第21章 决策支持

21.1 引言

注意：本章的原作者为David McGoveran。

决策支持系统是用于对企业信息进行辅助分析的系统。系统的目的是帮助管理者“发现趋势、查明问题并进行智能化的决策”[21.7]。这种系统的来源——业务的研究、管理的行为和科学理论以及统计处理控制，等等，可以追溯到20世纪40~50年代，那时计算机还没有被广泛使用。基本思想是从企业操作型数据中收集并精简信息，使之能用来分析企业行为，并智能地改变这种行为。当时的客观条件下人们只能将数据精简到极小，常常会只比简单的摘要报告稍多一点。

到了60年代后期和70年代早期，哈佛大学和MIT的研究人员促进了计算机在决策处理中的应用[21.23]。开始时这种应用局限于自动化报表生成，虽然当时已经具备了某些基本的分析能力[21.2~21.3,21.6,21.26]。这些早期的计算机系统被称为管理决策系统；后来发展为管理信息系统。不过我们更愿意使用现代的术语——决策支持系统，因为管理信息系统的概念过于抽象，所有的信息系统（包括OLTP——联机事务处理系统）都可以被称为“管理信息系统”。

在70年代开发出了许多查询语言，围绕这些查询语言建立了许多特定的决策支持系统，使用报表生成器（如RPG）或数据获取工具（如Focus、Datatrieve和NOMAD）来实现。这些系统首次让具有一定技能的最终用户直接访问计算机数据存储；也就是说，让这些用户在数据存储上阐明商务相关查询，并直接执行这些查询，而无需IT技术人员的协助。

当然，上面提到的数据存储一般是一些简单文件——那时大多商业数据保存在这种文件中，后来保存在非关系型数据库中（当时关系型数据库正在研究之中）。即便在后来，人们也常常需要在数据库中将数据提取出来，然后拷贝到文件中以便被决策支持系统使用。直到80年代，关系型数据库才开始取代简单文件在决策支持系统中的位置。实际上，决策支持、特定查询和报表都出现于关系技术在商业上的早期应用中。

尽管SQL产品现在已经得到了广泛的应用，抽取处理（extract processing）的思想——即从操作型环境中拷贝数据到其他环境——仍然是很重要的；它允许用户按自己的意愿来操作这些抽取出来的数据，而无须和操作型环境打交道。显然，在决策支持中需要经常进行这种抽取工作。

需要注意的是，决策支持实际上并不属于数据库技术，它只是数据库技术的应用，更准确地说，它是由多个这种应用所组成，这些应用彼此独立而又相互关联。目前正在研究的相关问题包括数据仓库、数据集市、操作型数据存储、联机分析处理（OLAP）、多维数据库和数据挖掘等，我们将在以后的章节中予以讨论。

说明：我们很快会注意到在以上的这些领域中存在一个共同点，就是很少会用到良好的逻辑设计原则。决策支持经常是相当特定的应用，此时更需要多考虑物理实现中的细节，对

逻辑设计则不会要求那么严格。由于有这些因素，本章中使用 SQL，而不是Tutorial D，作为示例的基础。同时使用“更模糊的”SQL术语，如行、列和表，而不是元组、属性、关系值和关系变量。另外还使用逻辑模式和物理模式，作为第2章中的概念模式和内模式的同义词。

本章的各节是这样安排的：在21.2节中讨论决策支持的某些特征，这些特征容易引起误导的设计。在21.3节中给出处理这些问题的方法。在21.4节对数据准备（将操作型数据转换为可供决策支持使用的形式）的结果进行检验，同时简要介绍“操作型数据存储”。在21.5节中讨论数据仓库、数据集市和“多维模式”。在21.6节中讨论联机分析处理（OLAP）和多维数据库。在21.7节中讨论数据挖掘。在21.8节中给出本章小结。

21.2 决策支持的特征

决策支持数据库具有一些特征，最主要的一点是：这种数据库主要（并不是完全）是只读的。更新操作一般仅限于周期性的上载和刷新操作（即偶尔的 INSERT-DELETE操作，几乎没有UPDATE操作）。注意：有时会更新某些辅助的工作表，但是一般的决策支持处理基本上不会更新决策支持数据库本身。

决策支持数据库中还有一些值得关注的特征（将在 21.3节中予以详细阐述），前三个为逻辑特征，后三个为物理特征。

- 列常常在组合中使用。
- 一般不考虑完整性约束（假定数据在载入时就是正确的，以后不再更新）。
- 码中常常包含时间成分（见第22章）。
- 数据库会变得很庞大（特别是随着时间推移，企业事务[⊖]细节数据会不断增加）。
- 数据库中建立索引的负担非常繁重。
- 数据库中包含多种受控冗余（controlled redundancy）（见第1章）。

决策支持查询也有自己的特征，特别是它们会变得相当复杂。复杂性中包括：

- 布尔表达式复杂性：决策支持查询的WHERE子句经常会涉及复杂表达式——难以书写、难以理解而且系统也难以适当处理这些查询（特别是对于传统优化器，因为它们只能对有限的几种访问策略进行评估）。一个常见的问题是涉及时间的查询；现有的系统一般不能很好地查询出在指定时间间隔中最大时间戳所在的行（见第 22章）。如果查询中还包括连接操作，那就会变得非常复杂。这时的性能无疑是很差的。
- 连接复杂性：决策支持查询常常需要访问多种事实。在完全规范化设计的数据库中，这种查询将涉及大量连接操作。不幸的是，连接处理技术无法满足不断增长的决策支持查询需求[⊜]。因此设计人员会对数据库进行逆规范化，预连接（prejoining）某些表。但是，正如第12章中所提到的，这种方法很难奏效，经常是导致另外的问题。而且这种避免连接的期望使得我们无法有效地使用关系操作，当从数据库中检索大量数据时，连接处理必须在应用中完成，而不是由DBMS来完成这项工作。

⊖ 在本章和以后章节中我们将企业事务(如产品的销售)与第四部分中的事务区分开来。

⊜ 本文作者(McGoveran)在1981年开发了早期的决策支持系统，他发现三个中等大小的表进行连接操作也需消耗几个小时，而四个至六个表的连接更加让人无法忍受。现在，六个到十个大表的连接是很常见的。但是在查询中出现更多表的连接依然是个技术上难以处理的问题。超过十二个表连接的查询会是一个冒险——而这种查询又很可能是必需的。

- 函数复杂性：决策支持查询经常会涉及到统计函数和其它数学函数。很少有产品支持这些函数。因此查询常常被划分为一系列子查询，然后在其中插入一些用户编写的过程来进行所需的计算。这种方法的缺点是需要检索大量的数据，从而导致整个查询变得难以编写和理解。
- 分析复杂性：企业问题很少能在单个查询中予以解答。一方面因为用户难以编写特别复杂的查询，另一方面是因为SQL的实现中存在某些限制，从而无法处理这种查询。解决方法之一是将这种查询划分为一系列子查询，并将中间结果保存在辅助表中。

以上列举了决策支持数据库和决策支持查询的特征，讨论的重点集中于有关性能的设计——特别是批插入和特定检索的性能。不过这种情况只应该影响到数据库的物理设计，而非逻辑设计（在下一节详细阐述）。不幸的是，如同在21.1节中提到的，决策支持系统的销售商和用户常常无法正确区分逻辑和物理问题[⊖]，实际上他们经常会完全放弃逻辑设计。这样的后果是，为了处理上面所讨论的不同特征导致了特殊化，并且在平衡正确性、可维护性、性能、缩放性和可用性需求时会遇到难以克服的困难。

21.3 决策支持的数据库设计

在本书的前面部分（第三部分引言）中已经陈述了我们的观点，即数据库设计至少应分为两个阶段，即逻辑设计和物理设计：

a. 首先应该进行逻辑设计。在这个阶段，重点放在关系正确性（relational correctness）上。表要表达适当的关系，从而保证关系操作的正确执行。要指定域（类型），然后在域上定义列，这样列之间的依赖关系（比如FD）就可以识别出来。最后进行规范化，定义完整性约束。

b. 然后根据逻辑设计来进行物理设计。在这个阶段，重点放在存储效率和性能上。从理论上讲，任何物理方案都是可行的，只要在逻辑和物理模式之间存在一种保持信息的变换，能用关系代数予以表达[2.5]。正是由于这种变换的存在，保证了在物理模式中存在一些关系型视图，使得它看起来与逻辑模式相似，反之亦然。

当然，逻辑模式会不断变化（例如，为了适应新的数据或新发现的依赖关系），从而要求物理模式作相应的变化。我们关注的是当修改物理模式的时候，能否保持逻辑模式不变。例如，假设表SP（发货）和P（零件）之间的连接是目前的主要访问模式，我们准备在物理层“预连接”表SP和P，从而降低I/O和连接开销。但是，如果要实现物理数据独立性，逻辑模式就应该保持不变（当然，如果我们希望得到性能提升，就必须让查询优化器知道“预连接”的存在，并适当地使用它）。此外，如果以后访问模式改为只访问单个表而不是多表连接，我们应该能够再次改变物理模式，使得表SP和P在物理上分离，并且不会影响到逻辑层。

以上的阐述表明，保持物理数据独立性的问题基本上就是支持视图更新的问题。正如在第9章中提到的，在理论上所有的关系视图都是可更新的。同样，在理论上，如果从逻辑模式导出满足以上阐述的物理模式，就可以实现最大的物理数据独立性：任何在逻辑模式中的更新都可以自动转换为相应的物理模式中的更新，反之亦然。但是物理模式的改变并不会要求改变逻辑模式。注意：改变物理模式的唯一原因只能是出于存储效率和性能上的考虑。

⊖ 数据仓库和OLAP专家应该为此负责；他们常常认为对于决策支持系统无须进行关系模型设计，声称关系模型不能表达数据，并因此避开关系模型设计。这种看法导致了无法将关系模型从物理模型中分离。

但是，不幸的是，现有的SQL产品不能正确地支持视图更新。因此，在这些产品中就只能考虑可允许的物理模式。具体地说，如果（a）将逻辑层的基表看作“视图”，将其在物理层的存储看作基表；那么（b）物理模式必须满足：在这些“视图”上所有可能的逻辑更新都应该更新相应的“基表”。注意：实际上，我们可以使用存储过程、触发器和中间件来模拟正确的视图更新机制。但是，这些技术不在本章的讨论范围之内。

1. 逻辑设计

逻辑设计的规则与数据库的用途无关——对于不同的应用使用相同的规则。因此，在OLTP或决策支持应用中的数据库逻辑设计没有什么不同：它们都采用相同的实际步骤。现在回顾一下第21.2节中提到的决策支持数据库的三个逻辑特征，以及它们对逻辑设计所产生的影响。

- 列的组合和更少的依赖

  决策支持查询——可能还有更新——常常把列的组合视为一个整体，组合中的列不会被单独访问（地址就是一个很明显的例子）。称这种列的组合为复合列。从逻辑设计中视图的观点来看，这种复合列可以被视为没有组合的简单列。更具体地说，令CC为复合列，C是同一个表中的简单列，那么C和CC中的组件之间的依赖关系实际上可归纳为C和CC之间的依赖关系。进一步来说，仅涉及CC中的组件之间的依赖关系是不相关的，可以被忽略。这样依赖关系的总数会减少，逻辑设计变得更简单，涉及到的列和表的数目也会减少。

  注意：支持复合列的数据库系统并不太多，关键在于它是否支持用户自定义类型。参见参考文献[21.11]和第5章、第25章。

- 一般的完整性约束

  由于前文阐述的原因：（a）决策支持数据库主要是只读的；（b）当数据库中数据上载或刷新时进行数据完整性检验。大家可能会认为在逻辑模式中无须声明完整性约束。实际上并不是这样的。就算数据库真的是只读的，可以保证完整性约束，但是约束中还包含了不可忽略的语义信息。第8章（第8.10节）中提到，约束是用来定义表和整个数据库的语义。约束的声明告诉用户数据的含义，从而帮助他们准确描述查询。同时约束的声明还可以向优化器提供至关重要的信息（参见第17章17.4节关于语义优化的探讨）。

  注意：当在一些SQL产品中声明某些约束时，会自动创建索引和其它的强制机制，从而增加数据上载和刷新的开销。设计人员也因此避免约束的声明。但是，这个问题是由于混淆了逻辑和物理问题：完整性约束的声明可以在逻辑层单独指定，在物理层指定相应的强制机制。可惜现在的SQL产品还不能将逻辑层和物理层完全分离。

- 时态码

  操作型数据库一般只涉及当前数据。决策支持数据库一般涉及历史数据，因此会给几乎全部数据盖上时间戳。这样在决策支持数据库的码中常常包含时间戳列。例如，在前面提到的供应商-零件数据库，假如需要显示每次发货时的月份。发货表应该如图21-1所示。新增的列MID（“month ID”）加入到表SP的码中。关于表SP的查询中也应指定时间戳。在第21.2节中会给出简略描述，第22章中进行了深入讨论。

  注意：为了在码中加入时间戳列可能需要重新设计数据库。例如，假设每次发货的数量是由发货的月份所决定的（图21-1中的示例数据符合这种约束）。在表SP中就存在函数依赖MID→QTY，因此它就不符合第五范式，需要进一步规范化为图21-2中的形式。不

幸的是，决策支持设计人员很少会关心这种传递依赖。

SP

S#	P#	MID	QTY
S1	P1	3	300
S1	P1	5	100
S1	P2	1	200
S1	P3	7	400
S1	P4	1	200
S1	P5	5	100
S1	P6	4	100
S2	P1	3	300
S2	P2	9	400
S3	P2	6	200
S3	P2	8	200
S4	P2	1	200
S4	P4	8	200
S4	P5	7	400
S4	P5	11	400

图21-1 表SP中的示例数据，包含month IDs

SP

S#	P#	MID
S1	P1	3
S1	P1	5
S1	P2	1
S1	P3	7
S1	P4	1
S1	P5	5
S1	P6	4
S2	P1	3
S2	P2	9
S3	P2	6
S3	P2	8
S4	P2	1
S4	P4	8
S4	P5	7
S4	P5	11

MONTH_QTY

MID	QTY
1	200
2	600
3	300
4	100
5	100
6	200
7	400
8	200
9	400
10	100
11	400
12	50

图21-2 对图21-1规范化以后的结果

2. 物理设计

在第21.2节中提到决策支持数据库趋向很大，索引任务繁重，而且涉及多种受控冗余(controlled redundancy)。下面详细讨论物理设计问题。

首先考虑分区（也称为分片）。分区是处理“大型数据库”的一种方法；基于物理存储的考虑，它将给定的表划分为不相交的分区或分片（参见第20章中关于分片的描述）。分区可以有效地提高被查表的可管理性和可用性。一般给每个分区分配一定的专用资源（如磁盘空间、CPU），并尽量减少分区之间的资源竞争。按照分区函数将表垂直划分[⊖]，以选取列的值（分区的码）作为参数，返回分区号或地址。这种函数一般支持区域、哈希和循环分区（参见第17章中的参考文献[17.58]的注释）。

现在来考虑索引。众所周知，使用合适的索引可以显著减少I/O的负担。大多数早期的SQL产品只提供一种索引，即B树，不过现在出现了更多的索引技术，特别是在决策支持数据库中，包括位图、哈希、多表、布尔和函数型索引。下面简略地进行说明：

- B树索引：B树索引能提高区域查询的效率（除非要访问的行数过多）。对B树的更新也相当有效。
- 位图索引：假设表T（已建索引）中有n行，C是T中的列，C上的位图索引是一个n位的向量，与C所在值域中的每一个值对应。根据第R行的C列的值来设置索引相应的位。当

⊖ 垂直分片虽然有很多好处，却不常用，因为大多数产品不支持它。

值域比较小时可以明显提高查询效率。一些关系运算（连接、并、相等约束等）可以通过对索引进行简单的位运算（与、或、非）即可得到。只有当获取最终的检索结果时才会访问表中的实际数据。

- 哈希索引：使用哈希索引可以提高访问特定行（不是某些区域）的效率。只要哈希函数不需要其他的键值，计算的开销只会随着行数的增加线性增长。哈希技术也可以提高连接效率，参见第17章。
- 多表索引（连接索引）：一般说来，多表索引中包含了指向多表中的记录的指针，而不是只限于单个表。多表索引可以提高表连接的效率，并检验多表（即数据库）完整性约束。
- 布尔索引（表达式索引）：布尔索引显示表中的哪些行对于特定的布尔表达式为真。当布尔表达式是某些约束条件的公共部分时，布尔索引会很有用。
- 函数型索引：函数型索引不是简单的行值的索引，而是这些行值对应的特定函数值的索引。

除了上述的这些索引，有人还提出了混合索引（由上述索引组合而成），很难将它归类。另外还有一些专用索引（如R树，用于处理几何数据）。本书中不讨论这些索引，详细介绍参见[12.27]。

最后来讨论受控冗余，受控冗余用来减少I/O和资源争用。第1章中已经提到，这种冗余是由DBMS管理而对用户透明（注意：冗余是完全在物理层而不是逻辑层被控制，因此不会影响逻辑层的正确性）。受控冗余分为两类：

- 第一种涉及到原始数据的精确拷贝或复制品(replicas)。注意：此外同样存在着非精确的复制，即拷贝管理。
- 第二种涉及到除原始数据外的导出数据，一般形式为汇总表或计算（或导出）列。

下面分别进行讨论。

在第20章中已经介绍了关于复制（replication）的基本概念（第20.3节和第20.4节，特别在第20.4节中的“更新传播”小节中予以详细阐述）；这里只重复讨论中的几个要点并予以评论。首先要提及的是复制可以是同步的，也可以是异步的。

- 在同步的情况下，如果某个给定复制品被更新了，那么包含相同数据的其它复制品也应该在同一个事务中被更新，因此理论上讲只能存在数据的一个版本。大多数产品是使用触发器程序来实现同步复制，这种实现可能是隐含地由系统管理。不过，同步复制有一个缺点，当更新任何一个复制品时给所有的事务增加了开销（这样可能导致可用性）。
- 在异步的情况下，对某个复制品更新后，会在以后将这种更新传播给其它的复制品，而不要求在同一个事务内全部更新。因此，在异步复制中就有时间延迟或等待时间的概念，在这个时间间隔内，复制品之间可能会不一致（当然复制品这个术语也就不是很贴切了）。大多数产品采用读取事务日志或需传播的稳定更新队列的方法来实现异步更新。[⊖]

异步更新的好处是将复制的开销从更新事务中分离出来，这种事务一般是“关键任务”

[⊖] 同时要注意到复制品的不一致性可能会很难发现和避免。特别地，更新操作的顺序也会引起冲突。例如，事务T1要在复制品RX中插入一条记录，事务T2随后删除这条记录，RY是RX的复制品。如果更新操作被传播到RY，但次序颠倒(例如因为路由延迟)，T2发现没有需要删除的记录，而T1随后插入了这条记录。结果就是RY中包含了这条记录，而RX中却没有。冲突管理和强制一致性是难题，超出了本书讨论的范围。

型事务或对性能有较高要求的事务。而它的缺点则是数据的不一致性，可以在逻辑层看到这种冗余，严格地说，此时“受控冗余”这个术语也就不是很贴切的了。

我们发现，至少在商业领域，“复制”主要指的是异步复制（参见第21章）。

复制和拷贝管理（copy management）之间的根本区别是：在复制中，对某个复制品的更新会“自动”传播给其它所有的复制品。相反地，在拷贝管理中没有这种自动传播；数据的拷贝是由一些批处理或后台进程来创建和管理的，这些进程与更新事务在时间上是分离的。拷贝管理一般比复制效率高，因为可以同时拷贝大量数据，缺点则是拷贝在大多数时间都与基数据不一致。实际上，用户一般要知道数据是什么时候被同步的。拷贝管理常常被简化，以保证对某些“主要拷贝”模式的一致性更新（参见第20章）。

我们要考虑的其它冗余是计算列和汇总表，它们在决策支持环境中非常重要，用来保存预先计算的数值（根据数据库中的其它数据计算出来），这样，在查询的时候就无须重复计算。计算列的值是从同一条记录中的其它列计算得到[⊖]，汇总表是对其它表的聚集运算（求和、平均值、计数，等等）结果，聚集运算常常是在相同细节数据的不同分组上进行预计算（参见第21.6节）。注意：如果计算列和汇总表真的属于受控冗余的范畴，就应该是对用户透明的，可是在现在的产品中却不是这样的。

汇总表和计算列一般通过系统管理的触发器程序来实现，当然也可以由用户编写的程序实现。第一种方法可以保证基数据和导出数据之间的一致性，而第二种方法则很可能导致不一致性。

3. 常见设计错误

在本小节中，我们评论决策支持环境中常见的一些设计错误：

- 重复行：决策支持设计人员经常说他们的数据没有唯一的标识符，因此允许重复。参考文献[5.3]和[5.6]中详细阐述了为什么重复行的存在是个错误；其实导致这种错误出现的原因是因为物理模式并不是从逻辑模式中导出，甚至根本就没有设计逻辑模式。而且在这种设计中，行经常缺乏同义的语义——也就是说，它们根本不是同一个断言的实例（参见第3章3.4节和第18章）。注意：有时会故意允许重复行的出现，特别是当设计人员具有面向对象的知识背景时（参见第24章24.2节）。
- 逆规范化：出于消除连接运算和减少I/O的考虑，设计人员常常会预连接表，引入各种导出列，等等。这种做法可以在物理层出现，但不应该出现在逻辑层中。
- 星型模式：“星型模式”（多维模式）是试图“短路(short-circuit)”正常设计技术的结果，实际上弊大于利。当数据增长时会影响性能和灵活性，而且通过物理层重新设计来解决这种问题又会导致应用的改变（因为星型模式是物理模式）。注意：在下面的第21.5节中予以详细探讨。
- 空值：设计人员经常希望通过允许列中出现空值来节省空间（如果列中包含的是变长数据类型，而且产品中在物理层通过清空列的方法来实现空值的话，也许可以节省空间）。但是可以通过良好的设计来避免空值，并能够提供更好的存储效率和更好的I/O性能。
- 汇总表的设计：人们一般不去考虑汇总表的逻辑设计问题，因此导致了失控冗余而且难以维护数据的一致性，用户不能明白汇总表的真正语义，无法正确阐述所需的查询。为

[⊖] 换句话说，计算值可以从同一个表或不同表的多行数值中层出。然而，这也意味着要保证同步更新，对载入和刷新操作可能会产生负作用。

了避免这种问题，所有属于相同聚集层次的汇总表应该被设计为构成了自己的数据库。通过（a）禁止跨聚集层次的更新；和（b）总是从细节数据中聚集来更新汇总表，可以避免一些循环更新的问题。

- “多重导航路径”：决策支持设计人员和用户经常错误地说可以通过“多重导航路径”得到所需数据，相同的结果可以通过不同的关系表达式得到。有时这些表达式是等价的，例如A JOIN (B JOIN C)和(A JOIN B) JOIN C（参见第17章）；有时是由完整性约束来保证它们等价（同样参见第17章）；但有时它们根本不等价！假设表A、B和C都有公共列K，那么“通过K路径从A到B然后到C”一般上不会等价于“直接通过K路径从A到C”。

很显然，这些错误会让用户迷惑，不知道应该如何表述要求，是否能得到所需的查询结果。要解决这些问题，可以对用户进行适当培训，使优化器正常工作，但是因为某些问题出自于设计人员允许在逻辑层出现冗余，并允许用户直接访问物理模式，所以要通过正确的设计来解决。

总的来说，由决策支持需求引起的很多设计难题可以依据设计原则予以解决。实际上，也是因为没有遵循设计原则才会出现这些问题。

21.4 数据准备

围绕决策支持的很多问题首先是关于数据的获取和准备工作。从不同数据源中提取数据，进行清洗、转换和合并整理，载入到决策支持数据库中，然后进行周期性的刷新。每一步操作都会涉及到各自的考虑事项[⊖]。我们将检视各个步骤，并在本节末尾简略讨论操作型数据存储（operational data store）。

1. 提取

提取是从操作型数据库和其它数据源中捕获数据的过程。很多工具可以用于辅助这项工作，包括系统提供的实用程序、定制的提取程序和商业的提取产品。提取过程要进行大量的I/O操作，可能会影响到关键任务的处理，因此一般是在物理层进行并行化操作。但是这种“物理提取”可能会丢失信息——尤其是联系信息——用物理方式表达的信息（比如指针或物理毗连性），从而导致后续处理上的问题。出于这方面的考虑，提取程序有时通过引入连续的记录号和更换影响外码的指针来保留这种信息。

2. 清洗

很少有数据源能充分保证数据质量，所以当数据进入决策支持数据库之前要进行清洗（一般成批清洗）。典型的清洗操作包括填充遗失的数据、纠正印刷错误和数据条目错误、建立标准的缩略语和格式、使用标准标识符来替代同义词，等等。无法清洗的错误数据将被驳回。注意：根据清洗过程得到的信息有时可以用来识别数据错误的原因，从而不断提高数据质量。

3. 转换和合并

在清洗过以后，还需要进行合适的转换，才能使数据满足决策支持的需要。决策支持中所需的数据形式为一组文件，各个表在物理上要区分开，这样在转换时就要求切分或合并数

⊖ 我们在前面提到，这些操作可以利用关系系统的集合特性，尽管实际上很少利用。

据源中的记录（参见第1章1.5节）。注意：在转换时可能会发现清洗时未发现的数据错误，这些错误数据同样也被驳回（如前所述，这些信息也可以用来提高数据源的数据质量）。

当需要对多个数据源进行合并时，转换就显得非常重要，称之为合并过程。这时，数据间的任何隐式联系都要被显式表示（通过引入显式的数据）。另外，还要维护和商务相关的日期时间信息，将数据源关联起来，此过程称之为“时间同步”。

出于性能上的考虑，转换操作常常是并行处理的，需要大量的I/O和CPU开销。

注意：时间同步是个难题。例如，假设我们要获取每个季度各个售货员的客户收入。客户和收入数据按财政季度保存在会计数据库中，而售货员和客户数据按日历季度保存在销售数据库中。很显然，需要将两个数据库合并起来。客户数据的合并很简单——只需匹配客户编号即可。然而时间同步则很困难，我们可以找到每个财政季度的客户收入，但却不知道此时是哪个售货员和这位客户打交道，也根本不知道每个日历季度中的客户收入。

4. 载入

DBMS销售商很注重载入操作的效率。“载入操作”包括：（a）将转换和合并过的数据移入决策支持数据库中；（b）检验一致性（integrity checking）；（c）建立必需的索引。下面对各个步骤作简略阐述：

a. 移动数据：现代的系统一般提供并行载入功能。有时在载入之前要预格式化数据为目标数据库中规定的内部物理格式（一种高效的替代技术是将数据载入到目标模式的镜像工作表中，在这些工作表上作完整性检验——参见下面b——然后成批地从工作表移动到目标表中）。

b. 完整性检验：大多数完整性检验工作可以在数据载入之前进行而无须涉及已经载入数据库中的数据，但是，某些约束检验必须涉及到已存入数据库中的数据；例如，唯一性约束就必须在载入时进行检验（或者在载入完毕后成批检验）。

c. 建立索引：索引的存在会明显减缓数据载入的速度，因为在大多数产品中每插入一条新记录就会对索引进行更新。因此，在载入前删除索引，载入后再重建索引有时不失为一个好主意。当新数据的数量相对已有数据的数量较小时就不值得这么做，因为创建索引的开销并不与表的大小成正比。而且，创建大索引可能会引起不可恢复的分配错误，索引越大就越可能出现这种错误。注意：大多数DBMS产品支持并行创建索引，以加速载入和索引创建的过程。

5. 刷新

大多数决策支持数据库（不是全部）要求周期性地刷新以保证数据的及时性。尽管某些决策支持应用要求全部删除所有数据然后再载入，一般在刷新中还是只涉及部分数据的载入。刷新与载入类似，但可能会在用户访问数据库时同时进行。参见第9章9.5节，以及参考文献[9.2]和[9.6]。

6. 操作型数据存储

操作型数据存储（ODS）是“面向主题的、集成的、可变的（可更新的）、最近的或几乎最近的数据集合”[21.19]。换句话说，它是一种特殊的数据库。术语“面向主题”表示数据属于特定的主题范围（例如，顾客、产品等）。操作型数据存储可用于（a）对提取的操作型数据进行物理重组，（b）提供操作型报表；以及（c）支持操作型决策。它同时还可以作为（d）一个合并点，如果操作型数据来自多个数据源。因此，ODS是多用途的。注意：由于

ODS中不积累历史数据，所以一般不会太大；换句话说，它们一般会经常或连续地使用操作型数据进行刷新[⊖]。因为刷新的频率很快，所以也解决了时间同步问题（参见上面的“转换和合并”小节）。

21.5 数据仓库和数据集市

操作型系统一般要求高性能、可预知的工作量、较小处理单元和高可用性。而决策支持系统一般有变化的性能需求、不可预知的工作量、较大处理单元和不稳定的可用性。这些差别使得很难将操作型处理和决策支持处理合并到一个系统中，特别是当需要考虑功能规划、资源管理和系统性能调优的时候。出于以上原因，操作型系统的管理人员不愿意在其系统上进行决策支持，这样就出现了双系统（dual-system）方法。

注意：事情并不总是这样；早期的决策支持系统实际上就存在于操作型系统之中，在低优先级或“批处理窗口”中运行。如果有足够的资源，这样做有很多好处——避免了额外的数据拷贝、重新格式化以及转换的开销。实际上，操作型事务和决策支持事务的集成正被日益重视，不过这个问题超出了本章讨论的范围。

在编写本章时，一般要从多个操作型系统（常常是分离的）中采集决策支持数据，然后存储在自己的数据存储中，称之为数据仓库。

1. 数据仓库

类似操作型数据存储（也类似数据集市——参见下一小节），数据仓库是一种特殊的数据库。这个术语来源于80年代后期[21.13, 21.17]，或者更早。参考文献[21.18]中将数据仓库定义为“面向主题的、集成的、稳定的、随时间变化的数据存储，用于决策支持”（稳定的含义是数据在插入后不再改变，但可以删除）。数据仓库出现的起因有两个：首先，决策支持中需要一个单一的、纯净的、一致的数据源；其次，无须变动操作型系统。

根据定义，数据仓库的工作量就是决策支持的工作量，因此是查询密集的（有时也会进行密集的批插入操作）；同样，数据仓库本身会不断变大（常常超过500GB，每年增长50%）。因此性能调优尽管可能，也显得很困难。可缩放性也是一个问题。关于这些问题的研究包括（a）数据库设计错误（参见第21.3节）；（b）仅用关系操作无法满足需要（参见第21.2节）；（c）在关系模型下的DBMS实现有很多缺点；（d）DBMS本身缺少可缩放性；（e）限制了功能和平台可缩放性的结构设计错误。（d）和（e）超出了本章讨论的范围，（a）和（b）已经在本章中予以讨论，（c）在本书的其它章节讨论。

2. 数据集市

数据仓库一般用来为决策支持提供单一的数据源。但是，自从90年代初数据仓库开始流行以来，逐渐意识到用户常常只是在数据仓库中某个相对较小的主题领域内生成报表或分析数据。实际上，用户偏向于当这个主题领域内的数据刷新时重复同样的操作。而且，其中的一些操作——例如，预测分析、模拟、建立“如果……会怎么样”的模型——会创建新的模式和数据，并进一步更新这些新数据。

在整个数据仓库的同一个子集上重复执行这些操作显然效率较低，根据用途来裁剪数据仓库，建立一些有限的、专用的“仓库”看来是个好主意。在某些情况下，为了加快访问数

⊖ 从操作型数据源到ODS的异步复制有时被用于此用途。这种情况下，数据一般可以在几分钟内刷新。

据的速度，需要能够直接从本地数据源中提取和准备所需数据，而无须等待全部数据都被载入数据仓库后再进行同步。由此产生了数据集市的概念。

现在对于数据集市还没有统一的定义，我们给出的定义是“特定的、面向主题的、集成的、可更新的、随时间变化的数据存储，用于支持特定子集的管理决策”。数据集市与数据仓库之间的主要区别在于它是特定的和可更新的。特定的含义是它只支持特定领域的商务分析；可更新的含义是用户可以更新数据，甚至可以创建新数据（新表）。

建立数据集市有三种方法：

- 直接从数据仓库中提取数据——在整个决策支持工作量上采用“分而治之”的方法，以达到更好的性能和可缩放性。提取出来的数据被载入到数据库中，数据库的物理模式类似于数据仓库的某个合适的子集。不过根据数据集市的“特定的”这一性质，可以进行简化。
- 尽管采用数据仓库是为了提供“单一的控制点”，还是可以建立独立的数据集市（即不从数据仓库中提取数据）。当出于某些原因（金融上的、操作上的或政治上的原因，或者数据仓库还没有建立）无法访问数据仓库时，可以采用这种方法。
- 有时采用“首先建立数据集市”的方法，此时必须建立数据集市，而整个数据仓库则是作为对不同数据集市的合并。

后两种方法都可能导致语义不匹配的问题。独立的数据集市很容易出现这种问题，由于数据库的设计是彼此独立的，所以无法检查语义的匹配。为了避免无法将数据集市合并到数据仓库中，要求（a）首先建立单一的数据仓库逻辑模型，（b）从数据仓库模式中导出各个数据集市的模式（当然可以扩充数据仓库模式，将新数据集市中的主题内容包含进来）。

关于数据集市的设计：设计决策支持数据库时的一项重要决定是数据库的粒度。粒度是数据聚集并保存在数据库中的最低层次。现在的大多数决策支持应用迟早会访问细节数据，这在数据仓库中很容易做到，而在数据集市中则比较困难。如果不是经常访问细节数据，那么从数据仓库中提取细节数据并存储到数据集市就是低效率的工作。另外也很难决定实际所需的最低聚集层次。这时可以在数据集市中只保存聚集数据，而当需要访问细节数据时从数据仓库中提取。同时不会对数据完全聚集，因为那样会产生大量的汇总数据，在第 21.6节中将详细讨论这个问题。

进一步的观点：因为数据集市用户常常采用特定的分析工具，数据集市的物理设计一般要符合特定工具的需要（参见第 21.6节中“ROLAP与MOLAP”的探讨）。这时一般采用“多维模式”，它无法遵循优良的关系设计规范。

3. 多维模式

假设现在我们希望收集商业交易的历史数据用于分析。正如第 21.1节中提到的，早期的决策支持系统一般将历史数据保存在一个简单文件中，通过顺序扫描来访问数据。当数据量增长时，人们就希望能从不同的角度来直接访问数据文件。例如，检索出涉及某个产品的所有商业交易，在特定时期的所有商业交易，或者与某位顾客有关的所有商业交易。

要支持这种功能，可以采用“多重目录”数据库 ㊀。在上面的例子中，数据库中有一个很大的中心数据文件（包含了商业交易数据），还有三个独立的“目录”文件——产品、时期和顾客。目录文件中包含了指向数据文件中的记录的索引，但是（a）可以由用户来设置条目

㊀ 与现代数据库中的目录概念不同。

（“目录维护”）；（b）其中可以包含附加信息（例如顾客的地址），从而可以从数据文件中删除这些附加信息。注意目录文件一般比数据文件要小得多。

这种组织方式比原来的设计（只有一个数据文件）要有效得多，减少了空间和 I/O开销。我们特别发现中心数据文件中的产品、时期和顾客信息缩减了，现在只包含相应的标识符。

关系数据库中对这种方法的模拟，是将数据文件和目录文件变成表（相应文件的映像），目录文件中的指针变成了目录文件映像表中的主码，数据文件中的标识符则变成了数据文件映像表中的外码。这些主码和外码一般都作了索引。数据文件映像称为事实表，目录文件映像称为维表。根据实体/关系图中的形状，将整个设计称为多维模式或星型模式（事实表被多个维表环绕，并与它们相连）。注意：术语“维”的起源参见第 21.6节。

再次以供应商-零件数据库为例，现在希望列出特定时期内的发货。时期用时期标识符（TP#）来标识，引入表TP来将这些标识符与时期相对应。修改过的发货表 SP和时期表TP如图21-3所示 ㊀。在星型模式中，表 SP是事实表，表 TP是维表（供应商表 S和零件表P也是维表——参见图21-4）。注意：在第22章中会详细讨论如何处理时期数据。

SP

S#	P#	TP#	QTY
S1	P1	TP3	300
S1	P1	TP5	100
S1	P2	TP1	200
S1	P3	TP2	400
S1	P4	TP1	200
S1	P5	TP5	100
S1	P6	TP4	100
S2	P1	TP3	300
S2	P2	TP4	400
S3	P2	TP1	200
S3	P2	TP3	200
S4	P2	TP1	200
S4	P4	TP3	200
S4	P5	TP2	400
S4	P5	TP1	400

TP

TP#	FROM	TO
TP1	*ta*	*tb*
TP2	*tc*	*td*
TP3	*te*	*tf*
TP4	*tg*	*th*
TP5	*ti*	*tj*

图21-3 事实表（SP）和维表（TP）的实例

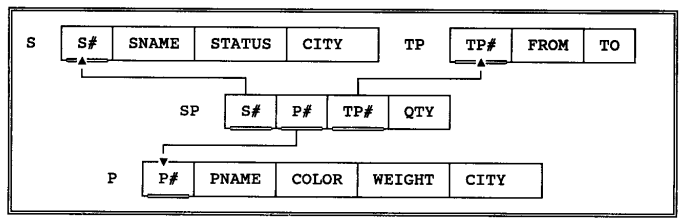

图21-4 供应商-零件数据库星型模式（包括时期）

查询星型模式数据库时一般会在维表中找到相应的外码组合，然后通过这些外码来访问事实表。假设在单个查询中要同时访问维表和事实表，最好的方法是使用星型连接。“星型连接”是一种特定的连接技术，它与常规的笛卡尔连接不同。在第 17章中提到查询优化器一般会尽量避免产生笛卡尔积 [17.54~17.55]；现在则是首先生成维表的笛卡尔积，然后再与事实表连接（基于索引的查找），这样做很有效。不过要求修改优化器来处理星型模式查询。

那么星型模式和关系设计之间有什么不同呢？实际上，如上所述的简单星型模式类似（甚至等同）于良好的关系设计。遗憾的是，在星型模式设计方法中经常会出现如下问题：

㊀ TP表中的FROM和TO列为时间戳类型。此处用符号简化表示。

1) 首先是特定性错误（基于直觉而非基于原则）。由于不遵循原则，当增加新数据类型或新的依赖时难以修改模式。实际上，很多星型模式都是对以前设计的修改，而以前的设计又大多是试验性的，包含错误的设计。
2) 星型模式是物理模式，而不是逻辑模式，在星型模式设计方法中不存在逻辑设计的概念。
3) 星型模式设计方法不一定会产生合理的物理设计（即它不一定会完全保留关系型逻辑设计中的所有信息），模式越复杂，这个缺点就越明显。
4) 因为缺少设计原则，设计人员常常会将多个不同类型的事实存放到同一个事实表中，因此事实表中的行和列就会产生语义分歧。而且有的列还可能只保存某些类型的事实数据，这样此列就必须允许空值存在。当引入越来越多的事实类型时，事实表就变得难以维护和理解，访问效率也逐渐下降。例如，我们可能希望在发货表中不仅能跟踪零件的发货信息，而且也能跟踪零件的购买信息，于是就要增加一个“标志”列来显示哪些行是关于购买零件，哪些行是关于零件的发货。正确的设计应该是将事实表分离成两个，每个表对应不同的类型。
5) 同样因为缺少设计原则，维表也会变得很不统一。当事实表中包含了不同聚集层次的数据时，很容易出现这种错误。例如，我们可能会（错误地）在发货表中增加一些行，用来显示每天、每月、每年甚至所有日期的零件总数，这样做将导致时期标识符（TP#）和表SP中的数量列（QTY）出现语义分歧。假设用YEAR、MONTH、DAY等列的组合来代替维表TP中的FROM列和TO列，那么这些YEAR、MONTH、DAY列就必须允许空值存在。而且还可能要增加一个标志列，以标明相应的时期类型。
6) 维表一般不是完全规范化的[⊖]。为了避免表连接的开销，设计人员常常会把截然不同的信息都保存到同一个维表中。最极端的情况是将所有可能访问到的信息都存放到同一个维表中，这种极端的、非关系型的无原则性几乎肯定会导致失控冗余的出现。

雪花模式是由一组星型模式组成，其中的维表是规范化的，名称的由来源于其实体/关系图的外观。星座模式和大风雪（或暴风雪）模式则是进一步的推广。

21.6 联机分析处理

术语OLAP（online analytical processing）出自1993年Arbor软件公司的白皮书中[21.10]，不过这个概念（以及“数据仓库”的概念）早就出现了。可以把OLAP定义为“关于数据的创建、管理、分析和报表生成的交互处理”——这些数据好像是存储于一个“多维数组”之中。不过我们首先用传统的SQL表来解释概念，然后再探讨其多维表达方法。

在分析处理中总会要求进行一些不同形式的聚集操作（按照不同的分组方法），常见的基本问题是可能的分组会变得非常庞大，而用户会用到大多数或全部的分组。现在的关系语言支持这种聚集操作，但是每个查询只能产生一个结果表（表中的行形式相同，表达相同的语义），那么为了得到n个不同的分组信息就要求分别进行n次不同的查询，产生n个不同的结果表。例如，在供应商-零件数据库中考虑以下查询：

1) 得到发货的总数。

⊖ 某本关于数据仓库的书中给出如下建议：“[反对]规范化……在多维数据库中对表进行规范化只是为了节省磁盘空间，却增加了时间开销……维表不应被规范化，否则会影响可用性”[21.21]。

2) 得到每个供应商的发货总数。

3) 得到每个零件的发货总数。

4) 得到每个供应商的每个零件的发货总数。注意：本例中如果使用供货商-零件-工程数据库可能会更真实，不过为了简化还是采用供应商-零件数据库。

假设只有两个零件P1和P2，发货表如下所示：

SP

S#	P#	QTY
S1	P1	300
S1	P2	200
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200

下面给出以上四个查询对应的SQL语句 ⊖ 和查询结果：

1)
```
SELECT SUM(QTY) AS TOTQTY
FROM   SP
GROUP  BY () ;
```

TOTQTY
1600

2)
```
SELECT S#,
       SUM(QTY) AS TOTQTY
FROM   SP
GROUP  BY (S#) ;
```

S#	TOTQTY
S1	500
S2	700
S3	200
S4	200

3)
```
SELECT P#,
       SUM(QTY) AS TOTQTY
FROM   SP
GROUP  BY (P#) ;
```

P#	TOTQTY
P1	600
P2	1000

4)
```
SELECT S#, P#,
       SUM(QTY) AS TOTQTY
FROM   SP
GROUP  BY (S#,P#) ;
```

S#	P#	TOTQTY
S1	P1	300
S1	P2	200
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200

这种方法的缺点非常明显：用户感觉分别阐述这些类似的查询是十分乏味的，而在相同的数据上一次又一次地执行这些查询显然需要过多的执行时间。于是人们希望找到一种方法能够（a）在一次查询中进行不同层次的聚集；（b）更有效地执行这些聚集操作。出于这种考虑，在GROUP BY子句中引入了GROUPING SETS、ROLLUP和CUBE选项。注意：这些选项已经被一些商业产品中支持，同时也被SQL3支持（参见附录B）。

GROUPING SET选项允许用户精确指定进行哪些特定分组操作。例如，下面的SQL语句实现了第2个和第3个查询：

```
SELECT    S#, P#, SUM (QTY)  AS  TOTQTY
FROM      SP
GROUP     BY GROUPING SETS((S#), (P#)) ;
```

GROUP BY子句会要求系统有效地执行两个查询，一个按S#分组，另一个按P#分组。注意：内层括号并不是必需的，因为每个“分组集合”只涉及单个列，使用括号只是为了增加可读性。

⊖ 也许应该称为“伪SQL语句”，因为在SQL/92中不允许将GROUP BY子句的操作数放在括号中，也不允许在GROUP BY子句中不出现操作数。

现在已经可以将多个分离的查询捆绑到一条语句之中，但是 SQL会把这些分离查询的结果都放到一个结果表中⊖ ！在本例中的结果表如下所示：

S#	P#	TOTQTY
S1	*null*	500
S2	*null*	700
S3	*null*	200
S4	*null*	200
null	P1	600
null	P2	1000

以上的输出结果可被视为一个表（至少是 SQL风格的表），却很难说它是一个关系。因为供应商行（P#列为null）和零件行（S#列为null）的语义不同，而TOTQTY的含义要根据它是出现在供应商行还是零件行来判定。这个“关系”的断言又该是什么呢？

我们还注意到，结果中的 null构成了另外一种“空缺信息”，它们既不表示“未知值”也不表示“无适用值”，很难弄清楚它们的含义。注意：SQL中至少提供了一种方法来将这些新的null值与其它null值区分开来，但是其技术细节过于冗长并强迫用户去逐行辨析。此处我们忽略这些细节，大家可以参考下面的例子：

```
SELECT    CASE GROUPING(S#)          — 参见附录A中的CASE
                  WHEN 1 THEN '?????'
                  ELSE S#
               AS S#,
               CASE GROUPING(P#)          — 参见附录A中的CASE
                  WHEN 1 THEN '!!!!!!'
                  ELSE P#
               AS P#,
               SUM (QTY)  AS TOTQTY
FROM      SP
GROUP     BY GROUPING SETS((S#), (P#)) ;
```

再次回到GROUP BY子句，另外两个GROUP BY选项ROLLUP和CUBE都可以视为由一些GROUPING SETS组合而成。首先来看ROLLUP，考虑下面的查询：

```
SELECT    S#, P#, SUM (QTY)  AS TOTQTY
FROM      SP
GROUP     BY ROLLUP(S#, P#) ;
```

上面的GROUP BY子句等同于：

```
GROUP     BY GROUPING SETS((S#, P#), (S#), ())
```

换句话说，上面的SQL语句实现了第4个、第2个和第1个查询。查询结果如下所示：

S#	P#	TOTQTY
S1	P1	300
S1	P2	200
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S1	*null*	500
S2	*null*	700
S3	*null*	200
S4	*null*	200
null	*null*	1600

⊖ 此表可视为结果的一个“外连接”——一种特殊的外连接。在第 18章中可以清晰地看到，“外连接”终究不是一种有效的关系操作。

术语上卷（ROLLUP）（在本例中）指的是对每个供应商进行数量上卷（即“沿着供应商维上卷”——参见下面的“多维数据库”小节）。一般说来，GROUP BY ROLLUP（*A*，*B*，...，*Z*），“沿着*A*维上卷”——指的是“按下列组合进行分组”：

(*A*, *B*, . . . , *Z*)
(*A*, *B*, . . .)
. . . . . .
(*A*, *B*)
(*A*)
()

注意，一般来说，存在许多分离的“沿着 *A*维上卷”（根据ROLLUP中用逗号分开的列来确定）。另外要注意GROUP BY ROLLUP（*A*，*B*）和GROUP BY ROLLUP（*B*，*A*）含义不同——即GROUP BY ROLLUP（*A*，*B*）中的*A*和*B*不具有对称性。

现在来看一下CUBE，考虑以下查询：

```
SELECT    S#, P#, SUM (QTY)  AS  TOTQTY
FROM      SP
GROUP     BY CUBE(S#, P#) ;
```

其中的GROUP BY子句在逻辑上等价于下面的子句：

```
GROUP BY GROUPING SETS((S#, P#), (S#), (P#), ())
```

换句话说，上面的SQL语句同时执行了前面的第4、3、2、1个查询。查询结果如下：

S#	P#	TOTQTY
S1	P1	300
S1	P2	200
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S1	*null*	500
S2	*null*	700
S3	*null*	200
S4	*null*	200
null	P1	600
null	P2	1000
null	*null*	1600

术语CUBE来源于OLAP中的事实：可以从多维数组或超立方体中的存储单元中找到数据的值。在本例中，（a）数据的值就是发货数量；（b）“立方体”只有两维，供应商维和零件维（所以，这个“立方体”其实是平面的！）；（c）两维的尺寸不同（因此这个“立方体”不是正方形而是长方形）。总之，GROUP BY CUBE（*A*，*B*，...，*Z*）等价于“按集合{*A*，*B*，...，*Z*}的所有可能的子集分组”。

在任一个GROUP BY子句中可以包括多个GROUPING SETS、ROLLUP和CUBE。

1. 交叉表格

OLAP产品中常常不是用SQL风格的表，而是用交叉表格（简称为“crosstab”）来显示结果。再回头看第4个查询（得到每个供应商的每个零件的发货总数），下面的交叉表格显示了查询结果。注意到供应商S3和S4在零件P1上的发货量为0，而在SQL中则为null（参见第18章）。实际上，使用第4个查询对应的SQL语句生成的结果表中，根本就没有与（S3，P1）、（S4，P1）对应的行！——因此使用交叉表格显然比原来的结果表更有价值。

	P1	P2
S1	300	200
S2	300	400
S3	0	200
S4	0	200

使用交叉表格可以更紧凑更易懂地表示第4个查询的结果。而且，它看起来也的确像一个关系表。不过，这个“表”中的列数是由实际数据决定的。在本例中，每个列对应一种零件（因此交叉表格的结构和每行的含义都是由实际数据决定）。交叉表格实际上并不是一个关系而是一份报表，格式为简单数组的报表（关系中拥有可从关系断言中推断出来的断言；但是，如果说交叉表格中也有断言的话，那么它无法从关系断言中推断出来，而是依赖于实际数据的值）。

在供应商-零件实例中，如上所示的交叉表格一般是二维的。各个维一般被视为自变量，表格中的单元则与相关变量对应。进一步的说明请参见下面的“多维数据库”小节。

下面的交叉表格显示了上面的CUBE查询结果：

	P1	P2	*total*
S1	300	200	500
S2	300	400	700
S3	0	200	200
S4	0	200	200
total	600	1000	1600

最右边的列中数值是各行的和（即各个供应商的总发货量），底部的行中数值是各列的和（即各个零件的总发货量），右下角的单元中数值是发货量总和。

2. 多维数据库

到目前为止，我们一直假设OLAP数据存储在传统的SQL数据库中（尽管曾多次提及“多维数据库”）。实际上，这是ROLAP（“关系型OLAP”）。但是，很多人认为MOLAP（“多维OLAP”）是更好的解决方案，下面的小节就对MOLAP进行探讨。

MOLAP涉及到多维数据库，从概念上讲，其中的数据存储在多维数组的单元中（注意：MOLAP的物理存储方式与其逻辑组织是十分相似的）。相应的DBMS平台称为多维DBMS。例如，可以把数据视为三维数组，分别对应于产品、顾客和时期。各单元中的值表示在相应时期内向相应顾客出售的相应产品的数量。前面的交叉表格也可以被视为这样的数组。

当已经深入了解数据和所有的关系时，就可以把涉及到的“关系变量”（不是程序语言中的变量）分为自变量或应变量。在前面的例子中，产品、顾客和时期是自变量，而发货量则是唯一的应变量。一般说来，自变量合起来决定了应变量的值（在关系数据库中，候选码就是这样的一些列，它们的值决定了其它列的值）。自变量构成了数组的各个维，并构成了数组的寻址模式 ⊖，应变量的值 —— 代表实际的数据 —— 则存储在数组单元中。注意：自变量（维变量）的值与应变量之间的区别类似于地点和内容的区别。

不过前面对多维数据库的描述是过分简单的，因为我们并不是很了解大多数数据。因此需要首先分析数据：从而进一步了解数据。由于对数据缺少了解，就无法判断哪些是自变量，

⊖ 此时数组单元用符号寻址，而不是采用传统的数组下标寻址。

哪些是应变量——一般根据经验（即假设）来选取自变量，然后测试生成的数组是否合适（参见第21.7节）。这种方法需要多次叠代、多次试验，会出现不少错误。因此系统中应该允许交换自变量和应变量，这种操作称为绕轴转动（pivoting）。其它操作包括数组转置和维的重新排序，同时允许增加维。

从以上的描述中可以得知，数组单元常常会是空的，也就是所谓稀疏数组。例如，假设在时期t中没有向顾客c出售产品p，那么单元（c, p, t）就是空的（或者为零）。多维DBMS支持对稀疏数组的高效存储（使用压缩技术）㊀。另外，空单元对应“遗失信息”，系统应该能处理这些单元——可惜经常是按照与SQL类似的方式来处理。空单元可能意味着信息是未知的，或者尚未获取数值，或者数值是不合适的，或者其它情况（参见第18章）。

自变量中常常存在分类层次，它决定了应变量可以聚集的方式。例如，存在如下的时间层次：从秒到分钟、小时、日、星期、月，一直到年。还可能存在如下的层次：从零件到套件、组件、集成电路板，一直到产品。相同的数据常常可以按照不同方式聚集（即相同的自变量可以属于不同的分类层次）。系统会支持分类层次上的“上钻”（drill up）和“下钻”（drill down）操作，上钻是从低层次到高层次的聚集，下钻则相反。此外还有很多对分类层次的操作（例如，对分类层次的再排列）。

注意：“上钻”和“上卷”之间存在着细微差别：“上卷”是创建所需分组和聚集的操作，“上钻”则是对这些聚集的访问操作。下面给出“下钻”的例子：已知总的发货量，要得到每个供应商的发货量。当然，必须存在更详细的数据才可以回答这个请求。

多维产品中还会提供大量的统计和数学函数，帮助用户阐述和验证自己的假设。同时提供可视化工具和报表生成工具。不过现在还没有多维查询语言的标准，目前正在进行相关的研究[21.27]。同时多维数据库也没有类似规范化理论的科学基础。

最后我们介绍一下综合了ROLAP和MOLAP方法的产品：HOLAP（“hybrid OLAP”）。目前很难说这三种方法哪个更好㊁。一般说来，MOLAP产品计算速度快但支持的数据容量较小，ROLAP产品支持更成熟的可缩放性、并发控制和管理机制。

21.7 数据挖掘

可以把数据挖掘描述为“探测型的数据分析”，它的目标是从数据中找到感兴趣的模式，用这些模式来决定商业策略或者发现不正常的情况（例如，信用卡突然被频繁使用可能意味着被人偷了）。数据挖掘工具在海量数据上应用了统计技术来查找这些模式。注意：数据量一般是很庞大的。数据挖掘数据库常常是特别大的，算法的可缩放性就十分重要。

图21-5所示的销售表中给出了某个零售商的售货交易信息㊂。商家希望能在这些数据上进行购物筐分析（“购物筐(market basket)”指的是单次交易中购买的所有产品），从而发现诸如购买鞋子的顾客常常也会同时购买袜子之类的信息。鞋子和袜子之间的相关性就是关联规

㊀ 如果在关系数据中模拟本例，则对于空值单元不会出现对应的(c, p, t)行，也不会出现“稀疏数组”或“稀疏表”的概念，因此也无需压缩。

㊁ 需要指出的是，人们常常说“表是平面的”（即二维的），而“现实数据是多维的”，因此关系模型不适用于OLAP。但这种说法混淆了表和关系的概念。在第5章中我们看到，表只是关系的映象！虽然表是二维的，关系却可能是多维的。具体地说，具有n个属性的元组就表示了n维空间中的一个点。

㊂ 注意：(a)主码是{TX#, PRODUCT}; (b)表满足依赖TX#→CUST#和TX#→TIMESTAMP，因此不属于BCNF；(c) 如果以{TX#}作为主码，则满足BCNF。

则的一个例子，可如下表示：

SALES

TX#	CUST#	TIMESTAMP	PRODUCT
TX1	C1	*d1*	Shoes
TX1	C1	*d1*	Socks
TX1	C1	*d1*	Tie
TX2	C2	*d2*	Shoes
TX2	C2	*d2*	Socks
TX2	C2	*d2*	Tie
TX2	C2	*d2*	Belt
TX2	C2	*d2*	Shirt
TX3	C3	*d2*	Shoes
TX3	C3	*d2*	Tie
TX4	C2	*d3*	Shoes
TX4	C2	*d3*	Socks
TX4	C2	*d3*	Belt

图21-5 销售表

```
FORALL tx(Shoes tx  Socks tx)
```

（其中“ Shoes ∈ *tx* ”是规则的前因，“ Socks ∈ *tx* ”是规则的结果， *tx*则代表任意一次交易）。

下面引入一些术语。全部交易的集合称为population，每个关联规则都有支持度和置信度。支持度指的是满足规则的交易占全部交易的百分比，置信度指的是同时满足前因和结果的交易占满足前因的交易的百分比（注意前因和结果中各自可以包含任意多项产品）。另外在本例中考虑如下规则：

```
FORALL  tx (Socks ∈ tx  Tie ∈ tx)
```

在图21-5中的示例数据中，population为4，支持度为50%，置信度为66.67%。

在给定数据上进行适当聚集可以得到更普遍的关联规则。例如，对顾客分组可以验证如下规则“ 如果某位顾客购买了鞋子，他（她）可能也会同时或在其它时候购买袜子 ”。

还可以定义其它类型的规则。例如，序列关联规则可用来标识随时间推移的购买模式（“ 如果某位顾客今天买了鞋子，他（她）就可能在五天内购买袜子 ”）。分类规则可用来辅助判断是否批准一项信用应用（“ 如果某位顾客的年收入超过 75 000美元，他（她）就可能比较值得信赖 ”），等等。类似于关联规则，序列关联和分类规则同样具有相应的支持度和置信度。

数据挖掘是个很大的课题，在此无法一一详述。最后简单介绍如何在供应商-零件数据库中进行数据挖掘。首先（在不涉及其它信息的情况下）可以使用神经归纳 (neural induction)来按经营项目（如刹车或发动机零件）对供应商分类，使用价值预测来预测出每个供应商最可能供应什么产品。然后使用人口统计的聚类把供应商和所在地区、运输地区联系起来。这时可以采用关联发现技术来找出每次运输中哪些零件会被同时装运，采用序列关联发现技术来找出当装运过发动机零件后一般会装运刹车零件，采用相似时间序列发现技术来找出某些零件的装运量随季节而变化。

21.8 小结

本章介绍了数据库技术在决策支持中的应用。其基本思想是从操作型数据中收集信息，然后将这些信息重新格式化，以帮助管理部门了解和修改企业行为。

首先指出了为什么决策支持系统要和操作型系统分离，主要因为决策支持数据库大多是只读的。决策支持数据库会变得很大，索引的开销繁重，同时涉及大量的受控冗余（特别是

在复制和聚集预计算中），码中一般有时间列，查询也很复杂。出于这些考虑，人们会在设计时特别强调性能问题；性能问题值得关注，但这并不是采用不好设计的理由。实际上，问题在于决策支持系统一般没有明确地区分逻辑设计和物理设计。

其次探讨了在为决策支持进行操作型数据准备时所涉及的步骤，包括提取、清洗、转换、合并、载入和刷新。还提及了操作型数据存储，可把它作为数据准备过程中的过渡阶段，也可以用来在现有数据上提供决策支持。

再次就是数据仓库和数据集市。数据集市可以视为一种特殊的数据仓库。我们解释了星型模式，其中包括一个大的中心事实表和多个小的维表。在简化时，星型模式和规范化的关系模式没有什么区别，而实际上星型模式与传统的设计理论存在许多不同，主要是出于性能上的考虑（还是同样的问题，星型模式在本质上更像物理模式而不是逻辑模式）。另外还提到了连接的实现策略（如星型连接）以及雪花模式。

接下来是OLAP。我们讨论了SQL GROUPING SETS、ROLLUP和CUBE（这些都是GROUP BY子句的选项，在单个SQL查询中实现多个不同的聚集操作）。SQL中把这些分离的聚集操作结果都放入单个表中（其中包含了很多空值），OLAP产品可以将这些表转化为交叉表格以便显示。在多维数据库中，从概念上讲，数据是存储在多维数组或超立方体中，数组的维代表自变量，单元中的值代表应变量。自变量一般涉及不同的分类层次，分类层次决定了应变量可以分组、聚集的方式。

最后谈到数据挖掘。由于一般没有充分理解决策支持，可以利用计算机的计算能力来帮助我们发现数据中的模式。我们简单介绍了几种规则 —— 关联规则、分类规则和序列关联规则，以及与其有关的支持度和置信度等概念。

练习

21.1 决策支持数据库和操作型数据库有什么主要区别？为什么一般要把决策支持应用和操作型应用分离开来？

21.2 决策支持中的操作型数据准备主要包括哪些步骤？

21.3 请区分受控冗余和失控冗余，并举例说明。为什么在决策支持环境中要强调受控冗余？如果是冗余失控，会出现什么后果？

21.4 请区分数据仓库和数据集市。

21.5 什么是星型模式？

21.6 星型模式一般都不是完全规范化的，如何处理这个问题？阐述其设计时采用的方法。

21.7 ROLAP和MOLAP之间有什么区别？

21.8 当数据包括四维，每一维中有三层分类层次时，有多少种方法进行汇总？

21.9 在供应商-零件-工程数据库（参见第4章的练习4.1）中，用SQL语言描述以下查询：

a. 找出每个供应商、零件和工程两两成对（即对于每对 S#-P#，P#-J#和J#-S#）所对应的发货次数和平均发货量。

b. 找出每个供应商、零件和工程任意组合和总共的最大、最小发货量。

c. 找出“沿供应商维”和“沿零件维”上卷的总发货量。警告：这里有个陷阱。

d. 找出每个供应商、零件和工程任意组合和总共的平均发货量。

在每个查询中给出SQL产生的查询结果（采用图4-5中的示例数据），并用交叉表格显

示。

21.10 在第21.6节的开始，我们给出了只包含6行的表SP。假设在表中另外增加如下的行（表示存在供应商S5，但他没有供应任何零件）：

S5	null	null

讨论它对第21.6节中所有查询的影响。

21.11 “多维模式”和“多维数据库”中的“维”是否是相同的概念？为什么？

21.12 考虑购物筐分析问题。简单书写关联规则（具有指定支持度和置信度）的发现算法。提示：某种（些）商品是“人们不感兴趣的”，指的是它（们）只在很少交易中出现。

参考文献和简介

21.1 Pieter Adriaans and Dolf Zantinge: *Data Mining*. Reading, Mass.: Addison-Wesley (1996).

尽管被描述为一本应用型的概述，该书实际上详细介绍了相关主题。

21.2 S.Alter: *Decision Support Systems: Current Practice and Continuing Challenges*. Reading, Mass.: Addison-Wesley (1980)

21.3 J. L. Bennett (ed.): *Building Decision Support Systems*. Reading, Mass.: Addison-Wesley (1981).

21.4 M. J. A. Berry and G. Linoff: *Data Mining Techniques for Marketing, QTY, and Customer Support*. New York, N. Y.: McGraw-Hill (1997).

很好地阐述了数据挖掘模型以及在商业中的可用性。

21.5 J. B. Boulden: *Computer-Assisted Planning Systems*. New York, N. Y.: McGraw-Hill (1975).

这本早期著作涉及的内容后来成为决策支持讨论的主题。正如标题所示，重点讨论传统管理规划。

21.6 R. H. bonczek, C. W. Holsapple, and A. Whinston: *Foundations of Decision Support Systems*. Orlando, Fla.: Academic Press (1981).

最早导入决策支持系统规范化方法的著作之一。重点讨论建模（经验和数学建模）和管理科学。

21.7 Charles J. Bontempo and Cynthia Maro Saracco: *Database Management: Principles and Products*. Upper Saddle River, N. J.: Prentice-Hall (1996).

21.8 P. Cabena, P. Hadjinian, R. Stadler, J. Verhees, and A. Zanasi: *Discovering Data Mining: From Concept to Implementation*. Upper Saddle River, N. J.: Prentice-Hall (1998).

21.9 C. L. Chang: “DEDUCE —— A Deductive Query Language for Relational Data Bases,” in C. H. Chen (ed.), *Pattern Recognition and Artificial Intelligence*. New York, N. Y.: Academic Press (1976)

21.10 E. F. Codd, S. B. Codd, and C.T.Salley: “Providing OLAP (Online Analytical Processing) to User-Analysts: An IT Mandate,” available from Arbor Software Corp. (1993).

本章中提到这篇论文首次给出了“ OLAP ”术语（尽管概念不同）。虽然论文一开始就说明“它不是要引入新的数据库技术，而是探讨更健壮的……分析工具”。但实际上却是在讨论一种新的数据库技术！—— 新的概念数据表达、新操作符、多用户支持（包括安全性和并发特征）、新存储结构以及新的优化特征；换句话说，讨论一种新的数据模型和一种新的数据库。

21.11 C. J. Data: “We Don't Need Composite Columns,” in C. J. Data, Hugh Darwen, and David McGoveran, *Relational Database Writings 1994-1997*. Reading, Mass.: Addison-Wesley (1998).

论文标题指出了引入复合列是没有意义的，应该支持合适的用户自定义类型。

21.12 Barry Devlin: *Data Warehouse from Architecture to Implementation* . Reading, Mass.: Addison-Wesley (1997).

21.13 B. A. Devlin and P. T. Murphy:“An Architecture for a Business and Information System,” *IBM Sys*. J. 27, No.1 (1998).

第一篇定义和使用“信息仓库”的论文。

21.14 Herb Edelstein: *Data Mining: Products and Markets*. Potomac, Md.: Two Crows Crop. (1997).

21.15 T. P. Gerrity, Jr.: “The Design of Man-Machine Decision Systems: An Application to Portfolio Management,” Sloan Management Review 12, No.2 (Winter 1971).

最早涉及决策支持系统的论文之一。描述了一个证券管理中的投资管理系统。

21.16 Jim Gray, Adam Bosworkth, Andrew Layman, and Hamid Pirahesh: “Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals,” Proc. 12th IEEE Int. Conf. on Data Engineering, New Orleans, La. (February 1996).

最早建议在SQL GROUP BY**子句中引入**CUBE**操作。**

21.17 W. H. Inmon: *Data Architecture: The Information Paradigm*. Wellesley, Mass.: QED Information Sciences (1998).

讨论了数据仓库概念的起源，并给出了数据仓库的特点。首次给出“数据仓库”术语。

21.18 W. H. Inmon: *Building the Data Warehouse*. New York, N.Y.: Wiley (1992).

描述数据仓库的第一本书。定义了开发数据仓库时的相关术语及关键问题。主要涉及概念和物理设计问题。

21.19 W. H. Inmon and R. D. Hackathorn: *Using the Data Warehouse*. New York, N.Y.: Wiley (1994).

面向数据仓库的用户和管理人员。探讨物理设计问题和操作型数据存储。

21.20 P. G. W. Keen and M.S.Scott Morton: *Decision Support Systems: An Organizational Perspective*. Reading, Mass.: Addison-Wesley (1978).

早期的决策支持经典著作。决策支持系统的分析、设计、实现、评价和开发。

21.21 Ralph Kimball: The Data Warehouse Toolkit. New York, N.Y.: John Wiley & Sons (1996).

在子标题“建造多维数据仓库的实用技术”中，重点讨论实际问题而非理论。认为系统在逻辑和物理上没有什么本质不同。

21.22 J. D. C. Little: “Models and Managers: The Concept of a Decision Calculus,” *Management Science* 16, No.8 (April 1970).

这篇论文介绍了Brandaid**系统，用于支持产品、推销、定价和广告决策。在决策制定管理设计中有四个要点：健壮性，容易控制，简单性和完备性。**

21.23 M. S. Scott Morton: “Management Decision Systems: Computer-Based Support for Decision Making,” Harvard University, Division of Research, Graduate School of Business Administration (1971).

介绍管理决策系统的经典文章，将决策支持应用到计算机系统中。建立了特定的“管理决策系统”，用于卫生产品计划。

21.24 K. Parsaye and M. Chignell: *Intelligent Database Tools and Applications*. New York, N.Y.: Wiley (1993).

第一篇介绍数据挖掘原理和技术的文章（文中使用“智能数据库”术语）。

21.25 A.Pirotte and P.Wodon: “A Comprehensive Formal Query Language for a Relational Data Base,” R. A. I. R. O. *Informatique/Computer Science 11*, No.2 (1977).

21.26 R.H.Sprague and E. D. Carlson: *Building Effective Decision Support Systems*. Englewood Cliffs, N.J.: Prentice-Hall (1982).

21.27 Erik Thomsen: *OLAP Solutions: Building Multi-Dimensional Information Systems*. New York, N.Y.: Wiley (1997).

最早讨论OLAP的综合性文章之一。侧重于多维分析概念和方法。

21.28 R. Uthurusamy: “From Data Mining to Knowledge Discovery: Current Challenges and Future Directions,” in U.m.Fayyad, G.Piatetsky-Shapiro, P.Smyth, and R.Uthurusamy (eds.): *Advances in Knowledge Discovery and Data Mining*. Cambridge, Mass.: AAAI Press/MIT Press (1996).

部分练习答案

21.8 **每一维中有8（=2^3）个可能的分组，所有总共有84=4096中汇总方法。 作为一个补充练习，可以考虑一下如何用SQL来进行这些汇总。**

21.9 **下面只给出相应SQL查询中的GROUP BY子句：**

a. `GROUP BY GROUPING SETS((S#, P#), (P#, J#), (J#, S#))`

b. `GROUP BY GROUPING SETS(J#, (J#, P#), ())`

c. **陷阱在于这个查询是含糊的——“沿供应商维上卷”可以有多种含义。不过下面给出其中一种解释对应的GROUP BY子句：**

```
GROUP BY ROLLUP(S#), ROLLUP (P#)
```

d. `GROUP BY CUBE(S#, P#)`

交叉表格不适合于显示超过二维的结果（维数越多，就显得越混乱）。对应于GROUP BY S#，P#，J#的交叉表格如下所示（部分）：

	P1				P2				...
	J1	J2	J3	...	J1	J2	J3	...	...
S1	200	0	0	...	0	0	0	...	...
S2	0	0	0	...	0	0	0	...	...
S3	0	0	0	...	0	0	0	...	...
S4	0	0	0	...	0	0	0	...	...
S5	0	200	0	...	0	0	0	...	...
..	...	...	...	...	...	...	...	...	...

总而言之：标题显得很笨拙，而且数组也是稀疏的。