提交维表

维表提供了事实表的上下文。虽然维表通常比事实表小得多，但它却是数据仓库的核心，因为它提供了查看数据的入口。我们经常说建立数据仓库其实就是建立维度。因此 ETL 团队在提交阶段的主要任务就是处理维表和事实表，将最有效的应用方式提交给最终用户。

流程检查

规划与设计：需求/现状 -> *架构* -> *实现* -> 测试/发布

数据流：抽取 -> 清洗 -> 规格化 -> *提交*

第 5 章和第 6 章是本书的关键章节；详细描述了如何将数据提交给最终用户或分析型应用。虽然可能在数据结构和提交流程中存在相当多变化，但最终 ETL 维表的结构相对稳定。

请注意我们坚持设计的高度一致性并不拘泥于一成不变的维度模型，关键在于要有一个可扩展的，可用的，可维护的体系架构。数据仓库的设计与标准的维度模型之间差异越大，就需要越多的客户化工作。大多数 IT 开发人员都能够胜任客户化工作，许多人也从开发中感受了智力挑战。但是在建立可扩展，可用，可维护的体系架构上，过多的客户化工作却是不可行的。

维度的基础框架

物理上所有的维度都应当是图 5.1 所示组件的最小子集。主键（Primary Key）是指包含了一个无意义的，唯一标识数字的字段。我们把这个无意义的数字称为代理（Surrogate）。数据仓库 ETL 过程应当常常要创建和插入这些代理键。换句话说，数据仓库拥有这些代理键值但并不把它赋给任何实体。

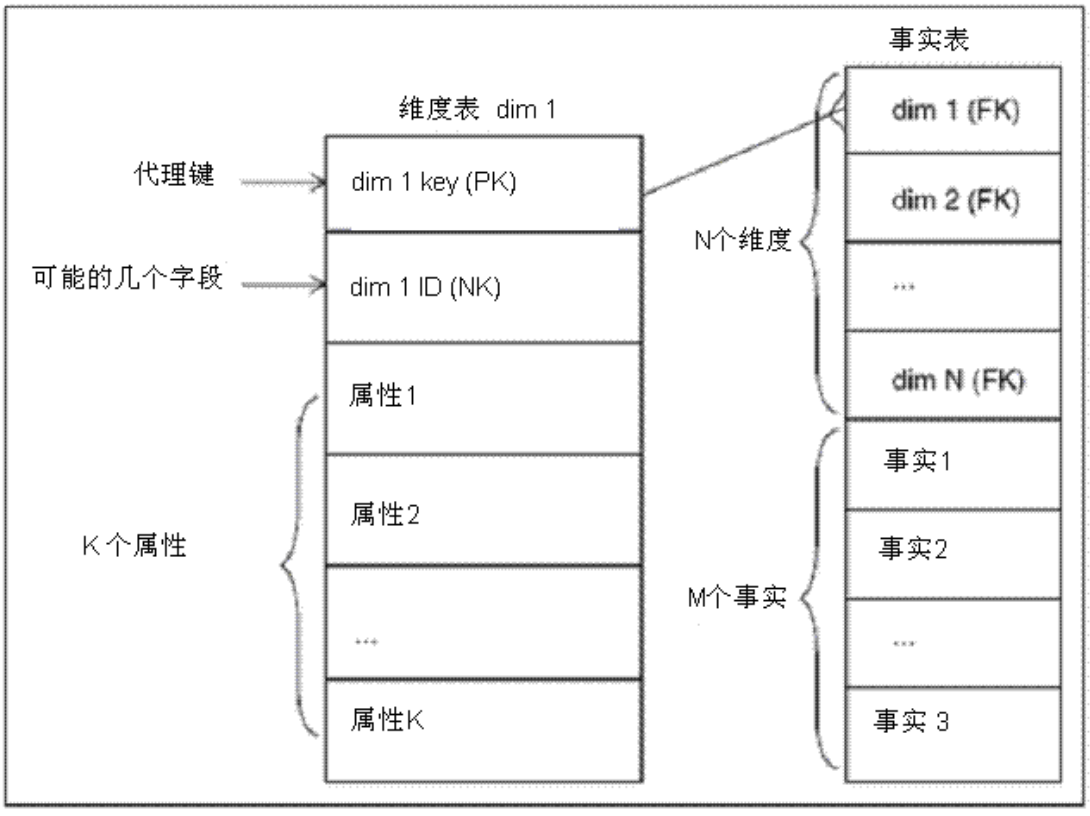

图 5.1　维表的基础结构

维度的主键用于连接事实表。由于所有的事实表都必须保持查找表的参照完整性，因此维表的主键所连接的字段就成为事实表的外键（Foreign Key）。在第二章的图 2.3 的保险案例中已有所阐述。在大多数关系型数据库中维表和事实表通过单一的字段进行连接可以获得最佳的性能。最后，当外键是数字型的时候事实表是最为紧凑的。

所有维表将其他的一个或多个字段组成维表的自然键（natural key）。如图 5.1 所示，ID 和其他的自然键字段组成了 NK，自然键并不是无意义的代理键，而是从源系统抽取而来的有意义的字段。比如，一个静态不变的员工维表中有常见的 EMP_ID 字段，它是人力资源部门赋予的员工号。EMP_ID 是员工维表的自然键。同时我们也会为其赋予代理键，这主要是为了满足以后人力资源系统的变化。

当维表是静态的并且不随时间变化时，那么代理键和自然键就是一一对应的关系。但在本章的稍后会看到有些维是缓慢变化的，那么我们就会为每个自然键产生多个代理键，以记录维度信息的历史变化。换句话说，在缓慢变化的维度中，代理键与自然键的关系为多对一。在我们的员工维表的例子中，每个变化的员工信息快照都有不同的唯一代理键与之对应，但对每个员工而言，都有相同的自然键（EMP_ID）。此逻辑会在本章的缓慢变化维一节中详细说明。

维度的组成除了主键和自然键外，还有描述属性（descriptive attributes）。 描述属性主要是文本型的，但也有数值型的。数据仓库架构中对维度会有大量的描述属性，比如员工，客户，产品等等。在某个维度中包含 100 个描述属性也不意外，只是希望这些属性都来自于干净的数据源。稍后会有详细说明。

数据仓库架构中对于周期性出现的指标量不应当出现在维表中，这些指标量通常出现在事实表中，而非描述属性。所有的描述属性应当是静态的，或者变化很慢，偶尔才发生变化的。指标事实和数值型属性之间的区别并不像听起来那么复杂。在 98%的案例中，区分是很明显的。剩下的 2%中，需要比较明显的参数来判断建模时到底是作为事实还是维度属性。例如，产品的单价经常是两种角色都会有，在最后的分析中，并不在乎是何种角色，在应用中可能会根据要求的不同而有所差异，但其信息内容却是相同的。但是如果单价是缓慢变化的，两种选择的差异就会重要得多。随着变化的频度加快，将会更倾向于指标量作为事实。

生成维度的代理键

通过关系型数据库创建代理键可能是目前最普遍的使用方法。但是，我们也看到这种趋势正在发生变化。过去，经常是通过数据库触发器创建和插入代理键。后来发现触发器在 ETL 过程中会带来严重的瓶颈，应当从进程中清除。而代理键作为数字型能够被接受，这些整数能够直接被 ETL 过程调用。数据库中的 ETL 过程比起数据库触发器，更大的提高了 ETL 的性能。但是，使用数据库产生代理键基本上不能保证产生的键值在数据仓库的各个环节保持同步 – 开发、测试和运营。 由于不同环节会在不同阶段加载，缺乏同步性会导致测试阶段开发者和用户之间的混淆。

为了效率，可以考虑使用第三方的 ETL 工具来维护代理键，来确保 ETL 过程中不同版本代理键的维护。

一种常见的解决手段是使用源系统的自然键加上时间戳。某些情况下可以采用智能代理键 – 比如精确的创建时间，但它并不能完全代替基于数字的代理键。比如在下列情况下智能代理键就不能使用：

- 定义。代理键就定义本身而言是无意义的。通过对代理键赋予一定的含义，扩展其职责范围，使其需要被维护。如果源系统的主键发生了改变或者进行了修订应该怎么办？ 相应的智能代理键需要进行更新，事实表中与之相关联的记录也需要更新。

- 性能。 源系统的时间戳降低了查询性能，作为数据仓库团队的一部分，我们不能控制源系统的内容，但必须处理好各种数据类型。这要求我们使用 **CHAR** 或 **VARCHAR** 数据类型来处理源系统的字母，数字等键值。另外，通过对这些键值添加时间戳，增加了 **16** 位字符甚至更多，使得这个字段非常笨拙。更糟的是，此键值不断的增加使得数据仓库事实表膨胀。存储这些数据和索引的空间会非常庞大，导致 **ETL** 和最终用户的查询性能急剧下降。并且，查询过程中 **VARCHAR** 类型的连接比 **INTEGER** 类型的连接要慢得多。

- 数据类型不匹配。有经验的数据仓库建模人员都使用 **NUMBER** 或者 **INTEGER** 数据类型来建立维度模型的代理键。这种数据类型禁止插入字符，也不允许使用时间戳。

- 源系统依赖度。智能代理键的使用依赖于源系统的维度属性在多长时间内发生变化。很多情况下，这些信息很难获得。没有可靠的字段审计维护方法，获得确切的变化时间戳几乎是不可能的。

- 异构数据源。自然键和时间戳的联合使用只支持同构数据源。在现实企业数据仓库环境中，往往多个不同的数据源有公共的维度。每个源系统有自己的应用场景，有各自的维度值。自然键加时间戳的方法缺乏对其他源系统的适应性。如果对每个自然键都加上各自相应的时间戳，那对维护是一个灾难。

使用智能键方法的可取之处在于 **ETL** 开发过程中建立第一个数据集市时的简单易用，尤其是直接在自然键后加上一个 **SYSDATE** 即可。但尽量不要这种简易方法，因为这种方法不能扩展到第二个数据集市中去。

维度的粒度

维度建模人员常常使用维度的粒度（Grain）这一概念。这意味着，对数据仓库架构和 ETL 团队而言，在业务上分析某个数据源，定义出维度的键值，确保此数据源相对应的粒度定义是个挑战。常见的例子就是商业客户维度。简单的讲此维度的粒度是客户，可以肯定的是给定了某个数据源文件，那么数据的粒度一定是由某些字段构成的。源文件中的数据异常和细微差别极有可能破坏最初对粒度的假设。当然，我们可以做一个简单的测试，看一看字段 A，B，C 是否能够组成源表的主键：

```
Select A, B, C, count(*)

From dimensiontablesource

Group by A, B, C

Having Count(*) > 1
```

如果此查询返回了记录，那么字段 A，B 和 C 就不能组成维表的主键（也就是粒度）。而且，此查询也能够帮助查出哪些行与假定不符。

抽取过程有可能会带来数据的冗余。比如，在非规范化的订单交易系统中，订单的配送号（Ship Via）并不是存放在一个专门的码表中，而是全部直接存放在订单交易表中，这样会有很多的重复。要创建维度模型，必须通过 SELECT DISTINCT 操作来建立 Ship Via 维表，这时候，源订单交易表中任何的数据异常都可能带来数据的冗余。

维度的基本加载计划

反倒 有些维度完全是由 ETL 团队建立，没有依赖外部的数据源，它们通常是将操作代码转化为文本的小型查找维度。这种情况下并没有真正的 ETL 过程。而这个小的维度表生成为一个关系表的最终形式。

真正重要的维度抽取是从一个或多个数据源开始的。我们已经描述过 ETL 数据流的四个步骤。这里有一些与维度相关的特别的想法。

对于比较大的、复杂的维度数据，例如客户，供应商，产品等往往是从多个数据源多次抽取而来。这需要注意识别跨数据源的相同维度实体，解决不同描述之间的冲突，以及在不同时间点的维度更新。这些课题会在本章中介绍。

数据清洗（Cleaning）包含了对数据的清洗，解决冲突（确认输送数据维度），应用业务规则以建立数据一致性的全部步骤。对于一些简单的维度，可能不会完全应用这些模块。但对于像员工，客户，产品等重要的大的维度，数据清洗模块就显得非常重要了，比如列的有效性校验，跨列的值检验，行的去重等等。

数据的规范化（Conforming）包含了整理数据仓库不同部分的相同或者相似维度字段的过程。比如，如果事实表记录的是计费交易和客户支持信息，那么这些事实表有可能都有自己的客户维度。在大型企业，原始的客户维度可能很不一样。更糟的是，计费客户维表和客户支持客户维表的数据结构都不一致，这时候就需要有一个规范化过程整理两个客户维表的字段，共享相同的域。在第 4 章中已经详细描述过规范化的步骤。规范化步骤会修改许多维度的描述属性,在此之后再存放规范化数据。

最后，维度提交需要管理缓慢变化维（SCD）问题，以正确的主键在适当的维度格式中（包含正确的主键，正确的自然键，以及描述属性）作为物理表将维度写入磁盘。创建和赋值代理键的过程就处于这个阶段。本章后续部分会描述不同情况下维度提交模块的细节。

扁平（Flat）维度和雪花（Snowflaked）维度

流程检查

规划与设计：需求/现状 -> 架构 -> *实现* -> 测试/发布

数据流：抽取 -> 清洗 -> 规格化 -> *提交*

维表应当是非规范化的扁平表。所有的层系和规范化结构在最后都应当扁平化。维度的实体必须有唯一的值与维表主键对应。大多数实体应当是中度或低度基数。比如，员工维度的性别字段会有三个基数（男，女或未知），地址中的州（美国）字段基数是 51。如果早期的维表是第三范式的，那么得到规范化维表比较容易，通过简单的查询就可以实现。如果所有的数据关系在数据清洗过程已经整理过，那么在维表的扁平化过程中可以保留这些关系。在维度模型中，数据清洗步骤独立于数据分发步骤，这样最终用户不必面对复杂的规范化结构。一些复杂的维度，比如商店，商品维度等，具有多个并行、内置的层系结构是很正常的。比如，商店维度可能既有地区层系，包含地址，城市，县，州等，又有商品销售区域层系，

包括地址，区域等。这两个层系共存于相同的商店维度。每个属性有唯一的值对应于维表主键。

如果维表规范化，并且层系的不同层次之间遵循多对 1 的关系，那么层系就创建了一个雪花型的结构。参见图 5.2。

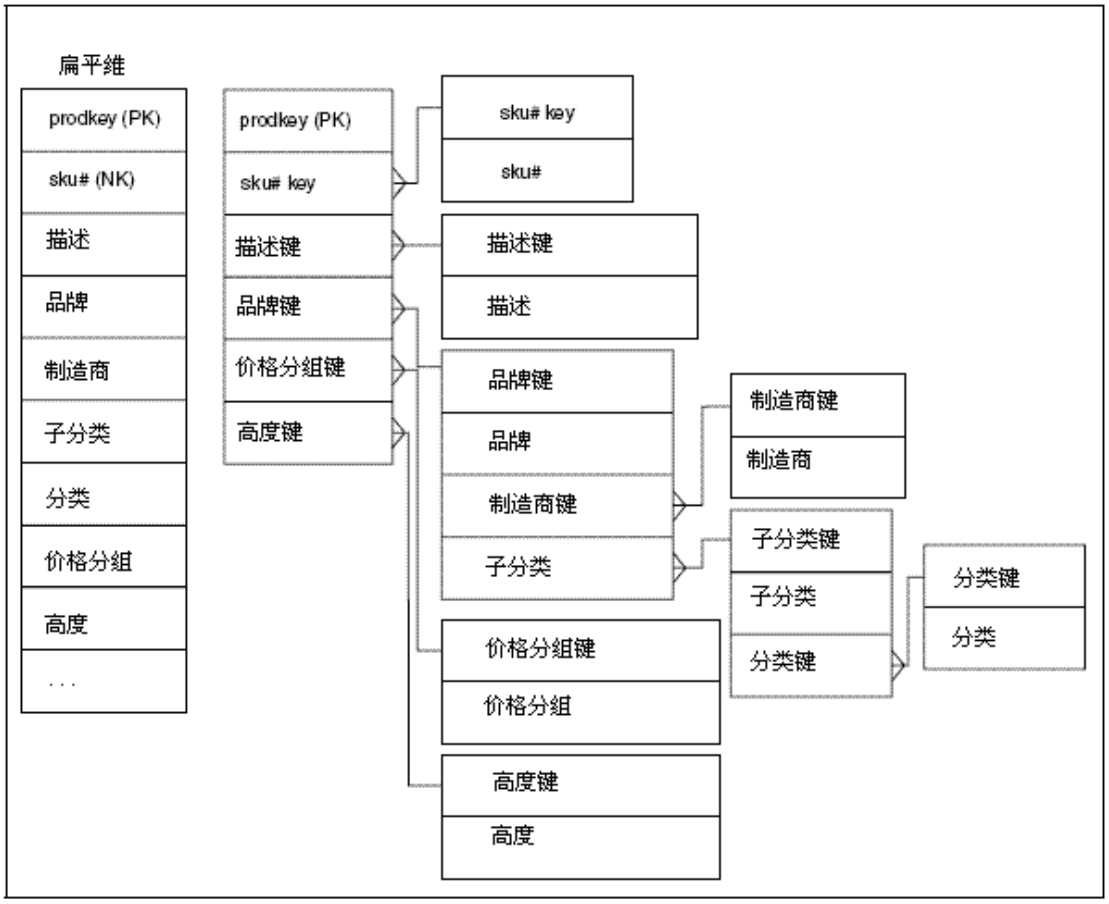

图 5.2　维表的扁平模型和雪花模型

图中维表两个版本的信息内容没有差异。差异在于雪花模型对最终用户的环境有一定的负面作用。这里有两个问题。首先，如果层系模型中严格的多对 1 关系发生了变化，规范化的表框架和表连接相应也必须改变，最终用户的环境也必须在相应层次上被记录，应用才能继续工作。扁平化维表没有这个问题。第二，复杂框架容易混淆最终用户，规范化框架在数据仓库的展示层需要包装。一般来讲，扁平化维表出现在用户界面直接带来的混淆较少。

但是有些情况下还是推荐使用雪花型的。有一些被描述为子维度的维度，这在本章的后序部分会有叙述。

如果实体对维度的主键有多个值，那么此属性就不能作为维度的一部分。比如，零售商店维度，现金登记员 ID 属性对每个商店都有多个值。如果维度的粒度是单个的商店，那么现金登记员 ID 就不能是该维度的属性。要包含现金登记员属性，维度的粒度就必须在现金登记员一级，而不是在商店上。但是由于现金登记员和商店是标准的多对 1 关系，新的现金登记员包含了商店的全部属性，商店的全部属性在现金登记员级别上具有单一的值。

每一次创建了新的维度记录，就必须赋给一个新的代理键。参见图 5.3。

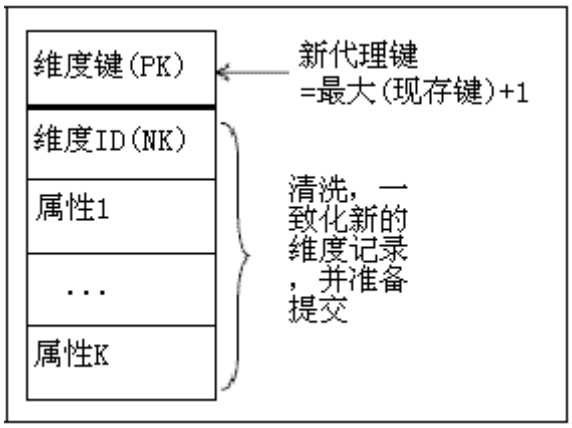

图 5.3　维表模型代理键赋值

这些无意义的整数是维度的主键。在集中式数据仓库环境中，维度的代理键可能是从单一的数据源生成而来。在这种情况下，主要的元数据包含了维表同时使用的最高的键值。但是，即使是在集中式的数据仓库中，如果有足够多的 ETL 作业在同时运行，也有可能引起对元数据元素的读写冲突。在分布式环境中可能没有这个问题。基于这些原因，我们推荐对每个维表单独建立代理键计数器。这样，不同的代理键是否有相同值无关紧要，重要的是数据仓库不会混淆不同的维度，由于其定义是无意义的，因此应用也不必分析代理键值的含义。

日期与时间维度

流程检查

规划与设计：需求/现状 -> 架构 -> *实现* -> 测试/发布

数据流：抽取 -> 清洗 -> 规格化 -> *提交*

实际上每个事实表都有一个或多个与时间相关的外键。度量的值发生在某个时间点，并且会随时间多次发生。

零售结构中的日期维度

图 5.4　日期维度所需属性

最常见有用的时间维度就是日历日期，粒度到天。此维度有许多属性，如图 5.4。只有少数属性（比如月，年）才会直接由 SQL 语句的日期表达式生成。节假日，工作日，财务期，月末标志以及其他的标志属性应当内嵌在日历日期维度中，并且通过在实际应用中利用日期维度实现全部日期标志。日历日期维度有一些很不寻常的属性，完全是在数据仓库项目一开始就指定的。它们通常没有常规的数据源。创建日历日期维度最好的办法就是花上一个下午手工建立一个日期表，十年的数据也不超过 4000 行。

每个日历日期维度都需要一个日期类型属性和一个完整的日期描述属性，如图 5.4。这两个字段组成了该表的自然键。日期类型属性几乎都有日期型值，但也肯定也包括一些至少不是日期型、或者数据有所损坏的数据情况。对于这些特殊数据的事实表外键必须指向日期表的非日期类型。日期维表中需要这些特殊记录，但需要根据不同的情况加以区分。对于非日期的情况，日期类型值是“不适用”或者“NA” 。完整的日期属性是一个时间戳，有些情况下空值是合法的。但是事实表的外键不可能是空值，因此从定义上违背了参考一致性。

理想情况下日历日期主键应当是无意义的代理键，但很多 ETL 团队仍然将可读的值作为键值，比如 20040718 意思是 2004 年 7 月 18 日。对于一些特殊情况的处理使用了一些技巧，比如，对于不适合日期表述的值使用 99999999 的值，在应用的时候不引用维表而是直接表述，因为它不是一个有效的时间值。

即使主代理键是无意义的整数，我们也建议对日期代理键按照顺序赋值，使日期维表的开始日期键值以 0 开始。这使得事实表基于日期维表能够按日期分区。换句话说，事实表的最早日期数据可能在某个物理位置上，而最新日期数据则在另一位置上。分区也使 DBA 便于为删除和重新引用的最新数据建立索引，从而加快加载进程。最后，对于不适合日期表述值的代理键数值可以是一个比较大的值，方便这些记录总在活动分区上。

尽管日期维度是最重要的时间维度，但我们仍然需要一个月维度，因为事实表的时间大多是基于月的。有时候，我们可能还需要建立周，季度，年等维度。月维度应当是一个物理独立的表，从日期维度中选择创建。比如，从日期表中提取每月的第一天和最后一天作为月维表的基础。也有可能基于日期维表创建一个视图来实现月维表，但一般来讲不推荐这种方式。这样的视图会比生成物理表导致更大的查询，同时，这种视图能应用于日期维表，但却不能应用于客户或产品这样的维度。比如，你不能基于产品维度建立一个品牌视图，因为你并不知道哪些产品永久的属于一个品牌。

在某些事实表中，时间的粒度可能低于日一级，到了分钟甚至是秒一级。我们不能建立包含秒的时间维表，这样的话一年的数据量就达到了 3100 万记录！但我们又想要在保留日期维表同时支持对分钟或是秒的精确查询。因此我们可以精确的计算一下事实表记录的时间间隔。基于这些原因，我们推荐图 5.5 的设计。精确时间的日作为外键关联到我们熟悉的日期维度表中，但在事实表中直接加入了一个时间戳作为查询所需额外的精度。这样设计事实表，而非维表上。这样一来，维表中就不必使用分秒了，因为跨事实表记录的时间间隔计算会很麻烦。在以前的工具箱书籍中，我们曾经建议过带有分秒的时间维度使用距午夜的偏移量来记录，但现在逐渐认识到这样会导致计算时间跨度很困难。

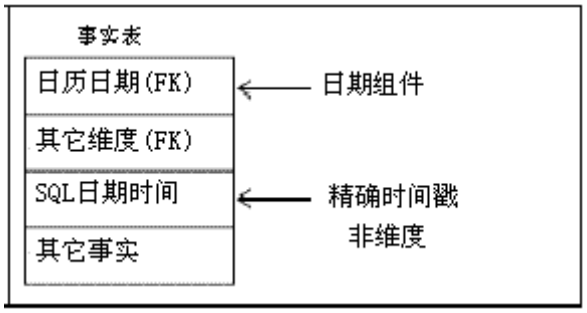

图 5.5　处理精确时间度量的事实表设计

同时，与日期维表不同，大多数情况下很少会有对分秒的描述。.

如果企业内有明确时间片定义的属性，比如轮班时间，广告时间等，那么可以将这些属性定义为对午夜时点的偏移量。如果粒度到分钟，那么这些每日时间的维度会造成 1440 条记录，如果粒度到秒会产生 86400 条记录。这时候也用得到以前描述的 SQL 日期时间戳的设计。.

大维度

流程检查

规划与设计：需求/现状 -> 架构 -> *实现* -> 测试/发布

数据流：抽取 -> 清洗 -> 规格化 -> *提交*

数据仓库中最有意思的维度是一些非常大的维度，比如客户，产品等等。一个大的企业客户维度往往有上百万记录，每条记录又有上百个字段。而大的个人客户维度则会超过千万条记录，这些个人客户维度有时也会有十多个字段，但大多数时候比较少见的维度也只有不多的几个属性。

真正大的维度总是由多个数据源衍生的。大的企业中，客户可以来自于几个账户管理系统。比如，银行中客户可以来自于抵押，信用卡，支票和储蓄等多个业务部门。如果银行想要创建一个所有部门的客户维表，那么就需要对这些各自独立的客户列表进行剔重，规范化并且合并。步骤如图 5.6 所示。

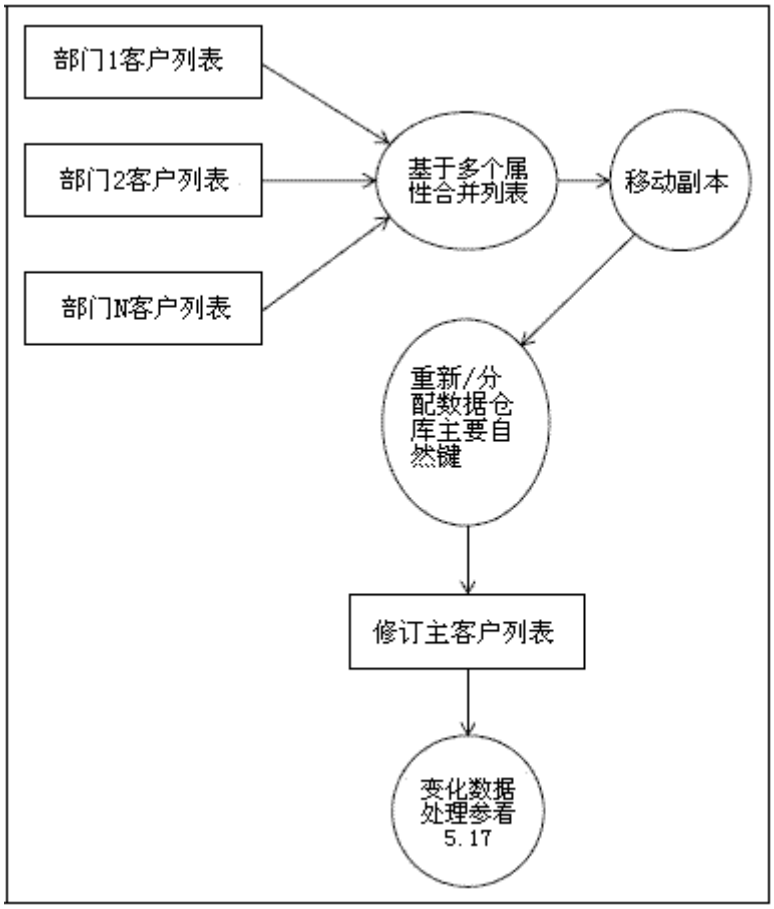

图 5.6　合并与剔重多个客户集

剔重步骤是数据清洗模块的一部分，每个客户必须与数据源完全一致。客户主要的自然键可以在这个时候创建，它有可能是一个在整个企业范围内唯一的客户 ID，不随时间的变化而变化。

规范化步骤是规范化模块的一部分，源系统中描述客户类似属性的全部属性需要转换成统一的属性值。比如，建立统一的客户地址字段。最后，在合并步骤，也就是提交模块的一部分，将不同数据源余下的属性统一起来，形成一个大的，广泛的维度记录。

本章的后面部分当我们讨论缓慢变化维的时候，我们会看到大维度是很容易发生变化的，如果要记录每一次变化就会形成新的维度记录。这在稍后再讨论。

小维度

流程检查
规划与设计：需求/现状 -> 架构 -> *实现* -> 测试/发布
数据流：抽取 -> 清洗 -> 规格化 -> *提交*

数据仓库中许多维度是一些小的查找表，只包含少量记录和一到两个字段。比如，许多交易级事实表有交易类型维度，提供每一笔交易的说明。创建这些表经常通过键入电子表格，，直接加载数据到最终物理维度表。最初的源表应当保留，因为经常会有添加新记录，像是新的交易类型引进新业务的情况发生。

尽管小维度可能出现在很多数据集市中，但它们不可能对所有事实表都相同，比如交易类型维度对每个生产系统都是不一样的。.

有些情况下，多个小维度可以合并成一个大一些的维表，提供解析操作类型值。这样可以减少关联到事实表外键的数量。有些数据源对事实表有多个操作代码，其中有很多的基数都较低。即使操作代码值之间没有明显的相关性，也可以创建一个冗余维度将各个代码合并到一起成为一个大的维度，从而简化设计。冗余维的 ETL 数据流如图 5.7 所示。

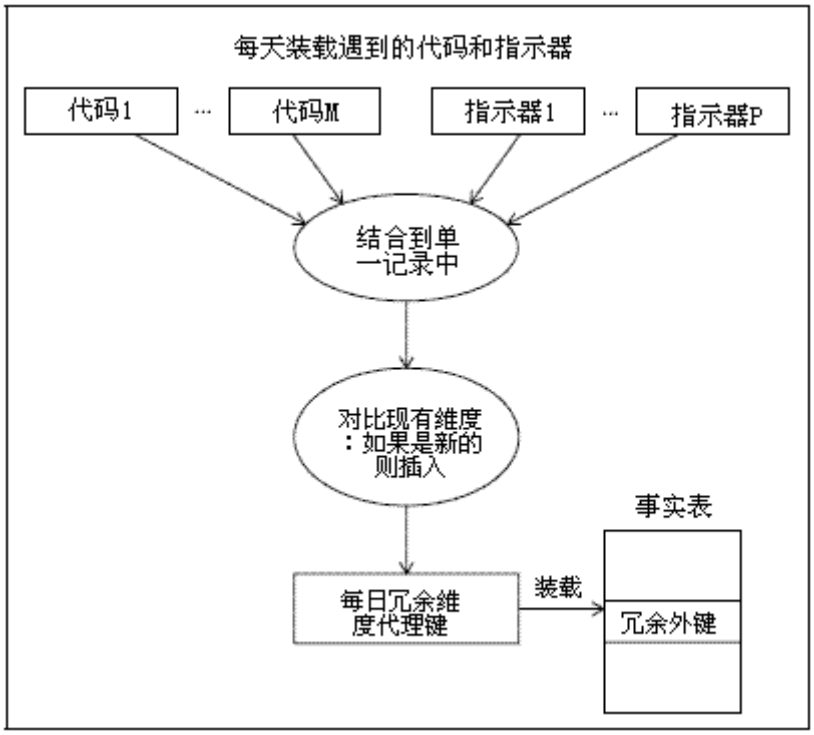

图 5.7　冗余维的 ETL 数据流

冗余维的记录可以是在整合数据时创建，而不是在此之前直接将各个独立代码表合并。这是因为增量建立的冗余维可能比笛卡儿积合并代码表要小得多。设计人员到底应该如何解决建立一个维表还是多个维表的问题，下一节会对此问题进行展开讨论。

一个维表还是多个

维表模型中，我们通常假设维表是相互独立的。

从严格的数据角度，经常并不是这样的。尽管你可以在多个商店销售多个商品，但商品

维度和商店维度并不总是相互独立的。有些商品只能在某些商店销售。一个好的统计人员能够描述出商品维度和商店维度的相关度。尽管这样却不能阻止我们建立相互独立的商店维度和商品维度。两个维度之间的相关度可以在销售事实表中得到准确的描述。

本例中如果将商品维度和商店维度合并将会是个灾难。想想看，如果商品维度是 100 万行记录，商店维度是 100 行记录，那么合并以后的维表记录数达到了 1 亿条！记录事实表不同维度的相关度就是一个很好的维度建模步骤例子：将维度相关度降级到事实表中。

不将商品和商店合并的另外一个理由是两个维度之间有不止一种关联关系。我们讨论过两个维度之间的销售关联度，但还可能有价格策略关联，库存关联，或者季节变化关联。总的来讲，跟踪这些复杂关系必须是让维度足够简单和独立，在一个或多个事实表中记录维度的关联性。

从这一点上，你可能已经相信所有的维度可以是独立的，但这是因为我们讨论的是一个稍微有些极端的例子。那么其它情况是否也是如此黑白分明吗？

首先，我们应该排除关联性完全相关重叠的两个维度。比如，如果商品和品牌是标准的多对 1 关系的话，在事实表中绝不能既有商品维度又有品牌维度。这种情况的商品和品牌属于一个层系，我们应当将它们合并到一个维度中。

有些情况下，两个维度并不属于同一个层系，但具有很强的相关性。从结果来看，如果其相关性足够强，并且合并后的结果足够小，这两个维度也应该合并在一起。否则，维度应当是独立的。应该从业务用户的角度考虑测试维度之间的相关性。如果两个维度之间的相关性得到最终用户认可并且固定不变，那么将其合并也是可行的。

请记住，维度合并的目的是为了提供有效的查询。按照我们的观点，当维度记录超过 10 万行记录，那么此维度就不算小维度了。也许随着时间的推进，这个强制的界限会得到缓解，但无论如何，一个 10 万行记录的维度都会是一种挑战！

角色维度

数据仓库架构会经常为相同的事实表多次指定一个维度。我们把它叫做角色维度。最常见的角色维度可能就是日期维度了。许多事实表，尤其是聚合快照事实表，都有多个日期外键。我们会在第六章讨论聚合快照事实表。参见图 5.8。另一个常见的角色维度例子就是员工维度，事实表的一笔交易中描述不同类型员工可能会有多个外键。参见图 5.9。.

在所有角色维度中，我们推荐建立一个统一的维表，然后基于此统一的维表建立不同的角色视图。参见图 5.10。例如，如果一个订单交易聚合快照事实表中我们有一个订单日期，配送日期，支付日期，偿还日期，那么我们首先建立一个通用的日期维度，然后创建四个对应的视图。如果每个视图的字段都以相同名字命名，那么应用开发者和最终用户就需要为其命名以便于在一个查询中进行区分。因此，我们建议创建不同的字段名以避免混淆。.

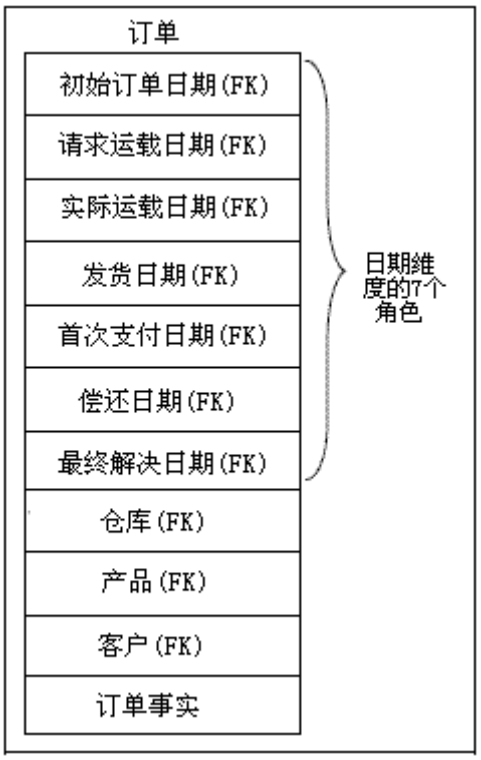

图 5.8　典型的快照事实表

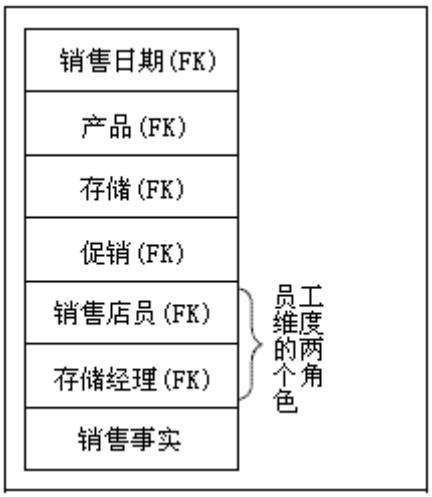

图 5.9　员工角色维

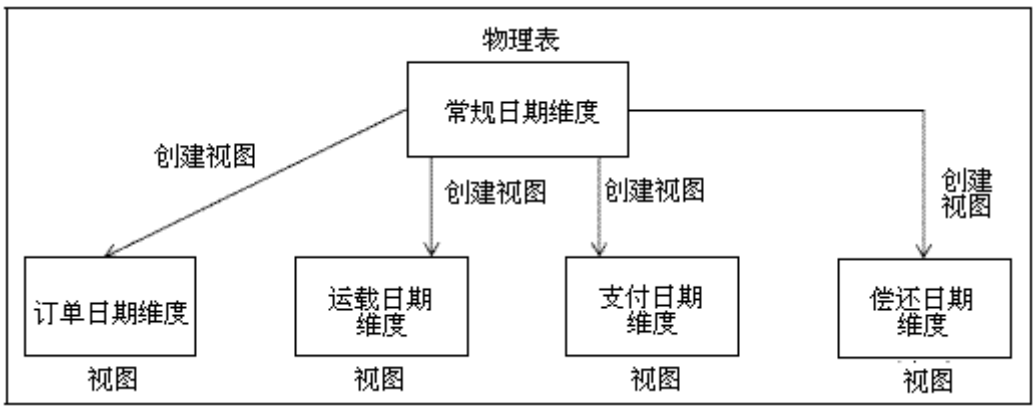

图 5.10　多个日期角色维

上述推荐的角色维度设计方法对 ETL 团队的工作没有影响。在这里讨论它的目的是帮助 ETL 团队无需建立多个物理的表，使用视图就可以达到相同的目的。

不要将角色维度技术作为创建超大维的借口。比如，在电信行业，几乎每一项记录都有一个地址，如果我们把每一个可能的地址都加入到一个统一的地址维表，那么该维表可能包含数百万记录。在这样大的维表上建立视图对性能无疑是个极大的灾难。这种情况下，为各个子集创建物理的表可能效果要好得多。

其它维度的子维度

流程检查
规划与设计：需求/现状 -> 架构 -> *实现* -> 测试/发布
数据流：抽取 -> 清洗 -> 规格化 -> *提交*

通常，我们将事实表中对维的引用看作是事实表的外键。但是，对维的引用有时也出现在其它维表中，那么正确的外键就应当是存储在父维表和事实表中。在有些书籍中，我们有时把对子维表的引用叫做外凸（Outtriger）。我们来讨论两个常见的例子。

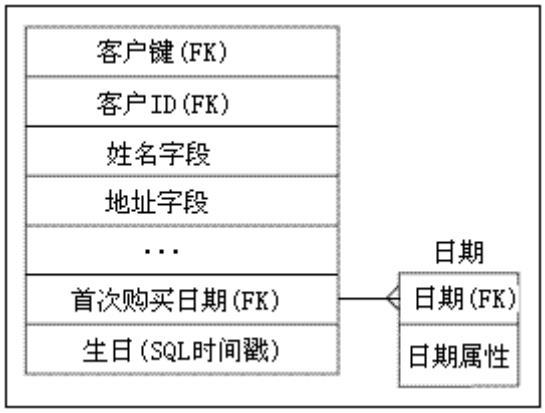

图 5.11　客户维的两个日期处理.

很多维度都有内嵌的日期。客户维度记录经常就有首次购买日期属性，这应当是以外键方式对日期维的引用，而不是 SQL 日期时间戳。参见图 5.11。这样，通过限定首次购买日期，应用程序就能够访问到日期的全部扩展属性。外键的引用实际上也是一种角色维，在日历日期维度的这种情况下,应当为这样的引用建立一个独立的视图。

由于日期维有一定的持续期，请注意并非所有的日期都会是对日期维的引用。像客户生日这样的属性可能会先于引用日期维度。如果出现了这种情况，客户生日属性必须是一个 SQL 日期时间戳，而非是外键。如图 5.11。

另一个常见的例子就是客户维与银行账号维。尽管一个账号有多个所有者，但通常也会指定一个主所有人。这个主所有人就应当是账号维对客户维的外键引用。如图 5.12。

在银行的例子中，我们没有处理多个客户关联一个账号的问题，只处理一个主所有人的情况。在本章的后面部分讨论多值维度和桥接表时会讨论这个问题。

ETL 维度分发模块必须将输入数据的所选字段转换为外键引用。

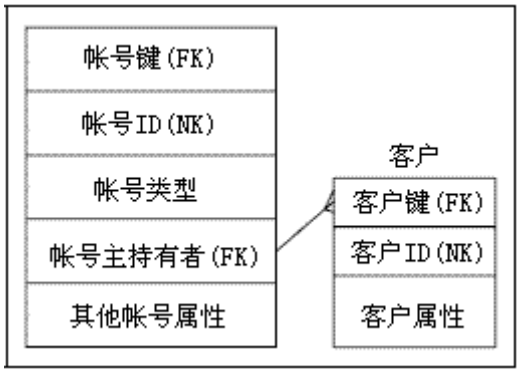

图 5.12　作为子维度的客户维

退化维

流程检查

规划与设计：需求/现状 -> 架构 -> *实现* -> 测试/发布

数据流：抽取 -> 清洗 -> 规格化 -> *提交*

一旦维表模型出现了父子关系，设计过程必然面临父维度的自然键问题。比如，如果事实表的粒度到订单一级，事实表的维度包含的全都是订单本身的信息，就如同是订单维度。在这里假定事实表每一条记录的维度值都是单值。如果我们关联客户，订单日期以及其它设计的维度，那么就只剩下了最初的订单号。我们将最初的订单号直接插入到事实表，就好像它是维度主键。如图 5.13。我们可以为订单号单建一个维度，但那样一来只是包含订单号而没有别的。基于此原因，我们把这种情况称之为退化维或者空维。

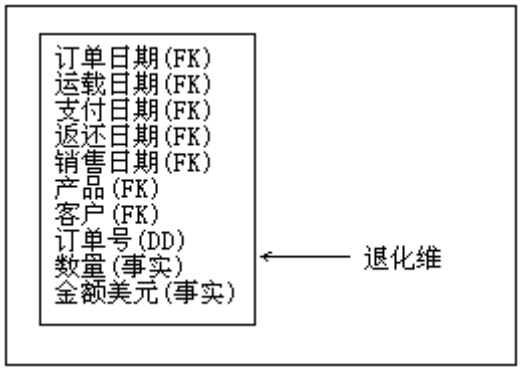

图 5.13　快照事实表的订单

这种情况总是出现在父子关系中，包括订单号，配送号，票据号，保单号等等。

有一个风险，一个综合性企业中各源系统产生的单据号可能被不同业务单位的 ERP 实例使用。针对这种情况，在基础订单号或销售单据号加上一个机构 ID，生成智能退化键值是一个不错的方法。

缓慢变化维

流程检查

规划与设计：需求/现状 -> 架构 -> *实现* -> 测试/发布
数据流：抽取 -> 清洗 -> 规格化 -> *提交*

当数据仓库被告知一条已经存在的维度纪录在某些方面发生了变化，会生成 3 种基本响应。我们称这 3 种响应为装载类型 1、2 和 3 的缓慢变化维。

类型 1 缓慢变化维（覆盖）

第一种类型 SCD 是对于已经存在的维度纪录一个或多个属性的复写。见图 5.14，当数据被校正而不需要保留历史纪录，或不需要执行以前的报表，可以选择第一种处理方式。

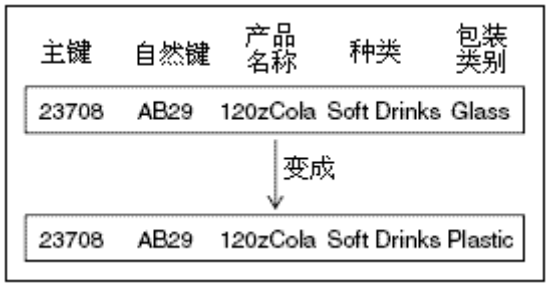

图 5.14　处理类型 1SCD

第一种方式的复写是潜在数据的一种更新，在 ETL 环境中，这种从早期不变的集结表的复写必须向前传递，以至于这些表被用于产生最终装载表，这种复写是受保护的。这个问题将在第八章阐述。

尽管向第一种缓慢变化维度插入新纪录需要依赖新的维度键值的生成。类型 1 的处理变化不会影响维度表键值或事实表键值，对数据的影响是这三种装载类型中最小的。如果聚合是直接建立在变化的属性上的，装载类型 1 对聚合事实表有一些影响。在第 6 章将详细描述这些影响。

一些 ETL 工具包含更新和插入功能。这些功能对于开发者很方便，但大大降低了性能。为了提高性能，在 ETL 处理中要将更新现有数据的过程和插入新数据的过程区分开，并分别向数据仓库回馈。在第一种类型环境中，可能会不知道引入的数据是由更新而来的还是新插入的。在插入占主导地位时，一些开发者会区分缓慢变化维或快速变化维，他们在缓慢变换维上使用 INSERT else UPDATE 逻辑，在快速变换维度上使用 UPDATE else INSERT 逻辑。我们希望这种做法还不是很普遍。

在大多数数据仓库执行过程中，大多数维度的大小是可以忽略的。如果装载小表不需要调用大量加载，类型 1 的变化可以通过标准 SQL DML 语句。基于从源系统中抽取的自然键，任何新纪录会分配新的代理键并添加到现有的维度上。现有纪录被更新。这种技术的性能与大量加载的装载方式比较起来，性能不占优势。如果表的大小合适的话，那么这种对性能的影响可以忽略不计。

一些 ETL 工具提供了专门的转化工具可以检测到记录是要被更新还是需要插入。然而，这种效用只有在表使用了候选纪录的主键的时候才会起作用。这种方法会更加损耗性能，应该避免。在使用 SQL 完成装载类型 1 维度时，为了减少性能上的损失，在 ETL 处理上应该把需要更新的现有纪录与需要插入的数据明确的区分开。

在 ETL 处理中第一类缓慢变化维有可能引起性能问题。如果通过 SQL 数据操作语言实现这种技术，大部分数据库管理系统会在日志中记录事件发生，损耗系统性能。

数据库日志由数据库管理系统在后台创建和维护。对于数据进入的转换处理用户无控制的情况下数据库日志显得尤为重要。在线交易处理情况下无法对用户的行为进行控制。比如，在一次更新的中途结束窗口活动，数据库管理系统也许会回滚，撤销，或更新失败。数据库日志具有记录这些的能力。

相反的，在数据仓库中，所有的数据装载都由 ETL 处理控制。如果处理失败，那么 ETL 应该有能力从错误中恢复和标记中断。这样的话，数据库日志就显得有些多余。

激活数据库日志会使大维度表装载效率下降。一些数据库管理系统可以允许在某些数据操作语言处理开始的时候关闭日志，而且当触发批量导入器的时候也要关闭数据库日志。

批量装载第一种类型维度变化

由于第一种类型复写数据，最简单的执行技术是使用 SQL 更新使所有的维度属性正确反映当前的值。遗憾的是，由于数据库日志的原因，更新是一种性能较差的处理，会使 ETL 装载处理负载变大。如果是大量的第一种类型维度转换，减小数据库管理系统负担的一个最好方法是使用批量装载器。在单独表中准备新的维度记录。从维度表中删除记录并重新批量加载。

类型 2 缓慢变化维（分区历史）

第二类缓慢变换维度是在维度实体中追踪变化，并正确关联到事实表的标准基本技术。基本原理很简单。当数据仓库被告知一条现有的维度纪录要更改，而不是重写的时候，数据仓库会在变化时发布一条新的维度纪录。新的维度纪录还会分配一个新的代理主键。在此之前，所有的有维度的事实表中，代理主键被用作外键。只要在发生改变的时刻代理键分配的得当，那么不会有任何一个已有的事实表中的键值需要改变或更新，已有的聚合事实表也不会需要重新计算。在本章的稍候的章节里会介绍处理最新变化的更为复杂的例子。

我们之所以说第二种缓慢变化维度完美的区分了历史表，是因为维度实体的每个细节版本都正确连接到事实表纪录。在图 5.15 中，用图示的方法使大家了解缓慢变化维度的概念。这是关于一个名叫 Jane Doe 的员工维度记录。Jane Doe 最开始是实习生，然后转为正式员工，最后成为管理人员。Jane Doe 的自然键是他的员工编码，这是在他工作期间一直保持不变的一个编码。事实上，自然键域是不能更改的唯一业务逻辑，而其他的员工属性纪录可以发生改变。Jane Doe 的职务升迁的时候，代理主键也随之发生改变，要随着事实表的变化而变化。因此，如果我们仅仅在员工编码（employee number）或者员工姓名（Jane Doe）上作限定的话，有可能会得到在事实表上的所有历史纪录。这是因为数据库从维度表里挑出了所有的代理主键，并在事实表里进行关联，得出所有数据。但是如果我们把限定设为 Jane Doe, manager，我们只会得到一个代理主键，只会看到事实表中有关管理者的那一部份记录。

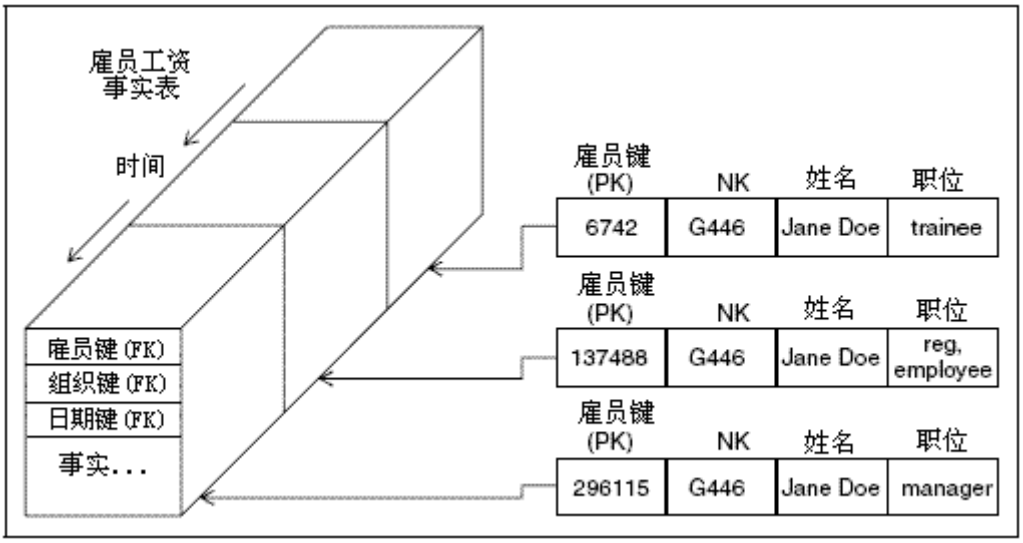

图 5.15　类型 2SCD 很好地分区了历史信息

如果一个维度的自然键可以变化，从数据仓库的观点上，它不是真正含义上的自然键。在信用卡处理上，信用卡号有可能会被设为自然键。我们都知道信用卡号码在业务处理上是可以发生变化的；因此，数据仓库要求使用更加符合基本原理的代理键。在这个例子里，一直使用最初的客户卡号码作为自然键是可行的，虽然这个客户号码后来有可能发生变化。在这样的情况下，客户目前的卡号将会是一个单独的字段，而不会被设计成为一个键值。

第二种缓慢变化维度需要在 ETL 环境中有一个良好的变化数据捕获系统。一旦源数据发生变化就要探测到，这样数据仓库就会生成新的维度纪录。我们在第 3 章讨论过许多有关捕获变化数据话题。假设在最差的情况下，源系统并不告知数据仓库其维度的变化，也不在数据更新的时候加上时间戳。这样，自从源系统最近一次下载维度以来，数据仓库不得不下载完整的维度，逐个域逐条记录察看维度发生的变化。这些需要在维度来源预先抽取（主要维度交叉涉及文件），明确装载到 ETL 系统中。见图 5.16。

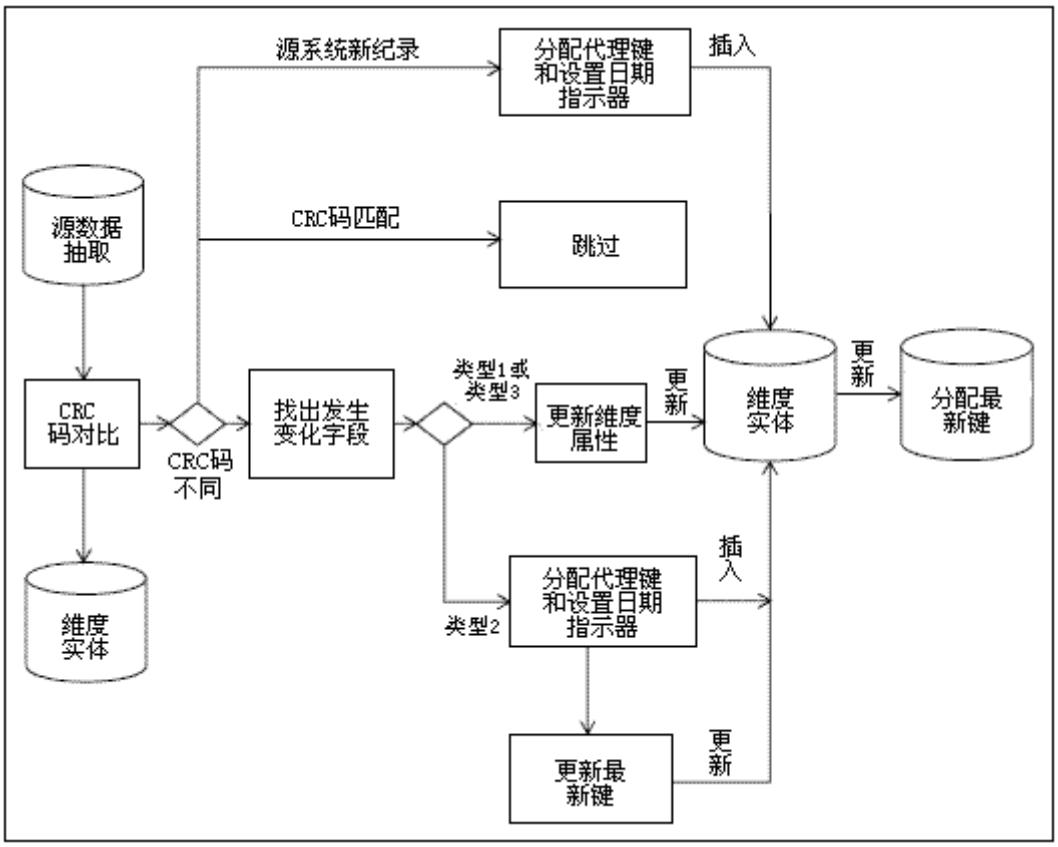

图 5.16　维表代理键管理

对于只有十几个字段，几千条纪录的小维度来说，比如说一个简单产品列表，在图 5.16 中所展示的那种变化的监测可以通过强制执行完成。今天所下载所有字段的所有纪录与昨天的所有字段的所有纪录相比较。增加的，变化的，删除的都需要被检测到。但是对于大维度，比如说一千万条加入医疗保险的患者名单，每名患者还有一百个描述字段，那么刚才所提到的那种对比所有纪录的方法效率就显得十分低下。如果是大维度的情况下，可以使用一种特殊编码作为循环冗余码校验。在这种情况下，使用一种被称为 CRC 的特殊编码计算并且关联到昨天的每条记录。CRC 的数据类型是长整形，大约 20 位，对于每条纪录的信息内容非常敏感。如果记录中有一个字母发生了变化，那么这条纪录的 CRC 码会变得与以前完全不同。这样我们捕获变化数据就会十分方便。我们只需把整条记录看作一个单独字符串，计算每条新建纪录的 CRC 码，然后对比 CRC 码。如果 CRC 码相同，我们立即转向下条纪录。如果不同，则停止对比纪录的每个字段，找出发生的变化。通过第 10 种因数使用 CRC 码技术可以提高变化数据捕获的效率。在本书写作之时，所有主要 ETL 产品都可以应用 CRC 码计算模型。而且，执行 CRC 对比的结果码可以很容易在课本上和互联网上找到。

一旦我们很明确的判定了更改的维度纪录，就可以知道使用哪种缓慢变化维度方法更为合适。通常，ETL 系统会为维度的每一列保留一个策略，这个策略决定了当变化发生时，应触发哪种变化类型。见图 5.16

为了识别在源系统中删除的纪录，可以读取源系统转换日志（如果有日志的话），或者通过我们描述的 CRC 码在 ETL 系统的匹配文件中通过自然键描述先前不能找到的纪录。但是无论哪种情况，执行删除操作一定要有明确的业务规则。在很多情况下， 被删除实体（比如一个客户）会不断出现在数据仓库中，这是因为删除实体在过去是正确有

效的。如果业务逻辑的最终状态是在数据仓库中不会出现已经被删除的实体，可以从对比环节中脱离出来，尽管它还继续存在于历史维度表和事实表。

除了转换日志文件，对删除数据的校验还需要大量的程序工作，所以只有需要的时候才使用。利用负数设定操作来对比数据仓库中自然键和源数据自然键，已经被证明是一种有效的方法。大多数数据库管理系统都支持关联设定操作和负数设定操作对数据进行对比。这些方法对评估源和目的数据之间的变化是非常有效的。但是如果要使用这些设定操作，两张表必须同时存在于数据仓库中，或已经创建了数据库之间的关联。一些 ETL 工具还支持异构系统间的设定操作。如果这对你的环境是至关重要的一点，那在选择工具的时候要把这点考虑进去。

在图 5.16 中，当为变化维度实体创建新的代理键，注意到我们更新了一个只有两列的查找表，也就是维度的最新键查找表。当装载事实数据时，这种小表非常有用。直到阅读第六章代理键管道（pipeline）部分的时候，我们都要保持这种观念。

我们同样可以通过在维度表中直接保存所有相关自然键来得到查找表带来的好处。后一种方法在判定装载的是自然键还是维度纪录的时候可能更为常用。这种方法在维度中正确关联可用自然键到用户。对比查找表这种方法的好处是代理键只在一个地方存在，排除了维度和映射表之间不同步的风险。在 ETL 过程中，程序从维度合适的列中挑选匹配引入的自然键值的自然键。在本章的稍候部分，我们会讨论如果建立一个匹配，程序如何适用于任何一个缓慢变化维度策略；如果没有发现键值，它能够产生一个代理键用于在下一阶段讨论的任何方法中并插入一条新记录。

很多数据仓库设计者都喜欢直接查找维度这种方法，这是因为它可以让用户明了数据沿袭。在维度中直接持有自然键，用户会很清楚的知道维度中的数据的来源，在源系统中就能够识别。此外，通过维护一个单独的映射表，维度的自然键减轻了 ETL 和数据库管理员的压力。最后，在有大量滞后到达数据片段的环境中无法应用到查找表的好处。

在这一部分，我们做一些变化数据捕获的假设，推断数据仓库维度记录是否发生改变以及原因。很明显，如果源系统移交了改变的纪录（这样可以避免我们之前提过的复杂对比程序），性能效率会非常卓越。如果变换纪录还有一个原因代码的话，那就更加完美了。这个代码我们在三种缓慢变化维度响应中提及过。

我们的方法允许我们对源系统中维度的变化做出响应，甚至这种变化没有做标记。如果一个数据库在初次装载时，就遇到了不配合的源系统，会遇到更困难的情况。在这种情况下，如果维度在源系统中被复写，那么就很难重新构建历史变化，除非原始处理日志文件还存在。

第二种缓慢变化维的精确时间戳

以前的那些讨论都是关于一个存在纪录变化被探测到之后，ETL 系统生成新的维度纪录。新维度纪录与事实表正确自动关联，这是因为在变化发生之后新的代理键被迅速地用于所有事实表的装载。没有时间戳来配合相应的工作。

话虽如此，在许多情况下，最好选择维度表的手段是提供关于类型 2 改变的可选择的有用的信息。

我们推荐在维度表中增加下面 5 个字段来处理类型 2 逻辑。

- 日历日期外键（变化日期）
- 行有效时间日期（变化的精确时间日期）
- 行结束时间日期（下一个变化的时间日期）
- 变化理由（正文字段）
- 当前标志（当前/期满）

这五个字段可以让维度支持强大的查询，甚至在查询里不需要事实表。日历日期外键允许终端用户使用业务日期（季节，假期，发工资日，和财务周期）来查询在某个业务周期内共有多少变化。举个例子，如果一张人力资源维度表，就可以查询在过去一个财务周期内共有多少人获得升迁。

两个 sql 时间戳定义了精确间隔，在时间间隔里当前维度纪录正确并完整地描述了实体。当发生转变时，行有效时间日期被设定为当前的时间日期，行结束时间日期被设定为未来的一个任意时期。当维度实体再次发生变化时，必须要重新访问原来的纪录，行结束日期被设定为一个适当的值。两个时间戳定义适当的间隔，所以查询可以针对某一个随机的特定时间日期，使用 SQL 语句中的 BETWEEN 逻辑可以立即提交数据。行结束时间日期要被设定为一个实际的值，这样即使 BETWEEN 语句的第二个日期为空值的时候，也不会返回错误信息。

普遍使用后台脚本处理数据库中没有各自元数据字段的修改数据，例如最后修改日期。利用维度行的有效日期时间的这些字段，将会导致数据仓库中的结果不一致。不要仰赖交易系统的元数据字段。经常使用系统日期，或者在类型 2 缓慢变化维中的源自行有效日期时间的日期。

变化字段的原因大概需要来自于最初的数据引用处理，增加变化的维度纪录。例如，在一个人力资源维度中，一次提升可以作为雇员维度里的一个单独的新维度纪录或一个时间戳。那么这次变化域的原因就是提升。事实上并不像说起来的那样简单，如果一个员工获得提升，很多不同的员工属性（职务等级，休假福利，头衔，机构等等）同时发生变化，人力资源系统在此时会提交变化的纪录；数据仓库就要把这些变化综合成为一条单独的新维度纪录并标示为“升迁”。像这样的综合潜在处理记录为一种聚合的超级处理，对一些源系统是非常有效的。我们看到一些与此相关的简单更新，比如说十几个关于升迁的微处理。数据仓库不应该在终端用户表上执行这么多的微处理，因为每一个单独的微处理并不一定有实际的业务意义。我们在图 5.17 中会描述这些处理。当前标志可以简单方便的重新得到维度中的所有当前纪录。但是对于两个时间戳来说它是多余的，因此可以在设计中忽略。当维度实体发生替代变化的时候，要把标志设为终止（EXPIRED）。

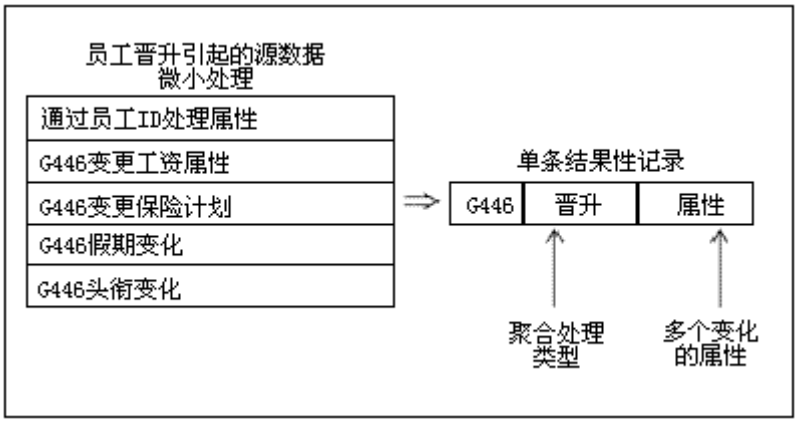

图 5.17　组合源系统的微处理

类型 3 缓慢变化维（交替实体）

当维度记录发生变化但属性原来的值仍然保持有效性的时候，要使用第三种缓慢变化维度。在销售地域分配经常会发生两种常见的业务情况，它的发生是两种普遍的业务状态，一是对销售任务分配的改变，这样原有的销售任务必须作为第二个可用选择继续下去，二是对产品种类指派的改变，那么原有的产品种类指派也必须作为第二个可用选择继续下去。数据仓库构架应该可以识别那些需要第三种类型的域。

在第三种缓慢变化维度中，当变化发生的时候，生成新的一列（如果事先并不存在的话）而不是新的一行。在新的主值复写之前，原来的值就被放置在这一列中。举个产品种类的例子，我们假定主域的名称是种类。执行第三种缓慢变化维度，我们改变维度表增加旧种类的字段，当变化发生时，把原来的种类值写入这个字段。然后向第一种缓慢变化维度那样复写产品域。见图 5.18，不需要在任何维度表中或任何事实表中更新键值。像第一种变换维度一样，如果聚合表需要直接建立在的第三种缓慢变换维度建立的字段上，那么要重新校验聚合表。我们会在第六章有所描述。

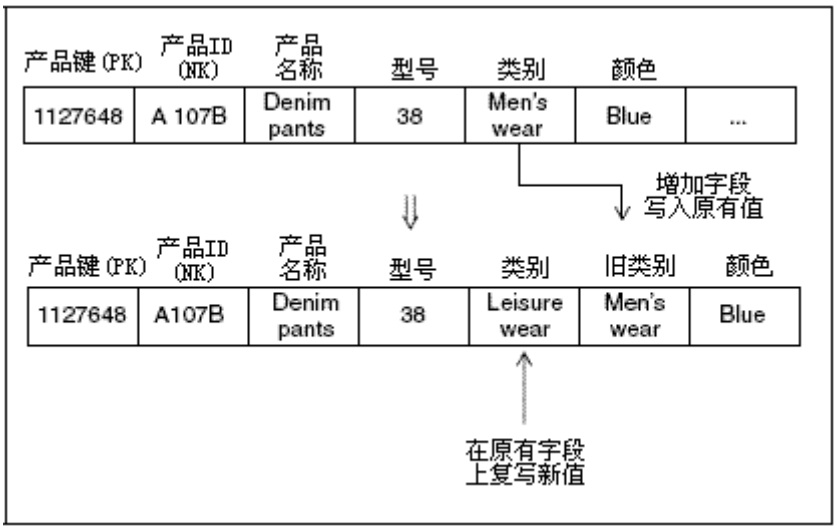

图 5.18　用于产品-种类描述的类型 3SCD 实现

在常规的数据流传递途径上，第三种方式不经常出现，相反,他们经常是一种 ETL 团队成员间口头上执行决定。产品类别经理说，“请把商标 X 从 Mens 运动装上拿下，放到皮革货品上，但是能够让我可以在原来的产品上随意追踪商标 X 的踪迹。”第三种方式的管理以手工的方式开始，如果变更的属性（在本例中为商标）没有交替字段的话，甚至可以包括结构计划的更改。

当一个包含第三种变化的新纪录被添加到维度中，一定会调用一条业务规则来判定如何来移入旧值。根据业务规则，当前值可能被复写到这个字段，或者它可能置为空。

我们经常把第三种缓慢变化维描述成支持交替的实体。在产品类别例子中，终端用户可以在两个版本的产品类别映射中选择。

第三种缓慢变化维方法通过创建基于原属性的交替字段任意编号的方式扩展到很多交替事实上。有时，当终端用户已经有一个各种口头事实的明确设想，这样的设计非常有效。

也许产品类别分布很有规律，但是用户希望得到各种时间跨度、各种产品类别的灵活查询。这种稍显笨拙的设计方式有一个好处，任何查询工具的用户接口不需要做程序，而且 SQL 不需要外联接或其他不寻常的逻辑。这个优势关键缺陷在于设计采用了位置依赖属性(交替字段)。

混合缓慢变化维

.响应属于第 3 种 SCD 类型的维度属性变化的结果是基于字段到字段的。在一个维度中通常可以包括第一种类型和第二种。当第一种类型字段发生变化，字段中的值被复写。第二种类型变化，就生成一个新纪录。在这个例子中，类型 1 变化需要拷贝所有持相同自然键的记录。如果它经常变化，（也许是一直更改最初错误的值）通过类型 2 的变化，必须在员工层面产生的所有副本中复写类型属性。

也有可能会在一条维度记录中混合三种不同的变化类型。见图 5.19。在这个例子中，销售团队的区域分配字段是第二种类型。当区域分配发生变化，生成新纪录，设定有效数据和开始数据。对每年的区域设定都是采用的第三种类型方法，执行多个交替实体。当前的分配字段是第一种类型，当前字段重新分配的时候，复写了销售团队纪录的所有副本。

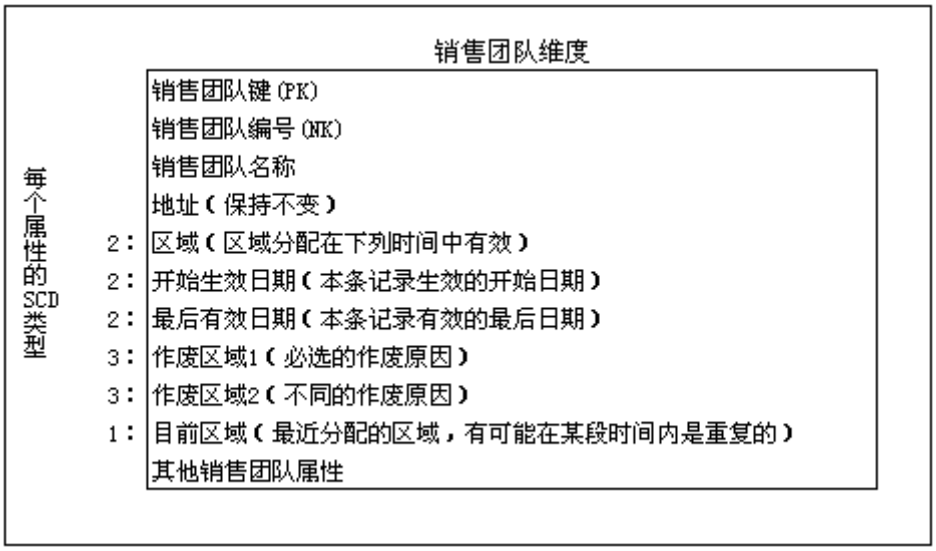

图 5.19　混合 SCD 显示了三种类型

滞后到达的维度记录和更正劣质数据

流程检查

规划与设计：需求/现状 -> 架构 -> *实现* -> 测试/发布

数据流：抽取 -> 清洗 -> 规格化 -> *提交*

滞后到达的数据需要通过不同的方法抽取或与常规数据不同的约束。很明显，劣质数据在数据清洗阶段就要筛选出来。

滞后到达的数据在数据仓库中呈现出一种复杂的状态。假设我们有一种产品名为 Zippy Cola，在其维度记录中，有一个公式表达字段值的格式为 A。由于使用第二种类型缓慢变化维度，Zippy Cola 会有很多条记录，很多属性比如包装类型和子分类等在过去的一两年中发

生了变化。

现在我们知道在一年前 Zippy Cola 的纪录格式变成了 B，而且以后会一直使用 B 格式。我们应该在一年前就处理了变化，但是失败了。在数据仓库中明确这个信息需要以下步骤：

1、插入带有代理键，格式为 B 的一条新纪录到产品维度，行有效时间设为 20030715，行结束时间设为下一条记录的行有效时间。我们需要找出最近的维度纪录，把它的行结束时间设定为我们最新插入纪录的时间日期

2、在产品维度表中察看 20030715 之前的纪录，找出有关 Zippy Cola 的纪录，然后在所有的记录中破坏性的以格式 B 复写公式表达字段。

3、找出 Zippy Cola 的所有事实记录，从 20030715 开始直到这个产品的第一次维度变化。然后破坏性的改变产品外键，设为我们在第一步中创建的自然键。

更新事实表纪录要在测试环境中小心处理，然后才可以实施到生产系统中。若数据库系统对更新操作有保护措施的话，要注意更新不会涉及到海量数据。从操作的目的上看，大量的更新会被分为不同的块，没有必要把时间浪费在百万记录更新失败的回滚上。

下面是一些很巧妙的建议，首先，我们要看到在 20030715 这个时间 Zippy Cola 12-ounce 发生了变化。这样的话，我们只需要执行第二步，而不需要新的维度纪录。

一般来说，在数据仓库中修改劣质数据包含一些逻辑。更改第一种类型字段很简单，我们只需要复写带有期望自然键所有字段的实例。当然，如果聚合表是基于被影响键值的话，还要重新计算。大家可以参看在第六章提及的事实表聚合更新部分。修改第二类型域就要深思熟虑了，因为要修改的可能会涉及时间跨度。

对滞后到达维度纪录的讨论实际上就是维度纪录的滞后到达版本。在实时系统中（会在第十一章讨论），我们会处理真实的最迟到达维度纪录，这些维度记录出现时，事实表纪录已经存在于数据仓库中了。在这种情况下，事实表代理键必须指向一个特殊的临时占位符，直到真实的维度记录生成。

层次维和桥接表

流程检查

规划与设计：需求/现状 -> 架构 -> *实现* -> 测试/发布

数据流：抽取 -> 清洗 -> 规格化 -> *提交*

有时，一个事实表必须支持维度以获取事实表中最低粒度的海量值。下面的例子在其它工具书中也常常被拿来举例：一个针对大量病人的有效诊断，或一个针对多用户分别交易的银行业务等。

如果事实表的粒度没有改变，一个海量值的维度必须通过一个称为桥接表的相关实体连接到事实表。

如图 5.20 健康诊断实例。

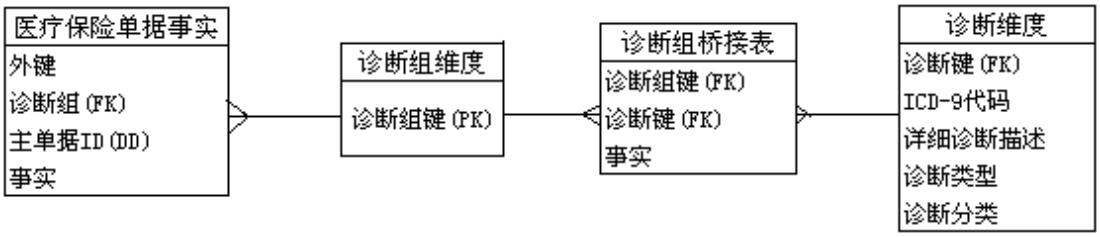

图 5.20　使用桥接表来表示多个诊断

为了避免桥接表和事实表的多对多连接，必须创建一组实体关联到海量值维度。以健康医疗诊断来举例，海量值维度是诊断，实体组是诊断组。这个诊断组就成为事实表的实际标准维度，桥接表记录诊断和诊断组之间的多对多关系。以银行业务为例，当一个业务活跃的记录连接到海量值的储户维度（因为一个业务可以拥有很多储户），这个实体组就是常见的业务维度。

ETL 团队的挑战是如何建立和维持组实体表。以健康医疗举例，病患的治疗记录被提供给系统，ETL 系统有权选择为每个病患的诊断设置一组唯一的诊断组，或者当相同诊断再次发生时重复使用诊断组。这个选择不是那么简单回答的。在设置门诊时，诊断组必须简单，每个相同的诊断将会治疗不同病人。在这种情况下，重复使用诊断组是恰当的选择，见图 5.21。但是在医院的环境中，对于每个病患和住院治疗，这个诊断组是非常复杂的，甚至在每个患者和每种住院治疗间的诊断组是唯一的。见图 5.22 和后续有关时间变化的桥接表的讨论。入院和离院的标志是简单方便的属性，在入院和离院被简单划分的时候可以从侧面判断是否允许诊断。

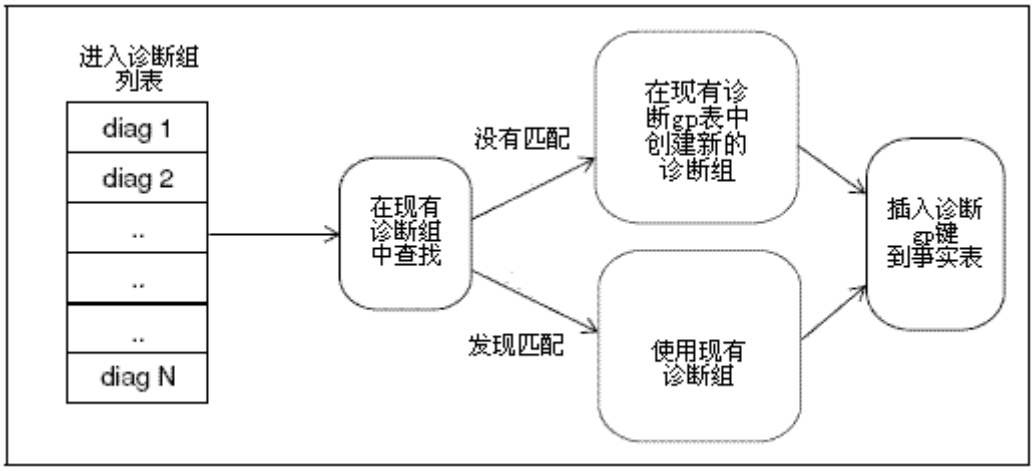

图 5.21　在外部患者设置中处理诊断组

管理加权系数

诊断组表已经在图 5.20 和 5.22 中举例说明了，包括如何明确地按比例分配每种诊断的实际附加(费用美元)的加权系数。当一个请求查询工具限制一种或更多种诊断的时候，工具可以选择乘以对应桥接表中实际附加的加权系数，从而生成一个正确的加权记录。没有考虑到加权系数的查询会导致有偏差的记录。我们清楚与 ETL 系统必须提供的清楚配置一样，加权系数也非常重要。这些配置要么可以从外在其它数据源明确获取，要么依赖于诊断组中诊断的数量简单计算出来。在后面的例子中，如果在集合中有 3 种诊断，那么每种诊断的加权系数就是 1/3=0.333。

图 5.22　适用于医院的基于时间变化的诊断组桥接表

在许多情况中，都是需要桥接表的，但是没有合理分配加权系数的基准。这是大家都能接受的。在这些情况下，用户不可能期望产生正确的加权系数的记录。这些前端复杂模型的议题，例如车祸事故，在《数据仓库工具书》第二版中已经探究到一定深度了。

随时间变化的桥接表

如果海量值维度是第二种缓慢变化维度，桥接表必须是随时间变化的。见图 5.23 中有关银行的举例。如果桥接表不随时间变化，就必须利用客户维度和账户维度的自然键。这些桥接表就会潜在得导致客户和账户之间关系的失真。如果在一个账户中增加或者删除客户，很难明确管理这些有自然主键的表。由于这些原因，桥接表就必须包含代理键。在图 5.23 中，随着客户和账户之间的关系，桥接表的变化非常灵敏。给定账户的新记录，连同开始时间戳和结束时间戳，必须随时添加到桥接表中：

- 账户记录经过 Type 2 更新
- 任何客户记录经过 Type 2 更新
- 一个客户从账户中增加或删除
- 加权系数经过调整

图 5.23　针对客户和随时间变化的桥接表

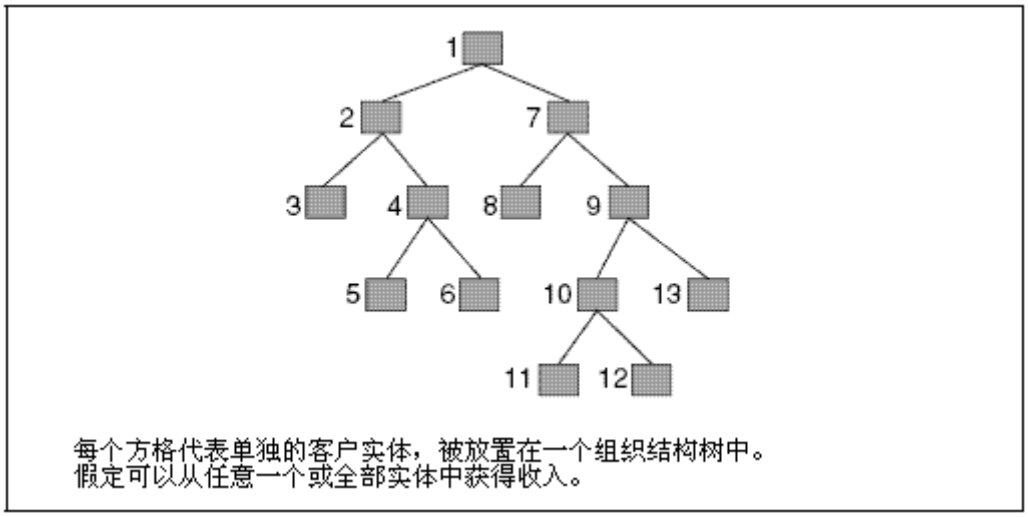

图 5.24　不规则的组织结构层次

不规则层次和桥接表

流程检查

规划与设计：需求/现状 -> 架构 -> *实现* -> 测试/发布

数据流：抽取 -> 清洗 -> 规格化 -> *提交*

不规则层次的不可测深度（层次数）在数据仓库中是重点。在图 5.24 中描述的组织层次是最初的例子。一个典型的组织层次是不平衡的，在深度上没有限制或规则可循的。

这里有 2 种方式构建不规则层次模型，这 2 种都有利弊。在图 5.24 中，在客户层次方面我们将讨论这些利弊的权衡。

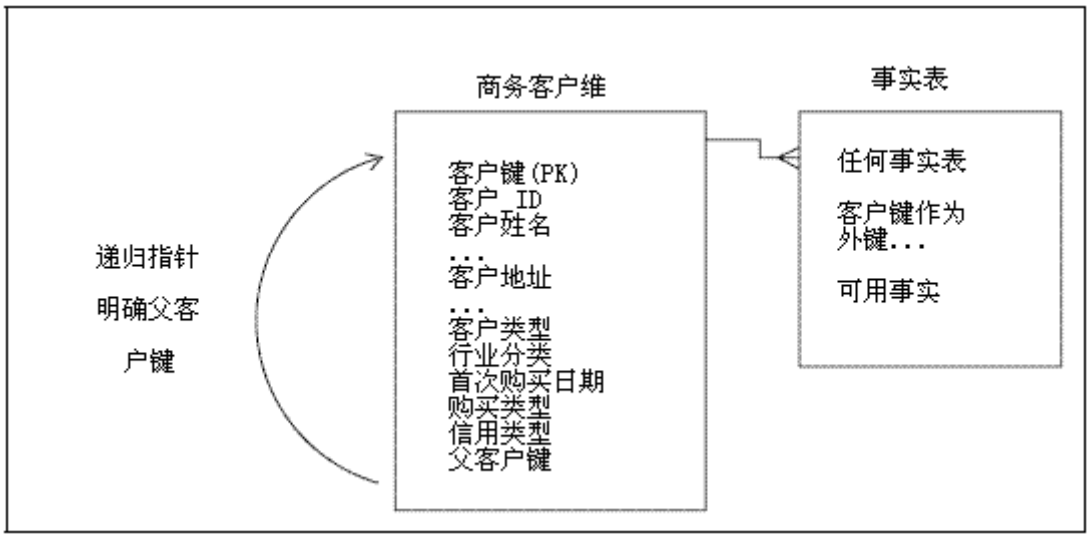

图 5.25　带有递归指针的客户维

递归指针的方法如图 5.25 所示：

- (+)在客户维度中完全深入层次关系
- (+)简单管理增加和移动层次的部分

但

- (−)当维度连接到庞大的事实表时，需要不规则 SQL 扩展查询，查询结果效率低下

- (−)当客户只有一个父亲（实体关系）时，只能表现出简单的树关系（不允许共享所有关系模型）
- (−)不支持不同层次间的转化
- (−)当层次改变时，由于整个客户维度经类型 2 改变，随时间变化层次过于灵敏。

层次桥接表方法如图 5.26 所示：

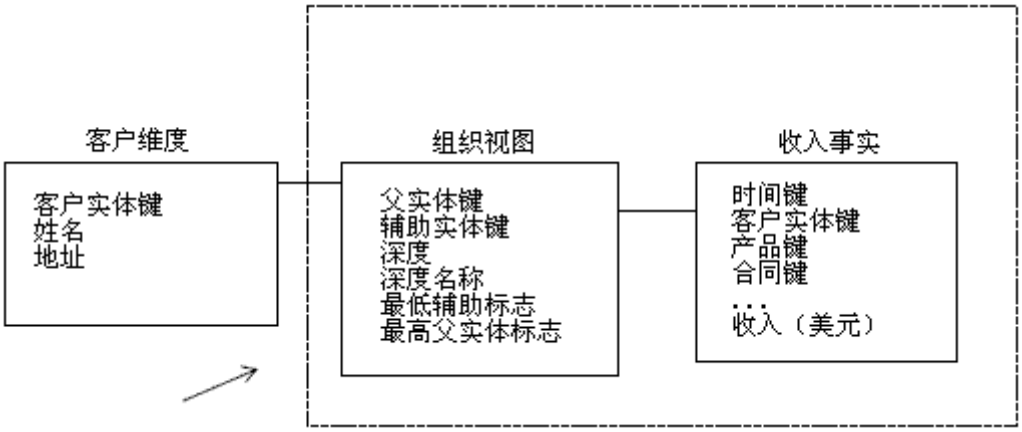

图 5.26　层次桥接表表示客户从属关系

- (+)在桥接表中独立层次关系，不影响客户维度。
- (+)支持单一查询语句的标准 SQL 语法查询，评估整个层次或者指定层次的部分。
- (+)能够很容易普遍的利用共享所有权和组件重用来操纵复杂树型结构
- (+)允许在不同层次间的立即转换，这是因为层次信息集中在桥接表中，桥接表在查询时被选择。
- (+)能够很容易普遍的操纵随时间变化 Type 2 不规则层次，而不影响客户的主要维度。

但是

- (−)需要为每个树关系中的父子关系产生一个独立的纪录，包括二级父母，三级父母，以此类推。尽管记录的精确数值依赖于树关系的结构，大致的浏览规则是树结点是记录的三倍，如图 5.24 所示，3/4 的纪录需要桥接表支持树关系。
- (−)包括比递归指针的方法更加复杂的逻辑，为了增加和移动树关系中的结构。
- (−)需要更新桥接表当 Type 2 改变发生在客户维度内部时。

技术关键点：生成层次桥接表

下文要讲的关于生成概略的层次桥接表的知识已经在 Ralph Kimball 2001 年 2 月的一本月刊上发表过了，因为它和本书 ETL 过程相关，所以我们在这里再次简略的提出来。

这本月刊的文章源自 Ralph’s 在 1998 年 9 月的名为《Help for Hierarchies》的文章(http://www.dbmsmag.com/9809d05.html)，它讲述了分层次的变量深度的结构，这些层次关系在传统意义上象征了相关数据库的递归关系。下面定义了一个简单的公司维度，它包含了外键 PARENT_KEY 和主键 COMPANY_KEY 的递归关系。

```
Create table COMPANY (

COMPANY_KEY INTEGER NOT NULL,

COMPANY_NAME VARCHAR2(50),
```

```
(plus other descriptive attributes... ),

PARENT_KEY INTEGER);
```

当在组织结构上存储信息生效时，不可能操纵或者回滚使用非程序化 SQL 层次里的事实，这些非程序化 SQL 可以通过业务查询工具生成。Ralph 的在最初论述中描述了桥接表解决类似于在组织树关系中每个公司到自身和到它之下每个子公司的每个独立路径记录的这些问题。

```
Create table COMPANY_STRUCTURE (

PARENT_KEY INTEGER NOT NULL,

SUBSIDIARY_KEY INTEGER NOT NULL,

SUBSIDIARY_LEVEL INTEGER NOT NULL,

SEQUENCE_NUMBER INTEGER NOT NULL,

LOWEST_FLAG CHAR(1),

HIGHEST_FLAG CHAR(1),

PARENT_COMPANY VARCHAR2(50),

SUBSIDIARY_COMPANY VARCHAR2(50));
```

在这个例子的最后两列，这张表里对脱离规范化的公司名称的不是必需的，但是规范化后很容易看出接下来该做什么。

接下来的 PL/SQL 存储过程示范了这种从 Oracle 库 COMPANY 表中生成层次桥接表的可实施的方法。

```
CREATE or Replace procedure COMPANY_EXPLOSION_SP as

CURSOR Get_Roots is

select COMPANY_KEY ROOT_KEY,

decode(PARENT_KEY, NULL,'Y','N') HIGHEST_FLAG,

COMPANY_NAME ROOT_COMPANY

from COMPANY;

BEGIN

For Roots in Get_Roots

LOOP

insert into COMPANY_STRUCTURE

(PARENT_KEY,

SUBSIDIARY_KEY,

SUBSIDIARY_LEVEL,
```

```
SEQUENCE_NUMBER,
LOWEST_FLAG,
HIGHEST_FLAG,
PARENT_COMPANY,
SUBSIDIARY_COMPANY)
select
roots.ROOT_KEY,
COMPANY_KEY,
LEVEL - 1,
ROWNUM,
'N',
roots.HIGHEST_FLAG,
roots.ROOT_COMPANY,
COMPANY_NAME
from
COMPANY
Start with COMPANY_KEY = roots.ROOT_KEY
connect by prior COMPANY_KEY = PARENT_KEY;
END LOOP;
update COMPANY_STRUCTURE
SET LOWEST_FLAG = 'Y'
where not exists (select * from COMPANY
where PARENT_KEY = COMPANY_STRUCTURE.SUBSIDIARY_KEY);
COMMIT;
END; /* of procedure */
```

当创建了桥接表时，这个解决方案利用 Oracle 的 CONNECT BY SQL 扩展到数据中的每条树关系。在这个过程中，CONNECT BY 是非常有用的，对于多方面查询，它不可能被特别的查询工具利用。如果某工具可以产生这种语法去探究递归关系，它是不可能在同一声明中连接到事实表的。即使 Oracle 将略微除去这任意的限制，查询时间的性能可能不是太好。

下列虚构的公司数据将帮助你理解 COMPANY STRUCTURE 公司结构表和 COMPANY EXPLOSION SP 公司探究过程。

```
/* column order is Company_key,Company_name,Parent_key */
insert into company values (100,'Microsoft',NULL);
insert into company values (101,'Software',100);
insert into company values (102,'Consulting',101);
insert into company values (103,'Products',101);
insert into company values (104,'Office',103);
insert into company values (105,'Visio',104);
insert into company values (106,'Visio Europe',105);
insert into company values (107,'Back Office',103);
insert into company values (108,'SQL Server',107);
insert into company values (109,'OLAP Services',108);
insert into company values (110,'DTS',108);
insert into company values (111,'Repository',108);
insert into company values (112,'Developer Tools',103);
insert into company values (113,'Windows',103);
insert into company values (114,'Entertainment',103);
insert into company values (115,'Games',114);
insert into company values (116,'Multimedia',114);
insert into company values (117,'Education',101);
insert into company values (118,'Online Services',100);
insert into company values (119,'WebTV',118);
insert into company values (120,'MSN',118);
insert into company values (121,'MSN.co.uk',120);
insert into company values (122,'Hotmail.com',120);
insert into company values (123,'MSNBC',120);
insert into company values (124,'MSNBC Online',123);
insert into company values (125,'Expedia',120);
insert into company values (126,'Expedia.co.uk',125);
/* End example data */
```

这个过程将会生成 27 家公司的记录和创建 110 家公司结构的记录，组成一个有 27 个节

点和 26 个子树的大树关系。对于大的数组，你能发现通过增加一组连接索引在 CONNECT BY 字段，提高执行结果效率。在例子中，你可以创建一个索引在 COMPANY_KEY,PARENT _KEY 和其它的在 PARENT _KEY, COMPANY_KEY。

如果希望使树关系结构原本的展现，下列查询显示了交错的辅助表：

```
select LPAD( ' ', 3*(SUBSIDIARY_LEVEL)) || SUBSIDIARY_COMPANY from

COMPANY_STRUCTURE order by SEQUENCE_NUMBER

where PARENT KEY = 100
```

在 Ralph 的最初论述中已经增加了 SEQUENCE_ NUMBER 的概念，它的数字节点从头到尾，从左到右。它允许修正 level-2 节点在它们匹配的 level-1 节点下被分类。

对于组织树关系的图解版本，看一下 Visio 2000 企业版，它有一个数据库或者文本文件驱动组织图表向导。利用 VBA 脚本教程的帮助，COMPANY _STRUCTURE 表的视图，和事实表，可能自动产生你需要的 HTML 页。

采用维度中的位置属性展现文本事实

对于关系型数据库用 SQL 语句存储一些严格约束的分析类型的接口，需要展现在维度记录上的复杂对比。考虑一个文本事实的例子如下：

假设我们测量关于所有客户的新旧程度，周期，强度的数值型值。可以邀请我们的数据挖掘的同事并且要求他们在这个被新旧程度，周期，强度标示的抽象三维中识别自然群。实际上，我们不要求全部数值化结果；我们希望对数据集市有意义的定义“行为”群。在运行群标识符数据挖掘步骤之后,我们发现，例如，八个自然的客户群。在研究如何在 RFT 维中定义群的中心之后，我们就能够设定八个“行为”群的行为描述。

- A：大量重复客户，信用度高，少产品回报。
- B：大量重复客户，信用度高，但是多产品回报。
- C：近来的新客户，没有确定性用额度的。
- D：非经常的客户，信用度高。
- E：非经常的客户，信用度差。
- F：以前的好客户，最近没有见到的。
- G：频繁窗口客户，几乎没有收益的。
- H：其他

我们可以观察标识 A 到 H 作为一个文本事实概述了一个客户的行为。在数据仓库中没有这么多的事实文本，但是行为标识似乎是最好的例子。我们可以想象一个发展中的行为标识测量的时间系列，来表示一个客户在这段时间内用数据表示每个月的状态。

```
John Doe: C C C D D A A A B B
```

这个时间序列是相当有启示作用的。我们如何组织数据仓库来生成这类报告？而且对于客户而言，我们如何引导他们到我们感兴趣的约束中，仅在这段时间内从 A 群到 B 群？我们甚至需要更复杂的查询，例如查找这些先前在第五、第四、第三，并且先前在第二或者第一阶段是 B 群或者 C 群的客户。

我们可以用不同的方式在这时间段上一系列原有的行为标识来建模。每个方式有同样的信息内容但是在自由运用上是完全不同的。让我们假设，每个月我们为每个客户产生一个新的行为标识。这三种方法是：

1、采用作为原有事实的行为标识，每个月为每个客户的事实表记录。

2、采用行为标识作为一个独立属性，缓慢改变客户维度记录（Type 2）。每个月为每个客户创建一个新客户记录。新记录的相同数字每个月选择#1。

3、采用 24 个月时间系列行为标识作为 24 个属性记录单个客户维度，Type 3 SCD 的一个变量的大量交替出现逼近真实值。

这个区间的完整点是选择 1 和选择 2 ，它们为每个行为标识产生了独立的记录，维持数据仓库有效不可置疑的结构。SQL 没有直接的方法在记录中引导约束。一个有效率，聪明的程序师当然能够做任何事，但是每个复杂约束需要亲手规划。没有标准查询工具能够有效提出设计选择 1 或 2 。

设计选择 3，如图 5.27 所示，完美的解决了查询问题。标准的查询工具依靠这些包含和用户需求一样多的行为标识设计，可以发布非常复杂跨约束查询，这是因为所有的约束目标是在相同记录中的。这个结果维度表也可以在每个低“集的势”的行为标识上使用位图索引技术，所以可以完美展现，甚至适合于复杂查询。

有几种方式可以维持过去文本事实的位置依赖的设置。依赖于如何创建申请，每个抽取样本时期特性可以向后移动，以便每个特定的物理字段总是最新的。没有变化之后，每个月对每个查询都会追踪当前的行为。维度的一个另外的字段应该识别当前最新的是哪一个实际月，因此，最终用户将会了解时间系列已经被更新后的时间。作为选择，维度的域已经能够固定日历注释。最终，本来分配的所有字段将会被填充，而且在那时，将会作一个决定来增加字段。

客户键 PK	客户ID NK	客户姓名	时段1	时段2	时段3	时段4		时段24
12766	A23X	John Doe	C	C	D	A	...	B

图 5.27　使用位置属性来识别文本事实

在维度中描述文本文件的时间系列使用依赖位置的字段，为 Type 3 SCD 的可选择实际设计方法和逻辑一样多。

小结

本章介绍了创建数据仓库维度的最新设计方法。切记 ETL 的底层存在很多不同的数据结构，包括平面文件，XML 数据设置，和整个关系设计，为了在前端最终数据展现的步骤做准备，我们转换所有的这些结构成为维度设计。

尽管维度表几乎总是比事实表小很多，但是维度表提供了数据仓库的结构，提供到事实表度量的引用。

特定的维度表设计具有实践性和普遍性。在这一个章节的每一项技术能在许多不同的主题域被应用，ETL 编码和实际管理是可以重用的。特别是，缓慢变化维度的三种类型（SCDs），已经变成数据仓库设计的基本词表。仅关注 Type 2 SCD 就可以传达处理随时间变化的完整内容，设计主键，创建集合体，然后执行查询。

详尽描述了创建维度的方法，现在我们关注事实表，这些事实表在我们的数据仓库中包含了测量的所有巨大的表。

北京易事通慧科技有限公司（简称“易事科技”，ETH）是国内领先的专注于商业智能领域的技术服务公司。凭借着多年来在商业智能领域与国内高端客户的持续合作，易事科技在商务智能与数据挖掘咨询服务、数据仓库及商业智能系统实施、分析型客户关系管理、人力资源分析、财务决策支持等多个专业方向积累了居于国内领先的专业经验和技能。

作为Solvento集团旗下的联盟公司，易事科技获得授权为客户和合作伙伴提供MicroStrategy产品，SPSS产品，Pervasive产品和i2产品的销售及技术服务。更多的信息请访问公司的官方网址 http://www.ETHTech.com，或拨打电话+8610 68008008。