提交事实表

事实表装有企业的度量数据。事实表与度量的关系非常简单。如果存在一个度量，则它可以被模型化为事实表的行。如果事实表的行存在，则它就是一个度量。那么什么是度量呢？一个关于度量通用的定义是：通过工具或比例等级可以测量观察的数量值。

在维度建模时，我们有意识地围绕企业的数字度量创建我们的数据库。事实表包含度量，维表包含关于度量的上下文。这种关于事物的简单视图被一次又一次的证明是最终用户直观理解我们的数据仓库的方式。这也是我们为什么通过维度模型打包和提交数据仓库内容的原因。

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 章描述了如何创建数据仓库的维表。也许从维表开始介绍会觉得很奇怪，因为度量以及事实表才是最终用户真正想要看的内容，但是维表是事实表数据的入口，事实只有通过维度解释才会变得有意义。由于第 5 章详细完整地描述了维，因此本章的内容多少会变得简单一些。

事实表基本结构

每一个事实表通过表的粒度来定义。事实表的粒度是事件度量的定义。设计者必须至始至终按照度量如何在现实世界中理解来规定事实表的粒度。例如，图 6.1 中的事实表的粒度为指定的零售发票上的单个货品。我们并不从定义这些字段的粒度开始，而是将粒度表示成为维度的外键和事实表的某些字段。粒度定义必须按照现实的，物理的度量意义来定义，然后才考虑维度和事实表中的其他字段等其他因素。

所有的事实表包含了一组关联到维表的外键，而这些维表提供了事实表度量的上下文。大多数的事实表还包括了一个或者多个数值型的度量字段，我们称之为事实（Fact）。请看图 6.1。某些事实表中还包还了一个或者多个特殊的近似维度字段，他们是在第 5 章中介绍的退化维度(Degenerate Dimensions)。退化维度存在于事实表，但是他们不是外键，不关联任何真正的维表。我们在图 6.1 中使用符号 DD 来标识退化维度。

图 6.1　最细粒度的销售交易事实表

在现实实践中，事实表一般都至少包含 3 个维度，而且绝大多数包含更多的维度。由于过去的 20 年中，随着数据仓库以及相应的软、硬件技术的成熟，事实表技术有了很大的提高，可以存储越来越多的最细粒度上量测值。越来越少的度量，越来越多的维度。在最初的零售销售数据仓库中，数据集仅仅是粗粒度的聚合值。这些早期的零售数据仓库通常只有 3 到 4 个维度（产品、市场、时间和促销）。现在，我们搜集销售交易的原子级别的零售数据。一个单独的销售交易中很轻易就包含了图 6.1 中的 10 个维度（日历日；产品；商店级别的现金帐簿；客户；员工；商店经理；价格范围；促销折扣；交易类型和付款方式）。更多的时候我们还要在目前的基础上增加新的维度：比如商店人口统计学指标，市场竞争事件和天气！

事实上，每个事实表都包含一组由表中的字段定义的主键。在图 6.1 中，事实表一个可能的主键是由两个退化维度：发票号（Ticket number）和行号（LineNumber）组成的联合键。这两个字段唯一的定义了出纳登记簿 cashier register 上的单个商品的度量事件。另外一个可能的等价主键是时间戳和出纳登记簿 cashier register 的组合。

如果在设计的时候没有給于足够的注意，那么就可能违反事实表上主键的假设：可能在同一个时段两个同样的度量事件会发生，但是数据仓库团队却没有意识到这一点。显然，每个事实表应该拥有一个主键，即使仅仅是出于管理的需要也应该在事实表设立主键。如果没有主键完整性，那么事实表中有可能存在两个或者更多一样的记录，而且没有办法按照独立的量测事件来区分他们。但是只要 ETL 团队保证单独的数据装载合理地表示唯一的量测事

件，通过在装载时为数据增加唯一的序号就可以在事实表中唯一标识记录。尽管唯一的序号和业务没有关联，而且不应该发布给最终用户，但是它在管理上保证了一个单独和可能的量测的发生。如果 ETL 团队不能够保证单独加载表示的是合法的独立的量测事件，那么在数据加载的时候数据上必须已经正确的定义了主键。

前面的例子表明需要所有的 ETL 作业可以在发布或者发生错误时有再次运行的能力，以保证不会错误的更新目标数据库。在 SQL 语法中，更新不变化的值通常是安全的，但是更新增量的值是危险的。如果主键已经强制定义，那么插入是安全的，因为插入重复的值会触发错误。如果限制是基于简单字段值，那么通常来说删除也是安全的。

确保参照完整性

在维度模型中，参照完整性意味着事实表中的每个字段使用的是合法的外键。换句话说，没有事实表记录包含了被破坏的或者未知的外键参照。

在维度模型中可能有两种情况会导致违反参照完整性：

1. 加载包含了错误外键的事实表记录
2. 删除了维表记录，而其主键在事实表中被使用。

如果没有注意参照完整性，那么就会非常容易破坏它。笔者曾经研究了很多没有严格保证参照完整性的事实表；每个案例中都能够发现严重的冲突。一个事实表记录违反参照完整性（记录中包含一个或者多个外键）不仅仅是讨厌，更严重的是这非常危险。推想一下，一条合理的记录正确的记录了量测事件，但是不正确的存储在数据库中。任何使用了坏维度的事实表查询将不能包含该记录，因为按照定义，维表和该事实表记录的关联不会发生。但是在动态聚合中忽略该坏维度的查询结果却包含了这条记录。

在图 6.2 中，我们显示了在 ETL 过程中可以保证参照完整性的三个主要位置。

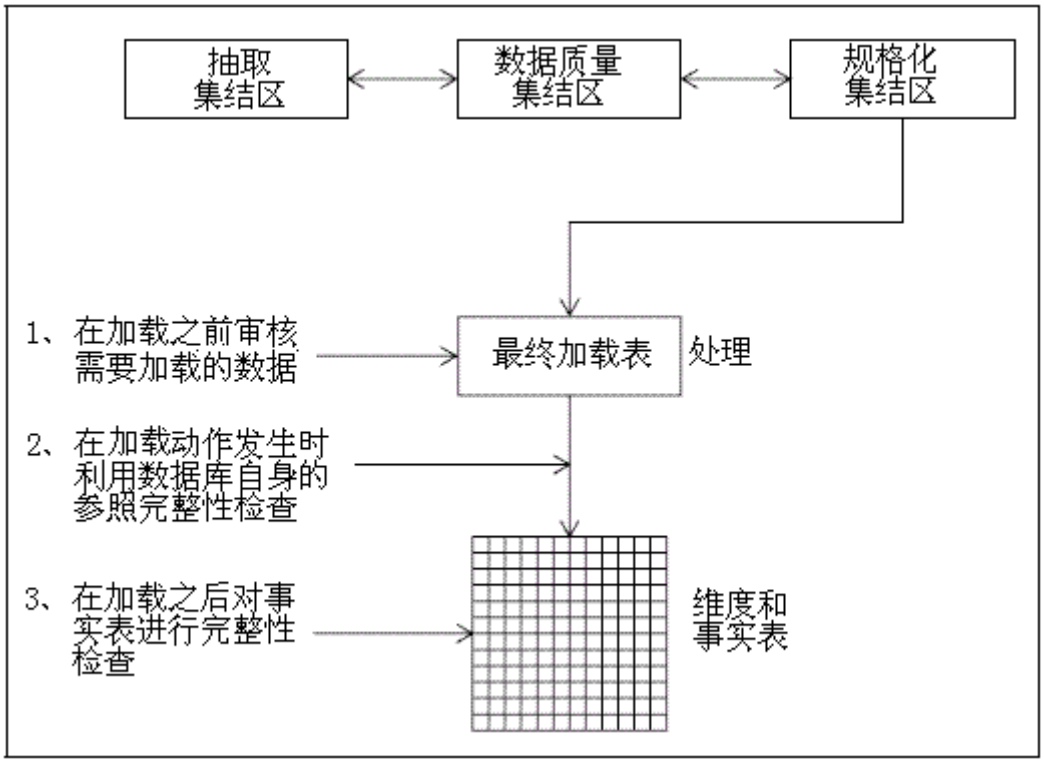

图 6.2　强制参考完整性选择

1 在将事实数据加载到最终表之前，仔细的审核和准备数据，在删除任何维表记录之前，仔细地审核

2 在每次事实表插入操作和每次维表删除操作动作发生时，使用数据库自己的参照完整性检查

3 加载后对事实表进行扫描检查，以发现和更正有问题的外键，解决参照完整性冲突。

从实践角度来讲，第一个选择更加可行。在数据加载到事实表中最后一步就是查找事实表记录中的自然外键，然后将他们替换成为维表中的代理键。这个过程将在下一节代理键环节中仔细地介绍。这个过程的核心是一个特殊的查找表，它包含了每一个外来的自然键所使用的代理键的值。如果这张表被正确的维护，那么事实表中的记录将满足参照完整性。同样在维表记录被删除的时候也需要尝试联结事实表和维表，只有在查询返回 null 的时候才能够删除该记录。

第二种选择利用数据库的参照完整性来检查的方法虽然很自然，但是在批量的加载成千上万条甚至几百万条记录的时候效率太低。当然这只是一个软件技术的问题，如 Reb Brick 数据库系统（现在 IBM 在销售）设计为在任何时候都维护参照完整性，它可以在 1 个小时内向事实表加载 100 万条记录，同时检查所有维度的参照完整性。

第三种在数据库更新完成后检查参照完整性的方法在理论上可以发现所有的冲突，但是花费的代价可能惊人的高，对参照完整性的检查需要使用：

```
select f.product_key

from fact_table f

where f.product_key not in (select p.product_key from

product_dimension p)
```

在拥有几百万行产品维度数据，以及数亿行事实表纪录的环境中，这是一个荒谬的查询。当然可以将查询仅仅限于当天加载的数据，但这需要假设时间维度的外键是正确的为前提。但是使用第一种选择时，这种方法可以作为一种额外的检查。

代理键管道

在建立事实表时，最终的 ETL 步骤是将新数据记录的自然键转化成正确的、当期的代理键。在这节，我们假设所有的记录已经加载到事实表中。换句话说，我们需要使用为每个维度实体（如客户或者产品）使用当前的代理键。我们将在这章的最后处理迟到的事实记录。

我们理论上可以通过在每个维表中获取最新的记录来为自然键获得当前的代理键，这在逻辑上是正确的，但是很慢。替代方法是为每一个维度维护一个专门的代理键查找表。这张表在新的维度实体创建的时候或者记录在发生类型 2 缓慢变化维度 2 的更新的时候被更新。我们在第 5 章讨论图 5.16 的时候介绍了这种表。

维表在插入或者缓慢变化维度 2 更新发生之后，在处理事实表之前，这个维表必须被完全的更新。在更新事实表之前的维表更新过程是维护维表和事实表参照一致性的一般过程。反向的过程在删除记录的时候发生，首先我们删除不需要的事实表记录，然后删除不再联结到事实表的维度记录。

不必因为事实表不再参照该记录而删除维表记录。即使事实表中没有引用该维

度实体，维度实体也可能需要存在或者保存在维表中。

当我们更新维表的时候，不仅仅要更新所有的维表记录，还要更新存储当期的数据仓库键和自然键关联关系的代理键查找表。

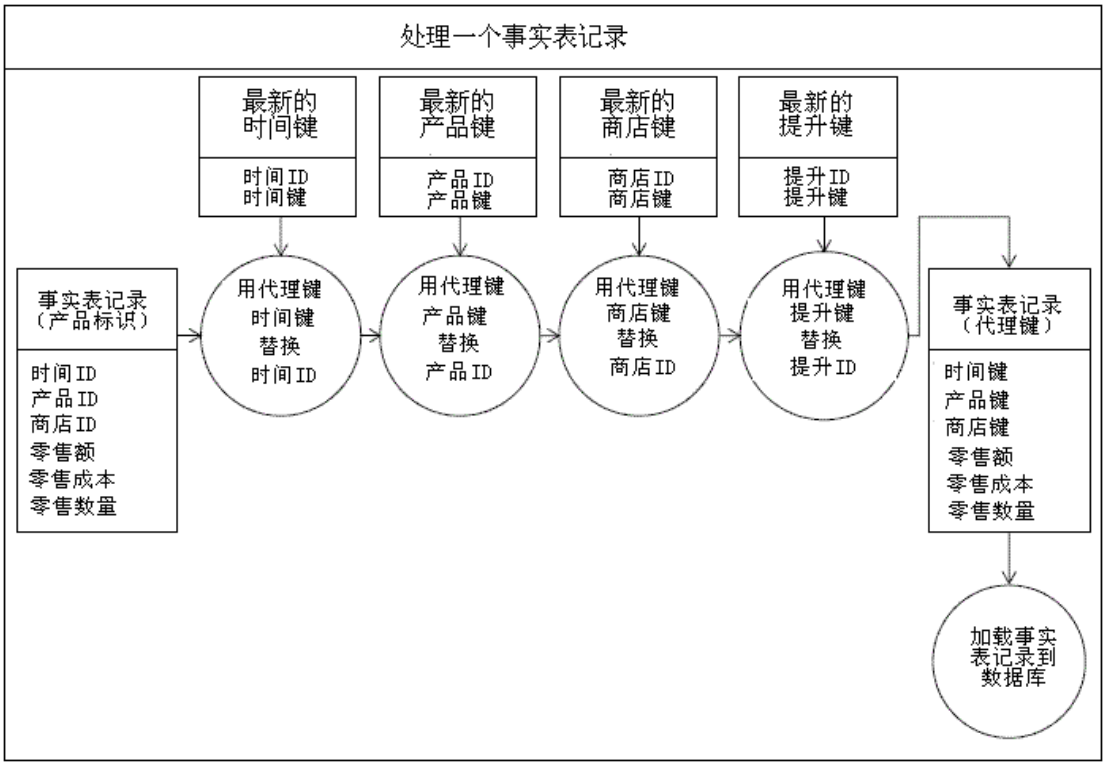

图 6.3　代理键管道

我们处理新增的事实表记录的过程非常容易理解。请看图 6.3。我们从新增事实表记录中获得每个自然键键值，然后替换成对应的当期代理键。注意我们的提法：替换。我们在事实表记录中并不保存自然键。如果你关心具体的自然键值，你可以关联维表记录得到。

如果你有 4 到 12 个自然键，每个新增的事实表记录需要 4 到 12 个单独的查找表来获得正确的代理键。首先，我们建立多线程的应用来流化所有的输入数据记录，使其通过图 6.3 中的所有步骤。这里的多线程，指的是记录#1 在完成连续的键查找和替换的同时，#2 在后面进行相应的处理，我们并不在第一个查找步骤中处理所有的新增记录，然后将整个文件交给下一步。在经过所有必要的处理过程后才将所有的记录写到磁盘意味着更高的效率。在处理过程中应该尽量充分使用内存。

如果可能，所有需要的查找表应该常驻内存，以便每条事实表记录在处理它的自然键的时候随机访问。这是将查找表从原始数据仓库维表中分离的一个原因。假设对于某个维度，我们用上百万记录的查找表，如果自然键有 20 字节而代理键为 4 字节，我们需要 24MB 的内存来保存查找表。在使用 4-8GB 内存的机器作为数据准备机 data-staging 的环境中，我们可以非常容易在内存中加载所有的查找表。

我们前面的讨论的架构是如何设计发挥最大性能。如果你仅仅是每天加载几万条数据，而且装载窗口很大，那么你不需要强调这种级别的性能。你可以定义事实表和维表的星型联结，将自然键转化成代理键，用 SQL 处理整个过程。这种情况下，如果新增的数据不能匹配维度（参照完整性失败）你还可以对某些维度定义外联结。

可以在这种环境中使用程序工具 awk，因为他支持在内存中使用自然键作为下标建立 Array 来完成自然键到代理键的翻译。这样，如果你定义 Translate_Dimension_A[natural_key] = surrogate_key，那么处理每一条事实表记录就非常简单：print Translate_Dimension_A($1), Translate_Dimension_B($2) 。

在一些重要的大事实表中，你可能会遇到不寻常的 monster 维度，例如居民客户，可能拥有上亿条记录。如果我们只有一个这样的维度，那么我们仍然可以定义一个快速传递 pipelined 代理键系统，虽然这样巨大的维表在查找的时候会大量的读取磁盘。秘密在于按照维度的自然键对使用的事实表和查找表进行排序。这样，代理键替换变成了在两个文件之间一次性的按照顺序的合并。这个过程相当的快，尽管没有在内存中处理。如果你有两个这样的 monster 查找表在你的处理流程中 pipeline 中，而且不能在内存中处理，那么对于非排序的维度 Key 键上的随机访问所带来的 I/O 是难以承受的。

由于在源系统中有可能没有参照完整性检查，这样对事实表记录进行代理键处理的时候要考虑处理一些无效的自然键。这种情况下，我们建议创建一个新的代理键，增加一条维表记录，标示为 Unknown。如果无法对数据最终更正，那么每个无效的自然键都应该给一个唯一的新的代理键。如果你的业务本身要求这些无效的自然键必须保持并且不被处理，那么在所有受到影响的维度中都只需要有一个单个的唯一的代理键（缺省的 Unknown 记录）。

使用维表而不是查找表

上一节介绍的查找表对于解决那些处理得事实表数据记录都是同阶段的，换句话说，都是当前的非常有效。如果有相当数量的事实记录延迟到达，那么查找表就不再适用，这时候必须使用维表作为正确的代理键的源。当然，这种假设的前提是维表的设计要遵循第 5 章的建议。

避免使用独立的查找表还可以简化事实表加载之前的 ETL 管理工作，因为这种情况下不需要同步维表和查找表。

某些 ETL 工具套件通过查找事实表记录中的自然键创建高速、内存缓存，然后通过实时查询维表来获得相应的代理键。 这种工作方式如果可行就可以避免使用查找表。可能的缺陷是在访问大维表时动态创建缓存的代价会增加。如果这个动态查找过程绑定事实记录中的自然键和时间戳，那么就可以查询那些搜索代理键的历史值，这种方法可以很高效的处理那些迟到的事实表记录。对于这个问题，您应该询问具体的 ETL 厂商。

基础粒度

由于事实表中存储了企业中所有的数值度量，可以猜想那会有很多的事实表类型。但是事实上，事实表可以归入三种基础类型。我们强烈推荐在每次设计的时候坚持使用这三种简单的类型。设计者通过混合使用这些简单类型构造更复杂的结构的时候，实际上是将避免发生严重错误的巨大的负担推给了最终用户的查询工具和应用。换句话说规则是每个事实表应

该有且只有一个粒度。

这三种事实表类型是：交易粒度，周期快照和聚合快照。我们将在下面讨论这三种粒度。

交易粒度事实表

交易粒度表示的是在特定时间、空间点上的一次瞬间的测量。典型的例子是零售交易。当产品通过扫描器时，扫描器就会发出蜂鸣声（只有扫描器发声），那么一个记录就被记录下来。交易粒度记录只有度量事件发生的时候才被记录。这样，交易粒度事实表既可以为空的也可包含成百亿的记录。

我们讨论过交易粒度事实表的原子粒度可以包含很多的维度。你可以参照图表 6.1 展示的零售扫描事件。

像一个零售商店环境中可能只有一个可以度量的事务类型。而其他的环境，如保险流程，在数据流中可能混杂了多种的交易类型。这种情形下，数字度量字段通常标识为 Amount，同时需要一个交易维度来解释是什么交易。参照图表 6.4，在交易粒度表中的数值度量一定是参照某个度量事件，不会有时间跨度，或者是其他的时间。换句话说，事实必须属于同一个粒度。

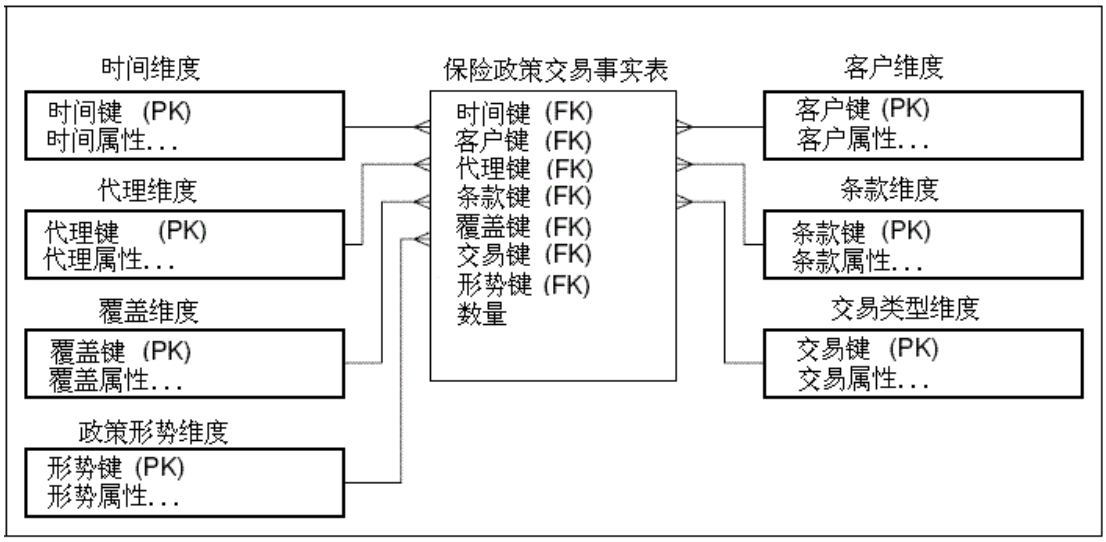

图 6.4　保险标准的交易粒度事实表

交易粒度事实表是三种类型事实表中最大和最详细的。由于单个的交易往往带有时间戳，交易粒度的事实表通常用于复杂和分析。例如，在保单处理环境中，需要交易粒度事实表来描述复杂的保单处理的交易过程，分析不同类型的交易处理的时间。在其他两种类型事实表中不存储这些信息。但是不是所有的周期快照和聚合快照都可以从交易粒度的事实表通过常规的聚合得到。在保险环境中，保费业务系统能够在每月为每个产品生成一种有利可图的保险费措施。这个度量值必须存进每月的周期快照中，而非交易粒度事务表。计算保费的商业规则非常的复杂，数据仓库不可能利用那些低粒度的交易来进行很高效的计算。

交易的时间可能为分，秒或微秒，这些时间需要通过日历组件以及相应的自然日历维度的外键系统来模式化，在事实表中按照第 5 章图 5.5 介绍的使用 SQL 中完整的日期时间数据类型存储。

由于交易粒度事实表具有不可预测的数据分布，前端应用不能在查询中假定任何主键集。这个问题当客户维度需要和人口统计学行为维度进行匹配的时候显得更为突出。如果限定过于的狭窄（例如某个日历日），查询可能不会有任何的结果，和人口统计学条件匹配的

客户被忽略了。考虑到这个问题的数据库架构师通过指定一个事实无关的覆盖表，该表中存储了所有意义的键的组合，以便确保应用能够使客户和人口统计学指标匹配。本章后面将讨论事实无关事实表。我们马上会在后续的章节中介绍周期快照事实表，他很好的解决了数据稀疏问题，但在主键上数据是稠密的。

在理想的环境中，同代的交易级别事实表记录是通过一个固定的时间间隔大批量的抽取到数据仓库中的。大多数情况下，大的事实表在典型的 DBMS 环境下应该按照时间分区。这样就允许 DBA 可以将时间最近的分区上删除索引，提高批量导入到该分区的速度。在数据加载完成后，需要恢复索引。如果分区可以被重命名和交换，那么在数据加载的时候，可以在数据加载的时候将事实表短暂的离线。这是一个复杂的问题，同索引策略和物理数据存储的不同有密切关系。

图 6.5　银行中帐户检查的周期快照

可能要在事实表上存在一个不依赖于分区逻辑的不可删除的索引。另外一些并行处理数据库技术物理上分发数据，这样最新的数据可能不存储在一个物理分区上。

当进入的交易数据是按照数据流的方式，而不是离散的基于文件的装载方式，我们将进入实时数据仓库的领域，关于实时数据仓库将在第 11 章介绍。

周期快照事实表

周期快照事实表表现的是一个时间段，或者规律性的重复。这类表非常适合跟踪长期的过程，例如银行账户和其他形式的财务报表。最常用的财务上的周期快照事实表通常有一个月粒度。在周期快照事实表中的数据必须符合该粒度（就是说，他们必须量测的是同一个时间段中的活动）。在图 6.5 中，我们显示了一个银行支票账户的按照月的周期快照。这个设计很显然的特点就是大量的事实。任何账户的数值度量都是基于时段且有意义的。由于这个原因，在其整个生命周期内，周期快照事实表的良好修改应该是增加该粒度上更多的事实。对于良好修改的介绍见后续章节。

在周期快照事实表中的日期维度是周期。所以，对于月度周期快照的日期维度是日历月的维度。我们在第五章对生成这样的聚合日期维度进行了讨论。

一个有趣的问题是在周期快照记录中如何处理非时间维度的代理键。由于周期快照只有在该周期过去以后才能够产生，最常见的非时间维的代理键的选择是在对应周期内的最后的值。例如，在图 6.5 中账户和机构维度的代理键应该是周期期末的值，尽管在该周期内账户和机构的描述可能改变。这些中间代理键在月度周期快照中并不出现。

周期快照事实表可以完全知道数据的分布，在图 6.5 中的账户活动事实表每月每账户有一个一条记录。只要账户还在活跃，应用就可以假定每个查询中涉及不同的维度。

发布给最终用户应用使用的事实表可以预测数据分布，但是原始的源表却不能。你需要将周期快照事实表的主要维度和原始的源表进行外连接来确保你为每种合理的键组合生成了记录，即使其中一些在当前的加载过程中还没有出现。

周期快照事实表和那些交易粒度表有着同样的加载特性。当数据加载到数据仓库过程中，在每个周期性加载过程中所有的记录将按照最近的时间分区进行集合分布。

然而，对于周期快照事实表的维护策略有两点细微的不同。传统的策略是当周期过去后，一次性的加载所有的记录。进一步的，周期快照维护了一个当前紧迫滚动周期。图 6.5 银行事实表可以有 36 个固定的时间周期，表示了过去 3 年的活动，另外还有一个特殊的第 37 个月，其记录在当前周期的每天晚上进行增量更新。如果最后一天的数据按照通常的方式正确的加载，那么第 37 个月的统计数据就是正确的。如果最后的周期快照不同于最后一天的加载，那么这个策略就不再吸引人了，因为 behind-the-scenes 后台总帐的调整在月末结账过程没有在常规的数据下载中得到反映。

当紧迫滚动周期整天不断地通过数据流的方式更新，而不是通过周期性文件加载，我们将进入实时数据仓库的领域，关于实时数据仓库将在第十一章介绍。

由于用于期末度量计算的业务规则非常复杂，创建一个持续更新周期快照可能很困难，甚至是不可能的。例如，对于保险公司，保费计算是在期末通过业务系统计算的，而且这些计算仅仅能够在期末计算。数据仓库不能很容易得在期中算出保费；另外，业务规则的复杂程度也超过了普通 ETL 转换逻辑处理的范畴。

聚合快照事实表

聚合快照事实表用于描述那些有明确开始和结束的过程，例如合同履行，保单受理以及

常见的工作流。聚合快照不适合长期连续的处理，如跟踪银行账户或者描述连续的生产制造过程，如造纸。

聚合快照事实表的粒度是一个实体从其创建到当前状态的完整的历史。图 6.6 显示了一个聚合快照事实表，它的粒度是发货发票上的行项目。

聚合快照事实表有几个特殊的属性。最明显的是图 6.6 中大量的日历日期外键。所有的聚合快照事实表都拥有一组日期，按照表的标准模式来创建。对于图 6.6 中发货发票行项目的标准模式为：订单日期(order date)、请求发货日期(requested ship date)、实际发货日期(actual ship date)、交付日期(delivery date)、上次付款日期(last payment date)、退货日期(return date)和结算日期(settlement date)。我们可以假设在发货发票创建的时候会创建一条记录。创建时，仅仅有订单日期和请求发货日期是已知的。这条记录插入了事实表，并且对两个已知的日期使用了外键。剩余的外键都是不可用的的，他们的代理键必须指向日历日维表中的对应 Not Applicable 的特殊的记录。随着时间过去，事件发生，原始的记录被修改，对应到其他日期的外键被修改指到了实际的日期。上次付款日期随着付款发生被更改了多次。

图 6.6　粒度为发票行级的累积快照事实表

对于正常的订单，退货和结算日期可能永远也不会修改。

在聚合快照事实表中的记录当事件发生的时候会被重写。注意在 Oracle 中，单条记录的大小依赖于其内容。所以 Oracle 中聚合快照事实表的大小总是在增长的。这将影响到磁盘块的使用。当由于有大量的改变造成大量块碎片的时候，卸载和重新加载记录是非常有必要的，这可以提升性能。另外的方式是根据两个维度，如日期和当前状态（Open/Closed）进行表分区。使用当前状态进行初始分区，当条目变为闭合状态 Closed 的时候，将其转移

到另外的分区。

一个聚合快照事实表是表现有确定开始和结束的处理过程的非常高效的，吸引人的方法。如果过程场景中有越多地过程可以由在事实表中时间定义，那么最终用户应用就越简单。如果最终用户经常需要了解复杂和不正常的状态，例如货物损坏或者发货地址错误，那么最合适的还是交易粒度事实表，它可以查看发货过程中的所有事件。

准备装载事实表

在本节中，我们将介绍如何高效的处理数据加载并克服常见的效率障碍。如果不能采用正确的方法，那么数据加载对于 ETL 开发者将是一个恶梦。接下来的 3 节中将概述你将面对的障碍。

管理索引

索引对于查询来说可以提升性能，但是在数据加载的时候起到的却是相反作用。如果不能很好的处理，那些使用了大量索引的表将导致你的处理变得非常缓慢。在开始加载数据以前，在前置任务中需要删除所有的索引，然后在后置任务中重建所有的索引。如果在加载过程中包含数据更新（Update）操作，需要将那些要执行更新的记录和其他仅仅需要简单插入的记录分开，单独处理。简单的说，请按照下列的步骤来避免索引导致的 ETL 处理瓶颈：

1. 从要插入的记录中分离那些需要执行更新操作的记录
2. 删除那些更新操作不需要的索引
3. 加载那些需要更新的记录
4. 删除剩余的索引
5. 通过批量加载方式执行数据插入
6. 重建所有的索引。

管理分区

为了管理或者提高查询性能，使用分区技术将数据库表（及他们的索引）在物理上分为一些小的表。分区技术的最终优势是，当一个查询要从 10 年的数据中获得一个月的数据的时候，无需扫描所有的数据，而是直接从包含该月数据的分区中获得数据。表分区可以极大的提高对大事实表的查询的性能。表分区对于最终用户是透明的。只有 DBA 和 ETL 团队才会关心分区。

最常用的事实表分区策略是按照日期字段来分区。原因是日期维度是预加载和静态的，我们可以明确的知道代理键是什么样的。通常我们都可以发现设计者为了分区的目的在事实表中加了时间戳字段，但是除非在用户查询的时候包含了时间戳的限定，否则优化器并不会使用分区。由于用户通常都会使用基于日期维度的限定，这就需要使用关联到日期维度的事实表主键进行分区，这样才可以让优化器利用分区。

对于按照时间间隔进行分区的表中经常使用的是年、季度、月。对于那些异常巨大的事实表可以使用周，甚至是日进行分区。通常数据仓库的设计者需要和 DBA 团队共同决定每个表的分区策略，需要通知 ETL 团队哪些分区表需要 ETL 维护。

如果 DBA 团队在分区管理上不占主导地位的话，那么 ETL 过程必须管理他们。如果你

的加载频率是月度，分区维护将很简单。当你的加载频率和分区不同，或者分区不是基于时间的，那么这个处理将会变得具有挑战性。

假设你的事实表按照年进行分区，并且前 3 年的数据由 DBA 团队创建。在 2004 年 12 月 31 日以后，当你试图加载数据的时候，在 Orcale 中，你将收到如下的错误：

```
ORA-14400: inserted partition key is beyond highest legal partition key
```

这时候，ETL 处理有两种选择：

- 通知 DBA 团队，等他们手工创建好下一个分区，然后继续
- 动态创建下一个分区，以备数据加载的需要

一旦可以识别进入数据的代理键，ETL 过程可以通过将最新的日期键和最新数据分区表中最大的值进行比较，判断新进的数据是否适合数据库中已有分区。

```
select max(date_key) from 'STAGE_FACT_TABLE'
```

同下面的值进行比较

```
select high_value from all_tab_partitions

where table_name = 'FACT_TABLE'

and partition_position = (select max(partition_position)

from all_tab_partitions where table_name = 'FACT_TABLE')
```

如果进入的数据是已定义的分区的下一年的数据，那么 ETL 过程可以使用过程前脚本创建下一个分区。

```
ALTER TABLE fact_table

ADD PARTITION year_2005 VALUES LESS THAN (1828)

--1828 is the surrogate key for January 1, 2005.
```

我们讨论的维护过程可以通过写存储过程，在每次加载前供 ETL 过程调用。存储过程中可以产生所需的 ALTER TABLE 语句，根据进入数据的年份插入需要的 1 月 1 日的代理键。

覆盖回滚日志

根据设计，任何关系型数据库管理系统都支持处理事务中错误处理。通过在日志中记录每一个事务，系统可以从未提交的事务错误中恢复。当错误发生的时候，数据库访问日志，撤销任何尚未提交的事务。提交（Commit）事务意味着你或者你的应用明确的通知数据库事务中每个处理都已经完成，事务应该永久的写入磁盘。

回滚日志又叫做 redo 重做日志，这对交易系统（OLTP）来说是异常重要的。但是在数据仓库环境中，所有的事务都是由 ETL 过程管理的，回滚日志仅仅是为了获得优化的加载性能增加的额外的特性。数据仓库不需要回滚日志的原因包括：

- 所有的数据通过受 ETL 系统管理的进程输入
- 数据批量的加载
- 如果加载过程失败，数据可以很容易的重新加载

每种数据库管理系统都有不同的日志功能，采用不同的方式处理回滚。

装载数据

完成一个新表的首次加载过有一些需要解决的问题。最大的挑战是一次加载极大量的数据。

- 单独处理数据插入。很多 ETL 工具（以及一些数据库）提供 update else insert 功能。这个功能非常方便，且有着非常简单的数据流程逻辑，但是性能非常的低。ETL 过程对已经存在的数据的更新逻辑中包括区分那些已经存储在事实表中记录和新数据。当处理大量的数据的任何时候，你想到的是数据批量加载到数据仓库。但是不幸的是，很多批量导入工具不支持更改已经存在的记录。通过分离需要更新的记录，你可以先处理更新，然后再执行批量的导入，这样获得最佳的加载性能。
- 利用批量加载工具。使用批量加载工具，而非使用 SQL 语句加载大量数据可以降低数据库负载，并极大的提高加载效率。
- 并行的加载。在加载大量数据的时候，将数据物理上分成不同的逻辑段。如果加载 5 年的数据，你可以做 5 个数据文件，每个文件中包含一年的数据。一些 ETL 工具允许你根据数据范围进行数据分区。一旦数据被分成均等的部分，运行 ETL 过程并行的加载所有的分段。
- 最小化物理更新。在表中更新记录操作需要耗用 DBMS 很多资源，最大的原因是数据库要生成回滚日志。要最小化对回滚日志的操作，可以采用批量的加载数据。如何处理那些需要更新的数据呢？很多情况下，最佳的方式是删除要更新的记录，然后批量的加载所有的数据。由于要做更新的数据和总的数据量的比率会极大的影响优化方式的选择，因此需要一些反复测试来判断针对具体情况的最终加载策略。
- 在数据库外进行聚合。在数据库之外进行排序，合并和聚合要比在 DBMS 内使用 SQL 语句，使用 COUNT 和 SUM 函数，GROUP BY 和 ORGER BY 关键字高效的多。ETL 过程需要将巨大数量的数据进行排序、合并放在进入关系型数据库准备区之前完成。很多 ETL 工具提供这些功能，但是专门的工具在操作系统级别执行排序/合并意味着为处理大数据集进行额外的投资。

ETL 过程应该最小化那些通过数据库批量加载所完成的更新和插入操作。如果需要大量的更新，请考虑通过批量加载工具截断和重新加载整个事实表。当更新量很小的时候，考虑分离那些需要更新的记录，对其单独的处理。

增量装载

增量加载用于周期性的加载，目的是同步数据仓库和相应的源系统。增量加载可以以任意的时间间隔甚至连续（实时的）进行。在写本文的时候，常用的加载的时间间隔为每天，但是这不是一个确定的或者最佳的加载时间间隔。用户通常倾向于按日加载，因为他们通常将日粒度的静态数据存储在数据仓库中，而避免加载瞬间变动的数据，因为数据不断的变化，将导致同一天内的报表结果不一致。

ETL 常规的数据加载实际上是将历史数据初始加载到数据仓库中处理的特例。一个有用的建议是保持历史加载和增量加载处理过程的一致性。ETL 团队必须将抽取处理的开始日期和结束日期可参数化，这样 ETL 程序既可以很灵活的加载小的增量部分，也可以一次加载全部的历史数据。

插入事实

当创建新的事实记录，你需要尽快地获得数据。如果利用数据库批量加载工具，那么事实表可能对于 SQL INSERT 语句来说太大了。使用 SQL INSERT 语句会导致产生一些数据库日志，这些日志的目的是错误恢复，但是对于数据仓库环境他们完全是多余的。如果加载程序失败了，ETL 工具能够从错误中恢复，并从错误点开始处理剩下的部分，而不需要使用数据库的日志。

在主流的 ETL 工具中，错误恢复是一个普遍的功能，不同的厂商处理错误、错误恢复的方式不同。请确认 ETL 厂商如何解释他们的错误恢复机制，并选择需要手工干涉最少的产品。在 ETL POC 中，请测试工具的错误恢复能力。

更新和纠正事实

我们已经在很多地方讨论了对数据仓库数据的更新，尤其是事实数据。大家都认可一点，除了缓慢变化维度，其他维度必须准确地反映出对应数据源中的数据。但是，对于在数据仓库中的事实数据得更新，仍然有几个问题需要讨论。

最大的争论是数据仓库必须反映在业务系统的所作的所有变化，这种观点仅仅是理论上的，现实中很多数据仓库不是这样的。毕竟数据仓库目的是支持对业务的分析，而不是针对数据的来源系统。数据仓库要正确地反映业务活动，就必须要准确地描述实际的事实。无论如何讨论，数据条目错误都不是一个业务事件（除非你建立一个数据集市专门来分析数据条目的精确性）。

记录下与正确记录发生矛盾的不必要的记录可能会导致矛盾的结果，会干扰分析结果。考虑下列的例子：一个公司发售了 1000 个苏打水容器，在源系统中记录为 12 盎司听装。在数据发布到数据仓库时发现了一个错误，应该为 20 盎司瓶装。发现问题后，源系统立即进行了更新。业务上从来就没有卖过 12 盎司听装，但是在执行销售分析的时候，业务上不需要知道发生过错误，相反的，保留错误的数据可能导致将结果错误的解释为 12 盎司听装。在数据仓库中通过 3 种基本的办法来更正数据错误：

- 消除事实
- 更新事实
- 删除和重新加载事实

所有三种策略都产生了实际的效果——1000 瓶 20 盎司每瓶苏打的销售。

消除事实

消除错误继承是创建一个错误记录的完整备份，但是度量是原始度量乘以-1。这样在反向事实表中负的度量就可以将原始的错误记录影响剔除。

使用消除事实而不是使用其他方法更正事实有很多原因。最主要的是审计的需要，如果需要对获取数据错误进行分析，那么消除错误将是一个好的建议。但是，当捕获实际的错误对于业务非常重要的时候，交易系统中应该有自己的数据条目审计能力。

使用消除事实，而不是更新或者删除的另外一个原因是考虑数据量和 ETL 效率。当事实表中有几千万记录的时候，必须考虑搜索和更新已有记录所带来的 ETL 性能的降低。ETL

团队有责任为优化性能提出自己的方案。你不可以基于技术标准来改变业务规则。如果业务上要求清除错误，而不是消除他们，那么你就有责任满足这个需求。本章讨论了一些可以保证你的过程是最优化的选择。

更新事实

在事实表中更新事实可以理解为是一种加强处理。在大多数的数据库管理系统中，为了做回滚保护，一个 UPDATE 操作会自动的在数据库日志中进行记录。而数据库记录日志将极大的降低加载的效率。最佳的更新事实表的方法是通过批量加载工具 REFRESH 表。如果你不得不使用 SQL 来 UPDATE 事实表，需要确保那些能够唯一标识行的列都进行了索引，另外的索引需要被删除。不必要的索引将极大地降低更新操作的性能。

删除事实

大部分人认为删除错误记录是更正事实表中错误数据的最佳方法。这样做的缺点是操作后报表将和以前的报表结果不一致。如果你采用这种方式，需要有办法更新原来的报表来保持一致性。大多数情况下，如果当前的版本能够正确地反映实际情况，那么更改数据也不是一件坏事情。

理论上，从事实表中删除事实在数据仓库中是被禁止的。但在实际中，在大多数数据仓库中删除事实是比较普遍的。如果你的业务上需要删除，那么可以采用两种方式：

- 物理删除。在大多数情况下，人们不希望看到源交易系统中不存在的数据。当需要物理删除的时候，你需要按照业务规则，删除不想要的数据。
- 逻辑删除。逻辑删除事实表记录是一种安全删除。一个逻辑删除需要利用一个称为 Deleted 的额外的字段，通常为 Bit 或者 Boolean 数据类型，作为事实表中的标记字段表示该字段是否已删除。对于逻辑删除方法需要特别注意的一点，是需要在每个查询中都使用限定来过滤到那些已经被逻辑删除的记录。

物理删除事实

物理删除意味着数据已经从数据仓库中永久的删除。在执行物理删除的之前，必须提醒用户一旦删除后，数据将不能再被访问。

用户通常有一个误解，一旦数据进入数据仓库，就永久的存在。所以当用户说他们从来没有（Never）理由查看（See）删除的数据，需要对这两个词进行澄清，确认客户知道自己说的正确的表达了自己的意思，且用了正确的词汇。

- 从来不（Never）。用户会很自然的按照当前的需求出发考虑问题，因为这是基于当前使用的数据。从来没有接触过数据仓库的用户不会习惯从历史的角度分析问题。有一句格言，你不能想象你不曾有的东西。在绝大多数情况下，当用户说从来不的时候，实际上是说很少（rarely）。请确认你的用户明确的知道物理删除是永久的删除记录。
- 查看（See）。当用户说查看，更多的是指在报表中的数据展现。因为用户通常对未加工的数据没有概念。所有的数据的发布都是通过一些发布方法来完成，例如商务智能工具或者报表，他们可以自动的过滤不需要的数据。最佳的方法是和负责数据

展现的团队确认这些需求，如果没有这样的团队，那么确保你的用户理解物理删除是从数据仓库中永久的删除记录。

一旦永久物理删除得到确认，下一步的问题就是选择什么样的策略来查找和删除不需要的事实。最简单的解决删除的方式是截断和重新加载事实表。但是截断和重新加载仅仅对较小的数据仓库适用。如果你有一个包含许多事实表的数据仓库，每个表中包含上百万或者上亿条的记录，那么不建议使用多个增量加载来截断和删除整个数据仓库。

如果在源系统中没有包含审计表来捕捉删除的记录，那么你必须在集结区中存储每次抽取的数据，然后和下一次数据加载进行比较来获得任意遗失的数据。

如果源系统中包含审计表，那么你将是幸运的。如果删除或者修改数据非常重要，或者需要在将来进行跟踪，那么交易系统中常常有审计表。如果在源系统中没有审计表，另外发现删除事实的方法就是比较源数据和集结区中包含最近一次加载数据的数据表，这意味着每天（或者每个 ETL 间隔），你必须在集结区中保留抽取数据的拷贝。

在 ETL 处理中，在上一次的抽取和本次抽取加载到准备区后，在两张表中执行 SQL MINUS。

```
Insert into deleted_rows nologging

select * from prior_extract

MINUS

select * from current_extract
```

MINUS 查询的结果为那些在源系统中被删除但是已经加载到数据仓库中的记录。在处理完成后，你可以删除 prior_extract 表，将 current_extract 表命名为 prior_extract，最后创建新的 current_extract 表。

逻辑删除事实

当物理删除被禁止或者需要被分析，你可以逻辑删除记录，但是物理上保留。逻辑删除需要利用一个称为 Deleted 的额外的字段，通常为 Bit 或者 Boolean 数据类型，作为事实表中的标记字段表示该字段是否已删除。需要特别注意对于逻辑删除方法，需要在每个查询中都使用限定来过滤到那些已经被逻辑删除的记录。

无事实的事实表

每个事实表的粒度是一个事件量测。在某些情况下，事件可以发生，但是没有具体的测量值。例如一个事实表用来记录交通事故事件。每个事件的发生是无可质疑的，维度设计是强制性且非常直接的。如图 6.7 所示。但是当维度装载后却有可能不存在事实。就像本例一样，事件跟踪经常会产生无事实（factless）的情况。

实际上，图 6.7 中的设计有一些非常有趣的特性。复杂的事故可能包含多个当事人，原告方和证人。他们通过关联表将当事人组、原告组和证人组连接起来。这种设计可以表示从最简单的单方交通事故到复杂的多车连环相撞事故。在这个例子中，随着时间的发展，当事故发生的时候，事故当事人、原告和证人被加入组中。

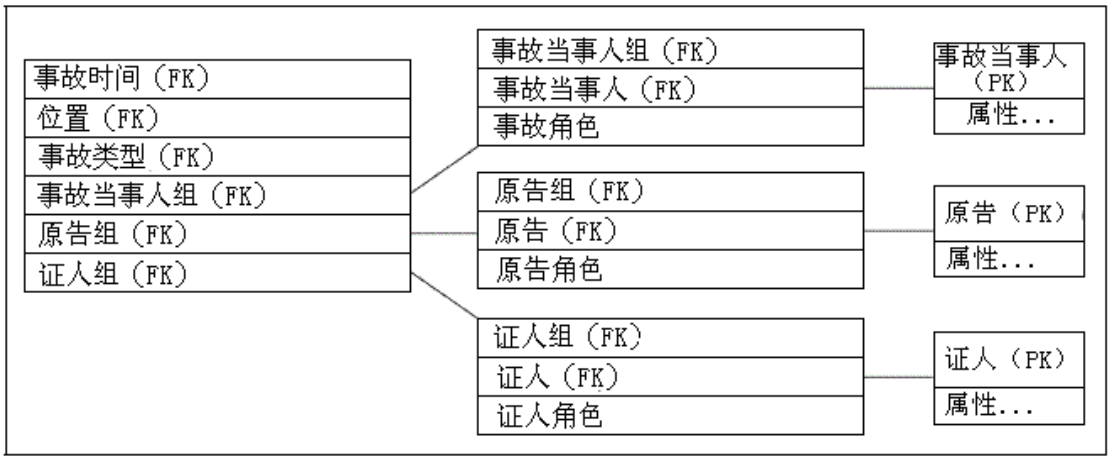

图 6.7　表达车与事故事件的无事实事实表

这种应用的 ETL 逻辑是用于判断新增的记录表示的是一个新的事故还是已有的。一个事故的主自然键需要在首次事故报告的时候被分配。通过事故当事人、原告和证人的重复记录可以用于发现理赔欺诈。

关于无事实事实表的另外的例子是覆盖，经典的例子是某天在某个商店促销的产品列表。这个表有四个外键，但是没有事实，如图 6.8 所示。这个表用于与典型的销售表关联，用于回答促销中销售了哪些产品？ 什么没有发生，这种查询的问题在数据仓库工具集（第二版 251-253 页）中有专门的介绍。由于每个商店的价格系统中有特价的记录，那么对于商店建立促销产品的 ETL 数据是很容易的。但是对于其他的促销活动，如特殊的展览或者媒体广告则需要相应的反馈数据，这些数据并非来自价格系统。利用展览在零售业的数据源处理上是个难题，因为通常在制造企业中这些数据源表示为进行展览的成本，最终，需要公正的第三方走访每个商店来获得准确的数据。

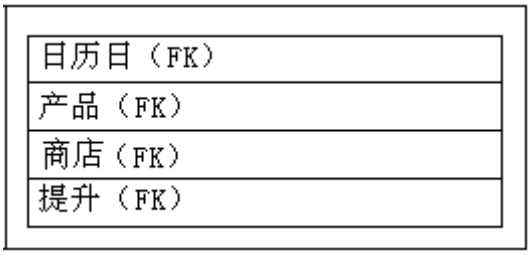

图 6.8　无事实的覆盖表

用类型 2 历史数据补充类型 1 事实表

一些环境主要使用类型 1 事实表，例如，一个完整的客户购买历史通常通过类型 1 的客户维度来显示客户最新的档案。在单纯的类型 1 的环境中，客户的历史描述不可用。在这种情况下，客户维表和完全的类型 2 的维表相比较小而且较简单，在类型 1 的维度中，自然键和维度的主键有 1 对 1 的关系。

但是在很多类型 1 的环境中，对于客户历史的访问需要进行分析。这种情况下可以采用 3 种方法：

1、在旁边同时维护完整类型的类型 2 的维度。这样做的好处就是可以维护主要的类型 1 维度的干净和简单。通过查询类型 2 的维表来查找某个时段内有效的旧的用户档案，使用同样的时间范围限定来访问事实表。这种方法对于那些显示即时性动作的事实表能够很好工

作，（immediate actions），即时性动作如零售销售，这样的事实表中客户就是量测事件。但是有些事实表记录了延时动作，如有争议消费发生后几个月后发生的退款。在这个案例中，客户档案所处的时间段没有和事实表的事件逻辑上交迭。另外的案例：当产品在到期日售出，并在一个月后产品资料已经改变后发生退货也会造成时间同步上的问题。这是另外一个延时动作事实表的例子。如果你的事实表表示的是延时的动作，那么不能采用这种方法。

2、按照完整的类型 2 维度创建主维度。缺点是，和类型 1 的维度相比更大更复杂。通过在所有查询中使用嵌入对维度的 SELECT 语句仅仅能获取当前客户维度记录的自然键，可以模拟类型 1 维度的效果；用这些自然键获取所有的用于最后和事实表关联的历史维度记录。

3、对于主要的维度使用完整的类型 2 的维表，同时在事实表代理键旁边存储对应的自然键。缺点是维表比类型 1 的维表大，且更复杂。但是如果最终用户应用关心的最新的客户记录，可以用自然键来和事实表进行关联，从而获得整个历史。这消除了方法 2 中的嵌入式查询。

优化更正

维度模型的一个最重要的优点是可以在不影响最终用户查询或者应用的前提下，对最终发布的框架进行一系列大的修改。我们把这称之为优化更正。这是维度模型世界和规范化模型世界最根本的区别，在后者这些修改将导致应用停止工作，因为物理框架被改变了。

对于维度框架有四种类型的优化更改：

1、在已经存在的事实表中增加同一粒度的事实

2、在已经存在的事实表中增加同一粒度的维度

3、在已经存在的维度中增加属性

4、增加已经存在的事实表和维表的粒度大小

图 6.9 图示了这四种类型的更改。前三种需要 DBA 对事实表和维表执行 ALTER TABLE 操作。应该优先考虑对存在的表的 ALTER TABLE 操作，而不是删除、重新定义事实表或者维表，最后再加载数据。

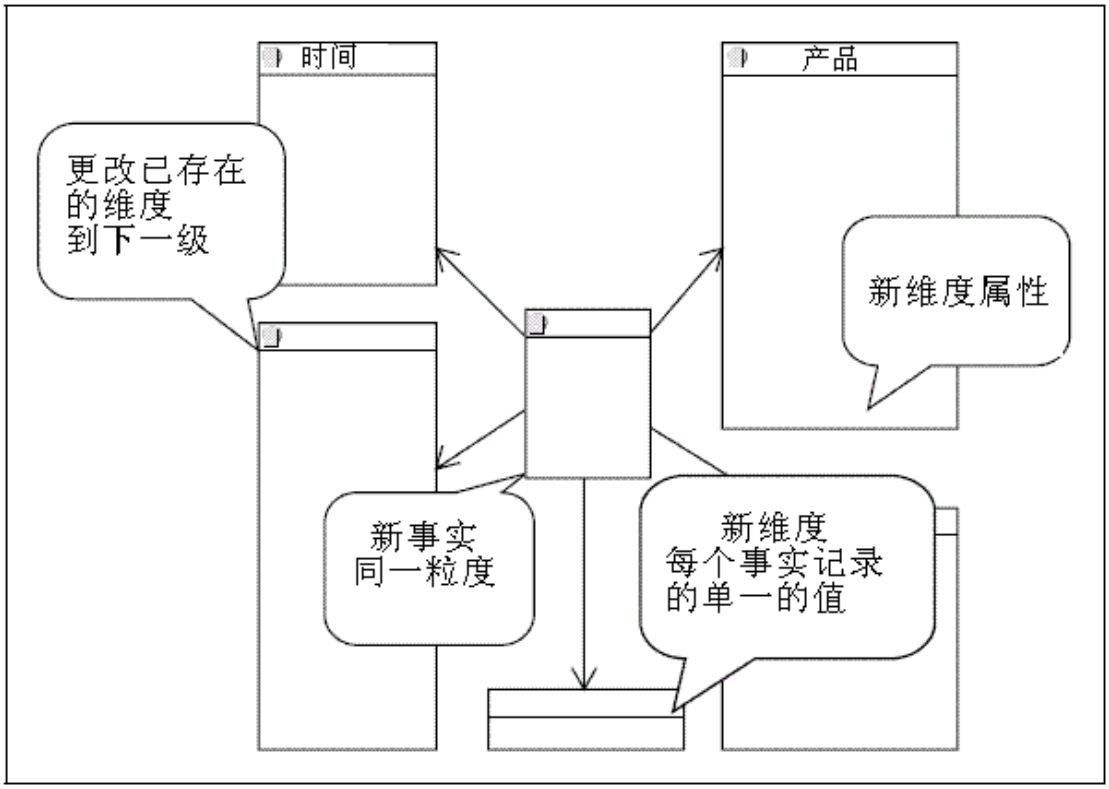

图 6.9　四种类型的优化更改

前三类更改带来一个问题，如何处理那些在更改事实、维度或者属性之前的旧的历史表。显然，如果有对应的旧的历史数据可用是最简单的。但大多数情况下，在维度模型中增加事实、维度或者属性的原因是他们在当前变成可用的。当这些改变仅仅是当前以及以前时候，我们可以按照下列的方式处理：

1、增加了一个事实。新增的事实的历史值存储为 null，当时间跨越事实增加点的时候，Null 值可以被正确的计算。计数和求平均的结果是正确的。

2、增加了一个维度。对于之前事实表中的记录，新增加的维度的外键必须指向维表中的非应用记录。

3、增加了一个维度属性。在类型 1 的维度中，不需要做任何的修改，新的实体在所有的维表记录中可用。在类型 2 的维度中，在增加实体之前的所有的记录对应的实体为 null。当记录的时间跨度包含增加实体时间点的时候，会有一点麻烦，但是也有办法将实体增加到这些记录。

第四种类型的优化优化更改 – 增加维度模型的粒度和前三种相比则要复杂的多。假设我们在记录如图 6.1 中展示的单独的零售销售记录。假设我们已经选择使用店铺维度来汇总地区销售，而不是使用现金收银机维度。在两种情况下，事实表的记录数是完全一样的，原因是事实表的粒度是单个的零售销售（在销售小票上的某行记录）。唯一的区别在于在基础粒度上使用现金收银机的视图还是店铺视图作为地区维度。但是由于现金收银机和它所属的店铺之间有确定的多对一关系，因此店铺实体是两种方式间共同的选择。如果这个维度称为地区，当在从店铺-地区改变成现金收银机-地区视角的时候，原来应用中的 SQL 不需要修改。

在不改变应用的前提下也有可能增加事实表的粒度。例如，周的数据可以改为日粒度数据。日期维度可以从周变为日，原来所用基于周的限定和分组规则仍然有效。

事实表中多个度量单位

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*

某些情况下，价值链中包含了在系统生产过程中的多个业务过程监控，或者在不同的检查点使用不同的度量，这样会导致数量表达的不一致。每个人都同意这一点，虽然数字是正确的，但是价值链中不同的人会希望按照不同的单位来查看。例如，制造业主管希望按照车皮或者集装箱来考察整个生产流程，而店铺主管需要按照运输包装、零售包装、销售单位或者购买单位（弹片的口香糖）查看数据。类似的，相同数量的产品可能有几种不同的计价，可能需要按照供货价格、列表价格、初始报价、最终价格来显示。这种需求，最终会导致对于每一条事实记录可能有多个基础数量。

考虑一下下面的情况，我们有 10 个基础的事实数量，5 种单位，4 种作价方式。一种错误的做法是事实表中建立 13 个数量事实，然后由用户或者应用开发者在维表中查找正确的转换因子，尤其是当用户利用事实表，不使用关联得到的不同时间点，然后用时间去查询产品维表的时候。另外将所有的事实组合按照不同的单位存储在主事实表中也是一种错误的做法，那样每条记录需要 10*5+10*4=90 个事实列！正确的折衷办法是建立有 10 个数量事实、4 个单位转换因子和 5 个作价因子的底层事实表。这里我们仅仅需要 4 个转换因子，而不是 5 个，因为基础事实已经按照其中的一种单位表示了，其他更大或者更小的单位仅仅需要通过乘法或者除法就可以得到。我们的物理设计现在为 10+4+4=18 个事实，如图 6.10。

当需要增加一个新的产品记录来反映这些因子(尤其是成本和价格)的微小变化的时候，在事实表中如此的封装事实可以降低对产品维度的压力。从变化的角度来看，这些因子更像事实，而非维度实体。

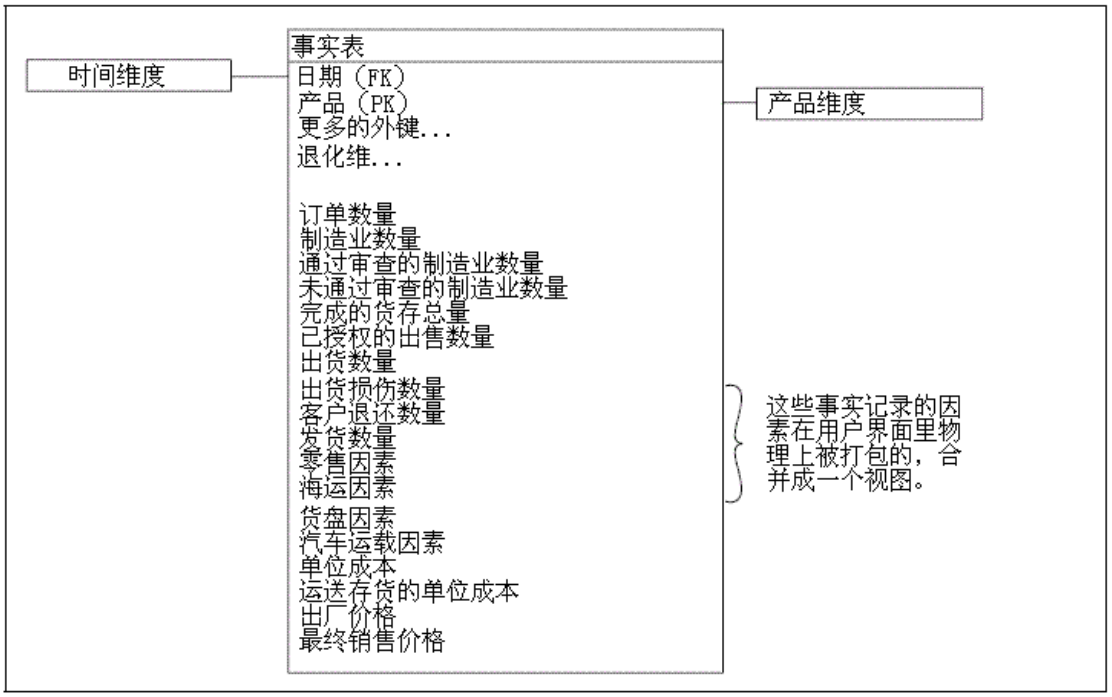

图 6.10　包含 10 个事实、5 种单位、4 个定价模式的物理事实表设计

我们现在可以将这个事实表通过一个或者多个视图发布给用户。最复杂的视图可能包含

了所有的 90 个组合，但是显然我们可以根据不同用户群所关注的单位和计价方式来创建更为简单的用户界面。

在多币种中收集收入

国际业务中经常使用不同的币种预定交易、搜集收入和付款。一个适用大多数情况的简单设计如图 6.11。销售交易中的主要金额使用本币。在某种意义上说，这就是交易的恰当 correct 值。为了更容易地在报表中使用，在交易事实记录中用第二个字段表示等值的某一单一标准货币金额，例如欧元。事实表中的两种金额的等值性规则涉及设计，可能的方式是利用本币对全球货币的日汇率。现在通过货币维表将币种限制为某一种货币类型，就可以很容易的将事实表中交易金额按照指定币种加在一起，世界范围内的交易可以使用全球货币字段加起来。注意，事实表中包含着一个独立于表示店铺位置的地理维度的币种维度。币种和国家有很强的关联性，但是又不是一对一的，一些国家在严重通货膨胀期内可能更改他们的货币标识。另外，欧盟的成员国必须能够正确的按照欧元和本国货币表示历史交易（在 2002 年 1 月 1 日之前）。

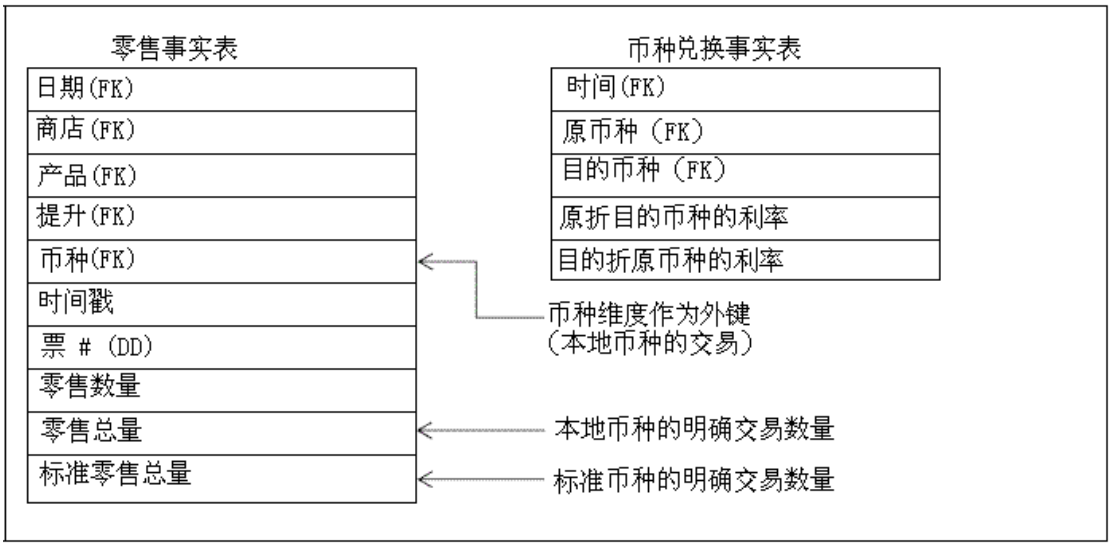

图 6.11　处理多币种的结构设计

如果你希望用第三方货币或者使用不同的时间，如期末的汇率来表示一组交易会产生什么问题呢？这时候，需要一张汇率表，如图 6.11 所示。汇率表中一般包括每天的本币和一种或多种全球货币之间的兑换率，因此，如果有 100 种本币和 3 种全球货币，那么每天需要有 600 条记录。那么对于 100 种币种，将有 10,000 个日交换率，理论上可以建立这样的汇率表，但是却不实际。因为，在现实中并非每对货币之间的汇率都存在。

迟到的事实

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*

在客户购买情境中，假设我们收到一个几个月前的购买记录，在决大多数操作型数据仓库中，我们希望将这个迟到的记录插入正确的时间位置，包括改变前面月份的销售汇总。但是我们必须为这条购买记录仔细地选择当时的维表记录。如果我们已经在类型 2 的缓慢变化维中为记录加了时间戳。处理过程中包括如下的步骤：

1、在每个维度中，找出购买发生时的对应的维表记录

2、使用上一步中的维表记录对应的代理键替换迟到的记录中的自然键

3、将迟到的事实记录插入相应的数据库物理分区，该分区中包含了其他的同期的事实记录。

这里有一些细节需要注意。首先我们假设维表记录中包含了两个时间戳，标识详细描述的生效和失效日期。这可以使查找正确的维表记录变得简单。

其次要注意的是我们假设有一个操作性数据仓库，并希望将迟到的记录插入到历史月份中。如果你使用的是书中所带的数据仓库，那么你不能修改历史的月度销售汇总值，即使这个值是错误的。有一种情况需要你仔细考虑，销售记录中的日期维度为订购日期，可能是今天，但是其他的客户、店铺和产品维度仍然指向了旧的描述。如果你遇到这种情况，你需要和财务部门仔细的讨论，确保他们理解你在做什么。一个折衷方式是在购买记录中使用两组日期维度，一组代表实际购买日期，另外一组代表订购日期。现在你可以既从订购又从操作的角度汇总销售。

第三个细节是如果要求迟到的购买记录所插入的数据库的正确的物理分区中包含了和该记录同代的兄弟节点。当你将一个物理分区从一种存贮形式转移到另外的存贮形式，或者执行了备份和恢复操作，将影响到一段时间内的所有的购买记录。绝大多数情况下，这是我们想做的。如果你基于日期维度定义物理分区，那么你可以确保在一个时间段发生的所有的事实记录在相同的物理分区。由于我们需要对日期维度使用代理键，这是需要代理键按照逻辑顺序分配的一种情况之一。

聚合

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*

提高大型数据仓库性能最重要的手段是在基础记录上建立一组适当的聚合记录。聚合可以显著的提高性能，一些情况下，查询的性能可以提高 100 倍，甚至 1000 倍。没有其他的方法可以获得如此的性能提升。作为数据仓库的 IT 主管，为了解决性能问题，在追加新的硬件投资之前，应该首先通过聚合来充分利用性能潜力。创建复杂的聚合所带来优势可以在大多数数据仓库软硬件平台上实现，包括所有主流的关系型的 DBMS，如 Oracle，Red Brick，Informix，Sybase 和 DB2，以及单 CPU, SMP 和 MPP 并行处理器架构。本节介绍的是如何构建数据仓库，最大化聚合的优势，在不需要复杂的元数据的情况下，如何创建和使用这些聚合表。

聚合导航是数据仓库文献中讨论的标准主题。图 6.12 利用简单的维度模型阐释了这个主题。

要点如下：

- 在适当设计的数据仓库环境中，需要建立多组聚合表，在数据仓库的关键维度中体现为常用的聚合级别。聚合导航的定义和支持仅仅针对维度型数据仓库。在一个完全规范化的环境中没有聚合导航。

- 聚合导航器是中间件的一部分，位于发起请求的客户端和 DBMS 之间，参见图 6.13
- 聚合导航器用于解释客户端的 SQL，在任何需要的时候，将访问基础粒度数据的 SQL 转化为利用聚合表的 SQL
- 聚合导航器使用特殊的元数据将访问基础粒度数据的 SQL 转化为利用聚合表的 SQL，该元数据描述了数据仓库聚合方式。

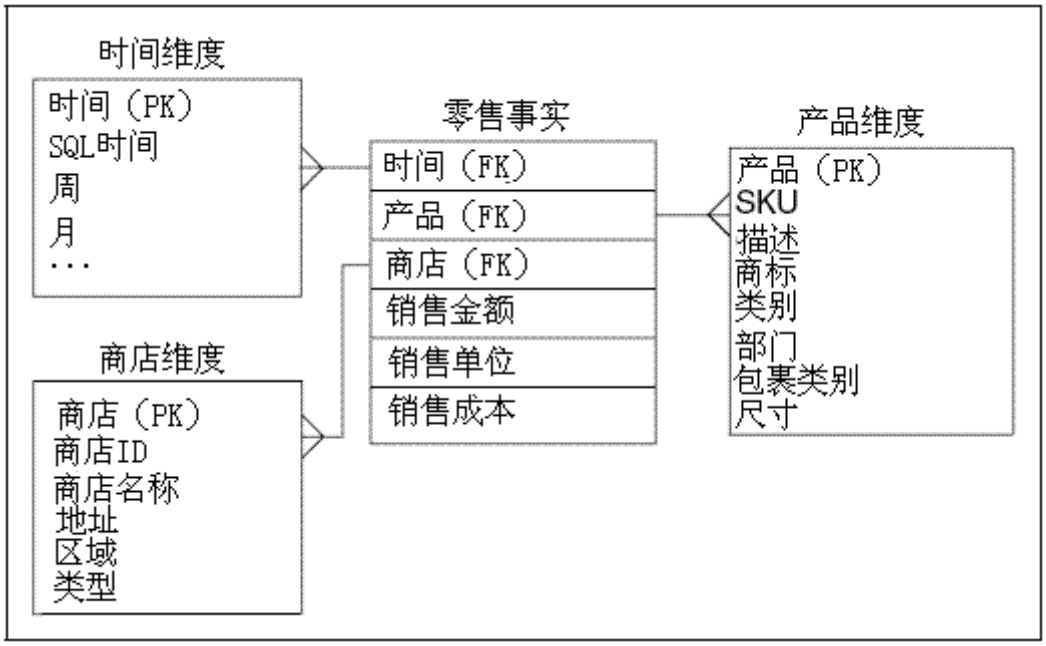

图 6.12　简单的日、产品、商店粒度的简单维度模型

在大型数据仓库中建立聚合的目的不仅仅是为了提升性能，一个好的聚合应该：

1、为尽可能多类型的用户查询提供较好的性能

2、在仓库中仅仅增加合理数量的额外存储。所谓合理是从 DBA 的观点出发，但是大多数的数据仓库 DBA 按照小于等于 2 的因子来控制增加的磁盘存储。

3、除了明显的性能提升，性能对于最终用户和应用开发者是透明的。换句话说，没有任何最终用户应用的 SQL 会直接引用聚合表！不管用户使用的是什么查询工具，都可以利用聚合的优势。

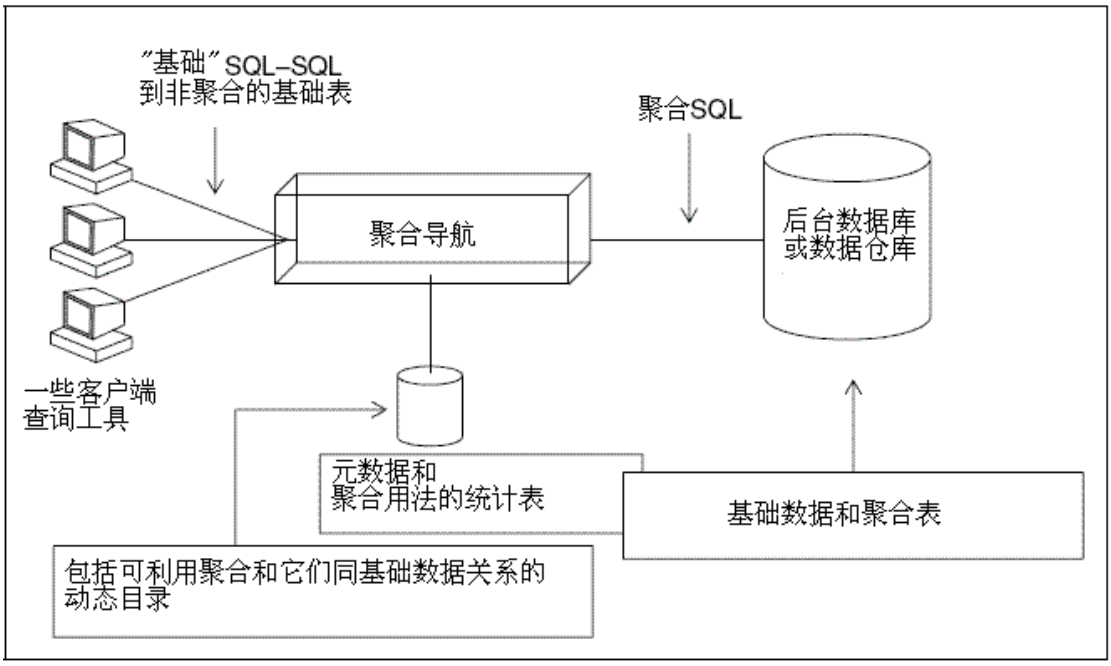

图 6.13　聚合导航架构

4、尽可能降低数据抽取系统的成本。当每次数据加载的时候，多数聚合表不可避免地会被重建，但是应该做到尽可能的自动化

5、尽可能的降低 DBA 管理的责任。通常支持聚合的元数据应该数量有限且易于维护。大多数的元数据应该通过监控用户查询自动的创建，并给出创建新聚合表的建议。

一个仔细设计的聚合环境可以达成我们的所有目标。但是未经过仔细设计，可能彻底的失败。下面是一系列设计需求，如果都考虑到了，那么就可以达成我们的预期。

设计需求#1

聚合信息必须存储在他们自己的表中，和基础粒度的数据分离。每个不同的聚合级别必须使用他们自己独自的事实表。

将聚合信息的独立存储在他们自己的事实表中。这一点非常的重要，并且带来一系列的好处。首先，本节描述的聚合导航模型在聚合信息使用独立的表存储的时候非常的简单，因为聚合导航器可以从 DBMS 的普通系统目录中获得绝大多数的信息，而不需要额外的元数据。其次，由于聚合信息存储在单独的表中，最终用户对于可加性事实不会产生重复计数，因为 SQL 访问的每个给定的事实表中的数据粒度都是一致的。第三，仅仅有少量的大额的条目。例如，整年的国家级的总销售额不需要硬塞进基础的事实表。通常，这些少量的大额记录迫使数据库设计者增加相应字段的宽度，浪费了磁盘空间。因为基础表非常的巨大，通常占到整个数据库的一半，通常需要尽可能的减少字段宽度。第四，当聚合表使用单独的表时，聚合的管理更加的模块化、分区化。不同的聚合表可以在不同的时间创建，利用聚合导航器，单独的聚合表可以在不影响其它数据的情况下离线和重新上线。

设计需求#2

无论什么情况，聚合表所使用的维表应该是基础事实表所使用维度的缩小版本。

注意：绝大多数维度的缩小版本其实是某个维度的整个删除。

换句话说，假设图 6.12 中所示的基础事实表上我们要建立类别（Category）级别的聚合，在产品维度上，将单独的产品聚合到类别级别，参见图 6.14。这里我们叫做 one-way 单行道类别聚合模式。

请注意，在这个案例中我们要求在时间维度或者店铺维度上进行聚合。在图 6.14 的表中标示了每天、每个店铺对某类型的产品类别销售了多少。我们的设计需求是告诉我们原始的产品表必须使用一个缩小的产品表进行替代，我们称之为类别（category）。对这个缩小的产品表的简单的看法就是将其考虑为仅仅包含从单个产品聚合到类别水平的字段。仅仅有少量的字段仍然保持唯一定义。

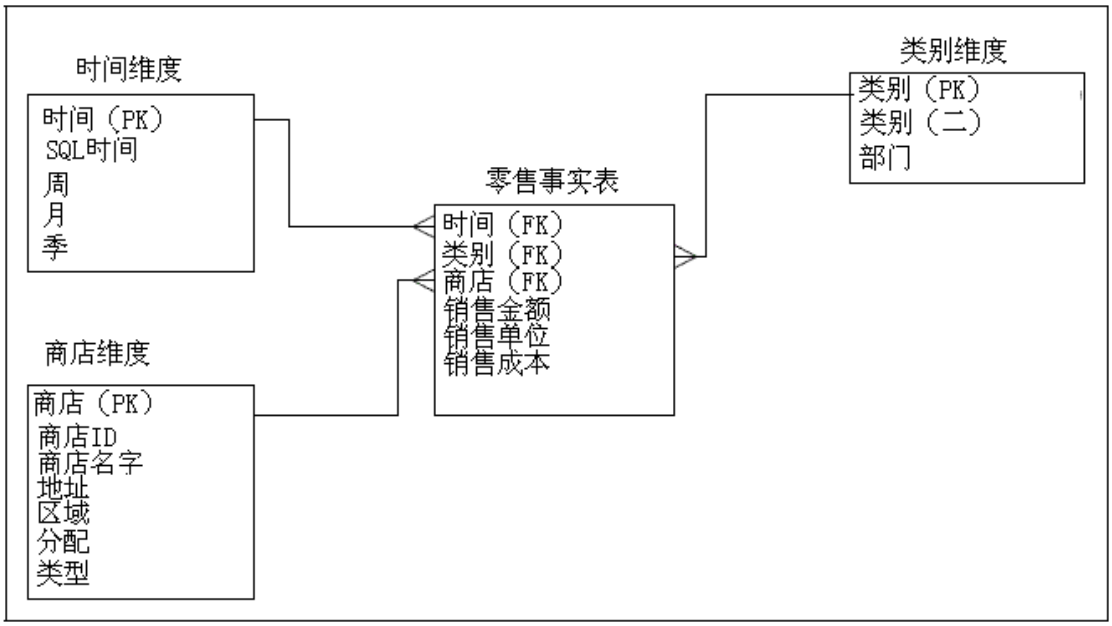

图 6.14　单行道类别聚合模式

例如，类别描述和部门描述将在类别级别进行定义，而且他们的字段名称应该和在基础表中使用的维表中命名一致。但是单独的 UPC 号码，包装尺寸以及滋味（flavor）将不在这个级别出现，而且不能出现在类别表中。

缩小的维表对于聚合导航来说非常的重要，因为任何聚合级别的范围可以通过查找缩小维表的系统目录描述就可以得到。换句话说，当我们查找系统目录表，我们可以发现他的目录描述和部门描述。如果查询的是产品滋味，那么我们马上知道聚合级不能满足要求，那么聚合导航器必须查找其他地方。

缩小的维表吸引人还在于避免了在原始的维表中为所有那些在更高聚合级别上的所不支持的维度实体填写 null 值。换句话说，由于我们在类别表中没有滋味（flavor）和包装尺寸（package size）字段，我们不需要为这些字段填写 null 值，同样在应用中也无需为这些 null 值增加判断。

尽管我们将目光放在了缩小的维表上，但是在建立高级别的聚合的时候，事实表中的度量的数目也会发生变化。大多数的基本的可加度量如销售额、销售数量、成本额将在各个聚合级别上存在，但是一些维度，如促销和一些事实如促销成本将仅仅在基础级别存在，在聚合表中需要被删除。

简化的设计需求#2 仅仅创建这样的聚合事实表，特定的维度被完整的删除，而不是简单的缩小。例如，在零售销售事实表中，地址或者店铺维度可以被删除，从而创建

一个基于国家的总销售额事实表。如果使用这种方法，其优点是建立聚合表的时候不会影响那些删除的维度。因此，你可以修改你的地理区域的定义，在我们例子中的表却不需要改变，尽管部分删减的聚合到地区的地理维度需要重新计算。这个方法不是万能的；在我们的例子中只有请求国家销售额的查询才可以使用前面提到的聚合表。

设计需求#3

基础的事实表以及所有相关的聚合表可以作为一个框架组（a family of schemas）关联在一起，以便聚合导航理解表之间的相互关系。在这个框架组中，任何单一框架由一个事实表和他所相关的维表构成。对于未聚合的数据仅仅有一个基础框架，但是会有一个或者多个聚合模型来表现计算汇总数据。图 6.12 是一个基础框架，图 6.14 是我们的模型中众多可能的聚合模型中的一个。

事实表家族，以及他关联到的所有完全的和缩小的维表的信息是这个设计中唯一需要的的元数据。

设计需求#4

强制任何最终用户或者应用者的 SQL 仅仅使用基础的事实表和关联的完全的维表。

所有的用户界面和所有最终用户应用中都有这样的设计需求。当用户检查数据库的图形视图的时候，他们看到的应该仅仅是和图 6.12 对等的图形。用户不会关心是否有聚合表的存在！同样的，所有报表生成工具或者其他复杂应用中的手写 SQL 应该仅仅参照基础事实表及其关联的完全的维表。

管理聚合，包括物化视图

依赖于不同的 DBMS 和前端工具，聚合有许多不同的物理变种。从上一节中我们讨论的设计需求中，我们发现聚合导航器的正确架构应该是位于 DBMS 之前的中间件，解释所有的 SQL 并检查所有可能的重定向。错误的架构是将聚合导航模型嵌入在一个前端工具中，这样聚合导航的优势不能为所有的 SQL 客户端使用。

在写本书的时候，有两种基本方法可以实现聚合导航。一种是支持前面章节中介绍的直接的缩小的事实表和维表的变种。另外的一种是通过某种短暂的查询动态的创建这些表，物化（materiallized）为实际的数据写入磁盘，后续的查询可以不需要重新计算查询就可以访问这些数据。Oracle 的物化视图就是第二种方法的一个例子。

无论是 explicit 直接缩小表方法，还是物化视图方法都要求 DBA 了解更新底层事实表后对聚合的影响。在刚刚更新完底层事实表的那个时点，聚合表是无效的，并且不能被使用。在大多数的环境中，实际上需要事实表在所有的聚合被重新计算之前，就立即发布给用户使用。

常规的按日追加事实表将会允许通过将数据增加到多个聚合 buckets 来进行聚合更新。但是维表内容或者维度逻辑发生显著变化的时候，聚合要被删除重建。例如，如果聚合依赖的事实的历史维度数据发生类型 1 的更改的时候需要聚合重新地被计算。反之亦然！如果变化的实体不是聚合的目标，那么聚合结果可以不做修改。例如，可以从滋味（flavor）实体和一个大的产品文件进行重新映射，但是类别聚合将不受影响。可视化这些依赖关系是管理

聚合的关键技能。

注意，只要迅速地执行改变，并且不包括迟到的数据，维表的类型 2 更新通常不需要任何聚合结果重建。类型 2 更新不会影响已经存在的聚合；当他们记录后，就是正确的。

通常，在均衡增加的管理代价和性能提升后，一个聚合事实表的大小至少或应该是原始事实表的十分之一。粗略的讲，聚合表带来的性能提示与数据压缩因子成正比。换句话说。一个比基础表小十倍的聚合表意味着十倍的性能提升。

如果整个聚合表仅仅占到整个事实表存储的 1%，那么意味着没有建立足够的聚合。整个的聚合存储占到 100%（意味着存储量翻倍）是合理的。

依靠传统的磁盘存储，同单处理器系统一样，大的硬件平台、并行的处理架构从聚合中获得性能提升是一样的，原因是性能提升完全依赖于减少总的 I/O。但是，硬件系统的销售和系统工程师否定了这一点，因为他们的任务就是销售更多的硬件，而不是通过聪明的数据结构提升性能。请当心这一点！

提交维度数据到 OLAP 立方体

基于服务器的 OLAP（联机分析过程）产品是数据仓库架构中的一种流行组件。OLAP 服务器提供两种主要的功能：

- 查询性能。使用聚合和特殊的索引和存储结构。OLAP 服务器自动的管理聚合和索引，回顾一下前面章节对聚合管理的讨论就可以明白他的价值。
- 丰富的分析。使用和 SQL 不同的用于复杂分析的语言。OLAP 服务器还提供存储复杂的计算以及安全设定，某些服务器还可以和数据挖掘技术集成。

任何情况下，OLAP 立方体的最佳数据源是存储在 RDBMS 中的维度型数据仓库。维度型数据仓库的语言：维度、主键、层系、实体和事实可以直接翻译为 OLAP 世界的术语。OLAP 引擎开发的主要目标是支持对维度模型的快速和复杂的查询。OLAP 引擎和关系型数据库和 ETL 工具不同，他不是用于清理数据或者确保参照完整性。尽管你的 OLAP 技术可以提供直接基于交易数据源创建立方体的功能，但是建议你还是尽量少用，你应该设计一个关系型数据仓库，使用本书描述的技术实现它。由这个关系型存储向你的立方体提供数据。

立方体数据源

基于服务器的不同的 OLAP 产品和版本有着不同的功能。一个显著的不同点在于进入立方体数据的所要求的最近的数据源。一些产品需要数据存储在文本文件中；另外的需要是源来自某一品牌的数据库；还有的可以从几乎所有的数据源加载数据。

如果你的立方体的数据源来自普通文件，你的 ETL 过程的最后一步显然是将正确的数据集写出。尽管从普通文件获取数据不够流行，并且相比从关系型数据库中数据获取速度慢，但是没有特殊的性能问题：所有的关系型数据库和 ETL 工具都可以高效的将查询的结果写入文件。

在关系型数据仓库中分析立方体处理的任何查询。确保这些查询是充分优化的。如果需要，增加索引和物化视图来提高这些查询的性能。

处理维

和在使用事实表之前需要处理关系维表一样，OLAP 维度需要在处理事实之前进行处理。根据你使用的 OLAP 工具的不同，你可以在一个事务中处理维度和事实，这样当错误发生的时候就可以进行回滚。这在理论上非常地吸引人，但是对于大型的 OLAP 数据库却不可行。很多大型的 OLAP 系统逐个的处理维度，通常是 ETL 模块处理相关关系型维表过程的最后一步。

ETL 系统设计需要注意一些 OLAP 维度的特点。首先，注意 OLAP 系统设计目标是易于使用并且对于维度层系或者层系间向上或者向下（例如从产品到品牌，再到类别）查询有很好的查询性能。在传统的关系型数据仓库中，你需要保证层系的级别间的参照完整性（例如，产品属于且仅属于某一个品牌）。但是在 OLAP 工具中，你仅仅需要保证参照完整性。层系的不同级别间的参照完整性在 OLAP 服务器中保证。如果发现不一致，维度处理或者失败，或者 OLAP 服务器可以按照假设的处理方式进行处理。如果你不希望两者发生，在 OLAP 处理之前全面的清理你的数据。

维数据的变化

OLAP 服务器处理不同类型的维度数据变化的差异很大。大多数 OLAP 服务器处理新的维表行记录和关系型数据模型优化 graceful 处理一样，例如增加了新的客户。更新那些健壮 Strong 层系之外的实体的值也是优化 graceful 的，因为没有基于该实体创建持久的聚合。对于组成 OLAP 维度层析的实体的修改需要 ETL 设计者认真的考虑。

让我们首先给一个案例，对于类型 2 的缓慢变化维度，即，每一个变化的成员增加了一行新的记录和代理键，可以被 OLAP 服务器 gracefully 优化处理。从 OLAP 服务器的观点，这就是一个新的客户，和完全的新客户没有任何区别。

对 Strong 层系实体的类型 1 的缓慢变化更新对于 OLAP 服务器的来说很难处理。OLAP Server 遇到的问题和关系型数据仓库使用该维度实体建立聚合表后遇到的问题一样。当维度层系重建以后，所有存在的聚合都不在可用。OLAP 服务器处理的方式很多，从不加提示的修改，到简单的无效化聚合然后利用后台处理重新创建，再到无效化维度和使用到该维度的所有的立方体，强制所有的 OLAP 数据库重建。你应该和你的 OLAP 服务器厂商核实他们对这种问题的处理方式。厂商不断在每个版本中增强这个功能，所以使用最新的版本或者核实当前版本的功能非常重要。

应该在 OLAP 维度处理之前核实数据源没有任何改变，否则建好的 OLAP 数据库将会不可用或者不一致。根据成本和系统使用方式，你可以选择如下的设计：

- 让 OLAP 和关系型数据仓库数据库分离：将 OLAP 处理推迟到方便的时候处理（例如下个周末）。
- 通过中止关系操作保持 OLAP 立方体和关系型数据仓库的同步（通常在准备区中汇聚变化）
- 通过每晚上的执行高昂代价的重建来同步 OLAP 立方体和关系型数据仓库。如果 OLAP 立方体是一个镜像或者在重建过程中依然对业务人员可用那么这就非常可行。

任何情况下都需要在 ETL 错误事件表中加载特殊的事件，或者通过 email 或者寻呼机通知操作员。

处理事实

很多人认为立方体中只能存储聚合数据，这同认为数据仓库中仅存储聚合数据一样，是一种过时的看法。基于服务器的 OLAP 产品能够管理很大数据量的数据，越来越多的用于存储关系型数据仓库相同粒度的数据。这个区别对于设计基于立方体的 ETL 系统非常重要。

大多数基于服务器的 OLAP 产品支持一定程度的增量处理；其他的仅仅支持完全更新。完整的立方体更新适用于汇总级立方体和少量的细节数据。要在大数据量的细节数据上获得好的处理性能，对事实数据使用增量处理就显得异常的重要。OLAP 技术中有两种类型的立方体增量处理方法：基于分区，增量事实。

加载到分区是一种吸引人的加载立方体数据子集的方法。如果你的 OLAP 技术支持分区，你的立方体是基于时间分区，通常按照周或者月，这样可以很容易的加载对应的时段的数据。如果每天处理基于周的分区，周一的数据将被删除重新加载多次，直到周日加载后结束分区。当然，这个技术并没有最大化加载效率，但是对于许多应用却是可以接受的。通常将 OLAP 立方体分区的周期设计为同关系型分区一致，将管理关系型分区的脚本扩展为同时管理 OLAP 分区。

对 OLAP 立方体分区可以同时大幅度提高查询和处理的性能。对基于一个或者多个维度进行分区后的立方体进行查询，只访问查询涉及的立方体，而不是对整个立方体访问。使用分区后，处理立方体可以采用前面章节介绍的伪增量更新，也可以支持多个分区处理作业并行的执行。如果你的立方体使用了复杂的分区设计，你的 ETL 系统应该支持并行的多个分区处理作业。如果你的 OLAP 服务器不能替你管理并行处理，那么需要将他纳入 ETL 系统设计，这样你可以配置同时并行的分区数。这时参数需要在系统测试的时候进行调优。

一些 OLAP 服务器支持真正的增量事实处理。可以采用某种方式标识出哪些是新的数据（通常使用处理日期），OLAP 服务器将他们追加到立方体或者立方体的分区。如果有迟到的事实，那么增量处理将比重新处理当前分区更好。

增量或者完全处理的替代方式是 OLAP 引擎监控源数据库的新的事务，自动将新的数据加载到立方体。这个处理很复杂，需要 OLAP 引擎和关系型引擎紧密集成。在写本书的时候，这种功能已经有一些初步的例子可以提供，但是更多的功能性的特性尚在开发中。

常见错误和难题

在处理事实过程中一个最常见的问题是参照完整性失败。一个正在处理的事实行没有对应的维度成员，如果你采纳本书或者其他工具数据中的建议并且使用代理键，在一般处理中就不会遇到事实参照完整性错误问题。即使如此，你也应该了解当特殊错误发生的时候，OLAP 服务器是如何处理的。

OLAP 服务器并不能够自动的定位那些更新的事实数据。在关系型数据库端建议采用的设计方式是使用会计簿式（ledgered）事实表，该表中有字段使用正或负的交易金额记录事实的变化。会计簿式设计可以支持对事实改变的审计，出于这个原因，这个表一般是独立的。在会计簿式事实表上建立的 OLAP 立方体能够接受 gracefully 优化变化，但是当立方体基于时间分区以后，支持迟到事实的处理逻辑可能会变得很复杂。

相反的，在事实表上创建的 OLAP 立方体要支持替代更新的话，几乎都需要在每次更新发生后完全处理。如果立方体很小，这也许不是问题。但是当立方体很大的时候，应该考虑是否业务用户接受按照周或者月批量的更新，或者放弃将那些常更新的子主题的数据并入 OLAP 数据库。还要注意一点，OLAP 处理为了处理新的事实行，但是不会识别那些更新的事实。ETL 系统必须触发特殊的处理。

非常规的完全更新

在写本书的时候，OLAP 技术还没有和关系型技术一样可靠。我们坚信关系型维度数据仓库应该和数据仓库中的记录一样持久，而 OLAP 立方体则是短暂的。很多运作了 6 个月，1 年或者更长一点时间的公司没有必要重新创建他们的立方体。无论如何，为了完全重建 OLAP 数据库的新的安装都需要开发和测试过程。对这个限制的推论就是，不应该在 OLAP 立方体中包含那些不在关系型数据仓库中或者不容易恢复到数据仓库中的数据。其中包括了回写数据（在预算应用中非常的普遍），这些数据应该直接或者间接的写入关系型数据库。

在 OLAP 服务器还没有像他的兄弟----关系型数据库一样可靠和开放的时候，他们被看作可以考虑的次要系统。OLAP 厂商当前正在开发的版本专注于可靠性和恢复能力，所以我们希望这种二流的状态尽快结束。

集成 OLAP Processing 到 ETL 系统

如果你的数据仓库包括了 OLAP 立方体，那么他们应该和其他任何系统组件一样被专业的管理。这意味着你应该和业务用户就数据流传和系统正常时间签订服务协议。尽管我们通常希望立方体的发布和关系型数据的数据刷新保持一致，以较低的频率刷新立方体是可以被接受的。和业务用户协商并签署一个协议，并且在不可避免的错误发生的时候及时通知他们。

尽管 OLAP 服务器供应商现在还没有致力于提供一个专业的工具来维护 OLAP 数据库，但在最低限度上他们也应该提供命令行工具，以及常用的立方体管理向导。如果通过命令行启动 OLAP 处理，那么你可以将它和 ETL 系统集成，但是集成度非常的低。单一厂商提供的 ETL 和 OLAP 可以支持更深层次的集成。

很多系统可以按照时间表，通常为每周或者每月，完全的重建整个 OLAP 数据库。在这种情况下，OLAP 处理仅仅需要验证关系型数据仓库周末或者月末的处理是否成功完成。ETL 系统可以通过检查系统的元数据获得成功的状态，如果成功，那么开始立方体处理。

对于大型系统，一种通常的集成架构包括在 ETL 系统中每个维表处理分支、模块或者 package 中的最后一步增加 OLAP 维度处理。像前面所说的，在加载处理立方体的脚本或者命令之前，你应该首先测试那些可能造成立方体无效的危险的数据操作。同样的事实表处理模块或者分支应该扩展为包括 OLAP 处理。所有处理的最终步骤应该更新元数据时间和成功或者失败的状态，并将这些信息发布给业务用户。

OLAP 综述

如果 OLAP 数据库是你的数据仓库的一部分，那么就需要严格管理，ETL 团队应该精通处理的特性，以及企业 OLAP 技术的特性，并将其作为技术上的可选项。这点非常重要，尤其是按照当前的潮流将更多的，甚至所有的数据仓库的数据加载到 OLAP 立方体的时候。要真正的从最佳粒度的立方体中获益，数据仓库团队必须拥有熟悉的面向关系型的 ETL 系统，并且将其同 OLAP 处理集成。

总结

在本章中，我们把事实表定义为含有企业数字度量的命脉。所有的事实表记录都包括两个主要部分：描述度量上下文的关键字以及我们叫做事实的度量本身。然后我们描述了事实表提供者的关键角色，它把事实表发布给出来供其它人使用。

我们发现参考完整性对于维度结构是非常重要的要求，同时我们还建议了三个强制参考完整性的位置。

然后我们描述了如何为数据仓库创建代理键通道 pipeline，以便准确跟踪维度实体的历史变化。

我们描述了三种类型的事实表结构：交易粒度、周期快照粒度和聚合快照粒度。按我们的经验，这三种粒度满足了所有可能的度量条件下的模型。永远不要把不同粒度的度量放到一个事实表中，以免可能把结果复杂化。坚持这一方法可以简化应用开发，另外也可以让最终用户更易理解数据的结构。

我们还提供了一些用于处理问题的特殊技术的建议，包括灵活修改事实表和维表、度量的多单位、迟到事实数据以及创建聚合等。

我们以装载 OLAP 立方体这一特定部分作为本章的结尾，这是关系型维度结构的近亲。

在本章中，我们描述了数据仓库 ETL 系统的四个主要步骤：抽取、质量保证、规范化以及结构化数据，从而生成维度结构供最终用户使用。在接下来的第 7 章和第 8 章中，我们会涉及在创建 ETL 系统过程中最常用到的软件工具，以及如何调度运行这样复杂的系统。