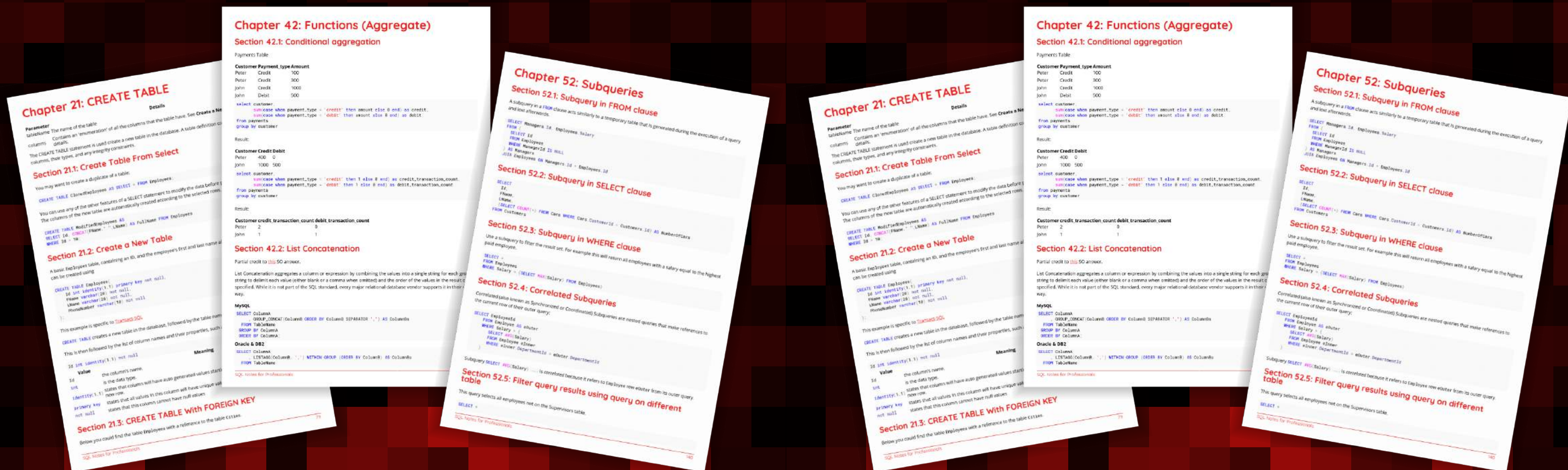


SQL

专业人员笔记

SQL

Notes for Professionals



100多页

专业提示和技巧

100+ pages

of professional hints and tricks

目录

关于	1
第1章：SQL入门	2
第1.1节：概述	2
第2章：标识符	3
第2.1节：未加引号的标识符	3
第3章：数据类型	4
第3.1节：DECIMAL和NUMERIC	4
第3.2节：FLOAT和REAL	4
第3.3节：整数	4
第3.4节：MONEY和SMALLMONEY	4
第3.5节：BINARY和VARBINARY	4
第3.6节：CHAR和VARCHAR	5
第3.7节：NCHAR和NVARCHAR	5
第3.8节：唯一标识符（UNIQUEIDENTIFIER）	5
第4章：NULL	6
第4.1节：查询中对NULL的过滤	6
第4.2节：表中的可空列	6
第4.3节：将字段更新为NULL	6
第4.4节：插入包含NULL字段的行	7
第5章：示例数据库和表	8
第5.1节：汽车修理店数据库	8
第5.2节：图书馆数据库	10
第5.3节：国家表	13
第6章：SELECT	14
第6.1节：使用通配符选择查询中的所有列	14
第6.2节：使用列别名的SELECT	15
第6.3节：选择单个列	18
第6.4节：选择指定数量的记录	19
第6.5节：带条件的选择	20
第6.6节：使用CASE进行选择	20
第6.7节：选择以保留关键字命名的列	21
第6.8节：使用表别名进行选择	21
第6.9节：选择多个条件	22
第6.10节：选择表但不加锁	23
第6.11节：使用聚合函数进行选择	23
第6.12节：根据列的多个值进行条件选择	24
第6.13节：获取行组的聚合结果	24
第6.14节：带排序结果的选择	25
第6.15节：选择空值	25
第6.16节：选择不同值（仅唯一值）	25
第6.17节：从多表中选择行	26
第7章：分组（GROUP BY）	27
第7.1节：基本的GROUP BY示例	27
第7.2节：使用HAVING子句过滤GROUP BY结果	28
第7.3节：使用GROUP BY统计给定列中每个唯一条目的行数	28
第7.4节：ROLAP聚合（数据挖掘）	29

Contents

About	1
Chapter 1: Getting started with SQL	2
Section 1.1: Overview	2
Chapter 2: Identifier	3
Section 2.1: Unquoted identifiers	3
Chapter 3: Data Types	4
Section 3.1: DECIMAL and NUMERIC	4
Section 3.2: FLOAT and REAL	4
Section 3.3: Integers	4
Section 3.4: MONEY and SMALLMONEY	4
Section 3.5: BINARY and VARBINARY	4
Section 3.6: CHAR and VARCHAR	5
Section 3.7: NCHAR and NVARCHAR	5
Section 3.8: UNIQUEIDENTIFIER	5
Chapter 4: NULL	6
Section 4.1: Filtering for NULL in queries	6
Section 4.2: Nullable columns in tables	6
Section 4.3: Updating fields to NULL	6
Section 4.4: Inserting rows with NULL fields	7
Chapter 5: Example Databases and Tables	8
Section 5.1: Auto Shop Database	8
Section 5.2: Library Database	10
Section 5.3: Countries Table	13
Chapter 6: SELECT	14
Section 6.1: Using the wildcard character to select all columns in a query	14
Section 6.2: SELECT Using Column Aliases	15
Section 6.3: Select Individual Columns	18
Section 6.4: Selecting specified number of records	19
Section 6.5: Selecting with Condition	20
Section 6.6: Selecting with CASE	20
Section 6.7: Select columns which are named after reserved keywords	21
Section 6.8: Selecting with table alias	21
Section 6.9: Selecting with more than 1 condition	22
Section 6.10: Selecting without Locking the table	23
Section 6.11: Selecting with Aggregate functions	23
Section 6.12: Select with condition of multiple values from column	24
Section 6.13: Get aggregated result for row groups	24
Section 6.14: Selection with sorted Results	25
Section 6.15: Selecting with null	25
Section 6.16: Select distinct (unique values only)	25
Section 6.17: Select rows from multiple tables	26
Chapter 7: GROUP BY	27
Section 7.1: Basic GROUP BY example	27
Section 7.2: Filter GROUP BY results using a HAVING clause	28
Section 7.3: USE GROUP BY to COUNT the number of rows for each unique entry in a given column	28
Section 7.4: ROLAP aggregation (Data Mining)	29

第8章：ORDER BY	31
第8.1节：按列号排序（而非名称）	31
第8.2节：使用ORDER BY和TOP根据列的值返回前x行	31
第8.3节：自定义排序顺序	32
第8.4节：按别名排序	32
第8.5节：按多列排序	33
第9章：AND 和 OR 运算符	34
第9.1节：AND OR 示例	34
第10章：CASE	35
第10.1节：使用 CASE 计算列中满足条件的行数	35
第10.2节：SELECT中的搜索CASE（匹配布尔表达式）	36
第10.3节：ORDER BY子句中的CASE	36
第10.4节：SELECT中的简写CASE	36
第10.5节：UPDATE中的CASE使用	37
第10.6节：CASE用于将NULL值排序在最后	37
第10.7节：ORDER BY子句中的CASE用于按两列的最低值排序记录	38
第11章：LIKE操作符	39
第11.1节：匹配开放式模式	39
第11.2节：单字符匹配	40
第11.3节：LIKE查询中的ESCAPE语句	40
第11.4节：搜索字符范围	41
第11.5节：按范围或集合匹配	41
第11.6节：通配符字符	41
第12章：IN子句	43
第12.1节：简单IN子句	43
第12.2节：使用带子查询的IN子句	43
第13章：使用WHERE和HAVING过滤结果	44
第13.1节：使用BETWEEN筛选结果	44
第13.2节：使用HAVING与聚合函数	45
第13.3节：WHERE子句与NULL/非NULL值	45
第13.4节：等值	46
第13.5节：WHERE子句仅返回符合其条件的行	46
第13.6节：AND 和 OR	46
第13.7节：使用 IN 返回列表中包含的值的行	47
第13.8节：使用 LIKE 查找匹配的字符串和子字符串	47
第13.9节：使用 WHERE EXISTS	48
第13.10节：使用 HAVING 检查组中的多个条件	48
第14章：跳过并获取（分页）	50
第14.1节：限制结果数量	50
第14.2节：跳过然后获取部分结果（分页）	50
第14.3节：从结果中跳过某些行	51
第15章：EXCEPT（差集）	52
第15.1节：选择数据集，排除在另一个数据集中存在的值	52
第16章：EXPLAIN 和 DESCRIBE	53
第16.1节：EXPLAIN Select 查询	53
第16.2节：DESCRIBE 表名;	53
第17章：EXISTS 子句	54
第17.1节：EXISTS 子句	54
第18章：JOIN	55

Chapter 8: ORDER BY	31
Section 8.1: Sorting by column number (instead of name)	31
Section 8.2: Use ORDER BY with TOP to return the top x rows based on a column's value	31
Section 8.3: Customized sorting order	32
Section 8.4: Order by Alias	32
Section 8.5: Sorting by multiple columns	33
Chapter 9: AND & OR Operators	34
Section 9.1: AND OR Example	34
Chapter 10: CASE	35
Section 10.1: Use CASE to COUNT the number of rows in a column match a condition	35
Section 10.2: Searched CASE in SELECT (Matches a boolean expression)	36
Section 10.3: CASE in a clause ORDER BY	36
Section 10.4: Shorthand CASE in SELECT	36
Section 10.5: Using CASE in UPDATE	37
Section 10.6: CASE use for NULL values ordered last	37
Section 10.7: CASE in ORDER BY clause to sort records by lowest value of 2 columns	38
Chapter 11: LIKE operator	39
Section 11.1: Match open-ended pattern	39
Section 11.2: Single character match	40
Section 11.3: ESCAPE statement in the LIKE-query	40
Section 11.4: Search for a range of characters	41
Section 11.5: Match by range or set	41
Section 11.6: Wildcard characters	41
Chapter 12: IN clause	43
Section 12.1: Simple IN clause	43
Section 12.2: Using IN clause with a subquery	43
Chapter 13: Filter results using WHERE and HAVING	44
Section 13.1: Use BETWEEN to Filter Results	44
Section 13.2: Use HAVING with Aggregate Functions	45
Section 13.3: WHERE clause with NULL/NOT NULL values	45
Section 13.4: Equality	46
Section 13.5: The WHERE clause only returns rows that match its criteria	46
Section 13.6: AND and OR	46
Section 13.7: Use IN to return rows with a value contained in a list	47
Section 13.8: Use LIKE to find matching strings and substrings	47
Section 13.9: Where EXISTS	48
Section 13.10: Use HAVING to check for multiple conditions in a group	48
Chapter 14: SKIP TAKE (Pagination)	50
Section 14.1: Limiting amount of results	50
Section 14.2: Skipping then taking some results (Pagination)	50
Section 14.3: Skipping some rows from result	51
Chapter 15: EXCEPT	52
Section 15.1: Select dataset except where values are in this other dataset	52
Chapter 16: EXPLAIN and DESCRIBE	53
Section 16.1: EXPLAIN Select query	53
Section 16.2: DESCRIBE tablename;	53
Chapter 17: EXISTS CLAUSE	54
Section 17.1: EXISTS CLAUSE	54
Chapter 18: JOIN	55

第18.1节：自连接	55
第18.2节：内连接与外连接的区别	56
第18.3节：连接术语：内连接、外连接、半连接、反连接等	59
第18.4节：左外连接	69
第18.5节：隐式连接	70
第18.6节：交叉连接（CROSS JOIN）	71
第18.7节：交叉应用（CROSS APPLY）与横向连接（LATERAL JOIN）	72
第18.8节：全连接（FULL JOIN）	73
第18.9节：递归连接（Recursive JOINS）	74
第18.10节：基本显式内连接	74
第18.11节：基于子查询的连接	75
第19章：更新（UPDATE）	76
第19.1节：使用来自另一张表的数据进行更新	76
第19.2节：修改现有值	77
第19.3节：更新指定行	77
第19.4节：更新所有行	77
第19.5节：捕获更新的记录	77
第20章：创建数据库	78
第20.1节：创建数据库	78
第21章：创建表	79
第21.1节：从选择创建表	79
第21.2节：创建新表	79
第21.3节：创建带外键的表	79
第21.4节：复制表	80
第21.5节：创建临时表或内存表	80
第22章：创建函数	82
第22.1节：创建新函数	82
第23章：TRY/CATCH	83
第23.1节：TRY/CATCH中的事务	83
第24章：UNION / UNION ALL	84
第24.1节：基本的UNION ALL查询	84
第24.2节：简单说明与示例	85
第25章：ALTER TABLE	86
第25.1节：添加列	86
第25.2节：删除列	86
第25.3节：添加主键	86
第25.4节：修改列	86
第25.5节：删除约束	86
第26章：插入（INSERT）	87
第26.1节：使用SELECT从另一张表插入数据	87
第26.2节：插入新行	87
第26.3节：仅插入指定列	87
第26.4节：一次插入多行	87
第27章：合并（MERGE）	88
第27.1节：合并以使目标匹配源	88
第27.2节：MySQL：按名称计数用户	88
第27.3节：PostgreSQL：按名称计数用户	88
第28章：cross apply, outer apply	90
第28.1节：CROSS APPLY 和 OUTER APPLY 基础	90

Section 18.1: Self Join	55
Section 18.2: Differences between inner/outer joins	56
Section 18.3: JOIN Terminology: Inner, Outer, Semi, Anti.	59
Section 18.4: Left Outer Join	69
Section 18.5: Implicit Join	70
Section 18.6: CROSS JOIN	71
Section 18.7: CROSS APPLY & LATERAL JOIN	72
Section 18.8: FULL JOIN	73
Section 18.9: Recursive JOINS	74
Section 18.10: Basic explicit inner join	74
Section 18.11: Joining on a Subquery	75
Chapter 19: UPDATE	76
Section 19.1: UPDATE with data from another table	76
Section 19.2: Modifying existing values	77
Section 19.3: Updating Specified Rows	77
Section 19.4: Updating All Rows	77
Section 19.5: Capturing Updated records	77
Chapter 20: CREATE Database	78
Section 20.1: CREATE Database	78
Chapter 21: CREATE TABLE	79
Section 21.1: Create Table From Select	79
Section 21.2: Create a New Table	79
Section 21.3: CREATE TABLE With FOREIGN KEY	79
Section 21.4: Duplicate a table	80
Section 21.5: Create a Temporary or In-Memory Table	80
Chapter 22: CREATE FUNCTION	82
Section 22.1: Create a new Function	82
Chapter 23: TRY/CATCH	83
Section 23.1: Transaction In a TRY/CATCH	83
Chapter 24: UNION / UNION ALL	84
Section 24.1: Basic UNION ALL query	84
Section 24.2: Simple explanation and Example	85
Chapter 25: ALTER TABLE	86
Section 25.1: Add Column(s)	86
Section 25.2: Drop Column	86
Section 25.3: Add Primary Key	86
Section 25.4: Alter Column	86
Section 25.5: Drop Constraint	86
Chapter 26: INSERT	87
Section 26.1: INSERT data from another table using SELECT	87
Section 26.2: Insert New Row	87
Section 26.3: Insert Only Specified Columns	87
Section 26.4: Insert multiple rows at once	87
Chapter 27: MERGE	88
Section 27.1: MERGE to make Target match Source	88
Section 27.2: MySQL: counting users by name	88
Section 27.3: PostgreSQL: counting users by name	88
Chapter 28: cross apply, outer apply	90
Section 28.1: CROSS APPLY and OUTER APPLY basics	90

第29章：删除	92
第29.1节：删除所有行	92
第29.2节：使用WHERE删除特定行	92
第29.3节：TRUNCATE子句	92
第29.4节：基于与其他表的比较删除特定行	92
第30章：TRUNCATE	94
第30.1节：从员工表中删除所有行	94
第31章：删除表	95
第31.1节：删除前检查是否存在	95
第31.2节：简单删除	95
第32章：DROP或DELETE数据库	96
第32.1节：DROP数据库	96
第33章：级联删除	97
第33.1节：ON DELETE CASCADE	97
第34章：授权与撤销	99
第34.1节：授予/撤销权限	99
第35章：XML	100
第35.1节：从XML数据类型查询	100
第36章：主键	101
第36.1节：创建主键	101
第36.2节：使用自动递增	101
第37章：索引	102
第37.1节：排序索引	102
第37.2节：部分索引或过滤索引	102
第37.3节：创建索引	102
第37.4节：删除索引，或禁用并重建索引	103
第37.5节：聚集索引、唯一索引和排序索引	103
第37.6节：重建索引	104
第37.7节：使用唯一索引插入	104
第38章：行号	105
第38.1节：删除除最后一条记录外的所有记录（一对多表）	105
第38.2节：无分区的行号	105
第38.3节：有分区的行号	105
第39章：SQL的Group By与Distinct	106
第39.1节：GROUP BY 与 DISTINCT 的区别	106
第40章：在列子集上查找重复项及详细信息	107
第40.1节：同名同生日的学生	107
第41章：字符串函数	108
第41.1节：连接	108
第41.2节：长度	108
第41.3节：修剪空格	109
第41.4节：大小写	109
第41.5节：拆分	109
第41.6节：替换	110
第41.7节：正则表达式（REGEXP）	110
第41.8节：子字符串	110
第41.9节：Stu	110
第41.10节：左 - 右	110

Chapter 29: DELETE	92
Section 29.1: DELETE all rows	92
Section 29.2: DELETE certain rows with WHERE	92
Section 29.3: TRUNCATE clause	92
Section 29.4: DELETE certain rows based upon comparisons with other tables	92
Chapter 30: TRUNCATE	94
Section 30.1: Removing all rows from the Employee table	94
Chapter 31: DROP Table	95
Section 31.1: Check for existence before dropping	95
Section 31.2: Simple drop	95
Chapter 32: DROP or DELETE Database	96
Section 32.1: DROP Database	96
Chapter 33: Cascading Delete	97
Section 33.1: ON DELETE CASCADE	97
Chapter 34: GRANT and REVOKE	99
Section 34.1: Grant/revoke privileges	99
Chapter 35: XML	100
Section 35.1: Query from XML Data Type	100
Chapter 36: Primary Keys	101
Section 36.1: Creating a Primary Key	101
Section 36.2: Using Auto Increment	101
Chapter 37: Indexes	102
Section 37.1: Sorted Index	102
Section 37.2: Partial or Filtered Index	102
Section 37.3: Creating an Index	102
Section 37.4: Dropping an Index, or Disabling and Rebuilding it	103
Section 37.5: Clustered, Unique, and Sorted Indexes	103
Section 37.6: Rebuild index	104
Section 37.7: Inserting with a Unique Index	104
Chapter 38: Row number	105
Section 38.1: Delete All But Last Record (1 to Many Table)	105
Section 38.2: Row numbers without partitions	105
Section 38.3: Row numbers with partitions	105
Chapter 39: SQL Group By vs Distinct	106
Section 39.1: Difference between GROUP BY and DISTINCT	106
Chapter 40: Finding Duplicates on a Column Subset with Detail	107
Section 40.1: Students with same name and date of birth	107
Chapter 41: String Functions	108
Section 41.1: Concatenate	108
Section 41.2: Length	108
Section 41.3: Trim empty spaces	109
Section 41.4: Upper & lower case	109
Section 41.5: Split	109
Section 41.6: Replace	110
Section 41.7: REGEXP	110
Section 41.8: Substring	110
Section 41.9: Stuff	110
Section 41.10: LEFT - RIGHT	110

第41.11节：反转	111
第41.12节：复制	111
第41.13节：SQL Select 和 Update 查询中的 Replace 函数	111
第41.14节：INSTR	112
第41.15节：PARSENAME	112
第42章：函数（聚合）	114
第42.1节：条件聚合	114
第42.2节：列表连接	114
第42.3节：求和	116
第42.4节：平均值()	116
第42.5节：计数	116
第42.6节：最小值	117
第42.7节：最大值	118
第43章：函数（标量/单行）	119
第43.1节：日期和时间	119
第43.2节：字符修改	120
第43.3节：配置和转换函数	120
第43.4节：逻辑和数学函数	121
第44章：函数（分析）	123
第44.1节：LAG和LEAD	123
第44.2节：PERCENTILE_DISC和PERCENTILE_CONT	123
第44.3节：FIRST_VALUE	124
第44.4节：LAST_VALUE	125
第44.5节：PERCENT_RANK和CUME_DIST	125
第45章：窗口函数	127
第45.1节：设置标志以判断其他行是否具有共同属性	127
第45.2节：使用LAG()函数查找“非顺序”记录	127
第45.3节：获取累计总和	128
第45.4节：将选中的总计行添加到每一行	128
第45.5节：获取多个分组中的最近N行	129
第46章：公共表表达式	130
第46.1节：生成数值	130
第46.2节：递归枚举子树	130
第46.3节：临时查询	131
第46.4节：递归向上遍历树	131
第46.5节：递归生成日期，扩展为包含团队排班示例	132
第46.6节：使用递归CTE的Oracle CONNECT BY功能	132
第47章：视图	134
第47.1节：简单视图	134
第47.2节：复杂视图	134
第48章：物化视图	135
第48.1节：PostgreSQL示例	135
第49章：注释	136
第49.1节：单行注释	136
第49.2节：多行注释	136
第50章：外键	137
第50.1节：外键说明	137
第50.2节：创建带外键的表	137
第51章：序列	139

Section 41.11: REVERSE	111
Section 41.12: REPLICATE	111
Section 41.13: Replace function in sql Select and Update query	111
Section 41.14: INSTR	112
Section 41.15: PARSENAME	112
Chapter 42: Functions (Aggregate)	114
Section 42.1: Conditional aggregation	114
Section 42.2: List Concatenation	114
Section 42.3: SUM	116
Section 42.4: AVG()	116
Section 42.5: Count	116
Section 42.6: Min	117
Section 42.7: Max	118
Chapter 43: Functions (Scalar/Single Row)	119
Section 43.1: Date And Time	119
Section 43.2: Character modifications	120
Section 43.3: Configuration and Conversion Function	120
Section 43.4: Logical and Mathmetical Function	121
Chapter 44: Functions (Analytic)	123
Section 44.1: LAG and LEAD	123
Section 44.2: PERCENTILE_DISC and PERCENTILE_CONT	123
Section 44.3: FIRST_VALUE	124
Section 44.4: LAST_VALUE	125
Section 44.5: PERCENT_RANK and CUME_DIST	125
Chapter 45: Window Functions	127
Section 45.1: Setting up a flag if other rows have a common property	127
Section 45.2: Finding “out-of-sequence” records using the LAG() function	127
Section 45.3: Getting a running total	128
Section 45.4: Adding the total rows selected to every row	128
Section 45.5: Getting the N most recent rows over multiple grouping	129
Chapter 46: Common Table Expressions	130
Section 46.1: generating values	130
Section 46.2: recursively enumerating a subtree	130
Section 46.3: Temporary query	131
Section 46.4: recursively going up in a tree	131
Section 46.5: Recursively generate dates, extended to include team rostering as example	132
Section 46.6: Oracle CONNECT BY functionality with recursive CTEs	132
Chapter 47: Views	134
Section 47.1: Simple views	134
Section 47.2: Complex views	134
Chapter 48: Materialized Views	135
Section 48.1: PostgreSQL example	135
Chapter 49: Comments	136
Section 49.1: Single-line comments	136
Section 49.2: Multi-line comments	136
Chapter 50: Foreign Keys	137
Section 50.1: Foreign Keys explained	137
Section 50.2: Creating a table with a foreign key	137
Chapter 51: Sequence	139

第51.1节：创建序列	139
第51.2节：使用序列	139
第52章：子查询	140
第52.1节：FROM子句中的子查询	140
第52.2节：SELECT子句中的子查询	140
第52.3节：WHERE子句中的子查询	140
第52.4节：相关子查询	140
第52.5节：使用不同表的查询过滤查询结果	140
第52.6节：FROM子句中的子查询	141
第52.7节：WHERE子句中的子查询	141
第53章：执行块	142
第53.1节：使用 BEGIN ... END	142
第54章：存储过程	143
第54.1节：创建并调用存储过程	143
第55章：触发器	144
第55.1节：CREATE TRIGGER	144
第55.2节：使用触发器管理已删除项目的“回收站”	144
第56章：事务	145
第56.1节：简单事务	145
第56.2节：回滚事务	145
第57章：表设计	146
第57.1节：良好设计表的属性	146
第58章：同义词	147
第58.1节：创建同义词	147
第59章：信息模式	148
第59.1节：基本信息模式搜索	148
第60章：执行顺序	149
第60.1节：SQL中查询处理的逻辑顺序	149
第61章：SQL中的整洁代码	150
第61.1节：关键字和名称的格式及拼写	150
第61.2节：缩进	150
第61.3节：SELECT *	151
第61.4节：连接	152
第62章：SQL注入	153
第62.1节：SQL注入示例	153
第62.2节：简单注入示例	154
鸣谢	155
你可能也喜欢	159

Section 51.1: Create Sequence	139
Section 51.2: Using Sequences	139
Chapter 52: Subqueries	140
Section 52.1: Subquery in FROM clause	140
Section 52.2: Subquery in SELECT clause	140
Section 52.3: Subquery in WHERE clause	140
Section 52.4: Correlated Subqueries	140
Section 52.5: Filter query results using query on different table	140
Section 52.6: Subqueries in FROM clause	141
Section 52.7: Subqueries in WHERE clause	141
Chapter 53: Execution blocks	142
Section 53.1: Using BEGIN ... END	142
Chapter 54: Stored Procedures	143
Section 54.1: Create and call a stored procedure	143
Chapter 55: Triggers	144
Section 55.1: CREATE TRIGGER	144
Section 55.2: Use Trigger to manage a "Recycle Bin" for deleted items	144
Chapter 56: Transactions	145
Section 56.1: Simple Transaction	145
Section 56.2: Rollback Transaction	145
Chapter 57: Table Design	146
Section 57.1: Properties of a well designed table	146
Chapter 58: Synonyms	147
Section 58.1: Create Synonym	147
Chapter 59: Information Schema	148
Section 59.1: Basic Information Schema Search	148
Chapter 60: Order of Execution	149
Section 60.1: Logical Order of Query Processing in SQL	149
Chapter 61: Clean Code in SQL	150
Section 61.1: Formatting and Spelling of Keywords and Names	150
Section 61.2: Indenting	150
Section 61.3: SELECT *	151
Section 61.4: Joins	152
Chapter 62: SQL Injection	153
Section 62.1: SQL injection sample	153
Section 62.2: simple injection sample	154
Credits	155
You may also like	159

欢迎随意免费分享此PDF，
本书最新版本可从以下网址下载：
<https://goalkicker.com/SQLBook>

本《专业人士的SQL笔记》一书汇编自[Stack Overflow Documentation](#)，内容由Stack Overflow的优秀人士撰写。
文本内容采用知识共享署名-相同方式共享许可协议发布，详见本书末尾对各章节贡献者的致谢。图片版权归各自所有者所有，除非另有说明。

这是一本非官方的免费书籍，旨在教育用途，与官方SQL组织或公司及Stack Overflow无关。所有商标和注册商标均为其各自公司所有者所有。

本书中提供的信息不保证正确或准确，使用风险自负

请将反馈和更正发送至web@petercv.com

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:
<https://goalkicker.com/SQLBook>

This *SQL Notes for Professionals* book is compiled from [Stack Overflow Documentation](#), the content is written by the beautiful people at Stack Overflow. Text content is released under Creative Commons BY-SA, see credits at the end of this book whom contributed to the various chapters. Images may be copyright of their respective owners unless otherwise specified

This is an unofficial free book created for educational purposes and is not affiliated with official SQL group(s) or company(s) nor Stack Overflow. All trademarks and registered trademarks are the property of their respective company owners

The information presented in this book is not guaranteed to be correct nor accurate, use at your own risk

Please send feedback and corrections to web@petercv.com

第1章：SQL入门

版本简称	标准		发布日期
1986	SQL-86	ANSI X3.135-1986, ISO 9075:1987	1986-01-01
1989	SQL-89	ANSI X3.135-1989, ISO/IEC 9075:1989	1989-01-01
1992	SQL-92	ISO/IEC 9075:1992	1992-01-01
1999	SQL:1999	ISO/IEC 9075:1999	1999-12-16
2003	SQL:2003	ISO/IEC 9075:2003	2003-12-15
2006	SQL:2006	ISO/IEC 9075:2006	2006-06-01
2008	SQL:2008	ISO/IEC 9075:2008	2008-07-15
2011	SQL:2011	ISO/IEC 9075:2011	2011-12-15
2016	SQL:2016	ISO/IEC 9075:2016	2016-12-01

第1.1节：概述

结构化查询语言（SQL）是一种专用的编程语言，设计用于管理存储在关系数据库管理系统（RDBMS）中的数据。类似SQL的语言也可以用于关系数据流管理系统（RDSMS）或“非仅SQL”（NoSQL）数据库。

SQL 包含三个主要的子语言：

1. 数据定义语言（DDL）：用于创建和修改数据库的结构；2. 数据操作语言（DML）：用于对数据库中的数据执行读取、插入、更新和删除操作；
3. 数据控制语言（DCL）：用于控制对数据库中存储数据的访问。

[维基百科上的SQL条目](#)

核心的DML操作是创建、读取、更新和删除（简称CRUD），这些操作通过语句INSERT、SELECT、UPDATE和DELETE来执行。还有一个（最近新增的）MERGE语句，可以执行所有三种写操作（INSERT、UPDATE、DELETE）。

[维基百科上的CRUD条目](#)

许多SQL数据库被实现为客户端/服务器系统；“SQL服务器”一词描述了这样的数据库。同时，微软开发了一个名为“SQL Server”的数据库。虽然该数据库使用的是SQL的一种方言，但与该数据库特定相关的信息不属于本标签的讨论范围，应参见SQL Server的官方文档。

Chapter 1: Getting started with SQL

Version	Short Name	Standard	Release Date
1986	SQL-86	ANSI X3.135-1986, ISO 9075:1987	1986-01-01
1989	SQL-89	ANSI X3.135-1989, ISO/IEC 9075:1989	1989-01-01
1992	SQL-92	ISO/IEC 9075:1992	1992-01-01
1999	SQL:1999	ISO/IEC 9075:1999	1999-12-16
2003	SQL:2003	ISO/IEC 9075:2003	2003-12-15
2006	SQL:2006	ISO/IEC 9075:2006	2006-06-01
2008	SQL:2008	ISO/IEC 9075:2008	2008-07-15
2011	SQL:2011	ISO/IEC 9075:2011	2011-12-15
2016	SQL:2016	ISO/IEC 9075:2016	2016-12-01

Section 1.1: Overview

Structured Query Language (SQL) is a special-purpose programming language designed for managing data held in a Relational Database Management System (RDBMS). SQL-like languages can also be used in Relational Data Stream Management Systems (RDSMS), or in "not-only SQL" (NoSQL) databases.

SQL comprises of 3 major sub-languages:

1. Data Definition Language (DDL): to create and modify the structure of the database;
2. Data Manipulation Language (DML): to perform Read, Insert, Update and Delete operations on the data of the database;
3. Data Control Language (DCL): to control the access of the data stored in the database.

[SQL article on Wikipedia](#)

The core DML operations are Create, Read, Update and Delete (CRUD for short) which are performed by the statements [INSERT](#), [SELECT](#), [UPDATE](#) and [DELETE](#). There is also a (recently added) [MERGE](#) statement which can perform all 3 write operations (INSERT, UPDATE, DELETE).

[CRUD article on Wikipedia](#)

Many SQL databases are implemented as client/server systems; the term "SQL server" describes such a database. At the same time, Microsoft makes a database that is named "SQL Server". While that database speaks a dialect of SQL, information specific to that database is not on topic in this tag but belongs into the SQL Server documentation.

第二章：标识符

本主题涉及标识符，即表名、列名及其他数据库对象名称的语法规则。

在适当情况下，示例应涵盖不同SQL实现的变体，或标明示例所使用的SQL实现。

第2.1节：未加引号的标识符

未加引号的标识符可以使用字母（a-z）、数字（0-9）和下划线（_），且必须以字母开头。

根据SQL实现和/或数据库设置，可能允许使用其他字符，有些甚至可以作为第一个字符，例如：

- MS SQL：@、\$、#及其他Unicode字母（来源）_____
- MySQL：\$（来源）_____
- Oracle：\$、#及数据库字符集中的其他字母（来源）_____
- PostgreSQL：\$及其他Unicode字母（来源）_____

未加引号的标识符不区分大小写。具体处理方式在很大程度上取决于SQL实现：

- MS SQL：保留大小写，敏感性由数据库字符集定义，因此可能区分大小写。
- MySQL：保留大小写，敏感性取决于数据库设置和底层文件系统。
- Oracle：转换为大写字母，然后像引用标识符一样处理。
- PostgreSQL：转换为小写字母，然后像引用标识符一样处理。
- SQLite：保留大小写；仅对ASCII字符不区分大小写。

Chapter 2: Identifier

This topic is about identifiers, i.e. syntax rules for names of tables, columns, and other database objects.

Where appropriate, the examples should cover variations used by different SQL implementations, or identify the SQL implementation of the example.

Section 2.1: Unquoted identifiers

Unquoted identifiers can use letters (a-z), digits (0-9), and underscore (_), and must start with a letter.

Depending on SQL implementation, and/or database settings, other characters may be allowed, some even as the first character, e.g.

- MS SQL: @, \$, #, and other Unicode letters (*source*)
- MySQL: \$ (*source*)
- Oracle: \$, #, and other letters from database character set (*source*)
- PostgreSQL: \$, and other Unicode letters (*source*)

Unquoted identifiers are case-insensitive. How this is handled depends greatly on SQL implementation:

- MS SQL: Case-preserving, sensitivity defined by database character set, so can be case-sensitive.
- MySQL: Case-preserving, sensitivity depends on database setting and underlying file system.
- Oracle: Converted to uppercase, then handled like quoted identifier.
- PostgreSQL: Converted to lowercase, then handled like quoted identifier.
- SQLite: Case-preserving; case insensitivity only for ASCII characters.

第3章：数据类型

第3.1节：DECIMAL和NUMERIC

固定精度和小数位数的十进制数。DECIMAL和NUMERIC在功能上等效。

语法：

```
DECIMAL ( precision [ , scale ] )
NUMERIC ( precision [ , scale ] )
```

示例：

```
SELECT CAST(123 AS DECIMAL(5,2)) --返回 123.00
SELECT CAST(12345.12 AS NUMERIC(10,5)) --返回 12345.12000
```

第3.2节：FLOAT和REAL

用于浮点数数值数据的近似数值数据类型。

```
SELECT CAST( PI() AS FLOAT) --返回 3.14159265358979
SELECT CAST( PI() AS REAL) --返回 3.141593
```

第3.3节：整数

使用整数数据的精确数值数据类型。

数据类型	范围	存储
bigint	-2^63 (-9,223,372,036,854,775,808) 到 2^63-1 (9,223,372,036,854,775,807)	8字节
int	-2^31 (-2,147,483,648) 到 2^31-1 (2,147,483,647)	4 字节
smallint	-2^15 (-32,768) 到 2^15-1 (32,767)	2 字节
tinyint	0 到 255	1 字节

第3.4节：MONEY 和 SMALLMONEY

表示货币或货币值的数据类型。

数据类型	范围	存储
money	-922,337,203,685,477.5808 到 922,337,203,685,477.5807	8 字节
smallmoney	-214,748.3648 到 214,748.3647	4 字节

第3.5节：BINARY 和 VARBINARY

固定长度或可变长度的二进制数据类型。

语法：

```
BINARY [ ( n_bytes ) ]
VARBINARY [ ( n_bytes | max ) ]
```

n_bytes 可以是1到8000字节之间的任意数字。 max 表示最大存储空间为2^31-1。

Chapter 3: Data Types

Section 3.1: DECIMAL and NUMERIC

Fixed precision and scale decimal numbers. **DECIMAL** and **NUMERIC** are functionally equivalent.

Syntax:

```
DECIMAL ( precision [ , scale ] )
NUMERIC ( precision [ , scale ] )
```

Examples:

```
SELECT CAST(123 AS DECIMAL(5,2)) --returns 123.00
SELECT CAST(12345.12 AS NUMERIC(10,5)) --returns 12345.12000
```

Section 3.2: FLOAT and REAL

Approximate-number data types for use with floating point numeric data.

```
SELECT CAST( PI() AS FLOAT) --returns 3.14159265358979
SELECT CAST( PI() AS REAL) --returns 3.141593
```

Section 3.3: Integers

Exact-number data types that use integer data.

Data type	Range	Storage
bigint	-2^63 (-9,223,372,036,854,775,808) to 2^63-1 (9,223,372,036,854,775,807)	8 Bytes
int	-2^31 (-2,147,483,648) to 2^31-1 (2,147,483,647)	4 Bytes
smallint	-2^15 (-32,768) to 2^15-1 (32,767)	2 Bytes
tinyint	0 to 255	1 Byte

Section 3.4: MONEY and SMALLMONEY

Data types that represent monetary or currency values.

Data type	Range	Storage
money	-922,337,203,685,477.5808 to 922,337,203,685,477.5807	8 bytes
smallmoney	-214,748.3648 to 214,748.3647	4 bytes

Section 3.5: BINARY and VARBINARY

Binary data types of either fixed length or variable length.

Syntax:

```
BINARY [ ( n_bytes ) ]
VARBINARY [ ( n_bytes | max ) ]
```

n_bytes can be any number from 1 to 8000 bytes. **max** indicates that the maximum storage space is 2^31-1.

示例：

```
SELECT CAST(12345 AS BINARY(10)) -- 0x000000000000000003039
SELECT CAST(12345 AS VARBINARY(10)) -- 0x00003039
```

第3.6节：CHAR 和 VARCHAR

固定长度或可变长度的字符串数据类型。

语法：

```
CHAR [ ( n_chars ) ]
VARCHAR [ ( n_chars ) ]
```

示例：

```
SELECT CAST('ABC' AS CHAR(10)) -- 'ABC      ' (右侧用空格填充)
SELECT CAST('ABC' AS VARCHAR(10)) -- 'ABC' (由于可变字符，无填充)
SELECT CAST('ABCDEFGHIJKLMNOPQRSTUVWXYZ' AS CHAR(10)) -- 'ABCDEFGHIJ' (截断为10个字符)
```

第3.7节：NCHAR 和 NVARCHAR

UNICODE 字符串数据类型，可以是固定长度或可变长度。

语法：

```
NCHAR [ ( n_chars ) ]
NVARCHAR [ ( n_chars | MAX ) ]
```

对于可能超过 8000 个字符的超长字符串，请使用 MAX。

第 3.8 节：UNIQUEIDENTIFIER

一个 16 字节的 GUID / UUID。

```
DECLARE @GUID UNIQUEIDENTIFIER = NEWID();
SELECT @GUID -- 'E28B3BD9-9174-41A9-8508-899A78A33540'
DECLARE @bad_GUID_string VARCHAR(100) = 'E28B3BD9-9174-41A9-8508-899A78A33540_foobarbaz'
SELECT
    @bad_GUID_string, -- 'E28B3BD9-9174-41A9-8508-899A78A33540_foobarbaz'
    CONVERT(UNIQUEIDENTIFIER, @bad_GUID_string) -- 'E28B3BD9-9174-41A9-8508-899A78A33540'
```

Examples:

```
SELECT CAST(12345 AS BINARY(10)) -- 0x000000000000000003039
SELECT CAST(12345 AS VARBINARY(10)) -- 0x00003039
```

Section 3.6: CHAR and VARCHAR

String data types of either fixed length or variable length.

Syntax:

```
CHAR [ ( n_chars ) ]
VARCHAR [ ( n_chars ) ]
```

Examples:

```
SELECT CAST('ABC' AS CHAR(10)) -- 'ABC      ' (padded with spaces on the right)
SELECT CAST('ABC' AS VARCHAR(10)) -- 'ABC' (no padding due to variable character)
SELECT CAST('ABCDEFGHIJKLMNOPQRSTUVWXYZ' AS CHAR(10)) -- 'ABCDEFGHIJ' (truncated to 10 characters)
```

Section 3.7: NCHAR and NVARCHAR

UNICODE string data types of either fixed length or variable length.

Syntax:

```
NCHAR [ ( n_chars ) ]
NVARCHAR [ ( n_chars | MAX ) ]
```

Use MAX for very long strings that may exceed 8000 characters.

Section 3.8: UNIQUEIDENTIFIER

A 16-byte GUID / UUID.

```
DECLARE @GUID UNIQUEIDENTIFIER = NEWID();
SELECT @GUID -- 'E28B3BD9-9174-41A9-8508-899A78A33540'
DECLARE @bad_GUID_string VARCHAR(100) = 'E28B3BD9-9174-41A9-8508-899A78A33540_foobarbaz'
SELECT
    @bad_GUID_string, -- 'E28B3BD9-9174-41A9-8508-899A78A33540_foobarbaz'
    CONVERT(UNIQUEIDENTIFIER, @bad_GUID_string) -- 'E28B3BD9-9174-41A9-8508-899A78A33540'
```


第 4 章：NULL

在 SQL 以及一般编程中，NULL 字面意思是“无”。在 SQL 中，更容易理解为“没有任何值”。

重要的是要将其与看似为空的值区分开来，例如空字符串"或数字0，这两者实际上都不是NULL。

还需注意不要将NULL用引号括起来，如'NULL'，这在接受文本的列中是允许的，但它不是NULL，可能导致错误和不正确的数据集。

第4.1节：查询中对NULL的过滤

在WHERE块中过滤NULL（即值的缺失）的语法与过滤特定值略有不同。

```
SELECT * FROM Employees WHERE ManagerId IS NULL ;
SELECT * FROM Employees WHERE ManagerId IS NOT NULL ;
```

注意，由于NULL不等于任何值，甚至不等于它自身，使用等号操作符= NULL或<> NULL（或!= NULL）总是返回UNKNOWN的真值，WHERE会拒绝该结果。

WHERE过滤掉条件为FALSE或UNKNOWN的所有行，只保留条件为TRUE的行。

第4.2节：表中的可空列

创建表时，可以声明列为可空或非空。

```
CREATE TABLE MyTable
(
  MyCol1 INT NOT NULL, -- 非空
  MyCol2 INT NULL      -- 可空
) ;
```

默认情况下，除主键约束中的列外，每一列都是可为空的，除非我们明确设置了NOT NULL约束。

尝试向不可为空的列赋值NULL将导致错误。

```
INSERT INTO MyTable (MyCol1, MyCol2) VALUES (1, NULL) ; -- 正常执行

INSERT INTO MyTable (MyCol1, MyCol2) VALUES (NULL, 2) ;
-- 无法插入
-- 值 NULL 插入到列 'MyCol1', 表 'MyTable' 中；
-- 该列不允许为空。插入失败。
```

第4.3节：将字段更新为NULL

将字段设置为NULL的操作与设置为其他任何值完全相同：

```
UPDATE Employees
SET ManagerId = NULL
WHERE Id = 4
```

Chapter 4: NULL

NULL in SQL, as well as programming in general, means literally "nothing". In SQL, it is easier to understand as "the absence of any value".

It is important to distinguish it from seemingly empty values, such as the empty string '' or the number 0, neither of which are actually NULL.

It is also important to be careful not to enclose NULL in quotes, like 'NULL', which is allowed in columns that accept text, but is not NULL and can cause errors and incorrect data sets.

Section 4.1: Filtering for NULL in queries

The syntax for filtering for NULL (i.e. the absence of a value) in WHERE blocks is slightly different than filtering for specific values.

```
SELECT * FROM Employees WHERE ManagerId IS NULL ;
SELECT * FROM Employees WHERE ManagerId IS NOT NULL ;
```

Note that because NULL is not equal to anything, not even to itself, using equality operators = NULL or <> NULL (or != NULL) will always yield the truth value of UNKNOWN which will be rejected by WHERE.

WHERE filters all rows that the condition is FALSE or UNKNOWN and keeps only rows that the condition is TRUE.

Section 4.2: Nullable columns in tables

When creating tables it is possible to declare a column as nullable or non-nullable.

```
CREATE TABLE MyTable
(
  MyCol1 INT NOT NULL, -- non-nullable
  MyCol2 INT NULL      -- nullable
) ;
```

By default every column (except those in primary key constraint) is nullable unless we explicitly set NOT NULL constraint.

Attempting to assign NULL to a non-nullable column will result in an error.

```
INSERT INTO MyTable (MyCol1, MyCol2) VALUES (1, NULL) ; -- works fine

INSERT INTO MyTable (MyCol1, MyCol2) VALUES (NULL, 2) ;
-- cannot insert
-- the value NULL into column 'MyCol1', table 'MyTable';
-- column does not allow nulls. INSERT fails.
```

Section 4.3: Updating fields to NULL

Setting a field to NULL works exactly like with any other value:

```
UPDATE Employees
SET ManagerId = NULL
WHERE Id = 4
```

第4.4节：插入包含NULL字段的行

例如，将一个没有电话号码且没有经理的员工插入到员工示例表中：

```
INSERT INTO 员工
    (编号, 名, 姓, 电话号码, 经理编号, 部门编号, 薪水, 入职日期)
VALUES
    (5, 'Jane', 'Doe', NULL, NULL, 2, 800, '2016-07-22') ;
```

Section 4.4: Inserting rows with NULL fields

For example inserting an employee with no phone number and no manager into the Employees example table:

```
INSERT INTO Employees
    (Id, FName, LName, PhoneNumber, ManagerId, DepartmentId, Salary, HireDate)
VALUES
    (5, 'Jane', 'Doe', NULL, NULL, 2, 800, '2016-07-22') ;
```

第5章：示例数据库和表

第5.1节：汽车修理店数据库

在以下示例中——汽车修理店业务数据库，我们有部门、员工、客户和客户车辆的列表。我们使用外键在各个表之间创建关系。

实时示例：[SQL fiddle](#)

表之间的关系

- 每个部门可能有0个或多个员工
- 每个员工可能有0个或1个经理
- 每个客户可能有0个或多辆车

部门

编号 名称

1	人力资源部
2	销售部
3	技术部

创建表的SQL语句：

```
CREATE TABLE 部门 (  
    编号 INT NOT NULL AUTO_INCREMENT,  
    名称 VARCHAR(25) NOT NULL,  
    PRIMARY KEY(编号)  
);  
  
INSERT INTO 部门  
    ([编号], [名称])  
VALUES  
    (1, '人力资源部'),  
    (2, '销售部'),  
    (3, '技术部')  
;
```

员工

编号	名字	姓氏	电话号码	经理编号	部门编号	薪水	入职日期
1	詹姆斯	史密斯	1234567890	空	1	1000	01-01-2002
2	约翰	约翰逊	2468101214	1	1	400	23-03-2005
3	迈克尔	威廉姆斯	1357911131	1	2	600	12-05-2009
4	约纳森·史密斯		1212121212	2	1	500	24-07-2016

创建表的SQL语句：

```
CREATE TABLE Employees (  
    Id INT NOT NULL AUTO_INCREMENT,  
    FName VARCHAR(35) NOT NULL,  
    LName VARCHAR(35) NOT NULL,  
    PhoneNumber VARCHAR(11),  
    ManagerId INT,  
    DepartmentId INT NOT NULL,
```

Chapter 5: Example Databases and Tables

Section 5.1: Auto Shop Database

In the following example - Database for an auto shop business, we have a list of departments, employees, customers and customer cars. We are using foreign keys to create relationships between the various tables.

Live example: [SQL fiddle](#)

Relationships between tables

- Each Department may have 0 or more Employees
- Each Employee may have 0 or 1 Manager
- Each Customer may have 0 or more Cars

Departments

Id Name

1	HR
2	Sales
3	Tech

SQL statements to create the table:

```
CREATE TABLE Departments (  
    Id INT NOT NULL AUTO_INCREMENT,  
    Name VARCHAR(25) NOT NULL,  
    PRIMARY KEY(Id)  
);  
  
INSERT INTO Departments  
    ([Id], [Name])  
VALUES  
    (1, 'HR'),  
    (2, 'Sales'),  
    (3, 'Tech')  
;
```

Employees

Id	FName	LName	PhoneNumber	ManagerId	DepartmentId	Salary	HireDate
1	James	Smith	1234567890	NULL	1	1000	01-01-2002
2	John	Johnson	2468101214	1	1	400	23-03-2005
3	Michael	Williams	1357911131	1	2	600	12-05-2009
4	Johnathon Smith		1212121212	2	1	500	24-07-2016

SQL statements to create the table:

```
CREATE TABLE Employees (  
    Id INT NOT NULL AUTO_INCREMENT,  
    FName VARCHAR(35) NOT NULL,  
    LName VARCHAR(35) NOT NULL,  
    PhoneNumber VARCHAR(11),  
    ManagerId INT,  
    DepartmentId INT NOT NULL,
```

```
Salary INT NOT NULL,
  HireDate DATETIME NOT NULL,
  PRIMARY KEY(Id),
  FOREIGN KEY (ManagerId) REFERENCES Employees(Id),
  FOREIGN KEY (DepartmentId) REFERENCES Departments(Id)
);

INSERT INTO 员工
  ([Id], [FName], [LName], [PhoneNumber], [ManagerId], [DepartmentId], [Salary], [HireDate])
VALUES
  (1, 'James', 'Smith', 1234567890, NULL, 1, 1000, '01-01-2002'),
  (2, 'John', 'Johnson', 2468101214, '1', 1, 400, '23-03-2005'),
  (3, 'Michael', 'Williams', 1357911131, '1', 2, 600, '12-05-2009'),
  (4, 'Johnathon', 'Smith', 1212121212, '2', 1, 500, '24-07-2016')
;
```

客户

编号	名字	姓	电子邮件	电话号码	首选联系方式
1	威廉·琼斯		william.jones@example.com	3347927472	电话
2	大卫	Miller	dmiller@example.net	2137921892	电子邮件
3	理查德·戴维斯		richard0123@example.com	NULL	电子邮件

创建表的SQL语句：

```
CREATE TABLE Customers (
  Id INT NOT NULL AUTO_INCREMENT,
  FName VARCHAR(35) NOT NULL,
  LName VARCHAR(35) NOT NULL,
  Email varchar(100) NOT NULL,
  PhoneNumber VARCHAR(11),
  PreferredContact VARCHAR(5) NOT NULL,
  PRIMARY KEY(Id)
);

INSERT INTO Customers
  ([编号], [名字], [姓氏], [邮箱], [电话号码], [首选联系方式])
VALUES
  (1, '威廉', '琼斯', 'william.jones@example.com', '3347927472', 'PHONE'),
  (2, '大卫', '米勒', 'dmiller@example.net', '2137921892', 'EMAIL'),
  (3, '理查德', '戴维斯', 'richard0123@example.com', NULL, 'EMAIL')
;
```

汽车

编号	客户编号	员工编号	车型	状态	总费用
1	1	2	福特 F-150	准备就绪	230
2	1	2	福特 F-150	准备就绪	200
3	2	1	福特野马	WAITING	100
4	3	3	丰田普锐斯	WORKING	1254

创建表的SQL语句：

```
CREATE TABLE Cars (
  Id INT NOT NULL AUTO_INCREMENT,
  CustomerId INT NOT NULL,
  EmployeeId INT NOT NULL,
  Model varchar(50) NOT NULL,
  Status varchar(25) NOT NULL,
```

```
Salary INT NOT NULL,
  HireDate DATETIME NOT NULL,
  PRIMARY KEY(Id),
  FOREIGN KEY (ManagerId) REFERENCES Employees(Id),
  FOREIGN KEY (DepartmentId) REFERENCES Departments(Id)
);

INSERT INTO Employees
  ([Id], [FName], [LName], [PhoneNumber], [ManagerId], [DepartmentId], [Salary], [HireDate])
VALUES
  (1, 'James', 'Smith', 1234567890, NULL, 1, 1000, '01-01-2002'),
  (2, 'John', 'Johnson', 2468101214, '1', 1, 400, '23-03-2005'),
  (3, 'Michael', 'Williams', 1357911131, '1', 2, 600, '12-05-2009'),
  (4, 'Johnathon', 'Smith', 1212121212, '2', 1, 500, '24-07-2016')
;
```

Customers

Id	FName	LName	Email	PhoneNumber	PreferredContact
1	William	Jones	william.jones@example.com	3347927472	PHONE
2	David	Miller	dmiller@example.net	2137921892	EMAIL
3	Richard	Davis	richard0123@example.com	NULL	EMAIL

SQL statements to create the table:

```
CREATE TABLE Customers (
  Id INT NOT NULL AUTO_INCREMENT,
  FName VARCHAR(35) NOT NULL,
  LName VARCHAR(35) NOT NULL,
  Email varchar(100) NOT NULL,
  PhoneNumber VARCHAR(11),
  PreferredContact VARCHAR(5) NOT NULL,
  PRIMARY KEY(Id)
);

INSERT INTO Customers
  ([Id], [FName], [LName], [Email], [PhoneNumber], [PreferredContact])
VALUES
  (1, 'William', 'Jones', 'william.jones@example.com', '3347927472', 'PHONE'),
  (2, 'David', 'Miller', 'dmiller@example.net', '2137921892', 'EMAIL'),
  (3, 'Richard', 'Davis', 'richard0123@example.com', NULL, 'EMAIL')
;
```

Cars

Id	CustomerId	EmployeeId	Model	Status	Total Cost
1	1	2	Ford F-150	READY	230
2	1	2	Ford F-150	READY	200
3	2	1	Ford Mustang	WAITING	100
4	3	3	Toyota Prius	WORKING	1254

SQL statements to create the table:

```
CREATE TABLE Cars (
  Id INT NOT NULL AUTO_INCREMENT,
  CustomerId INT NOT NULL,
  EmployeeId INT NOT NULL,
  Model varchar(50) NOT NULL,
  Status varchar(25) NOT NULL,
```



```
TotalCost INT NOT NULL,
PRIMARY KEY(Id),
FOREIGN KEY (CustomerId) REFERENCES Customers(Id),
FOREIGN KEY (EmployeeId) REFERENCES Employees(Id)
);

INSERT INTO Cars
([Id], [CustomerId], [EmployeeId], [Model], [Status], [TotalCost])
VALUES
('1', '1', '2', '福特 F-150', '准备好', '230'),
('2', '1', '2', '福特 F-150', '准备好', '200'),
('3', '2', '1', '福特 野马', '等待中', '100'),
('4', '3', '3', '丰田 普锐斯', '工作中', '1254')
;
```

第5.2节：图书馆数据库

在这个图书馆的示例数据库中，我们有作者、图书和图书作者表。

实时示例：[SQL fiddle](#)

作者和图书被称为基础表，因为它们包含关系模型中实际实体的列定义和数据。图书作者被称为关系表，因为该表定义了图书和作者表之间的关系。

表之间的关系

- 每位作者可以有一本或多本书
- 每本书可以有一位或多位作者

作者

(查看表格)

编号	姓名	国家
1	J.D. 塞林格	美国
2	F. 斯科特·菲茨杰拉德	美国
3	简·奥斯汀	英国
4	斯科特·汉斯尔曼	美国
5	杰森·N·盖洛德	美国
6	普拉纳夫·拉斯托吉	印度
7	托德·米兰达	美国
8	克里斯蒂安·温茨	美国

创建表的SQL语句：

```
创建表 Authors (
  Id INT 非空 自动递增,
  Name VARCHAR(70) 非空,
  Country VARCHAR(100) 非空,
  主键(Id)
);

插入到 Authors
```

```
TotalCost INT NOT NULL,
PRIMARY KEY(Id),
FOREIGN KEY (CustomerId) REFERENCES Customers(Id),
FOREIGN KEY (EmployeeId) REFERENCES Employees(Id)
);

INSERT INTO Cars
([Id], [CustomerId], [EmployeeId], [Model], [Status], [TotalCost])
VALUES
('1', '1', '2', 'Ford F-150', 'READY', '230'),
('2', '1', '2', 'Ford F-150', 'READY', '200'),
('3', '2', '1', 'Ford Mustang', 'WAITING', '100'),
('4', '3', '3', 'Toyota Prius', 'WORKING', '1254')
;
```

Section 5.2: Library Database

In this example database for a library, we have *Authors*, *Books* and *BooksAuthors* tables.

Live example: [SQL fiddle](#)

Authors and *Books* are known as **base tables**, since they contain column definition and data for the actual entities in the relational model. *BooksAuthors* is known as the **relationship table**, since this table defines the relationship between the *Books* and *Authors* table.

Relationships between tables

- Each author can have 1 or more books
- Each book can have 1 or more authors

Authors

(view table)

Id	Name	Country
1	J.D. Salinger	USA
2	F. Scott. Fitzgerald	USA
3	Jane Austen	UK
4	Scott Hanselman	USA
5	Jason N. Gaylord	USA
6	Pranav Rastogi	India
7	Todd Miranda	USA
8	Christian Wenz	USA

SQL to create the table:

```
CREATE TABLE Authors (
  Id INT NOT NULL AUTO_INCREMENT,
  Name VARCHAR(70) NOT NULL,
  Country VARCHAR(100) NOT NULL,
  PRIMARY KEY(Id)
);

INSERT INTO Authors
```

```
(Name, Country)
值
('J.D. 塞林格', '美国'),
('F. 斯科特·菲茨杰拉德', '美国'),
('简·奥斯汀', '英国'),
('斯科特·汉斯尔曼', '美国'),
('杰森·N·盖洛德', '美国'),
('普拉纳夫·拉斯托吉', '印度'),
('托德·米兰达', '美国'),
('克里斯蒂安·温茨', '美国')
;
```

书籍

(查看表格)

编号	标题
1	麦田里的守望者
2	九个故事
3	弗兰妮与祖伊
4	了不起的盖茨比
5	夜色温柔
6	傲慢与偏见
7	专业ASP.NET 4.5 C#与VB版

创建表的SQL语句：

```
CREATE TABLE Books (
    Id INT NOT NULL AUTO_INCREMENT,
    Title VARCHAR(50) NOT NULL,
    PRIMARY KEY(Id)
);

INSERT INTO Books
(Id, Title)
值
(1, '麦田里的守望者'),
(2, '九个故事'),
(3, 《弗兰妮与祖伊》),
(4, 《了不起的盖茨比》),
(5, 《夜色温柔》),
(6, 《傲慢与偏见》),
(7, 《专业ASP.NET 4.5 C#与VB编程》)
;
```

书籍作者

(查看表格)

书籍编号	作者编号
1	1
2	1
3	1
4	2
5	2

```
(Name, Country)
VALUES
('J.D. Salinger', 'USA'),
('F. Scott. Fitzgerald', 'USA'),
('Jane Austen', 'UK'),
('Scott Hanselman', 'USA'),
('Jason N. Gaylord', 'USA'),
('Pranav Rastogi', 'India'),
('Todd Miranda', 'USA'),
('Christian Wenz', 'USA')
;
```

Books

(view table)

Id	Title
1	The Catcher in the Rye
2	Nine Stories
3	Franny and Zooey
4	The Great Gatsby
5	Tender id the Night
6	Pride and Prejudice
7	Professional ASP.NET 4.5 in C# and VB

SQL to create the table:

```
CREATE TABLE Books (
    Id INT NOT NULL AUTO_INCREMENT,
    Title VARCHAR(50) NOT NULL,
    PRIMARY KEY(Id)
);

INSERT INTO Books
(Id, Title)
VALUES
(1, 'The Catcher in the Rye'),
(2, 'Nine Stories'),
(3, 'Franny and Zooey'),
(4, 'The Great Gatsby'),
(5, 'Tender id the Night'),
(6, 'Pride and Prejudice'),
(7, 'Professional ASP.NET 4.5 in C# and VB')
;
```

BooksAuthors

(view table)

BookId	AuthorId
1	1
2	1
3	1
4	2
5	2

6 3
7 4
7 5
7 6
7 7
7 8

创建表的SQL语句：

```
创建表 书籍作者 (  
    作者编号 整数 非空,  
    书籍编号 整数 非空,  
    外键 (作者编号) 引用 作者(编号),  
    外键 (书籍编号) 引用 书籍(编号)  
);  
  
插入到 书籍作者  
    (书籍编号, 作者编号)  
值  
    (1, 1),  
    (2, 1),  
    (3, 1),  
    (4, 2),  
    (5, 2),  
    (6, 3),  
    (7, 4),  
    (7, 5),  
    (7, 6),  
    (7, 7),  
    (7, 8)  
;
```

示例

查看所有作者（[查看实时示例](#)）：

```
SELECT * FROM Authors;
```

查看所有书名（[查看实时示例](#)）：

```
SELECT * FROM Books;
```

查看所有书籍及其作者（[查看实时示例](#)）：

```
SELECT  
ba.AuthorId,  
    a.Name 作者名,  
    ba.BookId,  
b.Title 书名  
FROM BooksAuthors ba  
    INNER JOIN Authors a ON a.id = ba.authorid  
    INNER JOIN Books b ON b.id = ba.bookid  
;
```

6 3
7 4
7 5
7 6
7 7
7 8

SQL to create the table:

```
CREATE TABLE BooksAuthors (  
    AuthorId INT NOT NULL,  
    BookId INT NOT NULL,  
    FOREIGN KEY (AuthorId) REFERENCES Authors(Id),  
    FOREIGN KEY (BookId) REFERENCES Books(Id)  
);  
  
INSERT INTO BooksAuthors  
    (BookId, AuthorId)  
VALUES  
    (1, 1),  
    (2, 1),  
    (3, 1),  
    (4, 2),  
    (5, 2),  
    (6, 3),  
    (7, 4),  
    (7, 5),  
    (7, 6),  
    (7, 7),  
    (7, 8)  
;
```

Examples

View all authors ([view live example](#)):

```
SELECT * FROM Authors;
```

View all book titles ([view live example](#)):

```
SELECT * FROM Books;
```

View all books and their authors ([view live example](#)):

```
SELECT  
    ba.AuthorId,  
    a.Name AuthorName,  
    ba.BookId,  
    b.Title BookTitle  
FROM BooksAuthors ba  
    INNER JOIN Authors a ON a.id = ba.authorid  
    INNER JOIN Books b ON b.id = ba.bookid  
;
```

第5.3节：国家表

在此示例中，我们有一个Countries表。国家表有多种用途，尤其是在涉及货币和汇率的金融应用中。

实时示例：[SQL fiddle](#)

一些市场数据软件应用如彭博社（Bloomberg）和路透社（Reuters）要求您向其API提供2位或3位的国家代码以及货币代码。因此，此示例表包含了2位ISO代码列和3位ISO3代码列。

国家

(查看表格)

编号	ISO	ISO3	数字代码	国家名称	首都	洲代码	货币代码
1	澳大利亚	AU	AUS	36	澳大利亚	堪培拉	大洋洲 澳元
2	德国	DE	DEU	276	德国	柏林	欧盟 欧元
2	IN	IND	356	印度	新德里	AS	印度卢比
3	LA	LAO	418	老挝	万象	AS	老挝基普
4	美国	美国	840	美国	华盛顿	NA	美元
5	津巴布韦	ZW	ZWE	716	津巴布韦	哈拉雷	非洲 津巴布韦元

创建表的SQL语句：

```
CREATE TABLE 国家 (
    编号 INT NOT NULL AUTO_INCREMENT,
    ISO VARCHAR(2) NOT NULL,
    ISO3 VARCHAR(3) NOT NULL,
    数字ISO INT NOT NULL,
    国家名称 VARCHAR(64) NOT NULL,
    首都 VARCHAR(64) NOT NULL,
    大陆代码 VARCHAR(2) NOT NULL,
    货币代码 VARCHAR(3) NOT NULL,
    PRIMARY KEY(编号)
)

INSERT INTO 国家 (ISO, ISO3, ISONumeric, CountryName, Capital, ContinentCode, CurrencyCode)
VALUES
    ('AU', 'AUS', 36, '澳大利亚', '堪培拉', 'OC', 'AUD'),
    ('DE', 'DEU', 276, '德国', '柏林', 'EU', 'EUR'),
    ('IN', 'IND', 356, '印度', '新德里', 'AS', 'INR'),
    ('LA', 'LAO', 418, '老挝', '万象', 'AS', 'LAK'),
    ('US', 'USA', 840, '美国', '华盛顿', 'NA', 'USD'),
    ('ZW', 'ZWE', 716, '津巴布韦', '哈拉雷', 'AF', 'ZWL')
;
```

Section 5.3: Countries Table

In this example, we have a **Countries** table. A table for countries has many uses, especially in Financial applications involving currencies and exchange rates.

Live example: [SQL fiddle](#)

Some Market data software applications like Bloomberg and Reuters require you to give their API either a 2 or 3 character country code along with the currency code. Hence this example table has both the 2-character ISO code column and the 3 character ISO3 code columns.

Countries

(view table)

Id	ISO	ISO3	ISONumeric	CountryName	Capital	ContinentCode	CurrencyCode
1	AU	AUS	36	Australia	Canberra	OC	AUD
2	DE	DEU	276	Germany	Berlin	EU	EUR
2	IN	IND	356	India	New Delhi	AS	INR
3	LA	LAO	418	Laos	Vientiane	AS	LAK
4	US	USA	840	United States	Washington	NA	USD
5	ZW	ZWE	716	Zimbabwe	Harare	AF	ZWL

SQL to create the table:

```
CREATE TABLE Countries (
    Id INT NOT NULL AUTO_INCREMENT,
    ISO VARCHAR(2) NOT NULL,
    ISO3 VARCHAR(3) NOT NULL,
    ISONumeric INT NOT NULL,
    CountryName VARCHAR(64) NOT NULL,
    Capital VARCHAR(64) NOT NULL,
    ContinentCode VARCHAR(2) NOT NULL,
    CurrencyCode VARCHAR(3) NOT NULL,
    PRIMARY KEY(Id)
)

INSERT INTO Countries
    (ISO, ISO3, ISONumeric, CountryName, Capital, ContinentCode, CurrencyCode)
VALUES
    ('AU', 'AUS', 36, 'Australia', 'Canberra', 'OC', 'AUD'),
    ('DE', 'DEU', 276, 'Germany', 'Berlin', 'EU', 'EUR'),
    ('IN', 'IND', 356, 'India', 'New Delhi', 'AS', 'INR'),
    ('LA', 'LAO', 418, 'Laos', 'Vientiane', 'AS', 'LAK'),
    ('US', 'USA', 840, 'United States', 'Washington', 'NA', 'USD'),
    ('ZW', 'ZWE', 716, 'Zimbabwe', 'Harare', 'AF', 'ZWL')
;
```


第6章：SELECT

SELECT语句是大多数SQL查询的核心。它定义了查询应返回的结果集，几乎总是与FROM子句一起使用，后者定义了应查询数据库的哪部分内容。

第6.1节：使用通配符选择查询中的所有列

考虑一个包含以下两个表的数据库。

员工表：

编号	名字	姓氏	部门编号
1	詹姆斯	史密斯	3
2	约翰	约翰逊	4

部门表：

编号	名称
1	销售部
2	市场部
3	财务部
4	IT

简单的选择语句

* 是通配符字符，用于选择表中所有可用的列。

当用作明确列名的替代时，它返回查询所选择的所有表中的所有列FROM。此效果适用于查询通过JOIN子句访问的所有表。

考虑以下查询：

```
SELECT * FROM Employees
```

它将返回Employees表中所有行的所有字段：

编号	名字	姓氏	部门编号
1	詹姆斯	史密斯	3
2	约翰	约翰逊	4

点符号

要从特定表中选择所有值，可以使用通配符字符结合点符号应用于该表。

考虑以下查询：

```
SELECT
Employees.*,
    Departments.Name
FROM
Employees
JOIN
```

Chapter 6: SELECT

The SELECT statement is at the heart of most SQL queries. It defines what result set should be returned by the query, and is almost always used in conjunction with the FROM clause, which defines what part(s) of the database should be queried.

Section 6.1: Using the wildcard character to select all columns in a query

Consider a database with the following two tables.

Employees table:

Id	FName	LName	DeptId
1	James	Smith	3
2	John	Johnson	4

Departments table:

Id	Name
1	Sales
2	Marketing
3	Finance
4	IT

Simple select statement

* is the **wildcard character** used to select all available columns in a table.

When used as a substitute for explicit column names, it returns all columns in all tables that a query is selectingFROM. This effect applies to **all tables** the query accesses through its JOIN clauses.

Consider the following query:

```
SELECT * FROM Employees
```

It will return all fields of all rows of the Employees table:

Id	FName	LName	DeptId
1	James	Smith	3
2	John	Johnson	4

Dot notation

To select all values from a specific table, the wildcard character can be applied to the table with *dot notation*.

Consider the following query:

```
SELECT
    Employees.*,
    Departments.Name
FROM
Employees
JOIN
```

```
Departments
ON Departments.Id = Employees.DeptId
```

这将返回一个数据集，包含Employee表中的所有字段，后面跟着Departments表中的Name字段：

	Id	FName	LName	DeptId	Name
1		詹姆斯	史密斯	3	财务部
2		约翰	约翰逊	4	IT

使用警告

通常建议在生产代码中尽可能避免使用*, 因为它可能导致多种潜在问题，包括：

1. 由于数据库引擎读取不需要的数据并将其传输到前端代码，导致过多的IO、网络负载、内存使用等。这在存在大字段（如用于存储长备注或附件文件的字段）时尤其值得关注。
2. 如果数据库需要将内部结果临时写入磁盘以处理比SELECT <columns> FROM <table>更复杂的查询，则会产生额外的IO负载。
3. 如果某些不需要的列是：
 - 支持计算列的数据库中的计算列在从视图中选择时，
 - 查询优化器本可以优化掉的表/视图中的列
4. 如果以后向表和视图中添加列，可能导致列名歧义，从而引发意外错误。例如SELECT * FROM orders JOIN people ON people.id = orders.personid ORDERBY displayname —— 如果在orders表中添加了一个名为displayname的列，以使用户为订单赋予有意义的名称，则该列名将在输出中出现两次，导致ORDER BY子句歧义，可能引发错误（在最新的MS SQL Server版本中显示“ambiguous column name”），如果不是这种情况，您的应用程序代码可能会开始显示订单名称，而本应显示的是人员名称，因为新列是返回的第一个同名列，等等。

考虑上述警告，何时可以使用*？

虽然在生产代码中最好避免使用*，但在对数据库进行手动查询以进行调查或原型开发时，使用*作为简写是可以接受的。

有时，您的应用程序中的设计决策使其不可避免（在这种情况下，尽可能优先使用 tablealias.*而不是*）。

当使用EXISTS时，例如SELECT A.col1, A.Col2 FROM A WHERE EXISTS (SELECT * FROM B where A.ID =B.A_ID)，我们并没有从B中返回任何数据。因此，连接是不必要的，且引擎知道不会返回B中的任何值，因此使用*不会有性能损失。同样，COUNT(*)也是可以的，因为它实际上并不返回任何列，只需要读取和处理用于过滤的那些数据。

第6.2节：使用列别名的SELECT

列别名主要用于缩短代码并使列名更易读。

代码变得更短，因为可以避免使用冗长的表名和不必要的列标识（例如，表中可能有两个ID，但语句中只使用一个）。结合表别名，这允许你在数据库结构中使用更长的描述性名称，同时保持针对该结构的查询简洁。

此外，有时它们是必须的，例如在视图中，为了给计算结果命名。

```
Departments
ON Departments.Id = Employees.DeptId
```

This will return a data set with all fields on the Employee table, followed by just the Name field in the Departments table:

	Id	FName	LName	DeptId	Name
1		James	Smith	3	Finance
2		John	Johnson	4	IT

Warnings Against Use

It is generally advised that using * is avoided in production code where possible, as it can cause a number of potential problems including:

1. Excess IO, network load, memory use, and so on, due to the database engine reading data that is not needed and transmitting it to the front-end code. This is particularly a concern where there might be large fields such as those used to store long notes or attached files.
2. Further excess IO load if the database needs to spool internal results to disk as part of the processing for a query more complex than SELECT <columns> FROM <table>.
3. Extra processing (and/or even more IO) if some of the unneeded columns are:
 - computed columns in databases that support them
 - in the case of selecting from a view, columns from a table/view that the query optimiser could otherwise optimise out
4. The potential for unexpected errors if columns are added to tables and views later that results ambiguous column names. For example SELECT * FROM orders JOIN people ON people.id = orders.personid ORDER BY displayname - if a column column called displayname is added to the orders table to allow users to give their orders meaningful names for future reference then the column name will appear twice in the output so the ORDER BY clause will be ambiguous which may cause errors ("ambiguous column name" in recent MS SQL Server versions), and if not in this example your application code might start displaying the order name where the person name is intended because the new column is the first of that name returned, and so on.

When Can You Use *, Bearing The Above Warning In Mind?

While best avoided in production code, using * is fine as a shorthand when performing manual queries against the database for investigation or prototype work.

Sometimes design decisions in your application make it unavoidable (in such circumstances, prefer tablealias.* over just * where possible).

When using EXISTS, such as SELECT A.col1, A.Col2 FROM A WHERE EXISTS (SELECT * FROM B where A.ID = B.A_ID), we are not returning any data from B. Thus a join is unnecessary, and the engine knows no values from B are to be returned, thus no performance hit for using *. Similarly COUNT(*) is fine as it also doesn't actually return any of the columns, so only needs to read and process those that are used for filtering purposes.

Section 6.2: SELECT Using Column Aliases

Column aliases are used mainly to shorten code and make column names more readable.

Code becomes shorter as long table names and unnecessary identification of columns (*e.g., there may be 2 IDs in the table, but only one is used in the statement*) can be avoided. Along with table aliases this allows you to use longer descriptive names in your database structure while keeping queries upon that structure concise.

Furthermore they are sometimes *required*, for instance in views, in order to name computed outputs.

所有版本的SQL

所有版本的SQL都可以使用双引号 (") 创建别名。

```
SELECT
FName AS "First Name",
      MName AS "Middle Name",
      LName AS "Last Name"
FROM Employees
```

不同版本的SQL

您可以使用单引号 (')、双引号 (") 和方括号 ([]) 在 Microsoft SQL Server 中创建别名。

```
SELECT
FName AS "First Name",
      MName AS '中间名',
      LName AS [姓氏]
FROM 员工表
```

两者都会产生以下结果：

名字	中间名	姓氏
詹姆斯	约翰	史密斯
约翰	詹姆斯	约翰逊
迈克尔	马库斯	威廉姆斯

该语句将返回带有指定名称（别名）的FName和LName列。这是通过使用AS操作符后跟别名，或者直接在列名后写别名来实现的。这意味着以下查询与上述查询具有相同的结果。

```
SELECT
FName "名",
      MName "中间名",
      LName "姓"
FROM 员工
```

名字	中间名	姓氏
詹姆斯	约翰	史密斯
约翰	詹姆斯	约翰逊
迈克尔	马库斯	威廉姆斯

但是，显式版本（即使用AS操作符）更易读。

如果别名是单个非保留字的单词，我们可以不使用单引号、双引号或括号来写它：

```
SELECT
FName AS FirstName,
      LName AS LastName
FROM 员工
```

FirstName	LastName
詹姆斯	史密斯
约翰	约翰逊
迈克尔	威廉姆斯

All versions of SQL

Aliases can be created in all versions of SQL using double quotes (").

```
SELECT
      FName AS "First Name",
      MName AS "Middle Name",
      LName AS "Last Name"
FROM Employees
```

Different Versions of SQL

You can use single quotes ('), double quotes (") and square brackets ([]) to create an alias in Microsoft SQL Server.

```
SELECT
      FName AS "First Name",
      MName AS 'Middle Name',
      LName AS [Last Name]
FROM Employees
```

Both will result in:

First Name	Middle Name	Last Name
James	John	Smith
John	James	Johnson
Michael	Marcus	Williams

This statement will return FName and LName columns with a given name (an alias). This is achieved using the AS operator followed by the alias, or simply writing alias directly after the column name. This means that the following query has the same outcome as the above.

```
SELECT
      FName "First Name",
      MName "Middle Name",
      LName "Last Name"
FROM Employees
```

First Name	Middle Name	Last Name
James	John	Smith
John	James	Johnson
Michael	Marcus	Williams

However, the explicit version (i.e., using the AS operator) is more readable.

If the alias has a single word that is not a reserved word, we can write it without single quotes, double quotes or brackets:

```
SELECT
      FName AS FirstName,
      LName AS LastName
FROM Employees
```

FirstName	LastName
James	Smith
John	Johnson
Michael	Williams

在包括 MS SQL Server 在内的系统中，还可以使用另一种变体<alias> = <column-or-calculation>, 例如：

```
SELECT FullName = FirstName + ' ' + LastName,
       Addr1     = FullStreetAddress,
       Addr2     = 城镇名
来自 客户详情
```

等同于：

```
选择 名字 + ' ' + 姓氏 作为 全名
      完整街道地址      作为 地址1,
      城镇名              作为 地址2
来自 客户详情
```

两者都会产生以下结果：

全名	地址1	地址2
詹姆斯·史密斯	123 任意街	镇维尔
约翰·约翰逊	668 我的路	任意镇
迈克尔·威廉姆斯	999 高端大道	威廉斯堡

有些人觉得使用=代替As更容易阅读，尽管许多人不推荐这种格式，主要是因为它不是标准格式，因此并非所有数据库都广泛支持。它可能会与=字符的其他用法产生混淆。

所有版本的SQL

此外，如果你需要使用保留字，可以使用方括号或引号进行转义：

```
SELECT
FName as "SELECT",
MName as "FROM",
LName as "WHERE"
FROM Employees
```

不同版本的SQL

同样，你可以用各种不同的方法在MSSQL中转义关键字：

```
SELECT
FName AS "SELECT",
MName AS 'FROM',
LName AS [WHERE]
FROM Employees

SELECT FROM WHERE
詹姆斯约翰      史密斯
约翰      詹姆斯约翰逊
迈克尔·马库斯·威廉姆斯
```

此外，列别名可以在同一查询的任何最终子句中使用，例如ORDER BY：

```
SELECT
FName 作为 FirstName,
LName 作为 LastName
FROM
```

A further variation available in MS SQL Server amongst others is <alias> = <column-or-calculation>, for instance:

```
SELECT FullName = FirstName + ' ' + LastName,
       Addr1     = FullStreetAddress,
       Addr2     = TownName
FROM CustomerDetails
```

which is equivalent to:

```
SELECT FirstName + ' ' + LastName As FullName
       FullStreetAddress      As Addr1,
       TownName                As Addr2
FROM CustomerDetails
```

Both will result in:

FullName	Addr1	Addr2
James Smith	123 AnyStreet	TownVille
John Johnson	668 MyRoad	Anytown
Michael Williams	999 High End Dr	Williamsburgh

Some find using = instead of As easier to read, though many recommend against this format, mainly because it is not standard so not widely supported by all databases. It may cause confusion with other uses of the = character.

All Versions of SQL

Also, if you *need* to use reserved words, you can use brackets or quotes to escape:

```
SELECT
FName as "SELECT",
MName as "FROM",
LName as "WHERE"
FROM Employees
```

Different Versions of SQL

Likewise, you can escape keywords in MSSQL with all different approaches:

```
SELECT
FName AS "SELECT",
MName AS 'FROM',
LName AS [WHERE]
FROM Employees

SELECT FROM WHERE
James John Smith
John James Johnson
Michael Marcus Williams
```

Also, a column alias may be used any of the final clauses of the same query, such as an ORDER BY:

```
SELECT
FName AS FirstName,
LName AS LastName
FROM
```



```
员工
ORDER BY
LastName 降序
```

但是，你不能使用

```
SELECT
FName 作为 SELECT,
      LName 作为 FROM
FROM
员工
ORDER BY
LastName 降序
```

来创建这些保留字（SELECT 和 FROM）的别名。

这将在执行时导致大量错误。

第6.3节：选择单个列

```
SELECT
电话号码,
      电子邮箱,
PreferredContact
FROM Customers
```

该语句将返回Customers表中所有行的PhoneNumber、Email和PreferredContact列。同时，列的返回顺序与SELECT子句中出现的顺序一致。

结果将是：

PhoneNumber	Email	PreferredContact
3347927472	william.jones@example.com	PHONE
2137921892	dmiller@example.net	EMAIL
NULL	richard0123@example.com	EMAIL

如果多个表被连接在一起，可以通过在列名前指定表名来选择特定表的列：[表名].[列名]

```
SELECT
Customers.PhoneNumber,
      Customers.Email,
Customers.PreferredContact,
Orders.Id AS OrderId
FROM
Customers
LEFT JOIN
Orders ON Orders.CustomerId = Customers.Id
```

*AS OrderId 表示 Orders 表的 Id 字段将作为名为 OrderId 的列返回。有关更多信息，请参见使用列别名进行选择。

为了避免使用冗长的表名，可以使用表别名。这减轻了在连接中为每个选择的字段编写长表名的负担。如果执行自连接（同一张表的两个实例之间的连接），则必须使用表别名来区分表。我们可以写成表别名如 Customers c 或

Customers AS c。这里 c 作为 Customers 的别名，我们可以这样选择，比如 Email：c.Email。

```
Employees
ORDER BY
LastName DESC
```

However, you may *not* use

```
SELECT
      FName AS SELECT,
      LName AS FROM
FROM
Employees
ORDER BY
LastName DESC
```

To create an alias from these reserved words (SELECT and FROM).

This will cause numerous errors on execution.

Section 6.3: Select Individual Columns

```
SELECT
      PhoneNumber,
      Email,
PreferredContact
FROM Customers
```

This statement will return the columns PhoneNumber, Email, and PreferredContact from all rows of the Customers table. Also the columns will be returned in the sequence in which they appear in the SELECT clause.

The result will be:

PhoneNumber	Email	PreferredContact
3347927472	william.jones@example.com	PHONE
2137921892	dmiller@example.net	EMAIL
NULL	richard0123@example.com	EMAIL

If multiple tables are joined together, you can select columns from specific tables by specifying the table name before the column name: [table_name].[column_name]

```
SELECT
      Customers.PhoneNumber,
      Customers.Email,
Customers.PreferredContact,
Orders.Id AS OrderId
FROM
Customers
LEFT JOIN
Orders ON Orders.CustomerId = Customers.Id
```

*AS OrderId means that the Id field of Orders table will be returned as a column named OrderId. See selecting with column alias for further information.

To avoid using long table names, you can use table aliases. This mitigates the pain of writing long table names for each field that you select in the joins. If you are performing a self join (a join between two instances of the *same* table), then you must use table aliases to distinguish your tables. We can write a table alias like Customers c or Customers AS c. Here c works as an alias for Customers and we can select let's say Email like this: c.Email.

```
SELECT
c.PhoneNumber,
  c.Email,
c.PreferredContact,
  o.Id AS OrderId
FROM
Customers c
LEFT JOIN
Orders o ON o.CustomerId = c.Id
```

第6.4节：选择指定数量的记录

SQL [2008标准](#)定义了FETCH FIRST子句，用于限制返回的记录数。

```
SELECT Id, ProductName, UnitPrice, Package
FROM Product
ORDER BY UnitPrice DESC
FETCH FIRST 10 ROWS ONLY
```

该标准仅被某些关系数据库管理系统的较新版本支持。其他系统提供了厂商特定的非标准语法。Progress OpenEdge 11.x也支持FETCH FIRST <n> ROWS ONLY语法。

此外，OFFSET <m> ROWS在FETCH FIRST <n> ROWS ONLY之前允许跳过若干行后再获取行。

```
SELECT Id, ProductName, UnitPrice, Package
FROM Product
按 单价 降序排列
偏移 5 行
仅获取前 10 行
```

以下查询在 SQL Server 和 MS Access 中受支持：

```
选择前 10 个 Id, 产品名称, 单价, 包装
来自 产品
按 单价 降序排列
```

在 MySQL 或 PostgreSQL 中执行相同操作必须使用 LIMIT 关键字：

```
SELECT Id, ProductName, UnitPrice, Package
FROM Product
按 单价 降序排列
限制 10
```

在 Oracle 中可以使用 ROWNUM 实现相同功能：

```
SELECT Id, ProductName, UnitPrice, Package
FROM Product
条件 ROWNUM <= 10
按 单价 降序排列
```

结果: 10 条记录。

编号	产品名称	单价	包装
38	Côte de Blaye	263.50	12 - 75 毫升瓶
29	Thüringer 烤香肠	123.79	50 袋 x 30 根香肠
9	三宫神户牛肉	97.00	18 - 500 克包装

```
SELECT
  c.PhoneNumber,
  c.Email,
c.PreferredContact,
  o.Id AS OrderId
FROM
  Customers c
LEFT JOIN
  Orders o ON o.CustomerId = c.Id
```

Section 6.4: Selecting specified number of records

The [SQL 2008 standard](#) defines the FETCH FIRST clause to limit the number of records returned.

```
SELECT Id, ProductName, UnitPrice, Package
FROM Product
ORDER BY UnitPrice DESC
FETCH FIRST 10 ROWS ONLY
```

This standard is only supported in recent versions of some RDMSs. Vendor-specific non-standard syntax is provided in other systems. Progress OpenEdge 11.x also supports the FETCH FIRST <n> ROWS ONLY syntax.

Additionally, OFFSET <m> ROWS before FETCH FIRST <n> ROWS ONLY allows skipping rows before fetching rows.

```
SELECT Id, ProductName, UnitPrice, Package
FROM Product
ORDER BY UnitPrice DESC
OFFSET 5 ROWS
FETCH FIRST 10 ROWS ONLY
```

The following query is supported in SQL Server and MS Access:

```
SELECT TOP 10 Id, ProductName, UnitPrice, Package
FROM Product
ORDER BY UnitPrice DESC
```

To do the same in MySQL or PostgreSQL the LIMIT keyword must be used:

```
SELECT Id, ProductName, UnitPrice, Package
FROM Product
ORDER BY UnitPrice DESC
LIMIT 10
```

In Oracle the same can be done with ROWNUM:

```
SELECT Id, ProductName, UnitPrice, Package
FROM Product
WHERE ROWNUM <= 10
ORDER BY UnitPrice DESC
```

Results: 10 records.

Id	ProductName	UnitPrice	Package
38	Côte de Blaye	263.50	12 - 75 cl bottles
29	Thüringer Rostbratwurst	123.79	50 bags x 30 sausgs.
9	Mishi Kobe Niku	97.00	18 - 500 g pkgs.

20	罗德尼爵士果酱	81.00	30 礼盒
18	卡纳封老虎	62.50	16 公斤包装
59	库尔达沃乳酪	55.00	5 公斤包装
51	曼吉姆干苹果	53.00	50 - 300 克包装
62	糖馅饼	49.30	48 个馅饼
43	怡保咖啡	46.00	16 - 500 克罐装
28	Rössle 酸菜	45.60	25 - 825 克罐装

供应商细节：

需要注意的是，Microsoft SQL 中的 TOP 在 WHERE 子句之后执行，如果表中存在指定数量的结果，则会返回这些结果，而 ROWNUM 是作为 WHERE 子句的一部分工作，因此如果在表开头指定的行数中不存在其他条件，则会得到零结果，即使表中可能还有其他结果。

第6.5节：带条件的选择

带有 WHERE 子句的 SELECT 的基本语法是：

```
SELECT column1, column2, columnN
FROM table_name
WHERE [条件]
```

该[条件]可以是任何SQL表达式，使用比较或逻辑运算符如 >, <, =, <>, >=, <=, LIKE, NOT, IN, BETWEEN 等指定。

下面的语句返回表 'Cars' 中状态列为 'READY' 的所有列：

```
SELECT * FROM Cars WHERE status = 'READY'
```

更多示例请参见 WHERE 和 HAVING。

第6.6节：使用 CASE 进行选择

当结果需要“即时”应用某些逻辑时，可以使用 CASE 语句来实现。

```
SELECT CASE WHEN Col1 < 50 THEN 'under' ELSE 'over' END threshold
FROM TableName
```

也可以链式使用

```
SELECT
CASE WHEN Col1 < 50 THEN 'under'
      WHEN Col1 > 50 AND Col1 <100 THEN 'between'
      ELSE 'over'
END 阈值
FROM 表名
```

也可以在另一个CASE语句内部使用CASE

```
SELECT
CASE WHEN Col1 < 50 THEN 'under'
      ELSE
CASE WHEN Col1 > 50 AND Col1 <100 THEN Col1
      ELSE 'over' END
END threshold
```

20	Sir Rodney's Marmalade	81.00	30 gift boxes
18	Carnarvon Tigers	62.50	16 kg pkg.
59	Raclette Courdavault	55.00	5 kg pkg.
51	Manjimup Dried Apples	53.00	50 - 300 g pkgs.
62	Tarte au sucre	49.30	48 pies
43	Ipoh Coffee	46.00	16 - 500 g tins
28	Rössle Sauerkraut	45.60	25 - 825 g cans

Vendor Nuances:

It is important to note that the TOP in Microsoft SQL operates after the WHERE clause and will return the specified number of results if they exist anywhere in the table, while ROWNUM works as part of the WHERE clause so if other conditions do not exist in the specified number of rows at the beginning of the table, you will get zero results when there could be others to be found.

Section 6.5: Selecting with Condition

The basic syntax of SELECT with WHERE clause is:

```
SELECT column1, column2, columnN
FROM table_name
WHERE [condition]
```

The [condition] can be any SQL expression, specified using comparison or logical operators like >, <, =, <>, >=, <=, LIKE, NOT, IN, BETWEEN etc.

The following statement returns all columns from the table 'Cars' where the status column is 'READY':

```
SELECT * FROM Cars WHERE status = 'READY'
```

See WHERE and HAVING for more examples.

Section 6.6: Selecting with CASE

When results need to have some logic applied 'on the fly' one can use CASE statement to implement it.

```
SELECT CASE WHEN Col1 < 50 THEN 'under' ELSE 'over' END threshold
FROM TableName
```

also can be chained

```
SELECT
CASE WHEN Col1 < 50 THEN 'under'
      WHEN Col1 > 50 AND Col1 <100 THEN 'between'
      ELSE 'over'
END threshold
FROM TableName
```

one also can have CASE inside another CASE statement

```
SELECT
CASE WHEN Col1 < 50 THEN 'under'
      ELSE
CASE WHEN Col1 > 50 AND Col1 <100 THEN Col1
      ELSE 'over' END
END threshold
```

```
FROM TableName
```

第6.7节：选择以保留关键字命名的列

当列名与保留关键字相同时，标准SQL要求将其用双引号括起来：

```
SELECT
"ORDER",
  ID
FROM ORDERS
```

注意，这会使列名区分大小写。

一些数据库管理系统（DBMS）有专有的引用名称方式。例如，SQL Server 使用方括号来实现此目的：

```
SELECT
  [Order],
  ID
FROM ORDERS
```

而 MySQL（和 MariaDB）默认使用反引号：

```
SELECT
`Order`,
  id
FROM orders
```

第6.8节：使用表别名进行选择

```
SELECT e.Fname, e.LName
FROM Employees e
```

Employees 表在表名后直接赋予别名“e”。这有助于消除在多个表具有相同字段名且需要明确指定要返回哪个表的数据时的歧义。

```
SELECT e.Fname, e.LName, m.Fname AS ManagerFirstName
FROM Employees e
      JOIN Managers m ON e.ManagerId = m.Id
```

请注意，一旦定义了别名，就不能再使用规范的表名了。即，

```
SELECT e.Fname, Employees.LName, m.Fname AS ManagerFirstName
FROM Employees e
      JOIN Managers m ON e.ManagerId = m.Id
```

会抛出错误。

值得注意的是，表别名——更正式地称为“范围变量”——被引入SQL语言中，以解决由INNER JOIN引起的重复列问题。1992年SQL标准通过引入NATURAL JOIN（在MySQL、PostgreSQL和Oracle中实现，但SQL Server尚未实现）修正了这一早期设计缺陷，其结果永远不会有重复的列名。上述示例有趣之处在于，表是基于

```
FROM TableName
```

Section 6.7: Select columns which are named after reserved keywords

When a column name matches a reserved keyword, standard SQL requires that you enclose it in double quotation marks:

```
SELECT
  "ORDER",
  ID
FROM ORDERS
```

Note that it makes the column name case-sensitive.

Some DBMSes have proprietary ways of quoting names. For example, SQL Server uses square brackets for this purpose:

```
SELECT
  [Order],
  ID
FROM ORDERS
```

while MySQL (and MariaDB) by default use backticks:

```
SELECT
  `Order`,
  id
FROM orders
```

Section 6.8: Selecting with table alias

```
SELECT e.Fname, e.LName
FROM Employees e
```

The Employees table is given the alias 'e' directly after the table name. This helps remove ambiguity in scenarios where multiple tables have the same field name and you need to be specific as to which table you want to return data from.

```
SELECT e.Fname, e.LName, m.Fname AS ManagerFirstName
FROM Employees e
      JOIN Managers m ON e.ManagerId = m.Id
```

Note that once you define an alias, you can't use the canonical table name anymore. i.e.,

```
SELECT e.Fname, Employees.LName, m.Fname AS ManagerFirstName
FROM Employees e
      JOIN Managers m ON e.ManagerId = m.Id
```

would throw an error.

It is worth noting table aliases -- more formally 'range variables' -- were introduced into the SQL language to solve the problem of duplicate columns caused by **INNER JOIN**. The 1992 SQL standard corrected this earlier design flaw by introducing **NATURAL JOIN** (implemented in MySQL, PostgreSQL and Oracle but not yet in SQL Server), the result of which never has duplicate column names. The above example is interesting in that the tables are joined on

名称不同的列（Id和ManagerId）进行连接，但不应基于同名列（LName、FName）进行连接，因此需要在连接之前对列进行重命名：

```
SELECT Fname, LName, ManagerFirstName
FROM Employees
    NATURAL JOIN
    ( SELECT Id AS ManagerId, Fname AS ManagerFirstName
      FROM Managers ) m;
```

注意，虽然必须为派生表声明别名/范围变量（否则SQL会抛出错误），但实际上在查询中使用它从来没有意义。

第6.9节：带有多个条件的选择

使用AND关键字向查询添加更多条件。

姓名 年龄 性别
山姆18男
约翰21男
鲍勃22男
玛丽23女

```
SELECT name FROM persons WHERE gender = 'M' AND age > 20;
```

这将返回：

姓名
约翰
鲍勃

使用OR关键字

```
SELECT name FROM persons WHERE gender = 'M' OR age < 20;
```

这将返回：

姓名
山姆
约翰
鲍勃

这些关键词可以组合使用，以允许更复杂的条件组合：

```
SELECT name
FROM persons
WHERE (gender = 'M' AND age < 20)
    OR (gender = 'F' AND age > 20);
```

这将返回：

姓名
Sam
Mary

columns with different names (Id and ManagerId) but are not supposed to be joined on the columns with the same name (LName, FName), requiring the renaming of the columns to be performed *before* the join:

```
SELECT Fname, LName, ManagerFirstName
FROM Employees
    NATURAL JOIN
    ( SELECT Id AS ManagerId, Fname AS ManagerFirstName
      FROM Managers ) m;
```

Note that although an alias/range variable must be declared for the dervied table (otherwise SQL will throw an error), it never makes sense to actually use it in the query.

Section 6.9: Selecting with more than 1 condition

The **AND** keyword is used to add more conditions to the query.

Name Age Gender
Sam 18 M
John 21 M
Bob 22 M
Mary 23 F

```
SELECT name FROM persons WHERE gender = 'M' AND age > 20;
```

This will return:

Name
John
Bob

using OR keyword

```
SELECT name FROM persons WHERE gender = 'M' OR age < 20;
```

This will return:

name
Sam
John
Bob

These keywords can be combined to allow for more complex criteria combinations:

```
SELECT name
FROM persons
WHERE (gender = 'M' AND age < 20)
    OR (gender = 'F' AND age > 20);
```

This will return:

name
Sam
Mary

第6.10节：选择时不锁定表

有时当表主要（或仅）用于读取时，索引不再起作用，每一点性能提升都很重要，可以使用不加锁的选择来提高性能。

SQL Server

```
SELECT * FROM TableName WITH (nolock)
```

MySQL

```
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;  
SELECT * FROM TableName;  
SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

Oracle

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;  
SELECT * FROM TableName;
```

DB2

```
SELECT * FROM TableName WITH UR;
```

其中 UR 代表“未提交读取”。

如果在有记录修改的表上使用，可能会产生不可预测的结果。

第6.11节：使用聚合函数进行选择

平均值

AVG() 聚合函数将返回所选值的平均值。

```
SELECT AVG(Salary) FROM Employees
```

聚合函数也可以与where子句结合使用。

```
SELECT AVG(Salary) FROM Employees where DepartmentId = 1
```

聚合函数也可以与group by子句结合使用。

如果员工被划分到多个部门，并且我们想要查找每个部门的平均工资，那么我们可以使用以下查询。

```
SELECT AVG(Salary) FROM Employees GROUP BY DepartmentId
```

最小值

MIN()聚合函数将返回所选值中的最小值。

```
SELECT MIN(Salary) FROM Employees
```

最大值

MAX()聚合函数将返回所选值中的最大值。

```
SELECT MAX(Salary) FROM Employees
```

计数

COUNT() 聚合函数将返回所选值的计数。

```
SELECT Count(*) FROM Employees
```

它也可以与 where 条件结合使用，以获取满足特定条件的行数。

Section 6.10: Selecting without Locking the table

Sometimes when tables are used mostly (or only) for reads, indexing does not help anymore and every little bit counts, one might use selects without LOCK to improve performance.

SQL Server

```
SELECT * FROM TableName WITH (nolock)
```

MySQL

```
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;  
SELECT * FROM TableName;  
SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

Oracle

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;  
SELECT * FROM TableName;
```

DB2

```
SELECT * FROM TableName WITH UR;
```

where UR stands for "uncommitted read".

If used on table that has record modifications going on might have unpredictable results.

Section 6.11: Selecting with Aggregate functions

Average

The **AVG()** aggregate function will return the average of values selected.

```
SELECT AVG(Salary) FROM Employees
```

Aggregate functions can also be combined with the where clause.

```
SELECT AVG(Salary) FROM Employees where DepartmentId = 1
```

Aggregate functions can also be combined with group by clause.

If employee is categorized with multiple department and we want to find avg salary for every department then we can use following query.

```
SELECT AVG(Salary) FROM Employees GROUP BY DepartmentId
```

Minimum

The **MIN()** aggregate function will return the minimum of values selected.

```
SELECT MIN(Salary) FROM Employees
```

Maximum

The **MAX()** aggregate function will return the maximum of values selected.

```
SELECT MAX(Salary) FROM Employees
```

Count

The **COUNT()** aggregate function will return the count of values selected.

```
SELECT Count(*) FROM Employees
```

It can also be combined with where conditions to get the count of rows that satisfy specific conditions.

```
SELECT Count(*) FROM Employees where ManagerId IS NOT NULL
```

也可以指定特定列以获取该列中的值的数量。注意 NULL 值不被计数。

```
Select Count(ManagerId) from Employees
```

计数也可以与 distinct 关键字结合使用以获得不同的计数。

```
Select Count(DISTINCT DepartmentId) from Employees
```

总和

SUM() 聚合函数返回所有行所选值的总和。

```
SELECT SUM(Salary) FROM Employees
```

第6.12节：从列中选择多个值的条件

```
SELECT * FROM Cars WHERE status IN ( 'Waiting', 'Working' )
```

这在语义上等同于

```
SELECT * FROM Cars WHERE ( status = 'Waiting' OR status = 'Working' )
```

即 value IN (<value list>) 是逻辑或（disjunction，逻辑 OR）的简写。

第6.13节：获取行组的聚合结果

基于特定列值计数行数：

```
SELECT category, COUNT(*) AS item_count
FROM item
按类别分组 category;
```

按部门获取平均收入：

```
选择 department, AVG(income)
从 employees
按部门分组 department;
```

重要的是只选择GROUP BY子句中指定的列或与聚合函数一起使用的列。

可以将WHERE子句与GROUP BY一起使用，但WHERE会在分组之前过滤记录：

```
选择 department, AVG(income)
从 employees
WHERE department <> 'ACCOUNTING'
按部门分组 department;
```

如果需要在分组完成后过滤结果，例如，只查看平均收入大于1000的部门，则需要使用HAVING子句：

```
选择 department, AVG(income)
从 employees
WHERE department <> 'ACCOUNTING'
按部门分组 department
HAVING avg(income) > 1000;
```

```
SELECT Count(*) FROM Employees where ManagerId IS NOT NULL
```

Specific columns can also be specified to get the number of values in the column. Note that NULL values are not counted.

```
Select Count(ManagerId) from Employees
```

Count can also be combined with the distinct keyword for a distinct count.

```
Select Count(DISTINCT DepartmentId) from Employees
```

Sum

The SUM() aggregate function returns the sum of the values selected for all rows.

```
SELECT SUM(Salary) FROM Employees
```

Section 6.12: Select with condition of multiple values from column

```
SELECT * FROM Cars WHERE status IN ( 'Waiting', 'Working' )
```

This is semantically equivalent to

```
SELECT * FROM Cars WHERE ( status = 'Waiting' OR status = 'Working' )
```

i.e. value IN (<value list>) is a shorthand for disjunction (logical OR).

Section 6.13: Get aggregated result for row groups

Counting rows based on a specific column value:

```
SELECT category, COUNT(*) AS item_count
FROM item
GROUP BY category;
```

Getting average income by department:

```
SELECT department, AVG(income)
FROM employees
GROUP BY department;
```

The important thing is to select only columns specified in the GROUP BY clause or used with aggregate functions.

There WHERE clause can also be used with GROUP BY, but WHERE filters out records *before* any grouping is done:

```
SELECT department, AVG(income)
FROM employees
WHERE department <> 'ACCOUNTING'
GROUP BY department;
```

If you need to filter the results after the grouping has been done, e.g, to see only departments whose average income is larger than 1000, you need to use the HAVING clause:

```
SELECT department, AVG(income)
FROM employees
WHERE department <> 'ACCOUNTING'
GROUP BY department
HAVING avg(income) > 1000;
```

第6.14节：带排序结果的选择

```
SELECT * FROM Employees ORDER BY LName
```

该语句将返回表Employees中的所有列。

Id FName LName PhoneNumber

2 约翰 约翰逊 2468101214

1 詹姆斯 史密斯 1234567890

3 Michael Williams 1357911131

```
SELECT * FROM Employees ORDER BY LName DESC
```

或者

```
SELECT * FROM Employees ORDER BY LName ASC
```

该语句改变了排序方向。

也可以指定多个排序列。例如：

```
SELECT * FROM Employees ORDER BY LName ASC, FName ASC
```

此示例将首先按LName排序结果，然后对于具有相同LName的记录，再按FName排序。这将给出类似于电话簿中的结果。

为了避免在ORDER BY子句中重复输入列名，可以改用列的编号。注意列编号从1开始。

```
SELECT Id, FName, LName, PhoneNumber FROM Employees ORDER BY 3
```

您也可以在ORDER BY子句中嵌入CASE语句。

```
SELECT Id, FName, LName, PhoneNumber FROM Employees ORDER BY CASE WHEN LName='Jones' THEN 0 ELSE 1 END ASC
```

这将使所有LName为“Jones”的记录排在结果的顶部。

第6.15节：带null的选择

```
SELECT Name FROM Customers WHERE PhoneNumber IS NULL
```

带null的选择语法不同。不要使用=, 改用IS NULL或IS NOT NULL。

第6.16节：选择唯一值（仅唯一值）

```
SELECT DISTINCT ContinentCode FROM Countries;
```

此查询将返回Countries表中ContinentCode列的所有DISTINCT（唯一、不同）值

ContinentCode

OC

EU

Section 6.14: Selection with sorted Results

```
SELECT * FROM Employees ORDER BY LName
```

This statement will return all the columns from the table Employees.

Id FName LName PhoneNumber

2 John Johnson 2468101214

1 James Smith 1234567890

3 Michael Williams 1357911131

```
SELECT * FROM Employees ORDER BY LName DESC
```

Or

```
SELECT * FROM Employees ORDER BY LName ASC
```

This statement changes the sorting direction.

One may also specify multiple sorting columns. For example:

```
SELECT * FROM Employees ORDER BY LName ASC, FName ASC
```

This example will sort the results first by LName and then, for records that have the same LName, sort by FName. This will give you a result similar to what you would find in a telephone book.

In order to save retyping the column name in the ORDER BY clause, it is possible to use instead the column's number. Note that column numbers start from 1.

```
SELECT Id, FName, LName, PhoneNumber FROM Employees ORDER BY 3
```

You may also embed a CASE statement in the ORDER BY clause.

```
SELECT Id, FName, LName, PhoneNumber FROM Employees ORDER BY CASE WHEN LName='Jones' THEN 0 ELSE 1 END ASC
```

This will sort your results to have all records with the LName of "Jones" at the top.

Section 6.15: Selecting with null

```
SELECT Name FROM Customers WHERE PhoneNumber IS NULL
```

Selection with nulls take a different syntax. Don't use =, use IS NULL or IS NOT NULL instead.

Section 6.16: Select distinct (unique values only)

```
SELECT DISTINCT ContinentCode FROM Countries;
```

This query will return all DISTINCT (unique, different) values from ContinentCode column from Countries table

ContinentCode

OC

EU

AS
NA
AF

[SQLFiddle 演示](#)

第6.17节：从多个表中选择行

```
SELECT *  
FROM  
table1,  
    table2  
  
SELECT  
table1.column1,  
    table1.column2,  
    table2.column1  
FROM  
table1,  
    table2
```

这在SQL中称为笛卡尔积，与集合中的笛卡尔积相同

这些语句从多个表中返回选定的列，合并到一个查询中。

从每个表返回的列之间没有特定的关系。

AS
NA
AF

[SQLFiddle Demo](#)

Section 6.17: Select rows from multiple tables

```
SELECT *  
FROM  
    table1,  
    table2  
  
SELECT  
    table1.column1,  
    table1.column2,  
    table2.column1  
FROM  
    table1,  
    table2
```

This is called cross product in SQL it is same as cross product in sets

These statements return the selected columns from multiple tables in one query.

There is no specific relationship between the columns returned from each table.

第7章：GROUP BY

SELECT查询的结果可以使用GROUP BY语句按一个或多个列进行分组：所有在分组列中具有相同值的结果会被聚合在一起。这会生成一个部分结果的表，而不是单一结果。GROUP BY可以结合聚合函数和HAVING语句使用，以定义非分组列的聚合方式。

第7.1节：基本的GROUP BY示例

为了便于说明，你可以把GROUP BY理解为“针对每个”的意思。下面的查询：

```
SELECT EmpID, SUM (MonthlySalary)
FROM Employee
GROUP BY EmpID
```

的意思是：

“给我每个**EmpID**对应的MonthlySalary的总和”

所以如果你的表看起来像这样：

+-----+		
员工编号 月薪		
+-----+		
1	200	
+-----+		
2	300	
+-----+		

结果：

+-----+	
1 200	
+-----+	
2 300	
+-----+	

Sum 函数似乎没有任何作用，因为一个数字的和就是该数字本身。另一方面，如果它看起来像这样：

+-----+		
EmpID MonthlySalary		
+-----+		
1	200	
+-----+		
1	300	
+-----+		
2	300	
+-----+		

结果：

|--|--|

Chapter 7: GROUP BY

Results of a SELECT query can be grouped by one or more columns using the **GROUP BY** statement: all results with the same value in the grouped columns are aggregated together. This generates a table of partial results, instead of one result. GROUP BY can be used in conjunction with aggregation functions using the **HAVING** statement to define how non-grouped columns are aggregated.

Section 7.1: Basic GROUP BY example

It might be easier if you think of GROUP BY as "for each" for the sake of explanation. The query below:

```
SELECT EmpID, SUM (MonthlySalary)
FROM Employee
GROUP BY EmpID
```

is saying:

"Give me the sum of MonthlySalary's **for each** EmpID"

So if your table looked like this:

+-----+		
EmpID MonthlySalary		
+-----+		
1	200	
+-----+		
2	300	
+-----+		

Result:

+-----+	
1 200	
+-----+	
2 300	
+-----+	

Sum wouldn't appear to do anything because the sum of one number is that number. On the other hand if it looked like this:

+-----+		
EmpID MonthlySalary		
+-----+		
1	200	
+-----+		
1	300	
+-----+		
2	300	
+-----+		

Result:

|--|--|


```
+-----+
|1|500|
+-----+
|2|300|
+-----+
```

那么它就会起作用，因为有两个 EmpID 为 1 的记录需要相加。

第7.2节：使用 HAVING 子句过滤 GROUP BY 结果

HAVING 子句用于过滤 GROUP BY 表达式的结果。注意：以下示例使用的是 Library 示例数据库。

示例：

返回所有写了多于一本书的作者（实时示例）。 [_____](#)

```
SELECT
a.Id,
a.姓名,
COUNT(*) 著书数量
FROM BooksAuthors ba
INNER JOIN 作者 a ON a.id = ba.authorid
GROUP BY
a.Id,
a.姓名
HAVING COUNT(*) > 1    -- 等同于 HAVING 著书数量 > 1
;
```

返回所有作者超过三位的书籍（实时示例）。 [_____](#)

```
SELECT
b.编号,
b.标题,
COUNT(*) 作者数量
FROM 书籍作者 ba
INNER JOIN 书籍 b ON b.id = ba.bookid
GROUP BY
b.编号,
b.标题
HAVING COUNT(*) > 3    -- 等同于 HAVING NumberOfAuthors > 3
;
```

第7.3节：使用 GROUP BY 统计给定列中每个唯一条目的行数

假设你想为某列中的某个值生成计数或小计。

给定这张表，“维斯特洛人”（Westerosians）：

姓名大家族效忠	
艾莉亚	史塔克
瑟曦	兰尼斯特
迈瑟拉·兰尼斯特	
雅拉	葛雷乔伊
凯特琳·史塔克	

```
+-----+
|1|500|
+-----+
|2|300|
+-----+
```

Then it would because there are two EmpID 1's to sum together.

Section 7.2: Filter GROUP BY results using a HAVING clause

A HAVING clause filters the results of a GROUP BY expression. Note: The following examples are using the Library example database.

Examples:

Return all authors that wrote more than one book ([live example](#)).

```
SELECT
a.Id,
a.Name,
COUNT(*) BooksWritten
FROM BooksAuthors ba
INNER JOIN Authors a ON a.id = ba.authorid
GROUP BY
a.Id,
a.Name
HAVING COUNT(*) > 1    -- equals to HAVING BooksWritten > 1
;
```

Return all books that have more than three authors ([live example](#)).

```
SELECT
b.Id,
b.Title,
COUNT(*) NumberOfAuthors
FROM BooksAuthors ba
INNER JOIN Books b ON b.id = ba.bookid
GROUP BY
b.Id,
b.Title
HAVING COUNT(*) > 3    -- equals to HAVING NumberOfAuthors > 3
;
```

Section 7.3: USE GROUP BY to COUNT the number of rows for each unique entry in a given column

Let's say you want to generate counts or subtotals for a given value in a column.

Given this table, "Westerosians":

Name	GreatHouseAllegience
Arya	Stark
Cercei	Lannister
Myrcella	Lannister
Yara	Greyjoy
Catelyn	Stark

珊莎 史塔克

没有 GROUP BY, COUNT 将简单返回行的总数：

```
SELECT Count(*) 西境人数
FROM Westerosians
```

返回...

西境人数
6

但通过添加 GROUP BY, 我们可以对给定列中的每个值进行 COUNT, 以返回某个大贵族家族中的人数, 例如：

```
SELECT GreatHouseAllegience 家族, Count(*) 西境人数
FROM Westerosians
GROUP BY GreatHouseAllegience
```

返回...

维斯特洛人的户号
史塔克 3
Greyjoy1
Lannister 2

通常将 GROUP BY 与 ORDER BY 结合使用, 以按最大或最小类别对结果进行排序：

```
SELECT GreatHouseAllegience 家族, Count(*) 西境人数
FROM Westerosians
GROUP BY GreatHouseAllegience
ORDER BY Number_of_Westerosians Desc
```

返回...

维斯特洛人的户号
史塔克 3
Lannister 2
Greyjoy1

第7.4节：ROLAP 聚合（数据挖掘）

描述

SQL 标准提供了两个额外的聚合运算符。它们使用多态值“ALL”来表示属性可以取的所有值的集合。这两个运算符是：

- 使用 data cube 它提供了比子句参数属性更多的所有可能组合。
- 使用 roll up 它提供了通过从左到右依次考虑属性（与子句参数中列出的顺序相比）获得的聚合结果。

支持这些特性的 SQL 标准版本：1999、2003、2006、2008、2011。

示例

Sansa Stark

Without GROUP BY, COUNT will simply return a total number of rows:

```
SELECT Count(*) Number_of_Westerosians
FROM Westerosians
```

returns...

Number_of_Westerosians
6

But by adding GROUP BY, we can COUNT the users for each value in a given column, to return the number of people in a given Great House, say:

```
SELECT GreatHouseAllegience House, Count(*) Number_of_Westerosians
FROM Westerosians
GROUP BY GreatHouseAllegience
```

returns...

House	Number_of_Westerosians
Stark	3
Greyjoy	1
Lannister	2

It's common to combine GROUP BY with ORDER BY to sort results by largest or smallest category:

```
SELECT GreatHouseAllegience House, Count(*) Number_of_Westerosians
FROM Westerosians
GROUP BY GreatHouseAllegience
ORDER BY Number_of_Westerosians Desc
```

returns...

House	Number_of_Westerosians
Stark	3
Lannister	2
Greyjoy	1

Section 7.4: ROLAP aggregation (Data Mining)

Description

The SQL standard provides two additional aggregate operators. These use the polymorphic value "ALL" to denote the set of all values that an attribute can take. The two operators are:

- with data cube that it provides all possible combinations than the argument attributes of the clause.
- with roll up that it provides the aggregates obtained by considering the attributes in order from left to right compared how they are listed in the argument of the clause.

SQL standard versions that support these features: 1999,2003,2006,2008,2011.

Examples

考虑下表：

食品	品牌	总量
意大利面	品牌1	100
意大利面	品牌2	250
披萨	品牌2	300

使用cube

```
select 食品, 品牌, 总量
from 表
group by 食品,品牌,总量 with cube
```

食品	品牌	总量
意大利面	品牌1	100
意大利面	品牌2	250
意大利面	全部	350
披萨	品牌2	300
披萨	全部	300
ALLBrand1		100
ALLBrand2		550
ALLALL		650

带汇总

```
select 食品, 品牌, 总量
from 表
按 食品,品牌,总金额 分组 带汇总
```

食品	品牌	总量
意面	品牌1	100
意面	品牌2	250
披萨	品牌2	300
意面	全部	350
披萨	全部	300
全部全部		650

Consider this table:

Food	Brand	Total_amount
Pasta	Brand1	100
Pasta	Brand2	250
Pizza	Brand2	300

With cube

```
select Food, Brand, Total_amount
from Table
group by Food, Brand, Total_amount with cube
```

Food	Brand	Total_amount
Pasta	Brand1	100
Pasta	Brand2	250
Pasta	ALL	350
Pizza	Brand2	300
Pizza	ALL	300
ALL	Brand1	100
ALL	Brand2	550
ALL	ALL	650

With roll up

```
select Food, Brand, Total_amount
from Table
group by Food, Brand, Total_amount with roll up
```

Food	Brand	Total_amount
Pasta	Brand1	100
Pasta	Brand2	250
Pizza	Brand2	300
Pasta	ALL	350
Pizza	ALL	300
ALL	ALL	650

第8章：ORDER BY（排序）

第8.1节：按列号排序（而非列名）

您可以使用列号（最左边的列为“1”）来指定排序的列，而不是用列名来描述该列。

优点： 如果你认为以后可能会更改列名，这样做不会破坏这段代码。

缺点： 这通常会降低查询的可读性（‘ORDER BY Reputation’一目了然，而‘ORDER BY 14’需要数数，可能还得用手指在屏幕上辅助计数。）

此查询根据 select 语句中相对列位置3的信息排序结果，而不是根据列名排序声望。

```
SELECT DisplayName, JoinDate, Reputation FROM Users ORDER BY 3
```

显示名称加入日期声望		
社区	2008-09-15	1
Jarrold Dixon	2008-10-03	11739
Geoff Dalgas	2008-10-03	12567
Joel Spolsky	2008-09-16	25784
Jeff Atwood	2008-09-16	37628

第8.2节：使用 ORDER BY 和 TOP 根据某列的值返回前 x 行

在此示例中，我们不仅可以使 用 GROUP BY 来确定返回行的 排序，还可以确定返回的行，因为我们使用了 TOP 来限制结果集。

假设我们想从一个未命名的热门问答网站中返回声誉最高的前5名用户。

没有 ORDER BY

此查询返回默认排序的前5行，在本例中默认排序是“Id”，即表中的第一列（尽管结果中未显示该列）。

```
SELECT TOP 5 DisplayName, Reputation
FROM Users
```

返回...

DisplayName	Reputation
社区	1
Geoff Dalgas	12567
Jarrold Dixon	11739
Jeff Atwood	37628
Joel Spolsky	25784

使用 ORDER BY

```
SELECT TOP 5 DisplayName, Reputation
FROM Users
```

Chapter 8: ORDER BY

Section 8.1: Sorting by column number (instead of name)

You can use a column's number (where the leftmost column is '1') to indicate which column to base the sort on, instead of describing the column by its name.

Pro: If you think it's likely you might change column names later, doing so won't break this code.

Con: This will generally reduce readability of the query (It's instantly clear what 'ORDER BY Reputation' means, while 'ORDER BY 14' requires some counting, probably with a finger on the screen.)

This query sorts result by the info in relative column position 3 from select statement instead of column name Reputation.

```
SELECT DisplayName, JoinDate, Reputation FROM Users ORDER BY 3
```

DisplayName JoinDate Reputation		
Community	2008-09-15	1
Jarrold Dixon	2008-10-03	11739
Geoff Dalgas	2008-10-03	12567
Joel Spolsky	2008-09-16	25784
Jeff Atwood	2008-09-16	37628

Section 8.2: Use ORDER BY with TOP to return the top x rows based on a column's value

In this example, we can use GROUP BY not only determined the *sort* of the rows returned, but also what rows *are* returned, since we're using TOP to limit the result set.

Let's say we want to return the top 5 highest reputation users from an unnamed popular Q&A site.

Without ORDER BY

This query returns the Top 5 rows ordered by the default, which in this case is "Id", the first column in the table (even though it's not a column shown in the results).

```
SELECT TOP 5 DisplayName, Reputation
FROM Users
```

returns...

DisplayName	Reputation
Community	1
Geoff Dalgas	12567
Jarrold Dixon	11739
Jeff Atwood	37628
Joel Spolsky	25784

With ORDER BY

```
SELECT TOP 5 DisplayName, Reputation
FROM Users
```

```
ORDER BY Reputation desc
```

返回...

DisplayName	Reputation
JonSkeet	865023
达林·迪米特罗夫	661741
巴鲁斯C	650237
汉斯·帕桑特	625870
马克·格拉维尔	601636

备注

某些版本的SQL（例如MySQL）在SELECT语句末尾使用LIMIT子句，而不是在开头使用TOP，例如：

```
SELECT 显示名称, 声望
FROM 用户
ORDER BY 声望 DESC
LIMIT 5
```

第8.3节：自定义排序顺序

要按部门对该员工表进行排序，可以使用ORDER BY 部门。但如果想要非字母顺序的不同排序，必须将部门的值映射为正确排序的不同值；这可以通过CASE表达式实现：

姓名	部门
哈桑	IT
优素福	人力资源
希拉里	人力资源
乔	IT
梅里	人力资源
肯	会计

```
SELECT *
来自 员工
按以下顺序排序 CASE 部门
    当 '人力资源' 时 1
    当 '会计' 时 2
    其他 3
END;
```

姓名	部门
尤素福	人力资源
希拉里	人力资源
梅里	人力资源
肯	会计
哈桑	信息技术
乔	IT

第8.4节：按别名排序

由于逻辑查询处理顺序，别名可以用于排序。

```
ORDER BY Reputation desc
```

returns...

DisplayName	Reputation
JonSkeet	865023
Darin Dimitrov	661741
BalusC	650237
Hans Passant	625870
Marc Gravell	601636

Remarks

Some versions of SQL (such as MySQL) use a LIMIT clause at the end of a SELECT, instead of TOP at the beginning, for example:

```
SELECT DisplayName, Reputation
FROM Users
ORDER BY Reputation DESC
LIMIT 5
```

Section 8.3: Customized sorting order

To sort this table Employee by department, you would use ORDER BY Department. However, if you want a different sort order that is not alphabetical, you have to map the Department values into different values that sort correctly; this can be done with a CASE expression:

Name	Department
Hasan	IT
Yusuf	HR
Hillary	HR
Joe	IT
Merry	HR
Ken	Accountant

```
SELECT *
FROM Employee
ORDER BY CASE Department
    WHEN 'HR' THEN 1
    WHEN 'Accountant' THEN 2
    ELSE 3
END;
```

Name	Department
Yusuf	HR
Hillary	HR
Merry	HR
Ken	Accountant
Hasan	IT
Joe	IT

Section 8.4: Order by Alias

Due to logical query processing order, alias can be used in order by.

选择 显示名称, 加入日期 作为 jd, 声誉 作为 rep
来自 用户
排序依据 jd, rep

并且可以使用SELECT语句中列的相对顺序。考虑上面相同的例子，不使用别名，而是使用相对顺序，比如显示名称是1，加入日期是2，依此类推

选择 显示名称, 加入日期 作为 jd, 声誉 作为 rep
来自 用户
ORDER BY 2, 3

第8.5节：按多列排序

SELECT DisplayName, JoinDate, Reputation FROM Users ORDER BY JoinDate, Reputation

显示名称	加入日期	声望
社区	2008-09-15	1
杰夫·阿特伍德	2008-09-16	25784
乔尔·斯波尔斯基	2008-09-16	37628
贾罗德·迪克森	2008-10-03	11739
杰夫·达尔加斯	2008-10-03	12567

SELECT DisplayName, JoinDate as jd, Reputation as rep
FROM Users
ORDER BY jd, rep

And can use relative order of the columns in the select statement .Consider the same example as above and instead of using alias use the relative order like for display name it is 1 , for Jd it is 2 and so on

SELECT DisplayName, JoinDate as jd, Reputation as rep
FROM Users
ORDER BY 2, 3

Section 8.5: Sorting by multiple columns

SELECT DisplayName, JoinDate, Reputation FROM Users ORDER BY JoinDate, Reputation

DisplayName	JoinDate	Reputation
Community	2008-09-15	1
Jeff Atwood	2008-09-16	25784
Joel Spolsky	2008-09-16	37628
Jarrod Dixon	2008-10-03	11739
Geoff Dalgas	2008-10-03	12567

第9章：与（AND）和或（OR）运算符

第9.1节：AND OR 示例

有一个表格

姓名	年龄	城市
鲍勃	10	巴黎
马特	20	柏林
玛丽	24	布拉格

```
select 姓名 from table where 年龄>10 AND 城市='布拉格'
```

结果是

姓名
玛丽

```
select 姓名 from table where 年龄=10 OR 城市='布拉格'
```

结果是

姓名
鲍勃
玛丽

Chapter 9: AND & OR Operators

Section 9.1: AND OR Example

Have a table

Name	Age	City
Bob	10	Paris
Mat	20	Berlin
Mary	24	Prague

```
select Name from table where Age>10 AND City='Prague'
```

Gives

Name
Mary

```
select Name from table where Age=10 OR City='Prague'
```

Gives

Name
Bob
Mary

第10章：CASE

CASE表达式用于实现if-then逻辑。

第10.1节：使用CASE计算满足条件的列中的行数

用例

CASE可以与SUM结合使用，仅返回符合预定义条件的项目计数。
(这类似于Excel中的COUNTIF函数。)

关键是返回表示匹配的二进制结果，这样匹配项返回的“1”可以被求和以计算匹配总数。

给定此表ItemSales，假设你想了解被归类为
“昂贵”的商品总数：

编号	商品编号	价格	价格评级
1	100	34.5	昂贵
2	145	2.3	便宜
3	100	34.5	昂贵
4	100	34.5	昂贵
5	145	10	实惠

查询

```
SELECT
    COUNT(Id) 作为 商品数量,
    SUM (
        CASE
            当 价格评级 = '昂贵' 则 1
            否则 0
        结束
    ) 作为 昂贵商品数量
FROM 商品销售
```

结果：

物品数量	昂贵物品数量
5	3

替代方案：

```
SELECT
    COUNT(Id) 作为 物品数量,
    SUM (
        CASE 价格评级
            当 '昂贵' 则 1
            否则 0
        结束
    ) 作为 昂贵商品数量
FROM 商品销售
```

Chapter 10: CASE

The CASE expression is used to implement if-then logic.

Section 10.1: Use CASE to COUNT the number of rows in a column match a condition

Use Case

CASE can be used in conjunction with SUM to return a count of only those items matching a pre-defined condition. (This is similar to COUNTIF in Excel.)

The trick is to return binary results indicating matches, so the "1"s returned for matching entries can be summed for a count of the total number of matches.

Given this table ItemSales, let's say you want to learn the total number of items that have been categorized as "Expensive":

Id	ItemId	Price	PriceRating
1	100	34.5	EXPENSIVE
2	145	2.3	CHEAP
3	100	34.5	EXPENSIVE
4	100	34.5	EXPENSIVE
5	145	10	AFFORDABLE

Query

```
SELECT
    COUNT(Id) AS ItemsCount,
    SUM (
        CASE
            WHEN PriceRating = 'Expensive' THEN 1
            ELSE 0
        END
    ) AS ExpensiveItemsCount
FROM ItemSales
```

Results:

ItemsCount	ExpensiveItemsCount
5	3

Alternative:

```
SELECT
    COUNT(Id) as ItemsCount,
    SUM (
        CASE PriceRating
            WHEN 'Expensive' THEN 1
            ELSE 0
        END
    ) AS ExpensiveItemsCount
FROM ItemSales
```

第10.2节：SELECT中的搜索CASE（匹配布尔表达式）

搜索型CASE在布尔表达式为TRUE时返回结果。

（这与简单CASE不同，简单CASE只能检查与输入的等价性。）

```
SELECT Id, 物品Id, 价格,
CASE WHEN 价格 < 10 则 '便宜'
      WHEN 价格 < 20 则 '实惠'
      否则 '昂贵'
END AS 价格评级
FROM 物品销售
```

编号	商品编号	价格	价格评级
1	100	34.5	昂贵
2	145	2.3	便宜
3	100	34.5	昂贵
4	100	34.5	昂贵
5	145	10	实惠

第10.3节：子句ORDER BY中的CASE

我们可以使用1、2、3.....来确定排序类型：

```
SELECT * FROM DEPT
ORDER BY
CASE DEPARTMENT
  WHEN '营销部' THEN 1
  WHEN '销售部' THEN 2
  WHEN '研究部' THEN 3
  WHEN '创新部' THEN 4
  ELSE 5
END,
CITY
```

ID	REGION	CITY	DEPARTMENT	EMPLOYEES_NUMBER
12	新英格兰	波士顿	市场营销	9
15	西部	旧金山	市场营销	12
9	中西部	芝加哥	销售	8
14	中大西洋	纽约	销售	12
5	西部	洛杉矶	研究	11
10	中大西洋	费城	研究	13
4	中西部	芝加哥	创新	11
2	中西部	底特律	人力资源	9

第10.4节：SELECT中的简写CASE

CASE 的简写变体对一个表达式（通常是列）与一系列值进行比较。该变体稍微简短一些，避免了重复多次写出被评估的表达式。虽然如此，仍然可以使用 ELSE 子句：

```
SELECT Id, ItemId, Price,
CASE Price WHEN 5 THEN '便宜'
           WHEN 15 THEN '实惠'
```

Section 10.2: Searched CASE in SELECT (Matches a boolean expression)

The *searched* CASE returns results when a *boolean* expression is TRUE.

(This differs from the simple case, which can only check for equivalency with an input.)

```
SELECT Id, ItemId, Price,
CASE WHEN Price < 10 THEN 'CHEAP'
      WHEN Price < 20 THEN 'AFFORDABLE'
      ELSE 'EXPENSIVE'
END AS PriceRating
FROM ItemSales
```

Id	ItemId	Price	PriceRating
1	100	34.5	EXPENSIVE
2	145	2.3	CHEAP
3	100	34.5	EXPENSIVE
4	100	34.5	EXPENSIVE
5	145	10	AFFORDABLE

Section 10.3: CASE in a clause ORDER BY

We can use 1,2,3.. to determine the type of order:

```
SELECT * FROM DEPT
ORDER BY
CASE DEPARTMENT
  WHEN 'MARKETING' THEN 1
  WHEN 'SALES' THEN 2
  WHEN 'RESEARCH' THEN 3
  WHEN 'INNOVATION' THEN 4
  ELSE 5
END,
CITY
```

ID	REGION	CITY	DEPARTMENT	EMPLOYEES_NUMBER
12	New England	Boston	MARKETING	9
15	West	San Francisco	MARKETING	12
9	Midwest	Chicago	SALES	8
14	Mid-Atlantic	New York	SALES	12
5	West	Los Angeles	RESEARCH	11
10	Mid-Atlantic	Philadelphia	RESEARCH	13
4	Midwest	Chicago	INNOVATION	11
2	Midwest	Detroit	HUMAN RESOURCES	9

Section 10.4: Shorthand CASE in SELECT

CASE's shorthand variant evaluates an expression (usually a column) against a series of values. This variant is a bit shorter, and saves repeating the evaluated expression over and over again. The ELSE clause can still be used, though:

```
SELECT Id, ItemId, Price,
CASE Price WHEN 5 THEN 'CHEAP'
           WHEN 15 THEN 'AFFORDABLE'
```

```
ELSE '昂贵'
END 作为 PriceRating
FROM ItemSales
```

需要注意的是，使用简写变体时，每个 WHEN 条件都会对整个语句进行评估。因此，以下语句：

```
SELECT
CASE ABS(CHECKSUM(NEWID())) % 4
WHEN 0 THEN '博士'
WHEN 1 THEN '硕士'
WHEN 2 THEN '先生'
WHEN 3 THEN '女士'
结束
```

可能会产生一个NULL结果。这是因为每次调用WHEN NEWID()时都会生成一个新的结果。等同于：

```
SELECT
CASE
WHEN ABS(CHECKSUM(NEWID())) % 4 = 0 THEN 'Dr'
WHEN ABS(CHECKSUM(NEWID())) % 4 = 1 THEN 'Master'
WHEN ABS(CHECKSUM(NEWID())) % 4 = 2 THEN 'Mr'
WHEN ABS(CHECKSUM(NEWID())) % 4 = 3 THEN 'Mrs'
END
```

因此，它可能会错过所有的WHEN情况，结果为NULL。

第10.5节：在UPDATE中使用CASE

关于价格上涨的示例：

```
UPDATE ItemPrice
SET Price = Price *
CASE ItemId
WHEN 1 THEN 1.05
WHEN 2 THEN 1.10
WHEN 3 THEN 1.15
ELSE 1.00
结束
```

第10.6节：CASE用于将NULL值排序到最后

这样，“0”代表已知值排在前面，“1”代表NULL值排在最后：

```
选择 ID
,地区
,城市
,部门
,部门员工数量
来自 部门
排序依据
CASE WHEN 地区 为 NULL 则 1
否则 0
END,
地区
```

```
ELSE 'EXPENSIVE'
END as PriceRating
FROM ItemSales
```

A word of caution. It's important to realize that when using the short variant the entire statement is evaluated at each WHEN. Therefore the following statement:

```
SELECT
CASE ABS(CHECKSUM(NEWID())) % 4
WHEN 0 THEN 'Dr'
WHEN 1 THEN 'Master'
WHEN 2 THEN 'Mr'
WHEN 3 THEN 'Mrs'
END
```

may produce a NULL result. That is because at each WHEN NEWID() is being called again with a new result. Equivalent to:

```
SELECT
CASE
WHEN ABS(CHECKSUM(NEWID())) % 4 = 0 THEN 'Dr'
WHEN ABS(CHECKSUM(NEWID())) % 4 = 1 THEN 'Master'
WHEN ABS(CHECKSUM(NEWID())) % 4 = 2 THEN 'Mr'
WHEN ABS(CHECKSUM(NEWID())) % 4 = 3 THEN 'Mrs'
END
```

Therefore it can miss all the WHEN cases and result as NULL.

Section 10.5: Using CASE in UPDATE

sample on price increases:

```
UPDATE ItemPrice
SET Price = Price *
CASE ItemId
WHEN 1 THEN 1.05
WHEN 2 THEN 1.10
WHEN 3 THEN 1.15
ELSE 1.00
END
```

Section 10.6: CASE use for NULL values ordered last

in this way '0' representing the known values are ranked first, '1' representing the NULL values are sorted by the last:

```
SELECT ID
,REGION
,CITY
,DEPARTMENT
,EMPLOYEES_NUMBER
FROM DEPT
ORDER BY
CASE WHEN REGION IS NULL THEN 1
ELSE 0
END,
REGION
```


ID	REGION	CITY	DEPARTMENT	EMPLOYEES_NUMBER
10	中大西洋	费城	研究	13
14	中大西洋	纽约	销售	12
9	中西部	芝加哥	销售	8
12	新英格兰	波士顿	市场营销	9
5	西部	洛杉矶	研究	11
15	空值	旧金山	市场营销	12
4	空	芝加哥	创新	11
2	空	底特律	人力资源	9

第10.7节：ORDER BY子句中的CASE，用于按两列中最小值排序记录

假设你需要根据两列中较小的值对记录进行排序。有些数据库可以使用非聚合的MIN()或LEAST()函数来实现这一点（... ORDER BY MIN(Date1, Date2)），但在标准SQL中，你必须使用CASE表达式。

下面查询中的CASE表达式查看Date1和Date2列，检查哪一列的值较小，并根据该值对记录进行排序。

示例数据

Id	Date1	Date2
1	2017-01-01	2017-01-31
2	2017-01-31	2017-01-03
3	2017-01-31	2017-01-02
4	2017-01-06	2017-01-31
5	2017-01-31	2017-01-05
6	2017-01-04	2017-01-31

查询

```
SELECT Id, Date1, Date2
FROM YourTable
按条件排序
    当COALESCE(Date1, '1753-01-01') < COALESCE(Date2, '1753-01-01') 时使用 Date1
    否则使用 Date2
结束
```

结果

编号	日期1	Date2
1	2017-01-01	2017-01-31
3	2017-01-31	2017-01-02
2	2017-01-31	2017-01-03
6	2017-01-04	2017-01-31
5	2017-01-31	2017-01-05
4	2017-01-06	2017-01-31

说明

如您所见，Id = 1 的行是第一行，因为 Date1 在整个表中记录最低，为 2017-01-01，Id = 3 的行是第二行，因为 Date2 等于 2017-01-02，这是表中第二低的值，依此类推。

所以我们将记录从 2017-01-01 到 2017-01-06 按升序排序，且不区分这些值是来自 Date1 还是 Date2 列。

ID	REGION	CITY	DEPARTMENT	EMPLOYEES_NUMBER
10	Mid-Atlantic	Philadelphia	RESEARCH	13
14	Mid-Atlantic	New York	SALES	12
9	Midwest	Chicago	SALES	8
12	New England	Boston	MARKETING	9
5	West	Los Angeles	RESEARCH	11
15	NULL	San Francisco	MARKETING	12
4	NULL	Chicago	INNOVATION	11
2	NULL	Detroit	HUMAN RESOURCES	9

Section 10.7: CASE in ORDER BY clause to sort records by lowest value of 2 columns

Imagine that you need sort records by lowest value of either one of two columns. Some databases could use a non-aggregated MIN() or LEAST() function for this (... ORDER BY MIN(Date1, Date2)), but in standard SQL, you have to use a CASE expression.

The CASE expression in the query below looks at the Date1 and Date2 columns, checks which column has the lower value, and sorts the records depending on this value.

Sample data

Id	Date1	Date2
1	2017-01-01	2017-01-31
2	2017-01-31	2017-01-03
3	2017-01-31	2017-01-02
4	2017-01-06	2017-01-31
5	2017-01-31	2017-01-05
6	2017-01-04	2017-01-31

Query

```
SELECT Id, Date1, Date2
FROM YourTable
ORDER BY CASE
    WHEN COALESCE(Date1, '1753-01-01') < COALESCE(Date2, '1753-01-01') THEN Date1
    ELSE Date2
END
```

Results

Id	Date1	Date2
1	2017-01-01	2017-01-31
3	2017-01-31	2017-01-02
2	2017-01-31	2017-01-03
6	2017-01-04	2017-01-31
5	2017-01-31	2017-01-05
4	2017-01-06	2017-01-31

Explanation

As you see row with Id = 1 is first, that because Date1 have lowest record from entire table 2017-01-01, row where Id = 3 is second that because Date2 equals to 2017-01-02 that is second lowest value from table and so on.

So we have sorted records from 2017-01-01 to 2017-01-06 ascending and no care on which one column Date1 or Date2 are those values.

第11章：LIKE操作符

第11.1节：匹配开放式模式

将 % 通配符附加到字符串的开头或结尾（或两者）将允许在模式的开始之前或结束之后匹配零个或多个任意字符。

在中间使用 '%' 将允许在模式的两部分之间匹配零个或多个字符。

我们将使用这个员工表：

Id	姓名	电话号码	经理Id	部门Id	薪水	入职日期
1	约翰	约翰逊2468101214		1	400	23-03-2005
2	索菲·阿穆森	2479100211		1	400	11-01-2010
3	罗尼·史密斯	2462544026	2	1	600	06-08-2015
4	乔恩	Sanchez	2454124602	1	400	23-03-2005
5	希尔德	克纳格	2468021911	2	800	01-01-2000

以下语句匹配员工表中所有FName包含字符串'on'的记录。

```
SELECT * FROM Employees WHERE FName LIKE '%on%';
```

编号	名字	姓氏	电话号码	经理编号	部门编号	薪水	入职日期
3	Ronny	Smith	2462544026	2	1	600	06-08-2015
4	Jon	Sanchez	2454124602	1	1	400	23-03-2005

以下语句匹配员工表中所有电话号码以字符串'246'开头的记录。

```
SELECT * FROM Employees WHERE PhoneNumber LIKE '246%';
```

编号	名字	姓氏	电话号码	经理编号	部门编号	薪水	入职日期
1	约翰	Johnson	2468101214	1	1	400	23-03-2005
3	罗尼·史密斯		2462544026	2	1	600	06-08-2015
5	希尔德	克纳格	2468021911	2	1	800	01-01-2000

以下语句匹配所有员工表中电话号码以字符串“11”结尾的记录。

```
SELECT * FROM Employees WHERE PhoneNumber LIKE '%11'
```

Id	姓名	电话号码	经理Id	部门Id	薪水	入职日期
2	索菲·阿穆森	2479100211		1	400	11-01-2010
5	希尔德	克纳格	2468021911	2	800	01-01-2000

所有员工表中名字（Fname）第3个字符为“n”的记录。

```
SELECT * FROM Employees WHERE FName LIKE '__n%';
```

（在“n”前使用两个下划线以跳过前两个字符）

编号	名字	姓氏	电话号码	经理编号	部门编号	薪水	入职日期
3	罗尼·史密斯		2462544026	2	1	600	06-08-2015
4	乔恩	Sanchez	2454124602	1	1	400	23-03-2005

Chapter 11: LIKE operator

Section 11.1: Match open-ended pattern

The % wildcard appended to the beginning or end (or both) of a string will allow 0 or more of any character before the beginning or after the end of the pattern to match.

Using '%' in the middle will allow 0 or more characters between the two parts of the pattern to match.

We are going to use this Employees Table:

Id	FName	LName	PhoneNumber	ManagerId	DepartmentId	Salary	Hire_date
1	John	Johnson	2468101214	1	1	400	23-03-2005
2	Sophie	Amudsen	2479100211	1	1	400	11-01-2010
3	Ronny	Smith	2462544026	2	1	600	06-08-2015
4	Jon	Sanchez	2454124602	1	1	400	23-03-2005
5	Hilde	Knag	2468021911	2	1	800	01-01-2000

Following statement matches for all records having FName **containing** string 'on' from Employees Table.

```
SELECT * FROM Employees WHERE FName LIKE '%on%';
```

Id	FName	LName	PhoneNumber	ManagerId	DepartmentId	Salary	Hire_date
3	Ronny	Smith	2462544026	2	1	600	06-08-2015
4	Jon	Sanchez	2454124602	1	1	400	23-03-2005

Following statement matches all records having PhoneNumber **starting with** string '246' from Employees.

```
SELECT * FROM Employees WHERE PhoneNumber LIKE '246%';
```

Id	FName	LName	PhoneNumber	ManagerId	DepartmentId	Salary	Hire_date
1	John	Johnson	246 8101214	1	1	400	23-03-2005
3	Ronny	Smith	246 2544026	2	1	600	06-08-2015
5	Hilde	Knag	246 8021911	2	1	800	01-01-2000

Following statement matches all records having PhoneNumber **ending with** string '11' from Employees.

```
SELECT * FROM Employees WHERE PhoneNumber LIKE '%11'
```

Id	FName	LName	PhoneNumber	ManagerId	DepartmentId	Salary	Hire_date
2	Sophie	Amudsen	24791002 11	1	1	400	11-01-2010
5	Hilde	Knag	24680219 11	2	1	800	01-01-2000

All records where FName **3rd character** is 'n' from Employees.

```
SELECT * FROM Employees WHERE FName LIKE '__n%';
```

(two underscores are used before 'n' to skip first 2 characters)

Id	FName	LName	PhoneNumber	ManagerId	DepartmentId	Salary	Hire_date
3	Ronny	Smith	2462544026	2	1	600	06-08-2015
4	Jon	Sanchez	2454124602	1	1	400	23-03-2005

第11.2节：单字符匹配

为了扩展结构化查询语言（SQL-SELECT）语句的选择范围，可以使用通配符字符，百分号（%）和下划线（_）。

下划线_字符可用作模式匹配中任意单个字符的通配符。

查找所有名字以“j”开头，以“n”结尾且名字长度恰好为3个字符的员工。

```
SELECT * FROM Employees WHERE FName LIKE 'j_n'
```

下划线（_）字符也可以多次用作通配符来匹配模式。

例如，这个模式会匹配“jon”、“jan”、“jen”等。

这些名字不会被显示：“jn”、“john”、“jordan”、“justin”、“jason”、“julian”、“jillian”、“joann”，因为在我们的查询中只使用了一个下划线，它只能匹配恰好一个字符，所以结果必须是3个字符的名字。

例如，这个模式会匹配“LaSt”、“LoSt”、“HaLt”等。

```
SELECT * FROM Employees WHERE FName LIKE '_A_T'
```

第11.3节：LIKE查询中的ESCAPE语句

如果你实现文本搜索作为LIKE查询，通常是这样做的：

```
SELECT *
FROM T_Whatever
WHERE SomeField LIKE CONCAT('%', @in_SearchText, '%')
```

然而，（除了当你可以使用全文搜索时不一定非得使用LIKE这一事实之外）当有人输入像“50%”或“a_b”这样的文本时，这会产生问题。

所以（而不是切换到全文搜索），你可以使用LIKE-escape语句来解决这个问题：

```
SELECT *
FROM T_Whatever
WHERE SomeField LIKE CONCAT('%', @in_SearchText, '%') ESCAPE '\'
```

这意味着\现在将被视为转义字符。这意味着，你现在只需在搜索字符串中的每个字符前加上\，即使用户输入了像%或_这样的特殊字符，结果也会开始变得正确。

例如。

```
string stringToSearch = "abc_def 50%";
string newString = "";
foreach(char c in stringToSearch)
    newString += @"\" + c;

sqlCmd.Parameters.Add("@in_SearchText", newString);
// instead of sqlCmd.Parameters.Add("@in_SearchText", stringToSearch);
```

注意：上述算法仅用于演示目的。当一个字形（grapheme）由多个字符（utf-8）组成时，例如string stringToSearch = "Les Mise\u0301rables";该算法将不起作用。你需要这样做

Section 11.2: Single character match

To broaden the selections of a structured query language (SQL-SELECT) statement, wildcard characters, the percent sign (%) and the underscore (_), can be used.

The _ (underscore) character can be used as a wildcard for any single character in a pattern match.

Find all employees whose FName start with 'j' and end with 'n' and has exactly 3 characters in FName.

```
SELECT * FROM Employees WHERE FName LIKE 'j_n'
```

_ (underscore) character can also be used more than once as a wild card to match patterns.

For example, this pattern would match "jon", "jan", "jen", etc.

These names will not be shown "jn","john","jordan", "justin", "jason", "julian", "jillian", "joann" because in our query one underscore is used and it can skip exactly one character, so result must be of 3 character FName.

For example, this pattern would match "LaSt", "LoSt", "HaLt", etc.

```
SELECT * FROM Employees WHERE FName LIKE '_A_T'
```

Section 11.3: ESCAPE statement in the LIKE-query

If you implement a text-search as LIKE-query, you usually do it like this:

```
SELECT *
FROM T_Whatever
WHERE SomeField LIKE CONCAT('%', @in_SearchText, '%')
```

However, (apart from the fact that you shouldn't necessarily use LIKE when you can use fulltext-search) this creates a problem when somebody inputs text like "50%" or "a_b".

So (instead of switching to fulltext-search), you can solve that problem using the LIKE-escape statement:

```
SELECT *
FROM T_Whatever
WHERE SomeField LIKE CONCAT('%', @in_SearchText, '%') ESCAPE '\'
```

That means \ will now be treated as ESCAPE character. This means, you can now just prepend \ to every character in the string you search, and the results will start to be correct, even when the user enters a special character like % or _.

e.g.

```
string stringToSearch = "abc_def 50%";
string newString = "";
foreach(char c in stringToSearch)
    newString += @"\" + c;

sqlCmd.Parameters.Add("@in_SearchText", newString);
// instead of sqlCmd.Parameters.Add("@in_SearchText", stringToSearch);
```

Note: The above algorithm is for demonstration purposes only. It will not work in cases where 1 grapheme consists out of several characters (utf-8). e.g. string stringToSearch = "Les Mise\u0301rables"; You'll need to do this

针对每个字形，而不是每个字符。如果你处理的是亚洲/东亚/南亚语言，或者说如果你想要正确的代码，应该针对每个字形簇（graphemeCluster）来处理。

另见ReverseString，C#面试题

第11.4节：搜索字符范围

以下语句匹配Employees表中FName以A到F字母开头的所有记录。

```
SELECT * FROM Employees WHERE FName LIKE '[A-F]%'
```

第11.5节：按范围或集合匹配

匹配指定范围内的任意单个字符（例如：[a-f]）或集合中的任意字符（例如：[abcdef]）。

此范围模式会匹配“gary”，但不会匹配“mary”：

```
SELECT * FROM Employees WHERE FName LIKE '[a-g]ary'
```

此集合模式会匹配“mary”，但不会匹配“gary”：

```
SELECT * FROM Employees WHERE FName LIKE '[lmnop]ary'
```

范围或集合也可以通过在范围或集合前添加^脱字符来取反：

此范围模式不会匹配“gary”，但会匹配“mary”：

```
SELECT * FROM Employees WHERE FName LIKE '[^a-g]ary'
```

此集合模式不会匹配“mary”，但会匹配“gary”：

```
SELECT * FROM Employees WHERE FName LIKE '[^lmnop]ary'
```

第11.6节：通配符字符

通配符字符用于SQL的LIKE操作符。SQL通配符用于在表中搜索数据。

SQL中的通配符有：%、_、[字符列表]、[^字符列表]

% - 代表零个或多个字符的替代符

```
例如：//选择所有城市名以“Lo”开头的客户
SELECT * FROM Customers
WHERE City LIKE 'Lo%';

// 选择所有 City 中包含模式 "es" 的客户
SELECT * FROM Customers
WHERE City LIKE '%es%';
```

_ - 单个字符的替代符

```
例如：// 选择所有 City 以任意单个字符开头，后跟 "erlin" 的客户
SELECT * FROM Customers
```

for each grapheme, not for each character. You should not use the above algorithm if you're dealing with Asian/East-Asian/South-Asian languages. Or rather, if you want correct code to begin with, you should just do that for each graphemeCluster.

See also [ReverseString, a C# interview-question](#)

Section 11.4: Search for a range of characters

Following statement matches all records having FName that starts with a letter from A to F from Employees Table.

```
SELECT * FROM Employees WHERE FName LIKE '[A-F]%'
```

Section 11.5: Match by range or set

Match any single character within the specified range (e.g.: [a-f]) or set (e.g.: [abcdef]).

This range pattern would match "gary" but not "mary":

```
SELECT * FROM Employees WHERE FName LIKE '[a-g]ary'
```

This set pattern would match "mary" but not "gary":

```
SELECT * FROM Employees WHERE FName LIKE '[lmnop]ary'
```

The range or set can also be negated by appending the ^ caret before the range or set:

This range pattern would *not* match "gary" but will match "mary":

```
SELECT * FROM Employees WHERE FName LIKE '[^a-g]ary'
```

This set pattern would *not* match "mary" but will match "gary":

```
SELECT * FROM Employees WHERE FName LIKE '[^lmnop]ary'
```

Section 11.6: Wildcard characters

wildcard characters are used with the SQL LIKE operator. SQL wildcards are used to search for data within a table.

Wildcards in SQL are:%, _ [charlist], [^charlist]

% - A substitute for zero or more characters

```
Eg: //selects all customers with a City starting with "Lo"
SELECT * FROM Customers
WHERE City LIKE 'Lo%';

//selects all customers with a City containing the pattern "es"
SELECT * FROM Customers
WHERE City LIKE '%es%';
```

_ - A substitute for a single character

```
Eg://selects all customers with a City starting with any character, followed by "erlin"
SELECT * FROM Customers
```

```
WHERE City LIKE '_erlin';
```

[charlist] - 匹配的字符集和范围

例如：// 选择所有 City 以 "a"、"d" 或 "l" 开头的客户

```
SELECT * FROM Customers
WHERE City LIKE '[adl]%';
```

// 选择所有 City 以 "a"、"d" 或 "l" 开头的客户

```
SELECT * FROM Customers
WHERE City LIKE '[a-c]%';
```

[^charlist] - 仅匹配不在方括号内指定的字符

例如：//选择所有城市名称不以字符 "a"、"p" 或 "l" 开头的客户

```
SELECT * FROM Customers
WHERE City LIKE '[^ap1]%';
```

or

```
SELECT * FROM Customers
WHERE 城市 NOT LIKE '[apl]%' and 城市 like '_%';
```

```
WHERE City LIKE '_erlin';
```

[charlist] - Sets and ranges of characters to match

Eg://selects all customers with a City starting with "a", "d", or "l"

```
SELECT * FROM Customers
WHERE City LIKE '[adl]%';
```

//selects all customers with a City starting with "a", "d", or "l"

```
SELECT * FROM Customers
WHERE City LIKE '[a-c]%';
```

[^charlist] - Matches only a character NOT specified within the brackets

Eg://selects all customers with a City starting with a character that is not "a", "p", or "l"

```
SELECT * FROM Customers
WHERE City LIKE '[^ap1]%';
```

or

```
SELECT * FROM Customers
WHERE City NOT LIKE '[apl]%' and city like '_%';
```


第12章：IN子句

第12.1节：简单的IN子句

获取具有给定任意id的记录

```
select *  
from products  
where id in (1,8,3)
```

上述查询等同于

```
select *  
from products  
where id = 1  
       or id = 8  
       or id = 3
```

第12.2节：使用带子查询的IN子句

```
SELECT *  
FROM customers  
WHERE id IN (  
    SELECT DISTINCT customer_id  
    FROM orders  
);
```

上述查询将返回系统中所有有订单的客户。

Chapter 12: IN clause

Section 12.1: Simple IN clause

To get records having **any** of the given ids

```
select *  
from products  
where id in (1,8,3)
```

The query above is equal to

```
select *  
from products  
where id = 1  
       or id = 8  
       or id = 3
```

Section 12.2: Using IN clause with a subquery

```
SELECT *  
FROM customers  
WHERE id IN (  
    SELECT DISTINCT customer_id  
    FROM orders  
);
```

The above will give you all the customers that have orders in the system.

第13章：使用WHERE和

HAVING过滤结果

第13.1节：使用BETWEEN过滤结果

以下示例使用商品销售和客户示例数据库。

注意：BETWEEN 运算符是包含边界的。

使用 BETWEEN 运算符与数字：

```
SELECT * From ItemSales
WHERE Quantity BETWEEN 10 AND 17
```

该查询将返回所有ItemSales记录，其数量大于或等于10且小于或等于17。结果将如下所示：

Id	SaleDate	ItemId	数量	价格
1	2013-07-01	100	10	34.5
4	2013-07-23	100	15	34.5
5	2013-07-24	145	10	34.5

使用 BETWEEN 运算符与日期值：

```
SELECT * From ItemSales
WHERE SaleDate BETWEEN '2013-07-11' AND '2013-05-24'
```

此查询将返回所有ItemSales记录，其SaleDate大于或等于2013年7月11日且小于或等于2013年5月24日。

Id	SaleDate	ItemId	数量	价格
3	2013-07-11	100	20	34.5
4	2013-07-23	100	15	34.5
5	2013-07-24	145	10	34.5

在比较日期时间值而非日期时，您可能需要将日期时间值转换为日期值，或加上或减去24小时以获得正确结果。

使用BETWEEN运算符处理文本值：

```
SELECT Id, FName, LName FROM Customers
WHERE LName BETWEEN 'D' AND 'L' ;
```

实时示例：[SQL fiddle](#)

此查询将返回所有名字按字母顺序介于'D'和'L'之间的客户。在此情况下，将返回客户#1和#3。名字以'M'开头的客户#2将不被包含。

Id FName LName

Chapter 13: Filter results using WHERE and HAVING

Section 13.1: Use BETWEEN to Filter Results

The following examples use the Item Sales and Customers sample databases.

Note: The BETWEEN operator *is* inclusive.

Using the BETWEEN operator with Numbers:

```
SELECT * From ItemSales
WHERE Quantity BETWEEN 10 AND 17
```

This query will return all ItemSales records that have a quantity that is greater or equal to 10 and less than or equal to 17. The results will look like:

Id	SaleDate	ItemId	Quantity	Price
1	2013-07-01	100	10	34.5
4	2013-07-23	100	15	34.5
5	2013-07-24	145	10	34.5

Using the BETWEEN operator with Date Values:

```
SELECT * From ItemSales
WHERE SaleDate BETWEEN '2013-07-11' AND '2013-05-24'
```

This query will return all ItemSales records with a SaleDate that is greater than or equal to July 11, 2013 and less than or equal to May 24, 2013.

Id	SaleDate	ItemId	Quantity	Price
3	2013-07-11	100	20	34.5
4	2013-07-23	100	15	34.5
5	2013-07-24	145	10	34.5

When comparing datetime values instead of dates, you may need to convert the datetime values into a date values, or add or subtract 24 hours to get the correct results.

Using the BETWEEN operator with Text Values:

```
SELECT Id, FName, LName FROM Customers
WHERE LName BETWEEN 'D' AND 'L' ;
```

Live example: [SQL fiddle](#)

This query will return all customers whose name alphabetically falls between the letters 'D' and 'L'. In this case, Customer #1 and #3 will be returned. Customer #2, whose name begins with a 'M' will not be included.

Id FName LName

- 1 威廉·琼斯
- 3 理查德·戴维斯

第13.2节：使用HAVING与聚合函数

与WHERE子句不同，HAVING可以与聚合函数一起使用。

聚合函数是一种函数，其中多个行的值根据某些条件被分组作为输入，以形成一个具有更重要意义或度量的单一值（Wikipedia）。

常见的聚合函数包括COUNT()、SUM()、MIN()和MAX()。

此示例使用示例数据库中的汽车表。

```
SELECT CustomerId, COUNT(Id) AS [汽车数量]
FROM Cars
GROUP BY CustomerId
HAVING COUNT(Id) > 1
```

此查询将返回拥有多于一辆车的客户的CustomerId和汽车数量。在本例中，唯一拥有多于一辆车的客户是客户#1。

结果将如下所示：

CustomerId	汽车数量
1	2

第13.3节：带有NULL/NOT NULL值的WHERE子句

```
SELECT *
FROM Employees
WHERE ManagerId IS NULL
```

该语句将返回所有ManagerId列值为NULL的员工记录。

结果将是：

编号	名	姓	电话号码	经理编号	部门编号
1	詹姆斯	史密斯	1234567890	NULL	1

```
SELECT *
FROM 员工表
WHERE 经理编号 IS NOT NULL
```

该语句将返回所有经理编号值不为NULL的员工记录。

结果将是：

编号	名	姓	电话号码	经理编号	部门编号
2	约翰	约翰逊	2468101214	1	1
3	迈克尔	威廉姆斯	1357911131	1	2
4	约翰纳森	史密斯	1212121212	2	1

- 1 William Jones
- 3 Richard Davis

Section 13.2: Use HAVING with Aggregate Functions

Unlike the **WHERE** clause, **HAVING** can be used with aggregate functions.

An aggregate function is a function where the values of multiple rows are grouped together as input on certain criteria to form a single value of more significant meaning or measurement ([Wikipedia](#)).

Common aggregate functions include **COUNT()**, **SUM()**, **MIN()**, and **MAX()**.

This example uses the Car Table from the Example Databases.

```
SELECT CustomerId, COUNT(Id) AS [Number of Cars]
FROM Cars
GROUP BY CustomerId
HAVING COUNT(Id) > 1
```

This query will return the CustomerId and Number of Cars count of any customer who has more than one car. In this case, the only customer who has more than one car is Customer #1.

The results will look like:

CustomerId	Number of Cars
1	2

Section 13.3: WHERE clause with NULL/NOT NULL values

```
SELECT *
FROM Employees
WHERE ManagerId IS NULL
```

This statement will return all Employee records where the value of the ManagerId column is **NULL**.

The result will be:

Id	FName	LName	PhoneNumber	ManagerId	DepartmentId
1	James	Smith	1234567890	NULL	1

```
SELECT *
FROM Employees
WHERE ManagerId IS NOT NULL
```

This statement will return all Employee records where the value of the ManagerId is *not* **NULL**.

The result will be:

Id	FName	LName	PhoneNumber	ManagerId	DepartmentId
2	John	Johnson	2468101214	1	1
3	Michael	Williams	1357911131	1	2
4	Johnathon	Smith	1212121212	2	1

注意： 如果将WHERE子句改为WHERE 经理编号 = NULL 或 WHERE

经理编号 <> NULL，则相同的查询不会返回结果。

第13.4节：等值

```
SELECT * FROM Employees
```

该语句将返回表员工表中的所有行。

编号	名	姓	电话号码	经理编号	部门编号	薪水	入职日期
创建日期	修改日期						
1	詹姆斯	史密斯	1234567890	NULL	1	1000	2002-01-01
2002-01-01							
2	约翰	约翰逊	2468101214	1	1	400	2005-03-23
2002-01-01							
3	迈克尔	威廉姆斯	1357911131	1	2	600	2009-12-05
NULL							
4	约翰森·史密斯		1212121212	2	1	500	2016-07-24
2002-01-01							

在你的SELECT语句末尾使用WHERE可以限制返回的行满足某个条件。在本例中，使用=符号进行精确匹配：

```
SELECT * FROM 员工 WHERE 部门编号 = 1
```

只会返回部门编号等于1的行：

编号	名	姓	电话号码	经理编号	部门编号	薪水	入职日期
创建日期	修改日期						
1	詹姆斯	史密斯	1234567890	NULL	1	1000	2002-01-01
2002-01-01							
2	约翰	约翰逊	2468101214	1	1	400	2005-03-23
2002-01-01							
4	约翰森·史密斯		1212121212	2	1	500	2016-07-24
2002-01-01							

第13.5节：WHERE子句只返回符合其条件的行

Steam商店页面有一个售价低于10美元的游戏专区。在他们系统的深处，可能有一个类似如下的查询：

```
SELECT *
FROM 商品
WHERE 价格 < 10
```

第13.6节：AND和OR

你也可以将多个操作符组合起来，创建更复杂的WHERE条件。以下示例使用了员工表：

```
编号名姓电话号码经理编号部门编号薪水入职日期
```

Note: The same query will not return results if you change the WHERE clause to WHERE ManagerId = NULL or WHERE ManagerId <> NULL.

Section 13.4: Equality

```
SELECT * FROM Employees
```

This statement will return all the rows from the table Employees.

Id	FName	LName	PhoneNumber	ManagerId	DepartmentId	Salary	Hire_date
CreatedDate	ModifiedDate						
1	James	Smith	1234567890	NULL	1	1000	01-01-2002
01-01-2002							
2	John	Johnson	2468101214	1	1	400	23-03-2005
01-01-2002							
3	Michael	Williams	1357911131	1	2	600	12-05-2009
NULL							
4	Johnathon Smith		1212121212	2	1	500	24-07-2016
01-01-2002							

Using a WHERE at the end of your SELECT statement allows you to limit the returned rows to a condition. In this case, where there is an exact match using the = sign:

```
SELECT * FROM Employees WHERE DepartmentId = 1
```

Will only return the rows where the DepartmentId is equal to 1:

Id	FName	LName	PhoneNumber	ManagerId	DepartmentId	Salary	Hire_date
CreatedDate	ModifiedDate						
1	James	Smith	1234567890	NULL	1	1000	01-01-2002
01-01-2002							
2	John	Johnson	2468101214	1	1	400	23-03-2005
01-01-2002							
4	Johnathon Smith		1212121212	2	1	500	24-07-2016
01-01-2002							

Section 13.5: The WHERE clause only returns rows that match its criteria

Steam has a games under \$10 section of their store page. Somewhere deep in the heart of their systems, there's probably a query that looks something like:

```
SELECT *
FROM Items
WHERE Price < 10
```

Section 13.6: AND and OR

You can also combine several operators together to create more complex WHERE conditions. The following examples use the Employees table:

```
Id    FName    LName    PhoneNumber    ManagerId    DepartmentId    Salary    Hire_date
```

创建日期	修改日期							
1	詹姆斯 史密斯	1234567890	NULL	1	1000	2002-01-01	2002-01-01	
2002-01-01								
2	约翰 约翰逊	2468101214	1	1	400	2005-03-23	2005-03-23	
2002-01-01								
3	迈克尔 威廉姆斯	1357911131	1	2	600	2009-12-05	2009-12-05	
NULL								
4	约翰森·史密斯	1212121212	2	1	500	2016-07-24	2016-07-24	
2002-01-01								

并且

选择*从员工哪里部门编号=1并且经理编号=1

将返回：

编号	名	姓	电话号码	经理编号	部门编号	薪水	入职日期	
创建日期	修改日期							
2	约翰 约翰逊	2468101214	1	1	400	2005-03-23	2005-03-23	
2002-01-01								

或者

选择*从员工哪里部门编号=2或者经理编号=2

将返回：

编号	名	姓	电话号码	经理编号	部门编号	薪水	入职日期	
创建日期	修改日期							
3	迈克尔 威廉姆斯	1357911131	1	2	600	2009-12-05	2009-12-05	
NULL								
4	约翰森·史密斯	1212121212	2	1	500	2016-07-24	2016-07-24	
2002-01-01								

第13.7节：使用IN返回列表中包含的值的行

此示例使用示例数据库中的汽车表。

SELECT *
从汽车
哪里总成本在(100, 200, 300)

此查询将返回价格为200的汽车#2和价格为100的汽车#3。请注意，这等同于使用带有OR的多个子句，例如：

SELECT *
从汽车
WHERE TotalCost = 100 OR TotalCost = 200 OR TotalCost = 300

第13.8节：使用LIKE查找匹配的字符串和子字符串

查看LIKE操作符的完整文档。

CreatedDate	ModifiedDate							
1	James Smith	1234567890	NULL	1	1000	01-01-2002	01-01-2002	
01-01-2002								
2	John Johnson	2468101214	1	1	400	23-03-2005	23-03-2005	
01-01-2002								
3	Michael Williams	1357911131	1	2	600	12-05-2009	12-05-2009	
NULL								
4	Johnathon Smith	1212121212	2	1	500	24-07-2016	24-07-2016	
01-01-2002								

AND

SELECT * FROM Employees WHERE DepartmentId = 1 AND ManagerId = 1

Will return:

Id	FName	LName	PhoneNumber	ManagerId	DepartmentId	Salary	Hire_date	
CreatedDate	ModifiedDate							
2	John Johnson	2468101214	1	1	400	23-03-2005	23-03-2005	
01-01-2002								

OR

SELECT * FROM Employees WHERE DepartmentId = 2 OR ManagerId = 2

Will return:

Id	FName	LName	PhoneNumber	ManagerId	DepartmentId	Salary	Hire_date	
CreatedDate	ModifiedDate							
3	Michael Williams	1357911131	1	2	600	12-05-2009	12-05-2009	
NULL								
4	Johnathon Smith	1212121212	2	1	500	24-07-2016	24-07-2016	
01-01-2002								

Section 13.7: Use IN to return rows with a value contained in a list

This example uses the Car Table from the Example Databases.

SELECT *
FROM Cars
WHERE TotalCost IN (100, 200, 300)

This query will return Car #2 which costs 200 and Car #3 which costs 100. Note that this is equivalent to using multiple clauses with OR, e.g.:

SELECT *
FROM Cars
WHERE TotalCost = 100 OR TotalCost = 200 OR TotalCost = 300

Section 13.8: Use LIKE to find matching strings and substrings

See full documentation on LIKE operator.

此示例使用示例数据库中的员工表（Employees Table）。

```
SELECT *
FROM Employees
WHERE FName LIKE 'John'
```

此查询只会返回名字完全匹配“John”的员工编号1。

```
SELECT *
FROM Employees
WHERE FName like 'John%'
```

添加%允许你搜索子字符串：

- John% - 将返回任何名字以“John”开头，后面跟任意字符的员工
- %John - 将返回任何名字以“John”结尾，前面有任意字符的员工
- %约翰% - 将返回任何名称中包含“约翰”的员工

在这种情况下，查询将返回员工编号为2的“约翰”以及员工编号为4的“约翰纳森”。

第13.9节：WHERE EXISTS

将选择TableName中在TableName1中有匹配记录的记录。

```
SELECT * FROM TableName t WHERE EXISTS (
    SELECT 1 FROM TableName1 t1 where t.Id = t1.Id)
```

第13.10节：使用 HAVING 检查组中的多个条件

订单表

客户编号	产品编号	数量	价格
1	2	5	100
1	3	2	200
1	4	1	500
2	1	4	50
3	5	6	700

要检查同时订购了产品编号2和3的客户，可以使用 HAVING

```
选择 customerId
来自 orders
where productID in (2,3)
group by customerId
having count(distinct productID) = 2
```

返回值：

customerId
1

该查询仅选择包含相关 productID 的记录，并通过 HAVING 子句检查分组

This example uses the Employees Table from the Example Databases.

```
SELECT *
FROM Employees
WHERE FName LIKE 'John'
```

This query will only return Employee #1 whose first name matches 'John' exactly.

```
SELECT *
FROM Employees
WHERE FName like 'John%'
```

Adding % allows you to search for a substring:

- John% - will return any Employee whose name begins with 'John', followed by any amount of characters
- %John - will return any Employee whose name ends with 'John', proceeded by any amount of characters
- %John% - will return any Employee whose name contains 'John' anywhere within the value

In this case, the query will return Employee #2 whose name is 'John' as well as Employee #4 whose name is 'Johnathon'.

Section 13.9: Where EXISTS

Will select records in TableName that have records matching in TableName1.

```
SELECT * FROM TableName t WHERE EXISTS (
    SELECT 1 FROM TableName1 t1 where t.Id = t1.Id)
```

Section 13.10: Use HAVING to check for multiple conditions in a group

Orders Table

CustomerId	ProductId	Quantity	Price
1	2	5	100
1	3	2	200
1	4	1	500
2	1	4	50
3	5	6	700

To check for customers who have ordered both - ProductID 2 and 3, HAVING can be used

```
select customerId
from orders
where productID in (2,3)
group by customerId
having count(distinct productID) = 2
```

Return value:

customerId
1

The query selects only records with the productIDs in questions and with the HAVING clause checks for groups

拥有 2 个 productID 而非仅 1 个。

另一种可能是

```
选择 customerId
来自 orders
group by customerId
having sum(case when productID = 2 then 1 else 0 end) > 0
    and sum(case when productID = 3 then 1 else 0 end) > 0
```

该查询仅选择至少包含一个 productID 为 2 且至少包含一个 productID 为 3 的分组。

having 2 productIds and not just one.

Another possibility would be

```
select customerId
from orders
group by customerId
having sum(case when productID = 2 then 1 else 0 end) > 0
    and sum(case when productID = 3 then 1 else 0 end) > 0
```

This query selects only groups having at least one record with productID 2 and at least one with productID 3.

第14章：跳过和获取（分页）

第14.1节：限制结果数量

ISO/ANSI SQL：

```
SELECT * FROM TableName FETCH FIRST 20 ROWS ONLY;
```

MySQL；PostgreSQL；SQLite：

```
SELECT * FROM TableName LIMIT 20;
```

Oracle：

```
SELECT Id,
       Col1
FROM (SELECT Id,
            Col1,
            row_number() over (order by Id) RowNumber
      FROM TableName)
WHERE RowNumber <= 20
```

SQL Server:

```
SELECT TOP 20 *
FROM dbo.[Sale]
```

第14.2节：跳过然后获取部分结果（分页）

ISO/ANSI SQL：

```
SELECT Id, Col1
FROM TableName
ORDER BY Id
OFFSET 20 ROWS FETCH NEXT 20 ROWS ONLY;
```

MySQL:

```
SELECT * FROM TableName LIMIT 20, 20; -- offset, limit
```

Oracle；SQL Server：

```
SELECT Id,
       Col1
FROM (SELECT Id,
            Col1,
            row_number() over (order by Id) RowNumber
      FROM TableName)
WHERE RowNumber 介于 21 和 40 之间
```

PostgreSQL；SQLite：

```
SELECT * FROM TableName LIMIT 20 OFFSET 20;
```

Chapter 14: SKIP TAKE (Pagination)

Section 14.1: Limiting amount of results

ISO/ANSI SQL:

```
SELECT * FROM TableName FETCH FIRST 20 ROWS ONLY;
```

MySQL; PostgreSQL; SQLite:

```
SELECT * FROM TableName LIMIT 20;
```

Oracle:

```
SELECT Id,
       Col1
FROM (SELECT Id,
            Col1,
            row_number() over (order by Id) RowNumber
      FROM TableName)
WHERE RowNumber <= 20
```

SQL Server:

```
SELECT TOP 20 *
FROM dbo.[Sale]
```

Section 14.2: Skipping then taking some results (Pagination)

ISO/ANSI SQL:

```
SELECT Id, Col1
FROM TableName
ORDER BY Id
OFFSET 20 ROWS FETCH NEXT 20 ROWS ONLY;
```

MySQL:

```
SELECT * FROM TableName LIMIT 20, 20; -- offset, limit
```

Oracle; SQL Server:

```
SELECT Id,
       Col1
FROM (SELECT Id,
            Col1,
            row_number() over (order by Id) RowNumber
      FROM TableName)
WHERE RowNumber BETWEEN 21 AND 40
```

PostgreSQL; SQLite:

```
SELECT * FROM TableName LIMIT 20 OFFSET 20;
```

第14.3节：跳过结果中的某些行

ISO/ANSI SQL：

```
SELECT Id, Col1
FROM TableName
ORDER BY Id
OFFSET 20 行
```

MySQL:

```
SELECT * FROM TableName LIMIT 20, 42424242424242;
-- 跳过20行，使用比表中行数更多的非常大数字来获取
```

Oracle：

```
SELECT Id,
       Col1
FROM (SELECT Id,
            Col1,
            row_number() over (order by Id) RowNumber
      FROM TableName)
WHERE RowNumber > 20
```

PostgreSQL：

```
SELECT * FROM TableName OFFSET 20;
```

SQLite:

```
SELECT * FROM TableName LIMIT -1 OFFSET 20;
```

Section 14.3: Skipping some rows from result

ISO/ANSI SQL:

```
SELECT Id, Col1
FROM TableName
ORDER BY Id
OFFSET 20 ROWS
```

MySQL:

```
SELECT * FROM TableName LIMIT 20, 42424242424242;
-- skips 20 for take use very large number that is more than rows in table
```

Oracle:

```
SELECT Id,
       Col1
FROM (SELECT Id,
            Col1,
            row_number() over (order by Id) RowNumber
      FROM TableName)
WHERE RowNumber > 20
```

PostgreSQL:

```
SELECT * FROM TableName OFFSET 20;
```

SQLite:

```
SELECT * FROM TableName LIMIT -1 OFFSET 20;
```

第15章：EXCEPT

第15.1节：选择数据集，排除在此其他数据集集中的值

```
--数据集模式必须相同
SELECT '数据1' as '列' UNION ALL
SELECT '数据2' as '列' UNION ALL
SELECT '数据3' as '列' UNION ALL
SELECT '数据4' as '列' UNION ALL
SELECT '数据5' as '列'
EXCEPT
SELECT '数据3' as '列'
--返回数据1、数据2、数据4和数据5
```

Chapter 15: EXCEPT

Section 15.1: Select dataset except where values are in this other dataset

```
--dataset schemas must be identical
SELECT 'Data1' as 'Column' UNION ALL
SELECT 'Data2' as 'Column' UNION ALL
SELECT 'Data3' as 'Column' UNION ALL
SELECT 'Data4' as 'Column' UNION ALL
SELECT 'Data5' as 'Column'
EXCEPT
SELECT 'Data3' as 'Column'
--Returns Data1, Data2, Data4, and Data5
```


第16章：EXPLAIN 和 DESCRIBE

第16.1节：EXPLAIN 选择查询

在 select 查询前加上 Explain 会显示查询的执行方式。这样你可以看到查询是否使用了索引，或者是否可以通过添加索引来优化查询。

示例查询：

```
explain select * from user join data on user.test = data.fk_user;
```

示例结果：

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	user	index	test	test	5	(null)	1	使用 where; 使用 index
1	SIMPLE	data	ref	fk_user	fk_user	5	user.test	1	(null)

在 type 列中你可以看到是否使用了索引。在 possible_keys 列中你可以看到执行计划是否可以从不同索引中选择，或者是否不存在索引。 key 告诉你实际使用的索引。 key_len 显示一个索引项的字节大小。该值越小，越多的索引项可以装入相同的内存大小，从而可以更快地处理。 rows 显示查询预计需要扫描的行数，数值越低越好。

第16.2节：DESCRIBE tablename;

DESCRIBE 和 EXPLAIN 是同义词。对表名使用 DESCRIBE 会返回列的定义。

```
DESCRIBE tablename;
```

示例结果：

COLUMN_NAME	COLUMN_TYPE	IS_NULLABLE	COLUMN_KEY	COLUMN_DEFAULT	EXTRA
id	int(11)	NO	PRI	0	auto_increment
test	varchar(255)	YES		(null)	

这里显示了列名，随后是列的类型。它显示该列是否允许null，以及该列是否使用索引。还显示了默认值，以及表是否包含任何特殊行为，如 auto_increment。

Chapter 16: EXPLAIN and DESCRIBE

Section 16.1: EXPLAIN Select query

An Explain infront of a select query shows you how the query will be executed. This way you to see if the query uses an index or if you could optimize your query by adding an index.

Example query:

```
explain select * from user join data on user.test = data.fk_user;
```

Example result:

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	user	index	test	test	5	(null)	1	Using where; Using index
1	SIMPLE	data	ref	fk_user	fk_user	5	user.test	1	(null)

on type you see if an index was used. In the column possible_keys you see if the execution plan can choose from different indexes or if none exists. key tells you the acutal used index. key_len shows you the size in bytes for one index item. The lower this value is the more index items fit into the same memory size an they can be faster processed. rows shows you the expected number of rows the query needs to scan, the lower the better.

Section 16.2: DESCRIBE tablename;

DESCRIBE and EXPLAIN are synonyms. DESCRIBE on a tablename returns the definition of the columns.

```
DESCRIBE tablename;
```

Exmple Result:

COLUMN_NAME	COLUMN_TYPE	IS_NULLABLE	COLUMN_KEY	COLUMN_DEFAULT	EXTRA
id	int(11)	NO	PRI	0	auto_increment
test	varchar(255)	YES		(null)	

Here you see the column names, followed by the columns type. It shows if null is allowed in the column and if the column uses an Index. the default value is also displayed and if the table contains any special behavior like an auto_increment.

第17章：EXISTS子句

第17.1节：EXISTS子句

客户表

编号	名字	姓氏
1	奥兹古尔	奥兹图尔克
2	优素福	梅迪
3	亨利	泰

订单表

编号	客户编号	金额
1	2	123.50
2	3	14.80

获取至少有一个订单的所有客户

```
SELECT * FROM Customer WHERE EXISTS (
    SELECT * FROM Order WHERE Order.CustomerId=Customer.Id
)
```

结果

编号	名字	姓氏
2	优素福	梅迪
3	亨利	泰

获取没有订单的所有客户

```
SELECT * FROM Customer WHERE NOT EXISTS (
    SELECT * FROM Order WHERE Order.CustomerId = Customer.Id
)
```

结果

编号	名字	姓氏
1	奥兹古尔	奥兹图尔克

目的

EXISTS、IN和JOIN有时可以用于相同的结果，然而，它们并不相等：

- EXISTS 应用于检查某个值是否存在于另一张表中
- IN 应用于静态列表
- JOIN 应用于从其他表中检索数据

Chapter 17: EXISTS CLAUSE

Section 17.1: EXISTS CLAUSE

Customer Table

Id	FirstName	LastName
1	Ozgur	Ozturk
2	Youssef	Medi
3	Henry	Tai

Order Table

Id	CustomerId	Amount
1	2	123.50
2	3	14.80

Get all customers with a least one order

```
SELECT * FROM Customer WHERE EXISTS (
    SELECT * FROM Order WHERE Order.CustomerId=Customer.Id
)
```

Result

Id	FirstName	LastName
2	Youssef	Medi
3	Henry	Tai

Get all customers with no order

```
SELECT * FROM Customer WHERE NOT EXISTS (
    SELECT * FROM Order WHERE Order.CustomerId = Customer.Id
)
```

Result

Id	FirstName	LastName
1	Ozgur	Ozturk

Purpose

EXISTS, IN and JOIN could sometime be used for the same result, however, they are not equals :

- EXISTS should be used to check if a value exist in another table
- IN should be used for static list
- JOIN should be used to retrieve data from other(s) table(s)

第18章：JOIN

JOIN 是一种将两个表中的信息合并（连接）的方法。结果是两个表的列拼接集合，连接类型（INNER/OUTER/CROSS 以及 LEFT/RIGHT/FULL，详见下文）和连接条件（两表中行的关联方式）决定了结果。

一张表可以与自身或任何其他表连接。如果需要访问多于两张表的信息，可以在 FROM 子句中指定多个连接。

第18.1节：自连接

一张表可以与自身连接，不同行通过某种条件相匹配。在这种用例中，必须使用别名以区分表的两个不同出现。

在下面的示例中，对于示例数据库 Employees 表中的每个员工，返回一条记录，包含该员工的名字以及对应的经理的名字。由于经理也是员工，因此表与自身连接：

```
SELECT
e.FName AS "Employee",
    m.FName AS "Manager"
FROM
员工 e
连接
员工 m
在 e.ManagerId = m.Id
```

此查询将返回以下数据：

员工	经理
约翰	詹姆斯
迈克尔	詹姆斯
约翰森	约翰

那么这是如何工作的？

原始表包含以下记录：

编号	名字	姓氏	电话号码	经理编号	部门编号	薪水	入职日期
1	詹姆斯	史密斯	1234567890	空		1	1000 01-01-2002
2	约翰	约翰逊	2468101214	1		1	400 23-03-2005
3	迈克尔	威廉姆斯	1357911131	1		2	600 12-05-2009
4	约翰森·史密斯		1212121212	2		1	500 24-07-2016

第一步是创建一个笛卡尔积，包含FROM子句中使用的所有表的所有记录。在本例中是员工表两次，因此中间表将如下所示（我已删除本例中未使用的字段）：

e.Id	e.FName	e.ManagerId	m.Id	m.FName	m.ManagerId
1	詹姆斯	空	1	詹姆斯	空
1	詹姆斯	空	2	约翰	1
1	詹姆斯	空	3	迈克尔	1

Chapter 18: JOIN

JOIN is a method of combining (joining) information from two tables. The result is a stitched set of columns from both tables, defined by the join type (INNER/OUTER/CROSS and LEFT/RIGHT/FULL, explained below) and join criteria (how rows from both tables relate).

A table may be joined to itself or to any other table. If information from more than two tables needs to be accessed, multiple joins can be specified in a FROM clause.

Section 18.1: Self Join

A table may be joined to itself, with different rows matching each other by some condition. In this use case, aliases must be used in order to distinguish the two occurrences of the table.

In the below example, for each Employee in the example database Employees table, a record is returned containing the employee's first name together with the corresponding first name of the employee's manager. Since managers are also employees, the table is joined with itself:

```
SELECT
    e.FName AS "Employee",
    m.FName AS "Manager"
FROM
Employees e
JOIN
Employees m
ON e.ManagerId = m.Id
```

This query will return the following data:

Employee	Manager
John	James
Michael	James
Johnathon	John

So how does this work?

The original table contains these records:

Id	FName	LName	PhoneNumber	ManagerId	DepartmentId	Salary	HireDate
1	James	Smith	1234567890	NULL	1	1000	01-01-2002
2	John	Johnson	2468101214	1	1	400	23-03-2005
3	Michael	Williams	1357911131	1	2	600	12-05-2009
4	Johnathon	Smith	1212121212	2	1	500	24-07-2016

The first action is to create a *Cartesian* product of all records in the tables used in the **FROM** clause. In this case it's the Employees table twice, so the intermediate table will look like this (I've removed any fields not used in this example):

e.Id	e.FName	e.ManagerId	m.Id	m.FName	m.ManagerId
1	James	NULL	1	James	NULL
1	James	NULL	2	John	1
1	James	NULL	3	Michael	1

1	詹姆斯	空	4	约翰纳森	2
2	约翰	1	1	詹姆斯	空
2	约翰	1	2	约翰	1
2	约翰	1	3	迈克尔	1
2	约翰	1	4	约翰纳森	2
3	迈克尔	1	1	詹姆斯	空
3	迈克尔	1	2	约翰	1
3	迈克尔	1	3	迈克尔	1
3	迈克尔	1	4	约翰纳森	2
4	约翰纳森	2	1	詹姆斯	空
4	约翰纳森	2	2	约翰	1
4	约翰纳森	2	3	迈克尔	1
4	约翰纳森	2	4	约翰纳森	2

下一步操作是仅保留满足JOIN条件的记录，因此任何别名为 e 表中
ManagerId 等于别名为 m 表的 Id 的记录：

e.Id	e.FName	e.ManagerId	m.Id	m.FName	m.ManagerId
2	约翰	1	1	詹姆斯	空
3	迈克尔	1	1	詹姆斯	空
4	约翰纳森	2	2	约翰	1

然后，评估SELECT子句中使用的每个表达式以返回此表：

e.FName	m.FName
约翰	詹姆斯
迈克尔	詹姆斯
约翰森	约翰

最后，列名 e.FName 和 m.FName 被它们的别名列名替换，这些别名通过 AS
操作符分配：

员工	经理
约翰	詹姆斯
迈克尔	詹姆斯
约翰森	约翰

第18.2节：内连接和外连接的区别

SQL有多种连接类型，用于指定结果中是否包含（不）匹配的行：INNER JOIN、LEFTOUTER JOIN、RIGHT OUTER JOIN和FULL OUTER JOIN（INNER和OUTER关键字是可选的）。下图强调了这些连接类型之间的区别：蓝色区域表示连接返回的结果，白色区域表示连接不会返回的结果。

1	James	NULL	4	Johnathon	2
2	John	1	1	James	NULL
2	John	1	2	John	1
2	John	1	3	Michael	1
2	John	1	4	Johnathon	2
3	Michael	1	1	James	NULL
3	Michael	1	2	John	1
3	Michael	1	3	Michael	1
3	Michael	1	4	Johnathon	2
4	Johnathon	2	1	James	NULL
4	Johnathon	2	2	John	1
4	Johnathon	2	3	Michael	1
4	Johnathon	2	4	Johnathon	2

The next action is to only keep the records that meet the **JOIN** criteria, so any records where the aliased e table
ManagerId equals the aliased m table Id:

e.Id	e.FName	e.ManagerId	m.Id	m.FName	m.ManagerId
2	John	1	1	James	NULL
3	Michael	1	1	James	NULL
4	Johnathon	2	2	John	1

Then, each expression used within the **SELECT** clause is evaluated to return this table:

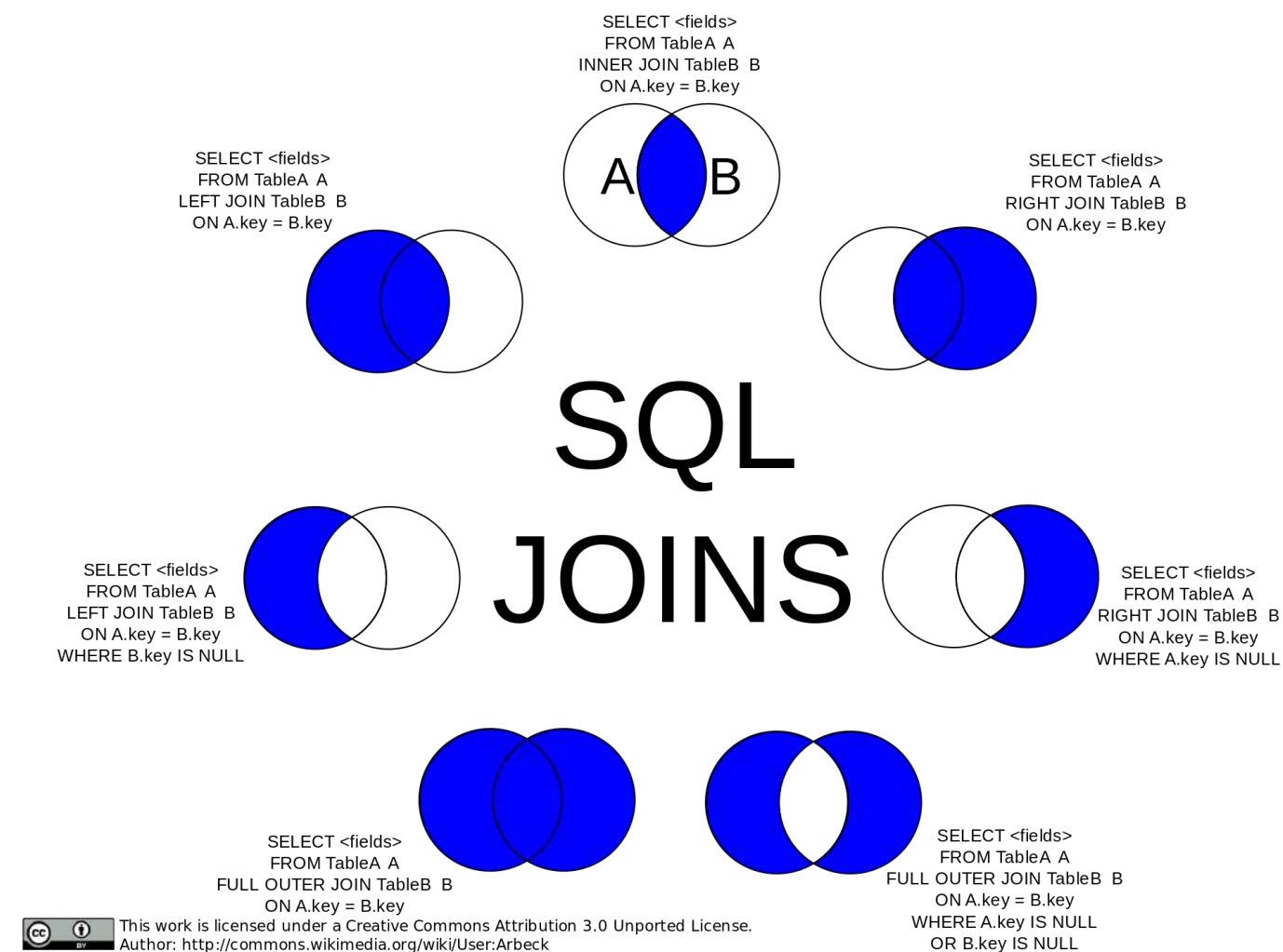
e.FName	m.FName
John	James
Michael	James
Johnathon	John

Finally, column names e.FName and m.FName are replaced by their alias column names, assigned with the AS
operator:

Employee	Manager
John	James
Michael	James
Johnathon	John

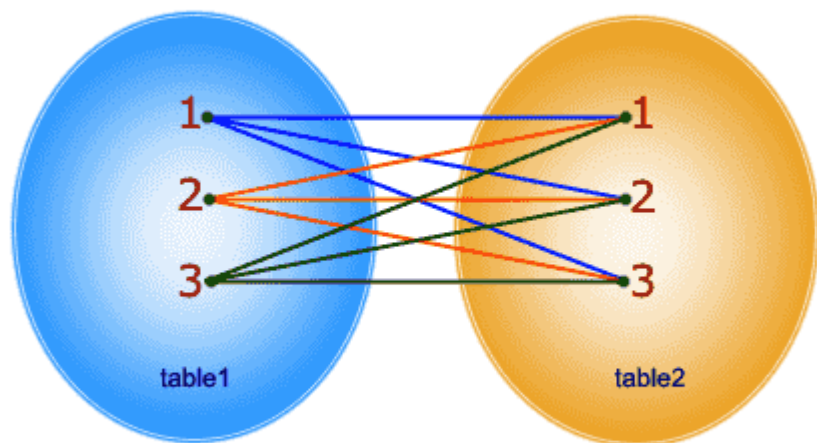
Section 18.2: Differences between inner/outer joins

SQL has various join types to specify whether (non-)matching rows are included in the result: **INNER JOIN**, **LEFT OUTER JOIN**, **RIGHT OUTER JOIN**, and **FULL OUTER JOIN** (the **INNER** and **OUTER** keywords are optional). The figure below underlines the differences between these types of joins: the blue area represents the results returned by the join, and the white area represents the results that the join will not return.



交叉连接SQL图示 (reference) : _____

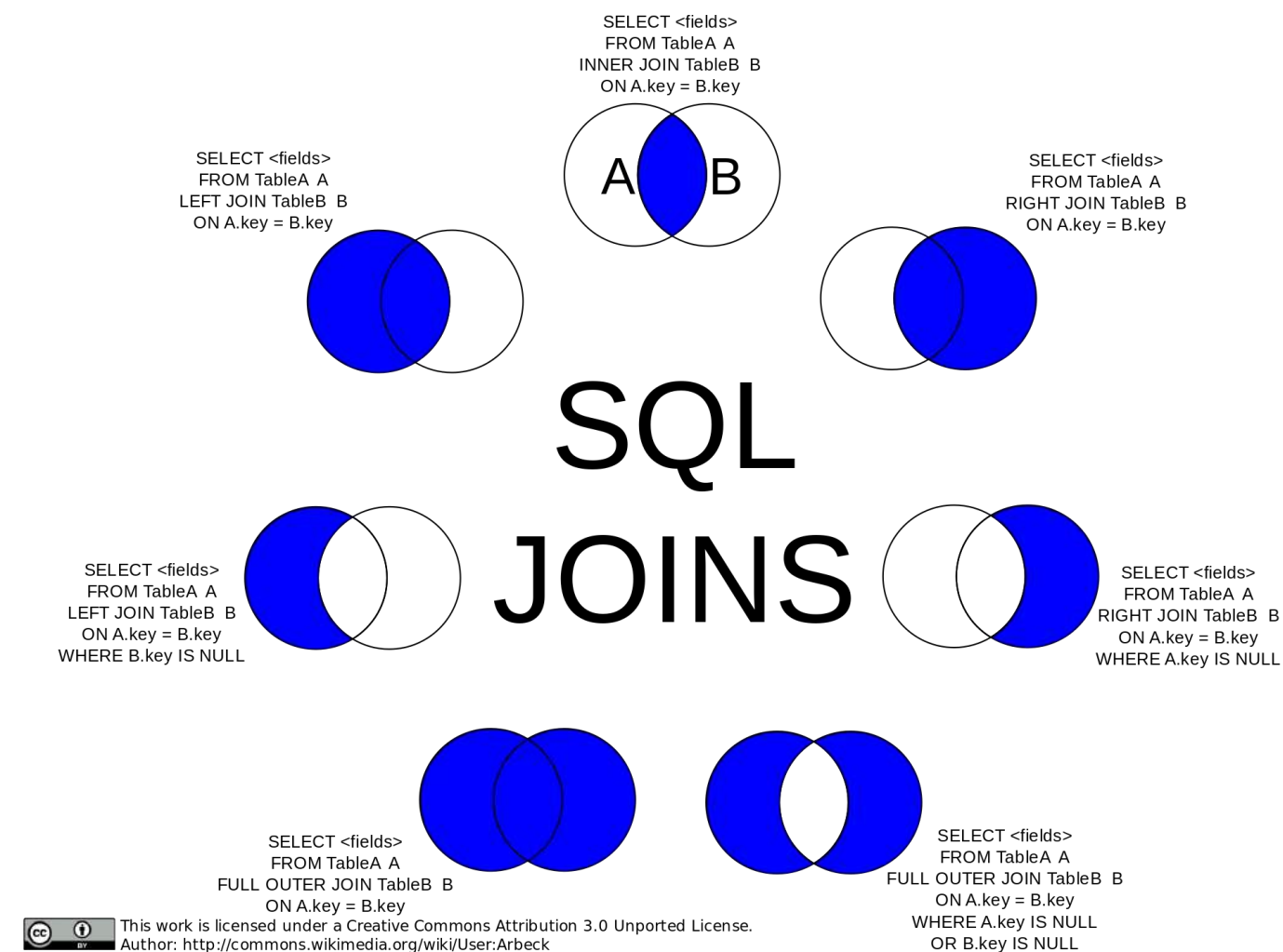
SELECT * FROM table1 CROSS JOIN table2;



In CROSS JOIN, each row from 1st table joins with all the rows of another table. If 1st table contain x rows and y rows in 2nd one the result set will be x * y rows.

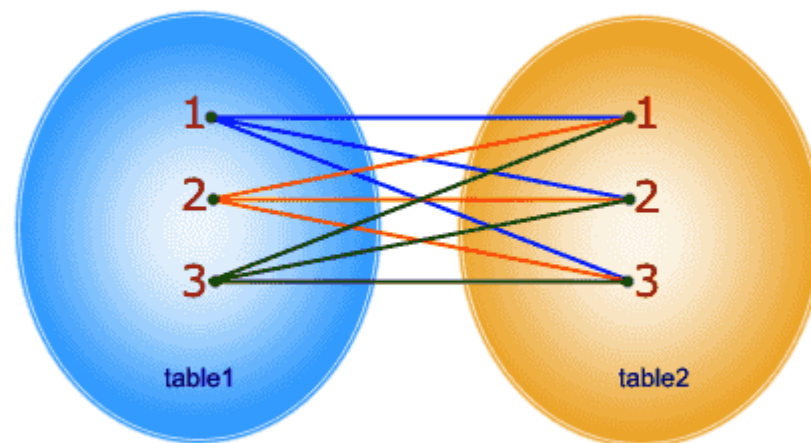
以下是this回答中的示例。 _____

例如，下面有两个表：



Cross Join SQL Pictorial Presentation (reference) :

SELECT * FROM table1 CROSS JOIN table2;



In CROSS JOIN, each row from 1st table joins with all the rows of another table. If 1st table contain x rows and y rows in 2nd one the result set will be x * y rows.

Below are examples from [this](#) answer.

For instance there are two tables as below :

A	B
-	-
1	3
2	4
3	5
4	6

注意，(1,2) 是 A 独有的，(3,4) 是共有的，(5,6) 是 B 独有的。

内连接

使用任一等效查询进行内连接，将得到两个表的交集，即它们共有的两行：

```
select * from a INNER JOIN b on a.a = b.b;
select a.*,b.* from a,b where a.a = b.b;
```

a	b
--+--	
3	3
4	4

左外连接

左外连接将返回表A中的所有行，以及表B中匹配的行：

```
select * from a LEFT OUTER JOIN b on a.a = b.b;
```

a	b
--+-----	
1	null
2	null
3	3
4	4

右外连接

类似地，右外连接将返回表B中的所有行，以及表A中匹配的行：

```
select * from a RIGHT OUTER JOIN b on a.a = b.b;
```

a	b
-----+-----	
3	3
4	4
null	5
null	6

全外连接

全外连接将返回A和B的并集，即A中的所有行和B中的所有行。如果A中的某条数据在B中没有对应的数据，则B部分为null，反之亦然。

```
select * from a FULL OUTER JOIN b on a.a = b.b;
```

a	b
-----+-----	

A	B
-	-
1	3
2	4
3	5
4	6

Note that (1,2) are unique to A, (3,4) are common, and (5,6) are unique to B.

Inner Join

An inner join using either of the equivalent queries gives the intersection of the two tables, i.e. the two rows they have in common:

```
select * from a INNER JOIN b on a.a = b.b;
select a.*,b.* from a,b where a.a = b.b;
```

a	b
--+--	
3	3
4	4

Left outer join

A left outer join will give all rows in A, plus any common rows in B:

```
select * from a LEFT OUTER JOIN b on a.a = b.b;
```

a	b
--+-----	
1	null
2	null
3	3
4	4

Right outer join

Similarly, a right outer join will give all rows in B, plus any common rows in A:

```
select * from a RIGHT OUTER JOIN b on a.a = b.b;
```

a	b
-----+-----	
3	3
4	4
null	5
null	6

Full outer join

A full outer join will give you the union of A and B, i.e., all the rows in A and all the rows in B. If something in A doesn't have a corresponding datum in B, then the B portion is null, and vice versa.

```
select * from a FULL OUTER JOIN b on a.a = b.b;
```

a	b
-----+-----	

```
1 | null
2 | null
3 | 3
4 | 4
null | 6
null | 5
```

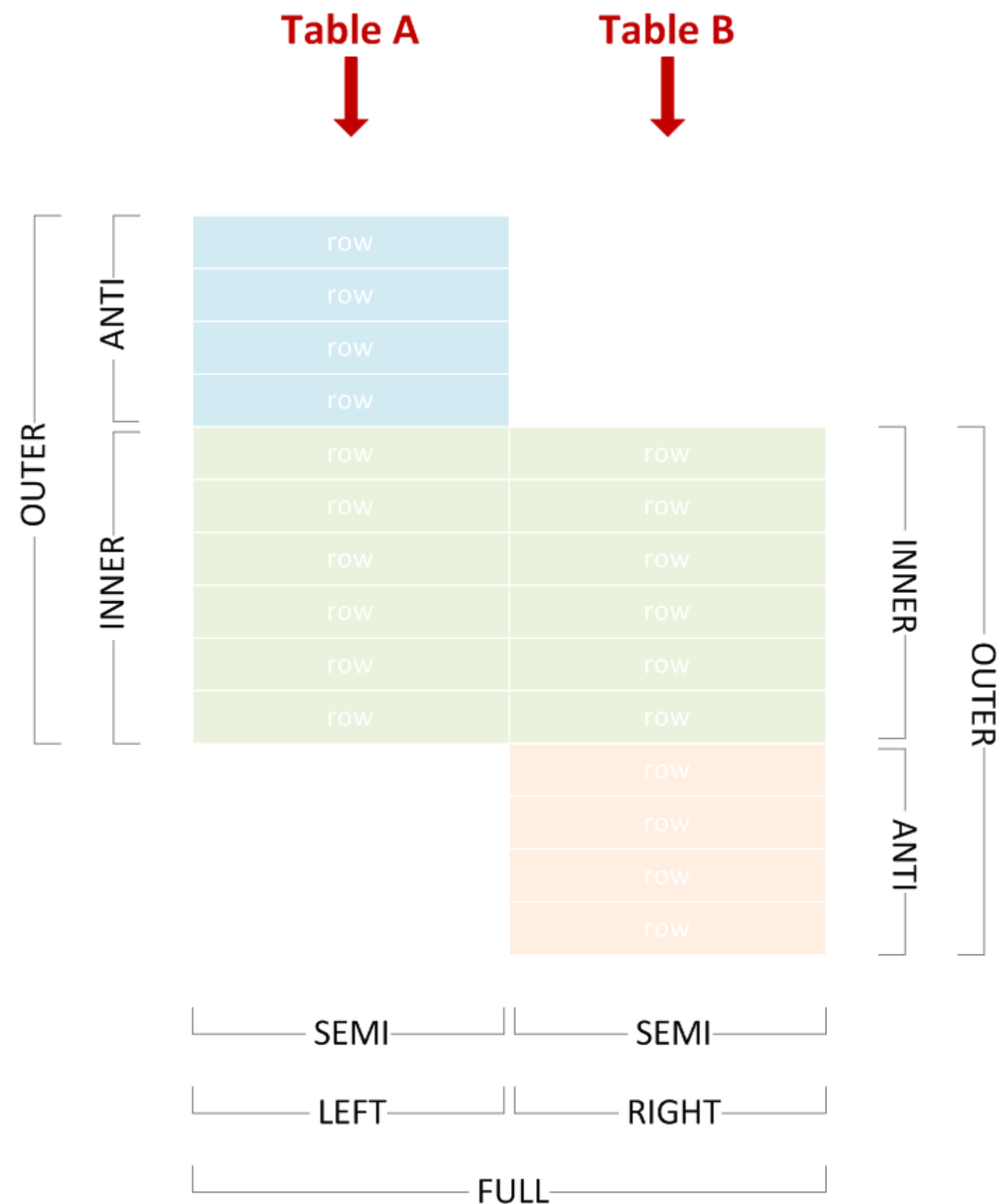
第18.3节：连接术语：内连接、外连接、半连接、反连接等

假设我们有两个表（A和B），它们的一些行匹配（相对于给定的连接条件，无论在具体情况下是什么）：

```
1 | null
2 | null
3 | 3
4 | 4
null | 6
null | 5
```

Section 18.3: JOIN Terminology: Inner, Outer, Semi, Anti..

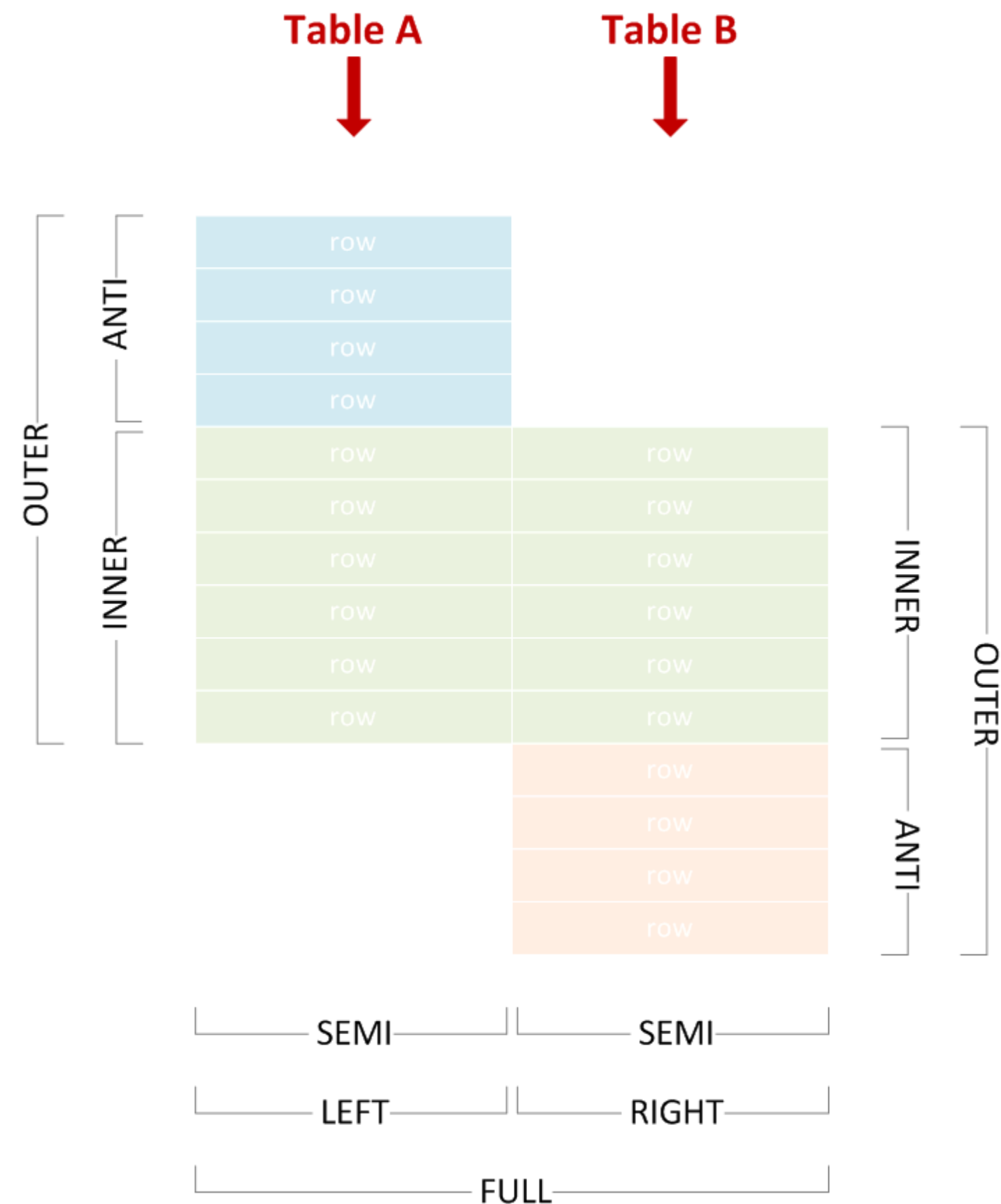
Let's say we have two tables (A and B) and some of their rows match (relative to the given JOIN condition, whatever it may be in the particular case):



我们可以使用各种连接类型来包含或排除任一方的匹配或不匹配行，并通过从上图中选择相应的术语来正确命名连接类型。

以下示例使用以下测试数据：

```
CREATE TABLE A (
  X varchar(255) PRIMARY KEY
```



We can use various join types to include or exclude matching or non-matching rows from either side, and correctly name the join by picking the corresponding terms from the diagram above.

The examples below use the following test data:

```
CREATE TABLE A (
  X varchar(255) PRIMARY KEY
```

```
);

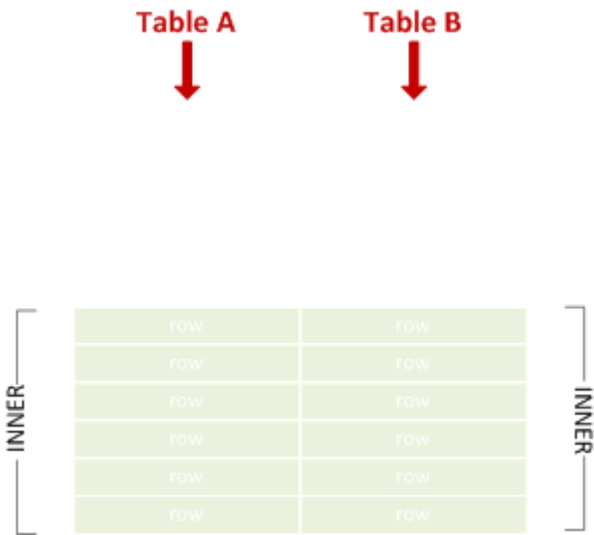
CREATE TABLE B (
  Y varchar(255) PRIMARY KEY
);

INSERT INTO A VALUES
  ('Amy'),
  ('John'),
  ('Lisa'),
  ('Marco'),
  ('Phil');

INSERT INTO B VALUES
  ('Lisa'),
  ('Marco'),
  ('Phil'),
  ('Tim'),
  ('Vincent');
```

内连接

合并匹配的左表和右表行。



```
SELECT * FROM A JOIN B ON X = Y;
```

X	Y
Lisa	Lisa
Marco	Marco
Phil	Phil

左外连接

有时缩写为“左连接”。结合匹配的左表和右表行，并包含不匹配的左表行。

```
);

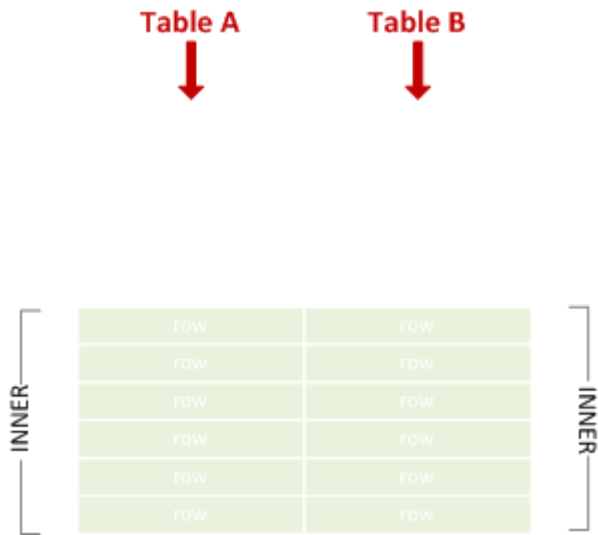
CREATE TABLE B (
  Y varchar(255) PRIMARY KEY
);

INSERT INTO A VALUES
  ('Amy'),
  ('John'),
  ('Lisa'),
  ('Marco'),
  ('Phil');

INSERT INTO B VALUES
  ('Lisa'),
  ('Marco'),
  ('Phil'),
  ('Tim'),
  ('Vincent');
```

Inner Join

Combines left and right rows that match.

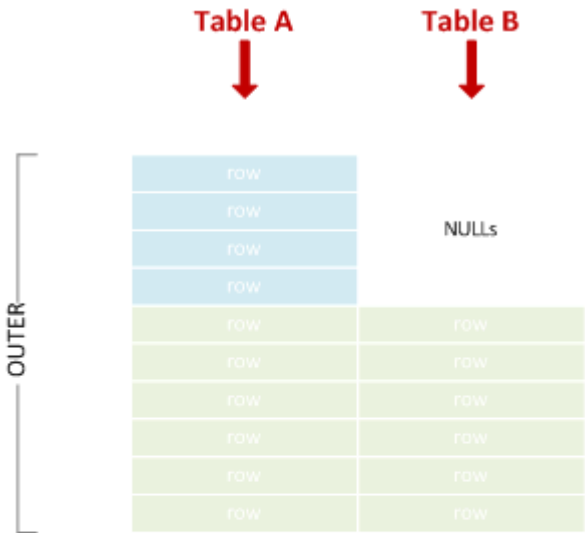


```
SELECT * FROM A JOIN B ON X = Y;
```

X	Y
Lisa	Lisa
Marco	Marco
Phil	Phil

Left Outer Join

Sometimes abbreviated to "left join". Combines left and right rows that match, and includes non-matching left rows.

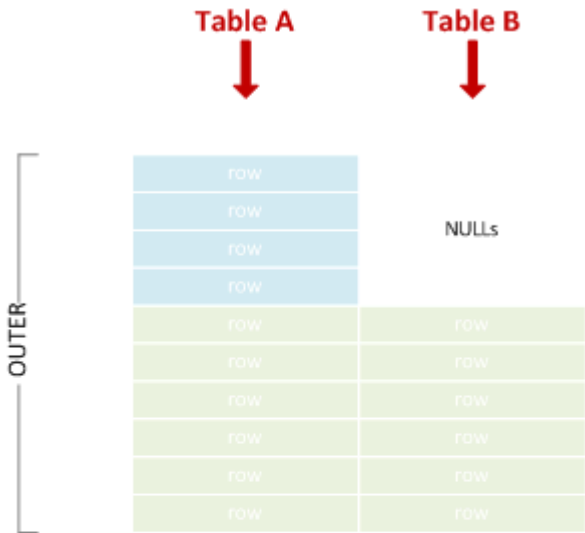


```
SELECT * FROM A LEFT JOIN B ON X = Y;
```

X	Y
Amy	NULL
John	NULL
Lisa	Lisa
Marco	Marco
Phil	Phil

右外连接

有时缩写为“右连接”。结合匹配的左表和右表行，并包含不匹配的右表行。

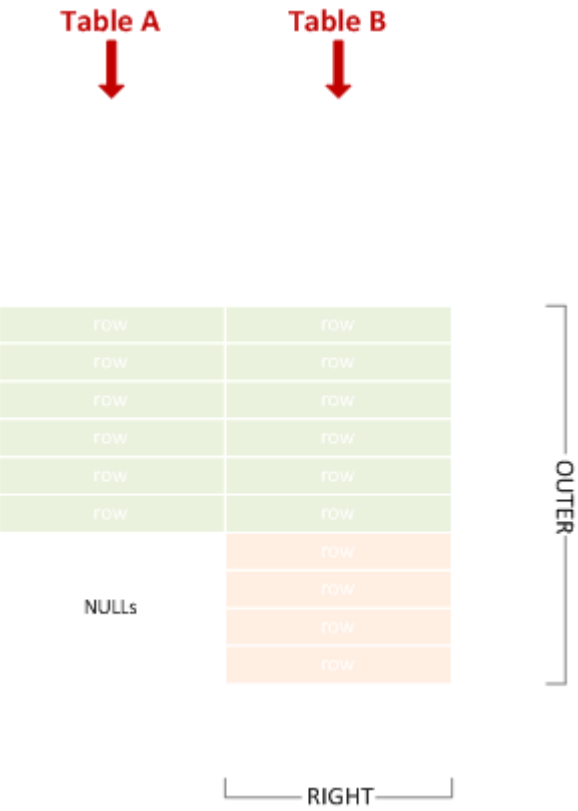


```
SELECT * FROM A RIGHT JOIN B ON X = Y;
```

X	Y
Amy	NULL
John	NULL
Lisa	Lisa
Marco	Marco
Phil	Phil

Right Outer Join

Sometimes abbreviated to "right join". Combines left and right rows that match, and includes non-matching right rows.

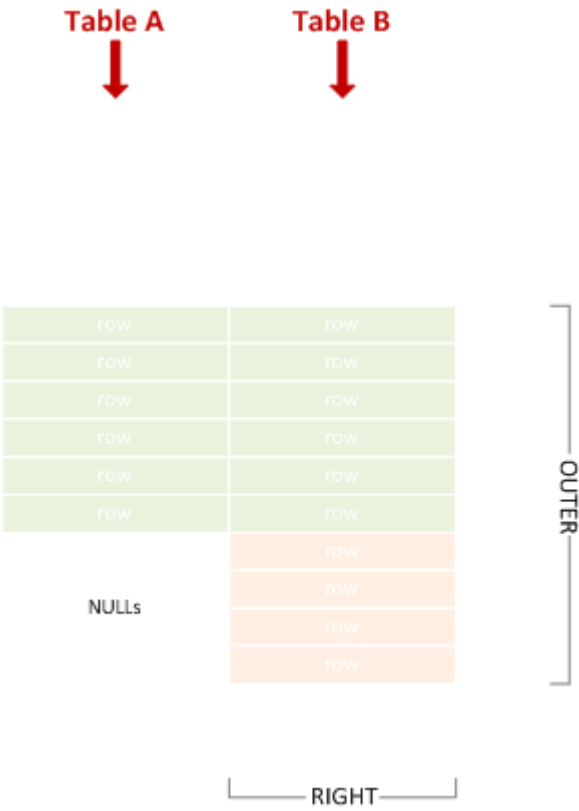


```
SELECT * FROM A RIGHT JOIN B ON X = Y;
```

X	Y
Lisa	Lisa
Marco	Marco
Phil	Phil
NULL	Tim
NULL	Vincent

全外连接

有时缩写为“全连接”。是左外连接和右外连接的并集。

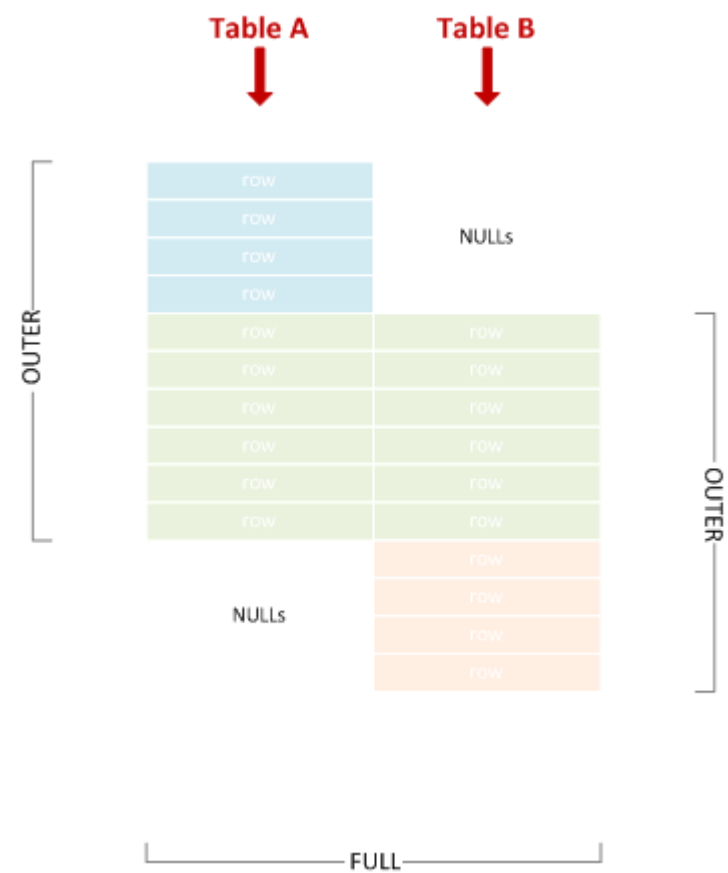


```
SELECT * FROM A RIGHT JOIN B ON X = Y;
```

X	Y
Lisa	Lisa
Marco	Marco
Phil	Phil
NULL	Tim
NULL	Vincent

Full Outer Join

Sometimes abbreviated to "full join". Union of left and right outer join.

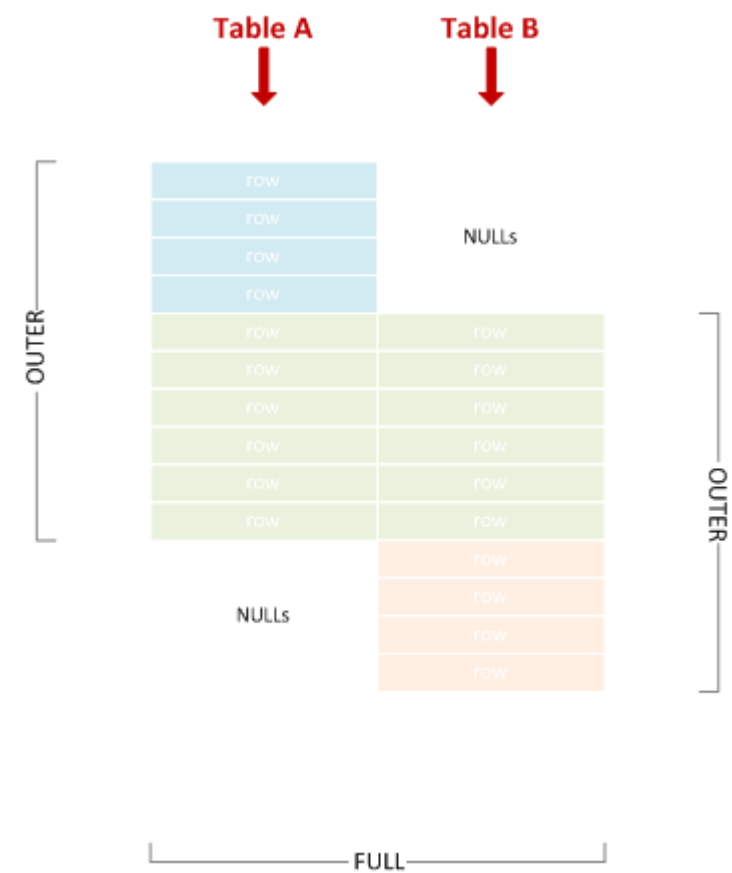


```
SELECT * FROM A FULL JOIN B ON X = Y;
```

X	Y
Amy	NULL
John	NULL
Lisa	Lisa
Marco	Marco
Phil	Phil
NULL	Tim
NULL	Vincent

左半连接

包含与右表匹配的左表行。



```
SELECT * FROM A FULL JOIN B ON X = Y;
```

X	Y
Amy	NULL
John	NULL
Lisa	Lisa
Marco	Marco
Phil	Phil
NULL	Tim
NULL	Vincent

Left Semi Join

Includes left rows that match right rows.

Table A
↓
Table B
↓

row
row
row
row
row
row

SEMI
LEFT

```
SELECT * FROM A WHERE X IN (SELECT Y FROM B);  
  
X  
-----  
Lisa  
Marco  
Phil
```

右半连接

包含与左表匹配的右表行。

Table A
↓
Table B
↓

row
row
row
row
row
row

SEMI
LEFT

```
SELECT * FROM A WHERE X IN (SELECT Y FROM B);  
  
X  
-----  
Lisa  
Marco  
Phil
```

Right Semi Join

Includes right rows that match left rows.

Table A
↓
Table B
↓

row
row
row
row
row
row

SEMI
RIGHT

```
SELECT * FROM B WHERE Y IN (SELECT X FROM A);
```

Y

Lisa
Marco
Phil

如您所见，左半连接和右半连接没有专门的 IN 语法——我们仅通过切换 SQL 文本中的表位置来实现该效果。

左反半连接

包含左表中不匹配右表的行。

Table A
↓
Table B
↓

row
row
row
row
row
row

SEMI
RIGHT

```
SELECT * FROM B WHERE Y IN (SELECT X FROM A);
```

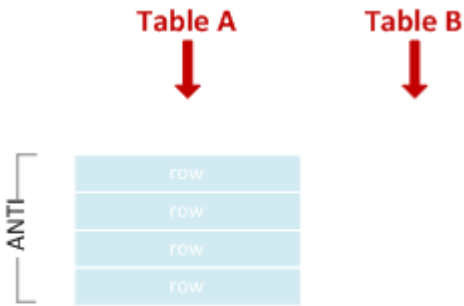
Y

Lisa
Marco
Phil

As you can see, there is no dedicated IN syntax for left vs. right semi join - we achieve the effect simply by switching the table positions within SQL text.

Left Anti Semi Join

Includes left rows that do **not** match right rows.



```
SELECT * FROM A WHERE X NOT IN (SELECT Y FROM B);
```

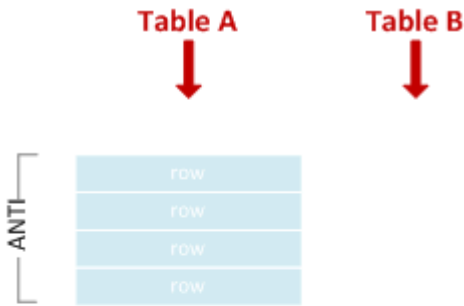
X

艾米
约翰

警告：如果您使用的是可为空的列上的 NOT IN，请务必小心！更多详情 [here](#)。

右反半连接

包含与左侧行不匹配的右侧行。



```
SELECT * FROM A WHERE X NOT IN (SELECT Y FROM B);
```

X

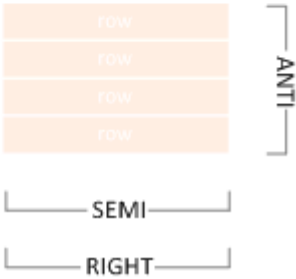
Amy
John

WARNING: Be careful if you happen to be using NOT IN on a NULL-able column! More details [here](#).

Right Anti Semi Join

Includes right rows that do **not** match left rows.

Table A
↓
Table B
↓



```
SELECT * FROM B WHERE Y NOT IN (SELECT X FROM A);
```

Y

Tim
文森特

如你所见，针对左反半连接和右反半连接，并没有专门的 NOT IN 语法——我们仅通过在 SQL 语句中切换表的位置来实现该效果。

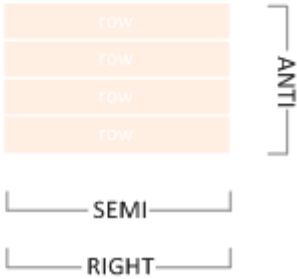
笛卡尔积连接

左表与右表所有行的笛卡尔积。

```
SELECT * FROM A CROSS JOIN B;
```

X	Y
艾米	丽莎
约翰	丽莎
丽莎	丽莎
马可	丽莎
菲尔	丽莎
艾米	马可
约翰	马可
丽莎	马可
马可	马可
菲尔	马可
艾米	菲尔
约翰	菲尔
丽莎	菲尔
马可	菲尔
菲尔	菲尔
艾米	蒂姆

Table A
↓
Table B
↓



```
SELECT * FROM B WHERE Y NOT IN (SELECT X FROM A);
```

Y

Tim
Vincent

As you can see, there is no dedicated NOT IN syntax for left vs. right anti semi join - we achieve the effect simply by switching the table positions within SQL text.

Cross Join

A Cartesian product of all left with all right rows.

```
SELECT * FROM A CROSS JOIN B;
```

X	Y
Amy	Lisa
John	Lisa
Lisa	Lisa
Marco	Lisa
Phil	Lisa
Amy	Marco
John	Marco
Lisa	Marco
Marco	Marco
Phil	Marco
Amy	Phil
John	Phil
Lisa	Phil
Marco	Phil
Phil	Phil
Amy	Tim

约翰 蒂姆
丽莎 蒂姆
马可 蒂姆
菲尔 蒂姆
艾米 文森特
约翰 文森特
丽莎 文森特
马可 文森特
菲尔 文森特

交叉连接等同于一个连接条件始终匹配的内连接，因此以下查询将返回相同的结果：

```
SELECT * FROM A JOIN B ON 1 = 1;
```

自连接

这仅表示一个表与其自身连接。自连接可以是上述讨论的任何连接类型。例如，下面是一个内自连接：

```
SELECT * FROM A A1 JOIN A A2 ON LEN(A1.X) < LEN(A2.X);
```

X	X
Amy	John
Amy	Lisa
Amy	Marco
John	Marco
Lisa	Marco
Phil	Marco
Amy	Phil

第18.4节：左外连接

左外连接（也称为左连接或外连接）是一种连接，确保左表的所有行都被表示；如果右表中没有匹配的行，则其对应字段为NULL。

以下示例将选择所有部门及在该部门工作的员工的名字。
没有员工的部门仍会出现在结果中，但员工姓名将显示为NULL：

```
SELECT Departments.Name, Employees.FName  
FROM Departments  
LEFT OUTER JOIN 员工表  
ON 部门表.Id = 员工表.DepartmentId
```

这将从示例数据库返回以下内容：

部门名称	员工名字
人力资源部	詹姆斯
人力资源部	约翰
人力资源部	约翰森
销售部	迈克尔
技术部	空

那么这是如何工作的？

John Tim
Lisa Tim
Marco Tim
Phil Tim
Amy Vincent
John Vincent
Lisa Vincent
Marco Vincent
Phil Vincent

Cross join is equivalent to an inner join with join condition which always matches, so the following query would have returned the same result:

```
SELECT * FROM A JOIN B ON 1 = 1;
```

Self-Join

This simply denotes a table joining with itself. A self-join can be any of the join types discussed above. For example, this is a an inner self-join:

```
SELECT * FROM A A1 JOIN A A2 ON LEN(A1.X) < LEN(A2.X);
```

X	X
Amy	John
Amy	Lisa
Amy	Marco
John	Marco
Lisa	Marco
Phil	Marco
Amy	Phil

Section 18.4: Left Outer Join

A Left Outer Join (also known as a Left Join or Outer Join) is a Join that ensures all rows from the left table are represented; if no matching row from the right table exists, its corresponding fields are NULL.

The following example will select all departments and the first name of employees that work in that department. Departments with no employees are still returned in the results, but will have NULL for the employee name:

```
SELECT Departments.Name, Employees.FName  
FROM Departments  
LEFT OUTER JOIN Employees  
ON Departments.Id = Employees.DepartmentId
```

This would return the following from the example database:

Departments.Name	Employees.FName
HR	James
HR	John
HR	Johnathon
Sales	Michael
Tech	NULL

So how does this work?

FROM 子句中有两个表：

编号	名字	姓氏	电话号码	经理编号	部门编号	薪水	入职日期
1	詹姆斯	史密斯	1234567890		空	1	1000 01-01-2002
2	约翰	约翰逊	2468101214		1	1	400 23-03-2005
3	迈克尔	威廉姆斯	1357911131		1	2	600 12-05-2009
4	约翰森·史密斯		1212121212		2	1	500 24-07-2016

和

编号	姓名
1	人力资源部
2	销售部
3	技术部

首先，从两个表中创建一个笛卡尔积，生成一个中间表。
满足连接条件 (*Departments.Id = Employees.DepartmentId*) 的记录以加粗显示；这些记录将传递到查询的下一阶段。

由于这是一个左外连接，所有来自连接左侧 (Departments) 的记录都会被返回，而右侧的任何记录如果不符合连接条件，则会被标记为NULL。在下表中，这将返回**Tech**对应的**NULL**

编号	名称	编号	名字	姓氏	电话号码	经理编号	部门编号	薪水	入职日期
1	人力资源部	1	詹姆斯	史密斯	1234567890		空	1	1000 01-01-2002
1	人力资源部	2	约翰	约翰逊	2468101214		1	1	400 23-03-2005
1	人力资源部	3	迈克尔	Williams	1357911131		1	2	600 12-05-2009
1	人力资源部	4	约翰森·史密斯		1212121212		2	1	500 24-07-2016
2	销售1	James		史密斯	1234567890		空	1	1000 01-01-2002
2	销售2	John		约翰逊	2468101214		1	1	400 23-03-2005
2	销售3	Michael		威廉姆斯	1357911131		1	2	600 12-05-2009
2	销售4	JohnathonSmith			1212121212		2	1	500 24-07-2016
3	技术1	James		史密斯	1234567890		空	1	1000 01-01-2002
3	Tech2	约翰		约翰逊	2468101214		1	1	400 23-03-2005
3	Tech3	迈克尔		Williams	1357911131		1	2	600 12-05-2009
3	Tech4	约翰森·史密斯			1212121212		2	1	500 24-07-2016

最后，SELECT 子句中使用的每个表达式都会被计算，以返回我们的最终表格：

部门名称	员工名字
人力资源部	詹姆斯
人力资源部	约翰
销售部	理查德
技术部	空

第18.5节：隐式连接

连接也可以通过在 from 子句中包含多个表，并用逗号，分隔，然后在 where 子句中定义它们之间的关系来实现。这种技术称为隐式连接（因为它实际上不包含 join 子句）。

There are two tables in the FROM clause:

Id	FName	LName	PhoneNumber	ManagerId	DepartmentId	Salary	HireDate
1	James	Smith	1234567890	NULL	1	1000	01-01-2002
2	John	Johnson	2468101214	1	1	400	23-03-2005
3	Michael	Williams	1357911131	1	2	600	12-05-2009
4	Johnathon Smith		1212121212	2	1	500	24-07-2016

and

Id	Name
1	HR
2	Sales
3	Tech

First a *Cartesian* product is created from the two tables giving an intermediate table.
The records that meet the join criteria (*Departments.Id = Employees.DepartmentId*) are highlighted in bold; these are passed to the next stage of the query.

As this is a LEFT OUTER JOIN all records are returned from the LEFT side of the join (Departments), while any records on the RIGHT side are given a NULL marker if they do not match the join criteria. In the table below this will return **Tech** with **NULL**

Id	Name	Id	FName	LName	PhoneNumber	ManagerId	DepartmentId	Salary	HireDate
1	HR	1	James	Smith	1234567890	NULL	1	1000	01-01-2002
1	HR	2	John	Johnson	2468101214	1	1	400	23-03-2005
1	HR	3	Michael	Williams	1357911131	1	2	600	12-05-2009
1	HR	4	Johnathon Smith		1212121212	2	1	500	24-07-2016
2	Sales	1	James	Smith	1234567890	NULL	1	1000	01-01-2002
2	Sales	2	John	Johnson	2468101214	1	1	400	23-03-2005
2	Sales	3	Michael	Williams	1357911131	1	2	600	12-05-2009
2	Sales	4	Johnathon	Smith	1212121212	2	1	500	24-07-2016
3	Tech	1	James	Smith	1234567890	NULL	1	1000	01-01-2002
3	Tech	2	John	Johnson	2468101214	1	1	400	23-03-2005
3	Tech	3	Michael	Williams	1357911131	1	2	600	12-05-2009
3	Tech	4	Johnathon	Smith	1212121212	2	1	500	24-07-2016

Finally each expression used within the **SELECT** clause is evaluated to return our final table:

Departments.Name	Employees.FName
HR	James
HR	John
Sales	Richard
Tech	NULL

Section 18.5: Implicit Join

Joins can also be performed by having several tables in the **from** clause, separated with commas , and defining the relationship between them in the **where** clause. This technique is called an Implicit Join (since it doesn't actually contain a **join** clause).

所有关系型数据库管理系统都支持它，但通常不建议使用这种语法。使用这种语法不好的原因是：

- 可能会意外产生笛卡尔积连接，从而返回错误的结果，尤其是在查询中有很多连接时。
- 如果你本意是进行笛卡尔积连接，那么从语法上看并不清楚（应写出 CROSS JOIN），而且在维护时有人可能会更改它。

下面的示例将选择员工的名字和他们所在部门的名称：

```
SELECT e.FName, d.Name
FROM   Employee e, Departments d
WHERE  e.DepartmentId = d.Id
```

这将从示例数据库返回以下内容：

e.FName	d.Name
詹姆斯	人力资源部
约翰	人力资源部
理查德	销售

第18.6节：交叉连接（CROSS JOIN）

交叉连接执行两个成员的笛卡尔积，笛卡尔积意味着一个表的每一行都会与连接中第二个表的每一行组合。例如，如果TABLEA有20行，TABLEB有20行，结果将是20*20=400行输出。

使用示例数据库

```
SELECT d.Name, e.FName
FROM   Departments d
CROSS JOIN Employees e;
```

返回结果为：

d.Name	e.FName
人力资源部	詹姆斯
人力资源部	约翰
人力资源部	迈克尔
人力资源部	约翰森
销售部	詹姆斯
销售部	约翰
销售部	迈克尔
销售部	约翰森
技术部	詹姆斯
技术部	约翰
技术部	迈克尔
技术部	约翰森

如果你想执行笛卡尔连接，建议明确写出CROSS JOIN，以突出这是你的意图。

All RDBMSs support it, but the syntax is usually advised against. The reasons why it is a bad idea to use this syntax are:

- It is possible to get accidental cross joins which then return incorrect results, especially if you have a lot of joins in the query.
- If you intended a cross join, then it is not clear from the syntax (write out CROSS JOIN instead), and someone is likely to change it during maintenance.

The following example will select employee's first names and the name of the departments they work for:

```
SELECT e.FName, d.Name
FROM   Employee e, Departments d
WHERE  e.DepartmentId = d.Id
```

This would return the following from the example database:

e.FName	d.Name
James	HR
John	HR
Richard	Sales

Section 18.6: CROSS JOIN

Cross join does a Cartesian product of the two members, A Cartesian product means each row of one table is combined with each row of the second table in the join. For example, if TABLEA has 20 rows and TABLEB has 20 rows, the result would be 20*20 = 400 output rows.

Using example database

```
SELECT d.Name, e.FName
FROM   Departments d
CROSS JOIN Employees e;
```

Which returns:

d.Name	e.FName
HR	James
HR	John
HR	Michael
HR	Johnathon
Sales	James
Sales	John
Sales	Michael
Sales	Johnathon
Tech	James
Tech	John
Tech	Michael
Tech	Johnathon

It is recommended to write an explicit CROSS JOIN if you want to do a cartesian join, to highlight that this is what you want.

第18.7节：CROSS APPLY 与 LATERAL JOIN

一种非常有趣的连接类型是 LATERAL JOIN（PostgreSQL 9.3+ 新增），在 SQL-Server 和 Oracle 中也称为 CROSS APPLY/OUTER APPLY。

基本思想是对你连接的每一行都应用一个表值函数（或内联子查询）。

这使得例如只连接另一个表中第一个匹配的条目成为可能。普通连接和 LATERAL 连接的区别在于，你可以在“CROSS APPLY”的子查询中使用之前连接的列。

语法：

PostgreSQL 9.3+

left | right | inner JOIN LATERAL

SQL-Server:

CROSS | OUTER APPLY

INNER JOIN LATERAL 等同于 CROSS APPLY
而 LEFT JOIN LATERAL 等同于 OUTER APPLY

示例用法（PostgreSQL 9.3+）：

```
SELECT * FROM T_Contacts

--LEFT JOIN T_MAP_Contacts_Ref_OrganisationalUnit ON MAP_CTCOU_CT_UID = T_Contacts.CT_UID AND
MAP_CTCOU_SoftDeleteStatus = 1
--WHERE T_MAP_Contacts_Ref_OrganisationalUnit.MAP_CTCOU_UID IS NULL -- 989

LEFT JOIN LATERAL
(
    SELECT
        --MAP_CTCOU_UID
        MAP_CTCOU_CT_UID
        ,MAP_CTCOU_COU_UID
        ,MAP_CTCOU_DateFrom
        ,MAP_CTCOU_DateTo
    FROM T_MAP_Contacts_Ref_OrganisationalUnit
    WHERE MAP_CTCOU_SoftDeleteStatus = 1
    AND MAP_CTCOU_CT_UID = T_Contacts.CT_UID

    /*
和
    (
    (__in_DateFrom <= T_MAP_Contacts_Ref_OrganisationalUnit.MAP_KTKOE_DateTo)
        AND
    (__in_DateTo >= T_MAP_Contacts_Ref_OrganisationalUnit.MAP_KTKOE_DateFrom)
    )
    */
    ORDER BY MAP_CTCOU_DateFrom
    LIMIT 1
```

Section 18.7: CROSS APPLY & LATERAL JOIN

A very interesting type of JOIN is the LATERAL JOIN (new in PostgreSQL 9.3+), which is also known as CROSS APPLY/OUTER APPLY in SQL-Server & Oracle.

The basic idea is that a table-valued function (or inline subquery) gets applied for every row you join.

This makes it possible to, for example, only join the first matching entry in another table. The difference between a normal and a lateral join lies in the fact that you can use a column that you previously joined **in the subquery** that you "CROSS APPLY".

Syntax:

PostgreSQL 9.3+

left | right | inner JOIN **LATERAL**

SQL-Server:

CROSS | OUTER **APPLY**

INNER JOIN LATERAL is the same as CROSS APPLY
and LEFT JOIN LATERAL is the same as OUTER APPLY

Example usage (PostgreSQL 9.3+):

```
SELECT * FROM T_Contacts

--LEFT JOIN T_MAP_Contacts_Ref_OrganisationalUnit ON MAP_CTCOU_CT_UID = T_Contacts.CT_UID AND
MAP_CTCOU_SoftDeleteStatus = 1
--WHERE T_MAP_Contacts_Ref_OrganisationalUnit.MAP_CTCOU_UID IS NULL -- 989

LEFT JOIN LATERAL
(
    SELECT
        --MAP_CTCOU_UID
        MAP_CTCOU_CT_UID
        ,MAP_CTCOU_COU_UID
        ,MAP_CTCOU_DateFrom
        ,MAP_CTCOU_DateTo
    FROM T_MAP_Contacts_Ref_OrganisationalUnit
    WHERE MAP_CTCOU_SoftDeleteStatus = 1
    AND MAP_CTCOU_CT_UID = T_Contacts.CT_UID

    /*
AND
    (
        (__in_DateFrom <= T_MAP_Contacts_Ref_OrganisationalUnit.MAP_KTKOE_DateTo)
        AND
        (__in_DateTo >= T_MAP_Contacts_Ref_OrganisationalUnit.MAP_KTKOE_DateFrom)
    )
    */
    ORDER BY MAP_CTCOU_DateFrom
    LIMIT 1
```

```
) AS FirstOE
```

对于 SQL-Server

```
SELECT * FROM T_Contacts

--LEFT JOIN T_MAP_Contacts_Ref_OrganisationalUnit ON MAP_CTCOU_CT_UID = T_Contacts.CT_UID AND
MAP_CTCOU_SoftDeleteStatus = 1
--WHERE T_MAP_Contacts_Ref_OrganisationalUnit.MAP_CTCOU_UID IS NULL -- 989

-- CROSS APPLY -- = 内连接
OUTER APPLY -- = 左连接
(
    SELECT TOP 1
        --MAP_CTCOU_UID
    MAP_CTCOU_CT_UID
        ,MAP_CTCOU_COU_UID
        ,MAP_CTCOU_DateFrom
        ,MAP_CTCOU_DateTo
    FROM T_MAP_Contacts_Ref_OrganisationalUnit
    WHERE MAP_CTCOU_SoftDeleteStatus = 1
    AND MAP_CTCOU_CT_UID = T_Contacts.CT_UID

    /*
和
    (
    (@in_DateFrom <= T_MAP_Contacts_Ref_OrganisationalUnit.MAP_KTKOE_DateTo)
        AND
    (@in_DateTo >= T_MAP_Contacts_Ref_OrganisationalUnit.MAP_KTKOE_DateFrom)
    )
    */
    ORDER BY MAP_CTCOU_DateFrom
) AS FirstOE
```

第18.8节：FULL JOIN（全连接）

一种较少为人所知的连接类型是FULL JOIN（全连接）。
（注：截至2016年，MySQL不支持FULL JOIN）

FULL OUTER JOIN返回左表的所有行和右表的所有行。

如果左表中有行在右表中没有匹配，或者右表中有行在左表中没有匹配，那么这些行也会被列出。

示例1：

```
SELECT * FROM Table1

FULL JOIN Table2
ON 1 = 2
```

示例2：

```
SELECT
    COALESCE(T_Budget.Year, tYear.Year) AS RPT_BudgetInYear
    ,COALESCE(T_Budget.Value, 0.0) AS RPT_Value
来自 T_Budget

FULL JOIN tfu_RPT_All_CreateYearInterval(@budget_year_from, @budget_year_to) AS tYear
```

```
) AS FirstOE
```

And for SQL-Server

```
SELECT * FROM T_Contacts

--LEFT JOIN T_MAP_Contacts_Ref_OrganisationalUnit ON MAP_CTCOU_CT_UID = T_Contacts.CT_UID AND
MAP_CTCOU_SoftDeleteStatus = 1
--WHERE T_MAP_Contacts_Ref_OrganisationalUnit.MAP_CTCOU_UID IS NULL -- 989

-- CROSS APPLY -- = INNER JOIN
OUTER APPLY -- = LEFT JOIN
(
    SELECT TOP 1
        --MAP_CTCOU_UID
        MAP_CTCOU_CT_UID
        ,MAP_CTCOU_COU_UID
        ,MAP_CTCOU_DateFrom
        ,MAP_CTCOU_DateTo
    FROM T_MAP_Contacts_Ref_OrganisationalUnit
    WHERE MAP_CTCOU_SoftDeleteStatus = 1
    AND MAP_CTCOU_CT_UID = T_Contacts.CT_UID

    /*
AND
    (
        (@in_DateFrom <= T_MAP_Contacts_Ref_OrganisationalUnit.MAP_KTKOE_DateTo)
        AND
        (@in_DateTo >= T_MAP_Contacts_Ref_OrganisationalUnit.MAP_KTKOE_DateFrom)
    )
    */
    ORDER BY MAP_CTCOU_DateFrom
) AS FirstOE
```

Section 18.8: FULL JOIN

One type of JOIN that is less known, is the FULL JOIN.
(Note: FULL JOIN is not supported by MySQL as per 2016)

A FULL OUTER JOIN returns all rows from the left table, and all rows from the right table.

If there are rows in the left table that do not have matches in the right table, or if there are rows in right table that do not have matches in the left table, then those rows will be listed, too.

Example 1：

```
SELECT * FROM Table1

FULL JOIN Table2
ON 1 = 2
```

Example 2:

```
SELECT
    COALESCE(T_Budget.Year, tYear.Year) AS RPT_BudgetInYear
    ,COALESCE(T_Budget.Value, 0.0) AS RPT_Value
FROM T_Budget

FULL JOIN tfu_RPT_All_CreateYearInterval(@budget_year_from, @budget_year_to) AS tYear
```

```
ON tYear.Year = T_Budget.Year
```

请注意，如果您使用软删除，则必须在WHERE子句中再次检查软删除状态（因为FULL JOIN的行为有点像UNION）；
这个细节很容易被忽视，因为您在连接子句中放了AP_SoftDeleteStatus = 1。

此外，如果您使用FULL JOIN，通常必须允许WHERE子句中的NULL；忘记允许某个值为NULL将产生与INNER JOIN相同的效果，而这不是您在使用FULL

JOIN时想要的。

示例：

```
SELECT
T_AccountPlan.AP_UID
,T_AccountPlan.AP_Code
,T_AccountPlan.AP_Lang_EN
,T_BudgetPositions.BUP_Budget
,T_BudgetPositions.BUP_UID
,T_BudgetPositions.BUP_Jahr
FROM T_BudgetPositions

FULL JOIN T_AccountPlan
ON T_AccountPlan.AP_UID = T_BudgetPositions.BUP_AP_UID
AND T_AccountPlan.AP_SoftDeleteStatus = 1

WHERE (1=1)
AND (T_BudgetPositions.BUP_SoftDeleteStatus = 1 OR T_BudgetPositions.BUP_SoftDeleteStatus IS NULL)
AND (T_AccountPlan.AP_SoftDeleteStatus = 1 OR T_AccountPlan.AP_SoftDeleteStatus IS NULL)
```

第18.9节：递归连接

递归连接通常用于获取父子数据。在SQL中，它们通过递归公共表表达式实现，例如：

```
WITH RECURSIVE MyDescendants AS (
SELECT Name
FROM People
WHERE 名称 = 'John Doe'

UNION ALL

SELECT People.Name
FROM People
JOIN MyDescendants ON People.Name = MyDescendants.Parent
)
SELECT * FROM MyDescendants;
```

第18.10节：基本显式内连接

基本连接（也称为“内连接”）从两个表中查询数据，其关系在join子句中定义。

下面的示例将从Employees表中选择员工的名字（FName），以及他们所在部门（Departments表）的名称（Name）：

```
SELECT Employees.FName, Departments.Name
FROM Employees
JOIN Departments
```

```
ON tYear.Year = T_Budget.Year
```

Note that if you're using soft-deletes, you'll have to check the soft-delete status again in the WHERE-clause (becauseFULL JOIN behaves kind-of like a UNION);
It's easy to overlook this little fact, since you put AP_SoftDeleteStatus = 1 in the join clause.

Also, if you are doing a FULL JOIN, you'll usually have to allow NULL in the WHERE-clause; forgetting to allow NULL on a value will have the same effects as an INNER join, which is something you don't want if you're doing a FULL JOIN.

Example:

```
SELECT
T_AccountPlan.AP_UID
,T_AccountPlan.AP_Code
,T_AccountPlan.AP_Lang_EN
,T_BudgetPositions.BUP_Budget
,T_BudgetPositions.BUP_UID
,T_BudgetPositions.BUP_Jahr
FROM T_BudgetPositions

FULL JOIN T_AccountPlan
ON T_AccountPlan.AP_UID = T_BudgetPositions.BUP_AP_UID
AND T_AccountPlan.AP_SoftDeleteStatus = 1

WHERE (1=1)
AND (T_BudgetPositions.BUP_SoftDeleteStatus = 1 OR T_BudgetPositions.BUP_SoftDeleteStatus IS NULL)
AND (T_AccountPlan.AP_SoftDeleteStatus = 1 OR T_AccountPlan.AP_SoftDeleteStatus IS NULL)
```

Section 18.9: Recursive JOINS

Recursive joins are often used to obtain parent-child data. In SQL, they are implemented with recursive common table expressions, for example:

```
WITH RECURSIVE MyDescendants AS (
SELECT Name
FROM People
WHERE Name = 'John Doe'

UNION ALL

SELECT People.Name
FROM People
JOIN MyDescendants ON People.Name = MyDescendants.Parent
)
SELECT * FROM MyDescendants;
```

Section 18.10: Basic explicit inner join

A basic join (also called "inner join") queries data from two tables, with their relationship defined in a join clause.

The following example will select employees' first names (FName) from the Employees table and the name of the department they work for (Name) from the Departments table:

```
SELECT Employees.FName, Departments.Name
FROM Employees
JOIN Departments
```


ON Employees.DepartmentId = Departments.Id

这将从示例数据库返回以下内容：

Employees.FName Departments.Name

詹姆斯	人力资源部
约翰	人力资源部
理查德	销售部

第18.11节：基于子查询的连接

当你想从子表/明细表获取聚合数据并将其与父表/主表的记录一起显示时，通常会使用子查询连接。例如，你可能想获取子记录的计数、子记录中某个数值列的平均值，或者基于日期或数值字段的顶部或底部行。此示例使用别名，这在涉及多张表时可以使查询更易读。下面是一个相当典型的子查询连接示例。在此案例中，我们检索父表采购订单（Purchase Orders）的所有行，并且仅检索子表采购订单明细（PurchaseOrderLineItems）中每个父记录的第一行。

```
SELECT po.Id, po.PODate, po.VendorName, po.Status, item.ItemNo,
       item.Description, item.Cost, item.Price
FROM PurchaseOrders po
LEFT JOIN
    (
        SELECT l.PurchaseOrderId, l.ItemNo, l.Description, l.Cost, l.Price, Min(l.id) as Id
        FROM PurchaseOrderLineItems l
        GROUP BY l.PurchaseOrderId, l.ItemNo, l.Description, l.Cost, l.Price
    ) AS item ON item.PurchaseOrderId = po.Id
```

ON Employees.DepartmentId = Departments.Id

This would return the following from the example database:

Employees.FName Departments.Name

James	HR
John	HR
Richard	Sales

Section 18.11: Joining on a Subquery

Joining a subquery is often used when you want to get aggregate data from a child/details table and display that along with records from the parent/header table. For example, you might want to get a count of child records, an average of some numeric column in child records, or the top or bottom row based on a date or numeric field. This example uses aliases, which arguable makes queries easier to read when you have multiple tables involved. Here's what a fairly typical subquery join looks like. In this case we are retrieving all rows from the parent table Purchase Orders and retrieving only the first row for each parent record of the child table PurchaseOrderLineItems.

```
SELECT po.Id, po.PODate, po.VendorName, po.Status, item.ItemNo,
       item.Description, item.Cost, item.Price
FROM PurchaseOrders po
LEFT JOIN
    (
        SELECT l.PurchaseOrderId, l.ItemNo, l.Description, l.Cost, l.Price, Min(l.id) as Id
        FROM PurchaseOrderLineItems l
        GROUP BY l.PurchaseOrderId, l.ItemNo, l.Description, l.Cost, l.Price
    ) AS item ON item.PurchaseOrderId = po.Id
```


第19章：更新（UPDATE）

第19.1节：使用另一张表的数据进行更新（UPDATE）

下面的示例为任何同时是客户（Customer）且当前在员工（Employees）表中未设置电话号码的员工填写PhoneNumber字段。

（这些示例使用了示例数据库中的员工和客户表。）

标准 SQL

使用相关子查询更新：

```
更新
员工
设置 电话号码 =
    (选择
c. 电话号码
    来自
客户 c
    条件是
c. 名 = 员工. 名
    且 c. 姓 = 员工. 姓)
条件是 员工. 电话号码 为空
```

SQL:2003

使用MERGE更新：

```
合并到
    员工 e
使用
客户 c
    条件是
e.FName = c.FName
    AND e.LName = c.LName
    AND e.PhoneNumber IS NULL
WHEN MATCHED THEN
    UPDATE
        SET PhoneNumber = c.PhoneNumber
```

SQL Server

使用INNER JOIN更新：

```
更新
Employees
SET
PhoneNumber = c.PhoneNumber
FROM
Employees e
INNER JOIN Customers c
    ON e.FName = c.FName
    AND e.LName = c.LName
WHERE
PhoneNumber IS NULL
```

Chapter 19: UPDATE

Section 19.1: UPDATE with data from another table

The examples below fill in a PhoneNumber for any Employee who is also a Customer and currently does not have a phone number set in the Employees Table.

(These examples use the Employees and Customers tables from the Example Databases.)

Standard SQL

Update using a correlated subquery:

```
UPDATE
    Employees
SET PhoneNumber =
    (SELECT
        c.PhoneNumber
    FROM
        Customers c
    WHERE
        c.FName = Employees.FName
        AND c.LName = Employees.LName)
WHERE Employees.PhoneNumber IS NULL
```

SQL:2003

Update using MERGE:

```
MERGE INTO
    Employees e
USING
    Customers c
ON
    e.FName = c.FName
    AND e.LName = c.LName
    AND e.PhoneNumber IS NULL
WHEN MATCHED THEN
    UPDATE
        SET PhoneNumber = c.PhoneNumber
```

SQL Server

Update using INNER JOIN:

```
UPDATE
    Employees
SET
    PhoneNumber = c.PhoneNumber
FROM
    Employees e
INNER JOIN Customers c
    ON e.FName = c.FName
    AND e.LName = c.LName
WHERE
    PhoneNumber IS NULL
```

第19.2节：修改现有值

此示例使用示例数据库中的汽车表（Cars Table）。

```
UPDATE Cars
SET TotalCost = TotalCost + 100
WHERE Id = 3 or Id = 4
```

更新操作可以包含被更新行中的当前值。在这个简单的例子中，TotalCost 对两行增加了100：

- 汽车编号3的TotalCost从100增加到200
- 汽车编号4的TotalCost从1254增加到1354

某列的新值可以由其先前的值或同一表中或连接表中任何其他列的值推导而来。

第19.3节：更新指定行

此示例使用示例数据库中的汽车表（Cars Table）。

```
更新
汽车
设置
      Status = 'READY'
条件
Id = 4
```

该语句将把'汽车'表中id为4的行的状态设置为“READY”。

WHERE子句包含一个对每行进行评估的逻辑表达式。如果某行满足条件，则其值被更新。否则，该行保持不变。

第19.4节：更新所有行

此示例使用示例数据库中的汽车表（Cars Table）。

```
UPDATE Cars
SET 状态 = '准备好'
```

该语句将把“Cars”表中所有行的“状态”列设置为“准备好”，因为它没有WHERE子句来过滤行的集合。

第19.5节：捕获已更新的记录

有时需要捕获刚刚更新的记录。

```
CREATE TABLE #TempUpdated(ID INT)

Update 表名 SET 列1 = 42
      OUTPUT inserted.ID INTO #TempUpdated
      WHERE Id > 50
```

Section 19.2: Modifying existing values

This example uses the Cars Table from the Example Databases.

```
UPDATE Cars
SET TotalCost = TotalCost + 100
WHERE Id = 3 or Id = 4
```

Update operations can include current values in the updated row. In this simple example the TotalCost is incremented by 100 for two rows:

- The TotalCost of Car #3 is increased from 100 to 200
- The TotalCost of Car #4 is increased from 1254 to 1354

A column's new value may be derived from its previous value or from any other column's value in the same table or a joined table.

Section 19.3: Updating Specified Rows

This example uses the Cars Table from the Example Databases.

```
UPDATE
      Cars
SET
      Status = 'READY'
WHERE
      Id = 4
```

This statement will set the status of the row of 'Cars' with id 4 to "READY".

WHERE clause contains a logical expression which is evaluated for each row. If a row fulfills the criteria, its value is updated. Otherwise, a row remains unchanged.

Section 19.4: Updating All Rows

This example uses the Cars Table from the Example Databases.

```
UPDATE Cars
SET Status = 'READY'
```

This statement will set the 'status' column of all rows of the 'Cars' table to "READY" because it does not have a WHERE clause to filter the set of rows.

Section 19.5: Capturing Updated records

Sometimes one wants to capture the records that have just been updated.

```
CREATE TABLE #TempUpdated(ID INT)

Update TableName SET Col1 = 42
      OUTPUT inserted.ID INTO #TempUpdated
      WHERE Id > 50
```

第20章：创建数据库

第20.1节：创建数据库

数据库通过以下SQL命令创建：

```
CREATE DATABASE myDatabase;
```

这将创建一个名为 myDatabase 的空数据库，您可以在其中创建表。

Chapter 20: CREATE Database

Section 20.1: CREATE Database

A database is created with the following SQL command:

```
CREATE DATABASE myDatabase ;
```

This would create an empty database named myDatabase where you can create tables.

第21章：CREATE TABLE

参数	详情
tableName 表名	
columns 列	包含表中所有列的“枚举”。详见创建新表了解更多细节。
CREATE TABLE语句用于在数据库中创建新表。表定义包括列列表、列类型及任何完整性约束。	

第21.1节：从SELECT创建表

您可能想要创建一个表的副本：

```
CREATE TABLE ClonedEmployees AS SELECT * FROM Employees;
```

您可以使用SELECT语句的任何其他功能，在将数据传递给新表之前对其进行修改。新表的列是根据所选行自动创建的。

```
CREATE TABLE ModifiedEmployees AS
SELECT Id, CONCAT(FName, " ", LName) AS FullName FROM Employees
WHERE Id > 10;
```

第21.2节：创建新表

一个基本的Employees表，包含ID、员工的名字和姓氏以及他们的电话号码可以使用以下语句创建

```
CREATE TABLE Employees(
    Id int identity(1,1) primary key not null,
    FName varchar(20) not null,
    LName varchar(20) not null,
    PhoneNumber varchar(10) not null
);
```

此示例特定于Transact-SQL

```
CREATE TABLE 在数据库中创建一个新表，后跟表名Employees
```

然后是列名及其属性列表，例如ID

Id int identity(1,1) not null	
值	含义
Id	该列的名称。
int	是数据类型。
identity(1,1)	表示该列将自动生成值，起始值为1，每次递增1新行。
主键	表示该列中的所有值都是唯一的
非空	表示该列不能为空值

第21.3节：使用外键的CREATE TABLE

下面您可以找到带有对表Cities引用的表Employees。

Chapter 21: CREATE TABLE

Parameter	Details
tableName	The name of the table
columns	Contains an 'enumeration' of all the columns that the table have. See Create a New Table for more details.
The CREATE TABLE statement is used create a new table in the database. A table definition consists of a list of columns, their types, and any integrity constraints.	

Section 21.1: Create Table From Select

You may want to create a duplicate of a table:

```
CREATE TABLE ClonedEmployees AS SELECT * FROM Employees;
```

You can use any of the other features of a SELECT statement to modify the data before passing it to the new table. The columns of the new table are automatically created according to the selected rows.

```
CREATE TABLE ModifiedEmployees AS
SELECT Id, CONCAT(FName, " ", LName) AS FullName FROM Employees
WHERE Id > 10;
```

Section 21.2: Create a New Table

A basic Employees table, containing an ID, and the employee's first and last name along with their phone number can be created using

```
CREATE TABLE Employees(
    Id int identity(1,1) primary key not null,
    FName varchar(20) not null,
    LName varchar(20) not null,
    PhoneNumber varchar(10) not null
);
```

This example is specific to [Transact-SQL](#)

```
CREATE TABLE creates a new table in the database, followed by the table name, Employees
```

This is then followed by the list of column names and their properties, such as the ID

Id int identity(1,1) not null	
Value	Meaning
Id	the column's name.
int	is the data type.
identity(1,1)	states that column will have auto generated values starting at 1 and incrementing by 1 for each new row.
primary key	states that all values in this column will have unique values
not null	states that this column cannot have null values

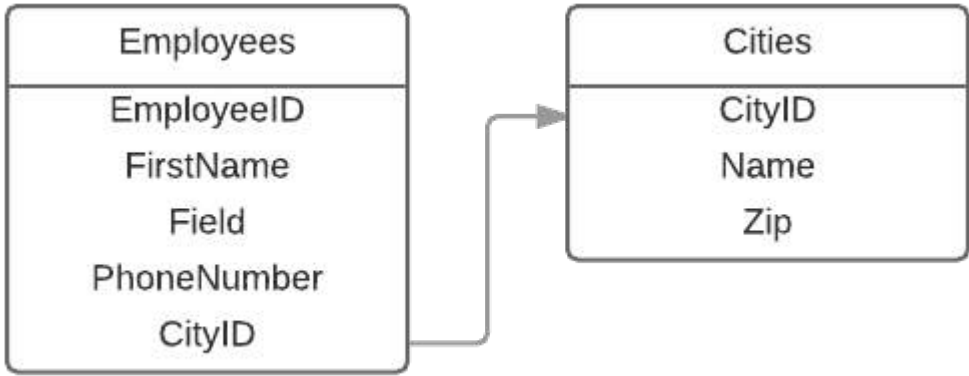
Section 21.3: CREATE TABLE With FOREIGN KEY

Below you could find the table Employees with a reference to the table Cities.

```
CREATE TABLE Cities(
    CityID INT IDENTITY(1,1) NOT NULL,
    Name VARCHAR(20) NOT NULL,
    Zip VARCHAR(10) NOT NULL
);

CREATE TABLE Employees(
    EmployeeID INT IDENTITY (1,1) NOT NULL,
    FirstName VARCHAR(20) NOT NULL,
    LastName VARCHAR(20) NOT NULL,
    PhoneNumber VARCHAR(10) NOT NULL,
    CityID INT FOREIGN KEY REFERENCES Cities(CityID)
);
```

这里您可以找到数据库图示。



表Employees的列CityID将引用表Cities的列CityID。下面您可以找到实现此功能的语法。

CityID INT FOREIGN KEY REFERENCES Cities(CityID)		
值	含义	
CityID	列名	
int	列的类型	
外键	使外键（可选）	
引用	使引用指向表城市的	
城市(CityID)	列CityID	

重要： 你不能引用数据库中不存在的表。请确保先创建表城市，然后再创建表员工。如果顺序相反，将会报错。

第21.4节：复制表

要复制表，只需执行以下操作：

```
CREATE TABLE newtable LIKE oldtable;
INSERT newtable SELECT * FROM oldtable;
```

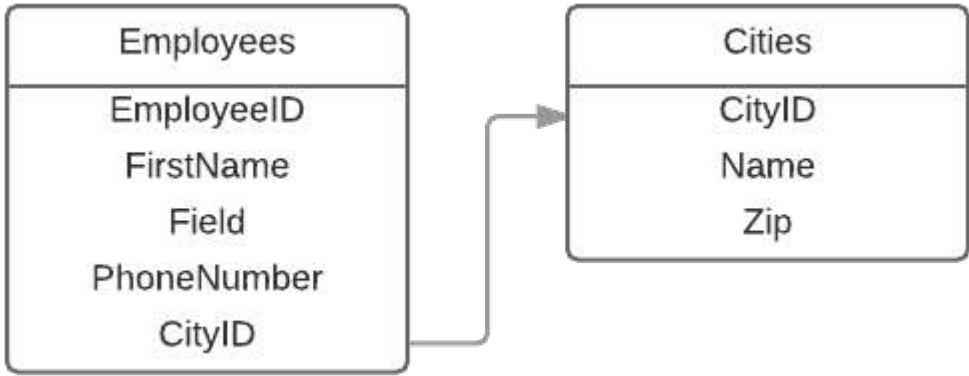
第21.5节：创建临时表或内存表

PostgreSQL 和 SQLite

```
CREATE TABLE Cities(
    CityID INT IDENTITY(1,1) NOT NULL,
    Name VARCHAR(20) NOT NULL,
    Zip VARCHAR(10) NOT NULL
);

CREATE TABLE Employees(
    EmployeeID INT IDENTITY (1,1) NOT NULL,
    FirstName VARCHAR(20) NOT NULL,
    LastName VARCHAR(20) NOT NULL,
    PhoneNumber VARCHAR(10) NOT NULL,
    CityID INT FOREIGN KEY REFERENCES Cities(CityID)
);
```

Here could you find a database diagram.



The column CityID of table Employees will reference to the column CityID of table Cities. Below you could find the syntax to make this.

CityID INT FOREIGN KEY REFERENCES Cities(CityID)		
Value	Meaning	
CityID	Name of the column	
int	type of the column	
FOREIGN KEY	Makes the foreign key (<i>optional</i>)	
REFERENCES	Makes the reference	
Cities(CityID)	to the table Cities column CityID	

Important: You couldn't make a reference to a table that not exists in the database. Be source to make first the table Cities and second the table Employees. If you do it vise versa, it will throw an error.

Section 21.4: Duplicate a table

To duplicate a table, simply do the following:

```
CREATE TABLE newtable LIKE oldtable;
INSERT newtable SELECT * FROM oldtable;
```

Section 21.5: Create a Temporary or In-Memory Table

PostgreSQL and SQLite

创建一个仅对会话本地有效的临时表：

```
CREATE TEMP TABLE MyTable(...);
```

SQL Server

创建一个仅对会话本地有效的临时表：

```
CREATE TABLE #TempPhysical(...);
```

创建一个对所有人可见的临时表：

```
CREATE TABLE ##TempPhysicalVisibleToEveryone(...);
```

创建一个内存表：

```
DECLARE @TempMemory TABLE(...);
```

To create a temporary table local to the session:

```
CREATE TEMP TABLE MyTable(...);
```

SQL Server

To create a temporary table local to the session:

```
CREATE TABLE #TempPhysical(...);
```

To create a temporary table visible to everyone:

```
CREATE TABLE ##TempPhysicalVisibleToEveryone(...);
```

To create an in-memory table:

```
DECLARE @TempMemory TABLE(...);
```


第22章：CREATE FUNCTION

参数	描述
函数名	函数名称
函数接受的参数列表	
返回数据类型	函数返回的类型。某些SQL数据类型 _____
函数体	函数的代码
标量表达式	函数返回的标量值

第22.1节：创建新函数

```
CREATE FUNCTION FirstWord (@input varchar(1000))
RETURNS varchar(1000)
AS
BEGIN
    DECLARE @output varchar(1000)
    SET @output = SUBSTRING(@input, 0, CASE CHARINDEX(' ', @input)
        WHEN 0 THEN LEN(@input) + 1
        ELSE CHARINDEX(' ', @input)
    END)

    RETURN @output
结束
```

此示例创建了一个名为FirstWord的函数，该函数接受一个varchar参数并返回另一个varchar值。

Chapter 22: CREATE FUNCTION

Argument	Description
function_name	the name of function
list_of_paramenters	parameters that function accepts
return_data_type	type that function returs. Some SQL data type
function_body	the code of function
scalar_expression	scalar value returned by function

Section 22.1: Create a new Function

```
CREATE FUNCTION FirstWord (@input varchar(1000))
RETURNS varchar(1000)
AS
BEGIN
    DECLARE @output varchar(1000)
    SET @output = SUBSTRING(@input, 0, CASE CHARINDEX(' ', @input)
        WHEN 0 THEN LEN(@input) + 1
        ELSE CHARINDEX(' ', @input)
    END)

    RETURN @output
END
```

This example creates a function named **FirstWord**, that accepts a varchar parameter and returns another varchar value.

第23章：TRY/CATCH

第23.1节：TRY/CATCH中的事务

由于无效的日期时间，这将回滚两个插入操作：

```
BEGIN TRANSACTION
BEGIN TRY
    INSERT INTO dbo.Sale(Price, SaleDate, Quantity)
    VALUES (5.2, GETDATE(), 1)
    INSERT INTO dbo.Sale(Price, SaleDate, Quantity)
    VALUES (5.2, 'not a date', 1)
    COMMIT TRANSACTION
END TRY
BEGIN CATCH
    THROW
    ROLLBACK TRANSACTION
END CATCH
```

这将提交两个插入操作：

```
BEGIN TRANSACTION
BEGIN TRY
    INSERT INTO dbo.Sale(Price, SaleDate, Quantity)
    VALUES (5.2, GETDATE(), 1)
    INSERT INTO dbo.Sale(Price, SaleDate, Quantity)
    VALUES (5.2, GETDATE(), 1)
    COMMIT TRANSACTION
END TRY
BEGIN CATCH
    THROW
    ROLLBACK TRANSACTION
END CATCH
```

Chapter 23: TRY/CATCH

Section 23.1: Transaction In a TRY/CATCH

This will rollback both inserts due to an invalid datetime:

```
BEGIN TRANSACTION
BEGIN TRY
    INSERT INTO dbo.Sale(Price, SaleDate, Quantity)
    VALUES (5.2, GETDATE(), 1)
    INSERT INTO dbo.Sale(Price, SaleDate, Quantity)
    VALUES (5.2, 'not a date', 1)
    COMMIT TRANSACTION
END TRY
BEGIN CATCH
    THROW
    ROLLBACK TRANSACTION
END CATCH
```

This will commit both inserts:

```
BEGIN TRANSACTION
BEGIN TRY
    INSERT INTO dbo.Sale(Price, SaleDate, Quantity)
    VALUES (5.2, GETDATE(), 1)
    INSERT INTO dbo.Sale(Price, SaleDate, Quantity)
    VALUES (5.2, GETDATE(), 1)
    COMMIT TRANSACTION
END TRY
BEGIN CATCH
    THROW
    ROLLBACK TRANSACTION
END CATCH
```

第24章：UNION / UNION ALL

SQL中的UNION关键字用于合并SELECT语句的结果，且不包含任何重复项。为了使用UNION并合并结果，两个SELECT语句应具有相同数量的列，且数据类型和顺序相同，但列的长度可以不同。

第24.1节：基本的UNION ALL查询

```
CREATE TABLE HR_EMPLOYEES
(
  PersonID int,
  LastName VARCHAR(30),
  FirstName VARCHAR(30),
  Position VARCHAR(30)
);

创建表 FINANCE_EMPLOYEES
(
  PersonID 整数,
  LastName 变长字符(30),
  FirstName 变长字符(30),
  Position 变长字符(30)
);
```

假设我们想提取所有部门中经理的姓名。

使用UNION，我们可以获取人力资源和财务部门中所有职位为经理的员工。

```
SELECT
  FirstName, LastName
FROM
  HR_EMPLOYEES
WHERE
  Position = '经理'
UNION ALL
SELECT
  FirstName, LastName
FROM
  FINANCE_EMPLOYEES
WHERE
  Position = '经理'
```

UNION语句会从查询结果中删除重复行。由于可能存在同名同职位的人员在两个部门中均有，因此我们使用UNION ALL，以避免删除重复项。

如果您想为每个输出列使用别名，只需将它们放在第一个选择语句中，如下所示：

```
SELECT
  FirstName as 'First Name', LastName as 'Last Name'
FROM
  HR_EMPLOYEES
WHERE
  Position = '经理'
UNION ALL
SELECT
  FirstName, LastName
FROM
```

Chapter 24: UNION / UNION ALL

UNION keyword in SQL is used to combine to **SELECT** statement results with out any duplicate. In order to use UNION and combine results both SELECT statement should have same number of column with same data type in same order, but the length of column can be different.

Section 24.1: Basic UNION ALL query

```
CREATE TABLE HR_EMPLOYEES
(
  PersonID int,
  LastName VARCHAR(30),
  FirstName VARCHAR(30),
  Position VARCHAR(30)
);

CREATE TABLE FINANCE_EMPLOYEES
(
  PersonID INT,
  LastName VARCHAR(30),
  FirstName VARCHAR(30),
  Position VARCHAR(30)
);
```

Let's say we want to extract the names of all the managers from our departments.

Using a **UNION** we can get all the employees from both HR and Finance departments, which hold the position of a manager

```
SELECT
  FirstName, LastName
FROM
  HR_EMPLOYEES
WHERE
  Position = 'manager'
UNION ALL
SELECT
  FirstName, LastName
FROM
  FINANCE_EMPLOYEES
WHERE
  Position = 'manager'
```

The **UNION** statement removes duplicate rows from the query results. Since it is possible to have people having the same Name and position in both departments we are using **UNION ALL**, in order not to remove duplicates.

If you want to use an alias for each output column, you can just put them in the first select statement, as follows:

```
SELECT
  FirstName as 'First Name', LastName as 'Last Name'
FROM
  HR_EMPLOYEES
WHERE
  Position = 'manager'
UNION ALL
SELECT
  FirstName, LastName
FROM
```

```
FINANCE_EMPLOYEES
WHERE
Position = '经理'
```

第24.2节：简单说明和示例

简单来说：

- UNION连接两个结果集，同时从结果集中去除重复项
- UNION ALL连接两个结果集，但不尝试去除重复项

许多人常犯的一个错误是使用UNION，尽管他们不需要去除重复项。对于大型结果集，额外的性能开销可能非常显著。

何时可能需要UNION

假设你需要根据两个不同的属性过滤一个表，并且你为每个列创建了单独的非聚集索引。UNION使你能够利用这两个索引，同时仍然防止重复项。

```
SELECT C1, C2, C3 FROM Table1 WHERE C1 = @Param1
UNION
SELECT C1, C2, C3 FROM Table1 WHERE C2 = @Param2
```

这简化了您的性能调优，因为只需要简单的索引即可优化执行这些查询。您甚至可能能够减少相当多的非聚集索引，从而提高源表的整体写入性能。

何时可能需要UNION ALL

假设您仍然需要根据两个属性过滤表，但不需要过滤重复记录（无论是因为这无关紧要，还是由于您的数据模型设计，数据在联合期间不会产生任何重复）。

```
SELECT C1 FROM Table1
UNION ALL
SELECT C1 FROM Table2
```

当创建视图以连接设计为物理分区在多个表中的数据时，这尤其有用（可能是出于性能原因，但仍希望汇总记录）。由于数据已经被拆分，数据库引擎去重没有价值，只会增加查询的额外处理时间。

```
FINANCE_EMPLOYEES
WHERE
Position = 'manager'
```

Section 24.2: Simple explanation and Example

In simple terms:

- UNION joins 2 result sets while removing duplicates from the result set
- UNION ALL joins 2 result sets without attempting to remove duplicates

One mistake many people make is to use a UNION when they do not need to have the duplicates removed. The additional performance cost against large results sets can be very significant.

When you might need UNION

Suppose you need to filter a table against 2 different attributes, and you have created separate non-clustered indexes for each column. A UNION enables you to leverage both indexes while still preventing duplicates.

```
SELECT C1, C2, C3 FROM Table1 WHERE C1 = @Param1
UNION
SELECT C1, C2, C3 FROM Table1 WHERE C2 = @Param2
```

This simplifies your performance tuning since only simple indexes are needed to perform these queries optimally. You may even be able to get by with quite a bit fewer non-clustered indexes improving overall write performance against the source table as well.

When you might need UNION ALL

Suppose you still need to filter a table against 2 attributes, but you do not need to filter duplicate records (either because it doesn't matter or your data wouldn't produce any duplicates during the union due to your data model design).

```
SELECT C1 FROM Table1
UNION ALL
SELECT C1 FROM Table2
```

This is especially useful when creating Views that join data that is designed to be physically partitioned across multiple tables (maybe for performance reasons, but still wants to roll-up records). Since the data is already split, having the database engine remove duplicates adds no value and just adds additional processing time to the queries.

第25章：ALTER TABLE

SQL中的ALTER命令用于修改表中的列或约束

第25.1节：添加列

```
ALTER TABLE Employees
ADD StartingDate date NOT NULL DEFAULT GetDate(),
    DateOfBirth date NULL
```

上述语句将在 Employees 表中添加名为StartingDate的列，该列不能为空，默认值为当前日期；以及名为DateOfBirth的列，该列可以为空。

第25.2节：删除列

```
ALTER TABLE Employees
DROP COLUMN salary;
```

这不仅可以删除该列中的信息，还会从 Employees 表中删除 salary 列（该列将不再存在）。

第25.3节：添加主键

```
ALTER TABLE EMPLOYEES ADD pk_EmployeeID PRIMARY KEY (ID)
```

这将在 Employees 表的字段ID上添加主键。在括号中包含多个列名（包括 ID）将创建复合主键。添加多个列时，列名必须用逗号分隔。

```
ALTER TABLE EMPLOYEES ADD pk_EmployeeID PRIMARY KEY (ID, FName)
```

第25.4节：修改列

```
ALTER TABLE Employees
ALTER COLUMN StartingDate DATETIME NOT NULL DEFAULT (GETDATE())
```

此查询将修改StartingDate列的数据类型，将其从简单的date更改为datetime，并将默认值设置为当前日期。

第25.5节：删除约束

```
ALTER TABLE Employees
DROP CONSTRAINT DefaultSalary
```

这将从员工表定义中删除名为 DefaultSalary 的约束。

注意： 在删除列之前，确保已删除该列的约束。

Chapter 25: ALTER TABLE

ALTER command in SQL is used to modify column/constraint in a table

Section 25.1: Add Column(s)

```
ALTER TABLE Employees
ADD StartingDate date NOT NULL DEFAULT GetDate(),
    DateOfBirth date NULL
```

The above statement would add columns named StartingDate which cannot be NULL with default value as current date and DateOfBirth which can be NULL in Employees table.

Section 25.2: Drop Column

```
ALTER TABLE Employees
DROP COLUMN salary;
```

This will not only delete information from that column, but will drop the column salary from table employees(the column will no more exist).

Section 25.3: Add Primary Key

```
ALTER TABLE EMPLOYEES ADD pk_EmployeeID PRIMARY KEY (ID)
```

This will add a Primary key to the table Employees on the field ID. Including more than one column name in the parentheses along with ID will create a Composite Primary Key. When adding more than one column, the column names must be separated by commas.

```
ALTER TABLE EMPLOYEES ADD pk_EmployeeID PRIMARY KEY (ID, FName)
```

Section 25.4: Alter Column

```
ALTER TABLE Employees
ALTER COLUMN StartingDate DATETIME NOT NULL DEFAULT (GETDATE())
```

This query will alter the column datatype of StartingDate and change it from simple date to datetime and set default to current date.

Section 25.5: Drop Constraint

```
ALTER TABLE Employees
DROP CONSTRAINT DefaultSalary
```

This Drops a constraint called DefaultSalary from the employees table definition.

Note: Ensure that constraints of the column are dropped before dropping a column.

第26章：插入

第26.1节：使用 SELECT 从另一张表插入数据

```
INSERT INTO Customers (FName, LName, PhoneNumber)
SELECT FName, LName, PhoneNumber FROM Employees
```

此示例将把所有员工插入到客户表中。由于两个表的字段不同，且您不想移动所有字段，因此需要设置要插入的字段和要选择的字段。相关字段名称不必相同，但数据类型必须相同。此示例假设 Id 字段已设置为标识规范，并将自动递增。

如果你有两个字段名称完全相同的表，并且只想将所有记录移动过去，可以使用：

```
INSERT INTO Table1
SELECT * FROM Table2
```

第26.2节：插入新行

```
INSERT INTO Customers
VALUES ('Zack', 'Smith', 'zack@example.com', '7049989942', 'EMAIL');
```

该语句将向Customers表中插入一条新行。注意，Id列未指定值，因为它会自动添加。但其他所有列的值必须指定。

第26.3节：仅插入指定列

```
INSERT INTO Customers (FName, LName, Email, PreferredContact)
VALUES ('Zack', 'Smith', 'zack@example.com', 'EMAIL');
```

该语句将向Customers表中插入一条新行。数据只会插入指定的列——注意未为PhoneNumber列提供值。但必须包含所有标记为not null的列。

第26.4节：一次插入多行

可以使用单条插入命令插入多行：

```
INSERT INTO tbl_name (field1, field2, field3)
VALUES (1,2,3), (4,5,6), (7,8,9);
```

对于同时插入大量数据（批量插入），存在数据库管理系统（DBMS）特定的功能和建议。

MySQL - [LOAD DATA INFILE](#)

MSSQL - [BULK INSERT](#)

Chapter 26: INSERT

Section 26.1: INSERT data from another table using SELECT

```
INSERT INTO Customers (FName, LName, PhoneNumber)
SELECT FName, LName, PhoneNumber FROM Employees
```

This example will insert all Employees into the Customers table. Since the two tables have different fields and you don't want to move all the fields over, you need to set which fields to insert into and which fields to select. The correlating field names don't need to be called the same thing, but then need to be the same data type. This example is assuming that the Id field has an Identity Specification set and will auto increment.

If you have two tables that have exactly the same field names and just want to move all the records over you can use:

```
INSERT INTO Table1
SELECT * FROM Table2
```

Section 26.2: Insert New Row

```
INSERT INTO Customers
VALUES ('Zack', 'Smith', 'zack@example.com', '7049989942', 'EMAIL');
```

This statement will insert a new row into the Customers table. Note that a value was not specified for the Id column, as it will be added automatically. However, all other column values must be specified.

Section 26.3: Insert Only Specified Columns

```
INSERT INTO Customers (FName, LName, Email, PreferredContact)
VALUES ('Zack', 'Smith', 'zack@example.com', 'EMAIL');
```

This statement will insert a new row into the Customers table. Data will only be inserted into the columns specified - note that no value was provided for the PhoneNumber column. Note, however, that all columns marked as not null must be included.

Section 26.4: Insert multiple rows at once

Multiple rows can be inserted with a single insert command:

```
INSERT INTO tbl_name (field1, field2, field3)
VALUES (1,2,3), (4,5,6), (7,8,9);
```

For inserting large quantities of data (bulk insert) at the same time, DBMS-specific features and recommendations exist.

MySQL - [LOAD DATA INFILE](#)

MSSQL - [BULK INSERT](#)

第27章：MERGE

MERGE（通常也称为UPSERT，即“更新或插入”）允许插入新行，或者如果行已存在，则更新现有行。关键是要原子性地执行整套操作（以保证数据保持一致），并防止在客户端/服务器系统中多条SQL语句的通信开销。

第27.1节：使用MERGE使目标表匹配源表

```
MERGE INTO targetTable t
  USING sourceTable s
    ON t.PKID = s.PKID
  WHEN MATCHED AND NOT EXISTS (
    SELECT s.ColumnA, s.ColumnB, s.ColumnC
    INTERSECT
    SELECT t.ColumnA, t.ColumnB, s.ColumnC
  )
    然后更新设置
t.ColumnA = s.ColumnA
  ,t.ColumnB = s.ColumnB
  ,t.ColumnC = s.ColumnC
  当目标中未匹配时
    然后插入 (PKID, ColumnA, ColumnB, ColumnC)
    值 (s.PKID, s.ColumnA, s.ColumnB, s.ColumnC)
  当源中未匹配时
    然后删除
;
```

注意：AND NOT EXISTS 部分防止更新未更改的记录。使用 INTERSECT 结构允许对可空列进行比较而无需特殊处理。

第27.2节：MySQL：按名称统计用户数

假设我们想知道有多少用户具有相同的名称。让我们按如下方式创建表 users：

```
创建表 users(
  id int 主键 自动递增,
  name varchar(8),
  count int,
  unique key name(name)
);
```

现在，我们刚刚发现了一个名叫乔的新用户，想要将他纳入考虑。为此，我们需要确定是否存在他的名字的现有行，如果存在，则更新该行使count加一；反之，如果不存在，则应创建该行。

MySQL 使用以下语法：insert ... on duplicate key update ...。在这种情况下：

```
insert into users(name, count)
  values ('Joe', 1)
  on duplicate key update count=count+1;
```

第27.3节：PostgreSQL：按名称统计用户

假设我们想知道有多少用户具有相同的名称。让我们按如下方式创建表 users：

Chapter 27: MERGE

MERGE (often also called UPSERT for "update or insert") allows to insert new rows or, if a row already exists, to update the existing row. The point is to perform the whole set of operations atomically (to guarantee that the data remain consistent), and to prevent communication overhead for multiple SQL statements in a client/server system.

Section 27.1: MERGE to make Target match Source

```
MERGE INTO targetTable t
  USING sourceTable s
    ON t.PKID = s.PKID
  WHEN MATCHED AND NOT EXISTS (
    SELECT s.ColumnA, s.ColumnB, s.ColumnC
    INTERSECT
    SELECT t.ColumnA, t.ColumnB, s.ColumnC
  )
    THEN UPDATE SET
      t.ColumnA = s.ColumnA
      ,t.ColumnB = s.ColumnB
      ,t.ColumnC = s.ColumnC
  WHEN NOT MATCHED BY TARGET
    THEN INSERT (PKID, ColumnA, ColumnB, ColumnC)
    VALUES (s.PKID, s.ColumnA, s.ColumnB, s.ColumnC)
  WHEN NOT MATCHED BY SOURCE
    THEN DELETE
;
```

Note: The AND NOT EXISTS portion prevents updating records that haven't changed. Using the INTERSECT construct allows nullable columns to be compared without special handling.

Section 27.2: MySQL: counting users by name

Suppose we want to know how many users have the same name. Let us create table users as follows:

```
create table users(
  id int primary key auto_increment,
  name varchar(8),
  count int,
  unique key name(name)
);
```

Now, we just discovered a new user named Joe and would like to take him into account. To achieve that, we need to determine whether there is an existing row with his name, and if so, update it to increment count; on the other hand, if there is no existing row, we should create it.

MySQL uses the following syntax：insert ... on duplicate key update In this case:

```
insert into users(name, count)
  values ('Joe', 1)
  on duplicate key update count=count+1;
```

Section 27.3: PostgreSQL: counting users by name

Suppose we want to know how many users have the same name. Let us create table users as follows:

```
create table users(  
    id serial,  
    name varchar(8) unique,  
    count int  
);
```

现在，我们刚刚发现了一个名叫乔的新用户，想要将他纳入考虑。为此，我们需要确定是否存在他的名字的现有行，如果存在，则更新该行使count加一；反之，如果不存在，则应创建该行。

PostgreSQL 使用以下语法：insert ... on conflict ... do update ...。在这种情况下：

```
insert into users(name, count)  
values('Joe', 1)  
on conflict (name) do update set count = users.count + 1;
```

```
create table users(  
    id serial,  
    name varchar(8) unique,  
    count int  
);
```

Now, we just discovered a new user named Joe and would like to take him into account. To achieve that, we need to determine whether there is an existing row with his name, and if so, update it to increment count; on the other hand, if there is no existing row, we should create it.

PostgreSQL uses the following syntax : [insert ... on conflict ... do update ...](#). In this case:

```
insert into users(name, count)  
values('Joe', 1)  
on conflict (name) do update set count = users.count + 1;
```

第28章：cross apply, outer apply

第28.1节：CROSS APPLY 和 OUTER APPLY 基础

当右侧表达式为表值函数时，将使用 Apply。

创建一个部门表以保存部门信息。然后创建一个员工表以保存员工信息。请注意，每个员工都属于一个部门，因此员工表与部门表具有参照完整性。

第一个查询从部门表中选择数据，并使用 CROSS APPLY 对部门表的每条记录评估员工表。第二个查询则简单地将部门表与员工表连接，产生所有匹配的记录。

```
SELECT *
FROM Department D
CROSS APPLY (
    SELECT *
    FROM Employee E
    WHERE E.DepartmentID = D.DepartmentID
) A
GO
SELECT *
FROM Department D
INNER JOIN 员工 E
    ON D.部门ID = E.部门ID
```

如果你看他们产生的结果，完全是相同的结果集；它与 JOIN 有何不同，以及它如何帮助编写更高效的查询。

脚本#2中的第一个查询从部门表中选择数据，并使用 OUTER APPLY 对员工表进行评估，针对部门表的每条记录。对于员工表中没有匹配的那些行，如第5行和第6行所示，这些行包含 NULL 值。第二个查询简单地使用部门表和员工表之间的 LEFT OUTER JOIN。正如预期的那样，查询返回部门表中的所有行；即使对于员工表中没有匹配的那些行也是如此。

```
SELECT *
FROM Department D
OUTER APPLY (
    SELECT *
    FROM Employee E
    WHERE E.DepartmentID = D.DepartmentID
) A
GO
SELECT *
FROM Department D
LEFT OUTER JOIN 员工 E
    ON D.部门ID = E.部门ID
GO
```

尽管上述两个查询返回相同的信息，但执行计划会有所不同。但在成本方面差别不大。

现在到了看看 APPLY 操作符真正需要用到的地方。在脚本#3中，我创建了一个表值函数，该函数接受部门ID作为参数，并返回属于该部门的所有员工。接下来的查询从部门表中选择数据，并使用 CROSS APPLY 与我们创建的函数连接。

Chapter 28: cross apply, outer apply

Section 28.1: CROSS APPLY and OUTER APPLY basics

Apply will be used when when table valued function in the right expression.

create a Department table to hold information about departments. Then create an Employee table which hold information about the employees. Please note, each employee belongs to a department, hence the Employee table has referential integrity with the Department table.

First query selects data from Department table and uses CROSS APPLY to evaluate the Employee table for each record of the Department table. Second query simply joins the Department table with the Employee table and all the matching records are produced.

```
SELECT *
FROM Department D
CROSS APPLY (
    SELECT *
    FROM Employee E
    WHERE E.DepartmentID = D.DepartmentID
) A
GO
SELECT *
FROM Department D
INNER JOIN Employee E
    ON D.DepartmentID = E.DepartmentID
```

If you look at the results they produced, it is the exact same result-set; How does it differ from a JOIN and how does it help in writing more efficient queries.

The first query in Script #2 selects data from Department table and uses OUTER APPLY to evaluate the Employee table for each record of the Department table. For those rows for which there is not a match in Employee table, those rows contains NULL values as you can see in case of row 5 and 6. The second query simply uses a LEFT OUTER JOIN between the Department table and the Employee table. As expected the query returns all rows from Department table; even for those rows for which there is no match in the Employee table.

```
SELECT *
FROM Department D
OUTER APPLY (
    SELECT *
    FROM Employee E
    WHERE E.DepartmentID = D.DepartmentID
) A
GO
SELECT *
FROM Department D
LEFT OUTER JOIN Employee E
    ON D.DepartmentID = E.DepartmentID
GO
```

Even though the above two queries return the same information, the execution plan will be bit different. But cost wise there will be not much difference.

Now comes the time to see where the APPLY operator is really required. In Script #3, I am creating a table-valued function which accepts DepartmentID as its parameter and returns all the employees who belong to this department. The next query selects data from Department table and uses CROSS APPLY to join with the function

它为外部表表达式（在我们的例子中是部门表）的每一行传递部门ID，并对每一行评估该函数，类似于相关子查询。下一个查询使用 OUTER APPLY 代替 CROSS APPLY，因此与只返回相关数据的 CROSS APPLY 不同，OUTER APPLY 也返回非相关数据，并在缺失的列中放置 NULL。

```
创建函数 dbo.fn_GetAllEmployeeOfADepartment (@DeptID 作为 int)
返回表
AS
    返回
    (
        SELECT
            *
        FROM Employee E
        WHERE E.DepartmentID = @DeptID
    )
GO
SELECT
    *
FROM Department D
CROSS APPLY dbo.fn_GetAllEmployeeOfADepartment(D.DepartmentID)
GO
SELECT
    *
FROM Department D
OUTER APPLY dbo.fn_GetAllEmployeeOfADepartment(D.DepartmentID)
GO
```

那么现在如果你想知道，是否可以用简单的连接来替代上述查询？答案是否定的，如果你将上述查询中的 CROSS/OUTER APPLY 替换为 INNER JOIN/LEFT OUTER JOIN，指定 ON 子句（例如1=1）并执行查询，你将会得到“无法绑定多部分标识符 'D.DepartmentID'”的错误。这是因为使用 JOIN 时，外部查询的执行上下文与函数（或派生表）的执行上下文不同，无法将外部查询中的值/变量绑定为函数的参数。

因此，对于此类查询，需要使用 APPLY 运算符。

we created. It passes the DepartmentID for each row from the outer table expression (in our case Department table) and evaluates the function for each row similar to a correlated subquery. The next query uses the OUTER APPLY in place of CROSS APPLY and hence unlike CROSS APPLY which returned only correlated data, the OUTER APPLY returns non-correlated data as well, placing NULLs into the missing columns.

```
CREATE FUNCTION dbo.fn_GetAllEmployeeOfADepartment (@DeptID AS int)
RETURNS TABLE
AS
    RETURN
    (
        SELECT
            *
        FROM Employee E
        WHERE E.DepartmentID = @DeptID
    )
GO
SELECT
    *
FROM Department D
CROSS APPLY dbo.fn_GetAllEmployeeOfADepartment(D.DepartmentID)
GO
SELECT
    *
FROM Department D
OUTER APPLY dbo.fn_GetAllEmployeeOfADepartment(D.DepartmentID)
GO
```

So now if you are wondering, can we use a simple join in place of the above queries? Then the answer is NO, if you replace CROSS/OUTER APPLY in the above queries with INNER JOIN/LEFT OUTER JOIN, specify ON clause (something as 1=1) and run the query, you will get "The multi-part identifier "D.DepartmentID" could not be bound." error. This is because with JOINS the execution context of outer query is different from the execution context of the function (or a derived table), and you can not bind a value/variable from the outer query to the function as a parameter. Hence the APPLY operator is required for such queries.

第29章：删除

DELETE 语句用于从表中删除记录。

第29.1节：删除所有行

省略WHERE子句将删除表中的所有行。

```
DELETE FROM Employees
```

有关TRUNCATE的详细信息，请参阅TRUNCATE文档，了解TRUNCATE性能更优的原因，因为它忽略触发器和索引，并且只记录日志以删除数据。

第29.2节：使用WHERE删除特定行

这将删除所有符合WHERE条件的行。

```
DELETE FROM Employees
WHERE FName = 'John'
```

第29.3节：TRUNCATE子句

使用此命令将表重置到创建时的状态。它删除所有行并重置诸如自动递增等值。同时，它也不会为每一行删除操作单独记录日志。

```
TRUNCATE TABLE Employees
```

第29.4节：基于与其他表的比较删除特定行

如果表中的数据与其他表中的某些数据匹配（或不匹配），则可以删除该数据。

假设我们想在数据加载到目标（Target）后，从源（Source）中删除数据。

```
从 Source 删除
条件是存在 ( 选择1 -- SELECT 中的具体值无关紧要
              来自 Target
              其中 Source.ID = Target.ID )
```

大多数常见的关系型数据库管理系统（如 MySQL、Oracle、PostgreSQL、Teradata）允许在删除操作中进行表连接，从而以简洁的语法实现更复杂的比较。

在原始场景中增加复杂度，假设 Aggregate 是每天从 Target 构建的，且不包含相同的 ID，但包含相同的日期。我们还假设只有在当天的聚合数据填充完成后，才想从 Source 中删除数据。

在 MySQL、Oracle 和 Teradata 中，可以使用以下方式实现：

```
从 Source 删除
条件是 Source.ID = TargetSchema.Target.ID
      且 TargetSchema.Target.Date = AggregateSchema.Aggregate.Date
```

在 PostgreSQL 中使用：

Chapter 29: DELETE

The DELETE statement is used to delete records from a table.

Section 29.1: DELETE all rows

Omitting a WHERE clause will delete all rows from a table.

```
DELETE FROM Employees
```

See TRUNCATE documentation for details on how TRUNCATE performance can be better because it ignores triggers and indexes and logs to just delete the data.

Section 29.2: DELETE certain rows with WHERE

This will delete all rows that match the WHERE criteria.

```
DELETE FROM Employees
WHERE FName = 'John'
```

Section 29.3: TRUNCATE clause

Use this to reset the table to the condition at which it was created. This deletes all rows and resets values such as auto-increment. It also doesn't log each individual row deletion.

```
TRUNCATE TABLE Employees
```

Section 29.4: DELETE certain rows based upon comparisons with other tables

It is possible to DELETE data from a table if it matches (or mismatches) certain data in other tables.

Let's assume we want to DELETE data from Source once its loaded into Target.

```
DELETE FROM Source
WHERE EXISTS ( SELECT 1 -- specific value in SELECT doesn't matter
               FROM Target
               Where Source.ID = Target.ID )
```

Most common RDBMS implementations (e.g. MySQL, Oracle, PostgreSQL, Teradata) allow tables to be joined during DELETE allowing more complex comparison in a compact syntax.

Adding complexity to original scenario, let's assume Aggregate is built from Target once a day and does not contain the same ID but contains the same date. Let us also assume that we want to delete data from Source *only* after the aggregate is populated for the day.

On MySQL, Oracle and Teradata this can be done using:

```
DELETE FROM Source
WHERE Source.ID = TargetSchema.Target.ID
      AND TargetSchema.Target.Date = AggregateSchema.Aggregate.Date
```

In PostgreSQL use:

```
从 Source 删除
使用 TargetSchema.Target, AggregateSchema.Aggregate
WHERE Source.ID = TargetSchema.Target.ID
      AND TargetSchema.Target.DataDate = AggregateSchema.Aggregate.AggDate
```

这本质上导致了 Source、Target 和 Aggregate 之间的内连接。当 Target 中存在相同 ID 且这些 ID 对应的日期也存在于 Aggregate 中时，删除操作会在 Source 上执行。

相同的查询也可以写成（在 MySQL、Oracle、Teradata 上）：

```
DELETE Source
FROM Source, TargetSchema.Target, AggregateSchema.Aggregate
WHERE Source.ID = TargetSchema.Target.ID
      AND TargetSchema.Target.DataDate = AggregateSchema.Aggregate.AggDate
```

在某些关系数据库管理系统（如 Oracle、MySQL）中，Delete 语句中可以显式指定连接，但并非所有平台都支持（例如 Teradata 不支持）。

比较可以设计为检查不匹配的情况，而不是匹配的情况，适用于所有语法风格（参见下文 NOT EXISTS）

```
从 Source 删除
WHERE NOT EXISTS ( SELECT 1 -- SELECT 中的具体值无关紧要
                  FROM Target
                  其中 Source.ID = Target.ID )
```

```
DELETE FROM Source
USING TargetSchema.Target, AggregateSchema.Aggregate
WHERE Source.ID = TargetSchema.Target.ID
      AND TargetSchema.Target.DataDate = AggregateSchema.Aggregate.AggDate
```

This essentially results in INNER JOINS between Source, Target and Aggregate. The deletion is performed on Source when the same IDs exist in Target AND date present in Target for those IDs also exists in Aggregate.

Same query may also be written (on MySQL, Oracle, Teradata) as:

```
DELETE Source
FROM Source, TargetSchema.Target, AggregateSchema.Aggregate
WHERE Source.ID = TargetSchema.Target.ID
      AND TargetSchema.Target.DataDate = AggregateSchema.Aggregate.AggDate
```

Explicit joins may be mentioned in Delete statements on some RDBMS implementations (e.g. Oracle, MySQL) but not supported on all platforms (e.g. Teradata does not support them)

Comparisons can be designed to check mismatch scenarios instead of matching ones with all syntax styles (observe NOT EXISTS below)

```
DELETE FROM Source
WHERE NOT EXISTS ( SELECT 1 -- specific value in SELECT doesn't matter
                  FROM Target
                  Where Source.ID = Target.ID )
```


第30章：TRUNCATE

TRUNCATE 语句删除表中的所有数据。这类似于无筛选条件的 DELETE，但根据数据库软件的不同，具有某些限制和优化。

第30.1节：从员工表中删除所有行

```
TRUNCATE TABLE Employee;
```

使用 truncate table 通常比使用 DELETE TABLE 更好，因为它忽略所有索引和触发器，直接删除所有内容。

DELETE 表是基于行的操作，这意味着每一行都会被删除。TRUNCATE 表是基于数据页的操作，整个数据页会被重新分配。如果你的表有一百万行，truncate 表的速度会远快于使用 delete 表语句。

虽然我们可以使用 DELETE 删除特定的行，但不能 TRUNCATE 特定的行，只能一次性 TRUNCATE 所有记录。删除所有行然后插入新记录时，自动递增的主键值会从之前插入的值继续增加，而使用 TRUNCATE 时，自动递增的主键值也会被重置，从1开始。

注意，截断表时，不能存在外键，否则会报错。

Chapter 30: TRUNCATE

The TRUNCATE statement deletes all data from a table. This is similar to DELETE with no filter, but, depending on the database software, has certain restrictions and optimizations.

Section 30.1: Removing all rows from the Employee table

```
TRUNCATE TABLE Employee;
```

Using truncate table is often better then using DELETE TABLE as it ignores all the indexes and triggers and just removes everything.

Delete table is a row based operation this means that each row is deleted. Truncate table is a data page operation the entire data page is reallocated. If you have a table with a million rows it will be much faster to truncate the table than it would be to use a delete table statement.

Though we can delete specific Rows with DELETE, we cannot TRUNCATE specific rows, we can only TRUNCATE all the records at once. Deleting All rows and then inserting a new record will continue to add the Auto incremented Primary key value from the previously inserted value, where as in Truncate, the Auto Incremental primary key value will also get reset and starts from 1.

Note that when truncating table, **no foreign keys must be present**, otherwise you will get an error.

第31章：DROP 表

第31.1节：删除前检查是否存在

MySQL 版本 ≥ 3.19

如果存在则删除表 MyTable;

PostgreSQL 版本 ≥ 8.x

如果存在则删除表 MyTable;

SQL Server 版本 ≥ 2005

如果存在(选择 * 从 Information_Schema.Tables
其中 Table_Schema = 'dbo'
并且 Table_Name = 'MyTable')
删除表 dbo.MyTable

SQLite 版本 ≥ 3.0

如果存在则删除表 MyTable;

第31.2节：简单删除

删除表 MyTable;

Chapter 31: DROP Table

Section 31.1: Check for existence before dropping

MySQL Version ≥ 3.19

DROP TABLE IF EXISTS MyTable;

PostgreSQL Version ≥ 8.x

DROP TABLE IF EXISTS MyTable;

SQL Server Version ≥ 2005

If Exists(Select * From Information_Schema.Tables
Where Table_Schema = 'dbo'
And Table_Name = 'MyTable')
Drop Table dbo.MyTable

SQLite Version ≥ 3.0

DROP TABLE IF EXISTS MyTable;

Section 31.2: Simple drop

Drop Table MyTable;

第32章：删除或删除数据库

第32.1节：删除数据库

删除数据库是一条简单的一行语句。删除数据库将会删除该数据库，因此如果需要，务必确保已备份数据库。

以下是删除员工数据库的命令

```
DROP DATABASE [dbo].[Employees]
```

Chapter 32: DROP or DELETE Database

Section 32.1: DROP Database

Dropping the database is a simple one-liner statement. Drop database will delete the database, hence always ensure to have a backup of the database if required.

Below is the command to drop Employees Database

```
DROP DATABASE [dbo].[Employees]
```

第33章：级联删除

第33.1节：ON DELETE CASCADE

假设你有一个管理房间的应用程序。
进一步假设你的应用程序是基于每个客户（租户）来操作的。
你有多个客户。
因此你的数据库将包含一个客户表和一个房间表。

现在，每个客户都有N个房间。

这意味着你的房间表上应该有一个外键，引用客户表。

```
ALTER TABLE dbo.T_Room WITH CHECK ADD CONSTRAINT FK_T_Room_T_Client FOREIGN KEY(RM_CLI_ID)
REFERENCES dbo.T_Client (CLI_ID)
GO
```

假设客户转用其他软件，你必须在你的软件中删除他的数据。但如果你这样做

```
DELETE FROM T_Client WHERE CLI_ID = x
```

那么你会遇到外键冲突，因为当客户还有房间时，你不能删除该客户。

现在你得在应用程序中编写代码，先删除客户的房间再删除客户。进一步假设将来数据库中会增加更多外键依赖，因为应用功能扩展。太糟糕了。每次数据库修改，你都得在N个地方调整应用代码。可能还得调整其他应用的代码（例如与其他系统的接口）。

有比在代码中处理更好的解决方案。
你可以直接在外键中添加ON DELETE CASCADE。

```
ALTER TABLE dbo.T_Room -- WITH CHECK -- SQL-Server 可以指定 WITH CHECK/WITH NOCHECK
ADD CONSTRAINT FK_T_Room_T_Client FOREIGN KEY(RM_CLI_ID)
REFERENCES dbo.T_Client (CLI_ID)
ON DELETE CASCADE
```

现在你可以这样说

```
DELETE FROM T_Client WHERE CLI_ID = x
```

当客户端被删除时，房间会自动被删除。
问题已解决——无需更改应用程序代码。

有一点需要注意：在微软SQL Server中，如果你有一个自引用的表，这种方法是行不通的。所以如果你尝试在递归树结构上定义删除级联，如下所示：

```
IF NOT EXISTS (SELECT * FROM sys.foreign_keys WHERE object_id =
OBJECT_ID(N'[dbo].[FK_T_FMS_Navigation_T_FMS_Navigation]') AND parent_object_id =
OBJECT_ID(N'[dbo].[T_FMS_Navigation]'))
ALTER TABLE [dbo].[T_FMS_Navigation] WITH CHECK ADD CONSTRAINT
[FK_T_FMS_Navigation_T_FMS_Navigation] FOREIGN KEY([NA_NA_UID])
REFERENCES [dbo].[T_FMS_Navigation] ([NA_UID])
ON DELETE CASCADE
```

Chapter 33: Cascading Delete

Section 33.1: ON DELETE CASCADE

Assume you have a application that administers rooms.
Assume further that your application operates on a per client basis (tenant).
You have several clients.
So your database will contain one table for clients, and one for rooms.

Now, every client has N rooms.

This should mean that you have a foreign key on your room table, referencing the client table.

```
ALTER TABLE dbo.T_Room WITH CHECK ADD CONSTRAINT FK_T_Room_T_Client FOREIGN KEY(RM_CLI_ID)
REFERENCES dbo.T_Client (CLI_ID)
GO
```

Assuming a client moves on to some other software, you'll have to delete his data in your software. But if you do

```
DELETE FROM T_Client WHERE CLI_ID = x
```

Then you'll get a foreign key violation, because you can't delete the client when he still has rooms.

Now you'd have write code in your application that deletes the client's rooms before it deletes the client. Assume further that in the future, many more foreign key dependencies will be added in your database, because your application's functionality expands. Horrible. For every modification in your database, you'll have to adapt your application's code in N places. Possibly you'll have to adapt code in other applications as well (e.g. interfaces to other systems).

There is a better solution than doing it in your code.
You can just add ON DELETE CASCADE to your foreign key.

```
ALTER TABLE dbo.T_Room -- WITH CHECK -- SQL-Server can specify WITH CHECK/WITH NOCHECK
ADD CONSTRAINT FK_T_Room_T_Client FOREIGN KEY(RM_CLI_ID)
REFERENCES dbo.T_Client (CLI_ID)
ON DELETE CASCADE
```

Now you can say

```
DELETE FROM T_Client WHERE CLI_ID = x
```

and the rooms are automagically deleted when the client is deleted.
Problem solved - with no application code changes.

One word of caution: In Microsoft SQL-Server, this won't work if you have a table that references itself. So if you try to define a delete cascade on a recursive tree structure, like this:

```
IF NOT EXISTS (SELECT * FROM sys.foreign_keys WHERE object_id =
OBJECT_ID(N'[dbo].[FK_T_FMS_Navigation_T_FMS_Navigation]') AND parent_object_id =
OBJECT_ID(N'[dbo].[T_FMS_Navigation]'))
ALTER TABLE [dbo].[T_FMS_Navigation] WITH CHECK ADD CONSTRAINT
[FK_T_FMS_Navigation_T_FMS_Navigation] FOREIGN KEY([NA_NA_UID])
REFERENCES [dbo].[T_FMS_Navigation] ([NA_UID])
ON DELETE CASCADE
```

```
GO

IF EXISTS (SELECT * FROM sys.foreign_keys WHERE object_id =
OBJECT_ID(N'[dbo].[FK_T_FMS_Navigation_T_FMS_Navigation]') AND parent_object_id =
OBJECT_ID(N'[dbo].[T_FMS_Navigation]'))
ALTER TABLE [dbo].[T_FMS_Navigation] CHECK CONSTRAINT [FK_T_FMS_Navigation_T_FMS_Navigation]
GO
```

这行不通，因为微软SQL Server不允许你在递归树结构上设置带有ON DELETE CASCADE的外键。原因之一是树可能是循环的，这可能导致死锁。

而PostgreSQL则可以做到这一点；
前提是树是非循环的。
如果树是有环的，你将会遇到运行时错误。
在那种情况下，你只能自己实现删除功能。

一句警告：
这意味着你不能再简单地删除并重新插入客户表，因为如果你这样做，它会删除“T_Room”中的所有条目.....（不再有非增量更新）

```
GO

IF EXISTS (SELECT * FROM sys.foreign_keys WHERE object_id =
OBJECT_ID(N'[dbo].[FK_T_FMS_Navigation_T_FMS_Navigation]') AND parent_object_id =
OBJECT_ID(N'[dbo].[T_FMS_Navigation]'))
ALTER TABLE [dbo].[T_FMS_Navigation] CHECK CONSTRAINT [FK_T_FMS_Navigation_T_FMS_Navigation]
GO
```

it won't work, because Microsoft-SQL-server doesn't allow you to set a foreign key with **ON DELETE CASCADE** on a recursive tree structure. One reason for this is, that the tree is possibly cyclic, and that would possibly lead to a deadlock.

PostgreSQL on the other hand can do this;
the requirement is that the tree is non-cyclic.
If the tree is cyclic, you'll get a runtime error.
In that case, you'll just have to implement the delete function yourselfs.

A word of caution:
This means you can't simply delete and re-insert the client table anymore, because if you do this, it will delete all entries in "T_Room"... (no non-delta updates anymore)

第34章：授权和撤销权限

第34.1节：授权/撤销权限

授予 `SELECT`, `UPDATE`
权限于 `Employees`
给 `User1`, `User2`;

授予User1和用户2对表Employees执行SELECT和UPDATE操作的权限。

撤销 `SELECT`, `UPDATE`
权限于 `Employees`
来自 `User1`, `User2`;

撤销User1和用户2对表Employees执行SELECT和UPDATE操作的权限。

Chapter 34: GRANT and REVOKE

Section 34.1: Grant/revoke privileges

`GRANT SELECT, UPDATE`
`ON Employees`
`TO User1, User2`;

Grant User1 and User2 permission to perform `SELECT` and `UPDATE` operations on table `Employees`.

`REVOKE SELECT, UPDATE`
`ON Employees`
`FROM User1, User2`;

Revoke from User1 and User2 the permission to perform `SELECT` and `UPDATE` operations on table `Employees`.

第35章：XML

第35.1节：从XML数据类型查询

```
声明@XmlInXML='<TableData>
<aaa Main="First">
<row name="a" value="1" />
  <row name="b" value="2" />
  <row name="c" value="3" />
</aaa>
<aaa Main="Second">
  <row name="a" value="3" />
  <row name="b" value="4" />
  <row name="c" value="5" />
</aaa>
<aaa Main="Third">
  <row name="a" value="10" />
  <row name="b" value="20" />
  <row name="c" value="30" />
</aaa>
</TableData>'

SELECT t.col.value('../@Main', 'varchar(10)') [Header],
t.col.value('@name', 'VARCHAR(25)') [name],
t.col.value('@value', 'VARCHAR(25)') [Value]
FROM   @xmlIn.nodes('//TableData/aaa/row') AS t (col)
```

结果

标题	名称	数值
第一组	a	1
第一组	b	2
第一组	c	3
第二组	a	3
第二组	b	4
第二组	c	5
第三组	a	10
第三组	b	20
第三组	c	30

Chapter 35: XML

Section 35.1: Query from XML Data Type

```
DECLARE @xmlIn XML = '<TableData>
<aaa Main="First">
  <row name="a" value="1" />
  <row name="b" value="2" />
  <row name="c" value="3" />
</aaa>
<aaa Main="Second">
  <row name="a" value="3" />
  <row name="b" value="4" />
  <row name="c" value="5" />
</aaa>
<aaa Main="Third">
  <row name="a" value="10" />
  <row name="b" value="20" />
  <row name="c" value="30" />
</aaa>
</TableData>'

SELECT t.col.value('../@Main', 'varchar(10)') [Header],
t.col.value('@name', 'VARCHAR(25)') [name],
t.col.value('@value', 'VARCHAR(25)') [Value]
FROM   @xmlIn.nodes('//TableData/aaa/row') AS t (col)
```

Results

Header	name	Value
First	a	1
First	b	2
First	c	3
Second	a	3
Second	b	4
Second	c	5
Third	a	10
Third	b	20
Third	c	30

第36章：主键

第36.1节：创建主键

```
CREATE TABLE 员工 (  
    Id int NOT NULL,  
    PRIMARY KEY (Id),  
    ...  
);
```

这将创建一个名为员工的表，‘Id’作为其主键。主键可用于唯一标识表中的行。每个表只允许有一个主键。

键也可以由一个或多个字段组成，称为复合键，语法如下：

```
CREATE TABLE 员工 (  
    e1_id INT,  
    e2_id INT,  
    PRIMARY KEY (e1_id, e2_id)  
);
```

第36.2节：使用自动递增

许多数据库允许在添加新主键时自动递增主键值。这确保每个键都是不同的。

MySQL

```
CREATE TABLE Employees (  
    Id int NOT NULL AUTO_INCREMENT,  
    PRIMARY KEY (Id)  
);
```

PostgreSQL

```
CREATE TABLE Employees (  
    Id SERIAL PRIMARY KEY  
);
```

SQL Server

```
CREATE TABLE Employees (  
    Id int NOT NULL IDENTITY,  
    PRIMARY KEY (Id)  
);
```

SQLite

```
CREATE TABLE Employees (  
    Id INTEGER PRIMARY KEY  
);
```

Chapter 36: Primary Keys

Section 36.1: Creating a Primary Key

```
CREATE TABLE Employees (  
    Id int NOT NULL,  
    PRIMARY KEY (Id),  
    ...  
);
```

This will create the Employees table with 'Id' as its primary key. The primary key can be used to uniquely identify the rows of a table. Only one primary key is allowed per table.

A key can also be composed by one or more fields, so called composite key, with the following syntax:

```
CREATE TABLE EMPLOYEE (  
    e1_id INT,  
    e2_id INT,  
    PRIMARY KEY (e1_id, e2_id)  
);
```

Section 36.2: Using Auto Increment

Many databases allow to make the primary key value automatically increment when a new key is added. This ensures that every key is different.

MySQL

```
CREATE TABLE Employees (  
    Id int NOT NULL AUTO_INCREMENT,  
    PRIMARY KEY (Id)  
);
```

PostgreSQL

```
CREATE TABLE Employees (  
    Id SERIAL PRIMARY KEY  
);
```

SQL Server

```
CREATE TABLE Employees (  
    Id int NOT NULL IDENTITY,  
    PRIMARY KEY (Id)  
);
```

SQLite

```
CREATE TABLE Employees (  
    Id INTEGER PRIMARY KEY  
);
```

第37章：索引

索引是一种数据结构，包含指向表内容的指针，这些内容按特定顺序排列，以帮助数据库优化查询。它们类似于书籍的索引，其中页面（表的行）通过页码进行索引。

存在多种类型的索引，可以在表上创建。当查询的WHERE子句、JOIN子句或ORDER BY子句中使用的列上存在索引时，可以显著提高查询性能。

第37.1节：排序索引

如果你使用的索引是按照检索方式排序的，`SELECT`语句在检索时就不会进行额外的排序。

```
CREATE INDEX ix_scoreboard_score ON scoreboard (score DESC);
```

当你执行查询时

```
SELECT * FROM scoreboard ORDER BY score DESC;
```

数据库系统不会进行额外排序，因为它可以按该顺序进行索引查找。

第37.2节：部分或过滤索引

SQL Server和SQLite允许创建不仅包含部分列，还包含部分行的索引。

考虑一个不断增长的订单数量，其order_state_id等于已完成（2），以及一个稳定数量的订单，其order_state_id等于已开始（1）。

如果你的业务使用如下查询：

```
SELECT id, comment
FROM orders
WHERE order_state_id = 1
AND product_id = @some_value;
```

部分索引允许您限制索引，仅包含未完成的订单：

```
CREATE INDEX Started_Orders
ON orders(product_id)
WHERE order_state_id = 1;
```

该索引将比未过滤的索引更小，从而节省空间并降低更新索引的成本。

第37.3节：创建索引

```
CREATE INDEX ix_cars_employee_id ON Cars (EmployeeId);
```

这将在表Cars中为列EmployeeId创建索引。该索引将提高服务器按EmployeeId中的值排序或选择的查询速度，例如以下查询：

```
SELECT * FROM Cars WHERE EmployeeId = 1
```

Chapter 37: Indexes

Indexes are a data structure that contains pointers to the contents of a table arranged in a specific order, to help the database optimize queries. They are similar to the index of book, where the pages (rows of the table) are indexed by their page number.

Several types of indexes exist, and can be created on a table. When an index exists on the columns used in a query's WHERE clause, JOIN clause, or ORDER BY clause, it can substantially improve query performance.

Section 37.1: Sorted Index

If you use an index that is sorted the way you would retrieve it, the `SELECT` statement would not do additional sorting when in retrieval.

```
CREATE INDEX ix_scoreboard_score ON scoreboard (score DESC);
```

When you execute the query

```
SELECT * FROM scoreboard ORDER BY score DESC;
```

The database system would not do additional sorting, since it can do an index-lookup in that order.

Section 37.2: Partial or Filtered Index

SQL Server and SQLite allow to create indexes that contain not only a subset of columns, but also a subset of rows.

Consider a constant growing amount of orders with order_state_id equal to finished (2), and a stable amount of orders with order_state_id equal to started (1).

If your business make use of queries like this:

```
SELECT id, comment
FROM orders
WHERE order_state_id = 1
AND product_id = @some_value;
```

Partial indexing allows you to limit the index, including only the unfinished orders:

```
CREATE INDEX Started_Orders
ON orders(product_id)
WHERE order_state_id = 1;
```

This index will be smaller than an unfiltered index, which saves space and reduces the cost of updating the index.

Section 37.3: Creating an Index

```
CREATE INDEX ix_cars_employee_id ON Cars (EmployeeId);
```

This will create an index for the column *EmployeeId* in the table *Cars*. This index will improve the speed of queries asking the server to sort or select by values in *EmployeeId*, such as the following:

```
SELECT * FROM Cars WHERE EmployeeId = 1
```

索引可以包含多个列，如下所示；

```
CREATE INDEX ix_cars_e_c_o_ids ON Cars (EmployeeId, CarId, OwnerId);
```

在这种情况下，如果条件集合的顺序相同，该索引对于按所有包含的列排序或选择的查询将非常有用。这意味着在检索数据时，可以使用索引找到要检索的行，而不是遍历整个表。

例如，以下情况将使用第二个索引；

```
SELECT * FROM Cars WHERE EmployeeId = 1 Order by CarId DESC
```

然而，如果排序不同，索引就没有同样的优势，如下所示；

```
SELECT * FROM Cars WHERE OwnerId = 17 Order by CarId DESC
```

该索引不太有用，因为数据库必须检索整个索引，涵盖所有 EmployeeId 和 CarID 的值，才能找到 OwnerId = 17 的条目。

（索引仍可能被使用；查询优化器可能发现先检索索引并基于 OwnerId 过滤，然后只检索所需行，比检索整个表更快，尤其是在表很大的情况下。）

第37.4节：删除索引，或禁用并重建索引

```
DROP INDEX ix_cars_employee_id ON Cars;
```

我们可以使用命令 DROP 来删除索引。在此示例中，我们将删除表 Cars 上名为 ix_cars_employee_id 的索引。

这将完全删除索引，如果索引是聚簇索引，则会移除任何聚簇。无法在不重新创建索引的情况下重建索引，重新创建索引可能会很慢且计算开销大。作为替代方案，可以禁用索引：

```
ALTER INDEX ix_cars_employee_id ON Cars DISABLE;
```

这允许表保留结构，以及关于索引的元数据。

关键是，这保留了索引统计信息，因此可以轻松评估更改。如果有必要，索引随后可以重建，而不是完全重新创建；

```
ALTER INDEX ix_cars_employee_id ON Cars REBUILD;
```

第37.5节：聚集索引、唯一索引和排序索引

索引可以具有多种特性，这些特性可以在创建时设置，也可以通过修改现有索引来设置。

```
CREATE CLUSTERED INDEX ix_clust_employee_id ON Employees(EmployeeId, Email);
```

上述SQL语句在Employees表上创建了一个新的聚集索引。聚集索引是决定表实际结构的索引；表本身会按照索引的结构进行排序。这意味着一个表最多只能有一个聚集索引。如果表上已经存在聚集索引，上述语句将

The index can contain more than 1 column, as in the following;

```
CREATE INDEX ix_cars_e_c_o_ids ON Cars (EmployeeId, CarId, OwnerId);
```

In this case, the index would be useful for queries asking to sort or select by all included columns, if the set of conditions is ordered in the same way. That means that when retrieving the data, it can find the rows to retrieve using the index, instead of looking through the full table.

For example, the following case would utilize the second index;

```
SELECT * FROM Cars WHERE EmployeeId = 1 Order by CarId DESC
```

If the order differs, however, the index does not have the same advantages, as in the following;

```
SELECT * FROM Cars WHERE OwnerId = 17 Order by CarId DESC
```

The index is not as helpful because the database must retrieve the entire index, across all values of EmployeeId and CarID, in order to find which items have OwnerId = 17.

(The index may still be used; it may be the case that the query optimizer finds that retrieving the index and filtering on the OwnerId, then retrieving only the needed rows is faster than retrieving the full table, especially if the table is large.)

Section 37.4: Dropping an Index, or Disabling and Rebuilding it

```
DROP INDEX ix_cars_employee_id ON Cars;
```

We can use command DROP to delete our index. In this example we will DROP the index called *ix_cars_employee_id* on the table *Cars*.

This deletes the index entirely, and if the index is clustered, will remove any clustering. It cannot be rebuilt without recreating the index, which can be slow and computationally expensive. As an alternative, the index can be disabled:

```
ALTER INDEX ix_cars_employee_id ON Cars DISABLE;
```

This allows the table to retain the structure, along with the metadata about the index.

Critically, this retains the index statistics, so that it is possible to easily evaluate the change. If warranted, the index can then later be rebuilt, instead of being recreated completely;

```
ALTER INDEX ix_cars_employee_id ON Cars REBUILD;
```

Section 37.5: Clustered, Unique, and Sorted Indexes

Indexes can have several characteristics that can be set either at creation, or by altering existing indexes.

```
CREATE CLUSTERED INDEX ix_clust_employee_id ON Employees(EmployeeId, Email);
```

The above SQL statement creates a new clustered index on Employees. Clustered indexes are indexes that dictate the actual structure of the table; the table itself is sorted to match the structure of the index. That means there can be at most one clustered index on a table. If a clustered index already exists on the table, the above statement will

失败。（没有聚集索引的表也称为堆。）

```
CREATE UNIQUE INDEX uq_customers_email ON Customers(Email);
```

这将在表Customers中的列Email上创建一个唯一索引。该索引不仅像普通索引一样加快查询速度，还会强制该列中的每个电子邮件地址唯一。如果插入或更新的行中Email值不唯一，默认情况下，插入或更新将失败。

```
CREATE UNIQUE INDEX ix_eid_desc ON Customers(EmployeeID);
```

这将在Customers表上创建一个索引，同时创建一个表约束，要求EmployeeID必须唯一。（如果该列当前不唯一，则此操作将失败——在这种情况下，如果有员工共享同一个ID。）

```
CREATE INDEX ix_eid_desc ON Customers(EmployeeID Desc);
```

这将创建一个按降序排序的索引。默认情况下，索引（至少在MSSQL服务器中）是升序的，但可以更改。

第37.6节：重建索引

随着时间的推移，B树索引可能因更新/删除/插入数据而变得碎片化。在SQL Server术语中，我们可以有内部碎片（索引页半空）和外部碎片（逻辑页顺序与物理顺序不对应）。重建索引与删除并重新创建索引非常相似。

我们可以使用以下命令重建索引

```
ALTER INDEX index_name REBUILD;
```

默认情况下，重建索引是离线操作，会锁定表并阻止对其进行DML操作，但许多关系数据库管理系统允许在线重建。此外，一些数据库供应商提供了索引重建的替代方案，如REORGANIZE (SQLServer) 或 COALESCE/SHRINK SPACE(Oracle)。

第37.7节：带唯一索引的插入

```
UPDATE Customers SET Email = "richard0123@example.com" WHERE id = 1;
```

如果在Customers的Email列上设置了唯一索引，则此操作将失败。但是，可以为这种情况定义备用行为：

```
UPDATE Customers SET Email = "richard0123@example.com" WHERE id = 1 ON DUPLICATE KEY;
```

fail. (Tables with no clustered indexes are also called heaps.)

```
CREATE UNIQUE INDEX uq_customers_email ON Customers(Email);
```

This will create an unique index for the column *Email* in the table *Customers*. This index, along with speeding up queries like a normal index, will also force every email address in that column to be unique. If a row is inserted or updated with a non-unique *Email* value, the insertion or update will, by default, fail.

```
CREATE UNIQUE INDEX ix_eid_desc ON Customers(EmployeeID);
```

This creates an index on Customers which also creates a table constraint that the EmployeeID must be unique. (This will fail if the column is not currently unique - in this case, if there are employees who share an ID.)

```
CREATE INDEX ix_eid_desc ON Customers(EmployeeID Desc);
```

This creates an index that is sorted in descending order. By default, indexes (in MSSQL server, at least) are ascending, but that can be changed.

Section 37.6: Rebuild index

Over the course of time B-Tree indexes may become fragmented because of updating/deleting/inserting data. In SQLServer terminology we can have internal (index page which is half empty) and external (logical page order doesn't correspond physical order). Rebuilding index is very similar to dropping and re-creating it.

We can re-build an index with

```
ALTER INDEX index_name REBUILD;
```

By default rebuilding index is offline operation which locks the table and prevents DML against it , but many RDBMS allow online rebuilding. Also, some DB vendors offer alternatives to index rebuilding such as REORGANIZE (SQLServer) or COALESCE/SHRINK SPACE(Oracle).

Section 37.7: Inserting with a Unique Index

```
UPDATE Customers SET Email = "richard0123@example.com" WHERE id = 1;
```

This will fail if an unique index is set on the *Email* column of *Customers*. However, alternate behavior can be defined for this case:

```
UPDATE Customers SET Email = "richard0123@example.com" WHERE id = 1 ON DUPLICATE KEY;
```


第38章：行号

第38.1节：删除除最后一条记录外的所有记录（1对多表）

```
WITH cte AS (  
    SELECT ProjectID,  
           ROW_NUMBER() OVER (PARTITION BY ProjectID ORDER BY InsertDate DESC) AS rn  
    FROM ProjectNotes  
)  
DELETE FROM cte WHERE rn > 1;
```

第38.2节：无分区的行号

根据指定的顺序包含行号。

```
SELECT  
    ROW_NUMBER() OVER(ORDER BY 名字 ASC) AS 行号,  
    名字,  
    姓氏  
FROM 员工
```

第38.3节：带分区的行号

使用分区条件根据其行号进行分组。

```
SELECT  
    ROW_NUMBER() OVER(PARTITION BY 部门编号 ORDER BY 部门编号 ASC) AS 行号,  
    部门编号, 名字, 姓氏  
FROM 员工
```

Chapter 38: Row number

Section 38.1: Delete All But Last Record (1 to Many Table)

```
WITH cte AS (  
    SELECT ProjectID,  
           ROW_NUMBER() OVER (PARTITION BY ProjectID ORDER BY InsertDate DESC) AS rn  
    FROM ProjectNotes  
)  
DELETE FROM cte WHERE rn > 1;
```

Section 38.2: Row numbers without partitions

Include a row number according to the order specified.

```
SELECT  
    ROW_NUMBER() OVER(ORDER BY Fname ASC) AS RowNumber,  
    Fname,  
    LName  
FROM Employees
```

Section 38.3: Row numbers with partitions

Uses a partition criteria to group the row numbering according to it.

```
SELECT  
    ROW_NUMBER() OVER(PARTITION BY DepartmentId ORDER BY DepartmentId ASC) AS RowNumber,  
    DepartmentId, Fname, LName  
FROM Employees
```


第39章：SQL中的Group By与Distinct

第39.1节：GROUP BY与DISTINCT的区别

GROUP BY用于与聚合函数结合使用。考虑以下表格：

订单号	用户ID	店铺名称	订单金额	订单日期
1	43	店铺A	25	20-03-2016
2	57	店铺B	50	22-03-2016
3	43	店铺A	30	25-03-2016
4	82	店铺C	10	26-03-2016
5	21	店铺A	45	29-03-2016

下面的查询使用GROUP BY来执行聚合计算。

```
SELECT
storeName,
COUNT(*) AS total_nr_orders,
COUNT(DISTINCT userId) AS nr_unique_customers,
AVG(orderValue) AS average_order_value,
MIN(orderDate) AS first_order,
MAX(orderDate) AS lastOrder
FROM
orders
GROUP BY
storeName;
```

并将返回以下信息

storeName	total_nr_orders	nr_unique_customers	average_order_value	first_order	lastOrder
店铺A	3	2	33.3	2016-03-20	2016-03-29
商店 B	1	1	50	2016-03-22	2016-03-22
商店 C	1	1	10	26-03-2016	26-03-2016

而DISTINCT用于列出指定列的唯一不同值组合。

```
SELECT DISTINCT
storeName,
userId
FROM
orders;
```

storeName	userId
店铺A	43
店铺B	57
店铺C	82
店铺A	21

Chapter 39: SQL Group By vs Distinct

Section 39.1: Difference between GROUP BY and DISTINCT

GROUP BY is used in combination with aggregation functions. Consider the following table:

orderId	userId	storeName	orderValue	orderDate
1	43	Store A	25	20-03-2016
2	57	Store B	50	22-03-2016
3	43	Store A	30	25-03-2016
4	82	Store C	10	26-03-2016
5	21	Store A	45	29-03-2016

The query below uses GROUP BY to perform aggregated calculations.

```
SELECT
storeName,
COUNT(*) AS total_nr_orders,
COUNT(DISTINCT userId) AS nr_unique_customers,
AVG(orderValue) AS average_order_value,
MIN(orderDate) AS first_order,
MAX(orderDate) AS lastOrder
FROM
orders
GROUP BY
storeName;
```

and will return the following information

storeName	total_nr_orders	nr_unique_customers	average_order_value	first_order	lastOrder
Store A	3	2	33.3	20-03-2016	29-03-2016
Store B	1	1	50	22-03-2016	22-03-2016
Store C	1	1	10	26-03-2016	26-03-2016

While DISTINCT is used to list a unique combination of distinct values for the specified columns.

```
SELECT DISTINCT
storeName,
userId
FROM
orders;
```

storeName	userId
Store A	43
Store B	57
Store C	82
Store A	21

第40章：在列子集上查找重复项及详细信息

第40.1节：姓名和出生日期相同的学生

```
WITH CTE (StudentId, FName, LName, DOB, RowCnt)
as (
SELECT StudentId, FirstName, LastName, DateOfBirth as DOB, SUM(1) OVER (Partition By FirstName,
LastName, DateOfBirth) as RowCnt
FROM tblStudent
)
SELECT * from CTE where RowCnt > 1
ORDER BY DOB, LName
```

此示例使用公共表表达式（CTE）和窗口函数，展示所有重复行（基于部分列）并排显示。

Chapter 40: Finding Duplicates on a Column Subset with Detail

Section 40.1: Students with same name and date of birth

```
WITH CTE (StudentId, FName, LName, DOB, RowCnt)
as (
SELECT StudentId, FirstName, LastName, DateOfBirth as DOB, SUM(1) OVER (Partition By FirstName,
LastName, DateOfBirth) as RowCnt
FROM tblStudent
)
SELECT * from CTE where RowCnt > 1
ORDER BY DOB, LName
```

This example uses a Common Table Expression and a Window Function to show all duplicate rows (on a subset of columns) side by side.

第41章：字符串函数

字符串函数对字符串值执行操作，并返回数值或字符串值。

使用字符串函数，例如可以合并数据、提取子字符串、比较字符串，或将字符串转换为全大写或全小写字符。

第41.1节：连接

在（标准ANSI/ISO）SQL中，字符串连接的运算符是||。除SQL Server外，所有主流数据库均支持此语法：

```
SELECT 'Hello' || 'World' || '!'; --返回 HelloWorld!
```

许多数据库支持CONCAT函数来连接字符串：

```
SELECT CONCAT('Hello', 'World'); --返回 'HelloWorld'
```

一些数据库支持使用CONCAT连接超过两个字符串（Oracle 不支持）：

```
SELECT CONCAT('Hello', 'World', '!'); --返回 'HelloWorld!'
```

在某些数据库中，非字符串类型必须进行强制转换或转换：

```
SELECT CONCAT('Foo', CAST(42 AS VARCHAR(5)), 'Bar'); --返回 'Foo42Bar'
```

一些数据库（例如 Oracle）执行隐式无损转换。例如，CONCAT用于CLOB和NCLOB会产生NCLOB。数字和varchar 2的CONCAT结果是varchar2，等等：

```
SELECT CONCAT(CONCAT('Foo', 42), 'Bar') FROM dual; --返回 Foo42Bar
```

一些数据库可以使用非标准的+运算符（但在大多数数据库中，+仅对数字有效）：

```
SELECT 'Foo' + CAST(42 AS VARCHAR(5)) + 'Bar';
```

在 SQL Server 2012 之前的版本中，不支持CONCAT，+是连接字符串的唯一方式。

第41.2节：长度

SQL Server

LEN函数不计算尾部空格。

```
SELECT LEN('Hello') -- 返回5

SELECT LEN('Hello '); -- 返回5
```

DATALENGTH函数会计算尾部空格。

```
SELECT DATALENGTH('Hello') -- 返回5

SELECT DATALENGTH('Hello '); -- 返回6
```

Chapter 41: String Functions

String functions perform operations on string values and return either numeric or string values.

Using string functions, you can, for example, combine data, extract a substring, compare strings, or convert a string to all uppercase or lowercase characters.

Section 41.1: Concatenate

In (standard ANSI/ISO) SQL, the operator for string concatenation is ||. This syntax is supported by all major databases except SQL Server:

```
SELECT 'Hello' || 'World' || '!'; --returns HelloWorld!
```

Many databases support a **CONCAT** function to join strings:

```
SELECT CONCAT('Hello', 'World'); --returns 'HelloWorld'
```

Some databases support using **CONCAT** to join more than two strings (Oracle does not):

```
SELECT CONCAT('Hello', 'World', '!'); --returns 'HelloWorld!'
```

In some databases, non-string types must be cast or converted:

```
SELECT CONCAT('Foo', CAST(42 AS VARCHAR(5)), 'Bar'); --returns 'Foo42Bar'
```

Some databases (e.g., Oracle) perform implicit lossless conversions. For example, a **CONCAT** on a **CLOB** and **NCLOB** yields a **NCLOB**. A **CONCAT** on a number and a varchar2 results in a varchar2, etc.:

```
SELECT CONCAT(CONCAT('Foo', 42), 'Bar') FROM dual; --returns Foo42Bar
```

Some databases can use the non-standard + operator (but in most, + works only for numbers):

```
SELECT 'Foo' + CAST(42 AS VARCHAR(5)) + 'Bar';
```

On SQL Server < 2012, where **CONCAT** is not supported, + is the only way to join strings.

Section 41.2: Length

SQL Server

The LEN doesn't count the trailing space.

```
SELECT LEN('Hello') -- returns 5

SELECT LEN('Hello '); -- returns 5
```

The DATALENGTH counts the trailing space.

```
SELECT DATALENGTH('Hello') -- returns 5

SELECT DATALENGTH('Hello '); -- returns 6
```

需要注意的是，DATALENGTH返回的是字符串底层字节表示的长度，这取决于用于存储字符串的字符集等因素。

```
DECLARE @str varchar(100) = 'Hello ' -- varchar通常是ASCII字符串，每个字符占用1字节

SELECT DATALENGTH(@str) -- 返回6


DECLARE @nstr nvarchar(100) = 'Hello ' -- nvarchar是Unicode字符串，每个字符占用2字节
SELECT DATALENGTH(@nstr) -- 返回12
```

Oracle

语法：Length (char)

示例：

```
SELECT Length('Bible') FROM dual; --返回 5
SELECT Length('righteousness') FROM dual; --返回 13
SELECT Length(NULL) FROM dual; --返回 NULL
```

另见：LengthB, LengthC, Length2, Length4

第41.3节：修剪空格

Trim用于移除选中内容开头或结尾的空白字符

在MSSQL中没有单独的TRIM()函数

```
SELECT LTRIM(' Hello ') --返回 'Hello '
SELECT RTRIM(' Hello ') --返回 ' Hello'
SELECT LTRIM(RTRIM(' Hello ')) --返回 'Hello'
```

MySql和Oracle

```
SELECT TRIM(' Hello ') --返回 'Hello'
```

第41.4节：大小写转换

```
SELECT UPPER('HelloWorld') --返回 'HELLOWORLD'
SELECT LOWER('HelloWorld') --返回 'helloworld'
```

第41.5节：拆分

使用字符分隔符拆分字符串表达式。注意STRING_SPLIT()是一个表值函数。

```
SELECT value FROM STRING_SPLIT('Lorem ipsum dolor sit amet.', ' ');
```

结果：

value

Lorem
ipsum
dolor
sit

It should be noted though, that DATALENGTH returns the length of the underlying byte representation of the string, which depends, i.a., on the charset used to store the string.

```
DECLARE @str varchar(100) = 'Hello ' --varchar is usually an ASCII string, occupying 1 byte per char
SELECT DATALENGTH(@str) -- returns 6


DECLARE @nstr nvarchar(100) = 'Hello ' --nvarchar is a unicode string, occupying 2 bytes per char
SELECT DATALENGTH(@nstr) -- returns 12
```

Oracle

Syntax: Length (char)

Examples:

```
SELECT Length('Bible') FROM dual; --Returns 5
SELECT Length('righteousness') FROM dual; --Returns 13
SELECT Length(NULL) FROM dual; --Returns NULL
```

See Also: LengthB, LengthC, Length2, Length4

Section 41.3: Trim empty spaces

Trim is used to remove write-space at the beginning or end of selection

In MSSQL there is no single TRIM()

```
SELECT LTRIM(' Hello ') --returns 'Hello '
SELECT RTRIM(' Hello ') --returns ' Hello'
SELECT LTRIM(RTRIM(' Hello ')) --returns 'Hello'
```

MySql and Oracle

```
SELECT TRIM(' Hello ') --returns 'Hello'
```

Section 41.4: Upper & lower case

```
SELECT UPPER('HelloWorld') --returns 'HELLOWORLD'
SELECT LOWER('HelloWorld') --returns 'helloworld'
```

Section 41.5: Split

Splits a string expression using a character separator. Note that STRING_SPLIT() is a table-valued function.

```
SELECT value FROM STRING_SPLIT('Lorem ipsum dolor sit amet.', ' ');
```

Result:

value

Lorem
ipsum
dolor
sit

```
amet.
```

第41.6节：替换

语法：

REPLACE(要搜索的字符串 , 要搜索并替换的字符串 , 要放入原字符串的字符串)

示例：

```
SELECT REPLACE( 'Peter Steve Tom', 'Steve', 'Billy' ) --返回值：Peter Billy Tom
```

第41.7节：正则表达式（REGEXP）

MySQL 版本 ≥ 3.19

检查字符串是否匹配正则表达式（由另一个字符串定义）。

```
SELECT 'bedded' REGEXP '[a-f]' -- 返回True
SELECT 'beam' REGEXP '[a-f]' -- 返回False
```

第41.8节：子字符串

语法为：SUBSTRING(string_expression, start,length)。请注意，SQL 字符串的索引从 1 开始。

```
SELECT SUBSTRING('Hello', 1, 2) --返回 'He'
SELECT SUBSTRING('Hello', 3, 3) --返回 'llo'
```

这通常与 LEN() 函数结合使用，以获取长度未知的字符串的最后 n 个字符。

```
DECLARE @str1 VARCHAR(10) = 'Hello', @str2 VARCHAR(10) = 'FooBarBaz';
SELECT SUBSTRING(@str1, LEN(@str1) - 2, 3) --返回 'llo'
SELECT SUBSTRING(@str2, LEN(@str2) - 2, 3) --返回 'Baz'
```

第41.9节：Stu

将一个字符串插入另一个字符串中，替换某个位置上的0个或多个字符。

注意：start 位置是从1开始索引（索引从1开始，而不是0）。

语法：

STUFF (character_expression , start , length , replaceWith_expression)

示例：

```
SELECT STUFF('FooBarBaz', 4, 3, 'Hello') --返回 'FooHelloBaz'
```

第41.10节：LEFT - RIGHT

语法是：
LEFT (字符串表达式，整数)

```
amet.
```

Section 41.6: Replace

Syntax:

REPLACE(String to search , String to search for and replace , String to place into the original string)

Example:

```
SELECT REPLACE( 'Peter Steve Tom', 'Steve', 'Billy' ) --Return Values: Peter Billy Tom
```

Section 41.7: REGEXP

MySQL Version ≥ 3.19

Checks if a string matches a regular expression (defined by another string).

```
SELECT 'bedded' REGEXP '[a-f]' -- returns True
SELECT 'beam' REGEXP '[a-f]' -- returns False
```

Section 41.8: Substring

Syntax is: **SUBSTRING** (string_expression, start, length). Note that SQL strings are 1-indexed.

```
SELECT SUBSTRING('Hello', 1, 2) --returns 'He'
SELECT SUBSTRING('Hello', 3, 3) --returns 'llo'
```

This is often used in conjunction with the **LEN()** function to get the last n characters of a string of unknown length.

```
DECLARE @str1 VARCHAR(10) = 'Hello', @str2 VARCHAR(10) = 'FooBarBaz';
SELECT SUBSTRING(@str1, LEN(@str1) - 2, 3) --returns 'llo'
SELECT SUBSTRING(@str2, LEN(@str2) - 2, 3) --returns 'Baz'
```

Section 41.9: Stuff

Stuff a string into another, replacing 0 or more characters at a certain position.

Note: start position is 1-indexed (you start indexing at 1, not 0).

Syntax:

STUFF (character_expression , start , length , replaceWith_expression)

Example:

```
SELECT STUFF('FooBarBaz', 4, 3, 'Hello') --returns 'FooHelloBaz'
```

Section 41.10: LEFT - RIGHT

Syntax is:
LEFT (string-expression , integer)

RIGHT (字符串表达式, 整数)

```
SELECT LEFT('Hello',2)  --返回 He
SELECT RIGHT('Hello',2) --返回 lo
```

Oracle SQL 没有 LEFT 和 RIGHT 函数。它们可以用 SUBSTR 和 LENGTH 来模拟。

SUBSTR (字符串表达式, 1, 整数)

SUBSTR (字符串表达式, length(字符串表达式)-整数+1, 整数)

```
SELECT SUBSTR('Hello',1,2)  --返回 He
SELECT SUBSTR('Hello',LENGTH('Hello')-2+1,2) --返回 lo
```

第41.11节：REVERSE

语法是：REVERSE (字符串表达式)

```
SELECT REVERSE('Hello') --返回 olleH
```

第41.12节：复制

REPLICATE函数将字符串自身连接指定次数。

语法为：REPLICATE (字符串表达式 , 整数)

```
SELECT REPLICATE('Hello',4) --返回 'HelloHelloHelloHello'
```

第41.13节：SQL中Select和Update查询的Replace函数

SQL中的Replace函数用于更新字符串内容。MySQL、Oracle和SQL Server中调用该函数为REPLACE()。

Replace函数的语法是：

```
REPLACE (str, find, repl)
```

下面的示例将Employees表中出现的South替换为Southern：

FirstName	Address
詹姆斯	纽约南部
约翰	波士顿南部
迈克尔	圣地亚哥南部

选择语句：

如果我们应用以下替换函数：

```
SELECT
名字,
    REPLACE (地址, 'South', 'Southern') 地址
FROM 员工
ORDER BY 名字
```

结果：

RIGHT (string-expression , integer)

```
SELECT LEFT('Hello',2)  --return He
SELECT RIGHT('Hello',2) --return lo
```

Oracle SQL doesn't have LEFT and RIGHT functions. They can be emulated with SUBSTR and LENGTH.

SUBSTR (string-expression, 1, integer)

SUBSTR (string-expression, length(string-expression)-integer+1, integer)

```
SELECT SUBSTR('Hello',1,2)  --return He
SELECT SUBSTR('Hello',LENGTH('Hello')-2+1,2) --return lo
```

Section 41.11: REVERSE

Syntax is: REVERSE (string-expression)

```
SELECT REVERSE('Hello') --returns olleH
```

Section 41.12: REPLICATE

The REPLICATE function concatenates a string with itself a specified number of times.

Syntax is: REPLICATE (string-expression , integer)

```
SELECT REPLICATE ('Hello',4) --returns 'HelloHelloHelloHello'
```

Section 41.13: Replace function in sql Select and Update query

The Replace function in SQL is used to update the content of a string. The function call is REPLACE() for MySQL, Oracle, and SQL Server.

The syntax of the Replace function is:

```
REPLACE (str, find, repl)
```

The following example replaces occurrences of South with Southern in Employees table:

FirstName	Address
James	South New York
John	South Boston
Michael	South San Diego

Select Statement：

If we apply the following Replace function:

```
SELECT
    FirstName,
    REPLACE (Address, 'South', 'Southern') Address
FROM Employees
ORDER BY FirstName
```

Result:

FirstName	Address
詹姆斯	纽约南部
约翰	波士顿南部
迈克尔	南圣地亚哥

更新语句：

我们可以通过以下方法使用替换函数对表进行永久更改。

```
更新 员工
设置 城市 = (地址, '南', '南部');
```

更常见的方法是将其与 WHERE 子句结合使用，如下所示：

```
更新 员工
设置 地址 = (地址, '南', '南部')
条件 地址 LIKE '南%';
```

第41.14节：INSTR

返回子字符串首次出现的位置索引（未找到则返回0）

语法：INSTR (字符串, 子字符串)

```
选择 INSTR('FooBarBar', 'Bar') -- 返回 4
选择 INSTR('FooBarBar', 'Xar') -- 返回 0
```

第41.15节：PARSENAME

数据库：SQL Server

PARSENAME 函数返回给定字符串（对象名）的特定部分。对象名可能包含类似对象名、所有者名、数据库名和服务器的字符串。

更多详情 [MSDN:PARSENAME](#)

语法

```
PARSENAME('要解析的字符串名',部分索引)
```

示例

要获取对象名，使用部分索引1

```
SELECT PARSENAME('ServerName.DatabaseName.SchemaName.ObjectName',1) // 返回 `ObjectName`
SELECT PARSENAME('[1012-1111].SchoolDatabase.school.Student',1) // 返回 `Student`
```

要获取架构名，使用部分索引2

```
SELECT PARSENAME('ServerName.DatabaseName.SchemaName.ObjectName',2) // 返回 `SchemaName`
SELECT PARSENAME('[1012-1111].SchoolDatabase.school.Student',2) // 返回 `school`
```

要获取数据库名称，请使用部分索引3

FirstName	Address
James	Southern New York
John	Southern Boston
Michael	Southern San Diego

Update Statement：

We can use a replace function to make permanent changes in our table through following approach.

```
Update Employees
Set city = (Address, 'South', 'Southern');
```

A more common approach is to use this in conjunction with a WHERE clause like this:

```
Update Employees
Set Address = (Address, 'South', 'Southern')
Where Address LIKE 'South%';
```

Section 41.14: INSTR

Return the index of the first occurrence of a substring (zero if not found)

Syntax: INSTR (string, substring)

```
SELECT INSTR('FooBarBar', 'Bar') -- return 4
SELECT INSTR('FooBarBar', 'Xar') -- return 0
```

Section 41.15: PARSENAME

DATABASE : SQL Server

PARSENAME function returns the specific part of given string(object name). object name may contains string like object name,owner name, database name and server name.

More details [MSDN:PARSENAME](#)

Syntax

```
PARSENAME('NameOfStringToParse',PartIndex)
```

Example

To get object name use part index 1

```
SELECT PARSENAME('ServerName.DatabaseName.SchemaName.ObjectName',1) // returns `ObjectName`
SELECT PARSENAME('[1012-1111].SchoolDatabase.school.Student',1) // returns `Student`
```

To get schema name use part index 2

```
SELECT PARSENAME('ServerName.DatabaseName.SchemaName.ObjectName',2) // returns `SchemaName`
SELECT PARSENAME('[1012-1111].SchoolDatabase.school.Student',2) // returns `school`
```

To get database name use part index 3

```
SELECT PARSENAME('ServerName.DatabaseName.SchemaName.ObjectName',3) // 返回 `DatabaseName`  
SELECT PARSENAME('[1012-1111].SchoolDatabase.school.Student',3) // 返回 `SchoolDatabase`
```

要获取服务器名称，请使用部分索引4

```
SELECT PARSENAME('ServerName.DatabaseName.SchemaName.ObjectName',4) // 返回 `ServerName`  
SELECT PARSENAME('[1012-1111].SchoolDatabase.school.Student',4) // 返回 `[1012-1111]`
```

如果指定的部分在给定的对象名称字符串中不存在，PARSENAME将返回null

```
SELECT PARSENAME('ServerName.DatabaseName.SchemaName.ObjectName',3) // returns `DatabaseName`  
SELECT PARSENAME('[1012-1111].SchoolDatabase.school.Student',3) // returns `SchoolDatabase`
```

To get server name use part index 4

```
SELECT PARSENAME('ServerName.DatabaseName.SchemaName.ObjectName',4) // returns `ServerName`  
SELECT PARSENAME('[1012-1111].SchoolDatabase.school.Student',4) // returns `[1012-1111]`
```

PARSENAME will returns null is specified part is not present in given object name string

第42章：函数（聚合）

第42.1节：条件聚合

付款表

客户	付款类型	金额
彼得	信用	100
彼得	信用	300
约翰	信用	1000
约翰	借记	500

```
选择客户,
    sum(case when payment_type = 'credit' then amount else 0 end) as credit,
    sum(case when payment_type = 'debit' then amount else 0 end) as debit
from payments
按客户分组
```

结果：

客户	信用	借记
彼得	400	0
约翰	1000	500

```
选择客户,
    sum(case when payment_type = 'credit' then 1 else 0 end) as credit_transaction_count,
    sum(case when payment_type = 'debit' then 1 else 0 end) as debit_transaction_count
from payments
按客户分组
```

结果：

客户	信用交易次数	借记交易次数
彼得	2	0
约翰	1	1

第42.2节：列表连接

部分归功于本SO答案。

列表连接通过将每组的值合并为单个字符串来聚合一列或表达式。可以指定用于分隔每个值的字符串（省略时为空格或逗号）以及结果中值的顺序。虽然这不是SQL标准的一部分，但每个主要的关系数据库供应商都以自己的方式支持它。

MySQL

```
SELECT ColumnA
    , GROUP_CONCAT(ColumnB ORDER BY ColumnB SEPARATOR ',') AS ColumnBs
FROM TableName
GROUP BY ColumnA
ORDER BY ColumnA;
```

Oracle 和 DB2

```
SELECT ColumnA
    , LISTAGG(ColumnB, ',') WITHIN GROUP (ORDER BY ColumnB) AS ColumnBs
FROM TableName
```

Chapter 42: Functions (Aggregate)

Section 42.1: Conditional aggregation

Payments Table

Customer	Payment_type	Amount
Peter	Credit	100
Peter	Credit	300
John	Credit	1000
John	Debit	500

```
select customer,
    sum(case when payment_type = 'credit' then amount else 0 end) as credit,
    sum(case when payment_type = 'debit' then amount else 0 end) as debit
from payments
group by customer
```

Result:

Customer	Credit	Debit
Peter	400	0
John	1000	500

```
select customer,
    sum(case when payment_type = 'credit' then 1 else 0 end) as credit_transaction_count,
    sum(case when payment_type = 'debit' then 1 else 0 end) as debit_transaction_count
from payments
group by customer
```

Result:

Customer	credit_transaction_count	debit_transaction_count
Peter	2	0
John	1	1

Section 42.2: List Concatenation

Partial credit to [this](#) SO answer.

List Concatenation aggregates a column or expression by combining the values into a single string for each group. A string to delimit each value (either blank or a comma when omitted) and the order of the values in the result can be specified. While it is not part of the SQL standard, every major relational database vendor supports it in their own way.

MySQL

```
SELECT ColumnA
    , GROUP_CONCAT(ColumnB ORDER BY ColumnB SEPARATOR ',') AS ColumnBs
FROM TableName
GROUP BY ColumnA
ORDER BY ColumnA;
```

Oracle & DB2

```
SELECT ColumnA
    , LISTAGG(ColumnB, ',') WITHIN GROUP (ORDER BY ColumnB) AS ColumnBs
FROM TableName
```

```
GROUP BY ColumnA
ORDER BY ColumnA;
```

PostgreSQL

```
SELECT ColumnA
      , STRING_AGG(ColumnB, ',' ORDER BY ColumnB) AS ColumnBs
FROM TableName
GROUP BY ColumnA
ORDER BY ColumnA;
```

SQL Server

SQL Server 2016 及更早版本

(包含 CTE 以鼓励DRY 原则)

```
WITH CTE_TableName AS (
    SELECT ColumnA, ColumnB
    FROM TableName)
SELECT t0.ColumnA
      , STUFF(
    SELECT ',' + t1.ColumnB
    FROM CTE_TableName t1
    WHERE t1.ColumnA = t0.ColumnA
    ORDER BY t1.ColumnB
    FOR XML PATH('')), 1, 1, '' ) AS ColumnBs
FROM CTE_TableName t0
GROUP BY t0.ColumnA
按 ColumnA 排序；
```

SQL Server 2017 和 SQL Azure

```
SELECT ColumnA
      , STRING_AGG(ColumnB, ',') WITHIN GROUP (ORDER BY ColumnB) AS ColumnBs
FROM TableName
GROUP BY ColumnA
ORDER BY ColumnA;
```

SQLite

无排序：

```
SELECT ColumnA
      , GROUP_CONCAT(ColumnB, ',') AS ColumnBs
FROM TableName
GROUP BY ColumnA
ORDER BY ColumnA;
```

排序需要子查询或公共表表达式（CTE）：

```
WITH CTE_TableName AS (
    SELECT ColumnA, ColumnB
    FROM TableName
    ORDER BY ColumnA, ColumnB)
SELECT ColumnA
      , GROUP_CONCAT(ColumnB, ',') AS ColumnBs
FROM CTE_TableName
GROUP BY ColumnA
ORDER BY ColumnA;
```

```
GROUP BY ColumnA
ORDER BY ColumnA;
```

PostgreSQL

```
SELECT ColumnA
      , STRING_AGG(ColumnB, ',' ORDER BY ColumnB) AS ColumnBs
FROM TableName
GROUP BY ColumnA
ORDER BY ColumnA;
```

SQL Server

SQL Server 2016 and earlier

(CTE included to encourage the [DRY principle](#))

```
WITH CTE_TableName AS (
    SELECT ColumnA, ColumnB
    FROM TableName)
SELECT t0.ColumnA
      , STUFF(
    SELECT ',' + t1.ColumnB
    FROM CTE_TableName t1
    WHERE t1.ColumnA = t0.ColumnA
    ORDER BY t1.ColumnB
    FOR XML PATH('')), 1, 1, '' ) AS ColumnBs
FROM CTE_TableName t0
GROUP BY t0.ColumnA
ORDER BY ColumnA;
```

SQL Server 2017 and SQL Azure

```
SELECT ColumnA
      , STRING_AGG(ColumnB, ',') WITHIN GROUP (ORDER BY ColumnB) AS ColumnBs
FROM TableName
GROUP BY ColumnA
ORDER BY ColumnA;
```

SQLite

without ordering:

```
SELECT ColumnA
      , GROUP_CONCAT(ColumnB, ',') AS ColumnBs
FROM TableName
GROUP BY ColumnA
ORDER BY ColumnA;
```

ordering requires a subquery or CTE:

```
WITH CTE_TableName AS (
    SELECT ColumnA, ColumnB
    FROM TableName
    ORDER BY ColumnA, ColumnB)
SELECT ColumnA
      , GROUP_CONCAT(ColumnB, ',') AS ColumnBs
FROM CTE_TableName
GROUP BY ColumnA
ORDER BY ColumnA;
```

第42.3节：SUM

Sum 函数对组内所有行的值求和。如果省略了 group by 子句，则对所有行求和。

```
select sum(salary) 总薪资
from 员工;
```

总薪资
2500

```
select 部门编号, sum(salary) 总薪资
from 员工
group by 部门编号;
```

部门编号	总薪资
1	2000
2	500

第42.4节：AVG()

聚合函数AVG()返回给定表达式的平均值，通常是列中的数值。
假设我们有一个包含全球各城市年度人口统计的表。纽约市的记录类似于以下内容：

城市名称	人口	年份
纽约市	8,550,405	2015
纽约市
纽约市	8,000,906	2005

要从包含城市名称、人口测量值和过去十年测量年份的表中选择美国纽约市的平均人口：

```
select city_name, AVG(population) avg_population
from city_population
where city_name = 'NEW YORK CITY';
```

注意查询中没有测量年份，因为人口是跨时间取平均的。

城市名称	avg_population
纽约市	8,250,754

注意：AVG() 函数会将值转换为数值类型。在处理日期时尤其需要注意这一点。

第42.5节：计数

你可以计算行数：

```
SELECT count(*) 总行数
FROM 员工;
```

总行数

Section 42.3: SUM

Sum function sum the value of all the rows in the group. If the group by clause is omitted then sums all the rows.

```
select sum(salary) TotalSalary
from employees;
```

TotalSalary
2500

```
select DepartmentId, sum(salary) TotalSalary
from employees
group by DepartmentId;
```

DepartmentId	TotalSalary
1	2000
2	500

Section 42.4: AVG()

The aggregate function AVG() returns the average of a given expression, usually numeric values in a column.
Assume we have a table containing the yearly calculation of population in cities across the world. The records for New York City look similar to the ones below:

city_name	population	year
New York City	8,550,405	2015
New York City
New York City	8,000,906	2005

To select the average population of the New York City, USA from a table containing city names, population measurements, and measurement years for last ten years:

```
select city_name, AVG(population) avg_population
from city_population
where city_name = 'NEW YORK CITY';
```

Notice how measurement year is absent from the query since population is being averaged over time.

city_name	avg_population
New York City	8,250,754

Note: The AVG() function will convert values to numeric types. This is especially important to keep in mind when working with dates.

Section 42.5: Count

You can count the number of rows:

```
SELECT count(*) TotalRows
FROM employees;
```

TotalRows

或者计算每个部门的员工数：

```
SELECT 部门编号, count(*) 员工数量
FROM 员工
GROUP BY 部门编号;
```

部门编号	员工数量
1	3
2	1

您可以对一列/表达式进行计数，效果是不会计数NULL值：

```
SELECT count(ManagerId) mgr
FROM EMPLOYEES;
```

mgr
3

(ManagerID 列中有一个空值)

你也可以在另一个函数中使用DISTINCT，比如COUNT，以仅查找集合中DISTINCT的成员来执行操作。

例如：

```
SELECT COUNT(ContinentCode) AllCount
, COUNT(DISTINCT ContinentCode) SingleCount
FROM Countries;
```

将返回不同的值。SingleCount 只会对各个大陆计数一次，而 AllCount 会包括重复项。

ContinentCode
OC
EU
AS
NA
NA
AF
AF

AllCount: 7 SingleCount: 5

第42.6节：最小值

查找列的最小值：

```
select min(age) from employee;
```

上述示例将返回 employee 表中 age 列的最小值。

语法：

Or count the employees per department:

```
SELECT DepartmentId, count(*) NumEmployees
FROM employees
GROUP BY DepartmentId;
```

DepartmentId	NumEmployees
1	3
2	1

You can count over a column/expression with the effect that will not count the NULL values:

```
SELECT count(ManagerId) mgr
FROM EMPLOYEES;
```

mgr
3

(There is one null value managerID column)

You can also use **DISTINCT** inside of another function such as **COUNT** to only find the **DISTINCT** members of the set to perform the operation on.

For example:

```
SELECT COUNT(ContinentCode) AllCount
, COUNT(DISTINCT ContinentCode) SingleCount
FROM Countries;
```

Will return different values. The *SingleCount* will only Count individual Continents once, while the *AllCount* will include duplicates.

ContinentCode
OC
EU
AS
NA
NA
AF
AF

AllCount: 7 SingleCount: 5

Section 42.6: Min

Find the smallest value of column:

```
select min(age) from employee;
```

Above example will return smallest value for column age of employee table.

Syntax:


```
SELECT MIN(column_name) FROM table_name;
```

第42.7节：最大值

查找列的最大值：

```
select max(age) from employee;
```

上述示例将返回 employee 表中 age 列的最大值。

语法：

```
SELECT MAX(column_name) FROM table_name;
```

```
SELECT MIN(column_name) FROM table_name;
```

Section 42.7: Max

Find the maximum value of column:

```
select max(age) from employee;
```

Above example will return largest value for column age of employee table.

Syntax:

```
SELECT MAX(column_name) FROM table_name;
```

第43章：函数（标量/单行）

SQL 提供了多种内置的标量函数。每个标量函数以一个值作为输入，并为结果集中的每一行返回一个值作为输出。

在 T-SQL 语句中，只要允许表达式的地方，都可以使用标量函数。

第 43.1 节：日期和时间

在 SQL 中，您使用日期和时间数据类型来存储日历信息。这些数据类型包括 time、date、smalldatetime、datetime、datetime2 和 datetimeoffset。每种数据类型都有特定的格式。

数据类型	格式
time	hh:mm:ss[.nnnnnnnn]
date	YYYY-MM-DD
smalldatetime	YYYY-MM-DD hh:mm:ss
datetime	YYYY-MM-DD hh:mm:ss[.nnn]
datetime2	YYYY-MM-DD hh:mm:ss[.nnnnnnnn]
datetimeoffset	YYYY-MM-DD hh:mm:ss[.nnnnnnnn] [+/-]hh:mm

DATENAME 函数返回日期中特定部分的名称或数值。

```
SELECT DATENAME (weekday,'2017-01-14') as Datename
Datename
星期六
```

您可以使用 GETDATE 函数来确定运行当前 SQL 实例的计算机的当前日期和时间。该函数不包含时区差异。

```
SELECT GETDATE() as Systemdate
Systemdate
2017-01-14 11:11:47.7230728
```

DATEDIFF函数返回两个日期之间的差值。

在语法中，datepart 是指定要用来计算差异的日期部分的参数。datepart 可以是年、月、周、日、小时、分钟、秒或毫秒。然后你需要在 startdate 参数中指定起始日期，在 enddate 参数中指定结束日期，以计算两者之间的差异。

```
SELECT SalesOrderID, DATEDIFF(day, OrderDate, ShipDate)
AS 'Processing time'
FROM Sales.SalesOrderHeader
SalesOrderID 处理时间
43659         7
43660         7
43661         7
43662         7
```

DATEADD函数使你能够向特定日期的某个部分添加一个时间间隔。

Chapter 43: Functions (Scalar/Single Row)

SQL provides several built-in scalar functions. Each scalar function takes one value as input and returns one value as output for each row in a result set.

You use scalar functions wherever an expression is allowed within a T-SQL statement.

Section 43.1: Date And Time

In SQL, you use date and time data types to store calendar information. These data types include the time, date, smalldatetime, datetime, datetime2, and datetimeoffset. Each data type has a specific format.

Data type	Format
time	hh:mm:ss[.nnnnnnnn]
date	YYYY-MM-DD
smalldatetime	YYYY-MM-DD hh:mm:ss
datetime	YYYY-MM-DD hh:mm:ss[.nnn]
datetime2	YYYY-MM-DD hh:mm:ss[.nnnnnnnn]
datetimeoffset	YYYY-MM-DD hh:mm:ss[.nnnnnnnn] [+/-]hh:mm

The DATENAME function returns the name or value of a specific part of the date.

```
SELECT DATENAME (weekday, '2017-01-14') as Datename
Datename
Saturday
```

You use the GETDATE function to determine the current date and time of the computer running the current SQL instance. This function doesn't include the time zone difference.

```
SELECT GETDATE() as Systemdate
Systemdate
2017-01-14 11:11:47.7230728
```

The DATEDIFF function returns the difference between two dates.

In the syntax, datepart is the parameter that specifies which part of the date you want to use to calculate difference. The datepart can be year, month, week, day, hour, minute, second, or millisecond. You then specify the start date in the startdate parameter and the end date in the enddate parameter for which you want to find the difference.

```
SELECT SalesOrderID, DATEDIFF(day, OrderDate, ShipDate)
AS 'Processing time'
FROM Sales.SalesOrderHeader
SalesOrderID Processing time
43659         7
43660         7
43661         7
43662         7
```

The DATEADD function enables you to add an interval to part of a specific date.

```
SELECT DATEADD (day, 20, '2017-01-14') AS Added20MoreDays

增加了20天
2017-02-03 00:00:00.000
```

第43.2节：字符修改

字符修改函数包括将字符转换为大写或小写字符，将数字转换为格式化数字，执行字符操作等。

lower(char) 函数将给定的字符参数转换为小写字符。

```
SELECT customer_id, lower(customer_last_name) FROM customer;
```

将返回客户的姓氏，从“SMITH”变为“smith”。

第43.3节：配置和转换函数

SQL中配置函数的一个例子是 @@SERVERNAME 函数。该函数提供运行SQL的本地服务器名称。

```
SELECT @@SERVERNAME AS 'Server'

服务器
SQL064
```

在 SQL 中，大多数数据转换是隐式发生的，无需用户干预。

对于无法隐式完成的转换，可以使用CAST或CONVERT函数。

CAST函数的语法比CONVERT函数简单，但功能有限。

这里，我们同时使用CAST和CONVERT函数将datetime数据类型转换为varchar数据类型。

CAST函数始终使用默认的风格设置。例如，它会使用格式YYYY-MM-DD表示日期和时间。

CONVERT函数使用您指定的日期和时间样式。在此例中，3指定日期格式为dd/mm/yy。

```
USE AdventureWorks2012
GO
SELECT  FirstName + ' ' + LastName + ' 于 ' +
        CAST(HireDate AS varchar(20)) AS 'Cast',
        FirstName + ' ' + LastName + ' 于 ' +
        CONVERT(varchar, HireDate, 3) AS 'Convert'
FROM    Person.Person AS p
JOIN    HumanResources.Employee AS e
ON      p.BusinessEntityID = e.BusinessEntityID
GO
```

Cast	Convert
大卫·哈密尔顿于2003-02-04被雇佣	大卫·哈密尔顿于04/02/03被雇佣

另一个转换函数的例子是PARSE函数。该函数将字符串转换为指定的数据类型。

在函数的语法中，您需要指定必须转换的字符串，AS关键字，然后是所需的

```
SELECT DATEADD (day, 20, '2017-01-14') AS Added20MoreDays

Added20MoreDays
2017-02-03 00:00:00.000
```

Section 43.2: Character modifications

Character modifying functions include converting characters to upper or lower case characters, converting numbers to formatted numbers, performing character manipulation, etc.

The lower(char) function converts the given character parameter to be lower-cased characters.

```
SELECT customer_id, lower(customer_last_name) FROM customer;
```

would return the customer's last name changed from "SMITH" to "smith".

Section 43.3: Configuration and Conversion Function

An example of a configuration function in SQL is the @@SERVERNAME function. This function provides the name of the local server that's running SQL.

```
SELECT @@SERVERNAME AS 'Server'

Server
SQL064
```

In SQL, most data conversions occur implicitly, without any user intervention.

To perform any conversions that can't be completed implicitly, you can use the CAST or CONVERT functions.

The CAST function syntax is simpler than the CONVERT function syntax, but is limited in what it can do.

In here, we use both the CAST and CONVERT functions to convert the datetime data type to the varchar data type.

The CAST function always uses the default style setting. For example, it will represent dates and times using the format YYYY-MM-DD.

The CONVERT function uses the date and time style you specify. In this case, 3 specifies the date format dd/mm/yy.

```
USE AdventureWorks2012
GO
SELECT  FirstName + ' ' + LastName + ' was hired on ' +
        CAST(HireDate AS varchar(20)) AS 'Cast',
        FirstName + ' ' + LastName + ' was hired on ' +
        CONVERT(varchar, HireDate, 3) AS 'Convert'
FROM    Person.Person AS p
JOIN    HumanResources.Employee AS e
ON      p.BusinessEntityID = e.BusinessEntityID
GO
```

Cast	Convert
David Hamiltion was hired on 2003-02-04	David Hamiltion was hired on 04/02/03

Another example of a conversion function is the PARSE function. This function converts a string to a specified data type.

In the syntax for the function, you specify the string that must be converted, the AS keyword, and then the required

数据类型。可选地，您还可以指定字符串值应格式化的文化。如果未指定，则使用会话的语言。

如果字符串值无法转换为数字、日期或时间格式，将导致错误。您需要使用CAST或CONVERT进行转换。

```
SELECT  PARSE('Monday, 13 August 2012' AS datetime2 USING  'en-US') AS   'Date in English'

      Date in English
2012-08-13 00:00:00.0000000
```

第43.4节：逻辑和数学函数

SQL 有两个逻辑函数– CHOOSE和IIF。

CHOOSE函数根据列表中的位置返回列表中的一个项。该位置由索引指定。

在语法中，index 参数指定项，是一个整数。val_1 ... val_n 参数标识值列表。

```
SELECT  CHOOSE(2, '人力资源', '销售', '行政', '市场营销'  ) AS   结果;

结果
销售部
```

在此示例中，您使用CHOOSE函数返回部门列表中的第二项。

IIF函数根据特定条件返回两个值中的一个。如果条件为真，则返回真值。否则返回假值。

在语法中，boolean_expression 参数指定布尔表达式。true_value 参数指定当 boolean_expression 计算为真时应返回的值，false_value 参数指定当 boolean_expression 计算为假时应返回的值。

```
SELECT  BusinessEntityID,  SalesYTD,
      IIF(SalesYTD  > 200000,  '奖金',  '无奖金')  AS   '奖金?'
FROM    Sales.SalesPerson
GO

BusinessEntityID    销售年初至今    奖金？
274                  559697.5639    奖金
275                  3763178.1787  奖金
285                  172524.4512   无奖金
```

在此示例中，您使用IIF函数返回两个值中的一个。如果销售人员的年初至今销售额超过200,000，则该人员有资格获得奖金。低于200,000的数值意味着员工不符合奖金资格。

SQL 包含多个数学函数，您可以使用它们对输入值进行计算并返回数值结果。

一个例子是SIGN函数，它返回一个表示表达式符号的值。值为 -1 表示负数表达式，值为 +1 表示正数表达式，0 表示零。

data type. Optionally, you can also specify the culture in which the string value should be formatted. If you don't specify this, the language for the session is used.

If the string value can't be converted to a numeric, date, or time format, it will result in an error. You'll then need to use CAST or CONVERT for the conversion.

```
SELECT PARSE('Monday, 13 August 2012' AS datetime2 USING 'en-US') AS 'Date in English'

      Date in English
2012-08-13 00:00:00.0000000
```

Section 43.4: Logical and Mathmetical Function

SQL has two logical functions – CHOOSE and IIF.

The CHOOSE function returns an item from a list of values, based on its position in the list. This position is specified by the index.

In the syntax, the index parameter specifies the item and is a whole number, or integer. The val_1 ... val_n parameter identifies the list of values.

```
SELECT CHOOSE(2, 'Human Resources', 'Sales', 'Admin', 'Marketing' ) AS Result;

Result
Sales
```

In this example, you use the CHOOSE function to return the second entry in a list of departments.

The IIF function returns one of two values, based on a particular condition. If the condition is true, it will return true value. Otherwise it will return a false value.

In the syntax, the boolean_expression parameter specifies the Boolean expression. The true_value parameter specifies the value that should be returned if the boolean_expression evaluates to true and the false_value parameter specifies the value that should be returned if the boolean_expression evaluates to false.

```
SELECT BusinessEntityID, SalesYTD,
      IIF(SalesYTD > 200000, 'Bonus', 'No Bonus') AS 'Bonus?'
FROM Sales.SalesPerson
GO

BusinessEntityID    SalesYTD    Bonus?
274                  559697.5639    Bonus
275                  3763178.1787    Bonus
285                  172524.4512   No Bonus
```

In this example, you use the IIF function to return one of two values. If a sales person's year-to-date sales are above 200,000, this person will be eligible for a bonus. Values below 200,000 mean that employees don't qualify for bonuses.

SQL includes several mathematical functions that you can use to perform calculations on input values and return numeric results.

One example is the SIGN function, which returns a value indicating the sign of an expression. The value of -1 indicates a negative expression, the value of +1 indicates a positive expression, and 0 indicates zero.

```
SELECT SIGN(-20) AS 'Sign'
```

Sign

-1

在示例中，输入是一个负数，因此结果窗格列出了结果 -1。

另一个数学函数是POWER函数。该函数返回一个表达式的指定次幂的值。

在语法中，float_expression 参数指定表达式，y 参数指定要将表达式提升到的幂。

```
SELECT POWER(50, 3) AS Result
```

结果

125000

```
SELECT SIGN(-20) AS 'Sign'
```

Sign

-1

In the example, the input is a negative number, so the Results pane lists the result -1.

Another mathematical function is the **POWER** function. This function provides the value of an expression raised to a specified power.

In the syntax, the float_expression parameter specifies the expression, and the y parameter specifies the power to which you want to raise the expression.

```
SELECT POWER(50, 3) AS Result
```

Result

125000

第44章：函数（分析函数）

您可以使用分析函数根据一组值来确定结果。例如，您可以使用此类函数来计算累计总和、百分比或组内的最高结果。

第44.1节：LAG和LEAD

LAG函数提供当前行之前的行数据，且这些行属于同一结果集。例如，在SELECT语句中，您可以将当前行的值与前一行的值进行比较。

您使用标量表达式来指定应比较的值。offset参数表示用于比较的当前行之前的行数。如果未指定行数，则默认使用一行。

default参数指定当offset处的表达式值为NULL时应返回的值。如果未指定该值，则返回NULL。

LEAD函数提供当前行之后的行数据，且这些行属于同一结果集。例如，在SELECT语句中，您可以将当前行的值与下一行的值进行比较。

您使用标量表达式来指定应比较的值。offset参数表示用于比较的当前行之后的行数。

您使用default参数指定当offset处的表达式值为NULL时应返回的值。如果未指定这些参数，则默认使用一行，且返回值为NULL。

```
选择 BusinessEntityID, SalesYTD,
    LEAD(SalesYTD, 1, 0) OVER(ORDER BY BusinessEntityID) AS "Lead value",
    LAG(SalesYTD, 1, 0) OVER(ORDER BY BusinessEntityID) AS "Lag value"
FROM SalesPerson;
```

此示例使用 LEAD 和 LAG 函数，将每位员工迄今为止的销售额与其上方和下方员工的销售额进行比较，记录按 BusinessEntityID 列排序。

BusinessEntityID	销售年初至今	领先值	滞后值
274	559697.5639	3763178.1787	0.0000
275	3763178.1787	4251368.5497	559697.5639
276	4251368.5497	3189418.3662	3763178.1787
277	3189418.3662	1453719.4653	4251368.5497
278	1453719.4653	2315185.6110	3189418.3662
279	2315185.6110	1352577.1325	1453719.4653

第44.2节：PERCENTILE_DISC 和 PERCENTILE_CONT

PERCENTILE_DISC 函数列出累计分布超过您通过 numeric_literal 参数提供的百分位数的第一个条目的值。

这些值根据 WITHIN GROUP 子句指定的行集或分区进行分组。

PERCENTILE_CONT 函数类似于 PERCENTILE_DISC 函数，但返回第一个匹配条目与下一个条目之和的平均值。

Chapter 4 4: Functions (Analytic)

You use analytic functions to determine values based on groups of values. For example, you can use this type of function to determine running totals, percentages, or the top result within a group.

Section 4 4.1: LAG and LEAD

The LAG function provides data on rows before the current row in the same result set. For example, in a SELECT statement, you can compare values in the current row with values in a previous row.

You use a scalar expression to specify the values that should be compared. The offset parameter is the number of rows before the current row that will be used in the comparison. If you don't specify the number of rows, the default value of one row is used.

The default parameter specifies the value that should be returned when the expression at offset has a NULL value. If you don't specify a value, a value of NULL is returned.

The LEAD function provides data on rows after the current row in the row set. For example, in a SELECT statement, you can compare values in the current row with values in the following row.

You specify the values that should be compared using a scalar expression. The offset parameter is the number of rows after the current row that will be used in the comparison.

You specify the value that should be returned when the expression at offset has a NULL value using the default parameter. If you don't specify these parameters, the default of one row is used and a value of NULL is returned.

```
SELECT BusinessEntityID, SalesYTD,
    LEAD(SalesYTD, 1, 0) OVER(ORDER BY BusinessEntityID) AS "Lead value",
    LAG(SalesYTD, 1, 0) OVER(ORDER BY BusinessEntityID) AS "Lag value"
FROM SalesPerson;
```

This example uses the LEAD and LAG functions to compare the sales values for each employee to date with those of the employees listed above and below, with records ordered based on the BusinessEntityID column.

BusinessEntityID	SalesYTD	Lead value	Lag value
274	559697.5639	3763178.1787	0.0000
275	3763178.1787	4251368.5497	559697.5639
276	4251368.5497	3189418.3662	3763178.1787
277	3189418.3662	1453719.4653	4251368.5497
278	1453719.4653	2315185.6110	3189418.3662
279	2315185.6110	1352577.1325	1453719.4653

Section 4 4.2: PERCENTILE_DISC and PERCENTILE_CONT

The PERCENTILE_DISC function lists the value of the first entry where the cumulative distribution is higher than the percentile that you provide using the numeric_literal parameter.

The values are grouped by rowset or partition, as specified by the WITHIN GROUP clause.

The PERCENTILE_CONT function is similar to the PERCENTILE_DISC function, but returns the average of the sum of the first matching entry and the next entry.


```
SELECT BusinessEntityID, JobTitle, SickLeaveHours,
    CUME_DIST() OVER(PARTITION BY JobTitle ORDER BY SickLeaveHours ASC)
    AS "Cumulative Distribution",
    PERCENTILE_DISC(0.5) WITHIN GROUP(ORDER BY 病假小时数)
    OVER(PARTITION BY 职位) AS "百分位离散值"
FROM 员工;
```

要找到匹配或超过0.5百分位数的行的确切值，您需要将该百分位数作为数值字面量传递给PERCENTILE_DISC函数。结果集中“百分位离散”列列出了累计分布超过指定百分位数的行的值。

BusinessEntityID	职位名称	病假小时数	累积分布	百分位离散
272	应用专家 55		0.25	56
268	应用专家 56		0.75	56
269	应用专家 56		0.75	56
267	应用专家 57		1	56

要基于一组值进行计算，您可以使用PERCENTILE_CONT函数。结果中的“百分位连续”列列出了结果值与下一个最高匹配值之和的平均值。

```
SELECT BusinessEntityID, JobTitle, SickLeaveHours,
    CUME_DIST() OVER(PARTITION BY JobTitle ORDER BY SickLeaveHours ASC)
    AS "Cumulative Distribution",
    PERCENTILE_DISC(0.5) WITHIN GROUP(ORDER BY 病假小时数)
    OVER(PARTITION BY 职位名称) AS "百分位离散",
    PERCENTILE_CONT(0.5) WITHIN GROUP(ORDER BY 病假小时数)
    OVER(PARTITION BY 职位名称) AS "百分位连续"
FROM 员工;
```

BusinessEntityID	职位名称	病假小时数	累积分布	百分位离散	百分位连续
272	应用专家 55		0.25	56	56
268	应用专家 56		0.75	56	56
269	应用专家 56		0.75	56	56
267	应用专家 57		1	56	56

第44.3节：FIRST_VALUE

您可以使用FIRST_VALUE函数确定有序结果集中第一个值，该值通过标量表达式进行识别。

```
SELECT StateProvinceID, Name, TaxRate,
    FIRST_VALUE(StateProvinceID)
    OVER(ORDER BY TaxRate ASC) AS FirstValue
FROM SalesTaxRate;
```

在此示例中，FIRST_VALUE函数用于返回税率最低的州或省的ID。OVER子句用于对税率进行排序，以获取最低税率。

州省ID	姓名	税率	FirstValue
74	犹他州销售税	5.00	74
36	明尼苏达州销售税	6.75	74
30	马萨诸塞州销售税	7.00	74

```
SELECT BusinessEntityID, JobTitle, SickLeaveHours,
    CUME_DIST() OVER(PARTITION BY JobTitle ORDER BY SickLeaveHours ASC)
    AS "Cumulative Distribution",
    PERCENTILE_DISC(0.5) WITHIN GROUP(ORDER BY SickLeaveHours)
    OVER(PARTITION BY JobTitle) AS "Percentile Discreet"
FROM Employee;
```

To find the exact value from the row that matches or exceeds the 0.5 percentile, you pass the percentile as the numeric literal in the PERCENTILE_DISC function. The Percentile Discreet column in a result set lists the value of the row at which the cumulative distribution is higher than the specified percentile.

BusinessEntityID	JobTitle	SickLeaveHours	Cumulative Distribution	Percentile Discreet
272	Application Specialist 55		0.25	56
268	Application Specialist 56		0.75	56
269	Application Specialist 56		0.75	56
267	Application Specialist 57		1	56

To base the calculation on a set of values, you use the PERCENTILE_CONT function. The "Percentile Continuous" column in the results lists the average value of the sum of the result value and the next highest matching value.

```
SELECT BusinessEntityID, JobTitle, SickLeaveHours,
    CUME_DIST() OVER(PARTITION BY JobTitle ORDER BY SickLeaveHours ASC)
    AS "Cumulative Distribution",
    PERCENTILE_DISC(0.5) WITHIN GROUP(ORDER BY SickLeaveHours)
    OVER(PARTITION BY JobTitle) AS "Percentile Discreet",
    PERCENTILE_CONT(0.5) WITHIN GROUP(ORDER BY SickLeaveHours)
    OVER(PARTITION BY JobTitle) AS "Percentile Continuous"
FROM Employee;
```

BusinessEntityID	JobTitle	SickLeaveHours	Cumulative Distribution	Percentile Discreet	Percentile Continuous
272	Application Specialist 55		0.25	56	56
268	Application Specialist 56		0.75	56	56
269	Application Specialist 56		0.75	56	56
267	Application Specialist 57		1	56	56

Section 44.3: FIRST_VALUE

You use the FIRST_VALUE function to determine the first value in an ordered result set, which you identify using a scalar expression.

```
SELECT StateProvinceID, Name, TaxRate,
    FIRST_VALUE(StateProvinceID)
    OVER(ORDER BY TaxRate ASC) AS FirstValue
FROM SalesTaxRate;
```

In this example, the FIRST_VALUE function is used to return the ID of the state or province with the lowest tax rate. The OVER clause is used to order the tax rates to obtain the lowest rate.

StateProvinceID	Name	TaxRate	FirstValue
74	Utah State Sales Tax	5.00	74
36	Minnesota State Sales Tax	6.75	74
30	Massachusetts State Sales Tax	7.00	74

1	加拿大商品及服务税 (GST)	7.00	74
57	加拿大商品及服务税 (GST)	7.00	74
63	加拿大商品及服务税 (GST)	7.00	74

第44.4节：LAST_VALUE

LAST_VALUE 函数提供有序结果集中最后一个值，该值由您使用标量表达式指定。

```
SELECT TerritoryID, StartDate, BusinessentityID,
    LAST_VALUE(BusinessentityID)
    OVER(ORDER BY TerritoryID) AS LastValue
FROM SalesTerritoryHistory;
```

此示例使用LAST_VALUE函数返回有序值中每个行集的最后一个值。

TerritoryID	StartDate	BusinessentityID	LastValue
1	2005-07-01 00.00.00.000	280	283
1	2006-11-01 00.00.00.000	284	283
1	2005-07-01 00.00.00.000	283	283
2	2007-01-01 00.00.00.000	277	275
2	2005-07-01 00.00.00.000	275	275
3	2007-01-01 00.00.00.000	275	277

第44.5节：PERCENT_RANK 和 CUME_DIST

PERCENT_RANK 函数计算某行相对于行集合的排名。百分比基于组中比当前行值更低的行数。

结果集中第一个值的百分比排名始终为零。集合中排名最高的-或最后一个-值的百分比排名始终为一。

CUME_DIST 函数通过确定小于或等于指定值的值的百分比，计算该值在一组值中的相对位置。这称为累积分布。

```
SELECT BusinessEntityID, JobTitle, SickLeaveHours,
    PERCENT_RANK() OVER(PARTITION BYJobTitle ORDER BYSickLeaveHours DESC)
    AS "Percent Rank",
    CUME_DIST() OVER(PARTITION BYJobTitle ORDER BYSickLeaveHours DESC)
    AS "Cumulative Distribution"
FROM Employee;
```

在此示例中，您使用了一个ORDER子句来对SELECT语句检索的行进行分区——或分组——基于员工的职位，且每个组内的结果根据员工使用的病假小时数进行排序。

BusinessEntityID	职位名称	病假小时数	百分等级	累积分布
267	应用专家	57	0	0.25
268	应用专家	56	0.3333333333333333	0.75
269	应用专家	56	0.3333333333333333	0.75
272	应用专家	55	1	1

1	Canadian GST	7.00	74
57	Canadian GST	7.00	74
63	Canadian GST	7.00	74

Section 4 4.4: LAST_VALUE

The LAST_VALUE function provides the last value in an ordered result set, which you specify using a scalar expression.

```
SELECT TerritoryID, StartDate, BusinessentityID,
    LAST_VALUE(BusinessentityID)
    OVER(ORDER BY TerritoryID) AS LastValue
FROM SalesTerritoryHistory;
```

This example uses the LAST_VALUE function to return the last value for each rowset in the ordered values.

TerritoryID	StartDate	BusinessentityID	LastValue
1	2005-07-01 00.00.00.000	280	283
1	2006-11-01 00.00.00.000	284	283
1	2005-07-01 00.00.00.000	283	283
2	2007-01-01 00.00.00.000	277	275
2	2005-07-01 00.00.00.000	275	275
3	2007-01-01 00.00.00.000	275	277

Section 4 4.5: PERCENT_RANK and CUME_DIST

The PERCENT_RANK function calculates the ranking of a row relative to the row set. The percentage is based on the number of rows in the group that have a lower value than the current row.

The first value in the result set always has a percent rank of zero. The value for the highest-ranked – or last – value in the set is always one.

The CUME_DIST function calculates the relative position of a specified value in a group of values, by determining the percentage of values less than or equal to that value. This is called the cumulative distribution.

```
SELECT BusinessEntityID, JobTitle, SickLeaveHours,
    PERCENT_RANK() OVER(PARTITION BY JobTitle ORDER BY SickLeaveHours DESC)
    AS "Percent Rank",
    CUME_DIST() OVER(PARTITION BY JobTitle ORDER BY SickLeaveHours DESC)
    AS "Cumulative Distribution"
FROM Employee;
```

In this example, you use an ORDER clause to partition – or group – the rows retrieved by the SELECT statement based on employees' job titles, with the results in each group sorted based on the numbers of sick leave hours that employees have used.

BusinessEntityID	JobTitle	SickLeaveHours	Percent Rank	Cumulative Distribution
267	Application Specialist	57	0	0.25
268	Application Specialist	56	0.3333333333333333	0.75
269	Application Specialist	56	0.3333333333333333	0.75
272	Application Specialist	55	1	1

262	首席财务官助理	48	0	1
239	福利专员	45	0	1
252	采购员	50	0	0.111111111111111
251	采购员	49	0.125	0.333333333333333
256	采购员	49	0.125	0.333333333333333
253	采购员	48	0.375	0.555555555555555
254	采购员	48	0.375	0.555555555555555

函数PERCENT_RANK对每个组内的条目进行排名。对于每个条目，它返回同一组中值较低的条目的百分比。

函数CUME_DIST类似，但它返回小于或等于当前值的百分比。

262	Assitant to the Cheif Financial Officer	48	0	1
239	Benefits Specialist	45	0	1
252	Buyer	50	0	0.111111111111111
251	Buyer	49	0.125	0.333333333333333
256	Buyer	49	0.125	0.333333333333333
253	Buyer	48	0.375	0.555555555555555
254	Buyer	48	0.375	0.555555555555555

The **PERCENT_RANK** function ranks the entries within each group. For each entry, it returns the percentage of entries in the same group that have lower values.

The **CUME_DIST** function is similar, except that it returns the percentage of values less than or equal to the current value.

第45章：窗口函数

第45.1节：如果其他行具有共同属性，则设置标志

假设我有以下数据：

表 items

id	name	tag
1	示例	unique_tag
2	foo	简单
42	bar	简单
3	baz	你好
51	quux	世界

我想获取所有这些行，并知道某个标签是否被其他行使用

```
SELECT id, name, tag, COUNT(*) OVER (PARTITION BY tag) > 1 AS flag FROM items
```

结果将是：

id	name	tag	flag
1	示例	unique_tag	false
2	foo	简单	true
42	bar	简单	true
3	baz	你好	false
51	quux	世界	false

如果您的数据库不支持 OVER 和 PARTITION，可以使用以下方法获得相同结果：

```
SELECT id, name, tag, (SELECT COUNT(tag) FROM items B WHERE tag = A.tag) > 1 AS flag FROM items A
```

第45.2节：使用

LAG()函数查找“序列错乱”记录

给定以下示例数据：

ID	状态	状态时间	状态由
1	一	2016-09-28-19.47.52.501398	用户_1
3	一	2016-09-28-19.47.52.501511	用户_2
1	三	2016-09-28-19.47.52.501517	用户_3
3	二	2016-09-28-19.47.52.501521	用户_2
3	三	2016-09-28-19.47.52.501524	用户_4

由ID值标识的项目必须按顺序从状态“一”移动到“二”，再到“三”，不能跳过状态。问题是找出违反规则、从“一”直接跳到“三”的用户（状态由）值。

LAG()分析函数通过返回每行前一行的值来帮助解决问题：

Chapter 45: Window Functions

Section 45.1: Setting up a flag if other rows have a common property

Let's say I have this data:

Table items

id	name	tag
1	example	unique_tag
2	foo	simple
42	bar	simple
3	baz	hello
51	quux	world

I'd like to get all those lines and know if a tag is used by other lines

```
SELECT id, name, tag, COUNT(*) OVER (PARTITION BY tag) > 1 AS flag FROM items
```

The result will be:

id	name	tag	flag
1	example	unique_tag	false
2	foo	simple	true
42	bar	simple	true
3	baz	hello	false
51	quux	world	false

In case your database doesn't have OVER and PARTITION you can use this to produce the same result:

```
SELECT id, name, tag, (SELECT COUNT(tag) FROM items B WHERE tag = A.tag) > 1 AS flag FROM items A
```

Section 45.2: Finding "out-of-sequence" records using the LAG() function

Given these sample data:

ID	STATUS	STATUS_TIME	STATUS_BY
1	ONE	2016-09-28-19.47.52.501398	USER_1
3	ONE	2016-09-28-19.47.52.501511	USER_2
1	THREE	2016-09-28-19.47.52.501517	USER_3
3	TWO	2016-09-28-19.47.52.501521	USER_2
3	THREE	2016-09-28-19.47.52.501524	USER_4

Items identified by ID values must move from STATUS 'ONE' to 'TWO' to 'THREE' in sequence, without skipping statuses. The problem is to find users (STATUS_BY) values who violate the rule and move from 'ONE' immediately to 'THREE'.

The LAG() analytical function helps to solve the problem by returning for each row the value in the preceding row:

```
SELECT * FROM (
  SELECT
t.*,
  LAG(status) OVER (PARTITION BY id ORDER BY status_time) AS prev_status
FROM test t
) t1 WHERE status = 'THREE' AND prev_status != 'TWO'
```

如果您的数据库没有 LAG(), 您可以使用以下方法获得相同的结果：

```
SELECT A.id, A.status, B.status as prev_status, A.status_time, B.status_time as prev_status_time
FROM Data A, Data B
WHERE A.id = B.id
AND B.status_time = (SELECT MAX(status_time) FROM Data where status_time < A.status_time and id =
A.id)
AND A.status = 'THREE' AND NOT B.status = 'TWO'
```

第45.3节：获取累计总数

给定以下数据：

date	金额
2016-03-12	200
2016-03-11	-50
2016-03-14	100
2016-03-15	100
2016-03-10	-250

```
SELECT date, amount, SUM(amount) OVER (ORDER BY date ASC) AS running
FROM operations
ORDER BY date ASC
```

将会得到

date	金额	运行总计
2016-03-10	-250	-250
2016-03-11	-50	-300
2016-03-12	200	-100
2016-03-14	100	0
2016-03-15	100	-100

第45.4节：将选中的总行数添加到每一行

```
SELECT your_columns, COUNT(*) OVER() as Ttl_Rows FROM your_data_set
```

id name Ttl_Rows

1	示例5	
2	foo	5
3	bar	5
4	baz	5
5	quux	5

与其使用两个查询先获取计数再获取行，不如使用聚合函数作为窗口函数，并将完整结果集用作窗口。

```
SELECT * FROM (
  SELECT
t.*,
  LAG(status) OVER (PARTITION BY id ORDER BY status_time) AS prev_status
FROM test t
) t1 WHERE status = 'THREE' AND prev_status != 'TWO'
```

In case your database doesn't have LAG() you can use this to produce the same result:

```
SELECT A.id, A.status, B.status as prev_status, A.status_time, B.status_time as prev_status_time
FROM Data A, Data B
WHERE A.id = B.id
AND B.status_time = (SELECT MAX(status_time) FROM Data where status_time < A.status_time and id =
A.id)
AND A.status = 'THREE' AND NOT B.status = 'TWO'
```

Section 45.3: Getting a running total

Given this data:

date	amount
2016-03-12	200
2016-03-11	-50
2016-03-14	100
2016-03-15	100
2016-03-10	-250

```
SELECT date, amount, SUM(amount) OVER (ORDER BY date ASC) AS running
FROM operations
ORDER BY date ASC
```

will give you

date	amount	running
2016-03-10	-250	-250
2016-03-11	-50	-300
2016-03-12	200	-100
2016-03-14	100	0
2016-03-15	100	-100

Section 45.4: Adding the total rows selected to every row

```
SELECT your_columns, COUNT(*) OVER() as Ttl_Rows FROM your_data_set
```

id name Ttl_Rows

1	example 5	
2	foo	5
3	bar	5
4	baz	5
5	quux	5

Instead of using two queries to get a count then the line, you can use an aggregate as a window function and use the full result set as the window.

这可以作为进一步计算的基础，避免额外自连接的复杂性。

第45.5节：获取多个分组中最近的N行

给定以下数据

用户ID	完成日期
1	2016-07-20
1	2016-07-21
2	2016-07-20
2	2016-07-21
2	2016-07-22

```
;使用CTE作为
(选择*,
    ROW_NUMBER() OVER (PARTITION BY User_ID
                        ORDER BY Completion_Date DESC) Row_Num
FROM    Data)
SELECT * FROM CTE WHERE Row_Num <= n
```

使用 n=1，您将获得每个user_id的最新一行：

User_ID	Completion_Date	Row_Num
1	2016-07-21	1
2	2016-07-22	1

This can be used as a base for further calculation without the complexity of extra self joins.

Section 45.5: Getting the N most recent rows over multiple grouping

Given this data

User_ID	Completion_Date
1	2016-07-20
1	2016-07-21
2	2016-07-20
2	2016-07-21
2	2016-07-22

```
;with CTE as
(SELECT *,
    ROW_NUMBER() OVER (PARTITION BY User_ID
                        ORDER BY Completion_Date DESC) Row_Num
FROM    Data)
SELECT * FROM CTE WHERE Row_Num <= n
```

Using n=1, you'll get the one most recent row per user_id:

User_ID	Completion_Date	Row_Num
1	2016-07-21	1
2	2016-07-22	1

第46章：公共表表达式

第46.1节：生成数值

大多数数据库没有原生方法生成用于临时使用的数字序列；然而，公共表表达式可以结合递归来模拟此类功能。

下面的示例生成了一个名为Numbers的公共表表达式，包含一列 i，行数为数字1-5：

```
--给表命名为`Numbers`，并创建一列`i`来存放数字
WITH Numbers(i) AS (
  --起始数字/索引
  SELECT 1
  --递归需要顶层的UNION ALL操作符
  UNION ALL
  --迭代表达式：
  SELECT i + 1
  --我们首先声明的表表达式，作为递归的源
  FROM Numbers
  --定义递归结束的条件
  WHERE i < 5
)
--像使用普通表一样使用生成的表表达式
SELECT i FROM Numbers;
```

GoalKicker.com - MySQL Notes for Professionals

Chapter 46: Common Table Expressions

Section 46.1: generating values

Most databases do not have a native way of generating a series of numbers for ad-hoc use; however, common table expressions can be used with recursion to emulate that type of function.

The following example generates a common table expression called Numbers with a column i which has a row for numbers 1-5:

```
--Give a table name `Numbers` and a column `i` to hold the numbers
WITH Numbers(i) AS (
  --Starting number/index
  SELECT 1
  --Top-level UNION ALL operator required for recursion
  UNION ALL
  --Iteration expression:
  SELECT i + 1
  --Table expression we first declared used as source for recursion
  FROM Numbers
  --Clause to define the end of the recursion
  WHERE i < 5
)
--Use the generated table expression like a regular table
SELECT i FROM Numbers;
```

i
1
2
3
4
5

This method can be used with any number interval, as well as other types of data.

Section 46.2: recursively enumerating a subtree

```
WITH RECURSIVE ManagedByJames(Level, ID, FName, LName) AS (
  -- start with this row
  SELECT 1, ID, FName, LName
  FROM Employees
  WHERE ID = 1

  UNION ALL

  -- get employees that have any of the previously selected rows as manager
  SELECT ManagedByJames.Level + 1,
         Employees.ID,
         Employees.FName,
         Employees.LName
  FROM Employees
  JOIN ManagedByJames
    ON Employees.ManagerID = ManagedByJames.ID

  ORDER BY 1 DESC -- depth-first search
)
SELECT * FROM ManagedByJames;
```

Level	ID	FName	LName
1	1	詹姆斯	史密斯
2	2	约翰	约翰逊
3	4	约翰纳森·史密斯	
2	3	迈克尔	威廉姆斯

第46.3节：临时查询

这些行为方式与嵌套子查询相同，但语法不同。

```
WITH ReadyCars AS (  
  SELECT *  
    从汽车  
  WHERE 状态 = 'READY'  
)  
SELECT ID, 型号, 总成本  
FROM ReadyCars  
ORDER BY 总成本;  
  
ID型号总成本  
1 福特 F-150 200  
2 福特 F-150 230
```

等效的子查询语法

```
SELECT ID, 型号, 总成本  
FROM (  
  SELECT *  
    从汽车  
  WHERE Status = 'READY'  
) AS ReadyCars  
ORDER BY TotalCost
```

第46.4节：递归向上遍历树结构

```
WITH RECURSIVE Jonathon的经理们 AS (  
  -- 从这一行开始  
  SELECT *  
    FROM 员工  
  WHERE ID = 4  
  
  UNION ALL  
  
  -- 获取所有之前选中行的经理  
  SELECT 员工.*  
    FROM 员工  
  JOIN Jonathon的经理们  
    ON 员工.ID = Jonathon的经理们.ManagerID  
)  
SELECT * FROM Jonathon的经理们;  
  
编号名字姓氏电话号码经理编号部门编号  
4 约纳森·史密斯 1212121212 2 1  
2 约翰 约翰逊 2468101214 1 1  
1 詹姆斯 史密斯 1234567890 空 1
```

Level	ID	FName	LName
1	1	James	Smith
2	2	John	Johnson
3	4	Johnathon Smith	
2	3	Michael	Williams

Section 46.3: Temporary query

These behave in the same manner as nested subqueries but with a different syntax.

```
WITH ReadyCars AS (  
  SELECT *  
    FROM Cars  
  WHERE Status = 'READY'  
)  
SELECT ID, Model, TotalCost  
FROM ReadyCars  
ORDER BY TotalCost;  
  
ID Model TotalCost  
1 Ford F-150 200  
2 Ford F-150 230
```

Equivalent subquery syntax

```
SELECT ID, Model, TotalCost  
FROM (  
  SELECT *  
    FROM Cars  
  WHERE Status = 'READY'  
) AS ReadyCars  
ORDER BY TotalCost
```

Section 46.4: recursively going up in a tree

```
WITH RECURSIVE ManagersOfJonathon AS (  
  -- start with this row  
  SELECT *  
    FROM Employees  
  WHERE ID = 4  
  
  UNION ALL  
  
  -- get manager(s) of all previously selected rows  
  SELECT Employees.*  
    FROM Employees  
  JOIN ManagersOfJonathon  
    ON Employees.ID = ManagersOfJonathon.ManagerID  
)  
SELECT * FROM ManagersOfJonathon;  
  
Id FName LName PhoneNumber ManagerId DepartmentId  
4 Johnathon Smith 1212121212 2 1  
2 John Johnson 2468101214 1 1  
1 James Smith 1234567890 NULL 1
```

第46.5节：递归生成日期，扩展为包含团队排班示例

```
DECLARE @DateFrom DATETIME = '2016-06-01 06:00'
DECLARE @DateTo DATETIME = '2016-07-01 06:00'
DECLARE @IntervalDays INT = 7

-- 转换序列 = 休息放松到白班再到夜班
-- RR (休息放松) = 1
-- DS (白班) = 2
-- NS (夜班) = 3

;WITH roster AS
(
    SELECT @DateFrom AS 排班开始, 1 AS 队伍A, 2 AS 队伍B, 3 AS 队伍C
    UNION ALL
    SELECT DATEADD(d, @IntervalDays, 排班开始),
           CASE 队伍A WHEN 1 THEN 2 WHEN 2 THEN 3 WHEN 3 THEN 1 END AS 队伍A,
           CASE 队伍B WHEN 1 THEN 2 WHEN 2 THEN 3 WHEN 3 THEN 1 END AS 队伍B,
           CASE 队伍C WHEN 1 THEN 2 WHEN 2 THEN 3 WHEN 3 THEN 1 END AS 队伍C
    FROM roster WHERE 排班开始 < DATEADD(d, -@IntervalDays, @DateTo)
)

SELECT 排班开始,
       ISNULL(LEAD(排班开始) OVER (ORDER BY 排班开始), 排班开始 + @IntervalDays) AS
排班结束,
       CASE TeamA WHEN 1 THEN 'RR' WHEN 2 THEN 'DS' WHEN 3 THEN 'NS' END AS TeamA,
       CASE TeamB WHEN 1 THEN 'RR' WHEN 2 THEN 'DS' WHEN 3 THEN 'NS' END AS TeamB,
       CASE TeamC WHEN 1 THEN 'RR' WHEN 2 THEN 'DS' WHEN 3 THEN 'NS' END AS TeamC
FROM roster
```

结果

即，第1周TeamA休息，TeamB上白班，TeamC上夜班。

	RosterStart	RosterEnd	TeamA	TeamB	TeamC
1	2016-06-01 06:00:00.000	2016-06-08 06:00:00.000	RR	DS	NS
2	2016-06-08 06:00:00.000	2016-06-15 06:00:00.000	DS	NS	RR
3	2016-06-15 06:00:00.000	2016-06-22 06:00:00.000	NS	RR	DS
4	2016-06-22 06:00:00.000	2016-06-29 06:00:00.000	RR	DS	NS
5	2016-06-29 06:00:00.000	2016-07-06 06:00:00.000	DS	NS	RR

第46.6节：Oracle CONNECT BY功能与递归CTE

Oracle的CONNECT BY功能提供了许多有用且复杂的特性，而这些特性在使用SQL标准递归CTE时并未内置。此示例使用SQL Server语法复制了这些特性（为完整性添加了一些内容）。这对Oracle开发者在其他数据库中发现层级查询缺少许多功能时非常有用，同时也展示了层级查询的一般应用。

```
WITH tbl AS (
    SELECT id, name, parent_id
    FROM mytable)
, tbl_hierarchy AS (
    /* 锚点 */
```

Section 46.5: Recursively generate dates, extended to include team rostering as example

```
DECLARE @DateFrom DATETIME = '2016-06-01 06:00'
DECLARE @DateTo DATETIME = '2016-07-01 06:00'
DECLARE @IntervalDays INT = 7

-- Transition Sequence = Rest & Relax into Day Shift into Night Shift
-- RR (Rest & Relax) = 1
-- DS (Day Shift) = 2
-- NS (Night Shift) = 3

;WITH roster AS
(
    SELECT @DateFrom AS RosterStart, 1 AS TeamA, 2 AS TeamB, 3 AS TeamC
    UNION ALL
    SELECT DATEADD(d, @IntervalDays, RosterStart),
           CASE TeamA WHEN 1 THEN 2 WHEN 2 THEN 3 WHEN 3 THEN 1 END AS TeamA,
           CASE TeamB WHEN 1 THEN 2 WHEN 2 THEN 3 WHEN 3 THEN 1 END AS TeamB,
           CASE TeamC WHEN 1 THEN 2 WHEN 2 THEN 3 WHEN 3 THEN 1 END AS TeamC
    FROM roster WHERE RosterStart < DATEADD(d, -@IntervalDays, @DateTo)
)

SELECT RosterStart,
       ISNULL(LEAD(RosterStart) OVER (ORDER BY RosterStart), RosterStart + @IntervalDays) AS
RosterEnd,
       CASE TeamA WHEN 1 THEN 'RR' WHEN 2 THEN 'DS' WHEN 3 THEN 'NS' END AS TeamA,
       CASE TeamB WHEN 1 THEN 'RR' WHEN 2 THEN 'DS' WHEN 3 THEN 'NS' END AS TeamB,
       CASE TeamC WHEN 1 THEN 'RR' WHEN 2 THEN 'DS' WHEN 3 THEN 'NS' END AS TeamC
FROM roster
```

Result

I.e. For Week 1 TeamA is on R&R, TeamB is on Day Shift and TeamC is on Night Shift.

	RosterStart	RosterEnd	TeamA	TeamB	TeamC
1	2016-06-01 06:00:00.000	2016-06-08 06:00:00.000	RR	DS	NS
2	2016-06-08 06:00:00.000	2016-06-15 06:00:00.000	DS	NS	RR
3	2016-06-15 06:00:00.000	2016-06-22 06:00:00.000	NS	RR	DS
4	2016-06-22 06:00:00.000	2016-06-29 06:00:00.000	RR	DS	NS
5	2016-06-29 06:00:00.000	2016-07-06 06:00:00.000	DS	NS	RR

Section 46.6: Oracle CONNECT BY functionality with recursive CTEs

Oracle's CONNECT BY functionality provides many useful and nontrivial features that are not built-in when using SQL standard recursive CTEs. This example replicates these features (with a few additions for sake of completeness), using SQL Server syntax. It is most useful for Oracle developers finding many features missing in their hierarchical queries on other databases, but it also serves to showcase what can be done with a hierarchical query in general.

```
WITH tbl AS (
    SELECT id, name, parent_id
    FROM mytable)
, tbl_hierarchy AS (
    /* Anchor */
```

```

SELECT 1 AS "LEVEL"
      --, 1 AS CONNECT_BY_ISROOT
      --, 0 AS CONNECT_BY_ISBRANCH
      , CASE WHEN t.id IN (SELECT parent_id FROM tbl) THEN 0 ELSE 1 END AS CONNECT_BY_ISLEAF
      , 0 AS CONNECT_BY_ISCYCLE
      , '/' + CAST(t.id AS VARCHAR(MAX)) + '/' AS SYS_CONNECT_BY_PATH_id
      , '/' + CAST(t.name AS VARCHAR(MAX)) + '/' AS SYS_CONNECT_BY_PATH_name
      , t.id AS root_id
      , t.*
FROM tbl t
WHERE t.parent_id IS NULL -- START WITH parent_id IS NULL
UNION ALL
/* 递归 */
SELECT th."LEVEL" + 1 AS "LEVEL"
      --, 0 AS CONNECT_BY_ISROOT
      --, CASE WHEN t.id IN (SELECT parent_id FROM tbl) THEN 1 ELSE 0 END AS
CONNECT_BY_ISBRANCH
      , CASE WHEN t.id IN (SELECT parent_id FROM tbl) THEN 0 ELSE 1 END AS CONNECT_BY_ISLEAF
      , CASE WHEN th.SYS_CONNECT_BY_PATH_id LIKE '%/' + CAST(t.id AS VARCHAR(MAX)) + '/' THEN 1 ELSE 0 END AS CONNECT_BY_ISCYCLE
      , th.SYS_CONNECT_BY_PATH_id + CAST(t.id AS VARCHAR(MAX)) + '/' AS
SYS_CONNECT_BY_PATH_id
      , th.SYS_CONNECT_BY_PATH_name + CAST(t.name AS VARCHAR(MAX)) + '/' AS
SYS_CONNECT_BY_PATH_name
      , th.root_id
      , t.*
FROM tbl t
      JOIN tbl_hierarchy th ON (th.id = t.parent_id) -- CONNECT BY PRIOR id = parent_id
WHERE th.CONNECT_BY_ISCYCLE = 0) -- NOCYCLE
SELECT th.*
      --, REPLICATE(' ', (th."LEVEL" - 1) * 3) + th.name AS tbl_hierarchy
FROM tbl_hierarchy th
      JOIN tbl CONNECT_BY_ROOT ON (CONNECT_BY_ROOT.id = th.root_id)
ORDER BY th.SYS_CONNECT_BY_PATH_name; -- ORDER SIBLINGS BY name

```

CONNECT BY 上述功能，附带说明：

- 子句
 - CONNECT BY：指定定义层级关系的条件。
 - START WITH：指定根节点。
 - 按顺序排列兄弟节点：正确排序结果。
- 参数
 - NOCYCLE：当检测到循环时停止处理分支。有效的层次结构是有向无环图，循环引用违反此结构。
- 运算符
 - PRIOR：获取节点父节点的数据。
 - CONNECT_BY_ROOT：获取节点根节点的数据。
- 伪列
 - LEVEL：表示节点与其根节点的距离。
 - CONNECT_BY_ISLEAF：表示无子节点的节点。
 - CONNECT_BY_ISCYCLE：表示存在循环引用的节点。
- 功能
 - SYS_CONNECT_BY_PATH：返回从根节点到该节点路径的扁平化/连接表示。

```

SELECT 1 AS "LEVEL"
      --, 1 AS CONNECT_BY_ISROOT
      --, 0 AS CONNECT_BY_ISBRANCH
      , CASE WHEN t.id IN (SELECT parent_id FROM tbl) THEN 0 ELSE 1 END AS CONNECT_BY_ISLEAF
      , 0 AS CONNECT_BY_ISCYCLE
      , '/' + CAST(t.id AS VARCHAR(MAX)) + '/' AS SYS_CONNECT_BY_PATH_id
      , '/' + CAST(t.name AS VARCHAR(MAX)) + '/' AS SYS_CONNECT_BY_PATH_name
      , t.id AS root_id
      , t.*
FROM tbl t
WHERE t.parent_id IS NULL -- START WITH parent_id IS NULL
UNION ALL
/* Recursive */
SELECT th."LEVEL" + 1 AS "LEVEL"
      --, 0 AS CONNECT_BY_ISROOT
      --, CASE WHEN t.id IN (SELECT parent_id FROM tbl) THEN 1 ELSE 0 END AS
CONNECT_BY_ISBRANCH
      , CASE WHEN t.id IN (SELECT parent_id FROM tbl) THEN 0 ELSE 1 END AS CONNECT_BY_ISLEAF
      , CASE WHEN th.SYS_CONNECT_BY_PATH_id LIKE '%/' + CAST(t.id AS VARCHAR(MAX)) + '/' THEN 1 ELSE 0 END AS CONNECT_BY_ISCYCLE
      , th.SYS_CONNECT_BY_PATH_id + CAST(t.id AS VARCHAR(MAX)) + '/' AS
SYS_CONNECT_BY_PATH_id
      , th.SYS_CONNECT_BY_PATH_name + CAST(t.name AS VARCHAR(MAX)) + '/' AS
SYS_CONNECT_BY_PATH_name
      , th.root_id
      , t.*
FROM tbl t
      JOIN tbl_hierarchy th ON (th.id = t.parent_id) -- CONNECT BY PRIOR id = parent_id
WHERE th.CONNECT_BY_ISCYCLE = 0) -- NOCYCLE
SELECT th.*
      --, REPLICATE(' ', (th."LEVEL" - 1) * 3) + th.name AS tbl_hierarchy
FROM tbl_hierarchy th
      JOIN tbl CONNECT_BY_ROOT ON (CONNECT_BY_ROOT.id = th.root_id)
ORDER BY th.SYS_CONNECT_BY_PATH_name; -- ORDER SIBLINGS BY name

```

CONNECT BY features demonstrated above, with explanations:

- Clauses
 - CONNECT BY: Specifies the relationship that defines the hierarchy.
 - START WITH: Specifies the root nodes.
 - ORDER SIBLINGS BY: Orders results properly.
- Parameters
 - NOCYCLE: Stops processing a branch when a loop is detected. Valid hierarchies are Directed Acyclic Graphs, and circular references violate this construct.
- Operators
 - PRIOR: Obtains data from the node's parent.
 - CONNECT_BY_ROOT: Obtains data from the node's root.
- Pseudocolumns
 - LEVEL: Indicates the node's distance from its root.
 - CONNECT_BY_ISLEAF: Indicates a node without children.
 - CONNECT_BY_ISCYCLE: Indicates a node with a circular reference.
- Functions
 - SYS_CONNECT_BY_PATH: Returns a flattened/concatenated representation of the path to the node from its root.

第47章：视图

第47.1节：简单视图

视图可以从基表中过滤某些行，或仅投影其中的某些列：

```
CREATE VIEW new_employees_details AS
SELECT E.id, Fname, Salary, Hire_date
FROM Employees E
WHERE hire_date > date '2015-01-01';
```

如果从视图中查询：

```
select * from new_employees_details

Id FName Salary Hire_date
4 Johnathon 500 24-07-2016
```

第47.2节：复杂视图

视图可以是非常复杂的查询（聚合、连接、子查询等）。只要确保为你选择的每一项添加列名即可：

```
创建视图 dept_income 为
选择 d.Name 作为 DepartmentName, 求和(e.salary) 作为 TotalSalary
来自 Employees e
连接 Departments d 在 e.DepartmentId = d.id
按 d.Name分组;
```

现在你可以像查询任何表一样从它查询：

```
SELECT *
来自 dept_income;

DepartmentName TotalSalary
人力资源部 1900
销售部 600
```

Chapter 47: Views

Section 47.1: Simple views

A view can filter some rows from the base table or project only some columns from it:

```
CREATE VIEW new_employees_details AS
SELECT E.id, Fname, Salary, Hire_date
FROM Employees E
WHERE hire_date > date '2015-01-01';
```

If you select form the view:

```
select * from new_employees_details

Id FName Salary Hire_date
4 Johnathon 500 24-07-2016
```

Section 47.2: Complex views

A view can be a really complex query(aggregations, joins, subqueries, etc). Just be sure you add column names for everything you select:

```
Create VIEW dept_income AS
SELECT d.Name as DepartmentName, sum(e.salary) as TotalSalary
FROM Employees e
JOIN Departments d on e.DepartmentId = d.id
GROUP BY d.Name;
```

Now you can select from it as from any table:

```
SELECT *
FROM dept_income;

DepartmentName TotalSalary
HR 1900
Sales 600
```

第48章：物化视图

物化视图是一种其结果被物理存储且必须定期刷新以保持最新的视图。因此，当不需要实时结果时，它们对于存储复杂、长时间运行查询的结果非常有用。物化视图可以在Oracle和PostgreSQL中创建。其他数据库系统也提供类似功能，例如SQL Server的索引视图或DB2的物化查询表。

第48.1节：PostgreSQL示例

```
CREATE TABLE mytable (number INT);
INSERT INTO mytable VALUES (1);

CREATE MATERIALIZED VIEW myview AS SELECT * FROM mytable;

SELECT * FROM myview;
```

```
number
-----
1
(1 行)
```

```
INSERT INTO mytable VALUES(2);

SELECT * FROM myview;
```

```
number
-----
1
(1 行)
```

```
刷新物化视图 myview;

SELECT * FROM myview;
```

```
number
-----
1
2
(2 行)
```

Chapter 48: Materialized Views

A materialized view is a view whose results are physically stored and must be periodically refreshed in order to remain current. They are therefore useful for storing the results of complex, long-running queries when realtime results are not required. Materialized views can be created in Oracle and PostgreSQL. Other database systems offer similar functionality, such as SQL Server's indexed views or DB2's materialized query tables.

Section 48.1: PostgreSQL example

```
CREATE TABLE mytable (number INT);
INSERT INTO mytable VALUES (1);

CREATE MATERIALIZED VIEW myview AS SELECT * FROM mytable;

SELECT * FROM myview;
```

```
number
-----
1
(1 row)
```

```
INSERT INTO mytable VALUES(2);

SELECT * FROM myview;
```

```
number
-----
1
(1 row)
```

```
REFRESH MATERIALIZED VIEW myview;

SELECT * FROM myview;
```

```
number
-----
1
2
(2 rows)
```


第49章：注释

第49.1节：单行注释

单行注释以--开头，直到行尾：

```
SELECT *  
FROM 员工 -- 这是一个注释  
WHERE 名字 = 'John'
```

第49.2节：多行注释

多行代码注释用/* ... */包裹：

```
/* 该查询  
返回所有员工 */  
SELECT *  
FROM 员工
```

也可以将这样的注释插入到一行的中间：

```
SELECT /* 所有列：*/ *  
FROM 员工
```

Chapter 49: Comments

Section 49.1: Single-line comments

Single line comments are preceded by --, and go until the end of the line:

```
SELECT *  
FROM Employees -- this is a comment  
WHERE FName = 'John'
```

Section 49.2: Multi-line comments

Multi-line code comments are wrapped in /* ... */:

```
/* This query  
returns all employees */  
SELECT *  
FROM Employees
```

It is also possible to insert such a comment into the middle of a line:

```
SELECT /* all columns: */ *  
FROM Employees
```

第50章：外键

第50.1节：外键说明

外键约束通过强制一个表中的值必须匹配另一个表中的值，来确保数据完整性。

外键必需的一个例子是：在大学中，一门课程必须属于一个系。该场景的代码是：

```
创建表 Department (  
    Dept_Code      CHAR (5)      主键,  
    Dept_Name      VARCHAR (20)  唯一  
);
```

使用以下语句插入值：

```
INSERT INTO Department VALUES ('CS205', '计算机科学');
```

下表将包含计算机科学分支提供的课程信息：

```
创建表 Programming_Courses (  
    Dept_Code      CHAR(5),  
    Prg_Code       CHAR(9) 主键,  
    Prg_Name       VARCHAR (50) 唯一,  
    外键 (Dept_Code) 引用 Department(Dept_Code)  
);
```

（外键的数据类型必须与被引用键的数据类型匹配。）

列Dept_Code上的外键约束只允许值存在于被引用的表中，Department。这意味着如果你尝试插入以下值：

```
INSERT INTO Programming_Courses Values ('CS300', 'FDB-DB001', '数据库系统');
```

数据库将引发外键违反错误，因为CS300不存在于Department表中。但当你尝试一个存在的键值时：

```
INSERT INTO Programming_Courses VALUES ('CS205', 'FDB-DB001', '数据库系统');  
INSERT INTO Programming_Courses VALUES ('CS205', 'DB2-DB002', '数据库系统 II');
```

那么数据库允许这些值。

使用外键的一些提示

- 外键必须引用父表中的唯一键（或主键）。
- 在外键列中输入NULL值不会引发错误。
- 外键约束可以引用同一数据库中的表。
- 外键约束可以引用同一表中的另一列（自引用）。

第50.2节：创建带有外键的表

在此示例中，我们有一个现有的表，超级英雄。

Chapter 50: Foreign Keys

Section 50.1: Foreign Keys explained

Foreign Keys constraints ensure data integrity, by enforcing that values in one table must match values in another table.

An example of where a foreign key is required is: In a university, a course must belong to a department. Code for the this scenario is:

```
CREATE TABLE Department (  
    Dept_Code      CHAR (5)      PRIMARY KEY,  
    Dept_Name      VARCHAR (20)  UNIQUE  
);
```

Insert values with the following statement:

```
INSERT INTO Department VALUES ('CS205', 'Computer Science');
```

The following table will contain the information of the subjects offered by the Computer science branch:

```
CREATE TABLE Programming_Courses (  
    Dept_Code      CHAR(5),  
    Prg_Code       CHAR(9) PRIMARY KEY,  
    Prg_Name       VARCHAR (50) UNIQUE,  
    FOREIGN KEY (Dept_Code) References Department(Dept_Code)  
);
```

(The data type of the Foreign Key must match the datatype of the referenced key.)

The Foreign Key constraint on the column Dept_Code allows values only if they already exist in the referenced table, Department. This means that if you try to insert the following values:

```
INSERT INTO Programming_Courses Values ('CS300', 'FDB-DB001', 'Database Systems');
```

the database will raise a Foreign Key violation error, because CS300 does not exist in the Department table. But when you try a key value that exists:

```
INSERT INTO Programming_Courses VALUES ('CS205', 'FDB-DB001', 'Database Systems');  
INSERT INTO Programming_Courses VALUES ('CS205', 'DB2-DB002', 'Database Systems II');
```

then the database allows these values.

A few tips for using Foreign Keys

- A Foreign Key must reference a UNIQUE (or PRIMARY) key in the parent table.
- Entering a NULL value in a Foreign Key column does not raise an error.
- Foreign Key constraints can reference tables within the same database.
- Foreign Key constraints can refer to another column in the same table (self-reference).

Section 50.2: Creating a table with a foreign key

In this example we have an existing table, SuperHeros.

此表包含一个主键ID。

我们将添加一个新表来存储每个超级英雄的能力：

```
CREATE TABLE HeroPowers
(
  ID int NOT NULL PRIMARY KEY,
  Name nvarchar(MAX) NOT NULL,
  HeroId int REFERENCES SuperHeros(ID)
)
```

列HeroId是表SuperHeros的外键。

This table contains a primary key ID.

We will add a new table in order to store the powers of each super hero:

```
CREATE TABLE HeroPowers
(
  ID int NOT NULL PRIMARY KEY,
  Name nvarchar(MAX) NOT NULL,
  HeroId int REFERENCES SuperHeros(ID)
)
```

The column HeroId is a **foreign key** to the table SuperHeros.

第51章：序列

第51.1节：创建序列

```
CREATE SEQUENCE orders_seq
START WITH      1000
INCREMENT BY    1;
```

创建一个起始值为1000，递增值为1的序列。

第51.2节：使用序列

对seq_name.NEXTVAL的引用用于获取序列中的下一个值。单个语句只能生成一个序列值。如果语句中有多个对NEXTVAL的引用，它们将使用相同的生成数字。

NEXTVAL可用于INSERT操作

```
INSERT INTO Orders (Order_UID, Customer)
VALUES (orders_seq.NEXTVAL, 1032);
```

它也可用于UPDATE操作

```
UPDATE Orders
SET Order_UID = orders_seq.NEXTVAL
WHERE Customer = 581;
```

它也可用于SELECT操作

```
SELECT Order_seq.NEXTVAL FROM dual;
```

Chapter 51: Sequence

Section 51.1: Create Sequence

```
CREATE SEQUENCE orders_seq
START WITH      1000
INCREMENT BY    1;
```

Creates a sequence with a starting value of 1000 which is incremented by 1.

Section 51.2: Using Sequences

a reference to *seq_name*.NEXTVAL is used to get the next value in a sequence. A single statement can only generate a single sequence value. If there are multiple references to NEXTVAL in a statement, they use will use the same generated number.

NEXTVAL can be used for INSERTS

```
INSERT INTO Orders (Order_UID, Customer)
VALUES (orders_seq.NEXTVAL, 1032);
```

It can be used for UPDATES

```
UPDATE Orders
SET Order_UID = orders_seq.NEXTVAL
WHERE Customer = 581;
```

It can also be used for SELECTS

```
SELECT Order_seq.NEXTVAL FROM dual;
```

第52章：子查询

第52.1节：FROM子句中的子查询

FROM子句中的子查询类似于在查询执行期间生成的临时表，执行结束后即被丢弃。

```
SELECT 经理.编号, 员工.薪水
FROM (
    SELECT 编号
    FROM Employees
    WHERE ManagerId IS NULL
) 作为 经理
JOIN 员工 ON 经理.编号 = 员工.编号
```

第52.2节：SELECT子句中的子查询

```
SELECT
编号,
名字,
姓氏,
(SELECT COUNT(*) FROM 车辆 WHERE 车辆.客户编号 = 客户.编号) 作为 车辆数量
FROM 客户
```

第52.3节：WHERE子句中的子查询

使用子查询过滤结果集。例如，这将返回所有薪水等于最高薪水员工的员工。

```
SELECT *
FROM 员工
WHERE 薪水 = (SELECT MAX(薪水) FROM 员工)
```

第52.4节：相关子查询

相关子查询（也称为同步子查询或协调子查询）是嵌套查询，它们引用外部查询的当前行：

```
SELECT 员工编号
FROM 员工 AS eOuter
WHERE 薪水 > (
    SELECT AVG(薪水)
    FROM 员工 eInner
    WHERE eInner.部门编号 = eOuter.部门编号
)
```

子查询 SELECT AVG(薪水) ... 是 相关的，因为它引用了外部查询中的 员工 行 eOuter。

第52.5节：使用不同表的查询过滤查询结果

此查询选择所有不在主管表中的员工。

```
SELECT *
```

Chapter 52: Subqueries

Section 52.1: Subquery in FROM clause

A subquery in a **FROM** clause acts similarly to a temporary table that is generated during the execution of a query and lost afterwards.

```
SELECT Managers.Id, Employees.Salary
FROM (
    SELECT Id
    FROM Employees
    WHERE ManagerId IS NULL
) AS Managers
JOIN Employees ON Managers.Id = Employees.Id
```

Section 52.2: Subquery in SELECT clause

```
SELECT
Id,
FName,
LName,
(SELECT COUNT(*) FROM Cars WHERE Cars.CustomerId = Customers.Id) AS NumberOfCars
FROM Customers
```

Section 52.3: Subquery in WHERE clause

Use a subquery to filter the result set. For example this will return all employees with a salary equal to the highest paid employee.

```
SELECT *
FROM Employees
WHERE Salary = (SELECT MAX(Salary) FROM Employees)
```

Section 52.4: Correlated Subqueries

Correlated (also known as Synchronized or Coordinated) Subqueries are nested queries that make references to the current row of their outer query:

```
SELECT EmployeeId
FROM Employee AS eOuter
WHERE Salary > (
    SELECT AVG(Salary)
    FROM Employee eInner
    WHERE eInner.DepartmentId = eOuter.DepartmentId
)
```

Subquery SELECT AVG(Salary) ... is *correlated* because it refers to Employee row eOuter from its outer query.

Section 52.5: Filter query results using query on different table

This query selects all employees not on the Supervisors table.

```
SELECT *
```

```
FROM 员工
WHERE EmployeeID 不在(SELECT EmployeeID
                        FROM Supervisors)
```

可以使用 LEFT JOIN 来实现相同的结果。

```
SELECT *
FROM Employees 作为 e
LEFT JOIN Supervisors 作为 s ON s.EmployeeID=e.EmployeeID
WHERE s.EmployeeID 是NULL
```

第52.6节：FROM子句中的子查询

你可以使用子查询来定义一个临时表，并在“外部”查询的FROM子句中使用它。

```
SELECT * FROM (SELECT city, temp_hi - temp_lo 作为 temp_var FROM weather) 作为 w
WHERE temp_var > 20;
```

上述查询找出weather表中日温差大于20的城市。结果是：

city	temp_var
圣路易斯	21
洛杉矶	31
洛杉矶	23
洛杉矶	31
洛杉矶	27
洛杉矶	28
洛杉矶	28
洛杉矶	32

.

第52.7节：WHERE子句中的子查询

下面的例子查找人口低于平均温度的城市（来自cities示例，平均温度通过子查询获得）：

```
SELECT name, pop2000 FROM cities
WHERE pop2000 < (SELECT avg(pop2000) FROM cities);
```

这里：子查询（SELECT avg(pop2000) FROM cities）用于在WHERE子句中指定条件。结果是：

姓名	pop2000
旧金山	776733
圣路易斯	348189
堪萨斯城	146866

```
FROM Employees
WHERE EmployeeID not in (SELECT EmployeeID
                        FROM Supervisors)
```

The same results can be achieved using a LEFT JOIN.

```
SELECT *
FROM Employees AS e
LEFT JOIN Supervisors AS s ON s.EmployeeID=e.EmployeeID
WHERE s.EmployeeID is NULL
```

Section 52.6: Subqueries in FROM clause

You can use subqueries to define a temporary table and use it in the FROM clause of an "outer" query.

```
SELECT * FROM (SELECT city, temp_hi - temp_lo AS temp_var FROM weather) AS w
WHERE temp_var > 20;
```

The above finds cities from the weather table whose daily temperature variation is greater than 20. The result is:

city	temp_var
ST LOUIS	21
LOS ANGELES	31
LOS ANGELES	23
LOS ANGELES	31
LOS ANGELES	27
LOS ANGELES	28
LOS ANGELES	28
LOS ANGELES	32

.

Section 52.7: Subqueries in WHERE clause

The following example finds cities (from the cities example) whose population is below the average temperature (obtained via a sub-qquery):

```
SELECT name, pop2000 FROM cities
WHERE pop2000 < (SELECT avg(pop2000) FROM cities);
```

Here: the subquery (SELECT avg(pop2000) FROM cities) is used to specify conditions in the WHERE clause. The result is:

name	pop2000
San Francisco	776733
ST LOUIS	348189
Kansas City	146866

第53章：执行块

第53.1节：使用 BEGIN ... END

```
BEGIN
  UPDATE Employees SET PhoneNumber = '5551234567' WHERE Id = 1;
  UPDATE Employees SET Salary = 650 WHERE Id = 3;
结束
```

Chapter 53: Execution blocks

Section 53.1: Using BEGIN ... END

```
BEGIN
  UPDATE Employees SET PhoneNumber = '5551234567' WHERE Id = 1;
  UPDATE Employees SET Salary = 650 WHERE Id = 3;
END
```

第54章：存储过程

第54.1节：创建和调用存储过程

存储过程可以通过数据库管理图形界面（[SQL Server示例](#)）创建，或者通过如下SQL语句创建：

```
-- 定义名称和参数
CREATE PROCEDURE Northwind.getEmployee
    @LastName nvarchar(50),
    @FirstName nvarchar(50)
AS

-- 定义要执行的查询
SELECT FirstName, LastName, Department
FROM Northwind.vEmployeeDepartment
WHERE FirstName = @FirstName AND LastName = @LastName
AND EndDate IS NULL;
```

调用该存储过程：

```
EXECUTE Northwind.getEmployee N'Ackerman', N'Pilar';

-- 或者
EXEC Northwind.getEmployee @LastName = N'Ackerman', @FirstName = N'Pilar';
GO

-- 或者
EXECUTE Northwind.getEmployee @FirstName = N'Pilar', @LastName = N'Ackerman';
GO
```

Chapter 54: Stored Procedures

Section 54.1: Create and call a stored procedure

Stored procedures can be created through a database management GUI ([SQL Server example](#)), or through a SQL statement as follows:

```
-- Define a name and parameters
CREATE PROCEDURE Northwind.getEmployee
    @LastName nvarchar(50),
    @FirstName nvarchar(50)
AS

-- Define the query to be run
SELECT FirstName, LastName, Department
FROM Northwind.vEmployeeDepartment
WHERE FirstName = @FirstName AND LastName = @LastName
AND EndDate IS NULL;
```

Calling the procedure:

```
EXECUTE Northwind.getEmployee N'Ackerman', N'Pilar';

-- Or
EXEC Northwind.getEmployee @LastName = N'Ackerman', @FirstName = N'Pilar';
GO

-- Or
EXECUTE Northwind.getEmployee @FirstName = N'Pilar', @LastName = N'Ackerman';
GO
```

第55章：触发器

第55.1节：CREATE TRIGGER

此示例创建了一个触发器，当在定义触发器的表（MyTable）中插入记录后，会向第二个表（MyAudit）插入一条记录。这里的“inserted”表是微软SQL Server使用的一个特殊表，用于存储INSERT和UPDATE语句中受影响的行；还有一个特殊的“deleted”表，用于DELETE语句执行相同的功能。

```
CREATE TRIGGER MyTrigger
ON MyTable
AFTER INSERT

AS

BEGIN
-- 将审计记录插入到 MyAudit 表
INSERT INTO MyAudit(MyTableId, User)
(SELECT MyTableId, CURRENT_USER FROM inserted)
END
```

第55.2节：使用触发器管理已删除项目的“回收站”

```
创建触发器 BooksDeleteTrigger
在 MyBooksDB.Books
删除后触发

AS

插入到 BooksRecycleBin
选择 *
来自 deleted；

GO
```

Chapter 55: Triggers

Section 55.1: CREATE TRIGGER

This example creates a trigger that inserts a record to a second table (MyAudit) after a record is inserted into the table the trigger is defined on (MyTable). Here the "inserted" table is a special table used by Microsoft SQL Server to store affected rows during INSERT and UPDATE statements; there is also a special "deleted" table that performs the same function for DELETE statements.

```
CREATE TRIGGER MyTrigger
ON MyTable
AFTER INSERT

AS

BEGIN
-- insert audit record to MyAudit table
INSERT INTO MyAudit(MyTableId, User)
(SELECT MyTableId, CURRENT_USER FROM inserted)
END
```

Section 55.2: Use Trigger to manage a "Recycle Bin" for deleted items

```
CREATE TRIGGER BooksDeleteTrigger
ON MyBooksDB.Books
AFTER DELETE

AS

INSERT INTO BooksRecycleBin
SELECT *
FROM deleted;

GO
```

第56章：事务

第56.1节：简单事务

```
开始事务
    插入到 DeletedEmployees(EmployeeID, DateDeleted, User)
        (选择 123, GetDate(), CURRENT_USER);
    从 Employees 删除 条件 EmployeeID = 123 ;
提交事务
```

第56.2节：回滚事务

当事务代码中出现错误且需要撤销时，可以回滚事务：

```
BEGIN TRY
    开始事务
        INSERT INTO Users(ID, Name, Age)
        VALUES(1, 'Bob', 24)

        DELETE FROM Users WHERE Name = 'Todd'
    COMMIT TRANSACTION
END TRY
BEGIN CATCH
    ROLLBACK TRANSACTION
END CATCH
```

Chapter 56: Transactions

Section 56.1: Simple Transaction

```
BEGIN TRANSACTION
    INSERT INTO DeletedEmployees(EmployeeID, DateDeleted, User)
        (SELECT 123, GetDate(), CURRENT_USER);
    DELETE FROM Employees WHERE EmployeeID = 123;
COMMIT TRANSACTION
```

Section 56.2: Rollback Transaction

When something fails in your transaction code and you want to undo it, you can rollback your transaction:

```
BEGIN TRY
    BEGIN TRANSACTION
        INSERT INTO Users(ID, Name, Age)
        VALUES(1, 'Bob', 24)

        DELETE FROM Users WHERE Name = 'Todd'
    COMMIT TRANSACTION
END TRY
BEGIN CATCH
    ROLLBACK TRANSACTION
END CATCH
```

第57章：表设计

第57.1节：良好设计表的属性

真正的关系型数据库必须超越仅仅将数据放入几个表中并编写一些SQL语句来提取数据的做法。

设计不良的表结构最多会降低查询的执行速度，甚至可能使数据库无法按预期运行。

数据库表不应仅被视为另一个普通表；它必须遵循一套规则，才能被真正视为关系型表。学术上称之为“关系”，以示区别。

关系表的五条规则是：

- 1. 每个值都是原子性的；每行中每个字段的值必须是单一值。
- 2. 每个字段包含的数据类型必须相同。
- 3. 每个字段标题必须有唯一的名称。
- 4. 表中的每一行必须至少有一个值使其在其他记录中唯一。

- 5. 行和列的顺序没有意义。

符合这五条规则的表：

编号	姓名	出生日期	经理
1	弗雷德	1971年11月2日	3
2	弗雷德	1971年11月2日	3
3	苏	08/07/1975	2

- 规则1：每个值都是原子的。Id、Name、DOB和Manager只包含单一值。
- 规则2：Id只包含整数，Name包含文本（我们可以补充说明文本长度不超过四个字符），DOB包含有效类型的日期，Manager包含整数（我们可以补充说明其对应于经理表中的主键字段）。

- 规则3：Id、Name、DOB和Manager是表中唯一的列名。
- 规则4：包含Id字段确保每条记录在表中都是唯一的，彼此不同。

设计不良的表：

编号	姓名	出生日期	姓名
1	弗雷德	1971年11月2日	3
1	弗雷德	1971年11月2日	3
3	苏	1975年7月18日星期五	2, 1

- 规则1：第二个姓名字段包含两个值——2和1。
- 规则2：DOB字段包含日期和文本。
- 规则3：有两个字段都叫“name”。
- 规则4：第一条和第二条记录完全相同。
- 规则5：此规则未被违反。

Chapter 57: Table Design

Section 57.1: Properties of a well designed table

A true relational database must go beyond throwing data into a few tables and writing some SQL statements to pull that data out.

At best a badly designed table structure will slow the execution of queries and could make it impossible for the database to function as intended.

A database table should not be considered as just another table; it has to follow a set of rules to be considered truly relational. Academically it is referred to as a 'relation' to make the distinction.

The five rules of a relational table are:

- 1. Each value is *atomic*; the value in each field in each row must be a single value.
- 2. Each field contains values that are of the same data type.
- 3. Each field heading has a unique name.
- 4. Each row in the table must have at least one value that makes it unique amongst the other records in the table.
- 5. The order of the rows and columns has no significance.

A table conforming to the five rules:

Id	Name	DOB	Manager
1	Fred	11/02/1971	3
2	Fred	11/02/1971	3
3	Sue	08/07/1975	2

- Rule 1: Each value is atomic. Id, Name, DOB and Manager only contain a single value.
- Rule 2: Id contains only integers, Name contains text (we could add that it's text of four characters or less), DOB contains dates of a valid type and Manager contains integers (we could add that corresponds to a Primary Key field in a managers table).
- Rule 3: Id, Name, DOB and Manager are unique heading names within the table.
- Rule 4: The inclusion of the Id field ensures that each record is distinct from any other record within the table.

A badly designed table:

Id	Name	DOB	Name
1	Fred	11/02/1971	3
1	Fred	11/02/1971	3
3	Sue	Friday the 18th July 1975	2, 1

- Rule 1: The second name field contains two values - 2 and 1.
- Rule 2: The DOB field contains dates and text.
- Rule 3: There's two fields called 'name'.
- Rule 4: The first and second record are exactly the same.
- Rule 5: This rule isn't broken.

第58章：同义词

第58.1节：创建同义词

```
CREATE SYNONYM EmployeeData
FOR MyDatabase.dbo.Employees
```

Chapter 58: Synonyms

Section 58.1: Create Synonym

```
CREATE SYNONYM EmployeeData
FOR MyDatabase.dbo.Employees
```


第59章：信息架构

第59.1节：基本信息架构查询

对于大型关系数据库管理系统的终端用户来说，最有用的查询之一是对信息架构的搜索。

这样的查询允许用户快速查找包含感兴趣列的数据库表，例如在尝试通过第三个表间接关联两个表的数据时，而事先并不知道哪些表可能包含与目标表共有的键或其他有用列。

以T-SQL为例，可以按如下方式搜索数据库的信息模式：

```
SELECT *
FROM INFORMATION_SCHEMA.COLUMNS
WHERE COLUMN_NAME LIKE '%Institution%'
```

结果包含匹配列的列表、它们所在表的名称以及其他有用信息。

Chapter 59: Information Schema

Section 59.1: Basic Information Schema Search

One of the most useful queries for end users of large RDBMS's is a search of an information schema.

Such a query allows users to rapidly find database tables containing columns of interest, such as when attempting to relate data from 2 tables indirectly through a third table, without existing knowledge of which tables may contain keys or other useful columns in common with the target tables.

Using T-SQL for this example, a database's information schema may be searched as follows:

```
SELECT *
FROM INFORMATION_SCHEMA.COLUMNS
WHERE COLUMN_NAME LIKE '%Institution%'
```

The result contains a list of matching columns, their tables' names, and other useful information.

第60章：执行顺序

第60.1节：SQL中查询处理的逻辑顺序

```
/*(8)*/ SELECT /*9*/ DISTINCT /*11*/ TOP
/*(1)*/ FROM
/*(3)*/      JOIN
/*(2)*/      ON
/*(4)*/ WHERE
/*(5)*/ GROUP BY
/*(6)*/ WITH {CUBE | ROLLUP}
/*(7)*/ HAVING
/*(10)*/ ORDER BY
/*(11)*/ LIMIT
```

查询的处理顺序及各部分的描述。

VT代表“虚拟表”，显示查询处理过程中如何生成各种数据

- 1.FROM：在FROM子句中的前两个表之间执行笛卡尔积（交叉连接），结果生成虚拟表VT1。
- 2.ON：对VT1应用ON过滤器。只有满足条件为TRUE的行才会被插入到VT2中。
- 3.OUTER（连接）：如果指定了OUTER JOIN（与CROSS JOIN或INNER JOIN相对），则将未找到匹配的保留表中的行作为外部行添加到VT2的行中，生成VT3。如果FROM子句中出现多个表，则步骤1到3会在上一次连接的结果与FROM子句中的下一个表之间重复应用，直到处理完所有表。
- 4.WHERE：对VT3应用WHERE过滤器。只有满足条件为TRUE的行才会被插入到VT4中。
- 5.GROUP BY：根据GROUP BY子句中指定的列列表，将VT4中的行分组，生成VT5。
- 6.CUBE | ROLLUP：向VT5的行中添加超级组（组的组），生成VT6。
- 7.HAVING：对VT6应用HAVING过滤器。只有满足条件为TRUE的组才会被插入到VT7中。
- 8.SELECT：处理SELECT列表，生成VT8。
- 9.DISTINCT：从VT8中删除重复行，生成VT9。
- 10.排序：VT9中的行根据ORDER BY子句中指定的列列表进行排序。生成一个游标（VC10）。
- 11.TOP：从VC10的开头选择指定数量或百分比的行。生成表VT11并返回给调用者。LIMIT在某些SQL方言中（如Postgres和Netezza）具有与TOP相同的功能。

Chapter 60: Order of Execution

Section 60.1: Logical Order of Query Processing in SQL

```
/*(8)*/ SELECT /*9*/ DISTINCT /*11*/ TOP
/*(1)*/ FROM
/*(3)*/      JOIN
/*(2)*/      ON
/*(4)*/ WHERE
/*(5)*/ GROUP BY
/*(6)*/ WITH {CUBE | ROLLUP}
/*(7)*/ HAVING
/*(10)*/ ORDER BY
/*(11)*/ LIMIT
```

The order in which a query is processed and description of each section.

VT stands for 'Virtual Table' and shows how various data is produced as the query is processed

1. FROM: A Cartesian product (cross join) is performed between the first two tables in the FROM clause, and as a result, virtual table VT1 is generated.
2. ON: The ON filter is applied to VT1. Only rows for which the is TRUE are inserted to VT2.
3. OUTER (join): If an OUTER JOIN is specified (as opposed to a CROSS JOIN or an INNER JOIN), rows from the preserved table or tables for which a match was not found are added to the rows from VT2 as outer rows, generating VT3. If more than two tables appear in the FROM clause, steps 1 through 3 are applied repeatedly between the result of the last join and the next table in the FROM clause until all tables are processed.
4. WHERE: The WHERE filter is applied to VT3. Only rows for which the is TRUE are inserted to VT4.
5. GROUP BY: The rows from VT4 are arranged in groups based on the column list specified in the GROUP BY clause. VT5 is generated.
6. CUBE | ROLLUP: Supergroups (groups of groups) are added to the rows from VT5, generating VT6.
7. HAVING: The HAVING filter is applied to VT6. Only groups for which the is TRUE are inserted to VT7.
8. SELECT: The SELECT list is processed, generating VT8.
9. DISTINCT: Duplicate rows are removed from VT8. VT9 is generated.
10. ORDER BY: The rows from VT9 are sorted according to the column list specified in the ORDER BY clause. A cursor is generated (VC10).
11. TOP: The specified number or percentage of rows is selected from the beginning of VC10. Table VT11 is generated and returned to the caller. LIMIT has the same functionality as TOP in some SQL dialects such as Postgres and Netezza.

第61章：SQL中的整洁代码

如何编写良好、可读的SQL查询，以及良好实践的示例。

第61.1节：关键字和名称的格式及拼写

表/列名称

格式化表/列名称的两种常见方式是驼峰命名法（CamelCase）和蛇形命名法（snake_case）：

```
SELECT FirstName, LastName
FROM Employees
WHERE Salary > 500;

SELECT first_name, last_name
FROM employees
WHERE salary > 500;
```

名称应描述其对象中存储的内容。这意味着列名通常为单数形式。
关于表名应该使用单数还是复数，这是一个备受讨论的问题，但实际上，更常见的是使用复数表名。

添加前缀或后缀如 tbl或 col会降低可读性，因此应避免使用。然而，有时为了避免与SQL关键字冲突，会使用它们，并且通常用于触发器和索引（这些名称通常不会在查询中提及）。

关键字

SQL关键字不区分大小写。但通常的做法是将它们写成大写。

第61.2节：缩进

没有广泛接受的标准。大家一致认为，将所有内容挤在一行是糟糕的：

```
SELECT d.Name, COUNT(*) AS Employees FROM Departments AS d JOIN Employees AS e ON d.ID = e.DepartmentID WHERE d.Name != 'HR' HAVING COUNT(*) > 10 ORDER BY COUNT(*) DESC;
```

至少，每个子句应放在新行，如果行太长，则应拆分：

```
SELECT d.Name,
       COUNT(*) AS Employees
FROM 部门 AS d
JOIN 员工 AS e ON d.ID = e.DepartmentID
WHERE d.Name != '人力资源'
HAVING COUNT(*) > 10
ORDER BY COUNT(*) DESC;
```

有时，SQL关键字引入子句后面的所有内容都会缩进到同一列：

```
SELECT  d.名称,
       COUNT(*) AS Employees
FROM    部门 AS d
JOIN    员工 AS e ON d.ID = e.DepartmentID
WHERE   d.名称 != '人力资源'
HAVING  COUNT(*) > 10
```

Chapter 61: Clean Code in SQL

How to write good, readable SQL queries, and example of good practices.

Section 61.1: Formatting and Spelling of Keywords and Names

Table/Column Names

Two common ways of formatting table/column names are `CamelCase` and `snake_case`:

```
SELECT FirstName, LastName
FROM Employees
WHERE Salary > 500;

SELECT first_name, last_name
FROM employees
WHERE salary > 500;
```

Names should describe what is stored in their object. This implies that column names usually should be singular. Whether table names should use singular or plural is a [heavily discussed](#) question, but in practice, it is more common to use plural table names.

Adding prefixes or suffixes like `tbl` or `col` reduces readability, so avoid them. However, they are sometimes used to avoid conflicts with SQL keywords, and often used with triggers and indexes (whose names are usually not mentioned in queries).

Keywords

SQL keywords are not case sensitive. However, it is common practice to write them in upper case.

Section 61.2: Indenting

There is no widely accepted standard. What everyone agrees on is that squeezing everything into a single line is bad:

```
SELECT d.Name, COUNT(*) AS Employees FROM Departments AS d JOIN Employees AS e ON d.ID = e.DepartmentID WHERE d.Name != 'HR' HAVING COUNT(*) > 10 ORDER BY COUNT(*) DESC;
```

At the minimum, put every clause into a new line, and split lines if they would become too long otherwise:

```
SELECT d.Name,
       COUNT(*) AS Employees
FROM Departments AS d
JOIN Employees AS e ON d.ID = e.DepartmentID
WHERE d.Name != 'HR'
HAVING COUNT(*) > 10
ORDER BY COUNT(*) DESC;
```

Sometimes, everything after the SQL keyword introducing a clause is indented to the same column:

```
SELECT  d.Name,
       COUNT(*) AS Employees
FROM    Departments AS d
JOIN    Employees AS e ON d.ID = e.DepartmentID
WHERE   d.Name != 'HR'
HAVING  COUNT(*) > 10
```

```
ORDER BY COUNT(*) DESC;
```

(这也可以在SQL关键字右对齐时完成。)

另一种常见的风格是将重要关键字单独放在一行：

```
SELECT
d.名称,
    COUNT(*) 作为 员工
来自
部门 AS d
JOIN
员工 AS e
    ON d.ID = e.DepartmentID
WHERE
d.名称 != 'HR'
HAVING
    COUNT(*) > 10
ORDER BY
    COUNT(*) DESC;
```

垂直对齐多个相似表达式可以提高可读性：

```
SELECT 型号,
        员工ID
FROM 车辆
WHERE 客户ID = 42
      AND 状态      = '准备好';
```

使用多行会使将SQL命令嵌入其他编程语言变得更困难。然而，许多语言都有多行字符串的机制，例如C#中的`@"..."`，Python中的`"""..."""`，或C++中的`R"(...)"`。

第61.3节：SELECT *

SELECT * 返回表中定义的所有列，顺序保持不变。

使用 SELECT * 时，查询返回的数据可能会随着表定义的变化而变化。这增加了应用程序不同版本或数据库之间不兼容的风险。

此外，读取比实际需要更多的列会增加磁盘和网络的输入输出量。

因此，您应始终明确指定您实际想要检索的列：

```
--SELECT *                不要
SELECT ID, FName, LName, PhoneNumber -- 要
FROM Employees;
```

(进行交互式查询时，这些考虑不适用。)

然而，SELECT * 在 EXISTS 操作符的子查询中无害，因为 EXISTS 无视实际数据（它只检查是否至少找到一行）。出于同样的原因，为 EXISTS 列出任何特定列没有意义，所以 SELECT * 实际上更合理：

```
-- 列出最近没有人被雇佣的部门
SELECT ID,
Name
FROM Departments
```

```
ORDER BY COUNT(*) DESC;
```

(This can also be done while aligning the SQL keywords right.)

Another common style is to put important keywords on their own lines:

```
SELECT
    d.Name,
    COUNT(*) AS Employees
FROM
    Departments AS d
JOIN
    Employees AS e
    ON d.ID = e.DepartmentID
WHERE
    d.Name != 'HR'
HAVING
    COUNT(*) > 10
ORDER BY
    COUNT(*) DESC;
```

Vertically aligning multiple similar expressions improves readability:

```
SELECT Model,
        EmployeeID
FROM Cars
WHERE CustomerID = 42
      AND Status      = 'READY';
```

Using multiple lines makes it harder to embed SQL commands into other programming languages. However, many languages have a mechanism for multi-line strings, e.g., `@"..."` in C#, `"""..."""` in Python, or `R"(...)"` in C++.

Section 61.3: SELECT *

SELECT * returns all columns in the same order as they are defined in the table.

When using SELECT *, the data returned by a query can change whenever the table definition changes. This increases the risk that different versions of your application or your database are incompatible with each other.

Furthermore, reading more columns than necessary can increase the amount of disk and network I/O.

So you should always explicitly specify the column(s) you actually want to retrieve:

```
--SELECT *                don't
SELECT ID, FName, LName, PhoneNumber -- do
FROM Employees;
```

(When doing interactive queries, these considerations do not apply.)

However, SELECT * does not hurt in the subquery of an EXISTS operator, because EXISTS ignores the actual data anyway (it checks only if at least one row has been found). For the same reason, it is not meaningful to list any specific column(s) for EXISTS, so SELECT * actually makes more sense:

```
-- list departments where nobody was hired recently
SELECT ID,
        Name
FROM Departments
```

```
WHERE NOT EXISTS (SELECT *
                  FROM 员工
                  WHERE DepartmentID = Departments.ID
                  AND HireDate >= '2015-01-01');
```

第61.4节：连接

应始终使用显式连接；隐式连接存在若干问题：

- 连接条件位于WHERE子句中的某处，与其他过滤条件混杂在一起。这使得更难看出哪些表被连接，以及连接方式。
- 由于上述原因，出错的风险更高，而且错误更可能在后期被发现。
- 在标准SQL中，显式连接是使用外连接的唯一方式：

```
SELECT d.Name,
       e.Fname || e.LName AS EmpName
FROM   Departments AS d
LEFT JOIN Employees AS e ON d.ID = e.DepartmentID;
```

- 显式连接允许使用USING子句：

```
SELECT RecipeID,
       Recipes.Name,
       COUNT(*) AS NumberOfIngredients
FROM   Recipes
LEFT JOIN Ingredients USING (RecipeID);
```

（这要求两个表使用相同的列名。

USING 会自动从结果中移除重复的列，例如，此查询中的连接返回一个单一的RecipeID列。）

```
WHERE NOT EXISTS (SELECT *
                  FROM Employees
                  WHERE DepartmentID = Departments.ID
                  AND HireDate >= '2015-01-01');
```

Section 61.4: Joins

Explicit joins should always be used; implicit joins have several problems:

- The join condition is somewhere in the WHERE clause, mixed up with any other filter conditions. This makes it harder to see which tables are joined, and how.
- Due to the above, there is a higher risk of mistakes, and it is more likely that they are found later.
- In standard SQL, explicit joins are the only way to use outer joins:

```
SELECT d.Name,
       e.Fname || e.LName AS EmpName
FROM   Departments AS d
LEFT JOIN Employees AS e ON d.ID = e.DepartmentID;
```

- Explicit joins allow using the USING clause:

```
SELECT RecipeID,
       Recipes.Name,
       COUNT(*) AS NumberOfIngredients
FROM   Recipes
LEFT JOIN Ingredients USING (RecipeID);
```

(This requires that both tables use the same column name.

USING automatically removes the duplicate column from the result, e.g., the join in this query returns a single RecipeID column.)

第62章：SQL注入

SQL注入是通过在表单字段中注入SQL代码，试图访问网站数据库表的行为。如果网页服务器没有防护SQL注入攻击，黑客可以欺骗数据库执行额外的SQL代码。

通过执行他们自己的SQL代码，黑客可以提升账户权限，查看他人的私人信息，或对数据库进行任何其他修改。

第62.1节：SQL注入示例

假设对您的网页应用登录处理程序的调用如下：

```
https://somepage.com/ajax/login.ashx?username=admin&password=123
```

现在在login.ashx中，您读取这些值：

```
strUserName = getHttpRequestParameterString("username");
strPassword = getHttpRequestParameterString("password");
```

并查询您的数据库以确定是否存在使用该密码的用户。

所以你构造了一个SQL查询字符串：

```
txtSQL = "SELECT * FROM Users WHERE username = '" + strUserName + "' AND password = '" + strPassword + "'";
```

如果用户名和密码中不包含引号，这样是可以工作的。

但是，如果其中一个参数确实包含引号，发送到数据库的SQL将会是这样的：

```
-- strUserName = "d'Alambert";
txtSQL = "SELECT * FROM Users WHERE username = 'd'Alambert' AND password = '123'";
```

这将导致语法错误，因为 d'Alambert 中的 d 后的引号结束了SQL字符串。

你可以通过转义用户名和密码中的引号来修正，例如：

```
strUserName = strUserName.Replace("'", "''");
strPassword = strPassword.Replace("'", "''");
```

不过，更合适的做法是使用参数：

```
cmd.CommandText = "SELECT * FROM Users WHERE username = @username AND password = @password";

cmd.Parameters.Add("@username", strUserName);
cmd.Parameters.Add("@password", strPassword);
```

如果不使用参数，并且忘记替换其中一个值中的引号，那么恶意用户（即黑客）可以利用这一点在你的数据库上执行SQL命令。

例如，如果攻击者是恶意的，他/她会将密码设置为

```
lol'; DROP DATABASE master; --
```

然后SQL语句将变成这样：

Chapter 62: SQL Injection

SQL injection is an attempt to access a website's database tables by injecting SQL into a form field. If a web server does not protect against SQL injection attacks, a hacker can trick the database into running the additional SQL code. By executing their own SQL code, hackers can upgrade their account access, view someone else's private information, or make any other modifications to the database.

Section 62.1: SQL injection sample

Assuming the call to your web application's login handler looks like this:

```
https://somepage.com/ajax/login.ashx?username=admin&password=123
```

Now in login.ashx, you read these values:

```
strUserName = getHttpRequestParameterString("username");
strPassword = getHttpRequestParameterString("password");
```

and query your database to determine whether a user with that password exists.

So you construct an SQL query string:

```
txtSQL = "SELECT * FROM Users WHERE username = '" + strUserName + "' AND password = '" + strPassword + "'";
```

This will work if the username and password do not contain a quote.

However, if one of the parameters does contain a quote, the SQL that gets sent to the database will look like this:

```
-- strUserName = "d'Alambert";
txtSQL = "SELECT * FROM Users WHERE username = 'd'Alambert' AND password = '123'";
```

This will result in a syntax error, because the quote after the d in d'Alambert ends the SQL string.

You could correct this by escaping quotes in username and password, e.g.:

```
strUserName = strUserName.Replace("'", "''");
strPassword = strPassword.Replace("'", "''");
```

However, it's more appropriate to use parameters:

```
cmd.CommandText = "SELECT * FROM Users WHERE username = @username AND password = @password";

cmd.Parameters.Add("@username", strUserName);
cmd.Parameters.Add("@password", strPassword);
```

If you do not use parameters, and forget to replace quote in even one of the values, then a malicious user (aka hacker) can use this to execute SQL commands on your database.

For example, if an attacker is evil, he/she will set the password to

```
lol'; DROP DATABASE master; --
```

and then the SQL will look like this:


```
"SELECT * FROM Users WHERE username = 'somebody' AND password = 'lol'; DROP DATABASE master; --";
```

不幸的是，这是一条有效的SQL语句，数据库会执行它！

这种类型的漏洞称为SQL注入。

恶意用户还可以做许多其他事情，比如窃取每个用户的电子邮件地址、窃取所有人的密码、窃取信用卡号码、窃取数据库中的任意数据等。

这就是为什么你总是需要对字符串进行转义。而且你迟早会忘记这么做，这正是你应该使用参数的原因。因为如果你使用参数，那么你的编程语言框架会为你做任何必要的转义。

第62.2节：简单注入示例

如果SQL语句是这样构造的：

```
SQL = "SELECT * FROM Users WHERE username = '" + user + "' AND password = '" + pw + "'";
db.execute(SQL);
```

那么黑客可能通过输入类似 pw' or '1'='1; 的密码来获取你的数据；生成的SQL语句将会是：

```
SELECT * FROM Users WHERE username = 'somebody' AND password ='pw' or '1'='1'
```

这条语句会使 Users 表中所有行都通过密码检查，因为 '1'='1' 永远为真。

为防止这种情况，使用SQL参数：

```
SQL = "SELECT * FROM Users WHERE username = ? AND password = ?";
db.execute(SQL, [user, pw]);
```

```
"SELECT * FROM Users WHERE username = 'somebody' AND password = 'lol'; DROP DATABASE master; --";
```

Unfortunately for you, this is valid SQL, and the DB will execute this!

This type of exploit is called an SQL injection.

There are many other things a malicious user could do, such as stealing every user's email address, steal everyone's password, steal credit card numbers, steal any amount of data in your database, etc.

This is why you always need to escape your strings. And the fact that you'll invariably forget to do so sooner or later is exactly why you should use parameters. Because if you use parameters, then your programming language framework will do any necessary escaping for you.

Section 62.2: simple injection sample

If the SQL statement is constructed like this:

```
SQL = "SELECT * FROM Users WHERE username = '" + user + "' AND password = '" + pw + "'";
db.execute(SQL);
```

Then a hacker could retrieve your data by giving a password like pw' or '1'='1; the resulting SQL statement will be:

```
SELECT * FROM Users WHERE username = 'somebody' AND password ='pw' or '1'='1'
```

This one will pass the password check for all rows in the Users table because '1'='1' is always true.

To prevent this, use SQL parameters:

```
SQL = "SELECT * FROM Users WHERE username = ? AND password = ?";
db.execute(SQL, [user, pw]);
```

致谢

非常感谢所有来自Stack Overflow Documentation的人员提供此内容，
更多更改可发送至web@petercv.com以发布或更新新内容

奥兹古尔·奥兹图尔克	第8章和第17章
3N1GM4	第7章
a1ex07	第37章
阿贝·米斯勒	第7章
阿比拉什·R·万卡亚拉	第5、6、11、27、30和32章
aholmes	第6章
艾丹	第21章和第25章
alex9311	第21章
阿尔米尔·武克	第21章和第37章
阿洛克·辛格	第6章
阿梅亚·德什潘德	第26章
阿米尔·普尔曼 امیر پورمند	第56章
阿姆农	第6章
安德里亚	第24章
安德里亚·蒙塔纳里	第36章
安德烈亚斯	第2章
安迪·G	第18章
apomene	第6章
阿瑞斯	第21章
阿尔克	第45章
阿尔皮特·索兰基	第6章
亚瑟·D	第41章
阿鲁尔库马尔	第13章和第41章
ashja99	第11章和第42章
阿塔福德	第24章
阿亚兹·沙阿	第11章
A_阿诺德	第18章
巴特·斯库伊特	第11章
巴茨	第41章
bhs	第45章
大鼻子	第5章
黑色主教	第25章
Blag	第17章
Bostjan	第5、7和13章
布兰科·迪米特里耶维奇	第18章
布伦特·奥利弗	第6章
brichins	第54章
carlosb	第37章和第39章
克里斯	第6章
克里斯蒂安	第5章
克里斯蒂安·萨格米勒	第6章
克里斯托斯	第6章
CL	第1、2、6、8、10、14、18、19、21、31、36、37、41、42、46、49、61和62章
克里斯蒂安·阿贝列拉	第30章
DalmTo	第30章
丹尼尔	第46章

Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content,
more changes can be sent to web@petercv.com for new content to be published or updated

Özgür Öztürk	Chapters 8 and 17
3N1GM4	Chapter 7
a1ex07	Chapter 37
Abe Miessler	Chapter 7
Abhilash R Vankayala	Chapters 5, 6, 11, 27, 30 and 32
aholmes	Chapter 6
Aidan	Chapters 21 and 25
alex9311	Chapter 21
Almir Vuk	Chapters 21 and 37
Alok Singh	Chapter 6
Ameya Deshpande	Chapter 26
Amir Pourmand امیر پورمند	Chapter 56
Amnon	Chapter 6
Andrea	Chapter 24
Andrea Montanari	Chapter 36
Andreas	Chapter 2
Andy G	Chapter 18
apomene	Chapter 6
Ares	Chapter 21
Arkh	Chapter 45
Arpit Solanki	Chapter 6
Arthur D	Chapter 41
Arulkumar	Chapters 13 and 41
ashja99	Chapters 11 and 42
Athafoud	Chapter 24
Ayaz Shah	Chapter 11
A_Arnold	Chapter 18
Bart Schuijt	Chapter 11
Batsu	Chapter 41
bhs	Chapter 45
bignose	Chapter 5
blackbishop	Chapter 25
Blag	Chapter 17
Bostjan	Chapters 5, 7 and 13
Branko Dimitrijevic	Chapter 18
Brent Oliver	Chapter 6
brichins	Chapter 54
carlosb	Chapters 37 and 39
Chris	Chapter 6
Christian	Chapter 5
Christian Sagmüller	Chapter 6
Christos	Chapter 6
CL	Chapters 1, 2, 6, 8, 10, 14, 18, 19, 21, 31, 36, 37, 41, 42, 46, 49, 61 and 62
Cristian Abelleira	Chapter 30
DalmTo	Chapter 30
Daniel	Chapter 46

丹尼尔·朗曼	第18章和第24章
dariru	第6章
达留什	第10章和第19章
达雷尔·李	第40章
达伦·巴特鲁普	第18章和第57章
达里尔	第6章、第55章、第56章和第58章
dasblinkenlight	第52章
大卫·曼海姆	第37章
大卫·派恩	第6章
大卫·斯皮莱特	第6章
day_dreamer	第6章
dd4711	第46章
dmfay	第48章
杜尔格帕尔·辛格	第6章
迪兰·范德伯格	第21章和第29章
埃米尔·罗兰	第20章
埃里克·VB	第6章
弗洛林·吉塔	第5章、第18章、第25章、第42章和第47章
飞行派怪兽	第5、6、19、36和37章
辩护	第5和18章
弗兰克·德尔农库尔	第6、18和41章
弗兰克	第7章
模糊逻辑	第46章
加卢斯	第60章
geeksal	第一章
吉迪尔	第45章
金门	第41章
guiguiblit	第9章
H. 保维伦	第21章
黑客	第59章
哈里什·贾纳尼	第11章
哈尔乔特	第50章
斧头	第41章
hellyale	第11章
HK1	第18章
HLGEM	第18章
黄辉	第6章
Horaciux	第37章
海内克·伯纳德	第30章
伊恩·肯尼	第42章
iliketocode	第6章
伊姆兰·阿里·汗	第6、41和42章
印加	第6章
IncrediApp	第55章
贾里德·胡珀	第6章
杰森·W	第24章
JavaHopper	第5章
贾伊迪普·贾达夫	第41章
Jaydles	第6、7、8和10章
Jenism	第37章
杰瑞·杰里迈亚	第45章
吉姆	第24章
乔·塔拉斯	第24章

Daniel Langemann	Chapters 18 and 24
dariru	Chapter 6
Dariusz	Chapters 10 and 19
Darrel Lee	Chapter 40
Darren Bartrup	Chapters 18 and 57
Daryl	Chapters 6, 55, 56 and 58
dasblinkenlight	Chapter 52
David Manheim	Chapter 37
David Pine	Chapter 6
David Spillett	Chapter 6
day_dreamer	Chapter 6
dd4711	Chapter 46
dmfay	Chapter 48
Durgpal Singh	Chapter 6
Dylan Vander Berg	Chapters 21 and 29
Emil Rowland	Chapter 20
Eric VB	Chapter 6
Florin Ghita	Chapters 5, 18, 25, 42 and 47
FlyingPiMonster	Chapters 5, 6, 19, 36 and 37
forsvarir	Chapters 5 and 18
Franck Dernoncourt	Chapters 6, 18 and 41
Frank	Chapter 7
fuzzy_logic	Chapter 46
Gallus	Chapter 60
geeksal	Chapter 1
Gidil	Chapter 45
Golden Gate	Chapter 41
guiguiblit	Chapter 9
H. Pauwelyn	Chapter 21
Hack	Chapter 59
Harish Gyanani	Chapter 11
Harjot	Chapter 50
hatchet	Chapter 41
hellyale	Chapter 11
HK1	Chapter 18
HLGEM	Chapter 18
HoangHieu	Chapter 6
Horaciux	Chapter 37
Hynek Bernard	Chapter 30
Ian Kenney	Chapter 42
iliketocode	Chapter 6
Imran Ali Khan	Chapters 6, 41 and 42
Inca	Chapter 6
IncrediApp	Chapter 55
Jared Hooper	Chapter 6
Jason W	Chapter 24
JavaHopper	Chapter 5
Jaydip Jadhav	Chapter 41
Jaydles	Chapters 6, 7, 8 and 10
Jenism	Chapter 37
Jerry Jeremiah	Chapter 45
Jim	Chapter 24
Joe Taras	Chapter 24

乔尔	第29章和第31章
约翰·奥多姆	第6章、第18章、第22章、第32章和第56章
约翰·斯莱格斯	第6章和第18章
约翰·史密斯	第51章
约翰·L·贝文	第一章
乔乔德莫	第21章
乔恩·陈	第13章
乔恩·埃里克森	第1章和第13章
乔恩·H	第6章
尤尔根·D	第12、13和42章
卡尔蒂凯扬	第28章
凯文·比约克·尼尔森	第41和43章
基兰·库马尔·马塔姆	第21章
克耶蒂尔·诺尔丁	第36章
尼克勒斯	第62章
兰基马特	第6章
L C I I I	第15章
利·里弗尔	第41章
莱克西	第25章
洛希塔·帕拉吉里	第11章
马克·伊安努奇	第6章和第18章
马克·佩雷拉	第6章和第11章
马克·斯图尔特	第43章
马塔斯·瓦伊特凯维休斯	第6、13、14、19、21和41章
马特乌什·皮奥特罗夫斯基	第41章
马特	第5、6和10章
马特·S	第6章
马修·惠特	第6章
毛里斯	第37章
米哈伊	第6和24章
米斯拉·钦塔	第8章和第25章
MotKohn	第10章
Mureinik	第10、18和45章
mustaccio	第6和45章
Mzzzzzz	第5章
Nathan	第42章
nazark	第8章
内莉亚·纳胡姆	第41章
Nunie123	第52章
奥德	第6章
奥仁	第6章和第11章
omini数据	第42章和第44章
onedaywhen	第6章
奥扎伊尔·卡弗雷	第25章
帕拉多	第8章和第37章
保罗·班布里	第30章
保罗·弗雷塔斯	第37章
彼得·K	第42章和第46章
弗朗西斯	第1、3、4、8、11、13、18、19、29、38、41、46、49、52和53章
普拉蒂克	第1、6和21章
普鲁克	第6章
拉西尔·希兰	第6章
拉霍林	第18章

Joel	Chapters 29 and 31
John Odom	Chapters 6, 18, 22, 32 and 56
John Slegers	Chapters 6 and 18
John Smith	Chapter 51
JohnLBevan	Chapter 1
Jojomdo	Chapter 21
Jon Chan	Chapter 13
Jon Ericson	Chapters 1 and 13
JonH	Chapter 6
juergen d	Chapters 12, 13 and 42
Karthikeyan	Chapter 28
Kewin Björk Nielsen	Chapters 41 and 43
KIRAN KUMAR MATAM	Chapter 21
KjetilNordin	Chapter 36
Knickerless	Chapter 62
Lankymart	Chapter 6
LCIII	Chapter 15
Leigh Riffel	Chapter 41
Lexi	Chapter 25
Lohitha Palagiri	Chapter 11
Mark Iannucci	Chapters 6 and 18
Mark Perera	Chapters 6 and 11
Mark Stewart	Chapter 43
Matas Vaitkevicius	Chapters 6, 13, 14, 19, 21 and 41
Mateusz Piotrowski	Chapter 41
Matt	Chapters 5, 6 and 10
Matt S	Chapter 6
Matthew Whitt	Chapter 6
mauris	Chapter 37
Mihai	Chapters 6 and 24
mithra chintha	Chapters 8 and 25
MotKohn	Chapter 10
Mureinik	Chapters 10, 18 and 45
mustaccio	Chapters 6 and 45
Mzzzzzz	Chapter 5
Nathan	Chapter 42
nazark	Chapter 8
Neria Nachum	Chapter 41
Nunie123	Chapter 52
Oded	Chapter 6
Ojen	Chapters 6 and 11
omini data	Chapters 42 and 44
onedaywhen	Chapter 6
Ozair Kafray	Chapter 25
Parado	Chapters 8 and 37
Paul Bambury	Chapter 30
Paulo Freitas	Chapter 37
Peter K	Chapters 42 and 46
Phrancis	Chapters 1, 3, 4, 8, 11, 13, 18, 19, 29, 38, 41, 46, 49, 52 and 53
Prateek	Chapters 1, 6 and 21
Preuk	Chapter 6
Racil Hilan	Chapter 6
raholling	Chapter 18

rajarshig	第26章
拉面厨师	第41章
重启	第42章
Redithion	第11章
里卡多·庞图阿尔	第22章
罗伯特·哥伦比亚	第6章和第41章
瑞安	第37章
瑞安·罗基	第60章
萨罗杰·萨斯马尔	第4章和第6章
希瓦	第5章
西比什·维努	第46章
西蒙·福斯特	第25章
西蒙娜	第7章
模拟体	第16章
索默工程	第6章
SQLFox	第27章
sqluser	第6章
斯坦尼斯洛瓦斯·卡拉šnikovas	第10章
斯特凡·斯泰格	第11、18、33和62章
史蒂文	第35章
斯蒂文	第61章
斯图	第31章
蒂莫西	第6章
tinlyx	第52章
托特·扎姆	第5、13、18、19、26和42章
Uberzen1	第23章
乌梅什	第29章
user1221533	第38章
user1336087	第6章
user2314737	第34章
user5389107	第5章
维克兰特	第11章
vmaroli	第11、19和41章
walid	第4和12章
韦斯利·约翰逊	第5章
威廉·莱德贝特	第42章
冬日战士	第6章
沃尔夫冈	第8章
异种恶魔	第18章和第29章
xQbert	第6章
耶胡达·沙皮拉	第50章
超立方体	第1章和第4章
尤里·费多罗夫	第6章
扎加	第12章
扎希罗·莫尔	第6章和第7章
zedfoxus	第6章
佐伊德	第27章
zplizzi	第26章
eləx	第10章和第41章
阿列克谢·涅乌达钦	第42章
拉胡尔·马克瓦纳	第18章

rajarshig	Chapter 26
RamenChef	Chapter 41
Reboot	Chapter 42
Redithion	Chapter 11
Ricardo Pontual	Chapter 22
Robert Columbia	Chapters 6 and 41
Ryan	Chapter 37
Ryan Rockey	Chapter 60
Saroj Sasmal	Chapters 4 and 6
Shiva	Chapter 5
Sibeesh Venu	Chapter 46
Simon Foster	Chapter 25
Simone	Chapter 7
Simulant	Chapter 16
SommerEngineering	Chapter 6
SQLFox	Chapter 27
sqluser	Chapter 6
Stanislovas Kalašnikovas	Chapter 10
Stefan Steiger	Chapters 11, 18, 33 and 62
Steven	Chapter 35
Stivan	Chapter 61
Stu	Chapter 31
Timothy	Chapter 6
tinlyx	Chapter 52
Tot Zam	Chapters 5, 13, 18, 19, 26 and 42
Uberzen1	Chapter 23
Umesh	Chapter 29
user1221533	Chapter 38
user1336087	Chapter 6
user2314737	Chapter 34
user5389107	Chapter 5
Vikrant	Chapter 11
vmaroli	Chapters 11, 19 and 41
walid	Chapters 4 and 12
WesleyJohnson	Chapter 5
William Ledbetter	Chapter 42
wintersolider	Chapter 6
Wolfgang	Chapter 8
xenodevil	Chapters 18 and 29
xQbert	Chapter 6
Yehuda Shapira	Chapter 50
ypercube□□	Chapters 1 and 4
Yury Fedorov	Chapter 6
Zaga	Chapter 12
Zahiro Mor	Chapters 6 and 7
zedfoxus	Chapter 6
Zoyd	Chapter 27
zplizzi	Chapter 26
eləx	Chapters 10 and 41
Алексей Неудачин	Chapter 42
Рахул Маквана	Chapter 18

你可能也喜欢

CSS

Notes for Professionals




200+ pages
of professional hints and tricks

GoalKicker.com
Free Programming Books

HTML5

Notes for Professionals



100+ pages
of professional hints and tricks

GoalKicker.com
Free Programming Books

JavaScript

Notes for Professionals



400+ pages
of professional hints and tricks

GoalKicker.com
Free Programming Books

You may also like

CSS

Notes for Professionals




200+ pages
of professional hints and tricks

GoalKicker.com
Free Programming Books

HTML5

Notes for Professionals



100+ pages
of professional hints and tricks

GoalKicker.com
Free Programming Books

JavaScript

Notes for Professionals

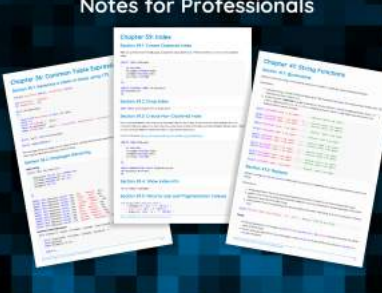


400+ pages
of professional hints and tricks

GoalKicker.com
Free Programming Books

Microsoft SQL Server

Notes for Professionals



200+ pages
of professional hints and tricks

GoalKicker.com
Free Programming Books

MongoDB

Notes for Professionals




60+ pages
of professional hints and tricks

GoalKicker.com
Free Programming Books

MySQL

Notes for Professionals



100+ pages
of professional hints and tricks

GoalKicker.com
Free Programming Books

Microsoft SQL Server

Notes for Professionals



200+ pages
of professional hints and tricks

GoalKicker.com
Free Programming Books

MongoDB

Notes for Professionals




60+ pages
of professional hints and tricks

GoalKicker.com
Free Programming Books

MySQL

Notes for Professionals



100+ pages
of professional hints and tricks

GoalKicker.com
Free Programming Books

Oracle Database

Notes for Professionals



100+ pages
of professional hints and tricks

GoalKicker.com
Free Programming Books

PostgreSQL

Notes for Professionals



60+ pages
of professional hints and tricks

GoalKicker.com
Free Programming Books

PHP

Notes for Professionals



400+ pages
of professional hints and tricks

GoalKicker.com
Free Programming Books

Oracle Database

Notes for Professionals



100+ pages
of professional hints and tricks

GoalKicker.com
Free Programming Books

PostgreSQL

Notes for Professionals



60+ pages
of professional hints and tricks

GoalKicker.com
Free Programming Books

PHP

Notes for Professionals



400+ pages
of professional hints and tricks

GoalKicker.com
Free Programming Books