SQLReports
Yet Another Relational Database Reporting System

Sergey Afonin, Alexander Kozitsyn and Ivan Astapov
Institute of Mechanics, Moscow State University, Michurinskij av. 1, Moscow, Russia

Keywords: Reporting software, SQL, Template Language, Django.

Abstract: Popular web application frameworks, such as Django, do not provide efficient tools for rapid report development. Specialized reporting software are targeted at visual representation and can generate perfect printed versions of the report. In this paper we describe a minimalistic but quite powerful reporting system developed for a web-based university information system. This reporting system supports zero-programing report development, parametrization of SQL queries, interactive results processing by means of client-side JavaScript libraries, and cross-report references. Main features are similar to well-known reporting systems, such as JasperReports, with more attention to reuse of developed reports, simplicity and interactivity.

1 INTRODUCTION

For almost every modern programming language, such as Java, Ruby, PHP or Python, dozens web application frameworks exist. Most known examples include Ruby on Rails, Symphony, Drupal, Django. Such frameworks provide rapid web application development capabilities, including transparent access to underlying database system by means of the Object Relational Mapping approach. Objects of the programming language are mapped into relational structure and application developer may query the database in terms of object fields, rather than direct joining of tables in SQL. This approach is extremely useful and effective for developing web forms for data browsing and editing because at each time application deals with one or several objects, usually with complex structure.

Nevertheless, data editing is not the only task of an information system. The second part of an information system, probably the most important one, is the reporting subsystem. Users require information from the database using various filtering or aggregation constraints and their information needs change frequently over time. In addition to its main purpose, aggregated representation of information may be used as a means for data quality control: it is difficult to find an error in individual record using simple browsing but many errors reveal itself when aggregated view is used. The above mentioned frameworks provide no support for rapid report development and deployment. One can easily implement any specific report, say in Django, but this requires modification of the application's source code which is unacceptable for large application. Special reporting software, such as JasperReports (Danciu and Chirita, 2007), CrystalReports (Ganz, 2007), CoDe (Risi et al., 2014) or many patented systems (Warren, 2013; Yeh and Kundu, 2006; Bennett and Hu, 2013; Tabb and Herrmann, 1998) are targeted to generation of perfect printed reports. Printed version of a report is the final product. In this paper we describe a simple web-centric reporting subsystem SQLReports[1], that can be used for rapid development of reports suitable for both reporting and dynamic data exploration and cleaning. Key properties are:

- report creation requires no "non-SQL" programming;
- queries may be parameterized;
- one query per report;
- cross-references between reports.

The expected usage scenario assumes that a technically skilled person creates an SQL query constituting the core of the report, and provide some basic formatting preferences, such as titles of fields and parameters. When a user requests for this report, a parameters filling form might appear. The user is asked to choose values from drop-down lists of values, or to enter arbitrary values for textual fields. Then the

[1] Available at https://bitbucket.org/serg_msuru/sqlreports

Afonin S., Kozitsyn A. and Astapov I..
SQLReports - Yet Another Relational Database Reporting System.
DOI: 10.5220/0005114205290534
In *Proceedings of the 9th International Conference on Software Engineering and Applications* (ICSOFT-EA-2014), pages 529-534
ISBN: 978-989-758-036-9

SQL-query associated with the report is executed, and resulting data are passed to the user in simple tabular format. Sorting and additional filtering are implemented on the client's side using JavaScript libraries. Report's result page also contains configurable set of control elements aimed at quick modification of report's parameters and report reevaluation. Some cells of the resulting table may be "clickable": a click either runs dependent report, or opens specified URL. In both cases new HTTP query may include values from the current row or values of report's input parameters. For example, top-level report computes the total number of employees satisfying searching criteria, while dependent report, attached to the number cell, lists all matched employees, providing further links to individual records. Report's result may be exported in CSV, XML, or similar format. PDF generation is also possible by providing appropriate layout description.

During the course of a university information system development we have identified a common pattern in users' activity: select set of objects satisfying specific conditions, choose a subset, and apply some processing function to this subset or to each member. SQLReports partially fits this pattern by providing tools for objects selection (a user can mark some rows) and processing selected records by a specified function. The latter should be implemented in low level programming language, in our case in Python, because records processing is very specific. This select/mark/process scheme is useful for data cleaning task, when authorized users review information entered by other users.

The layout of the paper is as follows. In the next section we describe in more details how reports are created and executed. In Section 3 a template language is defined. In Section 4 possible security issues are discussed.

2 REPORTS DEVELOPMENT AND EXECUTION

Steps required for report developments may be divided into two classes: "compile-time" actions performed by a technically skilled person, and run-time steps related to input parameters processing and query evaluation. Let us note that most steps (with the only exception of report's construction) are performed at run-time. Following components may be identified:

- report constructor — GUI for specifying key properties of reports;
- report's parameters form — automatically generated web-form that allows a user to choose or specify values for report's parameters;
- query generator — server-side code that generates SQL-query, evaluate it, and generates data to be displayed to user;
- results page — client-side code for browsing and searching report's result.

A typical client-server interaction includes the following steps. (1) A user submits a request for some report. This is an HTTP GET or POST request containing report *identifier* and, optionally, some values for input parameters. (2) The server verifies that all input parameters of the requested report are bound to some values. If the requested report requires some parameters not included into original request, then an automatically generated HTML form (the *parameters form*) is returned to the user. (3) User provides values for requested parameters, if any. (4) Finally, when parameters form is completed, real SQL query is instantiated from parameterized report's query and submitted to database server. (5) User browse results page on its side using an interactive JavaScript code, modifies some of report's parameters and re-evaluate report, if required.

A report consists of:

- parameterized SQL query;
- description of input parameters;
- description of output parameters (i.e., selected fields);
- access control information.

The core of each report is a parameterized SQL-query. Supported options for query parametrization are described in Section 3. For the moment we can assume that the query is a usual SQL query with named placeholders. Each input parameter is specified by its name, type, title, visibility and interactiveness flags, and, optionally, a list of values. Meaning of name, type and title fields is straightforward. Data type affects visual control elements and validation code generated for each parameter. Visibility flag controls whether or not a user is allowed to enter values for this parameter via parameter form. Hidden input parameters must be provided in original request or be defined as parameters with default values (e.g. "year" can be substituted by current year by default). If a parameter is marked as interactive, then its value may be changed on the report's result page, e.g. that page contains control elements for changing parameters and report re-evaluation.

Possible values for input parameters may be restricted by a statically or dynamically generated list of values. Static lists of values are stored in the database

and may be used to encode domain specific values, such as male/female, yes/no/unspecified, etc. Static lists of values may be created and modified via report constructor's GUI.

Dynamic lists of values are generated by arbitrary SQL query that selects two fields: key and label. Keys could be of arbitrary data type. Labels are displayed to user. Key of selected item is passed to the server as parameter's value. List generator query may be parameterized as well, with the only restriction that all parameters should be bound before this query is evaluated, i.e. all parameters are either passed with the first request, or have default values (current user, current year, etc.). All keys returned by an SQL query are digitally signed in order to eliminate the possibility of bypassing defined constraints on possible values.

Output parameters, i.e. selected fields, are specified by name, type, label, visibility flag and action URL attached to that field. If an action URL is specified for output parameter then a user sees values for that field as hyperlinks. Action URL is an arbitrary URL with Django-like template syntax: if the URL contains substring of the form `{{name}}`, then this string is replaced by the value of parameter named "name". Both input and output parameters may be used in URL templates, i.e. report's input parameters may be passed from one report to another. In short, action URL may references any value from the *current row*, or any input parameter. Report's output parameter may be marked as hidden, so it can be referenced in URL template string but not presented to user. Typical example of hidden parameter is primary key of some object.

In order to simplify reports development, the report constructor supports "magnetic links" between reports, or between report and external URL. If report developer asks to setup a link for some output parameter of report X to external URL or another report Y (i.e. make this column "clickable"), then all parameters of Y are linked to input or output parameters of X, provided that their names match. A predefined set of URL templates to external resources may be specified as well. Using magnetic links and unified convention for parameters naming one can define cross reference links in one click. Another trick that improve user performance is related to input and output parameters description. Input and output parameters are automatically created from the code of parameterized SQL query. When an administrator updates SQL code the system parses it and creates records for all parameters not presented in the previous version of the code.

If a name of input or output parameter ends with double underline symbol then it is considered as a "magic" parameter. Magic parameters are always hidden. A report may be used for selecting and processing some records. If report's query selects a field with predefined name `checkbox_id__`, then each row in the output table is associated with a checkbox control. User can choose some rows and submit them to special processing function. The name of a function that should be called is determined by report's input parameter `checkbox_fn__` which is passed through to result page. Checkbox processing function checkbox_fn (a kind of callback function for reporting server) will be provided with two lists of identifiers: list of IDs that were shown to the user, and list of IDs marked by user. Two lists make it possible to determine what checkboxes were deselected in user interface.

End user interface is implemented on top of DataTable Javascript library[2] that provides flexible filtering and sorting capabilities out of the box, with almost no programming efforts.

3 SQL TEMPLATE LANGUAGE

In many practical situations actual SQL query that should be submitted to database system depends on values of input parameters, or some external conditions. For example, if a user, who is running the report, belongs to distinguished group, then additional fields should be displayed. Template language of SQLReports provides means for query modification at run time. Four possibilities are provided:

- parameter binding;
- parameter's value embedding;
- query inclusion;
- conditional generation of SQL queries.

Parameter *binding* is the usual method for passing values to SQL queries. For example, in the following query the value of parameter "full_name" will be used as is and the value does not affect the query.

```
SELECT employee_id FROM employee
WHERE employee_name = %(full_name)s
```

When value *embedding* is used, parameter name is replaced by its value inside SQL query. In contrast to parameter binding, embedding changes structure of a query. This method may be used for passing fragments of SQL code into report's query. For example, given a query

```
SELECT %(!field)s, %(!fun)s(salary)
FROM employee
GROUP BY %(!field)s
```

[2] http://datatables.net/

one can pass to parameter "fun" values such as "MAX", or "AVG", and "employee_id" as a value for "field". By choosing appropriate values for embeddable parameters report's developer may change result drastically. We adopted convention that a parameter is embeddable if its name starts with exclamation.

Query inclusion is used for insertion of another report's SQL-query in specified position. As sub-query (included into top-level query) might have input parameters, top-level query may provide specific values for that parameters. A value can be either a constant, or a name of parameter. For example, in the following query value for "param1" (input parameter of a sub-query) is copied from top-level query's parameter "param3", and other parameters get constant values.

```
SELECT subq.fullname, count(*)
FROM (
/* include list_by_name(param1=param3,
      param2=14, param3='abc') */
) subq
GROUP BY subq.fullname
```

If some parameter is not explicitly bound in the include command, then the value of top-level report's parameter with the same name is used. Query inclusion may be used to avoid copying of fragments of SQL code between reports. In previous example nested query may lists some records related to persons, and top level query generates aggregated view. Each number generated by the top-level report may be clickable and connected to list_by_name report. Query inclusion guarantees that two reports always produce consistent results. In some sense, query inclusion correspond to database views, but this approach is more flexible due to parameter substitution.

The last option for query parametrization allows conditional insertion of SQL code. Original query may contain fragments enclosed into comment lines describing conditions that should be satisfied in order to include this fragment of code. For example,

```
SELECT emp.department_id, sum(employee_weight)
FROM employee emp
/* if top_floors=1 */
    , department dep
WHERE emp.department_id = dep.department_id
      AND
      dep.department_floor > 10
/* endif */
GROUP BY emp.department_id
HAVING sum(emp.employee_weight) > 1000
```

The motivating example for conditional insertion of SQL code was related to "OLAP-type" reports. Suppose that for some object a large number of properties may be computed by means of selecting data from related tables. Some users are interested in one subset of properties, other users might deal with another subset of objects' properties. Building one query that always perform all possible joins of relational tables might be inefficient because database systems can not eliminate unnecessary joins if some columns are not selected. In this particular situation conditional expressions in parameterized query may be used for building more or less efficient query. If a user requests for some field, all required table will be joined, and only then. Certainly, a report developer should care about possible name conflicts if one table should appear multiple times in report's query. In the simplest and not so rare scenario when objects' properties are obtained by sub-queries returning one row for each object it is possible to choose a fresh copy of a table each time. Building an efficient query in general case is much more difficult problem and it is unlikely that it can be solved in terms of any template language.

Conditional code can be combined with query insertion command like this:

```
/* if ... */
/* include ... */
/* endif */
```

In order to instantiate a query suitable for submission to database server it is suffice to recursively perform two steps: (1) include all sub-queries (recursive step) and (2) eliminate unnecessary fragments of the query in accordance with conditional constraints.

4 SECURITY ISSUES

Passing and embedding of arbitrary SQL code into main report's query is obviously a big security issue. Even parameters binding could be dangerous because a user can manually substitute one primary key with another, thus accessing information he is not permitted. From the other hand, passing primary keys from one report to another is a convenient way for cross-reports links.

In order to address security issues, report's parameters are divided into two classes: the so-called free parameters, and restricted (technical) parameters. A user is able to provide arbitrary values for free parameters (provided that type restrictions are satisfied). These values are intended to be passed into SQL query via parameters binding, so no SQL injection is possible at this point. As for restricted parameters, only a limited number of values are accepted by the server. These values are obtained from the authorized sources and digitally signed at server's side. Digital signature validates field value, login name of the current user, and server's secret. This scheme guarantees that users can not forge values for hidden parameters (due to server's secret) and can not share valid values with other users (due to username in the signature). When a list of values is generated, all selected

keys are signed as well. It is possible that a specific key value is allowed in one report, but not allowed for this user in another report. For example, it is allowed to print all employees of a department (report X with department ID as a parameter value), but not their salaries (report Y with department ID). In order to prevent "signed keys stealing" the server's secret include randomly generated report instance identifier.

Signing individual parameters may not be very convenient if a reference to report containing a lot of parameters has to be generated. It is possible to publish a GET request on a web page using custom Django template tag `sign_url`.

```
<a href="{% sign_url "/sqlreports/?..." %}">
...</a>
```

In this case the entire string is signed in an OAUTH-like fashion: all report parameters are sorted by name and then by value (it is allowed to insert additional key/value pairs into request, e.g. session id, that will not be included into signature) . If signed request arrives to reports server then all it's values are considered trusted.

In addition to parameter checking procedures, role-based access control (Ferraiolo et al., 1999; Giordano and Polese, 2013) may be used to prevent users from executing arbitrary reports. Issues related to access control lay out of the scope of this article.

Report inclusion command can be dangerous, even in the simples from when all parameters are just copied from top-level report. Indeed, if a report defines an input parameter then report's developer can pass arbitrary value. The main question here is: Should we trust report's developers, or they are malicious by default? This question is especially relevant if we allow users to share their reports. In this work we assume that report's developer are all trusted. Nevertheless, if an underlying databases system provides methods for advanced access control, then additional security checks may be implemented. For example, a reasonable security policy may state that a user is allowed to access any data from his own department or institution, but should not be able to access records form other departments. In Oracle database system it is possible to enforce such constraints by calling one stored procedure before each report. This procedure will setups effective id for the session and all subsequent select queries will only be able to select records related to that department. Clearly, that such constraints are application-specific and can not be realized in a general purpose reporting server.

5 CONCLUSION

Many open source and commercial reporting systems are available. These systems share common features like parameterized SQL queries, formatting directives, data post-processing, cross-references between reports. In this work we propose a minimalistic system that aimed to rapid (one-click) development of interactive reports with standard tabular representation of report's results. We believe that this approach may be useful for solving both quick data analyzing and data cleaning tasks.

It is not clear whether or not powerful constructions should be added the template language. Parameters bindings and query inclusion, and cross-reports links are simple concepts. It is not very difficult to start using this concepts for someone familiar with SQL. Conditional processing of SQL queries may be quite powerful, provided that the parameterized queries remain readable. More sophisticated means, such as post-processing functions, loops over resulting rows, filtering constraints, variables, and arithmetic operations push the language toward universality (if not Turing completeness), but by the cost of a much more specific language. Some simple features are definitely should be implemented. For example, it seems that a boolean variables will be useful for elimination of repeated constraints in *if*-clauses (one can expect what the same condition might appear in SELECT, FROM and WHERE parts of a query if some tables should be joined only if condition holds). In any case, further developments of template language should be based on user's feedback.

As it was mentioned in (Gjorgjevikj et al., 2011), most of database users are not familiar with technology basics. In an information system we are working on no more then one percent out of 15 thousand users are able and willing to develop SQL queries. From the other hand, a report developed by one user may be useful for other users. Apart from improvements of template language, a possible direction of future work may include the development of reports sharing tools. This requires advertising/navigation/searching tasks (how a user can discover what was already done by others?) and some elements of version control: if a report depends on a report of another user and that report has changed, should we automatically update the first report? It seems that there is no general solution that fits all needs and one should support at least version freezing and "dynamic" links pointing to the most recent version of a report. Finial decision about "linking" strategy can be made by report developer only.

REFERENCES

Bennett, D. and Hu, D. (2013). System for database reporting. US Patent 8,620,952.

Danciu, T. and Chirita, L. (2007). Basic notions of JasperReports. In Danciu, T. and Chirita, L., editors, *The Definitive Guide to iReport*, pages 15–23. Apress.

Ferraiolo, D. F., Barkley, J. F., and Kuhn, D. R. (1999). A role-based access control model and reference implementation within a corporate intranet. *ACM Trans. Inf. Syst. Secur.*, 2(1):34–64.

Ganz, C. J. (2007). Crystal Reports and BusinessObjects XI. In *Pro Crystal Enterprise/Business Objects XI Programming*, pages 199–227. Apress.

Giordano, M. and Polese, G. (2013). Visual computer-managed security: A framework for developing access control in enterprise applications. *IEEE Software*, 30(5):62–69.

Gjorgjevikj, D., Madjarov, G., Chorbev, I., Angelovski, M., Georgiev, M., and Dikovski, B. (2011). ASGRT: Automated report generation system. In Gusev, M. and Mitrevski, P., editors, *ICT Innovations 2010*, volume 83 of *Communications in Computer and Information Science*, pages 369–376. Springer Berlin Heidelberg.

Risi, M., Sessa, M., Tucci, M., and Tortora, G. (2014). Code modeling of graph composition for data warehouse report visualization. *Knowledge and Data Engineering, IEEE Transactions on*, 26(3):563–576.

Tabb, L. and Herrmann, C. (1998). Methods for hypertext reporting in a relational database management system. US Patent 5,787,416.

Warren, W. (2013). Efficient delivery of cross-linked reports with or without live access to a source data repository. US Patent 8,521,841.

Yeh, A. and Kundu, A. (2006). Method and system for a reporting information services architecture. US Patent 7,051,038.