A Lightweight Framework for Reproducible Parameter Sweeping in Information Retrieval

Richard Eckart de Castilho Iryna Gurevych
Ubiquitous Knowledge Processing (UKP) Lab
Technische Universität Darmstadt
Darmstadt, Germany
http://www.ukp.tu-darmstadt.de

ABSTRACT

Information retrieval experiments consist of multiple tasks like pre-processing or evaluation, each subject to various parameters affecting their results. Dependencies between tasks exist such that one task may have to use the output of another. While many scientific workflow systems come with sophisticated graphical authoring tools and execution environments, they do not integrate well with integrated development environments used for programming. The framework introduced in this paper is lightweight and integrates seamlessly with Java-based experimental setups and development environments while still providing support for declaratively setting up dataflow-based parameter sweeping experiments. To reduce the computational effort of running an experiment with many different parameter settings, the framework uses the tasks and the dataflow dependency information to maintain and reuse intermediate results whenever possible.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.5 [**Software Engineering**]: Design Tools and Techniques; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval

Keywords

Workflow, Dataflow, Experimentation

1. INTRODUCTION

In this paper, we introduce a flexible lightweight framework for parameter sweep experiments geared towards evolution, efficiency and reproducibility of experiments running on a single machine. We present an application of the framework in information retrieval (IR) where it is used to model the tasks[1] of preprocessing topics, preprocessing and indexing documents, retrieval and evaluation.

[1]The name *Task* is an allusion to name given to processing steps in the build tool *Ant* (`http://ant.apache.org/`)

DESIRE 2011 Glasgow, UK

Scientific workflows have been subject of great interest in the past, in particular in the context of grid computing and more recently cloud computing [2, 5]. In IR, grid and cloud technology is used to scale to large datasets and supported by information retrieval frameworks such as Terrier [4], GridLucene [3] or more recently Katta.[2] Modeling IR experiments as a workflow in order to investigate the effects of variations in experimental parameters affecting preprocessing, indexing and retrieval is not within the scope of these frameworks.

Solutions for implementing, sharing and running scientific workflows like Taverna [1] or Kepler [2] are popular in areas such as bioinformatics or astronomy where large amounts of data have to be processed using computationally intensive algorithms. These systems principally serve as an interface for grid computing and may optionally invoke web-services to do the actual processing. While many of them are very sophisticated, coming with graphical workflow editing and monitoring tools, they are less suited to prototyping experiments and to running smaller experiments, particularly such that require a significant amount of programming. In such a scenario, one would prefer a workflow system that integrates well with the preferred development environment. For example, we rely very much on the refactoring capabilities of Eclipse as the implementation of an experiment evolves. Restricted workflow authoring environments would be inconvenient as they do not support code refactoring, debugging etc.

Furthermore, most workflow systems are geared towards running a workflow from the beginning to the end. Intermediate results generated by each task can be inspected but can typically not be reused across several executions of the workflow. Especially when running experiments on a single machine, it is convenient to be able reuse intermediate results from previous runs, e.g. from preprocessing steps, if the code and the parameters used to generate the results did not change. Thus, unnecessary repetitions of the same steps can be avoided.

Finally, workflow systems provide for reproducible experiments, because the workflow often fully specifies the complete experiment. A researcher usually does not aim for reproducible experiments when developing on a single workstation. Often different steps like preprocessing, algorithm execution and evaluation are different programs manually run one by one. From thus, we conclude that is desirable to be able model the complete experimental setup with little programming overhead. A suitable workflow framework

[2]Katta: `http://katta.sourceforge.net/`

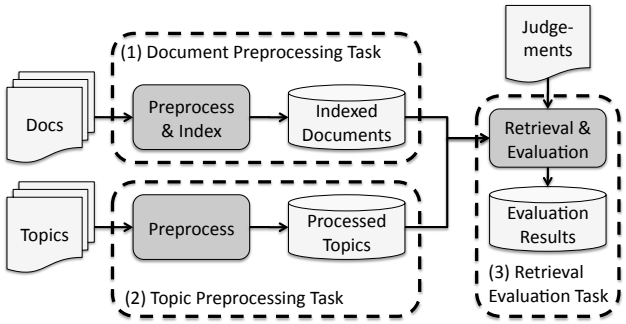

Figure 1: Tasks in a typical IR experiment

should be easy to use and should not cause interoperability problems with other software libraries and frameworks used in the experimental setup.

2. FRAMEWORK CHARACTERIZATION

The framework presented in this paper provides a lightweight and flexible alternative to popular (heavyweight) workflow systems addressing the requirements mentioned in section 1:

Evolution - the framework is completely implemented in Java and workflows can be implemented in any Java-compatible language, e.g. in Groovy. All refactoring facilities of a development environment like Eclipse can be used, facilitating the evolution and maintenance of experiments.

Efficiency - intermediate results are preserved along with all parameter settings that were used to generate them (data provenance). Based on this and the knowledge of the dependencies between tasks, the framework can automatically detect if an intermediate result can be reused.

Reproducibility - the framework facilitates the integration of all computational steps related to an experiment into a single experimental setup that can be run end-to-end with no manual intervention. Combined with the data provenance information makes experiments reproducible.

In addition, the framework provides explicit support for:

Parameter sweeps - a parameter space is explicitly modeled in a declarative manner. An experimental setup can be easily run with many parameter combinations.

Reporting - after the execution of each task, an explicit reporting step is provided. This allows hooking in task-specific reports that visualize the results. Convenience classes are supported to generate tables and scalable PDF charts.

Declarative style - writing the experimental setup in a declarative style is supported through the explicit modeling of tasks, task dependencies and parameters. Boiler-plate code is reduced through the use of dependency injection.

3. INFORMATION RETRIEVAL SCENARIO

To demonstrate the capabilities of our framework, we assume a typical IR experiment consisting of three tasks (Figure 1) :

1. preprocessing and indexing of documents;
2. preprocessing of topics;
3. retrieval and evaluation.

The experiment should be run on two different datasets, using two different retrieval models (Lucene's boolean vector space model and Terrier's BM25), and index by stems and by lemmas.

```
Task topicPreprocessingTask = new TopicPreprocessingTask();
Task evaluationTask         = new EvaluationTask();
evaluationTask.addImportLatest(
  EvaluationTask.IN_TOPIC_XMI,
  TopicPreprocessingTask.OUT_XMI,
  topicPreprocessingTask.type) ;
```

Listing 1: Data dependency: evaluation task imports preprocessed topics from the topic preprocessing task

We start by modeling the three tasks and their dependencies (Section 3.1). Then, we set up the parameter space (Section 3.2), and we configure a batch task which will execute the three experiment tasks with each possible parameter combination (Section 3.3). Finally, we address how to inspect and evaluate the output produced by each tasks. Examples given in this paper are implemented in Groovy for brevity and are limited to illustrative code snippets that should underline the declarative nature of the framework.

3.1 Tasks and Dataflow Dependencies

Our experimental scenario requires three tasks to be performed. The first two tasks (preprocessing and indexing of documents; preprocessing of topics) can be performed independently of each other. The third task (retrieval evaluation), however, depends on the output generated by the other two. Using our framework we can declare dataflow dependencies (Listing 1) between the tasks which serve two purposes:

Data provenance and preservation: Each time a task is executed it is provided with a *context* which can be used to access input data and to write output data (backed by a directory on the file system). In order to access some output produced from another task, a task has to import that output. It is also possible to import data from an URL or from the file system. Data that is not imported from another task can be copied to the context in order to record and preserve all data that was used to produce the output. Data that is imported from another task is only copied if the importing task declares that it intends to modify the data. After a task has been executed the context is closed and never again modified. Currently this approach allows for provenance tracking, for detailed inspection, error analysis and preservation of experimental results. Ultimately we aim to also capture the code that was used to produce a context to allow a comparison between different versions of an experiment and possibly retroactive debugging.

Execution order: The dataflow dependencies also inform the framework about the order in which the tasks have to be executed. At execution time, all tasks are put into a queue and processed one after the other. By means of the dataflow dependencies, the framework can check for a given tasks whether all tasks producing the necessary inputs have already run before. If this is the case, the task is executed, otherwise the task execution is deferred. Dependencies can even depend on parameters, i.e. depending on the parameter configuration a task can choose to import a different output, import an output from a different task or not import at all. Listing 2 illustrates how to implement a dynamic import. More on parameters in Section 3.2.

3.2 Parameter Space

```
@Discriminator String indexPort;
@Discriminator String indexTask;
public void setIndexPort(String aPort) {
  indexPort = aPort;
  addImportLatest(EvaluationTask.IN_DOCUMENT_INDEX,
     indexPort, indexTask);
}
```

Listing 2: Parameter *indexPort* is used to set up a dataflow dependency by implementing a setter

The parameters of our experiment are the *dataset* and the retrieval *model* to be used. In addition we want to see if lemmas or stems are better suited to our retrieval task, so we have a third parameter *termSelector*. These three parameters make up the dimensions of the parameter space. The examples in this section illustrate the declarative style for setting up the parameter space provided by the framework. The simplest way to model the three parameters would be to use simple *discrete dimensions*:

```
def dimDataSet      = Dimension.create(
  'dataSet',      'dataSetEn', 'dataSetDe');
def dimModel        = Dimension.create(
  'model',        'BM25', 'Boolean+VSM');
def dimTermSelector = Dimension.create(
  'termSelector', 'Stem', 'Lemma');
```

Listing 3: Naive parameters (Discrete dimensions)

These do not cover a lot of information necessary to actually conduct the experiment, such as the location from where the data can be read from, the data format, the language of the data, the location of the index, the location of the judgements needed for evaluation and so on, which we will later need to actually implement the experiment tasks. Thus, we will set up a more detailed parameter space (Table 1).

3.2.1 Parameter bundles

We use a *dimension bundle* for the *dataSet* parameter (Listing 4). A dimension bundle specifies a set of parameters that have to be set simultaneously and allows us to set all parameters related to the dataset simultaneously.

Similarly the retrieval model parameter (*model*) breaks down into a number of more specific parameters (Listing 5). The parameter *indexEngine* indicates if Terrier or Lucene should be used, the *indexTask* and *indexPort* indicates how access the index that is generated by the document preprocessing and indexing task (cf. Section 3.1). This is needed

Table 1: Tasks affected by parameters

Parameter	Document Preprocessing Indexing	Topic Preprocessing	Retrieval Evaluation
dataSet			
language	✓	✓	▶
documentsPath	✓	–	▶
topicsPath	–	✓	▶
judgementsPath	–	–	✓
model			
indexEngine	–	–	✓
indexTask	–	–	✓
indexPort	–	–	✓
termSelector	✓	✓	▶

Legend
✓ - task is directly affected by the parameter
▶ - task is indirectly affected through a dataflow dependency

```
def dataSetEn = [
  __bundleId:      'dataSetEn',
  language:        'en',
  documentsPath:   '/data/en/docs',
  topicsPath:      '/data/en/topics',
  judgementsPath:  '/data/en/judgement.qrels'];
def dataSetDe = [
  __bundleId:      'dataSetDe',
  language:        'de',
  documentPath:    '/data/de/docs',
  topicPath:       '/data/de/topics',
  judgementsPath:  '/data/de/judgement.qrels'];
def dimData = Dimension.createBundle(
  'dataSet', dataSetEn, dataSetDe);
```

Listing 4: Data set (dimension bundle)

because, in our scenario, the document indexing tasks actually writes two indexes, one using Terrier and another one using Lucene. Since Terrier supports multiple weighting models, we also have to specify which one to use (BM25).

```
def modelLucene = [
  __bundleId:      'Boolean+VSM',
  indexEngine:     'Lucene',
  indexTask:       docIndexingTask.type,
  indexPort:       OUT_INDEX_LUCENE];
def modelBM25 = [
  __bundleId:      'BM25',
  indexEngine:     'Terrier',
  indexTask:       docIndexingTask.type,
  indexPort:       OUT_INDEX_TERRIER,
  weightingModel:  BM25];
def dimModel = Dimension.createBundle(
  'model', modelLucene, modelBM25);
```

Listing 5: Retrieval model (dimension bundle)

3.2.2 Closures as Parameters

We model *termSelector* parameter dimension using a different approach (Listing 6). The preprocessing task in our experiment generates stems and lemmas, but we might also have in mind to try additional approaches like up synonyms in WordNet and index them as well. The *ClosureDimension* allows us to extract volatile parts of the implementation into a parameter, so it becomes easy to experiment with variations. The two closures provided here select all terms from the preprocessed text and extract either the lemmas or stems from it. These closures are be executed by the tasks preprocessing documents and topics.

```
def dimTermSelector = new ClosureDimension(
  'termSelector', [
  Stems:  {text -> select(text, Stem).collect {it.↩
       value}},
  Lemmas: {text -> select(text, Lemma).collect {it.↩
       value}}]);
```

Listing 6: Index term selection (closure dimension)

Finally we set up a parameter space from all the parameter dimensions (Listing 7).

```
ParameterSpace pSpace =
  [dimDataSet, dimTermSelector, dimModel];
```

Listing 7: Parameter space

3.2.3 Parameter Injection

Tasks are first-class objects and typically implemented by subclassing one of several convenient base-classes. A task class can subscribe to a parameter from the parameter space

by declaring a field with the name of the parameter and annotating it with @Discriminator or @Property.[3] Listing 8 shows how the parameters for the retrieval and evaluation task can be declared.

```
@Discriminator File                documentsPath;
@Discriminator String              language;
@Discriminator DiscriminableClosure termSelector;
```

Listing 8: Parameter injection

3.3 Parameter sweep

Based on the parameter space, the dataflow dependencies, and the persisted task context (Section 3.1), the framework can perform a parameter sweep. The sweep is performed by stepping through each of the possible parameter combinations. For each combination, all tasks are executed as described in Section 3.1. However, the framework can be configured to detect that a task has already been run with a particular parameter combination in a previous experiment run and to re-use the data produced in that run. This allows to interrupt and resume a parameter sweep experiment.

When running our experimental scenario, the framework uses the dataflow dependency information to perform a total of 16 task executions compared to 24 executions if all tasks would be re-executed on a parameter change:

- 4 executions of the document pre-processing and indexing task,
- 4 executions of the topic pre-processing task,
- 4 executions of the retrieval and evaluation task.

```
def batchTaskConfig = [parameterSpace : pSpace,
  executionPolicy: ExecutionPolicy.USE_EXISTING ];
def batchTask = batchTaskConfig as BatchTask;
batchTask.addTask(docIndexingTask);
batchTask.addTask(topicIndexingTask);
batchTask.addTask(evaluationTask);
```

Listing 9: Parameter sweep task

3.4 Reporting

The output produced by the tasks may not be well suited for human inspection, e.g. it may be binary data, compressed data, XML data or some other hardly readable format. It is also likely that the output is not the final result we desire – it may be only the raw data on which the an evaluation still needs to be run. In order to post-processed the task outputs, the framework allows to attach *reports* to each task. The duty of a report is to transform the task output into some form more suitable for human consumption, e.g. to generate comparative charts, tabular overviews, error analysis reports, etc. As reports do not modify the task outputs, it is possible to modify the reporting code or develop new reports and to apply them, even after the experiment run is already complete. The separation of processing and reporting also helps in keeping the implementation for either of them as simple as possible.

While the rest of the framework is domain independent, the reports are quite specific to a domain. For example, we have implemented reports for the IR domain that use *trec_eval*[4] to calculate mean average precision, precision/recall and other retrieval evaluation measures and render that those charts. Other reports support error analysis by showing detailed information comparing the retrieved documents to the relevant documents. Some reports can be attached to parameter sweep tasks and provide comparative overviews over the results for different parameter combinations.

[3]Properties are not included in the dataflow dependency evaluation and must not have any effect on the output produced by the task. They can be used to configure reports.

4. CONCLUSIONS

In this paper we have introduced a framework for batch processing using dataflow dependencies to model parameter sweep experiments that are made up of multiple interdependent tasks. We have given an illustration of how this framework can be used in the context of IR to facilitate running experiments with different datasets, different kinds of preprocessing and different kinds of retrieval models. Framework has a flexible lightweight design and can easily used to orchestrate Java-based experiments. It can serve as an alternative to heavyweight grid- or cloud-based frameworks for small experiments and for prototyping experiments. In particular during phases of intensive development, the fact that the framework is not an application, but an provides an API to be used from Java or compatible languages, is helpful as debugging and refactoring provisions of the integrated development environment of choice can be fully used.

In future, we plan to extend the framework towards support for tasks that require more processing power by allowing to run experiment workflows not only locally, but also on a computing cluster. We aim to do so without sacrificing the simplicity of the API and the refactoring and debugging capabilities when running in an integrated development environment.

5. REFERENCES

[1] D. Hull, K. Wolstencroft, R. Stevens, C. Goble, M. R. Pocock, P. Li, and T. Oinn. Taverna: a tool for building and running workflows of services. *Nucleic Acids Research*, 34(suppl 2):W729–W732, 1 July 2006.

[2] B. Ludäscher, I. Altintas, S. Bowers, J. Cummings, T. Critchlow, E. Deelman, D. D. Roure, J. Freire, C. Goble, M. Jones, S. Klasky, N. Podhorszki, C. Silva, I. Taylor, and M. Vouk. Scientific process automation and workflow management. *Scientific Data Management: Challenges, Existing Technology, and Deployment.*, pages 476–508, 2009.

[3] E. Meij and M. de Rijke. Deploying lucene on the grid. In *Proceedings SIGIR 2006 workshop on Open Source Information Retrieval (OSIR2006)*, 2006.

[4] I. Ounis, C. Lioma, C. Macdonald, and V. Plachouras. Research directions in Terrier. *UPGRADE Special Issue on Web Information Access, Invited Paper*, VIII(1):49–56, Feb 2007.

[5] D. Yuan, Y. Yang, X. Liu, and J. Chen. A cost-effective strategy for intermediate data storage in scientific cloud workflow systems. *2010 IEEE International Symposium on Parallel Distributed Processing IPDPS*, pages 1–12, 2010.

[4]trec_eval: `http://trec.nist.gov/trec_eval/`