离散数据集模型

摘要

关系代数是当前最常用的结构化数据计算的代数体系，但已经历五十多年并没有关键性的改进。基于关系代数理论实现的 SQL 在用于数据计算时，无论是编码复杂度还是运行效率都难如人意。本文中将提出新的代数体系，即离散数据集模型，以解决关系代数的问题。离散数据集理论在保持关系代数集合化特性的基础上，引入了离散性，从而可以方便地解决有序计算并实现更彻底的集合化，极大地降低了编码复杂度，同时也为高效的工程实现打下了理论基础。

关键字

离散性、集合化、序列、结构、记录、字段、序表、游标、实表、组表

0 准备

1. 数据

数据是一个不给定义的基本概念，它能够通俗地解释为信息系统中处理的信息基本单元，其实可以是任何可被计算机表示的信息。

每个数据均有**数据类型**，不同类型的数据在处理过程中允许进行的运算和遵循的规则不同，不同类型可以转换。常见的类型有：

1. **数值**：可分为**整数**和**实数**，能够进行各种数学运算；
2. **字符**：可以进行比较运算；多个字符可以连结成**字串**；
3. **逻辑**：真假值，可进行与、或、非等运算；

事实上，由于数据的多样性，数据类型也可能有无穷多种。

定义三个常用常数：

null 空值或逻辑假值，我们约定$\forall x\neq$null 有 null<x

inf 表示数值的无穷大，-inf 表示负无穷大

true 逻辑真值，我们约定任何非空值也可表示逻辑真值

我们约定，对任何数值 x，有 x+null==x-null==x。

注：本文中将延用 C 语言习惯，用两个==表示相等比较，单个=用于表示赋值。

数据被命名后称为**变量**，变量的值在计算过程中会变化。

2. 集合

集合的研究不是本文目标，这里仅对用到的术语和符号作一些回顾和约定。

一些确定的数据构成**集合**，构成集合的数据称为集合的**元素**。我们在{}中写上元素表示集合，如{1,2,3,…}是所有自然数的集合。

{x|$P(x)$}表示一切具有性质 P 的数据构成的集合。

|X|表示集合 X 的元素个数（|x|同时也是绝对值运算符），元素个数为有限整数的集合称为**有限集**。本文涉及的绝大多数集合都是有限集。没有元素的集合称为**空集**，即{}，显然|{}|=0。

集合的交、并、差以及属于、包含等概念及符号将继续延用数学上的约定，其中用\表示差集运算符。

自然数集用字母 **N** 表示。用 N{n}表示有限集{1,…,n}，即前 n 个自然数。

有限集合的元素次序被关注时被称为**有序**集合，为与无序集合区分，我们在[]而不是{}中依次写上元素以表示有序集，如 [a_1,…,a_n]。

有序集相等判断的规则定义为：

[a_1,…,a_n]==[b_1,...,b_n] ⇔ a_1==b_1,…,a_n==b_n

类似地，用 N[n]表示有序集[1,…,n]。

3. 函数

某个数据 x 能够经过一系列确定的运算 f 后得到另一个数据 y，这个运算系列 f 被称为**函数**，记作 y=f(x)，x 称为 f 的**参数**，y 为**返回值**。从参数获得返回值的过程称为**计算**函数。

函数 f 的参数 x 是有序集合 [x_1,...,x_n]时，我们经常将把 f(x)写成 f(x_1,...,x_n)的形式，这样

可书写更为简洁。我们把 $x_1,...,x_n$ 都称为 f 的参数，f 称为**多元函数**，x_i 是 f 的第 i 个参数。

多元函数 $f(x_1,\ldots,x_n)$的某些参数在书写时可以省略，只要 f 的运算过程能够处理这种情况即可。特别地，设非负整数 $k<n$，若函数 $f(x_1,...,x_k)$和 $f(x_1,...,x_n)$都有定义时，我们将把 $f(x_1,...,x_k)$理解为 $f(x_1,...,x_n)$中后 $n-k$ 个参数省略的特殊情况。

多元函数的某个参数可能还是有序集，这时我们约定使用冒号来分隔这个序列的成员，即把 $f([x_1,...,x_n],[y_1,...,y_m],\ldots)$写成 $f(x_1:...:x_n,y_1:...:y_m,...)$的形式，以避免书写太多的方括号。

在冒号书写形式中，x_i 或 y_i 仍然可能还是有序集，我们再约定用分号作为第一层参数的分隔符，逗号用作第二层，冒号用作第三层。即把 $f([x_1,...x_n]:y...,z,...)$写成 $f(x_1:...:x_n,y,...;z;...)$。

更深的层次不常见，就不再简写了。

某一组函数的参数以及运算过程很相似时，我们将这一组函数书写成 $f@o(...)$的形式，整组函数被命名为 f，o 称为 f 的**选项**。可以将函数组 f 理解成由选项 o 决定运算过程的同一个函数，选项 o 也可以理解为 f 的某个特殊参数。

我们约定选项由若干个字符构成，这些字符没有次序，即有 $f@o_1o_2==f@o_2o_1$。

某些函数的第 1 个参数总是同一种类型的数据，我们也常常用对象式的写法，即把 $f(A,\ldots)$写成 $A.f(\ldots)$的形式，此时 A 称为**对象**，f 称**针对** A 的函数。在这种书写形式下，我们将认为对于相同的 f，$f(\ldots)$和 $A.f(\ldots)$也是不同的函数。

这里没有采用数学上常用的集合映射的方式来定义函数，而是使用了程序语言的函数概念，这样会更方便地解释函数的运算过程，特别地，我们还会强调函数在计算过程中会完成某些动作，而不仅仅是为了返回某个值。

离散数据集体系中的基本运算大都会以函数的形式定义出来，而不象传统数学运算使用符号，这是因为运算种类太多且参数较为复杂。

将代表数据、函数等基本项的符号用一些运算操作符号按一定规则书写出来的字串，称为**表达式**。当表达式中符号或标识符代表的数据都已确定时，这个表达式可以被计算得到一个数据，称为表达式的**计算结果**。

我们这里不对表达式的概念及书写规则作详细讨论，而是简单延用计算机语言化的数学表达式。如 3+5*a、2*ln(x)等，与数学表达式不同，这里乘号用*表示且不能省略，函数参数一定要用括号括起来，运算符（如逻辑运算与,或,非等）则采用 C 语言的符号体系（&&,||,!等）。

一个函数的运算过程可以用含有其参数的表达式来定义，也就说可以用字串来表示一个函数。我们可以把这种用表达式描述的函数也看成是一种数据类型，而用于另一个函数的参数，这类参数被称为**函数参数**，函数参数在我们后面的讨论中非常常用。

一些常用函数及语法:

if(x,y,z)	如果 x 真（即非空）则返回 y 否则返回 z
if(x_1:y_1,...,x_k:y_k;y)	if(x_1,y_1,if(x_2,y_2,...,if(x_k,y_k,y)...))
cmp(x,y)	if(x==y,0,if(x>y,1,-1))
max(x,y)	if($x\geq y$,x,y)
min(x,y)	if($x\leq y$,x,y)

$(x_1,\ldots,x_n)$	依次计算 $x_1,\ldots,x_n$ 后返回 x_n，无歧义时外层括号可不写

4. 代码

如上所述，我们把函数理解成某种运算过程。有些复杂的过程很难用表达式书写，我们将使用程序语言的伪代码来定义。

本文中使用的伪代码在 C 语言语法风格基础上进一步简化，具体有如下不同点：

1) 所有局部变量会用斜体，保留字和特殊变量会用正体；
2) 不声明变量的数据类型，直接使用；
3) 无条件循环写成 for ()；
4) 不使用{}来表示循环体或分支内容，而使用缩进，即向右缩进的内容即是循环体和分支执行内容，代码缩进向左回退时表示循环体或分支内容结束；

1　序列

1.1　序列

1. 序列

长度为 n 的有序集合称为 ***n* 序列**，n 称为序列的**长度**，序列 A 的长度用符号$|A|$表示，其第 i 个成员可以用 $A(i)$表示，称为 A 的第 i 个**成员**，i 称为 $A(i)$的**序号**。当 $i<1$ 或 $i>|A|$时，我们约定 $A(i)$是 null，同时约定 A(null)也是 null，以及|null|=0。

字串可以看成是成员为字符的序列，为书写方便，我们把字串$[a_1,...,a_n]$写成 $a_1a_2...a_n$（其中 a_i 是字符），有时为与其它类型数据区分还可能加上引号，即写成"$a_1a_2...a_n$"。

满足 $i\leq j\Rightarrow A(i)\leq A(j)$或 $i\leq j\Rightarrow A(i)\geq A(j)$的序列 A 称为**有序列**。

满足 $i<j\Rightarrow A(i)\leq A(j)$的序列 A 称为**递增列**，由集合 S 元素构成的递增列记作$[S]$。

由自然数构成的序列称为**数列**，数列包含于 N$\{n\}$时称为**{*n*}数列**。

成员 m 在序列 A 中的**位置**用 $A\#m$ 表示，定义为集合$\{i\in N|A(i)==m\}$中的最小值，若该集合为空集，则定义 $A\#m$ 为 0。

满足 $A\#A(i)=i$ 的序列称为**单列**，$n\{n\}$单数列称为 n **置换**，可以证明 n 置换的成员将构成 N$\{n\}$。

A 是 n 序列，p 是 $m\{n\}$数列，则 m 序列$[A(p(1)),\ldots,A(p(m))]$称为 A 的**产生列**，记作 $A(p)$。

序列 $A=[a_1,...,a_n]$，序列 $B=[a_1,...,a_n,b_1,...,b_m]$称为 A 的 ***m* 延长列**。

若存在置换 p 使 $B==A(p)$，则称 B 是 A 的**置换列**，容易证明 A 也是 B 的置换列。

几个常用函数：

to(a,b)	$a\leq b$ 时，$[a,a+1,...b]$；$a\geq b$ 时，$[a,a-1,...,b]$
to(n)	to(1,n)，即 N$[n]$
A.**to**(a,b)	A(to(a,b))
A.**to**(k)	A(to(k))
A.**to**(k,)	A(to(k,$\|A\|$))
A.**m**(i)	if(i>0, $A(i)$, $A(\|A\|+1+i)$)

2. 集合运算

设有 n 序列 A 和 m 序列 B：

序列$[A(1),...,A(n),B(1),...,B(m)]$称为 A、B 的**和列**，记作 $A|B$，显然$|A|B|=|A|+|B|$。

记 $p=[\{i\in N|B\#A(i)==0\}]$，$A(p)$称为 A、B 的**差列**，记作 $A\backslash B$。

$A\backslash(A\backslash B)$称为 A、B 的**交列**，记作 $A\hat{}B$，注意 $A\hat{}B$ 不一定和 $B\hat{}A$ 相同。

$A|(B\backslash A)$称为 A、B 的**并列**，记作 $A\&B$，同样地，$A\&B$ 不一定和 $B\&A$ 相同。

x 不是序列时，我们约定 $A|x$,$x|A$,$A\backslash x$,$A\&x$,$x\&A$ 分别相当于 $A|[x]$,$[x]|A$,$A\backslash[x]$,$A\&[x]$,$[x]\&A$。但是，我们约定 null$|A==A|$null$==A\backslash$null$==A\&$null$==$null$\&A==A\hat{}$null$==$null$\hat{}A==A$。

k 是正整数时，$A|...|A$（k 个）记作 $A*k$ 或 $k*A$。

字串 $a=a_1a_2...a_n$ 和 $b=b_1b_2...b_m$，则字串 $a_1...a_nb_1...b_m$ 为 a 与 b 的**连结**，记作 $a+b$。

递归定义**序列比较函数 cmp**(A,B)：

当 $n==m==1$ 时，返回 cmp(A(1),B(1))

当 $n==m>1$ 时，记 c=cmp(A(1),B(1))，返回 if(c==0,cmp(A.to(2,),B.to(2,)),c)

当 $n>m$ 时，记 c=cmp(A.to(m),B)，返回 if(c==0,1,c)；

当 $n<m$ 时，记 c=cmp(A,B.to(n))，返回 if(c==0,-1,c)；

可以证明

$A==B \Leftrightarrow$ cmp(A,B)==0

继而可以定义

$A==B$	cmp(A,B)==0
$A\neq B$	cmp(A,B)≠0
$A>B$	cmp(A,B)>0
$A\geq B$	cmp(A,B)≥0
$A<B$	cmp(A,B)<0
$A\leq B$	cmp(A,B)≤0

可以证明，这些比较规则满足传递性，即 $A\leq B$ && $B\leq C \Rightarrow A\leq C$。

对 n 序列 A 和 m 序列 B，定义 **cmp**(A;B)

$n\geq m>0$ 时	cmp(A.to(m),B)
其它情况	cmp(A,B)

特别地，cmp(A;B)当 A,B 是单值时被解释为 1 序列。

3. 修改

设有 n 序列 A 和正整数 $k\leq n$ 及数据 x，p 是 m{n}数列，X 是 m 序列，定义函数

A.**insert**(k,x)	A.to(k-1)|x|A.to(k,)，特别地，k==0 时返回 A|x
A.**modify**(k,x)	A.to(k-1)|x|A.to(k+1,)
A.**modify**(p,X)	A.modify(p(1),X(1)),…,A.modify(p(m),X(m))
A.**delete**(k)	A(to(n)-k)
A.**delete**(p)	A(to(n)-p)，

我们约定，这些函数的返回值将仍用变量 A 命名，称为**赋值**函数，相当于对 A 做变换。

A.modify(k,x)和 A.modify(p,X)可分别书写成 A(k):=x 和 A(p):=X 的形式。

1.2 基础运算

1. 迭代函数

我们定义序列的三个基础函数。

对于 n 序列 A 和表达式 x 及数据 a，递归定义**迭代函数** A.**iterate**(x,a)：

n==0 时，返回 a;

$n>1$ 时，令~~=A.to(n-1).iterate(x,a), #=n,~=A(n)，计算 x 并将结果返回；

作为函数参数的 x 是一个表达式，我们约定在其中用符号#,~,~~表示 x 的参数。

运算过程用伪代码写出来如下：

```
~~=a;
for (i=0;i<n;i++)
    #=i+1,~=A(i),~~=x;
return ~~;
```

对于 *n* 序列 *A* 和表达式 *x*，定义**产生函数**

A.**new**(*x*)　　*A*.iterate(~~|*x*,null)

这里的 *x* 中一般不再有符号~~。

容易证明 *A*.new(~)==*A*，*A*.new(#)==to(|*A*|)。

A.new(*x*)很常用，我们经常会将它简写为 *A*.(*x*)，同时，我们约定 *A*.()==*A*.(~)==*A*。

对于某个数据 *a*，我们用 *a*.(*x*)表示[*a*].(*x*)(1)，在 *x* 中可以使用~表示 *a* 本身，类似地 *a*.()==*a*.(~)==*a*。

在没有歧义时，我们还可能将 *a*.(*x*)简写为 *a.x*，并约定 *A.x* 被理解为 *A*(1).*x*。

对于 *n* 序列 *A* 和表达式 *x*，定义**执行函数**

A.**run**(*x*)　　*A*.iterate((*x*,*A*),null)

A.run()函数仅仅是计算 *x*，仍然返回 *A* 本身。

类似地，也可以定义 *a*.run(*x*)，在计算 *x* 后仍返回 *a*。

2. 循环函数

我们将用上面的三个基础函数来定义其它运算，首先基于迭代函数可以定义一些序列相关的**聚合函数**：

A.**sum**()　　*A*.iterate(~~+~,null)

A.**count**()　　*A*.iterate(~~+if(~,1,0),null)

A.**avg**()　　*A*.sum()/*A*.count()

A.**max**()　　*A*.iterate(max(~~,~),null)

A.**min**()　　*A*.iterate(min(~~,~),inf)

当 *A* 是序列构成的序列时，还可以定义一些集合类的聚合函数：

A.**conj**()　　*A*.iterate(~~|~,null)

A.**union**()　　*A*.iterate(~~&~,null)

A.**isect**()　　*A*.iterate(~~^~,null)

针对某个序列并基于迭代函数定义的函数可以通俗地称为**循环函数**，上面定义的聚合函数都是循环函数。循环函数不是一个严格定义，它是一大类函数的统称，我们不讨论循环函数的整体性质，而只是关注语法规则。

书写上无歧义时，我们约定针对序列 *A* 的循环函数 *A.f*(*x*)=*A*.(*x*).*f*()；另外对于自然数 *n*，再约定 *n.f*(*x*)=to(*n*).*f*(*x*)。这里 *x* 作为函数参数的表达式，可以在其中引用~和#。

综合举例：5.sum(~*~)将计算 1,2,3,4,5 的平方和。

循环函数可能嵌套使用，即 *A.f*(*x*)中的 *x* 可能会引用另一个循环函数 *B.g*(*y*)，我们约定在 *y* 中的~和#将被解释成内层序列的，也就是 *B* 的，而外层序列的~和#将再冠以序列的变量名写作 *A*.~和 *A*.#以示区分。

如 *A*.sum(*B*.sum(*A*.~*~)) 将计算 *A*,*B* 两个序列的成员的两两乘积之和，而 *A*.sum(*B*.sum(~*~))的计算结果则是 *B* 的成员的平方和的|*A*|倍。

有时用变量名也不能区分时，我们定义 get 函数表示嵌套循环函数中内层序列表达式中引用外层序列成员：

get(*l*,~)　在内层循环函数返回上外面第 *l* 层循环函数的~

get(*l*,#)　在内层循环函数返回上外面第 *l* 层循环函数的#

当前层的层号约定为 0。~相对常用，可以省略不写；~~很少用到，就不再定义了。

上面的 *A*.sum(*B*.sum(*A*.~*~))也可以写成 *A*.sum(*B*.sum(get(1)*get(0))。

在迭代/循环函数的函数参数 *x* 中，除了#,~,~~外，我们还可以设计符号引用相邻成员:

~[*i*]　　*A*(#+*i*)

~[*a*,*b*]　　*A*.to(#+*a*,#+*b*)，*a* 省略时为 *A*.to(#+*b*)，*b* 省略时为 *A*.to(#+*a*,)

这样，可以用 *A*.(~-~[-1]) 计算相邻差，用 *A*.(~[-1,1].avg())计算移动平均。

我们还可以扩展 get 函数，使之支持这种相邻引用

get(*l*,~:*i*)　　相应层次的~[*i*]

get(*l*,~:*a*:*b*)　　相应层次的~[*a*,*b*]

我们还可以定义**定位计算**函数在循环函数之外进行位置相关的计算：

A.**calc**(*k*,*x*)　　*A*(*k*).(*x*)，在 *x* 中可以使用#,~和[]等符号

A.**calc**(*p*,*x*)　　*p*.(*A*.calc(*p*.~,*x*)，*p* 是数列

SQL 没有次序和位置的概念，在集合计算中没有提供#和[]等符号，很难实现相关的运算。

1.3　导出运算

1. 定位选出

循环函数返回成员在序列中的序号或序号构成的数列时，被称为**定位函数**。

一些常用的定位函数：

A.**pselect**@**a**(*x*)　　*A*.iterate(if(*x*,~~|#,~~),null)

A.**pmax**@**a**(*x*)　　*A*.iterate(if(*x*==*A*(~~(1)),~~|#,if(*x*>*A*(~~(1)),[#],~~),null)

A.**pmin**@**a**(*x*)　　*A*.iterate(if(*x*==*A*(~~(1)),~~|#,if(*x*<*A*(~~(1)),[#],~~),null)

注意，我们在这里使用了函数选项@a。

这些定位函数返回的是一个数列，但更常见的情况是我们常常只对其中某个成员感兴趣，这样我们把不带选项的函数定义为返回结果集的第一个成员，即：

A.**pselect**(*x*)　　*A*.pselect@a(*x*)(1)

A.**pmax**(*x*)　　*A*.pmax@a(*x*)(1)

A.**pmin**(*x*)　　*A*.pmin@a(*x*)(1)

返回序列的某个产生列的函数称为**选出函数**，可以用定位函数来定义：

A.**select**(*x*)　　*A*(*A*.pselect@a(*x*))

@**1**　　*A*(*A*.pselect(*x*))

A.**maxp**(*x*)　　*A*(*A*.pmax(*x*))

@**a**	*A*(*A*.pmax@a(*x*))
A.**minp**(*x*)	*A*(*A*.pmin(*x*))
@**a**	*A*(*A*.pmin@a(*x*))

注意 *A*.max(*x*)==*A*.(*x*)(*A*.pmax(*x*))≠*A*(*A*.pmax(*x*))==*A*.maxp(*x*)，只有当 *x* 为~（或省略）时才有 *A*.max()==*A*.maxp()。*A*.max(*x*)是返回最大的 *x* 值，而 *A*.maxp(*x*)返回使 *x* 最大的 *A* 的成员。

min 的情况类似。

从定位和选出函数的设计可以看出离散数据集与关系代数的理念不同，离散数据集强调集合的有序性，成员在集合中的位置是非常重要的信息，因而专门提供了定位运算，用于实现过滤的选出运算则是用定位运算来定义的。使用选出运算返回原集成的成员，而不象 SQL 那样返回新的计算值，这是为了强调成员的**离散性**，也就是成员可以游离于所在集合之外再参与运算或组成新集合，这也是离散数据集的命名原因。

2. 排序

对于 *n* 序列 *A*，定义**排名函数**：

A.**rank**(*y*,*x*)	|*A*.iterate(if(*x*<*y*,~~|*x*,~~),null)|+1
@**i**	|*A*.iterate(if(*x*<*y*,~~&*x*,~~),null)|+1
A.**ranks**(*x*)	记 *X*=*A*.(*x*)，返回 *X*.(*A*.rank(*X*.~,*x*))
@**i**	*X*.(*A*.rank@i(*X*.~,*x*))

这里 *x* 是函数参数，而 *y* 不是。

对于 *n* 序列 *A*，递归定义**排序函数** *A*.**psort**()：

n==1 时，返回[1]；

n>1 时，记 *p*=*A*.to(*n*-1).psort()，*a*=*A*(*n*)，*k*=*p*.count(~≤*a*)+1，返回 *p*.insert(*k*,*n*)；

从定义上可以看出，*A*.psort()也可以理解为循环函数（可以用迭代函数定义，但计算式较复杂）。

继续定义：

A.**psort**(*x*)	*A*.(*x*).psort()
A.**sort**(*x*)	*A*(*A*.psort(*x*))

A.psort 也可以理解为定位函数，而 *A*.sort 则是选出函数。

可以证明，对于 *n* 置换 *p*，有 *p*.psort().psort()=*p*，即 psort 是自己的逆函数。

再定义 **TOP 函数**，*k* 是自然数：

A.**ptop**(*k*,*x*)	if(*k*>1,*A*.psort(*x*).to(*k*),*A*.pmin@a(*x*))
A.**top**(*k*,*x*)	*A*.(*x*)(*A*.ptop(*k*,*x*))
A.**top**(*k*;*x*)	*A*(*A*.ptop(*k*,*x*))

大多数情况下，*A*.top 函数的 *k* 参数取值都不大，虽然它和排序有关，返回值是个集合，但我们经常把它认定为聚合函数。

缺乏离散性的 SQL 无法做到彻底的集合化，TOP 运算只能是结果集输出时进行，而不能作为聚合函数。

对位排序可以用已有的函数定义出来：

A.**align**@**s**(*C*:*x*,*y*)　　*C*.conj(*A*.select(*C*.*x*=*A*.*y*))

将序列 *A* 按照序列 *C* 的次序排列。

在 SQL 中需要用连接方法来实现对位计算，但会丢失次序信息。

3. 分组

对于 *n* 序列 *A*，递归定义**等值分组函数** *A*.**group**@**p**()：

n==1 时，返回[[1]]；

n>1 时，记 *G*=*A*.to(*n*-1).group@p()，*a*=*A*(*n*)，*k*=*G*.pselect(*A*(~(1))=*a*)

返回 if(*k*>0,*G*.modify(*k*,*G*(*k*)|*n*),*G*|[[*n*]])

类似地可以分析得出，*A*.group@p 也可以视为循环函数。

继续定义：

A.**group**@**p**(*x*)　　*A*.(*x*).group@p()

A.**group**(*x*)　　*A*.group@p(*x*).(~.(*A*(~)))

@**1**　　*A*.group(*x*).(~(1))

分组的实质动作是将一个集合拆成一些小集合，所以返回值是一个序列的序列。SQL 的集合化不够彻底，不能保持分组的中间结果，必须强制聚合。

A.group@1()相当于 SQL 中的 distinct 运算，还可以定义**异值计数函数**：

A.**icount**(*x*)　　|*A*.group(*x*)|

A.icount()当然也能被认为是聚合函数。

对于 *n* 序列 *A*，递归定义**有序分组函数** *A*.**group**@**op**()：

n=1 时，返回[[1]]；

n>1 时，记 *G*=*A*.to(*n*-1).group@op()，*k*=|*G*|

返回 if(~=~[-1],*G*.modify(*k*,*G*(*k*)|*n*),*G*|[[*n*]])

类似地继续定义：

A.**group**@**op**(*x*)　　*A*.(*x*).group@op()

A.**group**@**o**(*x*)　　*A*.group@op(*x*).(~.(*A*(~)))

还可以定义按**条件有序分组**：

A.**group**@**i**(*x*)　　*A*.group@o(~[,0].count(*x*))

离散数据集体系中序列是有序的，可以方便地定义出按次序分组。SQL 基于无序集合，很难实现有序分组运算。

对位分组也可以用已有的函数定义出来。

A.**align**@**a**(*C*:*x*,*y*)　*C*.(*A*.select(*C*.*x*=*A*.*y*))

将序列 *A* 按照 *C* 分组。

与前面的具有完全划分性质的 *A*.group 函数不同，*A*.align@a 不是完全划分，可能有 *A* 的成员没有被分到任何一个组中。

2　序表

2.1　序表

1. 结构

由字串构成的单列称为**数据结构**，简称**结构**，结构的成员又称为**字段**。

对于 *k* 结构（长度为 *k*）*S*，设有 *k* 序列 *x*，2 序列 *r*=[*S*,*x*]称为以 *S* 为结构的**记录**，*S* 称为 *r* 的结构，记作 S[*r*]（即 *r*(1)），*S* 的字段也称为 *r* 的字段。

F 是记录 *r* 的字段，令 *i*=S[*r*]#*F*，我们将 *r*(2)(*i*)简记为 *r.F* 或 *r.#i*，称为 *r* 的**字段值**。用 *r.F*=*x* 表示对记录字段赋值，即相当于 *r*(2)(*i*)=*x*。

由记录构成的序列称为**排列**。成员的结构相同的排列 *T* 称为**序表**，此结构也称为 *T* 的**结构**，记作 S[*T*]，其字段也称为序表的**字段**。

定义结构相关的函数：

r.**fno**()	|S[*r*]|
r.**fno**(*F*)	S[*r*]#*F*
r.**fname**(*i*)	S[*r*](*i*)
T.**fno**()	|S[*T*]|
T.**fno**(*F*)	S[*T*]#*F*
T.**fname**(*i*)	S[*T*](*i*)

序表概念和关系代数的表是类似的，不过离散数据集很强调数据的有序性，因此序表被命名成这样，而且是以序列为基础定义的。

从定义中还可以看到，与关系代数不同的是，这里的记录并不必须依赖于序表而可以独立存在。不过，通常讨论中记录会属于某个序表，该序表就被通俗地称为记录所在的序表。在上下文无歧义时，我们用符号 **T**[*r*]表示记录 *r* 所在的序表，T[*r*]不是一个严格定义，记录可能没有所在序表，也可能有多个。

2. 语法

当 *A* 是排列时，循环函数 *A.f*(…,*x*,…)的函数参数 *x* 中还可以引用 *A* 中成员的字段。我们约定在 *x* 中直接用字段 *F* 表示~.*F*；类似地，*F*[*i*]表示~[*i*].*F*，*F*[*a*,*b*]表示~[*a*,*b*].(*F*)（注意这是一个序列）。

对于嵌套的情况也是就近引用原则，*A.f*(*B.g*(*x*))的 *x* 中，*F* 将优先被解释为 *B* 的字段，如果要强制引用 *A* 的字段，则会写成 *A.F*。类似地，嵌磁情况也可以使用 get 函数：

get(*l*,*F*)	向外 *l* 层的 *F*
get(*l*,*F*:*i*)	向外 *l* 层的 *F*[*i*]
get(*l*,*F*:*a*:*b*)	向外 *l* 层的 *F*[*a*,*b*]

对于字段引用和嵌套引用的风格，离散数据集和 SQL 是一致的。不过离散数据集的运算并不只针对有结构的记录，因此会多了~符号；而且，离散数据集比比关系代数更强调成员在集合（序列）中的位置（序号），所以提供了#和[]符号。

3. 主键

对于序表 T，如果有 P=S[T](p)=[P_1,..,P_k]是 S[T]的递增产生单列，且满足对于 T 的任意两个不同成员 $r_1 \neq r_2$ 都有[$r_1.P_1$,…,$r1.P_k$]≠[$r_2.P_1$,…,$r_2.P_k$]，则称 P 对 T 是**唯一**的。我们在所有对 T 唯一的 P 中选择其中某一个称为 T 的**主键**，用符号 **K**[T]表示。在有多个可选择的 P 时，选择哪一个并不影响后续的讨论，通常我们会选择长度较短的那个 P。主键的长度为 1 时则称为**单主键**，并经常直接用 P_1 而不再用序列形式的[P_1]表示，长度大于 1 时称为**多主键**。

确定了 T 的主键后，可以针对序表 T 中的记录 r 定义函数：

r.**key**(T)　　返回序列[$r.P_1$,…,$r.P_k$]

当上下文中 T 已经很明确时，r.key(T)可以简写成 r.key()。

T 是单主键时，不产生歧义时，我们也常常认为 r.key(T)的返回值是 $r.P_1$。

特别地，我们再约定 null.key(T)=null。

对于有主键的序表 T，我们定义两个函数：

T.**pfind**(k)　　T.pselect(k==~.key())

T.**find**(k)　　T(T.pfind(k)) 或 T.select@1(k==~.key())

pfind 和 find 没有函数参数，则不再适用上述嵌套语法规则，参数中引用的~,#以及字段将被认为更上一层的，可以先计算出来再进入 pfind 和 find 的计算。

另外，如果 T 是单主键时，为了简化书写，我们约定：当 k 不是序列时，T.pfind(k)等价于 T.pfind([k])、T.find(k)等价于 T.find([k])。

离散数据集中的主键和关系代数是类似的，相当于延用了这个概念。

对某些有多主键的序表 T，设 K[T]= [K_1,…,K_t]，我们有时会选定其中最后一个主键 K_t 为**时间键**。当表 T 的主键中有时间键时，我们重新定义这三个函数：

r.key(T)　　[$r.K_1$,…,r.K_{t-1}]

T.pfind(k)　　T.pselect@a(k'==~.key()).select(T(~),K_t<=k(t)).maxp(T(~),K_t)

T.find(k)　　T.select(k'==~.key()).select(K_t<=k(t)).maxp(K_t)

其中 k'=k.to(t-1)

时间键用于解决实际应用中常见的快照维表问题，关系代数中没有对应的概念。

4. 数据修改

序表 T 的数据修改函数定义如下：

T.**insert**(k,x_i:F_i,…)　　记 r=[S[T],T.fno()*[null]]，$r.F_i$=x_i，返回 T.insert(k,r)

T.**insert**(k:A,x_i:F_i,…)　　A.(T.insert(k+A.#-1,x_i:F_i,…)).m(-1)

T.**modify**(k,x_i:F_i,…)　　T(k).run(F_i:=x_i,…)，返回 T

T.**modify**(k:A,x_i:F_i,…)　　A.(T.modify(k+A.#-1,x_i:F_i,…)).m(-1)

T.**delete**(…)　　和序列相关函数的定义相同

和序列的类似，我们约定这些方法会修改变量 T 本身。

在表达式 x_i 中，我们约定其中出现的字段 F 优先引用 A 的，再引用 T 的（即认为 A 在内层，T 在外层）。

2.2 产生

1. 对位产生

针对序列 *A* 和序表 *T* 定义**对位产生函数**：

A.**new**(x_i:F_i,…)　　记 *S*=[F_i,…]，返回序表 *A*.([*S*,[x_i,…]])

T.**new@a**(x_i:F_i,…)　　记 *S*=S[*T*]|[F_i,…]，返回序表 *T*.([*S*,~(2)|[x_i,…]])

表达式 x_i 中的字段 *F* 将优先引用 *A* 和 *T* 的，再引用结果序表的。

T.new@a 又称为**导出函数**，它的返回值将拥有和 *T* 相同的主键（如果 *T* 有的话）

类似地，可以定义针对某个数据 *a* 的产生函数：

a.**new**(x_i:F_i,…)　　[*a*].new(x_i:F_i,…)(1)

和针对某个记录 *r* 的产生函数：

r.**new@a**(x_i:F_i,…)　　[*r*].new@a(x_i:F_i,…)(1)

以及针对某个自然数 *n* 的产生函数：

n.**new**(x_i:F_i,…)　　to(*n*).new(x_i:F_i,…)

直接生成记录的函数：

new(x_i:F_i,…)　　1.new(x_i:F_i,…)(1)

并约定 null.new(…)=null。

2. 分组聚合

前述的分组运算后将得到集合的集合，常常还要做进一步的聚合运算，而且每个子集可能聚合出多个值，这个就需要一个序表来承接返回值（每个聚合值是一个字段）。使用已定义过的 new 函数就实现这个动作。由于这种运算很常见，我们直接在 group 函数中增加定义：

A.**group**(x_i:F_i,…;y_i:G_i,…)　　*A*.group([x_i,…]).new(~.x_i:F_i,…,y_i:G_i,…)

在表达式 y_i 中可以引用~，如~.sum(*F*)表示求和。

特别地，当没有 y_i:G_i 参数部分时，函数将返回由不同的 x_i,…构成的序表，相当于 SQL 中的 DISTINCT 运算。

延伸后的 group 函数也可以支持@o 选项：

A.**group@o**(x_i:F_i,…;y_i:G_i,…)　　*A*.group@o([x_i,…]).new(~.x_i:F_i,…,y_i:G_i,…)

绝大多数聚合都是针对当前子集的，也都要书写~.，这看起来比较麻烦，因此我们定义一个专门用于聚合的分组函数：

A.**groups**(x_i:F_i,…;y_i:G_i,…)　　*A*.group([x_i,…]).new(~.x_i:F_i,…,~.y_i:G_i,…)

这里的 y_i 就只能针对当前分组子集聚合，原则上可以使用我们在前面定义过的任何聚合数，包括 sum/count/avg/min/max，也包括 top/icount。

groups 函数基于分组子集定义，但大多数能用迭代函数定义的聚合函数在计算时并不需要保留原序列，不需要先计算出完整的分组子集后再计算聚合值，这样可以获得更好的性能。

这种有聚合的分组运算将返回一个序表。延用上述符号，容易证明，*A*.group 和 *A*.groups 函数的返回序表可以使用 F_i,…作为主键（*A*.group@o 不可以），我们可以缺省地认为分组聚

合的返回序表都有这一套主键。

3. 扩展

分组聚合运算一般会将一个集合聚合成更小（成员数更少）的集合，我们再定义它的逆运算，即将一个小集合扩展成更大（成员数更多）的集合，称为**扩展**函数。

A.**news**(*v*;x_i:F_i,…)　　*A*.conj(*v*.new(x_i:F_i,…))

其中 *v* 是返回序列的函数参数，x_i 中的~优先解释为 *v* 的。

T.**news**@**a**((*v*;x_i:F_i,…)　*T*.conj(*v*.new(*T*.C_i:C_i,…,x_i:F_i,…))

其中 C_i,…是 *T* 的字段。

左连接选项@**1**，返回

T.conj(if(|*v*|>0,*v*.new(*T*.C_i:C_i,…,x_i:F_i,…)),new(*T*.C_i:C_i…,null:F_i,…))

需要注意的是，*T*.news@a 的返回值不一定还有主键。

2.3　连接

1. 外键连接

设序表 *D* 有单主键，序表 *T* 有字段 *K* 满足 *T*.(*K*)⊆*D*.(~.key())，则称 *T* 有指向**维表** *D* 的**外键** *K*。对于记录 *r* 和序表 *T*，定义外键解析函数：

r.**switch**(*K*,*D*)　　*r*.run(*K*=*D*.find(*K*))

T.**switch**(*K*,*D*)　　*T*.run(*K*=*D*.find(*K*))

外键解析之后，可以直接在 *T* 的表达式中引用 *D* 的字段 *F*，如 *T.K.F*。

使用

r.run(*K*=*K.P*)

T.run(*K*=*K.P*)

可以将解析过的外键还原。

switch 函数可以延伸成一次解析多个外键的运算：

r.**switch**(K_i,D_i;…)　　*r*.run(K_i=D_i.find(K_i),…)

T.**switch**(K_i,D_i;…)　　*T*.run(K_i=D_i.find(K_i),…)

序表 *D* 有多主键 *P*，序表 *T* 的字段 K_i,…满足 *T*.([K_i,…])⊆*D*.(~.key())，仍称 *T* 有指向维表 *D* 的外键 K_i,…。

多主键维表不能使用 switch 函数解析，需要**外键关联函数**：

r.**join**(K_i:…,*D*,*K*)　　*r*.new@a(*D*.find([K_i,…]):*K*)

T.**join**(K_i:…,*D*,*K*)　　*T*.new@a(*D*.find([K_i,…]):*K*)

外键关联之后，可将维表记录当成新加入字段 *K* 的值，在 *T* 相关的表达式中可以用维表字段 *F*，如 *T.K.F*。

类似地，join 函数也可以延伸成一次解析多个外键的运算：

r.**join**(K_i:…,*D*,*K*;…)　　*r*.new@a(*D*.find([K_i,…]):*K*;…)

T.**join**(K_i:…,*D*,*K*;…)　　*T*.new@a(*D*.find([K_i,…]):*K*;…)

特别地，当维表 *D* 有时间键且除时间键外只有一个主键时，可以修改 switch 函数的定

义：

r.**switch**(*K*:K_t,*D*)　　*r*.run(*K*=*D*.find(*K*,K_t))

T.**switch**(*K*:K_t,*D*)　　*T*.run(*K*=*D*.find(*K*,K_t))

涉及有时间键的维表时的 join 函数定义不用改变。

2. 主键连接

我们针对没有时间键的表 *T* 将 find 函数做延伸定义：

T.find@**s**(*k*)　　记 *n*= min(|*k*|,|K[*T*]|)，返回 *T*.select(~.key().to(*n*)==*k*.to(*n*))

与 find 不同，find@s 将返回一个排列。

然后再来定义常见的**等值连接函数 join**(T_1:F_1;T_2:F_2;…;T_n:F_n)：

n=1 时，返回 T_1.new(~:F_1)

n>1 时，记 *T*=join(T_1:F_1;T_2:F_2;…;T_{n-1}:F_{n-1})

返回 *T*.news@a(T_n.find@s(k(*T*.~);~:F_n)

其中 k(*x*)=*x*.(~.key()).maxp(|~|)是个中间函数

左连接选项@**1**：

返回 *T*.news@a1(T_n.find@s(k(*T*.~);~:F_n)

全连接选项@**f**：

在左连接的返回值 *T* 的基础上，记 *S*=F_n*T*.groups(F_n)

返回 *T*.insert(0:*S*,null:F_1,…,*S*.~:F_n)

与关系代数不同之处在于，离散数据集支持记录作为字段取值，join 函数的返回值是一个序表，其字段取值是参与连接的序表的成员，也就是记录。

容易证明，join 函数的返回序表的长度，不会超过作为参数的序表 T_i 的长度之和。

3. 完全叉乘

前面定义的连接运算是基于主键关联的。不基于主键时，很可能出现多对多的情况，而这时的运算结果经常是没有业务意义的错误结果。我们在离散数据集中禁止了这种情况，这并不会使连接运算的适用范围损失太多，几乎有业务意义的等值连接运算都可以被描述。

关系代数的连接运算的定义非常简单，不涉及到主键。定义简单的好处在于描述能力很强，各种情况都适用，但没有体现不同场景下的不同特征，思维和书写更复杂，而且易出错。

离散数据集则将连接运算分成几种情况分别处理，从 switch 函数和 join 函数的定义可以看到，连接运算总是会涉及主键，这样能更清楚地反映该运算的业务本质，避免出错。特别重要的是，在引入了主键关联性之后，可以实现更高效的运算性能，在大数据场景下尤为明显，而关系代数则很难利用运算的特征提高性能。

不过，多对多的连接虽然少见，但仍然是有必要的。我们再补充定义一个**非等值连接函数 xjoin**(A_1:F_1,x_1;…;A_n:F_n,x_n)：

n=1 时，返回 A_1.select(x_1).new(~:F_1)

n>1 时，记 *T*= xjoin(A_1:F_1,x_1;…;A_{n-1}:F_{n-1},x_{n-1})，表达式 *v*=A_n.select(x_n)

返回 *T*.news@a(*v*; ~:F_n)

左连接选项@**1**，返回 *T*.news@a1(*v*; ~:F_n)

xjoin 函数没有关联键，无法定义出全连接。

与 join 函数类似，xjoin 也是返回以被连接序列成员为字段取值的序表。

xjoin 函数返回序表的长度最大可能是作为参数的序列 A_i 的长度之积。

非等值连接即相当于关系代数的连接运算了。其实，我们也可以用定义 xjoin 函数的方法为定义出不是主键关联的等值连接方案，但意义不大。

非等值连接很难进行性能优化。

3　游标

3.1　游标

1. 游标

在序列 *cs* 上定义两个函数：

cs.**append**(*x*)　　令 *cs*=*cs*|[*x*]

cs.**fetch**()　　若 *cs*==[]则返回一个结束符 **EOC**

否则，令 *r*=cs[1]和 *cs*=*cs*.to(2,)，并返回 *r*

@**0**　　不做 *cs*=*cs*.to(2,)动作

此时我们称 *cs* 为一个**游标**。

游标概念是为解决外存计算而发明的。我们在使用游标时将做如下约定：对游标只能执行 fetch 函数取出成员后再运算，不能使用序列的其它函数。

这样，虽然我们用序列来定义了游标，但并不能把游标再理解成序列。游标将被看成是一个定义了 fetch 函数的数据对象，它可以按次序取出数据，直到结束。要说明一个游标，需要且只需要解释清楚每次针对它的 fetch 函数将返回的内容。

和理解成序列的不同点在于，游标成员有可能是在 fetch 过程中临时追加的，大多数情况下并没有一个事先准备好的序列。为了清楚地解释 fetch 过程中如何临时追加成员，我们会使用到 append 函数，而 append 函数的作用也仅限于此，它不会参与游标的其它运算。

对游标反复执行 fetch 函数直到获得结束符 EOC 的过程，称作对游标的**遍历**。

我们再定义两个批量取数据的函数：

cs.**fetch**(*n*)　　*n*.iterate(~~|*cs*.fetch(),[]).select(~≠EOC)

相当于一次取出 *n* 个成员，但有可能提前碰到结束符。

cs.**fetch**(*n*,*x*)

n.iterate(~~|if(~~[1].*x*==*cs*.fetch@0().*x*,*cs*.fetch(),EOC),[]).select(~≠EOC)

即取出 *n* 个或当 *x* 变化时停止取出。

2. 导出游标

导出游标是基于某个已有游标定义出来的新游标，在遍历导出游标的过程中会导致对原游标的的遍历。

我们先看一种基于单个游标导出新游标的框架，记作

DC[*cs*;*x*:*a*,*z*]

其中 *cs* 是原游标，*x*,*a*,*z* 分别是三段程序（简单情况可以用表达式来表示）。设该框架定义的游标记为 *dc*，在 *x*,*z*,*a* 中会使用 *dc*.append 方法来向 *dc* 追加数据。

我们来解释 DC[*cs*;*x*:*a*,*z*]定义的游标 *dc* 的 fetch 动作：

1) 先执行 *a* 进行初始化，并执行

r=*cs*.fetch();

eoc=false;

2) *dc*.fetch()的计算过程用程序语言定义如下：

```
for ()
    if (dc≠[]) return dc.fetch();
    if (eoc) return EOC;
    if (r≠EOC)
        x;              // 执行 x 可能会用 dc.append 产生 dc 成员
        r=cs.fetch();
    else
        z;              // 执行 z 可能会用 dc.append 产生 dc 成员
        eoc=true;
```

a 是初始化动作，*x* 是每次对原游标 fetch 后的动作，*z* 是结束时的动作。*a* 和 *z* 可以省略，表示没有相应的动作。

从上面这个 fetch 过程中可以看出来，在对 *dc* 遍历之前，并不需要先遍历 *cs* 并事先计算出 *dc* 的所有成员，*dc* 的成员是在遍历它的过程中逐步产生的，并且会导致对 *cs* 的遍历。这符合我们对游标的期望：不必一次性把所有数据都取出。

DC 是个用于定义其它游标函数的框架，为了书写方面，我们约定在用 DC 定义其它游标函数时，用正写的 **r** 和 **dc** 表示上述过程中的 *r* 和 *dc*，并且约定 *dc* 的初始值是[]。

上面定义的 DC 框架是针对单个原游标产生新游标，我们还可以把它推广成针对多个游标生产新游标的框架：

DC[*cs*;*x*:*a*,*z*]

其中 *cs* 是游标序列，*x*,*a*,*z* 分别是三段程序。其定义的游标 *dc* 的 fetch 动作解释如下：

1) 先执行 *a* 进行初始化，并执行

```
r=cs.(~.fetch());
eoc=false;
```

2) *dc*.fetch()的计算过程用程序语言定义如下：

```
for ()
    if (dc≠[]) return rc.fetch();
    if (eoc) return EOC;
    if (r.select(~≠EOC)≠[])        // 还有未结束的游标
        x;          // 执行 x 可能会用 dc.append 产生 dc 成员
                    // 同时计算出需要继续取数的游标序号集合 I
        I.run(r(~)=cs(~).fetch());
    else
        z;          // 执行 z 可能会用 dc.append 产生 dc 成员
        eoc=true;
```

类似地，针对多个游标的导出游标也满足上述说法：不需要先遍历所有 *cs* 的成员并事先计算出 *dc* 的成员，*dc* 的成员在遍历它的过程逐步产生。

类似地，在用 DC 框架定义针对游标序列的游标函数时，我们用 **I** 表示上面中的 *I*。

基于单个游标的导出游标框架称为**单路**游标导出框架，基于游标序列的导出游标框架

称为**多路**游标导出框架。

3.2 遍历中计算

1. 单路计算

我们来用单路 DC 框架将针对序列/序表的部分函数在游标上重新定义。

简单对位：

cs.**new**(...)　　DC[*cs*;dc.append(r.new(...))]

@**a**　　上面 new 中用@a

无歧义时的 *cs*.new(…)也可写成 *cs*.(…)

cs.**run**(...)　　DC[*cs*;dc.append(r.run(...))]

cs.**select**(*x*)　　DC[*cs*;if(*x*,dc.append(r))]

cs.**switch**(...)　　DC[*cs*;dc.append(r.switch(...))]

cs.**join**(…)　　DC[*cs*;dc.append(r.join(...))]

有序分组：

cs.**group**(*k*)　　DC[*cs*;*x*:*a*,*z*]

k 省略用~

其中　*a*　*g*=[]

x　if(*g*(1).*k*==r.*k*,*g*=*g*|[r],(dc.append(*g*),*g*=[r]))

z　~~.append(*g*)

cs.group 函数的成员是集合，将相邻成员相同的 *k* 值汇成子集。用于分组时要求游标 *cs* 对 *k* 值有序，相当于序列的 *A*.group@o 函数。

这个函数可以再增加聚合能力：

cs.**group**(x_i:F_i,…;y_i:G_i,…)

cs.group([x_i,…]).new(~.x_i:F_i,…,y_i:G_i,…)

扩展函数：

cs.**conj**(*x*)　　DC[*cs*;(*g*=r.*x*,*g*.(dc.append(*g*.~)))]

x 省略用~

cs.**news**(*v*;...)　　*cs*.(~.news(*v*;…)).conj()

相当于 *cs*.group 的逆运算，将集合成员的游标再拆分。

2. 多路归并

再用多路 DC 框架定义多游标归并和连接运算。

有序归并函数：

cs.**mergex**(*k*)　　DC[*cs*;*x*]

k 省略用~，其中 *x* 为

p=~.pselect@a(~≠EOC);

i=*p*.minp(r(~).*k*);

I=[*i*];

dc.append(r(*i*));

mergex 返回所有游标成员的和集，当所有游标对 *k* 有序时，返回游标仍将保持对 *k* 有序的性质。

可以简单改造 mergex 函数定义中的 *x* 的最后一句为

if (r.select(r(1)≠~)==[]) dc.append(r(*i*));

即可让它实现交集运算，命名为 *cs*.mergex@**i**(*k*)。

不重复的并集（会使用到长度大于 1 的 I）和差集（仅两个游标）也可以类似实现，这里不再赘述。

如果游标 *cs* 的所有成员可以构成一个序表（结构相同的记录）时可以定义出主键，那么我们也称该主键是游标 *cs* 的主键，也用 **K**[*cs*]表示。

基于主键的概念，我们可以定义针对游标的主键连接函数：

joinx(cs_1:F_1;…;cs_n:F_n)　　DC[[cs_1,…,cs_n],*x*]

为书写方便，我们约定 EOC.key()=[]；*x* 编写如下：

```
k=r.(~.key());
m=k.maxp(~.len());
if (k.select(cmp(m;~)≠0)==[])              // 键相等
    ~~.append(new(r(1):F1,…,r(n):Fn));
    I=k.pmax@a(~.len());                   // 键最长的游标继续取数
else                                       // 键最小且最长的游标继续取数
    m=k.select(~≠[]).min();
    I=k.pselect@a(cmp(~;m)==0).maxp@a(k(~).len()).minp@a(k(~))
```

用类似方法还可以定义出左连接@**1** 和全连接@**f**，过程会更复杂一些，在键值不等时会有更多判断，但并没有根本性的障碍，这里也不再赘述。

和序表的连接函数不同，游标连接时要求各路游标对键有序，这样才能做到每个游标只遍历一次即可计算出连接结果。事实上，这在实际场景中是常常可以被满足的条件（数据对主键有序），而这种归并计算的复杂度远远低于传统 HASH 算法，特别是在涉及外存计算时的优势将更为明显。

3.3　遍历后计算

1. 聚合

前面写过针对有序游标的分组聚合运算，但游标数据不一定有序。

回顾前面针对序列定义的迭代函数 *A*.iterate，我们发现，要完成这个计算，只需要保持住每轮计算后的~~，下次再用新的~与~~一起计算，而不需要将整个 *A* 都保持住。

利用这个特征，我们仿照序列来来定义针对游标的分组聚合函数：

cs.**groups**(x_i:F_i,…;y_i:G_i,…)　其中 y_i 是可以用迭代函数计算的聚合函数

函数将返回一个序表，计算过程如下：

```
g=[];
for ()
    r=cs.fetch();
    if (r==EOC) break;
```

```
k=[r.x_i,…];
p=g.pfind(k);
if (p==null)                          //找不到
    k=g.len()+1;
    g.insert(0,x_i:F_i,…,a_i:G_i,…);   // a_i 是 y_i 作为迭代函数的初值
g(k).run(G_i=I[y_i],…);               // I[y_i]表示计算迭代函数 y_i
        // 从 y_i 中可以获得当前值~，G_i 中保持有迭代值~~
return g;
```

特别地，如果 x_i 参数省略时（那个分号不可省略），我们约定上面计算过程中，只要 g 不空，g.pfind(k)总是返回 1，即最终会返回一条记录。

与序列不同，游标并不需要 sum/count 这类简单聚合函数，它们可以用无 x_i 参数的 groups 表示（当然序列的这些函数也可以）。因为游标的遍历成本较高，一次遍历时要尽量多计算结果，因此我们会习惯于将简单聚合运算看成是特殊的分组聚合，返回序表把多个聚合值都计算出来。而序列没有这种额外的遍历成本，用多个简单函数会更方便。

2. 排序

对游标的排序运算无法使用导出游标的框架来定义了，它需要把游标的所有成员都取出来才能完成运算。但我们说过，游标是为了实现外存计算而设计的，言外之意是不可能把游标的所有成员同时取出来计算，只能逐步取出。

这时候，我们需要设计一个缓存机制，用

EC[A]

表示一个游标，其中 A 是一个序列。EC[A]将把 A 的成员缓存到外部，并组织成一个游标，可以用 fetch 函数逐步取出数据。具体到工程实现上，可以将 EC[A]理解为把 A 写入外存。

现在我们可以定义游标的排序函数：

cs.**sortx**(x)

其计算过程为：

```
ec=[];
for ()
    A=cs.fetch(n);                  //n 表示可以同时取出最多数据量
    if (A==[]) break;
    ec=ec|[EC[A.sort(x)]];
return ec.mergex(x);
```

返回值仍然是游标，依次取出成员就能得到有序的结果。

分组运算也可能产生巨大的结果，有了排序运算后，我们可以来定义大分组聚合：

cs.**groupx**(x_i:F_i,…;y_i:G_i,…)

    cs.sortx([x_i,…]).group([x_i,...]).new(~.x_i:F_i,…,y_i:G_i,…)

4 组表

4.1 组表

1. 实表

设数据结构 $S=[F_1,\ldots,F_m]$，**实表** T 由一批以 S 为结构的记录$[r1,\ldots,r_n]$构成的有序集合。称 S 为 T 的结构，$F_1,\ldots,F_m$ 是 T 的字段。

如果 T 对 $F_1,\ldots,F_k$ 有序（$0\le k\le m$），即满足：

$$i<j \Rightarrow [r_i.F_1,..,\ r_i.F_k])\le [r_j.F_1,..,\ r_j.F_k]$$

则称 $F_1,\ldots,F_k$ 为 T 的**维**。

如果上面的不等式总是严格成立，即

$$i<j \Rightarrow [r_i.F_1,..,\ r_i.F_k])< [r_j.F_1,..,\ r_j.F_k]$$

则称 $F_1,\ldots,F_k$ 是 T 的**键**，显然，此时如果 T 的成员构成一个序表，则 $F_1,\ldots,F_k$ 也是该序表的键。我们也用 K[T]表示 T 的键。

这里的 k 可以为 0，此时我们也称实表 T 没有维。

类似于序表，我们也可以设定实表最后一个键是时间键。

设有实表 $T=[r_1,\ldots,r_n]$和长度为 m+1 的数列 p 满足 $p(1)==1$，$p(m+1)==n$ 且

$p(i+1)-p(i)==$ int(n/m) 或 int$(n/m)+1$　　int 为取整函数

若 T 有键为 $K_1,\ldots,K_k$，则称 $r_{p(i)}.([K_1,\ldots,K_k])$是 T 的 ***m* 分位键**中的第 i 个。$k==1$ 时，则直接用 $r_{p(i)}.K_1$ 表示分位键。

实表用于处理大数据外存计算。它的定义看起来和序表相同，之所以没有继续延用序表概念，是因为外存计算会有很多限制，无法实施序表上定义的所有运算，需要重新定义。也就是说，实表虽然和序表相似，但其上可执行的运算与序表会有较大差别，它是为了解决高性能外存计算而设计的，就计算能力上并没有比序表更多。

2. 取出

在实表 $T=[r_1,\ldots,r_n]$上定义游标函数 T.**cursor**()，返回将依次取出这些记录的游标，即游标的 fetch 函数将依次返回 $r_1,\ldots,r_n$。实表游标可以理解为一个事先已经准备好数据的游标，不需要再执行 append 动作。

我们更常用的是带条件的实表游标函数

T.**cursor**(w)　　T.cursor().select(w)

其中 w 是由 T 的字段构成的条件表达式。特别地，w 常常会由 T 的维构成，在工程上可以利用 T 对其维有序的条件快速地实现这个函数，不需要针对 T 中每个记录计算 w。

使用 T.cursor(w)函数产生的游标 cs 称为基于 T 的实表游标，显然 T 有键时，cs 也一定有键且与 T 的键相同，我们也用 **K**[cs]表示 cs 的键。

设实表 $T=[r_1,\ldots,r_n]$有键 $K_1,\ldots K_m$，$l\ge m$，定义按键值取值函数

T.**find**($[k_1,\ldots,k_l]$)　　T.cursor($[K_1,\ldots,K_m]==[k_1,\ldots,k_m]$).fetch()(1)

即返回键值为$[k_1,\ldots,k_l]$的记录，当 $l==1$ 时，可以将 T.find($[k_1]$)简写成 T.find(k_1)。

注意这里用做参数的序列长度可能超过 *T* 的键长，只取前一部分用于比较。

对于有时间键的实表 *T*，find 函数定义类似改造为：

T.**find**([k_1,…,k_t])　　*T*.cursor([K_1,…,K_{t-1}]==[k_1,…,k_{t-1}]).fetch().maxp(K_t)

针对无时间键的实表 *T*，也将 find 函数做类似序表的延伸，当 $l \le m$ 时，定义

T.find@**s**([k_1,…,k_l])　　*T*.cursor([K_1,…,K_l]==[k_1,…,k_l]).fetch()

与 find 不同，find@s 将返回一个序表（排列）。

由于 *T* 对键有序，在工程实践时可以采用二分法快速找到相应的记录，而不必真地把游标遍历一次。

再定义批量查询函数，设 *K* 为递增序列：

T.find@**k**(*K*)　　*K*.(*T*.find(~))

实现时也可以利用 *K* 的递增性，只对 *T* 进行一次从前到后的查找就可以快速完成运算，而不必多次反复查找。

4.2　组表连接

1. 主键连接

设有有键实表游标 *cs* 和有键实表 $T_1,\ldots,T_n$，定义连接函数

joinx(*cs*:F_0;T_i:F_i;…)　　*cs*.new(~:F_0,x_i:F_i,…)

|K[T_i]|≤|K[*cs*]|时，x_i=T_i.find(~.key()))

|K[T_i]|>|K[*cs*]|时，x_i=T_i.find@s(~.key()))

cs 和 T_i 都对键有序时，工程实现时可以用 *cs* 和 T_i 之间实现同步遍历，并用 *cs* 的键值迅速定位 T_i 的记录，这样可以利用 *cs* 上带有的条件去过滤 T_i 的记录。

实表游标对键有序，可以直接执行基于有序游标的 joinx 函数实现归并运算。如果我们知道这些游标是基于实表的，则可以使用某一个实表的分位键将实表划分成若干段，再用这个分位键去划分其它实表（使用有过滤的 cursor），这样可以很容易实现并行计算。而 SQL 的连接运算不假定任何有序性，无法利用这些特点来提高性能。

2. 外键连接

设 *D* 是有键实表，*cs* 是游标，扩展游标上的外键连接函数：

cs.**join**(K_i:…,*D*,*K*)　　*cs*.new@a(*D*.find([K_i,…]):*K*)

也可以支持同时多组参数

cs.**join**(K_i:…,*D*,*K*;…)　　*cs*.new@a(*D*.find([K_i,…]):*K*,...)

D 有时间键时这个定义也仍然有效。

外键连接的维表很大时，可以利用 *D* 对键的有序性实现高速连接。

如果 *cs* 的数据量较小，可以使用 *D*.find@k 取出关联键对应的记录，以避免对大维表的遍历。

如果 *cs* 的数据量也很大，可以利用 *D* 的分位键将 *cs* 的数据划分成若干个段以缓存，然后针对每个段和 *D* 对应段进行关联，这样只需要将 *cs* 的数据做一次分段缓存即可。而 SQL 体系下需要对两个关联大表分别做 Hash 分段缓存，而且 Hash 分段的效果依赖于 Hash 函数，有可能会发生要二次缓存的情况；而上述单边分段的方法不可能出现二次缓存。

3. 附表

设有键实表 T 和 A，满足如下条件：

1） 记 k=|K[T]|，K[A].to(k)==K[T]

2） 对任何 $r\in A$，有 T.find(r.key())≠null

此时可以认定 A 是 T 的**附表**，称 T 是 A 的**主表**。附表关系是单向的，我们认定 A 是 T 的附表后，则不再认定 T 是 A 的附表。

设实表 $T_0,T_1,\ldots,T_n$ 满足：对任何 j>0，总存在 $i\geqslant 0$，使得 T_j 是 T_i 的附表。我们称这一组实表 $T_0,T_1,\ldots,T_n$ 合并构成一个**组表** T，其中 T_0 称为 T 的**基表**，所有 T_i 称为 T 的实表。

设 T 是某个组表的实表，T_i 是 T 在同一个组表中的主表或附表，扩展实表游标函数

T.cursor@**x**(T_i:F_i,...;w)　　joinx(T.cursor(w),T_i:F_i,…)

即同时将主表或附表的记录也取出来。

组表在实现时可以减少存储量和计算量，从而提高性能。主表和附表有共用的维，只要存储一次就可以了。而且在计算时天然是关联的，连接时不需要做比较运算。

5　辅助阅读材料

我们在摘要中说，离散数据集的设计目标是为了解决关系代数的各种问题。但前面的文字仅是在定义离散数据集的数据类型及其上的运算，并没有涉及背后的原因。仅仅阅读上面的正文，很难理解为什么要把计算模型设计成这样。

为此，我们在这里增补一些内容来解释离散数据集与关系代数之间的关键差异，从而理解离散数据集的优势。

人们在关系代数基础上发展了程序语言 SQL，我们也基于离散数据集模型开发了程序语言，命名为 SPL（Structured Process Language）。有时我们将用 SQL 和 SPL 的代码示例以体现关系代数和离散数据集的差异。

1. 普遍集合

关系代数中有集合的概念，但只处理由记录（也就是关系，这里我们用偏工程的术语，更易于理解）构成的集合（即表）。

离散数据集的集合概念更为普遍，可以处理任何数据构成的集合，即序列。这些集合，无论是不是由记录构成，都能执行一些共同运算，如聚合、过滤等。

而在关系代数中，单列数值构成的集合要理解成单字段的数据表，显得有些累赘。

特别地，离散数据集还支持由集合构成的集合，这样才能实现真正的分组。

2. 游离记录

离散数据集中的记录是一种基本数据类型，它可以不依赖于数据表而独立存在。数据表是记录构成的集合，而构成某个数据表的记录还可以用于构成其它数据表（排列）。比如过滤运算就是用原数据表中满足条件的记录构成新数据表，这样，无论空间占用还是运算性能都更有优势。

关系代数没有可运算的数据类型来表示记录，单记录实际上是只有一行的数据表，不同数据表中的记录也不能共享。比如，过滤运算时会复制出新记录来构成新数据表，空间和时间成本都变大。

特别地，因为有游离记录，离散数据集允许记录的字段取值是某个记录，这样可以更方便地实现外键连接。

3. 有序性

关系代数是基于无序集合设计的，集合成员没有序号的概念，也没有提供定位计算以及相邻引用的机制。SQL 实践时在工程上做了一些局部完善，使得现代 SQL 能方便地进行一部分有序运算。

离散数据集中的集合是有序的，集合成员都有序号的概念，可以用序号访问成员，并定义了定位运算以返回成员在集合中的序号。离散数据集提供了[]符号以在集合运算中实现相邻引用，并支持针对集合中某个序号位置进行计算。

有序运算很常见，却一直是 SQL 的困难问题，即使在有了窗口函数后仍然很繁琐；SPL 则大大改善了这个局面。

例 1：计算股价最高三天的涨幅

SQL：即使有窗口函数，仍然要复杂动作才能实现定位计算

```
SELECT date, price, seq, rate
FROM (
    SELECT date, price, seq,
        price/LAG(price,1) OVER (ORDER BY date ASC) rate
    FROM (
        SELECT date, price,
            ROW_NUMBER() OVER (ORDER BY price DESC) seq
        FROM stock ) )
WHERE seq<=3
```

SPL：有序集合支持下的定位计算很简单

```
T=stock.sort(date)
P=T.ptop(3,price)
T.calc(P,price/price[-1]-1)
```

例 2：计算一支股票最长连续上涨了多少天？

SQL：没有方便的相邻引用机制，要转换成复杂的分组问题

```
SELECT MAX(ContinuousDays)
FROM (
    SELECT COUNT(*) ContinuousDays
    FROM (
        SELECT SUM(RisingFlag) OVER (ORDER BY date) NoRisingDays
        FROM (
            SELECT date,
                CASE WHEN price>LAG(price) OVER (ORDER BY date) THEN 0
                ELSE 1 END RisingFlag
            FROM stock ) )
    GROUP BY NoRisingDays )
```

SPL：有了相邻引用和迭代函数，很容易按自然思维实现

```
n=0
stock.sort(date).max(if(price>price[-1],n+1,0))
```

4. 离散性与集合化

关系代数中定义了丰富的集合运算，即能将集合作为整体参加运算，比如聚合、分组等。这是 SQL 比 Java 等高级语言更为方便的地方。

但关系代数的离散性非常差，没有游离记录。而 Java 等高级语言在这方面则没有问题。

离散数据集则相当于将离散性和集合化结合起来了，既有集合数据类型及相关的运算，也有集合成员游离在集合之外单独运算或再组成其它集合。可以说 SPL 集中了 SQL 和 Java 两者的优势。

有序运算是典型的离散性与集合化的结合场景。次序的概念只有在集合中才有意义，单个成员无所谓次序，这里体现了集合化；而有序计算又需要针对某个成员及其相邻成员进行计算，需要离散性。

在离散性的支持下才能获得更彻底的集合化，才能解决诸如有序计算类型的问题。

离散数据集是即有离散性又有集合性的代数体系，关系代数只有集合性。

5. 分组理解

分组运算的本意是将一个大集合按某种规则拆成若干个子集合，关系代数中没有数据类型能够表示集合的集合，于是强迫在分组后做聚合运算。

离散数据集中允许集合的集合，可以表示合理的分组运算结果，分组和分组后的聚合被拆分成相互独立的两步运算，这样可以针对分组子集再进行更复杂的运算。

关系代数中只有一种等值分组，即按分组键值划分集合，等值分组是个完全划分。

离散数据集认为任何拆分大集合的方法都是分组运算，除了常规的等值分组外，还提供了与有序性结合的有序分组，以及可能得到不完全划分结果的对位分组。

基于有序分组运算，上述例 2 的 SQL 实现思路用 SPL 写出来也更为简单：

```
stock.sort(date).group@i(price>price[-1]).max(~.len())
```

6. 聚合理解

关系代数中没有显式的集合数据类型，聚合计算的结果都是单值，分组后的聚合运算也是这样，只有 SUM、COUNT、MAX、MIN 等几种。特别地，关系代数无法把 TOPN 运算看成是聚合，针对全集的 TOPN 只能在输出结果集时排序后取前 N 条，而针对分组子集则很难做到 TOPN，需要转变思路拼出序号才能完成。

离散数据集提倡普遍集合，聚合运算的结果不一定是单值，仍然可能是个集合。在离散数据集中，TOPN 运算和 SUM、COUNT 这些是地位等同的，即可以针对全集也可以针对分组子集。

例 3：选出每天价格最高的三支股票

SQL：先要计算出序号再选出

```
SELECT *
FROM (
    SELECT *,
        ROW_NUMBER() OVER (ORDER BY price DESC PARTITION BY date) seq
    FROM stock )
WHERE seq<=3
```

SPL：把 TOPN 当作聚合运算直接分组汇总

```
stock.groups(date;top(3;-price):top3).conj(top3)
```

SPL 把 TOPN 理解成聚合运算后，在工程实现时还可以避免全量数据的排序，从而获得高性能。而 SQL 的 TOPN 总是伴随 ORDER BY 动作，理论上需要大排序才能实现，需要寄希望于数据库在工程实现时做优化。

7. 连接理解

关系代数中，连接运算被定义为笛卡尔积再过滤，其中没有约定过滤条件的要求，这样，理论上只要运算结果是笛卡尔积的子集，都可以认为是连接运算。

离散数据集将连接运算分成了三种情况：

1. 外键连接

表 A 的某些字段与表 B 的主键关联，B 表称为 A 的外键表，A 表称为事实表。

2. 同维连接

表 A 的主键与表 B 的主键，A 表和 B 表互称为同维表。

3. 主子连接

表 A 的主键与表 B 的主键的部分字段关联，A 表称为 B 的主表，B 表称为 A 的子表。

同维连接和主子连接可以再合并称为主键连接。

可以看出，离散数据集中定义的连接运算有这样的特点：

1. 只有等值连接（连接条件是关联字段相等）
2. 都会和某个表的主键相关

在实际应用过程中，绝大多数连接运算都是等值连接，而且有业务意义的连接也几乎都和主键相关，和主键（指逻辑上的）无关的连接运算，通常会是个错误，因为会导致多对多的关联效果。

离散数据集也设计了笛卡尔积后再过滤的运算，以处理极少数用上述定义不能覆盖的场景，但并不把这种运算看成是常规的连接。

离散数据集对连接运算进行改造后，会获得一些好处。

把外键连接和主键连接区分开，可以更清晰看出数据表之间的关系结构。外键表通常是用于进一步说明某些字段的详细信息，在理解事实表时可以先忽略它。外键连接和主子连接虽然都可能有一对多的情况，但对于理解业务时的地位并不等同。

这样定义的连接剔除了多对多的情况，在处理较多表关联时，不会因漏写某个关联条件而产生多对多的错误逻辑（常常可能把数据库跑死）。

在游离记录的支持下，可以把事实表中的字段直接赋值为外键表的相应记录，这样书写起来要更直观且不易出错。

例 4：取出部门经理是中国人的美国人员工

SQL：关联关系不直观，同表两次连接要起别名

```
SELECT A.* FROM employee A
    JOIN department B ON A.department=B.id
    JOIN employee C ON B.manager=C.id
WHERE A.nation='US' AND C.nation='China'
```

SPL：在建立外键关系之后，关联条件很直观

```
employee.switch(department,department)
department.switch(manager,employee)
//上面动作可以在加载数据时做好，实际查询只有下面一句
employee.select(nation=="US" && department.manager.nation="China")
```

离散数据集的连接定义，还有在性能优化上的巨大优势，我们会在后面再谈到。

关系代数对连接的定义很简单，好处是可以涉及到非常广泛的范围，但同时也会缺失一些关键特征，而无法利用这些特征简化代码书写和实现性能优化。

8. 外存计算

关系代数是个纯数学层面的理论体系，没有考虑内存和外存的区别。

离散数据集则考虑了工程实现的方案，为外存计算做了专门的设计。

离散数据集抽象了游标对象，用于处理只能顺序访问不能随机访问的数据，这符合不特定存储设备上的外部数据的特征。提出并利用导出游标的框架，将大多数游标运算的与对应的序表运算设计得非常相似，尽量提高外存计算的透明性。基于游标运算编写的计算逻辑，可以很轻松地应用到这些外部存储的数据上。

离散数据集中还设计了提供有限随机访问能力的组表用于外部存储，这是根据硬盘这种最常见存储设备的特征设计的。同样地，基于组表运算编写的计算逻辑，也很容易在硬盘上高速实现。

附表机制允许一个组表同时存储多个关联的数据表。主键连接的关系通常在数据结构设计时就已经确定，不会在计算过程中临时指定表与关联字段。将主键连接的若干表事先存储在一起，相当于预先关联，可以减少存储量及关联计算量，获得更优性能。这也是利用了离散数据集对连接运算的理解，外键连接情况就不可以事先关联。

9. 有序下的高性能

离散数据集特别强调有序集合，利用有序的特征可以实施很多高性能算法。这是基于无序集合的关系代数无能为力的，只能寄希望于工程上的优化。

这里简要罗列一些利用有序特征后可以实施的低复杂度运算。

1)

组表对键有序，相当于天然有一个索引。对键字段的过滤经常可以快速定位，以减少外存遍历量。随机按键值取数时也可以用二分法定位，在同时针对多个键值取数时还能重复利用索引信息。

2)

通常的分组运算是用 HASH 算法实现的，如果我们确定地知道数据对分组键值有序，则可以只做相邻对比，避免计算 HASH 值，也不会有 HASH 冲突的问题，而且非常容易并行。

3)

组表对键有序，两个大组表之间主健连接可以执行更高性能的归并算法，只要对数据遍历一次，不必缓存，对内存占用很小；而传统的 HASH 分堆方法不仅比较复杂度高，需要较大内存并做外部缓存，还可能因 HASH 函数不当而造成二次 HASH 再缓存。

键有序的组表，还可以利用分位点拆分成多段，实现并行计算。HASH 算法要实现并行时需要占用较大内存，难以将并行度提高。

若参与连接的某一个组表被过滤后变小，则可以利用键有序的特征，快速定位另一个组表中对应的数据，避免全表遍历。

4)

大组表作为外键表的连接。事实表小时，可以利用外键表有序，快速从中取出关联键值对应的数据实现连接，不需要做 HASH 分堆动作。事实表也很大时，可以将外键表用分位点分成多个逻辑段，再将事实表按逻辑段进行分堆，这样只需要对一个表做分堆，而且分堆过程中不会出现 HASH 分堆时的可能出现的二次分堆，计算复杂度能大幅下降。

其中 3 和 4 利用了离散数据集对连接运算的改造，如果仍然延用关系代数的定义（可能产生多对多），则很难实现这种低复杂的算法。

结语

以上是离散数据集和关系代数的重点差异。在工程实现方面，离散数据集还有一些优势，比如更易于并行、大内存预关联提高外键连接性能等，以及一些在应用时的注意事项，比如怎样保证在数据在物理上有序、如何存储数据以支持随意分段并行等。限于篇幅，我们舍去了这些理论色彩相对较弱的内容，对已写内容也没有做深入的展开解释，不过已经可以从中窥探出这两种代数体系的不同。

我们已经完成了 SPL 的工程实现，在乾学院上还有更多 SPL 的相关资料，包括技术原理以及实践案例，以及与关系数据库上 SQL 在敏捷计算和性能提升方面的差异对比，感兴趣的读者可以前去参考。

本文仅涉及 OLAP 业务，也就是只关注数据计算，对于 OLTP 业务没有提及。可以说当前的离散数据集更适合用于数据仓库的理论基础。事实上，关系代数作为 OLTP 业务的理论基础也有诸多问题，比如无法支持数据结构的多样性、实现事务一致性的成本太高等。我们正在研究，在未来将发布进一步的想法，以解决关系代数处理 OLTP 业务时的困难。

关系代数发布于 50 年前，当时的应用需求、计算机硬件环境和当前相比，都有了巨大的变化。当时非常适应的模型体系现在不适应也是很正常的，从这个意义上讲，离散数据集可以认为是关系代数的一个进化版，但并不是完全向上兼容版。