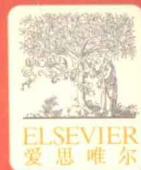


TURING

图灵程序设计丛书



[美] Joe Celko 著  
王渊 钟鸣 朱巍 译

# SQL 权威指南

(第4版)

Joe Celko's  
SQL for  
Smarties

Advanced SQL  
Programming

Fourth Edition

- 世界级数据库专家Joe Celko经典力作
- 掌握高级技术，精通SQL编程的不二之选
- 揭示SQL标准背后鲜为人知的理论与实践考量



人民邮电出版社  
POSTS & TELECOM PRESS

---

提供各种书籍的pd电子版代找服务，如果你找不到自己想要的书的pdf电子版，我们可以帮您找到，如有需要，请联系QQ1779903665.

PDF代找说明：

本人可以帮助你找到你要的PDF电子书，计算机类，文学，艺术，设计，医学，理学，经济，金融，等等。质量都很清晰，而且每本100%都带书签索引和目录，方便读者阅读观看，只要您提供给我书的相关信息，一般我都能找到，如果您有需求，请联系我QQ1779903665。

本人已经帮助了上万人找到了他们需要的PDF，其实网上有很多PDF,大家如果在网上不到的话，可以联系我QQ，大部分我都可以找到，而且每本100%带书签索引目录。因PDF电子书都有版权，请不要随意传播，如果您有经济购买能力，请尽量购买正版。

**声明：本人只提供代找服务，每本100%索引书签和目录，因寻找pdf电子书有一定难度，仅收取代找费用。如因PDF产生的版权纠纷，与本人无关，我们仅仅只是帮助你寻找到你要的pdf而已。**

“Joe Celko的这本书无疑已经成为每一位SQL程序员案头必备的权威指南。”

“本书前几版我都看过，Joe Celko这本书的最大特点就是，一旦你读了开头，就不想停下来了。”

——亚马逊读者评论

Joe Celko's SQL for Smarties

Advanced SQL Programming Fourth Edition

# SQL权威指南 (第4版)

本书是世界著名数据库专家Joe Celko经典著作的最新版，它揭示了制定SQL标准的理论及实践考量，从新颖的角度剖析了解决SQL编程中各种问题的思路，是SQL进阶不容错过的必读之作。

SQL入门不难，难的是进阶和提高。作为SQL标准委员会成员，Celko在本书中抛开商业数据库产品，对SQL语言本身进行了全面、深入、透彻的分析。本书以ANSI SQL-89为基础，兼顾SQL-92特性，全面讲解了关系型数据库设计、优化、操作方面的各种关键问题，SQL数据类型、查询、分组、集合操作、优化、数据伸缩及编码、事务与并发控制、模式等专业SQL程序员必须理解和掌握的中高级主题应有尽有。作者语言诙谐、视角独特，读来常常令人有醍醐灌顶、相见恨晚之感。书中配有大量简明易懂的示例代码，也是本书广受读者赞誉和推崇的关键。

本书适合具有一定SQL编程经验的中高级SQL程序员或DBA学习参考。



Joe Celko

世界著名的数据库专家，曾担任ANSI SQL标准委员会成员达10年之久，参与了SQL-89和SQL-92标准的制定，是世界上读者数量最多的SQL图书作者之一。他曾撰写过一系列专栏，并通过他的新闻组支持和推动了数据库编程技术以及ANSI/ISO标准的发展。除本书外，他还撰写了多部SQL经典著作，包括《SQL编程风格》、《SQL解惑》和《SQL权威指南》，上述作品的中文版均已由人民邮电出版社出版。

本书译自原版Joe Celko's SQL for Smarties: Advanced SQL Programming, Fourth Edition, 并由Elsevier授权出版。



图灵社区: [www.ituring.com.cn](http://www.ituring.com.cn)

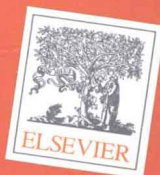
新浪微博: @图灵教育 @图灵社区

反馈/投稿/推荐信箱: [contact@turingbook.com](mailto:contact@turingbook.com)

热线: (010)51095186转604

**分类建议** 计算机/程序设计/SQL

人民邮电出版社网址: [www.ptpress.com.cn](http://www.ptpress.com.cn)



ISBN 978-7-115-29663-4



9 787115 296634 >

ISBN 978-7-115-29663-4

定价: 99.00元



TURING

图灵程序设计丛书

[美] Joe Celko 著

王渊 钟鸣 朱巍 译

# SQL 权威指南

(第4版)

**Joe Celko's  
SQL for  
Smarties**

Advanced SQL  
Programming

Fourth Edition

人民邮电出版社  
北京

## 图书在版编目 (CIP) 数据

SQL 权威指南: 第 4 版 / (美) 塞科 (Celko, J.) 著;  
王渊, 钟鸣, 朱巍译. -- 北京: 人民邮电出版社,  
2013.1

(图灵程序设计丛书)

书名原文: Joe Celko's SQL for Smarties :  
Advanced SQL Programming, Fourth Edition

ISBN 978-7-115-29663-4

I. ① S… II. ① 塞… ② 王… ③ 钟… ④ 朱… III. ①

关系数据库 - 数据库管理系统 - 指南 IV. ①

① TP311.138-62

中国版本图书馆 CIP 数据核字 (2012) 第 242360 号

## 内 容 提 要

本书为 SQL 名著中文版, 兼顾技术与实践, 全面细致介绍高级技术, 致力于打造 SQL 编程专家。本书阐释了数据库设计、优化和操作的各方面内容, 提供了成为 SQL 编程专业人士所需的技术与技巧、针对新旧挑战性难题的优秀解决方案、专业的思考方式 (以保证程序的正确性与高效性), 并涉及了数据库设计与规范化、SQL 数据类型、查询、分组、集合操作、优化等主题。另外, Joe Celko 以通俗易懂的语言叙述了一些关键问题, 比如避免使用过多 NULL 的原因及查询优化方式等。

本书适合中高级 SQL 编程人员学习参考。

## 图灵程序设计丛书 SQL 权威指南 (第 4 版)

- 
- ◆ 著 [美] Joe Celko  
译 王 渊 钟 鸣 朱 巍  
责任编辑 毛倩倩
- ◆ 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 电子邮件 315@ptpress.com.cn  
网址 <http://www.ptpress.com.cn>  
北京鑫正大印刷有限公司印刷
- ◆ 开本: 800 × 1000 1/16  
印张: 41.75  
字数: 1039 千字  
印数: 1-3 500 册
- 2013 年 1 月第 1 版  
2013 年 1 月北京第 1 次印刷
- 著作权合同登记号 图字: 01-2011-1853 号  
ISBN 978-7-115-29663-4
- 

定价: 99.00 元

读者服务热线: (010)51095186 转 604 印装质量热线: (010)67129223

反盗版热线: (010)67171154

# 版 权 声 明

*Joe Celko's SQL for Smarties: Advanced SQL Programming, Fourth Edition* by Joe Celko, ISBN: 978-0-12-382022-8 .

Copyright © 2011 by Elsevier. All rights reserved.

Authorized Simplified Chinese translation edition published by the Proprietor.

Copyright © 2013 by Elsevier (Singapore) Pte Ltd.

All rights reserved.

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, Macao SAR and Taiwan Province. 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、澳门特别行政区和台湾地区）出版与发行。未经许可之出口，视为违反著作权法，将受法律之制裁。

本书贴有 Elsevier 防伪标签，无标签者不得销售。

## 第 4 版简介

与之前推出的第 1 版、第 2 版以及第 3 版一样，第 4 版也面向想要了解高级编程技巧的 SQL 编程人员。本书读者需要具有一年以上实际 SQL 编程经验。这不是一本入门书，所以我希望亚马逊网站的评论中不会出现前几版上市时的抱怨。

本书第 1 版出版于十年前，并被部分 SQL 程序员奉为经典。在我访问过的几乎所有软件公司中都能看到程序员的桌子上放着这本书。最棒的是，我发现有即时贴伸出书本：太棒了，看来他们常常要用到这本书，所以才会用即时贴当书签。

### 十年间的变化

层次数据库及网络数据库仍然在大公司的遗留系统中运行。财富 500 强公司中仍存在 IMS (Information Management System, 信息管理系统) 和传统文件，尽管 SQL 工作者并不愿意承认这点。不过 SQL 工作者依然可以感到自豪，这十年来，基于 SQL 的系统取得了很大进展。现在，我们拥有了几乎无所不包的应用程序，以及重要的、更小的数据库。

尽管 OO 编程依然牢牢占据主导地位，但是在下一个十年里，函数式编程或许能够抢占 OO 编程的一些空间。尽管对象以及对象关系数据库找到了适合的市场，但仍没有机会占据主流地位。

2010 年后，XML 不再流行。从技术层面讲，XML 是一种用于描述数据以及将数据从某一平台移植到另一平台的语法，而它的支持工具也提供了搜索以及重新格式化的功能。INCITS H2 (前身为 ANSI X3H2 数据库标准委员会) 下设了一个 SQL/XML 委员会，以确保 XML 能与其支持工具一同工作。

数据仓库不再是只有大型企业才能使用的奢侈品。由于硬件和软件价格的下降，中型企业现在也能使用数据仓库了。编写 OLAP 查询与编写 OLTP 查询不同，也许需要专门写一本讲 OLAP 查询的“Smarties”书。

开源数据库做得很棒，也越来越符合标准。LAMP (Linux、Apache、MySQL 以及 Python/PHP) 平台占领了绝大多数网站。Ingres、Postgres、Firebird 以及其他数据库实现了 ANSI SQL-92 标准特性、大多数的 SQL-99 特性以及一些 SQL:2003 特性。

纵列数据库 (columnar database)、并行以及开放式并发机制开始出现在商业产品中，而不再局限于实验室内。SQL 标准总是在不断改变，但并不是每一次都变得更好。标准的某些部分

变得更趋于关系及集合，而其他部分则很明显地往使用过程式思想、处理非关系型数据的方向发展，这类标准建立在文件系统模型之上。引用 David McGoveran 的一句话：“委员会从未见过一个不喜欢的特性。”这句话，看来说得没错。

ANSI/ISO SQL-92 标准是一组公有的子集，通用于各类 SQL 产品，使这些产品能够为人所用。事实上，几年前美国政府将 SQL-99 标准描述为“仍在开发的标准”，要求联邦政府的合同必须与 SQL-92 标准兼容。

在开发兼容 SQL-92 标准的产品时，我们可以使用 FIPS-127 一致性测试套件对产品进行测试，这样所有供应商均可朝着同一个方向迈进。不过遗憾的是，克林顿政府中止了这个规定，而一致性问题又开始浮现。Whitemarsh 信息系统公司董事长 Michael M. Gorman 曾担任数据库标准委员会 INCITS H2 秘书长二十余年，他有一篇关于一致性问题的论文发表在 Wiscorp.com，该网站中其他关于 SQL 历史中政治方面的文章也值得一读。

今天，SQL-99 是编写在绝大多数平台上的可移植代码的标准。不过由于厂商支持 SQL:2003 特性的速度很快，我并不认为要局限于各个平台的最小交集。

## 第4版的新内容

在第2版中，我曾删除了书中的一些理论，并将这部分理论移到了 *Joe Celko's Data and Databases: Concepts in Practice* 一书中。我找不到任何理由将这部分理论移回到第4版。

由于树及层次技术相关内容足以编成一本书，因此我将这部分知识扩展并移到 *Joe Celko's Trees and Hierarchies in SQL for Smarties* 中。不过本书也简单提及了树及层次技术。

由于本书面向高级程序员，因此我将适合新手的编程技巧移到了 *Joe Celko's SQL Programming Style* 一书中讲述。本书适合那些编写真正 SQL 语句的读者，而非使用某些 SQL “方言”或伪装成 SQL 的原生语言的读者。实际上，将标准 SQL 语言翻译成他们所使用的 SQL “方言”不会太麻烦。

我尝试在方案中嵌入注解，以说明为什么该方案能够生效。我希望这种方式能帮助读者了解底层原理，以便将此原理应用到其他情形。

许多人为本书提供了素材，他们有些直接告诉我这些素材，有些则是通过新闻组提供了素材，而我无法对这些提供素材的朋友一一表示感谢，但我会尽量在他们提供的代码后面贴上他们的名字。为了避免遗漏某些朋友，我在此列出向我提供了素材或思路的朋友名单：Aaron Bertrand、Alejandro Mesa、Anith Sen、Craig Mullins（为本书的多个版本进行了校对）、Daniel A. Morgan、David Portas、David Cressey、Dawn M. Wolthuis、Don Burleson、Erland Sommarskog、Itzak Ben-Gan、John Gilson、Knut Stolze、Ken Henderson、Louis Davidson、Dan Guzman、Hugo Kornelis、Richard Romley、Serge Rielau、Steve Kass、Tom Moreau、Troels Arvin、Vadim Tropashko、Plamen Ratchev、Gert-Jan Strik。另外，还要感谢其他为我提供了素材，而我却忘记了名字的朋友们。



## 更正和补充<sup>①</sup>

请将所有更正、补充、建议、改进以及替代方案发给出版商或我本人，尤其是当你发现了某些更好的方法时，请一定告知。

出版商网站：[www.mkp.com](http://www.mkp.com)。

---

<sup>①</sup> 读者也可免费注册图灵社区 ([ituring.com.cn](http://ituring.com.cn)) 并为本书提交勘误。——编者注

# 目 录

第 1 章 数据库与文件系统 .....	1	3.5.2 为约束使用主键和断言 .....	23
1.1 实体表 .....	3	3.6 字符集相关结构 .....	25
1.2 关系表 .....	3	3.6.1 创建字符集 .....	25
1.3 行与记录 .....	3	3.6.2 创建排序规则 .....	26
1.4 列与字段 .....	4	3.6.3 创建翻译 .....	26
1.5 模式对象 .....	5	第 4 章 定位数据和特殊数值 .....	27
1.6 CREATE SCHEMA 语句 .....	6	4.1 显式的物理定位器 .....	27
第 2 章 事务与并发控制 .....	8	4.1.1 ROWID 和物理磁盘地址 .....	27
2.1 会话 .....	8	4.1.2 标识列 .....	27
2.2 事务与 ACID .....	9	4.2 生成的标识符 .....	30
2.2.1 原子性 .....	9	4.2.1 GUID .....	30
2.2.2 一致性 .....	10	4.2.2 UUID .....	31
2.2.3 隔离性 .....	10	4.3 序列生成函数 .....	32
2.2.4 持久性 .....	10	4.4 预分配值 .....	33
2.3 并发控制 .....	11	4.5 特殊序列 .....	34
2.3.1 三种现象 .....	11	4.5.1 Series 表 .....	34
2.3.2 隔离级别 .....	12	4.5.2 素数 .....	35
2.4 保守式并发控制 .....	13	4.5.3 随机顺序值 .....	37
2.5 快照隔离与乐观式并发 .....	14	4.5.4 其他序列 .....	39
2.6 逻辑并发控制 .....	16	第 5 章 基础表和相关元素 .....	40
2.7 死锁与活锁 .....	16	5.1 CREATE TABLE 语句 .....	41
第 3 章 数据库模式对象 .....	17	5.1.1 列约束 .....	41
3.1 CREATE SCHEMA 语句 .....	17	5.1.2 DEFAULT 子句 .....	43
3.2 CREATE PROCEDURE、CREATE FUNCTION 以及 CREATE TRIGGER 语句 .....	18	5.1.3 NOT NULL 约束 .....	43
3.3 CREATE DOMAIN 语句 .....	18	5.1.4 CHECK() 约束 .....	44
3.4 创建序列 .....	19	5.1.5 UNIQUE 以及 PRIMARY KEY 约束 .....	46
3.5 创建断言 .....	19	5.1.6 REFERENCES 子句 .....	47
3.5.1 为模式级约束使用视图 .....	20	5.2 嵌套 UNIQUE 约束 .....	49
		5.2.1 重叠键 .....	52

5.2.2 单列唯一性与多列唯一性	54	6.4.1 一个常见的错误	88
5.3 CREATE ASSERTION 约束	62	6.4.2 一处改进	89
5.4 临时表	62	6.5 重写技巧	94
5.5 表操作	63	6.5.1 数据表和生成器代码	95
5.5.1 DROP TABLE <表名>	64	6.5.2 用计算替代查找	96
5.5.2 ALTER TABLE	64	6.5.3 斐波那契数列	96
5.6 避免属性分割	65	6.6 谓词函数	97
5.6.1 表级属性分割	66	6.7 过程化解和逻辑分解	98
5.6.2 行级属性分割	67	6.7.1 过程式分解方案	99
5.7 在 DDL 中表现类层次关系	68	6.7.2 逻辑分解方案	100
5.8 显式物理定位器	70		
5.9 自增列	70	<b>第 7 章 过程式结构</b>	<b>102</b>
5.9.1 ROWID 与物理磁盘地址	72	7.1 创建过程	102
5.9.2 标识列	72	7.2 创建触发器	103
5.9.3 对比标识列和序列	73	7.3 游标	106
5.10 生成标识符	73	7.3.1 DECLARE CURSOR 语句	106
5.10.1 行业标准的唯一标识符	73	7.3.2 ORDER BY 子句	107
5.10.2 国防部的唯一标识符	74	7.3.3 OPEN 语句	113
5.10.3 序列生成函数	75	7.3.4 FETCH 语句	113
5.10.4 唯一值生成器	75	7.3.5 CLOSE 语句	114
5.10.5 验证源	76	7.3.6 DEALLOCATE 语句	114
5.11 关于重复行	77	7.3.7 如何使用游标	114
5.12 其他模式对象	78	7.3.8 位置更新及删除语句	117
5.13 临时表	79	7.4 序列	117
5.14 CREATE DOMAIN 语句	79	7.5 生成列	118
5.15 CREATE TRIGGER 语句	80	7.6 表函数	119
5.16 CREATE PROCEDURE 语句	80		
5.17 DECLARE CURSOR 语句	81	<b>第 8 章 辅助表</b>	<b>121</b>
5.17.1 如何使用游标	83	8.1 序列表	121
5.17.2 位置更新及删除语句	84	8.1.1 对列表进行枚举	122
		8.1.2 将序列映射为循环	124
		8.1.3 取代迭代循环	125
<b>第 6 章 过程式、半过程式以及 声明式编程</b>	<b>86</b>	8.2 查找辅助表	127
6.1 软件工程基本原理	86	8.2.1 简单转换辅助表	128
6.2 内聚性	86	8.2.2 多转换值辅助表	128
6.3 耦合度	87	8.2.3 多参数辅助表	129
6.4 大跨越	88	8.2.4 范围辅助表	129

8.2.5 层次结构辅助表	130	10.5.1 NULLIF() 函数	187
8.2.6 “一个真正的查找表”	131	10.5.2 COALESCE() 函数	187
8.3 辅助函数表	133	10.6 数学函数	189
8.3.1 用辅助表求反函数	134	10.6.1 数学运算符	189
8.3.2 用辅助函数表进行插值	141	10.6.2 指数函数	191
8.4 全局常量表	143	10.6.3 标量函数	192
8.4.1 预分配值	143	10.6.4 将数值转换为文字	192
8.4.2 素数	144	10.7 唯一值生成器	193
8.4.3 斐波那契数列	144	10.7.1 存有间隙的序列	194
8.4.4 随机顺序值	145	10.7.2 预分配数值	194
8.5 把过程代码转换成表时的注意事项	147	10.8 IP 地址	195
第 9 章 规范化	152	10.8.1 CHAR(39) 存储	195
9.1 函数依赖和多值依赖	154	10.8.2 二进制存储	196
9.2 第一范式 (1NF)	154	10.8.3 使用多个单独的 SMALLINT	196
9.3 第二范式 (2NF)	158	第 11 章 SQL 中的时间数据类型	197
9.4 第三范式 (3NF)	159	11.1 关于日历标准的说明	197
9.5 基本关键字范式 (EKNF)	160	11.2 SQL 时间数据类型	199
9.6 Boyce-Codd 范式 (BCNF)	161	11.2.1 时间的内部表示	200
9.7 第四范式 (4NF)	162	11.2.2 日期格式标准	200
9.8 第五范式 (5NF)	163	11.2.3 处理时间戳	201
9.9 域-键范式 (DKNF)	164	11.2.4 处理时间	202
9.10 规范化的实用技巧	171	11.2.5 时区和夏令时	203
9.11 键类型	172	11.3 INTERVAL 数据类型	204
9.11.1 自然键	172	11.4 时间算术	206
9.11.2 人工键	172	11.5 时间数据模型的特性	207
9.11.3 对外暴露的物理定位器	173	11.5.1 为持续时间建模	207
9.12 非规范化的实用技巧	174	11.5.2 持续时间之间的关系	209
第 10 章 SQL 的数值数据	180	第 12 章 字符数据类型	211
10.1 数值类型	180	12.1 SQL 字符串问题	211
10.2 数值类型的转换	183	12.1.1 字符串相等问题	212
10.2.1 数值的舍入和截断	183	12.1.2 字符串排序问题	212
10.2.2 CAST() 函数	185	12.1.3 字符串分组问题	213
10.3 四则运算函数	185	12.2 标准字符串函数	213
10.4 算术运算和 NULL	186	12.3 常见的厂商扩展	214
10.5 值与 NULL 的相互转换	187		

12.4	Cutter 表	222		执行删除	249
12.5	嵌套替换	223		15.1.4 在相同表内进行删除	250
第 13 章	NULL : SQL 中的缺失数据	224		15.1.5 不用声明引用完整性 在多个表中进行删除	252
13.1	空表和缺失表	225	15.2	INSERT INTO 语句	253
13.2	列中的缺失值	225	15.2.1	INSERT INTO 子句	253
13.3	上下文和缺失值	226	15.2.2	插入的性质	254
13.4	比较 NULL	227	15.2.3	批量装载和卸载实用程序	255
13.5	NULL 和逻辑	228	15.3	UPDATE 语句	255
13.5.1	子查询谓词中的 NULL	229	15.3.1	UPDATE 子句	255
13.5.2	逻辑值谓词	231	15.3.2	WHERE 子句	256
13.6	算术中的 NULL 值	231	15.3.3	SET 子句	256
13.7	函数中的 NULL 值	231	15.3.4	利用第二张表进行更新	257
13.8	NULL 和宿主语言	231	15.3.5	在 UPDATE 中使用 CASE 表达式	259
13.9	NULL 的设计忠告	232	15.4	常见厂商扩展的缺陷说明	261
13.10	关于多 NULL 值的说明	234	15.5	MERGE 语句	263
第 14 章	多列数据元素	237	第 16 章	比较或 theta 操作	266
14.1	距离函数	237	16.1	数据类型转换	266
14.2	在 SQL 中存储 IPv4 地址	239	16.1.1	日期显示格式	267
14.2.1	使用单个 VARCHAR(15) 列 表示 IPv4 地址	239	16.1.2	其他显示格式	268
14.2.2	使用一个 INTEGER 列表示 IPv4 地址	239	16.2	SQL 中的行比较	268
14.2.3	使用四个 SMALLINT 列表 示 IPv4 地址	240	16.3	IS [NOT] DISTINCT FROM 操作符	270
14.3	在 SQL 中存储 IPv6 地址	241	第 17 章	值化谓词	271
14.4	货币与其他单位的转换	242	17.1	IS NULL 谓词	271
14.5	社会安全号	242	17.2	IS [NOT] {TRUE   FALSE   UNKNOWN} 谓词	272
14.6	有理数	245	17.3	IS [NOT] NORMALIZED 谓词	273
第 15 章	表操作	246	第 18 章	CASE 表达式	275
15.1	DELETE FROM 语句	246	18.1	CASE 表达式	275
15.1.1	DELETE FROM 子句	246	18.1.1	COALESCE() 和 NULLIF() 函数	278
15.1.2	WHERE 子句	247	18.1.2	带 GROUP BY 的 CASE 表达式	278
15.1.3	根据辅助表中的数据				

18.1.3 CASE、CHECK()	280	22.5 EXISTS() 和引用约束	320
子句和逻辑蕴涵	280	22.6 EXISTS 和三值逻辑	320
18.2 子查询表达式和常量	283		
18.3 Rozenshtein 特征函数	283	第 23 章 量子子查询谓词	323
第 19 章 LIKE 与 SIMILAR TO 谓词	285	23.1 标量子查询比较	323
19.1 使用模式的技巧	285	23.2 量词和缺失数据	324
19.2 NULL 值和空字符串的谓词结果	287	23.3 ALL 谓词和极值函数	326
19.3 LIKE 并不是相等	287	23.4 UNIQUE 谓词	327
19.4 用联结消除 LIKE 谓词	287	23.5 DISTINCT 谓词	328
19.5 CASE 表达式和 LIKE 搜索条件	288	第 24 章 简单 SELECT 语句	329
19.6 SIMILAR TO 谓词	289	24.1 SELECT 语句执行顺序	329
19.7 字符串的有关技巧	291	24.2 单级 SELECT 语句	329
19.7.1 字符串的字符内容	291		
19.7.2 搜索与声明一个串	291	第 25 章 高级 SELECT 语句	336
19.7.3 创建字符串中的索引	292	25.1 关联子查询	336
第 20 章 BETWEEN 和 OVERLAPS 谓词	293	25.2 嵌入的 INNER JOIN	340
20.1 BETWEEN 谓词	293	25.3 OUTER JOIN	341
20.1.1 NULL 值的结果	294	25.3.1 OUTER JOIN 的一些历史	342
20.1.2 空集的结果	294	25.3.2 NULL 和 OUTER JOIN	346
20.1.3 程序设计技巧	295	25.3.3 NATURAL JOIN 与搜索式 OUTER JOIN	347
20.2 OVERLAPS 谓词	296	25.3.4 OUTER JOIN 自联结	348
第 21 章 [NOT] IN() 谓词	305	25.3.5 两次或多次 OUTER JOIN	349
21.1 优化 IN() 谓词	306	25.3.6 OUTER JOIN 和聚合函数	351
21.2 用 IN() 谓词替换 OR	309	25.3.7 FULL OUTER JOIN	351
21.3 NULL 和 IN() 谓词	309	25.4 UNION JOIN 操作符	352
21.4 IN() 谓词和引用约束	312	25.5 标量 SELECT 表达式	353
21.5 IN() 谓词和标量查询	313	25.6 旧 JOIN 语法与新 JOIN 语法	354
第 22 章 EXISTS() 谓词	315	25.7 受约束的 JOIN	355
22.1 EXISTS 和 NULL	316	25.7.1 库存和订单	355
22.2 EXISTS 和 INNER JOIN	318	25.7.2 稳定的婚姻	356
22.3 NOT EXISTS 和 OUTER JOIN	318	25.7.3 将球装入盒中	360
22.4 EXISTS() 和量词	319	25.8 Codd 博士的 T 联结	363
		25.8.1 Stobbs 方案	366
		25.8.2 Picere 方案	367



25.8.3 参考文献	368	27.1.1 按范围分区	393
<b>第 26 章 虚拟表: 视图、派生表、CTE 及 MQT</b>	369	27.1.2 单列范围表	394
26.1 查询中的视图	369	27.1.3 用函数进行分区	394
26.2 可更新视图和只读视图	370	27.1.4 按顺序分区	395
26.3 视图的类型	371	27.1.5 使用窗口函数进行分区	397
26.3.1 单表投影和限制	371	27.2 关系除法	398
26.3.2 计算列	371	27.2.1 带余除法	399
26.3.3 转换列	372	27.2.2 精确除法	400
26.3.4 分组视图	372	27.2.3 性能说明	400
26.3.5 联结视图	373	27.2.4 Todd 的除法	401
26.3.6 视图的联结	374	27.2.5 带 JOIN 的除法	403
26.3.7 嵌套视图	375	27.2.6 用集合操作符进行除法	403
26.4 数据库引擎如何处理视图	376	27.3 Romley 除法	404
26.4.1 视图列列表	376	27.4 RDBMS 中的布尔表达式	407
26.4.2 视图物化	376	27.5 FIFO 和 LIFO 子集	408
26.4.3 内嵌文本扩展	377	<b>第 28 章 分组操作</b>	411
26.4.4 指针结构	378	28.1 GROUP BY 子句	411
26.4.5 索引和视图	379	28.2 GROUP BY 和 HAVING	412
26.5 WITH CHECK OPTION 子句	379	28.3 多层次聚合	415
26.6 删除视图	383	28.3.1 多级聚合的分组视图	415
26.7 视图与临时表的使用提示	384	28.3.2 多层次聚合的子查询表达式	416
26.7.1 使用视图	384	28.3.3 多层次聚合的 CASE 表达式	417
26.7.2 使用临时表	385	28.4 在计算列上分组	418
26.7.3 用视图扁平化表	385	28.5 成对分组	418
26.8 使用派生表	387	28.6 排序和 GROUP BY	420
26.8.1 FROM 子句中的派生表	387	<b>第 29 章 简单聚合函数</b>	422
26.8.2 包含 VALUES 构造器的派生表	388	29.1 COUNT() 函数	422
26.9 公用表表达式	389	29.2 SUM() 函数	426
26.10 递归公用表表达式	390	29.3 AVG() 函数	427
26.10.1 简单增量	391	29.3.1 空组的平均数	428
26.10.2 简单树遍历	391	29.3.2 多个列上的平均值	429
26.11 物化查询表	392	29.4 极值函数	430
<b>第 27 章 在查询中分区数据</b>	393	29.4.1 简单的极值函数	430
27.1 覆盖和分区	393	29.4.2 广义极值函数	432
		29.4.3 多条件极值函数	438

29.4.4	GREATEST() 和 LEAST() 函数	439	第 31 章	SQL 中的描述性统计	463
29.5	LIST() 聚合函数	442	31.1	众数	463
29.5.1	使用递归 CTE 的 LIST 聚合函数	442	31.2	AVG() 函数	464
29.5.2	交叉表的 LIST() 函数	443	31.3	中值	464
29.6	PRD() 聚合函数	443	31.3.1	中值编程问题	465
29.6.1	通过表达式实现 PRD() 函数	444	31.3.2	Celko 第一中值	466
29.6.2	通过对数实现 PRD() 聚合函数	445	31.3.3	Date 第二中值	467
29.7	位运算符聚合函数	447	31.3.4	Murchison 中值	468
29.7.1	OR 位运算符聚合函数	448	31.3.5	Celko 第二中值	468
29.7.2	AND 位运算符聚合函数	449	31.3.6	Vaughan 提出的应用视图 的中值	470
第 30 章	高级分组、窗口聚合 以及 SQL 中的 OLAP	450	31.3.7	使用特征函数的中值	470
30.1	星模式	450	31.3.8	Celko 第三中值	473
30.2	GROUPING 操作符	451	31.3.9	Ken Henderson 的中值	475
30.2.1	GROUP BY GROUPING SET	451	31.3.10	OLAP 中值	476
30.2.2	ROLLUP	452	31.4	方差和标准偏差	478
30.2.3	CUBE	452	31.5	平均偏差	479
30.2.4	SQL 的 OLAP 示例	453	31.6	累积统计	479
30.3	窗口子句	454	31.6.1	运行差分	479
30.3.1	PARTITION BY 子句	454	31.6.2	累积百分比	481
30.3.2	ORDER BY 子句	454	31.6.3	序号函数	483
30.3.3	窗口帧子句	455	31.6.4	五分位数和相关统计	486
30.4	窗口化聚合函数	456	31.7	交叉表	486
30.5	序号函数	457	31.7.1	通过交叉联结建立交叉表	489
30.5.1	行号	457	31.7.2	通过外联结建立交叉表	490
30.5.2	RANK() 和 DENSE_RANK()	457	31.7.3	通过子查询建立交叉表	490
30.5.3	PERCENT_RANK() 和 CUME_DIST()	457	31.7.4	使用 CASE 表达式建立 交叉表	491
30.5.4	一些示例	458	31.8	调和平均数和几何平均数	491
30.6	厂商扩展	460	31.9	SQL 中的多变量描述统计数据	492
30.6.1	LEAD 和 LAG 函数	460	31.9.1	协方差	492
30.6.2	FIRST 和 LAST 函数	461	31.9.2	皮尔森相关系数 $r$	493
30.7	一点历史知识	462	31.9.3	多变量描述统计中的 NULL 值	493
			31.10	SQL:2006 中的统计函数	494
			31.10.1	方差、标准偏差以及 描述统计	494
			31.10.2	相关性	494

31.10.3 分布函数	495	34.2.1 没有 NULL 值和重复行时的 INTERSECT 和 EXCEPT 操作	534
<b>第 32 章 子序列、区域、 顺串、间隙及岛屿</b>	<b>496</b>	34.2.2 存在 NULL 值和重复行时的 INTERSECT 和 EXCEPT 操作	535
32.1 查找尺寸为 $n$ 的子区域	496	34.3 关于 ALL 和 SELECT DISTINCT 的一个说明	536
32.2 为区域编号	497	34.4 相等子集和真子集	536
32.3 查找最大尺寸的区域	499	<b>第 35 章 子集</b>	<b>538</b>
32.4 界限查询	502	35.1 表中的每个第 $n$ 项	538
32.5 顺串和序列查询	503	35.2 从表中选取随机行	539
32.6 数列的求和	506	35.3 CONTAINS 操作符	543
32.7 交换和平移列表值	509	35.3.1 真子集操作符	543
32.8 压缩一系列数值	510	35.3.2 表的相等操作	544
32.9 折叠一系列数值	510	35.4 序列间隙	547
32.10 覆盖	511	35.5 重叠区间的覆盖问题	549
<b>第 33 章 SQL 中的矩阵</b>	<b>516</b>	35.6 选取有代表性的子集	552
33.1 通过命名列进行访问的数组	516	<b>第 36 章 SQL 中的树和层次结构</b>	<b>556</b>
33.2 通过下标列进行访问的数组	519	36.1 邻接列表模型	557
33.3 SQL 的矩阵操作	520	36.1.1 复杂约束	557
33.3.1 矩阵等式	521	36.1.2 查询的过程遍历	559
33.3.2 矩阵加法	521	36.1.3 更改表	560
33.3.3 矩阵乘法	522	36.2 路径枚举模型	560
33.3.4 矩阵转置	523	36.2.1 查找子树和节点	561
33.3.5 行排序及列排序	524	36.2.2 找出层次和后代	561
33.3.6 其他矩阵操作	524	36.2.3 删除节点和子树	562
33.4 将表扁平化为数组	524	36.2.4 完整性约束	562
33.5 比较表格中的数组	526	36.3 层次结构的嵌套集合模型	563
<b>第 34 章 集合操作</b>	<b>528</b>	36.3.1 计数特性	564
34.1 UNION 和 UNION ALL	528	36.3.2 包含特性	564
34.1.1 执行顺序	530	36.3.3 下级节点	565
34.1.2 混合使用 UNION 和 UNION ALL 操作符	531	36.3.4 层次聚合	566
34.1.3 对同一表中的列执行 UNION 操作	531	36.3.5 删除节点和子树	566
34.2 INTERSECT 和 EXCEPT	531	36.3.6 将邻接列表转换为嵌套集合 模型	567
		36.4 其他表现树和层次结构的模型	569

第 37 章 SQL 中的图	570	38.9 2000 年问题	616
37.1 邻接列表模型图	570	38.9.1 零	616
37.1.1 SQL 和邻接列表模型	571	38.9.2 闰年	617
37.1.2 路径与 CTE	572	38.9.3 千年问题	618
37.1.3 环状图	577	38.9.4 旧数据中的怪异日期	619
37.1.4 邻接矩阵模型	579	38.9.5 后果	619
37.2 分割嵌套集合模型表示的图节点	580	第 39 章 优化 SQL	620
37.2.1 图中的所有节点	581	39.1 访问方法	621
37.2.2 路径端点	581	39.1.1 顺序访问	621
37.2.3 可达节点	582	39.1.2 索引访问	621
37.2.4 边	582	39.1.3 散列索引	622
37.2.5 入度和出度	582	39.1.4 位向量索引	622
37.2.6 源节点、汇聚节点、孤立节点 和内部节点	583	39.2 如何建立索引	622
37.2.7 将无环图转化为嵌套集合	584	39.2.1 使用简单查询条件	623
37.3 多边形中的点	586	39.2.2 简单字符串表达式	624
37.4 图论参考书目	588	39.2.3 简单时间表表达式	625
第 38 章 时间查询	589	39.3 提供额外信息	626
38.1 时间数学	589	39.4 谨慎建立多列索引	627
38.2 个性化日历	591	39.5 考察 IN 谓词	627
38.3 时间序列	592	39.6 避免 UNION	629
38.3.1 时间序列中的间隙	593	39.7 联结胜于嵌套查询	629
38.3.2 连续时间段	595	39.8 使用更少的语句	630
38.3.3 相邻事件中缺失的时间	600	39.9 避免排序	631
38.3.4 查找日期	603	39.10 避免交叉联结	634
38.3.5 时间的起始点和结束点	604	39.11 了解优化器	635
38.3.6 开始时间和结束时间	605	39.12 在模式更改后重编译静态 SQL	636
38.4 儒略日	606	39.13 临时表有时能带来方便	637
38.5 其他时间函数	609	39.14 更新统计数据	639
38.6 星期	610	39.15 不要迷信较新的特性	639
38.7 在表中对时间建模	612	参考文献	642
38.8 日历辅助表	614		

## 第1章

## 数据库与文件系统



给我们带来麻烦的往往不是我们完全不知道的事情，而是那些我们自以为了解了，但其实不然的事情。

——阿蒂默斯·沃德（1834—1867），美国作家、幽默作家

数据库和 RDBMS 与那些伴随着 COBOL、FORTRAN、C、BASIC、PL/I、Java 等过程语言或面向对象语言一起出现的文件系统截然不同，SQL 语言本身没有 I/O 系统，因此我们通常把 SQL 解读为“几乎不能称为语言”（Scarcely Qualifies as a Language）。SQL 依赖于宿主语言，以便从终端用户获取数据或为其返回数据。

编程语言通常都以某一底层模型为基础。如果了解这个模型，就能更好地理解这门语言。例如，FORTRAN 语言建立在代数学理论之上，虽然这并不表明 FORTRAN 语言酷似代数学，但如果你了解代数学，那么对 FORTRAN 语言就不会完全陌生，你可以在赋值语言中写出表达式，也可以猜出那些没有见过的库函数所代表的意思。

在大多数其他编程语言中，程序员习惯于对文件进行处理。文件的设计源于纸质表格，这些文件都存储在计算机里并且非常依赖宿主编程语言。FORTRAN 语言很难读取 COBOL 文件内容，反之亦然。事实上，在相同语言编写的程序之间共享文件也是非常困难的事情。

文件的最基本形式是一组记录，这些记录按照一定的次序记录在文件里，而我们能够通过物理位置引用记录。我们可以打开一个文件，读取第一条记录，之后依次读取后续的记录，直到读完最后一个记录。当读完最后一个记录后再次读取将会返回 EOF（End Of File）值。我们可以在这些记录中来回切换，并且一次操作一条记录，这些操作对于其他程序打开的其他文件没有任何影响，只有程序才能修改文件。

SQL 语言的模型是集合数据，而不是物理文件。SQL 语言的“工作单位”是整个模式（schema），而不是单个表格。

我们在学校里学过数学中集合的概念。集合是无序的，集合中的成员都属于同一类型。当我们对集合进行操作时，这些操作一次作用于全部成员。这就意味着，当我要从正整数集合中选择奇数子集时，返回的是所有奇数值组成的一个集合。我并没有一个个地对整数进行扫描，然后将满足条件的奇数组成集合，仅仅是定义了奇数的规则“如果将整数除以 2 时余数为 1，那么该数为奇数”，这个规则可用于对任何整数进行奇数判定。集合模型具有很多优点，并行处

理只是其中之一。

FORTRAN 并不是一门完美的代数学语言，同样地，SQL 也不是完美的集合语言，后面的内容将展示这一点。但如果对 SQL 语言中的某些内容存疑，考虑一下如何在集合中阐述这些内容，也许就能得到正确的答案。

SQL 语言分三部分，即三个子语言：

- DDL (Data Declaration Language, 数据描述语言)；
- DML (Data Manipulation Language, 数据操作语言)；
- DCL (Data Control Language, 数据控制语言)。

DDL 用于定义数据库的内容，并维护数据完整性。存储在文件中的数据并没有完整性约束、默认值以及关系的概念。与一个老程序员讨论如何使用 FORTRAN 语言读取 COBOL 文件，而且要求无误地获取操作结果，会得到这样的答案：如果一个程序“乱动”数据，那么另一个程序就会表现异常。

在 DDL 语句上花费精力越多，你的 RDBMS 就越能更好地工作。DDL 和 DML 及 DCL 语言一起工作；SQL 是一个整体，而不是这三种语言无关联的简单拼凑。

DML 是大多数读者执行查询、插入、更新以及删除操作并赖以以为生的语言。如果对数据进行了规范化处理，并创建了良好的模式，那么你在工作中将事半功倍。每次编译过程式语言代码时候，操作步骤都是一样的，SQL 语言则不然，每次处理一个查询或其他语句时，执行计划会根据数据库的当前状态而变更。正如柏拉图在《克拉底鲁篇》中所说：“所有事物都在改变，没有永恒之物”(Everything flows, nothing stands still)。

DCL 不是数据安全相关的语言，而是一门关于访问控制的语言。DCL 语言并不加密数据；SQL 标准中没有提及数据加密，不过各个厂商为用户提供了一些可选方案。大多数 SQL 书籍并不会强调 DCL 语言，我也不打算在此花费太多篇幅。

有必要为 DCL 语言单独写一本微型书，它就好像是三脚凳中容易被忽视的那根支撑脚。也许以后我会写一本这样的书。

现在让我们看一些基本概念。如果你对传统文件系统中数据处理的过程已经有所了解，那首先应该抛弃文件处理中的一些认识。

(1) 数据库模式不是文件集合。各个文件之间不具有任何关系，应用程序负责处理文件之间的关系。SQL 标准没有提及任何与物理存储相关的问题，而文件建立于连续的物理存储之上。文件系统的出现伴随着打孔卡的产生，而磁带模拟出了文件系统，之后便是早期的文件系统。因为文件系统是所有问题的根源，所以我把它列入了第一条。

(2) 数据表不是文件。表是模式的一部分，而模式是一个工作单元，不能在同一个模式下存在多个同名表。文件系统负责为加载到某一物理驱动器上的文件分配名称，而数据库中的表也有表名。文件本身是真实存在的，而表却可以是虚拟的(视图、通用表表达式，查询结果等)。

(3) 行不是记录。记录的意义由读取它们的应用程序决定。记录是有先后次序的，所以可以对记录执行第一个、最后一个、下一个、前一个等操作；而行并没有物理顺序(ORDER BY 语句是游标的子句)。记录通过指针以及记录号指向其物理位置。行则通过关系键进行定位，关系键的概念建立在数据模型中属性子集唯一的基础之上。标准中并没有规定这一机制，因此在各



个 SQL 版本中有很大差异。

(4) 列不是字段。字段的意义是由读取字段的应用所决定的。对于不同的应用而言, 字段可能具有不同的含义。记录中依次排列着各个字段, 字段本身不具有数据类型、约束以及默认值。这就是主动数据与被动数据之间的对比! 除此之外, 列是可以为空的, 而字段没有这样的概念。字段必须是客观存在的, 而列可以是计算出来的或虚拟的。如果想有一个计算出来的列值, 便可以通过应用程序进行计算, 文件并不负责计算。

另一个概念上的区别在于文件常常是整个业务流程的相关数据。文件自身必须包含足够的数以辅助应用程序进行业务逻辑处理。文件往往都是混合数据, 这些数据可以使用业务流程中的业务名称进行描述, 例如, “工资表”或其他类似的名称。表可以是业务流程中的实体或关系。这意味着一个文件中的数据通常都会放置到多个表中。表倾向于存储可以用一个单词描述出来的纯数据, 工资单可以通过员工打卡表、员工表、项目表等多个数据表进行描述。

## 1.1 实体表

实体是物理或逻辑上具有某一含义的事物。一个人、一笔买卖或一件货物都是实体。在关系型数据库中, 实体由它的属性定义。实体以表中行的形式体现, 数据行中的每一列代表了实体的一个属性。属性的值是一个标量值。

为了提醒用户表是实体的集合, 我倾向于使用描述实体功能的集合名词或复数名词定义系统中相关的表名。因此, “Employee”不是一个好的表名, 因为这是单数形式, 而复数形式的“Employees”更好些。“Personnel”最适合用作表名, 因为它具有集合性, 同时也不会被误以为是各自无关个人的集合。这个名字同时也符合 ISO 11179 标准对元数据的要求。我的另外一本书《SQL 编程风格》中详细描述了这些标准。

如果多个表具有完全相同的结构, 那么这些表是同一类元素的集合。但应该将同一类的数据元素归入到一个集合中! 反之, 文件是物理上独立的存储单元, 这些文件可能非常相似: 每个磁带/磁盘文件都代表了处理过程的一个步骤, 如获取原始数据、编辑数据、最后将数据存档。而在 SQL 中, 这些应该是表中的一个状态标志。

## 1.2 关系表

在 SQL 中, 将表中的列指向一个或多个实体表, 可以表现关联关系。

脱离了实体, 关系就没有意义, 不过关系可以包含自身的属性。举例来说, 一个演艺合同可能会涉及经纪人、雇主、演出人员。付款方式是合同本身的属性, 而不是这三方的属性, 这表明有一列必须引用其他表。文件和字段无法做到这点。

## 1.3 行与记录

行不同于记录。应用程序中定义并使用记录。而行则不然, 行由数据库模式定义, 而不是应用程序。应用程序中的读写语句将用到字段的名称。数据库模式为行进行命名。同样地, 读取字段的物理顺序至关重要。读取 a、b、c 字段与读取 c、a、b 字段得到的内容是不同的, 但

在数据库中选择 (select) a、b、c 与选择 c、a、b 获得的数据是完全一致的。

所有的空文件都很相似，它们都是操作系统中具有名称的目录项，这些空文件存储了 0 B 的内容。与它们相比，空表虽然没有数据行，但依然包含列、约束、安全权限以及其他结构。

集合理论中空集是一个完全有效的集合，SQL 中这一点与集合理论一致。SQL 的集合模型与标准数学中的集合理论的差异之处在于，集合理论中只有一个空集合，而在 SQL 集合模型中，由于每个表都具有不同的数据结构，因此如果某处不能使用集合的非空版本，那么该集合的空集一定也不适用。

同一表中的行还具有以下特点：这些行的数据结构完全相同，在模型中它们表示“同一类事物”。在文件系统中，记录在长短、数据类型以及结构上都有所不同，而数据流中的标志将告知应用程序如何解析数据。最常见的例子包括 Pascal 中的变体记录 (variant record)，C 语言的 struct 语法以及 COBOL 的 OCCURS 子句。

COBOL 语言中的 OCCURS 关键字以及 Pascal 语言中的 VARIANT 记录都包含了一个数值，通过该数值告诉在程序当前的记录中，子记录结构的重复次数。

C 语言中的联合体 (union) 虽然不是变体记录，但却可以基于相同的物理内存进行变体映射。例如：

```
union x { int ival; char j[4]; } mystuff;
```

该语句定义了 mystuff 联合体，该联合体既可以是整型（在大多数 C 编译器中是 4 字节，不过这段代码是不可移植的），也可以是 4 字节的数组，这取决于通过 mystuff.ival 还是 mystuff.j[0] 调用它。

比这更重要的差异是，文件通常都包含有汇总其他记录子集统计信息的记录，这也称为控制小计报表 (control break report)。没有规定明确同一个文件中的记录需要以什么方式关联起来，文件只是包含二进制数据的数据流，读取文件的程序负责解释文件内容。

## 1.4 列与字段

记录中字段的含义由读取该字段的应用程序定义，而表中数据行的某一列的含义是由数据库模式定义的。列的数据类型总是标量。

在 READ 和 INPUT 语句中，程序将依照应用程序变量在语句中的顺序读取这些变量，因此变量的顺序非常重要。在 SQL 语言中，只能通过列名来引用列。虽然 SELECT \* 语句和 INSERT INTO <表名> 等语句中使用了会扩展成列名列表的缩写，这些扩展后的列名顺序与表定义中列名的顺序相同，不过这些缩写也仅仅是解析成命名列表的缩写而已。

SQL 语言中 NULL 值的作用不同于其他语言。字段中并不支持可作为字段、记录或文件本身一部分的“丢失数据标记”。除此之外，不能像 SQL 的 DEFAULT 和 CHECK() 语句那样为记录的字段添加约束。

文件本身是非常被动的，它会毫不反对地接受应用程序丢给它的所有内容。与此同时，文件之间毫无关联，而这只是因为它们每次只与一个应用程序联系，所以并不知道其他文件是什么样子。

数据库活动通过触发器、约束以及声明引用完整性等方式，“主动”寻求维护所有数据的正确性。

DRI (Declarative Referential Integrity, 声明引用完整性) 表明, 表中的数据实际上与另一张表 (也有可能是同一张表) 的数据存有某一特定的关系。数据库可能通过与 DRI 相关的引用操作进行自我更改。例如, 可能会有这样一条业务规则, 规定不能销售仓库中没有的产品。

可以对订单 (Orders) 表使用 REFERENCE 语句声明对仓库 (Inventory) 表的引用, 并定义级联删除 (ON DELETE CASCADE) 引用操作, 强制上述规则。与 DRI 相比, 触发器是更为通用的方式。触发器是一段程序代码, 可以在 INSERT INTO 和 UPDATE 语句执行前、执行时或执行后运行。使用触发器除了可以完成 DRI 可以完成的所有操作, 还能完成其他功能。

然而, 使用触发器会带来一些问题。尽管从 SQL-92 标准开始, 就已经为触发器定义了标准的语法, 但大多数厂商并没有实现这个语法, 而是使用专有的语法。第二个问题是, 触发器无法向优化器输入类似于 DRI 的信息。在这一节的示例中, 从 Orders 表中可以知道每个产品编号, 在 Inventory 表中也使用相同的产品编号。优化器可以在设置 EXISTS() 谓语句以及往查询中添加联结时使用这些信息。但是我们无法通过解析过程式触发器代码确定产品编号间的关系。

SQL-92 标准中出现的 CREATE ASSERTION 语句允许数据库以整个库为单位, 强制系统满足指定条件。断言不同于 CHECK() 子句, 不过其差异很难被觉察到。当往表里添加数据行时, 将执行 CHECK() 子句。

如果表是空的, 那么所有的 CHECK() 子句都将返回有效值 TRUE。因此, 如果想确认 Inventory 表非空, 我们写下了以下语句。

```
CREATE TABLE Inventory
( ...
CONSTRAINT inventory_not_empty
    CHECK (( SELECT COUNT(*) FROM Inventory) > 0),
... );
```

但是这个语句不能成功执行, 然而, 当编写以下语句时:

```
CREATE ASSERTION Inventory_not_empty
    CHECK (( SELECT COUNT(*) FROM Inventory) > 0);
```

我们获得了预期效果。这是因为断言不是在表级, 而是在模式级进行检查的。

## 1.5 模式对象

尽管大多数数据库操作是基于表进行的, 但数据库并不仅仅是数据表的集合。一个完整的数据库除了表, 还包括存储过程、自定义函数以及用户创建的游标。此外还有索引和其他一些存取方法, 而这些是用户不能直接使用的。

本章将概述数据库用户可创建的模式对象。SQL 标准将数据库用户划分为普通用户 (USER) 和管理员用户 (ADMIN), 模式对象的创建、修改、删除操作需要管理员权限, 而普通用户则能够调用模式对象并存取调用结果。

## 1.6 CREATE SCHEMA 语句

SQL 标准定义了 CREATE SCHEMA 语句，通过该语句能够一次性创建一个完整的数据库模式，事实上，几乎每种数据库产品都提供了辅助程序，帮助用户分配物理存储并创建数据库模式。这些辅助程序通常具有较大的差异，其中一些使用专有的语法来进行物理存储的分配。

数据库模式必须具有名称和默认字符集（多年前通常是 8 位的 ASCII 字符集或 ISO 标准定义的简单拉丁字符集）。现在更常见的是 Unicode（16 位）字符集。CREATE SCHEMA 语句提供了一个可选的“授权”子句，为安全性创建了一个访问控制所需的授权标识符。在此之后，就是一组模式元素的集合：

```
< 模式元素 > ::=
    < 域定义 > | < 数据表定义 > | < 视图定义 >
    | < 授权语句 > | < 断言定义 >
    | < 字符集定义 >
    | < 排序规则定义 > | < 翻译定义 >
```

在 SQL 数据库中，模式就好比是数据库的骨架，它定义了模式对象的结构以及操作规则，而模式中的各类数据就好比是依附在骨骼之上的血肉。

表是 SQL 语言中唯一的数据结构，SQL 中的表可以是持久化的（基础表），可以用于临时存储（临时表），可以是虚拟表（视图、公用表表达式和衍生表）。以上这些表在性能上没有差异，但在实现上是不同的。SQL 语言只使用一种数据结构（表），这大大简化了 SQL 处理。由于所有操作的返回类型也是表，因此无需对返回结果进行结构转换、编写特殊的操作符或是处理语言中的不规则问题。

< 授权语句 > 限制用户只能访问特定的模式元素；断言可以被看做应用于整个模式的约束条件（constraint），< 断言定义 > 目前并没有广泛使用。而 < 字符集定义 >、< 排序规则定义 > 和 < 翻译定义 > 则用于处理数据的显示。开发人员和普通用户并不需要关心上述模式对象，也不需要对这些对象进行设置，DBA（数据库管理员）将负责这些对象的设置和维护工作。

从概念上讲，表是零行或多行数据的集合，而行是一列或多列的集合。这个层次关系很重要。我们可以在模式、表、行或列级别上执行操作。例如 DELETE FROM（删除语句）并不会移除除列，而是将行移除，并留下模式中的基础表。无法删除行中的某列。

每一列都有一个特定的数据类型以及约束，该数据类型及约束便构成了某一抽象域的具体实现。由于只能通过 SQL 访问表，因此表的物理实现方式并不重要。数据库引擎负责处理所有的细节，因此不必担心处理文件时所需要处理的内部事件。事实上，几乎没有两款 SQL 产品会使用相同的内部数据结构。

对于习惯于文件系统或 PC 机的程序员而言，有两个常犯的概念性错误：第一个是认为表就是文件；第二个是认为表是电子表格。表与这两者表现都不相同。如果不了解这些基本概念，那你早晚会遇到麻烦。

很容易将表想象成文件，把行想象成记录，把列想象成字段；这些概念让人觉得熟悉。当将 SQL 中的数据转移到宿主语言时，这些数据必须转换成宿主语言数据类型和数据结构，以供

显示和使用。而宿主语言也都有内建的文件系统。

使用文件系统和 SQL 的一大区别在于，宿主语言加载 SQL 的方式不同。在使用文件系统时，应用程序需要独立地打开/关闭各个文件，而使用 SQL 时，程序每次执行连接/断开连接操作时，最小的工作单位是整个模式。虽然由于权限的关系，宿主程序也许无法查看或操作所有的表和其他模式对象，但这些对象已经是建立连接后加载数据的一部分了。

程序在文件中定义字段，而 SQL 在模式中定义列。FORTRAN 语言使用 FORMAT 和 READ 语句从文件中获取数据，与之相似，COBOL 程序在数据部（data division）中定义字段，并使用 READ 语句读取字段。诸如此类，对于每一种第三代编程语言而言，文件读写的概念都是相同的，只是在语法和选项上有所差异。

在每个程序中，文件系统使我们可以用不同的名称引用同一数据。如果文件的布局发生了变化，就必须重写使用该文件的所有程序。另外，所有的空文件都没有区别。试图读取空文件时，程序将根据返回的 EOF（文件末尾）标志执行后续操作。数据库模式定义了表的列名和数据类型，在合理的限度下，在宿主程序不知情的情况下可以对数据表进行修改。

宿主程序只需要关心如何将数据库中的数值转变成自己的变量。还记得高中数学课上学到的空集吗？空集也是一种有效集合。当一个表没有数据时，虽然它没有数据行，但仍包含列。因为表没有最后记录，因此不会通过设置 EOF 标志抛出异常。

数据库与文件系统的另一大区别是，前者能够为表和列设置约束。约束是定义每次事务后数据库都需要满足哪些要求的规则。这也意味着，相对于被动的文件系统，数据库更像是对象的集合。

表并不是电子表格，尽管在屏幕上或打印输出时，它们看起来非常相似。我可以访问电子表格的行、列、单元格，也可以通过鼠标光标导航访问一组单元格。表没有导航的概念。电子表格中的单元格除了存储数据，还能存储指令。电子表格中的行和列并没有实质的区别，将行和列翻转之后，仍然可以获得正确的结果。SQL 表则不然。

它们唯一的共性是，电子表格也是一个声明性编程语言，只不过恰巧是一门非线性语言。

很久以前，当人类还居住在“洞穴”<sup>①</sup>里，使用具备批处理文件系统的大型计算机时，事务处理是一件简单的事情。与主文件不同，这些事务将被打包成事务文件。对事务文件可以进行排序、编辑操作，还可以选择磁带驱动器上的主文件执行事务文件。这个操作将生成一个新的主文件，旧的主文件和事务文件将记录在磁带中，并将被放置在公司地下室的大橱柜中。

磁盘、多用户系统、数据库产生以后，事情变得复杂起来，而 SQL 的出现加剧了这种复杂性。不过幸好用户不需要了解事务处理的细节。下面将介绍关于事务的第一层细节。

## 2.1 会话

用户会话源于用户初次连接到数据库。登录数据库系统就好像拨打电话号码，不过需要具有密码。用户登录的标准 SQL 语法如下所示：

```
CONNECT TO <连接目标>
```

```
<连接目标> ::=
```

```
<SQL 服务器名称>
```

```
[AS <连接名称>]
```

```
[USER <用户名>]
```

```
| DEFAULT
```

不过，不同厂商的 SQL 产品的登录语法可能会有差异，也许操作系统级别的登录过程也会造成差异。

一旦建立连接，用户便能访问数据库中他具备权限的所有部分。在这个会话中，用户可以执行任意次事务。在用户执行 COMMIT WORK（提交事务）操作之前，他所执行的插入、更新、删除行的操作的变更结果并不会被数据库永久保留。

不过，如果用户不想将这些操作变动持久化到数据库中，那可以发起 ROLLBACK WORK（回滚）命令，之后数据库将停留在这些操作之前的持久化状态。

---

<sup>①</sup> 这是作者的一种幽默写法，以此表示这是在“很久以前”。——译者注



## 2.2 事务与 ACID

为了方便记忆，事务的四大属性被归纳为 ACID 属性。事务处理系统四个属性的首字母组成了这个缩写，这四个属性是：

- 原子性 (Atomicity)；
- 一致性 (Consistency)；
- 隔离性 (Isolation)；
- 持久性 (Durability)。

### 2.2.1 原子性

原子性意味着要么整个事务都持久化到数据库中，要么都不被持久化。在标准 SQL 中，COMMIT 语句成功执行后数据将被持久化，而 ROLLBACK 语句则将清除事务，并将数据库恢复到事务开始之前的（持久化的）状态。

用户可以显示调用 COMMIT 或 ROLLBACK 语句，而数据库引擎在发现错误时也会。在发生错误时，只要没有进行相反的配置，大多数 SQL 引擎都会执行 ROLLBACK 操作。

原子性意味着，当试图往表里插入 100 万行时，如果其中的一行违背了引用约束，那么所有的行都会被拒绝插入，而数据库会自动执行 ROLLBACK 操作。

这儿有一个需要权衡的地方，执行长事务时，一个很小的错误都可以让你紧皱眉头，但如果在一个会话中使用多个短事务，其他用户能够在你的事务执行间隔期间访问数据库，并且对数据库进行可能让你出乎意料的修改。

SQL:2006 标准定义了 SAVEPOINT（保存点）机制，该机制附带有 Chain 的可选项。保存点好比是事务会话的“书签”，事务在执行过程中可以设置保存点，并将事务本地回滚到保存点处。在之前的例子中，我们每 1000 行插入一个保存点，当第 999 999 行插入发生错误时，将导致事务回滚，此时数据库引擎仅仅将最后一个保存点之后的工作移除，事务将恢复到保存点之前的未提交状态（例如，行 1~999 000 已插入状态）。

保存点的语法如下所示：

```
< 保存点语句 > ::= SAVEPOINT < 保存点标识 >
< 保存点标识 > ::= < 保存点名称 >
```

实现中定义了每个 SQL 事务可以包含的最大保存点数，保存点之间可以互相嵌套，可以通过以下语句创建当前所在的保存点级别：

```
< 保存点级别声明 > ::=
NEW SAVEPOINT LEVEL | OLD SAVEPOINT LEVEL
```

通过以下语句可以取消保存点：

```
< 释放保存点语句 > ::= RELEASE SAVEPOINT
< 保存点说明 >
```

提交语句持久化了保存点级别中所做的工作或保存点链中所有的工作。

< 提交语句 > ::= COMMIT [WORK] [AND [NO] CHAIN]

同样地，可以回滚当前保存点链中整个会话所做的所有工作，也可以回滚保存点链或返回指定的保存点。

< 回滚语句 > ::= ROLLBACK [WORK] [ AND [NO] CHAIN] [< 保存点句子 >]

< 保存点句子 > ::= TO SAVEPOINT < 保存点说明 >

关于原子性，我要说的就是这些。你需要查看特定的产品，确认该产品是否具备上述的这些东西。常用的保存点替代方案是将工作划分成工作块，使用热门的程序以事务的方式运行这些块。也可以用 ETL<sup>①</sup> 工具将数据处理完之后再载入数据库。

### 2.2.2 一致性

事务刚开始时以及持久化到数据库之后，数据库都处于一致性状态。一致性状态意味着数据库同时满足数据完整性约束、关系完整性约束以及其他约束条件。

然而，这并不意味着在事务处理过程中数据库可以处于非一致性的状态。为了更好地对事务进行控制，标准 SQL 都具备将约束声明为可延迟（DEFERRABLE）约束和不可延迟（NOT DEFERRABLE）约束的能力。不过这两种约束都需要满足一个规则，即会话结束时系统必须满足所有约束条件。当事务中包含多个语句或触发了其他数据表事件时，满足上述规则就会变得较为复杂。

### 2.2.3 隔离性

事务之间是相互隔离的，隔离性意味着事务在运行过程中与其他事务互相隔离，因此又称为串行性。就好像我们之前使用批处理系统时一样，串行性执行是保障隔离性的一个方法，但实际上，这并不是一个好主意，系统必须决定如何将各个事务交织在一起，已达到隔离的效果。

在实践中，由于事务在执行时有可能能查看，也可能不能查看其他事务执行插入、更新和删除的数据，实现隔离性非常复杂，详见 2.3.2 节。

### 2.2.4 持久性

数据库存储在持久化介质中，因此即使数据库程序被毁坏，数据库本身还是持久化的。再者，在数据库系统恢复之后，数据库也会恢复到一致性状态。持久性属性的重要事项既包括数据库运行时的磁盘写入操作，也包括日志文件和备份程序。

如果在同一时刻只有一个用户访问数据库系统，那将是多么美好的一件事情，不过我们需要数据库系统的原因之一便是在同一时刻需要满足多个用户在自己的会话中同时访问系统的需要。这便引发了并发控制。

---

<sup>①</sup> 是 Extract-Transform-Load 的缩写，用于描述将资料从来源端经过提取、转换、载入至目的端的过程。——译者注

## 2.3 并发控制

并发控制是事务处理的一部分，就像交通灯系统一样，并发控制确保在多用户访问共享的数据库时，不会“撞到”其他用户。避免所有问题的方法之一，便是每一时刻只允许一个用户访问数据库。这个解决方案的唯一问题是，会延长对其他用户的响应时间。银行柜员机系统或飞机订票系统同时会有数以万计的用户等待访问系统，你可以认真地想象一下，如果这些系统采用这一方案，会是怎样的一幅场景。

### 2.3.1 三种现象

如果只是对数据库执行查询操作，那么 ACID 属性永远都能满足。问题是，当多个事务想同时修改数据库时，在 SQL 模型中，事务可以通过三种方式影响其他事务。<sup>①</sup>

□ **P0 (脏写)** 事务 T1 修改了一项数据，然后事务 T2 在 T1 执行提交或回滚操作之前也对该数据进行了修改。如果 T1 或 T2 之后执行了回滚操作，那么哪个数据是正确的不能确定的。脏写之所以不好，原因之一是违背了数据库的一致性原则。假设在  $x$  和  $y$  之间存在一个约束条件（如  $x=y$ ），并且 T1 和 T2 在独立运行的情况下都能确保约束条件的一致性。然而，当这两个事务写  $x$  和  $y$  数据的顺序不同时，很容易发生违背约束条件的事情，这些只发生在存在脏写的情况下。

□ **P1 (脏读)** 事务 T1 修改了一行，事务 T2 在 T1 提交事务之前读取了该行数据，如果 T1 之后执行了回滚操作，那么 T2 读取的将是永远不被提交的行，这一行数据是“从未存在”的数据。

□ **P2 (不可重复读)** 事务 T1 读取了一行，事务 T2 之后修改或删除了该行数据并提交了操作。如果 T1 试图再读取这一行数据，那么它将获取已经修改了的数据，或者发现这一数据已不存在。

□ **P3 (幻象)** 事务 T1 读取了满足某些查询条件的行集  $N$ 。事务 T2 之后执行语句，生成了一行或多行满足事务 T1 的查询条件的行。如果事务 T1 之后重复之前查询满足特定查询条件的读操作，那么将得到不同的行集。

□ **P4 (更新丢失)** 更新丢失的异常情况发生在以下场景，当事务 T1 读取一个数据项后，事务 T2 更新该数据项（也许基于 T2 前一次读到的数据），再之后 T1 更新该数据项（基于 T1 之前的读操作）并提交操作。

这些现象并不总是不好的。如果数据库只是用于查询，在工作日中不需要进行任何修改，那么上述问题通通不会发生。如果不需要防止这些问题的发生，那么数据库系统运行起来将会快很多。如果只在特定的情况下执行更新，那么这些问题也是可以接受的。

假设有一张表记录了世界上的所有汽车信息。若我想查询开红色跑车的司机的平均年龄，这个查询操作会执行一段时间，在这段时间之内，有的汽车会被撞毁或被买卖，与此同时，也有新的汽车不断被制造出来。不过在查询运行期间，即使上述三类数据库现象存在，平均年龄

<sup>①</sup> 标题提及的三种现象是 P1、P2 和 P3，之后出现的三种现象指的也是这三种。——译者注

也不会有太大的变化，这三类现象造成的小数点两位后的数据变化可以忽略不计。

不过，我们绝不希望当夫妇中的一位往联名账户中存款，而另一位进行取款的时候发生任何上述事件，这也迫使事务隔离级别的产生。

最初的 ANSI 模型仅仅记录了 P1、P2 和 P3 这三类现象，在编号为 MSR-TR-95-21 的微软研究技术报告“A Critique of ANSI SQL Isolation Levels”中，首次出现了其他两类现象的定义。这个报告是由 Hal Berenson、Phil Bernstein、Jim Gray、Jim Melton、Elizabeth O’Neil 和 Patrick O’Neil 于 1995 年发表的。

### 2.3.2 隔离级别

在标准 SQL 中，用户在他所处的会话中设置隔离级别，隔离级别能够避免我们前面提及的几类现象，并为数据库提供一些其他的信息。〈设置隔离级别语句〉语法如下所示：

```
SET TRANSACTION < 事务模式列表 >
< 事务模式 > ::=
    < 隔离级别 >
    | < 事务访问模式 >
    | < 诊断范围大小 >
< 诊断范围大小 > ::= DIAGNOSTICS SIZE < 条件数 >
< 事务访问模式 > ::= READ ONLY | READ WRITE
< 隔离级别 > ::= ISOLATION LEVEL < 隔离级别类型 >
< 隔离级别类型 > ::=
    READ UNCOMMITTED
    | READ COMMITTED
    | REPEATABLE READ
    | SERIALIZABLE
```

〈诊断范围大小〉选项通知数据库建立指定大小的错误信息列表。这是一个标准 SQL 特性，所以你可能在特定的产品中找不到该特性。引入该特性的原因是在一条语句中可能会有很多错误，引擎应该找到所有的这些问题，并当用户在宿主程序中调用 GET DIAGNOSTICS 语句时在诊断区域显示这些错误。

〈事务访问模式〉意如其名，READ ONLY 选项意味着这是一条查询语句，SQL 引擎能获知这些，之后便不再做其他处理。而 READ WRITE 选项告诉 SQL 引擎数据行可能会改变，SQL 引擎因此需要观察三种现象。

〈隔离级别〉子句是很重要的子句，绝大多数现代 SQL 产品中都实现了这个特性。事务的隔离级别定义了事务受其他并发事务影响的程度。默认的事务隔离级别是 SERIALIZABLE，用户可以通过〈设置事务语句〉显式地修改隔离级别。

每个隔离级别都确保了事务要么全部执行完、要么全不执行，并且没有更新会丢失。当 SQL 引擎发现不能保证多条并发事务的可串行性或检测到不可恢复的错误时，引擎自身会启动 ROLLBACK WORK 语句。

让我们看看隔离级别和三类现象表（参见表 2-1），表中的“是”表示在当前隔离级别下，

可能会发生对应的现象。

表 2-1 隔离级别和三类现象

隔离级别	P1	P2	P3
可串行化 (SERIALIZABLE)	否	否	否
可重复读 (REPEATABLE)	否	否	是
提交读 (READ COMMITTED)	否	是	是
未提交读 (READ UNCOMMITTED)	是	是	是

串行化隔离级别保证了并行事务的运行结果与这些事务按照一定的次序依次执行的结果完全一致。串行化执行是指，在前一个事务没有执行结束的情况下，后一个事务不能开始。该级别下的用户就好像排成一对，等待着对数据库的完全访问。

可重复读隔离级别确保了每个用户在其会话中使用的是同一个数据库映像。

在提交读隔离级别中，运行于该会话中的事务能在会话运行时看到其他事务提交的行数据。

在未提交读隔离级别中，运行于该会话中的事务能在会话运行时看到其他事务创建但尚未提交的行数据。

即使不考虑隔离级别，现象 P1、P2、P3 也不应该在以下操作引起的对模式定义的隐式读操作中出现：执行 SQL 语句；完整性约束检查；与引用约束相关的引用操作。我们并不希望模式本身因用户而改变。

### 游标固定隔离级别

游标固定隔离级别 (CURSOR STABILITY isolation level) 为 SQL 游标引入了“提交读”使用的锁机制。当执行 FETCH 游标操作时，需要额外执行读操作，以申请在游标对象上的锁资源。当游标被移除或关闭（可能由提交操作导致），这个锁才会被释放。当然，获得游标的事务可以更新行，即使在随后的 FETCH 操作中游标移到了其他位置，只要事务尚未提交，该事务仍会持有这些行的写锁，这也使游标固定隔离级别强于提交读，弱于可重复读隔离级别。<sup>①</sup>

SQL 系统普遍实现了游标固定隔离级别，以防止游标读取行时发生丢失更新的现象。在一些系统中，提交读是游标固定的增强。ANSI 标准允许这种实现。

SQL 标准没有表示如何实现这些隔离结果。不过有两类基本的并发控制思想——乐观式并发控制和保守式并发控制。在这两类并发控制范畴之内，各个厂商给出了自己的实现。

## 2.4 保守式并发控制

保守式并发控制建立在这样的观点之上：事务往往都会与其他事务发生冲突，所以我们需要设计一个预防问题产生的系统。

所有的保守式并发控制都使用锁机制。锁是数据库中的一个标志位，标记了用户对模式对象的独占访问。想象一下飞机厕所的大门，以及大门上的“使用中”标志。

<sup>①</sup> 这里是从并行安全性角度进行比较得出来的结论。——译者注

但是同样地，我们会发现其他类型的锁模式，例如 z/OS 平台的 DB2 使用门模式，这种模式与传统的锁模式有一点差异。其中重要的差异在于使用的锁级别，而设置开关标志既耗时间又耗资源。如果对整个数据库加锁，那么得到的就是一个串行批处理系统，因为在每一个时刻只有一个事务处于激活状态。在实际应用中只会执行系统维护工作时加这种锁。如果针对表级进行加锁，性能会由于用户必须等待常用表变为可用状态而大受影响。然而，也有一些事务对整个表进行操作，这些事务也只使用一个标志<sup>①</sup>。

如果对表加行级锁，那么其他用户可以访问表的剩余部分，这样就能获得对这张表的最好的共享访问。与此同时，由于存在数量众多的标志需要处理，系统性能会很糟糕。这种行级锁的方法通常并不实用。

页级锁介于表级锁和行级锁之间。页级锁为某张表中包含期望值的行子集加锁。由于数据表存储单元往往是由物理磁盘存储中的页实现的，因此把这类锁叫页级锁。页级锁的性能取决于数据在物理存储中的统计分布，不过通常都会有很好的折中。

## 2.5 快照隔离与乐观式并发

乐观式并发控制建立在这样的理论之上：事务通常不会与其他事务发生冲突，所以我们需要建立一个把事务冲突当做异常进行处理的系统。

在快照隔离中，每个事务读取的数据，是事务开始时该（已提交的）数据的快照，这个时间点我们记做开始时间戳或“t-0”，这个时间可以是事务执行第一次读操作前的任何时刻。由于事务访问的数据都是对该数据的私有副本，因此快照隔离中的事务执行读操作时永远不会被阻塞住。但这也意味着在任何时刻，每个数据项可能都会存在多个版本，这些版本是由当前事务或已经提交的事务所创建。

当事务 T1 准备提交时，将获得提交时间戳，这个时间戳比现有的开始时间戳和提交时间戳都要晚些。事务 T1 只有在以下情况下才能成功提交：没有任何事务的提交时间戳在 T1 的执行间隔期间 [ 开始时间戳, 提交时间戳 ] 内与 T1 对同一数据执行写操作。否则，T1 将回滚。这种“第一个提交者提交成功”的策略避免了丢失更新问题（现象 P4）。当 T1 提交时，它所做的变化对于所有起始时间戳大于 T1 提交时间戳的事务都是可见的。

因为事务的读写操作发生的时间点不同，所以快照隔离是非串行化的。假设有很多事务在处理相同的数据，系统同时满足约束条件“(x+y) 为正数”，每个事务对 x 值和 y 值的修改都要满足约束条件。尽管事务 T1 和 T2 在单独孤立运行时都表现正常，但当两个事务合在一起运行时将违背约束条件。可能的问题是：

□ **A5（数据项约束冲突）** 假设约束 C 是关于数据库数据项 x 和 y 的数据库约束条件。下面是两类可能出现的约束冲突异常。

□ **A5A 读偏离** 假设事务 T1 读取 x 的数据，之后事务 T2 更新了 x 和 y 的数据并提交操作。如果此时 T1 读取 y 的数据，可能会获取到不一致状态的数据，并因此产生不一致状态的输出。

<sup>①</sup> 这里指的是表级锁标志。——译者注



□ **A5B 写偏离** 假设 T1 读取  $x$  和  $y$  值，这些值满足约束条件  $C$ ，之后事务 T2 读取  $x$  和  $y$  值，对  $x$  值进行了修改，然后提交了事务。在此之后 T1 更新了  $y$  值。如果在  $x$  和  $y$  之间存在约束条件，那么这个操作很可能会违背该约束条件。

模糊读 (P2) 是读偏离在  $x=y$  情况下的简化。更常见的情况是一个事务读取两个不同但却有关联的数据 (如引用完整性)。

在银行系统中，只要储户常用账户余额总数不为负数，而账户的余额允许为负数，银行的约束条件就可能会导致写偏离 (A5B) 现象，产生 H5 所描述的异常现象。<sup>①</sup>

只有在以下情况下 A5A 和 A5B 异常才会产生：事务 T1 读取数据，之后事务 T2 在 T1 提交之前对数据进行了修改。所以只要杜绝了 P2 (不可重复读) 现象，A5A 和 A5B 异常都不会产生。这也说明了只有当隔离级别低于可重复读 (REPEATABLE READ) 的情况下，才会产生 A5A 和 A5B 现象。

在 ANSI SQL 对可重复读的定义中，在其严格解释中虽然提及了可重复读是行约束条件的退化形式这一特性，但却遗漏了可重复读的一般概念。具体地说，表 2<sup>②</sup> 描述的基于锁的可重复读隔离级别有效地防止了行级约束冲突，而表 1<sup>③</sup> 描述的 ANSI SQL 定义，只避免了 A1 (脏读) 和 A2 (模糊读) 异常，却没有防止行级约束冲突。

重新回到快照隔离级别的话题上，与读提交 (READ COMMITTED) 相比，快照隔离强得出奇。<sup>④</sup>

在数据库产生前几十年，事务冲突的处理方法就已经存在了。当公司的中心数据部门开始往缩微胶卷中存储数据时，就会手工地运行该方法。你不需要获得缩微胶卷，不过中心数据部门会为你制作盖有时间戳的复印件，你只需把这个复印件放到办公桌上，对复印件进行修改，然后返还给中心数据部门。中心数据职员会检查你更新文件的时间戳，对复印件进行拍照，并将照片放到缩微胶卷的卷尾。

但是假如另一用户也来到了中心数据部门，并携带了相同文件的复印件 (盖有时间戳)，将会发生什么事？中心数据职员必须检查两个文件的时间戳然后再做决定。如果第一个用户试图对文件进行更新，而第二个用户此时正在使用其持有的文件副本，那么中心数据职员或者保存第一份的副本，等待第二个用户返回他的副本，或者会直接返回职员 1 的副本。拿到这两份副本之后，该职员把它们叠在一起，放到灯下，检查副本之间是否存在冲突。如果两个副本的更新都能够写入数据库则执行此操作，如果存在冲突，那么该职员要么采用特定的规则解决冲突问题，要么同时拒绝这两个事务。这可以看做一种在事后进行的行级锁。

① H5 现象可以通过以下事件链表示：r1[x=50] r1[y=50] r2[x=50] r2[y=50] w1[y=40] w2[x=40] c1 c2，其中  $r$  表示读操作， $w$  表示写操作， $c$  表示提交事务，而  $a$  表示回滚事务。——译者注

② 请参考“A Critique of ANSI SQL Isolation Levels”一文中的表 2。——译者注

③ 请参考“A Critique of ANSI SQL Isolation Levels”一文中的表 1。——译者注

④ 论点：可提交读 (READ COMMITTED) « 快照隔离级别 (Snapshot Isolation)。

论证：在快照隔离级别中，第一个提交者赢的策略排除了 P0 (脏写)，而时间戳机制也防止了 P1 (脏读)，因此快照隔离不比可提交读差，除此之外，在可提交读机制下，可能会产生 A5A 现象，但在快照级别时间戳机制下不会产生，因此得出该结论。——译者注

## 2.6 逻辑并发控制

逻辑并发控制（logical concurrency control）建立在这样一个观点之上：计算机能够对等待执行的查询队列进行谓词分析和逻辑处理，以判定数据库允许哪些语句同时执行。

很明显，SELECT 语句并不会修改数据，因此都能同时执行。不过执行完之后，判定哪些语句与其他语句之间存在冲突是件比较棘手的事情。例如，因为主键外键约束，针对不同表进行更新操作的两条 SQL 语句也许只被允许按特定的顺序执行。而对同一张表执行更新的两个语句有可能不被允许，因为它们修改的是表中相同的行，会给行带来不同的最终状态。

然而，第三对针对同一张表执行更新的语句却有可能被允许执行，因为这两个语句修改的是表的不同行，与对方并无冲突。

除此之外，还存在一个问题，语句在等待执行时可能耗费过多的时间。这就是下一节将谈论的活锁的一种。通常的解决方案是为每个等待了一定时间的事务设置优先级数并按等待时长递减，这样每个事务都能达到优先级 1，并在其他事务之前运行。

这个方法也允许设置事务优先级，使其高于队列中其他事务的优先级。尽管这有可能造成活锁，但这并不是问题，这个方法还能终止不重要作业，以有利于更重要作业的执行。例如相较于玩跳棋，打印工资支票就是比较重要的作业。

## 2.7 死锁与活锁

除了硬件错误之外，还有其他事件会导致事务执行失败。死锁就是其中一种事件，当多个用户互相持有了对方申请的资源，却不愿意释放自己持有的资源锁，就会导致死锁。为了更清楚地说明死锁，假设用户 A 和用户 B 需要访问表 X 和 Y。用户 A 获得了表 X 的资源锁，用户 B 则锁住了表 Y，它们都停下来等待自己需要的资源被释放，但这些资源永远不会被释放。死锁的通常解决方法是，数据库管理员（DBA）结束死锁相关的一个或多个会话，并回滚会话所做的工作。

活锁包含了这样的场景：一位用户等待资源，但其他一些用户总在该用户得机之前牢牢地抓取了该资源，持有资源的用户并不处于死锁状态，但作为一个整体，这些用户永远不会释放资源。为了更清楚地说明活锁，我们假设用户 A 需要访问表 X 的所有行，但该表一直被上百个其他用户进行更新操作。因为用户必须对该表所在的所有页加锁才能访问，用户等待所有的页都可用，但这永远不会发生。

在一些系统里，数据库管理员（DBA）可以结束一个或多个相关会话，并回滚这些会话所做的工作，以结束活锁状态。在一些系统里，DBA 可以提高活锁会话的优先级，以使该会话有机会获取资源，来解决活锁问题。

这些知识都很重要，每个数据库系统都有自身的事务处理和并发控制的方式。懂得这些知识是数据库管理员的工作职责，应用程序开发人员则不需要太过关注，不过了解一些幕后的知识总是好的。

# 数据库模式对象

# 3

尽管大多数数据库操作是基于表进行的，但数据库并不仅仅是数据表的集合。一个完整的数据库除了包括表之外，还包括了存储过程、自定义函数以及游标，这些都是用户创建的。此外还有索引和其他一些方法，而这些是用户不能直接使用的。

本章将对数据库用户可创建的模式对象进行概述。SQL 标准将数据库用户划分为普通用户（USER）和管理员用户（ADMIN），模式对象的创建、修改、删除操作需要管理员权限，而普通用户则能够调用及查看模式对象。

## 3.1 CREATE SCHEMA 语句

SQL 标准定义了 CREATE SCHEMA 语句，通过该语句能够一次性创建一个完整的数据库模式。事实上，几乎每款数据库产品都提供了辅助程序，帮助用户分配物理存储，并创建数据库模式，这些辅助程序通常具有较大的差异，其中一些使用专门的语法来进行物理存储的分配。

数据库模式必须包含名称和默认字符集（通常是 ASCII 字符集或 ISO 标准定义的简单拉丁字符集）。CREATE SCHEMA 语句中提供了一个可选的“授权”子句，为模式创建了一个访问控制所需的授权标识符。模式创建完成之后，就是一组模式元素的集合：

```
< 模式元素 > ::=  
    < 域定义 > | < 数据表定义 > | < 视图定义 >  
    | < 授权语句 > | < 断言定义 >  
    | < 字符集定义 >  
    | < 排序规则定义 > | < 翻译定义 >
```

在 SQL 数据库中，模式就好比是数据库的骨架，它定义了模式对象的结构以及操作规则，而模式中的各类数据就好比是依附在骨骼之上的皮肉。

表（table）是 SQL 语言中唯一的数据结构。SQL 中的表可以是持久化的（基础表），可以用于临时存储（临时表），可以是虚拟表（视图、公用表表达式和衍生表），在需要时甚至可以是物化的。以上这些类型在性能上没有差异，但在实现上是不同的。SQL 语言只使用一种数据结构（表），这大大简化了对于 SQL 的处理。由于所有操作的返回类型也都是表，因此无需对返回结果进行结构转换、编写特殊的操作符或是处理语言中的不规则问题。

< 授权语句 > 限制用户只能访问已授权的模式元素。断言可以被看做应用于整个模式的约

束条件, < 断言定义 > 目前并没有被广泛使用。而 < 字符集定义 >、< 排序规则定义 > 和 < 翻译定义 > 则用于处理数据的显示。开发人员和普通用户并不需要关心上述模式对象, 也不需要对这些对象进行设置, DBA (数据库管理员) 将负责这些对象的设置和维护工作。

## CREATE TABLE 和 CREATE VIEW 语句

表和视图是 SQL 操作的基本单位, 我们将在以后专门设章进行讲解。

## 3.2 CREATE PROCEDURE、CREATE FUNCTION 以及 CREATE TRIGGER 语句

过程构造语句将使用 SQL/PSM (持久化存储模块技术) 或其他语言编写的过程代码模块写入数据库中。这些构造语句可以按需调用, 且我们将在后面专门设章进行讲解。

## 3.3 CREATE DOMAIN 语句

域是标准 SQL 中的一种模式元素, 可用来声明内联宏, 从而在模式中放入常用的列定义。域相关的 SQL 语句语法如下所示:

```
< 创建域 > ::=
CREATE DOMAIN < 域名 > [AS] < 数据类型 >
    [< 默认子句 >]
    [< 域约束 >...]
    [< 排序规则子句 >]

< 域约束 > ::=
    [< 约束名定义 >]
< check 约束定义 > [< 约束属性 >]

< 修改域 > ::=
    ALTER DOMAIN < 域名 > < 修改域动作 >

< 修改域动作 > ::=
    < 修改域默认值子句 >
| < 删除域默认值子句 >
| < 添加域约束定义 >
| < 删除域约束定义 >
```

需要重点注意, 域只能声明为基本数据类型, 用其他域类型声明域是不允许的。域声明成功后, 将取代相关数据类型, 用于声明相关的数据列。

在 CHECK() 子句中, 我们可以写入域数据检验代码, 对数字值、数据范围、列表以及其他各类情况进行检验, 下面这段代码框架概要地定义了美国各州编码域。

```
CREATE DOMAIN StateCode AS CHAR(2)
DEFAULT '??'
CONSTRAINT valid_state_code
```

```
CHECK (VALUE IN ('AL', 'AK', 'AZ', ...));
```

如果在一处定义了域，就不需要担心使用该域声明的列数据是否正确。如果不使用 DOMAIN 子句，那么便无法避免地需在数据库的多张表中对列定义复制 CHECK() 子句。用户可以通过 ALTER DOMAIN 和 DROP DOMAIN 语句编辑或删除域。

## 3.4 创建序列

序列可以理解为数值序列生成器。序列的调用方式与函数调用一致，每次调用后，将返回序列中的下一数据。

在我之前的书籍中，我使用表“Sequence”存储从 1 到  $n$  的整数集合，由于 Sequence 已经被设置为 SQL 保留字，因此在本书中我改用“Series”表名。创建序列的语法如下所示：

```
CREATE SEQUENCE <序列名> AS <数据类型>
START WITH <起始值>
INCREMENT BY <递增值>
[MAXVALUE <最大值>]
[MINVALUE <最小值>]
[[NO] CYCLE];
```

如果要获取序列中的值，则需要调用以下语句：

```
NEXT VALUE FOR <序列名>
```

在某些情况下，我们需要对序列进行重置，此时需要使用以下语句修改可选子句或重置序列计数：

```
ALTER SEQUENCE <序列名>
RESTART WITH <起始值>; -- 重新计数
```

调用以下语句，可以删除现有序列：

```
DROP SEQUENCE <序列名>;
```

尽管序列作为数据库的一类特性正在被越来越多的数据库产品所接受，但这一现象应该避免，因为序列属于非关系化（nonrelational）的一种扩展，它的使用方式并不是基于集合的操作方式，而是更接近于序列文件或过程函数的使用方式。可以在 Oracle、DB2、Postgres 以及 Mimer 等数据库产品中使用序列。

## 3.5 创建断言

在标准 SQL 中，CREATE ASSERTION 语句允许创建一类能够应用于指定模式下所有表的约束条件，创建断言的语法如下所示：

```
<断言定义> ::=
CREATE ASSERTION <约束名称> <断言检查>
[<约束属性>]
```

```
<断言检查> ::=  
CHECK (<查询条件>)
```

很容易想到,用户可以通过 `DROP ASSERTION` 语句删除断言,不过并不存在 `ALTER ASSERTION` 语句。由于在断言中可以使用模式下的所有表,因此其功能强于单个表的 `CHECK()` 约束。对于空表而言, `CHECK()` 检测返回值为 `TRUE`。

如下所示,在不使用断言的情况下,很难实现这样的一条检验规则,即公司的员工总人数必须与健康计划表中员工总数吻合:

```
CREATE ASSERTION Total_Health_Coverage  
CHECK (SELECT COUNT(*) FROM Personnel) =  
      + (SELECT COUNT(*) FROM HealthPlan_1)  
      + (SELECT COUNT(*) FROM HealthPlan_2)  
      + (SELECT COUNT(*) FROM HealthPlan_3);
```

由于 `CREATE ASSERTION` 作用于整个模式,针对数据表的约束名也是作用于模式全局,而不是仅仅作用于约束声明对应的数据表。

### 3.5.1 为模式级约束使用视图

如果不使用断言,那么必须使用过程和触发器才能产生断言的效果。如下所示,下面的连锁店铺数据库模式中包含了三张数据表:

```
CREATE TABLE Stores  
(store_nbr INTEGER NOT NULL PRIMARY KEY,  
 store_name CHAR(35) NOT NULL,  
 ...);  
  
CREATE TABLE Personnel  
(emp_id CHAR(9) NOT NULL PRIMARY KEY,  
 last_name CHAR(15) NOT NULL,  
 first_name CHAR(15) NOT NULL,  
 ...);
```

这两张表定义了店铺和员工的相关信息,下面这张表用于记录店铺和员工的对应关系,详细描述了雇员在哪家店铺工作、从事什么工作以及开始工作的时间信息:

```
CREATE TABLE JobAssignments  
(store_nbr INTEGER NOT NULL  
  REFERENCES Stores (store_nbr)  
  ON UPDATE CASCADE  
  ON DELETE CASCADE,  
 emp_id CHAR(9) NOT NULL PRIMARY KEY  
  REFERENCES Personnel(emp_id)  
  ON UPDATE CASCADE  
  ON DELETE CASCADE,  
 start_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
```

```

end_date TIMESTAMP,
CHECK (start_date <= end_date),
job_type INTEGER DEFAULT 0 NOT NULL -- unassigned = 0
CHECK (job_type BETWEEN 0 AND 99),
PRIMARY KEY (store_nbr, emp_id, start_date));

```

我们引入了 `job_type` 字段，用于描述员工所从事的工作类型，`0` 表示没有安排工作，`1` 表示负责打包的小工，依次类推，`99` 表示店铺经理。与此同时，我们规定每个店铺只能有一个经理，在标准 SQL 中，我们需要编写以下约束来确保店铺经理的唯一性：

```

CREATE ASSERTION ManagerVerification
CHECK (1 <= ALL (SELECT COUNT(*)
FROM JobAssignments
WHERE job_type = 99
GROUP BY store_nbr));

```

这个语句实际上比看起来更微妙些，如果将 `<=` 修改成 `=`，则商店里只要有雇员的话，就必须有一个经理。

不过就像我们所说的那样，绝大多数的 SQL 产品仍不允许对整张表应用 `CHECK()` 约束，这些产品同时也不支持模式级 `CREATE ASSERTION` 语句。

那么如何做到对整张表应用 `CHECK()` 约束呢？可以使用触发器，触发器需要使用厂商专用的程序语言来实现。尽管在 SQL/PSM 标准中定义了触发器，大多数厂商实现的触发器模型都有很大差别，在触发器的主体部分也都使用了特有的 4GL 语言。

我们需要一组触发器，以对 `INSERT` 和 `UPDATE` 操作后的表状态进行检验。如果删除了一个雇员，并不会使店铺经理的数量超过 1。`INSERT` 和 `UPDATE` 操作相关的触发器的主干部分是这样的：

```

CREATE TRIGGER CheckManagers
AFTER UPDATE ON JobAssignments -- INSERT 也是一样的
IF 1 <= ALL (SELECT COUNT(*)
FROM JobAssignments
WHERE job_type = 99
GROUP BY store_nbr)
THEN ROLLBACK;
ELSE COMMIT;
END IF;

```

作为一个数据库狂热者，假设我希望能够有一个纯 SQL 解决方案，该方案采用声明式的方式解决这个问题，以避免只能使用现代 SQL 产品的限制。

创建两张表，第一张表是仅仅存储店铺经理信息的人员表，这张表以员工身份号作为主键。需要注意的是在 `job_type` 列使用默认值 (`DEFAULT`) 和约束检查 (`CHECK()`)，以确保这张表只存储了经理信息。

```

CREATE TABLE Job_99_Assignments

```

```

(store_nbr INTEGER NOT NULL PRIMARY KEY
    REFERENCES Stores (store_nbr)
    ON UPDATE CASCADE
    ON DELETE CASCADE,
emp_id CHAR(9) NOT NULL
    REFERENCES Personnel (emp_id)
    ON UPDATE CASCADE
    ON DELETE CASCADE,
start_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
end_date TIMESTAMP,
CHECK (start_date <= end_date),
job_type INTEGER DEFAULT 99 NOT NULL
    CHECK (job_type = 99));

```

第二张表是存储了店铺里面除经理之外的所有雇员信息的人员表，这张表也将员工身份号作为主键。需要注意的是在 `job_type` 列使用默认值（`DEFAULT`）和约束检查（`CHECK()`），以确保这张表没有存储经理信息：

```

CREATE TABLE Job_not99_Assignments
(store_nbr INTEGER NOT NULL
    REFERENCES Stores (store_nbr)
    ON UPDATE CASCADE
    ON DELETE CASCADE,
emp_id CHAR(9) NOT NULL PRIMARY KEY
    REFERENCES Personnel (emp_id)
    ON UPDATE CASCADE
    ON DELETE CASCADE,
start_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
end_date TIMESTAMP,
CHECK (start_date <= end_date),
job_type INTEGER DEFAULT 0 NOT NULL
    CHECK (job_type BETWEEN 0 AND 98) -- 并不包含编号 99
);

```

从这两张表出发，创建一个包含了公司所有的工作安排信息的合并（`UNION-ed`）视图，把这些信息展现给用户：

```

CREATE VIEW JobAssignments (store_nbr, emp_id, start_date,
    end_date, job_type)
AS
(SELECT store_nbr, emp_id, start_date, end_date, job_type
    FROM Job_not99_Assignments
UNION ALL
SELECT store_nbr, emp_id, start_date, end_date, job_type
    FROM Job_99_Assignments)

```



这两张表中主键和 `job_type` 的约束条件共同确保了每个店铺至多只能有一个经理。下一步是为视图增加前触发器 (`INSTEAD OF trigger`) 或编写存储过程，以方便用户执行插入、更新以及删除操作。下面是一个忽略了错误处理和输入验证的简单的存储过程：

```
CREATE PROCEDURE InsertJobAssignments
(IN store_nbr INTEGER, IN new_emp_id CHAR(9), IN new_start_
    date DATE, IN new_end_date DATE, IN new_job_type INTEGER)
LANGUAGE SQL
IF new_job_type <> 99
THEN INSERT INTO Job_not99_Assignments
    VALUES (store_nbr, new_emp_id, new_start_date,
        new_end_date, new_job_type);
ELSE INSERT INTO Job_99_Assignments
    VALUES (store_nbr, new_emp_id, new_start_date,
        new_end_date, new_job_type);
END IF;
```

类似地，解雇员工的函数如下所示：

```
CREATE PROCEDURE FireEmployee (IN new_emp_id CHAR(9))
LANGUAGE SQL
IF new_job_type <> 99
THEN DELETE FROM Job_not99_Assignments
    WHERE emp_id = new_emp_id;
ELSE DELETE FROM Job_99_Assignments
    WHERE emp_id = new_emp_id;
END IF;
```

如果开发人员试图使用 `INSERT`、`UPDATE` 或 `DELETE` 直接对 `Job_Assignments` 视图进行修改，将会收到错误信息：提示视图包含了 `UNION` 操作，无法更新。从一方面看这是件好事，因为开发人员被强迫使用存储过程。

再次重复一遍，这是在特定限制下的一种编程方案练习。与使用视图方案相比，使用触发器也许能获得更好的性能。

### 3.5.2 为约束使用主键和断言

下面给出了上一节提及的“仓库和员工”问题的另一个实现版本。

```
CREATE TABLE JobAssignments
(emp_id CHAR(9) NOT NULL PRIMARY KEY -- 没有员工同时受雇于两家店铺
REFERENCES Personnel (emp_id)
ON UPDATE CASCADE
ON DELETE CASCADE,
store_nbr INTEGER NOT NULL
REFERENCES Stores (store_nbr)
ON UPDATE CASCADE
```

```
ON DELETE CASCADE);
```

在 `emp_id`（社会保险号）上创建的主键确保了没有人同时在两个店铺工作，而每个店铺可以拥有多个员工。理想情况下，需要一个约束来检查是否为每个员工都安排了工作。

下面是对该问题的第一种尝试：

```
CREATE ASSERTION Nobody_Unassigned
CHECK (NOT EXISTS
  (SELECT *
   FROM Personnel AS P
     LEFT OUTER JOIN
     JobAssignments AS J
     ON P.emp_id = J.emp_id
     WHERE J.emp_id IS NULL
    AND P.emp_id
      IN (SELECT emp_id FROM JobAssignments
        UNION
        SELECT emp_id FROM Personnel)));
```

不过，这种方法有点儿过了，而且并没有阻止员工在多个店铺工作。`Personnel` 表和 `JobAssignments` 表也许都在 `emp_id` 列建立了索引，所以调用 `COUNT()` 方法成本较低。下面的断言同样能够起作用：

```
CREATE ASSERTION Everyone_assigned_one_store
CHECK ((SELECT COUNT(emp_id) FROM JobAssignments)
 = (SELECT COUNT(emp_id) FROM Personnel));
```

乍一看，这种实现方式着实让那些期望使用 `JOIN` 来实现员工和职位一对一映射关系的人们吃惊，但是使用 `JOIN` 建立映射引入了主键 - 外键的需求。任何未分配工作的员工都会使 `Personnel` 表比 `JobAssignments` 表大，而 `JobAssignments` 表记录的每个员工在人员表中都有一个对应信息。好的优化器会将语句当做谓词使用，这也是使用 `DRI`（`Declarative Referential Integrity`，声明式引用完整性）替代触发器和应用程序端逻辑的原因。

你将需要使用存储过程，以便在一个事务内将数据同时插入两张表。更新和删除也会造成级联，并清除任务分配信息。

让我们对细则进行一点修改，允许员工同时在多个店铺工作。既然我们想让员工在多个店铺工作，那么需要修改 `JobAssignments` 表中的键，如下所示：

```
CREATE TABLE JobAssignments
(emp_id CHAR(9) NOT NULL
 REFERENCES Personnel (emp_id)
 ON UPDATE CASCADE
 ON DELETE CASCADE,
store_nbr INTEGER NOT NULL
 REFERENCES Stores (store_nbr)
 ON UPDATE CASCADE
```

```
ON DELETE CASCADE,
PRIMARY KEY (emp_id, store_nbr));
```

之后，我们在断言中使用 `COUNT(DISTINCT ...)` 语句：

```
CREATE ASSERTION Everyone_assigned_at_least_once
CHECK ((SELECT COUNT(DISTINCT emp_id) FROM JobAssignments)
= (SELECT COUNT(emp_id) FROM Personnel));
```

需要注意唯一性约束和断言同时出现的情况。当其中一个或两个都发生了变化，就会导致这个规则的变动。

## 3.6 字符集相关结构

有很多处理字符的模式级结构。可以创建一个用于处理多语言或有特殊用途的命名字符集，为这些字符定义一个或多个排序规则序列，并将一个字符集翻译成另一个字符集。

现在，Unicode 标准以及相关厂商特性被普遍使用。大多数的字符都已经有了 Unicode 名称以及相关的排序规则。例如，在 ISO 8859-1 中定义了 SQL 文本使用 Latin-1 字符集。这是个使用在 HTML 上的字符集，包含了拉丁字母中的 191 个字符。这也是美洲、西欧、大洋洲、非洲最常用的字符集，同时也被东亚语作为标准罗马拼音使用。

从 1991 年开始，Unicode 协会、ISO（国际标准化组织）和 IEC（国际电工委员会）协会一起工作，以便制定能同时容纳 Unicode 标准和 ISO/IEC 10646 标准的新标准：UCS（通用字符集）标准。Unicode 和 ISO/IEC 10646 目前确认了大概 100 000 个字符，这些字符的编码空间中已经包含了超过 100 万编码点<sup>①</sup>。Unicode 和 ISO/IEC 10646 还定义了许多通用编码，这些编码能够表示每一个可用的码点。Unicode 和 UCS 标准编码格式可以使用 1 到 4 个八位值进行编码（UTF-8），可以使用 1 到 2 个 16 位值进行编码（UTF-16），还可以使用一个 32 位值（UTF-32 和 UCS-4）。早先还有一个使用 1 个 16 位码值的编码格式（UCS-2），能够表示当前可用码点总数的 1/17。这些编码格式中，只有 UTF-8 字节序列的顺序是固定的，其他编码格式的字节顺序依赖于平台相关的字节顺序，这些字节顺序可以通过特殊代码或其他方式进行修改。

### 3.6.1 创建字符集

许多 SQL 产品中没有创建字符集的相关语法。SQL 厂商默认采用基于本地语言设置的系统级字符集。

```
< 字符集定义语句 > ::=
CREATE CHARACTER SET < 字符集名称 > [AS]
< 字符集来源 > [< 排序规则子句 >]

< 字符集来源 > ::=
GET < 字符集标准 >
```

<sup>①</sup> 根据字符的数目，可以设定整数值的上限，这个整数范围称为编码空间，其中的一个特定整数称为一个码点。——译者注

< 排序规则子句 > 通常都是默认的，不过还可以使用命名排序规则。

### 3.6.2 创建排序规则

```
< 排序规则定义 > ::=
CREATE COLLATION < 排序规则名称 >
    FOR < 字符集标准 >
    FROM < 已知排序规则名 > [< 填充特征 >]

< 填充特征 > ::= NO PAD | PAD SPACE
```

< 填充特征 > 选项与如何比较字符串有关。短字符与长字符的前缀相等，如果比较规则设置了不填充（NO PAD）特性，那么认为短字符小于长字符。如果比较规则设置了填充（PAD SPACE）特性，为了能与长字符进行比较，会在短字符末尾填充足够多的空白字符，使其长度与长字符相同。SQL 数据库通常在短字符后填充空白字符，然后依次对每个字符按位置进行比对。

### 3.6.3 创建翻译

下列语句定义了如何从一个字符集映射到另一个字符集。该语句的重点在于命名了这个映射。

```
< 翻译定义 (transliteration definition) > ::=
CREATE TRANSLATION < 翻译名 >
    FOR < 源字符集规范 >
    TO < 目标字符集规范 >
    FROM < 翻译源 (transliteration source) >

< 源字符集规范 > ::=
< 字符集规范 >

< 目标字符集规范 > ::=
< 字符集规范 >

< 翻译源 > ::=
< 已知翻译源 > | < 翻译例程 >

< 已知翻译源 > ::= < 翻译名 >

< 翻译例程 > ::= < 特定例程标识符 >
```

需要注意的是，这里使用了简单的映射，这些映射像极了使用嵌套的 REPLACE() 函数，同时也像调用了个例程进行处理。为这些直译命名是为了能够使用 TRANSLATE() 函数取代一堆嵌套的 REPLACE() 函数。具体语法非常简单：

```
TRANSLATE (< 字符值表达式 > USING
    < 翻译名 >)
```

DB2 和其他的实现扩大了对 TRANSLATE() 的应用，使其能够对字符串进行处理，因此可以使用一个表达式实现很多的编辑工作。我们将会在讲解字符串函数的时候再接触这些内容。

# 定位数据和特殊数值

# 4

SQL 产品具有能够帮助定位物理存储数据的特性。其中一些特性是从硬件中获取的，另一些则不依赖于硬件。最常用的特性包括序列号、随机数以及具有特殊属性的数学序列。

## 4.1 显式的物理定位器

SQL 语句使用键是理所当然的，键是一个完全脱离了物理存储的逻辑概念。遗憾的是，不好的 SQL 程序员会使用特定功能，以使硬件生成显示的物理定位器。这些生成的数字代表了硬件发生的事件，或在硬件中的位置，对逻辑模型没有任何作用。

不要将显式的物理定位器和代理键（surrogate key）混为一谈，Codd 博士说过“数据库用户能让系统生成、销毁代理键，但这些用户既不能控制这些键值，也不能查看这些值”（见 ACM TODS, 409~410）。可以想想大多数 SQL 实现中索引的工作方式。

### 4.1.1 ROWID 和物理磁盘地址

Oracle 使用了 ROWID 这一特殊变量，用于显示存储在硬盘驱动器中行（row）的物理地址。使用 ROWID 是最快的定位数据表中行的方法，因为读写磁头能够立刻指向行存储地址。Oracle 在逻辑层通过 ROWID 暴露底层物理存储位置，这意味着它在存储表中数据行时使用连续的存储空间。这也表明 Oracle 不能使用散列化、分布式数据库、动态位向量以及其他许多面向 VLDB（Very Large DataBase，超大型数据库）的新技术。假如由于某种原因，数据库位置移动或被重新组织，ROWID 会改变。

### 4.1.2 标识列

标识（IDENTITY）列是数据库引擎为每个新增行自动生成唯一数的一种机制。在需要为表的每个新增行设置唯一标识时，可以为该表添加标识列。为了确保表中每个新增行都具有唯一数值标识，可以在标识列上定义一个唯一索引，也可以将标识列声明为表的主键。表一旦创建，就不能通过修改表的声明方式增加标识列。

如果往数据表中插入行时显式指定了标识列的数值，那么系统就不会更新下一个生成的标识列值，这就可能会导致新增值与表中已有值的冲突。如果通过唯一索引或主键方式确保了标

识列中数值的唯一性，那么标识列上的重复值会导致生成错误消息。下面就是标识列的 BNF 范式形式：

```
<列名> INTEGER NOT NULL GENERATED [ALWAYS | BY DEFAULT]
AS IDENTITY ( START WITH <起始值>, INCREMENT BY <增量值> )
```

输入的第一行的列值为<起始值>，随后依次添加的每一行的关联列值都比上一个列值增加<增量值>。如果定义 IDENTITY 列时指定了 GENERATED ALWAYS，那么 IDENTITY 列值总是由 SQL 引擎生成。应用不能提供显式的值。而标识了 GENERATED BY DEFAULT 的 IDENTITY 列值允许被应用显式赋值。如果应用不为列赋值，那么 SQL 引擎会生成值。由于应用控制值，SQL 引擎不能保证该值的唯一性。GENERATED BY DEFAULT 子句用于数据传播，前提是其目的是复制已有表的内容。该子句也可用于卸载或恢复表。

尽管 IDENTITY 列与序列有相似之处，不过它们之间也有一些不同点。IDENTITY 列具有以下特点。

(1) 只有在表创建时才能将 IDENTITY 列定义为表的组成部分。表一旦创建，就不能为其添加 IDENTITY 列了。(不过可以修改已有 IDENTITY 列。)

(2) Sybase/Microsoft T-SQL 列中会为单个表生成列值。如果使用 GENERATED ALWAYS 定义 IDENTITY 列，那么总是由数据库引擎生成列值。应用在修改表内容时，不能设置该值。ANSI 序列则是独立的对象，可以出于任何用途为任何表生成数值。

IDENTITY 列以在插入过程中暴露机器的部分物理状态为基础，这违背了 Codd 博士关于定义关系型数据库的规则(也就是 Codd's rule #8，物理数据独立性)。这个错误几乎是无法纠正的。

早期的 SQL 产品建立在已有文件系统基础上。数据保存在物理连续的磁盘页上，保存在物理连续的数据行上，并由物理连续的列组成。简单来说，就好像放置穿孔卡片的桌台或磁带。大多数这种自增特性都是尝试重新获得 SQL 取出的物理序列，所以可以假设我们使用的是物理连续的存储。

不过物理连续的存储只是创建关系型数据库的一种方法，而且该方法不总是最好的。不过除此之外，关系型数据库的完整构想是：用户不应该了解事物是如何存储的，他们只需根据具体产品的特定发布版本中的具体物理层表现，编写较直接操作文件系统少得多的代码。

生成标识列的具体方法因产品而异。不过结果却是完全一样的——它们的行为是不可预料且冗余的。如果在表中已经定义了合适的键，标识列就是冗余的。关于插入一个简单整数比插入一个长列是否“成本更低”曾引起争论，所以我们还是使用 IDENTITY 列作为键列。在现代关系型数据库管理系统中，这一论点显示是不正确的。事实上，许多散列算法运行得更好，这些散列算法作用于长的复合键，这使得创建完美散列值变得更容易。

使用自增数作为键的另一个主要缺点是，不能对自增数进行校验位检测，所以无法检验这些值是否有效。(关于校验位检测，可以参考 Joe Celko's *Data and Databases: Concepts in Practice*。)

为什么使用自增数呢？对于如何获取主键这一问题而言，使用系统生成值是一个快速而简单的答案。这个方法既不需要任何研究，也不需要实际数据建模。同样地，药物滥用也是简单快速的治病方法。我对于这两种方法不做评价。

Sybase/SQL Server 族允许使用表的 **IDENTITY** 属性声明一个数值伪列。伪列值是尝试往表中写入物理数据的次数。需要注意“尝试”这一词，插入失败或事务回滚都会在数值编号中产生缺口。**IDENTITY** 属性完全是关系型的，不过经常被不称职或新 SQL 程序员所误用，使得表看起来既像通过记录位置数访问的序列磁带文件，又像是模拟指针访问一个非关系型文件系统或网络数据库管理系统。

下面让我们看看具体会导致的逻辑难题。首先尝试创建一个包含两列的表，并将这两列都标识为 **IDENTITY** 列。如果无法将多个列声明为同一特定数据类型，那么很显然这个类型完全不是数据类型。SQL Server/Sybase 族把 **IDENTITY** 用作表的特性，所以每个表中只允许有一个标识。

下一步，创建一个包含一列的表，并将该列设置为 **IDENTITY** 列。现在从中尝试插入、更新以及删除不同数值。如果无法插入、更新或删除表中的行，那么这个表就完全不是表。

首先，创建包含有一个 **IDENTITY** 列和几个其他列的简单表，然后执行以下语句：

```
BEGIN
INSERT INTO Foobar ( a, b, c ) VALUES ( 'a1', 'b1', 'c1' );
INSERT INTO Foobar ( a, b, c ) VALUES ( 'a2', 'b2', 'c2' );
INSERT INTO Foobar ( a, b, c ) VALUES ( 'a3', 'b3', 'c3' );
END;
```

下面的语句在逻辑上与上述语句完全等效：

```
INSERT INTO Foobar ( a, b, c )
VALUES ( 'a1', 'b1', 'c1' ), ( 'a2', 'b2', 'c2' ), ( 'a3',
    'b3', 'c3' );
```

或者，也可以使用以下语句：

```
INSERT INTO Foobar ( a, b, c )
SELECT x, y, z
FROM Floob; -- 假设 Floob 中有三行数据
```

使用这些语句往表中插入几行数据。注意在第一个程序块中 **IDENTITY** 列会依照显示的顺序依次编号。如果删除某行，那么在序列中出现的间隔不会被填充上，而序列仍会从这张表中该列所用过的最大数值继续。

第二和第三段语句对于行的插入顺序是不受约束的。查询语句的查询结果是一个表，而表是一个无序集合，那么应该如何为标识列编号呢？这些语句一次性地将完整的集合写入到 **Foobar** 表中，而不是一次只写一行。因此，存在  $n!$  种对  $n$  行进行编号的方式。应该选择哪一种编号方式呢？答案是编号顺序与所使用结果集的物理存储顺序相同。用非关系型术语来表示：编号顺序遵循“物理顺序”。

不过现实比这个还要严峻些，如果再次执行相同的查询，而此时统计信息有所变动，或者已经移除或添加索引，那么新的执行计划可能会以不同的物理顺序返回结果集。如何从逻辑模型角度解释为什么在第二次查询中，相同的数据行却返回了不同的标识数值？在关系型模型中，如果所有的属性值都相同，那么这两个行应该被认为是相同行。

下列语句执行之后，数据库应该与执行之前完全一样。该语句在同一事务中删除并重新插

入了相同数据。

```
BEGIN ATOMIC
DELETE FROM Foobar
WHERE identity_col = 41;
INSERT INTO Foobar VALUES ( << 未删除之前第 41 行中的数据 >>);
END;
```

但是上述语句修改了 **IDENTITY**。因为 DML 语句无法修改 **IDENTITY**，所以可以通过 **UPDATE** 语句在两行之间交换列值，完成类似的工作。

请试着思考一下，在两个数据库之间执行复制，这两个数据库之间只是索引结构不同或缓存大小不同，或者存在某些会导致在相同语句中偶尔会造成执行计划不同的差异之处，想试试维护或移植这样的系统吗？

第 3 章已经讨论了 **CREATE SEQUENCE** 构造语句。SQL-2003 标准中采纳了该语句，而该语句与 **IDENTITY** 有所混淆。序列对象有如下属性：

- (1) 序列属于数据库对象，它不和任何一张表绑定；
- (2) 序列生成的顺序值可以被任何 SQL 语句使用。

因为序列对象可以被任何应用使用，所以存在两个表达式控制序列对象，分别用于获取指定序列的下一个数值以及表达式执行之前的序列生成值。**PREVVAL** 表达式返回当前会话中为前面的语句中使用的指定序列最新生成的数值。**NEXTVAL** 表达式返回指定序列的下一个值。使用这些表达式，可以使作用于不同表的多个 SQL 语句使用相同的数值。

尽管上述内容并不是 **PREVVAL** 和 **NEXTVAL** 的所有特性，不过了解这些特性已经足够让我们根据当前的数据库设计以及使用数据库的应用确定应该使用哪个了。

## 4.2 生成的标识符

有很多种生成标识符的方案，这些标识符在任何数据库之间都是唯一的。两种最常用的标识符包括微软的 **GUID**(Global Unique Identifier，全局唯一标识符)和开源社区的 **UUID**(Universal Unique Identifier，通用唯一标识符)。

### 4.2.1 GUID

**GUID** 是由 UTC 时间和所在设备网络地址混合在一起，形成的具有唯一性的显式物理定位器。微软表示 **GUID** 值能够在大约一个世纪内保持唯一。下面的表述来自维基百科相关页面(<http://en.wikipedia.org/wiki/GUID>)。

用于生成新的 **GUID** 值的算法已被广泛批评。一方面，用户网卡的 **MAC** 地址被作为 **GUID** 数的基数，举例来说，这意味着对于文档能够追溯到创建该文档的计算机。当发现这一现象后，微软修改了该算法，使之不再包含 **MAC** 地址。在定位“梅丽莎蠕虫”病毒作者位置时使用了该隐私漏洞。

除了具有显式物理定位器的常见问题，存储每个 **GUID** 值需要 16 字节，而在大多数机器上



存储一个简单的整数只需要 4 字节。

使用 GUID 构造的索引和主键在性能上不如较短的键列 (key column)。之所以提到这些,是因为许多新手辩称使用 GUID 键能提高性能。这些辩解是错误的,而且在现在的硬件条件下,这种性能级别并不是真正的问题。硬件为 64 位的计算机越来越常见,正如磁盘驱动器的速度越来越快。

真正的问题在于很难对 GUID 值进行解释,所以出于验证的目的直接处理 GUID 并追踪到它们所在的数据源很困难。事实上, GUID 并没有排序顺序,所以无法注意到丢失了数据,也不能使用它们对结果排序。我们能做的就是使用包含了正则表达式的 CHECK() 方法对这类字符串进行验证,其中字符串是由数字和 A~F 字母组成并使用 4 个破折号来分隔的 36 个字节字符串。

包含聚合函数的查询中不能包含 GUID;首先需要将其类型转化为 CHAR(36) 并使用字符串值。你首先想到的可能是将其转化成长整型,但这两种数据类型并不兼容。GUID 数据类型的其他特性是非常专有的,无法迁移出微软环境。

## 4.2.2 UUID

UUID 是 OSF (Open Software Foundation, 开放软件基金会) 所指定的标准,是 DCE (Distributed Computing Environment, 分布式计算环境) 的组成部分。UUID 的设计目的是促使分布式系统不需要专门的中心协调,就能唯一标识信息。

UUID 作为标准的一部分,记录在 ISO/IEC 11578:1996 标准“信息技术—开放系统互连—远程过程调用 (RPC)”中,而最近也出现在 ITU-T Rec. X.667 | ISO/IEC 9834-8:2005 标准中。互联网工程任务组 (IETF) 发布了提议标准 RFC 4122,从技术角度看,该标准等同于 ITU-T Rec. X.667 | ISO/IEC 9834-8 标准。

目前存在五个版本的 UUID 我们完全可以用 UUID 标识某事物,而不必担心标识符会被任何人无意中用于标识其他事物。UUID 是一个 16 字节 (128 位) 数,包含了 32 个十六进制数字,这些数被连字号分隔为五组来显示,其表示形式为 8-4-4-4-12<sup>①</sup>,总共有 36 个字符 (32 个数字和 4 个连字号)。

UUID 第一个版本的生成方案使用了生成机器的 MAC 地址、纳秒时钟周期以及一些数字运算。这一方案既暴露了生成 UUID 的机器的标识,又暴露了生成时间。

第二个版本的 UUID 与第一版很相似,使用公式对 POSIX 的用户 ID (UID) 和 POSIX 的用户组 ID 域 (GID domain) 进行处理。

第三版的 UUID 使用 MD5 将 URL、完全限定域名、对象标识符以及其他数据元素转化成 UUID。MD5 (Message-Digest algorithm 5, 信息摘要算法 5) 是一个应用广泛的生成 128 位散列值的加密散列函数,MD5 算法已经列入了因特网规范 (RFC 1321)。从 2004 年开始,研究人员发现了 MD5 越来越多的问题。今天,美国国土安全部表示 MD5 “应视为可破译的算法,不适用于在未来使用”,并要求 2010 年之前大多数美国部门的应用迁移到 SHA-2 族散列算法中。第三版的 UUID 是格式为 xxxxxxxx-xxxx-3xxx-yxxx-xxxxxxxxxxxx 的十六进制字符串,而 y 则允许

<sup>①</sup> 数字表示该组包含的十六进制数的数量。——译者注

为十六进制数 8、9、A 或 B。

第四版的 UUID 使用了只依赖于随机数的方案。这个算法既设置两个保留位，也设置版本号，并通过随机或伪随机数据源设置所有其他位的数值。第四版的 UUID 的格式为 xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx，其中的 x 值为十六进制数字，y 为十六进制数字 8、9、A 或 B。

第五版的 UUID 方案使用了 SHA-1 执行散列操作，除此之外与第三版完全一致。RFC 4122 声明第五版优于第三版的基于名称的 UUID。需要注意，为了使 UUID 长度有效，160 位 SHA-1 散列值被截断成 128 位。

下面将让你了解 UUID 产生重复值的概率，如果在未来 100 年里，每秒产生十亿个 UUID 值，那么出现一个重复值的概率接近 50%。需要注意的是，这以 UUID 生成机制“公平行事”并且不会产生错误为前提。

### 4.3 序列生成函数

数据库包含了 COUNTER (\*)、NUMBER(\*)、IDENTITY 等专有特性，每次在表达式中调用这些函数时，将返回一个新生成的递增值。这也是生成唯一标识符的方法之一。这些特性可以是函数调用，也可以是列属性，这取决于具体的产品。这些特性同时也是讨厌的、非标准的、非关系型的特性扩展，因此如果可能的话，需要避免使用。

稍后我们将花费一些时间讨论在标准 SQL 中获取序列和唯一值的方法，这些方法不依赖于特性代码，也不使用硬件中的显式物理定位器。

#### 唯一值生成器

任何有用的唯一值生成器所具备的最重要特性是：从不生成两个相同的值。序列整数是厂商在他们的实现产品中替换正规键值的第一种方法。

本质上，这些生成器是 SQL 产品中的一段代码，这段代码查看上一次分配的值并将其加 1 作为下一数值。让我们从头开始，自己实现这一过程。首先，创建一个名为 GeneratorValues 的表，该表包含一行两列。

```
CREATE TABLE GeneratorValues
(lock CHAR(1) DEFAULT 'X' NOT NULL PRIMARY KEY -- 仅一行
CHECK (lock = 'X'),
keyval INTEGER DEFAULT 1 NOT NULL -- 仅正数
CHECK (key_val > 0));

-- 让所有用户使用这张表
GRANT SELECT, UPDATE(key_val)
ON TABLE GeneratorValues
TO PUBLIC;
```

现在需要为这张表创建一个函数，以获取并增加数值：

```
CREATE FUNCTION Generator()
RETURNS INTEGER
```

---

提供各种书籍的pd电子版代找服务，如果你找不到自己想要的书的pdf电子版，我们可以帮您找到，如有需要，请联系QQ1779903665.

PDF代找说明：

本人可以帮助你找到你要的PDF电子书，计算机类，文学，艺术，设计，医学，理学，经济，金融，等等。质量都很清晰，而且每本100%都带书签索引和目录，方便读者阅读观看，只要您提供给我书的相关信息，一般我都能找到，如果您有需求，请联系我QQ1779903665。

本人已经帮助了上万人找到了他们需要的PDF，其实网上有很多PDF,大家如果在网上不到的话，可以联系我QQ，大部分我都可以找到，而且每本100%带书签索引目录。因PDF电子书都有版权，请不要随意传播，如果您有经济购买能力，请尽量购买正版。

**声明：本人只提供代找服务，每本100%索引书签和目录，因寻找pdf电子书有一定难度，仅收取代找费用。如因PDF产生的版权纠纷，与本人无关，我们仅仅只是帮助你寻找到你要的pdf而已。**