The Data Warehouse ETL Toolkit

目录

前言及致谢.....1
致谢.....1
作者简介.....2
1 介绍.....3
1.1 全书概览：两条并存的主线.....5
1.2 规划与设计主线.....6
1.3 数据流主线.....8
1.4 本书如何组织.....10
1.5 谁将阅读本书.....16
1.6 总结.....16
第一部分 需求、现状和架构.....18
2 一切围绕需求.....18
2.1 需求.....18
2.2 架构.....23
2.3 数据仓库的任务.....35
2.4 ETL 小组的任务.....40
3 ETL 数据结构.....41
3.1 是否集结数据.....41
3.2 设计集结区.....43
3.3 ETL 系统中的数据结构.....47
3.4 规划和设计标准.....59
3.5 总结.....62
第二部分 数据流.....64
4 抽取.....64
4.1 第 1 部分：逻辑数据映射.....65
4.2 逻辑数据映射内部.....67
4.3 创建逻辑数据映射.....70
4.4 集成异构数据源.....80
4.5 第 2 部分：从不同平台进行抽取的挑战.....82
4.6 主机数据源.....83
4.7 平面文件.....97
4.8 XML 数据源.....100
4.9 Web 日志数据源.....104
4.10 ERP 系统数据源.....108
4.11 第 3 部分：抽取变化数据.....111
4.12 总结.....117

5 清洗和规范化.....118
5.1 定义数据质量.....120
5.2 假设.....120
5.3 第 1 部分：设计目标.....122
5.4 第 2 部份：清洗提交报告.....129
5.5 第 3 部份：过滤器及其度量.....136
5.6 第 4 部分：规范化报表.....154
5.7 小结.....165
6 提交维表.....167
6.1 维度的基础框架.....167
6.2 维度的粒度.....171
6.3 维度的基本加载计划.....171
6.4 扁平（Flat）维度和雪花（Snowflaked）维度.....172
6.5 日期与时间维.....174
6.6 大维度.....177
6.7 小维度.....178
6.8 一个维表还是多个.....180
6.9 角色维度.....181
6.10 其他维度的子维杜.....182
6.11 退化维.....184
6.12 缓慢变化维.....185
6.13 类型 1 缓慢变化维（覆盖）.....185
6.14 类型 2 缓慢变化维（分区历史）.....187
6.15 第二种缓慢变化维的精确时间戳.....191
6.16 类型 3 缓慢变化维（交替实体）.....193
6.17 混合缓慢变化维.....195
6.18 滞后到达的维度记录和更正劣质数据.....195
6.19 层次维和桥接表.....197
6.20 不规则层次和桥接表.....200
6.21 技术关键点：生成层次桥接表.....202
6.22 采用维度中的位置属性展现文本事实.....207
6.23 小结.....209
7 提交事实表.....211
7.1 事实表基本结构.....211
7.2 确保参照完整性.....213
7.3 代理键管道.....215
7.4 基础粒度.....218
7.5 准备装载事实表.....225
7.6 无事实的事实表.....233
7.7 用类型 2 历史数据补充类型 1 事实表.....234
7.8 优化更正.....235
7.9 事实表中多个度量单位.....237
7.10 在多币种中收集收入.....239
7.11 迟到的事实.....240

7.12 聚合 241
7.13 提交维度数据到 OLAP 立方体 248
7.14 总结 253
第三部分 实施与运行 255
8 开发 255
8.1 当前市场上提供的 ETL 工具包 256
8.2 当前脚本语言 258
8.3 时间是本质 259
8.4 使用数据库的块加载工具加速数据插入 277
8.5 管理数据库特性来提高性能 282
8.6 性能问题纠错 292
8.7 增长的 ETL 吞吐量 294
8.8 总结 300
9 操作 302
9.1 调度和支持 302
9.2 迁移到生产系统 315
9.3 清除历史数据 329
9.4 监控 ETL 系统 330
9.5ETL 过程调优 338
9.6ETL 系统安全 341
9.7 短期归档和恢复 343
9.8 长期归档和恢复 344
9.9 总结 347
10 元数据 348
10.1 定义元数据 348
10.2 业务元数据 357
10.3 技术元数据 360
10.4 ETL 生成的元数据 364
10.5 元数据标准和实践 374
10.6 效果分析 376
10.7 总结 376
11.职责 378
11.1 计划和领导 378
11.2 管理项目 385
11.3 总结 409
第 4 部分 实时流 ETL 系统 411
12 实时 ETL 系统 411
12.1 为什么要有实时 ETL 412
12.2 实时 ETL 的定义 413
12.3 实时数据仓库化的挑战和机遇 414
12.4 实时数据仓库的回顾 414
12.5 需求分类 419
12.6 实时 ETL 过程 426
13 结论 446

13.1 深化 ETL 的定义 446
13.2 数据仓库和 ETL 的未来 448

前言及致谢

致谢

首先我们要感谢阅读了数据仓库工具箱丛书的成千上万的读者，感谢你们对于撰写这本数据仓库 ETL 书籍的大力支持和鼓励。我们确信还会不断地从你们——数据仓库的使用者和开发者——那里学习到更多的东西。

非常感谢 Jim Stagnitto，是他鼓励 Joe 开始此书，并给了他足够的信心完成该项目。Jim 是事实上的第三作者，他在数据质量和实时 ETL 的章节中作出了主要的创造性贡献。

另外还要感谢 Jeff Coster 和 Kim M.Knyal，他们在前/后加载处理和 ETL 项目管理的讨论中做出了特殊的贡献。

我们有一个特别的读者小组，他们仔细阅读了本书手稿的第一版本，并给出了大量有用的建议。一般来说，对于已经“完成”的手稿再做修改是非常令人头疼的事，但是这种深层次的校读对于工具箱系列的书籍来说是一种传统，这次也不例外，并且非常成功。以字母顺序，这些读者包括：Wouleta Ayele, Bob Becker, Jan-Willem, Beldman, Ivan Chong, Maurice Frank, Mark Hodson, Paul Hoffman, Qi Jin, David Lyle, Michael Martin, Joy Mundy, Rostislav Portnoy, Malathi Vellanki, Padmini Ramanujan, Margy Ross, Jack Serra-Lima, 以及 Warren Thornthwaite。

我们还特别感谢 Robin Caserta（Joe 的妻子）和 Julie Kimball（Ralph 的妻子）在整个项目期间给予的支持，还有孩子们 Tori Caserta, Brian Kimball, Sara（Kimball）Smith 以及孙子 Abigail Smith（！），他们对于作者总是处于工作状态给予了充分的理解和耐心。

最后，Wiley 计算机图书小组又一次成为了本书完成的主要原因。感谢你们，Bob Elliott, Kevin Kent 和 Adaobi Obi Tulton。

作者简介

Ralph Kimball 博士，Kimball 集团的创始人，从 1982 年起就成为数据仓库领域的领导者，也是当今最著名的演说家、咨询专家、教师和作家之一。他的书籍包括<<数据仓库工具箱>>（Wiley,1996）、<<数据仓库生命周期工具箱>>（Wiley,1998）、<<Data Webhouse 工具箱>>（Wiley,2000）和<<数据仓库工具箱第二版>>（Wiley,2002）。他自 1995 年以来一直为<<智能企业>>杂志撰写专栏文章，并获得了 1999 年度“读者最爱奖”。

Ralph 在斯坦福大学获得了电子工程学博士学位，论文题目是人-机系统设计。他从 1972 年到 1982 年在 Xerox PARC 和 Xerox System 的开发部门任研究员、系统开发经理和产品市场经理。由于他在 Xerox Star 工作站（第一个视窗、图标和鼠标的商业产品）的系统设计工作，IEEE Human Factors Society 授予他 Alexander C. Williams 奖。从 1982 年开始到 1986 年，Ralph 是 Metaphor 计算机系统公司（第一家数据仓库公司）的应用部副总裁。在 Metaphor，Ralph 发明了“胶囊”工具，这是图形化数据流界面的第一个商业实现，现在已经应用在所有 ETL 工具中。从 1986 年到 1992 年，Ralph 成为 Red Brick 系统公司的创始人和 CEO，该公司提供一种非常快速的关系型数据库技术来实现决策支持。1992 年，Ralph 创办了 Ralph Kimball 协会，在 2004 年这个协会演变成为 Kimball 集团。Kimball 集团是一个高度专业化的数据仓库设计专家团体，这些专家因其在咨询、教育、演讲和著作方面的成就而享有较高的声誉。

Joe Caserta 是 Caserta Concepts, LLC 的创始人和负责人。他是非常有影响力的数据仓库专家，其专业技能来自于主流数据仓库工具和数据库的行业经验和实践应用。Joe 就读于纽约哥伦比亚大学的数据库应用开发和设计专业。

1 介绍

抽取-转换-加载（Extract-Transform-Load）系统是数据仓库的基础。一个设计良好的 ETL 系统从源系统抽取数据，执行数据质量和一致性标准，然后规格化数据，从而使分散的源数据可以集中在一起使用，最终再以可以展现的格式提交数据，以便应用开发者可以创建应用系统，也使最终用户可以制定决策。该书围绕这四个步骤进行组织。

ETL 系统既能成就数据仓库也能毁了它。因为虽然创建 ETL 系统是后台工作，对于最终用户并不可见，但是对于实施和维护一个典型的数据仓库系统来说，它所耗费的资源会很容易达到 70%。

ETL 系统能使数据明显地增值，它的工作也绝不是简单的把数据从源系统抽取到数据仓库中。特别是，ETL 系统能够：

- 消除数据错误并纠正缺失数据
- 提供对于数据可信度的文档化衡量
- 为保护数据获取相互作用的数据流程
- 把多个源数据整合到一起
- 将数据进行结构化供最终用户使用

ETL 是个既简单又复杂的题目。几乎所有人都能理解 ETL 系统的基本作用：把源中的数据加载到数据仓库中。另外大多数人也都认为在这个过程中清洗和转换数据是必要的，这就是简单的观点。然而一个无法更改的事实是，紧接着的下一步就要根据数据源、业务规则、现存软件系统以及特定的报表应用系统的不同，将 ETL 系统分拆成成百上千的小的子过程。这带给我们的挑战是，既要耐心地对待这上千个子过程，同时又要保持对整个 ETL 系统主要目标的简单视角。看看本书是如何应对这一挑战的吧！

<<数据仓库 ETL 工具箱>>是创建成功的 ETL 系统的实践性向导。该书并不是所有可行方法的调查和总结！相反，我们只是针对构建维度数据（Dimensional Data）这样的目标而建立了一系列的一致性技巧。维度模型被证明是创建数据仓

库最可预计的和最有效节省成本的方法。同时，由于不同数据仓库的维度结构大都类似，因此我们可以重用大量的代码模块和特殊的开发逻辑。

该书是规划、设计、创建和运行数据仓库后台的路线图。我们将传统 ETL 中的抽取、转换和加载扩展为更可操作的步骤：抽取、清洗、规格化和提交。当然我们并没有试图将 ETL 改为 ECCD！

在本书中，你将学习到以下内容：

- 规划&设计你的 ETL 系统
- 从多种可能的架构中选出最合适的
- 对实施过程进行管理
- 管理日常的操作
- 为 ETL 过程建立开发/测试/生产环境
- 理解不同的后台数据结构，包括平面文件、规范化框架、XML 框架和星型连接（维度）框架
- 分析和抽取源数据
- 创建完整的数据清洗子系统
- 将数据结构化为维度框架，以便更有效提交给最终用户、商务智能工具、数据挖掘工具、OLAP 立方体和分析应用系统
- 使用同一种技术将数据有效地提交到高度集中的或分布的数据仓库
- 调整个 ETL 过程使性能达到最优

以上观点是 ETL 系统中主要的大问题，但是尽可能的，我们还会提供更细层面上的技术细节：

- 针对列属性、结构、有效值和复杂业务规则实施数据清洗系统的关键执行步骤
- 将多个源的异构数据规格化为标准化的维表和事实表
- 创建可复用的 ETL 模块用于处理维表中自然时间变量，例如，三种类型

的缓慢变化维（SCD）

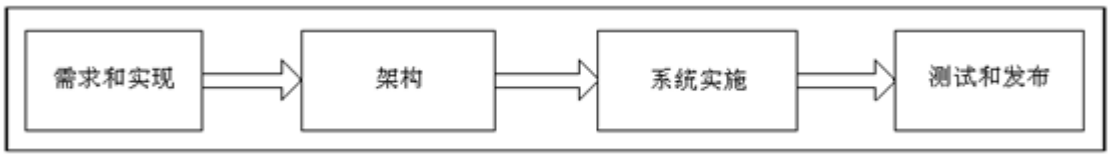
- 创建可复用的 ETL 模块用于处理多值维和层次维，这两者都需要相应的桥接表
- 针对海量事实表的加载进行处理
- 优化 ETL 过程以适应加载时窗的要求
- 如何将批处理和面向文件的 ETL 系统转换为连续的流式实时 ETL 系统

为方便起见，在提供 SQL 代码时我们选择 Oracle 作为缺省的数据库。然而，产生相同结果的类似代码也可以用于 DB2、Microsoft SQL Server 或者其它主流关系型数据库系统。

也许作为所有这些特定建议的副产品，希望我们在创建、部署和管理数据仓库 ETL 系统方面的激情能够感染您。

1.1 全书概览：两条并存的主线

创建 ETL 系统是一个非同寻常的挑战，因为它是如此不可避免的受现实的约束。ETL 小组必须在业务需求、源数据的格式和缺失、现有的遗留系统、员工的技能以及最终用户持续变化（而合理）的需求当中求生存。如果这些困难还不够，那就再加上预算受限，处理时窗太小，如果 ETL 系统不能向数据仓库及时提交数据则重要业务会慢慢停下来！



图表 1　ETL 的规划和设计主线

当创建 ETL 系统时，头脑里必须有两条并存的主线：规划&设计主线和数据流主线。在最高级别上，它们都很简单，两者在图中都从左至右顺序进行。它们相互作用使整个生命周期变得有意义。在图 1 中显示了规划&设计主线的 4 个步骤，在图 2 中显示了数据流主线的四个步骤。

流程检查

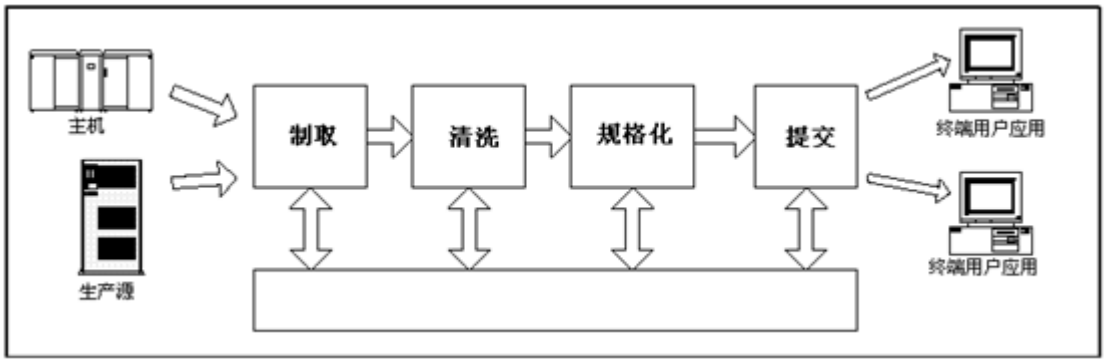
规划与设计：需求/现状 -> 架构 -> 实现 -> 测试/发布

数据流：抽取 -> 清洗 -> 规格化 -> 提交

1.2 规划与设计主线

规划与设计主线的第一步是描述所有的需求和现状，包括：

- 业务需求
- 数据评估和数据源现状
- 监察需求
- 安全需求
- 数据集成
- 数据延迟
- 归档和沿袭
- 最终用户提交界面
- 可用的开发技能
- 可用的管理技能
- 已有的许可证



图表 2　ETL 的数据流主线

我们将在第 2 章中详细讲述这些内容，但是我们必须事先弄清楚它们是怎样影响 ETL 系统的本质。在这一步骤中，以及在两条主线的所有步骤中，当我们讨

论指定步骤时，都会指出其在这本书的位置。

该主线的第二步是架构，在这里我们必须做出关于创建 ETL 系统创建方法的最主要的决定。这些决定包括：

- 手工编码还是使用 ETL 工具
- 批处理还是流数据处理
- 水平任务依赖还是垂直任务依赖
- 自动调度
- 异常处理
- 质量控制
- 恢复与重启
- 元数据
- 安全

规划与设计主线的第三步是系统实施。我们希望在实施步骤之前已经在前两个步骤上花费了相当的精力！这一步骤包括：

- 硬件
- 软件
- 编码
- 文档
- 特定质量检查

最后的步骤看起来像是系统管理，但是测试/发布步骤的设计与前面两个设计步骤至少同等重要。测试/发布包括下列设计内容：

- 开发系统
- 测试系统

- 生产系统
- 提交过程
- 升级方案
- 系统快照和回滚过程
- 性能调优

1.3 数据流主线

数据流主线可能是大多数读者最熟悉的，因为可以认为这是常规的 ETL（抽取－转换－加载）场景的一个简单的概括。当你阅读下面的列表时，你可以开始去想像规划&设计主线是如何影响下列每一因素的。抽取步骤包括：

- 读取源数据模型
- 连接并访问数据
- 调度源系统，截取通知和后台程序
- 捕获变化数据
- 将抽取的数据集结到磁盘

清洗步骤包括：

- 强制列属性
- 强制结构
- 强制数据和数值规则
- 强制复杂业务规则
- 创建元数据来描述数据质量
- 将清洗后的数据集结到磁盘

紧接着是规格化步骤，包括：

- 业务标志（在维表中）的规格化
- 业务度量和绩效指标（在事实表中）的规格化
- 复制
- 家庭关联信息的规格化（Householding）
- 国际化
- 将规格化的数据集结到磁盘

最后一个步骤，把完美的数据提交给最终用户应用。我们在第 5 章和第 6 章来讨论提交技巧，就像将在第 1 章中提到的那样，无论如何，在菜做好之后，总得把它端到客户面前！ETL 系统的数据提交包括：

- 加载平面型和雪花型维度
- 生成时间维
- 加载退化维
- 加载子维
- 加载缓慢变化维（包括类型 1、2、3）
- 规格化维和规格化事实
- 处理迟到维和迟到事实
- 加载多值维
- 加载 ragged 层次维
- 加载维中的文本型事实
- 为事实表运行代理健 pipeline
- 加载三种基础事实表粒度
- 加载和更新聚合表
- 将提交数据集结到磁盘

在学习上面这个列表时，你可能会说："看起来大多数列表中的内容是建模，不是 ETL。这些问题应该属于前台工作！"我们觉得不尽然。我们曾经与 20 多个数据仓库开发小组进行过沟通，超过一半的小组认为 ETL 系统设计应该与目标表的设计是同时的。这些人虽然也认为应该有两种不同的角色：数据仓库架构师和 ETL 系统设计师，但是，这两个角色却经常是由同一人担当！因此这也就解释了为什么本书包含了从数据源到目标维度数据库配置的所有内容。

数据流的四个基本步骤都由运行步骤进行监控，它贯穿了从抽取到提交的全部步骤。运行包括：

- 调度
- 作业执行
- 异常处理
- 恢复和重启
- 质量检查
- 发布
- 支持

理解这两个基本的主线（规划与设计主线和数据流主线）是阅读本书的真正目的。

1.4 本书如何组织

为了描述清楚这两条主线，我们把本书分成四个部分：

- 需求、现状和架构
- 数据流
- 实施和操作
- 实时流式 ETL 系统

本书从规划&设计的需求、现状和架构开始，因为我们必须为任何类型的 ETL 系统建立一个逻辑基础。本书的中间部分跟踪了从抽取步骤到提交步骤的整个数据流主线。在第三部分我们返回到实施和操作问题上。在最后部分，我们会涉及令人激动的实时流式 ETL 系统这一新领域。

第 1 部分：需求、现状和架构

本书的第 1 部分为后续的内容讲解布置了一个沙盘。即使大多数人都非常希望马上开始将数据迁移到数据仓库，我们还是必须回过头来看一下。

第 2 章：环境和需求

数据仓库中的 ETL 部分通常都面临异常复杂的设计挑战。在本章中我们在你牢记一种方法之前告诉你在需要考虑的需求列表上的一些实质性问题。我们还会介绍你必须遵循的主要结构上的决定（不管你是否这样认为）。

本章是定义数据仓库术语表——至少是本书需要的词汇——的最佳位置，这些词汇包括：

- 数据仓库（Data Warehouse）
- 数据集市（Data Mart）
- 操作型数据存储（ODS, Operational Data Store）
- 企业数据仓库（EDW, Enterprise Data Warehouse）
- 集结区(Staging Area)
- 展现区（Presentation area）

我们在描述 ETL 小组负责的任务同时，也描述了数据仓库的任务。我们简短的介绍了四种基础的数据流阶段：抽取、清洗、规格化和提交。最后还尽可能清晰的描述了为什么我们认为维度数据模型是每个数据仓库成功的关键。

第 3 章：ETL 数据结构

每个 ETL 系统都必须将数据集结为各种永久性的或准永久性的格式。当我们提到“集结（Staging）”时，意味着写数据到磁盘，也正是由于这个原因，ETL

系统有时称为集结区（staging area）。你可能注意到我们建议在每一格 ETL 主要步骤（抽取、清洗、规格化和提交）后都有几种形式的集结。我们在本章讨论采用不同集结形式的原因。

然后我们提供了在通常 ETL 系统中所需的重要数据结构的系统性描述，包括：平面文件、XML 数据集、独立的 DBMS 工作表、规范化的实体/关系框架和维度数据模型。为了描述的完整性，我们提到了一些特殊的表，包括用于验证重要数据集出处的合法性审计跟踪表，以及用于跟踪代理键的映射表。我们通常围绕这些类型的表总结出元数据以及命名标准的报告。本章的元数据部分只是一个初步介绍，因为元数据的问题太过重要，我们会在本书中多次提到。

第 2 部分：数据流

本书第 2 部分展示了用于从多种不同的源系统有效抽取、清洗、规格化和提交数据到理想的维度数据仓库中的实际步骤。我们从用于分析源系统的记录系统的选择和策略建议作为开始。该部分还包括一个重要的章节，即创建 ETL 中的清洗和规格化系统。后两章则把经过清洗和规格化后的数据整合成所需的维度结构，以便提交给最终用户环境。

第 4 章：抽取

我们会在本章的开始阶段介绍，数据分析完成之后，设计逻辑数据映射需要什么样的前提条件。我们会建议创建一个逻辑数据图，并告诉用户怎样利用这样的图来避免关键任务描述的含糊不清。逻辑数据映射提供给 ETL 开发者所需的功能特性来建造 ETL 过程。

数据仓库的一个主要任务是把来自整个企业中不同的旧应用系统的数据整合到一个统一的资料库中。本章提供了专门的技术性指导，用于集成整个企业中的异构数据源，包括主机系统、关系型数据库、XML 源、平面文件、Web 日志和 ERP 系统等。我们会讨论在集成这些数据源时遇到的障碍以及提供如何克服它们的建议。我们还会介绍在多个潜在的不一致数据源间规格化数据的概念，这是下一章要展开详述的主题。

第 5 章：清洗和规格化

在抽取数据之后，我们建议对它们进行清洗和规格化。清洗的意思是确认和修复数据中的错误和缺失。规格化的意思是解决潜在不一致的数据间的标记（Labeling）冲突，以便它们能够在企业数据仓库中一起使用。

本章提供了一些非同一般的特殊技巧和方法，这些技巧和方法对于清洗和规格化系统的建设非常有帮助。本章的主题集中在数据清洗的目标、技术、元数据和衡量方面上。

具体地说，技术部分主要是数据评估和数据清洗的关键方法，衡量部分主要是一些例子，包括如何实现触发告警的数据质量检查，以及如何为保证数据健康而进行的数据质量管理提供指南。

第 6 章：提交维表

本章和第 7 章是本书的关键章节。我们坚信数据仓库的总体目标是，以简单的、可操作的格式将数据提交给最终用户和分析应用系统。维表是业务度量的上下文，同时也是数据的入口点，因为它们是几乎所有数据仓库约束关系的目标，同时为每一行输出提供了有意义的标记。

加载维表的 ETL 过程是非常有挑战性的，因为它必须消除源系统的复杂性，并把数据转换成简单的、可以选择的维度实体。本章一步一步地解释了如何加载数据仓库维表，包括一些最先进的 ETL 技术。本章试图对以下内容能够做尽量清晰的说明：

- 分配代理键
- 加载缓慢变化维（类型 1、2、3）
- 为多值维和复杂层次维生成桥接表
- 对层次维进行扁平化处理，以及对雪花维有选择的进行扁平化处理

我们讨论了一些高级管理和维护问题，包括增量加载维、使用 CRC 代码跟踪维的变化以及迟到数据的处理等。

第 7 章：提交事实表

事实表包含了业务度量信息。在大多数数据仓库中，事实表显然比维表要大得多，但是同时它们更简单一些。在本章中，我们将解释所有事实表的基本结构，包括外键、退化维键和数字型事实本身。我们还会对事实表提供者这个角色进行描述，他负责把事实表提交到最终用户环境。

每一个事实表都应该使用代理键加载，代理键用于将原始事实记录的真实键映射到连接维表所需的正确对应的代理键。

我们描述了事实表的三种基本粒度，它们足以支持所有的数据仓库应用系统。

我们还描述了一些不常见的事实表，包括无事实的事实表，或者目的仅用于登记某个复杂事件的发生，例如汽车事故。

最后，我们讨论聚合表的基本架构，聚合表以物理方式存储汇总数据，正如索引一样，它的目的就是为了单纯的提高性能。

第 3 部分：实施和运行

本书第 3 部分假设读者已经对需求进行过分析，注意到了数据和可用资源的真实情况，并且已经对从抽取到提交的数据流进行了可视化处理。请牢记这些内容，在第 3 部分我们将更详细描述了 ETL 系统的实现和组织运行的主要方法。我们讨论了 ETL 系统中元数据的角色，最后还有 ETL 小组成员的不同职责。

第 8 章：开发

第 8 章描述了开发初始数据加载所需的技术，例如针对缓慢变化维重新生成历史数据，以及集成历史离线数据和当前在线交易数据，另外还有历史事实加载。

本章还提供了计算初次加载所需的时间估算技术，从而发现长时间运行的 ETL 过程，然后提出最小化风险的建议方法。

对于数据仓库项目来说，自动 ETL 处理显然是需要的，但是如何做呢？表加载的顺序和依赖关系是加载数据仓库成功的关键因素。本章回顾了 ETL 调度的基本功能，并且提供了执行 ETL 调度的标准和选项。一旦满足了这些基本原则，那么诸如 ETL 的强制参照完整性和元数据运行维护等主题就都可以进行核

查。

第 9 章：运行

本章我们以下面内容作为开始，包括调度各种 ETL 系统作业的方法，响应告警和异常，以及最后在满足所有依赖关系情况下完成作业运行。

我们会遍历将 ETL 系统迁移到生产环境的所有步骤。就像任何其它关键任务应用系统一样， ETL 系统的生产环境必须得到保障和支撑，因此我们描述了如何建立 ETL 系统保障级别，以确保在调度过程失败时起作用。

为了确定 ETL 系统的执行状况，我们制定了一些关键性能指标，同时我们还揭示了如何监控和捕获统计信息。一旦 ETL 关键性能指标收集完成，我们就拥有了足够的信息来检查 ETL 系统中的组件，以便尽可能地优化整个系统。

第 10 章：元数据

ETL 环境经常被认为应该负责存储和管理整个数据仓库的元数据。确实，ETL 系统是存储和管理元数据的最好的位置，因为 ETL 要对数据作合适的处理就必须知道数据个方面的情况。第 10 章定义了三种类型的元数据（业务、技术和流程），并展示了每种类型应用在 ETL 系统中的具体元素。本章提供了生成、发布和使用不同类型元数据的技术，还讨论了数据仓库在该部分可能增强的机会。最后讨论了元数据标准以及最佳实践，并提供了针对 ETL 的建议的命名标准。

第 11 章：职责和分工

ETL 过程的技术问题只是 ETL 生命周期的一部分。第 11 章主要讨论为了成功实施所需的生命周期中的管理问题。本章描述了 ETL 小组的职责和责任，然后描述了一个详细的项目规划，可以用于任何数据仓库环境的实施。一旦有了管理 ETL 系统的基础，本章就进入了更为详细的项目管理活动，例如项目任务分配、范围管理和小组建设。这有点非技术的章节将为 ETL 和数据仓库项目管理提供最大的帮助。它描述了一个有效小组所需的角色和技能；提供了可以在数据仓库每一阶段重复使用的丰富的 ETL 项目规划。本章还包括在 ETL 生命周期中管理者需要领导其小组的形式。甚至如果你不是管理者，本章也需要阅读，以便

充分理解你的角色如何与 ETL 小组中其它成员一起工作。

第 4 部分：实时流式 ETL 系统

由于实时 ETL 是相对较新的技术，我们更像是在讨论一个还没有完全实现的需求和解决方案。在本章中，我们会分享我们的经验，对实时数据仓库的最新挑战进行分析，并提供应对这些挑战的建议。本章提出了实时 ETL 的关键难点，以及真正实现的详细描述。

第12 章：实时ETL

在本章中，我们以定义实时需求开始。紧接着，我们回顾目前可选的各种架构并对它们进行评价。我们以一个决策矩阵结束本章，帮助你决定对于特定的数据仓库环境哪一种实时架构是合适的。

第13 章：总结

最后一章总结了本书的贡献，并提供了 ETL 和数据仓库作为一个整体在将来的展望。

1.5 谁将阅读本书

任何参与或者将要参与数据仓库建设的人都应该阅读本书。开发者、架构师和管理者将从本书获益，因为它包含提交一个面向维度的数据仓库的详细技术描述，还提供了关于所有后台活动的项目管理视图。

第 2、3 和 11 章提供了 ETL 的功能视图，数据仓库小组每一成员都能很容易地阅读，但其主要是针对业务发起人和项目管理者的。在你阅读这些章节时，希望能提高你的技术水平，最终能成为开发者的手册。本书是加载维度数据仓库所需任务的建议性向导。

1.6 总结

本书的目的是让 ETL 系统的建设过程变得可理解。本书展示了 ETL 系统带给数据仓库数据的增值作用。我们希望你能够喜欢本书，并发现它在你的工作中

是有帮助的。我们在整本书中尽量保持中立，以便你能选择应用你喜欢的技术。如果本书没有什么作用，我们希望它能鼓励你思考，并给供应商提供一些挑战，使其扩展其产品的功能来满足 ETL 小组的需要，以便使 ETL（和数据仓库）更加成熟。

第一部分　需求、现状和架构

2 一切围绕需求

理想情况下，你必须从最困难的挑战之一开始设计你的 ETL 系统，那就是：一切围绕需求。这意味着要集中收集所有已知的需求、现状和影响 ETL 系统的约束关系，我们把这些简称为需求。

需求是我们必须要面对的，并且系统也必须适应这些需求。尽管在需求框架内，你还有很多地方可以作出你自己的决定，验证你的判断，发挥你的创造力，但是，需求正如其字面含义，是必须要满足的。该章的第一节将试图帮你勾画出全部的需求类型，并且让你明白在开发 ETL 系统过程中需求是何等的重要。

紧接着需求后面，我们确定一组在 ETL 项目开始阶段需要制定的架构方面的选择。这些选择是重要的前提，因为它们将在后面的实现过程中决定你所做的一切。该架构会影响软件、硬件、编码习惯、人员和操作。

最后一节描述了数据仓库的任务。我们还仔细定义了数据仓库的主要架构组成，包括后台、集结区、操作型数据存储和展现区。我们还对数据集市和企业数据仓库作了详细而明确的定义。请仔细阅读本章，这里描述的定义和边界确定了整本书的逻辑。如果理解了我们的思想，你会发现为什么我们的方法会比其它任何数据仓库设计方法论更合乎逻辑。最后，作为本章的结尾，我们简单陈述了 ETL 小组的工作任务。

流程检查

规划与设计：*需求/现状* -> 架构 -> 实现 -> 测试/发布

数据流：还没有到数据流步骤。

2.1 需求

在本书的介绍部分，罗列了我们认为非常重要的需求类型。虽然需求的每个部分都可以精彩纷繁，但业务方面的需求始终是更基础的和更重要的部分。

业务需求

业务需求是数据仓库最终用户的信息需求。我们使用业务需求这个术语来狭义地表示最终用户作出决策所需的信息内容。其他方面的需求会扩展业务需求的定义，当然这也意味着 ETL 小组必须引入更多的信息源。

在这里需要注意的是，业务需求直接决定了数据源的选择。ETL 小组的核心工作就是不断地理解和检验业务需求。

在《数据仓库生命周期工具箱》一书中，我们描述了调查最终用户和收集业务需求的过程。许多情况下，最初对数据源的调查不一定完全反映数据的复杂性和局限性。ETL 小组常常要做出重要的判断——原始数据源是否能解决用户的业务需求。当然，ETL 小组也常常会发现数据源的额外的能力，从而扩展最终用户的决策支持能力。从更广泛的意义上讲，业务需求和数据源的内容都是不断变化的，需要不断地进行检验和讨论。

合规需求

近年来，尤其是 2002 年的 Sarbanes-Oxley 法案通过以后，企业上报的数据必须更加准确与完整，绝不允许篡改。当然，像电信这样的行业使用数据仓库来满足日常报表需求已经有很多年。但无论如何，对财务报表的要求已经变得越来越严格。

也许有些财务报表的问题跟数据仓库无关，但其他大部分的问题都能够在数据仓库范围内解决。对数据仓库典型的需求包括：

- 数据源的归档备份以及随后的数据存储
- 任何造成数据修改的交易记录的完整性证明
- 对分配和调整的规则进行完备的文档记录
- 数据备份的安全性证明，不论是在线还是离线进行

数据评估

正如 Jack Olson 在他的《数据质量：精度维度》一书中指出的那样：数据评估是设计任何数据使用系统的前提。他指出：“数据评估使用分析方法来检查数

据，充分了解数据的内容、结构和质量。好的数据评估能够处理海量数据，使用分析方法找到需要解决的全部问题。”这方面对于 ETL 小组尤其重要，因为可能有些数据源的内容没有被很好的理解。例如，可能有某一数据源很好地满足了生产部门的需求，比如订单系统，但对数据仓库来说却可能是个灾难，因为数据仓库需要使用的字段并不以订单系统为中心，订单系统的信息对于数据仓库的分析来讲远远不够。

数据评估是一个系统的检测过程，对 ETL 需要使用的数据源的质量、范围和上下文进行检查。从极端意义上讲，一个清洁的数据源是一个维护良好的数据源，只需进行最少量的数据转换和人工干预就可以直接加载到最终的事实表和维表。但对于脏数据源可能需要如下的步骤进行处理：

- 完全清除某些输入字段
- 补入一些遗失的数据，产生特殊的代理键
- 自动替换掉某些错误数据值
- 在记录级别上进行人工干预
- 对数据进行完全规范化的表述

从更深的意义讲，如果数据评估指出某些数据源缺陷很大，不能支撑业务目标，其结果可能关系到数据仓库的成败。数据评估步骤为 ETL 小组不仅提供了关于数据清洁程度的指导，还能防止 ETL 小组漏掉一些重要的里程碑。一定要预先进行数据评估！使用数据评估的结果去说服业务主管，安排现实的开发进度，认清数据源的限制，追加投资用于在源系统中进行数据捕获。在第 4 章我们将详细研究数据评估和数据质量算法。

安全需求

近年来所有 IT 领域对安全关注的程度都有极大的提高，但对于大多数数据仓库小组而言，安全问题依然是一个事后的、额外的负担，对安全性的考虑充满争议。数据仓库的目的是为决策者提供广泛的数据，而安全性则要求数据仓库只向用户提供应该访问的数据。

本书中我们推荐一种基于角色的安全定义方法，这种方法在应用一端控制能

够访问的数据仓库数据。这意味着最终用户的安全授权不是在数据仓库的物理表一级，而是在被称为目录服务器的 LDAP 服务器中定义，各应用系统的用户安全定义都在各自的应用系统中进行授权。本书的后续部分将会安全问题进行详细的讨论。

这样的安全处理方式对 ETL 小组最直接的好处在于，不必过多考虑安全性的设计和管理，但是 ETL 小组仍需要考虑对物理表的读/写权限。ETL 小组的开发环境应当是在位于包过滤网关后面的子网内，如果 ETL 环境位于整个企业内网，那么内网内的任何恶意代码都可能在不知不觉中被安装，从而截获对数据库的管理员口令。

另外，安全的范畴必须扩展到数据的物理备份。如果磁带或硬盘可从物理备份中轻易移走，那么安全性就存在一定的漏洞。

数据集成

数据集成对 IT 人员是一个比较大的题目，因为 IT 人员的最终目标是将全部系统无缝的集成到一起。“360 度的业务视图”就是对数据集成的业务解释。很多情况下，严格意义上的数据集成应当在数据进入数据仓库之前完成，但其实只有极少的时候才如此，除非企业只有一个单一的数据源，比如 ERP 系统，即使这样，通常情况下某些重要的交易系统也是位于 ERP 系统之外的。

本书中，数据集成采用规格化的维表和事实表。所谓规格化的维表是指在分离的数据库中建立公共维度实体（通常是文本标记和度量的标准单位），这样使用这些实体就能够构建横向钻取型的报表。在第 5 章和第六章中将会详细介绍这个过程。

规格化的事实表意味着在分离的数据库中建立公共业务度量，比如关键绩效指标（KPI）等，以便于通过差值和比率对这些数值进行比较。

ETL 系统中，数据集成是数据流程中一个独立的步骤，叫做规格化步骤。物理上讲，此步骤需要为一致性维度实体和事实设置通用名称。

数据延迟

数据延迟需求用于描述数据提交到最终用户的速度。数据延迟对系统的实施

和架构有巨大的影响。从这一点上讲，本书中介绍的大多数传统的面向批处理数据流方式的系统都能通过更优化的处理算法、并行处理或者更有效的硬件来提高性能。但从另一个角度讲，如果数据延迟需求太紧迫，ETL 系统的架构就必须从面向批处理架构转化成面向流的架构。这种转换不是一个渐变的过程，而是在数据提交的每个主要步骤都要做大的改动，需要重新实现。我们将在第 11 章中对这种面向流的实时系统进行介绍。

数据归档和数据线性化

我们在前面的章节中提到了这一点。但即使没有法律法规对数据存储的要求，每个数据仓库也需要保留不同的数据备份，原因也许是为了与新数据进行比较以生成新的记录快照，也许是为了重新处理。

本书中，我们推荐在每个主要的转换发生时集结数据。这些集结点发生在全部的 4 个步骤：抽取，清洗，转换与提交。那么，什么时候要将集结（将数据写入到磁盘）进行归档（将数据无限期地存储到永久的介质上）呢？

保守地讲，全部的集结数据都应当归档，除非已做出明确的决定，决不会恢复某个数据集。与之后要在 ETL 系统中重新处理数据相比，从永久性介质上读取数据几乎不算什么麻烦。当然，在经过一定时间后也不太可能仍按照旧的算法处理数据。

同时，每个集结/归档数据集都应当有与之相对的元数据，描述原始数据源和处理步骤。另外，对于合规需求而言，追踪记录所有的数据沿袭是严格必要的，应当成为每次归档流程的一部分。

最终用户提交界面

ETL 系统的最终步骤是将应用提交给最终用户。提交过程占据十分重要的位置。ETL 小组必须与建模小组紧密配合，对构成最终用户应用的数据结构和内容进行严格把关，务必使其简单快捷。将使用复杂、查寻缓慢的应用直接交给最终用户是不负责任的。最常犯的错误就是将完全规范化的数据模型直接交给最终用户就置之不理了。这也是为何在第 5 章和第 6 章会详细阐述建立最终提交给用户的维度模型的原因。

通常来讲，ETL 小组，数据建模人员以及最终用户应用开发人员需要紧密配合，开发针对详细需求的数据模型。而且，即使物理数据格式完全正确，不同的终端用户工具也有各自需要注意避免的用法，以及需要充分利用的优势。对 OLAP 立方体的数据准备也需要考虑相同的事情，我们将在第 6 章描述。

可用的技能

建立 ETL 系统时，重要的设计方面的决定必须由那些创建和管理系统的人做出。不要建立一个基于 C++处理模块的系统，即使这些编程技巧是公开的，也不必掌握这些技巧。如果你具有某些 ETL 工具的使用经验并且知道如何管理项目，那么在建立 ETL 系统时可能会有信心的多。

下一节中，我们将讨论到底是通过编码方式开发 ETL 系统还是直接使用 ETL 工具。在这里，我们的观点是抛开技术问题和许可证费用不提，也决不要使员工和管理人员使用此系统极不顺手。

已有的证可证

最后，很多情况下，主要的设计决策由上级管理部门做出，坚持使用已有的许可证。大部分情况下，这个要求是能够接受的，并且也是符合实际需要的。但有些情况下使用原有的许可证却是个错误。这是个很困难的处境，就算你对此很有信心，也可能是在需要拿工作做赌注了。如果执行上级管理部门的决定，继续使用已有的许可证，那就必须有充分的准备和坚定的信心去接受最后的结果，否则就可能需要另谋高就了。

2.2 架构

选择何种架构是 ETL 系统设计之初就必须作出的最基本的决定。架构的选择会影响一切，架构的任何修改几乎都意味着整个系统要从头开始实施。架构选择一旦作出，使之最有效发挥作用的关键是坚持使用这种架构。你应该有目的地阅读下列子节中的每部分，其目的是首先确定特定的架构选择，然后应用在 ETL 系统中的任何地方。另外，虽然该节中每一部分都很重要，但早期架构选择中最重要的是确定使用供应商的 ETL 工具还是自己手工编码方式来实现系统。几乎所有设计 ETL 系统的细节都基于这次选择。

流程检查

规划与设计：需求/现状 -> *架构* -> 实现 -> 测试/发布

数据流：还没有到数据流步骤。

ETL 工具还是手工编码（购买工具集还是自己写程序？）

在《智能企业》杂志的一篇精彩的文章（2003.5.31，Ralph Kimball 编辑）中，Gary Nissen 对这个问题总结概括后，回答是："看情况"。我们展开来分析一下 Gary 的观点。

基于 ETL 工具的优势

- 根据某个 ETL 工具厂商的说法："一个有价值的工具目的不在于能够解决琐碎的问题，而是要将不可能的事情变得可能。"
- 开发过程简单、快速、低成本。如果项目足够大或足够复杂，工具的成本会显得微不足道。
- 了解业务知识但不精通编程的技术人员也能够有效使用 ETL 工具。
- 很多 ETL 工具内部集成了元数据资料库，这样，源系统、目标数据库以及其他 BI 工具的元数据既可以同步到这个元数据资料库中。
- 大多数 ETL 工具在每个处理流程会自动产生元数据，从而强制所有的开发人员必须遵循一致的元数据驱动的方法论。
- 大多数 ETL 工具都有内置的调度器，帮助生成文档、简化创建过程以及变更管理。如果系统发生故障，ETL 工具会负责处理全部复杂的依赖关系以及差错控制。
- ETL 工具的元数据资料库能够自动的产生数据沿袭（便于反查）和数据依赖分析（便于前查）。
- ETL 工具为大多数源和目标系统设有预置的连接器。在技术层面，ETL 工具应当处理能够处理全部种类的数据类型转换。
- ETL 工具通常提供内嵌的加密和压缩功能。.
- 大多数 ETL 工具处理大数据集时具有较高性能。如果数据量非常巨大，或者在未来几年可能会非常巨大，那么应该考虑采用 ETL 工具。

- ETL 工具可以跨服务器的复杂的负载均衡，避免服务器死锁。
- 当底层框架变更时，会对后续流程和应用造成影响，大多数 ETL 工具可以自动的进行这种“变更-冲击”分析。
- 在指定的处理模块中，ETL 工具可以使用编程语言进行扩展。比如，如果其自带的模块不能达到要求的功能和性能，就可以在 ETL 工具的数据流程中添加一个自定义的 CRC（循环冗余校验）算法；或者，将一个自定义的周期算法作为数据质量检查步骤的一部分加入进去，以判断某个观测值是否合理，等等。

手工编码方式的优势

- 在手工编码方式下可使用自动单元检测工具。比如，JUnit 库(www.junit.org)是一个著名的、良好支持 Java 编写单元测试程序的工具，对其他语言也有一些类似的包。你可以使用某一个脚本语言，比如 TCL 或者 Python 来设置测试数据，在 ETL 过程中运行，然后验证结果。采用这些自动化测试流程能够极大的提高 QA 人员的工作效率和最终提交版本的质量。
- 面向对象的编程技术可以帮助你保持差错报告、有效性和元数据更新的转换一致性。
- 如果建立了自己的元数据访问接口，手工编码方式可以更直接的管理元数据。
- 对 ETL 系统的简要需求分析可以指导你快速使用基于文件的流程，而非数据库的存储过程。基于文件的处理流程要更加直接，编码更简单，测试更容易，也更便于理解。
- 可以不必考虑用户许可证问题。
- 可充分利用内部的编程人员。
- 基于某个工具的开发使你只局限于某个工具厂商的工具及其特殊的脚本语言。手工编码方式的系统可使用通用的编程语言。(公平的讲，所有 ETL 工具在各自的模块中都与标准的编程语言隔离开。)

- 手工编码方式可以提供无限的灵活性，几乎可以完成任何想做的事情，当然前提是你确实需要这样。很多情况下，一种独特的方法或一种特定的语言可以提供非常的优势。

这里我们还可以为采用 ETL 工具提出新的理由：ETL 工具自动完成文档的能力更强，可维护性更高，尤其是 IT 人员流失较严重的情况下。而反方的观点是，如果 ETL 开发小组有很强的软件开发能力，并且管理良好，文档和维护性不应当是一个大问题。

使用已证实的技术

建立数据仓库时许多初始成本是无法避免的。比如必须购买一些专用的服务器：至少一台数据库服务器，一台 BI 服务器，通常还有一台专用的 ETL 服务器。同时还需要数据库的许可证，BI 工具的用户许可证。在开始一个新的项目时可能还需要购买一些咨询服务以及其他的一些成本。建立数据仓库的这些成本是必须要花费的。但是，有一种成本通常被认为是可有可无，并经常作为成本消减的目标——这就是购买一个专门的 ETL 工具。不用 ETL 工具实现数据仓库是可能的，本书中也没有假定用户必须使用或不使用 ETL 工具。但是，从长远上看，购买 ETL 工具对于建立和维护数据仓库来讲是降低成本的。使用已经证明的 ETL 技术有一些特别的优势：

- 一次定义，多次使用。通过共享并重用业务规则和结构化程序，保持数据仓库中数据的一致性。
- 冲击分析。判断哪些表、列、过程会由于某些修改而受到影响。
- 元数据资料库。轻松地创建、维护和发布数据沿袭，继承数据建模工具中的业务定义，在 BI 工具中展现元数据。
- 增量聚合。动态更新聚合表，只将最新的和修改过的数据加入到聚合表中，而无需在每次数据加载时重建聚合表。
- 管理批处理加载。减少 Shell 脚本，执行条件加载，加载统计，自动 Email 通知等等。
- 轻松连接到多种复杂数据源，如 SAP 和主机系统等。

- 并发执行多线程操作。
- 供应商经验，包括针对维度模型的成功经验和数据仓库支持方面的记录跟踪。

与能够用到 ETL 工具的很多高级特性相比，购买一个 ETL 工具更重要的理由是可以帮助避免走太多的弯路。这些 ETL 工具的设计基于一个目的：准确执行用户想要完成的工作——加载数据仓库。大多数工具已经发展成为能够支持从不同的异构数据源抽取数据、完成复杂的数据转换、加载维度数据仓库的稳定、强大的 ETL 引擎，

在 ETL 配置中不要加入新的、未经测试的产品。像月度仪表盘这样的功能尽管很吸引最终用户，但若放在后台就太不计后果了。保守一些，等待 ETL 产品的成熟。与那些具有良好记录的厂商配合，确保在将来的 5 年中会不断的支持和完善这些产品。

批处理式数据流还是流式数据流

ETL 系统的标准架构是从数据源中周期性的以批处理方式抽取数据，流经整个系统，最后以批处理的方式对最终用户表进行批量更新。本书也是主要基于此种架构组织。但正如我们在第 11 章中描述的那样，如果数据仓库加载的实时性要求变得很急迫，批处理的方式就会被打破。替代的方法就是流式数据流，记录级数据从源系统不停顿地流向最终用户数据库和屏幕。

批处理方式到流方式的改变会改变所有的一切。尽管我们仍然必须支持基本的抽取、清洗、转换和提交步骤，但这些步骤都必须进行修改，以适应实时性记录处理需求。尤其是对于快速流方式而言，很多关于数据到达甚至参照完整性方面的常规假设都必须进行修改。比如，某个新客户的销售交易记录可能在这个客户的描述信息到达之前就已到达。甚至，即使客户身份得到识别后，增强/清洗/复制过的客户记录也可能在最初事件的数个小时或数天后才到达。所有这些都需要对处理逻辑和数据库进行更新，而这种更新对于面向批处理方式的数据流可能是要尽量避免的。

本节的开始部分，我们建议对整个数据仓库只采用某一种架构模式。很明显，无论是选择批处理方式还是流方式，这种选择都应当是基于不同的应用需求的。

在第 11 章中，我们将讨论两种方法的共同点，以及在流方式下应用批处理方法的效果。

水平方式还是垂直方式的任务依赖

水平方式组织任务流是指每个最终的数据库加载相互独立运行。因此，如果有订货和配送两项任务，这两项数据库加载任务会相互独立运行。这通常意味着抽取、清洗、转换和提交的步骤在两个工作流之间是非同步的。

垂直方式任务流会对多个离散的作业进行同步，这样最终的数据库加载会同步进行。尤其是多个系统使用共同的维表时，比如客户或供应商等，之前的步骤一定要同步。这样，如果之前的步骤没有全部执行完，后面的步骤，比如转换或提交等就不会往下运行。

调度自动化

另一个架构方面的选择是，在多大程度上使用自动化调度技术控制整个 ETL 系统。一种极端的方式是，每一个作业都是由输入一个命令行或者点击某个图标发起；另一种极端的方式是，有一个主调度工具管理全部的作业，判断作业是否成功运行，查看各种系统状态是否满足要求，处理人机交互，比如输出警报和作业流状态报告。

异常处理

异常处理不是在文件中放置一些随机的少量的警告和注释，而应当是系统地、统一地将整个 ETL 过程中抛出的全部异常事件放置在数据库中，包括进程的名称，发生异常的时间，诊断的严重性，应当采取的动作，以及异常的最终状态等等。因此，每个作业都需要向数据库写入异常报告的记录。

质量控制

类似的，用户应当对数据处理过程中产生的质量问题制定一种常规的应对机制。除了触发异常报告机制外，所有的质量问题还应当产生相应的审计记录附加到最终的维表或事实表的数据中。错误的或可疑的数据需要按照一定的方式进行处理，比如在缺失的文本数据中填入问号，或者，在提交到数据仓库之前使用最小偏差估计量的数值来替代错误数据等等。这些主题在第 4 章中会深入讨论。

恢复与重启

从一开始建立 ETL 系统的时候，就应当考虑如何保证系统从非正常结束状态下的恢复和重启能力。比如有一个 ETL 作业是从全部的产品种类中抽取某个品牌产品的销售业绩，这样的任务不允许执行两次。在设计每一个 ETL 作业时都需要这样来考虑问题，因为每个作业迟早都会出现非正常终止或者错误地执行多次的情况。无论如何，必须想办法防止发生这种情况。

元数据

来自于关系型数据库表和数据模型设计工具的元数据比较容易获取，但这些元数据可能只占系统全部元数据的 25%。还有 25%的元数据会在数据清洗过程中产生。对于 ETL 小组而言，最大的元数据难题是在哪里以及以何种方式存储流程信息。ETL 工具的一个重要的优势在于它们能够自动维护流程元数据。如果是使用手写编写 ETL 系统，用户则必须构建流程元数据的中央资料库，具体情况参看第 9 章。

安全性

本章在前面描述了推荐的安全架构，即针对最终用户的基于角色的安全模式。ETL 环境中的安全问题虽然并不比针对最终用户的安全性还要细化，但是，为了满足安全需求，要求针对 ETL 环境中的每个表和备份磁带都要有物理上和管理上的安全设置。最敏感和最重要的数据集需要操作系统打印出针对这些数据集的每一次访问和每个指令的详细报告。打印日志应当锁定在房间里的特定打印机，不对普通的 IT 人员开放。存档的数据集存储时应当带有校验，以表明它们没有被修改。

后台——准备数据

流程检查

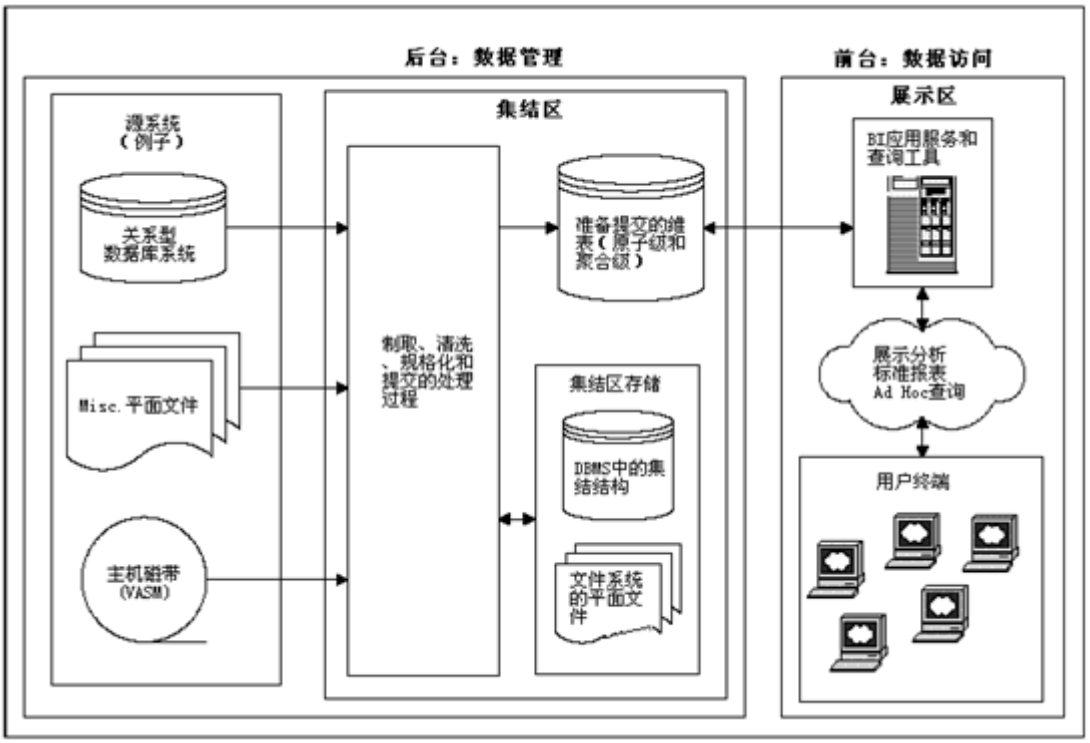
规划与设计：需求/现状 -> *架构* -> 实现 -> 测试/发布

数据流：抽取 -> 清洗 -> 规格化 -> 提交

数据仓库的后台和前台从物理上、逻辑上以及管理上都是分开的。换句话说，大多数情况下，后台和前台都基于不同的机器，依赖于不同的数据结构，由不同的 IT 人员管理。

图 1.1 显示了典型数据仓库中两个独立的组件。

准备数据，通常也叫做数据管理，是指获取数据并将数据转化成信息，最终将这些信息提交到前端的查询界面。后台不提供查询服务，请反复理解这句话。我们的数据仓库方法论假设在后台数据访问是被严格禁止的，这是前台的唯一目的。



图表 1.1　数据仓库的后台和前台

假设有一个餐厅。餐厅的客人是最终用户，提供的食物是数据。提供给餐厅里客人的食物完全符合客人的要求：干净，组织良好，每一片都能轻易辨认与食用。但是，在食物进入餐厅之前，是在厨房中由经验丰富的厨师仔细准备。食物要进行挑选、清洗、切片、烹饪以及摆放。厨房只是工作间，不允许客人访问。在好的餐厅，厨房是完全与客人隔离的——一旦暴露厨房，那里食物还处于半加工状态，是会严重影响客人的胃口的。如果客人需要了解食品准备的信息，厨师必须从厨房出来到餐厅中面对客人，在一个安全、干净、客人觉得比较舒服的环境中解释食物的准备过程。

集结区就是数据仓库的厨房。它只对经验丰富的数据集成专家开放。这是一个后台的处理工具，不允许最终用户访问。在这里，从源系统抽取来的数据进行

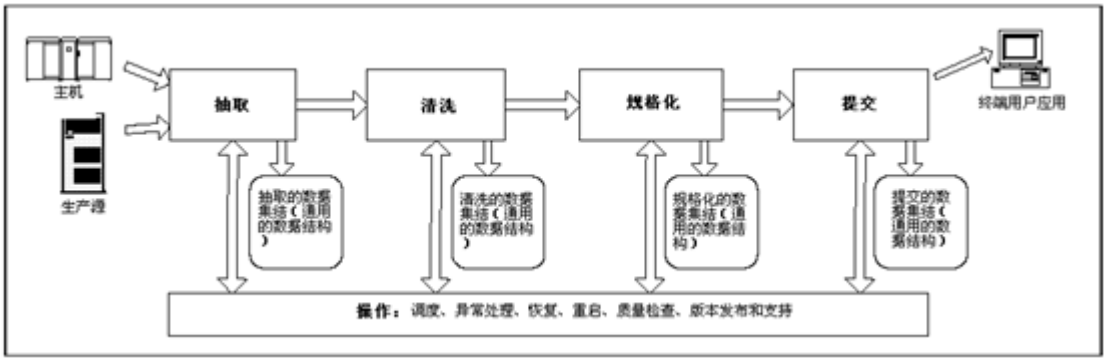
清洗、处理和准备，然后加载到数据仓库的展示层。由 ETL 过程产生的对最终用户有用的元数据都来自于后台，并在数据仓库的展示层进行展示。

后台系统禁止数据访问使得 ETL 小组不用关心下列事情：

- 在行、列和应用一级提供详细的安全性
- 建立提高查询性能的索引和聚合
- 提供不间断的服务运行时间
- 确保全部数据集的一致性

当然我们最终还是需要完成上述的要求，但只不过是在前台，而非后台而已。实际上，是否支持数据访问正是后台和前台的关键区别。如果有例外的情况，允许最终用户直接访问后台系统结构，那么以我们的观点来看，这将严重损害数据仓库。

回到后台系统，我们经常使用集结（Staging）这个词来描述后台的各个独立步骤。集结意味着临时的或永久的物理数据快照。在几乎每个数据仓库中都能发现 4 个集结步骤，如图 1.2 所示。这与本书前面介绍的四个步骤的数据流程一致，只不过使用集结步骤更加明显。在本书中我们假设每个 ETL 系统都是按照这四个步骤组织，每个步骤的数据在传递给下一个步骤的同时也并行写入到磁盘中。



图表 1.2　数据仓库的四个集结步骤

本书中间部分的章节会重点介绍这四个步骤，这四个步骤是：

1. 抽取。源系统的原始数据在进行大的转换之前通常直接写入到磁盘。来自于结构化源系统的数据（比如 IMS 数据库，或者 XML 数据集）在这一步中经常写入到文本文件或者关系型数据库表中。这使得最初的抽取尽可能简单和快速，

并且如果发生中断的话，重新抽取会有更大的灵活性。最初获取的数据在以后的步骤中可能会被多次读取，有些时候，在清洗步骤完成后，最初的数据会被丢弃，有些时候则可能归档以长期保存。也有可能最初的数据要保存至少一个抽取周期，以便于计算连续的抽取之间的不同之处。

我们可以在清洗和转换步骤进行重要的内容转换，但是解决遗留数据格式问题的最佳时机还是在抽取阶段。这些格式问题包括计算重复的组、重定义、重载列、执行低级别数据转化，比如，将位编码转化为字符，将 EBCDIC 转化为 ASCII，将小数转化为整数。具体细节我们将在第 3 章讨论。

2. 清洗。大多数情况下，源系统可接受的数据质量程度依据数据仓库要求的质量而不同。数据质量的处理可能包括几个独立的步骤，包括有效值检测（如是否是已有的邮政编码？是否在有效值范围内？）、一致性检测（如邮政编码与城市代码是否一致？）、删除重复记录（如是否有同一个客户出现两次而相关的属性略有不同？）、检测是否有复杂的业务规则和过程需要增强（如白金客户是否有相关的信用状态？）等等。数据的清洗转换可能需要人为的干预和判断。数据清洗步骤的结果往往半永久保存，因为需要的转换往往难度非常大，并且是不可逆的。另外，清洗过的数据是否需要返回到源系统以提高数据质量，从而减少抽取时可能发生的问题呢？这是个很有趣的问题。即使清洗过的数据不能物理返回到源系统，也应当具备数据异常报告机制以提高源系统的质量。这些数据的问题在最终的商务智能应用中也是非常重要的。

3. 规格化。 当多个数据源合并到数据仓库时就需要数据规格化了。除非每个数据源的文本标记完全相同，并且数值度量已经经过合理的算术计算，使得不同度量之间的差值和比率有意义，否则不能同时对多个分离的数据源进行联合查询。数据规格化过程比简单的数据清洗要重要的多。数据规格化需要整个企业范围内使用标准统一的口径和度量。本书中将在第 5 章和第 6 章深入讨论此步骤。

4. 提交。 后台任务的终点就是准备好数据以方便查询。这一步骤中至关重要的是将数据物理地组织成简单、对称的框架模型，我们称之为维度模型，或者星型模型。这种框架大大地降低了查询时间，简化了开发过程。许多查询工具都需要维度框架，也是构建 OLAP 立方体的必要的基础。本书中我们强烈地建议将

维度模型作为每个数据仓库后台的最终目标。在第 5 章中，我们将详细描述维度模型的结构，并为围绕着这样的结构建立数据仓库提供充分的理由。关于维度模型的更多介绍，请参看其它的工具箱系列书籍，特别是《数据仓库工具箱（第二版）》（Wiley，2002）。

图 1.2 好像是说我们必须顺序执行抽取、清洗、转换、提交四个步骤，每个步骤之间都有定义良好的清晰界限。但实际上，ETL 系统中可能会有多个同时执行的数据流，数据清洗的步骤经常嵌入到抽取的过程中。

ODS 已经被数据仓库吸收。

十年前，操作型数据存储（ODS）还是一个独立的系统，位于源交易系统和数据仓库之间。它主要是为了回答一些业务问题而建立的紧急抽取，比如“订单是否已配送？”或者“支付是否已完成？”等等。 当数据仓库 ETL 过程的数据有一定延迟或者数据已经被聚合处理时，ODS 是有价值的。很多情况下，ODS 是从一个数据源的紧急查询抽取，它自身也可以作为数据仓库的一个数据源，因为 ODS 也是从源交易系统抽取而来。有些时候 ODS 只提供此功能而不用于查询。这就是为什么 ODS 往往有两个方面：一方面是为了查询，一方面是作为数据仓库的数据源。

现在，ODS 作为独立于数据仓库外的一个离散系统已经不再必要了。现在的数据仓库常常是以日为单位进行抽取，并且很多实时技术允许数据仓库的数据差不多达到当前的时点。与过去相比，现在的数据仓库变得更加面向运营。数据仓库和 ODS 的功能完全重合，看不到在两个系统之间有明显的界限。

最后，不论是早期的 ODS 还是现在的数据仓库常常包含有接口，允许最终用户直接修改产生的数据。

前台——数据访问

在数据仓库展示层访问数据必须与 ETL 系统的建立和管理紧密配合。ETL 系统的最终目的就是为了在展示层中提供维度模型，便于查询工具、报表工具、仪表盘和 OLAP 立方体的访问。前台的数据就是最终用户实际看到的数据。

数据集市是前台一个重要的组成部分。它是为支撑某个业务流程而建立的维

表集合。有些作者也把业务流程叫做主题区。主题区是一个很模糊的概念，具有多重含义。比如，我们常常听见人们把主题区分为产品、客户以及订单等。但实际上，产品和客户实体与真正的以度量为中心的流程——比如订单有很大的差异。按照我们的观点，数据集市是以度量为中心的主题（比如订单），周围围绕着描述性实体，比如产品和客户。

虽然本书并不是专门介绍数据集市，但我们还是需要对它有很好的了解。

1. 数据集市基于数据来源，而非部门的视图。换句话说，在面向产品的企业只有一个订单数据集市。不同部门最终用户的查询工具访问数据集市时只有一个统一的、一致的订单状态。

2. 数据集市包含全部的原子级粒度数据，支持向下钻取到最低的数据级别。数据集市只包含聚合数据的观点是数据仓库设计人员最容易犯下的错误之一。聚合数据不包含最低层次的原子级数据意味着不能回答明细业务问题，不能钻取到详单。数据集市应当是一个金字塔状的维表结构，其基石就是原子级数据。

3. 数据集市既可以集中管理也可以分布式管理。换句话说，企业级数据仓库可能物理上集中在一台机器上，而数据集市则可能等到 ETL 集结区的数据整合达到某个层次后才进行部署，或者数据集市进行独立的开发，与数据仓库的维表事实表并不同步进行。我们认为完全集中式的数据仓库只是理想情况，并不现实。更加现实的情况是增量开发和部分分布的数据仓库。毕竟，企业处于不断的变化中，需要新的数据源和新的视角。因此在真实环境中，我们关注的是数据仓库增量的、可接受的开发策略，而非理想状况下的在数据仓库完成之前对全部信息进行管理。

前台还有很多的任务和责任，超出了本书的范畴。为了避免混乱，本书不讨论下列议题：

- 为提高查询性能对展示层数据进行索引
- 选择前台工具，包括查询工具，报表工具，以及仪表盘工具
- 编写 SQL 语句解决最终用户查询
- 数据挖掘技术

- 预测、行为打分和计算分配
- 最终用户能够访问的库表及应用安全性
- 支撑最终用户工具的元数据
- 最终用户培训和文档

本书关注的是 ETL 系统如何从源系统获取数据并将其提交到前台。

2.3 数据仓库的任务

数据仓库的任务是发布企业的数据资产，用于支持更加有效的决策制定。该任务描述中的关键词是发布。正如传统的杂志发行，其成功的起点和终点是读者，数据仓库成功的起点和终点是其最终用户。由于数据仓库是一个决策支持系统，因此主要的成功标准是数据仓库是否为企业的最重要的决策制定过程提供了帮助。虽然必须仔细管理硬件、软件、劳动力、咨询服务和维护的成本，但是其隐含的成本——支持企业重要决策方面的失败——可能会更大。数据仓库的由 IT 管理的可见成本是可能战术上的，但是更为重要的决策支持成本和收益却是战略上的。

交易数据库在企业中应用已经超过 30 年。虽然十几年来我们一直把各种数据输入到特定的交易应用系统中，但很清楚的是，从这些系统获取数据来进行分析是非常困难的。在数据库应用系统上已经花费了数十亿美元的投资，但它们的数据却像犯人一样被关在系统里面。为了从交易系统中把数据取出来已经花费了不可估算的时间，但就像在迷宫中游走一样，大部分的努力都以死胡同结束。ETL 系统必须完成这个任务，那就是以可用的方式把数据交给最终用户应用系统。

建设一个无所不包的、可靠的数据仓库是一件很有意义的任务，必须有一系列的标准组件来支撑。数据仓库最重要且基础的组件就是后台和前台。该书是关于后台的。

数据仓库是什么

数据仓库是将数据从原有交易系统数据库中提取出来，经过转换后形成有组织的信息的过程，它帮助进行数据分析，支持决策的制定。此过程包括从原始数

据格式到目标数据仓库格式的转换，大多数数据仓库项目中这至少要占去全部时间、人力和开销的 70%。经过多个数据仓库的实践，我们可以得到数据仓库的如下定义：

数据仓库是一个将源系统数据抽取、清洗、规格化、提交到维度数据存储的系统，为决策的制定提供查询和分析功能的支撑与实现。

此定义减少了关于数据仓库实施成本的误解。从历史上看，数据仓库项目最可见的部分是数据访问部分——通常是以产品的形式提供，还有一部分注意力放在了维度模型上。但如果只关注这些，会在数据仓库生命周期中留下巨大的隐患。就算建立了维度模型，数据访问工具也准备就绪，距离数据仓库的真正使用仍然有一段距离，因为需要等待 ETL 过程的完成。多花些时间和精力在后台的数据管理组件上，数据仓库建设者才能够更好地把握数据仓库的真正价值——帮助最终用户的决策——同时为建立数据仓库分配切合实际的预算。

意外的延误可能会使数据仓库项目失败的风险提高，但建立 ETL 过程决不是意外的延误。数据仓库小组都知道 ETL 工程花费会占据数据仓库的绝大多数时间。但如果数据仓库建设者知道数据仓库的部署依赖于 ETL 过程的实现程度的话就不会有这种感觉。ETL 系统在时间上面临的最大风险来自于未知的数据质量问题，我们会在第 4 章中讨论如何降低此种风险。

数据仓库不是什么

构成数据仓库的核心要素到底是什么？这个问题经常被误解。时至今日，就算你询问十位专家关于数据仓库的定义，你可能会得到十个不同的答案。这些答案最大的不同可能都会落到构成数据仓库项目的组件上。要澄清这些误解，任何想要加入数据仓库小组，尤其是 ETL 小组的人员必须要知道他的边界。

数据仓库的环境包括几个组件，每个组件都有自己的技术、工具和产品。最重要的是要记住每个单一的组件都不能构成数据仓库的全部。ETL 系统是数据仓库一个主要的组件，但也需要许多其他的组件才能构成完整的实施。根据我们的经验，我们可以看到数据仓库小组也不断的挣扎在错误观念中。关于数据仓库最常犯的 5 个错误是：

1. 某个产品。与很多产品提供商所声称的相反，你不能直接买到一个数据仓

库。数据仓库包含了系统分析，数据处理与清洗，数据转移，以及最后的维度模型和数据访问。没有一个单一的产品能完成数据仓库的全部过程。

2. 某种语言。 数据仓库不是一门语言，我们不必以学习一门编程语言的方式，诸如 XML，SQL,VB 等，去学习如何编码实现数据仓库。数据仓库由多个组件组成，每一部分或多或少都需要些编程。

3. 一个项目。一个数据仓库包括多个项目，将数据仓库简单的部署为一个项目的尝试几乎都失败了。成功的企业级数据仓库通常是以可管理的数据集市开始的，每个数据集市都可看成是单独的项目，带有自己的项目周期和预算。关键因素在于每个数据集市带有一致的维度和标准的事实表，这样便于将单个的数据集市集成到一个紧密的单元——企业级数据仓库中。随着各个数据集市项目的完成，企业级数据仓库将最终发展起来。因此，思考数据仓库更好的方法是将它看成一个过程，而非一个项目。

4. 一个数据模型。数据模型本身并不能构成数据仓库。通过定义我们可以知道数据仓库是一个全面的过程，必须包括 ETL 过程。毕竟，没有数据，再好的数据模型也没用。

5. 交易系统的一套备份。 另一个常见的错误是认为将交易系统复制成独立的报表系统就建立了数据仓库。就像数据模型本身不能构成数据仓库一样，只完成数据迁移过程而不重构数据存储也不能构成数据仓库。

业界术语使用不一致

本节中，我们归纳一些行业的术语，这些术语可能其它的作者有不同的称呼。可能现实中完全没有统一的行业术语，但至少在本书中使用这些术语我们能有一个清楚的立场。

数据集市

许多作者常常将数据集市定义为回答某个部门特殊的业务问题而建立的数据聚合。当然，这个定义有它自己的准则！但在本书中，我们将数据集市统一定义为，基于企业的原子级粒度的数据，面向业务流程的数据集合，只依赖于数据度量事件的本身，而非预期的用户问题。请注意下列关于数据集市不同的定义：

正确的定义	错误的定义
基于流程	基于部门
原子级数据基础	只是聚合级数据
基于数据度量	基于用户的业务问题

数据集市（也称为维度数据集市）对所有的访问者是相同的，都能访问底层的度量事件。而且，由于维度数据集市基于最基础的原子数据，因此在应用级很少发生改变；通过定义可以知道，数据集市包含了能够从数据源获取的全部明细数据。根据错误的定义建立的数据集市不能处理变化的业务需求，因为它们的数据已经被聚合了。

企业级数据仓库 (EDW)

有时候使用 EDW 这个名称来做为一种专门的实现方法（与未作为企业资产的企业数据仓库相对，EDW 通常泛指大规模的数据仓库资产）。很多人也将 EDW 称之为 CIF（Corporate Information Factory）。EDW 实现方法与本书中介绍的 DW 总线架构（Data Warehouse Bus Architecture）方法有本质的不同。EDW 中的很多主题，需要跟 DW 总线的方法对照着来解释。同时，如果将逻辑问题和物理实现问题分开来看会更有帮助。

从逻辑来看，这两者都提倡对分散在整个企业内部的不同数据源进行统一的定义。DW 总线架构采用规格化维表和规格化事实表的方式。而 EDW 的方法看起来非常的难以归类，你必须坚信如果企业信息有了一个统一的、高度规范化的 ER 模型，就能知道如何管理企业内成百上千张数据表。当然如果不考虑细节，我们也可以同意这两种方法在某一点上是一致的，那就是两种方法都试图对各个分布式数据源应用统一的标准。

即使这两种方法在创建企业数据的一致性表述这一点上达到了稍许的一致，一旦开始进入到物理设计和部署的阶段，EDW 和 DW 总线之间的差异又立刻显现出来了。

DW 总线架构采用一致的维表和一致的事实表，一致的维表具有公共的字段，这些字段的相关值域都是相同的。这一点保证了用户能够对事实表通过其关联的维表进行查询，并且也能够将列合并到最终的结果中。当然也包括横向钻取。更

广泛的讲，我们还能在分布式数据仓库中管理规格化维表和规格化事实表。我们从没有在 EDW 的方法论中看到过有类似的建议。有意思的是，即使是在物理集中存储的 EDW 中，也不得不将数据存储在相互独立的表空间中，这就有必要通过相同的逻辑复制规格化维表。但我们也从来没看见过 EDW 所描述的系统过程提倡这样做。哪些表需要做同步，何时同步？DW 总线架构详细地描述了这一点。

DW 总线方法中维表的非规范化特征允许管理随时间变化的维度（SCD 类型 1，2，3）。而在高度规范化的 EDW 环境中，我们没有看到如何建立和管理缓慢变化维的相关描述。各个实体都需要大量的时间戳，与维度方法相比需要更多的管理键。另外，这里我们描述的代理键方法实际上与维度模型无关。在 EDW 中，如果希望像 SW 总线架构一样能跟踪到缓慢变化维的维度信息变化状况，规范化的雪花型维表的根表就必须对相同数量的记录采取相同的键管理策略（使用代理键或者使用自然键加日期）。

DW 总线维度的非规范化特征使得我们能够用系统的方法去定义聚合表，这一最强大和成本最低的提高大型数据仓库性能的手段。维度聚合技术与规格化维表的使用密切相关。聚合事实表的收缩维度完全是 DW 总线中基础维表的一个子集。而在 EDW 中，没有系统的文档介绍处理聚合表的方法，对查询工具和报表工具如何使用这些聚合表也没有指导。这个问题牵扯到下面要讨论的“向下钻取”。

最重要的，大多数 EDW 架构有一个关键性假设，就是数据集市是由集中式数据仓库发布的。这些数据集市都是基于回答某个业务问题而建，这在前面的章节中已有介绍。如果用户问的问题需要原子级数据才能回答，这种假设就无法成立，又不得不离开汇总的数据集市，降级到后台第三范式的原子级数据上。以我们观点看来，这种方式的每一步都是错误的。两种架构相比，我们提倡的数据仓库总线架构在各个层面上都占优：通过规格化维表向下钻取到原子数据；缓慢变化维的统一编码；使用优化性能的聚合表；以及与查询服务隔离，保持后台集结数据的干净。

解决架构冲突：混合总线方法

有没有可能将两种方法融合呢？我们也在这样想。本书全书，我们都支持对

数据清洗使用规范化数据。通过规范化过程，强制整理脏数据源的多对一关系，数据源会因此而受益。然后我们要求 ETL 小组将规范化的数据结构转换为简单的维度结构，为其后的规格化和提交步骤做准备。这包括原子级的数据。在这一点上，企业对规范化物理结构的投资可以得到利用。我们把它称为混合总线方法。

数据仓库如何改变

正如我们书中写到的，数据仓库正经历着巨大的变化，这也许是自数据仓库诞生以来最大的变化。本书中提到的数据仓库的目标是支持决策制定无疑是正确的，但很多情况下数据仓库新的发展更关注于运营和实时性。尽管对于实时性应用，前台和后台组件仍然是必要的，但传统基于批处理方式的 ETL 过程正让位于基于流方式的 ETL 过程，用户驱动型查询和报表工具正让位于数据驱动和事件驱动型的仪表盘应用。我们将在第 11 章介绍这些新的进展和它们如何扩展数据仓库的概念。

2.4 ETL 小组的任务

最后我们来简要描述 ETL 小组的任务。从最高层面讲，ETL 小组的任务是建立数据仓库的后台。详细一些讲，ETL 系统必须：

- 将数据更有效的提交到最终用户工具
- 通过清洗和转换步骤增加数据的价值
- 保护数据沿袭并进行文档化

我们可以看到在几乎每个数据仓库中后台必须支持如下关键步骤：

- 从原始数据源抽取数据
- 保证数据质量，清洗数据
- 保证来自于各个数据源的数据一致性
- 为查询工具，报表和仪表盘提交物理格式的数据。

3 ETL 数据结构

数据仓库的后台部分经常被称为集结区（Staging Area）。在这里的上下文中，数据的集结过程指的是写入磁盘，并且我们建议在 ETL 数据流的四个主要检查点都要有数据集结。ETL 小组需要不同的数据结构来满足不同的数据集结需求，本章的目的就是描述可能需要的所有类型的数据结构。

本章不会描述所有必须抽取的源数据的类型，我们将在第 3 章中讨论这个话题。

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

我们还将尝试从数据结构的细节回头再看建议的集结区通用设计原则，包括规划和设计标准，并介绍支持集结所需要的元数据。元数据是一个大话题，我们将在第 4 章描述一些清洗和规格化步骤中特定的元数据。另外，从第 9 章直到本书结束我们还会集中地讨论元数据问题。

3.1 是否集结数据

是将数据存储在物理集结区还是在内存中直接处理？这个问题是 ETL 架构中最根本的选择之一。开发的 ETL 处理的效率很大程度上取决于能否很好地均衡物理 I/O 与内存处理。

能够在将数据写入集结表和保持在内存两种方法间取得理想的均衡是个很大的挑战，也是优化处理过程中必需考虑的问题。最终的决定取决于下面的两个彼此矛盾的目标：

- 将数据以最快的速度从数据源获取到最终目标
- 在处理过程发生错误时，能够进行恢复而无需从头开始

根据环境和业务需求的不同，数据集结的策略会有很大的不同。如果计划在内存中处理所有的 ETL 数据处理，不要忘记任何一种数据仓库，无论其架构和运

行环境如何，都包含了一个某种形式的集结区。之所以要在加载到数据仓库之前集结数据，主要是基于如下的考虑：

- 可恢复 在大多数的企业环境中，数据从源系统中抽取出来后，会进行一系列的重要的转换，假设对于某张表，其转换的工作量很大，那么根据我们的最佳实践，应该在数据一抽取完马上就进行集结。这些集结表（在数据库或者文件系统）可以作为恢复点。一旦转换过程发生错误，利用这些表，处理过程就无需再次访问源系统。同样的道理，如果加载过程失败，也无须重新进行转换。如果集结数据纯是为了恢复的目的，那么数据应该存储在文件系统中的顺序文件中，而不是数据库。以恢复为目的的数据集结对于从业务系统中抽取数据尤其重要，因为业务系统中的数据会被覆盖和修改。
- 备份 通常，巨大的数据量使得在数据库级别上进行可靠的数据仓库备份变得不可行。只要加载文件已经进行了保存、压缩和归档，那么我们就可以避免数据库故障所带来的灾难。如果集结表存储在文件系统，那么就可以被压缩成非常小的文件并存储在网络上。当需要重新向数据仓库中加载数据时，仅需要对加载文件解压缩并重新加载。
- 审计 很多时候，源系统和目标系统之间的数据沿袭在 ETL 代码中丢失，当审计 ETL 流程时，数据集结区的存在使得对 ETL 流程中的不同阶段的直接比较成为可能，因为这时候审计人员（或者程序员）可以简单的比较原始的输入文件和输出文件来检查逻辑转换规则。当源系统覆盖了历史数据时，集结数据特别有用。当一个事件发生后，你可能对于数据仓库中几天甚至几周的数据信息的完整性产生了疑问，这时候对相应时段的集结区的抽取数据进行检查将能够帮助你恢复对数据仓库的数据准确性的信心。

一旦决定要集结哪怕只是一部分数据，你都必须要为集结区选择一种合适的架构。对于任何的数据库，如果数据集结区没有认真的规划，最终将会失败。对于数据集结区的严密设计远比通常的应用设计更为重要，因为数据集结区的数据量是不断累积的（有时候会比数据仓库还要大）。下一节将讨论设计集结区时要

考虑的问题以及可能的选择。

3.2 设计集结区

集结区按照自己的方式，为最终的数据仓库展示区来存储数据。有时候，保存集结区数据是为了支持那些需要历史数据才能完成的功能，而其它时候，集结区数据会在每个处理流程完成后就被删除。为维护历史信息而使用的集结区通常称为持久集结区（persistent staging area）。而临时集结区中的数据则在每次加载过程后被删除。大多数的数据集结区都使用混合模式，即同时使用临时和持久的集结表。

请确保你已经仔细地思虑了在整个数据仓库操作过程中集结区所担当的角色。集结区绝对不仅仅是为了支持下一个作业所创建的临时文件。集结区文件可以在后续的过程发生了严重的问题的时候用于恢复工作流，还可以用于审计和对处理过的数据内容进行验证。

除了考虑集结区数据的存储方式，还需要在设计和部署集结区时考虑如下的基本规则。讨论下面规则均假定你是 ETL 小组的成员，如果不是，请跳过此节。为了数据仓库项目的成功，必须建立和实践下列规则：

- 数据集结区的所有者必须是 ETL 小组。数据集结区，以及其中的所有数据不应该对 ETL 小组外的用户开放。数据集结区不是为展示来设计的。这里没有支持查询的索引和聚合表。在数据集结区没有为数据访问和一致性提供保障。这些与数据访问相关的需求应该在展示区来处理。
- 无论任何原因，用户都不许进入数据集结区。在集结区中的数据必须被视为正在处理中（a construction site area）。允许非授权的个人进入集结区将导致一系列的问题。好奇的用户通常会错误地使用数据，降低数据仓库的完整性。
- 报表不能从集结区访问数据。数据集结区是一个工作站点，ETL 小组不需要通知用户就可以对表进行增、删、改操作。注意，这并不是说，程序员可以随意地对所有的表进行增加、删除和修改。而是说，集结区是一个工作区，不是展示区。这个区域是一个受控制的区域，在生产集结

区对表的修改和数据仓库展示层的生存周期是一致的。但是，同更改生产数据仓库的表不同，对数据集结区的表的修改不需要通知用户，这样报表可能会用到这些变更的表而产生错误。因此，不允许任何的报表访问集结区域，尤其当集结表是临时的。

- 只有 ETL 流程才能够读写集结区。数据仓库每天都会需要一些常规外部数据源所不能提供的数据集，例如数据质量状态类型表。即使数据仓库所需要的数据不是来自于现有的任何外部数据库系统，这些数据仍然需要同其他数据一样进入数据集结区。要注意，数据集结区不是交易环境，绝不允许手工输入任何数据。如果必须使用手工维护的数据表，那就需要在数据集结区外面开发一个应用程序，通过应用程序将结果数据提供给 ETL 小组，然后使用 ETL 流程将数据合并到集结区。

ETL 组拥有数据集结区的完全权限。这意味着 ETL 架构师可以在这个集结区内设计表，并根据需求和相应的 ETL 流程来决定数据表应该建在数据库中，还是使用文件系统。一旦集结区创建完成，ETL 架构师必须将整个数据存储的容量估算结果提供给数据库管理员（DBA）小组和系统管理员，管理员根据这个结果为数据库、文件系统和目录结构计算空间分配和参数设定。图 2.1 描述了一个简单的集结区规模估算表，这个样例主要针对 ETL 数据流最终阶段的提交表。针对源数据拷贝和清洗及规格化步骤之后的集结区，一个完整的数据规模跟踪系统拥有类似的数据规模估算表。

ETL 组拥有数据集结区的完全权限。这意味着 ETL 架构师可以在这个集结区内设计表，并根据需求和相应的 ETL 流程来决定数据表应该建在数据库中，还是使用文件系统。一旦集结区创建完成，ETL 架构师必须将整个数据存储的容量估算结果提供给数据库管理员（DBA）小组和系统管理员，管理员根据这个结果为数据库、文件系统和目录结构计算空间分配和参数设定。图 2.1 描述了一个简单的集结区规模估算表，这个样例主要针对 ETL 数据流最终阶段的提交表。针对源数据拷贝和清洗及规格化步骤之后的集结区，一个完整的数据规模跟踪系统拥有类似的数据规模估算表。

表名	更新策略	频率	ETL作业	行数	行长	增长情况	每月行数	每月字节数	初始大小	6月大小
S_ACCOUNT	Truncate / Reload	Daily	SAccount	39,930	27	New accounts	9,983	269,548	1,078,191	2.57
S_ASSETS	Insert / Delete	Daily	SAssets	771,000	78	New assets	192,875	15,044,250	60,177,000	143.47
S_BUDGET	Truncate / Reload	Monthly	SBudget	39,932	104	Refreshed monthly	9,983	1,038,232	4,152,928	9.89
S_COMPONENT	Truncate / Reload	On demand	SComponent	21	31	Components added to inventory	5	163	651	0.00
S_CUSTOMER	Truncate / Reload	Daily	SCustomer	38,103	142	New customers added daily	9,526	1,352,657	5,410,628	12.90
S_CUSTOMER_HISTORY	Truncate / Reload	Daily	SCustomerHistory	2,307,787	162	Refresh with each bulk load	576,927	93,462,134	373,848,534	891.32
S_CUSTOMER_TYPE	Truncate / Reload	On demand	SCustomerType	5	24	New customer types	1	26	105	0.00
S_DEFECT	Truncate / Reload	On demand	SDefect	84	27	New defect names	21	567	2,268	0.01
S_DEFECTS	Insert Only	Daily	SDefects	8,181,132	132	Transaction defects	2,045,283	269,977,356	1,079,909,424	2,574.70
S_DEPARTMENT	Truncate / Reload	On demand	SDepartment	45	36	Departments established	11	405	1,620	0.00
S_FACILITY	Truncate / Reload	Daily	SFacility	45,260	32	New or changed facilities worldwide	11,315	362,080	1,448,320	3.45
S_HISTORY_FAIL_REASON	Insert / Delete	On demand	SHistoryFailReason	6	27	New failure codes	2	41	162	0.00
S_OFFICE	Truncate / Reload	Daily	SOffice	14	56	New offices opened	4	196	784	0.00
S_PACKAGE_MATL	Truncate / Reload	Daily	SPackageMatl	54	18	New packaged material categories	14	243	972	0.00
S_PRODUCT	Truncate / Reload	Daily	SProduct	174,641	73	New products	43,660	3,187,198	12,748,790	30.40
S_PROVIDER	Truncate / Reload	On demand	SProvider	63	45	Service providers	16	709	2,836	0.01
S_REGION	Truncate / Reload	Daily	SRegion	333	37	New or changed global regions	83	3,080	12,321	0.03
S_RESPONSES	Insert Only	Daily	SResponses	5,199,095	105	Response transaction	1,299,774	136,476,244	545,904,975	1,301.54
S_SURVEY	Truncate / Reload	Daily	SSurvey	45,891	80	Survey conducted	11,473	952,238	3,808,953	9.08

图表 2.1　集结区规模估算表

规模估算表中列出了集结区中每张表的如下信息:

- 表名称。在集结区中的表或者文件的名字。在计算表中每个集结表都占用一行。
- 更新策略。这个字段表明表的维护方式。如果是一个永久表，可能的策略为追加数据、更新或删除。如果是临时集结表，在每个过程中都会被删减和重新加载。
- 加载频率。ETL 过程对表中的数据以什么样的频率加载和更改。通常是每日一次。还可以为每周、每月或者任意的时间间隔。在一个实时环境中，在集结区中的表可能不断的被更新。
- ETL 作业。集结区表通过 ETL 作业进行操作和更新。ETL 作业指的是操作集结区表或者文件的作业或者程序。当多个作业操作单个的表的时候，在估算表的这个字段中列出所有的作业。
- 初始行数。ETL 小组必须估计在集结区初始的时候每个表中的记录数。记录数通常和源表和目的表的行数有关。
- 平均行长度。为了估算空间大小，必须将每个集结区表的平均行长度提供给 DBA。在 Oracle 环境中，我们通常在开发环境中创建表，运行统计后从 ALL_TABLES 中收集这些信息。例如，在 Oracle 中 DBMS_STATS 包可

以用于生成相应的统计字段信息。

- 增长。虽然每张表都是按照调度周期进行更新的，但不会每次都增长。在估算表中的增长字段(GrowWith)是基于业务规则。你必须定义集结区中的表何时会增长，例如一个状态表只有在增加新状态的时候才会增长，尽管这些表每天都会被访问来查看是否有变化，但是增加新的状态的情况并不常发生。
- 预计每月行数。这个估计是根据历史和业务规则。DBA 需要根据预期的增长来给表分配空间。每月行数是计算每个月增长多少字节的重要因素。
- 预计每月字节数。预计每月字节数等于平均行长度乘以预计每月行数。
- 初始表大小。初始表大小通常用字节或者兆字节来表示。取值等于平均行长度乘以初始行数。
- 6 个月表大小。6 个月表大小的估算可以帮助 DBA 小组估计集结区数据库或文件系统的增长情况，通常用兆字节表示，计算公式为 （（平均行长度 * 初始行数）+（平均行长度 * 预计月行数 * 6））/1,048,576 。

到目前为止，我们的讨论还是架设在数据库管理系统（DBMS）创建集结表。而实际上，集结区通常会同时使用 DBMS 和文件系统的平文本文件。平文本文件在使用专门的 ETL 工具的时候非常重要。大多数的工具会使用文件系统的区域存储数据以优化其流程。大多数情况，需要使用 DBMS 之外的平文件来集结数据以获得高速的连续处理性能。这种情况下，需要使用文件系统规摸估算表。通常的方式是在开发区中放置文件，然后记录所占用的空间，作为正式空间分配时候的估计值。

本章下一节的内容将帮助你为集结表选择合适的结构。

ETL 架构需要在文件系统中分配和配置数据文件作为数据集结区的一部分，来支持 ETL 处理。使用文件系统的 ETL 工具的厂商会推荐合适的空间和系统配置选项来优化性能和扩展性。对于手工编码的 ETL 流程创建的数据文件的规模估算，可以使用标准的数据量估算表。

3.3 ETL 系统中的数据结构

在本节中，我们将介绍 ETL 系统中可能使用的几种重要数据结构类型。

平面文件

在很多情况下，没有必要在 DBMS 中对数据进行集结。如果不是使用专门的 ETL 工具而是在数据库中使用 SQL 完成所有的 ETL 任务，那么就需要创建 DBMS 表来存储所有的集结数据。但是，如果决定使用 ETL 工具，或者使用那些能够操作文本文件的 Shell 脚本或脚本程序，比如 Perl，VBScript 或者 JavaScript，那么，通常情况下，可以在文件系统中使用简单文件来存储集结数据。

当集结数据像数据库表那样按照行和列存储在文件系统中的时候，我们称之为平文件（Flat File）或者顺序文件（Sequential File）。如果操作系统是 UNIX 或者 Windows 时，那么数据将按照标准化的 ASCII 编码进行存储。ETL 工具或者脚本语言可以像操作数据库表一样方便地操作 ASCII 平文件，而且在某些情况下处理速度会更快！

DBMS 需要很高的代价来维护要处理的数据的元数据，这在一些简单的数据集结区环境下并不真正需要。而且根据经验，诸如排序、合并、删除、替换以及其他的一些数据迁移功能，在 DBMS 之外进行操作要快得多。有很多专门的程序工具可以用来处理文本文件。请记住，在使用脚本语言操作平文件的时候，你有义务为所作的转换提供元数据追踪表。如果元数据追踪（比如，为了合规的目的）和转换本身同等重要，你应该考虑通过 ETL 工具来完成这些操作，因为工具能够自动的提供元数据的上下文。

如下是有利于关系表的观点。

对数据的截断或者插入操作，通常要比写平文件快一些。而且，对于平文件来说没有真正的高效的数据更新这个概念，而这通常是反对使用平文件作为永久数据集结区的理由。操作系统工具或者厂商提供的工具通常并不支持查询或者其他随机的查找操作。最后，平文件不能为快速查找建立索引。

当对 ETL 系统中的集结区表读数据的时候，如果需要进行筛选或者需要在不同的表之间进行联接的时候，使用数据库存储是更好的选择。尽管专门的排序

包可以用来提高效率，但是关系型数据库最近已经在所专注领域取得了显著的进步。考虑到能够使用 SQL 并免费获得自动的数据库并行特性，数据库方案还是非常值得关注的。

使用何种平台来存储数据集结区表取决于很多因素，包括企业标准和实践。无论如何，我们已经发现，当基本的需求是下面所列之一的时候，在 ETL 过程中使用平文件将更加实际：

- 出于保护和恢复的目的而进行的源数据的集结。当数据从数据源中抽取的时候，必须快速地进入系统，选择所需要的数据，然后退出。如果数据抽取出来后，后续的 ETL 过程发生了错误，必须能够重新执行 ETL 过程而不需要再次访问源系统。要确保能够进行恢复而不需要频繁地访问源系统的最好的方法就是，将抽取出的数据在平文件中进行保存。数据进入平文件后，后续的处理发生任何的错误，都可以很容易的重新进行，这时候仅仅需要从集结区中的平文件中获取数据。我们在第 8 章中对如何使用平文件进行 ETL 处理错误恢复进行讨论。
- 数据排序。排序实际上是每个数据集成任务的先决条件。无论是做聚合、校验还是查找，数据的预先排序都可以提高性能。在文件系统中的数据排序有可能比在数据库中利用 ORDER BY 子句来获取数据效率更高。因为 ETL 的本质是要集成不同的数据，合并数据的效率是首要考虑的问题，而这需要大量的排序操作。ETL 系统的处理器周期中会有很大的部分在完成排序。如有可能，需要认真的在 DBMS 和专门的排序软件包中模拟最大的排序任务，根据结果来选择使用哪种方式来完成排序。
- 过滤。假设需要对源数据库中尚未没有建立索引的实体进行过滤，除了强制源系统创建索引——但这通常会妨碍事务的处理——还可以考虑将整个的数据集抽取到平文件，然后 grep（一个用于对简单数据文件进行筛选的 UNIX 命令）出那些符合要求的行。如果数据来自文件而非数据库，那么使用平那个文件进行筛选将更加可行。一个常见的例子就是 Web 日志文件。在处理点击流数据的时候，我们使用带-v 选项的 grep 命令来获得那些不包含特定内容，如.gif,.jpg 的行，这样，那些对图片的点击将被

排除在数据仓库访问范围之外。与平文件的筛选相比，将数据插入数据库表，建立索引，然后使用 WHERE 子句删除或者选取数据插入到另外的一张表中，效率要低得多。

- 替换/部分替换文本串。操作系统可以用 tr 命令以令人难以置信的速度操作文件，将任意的字符串转换为另外的字符串。在数据库中进行子串搜索和数据替换需要使用嵌套的 scalar 函数和 update 语句。这种操作在操作系统一级进行处理会比数据库操作高效得多。
- 聚合。在第 6 章讨论的聚合必须通过两种方式进行支持：在将数据加载到数据库之前的 ETL 数据流中；或者，如果聚合数据只用通过数据库筛选来获得，就在数据库中完成。如果在数据库外完成聚合，往往需要专门的排序软件包。如果是在数据库中完成，将会大量地使用数据库的排序功能，虽然偶尔也会将大量的数据导出，使用排序软件包。
- 源数据的引用。在规范化的事务处理系统中，通常会有一个唯一的参照表来支持其他的表。例如，一个通用状态表可以支持订单的状态、发货状态和付款状态。除了在源数据中反复地查询同一张参照表之外，更为高效的一种方法是一次性地将参照表抽取到数据集结区。ETL 工具将从集结区中查找所参照的数据。大多数的 ETL 工具可以将参照表加载到内存中，并使其在 ETL 整个处理过程中有效。从内存中访问参照表速度极快。另外，使用集结区中的参照表可以使源系统中的很多表联接可以被省略，这样对源系统的查询将更加的简单和高效。

XML 数据集

需要指出的是，XML 数据集在 ETL 系统中通常不用于永久存储集结区数据，它们更适合作为 ETL 系统的输入和输出的标准格式。当然 XML 数据集将来很有可能成为 ETL 系统和数据仓库查询表的永久数据集结格式，但是成为通用标准之前，XML 的层系能力必须和关系型数据库实现深度的结合。

XML 是一种数据通信语言。表面看起来， XML 利用普通文本文档的形式来存储数据以及元数据，但是不包含格式信息。XML 使用很多与 HTML 类似的符号来描述，但是与 HTML 文档架构很不相同。与 XML 相反， HTML 中包含了数据

和格式化信息，但是不包含元数据。

XML 和 HTML 之间的差异对于理解 XML 如何影响数据仓库尤为重要。XML 元数据中包含了在 XML 文档中唯一标识每一个条目的标签。例如，一个发票的 XML 描述如下：

<Customer Name=”Bob” Address=”123 Main Street” City=”Philadelohia” />

这里 Customer 是 XML 元素，在发票传输之前，就已经在发送和接收方完成定义。Customer 元素被定义为包含一系列的 XML 属性，包括 Name，Addressm，City 以及其他可能的内容。

XML 在描述层系结构方面具有很强的能力，例如描述使用嵌套重复字段的表单。这些层系结构并不直接映射为标准二维关系表的行和列。当数据仓库接收到 XML 数据集以后，需要一个复杂的抽取过程将数据转换到关系型数据仓库。现在已经有一些讨论，关于如何扩展关系型数据库，提供对 XML 层系结构的本地支持，但是这需要大量的基于关系数据库的 SQL 语法和语义的扩展，目前还没有具体的实现。

如果不考虑层系结构的复杂性，XML 是目前在不兼容系统中转移数据的最有效的中间层，因为 XML（以及附加的 XML Schemas）中提供了足够的信息，可以为关系型数据库生成完全的 CREATE TABLE 语句，并按照正确的列类型提供数据。XML 定义了数据共享的通用语言，这是它的优势。对大数据量传输而言，XML 的劣势在于 XML 文档结构的冗余。如果要传送几百万条结构类似和可预知的记录，那么需要寻找比 XML 更加有效的文件结构。

DTDs, XML Schemas 和 XSLT

使用 XML 需要双方通过交换特定的类型定义文档（DTD）来识别可能的标签。对 Customer 的 DTD 定义例子如下:

<!ELEMENT Customer (Name, Address, City?, State?, Postalcode?)>

<!ELEMENT Name (#PCDATA)>

再以类似的方式来定义 Address, City, State 和 Postalcode。

在 City，State 和 Postacode 后面的问号表示这个字段是可选的。#PCDATA 声

明表示 Name 是一个不带格式的字符串。

注意在 DTD 中包含的信息比 CREATE Table 中包含的信息要少，通常，不包含字段长度。

到目前为止，DTD 为双方交换 XML 建立了元数据理解的基础。一些工业标准组织已经定义了专属领域的标准 DTD。但是作为数据仓库，我们无法从 DTD 中得到建立关系表的所有的信息。为了改进这个问题，W3C 标准组织定义了继 DTD 之后的新工业标准 XML Schemas。XML Schemas 包含了大量的面向数据库的信息，包括数据类型以及 XML 元素之间的关联关系，换句话说，描述了表是如何联接的。

使用统一的 XML Schemas，在收到 XML 内容后，通过另一个规范 XSLT（可扩展的风格样式语言转换）来展示内容。事实上，XSLT 是将 XML 文档转换成另外一个 XML 文档的通用机制，但最常见的用法是用于将 XML 转换成用于屏幕输出的 HTML。

关系表

集结区数据可以存储在关系型 DBMS 中。尤其是没有使用专门的 ETL 工具时，使用数据库表就最合适。使用数据库存储集结区表有下列的优点：

- 直观的元数据。使用平文件的主要缺陷是缺乏直观的元数据。使用关系表存储数据，DBMS 自动的维护技术元数据，业务元数据可以很容易的附加在数据库表上。列名称、数据类型和长度以及基数的信息可以从数据库系统继承。表和列的业务描述通常作为元素加在 DBMS 数据目录中。
- 关系能力。实体之间的强制的数据完整性或参照完整性可以在关系数据库环境中很容易的维护。如果接收来自非关系型系统中的数据，在转换为维度模型之前，将数据集结在一个规范模型中就很有必要。
- 开放的资料库。一旦数据进入 DBMS，那么数据可以很容易的通过任何支持 SQL 的工具来访问（如果赋予了相应的权限）。进行质量保证测试和审计时，确保对数据的访问至关重要。
- DBA 支持。在很多企业环境中，DBA 小组仅对 DBMS 中的数据负责。数

据库外的数据，如文件系统通常不由 DBA 维护。如果集结区数据不在 DBMS 中，那么 ETL 小组必须承担空间分配、备份和恢复、归档以及安全等任务。

- SQL 接口。很多的时候我们需要写 SQL 来操作数据，按照正确的格式获得数据。大家都知道 SQL 是标准语言，易于书写且功能强大。在 IT 领域，恐怕 SQL 是被最广泛掌握的程序语言了。大多数的数据库系统都提供了强大的 SQL 函数，帮助用户节省手工编程的时间。例如，Oracle 提供 to_char()函数可以将任意类型的数据转换为不同格式的字符串。除了强制参照完整性检查，能够使用自带的 SQL 是在数据库环境中存储集结区数据的主要原因。

独立的 DBMS 工作表

如果决定在 DBMS 中存储集结区数据，在设计数据集结区模型时可以有不同的架构选择。在集结区设计表可能比设计事务处理系统或设计维度模型更加具有挑战性。别忘了，事务处理数据库为数据进入进行设计，维度模型为数据输出进行设计，而集结区设计则同时包含两者。因此，在集结区使用混合的数据架构是很正常的。

至于采用独立集结表的原因，可以用一句话来概括：最简单的就是最好的。之所以被称为“独立表”，是因为这些表与数据库中的其他表没有任何依赖性。在事务处理环境，这些表被称作孤表，因为它和模型中的其他表没有任何关系。因为独立集结表没有任何关联关系，因此独立表是在关系型数据库外建立存储的主要候选方式。

大多数时候，创建集结表目的是为了存储数据以便于能够使用 SQL 或脚本语言再次操作数据。很多情况下，尤其是小型数据仓库项目，独立表可以满足全部的数据集结区要求。

由于独立表不需要规范化，因此不能视作转储文件。转储文件通常任意的创建，无须考虑磁盘空间或者查询效率。独立文件或独立表的每个字段必须有主题和逻辑定义。多余的列会在任何独立表的设计中被忽略。对于数据库表，需要在所有的独立表上建立并实现一个合理的索引规划。由于访问独立表的进程只有

ETL 过程，这里没有必要建立在展示区中才用到的位图索引，位图索引是为最终用户工具和即席查询所用的。在 ETL 系统中，会更多地在单列或复合列使用 B 树索引。

三范式实体/关系模型

有一种观点认为，数据集结区或许应该是最终加载到数据仓库中的企业所有数据的中央资料库。然而，将数据集结区称为企业中央资料库是明显的用词不当，这会导致数据架构师认为这个区域必须完全的规范化。毕竟，哪一个称职的数据架构师会为企业数据留下为人诟病的冗余呢？在本章开始我们已经做了一个餐馆厨房的比喻，例如，考虑某种食物——比如说鱼——在能够上桌之前需要仔细的挑选、洗净、分割、切片并煎炒。现在假设餐馆还提供冰激凌，那么肯定不会像之前做鱼那样做冰激凌，冰激凌只需要挖出来就可以上桌。强制让所有的源系统使用相同的规范化过程——使之符合第三范式数据模型——就像要你按照作鱼的方法准备冰激凌一样得不合情理。

事实上，我们很少将数据集结区建成第三范式模型。在很多案例中，一个层系中的数据元素来自多个不同的数据源，具有不同的粒度，还包括很多来自非关系型数据源的外部数据。这种情况下，通常是在加载到维度数据模型之前，消除数据冗余和强制完整性。理想的处理方式是集中处理独立的“问题维表”，对它进行彻底检查以确保原先的脏数据已经被正确地清洗。记住规范化的主要结果是强制明确的多对一关系，除非需要人工检查模型的图形描述，否则，实体关系图的注释既不强求也无须解释。建模的时候应该具体情况具体考虑，只有在必要的时候才进行实体规范化。

不要假定数据集结区必须作规范化。记住本章开始所讲的设计 ETL 处理的两个目标：快速和可恢复。如果 ETL 过程中所作的集结既没有物理的数据操作，也不能加快速度或支持恢复，那么它就应该被移除。

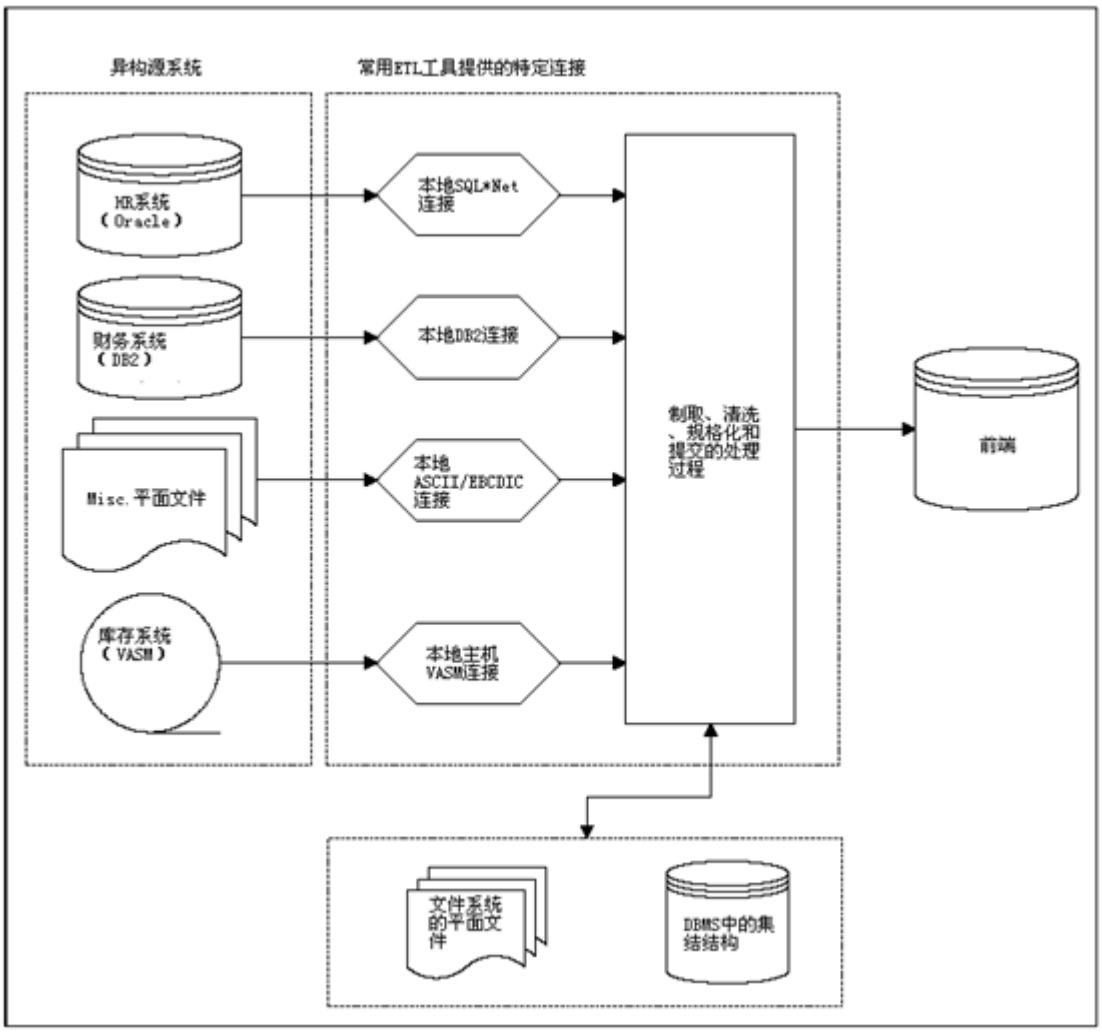
非关系数据源

通常，建立专门的集结区环境的一个重要的原因是为了集成非关系型数据。如果所有的数据都在一个 RDBMS 中，那么数据集成任务就没有什么挑战性了。集成异构数据源是 ETL 开发者经常要面对的挑战，尤其是在数据仓库扩展其数据

范围，包含进越来越多的主题域的时候。

在企业数据仓库项目中，很多数据源来自非关系型数据源或者没有关联关系的多个关系型数据源。非关系型数据源可能包括 COBOL Copy Book，VSAM 文件，平文件，电子数据表等等。

通常的经验会说，将所有的异构的数据源集成到一个 DBMS 中，但是这真的必要吗？ETL 工具在处理异构数据方面的强大能力使得将所有必须的数据存储在一个数据库中的需求变得不再那么重要。图 2.2 描述了一个平台无关的 ETL 工具如何将多个异构数据源从它们的源系统进行数据集成，然后迁移到数据仓库。注意 ETL 工具连接到物理的集结区数据库和外部文件，在需要的时候可以为中间的数据处理设置临时的数据落地。



图表 2.2　ETL 工具可以随意的使用集结区来集成异构的数据源

集成非关系数据源通常需要作一些完整性检查。数据完整性的保证不是没有代价的，通常需要在数据集结区中的实际的存储，并且需要对 ETL 过程进行客户化以保证业务规则的正确，而这些业务规则在源系统的关系型数据库中是自动维护的。关系意味着表之间有联系，通常为父子关系，这是由数据库来保证的。例如，如果一个订单表中有一个状态列，具体的值如果在一个独立的状态参照表中不存在就不能够录入。在这个场景中，状态表是订单表中对应列的父。状态表中的父不能随便地删除，除非所有的子已被删除，否则它的子就变成了孤儿记录。孤儿记录指的是任何没有父的子记录。同样，如果状态表中有外键引用到订单，那么任何的订单的主键也不能删除。孤儿记录的出现是参照完整性被破坏的标志。

非关系数据源不能保证参照完整性，非关系系统本质上是无关的表的集合。在很多遗留的事务系统中，父子关系只能通过前置应用来保证。不幸的是，经过多年运行，由于不可避免的脚本操作或者应用之外的数据操作，导致数据库系统中的任何数据完整性都已不能保证。可以肯定非关系数据源中多多少少都会有数据质量问题。

与事务系统不同，设计数据集结区的最佳实践是在 ETL 过程而不是在数据库中进行完整性检查。其中的区别在于，事务系统期望数据输入是正确的，并且人为的输入错误将导致错误抛出，提示重新输入正确的值。与此相反，ETL 过程则必须知道如何自动地处理数据异常。ETL 过程不能简单地拒绝所有的数据完整性错误，因为现在没有人会及时地重新输入正确的数据。我们需要为不同的数据质量错误场景定义不同的业务规则，并在 ETL 过程中实现这些规则。当错误的数据通过 ETL 流程时，有时你希望直接转换；有时不作修改直接加载；有时需要追加相关的代码或者追加一些对于影响及环境的描述，然后进行加载；或者如果数据不可接受，那就完全拒绝数据，并将其写入拒绝文件供进一步的调查。

不要过度使用拒绝文件！拒绝文件用于那些我们需要后续专门处理的转储数据。当记录进入拒绝文件后，除非在下一个主要加载步骤开始运行前完成对其处理，否则数据仓库和生产系统之间的同步就被破坏了。

基本的数据库参照完整性并不能完全处理上述的每一种情景。ETL 过程通常

都需要手工编码逻辑，来保证成功集成非关系数据源。

维度数据模型：从后台提交到前台的成果

维度数据结构是 ETL 过程的最终目标，这些表位于前台与后台之间。多数情况下，维度模型是把表传递给最终用户环境之前的最后一步物理集结步骤。

维度数据模型是目前为止大多数最终用户查询和分析的最流行的数据结构。它们创建简单，结构异常稳定，不随数据环境的变化而变化。它们能被最终用户直观地了解，是满足通常的关系型查询的最快的数据结构。维度模型也是构建所有形式的 OLAP 立方体的基础，因为 OLAP 立方体实际上是由专门的软件创建的维度模型。

本节仅仅简单的介绍维度模型中的主要的表类型。

工具箱系列中的另外一本书将详细地讨论维度模型，给出在不同业务环境中建模的指南和目标。我们假设在这本书中，你已经完全明白维度模型的重要性（甚至已经掌握了某本维度设计的技能！）。因此在这一节中，我们就直接介绍基础的维度模型的物理结构，而不再从业务角度阐述采用这种结构的理由。在本书的第 2 部分，我们将详尽地介绍所有已知的维度模型的变种， 并讨论 ETL 系统如何为这些结构提供数据。

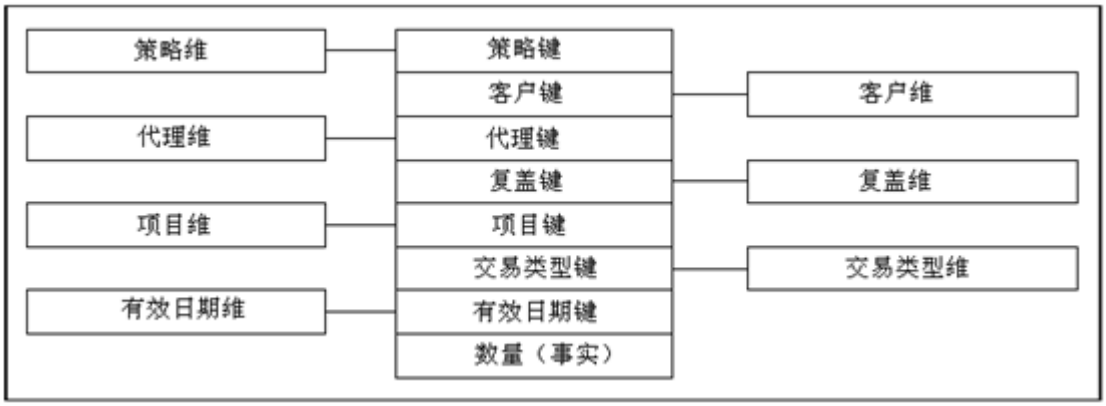
事实表

维度模型是围绕着度量过程建立的。一个度量指的是对先前未知值的观察。度量基本都是数值型的，而且大部分度量都可以随着时间重复，这构成了时间序列。

一次度量构建一条单独的事实表记录。相对的，一条事实表的记录对应一次特定的度量事件。很明显，观察到的度量将存储在事实表记录中，但我们同时还要将度量的上下文信息存储在同一条记录中。当然，尽管我们可以将这些上下文信息直接存储在事实表记录中，但更多的时候是通过创建一系列的维表将这些上下文的属性规范化的存储在到事实表之外，也就是构建一系列的维表，这组维表可以看作是“上下文簇”。

例如，如果某个度量是某次保费登记的数额，具体的相关信息包括某个保险

公司登记的某个保单、某个客户、某个代理、某个险种（比如碰撞伤害）、某个保险项目（比如汽车）、某个交易类型（比如确定保费）、在某个生效日期，那么与事实记录相关的维度就包括保单、客户、代理、险种、项目、交易类型和生效日期等等。图 2.3 描述了这个例子。



图表 2.3　保单交易的维度模型

事实表的粒度定义了构成事件表的记录的级别。在维度模型的规范中，粒度是在业务术语设计阶段而不是数据库术语设计阶段进行定义的。在上述保险的例子中，粒度应该是保险保单交易，在后续的设计过程中，当可用的维度确定之后，粒度通常以事件表的键的方式来描述。键的描述中将包含一些但通常不是全部的与事实表关联的维度的外键引用。我们假设保险事实表的键是“保单 X 交易类型 X 时间”。

维度模型的结构和内容仅仅取决于物理的度量过程本身。

维表

应该讲，维度模型的设计并不是只针对或者依赖某类特定的查询应用，事实上，它的设计目标是提供灵活的、适用于各种查询类型的、对称的框架。但是维度模型的设计还部分地依赖于设计人员的判断力。很多维度的设计可以直接参照原始的数据源结构，但是数据仓库小组还是能够添加来自其它数据源的维度，只要在度量事件发生的时候这些维度都是单值。比如，在上述保险的例子中，如果数据具备，那么就可以建立一个针对市场促销的维度。维度模型一个最大的优点是，在可以在度量事件的上下文支持某个维度的情况下灵活地增加维度。

类似的，最佳的维表设计应该包含对于维度实体的尽量详细的描述。所以在

上述保险模型中，客户维度应该有很多描述性的字段，我们称这些描述字段为维度属性。幸运的是，所有的维度设计都允许在整个数据仓库的生命周期内增量的加入维度属性。数据仓库架构师的职责是识别出这些需要增加的属性，并要求 ETL 小组将它们添加到物理模型中。

维度属性通常是文本型或者离散的数值型。维表应该只有一个主键，键值使用的是由 ETL 过程生成的无实际意义的整型值，这些值称作代理键。使用代理键的优点在工具箱系列的另外一本书中有介绍。在本书中，我们只是介绍如何创建代理键以及在很多重要情况下对代理键的管理。

每个维表中的主代理键应该与事实表中的相应的外键相匹配，当主-外键关系建立后，我们说表符合参照完整性。满足参照完整性是所有维度模型的基本要求。参照完整性管理失败，意味着一些事件表中的记录是孤儿记录，它们无法通过维度约束来访问。

原子事实表和聚合事实表

我们知道维度模型是支持用户查询的最佳的数据格式。有时我们会用到原子级别的某些元素，以实现数据在更高的级别的展示。无论如何，我们需要存储原子级数据以满足用户的某些特定约束的需求。通常情况下，业务人员不希望分析交易级的数据，因为每个维度的基数都是那样大，以致任何原子级的报表都会很多页，很难人工检查。但是，依然需要存储原子级的事实来产生用户需要的周期性快照。当用户需要原子级数据的时候，只是简单地将数据从集结区迁移到展示区就可以了。

在实践中，将集结区中的事实表进行分区是一个比较好的方法，因为原子数据的聚合通常是按照特定的时间周期进行的，比如月或者季。创建分区表减少了数据库的全表扫描，可以直接基于包含某个时间周期的数据分区创建聚合。分区同样减少了清理或归档旧数据的负担，使用表分区可以很简单的删除表中包含旧数据的部分。

集结区中的按照维度模型设计的表很多时候需要用来生成 OLAP 立方体。或者，也可以实现一种混合结构，在维度型 RDBMS 模型中存储数据量很大的原子数据层，原子层之上的日益增多的聚合结构按照 OLAP 立方体的方式进行存储。

有些 OLAP 系统提供了钻透的功能，在单一应用中可以从 OLAP 立方体下钻到最低级别的原子数据。

代理键映射表

代理键映射表示用来建立各个源系统的自然键到主数据仓库代理键之间的映射，映射表是维护数据仓库代理键的一种非常有效的方法。这些表结构紧凑，专门用于为高速处理。映射表中仅仅包含那些最近期要访问的代理键值及其对应的源系统中的自然键值。由于同一个维度可以有不同的源，因此映射表中要为每个源的自然键创建单独的列。

映射表无论是存储在数据库还是文件系统中都可以有同样的高效率。如果使用数据库，可以利用数据库顺序号生成器来创建代理键，如果索引使用恰当，键值的查找会非常得高效。

由于键映射表并没有分析价值，因此，不能将其建在数据仓库的展示层，也不能暴露给最终用户。

交叉参考：在第 5 章中会详细解释如何应用映射表来为维表创建代表键，并且在第 6 章中讨论应用映射表来生成事实表。

3.4 规划和设计标准

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

同企业环境中其它的数据库相比，数据集结区需要进行更多的管理和维护。有些集结区的管理像沙盒一样，开发人员可以随心所欲地创建、删除和修改表。自然，这种缺乏管理的环境下的故障诊断和影响分析要比正常的花更多的时间，导致项目费用的增加。

数据集结区必须是一个严格控制的环境。只有数据架构师才可以在集结区中设计或修改表。所有的物理改变都应该是数据库 DBA 来完成。而且，如果有一个开发人员需要某张表，那么很大的可能是另一个开发人员也会使用它。一个忠告是：如果你创建了一张表，一定要想到，由于某种原因这张表很可能会被其他

人以另外一种方式使用。人们，尤其是开发人员，在重新使用已有资源时经常会非常的有创意。

影响分析

影响分析的作用在于检查与对象（这里指的是表或者列）关联的元数据，并判断对象的变化对其内容和结构有何影响。对数据集结区对象的更改可能会破坏数据仓库的加载流程。允许任意修改数据集结区对象会对项目的成功造成危害。

一旦在数据集结区中创建了表，在做任何修改之前，都必须进行影响分析。很多 ETL 工具厂商提供影响分析功能，但是这些功能经常仅仅在 ETL 产品的 POC 阶段进行简单的评估，原因是它是个后台功能，直到数据仓库建立运行或者发生变更的时候才真正的变得重要起来。

影响分析，作为是 ETL 的一项功能，是一个非常繁重的职责，因为源系统和目标数据仓库可能不断地在改变，但是只有 ETL 过程知道这些分散的元素是怎么连接的。在数据仓库依赖的任何系统发生变化的时候都需要进行相应的影响分析，能否顺利地完成影响分析，ETL 项目经理、源系统 DBA 以及数据仓库建模小组之间的沟通至关重要。

元数据捕获

我们将在第 9 章对元数据进行深入讨论，但是事先必须了解在设计数据集结区的时候将会处理哪些类型的元数据。根据上下文的不同，元数据有不同的含义。元数据用于描述或支持其他的数据元素，涉及到组成数据仓库的每个组件。在数据仓库的范畴里，数据集结区不是元数据资料库。然而，和数据集结区相关联的很多元数据元素却对数据仓库非常有价值，必须展示给最终用户。

设计集结数据库的时候使用数据建模工具可以可视化地展示元数据，数据建模工具在它自己的资料库中存储这些可用的元数据。另外使用 ETL 工具会生成关于过程的元数据，并能够对其中的所有的转换过程进行展示。从集结区衍生出来的元数据类型包括：

- 数据谱系：所有数据仓库元数据库中最有趣的元数据可能就是数据谱系，或者称为逻辑数据映射，阐述了数据元素从原始数据源到最终数据仓库

目标之间是如何转换的。

- 业务定义：数据集结区中创建的所有表都是从业务定义中衍生出来的。业务定义可以从很多地方获得，包括数据建模工具、ETL 工具、数据库自身或者电子表格和 Word 文档。无论如何，需要使用在数据仓库展示层上获取业务定义来维持其一致性。
- 技术定义：尤其对于数据集结区，技术定义要比业务定义更加的普遍。要记住，如果没有文档记录，那么就意味着技术定义不存在！如果数据集结区中表的技术定义没有详细的文档，那么这张表将可能被一次次的重建，会在数据集结区中产生大量的数据重复，导致数据爆炸。技术定义应该描述数据元素的所有物理属性，包括结构、格式和位置。对集结区中所有表进行技术元数据文档化记录可以将不确定性降至最低，并提高重用性。
- 过程元数据：数据集结区表的加载过程的统计必须和数据仓库表加载的统计一起记录。尽管数据集结区加载过程的信息不需要展示给最终用户，但是 ETL 小组需要知道每个表中加载了多少记录，每个过程成功或失败的统计结果。而数据刷新频度方面的信息对 ETL 管理员和最终用户都是有用的。

数据集结区中所有的表和文件都应该由 ETL 架构师来设计。元数据必须很好地记录成文档。数据建模工具提供的元数据捕获功能将会减少文档的工作量。在设计集结区表的时候应该使用数据建模工具来捕获像相应的元数据，要记住从工具中获得的面向结构的元数据大约会占到 25%，另外 25%的元数据描述数据清理，而过程元数据将占 50%，请确认你选用的 ETL 工具能够提供所需要的统计信息。至少要提供每个过程的插入行数、更新行数、删除和拒绝行数等信息。另外，过程的开始时间、结束时间、持续时间应该不需要编码就可以得到。

命名规则

ETL 小组不应该替数据仓库小组开发命名规则，相反，他们应该使用数据仓库架构师所定义的命名规则，最佳实践的建议是集结区使用与数据仓库其他部分相同的命名规则。但是，因为数据集结区中有很多数据元素是不出现在展示层中

的，因此很有可能没有针对它们的命名标准。这个时候，需要与数据仓库小组和 DBA 组一同工作，在命名规则中增加针对数据集结区特有的命名规则。

很多 ETL 工具和数据建模工具坚持展示很长的字母顺序的表名列表。你需要非常仔细地对你的表名进行分组，使其在字母排序时在一起。

审计数据转换步骤

在复杂的 ETL 系统中，数据转换体现了复杂的业务规则。如果需要对系统中的全部数据转换过程进行全程审计，至少要包含下列述列表中的任务：

- 用代理键代替自然键
- 对实体进行组合和剔重
- 对维度的常用实体进行规格化
- 对计算进行标准化，创建规格化的 KPI
- 在数据清洗过程中对数据进行更正和强制转换

在一个多变的环境中，源系统的数据不断的在改变，数据仓库要求要有能力证明数据的准确性。这种情况下如何处理。ETL 过程必须维护在数据清理阶段之前的数据快照。

当数据被 ETL 过程修改（清洗）过，在操作前的数据为了审计的需要必须保留。此外，所有数据清洗逻辑的元数据必须不通过代码就可以访问。原始的源数据、数据清洗元数据和最终的维度数据必须在一同发送来支持对数据清洗转换的置疑。

在数据集结区存储抽取数据的快照可以满足审计的需要。处理前后的数据快照，数据清洗逻辑的元数据共同描述了在数据仓库中数据如何的演变，提供了数据质量的可信度。

3.5 总结

在本章中，我们回顾了在 ETL 系统中需要使用的主要的数据结构。我们开始讨论了为了临时存储和永久存储而在不同位置集结数据的例子。你需要兼顾技术

和投资的考虑。在这些案例中，你不仅仅需要存储数据，还需要记录数据创建的规则；如果这些信息不全，你需要证明数据没有被篡改。

一个成熟的 ETL 环境会混合文本文件、独立关系表、full-blown 规格化模型、以及其它可能的文件结构，特别是 XML 文档。但是无论何种情况，我们需要你记住 ETL 系统的核心是最终的结果：数据结构的建立目标是最终用户易于使用。当然我们建议这些表使用维度模型，使用事件表和维表。OLAP 立方体通常是通过维度模型创建的。

最后我们讨论了一些最佳实践，包括使用一致的设计标准，在你对表进行修改的时候，对你设计的表执行系统影响分析，另外，确保你在 ETL 系统的每个点上捕获了元数据。一个大的元数据目录中包含了 25%的表结构元数据，25%的数据清洗结果以及 50%的过程结果。元数据将在第 4 章和第 9 章进一步讨论。

现在我们已经组织了我们所有的工作数据的结构，在第三章中我们将深入探索组成数据仓库的源数据结构。

第二部分　数据流

4 抽取

一旦开始了数据仓库项目，你就会很快意识到，集成整个企业中所有不同的系统来得到数据仓库可用的状态是一个真正的挑战。没有数据，数据仓库就是无用的。集成的第一步就是成功地从主要的源系统抽取数据。

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

本书的其它章节主要集中在转换和加载数据到数据仓库，本章的重点是为项目建立访问需要的源系统的接口。每一数据源都有不同的特点需要分别管理，以便为 ETL 过程有效的抽取数据。

随着企业的发展，需要建设或者继承多个不同的计算机系统来帮助公司运作他们的业务：销售系统、库存管理、生产控制、总帐系统——这个列表还在不断增长。更糟糕的是，不但这些系统是分离的且产生于不同的时间，而且他们经常是在逻辑上和物理上不兼容。ETL 过程需要有效集成不同的系统：

- 数据管理系统
- 操作系统
- 硬件
- 通信协议

在开始创建抽取系统之前，需要一份逻辑数据映射，它描述了那些提交到前台的表中原始源字段和最终目标字段之间的关系。该文档将贯穿 ETL 系统的始终。我们将在本章第一部分介绍如何创建逻辑数据映射。本章第 2 部分列举了可能经常遇到的各种源系统。我们将深入研究每一种源系统，以便选择正确的处理方法。

本章结束部分介绍了变化数据和删除数据的捕获这一主题，15 年前我们还认

为数据仓库是永恒不变的：它是一个一次性写成的数据大库。随着这些年的丰富经验的积累，我们现在知道数据仓库经常需要修改、纠错和更新。本章的变化数据捕获抽取技术只是这一复杂问题解决的第一步，在后续的提交和操作章节中，仍将继续这一变化数据捕获主题。

4.1 第 1 部分：逻辑数据映射

在物理实施之前如果没有仔细地规划结构将会是一场灾难。正如任何其它形式的建筑物一样，在敲击第一个钉子前必须拥有一个蓝图。在开始开发一个单一的 ETL 处理前，确保你拥有合适的文档，以便该处理与已有的 ETL 策略、过程和标准在逻辑上和物理上保持一致。

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

逻辑数据映射描述了 ETL 系统中起点和终点之间的关系。

物理之前设计逻辑

直接跳到物理数据映射会浪费宝贵的时间，并将无法文档跟踪。这一节描述如何开发逻辑的 ETL 过程并使用它来制订出物理 ETL 执行流程。在开始任何物理 ETL 开发之前，确定以下的步骤已经完成：

1. 有一个规划。这个 ETL 过程必须用逻辑的和文档化的形式表示出来。逻辑数据映射由数据仓库架构师提供，并且它是为 ETL 团队创建物理 ETL 作业而制定的。该文档有时会作为数据流报告。逻辑数据映射是元数据的基础，随后将提交给测试员来保证数据质量，并且最终将提交给最终用户来详细描述在源系统和数据仓库之间到底做了些什么。

2. 确定候选的数据源。从最高级别的业务对象出发，确定你认为将支持业务人员进行决策的可能的候选数据源。并且确定这些源中你认为对最终用户的数据很重要的特定的数据元素，这些数据元素将成为接下来的数据评估步骤的输入。

3. 使用数据评估工具分析源系统。源系统中的数据必须在数据质量，完整性和适合使用方面进行仔细检查。根据于你的组织结构，数据质量可能是或者不是

由 ETL 小组负责，但是这个数据评估步骤必须由对最终使用数据仓库的决策者的需求有一定了解的人来负责。每一个源系统的数据都必须进行分析。任何监测到的异常数据都必须用文档记录下来，对任何进入数据仓库的数据都必须按照适当的业务规则进行修正是最好的选择。你必须要一直关注项目在这个阶段就停下来的可能性！如果数据不能支持业务目标，这个时候就要发现。在第四章中将更多的探讨数据评估。

4. 接收数据线和业务规则的遍历。一旦源数据完成数据评估步骤的质量保证并且理解了最终的目标数据模型，数据仓库架构工程师和业务分析师必须和 ETL 架构工程师及开发者一起完成为抽取，转换和加载的数据仓库的主题域的整个数据线和业务规则，并且最好他们理解这些规则。完全的理解这些数据线和业务规则是几乎不可能的，因为那样 ETL 小组必须遇到所有的数据实际问题，但这一步骤的目标是提交尽可能多的知识给 ETL 小组。数据评估步骤必须创建 ETL 特有的业务规则的两个子类：

4a. 在数据清洗步骤中需要进行改造的数据；

4b. 对分离的数据源的维度实体和可度量的数字事实强制一致性来获得标准的结构；

5. 充分理解数据仓库数据模型。ETL 小组必须充分理解数据仓库的物理数据模型。这种理解包括维度模型的概念，仅仅理解基本的表到表之间的映射是不够的，开发小组必须对如何使维度，事实及其他维度模型中的特定表一起发挥作用来实施一个成功的 ETL 解决方案有好的理解。切记 ETL 系统的主要目标是用最有效的方式将数据送给最终用户工具。

6. 验证计算和公式的有效性。与最终用户一起校验任何在数据链中任何指定的计算。这个规则来源于纽约市建筑行业的谚语“测两次，切一次”。正如你不想得到一座用错误尺寸的材料造成的摩天大楼，你同样不希望在数据仓库中部署不正确的度量指标。在 ETL 过程中花时间以错误的法法编码之前，确保你的计算公式都是正确的将是非常有用的。

4.2 逻辑数据映射内部

在深入你将遇到的不同的数据源细节之前，我们需要研究逻辑数据映射文档的实际设计。该文档包括整个企业针对数据仓库源系统的数据定义，目标数据仓库数据模型，以及从原有格式到最终目的转换所需要的完全数据操作。

逻辑数据映射的组成

逻辑数据映射（见图 3.1）通常用一个表或者电子表格格式来表示，它包括以下特定的组成部分：

- 目标表名称：数据仓库中出现的物理表名称；目标列名称：数据仓库表中的列名称；
- 表类型：表示这个表是事实表，维表或者子维表（支节）
- SCD（缓慢变化维）类型：对维表，这个部分表示是类型 1，类型 2 或者类型 3 的缓慢变化维。这个指标对维表中的不同的列可以是不同的。
- 比如在客户维中，名字可能属于类型 2（保留历史信息），而姓可能属于类型 1（覆盖）。这些 SCD 类型将在第五章展开详细探讨。
- 源数据库：源数据所在的数据库实例的名称。这里通常是指连接数据库所需的连接字符串。如果出现在文件系统中，它也可以是一个文件的名称。这时，还需要包含这个文件的路径。
- 源表名称：源数据所在的源表的名称。很多时候需要多个表。这时，只需将生成目标数据仓库相关表的所有表简单列出即可。
- 源列名称：生成目标所需的相关列。简单的列出装载目标列需要的所有列。源列之间的关联在转换部分记录。
- 转换：源数据与期望的目标格式对应所需的详细操作。这个部分通常用 SQL 或者伪代码来编写。

逻辑数据映射中的列有时是组合的。比如，源数据库，表名称和列名称可能被组合在一个源列中。这个组合列的信息可能用原点来分隔信息，如 ORDERS.STATUS.STATUS_CODE。如果不考虑格式，这个逻辑数据映射文档的内容

已经把进行有效的规划 ETL 过程的所有的关键的信息都提供了。

逻辑数据映射中的某些部分看起来很简单并且很直接。然而，当仔细研究的时候，该文档就会揭示许多 ETL 小组可能忽略的隐藏的需求。这个文档的主要的目标是为 ETL 开发者提供一个清晰的蓝图，它精确的说明可以从 ETL 过程获得什么。这个表必须清晰地描述在转换过程中包含的动作流程，不能有任何疑问的地方。

看一下图 3.1。

Target					Source				Transformation
Table Name	Column Name	Data Type	Table Type	SCD Type	Database Name	Table Name	Column Name	Data Type	
EMPLOYEE_DIM	EMPLOYEE_KEY	NUMBER	Dimension	1				NUMBER	Surrogate key.
EMPLOYEE_DIM	EMPLOYEE_ID	NUMBER	Dimension	1	HR_SYS	EMPLOYEES	EMPLOYEE_ID	NUMBER	Natural Key for employee in HR system
EMPLOYEE_DIM	BIRTH_COUNTRY_NAME	VARCHAR2(75)	Dimension	1	HR_SYS	COUNTRIES	NAME	VARCHAR2(75)	select c.name from employees e, states s, countries c where e.state_id = s.state_id and s.country_id = c.country
EMPLOYEE_DIM	BIRTH_STATE	VARCHAR2(75)	Dimension	1	HR_SYS	STATES	DESCRIPTION	VARCHAR2(255)	select s.description from employees e, states s where e.state_id = s.state_id
EMPLOYEE_DIM	DISPLAY_NAME	VARCHAR2(75)	Dimension	1	HR_SYS	EMPLOYEES	FIRST_NAME	VARCHAR2(75)	select initcap(salutation) ||' '||initcap(first_name) ||' '|| initcap(last_name) from employee
EMPLOYEE_DIM	BIRTH_DATE	DATE	Dimension	1	HR_SYS	EMPLOYEES	DOB	DATE	trunc(DOB)
EMPLOYEE_DIM	SALUTATION	VARCHAR2(12)	Dimension	1	HR_SYS	EMPLOYEES	SALUTATION	VARCHAR2(12)	initcap(salutation)
EMPLOYEE_DIM	FIRST_NAME	VARCHAR2(30)	Dimension	1	HR_SYS	EMPLOYEES	FIRST_NAME	VARCHAR2(30)	initcap(first_name)
EMPLOYEE_DIM	LAST_NAME	VARCHAR2(30)	Dimension	1	HR_SYS	EMPLOYEES	LAST_NAME	VARCHAR2(30)	initcap(last_name)
EMPLOYEE_DIM	MARITAL_STATUS	VARCHAR2(12)	Dimension	2	HR_SYS	MARITAL_STATUS	DESCRIPTION	VARCHAR2(12)	select nvl(m.name,'Unknown') from employee e marital_status m where e.marital_status_id = m.marital_status_id
EMPLOYEE_DIM	DIVERSITY_CATEGORY	VARCHAR2(30)	Dimension	1	HR_SYS	EMPLOYEES	EEO_CLASS	VARCHAR2(30)	decode(eeo_class,null, 'Not Stated', decode(eeo_class,'N', 'Not Stated',eeo_class))
EMPLOYEE_DIM	GENDER	VARCHAR2(12)	Dimension	1	HR_SYS	EMPLOYEES	SEX	VARCHAR2(12)	nvl(sex, 'Unknown')
EMPLOYEE_DIM	EMPLOYEE_STATUS	VARCHAR2(24)	Dimension	1	HR_SYS	EMPLOYEES	STATUS	VARCHAR2(24)	select es.name from employee e employee_status es where e.employee_status_id = m.employee_status_id
EMPLOYEE_DIM	POSITION_CODE	VARCHAR2(12)	Dimension	2	HR_SYS	POSITIONS	POSITION_CODE	VARCHAR2(12)	select p.code from employees e, positions p where p.position_id = e.position_id
EMPLOYEE_DIM	POSITION_CATEGORY	VARCHAR2(30)	Dimension	2	HR_SYS	POSITIONS	POSITION_CATEGORY	VARCHAR2(30)	select p.category from employees e, positions p where p.position_id = e.position_id
EMPLOYEE_DIM	HIRE_DATE	DATE	Dimension	1	HR_SYS	EMPLOYEES	DATE_HIRED	DATE	trunc(date_hired)
EMPLOYEE_DIM	DEPARTMENT_CODE	VARCHAR2(12)	Dimension	2	HR_SYS	DEPARTMENTS	CODE	VARCHAR2(12)	select d.code from employee p, employee_department pd, departments d where p.employee_id= pd.employee_id and pd.department_id = d.department_id
EMPLOYEE_DIM	DEPARTMENT_NAME	VARCHAR2(75)	Dimension	2	HR_SYS	DEPARTMENTS	DESCRIPTION	VARCHAR2(75)	select d.description from employee p, employee_department pd, departments d where p.employee_id= pd.employee_id and pd.department_id = d.department_id
EMPLOYEE_DIM	PART_TIME_FLAG	VARCHAR2(1)	Dimension	1	HR_SYS	EMPLOYEES	PERCENTAGE	VARCHAR2(1)	select decode(sign(percentage-100),-1,'Y','N') from employee
EMPLOYEE_CONTRACT_FACT	EFFECTIVE_DATE_KEY	NUMBER	Fact	N/A	DW_PROD, HR_SYS	DATE_DIM, EMPLOYEE_CONTRACT	DATE_KEY	NUMBER	where employee_contract.eff_date = dw_prod.date_dim.cal_date
EMPLOYEE_CONTRACT_FACT	END_DATE_KEY	NUMBER	Fact	N/A	DW_PROD, HR_SYS	DATE_DIM, EMPLOYEE_CONTRACT	DATE_KEY	NUMBER	where employee_contract.end_date = dw_prod.date_dim.cal_date
EMPLOYEE_CONTRACT_FACT	CURRENCY_KEY	NUMBER	Fact	N/A	DW_PROD, HR_SYS	CURRENCY_DIM, EMPLOYEE_CONTRACT	CURRENCY_KEY	NUMBER	where employee_contract.currency_code = dw_prod.currency_dim.currency_code
EMPLOYEE_CONTRACT_FACT	RATE_TYPE_KEY	NUMBER	Fact	N/A	DW_PROD, HR_SYS	RATE_TYPE_DIM, EMPLOYEE_CONTRACT	RATE_TYPE_KEY	NUMBER	where employee_contract.rate_type_id = dw_prod.rate_type_dim.rate_type_id
EMPLOYEE_CONTRACT_FACT	PROJECT_KEY	NUMBER	Fact	N/A	DW_PROD, HR_SYS	PROJECT_DIM, EMPLOYEE_CONTRACT	PROJECT_KEY	NUMBER	where employee_contrac.project_code = dw_prod.project_dim.project_code
EMPLOYEE_CONTRACT_FACT	EMPLOYEE_ROLE_KEY	NUMBER	Fact	N/A	DW_PROD, HR_SYS	EMPLOYEE_ROLE_DIM, EMPLOYEE_CONTRACT	EMPLOYEE_ROLE_KEY	NUMBER	where employee_contract.employee_role = employee_role_dim.contractor_role_name
EMPLOYEE_CONTRACT_FACT	CONTRACT_TYPE_KEY	NUMBER	Fact	N/A	DW_PROD, HR_SYS	CONTRACT_TYPE_DIM, EMPLOYEE_CONTRACT	CONTRACT_TYPE_KEY, CONTRACT_TYPE_ID	NUMBER	where dw_prod.employee_contract.contract_type = contract_type_dim.contract_type_id
EMPLOYEE_CONTRACT_FACT	CONTRACT_NUMBER	NUMBER	Fact	N/A	DW_PROD, HR_SYS	EMPLOYEE_CONTRACT	CONTRACT_NUMBER	NUMBER	Degenerate Dimension
EMPLOYEE_CONTRACT_FACT	RATE_AMOUNT_LOCAL	NUMBER	Fact	N/A	DW_PROD, HR_SYS	EMPLOYEE_CONTRACT	AMOUNT	NUMBER	sum(amount)
EMPLOYEE_CONTRACT_FACT	RATE_AMOUNT_USD	NUMBER	Fact	N/A	DW_PROD, HR_SYS	EMPLOYEE_CONTRACT	AMOUNT	NUMBER	select ec.amount * avg(cc.conversion_rate) from employee_contract ec, currency_conversion cc where ec.currency = cc.from_currency and cc.effective_Date between ec.effective_date and ec.end_date group by ec.amount

图表 3.1　逻辑数据映射

仔细观察这个图，你可能会注意到一些注意事项，如果它们没有被关注，可能会引起许多故障诊断和调试的工作甚至最终延误这个项目。比如，你可能注意到 STATE 在源和目标中的数据类型已经从 255 字符转换为 75 字符。尽管数据分析文档可以支持数据范围的减小，但将来创建任何大于 75 字符的值时，就有可能丢失这些数据。而且，一些 ETL 工具实际上可能终止或者使整个包含这些数据溢出错误的过程失败。请注意 STATE 的转换说明没有明确定义这一数据转换—转换是隐含的。根据定义，不应该有一个明确定义的账户是隐含转换的。隐含转换

在破坏你的处理流程方面是常见和臭名昭著的。为了避免不幸的发生，ETL 小组必须假定对这些类型的隐含数据转换都负责任地进行了明确的处理。

ETL 工具包通常持续跟踪这些隐含的数据转换并提供报告来标识任何的这种类型的转换。

表类型给了我们数据加载过程执行的次序—先是维表，然后是事实表。

与表类型一起，加载维表过程 SCD 类型显得至关重要。正如我们在这一章的前面所解释的，表结构的本身无法揭示缓慢变化维的策略是什么。错误地解释缓慢变化维策略将引起数周开发时间的浪费。在开始开发加载过程之前，需要清楚地了解哪些列需要保留历史信息以及如何获取历史信息所需的策略。这个列的值随着时间的推移可能发生改变。通常在单元测试的时候，当你第一次挑选用户来观察数据仓库中的数据时，他们看到了不想得到的结果。即使数据建模者极其努力的尝试，SCD 的概念还是很难向用户表达，并且一旦他们开始进行维度的加载时，他们总是想要与你的 SCD 策略背道而驰。这种需求是很常见的，这需要数据仓库项目经理来处理并改变管理流程。

映射中的转换是这个流程中的小阶段，这里是技术性好的开发者首先关注的。但你必须迫使自己不要只集中精力在编码上，在进入转换之前要仔细检查整个映射。这个转换可能包含了整个解决方案或者根本就什么都不是。最常见的，转换可能用 SQL 来表示。SQL 可能是也可能不是完整的语句。通常，它是那些不能在映射用其他元素表达的代码段，比如 SQL 中的 WHERE 子句。在其他时候，转换也可能是一种方法，没有特殊的 SQL，只有普通的英语说明，比如从一个平面文件进行预加载，或者基于数据库之外的标准进行加载转换，或者拒绝已知的异常数据到一个拒绝文件中的指导性说明。如果转换是空的，这说明映射是直接从源到目标的，没有任何转换的需求。

以上是逻辑数据映射的全部内容，在开始任何实际的编码开始之前和 ETL 开发者一起对文档做一个全面的走查。

使用工具设计逻辑数据映射

一些 ETL 和数据建模工具能直接抓取逻辑数据映射信息。人们很自然的想要在这些工具中直接表示数据映射。输入信息到工具中使得我们可以共享元数据，

这是不错的选择。但在写信息的时候，并没有说明与逻辑数据映射相关的适当的数据元素的标准。在各种工具中可用的实际元素差别很大。正如数据仓库中的元数据标准是成熟的一样，对逻辑数据映射中的元素定义也应该建立一个标准。建立元数据标准将使得这些工具对这一目的变得更加一致和可用。你要调查你当前的工具包对存储逻辑数据映射来说是否可用并充分利用已有的任何功能。然而，如果你的工具没有实现所有你需要的元素，你将不得不把逻辑数据映射放在不同的地方，这使你的维护工作变成一件非常麻烦的事件。请密切关注各种产品在这方面所做的改进。

4.3 创建逻辑数据映射

数据仓库的成功主要来源于这样的情形，所有的数据放在一个逻辑位置，使用户可以执行交叉功能分析。在后台，ETL 小组无缝的整合和转换分散的无组织的数据并使它看起来好像从一开始就在一起的一样。数据仓库成功的主要标准之一是它存储的数据是干净的和一致的。统一的数据存储需要对每一个源系统有相当深入地了解。对数据源中的数据甚至源系统本身理解的重要性常常在 ETL 的项目规划阶段被忽略和低估。在源系统得到确认和分析之前完整的逻辑数据映射是不存在的。源系统分析通常分为两个主要阶段：

- 数据发现阶段；
- 异常检测阶段；

数据发现阶段

一旦理解了目标需要做成什么样子后，就要确认和检查数据源了。部分或者全部的源系统可能在数据建模期间发生很大变化，但并不需要过多地考虑这些。通常，只有主要的源系统在数据建模期间被确认出来。要知道数据建模工程师的主要目标是建立一个数据模型，这一点很重要。任何来自数据建模期间的逻辑数据映射都只是一个副产品----一个起点。况且，数据建模工程师花大部分的时间和最终用户在一起，所以在逻辑数据映射中定义的源系统可能不是真的初始的或者优化的源—记录系统。这需要 ETL 小组更深入到数据的需求中，确定每一个需要加载到数据仓库中的源系统、表和属性。为每一元素确定适当的源或者记录系

统是一个挑战，必须仔细评估。完整的分析可以减少在 ETL 过程中由于使用错误的源而导致的数周时间的工期延误。

收集和文档化源系统

源系统通常由各种文档组成，包括会谈纪录，报表，以及数据建模人员的逻辑数据映射。ETL 小组通常需要更多的调研。和团队系统以及业务分析师一起工作来找到适当的源系统。在大型的组织架构中，你必须问“还有谁在使用这些数据？”这样的问题，并找出每一个用户组的数据源。一般的组织架构都拥有数不清的不同系统。ETL 小组有责任保持对这些系统的跟踪来发现和研究它们，并确定作为一个数据仓库的数据源是否有用处。

保持跟踪源系统

一旦源系统被确定，就要说明这些系统的意义以及谁对他们负责。图 3.2 就是为这个目的建立的图表。这个图表，也就是源系统跟踪报告，使得我们不用总是麻烦系统管理员或者业务管理者。如果走运的话，数据建模工程师将开始制作这个列表。不管谁定的初稿，这个列表的维护应该是 ETL 小组和数据建模小组共同努力来完成。如果在分析过程中认为一个源系统不适合作为数据仓库的源，保留它在这个列表中，并说明不使用它的原因；可能在后续阶段会用得到。

Subject Area	Initiative Name	Business Name	Priority	Department/ Business Use	Business Owner	Technical Owner	DBMS	Platform	Daily Data Count	Data Size (GB)	Transactions/ Day	Comments
Human Resources	PeopleSoft	HR Information	1	Human Resource Mgmt	Daniel Burns	Margaret Kerlin	Oracle	AIX on RS6000	850	3	8,000	Everything revolves around HR
Finance	Oracle Financials	Oracle Financials	2	General Accounting	Mabel Kerr	Edward Byiul	Oracle	AIX on S80	400	80	20,000	Driven by audit; departments fussing & doing data
Materials Management	WMS	Warehouse Management System	3	Manufacturing Planning & Control	Annabel Gluts	Christina Haym	Oracle	AIX on S80	350	20	5,000	Initiative to cut inventory of slow movers. Need analyst at support
Operations	BOMM	Bill of Materials Management	4	Production Planning	Lucy Kent	George Ormsmith	Oracle	AIX on RS/6000	40	8	700	Need analysis of "kits". Want to build kits in-house
Marketing	MCM	Marketing & Campaign Management	5	Marketing	Elizabeth Impler	Brian Birdsburg	SQL Server	W2K on Compaq	50	350	Various	Needs Data Mining Capabilities
Human Resources	Positions Website	Positions Website	5	Human Resource Mgmt	Florence Ipter	Andrew Acenthal	SQL Server	W2K on Compaq	50	1	10,000	Manager is new, eager for DW
Purchasing	PAS	Purchasing & Acquisition System	6	Purchasing Department	Guy Crowell	Rybil Mal	SQL Server	NT on Compaq	75	4	1,500	
Customer Service	CSS	Customer Service System	7	Customer Care	Bernard Beales	Arthur Alvarez	Netiss/ Domino	NT on Compaq	275	4	1,500	Plans to move application to Oracle next year
Operations	QCS	Quality Control System	8	Quality Control	Thomas Ryan	Elias Cloye	SQL Server	NT on Compaq	450	12	5,000	Needs failure rate analytics by product, by vendor, by year
Sales	SFA	Sales Force Automation	9	Sales	Anthony Arlen	Emma Inglorm	DB2	AS/400	120	10	2,000	Politically challenging. Interim reporting system in place

Subject Area	Interface Name	Business Name	Priority	Department/ Business Use	Business Owner	Technical Owner	DBMS	Platform	Daily User Count	DB Size (GB)	Transactions/ Day	Comments
Human Resources	PeopleSoft	HR Information	1	Human Resource Mgmt	Daniel Burnis	Margaret Kartin	Oracle	AIX on RS/6000	850	3	8,000	Everything revolves around HR
Finance	Oracle Financials	Oracle Financials	2	General Accounting	Mabel Karr	Edmund Bykel	Oracle	AIX on S80	400	60	20,000	Driven sponsor ; Department likes slicing & dicing data
Materials Management	WMS	Warehouse Management System	3	Manufacturing Panning & Control	Annabel Giutu	Christina Hayim	Oracle	AIX on S80	350	200	5,000	Initiative to cut Inventory of slow movers; Need analytical support.
Operations	BOMM	Bill of Materials Management	4	Production Planning	Lucy Kard	George Cimemblis	Oracle	AIX on RS/6000	60	8	TBD	Need analysis of "kits", Want to build kits in-house
Marketing	IMCM	InternalMarketing & Campaign Management	5	Marketing	Elizabeth Impzar	Brian Bridalurg	SQL Server	W2K on Compaq	50	350	Varies	Needs Data Mining Capabilities
Human Resources	Positions Website	Positions Website	5	Human Resource Mgmt	Florence Iyzar	Andrew Aoanthal	SQL Server	W2k on Compaq	50	1	10,000	Manager is new, eager for DW
Purchaing	PAS	Purchaing & Acquision System	6	Purchaing Department	Guy Croendil	Sybil Mai	SQL Server	NT on Compaq	75	4	1,500	
Customer Service	CSS	Customer Service System	7	Customer Care	Bernard Beais	Arthur Aticcio	Notes/ Domino	NT on Compaq	275	6	1,500	Plans to move application to Oracle next year.
Operations	QCS	Quality Control System	8	Quality Control	Thomas Fryar	Elias Caye	SQL Server	NT on Compaq	450	12	500	Needs failure ratio analytics by product, by vendor, by year
Sales	SFA	Sales Force Automation	9	Sales	Anthony Aran	Emma Inalonm	DB2	AS/400	1200	10	2,500	Politically challenging; Interim reporting system in place

图表 3.2　源系统跟踪报告

源系统跟踪报告也可以作为数据仓库的后续阶段的概要。如果有 20 个源系统被确定在这个列表中，并且阶段 1 包括两个或者三个系统，则规划这个项目要很长很长的时间。

- 主题域：系统的数据集市常用的名称。
- 接口名称：源系统支持的交易应用系统的名称。
- 业务名称：业务用户通常使用的系统名称。
- 优先级：用于确定将来阶段的位置顺序，优先级通常在数据仓库总线矩阵完成后设置。
- 部门/业务用法：使用数据库的主要部门，比如：会计、人力资源等等。如果有许多部门使用这个应用，指出这个业务用法，比如：库存控制、客户跟踪等等。
- 业务所有者：与使用应用或者数据库的相关问题的联系人或者小组。一般来说该人或者该小组是这个主题域的数据负责人。
- 技术所有者：一般是负责维护数据库的 DBA 或者 IT 项目经理。
- DBMS: 源数据库管理系统的名称。大多数是关系型数据库，比如 ORACLE, DB2, 或者 SYBASE。也可能是非关系型数据存储如 LOTUS NOTES 或者 VSAM。
- 生产服务器/OS：这个列包含数据库运行的服务器的物理名称。也包括操

作系统。当为 ETL 设计操作系统级的脚本时，就需要这个列。比如，不能在 NT 上使用 UNIX SHELL 脚本。

- #日常用户：让你知道组织中有多少操作型人员使用这些数据。这个数字并不是潜在的数据仓库的最终用户数。
- DB 大小：DBA 可以提供这个信息。知道源数据的记录数有助于你确定 ETL 的优先级和将要付出的工作量。一般来说，越大的数据库的优先级别越高，因为当在交易系统中查寻大表或者几个表关联查询时性能常常变得很差。
- DB 复杂度：系统中的表和视图对象的数目。
- #每日交易数：估计出该数字使你对增量加载过程所需要的容量有个认识。
- 注解：通常在研究数据库的时候用来做一般观察报告。可能包括关于未来数据库版本的注释或者为什么某些实体是或不是一个记录系统的原因。

确定记录系统

如同数据仓库世界中的许多术语一样，记录系统有许多不同的定义—这些差别取决于你问谁。我们定义的记录系统相当简单：数据的发源地。记录系统的这个定义非常重要，因为大多数的企业数据是跨越许多不同的系统冗余存储的。企业这样做是为了让这些末集成的系统共享数据。这样相同的数据很容易在企业中复制，移动，操作，转换，修改，清洗，或者被破坏，就造成了同样的数据有多个不同版本。大多数情况下，数据链末端的数据和原始的数据—记录系统已经大不相同。我们曾经在一个项目中四次将原来确定的源数据从记录系统中删除。在系统 3 和 4 之间的处理过程中，数据通过一个试图清洗数据的算法来转换。这个算法有一个未知的缺陷，它将其他字段数据插进来从而破坏了数据。这个缺陷在项目的数据发现阶段被 ETL 小组发现了。

处理导出数据。

你可能对周围的导出数据感到有些混淆。是否 ETL 过程应接收源系统中的计

算过的列作为记录系统？还是只有那些导出数据的基本元素才是合适的？这个问题的答案部分基于是否这个计算列是可加的。不可加的度量不能被用户在查询中组合，而可加度量却可以。因此你必须使用基本的元素来计算不可加度量。但要考虑清楚，是否你要在 ETL 过程中重建这些计算，如果这样做，你还要负责使这些计算和定义这些计算的业务规则保持同步。如果源系统中的业务逻辑改变了，ETL 过程将不得不重新修改和部署。因此，获取计算当作元数据以便让用户能够理解数据是如何得到的是很有必要的。

除非有确凿的证据表明原始的数据是不可靠的，否则我们建议你不要偏离我们的记录系统定义。紧记数据仓库的目标是在所有的主题域中可以共享一致的维度信息。如果你选择不用记录系统来装在你的数据仓库，那么一致的维度将变成不可能。如果你要为特定的需求用不同版本的数据来增加你的维度，则这些应该做为一致维度的额外的属性进行存储。

但是，每一种规则都会有例外。确定数据库名称或者文件名可能并不像想象中那样容易—尤其是当面对的是遗留的系统时。在一个项目中，我们曾经花费了数周的时间去获取订单数据库。每一个我们询问的人给这个数据库的说法都不同。后来我们发现每个地方都有一个本地的数据库版本。由于数据仓库的目标是跨越整个组织共享报表，我们开始记录每一个需要用 ETL 来迁移数据的本地数据库的名称。在我们研究这些数据库名称的时候，有一个开发员过来说，“你可以从这个数据复制程序中获得你需要的所有的数据库列表”。令我们惊奇的是，那儿竟然已经有了一个将本地数据库复制到一个中心资料库的程序。没有重建整个循环，我们选择使用这个程序来获得统一的数据库名称并从中心资料库加载数据到数据仓库。尽管真正的原始数据存在于每一个本地数据库中，然而使用中心资料库是最有效和最可靠的解决方案。

距离原始数据源越远，抽取到损坏数据的风险就越大。除非特殊情况，要保持实际的数据源只来自记录系统。

分析源系统：使用来自数据评估的结果

一旦确定了记录系统，下一步就是分析源系统以获得对其内容的更好的理解。如果是基于关系型技术，一般通过获得系统的 ER 图来对所选的记录系统加

深理解。如果没有 ER 图（不要感到奇怪），则自己创建它们。ER 图可以通过对数据库进行反向工程来获得。反向工程是一种通过已知的数据库元数据来生成 ER 图的技术。许多数据评估工具可以很容易做到这点。几乎所有的标准的数据建模工具都有这个功能，主流 ETL 工具中有一些也有这个功能。

利用反向工程获得一个记录系统的 ER 图是很有用的。但他和利用正向工程从复杂的 ER 模型来建立简单维度模型结构不同。当在拥有上百张表的规范化环境中只见树木不森林时，数据建模工具实际上在这正向工程中往往会失败的很惨。

在你深入 ER 图之前，查看一下数据库中的表和列的更高级别的描述。如果可以找到，它可能以非结构化文本描述的形式存在，可能已经过了时，但从概述开始远比试图通过在巨量的细节来发现概述要好得多。当然，别忘了听取源系统的头头们的意见，他们可以告诉你谁了解这些源系统中所有复杂的逻辑和已经发生的不断增长的变化！

具有浏览 ER 图的能力是执行数据分析的基础。ETL 小组中的每一个成员都要能读懂 ER 图并能很快认出实体关系。图 3.3 图解一个简单的 ER 图。

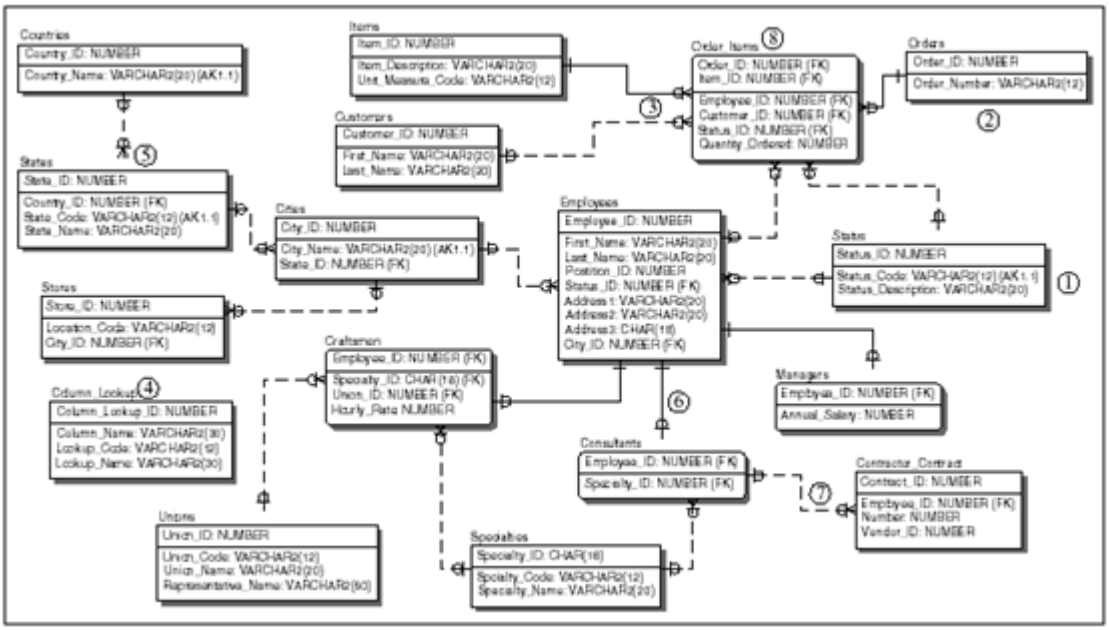


图 3.3 实体关系图

在以下数字标号的列表中，我们解释了在这个阶段你最需要搞清楚的最主要特性，包括唯一标识，是否可为空以及数据类型。这些是数据评估的主要输出结

果。但更重要的是，我们还解释了当表彼此相关时如何来标识；以及哪些列在多个表中有依赖关系。图 3.3 中的 ER 图概要中的主要特性如下：

1．唯一标识和自然键。唯一标识指的是能唯一的确定表中的一行的列。这个定义可能被误解，所以我们需要对此进行深入的研究。从参考完整性的观点出发，唯一标识就是一个表的主键。大多数时候，主键是人造的，虽然从 ETL 的观点来说它是唯一的，但并不能提供足够的信息来确定是否这一行是唯一的。在每一个设计良好的事务性表中，除了主键，必须至少还有一个自然键。自然键用来唯一描述该行的业务作用。比如，一个状态表可能包含一个 status_id ,一个 status_code 和 status_description 。很明显 status_id 是主键，但取决于业务规则，为了 ETL 的目标，status_code 可能才是唯一标识的自然键。选择正确的自然键的时候要特别小心，尤其当加载缓慢变化维时。

2．数据类型。记住，作为 ETL 分析师，你没有得到任何授权。列名称不能推断出数据类型。仅仅因为列的名称为 Purchase Order Number，是否我们就可以肯定这个列里只有数字，没有字母？另外，如果只有数字，是否有前导零？这些零对最终用户重要吗？在一个具体项目中，当建立人力资源数据集市时，我们遇到一个名为 SCORE 的源列；它是一个 CHAR(2)数据类型。在目标列，它是一个 NUMBER 数据类型。结果，我们发现大约 80％的数据是数字，其余的是字母（A-D 和 F）。ETL 程序需要转换任何的字母为它们对应的数字。DATE 和 TIME 元素存储为文本是常见的问题，当要将日期加载到数据仓库的时候，就由 ETL 程序来转换它们。

3．表之间的关系。理解表之间是如何关联的对获取数据时保证准确地连接是至关重要的。如果运气好，ER 图有关联表的连接线。鉴定表之间的关系包括分析关联线。不幸的是，数据处理者通常不得不仔细的察看 ER 图来确定表关联。当从不同的数据源往一个目标表上加载数据时，一种好的做法是将所有的数据源放在一个数据建模工具中并创建所有的关系。这种集成的 ER 图使得逻辑数据映射更加容易创建。

4．离散关系。在源系统的设计中，你不难见到用一个查找表来存储这个数据库中所有表的静态参考数据的情况。这个查找表中有一个列来表示哪一个表和

列由由相应的行组来支持。这对于不熟悉的人来说需要花费时间去理解。需要详细的以文档记录每一个行组以及相应的表和列。这些信息在将来映射多个维度时会用到。

5．关联和列的基数。推算查询的结果需要知道关联的基数。用鱼尾纹表示法，一个单横线表示基数为 1，只有一个唯一的对应。一条线加一个圆圈表示有 0 或 1 个对应。线的下面加 3 条线像鱼尾纹形状表示可以有多个对应。在关系型数据库中，所有的表关联都是以下关联中的一种：

- 一对一：一对一关系只在超类/子类场景中以及垂直表分割的情况下出现。一对一关系可以通过查看是否每一个表中该列都是主键来确定。
- 一对多：对外键参考来说这是最常见的关系。它很容易确定，注意表中的非键属性要参考另外一个表的主键。我们称这种非键属性为外键，并且我们坚持所有的外键都是完备的，也就是说，它是其对应主键的一个实例。
- 多对多：这种关系通常涉及三个表，其中有两个一对多关系。更确切地说，两个表之间有一个关联表。中心表或者关联表有一个联合主键和两个外键，一个对应其中一个表的主键，另一个对应另一个表的主键。

通常情况下，源系统的数据字典中没有一致的外键或者参考完整性定义。这些问题也可能通过简单的列名匹配和更复杂的数据评估来发现。

确保仔细研究了在源、中间集结表以及最终要提交的表中的所有的数据类型。数据建模小组常常创建和源所用的并不完全匹配的数据元素。有时候，你发现数据类型是故意设计成不匹配的。比如，即使当前系统只用数字并且是数字类型字段，但某些设计者还故意使所有的编码字段都支持字母字符。同时，准确评估每一个字段的长度。在某些情况中，目标数据仓库的数据长度或数字精度可以比源数据库小些。目标中的较短的数据长度会导致数据截断（数据丢失!）。当你发现不一致时，和数据建模小组一起检查以确定他们的意图。或者等待数据库纠正错误，或者获取转换和截断的业务规则。

数据内容分析

理解数据的内容是确定最好获取数据方法的关键。通常，在你开始和数据一起工作之前，你就已经意识到存在一些不规则数据。通常遇到的不规则数据列表如下：

- NULL 值：一个未处理的 NULL 值足以毁掉任何 ETL 程序。当出现在外键列中时，NULL 值是最危险的。以一个包含 NULL 值的列为基础来关联两个或更多表将导致数据丢失！切记，在关系型数据库中 NULL 并不等于 NULL。这就是为什么那些关联失败的原因。请检查源数据库中每一个外键是否有 NULL 值。如果存在 NULL 值，必须对表进行外关联。外关联返回所有的行，不管关联表的值是否匹配。如果 NULL 数据不是一个外键列而是一个数据仓库需要的列，你必须有一个处理 NULL 数据的业务规则。我们不希望在数据仓库中存储 NULL 值，除非它真的是一个未知的度量。只要可能，在往数据仓库中加载数据时一定要建立默认值来代替 NULL 值。
- 非日期字段中的日期：日期是非常奇特的元素，它们是唯一的有多种格式、字面上包含不同值但又意思相同的逻辑元素。幸运的是，大多数数据库系统支持多数的显示格式但只用一个标准格式来存储（针对此特定数据库）。然而有许多情形日期是以文本字段存储的，在遗留的旧系统中尤其常见。非日期字段中的可能的日期格式是多种多样的。以下是一些用文本来存储日期格式数据的例子：

13-JAN-02

January 13,2002

01-13-2002

13/01/2002

01/13/2002 2:24 PM

01/13/2002 14:24:49

20020113

200201

012002

只要你能想得到，一个日期的各种格式可以写满数页纸。当源数据库系统对日期数据不做控制或规范时，你必须非常仔细地检查以保证确实得到你期望的东西。

不管如何仔细地分析，我们都建议在从关系型数据库系统中导出数据时使用外连接，因为通常远程系统的参考完整性很难保证的。

在 ETL 过程中收集业务规则

你可能以为在 ETL 过程的这个阶段所有的业务规则都必须收集完毕。没有清楚所有的业务规则之前数据建模者怎么能创建数据模型，不是吗？错。数据模型小组需要的业务规则和 ETL 小组需要的业务规则有很大的区别。比如，数据建模定义的状态维可能是这样的：

Status Code——状态是一个 4 位数的代码来唯一的标识产品的状态。这个代码有一个简短的描述，通常是一个词，还有一个完整的描述，通常是一句话。

相反的，ETL 定义的状态可能是这样：

Status Code——状态是一个 4 位数的代码。然而，还有些遗留的代码只有 3 位数在某些情况下仍然会用到。所有的 3 位数代码必须转换为相应的 4 位数代码。这个代码的名称中可能会包含有单词 OBSOLETE（已不用的）。OBSOLETE 要从代码名称中删除并且这些废弃的代码必须设置废弃标志为‘Y’。当进入源系统时，无论如何使用，描述应该总是句子形式的。

在数据仓库项目中，业务规则对 ETL 处理来说更像技术性问题而不像任何的业务规则收集。无论它们技术上如何体现，这些规则仍然不是业务——ETL 小组不能投入到制定规则的业务中。ETL 设计者的任务是将用户需求转换成可用的 ETL 定义并且用一种业务用户能理解的方式清晰的说明这些技术定义。ETL 数据定义的过程是一个演化的过程。当你发现无法用事实证明的数据异常时，用文档记录下来并和业务人员（只有他们可以指示如何处理这些异常时）一起来讨论。任何从这些会议得到的转换都必须用文档记录，得到批准并签字。

4.4 集成异构数据源

这一章的前面部分陈述了许多当生成数据仓库时可能遇到的常见的数据系统。该部分讨论的是将面临的不同数据源集成的挑战。然而在可以整合数据之前，必须了解什么是数据集成。整合数据不仅仅是简单的将不同来源的数据收集并存储在一个资料库中。要更好的理解什么是集成，设想一下公司并购。在公司并购过程中，一个或者多个公司加入到其它类似（或不相似的）的公司中。当并购发生时，由业务决定哪一个公司存活以及哪一个公司被新的母公司合并。往往，当母公司发觉在某些它的附属的公司的实践和技术中的价值需要被合并到它的重建的组织中时就需要进行谈判了。一个成功的公司合并是形成一个具有单一业务利益的整体组织。这时需要英雄所见略同，以这种方式来统一业务术语（维度属性）和关键绩效指标（事实表中的事实）。如果你希望像公司合并一样来集成你的数据，那么你的数据仓库将是一个支持业务利益的有组织的单一信息源。

但对那些只完成了一半的合并，将允许他们的子公司做自己的业务，此时将会怎样呢？由于这些公司没有整合——他们只是联合，这种情况会引起一些问题。当建立一个数据仓库时，集成在几个地方发生。最直接的数据集成的形式是进行维度规范化。在数据仓库中，规范化的维度是统一整个企业分离的数据系统的一致性设计。

当一个维度是由来自几个独立系统的数据组成时，很重要的一点是把这些系统中的每一个唯一标识包含到数据仓库的目标维度中去。那些标识必须是用户可见的，确保他们能理解这个维度与其数据的对应关系，这样才可以回溯到他们的交易系统中。

当那些特殊的维度没法完全整合时会怎么样呢？不幸的是，这个问题更像一个政策问题而不是技术问题。整合维度和事实对于数据仓库项目的成功是尤其关键的。如果你的项目最后提供的是没有跨业务主题域的、完全不同的维度，那你的目标就没有完成。第 5 章我们将详细讨论加载维度的问题，但我们在这里要提及一些特殊的技巧，用于从一个完全不同的源系统环境加载规范化的维度。

1．标识源系统。在数据评估的逻辑数据映射的设计阶段，数据仓库小组必须一起找出目标维度和事实所在的各种源系统。数据仓库结构设计师必须尽可能

地发现数据仓库中每一个元素的潜在源并试着为每一个元素指定一个记录系统。要把这个记录系统当作将要加载的数据的最终的源。

2．理解源系统（数据评估）。一旦确定了源系统，你必须对每一个系统进行全面的分析。这也是数据评估的一部分。源系统的数据分析将揭示不可预知的数据异常和数据质量问题。这个阶段将发布关于元素在源系统的可靠性报告。在项目的这个阶段，如果仍然存在数据质量问题或者由于任何原因数据的可靠性仍然是问题，记录系统的可能要重新分配。

3．创建记录匹配逻辑。一旦理解了所有考虑中的系统的所有实体的所有属性，则下一步目标就是设计匹配规则使得分离的系统中的实体可以关联起来。有时，匹配规则就像标识各种客户表的主键一样简单。但是更多时候分离的系统并没有共享的主键。因此，你必须基于模糊的逻辑来关联表。也许社会保险号码可以作为你的客户的唯一标识，或者可能还需要再联合姓名、e-mail 地址等电话号码等一起考虑。我们不是要提供一个匹配方案，而是为了获得对客户关联关系的理解。在最终的匹配逻辑中必须包含和体现各种业务领域。不要忘记要确保这个记录匹配逻辑与各种法律保密条款相一致，比如医疗卫生服务领域的 HIPAA。

4．建立生存规则。一旦记录系统已经标识出来并且匹配逻辑已经核准，则可以创建当 ETL 过程中发生数据冲突时的生存记录。这意味着如果一个客户表在应付帐款、产品控制和销售系统都有时，业务规则必须决定当属性重叠时哪一个系统占有压倒性的地位。

5．建立非键属性的业务规则。记住，维度（以及事实）一般源自某个系统中的许多表和列。此外，许多源系统可以（其实通常这样做）包含最终输入一个目标维度的不同属性。举例来说，部门列表可能最初在你的 HR 部门；然而，部门的会计科目可能来自你的财务系统。即使 HR 系统可能是记录系统，但某些属性可能来自其它系统更可靠些。当属性存在几个系统中，但是不在记录系统中时，为非键属性分配业务规则显得尤其重要。在这些情况下，用文档记录并发布数据线元数据是非常重要的，这可以在用户没有看到他们期望的内容时避免对数据仓库的完整性产生怀疑。

6．加载一致的维度。数据集成过程的最后一环是物理加载一致的维度。这

个步骤要考虑缓慢变化维（SCD）类型并根据需要更新最新数据。加载一致的维度的详细信息请参考第 5 章。

数据仓库的吸引人之处就在于它具有真正集成的数据，同时也让用户从他们的角度查看维度。一致的维度和事实是企业数据仓库的基石。

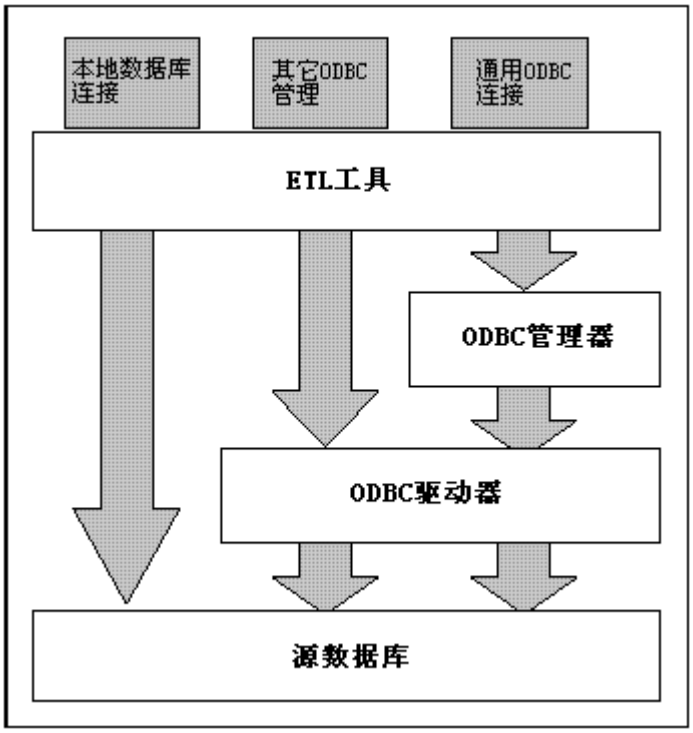
4.5 第 2 部分：从不同平台进行抽取的挑战

每一个数据源可能是在不同的 DBMS 中也可能在不同的平台上。尤其遗留的和专用的数据库和操作系统可能需要不同的过程语言来连接他们的数据。在企业级的数据仓库项目中，要准备连接到限定某种特殊语言的源系统。即使没有技术上的限制，部门或者子系统可能（并且通常是）有一种标准语言允许访问他们的数据。我们已经遇到的使用的标准包括 COBOL、FOCUS、EasyTrieve、PL/SQL、Transact-SQL 和 RPG。如果某种语言超出了你的 ETL 工具的使用范围或者根据经验变得无效，就需要向源系统的所有者求援把数据导出到平面文件格式中。

通过 ODBC 连接不同的源

开放数据库连接（ODBC）的建立使得用户可以从他们的 Windows 应用程序中去访问数据库。ODBC 建立的初衷是使得应用程序变成通用的，意思是说，如果一个应用程序的底层数据库改变了——比如从 DB2 变成了 Oracle，这个应用程序不需要重新编码和编译就可以适应这个变化。只需简单的改变 ODBC 驱动程序即可，这对应用程序来说是透明的。可以获得实际任何平台上的每一种关系数据库的 ODBC 驱动程序。也可以使用 ODBC 来访问平面文件。

ODBC 的灵活性是以牺牲性能作为代价的。在将数据提交到数据操作过程前，ODBC 增加了几个层次的处理。对那些通过 ODBC 处理数据的 ETL 过程，在 ETL 系统和后台数据库之间增加了两层。图 3.4 说明了 ODBC 环境中包含的层次。



图表 3.4　ETL 流程中的 ODBC 结构图

ODBC 管理器。ODBC 管理器是一个从 ETL 程序接收 SQL 并发送到相应的 ODBC 驱动程序的应用程序。它也维护应用程序和 ODBC 驱动程序之间的连接。

ODBC 驱动程序。ODBC 驱动程序是 ODBC 环境中的实际处理核心。ODBC 驱动程序将 ODBC SQL 转换为到后台数据库的本地 SQL。

正如你所担心的，一旦使用 ODBC 你将失去许多关系数据库特殊的功能。尤其是非 ANSI 标准的 SQL 命令不会被 ODBC 管理器接受，因为它要保持开放性。ODBC，尤其是微软的 OLE DB 和.NET 组件，在近年来已经得到了显著的改善，但是对于性能和本地关系数据库功能要求非常高的情况则必须首先考虑本地数据库驱动程序。只是不要因噎废食。ODBC 对某些不容易导出的比较麻烦的数据源可以提供一个统一格式的接口。

4.6 主机数据源

主机在上世纪 60 年代中期出现，被广泛用在全球那些最大型的企业中。主机和其它计算机的唯一区别是硬件结构。非主机，包括小型机和微型计算机，使用他们的中央处理器(CPU)进行所有的实际处理，包括从硬盘和其它外围设备取

数据。相反的，主机有特定的结构加强外围设备通道来处理输入/输出，CPU 可以专注处理数据，比如计算和均衡。

在许多大型企业，许多的日常业务数据由主机系统（某些小型机系统，比如 IBM AS/400）来处理和存储，从这些系统中集成数据到数据仓库时会面临一些特殊的挑战。以下是几种 ETL 小组必须熟悉并利用特定技术来处理的特殊的主机系统：

- COBOL copybooks
- EBCDIC 字符集
- 数字数据
- 重定义字段
- 压缩的十进制字段
- 多 OCCURS 字段
- 多记录类型
- 各种记录长度

接下来的部分将讨论这些主机特有的数据并提供特定的技术来进行处理。

处理 COBOL Copybook

COBOL 仍然是主机上的主要编程语言，数据的文件格式用 COBOL copybooks 来描述。一个 copybook 为主机数据文件定义字段名称和相关的数据类型。和其它在 ETL 过程中遇到平面文件一样，主机平面文件也只有两种数据类型：文本和数字。然而，数字值有许多存储方式，你必须充分理解来进行正确的处理。同样，日期仅简单的存储为数字（或文本）字符串，通常需要转换成为日期字段存储在数据仓库中。

		长度	位置		
01 EMP-RECORD.					
05 FIRST-NAME	PIC X(10).	10	1	-	10
05 MIDDLE-INITIAL	PIC X.	1	11	-	11
05 LAST-NAME	PIC X(15).	15	12	-	26
05 SSN	PIC X(9).	9	27	-	35
05 EMP-DOB.		8	36	-	43
10 DOB-YYYY	PIC 9(4).	4	36	-	39
10 DOB-MM	PIC 9(2).	2	40	-	41
10 DOB-DD	PIC 9(2).	2	42	-	43
05 EMP-ID	PIC X(9).	9	44	-	52
05 HIRE-DATE.		8	53	-	60
10 HIRE-YYYY	PIC 9(4).	4	53	-	56
10 HIRE-MM	PIC 9(2).	2	57	-	58
10 HIRE-DD	PIC 9(2).	2	59	-	60
05 TERM-DATE.		8	61	-	68
10 TERM-YYYY	PIC 9(4).	4	61	-	64
10 TERM-MM	PIC 9(2).	2	65	-	66
10 TERM-DD	PIC 9(2).	2	67	-	68
05 TERM-REASON_CODE	PIC X(2).	2	69	-	70

图 3.5　描述一个雇员记录的简单 copybook

图 3.5 说明了一个 70 字节的、固定长度的记录，用于描述一个简单的雇员记录。注意字段名称前有一个层级号。嵌套的层级号用来对相关的字段进行分组。COBOL 程序可以在定义的任何层次级访问字段名称。比如，一个程序可以访问 HIRE-DATE 来获得整个入职日期或者 HIRE-YYYY 来只获得年的部分。

文本和数字的数据类型用 PIC 语句来标志。PIC X 表示文本字段，而 PIC 9 代表这个字段是数字。字段长度是紧跟其后的数字。比如，语句 PIC 9(4)表示一个 4 字节的数字字段，而 PIC X(15)表示一个 15 字节的文本字段。PIC 语句也可以选择重复 X 或者 9 数据类型标志来表示，比如 PIC 9999 是一个 4 字节数字字段。

图 3.5 中的数据文件可以很容易通过 FTP 传输并加载到数据仓库中，因为所有的数据都包含显示格式。但是在试图从主机传输这个文件到数据仓库平台时，需要简单学习一下在 UNIX 和 Windows 平台上常见的 ASCII 字符集与使用在主机上的 EBCDIC 字符集之间的差别。

EBCDIC 字符集

大多数数据仓库所在的老的主机系统和基于 UNIX 和 Windows 的系统都采用二进制位和字节存储。每个字节由 8 个二进制位组成，每个二进制位表示一个二进制数。一个字节可以表示的最大数字是 255（就是 2 的 8 次方－1）。因此，可以用字节描述的唯一字符（比如，A-Z,a-z,0-9,标点符号和特殊字符）的数量是 256

（包括字符 0）。

转换 EBCDIC 到 ASCII

你可能认为既然两种系统都使用二进制位和字节存储，数据从主机系统到你的 UNIX 或者 Windows 系统应该是直接可用的。但是 UNIX 和 Windows 系统使用美国标准信息交换码（ASCII）字符集，而主机使用不同的字符集，即著名的扩展二进制代码十进制互换码（EBCDIC）。EBCDIC 多少有些类似 ASCII 的字符集，但是用不同的 8 位组合来表示他们。

比如，小写字母 a，在 ASCII 中，字母 a 的字符编码是 97（01100001），但是在 EBCDIC，字符编码 97 是/（前向斜杠）。在 EBCDIC 中 a 是字符 129（10000001）。事实上，没有一个字符在 ASCII 和 EBCDIC 中的字符编码是一样的。要在 UNIX 或者 Windows 系统中使用来自主机系统的数据，首先必须将它从 EBCDIC 转换为 ASCII。

在平台间转换数据

幸运的是，从 EBCDIC 到 ASCII 的数据转换是相当简单的，并且实际上它是自动的。假设你使用文件传输协议（FTP）来从主机系统转移数据到数据仓库平台。一个 FTP 连接需要两个节点——主机和客户端。当在两个系统之间创建 FTP 连接时，FTP 客户端会识别 FTP 主机的操作系统环境，由主机决定在两个系统之间传输数据时是否需要进行转换。因此，当在主机和 UNIX 或者 Windows 系统之间建立 FTP 连接时，FTP 主机将从 EBCDIC 到 ASCII 转换数据。另外，当数据在 UNIX 和 Windows 中时，FTP 增加特殊的换行和回车字符来表示一行（或记录）的结束。如果从 UNIX 或者 Windows 移动数据到主机上，FTP 也可以从 ASCII 转换到 EBCDIC。

如果收到的主机数据是在盒状磁带或者 CD-ROM 上而不是通过 FTP，则需要在数据仓库系统中指定将数据从 EBCDIC 转换成 ASCII。这种转换可以用 UNIX 的 dd 命令带 conv=ascii 选项来执行。对 Windows 来说，可以在因特网上获得一个 dd 命令的端口——这也适用于许多其它有用的 UNIX 命令。另外，处理字符转换的商业产品也很多。所有的 ETL 工具套件都可以处理这种转换。大多数功能强大的专门为 ETL 设计的工具可以直接从 EBCDIC 转换成 ASCII。

如果源数据在主机系统上，则 ETL 工具具有从 EBCDIC 到 ASCII 的数据转换

功能是很重要。如果可能，最好是在主机上进行转换以避免发生任何较小值的和压缩数字的损失。如果数据通过磁带或者其它介质获取，则这种转换必须由非主机环境中的 ETL 工具来实现。至少，这个 ETL 工具必须自动执行 FTP 以及在流中处理文件，并直接通过 ETL 过程将数据从主机提交到目标数据仓库中。

最后一点，虽然主机和 UNIX 或 Windows 系统使用不同的字符集，从一个系统向另一个系统转换数据也是一个相当简单的任务——简单，就是说，除非主机数据有些其它典型的主机环境特有的内容。后面的部分讨论主机数据可能具有的一些特征以及 ETL 过程中建议的处理策略。

处理主机数字类型数据

当你开始和量化的数据元素打交道时，比如美元总数、计数和余额，你会发现这些数字比眼睛看到的包含更多的信息。例如，通常在数字数据中找不到小数点，因为小数点是隐含的。比如，值 25,000.01 存储为 002500001。更糟糕的是，值 2,500,001 也是以同样的方式存储。那么主机 COBOL 程序是如何知道它是 25,000.01 而不是 2,500,001 呢？这在 PIC 语句中表示。下一部分会讨论 COBOL copybook 中的 PIC 语句的重要性和强大功能。

使用 PIC

你可以看到，在图 3.6 中 PIC 语句可以给同一个数据值不同的意思。要精确的处理来自一个旧的主机系统的数字值，你必须在传输它到数据仓库之前首先转换它为显示格式；否则， ETL 工具必须在 UNIX 或 Windows 平台上处理和解释这些主机值。要分解十进制数成为小数值，你可能认为简单的用这个数字值除以 10 的幂数(小数点位置数)就可以得到。如果所有的数字值只用隐含的小数点来存储，则这样做是正确的。然而，实际上不是那么简单，还必须考虑带符号的数字值问题。

PIC子句	值	数据
PIC 9(9)	2,500,001	002500001
PIC 9(7)V99	25,000.01	002500001
PIC 9(6)V9(3)	2,500.001	002500001
PIC S9(W) DISPLAY SIGN LEADING SEPARATE	2,500.001	002500001
PIC S9(7)V99 DISPLAY SIGN LEADING SEPARATE	25,000.01	+002500001
PIC S9(7)V99 DISPLAY SIGN LEADING SEPARATE	(25,000.01)	-002500001
PIC S9(7)V99 DISPLAY SIGN TRAILING SEPARATE	25,000.01	002500001+
PIC S9(7)V99 DISPLAY SIGN TRAILING SEPARATE	(25,000.01)	002500001-
PIC S9(7)V99	25,000.01	00250000A
PIC S9(7)V99	(25,000.01)	00250000J

图 3.6　COBOL copybook 中的 PIC 语句表示一个数字值的十进制位置。

在主机数据中，正负符号可能在数字值的前面或者后面。甚至，正负符号可能嵌入到数字值中。

最常见的格式是分区数字，符号嵌入到数字的最后一位数中，如图 3.6 中的最后两行所示。因此，在最后位置的 A 如何意味着既表示数字 1 也表示符号＋？同样的，J 又如何既表示 1 也表示-？方法是最后一个字节被当作两个独立的半字节（每个包含 4 比特），每个半字节独立解释－－当然，是用 16 进制！

对正数，第一个半字节设位 C，16 进制值是 1100，而负数设位 D，16 进制值为 1101。第二个半字节设成相应的数字的 16 进制值。当将第一个半字节(1100 为正数，或 1101 为负数）和第二个半字节合并时，你就得到了相应的 EBCDIC 字符，如图 3.7。

到现在，可能为如何从主机系统导出数值型数据伤透了脑筋。那么好，在试图解决这个问题之前，这里仍然有一个更加困难的问题可能在大多数旧主机系统中会遇到。

BIN	HEX	EBCDIC
11000000	C0	不可打印
11000001	C1	A
11000010	C2	B
11000011	C3	C
11000100	C4	D
11000101	C5	E
11000110	C6	F
11000111	C7	G
11001000	C8	H
11001001	C9	I
11010000	D0	不可打印
11010001	D1	J
11010010	D2	K
11010011	D3	L
11010100	D4	M
11010101	D5	N
11010110	D6	O
11010111	D7	P
11011000	D8	Q
11011001	D9	R

图 3.7　十六进制到 EBCDIC

解压压缩的数字

虽然目前计算机硬盘相对便宜，但是在过去磁盘存储几乎是计算机系统中最昂贵的部件。为了节省磁盘空间，软件工程师设计了创新的格式来存储数值型数据，它只需要比数字中的数位更少的字节。这些格式中最普遍使用的是 COMP-3，又称之为压缩数字。

在许多主机系统中，大多数(不是全部)数字数据是以 COMP-3 格式存储的。COMP-3 格式是一种简单的磁盘空间节省技术，使用半字节－或四位字节而不是全字节来存储数字位。每个数字位可以用一个四位字节以二进制格式存储。一个 COMP-3 数字字段的最后一个半字节存储数字值的符号（正号/负号）。使用半字节存储数字位和使用显示格式存储相比几乎节省了一半的空间。但是这种简单的空间节省技术给 EBCDIC 到 ASCII 字符集转换制造了一个大麻烦。

这种转换难题的一个直接结果是，包含数字值并且使用数字存储格式如分区数字或 COMP-3（还没有提到 COMP、COMP-1 和 COMP-2）的主机数据不能简单的从 EBCDIC 转换到 ASCII，然后在 UNIX 或 Windows 数据仓库中进行处理。

以下这些技术在进行主机数字数据的集成过程中一定会用到的：

- 在将数据传输到数据仓库系统之前，使用一个用 COBOL、汇编语言或第四代语言如 SAS、Easytrieve 或 FOCUS 写的简单程序来按照显示格式重新格式化主机的数据。一旦数据按照这种格式重新格式化，它就可以像本章前面所说的那样通过 FTP 转换到 ASCII。
- 以本地 EBCDIC 格式传输数据到数据仓库系统。这个选项只有当你的 ETL 工具或过程可以处理 EBCDIC 数据才有用。以下几种类型的工具可以执行这个任务。

1. 使用可以处理本地 EBCDIC 的成熟的 ETL 工具，包括精确的处理以任何主机类型的数字格式存储的数字数据。
2. 在数据仓库平台上使用一个能重新格式化数据到显示格式的应用程序。如果收到 EBCDIC 数据并且没有某个 ETL 工具的帮助来写 ETL 过程，我们强烈建议你购买一个可以执行数字格式转换和 EBCDIC 到 ASCII 转换的应用程序。有些可得到的相对便宜的商业程序可以很好的处理这种任务。

使用重定义字段

为了不浪费空间(过去通常存储是很昂贵的)，主机工程师设计了 REDEFINES，这种方式允许相互分割的数据元素占用相同的物理空间。图 3.8 包含了一个来自 COBOL Copybook 的摘要，可用来说明主机数据文件中的 REDEFINES 的概念。这个摘要描述了用来表示一个雇员的工资信息的数据字段。注意到 EMPLOYEE-TYPE，这是一个表示这个雇员是否是全职的的按小时计薪的单字节代码。另外，注意到两个独立序列的字段存储这个雇员的工资信息。这个字段集是否使用取决于是否这个雇员是全职的或是按小时计薪的。全职的雇员工资在三个字段（PAY-GRADE, SALARY, PAY-PERIOD）中表示，总共占了 8 个字节。按小计时计薪的雇员使用一个不同的字段集，需要占 7 个字节（PAY-RATE, JOB-CLASS）。

	长度	位置		
05 EMPLOYEE-TYPE PIC X.	1	71	-	71
88 EXEMPT VALUE 'E'.				
88 HOURLY VALUE 'H'.				
05 WAGES.	8	72	-	79
10 EXEMPT-WAGES.	8	72	-	79
15 PAY-GRADE PIC X(2).	2	72	-	73
15 SALARY PIC 9(6)V99 COMP3.	5	74	-	78
15 PAY-PERIOD PIC X.	1	79	-	79
88 BI-WEEKLY VALUE '1'.				
88 MONTHLY VALUE '2'.				
10 NON-EXEMPT-WAGES REDEFINES EXEMPT-WAGES.	8	72	-	79
15 PAY-RATE PIC 9(4)V99.	6	72	-	77
15 JOB-CLASS PIC X(1).	1	78	-	78
15 FILLER PIC X.	1	79	-	79

图表 3.8 COBOL copybook 中的 REDFINES 子句

因为一个雇员要么是全职的的要么是按时计薪的，不可能两者都是，这两个字段集同时只有一个用到。全职工资字段占据文件中从 72 到 79 的位置，而按时计薪工资字段占据从 72 到 78 的位置。此外，注意到全职和按时计薪工资即使在相同的位置也使用不同的数据类型。当读取雇员记录时，程序必须决定如何基于位置 71 的 EMPLOYEE-TYPE 的值来解释这些位置的信息。

相同的位置可能同时有多个 REDEFINES 与之相关，所以有多于两个用法的可能，相同的位置可以有两个、三个或更多可能的用法。REDEFINES 引入了一种更加复杂的状况，所以仅仅有 EBCDIC 到 ASCII 字符集转换是不够的。

当遇到多 REDEFINES 源时，应该考虑对源数据定义每个独立传输的抽取逻辑，如果后续处理差别很大的话（如全职工资和按时计薪工资的例子）。这允许为每个抽取建立独立的代码而不是为两种情况设计一个包含许多种情况的复杂逻辑。

多重 OCCUR 子句

主机和 COBOL 先于关系数据库和 Edward Codd 的规格化规则。先于使用关系理论来设计数据库，主机 COBOL 程序使用 OCCURS 子句来处理重复分组，方法是在一个数据文件中定义重复的数据字段。比如，在图 3.9 你可以看到存储关于绩效等级信息的雇员记录的区域。这条记录设计用来对多达 5 个绩效等级进行跟踪。但并不是创建 5 次需要的字段(记住，这先于关系理论，因此这里没有一个

带外关键字并指向雇员表的独立的绩效等级表)它们只在一个特殊 OCCURS 字段中命名一次。这个 OCCURS 子句表示这个字段重复的次数值。本质上，OCCURS 子句在文件中定义了一个数组。因此，在雇员记录中，第一个绩效等级的数据占据从 80 到 99 的位置，第二等级从 100 到 199，第三等级从 120 到 139，第四等级从 140 到 159，接着第五等级以及最后一个等级则从 160 到 179。

		长度	位置		
05 PERFORMANCE-RATING-AREA	PIC X(100).	100	80	-	179
05 PERFORMANCE-RATINGS		100	80	-	179
REDEFINES PERFORMANCE-RATINGS-AREA.					
07 PERFORMANCE-RATING	OCCURS 5 TIMES.	20	80	-	99
15 PERF-RATING	PIC X(3).	3	80	-	82
15 REVIEWER-ID	PIC X(9).	9	83	-	91
15 REVIEW-DATE.		8	92	-	99
20 REVIEW-DATE-YYYY	PIC 9(4).	4	92	-	95
20 REVIEW-DATE-MM	PIC 9(2).	2	96	-	97
20 REVIEW-DATE-DD	PIC 9(2).	2	98	-	99

图表 3.9　带有 OCCURS 子句的 COBOL copybook 在数据记录中定义重复组

在大多数情形下，ETL 处理必须对主机文件中的 OCCURS 部分的任何数据进行规格化。即使这可能可能用手工编码的 ETL 过程来管理重复数据，但在这里还是强烈建议你使用成熟的 ETL 工具，它们允许你使用 COBOL copybook 来定义输入或至少允许你以某种方式手动定义输入文件数组。如果你的工具不支持输入数组，你只能坚持费力地写代码来处理来自旧主机系统的源记录中的重复分组。

有时程序员使用 OCCURS 来存储不同的事实在一个数组中，而不是存储同一个事实 N 次。比如，假设 O-DATE 发生 4 次。第一个日期是 CREATE（创建），第二个是 SHIP（运输），第三个是 ACKNOWLEDGE（确认），第四个是 PAYMENT（付款）。因此，这时候你不必规格化这个 OCCURS 数据，只要为数组中的每个位置创建独立字段即可。

为确保数据集成，模型数据结果来自一个 COBOL OCCURS 子句的规格化方法(一个主表和一个子表)在数据仓库的集结区。在独立的表中集结这些数据是一个好办法，因为这个过程的结果很可能是加载数据到一个事实表和一个维表、两个独立维表或两个独立事实表中。我们发现在这种情况下，将数据集成到数据仓库之前把它安置下来是很有意义的。

管理多主机记录类型文件

多记录类型文件的概念在讨论 REDEFINES 的部分有简单涉及。前面讨论的 REDEFINES 和我们现在介绍的概念的主要区别是，多记录类型文件不是记录的一部分包含多个定义，而是整个记录有多个定义。多记录类型通常将一个单独的逻辑记录跨越存储在 2 个或更多的物理记录上。图 3.10 包含一个 COBOL copybook 的扩展，说明重定义整个记录的概念。

在图 3.10 中，REDEFINES 子句应用到整个记录。因此现在文件不仅仅包含一个雇员的基本信息，同时也包括一个雇员在这个公司的工作经历。在这个文件中，每一个雇员至少有两个记录：一个是 EMP-RECORD，另一个是 JOB-RECORD。当一个雇员换了新工作，文件增加一个新的 JOB-RECORD。因此一个雇员的全部工作经历包含在这个文件的两个或更多记录中：一个 EMP-RECORD 和一个或多个 JOB-RECORD。

		长度	位置		
01 EMP-RECORD.		70	1	-	70
05 REC-TYPE	PIC X(1).	1	1	-	1
05 FIRST-NAME	PIC X(10).	10	2	-	11
05 MIDDLE-INITIAL	PIC X.	1	12	-	12
05 LAST-NAME	PIC X(15).	15	13	-	27
05 SSN	PIC X(9).	9	28	-	36
05 EMP-DOB.		8	37	-	44
10 DOB-YYYY	PIC 9(4).	4	37	-	40
10 DOB-MM	PIC 9(2).	2	41	-	42
10 DOB-DD	PIC 9(2).	2	43	-	44
05 EMP-ID	PIC X(8).	8	45	-	52
05 HIRE-DATE.		8	53	-	60
10 HIRE-YYYY	PIC 9(4).	4	53	-	56
10 HIRE-MM	PIC 9(2).	2	57	-	58
10 HIRE-DD	PIC 9(2).	2	59	-	60
05 TERM-DATE.		8	61	-	68
10 TERM-YYYY	PIC 9(4).	4	61	-	64
10 TERM-MM	PIC 9(2).	2	65	-	66
10 TERM-DD	PIC 9(2).	2	67	-	68
05 TERM-REASON_CODE	PIC X(2).	2	69	-	70
01 JOB-RECORD REDEFINES EMP-RECORD.		70	1	-	70
05 REC-TYPE	PIC X(1).	1	1	-	1
05 DIVISION	PIC X(8).	8	2	-	9
05 DEPT-ID	PIC X(3).	3	10	-	12
05 DEPT-NAME	PIC X(10).	10	13	-	22
05 JOB-ID	PIC X(3).	3	23	-	25
05 JOB-TITLE	PIC X(10).	10	26	-	35
05 START-DATE.		8	36	-	43
10 START-YYYY	PIC 9(4).	4	36	-	39
10 START-MM	PIC 9(2).	2	40	-	41
10 START-DD	PIC 9(2).	2	42	-	43
05 END-DATE.		8	44	-	51
10 END-YYYY	PIC 9(4).	4	44	-	47
10 END-MM	PIC 9(2).	2	48	-	49
10 END-DD	PIC 9(2).	2	50	-	51
05 FILLER	PIC X(19).	19	52	-	70

图表 3.10　在同一文件中重新组织多个记录类型

在这个文件中，记录的物理顺序是至关重要的，因为 JOB-RECORD 没有任何信息来连接它们到相应的 EMP-RECORD 上。一个雇员的 JOB-RECORD 紧跟在其 EMP-RECORD 之后。因此要精确的处理一个雇员的工作经历，你必须把两个或更多物理上相邻的记录当作一个逻辑记录。

使用多记录类型的好处是它再次节省了空间。另一个选择(没有使用关系理论)是用一个相当宽的、浪费空间的记录来存储所有的数据再同一个记录中。假设，例如想跟踪到前 5 个职位的工作经历，则必须向每个雇员记录增加 255 个字节（基本 EMP-RECORD，加上 5 次出现的 JOB-RECORD 字段（5×51 字节））。但工作经历字段的数量是不定的——它取决于一个雇员保持多少个工作。

通过使用多记录类型，主机系统可以仅仅存储需要的工作经历记录，因此只有一个工作的雇员只有一个 JOB-RECORD(70 字节包括 FILLER)，在这个文件中节省了 185 字节。此外，也不再限制文件中的工作数量。每个雇员可以增加无限的 70 字节 JOB-RECORD 的个数。

雇员例子只有两个记录类型，但多 REDEFINES 可以用来创建任何数量的记录类型，这些可以组合在一个单一逻辑记录中。如果我们展开这个雇员例子，你可能想到第三个记录类型用来存储雇员的边际利益和第四个类型存储雇员的家庭成员数。

ETL 过程必须通过在内存变量中保持来自第一个物理记录的值在一个集中(它只是逻辑记录的第一部分)来管理多记录类型，这样就可以和后续记录的其它数据进行关联。

处理主机变化记录长度

在前面部分，我们讨论了如何使用多记录类型将一个单一实体跨越存储到两个或更多记录的方法。一个可变的记录长度是另一种用在主机文件存储位置信息的方法。它不是用独立的 JOB-RECORD 来存储一个雇员的工作经历，而是每个工作都存储在一个 OCCURS 工作经历段中。此外，如图 3.11 所示，记录的这些段没有固定数量的段数，段的数量在 0 到 20 之间变化，这基于在 DEPENDING ON JOB-HISTORY-COUNT 子句中的数字值。每次增加雇员工作(直到最大值 20)，记录长度都会增加 50 字节。

		长度	逻辑位置		
01 EMP-RECORD.		70	1	-	70
05 REC-TYPE	PIC X(1).	1	1	-	1
05 FIRST-NAME	PIC X(10).	10	2	-	11
05 MIDDLE-INITIAL	PIC X.	1	12	-	12
05 LAST-NAME	PIC X(15).	15	13	-	27
05 SSN	PIC X(9).	9	28	-	36
05 EMP-DOB.		8	37	-	44
10 DOB-YYYY	PIC 9(4).	4	37	-	40
10 DOB-MM	PIC 9(2).	2	41	-	42
10 DOB-DD	PIC 9(2).	2	43	-	44
05 EMP-ID	PIC X(8).	8	45	-	52
05 HIRE-DATE.		8	53	-	60
10 HIRE-YYYY	PIC 9(4).	4	53	-	56
10 HIRE-MM	PIC 9(2).	2	57	-	58
10 HIRE-DD	PIC 9(2).	2	59	-	60
05 TERM-DATE.		8	61	-	68
10 TERM-YYYY	PIC 9(4).	4	61	-	64
10 TERM-MM	PIC 9(2).	2	65	-	66
10 TERM-DD	PIC 9(2).	2	67	-	68
05 TERM-REASON_CODE	PIC X(2).	2	69	-	70
05 JOB-HISTORY-COUNT	PIC 9(2).	2	71	-	72
05 JOB-HISTORY OCCURS 20 TIMES DEPENDING ON JOB-HISTORY-COUNT.		varies	73	-	varies
05 DIVISION	PIC X(8).	8	73	-	80
05 DEPT-ID	PIC X(3).	3	81	-	83
05 DEPT-NAME	PIC X(10).	10	84	-	93
05 JOB-ID	PIC X(3).	3	94	-	96
05 JOB-TITLE	PIC X(10).	10	97	-	106
05 START-DATE.		8	107	-	106
10 START-YYYY	PIC 9(4).	4	107	-	110
10 START-MM	PIC 9(2).	2	111	-	112
10 START-DD	PIC 9(2).	2	113	-	114
05 END-DATE.		8	115	-	114
10 END-YYYY	PIC 9(4).	4	115	-	118
10 END-MM	PIC 9(2).	2	119	-	120
10 END-DD	PIC 9(2).	2	121	-	122

图表 3.11　使用 DEPENDING ON 表示的 COBOL copybook 中的变长记录

在 copybook 中使用 DEPENDING ON 子句的可变长记录使直接的 EBCDIC 到 ASCII 字符集转换变得无效。以下列出一些可以减少在 ETL 过程由于可变长记录引起的数据破坏的风险的方法。

- 在主机上转换所有数据到显示格式并转成固定长度记录，为所有不用可变段的每个记录的末尾增加空格。
- 以二进制格式传输文件到数据仓库平台。这种技术需要有工具可以解释本章所讨论的所有的主机数据格式的细微差别。功能强大的专用 ETL 工具可以处理大多数或所有这些情况。如果你的数据仓库项目没有这样的工具，各种价格范围的第三方的应用程序都可以用来解释和转换主机数

据到 UNIX 或 Windows 平台。

- 最后一个选择是开发自己的代码来处理所有已知的、在处理旧数据时可能遇到数据格式问题。然而，这个选择一定要慎重，处理所有可能的数据场景的时间和精力的付出可能超过在主机上开发重新格式化程序或购买一个实用程序来处理主机数据的成本。

从 IMS, IDMS, Adabase 和 Model 204 抽取

如果使用任何以上这些系统，则需要特殊的抽取器。对于初学者，这些中的每一个都可以使用 ODBC 通道。详细讨论从这些旧数据库系统抽取的技术超出了本书的范围，但是它们可能对某些人是很重要的。

4.7 平面文件

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

平面文件广泛应用于任何数据集结应用系统。在大多数数据仓库环境中，你不可能避免平面文件。ETL 过程利用平面文件至少有三个原因：

- 传输源数据。当数据源在主机系统或外部系统中时，以平面文件 FTP 到数据集结区是很平常的。来自个人数据库或电子表格的数据常常也通过平面文件传输。
- 工作表/集结表。工作表是 ETL 过程为自己使用专门创建的。最常见的情况是，使用平面文件是因为直接从文件系统读写的 I/O 速度远远快于插入和查询 DBMS 系统。
- 块加载准备。如果 ETL 工具不支持数据流程中的块加载，或是想为安全或归档原因装载一个文件，在所有的数据转换完成后，你需要在文件系统上创建一个平面文件。一旦创建了这个平面文件，则块加载过程就可以读取这个文件并加载它到数据仓库中。

不是所有的平面文件都是相同的。平面文件基本上分为两种：

- 固定长度的
- 分隔符分隔的

处理固定长度平面文件

有时，不能从原始系统得到数据仓库需要的物理数据。在这种情况下，需要一个由支持源系统的程序员创建的平面文件。通常，这些文件是固定长度的——也称之为基于位置的平面文件。有时我们身处数据仓库项目中，它需要来自遗留源系统并且需要非常复杂计算的数据。计算的名称已经给了数据仓库架构师，但数据源及其计算规则却相当不清晰。在一段耗费时间的调查和研究工作之后，在 1977 年用 COBOL 写出了一份已发现的字段的报告。自然而然的，负责写这份报告的工程师离开了公司。此外，再也找不到源代码。这样这些计算就不再复杂，因为它们根本就不存在。

不幸的是，不能告诉业务用户说这些数据不能提供，并且在数据仓库中是不可用的。小组的解决方案是重定向这个包含所需数据的为输出平面文件，并且把来自报告的预生成的数据作为数据源。由于这个报告的自然属性——固定宽度列——而可以简单的把它当作一个基于位置的平面文件并在后续的 ETL 过程中处理。

处理固定长度平面文件需要布局文件来准确描述文件中的字段，如图 3.12 所示。固定长度布局文件应包含文件名称、字段开始位置、字段长度、字段数据类型（一般为文本或数字）。有时，结束位置也要提供，因为如果没有的话，就必须基于开始位置和长度来计算每个字段的结束位置，而这在 ETL 工具中是必须的。(如 Pervasive 的 Structured Schema Designer)。

字段名称	长度	开始	结束	类型	注释
记录类型	2	1	2	AlphaNumeric	记录类型，如用“H”表示标题，或者用“D”表示详细记录
SSN	9	3	11	Numeric	雇员的社会保险号
名字	20	12	31	AlphaNumeric	雇员的名字
中间名	1	32	32	AlphaNumeric	雇员的中间名
姓	20	33	52	AlphaNumeric	雇员的姓
名字前缀	5	53	57	AlphaNumeric	Jr. Sr. 等等
生日	8	58	65	Numeric	雇员的生日，用“YYYYMMDD”表示
状态编码	6	66	71	Numeric	雇员的状态（“A”，“R”，“T”等）
分公司编码	2	72	73	Numeric	雇员分公司编码
部门编码	2	74	75	Numeric	雇员在分公司的部门编码
位置编码	2	76	77	Numeric	雇员在组织结构中的位置编码
筛选	1	78	78	AlphaNumeric	筛选空格
添加日期	8	79	86	Numeric	添加记录到系统的日期
修改日期	8	87	94	Numeric	记录最后一次修改日期

图表 3.12　固定长度平面文件显示

大多数 ETL 工具很可能必须手工一次性输入平面文件的布局文件。输入布局之后，工具会记住这个布局并在每次访问实际的平面文件时使用相同的布局。如果文件布局改变了或数据超出了原来分配的位置，ETL 处理程序会失败。不幸的是，不像 XML，当你处理固定长度平面文件时不会发生隐含的布局文件有效性检验——在数据处理之前明确的预处理测试必须成功通过。

当处理固定长度平面文件时，尝试检验文件中数据的位置是否是正确的。一种快速检验位置有效性的办法是测试任意日期（或时间）字段以确认其是一个有效的日期。如果产生了移位，日期字段很可能包含希腊字母或不合逻辑的数字。其它具有非常特定含义的字段也可以用同样的方法来测试。XML 提供更加具体的有效性检验功能。如果数据校验或一致性是一个问题，试着说服数据提供者以 XML 格式提交数据。

基于位置的平面文件一般在文件系统中以.TXT 作为后缀。然而，基于位置的平面文件实际可以用任何文件扩展名——或根本没有——都可以以同样的方式处理。

处理有分隔符的平面文件

平面文件一般用一套分隔符分割文件中的数据字段。分隔符代替了使用位置来说明字段的开始和结束位置。分隔符文件可以用任意符号或符号组来分割平面文件的字段。最常用的分隔符是逗号。逗号分隔符文件常常以.CSV 作为文件扩展

名。然而很显然，其它特定应用程序的分隔符平面文件可能简单的有个.TXT 扩展名或根本没有扩展名。

大部分 ETL 工具都有分隔符文件向导，一旦开发者指定了实际的分隔符号，就会扫描平面文件或它的样本，来检验文件中的分隔符和指定布局文件。通常，分隔符文件的第一行包含列名称。ETL 工具应该可以识别第一行中的列名称并在元数据层分配逻辑列名称，在接下来的后续数据处理中则忽略这一行。

和基于位置的平面文件一样，分隔符文件也没有隐含的校验方法。需要 ETL 小组进行明确的校验测试并将其加入数据处理例行测试中。

4.8 XML 数据源

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

扩展标志语言（XML）慢慢的但的确正在成为共享数据的标准。这个以规则格式文档存储数据的新规范已经产生了许多令人振奋的消息。在它的大力推广之后，我们现在认为本质上所有的数据仓库源都应该包含 XML。但至今为止，我们已经看到共享因特网数据的方法并没有改变多少。另一方面，共享外部数据的方法在过去几年已经完全成熟或开始几乎完全是 XML。

XML 已经显示出成为企业之间交换数据的通用语言。如果你的数据仓库包括来自外部源的数据——那些来自企业之外的——可能那些源都将以 XML 提供。

要处理 XML，首先必须清楚它的工作原理。XML 包含 2 个重要的元素：它的元数据和数据本身。XML 元数据可以用多种方式提供。以下的部分说明不同格式的 XML 元数据以及这些对 ETL 意味着什么。

字符集

字符集是用来显示和打印计算机输出的唯一符号组。大多数关系型数据库管理系统的默认字符集是 ISO8859-15 (Latin 9)。这个字符集通过启用欧元符号：€ 代替了 ISO8859-1 (Latin 1) ，这个拉丁字符集用在西方国家来支持基于英语字母表

的语言。然而，由于 XML 是主要用于互联网的语言，它必须支持全世界的语言和字母表，而不仅仅是西方国家的。因此，XML 支持 UTF-8 字符集。UTF-8 是一种保留基本 ASCII 编码方法并且也支持统一字符编码（ISO10646）的通用字符集(UCS)。UTF-8 支持来自全世界的大多数语言和字母表。

许多问题的产生是由于源 XML 文档和目标数据仓库基于不同的字符集造成的。当然，这种不同步的字符集在集成独立的系统(不仅仅是 XML 数据集)时总是存在危险。但在绝大多数情况下，这种风险是极小的，因为数据库系统几乎毫无例外的使用拉丁字符集。不使用基于拉丁字符集而是使用统一字符编码来支持特殊字母表或字符的组织，应该采用一个企业范围的标准字符集来避免集成的难度。无论何时遇到集成数据的需求，尤其使用 XML 集成来自外部源的数据，就必须做好处理不同的字符集的准备。比较好的一点是 XML 至少可以通过适当的元数据来标记文档，以此来指定使用的字符集。例如，标签<?xml version=”1.0” encoding=”UTF-8” ?> 表示这个 XML 文档编码采用 UTF-8 字符集。

XML 元数据

我们常常听到有人说 XML 从存放数据角度来讲没有比平面文件强什么。在我们看来，确实是这样的。。唯一使得 XML 和平面文件有点类似的地方是它们都存储在文件系统而不是数据库中。并且事实上，许多数据库系统增加了读取、创建和存储 XML 的能力，称之为内部嵌入 XML。

XML 是一个特殊的实体，因为它存储数据但又被看作一种语言。它不是一个应用程序，因此，它依赖于其它应用程序来使它工作。然而它不仅仅是数据因为它内嵌了标签。XML 文档中的标签是为什么 XML 如此功能强大的原因。但是自定义数据也带来了一定的成本。标签，包含它的元数据，可能占据 XML 文件大小的 90％，只留下大约 10％的 XML 文件是实际的数据。如果你当前的文件是 XML，它们可能 10 倍于它们初始的大小－－即正好存储同样行数的数据。

具有讽刺意味的是，数据仓库的主要目标是保持数据尽可能的收缩以使得处理的尽可能的快，虽然 XML 标签会引起背离这个目标的大量数据处理，但是许多人还是坚持使它成为一个数据交换的标准。标签不仅仅增加数据文件的大小，而且它们还增加了相当的复杂度。由于这些固有的复杂性，绝对不要试图写自己

的 XML 文档界面来解析 XML 文档。XML 文档的结构相当棘手，并且 XML 解析器的构造是它本身要解决的问题——而不是由数据仓库小组来设计。市场上有许多 XML 解析器（或者处理器），并且现在大多数 ETL 厂商在他们的产品中也会包含它们。

不要试图手工解析 XML 文件。XML 文档需要一个 XML 处理器引擎来处理。许多主流的 ETL 工具现在都在他们的产品中包含 XML 处理器。确认所用的 ETL 工具确实具有它们声称的 XML 的处理能力。

要处理一个 XML 文档，必须首先知道这个文档的结构。一般 XML 文档的结构放在一个独立的文件中。下面这些部分将讨论可能伴随 XML 以及提供 XML 文档结构的每个可能的元数据文件。

DTD（文档类型定义）

正如那些将 XML 看作数据源而不是编程语言的人一样，我们将 DTD 等同于 COBOL 的布局文件。它是一个描述 XML 文档或文件的数据结构的文件。在 XML 文档中可以嵌入定义信息，但是为了有效性检验，应该让元数据和实际数据文件相互独立。DTD 可能相当复杂，包含以下这些允许的 XML 数据结构：

- 基本数据。如果一个元素必须只包含数据，它用标签#PCDATA 声明。
- 元素结构。在 DTD 中一个元素的结构是在一个元素中用元素名称列表来指定。比如，<!ELEMENT OrderLineItem (ProductID, QuantityOrdered,Price)>表示一个订单项（order line item）由产品编号（product ID）、订货量（quantity ordered）和订货时的产品价格组成。
- 混合的内容。当允许数据或元素中任意一个时，用 PCDATA 声明来表示允许基本数据及元素名来表示允许嵌套元素。
- 是否可为空。这不是一个类型！在 XML 中，你用’nill=’或’nillable=’标签来表示如果一个字段提否可以是 NULL。在 DTD 中，会发现用一个问号（？）来表示子元素是可选的。比如，代码<!ELEMENT Customer (FirstName, LastName,ZipCode?, Status)>表示名、姓以及状态是必须的，但邮政编码是可选的。

- 对应关系。用加号(＋)表示一对多。比如，<!ELEMENT Customer (FirstName, LastName,ZipCode+, Status)>表示客户可以有多个邮政编码。
- 允许的值。类似于约束性检查，XML 通过列出用竖线（|）分隔的可接受的值来指定允许的值。比如，<!ELEMENT State (Alabama|Louisiana|Mississippi)>表示州必须包含 Alabama、Louisiana 或 Mississippi。

在写这本书时候，XML 规范仍然在修订和改变中。今天，DTD 多数被当作过时的技术。因为 XML 正在发展更多的数据中心角色，类似于关系型数据库，大多数的 DTD 将被当作标准元数据的 XML 结构所取代。下一部分将讨论 XML 结构以及它们与 DTD 之间的不同。

XML 结构

XML 结构是 DTD 的继任者。XML 结构比 DTD 更加丰富和实用，因为它们是设计用来扩展 DTD 的。XML 结构允许直接定义一个 SQL CREATE TABLE 语句。这在简单的 DTD 中是绝不可能的，因为在 DTD 中没法指定详细的数据类型和字段长度。XML 结构的一些特征包括：

- 出现在 XML 文档中的元素
- 出现在 XML 文档中的属性
- 子元素的数量和顺序
- 元素和属性的数据类型
- 元素和属性的默认值和固定值
- 添加的可扩展性
- 支持命名空间
- 命名空间

XML 越来越来普遍的原因是它强制独立的数据源产生一致的和可预期的数据文件。但实际是不同的系统中同一个实体的元素在意义和用法上总是有些微小的差别。比如，如果得到一个同时来自人力资源系统和交易系统的客户文件，它

们可以对这个客户有不同的定义。一个部门可能有些交易与组织进行，同时又与个人进行其它交易。虽然两者都是客户，但组织和个人的属性差别还是很大的。为了减少这种情况，XML 文档可以引用命名空间。命名空间指示到哪里取得元素或属性的定义。同一个实体可以基于它声明的命名空间有不同的意思。同一个客户实体，带有标签<Customer xmlns=http://www.website.com/xml/HRns>命名空间与引用<Customer xmlns=http://www.website.com/xml/OPSns>的同一个实体具有不同的含义。

因为 XML 对基于 Web 的应用程序是新兴的数据源，这将很可能对数据仓库数据源产生越来越多的影响。当选择 ETL 工具时，确认它可以天然处理 XML 和 XML 结构。

4.9 Web 日志数据源

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

实际上世界上几乎每个公司都有自己的网站。每个网站之下也都有日志——网络日志——记录每个对象从 Web 服务器上传或下载的服务。网络日志非常重要，因为它们揭示了用户在网站上的交互情况。

这里的网络日志不是 weblog 或 blog！我们的网络日志指的是每一个 Web 服务器自动产生的控制文档。blog（博客）是一种由个人发布和维护的日记，主要是青少年，所有人都可以阅读它们。

理解的网站上的客户行为就像跟着一个客户逛商店并记录他或她的一举一动一样有价值。想象一下，如果知道客户在逛你的商店时的一举一动，则将可以更好的组织你的商店，并知道有多少机会可以销售他们更多的商品。网络日志提供了类似的信息。分析网络日志的活动并存储结果在数据集市来分析客户行为被称之为点击流数据仓库。

交叉参考：点击流数据仓库的更多信息请参考一本优秀的图书《Clickstream Data Warehousing》，作者是 Mark Sweiger, Mark R. Madsen, Jimmy Langston, and

Howard Lombard，Wiley 出版社 2002 年出版。

从数据建模的角度看，点击流数据集市可能相对数据仓库中的其它主题没有什么挑战。然而，ETL 过程与其它可能遇到的数据源有很大的不同。差别在于点击流的源是基于文本日志的，这些日志又必须和其它源系统集成。还好，基于文本日志的格式是标准化的。这个标准由世界网络协会（W3C，World Wide Web Consortium）维护。

W3C 通用和扩展格式

即使网络日志的格式是标准化的，它的格式和内容仍然有很多不同。Web 服务器的操作系统和控制日志内容的参数设置会影响写入日志的内容。如果不考虑操作系统，网络日志有一些通用的字段，一般包括：

- 日期。这个字段有一个通用的日期格式——一般为 dd/mm/yyyy。如果在 ETL 过程中时区被调整了，则必须合并日期和时间并一起做调整；否则，可能会有一天的误差。可以在加载时再把它们分开。
- 时间。这是网络点击的时间。格式是 HH:MM:SS 并且一般设置为格林威治标准时间（GMT）。然而，时区也可以更改。确定知道 Web 服务器设置的时区，这通常也包括为数据集市把时间转换成本地时间。
- c-ip。这是用户的互联网服务提供商（ISP）的 IP 地址。它是一个标准的 IP 地址，可以用来为域名系统（DNS）查找并推算用户来自哪里。使用 c-ip 几乎没有什么可信度，因为众所周知它是没有规则的，比如 AOL（拥有最大量用户的 ISP）给上百万的用户同一个 IP 地址并显示所有他们的用户都来自美国的同一个州，即使他们可能实际上来自全世界各个角落。
- 服务名。这涉及到运行在客户计算机上的互联网服务，比如，w3svc1, w3svc2, w3svc3 等等。这个字段识别来自多网站或域环境的站点和日志。这个字段对单一站点环境通常设为关闭。
- s-ip。这是服务器的 IP 地址。它是标准的 IP 地址格式。在一个多网络服务器的环境中用来识别某个 Web 服务器是很有用的。它也可以用来做负载平衡分析。

- cs-method。这个字段中只有两个可能值：POST 或 GET。一般只有 GET 的记录存储在点击流数据集市中。
- cs-uri-stem。这是资源可访问性（即 HTML 或 ASP 页面请求）。
- cs-uri-query。这是客户发来的查询。这个字段包含高度客户化的且非常有价值的数据。我们把这个和 cookie（稍后将讨论）称为网络日志的重要信息。这个字段用 label=value 表示并用符号（&）作分隔符，当然也可以用其它的符号。Cs-uri-query 的更多分析在本章的稍后的”名称值对”部分讨论。
- sc-status。这是 HTTP 状态，比如，302（重定向），402（错误），及 200（ok）。HTTP 状态代码的完全列表可以在网络上搜索 HTTP 状态代码来查找。我们建议预先加载所有可能的代码和描述到 HTTP 状态维。
- sc-bytes。这是服务器发送的字节数，一般当作点击的一个事实。
- cs（用户代理）。这是客户使用的浏览器类型和版本。用户代理以及日期和时间可以用来唯一的确定一名访问者。

更多关于标识网站唯一用户的信息，可以参考《数据仓库工具箱：建立基于 web 的数据仓库》，作者 Ralph Kimball and Richard Merz ，Wiley 出版社 2000 年出版。

- cs（Cookie）。这是 cookie 发送或接收的任意内容。和 cs-uri-query 一起，这个字段是 web 日志中的另一种重要信息。Cookie 是高度可自定义的且非常有价值。它可以清楚的识别用户以及关于用户会话的许多其它特征。
- cs（参考）。这是指引用户访问当前站点的网站的 URL 说明。

以下的字段出现在 W3C 扩展格式中，但这不是全部的列表。如果想要了解环境中所有字段的列表，可以参考 web 服务器文档或 W3C 网站（www.w3c.org）。

- 服务器名称。这是日志产生的服务器的名称。这与 web 服务器的 IP 地址 s-ip 应该是一一对应的。
- Cs-用户名称。这是用户名称并且只有当方法为 POST 时才有值。

- 服务器端口。这是客户端连接的端口号。
- 接收字节。这是用户接收的字节数目。
- 消耗时间。这是这个动作消耗的时间。
- 协议版本。这是客户端使用的协议版本，比如 HTTP 1.0, HTTP 1.1。

Web 日志中的名称值对

Web 日志由各种不同内容的标准字段组成。对于其中最主要的部分，无需太多的转换逻辑就可以导出 web 日志的内容。尤其对于日期、时间、c-ip、服务名称、s-ip、cs-method、cs-uri-stem、sc-status 和 sc-bytes 来说更是这样。然而，例如 cs-uri-query 和 cs（cookie）这样的字段根本就不是标准的。事实上，两个不相关的网站在这些字段中几乎不可能有相同的内容。cs-uri-query 和 cs（cookie）包含自定义的名称值对来获取一个交互的特定属性，这些交互对业务来说很重要。

cs-uri-query 一般包含关于交互的详细信息，比如页面上提供的产品。请看下面这个 cs-uri-query 例子：

/product/product.asp?p=27717&c=163&s=dress+shirt

看一看这个查询字符串并注意以下的部分：

- /product/——查询字符串的开始部分表示要执行的程序所在的目录。目录总是在前面和后面加一个斜杠’/’。在这个例子中，product.asp 程序在产品目录下。如果程序在根目录下，在程序名称前加一个单斜杠。
- Product.asp——这是产生 web 页面的可执行程序文件。通用的可执行程序包含扩展名：.asp(动态服务页面)或.jsp（java 服务页面）。程序文件紧跟在目录后面。
- ？——问号表示后面的参数要传递给程序文件。在这个例子中，有三个参数：p，c 和 s。

在问号之前的查询字符串是标准的，问号后面是程序的自定义参数的位置。网站开发者可以为每个程序文件定义不同的参数集合。在 web 日志的名称值对中可以获得这些参数。在这个例子中，你可以看到三个参数，之间用符号（&）

分隔。

- p 表示产品号码
- c 表示产品类别号码
- s 表示用户输入用来查找产品的搜索字符串

符号（&）是 web 日志中最通用的参数分隔符，但并不是唯一的。确认在分析阶段中扫描日志确实是可视化的，保证可以识别参数分隔符。

注意到在 s=参数 中搜索字符串写着 dress＋shirt。实际上，用户输入的是 dress shirt。由于 HTTP 不能处理空格，因此这个+是 web 浏览器自动插入的。当处理查询字符串中的文本型描述信息时必须非常小心。在用它们来查找或存储到数据仓库中之前，ETL 过程必须将文本型描述中的任何＋都用空格来代替。

这一部分的目标是揭示 web 日志的信息，给点击流数据集市项目开个头。再次强调，如果要深入了解或规划在不久的将来实施一个点击流项目，则应该重点关注这个部分前面提到的两本书中的任何一本。

4.10 ERP 系统数据源

流程检查

规划与设计：*需求/现状* -> *架构* -> 实现 -> 测试/发布

正如本节所述，ERP 系统的存在给 ETL 系统的规划和设计带来很大的影响。包括通过各种方式将 ERP 系统当作一个简单的固定长度的数据源，到把 ERP 系统当作包含所有组件(包括 ERP 系统)的数据仓库。我们在这一部分给出相应的建议。

数据流：抽取 -> 清洗 -> 规格化（也许） -> 提交

企业资源计划（ERP）系统的在建设时也面临着数据仓库正在面临的问题之，即集成异构的数据。ERP 系统设计为企业集成的解决方案，它使得企业的每个主要实体，比如销售、会计、人力资源、库存和产品控制都在同一个平台、数据库和应用框架上来完成。

和想象的一样，ERP 系统是非常复杂并且难以实施，往往需要数月或数年时

间来定制开发，因此它包含了详细的功能来满足运行一个特殊业务的所有需求。但正如不可能提供一个最完整的解决方案一样，虽然进行过很多努力，仍极少看到整个企业只用一个 ERP 系统来运作整个公司。

ERP 系统的复杂性人人皆知，而且由于它实际上是一个框架而不是一个应用程序，因此其数据模型往往很丰富，通常包含数以千计的表。此外，由于它们的灵活性，支持 ERP 处理的数据模型非常难以确定。市场上主流的 ERP 系统包括 SAP、PeopleSoft、Oracle、Baan 和 J.D.Edwards 等。

由于由无数的表和属性组成，以及通常 ERP 实施的复杂度，因此把这些系统和任何其它交易源系统一样进行利用是错误的。随便进行系统和数据分析是费时费力的，并且为以后出错带来了更多的可能。如果数据仓库的数据源是一个已存在的 ERP 系统，最好是获得对这个特定 ERP 系统的底层数据库结构非常了解且以及对应用的业务对象具有丰富经验的人的帮助。

作为帮助，许多主流 ETL 厂商现在都提供 ERP 适配器来连接主流 ERP 系统。如果以 ERP 系统为数据源，那么就应该充分利用这些适配器。它们可以帮助你探索这些系统的元数据并更好地理解这个应用。

对 SAP 的特殊考虑

因为 SAP 在 ERP 市场的支配地位，经常有人问起在数据仓库中关于 SAP 角色的问题。以下我们的最原始的建议。

你可能已经听过这样的观点，从决策支持的角度来看，SAP ERP 像一个黑洞：数据流进去了，但没法将信息取出来。为什么？

一个 SAP ERP 实施好像是有一个包含上万张物理表的数据基础，几乎没有任何 DBMS 定义的表间关系，并且实体和属性名称采用的是德文缩写的！因此，SAP RDBMS 对实际的实施来说，是专用的和不可理解的。SAP ERP 产生一个广泛的操作型报表库，但这些一般难以满足大多数业务部门全面的决策支持的需求。这并不是设计的缺陷。SAP 的 OLTP 数据结构对基本业务报表需求缺乏支持，比如历史交易和主要数据情况的查询，可理解的和简单的数据结构浏览，以及灵活的查询执行功能。

一些早期实施了 **SAP** 的企业试图在它们的数据仓库中创建 **ERP** 主题域来让它们的 **ERP** 中的操作型数据保持独立。实事证明，几乎没有特定的工具可以帮助它们完成这个难以实现的任务，许多努力并没有获得显著的效果。

为了满足这一不断增长的需求，**SAP** 设计了一个 **ERP** 自身的决策支持的扩展，叫做业务信息仓库（**SAP BW**）。**SAP BW** 的早期版本相当简单，并且主要由 **SAP** 特定的 **ETL** 提供的专用 **OLAP** 资料库组成，因此缺少许多同期的数据仓库所拥有的基本结构元素。最新发布的 **SAP BW** 已经有了显著的增强，现在已经包含了许多同期数据仓库的核心原则和结构：更好的支持非 **SAP** 数据源以及主流数据资料库（集结区，**ODS**，数据仓库，维度数据集市和 **OLAP** 立方体）。这些资料库中的一部分支持向第三方报表工具的开放访问。

SAP BW 提供了有吸引力的价格和，以此来吸引 **CIO** 们的注意力。与此同时，同期的数据仓库架构师可能被要求考虑在企业数据仓库视图中 **SAP BW** 的角色。

在下表中，我们介绍了在整个企业 **DW** 战略中针对不同的 **SAP BW** 角色的 **pros**，**cons** 和建议。我们认识到这是一个快速发展的领域，会有许多变化，这其中几乎没有完全满足所有需求的解决方案。**BW** 和 **ETL** 工具的功能将逐步改变，但我们还是希望我们的评估过程对你有用并且可延伸到特定的问题中。

BW 作为企业数据仓库	利用 SAP BW 作为企业数据仓库策略的基本核心
	我们不推荐使用 BW 作这个角色。使用 BW 作为一个企业 BI 结构的中央元件在目前结构是是不稳固的。正如这里所写的，BW 没有为非 SAP 数据的清洗/集成或非 SAP 分析的发布提供相应的能力。打包 BI 方案也趋向于减少通过分析能力比较不同的机会。尽管如此，那些没有把 BI 当作一个比较不同策略的领域并且他们的报表需要以 SAP 为中心并对其评价很高的组织仍可能考虑这样使用 SAP BW。
BW 在维度数据仓库总线	利用 SAP BW 作为一组数据集市的集合，这些数据集市在广泛分布的维度数据仓库总线结构中进行相应的操作。
	虽然这可能有一天会成为利用 BW 的一种合适的方式，我们发现（正如这里所写的）BW 并不完全适合在一个广泛的维度数据仓库总线结构中当作 ERP 中心数据集市角色。当我们写这本书的时候，SAP 宣布了一个叫做 Master Data Management 的功能，声称可以处理产品说明和客户的交叉描述问题功能。这种独特的功能能够创建企业级一致的维度还是合并外部一致的维度到 ETL 过程并且应用到相应的事实表，还不好判断。利用跨越主题域的一致维度和事实是维度数据仓库总线结构的核心宗旨。尽管如此，已经拥有了维度数据仓库和 BW 的 IT 组织可能会选择扩展 BW 来利用外部一致维度这一任务，这样就允许它在总线结构即插即用。警告：通过 SAP 升级维护这些扩展可能不是一个简单的任务。

ETL 和集结区	利用 SAP BW 作为将 ERP 数据导入下游维度数据仓库的通道和集结区。
	这个选项刚出现的时候相当引人注目，因为它的目的是简化和减少工作量来将 ERP 数据导入维度数据仓库，并且同时也为 SAP 专用报表提供标准 BW 报表能力。但是当将这个选项与利用带有很好的 SAP 适配器（见下一节“放弃 BW”）的专用第三方 ETL 工具这个选择相比，它可能就不再是最佳方案了。用 SAP 作为数据源，通过 ETL 工具为数据仓库导出大量附加结果到表集合，一般可以提供更好的可控性、灵活性和元数据的一致性。我们建议这个在 SAP 中的 ETL 系统的实施选项只有当组织有不成熟的 ETL 小组并且时间很紧近且需要出 ERP 报表时才选择，谁能在容量狭小的 SAP BW 报表中发现价值很高的信息。不如认真寻找带有专用 SAP 适配器的成熟的 ETL 工具。见下一节“放弃 BW”。
放弃 BW	利用大多数优秀的 ETL 工具提供的 SAP 适配器来执行完全集成 ERP 主题域到独立的企业维度数据仓库总线结构中。
	我们绝对相信购买胜过自己做，这是非常合适的。但是考虑到 SAP 在创建或发布一致维度或利用外部一致维度方面跟踪记录的缺陷，这是我们建议的默认 BW 结构形式。

4.11 第 3 部分：抽取变化数据

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

在初始化加载时，捕获源数据中的数据内容的变化不是很重要，因为很可能会导出整个数据源或其中的一部分。然而一旦初始加载完成，捕获源系统中的数据变化立即变成非常重要的任务。如果等到初始加载完成才开始规划数据变化的捕获技术，则将遇到很大的麻烦，捕获数据变化绝不是一个简单的任务，必须制定相应的策略来在项目中捕获源数据中不断增长的变化。

ETL 小组在后续的加载过程中负责捕获数据内容的变化。需求由用户提出，而这些需求的实际可行性由源系统的 DBA 小组决定（如果运气好的话）。更多的时候并不是这样，需要针对特定的情形做一些研究来决定最有可能的增量加载策略。在这一节，我们提供几种选项并讨论每一种的优缺点。当然，不必为每一种情形选择所有这些技术。选择在整个项目过程中最能满足每个 ETL 需求的方法。

为识别源系统中变化的数据制定适当的策略可能要进行一些分析工作。当分析源系统时，绝对不要假设所见即所得。在许多案例中，还有许多未加使用的甚至是禁止使用的审计列，或更糟糕一些的是完全不一致的列用法。确保为增

量加载过程分配了足够的分析时间来调查和决定捕获数据内容变化的最佳方法。

监测变化

当管理者谈论数据仓库维护时，他们最经常讨论的是保持数据最新，也就是数据真正反映了公司业务系统的状态。捕获源系统内容的变化对一个数据仓库的成功至关重要，数据内容的维护则依赖于增量加载过程。有几种捕获源数据变化的方法，每一种在它们相应的环境中都是有效的。

使用审计列

在大多数情形下，源系统都包含审计列。审计列附在每个表的最后用来存储记录增加或修改的日期和时间。审计列一般通过数据库触发器产生，当插入或更新记录时自动生成。有时，为了提高性能，这些列由前端应用程序而不是由数据库触发器产生。当这些字段不是由数据库触发器以外的任意其它方法加载时，必须要特别注意它们的完整性。必须分析并测试每个列以确保其是表示变化数据的可靠的数据源。如果发现任何 NULL 值，则必须另寻替代办法来监测数据变化。

阻止 ETL 过程使用审计列的最常见情形是当这些字段是由前端应用程序产生的并且 DBA 小组允许后台脚本修改数据时。如果是这种情形，则将在增量加载过程中面临很高的错过变化数据的风险。最小化这一风险防范措施之一是规定所有的后台脚本由一个质量保证小组来验证，在验收之前坚持并测试由脚本产生的审计字段。

一旦确信审计列是可靠的，则需要制定一个策略来利用它们。有各种各样利用审计列来捕获变化数据的方法，并且所有的方法都拥有同样的逻辑目标：比较每个记录最后修改的日期和时间与上次加载后的最后的日期和时间，取它们中较晚的值。

我们已经发现一个利用源系统中审计列的有效方法。实际上，这个过程就是从创建日期和最后修改日期列中选择最后日期和时间。有些最后修改列是随着插入或更新记录的变化而修改的，其它最后修改列在记录插入时是 NULL，只有当这个记录更新时才插入相应日期值。当最后修改日期没有填充时，为了不丢失新记录必须给定任意一个非常早的日期作为默认值。以下代码可以帮助解决 NULL

修改日期：

```
select max(greatest(nvl(create_date,'01-JAN-0001'),
nvl(last_mod_date,'01-JAN-0001')))
```

对事实表中的行只插入不更新的情形，可以简单的从源系统选择创建日期和时间大于上次加载的最后日期和时间来实现，从而忽略最后修改日期列。

由于事实表和维表可以来源于许多不同的表和系统，并且由于事实表只包含外键和度量，因此不能直接存储审计日期列到事实表中。在每次导出的时候，需要建立一个 ETL 的最后修改表来捕获每个源表和在源系统审计列找到的最大日期。如果事实表需要对行数进行审计统计，考虑使用第 4 章所讲述的建立一个审计维的方法。

数据库日志的获取和提取

日志获取是在调度点（一般在深夜）对数据库重做日志进行有效的快照并把它作为对 ETL 加载过程中所关心的表产生影响的数据源。提取包含重做日志的登记，并从中捕获相应事务数据。抓取日志可能是所有技术中最笨的。事务日志经常会溢出，这意味日志被写满并阻止新事务的产生。当这种溢出发生在一个生产环境时，负责的 DBA 最快的反应会是清空日志的内容以使业务操作重新恢复。但是当日志清空时，里面所有的事务也都丢失了。如果已经用尽了所有的其它技术，然后发现日志抓取是为寻找新的或变化的记录的最后求助的手段，那么就试图说服 DBA 创建一个特殊的日志来满足这一特殊需要。大概只需要源数据库成百张表中很少的某些交易表，当插入或更新这些表时触发动作写入专用日志中。

如果想继续日志提取，我们建议调查市场上可用的 ETL 工具来寻找验证有效的方案而不是试图从抓取开始手工写这个处理过程。许多实时 ETL 方案提供商都利用了日志提取技术。

按时抽取

选择所有创建或修改的日期等于 SYSDATE-1 的行，即获得昨天的所有记录。看起来很不错，对吗？其实是错误的。纯粹的按照时间来加载记录是大多数刚开始进行 ETL 开发的人常犯的错误，这种处理方式极其不可靠。

当从中间处理失败后重启时，基于时间的数据选择会加载重复的行。这意味着无论什么原因使处理失败都需要人工干预和数据清洗。其间，如果某天的加载处理没有运行并错过了一天，存在的风险是错过那天的数据将永远不会进入数据仓库中。因此除非 ETL 处理非常简单并且数据量特别小，否则不要纯粹基于时间加载数据。

排除处理

排除处理在集结区保存了每个上次导出数据副本为将来所用。在下一次运行过程中，整个源表被拿到集结区与最后一次处理留下的数据副本进行比较。只有不同（增量）部分会送到数据仓库。虽然这不是最有效的技术，但排除处理是所有捕获变化数据的增量加载技术中最可靠的。因为这个处理进行了逐行的比较并查找出变化的行，它原则上不可能遗漏任何数据。这种技术对找到那些从源数据中删除的行也有帮助，而这些删除的行往往被其它的技术遗漏掉。

这种技术可以在数据库管理系统的内部或外部完成。如果选择使用 DBMS，则为了提高效率必须成批的将数据加载到集结数据库来处理。

初始和增量加载

创建两个表：previous_load 和 current_load。

初始化处理批量加载到 current_load 表。因为初始加载过程中变化检测是无关的，因此数据连续被转换并加载到最终目标事实表中。

当这个处理完成后，就删除这个 previous_load 表，并把 current_load 表重命名为 previous_load 表，然后创建一个空的 current_load 表。由于这些任务都没有涉及数据库日志，因此速度会非常快！

下一次加载过程运行时，这个 current_load 表就填充了数据。

选择 current_load 表 MINUS（减去）previous_load 表，转换并加载结果集到数据仓库中。

完成后删除 previous_load 表并再次重命名 current_load 表为 previous_load。最后，创建一个空的 current_load 表。

由于在数据库管理系统中 MINUS 是非常慢的技术，因此最好使用 ETL 工具或

第三方应用程序来执行例行的排除处理。

抽取的技巧

当进行抽取处理时考虑以下几点：

- 强制列索引。和 DBA 一起工作来确保源系统中所有在 WHERE 语句中的列都有索引，否则将可能引起对整个生产数据库的全表扫描。
- 获取需要的数据。优化的查询会正确返回需要的数据。不应该获取整个表，然后在 ETL 工具中再过滤掉不想要的数据。有一种情况可能不遵守这个规则，就是交易系统的 DBA 拒绝对你的查询需要限制返回行的列索引时。另一个例外是当必须下载整个源数据表来查询增量时。
- 谨慎使用 DISTINCT。DISTINCT 语句非常慢。在抽取查询中执行 DISTINCT 和在 ETL 工具中对结果进行聚合或分组两种方式如何平衡是一个挑战，一般会随着数据源中重复百分比的不同而不同。因为有许多其它因素可以影响这种决定，所有我们所能建议的就是谨慎的测试每个策略，从而找出最有效的结果。
- 谨慎使用 SET 操作符。UNION，MINUS 和 INTERSECT 都是 SET 操作符。这些和 DISTINCT 一样都是非常慢的。可以理解的是有时这些操作符是不可避免的。一个技巧是使用 UNION ALL 而不要使用 UNION。UNION 执行和 DISTINCT 相当，处理缓慢，而 UNION ALL 的缺点是会返回重复行，因此都要谨慎处理。
- 根据需要使用 HINT。大多数数据库支持 HINT 关键字。可以使用 HINT 做许多事情，但最重要的是强制查询使用常规的索引，尤其当使用 IN 或 OR 操作符时这个功能尤其重要，因为这时往往会进行全表扫描而不是使用索引，即使有可用的索引存在。
- 避免 NOT。如果有可能，尽量避免不等于的限制和连接。无论使用关键字 NOT 还是操作符‘<>’，数据库此时也会选择扫描整个表而不是利用索引。
- 避免在 where 子句中使用函数。这是很难避免的一种，尤其是当包含日

期和类似日期的处理时。在犯下在 WHERE 语句中使用函数的错误前，先要用不同的技术进行试验。无论如何尽量尝试使用比较关键字而不是函数。比如：

LIKE 'J%' 而不是 SUBSTR('LAST_NAME',1,1) = 'J'

EFF_DATE BETWEEN '01-JAN-2002' AND '31-JAN-2002' 而不是

TO_CHAR(EFF_DATE, 'YYY-MON') = '2002-JAN'

抽取查询的目的是获得所有相关的自然键和度量。它可能和从一个表选择多个列那么简单，也可能和实际创建不存在的数据那样复杂，范围可能从很少几个表的关联到不同数据源的许多表的关联。在一个特定项目中，我们必须建立一个周期性的快照事实表来表示库存中每个产品的销量，即使在这个期间没有销售任何产品。我们不得不产生一个产品列表，获得所有产品的销售，并执行一个产品列表和产品销售之间的外关联，没有销售的产品的销售量默认为 0。

监测数据源中删除或覆盖的事实记录

如果没有删除或覆盖事件发生的通知机制，对于源系统中删除或覆盖了的度量（事实）记录如何发现可能会给数据仓库带来非常大的挑战。由于一般不可能重复抽取旧的交易记录，关于如何查找这些被删除的和修改的数据，我们能提供最好的方法如下：

- 和源系统的所有者商讨，如果可能，明确的通知所有删除或覆盖的度量。
- 周期性的检查来自源系统的度量的历史汇总并通知 ETL 人员有些内容变化了。当发现变化时，尽可能的钻取下去找到这个变化。

当识别出一个删除或修改的度量记录时，就可以使用前面所述的最后返回数据技术了。对于删除或修改事实记录的情形，不仅在数据仓库中执行删除或更新，我们还推荐插入一条新的记录来实现在事实表中消除或抵消初始放的值。在许多应用中，这将汇总报表事实得到正确的数量（如果它是添加的）并提供一种纠错发生时的审计跟踪功能。在这些情形下，它可能也方便的携带额外的管理时间戳来识别这些数据库行为发生的时间。

4.12 总结

在本章，我们深入探讨了 ETL 数据流中的抽取这一步骤。我们建议再回到刚开始的地方来确认抽取是非常有价值的部分！可以通过一个数据评估工具来决定做或不做这一工作，工具将告诉你是否数据质量满足了业务目标。

下一个大的步骤是准备连接到初始源数据到最终数据的逻辑数据映射。也许逻辑数据映射最重要的部分是应用在输入和输出之间的转换规则描述。当实际执行 ETL 系统时将会发生一些变化，因此应该经常检查并定期更新逻辑数据映射。如果维护的好，它可能是 ETL 系统中最有价值的描述。有时，如果由一个新人来了解做过的东西，他或她应该从逻辑数据映射开始。

本章的中心是带你了解将来可能遇到的各种各样的源系统。其中一些抽取复杂度是相当高的，当然，没有什么比实际经验更有价值了。

本章的最后部分介绍了只导出新数据、变化的数据甚至是已经删除的数据的方法。在后续的章节，我们将指出在这些情况下必须采取的特殊处理措施。

5 清洗和规范化

清洗和规范化在 ETL 系统中是最富价值的的步骤。当然，数据抽取和加载步骤绝对必需，只是它们做的仅仅是改变数据位置和改变数据格式。清洗和规范化实际上真正的改变了数据，并对数据能否用于预期的目标起决定作用。

在本章，我们将会跟您一起建立三份重要的可提交内容：数据评估报告(data-profiling report)，错误事件事实表(error event fact table)，审计维(audit dimension)。围绕着这三份提交内容，你可以建立一个功能强大的清洗和规范化系统。

清洗和规范化步骤会生成有效的元数据。回顾原始数据源时，这些元数据可以作为源系统的错误诊断报告。最终，仅仅通过改变源系统采集数据的方式就可以修正这些脏数据。记得我们说过"业务流程再造"么？

在清洗和规范化步骤中产生的元数据，将伴随着真实数据一起摆在用户桌面上，或者，至少它应该是这样的。ETL 小组必须使清洗和规范化的元数据可用，这也是引起审计维的原因。

请仔细阅读本章，这非常重要。本章作了一系列的努力为通常难以归类的问题提供了特定的技术和结构。本章篇幅比较长，也许你应该至少阅读两遍，但我们确信这将为你在 ETL 系统中构建数据清洗和规范化方面提供有用的指导。

如果你是 ETL 系统设计方面的新手，你可能会问"我该从哪里开始呢？"也许最好的答案是：先尽你所能做好数据评估分析。这对于你更清楚地认识你将要处理的脏数据或不可靠数据的风险大有帮助。借助于从数据评估中获得的认识，你就能够对问题进行分解，并且在设计简单的错误事件事实表和简单的审计维时有更多的自信。

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

本章按照下面四个与数据质量相关的话题逐步展开：

- 第一部分：设计目标
- 第二部分：清洗提交
- 第三部分：快照及其度量
- 第四部分：规范化提交

这是对于数据质量的自上而下的阐释。在目标部分，我们会督促着你实现"全面、快速、准确、清晰"地设计目标。清洗和规范化过程中我们更多关注的是约束关系与控制而不是数据之上的内容。在某种意义上，这是数据仓库小组不太熟悉的一个领域。在提交阶段，我们会介绍清洗子系统的主要的结构：错误事件表和审计维。我们还会推荐你学习附录 B 中关于 Jack Olson 的著作，用以针对事先的数据评估步骤的结果设计一个有机的结构（the results of your up-front data-profiling pass.）

再往下的过滤器部分，我们会在很多地方定义检查点和筛选器，用来对数据质量进行测量。使用过滤器，我们建立了一个统一的方案来捕捉数据质量事件并对这些事件采取相应动作作为响应。

定义：数据质量过滤器

在本章中，我们会反复提到数据质量过滤器。我们使用过滤器这个词既可以用来表示报表，又可以用来表示筛选。这样，数据质量过滤器在物理上可以被 ETL 团队看作是关于数据质量的状态报告，但同时也是一种防止脏数据通过的“墙”。

在第四部分，我们将描述规范化步骤的主要的提交内容：规范化的维和事实，以及它们如何被传递（handed off）。同时我们也建议了一些元数据方法，用以持续追踪组织所做的决策，以此来标准化维和事实。

最后的度量部分是针对特定数据质量问题的实施指南。与第三章中关于不同类型的抽取问题的详细描述一样，这些数据质量度量将是一个在其之上进行建设的合理的基础。

本章的内容大量的借鉴了 Jack Olsen 和 Larry English 这样的数据质量方面的顶尖级专家的成果。他们的简明而直接的数据质量度量技术在本章大量的使用，并

且成为创建有效的数据质量 ETL 流程的模版。

5.1 定义数据质量

首先我们应该在一些基础词汇上达成共识，集中在“精确性”。精确的数据意味着这样的数据：

正确的：数据的值和描述真实的反映了它需要表达的对象，并且这种表达是可信的。例如，某一位作者当前居住的城市叫 New Hope，那么，关于家庭地址的“精确的”数据应该包含正确的城市名称 New Hope。

明确的：数据的值和描述应该是只有一个含义。例如，在美国至少有十个城市叫 New Hope，而在宾夕法尼亚州只有一个城市叫 New Hope。那么，在“精确的”数据中，描述这个城市中的地址是应该包含城市名 New Hope 以及州名称宾夕法尼亚，这样的数据才是“明确的”。

一致的：数据的值和描述用一种不变的标识约定来表达其含义。例如，美国的宾夕法尼亚洲可能在数据中表达为 PA，Penn 或 Pennsylvania。为了满足一致性的要求，关于当前家庭住址的的“精确”数据应该只使用一种约定方式来表达州名称(比如使用全名 Pennsylvania)，并且坚持这一种用法。

完整的：关于完整性有两个方面。

第一点是要确保数据中为每个实例定义了（不能为空）特定的值和描述，例如，要确保所有应该有当前住址信息的记录实际上确实有值。

第二点是要确保记录的总数量是完整的，或者确保在整个信息流中没有在任何地方丢失记录。

与完整性相关的一个问题是数据中的缺失值的多种可能的含义。用“空”来描述的缺失值可能代表真实的数据未知，或者不可用。缺失值可以用空格、空字符串或其他自创的描述（比如，不知道或拒绝回答）来表示。

5.2 假设

在本章我们会对清洗和规范化发生的环境做一些基本的假设。首先，ETL 作

业流中有一些不同的断点，在这些断点的位置可以插入数据质量处理过程。在我们整体的 ETL 数据流模型中，有两个这样的点是显而易见的：

- 第一个里程碑是在数据抽取完毕之后——这意味着数据已经从多个数据源抽取出来并被放置在一个按主题组织的物理的或者逻辑的结构中。例如，来自各个数据源的客户信息被放入某个表、数据文件或内存结构中，无论原始的数据源情况如何不同，最终的这个目标结构的配置是相同的。这样的结构可用于把来自多个数据源的输入数据沉积并排列等待 ETL 处理。在这里几乎没有做任何数据清洗或整合；来自多处的数据只是简单的改变结构，等待下一步的处理。当然这并不意味着来自多个数据源的数据必须并行地进行处理，而只是说我们将使用某个独立于数据源的结构作为中间存储。当然，即使使用了针对每种数据源特定的中间结构和处理流程，我们所描述的数据质量技术也依然有效，只是在本章末尾描述的那个元数据的例子可能要作稍微的调整。即便如此，我们依然建议在这个阶段多运行一些数据质量处理流程，以此来获得组织中的数据质量状态的精确的视图(虽然数据依然是原来的样子)；同时使用这些数据质量处理流程，也可以清除那些毫无希望的缺陷数据，以免这些数据破坏你的数据清洗和整合处理。
- 第二个里程碑是在数据清理和规范化之后——这意味着数据已经成功地通过了 ETL 流程中所有的数据准备和整合步骤，为提交步骤的最终打包做好了准备。我们建议在这个阶段执行更多的数据质量处理，作为数据清洗和整合软件的安全过滤网。实质上，就是在正式提交它们之前，通过一系列的质量保证检查确保其提交内容的真实性。

为了简化起见，本章假定使用的是批量 ETL 处理，而不是实时或准实时处理。我们假设批量处理能够代表了大多数真实的 ETL 环境中所用到的技术，这样我们就可以只关注数据质量问题，而不用考虑复杂的实时 ETL。在第 11 章中会专门论述相关的话题。

5.3 第 1 部分：设计目标

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

这一部分描述了实现数据质量初始目标的相应压力以及 ETL 队伍必须要平衡的优先级冲突问题。我们建议了一些方法来实现这种平衡，同时明确地叙述了一种数据质量策略来满足重要用户参于者的需求。

了解关键参于者

数据质量子系统必须对数据仓库管理员，信息管理员，以及信息质量负责人等这些角色进行支持。虽然这些角色有可能在实际的人员之间以不同的方式分配，但这种角色的划分依然很有意义。

数据仓库管理员

数据仓库管理员负责数据仓库运行期间需要制定的日常决策，确保其是内/外部的数据源的精确反映，并确保这些数据是按照适当的业务规则和原则进行处理的。

清洗和规范化子系统应该支持数据仓库管理员和周边的业务环境，方法是通过提供一个应用到数据上（把数据装载到数据仓库中）的转换的历史，其中包括所有异常情况的详细的审计。

信息驾驶员

信息驾驶员的主要责任是定义信息策略。职责包括：对分析目标进行正规化定义，选择适当的数据源，设置信息生成原则，组织和发布元数据，并且为适当的应用进行限制。

清洗和规范化子系统应该支持信息驾驶员，方法是提供如下一些度量指标，包括操作性数据仓库对已确定的业务规则的日常满足情况，可能用于测试这些规则边界的数据源问题，以及可能造成针对特定应用来说源数据是否合适的数据质量问题。

信息质量负责人

信息质量负责人负责检测、修整和分析数据质量问题，他与信息驾驶员一起，制定脏数据的处理规则，设定可发布的质量门限值，并平衡完备性与快速性，以及在下一部分描述的纠正和透明之间的矛盾。

清洗和规范化子系统应该支持数据质量负责人，通过提供一系列的度量指标，用以描述数据仓库 ETL 过程中的数据质量问题的发生频率和严重程度。这些记录应该是一个完整的历史审计，通过这些信息，数据质量负责人可以清楚地评估随着时间的推移数据质量的改进的成功与否。

维表管理员

维表管理员负责创建和发布整个组织使用的一个或多个规范化的维表。可能会有多个维表管理员，分别负责不同的维表。维表管理员实现了对通用的描述标签达成一致，这些标签由整个数据仓库不同的使用者来访问。维表管理员还要创建和分配代理键并为每一次发布的维表分配版本号，以便在目标事实表环境中使用。当一个新的维表发布到数据仓库环境中的时候，必须同时复制到所有的目标位置，这样可以确保同步装载新版本的维表。维表管理员的工作非常明确：一个规范化的维表必须只能具有唯一的、一致的源。在本章的后半部分我们还会赋予维表管理员更多的职责。

事实表提供者

事实表提供者一般由本地的 DBA 担任，它拥有给定事实表的唯一实例。事实表提供者负责从不同的维表管理员处接收维表，将本地原始键转化为规范化维表中的代理键，并更新事实表使其对用户环境可用。如果有些维表记录的接收延时，事实表提供者可能将不得不在已有的事实表上作复杂的修正。最后。事实表提供者还要负责创建和管理聚合表，这些聚合表用于存储汇总记录以提高特定查询的性能。在本章的后半部分我们还会赋予事实表提供者更多的职责。

竞争因素

如图 4.1 所示，数据质量系统的目标可以用这四个相关的优先级来描述。

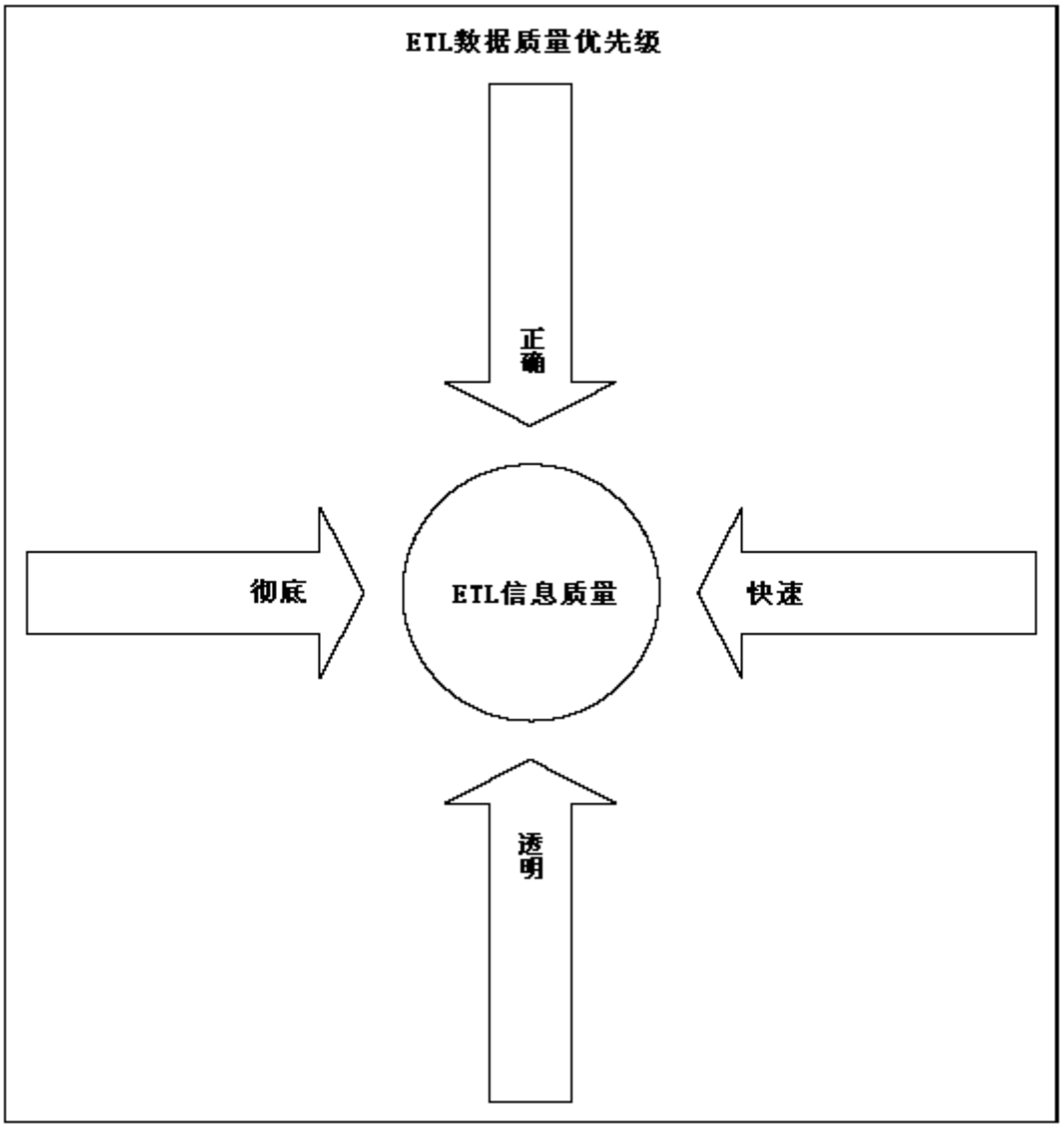


图 4.1　数据质量优先级

确保彻底

数据清洗子系统必须保证在检测、纠正和文档化发布给业务环境的信息质量方面的彻底性。最终用户希望能够把数据仓库作为一个可信赖的信息来源，在此之上来构建他们的管理度量指标、策略和规则。

确保快速

整个 ETL 过程流必须能够在越来越短的时间里处理不断增长的数据量。一些最新的也是最有趣的客户想要访问的信息非常的详细和个性化（比如 web 点击率）这会导致海量的数据被装载到数据仓库。

确保正确

事实上，在源系统或尽量接近源系统的位置改正数据质量问题当然是提升组织信息资产的最佳方式，这会降低高昂的成本，并降低劣质数据的产生机会。但是，现实情况是，大多数的组织并没有建立正规的数据质量环境和信息质量负责人制度，在这种情况下，数据仓库小组是最有可能第一个发现困扰组织多年的质量问题的人，那么，理所当然的，数据仓库小组应该担当起矫正这些问题的责任。

确保透明

数据仓库必须暴露出错误，并对那些损害组织中数据质量的系统和业务实践加以注意。这些发现最终会导致业务流程的再造，而这又会提升源系统和数据项过程的水平。试图在源系统掩盖数据质量问题的“英勇行为”会造成这样一种结果，修复的代价远远大于问题本身。

平衡冲突的优先级

很显然，对于清洗系统来讲，完全同时达到这些目标是不可能的。必须要作适当的平衡，这反映了不同情况下的优先级。

完备性与快速性

数据质量 ETL 无法在快速性和完备性方面同时达到最优。相反，我们试图在图 4.2 的指数关系曲线上找到那个合适的点，来实现我们的平衡。

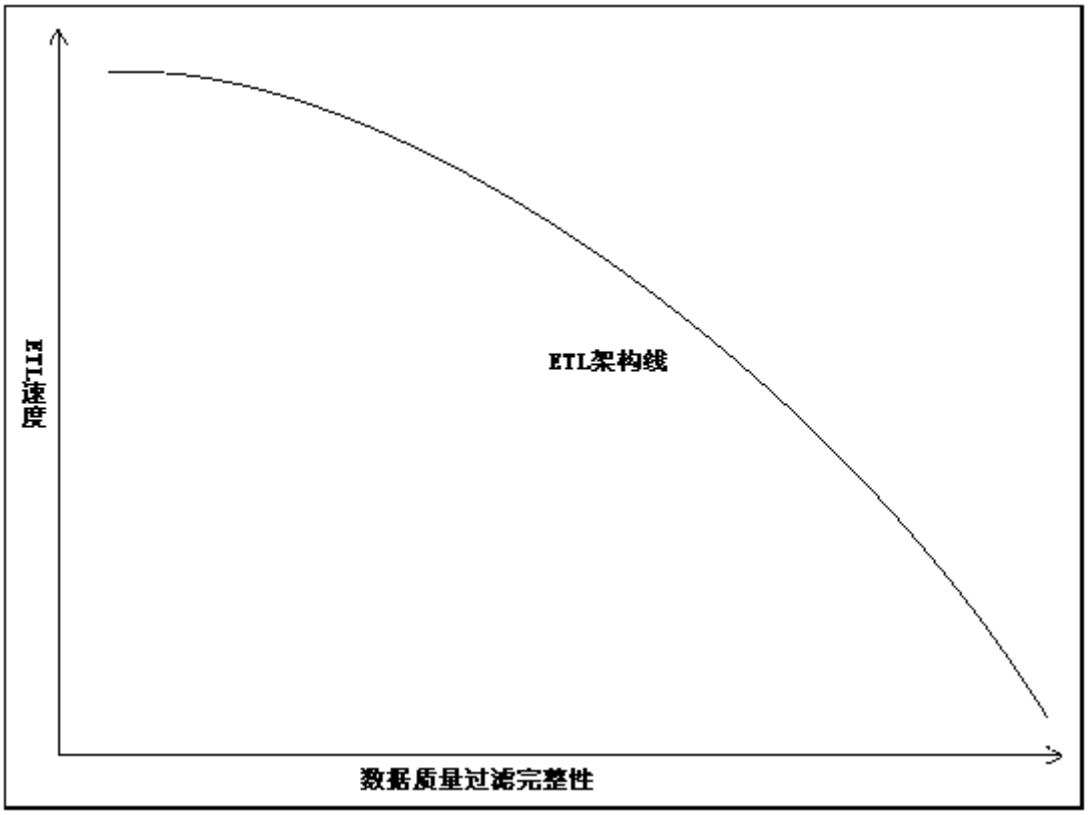


图 4.2　完整性对比速度

一种潜在的有启发意义的实现这种平衡的方法是通过粗略回答一些待建数据仓库中的数据质量方面的潜在问题来找到这个平衡点。比如：

- 在什么点数据变得过时？

以及

- 保证这些数据的正确性有多重要？

如果数据仓库小组必须要作出决定，比如，在数据质量方面更高程度的自信和发布周期不能多于一天之间，到底该选择哪个？又比如，某个数据仓库每天发布数据，可能会选择画费一整天的时间的延迟来获得更高的数据质量，方法也许是通过更多的差异检验统计或者数据标准化和匹配，或者甚至采用手工的检查或审计。如果周一的交易数据因此而在周三而不是在周二发布，是否是可被接受的替换方式？回答这样的问题并不容易。

校正与透明

数据清洗过程通常用来改正脏数据，但同时会给组织提供一份不加掩饰的原

始的视图。这里实现适当的平衡是非常必要的：完全透明的系统会产生一个功能不足的商务智能系统， 使得无法深入分析；而过分纠正的系统会掩盖操作上的不足，从而减缓组织的进展。

解决的办法是对不同的错误建立一个可判断的原则界限，这些错误由清洗过程进行纠正或者标识，并生成一个容易使用的审计工具（审计维），用来如实记录那些修改、标准化以及错误检测和数据再造组件的基本规则和设想。

从制造业的质量管理理论中学习

制造业的质量管理已经有至少 30 年的发展历史，最基本的理论来源是 W. Edwards Deming 的全面质量管理(TQM)架构。在考虑数据质量问题时，Deming 的关于管理 TQM 的 14 个要点同样值得思考，虽然这已经超出了本书所论述的范围。但是也许 Deming 的主要观点是生产质量需要整个组织的所有部门都要为质量问题负责，绝不仅是装配线末端的单独一个检查员的任务。

数据质量管理可以从制造业的质量管理中借鉴很多东西，在这方面重要的一个步骤是是 IT 组织中出现了集中控制的数据质量小组。关心数据质量问题的数据仓库小组成员不能独立于数据质量小组单独采取行动。在这一章中我们定义的过滤器可以补充数据质量小组使用的其它过滤器以及评估能力。这将从所有规格数据质量整合结果到一个完整的数据库，而不仅是数据仓库。大多数来自 ETL 过滤器的问题结果都将是要求改善源系统，而不是要求做更多的清洗。而所有的这些改善源系统数据质量的需求都需要由数据质量小组来完成。

阐述策略

图 4.3 所示的是一种方法，用于把数据仓库项目中面对的数据质量挑战进行分类，以及把由 ETL 数据质量子系统处理的内容分隔开来。

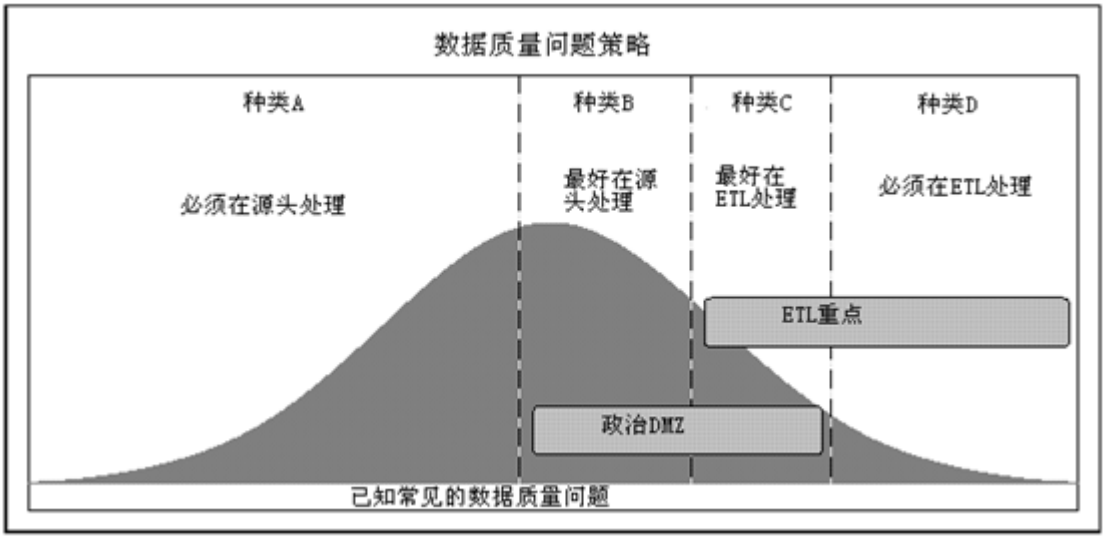


图 4.3 数据质量问题策略

- 不论甚么原因导致的 A 类问题,都必须简单地在数据源进行处理。例如缺失的客户投诉主题或者录入到字段中的主观取得的客户对销售电话接受能力的假信息。还没有什么技术方法来获取或者再造这些信息。它们要么从数据源被正确地获取，要么被遗失。当处理这些 A 类数据质量问题时，清洗子系统应该把他们看作是数据源的缺陷,把所有的明显的假信息从原始报表和分析维和事实中删除，并清楚地标明这些信息是缺失或者伪造，从而使管理工作直接集中在源系统的问题上面。在大多数的数据仓库项目中，大部分数据问题都属于这类问题，所以必须要检测数据质量问题，并与最终用户群体做清晰地沟通。

- D 类(我们先跳过来) 数据质量问题只能在 ETL 系统独自解决。例如来自独立的第三方的数据提供者的缺失的和不完整的信息，这可以通过集成完全修正，另外还有来自固定的操作型源系统的坏数据的修正。D 类问题在大多数数据仓库项目中相对很罕见，ETL 系统被授权创建这些值来来修正数据缺陷，但是必须要保证它的策略和行为能够通过描述的和完善的元数据对用户可见。

- B 类问题应该在数据源处理，即使有一些创造性的方法能推断或再造那些丢弃的信息。A 类问题和 B 类问题的分界线是技术方面的，而不是策略方面的。如果一个给定的数据问题能很有把握地在技术方面解决，那它很明显地是属于 A 类问题范畴。

- 因为种种原因，C 类问题最好被定位于数据质量 ETL 过程处理，而不是在于数据源。同样，C 类和 D 类的界限也是技术上的并非策略上的。如果一个给定的数据质量问题能被合理地数据源在处理，它很明显也是属于 D 类范畴。

经过划分和解决我们的数据质量问题，我们发现真正模糊的唯一界限是在 B 类和 C 类之间：从技术立场上来看，问题既可以在数据源可以在 ETL 系统进行处理。这叫做策略上的 DMZ(非军事区)。

5.4 第 2 部份：清洗提交报告

改善数据质量的一项重要任务必须以严格的测量为基础。这应该包括保证你发现的各种数据质量问题类型的记录的精确性，何时发现，看到了什么，以及结果是什么。此外，你需要能够回答来自数据仓库管理员，信息驾驶员和数据质量负责人的关于你做的处理以及发现的数据质量内部的相关问题，比如：

- 数据质量正在逐渐好转或是变得更糟？
- 哪一个数据源系统会产生最多的/最少的数据质量问题？
- 在整个细查数据质量问题的过程宫，有没有有趣的模式或者发现一些趋势？
- 数据质量水平和作为整体组织的效率之间有没有任何明显的相关性？

也许数据仓库管理员会问：

- 在我的 ETL 过程中，哪个数据质量过滤器消耗了最多/最少的时间?
- 对于那些不再数据中出现的已发现的问题类型，是否可以去掉它们的数据质量过滤器？

数据清洗子系统在整个 ETL 过程流中跟随在抽取步骤之后。下面将要讨论的三个主要提交报告包括：

- 数据评估结果
- 错误事件表

- 审计维

数据评估提交报告

数据清洗在构造 ETL 系统的第一个步骤之前就要真正开始。我们已经强烈地建议在开始计划和设计阶段之前执行充分的数据源的数据评估分析。好的数据评估采用一种特定的元数据资料库的描述方式：

- 结构定义
- 业务对象
- 域
- 数据源
- 表定义
- 别名
- 数据规则
- 数据值规则
- 需要处理的问题

数据评估不仅仅是对你的原始数据源的一种好的定量评估方式，这种输出还应该对下面描述的两种可提交的内容有很大的影响。Jack Olson 的书（数据质量：精确维）中附录 B 中有一个完事的列表，内容是前述列表的扩展子类，他们应该提前通过数据评估分析来创建，以此来组成元数据资料库的基础。

清洗提交报告#1：错误事件表

第一个主要的数据清洗的提交报告是一个叫错误事件表的事实表和一组维表。这一提交报告组织成一个维度数据模型,也就是一个维度的星状模型(见图 4.4)。

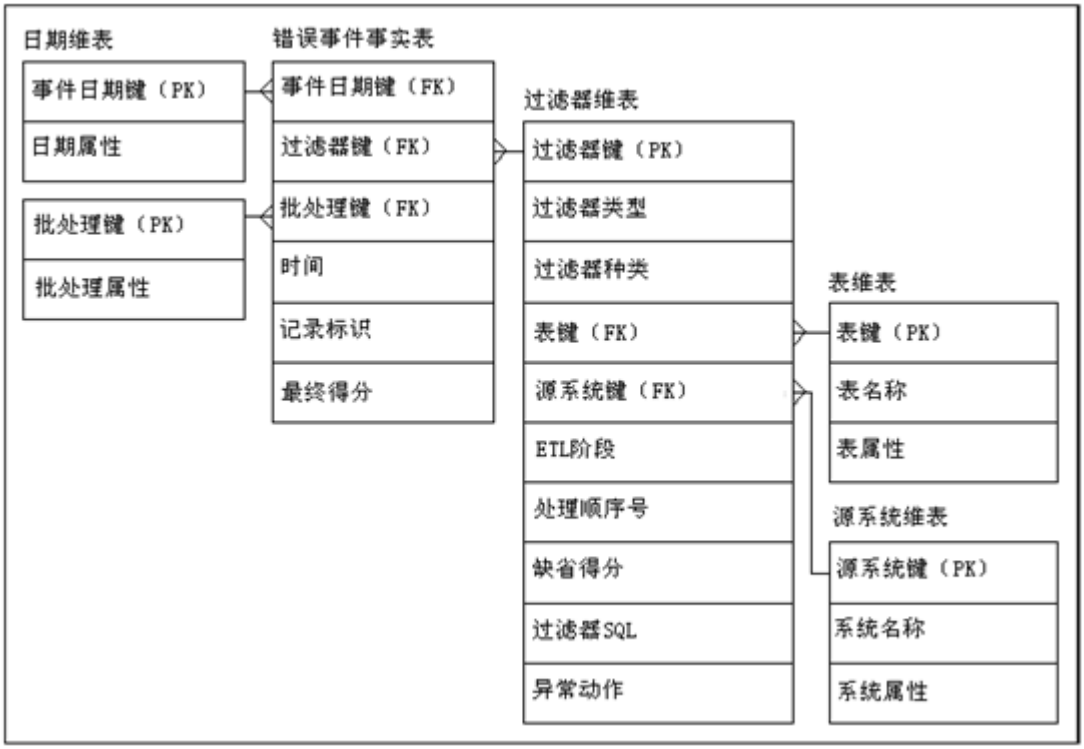


图 4.4　错误事件表结构

每一个被数据清洗子系统发现的数据质量错误或问题都被记录在错误事件事实表中。换句话说，这一个事实表的粒度是每个数据质量检验的每一个错误实例。记住一个质量检验是一个过滤器。所以，如果你运行 10 个不同的针对一些数据集的过滤器，且每个过滤器发现 10 个缺陷记录，总共就有 100 个记录会被写到错误事件事实表。

事件日期是一个标准的维，表示为日历日期。日期时间被从午夜以后按秒数记录在事实表里，以一个整数表示。

批处理维为整个批处理过程的每一次执行创建一个记录，并且通常都包含有意义的时间戳,以及处理的记录数量。

过滤器维包含描述每个已应用的数据质量检查或过滤信息的内容。它不是一次特定运行的描述(那是事实表记录的东西)，而是过滤可以做什么以及应用到哪里。它的属性之一 ,缺省的严重程度得分，为可能遇到的每个错误类型都定义一个严重程度值。 这些对严重程度评分很多作为最后严重程度得分错误事件事实表的基础。例如当海量数据累积起来后，最后的严重程度得分可能是比其缺省值得分高。

过滤器维的属性如下：

- ETL 阶段描述在整个 ETL 过程中数据质量过滤器被应用的阶段。
- 处理顺序编号是一个基本的调度/依赖设计,用于通知整个 ETL 主过程该过滤器的顺序号。在同一 ETL 阶段，拥有相同的处理顺序编号的数据质量过滤器可以并行运行。
- 缺省的严重程度评分即为应用到每个异常的错误严重程度得分，这些异常在压力处理规则之前由过滤器发现。这一缺省评分将增加或减少在事实表中的最终的严重程度得分。
- 异常动作属性将告诉整个 ETL 过程在发现这种类型的错误时是否通过该条记录，或者拒绝该记录，或者停止整个 ETL 过程。
- 过滤器类型和种类名称用来按主题把相关的数据质量过滤器分组，例如完整或者验证或者越界。
- 最后， SQL 语句捕获用于执行数据质量检验的实际 SQL 片断或者过滤 SQL。如果可行，这 SQL 应该返回数据质量过滤器捕获的所有行的唯一标识符集，这样就能在错误事件事实表中插入新的记录。

为了生成报告,把每个过滤器与其细查的表或一组列相关联是很非常有用的，这样信息质量负责人就能运行确定数据质量问题发生处的报表，并长期跟踪这些问题。这就是在过滤器维中定义表外键的目的。

源系统维用来识别有缺陷数据的数据源。因为数据质量过滤即可以运行在属于单一数据源的集结数据上，也可以运行在那些从很多数据源提取出来的数据，因此错误事件可以与特定的（虚拟的）集成的数据源系统相关联。

那些有问题记录具有唯一标识符，从而可以让错误事件直接地被追踪到问题记录。这些唯一标识符在事实表中表示为退化维，它是由 ROWID 或是其他指向可疑记录的指针构成的。注意这个设计还有一个隐含的功能就是维护过滤器维表的标识和真实记录间的参考完整性。如果你删除了真实的记录,过滤器记录将会变成一个“孤儿”。 过滤器种类字段只是被用于分类那些被过滤器发现的错误类型。可能的数值包括： 不正确的，模棱两可的，不一致的和不完整的,这使得分

析师可以对错误事件进行有意义的类。

在 ETL 系统中，错误事件事实表是捕获、分析并且控制数据质量的核心表。所有来自 ETL 过程的所有错误事件都应该被写到该表中。过滤器维当然就是这一个表的主要驱动器。这一架构是 ETL 系统的主控制面板的基础。

清洗提交报表#2： 审计维

在前面章节描述的错误事件事实表获取数据清洗事件，事件粒度是在 ETL 系统中的任何表中的单条记录。显然，这些事件不可能在最终提交到前端的表的个别记录的粒度上发生。为了把数据质量指示器和最终用户的事实表进行关联,我们需要在这些表的粒度上单独赋值建立一个维，我们把它叫做审计维。审计维描述了被递交到前端的事实表记录的完整数据质量上下文。

审计维按照字面上与数据仓库中的每个事实记录相关联，它将保存 ETL 过程中里程碑的时间戳和输出结果,以及明显的错误和他们在记录上发生的频率，另外还有整个数据质量评分。审计维记录在清洗和规范化事实表记录处理的最后一步创建，并且必须包括应用到该记录的修复描述。

审计维取得每一事实表记录的特定数据质量上下文。 但这通常不会引起审计大小记录的巨量增加，因为审计维的目的是描述遇到的每种数据质量类型。在理想情况下，对于一个将新数据的载入到事实表完整干净的过程，只有一条审计记录产生。如果该过程除了一些因为异常值而触发了越界检查的输入记录外都是干净的，则将生成两条审计记录： 一条是为正常的记录和一条是为越界记录。绝大多数事实记录会使用代理键作为正常的审计记录，极少数异常的事实记录会使用代理键作为越界审计记录。

图 4.5 给出了有代表性的审计维设计。

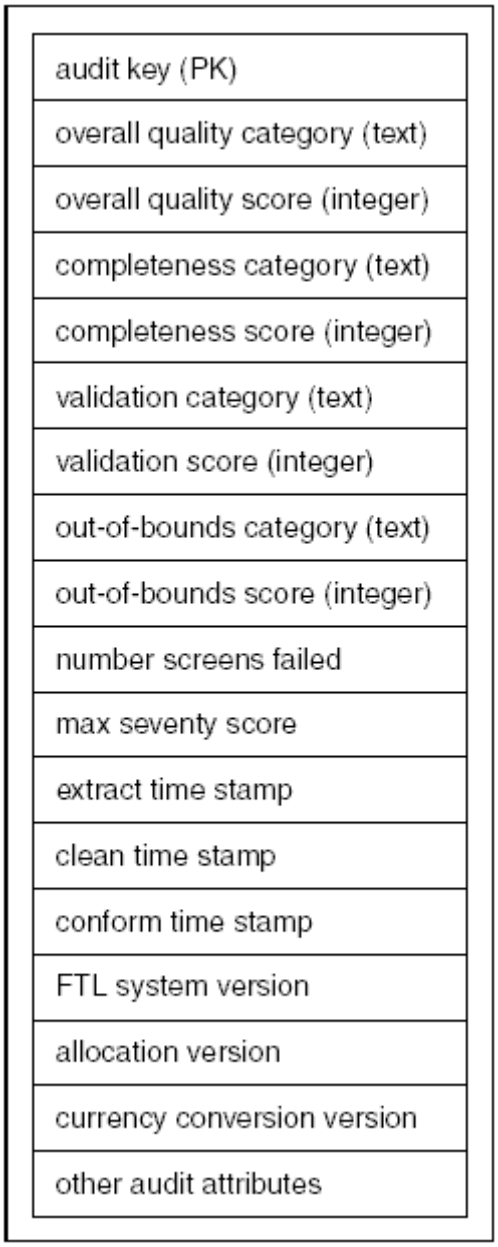


图 4.5　审计维

数据质量属性和其评分是通过为整合的记录以及它的相关数据源系统的记录进行所有错误事件事实的检查来计算。审计维包含一些错误事件事实表的属性，它们是通过累计事实记录的错误评分和与其相关的规范化维实例的评分，以及每条创建了集成维和事实的源记录而得到的。如果你为每个过滤器进行分类，则对于每一个分类的累计数据质量得分都能以本文的和数字的形式在审计维中记录描述属性。文本表格对于错误条件的定性描述的标签报表非常有用。数据质量的完整性，检验和越界，以及图 4.4 中展示的审计维属性等就是这一技术的例子。

同样地，你还可以计算错误事件的总数和针对每一事件的最大严重程度评分，作为审计维中有意义的属性记录。最后，审计维是一个完美的占位符，它记录了所有的时间戳和你已经收集的 ETL 编码的系统属性，用于支持数据仓库经理的分析和解决问题。

也许最大的回报是来自把审计维开放给最终用户群体。最终用户的查询和报表现在既可以在正常模式下执行，也可以在工具模式下运行。只需简单地把审计维的属性之一拖曳进查询和报表中，原始结果就会按照不同的数据质量条件分为多个部分。例如，只需把越界分类的属性拖曳到销售总额报表中，大量的店铺就会被划分成三行(正常的销售店,异常高的店铺，和异常低的店铺)。

注意我们已经偷偷地将一些全局的元数据语境放进审计维！在表的下面部分是一些全局发布版本和版本号。因此，如果你在报表期间改变了你的收入分配结构,你可以拖曳分配逻辑版本号属性到报表中，它将会把每个结果集的行分开进入到用旧结构和新结构进行计算的两部份。我们在此添加了数据状态的元数据。

审计维的最佳点

目前还没一个能被广泛接受的、比较成熟的计算事实记录全部的数据质量得分的方法。定义一种表示实际验证过的数据质量水平的方法是一种挑战,可以以某种形式执行这样方法，使其能随着时间的推移，对一组数据质量过滤器预先进行调整。毕竟，每次当信息质量负责人要调整过滤器的时候，你不想要在数据仓库中为所有的事实再次调整数据质量评分。例如，如果执行很少的过滤器，实际验证过的数据质量水平要比在之后加入到 ETL 流中更多更全面的过滤器要低。

一种计算所有有效事实数据评分的技术是对与事实相关的所有错误事件记录的错误事件严重程度评分进行加和。当然，这里要假设数据源到目标的 ID 映射是作为 ETL 匹配数据集成功能的一个副产品产生的(稍后在这个章节中将会介绍)。观测事件得分的总和是从一个最坏情况错误得分中减去决定在审计维被用的全部有效数据质量的得分后得到的。最坏情况错误得分所有用于抽取的，清洗的和规范化的数据过滤器的最严重错误得分之和。因此，如果十个不同的过滤器在一个事实表记录和九个维表记录上执行——-每个过滤器都可以生成一个最坏情况，即数据质量严重性得分的 10 分——则所有的最坏情况得分总数是 100。

再次声明： 如果每个过滤器在应用的过滤器中都发现缺陷那么总共的数据质量严重程度得分会是 100 。了解了这些，你可能会选择给出完全有缺陷的事实一个零得分，而给没有任何错误的事实得分 100。因此，这一技术提供了针对实际应用的一系列过滤器的数据质量全面的度量方法。如果组织选择在 ETL 过程中增加更多的过滤器，则验证的数据质量得分的就很有可能增加。这似乎很合理,因为组织现在认为它的数据处于一个较高的数据质量水平上。

针对每个事实表能定制唯一的审计维结构。换句话说，你可能选择创建一个审计维设计家族，而不是强制所有审计维包含相同的信息。这将使得对单独事实质量的单独诊断记录到一个单独的审计维记录中，这里的关键是要保存这一个表的维度信息。

该部分已经讨论了审计维的设计，其中描述了属于事实表记录的数据质量诊断和所采取的动作相关的内容。这样，它为所有的事实表建立了一个清晰的维度模型。但是是否可能为维表设置一个审计维呢？我们的回答是“不”；你也不需要这样做。我们更喜欢在维表中直接嵌入数据质量诊断和动作。针对整个数据可靠性的数据质量诊断应该作为附加字段包含在维表中。类型 1 对维信息的修改（覆盖）也可用这种方式描述。类型 2 的修改(在一个特定的时间点对属性进行变更) 则使用面更广，包括时间戳和原因代码,这能完成个别的审计维的很多目的。如果需要针对数据仓库结构的所有变化的完整审计轨迹来生成报告,则你需要设计特定的结构来分别记录这些变化。

5.5 第 3 部份：过滤器及其度量

我们现在准备好做一些详细的设计。该部分是在大多数数据清洗引擎的核心描述了一组基本的检验和测试。它描述了这些功能的作用,如何做，以及它们如何互相作用以提交清洗好的数据到维度数据仓库中。我们非常感谢 Jack Olsen 创造了在后面部分介绍的架构和词汇表，这些描述在他的著作中：<<数据质量，精确维度>>。

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

异常检测阶段

数据异常指的是那些不适合和其它数据放到一起的数据。还记得当你还是小孩子的时候，有人会给你一张图片，然后问你“这图片有什么问题吗？”。你会指出车轮是方的或者停止标志挂颠倒了之类。数据异常就是数据库中是方形车轮。发现这些异常需要有特定的技术而且需要仔细地分析研讨。在该部分我们将介绍已经在我们的数据仓库项目中被证明是成功的异常检测技术。

期待的时候该期待什么

在 ETL 过程创建时暴露数据异常是导致 ETL 部署延时的主要原因。发现数据异常会花很多的时间和分析。提前做这样的分析,可以节省时间而且减少犯错。较好的方式是花时间一次又一次重建同一 ETL 作业，此时可以尝试去纠正由于没有被发现的数据异常导致的错误映射。

可以通过一些发生在数据仓库以外的数据质量问题来发现数据异常。但是发现数据异常通常是 ETL 团队的职责,除非你的项目已经制定了成熟的数据质量分析子系统的预算。

数据采样

最简单的检测异常的方法是对有问题的列进行分组时计算该表的行数。这个简单的查询，结果展示在图 4.6 中，揭示了值的分布和可能损坏的数据。

```
Select state, count(*)
From order_detail
Group by state
```

如图 4.6 所示,数据异常立即暴露了出来。在结果集中的异常就是数据异常，强烈建议通知给业务负责人，以便在源系统中进行整理。

State	Count(*)
Rhode Island	1
Missisippi	2
New Yourk	5
Connecticut	7
New Mexico	43,844
Vermont	64,547
Mississippi	78,198
Utah	128,956
Wyoming	137,630
Missouri	148,953
Rhode Island	182,067
Minnesota	195,197
North Dakota	203,286
Michigan	241,245
Washington	274,528
Pennsylvania	287,289
Montana	337,128
Louisiana	341,972
Virginia	346,691
West Virginia	359,848
Delaware	422,091
Iowa	456,302
Massachusetts	480,958
Tennessee	483,688
New York	494,332
Nevada	506,235
South Dakota	514,485
Indiana	553,267
Connecticut	562,436

图 4.6　值分布查询结果

技术点：数据评估工具正好就是用于执行这种类型数据采样的。

分析源数据听起来很容易，是吧？但当你的源表有数亿行，并有 250,000 个不同的取值时会怎么样呢？分析海量数据源的最好方法是的数据采样。我们已经为采样数据使用过许多不同的技术,从简单地选择开始的 1,000 行到使用最精细的算法，没有哪一个是效果很明显的。我们发现在如下的查询中，只是简单计算表的行数，而且均匀地把表分成几个特定的部分，就可以准确地采样，而不用考虑表中数据的值：

```
select a.*

from employee a,

(select rownum counter, a.*
```

```
from employee a) B
where a.emp_id = b.emp_id and
mod(b.counter, trunc((select count(*)
from employee)/1000,0)) = 0
```

为了检验更多或更少的数据，只需改变想要采样返回的行的数目 1,000 即可。

另一种方式是把一个列的随机数目加入到数据中,这可以用来排序选择整个表中任意需要的部分。

一旦你得到这些采样数据，你可以进行常用的值分布分析。通过任意其它方式选择数据，包括选择全部数据，可能会歪曲你的测试结果。

我们已经注意到一个常用的错误，那就是选择一个日期区间来缩小结果集。数据异常通常发生于应用程序的缺陷中或者由于未经训练的操作员的误操作。我们已经碰到的大多数的异常都是临时发生的；则纠正应用程序或者更换操作人员，都会造成异常消失。因此在一定日期区间中选择数据可以很容易的错过这些异常。

约束的类型

把各种不同类型的数据质量检查分为四个大的类型是有用的：

- 列属性约束
- 结构约束
- 数据约束
- 值约束

列属性约束

列属性约束保证由源系统输入的数据包含系统期望的数值。较有用的列属性约束检查包括如下过滤器：

- 检查列的空（Null）值
- 超出期望的最高和最低范围的数字值

- 长度超长和超短的列
- 包含有效值列表之外的数值
- 匹配所需的格式或一组格式
- 在已知的错误值列表中命中数，之所以用这个列表是由于可接受值的列表太长了
- 拼写检查器

一些特定的过滤器技术将在这一个章节的后面部分介绍，这些技术用于执行精确地有效性的检查和异常捕获。基于这些过滤器的检查结果，ETL 作业流可以选择：

1. 让没有错误的记录通过
2. 让记录通过，同时标记有错误的列值
3. 拒绝记录
4. 停止 ETL 作业流

一般的情况会选择第 2 项,即让记录通过 ETL 过程流并记录任何遇到的已验证的错误到错误事件事实表中，使得这些错误对最终用户可见，从而避免由于虚假的完整性而造成数据仓库的可信度的损害。有很严重缺陷的数据记录是不可能出现在数据仓库中，或者会严重损害数据仓库可信度，所以可以被完全的忽略，当然也要把错误事件适时的记录到错误事件事实表中。最后,影响整个 ETL 批处理过程的数据完整性的数据验证错误应该完全停止批处理过程，这样数据仓库经理能更进一步地进行调查。过滤器维包含一个异常动作列，它与每种过滤器的这三种可能动作相关联。

结构约束

列属性约束重点是在单独每个列上,但是结构约束却是把重心集中在列之间的关系上。我们通过确认每个表有合适的主键和外键，并且遵循参照完整性来约束结构。我们检查每组字段里显性的的和隐性的层次和关联关系，比如构建一个合法的邮件地址。结构约束还检查层次父子关系，来确定每个子节点有一个父节

点或者在一个家族中是根节点。

数据和值规则约束

对于数据和值规则域的范围，从简单的业务规则，比如如果客户处于优先状态,则透支上限至少是$1000，到更复杂的逻辑检验,比如一个商业客户不能够同时是一个有限责任合作伙伴和类型 C 公司。值规则是这些数据上合理性检验的扩展，并且可以用聚合值业务规则的形式，例如一个临床医师正在报告需要 MRI（核磁共振成像）的挫伤手肘者的不可能的数目。值规则也能提供一个数据可能不正确的概率警告。确有叫做 Sue 的男孩，至少在 Johney Cash 的歌里，但是也许这个记录应该被标记以等待检查。如果这个记录是不正确的,你不会知道是名字还是性别应该被改正。

这些发现都很难包含在错误事件事件表中，因为这些错误包括许多记录。要确定哪个记录不正确是几乎不可能的。这样就只剩下两个选择：把所有这些记录（例如需要 MRI 的挫伤手肘者）标记为有问题，或者创建一个虚拟的聚合表，在这个表里错误可以用发生的次数来表示。

驱动过滤器设计的度量

这一个部分讨论进入数据仓库的数据清洗的基础都需要做些什么，包括简单的探测、捕获、定位普通的数据质量问题的方法，以及提供给企业整个数据流的可视化能力和随着时间的进展数据质量逐步得到改善的各个方面。

整个处理流

一系列的数据质量过滤器或错误检验程序按照元数据中定义的规则排队等待执行。每个过滤器都描述在创建的过滤器维中，这同时也作为本章前面部分错误事件结构的一部分。当过滤器执行的时候,每个错误的发生都记录为一个错误事件记录。 每一错误检查的元数据也会记录了错误事件的严重程度。最严重的数据质量错误被归类为致命错误，它将会引起整个 ETL 过程中止。 举一个发生致命错误事件的例子，比如来自多个商店的日销售数据完全丢失，或者在一个重要列中出现了一个不可能的非法数值，而针对这个值没有相应的转换规则。

当每一个数据质量检验已经执行后，就会在错误事件事实表中查询在整个数

据质量处理过程中遇到的致命事件。 如果没有发现，继续正常的 ETL 过程；否则向整个 ETL 处理返回暂停指令,然后将按次序执行 ETL 停机过程，并且通知数据仓库系统管理员或信息质量驾驶员。这个过程描述在图 4.7 中。

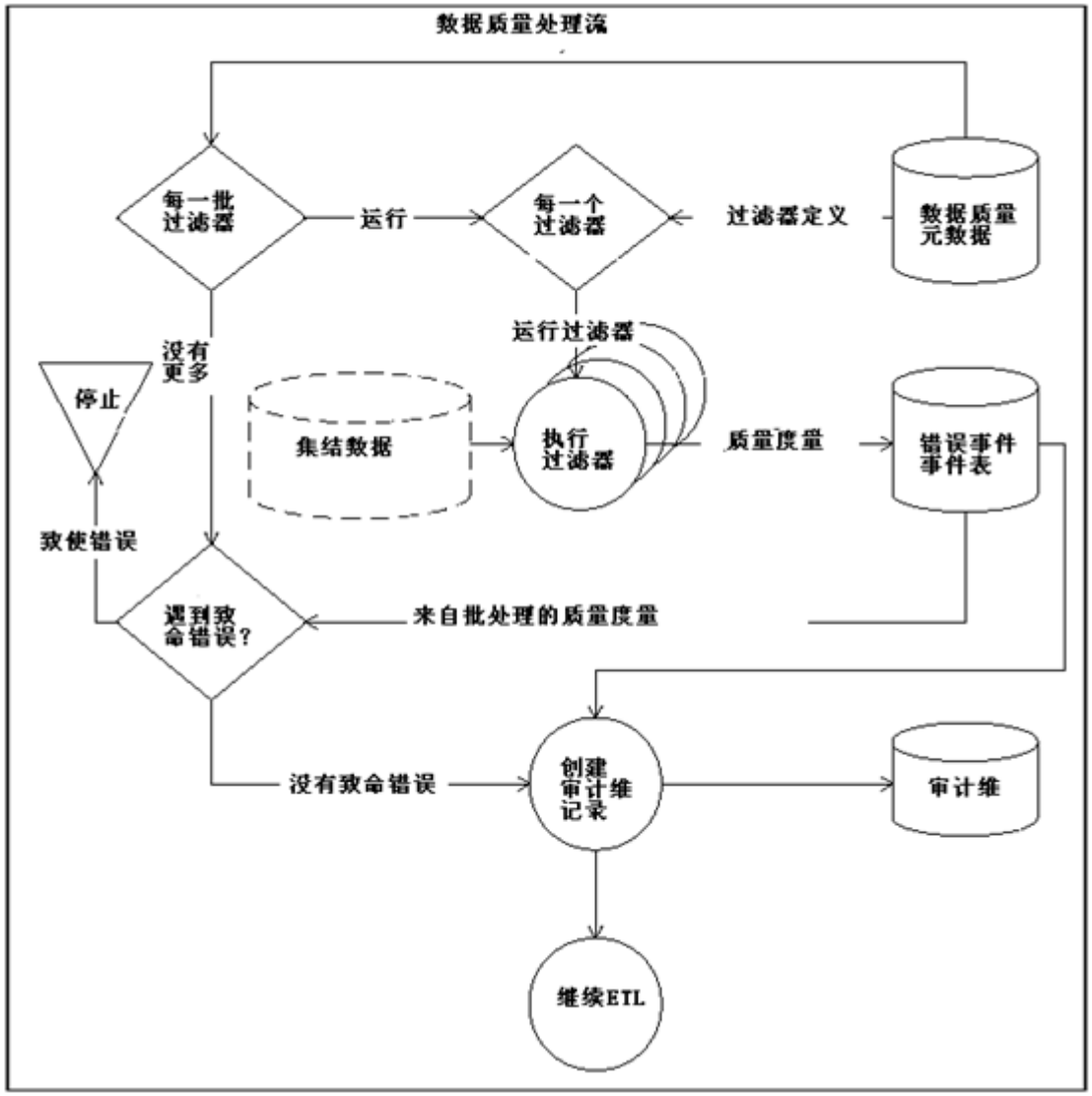


图 4.7　整个处理流

为了更好的性能，数据清洗子系统的处理流程的目标是触发可以并行运行的批过滤器。这些过滤器识别出数据质量问题，并在错误事件事实表中插入记录。为了最大程度减少数据库的冲突问题，应该在错误事件事实表上避免不需要的索引或者约束，以便当从多个过滤器处理同时写记录时不会引起问题。调用过程在触发下一个批过滤器前要等待每一个批过滤器执行完成，直到没有剩下的批过滤器要执行。正如在本章前面所述的，过滤器元数据表中的处理顺序号是用于调度过滤器的。具有相同的处理顺序号的过滤器能并行执行。标准的数据仓库作业调

度工具也可以用来做更复杂的过滤器调度并管理他们之间的依赖关系。

当清洗子系统完成清洗处理并规范化记录后，还需要做一些额外的工作，为审计维给出整个数据质量得分。这通过把在流程中清洗和规范化过的记录的错误事件事实和那些相关的源记录（如果这种关系是可用的)进行聚合来实现，并作为 ETL 集成/匹配过程的附属品来保存。有意思是，过滤器也能被应用到错误事件事实表本身,可以创建一些特定的过滤器，来度量在整个数据清洗作业流中任何阶段积累起来的数据质量错误的数目和类型。这项技术将在下一部分中进行更进一步的描述。

建议的运行过滤器的方法是建立一个通用的软件模型，它可以执行任何过滤器，只需要设置批处理 ID 和过滤器代理键作为参数即可。这一模型为过滤器抽取元数据并构造一个动态的 INSERT 语句，它将为每一个过滤器返回的有问题记录组装错误事件事实表。动态 INSERT 语句的通用形式如下：

```
INSERT INTO data_quality_error_event_fact
(etl_batch_surrogate_key, day and time of day surrogate keys,
list of values from the Screen Meta Data record,
offending_record_surrogate_key)
SELECT offending_record_surrogate_keys provided by the Screen's SQL
Statement
```

持续运行

数据清洗子系统的指导原则是发现并且记录存在的数据质量错误,而不是跳过记录或者是停止 ETL 过程。数据质量问题在数据仓库生命周期中是一个不幸并且存在的事实，业务经理不得不面对不完善和不准确的数据每天作出艰难的决定。这种情形不是一个晚上就能解决的。相反，应该为整个组织提供一些工具来精确度量正在使用的数据的质量，并在整个提高数据质量过程中了解其过程。

也就是说，数据清洗子系统也必须提供一些处理意想不到情况的机制，这些情况既包括那些由于太严重的缺陷而不被允许进入数据仓库的记录，也包括那些很严重的系统级的缺陷，能导致整个 ETL 过程停止的记录。由于实际的和策略的

原因，触发这些异常的应对措施的阈值必须进行平衡，既要使数据仓库保持对业务来说是一个有用的可行的工具，又要提供足够的数据质量保证，从而对于最终用户来说保持系统可信性。这是一个困难的平衡，而且要随着时间变化不断的调整。因此 ETL 数据质量子系统应该支持调节这些阈值的能力，并当遇到数据质量错误时修改相应的动作。

在一些情况下，如果检测到过多低级错误，就要采取异常动作。举例来说，一个客户地址记录中无效的美国州的邮编通常会有一个错误事件写到数据质量主题域，但不会终止整个的 ETL 过程。但如果一批的记录中都有非法的美国州邮编,则可能表示在上一级的处理过程过程中有严重问题，严重到影响 ETL 处理流中所有数据的完整性。针对这种情况，建议的处理方式是创建额外的数据质量过滤器，让它直接地针对错误事件事实表运行，计算在整个数据质量批处理中捕获的数据质量错误事件记录的数目，并且触发异常处理程序。

注意这些特别的过滤器，它们是把发现的错误写回到错误事件事实表中。它们读写同一张表中，这也容易造成数据库冲突。与其它大多数数据质量过滤器处理方式不一样，它并不是针对每个有问题的记录都写到事实表中，而是统计来自特定表的某种特定类型的错误数，并只有当统计值超过了允许的上限时才写一条记录到错误事件事实表。这样将可以避免大多数冲突问题。

过滤器

在运行过滤器之前,应该要先建立一个全局的数据评估基本规则，其中包括定义列的一些属性，如无效值，数字列的范围,字符列长度限制以及表大小。对于一个又一个数据源,没有比深入研究数据本身来确定高质量样本数据的特征更合适的方法了。这一研究过程应该包括数据提供者的技术文档研究和源数据本身逐列查看。对于每个被载入数据仓库的数据源，一个数据评估的检查列表应该包括：

- 为抽取的表提供每天记录数的历史
- 提供每天的关键业务度量的总数历史
- 确定需要的列
- 确定需要唯一的列集合

- 确定允许(以及不允许)为空的列
- 确定数字字段的可接受数值范围
- 确定字符列的可接受的字符长度
- 为所有列定义明确的有效数值集
- 确定列中无效值(不在有效值集合中)出现的频繁程度

如果没有专用的数据评估工具，数据评估的好处通过手写 SQL 也可以实现部分，只是这需要一支专家队伍，以及大量的时间和辛苦的努力。这和在本书一开始就讨论的手工方式和工具方式的区别以及为何要选择完整的 ETL 工具相对应。换句话说，厂商提供的工具正在逐渐地抬高这一门槛，这使得自己开发方式越来越不能满足需求，除非的需求相对非常简单。数据评估实验的结果应该由信息质量负责人来维护，因为他接下来可以直接地把它们运用到数据质量过滤器的元数据定义中，而这可以驱动 ETL 的数据质量进程。

已知表行数

在某些情况下，信息质量负责人通过业务规则完全可以知道来自一个给定数据提供者某种数据类型的记录数。举个例子来说，比如一个仓库每周的库存清单，其中每个部分都会被记录，即使是零。在其他的情况下，信息质量负责人可以基于历史信息指定来自某个数据提供者的可接受的记录数范围，并创建过滤器来检测过高或过低的记录数。已知表的记录数工作能用简单的过滤器 SQL 来处理处理，如下所示：

```
SELECT COUNT(*)

FROM work_in_queue_table

WHERE source_system_name = 'Source System Name''

HAVING COUNT(*) <> 'Known_Correct_Count'
```

因为这是一个表级的过滤器，错误事件事实表中清洗过或者规范化过的识别符应该置为空。

列空值

决定数据记录中那些列是非空的（与允许为空相比）是非常重要的，同时通常随着数据源系统改变而变化。举例来说,一个销售网点的应用系统可能允许客户地址属性为空，但是一个物流系统客户地址属性就必须是非空的了。元数据结构的目的是基于每个数据源捕获空值规则。在维度模型中，完整的记录往往比源数据有更严格的空值规则。因为几乎所有维的属性列都要求非空，即使仅是“未知”、“不可用”或“不能用”之类的字符串也可以。

系统自动给空的文本字段加上一个实际的值可以消除二义性，不管这个字段值是丢失了还是真的为空。这一技术也简化很多 SQL 查找表；不幸地是，关系数据库处理空白文本字段的方法和空值文本字段的方法是不同的。 即使没有给空文本字段赋值，我们也建议至少要转换所有的空文本字段为空白文本字段。

检测空值的建议方法是建立一个指定源的空值 SQL 语句库,它返回有问题的记录的唯一标识符，如下所示：

```
SELECT unique_identifier_of_offending_records

FROM work_in_queue_table

WHERE source_system_name = 'Source System Name'

AND column IS NULL
```

对于来自完整记录的过滤器错误，应该稍微调整 SQL，在其中使用特定的源系统名称，如下：

```
SELECT unique_identifier_of_offending_records

FROM work_in_queue_table

WHERE source_system_name = 'Integrated'

AND column IS NULL
```

过滤器返回的每个有问题的记录都会被插入到错误事件事实表中，而且那些有问题的行的唯一标识符也被作为退化维写入到事实表中。

列的数字和日期范围

虽然许多数值和日期在关系数据库表中都可以接受很大范围的值，不过从数

据质量的角度来看,它们应该加以更严格的有效值范围限制。如果单一某个客户交易了一百万个单位，这可信吗? 也许是的，如果我们的生意是一个全球的 B2B 交易，但是如果这是消费者零售点交易则就不可能了。你一定想要你的数据清洗子系统能够发现并且记录包含在信息质量负责人定义的合法数值范围之外的实例。在某些情况下，尤其是那些第三的 ETL 计算使用到的数字列，这些数值范围将会由信息驾驶员定义。在这里再一次提醒，整合数据的列可能有一些和其他任何数据源都不相同的合法的数值范围，因此，你需要用不同的过滤器来验证它们。一个过滤这些潜在的错误的 SQL SELECT 语句的例子如下所示：

```
SELECT unique_identifier_of_offending_records

FROM work_in_queue_table

WHERE source_system_name = 'Source System Name'

AND numeric_column IS NOT BETWEEN min AND max
```

假如我们有一个事实表，它跟踪 600 个商店的每日销售额，其中每个商店有 30 个部门。 这样每天收到 18,000 个销售额记录。使用一个以标准偏差为基础的快速统计检测,可以让我们判断这 18,000 个销售额数目的合理性。 这一技术也让我们很快地更新统计基值，从而为明天数据的载入做好准备。

记住：标准偏差是差值的平方根。差值是每一个历史数据点和数据点的平均数之差的平方的总和，再除以 N-1,这里 N 是数据的天数。不幸的是，这一个公式需要我们遍历销售历史的全部时间，虽然可能，但是在快速运行的 ETL 环境中，这样的计算毫无吸引力。但是如果我们已经掌握 SUM SALES 和 SUM SQUARE SALES，我们能把方差的计算公式写出这样：(1/(N-1))*(SUM SQUARE SALES - (1/N)*SUM SALES*SUM SALES)，检查公式吧!

因此如果用 VAR 缩写我们的偏差公式，则数据有效性检查就如下所示：

```
SELECT s.storename, p.departmentname, sum(f.sales)

FROM fact f, store s, product p, time t, accumulatingdept a

WHERE

(first, joins between tables... )
```

```
f.storekey = s.storekey and f.productkey = p.productkey and
f.timekey = t.timekey and s.storename = a.storename and
p.departmentname = a.departmentname and
(then, constrain the time to today to get the newly loaded data... )
t.full_date = #October 13, 2004# and
(finally, invoke the standard deviation constraint... )
HAVING ABS(sum(f.sales) - (1/a.N)*a.SUM_SALES) > 3*SQRT(a.VAR)
```

我们扩展先前介绍中的 **VAR** 并使用 **a.**做 **N**，**SUM SALES** 和 **SUM SQUARE SALES** 的前缀。并且已经假定部门按产品分组，因此可以在产品维中使用聚合值。

在这一个模型之上包括两个以下两个查询的运行：高于平均值三个标准偏差以上的销售额和低于平均值三个标准偏差以下的销售额。也许对于这二种情形有不同的解释。如果在 **SQL** 中不喜欢写这样的 **HAVING** 子句，还可以不使用 **ABS** 函数。如果通常情况下每天得到波动的销售额（比如，周一和周二比起周六来低很多），则可以加入一个 **DAY OF WEEK** 字段到聚合的部门表中，并且约束到合适的日范围。这样就不会把正常的每天波动状况混杂在标准偏差测试中了。

当用前面的 **SELECT** 语句完成输入数据检查的时候，就可以更新 **SUM SALES** 和 **SUM SQUARE SALES**，只要把今天的销售额和今天的销售额平方分别加到聚合的部门表中即可。

列长度限制

文本列的字符串长度的过滤对于 Staged 的和完整的记录错误都是很有帮助的。例如，可以检查客户的名字，过滤掉确定太长或者太短的值。如下的 SQL SELECT 语句就是执行这样过滤的例子：

```
SELECT unique_identifier_of_offending_records
FROM work_in_queue_table
WHERE source_system_name = 'Source System Name'
AND LENGTH(numeric_column) IS NOT BETWEEN min AND max.
```

列的明显的有效值

对于一个给定的由它的源系统定义的离散的有效值集合的列，可以在处理的列中查找未知的值来过滤异常。同样，也可以针对来自任何数据提供者的列使用有效值的通用列有效性参考表来，从而把它作为 staging 过滤器来处理。因此,代表性的 SQL 语句如下所示：

```
SELECT unique_identifier_of_offending_records
FROM work_in_queue_table Q
WHERE source_system_name = 'Source System Name'
AND column NOT EXISTS
( SELECT anything
FROM column_validity_reference_table
WHERE column_name = "column_name"
AND source_system_name = 'Source System Name'
AND valid_column_value = Q.column_value
)
```

列的明显的无效值

对于一个给定的列，经常含有一些已知不正确的值，并且这里并没有离散的有效值集合，那么信息质量负责人可能选择使用这些无效值的过滤器。例如在一个客户名称字段中，“未知”这样的字符串会经常出现，但这里并没有定义全部的有效客户名称集合。则这些明显的无效值过滤器显然并不是试图完全过滤出所有的无效数值，而只是选出那些经常出现的错误。像名字和地址的标准化及匹配这样的其他数据清洗技术，应该更适合这些工作。下而是一个含有问题字符串的过滤器 SQL 语句。

```
SELECT unique_identifier_of_offending_records
FROM work_in_queue_table
```

```
WHERE source_system_name = 'Source System Name'
AND UPPER(column) IN ("UNKNOWN", "?", list of other
frequent offenders... )
```

更好的方法是把数值与记录了频繁出现的无效值的表进行比较，如下所示：

```
SELECT unique_identifier_of_offending_records
FROM work_in_queue_table Q
WHERE source_system_name = 'Source System Name'
AND EXISTS
( SELECT 'Got One' FROM Table_Of_Frequent_Offenders WHERE column_name =
Q.column_name)
```

如果一个列的有效数值的集合太大而无法明确定义或者是未知的，那么这类过滤器只拥有有限的值集合，但是在某些有用的情况下可以使用最近发现的错误规则；并不断重复这些错误规则。

检查表行数的合理性

这一类的过滤器功能强大但是实现也更复杂。它试图确定来自某个数据源的行的数目是合理的，也就是说行数落在基于先前历史有效的记录数的有限范围内。为了测试表行数的合理性，可以选择使用一些简单的统计测试方法，比如根据前一个类似值的平均值计算标准偏差值，或者是选择更高级更专业的预测值，例如 X.12 标准，或者甚至像 ARIMA(自回归的整合移动平均)这样的技术。如果你对这样功能强大的统计工具感兴趣，则你将需要一个优秀统计师几个星期的咨询。一个正好完成这样的统计的地方是在市场研究部门，如果有如此一个部门的话。

如图 4.8 所示的数据 staging 表记录数表每天从每个数据源的每张表中获取记录数，每天每个数据源对应一行。

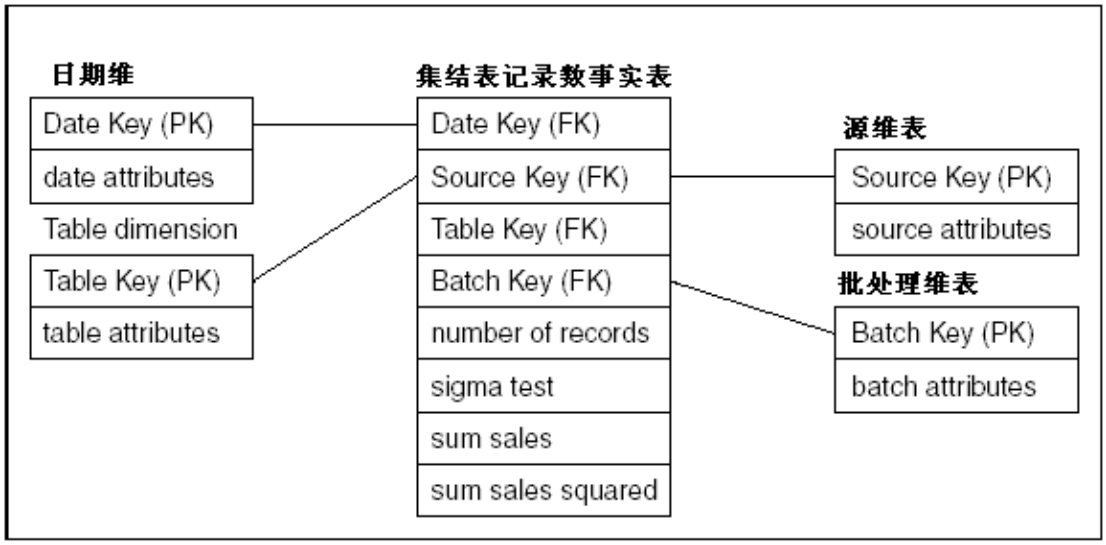


图 4.8　表级实用元数据

图 4.8 用维度结构展示了这些表。一些 ETL 工具在正常操作过程中作为副产品生成这样的表。使用数据 staging 表记录数表，这一个过滤器可能需要两路 SQL 来处理，如下所示：

```
SELECT AVERAGE(Number_of_Records).3 * STDDEV(Number_of_Records),
AVERAGE(Number_of_Records) + 3 * STDDEV(Number_of_Records)
INTO Min_Reasonable_Records,
Max_Reasonable_Records
FROM data_staging_table_record_count
WHERE source_system_name = 'Source System Name"
;
SELECT COUNT(*)
FROM work_in_queue_table
WHERE source_system_name = 'Source System Name"
HAVING COUNT(*) NOT BETWEEN
Min_Reasonable_Records AND Max_Reasonable_Records
```

;

聪明的 SQL 高手可以实现前面提到的过滤器，既可以使用多路 SQL(如前面 SQL 所示)，其中每个 SQL 都针对一个数据源，或者使用单个过滤器根据特定的数据质量需求和需要的严重程度得分灵活性来验证所有数据源的表记录数合理性。信息质量负责人也可以根据应用的不同的标准偏差容许度和不同的严重程度得分来为相同的表和源系统定义多个过滤器，举例来说，记录偏移平均值两个标准偏差的严重性错误为低，评定偏移平均值三个标准偏差为高严重性错误，并且如果偏离四个标准偏差，需要彻底地停止整个 ETL 过程。

表行数过滤器能很容易被扩展到数据仓库的任何额外的度量上，以支持合理性检测。 举例来说，添加一个总销售额度量到图 4.8 的表，当销售额严重偏离时，过滤器就能记录下这一情况：

```
SELECT AVERAGE(Total_Sales_Dollars).3
* STDDEV(Total_Sales_Dollars),
AVERAGE(Total_Sales_Dollars) + 3
* STDDEV(Total_Sales_Dollars)
INTO Min_Reasonable_Sales_Dollars,
Max_Reasonable_Sales_Dollars
FROM staging_table_record_count
WHERE source_system_name = 'Source System Name"
;
SELECT SUM(Total_Sales_Dollars)
FROM work_in_queue_table
WHERE source_system_name = 'Source System Name"
HAVING SUM(Total_Sales_Dollars) NOT BETWEEN
Min_Reasonable_Sales Dollars AND
```

Max_Reasonable_Sales_Dollars

检查列分布合理性

具有发现数据维属性偏离了常态分布能力的过滤器是另外一种功能强大的过滤器。当一个拥有一组离散的有效值集合的列由异常偏离的数据分布组成时，这个过滤器就能够发现并且捕获这一情况。举例来说，被过滤的列可能是在来自自动售货系统(SFA)的一个销售电话事实表的产品列表。假设历史经验证明大多数的销售电话都集中在产品 A(这一商品有较高的退货率) 而只有少许电话集中在产品 B(退货率低的产品)。你需要设计一个过滤器来提醒信息质量负责人，以针对那些突然产品 A 的销售电话减少或者产品 B 的销售电话增多时的情况。

可以使用和前面所述的表行数合理性技术来创建这一过滤器。另外，还需要一个 staging 表来保存历史记录数和随时间变化的列的有效记录数，由此还可以为过滤器计算平均值和标准偏差。因为对一个给定的列和许多有离散分布有效值集合的列，可能会出现很多值，因此需要偏离你的元数据标准和为表及列集合设定的计划 staging 表，并由过滤器进行审查。当然，这样就增加了数据 staging 表的数目，但是它使得 ETL 架构在物理实现这些有潜在的大表的时候显得更加灵活。某些情况下，这种非通用的数据 staging 方法创建的表会过大而不适用于高效的 ETL 过程，因此就可以使用前面提到的判断数据的平均值和标准偏差的统计技术。

注意前面提到的统计方法同样可以支持多列过滤，即测试多个列组合的有效值的合理性。在本章前面前面，我们把它叫做值规则约束。例如，按产品和商店来审查每天的销售额，或按产品、商店和星期审查，找到偏离历史标准的不合理结果。

修改图 4.8 中的表，加入产品作为一个维，使得我们可以得到按产品的每天销售电话次数。使用这个表，过滤器能按历史的产品编码和源销售电话总数的平均值和当前的 ETL 批处理的值相比较。 对于那些平均超过已经建立的标准偏差(正如在过滤器定义的 SQL 语句中中的定义) 的阈值的那些产品应该记录错误事件到事实表中。

处理这类使用提及过的技术的过滤器，需要由主流的过程性 SQL 语言扩展很

好的支持的级别上进行过程性编程。这一过程性 SQL 可以能被包含在过滤器 SQL 语句定义中或在它之外处理。重要的是，ETL 构架在为所有过滤器维护过滤器元数据实例以及为所有过滤器发现的所有错误事件生成错误事件事实表方面需要具有连续性。

无论选择用什么方法实现，由过滤器创建的错误事件事实都被认为是表级的过滤器，因此错误事件事实表的清洗/规范化记录标识符应该为 NULL。

通用的数据和值规则合理性

数据和值规则正如本章前面所定义那样，是与主题密切相关的，因此我们不能够给出一个实现的特定检测清单。但是清晰的合理性查询与本节中例子中给出的简单数据列和结构检验很相似。

5.6 第 4 部分：规范化报表

数据集成意味着创建规范化的维，以及通过组合来自多个数据源的最有效信息为一个综合的视图来创建的事实实例。为了实现这些，导入的数据需要结构统一，并过滤掉无效纪录，将内容术语标准化，删除重复记录，从而产生新的规范化后的映像。在这部分，我们将按三个构建规范化维和事实的步骤来描述：

- 标准化
- 匹配和删除重复记录
- 生存

当我们规范化数据时，我们可以把来自三个数据提供者的性别代码(M，F)，(M,W),或 (Man，Woman)转换为标准的性别维属性(Male，Female)。同样我们也可以用专门的工具来规范化姓名、地址等信息。

跨越多个数据源、多个数据集市，以及访问分布式的数据仓库的多个远程客户端来规范化描述属性，对于数据仓库架构师和 ETL 团队来讲是一个关键的开发步骤。在其它一些工具箱书籍中，也有许多关于这一主题的技术性和管理性的描述。ETL 团队所关心的是捕获重复的和不一致输入信息，并支持维表管理员和事实表提供者的需要。

规范化维

不管硬件构架如何，数据仓库从某种程度来说都是分布的，因为每种类型的度量总是必须存在于各自的事实表中。在 ER 模型中相同的语句是真实的。因此，对于从多个分离的事实表组合数据的最终用户应用来说，我们必须为这些事实表提供统一的界面，这样数据才可以被整合。我们称这类一致的界面叫做规范化维和规范化事实。

规范化维对于每一个可以被关联的事实表来说都是相同的。通常，规范化维对每一事实表都是相同的。规范化维的一个更准确的定义是：

如果可以共享一个或更多的取自同一个域的值的属性，那么这两个维是规范化的。当使用规范化维在不同的事实表间钻取时，应用请求必须使用这一相同的属性来作为约束和分组的基础。

图 4.9 举例说明了支持规范化产品维的三个事实表间的钻取操作。

来自规范化维的属性 ↓

Product	事实表 1 Manufacturing Shipments	事实表 2 Warehouse Inventory	事实表 3 Retail Sales	计算列 Turns
Framis	2940	1887	761	21
Toggle	13338	9376	2448	14
Widget	7566	5748	2559	23

过程：1）打开到每一源的单独连接
2）分析所有3个答案集
3）合并答案集到规范化的行标题（product）

图 4.9　在三个事实表间钻取

经常需要规范化的维的例子包括：客户维、产品维、地理维、促销维和日期维（时间）。中心数据仓库设计团队的主要职责是创建、发布、维护和约束规范化维。

创建规范化维对于一个组织来说是非常重要的一个步骤，我们在《数据仓库生命周期工具箱》一书里描述了组织决策以及实现规范化维定义的整个过程。一个规范化的客户维是一个客户的主表，含有干净的代理客户键和许多维护的很好

的描述每一客户的属性信息。规范化的客户维是一个从多个旧系统以及外部数据源汲取融合了许多数据的结果。例如，客户维的地址字段应该包含企业中任何位置的每个客户的可邮寄的和已知的地址信息。这通常是中心数据仓库团队的职责，创建规范化的客户维并作为企业其它部分的的数据源，既提供给数据仓库也提供给旧系统使用。

规范化的产品维是企业达成一致的主要产品列表，包括所有的产品的分类和产品属性。一个好的产品维，就像一个好的客户维，应该拥有至少 50 个单独的文本属性。

规范化的日期维几乎总是一张粒度到日期的表，一般有十年或更多。每一个日期都对应许多有用的属性，这些属性来自与企业合作的许多国家和地区的法定日历，另外还有只为内部管理所用的特定的财务日历周期和市场季节日历。

规范化维对于数据仓库来讲非常重要。如果不严格的坚持规范化维，数据仓库将就不能作为一个整体发挥作用。如果一个像客户或产品这样的维是不规范化的，则多个事实表将不能被简单地在一起使用，或者更糟的是，尝试使之结合起来的将会导致错误的结果。更肯定地说，规范化维使得在同一数据仓库空间里的多个事实表上使用单一的维表成为可能，并且面向统一的用户界面和统一的数据内容，以及统一的属性描述，这样就可以在不同的事实表间进行聚合。

设计规范化维

确定并设计规范化维需要花数周的时间。大多数规范化维很自然的被定义成最小的粒度（原子级）级别。客户维和产品维的粒度自然的将在最小粒度，以便与原系统的条目相对应。日期维的粒度通常为天。

应用

一旦中心数据仓库团队成功的为企业定义并提供了主要的规范化维，则对于那些事实表的拥有者来使用这些维就显得非常重要。对使用规范化维达成一致意见并不是一个技术决定，而是一个业务策略，并且企业数据仓库发挥作用的关键。规范化维的使用应该获得最高级别的领导的支持，这对企业的 CIO 来说可是件头痛的事。

规范化维的可变更性

对于那些已知的特定事实表的数据范围仅仅只是子集的性况，有可能为这些事实表创建规范化维表的子集。举个例子：主产品维表可以被限制为只有部分产品，这些产品只在某个特殊地区生产，并且数据集市也只适用于这一地区。由于缩小了的维表保留原始维表的所有属性并且和初始粒度一样，我们把它称为简单的数据子集。

聚合的数据子集系统地从初始的维表删除行和列。例如，把日期维表从天聚合到月非常普遍。在这个例子中，我们可能仅需要保存每个月的第一天的记录，同时我们也必须删除所有那些仅仅在日粒度出现的像“星期几”和“节假日标记”这样的属性。如图 4.10 所示。

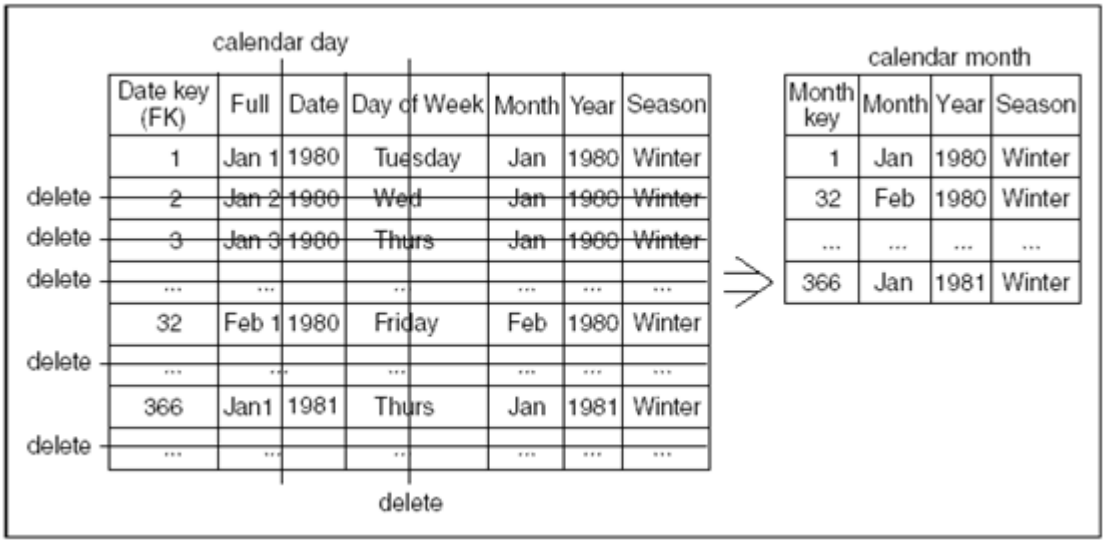


图 4.10　创建规范化的日历月份表

也许你会对如何在一个被子集化了的规范化维环境中创建查询感到困惑？哪一张维表可以用？以及用在哪里？事实上，它比听起来要简单的多。每一个维表和其关联的事实表相对应，任何在事实表间钻取的应用都不可避免的要用多路 SQL 来分别并按顺序查询每一个数据集市。通常情况是每一个钻取报表的每个列都生成一个独立的 SQL 查询。如果不使用规范化维，则只有当报表中使用的维属性在每个维表中找到时才能完成报表执行。由于维被规范化了，业务问题答案才得以保证一致。如果我们创建了规范化的事实定义，这个数量就可以比较了。

规范化事实

我们已经花了很长的篇幅来讲述关于如何创建规范化维来把数据集市绑定在一起这一主要任务。但这也只完成了上层构架的 80%，剩下的 20%是就建立标准的事实定义。

幸运地是，在确定规范化维的同时也确定了标准的事实定义。当我们用相同的术语贯穿所有事实表以及创建单个报表在前面部分描述过的事实表间钻取的时候，我们都需要标准的事实定义。

建立规范化维是一个协作的过程，因为每一张事实表的管理员都必须就使用规范化维达成一致。在规范化会议中，管理员也需要确定在每一张事实表里呈现的类似的事实。例如，许多事实表可以报告收入信息。如果最终用户应用期望添加或比较这些来自多个事实表收入度量，则定义这些收入度量的业务规则必须是相同的。也许有的表是在月末记录收入度量，但是另外的表则可能从帐单期的聚合得到收入信息。或许有的表是销售的总量，但是另外的表则仅仅是可接受的通用会计原则（GAAP）范围内一部分销售量。

规范化的事实可以直接进行比较，甚至可以直接进行数学计算，如求和、计算比率等。如果事实表的管理员达成一致意见，则对于一部分或者所有的事实表的数据准备步骤就可能是对事实的转换，以满足通用定义的需要。

事实表提供者

尽管这一部分超过了操作讨论的范围，但是我们仍要完成本章描述的规范化视图。在下一部分，我们将定义维度管理员这一角色，他是准备并发布规范化维到整个企业的中心权威人士。事实表提供者是维度管理员的接受客户端。事实表提供者拥有一张或者更多的事实表，并且负责最终用户如何访问他们。如果事实表参与了整个企业范围内的钻取操作，则他们必须使用由维度管理员提供的规范化维，并且必须仔细准备数字化的事实，这些事实被组织确认为规范化（标准）事实。

维表管理员：发布规范化维来影响事实表

规范化维必须是集中管理的对象。组织必须要认命一个主要的维度管理员来管理和发布任何规范化维。

当维表管理员者发布了维度的新版本时，事实表提供者有义务立刻更新本地的维表拷贝。在理想情况下，发布的维度在每一个纪录都包含一个版本号，当钻取应用在准备报表的最后一步把分隔的答案绑定在一起的时候，则必须要使版本号相等。如果事实表提供者滞后更新维，则由于版本号不匹配，钻取应用将失败。尽管这听起来有些苛刻，但是强制执行这一规定是非常重要的；同一维度的不同版本将导致隐藏的，不可预知的错误。

每一个规范化维都必须在每一条记录上都有一个类型 1 的版本号字段（如果不了解，可以详见第五章有关于类型 1、2 和 3 缓慢变化维的讨论）。一旦维表管理员针对各个事实表发布了维，则在每一条记录中的版本号就会被覆盖。对于任何从两个及以上的事实表使用两个及以上的维拷贝进行关联数据的钻取查询都必须要确保维度版本号的严格匹配。这就需要维表管理员同步复制任何修订过的维度到所有事实表。在支持事实表间钻取查询的环境中，强制维度版本的失败是非常严重的错误，因为虽然请求可能会非常好的完成，但是汇总和分组则可能完全错误，并且也没有在最终报表上检测不一致的有效方法。

对于单一机器、单一 DBMS 上的单一表空间来说，由于只需要有一份维表的拷贝，规范化维管理就显得非常简单。这一拷贝在查询时关联到表空间中已存在的所有事实表。然而，这一好处只体现在最小最简单的数据仓库中。一旦事实表出现在是多个表空间，多个 DBMS，或者多个远程设备上时，则维表管理员必须要负责上一段描述过的所有职责，以便完成跨越多个数据集的钻取操作。

必须再次强调，事实表提供者和维度管理员的角色不但对在物理上分布的数据仓库环境中有效，也适用于高度集中的由一组 DBA 管理的单机数据仓库。一旦事实表分散在各自的表空间中，则所有这些问题都有相关性，因为他们必须拥有多个物理的维表拷贝。

规范化维的详细提交步骤

创建规范化维不仅仅只是在某一标准描述的维度属性上达成一致。在下面的步骤中，涉及类型 1、2 和 3 的缓慢变化维的内容将在第五章中解释。维度管理员必须要做：

1、增加新的记录到规范化维，产生新的代理键。

2、为类型 2 的变化增加新纪录到现有维度条目中（在某个时间点及时的真实的物理变化），并生成新的代理键。

3、为类型 1（覆盖）和类型 3（交替变化）的变化修改记录，但不改变代理键。一旦完成了任何类型 1 或 3 的变化修改，则更新维度的版本号。

4.为所有的事实表提供者同步复制修订后的维度。

接收的事实表提供者需要完成更复杂的工作。他必须要：

1、接收或者下载更新的维。

2.处理那些标记为新并且当前在用的维度记录，修改映射到代理键的当前键。

3、处理那些标记为新并且已经过时的维度记录，这将触发一个复杂的过程来进行通常的代理键处理（在第 5、6 章中描述）。

4、用正确的代理键替换自然键后，把所有的新记录添加到事实表中。

5、为纠正的错误、聚合的快照和过期的维更新来修改所有的事实表，最好是使用按分区的方式来完成。详见第 9 步的注释。

6. 删除无效的聚合。只有当类型 1 或者类型 3 的变化发生在聚合的目标属性上，或者历史事实记录在第五步中被修改的时候，当前的历史聚合才会变得无效。其他属性的改变不会使聚合无效。例如，某产品的口味属性的改变就不会使基于类别属性的聚合变得无效。

7.重新计算受影响的聚合。如果维度的新版本没有更新版本号，则只有当加载了新的事实数据时才需要重新处理。如果维度的版本号改变了，则在第 6 步中的删除聚合表后，整个的历史聚合将重新计算。OLAP 系统会自动处理这些步骤。

8.确保所有的基础表和聚合表的数据质量，确保正确计算聚合表。

9.把更新的事实表和维表上线。有可能使事实表（或者更确切的说事实表的一部分）短期下线的的详细策略在《生命周期工具箱》一书中有描述。

10.通知最终用户数据仓库已被更新。告诉用户是否发生了主要的变化，包括更新维度版本号，添加过时的记录以及修改历史聚合。

实现规范化模块

为了实现规范化维和事实，规范化子系统需要参考元数据，这些元数据捕获了来自源系统的有效值集和规范化的维度属性值以及规范化的事实值之间的关系。

许多 ETL 工具支持这些类型的映射，可能用预先建立的元数据属性，或者让 ETL 小组针对源表对象使用扩展元数据属性。图 4.11 显示了一个支持数据规范化的元数据表的例子。表和列的实体分别捕获了每个表和它相关列的元数据。事实表记录了来自每一源系统的正式的规范化值定义。整个源系统的标识符在源系统表里得到。列维度包含了被映射到正式的规范化值的源值。因此,正如在前面简单的例子中引用到的，如果 Male 和 Female 是性别的目标规范化值，则事实表将会在源系统 A 中用 M 关联 Male，用 F 关联 Female；而在源系统 B 中，用 M 关联 Male，但是用 W 关联 Female；在源系统 C 中，用 Man 关联 Male，用 Woman 关联 Female。

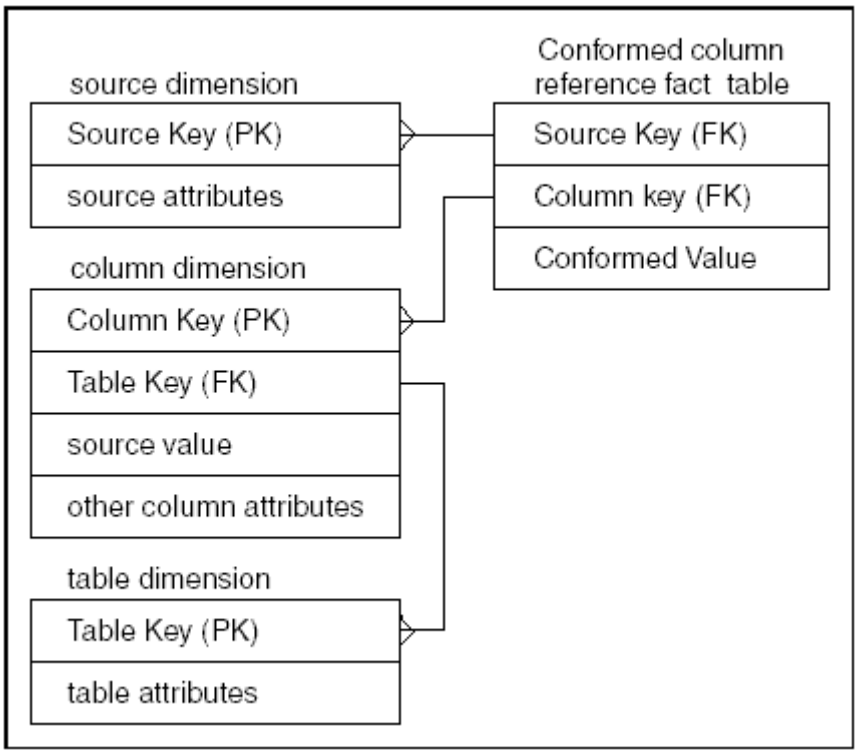


图 4.11 规范化的列支持结构

对于记录中含有无效值的列，即值不在列的维表的有效值范围内，应该用一个来自标准化值参考表的预定义的数值（如 Unknown）来替换；这一替换操作应该记录到错误事件事实表中。

很重要的是，对于不能被标准化的假的或者非法的数据应该让其在下游 ETL 流程（如匹配）中消失，并且对最终用户也不可见。

更复杂形式的标准化通常被用在处理名字和地址的数据清洗方面。支持这一过程的专用软件工具的功能对于 ETL 团队来说想要进行复制是非常困难的。一定要参看一下第 7 章列出的这个竞技场中的主要运动员。在某些情况下，标准化的值使用某种技术按概率获得，这种技术使用统计方法根据常用的已知名称或者地址来排列那些不准确的数据。要知道一些概率标准化工具也有自调整的集成引擎，它们可以随时间累计学习更多的数据分布情况，特别是针对特殊的应用，然后合理地调整它们的处理算法。这是一个强大的功能，同时也向 ETL 构架师的能力提出了挑战，他们要测试出哪一种数据集成引擎会变得越来越聪明。大多数的标准化工具对其再造数据过程及处理数据过程中遇到的异常和错误进行反馈。捕获,保存和挖掘数据集成的线性过程非常重要，并用之后留下的日志表创建错误事件事实表。

匹配驱动去重复

匹配，或叫做去重复，包括标准化记录重复值的删除。在某些情况下，重复纪录可以通过在一些关键列中特定值的出现来很容易地探测到，比如社会保险号码、电话号码或交易卡号码。不幸地是这样轻松的情况是很少见的。在其他很多时候，并不能发现这样明确的匹配，删除重复值的可用的唯一线索是几乎能匹配的多个列的相似性。还有更困难的情况,确定找到了多个定义匹配的列，但是它们又互相矛盾。

专用数据集成匹配工具现在已经非常成熟了，并且在广泛的使用，就是用来处理这些非常特殊的数据清洗问题的。这些工具通常和数据标准化工具有关系密切，并且经常打包一起出售。

匹配软件必须把数据流中的记录集与规范化维记录相比较并且返回如下内容：

- 匹配相似性的量化得分值
- 在标准化记录范围内把输入记录与规范化维实例链接起来匹配键的集

这样，一个通过匹配处理过程运行的输入记录可以是匹配空或者是一个规范化维的记录，或者匹配 0 个，1 个或者是更多的在批处理队列中的其他输入记录。在任何情况下，匹配软件的作业将把这些详细描述了导出匹配关系的输入记录和匹配键进行联合。这些匹配键被在下一部分描述的 survivorship 模型使用，它指出了哪一个记录匹配到了另一个记录，然后把它作为整合到单一集成记录的候选。

许多数据匹配工具也包括匹配得分，或叫匹配置信度量，那描述了获得匹配的相似度。通常，这些匹配得分通过创建多个匹配方法或多路来获得，每一路的匹配相似度得分的结果应用到推荐的一组匹配键和一个全局权重得分。

需要有稳定的去重复能力的组织可以选择维护一个之前匹配数据的永久性库，其中每个数据项仍然和单一的数据提供者相关联,使用这个统一的的库来改进匹配的结果。这样，匹配引擎不但可以在规范化维记录上应用匹配过程，而且还可以在之前匹配的来自所有源系统的维记录的完整集上应用。这一方式可以得到更好的匹配结果，因为匹配内容更丰富了，而且能更好的面对处理匹配规则的变化，因为现在不需要通过整个数据集成过程运行所有源系统的数据就可以满足了。但是它也使匹配过程变得复杂，因为匹配可能发生在数据源内部和外部，甚至是完全规范化的数据集。

正如本文所写的，匹配工具并非是那种能插入到 ETL 过程中而且自动知道如何去做的简单的拧钥匙实现方式。相反，它们需要基于组织的数据做大量的评估和训练工作，决定合并哪些属性 (匹配过程或者角度) 的匹配策略的建立很可能是重复的先兆，把不同的角度集成到一个匹配策略的调整，以及基于组织的承受能力设置积极的或保守的匹配和不匹配阈值。ETL 工具包厂商正在试图完成这一应用目标，然而，应该检查一下这些产品的内嵌的匹配转换器。

共存：规范化的最后一步

共存性是指整合一组匹配（去重复）记录到统一的映像，它把每一条匹配的记录组合成最高质量的列值来建立规范维记录。这就建立了业务规则，它为来自所有可能的数据源的列值定义了一个架构，并捕获了生成共存记录（规范化记录）时应用的源数据到目标数据的映射。

此外，共存性必须能够针对列的组合，而不仅是单个的列。这是为了某些情况的需要，这时单个可共存列的组合可能导致毫无意义的混乱结果，例如，把来自三个不同的源系统的 1，2，3 地址行组合在一起生成一个组合地址，则这个地址记录的可信度还没有这三条任何一个高。对于这种情况，最好是创建规则来要求某些列的联合（共存块）能够在一起共存：所有的或者没有。图 4.12 中所示的元数据表支持最常用的共存需求。

- 共存源到目标映射表捕获了在源数据列（已清洗但没有规范化的输入数据）与目标数据列（规范化维表的列）之间的数据集成映射关系。为了兼顾灵活性与简便性，允许任何列的组合可用做到任何目标数据列的组合的源，这也使得 ETL 架构为了合理地生成它带来了的负担（不仅要考虑参数完整性，还包括更加复杂的结构）。
- 共存块表把映射的源到目标的分组成块，这些块必须能够在一起共存（合理解决前面提到 1，2，3 类地址的问题）。共存块允许只有一个源和一个目标，因此通过强制所有共存性按块执行，可以简化元数据模型和共存的处理过程。该表包含一个排序，它使得可以使用动态 SQL 来确定源系统字段块的优先级，该 SQL 按照共存块源排序优先级按顺序在每一块中查找非空值，并根据匹配键是（UPDATE）否（INSERT）已经存在规范化记录的代理键来设置合适的 INSERT 或 UPDATE 语句。

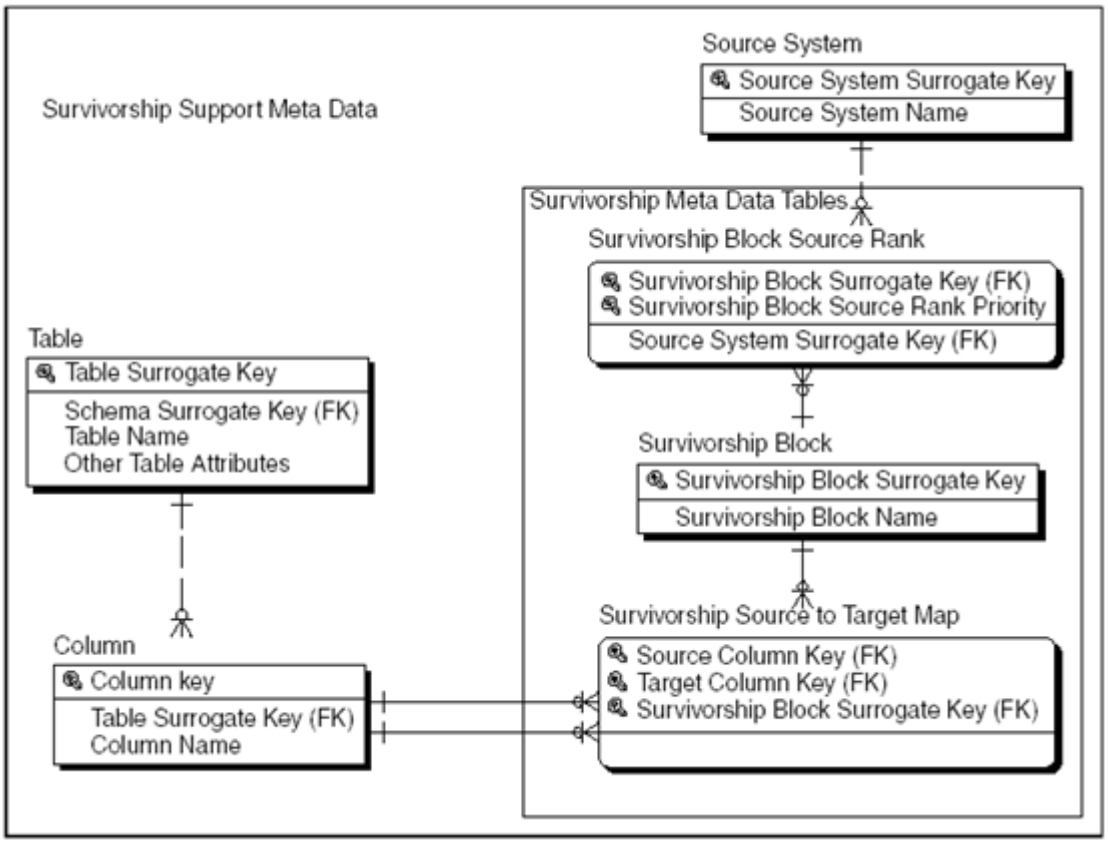


图 4.12　生存支持元数据

对于去重复过程成功的把分离的源实体联合成为一个单一的实体（如客户实体）的情况，如果这些源实体在各自的源系统中已经指定了主键，则需要有一个表对这些过时的主键进行维护，以提高对源系统数据进行去重复处理的速度。

提交

提交是 ETL 最后的重要步骤。在此步骤中，已经清洗并规范化过的数据被实际写入到维结构中供最终用户和应用系统访问。在只由一个表空间组成供最终用户访问的最小经数据仓库中，维表只是简单地写入这个表空间。但对于大型的数据仓库，从含有多个表空间到更广泛的分布式数据集市自治网络，维表必须以一致的方式仔细地发布。提交非常重要，我们将在第 5 章和第 6 章中详述。

5.7 小结

回顾以上细节，本章主要覆盖了四个大的主题：目标、技巧、元数据和度量。

数据清洗和规范化的目标在于减少数据中的错误，提高数据的质量和可用

性，并标准化整个企业共享的关键描述属性和量化度量。

数据质量技巧涵盖了从数据库级别校验单个字段的定义检查（列属性约束），到校查字段到字段之间的一致性（结构约束），再到数据的特定业务规则的检查（数据和值规则约束）。数据质量处理的最后阶段（规范化和去重复）也就实现了，在这一阶段解决跨越多个数据源的数据差异问题。

数据质量元数据包括所有技术的定义和业务规则。我们描述了构建一组数据过滤器的方法，每个过滤器都表示一个数据质量 investigation。一些过滤器作为每一个 ETL 过程的一部分定期执行，还有一些是偶尔运行来进行检查或特定的 investigation。常规过滤器提供了诊断指标和度量，并存储在详细的错误事件事实表和与事实表关联的审计维表中。这些审计维非常有意义，因为在某种意义上它们把元数据提升为真实数据。数据质量指标可以参与到最终用户的查询中，就像它们是普通数据一样。

最后，我们建议的数据质量度量是 ETL 团队需要的度量的起始集，它是为了创建一个综合的数据质量处理流程。

当数据经过了数据质量处理流程后，就已经为最后的提交步骤做好了准备，这将在第 5、6 章中详细描述。

6 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。

6.1 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。

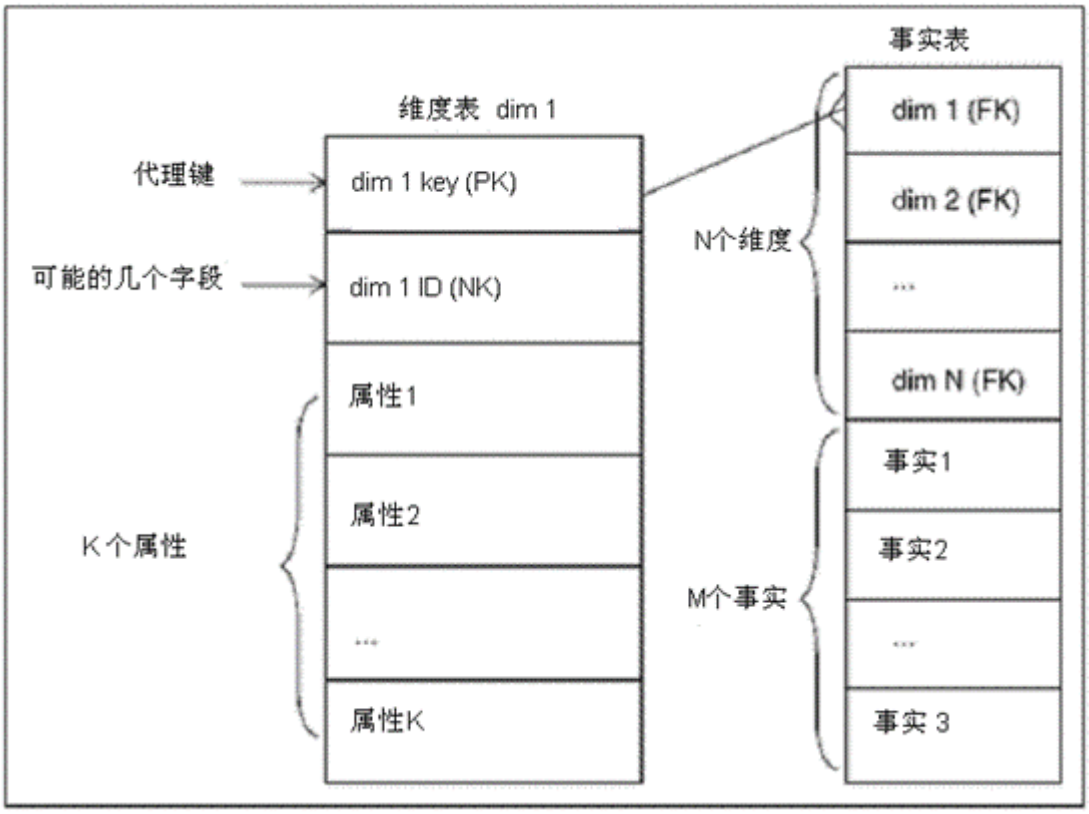


图 5.1 维表的基础结构

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。

6.2 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。

6.3 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。

6.4 扁平（Flat）维度和雪花（Snowflaked）维度

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。

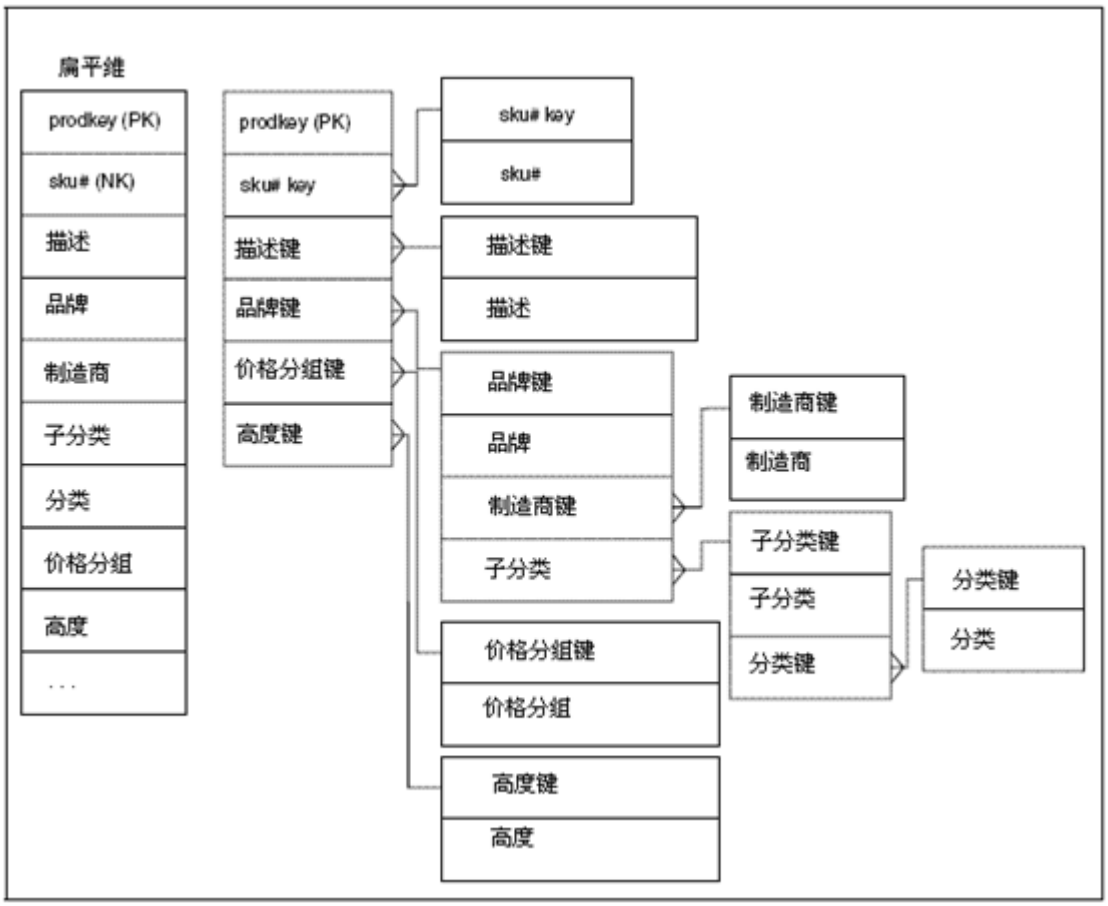


图 5.2　维表的扁平模型和雪花模型

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。

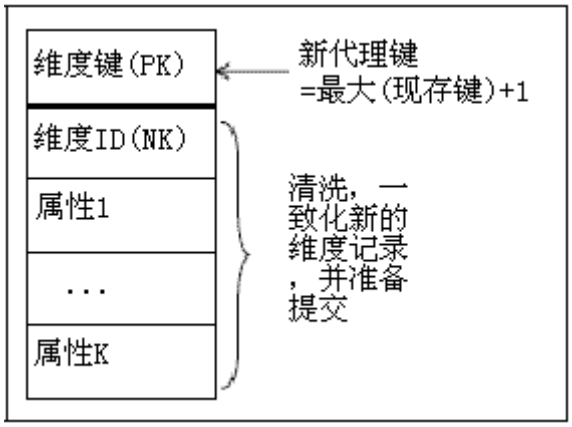


图 5.3　维表模型代理键赋值

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。

6.5 日期与时间维

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。

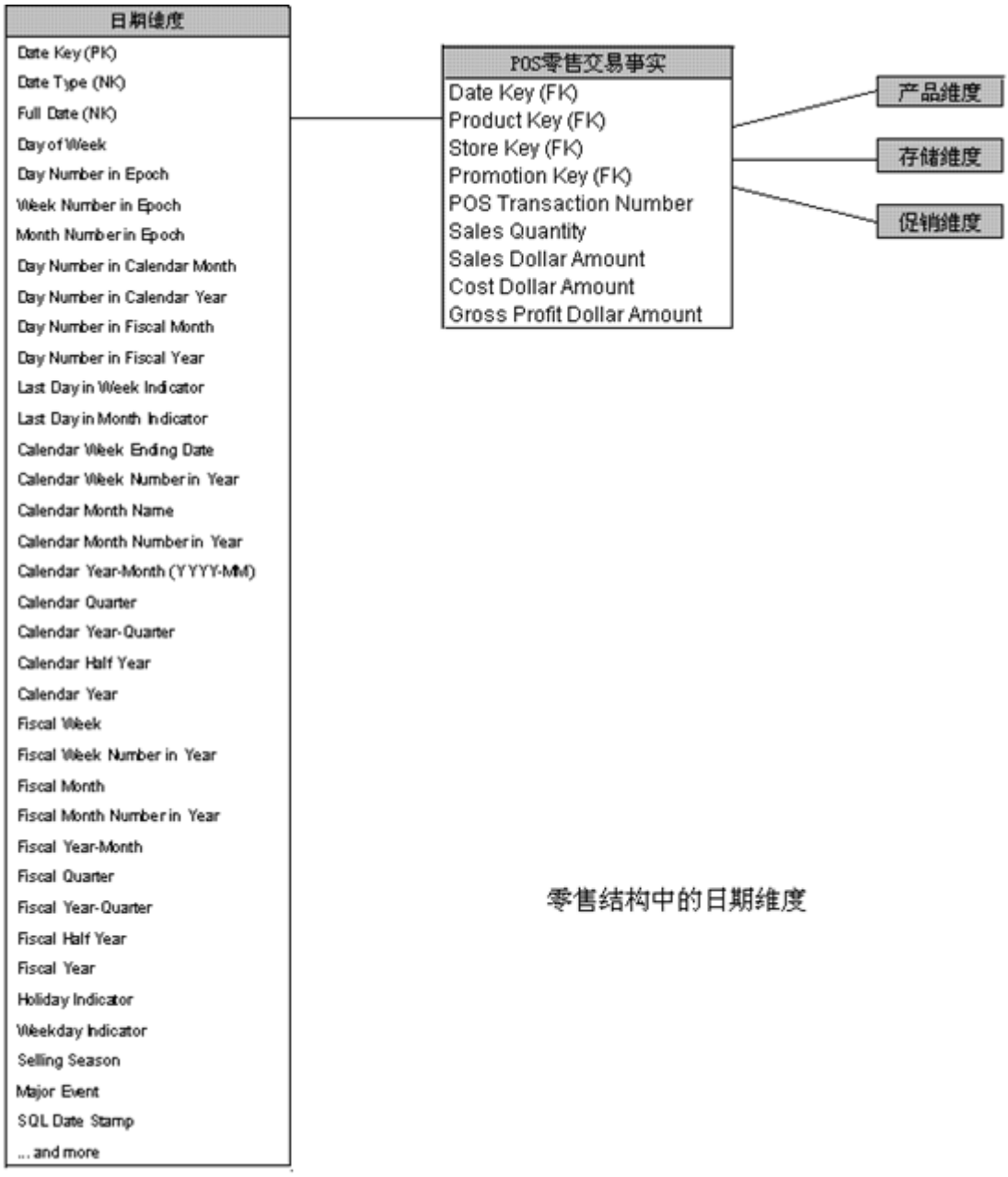


图 5.4　日期维度所需属性

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。

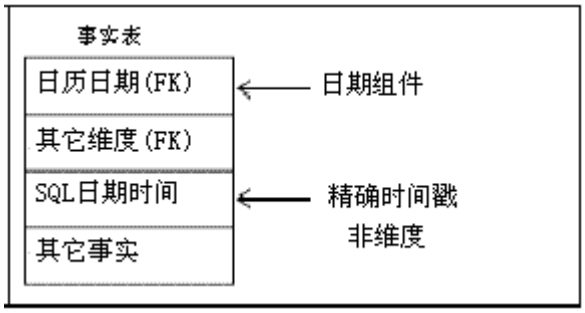


图 5.5　处理精确时间度量的事实表设计

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。

6.6 大维度

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。

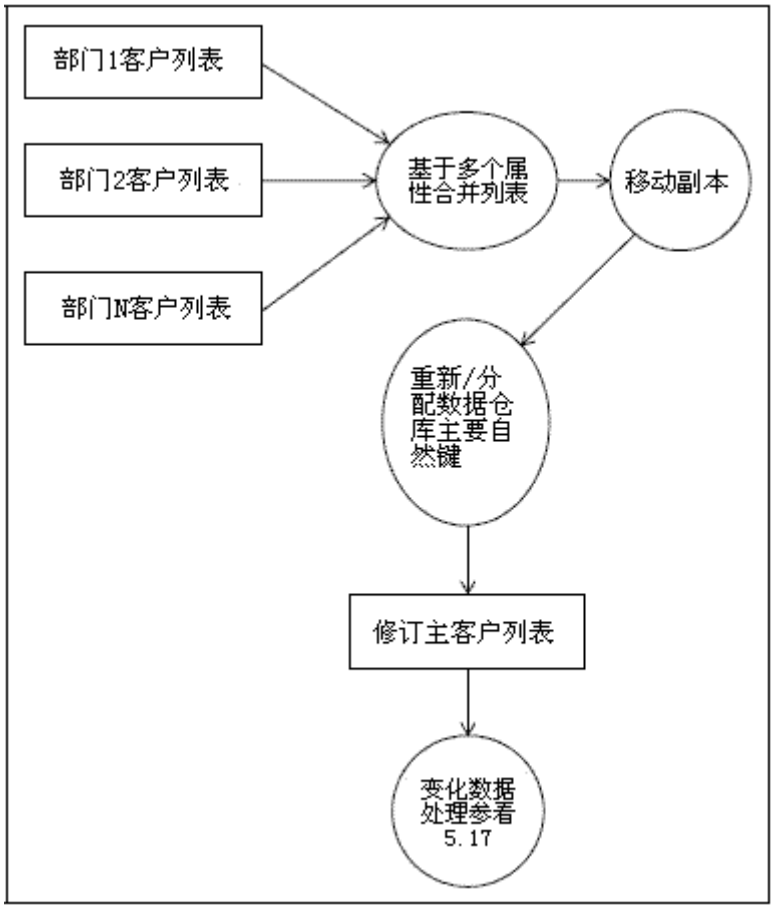


图 5.6　合并与剔重多个客户集

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。

6.7 小维度

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。

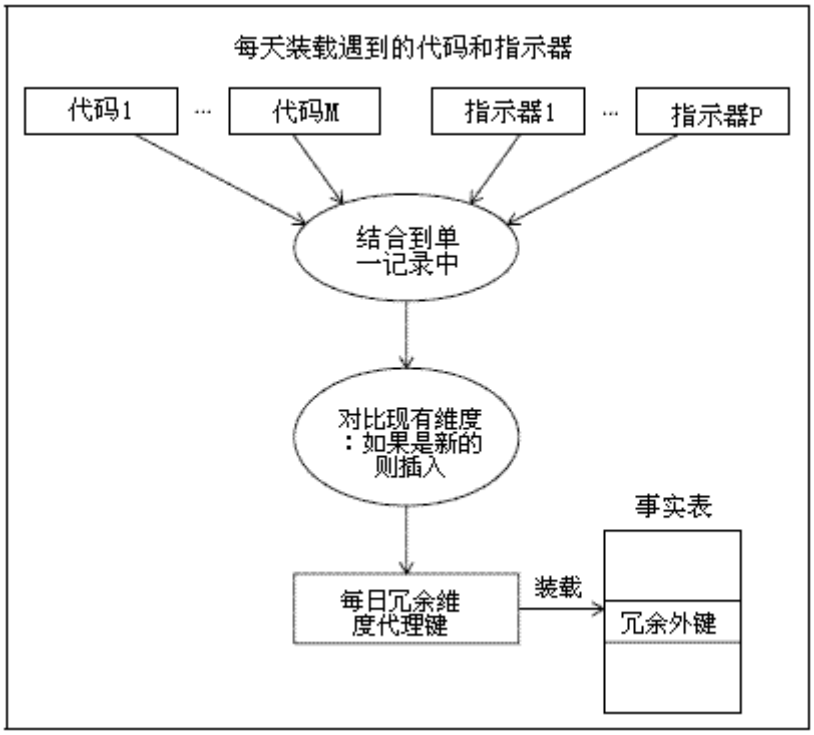


图 5.7　冗余维的 ETL 数据流

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。

6.8 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！

6.9 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。.

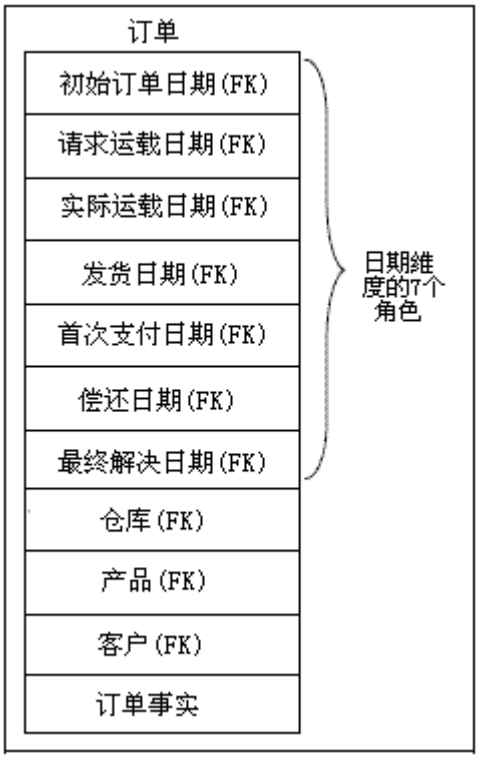


图 5.8　典型的快照事实表

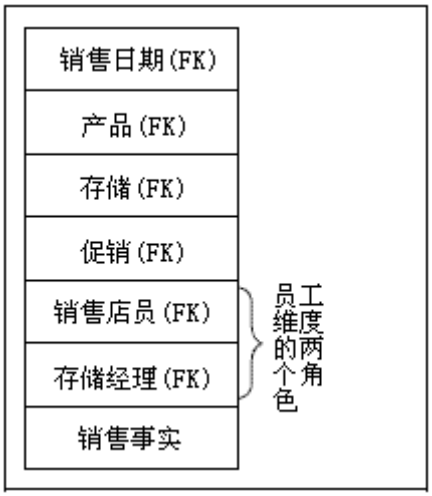


图 5.9　员工角色维

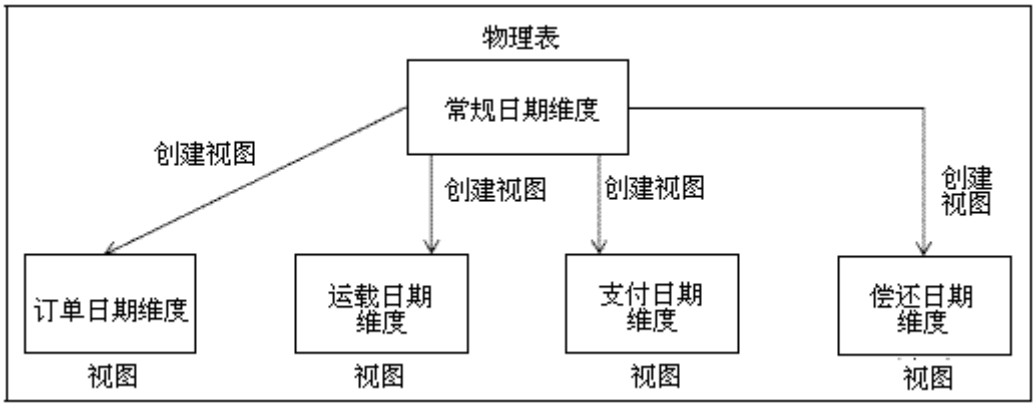


图 5.10　多个日期角色维

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。

6.10 其他维度的子维杜

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。

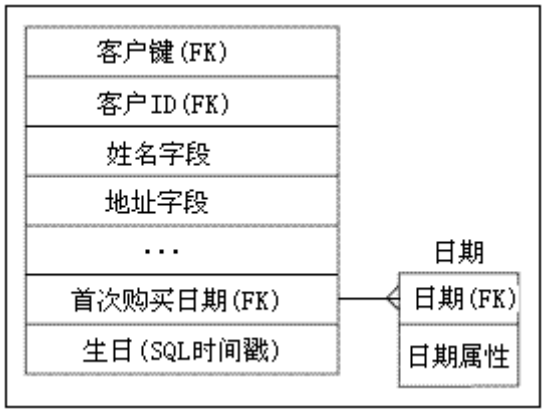


图 5.11　客户维的两个日期处理.

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。

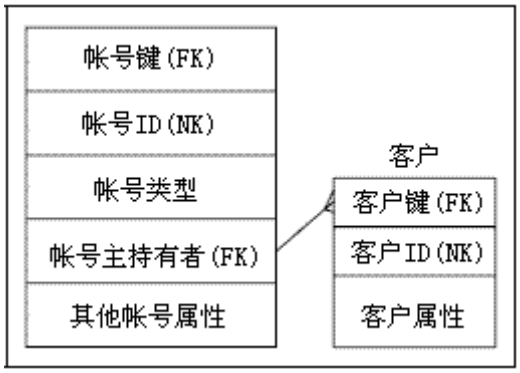


图 5.12　作为子维度的客户维

6.11 退化维

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。

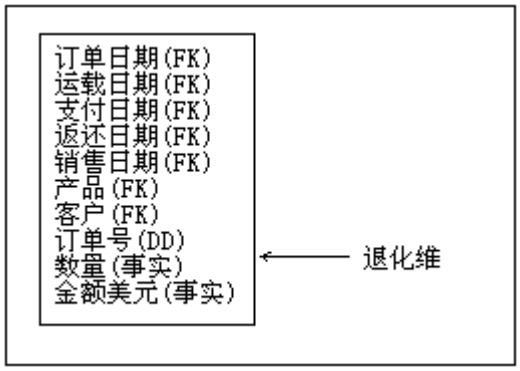


图 5.13　快照事实表的订单

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。

6.12 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。

6.13 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。

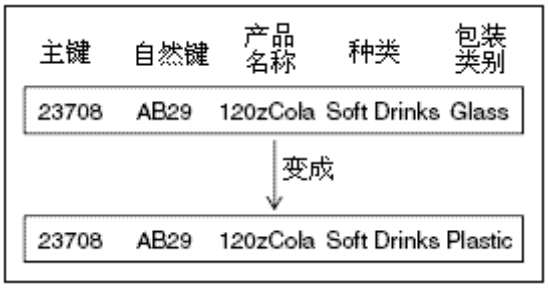


图 5.14 处理类型 1SCD

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。

6.14 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。

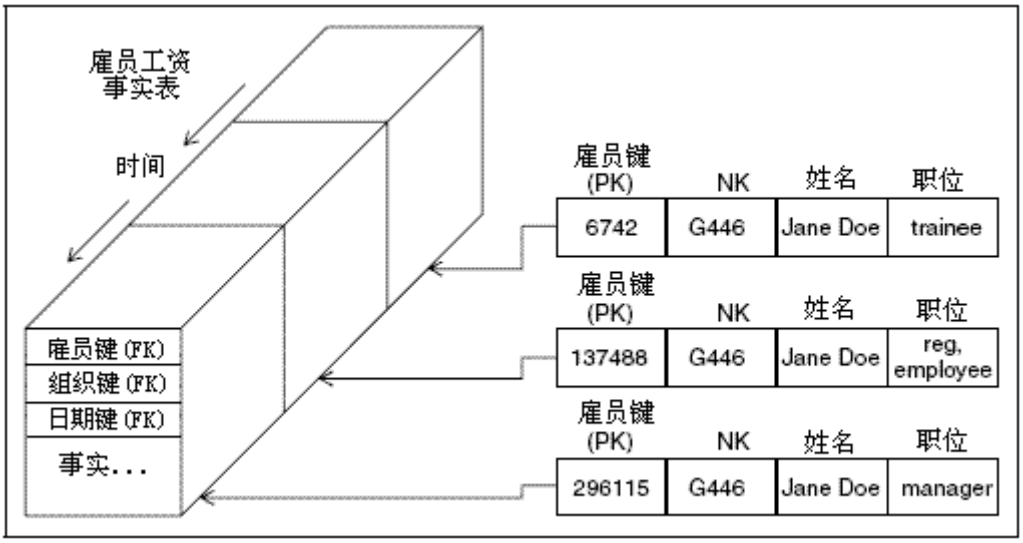


图 5.15　类型 2SCD 很好地分区了历史信息

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。

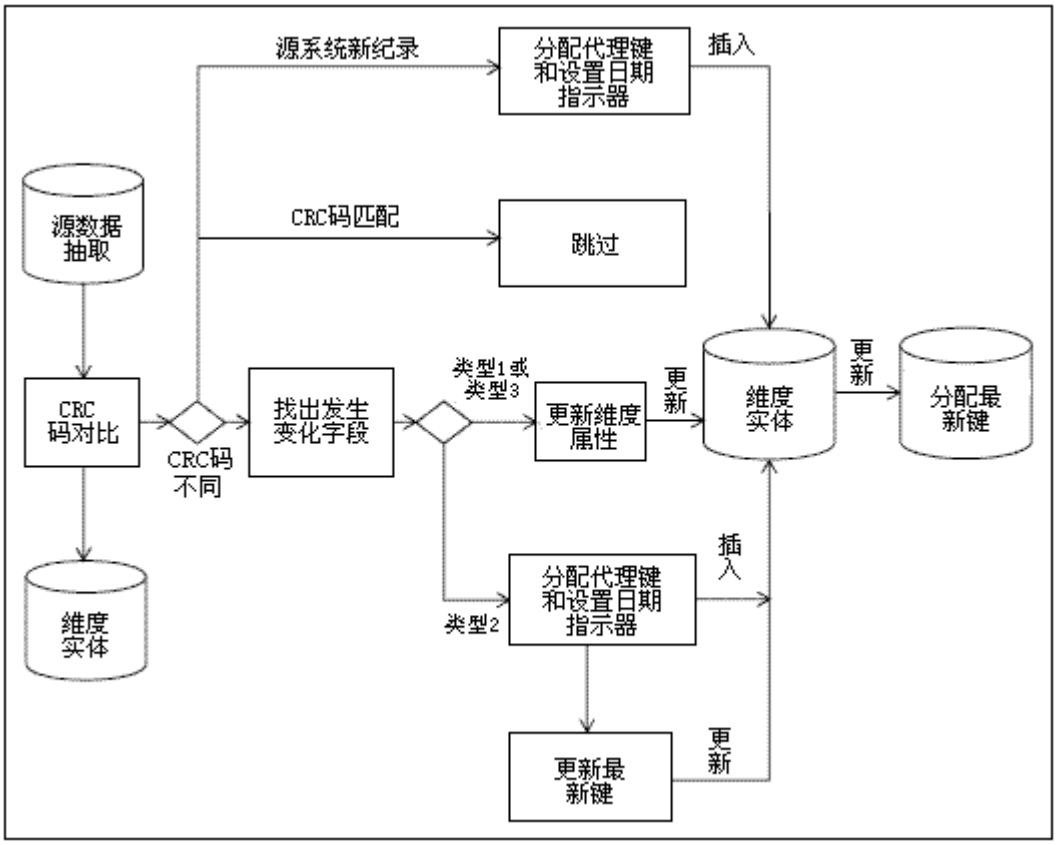


图 5.16　维表代理键管理

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。

6.15 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）。

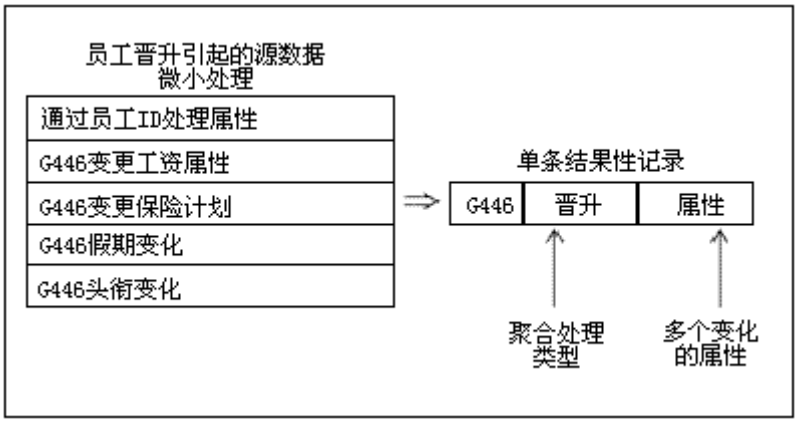


图 5.17　组合源系统的微处理

6.16 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。

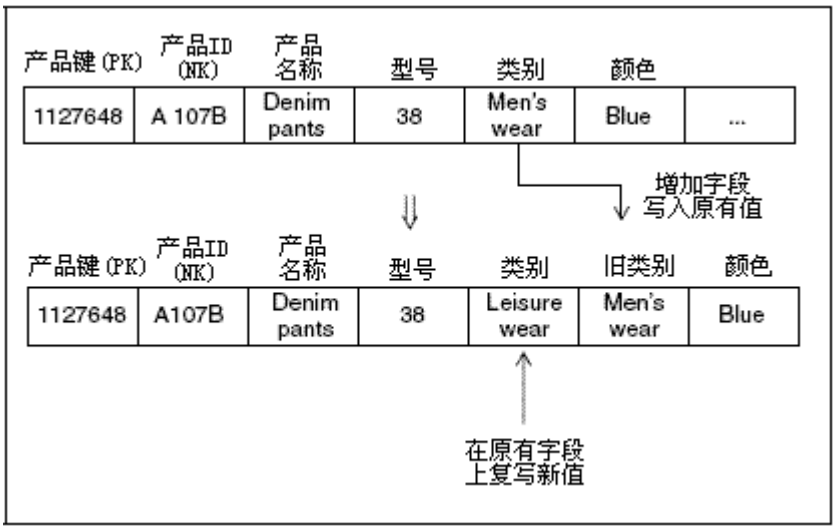


图 5.18　用于产品-种类描述的类型 3SCD 实现

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)。

6.17 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。

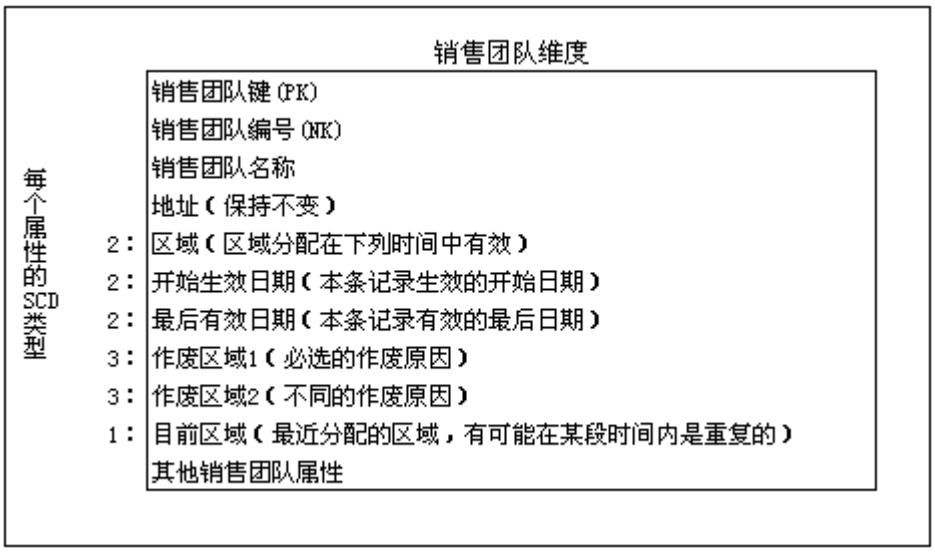


图 5.19　混合 SCD 显示了三种类型

6.18 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。

6.19 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。

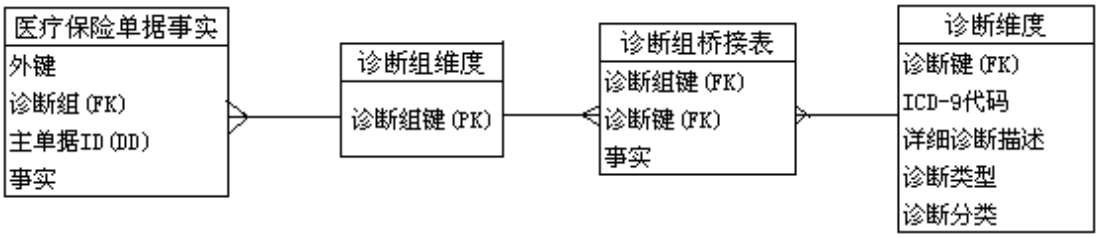


图 5.20　使用桥接表来表示多个诊断

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。

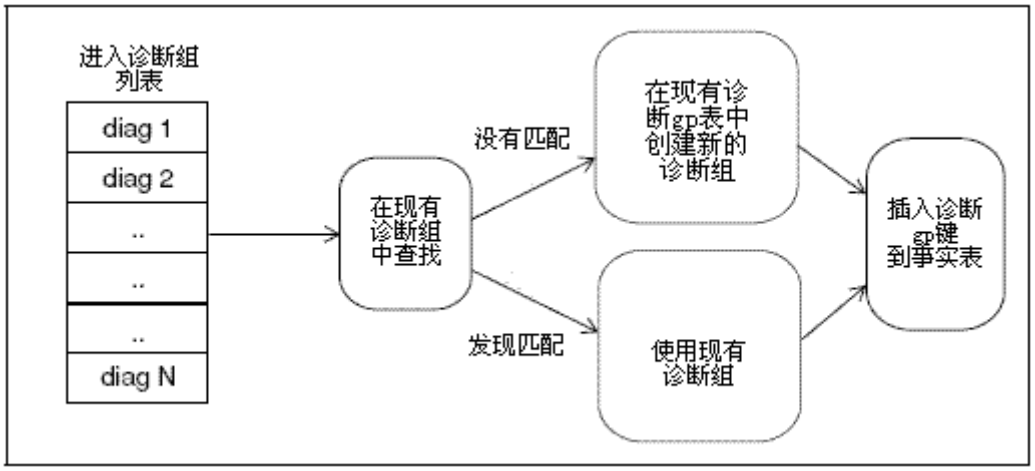


图 5.21　在外部患者设置中处理诊断组

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

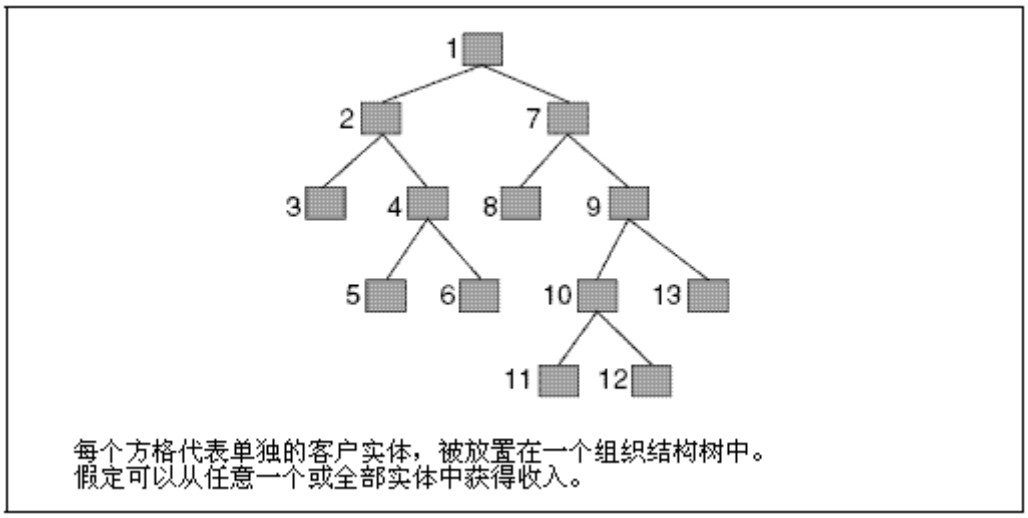


图 5.24　不规则的组织结构层次

6.20 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。

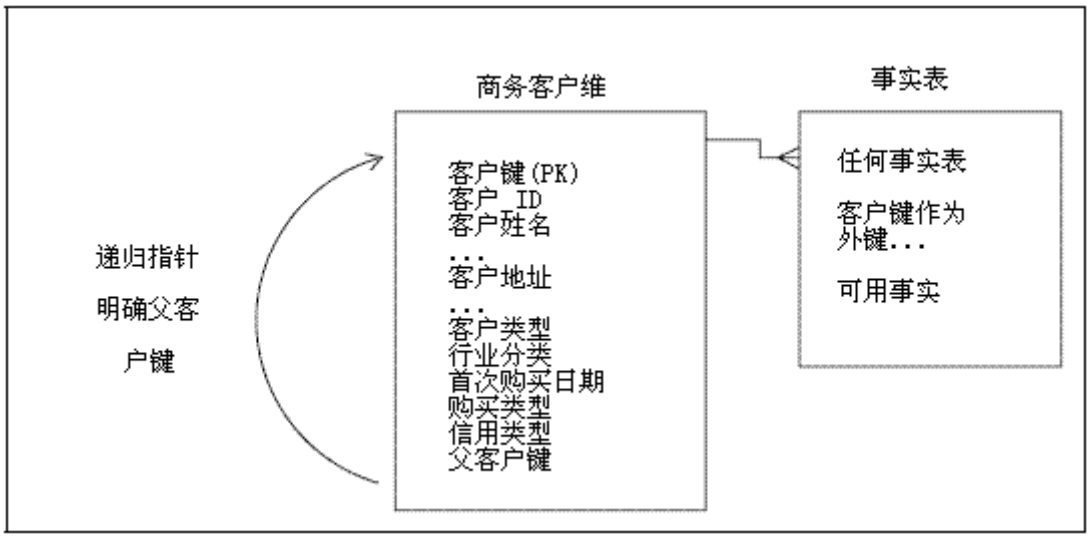


图 5.25　带有递归指针的客户维

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：

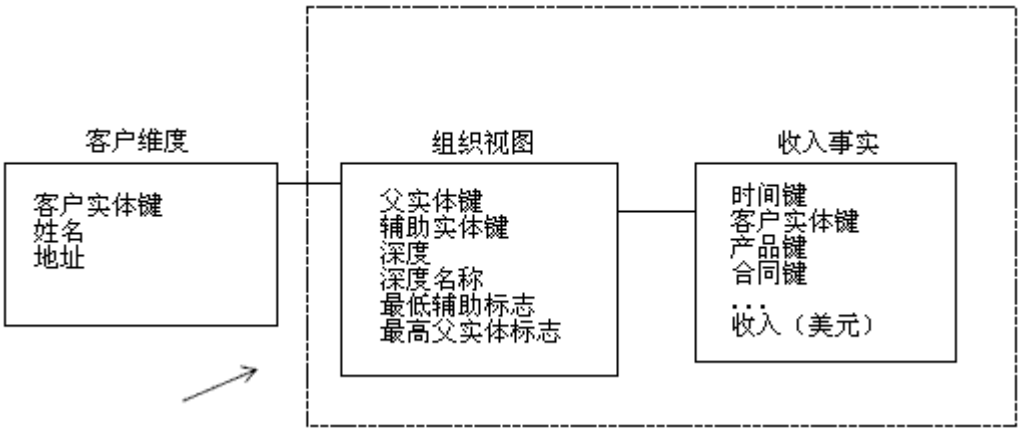


图 5.26　层次桥接表表示客户从属关系

(+)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。

6.21 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。

6.22 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。

6.23 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。

7 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。

7.1 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。

7.2 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。

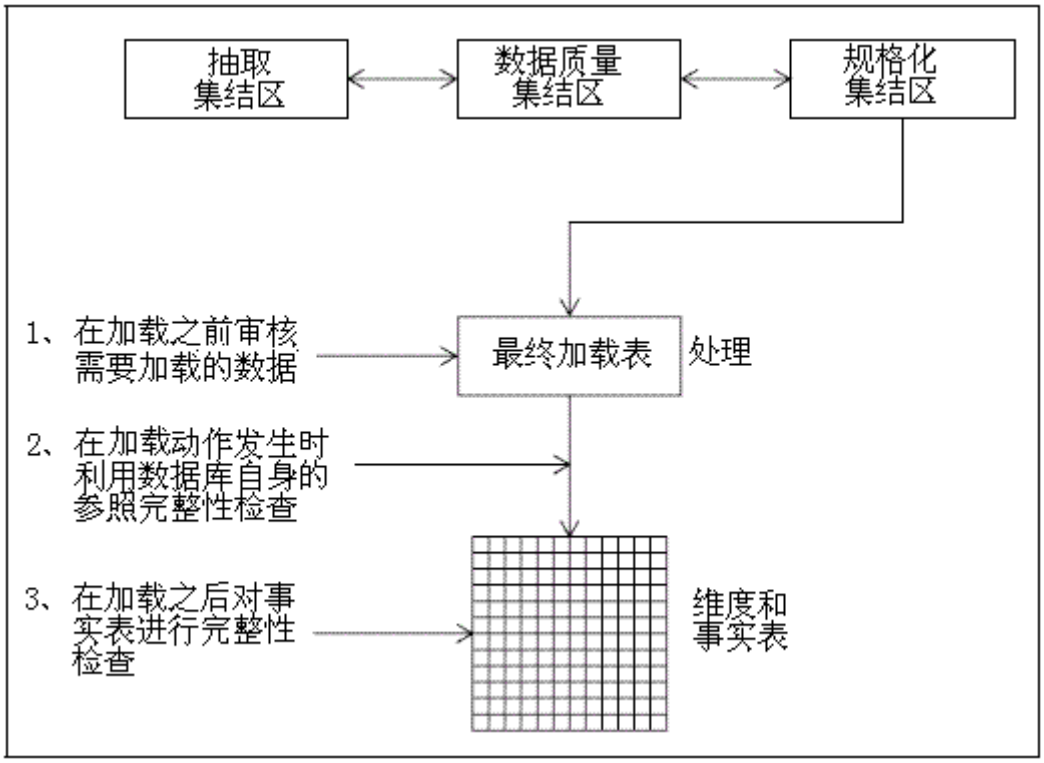


图 6.2　强制参考完整性选择

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。

7.3 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。

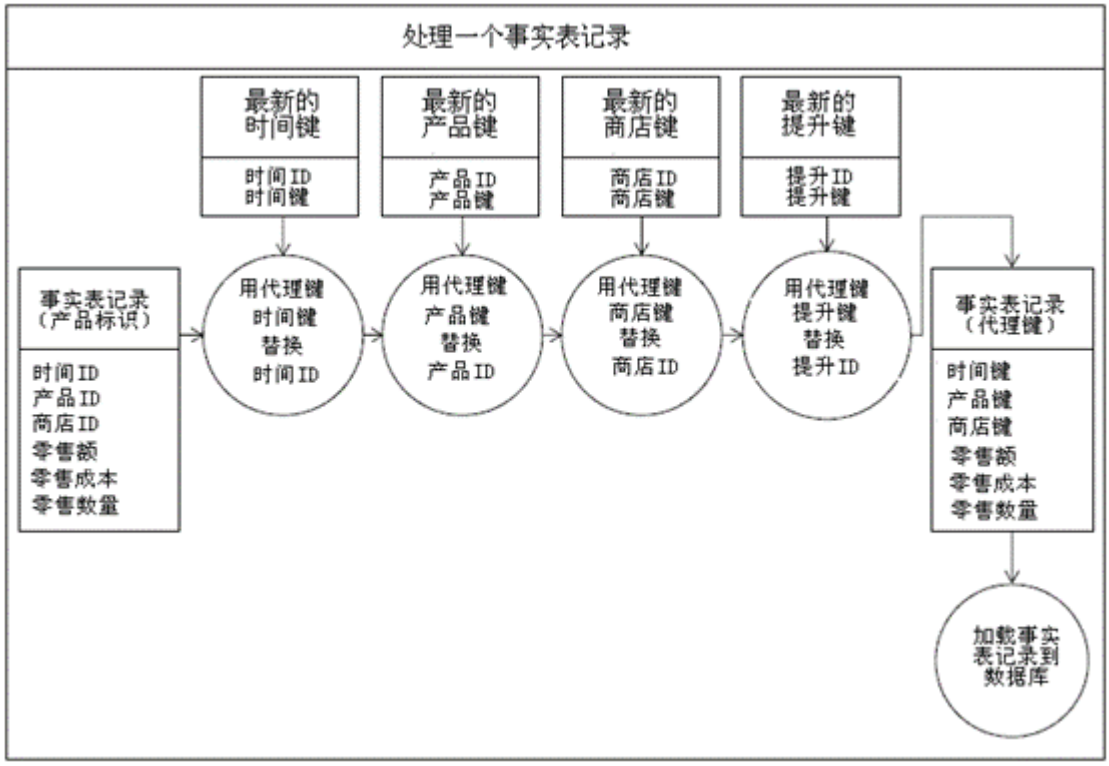


图 6.3　代理键管道

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。

7.4 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。

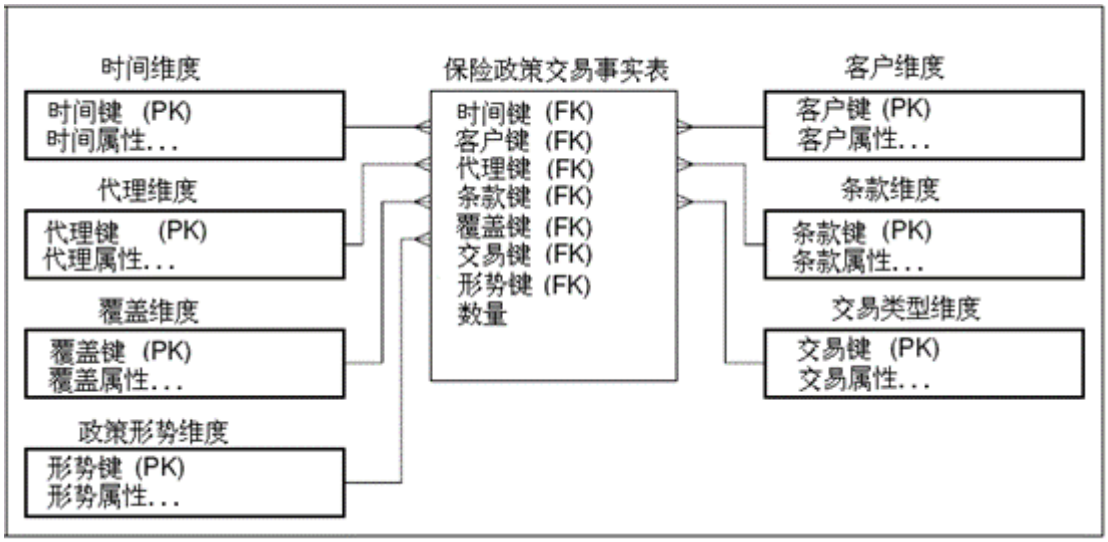


图 6.4　保险标准的交易粒度事实表

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。

月报(FK)
帐户(FK)
机构(FK)
家庭成员(FK)
期末余额(fact)
变更余额(fact)
日平均额(fact)
保证金数(fact)
保证金总计(fact)
回收款数(fact)
回收款总额(fact)
罚款总额(fact)
支付总利息(fact)
日平均储备数(fact)
自动取款机回收数(fact)
外币系统自动取款机回收数(fact)
网上支付回收数(fact)
网上支付回收总额(fact)

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。

7.5 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。

7.6 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。

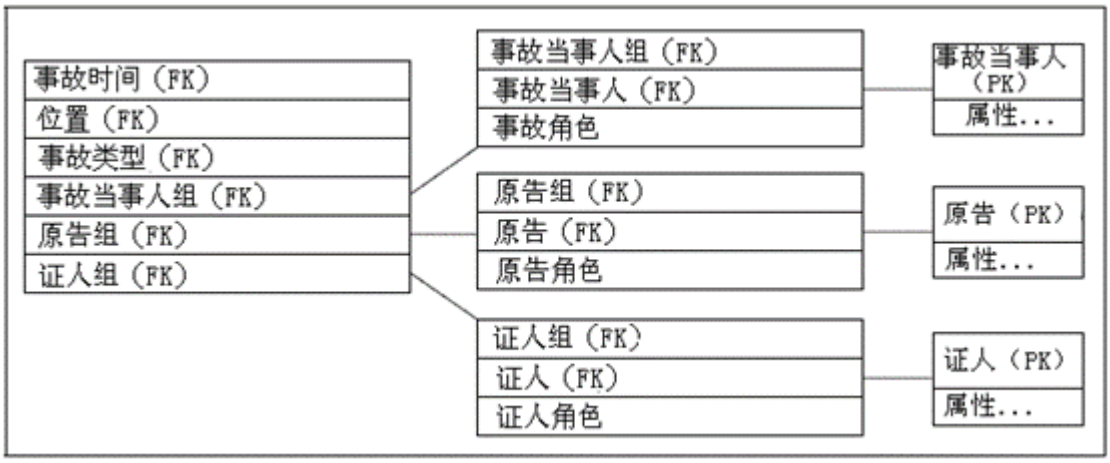


图 6.7　表达车与事故事件的无事实事实表

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。

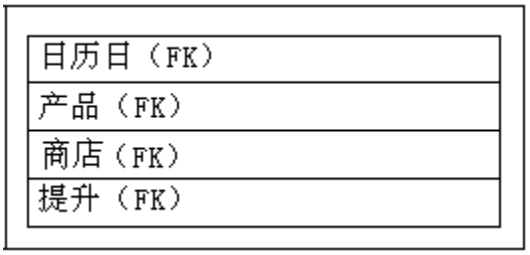


图 6.8　无事实的覆盖表

7.7 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。

7.8 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。

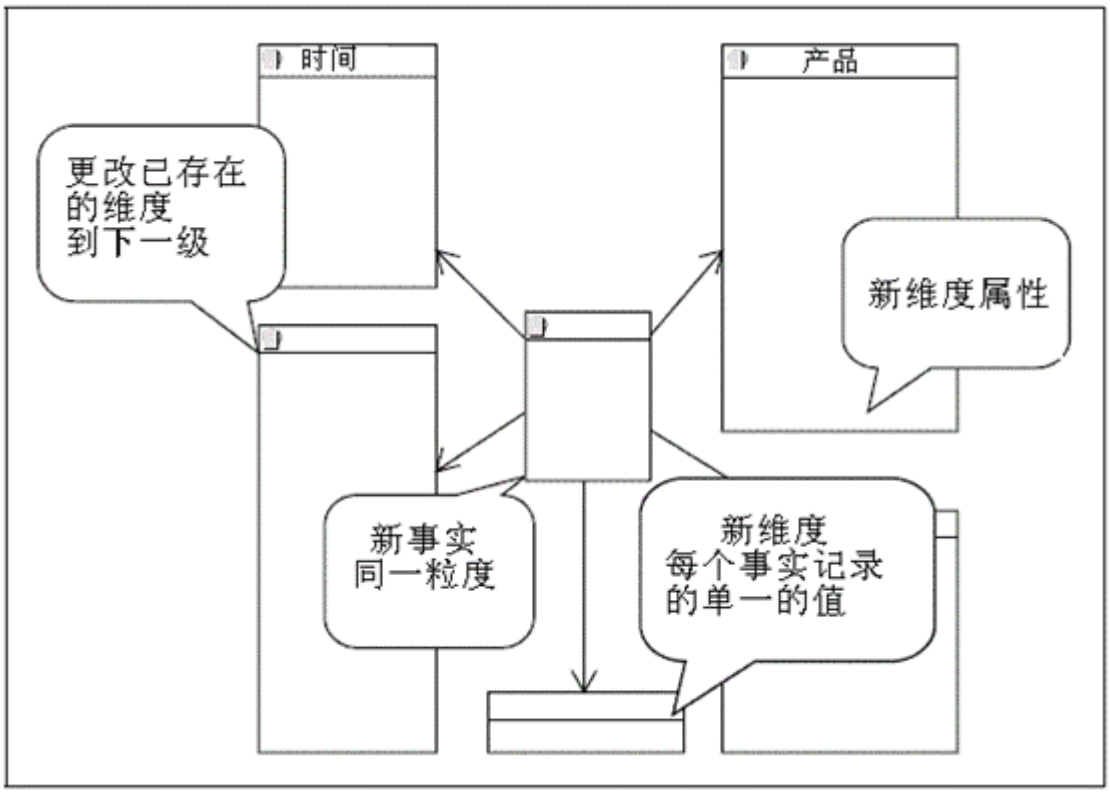


图 6.9　四种类型的优化更改

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。

7.9 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。

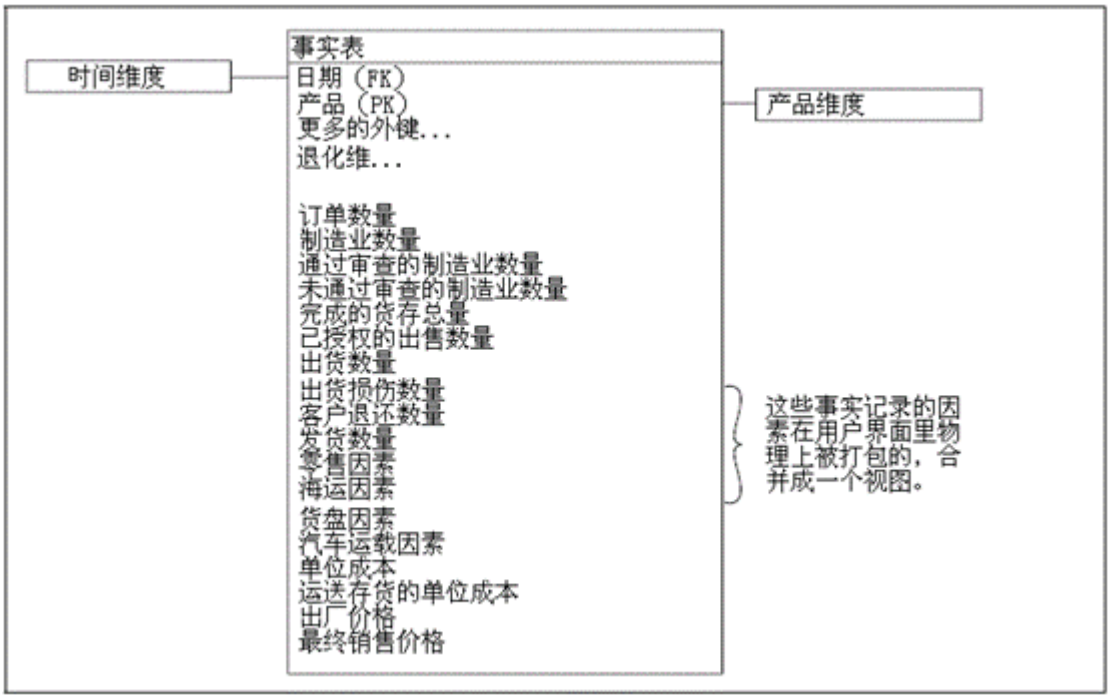


图 6.10　包含 10 个事实、5 种单位、4 个定价模式的物理事实表设计

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。

7.10 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）。

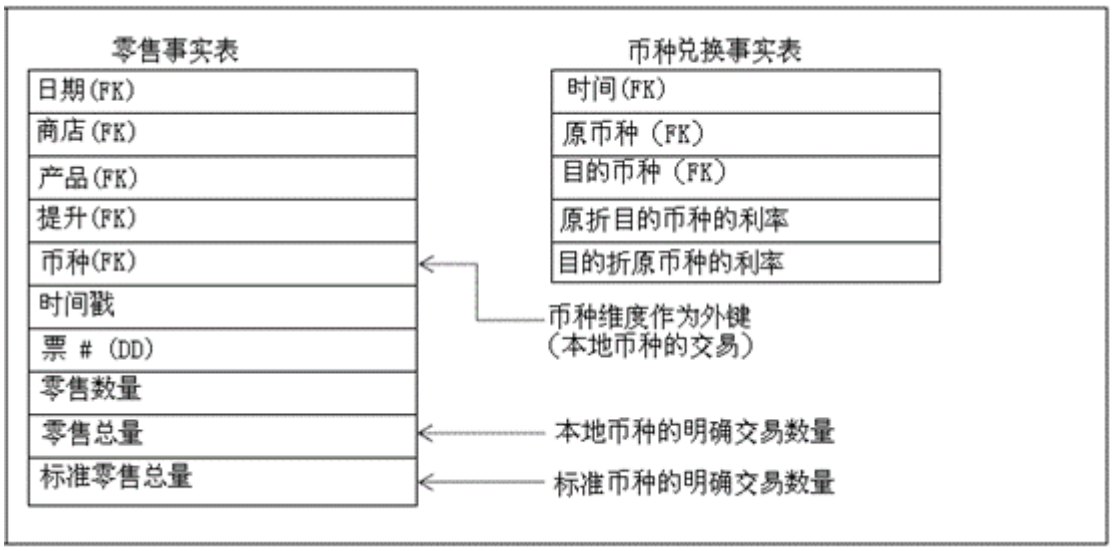


图 6.11　处理多币种的结构设计

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。

7.11 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。

7.12 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。

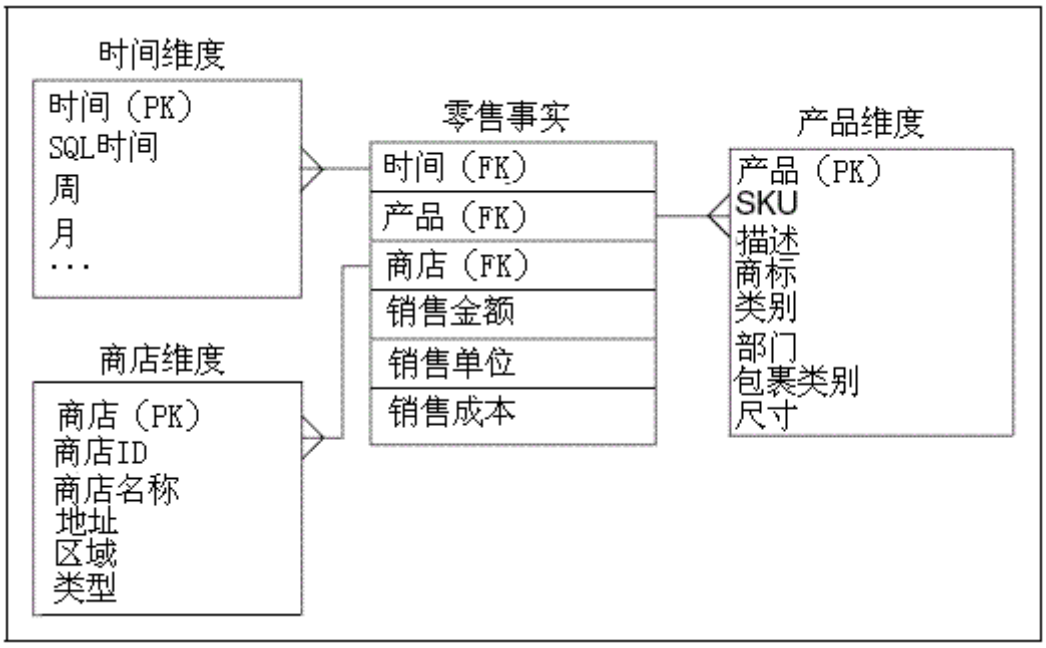


图 6.12　简单的日、产品、商店粒度的简单维度模型

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。

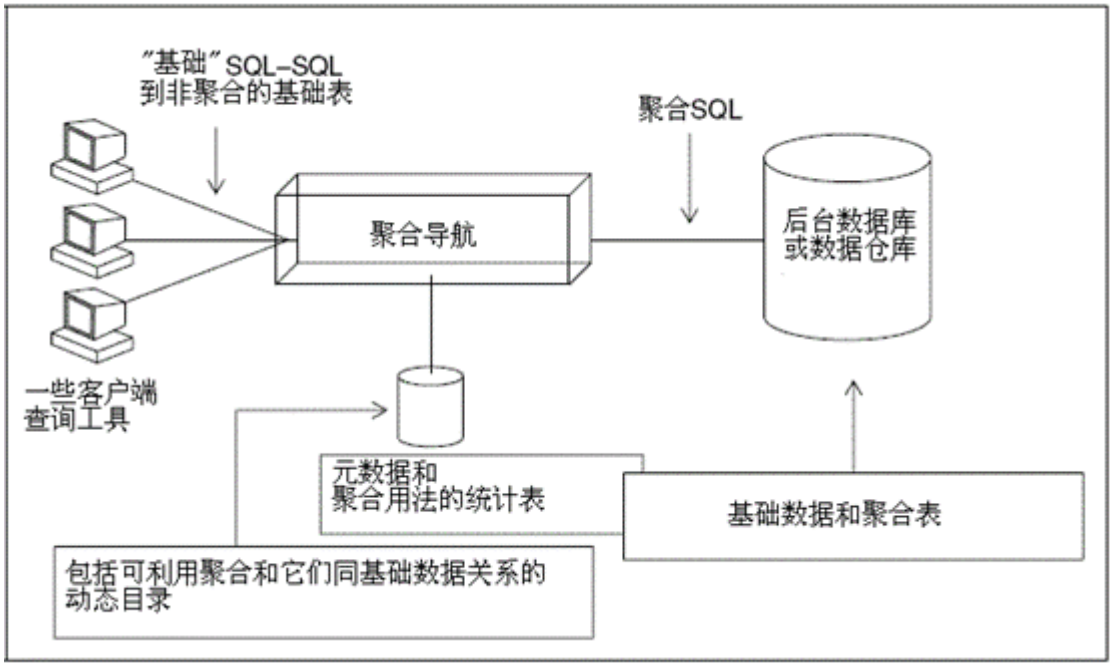


图 6.13　聚合导航架构

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。

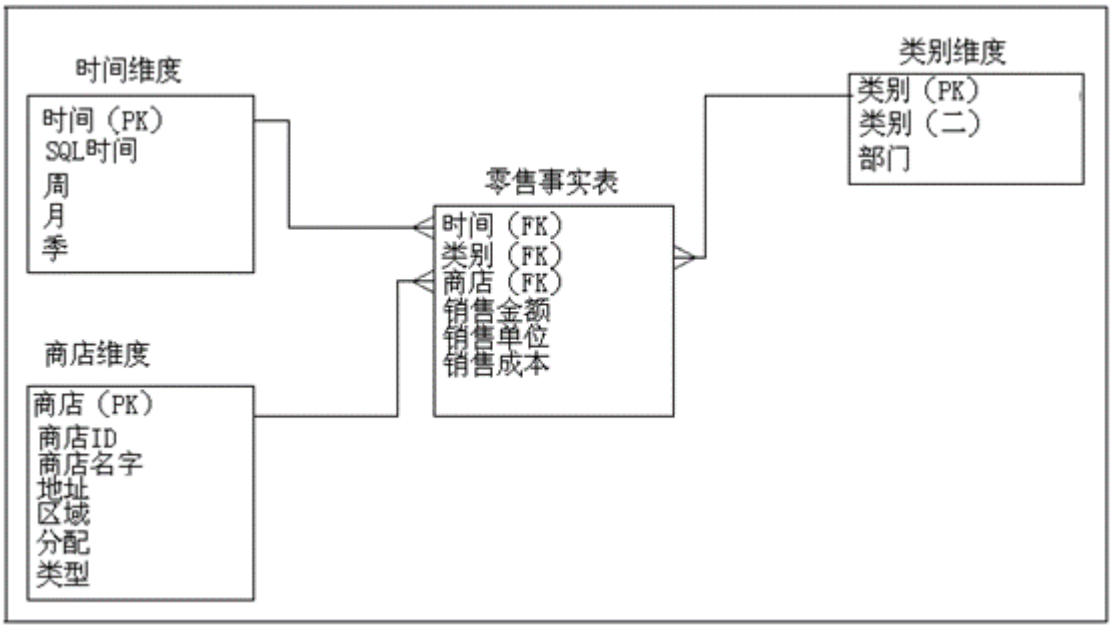


图 6.14　单行道类别聚合模式

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！

7.13 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。

7.14 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。

第三部分 实施与运行

8 开发

没有准备，你就准备失败！

——本杰明.富兰克林

如果你按先后次序读这本书，到现在已经可以了解建立 ETL 系统所面临的挑战的详细模型。我们已经介绍了所需要的数据结构（第二章），必须连接的数据源范围（第三章），清洗和一致化数据的完整结构 a comprehensive architecture（第四章），以及组成最终提交内容的所有目标维度表和事实表（第五和第六章）。我们真诚的希望可以为你的 ETL 系统建设有所启发！

你正处在一个可以为 ETL 系统绘制一个处理流程图的位置，这张图可以清楚的标识出合理的抽取、清洗、规范化以及发布模型的详细信息。

现在是决定你的 ETL 系统开发平台以及开发方式的时候了。如果你是个 ETL 新人，你会遇到一个很大的两难选择：是购买专业的 ETL 工具包，或者计划自己手工编译程序和脚本语言。对这种选择我们曾经在第一章尝试给出了一个公平的评估。或许你应该回过头再读一下那部分。

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*

在下一节，我们给出一张简要列表，包括主要的 ETL 工具包、数据评估系统、数据清洗系统和可用的脚本语言。我们要想写一本在几年内都是有用的书，所以请理解我们只是给出一般的指导。我们建议你搜索互联网来找出每个厂商和脚本语言的最新版本。

我们希望这些必须执行的一系列基本的底层转换过程是一个有趣的旅程。我们已经介绍了一些简单的 UNIX 部件如 ftp，sort，gawk 和 grep，不过要了解专业 ETL 工具包都有相应的数据流模式可以替代这些例子。

本章的下半部分重点介绍 DBMS 高速执行大量数据装载的特殊技术，强制参考完整性，并行处理的优势，以及如何排除性能问题。

8.1 当前市场上提供的 ETL 工具包

在第一章，我们讨论了购买厂商的 ETL 工具包和手写 ETL 系统之间的优势和劣势。从数据仓库的观点，ETL 市场有三个类别：主流 ETL 工具，数据评估和数据清洗工具。

按照字母顺序，本书提及的主流 ETL 工具包厂商，当某公司有其它类别的产品时以产品的名称：

- Ab Initio
- Ascential DataStage
- BusinessObjects Data Integrator
- Cognos DecisionStream
- Computer Associates Advantage Data Transformation
- CrossAccess eXadas
- DataHabitat ZeroCode ETL
- DataMirror Transformation Server
- Embarcadero DT/Studio
- ETI(Evolutionary Technologies International)
- Hummingbird ETL
- IBM DB2 Data Warehouse Manager
- Informatica (PowerCenter and SuperGlue)
- Information Builders iWay
- Mercator Inside Integrator (acquired by Ascential)
- Microsoft SQL Server DTS (Data Transformation Services)

- Oracle9i Waredhouse Builder
- Pervasive
- Sagent Data Flow Server (acquired by Group 1)
- SAS Enterprise ETL Server
- Sunopsis

大多数这里列出的名称是它们的所有者拥有版权的。

写本书时主要的数据评估厂商列表如下：

- Ascential (ProfileStage)
- Evoke Software
- Pervasive
- SAS
- Trillium/ Harte Hanks (with the Avelino acquisition)

写本书时主要的数据清洗厂商如下：

- Ascential (acquisition of IBM)
- First Logic
- Group 1
- Pervasive
- SAS DataFlux
- Search Software America
- Trillium (acquired Harte Hanks)

如果在互联网上搜索这些产品，你会得到它们当前的状态和大量其他的信息。

ETL 工具包一般将它们的功能打包成为一系列的转换。每个转换执行一种特定的数据处理。这些转换的输入和输出是相互兼容的，因此这些转换可以很容易

被串在一起，一般以图形界面的方式表达。转换的典型类别由十几个例子组成，每个类别包括：

- 聚合
- 通用表达式
- 过滤
- 关联
- 查找
- 规范化
- 分级
- 序列生成器
- 分类器
- 源阅读器 （适配器）
- 存储过程
- 更新
- XML 输入和输出器
- 你可以用多种语言中的多种函数写你的转换

8.2 当前脚本语言

在多平台（一般为 UNIX、Linux、Windows， IBM 主机）可用的脚本语言包括：

- JavaScript
- Perl
- PHP
- Python

- Tcl

所有这些脚本语言可以出色的读和写文本文件和调用排序例程包括本地 OS 排序包像 SyncSort 和 CoSort 这样的商业包一样。一些还有很好的类似商业 DBMS 一样的界面。

当然，我们也可以采用 C、C++或其它类似的方法。虽然所有 ETL 工作确实经常采用这些方法，但在这个比较低的级别建设整个 ETL 是非常罕见的。

8.3 时间是本质

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*

整个 ETL 系统中，时间，或更精确的，吞吐量是主要关心的内容。这种转换处理任务设计主要目的归根结底是使得数据装载到展现表中最快并使得最终用户能快速的从这些表中得到响应。有时，当清洗难处理的或脏数据时吞吐量很大。

推或拉

在每个数据仓库中，原始数据来自平面文件系统是不可避免的。合并这些数据到数据仓库的第一步是将它移动到 ETL 服务器。平面文件可以从源系统以推或拉的方式移动到到 ETL 服务器上。

假如推可以看作是源系统主动将文件推向 ETL 服务器，而拉可以看作 ETL 服务器主动从文件服务器上抽取文件。

哪一种方法更好？这个问题的真实答案是，可能两种都一样好。然而，更重要的问题是什么时候？——如“什么时候源文件可以移动了？”

在许多情形中,，需要移动的源文件来自操作型业务系统，并且通常直到操作型系统夜间批处理完成之后，才将要移动到 ETL 服务器的文件准备好。这可能会导致在文件准备好之前就开始了转换处理，这种情形装载到数据仓库的数据可能是错误的或不完整的。在这些情况下，让主系统推源文件有以下的优势：

推送源文件的 FTP 步骤可以嵌在操作型系统的批处理中，这样当主系统把文

件准备好后就可以立刻推送，因此在尽可能早的时候开始 ETL 处理并在装载期限最小化空闲时间。

准备源文件过程中的错误可以避免文件转换有一个错误的开始，可以避免错误或不完整的数据在开始阶段就装载到数据仓库中。

为了支持从主系统推送源文件，ETL 服务器必须有一个 FTP 主机服务配合运行。

在许多情形中，中断的 FTP 处理必须重新开始。下载的文件越大，批处理期限越紧，简单 FTP 的风险越高。如果这对你非常重要，你应该找一个可续传的 FTP 程序并/或验证你的 ETL 工具包是否可以续传中断传输。

这好像等同于 ETL 处理需要文件在任何 ETL 处理需要它们的时候出现。在这些情形，当需要的时候 ETL 服务器可以拉这些文件。ETL 服务器必须建立一个 FTP 连接到主文件服务器上。在一个 Unix shell 脚本或 Windows 批处理文件中运行 FTP 是很简单的。在这两个平台，文件传输命令可以通过一个外部命令文件传递给 FTP，如以下的例子。

在下面几页，我们介绍许多底层的数据操作命令，包括排序、抽取子集、调用模块装载器和建立聚合。为了清楚些，我们显示每个命令的命令行文本。显然，在商业 ETL 工具包中，所有这些命令可能通过鼠标单击激活图形界面来实现。紧记！

你可以在一个 Windows 批处理文件中嵌入以下命令：

ftp –n –v –s:transger.ftp

-n 关闭登录提示选项

-v 关闭远程消息

-s: 指定命令文件，在这里是“transger.ftp”

命令文件 transger.ftp 的内容可能如下：

open hostname

user userid password

```
cd /data/source
lcd /etl/source
ascii
get source_1.dat
get source_2.dat
get source_3.dat
… … …
get source_2.dat
bye
```

在 UNIX 系统，命令是相同的，但传输命令文件的语法可能有小的差别。

用标志确保传输

不管你是推送或拉你的平面文件数据源，你都需要确保你的文件传输是正常完成，没有错误的。处理部分传输的文件可能导致不可靠或不完整的数据被装载到数据仓库中。

确保你的传输完全的一个简单的方法是使用标志（或信号）文件。标志文件没有有意义的内容，它的存在仅仅表示它相关的文件的可读性。

- 在推送方法中，在最后一个真正的源文件推送后发送一个标志文件。当 ETL 收到这个标志文件时，它表示所有的源文件已经全部收到，并且 ETL 处理现在可以安全的使用源文件。如果源文件的传输中断了，这个标志文件不会发送，ETL 处理暂停直到错误被纠正和源文件重新发送。
- 标志文件也可以用在拉的环境中。在这种情形，源主机发送一个标志文件只是通知 ETL 服务器源文件准备好了。一旦 ETL 服务器接收到这个标志文件，它可以初始化 FTP 过程来拉源文件，开始 ETL 处理。

任何一个方法，ETL 处理都必须包含一种方法，查询本地文件系统来检查标志文件的存在。大多数专用的 ETL 工具都包含这个功能。如果你人工开发 ETL 处理，Windows NT/2000 服务器调度任务或 Unix 守护程序作业可以完成这个任务。

预装载中的数据排序

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*

某些通用 ETL 处理要求源数据按照特定次序排序来获得想要的结果。比如聚合和关联平面文件源这类的 etl 处理，数据就需要预排序。一些 ETL 工具可以在内存中处理这些任务；但是，聚合或关联非排序数据明显比同样对排序的数据处理需要更多资源和消耗时间。

当源数据在数据库中时，排序很容易通过数据库，查询数据的 SQL 的 order by 语句来实现。但是如果源数据是来自平面文件，在开始 ETL 处理之前你需要一个排序程序来将数据整理成正确的次序。

在主机系统中排序

每一种主机系统都有 IBM 的 DFSORT 或者 SyncSort 的 SORT 程序。Syncsort 和 DFSORT 命令基本相同并且相当简单。几乎很少例外，主机数据文件都以固定长度格式存储，因此大多数排序的执行只需简单指定要排序的数据元素的位置和长度。我们用下面这个销售文件例子来说明主机排序如何进行。

20000405026Discount Electronics 00014SUBWOOFER

^^^^00000150000198000002970 00

20000406005City Electronics 00008AMPLIFIER

^^^^00000350000261000009135 00

20000407029USA Audio and Video 00017KARAOKE MACHINE

^^^^00000200000088200001764 00

20000410010Computer Audio and Video 00017KARAOKE MACHINE

^^^^00000100000088200000882 00

20000411002Computer Audio and Video 00017KARAOKE MACHINE

^^^^00000350000088200003087 00

20000411011Computer Audio and Video 00008AMPLIFIER

^^^^0000010000026100000261000

20000415019Computer Discount 00018CASSETTE PLAYER/RECORDER

^^^^0000020000006840000136800

20000418013Wolfe''s Discount 00014SUBWOOFER

^^^^0000025000019800000495000

20000418022USA Audio and Video 00008AMPLIFIER

^^^^0000015000026100000391500

20000419010Computer Audio and Video 00023MP3 PLAYER

^^^^0000010000017640000176400

20000419014Edgewood Audio and Video 00006CD/DVD PLAYER

^^^^0000020000044100000882000

20000419016Computer Audio and Video 00014SUBWOOFER

^^^^0000030000019800000594000

20000419021Computer Audio and Video 00014SUBWOOFER

^^^^0000035000019800000693000

20000419028Bayshore Electronics 00020CD WALKMAN

^^^^0000015000004140000062100

这个文件的 COBOL copybook 应该是这样：

01 SALES-RECORD.

05 SALE-DATE PIC 9(8).

05 CUSTOMER-ID PIC X(3).

05 CUSTOMER-NAME PIC X(27).

05 PRODUCT-ID PIC X(5).

05 PRODUCT-NAME PIC X(28).

05 UNIT-COST PIC 9(4)V99 COMP-3.

(this takes up only 4 physical bytes)

05 UNITS PIC 9(7).

05 UNIT-PRICE PIC 9(7)V99.

05 SALE-AMOUNT PIC 9(7)V99.

SORT 命令的基本结构是：

SORT FIELDS=(st, len, dt, ad)

这里 st 表示开始位置，len 表示长度，dt 表示数据类型，ad 表示排序次序（正序或反序）。因此，按照 customer-id 对我们的销售文件排序的代码如下：

SORT FIELDS=(9,3,BI,A)

意思是，按照 9 到 11 位以正序排序，以二进制方式处理数据。如果要执行多字段排序，简单的提供每个额外排序字段的 st，len，dt，ad 参数就可以了。

比如，假设你的 ETL 任务是按照年，产品和客户汇总这些销售数据。源数据是每天的销售，它的自然次序也是按天排序。根据自然次序汇总数据是相当复杂的任务，需要创建、管理和定位内存的数组变量来保存汇总，直到最后一个输入记录被处理并且再次定位内存数组来装载到数据仓库。但是对按照聚合键（年＋产品 id＋客户 id）预排序的源数据，ETL 工具汇总就相当简单。按照聚合键排序这些数据的命令如下：

SORT FIELDS= (1,4,BI,A,39,5,BI,A,9,3,BI,A)

一旦按照这种方式排序，ETL 处理汇总数据就变得非常有效了。当读取源记录时，键字段年的值、产品 id 和客户 id 与前面的记录的键值比较，这由内存变量保存。只要键值相同，单位和销售量就加到累计的内存变量中。当键值变化了，前一个键的汇总值就从内存变量装载到数据仓库中，接着这个内存变量被重新设定新键来累加新的汇总值。

正如前面的章节讨论的，主机数据常常有某些针对主机的特殊问题。SORT

有丰富的数据类型设置来帮助你应对这些挑战。虽然在许多情况下使用 BI（二进制）作数据类型，但是还有一系列可选的数据类型来处理特殊的情况，如以下的表 7.1 所列。

表 7.1 可选的主机数据类型

数据类型	用法
PD	用作专门的数字值排序，按照压缩十进制（或 COMP-3）格式排序。
ZD	用作专门的数字值排序，按照分区十进制格式排序。
AC	用作专门的 ASCII 代码相关的数据排序，而不是按照主机本地的 EBCDIC 代码。当数据是从主机传输到 Unix 或 Windows 上的 ETL 过程时，用这种格式为混合的（字母数字混合编码的）字段排序。
Dates	你可能会遇到带 2000 前格式（没有明确的世纪）的日期在遗留系统文件中你可能会在历史系统文件中遇到公元 2000 年以前的日期时（那种不带明确世纪，例，98），SORT 有丰富的数据类型设置来处理这些日期并赋给它们相应的世纪。

这个表只提供了可用数据类型的很少一部分。还有许多其它可用的数据类型可以针对你可能遇到的数字集和其它数据格式的数据类型。

你也可以在组合索引中混合数据格式。比如，按照年和反序单位成本来排序销售文件使用以下命令：

```
SORT FIELDS= ( 1,4,BI,A,72,4,PD,D)
```

在 Unix 和 Windows 系统中排序

在 Unix 和 Windows 系统中的平面文件是基于 ASCII 字符的，没有难处理的陈旧的数据格式（压缩数字，等等）混合编制在旧主机系统上来节省磁盘空间。但是这些系统有它们自己的挑战。

你要面对的最常见的排序挑战是分隔符文件或其它非结构化数据文件。在这里，非结构化指的是每个记录没有按照等长的列整理的数据。同样的，你不能指定排序键位置。

相反，sort 实用程序必须能够使用分隔符来分析记录。（当然，主机实用程序 SyncSort 和 CoSort 在 Unix 和 Windows 平台也可以使用。）以下的抽取显示了本章前面的同一个销售数据现在用逗号分隔符格式化的文件。

04/05/2000,026,Discount Electronics,

00014,SUBWOOFER,124.74,15,198.00,2970.00

04/06/2000,005,City Electronics,00008,AMPLIFIER,164.43,35,261.00,9135.00

04/07/2000,029,USA Audio and Video,00017,KARAOKE MACHINE,

55.57,20,88.20,1764.00

04/10/2000,010,Computer Audio and Video,00017,KARAOKE MACHINE,

55.57,10,88.20,882.00,

04/11/2000,002,Computer Audio and Video,00017,KARAOKE MACHINE,

55.57,35,88.20,3087.00,

04/11/2000,011,Computer Audio and

Video,00008,AMPLIFIER,164.43,10,261.00,2610.00

04/15/2000,019,Computer Discount,00018,CASSETTE

PLAYER/RECORDER,43.09,20,68.40,1368.00

04/18/2000,013,Wolfe''s Discount,

00014,SUBWOOFER,124.74,25,198.00,4950.00

04/18/2000,022,USA Audio and

Video,00008,AMPLIFIER,164.43,15,261.00,3915.00

04/19/2000,010,Computer Audio and Video,00023,MP3

PLAYER,111.13,10,176.40,1764.00

04/19/2000,014,Edgewood Audio and Video,00006,CD/DVD

PLAYER,277.83,20,441.00,8820.00

04/19/2000,016,Computer Audio and

Video,00014,SUBWOOFER,124.74,30,198.00,5940.00

04/19/2000,021,Computer Audio and

Video,00014,SUBWOOFER,124.74,35,198.00,6930.00

04/19/2000,028,Bayshore Electronics,00020,CD WALKMAN,

16.08,15,21.40,621.00

Unix sort 命令的基本语法如下：

sort +start_field_number -stop_field_number file

字段编码从 0 开始，因此在这个销售文件，字段的编码如下：

(0) SALE-DATE

(1) CUSTOMER-ID

(2) CUSTOMER-NAME

(3) PRODUCT-ID

(4) PRODUCT-NAME

(5) UNIT-COST

(6) UNITS

(7) UNIT-PRICE

(8) SALE-AMOUNT

sort 程序的默认分隔符是空格。--t 选项允许你指定一个替代的分隔符。重复第一个例子，按照客户 id 排序，sort 命令应该如下：

sort –t , +1 -2 sales.txt > sorted _sales.txt

意思是，从列 1（客户 id）开始排序，在列 2（客户名称）停止排序。排序输出目前为标准输出（终端），因此你重定向输出到一个新的文件：sorted_sales.txt。

如果没有指定停止列，在一个包含每个列的组合键排序从指定的开始列的开始排序。

看如何实现本章前面的聚合键排序（年＋产品 id＋客户 id）来为一个聚合 ETL 处理准备数据。年是字段 0 的最后部分，产品 id 是字段 3，而客户 id 是字段 1。

主要挑战是限制排序只对日期的年部分使用。如下：

sort -t, +0.6 -1 +3 -4 +1 -2 sales.txt > sorted_sales.txt

对日期（字段 0）的年的部分排序，在这个字段中紧跟一个句点来指定开始的字节。和字段编号一样，字节编码从 0 开始，因此＋0.6 意思是从日期的第七个字节开始排序。

同样的，这些 Unix 排序例子使用字母排序方法。然而，字母排序并不适合用在定量的数字字段。比如，对销售文件的单位成本按字母顺序排序将得到错误的结果－－成本是 16.08 的 CD WALKMAN 将排在成本为 124.74 的 SUBWOOFER 后面。要解决这个问题，你必须指定单位成本字段是数字，如下：

sort -t, +5n -6 sales.txt > sorted_sales.txt

要改变排序的次序，从正序到反序，用—r（反向）选项。比如，销售数据按照年和单位成本的反序排序如下：

sort -t, +0.6r -1 +5n -6 sales.txt > sorted_sales.txt

表 7.2 列出了其它有用的排序选项。

表 7.2 UNIX 排序命令的开关

数据类型	用法
-f	在字母排序字段中忽略
-b	忽略前导空格
-d	忽略标点符号
-i	忽略不可打印字符
-M	按照 3 个字母的月份缩写排序（比如，JAN 在 FEB 前面，依此类推）

Unix 实用程序命令的丰富的设置，包括 sort，grep 和 gawk 到没有什么名的关键程序，已经被传送到 Windows 操作系统并以免费软件的方式提供。有许多网站你可以获得这些实用程序。我们发现一个相当全面的集在一个压缩文件在 http://unxutils.sourceforge.net/上。

切割多余部分（过滤）

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*

源文件常常包含与数据仓库不相关的数据。在某些情形下，只有源文件中的很少一部分纪录是数据仓库需要的。还有一些情况，一个条长记录只有很少数据元素是需要的。在处理中尽可能早的排除不需要的数据是一种可以加速 ETL 处理的一种明确的方法。在源系统建立抽取文件可以获得最大的性能收益，这是因为，这样做除了可以改善 ETL 处理本身之外，还可以等比例减少文件传输时间。无论你是通过摘取－文件的部分记录还是每个记录的部分字段来压缩源文件，在 ETL 处理中，都节省了数据传输到 ETL 服务器的时间，也节省了 I/O 时间，让内存处理更小的文件。

最容易建立的抽取文件是那些只需要源文件的部分记录的。这种抽取类型一般可以用实用程序创建，一般也是系统上最有效的运行程序。

以下部分讨论在主机系统和 Windows 和 Unix 系统上创建抽取。

在主机系统上抽取源文件的部分记录

在主机系统，SORT 实用程序可能是除了写 COBOL 或第四代语言（SAS, FOCUS, 等等）程序之外的最快和最容易的创建抽取文件的方法了。

最简单的情形是建立一个只需要子集的抽取。SORT 允许你指定源记录来包含或不包含到抽取文件中。

```
INCLUDE COND=(st,len,test,dt,val)

OMIT COND=(st,len,test,dt,val)
```

这里 st 表示输入字段的开始位置，len 是它的长度，test 是布尔测试来执行，dt 是输入字段的数据类型，val 是进行比较的值。比如这个例子，我们用本章前面排序例子的销售文件作为样本。以下是如何抽取只包含从 2000 年及以后的销售记录。

```
SORT FIELDS=COPY

INCLUDE COND=(1,4,GE,CH,C'2000')
```

这也可以用一个不包含语句来写：

```
SORT FIELDS=COPY
OMIT COND=(1,4,LT,CH,C'2000')
```

组合条件通过以 And 或 OR 语句联合条件集来建立。要选择只有 2000 年及以后和客户 id 超过 010 的记录，代码如下：

```
SORT FIELDS=COPY
INCLUDE COND=(1,4,GE,CH,C'2000', AND,9,3,GT,CH,C'010')
```

更加复杂的条件可以用 AND，OR 和圆括号的组合控制执行的顺序来建立。你甚至可以搜索包含某个值的字段。比如，选择名字包含 discount 的客户，包含语句的代码如下：

```
SORT FIELDS=COPY
INCLUDE COND=(12,27,EQ,CH,C'Discount')
```

因为指定的 27 字节的输入字段比常量 discount 长，SORT 搜索整个输入字段寻找这个常量。（这等于 SQL where 语句中的 LIKE ‘%Discount%’）

抽取源文件字段的子集

建立一个只包含数据仓库 ETL 处理所必需字段的抽取文件对 ETL 源文件的大小有巨大影响。事实上很少见到这样的源文件——包含大量的数据元素，但只有很少的部分是 ETL 处理所需的。抽取只包含所需字段对文件大小有很大的影响即使源文件记录很少。考虑到一些源文件有上百万记录，只抽取所需字段可以减少成十或成百兆数据传输到 ETL 服务器处理。

使用 OUTFIL OUTREC 语句，SORT 也可以处理选择一个字段子集到一个源文件中来压缩所有必须传输到 ETL 服务器的数据的任务。我们原来的例子中的销售文件的记录长度达 100 字节。假设你的 ETL 处理只需要销售日期、客户 id、产品 id、单位价格、数量和销售量字段，这加起来总共 36 字节。只包含这些字段的抽取大约只有整个源文件大小的 1/3。要压缩更多，你可以只选择 2000 年之后的记录。

```
SORT FIELDS=COPY
INCLUDE COND=(1,4,CH,GE,C'2000')
OUTFIL OUTREC=(1,8,9,3,39,5,72,4,76,7,92,9)
```

在这个最简单的例子中，OUTREC 语句由简单的字段的开始位置和长度组成并复制到抽取文件中。然而，仍然还有一些难处理的主机数据。单位成本、单位和销量仍然以主机的格式存储。当传输到 ETL 服务器时，这些以本地主机格式存储的字段无法直接使用。要使得它们在 UNIX 或者 Windows 的 ETL 服务器上可用，必须要按照显示格式对这些数字字段进行重新格式化。

```
SORT FIELDS=COPY
INCLUDE COND=(1,4,CH,GE,C'2000')
OUTFIL OUTREC=(1,8,9,3,39,5,
72,4,PD,EDIT=IT.TT,LENGTH=7,
76,7,ZD,EDIT=IT,LENGTH=7,
92,9,ZD,EDIT=IT.TT,LENGTH=10)
```

在这种格式中，单位成本在源文件中是按照压缩数字格式存储，现在解压缩成一个 7 字节的显示字段，形式如 9999.99。同样的，单位和销量重新格式化为显示格式如 9999999 和 9999999.99。

显然，主机的 SORT 实用程序是你处理主机数据的一个好帮手。在前面的例子中引用的技术仅仅是它的丰富功能的一个子集。精通 SORT 的用法会让你在处理主机数据的时候很得心应手。

在 Unix 和 Windows 系统中抽取源文件的子集

现在，让我们来看看如何在 Unix 和 Windows 系统中完成同样的抽取。我们会再一次用到 Unix 的实用程序 gawk，在 Windows 上也可以使用 Gawk 编程语言 awk 的 GNU 版本。Awk 的基本功能是搜索包含某个样本的文件中的行（或其它文本单元）。

使用 gawk 的语法如下：

gawk -fcmdfile infile > outfile

选项--f 指定包含要执行的 gawk 命令的文件。Infile 指定输入源文件，而 outfile 指定 gawk 输入指向的文件。

第一个抽取任务是仅仅选择 2000 年之后的销售数据。Gawk 命令应该是这样：

gawk -fextract.gawk sales.txt > sales_extract.txt

extract.gawk 命令文件包含如下内容：

```
BEGIN {
FS=",";
OFS=","}
substr($1,7,4) >= 2000 {print $0}
```

BEGIN {. . }部分包含在第一个源记录处理前要执行的命令。在这个例子，FS=”,” 和 OFS=”,”规定输入和输出文件中字段的分隔符是逗号。如果不指定，默认的分隔符是空格。

抽取逻辑写在语句 substr($1,7,4)>=2000 中。意思是选择从第一（$1）开始的十个字节中的第七个开始的部分大于等于 2000 的记录。

{print $0}语句意思是输出所选择的记录的整个源记录（$0）。

组合条件通过连接符号&&(和)或者||(或)的连接来创建。仅仅选择从 2000 年和客户编码超过 100 的记录，用以下语句：

```
BEGIN {
FS=",";
OFS=","}
substr($1,7,4) >= 2000 && $1 > "010" {print $0}
```

并找出包含折扣的客户姓名记录：

```
BEGIN {
FS=",";
```

```
OFS=","}
$3~ /Discount/ {print $0}
```

正如你所看到的，gawk 命令包含许多知识，你可以非常容易建立节省时间和空间的抽取。

抽取源文件字段的子集

你也可以使用 gawk 来创建字段抽取。同上，假设你想要创建一个只包含销售日期、客户编号、产品编号、单位成本、单位和销量字段的抽取。同时，也只要抽取 2000 年之后的记录。代码如下：

```
BEGIN {
FS=",";
OFS=","}
substr($1,7,4) >= 2000 {print $1,$2,$4,$6,$7,$9}
```

这个 print 语句指定包含在输出中的字段。

请注意，在 gawk 中，字段编号从$1 开始，因为$0 指的是整个输入记录。这与 sort 命令是不同的，那里$0 代表第一个字段。

现在假设你要创建一个固定字段长度而不是分隔符文件的抽取文件。好，gawk 同样可以很好的处理。如下是和前面一样的例子抽取到一个固定长度文件：

```
BEGIN {
FS=",";
OFS=","}
{substr($1,7,4) >= 2000
{printf "%-11s%-4s%-6s%07.2f%08d%010.2f\ n", $1,$2,$4,$6,$7,$9}}
```

在这里，printf 命令（格式化打印）代替了一般的 print 命令。Printf 后跟格式化字符串。%符号代表格式开始，并且格式的数字必须和输出字段的数量是一致的。格式通常有如下用法：

- %ns：对文本字符串，n 是最小输出长度
- %0nd：对十进制数字，n 是最小输出长度
- %0n.mf：对浮点数字，n 是总的数字位数，m 是小数位数

默认输出方式是右对齐的。要数据左对齐（如你想要大多数文本字段显示的），如前面的例子在字段长度前加一个短划线。要在数字格式左补 0，长度前加 0（如，%07.2f）。新行标志\n 在格式字符串的末尾告诉 gawk 将每个输出记录写到新的一行。

在主机系统创建聚合抽取

假设你要按照月、客户和产品合计单位和销量来汇总销售文件。以下是主机 SORT 命令的实现：

```
INREC FIELDS=(1,4,5,2,9,3,39,5,3Z,76,7,3Z,91,9)

SORT FIELDS=(1,14,CH,A)

SUM FIELDS=(15,10,ZD,25,12,ZD)
```

这里有两个新命令——INREC 和 SUM 。INREC 的作用如同 OUTREC，不同的是它在排序操作处理前操作输入记录。这个例子中，输入记录重新格式化只包括汇总所需要的字段——年、月、客户编号、单位和销量。注意到有两个 3Z。这是因为单位和销量左补 0 以避免 SUM 操作时发生运算溢出，因此要为这些字段的长度每个增加 3 个字节。

接着，SORT 命令指定用来排序的字段。SORT FIELDS 作为 SUM 操作的关键字，一般像一个 SQL 的 GROUP BY 语句。但是注意到 SORT FIELDS 用的是格式化了的记录布局——因此 SORT FIELDS 可以简单的用位置 1 到 14 来定义，现在这里包含年、月、客户编号和产品编号。

最后，SUM 命令指定要求和（或聚合）的数值字段。已格式化字段位置（在本例中为长度）再次用到。因为单位在源文件中占据 76－82 的位置（总共 7 个字节），在格式化了的记录，单位占据 15－25 的位置（10 字节）。

使用这种技术，输出文件只有 36 字节宽（而初始源记录的宽是 100 字节）

并且在文件中只包含一条每个年、月、客户编号和产品编号组合的记录。网络和 ETL 服务器将非常欢迎使用这种压缩源文件的方式。

注意，交易系统一般配置最小的可用临时空间。这可能会影响你做在抽取过程中在源中压缩数据的决定。见本章这个主题后边，在“使用聚合和 GROUP BY”的讨论。

在 UNIX 和 Windows 系统创建聚合抽取

在 UNIX/Windows 上完成同样的聚合更复杂些，但是也不是非常复杂。你需要同时用到 sort 和 gawk 实用程序。Sort 输出到 gawk 命令使用管道字符|来处理。命令如下：

```
sort -t, +0.6 -1 +0.0 -0.2 +1 -2 +3 -4 | gawk -fagg.gawk > agg.txt
```

首先，回顾排序。

```
+0.6 -1 = year

+0.0 -0.2 = month

+1 -2 = customer-id

+3 -4 = product-id
```

agg.gawk 命令文件如下。我们已经加入了注释（前导＃）来解释它如何起作用：

```
# set delimiters

BEGIN {

FS=",";

OFS=","}

#initialize variables for each record

{inrec += 1}

{next_year=substr($1,7,4) substr($1,1,2)}

{next_cust=$2}
```

```
{next_product=$4}
#after a new year and month record
#write out the accumulated total_units and total_sales for the
prior year
inrec > 1 && ( \
next_year != prev_year \
|| next_cust != prev_cust \
|| next_product != prev_product ) \
{print prev_year,prev_cust,prev_product,total_units,total_sales}
#accumulate the total_sales sales and count of records
{total_units += $7}
{total_sales += $9}
#if the year changed reinitialize the aggregates
next_year != prev_year {
total_units = $7;
total_sales = $9}
#store the year (key) of the record just processed
{prev_year = next_year}
{prev_cust = next_cust}
{prev_product = next_product}
#after the last record, print the aggregate for the last year
END {print prev_year,prev_cust,prev_product,total_units,total_sales}
```

它比主机排序复杂一些。它只是在处理记录时简单的保持跟踪关键字值并当关键字改变时输出记录。注意 END 命令。这个命令确认最后的聚合记录是否输

出。一旦你熟悉这些操作并学会控制流工作的原理，你可以很快聚合它们。

8.4 使用数据库的块加载工具加速数据插入

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*

用排序，抽取和聚合等方法把你的源数据尽可能快地加载到 ETL 服务器以后，你将面临把大量必要的数据加载到数据仓库，这个时候用到了数据库管理系统中的块加载功能。在数据库交互方面块加载器比普通 SQL 语句更高效，它将大大提升 ETL 的性能。我们使用 ORACLE 的 SQL*LOADER 来说明块加载器的优越性。同样你也可以在其它大多数数据库管理系统中找到块加载功能。

重要提示：大多数块加载器在向数据库插入数据时都有一定的限制。例如，他们在插入大批量的数据时确实获得很大的好处，但是如果在你的处理中包含了对现存记录的更新，可能你就没有那么幸运了。根据你需要插入和更新的记录数量，你将发现仔细地预处理你的输入数据后－将插入与更新分开数据可以以块加载的方式运行，这样至少插入的数据可以以块加载方式运行。注意 IBM 的 Red Brick 系统可以支持更新或者插入逻辑作为它的块加载器的一部分。

基于传统路径查询方法，SQL*LADER 使用 INSERT 语句来加载数据到表中，数据库的操作与常规 SQL 处理中的插入语句以同样的方式来处理。所有的索引都要维护，主键、参考完整性和其他大部分的约束都起作用，并启动插入触发器。这时候使用 SQL*LOADER 的主要好处是：方法简单和使用最少的代码来从平面文件加载数据。

激活 SQL*LOADER 的语法有多种格式，下面我们将用 SQL*LOADER 加载数据到 ORACLE 表中，例子还是用前面用过的销售文件做示范。

```
sqlldr userid=joe/etl control=sales.ctl data=sales.txt log=sales.log

bad=sales.bad rows=1000
```

控制文件 Sales.ctl 应该包含以下内容：

```
LOAD DATA
APPEND INTO TABLE SALES
FIELDS TERMINATED BY "," OPTIONALLY ENCLOSED BY '"'
(SALE_DATE DATE(20) "MM/DD/YYYY",
CUSTOMER_ID,
CUSTOMER_NAME,
PRODUCT_ID,
PRODUCT_NAME,
UNIT_COST,
UNITS,
UNIT_PRICE,
SALE_AMOUNT)
```

当然，目标表 SALES 应该已经存在 ORACLE 数据库。这个简单的例子用传统 SQL 的 INSERT 语句加载数据到数据库中。接着我们来寻找提升数据加载性能的方法。

对于 SQL*LOADER 来说，性能增强模式是直接模式。将先前的加载改变为用直接模式，只需简单的在 sqlldr 命令中加入 direct=true 参数。显示如下;

```
sqlldr userid=joe/etl control=sales.ctl data=sales.txt log=sales.log
bad=sales.bad rows=1000 direct=true
```

下面我们将讨论直接模式是如何提升性能的：

1. SQL*LOADER 对表使用一个排它锁，防止所有其它操作。

2. 在直接模式中不能强制使用数据库约束（主键和唯一键约束，外键约束等等）。如果发生冲突，相关的索引将处于不稳定状态，在重建索引之前需用手动清除。

3. 外键约束在直接加载过程不可用，在加载后才可重新启用。为保证兼容性，

不仅新增记录，所有的行都要根据约束进行检查。

4. 在直接模式中，插入触发器在行插入时不可以被启动。所以有必要的话，独立开发一个进程来执行通常由触发器处理的操作。

所以直接加载的高效率并不是免费的。我们必须非常小心的使用直接加载，因为如果你遇到脏数据，这些脏数据可能阻止你在装载完成后重建主外键约束。但是如果你有强壮的处理方法来确保被加载的数据是干净的话，直接加载大块源文件是最好的方法。

无论你使用传统的方法还是直接路径查询模式方法，SQL*LOADER 有比前面简单例子更丰富的功能。以下列出一些关键功能：

- 处理固定宽度，分隔符分割的和多行输入
- 从多个文件中接受输入
- 加载到多个目标表
- 加载到分区表
- 有效的管理和更新索引

其他方面，编程功能允许根据条件处理、值分配等等。然而，应该避免使用这些功能，因为他们通常逐条处理每一条输入记录，这将大大降低块加载的性能。毕竟，选择块加载器的首要因素是性能。

块加载的准备

今天市场上的大多数 ETL 工具都可以直接抽取数据，并通过数据库块加载器来加载到数据库表。但是不是所有的工具都用同样的方法来使用块加载器。其中的一些工具比较其它工具性能更高，还有一些工具需要外部插件来兼容块加载器。无论你如何传输数据，对一个 ETL 开发者来说， 理解如何为块加载器的处理准备数据是非常重要的。

把数据导入数据仓库，块加载是一个非常有效方法。块加载器是一个数据库之外的实用程序，它存在的唯一目的就是快速的把大量数据导入数据库。每种数据库管理系统都有不同的、专用的块加载程序。最常见的块加载程序列在表 7.3

中。

一般来说，这些块加载的作用是相同的。为了说明块加载器的功能，我们以 Oracle 的 SQL*Loader 为例子；在写这本书的时候，我们认为在数据仓库领域它是块加载器的公认命名者。

表 7.3 块加载器

数据库管理系统	块加载器名称	注释
Oracle	SQL*Loader	需要一个控制文件来描述数据文件的编排。优化性能的两个重要参数： DIRECT={TRUE | FALSE} PARALLEL={TRUE | FALSE}
Microsoft SQL Server	Bulk Copy Program（BCP）	微软同样提供 BULK INSERT，它比 BCP 更快。 它将节省大量的时间，因为它不需要利用 Microsoft Netlib API。
IBM DB2	DB2 Load Utility	DB2 接受来自 ORACLE 的控制文件和数据文件作为输入源。
Sybase	Bulk Copy Program（BCP）	同样支持 DBLOAD，通过使用参数 BULKCOPY=“Y”。

一旦你理解块加载的一般概念，加载器之间的相似性使得掌握每个特定的实用程序是一件简单的事。

块加载器一般需要两个文件来执行：

- 数据文件。将要加载到数据仓库，包含实际数据的数据文件。数据可能是各种不同的文件格式和布局，包括各种各样的分隔符。所有这些参数都在控制文件中定义。
- 控制文件。控制文件包含数据文件的元数据。各种不同参数的列表是可扩展的。下面是 SQL*Loader 的 控制文件基本元素的列表：

  1. 源文件的位置
  2. 列和字段布局的说明
  3. 数据类型的说明

4. 从源到目的的数据影射
5. 源数据约束
6. 缺失数据的默认说明
7. 去除空格和 tab 键的方法说明
8. 相关文件的名称和位置（例如，事件日志，拒绝和丢弃记录的文件）

关于 SQL*Loader 命令语法和使用的全面指南，请参考：Oracle SQL*Loader：The Definitive Guide, by Jonathan Gennick and Sanjay Mishra (O’Reilly & Associates, 2001).

尽管在使用块加载加载数据到数据仓库之前，你必须花费更多的 I/O 性能来将数据写到物理文件中，但是它比数据直接访问数据库并用 SQL INSERT 语句来加载快得多。

许多 ETL 工具可以使用块加载工具来将数据直接以管道方式输送到数据库中，在数据到达它的最终目的地——数据仓库事实表之前不在磁盘上停留。其他的可以在文件系统上创建控制文件和数据文件。从那里，你需要写命令行脚本来调用块加载器将数据加载到目标数据仓库。

购买 ETL 工具的主要目的是最小化程序的手写代码，无论是抽取，转换或者装载数据。但是市场上没有一个工具能完全地解决每一种技术挑战。你将发现通过块加载器或者第三方的装载工具来达到无缝的流水线一样的数据传输是一个很大的挑战。在你决定不用块加载之前，最好用工具插件和其他应用扩展如命名管道来做实验，并排除所有可能的选择。

如果你还没有购买 ETL 工具，请在 POC 中测试你的 DBMS 的加载器与潜在的产品是否兼容。如果你已经有 ETL 工具但不能为块装载准备数据，不要把它立刻放弃。你需要手工准备数据。配置你的 ETL 工具来把你的数据输出为一个以逗号作为分隔符的平面文件。接着，基于这个输出文件的规格和需要的装载参数创建一个控制文件。只有当源或目的的物理属性被改变时，如插入新列或者改变数据类型，控制文件才需要修改。

8.5 管理数据库特性来提高性能

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*

正如我们所知道的，数据库中的数据比数据表中的数据要多。强大的特性如索引、视图、触发器、主外键约束和列约束等将数据库管理系统区别于平面文件系统。当数据库增长时管理这些特性将消耗大量的系统资源，结果可能大大的降低了 ETL 装载过程的性能。

根据这样的思路，首先要做的事情是回顾数据库的设计并去除任何不必要的索引、约束和触发器。接着考虑以下提高装载性能的选项：

1. 加载数据之前去除外键（参考完整性）约束。有外键约束时，每一行加载到数据库系统的外键列的数据都要与其浮标的主键值进行比较。通过去除多外键约束的事实表中的外键约束可以大大提高性能。

虽然如此，记住，当你加载数据之后启用外键约束时数据库校验表中的每一行（不仅仅新增的行）。确保你的外键列是索引的保证重新启用约束时它本身不会成为瓶颈。

2. 保持数据库统计表详细到天。数据库统计表由数据库管理系统管理，它跟踪所有表的大小，索引中唯一值的大小和数量以及其他关于数据如何有效存储在数据库中的事实。当一个 SQL SELECT 语句提交到数据库管理系统，它使用这些统计表来判断提供请求的数据的最快的访问路径。最理想的，你应该在每次加载之后更新统计表。然而，如果你的加载处理是经常性的（每天）并且每天数据库大小的变化比利时非常小的，每周或每月更新统计表应该能满足保持比较高的性能。分区存储大表能提高更新统计表的速度，因为统计表不必刷新所有静态（或接近静态的）分区而仅仅是当前的分区。

3. 在数据库中重组碎片数据。当经常性的进行行更新和/或删除会使得表变成碎片，从而导致响应时间变慢。

当处理大的事实表时，最小化发生类似碎片的一种方法是创建分区表。分区

表一般按照时间区间来组织（比如，一个销售表按照每年进行分区）。当一年结束，这个分区包含这一年的销售数据可能就保持稳定因此不再容易产生碎片了。ETL 工具能够自动基于 DBMS 字典指定的分区方案来平滑的加载数据。

虽然如此，本年的数据还是经常被删除和重新加载，因此当前的分区成了碎片。重组一个碎片的表就是重写表中的数据到连续存储块并去除由于行更新和删除产生的死空间。如果你的加载处理执行更新和删除在大（事实）表上，考虑每个月或更频繁如果允许的重整该表。如果每次加载都产生明显的碎片，这样的重整可以设置为每次数据加载完成后进行。

此外，分区可以减少重整表花费的时间。那些旧的、静态的分区几乎很少需要重整。只有当前的分区需要重整。

次序

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*

当你加载一个数据仓库时，一个批处理中作业的排列次序是至关重要的，主要原因是 ETL 需要在数据仓库中强制参考完整性。参考完整性（RI）意味着每一个外键必须存在一个主键。因此，每一个外键，作为参考完整性关系中的子，必须有一个父主键。没有相对应的父的外键被称为孤儿。ETL 处理要避免在数据仓库中创建孤儿。

在交易系统中，参考完整性一般在数据库管理系统中执行。数据库级 RI 施行在一个交易环境中是必须的，因为人们输入的数据是每次一行-很容易出错。错误或者违反 RI 的操作会导致数据不可用，对业务没有任何用处。用户发现在数据输入过程中有数种方式导致无意的数据不可用。一旦数据被破坏，就失去了任何价值，因此这样的代价是必须的。

执行参考完整性

与易受不稳定的数据输入行为攻击的交易系统不同，数据仓库的数据是通过一个可控的过程-ETL 系统以块方式加载的。ETL 过程在进行每个实际的生产数据加载之前是经过测试和验证的。数据进入数据仓库的入口是可控的和可管理的环

境。数据库级的 RI 约束一般在数据仓库中不起作用，因为数据仓库依赖 ETL 来保证其完整性。

RI 在 DBMS 中一般不用的另外一个原因是这样会在数据库级最小化管理费用来增加加载性能。当数据库中的 RI 启用时，每行数据在插入之前都要测试是否符合 RI—也就是每一个外键都有一个父在他所要参考的表中。

在数据仓库环境中的 RI 远远比交易系统中的简单。在交易系统中，每个表可能实际上都与其它表相关，引起相关表的复杂网。在维度的数据仓库中，规则很简单：

- 事实表中的每一个外键都必须在维度表中有一个相关的主键。
- 维度表中每个主键不必在事实表有一个相关的外键。

那样的关联在规范化模式中通常称之为零对多关系。如果你已经有了维度数据仓库的指南，或者已经读过一些工具箱这样的书籍，你会明白并不是所有的维度模型都这样简单。实际上，事实可能与一个维度中的多个记录相关（以一个桥连接表，在第 5 和第 6 章中说明）因此维度可能是雪花模型。另外对事实和维度，ETL 必须以支架和层次表来处理。ETL 团队必须了解在维度模型中的每个类型的表对于有效的加载数据仓库的目的和功能。回顾第二章找到更多一个维度模型中的不同类型的表的信息。

以下的列表提供一个对一个假设的数据集市的加载过程的顺序位置的指导。

- 子维度(支架)
- 维度
- 桥连接表
- 事实表
- 层次映射
- 聚合(缩小的)维度
- 聚合事实表

子维度

一个子维度，正如在第五章所说的，是直接将一个维度附加到另外一个维度上，这时候的设计叫做许可雪花模型（permissibly snowflaked）。一个子维度也可能扮演一个主维度的角色。日历日期维度就是一个很好的例子，它常常是一个主维度同时是一个子维度。

子维度一般第一个加载到数据仓库中，因为依赖关系链始于最远的表，也就是子维度。事实表依赖于维度表，而维度依赖于子维度。因此，在结构中其他下游的表被装载之前，子维度必须先加载并且定义好它们的关键字。注意：这取决于你的业务需求和你的具体环境，因为有可能有些子维度极少用到并且不是紧急任务的需要。这就意味着如果因为失败导致子维度加载没有成功，继续进行与之相关的维度的加载过程也是可以接受的。

维度

一旦子维度加载了，你就可以加载维度了。维度需要用子维度中的代理键来做对照，因此它可以在加载过程中插入到维度中。当然，没有子维度的维度可以在开始就加载，不需要等任何的子维度完成加载。

的维度没有从属关系，可以同时加载和利用并行处理。大的维度也可以用这样的方式，但是在你执行这样的策略之前要测试它们的性能是达到最佳的效果的。有时，独立的加载大维度能通过减少连接资源来提高性能。不幸的是，在这些案例中考验和错误是最好的行为准则。

维度加载必须在继续处理之前完全成功。如果维度加载失败，调度程序必须在那个失败点上停止加载过程来阻止余下的作业的加载。如果在没有维度信息支持下继续进行处理，这个数据仓库将是不完整的，也被认为是已经破坏的和不可靠的。强制作业之间的从属关系对数据仓库维持可信度是至关重要的。

桥连接表

当一个事实表的记录对应多个维度表记录时，桥连接表位于维度表和事实表之间。桥连接表也用在维度表和多个子维度表关系中。比如，当一个事实表是医疗帐单数据库中病人治疗记录粒度上的数据时，同时处理多个病人的诊断的有效性，这时就需要一个桥连接表。在病人诊断维度加载之后，扫描治疗事务表来判断哪些诊断是一起处理的。于是，桥连接表以一个代理键来聚合诊断记录加载到

分组中。

不是所有的数据集市都包含桥连接表，但当发生时，这些表必须在维度表加载完成之后事实表加载开始之前立即加载。如果一个事实表依赖于一个桥连接表，这个桥连接表必须完全加载成功之后才能执行这个事实表的加载。如果你试图在这个桥连接表部分加载就开始加载事实表，事实表中的分组就被漏掉了，因此事实表中的数据与桥连接表进行关联的时候就会被禁止。

交叉-参考：桥连接表加载的信息可以在第 5 章找到；在加载事实表时使用桥连接表来发现分组在第 6 章。

事实表

在维度数据模型中事实表实际上依赖于所有其他表因此一般在最后加载。一旦子维度、维度和桥连接表加载完成，事实表需要的所有参照表都有了可以准备加载。记住，这里的 RI 是强制的，因此你必须确保事实表中的每一个外键都有来自他相关维度或桥连接表的相应的主键。

事实表一般在数据仓库加载中是各种表类型中最耗时的；你应该在所有与这个事实表相关的表都已经加载后才开始加载事实表。不要等到数据仓库中所有的维度表加载完成才开始进行事实表的加载。在开始相关事实表加载开始之前只有与该事实表直接相关的维度表和桥连接表需要完全加载。

由于存储在事实表中的数据容量一般非常大，以并行处理来加载是不错的思路。调度程序应该将 ETL 过程分为多个线程，这样可以同时运行并从并行处理获益。下一章进一步探讨优化事实表加载。

层次映射表

层次映射表专门设计来在维度中的层次中来回移动。见第 5 章。层次影射表不依赖于事实或桥连接表（当然，除非这个事实表本身包含层次）。一般来说，层次表可以紧跟在与之相关的维度表之后加载，但我们建议在长时间运行的事实表加载开始和结束这个数据集市处理的末端加载它们，更快。

无论层次在批处理中物理位置如何，它的成功或失败应该都不影响这个批处理中的其它处理。不要因为一个层次影射表处理失败就终止已经开始的事实表的

处理。影射表可以独立重新开始任何事实表加载。

聚合与分组对性能的影响

聚合函数和 Group By 语句要求数据库利用大量临时表空间。临时表空间是 DBMS 管理的一个特殊区域，用来存储需要处理的某些包括排序查询的工作表。大多数 DBMS 试图在内存中执行所有排序，在分配的内存满了之后就通过将数据写入临时表空间继续处理。如果你试图用 SQL 在数据仓库中创建聚合表，那不会有什么问题。

SQL 在执行它的服务器上进行处理。这就意味着如果你试图在你的抽取查询中进行数据聚合，你将很可能撑破源交易系统分配的临时表空间。根据设计，交易系统保留的临时表空间与数据仓库分配的相比之下是非常小的。当你需要建立一个聚合表时，比较好的处理是利用 ETL 引擎或第三方工具的特殊功能来加快排序数据的速度。

你应该使用支持增量更新到聚合的专门工具来逐步调整你的聚合。

不要试图用带 Group By 语句的 SQL 在你的数据抽取查询中执行聚合。Group By 语句用语句中所有的列来建立一个隐含排序。交易系统一般不会配置来处理巨大的排序操作，因为这样的查询可能摧毁源数据库。抽取必要的原子级数据并随后在 ETL 管道中利用 ETL 引擎或专用排序程序来聚合。

使用标量函数对性能的影响

标量函数有一个输入值并返回一个值作为输出。标量函数通常有一个或多个参数。一般来说，函数给查询性能增加压力，尤其是那些必须一个字符一个字符的评估值的函数。以下的函数列表是众所周知的性能障碍者：

- SUBSTR（）
- CONCAT（）
- TRIM（）
- ASCII（）
- TO_CHAR ()

这个列表不是全部，只是作为一个例子来考虑在数据库中的各种不同类型的函数。比如，TO_CHAR ()是一个数据类型转换函数。如果 TO_CHAR ()降低了性能，你可以想得到 TO_DATE()和 TO_NUMBER ()也是一样。试着用操作符代替数据库函数。比如，在 Oracle 中，CONCAT（）函数可以用双竖线||来代替连接两个字符串。

数据库在处理函数方面变得越来越好。Oracle 已经引入基于函数的索引，这样对包含基于函数约束的查询的响应时间就加快了。可以查看其他数据库厂商的高级功能比如他们集成了 ETL 功能到他们的基本产品中。

避免使用触发器

数据库触发器是由数据库的事件触发执行的存储过程。事件如删除、插入或更新数据都是与数据库触发器相关的通用事件。问题是每个事件都是由一条记录要进入数据库时触发的，因此数据库必须在每个记录之间执行完这个存储过程。触发器在降低处理速度方面是声名狼藉的。

如果你需要基于事件执行的一个处理，使用 ETL 引擎来执行这个任务，尤其是执行像附加审计元数据到记录中或执行业务规则这样的任务时。ETL 引擎可以执行这样的任务在内存中无须 I/O。

克服 ODBC 瓶颈

第 3 章提供深入分析开放数据库连接管理器（ODBC）的各个层，但是在这里还是值得再次提到这个话提，在你的 ETL 引擎和数据库之间进行通信时 ODBC 并不是必须的，它可以—应该—避免。ODBC 在每个 SQL 语句上增加代码层。它相当于在引导一个类时使用一个转换器。这个消息最终可以通过但是这是很慢的过程。因此有时候，时间就耗费在这个转换上了。

在处理中尝试获得本地驱动程序来参与在 ETL 引擎和数据库之间的通信。记住，就像一条链子，它的强度取决于它的最薄弱的一环一样，ETL 只是与它的最慢的部分一样快。如果你在你的 ETL 解决方案中包含 ODBC，你将无法获得最佳的性能。

利用并行处理的优势

以并行方式处理 ETL 可能是提高性能的最强大的方法。每次你增加一个处理，吞吐量相应的增加。这个部分不讨论技术架构的可选项（SMP，MPP，NUMA 等等）。相反，我们提供 ETL 并行处理与顺序处理相比的优势。

并行处理，最简单的定义就是一次同时处理多个操作。你可以想像，任何一个假设的 ETL 处理都存在三个主要操作--抽取、转换和加载。你可以，也应该，尽可能的从并行处理多个 ETL 任务来获得优势。

并行抽取查询

并行抽取查询进行的有效方式是逻辑上分割数据集到相等大小的子集中。我们说逻辑上分区是因为数据分区通常是一个物理数据库的功能。这时候，你可以基于一个属性的范围来分割数据。比如，你可以按年分割有效日期。因此，如果你有十年的数据，你就有十个逻辑分区。每一个分区由一个独立的 SQL 语句来检索并同时执行。这个方法的潜在问题是数据库将每个 SQL 语句标志为独立的处理，并试图最小化分配给每个处理的内存。因此，如果抽取查讯对内存需求非常强烈的抽取查询时，有可能由于复制和执行这样的复杂处理而导致服务器达到他的极限。

幸运的是，大多数 DBMS 有并行处理查询的能力，实际上它是同样的处理同时集约的管理内存。最优化并行解决方案一般由两种技术组成—产生几个抽取查询，每个有不同范围的值，接着以数据库特定的并行查询技术来并行执行每一个处理。

那些支持它的每个数据库都有它本身执行并行查询的语法。在 Oracle 中，通过在建表时设置 degree 参数来启用并行功能，或者在表创建后修改表来启用并行查询。运行下面的查询来检查看表的并行查询参数是什么：

```
Select table_name, degree, instances from all_tables where

table_name = '<TABLE_NAME>'
```

前面的查询返回三列：

- Table Name ：并行性检查的表的名称。
- Degree ：处理一个查询时在每个实例上使用的并发线程数量

- Instances ：查询可以跨越来处理一个查询的数据库实例的数量

你不必使用 Oracle Parallel Server 来运行并行处理。只要并行程度设置大于 1，查询运行就按照设定处理同样数量的并行。然而，要跨越实例，必须要有多个活动的实例并且要有正在运行的 Oracle Parallel Server。

不幸的是，大多数事务表的并行程度默认是设为 1 的。你可能会发现，源系统的 DBA 不愿意为数据仓库团队修改表。所幸，你不必需要他们这样做。因为抽取查询是一个静态的、可重用的 SQL 语句，它允许插入一个提示来覆盖物理的并行程度来引导 DBMS 在运行中并行查询！动态并行功能是一种强健的机制，对加速抽取查询来说是无价的。

要动态并行一个查询，插入一个提示来指定要并发运行的线程的数量和跨越的实例的数量。

```
select /*+ full(products) parallel(products,4,1) */
product_number, product_name, sku, unit_price from products
where product_status = 'Active'
```

查询中的提示以/*+为开始，以*/为终止的标志。注意到这个提示使得查询以四个不同的线程在一个实例上动态执行。虽然有四个执行线程，处理一般也很难接近四倍的总吞吐量。显然，其它变量，比如内存和源系统与表的物理属性，这些变量 ETL 团队没有控制权但也会影响性能。因此，不要期望性能与指定并行度的数量成百分之百的正比的增长。咨询 DBMS 用户指南来计算为特定的情形决定最佳的并行度设置。

并行转换

如果你用 SQL 来做转换逻辑，可以在任何 SQL DML 语句中用上一节的提示。然而，如果你正在用的是专门的 ETL 工具，那么直到现在你可能有两个并行你的转换的选择：

1. 购买一个可以进行本地并行操作的工具。
2. 手动复制一个过程，分割输入数据，然后并行执行这些过程。

显然，你努力想要第一种选择。然而，有些工具并不支持作业本地并行化。如果数据集巨大，并行化不是一个好选择但却是必要的。幸运的是，ETL 厂商认识到数据量增长是非常迅速的，因此他们正在快速的增加并行化功能到他们的工具集中。

如果没有工具（或一个工具的附加功能）来对转换过程进行并行处理，直接按照厂商的指南来获得最佳结果。

另一方面，如果要手动复制过程，你应该遵循以下步骤：

1. 分析源系统来决定分割数据的最佳方式。如果源表是分区表，就用分区列。如果不是分区表，检查日期字段，如生效日期，增加日期等等。通常，按照日期分区能将数据很好的、平滑的分配数据量到各个分区中。常常，在订货这样的例子中，数据量增长跨越时间分区（业务良好的标志）。在那样的情形，考虑按照主键的范围分区或者建立一个哈希分区，可能在主键上做 MOD，这是均匀分割数据的一个简单方法。

2. 下一步是复制 ETL 处理，倍数等于想要同时运行的并行线程的数量。寻找一个工具来最小化多余代码的量，如果有四个 ETL 处理的副本，所有四个副本都需要维护。利用可以执行同样的作业在不同的批处理以不同的数据集的工具更好。

3. 最后，设置几个批处理作业，每个对应一个处理，基于第一步决定的值范围来收集和输送合适的数据集。如果源系统是非常容易变化的，我们建议运行一个预处理扫描源数据并决定为每个复制的 ETL 作业均匀分配数据集的最佳范围。那些范围（开始值和结束值）应该作为参数传给 ETL 作业使得这个处理成为一个完整的自动的解决方案。

如果你确实有大量的数据需要放入数据仓库，顺序的处理所有的 ETL 操作是不够的。那就必须找到一种 ETL 工具，可以提供天然的以并行方式处理多个操作来获得最佳吞吐量（这里并行功能是转换引擎直接内置的，不是执行并行扩展）。

并行最终加载

上一节讨论并行化抽取查询，我们的假设是你不能控制数据库的结构因此需

要增加一个数据库提示来将查询分成多个同时运行的线程。然而，在目标方面--这个数据仓库的展现区域，你确实，或者至少应该对如何设计数据结构有一些发言权。数据仓库团队最感兴趣的就是在创建表的时候让表具有多种并行度。

这一章的前面部分，我们介绍了最小化 SQL 插入、更新和删除以及利用块加载器的方法。此外，当使用 Oracle 的 SQL Loader 时，你应该确保设置 DIRECT 参数为 TRUE 来防止产生不必要的日志。

现在我们将要介绍另一种提供抽取和转换并行处理的技术：产生 SQL Loader 多处理--每个分区一个--并且并行运行。当你同时运行许多 SQL Loader 处理时，必须设置 PARALLEL 参数为 TRUE。没有比下面这三条规则更快的方法来加载一个数据仓库了--至少在写这本书的时候还没有：

- 利用块加载器.
- 关闭日志.
- 并行加载.

使用块加载器的更多信息可以在第 8 章找到。Oracle SQL Loader 的详细参考请阅读 Oracle SQL*Loader：The Definitive Guide ，作者：Jonathan Gennick 和 Sanjay Mishra (O’Reilly & Associates 2001)

8.6 性能问题纠错

无论你的 ETL 系统多么有效，仍然有可能遇到性能问题。然而，正如 Robin Williams 在电影 Good Will Hunting 中所说的一样，“这不是你的错。”当你正在处理的是非常庞大的数据集时，有时要根据他们的规则来判断。不只一次，我们已经完全配置好所有的条件了，但是由于一些无法解释的原因，它就是不起作用！

当你发现一个奇怪的作业使性能相当缓慢时，不要删除它。直接采取一种程序化的方法来找出处理中引起瓶颈的操作并标出那个特殊的操作。监视如 CPU，内存，I/O 和网络流量这些地方来判断任何高层次的瓶颈。

如果在实际的 ETL 处理之外没有检测到任何实际的瓶颈，你需要深入到代码中分析。使用排除过程来缩小潜在的瓶颈范围。要排除操作，你必须能够区分每

个操作并且分别测试它们。如果整个处理都是用 SQL 或其它过程语言手工编写，那么代码分割看来是非常困难的。实际上所有的 ETL 工具都提供一种用来将处理分割成独立部分的机制以帮助判断无法预知的瓶颈。

最佳的策略是从抽取过程开始；然后顺序沿着每个激素、对照、聚合、重格式化、筛选或其它任何转换过程的操作；最后测试实际数据加载到数据仓库的 I/O。

要开始分割过程来检测瓶颈，复制 ETL 作业并根据需要修改这个 ETL 的副本来包含或排除相应的部分。当你一步一步执行整个处理，可能需要删除这个副本并重新复制作业来恢复所做的改变来测试以前的部分。按照以下步骤来分割 ETL 处理的各个独立部分来标志出瓶颈：

1. 隔离并执行抽取查询。通常，抽取查询是处理的第一个操作并通过管道直接传输数据到下面的转换中去。要隔离这个查询，临时删除所有转换以及这个抽取查询之后的数据库交互操作并将查询结果直接写到一个文本文件中。还好，ETL 工具可以给出查询的执行时间。如果没有，使用外部监视工具，或在 Oracle 中，在执行处理之前使用 SET TIMING ON 命令。这个设置在查询执行完成时自动显示执行时间。如果这个抽取查询返回这些记录的实际速度与整个处理中的相比不是更快的话，那你就找到你的瓶颈了，因此你需要调优你的 SQL；否则，开始下一步。

注意：根据我们的经验，糟糕的 SQL 是大多数性能低下的共同原因。

2. 关闭筛选。信不信由你，有时将数据载入 ETL 作业再对数据进行筛选可能引起瓶颈。要测试这个假设，临时关闭或移走任何这个抽取查询之后的 ETL 筛选。当你执行这个处理，观察吞吐量。要知道这个过程可能很长时间，但是重要的是这个过程到底处理了多少数据量。如果没有筛选的吞吐量实际更快，考虑在抽取查询中应用一个约束来过滤不要的数据。

3. 去掉查找。索引数据在被 ETL 处理使用之前缓存在内存中，取决于你使用的产品。如果检索大量数据到你的查找中，缓存处理会消耗大量时间来将所有的数据载入到内存（或硬盘）中。每次关闭一个查找，然后运行处理。如果观察到随着一个或多个查找关闭吞吐量有了提升，你必须最小化检索进入缓存的行和列

的数量。注意，即使你不缓存你的查找，你仍然需要最小化查找查询返回的数据量。紧记你只需要备查找的列以及被选择进入你的查找表的列（在大多数情形，维度的代理键和自然键）。任何其它数据通常只是引起不必要的 I/O 因此应该被除去。

4. 谨慎的排序和聚合。排序和聚合往往是消耗资源的。排序尤其厉害，因为他们需要整个数据集都装载到内存中来处理。关闭或移走任何资源敏感的转换如排序和聚合并运行处理。如果没有这些部分你可以观察到实际的性能提高，移动那些操作到操作系统中。很多时候，在数据库和 ETL 工具之外进行排序或预先排序更快。

5. 隔离和分析每个计算或转换。有时最简单的转换会引发最严重的性能问题。在检测瓶颈时，每次只执行一个操作，察看有没有类似暗示默认值或者数据类型转换这样的操作。这些看似对性能无关紧要的操作有可能给 ETL 处理带来巨大影响。隔离和分析每个计算或转换是检测和补救瓶颈的一个好方法。

6. 消除更新策略。一般来说，在 ETL 产品中打包的更新是出名的慢，在这里并不推荐应用在大量数据装载中。如果工具升级了，在应用新版本到生产环境之前一定要测试。如果更新策略影响了系统性能，那么一定要先隔离插入，更新，删除等操作，然后在一个专门的线程里执行操作。

7. 检查数据库 I/O。如果抽取查询和 ETL 管道流程的其他转换都有效执行，那么就该检查一下目标数据库。对目标数据库的监测比较简单，将原本插入数据库表的数据装载到一个平面文件中，如果有比较明显的效率提升，那么就说明你还要好好准备你的装载目标表。记得要去除表上所有约束，索引以及块装载器。如果还不能达到预期的性能，建议在部分数据装载中使用平行装载策略。

8.7 增长的 ETL 吞吐量

这部分是关于分类的一个摘要。ETL 开发组都期望能够生成一个有最大吞吐能力的 ETL 流程。为此，我们推荐 10 条规则来办帮助 ETL 设计提高处理能力到一个较高的水平。这几条规则同时适用于手工编码方式和工具设计方式。

1 减少 I/O。 最好不使用 staging tables。以管道的方式处理 ETL 流程，保证

数据从开始抽取阶段到最后装载都是在内存中处理。

2 消除数据库读写。 如果必须使用 staging tables，当你把数据写入磁盘的时候，最好用平面文件替代数据库表。

3 尽可能的过滤。 尽可能的在处理的上游减少纪录数。避免不必要的数据进入数据仓库目标表。Avoid transforming data thatnever makes its way to the target data

4 分区和并行。 增强处理能力的最好方法是用多线程并行处理数据。

- 用并行 DML 在源系统并行查询
- 用管道方式并行处理数据和 staging
- 分区设计，并行装载目标表

5 更新增量聚合。重新构建聚合计算对处理消极影响很大，应改尽量避免。应该 process deltas only 并且增添纪录到现有的表中。

6 只抓取所需要的（行和列）。同筛选的建议一样，不要在处理中添加不需要的纪录。同样的，也不要选取不必要的列。

7 块装载/减少日志

- 使用数据库块转载功能
- 减少更新操作，用删除和插入替代
- 关闭日志
- 将 DIRECT 设置为 TRUE

8 删除数据库约束和索引

数据库外键约束对处理能力来讲是一个很大的开销。他们应改被永久删除（除非它们是由聚合策略所建立的）。如果必须外键，那么在 ETL 处理之前删除它们，然后在后置任务上再加上。在更新和删除操作上保留索引，来支持查询语句中的 where 子句。在插入操作中删除所有残留索引，然后在后置任务中重新建立。

9 消除网络拥堵。保证供作中的文件是在本地磁盘驱动器上。同样的，将 ETL 服务器放置在数据仓库服务器上。

10 让 ETL 系统处理工作。最小化对数据库功能的依赖。尽量避免使用存储过程，函数，数据库键值发生器，触发器。确定副本。

还有很多类似这些的规则，已经在本书的前面章节讨论过了。为了有一个更加全面详细的认识，可以查看本书的第 5，6，7 章的细节内容，来获得最佳设计策略。

以上这几条规则，在下面会简要讨论。

提示：重新建立索引可能会占用大量时间，建议在大数据量目标表上建立分区。这样不仅可以在分区上 truncate 或重新转载数据同时保证其他分区上表的完整性，还可以删除和充新建立分区上的索引，而不必担心数据维护。重新建立索引子集可以在 ETL 后置任务中节省大量时间。

关于减少输入输出的争论

由于数据库和操作系统之间输入输出相互作用是不同的，在这一章就不从技术操作上来解释 I/O。但是我们还是要重申在最小程度上减少 I/O。很明显，我们可能因为各种原因涉及各种数据。首要原因是当我们需要最小化对源系统读取。在这些情况下，最好在抽取结果生成的时候就把他们写入磁盘。这样，如果发生错的话，你就可以根据已经保存的副本上处理数据，而不用重新访问源系统。

过多的 I/O 会在很大程度上影响性能，大多数情况，当各个处理的数据吞吐量增大的时候可以忽略中间表或中间文件而不会有功能上的损失。如果发现生成了很多 staging tables，并且很多任务读写 staging tables。那么立即停止，回到上一个步骤分析整个方案。通过消除 staging tables，不仅可以减少 I/O（最大的性能阻碍），还可以帮你减少需要维护的用户数，简化批处理和时序安排。

消除数据库读写

ETL 处理会因为各种原因要求数据处理在磁盘上，在磁盘上进行分类，聚合计算，中间计算或由于安全问题而进行副本保留。可以选择使用数据库数据或者是平面文件数据。数据批量下载到数据库中比到平面文件中需要多得多的资源。

ETL 工具可以对平面文件如数据库那样容易的操作数据。因此，如果可能的话，在 data-staging 区连续使用平面文件是一个不错的选择。

但是，由于数据仓库的最终目标是中呈现数据时的保证最佳响应时间，并且解决方案是与数据库管理系统相关的。可是，尽管 ETL 过程中可能需要读取中间数据，但它在最终用户对数据仓库的展现层进行操作时并不会同时查询数据。尽管数据仓库要支持不可预见的查询，ETL 的集结处理也是静态和重复性的。

优化的性能可以直接从由对数据库集结区中间表的操作转换为对平面文件的操作体现出来。但是，这样做的缺点是，在元数据库中对数据库的相关纪录将会丢失。所以选择使用平面文件前你必须手动维持连接任何和文件相连的元数据（除非你的 ETL 工具能直接取得元数据）。

尽快进行筛选

这个技巧指出了在 ETL 设计中最常见的错误。无论何时，只要我们要对现存 ETL 处理的设计进行回顾，第一件事就是查看筛选的设置。筛选是大多数 ETL 产品的组成部分，它的作用是对抽取的数据进行限制。筛选的作用是非常显著的，在很多时候你需要对没有编入索引的源系统的域进行限制。比如你要在一个没有编入索引的域中用 SQL 抽取的方法来应用限制，源数据库需要对整个表进行扫描查找，这是一个非常慢的过程。相反地，如果你的源系统对你需要限制的域做了索引，使用这个域来做筛选，去排除不需要提取的记录，这种方法是首选。

我们经常看到筛选是被放置在非常复杂的计算或处理密集型和 I/O 密集型的数据查找的下游。当然，有时你在应用筛选之前必须完成某些计算。比如，当你要计算一个网页的停留时间，你必须计算当前页面和在你除去不想要的页面之前的下一个页面之间的差分。

一般来说，在需求允许的情况下，你应该尽量保存而且应用 ETL 筛选在数据流的上游。特别是，当筛选能立即被放置在从源系统中抽取数据的最初的 SQL 状态下，并且在任何计算和查找发生之前，这样做，筛选的优势就显示出来了。如果你做完数据转换之后将他们丢弃，会浪费了宝贵的处理过程资源。

只有当源系统数据库没有合适的索引来支持你的限制，我们才选择应用 ETL 筛选来降低处理行的数量而不是通过 SQL 抽取来应用限制。因为其它因素比如

表的大小、SQL 的复杂性和网络的配置等，这些在数据获取性能中也是很重要的因素，它们在确定最佳的解决方案之前的测试是很有意义的。

分区和并行

对 ETL 过程进行分区和并行不仅仅是一个设计问题；它要求特定的硬件、软件和软件解决方案。你能够在没有执行 ETL 并行和 visa versa 的情况下对数据分区。但是如果要在没有分区的情况下做并行，将会导制瓶颈。对不可预知的源数据，一个有效的分区和并行策略是在你的目标表中创建复述的分区和在 ETL 过程中应用相同的分区逻辑。但是要注意：这种的分区对数据仓库 ad-hoc 查询可能不是一个最佳的解决方案。注意要与数据仓库的构建紧密结合来实施最合适的分区策略。参照这一章的前部分关于平行技术和优势。

更新增量聚合

这个聚合是指存在于数据仓库的特别是对降低查询响应时间数据仓库总结性表。聚合在数据仓库中可以增强性能。因为查询必须扫描数千万行的数据，而现在可以只是扫描几百行就能实现相同的效果。这样极大幅度的降低访问的数据量。这是因为在 ETL 机械的滚动过程中结合附加的事实数据。更加综合的聚合取决于复杂的商业规则，这些商业规则并不是我们在多维世界所说的聚合。记住一个聚合是用于与一个查询重写性能是相结合的。这个性能适用一个相对简单的规则来判定汇总是否能够被使用，而不是在查询时间对原子数据的动态汇总。

聚合是在成熟的数据仓库环境下应用多种不同的方式计算：

- 仅对最近载入的数据计算汇总数据。比如在产品和地理维度上在 DBMS 外对最近载入的数据进行分类和汇总。换句话说，当本地 OS 的分类速度快的时候不要使用 DBMS 内的分类。不要忘了，汇总计算是由一个分类操作后再合计操作组成的（建立对行的分组）。
- 通过在适当的位置增加或减少数据来修改现存的聚合。这个选项是对汇总的调节，一个现存的汇总生成将使时间延长，当这段时间包含着当前的下载时间，它将会被修改。或者当汇总的标准被改变的时候，现存的汇总也会被修改。这个是会出现的，比如说，如果一个产品类别的定义被修改，而且一个汇总在这个类别级别上存在，或者在这个类别级别上

有一个 roolup。如果对这个类别的调节是非常复杂的，一个性能保证的检查需要可以运行，通过下面的原子数据精确地检查汇总。

- 从原子数据完整地计算汇总。这个选项，叫做完整的滚动 roolup，是用于当一个新的汇总被定义或才当之前的两个选项对管理者来说是太复杂了。在最后增加的 ETL 过程，当只有几百行的记录是新的或者被修改，我们如果再次取得数万或者数百万（甚至数亿）行记录就没有意义。你必须选择一种机制来只从源系统重新获得 detas。有几种方式来实现对已改数据的获取，这取决于源业务系统中可利用的资源。参照第六章关于在源系统中许多获取已改变数据不同技术的列举和决定在你特定的情况下最合适的的技术。

只装载你所需要的数据

只为获取几百行新的数据或者获取来自最新从 ETL 过程中增加的数据而加载上百或上千（甚至上亿）行数据是毫无意义的。你必须为获取源系统中经处理过的“绝对值”数据选择一种机制。有多种数据转换的方法去获得源系统中有用的数据。第六章中提到的多种为获取源系统中有用数据的解决方法，还有为个别技术需求提出的建议和解决方法。

为了增量下载，一旦将行记录裁减到一个易于管理的大小，接下来你必须保证不会返回不需要的多余列。在 ETL 工具做查找对照工作时，返回过多的列是一个常见的错误。一些 ETL 工具会在表中自动选择所有的列，不管他们是否被需要或者对查询是否有用。要特别注意那些明确的不用选择的列，这些列在处理过程中是不重要的。当你要查找译注键,你只需要一个维表的自然键和需要译注键。维表中任何其它的列的在译注键的查询过程是多余的。

批量装载/消除日志

批量装载是在数据仓库中一次插入一行数据的另一个可选方法。就像是一个处理系统。利用批量装载最大的优势就是你可以不用数据日志并且并行装载。写入日志会消耗大量资源，和 I/O 一样在数据仓库环境下日志不是必需的。对批量装载的特定装载技术和优势是贯穿于这本书中。

消除数据库的限制和索引

另外一种在下载数据仓库中有下面影响的方式是从 ETL 过程中的对象中消除所有的限制和索引。记住，数据仓库并不是业务性的。所有的数据是通过一个受控制，受管理的机制——ETL 来输入的。所有的 RI 应该在 ETL 过程中加强，使 RI 在数据仓库水平变得多作并且不必要的。在一个表被加裁后，ETL 必须运行一个处理过的程序来重新建立任何一个被消除的索引。

消除网络瓶颈

不管什么时候都需要通过网络来传送数据，这个操作是易受攻击的，并且使性能降低。取决于你的基础构造，有时候可以在数据仓库服务器上运行 ETL 引擎来减少网络瓶颈。更进一步，通过在内部磁盘驱动下储存所有的集结数据，而不是通让数据在网络中流动，只在 ETL 过程中“触碰”一下数据。.

这个建议是一个让人为难的规定，有可能在数据仓库服务器中加入 ETL 引擎可能会使性能更差，而不是更好。这需要 ETL 工作组与数据仓库，硬件销售商构建出一个较好的解决方案。

使 ETL 引擎工作

ETL 产品是专门为抽取、转换和装载大量数据所设计的，而不是其它没有数据的仓库解决方案。在允许范围内，大多数的数据库是为支持业务处理和操作性应用而设计的。数据库存储过程对加载数据的应用有很好支持作用，但对一次性大数据集的处理并不是最佳的。指针的使用-------每一个记录在进入下一个之前都被分别分析-----在处理大数据集的时候是非常慢并且经常导制不可接受的性能。不是在数据库中利用 ETL 引擎，而不是存储过程来操作和管理数据是很有益的。

8.8 总结

这一章提供了需要选择来开发 ETL 系统的技术的一个总的摘要和一些例子。

必须从选择一个开发环境开始：或者是一个从我们所列出来的供应商中专注的 ETL 工具包，或者是一个由脚本语言与从一个低水平的编程语言所驱动的基于操作系统需要的开发环境。

在这一章的下半部分，我们提供一些为了实现高速的批量下载，加强 RI，实现并行的优势，计算维度汇总和麻烦的性能问题的指导。在 DBMS-专业技术指导。

现在我们已经对在第 8 章中操作和管理这个我们已经建立的有用的技术包做好准备。

9 操作

“整个系统的速度取决于最慢的部件”。

——Gene Amdahl

开发装载数据仓库的 ETL 过程仅仅是 ETL 开发周期中的一部分，生命周期中的其它阶段主要是执行这些过程。在装载数据仓库过程中，不管作业是实时执行还是批处理方式执行，其调度时间、执行顺序和执行环境都是关键点。另外，当新作业创建后，新作业的执行必须与现有 ETL 过程无缝集成起来。本章假设已经创建了 ETL 作业，我们把重点放到 ETL 的操作策略上来。

在本章中，我们将描述如何创建一个 ETL 操作策略，来保证数据仓库的数据即时可用。本章前半部分，我们将描述在系统设计完成后支持 ETL 操作的关于 ETL 调度的一些技巧和技术。

本章后半部分描述了许多在作业或系统一级度量和控制 ETL 系统性的方法（我们在第 7 章中描述了数据库软件的性能问题）。可能会有数十种控制性的方法，我们将提供一个针对你的环境非常重要的平衡的方法。

本章最后，我们提供了一种简单而有效的在数据库、开发环境、QA 环境、生产环境和基础文件系统级别实现 ETL 系统安全性的方法。

本章描述了执行 ETL 操作的最佳实践方法。这些操作包括了初始化数据装载、执行和监控每天的数据流、容量计划、性能监控、元数据资料库维护和控制对后台数据库的访问。

9.1 调度和支持

ETL 执行策略主要分下面两种：

- 调度。ETL 调度是一个综合的应用，而不仅仅是安排作业在给定的时间点执行这么简单。在现实情况下，作业在哪天执行其实是无关紧要的。一个有效的调度包括 ETL 作业间的关联和依赖关系的指定，并作为一个可靠的机制来管理执行策略的物理实施。

- 支持。一旦部署了数据仓库，它便一直成为一个关键应用系统。用户，也包括一些下游应用系统，都会依靠数据仓库提供他们需要的信息。如果数据仓库没有持续装载数据，就被认为是失败了。因此，为了确保 ETL 过程运行并完成，数据仓库必须能够被监控活动状况，并且必须由专人来支持。

ETL 的可靠性、可用性和可管理性分析

数据仓库具有最好的维度数据模型，最优秀的商务智能工具，并为最高领导层服务。但是在它成为企业分析信息的可靠性来源之前，还不能证明它是真正完美的解决方案。

新数据仓库的目标是创建一个一致的、可靠的数据源的备份，以此来支持企业数据分析，从而提高业务能力。为了保证成功，ETL 和数据仓库团队必须完成以下三个关键标准：

- 可靠性。ETL 过程必须持续无误运行。数据必须在任何粒度级别都是可靠的。
- 可用性。就像数据仓库管理者在最初的启动会议上向发起人和用户承诺的那样，数据仓库必须稳定的运行。ETL 作业必须在分配的装载窗口中执行并完成。
- 可管理性。请牢记，数据仓库永远没有终点。因此它必须随着公司的成长具有修改和扩展的能力。ETL 过程也必须随着数据仓库的发展而发展。为了实现可扩展性，就需要尽可能地使过程变得简单，把复杂的流程切成更小的、更简单的单元。同时，也要避免作业启动过程的过于集中。另外，设计执行策略的很关键的一步是确保支持 ETL 的能力。ETL 团队必须要为所有的 ETL 部件和为每一次为失败准备的恢复程序文档提供元数据。如果你是手工编码方式建设系统，请确保具有相应的管理技能，并能够全面地控制一个长期的软件开发环境。

ETL 管理员必须为数据仓库每一个阶段进行评价，通过可靠性、可用性和可管理性（Reliability, Availability, Manageability，简称 RAM）标准来为项目打分。作业和调度方法必须通过这三个标准，获得好的得分，才能有资格去部署。如果

没有元数据或恢复文档，则 points are deducted and the processes must be revisited and enhanced or coredted。对于过度复杂事实上几乎不能维护的作业，也必须改进到生命周期的下一个阶段。每一次数据仓库的部署在进行生产之前都必须取得好的 RAM 分数。

ETL 调度 101

调度 ETL 过程只需使它们运行即可，那么为什么这里要用一章的篇幅来介绍它呢？本章不仅仅讲述执行，更主要的是讲述执行策略。策略是一套精细的、系统的行动计划。任何人都可以执行一个程序，但是开发一个执行策略却需要一定技巧。

例如，在数据仓库和 ETL 设计回顾时，用户抱怨说数据仓库直到上午 11 点才可用。根据这一信息，我们立刻开始查看 ETL 作业，找到瓶颈所在，然后才可以提出补救措施。很快我们发现作业的效率很高，从开始执行到结束不应该要花整整三个小时时间(好像有问题)。“没错”，项目的 ETL 开发者说：“每天大约早上 8 点钟我来上班的时启动它们，在三个小时后，大约 11 点钟执行完成。”我们带着怀疑的态度问了一下开发者关于自动化问题，因为在他的实施中好像并没有使用。他声称从未参加过 ETL 工具调度方面的培训，因此不得不手工启动作业。

即使是执行你的程序，你也必须系统地来完成它。非常重要的是，ETL 团队需要理解环境中的工具，并且有能力合理地调度并自动化 ETL 过程，从而持续地装载数据仓库。

调度工具

任何企业数据仓库都必须拥有一个健壮的企业级 ETL 调度机制。主流的 ETL 供应商都提供其核心的 ETL 引擎提供打包的调度器。有些甚至根据一天中不同时段提供多种方式来执行 ETL 作业，有些提供了更复杂的 ETL 执行方案，可以基于变化的关键条件来触发作业。

如果你对与 ETL 产品绑定在一起的打包调度器不满意，或者你愿意冒险使用非产品的 ETL 方式，那么你仍会有多种选择。不管是否购买专门的 ETL 调度器，还是使用已存在的生产调度机制，或者手工编码来执行 ETL 作业，生产用 ETL 调度器都应该满足相应条件，成为一个可变的企业级解决方案。

ETL 调度器必须的功能

接下来的部分将介绍一些自动化 ETL 过程可用的选项。许多选项都可用，并且每一个都随着成本和易用性不同而不同。在特定生产 ETL 环境中需要相应的功能。当选择（或建设）ETL 调度解决方案时，请确定包括下面描述的功能。

- 令牌识别

通常，数据仓库需要从外部数据源获取数据。外部数据源提供者非常广泛，因此 ETL 方案必须能够应对这些数据。外部数据源通常是文本文件或者 XML 格式。读取和处理这些数据一般都比较简单，挑战在于如何识别已经存在的数据。与数据库源不同，数据库源可以通过查看审计列来识别新记录，而外部数据源通常都通过 FTP 将数据文件装载到文件系统的目录中。由于每次的格式都是固定的，因此 ETL 过程可以处理这些数据。但是 ETL 系统如何知道外部源的数据文件已经到达并开始其处理过程？ETL 系统必须能够识别文件已经在文件系统中存在，并自动开始执行。这个过程叫做令牌识别。

令牌是一些由文件系统创建的文件，用来触发 ETL 事件。具有令牌识别功能的应用系统可以轮询一个目录（或数据库表），用来标识令牌文件（或记录行）的到达。当处理文本文件、Web 日志或者外部数据源时，必须要防止重复处理同一文件，还要确保当文件迟到时不会错失 ETL 过程执行。令牌文件被认为是一个令牌，因为它不是必须要被处理的实际文件，它可以是一个索引文件，仅仅通过其到达来告诉处理过程开始执行。

- 日内执行

如今每日定时处理的方式已经越来越不被接受了，因为对实时性的要求越来越高。ETL 处理必须拥有在一天当中多次执行的能力，甚至可能是随需执行。过去按月或按天增量装载是完全能够满足需求的，在实时技术还不存在的情况下，12 小时、6 小时、4 小时增量，甚至是每小时更新已经变得越来越普遍。这些聚合需求意味着不但你的 ETL 作业必须是高效的，而且你的调度系统也必须能够承受每天大量的处理过程。

此外，你的处理过程必须能够跨越午夜，在其分配的时间窗口以外重新启动。以硬编码 SYSDATE-1 方式查找“昨天”的数据，使用这种操作方式启动并从源数

据中选择数据是不恰当的。ETL 系统必须能够从数据源捕获新数据，不管这些数据是何时创建的，也不管在何时执行处理过程。

- 实时能力

实时执行是数据仓库不能被忽略的实际需求。它是如此重要，以至我们单开一章来讨论它。第 11 章列举了多种技术来进行实时 ETL 执行。实时 ETL 已经成为众多企业的家常便饭，越来越多的用户现在期望数据仓库可以不间断地更新，而且对“陈旧”的数据越来越不耐烦，实时 ETL 不是奢想，而是很实实在在的需求。

此外，随着数据仓库的发展，其价值正被越来越多的末知用户所认识。因为它提供了整洁、一致、可信赖的数据，数据仓库本身正成为一种源系统。交易应用系统越来越依靠数据仓库提供的标准数据源作为参考源数据。为了实现这一所谓的“内循环”模式，数据仓库就必须实时更新来支持操作应用系统。

- 命令行执行

ETL 产品大多把主要精力放在为它们的调度软件包提供友好的可视化用户界面（GUI），从而为缩短入门开发者的学习时间，也为资深的 ETL 专家节省开发时间。但是，大多数企业级系统操作环境都需要从命令行界面执行 ETL 作业的能力。因为日常维护人员往往也需要维护其它更多的应用系统，而很难针对每种应用系统学习另外不同的界面。因此，你的 ETL 应用系统必须能够从命令行执行数据仓库 ETL 过程，从而使你的系统操作成员可以胜任这项工作。注意，正是因为这种原因，主要的 ETL 工具产品都提供命令行执行选项。

- 通知和呼叫

一旦 ETL 系统开发和部署完成，它的执行方法就变成了自动操作。就像时钟一样，无需人为参与，也不会发生错误。如果执行过程中出现了问题，则必须以电子邮件方式通知维护人员。ETL 解决方案必须能够针对不同的人员、不同类型的作业以及不同类型的错误通知相应的人。在我们写这本书的时候，无线 PDF 和智能电话正迅猛发展，这些设备将成为个人操作的标准设备。这些设备上可以显示复杂的文本和图形信息，操作者可以远程向 ETL 系统发出指令。另外请注意下面的警告！

E-mail 通知和呼叫必须完全自动化，因为可能没有时间等待维护人员被人工通知。自动通知可以用下列方式实现：

集成 ETL 工具：一些主要的 ETL 产品在他们的调度应用中集成了呼叫和通知功能。这些功能也许并不十分稳定，但已经比以前好多了。最起码，你需要区分成功的装载和失败的装载，并通知相应的维护人员。另外，对于失败的关键信息必须能自动发送消息（例如，作业名称、失败时间、装载的行数、失败的行数以及最后动态失败消息等）。

第三方消息应用系统：一些公司提供了即时消息产品，用于支持 24*7 系统运行，最小化宕机时间。另外，操作管理和监控工具一般都具有通知功能，如果你的操作支持团队使用了这样的工具，则可以使用该功能。

客户化脚本：你可以选择在操作系统上手工编制脚本 来提供 e-mail 通知的执行策略。这些脚本必须能与 ETL 作业交互，并在需要的时候触发。

当设计你的客户 e-mail 通知系统时，一定要小心嵌在脚本中的邮件地址。脚本在文件系统中是简单的文本文件。脚本非常容易被一些垃圾邮件发送者获取，并且用垃圾邮件攻击你。因此如果有可能，一定要使用来自安全产品的加密技术或解决方案。

- 嵌套批处理

批处理是一组作业或程序，作为一个简单的操作一起运行。通常，ETL 作业被分组或分批，来装载一个数据集市。而由数据集市组合而成的数据仓库，则由一批数据集市装载批处理来装载。装载批处理的批处理称为嵌套批处理。每个嵌套批处理可以包括 ETL 作业的若干层。例如，由于复杂的数据或业务规则，一个单一维表可能需要多个 ETL 作业来装载它。这些维表作业可被组合在一起作为一个批处理、这个批处理被包含在另一个批处理中，为数据集市装载维表的剩余部分。数据集市批处理又被组合到数据仓库批处理中，从而具有三个层次的批处理。理论上，嵌套批处理的层数没有限制。

ETL 作业一般都以嵌套批处理方式执行，几乎很少会在生产环境中运行单独的 ETL 作业。数据集市通常需要求每一个维表和事实表至少有一个作业，正如你所看到的，在装载数据仓库过程中，多层嵌套批处理非常普遍。因此，你的方案

必须可以管理嵌套批处理。批处理管理包括下列内容：

图形界面：由于装载数据仓库时嵌套批处理的特性，ETL 批处理通常会变得非常复杂。选择批处理管理工具时，其必须具有像 Windows 资源管理器一样的目录结构导航能力。如果没有嵌套批处理的图形展示，管理就无从谈起。开发者应该可以通过 GUI 创建、删除、编辑和调度批处理，还可以通过拖拽方式在嵌套批处理和外部批处理移动作业。批处理管理最好通过图形化方式实现，并且还需要同图形化一致的逻辑命令标准。批处理之间依赖关系的可视化是维护的关键，可以清楚的理解哪个作业属于哪个批处理，另外还可以识别批处理之间的依赖关系。

依赖关系管理：当某个作业的执行取决于其它作业的成功完成时，就有了依赖关系。作业间的依赖规则由执行策略决定，并且必须由 ETL 调度系统在执行时加以约束。如果业务规则需要的话，批处理管理工具必须能够监控失败的作业并中止相应的批处理。例如，如果维表作业失败了，则必须停止装载事实表的过程。但并不是所有的情况都需要这样严格的批处理中止策略。例如，如果一个外关联验证失败，则通常仍将继续装载相应的维表。批处理管理工具应该足够健壮，按业务规则要求基于嵌套批处理来设置依赖关系。

参数共享：参数值需要在作业之间传递，或者在最外面批处理中一次性设置，然后在整个嵌套批处理中作为全局数据使用。批处理管理器必须包含参数管理功能。更多关于参数管理的内容在本章后面部分将要详细介绍。

平滑重启：如果一个作业在执行过程中发生错误该怎么办呢？你如何确切知道哪些作业已装载哪些还没有？如果重新启动的话，批处理管理工具必须能够系统地确定哪些记录已经处理并装载，从而只处理剩下的输入数据。必须要注意装载过程中的失败点。通常，如果只为支持重启这种情况，ETL 系统应该有一些策略点（数据被写到磁盘的处理步骤单元）。而且，如果某个 ETL 步骤需要人工干与来纠正数据，则更要特别注意。这些手工操作至少要保存在日志中，以便在 ETL 处理步骤必须要重新运行时可以再次调用。

顺序/并发执行：有些情况下需要顺序装载表。例如，当表间有依赖关系时，就必须在装载子表之前装载父表。与某维表的外关联是这种父子顺序的最好例

子，同样也适用于普通维表和事实表。在装载完所有维表之前不能装载事实表。另外，有时顺序装载表而不是并发装载是为了平衡服务器的负载。如果你试图一次性装载数据集市中的所有维表，则可能会由于资源消耗过多而造成 ETL 服务器不能运行。相反，对于长时间运行的过程，你可以把作业拆分成多个小作业，让他们并发执行来提高装载效率。假设有足够的资源可用，则尽可能多的使用独立过程并发执行来最大化处理效率，并最小化装载窗口。更多关于并发和并行处理的内容在本章后面有详细地介绍。

预先/后续执行动作。在 ETL 过程之前或之后简单运行脚本并不属于执行管理范围。批处理管理必须在运行核心 ETL 作业之前知道已经成功执行了预处理脚本。而且，如果核心 ETL 作业成功执行完毕，必须要触发后处理脚本。最后，脚本除了能够在作业一级执行，同样也要能在批处理一级执行。这对于批处理的并发执行来说尤其重要，在每次批处理执行过程中最后完成的作业是不确定的。而且，在所有作业完成后还需要触发一个后处理脚本。

元数据捕获。批处理管理器控制下的所有元数据都必须可以被捕获、存储和发布。在最好的情况下，元数据应该存储在开放的资料库里，以便在其它应用系统间共享。每一个 ETL 作业都有一个调度执行时间，其执行频率、参数和恢复过程等所有元数据的形式必须能被展示，并且很容易被需要支持装载过程的人（也包括业务用户）获取。至少，元数据必须拥有报告能力，以便用户和开发者深入到数据仓库 ETL 操作的内部去观察。请参考第 9 章来了解更多关于 ETL 元数据库的内容。

ETL 工具在失败恢复方面已经做的很好，但平滑重启仍是一个苛刻的需求，还没有完全实现。很多情况下，对于处理过程中失败的情况，最安全的办法仍是删除已经部分装载的数据，然后重新开始这个失败的 ETL 过程。如果你使用工具包自动恢复丢失的数据，则强烈建议花些时间来审查已经完成的数据，从而确保数据质量和完整性。

- 参数管理

ETL 系统在整个开发周期过程中在不同的环境间迁移。由于这个周期也包括 ETL 系统内的代码测试，因此不能在环境迁移间修改代码。在 ETL 系统中定义变

量不能在代码中将参数写死。参数是在代码中替代常量的一种方式，一个健壮的 ETL 调度系统必须拥有为执行 ETL 作业管理和传递参数的能力。参数方式增加了 ETL 作业的灵活性，因此可以在不对应用重新编码的情况下平滑地改变环境或抽取规则。例如，一般情况下 ETL 作业的生命周期是在开发环境中开发，在测试环境中测试，然后最终迁移到生产环境中来支持产生数据仓库。ETL 生命周期中的每一个环境都有其特定的数据源、处理策略和目标数据库，以及文件系统和目录结构。通过参数方式来改变这些环境，ETL 系统就可以无需修改代码来指定相应的文件或数据库。因此，必须参数化环境变量，并在运行时让调度器传递应用数据到这些变量。

常用的参数项列表包括：

服务器名

数据库或实例名

结构描述文件名

数据库连接信息（不要把口令放到文本文件中！）

根目录或用来找到有用的控制文件的目录

元数据库连接信息

调度器必须能管理两种类型的参数：

全局参数。全局参数可以同时支持多个 ETL 作业。一般情况下，ETL 作业可以有很多个全局参数。例如，目标数据库名可以设为全局参数。否则，必须为装载数据库的每一个作业重复维护这个参数。

局部参数。局部参数只在一个 ETL 作业内有效。局部参数在某个作业内修改其值，而不会影响到批处理中的其它参数。局部参数的一个例子是设置用来从源表获取数据的最早日期。

本地 ETL 工具调度器是获取一个健壮的参数管理系统最好的证明，因为调度器通常都在本地与 ETL 引擎集成在一起。与 ETL 引擎的本地集成使得调度器可以在两个部件之间有效地通信。由第三方提供参数管理的架构可能效率不太好，但更灵活。不支持运行时设置参数的 ETL 解决方案会失去 RAM 的管理能力。

在企业环境中，为 ETL 作业中的参数生成元数据是非常重要的。如果没有健壮的参数管理系统，参数也可以通过文件系统中的文本形式进行管理维护。通过使用文本文件，操作团队可以简单地修改参数文件，而无需进入 ETL 系统。

ETL 调度器方案选择

在前面部分，我们描述了企业 ETL 调度系统应该具有的功能，另外还有一些实现同样功能的选择。在本节，我们提供了 5 种创建 ETL 调度器解决方案的选择：

1. 集成的 ETL 工具
2. 第三方调度器
3. 操作系统
4. 实时执行
5. 客户化应用

关键是要选择一种健壮的解决方案来满足你所有的需求，这些需求是基于你关于装载数据仓库的作业的需要，另外数据仓库建设要控制在预算范围内。最后要请教身边的专家来做决定。下一节将评估这五种选择的优劣。

- 集成的 ETL 工具

一般的 ETL 工具都会协同调度系统来执行由其工具集创建的 ETL 作业。有些工具提供了最少的功能，而有些则拥有健壮的调度应用系统。如果不是强制使用操作支持团队已经建成的工具（作为标准的调度工具），而 ETL 工具又包括一个健壮的调度器，则最好使用集成的 ETL 调度器。集成的解决方案的优势如下：

ETL 厂商的产品支持。为两个应用使用同样的服务级别协议（SLA）。富有 IT 经验的人可能会对多厂商解决方案带来的混乱深有感触，需要说明的是，这从来不是某个厂商的问题，而是其它厂商与其兼容不好所带来的问题。使用同一厂商的产品包可以充分发挥厂商的支持能力，从而加快技术问题的解决。

调度器与 ETL 引擎的集成。集成包设计为在部件间传递参数，并且从本质上强调作业间的相关性。作业间的依赖关系指的是某个作业的执行可能依赖于另一个或一组作业的成功完成，这种关系对于合理地装载数据仓库和从 ETL 失败中恢

复来说非常关键。

ETL 组中的工具集知识。由于 ETL 工具集是 ETL 团队的专用工具，他们可以在不学习其它应用的情况下直接建立执行策略。此外，一旦某个 ETL 作业充分测试过，则它在生产过程中极少会产生失败的情况。当作业确实失败时，ETL 团队通常需要某种级别上参与其中的能力。在 ETL 工具集控制范围内保留 ETL 调度，则团队可以很容易切换到技术支持角色来帮助失败的 ETL 过程进行恢复。

- 第三方调度器

许多生产支持部门都制定了一个独立的调度系统标准，所有其它应用系统都必须适应它。在某些企业环境中，数据仓同其它应用系统一样，必须遵守生产支持团队制定的规则。在这种情况下，ETL 由支持整个企业环境所有应用的调度系统触发。操作企业级的调度系统是超出 ETL 团队知识范围的一个专项工作。对于失败恢复不能顺利完成的情况，ETL 团队需要紧密地与生产支持团队合作。

如果生产支持团队坚持要通过他们标准化的企业级调度应用来执行 ETL 作业，请确保它具有支持 ETL 执行策略所需的功能，包括作业间的依赖关系、参数管理和通知及告警。

- 操作系统

通过本地操作系统调度系统（例如 Unix Crontab 或 Windows 调度器）来执行 ETL 过程是很常见的作法。即使你已经有了一个非常好的 ETL 产品，很多生产支持团队还是需要在生产过程中用脚本来执行 ETL 作业，因为这是整个企业中所有应用系统最常用的方法。通常，任何应用都可以在操作系统一级通过命令行或脚本来执行。在 Windows 平台下，批处理、.BAT、VBScript 或者 Jscript 文件可以用来管理 ETL 流程的执行。在 Unix 平台下，Crontab 用来运行作业。操作系统调度器可以直接执行 ETL 作业或者通过脚本来执行。正像大多数程序员所知道的，脚本的优势在于非常灵活。可以使用脚本语言来创建非常健壮的应用类型的逻辑。大多数的 RAM 标准都可以用脚本来满足。此外，Perl、VBScript 或者 JavaScript 可以运行在 Unix 或 Windows 平台下，在执行装载数据仓库的作业同时处理复杂的业务逻辑。事实上，脚本语言也可以在作业内提供逻辑功能。然而，我们仍然建议用健壮的 ETL 工具创建和维护 ETL 作业。使用脚本而不是 ETL 调度工具的劣

势在于它没有元数据，任何关于 ETL 调度的有用的信息都在脚本中，因此必须由程序员来解释脚本中的信息。在执行脚本中维护元数据可以使用两种技术。

电子表格。ETL 管理员或程序员必须维护一个电子表格，其中包含了重要的元数据，包括参数、批处理中的作业、执行时间等等。

数据表：动态脚本方案是元数据驱动的。所有相关的元数据都存储在数据表中（数据库或者文件），它们在运行时传递给脚本。当集成的 ETL 调度器不可行时，元数据驱动脚本是个可行的方式，应该创建并使用。

- 实时执行

如果数据仓库的一部分是实时的，则需要选择一种机制（在 11 章详细介绍）来支持实时的需求。整个数据仓库实时加载是很少见的，通常，数据仓库的一部分可能会实时装载，而其它部分则周期性的批处理装载。必需要注意两种 ETL 技术的结合，以此来确保一个无缝的、一致的解决方案。

- 客户化应用

我们总会选择创建一个客户化的调度解决方案。然而，还没有一种合理评判客户化调度应用的方法，但这并不影响我们创建它。如果选择用脚本执行所有的作业，则有必要创建一个应用来管理它们，但是创建一个客户化的 GUI 则显得有些过分。通常情况是使用脚本编程，再加上元数据表，是客户化 ETL 调度较为可行的方案。

装载依赖

在作业间定义依赖关系也许是批处理 ETL 作业最重要的内容。如果一个子维表装载作业失败，也许你还可以继续装载这个维表，但是如果维表装载失败，你还能继续装载事实表吗？这肯定是不赞成的。作业间依赖关系属于元数据，装载过程必须要了解这一点。ETL 功能需要操作元数据来进行恰当地操作。如果一个事实表 ETL 过程在所有其相关维表成功装载之前执行，它将不会发生装载错误，因为没有数据约束。另外，如果事实表没有设计成可以执行更新操作，则在重新执行处理过程前，所有错误数据必须手工恢复或者删除掉。手工干预是纠正失败 ETL 装载成本最高的方式，在 ETL 作业间强制依赖关系可以节约大部分这种成本。

依赖关系保证了桥接表和维表间，层系映射表和维表间的逻辑一致。使用前面所述的列表作为作业依赖关系定义的参考。具体说来：

- 在子维表成功完成之前不要装载维表
- 在维表成功完成之前不要装载桥接表
- 在父表（包括桥接表和维表）装载完成之前不要装载事实表

然而，头脑中始终要记住一个明智的数据仓库格言：针对每一个拇指规则，都会有相应的其它四个手指副规则。例如，如果一个维表设计为更新用来关联自己到子维表的外键，则当子维表装载失败，没有必要停止装载整个数据集市。因此，任何时候作业没有按期望的完成，调度器都要产生一个告警。

元数据

想像一下如果火车在没有公布其运行时间表的情况下运行，其它人如何能知道这列火车的运行情况？在没有公布元数据的情况下开始执行策略对于用户来说是非常危险的。在本章前面部分，我们讲到调度器必须能捕获元数据，包括内容和批处理、嵌套批处理的时间表，元数据必须像对数据仓库团队一样对业务用户可用。批处理元数据就像数据仓库的火车运行表，它应该事先告之何时数据将到达，何时对用户可用。

调度系统还应该在数据迟到时告诉用户，这与前面部分讲述的失败通知不一样。数据可用性元数据是交互的关键，同时也是响应用户期望的关键机制。用来通知用户数据到达的元数据属于过程元数据的类型。过程元数据捕获 ETL 过程中的操作统计。典型的数据包括装载成功的行数、拒绝行数、消耗时间、每秒处理行数，以及已完成每行所用的时间。这些都是非常重要的过程元数据，因为它可以帮助用户了解情况，就像火车站的广播一样。

在清理和提交步骤中收集的元数据有多种操作角色。比如给 ETL 团队提供建议数据是否已经适合提交给最终用户。审计元数据中的数据意味着可与普通数据组合生成特定方式的数据质量报告，它既可以用于报表结果的可信度，还可以用于保证报表的一致性。因此，清理和提交元数据为提高源始数据的数据质量所采取动作的导向。

所有由批处理管理器控制的元数据都必须能被捕获、存储和发布。最好的情况是，元数据应该存储在开放的资料库中，从而可与其它应用共享。元数据至少应该有生成报告能力，因此用户和开发者可以深入了解数据仓库 ETL 操作的具体细节。

9.2 迁移到生产系统

迁移过程根据因素不同有许多种情况，这些因素包括指导思想、技术架构以及 ETL 工具集。通常，ETL 团队属于开发团队的一部分，应该避免由于提供第一级的针对数据仓库的生产支持而过于分心，除非 ETL 组足够强大，可以保证拥有一支专门的 ETL 生产支持队伍。

本章的目标是假设 ETL 团队仅作开发工作，当其生产环境的各种任务准备完毕时，就把工作交给生产支持团队。然而，这一过程可能由于组织架构和环境中所用的工具变化而有不同。该节内容应该作为结束 ETL 生命周期实现的参考。

数据仓库的操作支持

有意思的是，大多数书籍和文章都讨论数据仓库团队需要如何维护数据仓库。事实上，至少是基于我们的经验，数据仓库团队（包括 ETL 团队）只是开发者和分析员。他们收集所有业务需求，分析它们并且创建数据仓库。一旦创建完成，他们通常把它交给另一支队伍，由这支队伍监控和维护生产环境。

数据仓库架构师和模型师负责维度建模，ETL 管理者负责组装按维度设计的数据仓库，ETL 发开团队创建装载数据仓库的过程，质量保证团队写测试计划来测试它们。数据仓库需要移交给组织内部的团队，以便支持每天的操作。如果用户只是一个小公司，或者数据仓库仍旧处于初级阶段，开发团队可能会支持 ETL 生产阶段的操作。但是随着数据仓库的成长（添加更多的数据集市），开发团队需要从生产环境的支持和操作这些日常的工作中解脱出来。

一旦 ETL 过程经过了开发和测试，则第一级的数据仓库操作支持和 ETL 工作应该由一支专门作生产监控管理的特定队伍提供，而不是由数据仓库开发团队来完成。只有当操作支持团队被问题困扰已久而不能解决时才需要求助于数据仓库团队。

版本发布

一旦 ETL 团队克服了 ETL 开发过程中各种难题的挑战，并完成了管理装载数据仓库或数据集市需要的作业的创建，则这些作业必须打包并迁移到下一环境，与数据仓库管理团队已经实现的生命周期一致。

与 ETL 工具提供商仔细讨论一下 ETL 部署的问题。工具是否支持所有版本增量的脚本，以便可以用简单的命令行实现从测试系统到开发系统的迁移。或者是否支持所有文件一次性的打开、检查、关闭和转换吗。

针对每一个数据仓库版本，开发团队应该提供如图 8.1 所示的发布过程文档。

环境：质量管理环境（QA）
服务器名：从na-dwqasvr-01迁移到na-dwprdsvr-01
ETL程序：从DWQA迁移到DWPRD
数据库名：从DWQA迁移到DWPROD

文档文件名：DW_Release_5.0_SALES_PROD.doc
最后修改日期：16-Feb-04 7:10:00 AM
修改类型：Major Release
发布号：5.0

系统修改请求的发布如下：

修改请求	项目领域	修改类型	描述	沿袭报告名称	PVCS脚本名称	PVCS版本
10845	零售	数据模式/ETL模式	新的星型计划 增加到零售分析	sales_mart_data_lin ease.xls	sales_by_month_release_5_build 1.zip	

系统需要如下手续来移动数据集市到另外一个环境：

☑ 标准发布手续

☐ 手续

☐ 项目编码

- DDL、ETL脚本为 **Sales_By_Month_Release_5_Build1.zip.**

☐ 表/视图

- 在DW_PROD数据库中顺序执行下面的脚本

从 PVCSProject:/DWQA/sql 取得DDL脚本
DW_SALES_MART_STEP1_CreateTablesIndexes.sql Version 1.0
DW_SALES_MART_STEP2_CreateStagingTables.sql Version 1.0
DW_SALES_MART_STEP3_CreatePubSynonyms.sql Version 1.0

☐ 安全视图修改

无应用

☐ SQL包

- 如下的包需要移动到DWPROD

从PVCS取得脚本
DWQA/packages/PKG_SALES.sql Version 1.0

☐ 配置修改

- 包括在ETL服务器上登陆 **tntnsnames.ora**

无应用

☐ ETL作业

- 迁移所有的PVCS或者ZIP文件

从下面的文件中导入所有的任务到DWROD中
DW_SALES_MART_Release_5.0_ETL_Jobs.zip Version 1.0

☐ 复制SHELL脚本

- 从 **/dw/sales/qa/shell**to **/dw/sales/prod/shell** 复制文件

无应用

☐ 复制SQL脚本

- 从 **/dsprojects/DWDEV /sql** 复制文件到 **/dsprojects/DWQA /sql**

无应用

☐ 复制平面文件

无应用

☐ 复制/编辑参数文件

- 在 **/dsprojects/DWQA/paramfile/param.ini.** 修改参数

如果如下参数在.ini文件里不存在，那么把值设成如下：
TgtStageUserName：设成dwetluser
TgtStagePassword: 设成DB用户的密码

☐ 附加注释

☐ 执行步骤

- 如下步骤来完成当前的发布

1、做所有的上面提及的步骤
2、运行BatchSalesLoad.sh

图 8.1 数据集市发布文档

数据集市发布文档介绍发布以及提供用于迁移和支持发布所需的技术细节。文档包括下列内容：

- 环境：包括源和目标环境。环境迁移通常是指从开发环境到测试环境或

从测试环境到生产环境。如果还有其它更多的专用环境，如用户体验环境或质量管理（QA）环境，这些环境也包括在内，这依赖于你的项目处于生命周期的哪个阶段。

- 服务器名：迁移过程中环境中服务器的物理名称。列出了 ETL 和 DW 服务器列表。
- ETL 程序：列出了程序所在的目录。如果在使用某种 ETL 工具，这可以为发布使用组件来标识正确的程序或作业。
- 数据库名：迁移源或目的数据库。通常是开发环境到 QA 环境或从 QA 环境到生产环境。
- 文档文件名：包括迁移信息的文件名，其中包括恢复过程的操作步骤。
- 最后修改日期：发布文档修改的最近时间。
- 修改类型：关于发布类型的描述，类型包括主体发布、小版本发布或打包。请参考第 2 章关于发布类型更详细的解释。
- 发布号：数据仓库成为最后发布结果的版本号。
- 修改请求号：对应的是部署过程中作为范围管理过程规定并包含的请求号。
- 过程：迁移作业的操作步骤。通常在发布文档中提供以下标准信息内容：

  1. 项目代码：版本管理工具中的字段，用来查找用于创建数据仓库的版本的代码。
  2. 表/视图：为新的数据集市创建新结构和索引所用的数据定义语言（DDL）。
  3. 安全：支持发布的任何新安全策略或需求。
  4. SQL 包：ETL 使用的数据库存储过程。
  5. 配置修改：针对发布所需的全局设置和配置项，例如 TNSNames（Oracle 远程数据库名称的别名）或目标数据库连接名（ODBC）。
  6. ETL 作业：发布所需的 ETL 作业，通常指版本控制管理器或 ETL 工具

中某个域。

7. Shell 脚本：ETL 依赖的操作系统级前置/后置执行的 shell 脚本。
8. SQL 脚本：ETL 依赖的操作系统级前置/后置执行的 SQL 脚本。
9. 文本文件：文本文件列表，也包括新数据源或 staging 文件的路径。
10. 编辑参数文件：用于带有可管理参数的环境，在这里列出针对 ETL 过程所用的新的源数据系统数据库或任何新的修改了的参数。
11. 附加注释：任何其它关于发布的指示或注释，可用于帮助系统操作团队完成迁移工作。
12. 执行步骤：执行作业直接的指导。对于新的数据集市或独立的 ETL 过程，你可以指定运行的调度计划或频率。

一旦数据集市发布文档完成，ETL 团队会将文档交给实施团队。在迁移过程中，ETL 团队应该随时待命，以防可能发生的影响生产系统的紧急问题。对于第一次发布的数据仓库版本，可能 ETL 团队的快速响应还不是一个关键需求，但随着数据仓库逐渐成为关键业务应用，任何关于系统的问题都将对整个企业组织影响重大，必须着力避免。一旦迁移完成，ETL 团队可以再回到常规数据仓库的下一阶段开发工作中去。生产环境的日常支持应该由专门的生产支持团队提供。下一节将介绍数据仓库所需的不同的支持级别。

在生产环境中支持 ETL 系统

本章的开始部分介绍了如何使 ETL 过程在获取针对调度、批处理和迁移 ETL 作业方面多个技术方面的运行和共享的建议。本节重点介绍处于生产环境的 ETL 的支持。通常来讲，任何软件实施的支持都包括三个级别：

1.第一级支持。第一级支持通常指针对各种情况的帮助桌面。如果发生失败或者用户收到数据错误，则会通知第一级支持。第一级支持团队利用由 ETL 团队提供的专门的过程来做各种尝试，如果实在不能解决问题则进入下一级支持。

2.第二级支持。如果帮助桌面不能解决支持问题，则通常会通知系统管理员或 DBA。支持的第二级支持通常精通技术，可以提供常规架构类型失败的支持。

3.第三级支持。如果生产操作技术人员仍不能解决问题，下一个通知的是 ETL 管理者。ETL 管理者拥有解决大多数生产环境问题的知识。有时 ETL 管理者会针对特定的情况与开发者或外部供应商交流来获得建议。

4.第四级支持。当所有这些都失败时，则直接找源头去吧。第四级支持需要实际的 ETL 作业的开发者分析代码来发现 bug 或解决问题。如果问题是由供应商产品 bug 引起的，则去找相应的供应商来提供对产品的支持。

在较小的环境中，将支持级别三和级别四进行合并是可以接受的，但这种合并通常会给第二级支持团队带来很大的负担。我们不建议每次发生作业失败都去找 ETL 开发者。第一级支持不能仅是电话服务，在进入一级支持前他们必须要做最大的努力来解决生产环境中的问题。

获得最优的 ETL 性能

现在，你已经通过本书完成了一个优秀的 ETL 方案。但是不要高兴太早，工作还没有完成！随着数据仓库的扩展，必须要确保 ETL 方案能伴随着一起成长。一个可扩展的 ETL 方案意味着设计的过程能够在不需要重新设计的情况下处理更大量的数据。设计必须能有效的执行，以此获得处理装载比现有数据量大得多的数据时所需的性能。可扩展性和性能是设计数据仓库 ETL 时不可忽视的两个因素。

ETL 开发者、DBA 和数据仓库管理团队将从本章获益非浅，因为它描述了监控和提高现存 ETL 执行效率的策略。本章假设你已经完成了分析工作，设计好了逻辑数据流程以及实现了物理 ETL 过程，从而直接进入到应用新的或现有的 ETL 作业来获取更优技术性能的细节和技术。

在本章末尾，你将得到在 data-staging area 如何处理安全问题的一些技巧。我们尤其阐述了 FTP 协议的不足，并提供了加密和解密数据方式来保证有效的安全的方法。读完本章，你将能够对数据仓库团队提供专家级的 ETL 调优技术。本章所涉及的技术都是针对 data-staging 环境，而不能应用在数据仓库展示层调优。

如果你需目标数据仓库的优化信息，我们建议你阅读 Gary Dodge 和 Tim Gorman 写的<<Essential Oracle8i Data Warehousing:Designing,Building,and Maintaining Oracle Data Warehuses>>一书。

估计装载时间

评估把从交易系统的所有历史数据装载到数据仓库的初始化时间是非常困难的，尤其是当这个时间会持续几周甚至几个月时。在整本书中，我们根据 ETL 整体目标，往往都把抽取、转换、装载过程作为一个整体来看待。然而，当估计一个很长的初始化装载时，就很有必要将其 ETL 系统分成三个独立的过程：

- 从源系统抽取数据
- 将数据转换成维度模型
- 把数据装载到数据仓库

估计抽取过程时间

令人惊讶地是，从源系统抽取数据是 ETL 过程最耗时的部分。装载数据仓库的历史数据过程从一个查询抽取了大量的数据，而且联机交易处理（OLTP）系统并未设计成可以返回如此大的数据集。然而，一次性历史数据库装载与每天的增量装载是大不一样的。

但是无论如何，交易系统没有设计成按生成事实表所需的方式提供数据。ETL 抽取过程经常需要其它的方法，例如视图、光标、存储过程和相关子查询等。在抽取开始之前估计一下它将执行多长时间是非常有必要的。但以下不利因素为直接估计这一过程带来了不便：

- 测试环境的硬件通常比生产环境要差很多。基于测试环境估计的 ETL 过程执行时间可能是完全失真的，因为测试服务器和生产服务器之间存在着巨大的差别。
- 由于抽取历史数据的 ETL 作业可能持续几天，因此在完整数据集上执行测试是不可行的。我们都遇到过在项目中抽取作业连续运行很长时间后最终失败了，而再次执行后最终还是以失败告终，这样反复几次，还没有产生任何效果的情况下数天或数周已经过去了。

为了克服处理海量数据的困难，应该把抽取过程分成两个小的部分：

- 执行响应时间：从查询开始执行到开始返回数据为止的时间。

- 数据集获取时间：第一条返回的记录到最后一条返回的记录之间的时间。

由于初始化抽取装载数据量都是非常大的，因此建议取一部分数据进行估计。大多数情况下，装载中的事实表被分区，我们建议针对整个事实表的一个样本分区获取足够的数据来生成它。使用数据分区作为样本是因为数据库分区是把数据（或多或少）分隔成大小相等的部分，这样可以提供一个有效的基准。一旦知道了抽取一个分区的时间，则乘以事实表的分区数目就可以估计出整个抽取时间。该种方法的不足是把查询响应时间和数据获取时间合在一起，这将会造成估计失真。

- 计算查询响应时间

把两个过程分隔开的最好方法是使用查询监控工具，大多数 ETL 工具在它们的应用中都有监控工具。如果正在使用 ETL 工具，请记住涉及的数据表应该在主抽取过程开始前将装载到内存中去。因此，需要将缓存处理从原始抽取过程中分离出去。

- 计算数据获取时间

一旦抽取查询开始返回数据，则开始精确记录装载一部分数据所用的时间，其中要选择对你的估计是有意义的一部分数据。比如，如果要获得 2 亿条记录数据装载，也许 100 万是一个不错的测试选择。当数据装载刚刚达到 100 万条记录时停止作业，查看所用的时间，然后乘以 200（2 亿条记录/100 万测试记录）来估计整个历史数据装载的时间。

一旦测试了整个抽取作业，则将估计的样本作业放到生产环境中去执行，以防止在较小的环境中运行可能发生的结果失真情况。

估计转换过程时间

一般人都认为操作数据可能是非常耗时的过程，但令人意外的是实际的数据转换都是在内存里以惊人的速度来完成的。相对于它的兄弟过程，抽取和装载，物理转换数据所用的时间是非常不成比例的。如果你使用了带有光标的存储过程，则考虑重新设计系统吧。根据实际情况，最好是尽可能利用 ETL 工具而不用数据库存储过程来完成转换过程。大多数 ETL 处理都是在 I/O（物理磁盘的读和

写）上耗时。如果转换耗时没有比抽取和装载过程少很多，则一定在转换逻辑中有严重的瓶颈。估计转换时间的最简单方法是得到抽取时间和装载时间估计后再运行整个过程。一旦你有了这些统计值，减去抽取和装载的时间就是转换所用的时间。

估计装载过程时间

当计算装载时间时，需要确保数据转换不会造成延迟。即使数据是以流水线操作的形式从转换到装载的，为了估计准确，最好是将转换的数据放到文本文件中。

很多因素影响装载时间。需要考虑的最重要的两个因素是索引和日志。请确保在测试阶段的环境与现有生产环境的物理条件相一致。像数据获取一样，数据装载也是按比例处理的。这意味着可以只一次装载一个样本集数据（2 亿条中的 100 万条），然后再乘以 200 来得到整个装载估计时间。

长时间运行 ETL 过程的不足

ETL 过程的目标是从源交易系统选出数据、转换它并把它装载到数据仓库。ETL 团队的目的是在面对这些复杂任务的时候设计有效的过程来应对意外情况。

水平 ETL 系统流和垂直 ETL 系统流

ETL 系统本质上按水平或垂直方式组织。在水平方式组织中，一个特定的“抽取-清洗-规范化-提交”作业从头开始运行直到完成，与其它主要的数据流没有依赖关系或很少的关系。这样，一个客户定单 ETL 作业可以完成运行，而另一个库存跟踪 ETL 作业则可能会失败，这就留给组织中的决策制定者一个不一致的和不能接受的结果。

在垂直 ETL 系统组织中，多个 ETL 作业关联在一起，每一个作业中的并行步骤各自完成运行，但要等待其它作业以到达同一执行点。还是用上面的例子，客户定单和库存跟踪就需要都完成抽取步骤后才能进行清洗、规范化和提交步骤。

在 ETL 系统中使用水平方式还是垂直方式来组织，依赖于环境中的两个主要因素：

1. 需要多个 ETL 作业并行处理的详细的数据依赖关系。（即，在库存跟踪作

业失败时，也许因为客户定单作业没有定义好的产品编码。）

2. 最终用户修改部分数据的必要性（例如，定单已经修改，但运输却要延迟一天）。

分析失败类型

不幸的是，成功执行并完成的 ETL 过程依赖于多种因素。一旦 ETL 过程用于生产，失败通常的原因都超越了过程控制本身。生产用 ETL 失败的原因主要包括：

- 网络失败
- 数据库失败
- 磁盘失败
- 内存失败
- 数据质量失败
- 末公告的系统升级

为了熟悉环境中可能对 ETL 过程带来威胁的这些因素，下面详细介绍每一个因素并提供最小化风险的建议。

网络失败

网络通常是连接数据仓库各部件的最底层结构。无论是数据库服务器还是应用服务器，都通过内部网络连接在一起。随着电缆、路由器和节点的增多，网络故障的风险是不可避免的。网络失败永远不能完全避免，但是 ETL 可以采取措施最小化网络失败带来的危害。

减少这一风险的预防措施之一是把 ETL 引擎和目标数据仓库数据库放在同一台服务器。很明显，这种方式会使 ETL 作业和最终用户查询之间的资源争夺加剧，但是就最小化网络失败可能性来看，实际上它可以减少 50%的网络传输，因为数据可以从 ETL 引擎在服务器总线内部传递到目标数据仓库。很多情况下，如果数据仓库查询一般在白天发生，而 ETL 过程主要在夜间占用大多数系统资源，这种共存就显得更有意义。

数据库失败

切记，初始化装载并不只是发生在数据仓库实施的开始阶段。如果是使用数据仓库总线架构实现的企业级数据仓库，则数据仓库的每一阶段都需要初始化装载。每一数据集市都需要在增量更新之前完成历史数据的一次性装载。物理数据库失败是常见的不可预期的宕机情况。当今可用的技术（事实上把一切都冗余了）使得不可预期的宕机可以被避免，但必须要确保拥有复杂的服务水平协议（SLA）来指定可被接受的不可预期的宕机率。

此外，数据库并不是只有宕机才被认为是出了问题。ETL 团队的目标之一是管理所需的过程来装载数据仓库，而整个过程对用户透明。如果一个 ETL 过程中某个表锁住了，或者临时空间不足了，按用户的经验这也属于失败。因此这种类型的失败和数据库物理失败同样有害，会损害数据仓库的声誉。

磁盘失败

数据仓库中的数据存储也许是整个数据仓库所有潜在失败点中最薄弱的环节。通常，在 ETL 过程中有三个磁盘组：

- 源系统磁盘：通常情况下，ETL 过程仅是从源系统磁盘读数据，失败风险很小。然而，当从系统中抽取原始历史数据时一定要注意。多年历史数据的抽取可能是一个非常大的数据集。如果运行复杂的带有多个连接和 order by/group by 子句的抽取查询，也许将超出为这种类型的操作所分配的磁盘空间。为了减少这种风险，最好与源系统的 DBA 团队一起工作，在执行历史数据装载测试过程时监控临时空间使用率。并确保在运行整个过程之前分配了足够的空间。
- 集结区：集结区是 ETL 过程的工作台。通常，它包括一组集结的文件，描述了从源到最终维度目标的数据流中不同的步骤。数据一般在每一个主要的步骤（抽取、清洗、规范化和提交准备）后都要立即被集结。对该域的读和写主要有多种原因，其中包括为了数据连续性和数据保护，以及在转换过程中保持数据独立性。数据集结域可能有源系统和数据仓库组合的那么大，然而这通常很罕见，但是切记这是有可能的，并且数据集结数据库经常在数据仓库和 DBA 团队管辖范围之外。作为预防措施，经常性地检查数据集结环境的可用空间来确保运行不至于太慢。

- 数据仓库存储：数据仓库可能成长的比初始预期要快得多。尤其是索引和临时空间总是被低估，总是超过所分配的空间。当 ETL 过程试图写入末分配的磁盘时，该过程就可能崩溃。从磁盘空间这样的错误进行恢复往往是非常困难的。为了防止超过空间，你需要向 DBA 团队撒个小慌，夸大一下估计的初始化装载容量和数据仓库估计的三个月的大小。我们过去总是把估计加倍，但是现在，经历一些事情后，我们认为三倍才是比较安全的。我们建议把估计的初始化装载乘 3 以防止潜在的灾难。请相信我们：空间总是不会浪费的。

仅仅估算一下容量和利用的存储大小是远远不够的，ETL 团队应该经常监视这些量，并在达到某一告警值时采取措施。我们见到过告警值设在容量的 90%，但是这一告警被忽视了，直到 6 个星期后，砰！！！

内存失败

在下面三种环境中内存都可能失败：

- 源系统
- Staging area
- 数据仓库

这些环境在分配的内存超载时都同样脆弱，会造成 ETL 过程的失败。内存超载并不一定会使过程崩溃，但当开始使用虚拟内存（操作系统让数据在 RAM 与磁盘间交互）时会严重降低运行速度。请确保咨询 ETL 应用供应商来获得关于针对特定历史数据装载的缓存设置方面的建议。

如果硬件坏了，则需要纠正这一问题并重新启动过程（除非 ETL 工具可以平滑地恢复）。

临时空间

临时空间是数据库中的域，在进行排序或连接数据时使用。当在一个针对交易处理的数据库环境中执行数据仓库类型的查询时，临时空间经常被耗尽。由于这一主要原因，为了避免临时空间的耗尽，历史数据应该以简单格式从源系统抽取到 staging area，然后再在专门的 staging 环境中完成转换。如果填满了所有数

据库环境中的临时空间，处理过程都将停止。

当发生了临时空间失败的情况时，DBA 团队需要分配更多的空间，然后重启处理过程。根据临时空间失败发生的位置和时间不同，总是需要清洗数据。由于数据仓库设计用来进行查询，因此临时空间应该足够大。这种问题一般都发生在源系统环境中，因为那里 ETL 团队并没做有效的控制。

数据空间

在数据仓库中，数据应该与索引分开存储，以此减少冲突并减轻管理空间的负担。估计历史数据装载大小的一种有效的方法是只装载一小部分样本数据。装载一个分区表的一个简单分区是一个不错的基准。然后就可以把已装载数据所用的空间和表的分区数做乘积，就可以得到估计的大小。第 4 章阐述了提供关于估计数据空间的更多信息。

在大量装载之前，一定要保证 data-staging 数据库和数据仓库数据库拥有足够的空间。磁盘空间失败是一种严重的失败，需要手工进行数据清洗和重启装载过程。

索引空间

索引空间的估计是非常科学的一件事，因为索引并不像数据表那样成比例地增长。我们不会太详细讲解索引空间的问题，它的计算非常复杂，数据仓库架构、数据建模者和 DBA 应该在装载开始之前合理创建索引空间。通常分配的索引空间至少应该与基础表大小差不多。

当装载历史数据时，装载之前一定要把目标表的索引删除掉，并在装载完成后重建索引。通过删除和重建索引，不仅可以提高性能，还可以将索引空间不足的情况与装载失败隔离开来。一旦完成了表装载，总是可以为索引分配更多的空间，并在不影响数据表的情况下重建索引。

文本文件空间

由于数据库中为数据分配的空间可能不够，因此文件系统中的空间也必须进行监视以防止超过文本文件分配空间的限制。幸运的是，staging area 中需要的空间是按 ETL 团队的需求分配的，因此如果按照第 4 章中对文件系统需求估计的

建议，对于处理历史数据装载将是安全的。一些功能较强的 ETL 工具包括检查点功能，可以保证过程中到达某一特点检查点的任何一条记录都被安全地写到磁盘中。但是，这些检查点文件写到文件系统，可能正好成为增加文件空间的罪魁祸首。检查点、缓存、哈希表、平面文件、临时空间或者任何其它 data –staging 文件都会填满磁盘空间。如果一旦由于任何一种文件超过了分配的空间造成过程崩溃，需要增加空空间，都建议从头开始重新处理，而不是试图去挽救已经处理的数据。

数据质量失败

生产环境中的数据质量失败是一种很容易被检测到的灾难性管理失败，例如字段丢失或者违反参考完整性，或者是在长时间运行过程中逐渐达到的数据质量告警的阈值。生产环境中数据质量失败应该是异常事件，往往需要专家参与。也许作业可以在已知异常数据的情况下完成运行，或者作业需要停止并修复源数据。

末声明的系统升级

也许末声明的系统升级唯一的好处是它造成的错误通常是彻底显而易见的。在这个过程中，ETL 作业停止了，一般在这种混乱状态下不会进行任何简单的修复。如果是这样，则必须要恢复到系统升级之前。这种情况对于很多其它的软件修改变化都是一样的。对于关键系统，在安装升级版本到生产环境 ETL 系统之前，必须要在测试系统环境中进行充分的升级测试。

恢复问题

当过程失败时，ETL 团队下意识的反应是试图挽救失败点之前所处理的内容。如果足够幸运并使用了具有检查点功能的 ETL 工具，则可以重新启动处理过程，并将失败时丢失的数据找回来。虽然有产品供应商的保证，我们还是对检查点技术的可靠性表示怀疑。如果处理在中间过程中失败，最好的方法是清除数据并从头开始处理。可以把处理过程分成若干个子过程来使恢复工作变得简单和有效，这样就可以不用重新处理整个历史数据装载，而只处理数据的某个子集就可以了。

最小化装载失败风险

以下是处理历史数据的几个基本规则：

- 分隔过程：使用日期或范围或自然键来把过程分隔成更小的可管理的单元。当失败发生时，只有这部分数据需要重新装载。
- 利用恢复点：为了安全，在每个主要的中间处理（例如整体抽取，关键的转换，或者分配了代理键等）之后将数据写入文本文件。
- 并行装载：不仅是数据装载，也包括 ETL 的每个组件都应该并行运行来减少处理数据所消耗的时间。
- 维护元数据：操作元数据（例如最后装载日期或者装载记录数）是检测失败发生时 ETL 每个部件状态的关键。

9.3 清除历史数据

当设计任何数据库应用系统时，都会创建一个矩阵表格来跟踪插入、修改、删除和选择数据的处理过程。这个矩阵表通常指的是 CRUD 矩阵表（创建、读取、修改和删除）。CRUD 矩阵表保证了每一个实体都有一个以上四种方式之一的数据操作过程。当开发应用软件时，矩阵表中的 D 是非常常见的，它的意思是这里的数据可以被修改和读取，但不能执行常规的处理来删除它。当没有开发常规的处理来清除历史数据时，通常会导致两件事情：会在系统上运行后台脚本来删除历史数据，或者记录不确定地留在系统中。可以想像，两种方案都不适合于数据仓库。

随着每个主题域计划完成，必须也有相应的清除处理过程。如果数据量相对很小，并且末来 10 年或更长时间的数据量不太会影响性能，则 ETL 并不需要马上开发。然而，清除策略元数据仍必须和初始化实施一起被收集和发布。

应该由 DBA 而不是 ETL 团队来完成数据仓库数据的存档工作。然而，对于从数据仓库中永久性地删除数据则必须由 ETL 团队来完成。关于所删除数据的业务规则必须由经过充分测试和质量保证的 ETL 过程来执行。

9.4 监控 ETL 系统

依赖于数据仓库的业务数据按照经过事先商定的频率（或持续不断的）来刷新。完成这个任务过程中的失败会导致数据仓库的可靠性和可信赖性被怀疑。因此，数据仓库如果没有有效的和持续的数据反馈就不能成功。ETL 团队必须监控和评估 ETL 作业来确保它们有效率地运行，以及数据仓库是否以有效的方式装载。ETL 监控使得处理过程的多个方面需要考虑。ETL 系统范围之外的资源如硬件和基础部件的管理和使用，以及源和目标环境，都是 ETL 系统的整体效率中的关键。在这里我们介绍几个 ETL 性能指标，来告诉你过程执行的好还是不好。这些指标是操作型元数据的一部分，应该存储在资料库中，以便被整个 ETL 团队随时分析。

度量 ETL 特定性能指标

对于那些负责系统或数据库管理的人来说，他们比较了解那些在环境中捕获的特定的度量来保证适当的性能。就像所期望的那样，ETL 系统也有自己的性能指标集。ETL 指标是专门用来描述实际数据的真实移动和管理情况的，它们是典型性能指标之下的指标，这里“之下”指的是不在操作系统一级或硬件资源一级来度量，而是在 ETL 处理本身过程中。

ETL 效率的度量大多数都用来表示处理数据所用的实际时间。请记住：ETL 系统的目标除了创建高质量的信息外，还有在分配的装载窗口内装载数据仓库。但是如果一个作业需要 20 分钟时间来完成，这是好还是不好呢？确实没办法知道，除非知道在这段时间里有多少条记录被处理。例如，如果在 20 分钟时间里处理了 5000 万条记录，则这是相当惊人的效率。而如果只处理了 100 条记录，显然效率就太差了。下面是 ETL 相关的一些度量，在研究装载性能方面非常有用。

- 耗时（秒）：这是其它计算的最直接的基础数据。耗时指的是 ETL 过程从开始到结束所用的时间（秒）。例如，如果一个过程从 4:00 a.m.开始并到 4:15 a.m.结束，则用时是 900 秒。
- 每秒处理的记录数：这相当于每秒装载的记录数，除非当源数据比目标数据要大的时候（聚合装载），这时就等于每秒读取的记录数。一个每秒

记录数的例子是在 15 分钟时间里处理了 1000000 行记录，则为 (1000000/(15*60))=1111.11 条记录/秒。

- 每秒读取的记录数：从源系统用 SQL 读取数据结果的记录数，再除以所用时间（秒）。数据再转给 ETL 流程中下游转换过程，这里根据处理过程的不同记录条数可能增加也可能减少。
- 每秒写入记录数：转换之后提交到目标表的记录数，再除以所用时间（秒）。对于有多个目标表的情况，指的是插入到所有表的记录数总合再除以所用时间（秒）。写入的记录数可能多于也可能少于读取的记录数。
- 吞吐量：吞吐量指每秒处理的记录数乘以每条记录的字节数。吞吐量和其它所有性能指标一起，可以用来作为提高性能的依据。

大多数主要的 ETL 工具都提高了必需的指标来度量 ETL 性能。你可以利用 ETL 系统来为消耗了比经验值更多或更少的时间的 ETL 作业触发告警。

在一个处理流程中，当一个处理部件的吞吐量不能处理前一个部件的输出时，就会产生瓶颈。例如，一个批装载可以按每秒 1000 条记录的速度向磁盘写数据，而磁盘只能写入每秒 800 条记录，因此在磁盘这个部件上就发生了吞吐量瓶颈。结果是，整个处理过程只能以最慢的部件的速度来处理。

度量基础性能指标

下一个要检查的部件是 data-staging area 架构。不同的监控软件包都提供了多个可用的处理度量，或者以手工编码 ETL 方式写入日志。只有少数供应商方案本身就提供了直接的影响 ETL 性能的指标。关于 ETL 性能重要的度量指标只要在过程运行的监控过程中才能获取。处理过程中提供的直接指示可能是瓶颈的度量包括：

- CPU 利用率
- 内存分析
- 服务冲突

实质上，网络流量和其它已知的基础部件性能指标可能影响 ETL 的性能。不幸的是，这些指标不是很稳定和具有一致性，因此可靠性较差。此外，网络流量

本身很难确定或在测试环境中复现。如果你怀疑有网络问题，请联系网络管理员寻求帮助。

CPU 利用率

ETL 过程运行在其服务器上的中央处理单元（CPU）上。CPU 是处理器或芯片，实际中用于操作计算机所需的计算功能，使软件运转，以及完成 ETL 目标。大多数 ETL 服务器包含多个处理器来处理抽取、转换和装载数据到数据仓库所需的大量计算功能。在 ETL 工具中一般是不能得到 CPU 利用率报告的，因为这超出了 ETL 系统的范围。然而，如果在 UNIX 上运行处理过程，可以使用 SAR －u 命令来系统中每个处理器的使用列表。

在基于 Windows 的操作系统中，可以使用图形化界面性能监视器，在其中可以添加新的计数器到目前可用的性能日志中。在 Windows XP 中，在控制面板的管理工具中可以找到性能监视器。为了添加新的计数器，打开性能监视器，右键点击系统监视细节面板，然后点击添加计数器，在这里可以选择处理器作为性能对象，然后再选择相应的计数器。性能监视器将创建一个日志文件来捕获关于 CPU 利用率的统计以便进行分析。

如果发现在 ETL 过程中 CPU 经常达到能力上限，则需要添加处理器。这些 CPU 监视工具还帮助检查是否 ETL 过程在所有可用的处理器上平均分配负载。

内存分配

RAM 可以在多种不同的位置分配给 ETL 过程，但总的来说，服务器上的内存必须物理可用才行。如果购买了 ETL 工具，供应商应该提供硬件配置说明和工具建议和处理现有装载数据量所需的 RAM 数。一旦在服务器上安装了 RAM，内存必须分配给处理过程。对于大多数 ETL 工具，内存使用可以在作业级或者批处理级设置。

为了实现最好的效率，对每一 ETL 过程必须分析适当的内存。如果 ETL 过程不断地读和写磁盘，而不是在内存处理数据，则这种方案将会非常低效。ETL 过程分配的适合的内存比其它设置更能影响转换性能，因此一定要确保设置是正确的。

ETL 工具可以告诉你每个过程分配了多少内存，过程实际使用了多少，以及使用了多少虚拟内存（磁盘缓存），在 POC 测试过程中要确保由 ETL 工具提供了这些操作元数据。如果应该在 RAM 中的大量数据正在写入磁盘，你应该为过程分配更多的内存或者给你的服务器添加更多的物理内存。

有些 ETL 工具使得内存管理对开发团队完全透明，而其它的则可能需要手工配置特定项。主流 ETL 工具中的内存设置包括：

- 共享内存：当 ETL 引擎读和写数据时，它们使用了内存中一块专用区域叫做共享内存。共享内存是数据在进入或退出 ETL 过程的物理转换步骤排队等待的位置。如果作业没有足够的共享内存，就会产生很多的磁盘缓存。相反地，如果分配了过多的共享内存，则会保留大量不必要的 RAM，其它过程也不能使用。ETL 供应商应该根据处理的数据量大小提供如何计算最优的共享内存设置的指导。有些 ETL 引擎可能会试图动态管理共享内存，寻找一种工具使你可以针对特定情况不考虑系统分配的设置，这种情况下引擎也许不能进行估算。
- 缓存块大小：缓存块设置是分配性能相关设置时需要考虑的关键因素。缓存块大小适合设置依赖于转换流程中数据的行大小。如果工具需要或者允许自定义调整缓存块大小，ETL 供应商可以提供优化设置的计算建议。

当程序需要比物理可用内存更多的内存时，操作系统（或应用系统）把不能放到内存的数据写到磁盘。由于需要处理大量的数据，程序必须能够读/写磁盘而不是使用 RAM。虚拟内存通常是指页面交换，因为内存是按页面存储的，当需要更多的页面时，就会在磁盘和 RAM 间交换。页面交换是性能杀手，应该在 ETL 过程中尽量避免。如果检测到连续不断的页面交换，就应该为 ETL 服务器和 ETL 过程添加更多的 RAM 了。

如果选择手工编码方式实现 ETL，则可以用 vmstat 命令行手工进行内存使用监视。Vmstat 命令报告了虚拟和实际内存使用情况，以及页面活动和磁盘操作情况。

服务器争夺

另一个潜在的性能杀手是服务器争夺。当多个处理过程试图使用同一资源时就发生了冲突。可能会遇到内存争夺、磁盘访问争夺或者数据访问争夺。最常见的冲突是两个 ETL 过程试图访问同一数据。ETL 过程可能会引起死锁。当过程 A 试图把过程 B 关在外面，而过程 B 也试图把过程 A 关在外面时就发生了死锁，系统宕机。通常，DBMS 在管理数据访问争夺方面做的很好，但在 ETL 开发者开发过程中的某些点上还是会发生数据访问争夺。当 ETL 过程并发而不是并行运行时，ETL 系统非常容易发生服务争夺。当这种争夺发生时，ETL 过程不断地争抢资源，从而产生冲突。最好的防卫方式是避免并发过程，除非每一过程都有专门的过程流并有指定的数据区。

内存争夺

当多个应用或过程在同一服务器上运行时，它们都需要物理 RAM 来进行操作。不幸的是，RAM 是有限的资源，每个过程一定会争夺它。如果并发执行 ETL 过程，则可能会产生内存争夺问题。当并发执行过程时，作业级的内存分配设置就成为关键。每一 ETL 产品为了减少内存争夺都有其自己的建议。基本规则是，为小作业分配最小的内存，把空间留给大作业，例如第 2 类缓慢变化维、桥接表或者事实表。ETL 工具应该能在其工具内部管理内存来避免内存争夺。任何情况下，工具应该提供操作型元数据来展示内存争夺情况。如果工具不能实现，UNIX 的 SAR 命令可以用来帮助检测内存争夺情况。SAR 命令对于检测那些和 ETL 工具一起运行并竞争同一 RAM 的过程的内存使用情况非常有用。如果可能的话（预算允许），一定要确保 ETL 引擎是数据仓库装载过程中服务器上唯一运行的进程。如果 ETL 进程没有在规定的装载窗口内完成，则其效率就值得怀疑了。有效的监视经常可以发现大多数装载延迟并不是 ETL 无效率的结果，而是由于外部进程与 ETL 运行在同一时间，并且一起竞争服务器资源。

磁盘争夺

大多数磁盘都有在给定时间内读写的数据量和访问连接数的限制。当达到限制值时，ETL 过程将排队等待对磁盘的访问。如果在同一磁盘的同一数据文件进行多个表的数据并发装载，则会发生忙点现象。忙点指的是在磁盘上某个区域重复读写访问。如果使用 Oracle，可以使用下面的 SQL 来检测源数据库、staging

数据库或目标数据库的忙点：

```
select d.name datafile_name, f.phyrds reads_count, f.phywrts
writes_count
from v$datafile d, v$filestat f
where f.file# = d.file#
order by greatest(f.phyrds, f.phywrts) desc
```

该查询按照读和写的数目降序排列查询的结果，读写最多的数据文件排在最上面。如果有一些数据文件与其它部分不成比例，则需要重新配置 staging 表的物理属性，以便分布更均匀一些。

磁盘争夺也发生在数据库外部。ETL 引擎使用临时区域和隐含创建的文件来保持过渡过程数据。此外，开发者也公开的在文件系统创建 staging 表、配置和参数文件。如果要从操作系统角度获取磁盘活动信息，可以使用 UNIX 命令行 IOSTAT。IOSTAT 命令列出了每一磁盘及其相关的信息：

- 磁盘名
- 每秒读次数
- 每秒写次数
- 每秒读字节数（KB）
- 每秒写字节数（KB）
- 服务的交易等待平均数（队列长度）
- 正在服务的活动交易平均数（从队列中删除但还没有完成）
- 平均服务时间（毫秒）
- 有等待服务的交易的时间百分比（非空队列）
- 磁盘忙时百分比（处理交易）

关于如何解决磁盘争夺的信息在本章后面提供。

数据库争夺

如果 ETL 过程试图在同一时间更新同一表的记录，就会经常发生数据库争夺问题。本来，管理数据库争夺是 DBMS 的作业，但是当进程争夺同一资源时就会发生死锁，使得进程永久等待下去。请参考专门的 DBMS 参考手册或者联系本地 DBA 获得检测数据库争夺的最佳步骤。

处理器争夺

有时，如果硬件没有配置成并行运行，当试图在软件一级并行处理时就会发生问题。当有多个进程（超过可用的进程数）试图在同一时间运行时，就会使 CPU 满负荷并引起严重的性能问题。可以使用 UNIX 的 SAR 命令或者 Windows 的 PerfMon 命令来获取 CPU 使用率的统计。

度量数据仓库利用率来帮助管理 ETL 过程

在本章前面提到的供应链例子中，你会注意我们确定了四个关键部件，用来帮助把原始数据转换成为用户使用的有用的格式的结果。至此，我们已经描述了如何在 ETL 系统范围内以及 ETL 环境的硬件和基础架构内监控活动。现在我们来看一看数据仓库展示层的重要度量指标。

这里的度量并不直接影响 ETL 系统，但是捕获和分析它们仍很重要，因为它们对装载过程有影响。我们希望后面列表中的度量已经被数据仓库团队收集好，用来帮助管理他们的用户经验以及删剪掉数据仓库中存储的不再使用的数据。

ETL 团队应该充分利用数据仓库使用报告，从中寻找重新安排装载调度、修改装载频率或者清除装载睡眠表的维护作业等方面的机会。例如，如果一个表只在一个月的第一天被访问，它就不应该天天更新。另一个有功效的方法是分析索引使用率。ETL 过程的很大部分工作是在每次装载后重建数据仓库中的索引。如果使用率分析结果显示有些索引从不使用，则它的重建过程就应该从 ETL 流程中删除。支持 ETL 作业管理的利用率度量包括：

- 表利用率：表利用率报告的内容可能不尽相同，但是一个有用的报告应该包括以下内容：数据表列表、第一次和最近访问数据表的时间、和该数据表相关的查询数以及查询该表的不同用户数。一个月才使用一次的表应该从按天装载进程中删除并转向按月频率装载进程中。对于除了刷新以外不断被利用的表，应该使用一些高可用性技术。高使用率表一般

在后台都装载了一个复本，一旦装载完成，两个相同结构的表就改变名称。该技术使得在数据刷新过程使表仍在线可用。

- 索引利用率：索引是数据仓库表性能提高的关键，但会给 ETL 带来负担。因为很多情况下，每一次数据装载都会删除并重建索引。当数据仓库架构为展现层创建维度结构时，一般都倾向于尽可能多的对列建索引，以防止用户较低的执行效率。但事实情况是很多索引列从没有约束关系，索引也从末被使用。索引利用率报告能发现那些睡眠索引，数据仓库团队可以合理地处理它们。
- 聚合表利用率：聚合表通常和索引在同一时期创建，当需要时就创建它。但和索引的情况一下，有些聚合表创建了但从没被使用过。或者随着时间的迁移，它们越来越没有用处，最终进入睡眠状态。聚合表利用率报告可以发现那些不再有用而应该被删除的表。
- 睡眠数据：睡眠数据报告总是会受关注，因为数据仓库是按照用户交流的结果创建的，这些交流用来找到那些执行完成作业所需的分析的数据元素。然而由 ETL 每天刷新的表处于无用状态是不可避免的。即使表有用，有些列也可能从没被选择使用过。我们总是很关心事实表列利用率，因为它对于发现最复杂的导出度量的利用率（由于它们的定义没有满足用户的需求）非常有用。数据睡眠报告可以帮助 ETL 团队确定那些从未使用的度量和维度属性。

有多种方法收集数据仓库利用率的统计值。有些数据库管理系统本身提供使用率信息。然而，一定要测试打开和关闭使用率报告功能对性能的影响，它可能会影响查询响应时间或者 ETL 装载时间。一种不影响性能的跟踪利用率统计的方法是使用像 Teleran Technologies（www.teleran.com）这样的中间件。这些数据仓库监控工具在数据库之外的网络包一级捕获 SQL 语句和数据。我们确信还有其它工具提供数据库利用率统计，可以在 www.google.com 中试着用 data warehouse usage tracking 来查找这一领域的供应商列表。

9.5ETL 过程调优

为了更好的理解如何优化 ETL 进程，必须要熟悉数据库是如何工作的。大多数数据库管理系统可用的功能不可以在数据仓库环境中使用，而有些几乎从不在交易系统中使用的特性在数据仓库中不仅有用，而且是功能首选。很多设计决策是基于开发的进程所迁移的数据量。例如，如果有一个非常小且数据稳定的维表，则使用 update 语句增量更新维表现有数据是可行的。但是如果维表有 2000 万行记录之大，且数据变化频繁，则删除并批量装载该表也许更有效一些。可以使用数据仓库架构师或项目经理创建的报告来了解初始化阶段有多少数据将被装载，以及实施 6 个月之后计划的增长情况，从而计划可能的容量，以及确定 ETL 系统的扩展性，并在之后一直跟踪实际的增长情况。

后面部分重点介绍在可控的 ETL 环境中不需要而应该避免的功能，并提供实施过程中更快且同样效果的解决方案。

了解数据库

在创建关系型数据库之前，数据存储在平面文件中。这些文件往往数据质量较差，主要表现在重复的分组和元素，没有主键，以及表间没有强制关系等等。整个数据库中的一切都是重复的，此外，在数据库一级不存在数据验证。当时，数据库只是一个没有关联的文件集合。简单的说，当时的情况简直一团糟。

在 1970 年，E.F.Codd 发明了关系代数，这成为关系型数据库系统的设计基础。关系型数据库提供了以下功能：强制参考完整性、数据唯一、主键、约束检查、外键等等。结果是数据清晰可靠，但是降低了操作速度。关系型数据库的每一特性对于交易处理都意义重大，不再需要程序额外的处理来执行后台错误检查和控制。在本节，我们讨论作为 ETL 团队中一员经常会遇到的不同的数据库特性，并提供克服数据库问题的建议。但是，这并不是什么 DBA 培训。事实上，本节的内容并不能使你成为数据库管理员。本章的目标是帮助 ETL 团队以及 DBA 来了解优化 ETL 过程所需要的特定考虑。优化 ETL 相关的大多数工作都在数据库之外。本章的内容用来深入了解如何让数据库处理大量的数据，以及提供关于提高处理速度的建议和技巧。

插入、更新、删除

数据操作语言（DML）有四个主要的动作：select, insert, update 和 delete。四个 DML 操作都可以用来在不同的数据库操作数据。切记，DBMS 主要设计用来避免交易失败。因此，作为预防措施，理论上每种 DBMS 都维护一个回滚日志。回滚日志记录了 DML 操作并提供了 DML 提交后结果的取消机制。对于中间交易失败的情况，DBMS 自动回滚完成了一半的交易结果，使数据回到交易开始前的状态。

日志的作用

每一种类型的 DML 都会以不同方式影响回滚日志。Select 语句不会写入日志，因为它们不会改变现有数据。在大多数数据库中，insert 语句都会在随机输入数据或者交易中间失败时写入日志，DBMS 可以简单的回滚记录，而不是删除或清除它。Udpate 和 delete 都需要写入回滚日志。Delete 需要较大的成本，因为它要在删除发生之前保存旧记录，update 需要所有 DML 语句中最大的成本，因此处理起来非常慢。

索引的作用

使用数据仓库是因为它比交易系统更快更可靠。数据仓库提供的速度优势是由于以下特性带来的：

- 维度模型有意地建立维表和事实表索引
- 良好的索引策略
- 物理存储聚合记录
- 并行查询机制

设计技巧和维度数据模型的优势在整本书都有涉及。在本节，我们来讨论一下索引。

索引是数据仓库中查询响应时间的基石。数据库中每一次查询都至少要使用一个索引。不幸的是，由于索引帮助数据仓库用户查询，因此 ETL 团队就有很多工作量是在 ETL 进程中管理现有索引。索引管理是大多数 ETL 进程中的必要部分。

在进入数据仓库装载阶段索引管理的技巧之前，首先回顾一下大多数数据库可用的不同的索引类型。一般有两种主要类型的索引，理解这两种类型之间的区别非常重要：

- B 树索引：B 树或者平衡树索引在反向的树结构中保存键值以及指针。B 树索引对于高 cardinality 的列比较理想，其中高 cardinality 指的是不同值的数量。反向树结构利用一种非常高效的分隔筛选技术来找到某个特定的值（或值范围）。B 树索引对于常规的查询非常有用，但是对于像数据仓库这样的 ad-hoc 环境支持起来就不太灵活了。B 树索引不灵活是由于你不能临时动态地组合索引列来创建复合索引，以此来解决新的、不可预期的查询。所有索引必须在之前创建同，因此 DBA 必须要猜测哪些列将包含在查询中。此外，列放置的顺序位置也决定了它们能否被使用。结果是 DBA 团队必须创建很多很多的复合 B 树索引以及很多不同顺序的同一批列的组合。
- 位图索引：位图索引与 B 树索引功能完全不同。位图索引更适于低 cardinality 的列。多个单列位图索引可以动态关联在一起，以此创建所需的组合索引，从而支持 ad-hoc 查询。由于它们的灵活性，在数据仓库中事实表上每一个代理键上创建单列位图索引就是很通用的方法。

现在我们回头看看索引如何影响 DML 操作。B 树索引的每一个条目都包含了一个且仅有一个行号，它指向回到基础表的相应记录。相反，位图索引在索引中为每个值包括了一个行号的范围。如果值被修改，则该行号范围对应的每一条记录就被锁住。最终，每当记录修改时，为了管理所有被锁住的记录，就给了数据库相当大的压力。不幸的是，事实表通常都有很多位图索引，大量的数据更新会带来非常大的性能下降。

因此，我们建议在对事实表进行数据操作前删除所有的位图索引。另外最好把事实表分区，这样就可以只删除当前分区的本地索引（假设表是按照日期键分区的）。B 树索引在 ETL 过程执行前也需要删除。统计显示，在大多数情况下，删除数据仓库的索引后，装载表并创建索引比带着索引装载表要快得多。

约束关系和外键

关系型 DBMS 中的外键强制了表间数据的完整性。例如，只有在定单状态表中存在有效值的情况下才可以输入定单状态。但是在数据仓库中，交易已经发生并且已经在源系统中验证过，因此所有的外键和约束关系都应该禁止掉，尤其是在 ETL 过程中。对于 DBA 团队来说，删除外键可能是很大的挑战。因此必须在整个 ETL 过程中与 DBA 团队一起工作，并且要解释清楚，事实表记录如果没有来自相关维表的代理键就不能简单存在。而且，必须指出的是，维表的原始键用来确保记录不会重复插入。一旦 DBA 团队明白了 ETL 是真正可控制可管理的环境，他们就会意识到数据库约束和外键是多余的，这样只会降低 ETL 处理的效率，而不会带来任何好处。

把数据装载到数据仓库最快的方式是用数据库的批量装载，下面四个步骤用于数据库准备：

- 尽可能消除 DML 语句
- 禁止回滚日志功能
- 删除现有索引
- 消除数据库外键和约束

一旦完成了上述四个步骤，就做好了使用数据库批量装载工具的准备。在第 7 章里有关于使用批量装载的向导。

9.6ETL 系统安全

在数据集结区中的数据库安全比在数据仓库展现数据库里要容易实现得多。一般来说，除了 ETL 引擎和程序，没人能够读写 data-staging area。大多数数据库使用角色和用户在数据库一级强制安全性。角色机制是这样的，它使安全管理员可以把许多用户组成一组，而这组用户拥有相同的数据库访问权限。每一用户都有其自己的用户 ID 和验证机制。当用户通过系统验证后，他就被授予其相关的角色。

如果没有角色，安全管理员就必须对每个用户分别授予适当的权限。用户是使用数据库的个体。

通常，需要创建一个具有下列权限的数据仓库管理角色：

- Select, Insert, Update 和 Delete 所有对象的权限
- TRUNCATE TABLE 权限
- 使用批装载权限
- 删除和创建索引权限

对于高度敏感数据，例如赔偿率或销售潜在客户，这些数据应该在 ETL 团队抽取之前由源系统加密。通常，列一级的安全通过使用数据库视图来控制。视图是基于表之上的对象，可以隐藏敏感列数据。然而，ETL 团队必须能够从源系统选择数据，以及能够在数据仓库上执行任意 DML 语句。所以，视图就不是隐藏敏感数据的有效机制了。此外，ETL 团队不应该负责为敏感数据加密，这个工作应该是源系统安全管理员的责任。

开发环境安全

在开发环境中，ETL 团队中每个人都授予了 DWETL 角色权限（所有对象的 DML 和 TRUNCATE 权限），也正是在这里创建所有的 staging 表。虽然 data-staging area 被 ETL 团队拥有，但有时表创建权限却被数据仓库架构师或在 BA 控制。有些时候，ETL 架构师在没有授权情况下也具有在 data-staging area 创建表的权限。

此外，任何 ETL 团队成员可以添加、删除或修改文件系统中指定目录中的文件。ETL 团队之外没有人具有访问 data-staging 环境的权限。

生产环境安全

ETL 团队通常只具有生产环境的读权限。有时，在安全度要求非常高的环境中，例如银行，可能就根本没有访问权限。生产环境存在 DWETL 角色，但是只有 ETL 引擎使用的用户 ID 和口令在生产环境中可以被创建。如果为了某种非常特殊的原因，ETL 团队成员必须要有生产环境的写权限，例如修改一个只有在生产环境中存在的错误，则应该只临时授予权限，在修复完成后马上回收权限。

FTP 的不足

必须把所有人挡在 FTP 之外，只有 FTP 进程可以在指定目录中进行写入或删

除操作。此外，只有 ETL 引擎可以读这个目录。如果处在某种紧急情况下，则应该临时授予权限给预先确定的管理员，并在问题解决后马上回收。

加密/解密

最大的性能障碍是磁盘的读写（I/O）。然而，为了安全必须对流转的数据加密。通常，流转的数据安全包括下面几个步骤：

- 加密数据并存储在磁盘上
- 读取并在网络上传输加密数据
- 存储加密数据在 data-staging 服务器上
- 解密数据并存储在磁盘上
- 转换并加载解密数据到数据仓库

也可以把上面步骤减少为 3 步，当源数据读入内存时进行加密，然后传输加密数据，最后在进入转换和装载进程时解密数据。可以看到，第二种方案从数据读取直到装载入数据仓库都没有涉及磁盘。这种过程叫做流中解密。有些 ETL 工具支持流中解密功能。如果需要在流过程中加密或解密来提高性能，一定要选择支持该功能的工具；否则你就需要在 Java 中写自己的小程序了。对于 Java 小程序的情况，请确保 ETL 工具至少可以嵌入外部进程（例如流中解密小程序），从而不需要过多的磁盘 I/O。

9.7 短期归档和恢复

有多个原因需要在 ETL 系统中保存多个 data-staging 结果。在本书中，我们确定了需要短期重启容量，当没有合适变更数据捕获系统时把按天抽取数据进行对比来检测差别，以及法律和财务审计方面需求。所有这些归档和恢复情景都应该是 IT 人员熟悉的挑战。备份数据到当前介质并确保它能恢复，确保拥有可信赖的审计轨迹来标明所有对数据的访问和修改，确保数据物理上是安全的，并按保护在线数据的方式保护存档数据。

但是如果数据要保存多年该是什么样子呢？

9.8 长期归档和恢复

数据仓库管理员的重要目标之一是保存历史数据。我们以多种方式保存企业历史信息，成为公司的档案保管人。我们并不经常许诺使所有历史数据保持在线可用，但我们经常声明为了安全要将其保存在某处。当然，为了安全保存它意味着当有人有兴趣查看这些数据时能够再次恢复这些数据。

大多数数据仓库管理员都忙于以下事情：建设数据仓库，避免烟道型数据集市，配合新的数据库技术，以及满足对 Web 的大量需求。这样我们的任务就变成了存档，把数据备份到磁带上，然后就忘记了这些磁带的存在。或者也许我们还把数据追加到最初的事实表上，但我们并没有真正想过这些旧数据能做什么。

但是计算机业界有一种普遍的观点是，保存数字信息并不算完，这就带来了一个严重和难以解决的问题：

数据仓库还需要保持旧数据吗？

大多数数据仓库管理员都是被市场这样的部门的紧急需求驱动的，都只是为了战术的考虑。很少有市场部门会关心超过 3 年以上的旧数据，因为我们的产品和市场变化的如此之快。因此，就只需思考市场客户，而把不再满足需求的数据放弃掉。

但是稍微思考一下我们就会意识到，我们仍然需要大量的数据仓库中的一些数据，这些数据绝对必须要保存。这些数据包括：

- 针对法律、财务和税收而用的详细销售记录
- 长期跟踪具有战略意义的趋势调查数据
- 政府调整或与跟踪相一致所需的所有记录
- 必须保存 100 年以上的医疗记录
- 支持专利声明的临床测试和试验结果
- 有毒废弃物处理文档、燃料传输和安全检查
- 对任何人、任何时候可能拥有历史值的所有其它数据

面对这个列表，我们必须承认需要一个计划来获取所有这些类型的 5 年、10 年甚至 50 年的数据。我们都清楚这可能将是一个很大的挑战。磁带以什么方式持续多久来保存？CD-ROM 或 DVD 能解决问题吗？将来还能读取这些格式吗？几年前我们还有一些 8 寸软盘，但现在绝对不可恢复也没有用了。突然，这听起来像是一个非常艰难的项目。

介质、格式、软件和硬件

当我们开始真正考虑长期保存数字数据时，我们发现出了大麻烦了。先让我们从存储介质开始。关于物理介质实际的寿命有很多的不同意见，像磁带和 CD-ROM 光盘，寿命估计在 5 年到数十年之间。但是，当然，我们的介质可能没有档案质量，它们可能没有以一种最优的方式存储和处理。我们必须平衡供应商的乐观许诺与相关专家的实际评价之间的差异，专家认为今天我们拥有的大多数磁带或物理介质上的数据在超过 10 年以后都会面临可用性问题。然而，所有关于物理介质保存期的争论与关于格式、软件和硬件的争论比较起来显得就不足为重了。所有的数据对象在物理介质上都是按现在的格式来编码。所有的一切，从介质的密度，到目录的编排，再到最终更高级别的特定应用的数据编码，都会像堆叠起来的纸牌一样最终倒掉。拿 8 寸软盘举例，靠什么来读取嵌入的数据呢？可能需要支持 8 寸软盘驱动器的硬件配置，还需要 8 寸驱动器的软件驱动程序，以及最初写数据到文件的应用程序。

过时的和废弃的格式

在数字保护主义者的词典里，过时的格式就不再支持了，但是仍有现存的硬件和软件可以读取和显示这些原始格式的数据内容。陈旧的格式传递给下层领域。我们关心的 8 寸软盘是一种陈旧的格式，再也不会恢复它上面的数据。以 A 类线形文字著名的腓尼基人文字系统也是一种陈旧的格式，很明显它永远地消失了。我们的软盘或许要比 A 类线形文字的解晰要简单一些。

硬拷贝、标准和博物馆

针对恢复不同格式的旧数据这项困难工作，我们设计了多种方案。一种简单的方案是通过硬拷贝还原一切，也就是说，把所有数据打印到纸上。当然，这样做将屏蔽所有数据格式、软件和硬件的问题。如果只有非常少量的数据，这是一

种不错的选择，总比丢失数据要好多了，但也存在多个致命的缺点。对于现在来说，拷贝到纸上并没有可扩展性。打印一个 GB 的 ASCII 字符，如果每页打印 4000 个字符，则需要 250000 页，一 TB 数据将需要 250000000 页！请记住我们不能采取欺骗手段，将纸张放到 CD-ROM 或磁带上，因为这将再次引入数字格式问题。最后，我们也许将认真折衷考虑数据结构、用户界面和系统最初认为的行为来呈现和解释数据。大多数情况下，纸质备份将破坏数据的可用性。

第二种方案是建立数据展现和存储的标准，这将保证一切数据都能以永远可读的格式来表示。在数据仓库中，接近这一标准的唯一数据是以 ANSI 标准格式存储的关系数据。但是绝大多数关系型数据库的实现都对数据类型、SQL 语法和相关元数据做了很大的扩展，以此来提供所需的功能。当我们把带有应用系统和元数据的整个数据库卸载到磁带上时，不管它是来自 Oracle 还是 DB2，我们并不能确信 30 年或 50 年之后仍能使用这些数据。狭义 ANSI 标准 RDBMS 定义之外的其它数据更是纷繁多样，市场上还没有任何一种机制，能够把所有可能的 OLAP 数据存储机制集成到单一的物理标准，以此来保证到标准格式间的无损转换。

最后一种经典的方案是支持“博物馆”，这里保留了所有硬件、操作系统和应用软件的旧版本，这样旧数据就可以被读取了。该方案至少理解了问题的核心，为了解释旧数据就必须真正拥有旧软件。但是博物馆思想也不具有扩展性，也不能深入解决问题。我们如何让 Digital Data Whack 9000 工作 50 年？当最后一个人死了时会怎样？当一个使用旧数据的人迁移数据到流行的 DVD ROM 介质时，Digital Data Whack 9000 界面如何面对 DVD？有人会为过时的旧机器写新的驱动吗？也许它还是个 8 位总线。

刷新、迁移、模拟和封装

有些专家建议说，IT 机构应该周期性地从旧介质物理移动数据到新介质，以此刷新数据的存储。关于刷新的更进一步是迁移，这里数据不仅是物理转换，而且还重新格式化，以便被现有的应用系统读取。刷新和迁移确实解决了一些短期保存危机问题，因为如果你成功完成了刷新和迁移，就可以从旧介质和旧格式的问题中解脱出来。但是从长远来看，这种方法至少有两个严重的问题。首先，迁移是需要消耗大量人力，自定义工作对于作业几乎没有任何杠杆作用，反而可能

会带来原始功能的损失。此外，更严重的是，迁移不能处理主要的机制转换。我们都希望从版本 8 的 RDBMS 迁移到版本 9，但是当异构数据库系统（HDS）占据一切时会怎样？没有人（包括我们）知道什么是 HDS 这一事实就表达了我们的观点。毕竟，当从网络数据库到关系型数据库发生机制转换时，我们不能够迁移很多数据库，是这样吧？

上面我们描述了很多的困难，如果这样，专家们还有什么希望长期保存数字信息呢？如果你对一主题非常感兴趣，并想在 50 年后仍能访问存储在数字数据仓库中的数据，请阅读 Jeff Rothenberg 的论文 Avoiding Technological Quicksand, Finding a Viable Technical Foundation for Digital Preservation，这是 Council on Library and Information Resources (CLIR)的报告。这个 41 页的报告可以在 www.clir.org/pubs/reports/rothenberg 的链接上获得 PDF 版本文件。该报告写的非常好，有较高的推荐率。

Jeff 对于这一问题，主要思想是建议开发一个模拟系统，虽然它是运行在现代硬件和软件上，但却完全模拟旧硬件。模拟选择在硬件级是因为硬件模拟被证明是重新创建旧系统的技术。他还描述了把旧数据集与解释该数据集所需的元数据封装在一起的必要性，以及针对模拟本身的所有规范。通过把所有内容保留在一个封装包里，数据就可以历经岁月，50 年后仍可以根据需要恢复一切内容。你需要做的是说明现有硬件上的模拟规范。图书馆已经完全解决了数字保存问题。在 Google 的搜索引擎里查找 embrittled documents，需要学习它们的技术并适应我们的仓库需求。

9.9 总结

在本章中，我们提供了典型 ETL 系统的操作框架整体介绍。本章前半部分重点介绍主要的调度方法。后半部分细述了随着系统成长和变得日益复杂的工作而带来的管理性能问题。最后，还为 ETL 系统安全性提出了一个简单框架。

10 元数据

元数据是个很有趣的话题，几乎所有的在数据仓库应用领域里的工具包括商务智能工具， ETL 工具，数据库和专门的资料库都拥有元数据的解决方案；很多书籍都给出了很好的元数据指导。然而，在数据仓库实现和发展的这些年之后，我们仍然会遇到端到端的元数据的问题。然而，大部分数据仓库还是手工维护那些独立存在于控件中的元数据。本章简单介绍了 ETL 小组作为消费者或厂商应该知道的元数据部分，而不是对元数据知识的夸夸其谈。我们制定了一系列的元数据结构来支持 ETL 工作组的工作。

流程检查

规划与设计：需求/现状 -> *架构* -> *实现* ->发布到生产系统

数据流：*抽取* -> *清洗* -> *规格化* -> *提交*

因为 ETL 系统是数据仓库的核心，它时常承担管理和储存数据仓库大量元数据的职责。可以认为没有比 ETL 系统更好的位置来存储和管理元数据了，这是因为系统环境要完全了解所有数据的细节特性才可以正确的发挥作用。在数据仓库中 ETL 处理程序是元数据最重要的创建者——数据沿袭（the data lineage）。数据沿袭追踪数据从源系统和文件中的精确位置直到最终被装载之前。数据血统包括源数据库系统的数据定义和在数据仓库中最终静止状态。如果你使用一个 ETL 工具，除了数据沿袭之外，属性也被存放在数据仓库中。但是 ETL 环境真的能在数据仓库中捕获和管理元数据？目前还是不可以。

如果你想在数据仓库元数据上探究细节问题，元数据解决方案是一个值得探究的方向。使用元数据模型，资料库，XML 和企业入口按照 Adrienne Tannenbaum(Addison Wesley 2002)要求生成信息。利用元数据，知识库，XML，和通过按照 Adrienne Tannenbaum (Addison Wesley 2002)的要求规划入口产生信息我们现在要开始准确的定义元数据的具体含义了。

10.1 定义元数据

在执行元数据方面落后的一个主要原因是元数据的精确定义是模糊不清的，

而且精确定义元数据是一个非常困难的工作。在 1998 年我们已经在《数据仓库生命周期工具箱》中定义过元数据。但是当看到这些词的时候，我们还是会感到吃惊。以下是我们所涉及的：

什么是元数据

元数据是数据仓库世界中令人惊异的话题。鉴于还不是很精确的了解它，不明确它在哪里，我们将会花费更多的时间来了解它，更多的时间来考虑它，如果我们没有什么收获的话会更加得感到羞愧。多年以前，我们很坚决地认为，元数据是有关于数据的数据。但这并不是很有帮助,它并没在我们的脑海中描绘出元数据里面准确的清晰结构。这种失真的观点最近逐渐清晰，而且我们更确定的发现了更多关于后台元数据和前端元数据的信息。后台元数据是与处理相关的，指导抽取清洗装载工作。前端元数据更偏向描述性和使我们的查询工具和报表工具更加稳定。虽然，处理和描述性元数据交叠，但是在我们的头脑中把它们区分开来是很有用处的。

后台帮助 DBA 将数据加载入数据仓库，同时这也是业务用户感兴趣的地方，他们会问起数据是从什么地方来等问题。前端元数据主要是为终端用户服务的，他的好处不仅可以使我们工具功能运行更加平稳，而且可以作为描述所有数据的业务数据字典。

即使这些定义很有帮助，可还不能指导数据仓库管理者到底应该做什么。但可以把传统的 IT 观点运用元数据上。至少我们应该：

- 制定完善的良好的注释目录
- 判定每一部分究竟有多重要
- 指定专人负责
- 判定一个和谐一致的工作规则
- 判定是否自己构建还是购买
- 专门存储用以备份和恢复
- 对需要他的人们开放

- 保证质量，保持完整和及时更新
- 实时控制

所有这些任务的文档都可以很好地解决问题了。

到现在为止，麻烦只有一个，我们还没有真正指出元数据到底是什么。我们注意到在上面的列表中的最后一项实际上并不是元数据，它是关于元数据的数据。藉由深入挖掘, 我们了解我们或许需要元数据的元数据。

源系统元数据

为了更好的理解他，让我们试着给所有的元数据类型制作一张列表。当然，第一次是不会很成功的，但我们会从中学到很多。先让我们看看源系统，它可以是主机系统，非大型机，用户桌面，第三方数据提供者，或在线数据源。我们这里假设在数据源端只做读取数据和将数据抽取到数据分段区域，数据分段区域可以是主机或下游机器。

数据源规范：

- 知识库
- 来源计划
- 拷贝纪录
- 所有权或第三方来源计划
- 存档主机数据的原由格式
- 相关源系统数据表和 DDL
- 电子数据表
- Lotus Notes 数据库
- 表达图形（例如，PowerPoint）
- URL 来源规格说明：
- 来源描述信息：

1. 各来源的所有权描述信息

2. 各来源的业务描述：
3. 最初来源的更新频率
4. 各来源使用的司法局限性
5. 存取方法，读取权限，和各来源存取口令

- 处理信息：

6. 主机或源系统工作计划
7. 使用 COBOL/JCL 或 C 或 Basic 或其他语言来实现抽取
8. 如果使用工具的话，那么还包括这种自动化抽取工具的设置
9. 特殊抽取的结果，包括抽取时间和工作完成率

数据集结元数据

现在我们列出所有数据集结元数据。我们在主机上通过手工代码 COBOL 或使用自动化工具。或者我们把 FLAT 文件或多或少抽取到一台不同机器的单独数据分段存储区。无论哪种情况，我们都不得不关心元数据的描述

数据需要的信息：

- 数据传输计划和特殊传输结果
- 数据传送区的文件用法，包括持续时间消耗和所有者信息

维度表管理：

- 规范化维度的定义和规范化事实的定义
- 关联源的 Job 规范，剥除域，查找属性
- 降低每一个引入描述属性维度规则的变化速度（例如，重写创建新纪录或创建新域）
- 每一个生产键所分配的当前代理键，也包括在内存中执行映射的查找表。
- 前一天生产维度的副本，作为比较差异的基础

转换和聚合：

- 数据清洗规范
- 数据增加和映射转换（例如，扩展缩写和提供详述）
- 为数据挖掘所准备的数据转换（例如：解释空值和数字的度量 ）
- 目标计划设计，源和目标数据流，和目标数据所有权
- 数据库管理系统的导入脚本
- 聚合定义
- 聚合用法统计，基础数据表使用统计，以及潜在的聚合
- 聚合更改日志

审计，工作日志和文档：

- 数据沿袭和审计纪录（数据确切来源和来源时间）
- 数据转换时间日志
- 数据转换运行时间日志，成功记录摘要，时间戳
- 数据软件版本号
- 抽取过程的业务描述
- 抽取文件，抽取软件和抽取元数据的安全设置
- 数据转换的安全设置(授权口令)
- 数据分段传送区存档文件日志和恢复进程
- 数据分段传送区存档文件日志安全设置

DBMS 元数据

一旦我们最终决定将数据转换导入到数据仓库或数据集市的数据库管理系统，元数据的另一部分开始起作用：

- 数据库管理系统系统表目录
- 分区设置
- 索引

- 磁盘带区规范
- 程序处理提示
- 数据库管理系统级别和安全的权限和授权
- 视图的定义
- 存储过程和 SQL 管理脚本
- 数据库备份，状态备份程序和备份安全性

前端元数据

在前端，我们也扩展了元数据，包括：

- 纵列表分组的业务名称和描述等等
- 模糊查询和报表定义
- 连接规范工具设置
- 灵活打印规范说明（以一种易读的方式重新标记区域）
- 最终用户文档和培训帮助，包括供应商提供和 IT 提供
- 网络安全用户授权
- 网络安全鉴定证书
- 网络安全使用统计，包括试图读取的日志和用户 ID
- 个人使用者描述包括人力资源链接
- 提高影响读取权限的传输流畅性
- 连接到合约人和合作者可以追踪读取权限的影响范围
- 数据原理，数据库表，和视图报告的使用和读取映射
- 资源回收统计
- 收藏的网页（作为所有数据仓库访问的范例）

现在我们可以发现为什么我们不能准确全面地描述出什么是元数据了，它包含了所有的东西，甚至是数据本身。突然之间，数据似乎是最简单的部分。在某

种程度上，元数据是数据仓库的 DNA。它定义了所有元素和各元素之间如何协同工作。

所以，你如何捕获和管理这些元数据单元？你不能，至少 ETL 团队是不能完成的。纵观过去的几十年里社团协会，联盟，委员会，企业联合起来解决有关元数据的困惑。但是直到今天，还没有形成一个较为普遍的解决方案。我们发现，作为一名 ETL 团队的成员，你需要一定的元数据做你的工作。它能很方便的聚焦到我们刚才所列出的项目中，并且把那些项目分成以下三类：

1 业务元数据 在业务层面上描述数据的含义

2 技术元数据 描绘数据的技术方面包括数据的属性，例如数据类型，长度，沿袭，数据评估结果等等

3 过程处理元数据 介绍运行 ETL 处理的统计信息，包括度量标准比如纪录导入成功，记录丢弃，处理时间等等

除了这三种元数据之外我们还应该考虑到元数据的另一个方面：标准。标准是 IT 界各企业共同合作和维持的另一项尝试。在第 10 章我们会定义元数据管理员的角色，并给这一角色指定一系列责任。你们的组织在适当的位置上很可能会有很多标准，而你也可以通过本书所介绍的方法采用适合的数据仓库和 ETL 标准。如果你找的是更深一层次的元数据标准研究，我们会在下一部分给你详细的介绍。

当你阅读这一章节的时候，请参照图 9.1 辅助理解。图 9.1 向我们展示了 3 种主要的 ETL 系统元数据，并在正文中会讨论各自部分的元数据表。

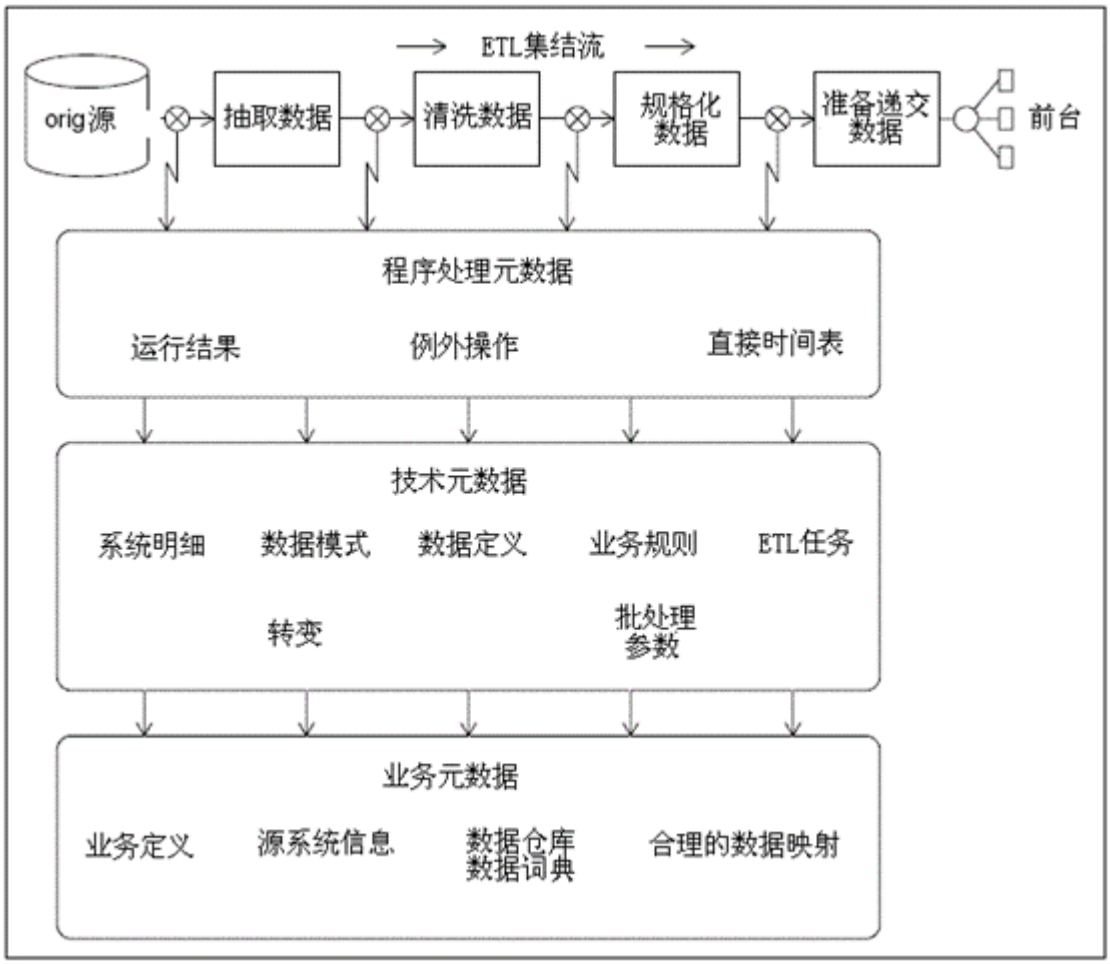


图 9.1　数据仓库后端元数据源

这张清单列出了到目前为止在本书中提到的希望使用和收集元数据的所有地方：

- 每一个数据阶段的起源和处理步骤（第一章）
- 元数据资料库作为第三方 ETL 工具的一个有利条件（第一章）
- 元数据需求构架：源表，清洗和处理（第一章）
- 为终端用户介绍有用的元数据（第二章）
- 抽取转换应用（第二章）
- 规范化元数据(第二章)
- 元数据的 XML 描述（第二章）
- 平面文件中元数据的缺失（第二章）

- 分析元数据的效果（第二章）
- 计划创建描述沿袭的元数据，业务定义，技术定义以及处理过程（第二章）
- 逻辑数据视图（第三章）
- 抽取过程中源数据的捕获计算（第三章）
- 源数据库描述（第三章）
- ETL 工具读取 ERP 系统元数据（第三章）
- 数据评估的结果（第四章）
- 错误事件跟踪事实表（第四章）
- 审计维度（第四章）
- 代理键最高值（第五章）
- 聚合数据（第六章）
- 处理数据搭建 OLAP 架构（第六章）
- 装载到控制文件（第七章）
- 支持恢复处理的元数据（第八章）
- ETL 系统参数（第八章）
- 任务依赖（第八章）
- 任务操作性统计，比如性能和资源的使用（第八章）
- 元数据资料库报告（第八章）
- 数据表净化规则（第八章）

我希望看到如此之长的清单时不要沮丧，本章的其他部分将会为特定元数据追踪信息。

10.2 业务元数据

由谁在数据仓库中负责元数据，这经常是个有争议的话题。有人说它是数据仓库业务分析人员的任务，应该在收集需求的阶段被创建。另一些人认为，源系统业务分析员应该创建业务期限，这是因为大多数数据仓库属性是在源系统中建立的。但是，还有一部分人认为创建和维护业务元数据应该是数据建模人员的一部分工作，因为它本身就是逻辑数据模型的一部分。

你无法平息这些争论，但是有一些业务元数据，ETL 团队会受到它的影响，而且还需要维护它来准确反映事实。ETL 团队不用去理会如何生成业务元数据，但如果发生变化，一定需要适当的人员来沟通。从 ETL 的角度来看，业务元数据是一个代理的元数据。代理元数据是通过一个系统获得，可以直接应用于另一个系统而不需要外在处理。一些商务智能工具被设计用来读取 ETL 资料库中的业务逻辑定义，并把它展现给用户。为数据和数据的元数据提供一个一站式服务。

为了让这更有趣些，我们必须要让你明白数据仓库可以理解为一个不同业务定义的集合场所。这些定义是为同一个属性服务的。记住，通过设计，在数据仓库中的数据可来源于多个系统，例如，市场部门定义了一个客户，这个客户有一个已经注册的账户。然而销售部门会认为只有实际发生购买的才可以成为客户。作为一个被推荐的实例，数据仓库管理员应该把所有重叠元素的所有者集中在一起，让他们从企业的角度接受一个单一的概念。我们在本书中描述的这个过程被看作是规范化业务定义、标签和估量标准，这也许说它是来源于一整个新系统或者是来源于众多源系统中的一个。定义的企业规范存放在数据仓库和 ETL 工具中。

ETL 小组作为数据仓库后台的一部分，不应该涉及到创建业务元数据库。然而，你应该理解所造数据的用途，同时当你需要业务定义的时候可以参考一下。

业务定义

数据仓库团队经常要设计查询性能的数据库结构，和优化装载性能的 ETL 处理程序。一旦数据库设计完成，导入数据，小组的工作会集中到数据仓库的商务智能上，开发良好的用户界面，生成漂亮的图形化报表。但是对业务，在数据仓

库方法中最重要的因素是对可用元素的定义。如果用户不能理解数据，对数据产生误解，那么定义的原理对组织来说完全没有意义。业务定义对于数据仓库来说是至关重要的。不仅仅是终端用户需要业务定义，ETL 也需要业务定义给出他们正在操作数据的上下文关系。如果你试图在完全定义业务之前描述数据模型和数据沿袭的话，你将会意识到业务定义对于 ETL 团队来说是多么的重要。如果你急于完成数据沿袭以便 ETL 开发工作尽早开始，你将会被迫向 ETL 团队频繁解释数据模型中的主要数据元素的目的用途。因为他们没有足够的信息对数据开展工作。因此，在 ETL 小组获取数据沿袭之前，开发的重点应该从加快数据沿袭中转移出来，集中整合业务定义之间的关联和数据仓库的数据元。一个典型的业务定义矩阵包括 3 各主要组成部分:

- 物理表和列名称　数据仓库中数据元素的业务解释是基于数据库中真实的表和列名称。如果商务智能工具仅展现业务，而完全隐藏数据结构的物理实例的话，那么物理名称不需要展现给最终用户。然而 ETL 团队只处理物理名称，这就需要把物理名称和数据元素的业务定义适当的联系在一起。
- 业务列名称　数据库所存储的数据元素在技术上是由前缀后缀和下划线组成的。业务组需要这种技术性名称和有实际意义名称之间的翻译。例如，业务名称为 EMP STTS CD，有可能是员工身份代码或者仅仅是员工状态。我们无法在这种模糊不清的情况下工作。记住，业务名称是要被商务智能工具展现出来的。此外，业务名称经常会成为用户报表的行和列标题。
- 业务定义　业务定义是指描述业务属性含义的一两句话。数据仓库中的每个属性一定会有一个业务定义。如果无法用业务定义属性，那么通常意味着这个属性没有分析价值，有可能不需要存放在数据仓库中。如果业务上要求它必须存在于数据仓库中，那么它一定会有业务定义与之相联系。

业务定义矩阵可以简单到是三列电子表格。然而，你应该努力尝试让各种特殊的元数据在技术环境允许的情况下尽可能的集中和共享。事实上，所有的 ETL

工具都支持捕获和存储元数据。ETL 工具应该和数据模型工具、数据库一起获得业务定义，并通过商务智能工具把业务名称和业务定义展现给最终用户。

源系统信息

ETL 团队需要了解在数据仓库中读取每一张表的所有隐藏细节。就像你需要雇用一个人来填补目前 ETL 团队中的空缺，在雇用员工之前，HR 部门会寻找候选人，预先筛选他们，然后把合适的人选介绍给你，安排正式的面试。所有你可以保证这些人是适合这个空缺的职位的。在面试过程中，你在认为候选人适合该职位之前就也可发现一些弱点。当迁移到数据仓库的时候，数据模型就好像那个空缺的职位，数据仓库架构师寻找并预先筛选出数据源，然后数据必须要经过分析来发现它们自身的弱点。制定数据改造计划，使数据能够较为理想的被导入到数据仓库中。一些数据可能会非常完美，其它数据可能就需要做些转换或者因为数据质量的问题而被丢弃。当你分析源系统时，你需要特定的数据仓库，至少你需要以下元数据属性。

- 数据库或文件系统 当涉及到源系统或文件的时候，这个名称经常会用到。这不是一个技术上的服务器或者数据库实例。像有关销售的数据库或资产管理系统这样的名称对于这样的元数据片是很常见的。
- 表规范 ETL 团队一定要了解表的用途，表的大小，主键和预备键，和所有列的清单。
- 排异处理规则 你一定要多了解数据质量的相关内容，仔细考虑 ETL 程序是如何操作数据的。
- 业务定义 非常有用。尽你最大努力来做好这些，那种一两句的定义对于你了解数据的意义是没有任何价值的。
- 业务规则 每个表都有自己的业务规则。对于理解数据和测试异常时，业务规则十分有必要的。每个业务规则都会被测试，每个规则的异常都会记录成文档，被源数据系统或 ETL 程序处理。业务规则包括数据表插入新纪录，更新和删除的记录。如果你很幸运，业务规则会在源数据库管理系统(DBMS)以参照完整性，强制检查和触发的形式存在。

在数据仓库项目中数据分析阶段需要大量时间研究源系统。缺少源系统元数据会导致数据仓库小组的额外研究并且要解决更多的故障。为了缩减成本，所有的源系统元数据，必须在 ETL 程序开发之前提供给 ETL 团队。

数据仓库数据字典

当我们谈到数据字典时，并不是指 DBMS 的目录。数据仓库数据字典是关于所有数据元素和他们业务定义的清单。与源系统业务定义相似，数据仓库数据字典包括物理表和列名称，业务名称和业务定义。对于数据仓库，数据业务元数据电子表格就足够了。很多数据仓库环境依赖于 ETL 资料库来存储数据字典，这是因为商务智能工具需要读取元数据来进行展示工作。

很多商务智能工具与 ETL 元数据库紧密相连的。当你选择工具时，要搞清楚你的 ETL 工具是否具有开放型资料库可以被查询工具读取或至少有一个适配器或代理来完成这个工作。

逻辑数据视图

逻辑数据视图是 ETL 工作组的生命线。第三章讲述了 ETL 规范角度上的逻辑数据视图的信息。从元数据的角度上看，逻辑数据视图是由从源到目的的映射，从逻辑上解释了,数据从源系统中抽取出来，到装载到数据仓库中，数据到底发生了什么。

逻辑数据映射是元数据的重要部分。ETL 工作组首先使用文档作为功能性规范来创建物理 ETL Job。然后与终端用户确认工作。当用户接受测试阶段发生问题时，文档可以提供指导。当数据进行质量保证测试的时候，当 ETL 给质量保证组提供各种视图时，文档也非常有用。最后，ETL 团队将文档提交给数据库管理组，并提供数据转换的信息，在程序发生问题的时候数据库管理员可以及时维护。

交叉参考：参考第三章在逻辑数据映射中创建准确元数据元素，如何创建，维护和使用信息。

10.3 技术元数据

技术元数据服务于多种用途，是 ETL 团队所涉及的元数据中最有趣的一个。

它包括了到列名称，数据类型，存储和 RAID 矩阵的配置。作为 ETL 团队的一个成员，你不需要太关心类似硬件的配置这些方面。数据仓库中数据元素的物理属性是最重要的。为了得到基本的元数据，需要了解你的源系统和目标数据仓库的物理数据模型。如果你的源包括平面文件，你需要每一个文件的文件格式。

系统详细目录

ETL 团队必须要十分了解数据仓库中每个系统的技术元数据，才能正确构建物理 ETL 工作。数据的技术定义或许是技术人员被问起有关元数据时首先想到的话题。毕竟，这才是真正的描绘数据的数据，它是数据的容器和框架结构。ETL 团队必须至少了解数据定义的三种环境：

- 源数据库
- 分段处理区域表（例如，抽取，清洗，转化，确认，为传输作准备）
- 数据仓库展现区

最好为每个环境提供一个实体关系图，每个系统至少应有一个清单包含以下元素：

- 表 表或文件的一份详尽清单，这份清单有可能在抽取和装载过程中使用。经常，只提供逻辑数据映射的源系统数据表。有一些关联表没有被列入清单但也是必需的。当源系统有多对多关系的时候，一个设计很好的系统来关联数据表维持这种关系。
- 列 对于每张表，你都会需要一张数据映射所要求列的清单。源系统数据库管理员能够提供你所需要的列的清单。这样会有助于工作。
- 数据类型 数据表每一列都有一个数据类型。在不同数据库系统有些数据类型不尽相同。很幸运，大部分专门的 ETL 工具可以转换相应的数据类型。例如，SQL Server 中的 INTEGER 类型在导入到 Oracle 时可以自动转成 NUMBER 型。要知道数据库管理员可以自定义数据类型作为用户定义数据类型。用户定义数据类型基于数据库核心数据类型，可以在此基础上扩充其定义，包括列长度，是否可以为空，是否可以有特殊字符，比如电话号码。

- 关系型数据库 关系型数据库支持参照完整性。参照完整性和表之间的关联关系能够保证数据的唯一一致的数据导入。数据之间的关联关系是通过表与表之间的主外键连接展示的。

数据模型

在物理模型图表(规格化的或维度的)上的数据模型其实只是一个元数据展示,而且元数据本身没有什么特别。但是，他们可以成为宝贵的财富，因为它可以使你的 ETL 工作组只用瞥一眼就可以了解和确认关联关系。即使列出的逻辑数据视图,你不能指望它是完全彻底的。我们建议，在你办公室的墙上悬挂上标示所有源系统(数据仓库)的物理模型。在图上标示出通过筛选明确的目标。此外，这样做还有一个直接的好处,可以增强作决定的决心,给项目经理们留下积极的印象。

数据定义

每一个潜在的数据存储中的数据定义应该是一致的。数据每一次进入数据库或者进入文件后，数据质量很容易发生变化。如果在各个环境中的数据定义各不相同，那么 ETL 团队就要做数据转换来避免数据不一致带来的灾难。除了上一节所列举的属性外,下列数据定义元数据的内容必须提供给 ETL 团队.

- 表名称 表和文件的物理名称
- 列名称 表和文件中列的物理名称
- 数据类型 表的数据类型可以分为以下几类：普通类型包括数字型，字符型，日期型，二进制。在绝大部分关系型数据库中还可以允许自定义数据类型。这种类型是基于普通类型大部分由强制的格式规则定义的。数据类型是相互排他的，不能共存于一个列中。
- 域 数值要进入的列被称为域。域通过外键，检查约束或数据库顶端应用来执行。如果将应用加给域，设计团队必须给 ETL 团队提供一张允许值的清单。
- 参照完整性 如果你的数据来源于数据库,你可以通过外键来指出另一张表的主键来保证数据的唯一性。如果要将参照完整性加给应用层，那么设计组一定要提供参照完整性的规则。数据仓库中的参照完整性被认为

不是必要的，这是因为所有的数据都是经过 ETL 处理以受约束的形式进入数据库的，在数据库级不需要强制的约束完整性。

- 约束 约束是业务规则的另一种物理执行。数据库约束可以消除空值，增强外键查找等等
- 缺省值 万一实际值是不可获得的，ETL 元数据的缺省值可以分配为字符串型，数字型，日期型或 bit 型。在源系统中，列的缺省值经常是被分配为数据库级别的。在数据仓库中，缺省值的指派发生在 ETL 处理中。数据仓库中的缺省值最好坚持始终如一的一致性。
- 存储过程 存储过程，存储了已经在数据库中写好的 SQL 语句。可以通过存储过程看到你的源数据是如何使用的。每个数据仓库项目都不可避免的要涉及到源系统的分析型存储过程。
- 触发 当数据库系统中的记录要增加，删除或更新时通过触发自动执行 SQL 程序。像存储过程一样，通过触发，也可以知道数据是如何使用的。触发经常通过向加入到表中的数据增加额外的检查来增强外键的约束。当表中的数据发生变化或被删除时，触发还承担审核表的责任。审核表是数据仓库中删除数据的重要审核来源。

业务规则

业务规则可以分为业务或技术元数据。当谈到业务规则的时候我们喜欢把它归为技术，这是因为它对于 ETL 过程来说实在太重要了，而 ETL 过程不折不扣地属于技术范畴。所有的业务规则都要以编码的形式被包含在 ETL 过程中。业务规则可以包括起始域中的任何允许值，缺省值，和计算。在源系统中，业务规则在存储过程，强制约束或数据库触发中被执行。但是业务规则还是最常出现在应用的程序中。以前的那些系统中，特别是大型机环境，不存在实际的源代码，只有应用的编译部分。在这样的情况下，业务规则很难取到，就要经常进行数据分析，或询问原来编写应用的程序员。业务规则的元数据会在功能性或者技术性文档和本地程序语言的源代码或伪代码之间进行改变。

业务规则必须和逻辑数据视图紧密结合在一起。有时，业务规则会在逻辑数据视图中被忽略直到执行完第一次 ETL 处理之后才被人注意，或被用户在 UAT

中被发现。当有新的业务规则时，逻辑数据视图的元数据必须更新来反映新的规则。

10.4 ETL 生成的元数据

到目前为止，我们一直关注的是在 ETL 环境之外生成的元数据或者是其他源系统提供给 ETL 团队的元数据。本章的其余部分将会介绍 ETL 生成的元数据。ETL 团队可以通过这种元数据来管理 ETL 程序，而终端用户或数据仓库成员也可以更好的理解数据仓库中的数据。

当 ETL 物理程序被建立之后，一定要生成明确的元数据来捕获每个处理的内部工作。ETL 元数据可以被分为四类：

- ETL 任务元数据　ETL 任务可以被理解为存储所有转换的容器。任务元数据非常有价值，他包含了数据仓库中各元素数据沿袭。每个 ETL Job 任务——从抽取到装载，包括所有的转换——都可以被 ETL 任务元数据捕获。
- 转换元数据 每个任务都由多个转换组成。任务中数据处理的任何形式都是由专门转换工作来执行的。
- 批处理元数据 批处理是一种执行任务集合的技术。批处理应该具有配置执行连续性和并行处理的能力。而且，包含分支批处理。分支处理在数据仓库中非常常见。你可以有一个装载维度的批处理和一个装载事实的批处理。这些批处理可以集中装载数据到明确的数据集市。批处理被定制为阶段执行或者依照触发计划执行。
- 处理元数据 每当执行批处理，就会生成处理元数据。对于描述数据是否被成功装载到数据仓库中，处理元数据十分重要。

如果你不是一个 ETL 新手的话，你大概会注意到元数据种类并不是按照容器顺序列出的。而是以它们出现或者生成的顺序列出的。例如，每个批处理都包含不同的任务，你在批处理任务之前肯定是应建立好各个任务了。ETL 元数据的每一种类型都包含自己明确的属性，这些属性需要被创建，维护和发布。而并发的部分会检查在 ETL 环境中技术型元数据的特性。为了保持本书的一致性，我们谈到数据映射的物理执行时，把它看作任务，而把它的容器看作批处理。这些术语

和一些 ETL 工具一致。我们对于术语的指定并不推断任何的特殊技术。如果你的技术不允许用任务调用从源到目的的物理映射,可以用我们自己定义的替代任务取得元数据

如果你不用某种专门的工具来执行你的 ETL 解决方案，你就没有借口去生成主要元数据。工具意味着减轻工作的负担。没有工具，你将不得不手工生成和维护元数据。如果你必须手工创建元数据，可以在 PVCS 或 SourceSafe 这样的版本控制工具中使用电子表格。

ETL 任务元数据

ETL 元数据可以是一个技术含量非常高的话题，那些终端用户往往是对此不甚理解。那些包含程序代码的从源到目的的映射对于业务用户特别是不懂技术的终端用户来说是非常神秘的。然而，这些映射对于理解数据仓库中数据的真实沿袭是非常重要的。当数据的可信性遭到质疑，或需要提供数据的完整需求时，数据仓库小组也要在元数据中寻求答案。

描述从源到目的映射的最佳办法是利用一个专门的 ETL 工具。这些工具可以给用户提供信息报告。要求你可能的 ETL 产品提供商向你明确地展示他们工具的描述能力。

图 9.2 向我们展示了 ETL 任务元数据被创建，存储，和发布的元素。下面的这些元素需要你去追踪，来帮助管理任务和向你的经理和用户发布他们的身份和功能。

任务名称	任务用途	源表/文件	目的表/文件
browser_platform_dim	加载 browser_platform_dim 表	stg_web_log	browser_platform_dim
claim_reason_dim	加载 claim_reason_dim 表	claim_reason	claim_reason_dim
claim_status_dim	加载 claim_status_dim 表	claim_status	claim_status_dim
claims_agent_dim	加载 claims_agent_dim 表	claims_agent	claims_agent_dim
claims_fact	加载 claims_fact表	claims_fact	claims_fact
clickstream_customer_dim	加载 clickstream_customer_dim 表	weblogs, customer	clickstream_customer_dim
customer_credit_card_dim	加载 customer_credit_card_dim表	customer_credit_card	customer_credit_card_dim
customer_dim	加载 customer_dim 表	stg_web_log	customer_dim
date_dim	加载 date_dim 表	date_dim.csv	date_dim
delivery_option_dim	加载 delivery_option_dim 表	delivery_option	delivery_option_dim
mktg_stg_tables	加载 mktg_stg_tables表	campaigns.csv	mktg_stg_tables
page_dim	加载 page_dim 表	stg_web_log	page_dim
page_events_dims	加载 page_events_dims 表	stg_web_log	page_events_dims
page_events_fact	加载 page_events_fact 表	stg_web_log	page_events_fact
partner_promotion_dim	加载 partner_promotion_dim 表	partner, promotions	partner_promotion_dim
payment_status_dim	加载 payment_status_dim表	payment_status	payment_status_dim
product_dim	加载 product_dim 表	product	product_dim
server_dim	加载 server_dim 表	weblogs, servers.csv	server_dim
stg_web_log	加载stg_web_log表到数据仓库	weblogs	stg_web_log
time_dim	加载 time_dim表	time_dim.csv	time_dim

图 9.2 ETL 任务元数据

- 任务名称 物理 ETL 任务的名称
- 任务用途 最初中心任务的简要描述
- 源表/文件 所有源数据的名称和路径位置
- 目的表/文件 在转换完成之后所有结果数据的名称和目录路径
- 丢弃文件名称 丢弃文件的名称和路径位置。那些来存储在装载过程中没有没载入目的地的数据表或文件，我们称之为丢弃文件
- 预处理前置任务 在任务被执行之前需要处理任务或脚本
- 后置任务 在任务处理之后需要执行的任务或脚本

任务

一个任务是一系列转换的集合，这些转换执行物理上的抽取，转换和装载程序。一个任务的元数据是物理的源到目的映射。任务应该根据装载的目标表或文件来命名。如果你的 ETL 处理有很多分段，每个任务必须包含前缀来暗示它的目的。ETL 任务大致上可以被分为三类：

- 抽取 EXT_<table name>暗示这个任务的主要目的是从源系统中抽取数据
- 中间阶段（例如，清洗和规范化） STG_<table name> STG 前缀意味着这个任务并不涉及到源或目的。他只是一个存在于阶段区的中间过程。如果处理不止一次的涉及到阶段区，那么在前缀后面附加上计数器（例如 STG1, STG2, STG3 等等）
- 目标 TRG_<table name> 意味着任务是装载数据到目标数据仓库。我们以前还见过 FAC_<tablename>和 DIM_<table name>的命名方式来暗示目标表是事实表或维表。我们还没有看到这样做的价值，但是这种习惯可以接受。

转换元数据

转换元数据是关于 ETL 处理构造的信息。ETL 开发人员把大部分时间用于创建和复用数据转换。转换是由客户化函数，存储程序，常规程序组成的。这些程序包含了指针，循环，内存变量。这样很难以文档的形式记录或提供元数据。在 ETL 处理过程中的任何对数据的操作就被认为是转换。如果你用 SQL 来写 ETL 的话，你需要辨别你程序中每一个不同的部分，用转换元数据中常用的属性把他们分类标注。

专门的 ETL 工具为数据仓库环境预先定义了通用转换，并把他们打包提供给使用者。预先建立的转换加速了 ETL 的开发，而且还可以在暗中捕获转换元数据。在大部分 ETL 任务中的通用数据转换包括：

- 源数据抽取 这个抽取可以像 SQL 中的 SELECT 语句一样简单，或者像是 FTP, 或是读取 XML DTD 或大型机的副本卷
- 代理键生成器 这些可以简单调用一个数据库序列或者包含复杂的常规程序来管理内存或包含第三方软件。插入到数据仓库中的最后数字，也是需要维护和呈现的元数据
- 查找 主要用来从事实表装载维度中或阶段区中的参照完整性获得代理键。如果你使用未加工的 SQL, 这将包括所有的内外连接和 IN 语句
- 筛选 这条规则决定了哪些行被装载和导入。元数据是业务规则或强制约

束用来应用过滤的。在 ETL 处理中随处都可以用到筛选。在 ETL 处理中最好尽早使用筛选来过滤你的数据。

- 路由器 有条件的路由发送，就像 CASE 语句那样
- 联合 通过一致的列的定义合并两个传递途径
- 聚合 当事实表和最低级别的转换不是同一粒度的时候，你需要对元数据进行聚合。有关聚合的元数据包括任何的计算，例如聚合函数，函数本身——计数，汇总，均值，分组等——和聚合函数分组列。分组列声明了聚合粒度。
- 异构连接 当元数据来自于不同系统，他们通常是在任何单个数据库之外连接（除非你使用数据库 link）。在数据库系统环境之外连接不同系统的方法需要在元数据中定义和描述。
- 更新策略 更新策略包含业务规则，业务规则决定了一条记录是否会被增加，更新或删除。它也包含了维度变化策略。
- 目标装载器 目标装载器告诉 ETL 处理哪个数据库的哪张表，哪几列需要被装载。另外，当有散列装载的时候，这个文档被用来载入数据到数据仓库中。
- 每个转换获得数据，操作数据到某个程度，然后传送数据到任务队列中的下一个转换。描述转换的元数据属性包括：
- 转换名称 单一的任务包含多个转换，把每一个辨别区分清楚对于管理和维护来说非常重要。每个转换必须有唯一的名称，命名要有含义，还要遵从标准命名规则。本章将会描述转换的命名规则。
- 转换的意图 转换的意图必须容易辨别。许多 ETL 工具用不同色彩来标记预先定义的转换。如果用手工代码的方式处理 ETL 程序，那么要确保元数据的内容要包含代码和转换矩阵的信息。
- 输入列 数据元素反馈给转换过程
- 物理计算 操作计算的实际代码

- 逻辑计算 物理计算的文本等价物。在代码含义模糊不清时有用。
- 输出列 数据转换的结果，可用于下一个转换

转换的命名习惯

转换是 ETL 任务的一个构成部分。每种转换在命名格式上都会稍有不同。易于可维护性的原因，在建立 ETL 转换时，最好遵从以下的命名规则：

- 源数据抽取：SRC_<table name>
- 代理键生成器：SEQ_<name of surrogate key column being populated>
- 查找：LKP_<name of table being looked up or referenced>
- 筛选：FIL_<purpose of filter>(例如，FIL_SUPPRESS_BOTS 禁止在网站上点击)
- 聚合：AGG_<purpose of aggregate> (例如，聚合网站月度点击量)
- 异构连接：HJN_<name of first table>_<name of second table>
- 更新策略：UPD_<type of strategy (INS, UPD, DEL,UPS)>_<name of target table>
- 目标装载器：TRG_<name of target table>

批处理元数据

当所有的任务被设计和建立之后，接下来的是这些任务的计划执行。对于那些负责在数据仓库中增加导入任务的小组来说，转载进度是元数据中至关重要的一部分。图 9.3 描绘了对于数据集市来说一个典型的装载是如何进行的。图中装载进度包含以下元数据属性：

```
B_Daily_DW_Load - 定时在每天早上的12：30
        S_DW_Prepare_logs - 这个动作在数据库的集结区为查询clickstream_exclude表创建一个
                            exculde.txt文件，这些项需包括所有映射区域，并且 'bots'不在
                            page_events_fact表。
                Pre-session process  - ./Scripts/del_files.pl - 删除当天的部分日志
                Post-Session process - ./Scripts/create_output_file.pl - 创建文件output.txt，包括所有
                                       历史pages_views，这个动作在创建exculde.txt成功基础
                                       之上。
        S_DW_DB_Dim - 所有的并发动作依赖数据仓库的数据
                B_DW_Content_Dim_Load
                        S_Content_Dim_Product - 在最近一次数据仓库装载之后，装在新的Product
                        S_Content_Dim_Category - 在最近一次数据仓库装载之后，装在新的Category
                        S_Content_Dim_Ad - 在最近一次数据仓库装载之后，装在新的Ad
                        S_Content_Dim_Link - 在最近一次数据仓库装载之后，装在新的Link
                S_Last_Process_Date - 在数据仓库集结区更新S_Last_Process表
B_DW_DIMS_AND_FACTS
        S_DW_Stg_Web_Log - 装载所有的Web_Log排除在数据仓库集结区的
                        S_DW_Page_events_dims - 并发装载所有数据仓库维表
                        B_DW_Load_Facts_tables - 并发装载事实表
                        S_DW_Hits_Fact - 装载Hits_Fact表
                        S_DW_Pages_Events_Fact - 装载Page_Events_Fact表
        Post-Session process - ./Scripts/update_brio_repo.sql - 这个脚本通过触发brio服务来
                                                                生成报表
                        S_DW_load_audit - 在数据仓库里增加一个动作来检查日志表
        Post-Session process - /Scripts/move_logs.bat - 完成之后，把文件日志从/incoming移动
                                                        到outgoing目录
```

图 9.3　点击流数据集市装载调度

装载进度

- 依赖的批处理 一般一个批处理会嵌套包含好几层的批处理任务，来并行执行多个任务或维持任务与任务之间的完整性。例如，一个装载数据集市维度的批处理必须在执行事实表任务之前装载成功。记住，数据仓库并不完全依赖参照完整性。通过从属的批处理是在数据仓库中实施参照完整性的一个办法。
- 频度 数据仓库中的数据导入频度有：每月，每周，每日或连续装载。这一部分的元数据就是定义批处理的执行频率的。
- 执行进度 如果一个任务是每日执行，那么元数据的属性就会捕获到每次批处理执行的时间。如果是每月执行，那么在元数据中会纪录处理执行的那个具体日期。批处理必须具备在任何预定时间预定计划下执行任务的能力。
- 恢复步骤 当处理失败的时候，需要有一个恢复的动作。恢复有可能是一个很长时间的处理，经常在另一份独立的文件中提出。进行恢复处理要让负责 ETL 批处理的小组了解进程。

数据质量错误事件元数据

在第四章中已经对元数据描述数据质量有了深入的讲述，对三张主要表进行了详细讲述。但是为了保持一致，我们在这里列出了清洗和规范化阶段的元数据索要捕获的数据元素。

首先，筛选表包含：

- ETL 注入阶段 描述所有 ETL 处理中要用到数据质量筛选的阶段
- 处理顺序号 是一种简单的时序安排，它通知 ETL 主处理程序执行筛选的顺序。
- 在同一个 ETL 处理阶段里有相同处理顺序号的数据质量筛选可以并发执行。
- 严重性评分 用来定义错误的严重性，
- 异常反应 异常反应属性会告诉所有 ETL 处理当发生错误时是否让该数据通过，或拒绝数据，或停止整个 ETL 处理过程。
- 筛选分类名称 根据描述给定数据质量筛选分类，例如，完全，验证或拒绝进入
- SQL 声明 记录进行数据检查的 SQL 或 SQL 程序的真实片断。如果这些 SQL 可以应用，将返回那些不符合数据质量规则数据的唯一标示符。所以，它可以用来向事件事实表中插入数据

然后，主要错误事件事实表包括以下几个方面

- 存储数据标示 唯一标示错误纪录
- 错误严重性评分 从 1 到 100 给错误赋值

错误事件事实表用外键关联日期，时间，ETL 批处理，表，和源系统维度。这些维度提供了衡量错误事件事实表的内容。

审计维度包括以下方面：

- 全面数据质量评分
- 完全性评分

- 校验评分
- 超限度评分
- 筛选域的数目
- 最大错误事件
- 清洗和规范化时间戳，包括某个具体 ETL 任务执行的开始时间和结束时间
- 全面 ETL 时间戳，包括完整端对端 ETL 任务的开始时间和结束时间
- ETL 发布版本号
- 其他审计版本号，比如配置版本，目前转换逻辑版本，所依赖的业务逻辑环境

处理执行元数据

事实上数据仓库中所有的处理元数据都是由 ETL 过程生成的。每次一个任务或批处理执行，统计或成功指示器都需要被捕获。导入统计是元数据的一个重要部分。他所收集的信息包括 ETL 处理执行和实际装载结果。

运行结果

元数据元素可以帮助你理解 ETL 任务中的活动和批处理或评估成功的处理：

- 主题名称 可以是数据集市或者描述为某个特定区域而执行批处理。
- 任务名称 执行程序的名称
- 处理行 从源数据系统中读取或处理的行数统计和百分比统计
- 成功行数 装载到数据仓库中数据的总数和百分比
- 失败行数 被数据仓库拒绝的数据总数和百分比
- 最近错误代码 在数据装载过程中，最近的数据库或者 ETL 异常的错误代码
- 最近错误 最近错误的文本描述
- 读取能力 用来衡量 ETL 处理性能的。用行/秒来描述。当源系统读取发

生瓶颈时，记录读取能力。

- 写能力 用来衡量 ETL 处理性能的。用行/秒来描述。当写入目标数据仓库遭遇瓶颈时，记录写能力。
- 开始时间 任务开始时的日期，时间和分秒
- 结束时间 任务结束时的日期，时间和分秒，这个时间是不考虑任务是否成功的时间
- 耗时 与开始时间和结束时间不同，这是性能分析的一个重要指标。在大多数情况下，只记录每秒钟读取和写入多少行是不够的，因为这项指标在很大程度上还取决于你导入数据的总量。
- 源文件名称 ETL 抽取涉及的数据的表或文件的名称，应该不止一张表和文件
- 目标文件名称 ETL 涉及目标的数据的表或文件的名称，应该不止一张表和文件

处理执行元数据在数据存储中保留，以便进行趋势分析。分析元数据可以发现 ETL 处理的瓶颈。可以保证数据仓库性能的可控性。同时也可以衡量数据质量。

异常处理

在执行过程中可能引起的数据纪录异常条件和可以采取的措施：

- 主题名称 可以是数据集市或关于批处理的描述
- 任务名称 执行程序的名称
- 异常条件 异常条件的标准设置
- 严重性
- 采取措施
- 操作员
- 结果

批处理调度

批处理是一系列要执行的 ETL 任务调度的集合。批处理的名称应该可以反映出其所属主题，任务执行的频率和任务中批处理执行方式是并行还是串行。

10.5 元数据标准和实践

元数据的一个值得投资的方面是元数据标准。很多组织试图在各种层次上标准化元数据。如果你对标准很感兴趣，比如说命名规则或域标准，你会发现这些由国会图书馆维护的标准非常有用(www.loc.gov/standards/standard.html)。此外，也提供其他标准组织提供的链接。在他们的网站上，你还可以发现以下链接：

- 元数据编码和传输标准(METS)：有关编码标准的描述，管理，和结构元数据的数字图书馆。
- 美国国家标准协会(ANSI)：美国非官方标准化和一致性评估系统组织。
- 标准化国际组织(ISO)：建立，发展和促进标准化的国际交流。

元数据所涉及的 ETL 处理不仅仅包括数值上的标准和惯例，还包括元数据存储和发布的方法论。连接组织和元数据存储的组织包括：

- Dublin Core DCMI（The Dublin Core Metadata Initiative）是一个发展元数据标准的开放性论坛。DCMI 主持周期小组活动和讨论会在世界范围内促进元数据标准和实践。可以在 www.dublincore.org 上找到有关 DCMI 得更多信息。
- 元数据联合会 Meta Data Coalition (MDC)成立于 1995 年，成员大约包括 50 个厂商和最终用户。 他们致力于研究不同数据仓库产品环境的元数据交换技术。MDC 还在建立和壮大元数据发布的一致性方法做出贡献。2000 年，MDC 与 OMG（Object Management Group）合并。
- 标准数据仓库元模型 标准数据仓库元模型（CWM）是 MDC 和 OMG 合并后的产物。本书有一章专门用来描述这个主题。在 OMG 网站上 www.omg.org/cwm 也可以获得有关 CWM 的详细信息。

建立标准

为了维护你的企业级数据仓库所有 ETL 过程中可管理的任务，数据仓库组必

须要建立标准，而 ETL 团队要按此实施。无论你是否会遵从上面那些组织的建议或遵从在本书列出的那些习惯，但从长远的角度上讲，你的组织最好要关注一下下列列出的标准：

- 命名习惯 一般组织都会在目前软件或数据库开发团队中有一套命名习惯。数据仓库也可以遵守这些习惯。事实上，当命名列的时候我们更习惯偏离这些标准，而是经常必须按照组织政策命名。所以，所有的组织政策要在文字上归档，并提供给数据仓库组。
- 体系结构 最佳的实践指导方针应该体现在 ETL 环境元数据中。在许多情况下，高级构架是在 ETL 团队成立之前就设计好的，或是由 ETL 供应商提供的。是否在数据仓库中执行 ETL 引擎或是在一个专门的服务器中？是否要有一个稳定的存储区？目标数据仓库是否要规格化等
- 基础构造 你的解决方案是基于 Windows 或 UNIX 主机系统或 AS/400？像这种组织的一些标准会影响到 ETL 产品或硬件配置的选择。一些 ETL 引擎只能在 UNIX 或大型机上运行，而其他的则可以在 Windows 上运行。你必须在 ETL 环境部件决定购买之前制定基础构造的元数据。一旦建立好 ETL 环境，就要把基础构造元数据归档，并和内部构造支持组一起维护该数据。

命名习惯

数据仓库环境中各个对象的命名习惯应该在 ETL 团队开始编码之前就建立好。表，列，强制约束，索引等命名习惯应该由 DBA 组或数据库管理员提供。而 ETL 团队要遵从这些规则。

如果你觉得现有的命名规则不适合你们 ETL 环境的话，你要把想要替换的命名规范记入文档，并提交给你们的标准委员会。一旦批准的话，标准委员会应该将现有的命名标准和新的 ETL 习惯相融合。

这部分我们提及的那些组织都没有对 ETL 处理提供一套命名的标准。如果你的 ETL 处理只是由一系列的 SQL 组成，而又没有明确提出的命名定义的话，那么在命名上遵从你们原来内部程序和名称标准。

大多数 ETL 工具的数据转换都推荐使用命名规则。不管是用哪种 ETL 工具，或是手工代码，都最好遵从命名习惯。你的工具可能经常变更，不止使用一种工具，但你每次更换工具时不会每次都建立一套新标准。我们在本书中并没有提及在转换上使用厂商们推荐的命名规则。但是，你也可以根据你自己的意图稍稍更改命名规则，以保证环境的连续性。

10.6 效果分析

维护 ETL 元数据的一个有利条件是它可以帮助进行效果分析。通过效果分析，你可以列出数据仓库环境中的所有可以改变特征。还可以分析这些变化所带来的影响，这些影响可能是数据仓库的任何组成部分。一个效果分析方案必须能够解答以下问题：

- 哪些 ETL 任务需要这些阶段表？
- 这些源系统表是否被元数据使用？
- 删除这些源系统列是否会影响 ETL 处理？
- 这些维度来自哪个源系统？
- 如果我们将数据类型从 VARCHAR(2000)转换到 CLOB 型，哪个 ETL 任务和数据仓库表需要修改?

使用工具设计 ETL 处理应该可以回答所有这些问题。不使用 ETL 工具，你需要维护电子表格来记录源系统所有表，列，到他们装载到数据仓库中的所有信息。每次更改一个 ETL 任务，就要手工更新电子表格。

10.7 总结

本章，我们回顾了数据仓库元数据的一些混乱现象，讲述了数据仓库元数据的重要性，然后将 ETL 元数据分为三类：

1.业务元数据：从业务角度描述数据的含义，从不同组成部分来完整业务的定义，源系统信息，数据仓库数据字典，和逻辑数据视图

2. 技术元数据：描述数据所反映的技术方面，包括数据类型，长度和数据血

统，和数据表组成。通过这些信息追踪系统详细清单，数据模型，数据定义，业务规则，ETL 任务定义，详细数据转换和批处理任务定义。

3. 处理执行元数据：描述 ETL 处理结果的统计信息，包含的测评指标包括成功执行行数，丢弃行数，装载时间。我们还在文中提出，在清洗和规范化过程中比较重要的元数据，包括筛选维度表，错误事件事实表，和审计维度表。所有的元数据都是由追踪执行结果表，异常处理表，和直接操作计划表组成的。

11.职责

在本章中，我们将讨论开发管理以及一个成功的 ETL 系统的管理。我们可以把这一章放在本书的开始，在种种 ETL 系统的职责彻底的讨论之前，但是我们考虑到将内容放在本书的最后，读者可以更好的理解如何高效的管理团队。

本章的前一部分关注于计划和领导问题，后一部分将深入到管理 ETL 系统的细节中。这些观点从《数据仓库生命周期工具箱》中发展而来。

流程检查

规划与设计：需求/现状 -> 架构 -> *实现* ->发布

数据流：*抽取* -> *清洗* -> *规格化* -> *提交*

11.1 计划和领导

在某些角度上，数据仓库和 ETL 处理就像其他的软件开发项目。当数据仓库团队建立的时候，通常需要 3 个专家。下面列出了在数据仓库项目初始阶段需要的一般角色，列表中包括了主要的角色和次要的角色（使用圆括号），在小型团队中可以兼任。

- 数据模型师（项目经理）。数据模型师必须接受过维度数据模型建模的专门训练，学习过维度模型的理论。
- ETL 架构师/程序员（DW 架构师）。ETL 程序员以及 ETL 架构师通常是 SQL 和数据库的专家。这个人负责建立 ETL 系统、数据仓库环境的技术框架，并负责设计物理的 ETL 流程。
- 应用专家（业务分析师）。负责搜集并文档化业务、分析和报表需求。这个专家需要设计前端的接口，以及数据仓库中的初始报表。这个职位通常称为商务智能专家。

当启动一个数据仓库项目的时候，通常是由一组专门的人来为整个企业应用的设计基础部分。如果没有一个完整的计划和方法论去构建应用的基础，那么后续的所有工作可能都是无效的。

拥有专门的领导

数据仓库是一个需要专门知识的复杂工程，这些知识并不为大多数 IT 管理者所掌握。最基本的，数据仓库需要一个专门的项目经理，他需要具有使用维度模型原则实施数据仓库的经验。如果你的数据仓库发展的很大，那么每一个组成部分和子部分都需要一个专门的项目经理。一个成熟的数据仓库必须有独立的来负责 ETL、数据建模和商务智能的经理，并且需要一个项目经理在总体上控制数据仓库项目中各个部门，以确保在各个域中实现一致的方案。

通过一个人来管理整个数据仓库项目是值得讨论的，但是我们强烈的建议为每个领域指定专家。不同的领域需要不同的技能，导致了某个人无法覆盖所有的知识。请牢记，个人强不过团队。

团队的力量永远强过个人，但是这不意味着可以通过投票来完成设计！决定设计适合采用专制方式，这样可以维持其一致性。

周密计划，逐步实施

当你从头创建一个数据仓库的时候，最困难的是去想象从现在的单个数据集市扩展成一个规模上超过公司的任何一个现有应用的主要的企业核心系统。但是这个图景通常被遗忘，因为数据仓库通常采用另外的方法实施：在开发下一个数据集市的时候，他们开始完成单个的业务流程或者数据集市，例如人力资源、或者渠道管理。

数据仓库架构师必须采用已知的数据仓库总线架构技术，该技术勾勒出了数据仓库的架构，让所有的结果数据集市按照我们本书中详尽描述的使用规范化的维度和事实按照整合的方式进行工作。

数据仓库总线架构的处理过程中包括了一个数据仓库总线矩阵，需要创建一个所有维度的列表，并将其和数据仓库中的不同的数据集市关联起来。总线矩阵可以帮助架构师在直观上发现哪些数据维度在数据仓库中不同的数据集市之间被共享或者需要规范化。一旦总线矩阵创建，物理的数据集市可以一起被创建。图 10.1 给出了一个简单的数据仓库总线矩阵的例子。

数据集市/维度	DIM_AGREEMENT_TYPE	DIM_CURRENCY	DIM_DATE	DIM_DEPARTMENT	DIM_ABSENCE_REASON	DIM_OFFICE	DIM_EMPLOYEE	DIM_POSITION	DIM_ABSENCE_TYPE	DIM_EMPLOYEE_RATING	DIM_COMPENSATION_TYPE
雇员月度快照		X	X	X		X	X	X			
雇员转换交易	X	X	X	X		X	X	X			
专门交易			X	X	X	X	X	X	X		
补偿月度概要		X	X	X		X	X	X		X	
费用月度概要			X	X		X	X	X		X	
补偿交易		X	X	X		X	X	X			X
费用交易			X	X		X	X	X		X	

图 10.1　数据仓库总线矩阵

正如某些维度在整个的数据仓库中被重复使用一样，在你创建 ETL 过程的时候某些 ETL 方法会被反复的重用。例如：你的 ETL 处理通常包含为某个维度生成代理键，那么生成代理键的代码可以被重用来生成数据仓库中的所有的代理键，不同的仅仅是参数。如果你有一定的软件开发背景，那么你一定听说过：一次书写，多次使用。这个格言意味着尽可能的重用代码。代码重用不仅仅节省了开发时间，也确保了代码间的一致性。

你需要一个高效的重用策略。要建立一个鼓励开发者分享想法和信任彼此的工作环境，下面的一些提示可以帮助你：

- 作为一个小组取得决策的一致：通过例会来讨论技术和功能策略，以一个小组来解决困难。
- 在共享代码的时候分享想法：ETL 开发不是一个竞技项目。共同工作、与团队中其他人分享案例。我们已经花费了大量时间解决了某个困难的问题，通过向其他人解释情况，那么解决的思路就会被别人掌握。在简

单的讨论过程中，你会惊奇于产生了多少好的想法。

- 使用资料库。许多 ETL 工具为了重用代码提供了资料库。请确认你的工具中包含了一个资料库，可以利用它在不同的加载方式或者开发者之间进行代码重用和共享。如果你还没有购买专门的 ETL 工具，至少使用一个源码资料库，例如 SourceSafe 或者 PVCS。在一个复杂环境中，你可能需要开发多个单独的资料库，但是作为一个虚拟的资料库进行管理。

一旦你的团队学会了共同工作，所有的代码方法存储在你的资料库中，那么开发的效率的提高是显而易见的。另外共同工作，也帮助开发者对整个项目有了一个整体的了解。要避免将开发者按照主题域来划分。随着经验的积累，每个开发者会成为他或她所开发领域的专家。这对于团队涉足新的领域是非常有益的。拓宽 ETL 开发者的视野提升了 cross-functional panning 并且提高了士气。

应该鼓励你的 ETL 团队尽可能的分享和重用他们的工作成果。确保在资料库中为共享的代码关联正确的元数据。元数据可以标识代码的用途以及介绍使用方式。但是对于跨操作系统和 DBMS 平台代码的重用要有一个正确的期望。

雇用合格的开发者

熟练的开发人员是企业的无价的财富。但是，单纯拥有技术不能表明它是一个专家。这些年，我们曾经面试了很多 ETL 开发员，他们懂很多工具内外的东西，但是却对维度概念，如层级映射表一无所知，开发员必须能够在接受最少培训的情况下迅速的掌握并应用一种新的技术。在我们面试的时候，我们会用很少的时间谈使用的工具的特性，而是把更多的时间用于技术和功能的问题如何解决。我们寻找那些聪明的、有成为合格的 ETL 团队成员的潜质的人，而不是那些只有技术技能的人。

在面试的过程中，专门问一些那些应试人不知道如何回答的问题。观察他/她如何回答。记住，这时候重要的不是答案是否正确，而是处理问题的过程。现实是 ETL 和数据仓库可能会非常的复杂，甚至非常的特殊化。无法像分析原子构成那样分析，否则，科学家就可以提供一个规范了。当你建立你的团队的时候，确保你的开发者有意识主动的追求技术和专业。他必须能够和你以及你的项目一起成长，能够接受新的技术和方法。

和数据库专家一起组成团队

ETL 主管的部分职责是考察你的企业中的所有源系统，根据已有的系统为你的团队选择合适的技能组合。如果你使用专门的 ETL 工具，雇用专门的数据库专家可能就不再是关键问题。但是，即使是采用最好的工具，在 ETL 开发中也避免不了撸起袖子写 SQL 的情形。

列出 SQL 编码的提示和技术超出了本书的范畴，在市场上有好几种介绍 SQL 的书籍，当然都是基于 DBMS 查询介绍的。单独的工具可能不能完全满足你的 ETL 需求。当你面试那些 ETL 团队成员的候选人时，确认他们有业务系统 DBMS 以及主要 ETL 处理使用的数据库的 SQL 经验。

每种 DBMS 源系统都需要你对其了解，并能够利用产品特殊的 SQL 进行实施。确保你的团队能够操作多种数据库，这样你的 ETL 工具就可以在处理中和本地的 SQL 代码无缝的集成，而不再把它留给应用程序。

不要试图拯救世界

ETL 系统仅是数据仓库项目的一部分，某些想法可能会超出你的控制。注意数据仓库不是，也不会是完美的。你需要面对脏数据，很多时候，你没有办法清理。我们的哲学是最好的数据清理的方法是暴露他们。由于数据难于分析，才会考虑建立数据仓库。要是你的 ETL 处理提交了大量以前从未暴露的数据，尤其是提交到数据仓库的时候。如果你对于保证最终数据的数据质量的请求被忽略，请保持耐心。一旦脏数据被提交，主管将不得不开始对异常进行检查，那样你将会欣喜地见证到数据质量的改变，并且获得你需要的支持。

强制推行标准化

对于一个大型的 ETL 项目，需要尽可能早的建立标准。没有标准，开发人员将写出不一致的 ETL 作业，导致对已有的代码的维护任务变得非常恐怖。ETL 团队必须标准化他们的开发技术，提供一个一致的，可维护的代码环境。下面的列表包含了需要标准化的了大多数 ETL 处理：

- 命名约定。在 ETL 处理中建立和强制推行对对象和代码元素的命名的规范。可以从利用你们现有的命名标准开始，并在其中增加那些 ETL 工具

厂商建议的规范。

- 最佳实践。为建立 ETL 过程建立文档，并按照最佳实践的过程实施。在你的过程中尽量的标准化一些东西，例如从一次失败的处理中恢复的最佳方式，或者通常转换开始的位置。本书为你提供了针对策略的完整的建议，包括：

  1. 生成代理键。如果你决定使用数据库，ETL 工具或者任何其他的机制来生成代理键，那么在整个 ETL 作业中保持一致。
  2. 查找键。你可以使用映射表，物理维度，或者使用准备区技术来将自然键和他们的代理键关联起来。选择一种，并坚持使用。如果你混合使用这些技术，那么维护这些方法将是一场恶梦。
  3. 提供缺省值。一些方法和值可以被接受意味着使用默认的缺失值。记住在数据仓库中要对缺失值或者 NULL 值进行认真的处理，因为这些值可以导致报表中空的列或者行，因为一些数据库中在他们的索引中不包括 NULL 值。最好检查进入记录的 NULL 值，在 ETL 过程中使用实际的值，如单字节的 ？来代替。

监控、审计和发布统计信息

ETL 统计对于任何使用数据仓库的人来说都是有效的。如果你的数据仓库有专门的 Web 站点（应该具有），这就需要确保其中包括了对你的 ETL 过程的日统计信息。用户通常希望准确的知道什么时候一张表被装载，或者是否有任何的记录被拒绝。大多数 ETL 工具能够自动的生成加载统计。确保你的工具能够在每天数据加载后自动的发布需要的统计信息。

交叉参考：在第 9 章中给出作为元数据策略组成部分的统计元素列表，他们应该被发布。

维护文档

你的 ETL 处理的 那些文档内容对于数据仓库团队来说是无价的财富，并且已经成为财务或者通常报表在数据仓库建设阶段的必要资料。即便是那些纯逻辑数据映射规范，虽然仅仅只有开发人员才知道从源系统到数据仓库进行了如何的

处理。ETL 团队有责任严格的控制并维护数据仓库中每个元数据的文档。一些文档可能以元数据的形式存在，但并不是所有的文档形式都被认为是元数据形式。元数据作为一个复杂的实体已经在很多书中对其内容进行了介绍。无论你怎样去分类，一些文档内容都需要维护和发布。通常，很多工具设计了捕获处理过程的描述信息的字段，但是这些信息往往不存在。这样无可避免的，你需要以 Word，Excel，PPT 的形式提供解释你的 ETL 过程的文档。请使用如 SourceSafe 或者 PVC 等版本控制系统来维护你文档的完整性。

提供和使用元数据

元数据对于 ETL 过程的重用异常重要。事实上所有的 ETL 工具都能够捕获和利用元数据。不要不负责任的让你的团队在创建 ETL 处理的时候不建立元数据。每次当你创建新的 ETL 过程的时候，请牢记如果不能通过元数据获取，那么它就不存在。这在创建一个数据仓库环境的时候尤为重要。如果你不通过元数据公开你的工作，那么团队中的其他人就可能要从头创建那些你已经创建和测试的处理过程。

通常，ETL 工具资料库通常是元数据的资料库，有些 ETL 工具能够从其他的工具，如数据建模或者报表工具中自动的获取元数据并对关联的元素进行影响分析。一些商务智能工具可以利用 ETL 资料库，将元数据和数据仓库的用户接口进行集成。如果 ETL 环境中的元数据被发布，那么它们必须被维护，或者用工具可以将最新的信息发布给他们的用户。

保持简单

如果你相信存在采用更简单的方法去做什么，那么通常这种方法都存在。当你创建 ETL 过程的时候，请时常停下来回头从设计的角度审视你的工作。设计够直接么？设计中是否有一些可以避免的复杂情况呢？你的处理过程越复杂，那么维护的困难性就越大。通常，一个复杂的 ETL 设计几乎不可能再被修改。当业务需求或者源系统改变的时候，你的 ETL 作业必需足够的灵活，以适应变化。我们曾经接触过一个项目，ETL 作业是那么的复杂，没有人确切的知道它做什么，所以没有人能去修改它。费尽周折，最后我们被请来做反向工程，文档化并将其重新设计为一个改进的、可维护的过程。

彻底优化

对于如何精心的设计 ETL 将源系统转化成数据仓库中的有用的信息，在现实中没有任何的限制。但是作业执行多长时间的限制确实是存在的。ETL 开发人员所面临的挑战是在指定的加载窗口中完成抽取、清洗、转换和加载数据。加载窗口是指每个晚上运行 ETL 过程的时间。通常，数据仓库在加载窗口中是不可用的，因此将加载窗口变得尽量小就成了一种挑战或者压力。

11.2 管理项目

ETL 处理是数据仓库的关键过程，直到今天，它还是没有获得应有的重视。在早期的数据仓库中，主要关注的是前端工具，然后随着数据量的增加，维度数据建模成了新的热点。当数据仓库成熟度达到下一阶段时，ETL 将最终成为关注的焦点。

绝大多数设计人员同意在整个数据仓库项目中的 ETL 处理要至少占用 70%的工作量。ETL 团队来建立这些将数以亿计的记录从分散的系统中导入到一个一致的用户信息库的强健的处理过程，对这样团队的成功管理是对技术能力和执行能力的挑战。管理 ETL 团队需要奉献精神以及管理方法。

ETL 主管的职位建立是为了帮助数据仓库项目经理分担 ETL 管理职责。这个职位也给了业务接口人以信心：ETL 团队可以建立一个受控的、高效的环境来向数据仓库加载干净的、一致的数据。本章中包含的任务应该被数据仓库项目中的每个成员仔细阅读，以确保他们理解 ETL 处理并不是数据仓库项目的副产品，而是将整个项目连在一起的粘合剂。对于 ETL 主管，本章的内容提供了保证你的 ETL 项目成功的基本知识。

ETL 团队的责任

从最根本上说，ETL 团队的责任是从源系统中抽取数据，执行数据转换，将转换后的数据加载到目标数据仓库中。进一步，要得到最优的 ETL 结果，ETL 团队的责任包含下列的任务：

- 定义 ETL 范围

- 执行源系统数据的分析
- 定义数据质量策略
- 和业务人员一起工作获取和记录业务规则
- 开发和实现实际的 ETL 代码
- 创建和执行单元以及 QS 测试计划
- 实现产品
- 进行系统维护

要高效的管理你的团队完成上述的任务，我们结合这些任务的概要列出了一个实际的项目计划，并描述了针对每个任务的管理职责。

项目定义

尽管 ETL 处理仅仅是数据仓库整个生命周期中众多组成部分中的一个，但它是整个数据仓库的核心。同时 ETL 处理也是最难管理的一部分。当用户在项目初期阶段开始看到结果数据的时候，你将面对需求变化的冲击。假如没有一个合理的项目执行计划和变更管理策略，冠以 ETL 处理将是不可能的，一个永远没法完成的任务将会将项目推到失败的边缘。当你开始定义你的项目的时候，请牢记如下的准则：

- 为了一个顺畅的处理，ETL 的管理必须和数据仓库生命周期中的其他组件紧密合作。从规律上来说，数据仓库永远没有真正的结束。当一个新的需求产生，建模团队、ETL 团队和报表团队必须一起高效的工作来完成任务、达成目标。本章列出的步骤可以在不断的增加到数据仓库中的不同主题域中重用。正确选择本章列出的方法对于正确管理这些迭代的过程非常关键。
- 在估计工期和定义范围的时候正视现实。不要让数据建模人员或者不了解技术的业务接口人员对 ETL 工作的时间表作出不正确的决定。使用本章中的项目计划作为蓝本，并确认你的业务用户和接口人充分理解在加载数据仓库中包含的那些内容。

- 确保 ETL 团队在数据仓库项目启动会议中的活跃性。这样的会议适合将你的 ETL 团队介绍给关键的业务用户并讨论 ETL 的角色、目标和责任，以及时间表。创建一个鼓励合作的环境，这个会议将帮助与会者理解项目需求，并给你机会调整预期值。

项目计划

什么是计划？计划是一种活动、过程或者安排的方法。它是要完成的任务单。其目标是实现某种观念、想法、项目或者开发。因此，计划是帮助你达成期望的某种方法

--- Earl Prevette

ETL 主管的最终目标是成功的管理 ETL 的处理，并将处理集成到生命周期中的其他阶段。你可能假设管理 ETL 处理和管理其它实施是一样的，但实际上差别相当大。在本节中，我们将介绍那些帮助我们获得成功实现的方法。另外我们会揭示许多你可能会面对的障碍，并提供一些降低这些风险的建议。

开始定义项目计划中的可替换部分的首要条件是，你需要完成一些家务管理职责。这些任务包括决定你 ETL 工具的选择，以及组织你的 ETL 团队。

这两个任务的先后顺序非常的重要。你可能会将选择你的 ETL 工具放在组织团队之前。这样将使你可以招聘那些对你选择的工具熟悉的人。

选择工具

如同在第一章中所介绍的，ETL 主管必须在购买 ETL 产品和手工创建 ETL 处理中做出选择。对于每一种选择都会有很多的理由。但是，从企业数据仓库的成功性和管理接口人员的期望来考虑，我们认为没有时间去手工创建，即使是考虑到数据仓库的迭代属性。无论再小的项目也可以从专门的 ETL 工具的转换可重用性中获益。这些工具的已有的现成功能将会花掉几个月的时间去设计，更不用提作数据转换处理的实际编码。通过这些工具可以减少开发时间，使其可以作为任何的数据仓库项目的解决方案。

此外，ETL 工具是专门设计来解决那些当前甚至未来的问题。对于购买还是开发的问题，我们听到的所有常见的理由是程序人员懂 SQL，为什么还要学习一

种新的工具，因为他们都可以转移数据。当我们听到这个问题的时候两个类似的问题浮现眼前：首先，如果你仅仅知道的工具是锤子，那么你周围的所有的东西都会被认为是钉子，安装螺丝钉将会变得非常困难、费力和随意。第二个是秘书没有时间学习文字处理，因为他们总是忙于打字。这两个问题听起来多么荒唐，但这和不训练你的 SQL 程序员去学习利用专用技术的 ETL 工具来完成任务何其相似。

为了帮助决策，我们建议你记录你的工具选择标准。选择一些诸如吞吐性能、易于修改、厂商支持等 POC 因素，然后针对这些工具完成 POC（包括手工开发的比较），最后选择符合你标准的产品。基于 POC 的结果，你可以决定是开发还是购买产品。如果你决定购买，你需要确认哪种产品符合你的要求。

为项目配置人员

管理 ETL 过程的一个关键因素是建立一个高水平的团队。你的团队成员必须拥有必备的技能来完成交给他们的任务。一个认证训练的团队是你成功地关键。确保你的所有团队成员使用公司文化，并且可以共同工作是非常重要的。

在数据仓库建设到达当前水平之前，所有项目的职责通常是由少数的数据仓库专家完成。这些核心的专家和业务用户交流，记录需求，设计数据模型，加载数据库等等。当数据仓库项目进一步发展后，我们发现每个特殊的任务都需要一组不同的技能，没有一个人可以成为所有问题的专家。

ETL 团队角色和职责

对于 ETL 团队角色的组成是需要考虑的任务。如果你拥有相应的知识，你可以雇佣和培训你内部的员工。否则你需要和招聘人员一起寻找合适的专家来建设你的团队。

下面的列表介绍了我们建立一个最佳的 ETL 团队所需的基本的角色和职责。

为每个角色都雇佣一个人是最理想的。但是，根据项目的实际情况，考虑到条件的限制，更现实的是让人们相互承担一些彼此的职责。记住当你组成一个项目团队的时候，你的主要目标是确保所有的职责都可以被履行。你没有必要为每个角色制定一个专门的人。

- ETL 主管。这个人负责 ETL 团队日常的管理，以及负责和 ETL 相关的数据仓库的运行维护，他对数据抽取、转化和数据仓库中的加载处理，以及测试和质量保证负有责任。他需要为 ETL 环境制定标准和流程，包括命名规范以及最佳开发和设计的训练。
- ETL 架构师。主要的职责包括设计 ETL 环境的架构和底层结构，为 ETL 开发团队设计逻辑数据映射。这个架构师必须对业务需求和源业务系统有很好的理解。ETL 架构师负责为团队解决复杂的技术问题，以及将 ETL 过程迁移到生产环境。
- ETL 开发员。这个人负责建立事件的 ETL 处理。在实际编码以前，他和 ETL 架构师合作解决任何规范中有歧义的地方。开发员负责开发专门的 ETL 方法，并测试他们的可靠性，确保其符合业务需求。通常数据仓库项目中都会安排多个 ETL 开发员。
- 系统分析师。负责业务需求定义，以及在整个数据仓库生命周期中定义这些需求。他和数据仓库团队以及业务人员合作紧密。
- 数据质量专家。数据仓库的质量包括了内容的质量以及数据仓库中的信息的结构。数据质量专家通常向 ETL 主管汇报工作，有时候也直接向数据仓库项目经理汇报。数据质量专家的主要工作是帮助系统分析师和 ETL 架构师来保证业务规则和数据定义在整个 ETL 处理中是延续的。
- 数据库管理员（DBA）。DBA 的主要职责是将数据库的逻辑设计翻译成物理的结构，并维护物理的数据库。而且，DBA 需要和 ETL 团队紧密地工作，确保新的处理不会和已有的数据冲突。在某些环境中，一旦 ETL 过程迁移到生产环境，就归 DBA 管理。
- 维度主管。他负责定义、创建和发布一个或者多个规范化的维度给整个数据仓库社区。这是一个需要权力集中的职位。规范化的维度需要打上版本标签，同时为所有的事实数据提供者客户所使用。可能在一个企业内部有多个维度主管，原因是可能每个维度可能独立性很强。在任何情况下，一个维度只能有单一的维度主管。
- 事实表提供者。事实表提供者拥有某个专门的事实表。在一个规范化的

维度环境中，事实表提供者周期性地接收并更新维度主管那里下发的维度，将事实表中的自然键转换成为维表中的代理键，然后将事实表发布给相应的用户群。

ETL 项目团队人员选择

有一句格言：你的下属决定了你的水平，这句格言揭示了关键业务系统，如数据仓库中 ETL 处理中一个重要的特点。一个聪明的避免项目失败的方法是建议一个专家团队来实施。这一节将对建设你的 ETL 团队的给出一些建议。

和新人一同工作

有总比没有好，在建立你的 ETL 团队的时候你需要把眼光放在公司以外。通常，企业总在寻找最佳的候选人。但是就像数据仓库需要完全可靠的信息给他的用户一样，你需要提供精确的需求给招聘人员。在提供工作描述的时候尽最大可能的细致，这样确保你见到的候选者拥有你希望的技能和工作习惯。让招聘人员了解你的环境的具体信息，尤其是你的开发语言，供应商以及数据库。另外描述你的团队的文化，以及你需要什么样的人。提供的信息越详细就越有可能找到符合你条件的候选者。

从专门的数据仓库公司招聘的人往往拥有数据仓库行业的知识以及所支持的工具的技能。在简历转发给你之前这些人应该是经过初步筛选的，限制需要你评估的简历和人员的数量。其它的工作应该会占用你的更多时间，面试新人的时间需要花的值得。

内部雇用与外部

从内部或者外部资源创建你的团队有很多好处。从内部雇用的好处如下：

- 从企业内部寻找和雇佣成员的最大的好处是这个内部的人员已经对企业内部的结构和 IT 系统非常了解。他们知道每个人负责什么；哪个人可以回答问题；如何按照规矩做事。如果你够幸运，这个人可能已经拥有了你团队角色所需要的技能。如果一个人没有需要的技能，但是有潜力并愿意接受培训，那么他就是一个非常值得考虑的候选人。
- 为有热情的内部的员工提供学习新东西的机会将保持他们的竞争力以及

满足他们的需要。

- 内部雇用在经济上考虑非常有利：这比通过招聘人员、花钱安置、退还面试成本、花钱重新安置等经济的多。

如果你从外面雇用，你雇用的人应该拥有你在寻找的技能，以及在不同企业文化中使用这些技能的经验。这些经验远远超出了他的表面价值。经验可以给你的团队增加价值，节省你的时间和金钱。

选择团队成员

一旦你获得了一把简历，那么我们建议你在带他们来之前先和候选人安排一次电话面试。在电话里问一些关键的问题，直接考察他们的沟通技巧，以及对问题的理解程度。

对通过电话面试的候选人应该进行面对面的面世。你的候选人应该接受 ETL 主管、技术开发人员以及功能分析师的提问。让候选人同功能分析师以及技术人员见面将使你能够衡量他们的熟练程度。我们有一些不愉快的经历，团队成员技术很熟练，但是没有甚至不能领会功能要求。他们的无能导致其他的团队成员需要额外花时间来确保他们的工作符合业务要求。

你需要坚信不疑，你的团队的所有潜在成员需要有充分的 ETL 处理设计的知识，充分地掌握所需工具，以及对业务流程的正确理解，领会功能需求。和团队其他成员的协同能力也是关键，在你面试的时候，一定要询问他以前项目中的团队工作方式。

在面试中，对你和招聘人员最基本的要求是不仅要充分了解你们要找的角色，而且要知道问哪些问题。图 10.2 时一张面试调查表，提供了面试过程中的问题。使用这个问卷可以确保你的候选人的知识满足你的角色要求。如何回答这些问题则超过了本书的范畴。

简单调查表

分析
1. 什么是逻辑数据视图？对于ETL团队它意味着什么？
2. 数据仓库项目中搜寻数据阶段的主要目标是什么？
3. 如何确定档案系统？

架构
4. ETL过程的4个基本数据流程步骤是什么？
5. 数据集结区可以接受的数据架构是什么？每个得简单描述。
6. 在ETL过程中，出于安全的考虑，什么时候数据要设置在磁盘上？

抽取
7. 描述从异种数据源中抽取数据的技术。
8. 处理ERP源数据的最佳方法？
9. 描述数据库自然连接与通过ODBC之间的差别。
10. 描述三种缓慢变化维度。

数据质量
11. 四种主要的数据质量检查方法是什么？每种说出一个执行方法。
12. 在ETL的每个集结阶段是否都要对数据质量进行评估？
13. 数据质量检查部分的重要交付物是什么？
14. 在数据仓库中数据质量如何量化？

建立映射
15. 什么是代理键？解释代理键管道如何工作？
16. 为什么在ETL处理中日期要特殊对待？
17. 解释一致化维度中三个基础提交步骤。
18. 给三种基础事实粒度，并描述每一个的ETL 方法。
19. 当维度记录和单个事实表关联时，桥接表如何提交到分类的维度表？
20. 延迟到达的数据如何影响维度表和事实表？他们的处理方法是什么？

元数据
21. 描述ETL 元数据的不同类型，提供每个类型的一个样例。
22. 为捕获操作性元数据提供可行的机制。
23. 描述业务元数据和技术元数据的方法。

优化/操作
24. 陈述数据仓库中表的主要类型，以及他们装载的顺序。
25. ETL 支持模型四个层次的特性都是什么？
26. 你认为ETL 处理性能的瓶颈一般都是什么？
27. 描述如何减少大ETL任务的装载时间？

实时ETL
28. 描述实时ETL的框架。
29. 解释不同的实时方法，以及他们如何应用到不同的业务需求中。
30. 列出实时ETL可能面临的风险，并描述如何克服他们。

图 10.2　面试问卷样例

构建和保持一个成功的 ETL 团队

一旦你建立了你的团队，你作为主管的主要职责就开始了。保持一个一流的团队对你来说是一个最大的挑战。对那些 ETL 高水平人员的需求是很大的，招聘人员会毫不犹豫的从你的眼皮底下将他们猎走。我们发现要保留你的项目中主要的 ETL 开发人员和架构师的最佳方式是保持技术挑战性。根据我们的经验，一个无聊的技术专家最有可能离开。安排项目使你的团队成员感兴趣和兴奋是你的职责。

ETL 开发人员完成的任务并不简单。他们的职责是要将无组织的、分离的数

据转换为一致的有价值的信息，是一个令人激动，但很多时候令人精疲力竭的工作。不要忽视它们，关注他们的需求，知道是什么让他们奋发激情。我们曾经和一些开发人员一起工作，他们非常喜欢清理数据。他们乐于从垃圾中找到一致可信的数据。而其他人却不能忍受这一点，他们认为既然数据如此重要，那么他们从源头就应该是干净的，这些开发人员不可能胜任这样的挑战：将几乎无法用 SQL 操作的数据从可怕的复杂的模型转化到简单的维度中。另外的一些开发人员却喜欢和时间赛跑，如果一个 ETL 处理应该需要一周开发，他们却疯狂的工作，在短短几天内将他完成。你作为主管的部分职责是了解你的开发人员的类型，让他们接受挑战。

如果团队成员渴望能够接受更多的责任，授权好了！你的职责是发现每个人的期望，尝试满足他们，另外，你需要提供团队成员需要的培训，让他们成为所从事工作的专家。如果你不能让你的队员进步，那么他们将会离开项目，跑到更具有挑战性的环境中。

和你的下属保持和谐的一种高效的方法是周例会。ETL 环境是变动非常频繁的，超过一周没有得到你的团队的反馈可能会对项目造成危害。这个具有双重目的的会议可以确保项目的成员达到他们的目标，同时你也达到你的目标。给他们授权，鼓励他们做合理的决定。而且，你应该建立这样一个环境，团队成员可以表达他们的意见，交流开发需求等等。他们必须能够信任他们的 ETL 主管并且接受他们矫正的错误。一个被很正确管理的组员应当是一个满意的人。

外包 ETL 开发

在我们写本书的时候，IT 外包是一个热点。尽管实际的数字小于预想的值。在 2003 年，美国在 IT 预算上的投资是$119 亿，仅仅有不超过 5%的 IT 被外包出去。在最近的报告中，一些混淆视听的外包收入被人辨识出来，管理外包项目包括了额外的通讯、到国外的出差，被调整的期望和交付并没有出现在原来的财务报表中。我们不是说外包不是一个坏主意，但是我们需要提醒大家，外包是一个距离数据仓库非常远的概念。

数据仓库必须和数据源 de jure，并且管理的优先性改变，以及最终用户一致。正如我们说过多次的，数据仓库不是一个项目（提供规范和最终交付），而是一

个不断发展的过程。数据仓库开发任务是迭代和变化的。事实上，这也是我们喜欢维度模型的一个原因；它是一种弹性十足的架构，能够适应新的要求以及范围的变化。

由于这些原因，我们不看好将很多的数据仓库开发任务外包给异地的团队，不和源数据提供者和最终用户保持沟通。记住数据仓库是一个决策支持系统，对其成功的判断标准只有一条：是否能够有效的支持决策。

ETL 系统的开发体提供了外包的机会，前提是你有特殊可以被完整的定义的转换。

项目计划指南

注意你已经选择了你要使用的工具，并组成了你的 ETL 项目团队，你现在可以专心致力于 ETL 项目计划了。在本节中，我们提供了一个可以供任何 ETL 团队使用的详细的项目计划。

由于数据仓库建设具有迭代的特性，因此项目计划应该可以在数据仓库的任意阶段复用。坚持使用这些指南可以确保整体的规范化，注意不要忽略其中的任何步骤。

每一个步骤将在本章相应的部分作详细介绍。管理 ETL 处理的主要步骤在图 10.3 的项目计划中列出。

任务	任务描述	子任务		角色
1	建立开发环境	1	搭建硬件环境	DBA
		2	安装软件/工具	DBA/ETL架构师
		3	创建测试环境和文档模版	ETL管理员/ETL架构师
2	业务需求分析	1	回顾现存的数据模型文档	ETL架构师/系统分析人员
		2	定义，并将ETL规则文档化	ETL架构师/系统分析人员
		3	分析源系统	ETL架构师/系统分析人员
		4	定义项目可能的范围	ETL管理员
		5	获得用户信息	ETL管理员
3	设计逻辑数据视图	1	回顾数据仓库数据模型	ETL架构师
		2	回顾业务逻辑	ETL架构师
		3	分析源系统	ETL架构师
		4	创建逻辑数据视图文档	ETL架构师
4	数据质量策略	1	定义数据质量规则	ETL管理员/数据质量专员
		2	将数据缺陷记入文档	ETL管理员/数据质量专员
		3	设计缺陷修改	ETL管理员/数据质量专员
		4	确认清洗逻辑	ETL管理员/数据质量专员
		5	将规则纳入逻辑数据视图	ETL管理员/数据质量专员
5	建立物理ETL处理	1	回顾逻辑视图	ETL开发人员
		2	创建简单维度的装载进程	ETL开发人员
		3	开发复杂SCD-2维度进程（历史）	ETL开发人员
		4	开发复杂SCD-2维度进程（增量）	ETL开发人员
		5	开发事实表进程（历史）	ETL开发人员
		6	开发事实表进程（增量）	ETL开发人员
		7	自动化进程	ETL开发人员

6	**测试ETL流程** 一单元 一质量保证（QA） 一用户认可（UAT）	1	创建测试环境	DBA/ETL架构师
		2	创建测试计划和脚本	系统分析师
		3	装载测试数据（历史的＆增量的）	ETL开发者
		4	执行单元测试脚本	系统分析师
		5	验证数据质量控制	系统分析师
		6	验证数据装载	系统分析师
		7	验证业务规则	系统分析师
		8	获得认可	ETL管理者
7	**ETL部署**	1	创建生产支持文档	ETL架构师
		2	创建失败恢复过程文档	ETL架构师
		3	创建生产环境	DBA/ETL架构师
		4	装载历史数据	ETL架构师
		5	为增量处理初始化ETL调度	ETL架构师
8	**数据仓库维护**	1	为已知的问题开发审计报告	ETL架构师
		2	定期查看ETL日志来确保一致有效的装载	ETL架构师

图 10.3　ETL 项目计划

建立开发环境

要执行完整的数据分析和开始开发任何源系统的 ETL，最佳的方式是让 DBA 团队给你建立一个开发环境。使用一个独立的开发环境保证了数据分析和 ETL 的开发不会受到生产交易系统的影响。一旦环境建立，ETL 架构师和 DBA 团队可以一起工作，安装必要的软件，以及执行数据分析和进行开发所需要工具的安装。

在项目的开始阶段，要记录创建开发环境的所需的步骤。从本文中学到的文档标准可以最大化降低你未来犯错的机会以及最小化后续增加系统时带入的风险。

业务需求分析

尽管在数据建模阶段通过分析已经记录了很多的业务规则，但是 ETL 架构师的职责是实现这些规则。通常，ETL 架构师和系统分析师会审阅所有存在的文档，

然后和数据建模人员一起讨论遇到的问题。

ETL 架构师和系统分析师必须对源系统和系统内的数据有非常全面的了解，这点非常重要。一定不要压缩完成对这些分析的时间，并且要记住，在没有完成对所有的源系统进行彻底的分析之前，不要开始创建逻辑数据映射。很平常地，根据涉及的范围，ETL 架构师要和系统分析师、数据建模人员、源系统 DBA 开多方会议来讨论源系统的细节。这些讨论的目的是寻找创建逻辑数据映射以及最终编码的业务规则。

图 10.4 是搜集和记录业务规则，以及数据缺失的一个模版样例。这个表格可以分开用于跟踪数据清理或者 ETL 的细节，甚至两者。我们将这两者整合在了一个模版中，原因是为了一个一致的集成的解决方案，ETL 架构师通常需要同时跟踪业务规则和数据清理转换。这样将这些规则和转换详细地记录就变得很重要，不仅仅单纯是为了编码，更多的原因是这些文档是单元测试、系统测试、QA 测试以及用户验收测试的案例的基础。这些元数据将来还会被用来做最终用户的培训和程序文档。

业务逻辑与数据缺陷						
编号	业务逻辑与数据缺陷	位置（源数据清洗，ETL或两者都有）	源数据清洗	ETL	审计报告	报告发送
1	员工ID必须唯一	ETL	N/A -没有发现不唯一的员工编号	ETL处理会检查不唯一员工编号，如果发现相应的记录就会被作为编号处理。	报告包括PL_ID, FULL_NAME, HIRE_DATE和DEPARTMENT和所有的复制的员工编号记录	职员系统团队
2	当状态为“激活”时，标志没有设置为“Y”	源数据清洗	数据源团队一定要明确这些产品是否激活，对源数据进行适当的调整。	N/A -增加业务逻辑之后，将不会发生	N/A	N/A
3	E-mail地址必须唯一，但是在源系统中存在多个E-mail地址	ETL和源数据清洗	数据源团队必须要清洗多个E-mail地址，并设定业务逻辑组织多个E-mail进入源系统。但这个时候数据源团队没有足够的权限来执行这个业务逻辑。	由于这个时候数据源团队没有足够的权限来执行业务逻辑，必须要建立ETL处理去除重复而不是装载到数据仓库中，并将生成的报告提交给合适的团队。	报告包括多个E-mail地址的EMAIL_ADDRESS, EMPL_ID, FULL_NAME和DEPTARTMENT	系统管理团队（负责分配E-mail地址）
4	部门的关闭标志设为“Y”，但没有结束日期。	源数据清洗	数据源团队一定要明确该部门是否真的取消，并设置了结束日期。要设定业务规则。	N/A一作为添加业务逻辑的结果，不会再发生。	N/A	N/A
5	用户将产品进行分类，在源系统中上报。将这些分类引入到数据仓库中。	ETL	N/A -不需要对源数据清洗	用户给ETL团队提供产品分类列表。在集结表中生成装载到产品维度中，这样比较容易生成产品聚合的报表	N/A	N/A
6	在部门之上的产品编号必须唯一	ETL	建立产品列表，并设定业务逻辑防止不唯一编号进入数据仓库中。但这个时候数据源团队没有足够的权限来执行这个业务逻辑。	由于这个时候数据源团队没有足够的权限来执行业务逻辑，必须要建立ETL处理来明确不唯一的产品编号，防止他们装载到数据仓库中。	报告包括PRODUCT_ID, PRODUCT_DESCRIPTION和DEPARTMENT_NAME	职员系统团队

图 10.4　业务规则和数据检测跟踪表

理论上，重要的内容都要被记载。不幸的是，虽然实际项目中团队刚开始创建文档的时候有很好的命名规范，但是当发生变化后他们却很少回头进行更新，请不要这样做。保持文档实时更新到最新的状态对于一个项目的成功至关重要。不断更新的文档是对将来系统增强或者对下一阶段的工作进行影响分析的基础。而且，当前的文档确保了你把握着仓库的数据继承关系。维护你的文档可能需要时间和努力，但是总比你回过头在 ETL 处理，业务规则或者事实后的数据中寻找发生了什么改变要强的多。无需太多的想象就可以意识到浪费时间、增加成本以及纯粹的挫折都可以通过事先的计划，在改变 ETL 处理的同时修改你的文档来避免。

定义 ETL 项目的范围

定义范围包括决定和用文档记录在 ETL 处理的每个阶段中包括什么内容，以

及相关联的主体域、业务规则、转换和清洗策略。另外通常还包括在这个阶段中不包括那些内容。记录每个阶段的范围，并让业务人员查看并签字确认可以有助于你的管理，避免范围的变化。

在定义每个阶段的时候要现实一点。虽然你的用户相信你能实现你的承诺，但他们更喜欢经常在整个阶段中随时改变或者增加新的东西。对范围的改变必须通过谈判以及划分优先级，将低优先级的变更放在后续的阶段。在最终完成的文档中将可能改变范围的变更考虑在内。

在用文档记录业务规则、定义范围之后，请和用户一起浏览整个文档，并获得用户的签字确认。管理范围涉及的技术将在本章的“管理范围”中介绍。

设计逻辑数据映射

为了使设计逻辑数据映射更方便，ETL 架构师必须重新察看数据仓库数据模型和所有的业务规则文档。如果有遗留或者新的问题，需要召开额外的会议来获得答案。如果在业务需求分析阶段搜集的业务规则缺少细节信息，ETL 架构师需要对源系统进行手工的分析。一旦所有的问题都已经解决，那么 ETL 专家就可以着手创建逻辑数据映射文档。

交叉参考：第三章包含了创建逻辑数据映射的额外细节信息。

定义数据质量策略

除了能够友好地进行查询以外，数据仓库可否接受的主要因素是数据是否一致和可信。因此定义数据质量评估标准并定义一种策略是 ETL 处理的重要部分。

ETL 主管和数据质量专家共同负责定义数据质量规则。他们的任务是分析源系统的质量，并记录所有确定的数据缺失值。这个工作不仅仅确保了最大限度的清洁的数据进入数据仓库；更令源系统从数据质量考察中获益。这些分析可以发现源系统应用中的不足，为源系统的系统管理员提供更正的机会，避免了将来的数据缺失。

清洗数据的可选方式通常分成两类：

- 在源系统中清洗
- 在 ETL 中转换数据

在源系统中清洗数据无疑是最希望的和有效的选择。但是不幸的是，一方面由于资源的限制这个方案不可行，另一方面由于交易应用开发周期的复杂性和周期的原因及时性得不到保障。另外，数据清理通常要包含解决数据缺失的更正动作。

在所有的细节和最终期限确定并被认可以后，记录每个数据清洗问题的细节以及相关的清洗方案。创建一个项目计划来跟踪那些在源系统中或者 ETL 处理中清洗的问题，这可以帮助你控制用户对于数据缺失问题的期望。

我们建议和负责源系统数据清洗的管理员一起召开周会。通过这些会议可以回顾和更新项目计划，使计划能够准确的反映进度，并就新的问题进行讨论。这些会议将决定数据清洗的可行性，并对新的清理问题的解决策略提供一个取得一致意见的场所。确保源数据的清洗能够按计划进行，因为你的数据仓库依赖于被清洗的数据。花点时间查询源数据库来确保清洗工作的成功是一个好主意。

尽管数据在源系统中被清洗，但是源系统的所有者却不可能增加业务规则来保证数据不再变脏。所以最好由系统分析师和 ETL 架构师一起决定 ETL 代码是否需要阻止脏数据进入数据仓库。ETL 代码可以根据业务规则选择拒绝或者转换脏数据。

如果 ETL 代码使用了预防策略，无论是通过排除还是转换，ETL 处理最好能够提供审计报告来捕获和生成脏数据报表。这样的报表有助于对源数据的连续清洗，并提供了数据仓库中的干净数据到其源的关联。这些元数据还可以作为审计用途，可以用它来跟踪数据和其源的差异，从而判断数据拥有者的责任。请确保获得用户签名的业务规则和 ETL 使用的数据清洗逻辑。

创建物理的 ETL 处理

一旦数据分析完成，业务规则和逻辑映射就是最后的，ETL 架构师将和指定的 ETL 开发者一起预演整个逻辑数据映射。这个预演确保了 ETL 开发人员在开发编码之前就能够理解完整的需求。ETL 开发人员的职责是将逻辑数据映射转化为物理的 ETL 方法。无论是写 SQL 脚本还是使用专门的 ETL 工具，这些方法必须经过开发和测试的，在他们 ETL 架构进行迁移之前，结果数据必须得到开发人员的验证。

一旦同时有多个方法同时交给开发人员时，ETL 架构师通常需要准备 ETL 构建顺序文档作为开发指南。如图 10.5 所示，文档中包含了一个需要加载表的列表，创建方法的起点位置，以及在方法实现中需要注意问题的注释。在项目的初期阶段以及对于那些你团队中的新人，此文档非常重要。

表名	ETL任务创建顺序	描述
d_SHIP_TERM_FLAG	1	直接从平面文件中导出
d_DATE	1	直接从平面文件中导出
d_DEPARTMENT	2	从数据库中导出，带有约束
d_PRODUCT_TYPE	3	
d_VENDOR	3	
d_CURRENCY	4	
d_ORDER_TERM_FLAG	4	
d_REGION	4	
d_OFFICE	5	
d_STORE	5	
d_SHIP_TYPE	5	
d_CLIENT	6	
f_CLIENT_ORDER_TRANSACTION	7	
f_CLIENT_MONTHLY_SNAPSHOT	7	

图 10.5　ETL 创建顺序文档

测试 ETL 过程

绝大多数系统的生命周期理论包含了 3 个阶段的测试。在开发 ETL 过程中，对于新的源系统、主题域或者任何主要的版本的开发建议也采用 3 段论方法。在 ETL 项目开发的每个阶段都应该进行下列的三类测试：

- 单元测试。在开发完毕，开始 QA 测试之前进行单元测试。测试在开发环境中由 ETL 开发人员和系统分析师完成。
- 质量保证测试（QA）。当多个独立的小组在独立的生产环境的镜像中开发后通常要进行这个测试。环境由 DBA 和 QA 团队成员创建和控制。这

个环境用来保证所有的 ETL 处理能够按照预想执行，符合所有的业务规则和期限（加载窗口）的要求。利用模拟的生产环境，QA 小组可以确保 ETL 处理在生产环境中工作。

- 用户验收测试（UAT）。这个阶段通常由你的用户小组在 QA 环境中独立的受控的环境中完成。数据库由 DBA 团队成员控制。在较小的公司，出于减少维护费用出发，在 QA 测试完成后，可以将测试环境开放给用户进行用户验收测试。UAT 这个测试阶段非常有用，通过用户直接操作观察数据来确保处理过程按照预期运行。在 UAT 结束后可以获得你的用户的签字确认。一旦签字确认，你就可以准备将其部署到生产环境。

有些项目对于小的版本和修正 Bug，经常忽略用户验收测试阶段，直接从 QA 测试阶段进入生产环境部署。在这些案例中，在代码加入生产环境后，用户不可避免地会发现问题。省略用户验收测试是一种短视的行为，这使你无法在用户发现之前发现问题，这对于生产环境来说太迟了。

在测试新的 ETL 过程的时候，需要确保对已知的数据问题以及源系统的异常情况进行用户测试。这些测试不仅仅能够验证你的努力，输出干净的数据还会使用户满意，令他们乐于使用新的数据仓库。清洗数据的努力会获得一些正面的影响，用户将乐于正面的评价 ETL 和数据仓库项目，并将更多的主题域纳入数据仓库，项目经理开始将相应的数据进行转换，并加载到数据仓库。

开发测试用例

在 ETL 开发开始后，使用业务规则和缺失数据文档，系统分析师和 ETL 架构师共同负责开发单元测试、QA 测试和 UAT 的详细的测试计划。

测试计划应该包括测试各种业务规则场景的案例。将测试结果和预期结果进行校验可以确保 ETL 代码是正常的，以及转换完全按照设计实现。你的测试用例应该故意的将质量很差的数据加载到数据仓库。ETL 过程应该或者拒绝或者经过转换后完成加载，应该生成相应的审计报表。即使是质量差的数据也不是故意的加载，提供测试数据仓库中数据质量的查询语句，以确保数据清洗转换按照预期工作。

很可能，问题将在验证测试案例的过程中被发现。一些问题可能使你的代码

产生 Bug，另外的一些可能是由于用户按照新的格式操作数据造成的。在这个阶段，接到用户希望增加新的需求的请求非常正常，这些需要作为增量加入。千万小心：数据不是这里测试的唯一目的。管理初期的 ETL 是一个任务，但是随着增加 Bug 的修正、额外的需求以及不断变化的业务需求，处理可能会变得不可管理。深入的变更管理技术在本章的“管理范围”中进行了详细地介绍。

一个测试案例样例模版如图 10.6。目标是捕获你测试的需求，执行测试的详细步骤，期望的结果；测试的状态：通过或者失败。测试案例样例用于记录你希望记录的详细等级。这个模版适用于测试过程三个阶段中的任何一个。

部署 ETL

下面介绍 ETL 部署。要使得向产品环境的迁移尽量的平滑，需要创建产品支持文档。这些文档应该包括下列的信息：

- 最后的谱系报表
- 运行或者重启增量加载处理的过程
- 自动加载时间表的细节信息

创建和发布错误恢复过程文档很重要。一旦加载处理失败，用户应该有办法访问坏数据，或者过时的数据。在生产环境交给用户使用之前，必须有一个避免这种情况发生的计划。记录并测试你的错误恢复过程，当错误发生后，你可以快速的恢复数据，并及时地使其能为你的用户使用。

和 DBA 团队一起创建一个健壮的生产环境，加载你的历史数据，使用你的生产调度器开始 ETL 增量加载过程。在生产环境中使用数据测试历史数据和增量数据的加载，确保数据成功地被加载。

测试案例模版							
编号	测试主题	步骤	测试细节	从属	成功或失败	预期结果	注释
1	将在业务系统中新出现的产品成功增量加载到数据仓库中		新产品以及细节存储在业务系统中，运行增量加载。查询数据仓库实例，并验证结果。	测试结果基于成功的增量装载		数据仓库中产品编号89076与源系统相匹配	
		a	将QA实例记入源系统日志				
		b	在源系统中添加编号89076的产品以及细节。				
		c	保存新纪录				
		d	通知ETL经理或DBA执行增量加载				
		e	在查询工具中读取记录，并记入日志				
		f	执行查询Select * from d_Product where Product_Id = 89076			数据仓库中编号89076产品与源系统中一致	
		g	验证查询结果			查询结果与交易系统中一致	
2	增量装载中相同产品编号的记录无法进入到数据仓库中，同时自动生成报告。		多个相同产品以及细节存储在业务系统中，运行增量加载。查询数据仓库实例，并验证结果。	测试结果基于成功的增量装载		数据仓库中产品编号为89076只有一条记录。自动生成报告。意味着源系统中重复的记录被去除。	虽然在源系统中去除了重复记录，但是没有设置业务规则的权限来防止重复记录，所以必须要用ETL程序防止重复记录进入数据仓库，生成报告
		a	将QA实例记入源系统日志				
		b	在源系统中添加编号89076的产品以及细节。（已有相同的一条记录）				
		c	保存新纪录				
		d	通知ETL经理或DBA执行增量加载				
		e	在查询工具中读取记录，并记入日志				
		f	执行查询Select * from d_Product where Product_Id = 89076			返回1条记录	
		g	验证查询结果只有一条记录				
		h	验证是否自动生成报告有相同的产品编号			数据仓库中产品编号为89076只有一条记录。自动生成报告。意味着源系统中重复的记录被去除。	

图 10.6　测试案例模版

维护数据仓库(ETL)

根据你的企业组织的情况，数据仓库项目经理和 DBA 团队通常负责数据仓库正常维护。但是，你是 ETL 过程的所有者，除非安排了其他人，否则它的维护也是你的职责。

当进入生产环境以后，你需要对数据仓库中已知的内容问题进行持续的监控。开发捕获已知问题的审计报告是维护的一部分。这些审计报告应该利用业务规则和数据缺省值文档，这些审计报告需要能够自动的通过 E-mail 发给相关的联

系人。

在任何生产 IT 环境中，补丁和升级是不可避免的。这些补丁和升级通常和数据仓库环境关系密切，因为在一个解决方案中集成了如此多独立的工具。根据需要不断的使用补丁和升级，建议你建立系统维护时间表，并在维护时间内执行这些升级操作。所有的补丁和升级的开发必须经过完整的开发周期，包括开发环境中的单元测试，QA 测试和用户验收测试。通过测试的补丁和升级确保了正确的升级，保证了所有的处理能够按照预期运行。

当新版本或者增强版本发布到生产系统的时候，需要让客户努力地适应新的变化。沟通是必要的，沟通可以帮助用户为变化做好准备。你的用户可能会等待一个新的版本或者增强版本。要给他们一个明确的版本的时间表，这可以推进他们的数据仓库经验。

管理范围

在开始项目后，不需要多久你就会意识到定义范围并获得用户签字确认有多么的重要。当你需要应付大量需求变更的时候，项目很容易失控。

对 ETL 规范不去管理，在开发和测试阶段经常会收到增加的需求，任意的变更都会关系到项目的成功与否。问题肯定会被发现，新的想法会浮出水面，所有的都需要被马上实现。根据我们的经验，当数据仓库揭下神秘的面纱后，新的期望和需求会产生，当越来越多的主题域部署后，需求会指数级的增长。在你开始了解这个情况之前，你会被你和你的团队在承受能力之外的工作狂轰乱炸。记得某人提到过的范围蜕变吗？创造一种机制来跟踪和控制这些变化对你的成功至关重要。下一节将提供你需要跟踪变化所用到的文档，以及帮助你执行的推荐流程。

变更的跟踪

作为一个成功的 ETL 主管，需要一个能够跟踪需求改进、Bug 修改或者最初确认范围的修改的处理流程。下面列出了一些已经证实对于捕获和跟踪需求变化非常有效的元素。你可能希望能够跟踪和管理下列的信息，尽管这意味着需要创建一个简单的电子表格。捕获下列的元素可以帮助管理变化，及最小化范围蜕变。

为了这个目标，可以很容易的使用 Microsoft Access 或者其他小型的个人数据库应用来创建一个系统。对于大型的项目，你可以实现一个小的 VB 应用，或者利用已经投资的系统。当然，好的 ETL 工具可以提供这个能力。

- 主题范围。这是需求所对应的数据仓库中数据集市的名字。
- 请求日起。请求发生的日期。
- 变更描述。用于获取这个需求的概要性的描述。
- 优先级。高，中或者低。这是需求协商后的重要性等级。
- 变更类型。无论这个需求是一个新的需求，还是对现有处理的改变
- 状态。状态的值可以包括表示这个需求状态的任何内容。我们曾经使用：新需求、开发人员调研、开发人员开发、需要进一步信息、中止、通过单元测试、通过 QA、通过 UAT，就绪可供生产，等等。这个字段的值在整个需求的周期中变化。
- 提交者。变更需求的提交者的姓名。
- 属主。表明这段时间谁负责这个需求。通常开始于数据仓库或者 ETL 主管，然后分配到相应的开发人员、测试人员或者在整个需求周期中的其他责任人。
- 问题版本。当发现 Bug 或者提交请求时所使用的版本号。
- 修正后版本。需求实现后，发布到生产环境的版本号。
- 状态。打开或者关闭。只有当状态设为发布到生产环境或者取消后才可以将状态设为关闭。
- 关闭日期。状态改为关闭的时候的日期。
- 功能描述。这个信息描述了用户提出需求的经过。
- 技术性描述。这个信息通常由超级用户或者 ETL 架构师来填写。用于开发人员的编码实现改动或者新需求。

能够从这个列表中产生报表是非常有用的。如果你有资源来将其作为一个 IT 系统，请和你的 ETL 团队讨论他们对这个流程的建议，确保这个流程适合你团队

中的每一个人。

但是，对数据仓库生产环境的最小化更改仍然是最佳方案，如果可能的话。请牢记，改动的越多，对其他的处理影响的风险就越大。我们建议同相关的团队成员召开例会讨论每个需求的优先级。让你的客户认识到生产环境的最少变动是多么至关重要。确保他们理解完成每个需求所需的成本和投入。让他们问一下自己：他们的投资回报如何？有多大的好处值得做这样的修改？他们是否能够调整需求呢？

一旦最后确认需要改变，那么你必须讨论改变带来的影响。如果改变影响到其他的 ETL 过程，或者其他域，那么必须进行详尽的影响分析。要做的改动可能会带来 ETL 处理上许多新的改动。要确保增加这些新的改动到你的跟踪系统。

图 10.7 展示了一个变更/改进的申请表格。这个表格中包含了所有必要的信息，供你输入新的请求，进行开发和完成需求。这个表格配合你的需求变更跟踪系统，可以高效的管理需求变更流程。

变更通知书

变更范围 _____ 申请日期 _____

变更描述 _____

_____

优先权 _____ 变更类型 _____

提交人 _____ 所有人 _____

提交版本 _____

功能描述（附加）

技术描述（附加）

图 10.7 变更/改进申请表

版本发布时间表

通常，许多变更、需求改进、补丁和升级会作为一个单个的版本或体系捆绑在一起。每个版本必须通过所有的开发生命周期。在进入生产环境之前，必须需要单元、QA 和 UAT 测试。之后变更通过测试环节，或者 ETL 架构师进行方法迁移，或者 DBA 团队将代码发布到生产环境。

跟踪你的数据仓库的版本有助于解决生产环境中发现的问题。使用本章开始

介绍的跟踪机制控制你整个的版本发布。通常，下列的标准的版本标识技术对数据仓库和 ETL 环境都非常有效。由于很多数据仓库的代码版本都是由 ETL 团队开发和部署的，所有 ETL 主管遵循这样的规则尤其的重要。

版本号有一系列的 3 个由小数点分割的数字组成（##.##.##）。第一组数字表示主版本，第二组表示小版本号，第三组表示补丁。例如，版本 1.2.1 意味着数据仓库是他的第一个版本，有两个小的版本，应用了补丁 1。

在数据仓库环境中，主要版本发布通常表示有一个包含新的事实、维度、ETL 过程的新的主题域或者新的数据集市。一个小版本表示根本上的 ETL 修改，可能还包括小的数据结构的修改。补丁通常是紧急修订的结果，在生产系统中发现了一个关键错误，需要进行立即的补救。如果补丁和小的更改，或者小的更改和主要的更改同时发生，那么只有最左面的数字会增加，而右侧的数字清零。例如，如果版本 1.2.1 是生产版本，你有两个补丁、一个小的更改，以及一个预定要迁移的主要版本，将这些变更打包将构成一个单一的新的版本。在本例中，版本将是 2.0.0。

在两个间隔时间较长的紧急修订之间安排一次主要版本的绑定是一个好主意。通过安排发布主要版本，可以是月粒度，可以很容易的将小的修订绑定到受控的版本环境，从而减少代码迁移。

我们建议数据仓库版本策略非常的有效，尤其是你的项目使用数据仓库总线架构的时候。在这种情况下，总线矩阵的每个数据集市进入物理数据仓库将是一个主要的版本发布。如果你的数据仓库的版本是 1.0.210，那么你很可能没有使用这个矩阵，而且可能通宵未眠。

11.3 总结

本章中，我们最后一次从大量的 ETL 团队任务中退回来，以对整个团队获得完整的概念，有哪些人参与，他们应该考虑什么问题。我们必须牢记本章以及整本书专门讨论涉及企业数据仓库的后端。

我们通过描述 ETL 团队的规划、以及面临的领导能力挑战开篇，然后我们深入这些成员所面临的每一个专门的任务。在书的主要内容中提供了很多案例以及

非常细节的信息。

第 4 部分　实时流 ETL 系统

12 实时 ETL 系统

建立实时 ETL 数据仓库的解决方案常常需要分辨出某些不太可靠的业务目标，要理解不同的整合技术，要知道其他人运用成功的一些实际的方法，还要拓展工程的灵活性及创新性。这个领域体现了具有新技术、新方法、新词汇的全新理念。显然，这可能是解决问题的关键，而实时数据仓库还将成为早期具有巨大潜力的适配器，使其获得竞争力，降低风险而盈利。本章提出了构建实时 ETL 的四个过程，通过选择合适的实时 ETL 技术、特征、及方法来指导专业实验数据仓库。

1. 本章调研了实时数据仓库的技术状态的历史及业务情况。提供了一些背景知识，比如我们如何实现，及我们的目标是什么。

2. 其次，它描述了区分组织的实时需求的方式、方法，而这些方法在选择后来的设计结论时最有用。

3. 本章的中心是对一些实时报告以及整合服务机制的评估，并针对每一个方法提供了最合适的技术，以及针对这些技术的优势劣势的分析。

4. 最后，本书给出一个判定矩阵，它按需求分类，并使用了前边所描述的方法。他通过选择技术途径及方法论上指导 ETL 工作组。

必须指出，这份材料并不是建立实时 ETL 结果的诀窍，如上所述，这样的诀窍是不存在的。随着这项技术越来越普及，你必定要应对各种需求，而此时 ETL 还不十分完美，你就要应用你的创造性及积累的个人实践经验，来改革 ETL 结果，使其最适应于你所遇到的挑战。在完成此项工作的同时，你的工作也起到了帮助完善实时数据仓库技术的作用。

流程检查

规划与设计：本章触及到 ETL 系统的计划和设计的各个方面。我们希望本章能作为第 1 至第 10 章的补充。

数据流：本章所有的部分及技术能应用到现存的 ETL 系统框架中。但我们还要

强调，从分段批次 ETL 到连续 ETL 是一个深刻的完全的转变。

12.1 为什么要有实时 ETL

不久前，设计师们都在强烈维护这个理念，即数据仓库需要给业务决策者提供一组不变的数据，提供一个可靠的信息平台。针对某一时刻数据库的最新报告 t，业务用户都是以事务的生产应用为目标，因此用户必须了解，到昨天为止这个事件所发生的历史过程，同时还要看到许多 OLTP 系统今天所发生的事情。业务用户完全不需要接受这种分割，他们为什么不能只去一个地方就可以得到他们所需的所有信息呢？

那好，许多都改变了，数据仓库已经成为他自己牺牲品 become a victim of its own success。虽然在数据仓库中事物处理及事物再现之间的延迟一般小于 24 小时，但对于快速运行的垂直运行的系统，这个延迟就太长了。数据仓库要完成决策的任务，要给操作系统反馈回丰富的信息，使这个系统能提高处理执行过程、提供个性化服务并体现面向需求，推动信息的不断更新。

还有很多其他的重要因素，使数据仓库的实践者反思早就出现过情况：

- 客户关系管理（CRM）。现代 CRM 要求一个同步的、稳定的、完整的客户形象提供给操作系统。CRM 又在相当高的层面上直接或间接地服务于客户。尽管市场对于 CRM 卖主包装的需求而言，性能不能超出构架来购买，除非所有面对消费者的系统可由 CRM 单元来付清。尽管市场需要的都是已经打包好的 CRM 产品，但是这种需求是不可能实现现货供应；除非将所有的面向客户的系统都包装上 CRM 的外壳。在业务上，也需要集中消费者的实时信息来跨越传统的处理型 stovepipe 的应用。数据仓库当然完全需要从操作型应用上获取稳定不变的客户信息流来充实仓库中的客户信息，但事实上，越来越明显的是，操作型应用也依赖于数据仓库中丰富的客户信息。因此，可以预言，组织者已经开始开发结构化的交变器 architectural alternatives 使它能够支持没有显著特征但更为复杂的集成情况——操作型数据在操作系统和数据仓库同步进出——越发紧迫。

- 零滞后的企业业务处理模型。这里提倡的是理想化的速度效益和真实的单一模式。在一个实时、零滞后的企业中，信息在恰当的时间提交到一个恰当的地方就意味着获取最大的商业价值。某些人称这个为即时时间系统。正是企业对及时编目、实时供应链及客户化的指令和物理集成业务模式的需要，扩大了对绝对数据流和渗透信息的需求。现实中，真正的零滞后是不可能达到的理想状态，跨越各种生产系统和数据中心来统一信息是需要时间的。但是对于当前许多只能提供反映事物最佳状况的低滞后的数据仓库上的压力却是真实存在的。
- 全球化及互联网。最终，可能最有效的是全球化和网络化的共同作用，他使得数据仓库能在全球任一时间进行数据的存取与操作，能适应对数据仓库日益广泛的需求，这样就强烈地压缩了下载数据仓库需要的时间。

需要输入数据仓库的数据量不断扩大，而业务荡机时间（downtime）又要不断收缩，这对已超负荷工作的热爱数据仓库的 ETL 团队是一个挑战。如果你能设法在全天的时间中细水长流地不断输入数据仓库的数据，而不是猛地扩充数据载荷到因停机时间而收缩的时间期限，这可能会更容易些。所有这些因素促使数据仓库逐渐达到实时状态。

12.2 实时 ETL 的定义

实时 ETL 在许多情况下是对提供数据仓库式服务的一个误称。这个服务既不是真的实时也不是 ETL。而这个术语指的是可以在事物处理执行的几分钟内将数据马上但非同步地传输到数据仓库中的软件。多数情况下提交实时数据数据仓库化需要的方法，完全不同于使用定向批处理程序组数据仓库化的方法。整天单一的以更多频次地运行常规 ETL 程序组是不可能实现实时 ETL 的，无论是 OLTP 系统还是对数据仓库。反之， OLTP 系统的逻辑提交任务处理中的数据仓库也是不能这样工作的。让 OLTP 系统在着手它的下一个交易之前，等待数据仓库的数据下载到提交任务上，那简直是奢望。也没有任何同步的或两相的逻辑提交任务能实际跨越不同结构，不同粒度水平的系统。而你仅仅希望在事务可接受的限定时间内，将新的处理输入到数据仓库的特殊的实时分隔区（本章后边会有定义）内，并为每日的操作决策提供分析支持。就目前来说，这个过程就是我们对实时 ETL

的实际定义。

本章为了实现这些目标使用了数据仓库设计师熟悉的主流工具装置，研究了某些实际方法。然而，实时数据仓库是一个新的领域，充斥着各种各样的软件商的要求和较高的风险。本章所研究的实时 ETL 方法试图通过控制预期、重视成熟方法和实施战略工具选择来将风险减到最小。本章提供了在业务处理上和数据仓库可提供的范围内，为目标分配达到最短时间滞后的方法。

12.3 实时数据仓库化的挑战和机遇

实施实时数据仓库，给 ETL 的设计者提出了许多独特的挑战和机遇。从技术结构方面看，它要有大批量处理的能力，每天晚上批次处理 ETL 事务，使之形成整日的连续不断的 ETL 数据流。当事务开始依赖数据仓库中事务处理低滞后时间效应时，相对于实用性，系统就需要升级了。如果组织者选择本章所述的实时维管理方式，实用性就成为战略优势了。

从数据结构方面看，实时数据仓库挑战了以离散周期测定值建立的数据仓库的地位。后者只是提供了业务简单映射，而现在我们提倡的是更加综合，更加连续的瞬时信息系统。这种变化的发生是很细微的。例如，事实加载频率从每日一次增至每 15 分钟一次，但是如果事实加载和维度记录能连续进行那岂不是更加生动了吗。那么数据仓库就可以在所有的时间点上及时捕捉到事务处理的信息和他们前后的维度。维度的缓慢变化变为快速变化，数据仓库的支撑（bearing）变得更贴近实际的情况了。事实上实时数据仓库也支撑了实时维度的形成及同步化，它也涉及了操作系统本身的逻辑延伸。

12.4 实时数据仓库的回顾

数据仓库的实时方式能够很清晰的追溯到最原始的 ODS 系统。原始 ODS 的目的与现代实时数据仓库非常类似，但实时数据仓库的应用还影响到了新一代的硬件、软件技术的发展。下边将更详细论述这些思想。

第一代——操作性数据存储

操作性数据存储或称 ODS，是第一代数据仓库。它构造的框架试图在数据仓

库中分离出明显的框架结构及应用，来获得低滞后的报告。ODS 是半运行半支持决策的系统。试图在频繁更新和频繁访问的即时响应之间达到平衡。早期的 ODS 架构被描述成一个数据被整合并反馈到下层数据仓库的模式，因而起到扩展数据仓库 ETL 层的作用。后期的架构被描述成集成来自数据仓库 ETL 层数据的消费者，按照已存在的全面的架构和从操作型系统下载数据的及时性，将架构分为从类型 1 到类型 4，以及内在的或外在的 ODS。

实际上，ODS 已成为数据分级、数据清理、数据展现以及执行报表的架构元件汇集区。根据这些不同的作用，它是每一项任务的综合协调器，一个初级的、欠折衷的过渡体。

第二代——实时分隔区

逻辑或物理的实时分隔区的使用，就像 Ralph Kimball 原来所描述的那样，是一个用于从数据仓库中传递实时分析的有效可用的解决方法。使用这种方法，创建了离散的实时事实表，它的粒度和维数与每晚加载的静态数据仓库中相关的事实表相匹配。这种实时事实表只含有当天所发生的流量（这些尚未加载到静态数据仓库表中）。

图 11.1 显示了 2 组星型结构分别连接到实时和静态零售销售点事实表，他们的维度相同。

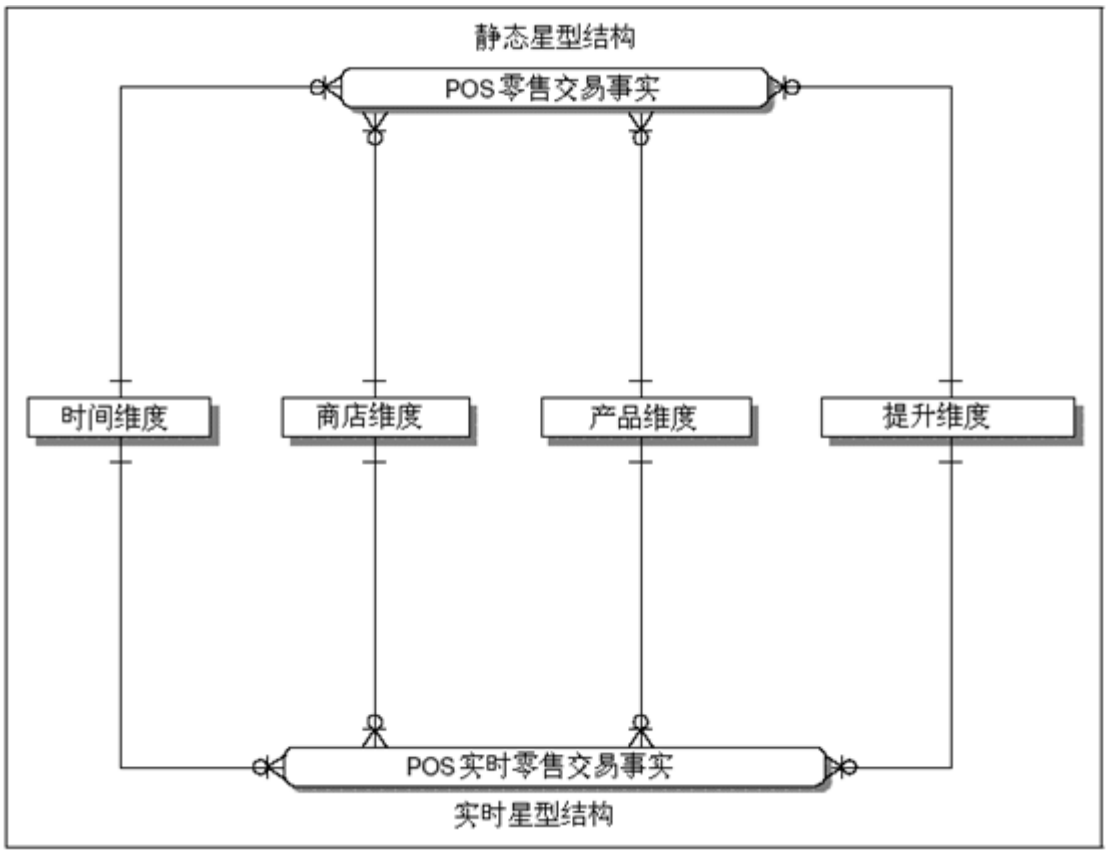


图 11.1　静态和实时星形结构之间的关系

每晚。实时分隔区的内容都被写进静态事实表中，然后实时分区即被清空，准备接收第二天的处理情况。图 11.2 列出了这个过程是如何工作的。实质上，这个方法将 ODS 实时报告的好处带给了数据仓库本身，使其在过程中消除了过多的 ODS 结构的开销。

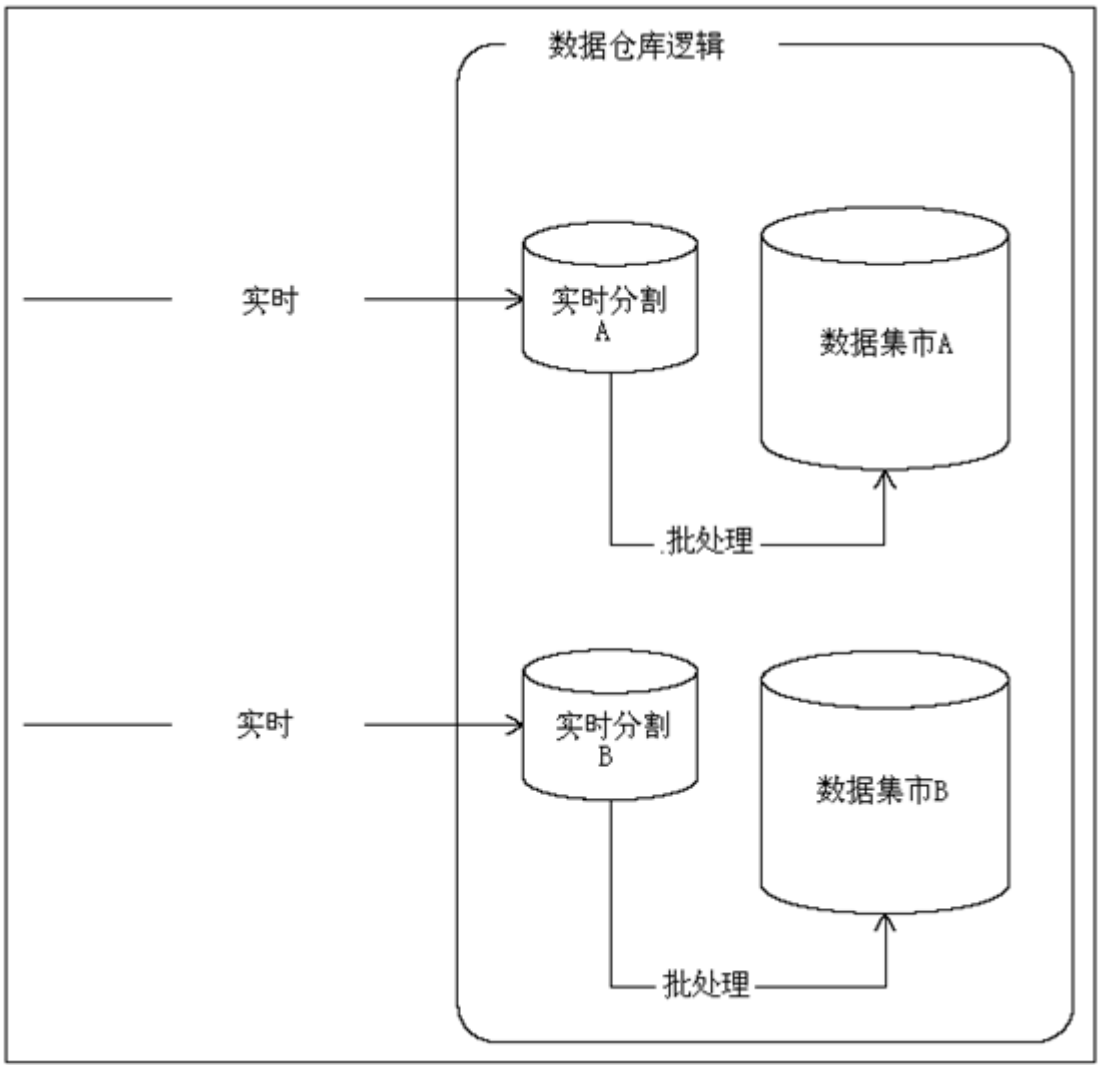


图 11.2　数据集市实时部分的逻辑关系

剩下整个一天的时间，事件逐步插入到实时事实列表中，客户对实时列表的查询,不会不完整,也不会被加载过程所中止。为了减小数据加载作用和查询回应次数所带的影响，实时事件列表的索引是最小的或不存在的。靠限制（一天内）列表中的数据量和将完整的实时事件列表存入内存中可以提高性能。不经意间，在实时和静态列表中的关联事件的观点就产生了。

如果单独的事件记录被逐步地插入到实时间隔区，这时需要某些策略来处理发生在夜间的大容量载荷产生的维度变化。例如，在白天出现新的客户记录，你可能会延迟将这些新变动的客户的一个批次加载到静态客户维数中。因为一般新的客户记录都是在晚上更新到进一步描述客户记录的客户维数中。而实时数据仓库就能选择保持更多的频次断点来改变维度简单映射或可以放弃时间点

（point-in-time）的概念而捕捉到所有刚发生的维度变化。

最近，本章所描述的问题与选择合适的方法来处理维度的变化有关，一些有效的方法是将数据在整个事务日内逐步地插入到实时间隔区，及对这种方法的利弊。

最近 CRM 的趋势

CRM 要求对每一位客户系统的历史包括对所有客户的每一个接触点都有一个全面的了解，并要洞察客户在市场中所面临的挑战及前瞻预期。在过去几年中完整地 CRM 系统已被广泛的应用于商务活动，以支持前期目标，使组织系统中最简单最普通的客户接触点一致化。然而，在分割客户来支持分析系统上，这些系统体现了对组织的重要优势，面对这种优势他们就不理解了。常常是那些比较老的，比较特殊的系统都还存在着，而这些支持客户相互作用的系统不再是 CRM 系统体系了。这些处理过程再没有找到他们返回 CRM 系统体系的方法。CRM 系统体系也是典型的用户智能装备不合格的组织机构，它需要被它的客户认识到作为智能合作者和伙伴，他们还缺乏收集、获得成功的机构，协调一致的客户、和跨企业的智能市场。进一步打破 CRM 体系市场，建立相对应的分析型 CRM 类型的运作型 CRM 扩大了这种分化。业务没有运作或分析型的客户，同样操作型和决策支持系统也要共同运作以服务于客户。

现在需要一种方法，这种方法可以将过去和现在的客户相互作用的组织数据一起快速整合，联合外部市场信息，用某些机制将数据转换为客户智能，这意味着组织中的每个人都可以共享资源。将这些东西集中在一起代表了混合数据仓库的技术和集成技术的应用。

CRM 的供应商敏锐的察觉到组织结构所面临的挑战，所以某些结构就将商务智能能力与他们的操作型 CRM 系统结合起来。这个结果只是基本的，过分简单化的，对结构的保护是困难的，最终难于形成不同于他人的有竞争力的产品。

第二代 CRM 入我们在本章定义的，不是可以购买和安装的应用程序。它要求是一个综合数据仓库，含有所有客户接触点、智能筛选和市场数据的利用；它要求是一个连续获取客户智能的数据仓库，要求是一个在跨企业间分享和同步传递客户信息的机制。能提供这种能力的任务，似乎更应放进同期 ETL 结构后备库

中。

维度管理者的战略角色

在数据仓库中维度总线结构中，逻辑和物理独立主题区域的联合维持了维度和事实的规范化，可以通过利用本章描述的维度管理系统完成。。通常维度管理者被认为是定义、维护、给所有数据中心发布特殊维度的一种角色。这些数据中心是经数据仓库中的传输总线相互作用的。

最后，实时数据仓库将把大型目标的准备通道提供给最当前的最重要的数据，并提供给企业的所有客户另外为了给数据仓库快速提交事实记录，在提供实时同步关键维度方面，诸如跨越所有组织的操作系统的客户和产品上，要建立巨大的竞争优势。这种同步信息作用要考虑维度管理作用的逻辑扩展，要是一个有效的、可靠的机构。这个机构提供数据仓库分布式分区的方法和给操作领域提供其他丰富的信息，在操作领域和数据仓库间形成闭环。

在战略实时数据仓库中客户维度的管理者不仅可以向数据仓库逐步插入新的合适的客户信息，而且可以与某些机构合作跨越相关操作系统提供同步客户信息。这个实时客户信息应包括数据仓库本身客户智能。

显然这些是很有雄心的目标。如本文所写，非整套的决策方案或 end to end 端对端衔接的工具装置，显著地简化了建造一个双向 EAI 或实时数据仓库解决方案的过程。但是，像这样的系统已经创建出来了，这些系统的基本结构几乎已经存在并变得更加完善。由这样的系统提供的商务差异的可能性被取消了。它就像当今早期的接收器，具有市场的有利条件，可推动将来更广泛的使用这样的系统。今天建造这样的系统应考虑将来能使这种组织能力发展为实时 EAI 或数据仓库华的解决方法。组织机构处于竞争的压力下，或现在需要通过友好的客户寻求存在差异的市场来越过这一关.

12.5 需求分类

很明显，这个话题给我们提供了很多架构方面的思考。提供更为复杂的一系列与主流实时数据仓库相关的优劣分析，，列出实时需求的范围是非常重要的。

在接下来的部分向您展示的是一些 litmus 测试问题，这些问题的答案可以帮

助您将您的企业组织需要的实时系统的能力进行分类,并且为您以后的任务选择适当的方法和工具。在本章临近结束的时候我们给您列出了一个矩阵，这个矩阵概括了这些讨论内容并且能够在方法和架构的选择上引导 ETL 团队。

数据的饱满度和历史的需求

降低 OLTP 与数据仓库间反应时间的发展成本和复杂性是遵循收益递减法则的，降低反应时间会非线性的增加复杂性和成本。因此，您需要为数据仓库必需的数据饱满度设置现实的目标和期望值。

同时您还需要彻底明确一个概念，即困难的商业需求设置既不能满足传统数据仓库日常的发布也不能满足 OLTP 系统的操作报告。当您考虑期望的实时数据仓库的需求时请参考下列红色标记的内容。

- 低于 5 分钟的响应：以这样低的响应速度生成报告将不能满足主流实时数据仓库的可靠性。这个时间是持续降低的，但是它同样需要进行超大量的处理和时间将信息从 OLTP 系统中移动、转换并装载到实时分区中。那些完全拥有超过 5 分钟刷新时间的信息量的企业组织应当考虑直接通过操作系统生成他们的报告。

企业信息集成程序不涉及到这种响应时间的限制并能够直接通过操作系统传输几乎最新的报告。然而他们也有其他必须考虑的特点和限制。企业信息集成系统以及这些限制将会在本章稍后讨论。

- 只需要很少或无需历史数据支持的单一数据源需求：这种报告不要求由数据仓库提供集成的历史数据，并且最好由操作系统本身寻址（addressed）。恰当的来说，他们应当呈现非常细致的 OLTP 系统报告（footprint）并且不能明显降低操作性能。如果网络可用的话，这些报告可以通过商业智能入口（BI portal）呈现，这个时候用户群则会认为这些是基于数据仓库的报告。
- 通过已存在的数据仓库生成面向不同用户的报告：这些报告的分发也许需要新的报告词汇和机制，以及会使已经很复杂的实时数据仓库发展成果更加复杂的因素。实时系统不是一个自动的项目终结者，举一个例子来说，实施系统架构师应当意识到在海运中和市场管理中所应用到的商

业词汇和矩阵很可能是有很大差别的。

- 特定分析的非实际需求：如果能够稍微对低响应数据部分进行特定分析的话，你就可能避免对一个完整的 ETL 系统流进行重新设计。也许你只需要简单的创建一个基于最新操作系统数据的快速报告，并将这个报告附加在传统数据仓库中根据昨天的数据所生成的报告中。
- 还没有成功实现数据仓库的企业组织：至少从目前来看，将尝试建立一个实时数据仓库作为最初的商业智能发展成果是不被推荐的，仅仅是因为它需要同时精通大量的学科。值得庆幸的是空间数据仓储架构和方法允许企业组织今后可以很好的添加实时报告功能。

以上这些红色标志的共同的象征应该是一个报告，这个报告要求的数据是在昨晚之前更新的而且必须能够满足最少 5 分钟的响应时间。这样的报告也许同样依赖于数据历史、报告词汇、已存在的非实时数据仓库的演示等方面的连续性。这些红色的标志是符合实时数据仓储 ETL 方法所描述的合适考虑对象。

接下来的部分我们将讨论一些关于实时报告的基本要求。

仅仅是报告还是一个综合过程？

企业组织需要一个单一的解决方法仅仅为了报告的目的而把操作数据移动到数据仓库中吗？或者说对于通过在操作系统自身和/或数据仓库间闭环转移的规范化的维度数据也同样有需求？举一个例子来说，将基于数据仓库的客户部分返回操作系统是否需要一个机制？这也许是最为影响选择实时数据仓储方法的一个问题。

当然，任何客户关系管理策略都可能需要共享时间列表和最为完整的客户信息，包括可操作的客户数据（例：最近的销售或投诉信息）和源于数据仓储或数据挖掘的客户市场信息，比如客户分割、资料以及寿命值。在企业组织可操作系统和决策支持系统间闭环循环的过程中，实时报告是否为第一步骤？

仅仅是事实还是包括维度变化？

商业人士和空间维度数据仓库架构师将世界定义为事实和维度，但是 OLTP 系统不会做出如此明确的区别。但是，作为工程师来说，你必须明白并将 OLTP

系统的商业操作按照终端用户的喜好做好分类并进行适当的设计。

实时报告需求是否专一的关注在诸如最近提交的订单、最近发送的电话、近期完成的股票交易以及近期拨打的销售电话等最新的事实吗？还是说它同样要考虑到一些诸如客户或产品记录的更新维度一类的最新操作？如果实时空间维度变化是报告所必需的，那么他是否是连续或缓慢变化的？换句话说，用户群在面临做出操作的时候是否需要一个准确的维度表，或者当更新发生时，是否所有或部分维度更新会被破坏性的进行过度重写？报告是否需要有可重复性？第一类型慢慢地改变维度，对维数属性的变动破坏性地重写期望值，导致连续重建历史的数据仓库，报告事件不是看他们操作的时间而是要看今天的维度状况。在这种情况下，通过不同的时间点相同的维度生成的报告可能会有轻微的或者显著的不同。如果第一类型转变用于聚集演算依据的话会造成历史聚集的无效，因此这种变动是危险的。

类型 2 和类型 3 在某些点上会缓慢的改变维度从而使得粒状维度图更加清晰，也许是每天都在改变，但是他们仍然不能捕获到发生在抽取时的维度变化。实时维度刷新可以使粒度提升到每隔几分钟或者捕获所有维度变化。

建筑学的含义不是微观的。通过采取加快捕获维度变化图像频率的方法，数据仓库远离了他的早期状态即周期性测量（快照）系统，继而转变为 0 响应时间的决策支持理想模型。当数据仓库和应用综合化技术开始相结合的时候，数据仓库实际上将成为管理企业操作的系统的一个真实的逻辑引伸。暂时作为当务之急的，ETL 系统大概需要被设计成为能为被测量的事实提供尽可能接近 0 的响应速度，但是同时还要允许部分或所有维度属性可以进行批量更新。

告警，持续处理或者无事件?

虽然通常情况下 ETL 系统对于维度化数据递交前台的界限有非常明确的定义，但是在许多情况下一个实时系统无法有这个界限。The architecture of front-end tools is affected at the same time。现有三种数据交付范例，都要求通过端对端的观点来达到所有从初始源到终端用户的方法。

- 告警：通过源数据条件强制用户界面进行更新。
- 持续处理：终端用户的应用程序为了实时更新屏幕而连续探查源数据。

- 无事件通知：如果某个特定事件在某一间隔时间内或者由于某些特定条件制约而没有发生，终端用户将被通知。

在以上每一个事件中，实时 ETL 系统将通过发送通报或者接受直接请求的方式与终端用户的应用程序相连接。

数据综合化或应用程序综合化?

假设实时数据仓库的要求同样适用于某些横跨可操作系统的综合的测量，您需要将您的要求分类为数据综合化或应用程序综合化。

- 总的来说，能够通过简单的在数据库间移动数据来满足综合化叫做数据综合化。通常，这些解决方案都是点对点的形式，对于异构数据库通过 ASCII 码文件抽取、触发、数据库链接和网关执行，或者通过同类数据库的复制服务或者表格快照方式执行的。精炼的说，在所有参与的应用程序间数据是共享的，完全绕过应用程序的逻辑。一些高结束数据综合化(higher-end data-integration)工具为数据的预定和移动提供集中化管理支持，支持更多企业控制和管理点对点数据综合化事务。
- 应用程序综合化（有时也叫做功能综合化）可以被描述为通过使用一些常见的中间设备把应用程序连通化从而建立新的商业解决方案。中间设备是一组可独立应用的软件组，可以提供捕捉、寻址以及在应用程序间执行可操作信息的手段。总之，连接器或适配器被用来连接所有参与的应用程序到综合化网络上，而 broker 则是用来根据公告和协议规则发送消息的。

点对点对比集线器广播

如果您的实时数据仓库同样支持某种程度的应用程序（或者功能）综合化，那么在选择构造方式上的一个重要因素是发布的数量和您预定的在未来（比如 24 个月）所期望的可支持系统。这个数字能够帮助您决定一个相对简单的点对点解决方案是否可以满足或者是否需要一个更加强壮的构造法。

图 11.3 告诉我们，即便只有相对少量的应用程序交换数据，点对点解决方案都可能要求有极大量的数据交换界面，每一个界面在来源或目标程序改变时都需

要进行维护。向综合化网络添加应用程序同样需要为所有公告和预定应用程序添加数据交换界面。尽管如此，拥有一份短小的应用程序列表的企业依然需要规范化的维度综合化，并且他们期待这份列表依然稳定，并在未来能够找到一种相当有吸引力的点对点的综合方法。它避免了创建 EAI 中间设备组件的复杂性，并且能够在一些数据综合化技术比如捕获、变换、流程（CTF）工具的应用过程中被部分支持。

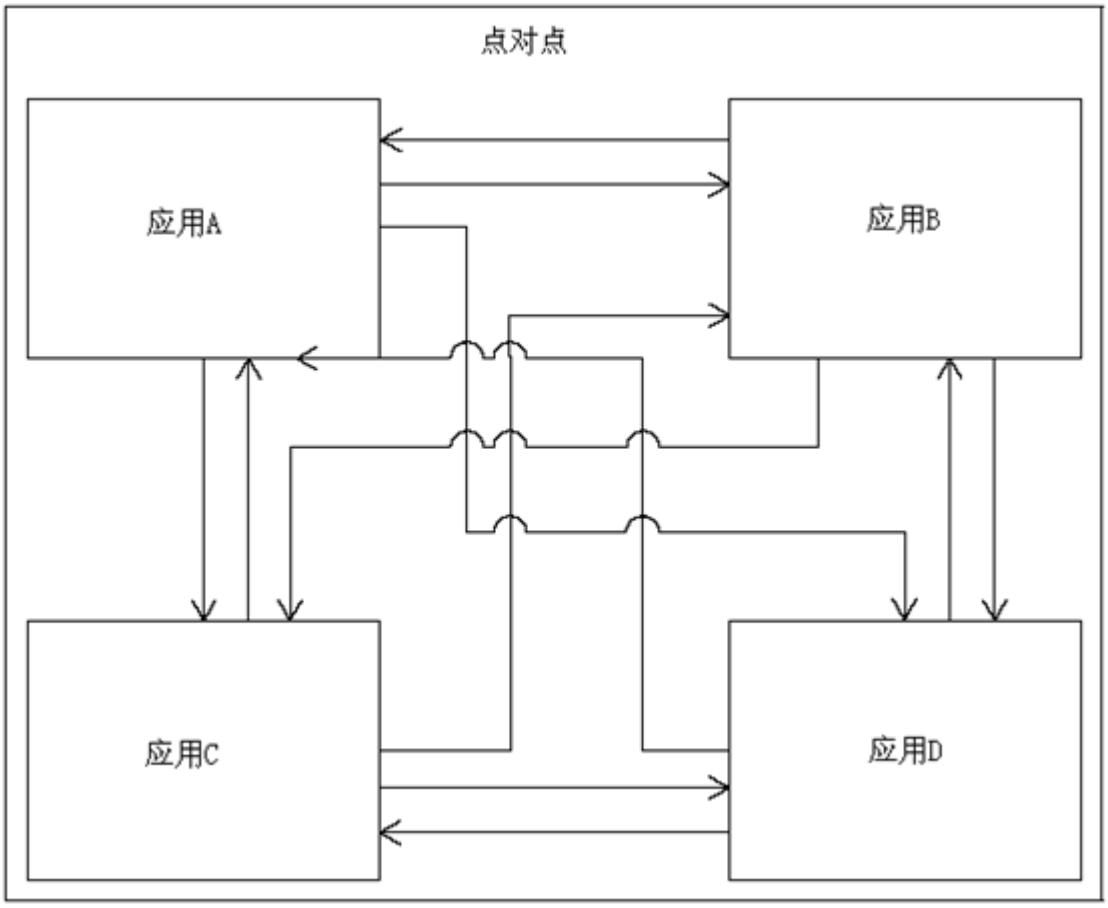


图 11.3　点对点应用集成

与点对点的架构对比，通过方法可以使用户界面和跨系统的附属物降到最低。（见图 11.4）

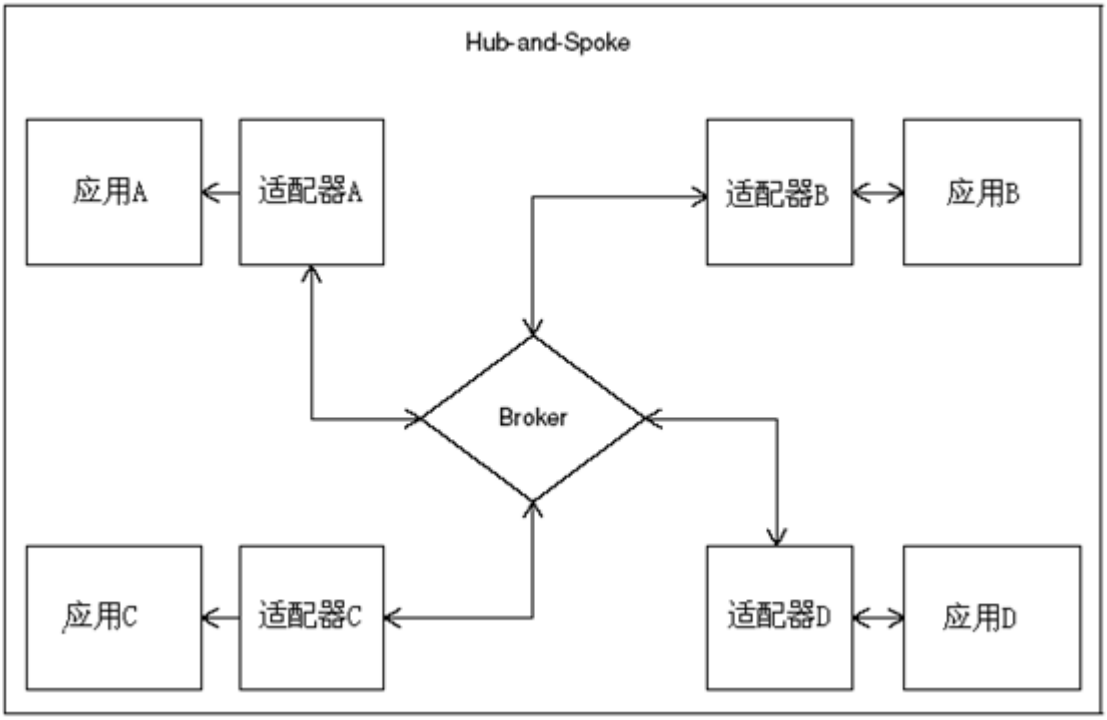


图 11.4 集线器广播应用集成

然而，建立 EAI 中间设备适配器和 broker 组件所带来的额外负担也是很大的。每一个综合化网络中的应用程序都需要一个可以将特定的操作转变为通用的消息并解释和执行的适配器。任何一个相关的主应用程序或者通用消息的改变都需要对适配器进行维护。

决定选用点对点还是的构造法并没有一个明确和快速的规则，但是那些期望综合化横跨 3 种或更多应用程序以及那些期望在未来增加综合化网络参加者的企业组织可以考虑集线器广播 EAI 构造法。这个公认的有弹性的界限同样可以有企业组织的适应程度以及对 EAI 传讯技术的承诺度来调整。

客户化数据清洗考虑因素

如果企业需要实时清洗并保持客户化数据的规范化，在选择方法上你需要额外考虑。在产生新的客户化关键上系统是否有一些适当的集中方法，以保证没有多余的客户化记录产生。这样的系统相当少见，而且它时常作用于管理数据仓库中的客户化维度以提供此服务给企业。

假设此系统是不适当地，在适合实时的客户化维度管理上，它可能产生假定匹配 (双倍的) 客户记录。目前，市面上有一些决定性和可能性的匹配工具可以

满足这些需求，但是很不幸的是，这些工具大部分只能在批处理模式下运行。客户化清洗的有效性支持确认目标地址，倾向于分割虚假记录，验证有效记录，此外，一类（householding）需要批处理。无论如何，这样已经很接近实时程度了，构造运行频率发生的微批处理 microbatches 架构，稍后在本章节中描述，首先我们分析有效的输入和输出。

12.6 实时 ETL 过程

通过创造性的回收利用一些已有的 ETL 技术和工具，一项成熟和前景广阔技术的可用于满足实时数据仓库需求。

小批处理 ETL

传统的 ETL 过程，在这一本书各处被描述为以文件为基础的方式，在每日，每周 和每月的批处理报告需求中被非常有效的选择。新的或更改的操作（事实记录）按照 en masse 运行，在每一次的下载中迅速捕获维度。因此，数据仓库中在批处理过程中更改维度是不可能的。因此，ETL 不适合于对于系统需要低延迟报告的数据和整合申请，对于需要获取更详细的维度变化的系统也不适合。但是，传统的 ETL 过程对于需要更多偶然潜在需求和复杂整合挑战的系统而言是简单的、直接的并且被验证为有效的方式。图 11.5 说明了传统的 ETL 过程。

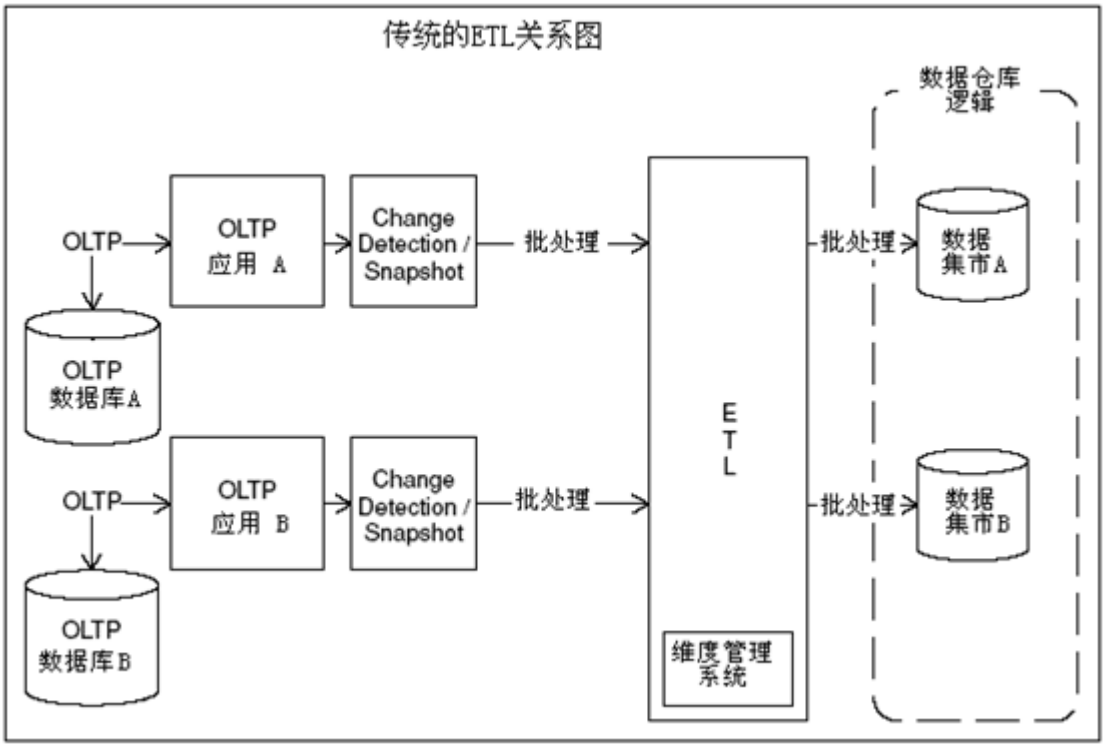


图 11.5　传统 ETL 方式

小批处理 ETL 与传统的 ETL 除了批处理频率被增加外（很有可能是每小时一次），在其他方面是非常类似的。频繁的微处理模式是以其它传统的 ETL 方式和直接将实时分类数据输入到数据集市来处理的。实时分类的数据每天一次备份到静态的数据集市并被清空。图 11.6 说明了微处理 ETL 过程。

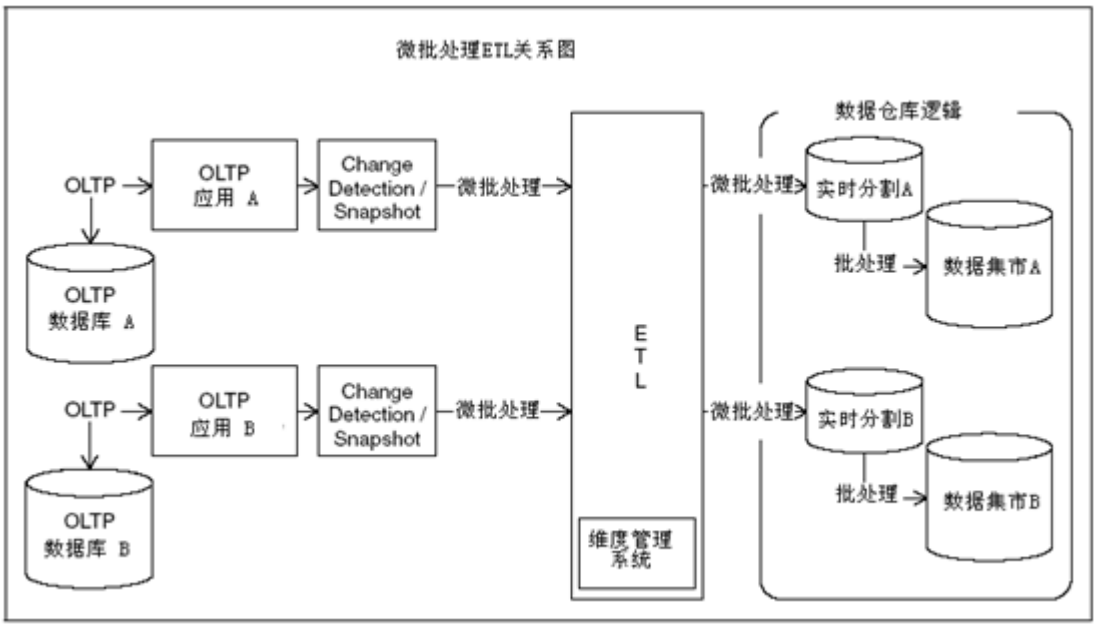


图 11.6 小批处理 ETL 方式

维度管理系统在类型 2 或 3 中慢慢地产生新的维度标准（images）更新维度，但是由于运行频率的增加，一整天更新的维度可能很快变更而且深入增长。在所有的字段中经常使用缺省值替代只为新实例产生的维度记录，忽略对维度的变更是很无奈的选择。这种折中的选择能满足于在一个给定的日子产生少量新记录的系统，也能涵盖从前一天起维度前后变化的潜在关系，但是它明显的地弱化了实时报告的一些优势。如果不可避免，处理快速变更维度的唯一实际的解决方法是正确使用细化维度，对于最频繁变更的大维度属性，使之产生单独的维度，而且藉此减少在 ETL 过程产生的新维度实例需要的数量。

有趣的综合选择是，在当天内，维度的专有备份与实时分类关联，此类变更的维度被视为类型 1。在当天内触发的简单重写产生变更。在当天工作结束时，直到关注在数据仓库的静态部分中的维度备份，任何的此类变更将被视为类型 2。举例来说，实时数据仓库的一个用户可以立即设置一个信用-价值的指标。

微处理 ETL 要求一个全面的工作管理，计划安排，从属关系，和减少错误的方式，在大部份的时间是无人看管下，对于运行给于足够强健的支持，在处理最常见的载入数据事务方面能够执行数据仓库发布的策略。一些可控事务的实效性支持这一个功能 ，但是传统发展型的任务很可能与微处理 ETL 数据仓库自动操作的任务相背离。

微处理 ETL 在 OLTP 系统上，也要求更加频繁检测新产生的和更新的交易记录，因此应考虑和谨慎处理加载对整个系统运行的影响。

以下是识别微处理 ETL 载入实时数据仓库的更新候选记录的也一些方法：

- 时间戳：验证性和有效性，时间戳主要作用于实时微处理 ETL,用于抽取区分候选数据产生和更新的记录的操作系统。简单任务中,当 ETL 过程运行时，这一个方法确实作用于在操作系统的所有更新发生时的时间戳频繁写入和读出。索引时戳提高了读出而且减少读出的开支，但是有时会惊人的增加的插入和更新操作的开支。ETL 工程师必须衡量这些关系。
- ETL 日志表：另外的方法是在 OLTP 环境下创建触发器，将新的独特的传统标识符和被改变的记录插入到一系列的特定的 ETL 日志表格中。这些

专门的日志独立存在以加速 ETL 处理，并且通过微处理 ETL 进程来决定哪一列与早先的微处理相比发生了改变。ETL 日志表格包含新的或是被改变的空间记录的独特标识符，还可能有状态值、时间戳以及一个对被改变的记录进行了最后操作的微处理 ETL 进程的运行标识符。微处理 ETL 进程将 ETL 日志表格加入到 ETL 运行标识符为空的可操作表格中，提取总值列然后删除(或者移入运行标识符)被提取的 ETL 日志表格。通过使用这种方法，由触发器驱动的插入行为不会过度使用 OLTP 系统，降低了企业花费在可操作系统上的费用。

- 数据库管理系统(DBMS)日志筛选：被作为有备份和补救作用的副产品而创建的 DBMS 审计日志文件，有时可以通过一种叫做日志筛选的特定功能来辨别新的和被改变的操作。有些日志-筛选功能可以及时地在某些特定点，有选择的提取和重建应用于数据库表格的 SQL 声明，不仅允许 ETL 知道哪些记录在最后一次提取后发生了改变，而且还知道这些记录中的什么元素发生了改变，信息可以被 ETL 进程用来直接在集结地修改目标表格。
- 网络探测器：这些功效可以监控某些我们感兴趣的网络流量，过滤并记录看到的这些流量。网络探测器经常被用来捕捉网络点击流量，因为它消除了从多台服务器中将网页日志联合在一起的需要，提供网络浏览者的 sessionizing 并且可以增加由动态网页反映出的实际网络内容的可见性。网络探测器是一种能够对在任何地方出现的要求数据仓储支持的流量进行选择的 ETL 过程，包括电信呼叫路由、制造业基础工作流程或者 EAI 通讯流量。

对于要求满足在每小时反应周期内都不出现在一个小时内维度更新的数据仓库需求来说，微处理 ETL 是个很不错的选择，而且它不要求在数据仓库和可操作系统间的维度数据的双向同步。显然这是提交近似实时数据仓储报告的最简单的方法。

企业应用集成

企业应用集成（EAI）是出现于复杂频谱的高端，有时也叫做功能集成。EAI

是一套支持真实应用程序综合化的技术，它允许每个独立的操作系统通过潜在的不同新方法进行互联操作。EAI 需要建立一套典型的能够通过消息的形式，在综合化网络中跨多种系统推动商业操作，并能够使该网络中所有系统从技术和独立性上完全独立与其他系统的适配器和 broker 组件。应用程序特定适配器负责处理所有创建和执行消息所必需的逻辑，brokers 负责以适当的方式并根据公告和条款规则对消息进行寻址。适配器和 brokers 之间通过独立程序消息进行通信，通常以 XML 来实施。当一次重大应用事件发生譬如顾客纪录的更新，触发器被触发，并且应用程序的适配器创造一则新消息。当适配器收到一个包含他所选择接受信息的消息时，比如从客户维度管理系统中得到的新近被确认的客户记录，适配器还负责在各个应用程序中启动相应操作。Broker 在适配器之间发送消息，消息的发送是基于标准协议。消息队列经常被部署在各应用和适配器之间，适配器与 broker 之间，形成异步消息的分段传递部署，保证了集成网络中传输和交易完整。

在图 11.7 中，应用 A 和应用 B 是独立运作的两个系统，但是它们之间可以通过 EAI 交互数据，相互操作。

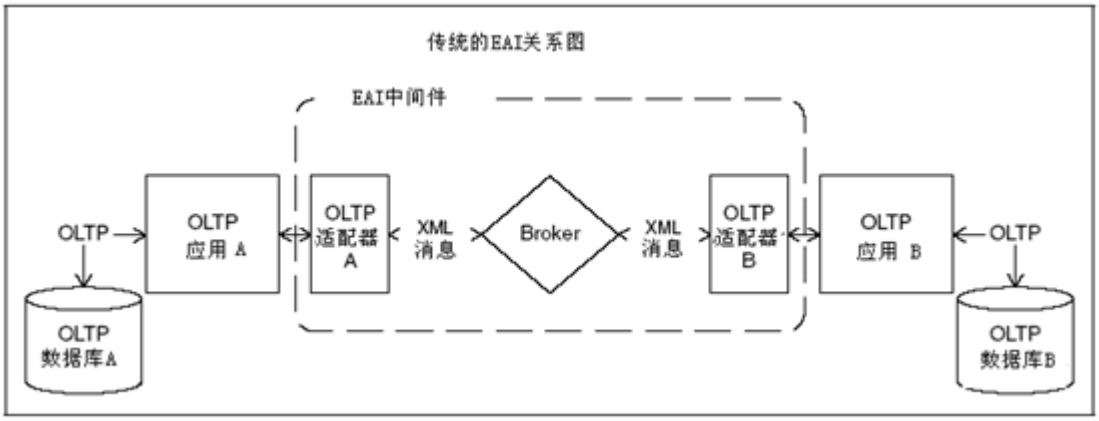


图 11.7 传统 EAI 架构

例如：修改系统 A 中一条客户信息，会触发到 A 的适配器，适配器生成一条描述该改变的 xml 格式消息，并把它发送到 broker，如果系统 B 从系统 A 处订阅了该种信息，那么 broker 就会向系统 B 的适配器发送该消息，适配器做出指令会影响全部或一部分有关该客户的信息。系统 AB 之间没有直接的交互，他们各自适配器负责捕获信息，翻译，做出指令。这是一个强有力的概念，他扩展了

EAI 网络。引入一个新的应用到 EAI 网络中，仅仅只需要一个新的适配器和规则。我们这里强调的“仅仅”，并不意味着适配器的作用是微不足道的。恰恰相反，每个适配器必须拥有在其主机系统上处理潜在复杂事务的能力，以及能够处理大量并行事务的能力，这些并行事务是由那些基于同一逻辑数据的应用产生的。假如不考虑集成方法，事务一定是在结构中的某个地方被处理，由 EAI 的适配器来做这件事是最佳的选择。

EAI 技术与实时数据仓库相结合，能发挥更大的功效。他们能够支持重要数据的同步处理，比如那些在各个应用上的客户信息。提供高效的方法，意味着源自信息资源的分布式数据仓库将横跨整个企业，譬如新客户的分割值。

实时-EAI-数据仓库通过维度管理将“单片”ETL 处理模块化。将维度管理划分为独立结构单元，每一个单元有自己的适配器，（数据集市适配器）负责数据集市实时分区的上的数据转换和装载事务。图 11.8 是关于实时 EAI 数据仓库的图示。

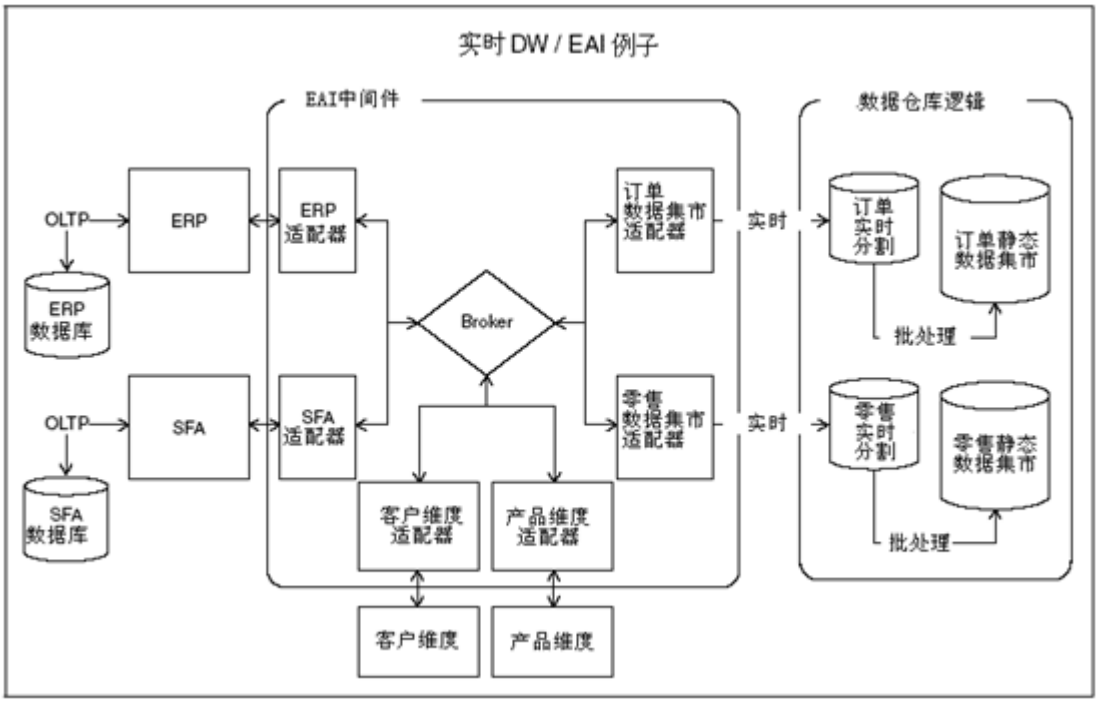


图 11.8　实时数据仓库 EAI 架构

一个典型的实际案例，可能涉及到一系列的 OLTP 系统，比如企业资源计划 ERP，销售自动化，客户产品维度管理系统（进行实时清洗和备份），订单销售主题的数据集市。

任何一个 OLTP 应用的客户或产品变化事务都会被适配器所捕获，发送不规则的维度消息给 broker,然后 broker 将维度消息发送给那些“订阅”了维度消息的系统，特别是维度管理系统。 系统将维度纪录以某种规则一致化，然后适配器将规则的维度消息返回给 broker，broker 再把规则的消息发送给所有订阅消息的系统，特别是数据集市和 OLPT 系统。

根据这个例子，ERP 系统更新一条客户纪录，ERP 的适配器检测到这个变化，生成标签为来源于 ERP 的非统一客户交易的 XML 格式消息，并把它发送给 broker. Broker 转发这条消息到客户维度管理，特别是从 ERP 订阅了不规则消息的系统。客户维度管理系统收到消息后将其放入工作队列（或堆栈），客户维度管理者处理这条事务，如果它导致一条或者多条规则客户纪录发生改变，适配器将捕捉到这些发生的变化，将修改后的客户纪录打包到源自客户维度管理的统一客户交易消息中，把他们发送到 broker。假定那些订单数据集市，销售数据集市，ERP，SFA 系统都订阅了源自客户维度管理的统一客户交易消息，那么 broker 将复制消息，并分发到所有的系统中。每个的系统的适配器对客户记录的变化作出反应，改变对应应用的纪录。

ERP 系统和 SFA 系统对统一后客户纪录的包容性，可以通过适配器来改变自己系统内部的客户数据改变，因此而触发新的一系列 EAI 事务。

在集成网络上，OLTP 系统或者维度管理系统上设计一个可选择的发布策略时，一定要避免无限循环，或 race 情形。

实际交易也会被 OLTP 适配器所捕获，作为订单和销售事实消息发送给 broker，然后 broker 再将消息发送给所有消息订阅者，特别是数据集市，数据集市适配器作所有需要的转变并将新的处理直接插入到数据集市的实时分割区。EAI 对于同步处理关键业务信息来说是一个强大的方法，逐步输入数据集市和发布，以及面向客户的 OLTP 系统的源自数据仓库的分布。但是，这种方法复杂而且实现的成本较高。对于那些有低延迟报告需求的组织来说，EAI 是一个非常好地解决办法。对于这种组织来说，当天维度更新数据的损失是不被允许的，或者要求数据仓库和操作型系统的维度数据的双向同步。

使用 EAI 构架来解决海量交易数据载入数据仓库，如果每个交易都是独立封

装成为一条 EAI 消息，每个消息占用很大的系统消耗，那么 EAI 方法显然会效率低下。在提交给这种设计方法之前，要明确是否会出现大量的消息通信甬堵，同样的也要明确你的 EAI broker 接口允许事务数据的契约交涉。

获取，转换，和流动

获取，转换，和流动(CTF) 是数据集成工具一种新的相关类别，这种数据集成工具被用来设计简化实时数据在不同数据库中转移。事务性应用的应用层是旁路的 bypassed. 相反地，直接库对库的交换被执行。事务，包括新事实和维度变化，可以直接快速从操作型系统向数据仓库进行数据移植，只需几秒钟时间。CTF 工具的转换功能基本上是基于与现今成熟 ETL 工具的对比之上的。所以实时数据仓库的 CTF 方案经常是操作环境中转移数据，使用 CTF 工具作数据转换，然后转移数据。数据转换工作包括数据格式的标准化，数据类型的重新定义，缩短或增长字段长度。一旦数据被转移，超出 CTF 工具能力的附加转换将会出现。后续的转换可以被小量批处理 ETL 调用或者通过触发器通过向传输区插入数据来触发。

在每个转换中，记录被直接写入到数据集市中的实时分区数据表中，这些并发的转换可能包含数据验证，维度数据清洗和匹配，维度数据的代理键查找，以及建立新的维记录需要慢慢改变，见图 11.9 所示。

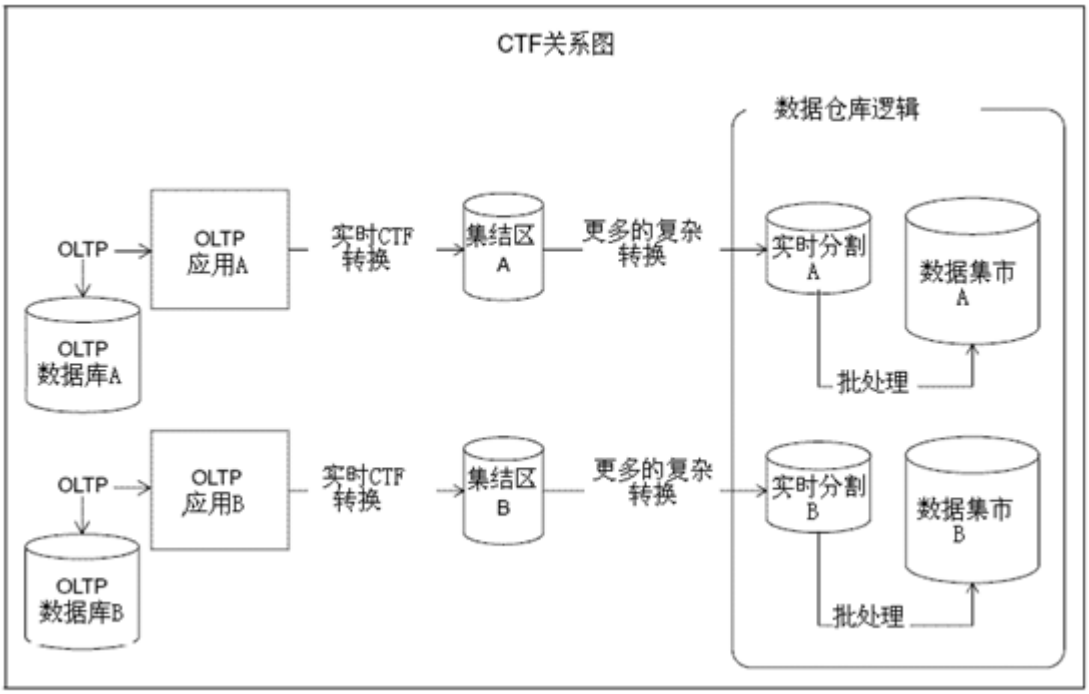


图 11.9 CTF 架构

一些 CTF 工具能够简化信息从数据仓库返回到操作性系统的批处理，例如客户数据的周期性更新。这是应为他们旁路 bypass 处理应用逻辑层。利用这些特征把批处理的负担转移到 CTF 管理员，来确保结果性更新数据不会混淆操作性系统的处理或交易事务的遗漏。

CTF 对于那些有实时上报，少量数据集成需求的组织来说是一个很好的方法，CTF 可以很好处理核心应用软件较低的共同活动时期,可以让数据同步发生混乱最低。CTF 融合了 EAI 的好处部分，同时避免了复杂。

企业信息集成

企业信息集成（EII）是另一种新方法来帮助企业较快提高商务智能系统的实时上报能力。在某种程度上讲，EII 是一种实质上的实时数据仓库，OLPT 系统中最近数据的逻辑视图，向业务用户展现适当的架构分析，通过 ETL 转换分发。

在传统 ETL 中，在 OLTP 系统中定义一系列的源结构，在数据仓库中定义目标结构。根据时间计划，一般是在晚上，触发 ETL 进程，ETL 工具抽取数据，转换格式，然后将其装载到数据仓库表中。EII 也是大致依照相同的原理来工作的，不同的是，EII 的目标文件可以是报表，电子表格，或者对象链接或嵌入数据库，或者 XML 对象。EII 触发器是当业务分析者需要更新第二个操作性信息时被触发。无论是否需要达到第二层操作型数据，EII 触发器都是通过业务分析员推动。EII 系统实际上产生了一系列的数据库查询，基本上是通过 SQL，在请求的那个时刻，申请所有被指定转换的结果数据，然后将数据释放给业务用户。

它的作用表现的相当有意思：一个零延迟报告引擎需要联合成熟的 ETL 工具数据集成的充足作用来增强。

还没有这样的先例：在 OLPT 系统中的数据同时存在于 EII 系统中。所以，描述趋势的报表还是要在数据仓库中产生。数据仓库本身也可以被定义为一个 EII 系统的源信息，所以，操作型系统的实时数据与数据仓库中历史数据的集成至少在理论上是可行的。EII 的转换能力，虽然很强大，但不是没有界限。它并不能在线支持所有的现代 ETL 和数据清洗功能（例如，模糊匹配），所以相应的降低了对他的预期。同样的，由于抽取的数据直接依靠 OLPT 系统，因此在抽取的频率和复杂度的管理上要参照在 OLPT 技术框架上的脚本大小的管理。

EII 的另一个优势是他的 ETL 继承性。EII 可以被看作是数据仓库的一个有效的原型设置。组织从同一个供应商那里选择 ETL 和 EII 工具，可能会发现成功的 EII 与数据仓库技术联系在一起的，数据仓库中的数据转换规则复用

如果将成功的 EII 主题转变成数据仓库主题范围可以重复利用在 EII 工具中开发的数据转换规则。组织选择从同一供应商提供的 ETL 和 EII 工具时，会发现成功的 EII 主题区域需要发展为数据仓库主题区域，后者可以通过复用已经成熟的 EII 工具的数据转换规则来推动。.EII 在全面数据仓库商务智能系统实时数据上报部分上也扮演着支持者的角色。在数据仓库和 EII 之间使用规则的维度和事实，作为维度化数据仓库总线结构的一部分，允许这些系统相互协作。On the fly，依照维度和事实，然而，说来容易做来难。在这种情况下，在最低程度上，你必须在 EII 查询上设置事实约束，拒绝已经装载数据仓库的事实来避免重复计算，联合新维度记录关注展现还没有加载进数据仓库的实时实际数据。

EII 对于那些要求零延迟实时上报的组织可能是一个非常关注的方法。对组织来说也许会有价值。那些组织相信他们需要发展实时数据仓库，但是并不确定它们的实时统计业务需求或业务需求改变的十分迅速。

最后，EII 对于改组中的企业也是引人注目的选择，它可以尽可能快的满足实时集成操作性上报。

实时维度管理

实时元（维度）管理，多次在本书中被提及，主要在客户信息中使用，转换新引入的客户信息，这些客户信息有可能是不完全，错误的或者冗余的。将这些客户插入到标准的客户记录中去。标准并不意味着完美，但是他们的维度纪录已经达到了组织能够达到的最好状态。在实际中，这就意味着所有的合理方法被用来消除冗余，除去脏数据，尽可能的将数据编译完整，同时指派代理数据键值。图 11.10 是实时维度管理的一个普通示意图。

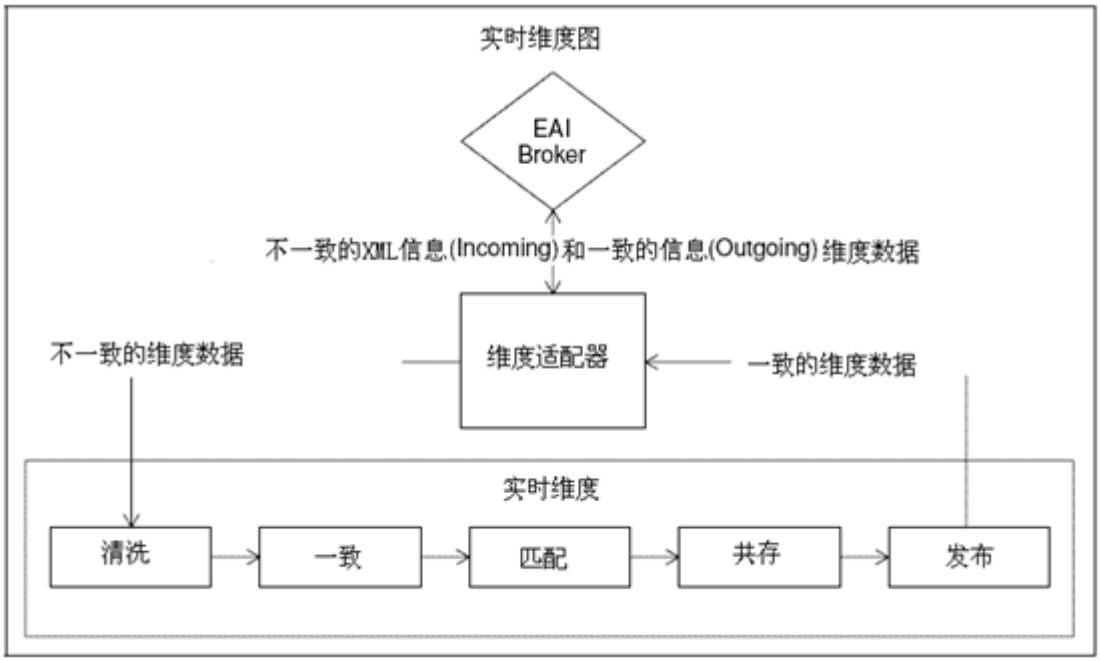


图 11.10 实时维度管理架构

EAI broker 是相同的 EAI 中间件组成部分，而中间件是在另一个 EAI 图表中被描述。.EAI broker 负责在适配器之间根据发出和接受规则“路由”消息。在维度管理中，EAI 是根据操作性系统的不规则维度零散信息改变到维度管理者进行路由处理，并将规则的维度改变返回需要接受信息的操作性系统上，或者是数据集市上。从实时维度管理得到的规则的客户信息一定也被包含在一系列 OLPT 系统的历史键值中，这些 OLTP 系统是被规则化进程连接在一起的。接着，这些键值被 OLTP 系统的适配器用来指认出如何应用规则化后的客户信息到各自的应用上。这些 OLTP 的结果性改变可以导致新纪录的产生，对原有数据的更新，或者两条或更多数据的合并，这些数据被维度数据管理认为是冗余的。在实时数据集市装载软件中也可以使用出现在引入到数据仓库替代键的事实记录的原有键值。

这种维度连续不断的变化是参与集成网络的各系统保持同步的一种机制。这种集成网络所发挥的作用能够超过简单的同步化。从 CEO 的观点来看，这些应用要协同工作。

除非是计划好的，一种反馈器能够在参与集成网络的各系统之间发挥作用。在一条非结构化维度纪录被维度管理者处理之后，应用生成一条维度信息记录自身的改变。你一定要小心处理这种反馈作用以及各种不利的处境，包括维度规范化信息的无限循环。一个稳固的 EAI 结构的重要性在于，能够解决这种类型的事

物。你一定要建立一套良好的策略来管理消息。以下是一个策略的例子：集成网络的各参与的应用，包括 OLTP 和维度管理者，在要更新现有的维度记录或生成新纪录时，这些应用选择发布消息。策略暗示了维度管理者，例如，不向所有的引入非结构化信息发布结构消息，只是那些可以引起一条或多条结构维度数据的改变的信息发布消息。同样的，一个 OLTP 系统接受了一个结构维度信息的话，只能是那种可以改变其维度表的那种信息。

维度管理者的适配器是 EAI 中间件的组件，适配器在实时维度管理系统中捕获各种非标准化数据的消息，并把他们放入维度管理者的队列中，监听发布事件，并被维度管理者所出发。随后，这些被转换成为 XML 格式，并发布到 broker。维度管理适配器隔绝了其他的 EAI 结构和任何 awareness 或者实时维度管理系统本身。

根据图 11.10，实际的实时规则维度，是下列子部分的典型模块。

- 清洗：清洗部分读取引入的不规则数据，从工作流中抛弃那些残缺的或非法的维度实例。他保证了查询可以被展示出来，这种查询有可能是跨越不同数据源的。还可以保证包含在属性里的数据值的合法性。
- 规范化：规范化部分从工作流中获取清洗后的数据，执行域对域转换。这种转换是原始系统经过验征后的值转换到一个规范化致域，是一系列的企业级数据根据各自维度属性所作的变换。在某种情况下，规范化的值可以到达目的地，通过查找表映射所有的源系统值到一致值。另一方面，规范化值的到达是概然说，使用模糊统计计算获得起始关系。符合概然说标准的特殊工具通常是被用来清洗整理街道地址的。把地址整理成为已知的格式；更正街道的拼写，城市和州的名称；更正邮政编码。其它一些模糊工具还可以更正名字的错误拼写；标准化常用名称。
- 匹配：匹配部分从工作流中获取清洗一致化后的数据，识别并排除冗余数据。专门的软件工具被用来帮助执行匹配功能，使用明确论和概然论的匹配技术。关于这些工具工作的详细说明超出了本章的描述标准，本文在这里只能说，这些工具是开发者定义很多匹配规则和匹配运算。每个通过和新产生的一个联合记录的匹配，匹配引擎记录这些匹配的相似

性，这是全面模糊性匹配的一种平衡。最后的记录是比较由维度管理者定义的高（明确匹配）低（明确不匹配）两种极端，还包括定义的匹配组键值。在高低端值之间的纪录用特殊的方法来标记，特别是回顾指南。匹配的定义和高低端值的设定是一个半科学半艺术的工作。他很大程度上依赖于企业对于错误的容忍性，低延迟的需要，政策和调整需要考虑的事项，和回顾指南人员构成。

不足匹配（保持一致但是客户记录与分离的维度管理不恒等）大体上被认为比过渡匹配更保守而且较低插入，并且被认为是规范的。在实时 EAI 环境中消除一个错误匹配不是那么容易完成的事情，而且常常意味着客户处理已经被 OLTP 系统错误整理过了，是在 OLTP 系统对维度管理者手工分离和中继转发的融合请求做出反应时作出的错误处理。

在实时环境中，需要手工维护的纪录对于不匹配状态是一种典型的缺省状态，而且手工维护可以延迟执行。由于性能的原因，当处理大维度时例如零售业的客户维度，一定要限定候选键,使其达到合理的实时性能。抽取候选键可以大幅度加快匹配处理，但是有时会在传递用来匹配的候选纪录时出错。因此，实时在线匹配有时会危及安全，同时，整个维度的周期性重新匹配也许是必需的。这种单片集成电路的重新匹配处理能够创造大量的一致维度信息和操作性系统的更新，这些都是你需要小心处理的。特殊元数据是匹配部分中用来描述匹配变化逻辑，匹配极限，和匹配过载信息。过载匹配是指那些永远不需要匹配或永远要匹配的纪录。我们经常需要开发一个专门的用户接口来支持手工匹配处理和手工维护的元数据，也包括变化逻辑，匹配极端和候选键处理。

- 共存：一旦纪录被认为是做匹配，维度的最好反映一定是从匹配纪录蒸馏而来的用来建立完整和正确的组合映像。一组记录一旦被定义成另一匹配，维度的最优值必须从创建完整和精确合成值的匹配记录中萃取出来。这种萃取处理经常被作为共存被提及，这是因为他保证了最好的维度属性源在全部维度规范化处理中的存在。存在处理利用业务逻辑，在属性对属性基础上，识别源系统优先权被应用在存在规范化维度映像。这种共存处理利用确定的业务规则，基于属性对属性，源系统在已存在提供的统一的维度值中排列优先顺序。非空源属性值存在于最终的完整

的维度纪录中，这些记录是理想元数据基于层级源系统规则所捕获。存在部分也应该支持属性的定义组，这些定义组是当同源纪录作为避免摘取奇怪数据的阻碍时存在的，例如，地址行 1 和地址行 2 是来自于不同源纪录的。存在模块也能够处理明显的时间点代理键的产生。

所以当一个客户纪录变化了 10 次之后一定会有 10 个不同的代理键值。然而每一个都应含有相同的全面客户代理键。这两种键值的处理方法都是必须的，。因为，维度管理是为两大部分服务的：数据集市，数据集市必须使用时间点代理键；OLTP 系统，它只需要维度纪录的同代映射。

- 发布：一旦维度映像充分集成（清洗，一致化，匹配和存在），你必须要决定作为结果存在的数据是否为全新或与先前的维度映像是否不同，从而授权发布给集成网络。大部分维度管理者会选择性发布来防止无限发布循环。如果发布是经过授权的，发布部分会唤起维度管理者的适配器，该适配器一直处于连续监听发布请求的状态，所以他可以收集所有的或一部分的维度纪录，并把它转换成一致的 XML 维度消息。将消息发送到 EAI broker 中在企业内部分配。唤起维度管理者适配器通常会占用表空间，表空间是被应用来在特殊存储中一致化数据或触法监听适配器时更新一条活多条记录的。

设计发布规则要保证足够的维度同步化，它有可能贯穿在整个企业中。同时还要避免无穷回应或空转条件，避免钻取到详细发布应用和规则，这些可以被 EAI broker 轻易处理。

当 ETL 构架师设计实时维度管理者时一样要小心处理分析业务需求而且有时要勇于处理关于组织政策的阻力。很多寻求真实设想版本的经理人， 而不是其他人。技术上，架构师必须判定什么时候什么地点完成全部或部分维度纪录，这些记录已经在应用间一消息的方式被传递。这种情况应该会引起一致化数据的发布。如何处理可能的争论和空驶条件，如何最好自动平衡应用需求，是否使用较少磁盘的直接贯穿方式还是微批处理方式？这些我们将在下一部分讨论。

听起来很复杂？是的，是这么复杂。但是实时维度经理对于企业需要在整个企业范围同步丰富客户信息或其他关键维度信息来说，是一个真实的强有力的办

法。给那些早期采用者提供了竞争优势。

小批处理过程

当设计实时数据集市分区或维度系统时，你经常会面对一个普遍困难选择：解决方案应该包含直接处理或利用频繁的微批处理？目前的工具都是支持低延迟数据环境的，比如说 CTF，常常缺乏已经存在的 ETL 工具的转换能力。一批 ETL 工具提供商已经开始提供他们工具的实时版本了，以更加事务性的方式处理信息，但有时功能会受到限制。实时维度管理系统的设计者，当选择明确或概然论匹配工具时，也常常会面对类似的困境：一些工具专门已批处理方式操作。你如何使批处理方式工具在实时环境中性能最佳？

对于那些面向批处理工具的接近实时传递性能，一个合理的折衷方法是设计一套方案来处理频度发生的微批处理，使用标准转变工作来控制元数据结构。

考虑图 11.11。在工作表中的每一件工作代表了任意一个或非标准维度处理纪录，对任意一个实时数据集市或维度管理者。这些工作被几个处理所贯穿，每个都在发布之前定义了处理表，例如清洗和一致化。在表 11.11 中的数据模型里，微批处理表代表了小批量的既定处理，每个微批处理都完成一定的工作。每个微批处理所完成的工作会被工作处理事件表所捕获，也会捕获工作事件状态属性成功或失败标志。

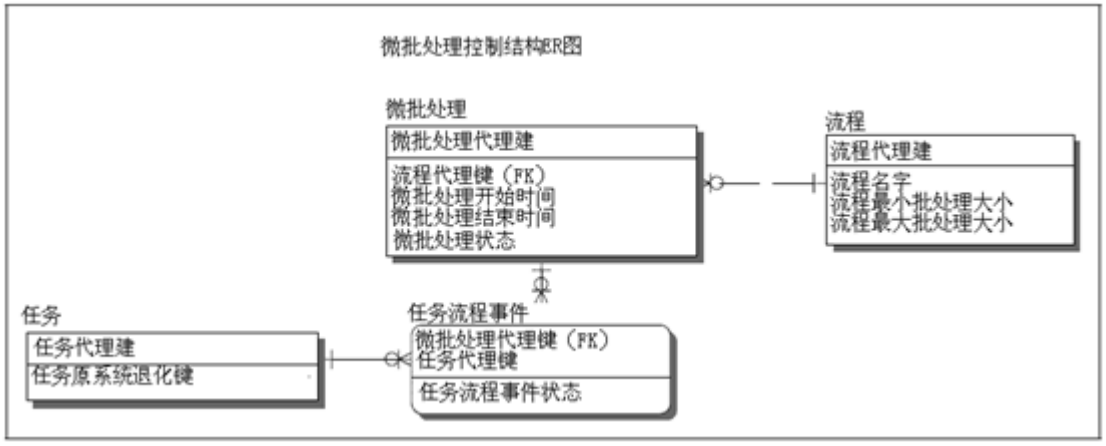


图 11.11　小批处理数据模型

处理连续执行，就像邮件收发的后台程序，寻找那些适当处理状态的工具。当他们发现可接受的最低数量工作就会调用微批处理来处理工作，生成工作处理

事件纪录及其适当状态值；然后他们继续寻找更多的工作来处理

建立模块处理后台程序的好处，与使用直接贯穿处理相反，好处在于他们可以独立发展，有独立可调的批处理规格，可以作为具有新特性的新工具被替换和独立加固升级。同样的，新处理可以轻易的被加入到工作流中，而且需要选择性处理的工作可以轻易的实现调节。例如专门的地址确认或信用价值评分。但是这种弹性消耗成本。微批处理流要求每个处理详细定义和表述接口，特别是数据库表的来源和去向。这种附加的 I/O 和强加的复杂性的影响是很明显的。在实际当中，可以通过设计个别的工作表来降低复杂度，工作表被用来作为所有处理的共同源头和目的。可以数据库内存中缓存来减少 I/O 压力。不过，微批量的方法和直接处理的执行不一样。控制性纪录和全部工作将被频繁分开以保证他们尽可能的小。一个典型微批处理的每个处理分支都是非常简单的。当每个批处理被调用时，将生成一条微批处理纪录，同时生成一定数量的工作事件处理纪录（处理表中最低数量和最高数量的详细批处理数量）。这种基本技巧给管理大量并发批处理提供足够信息，也保证了所有批量处理和工作处理有足够的审计信息。表 11.12 说明了如何工作。

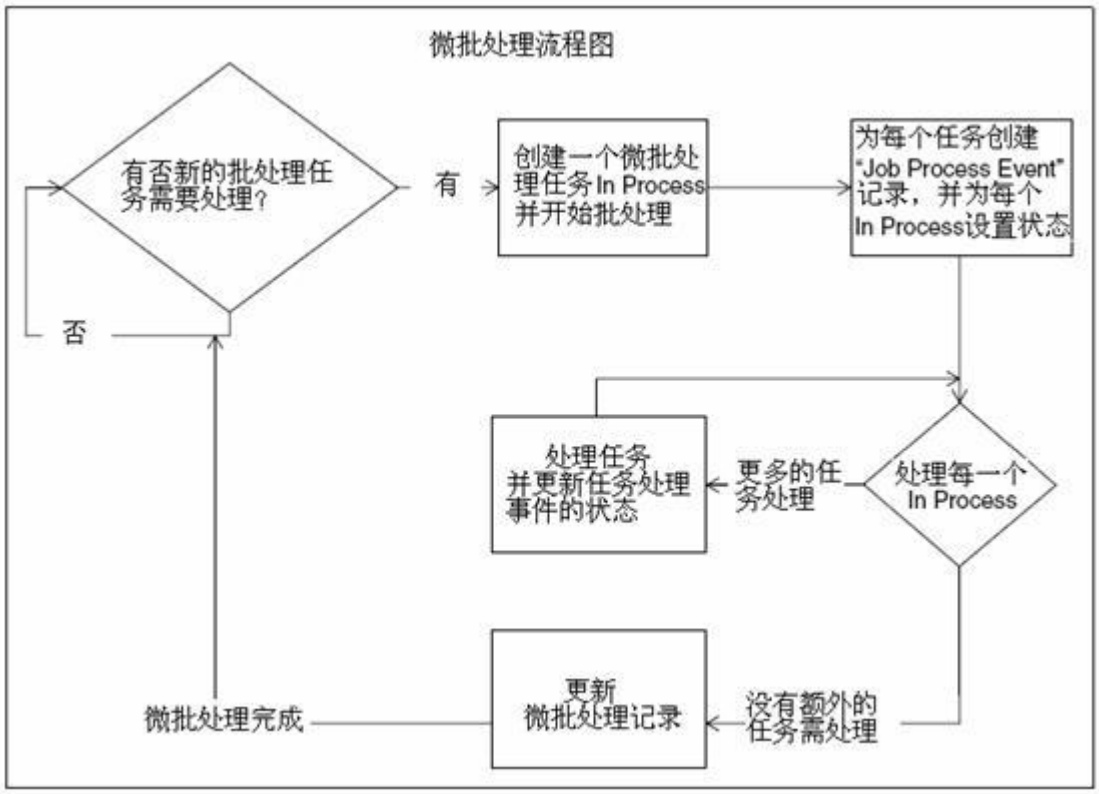


图 11.12　小批处理流图

表 11.12 描述了一个单一处理。各个处理连续运行，而且像邮件收发后台程序一样和其他处理共同完成同一工作，因此设置工作处理事件和微批处理状态值，然后继续。所以，一个维度管理者的数据清洗，一致化，匹配，存在和发布处理后台处理有时会在不同的工作任务上同时发生。一个数据集市的实时 CTF 系统可能会存在转换过程和代理键查找后台程序，等等。每个后台程序连续的查找任务进行处理。我们这篇文章中所讲的任务工作是指在给定处理中以最优方式处理的阶段，已经被处理过的任务纪录。

正如在图 11.12 中所看到的，任务处理事件的状态在处理中被设置，而且一个关系数据库处理的起始点是确定的。假如每个任务都处理过了（清洗，一致化等等），任务处理事件状态被更新为成功或失败。作为更替降低处理成本，这种更新发生在批处理结束时。一旦所有的任务都执行过，批处理完成，批处理控制表被更新。假如成功执行没有致命错误，在处理中加固所有的处理事件纪录，然后执行提交，转换结果被写入到数据库中。如果有失败发生或一个无法接受的大数量任务处理事件状态为失败，将会执行回滚处理，数据库将会回复到微批处理执行之前的状态。

回滚事件并不一定回滚错误信息或控制表中的状态值。很多关系型数据库提供了处理控制选项来支持这种约束。

应用于实时维度管理者的微批处理 ETL 在图 11.13 中表述。作为系列后台处理，从数据库表中读取，更新控制，分段运输，以及一致化的资料库。

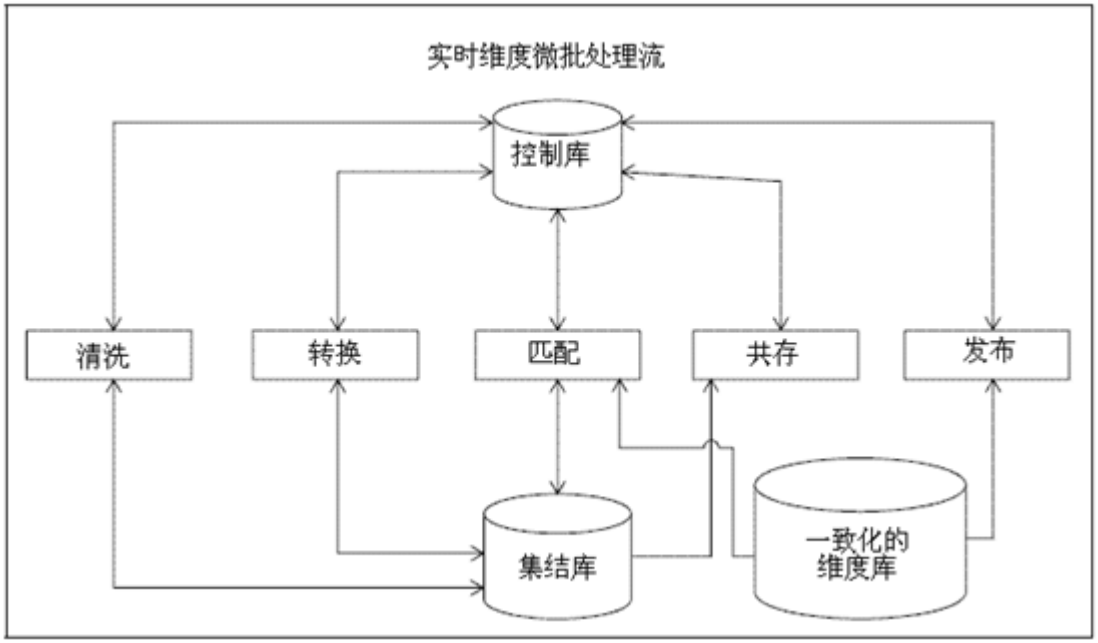


图 11.13　实时维度管理器的小批处理流

每个处理在任务事件表中更新状态值，在分段传输阶段修改和新建数据，或一致化维度。

- 清洗过程从分段传输库中读取未经验证的数据，并只向控制库中写入状态值。
- 在规范化过程中，从分段传输读取清洗后的没有一致化的数据，并向分段传输写入一致值来保证一致属性。
- 匹配阶段从分段传输读取一致化后的未经匹配的数据，向分段传输写入匹配键值。
- 存在阶段从分段传输读取经过匹配的未存在的纪录，并向一致化维度库中插入或更新纪录。
- 发布阶段从一致化维度库中读取经过一致化的纪录，唤起维度管理者适配器，然后适配器将纪录发布到集成网络中。

一个微批处理 ETL 系统也可以被用来转换 CTF 到实时数据仓库，这时的数据转换处理会比那些只被 CTF 工具支持的数据转换要复杂得多。CTF 可以被用来实现操作系统的近实时数据抽取和轻度转换服务，在分段传输阶段删除数据。从分段传输阶段，微批处理开始处理复杂转换和逐步插入到到实时数据集市分区表

中。同时支持实时数据上报和正规的夜间批处理使数据装载到统计数据集市表中，在处理过程中清空实时分区表，为第二天的转换作准备。

选择一个方法－－－一个决定向导

目前，实时数据仓库的整个区域还是十分混乱的。有非常多的方法和技术可以选择，被各种厂商和评论家所包围，但是在实际应用中没有多少成功的案例可供分析学习。所以选择一个适当的构架和方法是一个非常令人畏缩的任务。

下列的表格试图分析我们在这一章节讨论过的方法，形成帮助我们缩小选择范围的指导方针。表 11.1 是实时报告方法的对比矩阵。

表 11.1　实时报告决策向导矩阵

对比矩阵		EII ONLY	EII+ STATIC DW	ETL	CTF	CTF-MB-ETL	MB-ETL	EAI
		实时数据仓库里的企业综合信息	和传统的非实时数据仓库一致的企业综合信息	标准 ETL 过程	捕获，转换，传输到实时数据仓库	捕获，转换，ETL 微批处理传输到实时数据仓库	ETL 微批处理传输到实时数据仓库	企业应用传输到综合实时数据仓库
支持历史数据		√	√	√	√	√	√	
复杂报告综合数据	弱	√	√	√	√	√	√	√
	适中	√	√	√	√	√	√	√
	强	√	√	√		√	√	√
数据更新最大间隔	1 分钟	√	√					√
	15 分钟	√	√		√			√
	1 小时	√	√		√	√	√	√
	1 天	√	√	√	√	√	√	√

表 11.2 提供了实时维度管理系统的对比矩阵，他们都需要实时应用集成，可以通过批数据集成得到。

表 11.2　维度验证决策向导矩阵

对比矩阵		ETL	CTF	CTF-MB-ETL	MB-ETL	EAI	EAI-MB-ETL
		标准 ETL 过程	捕获，转换，传输到维度管理系统	捕获，转换，ETL 微批处理传输到维度管理系统	ETL 微批处理传输到维度管理系统	企业综合应用传输到维度管理系统	企业综合应用和 ETL 微批处理传输到维度管理系统
复杂报告综合数据	弱	√	√	√	√	√	√
	适中	√	√	√	√	√	√
	强	√		√	√	√	√
数据更新最大间隔	1 分钟	√				√	
	15 分钟		√			√	√
	1 小时		√	√	√	√	√
	1 天	√	√	√	√	√	√
企业综合	仅数据集市	√	√	√	√	√	√
	“虚”数据综合		√	√		√	√
	“实”数据综合		√			√	√
	应用综合					√	√

小结

实时 ETL 并不仅仅是一种潮流或是新特征。向实时数据传输迁移对于 ETL 管道的各个方面来说都是挑战，这种挑战是关于物理上的和逻辑上的。也许实时系统的最好设备是用流动 ETL 替代面向批操作的 ETL。在这一章中，我们介绍了实时 ETL 实践方法的艺术，并且指出了我们可能会面临的风险。

13 结论

为一个数据仓库设计并构建一个 ETL 系统是一种需要保持上述观点的演习，这是一种典型的具有复杂的后台却需要在开始就进行全面计划的任务。这很容易从某个源转换数据并立即放入那些可以查询的表中。然而，用户并不期望得到这些原始的结果，因为这样的做法没有扩展性，也不可管理的。

13.1 深化 ETL 的定义

我们回顾一下第一章中介绍的必须考虑的条件。这些包括业务需求，兼容性需求，数据评估结果，如安全性、数据一致性、数据潜伏期、压缩和血缘跟踪，以及用户使用的工具。你们也必须精通你们现有的技术和你们现有的遗留许可。是的，这是一个超越限制的问题。

如果你们时时将这些牢记在心，你们必须做一个很大的抉择：是买一个综合的 ETL 工具还是你们自己写脚本和程序来做？对任何一种选择来说我们都已经努力将这本书不要变得太厚，但是我们认为范围越大，项目周期越长，厂商提供的 ETL 工具就更适合。你们的工作是准备数据，不是做一个软件开发经理。

从我们的观点来说，这本书的实际价值是抽取、转换和加载这三个标准步骤的结构。这本书介绍了一系列为每个步骤设计的特定技术。这不是一本包含所有可能应用来创建一个 ETL 系统的书！我们已经将传统的 ETL 步骤分解成四个步骤：抽取，清洗，规范化，和提交。这本书中的这四个步骤交付的内容是特别区分的，包括：

- 抽取：为 ETL 系统选择特定的原始数据源并结合逻辑数据映射和数据评估结果成为一个计划的方法。通常从源系统开始，我们也建议某些转换在这里处理而不是在更加传统的清洗步骤去做。
- 清洗：为一个错误事件事实表，一个审计维度和一系列的数据质量筛子设计框架。我们说明如何使这些内容有效的整合到 ETL 系统中。
- 规范化：精确的定义一致的维度和一致的事实（通过充分说明维度管理

者的责任和义务以及维度和事实的公共策略）。规范化是现在在业界被称为主数据管理的基础。

- 提交：为所有范围的维度模型来细化结构化指定部分，包括缓慢变化维度，主事实表类型，以及为多值维度和层次结构所需的桥连接表。我们展示了如何设计所有的维度框架种类，同时我们提供为每个这些情形的特殊代理键的管理方法。

每一个步骤的内容提供了 ETL 元数据的基础。通过提供实际数据的状态来降低了处理 ETL 元数据的许多疑惑和困难，在清洗步骤中的审计维度直接验证这个观点。由于维度总是用来描述上下文的度量，我们发现在 ETL 系统中的提交部分，交付一个表就是另一种上下文。以这种思想，我们仔细的将所有的数据附加各种审计维度，就像用户用他们熟悉的工具一样。

在第 7 章—开发，我们介绍了许多你们需要的特殊的转换步骤和工具来建立 ETL 系统。如果你们自己写代码，我们提供的例子代码也是直接相关的。如果你们已经购买了 ETL 工具包，大多数的这些步骤和工具都可以在图形化的界面中表达你们的 ETL 数据流程。在第 7 章的第二部分，我们给出了 DBMS 特有技术的一些指南来提高性能，如高速的块加载，强制参考完整性，并行处理的好处，计算维度聚合以及性能问题纠错。

在第 8 章—操作，我们开始全面介绍在 ETL 环境中的作业调度，记住每个环境都有其独特的瓶颈。我们于是提出特定的控制文件来帮助你们进行每天的 ETL 系统管理。这些包括一个数据集市版本文件，一个 ETL 性能跟踪文件，以及一个使用的度量列表。我们对第 8 章的结论是推荐安全的和压缩的架构。

在第 11 章，我们开始介绍设计一个实时的数据仓库系统。实时对你们目前的 ETL 来说是太快了。但是更进一步的，迁移到实时模式几乎总是需要从批处理 ETL 到流程 ETL 的跳跃。当做这种跳跃时，可能你们的 ETL 系统的每一步和你们的用户工具将需要重新设计。显然，这是不能轻视的一个步骤。然而，几乎所有的批处理 ETL 中的重要步骤都必须在流程 ETL 设计中被用到。你们仍然需要抽取、清洗、规范化和提交。因为这些原因，我们在前 10 章的课程开发中作为实时设计的基础。

13.2 数据仓库和 ETL 的未来

IT 事实上只有两个相辅相成任务：获得数据和输出数据。获得的数据,也就是事务处理。过去 30 年来,很多组织已经投资了万亿美元以上的花费，来建立日剧增长的交易处理系统，这些系统是以操作性为目标来获取数据。但数据不能单向流动：在某些点上,我们必须消化数据，并从中获得价值。在商业界，有一个深刻的文化思维：只要我们能够看到所有的数据,我们就可以更有效地管理企业。这一文化思维如此根深蒂固, 我们理所当然地承认. 然而这是数据仓库的任务,并且这就是为什么数据仓库在我们所有的组织中是一个永久的实体,即使它有些改变。从这个角度来说, 最终,在输出数据的总投资将能与获得数据相匹敌，这似乎是合理的。

在过去五年里,许多重要的观点已经成为数据仓库的动力：

- 数据仓库的蜜月期已经结束. 企业对技术已经失去耐心,他们坚持说从数据仓库可以获得有用的商务结果. 这个主题的名字，至少就现在来说，就称为商务智能（BI）。BI 是由最终用户趋动的，并且 BI 的供应商都控制用户看到的最终显示结果界面。
- 数据仓库已经成为鲜明的操作业务.传统的数据仓库和业务型报表之间距离已经消失.这样，以操作型为中心引起了数据仓库两个大的需求。第一，数据仓库必须能处理企业中的原子交易。如果你要看某个订单是否已经发货，你不能看汇总数据. 数据仓库中的每个主题区域在单独的交易水平上必须有好的途径去获取原子数据。第二,企业中的许多业务观点必须能够实现实时操作，当然,在本书中，我们已经深入地谈到了对实时的定义和对实时的技术支持的挑战。
- 企业希望对他们的业务状况有一个全方位的了解（360 度的视图），而需要全方位了解的信息中最重要的是客户方面的信息。每一个关于客户的信息在企业的数据仓库中是期望能够得到的，而终端用户需要的是一个包含所有相关客户信息的统一视图。这对于数据仓库的数据清洗和规范化是一个巨大的负担，特别是如果没有考虑到操作系统中所有客户视图的合理化。虽然客户是趋使全方位需求的最重要的因素，但是产品、人

口、供应商在其它的环境中也具有相同的挑战。

- 最后,数据的爆炸性增长有增无减。在数据获取(尤其 RFIDs)和数据存储技术的进步正在使我们的数据仓库陷入不利境况，引起对建立可用于分析的每个数据粒度的期望。

因此，这些主题如何改变原来的 ETL 任务？

我们认为,最突出的现实问题就是开发和运行 ETL 系统的复杂性。正如我们所说的，这是一个约束过多的问题。再一次阅读第一章的需求列表，当数据的数据量、软件、硬件处理数量的迅速增长，编写你自己的系统是越来越不可行的。将来的系统是允许你集合高水平的建立逻辑块。

当今 ETL 系统的演变

其他技术领域也已经经历了类似的阶段，这个阶段复杂的开端只是简单地集中在工具集成的水平更加复杂，一个芯片的数百万集成电路设计和数百万行的代码的软件发展就是这个演变的例子。如果我们要保持输入数据量的不断增加，ETL 处理的发展不可避免地必须经历相同的过程。

这意味着 ETL 设计者必须渐渐地以系统集成、系统监控和系统建立块的集成为向导，而不是编程。简单来说，是因为没有足够的时间去做低水平的编程。

分析原子数据的主题更加明确，并将会加速发展。市场细化已经细分到个别的用户水平，并且市场分析将要实现能够查询到基于非常复杂特性的组合和连续行为的独立客户集。第六章中，当我们把事实文本放置在客户维度的时间序列位置上，我们看到分析系列行为的挑战。 再一次，我们重复我们的基本信念------必须重视 ETL 系统的主键分析模式，比如为了使终端用户的应用程序可用的系列行为分析。ETL 系统就像是一个精美的餐厅中的厨房：ETL 系统必须在食物端出厨房之前安排整理好“盘子”。

系列行为分析也会对查询分发系统造成更多的压力。RFID 标签就像是旅程中通过的门。每一个门就是数据收集的设备，它记录着 RFID 标签的通道。只有当不同的数据库中每一个门可能被集合成一个统一的数据视图时，才可能存在系列行为分析。这样，“旅程中”单独的 RFID 标签和整组的标签就可以被分析。显然

易见这是一个合并和规范化的挑战。最近的疯牛病恐慌是这个问题的一个很好的例子。每只牛上的 RFID 标已经标上，但是，没有人能够分析某只特定的有问题的牛从哪里来或者在哪，因为不同的 RFID 产生的数据库不能轻易获得或者被整合。

最后,应该回到这本书贯穿全文的主题，实际上也是作者主旨。数据仓库的价值已经能够响应企业的商业需求。在最后的分析， ETL 系统设计最重要的的特征是以商业为中心，以正确的导向来最有效的分发数据仓库的任务：输出数据为目标，从而来保持数据仓库的高水平的系统技能。