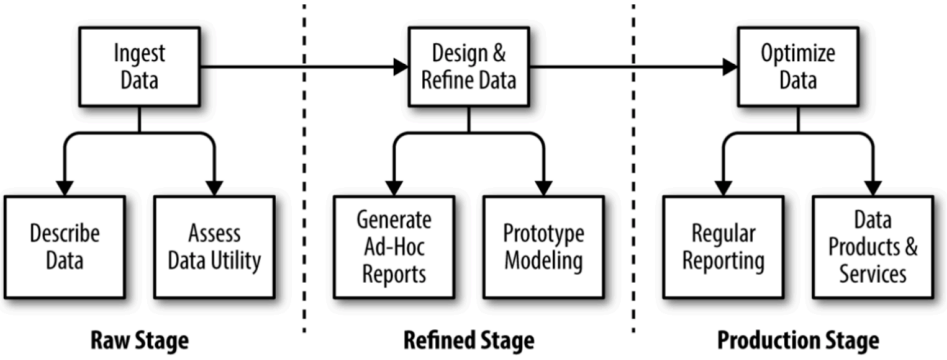
从数据整理到业务计算的最佳工具

1 概要

数据整理是为业务人员进一步分析数据或进入 BI 系统前最重要的环节。随着数据时代的到来，数据来源越来越多样(文件、大数据平台、数据库…)，为数据整理带来了很多挑战。在企业里数据通常是由 IT 人员负责，统一组织数据抽取、数据变换和加载数仓的流程，即 ETL，之后再提供给业务人员或可视化系统，数据从 Raw Stage 到 Refined Stage 再到 Production Stage 三个阶段才能最终拿来使用，整个过程纳入自动化管理。然而，实际上这种集中式的自动化处理流程，每增加一个数据需求都要依赖 IT 人员和 ETL 专属软硬件，冗长的数据开发周期、公用的 ETL 软硬件随着需求的增多越来越耗时，都让业务人员不能及时感知数据的变化（比如，销售部门想获得进行了某项促销活动前后的销售情况变化信息）。



因此，脱离专属人员和设备，另辟蹊径开展敏捷数据整理，为数据驱动的业务部门提供有力支撑变得越来越重要。即，数据从 Raw Stage 到 Refined Stage 后直接被业务人员用于桌面分析或导入 BI 系统后的自助分析。

数据导入 BI 系统后，BI 系统的自助部分只是在多维分析和关联查询这两个层面满足业务需求，从经验上看，最好情况也就能解决 30% 左右的问题而已，剩下 70% 左右或更多的需求，比如找出“销售额占到一半的前 n 个客户，并按销售额从大到小排序”，都会涉及到多步骤有过程的计算。而过程计算完全超出 BI 产品的设计目标，甚至可以不被认为是数据分析，但却是用户特别希望解决的问题。碰到这类问题，通常还是导出数据由业务人员自己用 Excel 等做桌面分析，但是，Excel 并不擅长处理多层次数据的关联运算，而且数据量大了也撑不住，在许多应用场景无法胜任。这类问题还是需要技术人员才能解决，SQL 难以处理有过程计算，Java 做结构化运算代码冗长、不易复用和维护，python(pandas)/R 的定位是数学风格的统计分析，虽然提供了 dataframe 对象用于处理结构化数据，但是还不够像 SQL 那么简单直观，易学易用。

那么，有没有用一种工具，既能敏捷的整理数据，又能轻松应对复杂的业务计算呢？润乾集算器的目标，就是为普通技术人员提供从数据整理到业务计算最便捷的途径，具备以下特性满足这类技术需求。

1. 连接性
   能够连接各种数据源
   文件（CSV,JSON,Excel…）

大数据平台（Hive,HDFS,MongoDB…）
云平台（AWS Redshift,AWS S3,Azure ADLS）
数据库（Oracle,DB2,Mysql,PostgreSQL,TD…）
应用（Salesforce,Tableau Server…）

2. 易用性
   - 即装即用

   无需安装额外的依赖工具包
   - 分步处理

   避免 SQL 式嵌套和单向式管道处理，每步结果可以随时引用，复杂问题化整为零
   - 易理解、易复用

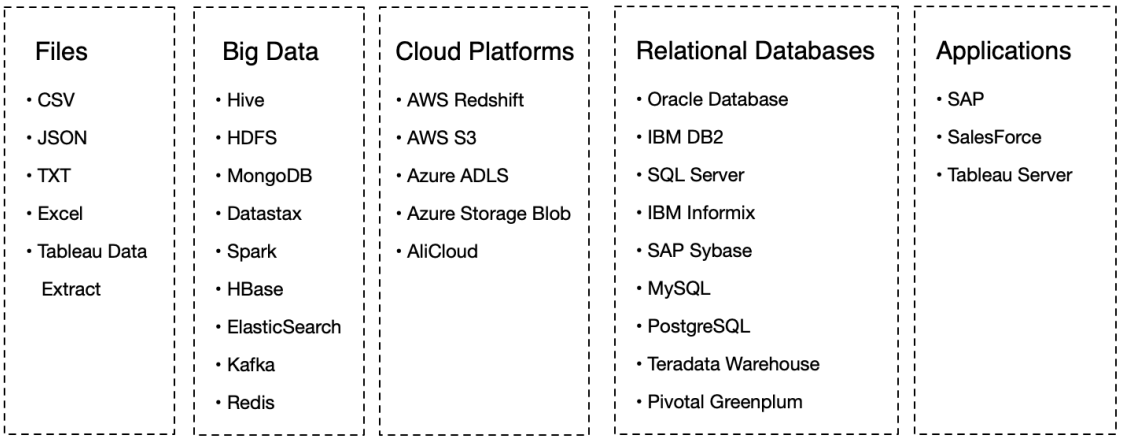
   易理解的必然是容易学习和掌握的；
   用最少和直观的脚本来解决问题，整理过程一目了然，雷同问题很少改动即可复用
   - 核心操作完备

   数据整理的核心操作是 Structuring、Enriching、Cleaning
   - 调试方便

   不能只靠输出来调试程序，支持单步、断点等高级调试模式
   - 模块化开发

   任务按模块拆分，并能集中整合形成处理流程
3. 大数据
   单机处理能力，支持 GB 规模、方便的数据分段、多线程/多进程并行计算、外存计算
   多机处理能力，支持 TB 规模、灵活的数据分布、分布式计算
4. 集成性
   报表集成，已完成的数据整理脚本，可与报表工具集成，作为报表数据源，运行结果直接为报表提供数据。
   ETL 集成，已完成的数据整理脚本，可与 ETL 工具集成，被 ETL 调度后自动运行，将临时的数据整理，纳入日常批处理。

2 连接性



集算器能从各种数据源中获取数据，根据用户需要仍在不断添加中。

3 易用性

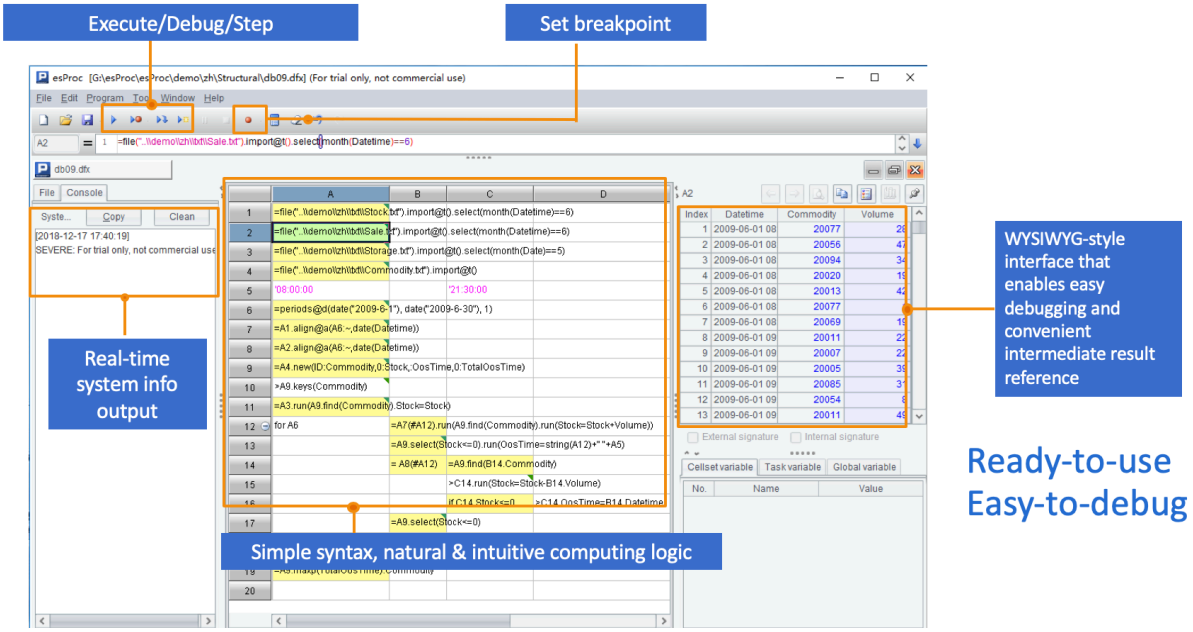
简洁编程环境

网格编程

集算器使用网格式编程，和在 Excle 表格里写表达式类似。在单元格里书写集算器脚本 SPL(Structured Process Language)，单元格内 SPL 执行的结果赋值给单元格地址，单元格地址作为变量名后续直接引用。代码按单元格顺序，先从左到右、后由上到下。

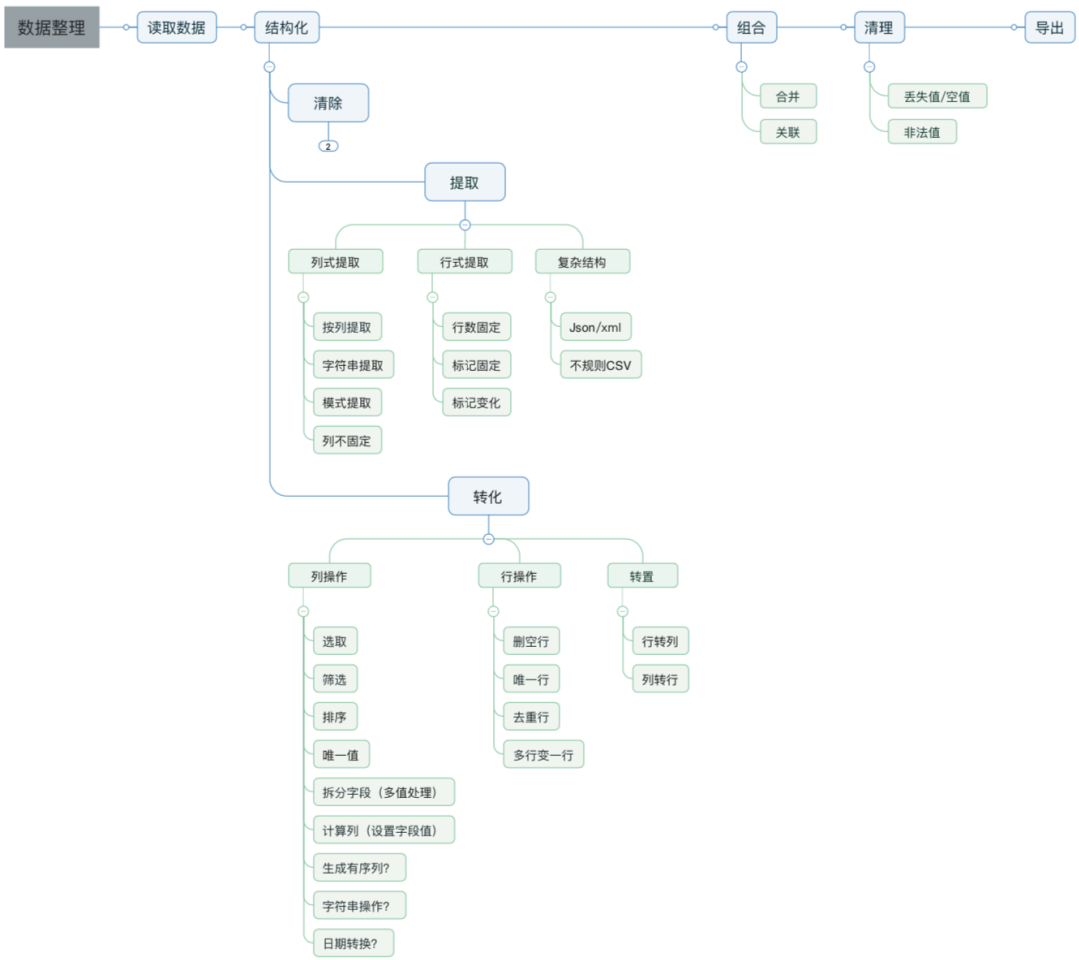
集算器 SPL 是专门为处理结构化数据设计的 DSL(Domain Specific Language)，不像通用语言，目标范围涵盖一切，通过短时间练习就能轻松掌握。分步式处理、中间结果引用，面对复杂需求比 SQL 更容易实现。数据结构简单、语法简洁，比 Python 更容易学习，更容易使用。

调试轻松


Execute/Debug/Step
Set breakpoint
Real-time system info output
WYSIWYG-style interface that enables easy debugging and convenient intermediate result reference
Simple syntax, natural & intuitive computing logic
Ready-to-use
Easy-to-debug

敏捷数据整理

总览



获取

	A	
1	=file("Migration.txt").read@n()	/读取普通文本，生成字符串数组，@n 是按行分隔
2	=file(“D.csv”).import@tc(name,sex,age)	/读取 CSV 文件，生成二维表，@t 首行为列名称，@c 逗号分隔
3	=file("sales_2013.xlsx").importxls@t()	/读取 Excel 文件，生成二维表，无参数（默认，第一个工作簿所有数据）

4	=file("sales_2013.xlsx").importxls@t(ID,Inv,Amount;1,3:40)	/读取 Excel 文件，有参数（分号前是要读入的列名称，1 指第一个工作簿，3:40 指读入 3~40 行的数据）
5	=wsc=httpfile("http://localhost:6080/myweb/servlet/testServlet?table=employee&type=json")	/建立 http 连接
6	=wsc.read().import@j()	/读取 http 接口，解析字符串成 json 格式字符，生成二维表
7	=connect("orc1").query("select * from orders where amount>=10000")	/读取数据库，orc1 是连接名称（设置参见“简单示例”）
8	=mcon=mongo_open("mongodb://127.0.0.1:27017/raqdb")	/连接 mongodb
9	=mongo_shell(mcon,"student.find()").fetch()	/读取 mongodb，生成二维表

结构化

清除

①文本 Migration.txt 是从网页表格数据直接复制来的，包含有为呈现使用的占位符和表头数据，需要再进一步整理前先清除掉。

```
Unemployment rates,,,,,,,,,,,,,,,,
As a percentage of total labour force,,,,,,,,,,,,,,,,
,Men,,,,,,,,Women,,,,,,,
,Native,,,,Foreign..born,,,,Native,,,,Foreign..born,,,
,1995,2000,2003,2004,1995,2000,2003,2004,1995,2000,2003,2004,1995,2000,2003,2004
Australia,8.4,6.6,6.0,5.6,10.6,6.5,6.5,5.5,7.7,5.8,6.1,5.7,9.6,7.0,6.5,5.6
Austria,3.6,4.3,4.4,4.3,6.6,8.7,9.7,11.2,4.6,4.2,4.0,4.3,7.3,7.2,6.6,10.7
Belgium,6.3,4.2,6.0,5.6,16.9,14.7,18.3,14.9,11.2,7.4,6.9,7.5,23.8,17.5,17.3,15.0
Canada,8.6,5.7,6.5,..,10.4,6.1,7.8,..,9.8,6.2,5.9,..,13.3,8.7,9.9,..
Czech Republic,..,..,5.8,7.0,..,..,9.0,12.4,..,..,9.6,9.6,..,..,15.7,13.5
Denmark,6.4,3.4,3.8,4.6,20.5,9.5,8.8,11.8,8.4,4.3,4.2,5.2,20.7,9.6,8.7,12.7
Finland,17.7,10.3,10.9,9.9,..,..,18.4,21.3,16.1,12.0,9.7,10.2,..,..,20.0,25.3
```

	A	
1	=file("Migration.txt").read@n().(replace(~,"..","")).export()	/读入文本，全局清理无用字符
2	=A1.import@c()	/将字符串 A1，按逗号分隔，生成二维表
3	=A2.to(6,)	/删除头部无用数据，只选取要处理的数据

A2：

_1	_2	_3	_4	_5	_6	_7	_8	_9	_10	_11	_12	_13	_14
Unemployment rate													
As a percentage of													
	Men								Women				
	Native				Foreignborn				Native				Foreignb
	1995	2000	2003	2004	1995	2000	2003	2004	1995	2000	2003	2004	1995
Australia	8.4	6.6	6.0	5.6	10.6	6.5	6.5	5.5	7.7	5.8	6.1	5.7	9.6
Austria	3.6	4.3	4.4	4.3	6.6	8.7	9.7	11.2	4.6	4.2	4.0	4.3	7.3
Belgium	6.3	4.2	6.0	5.6	16.9	14.7	18.3	14.9	11.2	7.4	6.9	7.5	23.8
Canada	8.6	5.7	6.5		10.4	6.1	7.8		9.8	6.2	5.9		13.3
Czech Republic			5.8	7.0			9.0	12.4			9.6	9.6	
Denmark	6.4	3.4	3.8	4.6	20.5	9.5	8.8	11.8	8.4	4.3	4.2	5.2	20.7
Finland	17.7	10.3	10.9	9.9			18.4	21.3	16.1	12.0	9.7	10.2	
France	9.1	7.7	7.3	8.0	16.6	14.5	15.4	13.8	13.6	11.3	9.2	9.9	19.0

A3:

_1	_2	_3	_4	_5	_6	_7	_8	_9	_10	_11	_12	_13	_14
Australia	8.4	6.6	6.0	5.6	10.6	6.5	6.5	5.5	7.7	5.8	6.1	5.7	9.6
Austria	3.6	4.3	4.4	4.3	6.6	8.7	9.7	11.2	4.6	4.2	4.0	4.3	7.3
Belgium	6.3	4.2	6.0	5.6	16.9	14.7	18.3	14.9	11.2	7.4	6.9	7.5	23.8
Canada	8.6	5.7	6.5		10.4	6.1	7.8		9.8	6.2	5.9		13.3
Czech Repu			5.8	7.0			9.0	12.4			9.6	9.6	
Denmark	6.4	3.4	3.8	4.6	20.5	9.5	8.8	11.8	8.4	4.3	4.2	5.2	20.7
Finland	17.7	10.3	10.9	9.9			18.4	21.3	16.1	12.0	9.7	10.2	
France	9.1	7.7	7.3	8.0	16.6	14.5	15.4	13.8	13.6	11.3	9.2	9.9	19.0

②文本 Sample.csv 每行头尾包含方括号，需提前清除。

[[04ae46c177169feac5f697eexxxx,1418601075,1.375579,103.960797,null, 1000.0]]

[[04ae46c177169feac5f697eexxxx,1418602016,1.381164,103.966164,null, 1000.0]]

[[04ae46c177169feac5f697eexxxx,1418603148,1.381164,103.966164,null, 1000.0]]

[[04ae46c177169feac5f697eexxxx,1418601994,1.381164,103.966164,null, 1000.0]]

...

	A	
1	=file("Sample.csv").import@c()	/读入文本，生成二维表
2	=A1.run(#1=replace(#1,"[[",""),#6=replace(#6,"]]",""))	/列上清理，第 1，6 列去括号

A1:

_1	_2	_3	_4	_5	_6
[[04ae46c1771	1418601075	1.375579	103.960797		1000.0]]
[[04ae46c1771	1418602016	1.381164	103.966164		1000.0]]
[[04ae46c1771	1418603148	1.381164	103.966164		1000.0]]
[[04ae46c1771	1418601994	1.381164	103.966164		1000.0]]

A2:

_1	_2	_3	_4	_5	_6
04ae46c17716	1418601075	1.375579	103.960797		1000.0
04ae46c17716	1418602016	1.381164	103.966164		1000.0
04ae46c17716	1418603148	1.381164	103.966164		1000.0
04ae46c17716	1418601994	1.381164	103.966164		1000.0

提取

列式提取

按列提取，续①

	A	
1	=file("Migration.txt").read@n().(replace(~,"..","")).export()	/读入文本，清理无用字符
2	=A1.import@c()	/按逗号分隔，生成二维表
3	=A2.to(6,)	/删除头部无用数据，只选取要处理的数据

4	=A3.new(_1:Country,"Men":Sex,"Native":Born,_2:Y1995,_3:Y2000,_4:Y2003,_5:Y2004)	/new()，新建二维表；_1:Country 按列位置抽取，给列命名 Country；"Men":Sex 初始化固定列
5	=A3.new(_1:Country,"Men":Sex,"Foreign..born":Born,_6:Y1995,_7:Y2000,_8:Y2003,_9:Y2004)	
6	=A3.new(_1:Country,"Women":Sex,"Native":Born,_10:Y1995,_11:Y2000,_12:Y2003,_13:Y2004)	
7	=A3.new(_1:Country,"Women":Sex,"Foreign..born":Born,_14:Y1995,_15:Y2000,_16:Y2003,_17:Y2004)	
8	=[A4,A5,A6,A7].conj().select(Country!=null)	/合并二维表，过滤空行

A4：

Country	Sex	Born	Y1995	Y2000	Y2003	Y2004
Australia	Men	Native	8.4	6.6	6.0	5.6
Austria	Men	Native	3.6	4.3	4.4	4.3
Belgium	Men	Native	6.3	4.2	6.0	5.6
Canada	Men	Native	8.6	5.7	6.5	
Czech Republic	Men	Native			5.8	7.0
Denmark	Men	Native	6.4	3.4	3.8	4.6
Finland	Men	Native	17.7	10.3	10.9	9.9
France	Men	Native	9.1	7.7	7.3	8.0

字符串提取

文本 T2.txt 中都是形如下行的串，需要提取含有字符 US 前的州名（LA）和 COOP:后的数字

COOP:166657,'New Iberia Airport Acadiana Regional LA US',200001,177,553

COOP:177562,'Bobo Dioulasso Airport BF',200001,322,682

COOP:179534,'La Tapoa Airport NE',200002,408,514

COOP:196410,'Caribou Municipal Airport ME US',200003,436,658

……

	A	B
1	=file("T2.txt").import@c()	
2	=A1.select(pos(_2, "US")!=null)	/过滤第二列，字符串必须含有"US"
3	=A2.derive(mid(_2, pos(_2," US")-2, 2):State)	/找到" US"的位置，左挪 2 个位置，向右提取 2 个字符
4	=A3.derive(right(_1, pos(_1,":")+1):ID)	/找到":"的位置，向右提取到末尾

集算器也提供了对正则表达式的支持以应对复杂的拆解需求。不过由于正则表达式的使用难度较大且性能较差，一般建议仍然用常规方法实现。

A2:

_1	_2	_3	_4	_5
COOP:166657	'New Iberia Airport Acadiana Regional LA US'	200001	177	553
COOP:196410	'Caribou Municipal Airport ME US'	200003	436	658
COOP:140559	'Robinson Army Airfield Camp Robinson AR US'	200004	152	880
COOP:176859	'Hilton Head Airport Hilton Head Island SC US'	200005	230	639
COOP:144475	'Salt Lake City International Airport UT US'	200006	246	673

A3:

_1	_2	_3	_4	_5	State
COOP:166657	'New Iberia Airport Acadiana Regional LA US'	200001	177	553	LA
COOP:196410	'Caribou Municipal Airport ME US'	200003	436	658	ME
COOP:140559	'Robinson Army Airfield Camp Robinson AR US'	200004	152	880	AR
COOP:176859	'Hilton Head Airport Hilton Head Island SC US'	200005	230	639	SC

A4:

_1	_2	_3	_4	_5	State	ID
COOP:166657	'New Iberia Airport Acadiana Regional LA US'	200001	177	553	LA	166657
COOP:196410	'Caribou Municipal Airport ME US'	200003	436	658	ME	196410
COOP:140559	'Robinson Army Airfield Camp Robinson AR US'	200004	152	880	AR	140559
COOP:176859	'Hilton Head Airport Hilton Head Island SC US'	200005	230	639	SC	176859

行式提取

行数固定

文本 Crime.txt，多行形成一组数据，且行数固定。

```
Reported crime in Alabama,
,
2004,4029.3
2005,3900
2006,3937
2007,3974.9
2008,4081.9
,
Reported crime in Alaska,
,
2004,3370.9
2005,3615
2006,3582
2007,3373.9
2008,2928.3
,
…
```

	A	B	C
1	=file("./Crime.txt").read@n()		/将文本读入成字串集合
2	=create(State,Year,Num)		/建立目标结果集
3	for A1.group((#-1)\8)		/按行号分组，每 8 行一组，(#-1)代表最末尾的行号
4		=output(A3)	/打印每个分组观察
5		=replace(A3.m(1),",","")	/获取每组第一个成员，将”,”清除
6		=state=right(B5,pos(B5,"in ")+3)	/根据”in ”位置，提取州名称
7		=A3.to(3,7)	/获取每组中 3~7 的成员
8		=B7.(state+","+~)	/本组预插入字符串集合
9		>B8.run(A2.record@i(~.array(),1))	/本组信息插入目标结果集，1 代表追加，改成 insert

标记固定

每段信息均包括“Reported”，这时只要将前面改成：

	A	B	C
1	=file("./Crime.txt").read@n()		/将文本读入成字串集合
2	=create(State,Year,Num)		/建立目标结果集
3	for A1.group@i(left(~,18)=="Reported crime in ")		/出现"Reported crime in "时会产生一个新分组
4		=output(A3)	/打印每个分组观察
5		…	
6			

标记变化

同一段信息的每一行都有相同属性的前缀（比如该段日志所属的用户号等），当这个前缀发生变化时就表示开始另一段信息了，这时仍然只要简单地修改 A3 代码即可处理：

3	for A1.group@o(left(~,6))	/前 6 个字符变化时产生一个新组
3	for A1;left(~,6)	/前 6 个字符变化时另起一轮循环

复杂结构

json/xml

Java 有足够多的类库用于解析和生成 json/xml，但缺乏后续计算能力。集算器支持多层结构数据，可以不丧失信息地将 json/xml 解析成可计算的内存数据表进一步处理。

设有如下格式的 json 数据：

```
{
 “order”:[
     {
      “client”:”raqsoft”,
      “date”:”2015-6-23”,
      “item” : [
           {
           “product”:”HP Laptop”,
           “number”:4,
           “price”:3200
          },
          {
           “product”:”DELL Server”,
           “number”:1,
           “price”:22100
          }]
     },…]
}
```

要写入数据库中 order 表，结构为：orderid,client,date；和 orderdetail 表，结构为：orderid,seq,product,number,price 的 orderdetail 表，orderid 和 seq 按顺序生成即可。

	A	
1	=file(“data.json”).read().import@j().order	
2	=A1.new(#:orderid,client,date)	
3	=A1.news(item;A1.#:orderid,#:seq,product,number,price)	
4	>db.update@i(A2,order)	
5	>db.update@i(A3,ordedetail)	

集算器可将多层 json 串解析成多层数据集，A2 的 item 字段取值又是一个表。
除了解析外，也可用集算器将多层数据集生成多层 json 串。

不规则 CSV

数据格式如下，要求每行从开始都有的字段各分一列，其他不够有多少项归为一列

```
col1,col2,clo3,col4
word1,date1,date2,port1,port2,....some amount of port
word2,date3,date4,
....
```

	A	B
1	=file("file.txt").import@st()	
2	=A1.(#1.split@c()).new(~(1):col1,~(2):col2,~(3):col3,~.to(4,).concat@c():col4)	/第 1-3 项各分一列，从第 4 项开始分为一列

转化（节选）

一般操作

	A	
1	=file("d:\\sales.xlsx").importxls@t()	/读入 Excel 中的所有字段
2	=A1.new(Client,OrderDate,Amount,SellerId)	/选出
3	=A2.select(year(OrderDate) <= 2015&&Amount > 5000||year(OrderDate) >= 2015)	/过滤
4	=A3.sort(Client)	/排序
5	=A3.id(Client)	/唯一值

多行变一行

将表一中 identifer 相同的图片链接合并在一起,忽略属性值为空的数据，结果如表二。

表一

Property1	Identifier	Property2	Image-Link
	A		Link_1
something	A	something	Link_2

Property1	Identifier	Property2	Image-Link
	A		Link_3
something	B	something	Link_1
	B		Link_2
	B		Link_3

表二

Identifier	Property1	Property2	Image-Link
A	something	something	Link_2, Link_1, Link_3
B	something	something	Link_1, Link_2, Link_3

代码示例：

	A	B
1	=file("OpenfineCase.txt").import@t()	/读取文本，生成二维表
2	=A1.group(Identifier)	/用 Identifier 分组
3	=A2.new(~.Identifier:Identifier, ~.id(Property1).select(~!=null).concat(","):Property1, ~.id(Property2).select(~!=null).concat(","):Property2, ~.id(ImageLink).select(~!=null).concat(","):ImageLink)	/各字段忽略空值，在组内合并

如有相同处理的字段很多：

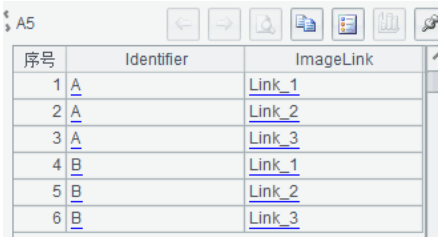
	A	B
1	=file("OpenfineCase.txt").import@t()	/读取文本，生成二维表
2	=A1.new(Identifier,Property1,Property2,ImageLink)	/调整字段顺序
3	=A2.group(Identifier)	/用 Identifier 分组
4	=A2.fname().to(2,).("~.id("+~+").select(~!=null).concat(\",\"):"+~)	/提取需要相同处理的字段名
5	=A3.new(Identifier:Identifier,${A4.concat(",")})	/各字段忽略空值，在组内合并

OpenRefine 对比参考:
https://github.com/OpenRefine/OpenRefine/wiki/Common-Use-Case-Examples

一行变多行

	A	B
4	...	/上例结果，表二

5	=A4.news(ImageLink.split@c(); Identifier, ~:ImageLink)	/逗号分隔项，一行变多行



日期转换

详参： SPL 的日期时间函数

行列转换

表一：

Class	Subject	MaxScore
Class 1	Math	77
Class 1	English	84
Class 1	Physics	75
Class 2	English	91
Class 2	Physics	66
Class 2	Math	78

表二：

Class	Math	English	Physics	MaxScore
Class 1	77	84	75	77
Class 2	78	91	66	84

	A	B
1	...	获取表一
2	=A1.pivot(Class;Subject,MaxScore)	/行转列，表一转表二
3	=A2.pirvot@r(Class;Subject,MaxScore)	/列转行，表二转表一

组合（多集合）

合交差并（纵向）

两个文件，早一点的是 old.csv，晚一点的是 new.csv,需要分别找出新增的、删除的、修改的数据行。源文件如下：

Olde.csv	New.csv
userName,date,saleValue,saleCount Rachel,2015-03-01,4500,9 Rachel,2015-03-03,8700,4 Tom,2015-03-02,3000,8 Tom,2015-03-03,5000,7 Tom,2015-03-04,6000,12 John,2015-03-02,4000,3 John,2015-03-02,4300,9 John,2015-03-04,4800,4	userName,date,saleValue,saleCount Rachel,2015-03-01,4500,9 Rachel,2015-03-02,5000,5 Ashley,2015-03-01,6000,5 Rachel,2015-03-03,11700,4 Tom,2015-03-03,5000,7 Tom,2015-03-04,6000,12 John,2015-03-02,4000,3 John,2015-03-02,4300,9 John,2015-03-04,4800,4

	A	B
1	=file("d:\\old.csv").import@tc()	=file("d:\\new.csv").import@tc()
2	=conj=[A1,B1].conj()	/两个集合的合集
3	=A1.sort(userName,date)	=A2.sort(userName,date)
4	=inter=[B3,A3].merge@i(userName,date)	/两个集合的交集
5	=new=[B3,A3].merge@d(userName,date)	/新旧销售记录的差集（新增的记录）
6	=delete=[A3,B3].merge@d(userName,date)	/旧新销售记录的差集（删除的记录）
7	=diff=[B3,A3].merge@d(userName,date,saleValue,saleCount)	/关键字作为普通字段计算差集，找到修改过的所有记录
8	=update=[diff,new].merge@d(userName,date)	/修改过的记录和新增记录之间的差集，等于更新的记录
9	return new,delete,update	/返回多个数据集

关联（横向）

emp.xls 的 Eid 字段对应 sales.xls 的 SellerId 字段，需要用 emp.xls 关联到 sales.xls。代码如下：

	A	B
1	=emp=file("d:\\emp.xls").importxls@t()	=sales=file("d:\\sales.xls").importxls@t()
2	=join@1(sales:s,SellerId;emp:e,EId)	
3	=A2.new(s.OrderID, s.Client, s.SellerId, s.Amount, s.OrderDate,e.Name, e.Dept, e.Gender)	

函数 join 执行连接运算，并将两个表改名为 s 和 e，默认内连接，@1 表示左连接，@f 表示全连接。之后从连接的表中取得需要的字段，组成新的结构化二维表格。结果：

A4

s.OrderID	s.Client	s.SellerId	s.Amount	s.OrderDate	e.Name	e.Dept	e.Gender
26	TAS	1	2142.4	2009-08-05			
84	GC	1	88.5	2009-10-16			
133	HU	1	1419.8	2010-12-12			
33	DSGC	1	613.2	2009-08-14			
32	JFS	3	468.0	2009-08-13	Rachel	Sales	F
43	KT	3	2169.0	2009-08-27	Rachel	Sales	F
99	RA	3	1731.2	2009-11-05	Rachel	Sales	F

清理

略

导出

	A	
1	=file("Migration.txt").read@n()	/读取普通文本，生成字符串数组，@n 是按行分隔
2	=file(“D.csv”).import@tc(name,sex,age)	/读取 CSV 文件，生成二维表，@t 首行为列名称，@c 逗号分隔
3	=file("sales_2013.xlsx").importxls@t()	/读取 Excel 文件，生成二维表，无参数（默认，第一个工作簿所有数据）
4	=file("sales_2013.xlsx").importxls@t(ID,Inv,Amount;1,3:40)	/读取 Excel 文件，有参数（分号前是要读入的列名称，1 指第一个工作簿，3:40 指读入 3~40 行的数据）
5	=wsc=httpfile("http://localhost:6080/myweb/servlet/testServlet?table=employee&type=json")	/建立 http 连接
6	=wsc.read().import@j()	/读取 http 接口，解析字符串成 json 格式字符，生成二维表
7	=connect("orc1").query("select * from orders where amount>=10000")	/读取数据库，orc1 是连接名称（设置参见“简单示例”）
8	=mcon=mongo_open("mongodb://127.0.0.1:27017/raqdb")	/连接 mongodb
9	=mongo_shell(mcon,"student.find()").fetch()	/读取 mongodb，生成二维表

简化复杂计算

多样性数据源

许多业务计算的数据源并不只来源于关系数据库，还可能是 NoSQL 数据库、本地文件、WebService 数据等。集算器自有的计算能力可以使这些计算能力不一的多样性数据获得通用一致的计算能力，而这将意味着更低的移植成本以及学习难度。

下面举两例说明常见的 json 计算问题。

1、JSON 分组汇总：order.JSON 存储着订单记录，现在要按时间段汇总每个月每个客户贡献的销售额，部分源数据如下：

```
[{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"}…
```

集算器对应的代码：

	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)

将 JSON 文件读为二维表，进行性条件查询，再进行分组汇总，其中 argBegin、argEnd 是参数。结果如下：

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、有一个 testServlet 可以返回 json 格式的员工信息字符串，需要按条件查询信息，并将结果以 json 形式返回 JAVA 主程序：

	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)

读取 httpfile 对象，解析 json 格式字符串，生成二维表，按照条件过滤数据，再转换为 json 格式字符串。其中参数 where 是动态的查询条件，形如：BIRTHDAY>=date(1981,1,1) && GENDER=="F"。

多源混合运算

关系数据库的事务一致性能力目前尚没有有效的替代者，交易系统仍然有必要使用关系数据库来建设。

这种情况下，要实现 T+0 全数据量的实时报表，我们就得把历史数据继续存放在当期的交易数据库中一起计算，历史数据常常要庞大得多，这会要求我们建设更大容量的数据库，成本当然会很高。而且即使愿意支付成本，这个数据量也不可能一直增长，太大了会影响到交易业务的性能，这就不可容忍了。

通常的办法是把部分历史数据被移出来做个分数据库，这样可以保证交易系统的正常运转，但要实现 T+0 报表就麻烦得多，会涉及到跨库运算。

许多数据库都支持跨库运算，但一般都要求同类型的数据库，但历史数据和当期交易数据的要求不同，数据量更大但不要求事务一致性，很可能使用另一种数据仓库来存储。

而且，即使是同构的数据库，数据库的跨库运算的方法一般也是将另一个库中的数据表映射成本库数据表，实际运算还是一个数据库在做，而且还多出许多数据传递的通讯成本，性能和稳定性都不好。

使用集算器就可以很好地完成这个混合计算任务了。

集算器自己有计算引擎，不依赖于数据库，各个数据库内的数据计算仍由各库进行。集算器可以使用多线程向各数据库同时发出 SQL 语句，由这些数据库并行执行，将各自的运算结果返回到集算器再汇总处理后传给报表工具去呈现。

显然，这种机制还方便横向扩展，历史库可以有多个，是否同类型的也无所谓。

而且，集算器还有服务器方式的集群运行模式，在集算服务器的支持下，历史数据不必一定存放到数据库中，还可以存储在 IO 性能更好的文件系统中，配合集群计算，可以在更低的成本下获得更好的性能。

下面举例说明：

某电信企业用库表 userService 存储用户服务信息，T+0 报表需要呈现各类电信产品的通话时长、通话次数、拨打本地时长、拨打本地次数。实际使用中发现数据量太大，查询效率很低，导致报表性能不佳。

改用计算层可以大幅提升性能，具体做法是将 userService 分地区存储于多个数据库，再用计算层进行并行计算。

以集算器为例，脚本如下：

	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)**	

语句 fork 并行开启四线程，每线程从对应的数据库取数，并将分组汇总的结果返回主线程。主线程合并各子线程计算结果，再次执行分组汇总，最终获得报表需要的结果集。

一般报表工具并不具备这种并行取数汇总的能力，需要借助 Java 等高级语言完成，但 Java 并行代码难以书写，缺乏结构化计算能力，比起独立的计算层尚有差距。

库外存储过程

数据库的存储过程本身编写难度并不小，遍历式计算代码的性能也不佳，而且可移植性很差，采用集算器相当于是算法外置的存储过程

下面举例说明：

1、计算“在每个州的销量均在前 10 名的优秀产品”

库表 stateSales 存储各州各产品的销售信息，有 3 个字段：state，product，amount，记录有重复。需要去除重复，并计算“在每个州的销量均在前 10 名的优秀产品”。

原先用存储过程计算优秀产品，代码如下：

```
01  create or replace package salesPkg
02  as
03      type salesCur is ref cursor;
04  end;
05  CREATE OR REPLACE PROCEDURE topPro(io_cursor OUT salesPkg.salesCur)
06  is
07    varSql varchar2(2000);
08    tb_count integer;
09  BEGIN
10   select count(*) into tb_count from dba_tables where table_name='TOPPROTMP';
11   if tb_count=0 then
12   strCreate:='CREATE GLOBAL TEMPORARY TABLE TOPPROTMP (
                stateTmp NUMBER not null,
                productTmp varchar2(10)  not null,
                amountTmp NUMBER not null
          )
          ON COMMIT PRESERVE ROWS';
13   execute immediate strCreate;
14   end if;
15   execute immediate 'truncate table TOPPROTMP';
16   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;
17    OPEN io_cursor for
18    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;
```

改用集算器实现同样的算法则可以避免这些耦合问题。以下是集算器脚本说明：

	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()**	

在 A2 去除重复，A3 将数据按州分组，B3 取得每个组排名前 10 的产品的组内序号，A4 按序号取得产品，A5 进行组间交集运算。

上述脚本清晰简单且能解释执行，开发效率高，维护方便。该脚本只需数据库 select 权限，修改和编辑无需额外权限。

2、计算“销售额占到一半的前 n 个客户”

有些数据库没有窗口分析函数（eg. Mysql），有些数据库没有存储过程（eg. Vertica），当遇到复杂的数据计算，往往只能通过 Python,R 等外部脚本来实现，但这些脚本语言和主流工程语言（Java）集成性不好，如果直接用工程语言实现类似 SQL 函数和存储过程的功能，经常只是针对某个计算需求编写冗长的代码，代码几乎不可复用。

即便拥有强大的分析函数，实现稍复杂的逻辑其实也不算容易，比如下面这种常见的业务计算，找出“销售额占到一半的前 n 个客户，并按销售额从大到小排序”，在 Oracle 中 SQL 实现如下：

```
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
```

按照销售额累计值从小到大排序，通过累计值大于“一半销售额”的条件，逆向找出占到销售额一半的客户。为了避免窗口函数在计算累计值时对销售额相同的值处理出现错误，用子查询先计算了排名。

下面是用集算器实现相同逻辑的代码：

	A	B
1	=connnect("verticalCon")	/建立数据库连接
2	=A1.query("select * from sales").sort("SALESAMOUNT:-1")	/按销售额从大到小排序
3	=A2.cumulate(SALESMOUNT)	/计算出销售额的累计值序列，此处取代数据库窗口函数
4	=A3.m(-1)/2	/根据最后的累计值，算出“一半销售额”
5	=A3.pselect("~>=A4")	/在累计值序列找到大于“一半销售额”值的所在位置
6	=A2(to(A5))	/找出“一半销售额”值所在位置及其前面的所有记录
7	>A1.close()	/关闭数据库连接
8	return A6	/结果返回

从上述代码我们可以看到，集算器利用一套简洁的语法取代了需嵌套 SQL+ 窗口函数才能实现的逻辑，并且具有通用一致性（任何数据源代码一致）。

SQL 难点解决(节选)

分组子集

除了数据表外，SQL 没有显式的集合数据类型，在分组时会强迫计算出聚合值。但有时我们感兴趣的不只是聚合值，还有分组子集，这时 SQL 就很难处理，要用子查询反复计算。

集算器有集合数据，也提供了返回子集的分组函数。这样就能方便地处理分组后运算。

比如找出总分 500 分以上的学生的各科成绩记录。SQL 需要先分组计算出各学生总分，从中过滤出 500 分以上的，再用这个名单与原成绩记录 JOIN 或用 IN 判断，较麻烦且要重复取数。而集算器则可以按自然思路写出来：

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

这种分组后却要返回子集明细记录的情况很多，分组聚合是用来实现某种过滤的中间步骤而不是结果。

有时即使是只要返回聚合值，但聚合计算较为特别，难以用简单聚合函数表示时，也需要保留分组子集用于再计算。

这类计算在现实中并不少见，但因为计算复杂，常常涉及较多的业务背景，不适合举例说明，这里改造了一个简化后的例子：

设有用户登录表 L 结构为：user（帐号），login（登录时刻）；现要计算出每个帐号最后登录时刻以及该时刻前三天内的登录次数。

找出最后登录时刻很容易，但如果不保留分组子集时则很难计算出那个时间段登录次数。用 SQL 需要先分组计算出最后登录时间，与原表 JOIN 后过滤相应时间段的记录再次分组汇总，不仅麻烦而且计算效率很低。而使用集算器保留了分组子集则容易实现分步式计算：

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

其中~就是按 user 分组后的子集。

如果数据有序还可以用高效的方法计算：

	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)	

有序聚合

取出每组的前 N 条、最大值对应记录等也是较常见的运算。显然，这些都可以用保留分组子集的方法实现，但由于这类运算较常见，集算器将其理解成某种聚合而提供了专门的函数，这样就可以采用和普通的分组汇总基本一致的处理方式。

SQL 没有集合数据类型，离散性也不好，无法提供返回结果是记录引用集合的聚合函数，这种运算就需要子查询等繁琐的方式，经常还会导致大排序而损失性能。

先看最简单的情况，用户登录表 L 结构为：user、login（登录时刻）、IP-address、…；列出每个用户首次登录的记录。

SQL 可以用窗口函数生成组内排序序号，并取出所有序号为 1 的记录，但窗口函数是在结果集上再计算的，因而必须用子查询再过滤的形式，写法有些复杂。而不支持窗口函数的数据库写起来就会更困难了。

集算器提供了 group@1 方法可直接取出每个分组的第一个成员。

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

这类日志数据经常存在文件中，且已经对时刻有序，用集算器就可以直接取出第一条而不必再排序。数据量大到内存放不下时也可以基于游标实现类似的运算。

股价表 S 的结构为：code（股票代码）、date、cp（收盘价）；计算每支股票最近的涨幅。

计算涨幅涉及到最后两个交易日的记录，使用 SQL 需要用两重窗口函数分别实施组内跨行计算再取出结果的第一行，写法繁琐。集算器提供了 topN 聚合函数，利用集合数据直接返回多条记录作为汇总值参与进一步计算。

	A	
1	=db.query("select * from S")	
2	=A1.groups(code;top(2,-date))	/最后 2 个交易日的数据
3	=A2.new(code,#2(1).cp-#2(2).cp:price-rises)	/计算涨幅

聚合函数并不会先计算出分组子集，而是直接在已有结果上累积，这样可获得更高的性能，而且在数据量大到内存放不下时还可以基于游标工作。

如果数据已有序，则可以更高效地用位置取出相应记录：

	A	
1	=db.query("select * from S order by date desc")	
2	=A1.groups(code;top(2,0))	/直接取前 2 条
3	=A2.new(code,#2(1).cp-#2(2).cp:price-rises)	

取出最大值对应记录、第 1 条最后 1 条等类似计算都是 topN 聚合的特例了。

有序分组

SQL 只提供与次序无关的等值分组，但有时分组的键值并不能在每条记录中找到，而是和记录的次序有关，这种情况，用 SQL 又需要使用窗口函数（或其它更麻烦的手段）制造出序号才能实现。

集算器提供了与次序相关的分组机制，方便用于与连续区间相关的计算。

收支表 B 结构为：month、income、expense；找出连续亏损达三月或以上的那些月份的记录。

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

group@o 表示在分组时只比较相邻记录，如果相邻值发生变化则会分出一个新组。这样就可以根据收入支出的比较把收支记录分成赢利、亏损、赢利、…这样的组，然后取出其中亏损且成员不少于 3 的组再合并起来。

还是这个表，希望计算收入最长连续增长了几个月。可以设计这样的分组机制：收入增长时和上月分作一个组，收入下降时则分出一个新组，最后统计组成员的最大值。

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

group@i 将在条件变化时分出一个新组，即收入降低时。

在窗口函数的支持下，SQL 也能实现本例和上例的思路，但写法非常难懂。

区间合并也是常见的有序分组运算。设有事件发生区间表 T 有字段：S（开始时刻）、E（结束时刻）；现在要将这些区间中重叠部分去除后再计算该事件实际发生的总时长。

	A	
1	$select S,E from T order by S	
2	=A1.select(E>max(E{:-1}))	/去除被包含的条目
3	=A2.run(max(S,E[-1]):S)	/去除重叠时间段
4	=A2.sum(interval@s(max(S,E[-1]),E))	/计算总时长
5	=A2.run(if(S<E[-1],S[-1],S):S).group@o(S;~.m(-1).E:E)	/合并有重叠的时间段

这里给了多种目标的处理方法，充分利用了跨行运算和有序分组的特点。SQL 要实现这种运算简单用窗口函数已经做不到了，需要用到很难理解的递归查询。

位置利用

对于有序的集合，有时我们需要直接用序号访问成员。SQL 延用了数学上的无序集合概念，要生成序号再用条件过滤才能访问指定位置的成员，这对许多运算造成很大的麻烦。

集算器采用了有序集合机制，允许直接用序号访问成员，这类运算要方便得多。

比如经济统计中常用到的在众多价格中找出中位数：

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

位置还可以用于分组。事件表 E 结构为：no（序号）、time、act，动作有开始、结束两种，现在要统计事件持续的总时长，即每一对开始和结束之间的时间之和。

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

#表示记录序号，group((#-1)\2)即将数据每两个分成一组，然后针对每组计算时长再合计即可。

根据位置还能进行相邻跨行引用。设有股价表 S 结构为：date（交易日）、cp（收盘价）；现列出计算出股价超过 100 元的交易日及当日涨幅。

	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)	

pselect 函数将返回满足条件的成员位置，使用这些位置就可以方便地计算涨幅，而不必象使用窗口函数时事先计算出所有涨幅再过滤。

有时我们要求分组的结果是连续区间，要补齐中间缺省的空子集。SQL 实现这个过程很麻烦，要手工先造出连续不断的分组区间再 left join 要统计的数据表，子查询将不可避免。集算器则可以直接利用位置来实现对位分组。

简化的交易记录表 T 结构为：no、date、amount。现需要按周统计累计的交易金额，没有交易记录的周也要列出。

	A	
1	=db.query("select * from T order by date")	
2	>start=A1(1).date	
3	=interval(start,A1.m(-1).date)\7+1	/计算总周数
4	=A1.align@a(A2,interval(start,date)\7)	/按周分组，可能有空集
5	=A4.new(#:week,acc[-1]+~.sum(amount):acc)	/汇总并计算累计

4 大数据

游标技术

当数据量太大而不能全部读入内存时，这时一般采用游标技术来处理。

游标的机理很简单，针对一个大数据源，按顺序每次读入一段数据进内存来处理，处理完毕后释放这些内存再读下一段，循环至所有数据全部读取完毕，处理也跟着完成。

游标的原则是只从前向后单向移动，所有数据只遍历一次。许多大数据运算都可以用游标完成，比如求和计数，结果集不大的分组汇总等，排序和大结果集分组汇总也可以用游标实现，不过需要使用中间结果的缓存机制。

多级游标

和其它的游标机制不同，集算器在游标上封装了计算功能，可以针对游标计算出结果，许多运算的返回结果仍然是个游标，可以连续操作。

	A	
1	**=file(“Products.txt”).import().primary@i(id)**	/读入商品列表并建立索引
2	**=file(“Sales.txt”).cursor()**	/建立游标，准备遍历
3	**=A2.select(quantity<=10)**	/过滤，仍返回游标
4	**=A3.switch(productid,A1:id)**	/建立连接指针，仍返回游标
5	**=A4.groups(;sum(quantity*productid.price))**	/求和汇总

A2 建立原始游标，A3 在 A2 基础上过滤，A4 在 A3 基础上建立连接指针，A5 最后针对 A4 计算出汇总结果。实质的数据遍历动作只在 A5 中的计算过程时才实施，A2,A3,A4 仅仅是记录游标信息，并不实际运算。但这种多级游标的代码看起来象是一步步地处理游标，理解和编写都很简单。

SQL 型的数据库都提供了游标，但只有简单的 fetch 数据功能，没有封装运算，更没有多级游标，要在这种游标基本的完成复杂运算会相当麻烦。

程序游标

多级游标非常方便，但每个环节的处理都要用一个函数写出来。有时某个环节的运算处理较为复杂，很难用一个函数拼出来，最好是写成一段代码才方便理解及将来维护。

集算器提供了程序游标方法，可以用一段代码定义一个游标。

	A	B	
1			/游标处理程序 sub.dfx
2	**=file(“Sales.txt”).cursor()**		
3	**for A2,1000**	**…**	/每次取出 1000 条记录处理
…		**…**	
…		**return …**	/处理后的结果返回

子程序分批读入数据进行复杂处理后分批返回，返回数据的记录数和读入数据的记录数没有关系。

	A	
1		/主程序
2	**=cursor("sub.dfx")**	/使用子程序
3	**=A2.fetch(100)**	/当作普通游标使用

主程序将程序定义的游标当作普通游标使用，cursor 函数将会缓存子程序返回的数据，主程序请求时即返回，缓存的数据不足时再去继续执行子程序进一步获取，获取后子程序暂停执行等待下一次数据获取请求，直到数据取完或主程序要求关闭游标时子程序才会彻底退出释放。这样就可以用程序代码实现多级游标中的某些环节。

有序游标

用户行为分析是常见的运算，其特点是：单用户内的运算很复杂，但跨用户的运算几乎没有，比如计算用户最近几周在线时长增长率，房贷帐户的利息及余额等；单用户数据量很小，可内存处理，全用户数据量很大，无法全读内存。这样最好是有一种机制使得每次读入一个用户的数据处理，算完后再读下一个。

前面讲过的有序分组对于游标也有效，集算器可以从有序游标中每次读入一组数据，这样就能方便地实现这类运算。

	A	B	
1	**=file(“userlogs.txt”).cursor()**		/按用户 id 排序的源文件
2	**for A1;id**	…	/从游标中循环读入数据，每次读出一组 id 相同
3		…	/处理计算该组数据

只要游标源数据有序就可以，无论来自数据库还是文件都可以。

外存计算

数据量大过内存的情况还是更普遍的大数据计算场景，这时只能将数据存放在外存中。本质上外存是不能直接运算的，所谓外存运算是指将外存的大数据分批读入内存后处理，其中有大的中间结果还需要缓存到外存中。

和内存相比，外存（在物理上也就是硬盘了）不仅速度更慢，关键还在于缺乏频繁、小量、随机访问的能力；固态硬盘有了更强的随机访问能力，但仍不能做到频繁小量访问。对硬盘上计算的优化原则主要有这么三点：一是减少对硬盘的访问，因为它慢，要用 CPU 换硬盘时间；二是尽量顺序读取，特别是对于机械硬盘，同时还要权衡并行访问量；三是每次读入较多数据，避免频繁小量访问。

游标复用

有时我们需要针对同一批数据获得不同的统计值，如果是内存数据则无所谓，反复遍历也不会影响性能，但外存数据最好是一次遍历获得尽量多的结果，比如分组汇总时可以在一次遍历把每组的计数与合计都计算出来，SQL 也是这么设计的，一个 GROUP BY 语句中可以 SELECT 多个汇总值。

情况再复杂一些就不是这样了，举个简单例子：计算一个大数据集的中位数。我们要将数据集排序，然后再数出位置在中间的成员，而这要事先知道总共有多少成员。但是，SQL 的语法设计不能让我们在排序的同时把成员个数也数出来，需要事先取得计数，也就是要多多扫描一遍数据集，对于外存大数据集，这常常比运算本身的时间还要多了。

集算器提供了在游标上绑定延迟汇总计算的语法，可以在遍历游标时顺便计算其它统计值。这样就可以在为排序遍历时同时把个数计出来。

	A	
1	**=file(“data.txt”).cursor()**	
2	**>A1.groups@x(;count(1);n)**	/绑定后计算的汇总，在遍历时再计算
3	**=A1.sortx(key)**	/排序，遍历过程中处理 A2 的绑定计算

4	**=A1.v(n).#1**	/取出绑定计算的结果，即总记录数
5	**=A3.skip((A4-1)\2).fetch@x(2-A4%2).avg(key)**	/取出中位数记录并计算中位数

这段代码只在 A3 中将原始数据遍历了一次，A5 中遍历的排序后的数据。

类似地，针对一个大数据集按不同分组口径统计的情况也很常见，SQL 需要写成多句 GROUP BY 对数据遍历多次。而集算器则可以在游标上绑定多组计算，一次遍历完成。

分段并行

在计算总量不可减少的情况下，采用多线程并行可以让多 CPU 核分担计算量，对性能提高非常明显。

我们先看文本数据源的情况。文本是很常见的外存数据源，将文本读入内存时需要解析成相应的数据类型对象后才能运算，这个过程很慢，特别是碰到日期时间类型的数据，可以想象一下分析那个日期串会有多麻烦，经常会发生 CPU 时间会超过硬盘访问时间的现象。这时候如果能采用多线程就能有效地提高这个性能。

要并行处理需要将源文件分段，每个线程处理其中一段。文本文件一般是每一行对应一条记录，每一行长度不一定相同。按行数分段显然没有意义，这需要每次都从头遍历，完全起不到提高性能的目标。按字节分段不需要遍历，但有可能分段点正好落在行的中间，造成一行被拆进两段，数据就错误了。

借鉴 Hadoop 的搞法，集算器使用了自动去头补尾的字节分段机制，即分段开始点所在的行被舍弃，分段结束点所在的行会被补齐，这样将确保每一段都由完整的行构成，不会有数据错误。

结合前面说过的集算器并行代码，可以很方便地写出并行计算程序。

	A	B	
1	**=file(“data.txt”)**		/源文件
2	**fork 4**	**=A1.cursor@t(amount;A2:4)**	/分作 4 段并行，分别建立游标
3		**=B2.groups(;sum(amount):a)**	/遍历游标计算 amount 之和
4	**=A2.conj().sum(a)**		/汇总每个线程的结果

上述代码把 data.txt 分作 4 段，产生 4 个线程分别用游标遍历每一分段并计算其中 amount 列的和，最后再汇总每段的返回值得到整个文件的 amount 列总和。

文本解析的时间经常比计算要长得多，有时候只要解析能够并行，计算本身是否并行并不重要。集算器对于读取数据提供了简单的内置并行选项，如果对数据读取次序不关心，比如求和运算就不在乎次序，可以更简单地写出代码。

	A	
1	**=file(“data.txt”).cursor@tm(amount)**	/定义并行取数的游标
2	**=A1.groups(;sum(amount))._1**	/遍历游标并汇总 amount 列

上面代码中，游标取数时会自动启动多线程并行，但计算 amount 合计时是串行的。

内存计算的并行数量基本只受 CPU 数量的限制，而外存运算还将受到硬盘的限制。单片硬盘在逻辑上就不能并发访问，多线程同时访问硬盘不同文件会造成磁头频繁移动，这消耗的时间比读取数据还要长，这时要为每个线程设置较大的数据缓冲区，但这又造成内存的占用。需要根据实际情况来权衡，集算器提供了相应的配置手段控制线程数量及缓冲区。

数据库表的分段就远没有文件自由了，不大适合进行分段计算，对于数据密集型的任务最好还是由数据库自行完成。有些计算密集型的任务，计算占用时间很多，而且复杂的计算在数据库内实现难度实在太大，需要读出到外部实现，这时候也可以采用分段并行机制。

一种办法是直接建立多个分表，每个线程分别处理若干个分表的数据，同一个分表不能再拆分给多个线程处理，这种方案下要拆分出较多分表才能让各线程负担相对均匀，而在数据库建立的表太多并不是个好设计，所以一般会让线程数和分表数基本相同，这会导致并行数被事先确定，较为死板。

还可以使用 WHERE 条件来分段，这样并行数可以很灵活，但最好事先建立索引并按这个索引去 WHERE，否则每次 WHERE 都会导致全表遍历，浪费数据读取的时间。

我们测试发现，Oracle 等数据库的 JDBC 很慢，读出计算时 JDBC 会成为一个瓶颈。如果数据库本身负担不重，这时也可以采用分段并行的方法取数，从而缓解 JDBC 的性能损失。

	A	B	
1	**fork 4**	**=connect(db)**	/分 4 线程，要分别建立连接
2		**=B1.query@x(“select * from T where part=?”,A2)**	/分别取每一段
3	**=A1.conj()**		/合并结果

实测表明，并行取数在数据库负担不重时能达到数倍的性能提升。

数据存储

除了文本文件外，集算器还提供了自有格式的二进制文件。

集算器的二进制文件中已经记录了数据类型，在读出时不需要再解析，这样会比文本好快得多。而且，集算器为二进制文件做了压缩，同样数据占用的硬盘空间一般能比文本要小三分之一到一半，读取性能也会更好。不过，压缩比并非越高越好，解压缩会占用 CPU 时间，压缩比越高的算法占用 CPU 时间越长，集算器的压缩算法非常简单，几乎不占用 CPU 时间，当然压缩比不是非常高。

整体算下来，二进制文件比文本能快出 3 到 5 倍的样子，比用 JDBC 从数据库中取数也快，特别是针对 ORACLE 这种 JDBC 很慢的情况更有巨大优势。所以，如果数据要反复使用时，以二进制格式存放会有更大的性能优势。

从文本或数据库到二进制文件的转换程序也非常简单：

	A	
1	**=file(“data.txt”).cursor@t()**	/定义文本文件游标
2	**=file(“data.bin”).export@z(A1)**	/写成可分段的二进制文件

集算器的二进制文件也支持分段并行，代码与文本文件几乎相同：

	A	B	
1	**=file(“data.bin”)**		/源文件
2	**fork 4**	**=A1.cursor@b(amount;A2:4)**	/@b 表示二进制文件，其它参数相同
3		**=B2.groups(;sum(amount):a)**	/后续计算语法完全相同
4	**=A2.conj().sum(a)**		

5 集成性

与报表工具集成

集算器可以和 BIRT、JasperReport 等报表工具轻松集成，下面以和 BIRT 集成为例，简要介绍一下集成过程，具体操作细节要参考官网相关文档，与其他报表或 BI 系统集成也都类似。

加载 JDBC 驱动

将三个集算器基础 jar 包（集算器[安装目录]\esProc\lib 目录下），拷贝至 Birt [安装目录]\plugins\org.eclipse.birt.report.data.oda.jdbc_4.6.0.v20160607212\driver 下。注意：不同 birt 版本 driver 的上层目录名略有不同

```
dm.jar              集算器计算引擎及 JDBC 驱动包
icu4j_3_4_5.jar     处理国际化
jdom.jar            解析配置文件
```

加载部署文件

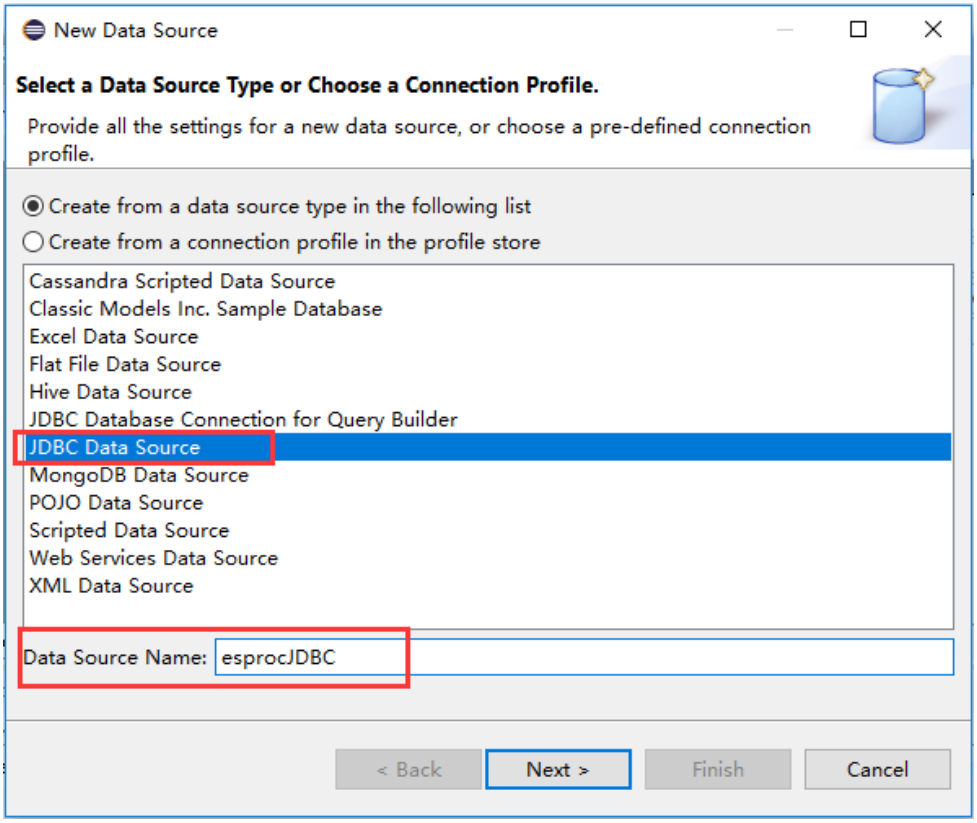
集算器还有个重要的配置文件 raqsoftConfig.xml（[安装目录]\esProc\config 目录下），该文件中配置了授权信息、集算器主路径、dfx 文件寻址路径等各类信息。同样将其拷贝至 Birt [安装目录]\plugins\org.eclipse.birt.report.data.oda.jdbc_4.6.0.v20160607212\driver 下。

打开 raqsoftConfig.xml，配置授权信息。

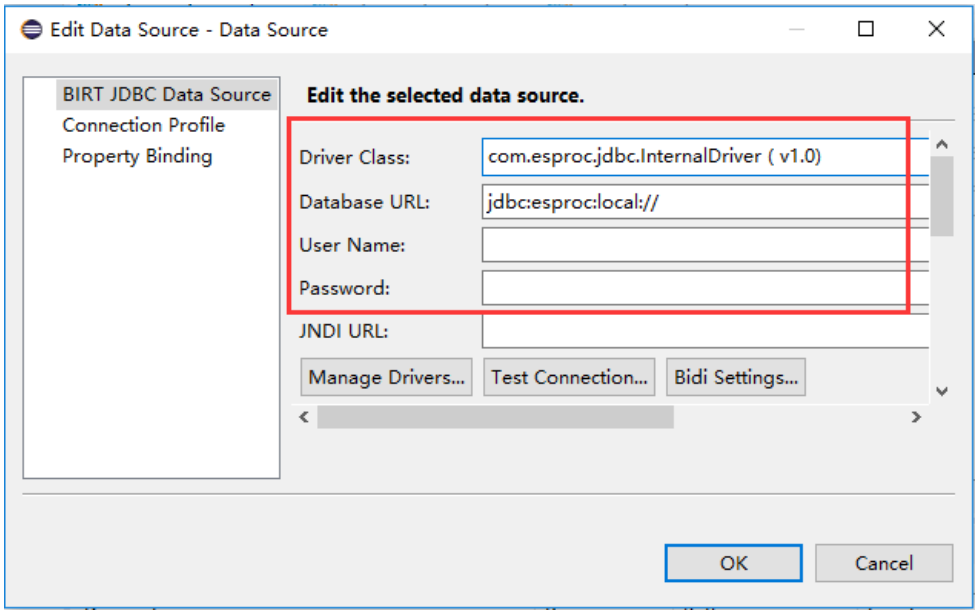
```
...
    <Esproc>
        <license>esproc.xml</license>
        <!-- 商业授权或试用授权，试用授权从官网直接下载-->
        <!-- 授权文件路径，可以是绝对路径，也可以是相对路径，相对路径是相对于类路径-->

    </Esproc>
...
```

新增数据源

新建报表，在“DataSources”下新建 JDBC Data Source 类型数据源，并定义数据源名称为：esprocJDBC。



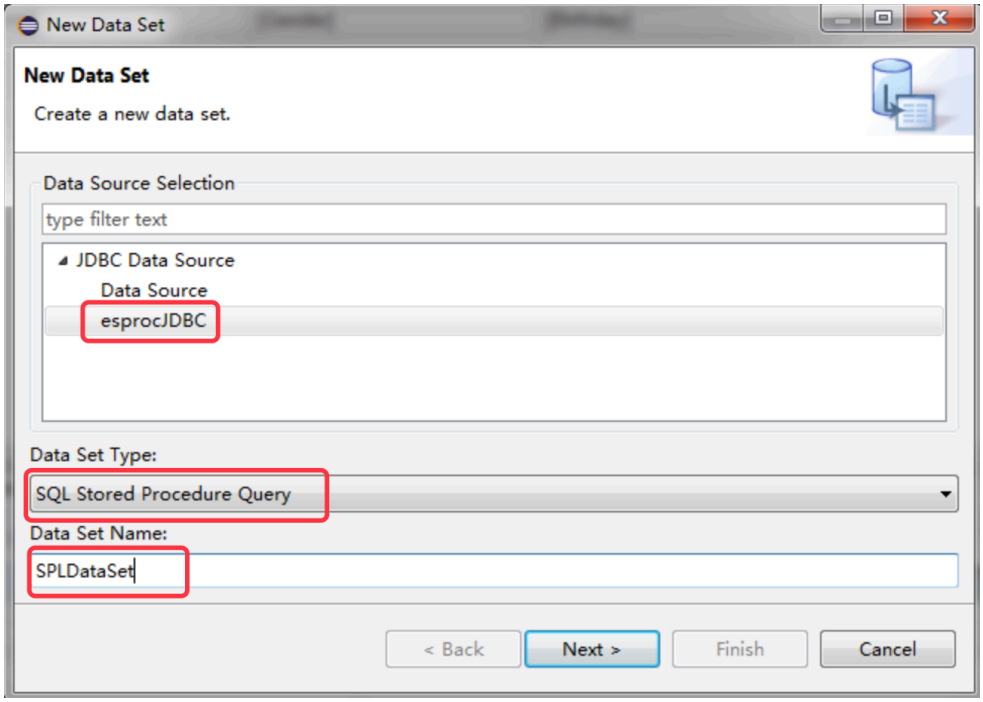
点击【Next】，在 Dirver Class 中选择驱动类名 com.esproc.jdbc.InternalDriver (v1.0)，填写 Database URL 为：jdbc:esproc:local://，用户名和密码为空。



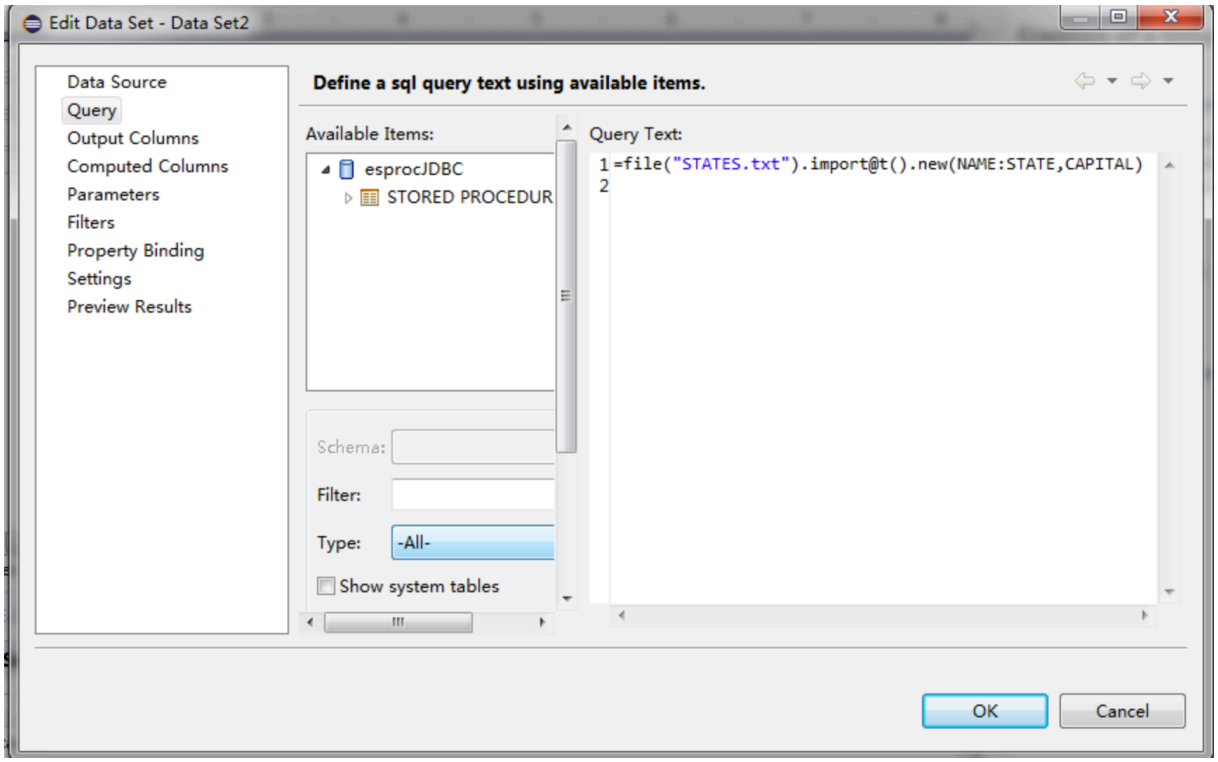
点击【Test Connection】测试连接，出现提示信息：” Connection successful. ”则 esprocJDBC 表示连接成功，点击【OK】，数据源创建完成。

新增数据集

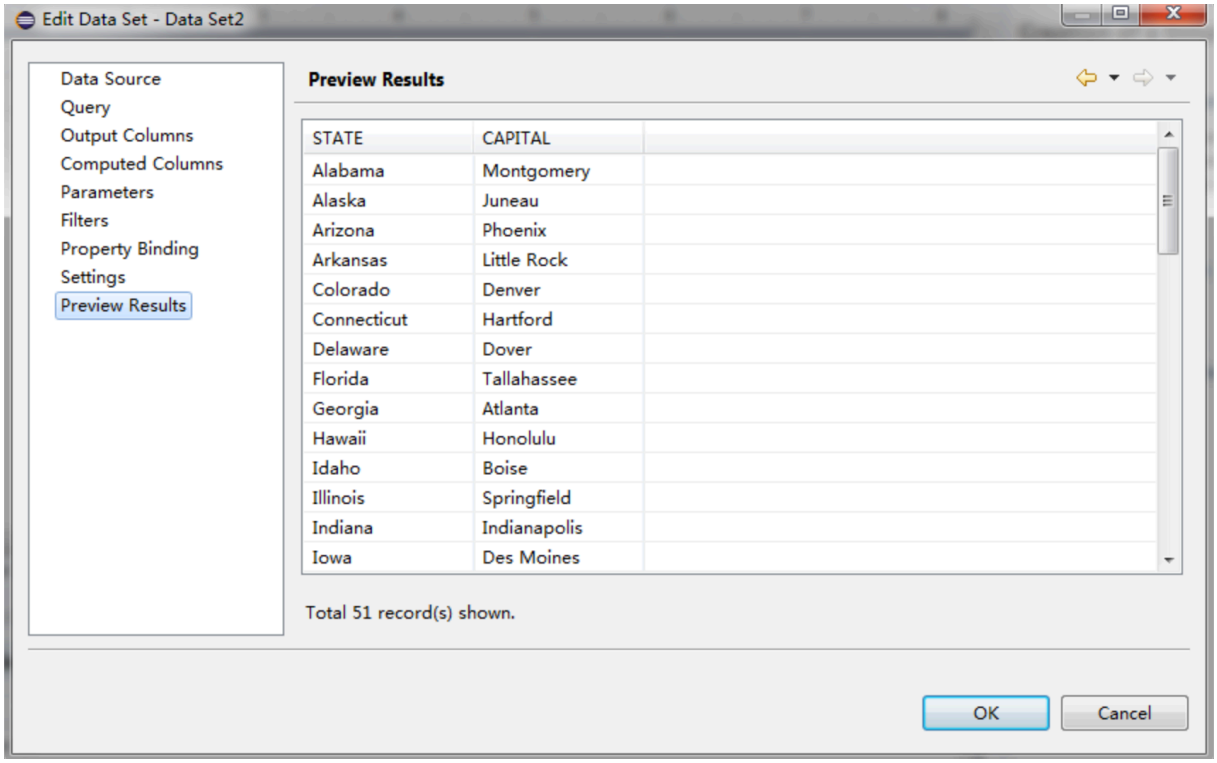
新建“Data Sets”，选择配置的集算器数据源（esprocJDBC），数据集类型选择存储过程（SQL Stored Procedure Query）,数据集名称定义为：SPLDataSet。



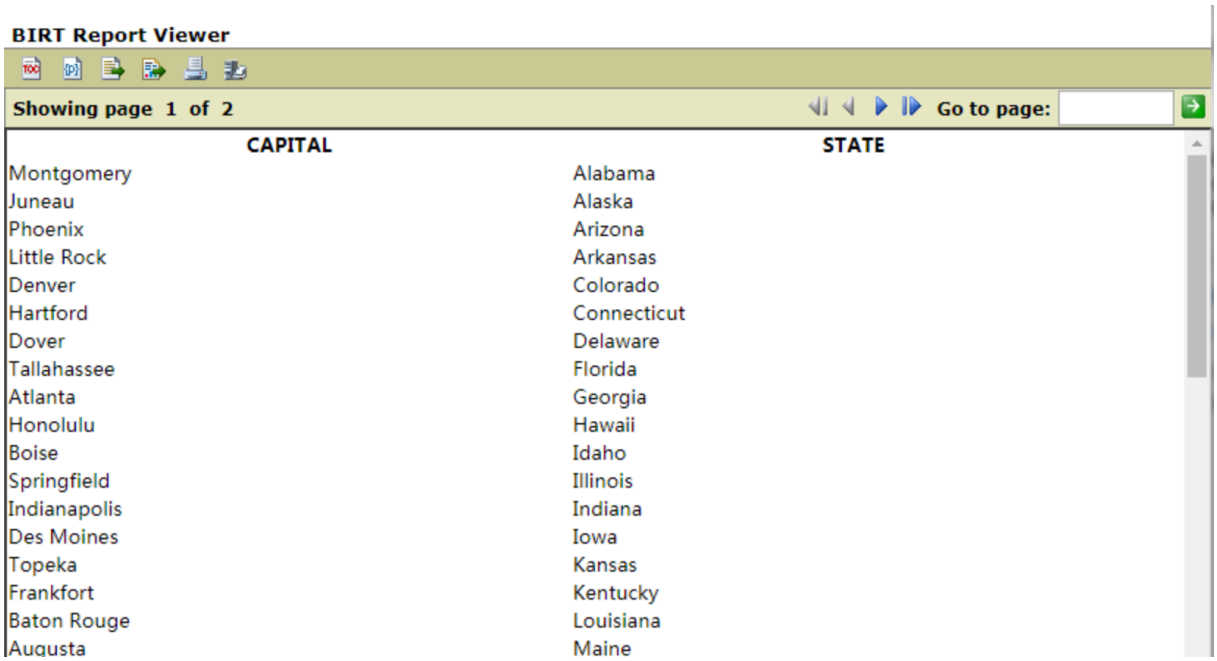
点击【Next】，填写查询语句，将例①中 B1 和 B2 的 SPL 语句合并成一句作为查询语句。



点击左侧的”Preview Results”可预览数据集：



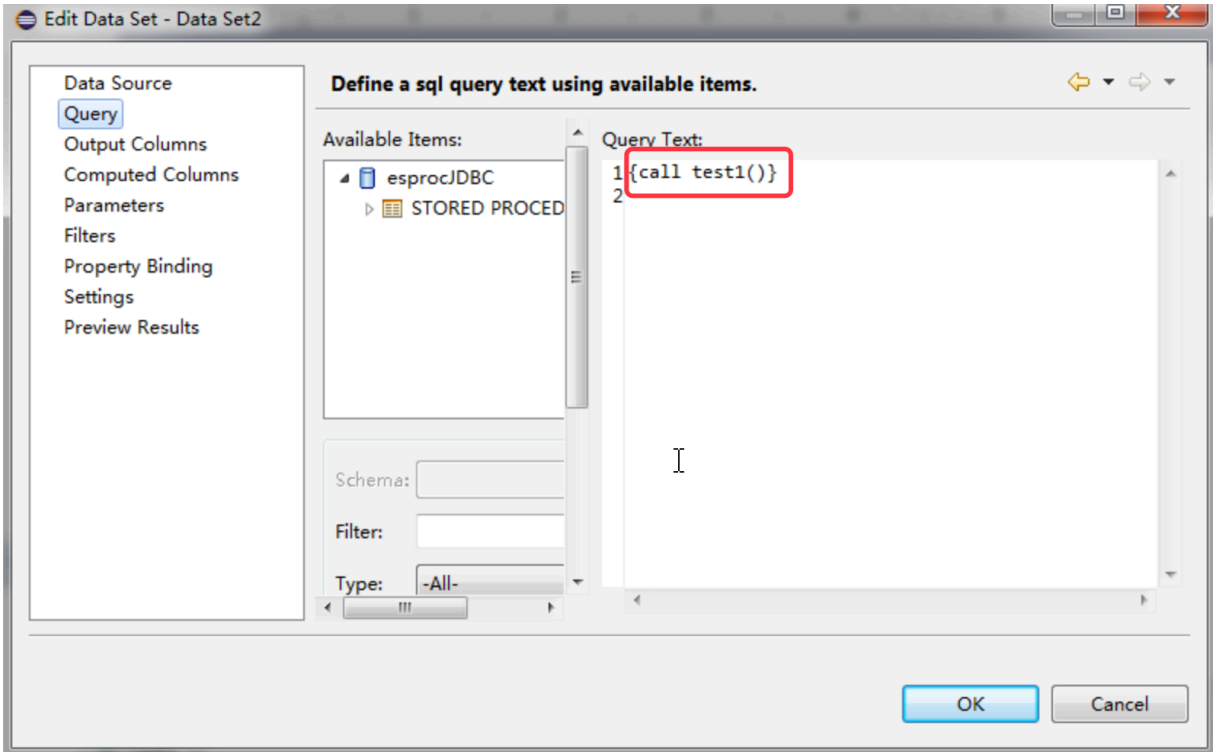
在报表中引用数据集，查看报表：



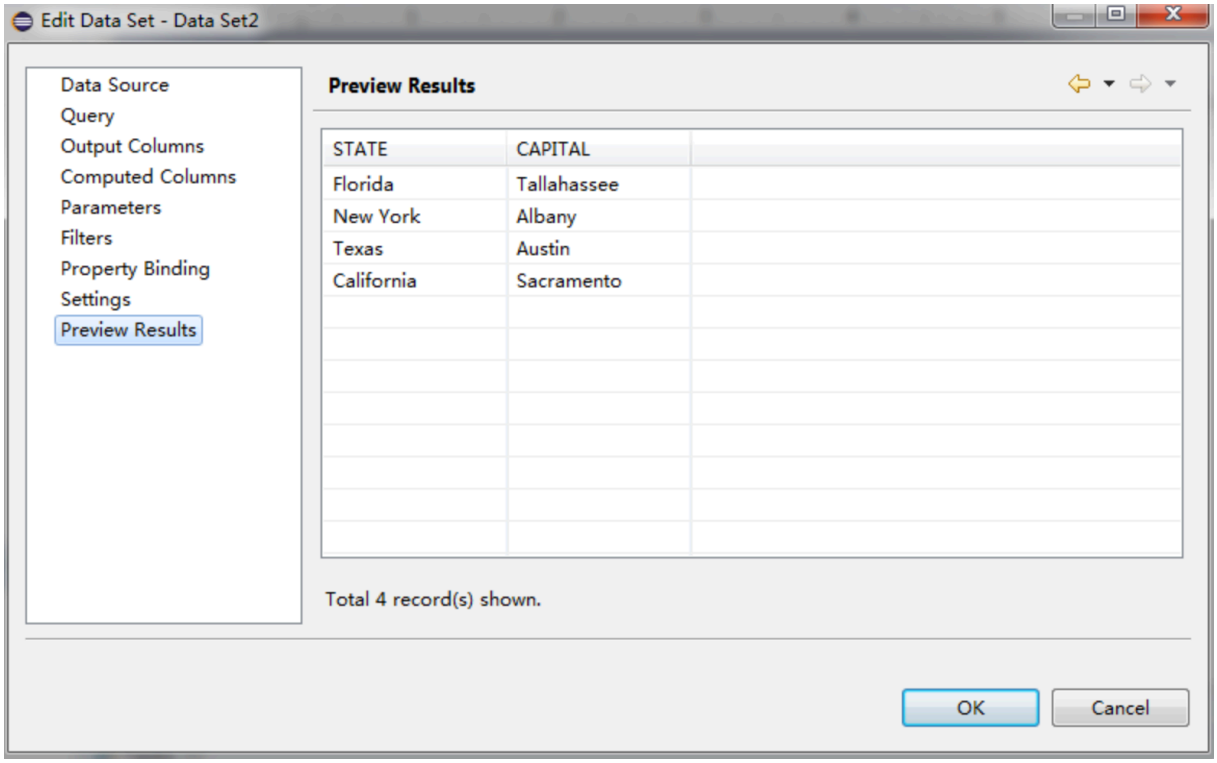
目前，在 BIRT 查询文本框能只能调用一句 SPL，如果想执行多条 SPL，只需用集算器 IDE 将包含多条 SPL 保存，经 IDE 保存的文件扩展名是 dfx。

	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)

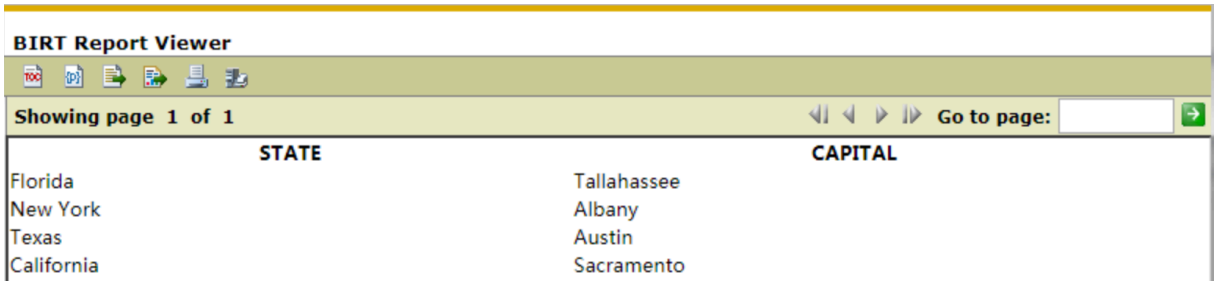
如使用例②，将从数据库取数语句注释掉，只执行从文本取数，然后筛选记录，保存文件为 test1.dfx，复制到部署文件 raqsoftConfig.xml 中<mainPath>C:\work\test_esproc</mainPath> 指定的目录下，存储过程里调用脚本文件名 test1 即可。



点击左侧的”Preview Results”可预览数据集，已是筛选后的结果：



在报表中引用数据集，查看报表：



与 ETL 工具集成

集算器还可以集成到 ETL 产品中，充当其内部算子，如在 Kettle 中调用集算器脚本，可以替代 js 脚本、java 脚本，减少流复制和流连接，提高程序运行效率，改善由于出错处理能力不足带来的程序运行风险。集算器精简的语法，网格式分步处理，大大提高了用程序处理数据过程的洞悉，并降低对开发人员的技术要求，Kettel 提供了直观的流程管理，都以 java 为基础，具有天然的互补性。

加载 JDBC 驱动

与报表工具集成类似，将三个集算器基础 jar 包 dm.jar，icu4j_3_4_5.jar，dom.jar（集算器[安装目录]\esProc\lib 目录下）拷贝至 kettle [安装目录]\data-integration\lib 目录下。

加载部署文件

将集算器配置文件 raqsoftConfig.xml（[安装目录]\esProc\config 目录下），拷贝至 Kettle[安装目录]\class 下，修改 raqsoftConfig.xml 文件，配置合法授权<license>...</license>和有效 dfx 文件路径<dfxPath>...</dfxPath>

新增数据源

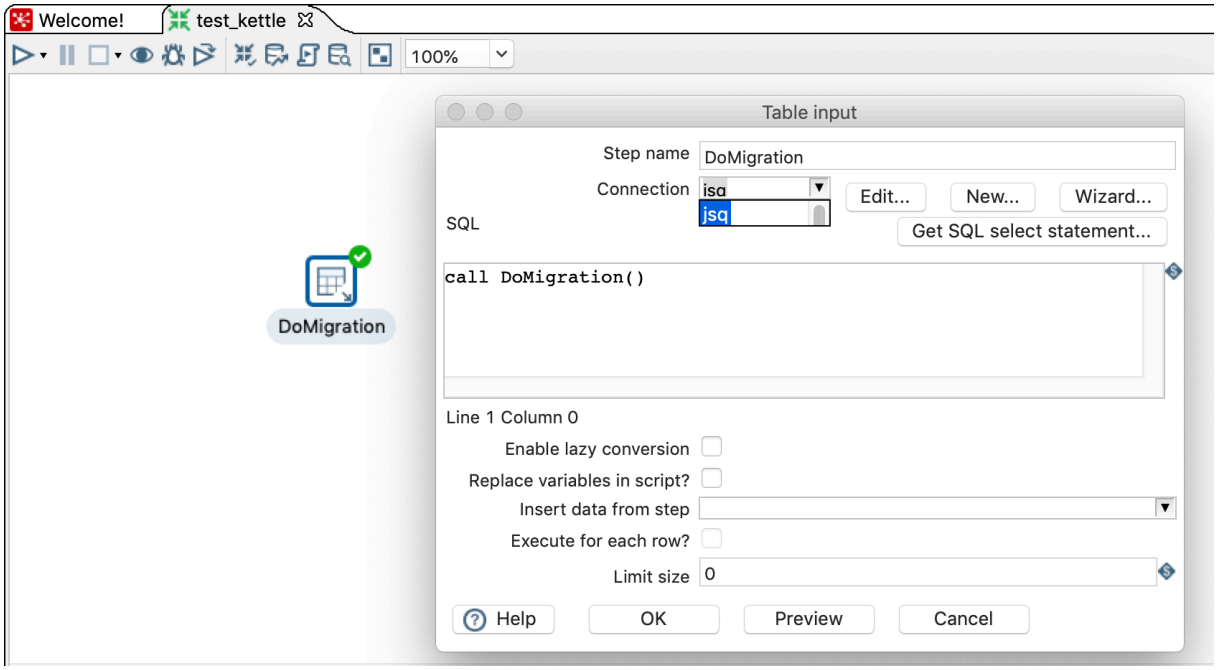
修改 Kettle[安装目录]\simple-jndi\jdbc.properties 文件，增加 jsq 数据源，以便在 kettle 中通过 jndi 访问集算器。

jdbc.properties

```
Hibernate/type=javax.sql.DataSource
Hibernate/driver=org.hsqldb.jdbcDriver
Hibernate/url=jdbc:hsqldb:hsql://localhost/hibernate
Hibernate/user=hibuser
Hibernate/password=password
Shark/type=javax.sql.DataSource
Shark/driver=org.hsqldb.jdbcDriver
Shark/url=jdbc:hsqldb:hsql://localhost/shark
Shark/user=sa
Shark/password=
PDI_Operations_Mart/type=javax.sql.DataSource
PDI_Operations_Mart/driver=org.postgresql.Driver
PDI_Operations_Mart/url=jdbc:postgresql://localhost:5432/hibernate?searchpath=pentaho_operations_mar
PDI_Operations_Mart/user=hibuser
PDI_Operations_Mart/password=password
live_logging_info/type=javax.sql.DataSource
live_logging_info/driver=org.postgresql.Driver
live_logging_info/url=jdbc:postgresql://localhost:5432/hibernate?searchpath=pentaho_dilogs
live_logging_info/user=hibuser
live_logging_info/password=password
jsq/type=javax.sql.DataSource
jsq/driver=com.esproc.jdbc.InternalDriver
jsq/url=jdbc:esproc:local://
jsq/user=
jsq/password=
```

新增表输入

将之前的例子按列提取 保存，文件命名 DoMigration.dfx,放置在<dfxPath>能找到路径下。新建 Transformation,在 Database connections 下用 JNDI 方式新建连接 jsq。新建表输入，在 SQL 中采用调用存储过程的方式，调用 DoMigration.dfx，操作和运行结果如下：



Execution Results

Logging | Execution History | Step Metrics | Performance Graph | Metrics | Preview data

First rows | Last rows | Off

#	Country	Sex	Born	Y1995	Y2000	Y2003	Y2004
1	Australia	Men	Native	8.4	6.5999999999999996...	6.0	5.5999999999999996...
2	Austria	Men	Native	3.6	4.2999999999999998...	4.4000000000000003...	4.2999999999999998...
3	Belgium	Men	Native	6.3	4.2000000000000001...	6.0	5.5999999999999996...
4	Canada	Men	Native	8.6	5.7000000000000001...	6.5	<null>
5	Czech Republic	Men	Native	<null>	<null>	5.7999999999999998...	7.0
6	Denmark	Men	Native	6.4	3.3999999999999999...	3.7999999999999998...	4.5999999999999996...
7	Finland	Men	Native	17.7	10.300000000000000...	10.900000000000000...	9.9000000000000003...
8	France	Men	Native	9.1	7.7000000000000001...	7.2999999999999998...	8.0
9	Germany	Men	Native	<null>	6.9000000000000003...	9.3000000000000007...	10.300000000000000...
10	Greece	Men	Native	6.1	7.4000000000000003...	5.7999999999999998...	6.5
11	Hungary	Men	Native	<null>	7.2999999999999998...	6.2000000000000001...	5.9000000000000003...
12	Ireland	Men	Native	12.0	4.4000000000000003...	4.7999999999999998...	4.9000000000000003...
13	Italy	Men	Native	9.3	8.4000000000000003...	7.0	6.4000000000000003...

6 对比其他工具

其他工具		集算器
Excel	数据表格呈现，格子是数据存储单元，数据只有一份，格值改变后原始数据消失	代码表格呈现，格子是语句执行单元，格值是每步执行结果，处理动作和结果自然统一
	主要用于数据的填写和一次性计算	主要用于数据的整理和多次计算
SQL	需要花精力先建表结构；字段只能是常规类型、英语式语法，计算逻辑表达受限，经常是问题不难，翻译解法困难；命令执行后，数据才可见	无需建表；字段支持高级类型（集合、指针）、函数式语法，计算逻辑描述容易，贴近自然思维；保留中间结果集，随时可见、反复使用。
	主要用于数据库命令式操作	主要用于分步数据处理和简化复杂计算

Python	依赖包众多；高级编程语言，应用广泛；数据类型多样，语法丰富、灵活；执行速度慢，大数据处理还需依赖外部计算引擎	即装即用；DSL 语言，领域专一；数据类型简单，语法简练、直观；Java 运行环境，执行速度快，自带大数据处理能力
	擅长数学统计类分析和应用	擅长常规型业务计算，应用集成容易

和 SQL 对比

例 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

例 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.

例 3：

To get the monthly contract growth rate of each salesman

```
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.

和 Python 对比

例 1：

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

```
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.

例 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.

例 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" .