

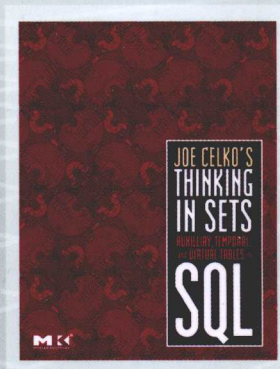
Joe Celko's Thinking in Sets  
Auxiliary, Temporal, and Virtual Tables in SQL

# SQL沉思录

[美] Joe Celko 著  
马树奇 等译

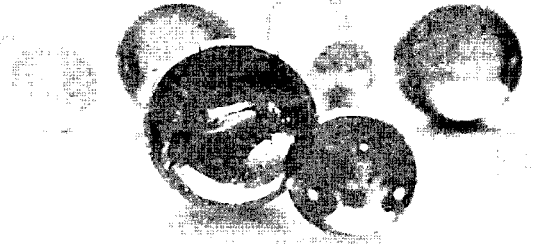
17283488182348127842647893  
88008482882379488872284888  
878482834823488823747264888

- 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



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

人

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

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

Joe Celko's Thinking in Sets: Auxiliary, Temporal, and Virtual Tables in SQL by Joe Celko, ISBN: 9780123741370.

Copyright © 2008 by Elsevier. All rights reserved.

Authorized Simplified Chinese translation edition published by the Proprietor.

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

Copyright© 2009 by Elsevier (Singapore) Pte Ltd. All rights reserved.

## **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

<b>第 1 章 SQL 是声明式语言，不是过程式语言</b> ..... 1	<b>第 3 章 数据访问和记录</b> ..... 29
1.1 不同的编程模型..... 1	3.1 顺序访问..... 29
1.2 不同的数据模型..... 3	3.2 索引..... 30
1.2.1 “列”不是“字段”..... 4	3.2.1 单表索引..... 31
1.2.2 行不是记录..... 6	3.2.2 多表索引..... 31
1.2.3 表不是文件..... 9	3.2.3 索引的类型..... 32
1.2.4 关系键不是记录定位器..... 11	3.3 散列..... 32
1.2.5 键的类型..... 12	3.3.1 数字选择..... 33
1.2.6 关系键的理想属性..... 14	3.3.2 除法散列..... 33
1.2.7 唯一，但并非不变..... 15	3.3.3 乘法散列..... 33
1.3 表作为实体..... 15	3.3.4 合并..... 33
1.4 表作为关系..... 16	3.3.5 表的查找..... 33
1.5 语句不是过程..... 16	3.3.6 冲突..... 34
1.6 分子、原子和亚原子型数据元素..... 17	3.4 位向量索引..... 34
1.6.1 分割表..... 17	3.5 并行访问..... 34
1.6.2 分割列..... 18	3.6 行和列存储..... 35
1.6.3 时间值的分割..... 19	3.6.1 基于行的存储..... 35
1.6.4 假造的非第一范式数据..... 19	3.6.2 基于列的存储..... 35
1.6.5 分子型数据元素..... 21	3.7 联结算法..... 36
1.6.6 异构数据元素..... 21	3.7.1 嵌套循环联结算法..... 37
1.6.7 检验分子型数据..... 22	3.7.2 排序合并联结算法..... 37
<b>第 2 章 硬件、数据量和维护数据库</b> ..... 23	3.7.3 散列联结算法..... 37
2.1 并行处理技术..... 23	3.7.4 Shin算法..... 38
2.2 廉价的主存储器..... 25	<b>第 4 章 查找表</b> ..... 39
2.3 固态硬盘..... 25	4.1 数据元素的名称..... 40
2.4 更廉价的二级存储器和三级存储器..... 25	4.2 多参数查找表..... 42
2.5 数据也在改变..... 26	4.3 常量表..... 43
2.6 思维方式并未改变..... 26	4.4 OTLT或MUCK表问题..... 45
	4.5 正确表的定义..... 48

第5章 辅助表	49	6.9.2 表表达式视图	88
5.1 序列表	49	6.9.3 表级CHECK()约束的视图	88
5.1.1 创建序列表	51	6.9.4 每个基表一个视图	88
5.1.2 序列构造器	51	第7章 虚拟表	90
5.1.3 替换迭代循环	52	7.1 派生表	90
5.2 排列	54	7.1.1 列的命名规则	91
5.2.1 通过递归进行排列	54	7.1.2 作用域规则	91
5.2.2 通过CROSS JOIN进行排列	55	7.1.3 公开的表名	93
5.3 函数	57	7.1.4 LATERAL()子句	94
5.4 通过表实现加密	59	7.2 CTE	96
5.5 随机数	60	7.2.1 非递归CTE	96
5.6 插值	63	7.2.2 递归CTE	97
第6章 视图	66	7.3 临时表	98
6.1 Mullins视图使用原则	66	7.3.1 ANSI/ISO标准	99
6.1.1 高效访问和计算	67	7.3.2 厂商的模型	99
6.1.2 重命名列	68	7.4 信息模式	99
6.1.3 避免增生	68	7.4.1 INFORMATION_SCHEMA声明	100
6.1.4 视图同步原则	68	7.4.2 视图及其用途的快速列表	101
6.2 可更新视图和只读视图	69	7.4.3 域的声明	102
6.3 视图的类型	71	7.4.4 定义模式	102
6.3.1 单表投影和限制	71	7.4.5 INFORMATION_SCHEMA断言	105
6.3.2 计算列	71	第8章 用表实现的复杂函数	106
6.3.3 转换列	72	8.1 没有简单公式的函数	106
6.3.4 分组视图	72	8.2 用表实现校验位	107
6.3.5 联合视图	73	8.2.1 校验位的定义	107
6.3.6 视图的联结	74	8.2.2 检错与纠错的对比	108
6.3.7 嵌套视图	75	8.3 算法的分类	109
6.4 用表构建类模型	76	8.3.1 加权和算法	109
6.4.1 SQL中类的层次结构	77	8.3.2 幂和校验位	111
6.4.2 通过ASSERTION和TRIGGER		8.3.3 Luhn算法	112
工作的子类	79	8.3.4 Dihedral Five校验位	113
6.5 数据库系统如何处理视图	79	8.4 声明不是函数, 不是过程	114
6.5.1 视图列的列表	79	8.5 用于辅助表的数据挖掘	118
6.5.2 视图的物化	80	第9章 时态表	120
6.6 嵌入式文本扩展	80	9.1 时间的本质	120
6.7 WITH CHECK OPTION子句	81	9.1.1 时间段, 不是时间子	121
6.8 删除视图	86	9.1.2 细分程度	122
6.9 过时的视图用法	87	9.2 ISO半开放时间模型	123
6.9.1 域的支持	87	9.2.1 用NULL表示永远	125



9.2.2	单时间戳表	125	11.2.7	不要害怕抛弃自己在DDL中的首次尝试	175
9.2.3	重叠的时间间隔	127	11.2.8	克制使用DML的冲动	176
9.3	状态转换表	134	11.2.9	不要以方框和箭头的方式思考	176
9.4	合并时间间隔	138	11.2.10	画圆和数据集示意图	177
9.4.1	游标和触发器	139	11.2.11	学习具体的产品	178
9.4.2	OLAP函数解决方案	140	11.2.12	把WHERE子句看做“超级变形虫”	178
9.4.3	CTE解决方案	141	11.2.13	使用新闻组、博客和因特网	178
9.5	Calendar表	142	11.3	不要在SQL中使用BIT或BOOLEAN	
9.5.1	用表提供星期值	142		标记	179
9.5.2	节假日列表	143	11.3.1	标记位于错误的层	179
9.5.3	报告期	145	11.3.2	标记使用不当使正确属性难以理解	181
9.5.4	自更新视图	145			
9.6	历史表	147			
<b>第 10 章</b>	<b>用非第一范式表清理数据</b>	<b>149</b>	<b>第 12 章</b>	<b>组特征</b>	<b>184</b>
10.1	重复的组	149	12.1	并不是按是否相等来分组	185
10.2	设计清理表	155	12.2	使用组, 不看里面是什么	186
10.3	清理操作使用的约束	157	12.2.1	半面向数据集的方式	187
10.4	日历清理	158	12.2.2	分组的解决方案	188
10.5	字符串清理	159	12.2.3	解决方案总结	189
10.6	共享SQL数据	161	12.3	根据时间分组	190
10.6.1	数据的发展	162	12.3.1	渐进式解决方案	190
10.6.2	数据库	162	12.3.2	整体数据解决方案	192
10.7	提取、转换和加载产品	163	12.4	其他使用HAVING子句的技术	192
10.7.1	加载数据仓库	164	12.5	GROUPING、ROLLUP和CUBE	194
10.7.2	全部用SQL来完成	165	12.5.1	GROUPING SET子句	194
10.7.3	提取、转换并加载	166	12.5.2	ROLLUP子句	195
			12.5.3	CUBE子句	196
			12.5.4	关于超级组的脚注	196
<b>第 11 章</b>	<b>以 SQL 的方式思考</b>	<b>168</b>	12.6	WINDOW子句	196
11.1	热身练习	168	12.6.1	PARTITION BY子句	197
11.1.1	整体, 不是部分	169	12.6.2	ORDER BY子句	198
11.1.2	特征函数	169	12.6.3	RANGE子句	198
11.1.3	尽早锁定解决方案	171	12.6.4	编程技巧	199
11.2	启发式方法	172	<b>第 13 章</b>	<b>将技术规范变为代码</b>	<b>200</b>
11.2.1	将规范表达为清晰的语句	172	13.1	不良SQL的标志	200
11.2.2	在名词前面添加“所有……的集合”几个字	172	13.1.1	代码的格式是否像另一种语言	200
11.2.3	删除问题语句中的行为动词	173	13.1.2	顺序访问假设	201
11.2.4	仍然可以使用存根	173			
11.2.5	不要担心数据的显示	174			
11.2.6	第一次尝试需要专门处理	175			

13.1.3	游标	201	16.2	关系式解决方案	232
13.1.4	糟糕的内聚度	201	16.3	其他种类的计算数据	233
13.1.5	表值函数	202	<b>第 17 章 约束类触发器</b>	<b>234</b>	
13.1.6	同一数据元素有多个名称	202	17.1	计算类触发器	234
13.1.7	数据库中的格式	202	17.2	通过CHECK()和CASE约束实现的 复杂约束	235
13.1.8	将日期保存到字符串中	203	17.3	通过视图实现复杂约束	237
13.1.9	BIT标记、BOOLEAN及其他 计算列	203	17.4	用约束实现视图操作	239
13.1.10	跨列的属性分割	203	17.4.1	3个基本操作	239
13.1.11	跨行的属性分割	203	17.4.2	WITH CHECK OPTION 子句	240
13.1.12	跨表的属性分割	203	17.4.3	WITH CHECK OPTION与 CHECK()子句	243
13.2	解决方法	204	17.4.4	视图的行为	244
13.2.1	基于游标的解决方案	204	17.4.5	联合视图	246
13.2.2	半面向数据集的解决方案	205	17.4.6	简单的INSTEAD OF触发器	247
13.2.3	完全面向数据集的 解决方案	207	17.4.7	关于INSTEAD OF触发器的 告诫	250
13.2.4	面向数据集代码的优点	207	<b>第 18 章 过程式解决方案和数据驱动的 解决方案</b>	<b>251</b>	
13.3	解释含糊的说明	207	18.1	删除字符串中的字母	251
13.3.1	回归到DDL	209	18.1.1	过程式解决方案	252
13.3.2	修改问题说明	211	18.1.2	纯粹的SQL解决方案	252
<b>第 14 章 使用过程及函数调用</b>	<b>213</b>		18.1.3	不纯粹的SQL解决方案	253
14.1	清除字符串中的空格	213	18.2	数独的两种求解方法	254
14.1.1	过程式解决方案#1	213	18.2.1	过程式解决方案	254
14.1.2	函数解决方案#1	214	18.2.2	数据驱动的解决方法	254
14.1.3	函数解决方案#2	217	18.2.3	处理已知数字	255
14.2	聚合函数PRD()	218	18.3	数据约束方法	257
14.3	在过程和函数中使用长参数列表	220	18.4	装箱问题	261
<b>第 15 章 对行编号</b>	<b>223</b>		18.4.1	过程式解决方法	261
15.1	过程式解决方案	223	18.4.2	SQL方式	262
15.2	OLAP函数	226	18.5	库存成本随时间的变化	264
15.2.1	简单的行编号	226	18.5.1	库存中使用的UPDATE语句	267
15.2.2	RANK()和DENSE_RANK()	227	18.5.2	回到装箱问题	268
15.3	节	228			
<b>第 16 章 保存计算数据</b>	<b>231</b>				
16.1	过程式解决方案	231			

# 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。

——Edsger Dijkstra<sup>①</sup>

<sup>①</sup> 埃德斯加·狄克斯特拉（Edsger Dijkstra）（1930年5月11日～2002年8月6日），荷兰计算机科学家，1972年图灵获奖获得者，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。

Edsger Dijkstra（参见文献[www.cs.utexas.edu/users/EWD/](http://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](http://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年版的美分钞票相同（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—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: