TURING 图灵程序设计丛书 数据库系列

ELSEVIER 爱思唯尔

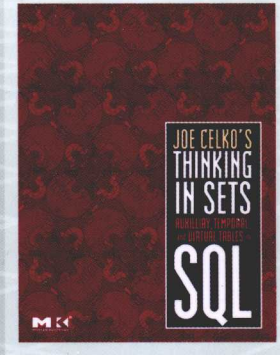
Joe Celko's Thinking in Sets

Auxiliary, Temporal, and Virtual Tables in SQL

SQL沉思录

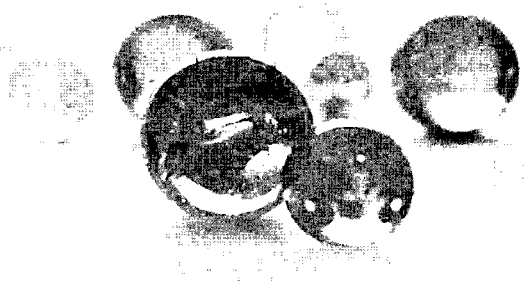
[美] Joe Celko 著
马树奇 等译

- 世界级SQL专家经典著作
- 深入揭示SQL编程本质
- 教你不同于过程和OO编程的全新思考方式



人民邮电出版社
POSTS & TELECOM PRESS

TURING 图灵程序设计丛书 数据库系列



Joe Celko's Thinking in Sets

Auxiliary, Temporal, and Virtual Tables in SQL

SQL沉思录

[美] Joe Celko 著
马树奇 等译

人民邮电出版社
北 京

图书在版编目（CIP）数据

SQL沉思录/（美）塞科（Celko, J.）著；马树奇等译.
—北京：人民邮电出版社，2009.11
（图灵程序设计丛书）
书名原文：Joe Celko's Thinking in Sets: Auxiliary, Temporal, and Virtual Tables in SQL
ISBN 978-7-115-21395-2

Ⅰ.S… Ⅱ.①塞…②马… Ⅲ.关系数据库－数据库管理系统 Ⅳ.TP311.138

中国版本图书馆CIP数据核字（2009）第162731号

内 容 提 要

本书通过大量的实例，详细说明了为提高SQL编程技术而必须面对的思想方法上的根本转变——由以过程式编程方式思考转变为以数据集的方式来思考。此外，本书还讨论了关于SQL编程中查找表、视图、辅助表、虚拟表的应用，并独到地阐明了如何在SQL系统中正确地处理时间值以及SQL编程中的其他技术难点。

本书适合广大数据库编程人员和SQL程序员学习参考。

图灵程序设计丛书

SQL沉思录

◆ 著 [美] Joe Celko
译 马树奇 等
责任编辑 傅志红
执行编辑 王军花

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子函件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京市艺辉印刷有限公司

◆ 开本：800×1000 1/16
印张：17.5
字数：414千字 2009年11月第1版
印数：1—3 000册 2009年11月北京第1次印刷
著作权合同登记号 图字：01-2009-5552号
ISBN 978-7-115-21395-2

定价：49.00元

读者服务热线：(010)51095186 印装质量热线：(010)67129223
反盗版热线：(010)67171154

版权声明

ISBN: 978-981-272-222-5.

Elsevier (Singapore) Pte Ltd.

3 Killiney Road

#08-01 Winsland House I

Singapore 239519

Tel：(65)6349-0200

Fax：(65)6733-1817

First Published 2009

2009年初版

Printed in China by POSTS & TELECOM PRESS under special arrangement with Elsevier (Singapore) Pte Ltd. This edition is authorized for sale in China only, excluding Hong Kong SAR and Taiwan. Unauthorized export of this edition is a violation of the Copyright Act. Violation of this Law is subject to Civil and Criminal Penalties.

本书简体中文版由Elsevier (Singapore) Pte Ltd.授权人民邮电出版社在中华人民共和国境内（不包括香港特别行政区和台湾地区）出版与销售。未经许可之出口，视为违反著作权法，将受法律之制裁。

献给我的侄女Olivia。每天陪她读完小画书，等她睡后我才会开始编写这本SQL手册！

前　言

本书讨论的是使用表而不是过程式代码的各种SQL编程技术。我一直跟大家说，学习SQL编程最重要的就是要摒弃原有的过程式编程，但对于那些一直都在以文件和过程代码的方式来思考的人们而言，仅仅这样说不行，还得具体说明如何采用声明式关系语言来完成相关工作。因此我编写了本书，展示实际技术，阐述思想方式。

与我的其他作品相同，本书的读者对象是那些希望掌握良好SQL编程技术的专业SQL程序员。本书假定读者对SQL语言足够了解，已经能够编写可以运行的代码，具有一年的SQL使用经验。

为什么需要具有一年的经验呢？我通过几十年讲授SQL课程发现，大多数人在由过程式编程语言（如FORTRAN、Cobol、Pascal、C系列语言）、面向对象编程语言等转向SQL语言时，都要经过多个阶段，而这段时间基本为一年。声明式语言与他们此前使用的所有编程语言都不相同。

学习一门新的编程语言就如同学习一门外语。一开始，人们会念错单词，并且总想使用自己母语的词序和语法。接下来，人们会通过努力，根据一种固定模式来正确地造句。最后，人们才能轻松地以该语言来思考和表述，并且不需要再刻意关注语言本身。

SQL编程的初级阶段只不过是闭着眼睛从别人的程序中把现有的代码复制过来，这并不是真正的编程。人们可能还会使用一种具有图形用户界面的工具来从文本文件中把SQL语句组合起来，甚至根本不需要显示自己的实际代码。

接下来的阶段是编写新的SQL代码，但仍然像使用自己最初熟悉的语言那样来操作。这种思维模式的问题在于错用术语。例如，将列称为字段，仿佛仍然在使用顺序文件系统一样。这种问题还表现在使用游标和时态表来模仿顺序文件系统，其中隐含着更大的危险和代价。

在使用SQL进行编程接近一年的时候，程序员的思维方式开始出现变化。他已经见过了优秀的SQL代码，了解了关于RDBMS（关系数据库管理系统）的相关技术，最终开始以SQL的方式来思考。走运的话，这段时间他可能还参加了大学或者其他机构开设的培训课程。

关于这种思维模式问题的最有趣的例子是，我几十年前曾经为一些工程师讲授大学课程，这些工程师只知道FORTRAN和C语言。他们认为SQL肯定也是一些WHILE循环或者IF-THEN结构。

目 录

第1章 SQL是声明式语言，不是过程式语言

在前言里，我谈到了一些FORTRAN程序员和一名LISP程序员的事，前者只会使用循环来解决问题，后者只会使用递归方式解决问题。这种情况并不少见，因为人们都喜欢使用自己了解的工具。下面讲一个笑话，不是真事：有人给一个数学家、一个物理学家和一个数据库程序员各发了一个橡皮球，并且让他们确定球的体积。

数学家认真地测量了直径，然后用球体积公式计算出了球的体积，或者认为这个球不很圆，就用三重积分计算了球的体积。

物理学家则在一个大烧杯中接满了水，把球放入水中，测量出排水量。他并不关心这个球是什么形状。

数据库程序员呢，他到橡皮球生产商的在线数据库里查了这个球的型号和产品序列号，根本不关心这是不是球。他获得了这个球的制造公差、设计形状和尺寸以及其他许多与整个橡皮球生产过程有关的参数。

这个故事说明：数学家知道如何计算，物理学家知道如何测量，而数据库技术人员知道如何查找数据。每个人都采用自己的工具来解决问题。

现在我们把问题扩展到仓库中成千上万个橡皮球。数学家和物理学家因此会花费大量的手工劳动完成任务，而数据库技术员只要下载一些信息，就能够得出橡皮球的工业标准（假设有这种标准）以及详尽得可以用于法庭辩论的文档。

1.1 不同的编程模型

自我完善的过程就是在学习新知识的同时，忘记老的习惯。

——Edsgar Dijkstra[①]

① 埃德斯加·狄克斯特拉（Edsgar Dijkstra）（1930年5月11日～2002年8月6日），荷兰计算机科学家，1972年图灵获奖得者，最先察觉“goto有害”的计算机科学大师。——编者注

编程模型有多种。过程式编程语言使用的是由流控制语句（WHILE-DO、IF-THEN-ELSE和BEGIN-END）控制的一系列过程步骤，借此把输入数据转换成输出数据。这是对编程的一种传统认识，因为这是著名的数学家约翰·冯·诺伊曼归纳出来的，后来也常被称为冯·诺伊曼模型。同样的源代码经相同的编译器编译之后，每次都生成相同的可执行模块。该程序在每次调用时都以完全相同的方式工作。这种模型中的关键字是可以预测和确定的。由于这种模型具有可确定性，所以主要用于一些数学分析。

另外，还有一些变化。一些语言使用了不同的流控制语句。FORTRAN和COBOL会在程序一开始就为数据分配全部存储区。后来的Algol系列编程语言会根据数据在程序块结构中的作用域动态地分配存储区。

Edsgar Dijkstra（参见文献www.cs.utexas.edu/users/EWD/）发明了一种非确定性语言。语句，又称为保护命令，既可以阻止语句的执行，也可以允许该语句的执行，而且在打开的语句之间没有确定的执行顺序。这种模型没有在商业化产品中实现，但它表明人们原来在编程中认为必备的因素（确定性）可以被丢弃。

函数式编程语言的基础是用一系列嵌套的函数调用来解决问题。在这些语言中，高阶函数可以转换自身的功能，这个概念非常重要。导数变换和积分变换就是这种高阶函数在数学上应用的实例。这种语言的目标之一是避免在程序中出现副作用，保证它们能够以代数的方式进行优化。特别是，一旦某个表达式与另一个表达式相等（某种意义上的相等），它们就可以替换，而不会影响整个运算结果。

APL是最成功的函数式编程语言。自从1962年Ken Iverson的*A Programming Language*一书出版之后，APL就成为流行的教学语言。IBM公司还为其台式机生产了一种专用的键盘，其中包含了APL中需要使用的容易记混的数学符号。大多数函数式语言只是在学校里使用，但也有一些在商业产品中得以应用，并延续至今。Erlang语言用于并行应用程序，R语言是一种统计语言，Mathematica是一种流行的符号化数学产品，Kx Systems使用K语言进行大数据量的财务分析。最近，ML和Haskell编程语言也在Linux和UNIX程序员中流行起来。

至此，我们又放弃了另一个此前被认为是十分基础的流控制概念：在这些语言中，没有流控制机制。

约束或者约束逻辑编程语言就是问题领域中的一系列约束。随着人们添加更多的约束，系统会确定哪些是可能的答案，哪些是不可能的。此类语言中最流行的是PROLOG，在Borland软件公司（www.borland.com）推出了价格低廉的学生版本后，该语言也在学校中流行过许多年。如果大家对这方面的内容感兴趣，可以访问Roman Barták的ON-LINE GUIDE TO CONSTRAINT PROGRAMMING（约束编程在线指导）网站（http://kti.ms.mff.cuni.cz/~bartak/constraints/index.html）。

在此我们彻底丢弃了算法的概念，只提供了对问题的说明。

面向对象（OO）编程建立在对象的概念之上，对象就是一个代码模块，其中既包含数据，又包含操作。这种编程模型是各个彼此独立又相互协作的对象构成的集合，不是由一个程序来调用其他函数。对象可以接收消息、处理数据或者向其他对象发送消息。

面向对象的思想是，人们可以独立地编写和维护每个对象，与任何具体的应用程序无关，需要的时候再把对象放到必要的位置。大家可以设想一下人们针对某种特定工作组成的社团。他们会接收客户的订单，处理订单，再返回结果。

许多年以前，INCITS H2数据库标准委员会（又称为ANSI X3H2数据库标准委员会）在美国南达科他州的拉皮德城召开过一次会议，Mount Rushmore和Bjarne Stroustrup作为特别代表参加了会议。Stroustrup先生使用透明胶片做了幻灯演示（这是在PowerPoint普及之前），介绍了贝尔实验室发明的C++和面向对象编程技术，我们也询问了相关的问题。

有一个问题是，应该如何在作为下一代SQL标准的工作模型中加入面向对象特性，当时下一代SQL标准的内部名称是SQL3。他回答说，贝尔实验室的才子们已经尝试了用4种不同的方法来解决这一问题，他们的结论是无法实现。面向对象技术对于编程十分理想，但对于数据处理却无能为力。

我曾经见到一些人努力在SQL中勉强地加入面向对象模型，只过一年就以失败告终。每个打错的字符都成了一种新的属性或者类，原本在关系模型中十分易用的查询现在却成了多个表外联结形成的怪物，冗余度呈指数级增长，编写约束几乎不可能，人们不得不放弃数据完整性以及其他特性。

既然有这么多编程模型，为什么不能有一种不同的数据模型呢？

1.2 不同的数据模型

想想简陋的穿孔卡片。穿孔卡片早在18世纪初就已经在法国用于控制织布机；后来到了1801年，Joseph Marie Jacquard在他的Jacquard织布机上使该方法趋于完善。

时间迅速来到1890年，一个名叫Herman Hollerith的人为该年度的美国人口普查发明了穿孔卡片及制表机。他的人口普查项目取得了巨大成功，Hollerith先生在1896年从政府机关离职，创办了制表机器公司（Tabulating Machine Company）。经过不断地合并和更名，该公司最终成为IBM公司，大家可能都听说过这家公司。

到20世纪70年代，“IBM卡片”及相关的机器已经在各处普及。最常见的卡片是IBM 5081，并且这个零件号已经成为了这种机器的代名词，甚至其他厂商也这么叫！穿孔卡片就是当时的数据处理技术。

卡片的物理特性决定了人们在此后的几十年中存储和处理数据的方式。这种卡片的大小与美国1887年版的美元钞票相同（8.255厘米宽，18.733厘米长）。采用这个尺寸的原因很简单：Hollerith在人口普查部门工作，他可以从街道对面的美国财政部领取抽屉以存放成摞的卡片。

卡片上有80列、12行网格，以适应穿孔的需要。这当然就是物理方面的原因。但是一旦建立了80列的惯例，它也就被确定下来了。替换键控穿孔机的早期视频终端就使用了80列、24或25行文本的形式，也就是高度相当于两张穿孔卡片，可能还留了一行用于记录出错消息。

20世纪70年代，磁带开始代替穿孔卡片，但它们也在模仿80列的惯例，尽管这样做已经根本不必要。许多早期的ANSI磁带标准中关于头部记录的内容都以此惯例为基础。老式系统只是顺其自然地把读卡器替换成了磁带机，但新的应用系统又继续在该标准之上建立起来了。

卡片的物理属性意味着必须按从左到右的顺序写入和读取数据。与此类似，成摞的卡片则按照从前到后的顺序写入和读取。

磁带文件也是采用相同的方法写入和读取数据，但它的好处是，当磁带掉到地上时，不会像一摞卡片那样散落得满地都是。与成摞的卡片相比，磁带的缺点在于人们不能有目的地手工重新排列其顺序。

卡片和磁带文件都属于相当被动的事物，应用程序送来的任何内容它们都一概接受。文件之间彼此也是独立的，原因就是它们一次只与一个应用程序连接，因此根本不知道其他文件是什么样的。

早期的磁盘系统也是模仿这一模型，按照一定的顺序读取物理上的连续存储区，数据的意义则是由读取数据的程序来决定。

过了一段时间之后，人们才发现磁盘系统原来可以将读/写头移动到磁盘上的任何物理位置。这样就出现了随机访问存储器。但具体到每个字段和每条记录中，人们至今仍然使用连续存储的概念。

关系模型是一个巨大的进步，因为它将数据的物理模型与逻辑模型彻底分开了。如果人们看到许多早期编程语言的技术说明，就会发现它们描述的是物理上连续的数据和存储方法。SQL只描述数据的行为，而不会涉及任何物理存储方法。

1.2.1　“列”不是“字段”

记录中的字段是由读取其数据的应用程序定义的。表中具体一行中的某列是由DDL（数据定义语言）中的数据库模式定义的，独立于任何应用程序。列的数据类型总是标量，并且可以是NULL（空值）。

这样会给文件带来问题。如果我们在一台磁带机上安装了错误的磁带，例如其中包含的是COBOL文件，然后用一个FORTRAN程序读取，就会产生毫无意义的输出。程序会简单地从磁带的开始位置计算字节数，然后把众多的字符分割并从左到右地填入每个字段。

在应用程序的READ或INPUT语句中，变量的顺序十分重要，因为程序就是按照这个顺序把数据读取到这些变量中。在SQL中，列的引用仅通过它们的名字来完成。不错，也可以使用像SELECT *这样缩写的子句以及像INSERT INTO <表名>这样的语句，它们会展开成一个由列名组成的列表，该列表中列名出现的物理顺序与对应的表声明中的顺序相同，上述语句只不过是其可解析成的命名列表语句的缩写。这是早期SQL技术的遗迹，那时人们也在对原有的知识进行更新，但同时还存在“面向记录”的思维方式。

SQL中NULL的用法与其他编程语言相比也是很独特的。SQL不支持在字段、记录或者文件中使用一些标记来表示缺少数据。在字段中也不能像在记录中那样添加约束，例如SQL中的DEFAULT和CHECK()子句。

字段没有数据类型。字段有具体的含义，由读取它们的程序定义，而不是由其自身定义。因此，假如一个穿孔卡片上的四个列包含的数据是1223，在一个程序中它可能表示的是一个整数，在另外一个程序中它表示的可能是一个字符串，而在第三个程序中它可能会被当做四个字段分别读入。

如何选择数据类型并非总是很明确的。新手经常会出现不加琢磨而盲目地选取数据类型的情况。我喜欢的一种方式是几乎在声明每个列时都使用VARCHAR (<可变长度>)，这里的<可变长度>是一个整数值，由具体SQL系统的默认值或者最大值来确定。在微软公司的产品中，经常使用的值是255和50。

举一个例子说明研究工作与实际设计工作之间的差别，假设要使用数字章节号对本书排序。如果在模型中将标题数字表示为字符串，那么在排序时就会丢失其自然顺序。

例如：

1.1
1.2
1.3
……
1.10

会被排序为：

1.1
1.10

1.2
1.3
……

如果在新闻组中提出这个问题，那么大家会给出多种不同的解决方案，其中包括使用递归函数、外部函数、专用的名称解析函数以及再为排序另外建立一个列。

我的解决方案是在每一节名称前填入前导数0，同时希望标题的数量不要超过99个。大多数出版商一般采用的最大标题级数是五级。

00.00
01.00
01.01
01.01.02
……

应该在DDL中使用SIMILAR TO谓词来实现，而不要试图在DML（数据操作语言）中用ORDER BY子句处理。

```
CREATE TABLE Outline
(section_nbr VARCHAR(15) NOT NULL PRIMARY KEY,
CHECK (section_nbr SIMILAR TO '[:digit:][:digit:]\.+'),
..);
```

当人们想让显示出来的标题不带前导数0时，可以在查询中使用REPLACE()或者TRANSLATE函数。我们在后面的内容中还会讨论到这些原则。

简而言之，列是活动的，并且具有自定义性；字段是被动的，其意义是由应用程序解释获得的。

1.2.2　行不是记录

行不是记录。记录是负责读取操作的应用程序定义的，就像字段那样。在应用程序的READ语句中，字段的名称告诉程序该把数据放到何处。READ语句中字段名的物理顺序十分重要。例如，READ a, b, c;与READ c, a, b;是不同的，因为其参数顺序不同。

表中的行是由数据库模式定义的，根本不是由程序决定的。在模式中通过列的名称而不是根据本地程序名或者物理位置来引用各列，这表示SELECT a, b, c FROM...语句和SELECT c, a, b FROM...语句获取的数据在进入宿主程序时是相同的。

所有的空文件都很相似，它们在操作系统的注册表中都是一个目录项，具有一个名称、一个长度为0的存储区以及一个指向其开始位置的NIL指针。尽管空表没有行，但它们也有列、约束、

安全权限以及其他结构。空表中的所有CHECK()约束都是TRUE，因此人们如果想对一些表施加业务规则，而这些表中可能包含空表的话，必须在表之外使用CREATE ASSERTION语句。

这样是为了与集合理论模型保持一致，在该理论模型中，空集就是很好的集合。SQL的集合模型与标准的算术集合论之间的区别在于，集合论中只有一个空集，但SQL中每个表都有一个不同的结构，因此不能将它们用于其非空版本不能使用的场合。

表中的行具有的另一个特点在于它们在结构上彼此都很相似，并且在模型中都属于“同类事物”。在文件系统中，记录的大小、数据类型和结构均可以不同，只要在数据流中设置标记，告诉读取程序如何解释就行了。最常见的例子是Pascal语言中的variant记录、C语言中的struct语法以及COBOL中的OCCURS子句。

下面是COBOL-85的一个例子。语法很好理解，即使是不懂COBOL的人也可以看懂。COBOL语言的程序中有一个数据声明部分，它使用了层次化的标题编号系统。其中的字段类型是字符串，由模板或者PICTURE子句描述。短横线的作用与SQL中下划线的作用相同。

```
01    PRIOR-PERIOD-TABLE.
  05    PERIOD-AMT PICTURE 9(6)
        OCCURS ZERO TO 12 TIMES
        DEPENDING ON PRIOR-PERIODS.
```

PRIOR-PERIODS字段的值控制着共有多少个PERIOD-AMT字段。ZERO选项是COBOL-85中新增的，在COBOL-74中规定它必须至少出现一次。

以Pascal语言为例，假设有一条图书馆条目的记录，表示的可能是一本书，也可能是一张CD。其声明部分如下：

```
ItemClasses = (Book, CD);
LibraryItems =
RECORD
  Ref: 0..999999;
  Title: ARRAY [1..30] OF CHAR;
  Author: ARRAY [1..16] OF CHAR;
  Publisher: ARRAY [1..20] OF CHAR;
CASE Class: ItemClasses
  OF Book: (Edition: 1..50; PubYear: 1400..2099);
     CD: (Artist: ARRAY [1..30] OF CHAR;
END;
```

ItemClasses是个标志，用于选择该使用CASE声明的那个分支。声明的顺序十分重要。大家可能也会注意到Pascal语言中的CASE声明就是SQL中CASE表达式的一个来源。

C语言中的union（联合）操作是另一个办法，可以完成前文所述的Pascal程序的操作。下列声明：