第二部分　数据流

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。

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

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。

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
```

^^^^000001500001980000029700 0

20000406005City Electronics 00008AMPLIFIER

^^^^000003500002610000091350 0

20000407029USA Audio and Video 00017KARAOKE MACHINE

^^^^000002000000882000017640 0

20000410010Computer Audio and Video 00017KARAOKE MACHINE

^^^^000001000000882000008820 0

20000411002Computer Audio and Video 00017KARAOKE MACHINE

^^^^000003500000882000030870 0

20000411011Computer Audio and Video 00008AMPLIFIER

^^^^000001000002610000026100 0

20000415019Computer Discount 00018CASSETTE PLAYER/RECORDER

^^^^000002000000684000013680 0

20000418013Wolfe"s Discount 00014SUBWOOFER

^^^^000002500001980000049500 0

20000418022USA Audio and Video 00008AMPLIFIER

^^^^000001500002610000039150 0

20000419010Computer Audio and Video 00023MP3 PLAYER

^^^^000001000001764000017640 0

20000419014Edgewood Audio and Video 00006CD/DVD PLAYER

^^^^000002000004410000088200 0

20000419016Computer Audio and Video 00014SUBWOOFER

^^^^000003000001980000059400 0

20000419021Computer Audio and Video 00014SUBWOOFER

^^^^000003500001980000069300 0

20000419028Bayshore Electronics 00020CD WALKMAN

^^^^000001500000414000006210 0

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。

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：
  - 源文件的位置

- 列和字段布局的说明
- 数据类型的说明
- 从源到目的的数据影射
- 源数据约束
- 缺失数据的默认说明
- 去除空格和 tab 键的方法说明
- 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。

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)

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。

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。

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。