第20章　分布式数据库

20.1　引言

我们在第2章结束的时候涉及到了分布式数据库的问题，在那里我们说，“对分布式数据库的支持意味着，一个单一的应用应该可以对数据进行透明的操作，这些数据在不同的数据库中分布、由不同的 DBMS管理、在不同的机器上运行、受不同的操作系统支持、被不同的通信网络连接在一起 —— 在这里透明的意思是指从逻辑角度看，应用程序所操作的数据好像是由运行在一台机器上的一个单一的 DBMS管理着”。我们现在将要更细致地讨论这些概念。具体而言，在本章里我们将明确地解释什么是分布式数据库，为什么这样的数据库会变得越来越重要，以及在分布式数据库领域中的一些技术难题是什么。

第2章还简要地讨论了客户 /服务器系统，它可以看作是一般分布式系统的一个简单的特例。在20.5节我们将专门讨论客户 /服务器系统。

本章的安排将在下一节的最后进行说明。

20.2　一些预备知识

我们先从一个指导性的定义开始（目前还不需要很严密）：

- 一个分布式数据库系统由一系列的场地组成，通过某种通信网络连接在一起，其中：
  a. 每个场地自身都有一个完全的数据库系统，但是
  b. 所有的场地都可以协同工作，使得任何场地上的用户都可以访问网络上任何地方的数据，就好像数据是存储在用户自己的场地上一样。

由此可见，所谓的“分布式数据库”实际上是一种虚拟的数据库，它的各个组成部分物理地存储在许多不同场地上的不同的“真实”数据库中（从效果上讲，它是这些真实数据库逻辑上的并集）。图20-1中给出了一个例子。

再一次提请注意，每个场地自身都有一个数据库系统。换句话说，每个场地有其自己本地的“真实”数据库、自己本地的用户、自己本地的 DBMS和事务管理软件系统（包括自己本地的封锁、日志、恢复等软件系统）还有自己本地的数据通信管理器（ DC管理器）。尤其要说明的是，一个用户可以操作自己本地场地上的数据，而完全感觉不到这个场地参与了一个分布式系统（至少这是一个目标）。因此，可以认为分布式数据库系统是独立场地上的独立的 DBMS之间形成的一种合作关系。由位于每个场地上的一个新的软件模块 —— 它在逻辑上是本地 DBMS的扩展 —— 来提供所需的合作功能，由这个新的模块与已经存在的DBMS一起构成我们所说的分布式数据库管理系统。

顺带说一句，通常都假设所有参与场地在物理上是分散的 —— 可能确实在地理上就是分散的，就像在图 20-1中所表示的那样 —— 虽然实际上只要它们在逻辑上是分散的就可以了。两个“场地”甚至在物理上可以处在同一台机器中（尤其是在最初的系统测试期间）。其实，分布式系统中所强调的重点是随着时间的推移来回变化的，最早的研究倾向于采

用地理的分布；但是最早的几个商业系统采用的却是本地分布，（比如）许多“场地”都在一幢大楼里，通过局域网（LAN）连接在一起。可是后来，广域网（WAN）的迅速兴起与发展又把兴趣重新拉回到了地理分布的可能性上。但是无论怎样，从数据库的角度来看，这些都没有太大的差别——关键是需要解决的（数据库）技术问题都是一样的——所以为了本章的需要，我们可以合理地认为图 20-1表示的就是一个典型的系统。

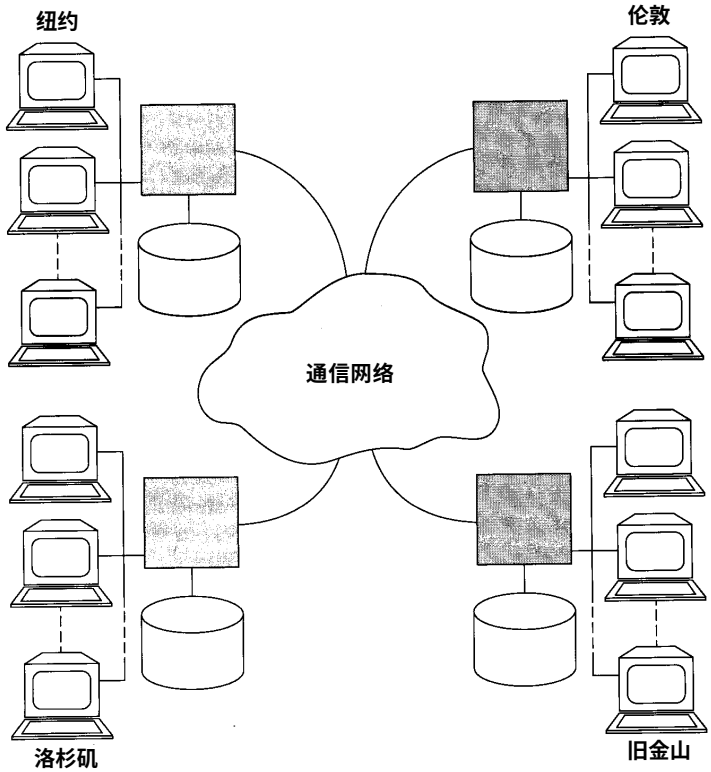

图20-1 一个典型的分布式数据库系统

注意：为了叙述简单，除非特别说明，我们假设系统是同构的，意思是说每个场地上运行的DBMS都是一样的。我们把这叫做严格同构假设。我们将在 20.6节研究放松这一假设的可能性——以及这种放松所包含的一些潜在内容。

1. 优越性

为什么分布式系统是有价值的？最基本的答案是企业自身经常就已经是分布式的，至少是在逻辑上（被分成分公司、部门、工作组，等等），而且也非常可能是在物理上（被分成工厂、车间、实验室，等等）——这就意味着数据通常就已经是分布的了，因为企业中的每个部门都会很自然地维护与自己工作有关的数据。这样，企业的整个信息资产就被分裂成通常所说的信息孤岛 (island of information)。而分布式系统所做的就是为把这些小岛联系在一起提供桥梁。换句话说，它使得数据库的结构能够反映企业的结构——本地数据可以保存在本地，即它在逻辑上所从属的地方——而同时可以在需要的时候存取异地的数据。

一个例子有助于明确上面的论述。还是来看看图 20-1。为了简单起见，假设只有洛杉矶和旧金山两个场地，而系统是银行系统，包括洛杉矶和旧金山各自本地账户的账户数据。那么

优越性是显而易见的：分布式的安排结合了处理的有效性（数据的存储靠近最经常被用到的地方）与增强的可存取性（很可能要通过网络从旧金山存取一个洛杉矶的帐号，反之亦然）。

能够使数据库的结构反映企业的结构（像刚才所说的那样）可能是分布式系统最大的优越性。当然还有非常多的其他优越性，稍后将在本章合适的地方分别讨论这些优越性。但是我们也应该看到还有一些不利的地方，其中最大的就是分布式系统都比较复杂，至少从技术的观点看是这样。当然在想法上，这种复杂性应该是系统实现人员的问题，而不是用户的问题。但是 —— 从实用的角度看 —— 用户很可能会面对复杂性的某些方面，除非采取了非常小心的预防措施。

2. 典型系统

为了在后面引用的方便，我们在这里简要地介绍一些比较知名的分布式系统。首先是原型系统。在众多的研究系统中，最著名的三个是（ a）SDD-1，这是由美国计算机公司（Computer Corporation of America）的研究部门在 20世纪70年代末期到80年代早期实现的[20.34]；（b）R*（“R星”），这是系统R原型系统的分布式版本，是在 80年代早期由IBM研究院（IBM Reaserch）实现的[20.39]；（c）分布式Ingres（distributed Ingres），这是Ingres原型系统的分布式版本，同样是在 80年代早期由加利福尼亚大学伯克利分校实现的 [20.36]。

对于商业系统而言，今天的大部分 SQL产品都提供了某种对分布式数据库的支持（当然所提供功能的程度不一样）。最著名的包括（ a）Ingres/Star，Ingres的分布式数据库组件；（b）Oracle的分布式数据库选件（ distributed database option）；（c）DB2的分布式数据支持（distributed data facility）。注意：以上列举的两类系统显然并不是意味着穷尽了所有的系统 —— 这些系统只是由于这样或者那样的原因而曾经产生过或者正在产生着很大的影响，或者是有着某些特别的内在特征。

值得一提的是以上所列举的这些系统，无论是原型系统还是商用系统，都是关系系统（至少它们都支持 SQL）。事实上，有很多理由说明为什么，如果一个分布式系统要成功，它必须是关系的；关系技术是（有效的）分布式技术的一个先决条件 [20.15]。在讲述本章的过程中，我们会看到有很多理由支持这一论述。

3. 一个基本的原则

现在我们可以阐述分布式数据库的基本原则[20.14]：

• 对用户来说，一个分布式系统应该看起来完全象一个非分布式系统。

换句话说，用户在使用分布式系统的时候，应该完全感觉不到系统是分布的。分布式系统的所有问题都是 —— 或者应该是 —— 内部或实现层次上的问题，而不是外部或用户层次的问题。

注意：上述段落里的“用户”是特指那些执行数据操纵操作的用户（最终用户或应用程序员）。所有的数据操纵操作在逻辑上应该是不变的。相反，数据定义操作在分布式系统中则需要进行一些扩展 —— 比如，一个在场地 *X*的用户就可以把一个关系分成不同的“片断”，存储在场地 *Y*和场地 *Z*上（见下一节关于分片的讨论）。

上面所说的基本原则会带来一些补充的规则或目标 ㊀，它们总共有 12个，我们将在

㊀ “规则”作为论文术语是在 [20.14]中首次被引入的 (而“基本原则”是指零规则 (Rule Zero))。但是，“目标”实际上应该是一个更好的术语 —— “规则”听起来过于教条了。在本章中我们将一直使用中性的术语“目标”。

下一节对它们进行讨论。为了引用方便，我们把这些目标列在这里：

1) 本地自治（local autonomy）；
2) 不依赖中心场地（no reliance on a central site）；
3) 可连续操作性（continuous operation）；
4) 位置独立性（location independence）；
5) 分片独立性（fragmentation independence）；
6) 复制独立性（replication independence）；
7) 分布式查询处理（distributed query processing）；
8) 分布事务管理（distributed transaction management）；
9) 硬件独立性（hardware independence）；
10) 操作系统独立性（operating system independence）；
11) 网络独立性（network independence）；
12) DBMS独立性（DBMS independence）；

需要了解的是这些目标既不是相互独立的，也肯定不会是毫无遗漏的，而且它们也不可能都是同等重要的（不同的用户在不同的环境中会给予不同的目标以不同的重要程度。事实上，其中的一些目标在某些环境下也许是完全不适用的）。但是，以这些目标为基础，通常可以有助于理解分布式技术；而且以它们为框架，还可以有助于规划一个特定的分布式系统的功能。因此在本章内容的主要部分中，我们将以它们为组织原则。 20.3节将对每个目标进行简要的讨论； 20.4节将从更为细节的角度着眼于某些特定问题； 20.5节（像前面所提到的）将讨论客户 /服务器系统；20.6节将深入地考查 DBMS独立性；最后， 20.7节的问题是对 SQL的支持，而 20.8节进行了小结并给出了一个结论式的说明。

最后要介绍的一点是：区分这样两种系统是非常重要的，一种是真正的、普遍意义上的分布式数据库系统，另一种是那种只能提供远程数据存取的系统（顺便提一句，所有的客户 /服务器系统都能做到这一点）。在一个远程数据存取系统中，用户也许可以操作远程场地的数据，甚至是同时操作许多远程场地上的数据，但是“ 远程和本地结合的接缝是显而易见的 ”。用户肯定知道 —— 或多或少地 —— 数据是在异地的，因而要采取相应的操作；相反，在一个真正的分布式数据库系统中，远程和本地结合的接缝被隐藏起来了（我们此处所说的“ 接缝被隐藏起来 ”意味着什么，是本章剩下部分里的很多地方所关心的）。在所有下面的讨论中，我们将用“ 分布式系统 ”这个术语来特指一个真正的、普遍意义上的分布式数据库系统（区别于一个简单的远程数据存取系统），而不使用精确的表述。

20.3 十二个目标

1. 本地自治

一个分布式系统中的场地应该是自治的。本地自治是指在一个给定场地上的所有操作由这个场地自己控制，场地 *X*上的成功操作不应该依赖于某个其他的场地 *Y*（相反的情况是场地 *Y*停机了也许意味着场地 *X*也不能运转了，即使场地 *X*自己没有任何问题 —— 这显然是一种不希望发生的情况）。本地自治还意味着本地数据是由本地拥有和管理的，并具有本地的可计算性：所有的数据都是“ 真正 ”属于某个本地数据库的，即使它们可以被其

他的、远程的场地访问。本地数据的安全性、完整性和存储形式之类的问题也是在本地场地的控制和管辖之下的。

实际上，本地自治的目标是无法完全达到的——在不少情况下，一个给定的场地 *X* 必须交出一定程度的控制权给某个场地 *Y*。因此自治目标也许应该这样表述更为精确：场地应该在尽可能大的程度上是自治的。可以参看对文献 [20.14]的注释以获取更多的细节。

2. 不依赖于中心场地

本地自治意味着所有的场地都必须得到同等的对待。因此，尤其是不应该为了某些集中服务——比如，集中查询处理、集中事务管理或者是集中命名服务——而依赖于一个中心“主”场地，以至于整个系统会依赖于一个中心场地。这样第二个目标其实是第一个目标的一个推论（如果第一个满足了，第二个毫无疑问会得到满足）。但是“不依赖于中心场地”本身就是非常重要的，即使是完全的本地自治无法达到。因此，把它做为一个单独的目标提出来是很有必要的。

而之所以不接受对中心场地的依赖至少是基于以下两个原因：首先，中心场地可能会成为一个瓶颈；其次，也是更重要的，系统将会是脆弱的——如果中心场地停止运转了，整个系统也将停止下来（即单点故障（ single point of failure））。

3. 可连续操作性

通常分布式系统的一个好处是它们可以提供更大的可靠性和更高的可用性：

- 可靠性（即系统可以在任何时刻启动并运行的可能性）是得到证明了的，因为分布式系统不是一个要么做要么不做的系统——在某些单独的部分，比如说单独的场地，出现故障的时候，它们仍然可以连续地进行操作（当然是在一个被降低的水平上）。
- 可用性（即系统在某个特定的时期内能够启动并始终运行的可能性）也是得到证明了的，一方面是由于上面的理由，另一方面是由于分布式系统具有数据复制的可能性（见第 6点中的进一步讨论）。

上述讨论适用于在系统某处发生了意外停机（即出现了某种故障）的情况。意外停机显然是令人讨厌的，但却是难以阻止的。相应地，计划停机应该是永远不需要的，也就是说，应该永远也没有必要停止系统的运行来执行某个任务，比如说是类似增加一个新的场地、或是在一个现存的场地上把 DBMS升级到新版本之类的任务。

4. 位置独立性

位置独立性（也叫做位置透明性）的基本概念是很简单的：用户不需要知道数据在物理上存储在哪里，但是应该仍然可以进行操作——至少是从逻辑的角度看——就好像数据是存储在用户自己本地的场地上一样。之所以要达到位置独立性是由于它简化了用户程序和终端操作。尤其是它允许数据在场地之间迁移，而不会造成程序和操作的失效。需要这种可迁移性是为了使数据可以在网络上移动以适应改变性能的要求。

注意：你无疑会意识到这一点，即位置独立性只是通常的（物理上的）数据独立性概念在分布情况下的扩展。实际上——稍微提前说一下——我们将看到，每个列出的目标，只要其中含有“独立性”的字眼，都可以看作是数据独立性的一种扩展。对于位置独立性我们在 20.4节中还要特别地多说一点（在“目录管理”小节中关于数据对象命名的讨论中）。

5. 分片独立性

如果为了物理存储的需要，可以把给定的关系分割成小块和片断，则说这个系统是支持数据分片的。使用分片是出于性能上的考虑：数据可以在最经常被用到的地方存储，这样大部分的操作就会是本地的，而网络开销也会降低。比如，考虑雇员关系 EMP，样本数据在图20-2的上半部分。在支持分片的系统中，我们可以定义如下两个分片（参照图 20-2的下半部分）：

```
FRAGMENT EMP AS
   N_EMP AT SITE 'New York' WHERE DEPT# = 'D1'
                            OR     DEPT# = 'D3',
   L_EMP AT SITE 'London'   WHERE DEPT# = 'D2' ;
```

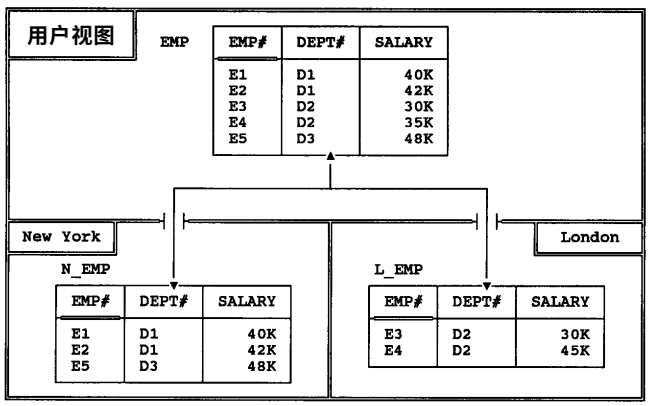

图20-2 一个分片的例子

注意：假设（a）雇员元组与物理存储之间是以某种直接方式映射的；（ b）雇员号和部门号是字符串而工资是数字；（ c）D1和D3是纽约的部门，而 D2是伦敦的部门，因此纽约雇员的元组存储在纽约的场地上，而伦敦雇员的元组存储在伦敦的场地上。系统内部的分片名称是 N_EMP和L_EMP。

分片有水平和垂直两种基本的类型，分别对应于关系操作中的选择和投影（图 20-2给出的是水平分片）。更一般地，一个分片可以由选择子操作和投影操作的任意组合来产生——这里的任意即除非有下列的情况：

- 对于选择操作而言，所有的选择操作必须构成一个第 12章所说的正交分解(12.6节)；
- 对于投影操作而言，所有的投影操作必须构成一个第 11、12章所说的无损分解。

这两条规则的最终影响是一个关系的所有分片将会是独立的，意思是说没有任何一个分片，或者是该分片的一个选择或投影，可以从其他分片中产生（如果我们真的想要在不同的地方存储相同的信息片断，可以用系统的复制机制——见下一小节）。

在分片上对原有关系的重构是通过适当的连接和合并操作来完成的（垂直分片进行连接，水平分片进行合并）。顺便提一句，可以看到在进行合并操作的时候，不需要考虑消除冗余，这要归功于上面两条规则中的第一条。

注意：在垂直分片的问题上我们要稍微精心一些。确实如上所述，这样的一个分片应该是无损的，因此，比如说对于关系 EMP，按照投影 {EMP#, DEPT#}和{SALARY}进行分片就不是有效的。但是，在某些系统中，存储的关系被认为有一个隐藏的、系统提供的“元组 ID”或是TID属性，简单地说，一个元组的 TID就是它的地址。这个 TID属性显然是这些关系的候选码。比如，如果关系 EMP包括这样一个属性，则关系按照投影 {TID,

EMP#, DEPT#}和{TID,SALARY}进行分片就是有效的，因为分片显然是无损的。同样要注意到这样一个事实， TID属性是隐藏的这一点与信息原则（ The Information Principle）并不矛盾，因为分片独立性意味着用户并不知道分片。

再提一句，易于分片和易于重构是分布式系统采用关系模式的众多原因中的两个，关系模型提供了完成这些任务所需的合适的操作 [20.15]。

现在我们得出了主要观点：支持数据分片的系统也必定要支持分片独立性（也叫做分片透明性） —— 即至少在逻辑上，用户的行为应该就像数据没有被划分的时候一样。分片独立性（和位置独立性类似）的意义在于它简化了用户程序和终端操作。尤其是，为了满足性能变化的需要，它允许数据在任何时候被划分（以及在任何时候被重新划分），而无须使任何上面所说的用户程序或操作失效。

分片独立性意味着用户将看到一个数据视图，在这个视图中各个分片是通过合适的连接和合并逻辑地重新组合在一起的。而为了满足用户的任何请求，系统优化器要负责决定应该物理地访问哪一个分片的数据。对于图 20-2所给出的分片的例子，如果用户的请求是

```
EMP WHERE SALARY > 40K AND DEPT# = 'D1'
```

则从分片的定义中（当然该定义存储在目录表里），优化器可以知道整个结果可以从纽约的场地上获得 —— 而完全不必访问伦敦的场地。

让我们更仔细地看看这个例子，用户所认识到的关系 EMP可能被（简单地）认为是底层分片 N_EMP和L_EMP的一个视图：

```
VAR EMP VIEW
    N_EMP UNION L_EMP ;
```

则优化器将用户的原始请求转化为如下形式：

```
( N_EMP UNION L_EMP ) WHERE SALARY > 40K AND DEPT# = 'D1'
```

这个表达式可以进一步被转化为：

```
( N_EMP WHERE SALARY > 40K AND DEPT# = 'D1' )
  UNION
( L_EMP WHERE SALARY > 40K AND DEPT# = 'D1' )
```

（因为选择操作对合并服从分配律）。通过目录表中分片 L_EMP的定义，优化器知道UNION操作的第二个操作数的计算结果为一个空关系（选择条件 DEPT#='D1' AND DEPT#='D2'永远也不可能为真）。这样整个表达式可以简化为只剩下：

```
N_EMP WHERE SALARY > 40K AND DEPT# = 'D1'
```

现在优化器知道只要访问纽约的场地就可以了。练习：考虑一下在处理如下请求的时候，优化器的工作会涉及哪些方面？

```
EMP WHERE SALARY > 40K
```

像在前面讨论中说到的，支持在划分关系上的操作和支持在连接、合并视图上的操作，这两者所遇到的问题有许多共同点（其实，这两个问题是同一个问题、是一样的 —— 它们只是表现在整个系统体系结构的不同方面而已）。特别是，对经过分片的关系的修改和对连接、合并视图的修改是一样的（见第 9章）。这也就同样意味着 —— 只是这样说并不严格 —— 对一个元组的修改可能会使这个元组从一个分片迁移到另一个分片，如果被修改的元组不再满足它原来所属分片的分片谓词的话。

6. 复制独立性

一个系统是支持数据复制的，如果一个存储的关系 —— 或更一般地，一个存储关系的某个分片 —— 可以由存储在许多不同场地上的不同的副本 /拷贝或复本（replica）来表示。比如（见图 20-3）：

```
REPLICATE N_EMP AS
    LN_EMP AT SITE 'London' ;
REPLICATE L_EMP AS
    NL_EMP AT SITE 'New York' ;
```

系统的内部复本叫做 NL_EMP和LN_EMP。

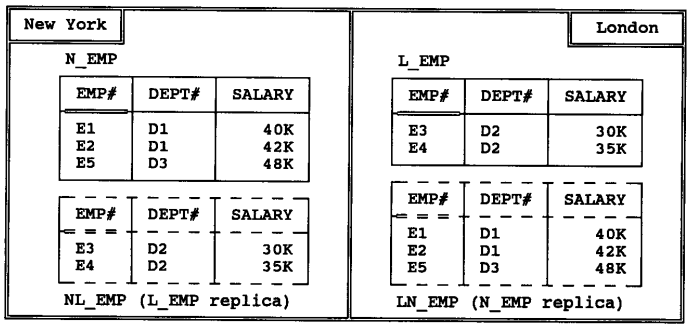

图20-3 一个复制的例子

采用复制的理由至少有两点：首先，它可以带来更优越的性能（应用程序可以在本地的副本上进行操作，而不必与远程场地通信）；第二，它还可以带来更高的可用性（一个被复制的对象总是可以用于处理的 —— 至少可以用于检索 —— 只要至少有一个副本还是可用的）。当然，复制带来的最大的问题是当一个复制对象被修改后，这个对象的所有副本必须进行修改：这就是更新传播问题。对于这个问题我们在 20.4节将会有更多的讨论。

顺便说明一下，分布式系统中的复制代表了受控冗余这个概念的一种特殊应用，受控冗余是我们在第 1章中曾经讨论过的。

现在在理想情况下，复制与分片一样，应该是“对用户透明的”。换句话说，支持数据复制的系统也应该支持复制独立性（也叫做复制透明性）—— 即至少在逻辑上，用户的行为应该就像数据实际上并没有被复制一样。之所以要提倡复制独立性（与位置独立性和分布独立性一样），是因为它简化了用户程序和终端操作；尤其是为了适应变化的需求，可以随时创建或销毁复本，而无须使任何上面所说的用户程序或操作失效。

复制独立性意味着，为了满足用户的任何请求，系统优化器要负责决定应该物理地访问哪一个复本的数据。在这里不再讨论这一问题。

作为本小节的结束，我们指出现在许多商用产品支持一种不具备完全的复制独立性的复制形式（即不是完全“对用户透明的”）。关于这一问题的进一步说明请看 20.4节中的更新传播一小节。

7. 分布式查询处理

在这个标题下有两个要点。

- 首先，来看看查询“给出供应红色零件的伦敦供应商”。假设用户是在纽约的场地

而数据是在伦敦的场地，再假设满足请求的供应商有 *n*个。如果系统是关系的，则查询将主要涉及两条消息 —— 一条是将查询请求从纽约发往伦敦，另一条是将 *n*个元组的结果集合从伦敦发回到纽约。另一方面，如果系统不是关系的，而是一次一纪录的系统，则查询将涉及 2*n*条消息 —— *n*条从纽约到伦敦要求“下一个”供应商， *n*条从伦敦到纽约返回“下一个”供应商。这个例子说明，一个关系系统的性能可能比一个非关系系统要高出若干个数量级。

- 其次，优化在一个分布式系统中比在一个集中式系统中更重要。基本的观点是，在一个像上面一样涉及多个场地的查询中，会有很多种在系统中移动数据的方法可以满足查询请求，而找到一个高效的策略是至关重要的。比如，对于一个将场地 *X*上的关系 *Rx*和场地 *Y*上的关系 *Ry*合并的查询请求来说，执行的方法可以是把 *Rx*迁移到 *Y*或是把 *Ry*迁移到 *X*或是把它们都迁移到第三个场地 *Z*上（等等）。在20.4节中有对于这一点的特别说明，它涉及前面提到过的查询（“给出供应红色零件的伦敦供应商的供应商号”）。简要概括一下这个例子，在一组特定的比较可能的假设下分析了 6种不同的查询处理策略，而响应时间最短的只要 1/10秒，最长的竟然要将近 6个小时！优化显然是至关重要的，而这也可以看作是另一个理由来解释，为什么分布式系统通常都是关系的（主要是关系的查询请求是可以优化的，而非关系的则不行）。

8. 分布式事务管理

事务管理有两个主要的方面，恢复控制和并发控制。两者在分布式环境中都需要扩展处理方式。为了解释扩展处理方式，我们需要引入一个新的术语 —— 代理。在一个分布式系统中，一个单独的事务可以涉及多个场地上的代码执行，尤其是它可以对多个场地进行修改。因此每个事务都可以看作是由许多代理组成，这里代理是指在一个场地上代表一个事务执行的进程。而系统需要知道哪两个代理是属于同一个事务的 —— 比如，很显然两个属于同一个事务的代理之间不能发生死锁。

现在先专门看看恢复控制：为了保证一个给定的事务在分布式环境中是原子的（都做或者都不做），系统必须保证这个事务的所有代理要么全部一起提交，要么全部一起回滚。通过第 14章讨论过的两阶段提交协议可以获得这样的效果。在 20.4节中将对于分布式系统的两阶段提交协议做进一步说明。

对于并发控制而言：同在非分布式系统中一样，在大多数分布式系统中并发控制主要是基于封锁的（最近，许多产品开始实现多版本控制 [15.1]；但是在实践中，传统的封锁机制看起来仍然是大部分系统的技术选择）。我们同样将在 20.4节中对这一问题进行更详细的讨论。

9. 硬件独立性

对于这个问题确实没有什么太多好说的 —— 标题已经说明了一切。现实世界中安装地的计算机包括了种类繁多的机型 —— IBM的机器、 ICL的机器、 HP的机器以及各种各样的PC和工作站，等等 —— 从而确实需要能够把所有这些系统中的数据集成起来，并给用户提供一个“单一系统映像”。因此就非常需要能够在不同的硬件平台上运行同样的 DBMS，进一步地讲，要能够让这些不同的机器做为对等的合作者参与到分布式系统中来。

10. 操作系统独立性

这个目标可以部分地看作是上一个目标的推论，它同样也确实没有过多讨论的必要。

很显然，同样的 DBMS应该不仅仅能够运行在不同的硬件平台上，而且也同样可以运行在不同的操作系统平台上 —— 包括在同样硬件平台上的不同操作系统 —— 从而这些（比如说） MVS版本的、 UNIX版本的、 NT版本的操作系统可以参与到同一个分布式系统中。

11. 网络独立性

这个目标也没有什么好说的。如果系统能够支持许多完全不同的场地，这些场地是基于完全不同的硬件、运行完全不同的操作系统的，这显然就需要系统也能够支持各种完全不同的通信网络。

12. DBMS独立性

对于这个目标，我们需要考虑放松对同构假设的限制会有什么样的结果。可以证明这个假设有一些过强了：实际上只要在不同场地上的 DBMS实体 都支持相同的界面就足够了 —— 它们并不需要是同一个 DBMS软件的拷贝。比如，如果 Ingres和Oracle都支持官方的 SQL标准，则让一个 Ingres场地和一个 Oracle场地在一个分布式系统的环境中交互是很可能的。换句话说，分布式系统可以是异构的，至少在某种程度上是这样的。

支持异构性无疑是非常必要的。事实上，现实世界中的绝大部分计算机设备不仅仅是运行着许多不同的机器和许多不同的操作系统，也同样经常运行着不同的 DBMS：而如果这些不同的 DBMS都能够在某种程度上参与到同一个分布式系统中，这将是一件非常好的事情。换句话说，理想的分布式系统应该提供 DBMS独立性。

但是，这是一个非常大的专题（在实践中也是同样重要的专题），我们只好给它专门分配一节。见后面的 20.6节。

20.4 分布式系统面对的问题

在这一节，我们要对在 20.3节中提到的某些问题进行更进一步的阐述。最重要的问题是通信网络的速度太慢 —— 至少“远距离网（ long haul）”或者广域网是这样。一个典型的广域网的数据传输速率大约是每秒 5~10KB；相反，典型的磁盘驱动器的数据传输率大约是每秒 5~10MB（另一方面，某些局域网支持与磁盘驱动器相同数量级的数据传输率）。这样产生的结果就是，在分布式系统中最重要的目标（至少在使用广域网的情况下，以及在某些使用局域网的情况下）是尽量减少对网络的利用 —— 即尽可能减少消息的数量和大小。这个目标接着会引出不少其他方面的问题 —— 以下是其中的一部分（即这里所列的并不是全部）：

- 查询处理
- 目录表管理
- 更新传播
- 恢复控制
- 并发控制

1. 查询处理

尽可能减少利用网络的目标意味着查询优化进程本身需要是分布的，查询执行进程也是同样。换句话说，整个查询优化进程将典型地由这样几个步骤构成，首先是各个参与场地上的本地优化，之后是全局优化。比如，假设场地 *X*提出一个查询 *Q*，再假设 *Q*涉及一个并操作，合并的分别是场地 *Y*上拥有一百个元组的关系 *Ry*与场地 *Z*上拥有一百万个元组

的关系 *Rz*。场地 *X* 上的优化器将选择一个全局策略来执行 *Q*；显然它应该决定把 *Ry* 移向 *Z* 而不是把 *Rz* 移向 *Y*（当然更不是把 *Ry* 和 *Rz* 都移向 *X*）。然后，一旦它决定将 *Ry* 移向 *Z*，则并操作在场地 *Z* 的实际执行过程就取决于场地 *Z* 上的本地优化器了。下面给出的、对这一点的更详细的举例说明是基于文献 [20.13] 中给出的例子，而它们也是从更早一些的由 Rothnie 和 Goodman 所写的论文 [20.33] 中引用的。

- 数据库（简化的，供应商与零件）：

```
S   { S#, CITY }  在场地 A 存储了 10 000 个元组
P   { P#, COLOR }  在场地 B 存储了 100 000 个元组
SP  { S#, P# }    在场地 A 存储了 1 000 000 个元组
```

假设每个存储的元组有 25 个字节长（200 位）。

- 查询（“给出所有供应红色零件的伦敦供应商的供应商号”）：

```
( ( S JOIN SP JOIN P ) WHERE CITY = 'London' AND
  COLOR = COLOR ( 'Red' ) ) ( S# )
```

- 中间结果的估计基数：

```
红色零件数          =   10
伦敦供应商的发货数  =   100 000
```

- 通信状态：

```
数据传输率          =   50 000 bps
访问时延            =   0.1 秒
```

现在考查以下可以用来处理这个查询的 6 种可能的策略，并利用下面的公式对每种策略 *i* 计算出其通信时间 *T*[*i*]

```
( total access delay ) + ( total date volume / data rate )
```

也就是（以秒为单位）

```
( number of messages / 10 ) + (number of bits / 50000 )
```

1) 把零件表发送到场地 A，并在场地 A 完成查询

```
T[1] = 0.1 + ( 100000 * 200 ) / 50 000
     ≈ 400 秒 (6.67 分钟)
```

2) 把供应商表和发货表发到场地 B，并在场地 B 完成查询

```
T[2] = 0.2 + ( ( 10000 + 100000 * 200 ) / 50000
     ≈ 4040 秒 (1.12 小时)
```

3) 在场地 A 先对供应商表和发货表进行连接，选择结果中供应商为伦敦的元组，然后对每个这样的元组，检查场地 B 上是否有相应的零件为红色的元组。每次这样的检查将涉及两条消息，一条查询一条响应。这些消息的传送时间与访问时延相比应该可以忽略。

```
T[3] ≈ 20000 秒 (5.56 小时)
```

4) 在场地 B 选择零件表中颜色为红色的元组，然后对每一个选出的元组顺次查找场地 A，看在发货表中是否有相应的零件与伦敦的供应商联系在一起的元组。同样每次这样的检查也涉及两条消息，这些消息的传送时间与访问时延相比应该可以忽略。

```
T[4] ≈ 2 秒
```

5) 在场地A连接供应商表和发货表，选择结果中供应商为伦敦的元组，并对结果表的S#和P#进行投影，然后将结果表发送到场地 B。在场地B上完成查询。

```
T[5] = 0.1 + ( 10000*0.200 ) / 50000
     ≈ 400 秒（6.67分钟）
```

6) 在场地B上选择零件表中颜色为红色的元组，然后把结果表发送到场地 A，在场地A上完成查询。

```
T[6] = 0.1 + ( 10*0.200 ) / 50000
     ≈ 0.1 秒
```

图20-4对结果进行了汇总。

策略	技术	通信时间
1	把P发送到A	6.67分钟
2	把S和SP发送到A	1.12小时
3	对每个伦敦的发货元组，检查零件是否为红色	5.56小时
4	对每个红色的零件，检查是否存在伦敦的供应商	2.00秒
5	把伦敦的发货表发送到B	6.67分钟
6	把红色的零件发送到A	0.10秒 (最佳)

图20-4 分布查询处理策略（小结）

几点说明：

- 这六种策略中的每一种都对问题提供了一种看似合理的解决方法，但是通信时间的差异却是巨大的（最慢的要比最快的慢二百万倍）。
- 对于一种策略的选择而言，数据传输率和访问时延都是非常重要的因素。
- 对于最差的策略而言，计算和I/O时间与通信时间相比是几乎可以忽略的（另一方面，对于比较好的策略来说，是不是这样就不一定了 [20.35]。对于快速局域网来说，这种忽略也许也是不可以的）。

此外，有些策略允许在两个场地上并行地进行处理，这样，对用户的响应时间也许在实际上会比一个集中式系统要少。但是，要注意的是我们忽略了由哪一个场地来接受最后的结果。

2. 目录表管理

在一个分布式系统中，系统目录表中不仅包括基本表、视图、权限等通常的目录数据，而且还包括许多必须的控制信息，以使得系统能够提供所需的位置独立性、分片独立性以及复制独立性。现在的问题是：目录表自己存储在哪里，怎样存储？下面是几种可能：

1) 集中式：整个目录表只在一个单独的中心场地上存储一次。

2) 完全复制：在每一个场地上都存储一个完整的目录表。

3) 分片式：每个场地只保存与本场地对象相关的目录表，整个目录表是所有这些不相交的本地目录表的并集。

4) 1与3的组合：与第三种方法一样，每个场地维护自己的本地目录表；此外，在一个单独的中心场地上维护一个所有本地目录表的统一副本，就像第一种方法那样。

上述的每一种方法都有它们自身的问题。方法 1显然与“不依赖于中心场地”的目标相矛盾；方法2会造成自治性的严重丧失，因为每次对目录表的修改都需要传送到每一个

场地上去；方法 3会使得非本地操作的代价非常高（平均下来，找到一个远程对象需要访问半数的场地）；方法 4的效率要比方法 3高很多（找到一个远程对象只需要进行一次远程目录表访问），但是它同样与“不依赖于中心场地”的目标相矛盾。因此在实践中，系统一般不使用这四种方法中的任何一种！我们会通过例子说明在 R*[20.39]中使用的方法。

为了解释 R*的目录表是如何构造的，首先需要说明一下 R*中的数据对象命名。目前数据对象命名在分布式系统中通常都是一个非常突出的问题。很有可能两个不同的场地 *X*和 *Y*都有一个叫做 *A*的数据对象，比如说是一个基本表，这就意味着需要一种机制 —— 大部分是通过场地名称的限定 —— 来进行“澄清”（即保证系统范围内的名称唯一性）。但是类似*X.A*和*Y.A*之类的限定名称如果暴露给用户，显然就会和位置独立性的目标冲突。因此，需要一种方法把用户所知道的名称映射到相应的系统所知道的名称。

下面是 R*针对这一问题的解决方法。首先 R*区分一个数据对象的外部名（printname）和系统名（system-wide name），其中外部名是指数据对象被用户正常引用时的名字（比如，在一个 SQL的SELECT语句中），而系统名是指数据对象的全局唯一内部标识。系统名有四个部分：

- 创建者 ID（creator ID）（创建这个数据对象的用户的 ID）；
- 创建者场地 ID（creator site ID）（CREATE操作被输入的场地的 ID）；
- 本地名（local name）（数据对象的非限定名称）；
- 生成场地 ID（birth site ID）（数据对象最初存储的场地的 ID）。

（用户 ID在场地上是唯一的，而场地 ID是全局唯一的）。比如，系统名

```
MARILYN @ NEWYORK . STATS @ LONDON
```

指的是一个数据对象，也许是一个基本表，它的本地名是 STATS，它由在纽约场地的用户 Marilyn创建，最初存储 ⊖ 在伦敦场地上。这个名称是保证永远不会改变的 —— 甚至在数据对象迁移到了另外一个场地上的时候也不会（见下）。

我们已经指出，用户一般通过他们的外部名来引用数据对象，构成一个外部名的是一个简单的非限定名 —— 或者是系统名的“本地名”部分（上面的例子中是 STATS），或者是系统名的一个同义词，这个同义词是通过 R*中特定的 SQL语句 CREATE SYNONYM来定义的。这里有一个例子：

```
CREATE SYNONYM MSTATS FOR MARILYN @ NEWYORK . STATS @ LONDON ;
```

现在用户既可以说（比如）

```
SELECT ... FROM STATS ... ;
```

也可以说

```
SELECT ... FORM MSTATS ... ;
```

在第一种情况下（使用本地名），系统通过假设使用所有的缺省值来推断系统名 —— 也就是，数据对象是由这个用户创建的、是在这个场地上创建的、最初也是存储在这个场地上的。顺便说一句，这些缺省假设的一个结果就是原有的系统 R的应用程序可以不加变化地在 R*上运行（即一旦系统 R的数据在 R*上进行了重新定义，要记住系统 R是R*的基础原型）。

⊖ 基本表在 R*中的物理存储实际上几乎与作者所知道的每一个系统都一样。

在第二种情况下（使用同义词），系统通过询问相关的同义词表来决定系统名。同义词表可以认为是目录表的第一个组成部分；每个场地为该场地上的每个用户维护一组这样的表，把用户所知道的同义词映射到系统名上。

在同义词表之外，每个场地还要维护：

1) 每个在该场地产生的数据对象的一个目录表表项；

2) 每个现在存储在该场地的数据对象的一个目录表表项。

假设现在用户发出一个与同义词 MSTATS有关的请求。首先系统在合适的同义词表中查找对应的系统名（这是一个纯粹的本地查找）。现在比如说它知道了生成场地是伦敦，于是它可以询问伦敦的目录表（不失一般性，我们假设这是一个远程查找 —— 第一次远程访问）。得益于上面的第一点，伦敦的目录表会包含这个数据对象的一个表项。如果数据对象仍然在伦敦，它现在就被找到了。但是，如果数据对象，比如说，已经被迁移到了洛杉矶，在伦敦的目录表表项会给出这一信息，则系统现在就可以询问洛杉矶的目录表（第二次远程访问）。得益于上面的第二点，洛杉矶的目录表将包含这个数据对象的一个表项。所以这个数据对象最多经过两次远程访问就可以被找到。

进一步讲，如果该对象被再一次迁移，比如说是到旧金山，则系统会做如下工作：

- 插入一个旧金山的表项；
- 删除洛杉矶目录表项；
- 将伦敦的表项修改为指向旧金山而不是洛杉矶。

最终还是至多通过两次远程访问就可以找到这个数据对象。而且这是一个完全的分布模式 —— 在系统中没有集中的目录表场地，从而没有单点故障。

说明一下，在 DB2的分布式数据设备中的数据对象命名模式与上面所说的相似，但是并不相同。

3. 更新传播

如同20.3节所指出的那样，数据复制的基本问题是，对于一个给定逻辑对象的修改必须传播到该对象的所有存储副本上去。立刻就会出现的一个问题是某个存储了该数据对象副本的场地可能在修改时是不可用的（由于场地故障或是网络故障）。因此最直接的策略，即将修改立即传播到所有副本上的策略，看起来是无法接受的了，因为这意味着如果当时有一个副本不可用的话，修改 —— 当然还有事务 —— 就将失败。实际上，从某种意义上说，在这种策略下数据的可用性比在无复制的情况下还要更差，因此，前面章节中所声称的数据复制的一个优点也就被削弱了。

处理这个问题的一个通常的模式（不是唯一可能的模式）是所谓的主副本模式，具体如下：

- 被复制对象的一个副本被指定为主副本。剩下的都为从属副本。
- 不同数据对象的主副本存储在不同的场地上（所以这也同样再次说明这是一个分布模式）。
- 一旦完成了对主副本的修改，修改操作就被认为是逻辑地完成了。拥有主副本的场地则要负责在某个接下来的时间里把更新传播到所有的从属副本上（但是，如果要想保持事务的 ACID属性，这个“接下来的时间”必须在 COMMIT之前。后面将有进一步的说明。）

当然，这个模式本身也会带来许多其他问题，其中的大部分已经超出了本书所讨论的范围。还要注意到，它同时还带来了与本地自治目标之间的冲突，因为在这种情况下一个事务可能会由于一个远程（主）副本的不可用而失败——尽管本地副本是可用的。

注意：如前所述，事务处理的ACID要求意味着所有的更新传播必须在相应的事务完成之前完成（“同步复制”）。但是，目前许多商用产品支持的是一种弱化形式的复制，在这种方式下修改的传播可以在某个稍后的时候（可能是某个用户指定的时候）来进行，而不必在相应事务完成的时间范围内（“异步修改”）。事实上，复制这个词已经被这些产品或多或少地篡改了，结果是——至少在商用市场上——它总是用来暗示修改的传播将延迟到相应事务的提交点之后（比如，见文献[20.1]、[20.18]和[20.21]）。显然，延迟传播方法的问题在于数据库无法再保证在任何时候都是一致的。事实上，用户甚至也许都不知道它是不是一致的。

我们以对于延迟传播方法的两点额外说明结束本小节：

1) 在一个有着延迟更新传播的系统里，复制的概念可以看做是第9章中快照想法的一种应用[⊖]。实际上，用另外一个词来形容这种复制会更好，那么我们就可以让“复本”一词保留它在一般论述中的通常含义了（即，一个准确的副本）。注意：我们并不是说延迟传播是一个不好的主意——比如在第21章我们将看到，在适当的情形下它显然是一种合适的方法。但是，最主要的是，延迟传播意味着“复本”并不是真正的复本，而系统也不是真正的分布式数据库系统。
2) 商用产品之所以用延迟传播来实现复制的一个原因（也许是主要原因）在于，替代方法——即在COMMIT之前修改所有复本——需要两阶段提交的支持（见下面），但是这会引起性能上的很大开销。这种情况说明了为什么在商业出版物中有时会碰见具有类似这种奇怪标题的文章——“复制与两阶段提交”。其之所以奇怪是因为表面上看起来，他们好像在比较两件完全不同的事情。

4. 恢复控制

如同在20.3节说明的，分布式系统的恢复控制典型地是基于两阶段提交协议的（或它的某个变体）。假如一个单独事务要和许多自治的资源管理者交互，则在任何类似这样的环境下，都需要两阶段提交协议。但是两阶段提交在一个分布式系统中尤为重要，因为所说的资源管理者——即本地的DBMS——是运行在不同的场地上，并因此是完全自治的。几点说明：

1) “不依赖于中心场地”的目标指出协调功能不可以指定给网络中某个特别的场地，而必须针对不同的事务由不同的场地来执行。通常它都是由被执行事务的启动场地来完成的，因此每个场地都必须既可以作为某些事务的协调者又作为另外一些事务的参与者（通常情况下）。
2) 两阶段提交进程需要协调者与每一个参与场地进行通信——这就意味着更多的消息传递与更大的开销。
3) 如果场地*Y*是由场地*X*协调的两阶段提交进程中的一个参与者，那么场地*Y*必须按照场地*X*所告诉它的去做（提交或是回滚，只要需要）——这是本地自治性的一种

[⊖] 除非快照被认为是只读的（除了周期性的刷新），虽然某些系统确实允许用户直接修改“复本”——比如，见文献[20.21]。当然，后面的这种能力会造成与复制独立性的冲突。

（也许是较小的）损失。

让我们回顾一下第14章所描述的最基本的两阶段提交进程。图20-5显示了在协调者与一个典型的参与者之间所进行的交互过程（为了简单起见，假设参与者是一个远程场地）。图中时间轴是从左向右的（无论长短！）。简单而言，事务要求执行一个COMMIT而不是一个ROLLBACK。在接到COMMIT请求后，协调者完成如下两阶段进程：

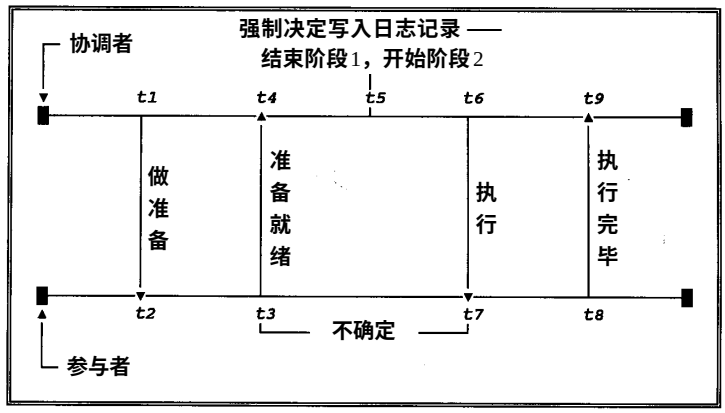

图20-5 两阶段提交

- 协调者要求每个参与者针对事务“做好完成任何请求的准备”。图20-5表明“做好准备”的消息在*t1*时刻被发出，在*t2*时刻被参与者接收到。这时参与者强制本地代理将一条日志记录写入物理日志中，然后向协调者发一条“准备就绪”的消息（当然，假如出现了任何问题——尤其是本地代理发生故障，它将发出“没有准备好”的消息）。在图中，响应消息——为简单起见被设为是“准备就绪”——是在*t3*时刻被发出的，协调者在*t4*时刻收到这一消息。注意（如同已经指出的）参与者这时丧失了一定的自治性：它必须按照接下来协调者所告诉它的去做。而且，任何被本地代理封锁的资源必须继续保持封锁，直到参与者接收到协调者的决定并依照执行。
- 从所有的参与者那里接收到响应以后，协调者将做出决定（如果所有的响应都是“准备就绪”，则决定提交，否则为回滚）。然后它强制将一条日志记录写入物理日志，记录下这一决定，时刻为*t5*。*t5*这一时刻标志着从阶段1到阶段2的过渡。
- 我们假设决定是提交。于是协调者要求所有的参与者“执行”（即为本地代理执行提交进程）；图20-5表明“执行”的消息在*t6*时刻被发出，在*t7*时刻被参与者接收到。参与者为本地代理完成提交之后，向协调者发回一条确认消息（“执行完毕”）。在图中，确认消息是在*t8*时刻被发出的，协调者在*t9*时刻收到这一消息。
- 当协调者接收到了所有的确认消息，整个进程就结束了。

当然，在实际情况中整个进程肯定比刚才所描述的要复杂得多，因为必须考虑到场地和网络故障的情况。比如，假设协调者场地在*t5*与*t6*之间的某个时刻*t*出现了故障，则在场地恢复之后，通过日志，重新启动的过程将发现有一个事务正处在两阶段提交进程的第二个阶段，于是将通过向所有的参与者发出“执行”的消息来继续这一进程（需要注意的是在该事务中，参与者在*t3*到*t7*期间是处在“不确定”状态下。如果协调者确实如所说的是在时刻*t*发生故障的，则这个“不确定”的状态将持续很长的一段时间）。

当然在理想的情况下，我们希望两阶段提交进程对于任何可能的故障都保持弹性。不幸的是，很显然这一目标是从根本上无法达到的——即面对任意可能的故障，不存在任何有限次的协议可以保证所有的参与者能够一致地提交一个成功的事务或是一致地回滚一个不成功的事务。反过来说，假设存在这样一个协议，则 *N*是这个协议所需要的最少的消息数目。现在再假设由于某种故障，这 *N*个消息的最后一个消息丢失了，则要么这个消息是不必要的，这与 *N*为最少数目的假设相矛盾；要么协议无法生效。任何一种可能都会引出矛盾，从而可以推出不存在这样的协议。

尽管这一事实让人不快，但是从改进性能的角度来看，至少还是可以对基本算法进行许多不同的加强的：

- 首先，如果在某个参与者场地上的代理是只读的，则这个参与者在第一阶段可以发出“忽略我”的响应消息，而协调者就可以在第二阶段真正地忽略它。
- 第二，如果所有的参与者在第一阶段的响应消息都是“忽略我”，则第二阶段就可以被完全地跳过。
- 第三，基本模式有两个很重要的变体叫做假想提交和假想回滚[20.15]，将在接下来的段落里对它们进行更详细的讨论。

一般而言，假想提交模式具有在成功的情况下减少所需消息数量的效果，而假想回滚则可以在失败的情况下减少所需消息的数量。为了解释这两种模式，首先要注意在上面所叙述的基本机制中，协调者要始终记住它的决定直到它从每一个参与者那里都接收到了一个确认消息。当然，原因是如果一个参与者在“不确定”的状态下崩溃了，则在重新启动的时候它必须要询问协调者以获知协调者的决定是什么。但是，如果所有的确认消息都被接收到了，协调者就知道所有的参与者都已经按照被告知的执行了，那么它就可以“忘记”这个事务了。

我们现在来看看假想提交。在这种模式下，要求参与者确认“回滚”（“不执行”）消息而不需要确认“提交”（“执行”）消息。而且如果决定是“提交”，协调者可以在广播了它的决定之后立即忘记这个事务。如果一个不确定的参与者崩溃了，则在重启动的时候它将（像通常一样）询问协调者；如果协调者仍然记着（存储着）这个事务（即协调者仍然在等待参与者的确认消息），那么决定肯定是“回滚”，否则肯定是“提交”。

当然，假想回滚正好相反：参与者被要求响应“提交”消息而不是“回滚”消息，而协调者可以在广播了它的决定之后忘记这个事务，只要决定是“回滚”。如果一个不确定的参与者崩溃了，则在重启动的时候它将询问协调者；如果协调者仍然记着（存储着）这个事务（即协调者仍然在等待参与者的确认消息），那么决定肯定是“提交”，否则肯定是“回滚”。

有趣的是（也是有些有违直觉的是）假想回滚似乎比假想提交更可取（我们说它“有违直觉”是因为绝大部分事务肯定是成功的，而假想提交可以减少成功情况下消息的数目）。假想提交的问题如下。假设协调者在第一阶段崩溃了（即在它做出决定之前），则在协调者场地重启动的时候，事务将回滚（因为它没有完成）。接下来，某个参与者询问协调者关于这个事务的决定，而协调者已经不记得这个事务了，那么就将假定决定是“提交”——这毫无疑问是不对的。

为了避免这种“错误提交”，协调者（如果采用假想提交的话）必须在开始第一阶段

的时候向它的物理日志中强制写入一条日志记录，给出事务中所有的参与者（现在如果协调者在第一阶段崩溃了，则在重启动的时候它将向所有的参与者广播“回滚”消息）。而对协调者日志的这一次物理I/O对每一个事务都是关键所在，于是假想提交就不像它给人的第一印象那样吸引人了。实际上，可以不夸张地说，至少在本书写作的时候，假想回滚已经是现有的实现系统事实上的标准。

5. 并发控制

如同在20.3节所说明的，绝大部分分布式系统中的并发控制都是基于封锁的。但是在一个分布式系统中，测试、设置以及释放封锁的请求都变成了消息（假设所考虑的数据对象在一个远程场地上），而消息就意味着开销。比如，一个事务 T要修改一个数据对象，而这个对象在 n个远程场地上都存在复本。如果每个场地都负责对存储在该场地上的对象进行封锁（就像在本地自治假设下所做的一样），则最直接的实现方式需要至少 $5n$条消息：

n条封锁请求
n条封锁授权
n条修改消息
n条确认消息
n条解锁请求

当然我们可以很容易地通过“搭载”消息来改进上述过程——比如，封锁请求和修改消息可以进行合并，封锁授权和确认消息也是同样——但即使是这样，修改所需要的全部时间也会比在一个集中式系统中要高出好几个数量级。

通常解决这一问题的方法是修改在上面“更新传播”一小节中提出的主副本策略。对一个给定的数据对象 A，拥有 A的主副本的场地将处理所有有关 A的封锁操作（要记住，一般情况下不同对象的主副本存储在不同的场地上）。在这一策略下，针对封锁而言，一个对象的所有副本的集合可以看作是一个单一的对象，而消息的总数也将从 $5n$减少到 $2n+3$（一条封锁请求、一条封锁授权、 n条修改消息、n条确认消息以及一条解锁请求）。但是需要再次注意的是，这个解决方案会带来自治性的（严重）丧失——如果一个主副本不可用了，一个事务就将失败，即使这个事务是只读的，而且有一个本地副本是可用的（注意不仅修改操作需要封锁主副本，而且检索操作也需要封锁主副本[20.15]，因此主副本策略的一个不好的副作用就是，它会降低检索和修改的性能和可用性）。

在一个分布式系统中进行封锁的另外一个问题是它会导致全局死锁。全局死锁是涉及两个或者多个场地的死锁。比如（参见图 20-6）：

1) 事务 $T2$在场地 X上的代理正在等待事务 $T1$在场地X上的代理释放一个锁；

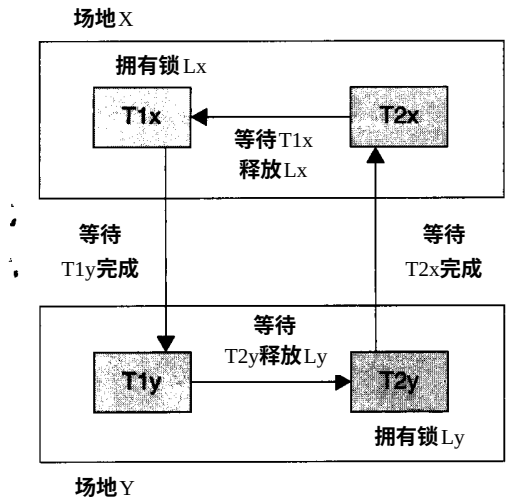

图20-6 一个全局死锁的例子

2) 事务*T1*在场地*X*上的代理正在等待事务*T1*在场地*Y*上的代理完成操作；

3) 事务*T1*在场地*Y*上的代理正在等待事务*T2*在场地*Y*上的代理释放一个锁；

4) 事务*T2*在场地*Y*上的代理正在等待事务*T2*在场地*X*上的代理完成操作：死锁！

像这样的一个死锁问题是任何一个场地只用该场地内部的信息都无法检测出来的。换句话说，在本地的等待图中并没有回路，但是，如果两个本地图被合成了一个全局图，就会出现一个回路。这就意味着全局死锁检测会带来进一步的通信开销，因为它要求独立的本地图以某种方式被集中到一起。

在关于R*的论文中描述了一个漂亮的（也是分布的）模式用于全局死锁检测（例如，见文献[20.39]）。注意：在第15章我们曾指出，在实际中并不是所有的系统都进行死锁检测——有些只是用超时机制来替代。由于显而易见的原因，这种方式对于分布式系统而言是尤为实际的。

20.5 客户/服务器系统

在20.1节中我们曾提到，客户/服务器系统可以看作是普遍意义上的分布式系统的一种特殊情况。更确切地说，一个客户/服务器系统是这样一个分布式系统：(a) 某些场地是客户场地，而某些场地是服务器场地；(b) 所有的数据都驻留在服务器场地；(c) 所有的应用都在客户场地运行；(d)“存在接缝”(不提供完全的位置独立性)。参见图20-7(即第2章中的图20-5)。

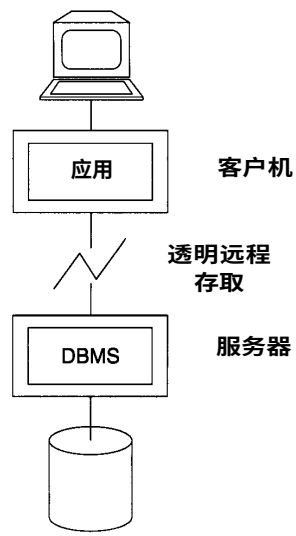

图20-7 一个客户/服务器系统

在写作本书的时候，可以说客户/服务器系统中蕴含着巨大的商业利益，而在真正的一般意义上的分布式系统中商业利益则相对少得可怜(虽然在下一节中我们将看到这种情况会有某种程度的改变)。我们依旧继续相信真正的分布式系统代表着一种重要的长期趋势，这就是我们为什么要在这一章里关注这类系统的原因，但是很显然专门针对客户/服务器系统进行一番讨论也是很需要的。

回忆一下，在第2章中，“客户/服务器”这个词主要是指一种体系结构，或是对职责的逻辑划分。客户是指应用(也叫做前端)，而服务器是指DBMS(也叫做后端)。但是恰恰是由于整个系统可以被清晰地划分为两个部分，也就有了将两个部分运行到不同的机器上的可能。而后面的这种可能性是如此的吸引人(原因众多——见第2章)，以至于“客户/服务器”这个词几乎只被应用于客户和服务器的确运行在不同的机器上的情况[⊖]。这种用法是普遍的，但也是草率的，将在下面对它进行修正。

我们提醒你在基本模式之上还可能存在许多变体：

- 许多客户也许可以共享同一个服务器(实际上，这是正常的情况)。

⊖ “两层系统”在本质上也被用来表达同样的意思(原因显而易见)。

- 一个单一的客户也许可以存取许多服务器，这种可能性又可以分为两种情况：
  a. 限制客户一次只可以存取一个服务器 —— 即每个独立的数据库请求必须只针对一个服务器，不可能出现在一个单一的请求中同时包含来自两个或多个不同服务器的数据的情况。而且用户必须要知道哪一个服务器存储着那一部分数据。
  b. 客户可以同时存取多个服务器 —— 即一个单一的数据库请求可以包含多个服务器的数据，这意味着在客户看来这些服务器好像实际上只是一个服务器，而用户也不须要知道哪一个服务器存储着那一部分数据。

  但是这里的情况 b实际上是一个真正的分布式系统（“ 隐藏了接缝 ”），它并不是通常情况下“ 客户 /服务器 ”的含义，只是我们在下面将忽略这一点。

1. 客户/服务器标准

在客户 /服务器处理中有许多生效的标准：

- 首先，一些客户 /服务器的特征被包括在 SQL标准SQL/92[4.22]中，对于这些特征的讨论将放到 20.7节中。
- 接下来还有ISO的远程数据存取标准，RDA[20.26~20.27]。RDA之所以重要的一个原因是某些与之非常相近的方法已经由 SQL Access Group（SAG）的成员实现了，SAG是那些承诺开放系统和协同工作的数据库软件提供商的联盟。注意：对我们而言，无需费劲地去区分ISO和SAG版本的RDA，一般地，我们用“ RDA ”这个名字来同时代表这两者。

  RDA的目的是为客户 /服务器通信定义格式和协议。它假设（ a）客户用标准形式的 SQL（基本是 SQL/92标准的一个子集）来表达数据库请求；同时（ b）服务器支持一个标准的目录表（同样遵循 SQL/92标准的定义）。然后它定义了一个在客户与服务器之间传递消息（ SQL请求、数据和结果以及分析信息）的特定的表达格式。
- 我们在这里要提到的第三个也是最后一个标准是 IBM的分布式关系数据库体系结构（DRDA）标准[20.25]（它是一个事实上的标准，而不是一个理论上的标准）。DRDA和RDA有着类似的目标，但是 DRDA在许多重要的方面与 RDA不同 —— 尤其是它倾向于反映出它的 IBM“ 血统 ”。比如， DRDA并不假设客户端使用的是标准版本的 SQL，而是代之以允许任何形式的 SQL。一个（可能的）后果就是获得更好的性能，因为客户端有可能利用某些服务器所特有的功能；另一方面这又会带来可移植性的损失，恰恰是因为那些服务器所特有的功能对客户端是开放的（即客户端知道与之对话的是哪一种服务器）。基于同样的想法， DRDA并不假设在服务器端有任何特定的目录结构。 DRDA的格式和协议与 RDA的非常不同（关键是， DRDA是基于IBM自己的体系结构和标准，而 RDA是基于国际的、与供应商无关的标准）。

关于RDA与DRDA进一步的细节已经超出了本书的范围，在文献 [20.23]和[20.30]中可以找到关于它们的一些分析和比较。

2. 客户/服务器应用程序编程

我们曾经说过，客户 /服务器系统是普遍意义上的分布式系统的一种特殊情况。如同在本节开始的引言中所说，一个客户 /服务器系统可以看作是这样一个分布式系统，所有的请求都源自于一个场地而所有的处理都在另一个场地上（为了简单起见，假设只有一个客户场地和一个服务器场地）。注意：显然在这个简单的定义下，一个客户场地完全不是一个真正的“ 数据库系统的场地，仅凭其自身而言 ”。因此，这种系统与在 20.2节中所

给出的一般意义上的分布式系统的定义相抵触（客户场地同样可以有其本地的数据库，但是这些数据库并不直接构成这种客户/服务器方案中的一部分）。

尽管如此，客户/服务器方法在应用程序编程中确实有其特定的实现方式（实际上一般的分布式系统也是一样）。在所有最重要的方面中，有一点我们已经在20.3节中关于目标7（分布式查询处理）的讨论中接触过了：也就是通过定义和设计，我们知道关系系统是集合层次的系统。在一个客户/服务器系统中（而且实际上一般地，在分布式系统中）比以往更为重要的是，应用程序员不仅仅“像使用一种存取方法那样使用服务器”，并编写针对记录层次的代码，而是要把尽可能多的功能捆绑进针对集合层次的请求中——否则执行性能将受到影响，这种影响是由执行所涉及的消息数量带来的（在SQL术语中，上面的表述意味着要尽可能避免使用游标——即避免FETCH循环和UPDATE与DELETE的CURRENT形式，见第4章）。

如果系统提供某种存储过程机制，在客户和服务器之间的消息还可以进一步减少。一个存储过程本质上是一个存储在服务器场地上的（并且是可以被服务器识别的）、预编译的程序，它通过远程过程调用（RPC）被客户执行。因此，对由于进行记录层次的处理所造成的性能损失，可以通过建立一个合适的存储过程，并在服务器场地上直接运行它来部分地抵消。

注意：尽管这与我们关于客户/服务器处理的论题有些偏离，还是应该指出存储过程的好处不仅仅在于提高性能，而且还包括：

- 这种过程还可以用来对用户隐藏各种与DBMS以及数据库相关的细节，从而与其他情况相比可以提供更大程度的数据独立性。
- 一个存储过程可以被许多客户共享。
- 可以在创建存储过程的时候，而不是执行它的时候进行优化（当然这一优点是相对那些通常进行运行时优化的系统而言的）。
- 存储过程可以提供更好的安全性。比如，一个给定的用户可以被授权执行一个给定的过程，但是却无法对该过程所存取的数据进行直接的操作。

一个不足之处在于不同的供应商在这方面会提供非常不同的功能软件，尽管——如同在第4章所提到的——在1996年已经对SQL标准进行了扩展，以包括某些存储过程支持（在持久存储模块（Persistent Stored Method）名下，PSM）。

20.6 DBMS独立性

现在回到对于一般分布式系统的十二个目标的讨论上来。大家应该记得，最后的一个目标是DBMS独立性。在20.3节对于这些目标的简短讨论中我们已经表明，严格的同构性假设是太强了，实际上所需要的只是在不同场地上的DBMS具有同样的接口。就像我们在这一节所说的：比如，如果Ingres和Oracle都支持官方的SQL标准——不多也不少！——则就有可能让它们在一个异构的分布式系统中充当同等的合作者（事实上，通常正是这种可能性会首先被提出来，做为支持SQL标准的理由之一）。我们将从考虑这种可能性的细节开始。注意：把讨论建立在Ingres和Oracle的基础之上，只是为了使事情更有针对性一些。而这些所讨论的概念毫无疑问是普遍适用的。

1. 通道

假设有两个场地 *X*和 *Y*分别运行 Ingres和Oracle系统，而与此同时在场地 *X*上的用户 *U*希望看到一个单一的分布式数据库，它包括来自于场地 *X*上Ingres数据库的数据和来自于场地*Y*上Oracle数据库的数据。通过定义，用户 *U*成为一个 Ingres用户，因此对用户而言，这个分布式数据库必须是一个 Ingres数据库。这样， Ingres而不是 Oracle就有责任提供所需要的支持。那么这种支持包括那些方面呢？

在原则上，这种支持非常简单明了： Ingres必须提供一个特殊的程序 —— 一个通常所说的通道(gateway) —— 其效果是“使 Oracle看起来像 Ingres一样”。参见图 20-8 ⊖。通道可能运行在 Ingres场地上或者 Oracle场地上或者（如图所示的那样）是在两个场地之间的某个通道自己的特殊场地上。但是不论它在哪里运行，它至少必须明确地提供下述的所有功能（可以发现其中的许多功能会带来具有重要特征的实现问题。但是，在 20.5节所讨论的RDA和DRDA标准确实可以解决其中的一些问题）。

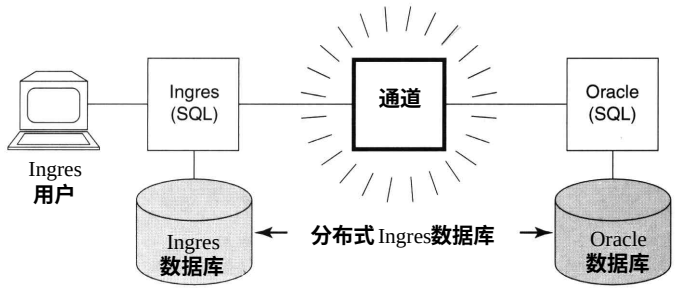

图20-8 一个假想的 Ingres提供的针对 Oracle的通道

- 实现在 Ingres和Oracle之间的信息交换协议 —— （与其他协议相比）它包括把从 Ingres发出的源 SQL语句的消息格式映射为 Oracle所要求的格式，并把由 Oracle发出的结果的消息格式映射为 Ingres所要求的格式。
- 为Oracle提供一种“关系服务器”的功能（类似于已经在大部分 SQL产品中实现的交互式SQL处理器）。换句话说，通道必须能够在 Oracle数据库上执行任意的 SQL语句。为了提供这一功能，通道必须利用对动态SQL的支持或者（更有可能的）是利用 Oracle场地上所提供的 SQL/CLI或ODBC之类的调用层接口（见第 4章）。注意：除此之外，通道也可以直接利用由 Oracle提供的交互式 SQL处理器。
- 在Ingres和Oracle之间进行数据类型映射。这个问题包括一系列与其他事情有关的子问题，这里所说的其他事情包括处理器之间的差异（比如，不同的机器字长）、字符代码之间的差异（对字符串的比较以及 ORDER BY请求的反应会给出非预期的结果）、浮点格式之间的差异（一类让人极其反感的问题）、对日期和时间支持上的差异（就作者所知，目前没有哪两个 DBMS在这方面提供了相同的支持），等等。对于这些事情的讨论可以参见文献 [20.15]。
- 将Ingres的SQL语言映射成 Oracle的SQL语言 —— 因为实际上 Ingres和Oracle都没有完全，即不多也不少地，支持 SQL标准。事实上它们都是只支持其中的某一部分功能，

⊖ “三层系统”一词有时（原因显而易见）被用来表示图中所示的安排方案，以及其他一些类似的涉及三个组成部分的软件配置方案（请特别参看下一小节关于“中间件”的讨论）。

而对另一些则不支持，同时还有些功能在两个产品中有同样的语法但却有着不同的语义。注意：在这种连接中，确实有些通道产品提供了一种传递转移的机制，利用这种机制，用户可以（比如说）直接用目标系统的语言书写一个查询，然后把它不作修改地通过通道传递到目标系统上进行执行。

- 将Oracle的反馈信息（返回代码之类的）映射为 Ingres的格式。
- 将Oracle的目录表映射为 Ingres的格式，从而使得 Ingres场地和在该场地上的用户可以知道在 Oracle数据库中有些什么。
- 处理在异构系统之间很容易出现的各种语义不匹配的问题（可以参见文献 [20.9]、[20.11]、[20.16]和[20.38]）。这可能会包括命名的差异（在 Ingres中可能使用 EMP#，而在 Oracle中可能使用 EMPNO）；数据类型的差异（在 Ingres使用字符串的地方在 Oracle中可能使用数字）；单位的差异（在 Ingres中可能使用厘米，而在 Oracle中可能使用英寸）；信息逻辑表达方式的差异（ Ingres可能忽略在 Oracle中使用空值的元组）；还有很多很多。
- 作为一个两阶段提交协议（ Ingres变体）的参与者（假设 Ingres的事务可以在 Oracle数据库上执行修改操作），通道是否能够真正执行这一功能取决于在 Oracle场地上的事务管理器的能力。值得指出的是，在本书写作的时候，商用的事务管理器（除了某些例外）基本都不提供这方面所需要的功能 —— 也就是让应用程序具有可以发出指令要求事务管理器“准备结束”的能力（而是相反，只具有发出指令要求事务管理器结束，即无条件提交或者回滚的能力）。
- 在Ingres需要的时候，保证 Ingres要求在 Oracle场地上进行封锁的数据能够确实得到封锁。同样，通道是否能够真正执行这一功能也基本取决于 Oracle的封锁结构是否与Ingres的相匹配。

至此我们只是讨论了在关系系统环境下的 DBMS独立性，那么其他的非关系系统呢？—— 即在一个本来的关系分布式系统中加入一个非关系的场地的可能性是多少呢？比如，是否可能提供从一个 Ingres或Oracle场地到一个 IMS场地的访问？同样，这样的能力在实践中是非常需要的，因为现在有大量的数据存储在 IMS以及其他在关系系统出现以前就已经存在的系统中 ㊀。但是这可以做到吗？

如果这个问题的意思是“它可以 100%的做到吗？”—— 即“所有的非关系数据可以通过一个关系接口进行访问，并在其上执行所有的关系操作吗？”—— 那么答案绝对是不可以，在文献 [20.16]中对此有详细的解释。但是，如果问题的意思是“可以提供一些有用的完成这样功能的工具吗？”，则答案显然是可以。不过这里不是给出详细讨论的地方，进一步的说明可以参见文献 [20.14~20.16]。

2. 数据存取中间件

上面一小节所描述的通道（有时更确切地叫做点对点通道）其实会受到许多局限。其中之一就是它几乎没有提供位置独立性，还有就是同一个应用程序可能需要利用许多不同的通道 —— 比如一个是针对 DB2的、一个是针对 Oracle的、一个是针对 Informix的 —— 而却不提供任何（比如说）对于跨越所有此类场地的连接操作的支持。这种情况的一个

㊀ 一般认为还有大约 85%的数据驻留在这样的系统中（即关系系统出现之前的系统中，甚至是文件系统中），而同时没有任何迹象表明用户要把这些数据转移到新系统中去。

后果就是（尽管有上面一小节中所提到的技术困难），功能愈加复杂的通道类产品在过去的几年中出现得越来越频繁了。事实上，整个所谓的中间件（也叫做中介）的业务现在从其自身角度而言，已经成为一个非常引人注目的产业。所以也许并不奇怪，“中间件”这个词还没有得到精确的定义：对于在不同程度上一起工作的不同系统而言，任何用于掩盖其间差异的软件——比如一个TP监控器——都有理由被认为是一个“中间件”[20.3]。但是在这里我们只关注可以叫做数据存取中间件的软件。在这类产品中有 Cohera公司的Cohera、IBM 公司的DataJoiner以及Sybase公司的OmniConnect和InfoHub。通过图中的说明，我们简要地介绍一下 DataJoiner产品[20.7]。

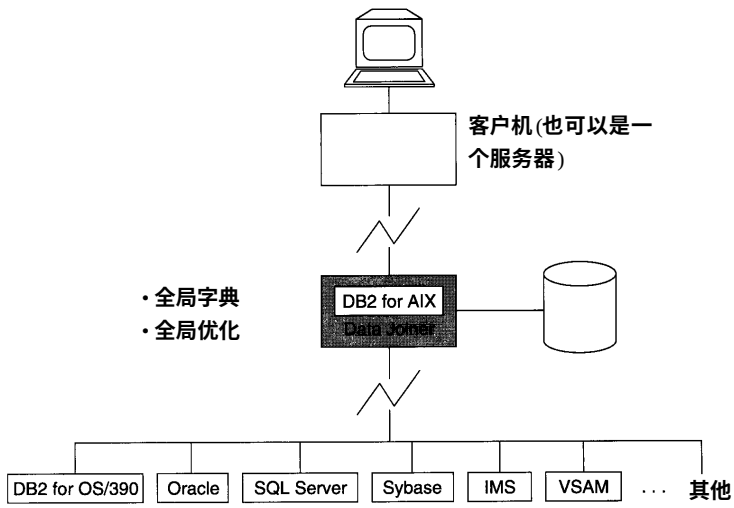

图20-9 作为数据存取中间件的 DataJoiner

DataJoiner有许多不同的配置方法（见图 20-9）。从一个独立的客户的角度而言，它看起来像一个正规的数据库服务器（即一个 DBMS），它存储数据、支持（ DB2类型的）SQL查询、提供一个目录表、执行查询优化，等等（实际上 DataJoiner的核心是 IBM的DBMS产品——DB2的AIX版本）。但是数据主要不是存储在 DataJoiner场地上（虽然提供了这样的功能），而是存储在幕后的任意数量的其他场地上，这些场地由许多不同的DBMS控制着（或者甚至是类似 VSAM这样的文件管理系统）。这样， DataJoiner向用户提供了一个有效的虚拟数据库，这个数据库是所有那些“幕后的”数据存储的联合体，它允许进行跨越这些数据存储的查询 ⊖，并利用其对于这些幕后系统的性能（以及网络特点）的了解来决定“全局最优的”查询计划。

注意：DataJoiner还具有模拟能力，能够在不直接支持某些特定的 DB2 SQL功能的系统上对这些功能进行模拟。在游标声明时的 WITH HOLE选项可以作为这样的一个例子。

直到现在我们所描述的系统仍然不是一个完全的分布式系统，因为在幕后的各个场地之间并不知道彼此的存在（即在协同的操作中它们不能被认为是同等的合作者）。但是，如果有任何新的“幕后”场地被添加进来，它同样可以作为一个客户场地，并通过

⊖ 重点是在“查询”的能力上，而修改的能力需要做一些限制，尤其是 (但不仅仅是)当幕后系统是比如IMS或其他的非 SQL系统的时候 (参见文献 [20.16])。实际上，在写作本书的时候， DataJoiner只支持跨越DB2、Oracle、Sybase和Informix场地的修改事务 (遵循两阶段提交)。

DataJoiner执行对某个或者所有其他场地进行访问的查询。整个系统因此构成一个常说的联邦系统，也叫做多数据库系统[20.19]。一个联邦系统是一个接近于完全本地自治的分布式系统（通常是异构的）。在这样的一个系统中，纯粹的本地事务由本地的 DBMS来管理，但是全局事务就是另外一回事了 [20.8]。

DataJoiner对每一个“幕后的”系统都有一个内置的驱动部件 —— 从功能上说就是上一小节中的点对点通道（这些驱动部件基本上都是利用 ODBC来访问远程的系统）。它同时还维护一个全局目录表，用来（尤其是）在遇到系统之间语义不匹配的时候告诉它应该如何处理。

我们注意到，类似 DataJoiner这样的产品对于第三方软件供应商来说是非常有用的。他们可以进行通用的工具开发（比如报表生成器、统计包，等等），而不必考虑有可能运行这些工具的不同的 DBMS产品之间的差异。

3. 小结

很显然，要提供完全的 DBMS独立性会有不少突出的问题，即使是所有参与的 DBMS都是专门的 SQL系统。但是潜在的效益同样也是巨大的，即使解决方案并不是完美的。也正是因为如此，才出现了许多数据存取的中间件产品，而且在不久的将来肯定还会有更多的产品出现。但是要提醒你的是解决方案肯定不可能是完美的 —— 尽管供应商们所说的恰恰相反。

20.7 SQL的支持

目前，SQL根本没有提供任何对于真正的分布式系统的支持。当然，在数据操纵方面不需要任何支持 —— （就用户而言）一个分布式系统最重要的是所有的数据操纵能力应该是不变的。但是对诸如 FRAGMENT、REPLICATE之类的数据定义操作，这种支持是需要的[20.15]，然而现在却没有得到提供。

另一方面，SQL又确实支持某些客户 /服务器的功能，特别是包括用来建立和断开客户/服务器连接的 CONNECT和DISCONNECT操作。实际上，一个 SQL应用在可以完成任何数据库请求之前都必须先通过执行一个 CONNECT操作来建立与服务器的连接（虽然这个CONNECT可能是隐式的）。一旦连接已经建立了，应用 —— 即客户 —— 就可以像通常一样执行 SQL请求了，而服务器将进行所需要的数据库处理。

SQL还允许已经和一个服务器建立连接的客户与另一个服务器连接。建立第二个连接会使第一个连接进入休眠状态，接下来的 SQL请求就由第二个服务器来处理，直到客户要么（a）重新和上一个服务器连接（通过另外一个新的操作， SET CONNECTION）；要么（b）再和另外一个服务器进行连接，从而使第二个服务器进入休眠状态（依此类推）。换句话说，在任何时候一个给定的客户都只能有一个活动的连接和若干个休眠的连接，而由该客户发出的所有数据库请求都会发到与之处于活动连接的服务器上，并由其处理。

注意：SQL标准还允许（但是并不要求）实现对多服务器事务的支持。也就是说，在一个事务中间，客户可以从一个服务器转向另一个服务器，从而使事务的一部分在一个服务器上执行而另一部分在另一个服务器上执行。尤其要注意的是如果允许修改操作以这种方式跨越服务器的话，执行必须假定支持某种两阶段提交，以提供标准所要求的事务的原子性。

最后，一个给定客户所建立的每一个连接（不管当前是活动的还是休眠的）最终都必须由一个合适的 DISCONNECT操作来切断（虽然像相应的 CONNECT一样，DISCONNECT在简单的情况下也许是隐式的）。

要了解更多的内容，请参看 SQL标准本身 [4.22]或者文献 [4.19]中的内容。

20.8 小结

在这一章里，我们给出了一个关于分布式数据库系统的简要讨论。用分布式系统的“十二个目标”作为组织讨论的基础，虽然这些目标并不是在所有情形下都是关联在一起的。还简要地考察了一些领域的技术问题，这些领域包括查询处理、目录表管理、更新传播、恢复控制和并发控制。还特别讨论了为了满足 DBMS独立性目标所涉及的问题（在20.6节对于通道和数据存取中间件的讨论）。我们又认真地探讨了目前在市场上非常流行的客户/服务器处理，它可以看作是一般意义下分布式处理的特殊情况。特别对 SQL中与客户/服务器处理有关的方面进行了汇总，并且着重指出用户要避免针对记录层次编码（在SQL术语中叫游标操作）。还简要描述了存储过程和远程过程调用的概念。

注意：有一个始终没有讨论的问题是分布式系统的（物理）数据库设计问题。实际上，即使忽略分片以及复制的可能性，决定哪些数据应该存储在哪些场地上的问题 —— 所谓的分配问题 —— 也是一个非常棘手的难题 [20.33]。对于分片和复制的支持只会使情况进一步复杂。

值得一提的另外一点是那些在市场上逐渐显露头角的所谓的海量并行计算机系统（见第17章对文献 [17.58]的注释）。这类系统基本上都是由大量通过高速总线连接在一起的独立的处理器构成的，每个处理器都有自己的存储器和磁盘驱动器，并且运行自己的 DBMS软件，整个数据库分布在全部所有的磁盘上。换句话说，这样一个系统本质上是“在一个盒子里”构成了一个分布式系统！—— 而在这一章中所讨论的（比如）关于查询处理策略、两阶段提交、全局死锁等所有的问题，都可以对应到这类系统中。

作为结论，我们指出，把分布式数据库的“十二个目标”（或者至少包括目标 4、5、6和8的某些子集）放在一起，它们看起来和 Codd所说的关系 DBMS的“分布独立性”准则是等价的。为了便于对照，我们把该准则叙述如下：

- 分布独立性（Codd）：“一个关系 DBMS要具有分布独立性…… [就是说]DBMS要有一种数据子语言保证应用程序和终端操作能够保持在逻辑上没有损失：
  1) 在第一次引入数据分布的时候（如果本来安装的 DBMS只管理非分布的数据）；
  2) 在数据被重分布的时候（如果 DBMS管理的是分布的数据）。”

最后要注意（如同在本章早些时候提到的）目标 4~6以及 9~12 —— 即所有在名字中包括“独立性”一词的目标，它们可以看做是类似数据独立性之类的概念 —— 在应用到分布式环境的时候 —— 的扩展。其实它们的目的都在于对应用投资的保护。

练习

20.1 定义位置独立性、分片独立性和复制独立性。

20.2 为什么分布式数据库系统几乎总是关系的？

20.3 分布式系统的优点是什么？不足又是什么？

20.4 解释下列术语：
主副本修改策略
主副本封锁策略
全局死锁
两阶段提交
全局优化

20.5 描述R*的数据对象命名模式。

20.6 一个成功的点对点通道取决于能否很好地协调（尤其是）它所涉及的两个 DBMS之间的差异。任选两个你熟悉的 SQL系统，尽可能多地指出它们之间的差异，包括语法和语义之间的差异。

20.7 调查任意一个你可以接触到的客户 /服务器系统，它支持显式的 CONNECT和DISCONNECT吗？它支持 SET CONNECTION或其他“连接类型”的操作吗？它支持多服务器事务吗？它支持两阶段提交吗？它用于客户 /服务器通信的格式和协议是什么？它支持什么样的网络环境？它支持什么样的客户与服务器硬件平台？它所支持的软件平台（操作系统、 DBMS）又是什么？

20.8 调查任意一个你可以接触到的 SQL DBMS，这个DBMS支持存储过程吗？如果支持的话，存储过程是如何创建的？它们又是如何被调用的？它们是用什么语言写的？它们是否支持完整的 SQL？它们是否支持条件分支结构（ IF-THEN-ELSE）？它们是否支持循环？它们如何向客户返回结果？一个存储过程可以调用另一个存储过程吗？如果是在另外一个场地上的存储过程呢？存储过程是做为调用它的事务的一部分吗？

参考文献和简介

20.1 Todd Anderson, Yuri Breitbart, Henry F. Korth, and Avishai Wool: “Replication, Consistency, and Practicality: Are These Mutually Exclusive?”, Proc. 1998 ACM SIGMOD Int. Conf. on Management of Data, Seattle, Wash. (June 1998).

这篇论文描述了异步（文章中称为懒惰式）复制模式中的三种模式，这些异步复制模式可以确保事务的原子性和全局可串行性而不需要利用两阶段提交。文章中还给出了有关它们的性能比较的模拟研究报告。在文献 [20.21]中提出的全局封锁是第一种模式；另外两种 —— 其中一个是悲观式的，另一个是乐观式的 —— 利用了一张复制图。这篇论文的结论是复制图模式的性能要明显地优越于封锁模式，而且“通常是大大地优于”。

20.2 David Bell and Jane Grimson: *Distributed Database Systems*. Reading, Mass.: Addison-Wesley (1992).

这是众多分布式系统的教科书中的一本（另外两本是文献 [20.10]和[20.31]）。这本书的一个显著的特点是包含了一个关于保健网络的扩展案例研究。而且在论调上也要比另外两本实用一些。

20.3 Philip A. Bernstein: “Middleware: A Model for Distributed System Services,” *CACM 39*, No. 2 (February 1996).

“对各种不同的中间件进行了归类，描述了它们的特性，揭示了它们的演化过程，并为理解今天以及明天的分布式系统软件提供了一个概念模型”（选自摘要）。

20.4 Philip A. Bernstein, James B. Rothnie, Jr., and David W. Shipman (eds.): *Tutorial: Distributed Data Base Management*. IEEE Computer Society, 5855 Naples Plaza, Suite 301, Long Beach, Calif. 90803 (1978).

一本来源众多的论文集，按照以下几个题目进行组织：

1) 关系数据库管理概述
2) 分布式数据库管理概述
3) 分布式查询方法
4) 分布并发控制方法
5) 分布式数据库可靠性方法

20.5 Philip A. Bernstein *et al*.: “Query Processing in a System for Distributed Databases (SDD-1),” *ACM TODS 6*, No. 4 (December 1981).

参见文献[20.34]的注释。

20.6 Philip A. Bernstein, David W. Shipman, and James B. Rothnie, Jr: “Concurrency Control in a System for Distributed Databases (SDD-1),” *ACM TODS 5*, No. 1 (March 1980).

参见文献[20.34]的注释。

20.7 Charles J. Bontempo and C. M. Saracco: “Data Access Middleware: Seeking out the Middle Ground,” *InfoDB 9*, No. 4 (August 1995).

一本以IBM的DataJoiner为重点的有用教程（虽然也提到了其他的产品）。

20.8 Yuri Breitbart, Hector Garcia-Molina, and Avi Silberschatz: “Overview of Multi-Database Transaction Management,” *The VLDB Journal 1*, No. 2 (October 1992).

20.9 M. W. Bright, A. R. Hurson, and S. Pakzad: “Automated Resolution of Semantic Heterogeneity in Multi-Databases,” *ACM TODS 19*, No. 2 (June 1994).

20.10 Stefano Ceri and Giuseppe Pelagatti: *Distributed Databases: Principles and Systems*. New York, N.Y.: McGraw-Hill (1984).

20.11 William W. Cohen: “Integration of heterogeneous Databases without Common Domains Using Queries Based on Textual Similarity,” Proc. 1998 ACM SIGMOD Int. Conf. on Management of Data, Seattle, Wash. (June 1998).

描述了一个针对通常所说的“垃圾邮件问题”的解决方法——即当两个不同的文本串，比如说是“AT&T Bell Labs”和“AT&T Research”，所指的是同一个对象的时候，把它们识别出来（当然这是一种特定的语义不匹配）。这个方法主要是推导出这些文本串的相似性，这些文本串是“用通常在统计信息检索中采用的向量空间模型来度量的”。按照论文的说法，这种解决方法要比“朴质推论方法（naive inference methods）”快得多，而且惊人地准确。

20.12 D. Daniels *et al*.: “An Introduction to Distributed Query Compilation in R*,” in H. -J. Schneider(ed.), *Distributed Data Bases*: Proc. 2nd Int. Symposium on Distributed Data Bases (September 1982). New York, N.Y.: Notrh-Holland (1982).

参见文献[20.39]的注释。

20.13 C. J. Date: “Distributed Databases,” Chapter 7 of *An Introduction to Database Systems: Volume II*. Reading, Mass.: Addison-Wesley (1983).

本章的一部分是基于这本早期的出版物。

20.14 C. J. Date: “What Is a Distributed Database System?”, *in Relational Database Writings 1985-1989*, Reading, Mass.: Addison-Wesley (1990).

这篇论文介绍了分布式系统的“十二个目标”（20.3节基本上是直接按照这篇论文安排的）。像在本章中提到的，本地自治的目标不是可以100%达到的，确实存在某些情况需要对这一目标进行某种程度上的折衷。为了便于参照，我们在这里归纳一下这些情况：

- 一个分片关系的个别分片无法被正常地直接存取，甚至从它们所存储的场地上也不行。
- 一个复制关系（或分片）的单独的副本无法被正常地直接存取，甚至从它们所存储的场地上也不行。
- 设*P*是某个被复制关系（或分片）*R*的主副本，*P*存储在场地*X*上。则每个要存取*R*的场地都要依赖于场地*X*，即使是在该场地上还存有*R*的另一个副本。
- 如果仅仅考虑存储它的本地场地的情况，对一个参与多场地完整性约束的关系不能进行修改存取，而必须在定义约束的分布式数据库中去考虑它的修改存取。
- 在两阶段提交进程中做为一个参与者的场地必须遵循相应的协调者场地的决定（即提交或者回滚）。

20.15 C. J. Date: “Distributed Database: A Closer Look,” in C. J. Date and Hugh Darwen, *Relational Database Writings 1989-1991*, Reading, Mass.: Addison-Wesley (1992).

这是文献[20.14]的续篇，更为深入地讨论了十二个目标中的大部分（虽然还是以教程的形式）。

20.16 C. J. Date: “Why Is It So Difficult to Provide a Relational Interface to IMS?”, in *Relational Database: Selected Writings*. Reading, Mass.: Addison-Wesley (1986).

20.17 R. Epstein, M. Stonebraker, and E. Wong: “Distributed Query Processing in a Relational Data Base System,” Proc. 1978 ACM SIGMOD Int. Conf. on Management of Data, Austin, Tx. (May/June 1978).

参见文献[20.36]的注释。

20.18 Rob Goldring: “A Discussion of Relational Database Replication Technology,” *InfoDB 8*, No. 1 (Spring 1994).

对于异步复制的一个很好的概述。

20.19 John Grant, Witold Litwin, Nick Roussopoulos, and Timos Sellis: “Query Languages for Relational Multi-Databases,” *The VLDB Journal 2*, No. 2 (April 1993).

给出了关系代数和关系演算在多数据库系统下的扩展。其中讨论优化的问题，并且表明每一个多关系的代数表达式都有一个等价的多关系的演算表达式（“这个定理的逆命题是一个很有趣的研究问题”）。

20.20 J. N. Gray: “A Discussion of Distributed Systems,” Proc. Congresso AICA 79, Bari, Italy (October 1979). Also available as IBM Research Report RJ2699 (September 1979).

一个概略的但却很不错的综述和教程。

20.21 Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha: "The Dangers of Replication and a Solution," Proc. 1996 ACM SIGMOD Int. Conf. on Management of Data, Montreal, Canada (June 1996).

"在工作负载扩充的情况下，在任何地方、任何时候、以任何方式修改事务级复制都会有不稳定的表现……这里提出了一个新的算法，允许（处在断开状态的）移动应用给出一个试探性的修改事务，这个修改可以在以后应用到主副本上"（选自摘要，稍有修改）。

20.22 Ramesh Gupta, Jayant Haritsa, and Krithi Ramamritham: "Revisiting Commit Processing in Distributed Database Systems," Proc. 1997 ACM SIGMOD Int. Conf. on Management of Data, Tucson, Ariz. (May 1997).

本文提出了一个叫做 OPT的新的分布式提交协议，这个协议（ a）易于实现；（ b）可以与传统协议并存；（ c）"在各种工作负载与系统配置的情况下提供了最佳的事务吞吐性能"。

20.23 Richard D. Hackathorn: "Interoperability: DRDA or RDA?", InfoDB 6, No. 2 (Fall 1991).

20.24 Michael Hammer and David Shipman: "Reliability Mechanisms for SDD-1: A System for Distributed Databases," *ACM TODS 5*, No. 4 (December 1980).

参见文献 [20.34]的注释。

20.25 IBM Corporation: *Distributed Relational Database Architecture Reference*. IBM Form No. SC26-4651.

IBM的DRDA对分布式数据库定义了如下四个层次的功能：

1) 远程请求
2) 远程工作单元
3) 分布式工作单元
4) 分布式请求

由于这些术语已经成为产业界事实上的标准（至少是在产业界的某些部分），我们在这里对它们做个简要的解释。注意："请求"和"工作单元"是 IBM的术语，分别对应于 SQL语句和事务。

1) 远程请求是指一个在场地 *X*上的应用可以把一个单独的 SQL语句发送到一个远程场地 *Y*上去执行。这个请求完全在场地 *Y*上执行和提交（或者回滚）。在场地 *X*上的那个应用会接着发送另一个请求到场地 *Y*上（也可能是到另一个场地 *Z*上），而不管第一个请求是成功还是不成功。
2) 远程工作单元（缩写为 RUW）是指一个在场地 *X*上的应用可以在一个给定的"工作单元"（即事务）中把所有的数据库请求发送到一个远程场地 *Y*上去执行。数据库对于该事务的处理也因此就完全在远程场地 *Y*上进行，但是要由本地场地*X*来决定该事务是提交还是回滚。注意： RUW实际上就是在单服务器情况下的客户 /服务器处理过程。
3) 分布式工作单元（缩写为 DUW）是指一个在场地 *X*上的应用可以在一个给定的"工作单元"（即事务）中把某些或者所有的数据库请求发送到一个或者多个远

程场地 *Y*, *Z*, ……上去执行。因此一般而言，数据库对这个事务的处理是分散到许多场地上的，每个独立的请求还是在某个单独的场地上执行完成的，但是不同的请求可以在不同的场地上执行。不过场地 *X*仍然是协调场地，即由这个场地决定事务是提交还是回滚。注意： DUW实际上就是在多服务器情况下的客户/服务器处理过程。

4) 分布式请求是在这四个层次中唯一实现了通常所说的真正的分布式数据库支持的。分布式请求是在分布式工作单元的基础上，加上允许单独的数据库请求（SQL语句）跨越多个场地 —— 比如，一个来自于场地 *X*的请求可以要求执行对分别来自于场地 *Y*和*Z*上的表的连接或合并。要注意只有在这个层次上，系统才可以说是提供了真正的位置独立性。而在前面的三种情况下，用户都必须对数据的物理位置有所了解。

20.26 International Organization for Standardization (ISO): *Information Processing Systems, Open Systems Interconnection, Remote Data Access Part 1: Generic Model, Service, and Protocol (Draft International Standard)*. Document ISO DIS 9579-1 (March 1990).

20.27 International Organization for Standardization (ISO): *Information Processing Systems, Open Systems Interconnection, Remote Data Access Part 2: SQL Specialization (Draft International Standard*). Document ISO DIS 9579-2 (February 1990).

20.28 B. G. Lindsay *et al*.: "Notes on Distributed Databases," IBM Research Report RJ2571 (July 1979).

（由R*开发小组最初的部分成员所完成的）这篇论文分为五章：

1) 复制的数据
2) 授权与视图
3) 对分布事务管理的介绍
4) 恢复功能
5) 事务的启动、转移和终止

第1章讨论了更新传播的问题。第 2章说的几乎完全都是在一个非分布式系统（像系统R那种类型的系统）中授权的问题，只是在结尾的时候才作了一些其他的说明。第3章考虑的是事务的启动和终止、并发控制以及恢复控制，都很简要。第 4章用来探讨（同样是）在非分布式的情况下的恢复。第 5章较为具体地讨论了分布事务管理，特别地，还给出了一个非常认真的关于两阶段提交的表述。

20.29 C. Mohan and B. G. Lindsay: " Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions," Proc. 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (1983).

参见文献 [20.39]的注释。

20.30 Scott Newman and Jim Gray: "Which Way to Remote SQL?", *DBP&D 4*, No. 12 (December 1991).

20.31 M. Tamer Özsu and Patrick Valduriez: *Principles of Distributed Database Systems* (2nd edition). Englewood Cliffs, N.J.: Prentice-Hall (1999).

20.32 Martin Rennhackkamp: "Mobile Database Replication," *DBMS 10*, No. 11 (October 1997).

价格低廉、非常便于携带的计算机和无线网络之间的结合，使得一种新的分布式数据库系统的出现成为可能，它不仅能带来特别的收益，（当然）也会有特殊的问题。尤其是这种系统中的数据说起来是可以复制到成千个“场地”上去——但是这些场地都是移动的、并且是经常掉线的，它们的操作特点更是与较为传统的场地非常不同（比如在通信开销中还要考虑电池的使用率和连接的时间）等等。对于这类系统的研究是最近开始的（相关文献有[20.1]和[20.21]），而这篇短文强调了其中的一些主要概念和问题。

20.33 James B. Rothnie, Jr., and Nathan Goodman: “A Survey of Research and Development in Distributed Database Management,” Proc. 3rd Int. Conf. on Very Large Data Bases, Tokyo, Japan (October 1977). 在文献[20.4]中也做了发表。

一个非常有用的早期综述，分以下几个方面进行了讨论：

1) 修改事务的同步
2) 分布查询处理
3) 部件故障处理
4) 字典管理
5) 数据库设计

在这些的最后提到了物理设计问题——即我们在20.8节中所说的分配问题。

20.34 J. B. Rothnie, Jr., *et al.*: “Introduction to a Systyem for Distributed Databases (SDD-1),” *ACM TODS 5*, No. 1 (March 1980).

文献[20.5~20.6]、[20.24]、[20.34]和[20.40]都是关于早期的分布式原型SDD-1的。SDD-1运行在一组通过Arpanet互连的DEC PDP-10上，它提供了完全的位置独立性、分片独立性和复制独立性。下面我们选择这个系统的某些方面做一些说明。

查询处理：SDD-1的查询优化器（见文献[20.5]和[20.40]）大范围地利用了第6章所描述的半连接操作。在分布式查询处理中使用半连接操作的好处是它们具有减少网络中数据传输量的作用。比如，假设供应商关系变量S存储在场地A上，而发货关系变量SP存储在场地B上，而查询是“对供应商关系和发货关系进行连接”。可以不用（比如说）把整个S传送到B上，可以像下面这样做：

- 计算在场地B上对SP的S#的投影（TEMP1）。
- 将TEMP1发送到场地A。
- 在场地A上计算TEMP1和S关于S#的半连接，得到TEMP2。
- 将TEMP2发送到场地B。
- 在场地B上计算TEMP2和SP关于S#的半连接，结果就是先前查询的答案。

这个过程会明显减少了通过网络传输的数据总量，当且仅当

```
size (TEMP1) +size (TEMP2) <size (S)
```

这里一个关系的“大小（size）”是指这个关系中元组的数目与一个单独元组的长度（比如用位来计算）的乘积。显然优化器要能够估计中间结果，比如说TEMP1和TEMP2的大小。

更新传播：SDD-1的更新传播算法是“立即传播”（这里没有主副本的概念）。

并发控制：并发控制是基于所谓的时戳技术，而不是封锁技术。其目的是为了避

免与封锁相关的消息开销，但是实际上代价似乎并没有真正地并发！有关细节已经超出了本书的讨论范围（虽然文献[15.3]的注释确实非常简要地描述了一下基本的想法）。更多的内容可以参见文献[20.6]或[20.13]。

恢复控制：恢复是基于一个四阶段提交协议，目的是为了使在协调者场地上的处理能够比传统的两阶段提交协议更有弹性，但是，遗憾的是它还是使得处理在相当大的程度上变得更为复杂。（同样）细节的讨论超出了本书的范围。

目录表：目录表是当作普通的用户数据来管理的——它可以被任意地分片，而分片也可以被任意地复制和重分布，就像其他数据一样。这种方法的优点是显而易见的。当然缺点就是，由于系统没有任何关于一个给定部分的目录表的位置的先验信息，它必须维护一个更高级别的目录表——字典定位器——以提供这样的确切信息！这个字典定位器是全复制的（即在每个场地上都存有一个副本）。

20.35 P. G. Selinger and M. E. Adiba: "Access Path Selection in Distributed Data Base Management Systems," in S. M. Deen and P. Hammersley (eds.), Proc, Int. Conf. on Data Bases, Aberdeen, Scotland (July 1980). London, England: Heyden and Sons Ltd. (1980).

参见文献[20.39]的注释。

20.36 M. R. Stonebraker and E. J. Neuhold: "A Distributed Data Base Version of Ingres," Proc. 2nd Berkeley Conf. on Distributed Data Management and Computer Networks, Lawrence Berkeley Laboratory (May 1977).

文献[20.17]和[20.36~20.37]都是关于分布式Ingres原型的。分布式Ingres由许多University Ingres构成，运行在许多互连的DEC PDP-11上。它（和SDD-1及R*一样）支持位置独立性；它也支持具有分片独立性的数据分片（不是通过投影，而是通过选择），以及具有复制独立性的对这些分片的数据复制。与SDD-1和R*不同，分布式Ingres并不假设通信网络的速度很慢；相反，它的设计对于"缓慢的"（远程的）网络和本地的（即相当快的）网络都可以处理（优化器明白这两种情况的差异）。查询优化算法基本上是本书第17章中所说的Ingres分解策略（Ingres decomposition strategy）的扩展，在文献[20.17]中有关于它的细节叙述。

分布式Ingres提供两种更新传播算法：一个是"性能优先"算法，它只是修改主副本然后就将控制交还给事务（而将修改的传播交由若干个子进程并行地执行）；另一个是"可靠"算法，它立即修改所有的副本（见文献[20.37]）。并发控制在两种情况下都是基于封锁的，而恢复是基于改进的两阶段提交。

对于目录表而言，分布式Ingres采用了把对数据库某些部分的目录表进行全复制和对其他部分使用纯粹的本地目录表记录相结合的方法。进行全复制的部分主要包括一个所有用户可见的关系变量的逻辑说明和一个关于关系变量如何进行分片的说明，其他部分包括对本地物理存储结构的说明、本地数据库的统计信息（供优化器使用）以及安全性与完整性约束。

20.37 M. R. Stonebraker: "Concurrency Control and Consistency of Multiple Copies in Distributed Ingres," *IEEE Transactions on Software Engineering 5*, No. 3 (May 1979).

参见文献[20.36]的注释。

20.38 Wen-Syan Li and Chris Clifton: "Semantic Integration in Heterogenous Databases Using

Neural Network," Proc. 20th Int. Conf. on Very Large Data Bases, Santiago, Chile (September 1994).

20.39 R. Williams *et al*.: "R*: An Overview of the Architecture," in P. Scheuerman (ed.), *Improving Database Usability and Responsiveness*. New York, N. Y.: Academic Press (1982). 同时也是 IBM Research Report RJ3325 (December 1981).

文献[20.12]、[20.29]、[20.35]和[20.39]都是关于R*的，它是最初的系统R原型的分布版本。R*提供了位置独立性，但是没有分片和复制，因此也就没有分片独立性和复制独立性。同样的原因，更新传播的问题也就不存在了。并发控制是基于封锁的（要注意任何被封锁的对象都只有一份，主副本问题也不存在）。恢复是基于改进的两阶段提交。

20.40 Eugene Wong: "Retrieving Dispersed Data form SDD-1: A System for Distributed Databases," 在文献[20.4]中。

参见文献[20.34]的注释。

20.41 C. T. Yu and C. C. Chang: "Distributed Query Processing," *ACM Comp. Surv. 16*, No. 4 (December 1984).

一个对于分布式系统中查询优化技术的教程式概论，包含了大量的书目摘要。