大数据技术丛书

Greenplum

构建实时数据仓库实践

王雪迎 著

清華大學出版社
北京

内 容 简 介

Greenplum 分布式数据库具有可选存储模式、事务支持、并行查询与数据装载、容错与故障转移、数据库统计、过程化语言扩展等方面的功能特性，因此 Greenplum 成为一款理想的分析型数据库产品。本书详解 Greenplum 数据仓库构建与数据分析技术，配套示例源码。

本书共分 10 章。内容包括数据仓库简介、数据仓库设计基础、Greenplum 与数据仓库、Greenplum 安装部署、实时数据同步、实时数据装载、维度表技术、事实表技术、Greenplum 运维与监控、集成机器学习库 MADlib。

本书适合 Greenplum 初学者、大数据分析系统设计与开发、数据仓库系统设计与开发、DBA、架构师等相关技术人员阅读，也适合高等院校大数据相关专业的师生作为实训教材。

图书在版编目（CIP）数据

Greenplum 构建实时数据仓库实践 / 王雪迎著. —北京：清华大学出版社，2022.7
（大数据技术丛书）
ISBN 978-7-302-61165-3

Ⅰ. ①G… Ⅱ. ①王… Ⅲ. ①关系数据库系统 Ⅳ. ①TP311.132.3

中国版本图书馆 CIP 数据核字（2022）第 110482 号

责任编辑：夏毓彦
封面设计：王　翔
责任校对：闫秀华
责任印制：
出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编：**100084
社 总 机：010-83470000　　**邮　　购：**010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn

印 装 者：
经　　销：全国新华书店
开　　本：190mm×260mm　　**印　　张：**22.5　　**字　　数：**576 千字
版　　次：2022 年 8 月第 1 版　　**印　　次：**2022 年 8 月第 1 次印刷
定　　价：89.00 元

产品编号：086668-01

推　荐　序

自从人类诞生以来，人们利用不同形式的生产工具推动社会不断向前发展，继农业时代、工业时代之后，随着科技革命和产业变革的深入演进，当今已进入数字时代的快速发展时期。数字经济以数据资源作为关键生产要素、以现代信息网络作为重要载体、以信息通信技术的有效使用作为效率提升和经济结构优化的重要推动力。数字化转型正在驱动生产方式、生活方式和治理方式发生深刻的变革，对世界经济、政治和科技格局产生深远影响。

数据要素是数字经济深化发展的核心引擎，企业作为数字经济发展的主力军，在研发生产、企业管理及市场经营的时候，一方面产生了各类海量数据，另一方面又迫切需要从海量聚集的数据中挖掘数据价值，从而推动技术、产品及各种模式的创新，增强企业市场竞争力，为全社会的经济发展带来强劲推动力。

中国作为全球数字时代的领先者，各领域的高速发展迫切需要在数据管理、数据利用等方面有更加高效、智能的工具平台，降低经济体在数据治理方面的成本，提升数据服务质量和体验。

王雪迎作为国内数据工程领域的实践者和教育者，在数据领域已从业近三十多年，在工作中不断跟踪、实践和总结最新的数据技术。2021 年 3 月，公司组织专家研讨行业数据应用技术方案，我对王雪迎抱怨，目前市面上的各类数据库、数据挖掘软件五花八门，对中小企业来讲难以选择，OLTP、OLAP 以及 AI 三位一体的实施应用成本太高，制约了中小企业在数据利用方面的能力。2022 年年初，王雪迎说通过一年的探索验证，总结了一些新经验编撰成书并即将出版，这就是《Greenplum 构建实时数据仓库实践》。

世间万物在变化，市场和客户在变化，产品和技术在变化，经营环境在变化，企业本身也在变化。在不断变化中，如何用一种优雅、简洁和高效的方式，通过数据感知变化，将是数字时代下每个数据从业者都要思考和面临的问题。

这本由清华大学出版社出版的《Greenplum 构建实时数据仓库实践》，行文流畅、案例典型、体系完美，理论与实践相结合。希望它能为国内数据工程领域带来了“与时俱进”的变化，也希望这种变化为我们广大从业者带来更多的提升可能，从而实现各自更大的价值。

杨兴兵
北京世元科技有限公司总经理
2022 年 3 月

前　言

从 Bill Inmon 在 1991 年提出数据仓库的概念起，至今已有三十年的时间。在这期间人们所面对的数据，以及处理数据的方法都发生了翻天覆地的变化。起初数据仓库系统运行在单机或小型集群之上，程序以批处理方式周期性运行 ETL 作业。最为常见的执行方式是在每天业务低峰期处理前一天产生的业务数据，即所谓的 T+1 模式。后来随着互联网和移动终端等应用的普及，需要处理的数据量不断增大，出现了大数据的概念，以 Hadoop 及其生态圈组件为代表的新一代分布式大数据处理平台逐渐流行。近年来随着业务领域的不断拓展，人们对数据分析的实时性要求越来越高，离线批处理方式所产生的延时已不能满足需求。以 Hadoop 为代表的分布式框架并没有给出实时计算解决方案，于是便出现了 Storm、Spark Streaming、Flink 等实时计算框架，可提供秒级的响应时间，在此基础上实时数据仓库应运而生。

作为 DBA，我更倾向于采用一种不编程、组件少、门槛低、易上手、纯 SQL，并能处理包含历史全量数据的方案，用来实现实时数据仓库。不可否认，SQL 仍然是数据库、数据仓库中最常使用的开发语言，也是传统数据库工程师或 DBA 的必会语言，从它出现至今一直被广泛使用。首先，SQL 有坚实的关系代数作为理论基础，经过几十年的积累，查询优化器已经相当成熟。再者，对于开发者，SQL 作为典型的非过程语言，其语法相对简单，但语义却相当丰富。据统计 95%的数据分析问题都能用 SQL 解决，这是一个相当惊人的结论。

本书介绍的实现方案能满足以上所有要求，涉及的具体技术包括：MySQL 主从复制，保证为业务系统提供可靠的数据库服务，并提供数据来源；Canal Server 实时获取增量 MySQL binlog，并将其传入 Kafka 消息队列；Kafka 将消息持久化，同时提供可伸缩、高吞吐的消息服务；Canal ClientAdapter 负责消费 Kafka 中的消息，将数据流传输到 Greenplum 数据库；Greenplum 作为数据仓库系统，提供实时 ETL 功能，自动维护操作数据存储（ODS）、维度表与事实表。

Greenplum 分布式数据库采用无共享（Shared-Nothing）的大规模并行处理（MPP）架构，能充分利用集群的硬件资源，将并行处理发挥到极致。Greenplum 具有可选存储模式、事务支持、并行查询与数据装载、容错与故障转移、数据库统计、过程化语言扩展等方面的功能特性，正是它们支撑 Greenplum 成为一款理想的分析型数据库产品。

本书内容

全书共分 10 章。第 1 章说明数据仓库相关的基本概念，包括数据仓库定义、操作型系统与分析型系统、ETL、数据仓库架构等。第 2 章介绍三种主流的数据仓库设计模型，即关系数据模型、维度数据模型和 DATA VAULT 模型。第 3 章介绍 Greenplum 系统架构、功能特性、主要优缺点，以及为何适用于数据仓库应用。第 4 章详解 Greenplum 的安装部署问题。第 5

章介绍实时数据同步的实现，包括 MySQL 数据复制在实时数据仓库架构中所起的作用，如何使用 Kafaka，以及 Maxwell + Kafka + Bireme 和 Canal Server + Kafka + Canal ClientAdapter 两种具体实现。第 6 章用一个销售订单示例说明如何使用 Greenplum 的规则（rule）实现实时自动数据装载。第 7 章和第 8 章分别详解多维数据仓库中常见的维度表和事实表技术，及其在 Greenplum 中的实现。第 9 章介绍 Greenplum 主要的、例行的与推荐的运维与监控工作。第 10 章作为完整数据分析体系的组成部分，介绍如何在 Greenplum 中集成 MADlib，实现基于 SQL 的机器学习。

读者对象

本书所定位的读者是大数据分析系统设计和开发、数据仓库系统设计和开发、DBA、架构师等相关技术人员。所有的描绘场景与实验环境都基于 Linux 操作系统。假设读者已具有一定的数据库、数据仓库、SQL 与 Linux 基础。

源码下载

本书配套的源码，需要使用微信扫描下面二维码获取，可按扫描后的页面提示，把下载链接转发到自己的邮箱中下载。如果发现问题或疑问，请用电子邮件联系 booksaga@163.com，邮件主题为“Greenplum 构建实时数据仓库实践”。

致谢

在本书编写过程中，得到了很多人的帮助与支持。首先，感谢我所在的公司——优贝在线提供的平台和环境，感谢同事们在工作中的鼎力相助。没有那里的环境和团队，也就不会有这本书。其次，感谢清华大学出版社图格事业部的老师和编辑们，他们的辛勤工作使得本书得以尽早与读者见面。再次，感谢 CSDN 提供的技术分享平台，给我有一个将博客文章整理成书的机会。最后，感谢家人对我一如既往的支持。

由于水平有限，书中疏漏之处在所难免，希望读者批评指正。

编　者

2022 年 5 月

目　录

第1章

数据仓库简介

对于每一种技术，都先要理解相关的概念和了解它之所以出现的原因，这对于我们继续深入学习其技术细节大有裨益。实时数据仓库首先是数据仓库，只是它优先考虑数据的时效性问题。因此，本章将介绍业界公认的数据仓库的定义，它和操作型数据库应用的区别，以及为什么我们需要数据仓库。

在对数据仓库的概念有了基本的认识后，有必要单独说明一下 ETL（Extralt-Transform-load，用来描述将数据从来源端经过抽取、转换、装载至目的端的过程）这个创建数据仓库过程中最重要的概念，然后向读者介绍四种常见的数据仓库架构。本章最后描述实时数据仓库的产生背景、特定需求和使用场景，并列举一些常见的实时数据仓库技术架构。

1.1 什么是数据仓库

数据仓库的概念可以追溯到 20 世纪 80 年代，当时 IBM 的研究人员开发出了“商业数据仓库”。本质上，数据仓库试图提供一种从操作型系统到决策支持环境的数据流架构模型。数据仓库概念的提出，是为了解决与这个数据流相关的各种问题，主要是解决多重数据复制带来的高成本问题。在没有数据仓库的时代，需要大量的冗余数据来支撑多个决策支持环境。在大组织里，多个决策支持环境独立运作是典型的情况。尽管每个环境服务于不同的用户，但这些环境经常需要大量相同的数据。处理过程包括收集、清洗、整合来自多个数据源的数据，并为每个决策支持环境做部分数据复制。数据源通常是早已存在的操作型系统，很多是遗留系统。此外，当一个新的决策支持环境形成时，操作型系统的数据经常被再次复用。用户访问这些处理后的数据。

1.1.1 数据仓库的定义

数据仓库之父 Bill Inmon 在 1991 年出版的 *Building the Data Warehouse* 一书中首次提出了被广为认可的数据仓库定义。Inmon 将数据仓库描述为一个面向主题的、集成的、随时间变化的、非易失的数据集合，用于支持管理者的决策过程。这个定义有些复杂并且难以理解，下面我们将它分解开来进行说明。

1. 面向主题

传统操作型系统是围绕公司的功能性应用进行组织的，而数据仓库是面向主题的。主题是一个抽象概念，简单说就是与业务相关的数据的类别，每一个主题基本对应一个宏观的分析领域。数据仓库被设计成辅助人们分析数据。例如，一个公司要分析销售数据，就可以建立一个专注于销售的数据仓库，使用这个数据仓库，就可以回答类似于“去年谁是我们这款产品的最佳用户”这样的问题。这个场景下的“销售”就是一个数据主题，而这种通过划分主题定义数据仓库的能力，就使得数据仓库是面向主题的。主题域是对某个主题进行分析后确定的主题的边界，如客户、销售、产品都是主题域的例子。

2. 集成

集成的概念与面向主题密切相关。还用销售的例子，假设公司有多条产品线和多种产品销售渠道，而每个产品线都有自己独立的销售数据库。此时，要想从公司层面整体分析销售数据，必须将多个分散的数据源统一成一致的、无歧义的数据格式，然后再放置到数据仓库中。因此数据仓库必须能够解决诸如产品命名冲突、计量单位不一致等问题。当完成了这些数据整合工作后，该数据仓库就可称为是集成的。

3. 随时间变化

为了发现业务变化的趋势、存在的问题，或者新的机会，需要分析大量历史数据。这与联机事务处理（OLTP）系统形成鲜明对比。联机事务处理反应的是当前时间点的数据情况，要求高性能、高并发和极短的响应时间。出于这样的需求考虑，联机事务处理系统中一般都将数据依照活跃程度分级，把历史数据迁移到归档数据库中。而数据仓库关注的是数据随时间变化的情况，并且能反映在过去某个时间点的数据是怎样的。换句话说，数据仓库中的数据是反映了某一历史时间点的数据快照，这也就是术语“随时间变化”的含义。当然，任何一个存储结构都不可能无限扩展，数据也不可能只入不出地永久驻留在数据仓库中，它在数据仓库中也有自己的生命周期。到了一定时候，数据会从数据仓库中移除，移除的方式可能是将细节数据汇总后删除、将老的数据转储到大容量介质后删除或直接物理删除等。

4. 非易失

非易失指的是，一旦进入到数据仓库中，数据就不应该再有改变。操作型环境中的数据一般都会频繁更新，而在数据仓库环境中一般并不进行数据更新。当改变的操作型数据进入数据仓库时会产生新的记录，这样就保留了数据变化的历史轨迹。也就是说，数据仓库中的数据基本是静态的。这是一个不难理解的逻辑概念。数据仓库的目的就是要根据曾经发生的事件进行分析，如果数据是可修改的，将使历史分析变得没有意义。

除了以上四个特性外，数据仓库还有一个非常重要的概念，就是粒度。粒度问题遍布于数据仓库体系结构的各个部分。粒度是指数据的细节或汇总程度，细节程度越高，粒度级别越低。例如，单个事务是低粒度级别，而一个月全部事务的汇总就是高粒度级别。

数据粒度一直是数据仓库设计需要重点思考的问题。在早期的操作型系统中，当细节数据被更新时，几乎总是将其存放在最低粒度级别上；而在数据仓库环境中，通常不这样做。例如，如果数据被装载进数据仓库的频率是每天一次，那么一天之内的数据更新将被忽略。

粒度之所以是数据仓库环境的关键设计问题，是因为它极大地影响了数据仓库的数据量和可以进行的查询类型。粒度级别越低，数据量越大，查询的细节程度就越高，查询范围就越广泛，反之亦然。

大多数情况下，数据会以很低的粒度级别进入数据仓库，如日志类型的数据或点击流数据，此时应该对数据进行编辑、过滤和汇总，使其适应数据仓库环境的粒度级别。如果得到的数据粒度级别比数据仓库的高，那将意味着在数据存入数据仓库前，开发人员必须花费大量的设计和资源来对数据进行拆分。

1.1.2　建立数据仓库的原因

现在读者应该已经熟悉了数据仓库的概念，那么数据仓库里的数据从哪里来呢？通常数据仓库的数据来自各个业务应用系统。业务系统中的数据形式多种多样，可能是 Oracle、MySQL、SQL Server 等关系数据库里的结构化数据，可能是文本、CSV 等平面文件或 Word、Excel 文档中的非结构化数据，还可能是 HTML、XML 等自描述的半结构化数据。这些业务数据经过一系列的数据抽取、转换、清洗，最终以一种统一的格式装载进数据仓库。数据仓库里的数据作为分析用的数据源，提供给后面的即席查询（Ad-Hoc Query）、分析系统、数据集市、报表系统、数据挖掘系统等。

从存储的角度看，数据仓库里的数据实际上已经存在于业务应用系统中，那么为什么不能直接操作业务系统中的数据用于分析，而要使用数据仓库呢？实际上在数据仓库技术出现前，有很多数据分析的先驱者已经发现，简单的“直接访问”方式很难良好工作，这样做的失败案例数不胜数。下面列举一些直接访问业务系统无法工作的原因：

- 某些业务数据由于安全或其他因素不能直接访问。
- 业务系统的版本变更很频繁，每次变更都需要重写分析系统并重新测试。
- 很难建立和维护汇总数据来源于多个业务系统版本的报表。
- 业务系统的列名通常是硬编码，有时仅仅是无意义的字符串，这让编写分析系统更加困难。
- 业务系统的数据格式，如日期、数字的格式不统一。
- 业务系统的表结构为事务处理性能而优化，有时并不适合查询与分析。
- 没有适当的方式将有价值的数据合并进特定应用的数据库。
- 没有适当的位置存储元数据。
- 用户需要看到的显示数据字段，有时在数据库中并不存在。
- 通常事务处理的优先级比分析系统高，所以如果分析系统和事务处理运行在同一硬件之

上，分析系统往往性能很差。

- 有误用业务数据的风险。
- 极有可能影响业务系统的性能。

尽管需要增加软硬件的投入，但建立独立数据仓库与直接访问业务数据相比，无论是成本还是带来的好处，这样做都是值得的。随着处理器和存储成本的逐年降低，数据仓库方案的优势更加明显，在经济上也更具可行性。

无论是建立数据仓库还是实施别的项目，都要从时间、成本、功能等几个角度来权衡比较，认真研究一下是否真正需要建立一个数据仓库。这是一个很好的问题，当我们的组织很小、人数很少、业务单一、数据量也不大时，可能真的不需要建立数据仓库。毕竟要想成功建立一个数据仓库并使其发挥应有的作用还是很有难度的，需要大量的人、财、物力，并且即便花费很大的代价完成了数据仓库的建设，在较短一段时间内也不易显现出价值。在没有专家介入而仅凭组织自身力量建立数据仓库时，还要冒相当大的失败风险。但是，当我们所在的组织有超过 1000 名员工，有几十个部门的时候，它所面临的挑战将是完全不同的。在这个充满竞争的时代，做出正确的决策对一个组织至关重要。而要做出最恰当的决策，仅依据对孤立维度的分析是不可能实现的。这时必须考虑所有相关数据的可用性，而这个数据最好的来源就是一个设计良好的数据仓库。

假设一个超市连锁企业，在没有实现数据仓库的情况下，该企业会发现，想要分析商品销售情况是非常困难的。比如，哪些商品被售出，哪些没有被售出，什么时间销量上升，哪个年龄组的客户倾向于购买哪些特定商品，等等，这些问题都无从回答，而给出这些问题的正确答案正是一个具有吸引力的挑战。这只是第一步，必须搞清楚一个特定商品到底适不适合 18~25 岁的人群，以决定该商品的销售策略。一旦从数据分析得出的结论是销售该商品的价值在降低，那么必须实施后面的步骤以分析在哪里出了问题，并采取相应的措施加以改进。

在辅助战略决策层面，数据仓库的重要性更加凸显。作为一个企业的经营者或管理者，他必须对某些问题给出答案，以获得超越竞争对手的额外优势。回答这些问题对于基本的业务运营可能不是必须的，但对于企业的生存发展却必不可少。下面是一些常见问题的例子：

- 如何把公司的市场份额提升 5%?
- 哪些产品的市场表现不令人满意?
- 哪些代理商需要销售政策的帮助?
- 提供给客户的服务质量如何? 哪些需要改进?

回答这些战略性问题的关键一环就是数据仓库。就拿“提供给客户的服务质量如何？”这一问题来说，这是管理者最为关心的问题之一。我们可以把这一问题分解成许多具体的小问题，比如第一个问题是：在过去半年中，收到过多少用户反馈？可以在数据仓库上发出对应的查询，并对查询结果进行分析。之所以能够这样做，是因为数据仓库中含有每一条用户反馈信息。

你可能已经想到了，第二个问题自然就是：在这些用户反馈当中，给出“非常满意”“一般”“不满意”的人数分别有多少？第三个问题就是：客户所强调的需要改进的地方和广受批评的地方是哪些？这在数据仓库的用户反馈信息中也有一列来表示，它也能从一个侧面反映出客户关心的问题是哪些。以上三个问题的答案联合在一起，就可以得出客户服务满意度的结论，

并且准确定位哪些地方急需改进。

下面简单总结一下使用数据仓库的好处：

- 将多个数据源集成到单一数据存储，因此可以使用单一数据查询引擎展示数据。
- 缓解在事务处理数据库上因执行大查询而产生的资源竞争问题。
- 维护历史数据。
- 通过对多个源系统的数据整合，使得在整个企业的角度存在统一的中心视图。
- 通过提供一致的编码和描述，减少或修正坏数据问题，提高数据质量。
- 一致性地表示组织信息。
- 提供所有数据的单一通用数据模型，而不用关心数据源。
- 重构数据，使数据对业务用户更有意义。
- 向复杂分析查询交付优秀的查询性能，同时不影响操作型系统。
- 开发决策型查询更简单。

1.2　操作型系统与分析型系统

上一节已经多次提及操作型系统和分析型系统，本节将详细阐述它们的概念及差异。在一个大型组织中，往往都有两种类型的系统——操作型和分析型，而这两种系统大都以数据库作为数据管理、组织和操作的工具。操作型系统完成组织的核心业务，例如下订单、更新库存、记录支付信息等。这些系统是事务型的，核心目标是尽可能快地处理事务，同时维护数据的一致性和完整性。而分析型系统的主要作用是通过数据分析评估组织的业务经营状况，并进一步辅助决策。

1.2.1　操作型系统

相信从事过 IT 或相关工作的读者对操作型系统都不会感到陌生，几乎所有的互联网线上系统、MIS、OA 等都属于这类系统的应用。操作型系统是一类专门用于管理面向事务的应用的信息系统。“事务”一词在这里存在一些歧义，有些人理解事务是一个计算机或数据库的术语，另一些人所理解的事务是指业务或商业交易，这里使用前一种语义。那么什么是数据库技术中的事务呢？这是首先需要明确的概念。

事务是工作于数据库管理系统（或类似系统）中的一个逻辑单元，该逻辑单元中的操作被以一种独立于其他事务的可靠方式所处理。事务一般代表着数据改变，它提供“all-or-nothing”操作，就是说事务中的一系列操作要么完全执行，要么完全不执行。在数据库中使用事务主要出于两个目的：

- 保证工作单元的可靠性。当数据库系统异常宕机时，其中执行的操作或者已经完成或者只有部分完成，很多没有完成的操作此时处于一种模糊状态。在这种情况下，数据库系统必须能够恢复到数据一致的正常状态。

- 提供并发访问数据库的多个程序间的隔离。如果没有这种隔离，程序得到的结果很可能是错误的。

根据事务的定义，引申出事务具有原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Durability）的特点，也就是数据库领域中常说的事务的 ACID 特性。

- 原子性

数据库系统的原子性指的是事务中的一系列操作或全执行或不执行，这些操作是不可再分的。原子性可以防止数据被部分修改。银行账号间转账是一个事务原子性的例子。简单地说，从 A 账号向 B 账号转账有两步操作：A 账号提取，B 账号存入。这两个操作以原子性事务执行，使数据库保持一致的状态，即使这两个操作的任何一步失败了，总的金额数不会减少也不会增加。

- 一致性

数据库系统中的一致性是指任何数据库事务只能以允许的方式修改数据。任何数据库写操作必须遵循既有的规则，包括约束、级联、触发器以及它们的任意组合。一致性并不保证应用程序逻辑的正确性，但它能够保证不会因为程序错误而使数据库产生违反规则的行为。

- 隔离性

在数据库系统中，隔离性决定了其他用户所能看到的事务完整性程度。例如，一个用户正在生成一个采购订单，并且已经生成了订单主记录，但还没有生成订单条目明细记录。此时订单主记录能否被其他并发用户看到呢？这就由隔离级别决定。在数据库系统中，按照由低到高一般有读非提交、读提交、可重复读、串行化等几种隔离级别。数据库系统并不一定实现所有的隔离级别，如 Oracle 数据库只实现了读提交和串行化，而 MySQL、PostgreSQL 数据库则提供全部四种隔离级别。

隔离级别越低，多用户并发访问数据的能力越高，但同时也会增加脏读、丢失更新等并发操作的负面影响。相反，高隔离级降低了并发影响，但需要使用更多的系统资源，也增加了事务被阻塞的可能性。

- 持久性

数据库系统的持久性保证已经提交的事务是永久保存的。例如，如果一个机票预订报告显示一个座位已经订出，那么即使系统崩溃，被订出的座位也会一直保持被订出的状态。持久性可以通过在事务提交时将事务日志刷新至永久性存储介质来实现。

了解了事务的基本概念后，我们再来看操作型系统就比较容易理解了。操作型系统通常是高并发、低延迟的系统，具有大量检索、插入、更新操作，事务数量大，但每个事务影响的数据量相对较小。这样的系统很适合在线应用，这些应用有成千上万用户在同时使用，并要求能够立即响应用户请求。操作型系统常被整合到面向服务的架构（SOA）和 Web 服务里。对操作型系统应用的主要要求是高可用、高速度、高并发、可恢复和保证数据一致性，在各种互联网应用层出不穷的今天，这些系统要求是显而易见的。

1. 操作型系统的数据库操作

在数据库使用上，操作型系统常用的操作是增、改、查，并且通常是插入与更新密集型的，同时会对数据库进行大量并发查询，而删除操作相对较少。操作型系统一般都直接在数据库上修改数据，没有中间过渡区。

2. 操作型系统的数据库设计

操作型系统的特征是大量短的事务，并强调快速处理查询。每秒事务数（TPS）是操作型系统的一个有效度量指标。针对以上这些特点，数据库设计一定要满足系统的要求。

在数据库逻辑设计上，操作型系统的应用数据库大都使用规范化设计方法，通常要满足第三范式。这是因为规范化设计能最大限度地减少数据冗余，因而提供更快更高效的方式执行数据库写操作。关于规范化设计概念及其相关内容，将会在第 2 章“数据仓库设计基础”中作详细说明。

在数据库物理设计上，应该依据系统所使用的数据库管理系统的具体特点，做出相应的设计，毕竟每种数据库管理系统在实现细节上存在很大差异。下面就以 Oracle 数据库为例，简要说明在设计操作型系统数据库时应该考虑的问题。

- 调整回滚段。回滚段是数据库的一部分，其中记录着最终被回滚的事务的行为。这些回滚段信息可以提供读一致性、回滚事务和数据库恢复。
- 合理使用聚簇。聚簇是一种数据库模式，其中包含共用一列或多列的多个表。数据库中的聚簇表用于提高连接操作的性能。
- 适当调整数据块大小。数据块大小应该是操作系统块大小的倍数，并且设置上限以避免不必要的 I/O。
- 设置缓冲区高速缓存大小。合理的缓存大小能够有效避免不必要的磁盘 I/O。
- 动态分配表空间。
- 合理划分数据库分区。分区最大的作用在于可用性和可维护性，使得数据维护期间保持事务处理的性能。
- SQL 优化。有效利用数据库管理系统的优化器，使用最佳的数据访问路径。
- 避免过度使用索引。大量的数据修改会给索引维护带来压力，从而对整个系统的性能产生负面影响。

以上所讲的操作型系统都是以数据库系统为核心，而数据库系统为了保持 ACID 特性，本质上是单一集中式系统。在当今这个信息爆炸的时代，集中式数据库往往已无法支撑业务的需要（从某订票网站和某电商网站的超大瞬时并发量来看，这已是一个不争的事实）。这就给操作型系统带来新的挑战。分布式事务、去中心化、CAP 与最终一致性等一系列新的理论和技术，为解决系统扩展问题应运而生。这是一个很大的话题，要想说清楚需要很多的扩展知识和大量篇幅，故这里只是点到为止，不做展开。

1.2.2　分析型系统

在计算机领域，分析型系统是一种快速回答多维分析查询的实现方式。它也是更广泛范

畴的所谓商业智能的一部分（商业智能还包含数据库、报表系统、数据挖掘、数据可视化等研究方向）。分析型系统的典型应用包括销售业务分析报告、市场管理报告、业务过程管理（BPM）、预算和预测、金融分析报告及其类似的应用。

1. 分析型系统的数据库操作

在数据库层面，分析型系统操作被定义成少量的事务、复杂的查询、处理归档和历史数据。这些数据很少被修改，从数据库抽取数据是最多的操作，也是识别这种系统的关键特征。分析型数据库基本上都是读操作。

2. 分析型系统的数据库设计

分析型系统的特征是相对少量的事务，但查询通常非常复杂并且会包含聚合计算，例如今年和去年同时期的数据对比、百分比变化趋势等。分析型数据库中的数据一般来自于一个企业级数据仓库，是整合过的历史数据。对于分析型系统，吞吐量是一个有效的性能度量指标。

在数据库逻辑设计上，分析型数据库使用多维数据模型，通常被设计成星型模式或雪花模式。关于多维数据模型的概念及其相关内容，会在第 2 章“数据仓库设计基础”中作详细说明。

在数据库物理设计上，依然以 Oracle 数据库为例，简要说明在设计分析型系统数据库时应该考虑的一些问题。

- 表分区。可以独立定义表分区的物理存储属性，将不同分区的数据存放到多个物理文件上。这样做一方面可以分散 I/O；另一方面，当数据量非常大时，方便维护数据；再有就是利用分区消除查询数据时，不用扫描整张表，从而提高查询性能。
- 位图索引。当查询条件中包含低基数（不同值很少，例如性别）的列，尤其是包含这些列上的 or、and 或 not 这样的逻辑运算时，或者从有大量行的表中返回大量的行时，应考虑位图索引。
- 物化视图。物化视图物理存储查询所定义的数据，能够自动增量刷新数据，并且可以利用查询重写特性极大地提高查询速度，是分析型系统常用的技术。
- 并行化操作。可以在查询大量数据时执行并行化操作，这样可使多个服务器进程为同一个查询语句工作，使该查询得以快速完成，但是会耗费更多的资源。

随着数据的大量积累和大数据时代的到来，人们对于数据分析的依赖越来越强，因此分析型系统也随之越来越显示出其重要性。举一个简单的例子，在一家医院中，保存有 20 年的非常完整的病人信息。医院领导想看到最常见的疾病、成功治愈率、实习医生的实习天数等相关数据的详细报告。为了满足这个需求，应用分析型系统查询医院信息数据仓库，并通过复杂查询得到结果，然后将报告提交给领导做进一步分析。

1.2.3 操作型系统和分析型系统的对比

操作型系统和分析型系统是两种不同种类的信息系统。它们都与数据库技术相关，数据库提供方法支持这两种系统的功能。操作型系统和分析型系统以完全不同的方式使用数据库，不仅如此，分析型系统更加注重数据分析和报表，而操作型系统的目标是一个伴有大量数据改

变的事务优化系统。

对于学习数据科学及其相关技术的读者，了解这两种信息处理方式的区别至关重要。这也是理解商业智能、数据挖掘、数据仓库、数据模型、ETL 处理和大数据等系统的基础。

通过前面对两种系统的描述，我们可以对比它们的很多方面。表 1-1 总结了两种系统的主要区别。后面我们进一步讨论每一个容易产生疑惑的对比项，以帮助读者理解。

表 1-1　操作型系统和分析型系统对比表

对比项	操作型系统	分析型系统
数据源	应用的操作信息，一般是最原始的数据	历史的、归档的数据，一般来源于数据仓库
侧重点	数据更新	信息的检索或报表
应用	管理系统、交易系统、在线应用等	报表系统、多维分析、决策支持系统等
用户	终端用户、普通雇员	管理人员、市场人员、数据分析师
任务	业务操作	数据分析
数据更新	插入、更新、删除数据，要求快速执行，立即返回结果	大量数据装载，花费时间很长
数据模型	实体关系模型	多维数据模型
设计方法	规范化设计，大量的表和表之间的关系	星型模式或雪花模式，少量的表
备份	定期执行全量或增量备份，不允许数据丢失	简单备份，数据可以重新装载
数据的时间范围	从天到年	几年或几十年
查询	简单查询，快速返回查询结果	复杂查询，执行聚合或汇总操作
速度	快，大表上需要建索引	相对较慢，需要更多的索引
所需空间	小，只存储操作数据	大，需要存储大量历史数据

首先，两种系统的侧重点不同。操作型系统更适合对已有数据进行更新，所以是日常处理工作或在线系统的选择。相反，分析型系统提供在大量存储数据上的分析能力，所以这类系统更适合报表类应用。分析型系统通常用于查询历史数据，这有助于得到更准确的分析报告。

其次，因为这两种系统的目标完全不同，所以为了得到更好的性能，使用的数据模型和设计方法也不同。操作型系统数据库通常使用规范化设计，为普通查询和数据修改提供更好的性能。分析型数据库则具有典型的数据仓库组织形式。

基于这两个主要的不同点，我们可以推导出两种系统其他方面的区别。操作型系统上的查询更小，而分析型系统上执行的查询要复杂得多。所以通常操作型系统会比分析型系统快很多。

操作型系统的数据会持续更新，并且更新会立即生效。而分析型系统的数据更新是由预定义的处理作业同时装载大量的数据集合，并且在装载前需要做数据转换，因此整个数据更新过程需要较长的执行时间。

由于操作型系统要做到绝对的数据安全和可用性，所以需要实施复杂的备份系统。基本的全量备份和增量备份都是必须做的。而分析型系统只需要偶尔执行数据备份即可，这一方面是因为此类系统一般不需要保持持续运行，另一方面则是因为数据还可以从操作型系统中重复装载。

两种系统的空间需求显然都依赖于它们所存储的数据量。分析型系统要存储大量的历史数据，因此需要更多的存储空间。

1.3 抽取—转换—装载

前面已经多次提到了 ETL 一词，它是 Extract、Transform、Load 三个英文单词首字母的简写，中文意为抽取、转换、装载。ETL 是建立数据仓库最重要的处理过程，也是最体现工作量的环节，一般会占到整个数据仓库项目工作量的一半以上。

- 抽取：从操作型数据库获取数据。
- 转换：转换数据，使之转变为适用于查询和分析的形式和结构。
- 装载：将转换后的数据导入到最终的目标数据仓库。

建立一个数据仓库，就是要把来自于多个异构的源系统的数据集成在一起，放置在一个集中的位置用于数据分析。如果一开始这些源系统数据就是兼容的，当然最好，但情况往往不是这样。ETL 系统的工作就是要把异构的数据转换成同构的。如果没有 ETL，就不能对异构的数据进行程序化的分析。

1.3.1 数据抽取

抽取操作是从源系统获取数据给后续的数据仓库环境使用。这是 ETL 处理的第一步，也是最重要的一步。数据被成功抽取后，才可以进行转换并装载到数据仓库中。能否正确地获取数据直接关系到后面步骤的成败。数据仓库典型的源系统是事务处理应用，例如，一个销售分析数据仓库的源系统之一可能是一个订单录入系统，其中包含当前销售订单相关操作的全部记录。

设计和建立数据抽取过程，在 ETL 处理乃至整个数据仓库处理过程中，一般是较为耗时的任务。源系统很可能非常复杂并且缺少相应的文档，因此只是决定需要抽取哪些数据可能就已经非常困难了。通常数据都不是只抽取一次，而是需要以一定的时间间隔反复抽取，通过这样的方式把数据的所有变化提供给数据仓库，并保持数据的及时性。除此之外，源系统一般不允许外部系统对它进行修改，也不允许外部系统对它的性能和可用性产生影响，数据仓库的抽取过程要能适应这样的需求。如果已经明确了需要抽取的数据，下一步就该考虑从源系统抽取数据的方法了。

对抽取方法的选择高度依赖于源系统和目标数据仓库环境的业务需要。一般情况下，不能因为需要提升数据抽取的性能，而在源系统中添加额外的逻辑，也不能增加这些源系统的工作负载。有时，甚至都不允许用户增加任何“开箱即用”的外部应用系统，这叫作对源系统具有侵入性。下面分别从逻辑和物理两方面介绍数据抽取方法。

1. 逻辑抽取

有两种逻辑抽取类型：全量抽取和增量抽取。

（1）全量抽取

源系统的数据全部被抽取。因为这种抽取类型影响源系统上当前所有有效的数据，所以不需要跟踪自上次成功抽取以来的数据变化。源系统只需要原样提供现有的数据，而不需要附

加的逻辑信息（比如时间戳等）。一个全表导出的数据文件或者一个查询源表所有数据的 SQL 语句，都是全量抽取的例子。

（2）增量抽取

只抽取某个事件发生的特定时间点之后的数据。通过该事件发生的时间顺序能够反映数据的历史变化，它可能是最后一次成功抽取，也可能是一个复杂的业务事件，如最后一次财务结算等。必须能够标识出特定时间点之后所有的数据变化。这些发生变化的数据可以由源系统自身来提供，例如能够反映数据最后发生变化的时间戳列，或者是一个原始事务处理之外的、只用于跟踪数据变化的变更日志表。大多数情况下，使用后者意味着需要在源系统上增加抽取逻辑。

在许多数据仓库中，抽取过程不含任何变化数据捕获技术。取而代之的是，把源系统中的整个表抽取到数据仓库过渡区，然后用这个表的数据和上次从源系统抽取得到的表数据作比对，从而找出发生变化的数据。虽然这种方法不会对源系统造成很大的影响，但显然需要考虑给数据仓库处理增加的负担，尤其是当数据量很大的时候。

2. 物理抽取

依赖于选择的逻辑抽取方法和能够对源系统所做的操作和所受的限制，存在两种物理数据抽取机制：直接从源系统联机抽取，或者间接从一个脱机结构抽取数据。这个脱机结构有可能已经存在，也可能需要由抽取程序生成。

（1）联机抽取

数据直接从源系统抽取。抽取进程或者直连源系统数据库访问它们的数据表，或者连接到一个存储快照日志或变更记录表的中间层系统。

注　意
中间层系统不必和源系统物理分离。

（2）脱机抽取

数据不从源系统直接抽取，而是从一个源系统以外的过渡区抽取。过渡区可能已经存在（例如，数据库备份文件、关系数据库系统的重做日志、归档日志等），或者抽取程序自己建立。应该考虑以下的存储结构：

- 数据库备份文件。一般需要数据还原操作才能使用。
- 备用数据库。如 Oracle 的 DataGuard 和 MySQL 的数据复制等技术。
- 平面文件。数据定义成普通格式，关于源对象的附加信息（列名、数据类型等）需要另外处理。
- 导出文件。关系数据库大都自带数据导出功能，如 Oracle 的 exp/expdp 程序和 MySQL 的 mysqldump 程序，都可以用于生成导出数据文件。
- 重做日志和归档日志。每种数据库系统都有自己的日志格式和解析工具。

3. 变化数据捕获

抽取处理需要重点考虑增量抽取，也被称为变化数据捕获，简称 CDC。假设一个数据仓库系统，在每天夜里的业务低峰时间从操作型源系统抽取数据，那么增量抽取只需要过去 24

小时内发生变化的数据。变化数据捕获也是建立实时数据仓库的关键技术。

当我们能够识别并获得最近发生变化的数据时，抽取及其后面的转换、装载操作显然都会变得更高效，因为要处理的数据量会小很多。遗憾的是，很多源系统很难识别出最近变化的数据，或者必须侵入源系统才能做到。变化数据捕获是数据抽取中典型的技术挑战。

常用的变化数据捕获方法有时间戳、快照、触发器和日志四种。相信熟悉数据库的读者对这些方法都不会陌生。时间戳方法需要源系统有相应的数据列表示最后的数据变化。快照方法可以使用数据库系统自带的机制实现，如 Oracle 的物化视图技术，也可以自己实现相关逻辑，但会比较复杂。触发器是关系数据库系统具有的特性，源表上建立的触发器会在对该表执行 insert、update、delete 等语句时被触发，触发器中的逻辑用于捕获数据的变化。日志可以使用应用日志或系统日志，这种方式对源系统不具有侵入性，但需要额外的日志解析工作。关于这四种方案的特点，将会在本书第 5 章中具体说明。

1.3.2 数据转换

数据从操作型源系统获取后，需要进行多种转换操作。如统一数据类型、处理拼写错误、消除数据歧义、解析为标准格式等。数据转换通常是最复杂的部分，也是 ETL 开发中用时最长的一步。数据转换的范围极广，包括从单纯的数据类型转化到极为复杂的数据清洗技术。

在数据转换阶段，为了最终能够将数据装载到数据仓库中，需要在已经抽取来的数据上应用一系列的规则和函数。有些数据可能不需要转换就能直接导入到数据仓库。

数据转换一个最重要的功能是清洗数据，目的是只有“合规”的数据才能进入目标数据仓库。这一步操作在不同系统间交互和通信时尤其必要，例如，一个系统的字符集在另一个系统中可能是无效的。此外，由于某些业务和技术的需要，也需要进行多种数据转换，例如下面的情况：

- 只装载特定的数据列。例如，某列为空的数据不装载。
- 统一数据编码。例如，性别字段，有些系统使用的是 1 和 0，有些是“M”和“F”，有些是“男”和“女”，统一成“M”和“F”。
- 自由值编码。例如，将“Male”改成“M”。
- 预计算。例如，产品单价×购买数量=金额。
- 基于某些规则重新排序以提高查询性能。
- 合并多个数据源的数据并去重。
- 预聚合。例如，汇总销售数据。
- 行列转置。
- 将一列转为多列。例如，某列存储的数据是以逗号作为分隔符的字符串，将其分割成多列的单个值。
- 合并重复列。
- 预连接。例如，查询多个关联表的数据。
- 数据验证。针对验证的结果采取不同的处理方式，通过验证的数据交给装载步骤，验证失败的数据或直接丢弃，或记录下来做进一步检查。

1.3.3　数据装载

ETL 的最后步骤是把转换后的数据装载进目标数据仓库。这一步操作需要重点考虑两个问题：一是数据装载的效率问题；二是一旦装载过程中途失败，如何再次重复执行装载过程。

即使经过了转换、过滤和清洗，去掉了部分噪声数据，但需要装载的数据量还是很大的。执行一次数据装载可能需要几个小时的时间，同时需要占用大量的系统资源。想要提高装载效率，加快装载速度，可以从以下几方面入手：首先保证足够的系统资源；数据仓库存储的都是海量数据，所以需要配置高性能的服务器，并且要独占资源，不要与别的系统共用；其次在进行数据装载时，要禁用数据库约束（唯一性、非空性，检查约束等）和索引，当装载过程完全结束后，再启用这些约束，重建索引，这种方法会大幅提高装载速度；最后在数据仓库环境中，一般不使用数据库来保证数据的参考完整性，即不使用数据库的外键约束，它应该由 ETL 工具或程序来维护。

数据装载过程可能由于多种原因而失败，比如装载过程中某些源表和目标表的结构不一致而导致失败，而这时已经有部分表装载成功了。在数据量很大的情况下，如何能在重新执行装载过程时只装载失败的部分，这是一个不小的挑战。对于这种情况，实现可重复装载的关键是要记录下失败点，并在装载程序中处理相关的逻辑。还有一种情况，就是装载成功后数据又发生了改变（比如，有些滞后的数据在 ETL 执行完才进入系统，就会带来数据的更新或新增），这时需要重新执行一遍装载过程，已经正确装载的数据可以被覆盖，但相同数据不能重复新增。简单的实现方式是先删除再插入，或者用 replace into、merge into 等类似功能的操作。

装载到数据仓库里的数据，经过汇总、聚合等处理后交付给多维立方体或数据可视化、仪表盘等报表工具、BI 工具做进一步的数据分析。

1.3.4　开发 ETL 系统的方法

ETL 系统一般会从多个应用系统整合数据，典型的情况是这些应用系统运行在不同的软硬件平台上，由不同的厂商支持，各个系统的开发团队也是彼此独立的，与之伴随的数据多样性增加了 ETL 系统的复杂性。

开发一个 ETL 系统，常用的方式是使用数据库标准的 SQL 及其程序化语言，如 Oracle 的 PL/SQL 和 MySQL 的存储过程、用户自定义函数（UDF）等。还可以使用 Kettle 这样的 ETL 工具，这些工具都提供多种数据库连接器和多种文件格式的处理能力，并且对 ETL 处理进行了优化。使用工具的最大好处是减少编程工作量，提高工作效率。如果遇到特殊需求或特别复杂的情况，可能还需要使用 Shell、Java、Python 等编程语言开发自己的应用程序。

ETL 过程要面对大量的数据，因此需要较长的处理时间。为了提高 ETL 的效率，通常这三步操作会并行执行。当数据被抽取时，转换进程同时处理已经收到的数据。一旦某些数据被转换过程处理完，装载进程就会将这些数据导入目标数据仓库，而不会等到前一步工作执行完才开始。

1.4 数据仓库架构

前面三节介绍了数据仓库、操作型系统、分析型系统、ETL 等概念，也指出了分析型系统的数据源一般来自数据仓库，而数据仓库的数据来自于操作型系统。本节将从技术角度讨论数据仓库的组成和架构。

1.4.1 基本架构

“架构”是什么？这个问题从来就没有一个准确的答案。在软件行业中，一种被普遍接受的架构定义是指系统的一个或多个结构。结构中包括软件的构建（构建是指软件的设计与实现），构建的外部可以看到属性以及它们之间的相互关系。这里参考此定义，把数据仓库架构理解成构成数据仓库的组件及其之间的关系，那么就有了如图 1-1 所示的数据仓库架构图。

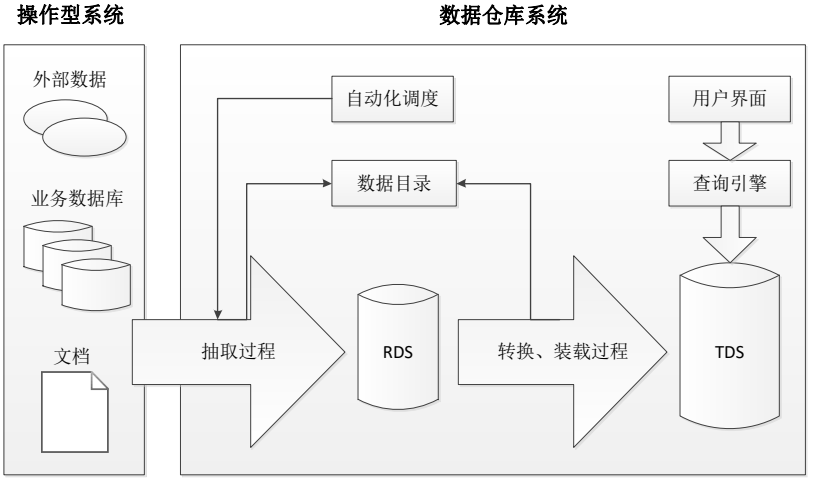

图 1-1 数据仓库架构

图 1-1 中显示的整个数据仓库环境，包括操作型系统和数据仓库系统两大部分。操作型系统的数据由各种形式的业务数据组成，这其中可能有关系数据库、TXT 或 CSV 文件、HTML 或 XML 文档，还可能存在外部系统的数据，比如网络爬虫抓取来的互联网数据等，数据可能是结构化、半结构化、非结构化的。这些数据经过抽取、转换和装载（ETL）过程进入数据仓库系统。

这里把 ETL 过程分成了抽取和转换、装载两个部分。抽取过程负责从操作型系统获取数据，该过程一般不做数据聚合和汇总，但是会按照主题进行集成，物理上是将操作型系统的数据全量或增量复制到数据仓库系统的 RDS（Raw Data Stores）中。转换、装载过程将对数据进行清洗、过滤、汇总、统一格式化等一系列转换操作，使数据转化为适合查询的格式，然后装载进数据仓库系统的 TDS（Transformed Data Stores）中。传统数据仓库的基本模式是首先用一些过程将操作型系统的数据抽取到文件，然后用另一些过程将这些文件转化成 MySQL 或 Oracle 这样的关系数据库的记录，最后用第三部分过程把数据导入进数据仓库。

RDS 是原始数据存储的意思。将原始数据保存到数据仓库里是个不错的想法。ETL 过程的 bug 或系统中的其他错误是不可避免的，保留原始数据使得追踪并修改这些错误成为可能。有时，数据仓库的用户会有查询细节数据的需求，这些细节数据的粒度与操作型系统的相同。有了 RDS，这种需求就很容易实现，用户可以查询 RDS 里的数据且不会影响业务系统的正常运行。这里的 RDS 实际上是起到了操作型数据存储（Operational Data Store，ODS）的作用，关于 ODS 相关内容将在 1.4.3 小节详细论述。

TDS 意为转换后的数据存储。这是真正的数据仓库中的数据。大量的用户会在经过转换的数据集上处理他们的日常查询。如果前面的工作做得好，这些数据将以保证最重要的和最频繁的查询能够快速执行的方式构建出来。

这里的原始数据存储和转换后的数据存储是逻辑概念，它们可能物理存储在一起，也可能分开。当原始数据存储和转换后的数据存储物理上分开时，它们不必使用同样的软硬件。传统数据仓库中，原始数据存储通常是本地文件系统，原始数据被组织进相应的目录中，这些目录是基于数据从哪里抽取或何时抽取建立（例如以日期作为文件或目录名称的一部分）；转换后的数据存储一般是某种关系数据库。

自动化调度组件的作用是自动定期重复执行 ETL 过程。不同角色的数据仓库用户对数据的更新频率要求也会有所不同，财务主管需要每月的营收汇总报告，而销售人员想看到每天的产品销售数据。作为通用的需求，所有数据仓库系统都应该能够建立周期性自动执行的工作流作业。传统数据仓库一般利用操作系统自带的调度功能（如 Linux 的 cron 或 Windows 的计划任务）来实现作业自动执行。

数据目录有时也被称为元数据存储，它可以提供一份数据仓库中数据的清单。用户通过它可以快速解决这些问题：什么类型的数据被存储在哪里？数据集的构建有何区别？数据最后的访问或更新时间，等等。此外，还可以通过数据目录感知数据是如何被操作和转换的。一个好的数据目录是让用户体验到系统易用性的关键。

查询引擎组件负责实际执行用户查询。传统数据仓库中，它可能是存储转换后数据的 Oracle、MySQL 等关系数据库系统内置的查询引擎，还可能是以固定时间间隔向其导入数据的 OLAP 立方体，如 Essbase cube。

用户界面指的是最终用户所使用的接口程序。可能是一个 GUI 软件，如 BI 套件中的客户端软件，也可能就是一个浏览器。

1.4.2　主要数据仓库架构

在数据仓库技术演化过程中，产生了几种主要的架构方法，包括数据集市架构、Inmon 企业信息工厂架构、Kimball 数据仓库架构和混合型数据仓库架构。

1. 数据集市架构

数据集市是按主题域组织的数据集合，用于支持部门级的决策。有两种类型的数据集市：独立数据集市和从属数据集市。

独立数据集市集中于部门所关心的单一主题域，数据以部门为基础部署，无须考虑企业级别的信息共享与集成。例如，制造部门、人力资源部门和其他部门都各自有他们自己的数据集市。独立数据集市从一个主题域或一个部门的多个事务系统获取数据，用以支持特定部门的

业务分析需要。一个独立数据集市的设计既可以使用实体关系模型，也可以使用多维模型。数据分析或商业智能工具直接从数据集市查询数据，并将查询结果显示给用户。一个典型的独立数据集市架构如图 1-2 所示。

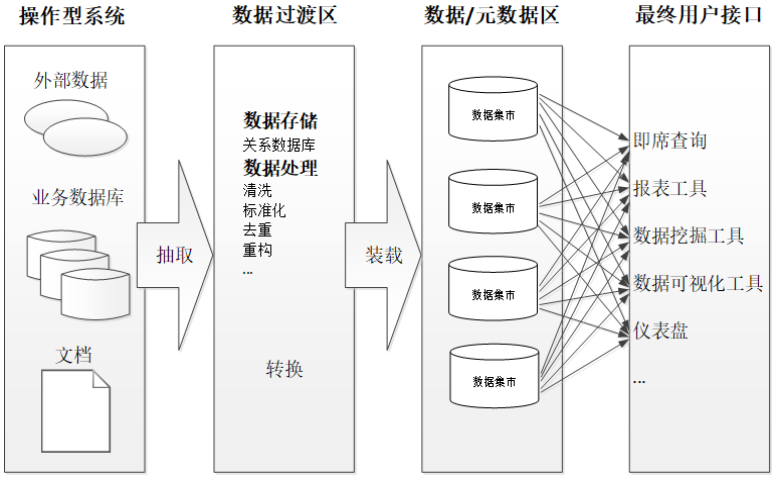

图 1-2　独立数据集市架构

因为一个部门的业务相对于整个企业来说要简单得多，数据量也小得多，所以建立部门的独立数据集市具有周期短、见效快的特点。如果从企业整体的视角来观察这些数据集市，我们会看到每个部门使用不同的技术，建立不同的 ETL 过程，处理不同的事务系统，而在多个独立的数据集市之间还会存在数据的交叉与重叠，甚至会有数据不一致的情况。从业务角度看，当部门的分析需求扩展，或者需要分析跨部门或跨主题域的数据时，独立数据集市会显得力不从心。而当数据存在歧义，比如同一个产品，在 A 部门和 B 部门的定义不同时，将无法在部门间进行信息比较。

另外一种数据集市是从属数据集市。如 Bill Inmon 所说，从属数据集市的数据来源于数据仓库。数据仓库里的数据经过整合、重构、汇总后传递给从属数据集市。从属数据集市的架构如图 1-3 所示。

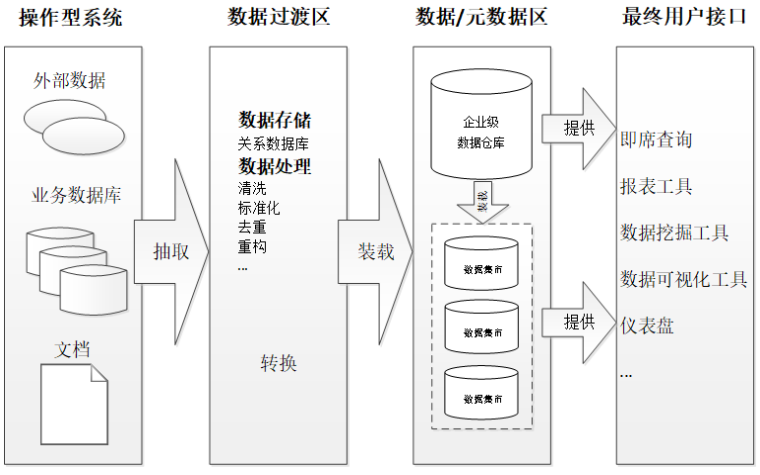

图 1-3　从属数据集市架构

建立从属数据集市的好处主要有：

- 性能：当数据仓库的查询性能出现问题，可以考虑建立几个从属数据集市，将查询从数据仓库移出到数据集市。
- 安全：每个部门可以完全控制自己的数据。
- 数据一致：因为每个数据集市的数据来源都是同一个数据仓库，有效消除了数据不一致的情况。

2. Inmon 企业信息工厂架构

Inmon 企业信息工厂架构如图 1-4 所示。

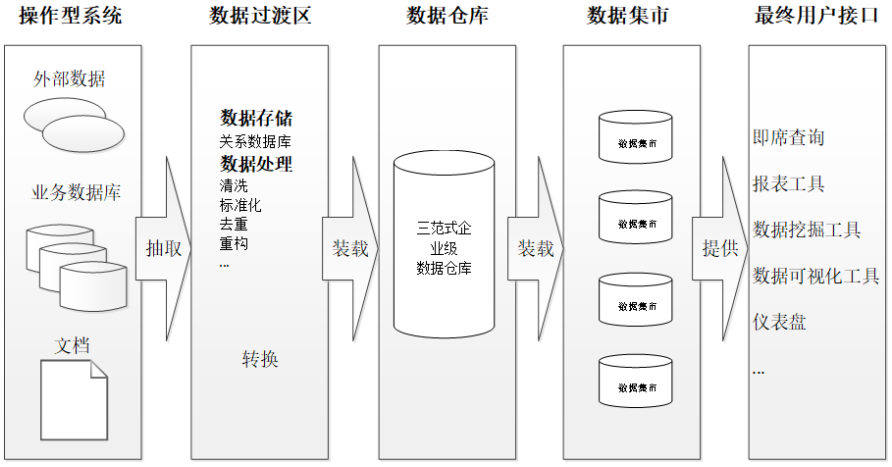

图 1-4　Inmon 企业信息工厂架构

我们来看图 1-4 中的组件是如何协同工作的。

- 应用系统：这些应用是组织中的操作型系统，用来支撑业务。它们收集业务处理过程中产生的销售、市场、材料、物流等数据，并将数据以多种形式进行存储。操作型系统也叫源系统，为数据仓库提供数据。
- ETL 过程：ETL 过程从操作型系统抽取数据，然后将数据转换成一种标准形式，最终将转换后的数据装载到企业级数据仓库中。ETL 是周期性运行的批处理过程。
- 企业级数据仓库：是该架构中的核心组件。正如 Inmon 数据仓库所定义的，企业级数据仓库是一个细节数据的集成资源库，其中的数据以最低粒度级别被捕获，存储在满足三范式设计的关系数据库中。
- 部门级数据集市：是面向主题数据的部门级视图，数据从企业级数据仓库获取。数据在进入部门数据集市时可能进行聚合。数据集市使用多维模型设计，用于数据分析。重要的一点是，所有的报表工具、BI 工具或其他数据分析应用，都从数据集市查询数据，而不是直接查询企业级数据仓库。

3. Kimball 数据仓库架构

Kimball 数据仓库架构如图 1-5 所示。

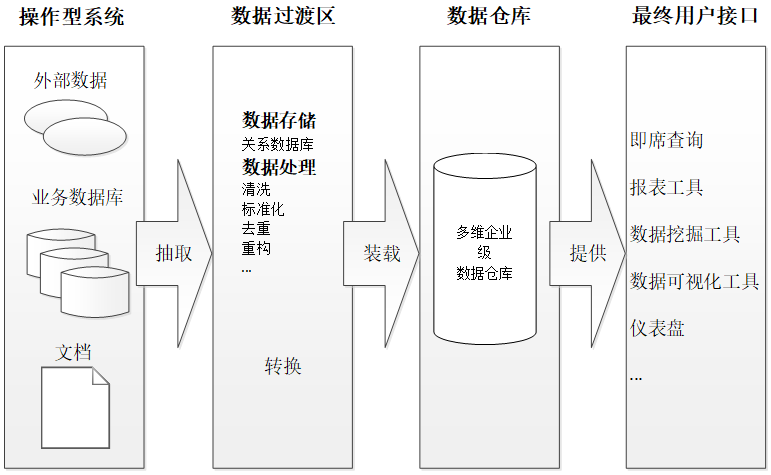

图 1-5　Kimball 数据仓库架构

对比图 1-4 可以看到，Kimball 与 Inmon 两种架构的主要区别在于核心数据仓库的设计和建立。Kimball 的数据仓库包含高粒度的企业数据，使用多维模型设计，这也意味着数据仓库由星型模式的维度表和事实表构成。分析系统或报表工具可以直接访问多维数据仓库里的数据。在此架构中的数据集市也与 Inmon 中的不同。这里的数据集市是一个逻辑概念，只是多维数据仓库中的主题域划分，并没有自己的物理存储，也可以说是虚拟的数据集市。

4. 混合型数据仓库架构

混合型数据仓库架构如图 1-6 所示。

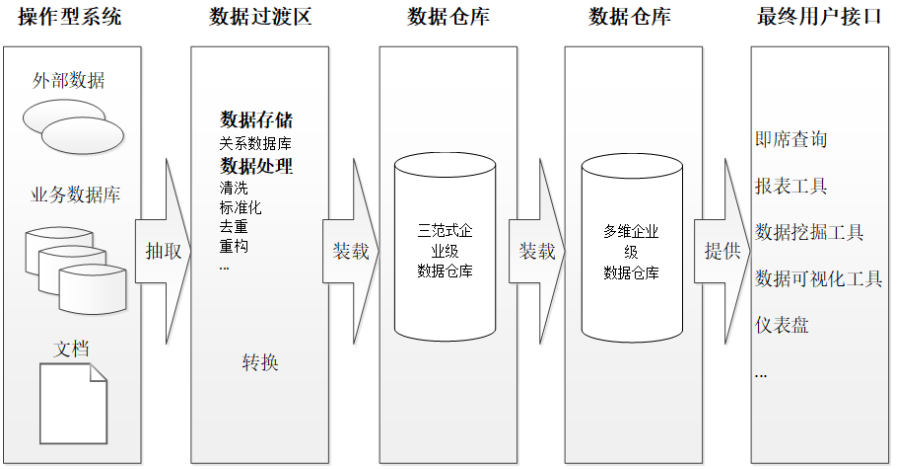

图 1-6　混合型数据仓库架构

所谓的混合型结构，指的是在一个数据仓库环境中，联合使用 Inmon 和 Kimball 两种架构。从架构图可以看到，这种架构将 Inmon 方法中的数据集市部分替换成了一个多维数据仓库，而数据集市则是多维数据仓库上的逻辑视图。使用这种架构的好处是，既可以利用规范化设计消除数据冗余，保证数据的粒度足够细，又可以利用多维结构更灵活地在企业级数据仓库实现报表和分析。

1.4.3　操作型数据存储

操作型数据存储又称为 ODS，是 Operational Data Store 的简写，其定义是这样的：一个面向主题的、集成的、可变的、当前的细节数据集合，用于支持企业对于即时性的、操作性的、集成的全体信息的需求。对比 1.1 节中数据仓库的定义不难看出，操作型数据存储在某些方面具有类似于数据仓库的特点，但在另一些方面又显著不同于数据仓库。

- 像数据仓库一样，是面向主题的。
- 像数据仓库一样，其数据是完全集成的。
- 数据是当前的，这与数据仓库存储历史数据的性质明显不同。ODS 具有最少的历史数据（一般是 30 天到 60 天），而尽可能接近实时地展示数据的状态。
- 数据是可更新的，这是与静态数据仓库又一个很大的区别。ODS 就如同一个事务处理系统，当新的数据流进 ODS 时，受其影响的字段被新信息覆盖。
- 数据几乎完全是细节数据，仅具有少量的动态聚集或汇总数据。通常将 ODS 设计成包含事务级的数据，即包含该主题域中最低粒度级别的数据。
- 在数据仓库中，几乎没有针对其本身的报表，报表均放到数据集市中完成；与此不同，在 ODS 中，业务用户频繁地直接访问 ODS。

在一个数据仓库环境中，ODS 具有如下几个作用：

- 充当业务系统与数据仓库之间的过渡区。数据仓库的数据来源复杂，可能分布在不同的数据库、不同的地理位置、不同的应用系统之中，而且由于数据形式的多样性，数据转换的规则往往极为复杂。如果直接从业务系统抽取数据并做转换，不可避免地会对业务系统造成影响。而 ODS 中存放的数据在数据结构、数据粒度、数据之间的逻辑关系上都与业务系统基本保持一致，因此抽取过程只需简单地复制数据，而基本上不再需要做数据转换，大大降低了复杂性，同时最小化对业务系统的侵入。
- 转移部分业务系统细节查询的功能。某些原来由业务系统产生的报表、细节数据的查询能够在 ODS 中进行，从而降低业务系统的查询压力。
- 完成数据仓库中不能完成的一些功能。数据仓库用户有时会要求查询最低粒度级别的细节数据，而数据仓库中存储的数据一般都是聚合或汇总过的数据，并不存储每个事务产生的细节数据。这时就需要把细节数据查询的功能转移到 ODS 来完成，而且 ODS 的数据模型是按照面向主题的方式组织的，可以方便地支持多维分析，即数据仓库从宏观角度满足企业的决策支持要求，而 ODS 则从微观角度反映细节事务数据或者低粒度的数据查询要求。

1.5　实时数据仓库

上一节介绍的架构已经有几十年的历史，经过了长时间的验证和打磨，已被证明是适合

于构建企业级数据仓库的经典解决方案。但是，近年来随着业务领域的不断拓展，尤其像互联网、无线终端 APP 等行业应用的激增，产生的数据量呈指数级增长，对海量数据的处理需求也提出了新的挑战。具体到数据仓库，尤其突出的一点是人们对数据分析的实时性要求越来越高，从而衍生出实时数据仓库的概念。为解决数据实时性问题，也涌现出一批相关的技术。

本节将解释什么是流式处理，然后讨论实时计算的基本概念和适用场景，它们都与实时数据仓库的实施密不可分。最后从技术实现的角度介绍几种流行的实时数据仓库架构。

1.5.1 流式处理

人们对数据流并不陌生，数据从业务系统产生，经过一系列转换进入数据仓库，再进入分析系统提供报表、仪表盘展现分析结果，最终经过数据挖掘和机器学习以辅助决策，整个过程就形成了一个数据流。当然除了直觉以外，严格的定义更有意义。数据流是无边界数据集的抽象表示。无边界意味着无限和持续增长，无边界数据集之所以是无限的，是因为随着时间的推移，新的记录会不断加入进来。这个定义已经被包括 Google 和 Amazon 在内的大部分公司所采纳。

除无边界外，数据流还有其他一些属性：有序、不可变、可重放。数据的产生总有先后顺序，这是数据流与数据库表的不同点之一，数据库表里的记录是无序的。数据一旦产生就不能被改变。假设你熟悉数据库的二进制日志（binlog）、预写日志（WAL）和重做日志（redo log）的概念，那么就会知道，如果往数据库表插入一条记录，然后将其删除，表里就不会再有这条记录，但日志里包含了插入和删除两个事务。可重放是数据流非常有价值的一个属性。对于大多数业务来说，重放发生在几天前（甚至几个月前）的原始数据流是一个很重要的需求。这可能是为了尝试使用新的分析方法以纠正过去的错误，或是为了进行审计。

知道什么是数据流以后，是时候了解“流式处理”的真正含义了。流式处理是指实时处理一个或多个数据流。流式处理是一种编程范式，就像请求与响应范式和批处理范式那样。下面对这三种范式进行比较，以便更好地理解如何在软件架构中应用流式处理。

1. 请求与响应

这是延迟最小的一种范式，响应时间处于亚毫秒到毫秒之间，而且响应时间一般非常稳定。这种处理方式一般是阻塞的，应用程序向处理系统发出请求，然后等待响应。在数据库领域，这种范式就是联机事务处理（OLTP）。

2. 批处理

这种范式具有高延迟和高吞吐量的特点。处理系统按照预定的时间启动处理进程，比如每天早上两点开始启动，每小时启动一次等。它读取所有的输入数据（从上一次执行之后的所有可用数据），输出结果，然后等待下一次启动。处理时间从几分钟到几小时不等，并且用户从结果里读到的都是滞后数据。

在数据库领域，它们就是传统的数据仓库或商业智能系统。它们每天装载巨大批次的数据，并生成报表，用户在下一次装载数据之前看到的都是相同的报表。从规模上来说，这种范式既高效又经济。但在最近几年，为了能够更及时、高效地做出决策，业务要求能够在更短的时间内提供可用数据，这就给那些为探索规模经济而开发，却无法提供低延迟报表的系统带来

了巨大的压力。

3. 流式处理

这种范式介于上述两种之间。某些业务不要求毫秒级的响应，不过也接受不了要等到第二天才知道结果。大部分业务流程都是持续进行的，只要业务报告保持更新，业务产品线能够保持响应，那么业务流程就可以进行下去，而无须等待特定的响应，也不要求在几毫秒内得到响应。

与前面介绍的数据仓库相比，流式处理的整个处理过程必须是持续的。一个在每天早上两点启动的任务，从数据流里读取 500 条记录，生成结果，然后结束，这样的流程不是流式处理。对数据仓库来说，也许从粒度的角度理解流式处理更容易。回想 1.1 节中提及的粒度概念，如果能将数据更新保持在最低的事务粒度级别上，实际就是做了数据仓库的流式处理，也即所谓的实时数据仓库。实时数据处理没有调度开销，只涉及任务监控。

1.5.2　实时计算

要做到实时读写数据，必须采用有别于传统数据仓库的实现技术，实时计算的概念和技术引擎应运而生，它们是成功创建实时数据仓库的前提条件。实时计算一般针对海量数据处理，并且要求响应时间为秒级。由于大数据兴起之初，以 Hadoop 为代表的分布式框架并没有给出实时计算解决方案，随后便出现了 Storm、Spark Streaming、Flink 等实时计算框架，而 Kafka、ES 的兴起使得实时计算领域的技术越来越完善。随着物联网、机器学习等技术的推广，实时流式计算将在这些领域得到充分应用。实时计算是流式处理的一种具体实现方式，因此必然具有无限数据、无边界数据处理、低延迟等特征。

现在大数据应用比较火爆的领域，比如推荐系统，在实践之初受技术所限，可能要一分钟、一小时，甚至更久才能对用户进行推荐，这远远不能满足需求，我们要更快地完成数据处理，而不是进行离线的批处理操作。实时计算的应用场景主要包括实时智能推荐、实时欺诈检测、舆情分析、物联网、客服系统、实时机器学习等。

在某些场景中，数据的价值随着时间的推移而逐渐减少，所以在传统数据仓库的基础上，逐渐对数据的实时性提出了更高的要求。于是随之诞生了实时数据仓库，并且衍生出了两种主流技术架构：Lambda 和 Kappa。

1. Lambda 架构

Lambda 架构如图 1-7 所示，虚线上面表示数据的逻辑处理流程，虚线下面是一组具体实现组件。

数据从底层的数据源开始，经过 Kafka、Flume 等组件进行收集，然后分成两条线进行计算：一条线是进入流式计算平台（例如 Storm、Flink、Spark Streaming 等），计算一些实时的指标；另一条线是进入批量数据处理离线计算平台（例如 MapReduce、Hive，Spark SQL 等），计算 T+1 的相关业务指标，这些指标通常需要隔日才能看见。

总体来说，Lambda 架构就是为了计算一些实时指标，在原来离线数据仓库的基础上增加了一个实时计算的链路，并对数据源做流式改造：把数据发送到消息队列，实时计算去订阅消息队列，直接完成指标增量的计算，并将结果推送到下游的数据服务中去，由数据服务层完成离线、实时结果的合并。

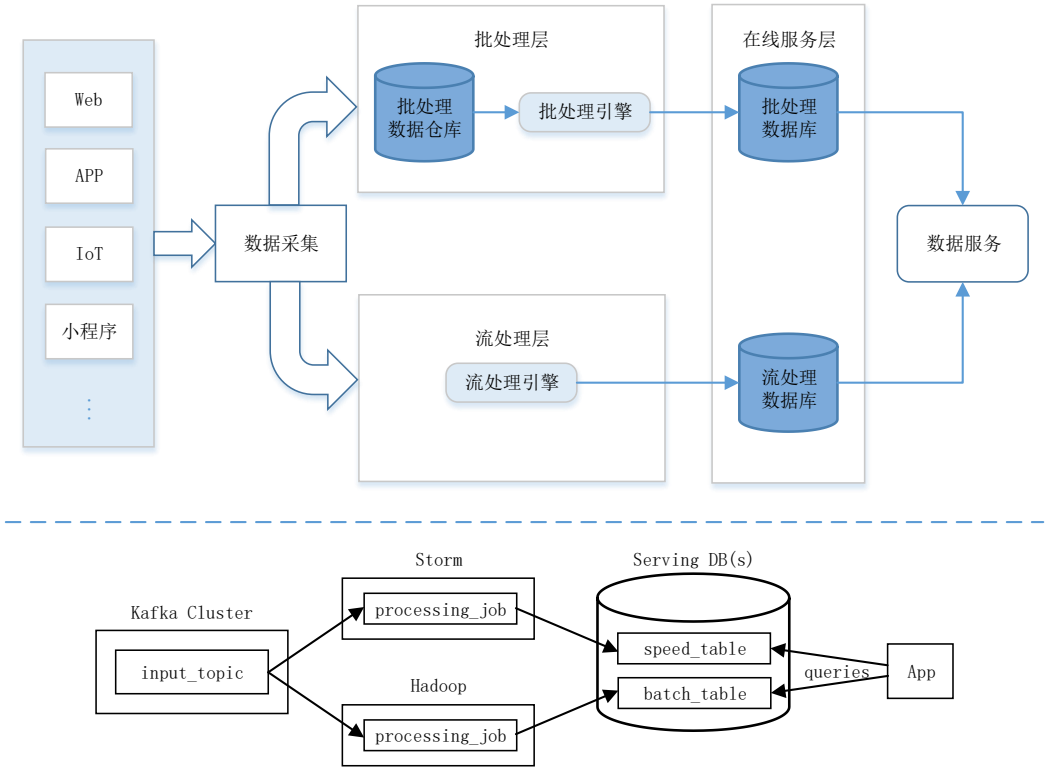

图 1-7　Lambda 架构

Lambda 属于较早的一种架构方式，早期的流处理不如现在这样成熟，在准确性、扩展性和容错性上，流处理层无法直接取代批处理层，只能给用户提供一个近似结果，还不能为用户提供一个一致和准确的结果。因此，在 Lambda 架构中出现了批处理和流处理并存的现象。

在 Lambda 架构中，每层都有自己所肩负的任务。批处理层使用可处理大量数据的分布式处理系统预先计算结果。它通过处理所有的已有历史数据来实现数据的准确性。这意味着它是基于完整的数据集来重新计算的，能够修复任何错误，然后更新现有的数据视图。输出通常存储在只读数据库中，更新则采用全量替换方式，完全取代现有的预先计算好的视图。流处理层通过提供最新数据的实时视图来最小化延迟。流处理层所生成的数据视图，可能不如批处理层最终生成的视图那样准确或完整，但它们几乎在收到数据后立即可用。当同样的数据在批处理层处理完成后，在流处理层的数据就可以被替代掉了。

Lambda 架构经历多年的发展，其优点是稳定，对于实时计算部分的计算成本可控，批量处理可以用晚上的时间来整体批量计算，这样就把实时计算和离线计算高峰分开。这种架构支撑了数据行业的早期发展，但是它也有一些致命缺点，并在当今时代越来越不适应数据分析业务的需求。

Lambda 架构的缺点如下：

- 使用两套大数据处理引擎：维护两个复杂的分布式系统，成本非常高。
- 批量计算在计算窗口内无法完成：当数据量越来越大时，夜间只有 4、5 个小时的时间窗口，已经无法完成白天约 20 个小时累计的数据，保证早上准时出数据已成为每个大数据

团队头疼的问题。

- 数据源一旦变化都要重新开发，开发周期长：每次需求变更后，业务逻辑的变化都需要针对 ETL 批处理和 Streaming 流处理做修改，整体开发周期长，业务反应不迅速。
- 资源占用增多：同样的逻辑计算两次，整体资源占用会增多（多出实时计算这部分）。

导致 Lambda 架构缺点的根本原因是要同时维护两套系统：批处理层和流处理层。我们已经知道，在架构中加入批处理层是因为从批处理层得到的结果具有高准确性，而加入流处理层是因为它在处理大规模数据时具有低延时性。那我们能不能改进其中某一层的架构，让它具有另外一层架构的特性呢？例如，改进批处理层的系统，让它具有更低的延迟，或者是改进流处理层的系统，让它产生的数据视图更具准确性和更加接近历史数据呢？另外一种在大规模数据处理中常用的架构——Kappa，便是在这样的思考下诞生的。

2. Kappa 架构

Kappa 架构可以被认为是 Lambda 架构的简化版，只是去除掉了 Lambda 架构中的离线批处理部分，如图 1-8 所示。

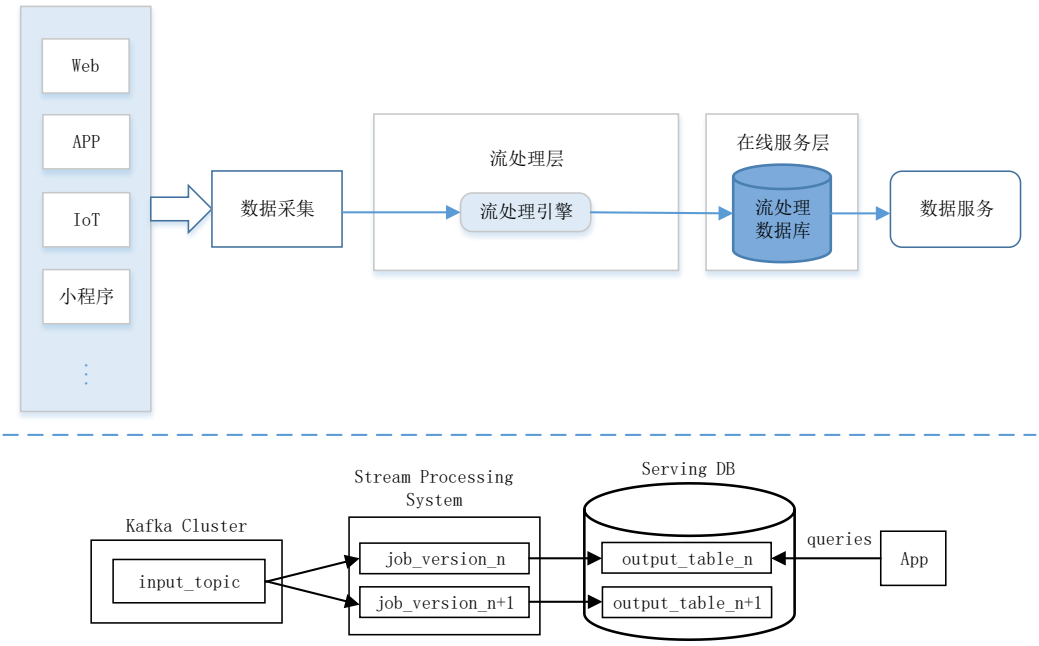

图 1-8　Kappa 架构

这种架构只关注流式计算，数据以流的方式被采集，实时计算引擎将计算结果放入数据服务层以供查询。Kappa 架构的兴起主要有两个原因：

- 消息队列（如 Kafka）支持数据持久化，可以保存更长时间的历史数据，以替代 Lambda 架构中批处理层的数据仓库部分。流处理引擎以一个更早的时间作为起点开始消费，起到了批处理的作用。
- 流处理引擎（如 Flink）解决了事件乱序下计算结果的准确性问题。

Kappa 架构更简单，实时性更好，所需的计算资源远小于 Lambda 架构，最大的问题是流式重新处理历史的吞吐能力会低于批处理，但这可通过增加计算资源来弥补。随着实时处理需求的不断增长，更多的企业开始使用 Kappa 架构，但这不意味着 Kappa 架构能够取代 Lambda 架构。Lambda 和 Kappa 架构都有各自的适用领域：对于流处理与批处理分析流程比较统一，且允许一定的容错，用 Kappa 比较合适；少量关键指标，例如交易金额、业绩统计等使用 Lambda 架构进行批量计算，增加一次校对过程。还有一些比较复杂的场景，批处理与流处理将产生不同的结果，如使用不同的机器学习模型、专家系统，或者实时计算难以处理的复杂情况，可能更适合 Lambda 架构。

1.5.3 实时数据仓库解决方案

从传统的经验来讲，数据仓库有一个很重要的功能是记录数据变化历史。通常，数据仓库都希望从业务上线的第一天开始有数据，然后一直记录到现在。但实时处理技术，又是强调当前处理状态的一门技术，所以当这两个相互对立的方案重叠在一起的时候，注定不是用来解决一个比较广泛问题的方案。于是，我们把实时数据仓库建设的目标定位为解决由于传统数据仓库数据时效性低而解决不了的问题。

实时数据仓库也引入了类似于离线数据仓库的分层理念，主要是为了提高模型的复用率，同时兼顾易用性、一致性以及计算成本。通常离线数据仓库采用空间换取时间的方式，所以层级划分比较多，从而提高数据计算效率。实时数据仓库的分层架构在设计上考虑到时效性问题，分层设计尽量精简，避免数据在流转过程中造成不必要的延迟响应，并降低中间流程出错的可能性。实时数据仓库分层架构如图 1-9 所示。

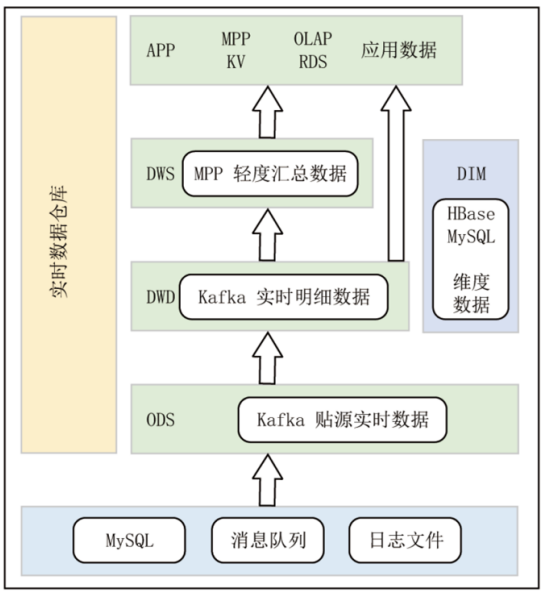

图 1-9 实时数据仓库分层架构

- ODS 层：以 Kafka 为支撑，将所有需要实时处理的相关数据放到 Kafka 队列中来实现贴源数据层。这一层是数据输入层，主要是埋点、流量、日志等消息数据的接入。

- DWD（Data Warehouse Detail）层：实时计算订阅业务数据消息队列，然后通过数据清洗、多数据源连接、流式数据与离线维度信息的组合等，将一些相同粒度的业务系统、维表中的维度属性全部关联到一起，增加数据易用性和复用性，得到最终的实时明细数据。
- DIM（Dimension）层：存放用于关联查询的维度信息，可以根据数据现状来选择存储介质，例如使用 HBase 或者 MySQL。对于实时 ETL、实时统计，或者特征加工时需要进行流数据和静态维表数据关联处理等情况，这一层是必需的。
- DWS（Data WareHouse Service）层：轻度汇总层是为了便于面向即席查询或者实时 OLAP 分析构建的轻度汇总结果集合，适合数据维度、指标信息比较多的情况。为了方便根据自定义条件的快速筛选和指标聚合，推荐使用 MPP 类型数据库进行存储。此层可视场景情况决定是否构建。
- APP 层：面向实时数据场景需求构建的高度汇总层，可以根据不同的数据应用场景决定所使用的存储介质或者引擎。APP 层已经脱离了数据仓库，这里虽然作为一个独立分层，但实际 APP 层的数据已经分布存储在各种介质中以供使用。

图 1-10 显示的是一个简化的、落地的，并基于 MySQL、Canal、Kafka、Greenplum 构建的实时数据仓库架构。本书后面讨论的实践部分都基于此架构进行设计开发。

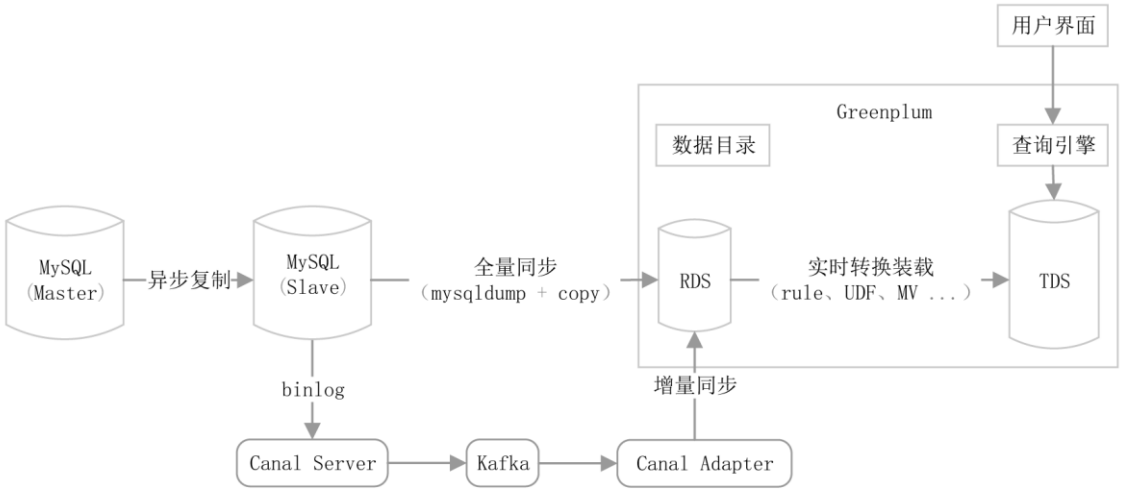

图 1-10　基于 MySQL、Canal、Kafka、Greenplum 的实时数据仓库架构

在真实的数据仓库项目中会涉及多种数据源，不同数据源产生的数据质量可能差别很大，数据库中的格式化数据可以直接导入大数据存储系统，而日志或爬虫产生的数据就需要进行大量的清洗、转化处理才能有效使用。几乎在所有领域的业务数据源中，关系数据库都占有绝对比重，而其中 MySQL 毋庸置疑是当今最流行的关系数据库系统。本书将 MySQL 作为唯一数据源，一是出于简化目的，因为后面的实践均给出代码级别的实操，不可能面面俱到；二是 MySQL 具有典型性，搞定 MySQL 的数据采集就可以解决实际应用中的大部分问题。

Canal Server 从 MySQL 从库产生的 binlog（开启 log_slave_updates）抽取增量数据变更日志，这样做有两个好处。首先，最重要的是它不会影响线上业务，因为 Canal Server 只是在从

库创建一个 Binlog Dump 线程，对 MySQL Server 的影响微乎其微。其次，从库可以随时启停复制，这样可以很容易地为下游组件确定一个增量数据同步起点，在进行首次全量数据同步时就可以有效利用这一点来实现。

Canal Server 作为数据生产者将记录数据变化的 binlog 以消息形式传递给 Kafka。Kafka 一方面可以将消息持久化存储，避免数据丢失，另一方面可以充当数据入仓前的缓冲区。Canal Adapter 作为数据消费者从 Kafka 接收消息，然后将数据写入 Greenplum 数据库。

数据进入 Greenplum 后，就可以利用它提供的规则（rule）、用户自定义函数（UDF）、物化视图（MV）等功能，自动、实时、对用户透明地执行一些复杂的 ETL 过程，以及对维度表和事实表的数据维护。Greenplum 是一种成熟的 MPP 架构的分布式数据库，提供了丰富全面的功能，并且性能优良，比较适合当作实时数据仓库的数据存储、数据处理和数据查询引擎。作为数据库管理系统，还可以利用 Greenplum 统一管理元数据。

图 1-10 所示架构具有门槛低、上手易、实施快的特点，整个构建过程只需要适当安装配置相关的软件，再利用 SQL 即可完成，不需要其他任何编程工作。当然，从来没有完美的架构，只有适合的解决方案。本架构明显的一个局限是只能处理 MySQL 一个数据源，而且始终以数据库提供的功能为核心。如果涉及非常复杂的处理逻辑，可以引入类似 Flink 的实时计算引擎，并在其上开发自己的应用程序是更好的选择。

1.6 小 结

（1）数据仓库是一个面向主题的、集成的、随时间变化的、非易失的数据集合，用于支持管理者的决策过程。

（2）数据仓库中的粒度是指数据的细节或汇总程度，细节程度越高，粒度级别越低。

（3）数据仓库的数据来自各个业务应用系统。

（4）很多因素导致直接访问业务系统无法进行全局数据分析的工作，这也是需要一个数据仓库的原因所在。

（5）操作型系统是一类专门用于管理面向事务的应用信息系统，而分析型系统是一种快速回答多维分析查询的实现方式，两者在很多方面存在差异。

（6）ETL 是建立数据仓库最重要的处理过程，也是最体现工作量的环节。

（7）构成数据仓库系统的主要组成部分有数据源、ODS、中心数据仓库、分析查询引擎、ETL、元数据管理和自动化调度。

（8）主要的数据仓库架构有独立数据集市、从属数据集市、Inmon 企业信息工厂、Kimball 多维数据仓库、混合型数据仓库。

（9）构建实时数据仓库的基础是流式处理与实时计算，Lambda 和 Kappa 是两个实时计算架构。Lambda 是早期架构，在传统离线批处理上增加了一条实时数据处理链路。Kappa 架构是 Lambda 架构的简化版，只保留了 Lambda 中的实时处理部分。

（10）实时数据仓库也引入了类似于离线数据仓库的分层理念，但更注重时效性，分层越少越好，减少分层也是为了减少中间流程出错的可能。

第2章

数据仓库设计基础

本章将首先介绍关系数据模型、多维数据模型和 Data Vault 模型这三种常见的数据仓库模型和与之相关的设计方法，然后讨论数据集市的设计问题，最后说明一个数据仓库项目的实施步骤。规划实施过程是整个数据仓库设计的重要组成部分。

关系模型、多维模型已经有很长的历史，而 Data Vault 模型相对比较新。它们都是流行的数据仓库建模方式，但又有各自的特点和适用场景。读者在学习了本章的内容后，可以根据实际需求选择适合的方法来构建自己的数据仓库。

2.1 关系数据模型

关系模型是由 E.F.Codd 于 1970 年提出的一种通用数据模型。由于关系数据模型简单明了，并且有坚实的数学理论基础，所以一经推出就受到了业界的高度重视。关系模型被广泛应用于数据处理和数据存储，尤其是在数据库领域，现在主流的数据库管理系统几乎都是以关系数据模型为基础来实现的。

2.1.1 关系数据模型中的结构

关系数据模型基于关系这一数学概念。在本小节中将解释关系数据模型中的术语和相关概念。为了便于说明，我们使用一个分公司—员工关系的例子。假设有一个大型公司，其在全国各地都有分公司，每个员工都分属于一个分公司，一个分公司有一个经理，分公司经理也是公司员工。分公司—员工关系如图 2-1 所示。

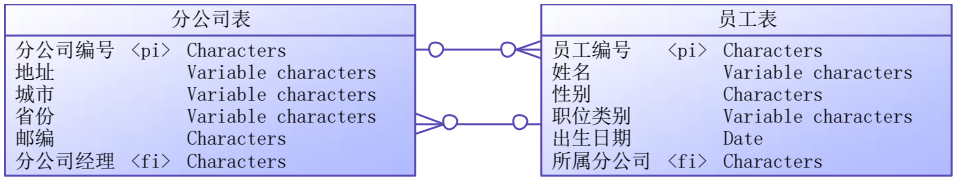

图 2-1　分公司—员工关系

1. 关系

关系是由行和列构成的二维结构，对应关系数据库中的表，如示例中的分公司表和员工表。注意，这种认识只是我们从逻辑上看待关系模型的方式，并不应用于表在磁盘上的物理结构。表的物理存储结构可以是堆文件、索引文件或哈希文件。堆文件是一个无序的数据集合；索引文件中表数据的物理存储顺序和逻辑顺序保持一致；哈希文件也称为直接存取文件，它通过一个预先定义好的哈希函数来确定数据的物理存储位置。

2. 属性

属性是由属性名称和类型名称构成的顺序对，对应关系数据库中表的列，如地址（Variable Characters）是公司表的一个属性。属性值是属性一个特定的有效值，可以是简单的标量值，也可以是复合数据类型值。

在关系数据模型中，我们把关系描述为表，表中的行对应不同的记录，表中的列对应不同的属性。属性可以以任何顺序出现，而关系保持不变，也就是说，在关系理论中，表中的列是没有顺序的。

3. 属性域

属性域即属性的取值范围，每一个属性都有一个预定义的值的范围。属性域是关系模型的一个重要特征，关系中的每个属性都与一个域相关。各个属性的域可能不同，也可能相同。属性域描述了属性所有可能的值。

属性域的概念是很重要的，因为它允许我们定义属性可以具有的值的意义。系统可因此获得更多的信息，并且可以拒绝不合理的操作。在本例中，分公司编号和员工编号都是字符串，但显然具有不同的含义，换句话说，它们的属性域是不同的。表 2-1 列出了分公司—员工关系的一些属性域。

表 2-1　分公司—员工关系的一些属性域

属性	属性域的定义	含义
分公司编号	字符：大小为 4，范围为 B001~B999	设置所有可能的分公司编号
地址	字符：大小为 100	设置所有可能的地址
员工编号	字符：大小为 5，范围为 S0001~S9999	设置所有可能的员工编号
职位类别	管理、技术、销售、运营、产品之一	设置所有可能的员工职位类别

4. 元组

元组是关系中的一条记录，对应关系数据库中的一个表行。元组可以以任何顺序出现，而关系保持不变，也就是说，在关系理论中，表中的行是没有顺序的。

5. 关系数据库

关系数据库是复制一系列规范化的表的集合。这里的规范化可以理解为表结构的正确性。在 2.1.4 节将详细讨论规范化问题。

以上介绍了关系数据模型的两组术语：“关系、属性、元组”和“表、列、行”。在这里它们的含义是相同的，只不过前者是关系数据模型的正式术语，而后者是常用的数据库术语。其他可能会遇到的类似术语还有实体（表）、记录（行）、字段（列）等。

6. 关系表的属性

关系表有如下属性：

- 每个表都有唯一的名称。
- 一个表中每个列有不同的名字。
- 一个列的值来自于相同的属性域。
- 列是无序的。
- 行是无序的。

7. 关系数据模型中的键

（1）超键

一个列或者列集，唯一标识表中的一条记录。超键可能包含用于唯一标识记录所不必要的额外的列，我们通常只对仅包含能够唯一标识记录的最小数量的列感兴趣。

（2）候选键

仅包含唯一标识记录所必需的最小数量列的超键。表的候选键有三个属性：

- 唯一性：在每条记录中，候选键的值唯一标识该记录。
- 最小性：具有唯一性属性的超键的最小子集。
- 非空性：候选键的值不允许为空。

在本例中，分公司编号是候选键，如果每个分公司的邮编都不同，那么邮编也可以作为分公司表的候选键。一个表中允许有多个候选键。

（3）主键

唯一标识表中记录的候选键。主键是唯一的、非空的。没有被选作主键的候选键称为备用键。对于例子中的分公司表，分公司编号是主键，邮编就是备用键，而员工表的主键是员工编号。

主键的选择在关系数据模型中非常重要，很多性能问题都是由于主键选择不当引起的。在选择主键时，我们可以参考以下原则：

- 主键要尽可能小。
- 主键值不应该被改变。主键会被其他表所引用，如果改变了主键的值，所有引用该主键的值都需要修改，否则引用就是无效的。
- 主键通常使用数字类型。数字类型的主键要比其他数据类型效率更高。
- 主键应该是没有业务含义的，它不应包含实际的业务信息。无意义的数字列不需要修改，

因此它是主键的理想选择。大部分关系数据库支持的自增属性或序列对象更适合当作主键。

- 虽然主键允许由多列组成，但应该使用尽可能少的列，最好是单列。

（4）外键

外键是一个表中的一列或多列的集合，这些列匹配其他（也可以是同一个）表中的候选键。

注　意
外键所引用的不一定是主键，但一定是候选键。

当一列出现在两张表中的时候，它通常代表两张表记录之间的关系。如例子中分公司表的分公司编号和员工表的所属分公司，它们的名字虽然不同，但是含义却相同。分公司表的分公司编号是主键，在员工表里所属分公司是外键。同样，因为公司经理也是公司员工，所以它是引用员工表的外键。主键所在的表被称为父表，外键所在的表被称为子表。

2.1.2 关系完整性

上一小节讨论了关系数据模型的结构部分，本小节讨论关系完整性规则。关系数据模型有两个重要的完整性规则：实体完整性和参照完整性。在定义这些术语之前，先要理解空值（NULL）的概念。

1. 空值

空值表示一个列的值目前还不知道或者对于当前记录来说不可用。空值意味着未知，也意味着某个记录没有值，或者意味着该值目前还没有提供。空值是处理不完整数据或异常数据的一种方式。空值与数字零或者空字符串不同，零和空字符串是值，但空值代表没有值，因此，空值应该与其他值区别对待。空值具有特殊性，当它参与逻辑运算时，结果取决于真值表。每种数据库系统对空值参与运算的规则定义也不尽相同。表 2-2~表 2-4 分别为大部分主流关系数据库系统（Oracle、MySQL、PostgreSQL、Greenplum 等）的非、与、或逻辑运算真值表。

表 2-2　逻辑非运算

	TRUE	FALSE	NULL
NOT	FALSE	TRUE	NULL

表 2-3　逻辑与运算

AND	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

表 2-4　逻辑或运算

OR	TRUE	FALSE	NULL
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

在本例中，如果一个分公司的经理离职了，新的经理还没有上任，此时公司经理列对应的值就是空值。

2. 关系完整性规则

有了空值的定义，就可以定义两种关系完整性规则了。

（1）实体完整性

在一个基本表中，主键列的取值不能为空。基本表指的是命名的表，其中的记录物理地存储在数据库中，与之对应的是视图。视图是虚拟的表，它只是一个查询语句的逻辑定义，其中并没有物理存储数据。

从前面介绍的定义可知，主键是用于唯一标识记录的最小列集合，也就是说，主键的任何子集都不能提供记录的唯一标识。空值代表未知，无法进行比较。如果允许空值作为主键的一部分，就意味着并不是所有的列都用来区分记录，这与主键的定义相矛盾，因此主键必须是非空的。例如，分公司编号是分公司表的主键，在录入数据的时候，该列的值不能为空。

（2）参照完整性

如果表中存在外键，则外键值必须与主表中的某些记录的候选键值相同，或者外键的值必须全部为空。在图 2-1 中，员工表中的所属分公司是外键，该列的值要么是分公司表的分公司编号列中的值，要么是空值（如新员工已经加入了公司，但还没有被分派到某个具体的分公司时）。

3. 业务规则

业务规则是定义或约束组织的某些方面的规则。业务规则的例子包括属性域和关系完整性规则。属性域用于约束特定列能够取得的值。大部分主流关系数据库系统（Oracle、MySQL、PostgreSQL、Greenplum 等）支持叫作 check 的约束，其用于定义列中可以接受的值，但这种约束是定义在属性域之上的，比属性域的约束性更强。例如，员工表的性别列就可以加上 check 约束，使它只能取有限的几个值。

2.1.3　关系数据库语言

关系语言定义了允许对数据进行的操作，包括从数据库中更新或检索数据所用的操作以及改变数据库对象结构的操作。关系数据库的主要语言是 SQL 语言。

SQL 是 Structured Query Language 的缩写，意为结构化查询语言。SQL 已经被国际标准化组织（ISO）进行了标准化，使它成为正式的和事实上的定义和操纵关系数据库的标准语言。SQL 语言又可分为 DDL、DML、DCL、TCL 四类。

DDL 是 Data Definition Language 的缩写，意为数据定义语言，用于定义数据库结构和模式。典型的 DDL 有 create、alter、drop、truncate、comment、rename 等。

DML 是 Data Manipulation Language 的缩写，意为数据操纵语言，用于检索、管理和维护数据库对象。典型的 DML 有 select、insert、update、delete、merge、call、explain、lock 等。

DCL 是 Data Control Language 的缩写，意为数据控制语言，用于授予或回收数据库对象的权限。典型的 DCL 有 grant 和 revoke。

TCL 是 Transaction Control Language 的缩写，意为事务控制语言，用于管理 DML 对数据的改变。它允许一组 DML 语句联合成一个逻辑事务。典型的 TCL 有 commit、rollback、savepoint、set transaction 等。

2.1.4 规范化

关系数据模型的规范化是一种组织数据的技术。规范化方法对表进行分解，以消除数据冗余，避免异常更新，提高数据完整性。没有规范化，数据的更新处理将变得困难，且异常的插入、修改、删除数据的操作也会频繁发生。为了便于理解，来看下面的例子。

假设有一个名为 employee 的员工表，它有九个属性：id（员工编号）、name（员工姓名）、mobile（电话）、zip（邮编）、province（省份）、city（城市）、district（区县）、deptNo（所属部门编号）、deptName（所属部门名称），表中的数据如表 2-5 所示。

表 2-5 非规范化的员工表

id	name	mobile	zip	province	city	district	deptNo	deptName
101	张三	13910000001 13910000002	100001	北京	北京	海淀区	D1	部门 1
101	张三	13910000001 13910000002	100001	北京	北京	海淀区	D2	部门 2
102	李四	13910000003	200001	上海	上海	静安区	D3	部门 3
103	王五	13910000004	510001	广东省	广州	白云区	D4	部门 4
103	王五	13910000004	510001	广东省	广州	白云区	D5	部门 5

由于此员工表是非规范化的，我们将面临如下问题。

- 修改异常：上表中张三有两条记录，因为他隶属两个部门。如果我们要修改张三的地址，必须修改两行记录。假如一个部门得到了张三的新地址并进行了更新，而另一个部门没有，那么此时张三在表中会存在两个不同的地址，导致数据不一致。
- 新增异常：假如一个新员工加入公司，他正处于入职培训阶段，还没有被正式分配到某个部门，如果 deptNo 字段不允许为空，我们就无法向 employee 表中新增该员工的数据。
- 删除异常：假设公司撤销了 D3 这个部门，那么在删除 deptNo 为 D3 的行时，会将李四的信息也一并删除，因为他只隶属于 D3 这一个部门。

为了克服这些异常更新，我们需要对表进行规范化设计。规范化是通过应用范式规则实现的。最常用的范式有第一范式（1NF）、第二范式（2NF）、第三范式（3NF）。

（1）第一范式

表中的列只能含有原子性（不可再分）的值。表 2-5 中张三有两个手机号存储在 mobile

列中，违反了 1NF 规则。为了使表满足 1NF，数据应该修改为如表 2-6 所示。

表 2-6　满足 1NF 的员工表

id	name	mobile	zip	province	city	district	deptNo	deptName
101	张三	13910000001	100001	北京	北京	海淀区	D1	部门 1
101	张三	13910000002	100001	北京	北京	海淀区	D1	部门 1
101	张三	13910000001	100001	北京	北京	海淀区	D2	部门 2
101	张三	13910000002	100001	北京	北京	海淀区	D2	部门 2
102	李四	13910000003	200001	上海	上海	静安区	D3	部门 3
103	王五	13910000004	510001	广东省	广州	白云区	D4	部门 4
103	王五	13910000004	510001	广东省	广州	白云区	D5	部门 5

（2）第二范式

第二范式要同时满足两个条件：满足第一范式；没有部分依赖。例如，员工表的一个候选键是{id，mobile，deptNo}，而 deptName 依赖于{deptNo}，同样地，name 仅依赖于{id}，因此不是 2NF 的。为了满足第二范式的条件，需要将这个表拆分成 employee、dept、employee_dept、employee_mobile 四个表，如表 2-7~表 2-10 所示。

表 2-7　满足 2NF 的员工表

id	name	zip	province	city	district
101	张三	100001	北京	北京	海淀区
102	李四	200001	上海	上海	静安区
103	王五	510001	广东省	广州	白云区

表 2-8　满足 2NF 的部门表

deptNo	deptName
D1	部门 1
D2	部门 2
D3	部门 3
D4	部门 4
D5	部门 5

表 2-9　满足 2NF 的员工—部门表

id	deptNo
101	D1
101	D2
102	D3
103	D4
103	D5

表 2-10　满足 2NF 的员工—电话话码表

id	mobile
101	13910000001

（续表）

id	mobile
101	13910000002
102	13910000003
103	13910000004

（3）第三范式

第三范式要同时满足两个条件：满足第二范式；没有传递依赖。例如，员工表的 province、city、district 依赖于 zip，而 zip 依赖于{id}，换句话说，province、city、district 传递依赖于{id}，违反了 3NF 规则。为了满足第三范式的条件，可以将这个表拆分成 employee 和 zip 两个表，如表 2-11、表 2-12 所示。

表 2-11　满足 3NF 的员工表

id	name	zip
101	张三	100001
102	李四	200001
103	王五	510001

表 2-12　满足 3NF 的地区表

zip	province	city	district
100001	北京	北京	海淀区
200001	上海	上海	静安区
510001	广东省	广州	白云区

在关系数据模型设计中，一般需要满足第三范式的要求。如果一个表有良好的主、外键设计，就应该是满足 3NF 的表。规范化带来的好处是，通过减少数据冗余提高了更新数据的效率，同时保证了数据完整性。然而，我们在实际应用中也要防止过度规范化的问题。规范化程度越高，划分的表就越多，在查询数据时越有可能使用表连接操作。而如果连接的表过多，会影响查询的性能。因此要依据业务需求，仔细权衡数据查询和数据更新的关系，制定最适合的规范化程度。还有一点需要注意，不要为了遵循严格的规范化规则而修改业务需求。

2.1.5　关系数据模型与数据仓库

关系数据模型可以提供高性能的数据更新操作，能很好地满足事务型系统的需求，这点毋庸置疑，但是其对于查询与分析密集型的数据仓库系统是否还合适呢？对这个问题的争论由来已久，基本可以分为 Inmon 和 Kimball 两大阵营，Inmon 阵营是应用关系数据模型构建数据仓库的支持者，Kimball 阵营则不支持。

Inmon 方法是以下面这些假设的成立为前提的。

- 假设数据仓库是以企业为中心的，初始的数据能够为所有部门所使用。而最终的数据分析能力是在部门级别体现，需要使用数据集市对数据仓库中的数据做进一步处理，以便为特定的部门定制它们。

- 数据仓库中的数据不违反组织制定的任何业务规则。
- 必须尽可能快地把新数据装载进数据仓库，这意味着需要简化数据装载过程或减少数据的装载量。
- 数据仓库的建立必须从一开始就被设计成支持多种 BI 技术，这就要求数据仓库本身所使用的技术越通用越好。
- 假设数据仓库的需求一定会发生变化，则数据仓库必须能完美地适应其数据和数据结构的变化。

基于这些假设，使用关系数据模型构建数据仓库的优势和必然性就比较明显了。

1. 非冗余性

为适应数据仓库有限的装载周期和海量数据，数据模型应该包含最少量的数据冗余。冗余越少，需要装载的数据量就越少，装载过程就越快。另外，数据仓库的数据源一般是事务型系统，这些系统通常是规范化设计的。如果数据仓库使用相同的数据模型，意味着数据转换的复杂性会降低，这同样可以加快数据装载的速度。

2. 稳定性

一方面，由于数据仓库的需求会不断变化，我们需要以一种迭代的方式建立数据仓库。众所周知，组织中最经常变化的是它的处理过程、应用和技术，如果依赖这三个因素中的任何一个建立数据模型，当它们发生改变时，肯定要对数据模型进行彻底修改。而这些问题，正是关系数据模型的通用性的用武之地。另一方面，由于变化不可避免，数据仓库模型应该能比较容易地将新的变化合并进来，而不必重新设计已有的元素和已经实现的实体。

3. 一致性

数据仓库模型最本质的特点是保证作为组织最重要资源的数据的一致性，而确保数据一致性正是关系数据模型的特点之一。

4. 灵活性

数据仓库最重要的一个用途是作为坚实的、可靠的、一致的数据基础，为后续的报表系统、数据分析、数据挖掘或 BI 系统提供服务。数据模型还必须支持为组织建立的业务规则，这就意味着数据模型必须比简单的平面文件功能更强。对此关系数据模型也是最佳选择之一。

关系数据模型已被证明是可靠的、简单的数据建模方法。应用其规范化规则，将产生一个稳定的、一致的数据模型。该模型支持由组织制定的政策和约定的规则，同时为数据集市分析数据提供了更多的灵活性，在数据存储以及数据装载方面也是最有效的。

当然，任何一种数据模型都不可能是完美无瑕的。关系数据模型的缺点也很明显，它需要额外建立数据集市的存储区，并增加相应的数据装载过程。另外，对数据仓库的使用强烈依赖于对 SQL 语言的掌握程度。

2.2 维度数据模型

维度数据模型简称维度模型（Dimensional Modeling，DM），是一套技术和概念的集合，用于数据仓库设计。不同于关系数据模型，维度模型不一定要引入关系数据库。在逻辑上相同的维度模型，可以被用于多种物理形式，比如维度数据库或是简单的平面文件。根据数据仓库大师 Kimball 的观点，维度模型是一种趋向于支持最终用户对数据仓库进行查询的设计技术，是围绕性能和易理解性构建的。尽管关系模型对于事务处理系统的表现非常出色，但它并不是面向最终用户的。

事实和维度是两个维度模型中的核心概念。事实表示对业务数据的度量，而维度是观察数据的角度。事实通常是数字类型的，可以进行聚合和计算，而维度通常是一组层次关系或描述信息，用来定义事实。例如，销售金额是一个事实，而销售时间、销售的产品、购买的顾客、商店等都是销售事实的维度。维度模型按照业务流程领域即主题域建立，例如进货、销售、库存、配送等。不同的主题域可能共享某些维度，为了提高数据操作的性能和数据一致性，需要使用一致性维度，例如几个主题域间共享维度的复本。术语“一致性维度”源自 Kimball，指的是具有相同属性和内容的维度。

2.2.1 维度数据模型建模过程

维度模型通常以一种被称为星型模式的方式构建。所谓星型模式，就是以一个事实表为中心，周围环绕着多个维度表。还有一种模式叫作雪花模式，是对维度做进一步规范化后形成的。本节后面将会讨论这两种模式。一般使用下面的过程构建维度模型：

- 选择业务流程。
- 声明粒度。
- 确认维度。
- 确认事实。

这种使用四步设计法建立维度模型的过程，有助于保证维度模型和数据仓库的可用性。

1. 选择业务流程

确认哪些业务处理流程是数据仓库应该覆盖的，是维度方法的基础。因此，建模的第一步是描述需要建模的业务流程。例如，需要了解和分析一个零售店的销售情况，那么与该零售店销售相关的所有业务流程都是需要关注的。为了描述业务流程，可以简单地使用纯文本将相关内容记录下来，或者使用“业务流程建模标注”（BPMN）方法，也可以使用统一建模语言（UML）或其他类似的方法。

2. 声明粒度

确定了业务流程后，下一步是声明维度模型的粒度。这里的粒度用于确定事实中表示的

是什么，例如，一个零售店的顾客在购物小票上的一个购买条目。在选择维度和事实前必须声明粒度，因为每个候选维度或事实必须与定义的粒度保持一致。在一个事实所对应的所有维度设计中，强制实行粒度一致性是保证数据仓库应用性能和易用性的关键。从给定的业务流程获取数据时，原始粒度是最低级别的粒度。建议从原始粒度数据开始设计，因为原始记录能够满足无法预期的用户查询。汇总后的数据粒度对优化查询性能很重要，但这样的粒度往往不能满足对细节数据的查询需求。不同的事实可以有不同的粒度，但同一事实中不要混用多种不同的粒度。维度模型建立完成之后，还有可能因为获取了新的信息，而回到这一步修改粒度级别。

3. 确认维度

设计过程的第三步是确认模型的维度。维度的粒度必须和第二步所声明的粒度一致。维度表是事实表的基础，也说明了事实表的数据是从哪里采集来的。典型的维度都是名词，如日期、商店、库存等。维度表存储了某一维度的所有相关数据，例如，日期维度应该包括年、季度、月、周、日等数据。

4. 确认事实

确认维度后，下一步也是维度模型四步设计法的最后一步，即确认事实。这一步识别数字化的度量，构成事实表的记录。它是和系统的业务用户密切相关的，因为用户正是通过对事实表的访问来获取数据仓库存储的数据。大部分事实表的度量都是数字类型的，可累加，可计算，如成本、数量、金额等。

2.2.2　维度规范化

与关系模型类似，维度也可以进行规范化。对维度进行规范化（又叫雪花化），可以去除冗余属性，是对非规范化维度做的规范化处理，在下面介绍雪花模型时，会看到维度规范化的例子。一个非规范化维度对应一个维度表，规范化后，一个维度会对应多个维度表，维度被严格地以子维度的形式连接在一起。实际上，在很多情况下，维度规范化后的结构等同于一个低范式级别的关系型结构。

设计维度数据模型时，会因为如下原因而不对维度做规范化处理：

- 规范化会增加表的数量，使结构更复杂。
- 不可避免的多表连接，使查询更复杂。
- 不适合使用位图索引。
- 查询性能原因。分析型查询需要聚合计算或检索很多维度值，此时第三范式的数据库会遭遇性能问题。如果需要的仅仅是操作型报表，可以使用第三范式，因为操作型系统的用户需要看到更细节的数据。

正如前面在关系模型中提到的，对于是否应该规范化的问题存在一些争论。总体来说，当多个维度共用某些通用的属性时，做规范化会是有益的。例如，客户和供应商都有省、市、区县、街道等地理位置的属性，此时分离出一个地区属性就比较合适。

2.2.3 维度数据模型的特点

维度数据模型有如下三个特点：

（1）易理解

相对于规范化的关系模型，维度模型容易理解且更直观。在维度模型中，信息按业务种类或维度进行分组，这会提高信息的可读性，也方便了对于数据含义的解释。简化的模型也让系统以更为高效的方式访问数据库。在关系模型中，数据被分布到多个离散的实体中，对于一个简单的业务流程，可能需要很多表联合在一起才能表示。

（2）高性能

维度模型更倾向于非规范化，因为这样可以优化查询的性能。在介绍关系模型时多次提到，规范化的实质是减少数据冗余，以优化事务处理或数据更新的性能。这里用一个具体的例子进一步说明性能问题。如图 2-2 所示，左边是一个销售订单的典型的规范化表示。订单（Order）实体描述有关订单整体的信息，订单明细（Order Line）实体描述有关订单项的信息，两个实体都包含描述其订单状态的信息。右边是一个订单状态维（Order Status Dimension），该维描述订单和订单明细中对应的状态编码值的唯一组合，它包括在规范化设计的订单和订单明细实体中都出现的属性。当销售订单事实行被装载时，参照在订单状态维中的适合的状态编码的组合设置它的外键。

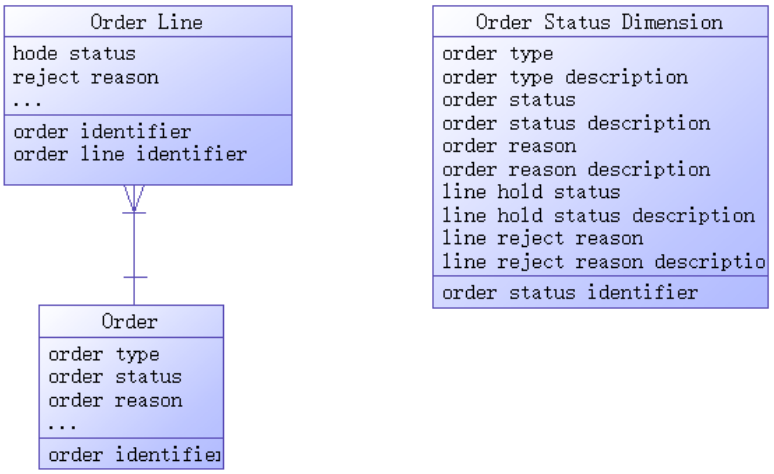

图 2-2　销售订单规范化表与销售订单维度表

维度设计的整体观点是要简化和加速查询。假设有 100 万个订单，每个订单有 10 条明细，订单状态和订单明细状态各有 10 种。一方面，如果用户要查询某种状态特性的订单，按 3NF 模型，逻辑上需要关联 100 万记录与 1000 万记录的两个大表，然后过滤两个表的状态值得到所要的结果。另一方面，事实表（图中并没有画出）按最细数据粒度有 1000 万记录，3NF 里的订单表属性在事实表里是冗余数据，状态维度有 100 条数据，只需要关联 1000 万记录与 100 条记录的两个表，再进行状态过滤即可。

（3）可扩展

维度模型是可扩展的。由于维度模型允许数据冗余，因此当向一个维度表或事实表中添加字段时，不会像关系模型那样产生巨大的影响，带来的结果就是更容易容纳不可预料的新增数据。这种新增可以是单纯地向表中增加新的数据行而不改变表结构，也可以是在现有表上增加新的属性。基于数据仓库的查询和应用不需要过多改变就能适应表结构的变化，老的查询和应用会继续工作而不会产生错误的结果。但是对于规范化的关系模型，由于表之间存在复杂的依赖关系，改变表结构前一定要仔细考虑。

2.2.4　星型模式

星型模式是维度模型最简单的形式，也是数据仓库以及数据集市开发中使用最广泛的形式。星型模式由事实表和维度表组成，一个星型模式中可以有一个或多个事实表，每个事实表引用任意数量的维度表。星型模式的物理模型像一颗星星的形状，中心是一个事实表，围绕在事实表周围的维度表表示星星的放射状分支，这就是星型模式这个名字的由来。

星型模式将业务流程分为事实和维度。事实包含业务的度量，是定量的数据，如销售价格、销售数量、距离、速度、重量等是事实。维度是对事实数据属性的描述，如日期、产品、客户、地理位置等是维度。一个含有很多维度表的星型模式有时被称为蜈蚣模式，显然这个名字也是因其形状而得来的。蜈蚣模式的维度表往往只有很少的几个属性，这样可以简化对维度表的维护，但查询数据时会有更多的表连接，严重时会使模型难于使用，因此在设计中应该尽量避免蜈蚣模式。

1. 事实表

事实表记录了特定事件的数字化的考量，一般由数字值和指向维度表的外键组成。通常会把事实表的粒度级别设计得比较低，使得事实表可以记录很原始的操作型事件，但这样做的负面影响是累加大量记录可能会非常耗时。事实表有以下三种类型：

- 事务事实表。记录特定事件的事实，如销售。
- 快照事实表。记录给定时间点的事实，如月底账户余额。
- 累积事实表。记录给定时间点的聚合事实，如当月的总的销售金额。

一般需要给事实表设计一个代理键作为每行记录的唯一标识。代理键是由系统生成的主键，它不是应用数据，没有业务含义，对用户来说是透明的。

2. 维度表

维度表的记录数通常比事实表少，但每条记录包含大量用于描述事实数据的属性字段。维度表可以定义各种各样的特性，以下是几种常用的维度表：

- 时间维度表。描述星型模式中记录的事件所发生的时间，具有所需的最低级别的时间粒度。数据仓库是随时间变化的数据集合，需要记录数据的历史，因此每个数据仓库都需要一个时间维度表。
- 地理维度表。描述位置信息的数据，如国家、省份、城市、区县、邮编等。

- 产品维度表。描述产品及其属性。
- 人员维度表。描述人员相关的信息，如销售人员、市场人员、开发人员等。
- 范围维度表。描述分段数据的信息，如高级、中级、低级等。

通常给维度表设计一个单列、整型数字类型的代理键，映射业务数据中的主键。业务系统中的主键本身可能是自然键，也可能是代理键。自然键指的是由现实世界中已经存在的属性组成的键，如身份证号就是典型的自然键。

3. 优点

星型模式是非规范化的，在星型模式的设计开发过程中，不受应用于事务型关系数据库的范式规则的约束。星型模式的优点如下：

- 简化查询。查询数据时，星型模式的连接逻辑比较简单，而从高度规范化的事务模型中查询数据时，往往需要更多的表连接。
- 简化业务报表逻辑。与高度规范化的模式相比，由于查询更简单，因此星型模式简化了普通的业务报表（如每月报表）逻辑。
- 获得查询性能。星型模式可以提升只读报表类应用的性能。
- 快速聚合。基于星型模式的简单查询能够提高聚合操作的性能。
- 便于向立方体提供数据。星型模式被广泛用于高效地建立 OLAP 立方体，几乎所有的 OLAP 系统都提供 ROLAP 模型（关系型 OLAP），它可以直接将星型模式中的数据当作数据源，而不用单独建立立方体结构。

4. 缺点

星型模式的主要缺点是不能保证数据完整性。一次性的插入或更新操作可能会造成数据异常，而这种情况在规范化模型中是可以避免的。星型模式的数据装载，一般都是以高度受控的方式，用批处理或准实时过程执行，以此来抵消数据保护方面的不足。

星型模式的另一个缺点是对于分析需求来说不够灵活。它更偏重于为特定目的建造数据视图，因此实际上很难进行全面的数据分析。星型模式不能自然地支持业务实体的多对多关系，需要在维度表和事实表之间建立额外的桥接表。

5. 示例

假设有一个连锁店的销售数据仓库，记录销售相关的日期、商店和产品，其星型模式如图 2-3 所示。

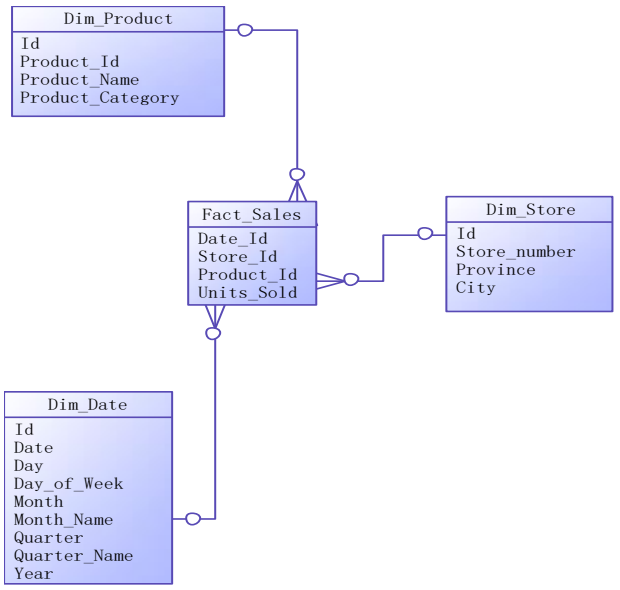

图 2-3　星型模式的销售数据仓库

Fact_Sales 是唯一的事实表，Dim_Date、Dim_Store 和 Dim_Product 是三个维度表。每个维度表的 Id 字段是它们的主键。事实表的 Date_Id、Store_Id、Product_Id 三个字段构成了事实表的联合主键，同时这个三个字段也是外键，分别引用对应的三个维度表的主键。Units_Sold 是事实表的唯一一个非主键列，代表销售量，是用于计算和分析的度量值。维度表的非主键列表示维度的附加属性。下面的查询可以回答 2021 年各个城市的手机销量是多少。

```
select s.city as city, sum(f.units_sold)
from fact_sales f
inner join dim_date d on (f.date_id = d.id)
inner join dim_store s on (f.store_id = s.id)
inner join dim_product p on (f.product_id = p.id)
where d.year = 2021 and p.product_category = 'mobile'
group by s.city;
```

2.2.5　雪花模式

雪花模式是一种多维模型中表的逻辑布局，其实体关系图类似于雪花的形状，因此得名。与星型模式相同，雪花模式也是由事实表和维度表组成。所谓的“雪花化”就是将星型模式中的维度表进行规范化处理，当所有的维度表完成规范化后，就形成了以事实表为中心的雪花型结构，即雪花模式。将维度表进行规范化的具体做法是，把低基数的属性从维度表中移除并形成单独的表。基数指的是一个字段中不同值的个数，如主键列具有唯一值，所以有最高的基数，而像性别这样的列基数就很低。

在雪花模式中，一个维度被规范化成多个关联的表，而在星型模式中，每个维度由一个单一的维度表所表示。一个规范化的维度对应一组具有层次关系的维度表，而事实表作为雪花模式里的子表，存在具有层次关系的多个父表。

星型模式和雪花模式都是建立维度数据仓库或数据集市的常用方式，适用于加快查询速度的重要性高于高效维护数据的场景。这些模式中的表没有特别的规范化，一般都被设计成低于第三范式的级别。

1. 数据规范化与存储

规范化的过程就是将维度表中重复的组分离成一个新表，以减少数据冗余的过程。正因为如此，规范化不可避免地增加了表的数量。在执行查询的时候，不得不连接更多的表。但是规范化减少了存储数据的空间需求，而且提高了数据更新的效率。这点在前面介绍关系模型时已经进行了详细的讨论。

从存储空间的角度看，典型的情况是维度表比事实表小很多。这就使得雪花化的维度表相对于星型模式来说，在存储空间上的优势没那么明显了。举例来说，假设在 220 个区县的 200 个商场，共有 100 万条销售记录。星型模式的设计会产生 1,000,200 条记录，其中事实表有 1,000,000 条记录，商场维度表有 200 条记录，每个区县信息作为商场的一个属性，显式地出现在商场维度表中。在规范化的雪花模式中，会建立一个区县维度表，该表有 220 条记录，商场表引用区县表的主键，有 200 条记录，事实表没有变化，还是 1,000,000 条记录，总的记录数是 1,000,420（1,000,000+200+220）。在这种作为子表的商场记录数少于作为父表的区县记录数的特殊情况下，星型模式所需的空间反而比雪花模式要少。如果商场有 10,000 个，情

况就不一样了，星型模式的记录数是 1,010,000，雪花模式的记录数是 1,010,220，从记录数上看，还是雪花模型多，但是，星型模式的商场表中会有 10,000 个冗余的区县属性信息，而在雪花模式中，商场表中只有 10,000 个区县的主键，而需要存储的区县属性信息只有 220 个，当区县的属性很多时，会大大减少数据存储所占用的空间。

有些数据库开发者采取一种折中的方式，底层使用雪花模型，上层用表连接建立视图模拟星型模式。这种方法既通过对维度的规范化节省了存储空间，同时又对用户屏蔽了查询的复杂性。但是当外部的查询条件不需要连接整个维度表时，这种方法会带来性能损失。

2. 优点

雪花模式是和星型模式类似的逻辑模型。实际上，星型模式是雪花模式的一个特例（维度没有多个层级）。在某些条件下，雪花模式更具优势：

- 一些 OLAP 多维数据库建模工具专为雪花模型进行了优化。
- 规范化的维度属性节省了存储空间。

3. 缺点

雪花模型的主要缺点是维度属性规范化增加了查询的连接操作和复杂度。相对于平面化的单表维度，多表连接的查询性能会有所下降。但雪花模型的查询性能问题，在近年来随着数据浏览工具的不断优化而得到缓解。

和具有更高规范化级别的事务型模式相比，雪花模式并不确保数据完整性。向雪花模式的表中装载数据时，一定要有严格的控制和管理，避免数据的异常插入或更新。

4. 示例

图 2-4 显示的是将图 2-3 的星型模式规范化后的雪花模式。日期维度分解成季度、月、周、日期四个表。产品维度分解成产品分类、产品两个表。商场维度分解出一个地区表。

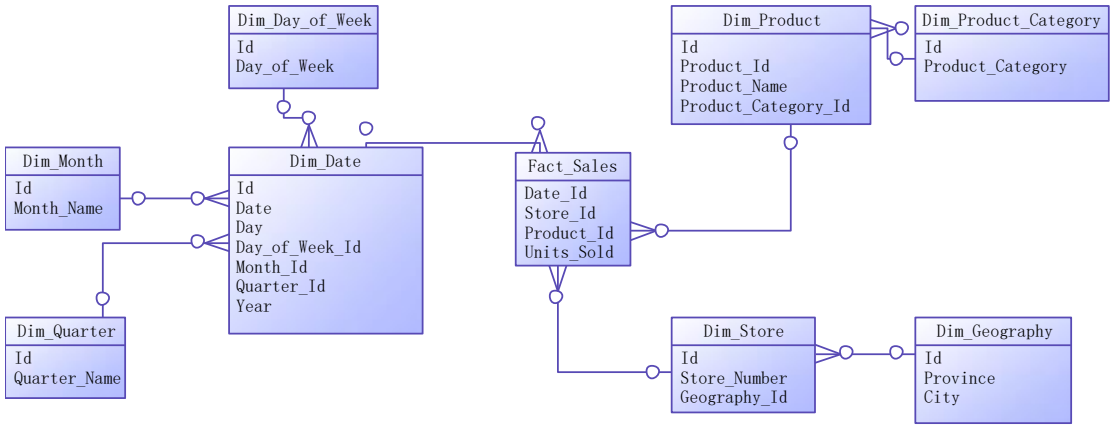

图 2-4　雪花模式的销售数据仓库

下面所示的查询语句的结果等价于前面星型模式的查询，可以明显看到此查询比星型模式的查询有更多的表连接。

```
select g.city,sum(f.units_sold)
  from fact_sales f
```

```
    inner join dim_date d on f.date_id = d.id
    inner join dim_store s on f.store_id = s.id
    inner join dim_geography g on s.geography_id = g.id
    inner join dim_product p on f.product_id = p.id
    inner join dim_product_category c on p.product_category_id = c.id
    where d.year = 2021 and c.product_category = 'mobile'
    group by g.city;
```

2.3　Data Vault 模型

Data Vault（DV）是一种数据仓库建模方法，用来存储来自多个操作型系统的完整的历史数据。Data Vault 方法需要跟踪所有数据的来源，因此其中每个数据行都要包含数据来源和装载时间属性，用以审计和跟踪数据值所对应的源系统。Data Vault 不区分数据在业务层面的正确与错误，它保留操作型系统的所有时间的所有数据，在装载数据时不做数据验证、清洗等工作，这点明显有别于其他数据仓库建模方法。Data Vault 建模方法显式地将结构信息和属性信息分离，能够还原业务环境的变化。Data Vault 允许并行数据装载，不需要重新设计就可以实现扩展。

2.3.1　Data Vault 模型简介

Data Vault 模型用于企业级的数据仓库建模，是 Dan Linstedt 在 20 世纪 90 年代提出的。在最近几年，Data Vault 模型获得了很多关注。Dan Linstedt 将 Data Vault 模型定义如下：

> Data Vault 是面向细节的、可追踪历史的、一组有连接关系的、规范化的表的集合。这些表可以支持一个或多个业务功能。它是一种综合了第三范式和星型模型优点的建模方法。其设计理念是要满足企业对灵活性、可扩展性、一致性和对需求的适应性要求，是一种专为企业级数据仓库量身定制的建模方式。

从上面的定义可以看出，Data Vault 既是一种数据建模的方法论，又是构建企业数据仓库的一种具体方法。在 Data Vault 建模方法论中，不仅定义了 Data Vault 各组成部分之间的交互方式，还包括了最佳实践来指导构建企业数据仓库。例如，业务规则应该在数据的下游实现，也就是说，Data Vault 只按照业务数据的原样保存数据，不做任何解释、过滤、清洗、转换。即使从不同数据源来的数据是自相矛盾的（例如同一个客户有不同的地址），Data Vault 模型也不会遵照任何业务的规则，如“以系统 A 的地址为准”。Data Vault 模型会保存两个不同版本的数据，对数据的解释将推迟到整个架构的后一个阶段（数据集市）。

2.3.2　Data Vault 模型的组成部分

Data Vault 模型有中心表（Hub）、链接表（Link）、附属表（Satellite）三个主要组成部分。中心表记录业务主键，链接表记录业务关系，附属表记录业务描述。

1. 中心表

中心表用来保存一个组织内的每个实体的业务主键，业务主键唯一标识某个业务实体。中心表和源系统表是相互独立的。当一个业务主键被用在多个系统时，它在 Data Vault 中也只保留一份，其他的组件都链接到这一个业务主键上，这就意味着业务数据都集成到了一起。表 2-13 列出了中心表的属性及其描述。

表 2-13　中心表的属性及其描述

属性	描述
主键	系统生成的代理键，供内部使用
业务主键	唯一标识的业务单元，用于已知业务的源系统
装载时间	数据第一次装载到数据仓库时系统生成的时间戳
数据来源	定义了数据来源（例如源系统或表）

2. 链接表

链接表是中心表之间的链接。一个链接表意味着两个或多个中心表之间有关联。一个链接表通常是一个外键，它代表着一种业务关系。表 2-14 列出了链接表的属性及其描述。

表 2-14　链接表的属性及其描述

属性	描述
主键	系统生成的代理键，供内部使用
外键{1…N}	引用中心表的代理键
装载时间	数据第一次装载到数据仓库时系统生成的时间戳
数据来源	定义了数据来源（例如源系统或表）

在 Data Vault 里，每个关系都以多对多的方式关联，这给模型带来了很大的灵活性。无论数据在源系统中是什么关系，都可以保存在 Data Vault 模型中。

3. 附属表

附属表用来保存中心表和链接表的属性，包括所有的历史变化数据。一个附属表有且只有一个外键引用到中心表或链接表。表 2-15 列出了附属表的属性及其描述。

表 2-15　附属表的属性及其描述

属性	描述
主键	系统生成的代理键，供内部使用
外键	引用中心表或链接表的代理键
装载时间	数据第一次装载到数据仓库时系统生成的时间戳
失效时间	数据失效时的时间戳
数据来源	定义了数据来源（例如源系统或表）
属性{1…N}	属性自身

在 Data Vault 模型的标准定义里，附属表的主键应该是附属表里参照到中心表或链接表的外键字段和装载时间字段的组合。尽管这个定义是正确的，但从技术角度考虑，我们最好还是增加一个代理键。使用只有一列的代理键更易维护。另外，对外键列和装载时间列联合建立唯

一索引，也是一个好习惯。

2.3.3　Data Vault 模型的特点

一个设计良好的 Data Vault 模型应该具有以下特点：

- 所有数据都基于时间来存储，即使数据是低质量的，也不能在 ETL 过程中处理掉。
- 依赖越少越好。
- 和源系统越独立越好。
- 设计上适合变化。
  - 源系统中数据的变化。
  - 在不改变模型的情况下可扩展。
- ETL 作业可以重复执行。
- 数据完全可追踪。

2.3.4　Data Vault 模型的构建

在 Data Vault 模型中，各个实体有着严格、通用的定义与准确、灵活的功能描述，这不但使得 Data Vault 模型能够最直观、最普遍地反映数据之间内含的业务规则，同时也为构建 Data Vault 模型提供了一致而普遍的方法。

Data Vault 模型的建立可以遵循如下步骤：

1. 设计中心表

首先要确定企业数据仓库要涵盖的业务范围；其次要将业务范围划分为若干原子业务实体，比如客户、产品等；然后，从各个业务实体中抽象出能够唯一标识该实体的业务主键，该业务主键要在整个业务的生命周期内不会发生变化；最后，由该业务主键生成中心表。

2. 设计链接表

链接表体现中心表之间的业务关联。设计链接表，首先要熟悉各个中心表代表的业务实体之间的业务关系，可能是两个或者多个中心表之间的关系。根据业务需求，这种关系可以是一对一、一对多、或者多对多的。

然后，从相互之间有业务关系的中心表中，提取出代表各自业务实体的中心表主键，这些主键将被加入到链接表中，组合构成该链接表的主键。同样出于技术的原因，需要增加代理键。

生成链接表时要注意，如果中心表之间有业务交易数据的话，就需要在链接表中保存交易数据。有两种方法，一是采用加权链接表，二是给链接表加上附属表来处理交易数据。

3. 设计附属表

附属表包含了各个业务实体与业务关联的详细的上下文描述信息。设计附属表，首先要收集各个业务实体在提取业务主键后的其他信息，比如客户住址、产品价格等；由于同一业务实体的各个描述信息不具有稳定性，会经常发生变化，所以在必要时，需要将变化频率不同的

信息分隔开来，为一个中心表建立几个附属表，然后提取出该中心表的主键，作为描述该中心表的附属表的主键。

当业务实体之间存在交易数据的时候，需要为没有加权的链接表设计附属表，也可以根据交易数据的不同变化情况设计多个附属表。

4. 设计必要的 PIT 表

PIT（Point-In-Time）表是由附属表派生而来的。如果一个中心表或者链接表设计有多个附属表的话，为了访问数据方便，就可以使用 PIT 表。

PIT 表的主键也是由其所归属的中心表提取而来，该中心表有几个附属表，PIT 表就至少应该有几个字段来存放各个附属表的变化对比时间。

建立 Data Vault 模型时应该参照如下的原则：

（1）关于中心表的原则

- 中心表的主键不能够直接“伸入”到其他中心表里面。也就是说，不存在父子关系的中心表，各个中心表之间的关系是平等的，这也正是 Data Vault 模型灵活性与扩展性之所在。
- 中心表之间必须通过链接表相关联，通过链接表可以连接两个以上的中心表。
- 至少有两个中心表才能产生一个有意义的链接表。
- 中心表的主键总是“伸出去”的（到链接表或者附属表）。

（2）关于链接表的原则

- 链接表可以跟其他链接表相连。
- 中心表和链接表都可以使用代理键。
- 业务主键从来不会改变，也就是说中心表的主键，即链接表的外键不会改变。

（3）关于附属表的原则

- 附属表必须连接到中心表或者链接表上才会有确定的含义。
- 附属表总是包含装载时间和失效时间，从而包含历史数据，并且没有重复的数据。
- 由于数据信息的类型或者变化频率快慢的差别，描述信息的数据可能会被分隔到多个附属表中去。

2.3.5 Data Vault 模型实例

下面用一个销售订单的例子说明如何将关系模型转换为 Data Vault 模型，以及如何向转换后的 Data Vault 模型装载数据。销售订单关系模型如图 2-5 所示，共有省、市、客户、产品类型、产品、订单、订单明细等 7 个表。

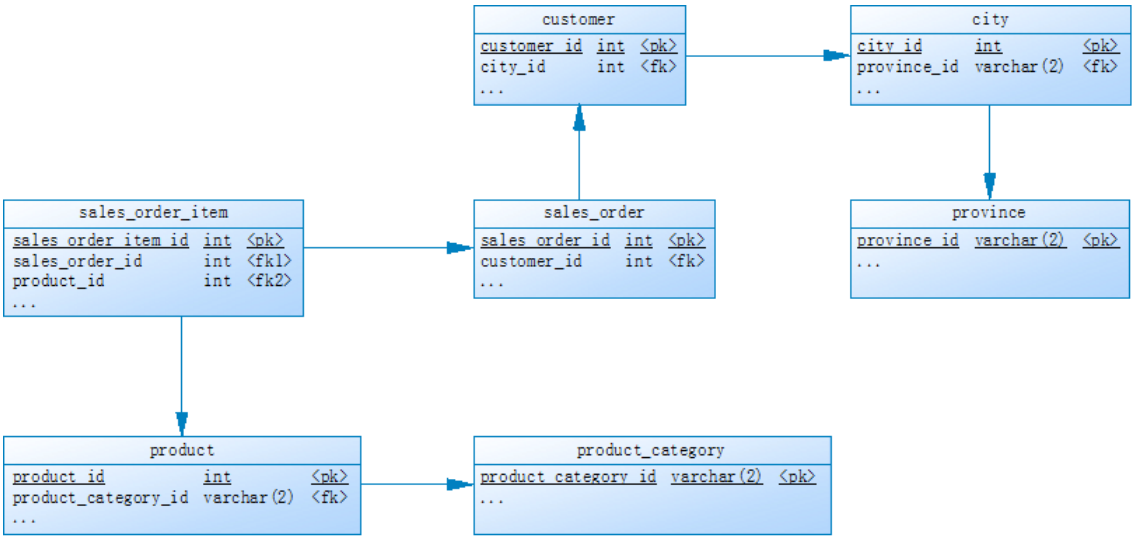

图 2-5　销售订单关系模型

1. 将关系模型转换为 Data Vault 模型

（1）转换中心表

转换中心表的具体步骤如下：

步骤 01 确定中心实体。示例中的客户、产品类型、产品、订单、订单明细这 5 个实体是订单销售业务的中心实体。省、市等地理信息表是参考数据，不能算是中心实体，实际上是附属表。

步骤 02 把第一步确定的中心实体中有入边的实体转换为中心表，因为这些实体被别的实体引用。把客户、产品类型、产品、订单转换成中心表。

步骤 03 把第一步确定的中心实体中没有入边且只有一条出边的实体转换为中心表。该示例中没有这样的表。

表 2-16 列出的是所有中心表。

表 2-16　销售订单中心表

实体	业务主键
hub_product_catagory	product_catagory_id
hub_customer	customer_id
hub_product	product_id
hub_sales_order	sales_order_id

每个中心表只有代理键、业务主键、装载时间、数据来源四个字段。在这个示例中，业务主键就是关系模型中表的主键字段。

（2）转换链接表

转换链接表的具体步骤如下：

步骤 01 把示例中没有入边且有两条或两条以上出边的实体直接转换成链接表。符合条件的是订单明细表。

步骤 02 把示例中除第一步以外的外键关系转换成链接表。订单和客户之间建立链接表，产品和产品类型之间建立链接表。

注　意
Data Vault 模型中的每个关系都是多对多关系。

表 2-17 列出的是所有链接表。

表 2-17　销售订单链接表

链接表	被链接的中心表
link_order_product	hub_sales_order、hub_product
link_order_customer	hub_sales_order、hub_customer
link_product_catagory	hub_product、hub_product_catagory

链接表中包含代理键、关联的中心表的一个或多个主键、装载时间、数据来源等字段。

（3）转换附属表

附属表为中心表和链接表补充属性。所有源库中用到的表的非键属性都要放到 Data Vault 模型的附属表中。

表 2-18 列出的是所有附属表。

表 2-18　销售订单附属表

附属表	所描述的表
sat_customer	hub_customer
sat_product_catagory	hub_product_catagory
sat_product	hub_product
sat_sales_order	hub_sales_order
sat_order_product	link_order_product

附属表中包含代理键、关联的中心表或链接表的主键、装载时间、失效时间、数据来源、关联的中心表或链接表所对应的关系模型表中的一个或多个非主键属性等字段。

转换后的销售订单 Data Vault 模型如图 2-6 所示。

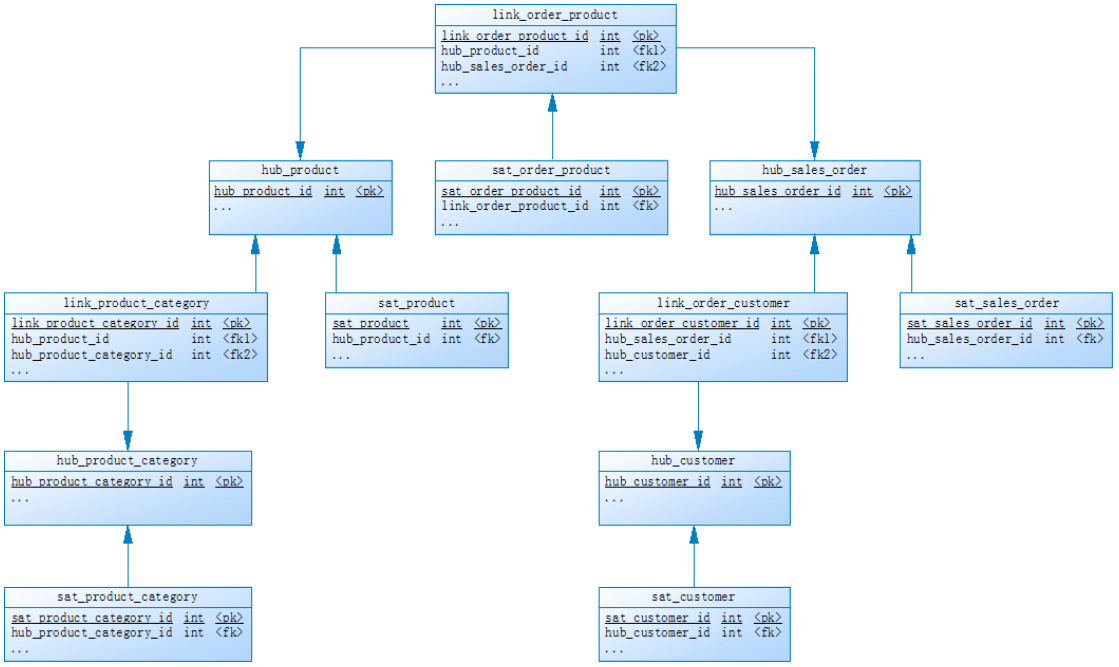

图 2-6　销售订单 Data Vault 模型

2. 向 Data Vault 模型的表中装载数据

现在 Data Vault 模型的中心表、链接表、附属表都已经建立好，需要向其中装载数据，数据的来源是关系模型中的表。假设 Data Vault 的表使用 MySQL 数据库建立，代理键使用自增列，装载时间使用时间戳数据类型。在插入数据时，这两列不用显式赋值，数据会自动维护。数据来源字段简单处理，就填写与之相关的表名。附属表的失效时间字段，初始值填写一个很大的默认时间，这里插入'2200-01-01'。

使用以下的 SQL 代码装载 hub_product 中心表、link_order_product 链接表、sat_order_product 附属表，其他表的装载语句类似，此处略。

```
-- 装载 hub_product 中心表
insert into hub_product (product_id,record_source)
select product_id,'product' from product;

-- 装载 link_order_product 链接表
insert into link_order_product(
      hub_sales_order_id,
      hub_product_id,
      record_source)
select hub_sales_order_id,
      hub_product_id,
      'hub_sales_order,hub_product,sales_order_item'
  from hub_sales_order t1,
      hub_product t2,
      sales_order_item t3
 where t1.sales_order_id = t3.sales_order_id
   and t2.product_id = t3.product_id;
```

```
-- 装载 sat_order_product 附属表
insert into sat_order_product (
      link_order_product_id,
      load_end_dts,
      record_source,
      unit_price,
      quantity)
select link_order_product_id,
      '2200-01-01',
'link_order_product,hub_sales_order,hub_product,sales_order_item',
      t4.unit_price,
      t4.quantity
  from link_order_product t1,
      hub_sales_order t2,
      hub_product t3,
      sales_order_item t4
 where t1.hub_sales_order_id = t2.hub_sales_order_id
   and t1.hub_product_id = t3.hub_product_id
   and t4.sales_order_id = t2.sales_order_id
   and t4.product_id = t3.product_id;
```

2.4 数据集市

在第 1 章中介绍了独立数据集市和从属数据集市两种架构，本节将继续讨论数据集市的概念、数据集市与数据仓库的区别、数据集市的设计等问题。

1. 数据集市的概念

数据集市是数据仓库的一种简单形式，通常由组织内的业务部门自己建立和控制。一个数据集市面向单一主题域，如销售、财务、市场等。数据集市的数据源可以是操作型系统（独立数据集市），也可以是企业级数据仓库（从属数据集市）。

2. 数据集市与数据仓库的区别

不同于数据集市，数据仓库处理整个组织范围内的多个主题域，通常是由组织内的核心单位，如 IT 部门承建，所以经常被称为中心数据仓库或企业数据仓库。数据仓库需要集成很多操作型源系统中的数据。数据集市的复杂度和需要处理的数据都小于数据仓库，因此更容易建立与维护。表 2-19 总结了数据仓库与数据集市的主要区别。

3. 数据集市设计

数据集市主要用于部门级别的分析型应用，数据大都经过了汇总和聚合操作，粒度级别较高。数据集市一般采用维度模型设计方法，数据结构使用星型模式或雪花模式。

表 2-19　数据仓库与数据集市的主要区别

对比项	数据仓库	数据集市
范围	企业级	部门级或业务线
主题	多个主题	单一主题
数据源	遗留系统、事务系统、外部数据的多个数据源	数据仓库或事务系统的少量数据源
数据粒度	较细的粒度	较粗的粒度
数据结构	通常是规范化结构（3NF）	星型模型、雪花模型或两者混合
历史数据	全部历史数据	部分历史数据
完成需要的时间	几个月到几年	几个月

正如前面所介绍的，设计维度模型先要确定维度表、事实表和数据粒度级别，下一步是使用主外键定义事实表和维度表之间的关系。数据集市中的主键最好使用系统生成的自增的单列数字型代理键。模型建立好之后，设计 ETL 步骤抽取操作型源系统的数据，经过数据清洗和转换，最终装载进数据集市中的维度表和事实表中。

2.5　数据仓库实施步骤

实施一个数据仓库项目的主要步骤是：定义项目范围，收集并确认业务需求和技术需求，逻辑设计，物理设计，从源系统向数据仓库装载数据，使数据可以被访问以辅助决策，管理和维护数据仓库。

1. 定义范围

在实施数据仓库前，需要制定一个开发计划。这个计划的关键输入是信息需求和数据仓库用户的优先级。当这些信息被定义和核准后，就可以制作一个交付物列表，并给数据仓库开发团队分配相应的任务。

首要任务是定义项目的范围。项目范围定义了一个数据仓库项目的边界。典型的范围定义是组织、地区、应用、业务功能的联合表示。定义范围时通常需要权衡考虑资源（人员、系统、预算等）、进度（项目的时间和里程碑要求）、功能（数据仓库承诺达到的能力）三方面的因素。定义好清晰明确的范围，并得到所有项目干系人的一致认可，这对项目的成功非常重要。项目范围是设定正确的期望值、评估成本、估计风险、制定开发优先级的依据。

2. 确认需求

数据仓库项目的需求可以分为业务需求和技术需求。

（1）定义业务需求

建立数据仓库的主要目的是为组织赋予从全局访问数据的能力。数据的细节程度必须能够满足用户执行分析的需求，并且数据应该被表示为用户能够理解的业务术语。对数据仓库中数据的分析将辅助业务决策，因此，作为数据仓库的设计者，应该清楚业务用户是如何做决策的，在决策过程中提出了哪些问题，以及哪些数据是回答这些问题所需要的。与业务人员进行

面对面的沟通，是理解业务流程的好方式，沟通的结果是使数据仓库的业务需求更加明确。在为数据仓库收集需求的过程中，还要考虑设计要能适应需求的变化。

（2）定义技术需求

数据仓库的数据来源是操作型系统，这些系统日复一日地处理着各种事务活动。操作型系统大都是联机事务处理系统，数据仓库会从多个操作型源系统抽取数据。但是，一般不能将操作型系统里的数据直接迁移到数据仓库，而是需要一个中间处理过程，这就是所谓的 ETL 过程。数据仓库需要知道如何清理操作型数据，如何移除垃圾数据，如何将来自多个源系统的相同数据整合在一起。另外，还要确认数据的更新频率。例如，如果需要进行长期的或大范围的数据分析，可能就不需要每天装载数据，而是每周或每月装载一次。

注　意

更新频率并不决定数据的细节程度，每周汇总的数据有可能每月装载（当然这种把数据转换和数据装载分开调度的做法并不常见）。

总之，在数据仓库设计的初始阶段，需要确定数据源有哪些、数据需要做哪些转换以及数据的更新频率是什么。

3. 逻辑设计

定义好了项目的范围和需求，就有了一个基本的概念设计。下面就要进入数据仓库的逻辑设计阶段。在逻辑设计过程中，需要定义特定数据的具体内容、数据之间的关系、支持数据仓库的系统环境等，其本质是发现逻辑对象之间的关系。

（1）建立需要的数据列表

细化业务用户的需求以形成数据元素列表。在很多情况下，为了得到所需的全部数据，需要适当扩展用户需求或者预测未来的需要，一般从主题域涉及的业务因素入手。例如，销售主题域的业务因素可能是客户、地区、产品、促销等。然后建立每个业务因素的元素列表，依据也是用户提出的需求。最后通过元素列表，标识出业务因素之间的联系。这些工作完成后，应该已经获得了如下信息：原始的或计算后的数据元素列表；数据的属性，比如是字符型的还是数字型的；合理的数据分组，比如国家、省市、区县等分成一组，因为它们都是地区元素；数据之间的关系，比如国家、省市、区县的包含关系等。

（2）识别数据源

现在已经有了需要的数据列表，下面的问题是从哪里可以得到这些数据，以及要得到这些数据需要多大的成本。此时需要把上一步建立的数据列表映射到操作型系统上。应该从最大最复杂的源系统开始，在必要时再查找其他源系统。数据的映射关系可能是直接的或间接的，比如销售源系统中，商品的单价和折扣价可以直接获得，而折扣百分比就需要计算得到。通常维度模型中的维度表可以直接映射到操作型源系统，而事实表的度量则映射到源数据在特定粒度级别上聚合计算后的结果。某些数据的获得需要较高的成本，例如，用户想要得到促销相关的销售数据就不那么容易，因为促销期的定义从时间角度看是不连续的。

（3）制作实体关系图

逻辑设计的交付物是实体关系图（Entity-Relationship Diagram，简称 ERD）和对它的说明

文档（数据字典）。实体对应关系数据库中的表，属性对应关系数据库中的列。ERD 传统上与高度规范化的关系模型联系密切，但该技术在维度模型中也被广泛使用。在维度模型的 ERD 中，实体由事实表和维度表组成，关系体现为在事实表中引用维度表的主键。因此，先要确认哪些信息属于中心事实表，哪些信息属于相关的维度表。维度模型中表的规范化级别通常低于关系模型中的表。

4. 物理设计

物理设计指的是将逻辑设计的对象集合，转化为一个物理数据库，包括所有的表、索引、约束、视图等。物理数据库结构需要优化以获得最佳性能。每种数据库产品都有自己特别的优化方法，这些优化对查询性能有极大的影响。比较通用的数据仓库优化方法有位图索引和表分区。

第 1.2.2 节中的“分析型系统的数据库设计”已经提到过位图索引和表分区。位图索引对索引列的每个不同值建立一个位图。和普通的 B 树索引相比，位图索引占用的空间小，创建速度快。但由于并发的 DML 操作会锁定整个位图段的大量数据行，所以位图索引不适用于频繁更新的事务处理系统。而数据仓库对最终用户来说是一个只读系统，其中某些维度的值基数很小，这样的场景非常适合利用位图索引优化查询。遗憾的是有些数据库管理系统如 MySQL，还没有位图索引功能。

大部分数据库系统都可以对表进行分区。表分区是将一个大表按照一定的规则分解成多个分区，每个表分区可以定义独立的物理存储参数。将不同分区存储到不同的磁盘上，查询表中数据时可以有效分布 I/O 操作，缓解系统压力。分区还有一个很有用的特性，叫作分区消除。在查询数据的时候，数据库系统的优化器可以通过适当的查询条件过滤掉一些分区，从而避免扫描所有数据，提高查询效率，这就是分区消除。

除了性能优化，数据仓库系统的可扩展性也非常重要。简单地说，可扩展性就是能够处理更大规模业务的特性。从技术上讲，可扩展性是一种通过增加资源，使服务能力得到线性扩展的能力。比方说，一台服务器在满负荷时可以为一万个用户同时提供服务，当用户数增加到两万时，只需要再增加一台服务器，就能提供相同性能的服务。成功的数据仓库会吸引越来越多的用户访问。随着时间的推移，数据量会越来越大，因此在做数据仓库物理设计时，出于可扩展性的考虑，应该把对硬件、软件、网络带宽的依赖降到最低。第 3 章会详细讨论数据仓库在 Greenplum 上的扩展性问题。

5. 装载数据

这个步骤实际上涉及整个 ETL 过程。需要执行的任务包括：在源和目标结构之间建立映射关系；从源系统抽取数据；对数据进行清洗和转换；将数据装载进数据仓库；创建并存储元数据。

6. 访问数据

访问步骤是要使数据仓库的数据可以被使用，使用的方式包括：数据查询、数据分析、建立报表图表、数据发布等。根据采用的数据仓库架构，可能会引入数据集市的创建。通常，最终用户会使用图形化的前端工具向数据库提交查询，并要求显示查询结果。访问步骤需要执行以下任务：

- 为前端工具建立一个中间层。在这个中间层里，把数据库结构和对象名转化成业务术语，这样最终用户就可以使用与特定功能相关的业务语言同数据仓库交互。
- 管理和维护这个业务接口。
- 建立和管理数据仓库里的中间表和汇总表。中间表一般是在原始表上添加过滤条件获得的数据集合，汇总表则是对原始表进行聚合操作后的数据集合。这些表中的记录数会远远小于原始表，因此前端工具在这些表上的查询会执行得更快。

7. 管理维护

这个步骤涵盖在数据仓库整个生命周期里的管理和维护工作中。这一步需要执行的任务包括：确保对数据的安全访问、管理数据增长、优化系统以获得更好的性能、保证系统的可用性和可恢复性等。

2.6 小 结

（1）关系模型、多维模型和 Data Vault 模型是三种常见的数据仓库模型。

（2）数据结构、完整性约束和 SQL 语言是关系模型的三个要素。

（3）规范化是通过应用范式规则实现的。第一范式（1NF）要求保持数据的原子性，第二范式（2NF）消除了部分依赖，第三范式（3NF）消除了传递依赖。关系模型的数据仓库一般要求满足 3NF。

（4）事实、维度、粒度是维度模型的三个核心概念。

（5）维度模型的四步设计法是：选择业务流程、声明粒度、确认维度和确认事实。

（6）星型模式和雪花模式是维度模型的两种逻辑表示。对星型模式进一步规范化，就形成了雪花模式。

（7）Data Vault 模型有中心表（Hub）、链接表（Link）、附属表（Satellite）三个主要组成部分。中心表记录业务主键，链接表记录业务关系，附属表记录业务描述。

（8）Data Vault 不区分数据在业务层面的正确与错误，它保留操作型系统的所有时间的所有数据，装载数据时不做数据验证、清洗等工作。

（9）数据集市是部门级的、面向单一主题域的数据仓库。

（10）数据集市的复杂度和需要处理的数据都小于数据仓库，因此更容易建立与维护。

（11）实施一个数据仓库项目的主要步骤是：定义范围、确认需求、逻辑设计、物理设计、装载数据、访问数据和管理维护。

第3章

Greenplum与数据仓库

Greenplum是一个分布式大规模并行处理数据库，在大多数情况下适合做大数据的存储引擎、计算引擎和分析引擎，尤其适合构建数据仓库。本章将重点介绍Greenplum的系统架构和主要功能。我们先从历史演进和所采用的MPP框架对Greenplum进行概要说明，然后描述其顶层架构，之后详细介绍其在存储模式、事务支持、并行查询与数据装载、容错与故障转移、数据库统计、过程化语言扩展等方面的功能特性，正是它们支撑Greenplum成为一款理想的分析型数据库产品。本章最后将简单对比Greenplum与另一个流行的大数据处理框架Hadoop，进而阐述选择前者的理由。希望读者通过阅读本章的内容，对Greenplum有一个基本认识，最重要的是理解为什么要使用它来建立数据仓库。

3.1 Greenplum简介

Greenplum是一个大规模并行SQL分析引擎，针对的是分析型应用。与其他关系数据库类似，它接收SQL，返回结果集。

3.1.1 历史与现状

Greenplum最早出现在2002年，比大名鼎鼎的Hadoop（约2004年面世）还要早一些。当时正值互联网行业经过近10年的由慢到快的发展，累积了大量数据。传统主机的向上扩展（Scale-up）模式在海量数据面前遇到了瓶颈，除造价昂贵外，在技术上也难于满足数据计算的性能需求。这种情况下急需一种新的计算方式来处理数据，于是分布式存储和分布式计算理论被提了出来，Google公司著名的GFS和MapReduce也从此引起业界的关注，可以支持向外扩展（Scale-out）的分布式并行数据计算技术登场了。Greenplum正是在这一背景下产生，它

借助于分布式计算思想，在流行的开源数据库 PostgreSQL 之上开发，实现了基于数据库的分布式数据存储和并行计算。

Greenplum 的名字据说源自创始人家门口的一棵青梅。初创公司召集了十几位业界大咖花了一年多的时间完成最初的版本设计和开发，用软件实现了在开放 X86 平台上的分布式并行计算，不依赖任何专有硬件，达到的性能却远远超过了传统高昂的专有系统。2006 年，当时的 Sun 微系统公司与 Greenplum 开始联手打造即时数据仓库。2010 年 EMC 收购了 Greenplum。2012 年 EMC、VMWare 和 Greenplum 又联手成立了新公司 Pivotal，之后由 Pivotal 公司商业运营 Greenplum 数据库。

Greenplum 于 2015 年 10 月开源，社区具有很高的知名度和热度，至今依然保持着几周发版的更新速度。2020 年 Pivotal 被兄弟公司 VMWare 收购，由 VMWare 继续运营商业产品，形成了商业 VMware Tanzu Greenplum 和开源 Greenplum 两条产品线。商业产品提供了比开源产品更多的功能，如与 EMC DD Boost、Symantec NetBackup 的整合，QuickLZ 压缩算法，替代过时的 gpcheck 的 gpsupport 实用程序等。

3.1.2 MPP——一切皆并行

Greenplum 采用无共享（Shared-Nothing）的大规模并行处理（MPP）架构，将实际的数据存储设备分成一个个段（Segment）服务器上的小存储单元，每个单元都有一个连接本地磁盘的专用独立的高带宽通道。各个段服务器可以通过完全并行的方式处理每个查询，同时使用所有磁盘连接，并按照查询计划的要求在各段间实现高效数据流动。Greenplum 基于这种架构可以帮助客户创建数据仓库（Greenplum 从开始设计的时候就被定义成数据仓库），充分利用低成本的商用服务器、存储和联网设备，通过经济的方式进行 PB 级数据运算，并且在处理 OLAP、BI 和数据挖掘等任务时其性能远超通用数据库系统。

并行工作方式贯穿了 Greenplum 功能设计的方方面面：外部表数据装载是并行的，查询计划执行是并行的，索引的建立和使用是并行的，统计信息收集是并行的，表关联（包括其中的数据重分布或广播及关联计算）是并行的，排序和分组聚合是并行的，备份恢复也是并行的，甚而数据库启停和元数据检查等维护工具也按照并行方式来设计。得益于这种无所不在的并行，Greenplum 在数据装载和数据计算中表现出强悍的性能。

Greenplum 建立在无共享架构上，让每一个 CPU 和每一块磁盘 I/O 都运转起来，无共享架构将这种并行处理发挥到极致。试想一台内置 16 块 SAS 盘的 X86 服务器，磁盘扫描性能在 2000MB/s 左右，20 台这样的服务器构成的集群 I/O 性能是 40GB/s，这样超大的 I/O 吞吐量是传统存储难以企及的。另外，Greenplum 还可以建立在 PostgreSQL 数据库实例级别上进行并行计算，可在一次 SQL 请求中利用到每个节点上多个 CPU 核的计算能力，对 X86 的 CPU 超线程有很好的支持，能提供更好的请求响应速度。

3.2　Greenplum 系统架构

Greenplum 是一个纯软件的 MPP 数据库服务器，其系统架构专门用于管理大规模分析型数据仓库或商业智能工作负载。从技术上讲，MPP 无共享架构是指具有多个节点的系统，每个节点都有自己的内存、操作系统和磁盘，它们协作执行一项操作。Greenplum 使用这种高性能系统架构分配 PB 级别的数据，并行使用系统的所有资源来处理请求。

3.2.1　Greenplum 与 PostgreSQL

Greenplum 6 版本基于 PostgreSQL 9.4 开源数据库，本质上是若干面向磁盘的 PostgreSQL 数据库实例，共同作为一个内聚的数据库管理系统（DBMS）。大多数情况下，Greenplum 在 SQL 支持、配置选项和最终用户功能方面与 PostgreSQL 非常相似。用户操作 Greenplum 数据库就像与常规 PostgreSQL 交互一样。

Greenplum 与 PostgreSQL 的主要区别为：

- 除了支持 Postgres 优化器外，还有自己的 GPORCA 优化器。
- Greenplum 数据库可以使用 Append-Optimized 存储格式。
- Greenplum 支持列存储，即逻辑上组织为表的数据，物理上以面向列的格式存储数据。列存储只能与 Append-Optimized 表一起使用。

Greenplum 对 PostgreSQL 的内部结构进行了修改和补充，以支持数据库的并行结构。例如，对系统目录、优化器、查询执行器和事务管理器组件做过修改和增强，能够在所有并行 PostgreSQL 数据库实例上同时运行查询。Greenplum 依赖 Interconnect（内部互连）在网络层支持不同 PostgreSQL 实例之间的通信，使得系统作为单一逻辑数据库运行。

较之标准 PostgreSQL，Greenplum 还增加了并行数据装载（外部表）、资源管理、查询优化和存储增强功能。Greenplum 开发的许多功能和优化也进入了 PostgreSQL 社区，促进了 PostgreSQL 的发展。例如，表分区是 Greenplum 首先开发的一个特性，现在已成为标准 PostgreSQL 的一部分。

Greenplum 顶层系统架构如图 3-1 所示。Master 是 Greenplum 数据库系统的入口，是客户端连接并提交 SQL 语句的数据库实例。Master 将其工作与系统中其他叫作 Segment 的数据库实例进行协调，这些数据库实例负责实际存储和处理用户数据。每个 Master 和 Segment 都是一个 PostgreSQL 数据库实例。

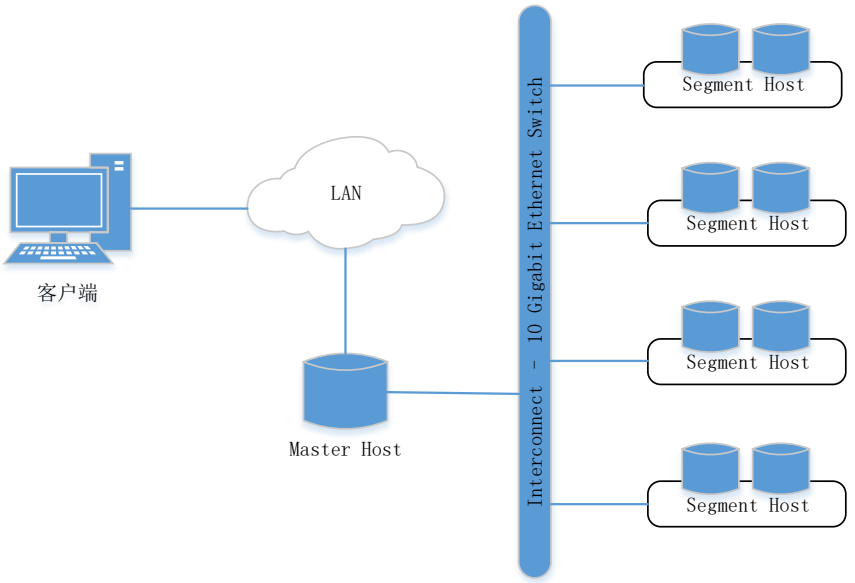

图 3-1　Greenplum 顶层系统架构

3.2.2　Master

Master 是 Greenplum 的系统入口，它接收客户端连接和 SQL 查询，并将工作分配给 Segment 实例。最终用户通过 Master 与 Greenplum 数据库交互，就像与典型 PostgreSQL 数据库交互一样。用户可以使用诸如 psql 之类的客户端程序或 JDBC、ODBC、libpq 之类的应用程序编程接口（API）连接到数据库。

Master 数据库实例中存储全局系统目录（Global System Catalog）。全局系统目录是一组系统表，其中包含关于 Greenplum 本身的元数据。Master 实例中不包含任何用户数据，用户数据仅驻留在 Segment 实例中。Master 验证客户端连接，处理传入的 SQL 命令，在 Segment 之间分配工作负载，协调每个 Segment 返回的结果，并将最终结果返给客户端程序。

Greenplum 数据库使用写前日志（WAL）进行主/备 Master 镜像。在基于 WAL 的日志记录中，所有修改都会在应用之前写入日志，以确保任何进程内操作的数据完整性。

3.2.3　Segment

Greenplum 的 Segment 实例是独立的 PostgreSQL 数据库，每个数据库存储一部分数据并执行一部分查询处理。当用户通过 Master 连接到数据库并发出查询时，将在每个 Segment 数据库中创建进程以处理该查询的工作。有关查询过程的更多信息，参见 3.3.3 节。

用户定义的表及其索引分布在所有可用的 Segment 中，每个 Segment 都包含互斥的部分数据（复制表除外，这种表会在每个 Segment 实例上存储一份完整的数据拷贝）。提供服务的数据库服务器进程在相应的 Segment 实例下运行。

Segment 在称为段主机的服务器上运行。一台段主机通常运行 2~8 个 Segment 实例，具体数量取决于 CPU 核、内存、磁盘、网卡和工作负载。所有段主机的配置应该相同，以避免木

桶效应。让 Greenplum 获得最佳性能的关键是将数据和负载均匀分布到多个能力相同的 Segment 上，以便所有 Segment 同时处理任务并同时完成其工作。

3.2.4　Interconnect

Interconnect 即内部互连，是 Greenplum 数据库系统架构中的核心组件，互连指的是 Segment 在网络间的进程间通信。Interconnect 使用标准以太网交换数据，出于性能原因，建议使用万兆网或更快的系统。

默认情况下，Interconnect 使用带有流量控制的用户数据报协议（UDPIFC）进行通信，通过网络发送消息。Greenplum 软件执行超出 UDP 提供的数据包验证，这意味着其可靠性相当于传输控制协议（TCP），性能和可扩展性超过 TCP。如果将 Interconnect 改为 TCP，Greenplum 数据库的可扩展性则限制为 1000 个 Segment 实例，UDPIFC 作为 Interconnect 的默认协议不受此限制。

Interconnect 实现了对同一集群中多个 PostgreSQL 实例的高效协同和并行计算，承载了并行查询计划生产、查询分派（QD）、协调节点上查询执行器（QE）的并行工作、数据分布、Pipeline 计算、镜像复制、健康探测等诸多任务。

3.3　Greenplum 功能特性

Greenplum 绝不仅仅只是“PostgreSQL + Interconnect 并行调度 + 分布式事务”这么简单，它还提供了许多高级数据分析管理功能和企业级管理模块。本节将主要介绍其中几个重要模块的特色功能。

3.3.1　存储模式

Greenplum 提供了几种灵活的存储模式。创建表时，可以通过设置本小节介绍的存储选项定义如何存储表数据，为工作负载选择最佳存储模式。为了简化建表时定义存储模式，可以通过 gp_default_storage_options 参数设置默认的存储选项。

1. Heap 存储

Greenplum 默认使用与 PostgreSQL 相同的堆（Heap）存储模型。堆表适用于 OLTP 类型的工作负载，数据通常在最初装载后进行修改。update 和 delete 操作需要存储行级别的版本控制信息以确保数据库事务处理的可靠性。堆存储适合小表，例如维度表，这些表通常在初始装载后进行更新。

行存堆表是默认的存储模式，建表时不需要额外语法：

```
-- 建表
create table foo (a int, b text) distributed by (a);

-- 查看表信息
```

```
\d foo
     Table "public.foo"
 Column|  Type   | Modifiers
-------+---------+-----------
 a     | integer |
 b     | text    |
Distributed by: (a)
```

Greenplum 6 版本中引入了全局死锁检测的新概念，以降低 update 和 delete 的锁级别。在 6 以前的版本中，update 和 delete 操作使用表级排它锁，也就是说，在 6 之前的版本中，一张表上同时只能有一个 update 或者 delete 语句被执行，其他的 update 或 delete 语句需要等待前面的语句执行完成之后才能获得所需要的锁。

从 6 版本开始，打开全局死锁检测后，堆表 update 和 delete 操作的锁将降低为行级排它锁，允许并发更新。全局死锁检测确定是否存在死锁，并通过取消一个或多个与最年轻事务相关联的后端进程来消除死锁。全局死锁检测由 gp_enable_global_deadlock_detector 参数控制，默认为 off。

```
$ gpconfig -s gp_enable_global_deadlock_detector
Values on all segments are consistent
GUC         : gp_enable_global_deadlock_detector
Master  value: off
Segment value: off
$ gpconfig -c gp_enable_global_deadlock_detector -v on
$ gpstop -arf
$ gpconfig -s gp_enable_global_deadlock_detector
Values on all segments are consistent
GUC         : gp_enable_global_deadlock_detector
Master  value: on
Segment value: on
```

另外，如果要进行高并发 insert、update、delete 操作，建议关闭 log_statement 参数（默认为 all），因为过多的日志输出也会影响这种操作的性能。

2. Append-Optimized 存储

Append-Optimized 存储表（简称 AO 表）适合于数据仓库环境中非规范化的事实表。事实表通常分批加载，并通过只读查询进行访问，是系统中最大的表。大型事实表采用 AO 存储可消除维护行级更新的多版本控制存储开销，每行可节省约 20 字节，这使得存储页结构更精简，更易于优化。而且 AO 表一般还会选择压缩选项，可以大大节省存储空间。AO 存储模型针对批量数据装载进行了优化，不建议使用单行 insert 语句。

通过 create table 的 with 子句定义存储选项，默认不指定 with 子句时，创建的是行存堆表。如果设置了 gp_default_storage_options 参数，存储模式与该参数的设置一致。下面是一个创建不带压缩选项的 AO 表的例子。

```
-- 建表
create table bar (a int, b text) with (appendoptimized=true) distributed by
(a);

-- 查看表信息
\d bar
```

```
Append-Only Table "public.bar"
 Column |  Type   | Modifiers
--------+---------+-----------
 a      | integer |
 b      | text    |
Compression Type: None
Compression Level: 0
Block Size: 32768
Checksum: t
Distributed by: (a)
```

appendoptimized 是以前 appendonly 的别称，在系统表中仍然存储 appendonly 关键字，显示存储信息时也将显示 appendonly。在可重复读或串行化隔离级事务中，不允许对 AO 表进行 update 或 delete。cluster、declare . . . for update 不适用于 AO 表。

3. 选择行存或列存

Greenplum 支持在 create table 时选择行存或列存，或者在分区表中为不同分区作不同选择，具体情况需要根据业务场景进行确切评估。建议绝大部分情况下选择行存，因为现在的列存技术容易导致文件数严重膨胀，后果更为严重。

从一般角度来说，行存具有更广泛的适用性，而列存对于一些特定场景可以节省大量 I/O 资源以提升性能，也可以提供更好的压缩效果。在考虑行存还是列存时可参考如下几点：

- 数据更新。如果一张表在数据装载后有频繁的更新操作，则选择行存堆表。列存表必须是 AO 表，所以没有别的选择。
- insert 频率。如果有频繁的 insert 操作，那么就选择行存表。列存表不擅长频繁地进行 insert 操作，因为在物理存储上列存表每一个字段都对应一个文件，频繁地进行 insert 操作将需要每次都写很多个文件。
- 查询涉及的列数。如果在 select 列表或 where 条件中经常涉及很多字段，选择行存表。列存表对于大数据量的单字段聚合查询表现更好，如：

```
    select sum(salary) ...
    select avg(salary) where salary > 10000
```

或者在 where 条件中使用单独字段进行条件过滤且返回相对少量的记录数，如：

```
select salary, dept ... where state='ca'
```

- 表中列数。当需要同时查询许多列，或者当表的行大小相对较小时，行存效率更高。对于列很多但只查询很少列时，列存表提供更好的查询性能。
- 压缩。列存表将具有相同数据类型的列数据连续存储在一起，因此对于相同的数据和压缩选项，往往列存的压缩效果更好，而行存无法具备这种优势。当然，越好的压缩效果意味着越困难的随机访问，因为数据读取都需要解压缩。不过 6 版本引入的 ZSTD 压缩算法具有非常优秀的压缩/解压缩效率。

下面语句创建一个不带压缩的列存表：

```
-- 建表
create table bar (a int, b text) with (appendoptimized=true, orientation=column)
```

```
distributed by (a);

    -- 查看表信息
    \d bar
    Append-Only Columnar Table "public.bar"
     Column|  Type   | Modifiers
    -------+---------+-----------
     a     | integer |
     b     | text    |
    Checksum: t
    Distributed by: (a)
```

4. 使用压缩（必须是 AO 表）

AO 表的压缩可以作用于整个表，也可以压缩特定列，可以对不同的列应用不同的压缩算法。表 3-1 总结了可用的压缩算法。

表 3-1 AO 表压缩算法

行或列	可用压缩类型	支持的压缩算法
行	表级	ZLIB、ZSTD、QUICKLZ（开源版本不可用）
列	列级或表级	RLE_TYPE、ZLIB、ZSTD、QUICKLZ（开源版本不可用）

选择 AO 表的压缩类型和级别时，需要考虑以下因素：

- CPU 性能。Segment 主机需要有足够的 CPU 资源进行压缩和解压缩。
- 压缩比/磁盘空间。磁盘空间占用最小化是一个因素，但也要考虑压缩和扫描数据所需的时间和 CPU 资源。我们需要找到有效压缩数据的最佳设置，从而不会导致过长的压缩时间或较慢的扫描速度。
- 压缩速度。QuickLZ 压缩通常使用较少的 CPU 资源，比 ZLIB 压缩速度快，但压缩率低。ZLIB 压缩率高，但压缩速度慢。例如，在压缩级别 1（compresslevel=1）下，QuickLZ 和 ZLIB 具有相当的压缩率，尽管速度不同。与 QuickLZ 相比，使用压缩级别为 6 的 ZLIB 可以显著提高压缩率，但压缩速度较低。ZSTD 则可以提供良好的压缩率或速度。
- 解压缩/扫描速度。压缩 AO 表的性能不仅取决于压缩选项，还与硬件、查询优化设置等因素有关。应该进行对比测试以确定合适的压缩选项。

不要在使用压缩的文件系统上创建压缩 AO 表，这样做只会来带额外的 CPU 开销。下面语句创建压缩级别为 5 的 ZLIB 压缩的 AO 表。

```
    -- 建表
    create table foo (a int, b text) with (appendoptimized=true, compresstype=zlib,
compresslevel=5);

    -- 查看表信息
    \d foo
    Append-Only Table "public.foo"
     Column |  Type   | Modifiers
    --------+---------+-----------
     a      | integer |
     b      | text    |
```

```
Compression Type: zlib
Compression Level: 5
Block Size: 32768
Checksum: t
Distributed by: (a)
```

5. 检查 AO 表的压缩与分布情况

Greenplum 提供了两个内置函数用以检查 AO 表的压缩率和分布情况。它们可以使用对象 ID 或表名作为参数，表名可能需要带模式名，如表 3-2 所示。

表 3-2　获取压缩 AO 表元数据的函数

函数	返回类型	描述
get_ao_distribution(name) get_ao_distribution(oid)	集合类型（dbid、tuplecount）	展示 AO 表的分布情况，每行对应 segid 和记录数
get_ao_compression_ratio(name) get_ao_compression_ratio(oid)	float8	计算 AO 表的压缩率。如果该信息未得到，将返回–1

压缩率得到的是一个常见的比值类型。例如，返回值 3.19 或 3.19:1 表示未压缩表的大小略大于压缩表大小的 3 倍。表的分布作为一组行返回，指示每个 Segment 上存储该表的记录数。例如，在一个有着四个 Segment 的系统上，dbid 范围为 0~3，函数返回类似下面的结果集。

```
=# select get_ao_distribution('lineitem_comp');
 get_ao_distribution
---------------------
(0,7500721)
(1,7501365)
(2,7499978)
(3,7497731)
(4 rows)
```

3.3.2　事务与并发控制

数据库管理系统中的并发控制机制使并发查询返回正确的结果，同时确保数据完整性。传统数据库的分布式事务使用两阶段锁协议，防止一个事务修改另一个并发事务读取的数据，并防止任何并发事务读取或写入另一个事务更新的数据，即读写相互阻塞。协调事务所需的锁会增加数据库争用，因而降低总体事务吞吐量。

Greenplum 沿用 PostgreSQL 多版本并发控制（MVCC）模型来管理堆表的并发分布式事务。使用 MVCC，每个查询都会取得一个查询启动时的数据快照。查询在运行时无法看到其他并发事务所做的更改，这可以确保查询所看到的是数据一致性视图。读取行的查询永远不会阻塞写入行的事务，写入行的查询不会被读取行的事务阻塞。与传统的使用锁来协调读写数据事务之间的访问相比，MVCC 允许更大的并发性。AO 表使用的并发控制模型与这里讨论的 MVCC 模型不同，它们适用于“一次写入，多次读取”的应用程序，这类应用从不或很少执行行级更新。

1. 快照

快照是在语句或事务开始时可见的一个行集，可确保查询在执行期间具有一致且有效的

数据视图。一个新事务开始时被分配一个唯一的事务 ID（XID），它是一个递增的 32 位整数。未包含在事务中的 SQL 语句被视为单语句事务，BEGIN 和 COMMIT 被隐式添加，效果类似于某些数据库系统（如 MySQL）中的自动提交。Greenplum 仅将 XID 值分配给涉及 DDL 或 DML 操作的事务，这些事务通常是唯一需要 XID 的事务。

当事务插入一行时，XID 与该行一起保存在 xmin 系统列中。当事务删除一行时，XID 保存在 xmax 系统列中。更新一行被视为先删除再插入，因此 XID 保存到已删除行的 xmax 和新插入行的 xmin。xmin 系统列和 xmax 系统列以及事务完成状态所确定的一系列事务，其中的行版本对当前事务是可见的。一个事务可以看到小于 xmin 的所有事务的执行结果（保证已提交），但不能看到任何大于或等于 xmax 的事务结果（未提交）。

对于多语句事务，还必须标识事务中插入行或删除行的命令，以便可以看到当前事务中前面语句所做的更改。cmin 系统列标识事务中的插入命令，cmax 系统列标识事务中的删除命令。命令标识仅在事务期间起作用，因此在事务开始时该值将重新从 0 开始累加。cmin 和 cmax 用于判断同一个事务内其他命令导致的行版本变更是否可见。

XID 是数据库实例的一个属性。每个 Segment 实例都有自己的 XID 序列，无法与其他 Segment 的 XID 进行比较。主机使用集群范围的会话 ID（gp_session_ID）与 Segment 协调分布式事务，Segment 维护分布式事务 ID 与其本地 XID 的映射。Master 使用两阶段提交协议在所有 Segment 之间协调分布式事务。如果事务在任何一个 Segment 上执行失败，它将在所有 Segment 上回滚。

可以通过 select 语句查看任意行的 xmin、xmax、cmin 和 cmax 系统列。

```
select xmin, xmax, cmin, cmax, * from tablename;
```

在 Master 上执行的查询返回的 XID 是分布式事务 ID。如果在单个 Segment 实例中运行该查询，那么 xmin 和 xmax 值将是该 Segment 的本地事务 ID。

Greenplum 将复制表（Replicated Table）的所有行分布到每个 Segment 上，因此每一行在每个 Segment 上都是重复的。每个 Segment 实例维护自己的 xmin、xmax、cmin 和 cmax 以及 gp_segment_id。Greenplum 不允许用户查询从 Master 访问复制表的这些系统列（将会得到一个字段不存在的错误信息），因为它们没有明确的单一值。

2. 事务 ID 回卷

如前所述，MVCC 模型使用 XID 来确定在查询或事务开始时哪些行可见。XID 是一个 32 位的整数，因此理论上 Greenplum 最大可以运行大约 42 亿个事务，之后 XID 将回卷重置。Greenplum 对 XID 使用模 2^{32} 的计算方式，这允许 XID 循环使用。对于任何给定的 XID，过去的 XID 大约有 20 亿，未来的 XID 大约有 20 亿。这里有个问题，当一行的版本持续存在了大约 20 亿个事务后，再循环使用时，该行的 XID 又从头开始计数，使它看似为一个新行。为了防止出现这种情况，Greenplum 有一种称为 Frozen XID 的特殊 XID，它比任何常规 XID 都要老。某一行的 xmin 必须在 20 亿次事务内替换为 Frozen XID，这也是 VACUUM 命令执行的功能之一。

每隔 20 亿个事务对数据库进行至少一次清理，就可以防止 XID 回卷。Greenplum 数据库监视 XID，并且在需要一次 VACUUM 操作时发出警告。当 XID 大部分不再可用，且在 XID 发生回卷之前，将发出警告：

```
    WARNING: database "database_name" must be vacuumed within
number_of_transactions transactions
```

发出警告时就需要一次 VACUUM 操作。如果没有执行所需的 VACUUM 操作，Greenplum 在 XID 发生回卷前且达到一个限度时，会停止创建新事务以避免可能的数据丢失，并发出以下错误：

```
    FATAL: database is not accepting commands to avoid wraparound data loss in
database "database_name"
```

有关从此错误中恢复的过程，参阅 https://docs.greenplum.org/6-17/admin_guide/managing/maintain.html#topic3__np160654。

服务器配置参数 xid_warn_limit 和 xid_stop_limit 控制何时显示这些警告和错误。xid_warn_limit 参数指定在 xid_stop_limit 之前多少个 XID 时发出警告。xid_stop_limit 参数指定在回卷发生之前多少个 XID 时发出错误并且不再允许创建新事务。

3. 事务隔离模式

SQL 标准描述了数据库事务并发运行时可能出现的三种现象：

- 脏读：一个事务可以从另一个并发事务中读取未提交的数据。
- 不可重复读：一个事务两次读取同一行得到不同的结果，因为另一个并发事务在这个事务开始后提交了更改。
- 幻读：在同一事务中执行两次查询可以返回两组不同的行，因为另一个并发事务添加了行。

SQL 标准定义了数据库系统可以支持的四个事务隔离级，以及每个级别下并发执行事务时所允许的现象，如表 3-3 所示。

表 3-3　SQL 事务隔离级

隔离级	脏读	不可重复读	幻读
Read Uncommitted	可能	可能	可能
Read Committed	不可能	可能	可能
Repeatable Read	不可能	不可能	可能
Serializable	不可能	不可能	不可能

Greenplum 默认的事务隔离级为 Read Committed，由 default_transaction_isolation 参数指定。Greenplum 的 Read Uncommitted 和 Read Committed 隔离级的行为类似于 SQL 标准的 Read Committed 级别，Serializable 和 Repeatable Read 隔离级的行为类似于 SQL 标准的 Repeatable Read 级别，只是还防止了幻读。

Read Committed 和 Repeatable Read 之间的区别在于，前者事务中的每个语句只能看到在语句启动之前提交的行，而后者事务中的语句只能看到在事务启动之前提交的行。在 Read Committed 隔离级下，如果另一个并发事务自事务开始以来已提交更改，则在事务中两次相同查询得到的数据可能不同。Read Committed 模式还允许幻读，在同一事务中运行两次查询可以返回两组不同的行。

Greenplum 的 Repeatable Read 隔离级可避免不可重复读和幻读。试图修改由另一个并发

事务修改的数据的事务将被回滚。如果应用程序不需要 Repeatable Read 隔离模式，则最好使用 Read Committed 模式以提高并发性。Greenplum 不保证并发运行的一组事务产生与串行化顺序执行相同的结果。若指定了 Serializable 隔离级，Greenplum 数据库将返回到 Repeatable Read。

对于 Greenplum 的并发事务，应检查并识别可能并发更新相同数据的事务。对识别出来的问题，可以通过使用显式的表锁，或要求冲突的事务更新一个虚行（该虚行表示冲突），来防止该问题发生。SQL 语句 set transaction isolation level 可以设置当前事务的隔离模式，必须在执行 select、insert、delete、update 或 copy 语句前设置。

```
begin;
set transaction isolation level repeatable read;
...
commit;

-- 隔离模式也可以指定为 BEGIN 语句的一部分
begin transaction isolation level repeatable read;
```

4. 删除过期行

更新或删除行会在表中保留该行的过期版本，当表中过期的行累积后，必须扩展磁盘文件以容纳新行。由于运行查询所需的磁盘 I/O 增加，性能下降——这种情况称为膨胀（bloat），应该通过定期清理表来进行管理。当过期的行不再被任何活动事务引用时，可以删除该行并重新使用它占用的空间。VACUUM 命令将过期行使用的空间标记为可重用。

不带 FULL 的 VACUUM 命令可以与其他查询同时运行。它会标记之前被过期行所占用的空间为空闲，并更新空闲空间映射。当需要空间分配给新行时，Greenplum 首先会查询该表的空闲空间映射，寻找有可用空间的页。如果没有找到这样的页，会为该文件追加新页。

不带 FULL 的 VACUUM 不会合并页或者减小表在磁盘上的尺寸。它回收的空间只是放在空闲空间映射中表示可用。为了防止磁盘文件大小增长，经常运行 VACUUM 非常重要。运行 VACUUM 的频率取决于表中数据更新和删除的频率（插入只会增加新行）。大量更新的表可能每天需要运行几次 VACUUM，以确保通过空闲空间映射能找到可用空间。在一个更新或者删除大量行的事务之后，运行 VACUUM 也非常重要。

VACUUM FULL 命令会把表重写为没有过期行，并且将表减小到其最小尺寸。表中的每一页都会被检查，其中的可见行被移动到前面还没有完全填满的页中，空页会被丢弃。表会被一直锁住直到 VACUUM FULL 完成。相对于常规 VACUUM 命令来说，VACUUM FULL 是一种非常昂贵的操作，可以用定期清理来避免或者推迟这种操作。最好是在一个维护期中运行 VACUUM FULL。一种替代方案是，用一个 CREATE TABLE AS 语句重新创建表并且删除掉旧表。

可以运行 VACUUM VERBOSE tablename 得到一份 Segment 上已移除的过期行数量、受影响页数以及可用空闲空间页数的报告。查询 pg_class 系统表可以找出一个表在所有 Segment 上使用了多少页，查询前应先对表执行 ANALYZE 确保得到准确的数据。

```
select relname, relpages, reltuples from pg_class where relname='tablename';
```

另一个有用的工具是 gp_toolkit 模式中的 gp_bloat_diag 视图，它通过比较表使用的实际页数与预期页数来鉴别表膨胀。

5. 管理事务 ID 示例

下面看一个 Greenplum 官方文档中提供的示例。这个简单的例子说明了 MVCC 的概念以及它如何使用 XID 管理数据和事务，展示的概念如下：

- 如何使用 XID 管理表上的多个并发事务。
- 如何使用 Frozen XID 管理 XID。
- 模计算如何根据 XID 确定事务的顺序。

示例有如下假设：

- 该表是一个包含 2 列和 4 行数据的简单表。
- 有效的 XID 值从 0 到 9，9 之后，XID 将在 0 处重新启动。
- Frozen XID 为−2（与 Greenplum 数据库里的实际值不同）。
- 事务在一行上执行。
- 仅执行插入和更新操作。
- 所有更新的行都保留在磁盘上，不执行删除过期行的操作。

表的初始数据如表 3-4 所示，xmin 的顺序即为行插入的顺序。

表 3-4　初始示例表

Item	amount	xmin	xmax
Widget	100	0	null
Giblet	200	1	null
Sprocket	300	2	null
Gizmo	400	3	null

表 3-5 显示了对 amount 列执行如下更新后的表数据。

- xid = 4**: update** tbl **set** amount=208 **where** item = 'widget'
- xid = 5**: update** tbl **set** amount=133 **where** item = 'sprocket'
- xid = 6**: update** tbl **set** amount=16 **where** item = 'widget'

粗体表示当前行，其他是过期行。可以通过 xmax 为 null 条件确定表的当前行（Greenplum 使用了稍微不同的方法来确定当前表行）。

表 3-5　更新示例表

item	amount	xmin	xmax
widget	100	0	4
giblet	**200**	**1**	**null**
sprocket	300	2	5
gizmo	**400**	**3**	**null**
widget	208	4	6
sprocket	**133**	**5**	**null**
widget	**16**	**6**	**null**

MVCC 使用 XID 值确定表的状态。例如下面两个独立事务同时运行。

- UPDATE 命令将 sprocket 数量值更改为 133（xmin 值为 5）。
- SELECT 命令返回 sprocket 的值。

在 UPDATE 事务期间，查询返回 300，直到 UPDATE 事务完成。对于这个简单的示例，数据库的可用 XID 值即将用完。当 Greenplum 即将用完可用的 XID 值时将执行以下操作：发出警告，指出数据库的 XID 值即将用完；在分配最后一个 XID 之前，Greenplum 停止接收事务，以防止两个事务分配同一 XID 值，并发出错误警告。

为了管理存储在磁盘上的 XID 和表数据，Greenplum 提供了 VACUUM 命令。在示例表上执行 VACUUM 会释放 XID 值，以便通过将 xmin 值更改为 Frozen XID，使表可以容纳 10 行以上。VACUUM 命令还会更改 XID 值为 obsolete 以指示过期行。Greenplum 中不带 FULL 的 VACUUM 操作会标记已经删除的行，并且对性能和数据可用性影响最小。

表 3-6 显示的是在示例表上执行 VACUUM 操作后的情况，该命令更新了磁盘上的表数据。对于磁盘上不再是当前的 widget 行和 sprocket 行标记为过时。对于当前的 giblet 和 gizmo 行，xmin 已更改为 Frozen XID，这些值仍然是当前表值（行的 xmax 值为 null）。这两行对所有事务都可见，因为当执行模计算时，xmin 值是 Frozen XID，比所有其他 XID 值都小。VACUUM 操作后，XID 值 0、1、2 和 3 可供使用。这里显示的执行方式与 Greenplum 中的 VACUUM 命令略有不同，但概念相同。

表 3-6　VACUUM 后的示例表

item	Amount	xmin	Xmax
widget	100	obsolete	obsolete
giblet	200	-2	null
sprocket	300	obsolete	obsolete
gizmo	400	-2	null
widget	208	4	6
sprocket	133	5	null
widget	16	6	null

当更新 xmin 值为−2 的磁盘行时，xmax 值会像往常一样替换为 XID，并且在访问该行的任何并发事务完成后，磁盘上的行将被视为过期，可以从磁盘删除过期行。对于 Greenplum 数据库，带有 FULL 选项的 VACUUM 执行回收磁盘空间的操作。

表 3-7 显示的是更多更新事务后磁盘上的表数据。XID 值已回卷并在 0 处重新开始，没有再执行 VACUUM 操作。

表 3-7　回卷 XID 的示例表

item	amount	xmin	xmax
widget	100	obsolete	obsolete
giblet	200	−2	1
sprocket	300	obsolete	obsolete
gizmo	400	−2	9
widget	208	4	6

（续表）

item	amount	xmin	xmax
sprocket	133	5	null
widget	16	6	7
widget	222	7	null
giblet	233	8	0
gizmo	18	9	null
giblet	88	0	1
giblet	44	1	null

3.3.3　并行查询

理解 Greenplum 的查询处理过程有助于写出更加优化的 SQL 语句。与其他数据库管理系统类似，Greenplum 有如下查询步骤：

步骤 01 用户使用客户端程序（如 psql）连接到 Master 实例，并向系统提交 SQL 语句。

步骤 02 Master 接收到查询后，由查询编译器解析接收到的 SQL 语句，并将生成的查询解析树递交给查询优化器。

步骤 03 查询优化器根据查询所需的磁盘 I/O、网络流量等成本信息，生成它认为最优的执行计划，并将查询计划交给查询分发器。

步骤 04 查询分发器向 Segment 分发查询计划。

步骤 05 查询执行器并行执行查询，将结果传回至 Master，最后 Master 向客户端返回结果。

1. 计划分发

Master 负责接收、解析和优化查询，它将并行查询计划分发给所有的 Segment，如图 3-2 所示。

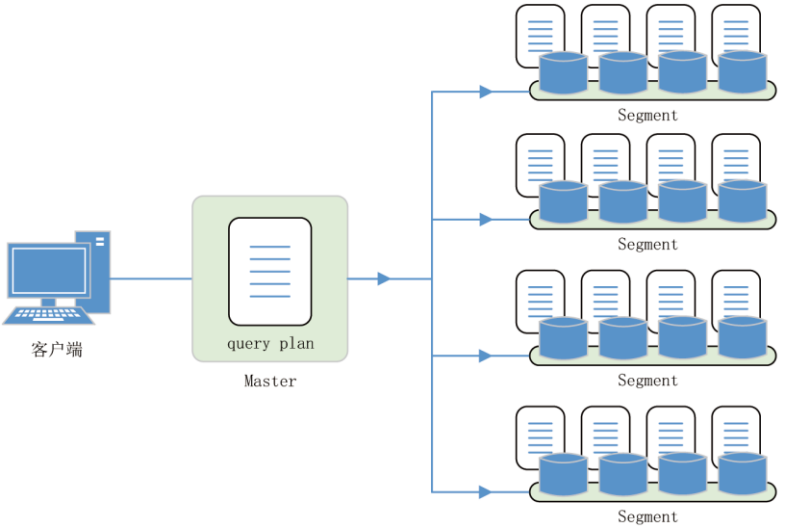

图 3-2　分发并行查询计划

每个 Segment 负责在其自己的数据集上执行本地数据库操作。大多数操作，如表扫描、连接、聚合和排序等在所有 Segment 上并行，每个操作都在一个 Segment 数据库实例上执行，与存储在其他 Segment 中的数据无关。

某些查询可能仅访问单个 Segment 上的数据，例如单行插入、更新、删除或对表分布键列进行过滤的查询。在此类查询中，查询计划不会分发给所有 Segment，而是只发给包含受影响行的 Segment，如图 3-3 所示。

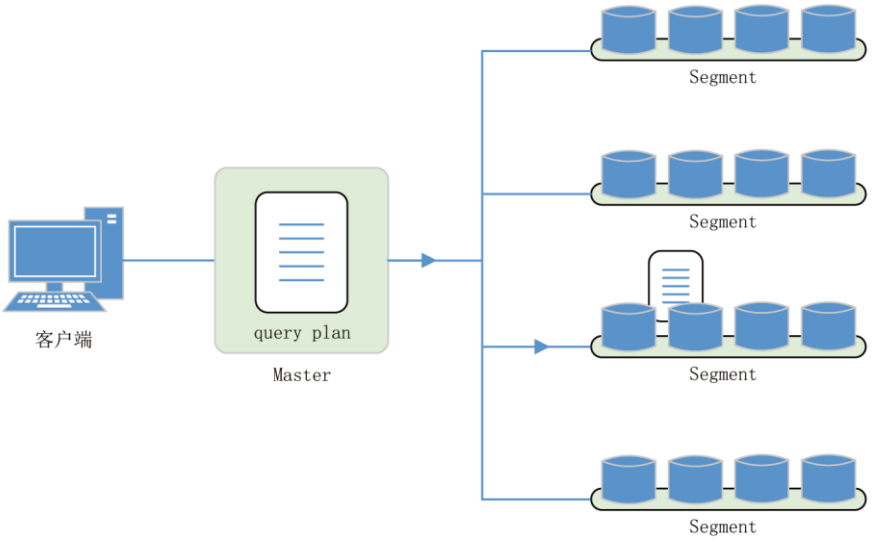

图 3-3　分发目标查询计划

2. 查询计划

一个查询计划是 Greenplum 为了产生查询结果而要执行的一系列操作。查询计划中的每个节点或步骤，表示一个数据库操作。查询计划由下向上被读取和执行。

除了通常的扫描、连接、聚合、排序等数据库操作，Greenplum 还有一种叫作 motion 的操作类型。查询处理期间，motion 操作通过内部互联网络在 Segment 实例间移动数据。并不是每个查询都需要 motion 操作。为了实现查询执行的最大并行度，Greenplum 将查询计划分成多个 slice，每个 slice 可以在 Segment 上独立执行。查询计划中的 motion 操作总是分片的，迁移数据的源和目标上各有一个 slice。

下面的查询连接两个数据库表。

```
select customer, amount
  from sales join customer using (cust_id)
 where datecol = '04-30-2021';
```

图 3-4 显示了为该查询生成的 3 个 slice。每个 Segment 接收一份查询计划的拷贝，查询计划在多个 Segment 上并行工作。

注意 slice 1 中的 Redistribute Motion 操作，它在 Segment 间移动数据以完成表连接。假设 customer 表通过 cust_id 字段在 Segment 上分布，而 sales 表通过 sale_id 字段分布。为了连接两个表，sales 的数据必须通过 cust_id 重新分布。因此查询计划在每个分片上各有一个 Redistribute Motion 操作。

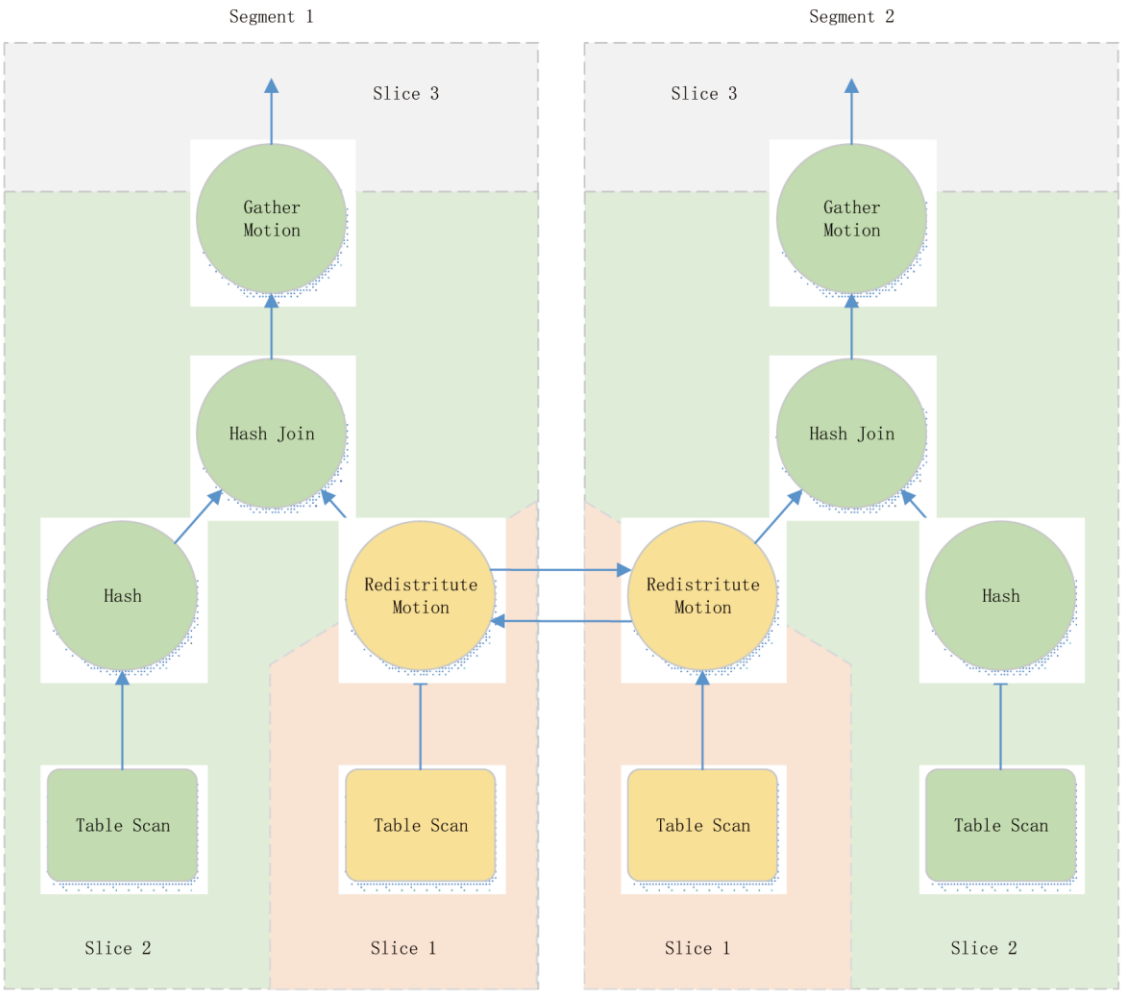

图 3-4　查询计划分片

在这个执行计划中还有一种叫作 Gather Motion 的操作。当 Segment 将查询结果发送回 Master，用于向客户端展示时，会使用 Gather Motion。因为查询计划中发生 motion 的部分总是被分片，所以在图 3-4 所示的顶部还有一个隐含的 slice 3。并不是所有查询计划都包含 Gather Motion，例如，CREATE TABLE x AS SELECT. . . 语句就没有 Gather Motion 操作，因为结果数据被发送到新表而不是 Master。

3. 并行查询

Greenplum 会创建许多数据库进程处理一个查询。Master 和 Segment 上的查询工作进程分别被称为查询分发器（QD）和查询执行器（QE）。QD 负责创建和分发查询计划，并返回最终的查询结果。QE 在 Segment 中完成实际的查询工作，并与其他工作进程互通中间结果。

查询计划的每个 slice 至少需要一个工作进程。工作进程独立完成被赋予的部分查询计划。一个查询执行时，每个 Segment 中有多个并行执行的工作进程。工作在不同 Segment 中的相同 slice 构成一个 gang。查询计划被从下往上执行，一个 gang 的中间结果数据向上流向下一个 gang。不同 Segment 的进程间通信是由 Greenplum 的内部互联组件 Interconnect 完成的。

图 3-5 显示了示例查询中 Master 和 Segment 上的工作进程，查询计划分成了 3 个 slice，两个 Segment 上的相同 slice 构成了 gang。

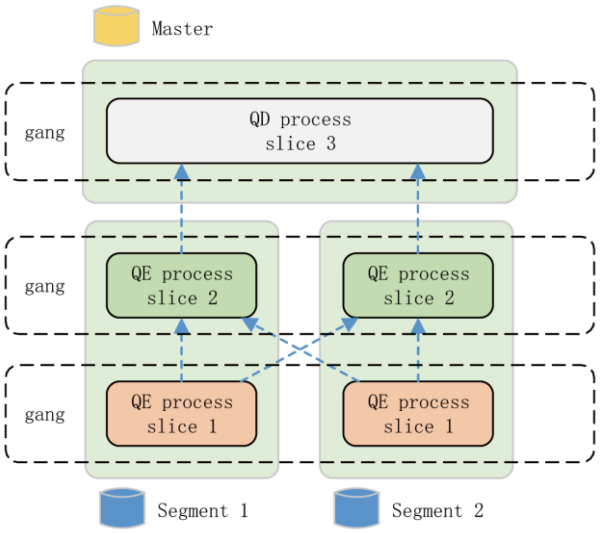

图 3-5　查询工作进程

4. GPORCA 查询优化器

Greenplum 默认使用的查询优化器是 GPORCA，但遗留的老优化器 Postgres 与 GPORCA 并存。GPORCA 在处理分区表查询、子查询、通用表表达式（CTE）、INSERT 语句、去重聚合等方面做了增强和改进。Greenplum 尽可能使用 GPORCA 生成查询的执行计划，当 GPORCA 没有启用或无法使用时，Greenplum 用老的查询优化器生成执行计划。可以通过 EXPLAIN 命令的输出确定查询使用的是哪种优化器。GPORCA 会忽略与老优化器相关的服务器配置参数，但当查询使用老优化器时，这些参数仍然影响查询计划的生成。相对于老优化器，GPORCA 在多核环境中的优化能力更强，并且在分区表查询、子查询、连接、排序等操作上提升了性能。图 3-6 显示的是 Greenplum 查询优化器。

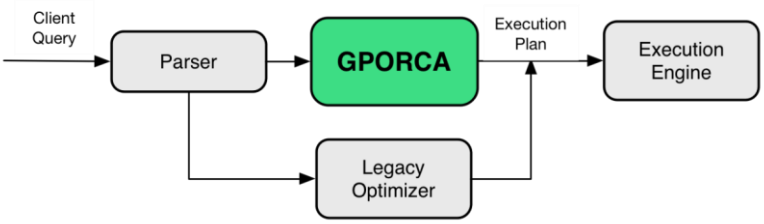

图 3-6　Greenplum 查询优化器

3.3.4　并行数据装载

在大型数据仓库中，必须在相对较小的维护时间窗口内装载大量数据。Greenplum 通过其外部表功能支持快速并行数据装载。用户还可以在单行错误隔离模式下装载外部表，以便在继

续装载格式正确的行的同时，将坏行过滤到单独的错误日志中。可以为装载操作指定错误阈值，以控制导致 Greenplum 取消装载操作的错误行数。通过将外部表与 Greenplum 的并行文件服务器 gpfdist 结合使用，可以从 Greenplum 系统获得最大的并行性和吞吐量，如图 3-7 所示。

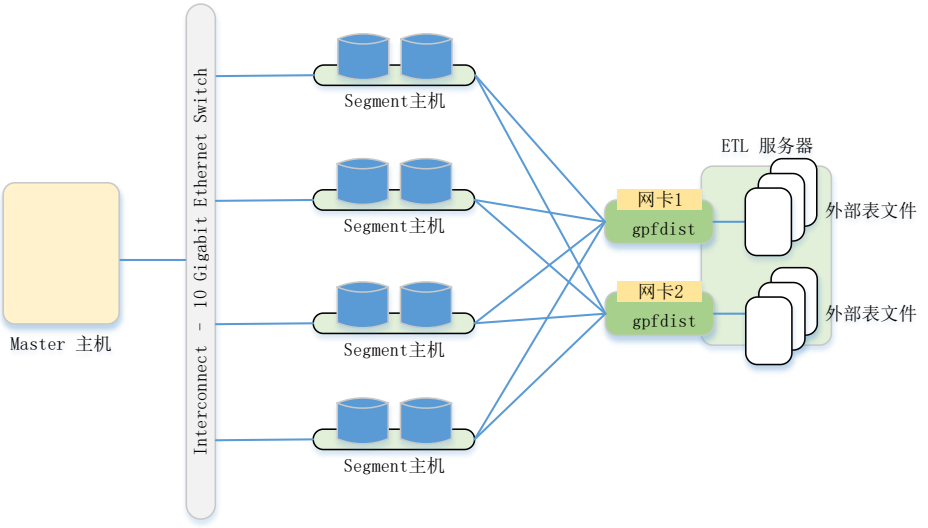

图 3-7　使用 gpfdist 的外部表

gpfdist 是 Greenplum 提供的一种文件服务器，它提供了良好的性能并且非常容易运行。gpfdist 利用 Greenplum 系统中的所有 Segment 读写外部表。gp_external_max_segs 配置参数控制可被单一 gpfdist 实例同时使用的 Segment 数量，默认值为 64。

另一个 Greenplum 实用程序 gpload 运行在 YAML 格式的控制文件中指定的装载任务中，可以在控制文件中描述源数据位置、格式、所需转换、参与主机、目标数据库以及其他详细信息。通过这种方式可以定义一个复杂的任务，并以可控、可重复的方式运行。

我们将在第 9 章“Greenplum 运维与监控”中详细介绍 gpfdist 和 gpload 技术，并给出具体示例。

3.3.5　冗余与故障转移

Greenplum 可以配置为高可用，以使得数据库集群更可靠地运行。如果不能接受数据丢失，Greenplum 要求 Mater 和 Segment 实例都必须开启高可用配置。也就是说，高可用配置不仅仅是可靠性的保证，也是数据安全的保证。Greenplum 支持为 Master 配置 Standby，为 Segment 配置 Mirror，以确保数据库中的每个角色都有备份，而不会出现单点故障。

1. Segment 镜像

部署 Greenplum 系统时，可以选择配置 Mirror Segment 实例，Mirror 允许数据库查询在 Primary Segment 不可用时自动切换到 Mirror 上。强烈建议在生产系统上配置 Mirror。Mirror 由事务日志复制过程保持最新，该过程同步 Primary 实例和 Mirror 实例之间的数据。Primary 向 Mirror 同步数据的时候，Primary 对于每一次写数据页都会通过消息发送到 Mirror。如果一个 Primary 无法向其 Mirror 发送数据，Primary 会把数据放入队列，超过

gp_segment_connect_timeout（默认为 10 分钟）后认为 Mirror 故障，从而将对应的 Mirror 标记为 down，而该 Primary 则变为更改跟踪模式。

Mirror 和 Primary 实例必须始终位于不同的主机上，以防止单个主机出现故障。在 Greenplum 初始化或扩容时，有两种可用的标准 Mirror 配置。默认配置为组镜像（Group Mirroring），将一台主机上所有 Primary Segment 的 Mirror 放置在群集中的另一台主机上，如图 3-8 所示。

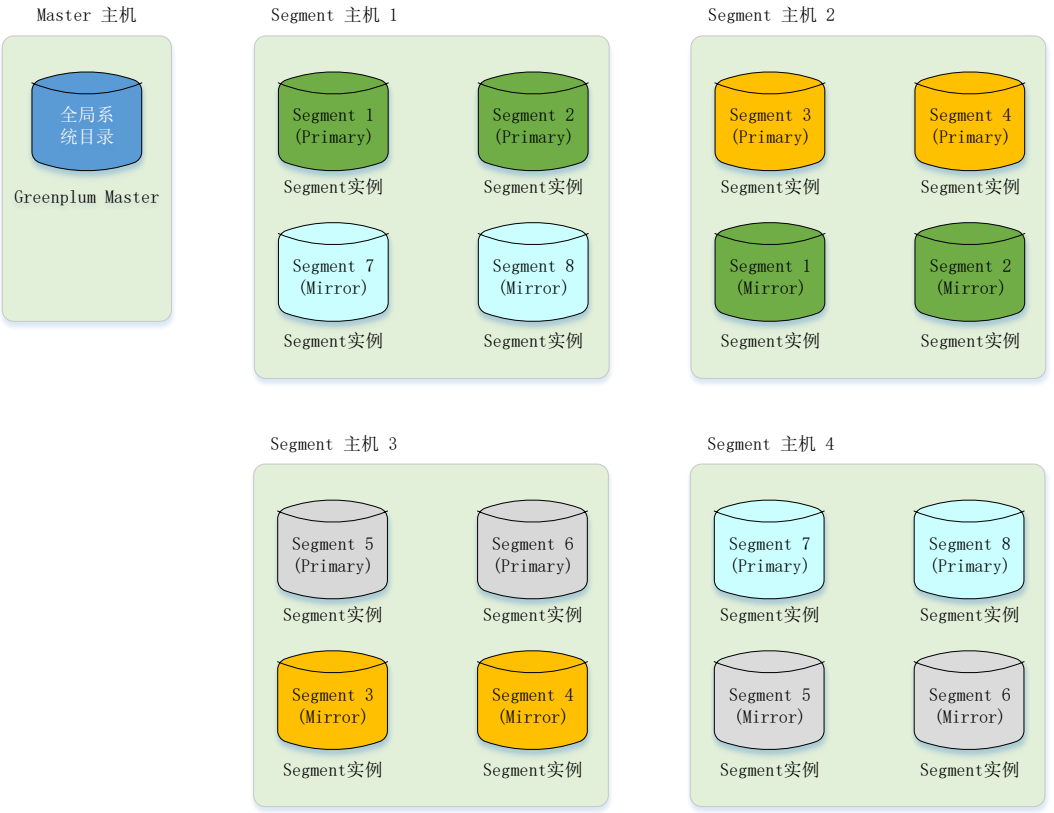

图 3-8　Segment 组镜像

另一种标准配置是扩展镜像（Spread Mirroring），将每个主机 Primary Segment 的 Mirror 扩展到其余主机上。这显然要求群集中的主机数多于每个主机的 Primary Segment 数。图 3-9 显示的是配置扩展镜像时如何分布 Segment 数据。

2. Segment 故障切换与恢复

在 Greenplum 中启用 Mirror Segment 时，如果 Primary 实例或所在主机宕掉，系统将自动切换到相应的 Mirror 实例，只要剩余 Segment 实例上的所有数据可用，Greenplum 数据库系统即可保持运行。

当无法连接到 Primary Segment 时，会在 Greenplum 系统目录中将该 Primary 实例标记为 down，并自动用其 Mirror 替换失效的 Primary 以继续提供服务。发生故障的 Segment 将停止运行，直到采取人为步骤使它重新联机。可以在系统启动和运行时恢复故障 Segment，恢复过程仅复制 Segment 停止运行期间丢失的增量差异数据。如果发生故障时有正在执行的事务，则

该事务将回滚并在新的 Segment 上自动重新启动。gpstate 程序可用于识别发生故障的 Segment，该程序显示系统目录表 gp_segment_configuration 中的信息。

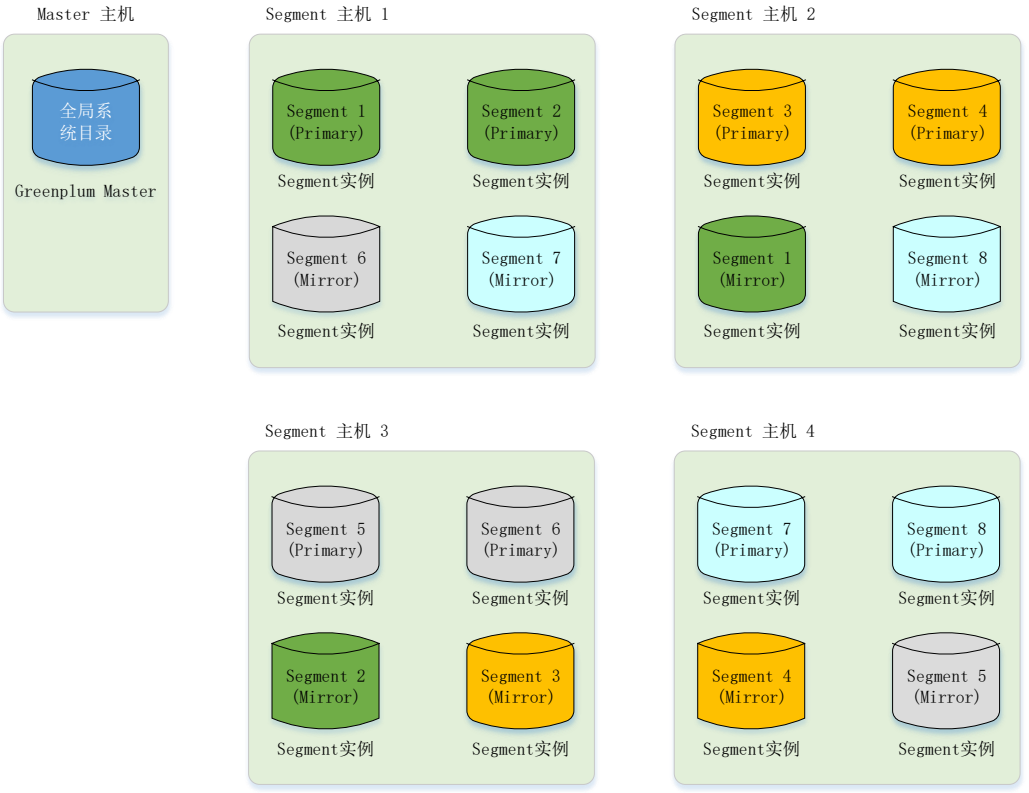

图 3-9　Segment 扩展镜像

如果由于 Segment 故障而导致整个 Greenplum 数据库无法运行（例如未启用 Mirror 或没有足够的 Segment 联机以访问所有用户数据），系统将自动关闭。客户端在尝试连接到数据库时会看到类似下面的错误，此时必须先恢复所有失败的 Segment，然后才能继续操作。

```
ERROR: All segment databases are unavailable
```

3. Master 镜像

如同 Primary 需要 Mirror 一样，可以在另一台主机上为 Master 实例配置一个镜像，按照惯例将其称为 Standby。Standby 是一个纯粹的容错节点，只作为 Master 出现问题时的热备，并要求与 Master 配置相同端口。在 Master 健康时，客户端只能通过 Master 来建立连接并执行 SQL 命令。当 Master 发生故障无法继续提供服务时，需要人为在 Standby 主机上执行 gpactivatestandby 命令来激活 Standby 作为新的 Master。到目前为止，Greenplum 没有实现 Standby 的自动激活。

可以为 Mater 和 Standby 配置同一个虚 IP，用于在 Master 和 Standby 之间漂移。当 Standby 被激活时，将虚 IP 漂移到 Standby 所在主机，这样客户端不需要修改连接信息就可以继续访问数据库。

Greenplum 中的 Master 节点镜像架构如图 3-10 所示。

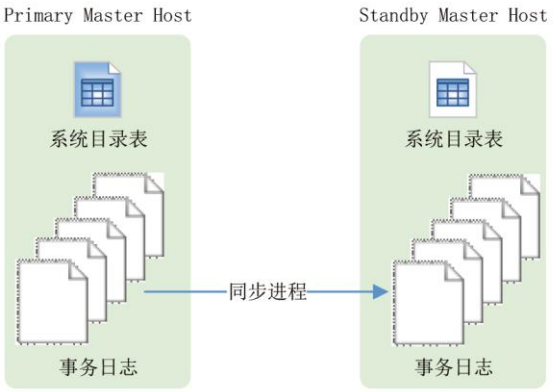

图 3-10　Master 镜像

Standby 通过 WAL 同步保持与 Master 的实时一致。由于 Master 不存储用户数据，在 Master 和 Standby 之间仅同步系统表数据。这些表的数据量与用户数据相比很小，并且较少发生变化。当这些系统表数据被更新时（如 DDL 所引起），就会自动同步到 Standby 从而保证与 Master 的一致性，所以 Standby 与 Master 可以保持实时同步。

Master 失效时，WAL 同步进程会自动停止。在 Standby 被激活时，冗余的 WAL 日志会被用来将数据库状态恢复到最后成功提交事务时的状态。激活的 Standby 实际上会成为 Greenplum 的新 Master，接收客户端的连接访问。一旦 Standby 被激活，旧的失败 Master 将脱离集群，如果要将其重新加入集群，需要使用 gpinitstandby 命令将其添加为 Standby 角色。

3.3.6　数据库统计

统计信息指的是数据库中所存储数据的元信息描述。查询优化器需要依据最新的统计信息为查询生成最佳执行计划。例如查询连接了两个表，一个表必须被广播到所有 Segment，那么优化器会选择广播其中的小表，使网络流量最小。

1. ANALYZE

ANALYZE 命令计算优化器所需的统计信息，并将结果保存到系统目录中。有三种方式启动分析操作：

- 直接运行 ANALYZE 命令。
- 在数据库外运行 analyzedb 命令行实用程序。
- 执行 DML 操作的表上没有统计信息，或者 DML 操作影响的行数超过了指定的阈值时，系统自动执行分析操作。

计算统计信息会消耗时间和资源，因此 Greenplum 会以对大表采样数据的方式建立一个小表，通过计算部分数据产生统计信息的估算值。如果是分区表，则从全部分区中采样。

大多数情况下，默认设置能够提供生成正确查询执行计划的信息。如果产生的统计不能生成优化的查询执行计划，管理员可以调整配置参数，通过增加样本数据量产生更加精确的统计。统计信息越精确，所消耗的 CPU 和内存资源越多，因此可能由于资源的限制，无法生成更好的计划。

此时就需要查看执行计划并测试查询性能，目标是要通过增加的统计成本达到更好的查询性能。

2. 系统统计

（1）表大小

查询优化器使用查询必须处理的数据行数和必须访问的磁盘页数等统计信息，寻找查询所需的最小磁盘 I/O 和网络流量的执行计划。用于估算行数和页数的数据分别保存在 pg_class 系统表的 reltuples 列和 relpages 列中，其中的值是最后运行 VACUUM 或 ANALYZE 命令时生成的数据。随着行的添加或删除，估算的准确性会降低，但操作系统始终可以提供磁盘页面的准确计数，因此只要 reltuples 与 relpages 的比例没有显著变化，优化器就可以生成足够准确的行数估计值，以选择正确的查询执行计划。

如果 reltuples 列的值与 SELECT COUNT(*)的返回值差很多，应该执行分析以更新统计信息。运行 REINDEX 命令完成重新创建索引后，relpages 列和 reltuples 列将重置为 0，此时应该在表上运行 ANALYZE 命令以更新这些列。

（2）pg_statistic 系统表与 pg_stats 视图

pg_statistic 系统表保存每个数据库表上最后执行 ANALYZE 操作的结果。pg_stats 视图以一种更友好的方式表示 pg_statistic 的内容。

为一列收集的统计信息因不同的数据类型而有所差异，因此 pg_statistic 表将适合相应数据类型的统计信息存储在四个槽位中，每个槽位由四列组成。例如，第一个槽位通常包含列的最常用值，由 stakind1、staop1、stanumbers1 和 stavalues1 列组成。

stakindN 列中的每一列都包含一个数字代码，用于描述存储在其槽位中的统计信息的类型。从 1 到 99 的 stakind 代码是为 PostgreSQL 数据类型保留的。Greenplum 数据库使用代码 1、2、3、4、5 和 99，0 表示槽位未使用。

新创建的表和索引没有统计信息。可以使用 gp_stats_missing 视图检查缺少统计信息的表，该视图位于 gp_toolkit 模式中：

```
select * from gp_toolkit.gp_stats_missing;
```

（3）统计更新

运行不带参数的 ANALYZE 会更新数据库中所有表的统计信息，这可能需要执行很长时间，因此最好在数据更改后有选择地分析表。也可以选择分析一个表列的子集，例如只分析 join、where、order by、group by、having 等子句中用到的列。

如果样本包含空页，则分析严重膨胀的表可能会生成较差的统计信息。因此在分析膨胀表前，最好先执行 VACUUM 操作。

（4）分析分区表和 AO 表

在分区表上运行 ANALYZE 命令时，它逐个分析每个叶级别的子分区。可以只在新增或修改的分区文件上运行 ANALYZE，避免分析没有变化的分区。analyzedb 命令行程序自动跳过无变化的分区，并且它是多会话并行的，可以同时分析几个分区，默认运行 5 个会话，会话数可以通过命令行的-p 选项设置，值域为 1～10。每次运行 analyzedb，它都会将 AO 表和分区的状态信息保存在 Master 节点数据目录中的 db_analyze 目录下，比如 /data/master/gpseg-1/db_analyze。下次运行时，analyzedb 比较每个表的当前状态与上次保存的状态，不分析没有变化的表或分区。Heap 表总是会被分析。

GPORCA 查询优化器需要分区表根级别的统计信息，而老的优化器不使用该统计。如果启用（默认）了 GPORCA 优化器，则还需要运行 ANALYZE 或 ANALYZE ROOTPARTITION 来刷新根分区统计信息。分析分区表的时间与分析具有相同数据的非分区表的时间类似，因为 ANALYZE ROOTPARTITION 不收集叶分区的统计信息（仅对数据进行采样）。

Greenplum 服务器配置参数 optimizer_analyze_root_partition 会影响在分区表的根分区上收集统计信息的时间。如果参数为 on（默认），则在运行 ANALYZE 时，不需要 ROOTPARTITION 关键字来收集根分区的统计信息。在根分区上运行 ANALYZE 时，或者在分区表的叶级别叶子分区上运行 ANALYZE，并且其他叶子分区具有统计信息时，将收集根分区统计信息。如果所有子分区的统计信息都已经更新，ROOTPARTITION 选项可用于只收集分区表的全局状态信息，这可以节省分析时间。

如果该参数处于禁用状态，则必须运行 ANALYZE ROOTPARTITION 以收集根分区统计信息。如果不使用 GPORCA 对分区表运行查询（将服务器配置参数 optimizer 设置为 off），可以将服务器配置参数 optimizer_analyze_root_partition 也设置为 off，以限制 ANALYZE 更新根分区统计信息。

3. 统计配置

（1）统计目标

统计目标指的是一个列的 most_common_vals、most_common_freqs 和 histogram_bounds 数组的大小。这些数组的含义可以从 pg_stats 视图的定义得到。可以通过设置服务器配置参数修改全局目标值，也可以使用 ALTER TABLE 命令设置任何表列的目标值。目标值越大，优化器评估质量越高，但 ANALYZE 需要的时间也越长。

default_statistics_target 服务器配置参数设置系统默认的统计目标。默认值 100 通常已经足够，只有经过测试确定要定义一个新目标时，才考虑提高或降低该值。例如，要将默认统计信息目标从 100 提高到 150，可以使用 gpconfig 程序：

```
gpconfig -c default_statistics_target -v 150
```

单个列的统计目标可以用 ALTER TABLE 命令设置。某些查询可以通过为特定列，尤其是分布不规则的列增加目标值以提高性能。如果将一列的目标值设置为 0，ANALYZE 将忽略该列。下面的命令将 notes 列的统计目标设置为 0，因为该列对于查询优化没有任何作用：

```
alter table emp alter column notes set statistics 0;
```

统计目标可以设置为 0～1000 的值，或者设置成-1，此时恢复使用系统默认的统计目标值。父分区表上设置的统计目标影响子分区，如果父表上某列的目标设置为 0，则其所有子分区上的该列统计目标也为 0。但是，如果以后增加或者交换了其他子分区，新增的子分区将使用默认目标值，交换的子分区使用以前的统计目标。因此如果增加或交换了子分区，应该在新的子分区上设置统计目标。

（2）自动收集统计信息

如果一个表没有统计信息，或者在表上执行的特定操作改变了大量数据时，Greenplum 可以在表上自动运行 ANALYZE。对于分区表，自动统计收集仅当直接操作叶表时被触发，它仅分析叶表。自动收集统计信息有三种模式：

- none：禁用自动收集。
- on_no_stats：在一个没有统计信息的表上执行 CREATE TABLE AS SELECT、INSERT、COPY 命令时触发分析操作。
- on_change：在表上执行 CREATE TABLE AS SELECT、UPDATE、DELETE、INSERT 或 COPY 命令，并且影响的行数超过了 gp_autostats_on_change_threshold 配置参数设定的阈值时触发分析操作。

依据命令是单独执行还是在函数中执行，自动收集统计信息模式的设置方法也不一样。如果是在函数外单独执行，gp_autostats_mode 配置参数控制统计模式，默认值为 on_no_stats。gp_autostats_mode_in_functions 参数控制在过程语言函数中执行表操作时的行为，默认情况下设置为 none。

在 on_change 模式下，仅当受影响的行数超过 gp_autostats_on_change_threshold 配置参数定义的阈值时，才会触发分析。此参数的默认值是一个非常高的值，为 2147483647，能有效禁用自动统计数据收集。on_change 模式可能触发意外中断系统的大型分析操作，因此不建议全局设置。on_change 在会话中可能很有用，例如在加载后自动分析表。

要禁用函数外的自动统计信息收集，将 gp_autostats_mode 参数设置为 none。

```
gpconfigure -c gp_autostats_mode -v none
```

将 gp_autostats_mode_in_functions 更改为 on_no_stats，可为函数中没有统计信息的表启用自动统计信息收集。

```
gpconfigure -c gp_autostats_mode_in_functions -v on_no_stats
```

如果要记录自动统计信息收集操作，将 log_autostats 系统配置参数设置为 on。

3.4　为什么选择 Greenplum

现在的 SQL、NoSQL、NewSQL、Hadoop 等技术，都能在不同层面或不同应用上处理大数据的某些问题，其中 Hadoop 是较早用来处理大数据集合的分布式存储计算基础架构。那么作为用户，面对这么多技术选型，我们何时以及为什么要选择 Greenplum 构建数据仓库呢？近年来笔者也尝试过一些 SQL-on-Hadoop 产品，从最初的 Hive，到 Spark SQL、Impala，再到 Kylin、HAWQ，在这些产品上进行了一系列 ETL、CDC、多维数据仓库、OLAP 实验。从数据库的角度看，笔者的总体感觉是这些产品与传统的 DBMS 相比，功能不够完善，性能差距较大，甚至很难找到一个相对完备的数据仓库解决方案。本节将以笔者个人的实践体验对比一下 Greenplum 与 SQL-on-Hadoop，并简述 Greenplum 的可行性和局限性。

3.4.1　Greenplum 还是 SQL-on-Hadoop

Greenplum 和 Hadoop 都是为了解决大数据并行计算而出现的技术，二者的相似点在于：

- 分布式存储数据在多个节点上。
- 采用分布式并行计算框架。
- 支持向外扩展来提高整体的计算能力和存储容量。
- 支持 X86 开放集群架构。

但两种技术在数据存储和计算方法上，也存在很多显而易见的差异：

- Greenplum 按照关系数据库行列表方式存储数据（有模式）；Hadoop 按照文件切块方式分布式存储（无模式）数据。
- 两者采用的数据分布机制不同，Greenplum 采用 Hash 分布，计算节点和存储紧密耦合，数据分布在记录级的更小粒度，一般在 1KB 以下；Hadoop FS 按照文件切块后随机分配，节点和数据无耦合，数据分布粒度在文件块级，默认为 64MB。
- Greenplum 采用 SQL 并行查询计划；Hadoop 采用 MapReduce 框架。

基于以上不同，体现在效率、功能特性等方面也大不相同。Greenplum 数据库在计算并行度、计算算法上比 Hadoop 更加优雅，效率更高。图 3-11 由 Pivotal 提供，显示相同硬件环境下，基于 MapReduce 的 Hive 和 Greenplum 在 TPC-H（商业智能测试）22 个 SQL 测试中的性能比较，可以看到两者的执行速度相去甚远。

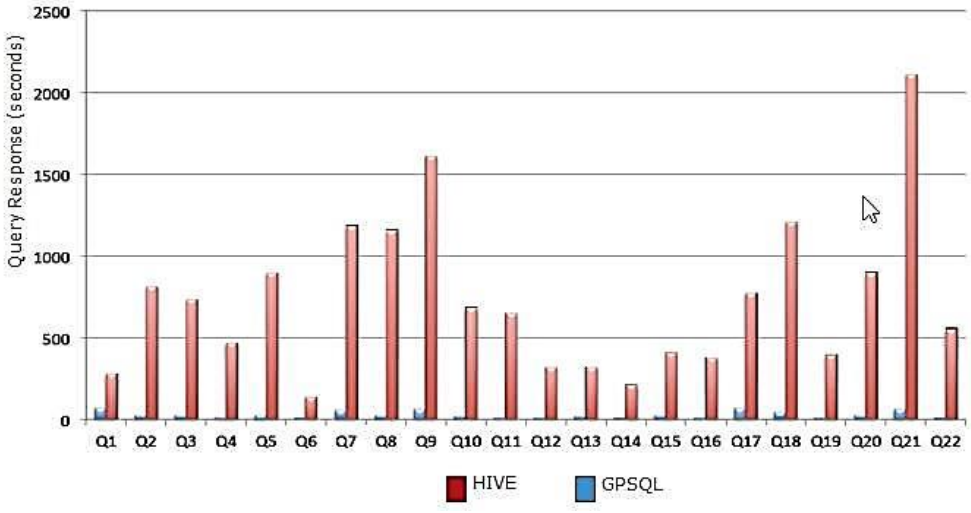

图 3-11　Hive 和 Greenplum 在 TPCH 中的性能比较

为了取得第一手数据，笔者做了以下两个简单的 Greenplum 与 MySQL 查询性能对比测试，以便有一个最初的直观体验。也许你会觉得拿分布式集群数据库与单机集中式数据库作比较有失公允，没错！笔者想说明的是：这两个查询都是线上实际在 MySQL 上运行的慢查询，而考虑 Greenplum 就是为了解决大数据量在 MySQL 上查不动的问题。而且这也并不是严格的对等测试，Greenplum 只是由三台测试机组成的集群，而 MySQL 使用的是线上高配服务器。

```
-- 查询 1:
select userid, target, relation_type, update_time
  from relation
```

```
 where userid = 717600270
   and relation_type in (1, 2)
 order by update_time desc
 limit 30;

-- 查询 2:
select a.*
  from moments_dynamic a -- force index (idx_user_all)
  join (select target from relation r
        where r.userid = 918046590
          and (r.relation_type = 1 or r.relation_type = 2)
        union all
       select 918046590) b
    on a.userid=b.target
 where dynamic_status = 0
   and dynamic_type in (1, 6, 8, 11, 13)
 order by id desc
 limit 80;
```

moments_dynamic 表有 79309341 行，relation 表有 499194521 行。查询 1，Greenplum 用时 44 ms，MySQL 用时 9210 ms；查询 2，Greenplum 用时 75ms，MySQL 用时 170 ms(force index (idx_user_all))。

在功能上 Greenplum 数据库采用 SQL 作为主要交互式语言。SQL 语言简单易学，具有很强的数据操纵能力和过程语言的流程控制能力，是专门为统计和数据分析开发的语言，丰富的功能和函数极大简化了数据操作和交互过程。

而对于 MapReduce 编程明显是困难的，在原生的 MapReduce 开发框架基础上进行开发，需要技术人员谙熟 Java 开发和并行原理，而这即便是技术人员也难以学习和操控。为了解决易用性问题，近年来 SQL-on-Hadoop 技术大量涌现出来，成为当前 Hadoop 开发使用的一个技术热点。其中，Hive 支持 MapReduce、Spark、Tez 三种计算框架；Spark SQL 采用内存中的 MapReduce；Impala、HAWQ 则借鉴 MPP 计算思想来做查询优化和内存数据管道计算，以此来提高性能。

虽然 SQL-on-Hadoop 比原始的 MapReduce 在易用性上有所提高，但在 SQL 成熟度和复杂分析上目前还与 Greenplum 数据库有较大差距，笔者在使用过程中对此深有体会：

- SQL-on-Hadoop 系统中，除了 HAWQ（HAWQ 从代码级别上可以简单理解成是数据存储在 HDFS 上的 Greenplum 数据库）外，其余系统对 SQL 的支持都非常有限，特别是分析型复杂 SQL，如 SQL 2003 OLAP WINDOW 函数，几乎都不支持，更不用说存储过程等数据库常规功能。以 Impala 为例，不支持 Date 数据类型，不支持 XML 和 JSON 相关函数，不支持 covar_pop、covar_samp、corr、percentile、percentile_approx、histogram_numeric、collect_set 等聚合函数，不支持 rollup、cube、grouping set 等操作，不支持数据抽样（Sampling）等数据分析中的常用操作。在 TPC-DS 测试中，Spark SQL、Impala、Hive 等只支持其中 1/3 左右的 SQL 测试。TPC-DS 是专门用于评测决策支持系统（大数据或数据仓库）的标准 SQL 测试集，包含 99 个 SQL。
- 由于 HDFS 本身只能追加数据的特性（Append-only），SQL-on-Hadoop 大多不支持行级数据更新（update）和删除（delete）功能。而像 Hive 虽然通过配置可以支持事务和行级

更新，但实现极为别扭，性能更是无法接受，基本不具实用价值。

- SQL-on-Hadoop 不擅长于交互式即席查询，多通过预关联的方式来规避这个问题。另外，在并发处理方面能力较弱，高并发大查询场景下，需要控制计算请求的并发度，避免资源过载导致的稳定性和性能下降等问题。笔者就曾多次遇到几个并发 Spark SQL 任务占用大量内存，最终出现 OOM 错误的情况。

反观专为大数据存储、计算、挖掘而设计的 Greenplum，它所拥有的丰富特性使其成为构建数据仓库等分析型应用的理想选择：

- 完善的标准支持。Greenplum 完全支持 ANSI SQL 2008 标准和 SQL OLAP 2003 扩展。例如，支持内连接、外连接、全连接、笛卡尔连接、相关子查询等所有表连接方式，支持并集、交集、差集等集合操作，并支持递归函数调用。作为一个数据库系统，提供这些功能很好理解。
- 除了包含诸多字符串、数字、日期时间、类型转换等常规标量函数以外，Greenplum 还包含丰富的窗口函数和高级聚合函数，这些函数经常被用于分析型数据查询。窗口函数包括 cume_dist、dense_rank、first_value、lag、last_valueexpr、lead、ntile、percent_rank、rank、row_number 等。高级聚合函数包括 median、percentile_cont (expr) within group (order by expr [desc/asc])、percentile_disc (expr) within group (order by expr [desc/asc])、sum(array[])、pivot_sum (label[], label, expr)等。
- 得益于 PostgreSQL 良好的扩展性（这里是 extension，不是 scalability），Greenplum 可以采用各种开发语言来开发用户自定义函数（UDF）。自定义函数部署到 Greenplum 后，能充分享受到实例级别的并行性能优势。建议把库外的处理逻辑部署为用 MPP 数据库的 UDF 这种库内方式来处理，这将获得意想不到的性能和方便。
- 支持分布式事务，支持 ACID，保证数据的强一致性。
- Greenplum 支持用“Hadoop 外部表”方式来访问、加载 HDFS 的数据。虽然 Greenplum 的 Hadoop 外部表性能大幅低于 MPP 内部表，但比 Hadoop 自身的 Hive 要快很多。Greenplum 还提供了 gpfdist 文件服务器，可并行读写本地文件系统中的文件，最大化数据装载性能。
- Greenplum 有完善的生态系统，可以与很多企业级产品集成，如 SAS、Cognos、Informatic、Tableau 等；也可以与很多种开源软件集成，如 Pentaho、Talend 等。

3.4.2 适合 DBA 的解决方案

Greenplum 最吸引人的地方是它支持 SQL 过程化编程，这是通过 UDF 实现的。编写 UDF 的语言可以是 SQL、C、Java、Perl、Python、R 和 pgSQL。数据库应用开发人员常用的自然是 SQL 和 pgSQL，PL/pgSQL 函数可以为 SQL 语言增加控制结构，执行复杂的计算任务，并继承所有 PostgreSQL 的数据类型（包括用户自定义类型）、函数和操作符。Greenplum 是笔者所使用过的分布式数据库解决方案中唯一支持 SQL 过程化编程的。对于习惯了编写存储过

程的 DBA（Database Administrator，数据库管理员）来说，这无疑大大提高了 Greenplum 的易用性。下面列举一些 UDF 提供的特性。

1. 给内部函数起别名

许多 Greenplum 的内部函数是用 C 语言编写的，这些函数在集群初始化时声明，并静态连接到 Greenplum 服务器。用户不能自己定义新的内部函数，但可以给已存在的内部函数起别名。下面的例子创建了一个新的函数 fn_all_caps，它是内部函数 upper 的别名。

```
    create function fn_all_caps (text) returns text as 'upper' language internal
strict;
```

2. 返回结果集的表函数

表函数返回多行结果集，在 FROM 子句中进行调用，就像查询一个表、视图或子查询。如果表函数返回单列，那么返回的列名就是函数名。下面是一个表函数的例子，该函数返回 channel 表中给定 ID 值的数据。

```
    create function fn_getchannel(int) returns setof channel as $$
       select * from channel where id = $1;
    $$ language sql;
```

3. 参数个数可变的函数

Greenplum 从 PostgreSQL 继承了一个非常好的特性，即函数参数的个数可变，在某些数据库系统中，想实现这个功能很麻烦。参数个数可变是通过一个动态数组来实现的，因此所有参数都应该具有相同的数据类型。这种函数将最后一个参数标识为 VARIADIC，并且参数必须声明为数组类型。下面是一个例子，实现类似原生函数 greatest 的功能。

```
    create or replace function fn_mgreatest(variadic numeric[]) returns numeric
as $$
    declare
       l_i numeric:=-99999999999999;
       l_x numeric;
       array1 alias for $1;
    begin
      for i in array_lower(array1, 1) .. array_upper(array1, 1)
      loop
        l_x:=array1[i];
        if l_x > l_i then
           l_i := l_x;
        end if;
      end loop;
      return l_i;
    end;
    $$ language 'plpgsql';
```

执行函数结果如下：

```
    db1=# select fn_mgreatest(variadic array[10, -1, 5, 4.4]);
     fn_mgreatest
    --------------
              10
    (1 row)
```

```
db1=# select fn_mgreatest(variadic array[10, -1, 5, 4.4, 100]);
 fn_mgreatest
--------------
          100
(1 row)
```

4. 多态数据类型

PostgreSQL 中的 anyelement、anyarray、anynonarray 和 anyenum 四种伪类型被称为多态类型，使用这些类型声明的函数叫作多态函数。多态函数的同一参数在每次调用函数时可以有不同数据类型，实际使用的数据类型由调用函数时传入的参数确定。当一个查询调用多态函数时，特定的数据类型在运行时被解析。

如果一个函数的返回值被声明为多态类型，那么它的参数中至少应该有一个是多态的，并且参数与返回结果的实际数据类型必须匹配。例如，函数声明为 assubscript(anyarray, integer) returns anyelement。此函数的第一个参数为数组类型，而且返回值必须是实际数组元素的数据类型。再比如，一个函数的声明为 asf(anyarray) returns anyenum，那么参数只能是枚举类型的数组。参数个数可变的函数也可以使用多态类型，实现方式是声明函数的最后一个参数为 VARIADIC anyarray。

下面看几个多态类型函数的例子。

①判断任意两个相同数据类型的入参是否相等：

```
create or replace function fn_equal (anyelement,anyelement)
   returns boolean as $$
begin
   if $1 = $2 then return true;
   else return false;
   end if;
end; $$
language 'plpgsql';
```

函数调用：

```
db1=# select fn_equal(1,1);
 fn_equal
----------
 t
(1 row)

db1=# select fn_equal(1,'a');
ERROR:  invalid input syntax for integer: "a"
LINE 1: select fn_equal(1,'a');
                        ^
db1=# select fn_equal('a','A');
ERROR:  could not determine polymorphic type because input has type "unknown"

db1=# select fn_equal(text 'a',text 'A');
 fn_equal
----------
 f
(1 row)

postgres=# select fn_equal(text 'a',text 'a');
```

```
 fn_equal
----------
 t
(1 row)
```

②遍历任意类型的数组，数组元素以行的形式返回：

```
create or replace function fn_unnest(anyarray)
returns setof anyelement
language 'sql' as
$$
  select $1[i] from generate_series(array_lower($1,1),array_upper($1,1)) i;
$$;
```

函数调用：

```
db1=# select fn_unnest(array[1,2,3,4]);
 fn_unnest
-----------
         1
         2
         3
         4
(4 rows)

db1=# select fn_unnest(array['a','b','c']);
 fn_unnest
-----------
 a
 b
 c
(3 rows)
```

③返回任意数组类型中的最大元素：

```
create or replace function fn_mgreatest1(v anyelement, variadic anyarray)
returns anyelement as $$
declare
   l_i v%type;
   l_x v%type;
   array1 alias for $2;
begin
   l_i := array1[1];
   for i in array_lower(array1, 1) .. array_upper(array1, 1) loop
      l_x:=array1[i];
      if l_x > l_i then
         l_i := l_x;
      end if;
   end loop;
   return l_i;
end;
$$ language 'plpgsql';
```

说明：

- 变量不能定义成伪类型，但可以通过参数进行引用，如上面函数中的 l_i v%type。

- 动态数组必须是函数的最后一个参数。
- 第一个参数的作用仅是为变量定义数据类型，所以在调用函数时传空即可。

函数调用：

```
db1=# select fn_mgreatest1(null, variadic array[10, -1, 5, 4.4]);
 fn_mgreatest1
---------------
          10
(1 row)

db1=# select fn_mgreatest1(null, variadic array['a', 'b', 'c']);
 fn_mgreatest1
---------------
 c
(1 row)
```

3.4.3 Greenplum 的局限

Greenplum 最大的特点总结起来就一句话：在低成本开放平台基础上，提供强大的并行数据计算性能和海量数据管理能力。并行计算能力是对大任务、复杂任务的快速高效计算，但如果我们指望 MPP 并行数据库能够像 OLTP 数据库一样，在极短的时间处理大量的高并发小任务，这个并非 MPP 数据库所长。请牢记，并行和并发是两个完全不同的概念，并发强调的是一同发起，并行强调的是一起执行。MPP 数据库是为了解决大数据问题而设计的并行计算技术，而不是大量小数据问题的高并发请求。

再通俗点说，Greenplum 主要定位在 OLAP 领域。利用 Greenplum MPP 数据库做大数据计算或分析平台是非常适合的，例如数据仓库系统、ODS 系统、历史数据管理系统、分析系统、数据集市等。而 MPP 数据库都不擅长做 OLTP 交易系统，所谓交易系统，就是高频的事务型小规模数据插入、修改、删除，每次事务处理的数据量不大，但每秒钟都会发生几十次甚至几百次以上交易型事务。这类系统的衡量指标是 TPS，适用的系统是 OLTP 数据库，如 MySQL。在本书 5.6.6 小节“消费延迟监控”中，会看到作为主打 AP 的 Greenplum，在同步适合 TP 的 MySQL 数据时，所表现出来的量化的性能差异。

除了 OLTP、OLAP 之分，近来还出现了所谓的 HTAP（Hybrid Transactional/Analytical Processing），即混合事务/分析处理。在笔者看来，HTAP 作为概念的意义大于实用意义，至少目前如此，就像概念车，外观上无比炫酷，但就是不会量产。从原理上讲，TP 与 AP 在需求、应用场景、性能衡量指标、建模与设计方法、优化策略等方面都截然不同（参见 1.2.3 小节“操作型系统和分析型系统的对比”），结果必然是在实现技术上大相径庭。正所谓鱼和熊掌不可兼得，虽然也有一些打上 HTAP 标签的产品，但终究还是在 AP 与 TP 之间做权衡，侧重其中之一，而在另一方面的表现则差强人意。

3.5　小　结

（1）Greenplum 是 MPP 架构的分布式数据库，针对分析型应用，尤其适合于数据仓库。

（2）Greenplum 建立在无共享架构上，其并行工作方式可以最大限度地发挥硬件能力。

（3）Master、Segment 和 Interconnect 是构成 Greenplum 的顶层组件。每个 Master 和 Segment 都是一个单独的 PostgreSQL 数据库实例。Master 是 Greenplum 的系统入口，负责接收客户端连接和 SQL 查询，并将工作分配给 Segment 实例。Master 实例只存储系统元数据，不存储任何用户数据。每个 Segment 存储部分用户数据，处理部分查询工作。Interconnect 是 Greenplum 数据库体系结构中的核心组件，实现了对同一个集群中多个 PostgreSQL 实例的高效协同和并行计算，承载了并行查询计划生产和分派、协调节点上查询执行器的并行工作、数据分布、Pipeline 计算、镜像复制、健康探测等诸多任务。

（4）Greenplum 支持 Heap 表和 AO 表，Heap 表支持行存，AO 表支持行存和列存，并可以使用压缩。

（5）Greenplum 继承了 PostgreSQL 的 MVCC 模型实现并发控制，支持 SQL 标准中的全部四种事务隔离级，默认隔离级为 Read Committed。

（6）每个 Segment 中有多个并行的工作进程协同处理一个查询。

（7）利用 gpfdist 外部表或 gpload 程序，可以向 Greenplum 并行装载外部数据，最大化数据装载性能。

（8）Primary/Mirror 提供 Segment 自动检测和故障切换，Mirror 有 group 和 spread 两种标准分布方式，默认为 group。Standby 提供 Master 热备，在需要时可手工激活。这两种机制共同为生产环境提供高可用、容错的 Greenplum 环境。

（9）Greenplum 查询优化器依赖表列的统计信息生成执行计划，可以通过配置自动收集这些统计信息。

（10）有别于 SQL-on-Hadoop 技术，Greenplum 更符合数据库特性，支持行级更新，通过 UDF 实现过程化编程，编写 UDF 的语言可以是 SQL、C、Java、Perl、Python、R 和 pgSQL。这些功能特性在提高易用性的同时提供了更高的性能，是更适合 DBA 的解决方案。

（11）Greenplum 不适合 OLTP 类型的应用场景。