Develop reports efficiently with esProc

1 Abstract

Report-developing is the essential module of many web application systems, and it’s still the basic function of most of the BI projects. With the coming of data era, the data comes from increasingly diverse sources (text, excel, monogdb, redis, es...), which brings great challenges for the data preparation of reports. Traditionally, it is better to first put the data out of the database into the database, and then use the computing power of the database (writing SQL or stored procedures) to prepare the data for the report. Due to the dependence on pre-import, the real-time performance of reports can not be guaranteed, and the development process of reports has also been lengthened. With the increasing demand for reports, the database has become more and more bloated, and the management cost has been rising.

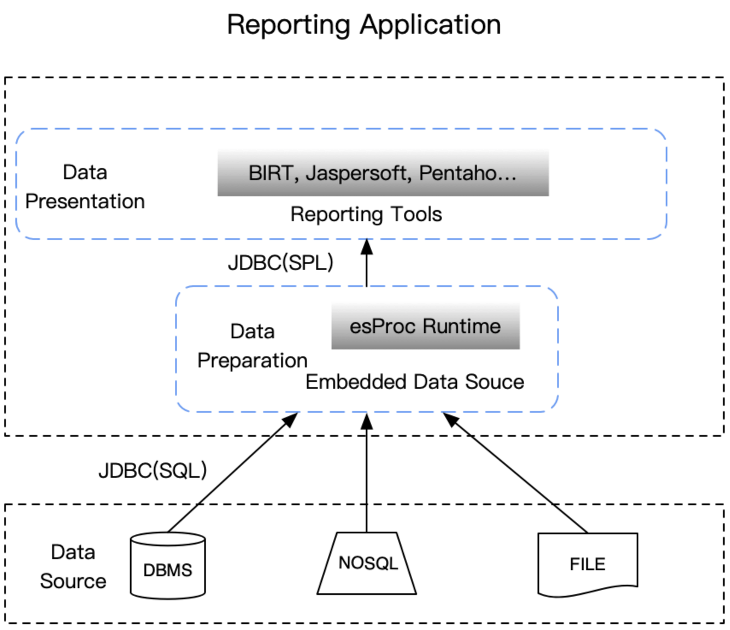
But if we use these data directly for report development, it is always time-consuming and laborious, and the performance of the final report is often not good enough. As programmers who are familiar with report development know, reporting tools provide only a few simple capabilities for computing such out-of-database data. When the computational requirements are complex, they need to be extended out of reporting tools and implemented in a user-defined way. This kind of computation is usually implemented in high-level language (Java, .net) by hard-coding, where strong data programming experience is required, often beyond the capabilities of ordinary report developers. The development cost of high-level language implementation is very high, and it is not reusable.

Even when data is in a database, computing complex reports often requires advanced window functions or stored procedure, which are exactly the shortcomings of open source databases（mysql, hive…）. It is much better for commercial databases in this respect, but the task is still not easy to implement. Usually, advanced extensions of SQL are needed. Since different database vendors have different SQL extensions, experts who are proficient with certain databases are neededed to complete the task. It's not easy to get familiar with all kinds of databases and become a master of SQL. How can ordinary report developers solve these problems easily in the same way? In addition, stored procedures and databases are coupled together, which will bring huge overhead to database operation and troublesome maintenance.

If there is a computing engine with the same computing power as the database, without importing data, directly calculating data from various sources, providing general high-level window functions and stored procedures, preparing data independently for reports, and solving the above problems, it will greatly improve the development progress and application effect of reports.

The above mentioned is the original intention of the design of the integrated version of esProc. Here, by introducing its application structure, esProc’s ease of use is illustrated with examples, and the practical value of esProc in the application scenario of the report is demonstrated with the open source reporting tools.

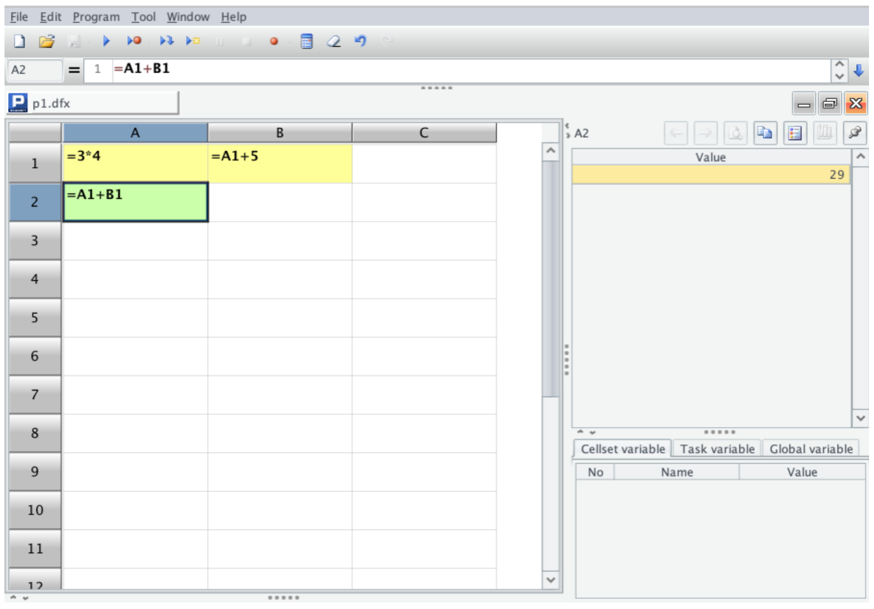
2 Application architecture



Just like when database acts as the data source of reporting tools, esProc can also be the data source for reporting tools through JDBC interface when it is embedded in the reporting tools. Only that SQL is excecuted when database is connected and SPL (esProc script language) is excecuted when esProc is connected to the reporting tools.

3 Development environment

Cell style coding



A2 = 1 =A1+B1

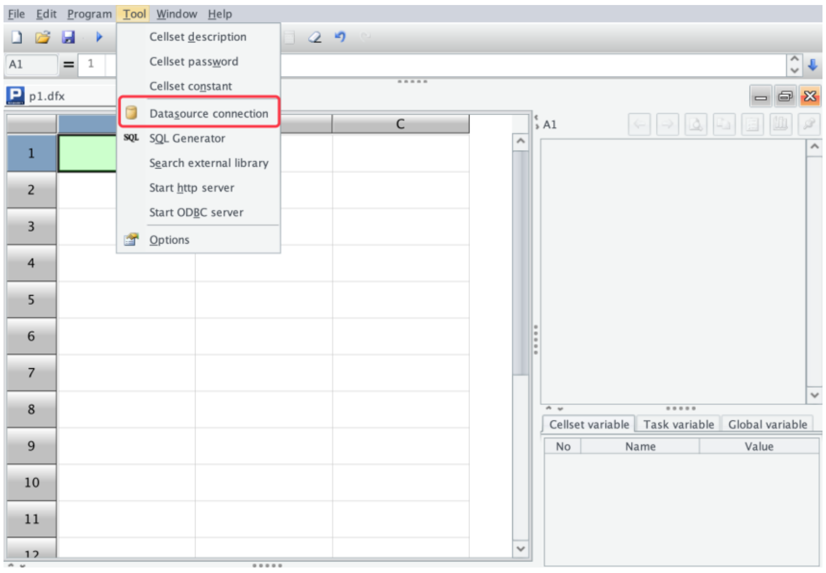
	A	B	C
1	=3*4	=A1+5	
2	=A1+B1		
3			
4			
5			
6			
7			
8			
9			
10			
11			

A2

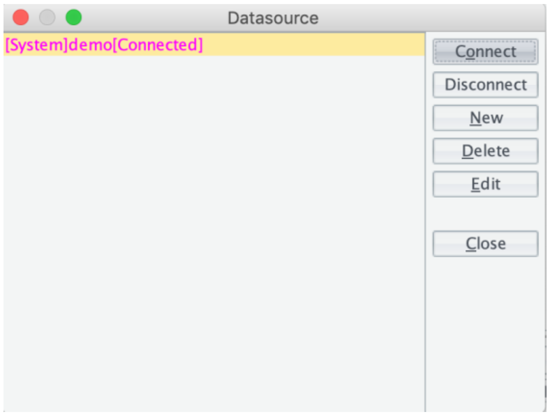
Value
29

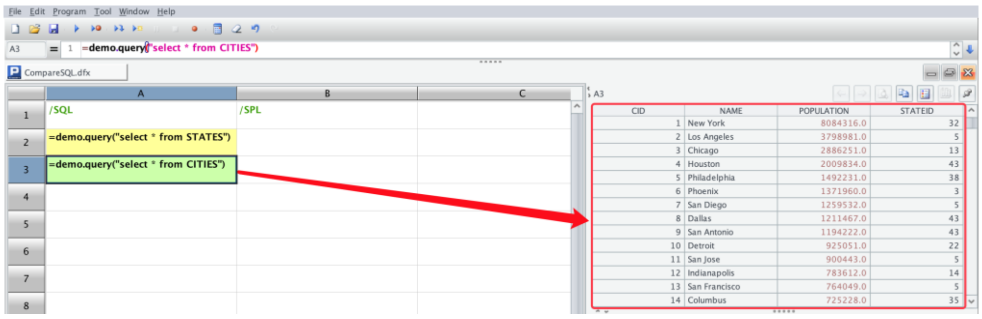
In esProc, the coding language is called SPL (Structured Process Language). The SPL script is written in cells, just like in Excel. The result of SPL execution within a cell is assigned to the cell address, and cell addresses are subsequently referred to directly as variable names. The codes are in cell order, first from left to right, then from top to bottom.

Datasource connection



Connect with database (the built-in HSQL sample database), which has the name **demo.**





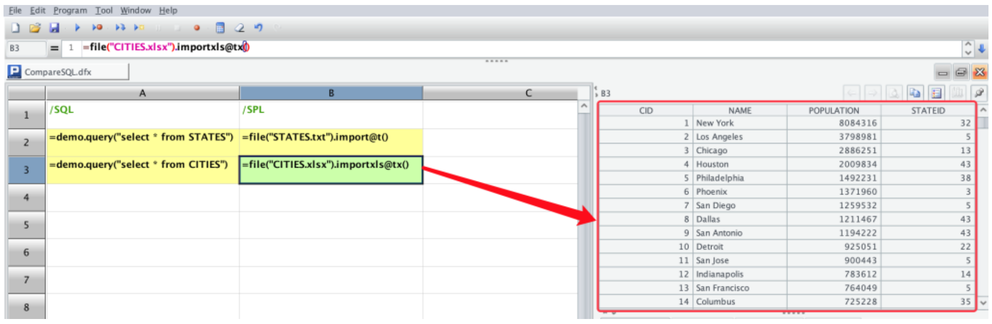
A3 =demo.query("select * from CITIES")

	A	B	C
1	/SQL	/SPL	
2	=demo.query("select * from STATES")		
3	=demo.query("select * from CITIES")		

A3

CID	NAME	POPULATION	STATEID
1	New York	8084316.0	32
2	Los Angeles	3798981.0	5
3	Chicago	2886251.0	13
4	Houston	2009834.0	43
5	Philadelphia	1492231.0	38
6	Phoenix	1371960.0	3
7	San Diego	1259532.0	5
8	Dallas	1211467.0	43
9	San Antonio	1194222.0	43
10	Detroit	925051.0	22
11	San Jose	900443.0	5
12	Indianapolis	783612.0	14
13	San Francisco	764049.0	5
14	Columbus	725228.0	35

From the demo database, import all records of the STATES table and CITES table.



B3 =file("CITIES.xlsx").importxls@tx()

	A	B	C
1	/SQL	/SPL	
2	=demo.query("select * from STATES")	=file("STATES.txt").import@t()	
3	=demo.query("select * from CITIES")	=file("CITIES.xlsx").importxls@tx()	

B3

CID	NAME	POPULATION	STATEID
1	New York	8084316	32
2	Los Angeles	3798981	5
3	Chicago	2886251	13
4	Houston	2009834	43
5	Philadelphia	1492231	38
6	Phoenix	1371960	3
7	San Diego	1259532	5
8	Dallas	1211467	43
9	San Antonio	1194222	43
10	Detroit	925051	22
11	San Jose	900443	5
12	Indianapolis	783612	14
13	San Francisco	764049	5
14	Columbus	725228	35

STATES.txt and CITIES.xlsx are tables exported from demo database in advance. Except for the different ways of data acquisition, the results of A2 and B2, A3 and B3 are the same. Once the data is loaded into esProc from various data sources, it becomes the data object inside the esProc, which makes no difference in use.

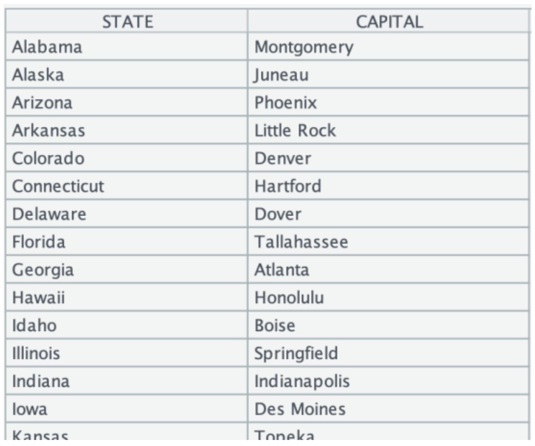
SPL Syntax

Report developers are familiar with SQL. Here are a few simple examples to get a preliminary understanding of SPL syntax.

①Select

	A	B
1		**=file("STATES.txt").import@t()**
2	**=demo.query("select NAME as STATE,CAPITAL from STATES")**	**=B1.new(NAME:STATE,CAPITAL)**

Select the required fields from the data table, and the results in A2 and B2 are the same:

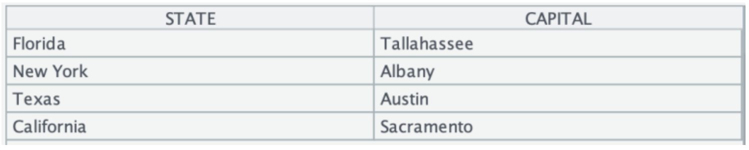


STATE	CAPITAL
Alabama	Montgomery
Alaska	Juneau
Arizona	Phoenix
Arkansas	Little Rock
Colorado	Denver
Connecticut	Hartford
Delaware	Dover
Florida	Tallahassee
Georgia	Atlanta
Hawaii	Honolulu
Idaho	Boise
Illinois	Springfield
Indiana	Indianapolis
Iowa	Des Moines
Kansas	Topeka

②Filter

	A	B
1		**=file("STATES.txt").import@t()**
2	**=demo.query("select NAME as STATE,CAPITAL from STATES where POPULATION>15000000")**	**=B1.select(POPULATION>15000000).new(NAME:STATE ,CAPITAL)**

In A2 and B2, the data of states with a population of more than 15,000,000 are selected and the results are the same:

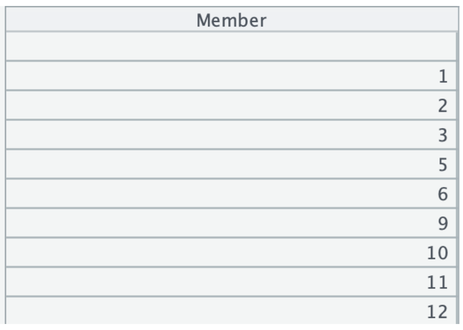


STATE	CAPITAL
Florida	Tallahassee
New York	Albany
Texas	Austin
California	Sacramento

③Distinct

	A	B
1		**=file("CITIES.xlsx").importxls@tx()**
2	**=demo.query("select distinct STATEID from CITIES")**	**=B1.id(STATEID)**

In A2 and B2, the distinct STATEID of all the cities in the **CITIES** table is acquired and the results are the same:

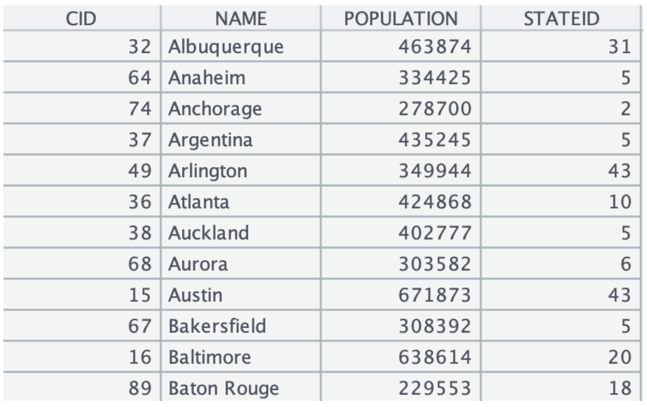


Member
1
2
3
5
6
9
10
11
12

④Order by

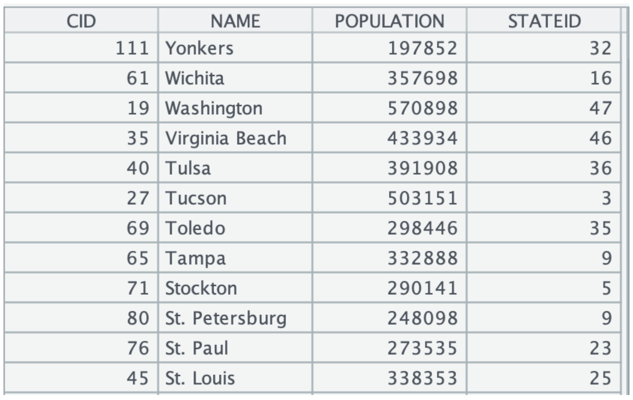
	A	B
1		=file("CITIES.xlsx").importxls@tx()
2	=demo.query("select * from CITIES order by NAME")	=B1.sort(NAME)
3	=demo.query("select * from CITIES order by NAME desc")	=B1.sort(NAME:-1)

in A2 and B2, sort data by city name in ascending order:



CID	NAME	POPULATION	STATEID
32	Albuquerque	463874	31
64	Anaheim	334425	5
74	Anchorage	278700	2
37	Argentina	435245	5
49	Arlington	349944	43
36	Atlanta	424868	10
38	Auckland	402777	5
68	Aurora	303582	6
15	Austin	671873	43
67	Bakersfield	308392	5
16	Baltimore	638614	20
89	Baton Rouge	229553	18

in A3 and B3, sort data by city name in descending order:

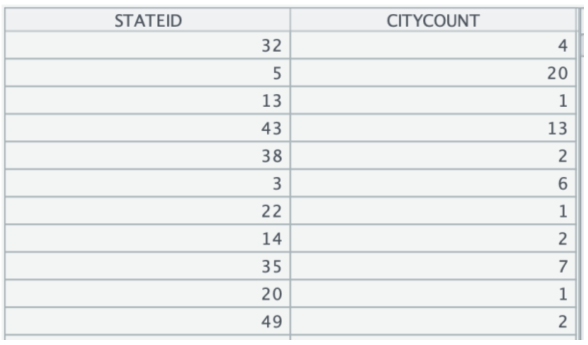


CID	NAME	POPULATION	STATEID
111	Yonkers	197852	32
61	Wichita	357698	16
19	Washington	570898	47
35	Virginia Beach	433934	46
40	Tulsa	391908	36
27	Tucson	503151	3
69	Toledo	298446	35
65	Tampa	332888	9
71	Stockton	290141	5
80	St. Petersburg	248098	9
76	St. Paul	273535	23
45	St. Louis	338353	25

⑤Group and aggregation

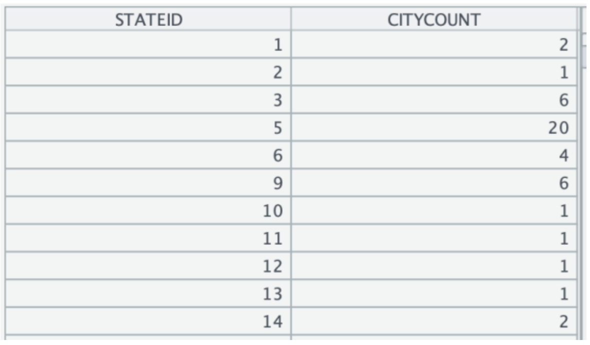
	A	B
1		=file("CITIES.xlsx").importxls@tx()
2	=demo.query("select STATEID, count(*) as CITYCOUNT from CITIES group by STATEID")	=B1.groups(STATEID;count(~):CITYCOUNT)

The SQL in A2 groups and counts the total number of cities of each state in **CITIES** table, and the result is as follows:



STATEID	CITYCOUNT
32	4
5	20
13	1
43	13
38	2
3	6
22	1
14	2
35	7
20	1
49	2

The ***groups*** function in B2 divides the data in the target table into groups. Semicolon is the function level separator, separating different types of parameters (parameters of the same type are separated by comma). ~ is wildcard, just like * in SQL, and here ~ represents a row of data abstractly. ***count()*** is aggregation function. The result is as follows:



STATEID	CITYCOUNT
1	2
2	1
3	6
5	20
6	4
9	6
10	1
11	1
12	1
13	1
14	2

The aggregation result is the same by the two types of methods, but in esProc sorting according to the grouping value is automatically achieved during the group and aggregation.

4 Integration in reporting tools

esProc can be easily integrated in reporting tools like BIRT or JasperReport. Below we'll introduce the integration process briefly, taking integration in BIRT as an example. For integration details,please refer to the documents listed on Raqsoft official website. The process of integration with other reporting tools or BI systems is similar.

Load JDBC drivers

Copy 3 esProc basic jars from esProc[installation directory]\esProc\lib to Birt[installation directory]\plugins\org.eclipse.birt.report.data.oda.jdbc_4.6.0.v20160607212\driver. Note: The upper directory names of driver are slightly different for different BIRT versions.

dm.jar	*esProc computing engine and JDBC drivers*

icu4j_3_4_5.jar	*Handling internationalization*
jdom.jar	*Parsing configuration files*

Load config file

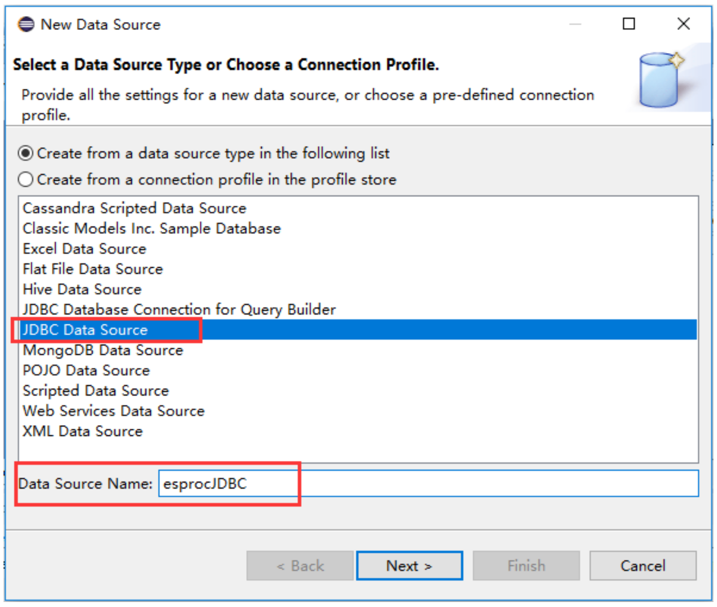
Copy raqsoftConfig.xml from [installation directory]\esProc\config to Birt[installation directory]\plugins\org.eclipse.birt.report.data.oda.jdbc_4.6.0.v20160607212\driver. The raqsoftConfig.xml file includes information like license, esProc main path, search path for dfx files, etc.

Open raqsoftConfig.xml, configure the license information.

```
…
    <Esproc>

            <license>esproc.xml</license>

             <!—Business license or trial license, trial license can be downloaded from official webset-->
         <!—License file path, which can be absolute path or relative path,the relative path is relative to the classpath-->


      </Esproc>

…
```

Add new datasource

Add new report, select “JDBC Data Source” under “DataSources”, and input “esprocJDBC” under “Data Source Name”.



New Data Source

Select a Data Source Type or Choose a Connection Profile.

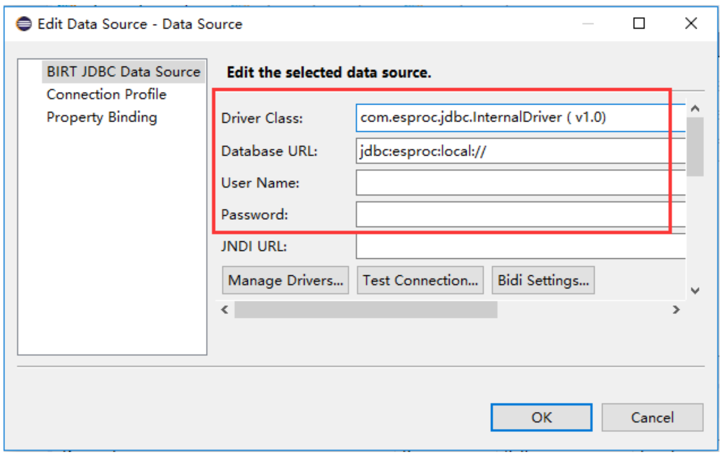
Provide all the settings for a new data source, or choose a pre-defined connection profile.

(●) Create from a data source type in the following list
( ) Create from a connection profile in the profile store

Cassandra Scripted Data Source
Classic Models Inc. Sample Database
Excel Data Source
Flat File Data Source
Hive Data Source
JDBC Database Connection for Query Builder
JDBC Data Source
MongoDB Data Source
POJO Data Source
Scripted Data Source
Web Services Data Source
XML Data Source

Data Source Name: esprocJDBC

Click【Next】, select **com.esproc.jdbc.InternalDriver (v1.0)** in the drop-down list of the Driver Class and enter the database URL **jdbc:esproc:local://**. Leave both the user name and the password blank.


Edit Data Source - Data Source

Edit the selected data source.

Driver Class: com.esproc.jdbc.InternalDriver (v1.0)

Database URL: jdbc:esproc:local://

User Name:

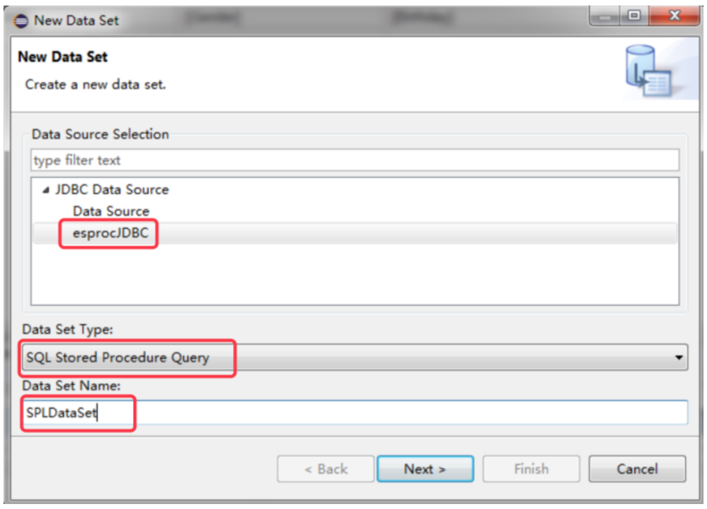
Password:

JNDI URL:

Click【Test Connection】, if "Connection successful" is prompted, then esprocJDBC is successfully connected. Then click【OK】, the new datasource is successfully created.

Add new dataset

Create a new data set by selecting the data source just configured. The data set type is **SQL stored procedure Query**. The name of data set is **SPLDataSet.**


New Data Set

Create a new data set.

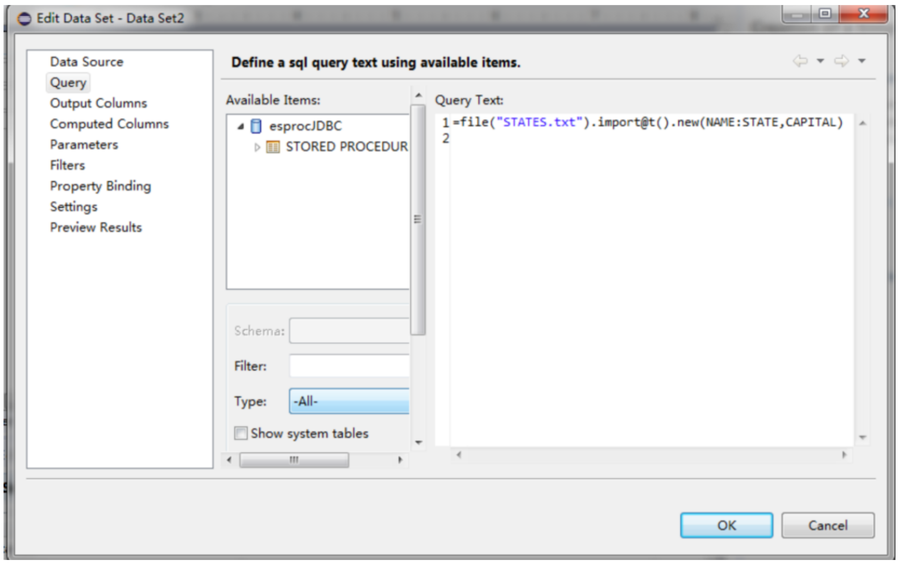
Data Source Selection

JDBC Data Source
Data Source
esprocJDBC

Data Set Type: SQL Stored Procedure Query

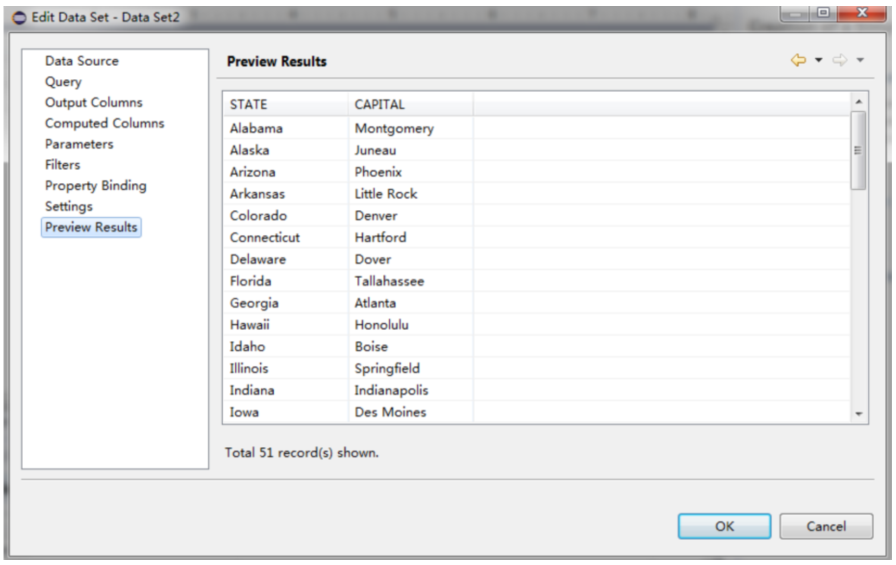
Data Set Name: SPLDataSet

Click【Next】, then define a query text. Combine the SPL codes in B1 and B2 of example① into one sentence, and enter as query text.



```
=file("STATES.txt").import@t().new(NAME:STATE,CAPITAL)
```

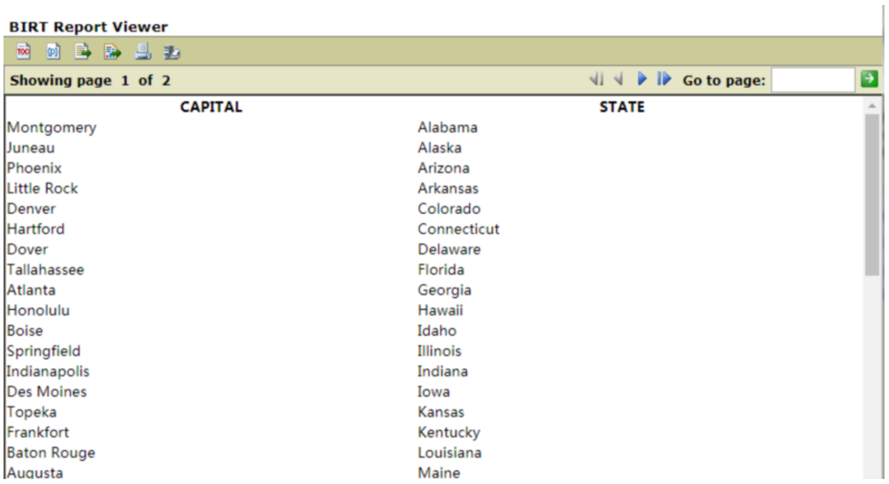
Click on "Preview Results", and the results are displayed as follows:



STATE	CAPITAL
Alabama	Montgomery
Alaska	Juneau
Arizona	Phoenix
Arkansas	Little Rock
Colorado	Denver
Connecticut	Hartford
Delaware	Dover
Florida	Tallahassee
Georgia	Atlanta
Hawaii	Honolulu
Idaho	Boise
Illinois	Springfield
Indiana	Indianapolis
Iowa	Des Moines

Total 51 record(s) shown.

Call the data set in a BIRT report, and the result is as follows:

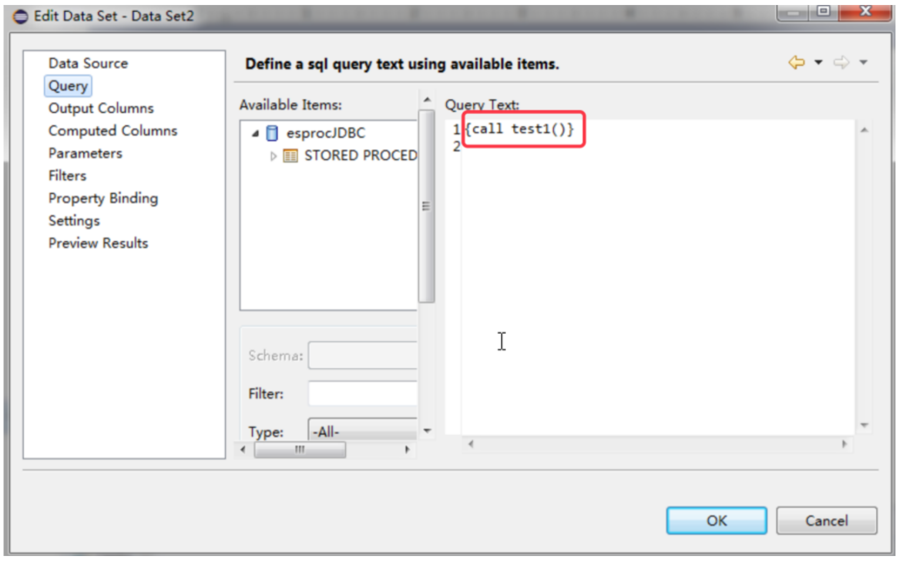


CAPITAL	STATE
Montgomery	Alabama
Juneau	Alaska
Phoenix	Arizona
Little Rock	Arkansas
Denver	Colorado
Hartford	Connecticut
Dover	Delaware
Tallahassee	Florida
Atlanta	Georgia
Honolulu	Hawaii
Boise	Idaho
Springfield	Illinois
Indianapolis	Indiana
Des Moines	Iowa
Topeka	Kansas
Frankfort	Kentucky
Baton Rouge	Louisiana
Augusta	Maine

At present, only one line of SPL codes can be called through BIRT query textbox. If you want to execute multiple lines of SPL codes, you need to save the SPL codes into a file under esProc IDE. The extension name of the file is dfx.

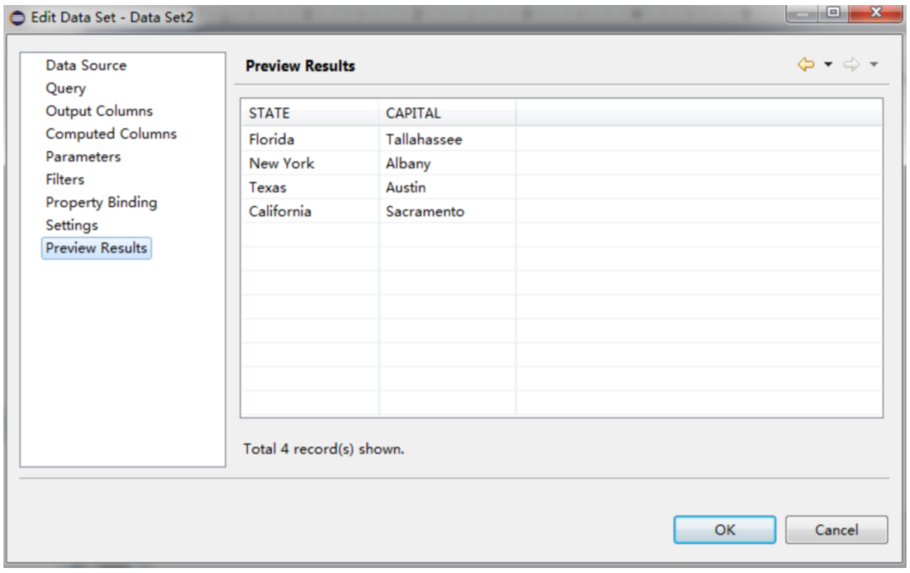
	A	B
1		=file("STATES.txt").import@t()
2	/=demo.query("select NAME as STATE,CAPITAL from STATES where POPULATION>15000000")	=B1.select(POPULATION>15000000).new(NAME:STATE ,CAPITAL)

As with the above mentioned example②, comment the codes that call data from the database, retain the codes that call data from text file, and then filter the records. Save the SPL codes into file test1.dfx. Copy the dfx file into the directory that is specified by <mainPath>C:\work\test_esproc</mainPath> from raqsoftConfig.xml. And then we only need to enter the SPL script file name **test1** into the query textbox under stored procedure.



```
{call test1()}
```

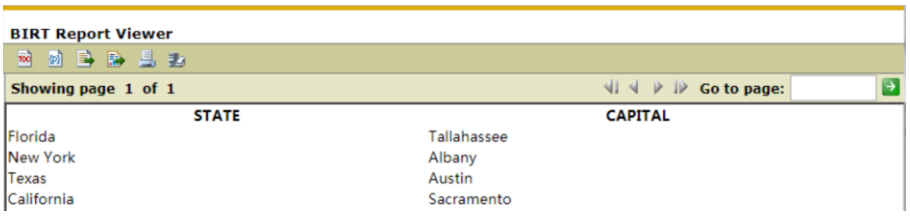
Click on the "Preview Results", and the filtered results are displayed:



STATE	CAPITAL
Florida	Tallahassee
New York	Albany
Texas	Austin
California	Sacramento

Total 4 record(s) shown.

Call the data set in a BIRT report, and the result is as follows:



STATE	CAPITAL
Florida	Tallahassee
New York	Albany
Texas	Austin
California	Sacramento

5 Application scenarioes

Multiple data sources

The source data of many reports doesn't only come from relational database. It may also come from NoSQL database, local files, WebService etc. The reporting tools lack the unified data acquisition interface and syntax for these non-relational database data sources, and some reporting tools can't perform the basic filtering operation. While some filtering or even join operations are normally needed in the creating process of a report. Reporting tools do have some computation ability, but they can only handle a certain amount of data because they are using in-memory calculating. When the data is huge, reporting tools will face the problem of capacity overloading. Moreover, most reporting tools can't handle multiple layer of data like json or XML, and they also lack the capability of flexible coding.

If esProc is used to prepare data for the reporting tools, the non-database data can be directly used, without the need to be imported into an intermediate database, and the workload is greatly reduced.

With its independent computational mechanism, esProc handles the various types of data sources in the same way, enabling a consistent computational ability in them. esProc's homogeneous approach of handling the heterogeneous data sources reduces code migration cost and the learning cost.

Here are two examples of common JSON computing problems.

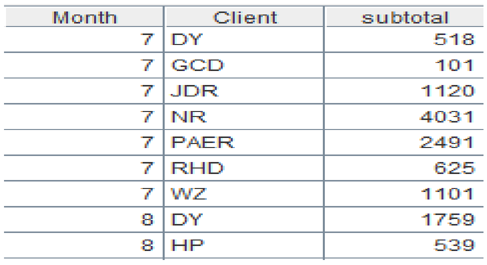
1、 JSON group and aggregation: *order.JSON* contains the order records. The requirement is to calculate the sales amount that each client contributes per month in a specified time period. Below is a selection of the source data:

```
[{OrderID:"26",Client:"TAS",SellerId:"1",Amount:2142,OrderDate:"05-08-2009 00:00:00"},
{OrderID:"33",Client:"DSGC",SellerId:"1",Amount:613,OrderDate:"14-08-2009 00:00:00"},
{OrderID:"84",Client:"GC",SellerId:"1",Amount:89,OrderDate:"16-10-2009 00:00:00"},
{OrderID:"133",Client:"HU",SellerId:"1",Amount:1420,OrderDate:"12-12-2010 00:00:00"},
{OrderID:"32",Client:"JFS",SellerId:"3",Amount:468,OrderDate:"13-08-2009 00:00:00"}…
```

The corresponding esProc codes:

	A
1	=file("D:\\order.JSON").read().import@j()
2	=A1.select(OrderDate>=argBegin && OrderDate<=argEnd)
3	=A2.groups(month(OrderDate):Month,Client;sum(Amount):subtotal)

Import the JSON file as a two-dimensional table and perform a conditional query and then the grouping and aggregation. *argBegin* and *argEnd* are parameters. Here's the result:



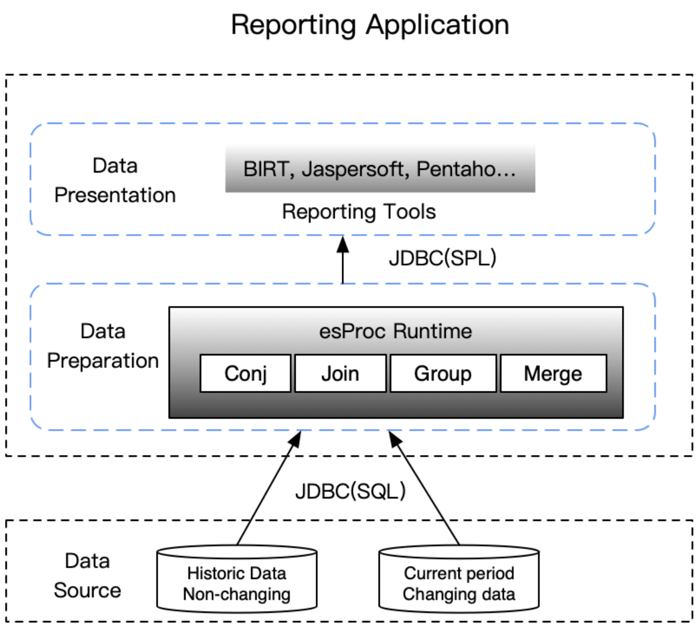
Month	Client	subtotal
7	DY	518
7	GCD	101
7	JDR	1120
7	NR	4031
7	PAER	2491
7	RHD	625
7	WZ	1101
8	DY	1759
8	HP	539

2、 testServlet returns JSON format strings of employee information. Now query the data according to the given condition and return result to Java main program in JSON format:

	A
1	=httpfile("http://localhost:6080/myweb/servlet/testServlet?table=employee&type=json")
2	=A1.read()
3	=A2.import@j()
4	=A3.select(${where})
5	=export@j(A4)

Read in the httpfile object and parse the JSON format strings to generate a two-dimensional table,and then filter the table according to the condition and export the result as JSON format strings. The parameter *where* is the dynamic querying condition, like BIRTHDAY>=date(1981,1,1) && GENDER=="F".

Cross-database hybrid operations



Relational databases' ability of maintaining transaction consistency hasn't yet been challenged. A transaction processing system is still established on the relational database.

Under this circumstance, to create a real-time T+0 report based on all available data, you need to keep huge historical data in the database for storing current transaction data. A database with huge capacity is thus required, which in turn increases the cost. Even if the cost is acceptable, the ever-increasing data amount will ultimately affect the transactional performance to an unacceptable level.

A general solution is to move a part of the historical data out to establish a sub-database, making sure that the transaction processing system operates without too much burden. But this will involve cross-database querying, complicating the development of a T+0 report.

Many databases support the cross-database operations, requiring that the databases involved should be of the same type. Different from handling current transaction data, historical data handling involves larger amount of data but requires no transaction consistency. This means users are likely to store the historical data in a data warehouse with a different structure.

However, even with databases of the same structure, the way of performing a cross-database operation is to map the data table in one database into the currently in use database. Actualy the operation takes place in one database, with more communication cost of data transmission, unsatisfactory performance and low steadiness.

eProc can handle this cross-database operation easily and effectively.

With its own computing engine, esProc is independent of the database. The database data, however, will still be handled in their place. esProc issues SQL statements to the involved databases through multiple parallel threads, databases will then perform the parallel processing and return results to the esProc for aggregation, and the aggregate results will be returned to the reporting tool for presentation. This system is easy to scale out, allowing multiple databases of historical data with different structures.

esProc also supplies the server-based cluster computing model. Rather than storing historical data in a database, esProc server allows placing the data in a file system with better I/O performance. The cooperation of the cluster computing and the storage plan brings a higher performance with lower cost.

Here is an example:

A telecom enterprise stores user service information with the table *userService*. T+0 report needs to show the duration of calls, the number of calls, the duration of calls to the local area and the number of calls to the local area of various telecom products. In actual use, it is found that the amount of data is too large and the query efficiency is very low, which results in poor report performance.

The performance can be greatly improved by using the computing layer instead. The specific method is to store the table *userService* in several databases in different areas, and then use the computing layer for parallel computing.

In esProc, the script is as follows:

	A	B
1	**[mysql1,mysql2,mysql3,mysql4]**	
2	**fork A1**	**=connect(A2)**
3		**=B2.query@x("select product_no,sum(allDuration) sallDuration,sum(allTimes) sallTimes,sum(localDuration) slocalDuration ,sum(localTimes) slocalTimes from userService where I0419=? group by product_no", argType)**
4	**=A2.conj()**	
5	**=A4.groups(product_no;sum(sallDuration):ad,sum(sallTimes):at,sum(slocalDuration):ld,sum(slocalTimes):lt)**	

Statement **fork** uses four threads in parallel. Each thread takes data from the corresponding database, and returns the grouped and aggregated results to the main thread. The main thread merges the calculation results of each sub-thread, performs the grouping and aggregation again, and finally obtains the result set required by the report.

General reporting tools do not have the ability to collect data in parallel and need to be completed by Java and other high-level languages. However, Java parallel code is difficult to write and lacks the ability of structured computing. Compared with independent computing layer, there is still a gap.

Out-database stored procedures

It's not easy to write a stored procedure. The traversal-style code only has an adequate performance but with very poor portability. The principle is that you should use as few stored procedures as possible.

esProc helps cut down the procedures in the database greatly. An algorithm will be stored and managed along with the report template in file system and become a part of the reporting module. This will reduce its coupling with the other parts of the application while won't add more coupling with other applications.

Yet stored procedures are necessary in some cases where they are used to pre-process the source data whose amount is too large to be moved out of the database.

Here are two examples:

1、 Calculate the excellent products whose sales amounts are within top 10 of each and every state.

Database table *stateSales* stores the sales information for products in each state, with three fields: state, product, amount. Some of the records are duplicated. Now we need to remove the duplicated records and calculate the excellent products whose sales amounts are within top 10 of each and every state, and finally represent the excellent products information in the report.

Previously, stored procedures were used to calculate excellent products. The codes are as follows:

```
create or replace package salesPkg
as
     type salesCur is ref cursor;
end;
CREATE OR REPLACE PROCEDURE topPro(io_cursor OUT salesPkg.salesCur)
is
   varSql varchar2(2000);
   tb_count integer;
BEGIN
  select count(*) into tb_count from dba_tables where table_name='TOPPROTMP';
  if tb_count=0 then
  strCreate:='CREATE GLOBAL TEMPORARY TABLE TOPPROTMP (
             stateTmp NUMBER not null,
             productTmp varchar2(10)  not null,
             amountTmp NUMBER not null
       )
       ON COMMIT PRESERVE ROWS';
  execute immediate strCreate;
  end if;
  execute immediate 'truncate table TOPPROTMP';
  insert into TOPPROTMP(stateTmp,productTmp,amountTmp)
select state,product,amount from stateSales a
 where not(
   (a.state,a.product) in (
     select state,product from stateSales group by state,product having count(*) > 1
   )
   and rowid not in (
        select min(rowid) from stateSales group by state,product having count(*)>1
       )
      )
    order by state,product;
    OPEN io_cursor for
    SELECT productTmp  FROM (
       SELECT stateTmp,productTmp,amountTmp,rankorder
       FROM (SELECT stateTmp,productTmp,amountTmp,RANK() OVER(PARTITION BY stateTmp ORDER BY
amountTmp DESC) rankorder
        FROM TOPPROTMP
       )
   WHERE rankorder<=10 order by stateTmp
   )
  GROUP BY productTmp
  HAVING COUNT(*)=(SELECT COUNT(DISTINCT stateTmp ) FROM TOPPROTMP);
END;
```

These coupling problems can be avoided by using esProc to implement the same algorithm. Following are the esProc scripts:

	A	
1	$select state， product， amount from stateSales	
2	=A1.group@1(state,product)	
3	=A2.group(state)	=A3.(~.rank(amount).pselect@a(~<=10))
4	=A3.(~(B3(#)).(product))	
5	=A4.isect()	

Remove the duplicates in A2, group data according to state in A3, get the sequence numbers of the top 10 products of each group in B3, get the product records by sequence numbers in A4, Calculate the intersection of groups in A5.

As mentioned in the *integration in the reporting tools* section, save the scripts mentioned above as topPro.dfx, and then it'll be called through JDBC in the reporting tool. The invocation method is the same as the ordinary stored procedure. The call statement is: call topPro ()

The above scripts are clear and simple, and can be interpreted and executed. The development efficiency is high and the maintenance is convenient. The scripts only need database *select* authorization, and no additional authorizations are required for modification and editing. The scripts and the report template are stored together in the file system, easy to be managed and maintained.

2、 Calculate "the top n customers with half of sales"

Some databases don't have window functions (eg. Mysql), and some databases don't have stored procedures (eg. Vertica). In case of complex data computing, external scripts in Rython or R are needed. However, these scripting languages and the mainstream engineering language (Java) are not well integrated. If we want to implement the functions like SQL functions or stored procedures directly in engineering languages, we usually need to write lengthy code for a particular computing requirement, and the code is almost non-reusable.

Even with powerful analysis functions, it is not easy to implement a slightly complicated logic. For example, the following common business calculation, find out "the top n customers whose sales account for half of the total sales, and sort them in descending order according to the sales volume". In Oracle, SQL is implemented as follows:

```
with A as
(selectCUSTOM,SALESAMOUNT,row_number() over (order by SALESAMOUNT) RANKING
from SALES)
select CUSTOM,SALESAMOUNT
from (select CUSTOM,SALESAMOUNT,sum(SALESAMOUNT) over (order by RANKING) AccumulativeAmount
from A)
where AccumulativeAmount>(select sum(SALESAMOUNT)/2 from SALES)
order by SALESAMOUNT desc
```

First rank the cumulative value of sales from small to large, through the condition that the cumulative value is greater than "half sales", the customers who account for half of sales are found in reverse. In order to avoid errors in processing the same value of sales by window function when calculating cumulative value, ranking is calculated by sub-query first.

Here's the code that implements the same logic with esProc:

	A	B
1	**=connnect("verticalCon")**	**/connect to database**
2	**=A1.query("select * from sales").sort("SALESAMOUNT:-1")**	**/Sort the salesamount in descending order**
3	**=A2.cumulate(SALESMOUNT)**	**/Calculate the cumulative salesamount in scequence**
4	**=A3.m(-1)/2**	**/Calculate “half sales”**
5	**=A3.pselect("~>=A4")**	**/Find the position in cumulative salesamount scequence fulfilling the condition**
6	**=A2(to(A5))**	**/Find the position of “half sales” and all records ahead of it**
7	**>A1.close()**	**/close database connection**
8	**return A6**	**/return result**

From the above code, we can see that esProc uses a simple set of syntax to implement the logic that needs nested SQL plus window functions to achieve, and has universal consistency (source code is consistent for any data).

Solution of SQL Difficulties

Grouped subsets

Except for data tables, SQL hasn't any other explicit set data type. So it can't help performing aggregation after data grouping. Besides aggregate values, you may also take an interest in grouped subsets. But it's difficult to perform data handling directly on these subsets, and subqueries are needed.

With set data type and grouping functions that return subsets, esProc can easily handle the post-grouping computations.

For example, you might want to find the records of all subjects for students whose total scores are equal to or above 500. With SQL, you need to group records to calculate the total score for every student, select the students whose scores are equal to or above 500, and then JOIN the resulting table with the original score table or use IN statement to find the desired records, which is roundabout and needs to retrieve data repeatedly. But in esProc, you can do this in a natural and straightforward way:

	A	
1	=db.query("select * from R")	
2	=A1.group(student).select(~.sum(score)>=500).conj()	

There are many scenarios that require the returning of the records of the subsets after data grouping. For those scenarios, the group and aggregate operations are intermediate steps towards completing a filtering, rather than the goal.

In some other cases, though only the aggregate values are desired, the aggregate operations are difficult to be expressed in simple aggregate functions and thus the grouped subsets need to be retained for the computations.

Those cases are not uncommon in real-world businesses. But as the computations often involve a lot of domain knowledge and are complicated, it's inconvenient to cite an example. Here's a simplified one.

The user login table *L* has two fields – user (ID) and login (time). Problem: Calculate the last login time of each user and the number of logins within 3 days before this time.

It's easy to find the last login time, but it's difficult to count the logins during the specified time period without retaining the grouped subsets. The SQL algorithm is like this: Group records to find the last login times; perform JOIN with the source table to find records during the specified time period and again group and aggregate these records. It's roundabout and inefficient. esProc, however, can retain the grouped subsets and thus can do this in a stepwise approach:

	A	
1	=db.query("select * from L")	
2	=A1.group(user;~.max(login):last,~.count(interval(login,last)<=3):num)	

In the code, ~ represents the subset obtained after data is grouped by user.
For ordered data, here's a more efficient way to get this done:

	A	
1	=db.query("select * from L order by login desc")	
2	=A1.group(user;~(1).login:last,~.pselect@n(interval(login,last)>3)-1:num)	

Order-related aggregation

It's also common to get the top N records and to find the record corresponding to the maximum value. Of course the computations can be handled on the retained grouped subsets. But as they are too common, esProc treats them as a kind of aggregation and provides a special function to handle in basically the same way as handling the ordinary grouping and aggregation.

SQL lacks Set data type and discreteness. It's unable to provide aggregate functions that can return a set of records. SQL's way is to use subqueries or other cumbersome tools, which often results in large-scale data sorting, which, in turn, causes performance loss.

Let's look at the simplest case. The user login table *L* has these fields – user, login (time), IP-address … Problem: Find the record of first login of each user.

SQL can use the window function to generate sequence numbers after intra-group sorting and retrieve all the records whose sequence numbers are 1. Window functions can only be employed on the result set, so a subquery and then a filter are needed. The code thus becomes complicated. For databases that don't support window functions, it's more difficult to do this.

esProc provides *group@1* function to directly retrieve the first member of each group.

	A	
1	=db.query("select * from L order by login")	
2	=A1.group@1(user)	

This type of log files is common and usually they are already ordered according to the time. esProc can get the first record directly without doing sorting. Cursor can be used to handle this if the amount of data is too big to be entirely loaded into the memory.

The stock price table *S* has three fields – code, date and cp (closing price). Problem: Calculate the latest rate of increase for each stock.

The calculation involves records of the last two trading days. SQL will use two levels of window functions to perform intra-group inter-row calculation and to retrieve the first row of the result. The coding is complicated. esProc provides the aggregate function *topN* to directly return the related records as aggregate values for further handling.

	A	
1	=db.query("select * from S")	
2	=A1.groups(code;top(2,-date))	/Get the records of the last two trading days
3	=A2.new(code,#2(1).cp-#2(2).cp:price-rises)	/Calculate the rate of increase

Instead of grouping records into subsets, esProc aggregate functions perform an accumulation, achieving a better performance. They can also work in a cursor-based mode to handle the big data that cannot be entirely loaded into the memory.

Records can be retrieved according to their sequence numbers if data is already ordered. This is more efficient:

	A	
1	=db.query("select * from S order by date desc")	
2	=A1.groups(code;top(2,0))	/Get the first two records directly
3	=A2.new(code,#2(1).cp-#2(2).cp:price-rises)	

To find the record corresponding to the maximum value, and to get the first/last record are special cases of the topN-style aggregation.

Order-related grouping

SQL only supports order-unrelated equi-grouping. Sometimes the grouping condition isn't a value which can be directly compared with every record. Instead, the grouping is more related to the order of the records. In this case, window functions (or other more inconvenient tools) should be used to generate sequence numbers first if SQL is used.

esProc offers ready-made functions for order-related grouping, making the computations involving continuous intervals more convenient.

The income & expense table *B* has three fields – month, income and expense. Problem: Find the records where a deficit exists for 3 or more continuous months.

	A	
1	=db.query("select * from B order by month")	
2	=A1.group@o(income>expense).select(~.income<~.expense && ~.len()>=3).conj()	

The *group@o* function compares only the adjacent records and creates a new group once the adjacent value changes. Each record can be profitable or unprofitable by comparing the income and expense fields. By comparing between adjacent records, records can be divided into groups like profitable, unprofitable, profitable…, get the unprofitable groups that have not less than 3 members and concatenate them.

You might also want to find in this table the maximum number of months when income increases continuously. The algorithm can be this: when the income increases, put the current record and the previous one into the same group; when the income decreases, put the record into a new group; finally get the maximum of the numbers of members among the continuous increasing groups.

	A	
1	=db.query("select * from B order by month")	
2	=A1.group@i(income<income[-1]).max(~.len())	

The *group@i* function will create a new group when the grouping condition changes, that is when the income decreases.

SQL can handle both scenarios with its window functions, but the code would be very hard to understand.

The merging of intervals is another common type of order-related grouping computation. The event interval table *T* has fields S (start time) and E (end time). Problem: Find the real length of time the whole event takes by removing the overlapping time intervals from these intervals.

	A	
1	**$select S,E from T order by S**	
2	**=A1.select(E>max(E{:-1}))**	**/Remove records where the time period is completely covered**
3	**=A2.run(max(S,E[-1]):S)**	**/Remove the time intervals which are completed covered**
4	**=A2.sum(interval@s(max(S,E[-1]),E))**	**/Calculate the total length of time**
5	**=A2.run(if(S<E[-1],S[-1],S):S).group@o(S;~.m(-1).E:E)**	**/Merge the overlapping time intervals by start time**

Here are solutions for different types of order-related grouping, which make the most use of the features of the inter-row calculation and the order-related grouping. SQL cannot handle those scenarios by simply using window functions. It needs to use the extremely difficult-to-understand recursive query.

Position-based operations

Sometimes you might want to use the sequence numbers to directly access members of an ordered set. SQL, which is based on the mathematical concept of unordered sets, will first generate the sequence numbers and perform conditional filtering before accessing members of the specified positions. This is very troublesome to many computations.

esProc, however, adopts a mechanism based on the concept of ordered sets, allowing accessing members directly with sequence numbers and bringing great conveniences.

For example, the median value of various prices is often desired in analyzing economic statistics:

	A	
1	**=db.query@i("select price from T order by price")**	
2	**=A1([(A1.len()+1)\2,A1.len()\2+1]).avg()**	

Sequence numbers can be used in data grouping as well. The event table *E* has three fields – no, time and act. The act field includes two types of values - start and end. Problem: Calculate the total length of time the events take, that is, the sum of every time period that each pair of start and end defines.

	A	
1	**=db.query@i("select time from E order by time")**	
2	**=A1.group((#-1)\2).sum(interval@s(~(1),~(2))**	

# represents the sequence number of a record. *group((#-1)\2)* places every two records in one group; and then calculate the length of total time for each group and perform aggregation.

You can make an inter-row reference between two neighboring records according to sequence numbers. The stock price table *S* has two fields – date and cp (closing price). Problem: List the trading days when the stock prices are above 100 and calculate the rates of price increase on those days.

	A	
1	=db.query("select * from S order by date")	
2	=A1.pselect@a(cp>100).select(~>1)	
3	=A2.new(A1(~).date:date,A1(~).cp-A1(~-1).cp:price-rises)	

The *pselect* function returns the sequence numbers of the members satisfying the specified condition. According to the result, you can calculate the rate of increase easily, without having to calculate the rates of increase for all days and then perform filtering as with the window functions.

Sometimes the grouping result is expected to be continuous ranges. So the missing empty subsets need to be supplied. It's troublesome to do this in SQL, because you should first create a set of continuous ranges manually and then *left join* the data tables under processing, during which you have to use the subquery. Yet esProc can use sequence numbers to realize the alignment grouping.

Here's a simple transaction record table *T*, including no, date and amount fields. Problem: Cumulate the transaction amounts week by week; the weeks without transaction records also need to be displayed.

	A	
1	=db.query("select * from T order by date")	
2	>start=A1(1).date	
3	=interval(start,A1.m(-1).date)\7+1	/Calculate the total number of weeks
4	=A1.align@a(A2,interval(start,date)\7)	/Group records by week; empty sets probably exist
5	=A4.new(#:week,acc[-1]+~.sum(amount):acc)	/Aggregate the weekly amount and calculate the cumulative amount

Dynamic column handling

A thorough set orientation should also include column-based set operations.

As an interpreted language, SQL can improvise a dynamic data structure. But SQL regards columns as one of the data attributes, which are static, so it doesn't provide any column-based set operations. Consequently, this becomes a headache in dealing with scenarios where the desired column data isn't supplied or where a standard approach is needed to handle many columns.

Inter-column aggregation

PE is a table recording results of physical education. It has the following fields – name, 100m, 1000m, long-jump, high-jump, and … There are four grades - A, B, C and D – for evaluating the results. Problem: Calculate the numbers of people in each grade for every event.

The algorithm is simple. You just need to union the results of all the events, group them and perform aggregate. SQL will use a long union statement to combine the results of all events. That's really tedious. It's complicated if the columns are indefinite because you need to obtain the desired column names dynamically from the database to perform the union.

esProc supports column-based set operations. The fully dynamic syntax makes coding simple and easy:

	A	
1	=db.query("select * from PE")	
2	=A1.conj(~.array().to(2,))	/Concatenate the results for every event from the second field onward
3	=A2.groups(~:grade;count(1):count)	/Grouping and aggregation

Standard approach for transpositions
For simple static transpositions, some databases supply *pivot* and *unpivot* statements to implement them. Databases that don't support the statements can do this using complicated conditional expressions and union statement. But usually the columns transposed from rows are dynamic. To handle this in SQL, you need to first generate the target columns and rows and then compose another statement dynamically for execution. SQL doesn't support step-by-step calculation, and the code is complicated and difficult to understand.

The student scores table *R* consists of these fields – student, semester, math, English, science, pe, art and …
Problem: Perform both the row-to-column transposition and column-to-row transposition to present data in a structure as this – student, subject, semester1, semester2 …

esProc offers *pivot* function to perform the simple transposition:

	A	B	C
1	=db.query("select * from R")		
2	=A1.pivot@r(student,semester;subject,score)		
3	=A2.pivot(student,subject;semester,score)		

To achieve the two-way transposition, A2 performs column-to-row transposition and A3 performs row-to-column transposition.

There is also a standard method which is easier-to-understand yet slightly complicated:

	A	B	C
1	=db.query("select * from R order by student,semester")		
2	=create(student,subject,${A1.id(semester).string()})		
3	for A1.group(student)	for 3,A1.fno()	=A3.field(B3)
4			>A2.record(A1.student|A2.fname(B3)|C3)
5	return A2		

A2 generates the target result set using a macro. The loop in A3 and A4 transposes rows and columns and insert the result in the result set, which is the standard procedure for performing transpositions in esProc. The stepwise approach makes code clear and easy-to-understand. The approach applies to static transposition or one-way transposition and the code would be even simpler. esProc's column access scheme and its flexibility characteristic of a dynamic language enables programmers to handle all types of transpositions, including static/dynamic transpositions, row-to-column transposition, column-to-row transposition, two-way transposition, in one standard approach.

Extended transpositions
Here's the account state table *T*:

seq	acount	state	date
1	A	over	2014-1-4
2	A	OK	2014-1-8
3	A	lost	2014-3-21
…			

Problem: Export the status of the accounts per day for a specified month; if there's no record for an account on a certain date, then use the status of the previous date:

account	1	2	3	4	5	6	7	8	9	…	31
A				over	over	over	over	OK	OK	…	OK
...											

Strictly speaking, the transposition is static. But it involves a lot of regular columns and is not easy to be expressed completely in a static way. It involves inter-column calculations as well, which are hard to be coded in SQL even using *pivot* statements.

The job can easily get done using the standard esProc way:

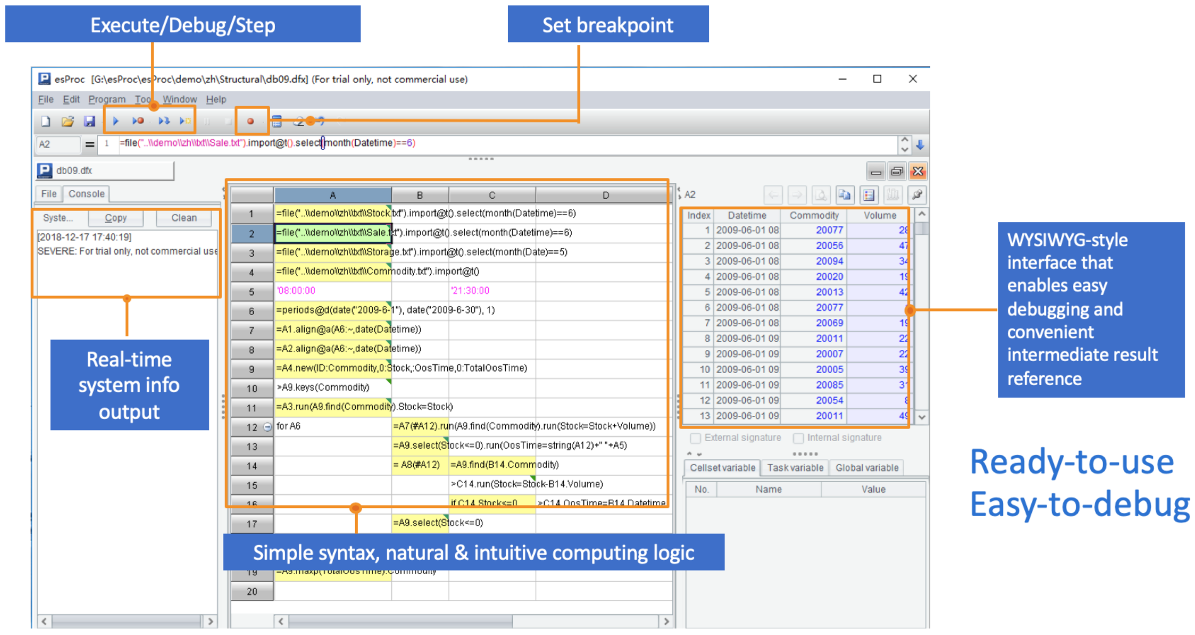
	A	B
1	=db.query("select * from T where year(date)=? and month(date)=?",2014,1)	
2	=create(account,${to(31).string()})	
3	for A1.group(account)	=31.(null)
4		>A3.run(B3(day(date))=state)
5		>B3.run(~=ifn(~,~[-1])
6		>A2.record(A3.account|B3)
7	return A2	

Here's only one loop because it is the one-way transposition. In B3-B5, the calculation of getting data to be inserted to the result set according to esProc syntax is a little complicated. Yet the whole procedure is the same.

6 Ease of use

esProc SPL is a DSL (Domain Specific Language) specifically designed to process structured data. It is unlike common languages, whose target range covers everything. It's easy to master through short practice. It has the properties of step-by-step processing and intermediate result reference. It is easier to implement than SQL in the face of complex requirements. The data structure is simple and the syntax is concise. It is easier to learn and use than Python.

Simple interface, easy debugging


Execute/Debug/Step
Set breakpoint
Real-time system info output
Simple syntax, natural & intuitive computing logic
WYSIWYG-style interface that enables easy debugging and convenient intermediate result reference
Ready-to-use
Easy-to-debug
esProc [G:\esProc\esProc\demo\zh\Structural\db09.dfx] (For trial only, not commercial use)
File Edit Program Tool Window Help
db09.dfx
File Console
Syste... Copy Clean
External signature
Internal signature
Cellset variable Task variable Global variable
No. Name Value

A2 = `=file("..\\demo\\zh\\txt\\Sale.txt").import@t().select(month(Datetime)==6)`

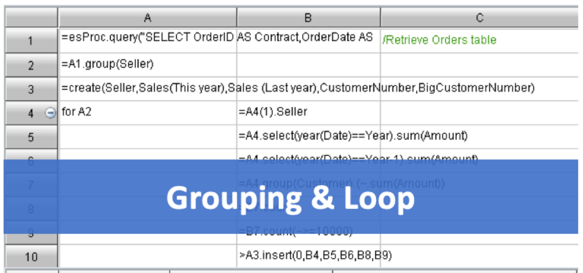
```
[2018-12-17 17:40:19]
SEVERE: For trial only, not commercial use
```

	A	B	C	D
1	=file("..\\demo\\zh\\txt\\Stock.txt").import@t().select(month(Datetime)==6)			
2	=file("..\\demo\\zh\\txt\\Sale.txt").import@t().select(month(Datetime)==6)			
3	=file("..\\demo\\zh\\txt\\Storage.txt").import@t().select(month(Date)==5)			
4	=file("..\\demo\\zh\\txt\\Commodity.txt").import@t()			
5	'08:00:00		'21:30:00	
6	=periods@d(date("2009-6-1"), date("2009-6-30"), 1)			
7	=A1.align@a(A6:~,date(Datetime))			
8	=A2.align@a(A6:~,date(Datetime))			
9	=A4.new(ID:Commodity,0:Stock,:OosTime,0:TotalOosTime)			
10	>A9.keys(Commodity)			
11	=A3.run(A9.find(Commodity).Stock=Stock)			
12	for A6	=A7(#A12).run(A9.find(Commodity).run(Stock=Stock+Volume))		
13		=A9.select(Stock<=0).run(OosTime=string(A12)+" "+A5)		
14		= A8(#A12)	=A9.find(B14.Commodity)	
15			>C14.run(Stock=Stock-B14.Volume)	
16			if C14.Stock<=0	>C14.OosTime=B14.Datetime
17		=A9.select(Stock<=0)		
19	=A9.maxp(TotalOosTime).Commodity			
20				

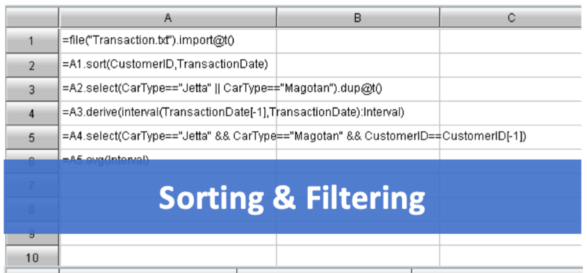
A2

Index	Datetime	Commodity	Volume
1	2009-06-01 08	20077	28
2	2009-06-01 08	20056	47
3	2009-06-01 08	20094	34
4	2009-06-01 08	20020	19
5	2009-06-01 08	20013	42
6	2009-06-01 08	20077	[illegible]
7	2009-06-01 08	20069	19
8	2009-06-01 09	20011	22
9	2009-06-01 09	20007	22
10	2009-06-01 09	20005	39
11	2009-06-01 09	20085	31
12	2009-06-01 09	20054	8
13	2009-06-01 09	20011	49

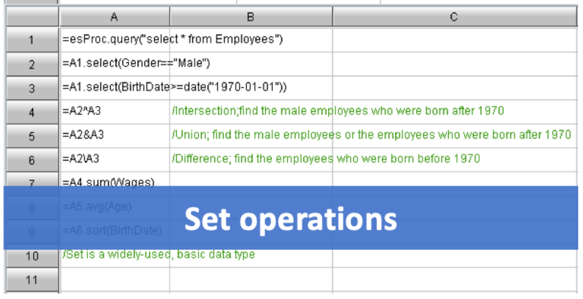
Syntax is simple, easy to learn and use.



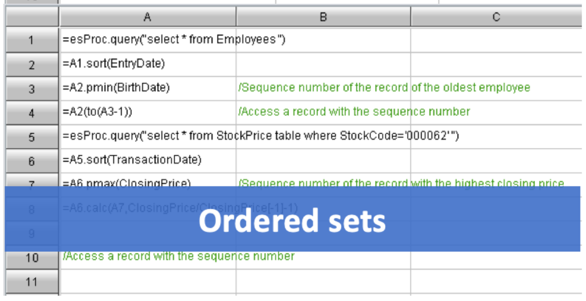
	A	B	C
1	=esProc.query("SELECT OrderID AS Contract,OrderDate AS		/Retrieve Orders table
2	=A1.group(Seller)		
3	=create(Seller,Sales(This year),Sales (Last year),CustomerNumber,BigCustomerNumber)		
4	for A2	=A4(1).Seller	
5		=A4.select(year(Date)==Year).sum(Amount)	
6		=A4.select(year(Date)==Year-1).sum(Amount)	
7		=A4.group(Customer).(~.sum(Amount))	
8		[illegible]	
9		=B7.count(~>=10000)	
10		>A3.insert(0,B4,B5,B6,B8,B9)	



	A	B	C
1	=file("Transaction.txt").import@t()		
2	=A1.sort(CustomerID,TransactionDate)		
3	=A2.select(CarType=="Jetta" || CarType=="Magotan").dup@t()		
4	=A3.derive(interval(TransactionDate[-1],TransactionDate):Interval)		
5	=A4.select(CarType=="Jetta" && CarType=="Magotan" && CustomerID==CustomerID[-1])		
6	=A5.avg(Interval)		
7			
8			
9			
10			



	A	B	C
1	=esProc.query("select * from Employees")		
2	=A1.select(Gender=="Male")		
3	=A1.select(BirthDate>=date("1970-01-01"))		
4	=A2^A3	/Intersection;find the male employees who were born after 1970	
5	=A2&A3	/Union; find the male employees or the employees who were born after 1970	
6	=A2\A3	/Difference; find the employees who were born before 1970	
7	=A4.sum(Wages)		
8	=A5.avg(Age)		
9	=A6.sort(BirthDate)		
10	/Set is a widely-used, basic data type		
11			

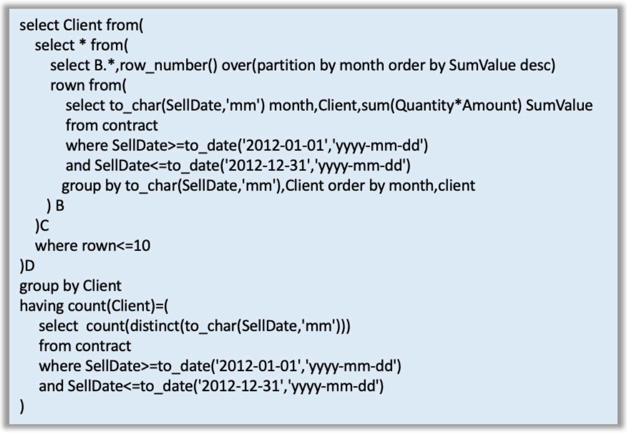


	A	B	C
1	=esProc.query("select * from Employees")		
2	=A1.sort(EntryDate)		
3	=A2.pmin(BirthDate)	/Sequence number of the record of the oldest employee	
4	=A2(to(A3-1))	/Access a record with the sequence number	
5	=esProc.query("select * from StockPrice table where StockCode='000062'")		
6	=A5.sort(TransactionDate)		
7	=A6.pmax(ClosingPrice)	/Sequence number of the record with the highest closing price	
8	=A6.calc(A7,ClosingPrice/ClosingPrice[-1]-1)		
9			
10	/Access a record with the sequence number		
11			

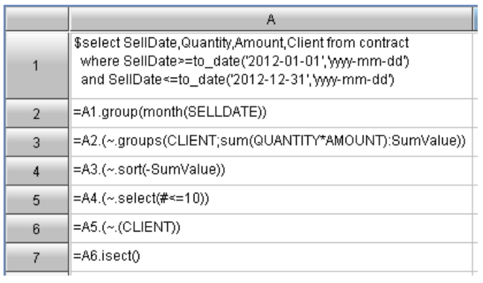
Comparison with SQL

Example 1：

To find clients whose sales amount are listed in the top 10 every month



```
select Client from(
   select * from(
      select B.*,row_number() over(partition by month order by SumValue desc)
      rown from(
         select to_char(SellDate,'mm') month,Client,sum(Quantity*Amount) SumValue
         from contract
         where SellDate>=to_date('2012-01-01','yyyy-mm-dd')
         and SellDate<=to_date('2012-12-31','yyyy-mm-dd')
        group by to_char(SellDate,'mm'),Client order by month,client
      ) B
   )C
   where rown<=10
)D
group by Client
having count(Client)=(
   select  count(distinct(to_char(SellDate,'mm')))
   from contract
   where SellDate>=to_date('2012-01-01','yyyy-mm-dd')
   and SellDate<=to_date('2012-12-31','yyyy-mm-dd')
)
```



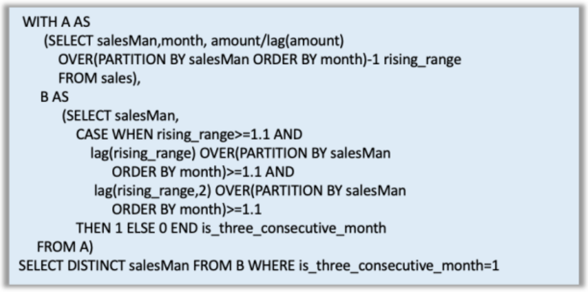
	A
1	$select SellDate,Quantity,Amount,Client from contract where SellDate>=to_date('2012-01-01','yyyy-mm-dd') and SellDate<=to_date('2012-12-31','yyyy-mm-dd')
2	=A1.group(month(SELLDATE))
3	=A2.(~.groups(CLIENT;sum(QUANTITY*AMOUNT):SumValue))
4	=A3.(~.sort(-SumValue))
5	=A4.(~.select(#<=10))
6	=A5.(~.(CLIENT))
7	=A6.isect()

SQL :
The script is difficult to read and maintain, and can not be simplified via step by step process

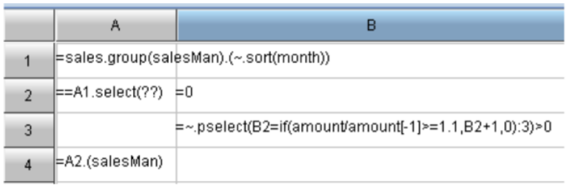
SPL :
The script is done in the cells and natural step by step computing is realized by mutual cell-name reference

Example 2：

To find out the salesmen whose sales volume have increased by 10% in three consecutive months



```
WITH A AS
   (SELECT salesMan,month, amount/lag(amount)
      OVER(PARTITION BY salesMan ORDER BY month)-1 rising_range
      FROM sales),
  B AS
      (SELECT salesMan,
         CASE WHEN rising_range>=1.1 AND
            lag(rising_range) OVER(PARTITION BY salesMan
                  ORDER BY month)>=1.1 AND
             lag(rising_range,2) OVER(PARTITION BY salesMan
                  ORDER BY month)>=1.1
         THEN 1 ELSE 0 END is_three_consecutive_month
   FROM A)
SELECT DISTINCT salesMan FROM B WHERE is_three_consecutive_month=1
```



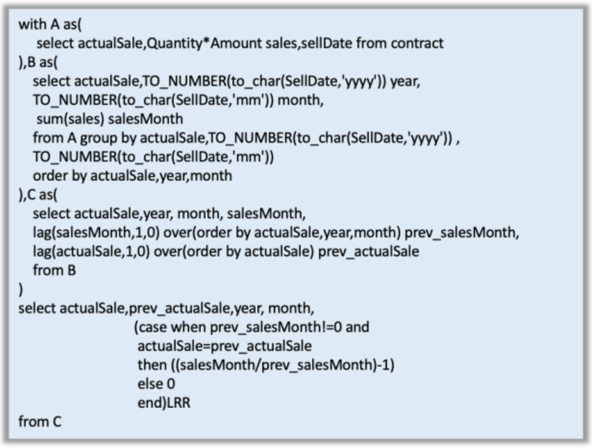
	A	B
1	=sales.group(salesMan).(~.sort(month))	
2	==A1.select(??)	=0
3		=~.pselect(B2=if(amount/amount[-1]>=1.1,B2+1,0):3)>0
4	=A2.(salesMan)	

SQL :
Some computing has to be done by sub query due to lack of set computing.

SPL :
Complex computing can be greatly simplified with explicit set, relative position, sequence reference and computing after grouping, etc.

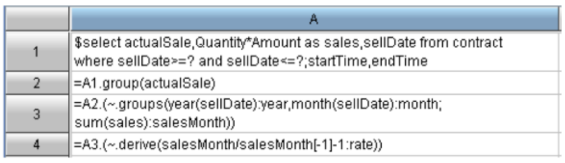
Example 3：

To get the monthly contract growth rate of each salesman



```sql
with A as(
    select actualSale,Quantity*Amount sales,sellDate from contract
),B as(
   select actualSale,TO_NUMBER(to_char(SellDate,'yyyy')) year,
   TO_NUMBER(to_char(SellDate,'mm')) month,
    sum(sales) salesMonth
   from A group by actualSale,TO_NUMBER(to_char(SellDate,'yyyy')) ,
   TO_NUMBER(to_char(SellDate,'mm'))
   order by actualSale,year,month
),C as(
   select actualSale,year, month, salesMonth,
   lag(salesMonth,1,0) over(order by actualSale,year,month) prev_salesMonth,
   lag(actualSale,1,0) over(order by actualSale) prev_actualSale
   from B
)
select actualSale,prev_actualSale,year, month,
                        (case when prev_salesMonth!=0 and
                         actualSale=prev_actualSale
                         then ((salesMonth/prev_salesMonth)-1)
                         else 0
                         end)LRR
from C
```

SQL :
Disordered data. Ordered computing can only be achieved by indirect script.



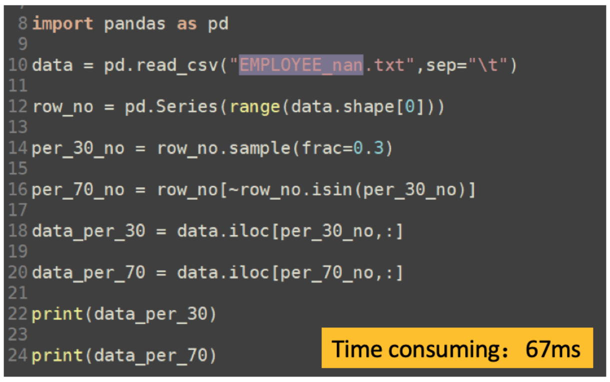
	A
1	$select actualSale,Quantity*Amount as sales,sellDate from contract where sellDate>=? and sellDate<=?;startTime,endTime
2	=A1.group(actualSale)
3	=A2.(~.groups(year(sellDate):year,month(sellDate):month; sum(sales):salesMonth))
4	=A3.(~.derive(salesMonth/salesMonth[-1]-1:rate))

SPL :
Totally supports ordered computing, and can easily solve order related problems, like sorting, ranking, top N, comparing with the previous or the same period, etc.

Comparison with Python

Example 1：

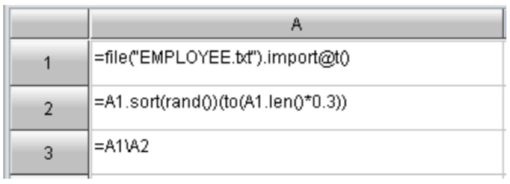
Sampling(Divide the data into 30% and 70% randomly)



```python
import pandas as pd

data = pd.read_csv("EMPLOYEE_nan.txt",sep="\t")

row_no = pd.Series(range(data.shape[0]))

per_30_no = row_no.sample(frac=0.3)

per_70_no = row_no[~row_no.isin(per_30_no)]

data_per_30 = data.iloc[per_30_no,:]

data_per_70 = data.iloc[per_70_no,:]

print(data_per_30)

print(data_per_70)
```

Time consuming：67ms

Python :
Need to show introducing class libraries, Set operations (difference sets) are slightly more complex. To view the results, you need to print them on display.



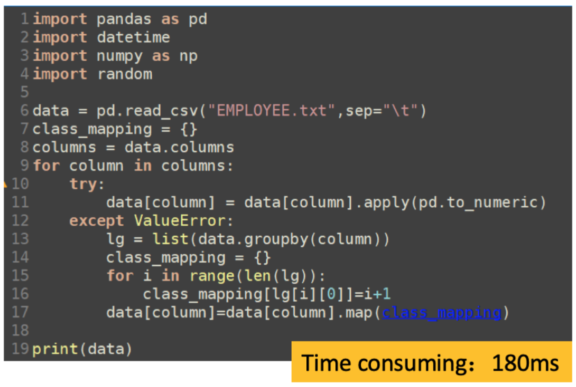
	A
1	=file("EMPLOYEE.txt").import@t()
2	=A1.sort(rand())(to(A1.len()*0.3))
3	=A1\A2

Time consuming：6ms

SPL :
Provides rich built-in libraries for direct use, Set operations are very simple to write. The results of each step can be viewed in the results panel.

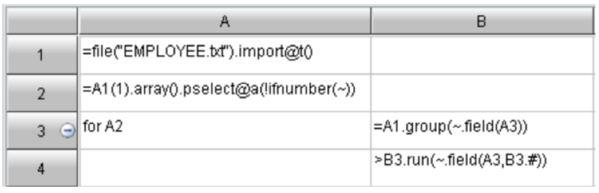
Example 2：

Data conversion(Number fields remain unchanged, and other fields are converted to numbers)



```python
import pandas as pd
import datetime
import numpy as np
import random

data = pd.read_csv("EMPLOYEE.txt",sep="\t")
class_mapping = {}
columns = data.columns
for column in columns:
    try:
        data[column] = data[column].apply(pd.to_numeric)
    except ValueError:
        lg = list(data.groupby(column))
        class_mapping = {}
        for i in range(len(lg)):
            class_mapping[lg[i][0]]=i+1
        data[column]=data[column].map(class_mapping)

print(data)
```

Time consuming： 180ms



	A	B
1	=file("EMPLOYEE.txt").import@t()	
2	=A1(1).array().pselect@a(!ifnumber(~))	
3	for A2	=A1.group(~.field(A3))
4		>B3.run(~.field(A3,B3.#))

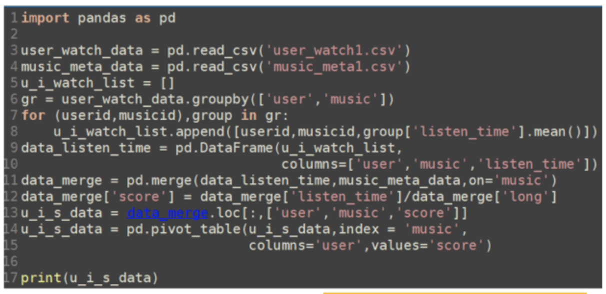
Time consuming： 15ms

Python：
The implementation is relatively simple, but the codes are not short. Identifying blocks of code by indentation is a challenge for users.

SPL：
The overall codes are very short, and the loop functions provided can easily traverse the data. Code blocks are clearly identified through the grid.

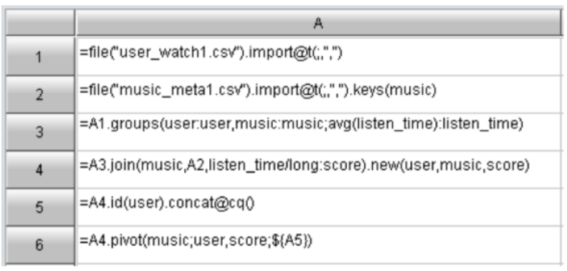
Example 3：

Generate PivotTable (music as index, user as field, score as score)



```python
import pandas as pd

user_watch_data = pd.read_csv('user_watch1.csv')
music_meta_data = pd.read_csv('music_meta1.csv')
u_i_watch_list = []
gr = user_watch_data.groupby(['user','music'])
for (userid,musicid),group in gr:
    u_i_watch_list.append([userid,musicid,group['listen_time'].mean()])
data_listen_time = pd.DataFrame(u_i_watch_list,
                                columns=['user','music','listen_time'])
data_merge = pd.merge(data_listen_time,music_meta_data,on='music')
data_merge['score'] = data_merge['listen_time']/data_merge['long']
u_i_s_data = data_merge.loc[:,['user','music','score']]
u_i_s_data = pd.pivot_table(u_i_s_data,index = 'music',
                            columns='user',values='score')

print(u_i_s_data)
```

Time consuming： 23ms



	A
1	=file("user_watch1.csv").import@t(;,",")
2	=file("music_meta1.csv").import@t(;,",").keys(music)
3	=A1.groups(user:user,music:music;avg(listen_time):listen_time)
4	=A3.join(music,A2,listen_time/long:score).new(user,music,score)
5	=A4.id(user).concat@cq()
6	=A4.pivot(music;user,score;${A5})

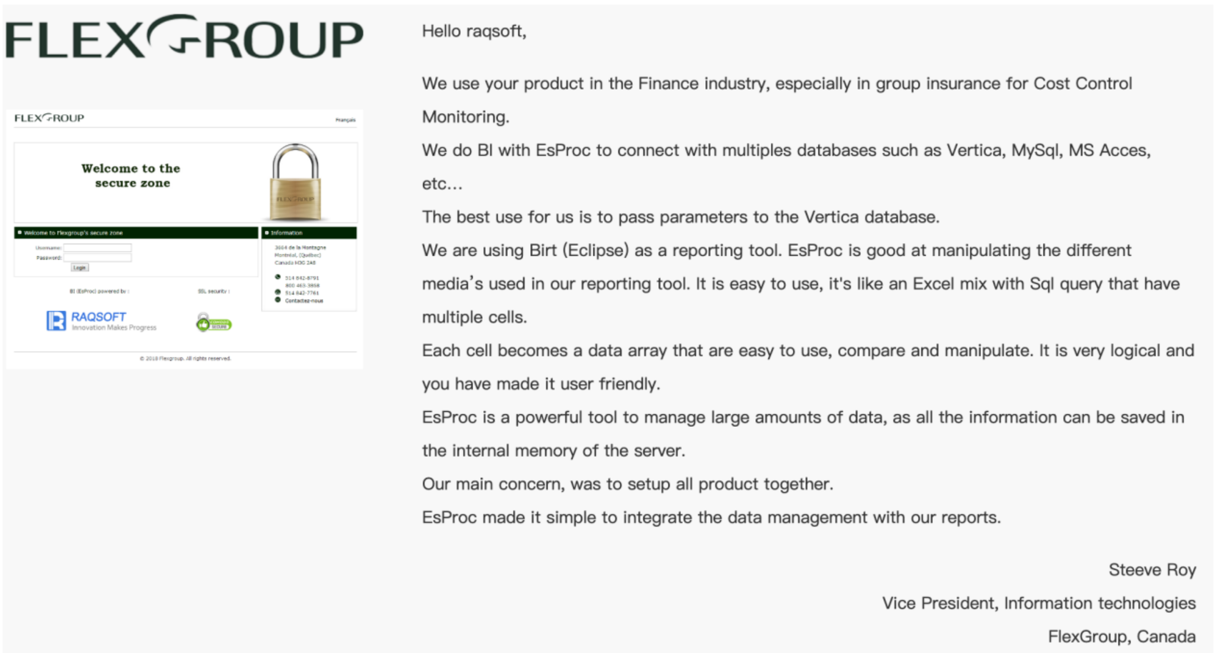
Time consuming： 1ms

Python：
Explicit “for” is needed to implement grouping and transpose actions .

SPL：
The process code is more concise, without any “for” .

7 Contact us

esProc is a set of middleware which runs on JVM and deals with structured data. It can be easily integrated with Java applications through JDBC interface, and provides portable and powerful computing logic for applications. After seven years of research and development and three years of commercial use, it has been widely used in China and recognized by some international users.



Hello raqsoft,

We use your product in the Finance industry, especially in group insurance for Cost Control Monitoring.

We do BI with EsProc to connect with multiples databases such as Vertica, MySql, MS Acces, etc…

The best use for us is to pass parameters to the Vertica database.

We are using Birt (Eclipse) as a reporting tool. EsProc is good at manipulating the different media’s used in our reporting tool. It is easy to use, it's like an Excel mix with Sql query that have multiple cells.

Each cell becomes a data array that are easy to use, compare and manipulate. It is very logical and you have made it user friendly.

EsProc is a powerful tool to manage large amounts of data, as all the information can be saved in the internal memory of the server.

Our main concern, was to setup all product together.

EsProc made it simple to integrate the data management with our reports.

Steeve Roy

Vice President, Information technologies

FlexGroup, Canada

In 2019, we began to expand the international market and develop partners. If you are a developer or distributor in the data industry and are interested in our products, you can contact us via the official website (www.raqsoft.com). Except the powerful products, we have the best experts in the industry. They found that there is no very handy tool in the process of data development, so they built esProc. They have rich experience and a thorough understanding of various data computing problems. If you are an engineer and have any problem with report calculation, welcome to seek advice from our product community (c.raqsoft.com).