

The May 2018 Version of Pyrrho

Notes on RESTView implementation

Malcolm Crowe, 19 July 2018

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 obvious 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.

Introducing Pyrrho's command line options

The March 2018 build of the Pyrrho runtime system makes the internal operations of Pyrrho more visible (the so-called -V, -E, and -H server startup flags), and these mechanisms are also explained here. For a full list of command line options for Pyrrho, see Pyrrho documentation.

On startup, the Pyrrho server OSPSvr.exe shows the following:

```
-d:\DATA -E -V -H Enter to start up
Pyrrho DBMS (c) 2018 Malcolm Crowe and University of the West of Scotland
6.2 (19 July 2018) www.pyrrhodb.com
Open Source Edition
LOCAL SERVER VERSION
APPEND STORAGE VERSION
PyrrhoDBMS protocol on 127.0.0.1:5433
Database folder \DATA\
HTTP service started on port 8180
HTTPS service started on port 8133
```

```
Command Prompt - ospcmd rv
Microsoft Windows [Version 10.0.10586]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\Users\66668214>cd \pyrrhodb\py*\op*\wi*

C:\PyrrhoDB\Pyrrho\OpenSource\Windows>ospcmd rv
SQL> select d,k from vu

-|-|
D|K|
-|-|
B|4|
C|1|
-|-|
SQL>
```

Start up the Pyrrho client OSpCmd with ospcmd rv, and do a simple Select:

```
select d,k from vu
```

VU actually defines two contributor databases for the RESTView W described by VU. There is an additional column U giving the URLs for the interaction (see the Appendix).

With the selected server flags, the OSPSvr command window now shows 4 extra lines:

```
select "232","276" from "224"
Transaction 4 (D 232#B7:18,K 276#B7:18)
B7:7: Selected ROW(D 232 CHAR,K 276 INTEGER)
B7:18: Index VU (D 232#B7:18,K 276#B7:18,U 297#B7:18)
```

```
select "232","276" from "224"
Transaction 4 (D 232#B7:18,K 276#B7:18)
B7:7: Selected ROW(D 232 CHAR,K 276 INTEGER)
B7:18: Index VU (D 232#B7:18,K 276#B7:18,U 297#B7:18)
```

The first line here is from the -V flag, and shows how the select request has been “compiled” by the server, replacing the current names of database objects by their positions in the transaction log. If you repeat these tests, these positions will be slightly different for you as the transaction log contains a note of your username and the database name. We see that (at least on my system) d has defining position 232, K has 276 and VU has 224. The transaction log can be inspected using the command table "Log\$". Up-to-date information about database objects can be inspected using the Role\$ tables, e.g. table "Role\$Column" (roles can rename database objects). System tables are documented in Pyrrho.pdf. During a transaction it is of course possible to have database objects that have not yet been given positions in the transaction log: these show with numbers such as ‘1, ‘2 etc (internally these are 0x40000001 etc).

The next three lines in the server window come from the -E flag, showing the stages of processing of the select command and the rowsets produced at each stage. The first line

```
Transaction 4 (D 232#B7:18,K 276#B7:18)
```

records the user transaction and some details of the result rowtype. The first user transaction run by the server shows as Transaction 4. We see that the result row type has two columns D and K as expected. The numbers 232 and 276 are the defining positions already noted.

We also see block numbers for the identifiers. These are allocated by the parser: Each transaction and each new parser instance gets a new number, so we see B7 in the block identifiers on these lines. The blockid showing for D and K corresponds to the defining source position 18 of FROM VU, since VU defined them.

```

      1      2
01234567890123456789012345678
select "232","276" from "224"
```

The symbols D and K from the select part of the query were symbol references in block B5:7. The next two lines in the server window show how the transaction result has been computed using two rowsets. The first one shown is the SelectedRowSet for the “query specification” B5:7 with the expected rowtype (here showing data types)

B7:7: Selected ROW(D 232 CHAR,K 276 INTEGER)

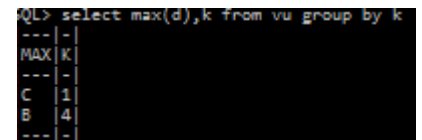
The second shows an IndexRowSet for the table VU (IndexRowSet because VU has a primary key index).

B7:18: Index VU (D 232#B7:18,K 276#B7:18,U 297#B7:18)

The indentation by one space shows this is the source rowSet for the one above. We also see that VU has another accessible column U. For a more complex selection, as we will see later, there will be additional lines for aggregation, joins, ordering etc. For example, with an aggregation query there would be a GroupingRowSet line showing the grouping columns:

```

select max("232"),"276" from "224" group by "276"
Transaction 6 (MAX(D 232) as MAX,K 276#B7:24)
B7:7: Grouping (MAX(D 232) as MAX,K 276#B7:24) groups [GROUP K 276]
      B7:24: Index VU (D 232#B7:24,K 276#B7:24,U 297#B7:24)
```



```

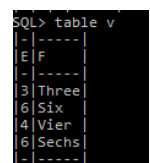
SQL> select max(d),k from vu group by k
---|---
MAX|K
---|---
C  |1
B  |4
---|---
```

The -H server flag allows us to monitor HTTP interaction with the MySQL server. To show the effect of the -H flag, let us look at tables V and W

table V

```

GF B9:7 (E 107#B9:7,F 125#B9:7)
table "144"
Transaction 8 (E 107#B9:7,F 125#B9:7)
B9:7: Remote TABLE(107 INTEGER,125 CHAR) RESTView http://root:admin@localhost:8078/db/t E,F
http://root:admin@localhost:8078/db select E,F from t
--> 4 rows
B9:7: Explicit
      Row(E=3,F=Three)
      Row(E=6,F=Six)
      Row(E=4,F=Vier)
      Row(E=6,F=Sechs)
```



```

SQL> table v
---|---
E  |F
---|---
3  |Three
6  |Six
4  |Vier
6  |Sechs
---|---
```

The -H flag has contributed two lines showing the HTTP interaction

```

http://root:admin@localhost:8078/db select E,F from t
--> 4 rows
```

We see the details of the GET request made over HTTP and the number of rows returned.

As before, the -V flag shows the compiled version of the select statement, here table “144”. In fact 144 is the RESTView – tables and views occupy the same namespace in the database RV.

Let us study the -E output lines.

```

GF B9:7 (E 107#B9:7,F 125#B9:7)
table "144"
Transaction 8 (E 107#B9:7,F 125#B9:7)
B9:7: Remote TABLE(107 INTEGER,125 CHAR) RESTView http://root:admin@localhost:8078/db/t E,F
B9:7: Explicit
      Row(E=3,F=Three)
```

```
Row(E=6,F=Six)
Row(E=4,F=Vier)
Row(E=6,F=Sechs)
```

the first line here is the “globalFrom” computed for the RESTView, which here has the two columns of V. The transaction number 8 follows use of block B7 above, and here we see block numbers B9. The defining positions for columns E and F are actually in an anonymous virtual table constructed during the definition of VIEW V which has position 144. Because this is a RESTView, the column references in the query specification at blockid B9:7 are used, as there is no real FROM table.

We see the RemoteRowSet with its description. We also see the resulting ExplicitRowSet containing a formatted version of the Json data returned from MySQL (the –E flag shows the rows if there are fewer than 10, otherwise just the number of rows). The rows are indented below Explicit. The B9:7 row is not indented because this is the current version of the result of the transaction, and not the source of the RemoteRowSet. We have retrieved all of the rows in the view: as we will see later, we usually transform the request so that only some of the data needs to be returned

Similarly, let us look at table W:

```
GF B11:7 (E 508#B11:7,D 528#B11:7,K 549#B11:7,F 570#B11:7)
table "591"
Transaction 10 (E 508#B11:7,D 528#B11:7,K 549#B11:7,F 570#B11:7)
B11:7: Remote TABLE(508 INTEGER,528 CHAR,549 INTEGER,570 CHAR) RESTView Using VU
B11:7U: Index VU (D 232#B11:7U,K 276#B11:7U,U 297#B11:7U)
http://root:admin@localhost:8078/db select E,'B' as D,4 as K,F from t
--> 4 rows
http://root:admin@localhost:8078/dc select E,'C' as D,1 as K,F from u
--> 3 rows
B11:7S: Explicit
Row(E=3,D=B,K=4,F=Three)
Row(E=6,D=B,K=4,F=Six)
Row(E=4,D=B,K=4,F=Vier)
Row(E=6,D=B,K=4,F=Sechs)
Row(E=5,D=C,K=1,F=Five)
Row(E=4,D=C,K=1,F=Four)
Row(E=8,D=C,K=1,F=Ate)
```

```
SQL> table w
| - | - | - | - |
| E | D | K | F |
| - | - | - | - |
| 3 | B | 4 | Three |
| 6 | B | 4 | Six |
| 4 | B | 4 | Vier |
| 6 | B | 4 | Sechs |
| 5 | C | 1 | Five |
| 4 | C | 1 | Four |
| 8 | C | 1 | Ate |
| - | - | - | - |
```

```
GF B11:7 (E 508#B11:7,D 528#B11:7,K 549#B11:7,F 570#B11:7)
table "591"
Transaction 10 (E 508#B11:7,D 528#B11:7,K 549#B11:7,F 570#B11:7)
B11:7: Remote TABLE(508 INTEGER,528 CHAR,549 INTEGER,570 CHAR) RESTView Using VU
B11:7U: Index VU (D 232#B11:7U,K 276#B11:7U,U 297#B11:7U)
http://root:admin@localhost:8078/db select E,'B' as D,4 as K,F from t
--> 4 rows
http://root:admin@localhost:8078/dc select E,'C' as D,1 as K,F from u
--> 3 rows
B11:7S: Explicit
Row(E=3,D=B,K=4,F=Three)
Row(E=6,D=B,K=4,F=Six)
Row(E=4,D=B,K=4,F=Vier)
Row(E=6,D=B,K=4,F=Sechs)
Row(E=5,D=C,K=1,F=Five)
Row(E=4,D=C,K=1,F=Four)
Row(E=8,D=C,K=1,F=Ate)
```

Note that the globalFrom and result table contain columns D and K from the using table VU as well as the remote data (effectively a join). This RESTView has defining position 591. The RemoteRowSet uses the VU table examined earlier, instead of a single URL, and retrieves all of the rows in the view. The –H flag information shows the two requests and the numbers of rows returned. The IndexRowSet line has a special blockid to show that it is a using table for the RESTView retrieval. . Note that columns D and K from the usingTable are supplied as constants to the remote database: this is useful if their values are used in expressions in other columns of a SELECT.

If we use select * instead of table, we get an extra SelectedRowSet as before.

Select * from w

```
SQL> Select * from w
| - | - | - | - |
| E | D | K | F |
| - | - | - | - |
| 3 | B | 4 | Three |
| 6 | B | 4 | Six |
| 4 | B | 4 | Vier |
| 6 | B | 4 | Sechs |
| 5 | C | 1 | Five |
| 4 | C | 1 | Four |
| 8 | C | 1 | Ate |
| - | - | - | - |
```

```

GF B13:15 (E 508#B13:15,D 528#B13:15,K 549#B13:15,F 570#B13:15)
Select * from "591"
Transaction 12 (E 508#B13:15,D 528#B13:15,K 549#B13:15,F 570#B13:15)
B13:7: Selected ROW(E 508 INTEGER,D 528 CHAR,K 549 INTEGER,F 570 CHAR)
B13:15: Remote TABLE(508 INTEGER,528 CHAR,549 INTEGER,570 CHAR) RESTView Using VU
B13:15U: Index VU (D 232#B13:15U,K 276#B13:15U,U 297#B13:15U)
http://root:admin@localhost:8078/db select E,'B' as D,4 as K,F from t
--> 4 rows
http://root:admin@localhost:8078/dc select E,'C' as D,1 as K,F from u
--> 3 rows
B13:15S: Explicit
Row(E=3,D=B,K=4,F=Three)
Row(E=6,D=B,K=4,F=Six)
Row(E=4,D=B,K=4,F=Vier)
Row(E=6,D=B,K=4,F=Sechs)
Row(E=5,D=C,K=1,F=Five)
Row(E=4,D=C,K=1,F=Four)
Row(E=8,D=C,K=1,F=Ate)

```

Note that in these illustrations we will probably have different transaction numbers from you as the examples are not all run in a single sequence.

Filters

The main topic in this paper is the way that Pyrrho transforms RESTView queries so as to optimise remote retrieval. For stored queries such as views, such rewriting must be done on a local copy of the query. 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. We begin with such simple filters, and return to consider more complex case at the end of this section.

Let us first illustrate the operation of the `-E` flag in showing the operation of filters. There are two levels for filters in the Pyrrho engine. There is a low-level filter where particular requirements on defining positions and associated values can be specified: this is called match. There is a higher-level filter based on SQL value expressions, corresponding to SQL where and having conditions. To assist with optimisation Pyrrho works with lists of such conditions anded together. Staying with the only normal table in our example Pyrrho database, let us look at `select D,K from VU where k>=4`

```

SQL> select D,K from VU where k>=4
-|-|
D|K|
-|-|
B|4|
-|-|

```

```

select "232","276" from "224" where "276">=4
Transaction 14 (D 232#B15:19,K 276#B15:19)
B15:7: Selected ROW(D 232 CHAR,K 276 INTEGER)
B15:19: Index VU (D 232#B15:19,K 276#B15:19,U 297#B15:19) where (K 276#B15:19)>=4)

```

`select D,K from VU where D='B'`

```

SQL> select D,K from VU where D='B'
-|-|
D|K|
-|-|
B|4|
-|-|

```

```

select "232","276" from "224" where "232"='B'
Transaction 16 (D 232#B17:19,K 276#B17:19)
B17:7: Selected ROW(D 232 CHAR,K 276 INTEGER)
B17:19: Index VU (D 232#B17:19,K 276#B17:19,U 297#B17:19) match D 232=B

```

We note that the filters have been applied at a lower level (IndexRowSet in this case). This is more efficient in general because fewer rows are involved in later operations such as join, and the match may be on a key column (as here) so that the index does not need to be traversed. The equality condition is implemented as a low-level match, rather than using expression evaluation.

To see the effect on RESTViews, first consider `select * from V where E=3`, and where `E>3`. We see the where condition is passed in to the HTTP request, so that only one row is returned from the remote server. (This matters a lot if the remote rowset is very large.) We will see that matches cannot be used for the remote retrieval.

```

SQL> select * from v where e=3
-|-|
E|F|
-|-|
3|Three|
-|-|

```

select * from V where E=3

```
select * from "144" where "107"=3
Transaction 18 (E 107#B19:16,F 125#B19:16)
B19:7: Selected ROW(E 107 INTEGER,F 125 CHAR)
B19:16: Remote TABLE(107 INTEGER,125 CHAR) RESTView http://root:admin@localhost:8078/db/t E,F match
E 107=3
http://root:admin@localhost:8078/db select E,F from t where E=3
--> 1 rows
B19:16: Explicit match E 107=3
Row(E=3,F=Three)
select * from V where E>3
```

```
SQL> select * from V where E>3
-|-|-|-----|
E|F|
-|-|-|-----|
6|Six|
4|Vier|
6|Sechs|
-|-|-|-----|
```

```
GF B21:16 (E 107#B21:16,F 125#B21:16)
select * from "144" where "107">3
Transaction 20 (E 107#B21:16,F 125#B21:16)
B21:7: Selected ROW(E 107 INTEGER,F 125 CHAR)
B21:16: Remote TABLE(107 INTEGER,125 CHAR) RESTView http://root:admin@localhost:8078/db/t E,F
http://root:admin@localhost:8078/db select E,F from t where E>3
--> 3 rows
B21:16: Explicit
Row(E=6,F=Six)
Row(E=4,F=Vier)
Row(E=6,F=Sechs)
```

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

Select * from w where k=1

```
GF B7:16 (E 508#B7:16,D 528#B7:16,K 549#B7:16,F 570#B7:16)
Select * from "591" where "549"=1
Transaction 4 (E 508#B7:16,D 528#B7:16,K 549#B7:16,F 570#B7:16)
B7:7: Selected ROW(E 508 INTEGER,D 528 CHAR,K 549 INTEGER,F 570 CHAR)
B7:16: Remote TABLE(508 INTEGER,528 CHAR,549 INTEGER,570 CHAR) RESTView Using VU
B7:16U: Index VU (D 232#B7:16U,K 276#B7:16U,U 297#B7:16U) match K 276=1
http://root:admin@localhost:8078/dc select E,'C' as D,1 as K,F from u
--> 3 rows
B7:16S: Explicit
Row(E=5,D=C,K=1,F=Five)
Row(E=4,D=C,K=1,F=Four)
Row(E=8,D=C,K=1,F=Ate)
```

```
SQL> select * from w where k=1
-|-|-|-----|
D|K|E|F|
-|-|-|-----|
C|1|5|Five|
C|1|4|Four|
C|1|8|Ate|
-|-|-|-----|
```

Here we see that no request was made to the first contributor.

Now the same with where conditions:

Select * from w where k>2

```
GF B25:16 (E 508#B25:16,D 528#B25:16,K 549#B25:16,F 570#B25:16)
Select * from "591" where "276">2
Transaction 24 (E 508#B25:16,D 528#B25:16,K 549#B25:16,F 570#B25:16)
B25:7: Selected ROW(E 508 INTEGER,D 528 CHAR,K 549 INTEGER,F 570 CHAR)
B25:16: Remote TABLE(508 INTEGER,528 CHAR,549 INTEGER,570 CHAR) RESTView Using VU where (K 276#B25:16U>2)
B25:16U: Index VU (D 232#B25:16U,K 276#B25:16U,U 297#B25:16U) where (K 276#B25:16U>2)
http://root:admin@localhost:8078/db select E,'B' as D,4 as K,F from t
--> 4 rows
B25:16S: Explicit
Row(E=3,D=B,K=4,F=Three)
Row(E=6,D=B,K=4,F=Six)
Row(E=4,D=B,K=4,F=Vier)
Row(E=6,D=B,K=4,F=Sechs)
```

```
SQL> Select * from w where k>2
-|-|-|-----|
E|D|K|F|
-|-|-|-----|
3|B|4|Three|
6|B|4|Six|
4|B|4|Vier|
6|B|4|Sechs|
-|-|-|-----|
```

Now let's have a filter on the remote table:

Select * from w where e=3

```
Select * from "591" where "508"=3
Transaction 26 (E 508#B27:16,D 528#B27:16,K 549#B27:16,F 570#B27:16)
B27:7: Selected ROW(E 508 INTEGER,D 528 CHAR,K 549 INTEGER,F 570 CHAR)
B27:16: Remote TABLE(508 INTEGER,528 CHAR,549 INTEGER,570 CHAR) RESTView Using VU match E 508=3
B27:16U: Index VU (D 232#B27:16U,K 276#B27:16U,U 297#B27:16U)
http://root:admin@localhost:8078/db select E,'B' as D,4 as K,F from t where E=3
--> 1 rows
http://root:admin@localhost:8078/dc select E,'C' as D,1 as K,F from u where E=3
--> 0 rows
B27:16S: Explicit
Row(E=3,D=B,K=4,F=Three)
```

```
SQL> select * from w where e=3
-|-|-|-----|
D|K|E|F|
-|-|-|-----|
B|4|3|Three|
-|-|-|-----|
SQL>
```


Here we see that the test E=3 is passed to each contributor. The above cases can be generalised: if instead of E we have an expression involving remote columns or columns from the using table, we can pass these to the remote database. On the other hand, we cannot pass where-conditions to the remote database if it references data from other tables. Finally, if the where condition has form C1 AND C2, and cannot be passed to the remote database we can consider the two subexpressions separately, as may be able to pass one of them to the remote contributor.

Aggregations

Next we look at how aggregations are handled with RESTViews.

select count(e) from w

QS B7:7 (SUM(C_12#B7:22) as COUNT COUNT)

GF B7:22 (C_12#B7:22)

B7:22: Remote ROW(C_12 INTEGER) RESTView Using VU

B7:22U: Index VU (D 232#B7:22U,K 276#B7:22U,U 297#B7:22U)

http://root:admin@localhost:8078/db select COUNT(E) as C_12 from t

--> 1 rows

http://root:admin@localhost:8078/dc select COUNT(E) as C_12 from u

--> 1 rows

B7:22S: Explicit

Row(C_12=4)

Row(C_12=3)

select count("508") from "591"

Transaction 4 (SUM(C_12#B7:22) as COUNT COUNT)

B7:7: Eval (SUM(C_12#B7:22) as COUNT COUNT)

This is the first time we have used a select list (previously we had select *) so we now also see the QS line showing the query specification. We see that the request that was sent to the contributors was select count(E) as C_79. Note also that the result has a SUM column instead of COUNT, while retaining the alias COUNT. We also see that the two partial sums from the contributors have been added together. Note that the tracing lines appear in a different order from the previous examples. This is to do with the order of construction and traversal of the rowsets.

```
SQL> select count(e) from w
-----
COUNT
-----
7
-----
```

select max(f) from w having e>4

QS B9:7 (MAX(C_30#B9:21) as MAX MAX)

GF B9:21 (C_30#B9:21)

B9:21: Remote ROW(C_30 CHAR) RESTView Using VU where (E 508#B9:21>4)

B9:21U: Index VU (D 232#B9:21U,K 276#B9:21U,U 297#B9:21U)

http://root:admin@localhost:8078/db select MAX(F) as C_30 from t where E>4

--> 1 rows

http://root:admin@localhost:8078/dc select MAX(F) as C_30 from u where E>4

--> 1 rows

B9:21S: Explicit where (E 508#B9:21>4)

Row(C_30=Six)

Row(C_30=Five)

select max("570") from "591" having "508">4

Transaction 8 (MAX(C_30#B9:21) as MAX MAX)

B9:7: Eval (MAX(C_30#B9:21) as MAX MAX)

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

```
SQL> select max(f) from w having e>4
---
MAX
---
Six
---
```

select count(*) from w having k>2

QS B11:7 (SUM(C_49#B11:23) as COUNT COUNT)

GF B11:23 (C_49#B11:23)

B11:23: Remote ROW(C_49 INTEGER) RESTView Using VU where (K 276#B11:23U>2)

B11:23U: Index VU (D 232#B11:23U,K 276#B11:23U,U 297#B11:23U) where (K 276#B11:23U>2)

http://root:admin@localhost:8078/db select COUNT(*) as C_49 from t

--> 1 rows

B11:23S: Explicit

Row(C_49=4)

```
select count(*) from "591" having "276">2
Transaction 10 (SUM(C_49#B11:23) as COUNT COUNT)
B11:7: Eval (SUM(C_49#B11:23) as COUNT COUNT)
We see the request just goes to one contributor.
```

```
SQL> select count(*) from w having k>2
|----|
|COUNT|
|----|
|4      |
|----|
SQL>
```

The following examples have trivial results here, but the strategy trace shows how the method will work in more complex cases.

```
select sum(e),char_length(f) as x from w group by x
QS B7:7 (SUM(C_12#B7:41) as SUM SUM,X@53#B7:41) groups [GROUP X]
GF B7:41 (C_12#B7:41,X@53#B7:41)
select sum("508"),char_length("570") as x from "591" group by x
Transaction 4 (SUM(C_12#B7:41) as SUM SUM,X@53#B7:41)
B7:7: Grouping (SUM(C_12#B7:41) as SUM SUM,X@53#B7:41) groups [GROUP X]
B7:41: Remote ROW(C_12 INTEGER,X INTEGER) RESTView Using VU
B7:41U: Index VU (D 232#B7:41U,K 276#B7:41U,U 297#B7:41U)
http://root:admin@localhost:8078/db select SUM(E) as C_12,CHAR_LENGTH(F) as X from t group by X
--> 3 rows
http://root:admin@localhost:8078/dc select SUM(E) as C_12,CHAR_LENGTH(F) as X from u group by X
--> 2 rows
B7:41S: Explicit
Row(C_12=6,X=3)
Row(C_12=4,X=4)
Row(C_12=9,X=5)
Row(C_12=8,X=3)
Row(C_12=9,X=4)
```

```
SQL> select sum(e),char_length(f) as x from w group by x
|---|---|
|SUM|X|
|---|---|
|14|3|
|13|4|
|9 |5|
|---|---|
```

Grouping by a non-remote formula:

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

```
QS B9:7 (SUM(C_49#B9:33) as COUNT COUNT,K2@101#B9:33) groups [GROUP K2]
GF B9:33 (C_49#B9:33,K2@101#B9:33)
select count(*),"549"/2 as k2 from "591" group by k2
Transaction 8 (SUM(C_49#B9:33) as COUNT COUNT,K2@101#B9:33)
B9:7: Grouping (SUM(C_49#B9:33) as COUNT COUNT,K2@101#B9:33) groups [GROUP K2]
B9:33: Remote ROW(C_49 INTEGER,K2 UNION(INTEGER,NUMERIC,REAL)) RESTView Using VU
B9:33U: Index VU (D 232#B9:33U,K 276#B9:33U,U 297#B9:33U)
http://root:admin@localhost:8078/db select COUNT(*) as C_49,(4/2) as K2 from t group by K2
--> 1 rows
http://root:admin@localhost:8078/dc select COUNT(*) as C_49,(1/2) as K2 from u group by K2
--> 1 rows
B9:33S: Explicit
Row(C_49=4,K2=2)
Row(C_49=3,K2=0)
```

```
SQL> select count(*),k/2 as k2 from w group by k2
|----|---|
|COUNT|K2|
|----|---|
|3      |0 |
|4      |2 |
|----|---|
```

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.

Select avg(e) from w

```
QS B7:7 (((SUM(C_12#B7:20S)*1)/SUM(D_12#B7:20)) as AVG AVG)
GF B7:20 (C_12#B7:20S,D_12#B7:20)
B7:20: Remote ROW(C_12 UNION(INTEGER,NUMERIC,REAL),D_12 INTEGER) RESTView Using VU
B7:20U: Index VU (D 232#B7:20U,K 276#B7:20U,U 297#B7:20U)
http://root:admin@localhost:8078/db select SUM(E) as C_12,COUNT(E) as D_12 from t
--> 1 rows
http://root:admin@localhost:8078/dc select SUM(E) as C_12,COUNT(E) as D_12 from u
--> 1 rows
B7:20S: Explicit
Row(C_12=19,D_12=4)
Row(C_12=17,D_12=3)
Select avg("508") from "591"
Transaction 4 (((SUM(C_12#B7:20S)*1)/SUM(D_12#B7:20)) as AVG AVG)
B7:7: Eval (((SUM(C_12#B7:20S)*1)/SUM(D_12#B7:20)) as AVG AVG)
```

```
SQL> Select avg(e) from w
|-----|
|AVG     |
|-----|
|5.14285714285714|
|-----|
```


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.

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

```

QS B9:7 ((SUM(C_35#B9:30)*SUM(C_35#B9:30)) as C_38 C_38,D 528#B9:30) groups [GROUP D
232]
GF B9:30 (C_35#B9:30,D 528#B9:30)
select sum("508")*sum("508"),"528" from "591" group by "232"
Transaction 8 ((SUM(C_35#B9:30)*SUM(C_35#B9:30)) as C_38 C_38,D 528#B9:30)
B9:7: Grouping ((SUM(C_35#B9:30)*SUM(C_35#B9:30)) as C_38 C_38,D 528#B9:30) groups [GROUP D 232]
B9:30: Remote ROW(C_35 INTEGER,528 CHAR) RESTView Using VU
B9:30U: Index VU (D 232#B9:30U,K 276#B9:30U,U 297#B9:30U)
http://root:admin@localhost:8078/db select SUM(E) as C_35,'B' as D from t group by D
--> 1 rows
http://root:admin@localhost:8078/dc select SUM(E) as C_35,'C' as D from u group by D
--> 1 rows
B9:30S: Explicit
Row(C_35=19,D=B)
Row(C_35=17,D=C)

```

```

SQL> select sum(e)*sum(e),d from w group by d
-----|-----|
C_38   |D       |
-----|-----|
361    |B       |
289    |C       |
-----|-----|

```

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.

`Select f,n from w natural join m`

```

QS B43:7 (F 570#B43:18,N 676#B43:32)
GF B43:18 (F 570#B43:18,E 508#B43:18)
Select "570","676" from "591" natural join "625"
Transaction 42 (F 570#B43:18,N 676#B43:32)
B43:7: Selected ROW(F 570 CHAR,N 676 CHAR)
B43:15: Join FD W primary M
B43:18: Remote ROW(570 CHAR,508 INTEGER) RESTView Using VU
B43:18U: Index VU (D 232#B43:18U,K 276#B43:18U,U 297#B43:18U)
http://root:admin@localhost:8078/db select F,E from t
--> 4 rows
http://root:admin@localhost:8078/dc select F,E from u
--> 3 rows
B43:18S: Explicit
Row(F=Three,E=3)
Row(F=Six,E=6)
Row(F=Vier,E=4)
Row(F=Sechs,E=6)
Row(F=Five,E=5)
Row(F=Four,E=4)
Row(F=Ate,E=8)

```

```

SQL> Select f,n from w natural join m
-----|-----|
F       |N       |
-----|-----|
Three   |Trois   |
Six     |Six     |
Vier    |Quatre  |
Sechs   |Six     |
Five    |Cinq    |
Four    |Quatre  |
-----|-----|

```

We see that 7 rows have been returned from the RESTView, but only 6 rows in the join. This is correct, as the join has been done on the local table m.

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:

`select e+char_length(f) as x,n from w natural join m (both terms in the expression are remote)`

```

QS B45:7 ((E 508#B45:38+CHAR_LENGTH(F 570#B45:38) as CHAR_LENGTH) as X X@615,N 676#B45:52)
GF B45:38 (E 508#B45:38,F 570#B45:38)
select "508"+char_length("570") as x,"676" from "591" natural join "625"
Transaction 44 ((E 508#B45:38+CHAR_LENGTH(F 570#B45:38) as CHAR_LENGTH) as X X@615,N 676#B45:52)
B45:7: Selected ROW(X@615 INTEGER,N 676 CHAR)

```

```

B45:35: Join FD W primary M
B45:38: Remote ROW(508 INTEGER,570 CHAR) RESTView Using VU
B45:38U: Index VU (D 232#B45:38U,K 276#B45:38U,U 297#B45:38U)
http://root:admin@localhost:8078/db select E,F from t
--> 4 rows
http://root:admin@localhost:8078/dc select E,F from u
--> 3 rows
B45:38S: Explicit
Row(E=3,F=Three)
Row(E=6,F=Six)
Row(E=4,F=Vier)
Row(E=6,F=Sechs)
Row(E=5,F=Five)
Row(E=4,F=Four)
Row(E=8,F=Ate)

```

```

SQL> select e+char_length(f) as x,n from w natural join m
|----|
|x|n|
|----|
|8|Trois|
|9|Six|
|8|Quatre|
|11|Six|
|9|Cinq|
|8|Quatre|
|----|

```

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

`select char_length(f)+char_length(n) from w natural join m` (one term is remote)

```

QS B13:7 ((CHAR_LENGTH(F 570#B13:44) as CHAR_LENGTH+CHAR_LENGTH(N 676#B13:58)) as C_107 C_107)
GF B13:44 (F 570#B13:44,E 508#B13:44)
select char_length("570")+char_length("676") from "591" natural join "625"
Transaction 12 ((CHAR_LENGTH(F 570#B13:44) as CHAR_LENGTH+CHAR_LENGTH(N 676#B13:58)) as C_107 C_107)
B13:7: Selected ROW(C_107 INTEGER)
B13:41: Join FD W primary M
B13:44: Remote ROW(570 CHAR,508 INTEGER) RESTView Using VU
B13:44U: Index VU (D 232#B13:44U,K 276#B13:44U,U 297#B13:44U)
http://root:admin@localhost:8078/db select F,E from t
--> 4 rows
http://root:admin@localhost:8078/dc select F,E from u
--> 3 rows

```

```

SQL> select char_length(f)+char_length(n) from w natural join m
|----|
|C_107|
|----|
|10|
|6|
|10|
|8|
|8|
|10|
|----|

```

```

B13:44S: Explicit
Row(F=Three,E=3)
Row(F=Six,E=6)
Row(F=Vier,E=4)
Row(F=Sechs,E=6)
Row(F=Five,E=5)
Row(F=Four,E=4)
Row(F=Ate,E=8)

```

`select sum(e)+char_length(max(f)) from w` (both aggregations remote)

```

QS B11:7 ((SUM(C_75#B11:40)+CHAR_LENGTH(MAX(C_77#B11:40) as C_77) as CHAR_LENGTH) as C_79 C_79)
GF B11:40 (C_75#B11:40,C_77#B11:40)
B11:40: Remote ROW(C_75 INTEGER,C_77 CHAR) RESTView Using VU
B11:40U: Index VU (D 232#B11:40U,K 276#B11:40U,U 297#B11:40U)
http://root:admin@localhost:8078/db select SUM(E) as C_75,MAX(F) as C_77 from t
--> 1 rows
http://root:admin@localhost:8078/dc select SUM(E) as C_75,MAX(F) as C_77 from u
--> 1 rows
B11:40S: Explicit
Row(C_75=19,C_77=Vier)
Row(C_75=17,C_77=Four)
select sum("508")+char_length(max("570")) from "591"
Transaction 10 ((SUM(C_75#B11:40)+CHAR_LENGTH(MAX(C_77#B11:40) as C_77) as CHAR_LENGTH) as C_79 C_79)
B11:7: Eval ((SUM(C_75#B11:40)+CHAR_LENGTH(MAX(C_77#B11:40) as C_77) as CHAR_LENGTH) as C_79 C_79)

```

```

SQL> select sum(e)+char_length(max(f)) from w
|----|
|C_79|
|----|
|48|
|----|

```

`select count(*),e+char_length(f) as x from w group by x`

```

QS B15:7 (SUM(C_125#B15:45) as COUNT COUNT,X@204#B15:45) groups [GROUP X]
GF B15:45 (C_125#B15:45,X@204#B15:45)
select count(*),"508"+char_length("570") as x from "591" group by x
Transaction 14 (SUM(C_125#B15:45) as COUNT COUNT,X@204#B15:45)
B15:7: Grouping (SUM(C_125#B15:45) as COUNT COUNT,X@204#B15:45) groups [GROUP X]
B15:45: Remote ROW(C_125 INTEGER,X INTEGER) RESTView Using VU
B15:45U: Index VU (D 232#B15:45U,K 276#B15:45U,U 297#B15:45U)
http://root:admin@localhost:8078/db select COUNT(*) as C_125,(E+CHAR_LENGTH(F)) as X from t group by X
--> 3 rows
http://root:admin@localhost:8078/dc select COUNT(*) as C_125,(E+CHAR_LENGTH(F)) as X from u group by X
--> 3 rows

```

```
B15:45S: Explicit
Row(C_125=2,X=8)
Row(C_125=1,X=9)
Row(C_125=1,X=11)
Row(C_125=1,X=8)
Row(C_125=1,X=9)
Row(C_125=1,X=11)
```

```
SQL> select count(*),e+char_length(f) as x from w group by x
|-----|
|COUNT|X|
|-----|
|3      |8|
|2      |9|
|2      |11|
|-----|
```

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.

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

```
QS B17:7 ((SUM(C_167#B17:45) as COUNT COUNT,(E 508#B17:45+CHAR_LENGTH(N 676#B17:60)) as X X@254) groups [GROUP X]
GF B17:45 (C_167#B17:45,E 508#B17:45)
select count(*),"508"+char_length("676") as x from "591" natural join "625" group by x
Transaction 16 ((SUM(C_167#B17:45) as COUNT COUNT,(E 508#B17:45+CHAR_LENGTH(N 676#B17:60)) as X X@254)
B17:7: Grouping ((SUM(C_167#B17:45) as COUNT COUNT,(E 508#B17:45+CHAR_LENGTH(N 676#B17:60)) as X X@254) groups [GROUP X]
B17:42: Join FD W primary M
B17:45: Remote ROW(C_167 INTEGER,508 INTEGER) RESTView Using VU
B17:45U: Index VU (D 232#B17:45U,K 276#B17:45U,U 297#B17:45U)
http://root:admin@localhost:8078/db select COUNT(*) as C_167,E from t group by E
--> 3 rows
http://root:admin@localhost:8078/dc select COUNT(*) as C_167,E from u group by E
--> 3 rows
B17:45S: Explicit
Row(C_167=1,E=3)
Row(C_167=1,E=4)
Row(C_167=2,E=6)
Row(C_167=1,E=4)
Row(C_167=1,E=5)
Row(C_167=1,E=8)
```

```
SQL> select count(*),e+char_length(n) as x from w natural join m group by x
|-----|
|COUNT|X|
|-----|
|1      |8|
|3      |9|
|2      |10|
|-----|
```

select sum(e)+char_length(f),f from w natural join m group by f

```
QS B19:7 ((SUM(C_206#B19:39)+CHAR_LENGTH(F 570#B19:39)) as C_209 C_209,F 570#B19:39) groups [GROUP F 570]
GF B19:39 (C_206#B19:39,F 570#B19:39,E 508#B19:39)
select sum("508")+char_length("570"),"570" from "591" natural join "625" group by "570"
Transaction 18 ((SUM(C_206#B19:39)+CHAR_LENGTH(F 570#B19:39)) as C_209 C_209,F 570#B19:39)
B19:7: Grouping ((SUM(C_206#B19:39)+CHAR_LENGTH(F 570#B19:39)) as C_209 C_209,F 570#B19:39) groups [GROUP F 570]
B19:36: Join FD W primary M
B19:39: Remote ROW(C_206 INTEGER,570 CHAR,508 INTEGER) RESTView Using VU
B19:39U: Index VU (D 232#B19:39U,K 276#B19:39U,U 297#B19:39U)
http://root:admin@localhost:8078/db select SUM(E) as C_206,F,E from t group by F,E
--> 4 rows
http://root:admin@localhost:8078/dc select SUM(E) as C_206,F,E from u group by F,E
--> 3 rows
B19:39S: Explicit
Row(C_206=6,F=Sechs,E=6)
Row(C_206=6,F=Six,E=6)
Row(C_206=3,F=Three,E=3)
Row(C_206=4,F=Vier,E=4)
Row(C_206=8,F=Ate,E=8)
Row(C_206=5,F=Five,E=5)
Row(C_206=4,F=Four,E=4)
```

```
SQL> select sum(e)+char_length(f),f from w natural join m group by f
|-----|
|C_209|F|
|-----|
|9     |Five|
|8     |Four|
|11    |Sechs|
|9     |Six|
|8     |Three|
|8     |Vier|
|-----|
```

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

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

```
QS B21:7 ((SUM(C_281#B21:56)+CHAR_LENGTH(N 676#B21:71)) as X X@366,N 676#B21:71) groups [GROUP N 676]
GF B21:56 (C_281#B21:56,E 508#B21:56)
select sum(char_length("570")+char_length("676") as x,"676" from "591" natural join "625" group by "676"
Transaction 20 ((SUM(C_281#B21:56)+CHAR_LENGTH(N 676#B21:71)) as X X@366,N 676#B21:71)
B21:7: Grouping ((SUM(C_281#B21:56)+CHAR_LENGTH(N 676#B21:71)) as X X@366,N 676#B21:71) groups [GROUP N 676]
B21:53: Join FD W primary M
B21:56: Remote ROW(C_281 INTEGER,508 INTEGER) RESTView Using VU
B21:56U: Index VU (D 232#B21:56U,K 276#B21:56U,U 297#B21:56U)
http://root:admin@localhost:8078/db select SUM(CHAR_LENGTH(F)) as C_281,E from t group by E
--> 3 rows
http://root:admin@localhost:8078/dc select SUM(CHAR_LENGTH(F)) as C_281,E from u group by E
--> 3 rows
B21:56S: Explicit
Row(C_281=5,E=3)
Row(C_281=4,E=4)
Row(C_281=8,E=6)
Row(C_281=4,E=4)
Row(C_281=4,E=5)
Row(C_281=3,E=8)
```

```
SQL> select sum(char_length(f))+char_length(n) as x,n from w natural join m group by n
|---|-----|
|X|N|
|---|-----|
|8|Cinq|
|14|Quatre|
|11|Six|
|10|Trois|
|---|-----|
```

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.

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 $G \cup J$. 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.

Select count(*) from w natural join m

```
QS B23:7 (SUM(C_337#B23:23) as COUNT COUNT)
GF B23:23 (C_337#B23:23,E 508#B23:23)
B23:20: Join FD W primary M
B23:23: Remote ROW(C_337 INTEGER,508 INTEGER) RESTView Using VU
B23:23U: Index VU (D 232#B23:23U,K 276#B23:23U,U 297#B23:23U)
http://root:admin@localhost:8078/db select COUNT(*) as C_337,E from t group by E
--> 3 rows
http://root:admin@localhost:8078/dc select COUNT(*) as C_337,E from u group by E
--> 3 rows
B23:23S: Explicit
Row(C_337=1,E=3)
Row(C_337=1,E=4)
Row(C_337=2,E=6)
Row(C_337=1,E=4)
Row(C_337=1,E=5)
Row(C_337=1,E=8)
Select count(*) from "591" natural join "625"
Transaction 22 (SUM(C_337#B23:23) as COUNT COUNT)
B23:7: Eval (SUM(C_337#B23:23) as COUNT COUNT)
```

```
SQL> Select count(*) from w natural join m
|-----|
| COUNT |
|-----|
| 6      |
|-----|
```

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

```
GF B63:13 (E 107#B63:13,F 125#B63:13)
PUT http://root:admin@localhost:8078/db/t/E=3 {"F": "Tri"}
update "144" set "125"='Tri' where "107"=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 update w set f='Eight' where e=8:

```
GF B67:13 (E 508#B67:13,D 528#B67:13,K 549#B67:13,F 570#B67:13)
PUT http://root:admin@localhost:8078/db/t/E=8 {"F": "Eight"}
PUT http://root:admin@localhost:8078/dc/u/E=8 {"F": "Eight"}
update "591" set "570"='Eight' where "508"=8)
```

```
SQL> update v set F='Tri' where E=3
0 records affected in rv
SQL> table w
|-----|
| E | D | K | F |
|-----|
| 3 | B | 4 | Tri |
| 6 | B | 4 | Six |
| 4 | B | 4 | Vier |
| 6 | B | 4 | Sechs |
| 5 | C | 1 | Five |
| 4 | C | 1 | Four |
| 8 | C | 1 | Ate |
```

Operations such as Insert into v values(9,'Nine') and delete from v where e=9 are also supported.

```
SQL-T>insert into V values (9,'Nine')
0 records affected in rv
SQL-T>table v
|----|
|E|F|
|----|
|3|Tri|
|6|Six|
|9|Nine|
|----|
SQL-T>delete from V where E=9
0 records affected in rv
SQL-T>table v
|---|
|E|F|
|---|
|3|Tri|
|6|Six|
|---|
```

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.

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

Appendix

1. 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')
```

2. 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
```

3. 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$"
```