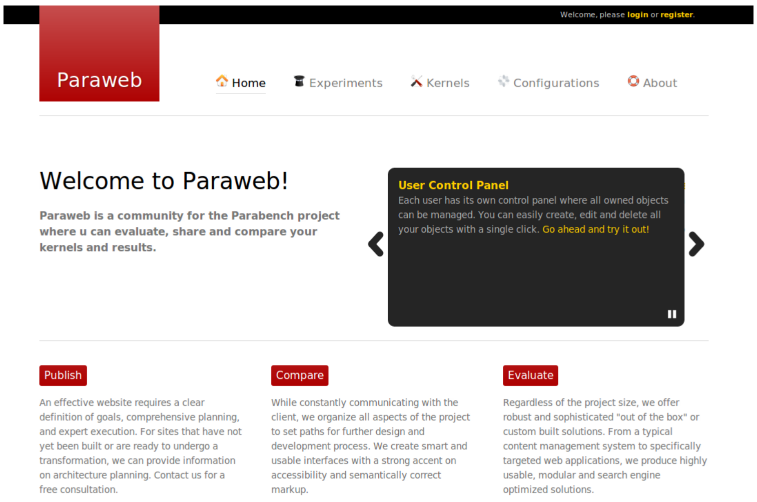
Introduction

THE main goal of this practical was the development of a community website for Parabench [1,2], where users can share their IO patterns, their benchmark results including visualizations and a description of their testing platform. To accomplish this, we choosed Django [3] as our webframework to build upon, Highcharts [4] as visualization tool and wrote a search interface based on machintags.

In the following we will introduce the basics of Django, Highcharts, how we produce plots with both of them and how our search-backend is working.

Motivation

In addition to our formal practical about parabench, there was still the intention to create a community around parabench, to provide real up to date patterns for the public demand and to focus the development of parabench in a direction, where the community needs it. Furthermore to lower the stress to gain good looking visualizations out of the benchmark resulted data.



Welcome to Paraweb!

Paraweb is a community for the Parabench project where u can evaluate, share and compare your kernels and results.

User Control Panel
Each user has its own control panel where all owned objects can be managed. You can easily create, edit and delete all your objects with a single click. Go ahead and try it out!

An effective website requires a clear definition of goals, comprehensive planning, and expert execution. For sites that have not yet been built or are ready to undergo a transformation, we can provide information on architecture planning. Contact us for a free consultation.

While constantly communicating with the client, we organize all aspects of the project to set paths for further design and development process. We create smart and usable interfaces with a strong accent on accessibility and semantically correct markup.

Regardless of the project size, we offer robust and sophisticated "out of the box" or custom built solutions. From a typical content management system to specifically targeted web applications, we produce highly usable, modular and search engine optimized solutions.

Figure 1: Paraweb preview

Contents

Chapter 1

Django

1.1 Introduction

Django [3] is a web framework written python. It offers many great features, is under active development and open source (BSD license). As other frameworks (CakePHP, Grails, Ruby on Rails, to mention a few) surly offer the same features—we have choosen Django because of its great documentation and community, and we are are quite familiar with python.
For a feature-list, please look at `http://docs.djangoproject.com/en/1.2/`. Worth mentioning features are

- the automatic und customizable admin site (`http://docs.djangoproject.com/en/1.2/ref/contrib/admin/`),
- build-in user management (`http://docs.djangoproject.com/en/1.2/topics/auth/`), which we have extended by some fields and integrated an user-registration, including the need of activating the user by pressing an email-send link,
- the comments-system (`http://docs.djangoproject.com/en/1.2/ref/contrib/comments/`),
- the paginator (`http://docs.djangoproject.com/en/1.2/topics/pagination/`) for splitting a page containing many elements into subpages containing a subset of elements and links to the other subpages,
- the forms-API (`http://docs.djangoproject.com/en/1.2/ref/forms/api/`) for HTML-forms creation and validating, and
- the caching-framework (`http://docs.djangoproject.com/en/1.2/topics/cache/`) for reducing serverload and increasing response-time.

Django has a clear MVC-design—please don't frighten as the names in Django are a little different[1] from what you are familiar with:

Model: models.py
"A model is the single, definitive source of data about your data. It contains the essential fields and behaviours of the data you're storing. Generally, each model maps to a single database table." (taken from `http://docs.djangoproject.com/en/1.2/topics/db/models/`)

View: are the templates (introduced at 1.3)

Controller: view.py
"A view function, or view for short, is simply a Python function that takes a Web request and returns a Web response. This response can be the HTML contents of a Web page, or a redirect, or a 404 error, or an XML document, or an image . . . " (taken from `http://docs.djangoproject.com/en/1.2/topics/http/views/`)

URLconf
"To design URLs for an app, you create a Python module informally called a URLconf (URL configuration). This module is pure Python code and is a simple mapping between URL patterns (as simple regular expressions) to Python callback functions (your views)." (taken from `http://docs.djangoproject.com/en/1.2/topics/http/urls/`)

Furthermore, we have

settings.py, where global settings for the project are stored, such as the database-configuration or the template path (`http://docs.djangoproject.com/en/1.2/ref/settings/`), and

manage.py, which is Django's commandline-tool for administrative tasks (`http://docs.djangoproject.com/en/1.2/ref/django-admin/`).

1.2 Setup

Django

Installation of Django is very simple: set up your database of choice and either install it from your operating system's repositories or get it from svn (`http://docs.djangoproject.com/en/dev/intro/install/`).

Due to the excellent documentation and beginner tutorial, we won't describe the initial setup of a django project, as we would just repeat the tutorial `http://docs.djangoproject.com/en/dev/intro/tutorial01/`.

[1]`http://docs.djangoproject.com/en/dev/faq/general/#mtv`

In order to not depend on the documentation on the internet, there is the possibility to get it for offline reading: `http://docs.djangoproject.com/en/1.2/internals/documentation/#internals-documentation`.

When initially creating the database scheme with `python manage.py syncdb` a superuser will be created interactively. But since we have an own auth backend for PortalUser, the authentification will not work yet for the admin panel. To fix this, an entry in the database table `core_portaluser` needs to be added manually (`id=1, timezone='Europe/Berlin'`).

Searchbackend

Our search is based on Haystack `http://haystacksearch.org/` providing an API to some searchbackends focusing on different work environments.

1. Install Haystack:

   ```
   sudo apt-get install python_setuptools python_pip
   sudo pip install haystack
   ```

2. Install Whoosh search engine (used for dev purpose):

   ```
   sudo pip install whoosh
   sudo mkdir -p /var/whoosh/paraweb_index
   sudo chmod 777 -R /var/whoosh/
   ```

   Alternatives and setup instructions can be found at `http://docs.haystacksearch.org/dev/installing_search_engines.html`.

3. Add haystack settings to settings.py:

   ```
   # Haystack Search Settings:
   HAYSTACK_SITECONF = 'paraweb.search_sites'
   HAYSTACK_SEARCH_ENGINE = 'whoosh'
   HAYSTACK_WHOOSH_PATH = '/var/whoosh/paraweb_index'
   ```

4. Create initial search indexes: `python manage.py rebuild_index`

5. Later you can update the indexes with: `python manage.py update_index`

Email

In order to send emails, you have to set the appropriate settings to your `settings.py` (`http://docs.djangoproject.com/en/1.2/topics/email/`).

For testing purposes, add

```
EMAIL_HOST = 'localhost'
EMAIL_PORT = 1025
```

and start with `python -m smtpd -n -c DebuggingServer localhost:1025` a test email-server at your localhost at port 1025 (`http://docs.djangoproject.com/en/1.2/topics/email/#testing-e-mail-sending`).

We will now focus on a few things that will be needed later and at the end we give some examples of where you can find some Django-features in our code.

1.3 Templates

Normally a website consists of a layout/design that hardly changes, only the content and minor things should alter. So, in early days, you had your many .html-files which all had to be modified in order to get a corporate-design—there is CSS for the visuals and layout, but think about menus or some header-definitions. In Django you just change your main layout-file to spread changes across your whole web-appearance. A very basic main-file (call it `main.html`) with a menu-sidebar and a content-area could look like:

Listing 1.1: Django template

```
<head><link rel="stylesheet" href="style.css" />
    <title>{% block title %}My amazing site{% endblock %}<
       /title>
</head>
<body>
    <div id="sidebar">
    {% block sidebar %}
    <ul>
        <li><a href="/">Home</a></li>
        <li><a href="/blog/">Blog</a></li>
    </ul>
    {% endblock %}
    </div>
    <div id="content">
        {% block content %}{% endblock %}
    </div>
</body></html>
```

Now comes Django's template inheritance for easy and fast site maintaining; say we have a page (`stuff.html`) containing some real content:

```
{% extends "main.html" %}
{% block title %}Cool Stuff{% endblock %}
{% block content %}
some content here.
{% endblock %}
```

With the `extend`-keyword, we say Django have to grab `main.html` and replace the right blocks from it with my new wrote in `stuff.html`.

You can also insert variables with `{{ variable }}` or loop through a list with

Listing 1.2: Django template loop

```
{% for entry in my_list %}
    <h2>{{ entry.title }}</h2>
    <p>{{ entry.body }}</p>
{% endfor %}
```

given by the view-function.

For paraweb we use a two-layer template-inheritance:

- global `base.html`
- section based template inheriting `base.html` (for example `r_base.html`)
- task based template inheriting `r_base.html` (for example `r_edit.html`)

This way, we can offer good working breadcrumbs too.

1.4 Deployment

Djangos documentation even covers the deployment of a website (`http://docs.djangoproject.com/en/1.2/howto/deployment/`). The recommended way is

- `mod_wsgi` (`http://docs.djangoproject.com/en/1.2/howto/deployment/modwsgi/`), but the documentation also covers
- `mod_python` (`http://docs.djangoproject.com/en/1.2/howto/deployment/modpython/`) and
- `FastCGI, SCGI or AJP` (`http://docs.djangoproject.com/en/1.2/howto/deployment/fastcgi/`).

For development, we just let Django handle static files (`http://docs.djangoproject.com/en/1.2/howto/static-files/`).

But this is not recommended for a productive website, though Apache, lighttp, . . . are specialized on this (`http://docs.djangoproject.com/en/1.2/howto/deployment/modpython/#serving-media-files`).

Chapter 2

Software Structure and Object Organization

Django projects are organized as collection of apps and the following introduces the apps that were used and developed to realize the community for Parabench.

2.1 Core

The core app manages core functionality and data models such as kernels or experiments. It provides functionality to create, manage and view those objects and their relations.

Kernel: Kernels represent I/O access patterns formed in the Parabench Programming Language (PPL) where each instance is associated with a ppl file that can be downloaded by users. Every kernel is owned by one user and can be tagged and classified by the owner. Additional meta attributes such as revision, which is increased every time a kernel is modified, and a published state can also be set. Kernels have a special type (`KernelType`) where the set of available types can be defined in the admin panel. This can be MPI, POSIX, or even other types since Parabench allows to define modules that can even be MPI send commands instead of I/O specific commands.

Configuration: There are two configuration types, `HardwareConfig` and `SoftwareConfig`. The former describes the pure hardware informations about the tested hardware and the latter all informations about the software setup used which can be file system, operating system and more. Configurations have static and dynamic fields, where the static ones include the number of nodes, number of data and metadata servers and so on. This is required since they are referenced within several other apps in order to allow statistical informations such as a cluster map with file system and client node mapping. Dynamic fields are represented by `Setups`. Like kernels, configurations have an owner.

Setups: Setups allow to dynamically define configuration properties without requiring changes in python code. A `SetupType` defines the type – whether it is a software or a hardware setup – and the name together with a slug of a type. Examples for types can be *vendor*, *file system*, *operating system* and so on. For each type, values can be defined which are represented by `SetupValue`. All setup relevant data can be managed in the admin panel and are available for user choices immediately.

Run: Runs represent the actual execution of a benchmark run on a computer system. Each run depends on a kernel and two configurations that can be set independently for hardware and software. Each run is evaluated by the plotting functionality independently which allows to have a quick overview over the results of a single run. Also runs are owned by a user.

Experiment: Experiments are the complete set of runs that have been executed. In fact, experiments can consist of several runs that have been executed on the same and/or different kernels as well as the same and/or different configurations, there is no limitation for combinations. Evaluation of experiments is done on all available runs within an experiment. Depending on the used kernel and configurations, different plots are shown with a different focus. This is described in a more detailed way in chapter 3. Together with kernels, configurations and runs, also experiments have a specific owner.

2.2 Tagging

The complete user space object classification and taxonomy feature is based on the tagging app which is based on django-tagging `http://code.google.com/p/django-tagging/` but has been modified in several places in order to serve required features. The original and official django-tagging app doesn't provide triple tag support. Instead, only single keywords can be used for tagging, which doesn't allow any classification of heterogeneous objects. A fork of this project could be found on Launchpad `https://code.launchpad.net/~gregor-muellegger/django-tagging/machinetags/`, which has been used as basis for further adjustments.

To ease entering of tags and classifications, an autocomplete feature has been implemented that is based on `http://code.google.com/p/django-tagging-autocomplete/`. When entering text, strings such as `:` and `=` trigger predicates and values from the tag database and offer a preselection of matching strings to the already entered string part. This has also been developed in order to increase classification quality in the object universe. Since users may invent new or type slightly different words which all have the same meaning and could therefore be treated as the same category—respectively namespace, predicate or value—autocompletion increases the probability to maintain a more consistent taxonomy and likewise may provide a lower maintenance effort for administrators.

2.3 Taxonomy

Additional features and helper routines for the tagging app are implemented in the taxonomy app in order to prevent too close coupling. This was necessary because the tagging fork was not yet official and is still pending to be integrated in the django-tagging app. In case of later merging with the latest official revision of this app, a separate app has been written that provides additional functionality that is:

Taxonomy View: The taxonomy view shows a universe of classified objects and is structured in three levels. Those levels are directly derived from the triple tag concept which is described in section 4.2.

Cloud Utils: Helper functions that are necessary to build the taxonomy tag clusters and clouds.

2.4 Admin Panel

The administration panel is the central management place for administrators and moderators. All objects that live in django can be managed, which also means users can be created, deleted and privileges can be edited in a fine grain level. It is also possible to create user groups which are for example able to only manage kernels. That means in general, object permission can be set on app level, object level and operation level. The latter includes creating, deleting and editing.

2.5 User Control Panel

In order to provide a seamless user experience when working with objects, a user control panel was developed. It allows to create, edit and delete core objects which are kernels, experiments, runs and configurations. The user only sees its own objects and is able to manage his own results. Certain informations which are not meant to be edited by users are hidden, which wouldn't easily be possible in the admin panel.

2.6 News

The front page is able to show basic news posts, for example to make announcements about updates or important events to the community.

NewsPost: Represents a post whose fields include a title, a short abstract which is shown on the front page that doesn't allow HTML code and a body text which is full HTML capable and is only shown in the detail view.

2.7 Rating

To provide a basic kind of quality management, next to the possibility to comment on significant objects—such as kernels and experiments—objects can also be rated in a five star system. This gives a quick feedback to both the object owner and the community to judge about an object's quality, may it be the evaluation, the richness and quality of the describing text or the relevance of the experiment or kernel itself. Object rating is handled in the rating app for which rabid ratings `http://msteigerwalt.com/widgets/ratings/v1.5/` is used.

2.8 Slider

Another feature to increase usability of the website is provided by the slider, also placed on the front page. It can be used to announce and introduce new and important features of the website in a prominent and eye catching way.

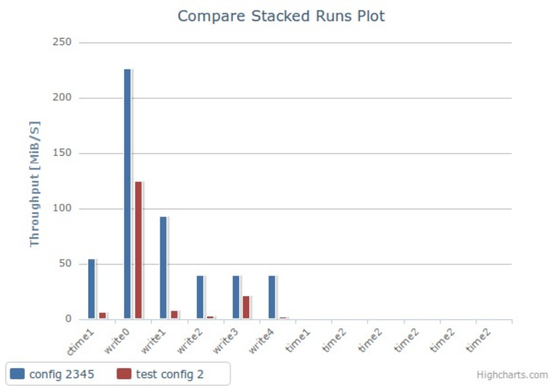
Slide: The model which represents a slide consisting of a title, a HTML capable body text and some meta informations. An admin for slides is available in the admin panel.

Chapter 3

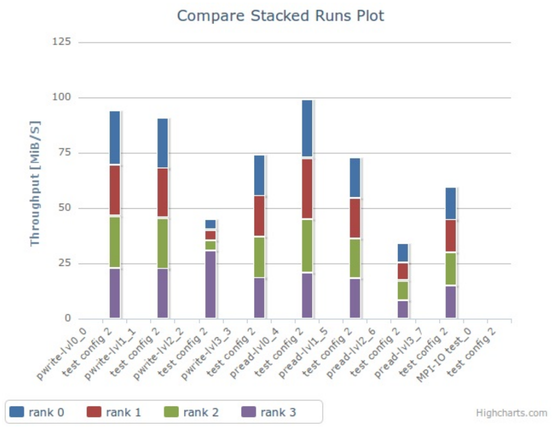
Highcharts

3.1 Introduction

We choose Highcharts [4] as our plotting library, because it is written in pure JS enabling us to even render big plots in modern browsers. It offers many different plot types and an easy interface, is visual appealing and uses complete svg support, which will be used for export to PNG, JPEG, PDF and SVG (see 3.2.4).



(a) basic column



(b) stacked column

Figure 3.1: Plot examples

As said above, usage is generally very easy and described well at `http://www.highcharts.com/documentation/how-to-use`. A short description:

1. Include the right .js in the header of the website, that should show a plot. Highcharts uses jQuery [5] or Mootools for common tasks—as we use jQuery for processing the result files, we let Highchats use jQuery too.

Listing 3.1: Highcharts Initialisation

```
<script src="http://ajax.googleapis.com/ajax/libs/
   jquery/1.3.2/jquery.min.js" type="text/javascript">
   </script>
<script src="/highcharts/highcharts.js" type="text/
   javascript"></script>
<script src="/highcharts/exporting.js" type="text/
   javascript"></script>
<script src="/highcharts/plots.src.js" type="text/
   javascript"></script>
```

2. Create a JSON[1] structure containing the options for our plot to use.

Listing 3.2: Highcharts JSON

```
var options = {
    chart: {
        renderTo: 'chart-container',
        defaultSeriesType: 'bar'
    },
```

some more options here ...

```
    series: [{
        name: 'Jane',
        data: [1, 0, 4]
    }]
};
```

3. Let Highcharts render the plot. `var chart = new Highcharts.Chart(options);`

3.2 How we use it

We put all plot-related functions into a single .js-file: `plots.src.js`. Besides the main plot-generating functions, there are a few helper ones:

- `function Plot_options(container)`:
  the general chart options,
- `function calculate_offset(options)`:
  the logic to set up the legend.

[1]JavaScriptObjectNotation: `http://www.json.org/`

- `function round_values(series)`:
  for rounding all plot values up to the second decimal place.
- `function check_undefined(series)`:
  for clearing all undefined values (see 3.2.2).

Right now, the steps for creating a plot are:

1. A Django view collects the needed runs—static by given address or dynamic by given option array—and gives those, together with a plot-template, to the template system.
2. The template system
   - merges our different html-files,
   - writes the run-addresses in an array and
   - send the resulting website to the client.
3. The browser fetches the run-xmls with jQuery/AJAX and processes the files with JavaScript:
   - reading result-values, their rank, id and name,
   - saving those to arrays, in order to create rank-, id-, kernel-, . . . aware plots.
   - Those arrays will be written to our options-object and
   - we make some finishing calculations:
     - if needed, merge and normalize some values.
     - sanity check for `undefined`-values.
     - get the right margin for the legend.
4. Then, let Highcharts do his magic.

Chart Options

Here are the general settings, which should apply to all plots on paraweb. Right now we have general settings without much modification. A short overview of the most changed settings:

- `options.title.text` defines the title of the chart.
- `options.plotOptions.column.stacking` defines the stacking behaviour.
- `options.series[i].data[]` contains our values.
  `options.series[i].name` is the name of the dataset.
- `options.xAxis.categories[]` contains the labels for the x-axis.
  `options.yAxis.title.text` defines the label of the y-axis.

For a comprehensive list of all possible options, please refer `http://www.highcharts.com/ref`.

Calculate Offset

We have to calculate an offset for the legend, because in general, there are too many labels for Highcharts standard floating behaviour. So we take the legend and set it beneath the plot. To be able to estimate the offset, we count

- the letters in the legend,
- the maximum length of a label,
- the number of categories and
- the maximum length of a series name.

Together with some empiric constants we compute how much lines the legend will need, and having the count of those lines, we finally have the height of our legend.

3.2.1 Processing

As for now, the processing part is not really generic like the other parts above. We mainly have one template for every different plot (see 3.2.3). Future work will focus on writing a generic processing backend, so that user can individually build and save their plot-generics. So in this part, we will mainly describe the repeating parts and some approaches for advanced processing.

The processing flow consists of:

run-addresses: the template-system will write all given run-addresses in `var runs[]`. For some tasks, it can be useful to store their names in `var run_names[]`:

```
var runs = [
    {% for run in selected_runs %}
        "{{ run.get_absolute_url }}download/"
        {% if not forloop.last %},{% endif %}
    {% endfor %}];
```

chart options: `new Plot_options("{{ container }}")` includes our general options-file. After that, you should alter some values, for example the charts-name or type.

loading and processing: the first statement just tells, that we wait with processing until the page and pictures are fully loaded. We really do not want to crash the browser while loading the page. Second half just says to commit synchronous AJAX-calls for every address in our `runs[]`. Synchronous, because we do not want different threads/processes to write concurrent to the same arrays. In some cases we even have 2-dimensional arrays (with different kernels or configurations).

Listing 3.3: jQuery AJAX

```
jQuery(document).ready(function() {
    for (var i = 0; i < runs.length; i++){
        jQuery.ajax({
            url: runs[i],
            async: false,
            success: function(data) {
                var helperarray = {
                    data: []
                };
```

Next part will be to declare some helper arrays—for logging different ranks, ids, names, . . . —and to write in those the extracted data. As written above, we use jQuery for XML-processing. This makes it easy to filter the relevant information.

Listing 3.4: jQuery XML

```
jQuery(data).find("Eventlist").each(function() {
    var $eventlist = $j(this);
    var type = $eventlist.attr("type");
    if (type == "CoreTime") {
        jQuery(data).find("Event").each(function() {
            var $event = $j(this);
            var rank = $event.attr("rank");
            var id = $event.attr("id");
            var label = $event.attr("name");
            var $tp = $event.find("Throughput");
            var avg = $tp.attr("avg");
            var value = parseFloat(avg)/1024/1024;
            if (isNaN(value)){ value = 0; }
            helperarray.data.push(value);
        });
    options.series.push(series);
    }
});
```

jQuery traverses the file, until it finds an `Event`-leaf, gives us this leaf and we can grab the needed attributes. Then we just have to bring the float-value into the right dimension (MB) and check, if it is still an usable value—JavaScript can't handle values too small. We add the value to our `helperarray[]` and finally add the values in the right order into our options-object.

The right order depends on the type of plot used. The best resource here is the great demo gallery, together with its code examples: `http://www.highcharts.com/demo/`.

postprocessing: include our `calculate_offset`-and `round_values`-routines, change the size of the DOM[2]-object containing our plot and the offset in the plot and at the end: plot it.

```
check_undefined(options.series);
round_values(options.series);
calculate_offset(options);

var chart = new Highcharts.Chart(options);
};//ready
```

3.2.2 Additional processing

`undefined`-Check

We have some situations, where we have undeclared variables. We can check this in JavaScript with the value- and type-check `===`. So we check for `undefined` with `value === "undefined"`. Avoid comparing with `==`! If the value is `null` we get for `value == "undefined"` as result `true`.

Listing 3.5: undefined-check

```
for (var i=0; i<options.series.length; i++){
  for (var j=0; j<options.series[i].data.length; j++){
    if (typeof options.series[i].data[j] === "undefined"){
      options.series[i].data[j] = 0;
    }
  }
}
```

Tow nested loops for all data-“points” and a comparison.

`rank`-awarness

For stacking or dividing by ranks, we have in the easiest case an array `var ranks = []` where we keep all found ranks. In the processing-part, we check if the rank exists.

Listing 3.6: Processing: check rank

```
var rank_exists = false;
for (var r_count=0; r_count < ranks.length; r_count++){
```

[2]Document Object Model: `http://en.wikipedia.org/wiki/Document_Object_Model`

```
        if (ranks[r_count] == rank){
            rank_exists = true;
            break;
        }
    }
```

Otherwise, we create a new `object` in our columns for the rank`name` and the `data`.

```
if (!rank_exists){
    options.series[rank] = {};
    options.series[rank].name = 'rank' + rank;
    options.series[rank].data = [];
    ranks.push(rank);
}
```

We continue with pushing the labels and category-names.

Merging columns

Merging columns is not that hard, but in combination of some processing-cases it can be hard to overlook. So we focus on a simple case as a start-point for more complex plots.

First, we need temporary array. While processing, we have to decide weather our value is initial or has to be added to some other values.

Listing 3.7: Highcharts merging columns

```
if (j==0){
    series.data.push(rounded);
} else {
    series.data[counter] += rounded;
}
```

This decision can be build of the iteration-count, already seen labels or ranks. After processing the actual run and before beginning with the new one, we normalize our values and save them to our options-array.

```
for (var k = 0; k < series.data.length; k++){
    series.data[k] = series.data[k] / runs[i].length;
}
options.series.push(series);
```

In more complex graphs you can save your normalized results in another temporary array, or have enough arrays for all iterations and spread the values afterwards.

3.2.3 Plotting Templates Overview

Here is an overview of our processing functions (located at `plots.src.js`) with a short description of what they are intended to do.

used in `r_detail.html` for processing one run

- `function graph(options, url)`
  Simple graph displays all found values as column.
  (x: labels, y: data)

- `function run_stacked_graph(options, url)`
  Graph displays ranks stacked by all found values as column.
  (x: ranks, y-stacked: values/labels)

used in `r_compare.html` for processing many runs with the same configuration and kernel

The runs are present in one array.

- `function extended_graph(options, url, names)`
  Graph displaying the values as columns by run, sorted by label.
  (x: runs by labels, y: values)

- `function stacked_graph(options, url)`
  Graph displaying the runs stacked by all found values as column.
  (x: runs, y-stacked: values/labels)

used in `e_detail.html` for processing many runs with the same kernel, even with different configurations

The runs are present in a three-dimensional array, for example structured like

```
[ //same kernel, but different configurations:
    [ // same kernel, same configurations:
    runA, runB, ... ],
    [ ... ], [ ... ]
]
```

- `function experiment_graph(options, url, names)`
  Graph displaying the different values as columns, ordered by configuration. A big version of `graph.html`
  (x: label, y: values)

- `function experiment_graph_label(options, url, names)`
  Graph displaying the average value of the different labels between runs and ranks.
  (x: labels, y: average values)

- `function experiment_graph_rank(options, url, names)`
  Graph displaying the average of all runs of the same configurations stacked by rank and ordered by label-name.
  Containing a small workaround for partitioning of the x-axes: due to the inability of Highcharts to generate ranked columns in a partitioned x-axes (will be released in the future), we move our columns on order to label one column for partitioning. (x: configurations by label, y-stacked: ranks)

3.2.4 Exporting Module

With Version 2.0 of Highcharts comes a new feature: printing and chart-export as PNG, JPEG, PDF and SVG. This is based on a server-side application for image-conversion. By default, Highcharts will send the SVG to `http://export.highcharts.com`, where it will be converted. A tutorial on how to run your own conversion-server can be found at `http://highcharts.com/documentation/how-to-use` at the very end.

Chapter 4

Search

As described before, the search functionality is based on Haystack, a framework that provides an abstraction layer for search in python. Each object that needs to be searchable is represented by a document that stores all search relevant informations. Acting like a template, all available object attributes and even attributes of related objects can be used. This is useful to provide search result in a wider context, e.g. when searching for `GPFS`, not only configurations with GPFS are listed in results, but also experiments that were executed on configurations that use GPFS as file system are listed. Listing 4.1 shows the document template for a kernel. For related attributes, django template tags are used. In this example, informations about related runs, experiments and configurations of a kernel are included in the document. To increase search hit ratio, different flavours or setup types are used. For example the plain text version, a version with a slugified type name, and the slug version that can be set by users.

Slugs are abbreviated strings for a more complex string. To give an example, a slug for `operating system` could be `os`. Furthermore, a slugified version of `operating system` would be `operating-system`.

Listing 4.1: Kernel index document

```
{{ object.name }}
{{ object.owner }}
{{ object.owner.org }}
{{ object.type }}
{{ object.description }}
{{ object.etags|join:", " }}

{% for run in object.run_set.all %}
    {{ run.name }}
    {{ run.description }}

    {% for experiment in run.experiment_set.all %}
        {{ experiment.name }}
```

```
            {{ experiment.description }}
            {{ experiment.etags|join:", " }}
        {% endfor %}

        {{ run.config.name }}
        {{ run.config.description }}
        {% for setup in run.config.setups.all %}
            {{ setup }}
            {{ setup.type.name|slugify }}={{ setup.value }}
            {{ setup.type.slug }}={{ setup.value }}
        {% endfor %}
    {% endfor %}
```

4.1 Search Engine

The recommended search engine for production usage is *solr*. It provides advanced search technologies and high performance with optionally setting up a search cluster for high load demands.

4.2 Triple Tags

We use a form of triple tags to store additional information useful for storing and searching. This kind of tag is standardized by W3C[1], was first devised in 2004 as an experiment in using del.icio.us as a collaborative geo-annotation database[2] and was adopted by flickr who used the term machine tags when referring to triple tags[3].

Basically, a triple tag is a normal tag with a particular structure:

`<namespace>:<predicate>=<value>`.

In order to allow a clean separation between objects, the triple tag concept is used in three different ways to allow fine grain object classification. Basically, the following three types are used:

Classification: Object classification – including kernels and experiments – uses all three fields of a triple tag to allow fine grain classifications without too high complexity. Objects can be classified by adding triple tags in the classification field of objects that support it. Object classifications are represented by tag clusters which are a list of tag clouds. Each namespace has a cloud of predicates whose entries appear bigger if there are more objects available that are classified with that particular

[1]called RDF: `http://www.w3c.org/RDF/`
[2]`http://www.brainoff.com/weblog/2004/11/05/124/`
[3]`http://www.flickr.com/groups/api/discuss/72157594497877875/`

namespace and predicate, this is the first level called object universe. The second level is represented by a tag cluster in a specific namespace where each tag cloud consists of predicate entries. The third and last level is a tag cloud in a specific namespace with a specific predicate and consists of value entries of different font sizes to indicate popularity.

Properties: For object properties, the namespace `property` is reserved, while predicates and values can be freely set by users. This allows to add additional informations to objects such as configurations. If there are no predefined fields available to appropriately describe an object – e.g. to provide high probability to regain objects from search – it is possible to add custom properties. A separate property field is available by objects that support it.

Tags: The most simple way to classify objects is to tag them. Tags use the reserved namespace `tag` and allow to set only the predicate of a triple tag. This coarse grain object classification is represented by tag clouds and is supported by kernels and experiments. Objects can be tagged using the tag field.

Examples are given below which show all three kinds of triple tags that are used. Queries can be done by also using wildcards in any part of the tags:

`hydrodynamics:weather=*` to get a list of objects that have been classified with hydrodynamics and weather but can have any value

`*:inmemory=*` to find any item with the predicate `inmemory` in any namespace and with any value

`behaviour=static` to find any object with a static behaviour as property, where the internal representation would be `property:behaviour=static`

`posix` to get a list of objects that have been tagged with `posix`. Internally this is represented as `tag:posix`

We hope this is a good way to find all relevant information searching for and with easy extensibility while maintaining low complexity for the end user.

Chapter 5

Future work

GENERALLY a popular website always evolves, especially it's a community website with demanding people. So we can't proclaim what will be important and what not. Saying that, we describe what in our view are some points needing to adjust/evolve. Because of the modular design, we split this in some parts too.

5.1 General

- add FAQ
- Add forum

5.2 Django

- Update to 1.2
- Verify Uploads as XML/PBL

5.3 Highcharts

- generalize processing to that point, where user can stick their own graph together
- test with big result files
- add possibility to process more values of the xml (min/max, coretime)

Listings

Bibliography

[1] Olga Mordvinova, Dennis Runz, Julian M. Kunkel, and Thomas Ludwig. I/O performance evaluation with Parabench – programmable I/O benchmark. *Procedia Computer Science*, 1(1):2119 – 2128, 2010. ICCS 2010.

[2] Dennis Runz and Christian Seyda. Programmable I/O-pattern benchmark for cluster file systems. `http://www.?`, september 2009.

[3] Django Software Foundation. Django Homepage. `http://www.djangoproject.com`, june 2010.

[4] Highslide Software. Highcharts Homepage. `http://www.highcharts.com`, june 2010.

[5] jQuery Project. jQuery Homepage. `http://jquery.com`, june 2010.