# RESTViews and Pyrrho v7

Malcolm Crowe, 19 Nov 2020 (Based on “The May 2018 version of Pyrrho”)

## An introduction to RESTViews

The essential idea with RESTView is that the Pyrrho database allows definition of views where the data is held on remote DBMS(s): at present, the only options are Pyrrho and MySQL. The remote DBMS is accessible via SQL statements sent over HTTP with Json responses.

For MySQL, a simple client called RestifD (source posted on github.com/MalcolmCrowe/restif) provides this HTTP service. The HTTP access provides the user/password combinations set up for this purpose within MySQL by the owners of contributor databases. In the use cases considered here, where a query Q references a RESTView V, we assume that (a) materialising V by Extract-transform-load is undesirable for some legal reason, and (b) we know nothing of the internal details of contributor databases. A single remote select statement defines each RESTView: the agreement with a contributor does not provide any complex protocols, so that for any given Q, we want at most one query to any contributor, compatible with the permissions granted to us by the contributor, namely grant select on the RESTView columns.

Crucially, though, for any given Q, we want to minimise the volume D of data transferred. We can consider how much data Q needs to compute its results, and we rewrite the query to keep D as low as possible. Obviously, many such queries (such as the select \* from V) would need all of the data. At the other extreme, if Q only refers to local data (no RESTViews) D is always zero, so that all of this analysis is specific to the RESTView technology.

We will add a set of query-rewriting rules to the database engine aiming to reduce D by recursive analysis of Q and the views and tables it references. As the later sections of this document explain, some of these rules can be very simple, such as filtering by rows or columns of V, while others involve performing some aggregations remotely (extreme cases such as select count(\*) from V needs only one row to be returned). In particular, we will study the interactions between grouped aggregations and joins. The analysis will in general be recursive since views may be defined using aggregations and joins of other views and local tables.

Any given Q might not be susceptible to such a reduction, or at least we may find that none of our rules help, so that a possible outcome of any stage in the analysis might be to decide not to make further changes. Since this is Pyrrho, its immutable data structures can retain previous successful stages of query rewriting, if the next stage in the recursion is unable to make further progress.

Although in this document the examples are mostly very simple, we aim to present the analysis in such a way as to demonstrate the applicability of the rules to more complex cases. In other studies, such as the Sierra Leone example, queries can reference multiple stored queries (view definitions) and functions. For now, RESTViews are only found in Pyrrho, but in principle we could have several stages where one RESTView is defined using other RESTViews. We also bear in mind that a query Q might involve joins of RESTViews possibly from the same remote database(s).

There are two types of RESTView corresponding to whether the view has one single contributor or multiple remote databases. In the simple exercises in this document, V is a RESTview with one contributor, and W has two. In the multiple-contributors case, the view definition always includes a list of contributors (the “using table”, VU here) making it a simple matter to manage the list of contributors.

The technical details can be found in the Pyrrho documentation (Pyrrho.pdf and SourceIntro.pdf in the distribution and were presented at DBKDA 2017 in Barcelona by Fritz Laux. For simple examples, see the definitions of V, W, VU and M in the Appendix of this paper. The database rv described there is used for the following illustrations.

## Tracking query rewriting

The test cases considered in this document are set up as follows. Everything assumes that RestifD.exe is running (it uses port 8078 and needs no configuration), and the MySQL server is on the local machine.

In MySQL, at a command line

**create database db;**

**use db**

**create table T(E integer,F nvarchar(6));**

**insert into T values(3,'Three'),(6,'Six'),(4,'Vier'),(6,'Sechs');**

**grant all privileges on T to 'root'@'%' identified by 'admin';**

**create database dc;**

**use dc**

**create table U(E integer,F varchar(7));**

**insert into U values(5,'Five'),(4,'Four'),(8,'Ate');**

**grant all privileges on U to 'root'@'%' identified by 'admin';**

In Pyrrho: Start up a command line with pyrrhocmd rv. Then at the SQL> prompt:

**create view V of (E int,F char) as get 'http://root:admin@localhost:8078/db/t'**

**create table VU (d char primary key, k int, u char)**

**insert into VU values ('B',4,'http://root:admin@localhost:8078/db/t')**

**insert into VU values ('C',1,'http://root:admin@localhost:8078/dc/u')**

**create view W of (E int, D char, K int, F char) as get using VU**

**create table M (e int primary key, n char, unique(n))**

**insert into M values (2,'Deux'),(3,'Trois'),(4,'Quatre')**

**insert into M values (5,'Cinq'),(6,'Six'),(7,'Sept')**

V uses a simple get, while W uses a “get using VU” where the using table VU defines W as a union of contributions from two remote databases, each specified by a table URL and a static filter on the first few columns of that table. In what follows, we will see how additional filters and aggregations can be passed to the remote databases before retrievals from the views. The aim should be never to retrieve all the rows of the view.

As a first step, consider just such a simple query to V:

**(A1) table V**

In v7, the RESTView definition for V is compiled into a RestRowSet chematically outlined as follows

{RestRowSet (E int, F char) GF: select E,F RQ: select E,F from “db/t” }

so that table V is computed by evaluating S into an explicit rowset matching V’s rowtype. We refer to the query select .. from url as the “global from” and this will be implemented on the remote database db by a similar-looking “remote query” targeting table t. In this simple case the GF and RQ are very similar, but in general there is a translation of select expressions in the GF to the select expressions in the RQ, to handle aggregations and built-in functions (Pyrrho has a function ColsForRestView).

These two elements (the RestRowSet and the GlobalFrom) are among the objects defined in the framing part of the RestView. Pyrrho refers to all such objects using uids rather than names, and after compilation these uids will be in the range defined by the extent of the RestView definition in the transaction log file. A view might be referenced more than once in a query, so referenced views are instantiated so that each uses a fresh set of uids in the transaction range.

**(A2) table W**

With the “get using” syntax, we can think of the resulting rowset as

{JoinRowSet (E int, D char, K int, F char)

{TableRowSet (D,K,U) VU } natural join

{RestRowSet GF: select (E,D,K,F) RQ: select \* from U }}

where we note that the url U is one of the columns of VU (a “lateral” join). This rewriting greatly simplifies the query processing involved. However, in general the RQ will depend on U, as the various remote DBMS targeted by U may have slightly different query syntax.

As we will see, the optimisations resulting from a full analysis of “get” RESTViews can generally be applied also in the more complex “get using” RESTViews.

## Filters

The main topic in this paper is the way that Pyrrho transforms RESTView queries so as to optimise remote retrieval. The first aspect of rewriting we consider is filters. If there are some columns of the RESTView that are not used in the given query, there is an obvious reduction, and if a where-condition can be passed to the remote database, this will also reduce the number of rows returned. If, for example, a lateral join is constructed where one of the derived tables is from a view, the global from may use a filter that refers to data defined elsewhere in the global query, so that there may be a new retrieval each time such data changes.

We begin with a simple filter on V and return to consider W and more complex cases at the end of this section.

**(B1) select \* from V where E=3**

We do not want the effect of this to be the retrieval of all of rows of db/t followed by selecting the row(s) with e=1. We can depict this undesirable version as

{FilterRowSet V(E,F) where E=3

{RestRowSet GF:select E,F RQ:select E,F from “db/t”}}

Instead we need to ensure that we only retrieve the rows of db/t that match the given where condition, i.e.

{FilterRowSet V(E,F)

{RestRowSet GF: select 3 as E,F RQ: select F from “db/t” where E=3}}

Here we notice that the filter is on a remote column, so we want the filter to be passed through to the remote query. This time there is a further reduction in the data traffic, since the constant column E=3 does not need to be included in the response from the remote contributor.

**(B2) select \* from V where E>3**

With an inequality filter such as E>3 we need to retain both columns in the remote query:

{SelectRowSet V(E,F)

{RestRowSet GF: select E,F RQ: select E,F from “db/t” where E>3}

With a filter on variable data such as E=X (e.g. X is defined at a higher level in a recursive query) these optimisations are still worth doing, even though we need to rebuild the RestRowSet each time X changes. Such a dependency needs to be flagged at the Needs stage of rowset building, so that the “lateral” join can be prepared.

Finally in this section, consider selections from W. First consider the case where the filter is on the usingTable:

**(B3) select \* from w where k=1**

We want the non-optimised version

{JoinRowSet (E int, D char, K int, F char)

{TableRowSet (D,K,U) VU} natural join

{RestRowSet GF: (E,D,K,F) RQ: select E,D,K,F from U }

where K=1}

to be rewritten as

{JoinRowSet (E int, D char, K int, F char)

{SelectRowSet (D,U) VU where K=1 } natural join

{RestRowSet (E,D,F) .. RQ: select E,D,F from U where K=1}

as this will reduce the number of requests to remote DBMS. As with the case above, there was an opportunity to reduce the columns and rows of the data returned from the remote contributors.

## Aggregations

Next, we look at how aggregations can be handled with RESTViews. The interesting cases are all for the “get using” case. Notice that the first two examples are so simple that each contributor DBMS only needs to return a single row.

**(C1) select count(e) from w**

Before optimisation we have

{EvalRowSet select count(e) from

{JoinRowSet (E,D,K,F)

{TableRowSet (D,K,U) VU} natural join

{RestRowSet GF: select E,D,K,F RQ: select E,D,K,F from U }}}

We will need an extra identifiers C1 to describe the transformation and use aliasing to rename columns. Note that we still need D and K to be returned for the natural join:

{EvalRowSet select sum(C1) as count

{JoinRowSet (C1)

{TableRowSet (D,K,U) VU} natural join

{RestRowSet GF: select C1,D,K RQ: select count(E) as C1,D,K from U}

The optimised version only receives one row per contributor.

**(C2) select max(f) from w having e>4**

Before optimisation we have

{EvalRowSet select max(f) from

{JoinRowSet select E,D,K,F from

{TableRowSet (D,K,U) VU} natural join

{RestRowSet GF: select E,D,K,F RQ: select E,D,K,F from U }}}

After:

{EvalRowSet select max(C2)

{JoinRowSet (C2)

{TableRowSet (D,K,U) VU} natural join

{RestRowSet GF: select D,K,C2 RQ: select D,K,max(F) as C2 from U where E>4}

We see that each contributor sent their maximum, and then we finally get the overall maximum.

With grouping operations, the grouping can take place in stages, but we need to group on the D,K columns too:

**(C3) select sum(e), char\_length(f) as x from w group by x**

Here the grouping is by a remote formula. Before optimisation

{GroupedRowSet select sum(E),char\_length(F) as X group by X from

{JoinRowSet (E,D,K,F)

{TableRowSet (D,K,U) VU} natural join

{RestRowSet GF: select E,D,K,F RQ: select E,D,K,F from U }}}

After:

{GroupedRowSet select sum(C3),X group by X from

{JoinRowSet (C3, X)

{TableRowSet (D,K,U) VU} natural join

{RestRowSet GF: D,K,C4,X RQ:

select D,K,sum(E) as C4,char\_length(F) as X from U group by D,K,X}}}

Grouping by a non-remote formula:

**(C4) select count(\*),k/2 as k2 from w group by k2**

Before:

{GroupedRowSet select count(\*),K/2 as X group by X from

{JoinRowSet (E,D,K,F)

{TableRowSet (D,K,U) VU} natural join

{RestRowSet GF: E,D,K,F RQ: select E,D,K,F from U }}}

After:

{GroupedRowSet select sum(C3),X group by X from

{JoinRowSet (C3,K/2 as X)

{TableRowSet (D,K,U) VU} natural join

{RestRowSet GF: D,K,C3 RQ: select D,K,count(\*) as C3 from U group by D,K}}}

For complex expressions containing aggregations the contributors provide partial sums which can be globally combined. Thus a global computation of AVG(x) requests SUM(x) and COUNT(x) from contributors, other functions can be constructed using SUM(x\*x) and so on.

**(C5) select avg(e) from w**

Before:

{GroupedRowSet select avg(e) from

{JoinRowSet (E,D,K,F)

{TableRowSet (D,K,U) VU} natural join

{RestRowSet GF: E,D,K,F RQ: select E,D,K,F from U }}}

After:

{EvalRowSet select sum(C4)/sum(C5) as avg from

{JoinRowSet (C4,C5)

{TableRowSet (D,K,U) VU} natural join

{RestRowSet GF: C4,C5,D,K

RQ: select sum(E) as C4,count(E) as C5,D,K from U group by D,K}}}

We see that AVG has been implemented using sum and count. This is only done for the purposes of RESTViews: for normal operations Pyrrho implements AVG (and STDDEV\_POP) as a primitive function.

The mechanism is clever enough to identify common subexpressions, e.g. sum(e)\*sum(e) .

We see that the remote database computed just one instance of sum(E), to be squared in QS. As with filters, if an expression contains references to tables other than the remote view and the using table, we may be unable to perform these rewriting steps.

## RESTView and Join

One of the steps in constructing a RESTView is to reduce the view to the columns needed for the given query. When the RESTView is used in a join, we need to ensure that columns needed for the joinCondition are added to the list of needed columns.

**(D1) select f,n from w natural join m**

Before:

{JoinRowSet (F,N)

{JoinRowSet (E,D,K,F)

{TableRowSet (D,K,U) VU} natural join

{RestRowSet GF: E,D,K,F RQ: select E,D,K,F from U }}} natural join

{TableRowSet (E,N) }

This time, there is no reduction, because all the remote columns are needed. The effect of the top-level natural join here is to lose rows where the E columns do not match.

## More tests for expression rewriting

Considering the different cases of expressions where left and right are aggregated and or local or not, and grouped or not, there are 12 cases to consider. Excluding cases where the role of the left and right operands can be swapped, the following tests are of interest:

1. select e+char\_length(f) as x,n from w natural join m (both terms in the expression are remote)

Note that once again there are 7 terms in the intermediate row set but only 6 in the join.

1. select char\_length(f)+char\_length(n) from w natural join m (one term is remote)
2. select sum(e)+char\_length(max(f)) from w (both aggregations remote)
3. select count(\*),e+char\_length(f) as x from w group by x

The following example shows an expression that cannot be passed into the remote query. With the selected optimisation strategy, we should group by e, and then we need to compute x at the same level as the join.

1. select count(\*),e+char\_length(n) as x from w natural join m group by x
2. select sum(e)+char\_length(f),f from w natural join m group by f

Similar to a previous example, in the following query, the remote query groups by the join column e.

1. select sum(char\_length(f))+char\_length(n) as x,n from w natural join m group by n

## Filters and Join

A filter can be moved to a factor of a join provided that factor provides all of the columns needed to compute the filter. A nice optimisation here would be to take advantage of available indexes and the join condition. Here a filter on n would select a single row of the join.

(F1) Select f,n from w natural join m where n='Cinq'

This is a sort of functional dependency not special to RESTViews (and not yet in Pyrrho), so we skip the details here.

## Aggregation and Join

If a grouped query uses a join whose non-remote factor has key J, an aggregation operation grouped by G can be shared with the factors of the join by applying a similar aggregation grouped by G∪J. In this example, the join column is the primary key of both factors. If there is no primary key defined, then all columns are needed to form the join.

(G1) Select count(\*) from w natural join m

We see that the aggregation is passed to the remote system, with grouping on the join column e. This does not save any effort in this case, but in general the aggregation will reduce the size of the remote rowset, even when grouping on the joined column.

## Updatable RESTViews

Although I haven’t got a convincing use case for this yet, as a technical matter it is important to support updatable views. Consider

(H1) update v set F='Tri' where E=3:

Note that Pyrrho does not know how many remote records have been updated (since MySQL does not report this at present), hence it merely reports “0 records affected in rv” the local database. The two rows retrieved from V are just where we have checked by asking for table v.

Next consider

(H2)update w set f=’Eight’ where e=8:

Operations such as

(H3) Insert into v values(9,’Nine’)

(H4) delete from v where e=9 are also supported.

## Subqueries

The above analyses and optimisations also need to be available when RESTViews are used in larger SQL queries. The analysis of grouping and filters needs to be applied top down so that as much as possible of the work is passed to the remote systems. Each RESTView target or subquery will receive a different request, and the results will be combined as rowsets of explicit values.

This contrasts with the optimisations used for local (in-memory) data, which instead aims to do as little work as possible until the client asks for successive rows of the results. In addition, detailed knowledge of table sizes, indexes and functional dependencies is available for local data, which helps with query optimisation.

## Strategy

From the above, the following strategies emerge. We will try to work on rowsets, rather than on the queries. We will work on the getusing case as get followed by join.

1. The RestRowSet works out the remoteCols from the view.Domain
2. RestView analysis is triggered on the complete set if a restview is referenced.
3. A remote column can be dropped from the request if nobody references it.
4. If a remote column is used as a simple filter, we can pass the filter to the remote contributor, and simplify everything by using its constant value.
5. If a remote column is aggregated, we can perform some or all of the aggregation in the remote, but we may need to group by the other visible remote columns.
6. With joins we need to preserve columns referenced in the join condition, and keep track of keys. But we do not attempt to construct remote joins (a different restview should be created for this).

## References

Crowe, M. K., Begg, C. E., Laux, F., Laiho, M. (2017): Data Validation for Big Live Data, in Schmidt, A, Laux, F., Hritovski D, Ohnishi, S-I. (eds): DBKDA 2017: Proceedings of the Ninth International Conference on Databases, Knowledge and Data Applications, Barcelona May 21-25, ISBN 978-1-61208-558-6 (IARIA), p. 30-36

Crowe, M. K. (2017): Restif: A Web server that provides a REST interface for a local MySQL server, <https://github.com/MalcolmCrowe/Restif>

PyrrhoDB: [www.pyrrhodb.com](http://www.pyrrhodb.com)

## Appendix

### Setup for tests

The test cases considered in this document are set up as follows. Everything assumes that RestifD.exe is running (it uses port 8078 and needs no configuration), and the MySQL server is on the local machine.

In MySQL, at a command line

create database db;

use db

create table T(E integer,F nvarchar(6));

insert into T values(3,'Three'),(6,'Six'),(4,'Vier'),(6,'Sechs');

grant all privileges on T to 'root'@'%' identified by 'admin';

create database dc;

use dc

create table U(E integer,F varchar(7));

insert into U values(5,'Five'),(4,'Four'),(8,'Ate');

grant all privileges on U to 'root'@'%' identified by 'admin';

In Pyrrho: We assume OSPSvr.exe is running, with flags such as –V -E -H. Start up a command line with ospcmd rv. Then at the SQL> prompt:

create view V of (E int,F char) as get 'http://root:admin@localhost:8078/db/t'

create table VU (d char primary key, k int, u char)

insert into VU values ('B',4,'http://root:admin@localhost:8078/db/t')

insert into VU values ('C',1,'http://root:admin@localhost:8078/dc/u')

create view W of (E int, D char, K int, F char) as get using VU

create table M (e int primary key, n char, unique(n))

insert into M values (2,'Deux'),(3,'Trois'),(4,'Quatre')

insert into M values (5,'Cinq'),(6,'Six'),(7,'Sept')

### Retrieval tests

For consistency of output, restart OPSSvr before each test. Use any combination of the following lines as input to ospcmd rv.

table v

select \* from V where e=6

table w

select \* from w where e<6

select \* from w where k=1

select count(e) from w

select count(\*) from w

select max(f) from w

select max(f) from w where e>4

select count(\*) from w where k>2

select min(f) from w

select sum(e)\*sum(e),d from w group by d

select count(\*),k/2 as k2 from w group by k2

select avg(e) from w

select f,n from w natural join m

select e+char\_length(f) as x,n from w natural join m

select char\_length(f)+char\_length(n) from w natural join m

select sum(e)+char\_length(max(f)) from w

select count(\*),e+char\_length(f) as x from w group by x

select count(\*),e+char\_length(n) as x from w natural join m group by x

select sum(e)+char\_length(f),f from w natural join m group by f

select sum(char\_length(f))+char\_length(n) as x,n from w natural join m group by n

Select count(\*) from w natural join m

### Tests on updatability

table v

update v set F='Tri' where E=3

insert into V values (9,'Nine')

table V

delete from V where E=9

update v set F='Tri' where E=3

table V

update w set f='Eight' where e=8

insert into w(D,E,F) values('B',7,'Seven')

table v

table w

delete from w where E=7

update v set f='Ate' where e=8

update w

table v

table w

table "Log$"