

# 目录

介绍	1.1
常见问题	1.2
Part 1	1.3
第1章	1.3.1
第2章	1.3.2
第3章	1.3.3
Part 3 SimpleDB内部实现	1.4
第12章——磁盘及文件管理	1.4.1
第13章——内存管理	1.4.2
第14章——事务管理	1.4.3
第15章——记录管理	1.4.4
第16章——元数据管理	1.4.5
第17章——查询处理	1.4.6
第18章——SQL语句解析	1.4.7
第19章——SQL Planning	1.4.8
第20章——数据库服务	1.4.9

## 1. Overview

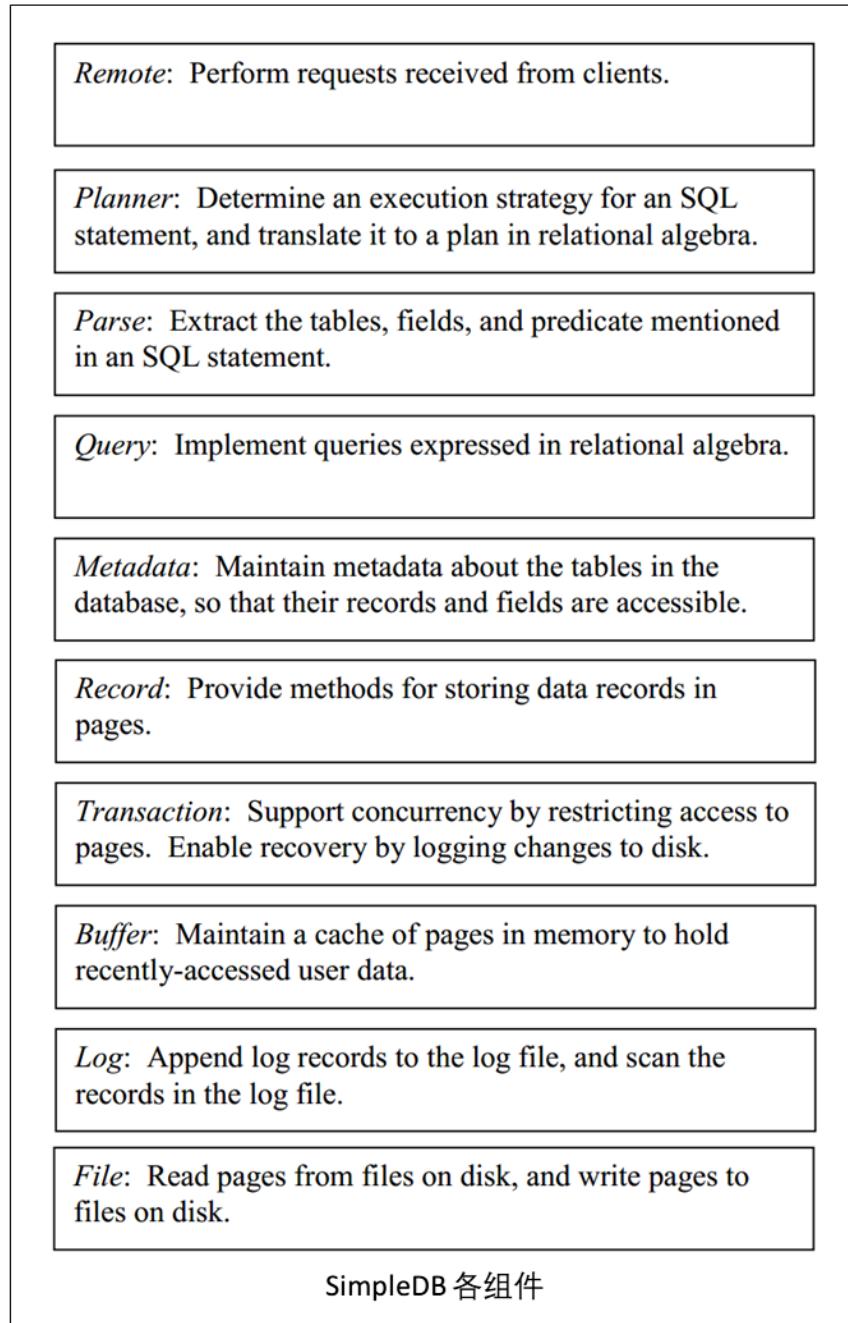
This is a simple relational database(named SimpleDB) implementation in Java. And this project is a playground-level database implementation described in the book "Database Design And Implementation" written by Edward Sciore, Boston College.

Just for educational use, I will follow the author to reimplement the SimpleDB. I have commented some code in Chinese for convenient reading and translated mainly chapters of part 3 in Chinese. You are also highly recommended to take a look about the original textbook which is really easy to understand.

这是一个简单的关系数据库（名为simpleDB）实现，Java语言实现，这是《Database Design And Implementation》作者Edward Sciore在书中提供的一个游乐场级别的数据库实现。为了方便学习使用，我会跟随作者的脚步去再次实现这个数据库。为方便大家阅读代码，我已经将部分关键代码添加了中文注释，并且尝试翻译了原书第3部分中的绝大部分章节，但仍然建议配合原书一起使用，因为相较于中文，英文没有那么地晦涩并且歧义更少，理解起来更简单。

## 2. 架构

SimpleDB的整体架构如下图所示，下层组件为上层组件提供服务：



### 3. 特点

#### 3.1 磁盘和文件管理

- 将文件块作为磁盘访问的最基本单元，缩短磁盘访问时间。
- 文件块中当前只支持int和string类型的读写。

#### 3.2 内存管理

- 通过维护缓冲池来固定最常使用的用户数据块，目前支持Naive缓冲页替换算法。

### 3.3 事务管理

- 支持日志来恢复数据库，采用的是undo-only恢复算法。
- 运用xlock和slock来控制多事务对块的并发访问。
- 事务对锁一直持有，直到事务commit或rollback。

### 3.4 记录管理

- 当前支持定长字段，定长记录。
- 一个记录文件中存储的是同类(homogeneous)的记录。
- 当前支持非跨块的记录(non spanned records)。
- 给客户端提供记录文件粒度的记录增、删、查、改方法，隐藏了底层的块、页细节。

### 3.5 元数据管理

- SimpleDB实现了4类元数据：
  1. 表的元数据，描述的是某张表的信息，例如该表每条记录的长度，每个字段的偏移量，类型。
  2. 视图的元数据，描述的是每个视图的属性，例如视图的名称和定义。
  3. 索引的元数据，描述的是每个索引的信息，包含索引名、被索引的表名、被索引的字段名。
  4. 数据统计的元数据，描述的是每张表的占用块数，已经各字段的值分布情况。
- 表的元数据存放在系统的catalog表tblcat和fldcat中，一张来存放表粒度的信息（例如记录长度），另一种表来存放表中各字段信息（例如字段名、字段长度、字段的类型）。表名和字段名最长字符限定为20。
- 视图的元数据存放在系统的catalog表viewcat中，视图的定义最长字符数我们限定成了150。
- 索引的元数据存放在系统的catalog表idxcat中，包含索引名、被索引的表名、被索引的字段名。
- 数据统计信息没有用表来存，而是在系统每次启动的时候重新计算出，对于小型的数据库，这一统计计算时间不是很长，因此不会拖长整个系统的启动时间。

### 3.6 查询处理

- 基于流水线的方式实现Scan。目前支持TableScan、SelectScan、ProductScan和ProjectScan,分别对应SQL语句中的表、谓词筛选、

笛卡尔积和输出Select列名。

- 对于一个SQL，可能存在多个等效的查询树，planner会比较这些查询树对应的Scan的执行代价，选择代价最低的那个。
- 目前支持形如"A = c"和"A = B"形式的谓词，前者代表例如 where StuId=1 的形式，而后者代表例如 where Stu.StuId = Exam.Id 的形式。

## 3.7 SQL解析

- SimpleDB中只支持SQL中的子集，具体有：
  1. 简单的查询，单表多表均支持
  2. 增删改记录
  3. 创建表、视图、索引
  4. 只支持where谓词，不支持group by等
- 采用递归下降解析法(recursive descent)解析SQL语句。

## 3.8 SQL Planning

- 目前实现了最简单的planning算法（包括query planning算法和 update planning算法），没有作SQL语义验证和plan代价分析。
- 设计好了Planner的接口，增强代码的plug-and-play capability。

## 3.9 C/S 通信

- 通过Java中的RMI机制来实现客户端和服务端之间的通信。
- 服务端的每个远程实现类对象都在一个独自的线程中执行，等待客户端通过存根对象发送消息，SimpleDB启动代码会创建一个 RemoteDriver 类型的远程实现对象，并把这个对象的存根对象注册到RMI的注册表中。
- 当客户端想要连接到数据库系统时，会先通过RMI注册表得到存根对象，并按照JDBC提供接口进行数据访问，目前实现了 SimpleConnection , SimpleStatement 等，分别对应JDBC中的 Connection , Statement 。

## Q&A

### 1. 本书的面向的读者？

本书的面向的读者是那些想要学习或研究数据库原理和底层实现的人，你最好对数据库的基本概念有个大概的了解，例如基本的SQL语句、数据库设计的几种范数、数据库的ACID原则等等，细节不清楚没关系，这本书就是给你展示细节的。本书假设你对操作系统、编译原理中的一些基本概念略有耳闻，例如磁盘访问、并发、多线程等等。

## 2. 关于错误

本书是译者在阅读英文版原书时翻译的手稿或者说笔记，难免会有很多错误，必然存在我的理解错误、口误、输入错误、翻译不准确等众多问题，如果你有任何问题，请先仔细思考或查阅相关资料，如果的确有问题，非常欢迎通过email联系交流。

## 3. 译本进度

目前只翻译了第12—20章，由于本人时间有限，因此会不定时地更新本译本，欢迎任何人共同加入到翻译此书的工作中来，如果你有意向，请email联系，也欢迎直接pull request。

## 4. 如何获得本书？

1. 你可以直接打开[SimpleDB中文版gitbook](#)查看。
2. 你也可以在本地安装nodejs和gitbook，并且编译成PDF文件格式查看。

本仓库所有译本默认可以转载，但请注明出处，也请勿用于商业用途。

译者：Liu Zhian

email：[liuzhian@whut.edu.cn](mailto:liuzhian@whut.edu.cn)

最后更新时间：2020/08/05

---

## 打赏

你的支持将会是我最大的动力！打赏记得备注哟~ ¡Salud :beers: :beers: :beers:



## 常见问题

### 1. 本书的面向的读者?

本书的面向的读者是那些想要学习或研究数据库原理和底层实现的人，你最好对数据库的基本概念有个大概的了解，例如基本的SQL语句、数据库设计的几种范数、数据库的ACID原则等等，细节不清楚没关系，这本书就是展示细节的。本书假设你对操作系统、编译原理中的一些基本概念略有耳闻，例如磁盘访问、并发、多线程等等。

### 2. 关于错误

本书是译者在阅读英文版原书时翻译的手稿或者说笔记，难免会有很多错误，必然存在我的理解错误、口误、输入错误、翻译不准确众多问题，如果你有任何问题，请先仔细思考或查阅相关资料，如果的确有问题，请通过email联系。

### 3. 译本进度

目前只翻译了第12—16章，由于本人时间有限，因此会不定时地更新本译本，欢迎任何人共同加入到翻译此书的工作中来，如果你有意向，请email联系，也欢迎直接pull request。

译者：刘知安

email：929910266@qq.com

最后更新时间：2020/02/26

## Part1

# 第一章

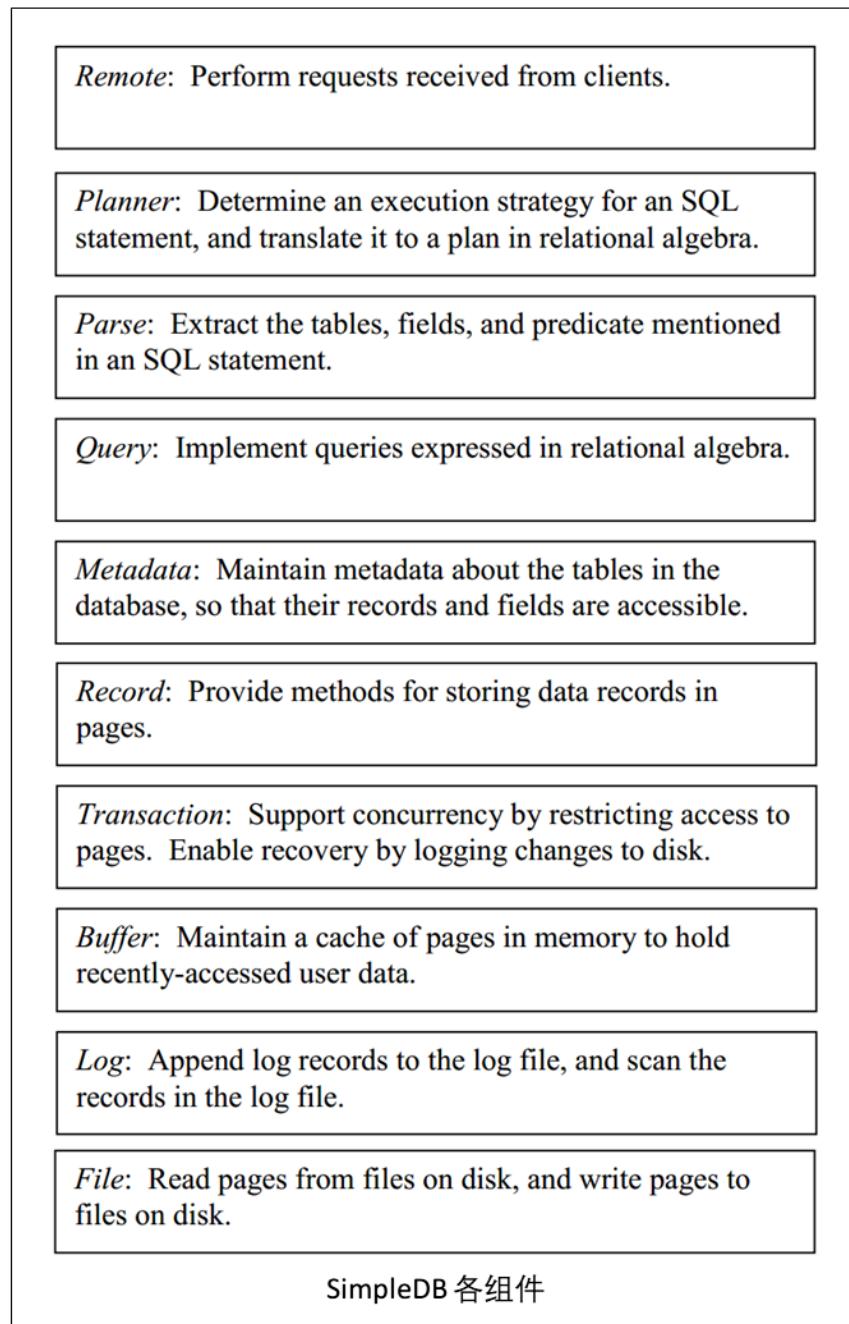
## 第二章

## 第三章

## Part 3 - 数据库服务器内部实现

现在我们已经学习了如何使用一个数据库系统，我们现在将移步至如何构建一个数据库系统的问题。理解现代数据库系统是如何工作的一个很好的方式就是学习一个真正的数据库系统的内部实现。这个 SimpleDB 数据库系统就是为了这个目的而构建的，我们将在本书的第3部分完成，它总共包含了大约3500行Java源代码，由12个packages中的85个类组织而成；并且在本书随后的第4部分——效率优化部分，代码量将缩小至大约一半，这虽然看起来有很多代码，但随后我们将会知道 SimpleDB 是一个单纯的系统骨架，其实现的采用了尽可能简单的算法，也只保留了一些最基本、也是必要的功能。商业化的数据库系统拥有和这一样的架构，但是使用了更复杂的算法、拥有鲁棒性更高的错误检查机制以及大量的警告（原文为**belts and whistles**），这样一来，系统明显会变得庞大。例如 Derby(一个数据库系统)，代码量大概是 SimpleDB 的100倍以上（20万行以上），Oracle就更大了。

在本书中，我们将数据库服务器分解成拥有10层组件的系统，每个组件只知道它下层的组件细节，并为它上层的组件提供服务，下图展示了该系统中每个组件的功能。



说明该系统中各个组件功能的一个较好方法就是按照一条SQL语句的执行去解析。例如，假设现在有一个JDBC的客户端通过 `executeUpdate` 方法，提交了一条删除SQL语句到数据库服务器，以下的动作将会发生：

- remote组件实现了 `executeUpdate` 方法，该方法为当前statement 创建一个事务，并将string类型的SQL语句传递给planner
- planner组件发送string类型的SQL语句传递给parser组件，parser组件会解析SQL语句，并返回一个 `DeleteData` 对象，该对象包含当前statement涉及的数据库表明和删除谓词（即删除成功or失败）。

planner组件会使用parser组件的数据去决定为当前statement创建一个计划 (plan) , 该计划表示了一个为了找到待删除记录而创建的关系代数查询 (relational algebra query) 。最后planner组件将创建的计划送给查询处理器 (query processor) 。

- 查询处理器 (query processor) 为这个plan创建一个扫描 (scan) , 这个扫描包含了执行该计划时需要的运行时信息。为了创建这个扫描, 查询处理器包含了从源数据管理器 (metadata manager) 获得的相关表模式 (schema) 。查询处理器随后创建一个对应该模式的记录管理器 (record manager)
- 查询处理器随后在刚创建的扫描上进行迭代, 迭代过程中, 它会检测数据库表中的每一条记录并且判断是否满足我们定义的删除谓词。每次需要获得一个记录 (record) 时, 它都显式地向记录管理器发出请求。如果一条记录需要被删除, 查询处理器会告诉记录管理器相应的执行。
- 记录管理器知道记录是怎样保存在文件中的, 当查询处理器请求一个记录的值时, 它会知道文件中的哪一块 (block) 包含了所需的记录, 并且计算相应的偏移量从而获得记录中的某个项具体的值 (value) 。它随后会向事务管理器发出通知, 去检索 (retrieve) 指定块指定位置的值。
- 事务管理器检查它的表锁 (注: 事务管理器为每个表维护一个锁, 所有表的锁信息也由一个lock table维护) , 从而确保其它事务不会使用当前块。如果还没有没有上锁, 则事务管理器则为当前块上锁, 并调用缓冲区管理器 (buffer manager) 的相关方法, 从而获得需要的值。
- 缓冲管理器的目的是将很多块的内容缓存在内存中, 如果被请求的块没有被缓存, 则缓冲管理器从缓存区中选取一个页, 并调用文件管理器 (file manager) 的相关方法, 将相关的块读入当前页, 亦可将当前页的内容写到磁盘中。 (注: page是内存中的概念, 而块是文件中的概念)
- 文件管理器将某个块的内如读/写入内存中的相应页中。

如上的这种架构的相应变种就是当前大多数商业关系数据库系统的变种。该架构严重受到磁盘I/O特性的影响, 因此在本书的这部分中, 我们从最底层——磁盘——开始, 一步步向上实现该数据库软件。

面向对象的开发者通常将一个对象对另一个类的某个方法调用称为 客户端 (client) , 客户端 这个术语其实和本书第二部分所提及的客户端-服务端很类似, 无论是哪种说法, 客户端对象都会通过另一个类对象的某个方法执行一次请求。要说区别, 那就是在 C-S 例子中, 请求是通过网络实现的 (注: 即客户端和服务端通过网络进行交互) , 而一般意义上的 client就没有这种限定。在本书的剩余部分, 我们提到的客户端都代表这面向对象层面一般术语的意思, 而 JDBC 客户端 则更倾向于 C-S 层面的意思。

## 第12章 磁盘和文件管理以及 simple.file 包的实现

在本章中，我们研究数据库系统如何将其数据存储在磁盘和闪存驱动器等物理设备上。我们研究了这些设备的属性，并考虑了可以提高其速度和可靠性的技术（例如RAID）。我们还将研究操作系统提供的用于与这些设备进行交互的两种接口（块级接口和文件级接口），并讨论哪种接口最适合数据库系统。最后，我们详细研究 SimpleDB 的文件管理器，研究其API及其实现。

### 12.1 持久化数据存储

数据库的内容必须保持持久性，以便在数据库系统或计算机出现故障时不会丢失数据。本节研究了两种特别有用的硬件技术：磁盘驱动器(disk drives) 和 闪存驱动器(flash drives)。

磁盘驱动器和闪存驱动器是最常用的持久化存储设备。

闪存驱动器没有磁盘驱动器运用的那么广泛。但是，随着技术的成熟，它们的重要性将会提高。我们从磁盘驱动器开始，然后再考虑闪存驱动器。

#### 12.1.1 磁盘驱动器

磁盘驱动器包含一个或多个可旋转的 盘片 (platter)。盘片具有很多同心 磁道 (tracks)，每个磁道由一系列字节组成。通过带有读/写头的可移动磁臂从盘片中读取字节（或者将字节写入盘片）。当磁臂移动至所需的磁道上时，磁头可以读取（或写入）字节它旋转时位于在其下方的字节。图12-1描绘了一个单盘片驱动器的俯视图。当然，该图不是按比例绘制的，因为典型的盘片具有成千上万的磁道。

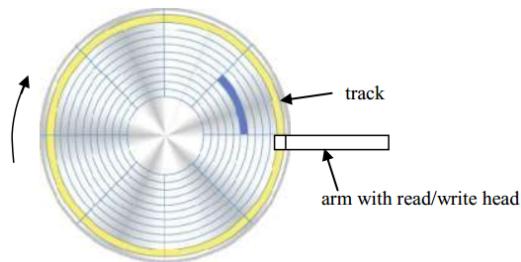


Figure 12-1: The top view of a one-platter disk drive

现代的磁盘驱动器通常都会有很多盘片，为了空间效率考虑，通常都会量成对的盘片背对背地连接在一起，形成看起来像双面盘片的样子；但从概念上讲，盘片的任一面应该是一个单独的盘片，每个盘片都有自己的读/写头，这些磁头理应是独立移动的；但实际上，它们都连接到一个 致动器 (actuator)，该致动器将它们同时移动到每个盘片上的同一磁道

上，而且，所有磁道中一次只能有一个读/写头处于活动状态，因为到计算机的数据路径 (datapath) 只有一个。图12-2描绘了多磁盘驱动器的侧视图。

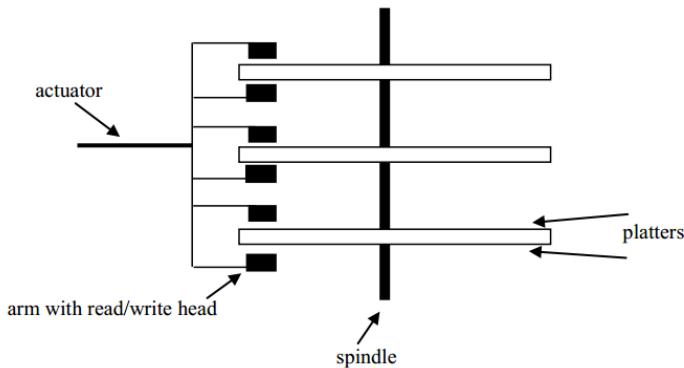


Figure 12-2: The side view of a multi-platter disk drive

我们现在考虑如何表征一个磁盘驱动器的性能。

一个磁盘驱动器的大体上的性能可以通过下面几个量来表示：它的容量(capacity)，转速(rotation speed)，传输速率(transfer rate)和平均寻道时间(average seek time)。

驱动器的容量(capacity)是可以存储的字节数。该值取决于盘片的数量，每个盘片的磁道数以及每个磁道的字节数。鉴于当前盘片或多或少的尺寸标准，很多磁盘制造商主要通过增加磁盘的密度来增加磁盘的容量，即将更多的磁道集成在一个盘片上，并且让每个磁道存储更多的字节。现在容量超过40GB的盘片很常见。

转速(rotation speed)是盘片旋转的速率，通常表示为每分钟旋转的圈数。典型速度范围为5400rpm至15000rpm不等。

传输速率(transfer rate)指的是字节通过磁盘头的读写，从而被传输到内存的速率。例如，整个磁道的字节可以在盘片旋转一周的时间内传输完成，因此，传输速率由盘片的旋转速度和每个磁道上的字节数共同决定。常见的速率为100MB/秒。

寻道时间(seek time)是制动器将磁盘的磁头从当前位置移动到所需轨道所花费的时间。这个值取决于需要移动多少个磁道，最低可以低至0（如果目标磁道与起始磁道相同），最高可达15-20毫秒（如果目标磁道和起始磁道刚好在一个盘片的最外侧和最内侧）。平均寻道时间通常是对制动器移动速度的合理估计。现代磁盘的平均寻道时间约为5毫秒。

考虑下面的例子：

一个拥有4个盘片，转速为10000rpm，且平均寻道时间为5ms的磁盘。每个盘片有10000个磁道，每个磁道上包含50万个字节，我们可以进行以下计算：磁盘的容量为： $5 \times 10^5$  字节/磁道  $\times 10^4$  磁道/盘片  $\times 4$  盘片/磁盘 =  $20 \times 10^9$  字节 (20 GB) 磁盘的传输速率为： $5 \times 10^5$  字节/转  $\times 10^4$  转/60s =  $8.3 \times 10^7$  字节/秒 (83兆字节/秒)

### 12.1.2 访问磁盘驱动器

一次 磁盘访问 (disk access) 指的是一次从磁盘到内存的读请求或者内存到磁盘的写请求。这些字节必须在某个盘片的某些磁道上的连续区域。

磁盘驱动器按照以下3个步骤执行一次磁盘访问：

- 首先将磁头移到目标磁道，如图前文所说，这个时间交寻道时间。
- 再等待盘片旋转，直到磁头下对应的字节为我们期望的首字节，这个时间被叫做 旋转延时 (rotational delay)
- 盘片继续旋转，磁头不断地将字节读取 (或者写入到磁道上)，直到最后一个字节。整个这个过程的用时被称为 传输用时 (transfer time)。

执行一次磁盘访问的总用时为寻道时间，旋转延时和传输用时的总和。

注意，上述的这些时间都是受到磁盘机械旋转(mechanical movement)的限制的，而机械旋转又显然比电旋转(electrical movement)慢很多，这也是为什么磁盘驱动器会比RAM慢很多的原因。寻道时间和旋转延时特别烦人，它们只不过就是每次磁盘操作前必须等待的时间罢了。

计算一次磁盘访问的准确寻道时间和旋转延时是不实际的，因为它需要知道磁盘的当前状态，相反，我们可以通过使用它们的平均值来估计这些时间。我们已经知道平均寻道时间。平均旋转延时也很容易计算，旋转延时可以低至0 (如果第一个字节恰好位于当前磁头下方)，以及高达一次完整的旋转用时 (即第一个字节刚刚已经错过了磁头)，平均而言，我们将不得不等待 $1/2$ 次旋转，直到盘片旋转到在我们想要的磁道位置，因此，平均旋转延时是旋转时间的一半。

传输用时也可以很容易通过传输速率计算得到，举例来说，如果一个磁盘的传输速率为  $b_1 B/s$ ，我们需要传输  $b_2 B$ ，则传输时间为  $b_2/b_1$  秒

继续我们之前的例子，我们可以估计一次磁盘访问的耗时如下：

考虑一个10000rpm的磁盘，平均寻道时间为5秒，传输速率为83Mb/s 平均旋转延时为： $60 \text{ s/min} \times 1 \text{ min}/10000 \text{ 转} \times 0.5 \text{ 转} = 3 \text{ ms}$  传输1字节用时为： $1 \text{ B} \times 1\text{s}/83\text{M B} = 1.2 \times 10^{-5} \text{ ms}$  传输1000字节用时为： $1000 \times 1.2 \times 10^{-5} \text{ ms} = 1.2 \times 10^{-2} \text{ ms}$  1字节的一次磁盘访问估计用时为：5ms（寻道时间）+ 3ms（平均旋转延时）+  $1.2 \times 10^{-5} \text{ ms}$ （传输用时）= 8.000012 ms 1000字节的一次磁盘访问估计用时为：5ms（寻道时间）+ 3ms（平均旋转延时）+  $1.2 \times 10^{-2} \text{ ms}$ （传输用时）= 8.012 ms

可以注意到，传输1000字节和传输1字节的整体估计用时几乎是一样的，也就是说，只访问很少字节对磁盘访问读写时间来说没有影响，实际上，如果你真的只想读一个字节，底层也不会有这样的支持。现代的磁盘都会将一个磁道分成很多的扇区 (sectors)；一次磁盘的读写必须是以整个扇区为最小操作单位的，具体每个扇区的大小由磁盘制造商来决定，在磁盘出厂时就已经被确定了，一个常见的扇区大小为512字节。

### 12.1.3 改进磁盘访问时间

因为磁盘驱动器很慢，有一些技术已经被提出来改进磁盘访问时间，我们将考虑三种技术：磁盘缓存 (disk cache)，扇区 (cylinder) 和 磁盘分条 (disk striping)

#### 磁盘缓存 (*Disk Cache*)

磁盘缓存是与磁盘驱动器绑定在一起的一块内存区域，该缓存区通常足够大从而存储许多磁道数据。每当磁盘驱动器从磁盘读取一个扇区时，它都会将该扇区的内容保存在其高速缓存中。如果缓存已满，则新扇区将替换旧扇区。当请求扇区时，磁盘驱动器将检查缓存，如果该扇区恰好在高速缓存中，则可以将其立即返回计算机，而无需实际的磁盘访问。

假设一个应用程序在相对较短的时间内多次请求相同的扇区。然后，第一个请求会将扇区带入缓存，随后的请求将从缓存中检索该扇区，从而节省磁盘访问时间。然而，这个功能对数据库系统不是特别重要，因为数据库系统已经以相同的方式进行了缓存（我们将在第13章中看到）。如果多次请求一个扇区，则数据库服务器将在自己的缓存中找到该扇区，甚至不必费心去磁盘。

以下是因为磁盘缓存很有价值的真正原因。假设磁盘从不一次性读取单个扇区，磁盘驱动器不只是读取请求的扇区，而是将包含该扇区的整个磁道读取到高速缓存中，且认为在不久的将来，该磁道上的其他扇区也会被请求。关键是，读取整个磁道并不会比读取单个扇区浪费很多时间（和我们上述计算时读取一个字节和读取1000个字节的例子类似）。特别地，这种情况将没有旋转延迟，因为磁盘可以从读/写头下方的任何扇区开始读取磁道，并在整个旋转过程中继续读取。以下是两种访问模式的访问用时对比：

读一个扇区的时间：寻道时间 + 1/2 旋转用时 + 磁头读取字节时旋转用时  
 读一个磁道的时间：寻道时间 + 磁头读取字节时旋转用时

也就是说，读单个扇区比一整个磁道所有扇区的时间大约少了二分之一全的旋转用时，所以假如数据库系统刚好像读磁道上的一个扇区，它会将整个磁道的所有扇区都读入到高速缓存中，以便节省时间。

磁盘缓存的真正作用其实就是允许磁盘 预提取 (pre-fetch) 磁道上的一些扇区。

### **柱面 (Cylinder)**

数据系统可以通过预提取扇区附近的相关信息从而减少磁盘访问时间，例如，存储文件的理想方法是将其内容放在盘片的同一磁道上。如果磁盘执行基于磁道的缓存，则此策略显然是最好的，因为整个文件将在一次磁盘访问时间内读取到。但是该策略即使没有缓存也很好，因为它消除了寻道时间——每次读取另一个扇区时，磁头都已经位于正确的磁道上了。

(注：一个文件中分散在磁盘不同磁道上的内容被称为碎片。许多操作系统（例如Windows）提供了一种碎片整理实用程序，该实用程序通过重新定位每个文件以使其扇区尽可能连续从而缩短磁盘访问时间。)

假设一个文件的内容分布在多个磁道，一个好的办法是将其内容存储在盘片的附近磁道上，以使得磁道之间的寻道时间尽可能短。但是，一个更好的办法是将其内容存储在其他盘片的同一磁道上。由于每个盘片的读/写头都一起移动，因此可以访问具有相同磁道号的所有磁道而无需任何额外的寻道时间。

具有相同磁道号的一组磁道称为柱面，因为如果你查看这些磁道，它们似乎描述了一个柱面的表面。实际上，可以将柱面视为一条非常大的轨道，因为其中的所有扇区都可以在零附加寻道时间内访问到。

### **磁盘分条 (Disk Striping)**

减少磁盘访问时间的另一办法是使用多个磁盘驱动器。

假设你想存储40GB的数据库，两个20GB的驱动器总是会比一个40GB的驱动器快。两个20GB的驱动器总是会比一个40GB的驱动器快的原因是因为它们拥有两个独立的制动器，因此可以同时处理两个不同的扇区读写请求。实际上，如果两个20GB的磁盘都总是繁忙，那么该磁盘的性能将是单个40GB磁盘的两倍。这种性能提升可以很好地扩展：通常来说，N个磁盘的速度约为单个磁盘的N倍。

(当然，几个较小的驱动器也比单个较大的驱动器昂贵，因此效率的提高是有代价的。)

多个小型磁盘只有保持繁忙状态才会更有效率。例如，假设一个磁盘包含频繁使用的文件，而另一块磁盘包含很少使用的存档文件。然后，第一个磁盘将完成所有工作，而其他磁盘大部分时间都处于空闲状态。因为几乎没有 同时性(simultaneity)，所以此设置的效率与单个磁盘几乎相同。

因此，问题在于如何在多个磁盘之间均衡负载。数据库管理员可以尝试分析文件使用情况，以便最佳地在每个磁盘上分发文件，但是这种方法不太现实：很难做到，也很难保证，并且随着时间的推移必须不断进行重新评估和修订。幸运的是，有一种更好的方法，称为 **磁盘分条(disk striping)**。磁盘条带化策略使用控制器将较小的磁盘在操作系统的层面予以隐藏，从而给人一种单个大磁盘的错觉。控制器将虚拟磁盘上的扇区请求映射到实际磁盘上的扇区请求。映射如下。假设有 $N$ 个小磁盘，每个小磁盘有 $k$ 个扇区。虚拟磁盘将具有 $N * k$ 个扇区；这些扇区以交替方式分配给实际磁盘的扇区。磁盘0将包含虚拟扇区0,  $N$ ,  $2N$ 等。磁盘1将包含虚拟扇区1,  $N + 1$ ,  $2N + 1$ 等。依此类推。术语 **磁盘条带化** 来自以下想象：如果你想象每个小磁盘都涂上了一种颜色，则虚拟磁盘看起来像有条纹一样，其各个扇区会被涂上交替的颜色（注：大多数控制器允许用户将条带定义为任意大小，而不仅仅是扇区。例如，如果磁盘驱动器执行的是基于磁道的磁盘缓存，则磁道可以构成良好的条带。最佳条带大小取决于许多因素，并且通常由反复试验确定），如图12-3所示：

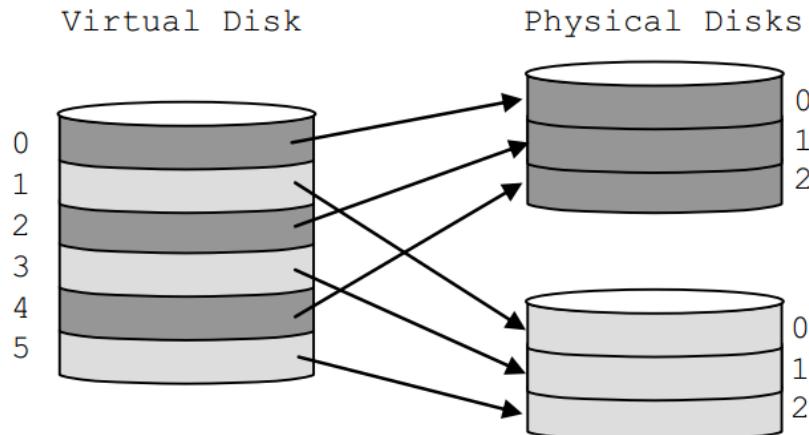


Figure 12-3: Disk striping

磁盘分条是有效的，因为它可以将整个数据库均匀地分布在各小磁盘之间。如果一个随机扇区请求来了，那么该请求将以相同的概率被分发到其中一个小磁盘，而且，如果有几个连续扇区的请求到达，它们将被发送到不同的磁盘。因而，确保了各磁盘尽可能均匀地工作。

#### 12.1.4 通过镜像提高磁盘可靠性

数据库用户希望其数据在磁盘上保持安全，并且不会丢失或损坏。不幸的是，磁盘驱动器并不完全可靠。盘片上的磁性材料可能退化，导致扇区变得不可读。或者，灰尘或震动可能会导致读/写磁头刮擦磁盘而损坏受影响的扇区（这被称为 **磁头碰撞**）。

防止磁盘故障的最明显的方法是保留磁盘内容的副本。例如，我们可以每晚对磁盘进行一次备份。当磁盘出现故障时，我们只需购买一个新磁盘并将备份复制到该磁盘上。该策略的问题在于，会丢失那些发生在上一次备份之后，且先于故障发生之前的所有的数据更改（译者注：举个例子来说，例如我们每晚11点进行磁盘备份，某一次，在凌晨1点发生了磁盘故障，我们可以买一个新的磁盘把上一次备份的数据进行拷贝，因为上一次拷贝是昨天晚上11点，如果在昨晚11点到今日凌晨1点间，发生了任何数据修改，我们是无论如何都无法恢复数据的）。解决此问题的唯一方法是在发生更改时将每个更改立马复制到备份磁盘。换句话说，我们需要保留两个相同版本的磁盘。这些两个相同内容的磁盘称为彼此的镜像。

与磁盘分条一样，需要一个控制器来管理两个镜像磁盘。当数据库系统请求读取磁盘时，控制器可以选择访问任一磁盘的指定扇区。当请求写磁盘时，控制器对两个磁盘执行相同的写操作。从理论上讲，这两个磁盘写操作可以并行执行，不需要额外的时间。但是，实际上，顺序写入镜像对防止系统崩溃很重要。问题是，如果系统在磁盘写入的过程中崩溃了，则该扇区的内容将丢失。因此，如果两个镜像并行写入，则扇区的两个副本都可能丢失，而如果依次以顺序的方式写入镜像，则至少有一个镜像不会被损坏。

假设镜像对中的一个磁盘发生故障。数据库管理员可以通过执行以下过程来恢复系统：

- 停止该系统
- 更换新的磁盘替代失效的磁盘
- 将好的磁盘（也就是镜像磁盘）的内容复制到新磁盘
- 重启系统

不幸的是，此过程并非万无一失。如果在将好的磁盘内容复制到新磁盘的过程中发生故障，则数据仍可能丢失。两个磁盘彼此都在几个小时内发生故障的可能性很小（对于今天的磁盘来说，这个概率大约是6万分之一），但是如果数据库很重要，那么这种小风险仍然可能是无法接受的。我们可以通过使用三个而不是两个镜像磁盘来降低风险。在这种情况下，只有在几个小时内所有三个磁盘都发生故障时，数据才会丢失；尽管非零，但这种可能性非常低，以至于可以视为地忽略。

磁盘镜像可以与磁盘分条共存，一个通常的做法是对分条的磁盘进行镜像，例如，一个存储40GB数据的系统可以用四个20GB的磁盘来实现：其中两个20G的磁盘作为分条磁盘，另外两个20GB的磁盘作为条带化磁盘的镜像，这样的配置既快速又可靠，如图12-4所示：

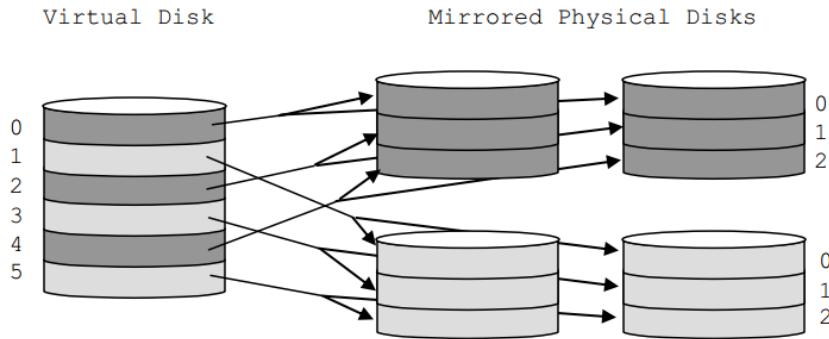


Figure 12-4: Disk striping with mirrors

### 12.1.5 通过存储奇偶校验码提高磁盘可靠性

镜像的缺点是它需要两倍的磁盘来存储相同数量的数据。当使用磁盘分条时，这种负担特别明显——如果我们要使用15个20GB驱动器存储300GB数据，那么我们将需要购买另外15个驱动器作为它们的镜像。大型数据库安装通过分条许多小磁盘来创建巨大的虚拟磁盘的情况并不罕见，并且购买相同数量的磁盘作为镜像磁盘的前景并不大。于是如何从故障磁盘中恢复而无需使用太多镜像磁盘将是一个很好的选择。

实际上，的确存在一种很好的办法，它可以使用一个磁盘来备份其他所有的磁盘，这种策略通过存储 奇偶校验码(parity) 来恢复磁盘。一串二进制码S的奇偶校验码定义如下：

- 如果S串中包含奇数个1，则奇偶校验码为1。
- 如果S串中包含偶数个1，则奇偶校验码为0。

也就是说，加上奇偶校验码后，S串和奇偶校验码总共包含的1的数量为偶数个。

奇偶校验具有以下有趣且重要的属性：只要我们也知道奇偶校验，就可以从其他位的值确定任意位的值。例如，假设 $S = 1, 0, 1$ 。 $S$ 的奇偶校验码则为0，因为它有偶数个1。假设现在我们丢失了第一位的值，因为奇偶校验为0，所以集 $? , 0, 1$ 必须具有偶数1，因此我们可以推断出丢失的位必须为1，可以对其他每个比特（包括奇偶校验比特）进行类似的推论。

奇偶校验的这种使用扩展到了磁盘。假设我们有 $N + 1$ 个相同大小的磁盘。我们选择其中一个磁盘作为奇偶校验磁盘，然后让其他 $N$ 个磁盘保存带化数据。通过查找所有其他磁盘的相应位的奇偶校验来计算奇偶校验磁盘的每一位。如果任何磁盘发生故障（包括奇偶校验磁盘），则可以通过逐点查看其他磁盘的内容来重建该磁盘内容。请参见图12-5：

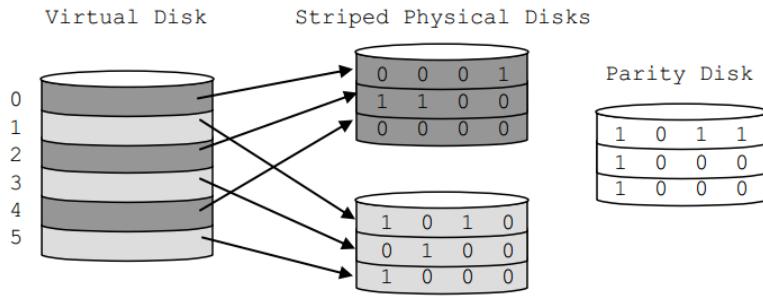


Figure 12-5: Disk striping with parity

这些磁盘被控制器管理，读写请求基本和磁盘分条时相同——控制器决定哪个磁盘上包含了被请求的扇区，然后再执行相应的读/写操作。区别在于在写操作是必须更新对应奇偶校验磁盘上的相应扇区，控制器可以通过查看请求扇区被修改后的二进制位来计算出更新后的奇偶校验码，因此控制器实现一次扇区写操作需要4次磁盘请求：首先它必须读取扇区数据和对应的奇偶校验扇区（目的是计算新的奇偶校验位），然后它也必须将新的内容写到扇区和奇偶校验扇区。

一个磁盘居然能够可靠地备份任意数量的其他磁盘，奇偶校验信息的使用有些神奇。但是，这种魔术带来了两个缺点：

1. 扇区写入操作更加耗时，因为它需要从两个磁盘进行读取和写入。经验表明，使用奇偶校验会使磁盘分条效率降低约20%。
2. 数据库更容易遭受不可恢复的多磁盘故障。我们考虑一下磁盘发生故障时会发生什么情况——重建故障磁盘需要所有其他磁盘，并且其中任何一个磁盘的故障都是灾难性的。如果数据库由很多小磁盘组成（例如100个），那么第二次故障的可能性就非常大。将这种情况与镜像进行对比，我们只需要从发生故障的磁盘中恢复数据时，仅要求其镜像磁盘不发生故障即可，这时同时损坏的可能性是很小的。

## 12.1.6 RAID

我们已经知道了3种使用多磁盘的方法：

- 磁盘分条从而加快磁盘访问时间
- 镜像确保磁盘损坏时恢复数据
- 奇偶校验确保磁盘损坏时恢复数据

上述所有的策略都使用了一个控制器从OS的角度来隐藏实际存在的多磁盘，并且给用户提供了一种幻觉——有一个超大容量的虚拟磁盘。控制器将每次虚拟的磁盘读写操作映射到底层实际的磁盘上，控制器可以以软件或硬件的形式实现，但是大多数的控制器还是以硬件的方式实现的。

这些策略是称为RAID的大量策略的一部分， RAID (Redundant Array of Inexpensive Disks) 代表廉价磁盘冗余阵列，有七个RAID级别：

- RAID-0 就是磁盘分条，没有任何防止磁盘故障的措施。如果条化磁盘之一发生故障，则整个数据库可能会被破坏。
- RAID-1 是镜像的磁盘分条。
- RAID-2 使用的是位条化而不是扇区条化，并使用基于纠错码而不是奇偶校验的冗余机制。事实证明，此策略难以实施且性能不佳，因此不再使用。
- RAID-3 和 RAID-4 使用磁盘分条和奇偶校验。它们的区别在于 RAID-3 使用字节条化，而 RAID-4 使用扇区条化。通常，扇区条带化往往更有效，因为它对应于磁盘访问的单位。
- RAID-5 与 RAID-4 相似，不同之处在于 RAID 奇偶校验信息不是将所有奇偶校验信息存储在单独的磁盘上，而是分布在数据磁盘之间。也就是说，如果有 N 个数据磁盘，则每个磁盘的第 N 个扇区都保存奇偶校验信息。这种策略比 RAID-4 更有效，因为不再有单个奇偶校验磁盘成为瓶颈。参见练习 12.5。
- RAID-6 与 RAID-5 类似，除了它保留两种奇偶校验信息。因此，此策略能够处理两个并发的磁盘故障，但是需要一个以上的磁盘来保存其他奇偶校验信息。

用的最多的 RAID 级别就是 RAID-1 和 RAID-5，这两种选择之间的真正区别就是镜像和奇偶校验位之间的区别。在数据库配置中，镜像往往是更可靠的选择，首先是因为它的速度和鲁棒性，其次是因为额外的磁盘驱动器的成本现在已经很低了。

### 12.1.7 闪存

磁盘驱动器在当前的数据库系统中很常见，但是它们有一个无法克服的缺点——它们的操作完全取决于旋转盘片和制动器的机械活动。与电子存储器相比，这个缺点使磁盘驱动器本来就很慢，并且还容易因失效、振动和其他冲击而损坏。

闪存是一种较新的技术，具有替换磁盘驱动器的潜力。它使用类似于 RAM 的半导体技术，但不需要不间断提供电源。由于其活动完全是电子的，因此有可能访问数据比磁盘驱动器快得多，并且没有活动部件受到损坏。

闪存驱动器当前的寻道时间约为 50us，大约为比磁盘驱动器快 100 倍。当前闪存驱动器的传输速率取决于它连接到的接口总线。通过快速内部总线连接的闪存驱动器和磁盘驱动器的速度相当；但是，外部 USB 闪存驱动器比磁盘驱动器的传输慢很多。

不过闪存是会耗尽的。每个字节只能重写一定的次数。达到最大值后，闪存驱动器将发生故障。当前，此最大值大约为数百万，但对于大多数数据库应用程序来说，这是相当高的。高端驱动器采用 耗损均衡 (wear-leveling) 的技术，该技术可将频繁写入的字节自动移动到写入较少的位置。此技术允许驱动器运行，直到驱动器上的所有字节达到其重写限制，而不仅仅是第一个字节。

闪存驱动器为操作系统提供了基于扇区访问的接口，这意味着闪存驱动器对于操作系统而言就像磁盘驱动器一样。可以将RAID技术用于闪存驱动器，尽管条化的重要性不高，因为闪存驱动器的寻道时间非常短。

采用闪存驱动器的主要障碍是其价格。目前闪存的价格大约是同类磁盘驱动器价格的100倍。尽管闪存和磁盘技术的价格将继续下降，但最终闪存驱动器将便宜到足以被视为主流。到那时，磁盘驱动器可能会被转移到档案存储和超大型数据库的存储中。

闪存还可以通过用作文件持久化的前端来增强磁盘驱动器。如果数据库完全存在于闪存中，则磁盘驱动器将永远不会被使用。但是，随着数据库变大，不常使用的扇区将迁移到磁盘。

就数据库系统而言，闪存驱动器与磁盘驱动器具有相同的属性：它是持久的，缓慢的，并且可以在扇区中进行访问。（它恰好比磁盘驱动器慢。）因此，我们将遵循当前的术语，并在本书的其余部分中将持久性存储器统称为“磁盘”。

## 12.2 磁盘的块级接口

不同的磁盘拥有多种硬件属性，例如，它们不是必须拥有相同的扇区大小，而且它们的扇区可以以不同的方式组织，操作系统负责隐藏这些不同的细节，并且为用户代码的文件读写提供统一的简单的接口。

**块 (block)** 的概念就是操作系统接口的核心。

一个块和一个扇区类似，不过块的大小是由操作系统来决定的，不同磁盘都拥有同样大小的块。

OS维护了块和扇区之间的映射，OS也会给每个磁盘上的每个扇区赋值一个 **块号 (block number)**；给定一个块号，操作系统知道实际的扇区地址。

不能直接从磁盘访问块的内容。取而代之的是，必须首先将包含该块的扇区读入 **内存页 (memory page)** 中并从那里进行访问。

页是内存中一个和块同样大小的区域。

如果一个客户端想要修改一个块中的内容，他必须将这个块先读到页中，然后修改页中的相应字节，然后再将该页写回到磁盘的相应块中。

一个OS必然会提供几种获取到磁盘块的方法，下面有4个示例方法：

- **readblock(n,p)** : 将磁盘第n块的字节读到内存的第p页中
- **writeblock(n,p)** : 将内存的第p页的字节写到磁盘第n块中
- **allocate(k,n)** : 从第n块开始，寻找连续的k个没被使用过的块，并标记为使用过，返回这连续k块的块首的块号。这个块号应该尽量接近n

- `deallocate(k, n)` : 从第n块开始, 将它后续的k块标记为未使用过 (包括第n块)

OS会知道磁盘上的哪些块可以被分配, 哪些不可以。这里大概有2中可以采用的策略: 磁盘映射 (disk map) 和 空闲列表 (free list)

- 一个disk map是一串连续的二进制位, 每一位对应磁盘的一个块。1代表块是空闲的, 0代表块是被分配了的。这个disk map也是被保存在磁盘中的, 通常被保存在一个磁盘的初始几个块中。OS可以简单地通过将disk map相应块的二进制位从0置为1从而取消分配该块; 亦可以从第n块开始将连续k位二进制位置为0从而分配连续的k个磁盘块, 当然前提是这k块是连续的并且之前的相应二进制都是1.
- 一个free list是一连串的 块 (chunk) , (注意, 这里说的块和磁盘的块不一样), chunk是连续的未被分配的blocks, 也就是说, chunk是比block更大一点的单位, 每个chunk的第一个block存储了两个值: 当前chunk的长度, 以及下一个chunk首块的block number, 你可以将chunk理解为一个单链表的结点。磁盘的第一个block包含了一个指向第一个chunk的指针 (注: 一般这里会用数组链表, 如果你忘记了这是什么, 可以Google一下)。当OS被调用, 请求分配k个连续的块时, 它会搜索free list从而获取一个足够大的chunk, 然后它 can 将这个chunk从free list中分配掉, 或者从中切出长度为k的块分配掉; 当OS被请求取消分配一些块的时候, OS简单地将这些块插入到free list中。

上面的文字描述可能不是很清晰, 不用担心, 图12-6示例了这两种不同的策略, 此时磁盘的第0,1,3,4,8,9块都已经被分配出去了。图12-6 (a) 展示了在磁盘的di0块存储的disk map; 0代表一个已经被分配的块。图12-6 (b) 展示了对应的free list状况, Block 0包含一个值2, 代表第一个chunk是从第2个block 开始的; Block 2 包含两个值1和5, 1代表当前chunk包含1个block, 5代表下一个chunk起始block的块号, 类似地, 第5个block中的内容表示的意思是当前chunk包含3个block, 且下一个chunk从第10个block开始; 而block 10 上的-1代表这是最后一个块了, 包含剩下的所有blocks了。

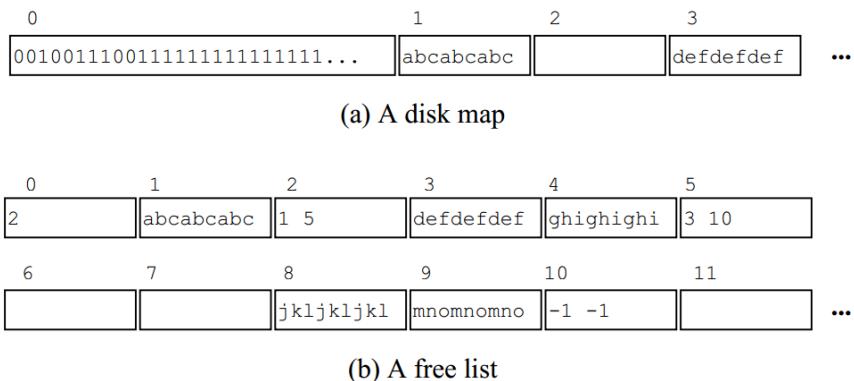


Figure 12-6: Two ways to keep track of free blocks

free list技术需要最小化的额外空间;我们需要的就是在block 0记录一个整数,用来指向列表中的第一个块。另一方面, disk map技术需要空间来维持map映射, 图12-6 (a) 假定这个map可以装进一个块中。通常来说 然而, 需要好几个块来保持这个map信息, 详情请参见练习12.6。disk map的一个好处就是可以让OS清楚地知道哪里有一个 洞 (hole) ,举例来说, 如果操作系需要一次支持分配多个块, 则通常选择磁盘映射 (disk map) 技术作为策略。

## 12.3 磁盘的文件级接口

OS提供了另一种, 高层次的磁盘接口, 被叫做 文件系统 (file system) 。一个客户端将一个文件看做是一些列字节, 在这个级别上, 用户不知道块这概念, 相反, 客户端可以读/写文件任何位置开始的任意长度的字节流。

Java中的 RandomAccessFile 提供了一个典型的文件系统API (译者注: 说到这里, 我不得不说一下, 在很多书中, 都会将RAM翻译成随机存取存储器, 我觉得这种翻译是不恰当的, 也是反中文常识的, 我觉得应该翻译成任意存取存储器, 对应地, 在这里我会翻译成任意访问文件。任意和随机显然是两个不同的概念, 不是吗? 任意是说, 我想读哪里就读哪里, 是受客户端控制的; 而随机首次会被理解成从任意位置开始读取, 是不受控制的概念)。每一个 RandomAccessFile 对象都拥有一个 文件指针 (file pointer) , 它指明了下一次文件读写操作时开始的字节位置, 这个文件指针可以被显式地赋值通过方法 seek() , 一次 readInt() 或者 writeInt() 的调用将页会移动文件指针, 移动过的数据就是我们读/写操作的数据。

以下是一个示例代码片段, 该片段增加了存储在文件 junk 中第7992至7995处字节的整数 (注: Java中一个Int整数占4个字节) 。

对 readInt() 的调用将读取字节7992处的整数, 并将文件指针移至字节7996。随后的seek调用会将文件指针设置回字节7992, 以便可以覆盖该位置处的整数。

```
RandomAccessFile f = new RandomAccessFile("junk","rws");
f.seek(7992);
int n = readInt();
f.seek(7992);
f.writeInt();
f.close();
```

注意, readInt() 和 writeInt() 的调用视磁盘被直接获取, 隐藏了磁盘块需要先被读到内存页中的细节。OS通常为这些页保留一些内存, 这些内存页称为 I/O缓冲区(I/O buffer) 。打开文件后, OS会从该池中分配一个页面供文件使用, 客户端对此不了解, 也不需要了解。

通过向用户隐藏其用到的I/O缓冲区，文件系统将掩盖它与磁盘之间的交互。例如，从上述代码很难明显的看出，方法 `writeInt()` 比 `readInt()` 需要更多的磁盘访问。调用 `readInt()` 要求操作系统将相应的块读入I/O缓冲区，然后在适当的位置返回该值。但是，对 `writeInt()` 的调用要求操作系统将块读入缓冲区，修改缓冲区中的某个位置，然后将缓冲区重新写回到磁盘。

一个文件可以被看做是一系列的块。例如，假如一个块的大小为4096字节（即4K），那么第7992个字节就在block 1中（注意，从0开始计数），像“第1块”这样的引用被称为 逻辑（logical）块引用，因为它们会告诉我们该块相对于文件的位置，而不是该块在磁盘上的位置。

给定一个特定的文件位置，`seek()` 方法会决定底层实际是哪一块包含了该位置。

`seek()` 方法执行两次转换：

- 它会将指定的字节位置转换为一个逻辑块引用
- 他会将逻辑块引用转换为一个物理块引用

第一个转化相对来说是很简单的——逻辑块号不过就是指定的字节数除以一个块的块大小，举例来说，假设一个块的大小为4K，那么第7992个字节就在第1个块，因为 $7992/4K=1$ （整数除法）。

第二个转化就比较难了，而且依赖于当前操作系统下的文件系统是怎么实现的。我们将会看到3种实现方式：`连续分配 (contiguous allocation)`，`基于扩展的分配(extent-based allocation)` 和 `索引分配(index allocation)`。

#### 连续分配 (*contiguous allocation*)

连续分配是最简单的了，它将文件安装连续块的方式存储起来。为了实现连续分配，OS只需要维护每个文件的长度和文件其实块的位置。OS会将这些信息存储在文件系统的目录中，图12-8展示了一个文件系统中包含两个文件的目录：一个长为48个块的名为 `junk` 的文件，它的起始块为 block 32，另一个长为16个块的名为 `temp` 的文件，它的起始块为block 80。

Name	1 <sup>st</sup> Block	Length
junk	32	48
temp	80	16

Figure 12-8: A file system directory for contiguous allocation

逻辑块引用到物理块引用的映射很容易——如果文件从block b开始，则文件的第N块位于磁盘的块 $b + N$ 中。这种简单的实现有两个问题：

1. 如果紧随一个文件其后的是另一个文件，则该文件无法扩展文件。图12-8中的 `junk` 文件就是这类问题的示例。因此，创建文件时必须使

用可能需要的最大块数，文件尚未满时，这会导致浪费空间。

2. 随着磁盘慢慢变满，它可能会有许多未被分配的小chunk，但又没有大的chunk，因此，即使磁盘有很多的空闲空间，也可能无法创建大文件。

第一个问题被称为 内部碎片问题 (internal fragmentation) ,第二个问题被称为 外部碎片问题 (external fragmentation) 。

内部水平是在文件里面浪费的空间，而外部碎片是指在文件外部浪费的空间，或者说文件之间浪费的空间。

### **基于扩展的分配(extent-based allocation)**

基于范围的分配策略是连续内存分配的变种，该策略可以减少内部碎片和外部碎片。在这里，OS将文件固定长度的扩展区序列存储在文件中，其中每个扩展区是一块连续的块。文件一次扩展一个范围。对于每个文件，OS会维护一个列表，列表中的每个元素都是每个扩展区的第一个逻辑块号。

举例来说，假如OS以8块的扩展区为单位存储文件，图12-9表示了一个包含2个文件的文件系统目录，这两个文件名为 `junk` 和 `temp` ,这两个文件和之前提到的文件大小相同，但是它们是以扩展的形式被存储的， `junk` 文件有6个扩展， `temp` 文件有2个扩展。

Name	Extents
<code>junk</code>	32, 480, 696, 72, 528, 336
<code>temp</code>	64, 8

Figure 12-9: A file system directory for extent-based allocation

为了找到磁盘上某文件的第N块，OS会首先搜索文件的扩展列表然后确定哪个扩展包含了这个块，然后再执行相应的计算，考虑下面这样一个例子：

假设OS的文件系统中有一个如图12-9所示的目录 为了找到文件 `junk` 的第21块： 包含第21块的扩展为  $21/8 = 2$  (整数除法) 该扩展区的首个块号为  $2*8 = 16$  所以第21块是第二个扩展区的第  $21-16 = 5$  块 而第二个扩展区的首块对应的实际物理块号为  $L[2] = 696$  所以第21块在磁盘上的实际位置为  $696+5=701$  块

现在考虑一下基于扩展的分配方式是怎么消除内部碎片的（译者注：我个人觉得并没有完全消除内部碎片，只是削弱了内部碎片的大小），是的，正是因为现在一个文件最多可以浪费不超过一个扩展的空间。外部空间一样可以类似地被消除，因为只要存在一个扩展区，那么它就可以被利用起来。

### **索引分配 (index allocation)**

索引分配采取了一个不同的策略——它甚至不会尝试去给文件开辟连续的 chunks，相反，磁盘中的每个block都被单独的分配（如果你想这么理解也可以，这就是扩展长度为1的基于扩展的分配），然后OS为了实现这种策略，会为每个文件维持一个 索引块 (index block)，该索引块中保存了这个文件是由那些块组成。也就是说，你可以把索引块中的内容看作是一个整形的数组，`ib[N]` 表示的是第N个逻辑块的实际物理块号。

图12-10 (a) 表示了一个包含2个文件的文件系统目录，这两个文件名为 `junk` 和 `temp`，`junk` 文件的索引块号为34，图12-10 (b) 展示了索引块中的前面部分整数，每个数字代表了该文件的每个块号。我们可以很清楚的看到，`junk` 文件的第一个逻辑块对应的物理块号为103.

Name	Index Block
junk	34
temp	439

(a) The directory table

Block 34:
32 103 16 17 98 ...

(b) The contents of index block 34

Figure 12-10: A file system directory for indexed allocation

这种策略的优势是每次只分配一个块，所以出现碎片问题的概率被降到了最低，但是这种方法的主要问题是文件会有一个最大的size，因为我们假定索引信息也是放在一个块内，而块本身也是有大小的，也就是说，索引块中存放的整数个数是有限的。UNIX文件系统通过支持多级索引块来解决这个问题，因此文件的最大size变的很大，详情请参考练习12.12和12.13。

在以上的3中实现策略中，OS需要将文件系统中的目录存在磁盘中，当需要将逻辑块引用转换为物理块引用时，`seek()` 方法会访问这些目录块。我们可以将这些磁盘访问视为文件系统的额外“开销”。操作系统试图尽量减少这种开销，但是又无法消除这种开销。因此，每一次 `seek()` 操作都可能导致一个或多个磁盘访问发生，这在某种程度上是不可预测的。

## 12.4 数据库系统和操作系统

OS提供了两个级别的磁盘访问： 块级支持 和 文件级支持 。我们应该选用哪一种级别来实现数据库呢？

选择使用块级支持的好处是，操作系统对整个磁盘块都有完全的控制权，举例来说，常用的块将被存储在磁盘磁道的中间部分，这样一来，寻道时间就会比较少，类似地，连续的块也可以被存储在附近的区域。使用块级支持另一个好处是，数据库系统没有OS关于文件大小的限制，这样也就是允许数据库表可以比文件系统的最大文件size还要大，或者甚至横跨多个磁盘。

然而，另一方面，使用块级存储又会有如下一些缺点：这种策略的实现很复杂，要求我们将磁盘看做是一个个raw的disk，而且需要自己来管理挂载和解挂之类的操作——也就是说，磁盘块也不再是文件系统的一部分，需要数据库管理员为了优化（fine-tune）系统，那么他必须对块访问有很深刻的理解。

另一种策略是使用OS提供的基于文件系统的数据库系统实现，例如，每个表都被存放在一个文件中，数据库系统可以使用文件级别的操作来访问记录，这种方式的实现就会简单很多，而且运行OS将实际的底层磁盘访问隐藏掉，但是这个场景让人不太能接受的原因也有两个：

- 数据库系统需要知道块的边界在哪里，因而它可以更高效地组织和检索数据。
- 数据库系统需要管理它自己的页，因为OS提供的I/O缓冲区可能对于数据库的查询来说不是很合适。

我们将会在后面的章节中遇到这些问题。

一个折中的办法就是数据库系统将所有的数据存储在一个或多个操作系统文件中，但是把这些文件视为raw disks，也就是说，数据库系统通过使用逻辑文件块来访问它的“磁盘”（其实是文件）。OS负责将每个逻辑文件块引用映射到真实的物理磁盘块中，通过 seek() 方法。因为 seek() 方法可能偶尔才会引发磁盘访问，数据库系统将不会完全控制底层的物理磁盘，然而，与数据库实际访问数据时带来的磁盘访问想必，seek() 方法带来的额外磁盘访问就不是那么的明细了。因此，数据库系统不仅能够使用OS的高级接口，同时也能对磁盘访问时间有一个明显的控制。

这种这种的策略被使用在大量的现代数据库系统中，例如 MS Access（该系统将所有信息都保存在单个的.mdb文件中），Oracle的数据库也是如此（不过是放在多个OS级别的文件中）。我们的SimplDB也将使用这种和Oracle类似的折中策略。

## 12.5 SimpleDB的文件管理器

正如我们先前看到的一样，OS为我们的数据库系统提供了服务，如低级别的磁盘处理和文件系统，在一个数据库系统中，与OS交互的部分被称为 文件管理器(file manager)。

文件管理器是数据库中负责与OS进行交互的组件。

在本节中，我们将展示SimpleDB数据库系统中的文件管理器，12.5.1小节将涵盖了文件管理器的API，我们将会看到文件管理器如何允许客户端访问任意文件的任意块，如何允许客户端将一个块读到内存页中，如何允许客户端将一个内存页中的信息写回文件块中。文件管理器也支持客户端在一个页的任意一个位置放置一个值，当然也支持从任意位置开始检索一个值。12.5.2小节将展示文件管理器如何用Java实现。

### 12.5.1 使用文件管理器

SimpleDB数据库将数据存放在几个文件中。对于每个表，会有一个文件；对于每个索引，也会有一个文件；对于日志文件和目录文件也是如此。SimpleDB的文件管理器提供了对于这些文件的块级访问，通过包 `simpledb.file` 中的方法。这个包内包含了3个类：`Block`，`Page` 和 `FileMgr`，他们的API如下所示：

```
// Blcok.java
public class Block {
    public Block(String fileName, int blkNum);
    public String filename();
    public int number();
}

// Page.java
public class Page {
    public Page();
    public int getInt(int offset);
    public void setInt(int offset,int val);

    public int getString(int offset);
    public void setString(int offset,String val);

    public void read(Block blk);
    public void write(Block blk);
    public Block append(String fileName);
}

// FileMgr.java
public class FileMgr {
    public FileMgr(String dirname);
    public boolean isNew();
    public int size(String fileName);
}
```

一个 `Block` 类对象标识的是一个指定的文件块，通过给定文件名和逻辑块号唯一标识。例如，以下语句

```
Block blk = new Block("student.tbl", 23);
```

它创建了一个新的块对象，引用的是文件 `student.tbl` 的第23个逻辑块  
 (译者注：这里是逻辑块号，实际的物理磁盘块是由OS来掌管的)。

一个 `Page` 类对象维持了一个磁盘块的内容，以及对应OS的I/O缓冲区。  
 方法 `setInt()` 和 `setString()` 在页的指定起始位置开始设置了一个  
 新的值，方法 `getInt()` 和 `getString()` 检索了指定位置开始的已经  
 存储好的值。当前虽然只有Int和String两个数类型被支持，其他的数据类  
 型同样也可以支持，详情请参见联系12.17。一个客户端可以在一个页的  
 任意开始位置存储一个值，但是也必须知道这个位置之前存储的是什么  
 值。从错误的其实位置读取一个值会引发不可预测的结果和错误。

方法 `read()`，`write()` 和 `append()` 负责实际的磁盘访问。`read()`方  
 法将指定块的内容读取到内存页中；`write()`方法则刚好做相反的事情；  
`append()`方法会在指定文件末尾分配一个新的块，并且将该块中的内容初  
 始化为当前页中的内容，并且返回新分配块的逻辑引用。

`FileMgr` 类则处理的是实际与OS的交互操作。它的构造函数接受一个  
 string作为参数，该字符串指明了这个数据库文件存放在哪个目录下。假  
 如该目录不存在，则会为创建一个新的数据库从而创建该目录。方  
 法 `isNew()` 就是判断这个情况的，这个方法被用来正确的初始化数据库  
 服务端。方法 `size()` 则返回的是指定文件的块数，它将允许客户端直  
 接移动到文件的末尾。

系统的每个实例都会有一个 `FileMgr` 对象，这会在系统启动时被创建。  
 类 `server.SimpleDB` 会创建这个 `FileMgr` 类的实例对  
 象，`server.SimpleDB` 类会有一个静态的方法 `fileMgr()` 返回这个创  
 建的对象。

下面的示例代码演示了如何使用这些代码，这个代码片段有3个小节。第一  
 小节在 `studentdb` 这个目录下初始化了SimplDB系统，并且获得了一  
 个文件管理器，并确定了 `junk` 文件的最后一个块，并创建了一  
 个 `Block` 类的引用指向它；第二小节将3992位置开始的int数加1，然后写  
 回到块中去；第三小节将字符串“hello”存储到了另一个页的第20个字节开  
 始的地方，然后将该页的内容追加到一个新的文件块中。随后，将那个块  
 又重新读到第3个页中，并将“hello”的值提取到变量s中，并且将新块的  
 块号和s打印出来。

```
SimpleDB.init("studentdb");
FileMgr fm=SimpleDB.fileMgr();
int fileSize=fm.size("junk");
Block blk =new Block("junk",fileSize-1);

Page p1=new Page();
p1.read(blk);
int n=p1.getInt(3992);
p1.setInt(3992,n+1);
p1.write(blk);

Page p2=new Page();
p2.setString(20,"hello");
blk=p2.append("junk");
Page p3=new Page();
p3.read(blk);
String s=p3.getString(20);
System.out.println("Block: "+blk.number()+" contains "+s);
```

## 12.5.2 实现文件管理器

我们现在考虑如何将文件管理器中的3个类实现一下。

### **Block** 类

Block 类 的代码如下所示。除了直接实现 filename() 和 number() 方法，我们也要实现 equals() , hashCode() 和 toString() 方法。

```
package simpledb.file;

public class Block {
    private String fileName;
    private int blkNum;

    public Block(String fileName, int blkNum) {
        this.fileName = fileName;
        this.blkNum = blkNum;
    }

    public String filename() {
        return fileName;
    }

    public int number() {
        return blkNum;
    }

    @Override
    public int hashCode() {
        return toString().hashCode();
    }

    @Override
    public boolean equals(Object obj) {
        Block blk = (Block) obj;
        return fileName.equals(blk.fileName) && blkNum == blk.blkNum;
    }

    @Override
    public String toString() {
        return "[File: " + fileName + ", block number: " +
    }
}
```

**Page** 类 Page类的实现如下：

```
public class Page {  
    public static final int BLOCK_SIZE = 400;  
    public static final int INT_SIZE = Integer.SIZE / Byte.  
  
    public static final int STR_SIZE(int n) {  
        float bytesPerChar = Charset.defaultCharset().newEncoder().  
            // 指示字符串长度的整数 + 各字符占的字节数  
            return INT_SIZE + n * ((int) bytesPerChar);  
    }  
  
    // 页中的内容  
    private ByteBuffer contents = ByteBuffer.allocateDirect(BLOCK_SIZE);  
    private FileMgr fileMgr = SimpleDB.fileMgr();  
  
    public Page() {}  
  
    public synchronized int getInt(int offset) {  
        contents.position(offset);  
        return contents.getInt();  
    }  
  
    public synchronized void setInt(int offset, int val) {  
        contents.position(offset);  
        contents.putInt(val);  
    }  
  
    public synchronized String getString(int offset) {  
        contents.position(offset);  
        int len = contents.getInt(); // 获取字符串的长度  
        byte[] byteVal = new byte[len];  
        contents.get(byteVal);  
        return new String(byteVal);  
    }  
  
    public synchronized void setString(int offset, String val) {  
        contents.position(offset);  
        byte[] byteVal = val.getBytes();  
        int len = byteVal.length; // 获取字符串的长度  
  
        contents.putInt(len);  
        contents.put(byteVal);  
    }  
  
    public synchronized void read(Block blk) {  
        fileMgr.read(blk, contents);  
    }  
}
```

```

public synchronized void write(Block blk) {
    fileMgr.write(blk, contents);
}

public synchronized Block append(String fileName) {
    return fileMgr.append(fileName, contents);
}
}

```

这个类定义了三个public的常量，`BLOCK_SIZE`，`INT_SIZE` 和 `STR_SIZE`，其中 `BLOCK_SIZE` 指定了一个块的字节数，在这里被设置成了400，这个值相对来说还是很小的，但我们还是设置成了它，是因为想要模拟更多block的情况。在商用的数据库系统中，这个值通常被设置为和OS块同样的大小，一个典型的值就是4K字节。

`INT_SIZE` 和 `STR_SIZE` 这两个常量指定了一个将要被存储的int整数和字符串的长度，int字节在Java中就是用4个字节来表示，然而，字符串就没有这样的标准了。在我们的SimpleDB中，一个字符串由两个部分共同来表示：第二个部分是字符串的字面量，而第一部分则是这个字符串字面量的长度，所以假如我们假设每个字符占用1个字节（正如英语中的字符一样），因而字符串 `joe` 将有一个占4字节的int整数和占3字节的字符串 `j`，`o`，`e` 共同组成。通常来说，`STR_SIZE(n)` 返回的值为  $4 + (n * k)$ ，其中 `k` 是在一个字符集中，表示一个字符需要的最大字节数。

（译者注：在我的windows机器上，JVM中字符编码格式为UTF-8，一个字符占3个字节，你可以

用 `Charset.defaultCharset().newEncoder().maxBytesPerChar()` 来查看你的机器环境。）

实际上，内存页的内容是由Java中的 `ByteBuffer` 对象来表示的，和一般的构造函数不太一样，`ByteBuffer` 类包含2个工厂方法 `allocate()` 和 `allocateDirect()`。在我们的 `Page` 类实现中，使用的是 `allocateDirect()` 方法，它会告诉编译器使用OS I/O缓冲区中的其中一个来维持字节数组。

一个 `ByteBuffer` 对象会跟踪缓冲区的当前位置指针，可以通过 `position()` 方法来改变它的位置。`Page` 类中的 `get/set` 方法简单地先将 `position` 指针移动到想要的位置，然后再调用 `ByteBuffer` 类中的相应方法来访问从 `position` 位置开始的一些列字节序列。`getString()` 方法首先读一个int类型整数，它代表的是这个字符串用字节序列表示时的长度；然后它继续读取后续的字节，并为之构造一个 `String` 类对象。`setString()` 方法则刚好执行的是差不多相反的操作。

所有 Page 类中的方法都是以 同步(synchronized) 的方式来实现的，这以为着只有1个线程可以在某个时间下执行。同步是当方法被多个可更新对象共享使用时，为了维持数据一致性而必须要实现的方法，例如上述代码中的 contents 变量。举例来说，假如 getInt() 方法没有实现同步，那么以下这种场景可能就会发生：假设有两个JDBC的客户端，每个可能会运行着他们自己的线程，并且都尝试读不同的整数到一个相同的内存页中。线程A下执行，它开始执行 getInt() 方法，但是在运行完该方法中的第一行后被打断了，也就是说，position已经被设置好了，但是还没开始真正读数据；这个时候线程B紧接着执行，并且也执行了 getInt() 方法，直到这个方法执行完毕，这个时候的position会被指向线程B想指的地方；现在线程A继续执行，可是position位置已经被B改变了，线程A却全然不知，因此线程A继续读出来的数据就会是错的，即从一个错误的起始位置不正确地读取了数据，这种错误是不可预测的。  
(简单点说，Page对象到时候是一个共享资源，涉及共享资源并发访问，就必须考虑同步问题。)

**FileMgr 类** FileMgr 类的实现代码如下所示，它的主要工作就是实现在 Page 类中定义的 read() , write() , append() 方法。FileMgr 类中的 read() 和 write() 方法会先将position移动到指定文件的正确位置（译者注：根据文件块的逻辑块号和块大小就可以确定），然后将指定内存页中的内容读/写到块中去。append() 方法则会先将position移动到文件的末尾去，然后将内存页中的内容追加到文件尾，返回的是一个新的块对象的引用，OS会为我们把文件块到底层磁盘块的细节完成，不需要我们多考虑。

```

public class FileMgr {
    private File dbDirectory;
    private boolean isNew;
    private Map<String, FileChannel> openFiles = new HashMap<String, FileChannel>();

    public FileMgr(String dbname) {
        // 默认路径为user的home路径
        String homedir = System.getProperty("user.home");
        dbDirectory = new File(homedir, dbname);
        isNew = !dbDirectory.exists();

        // 如果是新的数据库，则创建
        if (isNew && !dbDirectory.mkdir())
            throw new RuntimeException("Cannot create " + dbDirectory);
        // 删除临时表文件
        for (String filename : dbDirectory.list()) {
            if (filename.startsWith("temp"))
                new File(dbDirectory, filename).delete();
        }
    }

    public boolean isNew() {
        return isNew;
    }

    public int size(String fileName) throws IOException {
        FileChannel fc = getFile(fileName);
        return (int) fc.size() / Page.BLOCK_SIZE;
    }

    private synchronized FileChannel getFile(String fileName) {
        FileChannel fc = openFiles.get(fileName);
        // 如果map中没打开过
        if (fc == null) {
            File dbTable = new File(dbDirectory, fileName);
            RandomAccessFile f = new RandomAccessFile(dbTable, "r");
            fc = f.getChannel();
            openFiles.put(fileName, fc);
        }
        return fc;
    }

    synchronized void read(Block block, ByteBuffer buffer)
        try {
            buffer.clear();
            FileChannel fc = getFile(block.filename());

```

```

        fc.read(buffer, block.number() * buffer.capacity());
    } catch (IOException e) {
        throw new RuntimeException("Cannot read block " +
    }

}

synchronized void write(Block blk, ByteBuffer buffer) {
    try {
        buffer.rewind();
        FileChannel fc = getFile(blk.filename());
        fc.write(buffer, blk.number() * buffer.capacity());
    } catch (IOException e) {
        throw new RuntimeException("Cannot write block " +
    }
}

synchronized Block append(String filename, ByteBuffer buffer) {
    try {
        int newBlkNum = size(filename);
        Block newBlk = new Block(filename, newBlkNum);
        write(newBlk, buffer);
        return newBlk;
    } catch (IOException e) {
        throw new RuntimeException("Cannot append block " +
    }
}
}

```

注意，文件管理器总是以一个块大小的字节序列为单位，从相应的文件进行读/写的，而且总是在一个块的边界开始操作。这样做时，文件管理器将确保每次读取、写入或追加调用都只需要一次磁盘访问。

在Java中，一个文件在被使用之前必须被打开，SimpleDB的文件管理器会创建一个 `RandomAccessFile` 对象，然后再获得该文件的文件通道。方法 `getFile()` 管理了这些 `FileChannel` 对象，注意，这些文件是以“rws”的模式打开的，“rw”部分表示这个文件是为了读和写而打开，“s”表示告诉OS不要进行为了优化磁盘性能而作出的磁盘I/O延迟（disk I/O delay）。相反，每一次写操作都会被立即写回磁盘中。这个特点保证了数据库系统总是反映的是磁盘中真正包含的数据，这对于在第14章中会讲到的恢复管理（recovery management）尤为重要。

在整个SimpleDB中，总是只有一个 `FileMgr` 对象会被创建，这是通过包 `simplified.server` 中的 `Simple.init()` 方法实现的。`FileMgr` 的构造函数会确定指定的数据库文件夹是否存在，如果有必要，则会创建该目录。构造函数也会将所有的临时文件删除（这些临时文件可能由第22章的 物化运算符（materialized operators） 创建）

### 12.5.3 文件管理器单元测试

本章末尾的一些练习要求您修改SimpleDB的 `file` 包。你如何测试修改后的代码呢？一个不好的的测试策略是仅将修改后的代码插入服务器，然后查看JDBC客户端是否可以在该服务器上正常运行。使用这种策略不仅会导致调试困难而且十分乏味（例如，错误信息经常与实际的错误不符），而且我们无法知道JDBC客户端是否会对修改后的代码进行完全的测试。

一个更好的策略是进行 单元测试(`unit testing`)，使用这个策略，你将为每个类/包创建一个 测试驱动 (`test driver`) ;这个测试驱动应该测足够的测试用例从而让你自己信服——修改后的代码正常工作。例如，在 12.5.1小节中的代码片段就可以作为一个测试驱动的一部分测试用例，来测试 `simplesdb.file` 包中的代码。

通常来说，一个测试驱动应该在所测试代码的同一个包下，因此这样就可以调用一些包内私有的方法。因此，必须使用完全限定类名来运行代码，例如， `FileTest` 类就应该这样调用：

```
java simplesdb.file.FileTest
```

以下是 `FileTest` 测试类代码：

```

public class FileTest {

    public static void main(String[] args) {
        try {
            SimpleDB.init("studentdb");

            // 第0块
            Block blk = new Block("junk", 0);
            Page p1 = new Page();
            p1.read(blk);
            // 将第0块的第105字节开始的int整数加1
            int n = p1.getInt(105);
            assert (n == 0);
            p1.setInt(105, n + 1);
            p1.write(blk);

            // 重新读回第0块
            Page p2 = new Page();
            p2.read(blk);
            int added_n = p2.getInt(105);
            assert (added_n == n + 1);

            // 追加另一个block (即 block 1)
            Page p3 = new Page();
            p3.setString(20, "hola");
            blk = p3.append("junk");

            // 再次重新读回追加的block到内存
            Page p4 = new Page();
            p4.read(blk);
            String s = p4.getString(20);
            assert (s.equals("hola"));

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

## 12.6 本章总结

## 12.7 建议阅读

## 12.8 练习

# 第13章 内存管理及 包 `simpedb.log` 和 `simpedb.buffer` 的实现

在本章中我们将学习数据库中的两个组件：日志管理器（log manager）和缓冲管理器（buffer manager），它们各自负责特定的文件：日志管理器负责日志文件，而缓冲管理器负责用户的数据文件。

这两个组件面临的关键问题就是，如何高效地管理驻留在内存中的磁盘块内容，并对它们进行读写。一个数据库中的内容通常要比主存的大小大得多，因此组件可能需要将这些数据内容在磁盘和内存之间来回穿梭。我们将会学习到日志管理器和缓冲管理器的多种内存管理算法。日志管理器管理日志文件采用的是一种简单的，最优的算法；然而缓冲管理器采用的算法只能是一种最近似估计的策略。

## 13.1 数据库内存管理的两大原则

正如我们所看到的，数据库的数据是存储在磁盘块中的。为了获取到想要的块，数据库系统需要将其中的内容读入到内存页中。数据库系统将数据在内存和磁盘之间移动时，会遵循以下两个重要的原则。

内存管理的两个基本原则是：

1. 最小化磁盘访问次数
2. 不要依赖于虚存（virtual memory），虚存是操作系统中的概念。

### 原则1：最小化磁盘访问次数

考虑这样一个应用，它从磁盘中读入数据，然后在数据上做搜索，再执行一系列的计算并作出一些改变，最后将数据写回磁盘。我们怎样可以估计这些操作的执行用耗时？回想一下，RAM读写的速度大概比闪存快1000倍，比磁盘大约快10万倍，这意味着在绝大多数的实际场景下，将读/写数据块的用时，最少会比在RAM中处理数据的用时还要多。因此，数据库系统可以做的事情就是最小化磁盘访问次数。

最小化磁盘访问次数的一个办法是避免重复的磁盘访问，这种问题在多种计算场合都会出现，且差不多有一个标准的解决方案——那就是缓存（caching）。举例来说，CPU中有一个局部的硬件缓存单元用来缓存一些先前执行过的指令；假设下一条将要执行的指令在cache中，那么CPU不会从RAM中把这条指令load进来，而是从cache中直接获取到（译者注：CPU的执行速度和RAM的执行速度又差了好几个数量级，所以在CPU内部会存在cache，现在的CPU也包含多级cache）。再举一个例

子，一个浏览器也会将之前访问过的web页面缓存起来，下次用户请求相同的页面时（例如，点击回退上一个页面的按钮），浏览器会从缓存中得到网页内容而避免了再次通过网络请求内容。

数据系统会将磁盘块中的内容缓存在内存页中，它会跟踪哪些页包含了哪些块中的内容；通过这种方法，通过使用以及存在的内存页，就有可能可以满足客户端的请求，却避免了磁盘读写。类似地，数据库系统也只会在必要的时候将页中的内容写回到磁盘，希望可以通过单次的磁盘写入对页面的多次更改进行保存。

最小化磁盘访问非常重要以至于遍布在数据库系统的整个工作流程中。举例来说，数据库系统中选择的数据检索算法是特别选择的，因为数据库系统想要最小化磁盘访问。当一条SQL语句有多种可能的检索策略时，`planner` 会选择那个需要最少磁盘访问次数的策略。

### **原则2：不要依赖于虚存**

现代操作系统支持 虚存(virtual memory)，操作系统会给每个进程一个错觉，即它拥有大量的内存来存储其代码和数据。一个进程可以在其虚拟内存空间中任意分配对象；OS会负责将每个虚拟页面映射到实际的物理内存页面。

OS支持的虚存空间通常来说都远远大于一个计算机的物理内存空间。因为不一定实际的物理内存空间能放得下当前所有的虚存页，所以OS必须将一些虚存页暂时放到磁盘中。当一个进程访问一个不在主存中的虚存页时，一次 页换入换出 (page swap) 会被执行：具体来说，OS会选择一个物理页面，将该物理页面中的内容写入到磁盘中（如果有必要的话），然后将磁盘上对应的该虚存页的内容读入物理页中。

数据库系统管理磁盘块的最直接的办法就是给每个块一个它自己的页，例如，数据库系统可以为每个文件维护一个页数组，并为文件的每个块维护一个 槽 (slot)，这些页数组可能很大，但它们都是被放在虚存中的。当数据库系统需要相应的页时，OS的虚存管理机制会按照需求，在磁盘和内存中进行换入换出，这是一个很简单，且很容易实现的策略，不幸的是，这种策略有一个严重的问题：现在控制什么时候将页写回到磁盘的是OS，而不是数据库系统，这会引发两个危险的问题：

- 第一个问题是操作系统的页面交换可能会损害数据库系统在系统崩溃后恢复的能力。正如我们将在第14章中看到的那样，原因是被修改的页会有一个相关的日志记录，这些日志记录必须被在页被保存之前写回到磁盘（否则，在数据库系统崩溃后想要恢复时，就会找不到日志记录）。因为OS不知道这是日志，它可能将修改后的页换出而没有写写日志记录，因此破坏了恢复机制。
- 第二个问题是，OS不知道哪些页正在被使用，哪些页数据库系统根本不注意。OS可以作出有根据的猜测，例如LRU (Latest Recently Used) 置换算法，但是假如OS猜错了，它可能会将一些再次需要的

页换出，从而带来了两次不必要的磁盘访问。另一方面，一个数据库系统，对哪些页需要使用，有一个更明智的猜测。

因此，数据库系统必须管理它自己的页。通过分配相对少的可以完全装进物理内存的页，数据库系统就可以很好地管理页；这些页被称为数据库的 缓冲池 (buffer pool)。数据库系统会跟踪哪些页可以被换入换出。当一个块需要被读入页中时，数据库系统（而不是OS）会从缓存池在选择一个合适的页，如果有必要（译者注：什么是有必要？就是当页中的内容被修改了后），然后将该页中的内容（以及相应的日志记录）保存到磁盘，只有当这些工作完成了以后，才会将指定的块中的内容读到页中。（你可以看作是完成了一次数据库系统级别的swap）。

## 13.2 管理日志信息

无论什么时候用户对数据库系统进行修改时，数据库系统必须对这些更改进行跟踪，以防需要对这些操作进行撤回。描述一次修改的值被保存在一条 日志记录 (log record) 中，而日志记录又被保存在 日志文件 (log file) 中，新的日志记录总是被追加在日志文件的末尾。

日志管理器正是数据库系统中负责将日志记录写到日志文件中的那个部分。

日志管理器不知道日志记录中的内容——那是由 恢复管理器 (recovery manager) 负责的部分（参见第14章）。相反，日志管理器将日志视为一个只会增加的日志记录序列。

在本节中，我们将会学习日志管理器在将日志记录写到日志文件时是怎样管理内存的。文明将最开始考虑下述的算法，这是将日志记录追加到日志中的最直接的办法。

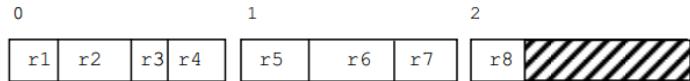
1. 在内存中分配页。
2. 将日志文件的最后一块读入该页中。
3.
  - a) 如果有页中有空间剩余，则将当前的日志记录追加到已有的所有日志记录。
  - b) 如果页中没有空间剩余，则分配一个新的空页，将当前日志记录放到这个

该算法每追加一条日志记录，需要一次磁盘读操作和一次磁盘写操作。这是非常简单的策略，但不是那么高效。

图13-2示例了日志管理器在上述3a过程中的执行流程。日志文件包含3个块，总共包含8条日志记录，用 $r_1$ 到 $r_8$ 进行了标识。每条日志记录可以有不同的大小，在block 0中包含4条日志记录，但在block 1中只装下了3条日

志记录，block 2中则还没满，只包含了1条日志记录。内存页中对应的是block 2中的内容，除了 $r_8$ 以外，内存页中还将一条新的日志记录 $r_9$ 放在了该页中。

Log File:



Log Memory Page:

Figure 13-2: Adding a new log record  $r_9$ 

现在假设日志管理器将内存页写回到文件的block 2中，从而完成以上算法。当日志管理器最终请求添加一条新的日志记录到日志文件时，它会执行步骤1和步骤2并且将block 2读入到内存页中。但是，请注意，这个磁盘读操作是不必要的，因为当前的日志页已经包含了日志文件block 2中的内容！（译者注：请思考下为什么？因为日志页是专门为日志管理器开辟的一系列内存页，而我们当前日志页中的内容肯定是上一次保存日志记录到文件后的样子啊！并没有人规定说，将日志页中的内容写回到日志文件后，必须把内存页中的内容也清空一遍，对吧？）因此，步骤1和步骤2都是不必要的，日志管理器只需要永久分配一个内存页，来保留上一次日志块中的内容即可。这样的结果是，所有的文件读操作全部都不再需要了，真的是妙~

其实，文件写操作也是有可能减少次数的，在上述的算法中，文件管理器在每次添加一条新的日志记录时，都会将日志页中的内容写回到磁盘，再看下图13-2，我们可以发现其实并不必要在 $r_9$ 被添加时立马将日志页写回到磁盘。只要日志页中还有空间剩余，每条新的日志记录可以被很简单地被加入到页中；当页满了的时候，日志管理器可以将页写回到磁盘中，然后再清除日志页中的内容，准备下一次开始添加新的日志记录。这个新的算法，对于日志文件中的每个块会相应进行一次磁盘写操作，而不是之前的每条日志记录，这明显更优。

我们的新算法有一个小问题：可能会出现某种突发情况，超出了日志管理器的控制范围（比方说，在日志页还没写满前，突然停电了，导致日志页中的某些日志记录可能还没写回到磁盘上），因此，在类似这种情况下，我们是反而需要在日志页满之前将日志页写回磁盘。出了停电这种突发情况，其实还有一种情况，也需要我们提前将日志页写回磁盘。回想一下，在相关的日志记录被写入磁盘之前，缓冲管理器是不能将已修改的数据页写入磁盘（这是为了可以确保到数据库的数据可以被恢复）。如果这些日志记录之一恰好位于日志页中但尚未持久化到磁盘上，则无论页面是否已

满，日志管理器都必须将其页面写入磁盘。（译者注：这就类似于文件操作中的 `flush()`，提供给了用户强制写回磁盘的权限，反正就是提前进行磁盘写操作。）

上述讨论产生了以下新的算法执行流程：

1. 分配一块永久的内存页来维持日志文件中最后一个块中的日志信息，把这个P设为一个指向该页的指针。
2. 当有一个新的日志记录到来时：
  - a) 如果P中没有空间剩余，则将P写入磁盘并清空P中的内容。
  - b) 将新的日志记录添加到P中。
3. 当数据库系统请求一个特殊的日志记录写操作时：
  - a) 确定该记录是否驻留在P中。
  - b) 如果在，将P写回磁盘。

换句话说，有2种原因导致内存页写回到磁盘：

- 日志记录强制被写回到磁盘
- 当日志页满了

因此，一个内存页可能被写回到同一个日志块中多次，但是因为文件写操作不可能被完全地避免，我们可以认为当前的这个算法是最优的。

## 13.3 SimpleDB的日志管理器

在本节中，我们将学习SimpleDB数据库系统的日志管理器。第13.3.1节介绍了日志管理器的API，这些API包括了将日志记录追加到日志文件的方法，以及读取日志文件中记录的方法。然后13.3.2小节显示了如何用Java来实现这个日志管理器。

### 13.3.1 日志管理器的API

SimpleDB日志管理器的实现在包 `simplesdb.log` 中，这个包有两个类：`LogMgr` 类和 `BasicLogRecord` 类；他们的API如下代码所示。

```

// LogMgr.java
public class LogMgr {
    public LogMgr(String logfile);

    public int append(Object[] rec);

    public void flush(int lsn);

    public Iterator<BasicLogRecord> iterator();
}

// BasicLogRecord.java
public class BasicLogRecord {

    public BasicLogRecord(Page logPage, int pos);

    public int nextInt();

    public String nextString();
}

```

数据库系统在启动时，会创建一个 `LogMgr` 类的对象，类 `server.SimpleDB` 会负责创建该对象时传入日志文件名给构造函数；并且有一个 `logMgr()` 方法返回该创建的对象。

方法 `append()` 会追加一条日志记录到日志文件，并返回一个整数。就日志管理器而言，一条日志记录只不过是一个任意长度的一系列值的数组，只不过每个值必须是整数或string类型，并且我们假设一条日志记录必须可以在一个内存页中放得下。`append()` 返回的整数标识了新的日志记录，这个标识符被称为 日志序列号 (Log Sequence Number, LSN)。

一个LSN标识了日志文件中的一条日志记录。

追加一条日志记录后并不会保证这条记录会立马被写入到磁盘中；相反，日志管理器会决定什么时候将日志记录写回到磁盘，正如13.2小节末尾概括的算法一样。一个客户端可以强制将一个指定的日志记录写回到磁盘，通过调用 `flush()` 方法，该方法的参数是日志记录的日志序列号；这个方法确保这条记录（准确地说，也包含追加这条记录之前的其他记录），会被写回到磁盘中。

一个客户端可以通过 `iterator()` 方法来读取日志文件中的所有日志记录，该方法返回的是一个Java的迭代器。每次调用迭代器的 `next()` 方法会返回一个对应的 `BasicLogRecord` 对象。一条基本的日志记录其实就是在页中的一系列的字节，调用者负责知道记录中有多少值，并且它们的具体取值是什么，读取一条日志记录中的唯一选择就是通过调

用 `nextInt()` 和 `nextString()` 方法， `nextString()` 假设日志记录中下一个值为 `string` 类型的值，然后读取到该字符串并返回，`nextInt()` 也是类似。

`iterator()` 方法返回的所有日志记录刚好是一个反的顺序，从最近的日志记录开始，不断往前移动，之所以以这样的顺序读取，是因为恢复管理器（后续章节会提到）希望这样查看日志记录。

```
SimpleDB.init("studentdb");
LogMgr logMgr = SimpleDB.logMgr();
int lsn1 = logMgr.append(new Object[]{"a", "b"});
int lsn2 = logMgr.append(new Object[]{"c", "d"});
int lsn3 = logMgr.append(new Object[]{"e", "f"});
logMgr.flush(lsn3);

Iterator<BasicLogRecord> iter = logMgr.iterator();
while (iter.hasNext()) {
    BasicLogRecord rec = iter.next();
    String v1 = rec.nextString();
    String v2 = rec.nextString();
    System.out.println("[" + v1 + ", " + v2 + "]");
}
```

上面的代码片段提供了一个如何使用上述API的示例，前半段代码将3条日志记录追加到日志文件，每天日志记录包含两个字符串，第一条记录是 `["a", "b"]`，第二条记录是 `["c", "d"]`，第三条记录是 `["e", "f"]`，接下来通过调用 `flush()` 方法，传递第三条日志记录的LSN进去，这次调用会确保3条日志记录都被写回到磁盘上；后半段的代码则迭代访问日志文件中的所有日志记录，以一种逆序的方式，然后打印出相应的字符串，最终的输出如图13-5 (b) 所示。

```
[e, f]
[c, d]
[a, b]
```

(b) The output of the code fragment

Figure 13-5: Writing log records and reading them

### 13.3.2 实现日志管理器

#### *LogMgr* 类

`LogMgr` 类的代码如下所示，它的构造函数传入的是日志文件的文件名，如果该文件为空，则创建一个新的空文件，并且追加一个新的日志块。构造函数也会分配专门来保存日志记录的一个日志页（取名为 `mypage`），并且初始化该页中的内容为日志文件的最后一块的内容。

```

public class LogMgr {
    // 标识最后一条日志记录的结束位置的指针，它本身也是在页中的内容
    // 即[LAST_POS...LAST_POS+3]这4个字节代表的整数标识了最后一条日志记录的结束位置
    public static final int LAST_POS = 0;

    private String logfile;
    // 日志页
    private Page mypage = new Page();
    private Block currentBlk;
    private int currentPos;

    public LogMgr(String logfile) throws IOException {
        this.logfile = logfile;
        int logSize = SimpleDB.fileMgr().size(logfile);
        // 如果日志文件为空，则为日志文件追加一个新的空块
        if (0 == logSize)
            appendNewBlock();
        else {
            // 否则先读入最后一块。
            // 注意，块号的下标从0开始，所以要减去1
            currentBlk = new Block(logfile, logSize - 1);
            mypage.read(currentBlk);
            // 最后一条日志记录的结束位置
            currentPos = getLastRecordPosition() + INT_SIZE;
        }
    }

    public synchronized int append(Object[] rec) {
        int recSize = INT_SIZE; // 一条记录的字节数
        for (Object obj : rec) {
            recSize += size(obj);
        }
        // 如果追加一条日志记录后，当前页放不下
        if (currentPos + recSize >= BLOCK_SIZE) {
            flush();
            appendNewBlock();
        }
        // 把当前这条日志记录中的值全部依次放入日志页中
        for (Object o : rec) {
            appendVal(o);
        }
        finalizeRecord();

        return currentLSN();
    }
}

```

```

public void flush(int lsn) {
    if (lsn >= currentLSN())
        flush();
}

public Iterator<BasicLogRecord> iterator() {
    flush();
    return new LogIterator(currentBlk);
}

/**
 * 返回当前LSN
 *
 * @return 即返回当前块的块号。
 */
private int currentLSN() {
    return currentBlk.number();
}

/**
 * 处理追加完日志记录后的动作。
 * <p>
 * 也就是：
 * 1. 先在当日志记录后面加上一个整数，用来标识上一条日志记录的：
 * 2. 再改变日志页的最开始的4个字节，用来直接标识最后一条日志记录
 * </p>
 * 这一部分类似一个逆着的数组链表，务必理清其中的逻辑
 */
private void finalizeRecord() {
    mypage.setInt(currentPos, getLastRecordPosition());
    setLastRecordPosition(currentPos);

    currentPos += INT_SIZE;
}

/**
 * 将日志记录中的值追加到日志页mypage中
 * <p>
 * TODO：目前该数据库系统只支持int和string类型，以后扩展后次方
 * 
 * @param obj 追加的值
 */
private void appendVal(Object obj) {
    if (obj instanceof String) {
        mypage.setString(currentPos, (String) obj);
    } else {
        mypage.setInt(currentPos, (Integer) obj);
    }
}

```

```

        currentPos += size(obj);
    }

    /**
     * 将当前页中的内容强制持久化到磁盘中
     */
    private void flush() {
        mypage.write(currentBlk);
    }

    /**
     * 计算一个obj的字节需要使用的字节数
     * <p>
     * TODO: 目前该数据库系统只支持int和string类型, 以后扩展后次方
     *
     * @param obj 待统计的对象
     * @return 字节数
     */
    private int size(Object obj) {
        if (obj instanceof String) {
            String strVal = (String) obj;
            return STR_SIZE(strVal.length());
        } else {
            return INT_SIZE;
        }
    }

    /**
     * 追加一个新的空块到日志文件
     */
    private void appendNewBlock() {
        // 设置最后一条日志记录的结束位置为0
        setLastRecordPosition(0);
        currentBlk = mypage.append(logfile);
        // 新分配的日志块肯定只有一个INT, 也就是4个字节
        // 该INT表示的是最后一条日志记录的结束位置, 在这里为0
        currentPos = INT_SIZE;
    }

    private void setLastRecordPosition(int pos) {
        // 第一个参数为offset, 第二个参数为具体的值
        mypage.setInt(LAST_POS, pos);
    }

    private int getLastRecordPosition() {
        return mypage.getInt(LAST_POS);
    }
}

```

回顾一下，LSN是用来标识一条日志记录的，例如，一个典型的LSN可能包含当前块号以及当前日志记录在该块的起始位置。然而，SimpleDB把事情简化了，而是只用一个块的块号来唯一标识当前日志记录，也就是说，一个块中的所有日志记录会有相同的LSN，这种简化丝毫不会降低程序的正确性，但的确会降低那么一些些的效率。

方法 `flush()` 会比较当前的块号和指定的LSN，如果指定的LSN更小，那对应的日志记录肯定已经被写回到了磁盘中；否则，`mypage` 会执行一次磁盘写操作。

`append()` 方法则会先一下计算当前待写入的日志记录需要用多少个字节来存储，从而确定当前日志页是否能够容纳下。如果容纳不下，它会首先将当前页中的内容写回到磁盘，然后调用一次 `appendNewBlock()` 方法来清除当前日志页，最后将新的日志记录写进去。这种策略和13.2小节末尾提到的算法有那么一丢丢不一样；那就是日志管理器通过追加一个空页来扩展日志文件，而不是追加一个满的页。这种策略更容易实现，因为它允许 `flush()` 方法假设这个块已经在磁盘中了。

一旦 `append()` 方法确保当前日志记录可以装进当前的日志页，那么它会依次将日志记录中的每个值都装到日志页中，最后执行 `finalizeRecord()` 方法。该方法会在新的记录后面再写一个int整数，用来标识上一条日志记录的其实位置，这样一来，页中的所有记录就像是一个倒着的链表一样。另外，页的第一个int，标识的是这个链表的头指针——在这里，因为链表是逆序的，也就是最后一条日志记录的起始位置。

0	
4	

(a) The empty log page

14	a	b	0	
4	9	14	18	

(b) The log page after appending the log record ["a", "b"]

28	a	b	0	c	d	14	
4	9	14	18	23	28	32	

(c) The log page after appending the log record ["c", "d"]

42	a	b	0	c	d	14	e	f	28	
4	9	14	18	23	28	32	37	42	46	

(d) The log page after appending the log record ["e", "f"]

Figure 13-7: Appending three log records to an empty log page

为了演示这个链表是怎样实现的, 请参阅图13-7。这个图演示了4次对日志页的操作: 当页被初始化的时候的状态, 以及3次添加日志记录后的状态。图13-7 (a) 只有一个值0, 这个值表示的是页中的最后一条记录在0位置结束 (也就是指向了自己), 因为当前页中根本没有日志记录。

图13-7 (b) 示例了当添加了日志记录 ["a", "b"] 后页中的内容, 回想一下, 一个只包含1个字符的字符串占用的字节数为5: 即4个字节来表示字符串长度 (当然, 这里就是数字1了), 另外一个字节表示字符串的字面量。这条日志记录后面的0表示上一条记录在0的位置结束, 其实也就是说, 这条记录是链中的最后一条记录。页首的值14表示链表的第一条记录 (在这里也是最后一条, 因为是逆的链表, 并且只有1个日志记录) 结束于14位置。

图13-7 (c) 和图13-7 (d) 演示了添加另外两条日志记录后的情况, 和图13-7 (b) 类似。

### LogIterator 类

LogMgr 类中的 iterator() 方法会先 flush 一下日志文件 (从而确保整个日志是在磁盘上), 然后再返回一个 LogIterator 对象。LogIterator 类实现了这个迭代器, 其代码如下所示。一个 LogIterator 对象分配一个内存页用来容纳日志文件的块, 构造器会

将迭代器的初始位置指向日志文件最后一块的最后一条记录。`next()` 方法使用相应的指针来往之前的日志记录不断迭代，最后返回一个当前位置对应的 `BasicLogRecord` 对象。

```
public class LogIterator implements Iterator<BasicLogRecord> {

    private Block blk;
    private Page page = new Page();
    private int currentRec; // 迭代器当前遍历的日志记录结束位置

    public LogIterator(Block blk) {
        this.blk = blk;
        page.read(blk);
        // 初始化为最后一条日志记录的结束位置
        currentRec = page.getInt(LAST_POS);
    }

    @Override
    public boolean hasNext() {
        return currentRec > 0 || blk.number() > 0;
    }

    @Override
    public BasicLogRecord next() {
        if (0 == currentRec)
            moveToNextBlock();
        // 继续往回迭代上一条
        currentRec = page.getInt(currentRec);
        return new BasicLogRecord(page, currentRec + INT_SIZE);
    }

    @Override
    public void remove() {
        throw new UnsupportedOperationException("Remove operation not supported");
    }

    private void moveToNextBlock() {
        // 上一个块
        blk = new Block(blk.filename(), blk.number() - 1);
        page.read(blk);
        currentRec = page.getInt(LAST_POS);
    }
}
```

### **BasicLogRecord** 类

`BasicLogRecord` 类的代码如下所示，构造函数接受一个日志页中指示一条日志记录的其实位置，会使用 `nextInt()` 和 `nextString()` 方法来读取后续的数据，然后在读的过程中会移动位置指针。

```
public class BasicLogRecord {

    private Page logPage;
    private int pos;

    public BasicLogRecord(Page logPage, int pos) {
        this.logPage = logPage;
        this.pos = pos;
    }

    public int nextInt() {
        int intValue = logPage.getInt(pos);
        pos += INT_SIZE;
        return intValue;
    }

    public String nextString() {
        String stringVal = logPage.getString(pos);
        pos += STR_SIZE(stringVal.length());
        return stringVal;
    }
}
```

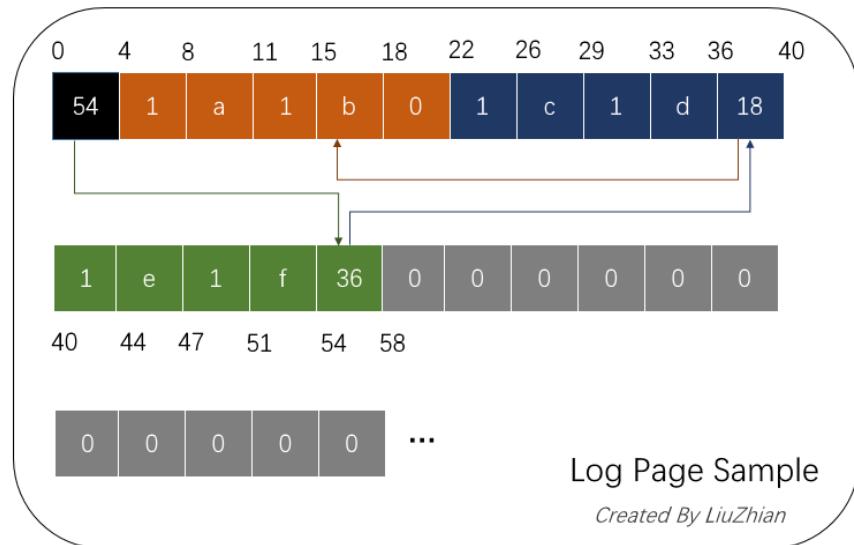
### 13.3.3 日志管理器测试结果（译者添加）

和之前类似，我们将测试代码写入一个 `LogTest` 类中，然后相应地修改一下 `server.SimpleDB` 中的初始化代码，最后执行之前提到的测试代码，运行后，用二进制文件编辑器打开日志文件，我的结果如下：

000	00	00	00	36	00	00	00	01	61	00	00	00	00	01	62	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	01
021	64	00	00	00	00	00	00	12	00	00	00	01	65	00	00	00	00	01	66	00	00	00	00	24	00	00	00	00	00	00	
042	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
063	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
084	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0A5	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0C6	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0E7	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
108	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
129	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
14A	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
16B	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
18C	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

在这里我们简单分析一下这个二进制文件中的内容，最开始的4个字节为 `00 00 00 36` 也就是十进制的54，对应的是图中第二行的 `00 00 00 24` 的第一个00，而 `00 00 00 24` 对应的十进制又为36，也就是说最后一条日志记录的上一条日志记录的结束位置为第36个字节；而第36个字节又对应的是图中第二行的 `00 00 00 12` 的第一个00，`00 00 00 12` 又代表的是十进制的18，这里我们可以知道，从紧接着 `00 00 00 12` 后面的第一个00开始，一直到 `00 00 00 24` 前面的最后一个00，就是日志文

件中的最后一条记录，也就是 `00 00 00 01 65 00 00,00 00 00 01 66 00 00`。看到我在这里特意打的逗号，想必你肯定知道是什么意思了！`00 00 00 01`就是十进制中的1，也就是我们底层表示一个字符串时的长度标识，而后面的3个字节，`65 00 00`就是‘e’的字面量，在我的机器上，Java用3个字节来表示一个英文字符。前面的两条记录信息可以用类似的方法解析出来，就不再赘述了，下图是这个二进制文件的“可视化”结果：



## 13.4 管理用户数据

日志文件以一种有限的、便于理解的方式被组织，日志管理器因此必须非常小心地调整其管理内存的方式；以simpleDB为例，日志管理器只需要一个专用的内存页就可以最佳地完成工作。

另一方面，JDBC应用程序完全不可预测地访问其数据。我们无法知道应用程序接下来要请求哪个块，以及是否会再次访问前一个块。即使在一个应用程序完全完成其块之后，我们也不知道另一个应用程序是否会在不久的将来访问这些块中的任何一个。本节将介绍在这种情况下，数据库系统如何有效地管理内存。

### 13.4.1 缓冲管理器

缓冲管理器是数据库系统中负责管理那些持有用户数据的内存页。

缓冲管理器维持着一系列固定数量的内存页，被称为 缓冲池（buffer pool），正如在本章的开头所提及的意义，缓冲池需要能在计算机的物理内存中装下，并且这些也应该来自OS维持的I/O buffer。

为了访问一个块，一个客户端会按照以下的协议与缓冲管理器进行交互：

1. 客户端请求缓冲管理器从缓冲池中 固定(pin) 一个页，用来维持块中的内容。

2. 客户端按照相应的逻辑，操作页中的内容。
3. 当客户端完成页的操作后，它告诉缓冲管理器 取消固定(unpin) 那个页。

当一个客户端正在pinning一个页时，我们称这个页是被 固定(pinned) 的状态；否则称这个页为 非固定(unpinned) 的状态。当一个页被客户端固定了后，缓冲管理器有义务确保该页一直可用；相反，一旦取消固定某个页，则允许缓冲区管理器将其分配给另一个块。

一个被固定的缓冲页的使用者为客户端。一个没被固定的缓冲页的使用者为缓冲管理器。

当一个客户端请求缓冲管理器为某个块而固定一个页时，缓冲管理器将会遇到以下可能的4种情况之一：

- 块中的内容已经在缓存中，而且：
  1. 这个页是被固定的，或者
  2. 这个页没被固定
- 块中的内容不在缓存中，而且：
  1. 缓冲池中至少有一个没被固定的页，或者
  2. 缓冲池中所有的页都被固定了

第一种情况会发生在，当一个或多个客户端正在访问某个块中内容的时候。由于一个页可以由多个客户端固定，因此缓冲管理器只需向该页面添加另一个引脚，然后将该页返回给客户端即可。正在固定该页的每个客户端都可以并发地同时读取和修改其值。缓冲管理器并不担心可能发生的潜在冲突；相反，这是并发管理器的工作（我们将在第14章中看到）。

第二种情况会发生在，当客户端使用完了缓冲区内容，但是还尚未重新使用这个缓冲区给另一个用户的时候，由于块中的内容仍然在缓冲页中，缓冲管理器可以通过简单地固定该页给客户端的方式，重用这个缓冲页。

第三种情况要求缓冲管理器将磁盘块中的内容读到内存缓存页中，这里有几个相关的步骤。缓冲管理器必须首先选择一个页来重新使用，这个页必须是没被固定的（因为被固定的页还在被客户端使用着）；其次，如果被选中的页被修改过，缓冲管理器必选先将该页中的内容写回到磁盘，这个操作被称为页的 *flushing* 操作；最后，块的内容可以被读到选中的页中，并且这个页可以被固定。

第四中情况会发生在，当缓冲区都大量被使用的适合，例如在23章将会说的的查询处理算法，在这种情况下，缓冲管理器无法满足客户端的请求。此时最好的解决办法就是缓冲管理器将这个客户端放在一个等待队列中。当一个缓冲页变成取消固定的时候，缓冲管理器可以将磁盘块中的内容读到页中，并固定，客户端就可以继续做后续的操作了。

### 13.4.2 缓冲区

缓冲池中的每个页都有一个相关的信息，例如这个页是否被固定，如果被固定了，固定的是哪个块上的内容，一个 缓冲区 (buffer) 就是包含这些信息的对象。缓冲池中的每个页都有自己的缓冲区。

每个缓冲区都会观察对页所作出的修改，并且对修改后的页写回磁盘负责。就像和日志一样，如果采用延迟写回磁盘的策略，缓冲区可以减少磁盘访问数，例如，如果一个页被修改了好几次，于是在所有的修改做完之后再将内存页写回磁盘，效率就会高的多。因此，一个合理的策略就是使缓冲区延迟将其页面写入磁盘，直到取消固定页面为止。

实际上，缓冲区可以甚至等更长的时间。假设一个修改后的页变成了非固定状态，但是没有写入磁盘。假如页再次被固定到相同的块（正如和第2种情况中所说的一样），客户端将会看到这些修改后的内容，正如被遗留的一样。这时，其实和这个页被写回磁盘，然后再读取回页中的情形，是一样的效果，但是没有磁盘访问。从某种意义上讲，缓冲区的也有点像是在内存中的磁盘块。任意一个想要使用块的客户端，将会被简单地带到缓存页中，然后客户端可以实现无需任何磁盘访问，从而读取或修改数据的操作。缓冲区需要将修改的页写回磁盘的原因只有两个：

- 要么缓冲区因为固定另外一个页面而被替换
- 要么恢复管理器需要将内容写回到磁盘，从而防止可能的系统崩溃发生。

### 13.4.3 缓存置换策略

缓冲区中的页最开始是没被分配的。一旦一个固定请求到来，缓冲管理器通过将被请求的块赋值到未分配的页上，从而填充缓冲池。一旦所有的页都被分配，缓冲管理器将会开始替换页，缓冲管理器可以选择缓冲池中的任意页，前提是该页面未被固定。

如果缓冲管理器需要替换一个页，而所有的缓存又被固定了，那么这个请求的客户端必须等待。因而，每个客户端有责任成为一个“好公民”，也就是说，一旦客户端不再需要某个缓存页后，应该立马将缓存页释放掉。一个恶意的客户端可能可以简单地通过固定所有的缓存并且让它们保持固定状态，从而使系统崩溃。

如果存在多于1个缓冲页是未固定的状态，那么缓冲管理器必须决定替换哪一个。这个选择会影响到后续执行的磁盘访问次数，例如，最糟糕的选择就是替换掉接下来将被访问的页，因为缓冲管理器将不得不又替换其他页。事实证明，最好的选择就是总是替换那些最长时间不会被使用的页。

既然缓冲管理器无法预测接下来哪些页将会被访问，于是它不得不被迫进行猜测。在这里，缓冲管理器几乎是和OS在swap虚存时的场景一样，但是，也有一个很大的区别：不像在OS中，缓冲管理器确确实实知道一个页当前是否被使用，因为没被使用的页正是那些没被固定的页。事实证

明，无法替换一个固定页的负担反而是一种幸运。客户端会通过负责任地固定页，从而不需要缓冲管理器真的去做一些糟糕的猜测。缓存替换策略只需要从那些当前不需要的页中作出选择，这也显得不是那么关键。

给定一系列没被固定的页，缓冲管理器需要决定，在这些页中的哪一个将会是那个最长时间不会被用到的页，例如，一个数据库通常有很多这样的页（例如16章会说到的目录文件），它们会在数据库的整个生命周期中时不时地被访问。缓冲管理器需要避免替换这些页，因为它们将会在相对很短的时间之内被重新固定。

和OS的虚存管理中一样，存在着一些尝试近似最好猜测的替换策略，我们将主要考虑4种替换策略：Naive，FIFO，LRU和Clock

图13-11介绍了我们将用来比较这几种替换算法的例子，(a)部分给出了一系列固定和取消固定一个文件中5个块的操作；(b)部分展示了缓冲池的结果状态，假设在这里只有4个缓冲区，只有当第5个块（即block 50）被固定时才需要替换页。然而，因为那时只有1个缓冲区是没被固定，缓冲管理器别无选择，也就是说，缓冲池将会和图13-11(b)所示的一样，无论采用什么页置换算法。

```
pin(10); pin(20); pin(30); pin(40); unpin(20);
pin(50); unpin(40); unpin(10); unpin(30); unpin(50);
```

(a) A sequence of ten *pin/unpin* operations

Buffer:	0	1	2	3
block#	10	50	30	40
time read in	1	6	3	4
time unpinned	8	10	9	7

(b) The resulting state of the buffer pool

Figure 13-11: The effect of some *pin/unpin* operations on a pool of 4 buffers

图13-11 (b) 中的每个缓冲区都有3个信息：块号，读入缓冲区的时间，被取消固定的时间，图中的时间对应的是图13-11 (a) 中操作的位置。

图13-11 (b) 中的缓冲区全是被取消固定的状态，假设现在缓冲管理器收到了两个固定请求：pin(60) 和 pin(70)

缓冲管理器将需要替换两个缓冲区，所有的缓冲区都可以替换，那它替换哪个呢？接下来的每个替换算法都有一个不同的回答

### Naive 策略

最简单的替换策略是依次遍历缓冲池，替换找到的第一个未固定的缓冲区。使用图13-11的示例，将块60分配给缓冲区0，将块70分配给缓冲区1。

此策略实现起来比较简单，但几乎没有其他建议。例如，再次考虑图13-11的缓冲区，并假设客户端反复固定和取消固定块60和70，如下所示：

```
pin(60); unpin(60); pin(70); unpin(70);pin(60); unpin(60);
pin(70); unpin(70)...
```

Naive替换策略将对这两个块使用只缓冲区0，这意味着每次固定块时都需要从磁盘中读取它们。这样带来的问题是缓冲池根本没有被完全利用起来，如果替换策略为块60和70选择了两个不同的缓冲区，则每个块可以只从磁盘读取一次——这是效率的巨大提高。

### FIFO 策略

Naive策略仅基于选择的便利性来选择一个缓冲区。FIFO策略则试图通过选择最近最少被替换 (least recently replaced) 的缓冲区 (即，待在缓冲池中时间最长的页面，从而最早被固定的页面也会最早被替换，也就是先进先出的意思)，从而变得更加智能。此策略通常比Naive策略更好，因为与最近获取的页相比，那些很早之前的页再次被使用到的可能性较小。在图13-11中，最早的页是“读入时间”值最小的页。因此，将块60分配给缓冲区0，将块70分配给缓冲区2。

FIFO是一个合理的策略，但是它也并不总是作出正确的决定，例如，一个数据库通常有一些进程使用的页，例如目录页，由于几乎每个客户端都会用到这些页面，因此，不要替换这些页才更有意义。然而，这些页最终将成为缓冲池中最早的页面，并且FIFO策略会选择替换它们。

FIFO策略的替换策略可以以两种方式来实现。第一种方式就是如图13-11 (b) 中所演示的那样，在每个缓冲页中维持一个当前页被替换的时间，替换算法到时候可以扫描一遍缓冲池，在那些没被固定的页中，选择最早的页来换掉。第二种更加高效的实现方法是，缓冲管理器维持一个指向缓冲区的指针列表，而这个列表又是以置换的时间为顺序进行排序的。置换算法会搜索该指针列表，列表中第一个没被固定的页将会被置换，然后该指针移动道列表的尾部。

### LRU 策略

FIFO策略是基于一个页是什么时候被添加到缓冲池的方法来进行置换的，一个和FIFO类似的策略是基于一个页是最后一次被访问的时间是什么时候的原则，改策略的缘由是近期没被使用的页面在不久的将来也不会被使用。而这个策略就被叫做 LRU策略 (Latest Recent Used)。在图13-11中，“取消固定时间”对应着该页最后被访问的时间，因此，block 60将会被赋值到缓冲池3中，block 70 将会被赋值到缓冲池0中。

LRU策略似乎是一个更高效的通用的策略，并且避免了替换一些经常被使用的页，并且FIFO的两种实现策略也可以适用于LRU算法中，唯一的变化就是，在LRU策略实现的时候，每次取消固定页面时（而不是在替换页面时），缓冲区管理器必须更新时间戳（对于第一种实现策略）或更新列表（对于第二种实现策略）。

### Clock 策略

时钟策略其实是上述几个策略的一个有趣组合，并且有一个简单直接的实现。正如和Naive策略中一样，该置换算法会扫描一遍缓冲池，选择第一个找到的未被固定的页，但区别就是，该算法在扫描时，总是以上一次置换的页为起点开始扫描，假如你把整个缓冲池看成是一个圈的话，那么该置换算法就像一个钟的指针一样在一个圈中打转，当发现一个可以置换的页面时时钟的指针停止转动，当下一次替换请求到来时，则又继续旋转。

图13-11 (b) 中的示例没有显示时钟的位置，其实它上一次置换的缓冲页为buffer 1，因此当前时钟的指针，指向的是紧跟buffer 1之后的缓冲区，也就是buffer 2。因此，继续上面的例子，如果后续到来的操作是 `pin(60); unpin(60); pin(70); unpin(70)`，那么block 60将被分配在buffer 2上，而blkcok 70则将被分配在buffer 1上。

时钟策略企图尽可能均匀地使用所有缓冲区。如果将页面是被固定了的，则时钟策略将会跳过该页，并且在检查了缓冲池中的所有其他缓冲区之前不会再考虑它。此功能使该策略具有LRU的风格，这个想法是，如果一个页被经常使用，则当一个替换请求到来时，我们最好不要替换这个经常被使用的缓冲区，因为它在不久的将来很有可能会被再次使用。如果真是这样，则将该页跳过，并给予它另一个不被替换的机会（译者注：因为此时时钟的指针指向它的下一个，需要再转一圈才有可能再考虑是否置换这个页）。

## 13.5 SimpleDB的缓冲管理器

在这一节中，我们将介绍SimpleDB数据库系统中的缓冲管理器，在13.5.1小节中将涵盖缓冲管理器的API，并且提供一个使用的用例，在13.5.2小节中将展示如何用Java来实现这些类。

### 13.5.1 缓冲管理器的API

SimpleDB缓冲管理器的实现在包 `simpledb.buffer` 中，这个包包含了两个类，`BufferMgr` 和 `Buffer`，以及一个接口 `PageFormatter`，如下所示：

```
// Buffer.java
public class Buffer {

    public int getInt(int offset);
    public String getString(int offset);
    public void setInt(int offset,int val,int txNum,int LSN);
    public void setString(int offset,String val,int txNum,LSN);
    public Block block();
}

// BufferMgr.java
public class BufferMgr {

    public BufferMgr(int numBuffers);
    public Buffer pin(Block blk);
    public void unpin(Buffer buffer);
    public Buffer pinNew(String fileName, PageFormatter pageFormatter);
    public void flushAll(int txNum);
    public int available();
}

// PageFormatter.java
interface PageFormatter {
    public void format(Page p);
}
}
```

在数据库系统被启动时，会创建一个 `BufferMgr` 对象，`simpedb.server` 包中的 `SimpleDB` 类会创建一个该对象，并传入一个缓冲池的大小给它的构造函数，有一个静态方法 `bufferMgr()` 会返回这个创建的对象。

一个 `BufferMgr` 对象拥有固定页面和取消固定页面的方法，`pin()` 方法返回一个固定到指定块的缓冲区，`unpin()` 方法则会取消固定这个缓冲区。一个客户端可以读取已固定的缓冲区中的具体内容，通过的是调用 `getInt()` 和 `getString()` 方法，下述的代码片段提供了一个使用缓冲区的例子，代码中先把文件 `junk` 的第3个块中的内容固定到缓冲区中，得到第392个字节和第20个字节开始的位置，然后取消固定该页。

```

SimpleDB.init("studentdb");
BufferMgr bufferMgr = SimpleDB.bufferMgr();
Block blk = new Block("junk", 3);
Buffer buffer = bufferMgr.pin(blk);
int n = buffer.getInt(392);
String str = buffer.getString(20);
// 客户端代码需要对unpin负责
bufferMgr.unpin(buffer);
System.out.println("Values are: " + n + ", " + str);

```

一个客户端可以通过调用缓冲区的 `setInt()` 和 `getString()` 方法，来修改块的值。然而，在修改之前，客户端必须首先将合适日志记录写到日志文件中，并且获得其LSN（译者注：忘了LSN是什么？就是一个日志的序列号，在我们现在的实现中，就简单地用块号来标识）。客户端将 LSN（和一个事务标识符）作为参数传入 `setInt()` 和 `getString()` 方法的。缓冲区会保存这些值，并在需要将日志记录强制写回到磁盘时使用它们。

下面的代码展示了缓冲区修改的例子，增加了第392个字节开始的整数，在调用 `setInt()` 方法之前，代码会创建一条日志记录，并且追加到日志文件，这条日志记录的LSN将传到 `setInt()` 方法中。

```

SimpleDB.init("studentdb");
BufferMgr bufferMgr = SimpleDB.bufferMgr();
Block blk = new Block("junk", 3);
Buffer buffer = bufferMgr.pin(blk);
int n = buffer.getInt(392);

LogMgr logMgr = SimpleDB.logMgr();
int myTxNum = 1; // 假设这里有个事务标识符1
Object[] logRec = new Object[]{"junk", 3, 392, n};
int LSN=logMgr.append(logRec);

buffer.setInt(392,n+1,myTxNum,LSN);
// 客户端代码需要对unpin负责
bufferMgr.unpin(buffer);

```

缓冲管理器并不介意具体创建的是一条怎样的日志记录，所以上述代码中的日志记录其实可以是任何东西，你想让它是什么，那它就是什么。但只是出于娱乐的目的，代码创建了一条和SimpleDB到时候真的日志记录格式差不多的日志记录。日志记录包含了4个值：文件名，块号，修改的值的位置（也就是起始字节数）和修改之前的值，如果到时候真的想要回撤所做出的修改，这些信息足够了！

`setInt()` 和 `setString()` 方法修改块中已经存在的值所以这些值创建的时候怎么初始化呢? SimpleDB假设每个块在被追加到文件时，都会被 格式化(formatted)，格式化这个操作，是通过一个实现了 `PageFormatter` 接口对象的 `format()` 方法来完成的。下面的代码片段演示了一个实现了 `PageFormatter` 接口的类例子，它会将在一个块中填满了一连串的“abc”字符串。这个formatter的代码本身倒没什么稀奇的，一个更有用且更加实际的实现会在后面具体实现小节中给出。

```
public class ABCStringFormatter implements PageFormatter {
    @Override
    public void format(Page p) {
        int recSize = STR_SIZE("abc".length());
        for (int i = 0; i + recSize <= BLOCK_SIZE; i += recSize) {
            p.setString(i, "abc");
        }
    }
}
```

`BufferMgr` 类中的 `pinNew()` 方法负责创建新的文件块，这个方法接受2个参数作为输入：文件名 和 一个页格式化器。这个方法会为这个新的块分配一个新的缓冲页，固定这个页，并且格式化好，随后将这个格式化好的页追加到文件后，并返回这个缓冲区给客户端。下面的代码展示了 `pinNew()` 方法是怎么使用的。代码现添加一个新的块到文件 `junk` 中（填满了字符串“abc”），随后又读取第一个值，从缓存页中获得该块的块号。

```
SimpleDB.init("studentdb");
BufferMgr bufferMgr = SimpleDB.bufferMgr();
PageFormatter pf=new ABCStringFormatter();
Buffer buffer = bufferMgr.pinNew("junk",pf);

String str=buffer.getString(0);
assert (str.equals("abc"));

int blkNum=buffer.block().number();
System.out.println("Appended block number: "+blkNum);
```

`BufferMgr` 类也包含了另外两个方法，方法 `flushAll()` 是用来清理一个指定事务的，它将flush该事务修改的每个缓冲区，从而使其更改永久生效。方法 `available()` 返回当前未被固定的缓冲区数量，该值会由第23章中将会提到的查询评估算法使用。

注意一下缓冲管理器是怎样把实际底层的磁盘访问细节向客户端忽略掉的。现在，客户端根本不知道它的行为（即代码调用）将引发多少次的磁盘访问，也不知道什么时候将发生磁盘访问。读磁盘的操作只有在调

用 `pin()` 方法时才会发生，更具体地说，应该是一个客户端请求了一个不在缓冲区中的块时，读磁盘的操作才会发生。而写磁盘的操作会在调用 `pin()` 和 `pinNew()` 方法时才会发生（假设替换的缓冲页已经被修改），以及会在调用 `flushAll()` 方法时发生（如果对应的事务修改了任何一个缓冲页）。

假设数据库系统有很多客户端，所有的客户端都使用很多的缓冲页。有可能缓冲池中的所有缓冲区都将被固定，在这种情况下，缓冲管理器不能立马满足一个 `pin()` 或者 `pinNew()` 请求，相反，它会将当前的调用线程放在一个等待队列上，知道有一个缓冲区可用，然后再返回该缓冲区。也就是说，客户端根本意识不到有这种缓冲连接，客户端只会发现程变慢了一些。

这种缓冲连接的方式，在有一种情况下会引发严重的问题（译者注：偷偷提前透露一下，等待有可能会产生什么问题？有可能死锁吗？）。考虑下面这样一个场景，客户端A和客户端B都需要2个缓冲页，但是只有2个缓冲页可用。假设客户端A固定了第一个缓冲页，然后B被添加到了等待队列，现在第二个缓冲页开始供A和B竞争（race）。假如客户端A先抢占到，A会接下来固定第二个缓冲页（这个时候B肯定还在等），然后在执行完自己的操作后，正常地把两个缓冲页全部取消固定，客户端B又可以固定、操作、取消固定，恩，这是一个没问题的场景。可是现在又假设，A和B在竞争时，B先抢占到，B嘛，想要2个缓冲页，而实际上只有1个，怎么办？等A用完。假设整个系统就2个客户端，于是A等B，B等A，此时A和B就产生了死锁现象。

在真实的数据库系统中，通常拥有几千个缓冲页并且好几百个客户端，这种死锁的情况很大概率不会发生，然而，缓冲管理器必须有处理这个可能的能力，SimpleDB采取的解决方案是最终每个客户端等缓冲页的时间

（就是等了多久），如果某个客户端等了很久（比方说，10秒），那么缓冲管理器假设这个客户端处于死锁状态，并且排除一个 `BufferAbortException`。客户端需要为处理这些异常负责，很典型的一个处理方案就是通过事务回滚，并且重新执行客户端的请求。

## 13.5.2 实现缓冲管理器

### **Buffer** 类

下面的代码包含了 `Buffer` 类的实现。

```

public class Buffer {
    private Page contents = new Page();
    private Block blk = null;
    // 当前缓冲页固定的次数，有点多线程中ReentrantLock的意思，一个客
    private int pins = 0;
    // 和事务相关 TODO
    private int modifiedBy = -1; // 当前缓冲区对应哪个事务
    private int logSequenceNum = -1; // LSN

    public int getInt(int offset) {
        return contents.getInt(offset);
    }

    public String getString(int offset) {
        return contents.getString(offset);
    }

    public void setInt(int offset, int val, int txNum, int
        // 和事务相关 TODO
        modifiedBy = txNum;
        // LSN的当前实现就是日志文件块的块号
        if (LSN >= 0)
            logSequenceNum = LSN;
        contents.setInt(offset, val);
    }

    public void setString(int offset, String val, int txNum,
        // 和事务相关 TODO
        modifiedBy = txNum;
        // LSN的当前实现就是日志文件块的块号
        if (LSN >= 0)
            logSequenceNum = LSN;
        contents.setString(offset, val);
    }

    public Block block() {
        return blk;
    }

    /**
     * 将缓冲页中的内容写回到磁盘，且在写回数据到磁盘前，追加一条日志
     * <p>
     * 此方法有点类似OS中处理内存的脏读位 (dirty-read) 的行为，当脏
     * OS会先将内存中的值写回文件块，再执行后续的磁盘文件读入内存的操作
     */
    void flush() {
        // 如果有修改页中内容：
        // 1. 先flush一下日志记录，
    }
}

```

```

    // 2. 再将内存页的内容写回磁盘
    if (modifiedBy >= 0) {
        SimpleDB.logMgr().flush(logSequenceNum);
        contents.write(blk);
    }
}

void pin() {
    pins++;
}

void unpin() {
    pins--;
}

boolean isPinned() {
    return pins > 0;
}

boolean isModifiedBy(int txNum) {
    return txNum == modifiedBy;
}

/**
 * 将指定的块中内容，赋值到缓冲页上。
 * <p>
 * 注意，在赋值前，要检查下当前页的内容是否被修改过！
 * 如果被修改过，必须先在写回磁盘前写一条日志记录，然后再执行相关
 * <p>
 * 有点类似Buffer类的构造函数。
 *
 * @param b 待写回的块
 */
void assignToBlock(Block b) {
    flush();
    blk = b;
    contents.read(blk);
    pins = 0;
}

/**
 * 将缓冲页的内容格式化，再追加到文件块
 * <p>
 * 注意，在追加磁盘块前，也要检查下当前页的内容是否被修改过！
 * 如果被修改过，必须先在写回磁盘前写一条日志记录，
 * 然后再执行缓冲页的格式化操作，再追加回磁盘块中。
 *
 * @param fileName

```

```

 * @param pfm
 */
void assignToNew(String fileName, PageFormatter pfm) {
    flush();
    pfm.format(contents);
    blk = contents.append(fileName);
    pins = 0;
}
}

```

一个缓冲页对象追踪了关于这个页的以下4种信息：

- 一个赋值到内容到该页上的块引用。假如没有块被赋值，这个值为 null。
- 这个缓冲页被固定的次数。每固定一次，该值加1；每取消固定一次，该值减1。
- 一个表示当前缓冲页内容是否被修改的整数。-1表示没被修改，否则标识的是使得作出该修改的事务标识符。
- 日志信息。如果当前缓冲页被修改，于是会持有一个最近的日志记录的序列化，即 LSN，LSN是非负的。如果一个客户端调用 Buffer 类的 setInt() 或 setString() 方法时，传入了一个负的LSN，那意味着将不生成一条对应这次修改的日志记录。

`flush()` 方法确保缓冲区的页中的内容和所赋值的块中的内容一致。如果缓冲区没被修改，那么这个方法什么也不干；如果被修改过，那么这个方法会先调用 `LogMgr.flush()` 方法确保对应日志记录被写到文件上，随后再将缓冲页中的内容写回到磁盘上。

`assignToBlock()` 和 `assignToNew()` 方法将缓冲区和一个磁盘块联系起来。在这两个方法中，缓冲区首先都会被 `flush` 一下，从而确保之前块的任何修改都被持久化到磁盘上了。`assignToBlock()` 方法通过把块中的内容读入到缓冲页中，从而实现缓冲区和块的联系；而 `assignToBlock()` 方法则通过执行页格式化操作，然后把这个页追加到文件尾部，并且将追加的块和当前缓冲页联系起来。

### **BasicBufferMgr** 类

在 SimpleDB 中，缓冲管理器被分成了两个类来实现：`BufferMgr` 和 `BasicBufferMgr` 类。`BasicBufferMgr` 类管理缓冲池，`BufferMgr` 则管理等待队列，`BasicBufferMgr` 类的实现如下。

```

public class BasicBufferMgr {

    private Buffer[] bufferPool; // 缓冲池
    private int numAvailable; // 空闲缓冲区数量

    public BasicBufferMgr(int numBuffers) {
        this.numAvailable = numBuffers;
        bufferPool = new Buffer[numAvailable];
        for (int i = 0; i < numBuffers; i++) {
            bufferPool[i] = new Buffer();
        }
    }

    synchronized void flushAll(int txNum) {
        for (Buffer buffer : bufferPool) {
            if (buffer.isModifiedBy(txNum)) {
                buffer.flush();
            }
        }
    }

    /**
     * 固定一个块的内容到一个缓冲区中
     *
     * @param blk 待固定的块
     * @return 固定成功的缓冲区对象 或 null (表示需要等待)
     */
    synchronized Buffer pin(Block blk) {
        Buffer buff = findExistingBuffer(blk);
        // 1. 如果没有缓冲区的内容就是待关联的块，则找一个没被固定的
        if (null == buff) {
            // TODO 怎么选取一个空闲的缓冲区可以用不同的策略
            buff = chooseUnpinnedBuffer();
            // 1.1 如果不存在没被固定的缓冲区，则返回null
            if (null == buff)
                return null;
            // 1.2 找到了一个没被固定的块，则将块的值赋上
            buff.assignToBlock(blk);
        }
        // 2. 如果存在一个缓冲区的内容就是待关联的块，此时有2种情况
        // 2.1 该缓冲区已经被固定，即 pins > 0,有可能是当前客户端
        //      在这里，我们并不关心是被缓冲区是被哪个客户端pin的。
        // 2.2 如果该缓冲区没被固定，即该缓冲区上的block是新替换的,
        if (!buff.isPinned())
            numAvailable--;
        // pins++
        buff.pin();
    }
}

```

```

        return buff;
    }

    /**
     * 以指定的格式化器，格式化缓冲区中页的内容，并将内容追加一个新块
     *
     * @param fileName 文件名
     * @param pfmt      指定的格式化器
     * @return 固定成功的缓冲区对象 或 null (表示需要等待)
     */
    synchronized Buffer pinNew(String fileName, PageFormat pfmt) {
        Buffer buff = chooseUnpinnedBuffer();
        if (null == buff)
            return null;
        buff.assignToNew(fileName, pfmt);
        numAvailable--;
        buff.pin();

        return buff;
    }

    /**
     * 减少一个缓冲区的固定次数
     * <p>
     * 注意，减少一次后不一定这个页就“自由了”
     *
     * @param buff 待取消固定的缓冲区
     */
    synchronized void unpin(Buffer buff) {
        buff.unpin();
        if (!buff.isPinned())
            numAvailable++;
    }

    /**
     * 得到“自由的”缓冲区数量
     *
     * @return int
     */
    int available() {
        return numAvailable;
    }

    /**
     * 查找是否存在一个缓冲区，其内容块引用与待查找的一致。
     *
     * @param block 待查找的块引用
     */
}

```

```

    * @return 如果存在，就返回那个缓冲区；否则返回null
    */
    private Buffer findExistingBuffer(Block block) {
        for (Buffer buff : bufferPool) {
            Block blkInBuffer = buff.block();
            if (blkInBuffer != null && blkInBuffer.equals(block))
                return buff;
        }
        return null;
    }

    /**
     * 在缓冲池中找一个没被固定的页
     * <p>
     * TODO：当前用的最简单的Naive算法，找到一个就OK，后期考虑其他策略
     *
     * @return 如果存在，就返回那个缓冲区；否则返回null
     */
    private Buffer chooseUnpinnedBuffer() {
        for (Buffer buff : bufferPool) {
            if (!buff.isPinned())
                return buff;
        }
        return null;
    }
}

```

方法 `pin()` 的功能是将指定块的内容赋值到一个缓冲区上，这个算法有两部分（可以回顾一下13.4.1小节）。第一部

分，`findExistingBuffer()`，这个方法会尝试去寻找一个已经被赋值为指定块内容的缓冲区，如果找到了，就返回这个缓冲区；然后算法的第二部分，`chooseUnpinnedBuffer()`方法，使用Naive策略来准备置换一个未固定的页，随后再使用那个缓冲区的`assignToBlock()`方法，把块的内容赋值到缓冲区上。注意，如果找不到一个没被固定的缓冲区（也就是即将进入等待队列），那么该方法返回null。

方法 `pinNew()` 的功能是将缓冲区上的内容追加到一个指定的文件尾部，这个方法和 `pin()` 方法很类似，但是这个方法会直接调用 `chooseUnpinnedBuffer()` 方法。（这里没必要调用 `findExistingBuffer()` 方法，因为块中的内容实际上还没存在，这也是为什么我们这里传递了一个格式化器进来，用以格式化块的初始内容）。然后再调用 `assignToNew()` 方法，会相应的创建一个新块。注意，如果找不到一个没被固定的缓冲区（也就是即将进入等待队列），那么该方法返回null。

### **BufferMgr** 类

`BasicBufferMgr` 类其实就完成了缓冲管理器的基本上所有功能，只有一个没实现：那就是当找一个未固定的缓冲池返回为`null`的情况，即上述 `synchronized Buffer pin(Block blk)` 方法返回`null`的情况。`BufferMgr` 类包装了 `BasicBufferMgr` 类，重新定义了 `pin()` 和 `pinNew()` 方法，具体代码如下所示。

```

public class BufferMgr {

    // 最长等待时间
    private static final long MAX_TIME = 10000;

    private BasicBufferMgr basicBufferMgr;

    public BufferMgr(int numBuffers) {
        basicBufferMgr = new BasicBufferMgr(numBuffers);
    }

    /**
     * 对BasicBufferMgr类中pin方法的包装
     *
     * @param blk
     * @return
     */
    public synchronized Buffer pin(Block blk) {
        try {
            long timestamp = System.currentTimeMillis();
            Buffer buff = basicBufferMgr.pin(blk);
            while (null == buff && !waitTooLong(timestamp))
                // this.wait(), 等待的对象是当前这个缓冲管理器,
                // 等待的目标是有一个未被固定的缓冲区
                wait(MAX_TIME);
            buff = basicBufferMgr.pin(blk);
        }
        // TODO 这里非常重要
        // 如果因为死锁或其他各种原因, 导致等待时间超过阈值 MAX_TIME
        if (null == buff)
            throw new BufferAbortException();

        return buff;
    } catch (InterruptedException e) {
        throw new BufferAbortException();
    }
}

public synchronized void unpin(Buffer buffer) {
    basicBufferMgr.unpin(buffer);
    // 一旦有一个缓冲区“自由”, 通知其他所有在缓冲管理器对象上等待
    if (!buffer.isPinned())
        notifyAll();
}

/**
 * 对BasicBufferMgr类中pinNew方法的包装
 *
 */

```

```

    * @param fileName
    * @param pageFormatter
    * @return
    */
    public synchronized Buffer pinNew(String fileName, PageFormatter pageFormatter) {
        try {
            long timestamp=System.currentTimeMillis();
            Buffer buff = basicBufferMgr.pinNew(fileName, pageFormatter);
            while (null == buff && !waitTooLong(timestamp)) {
                // this.wait(), 等待的对象是当前这个缓冲管理器,
                // 等待的目标是有一个未被固定的缓冲区
                wait(MAX_TIME);
                buff = basicBufferMgr.pinNew(fileName, pageFormatter);
            }
            // TODO 这里非常重要
            // 如果发生了死锁
            if (null == buff)
                throw new BufferAbortException();

            return buff;
        } catch (InterruptedException e) {
            throw new BufferAbortException();
        }
    }

    public void flushAll(int txNum) {
        basicBufferMgr.flushAll(txNum);
    }

    public int available() {
        return basicBufferMgr.available();
    }

    private boolean waitTooLong(long startTime) {
        return System.currentTimeMillis() - startTime > MAX_WAIT_TIME;
    }
}

```

在包装后的方法中，如果找不到一个未被固定的缓冲区，方法不会返回一个null，相反，会调用Java中的`wait()`方法。在Java中，每个对象都有一个等待队列，对象的`wait()`方法会打断当前正在执行的线程，并将它放到这个对象的等待队列中去。在上述代码中，只有当以下的两种情况之一发生时，线程才会被从等待队列中移除并且准备继续执行：

- 另外一个线程调用了`notifyAll()`方法（即当有一个`unpin`操作发生时）

- `MAX_TIME` 的时间到了，也就意味着等了很长时间。

当一个等待的线程恢复执行时，它会继续打转，尝试找到一个缓冲区。也就是说，一个等待线程将被重新放到等待队列上，除非等待的时间超过了预定的阈值或者有缓冲区变成了未固定状态。

方法 `unpin()` 会调用 `BasicBufferMgr` 类对象的 `unpin()` 对减少指定缓冲区的固定次数，随后会检查该缓冲区是否“自由”（译者注：即缓冲区的 `pins` 是否大于0），如果减少一次后，缓冲区变得“自由”了，那就会随之调用 `notifyAll()` 方法来将所有的等待线程全部唤醒，这些线程被唤醒后将抢占缓冲区，谁先抢到谁得，如果一个线程抢到后，又没有被固定的缓冲区，那么其他线程又必须再次等待，再次等待被唤醒....一直这样下去。

## 13.6 章末总结

- 数据库系统必须努力最小化磁盘访问次数。因此，它仔细管理用于保存磁盘块的内存页。管理这些页的数据库组件是 日志管理器 (`log manager`) 和 缓冲管理器 (`buffer manager`) 。
- 日志管理器负责将日志记录保存在日志文件中。由于日志记录始终附加在日志文件中，并且从未修改，因此日志管理器可以非常高效。它只需要分配一个页，并且具有一种简单的算法，可以将该页写入磁盘的次数最少。
- 缓冲管理器分配一些页，称为 缓冲池 (`buffer pool`) ，以用来处理用户数据。缓冲管理器应客户端的请求将缓冲区固定和取消固定到磁盘块。固定后，客户端访问缓冲区的页面，并在完成后取消固定缓冲区。
- 在两种情况下，修改后的缓冲区将被写入磁盘：①替换页时，②恢复管理器需要将页存储在磁盘上时。
- 当客户端要求将缓冲区固定到块时，缓冲管理器选择适当的缓冲区。如果该块已经在缓冲区中，则使用该缓冲区；否则，缓冲管理器将替换现有缓冲区的内容。
- 确定替换哪个缓冲区的算法称为缓冲区替换策略。四种有趣的替换策略是：①FIFO：选择最近一段时间内，被替换次数最少的未固定缓冲区。②LRU：选择最近一段时间内，被取消固定次数最少的未固定缓冲区。③时钟：从最后替换的缓冲区开始顺序扫描，选择的第一个未固定的缓冲区。

## 13.7 建议阅读

## 13.8 练习

# 第1章

## 第14章 事务管理和包 simpledb.tx

缓冲管理器允许多个客户端同时访问同一缓冲区，从中任意读取和写入值。这样的造成结果可能是混乱的：每次客户端查看页面时，页面可能具有不同（甚至不一致）的值，从而使客户端无法获得准确的数据库视图。或两个客户端可以在不经意间覆盖彼此的值，从而破坏数据库。

在本章中，我们将学习 恢复管理器（recovery manager） 和 并发管理器（concurrency manager），它们的任务是为何数据库的秩序和数据的一致性。每个客户端程序都按照一系列事务而组织，并发管理器调节这些事务的执行以使得它们的行为保持一致。恢复管理器从日志文件中读/写日志记录，以便于在必要的适合，可以 撤销（undo） 未提交事务所作出的更改。本章介绍了这两个管理器的功能以及实现它们的技术。

### 14.1 事务

#### 14.1.1 正确的代码执行结果却不正确

考虑一个航班预定数据库，有2个数据库表，它们的schema如下：

```
SEATS(FlightId, NumAvailable, Price)  
Cust(CustId, BalanceDue)
```

下面的代码包含了为指定的客户购买一张指定航班机票的JDBC代码、虽然代码本身没有bug，但是当多个客户端并发执行或者服务器崩溃的时候，很多问题都有可能产生。

```

public void reserveSeat(Connection conn, int custId, int flightId) {
    Statement stmt= conn.createStatement();
    String s;

    // 步骤1: 获取剩余票数和价格
    s="SELECT NumAvailable, Price FROM SEATS "+
       " WHERE FlightId= "+flightId;
    ResultSet rs= stmt.executeQuery(s);
    if(!rs.next()){
        System.out.println("Flight doesn't exist!");
        return;
    }
    int numAvailable = rs.getInt("NumAvailable");
    int price = rs.getInt("Price");
    rs.close();

    if(numAvailable==0){
        System.out.println("Flight is full!");
        return;
    }

    // 步骤2: 更新剩余票数
    int newNumAvailable = numAvailable - 1;
    s="UPDATE SEATS set NumAvailable =" +newNumAvailable+
       " WHERE FlightId= "+flightId;
    stmt.executeUpdate(s);

    // 步骤3: 获取并更新用欠款
    s="SELECT BalanceDue FROM CUST " +
       " WHERE CustId= "+custId;
    rs=stmt.executeQuery(s);
    int newBalance = rs.getInt("BalanceDue")+ price;
    rs.close();

    s="UPDATE CUST set BalanceDue =" +newBalance+
       " WHERE CustId= "+custId;
    stmt.executeUpdate(s);
}

```

下述的3个场景示例了这些问题。

①第一个场景，假设两个客户端A、B都并发地运行上述JDBC代码，具体的执行序列如下所示：

- 客户端A执行完步骤1的所有步骤后被打断
- 客户端B执行，直到完成
- 客户端A完成剩余操作

在这个例子中，两个线程都会使用相同的变量值 `numAvailable`，这样下来的结果是，两个座位都会被卖出，但是数据库中的剩余座位数只会被减少1次。

②第二个场景，假设线程C正在运行代码，然后再执行完步骤2后，系统崩溃了。在这种情况下，座位将被预定，但是顾客不会被收钱。

③第三个场景，假设一个客户端执行代码直到完成，但是被修改的页没有立即被写回磁盘因为缓冲的原因。假如服务器崩溃了，然后就再也没有办法知道哪个页最终需要写回到磁盘。如果上述代码中的第一次更新写回了磁盘（即剩余机票数），却第二次没有（即更新用户欠费），那么顾客得到了一张免费机票；如果第二次更新写回到了磁盘，第一次却没有，那么客户为一张根本系统中不存在的机票记录买了单；假如两次更新都没写回磁盘，这个情况到还好，就是整个交互记录丢失了，系统根本感受不到用户曾经买过票。

### 14.1.2 事务的属性

在上述的场景说明，如果客户端代码随意运行，那么数据就可能丢掉或损坏。数据库系统通过强制客户端程序包含事务来解决这些问题。

事务是一组操作，它们的行为被看做一整个操作来看待。一个事务必须满足 ACID 属性。

“一整个操作”的可以被概括为所谓的ACID属性：原子性(Atomicity)，一致性(Consistency)，独立性(Isolation) 和 持久性(Durability)。

- 原子性的意思是，一个事务要么做完，要么就不做。也就是说，要么事务中的所有操作都执行成功（事务commit了），要么事务执行失败（事务rollback了）
- 一致性的意思是，每个事务都会使得数据库系统处于一个一致性的状态。这意味着每个事务都是可以执行的完整工作单元，且独立于其他事务。
- 独立性的意思是，一个事务的行为在自己看来，就好像整个数据库只在执行当前这个事务。也就是说，如果多个事务并发执行，它们的结果应该和这些事务顺序执行的结果一样，不会因为并发而相互影响。
- 持久性的意思是，提交了的事务修改必须保证持久化到磁盘上。

14.1.1小节中示例的每一个问题都或多或少地违反了上述ACID原则。第一个场景违反了独立性原则，因为每个客户端都读到了相同的 `numAvailable` 变量值，然而顺序执行两个客户端的代码将导致第二个执行的客户端读到第一个用户修改后的数据；第二个场景违反了原子性原则；第三个场景违反了持久化原则。

原子性和持久性描述了commit操作和rollback操作的正确行为。

特别是，已提交的事务必须是持久的，而未提交的事务（由于显式回滚或系统崩溃）必须完全撤消其更改。这些功能是恢复管理器的责任，那是14.3节的主题。

一致性和平独立性描述了客户端并发操作的正确行为。

特别是，数据库服务器必须防止客户端之前发生冲突，一个典型策略的就是检测何时冲突可能发生，然后让一个客户端等待，直到冲突不再可能发生为止。这些功能都是并发管理器的职责，是14.4节的主题。

## 14.2 在SimpleDB中使用事务

在我们深入到恢复管理器和并发管理器的细节之前，对客户端如何使用事务有个初步的了解，会帮助我们更好地理解。在SimpleDB中，每个JDBC事务都有一个 `Transaction` 对象，它的API如下所示：

```
public class Transaction {

    public Transaction();
    public void commit();
    public void rollback();
    public void recover();

    public void pin(Block blk);
    public void unpin(Block blk);
    public int getInt(Block blk,int offset);
    public String getString(Block blk,int offset);
    public void setInt(Block blk,int offset,int val);
    public void setString(Block blk,int offset,String val);

    public int size(String fileName);
    public Block append(String fileName, PageFormatter pfm
}


```

`Transaction` 类的方法被分为3大类别，第一类包含了事务生命周期的方法，构造函数开始一个新事务，`commit()` 和 `rollback()` 方法结束一个事务，`recover()` 方法会为所有没提交的事务执行回滚方法。

第二类包含了访问磁盘块的方法，这些方法和缓冲管理器中的方法很类似，但一个重要的区别就是，一个事务会将中间存在的缓冲区向客户端隐藏掉，并且不会返回给客户端缓冲区。当客户端执行 `getInt()` 方法时，传入的是一个块的引用，事务会相应地寻找缓冲区，调用缓冲区的 `getInt()` 方法，然后把结果传递回客户端。事务把缓冲区的细节向

客户端隐藏掉，因此可以对并发管理器和恢复管理器进行必要的调用。例如，`setInt()`方法代码会获得适当的锁（并发控制锁），然后会在修改缓冲区之前，将缓冲区中当前的值写入日志文件（用于恢复）。

第三类包含了和文件管理器相关的两个方法。方法`size()`会读取文件的末尾，方法`append()`会修改它；这些方法会被并发管理器调用从而避免潜在的冲突。

下面的代码片段演示了对事务类的简单实用：

```
public class TransactionTest {

    public static void main(String[] args) {
        SimpleDB.init("studentdb");
        Transaction tx = new Transaction();
        Block blk = new Block("junk", 3);
        tx.pin(blk);
        int n = tx.getInt(blk, 392);
        String str = tx.getString(blk, 20);
        tx.unpin(blk);
        System.out.println("Values are " + n + " and " + str);

        tx.pin(blk);
        n = tx.getInt(blk, 392); // 在同一事务内，这条语句必要叫
        tx.setInt(blk, 392, n + 1);
        tx.unpin(blk);

        PageFormatter pfmt = new ABCStringFormatter();
        Block newBlk = tx.append("junk", pfmt);
        tx.pin(newBlk);
        String s = tx.getString(newBlk, 0);
        assert (s.equals("abc"));
        tx.unpin(newBlk);
        int newBlkNum = newBlk.number();
        System.out.println("The first string in block "
            + newBlkNum + " is" + newBlkNum);

        tx.commit();

    }
}
```

该代码由三部分组成，它们和第13章的测试用例执行相同任务。第一部分将文件`junk`的block 3固定，并从中读取两个值：偏移量为392的整数和偏移量为20的字符串。第二部分把该块的偏移量392处的整数自增1。

第三部分则格式化一个新页，将其附加到文件`junk`中，并读取存储在偏移量0中的值；最后，事务提交。

这个示例代码比第13章中对应的示例代码更简洁一些，主要是因为事务对象会负责管理缓冲和日志，其实同样重要的是，事务对象也会管理并发客户端相关的问题。

例如，观察代码我们可以发现，第一部分和第二部分都对 `getInt()` 方法进行了调用，我们需要调用两次吗？第二次调用是否总是返回与第一次调用完全相同的值？在没有并发控制的情况下，答案是“否”。假定此代码在执行第一次调用之后，但又在执行第二次调用之前被中断，并且另一个客户端在此中断期间执行并修改了偏移量392的值。然后，第二个调用实际上将返回与第一个调用不同的东西。（这是第8.2.3节中提到的“不可重复读取”）此方案违反了事务的独立性原则，因此 `Transaction` 对象需要对此负责，并确保这种情况不会发生。换句话说，由于并发控制的存在，我们的确不需要第二次调用 `getInt()`。

## 14.3 恢复管理器

恢复管理器是数据库服务端负责读取并且处理日志信息的部分。它主要有三大功能：写日志记录、回滚事务和数据库系统崩溃后恢复。这一节中我们将研究这些功能的细节。

### 14.3.1 日志记录

为了能够回滚事务，恢复管理器将有关事务活动的信息保存在日志中。特别是，在每次发生一个 `loggable` 的事件时，它会将一条相关的日志记录（`log record`）写入日志文件中。日志记录有四种基本类型：开始日志记录（`start log record`），提交日志记录（`commit log record`），回滚日志记录（`rollback log record`）和更新日志记录（`update log record`）。我们将遵循 SimpleDB 中的设定并假设有两种更新日志记录：一种用于更新整数，另一种用于更新字符串。

三种可以引发日志记录写入的事件为：

- 事务开始时的开始日志记录
- 事务完成时的提交日志记录或回滚日志记录
- 当事务修改某个值时的修改日志记录

另一个潜在的 `loggable` 的事件是在文件末尾附加一个块。然后，如果事务回滚，则可以从文件中释放由 `append()` 方法分配的新块。另一方面，让新块保留在那不做任何修改也不会有任何危害，因为它不会影响其他的事务。为了简单起见，我们将忽略追加日志记录（`log append record`）的可能性。练习 14.48 解决了这个问题。

作为示例，再次考虑上述使用事务的代码片段，该片段在文件 `junk` 的 `block3`，偏移量为 392 处的整数进行了自增 1。假设事务的 ID 为 27，并且该偏移处的旧值为整数 542。下面的序列中包含了从此代码生成的日志记录：

```
<START, 27>
<SETINT, 27, junk, 3, 392, 542, 543>
<COMMIT, 27>
```

请注意，每条日志记录都包含有关该记录的类型描述（START, SETINT, SETSTRING, COMMIT或ROLLBACK）及其事务的ID。更新日志记录包含五项其他内容：修改的文件名、块号、进行修改的位置的偏移量、该偏移量处的旧值，以及该偏移量处的新值。

通常，多个事务将并发地写入日志，因此给定事务的日志记录将被分散在整个日志中。

### 14.3.2 回滚

日志的一个作用是帮助恢复管理器回滚（roll back）一个指定的事务。

恢复管理器通过撤销修改来回滚一个事务。

由于这些修改已列在更新日志记录中，因此扫描日志，查找每个更新记录并将每个修改后值恢复到原始内容是相对简单的事情。算法14-5给出了该算法的描述。

Algorithm 14-5 事务T的回滚算法

1. 把最近的日志记录作为当前日志记录；
2. while(当前日志记录 != 事务T的开始日志记录){
  - if(当前日志记录 == 事务T的更新日志记录)
 在指定位置写入修改前的old value;
  - else
 当前日志记录 = 上一条日志记录;
}
3. 追加一条回滚日志记录到日志文件；

为什么此算法从日志文件末尾开始向前读取，而不是从开始位置向后读取？有两个原因：

- 第一个原因是，日志文件的开头部分一般已经完成的事务的日志记录，而且这些事务一般离现在都很久远，我们正在寻找的日志记录最有可能在日志的末尾，因此从末尾读取效率更高；
- 第二个更重要的原因是确保正确性。假设某个位置的值被修改多次，然后日志文件中将有对应该位置的多个日志记录，每个都有不同的值。要恢复的值应该来自这些日志记录中最早的那个，如果我们以相反的顺序处理日志记录，则刚好对应这种情况发生。

### 14.3.3 恢复

日志的另一个作用是恢复数据库。

系统恢复会在每次数据库系统开启时执行，它的作用是将数据库恢复至一个合理的状态。

合理的状态有两层意思，即：

- 未提交的事务都回滚好了。
- 提交了的事务都持久化到磁盘了。

当数据库系统先启动，再正常关闭后，数据库应该已经处于合理的状态，因为正常的关闭过程是会等到现有事务完成后再flush所有的缓冲区。但是，如果意外导致意外崩溃，则可能有未完成的事务丢失了中间某些执行步骤，由于系统无法完成它们，因此必须undo其作出的修改；也可能有已经提交了的事务，但是其修改尚未刷新到磁盘，这些修改必须redo。

在数据库恢复的过程中，由于未提交事务而带来的修改必须被undo，由于已经提交事务而带来的修改（即页中的内容没有flush到磁盘上去）必须redo。

恢复管理器假设，如果一个事务是提交了的事务，当且仅当该事务存在一条对应的提交日志记录或回滚日志记录。所以，如果一个事务在系统崩溃前已经提交了，但是它所创建的提交日志记录并没有写回到日志文件，那么恢复管理器会将这个事务看作是一个实际上并没有完成的事务。这个场景看起来好像不是那么公平，但是如果日志记录的存在，恢复管理器也真的没法做出其他的一些什么事情来。所有知道的事情全部在日志文件中，因为有关事务的所有其他信息在系统崩溃时被清除了。

实际上，回滚一个已经提交的事务不仅是不公平的，它也违反了ACID原则中的持久化原则，恢复管理器必须确保这个场景不会发生。

恢复管理器必须在完成一个事务提交操作前，将提交日志记录flush到日志文件上去。

回想一下，flush一条日志记录也会flush所有之前的日志记录（当然，所有之前的日志记录指的是在同一个日志页中的之前所有日志记录）。所有当恢复管理器在日志文件中发现一条日志提交记录时，它会知道该事务的之前的更新日志记录（如果存在的话）也肯定在日志文件中。

每条更新日志记录包含了修改前的old value和修改后的new value。其中old value用在undo的时候使用，而new value用在redo的时候使用，算法14-6展示了恢复算法。

**Algorithm 14-6 恢复算法**

```

// undo 步骤 (负责处理那些未提交的事务, 这些事务有可能已经将数据flush)
1. 对于每条日志记录(从日志文件尾部一直往前读){
    if(当前日志记录 instanceof 提交日志记录)
        将该事务添加到提交事务列表中
    if(当前日志记录 instanceof 回滚日志记录)
        将该事务添加到回滚事务列表中
    if( (当前日志记录 instanceof 更新日志记录) &&
        !提交事务列表.contains(当前事务) &&
        !回滚事务列表.contains(当前事务) )
        在指定的位置的数据恢复成old value
}

// redo 步骤 (负责处理那些已经提交了, 但未flush数据到磁盘的事务)
2. 对于每条日志记录(从日志文件头部一直往后读){
    if( (当前日志记录 instanceof 更新日志记录) &&
        提交事务列表.contains(当前事务) )
        在指定的位置的数据恢复成new value
}

```

阶段1 undo了未提交的事务，正如算法中描述的意义，我们必须从后往前读日志文件来确保正确性。从后往前读也意味着，如果某个事务已经提交，那么在读取到该事务的某条update log record之前肯定会先读到该事务的commit log record；所以当算法遇到任意一条update log record记录时，算法会知道是否需要做undo操作。

注意到，阶段1必须读取整个日志文件，例如，某个事务在进入无限循环之前可能已经对数据库进行了更改，因此，除非我们阅读日志的开头，否则将找不到该条更新记录。

阶段2 redo了已经提交的事务，由于恢复管理器不知道哪些缓冲区flush了，哪些没有flush，它会redo所有由提交事务带来的数据更改。

恢复管理器在执行阶段2时是从前往后读取日志文件的，恢复管理器知道哪些update log record需要被redo，因为在步骤1中我们已经得到了提交事务列表。注意，在redo的步骤中，我们也必须从后往前读取日志文件，假如一个提交的事务刚好修改了某个相同的值多次，那么最后一次恢复到的记录才应该对应是最后一次的修改的值。

请注意，恢复算法不需要考虑数据库的当前状态。恢复管理器将旧值或新值写入数据库，而无需查看这些位置的当前值，因为日志文件中的每条记录会准确地告诉恢复管理器，数据库中的内容是什么。此功能有两个后果：

- 恢复是幂等的。
- 恢复可能导致不必要的磁盘写入。

幂等是指执行恢复算法几次与执行一次恢复的效果相同。实际上，即使只运行了一部分恢复算法，此时立即再次重新运行恢复算法，你仍然会得到相同的结果。此属性对于算法的正确性至关重要。例如，假设数据库处于恢复算法中间时，系统崩溃，当数据库系统重新启动时，它将从头开始再次运行恢复算法。如果该算法不是幂等的，则重新运行恢复算法可能会损坏数据库。

由于此算法不会查看数据库的当前内容，因此可能会进行不必要的更改。例如，假定已提交事务所做的修改已写入磁盘；然后在阶段2中redo这些更改，将修改后的值设置为它们已经具有的值。你可以对算法进行修改，以使其不会进行这些不必要的磁盘写入。详情请参阅练习14.44。

#### 14.3.4 Undo-only 和redo-only 恢复

上一节的恢复算法执行undo和redo操作。一些数据库系统选择简化算法，以便仅执行undo操作或仅执行redo操作。也就是说，它执行算法的阶段1或阶段2，但不能同时执行。

##### ***Undo-only*** 恢复

如果恢复管理器确定所有提交的修改都已写入磁盘，则可以省略阶段2。恢复管理器可以通过在将提交记录写入日志之前将缓冲区强制到磁盘来达到这一目的。算法14-7将这种方法表示为一种算法，恢复管理器必须严格按照给定的顺序执行此算法的步骤：

Algorithm 14-7 Undo-only恢复下的事务提交算法

1. 将事务修改的页中的内容flush到磁盘上。
2. 写一条commit log record。
3. 将包含日志记录的日志页flush到日志文件上。

让我们将Undo-only恢复与Undo-redo恢复进行一个比较。Undo-only恢复的速度更快，因为仅需要一次遍历日志文件，而不是两次。日志也较小，因为更新日志记录不再需要包含新的修改值；但另一方面，提交操作会慢很多，因为它必须刷新修改后的缓冲区。如果我们假设系统崩溃很少发生，那么Undo-redo恢复就是最好的选择。事务不仅提交速度更快，而且由于延迟了缓冲区刷新，因此总的磁盘写次数应该更少。

##### ***redo-only*** 恢复

如果恢复管理器确定所有未提交事务对应修改的缓冲区内容尚未写入磁盘，则可以省略阶段1。恢复管理器可以通过使每个事务保持其缓冲区固定，直到事务完成，来实现这一目的。因为固定的缓冲区将不会被选择替换，因此其内容也不会被flush到磁盘上（译者注：忘记了吗？在第13中有

讲解道，我们在替换一个缓冲页前，必须flush一下之前的内容）。但是，当事务回滚时，我们需要“擦除”该事务对缓冲区做出的修改。算法14-8对回滚算法进行了必要的修订：

**Algorithm 14-8 redo-only恢复下的事务回滚算法**

对于每个事务修改的缓冲区：

- a) 将缓冲区标记为未分配的。（在SimpleD中，将缓冲区的块号设置为-1）
- b) 将缓冲区标记为未修改的。
- c) 将缓冲区取消固定。

redo-only恢复比Undo-redo恢复执行要快，因为可以忽略未提交的事务。但是，它要求为每个事务保证有1个缓冲区，用来固定它将修改的每个块，这增加了系统中缓冲区的争用。在大型数据库中，此争用会严重影响所有事务的性能，这使得redo-only恢复是一个冒险的选择。

考虑是否可以将Undo-only和redo-only技术相结合，来创建不需要阶段1或阶段2的恢复算法是很有趣的。详情请参阅练习14.19。

### 14.3.5 预写日志

我们需要更详细地检查一下恢复算法14-6的阶段1。回想一下，此步骤遍历日志，对未完成的事务中的每个更新记录执行undo操作。为了证明这一步骤的正确性，我们做出以下假设：**未完成事务的所有更新记录都将在日志文件中**，未持久化到磁盘的更新日志记录无法被undo，这意味着数据库将变得失效。

由于系统随时可能崩溃，万一日志管理器在将日志页flush到磁盘的过程中，系统崩溃了怎么办？因此，能满足此假设的唯一方法是：在每次完成某个更新数据操作后，立马让日志管理器在写入一个对应的更新日志记录，并flush到磁盘上。但是正如我们在13.2节中看到的一样，这种策略效率不太高，我们需要一个更好的方法。

让我们一起来分析一下各种可能出现问题的情况。假设一个未提交的事务修改了一个页，并且创建了一条更新日志记录，假如系统崩溃了，可能有以下的四种情况发生：

1. 修改的页和日志记录都被写入了磁盘。
2. 只有修改的页被写入了磁盘。
3. 只有日志记录被写入了磁盘。
4. 两个都没被写入磁盘。

我们一个个来分析：

- 如果第1种情况发生了，于是恢复算法将会找到对应的日志记录并且undo，把old value写回到磁盘的数据块中，这不会产生什么问题；

- 如果第2种情况发生了，于是恢复算法将找不到日志记录，所以它不会undo，它也不知道怎么undo，这是一个很严重的问题；
- 假如第3种情况发生了，于是恢复算法将会找到日志记录，然后执行undo操作，只不过这里和第1种情况的区别是，现在数据块中的数据其实根本不是new value，而是old value，但是恢复算法会认为是new value，于是把数据替换成old value，只不过现在就是给一个值再次赋予它本身，并不是什么错误，只是稍微有点浪费时间而已，反正是没错误的；
- 如果第4种情况发生，恢复算法找不到日志记录，但是数据也反正没写到磁盘上，没有发生实际的更改，也没有什么影响，没问题。

所以，第2种情况才是真正的问题所在，这个情况可以避免发生，只要恢复管理器确保一条更新记录总是会在修改相应的缓冲区页前被写入到磁盘上，这个策略被称为 预写日志 (write-ahead log)。注意，一条更新日志可能描述的是一个实际上根本没发生的数据修改情况，但一旦数据库发生了某些修改，我们总是可以在日志中找到相应的更新日志记录，从而找到相应的修改情况。（译者注：注意，这和我们在Algorithm 14-7 Undo-only恢复下的事务提交算法是不一样的）

在预写日志的策略中，只有在相应的更新日志记录全部被写回到文件后，一个修改的缓冲区才能被写回磁盘。

实施预写日志的标准方法是让每个缓冲区保存与其最近修改相对应的日志记录的LSN。在替换修改后的缓冲区页面之前，缓冲区告诉日志管理器将日志刷新到其LSN。LSN较高的日志记录不会受到影响，也不会被刷新。

### 14.3.6 静态检查点

日志包含对数据库的每次修改的历史记录。随着时间的流逝，日志文件的大小可能会变得非常大——在某些情况下可能甚至会大于数据文件。而我们在恢复期间又必须读取整个日志并undo/redo操作，因此，对数据库的更改是一项可怕的事。因此，存仅读取部分日志文件，从而对数据库进行恢复的恢复算法已经被提出。基本思想如下：

恢复算法可以停止搜索日志文件，当：

- 所有的更早的日志记录都是被已经完成的事务所创建的；并且
- 事务对应的缓冲区都被flush到了磁盘上。

第一个要点适用于恢复算法的undo阶段，它确保没有更多未提交的事务要回滚。第二个要点适用于redo阶段，并确保不需要重做所有先前提交的事务。请注意，如果恢复管理器实施的是undo-only恢复算法，则第二个要点永远是正确的。

恢复管理器可以在任何时间执行静态检查点操作，如算法14-9所示。该算法的步骤2确保满足第一个要点，步骤3确保满足第二个要点。

**Algorithm 14-9 执行一次静态检查算法**

1. 停止接受新的事务。
2. 等待存在的事务完成。
3. flush所有修改的缓冲区到磁盘。
4. 追加一条静态检查点日志记录，并且flush到日志文件中。
5. 开始接受新的事务。

静态检查点日志记录就是日志文件中的一个标记记录而已。当恢复算法14-6的阶段1在不断从后往前访问日志记录，直到遇到静态检查点日志记录时，算法就会知道更早的日志记录可以被忽略；于是它开始执行算法的阶段2，并且是从静态检查点日志记录开始，从前往后扫描日志记录。

恢复管理器不需要检查静态检查点日志记录之前的所有日志记录。

进行静态检查点的最好时机是在系统启动期间，并且是在恢复算法完成之后以及新事务开始之前。由于恢复算法刚刚完成了日志的处理，因此恢复管理器将不再需要再次检查那些日志记录。

例如，请考虑下面所示的日志文件，此示例说明了三件事：首先，一旦检查过程开始了，就无法启动任何新事务；第二，最后一个事务完成并且缓冲区被flush后，立即写入检查点记录日志记录；第三，其他事务可能会在写入检查点记录后立即开始。

```

<START, 0>
<SETINT, 0, junk, 33, 8, 542, 543>
<START, 1>
<START, 2>
<COMMIT, 1>
<SETSTRING, 2, junk, 44, 20, hello, ciao>
    //The quiescent checkpoint procedure starts here
<SETSTRING, 0, junk, 33, 12, joe, joseph>
<COMMIT, 0>
    //tx 3 wants to start here, but must wait
<SETINT, 2, junk, 66, 8, 0, 116>
<COMMIT, 2>
<CHECKPOINT>
<START, 3>
<SETINT, 3, junk, 33, 8, 543, 120>
```

### 14.3.7 非静态检查点

### 14.3.8 数据项的粒度

本节的恢复管理算法记录并还原值。即，每次修改值时都会创建一条更新日志记录，该日志记录包含该值的先前和新版本，我们称日志记录中的数据单元为 **恢复数据项 (recovery data item)**。

恢复数据项是被恢复管理器使用，位于一条更新日志记录中的数据部分的单元，这个数据项的大小被称为 **粒度 (granularity)**。

恢复管理器可以选择使用块或文件，而不是使用单个的值作为数据项。例如，假设选择块作为数据项。在这种情况下，每次修改块时都将创建一条更新日志记录，并将该块的先前值和新值存储在日志记录中。

每次记录一个块的优点是，如果使用undo-only恢复，则需要的日志记录会更少。假设一个事务固定一个块，修改几个值，然后取消固定它，我们可以将块的原始内容保存在单个日志记录中，而不必为每个修改后的值编写一个日志记录。当然，缺点就是更新日志记录本身将变得很大，无论块实际更改多少，都将保存该块的全部内容到日志记录中。因此，仅当事务倾向于对每个块进行大量修改时，日志记录整个块才有意义。

现在考虑一下如果将文件用作数据项的意义。一个事务将为它更改的每个文件生成一个更新日志记录，每条日志记录将包含该文件的全部原始内容。要回滚事务，我们只需要将现有文件替换为其原始版本即可。几乎可以肯定，这种方法不如使用值或块作为数据项实用，因为无论更改多少值，每个事务都必须复制整个文件。

尽管文件粒度数据项对于数据库系统不太切合实际，但非数据库应用程序经常使用它们。例如，假设您在编辑文件时系统崩溃，系统重新启动后，某些文字处理程序（例如Word）可以为您显示文件的两个版本：您最近保存的版本以及崩溃时已存在的版本。原因是那些文字处理程序不会将您的修改直接写到原始文件，而是直接写到副本。保存时，修改后的文件将被复制到原始文件中。此策略是基于文件的日志记录的粗略版本。

### 14.3.9 SimpleDB的恢复管理器

SimpleDB的恢复管理器的实现包 `simplesdb.tx.recovery` 中，类 `RecoveryMgr` 的API如下所示：

```
public class RecoveryMgr {
    public RecoveryMgr(int txNum);
    public void commit();
    public void rollback();
    public void recover();
    public int setInt(Buffer buffer,int offset,int newVal);
    public int setString(Buffer buffer,int offset,String ne
}
```

每个事务都会创建其自己的恢复管理器，恢复管理器中含有写入对应该事务某些日志记录的方法。例如，构造函数中会写一条开始日志记录到日志文件中；`commit()` 方法和 `rollback()` 方法也会将相应的提交日志记录或回滚日志记录写到日志文件中去；`setInt()` 和 `setString()` 方法中则会先将指定位置的old value先提取出来，再写一条对应的更新日志记录。`commit()` 方法和 `rollback()` 方法也会执行相应的回滚（或恢复）算法。

SimpleDB的代码实现遵从了下述的原则：

SimpleDB的恢复管理器使用的是Undo-only恢复算法，并且数据项的粒度是一个个的值。

正因为SimpleDB的恢复管理器使用的是Undo-only恢复算法，也就是说不需要执行redo操作，这进一步可以简化我们的日志记录，即现在在`SETINT`和`SETSTRING`日志记录中，不再需要保留new value，只保留一个old value即可。例如，原来的`<SETSTRING, 2, junk, 44, 20, hello, hola>`现在换成`<SETSTRING, 2, junk, 44, 20, hello>`即可。

不再需要保留new value，只保留一个old value即可。不再需要保留new value，只保留一个old value即可。不再需要保留new value，只保留一个old value即可。

重要的事情说3遍！！！因为你接下来将会看到，我们的代码就是这样实现的！

SimpleDB恢复管理器的实现代码主要可以分为3个关注点：

- 实现日志记录的代码
- 迭代日志文件的代码
- 实现回滚和恢复算法的代码

### ***LogRecord 接口***

一条日志记录其实就是通过一系列字节数组来实现的。日志记录的第一个值是一个标识日志记录操作类型的整数，操作类型可能是`CHECKPOINT`，`START`，`COMMIT`，`ROLLBACK`，`SETINT`，`SETSTRING`其中的一种，剩下的值就和具体的操作类型有关了——一条静态检查日志记录没有其他的值，而一条更新日志记录有5个另外的值，其他的日志记录又有其他自己的值。每种日志记录都实现了`LogRecord`接口，代码如下：

```
public interface LogRecord {  
  
    static final int CHECKPOINT = 0, START = 1,  
    COMMIT = 2, ROLLBACK = 3,  
    SETINT = 4, SETSTRING = 5;  
    int writeToLog();  
    int op();  
    int txNumber();  
    void undo();  
}
```

接口定义了方法 `writeToLog()`，该方法将记录追加到日志并返回其 LSN。该接口还定义了三种方法来提取日志记录的组件。方法 `op()` 返回记录的操作类型，方法 `txNumber()` 返回写入日志记录的事务的ID，除了静态检查记录以外，此方法对于其他所有日志记录都有意义（静态检查记录会返回一个虚拟ID值）。`undo()` 方法将还原存储在该记录中的所有更改。仅 `setint` 和 `setstring` 的日志记录将具有非空的`undo`方法，这些记录的方法会将缓冲区固定指定的块，将指定的值写入指定的偏移量，然后取消固定缓冲区。

每一种日志记录都会有相似的代码实现，我们看一下其中一种类型的实现就够了，例如 `SetStringRecord` 类，如下所示：

```

public class SetStringRecord implements LogRecord {
    private int myTxNum;
    private int offset;

    private String val;
    private Block blk;

    public SetStringRecord(int myTxNum, Block blk, int offset) {
        this.myTxNum = myTxNum;
        this.offset = offset;
        this.blk = blk;
        this.val = val;
    }

    /**
     * 根据一条BasicLogRecord来构造一条SetStringRecord。
     * 该构造函数是为了给 恢复/回滚 算法调用
     * <p>
     * 注意，一条更新日志记录的格式为：
     * <p>
     * <SETxxx,txNum,fileName,blkNum,offset,old value,new value>
     *
     * @param blr
     */
    public SetStringRecord(BasicLogRecord blr) {
        myTxNum = blr.nextInt();
        String fileName = blr.nextString();
        int blkNum = blr.nextInt();
        blk = new Block(fileName, blkNum);
        offset = blr.nextInt();
        val = blr.nextString();
    }

    /**
     * 将一条日志记录写入日志文件，返回LSN
     *
     * @return
     */
    @Override
    public int writeToLog() {
        Object[] rec = new Object[]{SETSTRING, myTxNum,
            blk.filename(), offset, val};

        LogMgr logMgr = SimpleDB.logMgr();
        return logMgr.append(rec);
    }

}

```

```

* 返回日志记录的操作符。
* <p>
* CHECKPOINT = 0, START = 1,
* COMMIT = 2, ROLLBACK = 3,
* SETINT = 4, SETSTRING = 5;
*
* @return integer
*/
@Override
public int op() {
    return SETSTRING;
}

@Override
public int txNumber() {
    return myTxNum;
}

@Override
public void undo() {
    BufferMgr bufferMgr = SimpleDB.bufferMgr();
    Buffer buff = bufferMgr.pin(blk);
    buff.setString(offset, val, myTxNum, -1);
    bufferMgr.unpin(buff);
}

public String toString() {
    return "<SETSTRING " + myTxNum + " " + blk + " " +
           + " " + val + ">";
}

}

```

该类有2个构造函数，第一个有日志记录属性的一些参数，并且简单地存储了一下它们而已，这个构造函数在每次将日志记录写回到磁盘前会被正确调用；另外一个构造函数是被回滚和恢复算法调用的方法，传入的是一个 `BasicLogRecord` 类的对象，然后我们可以通过这个对象来不断提取出相应的int或string类型的值。

`undo()` 方法则会将一个缓冲区与一个修改过的块固定起来，恢复存储的值，然后取消固定。`undo()` 方法实现中的一个小故障是：`setString()` 方法期待获得与当前修改相对应的日志记录的LSN。当然，在这种情况下，再次记录恢复的值是没意义的，所有在这里我们传入了一个dummy的LSN，即-1。

### 迭代日志文件

`LogRecordIterator` 类允许客户端从后往前遍历日志文件，每次迭代一条日志记录，代码如下：

```
public class LogRecordIterator implements Iterator<LogRecord> {
    // 先获得一个BasicLogRecord迭代器
    // 此迭代器迭代得到结果是一条条raw的日志记录
    private Iterator<BasicLogRecord> iter = SimpleDB.logManager()
        .getLogRecords();

    @Override
    public boolean hasNext() {
        return iter.hasNext();
    }

    @Override
    public LogRecord next() {
        BasicLogRecord blr = iter.next();
        int op = blr.nextInt();
        switch (op) {
            case CHECKPOINT:
                return new CheckpointRecord(blr);
            case START:
                return new StartRecord(blr);
            case COMMIT:
                return new CommitRecord(blr);
            case ROLLBACK:
                return new RollBackRecord(blr);
            case SETINT:
                return new SetIntRecord(blr);
            case SETSTRING:
                return new SetStringRecord(blr);
            default:
                return null;
        }
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

上述代码比较简单直接，构造函数调用了日志管理器的获得 `BasicLogIterator` 迭代器的方法，`BasicLogIterator` 迭代器迭代得到结果是一条条raw的日志记录。所以在 `LogRecordIterator` 迭代器

里的 `next()` 方法中，需要解析每条记录具体是什么类型的。首先，我们先读取每条日志记录最开头的4个字节，也就是日志记录的类型操作符，然后再根据具体的类型来构造不同类型的日志记录对象。

下面的代码片段演示了如何使用 `LogRecordIterator` 迭代器来逆序迭代访问日志文件：

```
Iterator<LogRecord> iter = new LogRecordIterator();
while (iter.hasNext()) {
    LogRecord rec = iter.next();
    System.out.println(rec.toString());
}
```

### 回滚和恢复

`RecoveryMgr` 类实现了恢复管理器的API，它的具体实现代码如下所示。代码中的每个方法实现的都是前面小节中提到的undo-only恢复算法。特别是，`commit()` 方法和 `rollback()` 方法会在写入日志文件之前，将该事务对应的缓冲区flush到磁盘上去；`doRollback()` 方法和 `doRecover()` 方法会从后往前遍历一遍日志文件。

```

public class RecoveryMgr {

    private int txNum;

    /**
     * 为指定的事务创建一个恢复管理器
     *
     * @param txNum 指定事务的ID
     */
    public RecoveryMgr(int txNum) {
        this.txNum = txNum;
        new StartRecord(txNum).writeToLog();
    }

    /**
     * 写入一条提交日志记录，并将日志记录flush到日志文件
     * <p>
     * 注意，在SimpleDB中，采用的是undo-only恢复算法，
     * 因此在提交日志记录被写入前，必须将，当前事务对应的，并且是修改
     * flush到磁盘上。
     * <p>
     * 详情参阅第14章中的算法14-7
     */
    public void commit() {
        SimpleDB.bufferMgr().flushAll(txNum);
        int lsn = new CommitRecord(txNum).writeToLog();
        SimpleDB.logMgr().flush(lsn);
    }

    public void rollback() {
        SimpleDB.bufferMgr().flushAll(txNum);
        doRollback(); // 把修改后的值，再改回来
        int lsn = new RollBackRecord(txNum).writeToLog();
        SimpleDB.logMgr().flush(lsn);
    }

    public void recover() {
        SimpleDB.bufferMgr().flushAll(txNum);
        doRecover(); // 把修改后的值，再改回来
        int lsn = new CheckpointRecord().writeToLog();
        SimpleDB.logMgr().flush(lsn);
    }

    /**
     * 写一条SetInt日志记录到日志文件，并返回其LSN
     * <p>
     * 对临时文件作的修改将不被保存，此时返回的LSN是一个负值，没有意
     */
}

```

```

    * @param buffer
    * @param offset
    * @param newVal
    * @return
    */
    public int setInt(Buffer buffer, int offset, int newVal) {
        int oldVal = buffer.getInt(offset);
        Block blk = buffer.block();

        if (isTemporaryBlock(blk))
            return -1;
        else
            return new SetIntRecord(txNum, blk, offset, oldVal);
    }

    /**
     * 写一条SetString日志记录到日志文件，并返回其LSN
     * <p>
     * 对临时文件作的修改将不被保存，此时返回的LSN是一个负值，没有意义。
     *
     * @param buffer
     * @param offset
     * @param newStr
     * @return
     */
    public int setString(Buffer buffer, int offset, String newStr) {
        String oldVal = buffer.getString(offset);
        Block blk = buffer.block();

        if (isTemporaryBlock(blk))
            return -1;
        else
            return new SetStringRecord(txNum, blk, offset, newStr);
    }

    /**
     * 回滚事务。
     * <p>
     * 该方法会遍历日志记录，调用遍历到的每条日志记录的undo()方法，直到该事务的START日志记录为止。
     */
    private void doRollback() {
        Iterator<LogRecord> iter = new LogRecordIterator();
        while (iter.hasNext()) {
            LogRecord rec = iter.next();
            if (rec.txNumber() == this.txNum) {

```

```

        if (rec.op() == LogRecord.START)
            return;
        else
            // 其实只有SetIntRecord和SetStringRecord
            // 的undo()方法才有具体的实现，其他日志记录类
            // 的undo()方法都是空方法。
            rec.undo();
    }

}

/***
 * 执行一次数据库恢复操作。
 * <p>
 * 该方法会遍历日志记录，无论何时它发现一个未完成事务的日志记录，
 * 它都会调用该日志记录的undo()方法。
 * <p>
 * 当遇到一个CHECKPOINT日志记录或日志文件尾时（从后往前读，所以
 * 恢复算法停止。
 */
private void doRecover() {
    // 已经提交事务的ID集合
    Collection<Integer> committedTxs = new ArrayList<>();
    Iterator<LogRecord> iter = new LogRecordIterator();
    while (iter.hasNext()) {
        LogRecord rec = iter.next();
        if (rec.op() == LogRecord.CHECKPOINT)
            return;
        // TODO: 这个地方和作者提供的原始代码不太一样
        // TODO: 译者认为，已经提交的事务应该包括已经commit和
        // TODO: 当然，按照作者提供的原始代码在执行效果上和当前
        // TODO: 因为， rollback日志记录的undo()方法为空！
        if (rec.op() == LogRecord.COMMIT || rec.op()==
            committedTxs.add(rec.txNumber());
        else if (!committedTxs.contains(rec.txNumber()))
            // 其实只有SetIntRecord和SetStringRecord
            // 的undo()方法才有具体的实现，其他日志记录类
            // 的undo()方法都是空方法。
            rec.undo();
    }
}

private boolean isTemporaryBlock(Block blk) {
    return blk.filename().startsWith("temp");
}
}

```

`doRollback()` 方法会读取指定事务的日志记录直到遇到START日志记录，每次它找到一个该事务的日志记录，都会调用该日志记录的 `undo()` 方法，当遇到该事务的START日志记录时，就会停止迭代。

(译者注：其实只有`SetIntRecord`和`SetStringRecord`的`undo()`方法才有具体的实现，其他日志记录类的`undo()`方法都是空方法，所以实际上，只有`SetIntRecord`和`SetStringRecord`类型的日志记录才会执行对应的回滚操作，这也十分合乎逻辑，之所以给其他日志记录也加上 `undo()` 方法是为了方便用一个统一的接口来管理代码）。

`doRecover()` 方法有差不多的实现，它读取整个日志文件的日志记录，直到遇到Checkpoint日志记录或者日志文件尾部（实际上是日志文件的头部），和 `doRollback()` 方法中一样，会undo那些没提交事务而造成的更新，区别就是，在处理的过程中，会维护一个已经提交的事务的ID集合。这个方法和算法14-6中的步骤有一些不同，区别在于代码实现中，对应那些已经回滚的事务，也会执行undo操作，虽然说这样的实现在最终的效果来看不会产生什么影响（具体可以参考代码中的注释），只是有点效率低。（译者注：上述实现是作者提供原始代码的修改版本，即已经提交的事务应该包括已经commit和rollback的事务，即已经rollback的事务不再进行回滚操作）

该代码中的另一个详细信息是 `setInt()` 和 `setString()` 方法不会 undo临时文件中的修改（即以“temp”名字开头的文件）。临时文件是在 物化查询处理 (materialized query processing) 过程中创建的（请参见第22章），它们之后会被删除，因此恢复其中的更改没有任何意义。

## 14.4 并发管理器

声明：14.4小节翻译的并不很好，甚至有时会让你觉得翻译的很僵硬，让人没有头绪。是的，请毫不犹豫地怀疑我的水平并对照英文版查看，如果你有更好的翻译见解，请email联系。

并发管理器是数据库服务器中负责正确执行并发事务的部分。在本节中，我们将研究执行“正确”意味着什么，并研究用于确保正确性的并发相关的算法。

### 14.4.1 顺序化调度(Serializable schedules)

译者注：我个人感觉把`schedules`翻译成调度不是特别的合适，无奈又感觉很难用一个中文词语来描述，这里的调度指的就是一系列按照一定顺序组织起来的操作。

我们把一个事务的 访问历史(history) 定义为一系列因数据库文件访问而引发的方法调用——即，`getXXX/setXXX()` 方法。例如，回顾一下 14.2小节中的示例代码，并且在这里只讨论代码前两段的事务，该事务对

应的访问历史如下，有点乏味：

```
int n = tx.getInt(new Block("junk", 3), 392);
String str = tx.getString(new Block("junk", 3), 20);
tx.setInt(new Block("junk", 3), 392, n+1);
```

实际上，表达一个事务的访问历史的更简单的方式是，只对受影响的块进行描述：

```
tx: R(junk,3); R(junk,3); W(junk,3);
```

也就是说，该事务的历史访问表明，该事务从文件 `junk` 的第3块中读取了2次，并且写入了该块1次。

一个事务的历史访问描述了该事务对应的数据库操作序列。

在这里，我们说的术语“数据库操作”是一种有意的含糊表达。上面的示例首先将数据库操作视为块中某个值的修改（int或string类型），然后视为对磁盘块的读/写。其他粒度也是可能的，我们将在14.4.8小节中讨论。在此之前，我们将假定数据库操作是对磁盘块的读取或写入。

当多个事务并发执行时，数据库系统将交错执行其线程，即周期性地中断一个线程，然后恢复另一个线程（在SimpleDB中，Java运行时环境会为我们自动地处理交错执行线程）。因此，并发管理器的实际数据库操作将会是一个，不可预测的，交替执行着的历史访问，我们称这种交替执行序列为一个 `调度 (schedule)`

一个 `调度 (schedule)` 是指一个事务对应的历史访问在数据库系统底层实际交错执行的序列。

并发控制的目的就是为了确保只有正确的调度被执行。但是，怎样的顺序才是“正确的”呢？好，我们考虑一个最简单的调度——所有的事务一个一个顺序地执行。这个调度中的操作不会交错地执行；也就是说，该调度就是简单地将每个事务的历史访问一个接一个地顺序执行，我们称这种调度为一个 `顺序调度 (serial schedule)`。

一个 `顺序调度 (serial schedule)` 是各事务访问历史不会交错执行的调度。

并发控制正是基于 `顺序调度的结果是正确的` 假设的，实际上，`顺序调度` 也没有并发。

使用 `顺序调度` 的结果来定义正确的结果，会引发一个有趣的事情，那就是：相同事务的不同的顺序调度可能会产生不同的结果。例如，考虑下述的两个事务  $T_1$  和  $T_2$ ，它们有如下相同的历史访问：

```
T1: W(b1); W(b2);
T2: W(b1); W(b2);
```

虽然这两个事务有相同的历史访问（即，它们都先对块 $b_1$ 进行一次写操作，然后再对块 $b_2$ 进行一次写操作），但是从事务的概念上讲，它们又不是相等的——例如，事务1可能在每个块的块首部写入一个值X，然而事务2在首部写入Y，如果事务1在事务2之前执行，那么块中最后的值将会保留事务2写入的值，也就是Y；如果反过来，事务2先执行，那么块中最后的值将会保留事务2写入的值，也就是X。

在这个例子中，事务1和事务2对块进行的修改不一样。并且，对于数据库系统的眼中看来，所有的事务都是平等的，没有说哪个最终结果比另外一个更正确。因此，我们必须承认，上述例子中的每个顺序调度都是正确的，也就是说，可能有好几个正确的结果。

给定一个 非顺序调度 (non-serial schedule) ，如果它的执行结果和某个顺序调度的结果相同，我们称它是可顺序化的。因为顺序调度的结果是正确的，因此，可顺序化调度的结果也是正确的。

可顺序化的调度是一种和顺序调度产生相同结果的一个调度。

注意，我故意在这里将Serializable翻译成顺序化而不是序列化，原因之一是在这里本身强调的是顺序执行的意思，原因之二是在Java语言中也有一个Serializable标记接口，在Java中应翻译成序列化，这里是为了与之区别开。

同样对于上述两个事务 $T_1$ 和 $T_2$ ，但是一个非顺序化调度：

```
W1(b1); W2(b1); W1(b2); W2(b2);
```

在这里，我们使用 $W_1(b_1)$ 来标识事务 $T_1$ 对块 $b_1$ 的写操作，其他的也类似。在该调度中，先执行事务1的前半部分，然后执行事务2的前半部分，再执行事务1的后半部分，最后执行事务2的后半部分。我们说这个调度是可顺序化的因为它和先执行事务1直到事务1完成，再执行事务2直到事务2完成的最后结果是一样的。另一方法，考虑下述另一个调度的情况：

```
W1(b1); W2(b1); W2(b2); W1(b2);
```

在该调度中，先执行事务1的前半部分，然后执行事务2的前半部分，再执行事务2的后半部分，最后执行事务1的后半部分。该调度的结果是，块 $b_1$ 最后会保留事务2写入的值，而块 $b_2$ 最后会保留事务1写入的值，这和上述任意一个顺序调度产生的结果都不一样，于是我们称，这个调度是 不可顺序化的 (non-serializable) 。

回顾一下ACID原则中的独立性原则，它的意思是说，每个事务执行时，应该就好像整个系统中只有一个事务在执行，一个非顺序化调度是没有这个属性的。因此，我们也更加必须承认非顺序化执行的调度是不正确的，这个分析可表达为如下的原则：

一个调度是正确的，当且仅当它是可顺序化的。

### 14.4.2 锁表

数据库系统负责确保所有的调度都是可顺序化的。一个常用的技术是使用 锁 (locking) 来推迟事务的执行。在14.4.3小节我们将会看到锁机制是如何被使用，从而确保顺序化的，在本节中，我们将简单介绍一下基本的锁机制是怎样工作的。

每个块有2种锁—— 共享锁`shared lock(slock)` 和 互斥锁`exclusive lock(xlock)`。如果一个事务拥有某个块的互斥锁，那么不允许其他的任何事务来对该块请求任何锁（共享锁和互斥锁都不能被满足）；如果一个事务拥有某个块的共享锁，那么允许其他事务来对该块请求共享锁，而不允许其他事务来对该块请求互斥锁。

锁表是系统中负责向事务授权锁的部分，在SimpleDB中，锁表是通过类 `LockTable` 来实现的，它的API如下所示：

```
public class LockTable {
    public void sLock(Block blk);
    public void xLock(Block blk);
    public void unLock(Block blk);
}
```

方法 `sLock()` 请求一个指定块的共享锁，如果该块的一个互斥锁已经分配给了另外一个事务，那么这个方法必须等待，直到互斥锁被释放了。方法 `xLock()` 请求一个指定块的互斥锁，如果该块的任意一个锁已经被其他事务持有，那么该方法会一直等待，直到两种锁都被释放。方法 `unlock()` 则会释放一个已经持有的锁。

下面展示了一个示例，演示了锁请求之间的交互。3个线程对于块 $b_1$ 和 $b_2$ 的请求如下：

```
Thread A: sLock(b1); sLock(b2); unlock(b1); unlock(b2);
Thread B: xLock(b2); sLock(b1); unlock(b1); unlock(b2);
Thread C: xLock(b1); sLock(b2); unlock(b1); unlock(b2);
```

假设线程之间的实际运行顺序是，每个线程执行一条语句后，都被打断，然后继续下一个线程，于是：

1. 线程A获得块 $b_1$ 的共享锁。
2. 线程B获得块 $b_2$ 的互斥锁。
3. 线程C请求块 $b_1$ 的互斥锁,但不能满足, 因为已经有线程A持有了块 $b_1$ 的共享锁, 因此线程C必须等待。
4. 线程A请求块 $b_2$ 的共享锁,但不能满足, 因为已经有线程B持有了块 $b_2$ 的互斥锁, 因此线程A必须等待。
5. 线程B请求块 $b_1$ 的共享锁, 可以被满足, 因为没有其他人当前持有块 $b_1$ 的互斥锁。 (线程C在等待块 $b_1$ 的互斥锁对此没有影响)
6. 线程B释放块 $b_1$ 的锁, 但这个操作对等待的进程没什么影响, 因为释放块 $b_1$ 的共享锁并不会让整个锁处于空闲状态, 因为线程A也持有块 $b_1$ 的共享锁。
7. 线程B释放块 $b_2$ 的锁, 通知其他等待块 $b_2$ 相关锁的线程。
8. 线程A因线程B发来的通知, 从而继续执行, 获得块 $b_2$ 的共享锁。
9. 线程A释放块 $b_1$ 的锁, 通知其他等待块 $b_1$ 相关锁的线程。
10. 线程C最终获得块 $b_1$ 的互斥锁。
11. 线程A和线程C随便以某种交替执行的方式执行下去, 直到线程都结束。

### 14.4.3 锁协议

现在我们来考虑一下如何用锁机制来保证所有的调度都是可顺序化的。考虑下述拥有如下历史访问的两个事务：

```
T1: R(b1); W(b2);
T2: W(b1); W(b2);
```

是什么导致他们的顺序调度有不一样的结果？事务1和事务2都对同一个块执行了写操作，那就意味着这两个事务对该块的操作顺序是有影响的——即谁最后运行，谁就是这个块的“赢家”，于是，我们称操作

$\{W_1(b_1), W_2(b_2)\}$ 有冲突。通常来讲，如果两个操作的顺序会产生不一样的结果，那么这两个操作就是有冲突的。如果两个事务有冲突的操作，那么他们顺序化调度就可能有不同的结果（但是都是正确的结果）。

上述的两个冲突是 **写写冲突(write-write conflict)**。第二种冲突是 **读写冲突(read-write conflict)**，例如，操作 $\{R_1(b_1), W_2(b_1)\}$ 就有冲突——如果事务1先执行，那么事务1读到的是块 $b_1$ 以前的数据，然后事务2再对块 $b_1$ 进行写操作；如果事务2先执行，那么事务2先对块 $b_1$ 进行写操作，然后事务1再执行读操作，所以读到的实际上是事务2作出的修改。

注意，是永远不可能发生 读读冲突(read-read conflict) 的，而且，值得肯定的是，对不同块的操作也是不会发生冲突的，这也是显然的。

之所以我们关心这些冲突的原因是，它们会影响一个调度的可顺序化性。

有冲突的操作在非顺序化调度下的执行顺序决定了等效的顺序化调度。The order in which conflicting operations are executed in a non-serial schedule determines the equivalent serial schedule.

在上述的例子中，如果  $W_2(b_1)$  先于  $R_1(b_1)$  执行，那么任何等效的顺序调度都必须是事务2先于事务1执行的。这意味着，如果你考虑事务1中所有与事务2有冲突的操作，要么它们必须先于事务2中存在冲突的所有操作前执行，要么它们必须全后于事务2中存在冲突的所有操作前执行，没有冲突的操作可以按任意顺序执行。

锁可以用来避免读读冲突和读写冲突，我们要求所有的事务都按照下述锁协议来使用锁：

#### 14-20 锁协议

1. 在读某个块前，请求获得该块的共享锁。
2. 在写某个块前，请求获得该块的互斥锁。
3. 在提交或回滚后释放所有获得的锁。

从该锁协议中，我们可以推断出两个重要的事实：

- 首先，如果一个事务获得了某个块的共享锁，那么将没有其他活动的事务写入该块（否则，一些事务在该块上仍将持有互斥锁）。
- 其次，如果一个事务获得了某个块的互斥锁，那么任何其他活动的事务都不会以任何方式访问该块（否则，一些事务仍将对该块持有锁）。

这些事实意味着由事务执行的操作将永远不会与另一个活动事务的先前操作冲突。换句话说，我们说明了以下事实：

如果所有的事务都遵循锁协议，那么：

- 调度总是都是可顺序化的（因此也是正确的）。
- 等效的顺序调度由事务提交的顺序而确定。

锁协议强制事务持有锁，直到事务完成，这样一来，会一定程度上限制系统的并发性。如果一个事务可以在不再需要某个块时释放它的锁，那就太好了，这样其他事务也就不必等待那么久了。但是，事务在完成前释放锁，会引发两个严重的问题：

如果事务提早释放锁，可能会发生两个问题：

- 它可能不再是可顺序化的了。
- 其他事务可以读到未提交的修改。

### 可顺序化问题

一旦一个事务释放了某个块的锁，那么它就不可能在不影响整个调度可顺序化性的前提下，持有其他块的锁。为了了解原因，请考虑这样一个例子：事务 $T_1$ 在获得块y的锁前释放了块x，即：

```
T1: ... R(x); UL(x); SL(y); R(y); ...
```

假设事务1在释放块x的锁和请求块y的共享锁之间被打断，也就是上述  $UL(x)$  操作之后， $SL(y)$  操作之前。在这个时候，事务1是易损的，因为块x和块y都是没被上锁的状态，假设这时另一个事务 $T_2$ 到来，对块x和块y进行读写，然后提交，并随后释放块。我们于是就来到了这样一个情景：事务1必须先于事务2顺序执行，因为事务1读取的是块x的上一个版本（即事务2写操作前的值）；另一方面，事务1又必须后于事务2顺序执行，因为事务1读取到块y的内容是事务2修改后的值。因此，生成的调度是不可顺序化的。

反过来同样正确——也就是说，如果一个事务在释放任何一个锁前，持有了所有的锁，那么产生的调度确保是可顺序化的（请参阅练习14.27）。这个变种被称为 **二段锁** (*two-phase locking*)，它个名字的由来是因为在这个锁协议下，一个事务有两个阶段——不断获得锁的阶段和释放锁的阶段。

二段锁只允许事务在获得了所有锁之后，才能开始释放锁。（译者注：在有的资料中也可能这样表述二段锁：一旦一个事务释放了它持有的任意锁，那么该事务就再也不能获得其他锁了。）

虽然理论上来说，二段锁是一个更通用的锁协议，但是一个数据库系统很难轻易利用它。通常，当事务完成它的最后一个块访问后（也就是锁即将准备被释放的时候），无论如何它都会准备提交，因此，完全通用的二段锁协议很少在实践中有效。

### 读到未提交数据

过早释放锁可能带来的另一个问题是事务可能会读到了未提交的数据，考虑下面这样一部分调度：

```
... W1(b); UL1(b); SL2(b); R2(b); ...
```

在这个调度中，事务1对块b进行写操作，然后再释放块；事务2随后请求块b的共享锁，并读取了块b中的内容。如果事务1最终提交了，那没什么问题；但是假如事务1执行了一次回滚操作，那么事务2也必须回滚，因为事务2的执行是基于一个不存在的修改，并且如果事务2回滚了，那么类似也可能造成其他的事务3、事务4...也回滚。这个现象被称为 **级联回滚** (*cascading rollback*)。

当数据库系统允许一个事务读取到未提交的数据时，它将提高系统的并发性，但是必须承受写数据的事务不会提交的风险。当然，回滚是一种不是很频繁发生的时间，于是级联回滚也对应发生的不是那么频繁。问题就在于，数据库系统是否要承担可能不必要的回滚事务带来的风险。大多数商业数据库系统都不会冒这个险，因此总是等到事务完成后才释放其互斥锁。

#### 14.4.4 死锁

虽然锁协议确保了调度将是可顺序化的，但是没有保证所有的事务都会提交。特别是，事务之间很有可能造成死锁。

我们已经在12.4.3小节中学习了死锁的例子，其中两个客户端线程相互等待另一个线程释放缓冲区，一个类似的问题也会在锁中出现。当有一圈环形的事务存在，并且第一个事务在等待第二个事务持有的锁，第二个事务在等待第三个事务持有的锁，按照这样的规律一直传递下去，直到最后一个事务等待第一个事务持有的锁时，死锁问题就出现了。在这样的场景下，没有一个等待着的事务可以继续，并且所有事务都会无限等待。为了简单起见，考虑下面这样一个历史访问，其中两个事务对相同的块进行写操作，但是顺序不一样：

```
T1: W(b1); W(b2);
T2: W(b2); W(b1);
```

假设事务1先获得了块 $b_1$ 的锁，于是现在事务1和事务2在块 $b_2$ 的锁上存在着竞争，如果事务1先得到锁，那么事务2会等待，事务1会最终指向完毕并且提交，然后释放所有的锁，随后事务2执行直到完毕，这种情况没什么问题；但是假如事务2先得到块 $b_2$ 的锁，那么死锁就会发生——事务1将等待事务2释放块 $b_2$ 的锁，而事务2又在等待事务1释放块 $b_1$ 的锁，没有人可以继续，它们会一直等下去。

通过维护一个 `wait-for` 图，并发管理器可以检测到死锁的发生与否。在这个图中，每个事务对应图中一个结点；如果事务1在等待事务2持有的锁，那么结点1就存在一个到结点2的边，边上的值就是等待的对象。每次请求一个锁或释放一个锁时，该图都会更新。例如，上述死锁场景的 `wait-for` 图如下图所示：

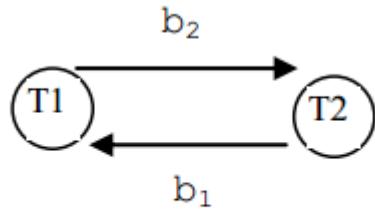


Figure 14-21: A waits-for graph

很容易证明，当且仅当图中存在环时，死锁就会发生，详情请参阅练习14.28。当事务管理器检测到死锁发生时，它会通过汇总回滚环中的任何事务来打破死锁现象。一个合理的策略是回滚那些“造成”环发生的锁请求对应的事务，尽管说也有很多其他可能的策略，详情请参阅练习14.29。

如果我们同时考虑等待缓冲区的线程和等待锁的线程，那么检测死锁将变得更复杂。例如，假设缓冲池只包含两个缓冲区，并考虑下述的场景：

```

T1: xlock(b1); pin(b4);
T2: pin(b2); pin(b3); xlock(b1);
  
```

假设事务1在获得块 $b_1$ 的锁后被打断了，然后事务2固定了块 $b_2$ 和 $b_3$ ，随后事务2会等待事务1释放块 $b_1$ 的锁，而事务1又在等待事务2取消固定某个块从而释放出一个缓冲区来，这样一来，死锁又发生了，但是 wait-for 图却是没有环的。

为了检测到上述的死锁场景，锁管理器必须不仅维持一个 wait-for 图，而且需要知道哪些事务在等待什么缓冲区。将这种额外的考虑因素纳入死锁检测算法中是相当困难的。有深入兴趣的读者可以尝试练习14.38。

使用 wait-for 图检测死锁的问题在于，该图在某种程度上难以维护，并且在图中检测环的过程非常耗时。因此，已经开发出了更简单的策略来近似死锁检测。这些策略是保守的，从某种意义上说，它们始终会检测到死锁，但也可能会将非死锁情况视为死锁。我们将考虑两种可能的策略，练习14.33中考虑了另一个。

第一个近似策略被称为 wait-die 策略，如算法14-22所示：

Algorithm 14-22 wait-die死锁检测策略

```

假设事务T1请求一个被事务T2持有的锁。
if(T1 is older than T2)
    T1等待该锁;
else if(T1 is newer than T2)
    T1回滚; (即die)
  
```

该策略确保死锁都会被检测出，因为 `wait-for` 图将只包含从 `older` 事务到 `newer` 事务的边。但是这个策略会将每个潜在的死锁都看做是回滚的原因，例如，假设事务  $T_1$  比事务  $T_2$  老，并且事务  $T_2$  请求一个已经被  $T_1$  持有的锁，虽然说这时不一定立马会产生死锁，但是有存在死锁的可能，因为在不久的将来某个点，事务  $T_1$  仍会请求一个被事务  $T_2$  持有的锁，因此该策略会抢先回滚事务  $T_2$ 。（也就是抢先回滚那个更新的事务）

第二个近似策略是使用时间限制来检测一个可能的死锁。如果一个事务已经等了一个预先设定好的时间阈值，则事务管理器可以假设它是死锁的，并且会将之回滚，如算法 14-23 所示：

**Algorithm 14-23 基于时间限制的死锁检测策略**

假设事务  $T_1$  请求一个被事务  $T_2$  持有的锁。

1. 事务  $T_1$  等待该锁。
2. 如果事务  $T_1$  待在等待队列上太久，则回滚事务  $T_1$ 。

不管死锁检测策略如何，并发管理器都必须通过回滚活动事务来打破死锁。希望通过释放该事务的锁，剩余的事务将能够完成。事务回滚后，并发管理器将抛出一个异常。在 SimpleDB 中，此异常称为 `LockAbortException`。与第 13 章的 `BufferAbortException` 一样，该异常会由终止的事务的 JDBC 客户端捕获，然后由客户端决定如何处理它。例如，客户端可以选择简单地退出，或者可以尝试再次运行事务。

#### 14.4.5 文件级冲突和幻象

到目前为止，我们已经考虑了由于读取和写入块而引起的冲突。另一类冲突涉及方法 `size()` 和 `append()`，它们分别会读取和写入文件的结束标记符。显然，这两种方法相互冲突：假设事务  $T_1$  首先调用 `append()`，然后事务  $T_2$  调用 `size()` 方法，那么  $T_1$  必须在  $T_2$  之前顺序执行。

这种冲突的后果之一被称为 `幻象问题(phantom problem)`。假设  $T_2$  必须多次读取文件的全部内容，并且它在每次迭代之前调用 `size()` 方法来确定要读取的块数。此外，假设在  $T_2$  第一次读取文件之后，事务  $T_1$  将一些附加值追加到文件中并提交。下次浏览文件时， $T_2$  将看到这些附加值，这违反了 ACID 原则中的独立性原则。这些附加值称为 `幻象(phantom)`，因为它们在  $T_2$  中已经神秘地出现了。

并发管理器如何避免这种冲突？我们看到的避免读写冲突的方法是让 $T_2$ 在它读取的每个块上获得一个共享锁。这样， $T_1$ 将无法向这些块写入新值。但是，这种方法在这里行不通，因为它将需要 $T_2$ 获取一个尚不存在的块的共享锁！（因为这个块是由事务 $T_1$ 新创建出来的块）

解决方案是允许事务给文件的结束标记符上锁（end-of-file marker）。

如果并发管理器允许事务给结束标记符上锁，那么幻象问题可以被避免。

特别地，一个事务需要对end-of-file marker获得互斥锁后才能调用块的 `append()` 方法，并且它需要对end-of-file marker获得共享锁才能调用 `size()` 方法。在我们上面的场景中，如果 $T_1$ 首先调用 `append()`，则 $T_2$ 将无法确定文件大小，直到 $T_1$ 完成为止；相反，如果 $T_2$ 已确定文件大小，则将阻止 $T_1$ 追加，直到提交 $T_2$ 。无论哪种情况，都不会发生幻像。

#### 14.4.6 多版本锁定

大多数数据库应用程序中的事务主要是只读的。只读事务在数据库系统中可以很好地共存，因为它们共享锁，而不必彼此等待。但是，它们与更新事务共存性不好。假设一个更新事务正在写入一个块，然后，所有想要读取该块的只读事务都必须等待，不仅要等到写入该块，还要等到更新事务完成为止；相反，如果更新事务要写入一个块，则需要等待，直到读取该块的所有只读事务都已完成。

换句话说，当只读事务和更新事务冲突时，无论哪个事务首先获得锁定，都会发生大量等待。鉴于这种情况很普遍，研究人员已经制定了可以减少这种等待的策略。一种这样的策略称为 多版本锁定(multiversion locking)。

##### 多版本锁定的原则

正如名字所展示的那样，多版本锁定通过存储每个块的多个版本来完成，它的基本思想如下：

- 块的每个版本都会附有写入该块的事务提交时间的时间戳。
- 当只读事务请求来自块的值时，并发管理器将使用离只读事务开始时最近的那次提交所对应块的版本。

换句话说，只读事务将看到已提交数据的快照，就像在事务开始时所看到的那样。注意术语“已提交的数据”，事务将看到在本事务开始之前，那些提交事务写入的数据，而看不到以后的提交事务所写入的数据。

下面将展示一个多版本锁定的一个例子。

考虑4个事务拥有如下的历史访问：

```

T1: W(b1); W(b2);
T2: W(b1); W(b2);
T3: R(b1); R(b2);
T4: W(b2);

```

假设这些事务都按照下述的调度执行：

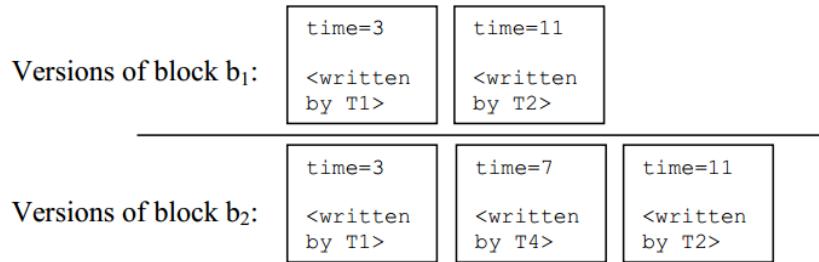
```

W1(b1); W1(b2); C1; W2(b1); R3(b1); W4(b2); C4; R3(b2); C3;

```

在这个调度中，我们假设事务都是从它的第一个操作开始，并且使用 $C_i$ 来表示事务 $T_i$ 的提交。更新事务 $T_1, T_2, T_4$ 遵循锁机制，你可以自己验证一下。事务 $T_3$ 是只读事务，且不遵循锁机制。

并发管理器存有的每个更新事务对应块的版本。因此，将有块 $b_1$ 的两个版本和块 $b_2$ 的三个版本，如下图所示：



每个版本上的时间戳是事务提交的时间，而不是写操作发生的时间。我们假设每个操作都耗费一个时间单位，因此事务 $T_1$ 的提交时间为3，事务 $T_4$ 的提交时间为7，事务 $T_3$ 的提交时间为9，事务 $T_2$ 的提交时间为11。

现在考虑一下只读事务 $T_3$ ，它的开始时间为5，意味着它应该看到的是这个时候对应已经提交了的值——即，只能看到事务 $T_1$ 作出的修改，而事务 $T_2$ 和 $T_4$ 作出的修改，通通看不到。因此，它只会看到块 $b_1$ 和 $b_2$ 在时间戳为3的值。请注意，即使时间戳为7时块 $b_2$ 的版本已经被记录下了，但事务 $T_3$ 在读取块 $b_2$ 的内容，仍将看不到时间戳为7时对应版本的内容，而看到的是时间戳为3时对应版本的内容，因为我们只以事务 $T_3$ 的开始时间为依据（也就是5，即读取块 $b_1$ 内容对应的时间），离时间戳5最近的一个版本是时间戳3，所以事务 $T_3$ 只能看到时间戳3时对应的内容。

多版本锁定的优点在于，只读事务不需要获取锁定，因此不必等待。并发管理器根据事务的开始时间选择请求的块的适当版本。更新事务可以同时对同一块进行更改，但是只读事务将不在乎，因为它看到该块的不同版

本。

多版本锁定仅适用于只读事务。更新事务需要遵循锁定协议，并根据需要获取slock和xlocks。原因是每个更新事务都读取和写入数据的当前版本（而不是以前的版本），因此可能发生冲突。但是请记住，这些冲突仅发生在更新事务之间，而不是只读事务之间。因此，假设更新事务相对较少，则多版本锁定的等待频率将大大降低。

### 实现多版本锁定

现在，我们已经了解了多版本锁定应该如何工作，让我们看一下并发管理器如何执行所需的操作。最基本问题是如何维护每个块的版本，一种简单但有些困难的方法是将每个版本显式地保存在专用的“版本文件”中。取而代之的是，我们将使用一种不同的方法，在该方法中，日志可用于重构块的任何所需版本，该实现工作流程如下。

更新事务的提交过程，以包括以下操作：

- 为此事务选择一个时间戳。
- 恢复管理器将时间戳作为事务提交日志记录的一部分。
- 对于事务持有的每个互斥锁，并发管理器固定该块，将时间戳写入该块的开头，然后取消固定该缓冲区。

每个只读事务在启动时都会被赋予一个时间戳。假设具有时间戳 $t$ 的事务请求块 $b$ 。并发管理器采取以下步骤来重建适当的版本：

1. 拷贝块 $b$ 的当前版本到一个新的页中。
2. 从后往前读取日志文件，遍历3次

2.1 构造一个在时间 $t$ 之后的事务列表。由于事务是以按照时间顺序提交的，所有当发现一条时间戳小于 $t$ 的提交日志记录时，停止往前读日志记录。2.2 构造一个未提交的事务列表，即寻找那些没有提交日志记录或回滚日志记录的事务。当遇到一个静态检查点日志记录时，可以停止往前读取日志记录。2.3 使用那些更新日志记录来撤销更改，是的拷贝块 $b$ 中的值为old values。

3. 修改后的拷贝块返回给只读事务。

也就是说，通过undo那些时间 $t$ 以后的修改，并发管理器会构造时间戳 $t$ 时各块的版本。实际上，可以重写上述算法，从而使得并发管理器在一次遍历就完成上述功能，而不是3遍，详细请参阅练习14.39

最后，我们注意到，一个事务需要指明它是否是只读事务，因为并发管理器对两种类型的事务（即 更新事务 和 只读事务）有不同的处理。在 JDBC 中，此规范由 `Connection` 接口的 `setReadOnly()` 方法来指明，例如：

```
Connection conn = ... // obtain the connection
conn.setReadOnly(true);
```

对 `setReadOnly()` 方法的调用被看作是对数据库系统的“提示”。如果系统不支持多版本锁定，则可以选择忽略该调用。

#### 14.4.7 事务隔离级别

正如我们所看到的，强制顺序化会导致大量等待，因为锁协议要求事务持有其锁定直到事务完成。因此，如果事务  $T_1$  恰好需要一个与  $T_2$  持有的锁发生冲突的锁，则  $T_1$  在  $T_2$  完成之前不能做任何其他事情。

多版本锁定非常吸引人，因为它允许只读事务在没有锁定的情况下执行，因此不必等待。但是，多版本锁定的实现有些复杂，并且需要其他磁盘访问来重新创建不同版本。此外，多版本锁定不适用于更新数据库的事务。

事务还有另一种减少等待锁的时间的方法——可以指定它确实需要完整的可顺序化性。在第8章中，我们看到了JDBC定义了四个事务隔离级别，图 14-25 总结了这些级别及其属性。

ISOLATION LEVEL	PROBLEMS	LOCK USAGE	COMMENTS
serializable	none	locks held to completion, lock on eof marker	the only level that guarantees correctness
repeatable read	phantoms	locks held to completion, no lock on eof marker	useful for modify-based transactions
read committed	phantoms, values may change	locks released early, no lock on eof marker	useful for conceptually separable transactions whose updates are “all or nothing”
read uncommitted	phantoms, values may change, dirty reads	no locks at all	useful for read-only transactions that tolerate inaccurate results

Figure 14-25: Transaction isolation levels

第8章将这些隔离级别与可能发生的各种问题相关联。图14-25的不同之处在于，还将这些级别与使用锁的方式相关联。Serializable隔离级别要求严格的共享锁定，而read-uncommitted隔离级别甚至不使用锁。显然，锁定的限制越少，等待的时间就越少。但是限制较少的锁定也会在查询结果中引入更多的不准确性：事务可能会看到 幻象(phantom)，或者它可能会在不同时间在某个位置看到两个不同的值，或者可能会看到未提交的事务写入的值。

同样重要的是要注意，这些隔离级别仅适用于数据读取。所有事务，无论其隔离级别如何，在写入数据方面均表现正常。他们必须获取适当的 `xlock`（包括eof标记上的`xlock`）并一直持有它们直到事务完成。原因是单个事务在执行某次查询时，可能可以容忍结果的不准确，但是不正确的更新会影响整个数据库系统，这是绝对不能容忍的。我们接下来简要讨论一下这些隔离级别。

##### **serializable** 级别

这个隔离级别对应锁机制，并且是唯一确保ACID原则中 独立性原则 (*isolation*) 的级别。理论上，每个事务应该使用serializable级别的隔离，但是在实际上，serializable级别的隔离对一些高性能应用来说速度太慢了。选择 非顺序化(*non-serializable*) 隔离级别的开发人员必须仔细考量可能出现的不准确结果的程度，以及这些不准确结果的可容忍程度。

#### ***repeatable read* 级别**

减少使用共享锁的第一种方法是不对文件的eof标记符进行共享锁定。在这种情况下，当有另一个事务在对文件追加一个块时，该事务将会发现，因此在每次检查文件时会看到新的幻象值。

#### ***read committed* 级别**

事务可以通过提前释放锁来进一步减少共享锁的影响。在讨论锁定协议时，我们检查了由于过早释放共享锁导致的问题。基本问题是：假设事务  $T_1$  在获得另一个共享锁之前释放了一个共享锁，并且在此间隔内事务  $T_2$  修改了两个未锁定的块。此次修改有效地将事务  $T_1$  分为两半——在  $T_2$  之前读取的值是一致的，在  $T_2$  之后读取的值是一致的，但是任一半上的值彼此不一致。

虽然一个JDBC事务可以指明为*read committed* 级别的事务，但是她无法告诉并发管理器什么时候释放共享锁。该具体决定留给各个数据库系统。Oracle做出了一个有趣并且很有用的选择，Oracle选择在事务中的每个SQL语句之后释放共享锁。因此，每个SQL语句的结果是一致的，但是结果不必彼此保持一致。从某种意义上说，每个语句的行为都类似于在一个单独的事务中。

如果事务包含多个活动，且这些活动在概念上是彼此独立的，则事务将选择*read committed* 级别的隔离。这样的事务几乎与一系列较小的事务相同。不同之处就在于，由事务执行的更新将表现为原子的，即一个“all or nothing”的单元。

#### ***read uncommitted* 级别**

最小限制地使用共享锁的方法是根本不使用锁。因此，使用*read uncommitted* 隔离级别的的事务执行非常快，但也遭受严重的一致性问题。除了上述的隔离级别产生的问题之外，该事务还可能带来读取未提交的事务所做的修改（称为“脏读取”）的问题，因此无法获取数据库的准确快照。

如果一个只读事务正在计算可接受近似答案的某种信息（例如计算统计信息的某次查询），则它将使用此级别的隔离。这样的查询的一个例子是是“每年，计算该年给定的所有成绩的平均值”。计算结果可能会错过一些更新或新成绩，但结果相对来说还是较为准确的。

### **14.4.8 数据项的粒度**

我们已经从锁定块的角度讨论了并发管理，但是和在恢复过程中一样，其他粒度的锁也是可能的：并发管理器可以锁定一个值，文件，甚至整个数据库，我们称一个被锁定的单元为 `并发数据项 (concurrency data item)`。

一个并发数据项是并发管理器锁定对象的单元。

并发控制的原则并不随着并发数据项的粒度变化而变化，我们上述所有的定义，协议和算法，都适用于任何粒度的并发数据项。因此，粒度的选择是一种实际的选择，它需要在效率和灵活性之间取得平衡，本节研究了一些折中方案。

并发管理器对每个数据项都维护了一把锁，更小的粒度很有用，因为它将允许更大程度的并发。例如，假设两个事务希望并发地修改相同块的不同部分，这些并发修改在值粒度下的锁定时可以实现，而在块粒度下的锁定是就无法实现。

然而，小的粒度需要更多的锁。值粒度往往会使数据项变得不切实际，因为它们需要大量的锁。在另一种极端情况下，将文件用作数据项的粒度将需要很少的锁定，但也会显著地影响系统的并发性——客户端需要获得整个文件的互斥锁，从而更新该文件中的任何部分。将块用作数据项的粒度是一个合理的折中方案。

顺便说一句，我们注意到某些操作系统（例如MacOS和Windows）使用文件粒度锁定来实现原始形式的并发控制。特别是，应用程序无法在没有持有某个文件的互斥锁的情况下写入该文件，并且如果该文件当前正被另一个应用程序使用，则它无法获得该文件的互斥锁。

有些并发管理器同时支持多个粒度的数据项，例如同时支持块和文件的粒度。一个只访问某个文件中一小部分块的事务可以以块的粒度为数据项来并发控制；而假如另一个事务会访问文件中的绝大部分，那么它可以使用文件粒度。这种方法将小粒度的灵活性与大粒度的便利性有机结合了起来。

另一种可能的粒度是使用 `数据记录 (data record)` 来控制并发。数据记录是由记录管理器来维护的部分，是下一章的主题。在本书中，我们已经把并发管理器作为是记录管理器的下层组件，可以参考Part 3的开头回忆一下。因此，并发管理器无法理解数据记录，因而也就无法给他们上锁。然而，在一些商用的数据库系统（例如Oracle）正是基于这样一种架构的，即并发管理器是位于记录管理器之上的组件，因此并发管理器可以调用记录管理器的相应方法，在这些商业系统中，数据记录就是那些最自然的并发数据项。

虽然说数据记录粒度看起来非常吸引人，但是它会引入幻象的问题。由于新的数据记录可以被插入到一个已经存在的块中，因此从一个块中读取所有记录的事务需要引入一种方法来防止其他事务将记录插入该块中。解决

方案是让并发管理器还支持粒度更大的数据项，例如块或文件。实际上，某些商业系统通过简单地强制事务在执行任何插入之前在文件上获取互斥锁来避免产生幻象问题。

#### 14.4.9 SimpleDB的并发管理器

SimpleDB的并发管理器是通过 `ConcurrencyMgr` 类来实现的，在包 `simplesdb.tx.cconcurrency` 包中。这个并发管理器实现了锁协议，使用的是块级的并发数据项粒度，其API如下所示：

```
public class ConcurrencyMgr {  
    public ConcurrencyMgr(int txnum);  
    public void sLock(Block blk);  
    public void xLock(Block blk);  
    public void release();  
}
```

每个事务都会创建其自己的并发管理器，并发管理器的方法和锁表有点相似，但是是transaction-specific的。每个 `ConcurrencyMgr` 对象维护了其对应事务的持有的锁，如果事务还没有持有相应的锁，则方法 `sLock()` 和 `xLock()` 会从锁表中请求持有对应的锁。方法 `release()` 在事务的最后释放持有的锁。

`ConcurrencyMgr` 类会使用 `LockTable` 类，它是SimpleDB的锁表，本节的其余部分将研究这两个类的实现。

##### ***LockTable* 类**

`LockTable` 类的实现如下所示：

```

public class LockTable {

    private static final long MAX_TIME = 10000; // 10 s
    private Map<Block, Integer> locks = new HashMap<>();

    /**
     * 请求持有指定块的 共享锁
     *
     * @param blk 指定的块
     */
    public synchronized void sLock(Block blk) {
        try {
            long timestamp = System.currentTimeMillis();
            // 当该块的互斥锁已经被持有时, 线程等待
            while (hasXLock(blk) && !waitTooLong(timestamp))
                // this.wait(), 等待的对象是当前这个锁表,
                // 等待的目标是该块的互斥锁被释放
                wait(MAX_TIME);

            // 死锁发生了
            if (hasXLock(blk))
                throw new LockAbortException();

            int val = getLockVal(blk); // 这个val肯定是个非常小的数
            locks.put(blk, val + 1);
        } catch (InterruptedException e) {
            throw new LockAbortException();
        }
    }

    /**
     * 请求持有指定块的 互斥锁
     * <p>
     * 我们假定事务已经获取了指定块的 共享锁。 (即locks对应 blk的entry)
     *
     * @param blk 指定的块
     */
    public synchronized void xLock(Block blk) {
        try {
            long timestamp = System.currentTimeMillis();
            // 当该块的共享锁已经被其他事务持有时, 线程等待
            while (hasOtherSLocks(blk) && !waitTooLong(timestamp))
                // this.wait(), 等待的对象是当前这个锁表,
                // 等待的目标是该块的共享锁被释放
                wait(MAX_TIME);

            // 死锁
            if (hasOtherSLocks(blk))

```

```

        throw new LockAbortException();

    locks.put(blk, -1); // 获得互斥锁，把锁表置为-1
} catch (InterruptedException e) {
    throw new LockAbortException();
}
}

public synchronized void unLock(Block blk) {
    int val = getLockVal(blk);
    if (val > 1)
        locks.put(blk, val - 1);
    else {
        locks.remove(blk);
        // 该块变得空闲，通知所有等待线程竞争
        notifyAll();
    }
}

/**
 * 判断指定块的互斥锁是否已经被占用。
 * <p>
 * getLockVal(block) 为 -1 时表示互斥锁被占用。
 *
 * @param block
 * @return
 */
private boolean hasXLock(Block block) {
    return getLockVal(block) < 0;
}

/**
 * 判断指定块的共享锁是否已经被持有。
 * <p>
 * getLockVal(block) 为 被持有的次数。
 *
 * @param block
 * @return
 */
private boolean hasOtherSLocks(Block block) {
    return getLockVal(block) > 1;
}

private int getLockVal(Block block) {
    Integer val = locks.get(block);
    if (null == val)

```

```

        return 0;
        return val;
    }

    private boolean waitTooLong(long startTime) {
        return System.currentTimeMillis() - startTime > MAX_WAIT_TIME;
    }
}

```

`LockTable` 对象会维护一个名为 `locks` 的 map，这个 map 中每个 entry 为已经分配锁的块，每个 entry 的值是一个 `Integer` 对象，值为 -1 时表示互斥锁已经被分配，一个正数表示当前分配的共享锁数量。

`sLock()` 方法和 `xLock()` 方法和 `BufferMgr` 类中的 `pin()` 和 `pinNew()` 方法很类似，每个方法都包含一个循环，循环中会调用 Java 里的 `wait()` 方法来等待获取相应的锁。`sLock()` 会调用 `hasXLock()` 方法来判断请求共享锁的块对应的互斥锁是否已经被持有，`hasXLock()` 方法在对应的 entry 值为 -1 时返回 `true`；类似地，`xLock()` 会调用 `hasOtherLocks()` 方法来判断请求互斥锁的块对应的互共享锁是否已经被持有，`hasOtherSLocks()` 方法在对应的 entry 值大于 1 时返回 `true`。

注意！注意！注意！这里有一个非常重要的 trick：并发管理器在获取到一个块的 `xlock` 前总是会先请求该块的 `slock`，所以，当锁表中的某个 entry 值大于 1 时，表示有其他的事务也对该块持有一个共享锁。

`unlock()` 方法要么从 `locks` 从移除一个 entry（当该块的互斥锁或共享锁只被 1 个事务持有时），要么将一个共享锁 entry 的值减 1。当从 `locks` 中移除一个 entry 后，会调用 Java 中的 `notifyAll()` 方法来唤醒那些等待锁的线程准备竞争 CPU 的调度，Java 内部的线程调度器会选择一个线程抢占到相应锁。当一个线程被分配到一个锁后，刚才都被唤醒的线程中的某些可能发现自己又需要等待...一直这样下去。

其实上述的代码在处理线程通知的时候并不高效，`notifyAll()` 方法会唤醒所有的线程。举例来说，如果线程 A 在等待锁表中块 1 的锁，而线程 B 在等待锁表中块 2 的锁，当线程 C 释放块 2 的锁时，线程 A 和线程 B 都会被唤醒，但是线程 A 又会发现自己需要等待，这其实是不太高效的。一方面，如果有相对较少的冲突的数据库线程同时运行，则此策略不是那么的 costly；另一方面，数据库系统应该比这更复杂。练习 14.50-14.51 会让你改善当前的这种等待/通知机制。

### ConcurrencyMgr 类

ConcurrencyMgr 类的代码如下：

```

public class ConcurrencyMgr {
    // 全局锁表，所有的事务共享锁表
    private static LockTable lockTbl = new LockTable();
    // 当前事务的持有锁情况
    private Map<Block, String> locks = new HashMap<>();

    /**
     * 为当前事务请求指定块的共享锁。
     * <p>
     * 如果当前事务未持有共享锁，才会去为此事务请求对应块的共享锁
     *
     * @param blk 指定的块
     */
    public void sLock(Block blk) {
        if (null == locks.get(blk)) {
            lockTbl.sLock(blk);
            locks.put(blk, "S");
        }
    }

    /**
     * 为当前事务请求指定块的互斥锁。
     * <p>
     * 如果当前事务未持有共享锁，才会去为此事务请求对应块的共享锁
     * <p>
     * 假设，事务在获取到一个块的xlock前总是会先请求该块的slock
     *
     * @param blk 指定的块
     */
    public void xLock(Block blk) {
        if (!hasXLock(blk)) {
            sLock(blk);
            lockTbl.xLock(blk);
            locks.put(blk, "X");
        }
    }

    public void release() {
        for (Block blk : locks.keySet())
            lockTbl.unLock(blk);
        locks.clear();
    }

    private boolean hasXLock(Block blk) {
        String lockType = locks.get(blk);
        return (lockType != null && lockType.equals("X"));
    }
}

```

```
    }  
}
```

虽然说对于每个事务都有一个并发管理器，但是所有的事务都共享一个锁表，这个是通过 `ConcurrencyMgr` 类中的静态成员 `lockTbl` 来实现的。而对于每个事务独有的锁，则由名为 `locks` 的map维护，对于该事务持有锁的每个块，该map中都会有一个对应的entry，entry的值为S或X，分别标识slock或xlock。

方法 `sLock()` 首先会检查一下当前事务是否已经对块持有锁，如果已经持有锁，也就没有必要去操作锁表了；否则，就会调用锁表的 `sLock()` 方法，并且等待直到锁被赋予。如果当前事务已经对指定块持有互斥锁，那么方法 `xLock()` 什么也不做；如果没持有互斥锁的话，那么会先调用 `sLock()` 方法获得该块的slock，然后再获得xlock。（回忆一下，我们在前面已经说到，锁表的 `xLock()` 方法会假设当前事务已经持有了指定块的slock）

## 14.5 实现SimpleDB的事务

14.2小节介绍了 `Transaction` 类的API，我们现在终于来到了讨论其实现的时间点到了。

`Transaction` 类使用到了 `BufferLsit` 类，该类管理了该事务固定的缓冲区。我们将逐个讨论他们的实现。

### *Transaction* 类

`Transaction` 类的实现如下，每个 `Transaction` 类的对象会创造它自己的恢复管理器和并发管理器，它也会创建一个 `BufferLsit` 类的对象来管理当前固定的缓冲区。

```
public class Transaction {  
    private static int nextTxNum = 0;  
    private RecoveryMgr recoveryMgr;  
    private ConcurrencyMgr concurMgr;  
    private int txNum;  
    private BufferList myBuffers = new BufferList();  
  
    public Transaction() {  
        txNum = nextTxNumber();  
        recoveryMgr = new RecoveryMgr(txNum);  
        concurMgr = new ConcurrencyMgr();  
  
    }  
  
    public int getTxNum() {  
        return txNum;  
    }  
  
    public void commit() {  
        myBuffers.unpinAll();  
        recoveryMgr.commit();  
        concurMgr.release();  
        System.out.println("transaction " + txNum + " comm:  
    }  
  
    public void rollback() {  
        myBuffers.unpinAll();  
        recoveryMgr.rollback();  
        concurMgr.release();  
        System.out.println("transaction " + txNum + " rolled  
    }  
  
    public void recover(){  
        SimpleDB.bufferMgr().flushAll(txNum);  
        recoveryMgr.recover();  
    }  
  
    public void pin(Block blk) {  
        myBuffers.pin(blk);  
    }  
  
    public void unpin(Block blk) {  
        myBuffers.unpin(blk);  
    }  
  
    public int getInt(Block blk, int offset) {  
        concurMgr.sLock(blk);  
        Buffer buff = myBuffers.getBuffer(blk);  
    }  
}
```

```

        return buff.getInt(offset);
    }

    public String getString(Block blk, int offset) {
        concurMgr.sLock(blk);
        Buffer buff = myBuffers.getBuffer(blk);
        return buff.getString(offset);
    }

    public void setInt(Block blk, int offset, int val) {
        concurMgr.xLock(blk);
        Buffer buff = myBuffers.getBuffer(blk);
        // 返回追加一条日志记录后的LSN
        int lsn = recoveryMgr.setInt(buff, offset, val);
        buff.setInt(offset, val, txNum, lsn);
    }

    public void setString(Block blk, int offset, String val) {
        concurMgr.xLock(blk);
        Buffer buff = myBuffers.getBuffer(blk);
        // 返回追加一条日志记录后的LSN
        int lsn = recoveryMgr.setString(buff, offset, val);
        buff.setString(offset, val, txNum, lsn);
    }

    /**
     * 获取文件的大小, 即块的数量
     *
     * @param fileName 指定文件名
     * @return 快数
     * @throws IOException
     */
    public int size(String fileName) throws IOException {
        // 模拟的文件EOF
        Block dummyBlk = new Block(fileName, -1);
        concurMgr.sLock(dummyBlk);
        return SimpleDB.fileMgr().size(fileName);
    }

    public Block append(String fileName, PageFormatter pfmt) {
        // 模拟的文件EOF
        Block dummyBlk = new Block(fileName, -1);
        concurMgr.xLock(dummyBlk);

        Block blk = myBuffers.pinNew(fileName, pfmt);
        unpin(blk);
        return blk;
    }
}

```

```
private static synchronized int nextTxNumber() {  
    nextTxNum++;  
    System.out.println("new transaction: " + nextTxNum);  
    return nextTxNum;  
}  
  
}
```

`commit()` 方法和 `rollback()` 方法执行了下面的事件：

- 调用恢复管理器相应的提交/回滚事务的方法。
- 调用并发管理器，释放当前事务持有的所有锁。
- 调用 `BufferList` 类的 `unpinAll()` 方法取消固定所有的缓冲区。

方法 `getInt()` 和 `getString()` 首先会获取到相应块的共享锁，然后从指定 `offset` 读取需要的值。方法 `setInt()` 和 `setString()` 则首先会获取到相应块的互斥锁，然后调用恢复管理器的相应方法来构造一条更新日志记录，并获取到该日志记录的 LSN，这个 LSN 随后被作为入参传给缓冲区的 `setInt()` 或 `setString()` 方法。

方法 `size()` 和 `append()` 视文件 EOF 为块号为 -1 的 dummy block，`size()` 会先获取到该块的共享锁，而 `append()` 则会获取到该块的互斥锁，这两个方法解决的是我们之前提到的幻象问题(phantom)。

### ***BufferLsit*** 类

`BufferLsit` 类管理了该事务当前固定的缓冲区，实现代码如下：

```

public class BufferList {

    private Map<Block, Buffer> buffers = new HashMap<>();
    // 已经的所有块
    private List<Block> pins = new ArrayList<>();
    private BufferMgr bufferMgr = SimpleDB.bufferMgr();

    Buffer getBuffer(Block blk) {
        return buffers.get(blk);
    }

    void pin(Block blk) {
        Buffer buff = bufferMgr.pin(blk);
        buffers.put(blk, buff);
        pins.add(blk);
    }

    Block pinNew(String fileName, PageFormatter pfmt) {
        Buffer buff = bufferMgr.pinNew(fileName, pfmt);
        Block blk = buff.block();
        buffers.put(blk, buff);
        pins.add(blk);
        return blk;
    }

    void unpin(Block blk) {
        Buffer buff = buffers.get(blk);
        bufferMgr.unpin(buff);
        pins.remove(blk);

        if (!pins.contains(blk))
            buffers.remove(blk);
    }

    void unpinAll() {
        for (Block blk : pins) {
            Buffer buff = buffers.get(blk);
            bufferMgr.unpin(buff);
        }
    }

    buffers.clear();
    pins.clear();
}
}

```

BufferList 类的对象需要知道两个事情：

- 哪些缓冲区已经被赋值到了指定的块。

- 每个块固定了多少次。

代码中使用了一个map来维护缓冲区，并用了一个list来维护固定的次数，该列表中包含了和固定次数相同多的Block对象（比如说，固定了块 $b_1$ 两次，那么list中就包含两个块 $b_1$ 对象），每次取消固定该块时，都会从列表中删除该块的一个副本。

方法 `unpinAll()` 执行事务提交或回滚时所需的与缓冲区相关的事件——它会让缓冲区管理器flush该事务修改的所有缓冲区，并取消固定所有仍被固定的缓冲区。

## 14.6 测试事务

为了做事务相关的单元测试，我们必须让几个事务并发地运行起来。Java中的 `Thread` 类让运行多线程变得很方便。例如，下述的示例代码创建了3个线程，它们模仿的是14.4.2小节中的线程ABC的执行场景。该程序的输出显示了尝试和授予锁的顺序。练习14.58包含了恢复管理器和并发管理器其他部分的测试驱动程序。

```

public class TransactionTest {

    public static void main(String[] args) {
        SimpleDB.init("studentdb");
        TestA tA = new TestA();new Thread(tA).start();
        TestB tB = new TestB();new Thread(tB).start();
        TestC tC = new TestC();new Thread(tC).start();
    }
}

class TestA implements Runnable {
    @Override
    public void run() {
        try {
            Transaction tx = new Transaction();
            System.out.println("Tx A --> TxNum: "+tx.getTxN());
            Block blk1 = new Block("junk", 1);
            Block blk2 = new Block("junk", 2);
            tx.pin(blk1);
            tx.pin(blk2);
            System.out.println("Tx A: read block 1 start");
            String blk_1_pos_20_val = tx.getString(blk1, 20);
            System.out.println("Tx A: read block 1 at pos 20: "+blk_1_pos_20_val);
            System.out.println("Tx A: read block 1 end");

            Thread.sleep(1000);

            System.out.println("Tx A: read block 2 start");
            int blk_2_pos_88_val = tx.getInt(blk2, 88);
            System.out.println("Tx A: read block 2 at pos 88: "+blk_2_pos_88_val);
            System.out.println("Tx A: read block 2 end");

            tx.commit();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

class TestB implements Runnable {
    @Override
    public void run() {
        try {
            Transaction tx = new Transaction();
            System.out.println("Tx B --> TxNum: "+tx.getTxN());
            Block blk1 = new Block("junk", 1);
            Block blk2 = new Block("junk", 2);
            tx.pin(blk1);
        }
    }
}

```

```
        tx.pin(blk2);

        System.out.println("Tx B: write block 2 start");
        tx.setInt(blk2, 88, 2);
        System.out.println("Tx B write block 2 at pos ");
        System.out.println("Tx B: write block 2 end");

        Thread.sleep(1000);

        System.out.println("Tx B: read block 1 start");
        String blk_1_pos_20_val = tx.getString(blk1, 20);
        System.out.println("Tx B  read block 1 at pos ");
        System.out.println("Tx B: read block 1 end");

        tx.commit();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}

class TestC implements Runnable {
    @Override
    public void run() {
        try {
            Transaction tx = new Transaction();
            System.out.println("Tx C --> TxNum: "+tx.getTxN());
            Block blk1 = new Block("junk", 1);
            Block blk2 = new Block("junk", 2);
            tx.pin(blk1);
            tx.pin(blk2);

            System.out.println("Tx C: write block 1 start");
            tx.setString(blk1, 20, "hello");
            System.out.println("Tx C write block 1 at pos ");
            System.out.println("Tx C: write block 1 end");

            Thread.sleep(1000);

            System.out.println("Tx C: read block 2 start");
            int blk_2_pos_88_val = tx.getInt(blk2, 88);
            System.out.println("Tx C  read block 2 at pos ");
            System.out.println("Tx C: read block 2 end");

            tx.commit();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```
    }  
}
```

上述测试代码的功能是，事务1读取块 $b_1$ 中offset 20处的字符串，也读取块 $b_2$ 中offset 88处的整数；事务2读取块 $b_1$ 中offset 20处的字符串，修改块 $b_2$ 中offset 88处的整数为2；事务3修改块 $b_1$ 中offset 20处的字符串为hello，读取块 $b_2$ 中offset 88处的整数。

### 14.6.1 如何运行测试代码

译者注：这一小节在原书中并没有提供，为了更好地方便读者测试，以下是我的测试方案。

因为现在整个系统的代码都是一个半成品，我们可以尽最大的可能将其按照我们想要测试的行为进行相应修改和测试。首先，为了完成上述测试用例，我们必须先创建一个包含3个块的 `junk` 文件，其实这也就是第12章中的测试代码，稍作修改如下：

```
public class FileTest {

    public static void main(String[] args) {
        try {
            SimpleDB.init("studentdb");

            // 第0块
            Block blk = new Block("junk", 0);
            Page p1 = new Page();
            p1.read(blk);
            // 将第0块的第105字节开始的int整数加1
            int n = p1.getInt(105);
            assert (n == 0);
            p1.setInt(105, n + 1);
            p1.write(blk);

            // 重新读回第0块
            Page p2 = new Page();
            p2.read(blk);
            int added_n = p2.getInt(105);
            assert (added_n == n + 1);

            // 追加另一个block (即 block 1)
            Page p3 = new Page();
            p3.setString(20, "hola");
            blk = p3.append("junk");

            // 再次重新读回追加的block到内存
            Page p4 = new Page();
            p4.read(blk);
            String s = p4.getString(20);
            assert (s.equals("hola"));

            // 追加另一个block (即 block 2)
            Page p5 = new Page();
            p5.setInt(88, 1);
            blk = p5.append("junk");
            assert (p5.getInt(88) == 1);

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

在上述代码中，我们创建了3个块 $b_0, b_1, b_2$ ，其中 $b_1$ 的内容全为0， $b_1$ 的offset为20的位置有一个字面量为 `hola` 的字符串， $b_2$ 的offset为88的位置有一个值为1的整数。

务必注意！！！在创建测试数据块时，我们先将 `server.SimpleDB` 类中 `init()` 方法的初始化日志管理器开关语句注释掉，即：

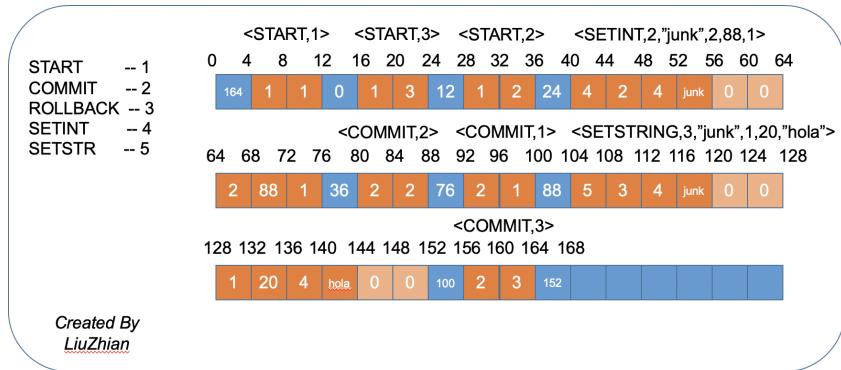
```
public static void init(String dirName) {
    // 初始化文件管理器
    initFileMgr(dirName);
    // 初始化日志管理器 TODO
    // initLogMgr("testLog.log");
    // 初始化缓冲管理器
    initBufferMgr(10);

    boolean isNew = fm.isNew();
    if (isNew) {
        System.out.println("creating a new database");
    } else {
        System.out.println("recovering the existing database");
    }
}
```

运行上述 `FileTest` 类的 `main` 方法后，我们再将 `server.SimpleDB` 类中 `init()` 方法的初始化日志管理器开关打开，并运行 `TransactionTest` 类的 `main` 方法，运行后控制台输出如下：

```
recovering the existing database
new transaction: 1
new transaction: 2
new transaction: 3
Tx A --> TxNum: 1
Tx B --> TxNum: 2
Tx C --> TxNum: 3
Tx A: read block 1 start    // 事务1请求b1的slock
Tx A  read block 1 at pos 20: hola  // 事务1成功得到b1的slock
Tx A: read block 1 end
Tx B: write block 2 start   // 事务2请求b2的xlock
Tx B write block 2 at pos 88 with value '2' success! // 事务2成功得到b2的xlock
Tx B: write block 2 end
Tx C: write block 1 start   // 事务3请求b1的xlock, 此时线程开始
Tx A: read block 2 start    // 事务1请求b2的slock, 此时线程开始
Tx B: read block 1 start    // 事务2请求b1的slock
Tx B  read block 1 at pos 20: hola  // 事务2成功得到b1的slock
Tx B: read block 1 end
transaction 2 committed // 事务2释放掉b1,b2的锁, 通知其他线程
Tx A  read block 2 at pos 88: 2      // 事务1成功得到b2的slock
Tx A: read block 2 end
transaction 1 committed // 事务1释放掉b1,b2的锁, 通知其他线程
Tx C write block 1 at pos 20 with value 'hello' success!
Tx C: write block 1 end
Tx C: read block 2 start    // 事务3请求b2的slock
Tx C  read block 2 at pos 88: 2 // 事务3成功得到b2的slock, 并且
Tx C: read block 2 end
transaction 3 committed // 事务3释放掉b1,b2的锁, 通知其他线程
```

如果你仔细研究一下上述的运行结果，你会发现这和当时我们在14.4.2小节最后的分析结果完全一致。最后，我还想在这里简单讨论一下生成的日志，用一个二进制文件编辑器打开日志文件，并严格按照我们之前定义日志记录的方式，可以解析得到如下图所示的日志：



你的控制台打印输出可能和上述不太一样（主要是事务对应的txNum不一样，事务请求锁的语句对应的位置不一样），但日志记录的顺序肯定是和上述一样的。BTW，通过解析以上日志文件，我发现我的机器在写入一个字符串时，总会在字面量的最后多写两个字节 `00 00`，我猜测这可能和底层文件读写时使用到的 `FileChannel` 有关。

```
public static final int STR_SIZE(int n) {  
    float bytesPerChar = Charset.defaultCharset().newEncoder()  
        // 指示字符串长度的整数 + 各字符占的字节数  
        .return INT_SIZE + n * ((int) bytesPerChar);  
}
```

还记得这个函数吗？在我的机器中，`bytesPerChar`是3，也就是说，字符串“`hola`”到时候会占用 $3 \times 4 + 4 = 16$ 个字节（即开头4个字节表示字符串长度，后12个字节表示字符串字面量，而实际上只会用到12个字节中的前4个），所以后8个字节全为0补齐。

其实再仔细研究一下14.4.2小节中的线程ABC的执行场景，就算我们把并发控制实现好了，但的的确确还是有可能出现死锁的。假设，线程B先得到块 $b_2$ 的xlock，然后线程C先得到块 $b_1$ 的xlock，随后线程A尝试获得块 $b_1$ 的slock时开始等待，随后线程B尝试获得块 $b_1$ 的slock时也开始等待线程C持有的锁，随后线程C尝试获得块 $b_2$ 的slock时也开始等待线程B持有的锁，这样就死锁了。如果出现了这种情况，你再运行一次上述的测试过程即可。

14.7 章末总结

- 当客户端程序能够随意运行时，数据可能会丢失或损坏。数据库系统强制客户端程序使用事务。
  - 事务是被看做为一整个造作的数据库操作集合，满足ACID原则，即原子性(atomicity),一致性(consistency),独立性(isolation),持久性(durability)。

- 恢复管理器是确保原子性和持久性的部分，它辅助读取并处理数据库的日志文件，它的功能有3个：①写日志记录 ②回滚事务 ③系统崩溃后恢复数据库。
- 每个事务在开始时会有一个开始日志记录，在修改数据时会有一个更新日志记录，在事务完成时会有一个提交日志记录或混回滚日志记录。此外，恢复管理器可以写入一个检查点日志记录，来提高下次读取日志文件时的效率。
- 恢复管理器通过从后往前读日志记录来回滚事务，使用的是事务对应的更新日志记录来撤销相应的修改。
- 恢复管理器会在系统崩溃后恢复系统。
- undo-redo恢复算法会undo那些未提交事务所作出的修改，会redo那些已经提交但是没有持久化到磁盘的修改。
- undo-only恢复算法假设已经提交的事务所作出的修改，在事务提交前已经被flush到磁盘上，因此只需undo那些未提交事务所作出的修改。
- redo-only算法假设修改过的缓冲区直到事务提交后才会flush到磁盘。该算法需要事务保持修改后的缓冲区固定，直到完成为止，但是避免了undo那些未提交事务所作出的修改。
- 预写日志(write-ahead logging) 策略要求在修改数据页前把更新日志记录flush到磁盘上，预写日志保证对数据库的修改总是存在于日志文件中，因此也总是可以undo。
- 将检查点记录添加到日志中，以减少恢复算法中需要考虑的日志记录数。
- 恢复管理器可以选择不同粒度的日志数据项，如值、记录、块、文件等，具体粒度的选择需要权衡，大粒度会导致更少的日志记录数，但是单个日志记录的大小会很大。对应地，并发控制器可以选择不同粒度的并发数据项，大粒度会更导致维护更少的锁，小粒度会增强系统的并发性。

## 14.8 建议阅读

## 14.9 练习

## 第15章 记录管理和包 simpledb.record

事务管理器可以从磁盘某个块的指定位置处读取一个值，也可以在某个位置写入一个指定的值，然而，给定文件的块，事务管理器却不知道在块中具体包含了哪些值，更不用说各个值的位置了。在数据库系统中负责解析文件结构的组件是 **记录管理器**，它会将文件组织成一系列的记录的集合，并且拥有遍历所有记录的方法，也有修改记录指定值的方法，在本章中，我们将学习一下记录管理器提供的功能，并且学习一下一些实现日志管理器的技术。

## 15.1 记录管理器

记录管理器 是数据库系统中负责解析文件结构的组件。

当我们设计一个记录管理器的时候，我们需要解决下面这些问题：

- 每一条记录都可以确保一个块就能放下吗？
  - 一个块中的所有记录都应该来自同一个表吗？
  - 每个字段应该用定长的字节数来表示吗？
  - 一个记录中的各个字段的值应该保存在哪里？

这一小节，我们将讨论这些问题。

### 15.1.1 跨块 VS 非跨块记录

假设记录管理器需要插入4条记录，每条记录的长度都是300字节，而每个块的大小为1000字节。显然，前面三个记录会占满块的前900字节，那对于第4条记录，我们该怎么办呢？有两种选择，如图15-1所示：

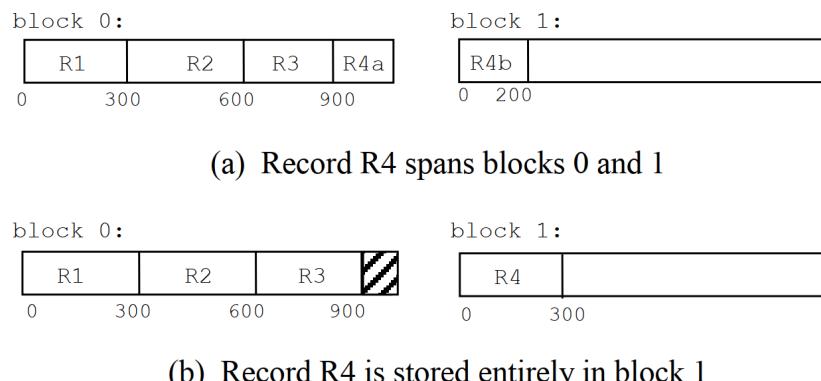


Figure 15-1: Spanned vs unspanned records

在图15-1 (a) 中, 记录管理器创建了一条跨块的记录——也就是说, 第4条记录的前100字节记录在第一个块中, 而后续的200字节则存放在后续的新块中。而在图15-1 (b) 中, 记录管理器直接把第4条记录完完整整地存放在一个新的块中。

跨块的记录, 也就是那些跨越了两个或多个文件块的记录。

记录管理器需要决定是否创建跨块的记录, 创建跨块记录的一个优点就是不会浪费磁盘空间, 在图15-1 (b) 中, 块中的100字节 (10%) 被浪费了。一个更加极端的例子就是, 如果一条记录的长度为501字节, 那么499个字节 (大约50%) 的磁盘空间将被浪费掉。

创建跨块记录的另一个好处就是记录的大小不会与块的大小有关系, 如果一条记录可以比一个块的大小还要长的话, 那么跨块的记录也必须支持。

当然, 创建跨块记录也有缺点, 即会增加记录管理器的复杂度。非跨块的记录很容易管理, 因为它的数据完全都在单个块中, 然而, 一个跨块的记录会跨越多个块, 也就意味值当重建记录数据时, 将需要读多个块, 鉴此, 我们很可能需要将这些块读到一个额外的地方。

### 15.1.2 同构文件 VS 非同构文件

如果一个文件中的所有记录都来自于同一张表, 那我们就说这个文件时同构 (homogeneous) 的。

记录管理器必须决定是否允许同构文件, 这是一个关于效率和灵活性之间的trade-off。

举例来说, 假设我们有两张表STUDENT和DEPT, 在同构的实现下, 我们会把所有的学生记录全部放在一个文件下, 并把所有的专业记录也全部放在另外一个文件下, 这种处理方式使得单表sql查询很容易完成——记录管理器只需要遍历一个文件中的所有块即可, 然而, 在这种情况下, 多表查询将变得不是那么地高效。假设有一个SQL查询, 需要join上述两张表, 例如“查询所有学生的名字以及对应的专业”, 这样以来, 记录管理器就需要在学生记录的块和专业记录的块之间来回穿梭, 为了找到匹配的记录 (我们将在第17章中更加详细地看到)。即使也可以在不进行过多搜索的情况下完成查询 (例如, 通过第21章将会降到的索引连接), 在 STUDENT和DEPT块之间交替时, 磁盘驱动器仍然会需要重复搜索。

在非同构的组织方式下, STUDENT和DEPT的记录将被放在同一个文件中, 这样一来, 我们就可以把某个学生记录和其对应的专业记录存放在相近的位置了。图15-2展示了这种组织情况下某个文件的前2个块, 这里我们假设每个块中有3条记录。该文件中首先包含一条专业记录, 紧接着是3条学生记录, 且这3个学生的专业都是CS, 这种组织方式对于某个join操作将会需要更少的块访问, 因为这些记录都聚集在同一个块 (或者相邻块) 上。

block 0:	block 1:			
10 compsci	1 joe 10 2004	3 max 10 2005	9 lee 10 2004	2 amy 20 2004 ...

Figure 15-2: Clustered, non-homogeneous records

然而，聚集存放会导致单表查询不是那么高效，因为同一个表中的所有记录会跨越很多额块。类似地，join一个不在相邻位置的表的操作也不是很高效。

### 15.1.3 定长字段 VS 变长字段

表中的每个字断都有一个预定义的类型，给予这些类型，记录管理器可以决定是否使用定长或变长的方式来表示字段。定长的字段用固定长度的字节数来表示该字段，而变长的字段基于数据值的不同而采用不同的存储方式。

固定长度的字段表示方式使用相同数量的字节来保存字段的值。

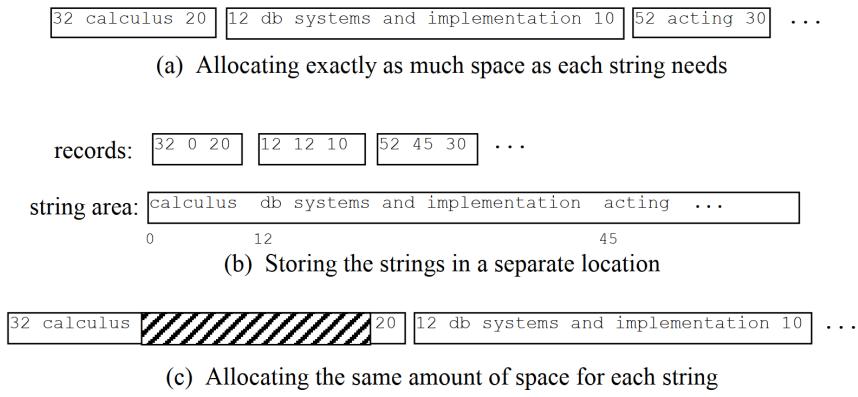
大多数字段都是天然定长的，比方说，integer和服点数都可以用4字节的二进制数来表示，实际上，所有的数值类型、时间、日期都是天然定长的。Java中的 String 类型就是天然变长的字段，因为字符串的字面量可以是任意长度的嘛。

可变长度的表示方式相比于固定长度表示来说就不那么理想，因为它们会引起严重的并发症。例如，考虑一条记录，它位于一个块的中间，且这个块中包含了多条记录，我们假设这条记录的某个字段被修改了。如果该字段是固定长度的，则在修改后，这条记录将保持相同的长度，并且可以在适当位置修改该字段。但是，如果该字段是可变长度的，则这条记录在修改后可能会变长（当然，也有可能会变短）。为了给这条更长的记录腾出空间，记录管理器可能需要重新排列块中的其他记录。实际上，如果修改后的记录太大，则可能需要将一条或多条记录移出该块，并放置在另一个块中。

因此，记录管理器会尽最大可能去使用定长的表达方式，举例来说，记录管理器可以选择三种不同的表达方式来表示一个字符串类型的字段：

- 变长的表达方式，在这种情况下，记录管理器会去申请和字符串等长的空间；
- 定长的表达方式，在这种情况下，记录管理器把字符串的值存储在记录以外的地方，在记录中存储的其实是指向该字符串的指针；
- 定长的表达方式，在这种情况下，记录管理器对字符串的所有可能取值，通通用一个相同长度的空间来表示。

以上三种表达方式可以用图15-3的情况来展示，图中展示了某个 COURSE 表中的一些记录：

Figure 15-3: Alternative representations for the *Title* field in COURSE records

第一种表达方式创建的是变长的记录，这些记录充分地利用了磁盘空间，但也会引发我们之前讨论过的问题（即修改或删除时比较麻烦）；另外两张方式则是定长的表达方式。

第二种表达方式把记录中的字符串单独拿出来，放到一个“字符串区”中，这个区域可以是一个独立的文件，甚至可以是一些目录（其中目录中的每个文件存放一个很大的字符串）。无论是上述的哪种方式，在记录中必定存在一个指向字符串实际所在位置的指针，这种表达方式使得记录是定长的，而且相对来说，长度较短。短小的记录是一件很好的事，因为它们可以用更少的块来存储，因而需要更少的块访问。但这种表达方式的缺点就是记录变得碎片化了（即分开存储在两个地方），也就是说，检索记录中的字符串字段会需要一次额外的块访问。

第三种表达方式的优点是，记录的长度是固定的，而且记录中的字符串也是存在这条记录之内。然而，这种表达方式的缺点则是，一条记录往往需要分配一块比实际所需更大的空间，如果某个字符串字段的所有可能取值长度相差很大，那么这样就会浪费很多的磁盘空间，导致存储一张表需要更大的文件，进而导致更多的块访问。

以上3种表达方式并没有说哪种更好，为了方便记录管理器选择合适的存储方式，标准的SQL中提供了3中不同类型的字符

串：char，varchar 以及 clob。char(n) 表示的是一个字符串的恰好有n个字符，而 varchar(n) 则是说一个字符串最多有n个字

符，clob(n) 也是说一个字符串最多有n个字

符，varchar(n) 和 clob(n) 的区别在于n的大小，varchar中的n一般来说都比较小，比如说，不大于4K；而clob中的n可以大到几G个字符。

（clob 是英文“character large object”的缩写）举例来说，我们假设我们的SECTION表中存在一个Syllabus字段，此时我们就可以用一个clob(8000)来定义该字段。

类型为 char 的字段对应的是上述的第三种字段表达方式。由于所有可能的取值都是相同的长度，因此在记录之间不会有空间浪费，并且这种定长的表达方式是最高效的。

类型为 `varchar` 的字段对应的是上述的第一种字段表达方式。由于SQL要求字符串的长度相对来说比较小，因此，我们知道，在记录中放置一个 `varchar`类型的字符串也不至于让一条记录变得很长。

如果一个字符串的长度很小（比方说，小于20），记录管理器也有可能会选择使用第三种固定长度的表达方式来存储`varchar`，因为这种情况下几乎不会浪费磁盘空间，更重要的是，定长的表达方式会给我们带来更多的方便。

类型为 `clob` 的字段则对应的是第二种字段表达方式，因为这种方式处理大字符串最有效，另外两种表达方式会把字符串直接存放在一条记录内，从而导致很长的记录占有很大的磁盘空间。然而，如果使用一块单独的“字符串区”来存储字符串，记录会变得更小，从而更好管理。

### 15.1.4 记录中各字段的组织方式

记录管理器也必须决定记录本身的结构，对于定长的记录来说，它需要计算下面这些值：

- 文件中每个记录的大小；
- 在一个记录中，各字段的偏移量。

最简单直接的策略是先确定每个字段的大小，然后将各字段依次排列在一条记录中。于是一条记录的大小也就是所有字段大小之和，并且每个字段的偏移量就是上一个字段的结束位置。

固定长度字段的大小取决于它的数据类型，例如：

- 整数需要4字节来保存；
- 对于 `char(n)` 和 `varchar(n)`，如果采用的是定长的表达方式，那么一个字符串需要\$4+n\*c\$个字节来保存，其中c是字符数，额外的4个字节正是用来表示字符串长度的整数。

以上这种紧致存储字段的方式对于基于Java实现的数据库系统比较合适，但在其他平台上可能就会出现问题，这个问题就是，我们有可能需要确保记录中的值在内存中是对齐的。在大多数计算机中，访问`integer`的机器码要求这个整数在内存的存放位置是4的倍数，在这种情况下，我们称整数必须4字节对齐。由于操作系统中的页总是关于\$2^N\$对齐的（N一般是个很大的数），因此每个页的第一个字节总是天然对齐的，因此记录管理器必须确保每个页上的每个整数都是4字节对齐的。如果上一个字段在一个不是4字节对齐的位置处结束，那么记录管理器必须添加一些字节使得对齐。

举例来说，假设我们有一张STUDENT表，其中每个字段包含3个`integer`字段和一个 `varchar(10)` 的字符串字段，其中整型的字段全部是对齐好的，因此不需要pad，然而这个字符串字段需要14个字节来保存（假设一

个字符占1个字节，在Java中往往不会是这样），那么为了实现4字节对齐，我们必须pad2个额外的字节，从而使得字符串结束的位置是4的倍数。

通常来说，不同类型的字段可能会需要不同数量的填充字节，双精度浮点数通常是8字节对齐的，而small int一般是2字节对齐的，负责字节对齐的组件就是记录管理器。一种简单的实现方式就是，对于一条记录中的各个字段，如果它们中的任何一个没有对齐，那就填充直到对齐；另一种比较聪明的方式则是，记录管理器会重新安排记录中各字段的位置，从而使填充的字节最少。举例来说，假设我们有以下的SQL声明语句：

```
create table T (A smallint, B double precision, C smallint,
D int, E int)
```

假设这些字段都是按照A-E的字段依次存储的，按照最简单的实现方式，于是字段A需要6个填充字节，字段C需要2个填充字节，其他字段不需要填充。而按照聪明一点的实现方式，记录管理器会把各字段按照 [B,D,A,C,E] 的顺序排列，这样以来，根本就不需要任何填充字节了，如图15-4所示。

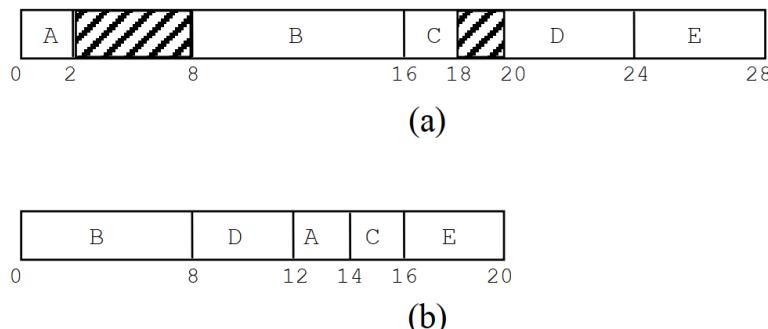


Figure 15-4: Placing fields in a record to establish alignment

除了字段之间需要填充外，记录管理器也需要在各记录之间填充。继续考虑图15-4 (a) 中的例子，一条记录需要用28个字节来存储，假设第一条记录在第0个字节处开始，从而意味着第二条字节将从第28个字节处开始，那么第二条记录中的字段B将从第36个字节处开始，这显然是不满足8字节对齐的，因此，为了解决上述的问题，记录管理器必须在每条记录结尾的地方再填充4个字节。

然而，幸运的是，对于Java程序，我们不需要考虑关于填充的细节，因为Java程序不会从字节数组中直接访问数值类型的值。举例来说，Java中从内存页中读取一个integer的方法为 `ByteBuffer.getInt()`，这个方法不会直接调用机器指令从而获得一个整数，而是从指定的位置开始构造一个整型数。这种策略相对于机器指令来说会慢一些，但是避免了对齐的问题。

## 15.2 实现保存记录的文件

现在我们已经知道了记录管理器必须解决的一些问题，以及相应可选的策略。现在我们来决定一下具体怎么实现，我们从最简单直接的实现开始讨论：即文件中的记录是同构的，非跨块的，并且是固定长度的。我们随后考虑其他的一些实现。

### 15.2.1 最直接的实现

假设我们现在想要创建一个同构的，非跨块的，并且长度固定的记录。非跨块也就是说我们可以把文件看做是一系列的文件块，其中每个块上都包含了每个块自己的记录；此外记录是同构的并且长度固定的特点意味着我们可以在每个块中开辟相同数量的数据记录，也就是说，我们可以把每个块看成是一个包含许多记录的数组，我们称这样的块在内存中对应的页为一个 `记录页 (record page)`。

一个记录页指的是包含定长记录数组的页。

记录管理器可以这样来实现记录页。先将整个页分为很多的 `槽 (slots)`，其中每个槽中含有一条记录和一个额外的字节，这个额外的字节是一个标志位，用来表示这个槽是空的还是在被使用，我们假设0代表空的，1代表在使用（其实我们可以用一个bit来设置标志位，详细请看练习15.8）。

举例来说，假设块的大小是400字节，并且一条记录的大小的26字节；于是每个槽的大小就是27字节，于是每个块可以容纳14个槽，伴随着22字节的浪费空间，图15-5展示了这个场景，图中只展示了4个槽，其中第0个槽和第13个槽当前已经包含有记录，而槽1和槽2是空的。

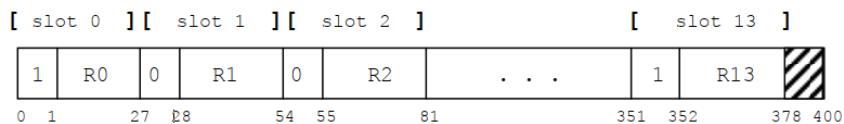


Figure 15-5: A record page with space for fourteen 26-byte records

记录管理器需要可以插入、删除、修改一个记录页中的记录，为了完成这些功能，记录管理器会用到下面的信息：

- 一条记录的长度
- 一条记录中各个字段的名称、类型、长度和偏移量

我们把这些值称为 `表信息 (table information)`。

例如，考虑一下STUDENT这个表，其定义和第2章中一样。一个学生记录包含3个整数和一个10字节的varchar字段。如果我们使用SimpleDB中的存储策略，那么每个整数需要4个字节，一个长度为10字符的字符串需要14个字节(译者注：中文字符另当别论，在我的机器中，每个中文需要3个字节来维护)。并且我们假设不需要padding，且varchar字段的实现策略

是，开辟可能最长字符串的空间，从而长度是固定的。于是图15-6给出了这个学生表的表信息，注意，`SName` 字段的长度该字段存储需要的字节数，而不是这个字段的最大字符数。

Record Length: 26			
Field Information:			
Name	Type	Length	Offset
SIid	int	4	0
SName	varchar	14	4
GradYear	int	4	18
MajorId	int	4	22

Figure 15-6: The table information for STUDENT

这个表信息将允许记录管理器来决定这个记录中哪个地方有什么值，例如：

- 第k个槽的空/使用中标志位的位置是  $(RL+1)*K$  ,其中RL是一条记录的长度。
- 第k条记录中字段F的位置是  $(RL+1)*K + 1 + OFF$  ,其中RL是一条记录的长度，而OFF是字段F在一条记录中的偏移量。

记录管理器因此可以很简答地完成插入、删除和修改记录的过程：

- 为了插入一个记录，记录管理器需要先检查一下记录页中的所有标志位，直到找到第一个0；随后将这个标志位设置为1，并且返回该槽的槽号。如果所有的槽当前的标志位都是1，也就是说当前块是满的，无法插入新的记录。
- 为了删除第k个槽中的记录，记录管理器只需要简单第将该槽的标志位设置为0，其他的什么都不用干。
- 为了修改第k个槽中的记录的某个字段（或者是初始化一个新的记录中的某个字段），记录管理器会先判断一下这个字段的位置，然后在那个位置写入相应的值。

记录管理器也可以很简单第执行检索记录的操作：

- 为了遍历一个块上的所有记录，记录管理器只需要先查看每个槽的标志位，每次标志位是1，记录管理器就知道这个槽是包含记录的。
- 为了检索第k个槽上的记录的某个字段的值，记录管理器先判断一下每个字段的位置，然后从相应的位置读出字段的具体值即可。

正如上面所述的操作一样，记录管理器需要一种方法来识别每一条记录。当记录是定长的时候，最简单直接的记录标识符就是它对应的槽号。

记录页中的每个记录都有一个ID，当记录是定长时，ID可以是槽号。

## 15.2.2 实现变长字段

定长字段的实现非常简单，在本小节中，我们将考虑一下如果加入了变长的字段，那么记录的实现会有怎样的变化。

我们首先会发现的问题就是一个记录中每个字段的偏移量将不再固定。特别地，含有变长字段的记录，记录中每个字段的偏移量可能都是随记录的不同而不同的。判断字段偏移量的唯一办法就是读出前一个字段，然后看下前面的字段在哪里结束的。如果记录中的第1个字段是变长字段，那么为了读第 $n$ 个字段，我们必须先读前 $n - 1$ 个字段。因此，记录管理器通常会把那些定长字段放置在每条记录的开头，以便于可以尽可能多地直接根据偏移量来读取字段的值；而那些变长的字段被放置在记录的后边儿，这样一来，前面的字段会有确定的偏移量，而后面的那些变长字段就没有固定的偏移量了。

其实，我们也会注意到这样一个问题，每个记录的长度也因此会变得不同，这样会带来两个很重要的问题：

- 通过用槽号乘上槽长度的方式来定位块中具体某个记录的方法将不再可能。现在，为了读取到某条记录的开始位置，我们必须获取到它上一条位置的结束位置。
- 修改某个字段的值将可能使得整个记录的长度都变化。在这种情况下，块中所有那些原来在修改的字段右边的字节全部要右移，更糟糕的是，如果修改后的字段值很长，那么这个修改后记录加上原来的记录长度总和可能会顶出整个块的长度，这种情况必须通过开辟一个溢出块(overflow block) 来解决。

一个溢出块指的是从 溢出区(overflow area) 申请的新块。那些顶出到块外面的记录都将被放在溢出块中。如果我们非常不幸于，一个客户端修改了很多记录中的很多变长字段的值，并且修改后的值都是很长很长的，那么就有可能很多记录都要被放到溢出块中去，所以，很可能我们需要多个溢出块来hold它们，于是，溢出块就可能是一个链式的结构。每个块（包括原始块）都会有一个指向下一个溢出块的引用，那么从概念上来说，原始块加上所有的溢出块才构成了我们所有的记录。

例如，考虑一下COURSE表，并且假设这个表中的title字段是以变成字符串形式存储的，图15-7 (a) 展示了这个表中当前包含的3条记录（其中title字段被放在了记录的结尾，因为其他字段都是定长的），图15-7 (b) 展示了将“DBSy”修改成“Database System Implementation”后的情况，假设一个块的大小是80字节，那么第3个记录将在这个块中装不下，于是它被放在了一个溢出块中，原始块会有一个指向溢出块的引用。

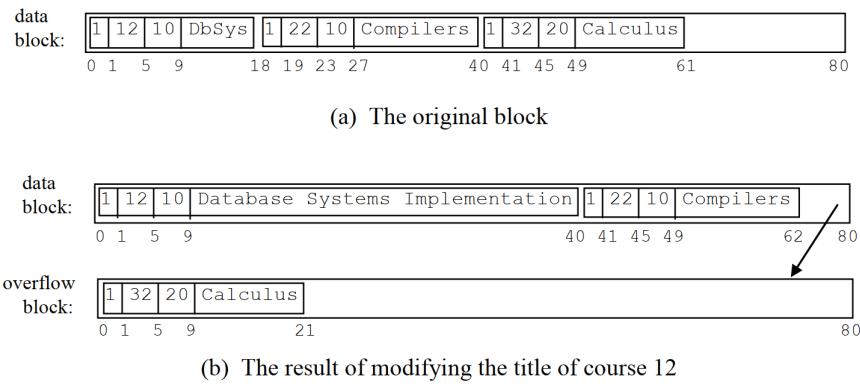


Figure 15-7: Using an overflow block to implement variable-length records

除了需要溢出块这个“缺点”，变长字段引入的复杂性其实并不会对整个系统的效率产生什么影响，当然，记录管理器现在需要做更多的事情：它需要每次在修改了某个字段的值后移动其他的记录，并且为了读取某个记录它必须读取前面的记录。然而，这些操作实际上不会引入额外的块引用，因此不是很耗时，但是，的确有一种常用的改进策略。

考虑一个删除记录的操作，在图15-8 (a) 展示了一个包含3个课程记录的块，和图15-7 (a) 中一样。现在，删除第22号课程记录，我们需要将标志位置为0，但是让记录中的数据原封不动地保留在那，如图15-8 (b) 所示的那样。由于新的记录不一定适合该记录所留下的空白空间，因此我们想通过从记录块中物理移除该记录，并将记录后面的那些记录左移。但是，这里存在一个问题：记录管理器使用记录的槽号作为其ID，因此移除已删除的记录将导致块中的后续记录具有错误的ID。

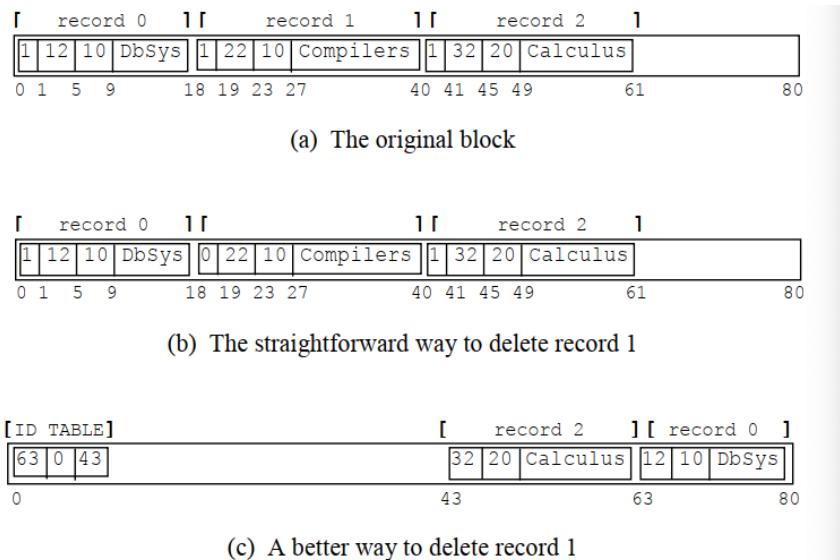


Figure 15-8: Using an ID table to implement variable-length records

该问题的解决办法是把记录的ID和槽号不再关联，也就是说，我们将添加一系列的整型数组到块中，称之为 ID表 (ID-table) 。数组中的每个 entry 表示的是一个ID，每个 entry 的值表示的是该ID对应的记录的起始地址；如果没有该记录，则这个 entry 的值为0，图15-8 (c) 展示了和图15-

8 (b) 中相同的数据情况，但是用了ID表。当前ID表中包含3个entry，其中两个分别指向第43字节起始和第63字节起始的记录，另外一个记录是空的。

ID表提供了一个间接的方式，指导记录管理器如何在块内部移动记。如果记录移动，则相应地调整其在ID表中的entry；如果将记录删除，则将其entry设置为0。插入新记录时，记录管理器在ID表中找到可用的entry并将其分配为新记录的ID。

ID表的存在，允许变长记录在一个块内移动，同时给每个记录都提供了一个定长的标识符。

注意，ID表会随着块中记录数的增加而扩张，ID表的数组大小因此必须是开放的，因为一个块可以装下的变长记录数是不定的。通常，ID表被放在块的一端，而那些记录被放在另一端，它们都各自朝着对方增长，这个情形可以在图15-8 (c) 中观察到，其中第一个记录在最右边。

此外也请注意一下ID表，现在就没必要设置空/使用中的标志位，如果一个记录存在，那么在ID表中必定存在一个指向该记录起始地址的entry，而空的记录肯定是一个值为0的entry（实际上不存在）。ID表同样也帮助记录管理器很快地找到下一个记录。为了移动到一个给定ID的记录，记录管理器只需很简单地使用ID表中存储着的偏移量；而移动至下一条记录，记录管理器只需要往下访问ID表，直到它找到下一个非0的entry。

### 15.2.3 实现跨块的记录

我们接下来考虑一下为了实现跨块的记录，我们要做出那些变化。

当记录不是跨块的时候，一个块中的第一条记录总是在相同的位置开始（例如，从第0号字节开始）。而在跨块的时候，情况就不是这样了。于是记录管理器必须在每个块中存储一些信息，用来判断记录在哪里。

这里有好几种可能的实现策略：

- 第一种策略就是让记录管理器在每个块的开始存一个整数，这个整数表示的是该块中第一条记录的偏移量。例如，考虑一下图15-9 (a) 中的情况，block 0的第一个整数是4，表示第一条记录的起始位置是4（也就是说，紧接着该整数就是第一条记录的起始位置），记录R2跨越了block 0和block 1，因此block 1中的第一条记录是R3，其起始位置是60。记录R3也是跨块的，跨越了block 1、block 2和block 3，因此记录R4是block 3中的第一条记录，偏移量为30.注意，block 2的第一个整数是0，表示在该块中没有记录在该块中开头。

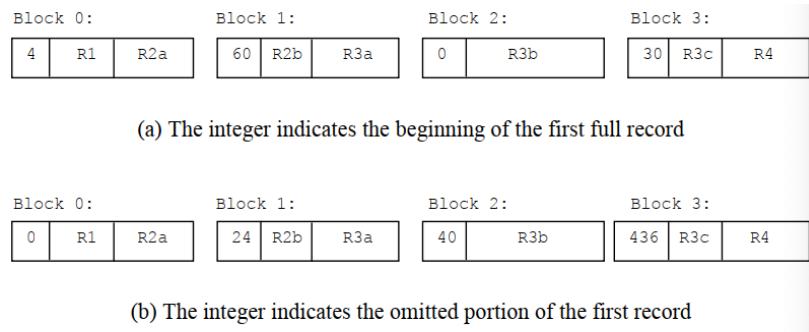


Figure 15-9: Two ways to implement spanned records

- 第二种策略也是在每个块的开始存一个整数，但是这个整数的意思不太一样，即，这个整数表示的是最开始的数据对应于该记录的第多少个字节。如果第一条记录是非跨块的，那么这个整数就是0（即从这条记录的第0个字节开始），否则，这个整数将会是该记录在前面块中的字节数。例如，考虑图15-9（b）中的情况，block 0中的值0表示的是记录R1是非跨块的，而block 1中的值24表示的意思是记录R2的前24个字节都在前面的块中，也就是说该块的第一个字节将是这条记录的第25个字节，而block 3中的值436表示的是记录R3的前436个字节都在前序块中（注意，前序块可能不知一个块，可能是多个块）。在这里，我们假设每个块的大小是400字节，由于记录管理器知道块的大小，于是它可以计算出当前记录是由几个块来装的。

除了上述的实现方案，记录管理器还可以选择以不同的方式来分割一个跨块的记录。

- 第一种方法就是尽可能地填充块，一旦到了块的边界，那就把记录分割开来，剩下的字节就装在文件的下一个块中。
- 第二种方法就是value-by-value地将记录保存在块中，当一个value写不下时，把这个value写在下一个块中，因此，会有很少很少的字节变得浪费。

第一种方法的有点就是完全没有一点儿浪费空间，但是缺点就是可能将一个value的值分布在两个块中（比如，一个int的前3个字节在block 1中，而第4个字节在block 2上），这样会给我们到时候读取值又带来一定困难，记录管理器必须把这些分散的字节给concat起来。

#### 15.2.4 实现非同类记录

假如记录管理器支持非同类记录，那么它也将需要支持变长记录，因为不同表中的不同记录不一定非要是相同长度的，也肯定不能规定是相同长度的。

在块中保存非同类记录将会带来两个相关的问题：

- 记录管理器需要知道块中每种记录的表信息。
- 给定一条记录，记录管理器需要知道它来自哪一个表。

记录管理器可以通过保存一个表信息数组来解决第一个问题，数组中的每个item都是一个表信息；记录管理器可以通过给每个槽添加一个额外的值来解决第二个问题，这个额外的值有时候被称为一个 **标签值(tag value)**，改值就是表信息数组的索引，指示了这条记录属于哪个表。

block 0:					block 1:					...
10 compsci	1 joe	10 2004	3 max	10 2005	9 lee	10 2004	20 math	2 amy	20 2004	

Figure 15-2: Clustered, non-homogeneous records

例如，再次考虑一下图15-2中的例子，该图展示的是一个非同类块的例子，其中的记录来自DEPT表和STUDENT表。记录管理器将会维护一个数组，来保存每个表的表信息，让我们假设DEPT表在数组的第0个item，STUDENT表在第1个item。于是每条DEPT记录对应的槽将有一个标签值0，每条STUDENT记录对应的槽将有一个标签值1。

记录管理器不需要改变太多的行为，当记录管理器访问一条记录时，它会从标签值来判断使用哪一个表信息。它然后可以使用该表信息来读/写任意字段，和在同类文件中的操作一样。

SimpleDB中的日志管理器其实就是一个非同类记录的很好的例子，还记得吗？我们在每种不同的日志记录最开始部分，都用了一个值来指示这是什么日志记录，到底是START Log Record，还是Update Log Record，还是COMMIT Log Record等等。日志管理器在遍历日志文件的时候将会根据这个标签值来决定怎么解析日志记录中的值。

## 15.3 SimpleDB的记录管理器

现在我们已经在概念上知道了记录管理器是怎么工作的，现在我们可以来实现这些具体的需求了，在本节中我们将研究一下SimpleDB的记录管理器是怎么实现的，而实现策略则是15.2.1小节中最简单的那个策略，本章末尾的一些练习将会要求你扩展成其他的实现策略。

### 15.3.1 管理表信息

SimpleDB的记录管理器使用类 `Schema` 和类 `TableInfo` 来管理记录的表信息，其API如下所示：

```

// Schema.java
public class Schema {
    public Schema();

    public void addField(String fldName,int type,int length);
    public void addIntField(String fldName);
    public void addStringFiled(String fldName,int length);
    public void add(String fldName,Schema schema);
    public void addAll(Schema schema);

    public Collection<String> fileds();
    public boolean hasFiled(String fldName);
    public int type(String fldName);
    public int length(String fldName);

}

// TableInfo.java
public class TableInfo {
    public TableInfo(String tblName,Schema schema);
    public TableInfo(String tblName,Schema schema,
                    Map<String,Integer>offsets,int recordLength);

    public String fileName();
    public Schema schema();
    public int offset(String fldname);
    public int recordLength();
}

```

一个 Schema 对象会维持的是一张表的schema，即，每个字段的名称、类型以及长度，这些信息对应的是用户在创建一个表示会指明的信息，并且不包含实际的物理存储信息。例如，string的length指的是最大允许的字符数，而不是实际物理存储一个字符串时用到的字节数。

一个schema可以看做是三元组 [fieldName, type, length] 的集合。类 Schema 包含5个方法来添加一个三元组到该元组集合中。方法 addField() 显式地添加一个三元组，而其他4个方法则是一些更加方便的方法，例如方法 addIntField() 和方法 addStringField() 则会直接添加指定类型的三元组，而 add() 和 addAll() 方法则会从一个现有的schema中复制某个三元组。 Schema 类也包含一些方法来检索字段集合中的所有名字、判断一个指定的字段是否在集合中、检索一个指定字段的类型和长度。

类 ClassInfo 则包含了一个表的物理信息，以及表对应的schema。它会计算字段和记录的大小、一条记录中字段的偏移量、以及存储记录的文件对应的文件名。该类有两个构造函数，对应两种不同的创

建 `TableInfo` 类对象的方法。

- 第一个构造函数会在创建一个表的时候被调用，该构造函数接受表名和对应的schema作为参数，并且会计算出相应字段的物理大小和偏移量。
- 第二个构造函数是会在检索一个已经创建好的表的信息时被调用，调用者会提供已经计算好的值。

该类同样也包含一些方法返回每个字段的名称、每条记录的长度、以及每个字段的偏移量。

下面包含了如何使用这个两个类的示例代码片段。代码中的第一部分创建了一个包含3个字段的COURSE表，然后创建了一个对应的 `TableInfo` 对象；第二部分则表里了schema中的每个字段，打印字段名称和偏移量。

```
Schema sch = new Schema();
sch.addIntegerField("cid");
sch.addStringField("title", 20);
sch.addIntegerField("deptid");
TableInfo ti = new TableInfo("course", sch);

for (String fldname : ti.schema().fields()) {
    int offset = ti.offset(fldname);
    System.out.println(fldname + " has offset " + offset);
}
```

### 15.3.2 实现Schema类和TableInfo类

实现上述两个类很简单，如下所示：

```
public class Schema {
    public static final int INTEGER = 0;
    public static final int VARCHAR = 1;

    private Map<String, FieldInfo> info = new HashMap<>();

    public Schema() {

    }

    public void addField(String fldName, int type, int length) {
        info.put(fldName, new FieldInfo(type, length));
    }

    public void addIntField(String fldName) {
        // int类型的长度设置为0, 这里指的是逻辑长度
        // 而不是实际物理存储所需字节长度
        addField(fldName, INTEGER, 0);
    }

    public void addStringField(String fldName, int length)
        addField(fldName, VARCHAR, length);
    }

    /**
     * 从一个已有的schema中添加一个字段到该schema中
     *
     * @param fldName 字段名
     * @param schema 源schema
     */
    public void add(String fldName, Schema schema) {
        int type = schema.type(fldName);
        int length = schema.length(fldName);
        addField(fldName, type, length);
    }

    /**
     * 将已有的schema中的所有字段到该schema中
     *
     * @param schema 源schema
     */
    public void addAll(Schema schema) {
        info.putAll(schema.info);
    }

    public Collection<String> fields() {
        return this.info.keySet();
    }
}
```

```
public boolean hasField(String fldName) {
    return this.info.keySet().contains(fldName);
}

public int type(String fldName) {
    return this.info.get(fldName).type;
}

public int length(String fldName) {
    return this.info.get(fldName).length;
}

private class FieldInfo {
    int type;
    int length;

    public FieldInfo(int type, int length) {
        this.type = type;
        this.length = length;
    }
}
}
```

在 Schema 类中，使用了一个内部类 FieldInfo 来维护字段三元组信息， FieldInfo 对象包含字段的长度和类型信息。

类型由常数INTEGER和VARCHAR来表示，和JDBC中的类型一样。一个字段的长度信息只有对字符串类型的字段才有意义，因为这代表的是字符数，而不是实际存储该字段所需的字节数，方法 addIntField() 会将int类型数据的type设置为0，这个0其实没什么意义，你也可以设置成-1或者其他，因为这个值不会被用到。

类 TableInfo 的实现如下所示，第一个构造函数中会计算每个字段的时间偏移量，所有的字段都是按找到来的顺序存储在哈希map中，实际上顺序会是随机的，该方法也会计算每个字段所需的字节数，于是整条记录的长度就是所有字段的长度和，并且每个字段的起始位置就是上一个字段的结束位置，也就是说，这里没考虑padding。

```
public class TableInfo {  
    private Schema schema;  
    private Map<String, Integer> offsets;  
    private int recordLen;  
    private String tblName;  
  
    public TableInfo(String tblName, Schema schema) {  
        this.tblName = tblName;  
        this.schema = schema;  
        offsets = new HashMap<>();  
        int pos = 0;  
        for (String fieldName : schema.fields()) {  
            offsets.put(fieldName, pos);  
            // 每个字段需要的字节数  
            // TODO  
            // 1. 这里没考虑字段的padding (即字节对齐)  
            pos += lengthInBytes(fieldName);  
        }  
        recordLen = pos;  
    }  
  
    public TableInfo(String tblName, Schema schema,  
                     Map<String, Integer> offsets, int recordLen) {  
        this.tblName = tblName;  
        this.schema = schema;  
        this.offsets = offsets;  
        this.recordLen = recordLen;  
    }  
  
    /**  
     * 返回保存该类型记录的文件名  
     *  
     * @return 文件名 (包含后缀)  
     */  
    public String fileName() {  
        return tblName + ".tbl";  
    }  
  
    public Schema schema() {  
        return schema;  
    }  
  
    public int offset(String fldname) {  
        return offsets.get(fldname);  
    }  
  
    public int recordLength() {  
        return recordLen;  
    }  
}
```

```

    }

    private int lengthInBytes(String fldName) {
        int fldType = schema.type(fldName);
        if (fldType == INTEGER) {
            return INT_SIZE;
        }
        else
            return STR_SIZE(schema.length(fldName));
    }
}

```

### 15.3.3 管理记录

SimpleDB中的类 RecordPage 管理一个页中的记录，其API如下所示：

```

public class RecordPage {
    public RecordPage(Block blk, TableInfo tableInfo, Transaction tx)
}

public boolean next();
public int getInt(String fieldName);
public String getString(String fieldName);
public void setInt(String fileName,int newVal);
public void setString(String fieldName,String newVal);

public void delete();
public boolean insert();

public void moveToID(int id);
public int currentID();
}

```

这些方法和JDBC中 ResultSet 中的接口很相似，特别地，每个 RecordPage 对象维护了一个“当前对象”，方法 next() 就是移动到下一条记录，当页中不存在下一条记录时，返回false。方法 getXXX()/setXXX() 则是访问当前记录中相应字段的方法， delete() 方法则是删除当前记录。为了和JDBC中保持一致，被删除的记录仍保留在当前记录中，客户端代码必须显式地调用 next() 方法来移动至下一条记录。

insert() 方法不会在当前记录上进行操作，相反，它会在页中某个位置插入一条空的记录，如果没有足够的空间来插入一条新记录，该方法返回false，新的记录对应的位置变成了当前记录。

`RecordPage` 的构造函数接受3个参数：对应的块、`TableInfo` 对象、以及一个 `Transaction` 对象，记录页会使用 `Transaction` 对象来固定和取消固定一个块，并且从中get/set相应的值。

记录页中的每条记录在插入的时候都有一个标识符，直到被删除前，都始终不会变。标识符对于我们将在第21章中讨论到的索引息息相关。`RecordPage` 类提供了2个和标识符交互的方法：方法 `moveToID()` 把当前记录设置成指定标识符对应的记录；方法 `currentID()` 则返回当前记录的标识符。

下面的代码片段继续了15.3.1小节中的测试代码，测试了COURSE表对应文件的block 3的记录。第一部分的代码插入了一条新的记录，其ID为82，`title`为“OO Design”，学院ID为20；第二部分的代码把所以`title`为“OO Design”的课程全部改成了“Java programming”，并且删除了由学院30开设的课程；（译者注：这部分和原书有点出入，只是为了更方便地执行测试，逻辑上没什么差别）

```

public class RecordTest {
    public static void main(String[] args) {

        Schema sch = new Schema();
        sch.addIntField("cid");
        sch.addStringField("title", 20);
        sch.addIntField("deptid");
        TableInfo ti = new TableInfo("course", sch);

        SimpleDB.init("liuzhian/studentdb");
        Transaction tx = new Transaction();
        Block blk = new Block(ti.fileName(), 3);
        RecordPage rp = new RecordPage(blk, ti, tx);

        // Part 1
        boolean ok = rp.insert();
        if (ok) {
            rp.setInt("cid", 82);
            rp.setString("title", "00 design");
            rp.setInt("deptid", 20);
        }
        ok = rp.insert();
        if (ok) {
            rp.setInt("cid", 80);
            rp.setString("title", "Java programming");
            rp.setInt("deptid", 30);
        }

        // Part 2 移到上第一条记录的前面, 准备遍历
        // 这里一定要移到-1而不是0, 因为searchFor()方法会先将current移到0
        rp.moveToID(-1);
        while (rp.next()) {
            int dept = rp.getInt("deptid");
            if (dept == 30)
                rp.delete();
            else if (rp.getString("title").equals("00 design"))
                rp.setString("title", "Object-Oriented");
        }

        tx.commit();
    }
}

```

### 15.3.4 实现记录页

SimpleDB实现的是基于槽结构的记录页，且空/使用中的标志位用的是一个4字节的int值，而不是一个bit，因为我们当前的SimpleDB不支持基于位的读写（当然，你可以进行相应的改写从而支持），`RecordPage`类的具体实现如下：

```

public class RecordPage {
    public static final int EMPTY = 0, INUSE = 1;

    private Block blk;
    private TableInfo tableInfo;
    private Transaction tx;
    private int slotSize;
    private int currentSlot = -1;

    public RecordPage(Block blk, TableInfo tableInfo, Transaction tx) {
        this.blk = blk;
        this.tableInfo = tableInfo;
        this.tx = tx;
        tx.pin(blk);      // 固定页
        slotSize = tableInfo.recordLength() + INT_SIZE; // 每个槽的大小
    }

    public int getInt(String fieldName) {
        int fieldPos = fieldPos(fieldName);
        return tx.getInt(blk, fieldPos); // 会获得块粒度的共享锁
    }

    public String getString(String fieldName) {
        int fieldPos = fieldPos(fieldName);
        return tx.getString(blk, fieldPos); // 会获得块粒度的共享锁
    }

    public void setInt(String fieldName, int newVal) {
        int fieldPos = fieldPos(fieldName);
        tx.setInt(blk, fieldPos, newVal); // 会获得块粒度的独占锁
    }

    public void setString(String fieldName, String newVal) {
        int fieldPos = fieldPos(fieldName);
        tx.setString(blk, fieldPos, newVal);
    }

    public boolean next() {
        return searchFor(INUSE);
    }

    public void delete() {
        int flagPos = currentPos();
        tx.setInt(blk, flagPos, EMPTY);
        // 被删除的记录仍保留在当前记录中,
        // 客户端代码必须显式地调用next()方法来移动至下一条记录。
    }
}

```

```

    /**
 * 主动释放已经固定的块，这个方法主要是给RecordFile类对象
 * moveTo()时调用，主动unpin掉已经固定的块。
 */
public void close() {
    if (blk != null)
        tx.unpin(blk);
    blk = null;
}

/**
 * 插入一条新的记录。
* <p>
* 首先会从头开始找到一个空的槽。
* 1. 如果存在，则将该槽的标志字节设置为INUSE，且currentSlot指向该槽。
* 2. 如果不存在，返回false
*
* @return
*/
public boolean insert() {
    currentSlot = -1;
    // 找到第一个空的槽
    boolean found = searchFor(EMPTY);
    if (found) {
        int foundSlotStartPos = currentPos();
        tx.setInt(blk, foundSlotStartPos, INUSE);
        // 这里只是将标志位进行了设置，并没有填充插入的记录的具体内容
    }
    return found;
}

/**
 * 移动到指定ID的槽
*
* @param id
*/
public void moveToID(int id) {
    currentSlot = id;
}

public int currentID() {
    return currentSlot;
}

/**
 * 获得当前槽的开始位置
*

```

```

    * @return
    */
    private int currentPos() {
        return currentSlot * slotSize;
    }

    /**
     * 获取指定字段的offset, 该计算公式为 (RL+ 4) * k + 4 + off
     * <p>
     * 其中, RL为一条记录的长度, 4为标志字节的Int长度, k为当前槽号,
     * off为指定字段在一条记录中的偏移量
     *
     * @param fileName
     * @return
     */
    private int fieldPos(String fileName) {
        int off = tableInfo.offset(fileName);
        return currentPos() + INT_SIZE + off;
    }

    /**
     * 检查如果以当前位置为新槽的起始位置, 解析一个槽, 是否会顶出一个
     *
     * @return true--不会超出, 即有效 false--超出, 必然无效
     */
    private boolean isValidSlot() {
        return currentPos() + slotSize <= BLOCK_SIZE;
    }

    /**
     * 找到下一个标志字节为指定值的槽
     *
     * @param flag 标志字节的具体取值
     * @return true--找到了, currentSlot为找到槽的槽号 false--
     */
    private boolean searchFor(int flag) {
        currentSlot++;
        while (isValidSlot()) {
            int stepSlotStartPos = currentPos();
            // 检查标志字节
            if (flag == tx.getInt(blk, stepSlotStartPos))
                return true;
            currentSlot++;
        }
        return false;
    }
}

```

构造函数会把当前槽号 `currentSlot` 初始化为-1，`next()` 方法和 `insert()` 方法都会调用一个私有的 `searchFor()` 方法来找到一个指定flag的槽，`next()` 方法会找下一个flag为INUSE的槽，而 `insert()` 方法会从头开始遍历，找到第一个flag为EMPTY的槽；`searchFor()` 方法则会不断地增加当前槽号（也就是向下遍历），知道找到一个指定flag的槽，或者超出了块（即 `isValidSlot()` 方法的功能）。

`getXXX()/setXXX()` 方法则会相应地调用事务类中的方法，两套方法的区别就是参数不太一样，事务的 `getXXX()/setXXX()` 方法接受的参数是页的指定偏移量，而 `RecordPage` 类中的 `getXXX()/setXXX()` 方法接受的参数则是字段的名称，且这些方法中会负责从字段名到字段偏移量的转化，用到的是 `tableInfo` 对象中的信息，和前面的解释完全一样，就不再赘述了。

`delete()` 方法则会简单地将某条记录所在槽的标志字节设置为EMPTY即可，注意被删除的记录仍保留在当前记录中，客户端代码必须显式地调用 `next()` 方法来移动至下一条记录。

运行上一节中的测试代码后，我们再次遍历一下日志记录，你的结果应该和我的结果差不多，遍历日志记录的代码也很简单。

```
// 遍历日志文件代码
Iterator<LogRecord> iter= new LogRecordIterator();
while (iter.hasNext())

{
    LogRecord record=iter.next();
    System.out.println(record);
}

// 遍历结果
<COMMIT 1>
<SETINT 1 [File: course.tbl, block number: 3] 76 1>
<SETSTRING 1 [File: course.tbl, block number: 3] 8 00 design>
=====
<SETINT 1 [File: course.tbl, block number: 3] 80 0>
<SETSTRING 1 [File: course.tbl, block number: 3] 84 >
<SETINT 1 [File: course.tbl, block number: 3] 148 0>
<SETINT 1 [File: course.tbl, block number: 3] 76 0>
=====
<SETINT 1 [File: course.tbl, block number: 3] 4 0>
<SETSTRING 1 [File: course.tbl, block number: 3] 8 >
<SETINT 1 [File: course.tbl, block number: 3] 72 0>
<SETINT 1 [File: course.tbl, block number: 3] 0 0>
<START 1>
```

至于每条日志记录代表什么意思，你自己稍微分析一下就知道了。从下往上数第1-4条是第一次插入的数据记录；而第5-8条日志记录对应第二次插入数据记录；第9-10条则对应的是part 2的相应修改了。

译者注：细心的你可能会发现，我们在定义Schema的时候不是按照cid、title、deptid的顺序定义的吗？为什么从日志记录来看，好像实际存储时是按照deptid、title、cid的顺序来存储的。是的！因为我们在TableInfo类中求各字段的offset时，会遍历schema中三元组字典的所有key，而遍历key的顺序是按照hash值而定的，所以这里可能会发生变化，当然，这种变化对我们毫无影响也必然不会有影响，因为无论什么时候我们找一个字段的值时，都是会按照字段名先去三元组字典中先查找一遍！

### 15.3.5 格式化记录页

一个记录页有一个指定的结构，即，它被分成了多个槽，其实每个槽的第一个整数表示该槽中是否有记录存在，因此一个记录页在被使用前，需要被正确地格式化。类 RecordFormatter 提供了这个服务，通过 `format()` 方法，代码如下：

```

public class RecordFormatter implements PageFormatter {

    private TableInfo tableInfo;

    public RecordFormatter(TableInfo tableInfo) {
        this.tableInfo = tableInfo;
    }

    @Override
    public void format(Page p) {
        int recordSize = tableInfo.recordLength() + INT_SIZE;
        for (int pos = 0; pos + recordSize <= BLOCK_SIZE; pos += recordSize) {
            // 1. 先设置标志位为空
            p.setInt(pos, EMPTY);
            // 2. 再添加默认的记录值
            makeDefaultRecord(p, pos);
        }
    }

    private void makeDefaultRecord(Page page, int position) {
        for (String fldName : tableInfo.schema().fields())
            // 每个字段在一条记录内的offset
            int offset = tableInfo.offset(fldName);
            // int的默认值为0
            if (tableInfo.schema().type(fldName) == Schema.Type.INT)
                page.setInt(position + INT_SIZE + offset, 0);
            // string的默认值为"", 即空串
            else
                page.setString(position + INT_SIZE + offset, "");
    }
}

```

`format()` 方法会遍历页中的所有槽，对每个槽，方法会先把`flag`设置为`EMPTY`，并且将每个字段都初始化为默认值：整数的默认值为0，字符串的默认值为空串。注意，`RecordFormatter` 对象必须知道相应的 `TableInfo` 对象，这样才可以判断每个槽和记录中每个字段的偏移量。

无论何时一个新的记录块append到文件尾部，都需要一个 `RecordFormatter` 对象，下面的示例代码演示了如何使用这个格式化器。代码的功能是追加一个新的块到COURSE表记录文件中，并且插入一条新的记录。注意，`insert()` 方法只是简单第将`flag`设置为`INUSE`，槽中记录具体的取值并没有修改。

```

SimpleDB.init("studentdb");
TableInfo ti = ... //info for the COURSE table

Transaction tx2 = new Transaction();

RecordFormatter recordFormatter = new RecordFormatter(ti);
Block blk2 = tx2.append(ti.fileName(), recordFormatter);
RecordPage rp2 = new RecordPage(blk2, ti, tx2);
rp2.insert();

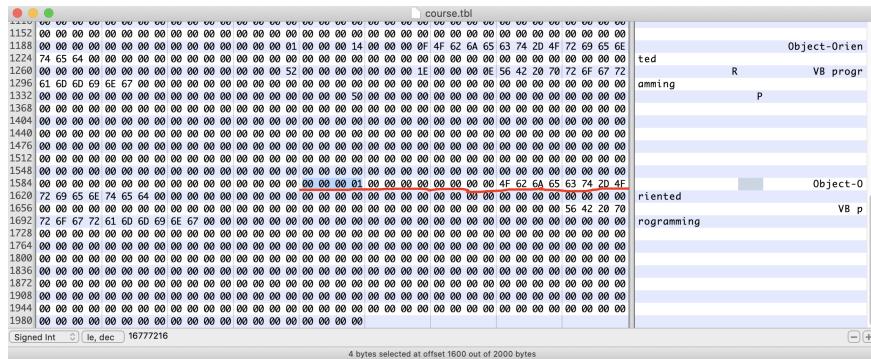
tx2.commit();

```

测试代码执行的结果和上一小节中类似，就不再赘述了。需要注意的是，必须将上述事务2紧接着事务1执行，因为系统中只有一个事务管理器，如果分两次执行（也就是run两次），那么会导致两个事务的ID都是1，造成日志记录事务ID的不一致，会引发严重的不可预测的后果！

- 细节（译者注）：

如果你真的按照上述的测试方式执行一遍，你会发现append出来的新块上的值并不是全为0，甚至还有一些字符串！如下图所示：



这是为什么呢？在 `RecordFormatter.makeDefaultRecord()` 方法中我们不是把int和string都设置为默认值，也就是0和“”吗？其实是这样，因为在按照当前的测试方案，我们开辟了10个缓冲区，并且在tx1提交后，上次的缓冲区肯定是被取消固定了，因此，在 `tx2.append()` 的中间肯定会又找到这个刚被tx1取消固定的缓冲区，并再次固定它。是不是有很奇怪为什么说肯定会找到这个缓冲区呢？因为我们用的是naive缓冲区置换算法呀！每次都是从头遍历，而且我们之前只用了第一个缓冲区。于是乎，新append的块所对应的缓冲区中的内容其实是上一次“遗留下”的值，只不过我们通过 `RecordFormatter.makeDefaultRecord()` 方法把这个缓冲区上的记录的值全部置为了默认值，即0和“”吗，也正是因为我们将字符串默认值置为了空串，所以在底层 `setString()` 的时候并没有改掉原来遗留的字符串，所以我想你应该明白了这是怎么一回事了。出现的当前这种情况并不是错误，我们逻辑上认为这些字符串不存在就够了！

### 15.3.6 管理文件中的记录

一个记录页只能维持相对来说很少数量的记录，而一个 记录文件(record file) 则可以在一个文件中维持很多数量的记录，其API如下所示：

```
// RecordFile.java
public class RecordFile {
    public RecordFile(TableModel tableInfo, Transaction tx);

    // 涉及移动当前记录位置的操作
    public void beforeFirst();
    public boolean next();
    public void moveToRID(RID rid);
    public void insert();
    public void close();

    // 访问当前记录具体数据的方法
    public int getInt(String fldname);
    public String getString(String fldname);
    public void setInt(String fldname, int newVal);
    public void setString(String fldname, String newVal);
    public RID currentRID();
    public void delete();
}

// RID.java
public class RID {
    public RID(int blkNum, int id);
    public int blockNumber();
    public int id();
}
```

和记录页中一样，记录文件也会维持一个当前记录，并且API中包含移动当前记录位置的方法，和访问当前记录具体数据的方法。方法 `next()` 会将当前记录指向记录文件中的下一条记录。如果当前块中没有下一条记录了，那么会继续访问文件的后续块，直到找到另一条记录。方法 `beforeFirst()` 将当前记录指向文件中第一条记录的最前面，当客户端需要遍历记录文件时，在遍历前总是需要调用一次该方法。

`getXXX()` 和 `setXXX()` 方法则是读/写当前记录的相应的字段值，`insert()` 方法会在文件中的某个地方插入一条新的记录，和 `RecordPage` 中不一样，这个插入方法总是会成功，如果在已经存在的块中找不到一个可以插入的地方，那么该方法会 `append` 一个新的块来放置新的记录。

文件中的一条记录可以用一对值来标识，即块号+块内的记录ID，这两个值通常被称为 RID记录标识符（是Record Identifier的缩写），记录标识符是通过类 RID 来实现的，该类的构造函数会保存块引用和一个快中的记录标识符，也有相应的访问方法。

RecordFile 类中包含两个和rid有关的方法，其中方法 moveToRID() 会将当前的记录指向rid对应的记录，该方法假设这个rid执行的记录的确是存在的，方法 currentRID() 返回当前记录的rid。

RecordFile 类提供的抽象级别与到目前为止我们所看到的其他类有明显不同的。也就是

说，Page，Buffer，Transaction 和 RecordPage 这些类的方法都适用于特定的块。而 RecordFile 类向其客户端隐藏掉了块结构的存在。通常，客户端将不知道（或不在乎）当前正在访问哪个块。它只需要在完成操作后关闭记录文件就OK了。

下面的代码展示了如何使用记录文件，代码假设记录文件 junk 中包含的是只有一个int字段A的记录，代码中插入了10000条记录到文件中，每条记录都有一个随机的整数值。然后从记录文件的头部开始遍历所有的记录，删除掉那些值小于100的记录。注意，对 rf.insert() 的调用将根据需要分配尽可能多的新块来保存记录；但是，客户端代码不知道这种情况正在发生。

```
public class RecordFileTest {  
    public static void main(String[] args) throws IOException {  
        SimpleDB.init("liuzhian/simpledb");  
        Transaction tx = new Transaction();  
        Schema schema = new Schema();  
        schema.addIntField("A");  
        TableInfo tableInfo = new TableInfo("junk", schema);  
  
        RecordFile recordFile = new RecordFile(tableInfo, tx);  
        for (int i = 0; i < 10000; i++) {  
            recordFile.insert();  
            int n = (int) Math.round(Math.random() * 200);  
            recordFile.setInt("A", n);  
        }  
  
        int cnt = 0;  
        recordFile.beforeFirst();  
        while (recordFile.next()) {  
            if (recordFile.getInt("A") < 100) {  
                recordFile.delete();  
                cnt++;  
            }  
        }  
        System.out.println("删除的记录数: " + cnt);  
        // recordFile.close();  
        tx.commit();  
    }  
}
```

### 15.3.7 实现记录文件

RecordFile 类的实现如下：

```

public class RecordFile {
    private TableInfo tableInfo;
    private Transaction tx;
    private String fileName;
    private RecordPage recordPage;
    private int currentBlkNum;

    public RecordFile(TableInfo tableInfo, Transaction tx)
        this.tableInfo = tableInfo;
        this.tx = tx;
        this.fileName = tableInfo.fileName();
        // 如果记录文件当前为空，则新追加一个块
        if (tx.size(fileName) == 0)
            appendBlock();
        moveTo(0); // 移动到第0个块
    }

    public void close() {
        recordPage.close();
    }

    // 涉及移动当前记录位置的操作
    public void beforeFirst() {
        moveTo(0);
    }

    /**
     * 判断是否存在下一条记录
     *
     * @return
     * @throws IOException
     */
    public boolean next() throws IOException {
        while (true) {
            // 有下一条记录
            if (recordPage.next())
                return true;
            // 没有下一条记录，并且是在最后一个块
            if (atLastBlock())
                return false;
            // 没有下一条记录，但是不是最后一个块
            moveTo(currentBlkNum + 1); // 移动到下一块
        }
    }

    public void moveToRID(RID rid) {
        moveTo(rid.blockNumber()); // 先移到指定块
        recordPage.moveToID(rid.id()); // 再移到块内指定ID的记
    }
}

```

```

    }

    /**
     * 插入一个记录。
     * <p>
     * 注意，插入总是成功的。
     * 1. 要么是在已有的块中找到一个空位置
     * 2. 要么是追加了一个新的块
     * <p>
     * 新记录的起始位置都可以由recordPage.currentPos()方法获取到
     * <p>
     * 此外，这里的插入其实只修改了记录的 EMPTY/INUSE 标志字节，
     * 客户端代码需要紧接着修改实际的值。
     * @throws IOException
     */
    public void insert() throws IOException {
        // 先移到第0块
        moveTo(0);
        // 当始终找不到一个插入的位置时
        while (!recordPage.insert()) {
            // 如果现在是在最后一个块，那肯定是要追加一个新块了
            if (atLastBlock())
                appendBlock();
            // 否则不断往后面的块里面找
            moveTo(currentBlkNum + 1);
        }
    }

    // 访问当前记录具体数据的方法
    public int getInt(String fldname) {
        return recordPage.getInt(fldname);
    }

    public String getString(String fldname) {
        return recordPage.getString(fldname);
    }

    public void setInt(String fldname, int newVal) {
        recordPage.setInt(fldname, newVal);
    }

    public void setString(String fldname, String newVal) {
        recordPage.setString(fldname, newVal);
    }

    public RID currentRID() {
        return new RID(currentBlkNum, recordPage.currentID);
    }
}

```

```

public void delete() {
    recordPage.delete();
}

/**
 * 移动至指定块
 *
 * @param specificBlkNum 指定的块
 */
private void moveTo(int specificBlkNum) {
    // 主动释放掉缓冲区中已经固定的，对应当前记录块的缓冲区
    if (recordPage != null) {
        recordPage.close();
    }
    currentBlkNum = specificBlkNum;
    Block blk = new Block(fileName, currentBlkNum);
    recordPage = new RecordPage(blk, tableInfo, tx);
}

/**
 * 追加一个新的记录块
 */
private Block appendBlock() {
    RecordFormatter recordFormatter = new RecordFormatter();
    return tx.append(fileName, recordFormatter);
}

/**
 * 判断当前是否在记录文件的最后一块
 *
 * @return
 * @throws IOException
 */
private boolean atLastBlock() throws IOException {
    return currentBlkNum == tx.size(fileName) - 1;
}
}

```

一个 `RecordFile` 对象会为当前的块维护一个记录页，`getXXX()`/`setXXX()`/`delete()` 方法只需调用记录页的相应方法。当前块更改时，将调用私有方法 `moveTo()`，该方法关闭当前记录页，然后为指定的块打开另一个记录页。

`next()` 方法的算法如下：

- 移动至当前记录页中的下一条记录。

- 如果当前记录页中没有下一条记录了，那么久移动至下一个记录块，并从块的第一条记录开始读起。
- 直到遇到文件的结尾（即最后一块也没有下一条记录了），结束。

一个文件的多个块可能为空（请参阅练习15.2），因此可能有必要循环浏览多个块。

`insert()` 方法尝试从当前块开始插入一条新记录，并在文件中移动直到找到一个不是满的块。如果所有块都已满，则它将新块追加到文件中并将记录插入该文件中。

而 `RID` 类的实现就很简单了，如下所示：

```
public class RID {
    private int blkNum;
    private int id;

    public RID(int blkNum, int id) {
        this.blkNum = blkNum;
        this.id = id;
    }

    public int blockNumber() {
        return blkNum;
    }

    public int id() {
        return id;
    }

    @Override
    public boolean equals(Object obj) {
        RID rid = (RID) obj;
        return blkNum == rid.blkNum && id == rid.id;
    }
}
```

## 15.4 章末总结

- 记录管理器是数据库系统中负责将记录存储到文件中的部分，它至少有以下的指责：
  - 在一条记录中修改某些字段。
  - 在一个记录块中修改某些记录。
  - 提供面向记录文件的记录访问方法。
- 很关键的一个问题是是否支持变长的字段。定长记录可以比较容易地实现，因为每条记录的每个字段都会在相同的位置出现，而更新一个

变长的字段就可能导致记录的长度顶出一个块的长度，因此需要 溢出块（overflow block），在simpleDB中，我们还是选取了定长记录的实现策略。

- SQL有3中不同类型的字符串：char、varchar和clob：
  1. char类型最自然地可以用一个定长的字段来实现。
  2. varchar类型最自然地可以用一个变长的字段来实现。
  3. clob类型最自然地可以用一个定长的字段来实现，只不过这个字符串的字面量会保存在另一个辅助文件中。
- 实现变长记录的一个很常用的技术就是构造一个ID表，ID表中的每个entry都指向一条记录的起始位置，一条记录可以通过只改变ID表中entry的值来实现在页中移动。
- 变长记录带来的另一个问题就是是否支持跨块（spanned record）的记录，跨块记录可以避免浪费空间（记录之间的外部碎片），但是跨块记录比较难实现。
- 另一个需要考虑的问题就是在记录文件中是否支持非同类记录，非同类记录允许将来自不同表的记录聚集在同一个块上。群集可以导致联结查询更加高效，但往往会使单表查询变得代价更高。记录管理器可以通过在每个记录的开头存储一个标识字段来实现非同类记录，标识表示该记录所属的表。
- 第四个问题是是如何确定记录中每个字段的偏移量。记录管理器可能需要填充字段，以便它们在适当的字节边界上对齐。定长记录中的字段对于每个记录具来说有相同的偏移量。而在可变长度记录中，可能需要遍历待搜索字段前面的那些字段。

## 15.5 建议阅读

## 15.6 练习

## 第16章 元数据管理和 包 simpledb.metadata

上一章中，我们研究了记录管理器是怎么在文件中存储记录的，然而，正如我们所看到的，一个文件如果仅仅是存储在那里，是没什么作用的，虽然我们在上一章中的相关测试代码中，每次运行程序前都会创建一个 Schema对象和对应的TableInfo对象，但是程序运行完，这些保存在内存中的东西就会丢失，因此，我们必须存储记录所在表的信息，从而下次从记录文件中解码出一条条数据，而所有表的表信息被统称为元数据(metadata)。在本章中，我们将会研究一下一个数据库系统中支持的几种元数据，以及它们各自的目的和功能，以及数据库系统中是怎么存储这些元数据的。

### 16.1 元数据管理器

元数据是关于数据库的信息，但不包括具体的记录内容。

数据库系统会维护一系列的元数据，例如：

- 表(table) 的元数据，描述的是表的记录的结果，例如记录的长度，各字段的类型和偏移量，记录管理器使用的TableInfo类对象其实就是这种元数据的例子。
- 视图(view) 的元数据，描述的是每个视图的属性，例如视图的定义和创建者。
- 索引(index) 的元数据，描述的是某个表已经创建的索引，query planner会使用这个元数据来判断一个query是否可以用索引来评估。
- 数据统计(statistical) 的元数据，描述的是表的大小，以及表中每个字段的统计信息（例如字段的最大值，平均值等等），这类元数据对估计一个查询的用时很有用。

前3类的元数据是在表、视图和索引创建时生成的，而数据统计信息会在数据库每次更新后都会变化。

元数据管理器是数据库系统中负责存储和检索元数据的部分。

元数据管理器会将元数据保存在系统的目录中(system catalog)，它也为客户端提供了检索相应元数据的方法。SimpleDB的元数据管理器是在类 MetadataMgr 中实现的，其API如下所示：

```

public class MetadataMgr {

    public void createTable(String tblName, Schema schema,
                           public TableInfo getTableInfo(String tblName, Transaction t)

    public void createView(String viewName, String viewDef,
                           public String getViewDef(String viewName, Transaction t)

    public void createIndex(String indexName, String tableName,
                           String fieldName, Transaction t)
    public Map<String, IndexInfo> getIndexInfo(String tableName,
                                                Transaction t)

    public StatInfo getStatInfo(String tableName, Transaction t)
}

}

```

API中为每种类型的元数据提供了两个方法——一个用来生成和保存元数据，另一个用来检索元数据。但是数据统计的元数据有点例外，生成数据统计元数据的方法会被系统内部调用，因此是私有的，在上述API中没有列出来。

SimpleDB会在启动时创建一个元数据管理器对象，该对象只有从 SimpleDB类的 `metadataMgr()` 静态方法获取到。

## 16.2 表的元数据

### 16.2.1 使用表的元数据

为了创建一个表，客户端会调用 `createTable()` 方法，传入表名和该表的schema，该方法会计算出该表的其他的元数据(例如，每条记录中各字段的偏移量、存储该表记录的文件名等)，并且将这些信息保存在catalog 中。方法 `getTableInfo()` 会查询catalog，提取出指定表的元数据，并构造出一个TableInfo类对象，把这个对象返回给客户端。

下述的代码片段演示了这些方法的使用，第1部分创建了一个DEPT表，这个表中包含2个字段：一个整型的 `Did`，以及一个`varchar(8)`的字符串字段 `DName`；第2部分（通常来说会在另一个不同的程序中运行）会检索出这张表的元数据，得到一个TableInfo类对象，然后打开相应的记录文件，打印出每个学院的名称。

```

SimpleDB.init("studentdb");
MetadataMgr mdMgr = SimpleDB.mdMgr();

// Part 1: Create the DEPT table
Transaction tx1 = new Transaction();
Schema sch = new Schema();
sch.addIntField("did");
sch.addStringField("dname", 8);
mdMgr.createTable("dept", sch, tx1);
tx1.commit();

// Part 2: Print the name of each department
Transaction tx2 = new Transaction();
TableInfo ti = mdMgr.getTableInfo("dept", tx2);
RecordFile rf = new RecordFile(ti, tx2);
while (rf.next())
    System.out.println(rf.getString("dname"));
rf.close();
tx2.commit();

```

## 16.2.2 实现表的元数据

元数据管理器将元数据保存在数据的目录(catalog)中，但是，怎么实现这个catalog呢？通常来说，数据库系统中采用的策略是，使用数据库表来保存catalog，也就是说，catalog其实也就是数据库中的一张表，只不过这是一张描述其他表的元数据的表。

数据库系统会把元数据存储在catalog表中。

SimpleDB会把表的信息维护在两张目录表中，表tblcat存储的是关于表粒度的信息，而表fldcat存储的是表中每个字段粒度的信息。这两张表的schema如下，其中下划线共同组成了表的主键：

`tblcat (TblName, RecLength)`

`fldcat (TblName, FldName, Type, Length, Offset)`

对于数据库中的每张表都会在tblcat表中存在一条对应的记录，而对于数据库中每张表中的每个字段，则都会在fldcat表中存在一条对应的记录。`RecLength`字段表示的是存储一条记录所需的字节数(不包括empty/inuse标志位)，该值会由对应的TableInfo对象指出，`Length`字段则表示的是表中某个字符串类型字段的最大字符数，正如第15章中Schema类中定义的一样。例如，下图表示的一个大学数据库的catalog表情形，注意一下数据库表的schema信息是怎么被保存到fldcat表中的，图中`Type`字段的4和12分别表示的是INTEGER和VARCHAR类型，这只是JDBC中定义的值，而我们的SimpleDB中分别用0和1来代表，这都只是一个约定而已，没什么太多的特殊意义。

tblcat	TblName	RecLength			
	student	26			
	dept	16			
	course	32			
	section	28			
	enroll	18			
fldcat	TblName	FldName	Type	Length	Offset
	student	sid	4	0	0
	student	sname	12	10	4
	student	majorid	4	0	18
	student	gradyear	4	0	22
	dept	did	4	0	0
	dept	dname	12	8	4
	course	cid	4	0	0
	course	title	12	20	4
	course	deptid	4	0	28
	section	sectid	4	0	0
	section	courseid	4	0	4
	section	prof	12	8	8
	section	year	4	0	20
	enroll	eid	4	0	0
	enroll	studentid	4	0	4
	enroll	sectionid	4	0	8
	enroll	grade	12	2	12

Figure 16-3: Catalog tables for the university database

Note that the constant „student“ is in lower case, even though the table was defined in upper case. The reason is that all table and field names in SimpleDB are stored in lower case, and constants in SQL statements are case-sensitive.

catalog表其实和数据库中的其他表没什么不一样，因此，可以像访问任何用户创建的表一样进行访问。例如，下面SQL查询可以检索出STUDENT表中所有字段的名称和长度：

```
SELECT FldName, Length
FROM fldcat
WHERE TblName="student";
```

细心的你可能会发现这里其实是由一个问题的，这也是在前面我故意隐瞒了一段时间的问题——就是——catalog表既然和普通表没什么区别，那catalog表的schema信息也要存在catalog表中呀，但是在上面的catalog表图示中，我们并没有找到这些记录！是的，这就有点类似一个递归的操作了！别担心，这是一个trick，看看 TableMgr 类的代码是怎么实现的吧！

```

public class TableMgr {
    // 表名和字段名最大长度
    public static final int TABLE_AND_FIELD_NAME_MAX_LEN =
    // tblcat表和fldcat表
    private TableInfo tblCatInfo, fldCatInfo;

    public TableMgr(boolean isNew, Transaction tx) throws ...
        // tblcat表
        Schema tblCatSchema = new Schema();
        tblCatSchema.addStringField("tblname", TABLE_AND_FI
        tblCatSchema.addIntField("reclength");
        tblCatInfo = new TableInfo("tblcat", tblCatSchema);

        // fldcat表
        Schema fldCatSchema = new Schema();
        fldCatSchema.addStringField("tblname", TABLE_AND_FI
        fldCatSchema.addStringField("fldname", TABLE_AND_FI
        fldCatSchema.addIntField("type"); // 字段类型
        fldCatSchema.addIntField("length"); // 对于非字符串字
        fldCatSchema.addIntField("offset"); // 各字段偏移量
        fldCatInfo = new TableInfo("fldcat", fldCatSchema);

        // catalog表tblcat和fldcat自身的schema信息也被存入元数据
        if (isNew) {
            createTable("tblcat", tblCatSchema, tx);
            createTable("fldcat", fldCatSchema, tx);
        }
    }

    /**
     * 将一个新表的元数据到相应catalog表中。
     * <p>
     * 会在元数据表tblcat和fldcat中插入相应的记录。
     *
     * @param tableName
     * @param schema
     * @param tx
     * @throws IOException
     */
    public void createTable(String tableName, Schema schema)
        TableInfo tableInfo = new TableInfo(tableName, schema);

        // 插入一条记录到tblcat表中
        RecordFile tblCatRecordFile = new RecordFile(this.t
        tblCatRecordFile.insert(); // 找到一个插入位置
        tblCatRecordFile.setString("tblname", tableName);
        tblCatRecordFile.setInt("reclength", tableInfo.recl

```

```

tblCatRecordFile.close();

// 为新建表的每个字段相应地插入一条记录到fldcat表中
RecordFile fldCatRecordFile = new RecordFile(this);
for (String fieldName : tblSchema.fields()) {
    fldCatRecordFile.insert(); // 找到一个插入位置
    fldCatRecordFile.setString("tblname", tblName);
    fldCatRecordFile.setString("fldname", fieldName);
    fldCatRecordFile.setInt("type", tblSchema.type);
    fldCatRecordFile.setInt("length", tblSchema.length);
    fldCatRecordFile.setInt("offset", tableInfo.offset);
}
fldCatRecordFile.close();
}

/**
 * 从元数据表中，构造一个已创建表的TableInfo对象。
 *
 * @param tblName
 * @param tx
 * @return
 * @throws IOException
 */
public TableInfo getTableInfo(String tblName, Transaction tx) {
    // 1. 先得到表粒度的信息
    RecordFile tblCatRecordFile = new RecordFile(this);
    int recordLen = -1;
    // 从最前面开始搜索
    // TODO
    tblCatRecordFile.beforeFirst();
    // TODO
    while (tblCatRecordFile.next()) {
        if (tblCatRecordFile.getString("tblname").equals(tblName)) {
            recordLen = tblCatRecordFile.getInt("recLen");
            break;
        }
    }
    tblCatRecordFile.close();

    // 2. 再得到字段粒度的信息
    RecordFile fldCatRecordFile = new RecordFile(this);
    Schema schema = new Schema();
    Map<String, Integer> offsets = new HashMap<>();
    // 从最前面开始搜索
    // TODO
    fldCatRecordFile.beforeFirst();
    // TODO
    while (fldCatRecordFile.next()) {
}

```

```

        if (fldCatRecordFile.getString("tblname").equa-
            String fieldName = fldCatRecordFile.getStr-
            int fileType = fldCatRecordFile.getInt("t")
            int fileLength = fldCatRecordFile.getInt('
            int fileOffset = fldCatRecordFile.getInt('

            // 将各字段的偏移量, 准备添加到TableInfo对象中去
            offsets.put(fieldName, fileOffset);
            // 将各字段, 添加到schema对象中去
            schema.addField(fieldName, fileType, fileLength);
        }

        fldCatRecordFile.close();

        return new TableInfo(tblName, schema, offsets, recLen);
    }
}

```

该类的构造函数会在系统启动的时候被调用，该构造函数会创建 `tblcat` 和 `fldcat` 表的 `schema`，并且构造出相应的 `TableInfo` 类对象，如果该数据是新创建的，则会创建这两个表。

`createTable()` 方法的作用则是将一个新创建出来的表的相关 `shcema` 信息插入到元数据表中，具体来说，会为每张表插入一条相应的记录到 `tblcat` 表中，也会为每张表的每个字段插入一条相应的记录到 `fldcat` 表中。

方法 `getTableInfo()` 则会从 `tblcat` 和 `fldcat` 表中去检索信息，遍历得到指定表的记录长度、各字段的 `offset` 等信息，并构造一个 `TableInfo` 对象返回。（译者注：注意，如果数据库表中不存在指定表名所对应的表，方法不会返回一个 `null` 对象，而是一个内容为一些“代表空值”的 `TableInfo` 对象，具体来说就是，`schema` 为一个成员对象什么都没有的对象，`offsets` 也是一个空的 map，`recLen` 为 -1）。

## 16.3 视图的元数据

视图 是一个动态地从一个 query 得到的虚拟表。比方说，有一张 `STUDENT` 表，表里有包含学生信息的各个字段，假设现在你只想要学生的姓名和电话，通过一个 SQL 查询可以很容易得到。现在你可以将查询结果对应的结构看做一个虚拟的“表”，下次你再想查询学生的电话，你就可以直接把这个定义的视图作为新的 SQL 查询的 `WHERE` 部分。

译者注：打个不是很恰当的比方，现在有一本书，书的目录有几十章，每章又有很多节，节下面又有小节，这样目录太多了对我们也不是很方便。假设你对本书第5章的第1节的第3小节（即 5.1.3）你总是看了很多遍都看不懂，每次都一级一级找下了不太方便，于是你想了一个办法，自己定义

了一个新的目录，目录中就是有2个条目：看不懂的，页码范围为35-50页；看了总是会忘的，页码范围为67—90页。视图就好比这个新的目录，你下次只要查你自己自定义的目录就可以很快地找到想看的内容，而不需要去原始目录中一级一级找下来。

OK，视图也是靠一个SQL语句来定义的，那么这个定义视图的SQL我们称其为视图的 定义(definition)，元数据管理器会在每次新建视图时存储其定义，并且在需要的时候检索出来。

SimpleDB的类 ViewMgr 就是为此负责的类，其代码如下：

```

public class ViewMgr {
    // 视图定义的最长字符串为150字节
    private static final int VIEW_DEF_MAX_LEN = 150;

    TableMgr tableMgr;

    public ViewMgr(boolean isNew, TableMgr tableMgr, Transaction tx) {
        this.tableMgr = tableMgr;

        // 如果是新创建的数据库，则会创建一个viewcat元数据表
        if (isNew) {
            Schema schema = new Schema();
            // 视图名、表名、字段名 的最大长度都是一样的，为20个字符
            schema.addStringField("viewname", TableMgr.TABL
            schema.addStringField("viewdef", VIEW_DEF_MAX_L
            tableMgr.createTable("viewcat", schema, tx);
        }
    }

    /**
     * 将一个新视图的元数据插入viewcat表中
     *
     * @param viewName
     * @param viewDef
     * @param tx
     */
    public void createView(String viewName, String viewDef,
        TableInfo viewCatTableInfo = this.tableMgr.getTable("viewcat");

        RecordFile viewCatRecordFile = new RecordFile(viewCatTableInfo);
        viewCatRecordFile.insert(); // 找到一个插入的位置
        viewCatRecordFile.setString("viewname", viewName);
        viewCatRecordFile.setString("viewdef", viewDef);
        viewCatRecordFile.close();

    }

    /**
     * 检索某个视图的定义
     *
     * @param viewName 指定视图名
     * @param tx
     * @return
     */
    public String getViewDef(String viewName, Transaction tx) {
        String result = null;
        TableInfo viewCatTableInfo = this.tableMgr.getTable("viewcat");
        RecordFile viewCatRecordFile = new RecordFile(viewCatTableInfo);
        viewCatRecordFile.setString("viewname", viewName);
        viewCatRecordFile.set
    }
}

```

```

    // 遍历各个视图元数据
    viewCatRecordFile.beforeFirst();
    while (viewCatRecordFile.next()) {
        if (viewCatRecordFile.getString("viewname").equals(viewName)) {
            result = viewCatRecordFile.getString("viewdef");
            break;
        }
    }
    viewCatRecordFile.close();
    return result;
}
}

```

`ViewMgr` 类会把视图的元数据存储在表 `viewcat` 中，每行记录对应一个视图，表 `viewcat` 的 schema 如下（下划线是主键）：

`viewcat (ViewName , ViewDef )`

构造函数也是在系统启动的时候会被调用，如果创建的数据库是新的，那么则会创建相应的 `viewcat` 元数据表，方法 `createView()` 和 `getViewDef()` 都会使用一个 `RecordFile` 对象来访问 `viewcat` 表。注意，视图的定义我们用的是一个 `varchar` 字符串来表示，也就是说，我们把视图定义的 SQL 语句最大长度限定在一个较小的长度，我们目前限定在 150 字符。当然，这种限制有点不是很合乎真实的情况，一个视图的定义可能很长，我们可以定义成一个 `clob` 类型的字符串，例如 `clob(9999)`，不过我们的 `simpleDB` 目前只采用的是定长记录的实现策略，所以设置成太大可能会浪费较多空间。（译者注：还记得什么是 `clob` 吗？`character large object`，就是很长的文本串，一般会存放在另一个文件中）。

## 16.4 数据统计的元数据

### 16.4.1 StatInfo API

另一种数据库管理的元数据是关于每个表的数据统计信息，例如一张表中有多少条记录、每个字段值得分布情况。这些统计信息会被 query planner 使用，从而估计一次查询的大概耗时。实验依据证明，一组好的数据统计信息可以明显地改善查询的执行时间，于是，很多商业数据库的元数据管理器会维持很多更细节、更有综合性的数据统计情况，例如表中各字段的值和范围的直方图统计情况，以及不同表之前字段值的相关性，等等。

为了简单，我们只考下面的 3 种统计信息：

- 表 T 占用的块数
- 表 T 的记录条数
- 表 T 中字段 F 的不同值的数量。

我们将以上3种统计信息分别用 $B(T)$ ,  $R(T)$ ,  $V(T, F)$ 来表示。

图16-7展示了一个学生数据库的数据统计情况，这些数据来自于一个每年接受900名学生的大学，并且每年开500节课，这个数据库中维护了该大学过去50年的数据。

T	B(T)	R(T)	V(T, F)
STUDENT	4,500	45,000	45,000 for F=SIId 44,960 for F=SName 50 for F=GradYear 40 for F=MajorId
DEPT	2	40	40 for F=DId, DName
COURSE	25	500	500 for F=CIId, Title 40 for F=DeptId
SECTION	2,500	25,000	25,000 for F=SectId 500 for F=CourseId 250 for F=Prof 50 for F=YearOffered
ENROLL	50,000	1,500,000	1,500,000 for F=EId 25,000 for F=SectionId 45,000 for F=StudentId 14 for F=Grade

Figure 16-7 Example statistics about the university database

上图中的数据是真实的数据，并且假设STUDENT记录文件的一个块可以容纳10条数据，DEPT记录文件的一个块可以容纳20条数据。

我们看下STUDENT表的 $V(T, F)$ 。因为 SId 是STUDENT表的主键，所以 $V(STUDENT, SId) = 45,000$ ; 而

$V(STUDENT, SName) = 44,960$ , 表示45,000名学生中有40名重名的。;  $V(STUDENT, GradYear) = 50$ 表示最近50年以来，每一年至少有一名学生毕业。 $V(STUDENT, MajorId) = 40$ 意味着40个学院中，每个学院在某个时候至少拥有一个专业。

在SimpleDB中，我们分别用 StatInfo 类中的方法 `blocksAccessed()`, `recordsOutput()` 和 `distinctValues()` 来实现 $B(T)$ ,  $R(T)$ ,  $V(T, F)$ 。其API如下：

```
public class StatInfo {
    public int blockAccessed();
    public int recordOutputs();
    public int distinctValues(String fileName);
}
```

元数据管理器（即 `MetadataMgr` 类）的 `getStatInfo()` 方法会返回一个指定的表的StatInfo对象，例如，考虑下面的代码片段，代码的功能是获得STUDENT表的数据统计信息，并且打印出 $B(STUDENT)$ ,  $R(STUDENT)$ 和 $V(STUDENT, MajorId)$ 的值。

```

SimpleDB.init("studentdb");
MetadataMgr metadataMgr = SimpleDB.metadataMgr();

Transaction tx = new Transaction();
TableInfo tableInfo=metadataMgr.getTableInfo("dept",tx);
StatInfo statInfo=metadataMgr.getStatInfo("dept",tx);

System.out.println(statInfo.blocksAccessed() + " " +
                    statInfo.recordsOutput() + " " +
                    statInfo.distinctValues("DName"));
tx.commit();

```

## 16.4.2 实现数据统计的元数据

数据库系统可以以两种不同的方式来管理数据统计的元数据：

- 第一种是将这些信息存储在数据库的catalog中，无论何时数据库发生改变时，元数据都会更新。
- 另一种是把这些信息存储在内存中，当数据库服务器初始化时，则会更新这些元数据。

第一种方法会对应两张表，我们分别取名为 `tblstats` 和 `fldstats`，它们的schema如下：

`tblstats (TblName, NumBlocks, NumRecords)`

`fldstats(TblName, FldName, NumValues)`

`tblstats` 表将会对每张表T维护一条记录，包含的是B(T)和R(T)信息。`fldstats` 表将会对每张表中的每个字段维护一条记录，包含的是V(T, F)信息。这种方法存在的一个问题就是实时地维护数据统计信息的代价，每次调

用 `insert()`、`delete()`、`setInt()` 和 `setString()` 方法时，都将需要更新这两张表，而且，既然涉及到更新到修改持久化的信息，那必然会涉及到额外的磁盘访问、写日志等操作，并且，整个数据库系统的并发性也会下降，试想一下，每次更新表T中的记录时，都将先获得包含表T数据统计信息块的xlock（互斥锁），这样也就会迫使那些需要读取表T数据统计信息的事务等待；此外，还可能会迫使那些表T的数据统计信息在同一个块上的其他表T'也等待，这显然会降低系统的整体并发性。

解决上述问题的一个策略是让事务在读到数据统计块前，不需要获取到slock，正如14.4.7小节中的read-uncommitted级别。该策略的可能会带来数据统计信息的不准确，但是这种不准确往往是可以接受的，因为毕竟这些统计信息是用来评估许多query计划的执行用时的，因此也不必要那么准确。

而第二种实现数据统计的元数据的策略则是完全放弃catalog表，把这些信息直接存储在内存中，这些统计信息相对来说还是数据量很小的，所以存放在主存中问题不大。但该策略的问题就是，每次服务区启动的时候，这些统计信息都要再算一遍，例如统计一下一个表记录占用的文件块数、每个字段值得分布等等。如果数据库不太大的话，这些计算倒不太怎么会影响服务区启动的耗时。

这种基于主存的策略在面对数据库表更新是有两种选择：①第一种和之前说的一样，就是在每次更新完数据库后，都更新一下这些数据统计信息②第二种则是让这些数据统计信息不更新，但要每隔一段时间都要重新计算一次。第二种方法也依赖于这样一个事实，即统计信息不是那么地精确，因此可以允许在刷新统计信息之前，让这些统计信息有点儿过时。

SimpleDB采取的就是上述的第二种数据统计元数据实现策略中的第二种更新方法，类 StatMrg 维护了一个变量（名为 `tableStats`）来保存每个表的信息。该类有一个共有的方法 `statInfo()` 返回具体表的信息，私有方法 `refreshStatistics()` 和 `refreshTableStats()` 则会重新计算这些数据。具体实现如下：

```

public class StatMgr {

    // 每执行100次检索表的数据统计元数据后，刷新一次数据库统计信息
    public static final int REFRESH_EVERY_CALLS = 100;
    private Map<String, StatInfo> tableStats;
    private int numCalls;
    private TableMgr tableMgr;

    public StatMgr(TableMgr tableMgr, Transaction tx) throws
        this.tableMgr = tableMgr;
        // 新建对象时统计一次
        refreshStatistics(tx);
    }

    public synchronized StatInfo getStatInfo(String tableName)
        this.numCalls++;
        if (numCalls > REFRESH_EVERY_CALLS)
            refreshStatistics(tx);
        StatInfo statInfo = tableStats.get(tableName);

        // 如果某个客户端在刚好完成上一次所有表的数据统计后，创建了新的对象
        // 而这时所有表的数据统计信息又还没到下一次重新计算的时候，那
        // 这时，我们再显示地执行一次查询指定表的数据统计元数据操作。
        if (null == statInfo) {
            refreshTableStats(tableName, tx);
            statInfo = tableStats.get(tableName);
        }
        return statInfo;
    }

    /**
     * 更新所有表的数据统计信息
     *
     * @param tx
     * @throws IOException
     */
    public synchronized void refreshStatistics(Transaction tx)
        tableStats = new HashMap<>();
        this.numCalls = 0;
        // 先获得tblcat表，获取各表的信息
        TableInfo tblCatTableInfo = tableMgr.getTableInfo("tblcat");
        RecordFile tblCatRecordFile = new RecordFile(tblCatTableInfo);

        // 更新每张表的数据统计信息
        tblCatRecordFile.beforeFirst();
        while (tblCatRecordFile.next()) {

```

```

        String tblName = tblCatRecordFile.getString("tblName");
        // 更新具体的某张表的统计信息
        refreshTableStats(tblName, tx);
    }
    tblCatRecordFile.close();
}

/**
 * 更新指定表的数据统计信息
 *
 * @param tblName
 * @param tx
 * @throws IOException
 */
private synchronized void refreshTableStats(String tblName, Transaction tx)
{
    int numRecords = 0;
    TableInfo tableInfo = tableMgr.getTableInfo(tblName);
    RecordFile recordFile = new RecordFile(tableInfo, tx);

    recordFile.beforeFirst();
    while (recordFile.next()) {
        numRecords++;
    }
    // 这个时候, recordFile肯定遍历到了表记录文件的最后一块,
    // 因此可以求得总块号(块号是从0开始的)
    int numBlocks = recordFile.currentRID().blockNumber();
    recordFile.close();

    // 新建出表的数据统计信息, 即块数+ 记录条数
    StatInfo statInfo = new StatInfo(numBlocks, numRecords);
    this.tableStats.put(tblName, statInfo);
}

}

```

类 StatMgr 维护了一个计数器, 用来统计 statInfo() 方法的调用次数, 如果调用次数大于一个给定值 (这里是100) , 那么方法 refreshStatistics() 会被调用, 来更新所有表的数据统计信息, 从而使得query planner估计某个query更准确。注意, 当查询某张表的数据统计信息时, 我们调用了 getStatInfo() , 但有的时候, 这个方法会返回一个null, 这是为什么呢? 试想一下这样一个场景:

如果某个客户端在刚好完成上一次所有表的数据统计后, 创建了一个新的表, 而这时所有表的数据统计信息又还没到下一次重新计算的时候, 那么可能会暂时查不到结果。这时, 我们再显示地执行一次查询指定表的

数据统计元数据操作。也即再调用一次 `refreshTableStats()` 方法即可。

`refreshStatistics()` 方法会遍历 `tblcat` 表中所有的表记录，循环的中则会调用 `refreshTableStats()` 方法为每张表统计一下它的信息。而 `refreshTableStats()` 方法的流程的话，也就很简单了，遍历表中的每个字段，统计记录的条数，并且文件的块数；至于每个字段的值分布情况，我们在这里暂时没有真的去统计，相反，我们会根据表中记录的条数来模拟字段值的数量，这里的模拟策略是 1/3 的记录数，这个模拟策略应该是很糟糕的，练习 16.22 会要求你纠正这个问题。

## 16.5 索引的元数据

### 16.5.1 使用索引的元数据

索引的元数据包含索引名、索引的表名、以及索引的各字段名。索引管理器是数据库系统中负责存储和检索索引元数据的部分，索引管理器支持两个方法：当索引创建时，`createIndex()` 方法会将索引的信息存储在 `catalog` 中；当需要检索索引时，`getIndexInfo()` 方法会从 `catalog` 中检索出来。

译者注：还记得我们之前在视图中举的看书的例子吗？而索引则可以看作是书的目录（即目录到实际页的映射），而视图则是新建的那张目录。

在 SimpleDB 中，`getIndexInfo()` 方法会返回一个包含 `IndexInfo` 对象的 map，map 的 key 是索引的字段。一个 `IndexInfo` 对象维护的是单个索引的元数据，其 API 如下：

```
public class IndexInfo {
    public IndexInfo(String idxName, String tblName, Sti
    public int blockAccessed();
    public int recordsOutput();
    public int distinctValues(String fieldName);
    public Index open();
}
```

构造函数会接受索引的元数据信息，SimpleDB 中的索引只能索引一个字段，这也就是为什么构造函数的第 3 个参数是一个 `String`，而不是 `list` 的原因。方

法 `blockAccessed()`、`recordsOutput()` 和 `distinctValues()` 则会给出索引的数据统计信息，和 `StatInfo` 类类似。方

法 `blockAccessed()` 返回的是搜索索引而需要访问的文件块数，方

法 `recordsOutput()` 和 `distinctValues()` 分别返回的是索引中记录的条数和索引字段所有可能取值的数量。

一个 `IndexInfo` 对象也有一个 `open()` 方法，该方法使用这些元数据去打开一个索引，返回一个 `Index` 对象，`Index` 对象包含搜索索引的方法，我们将在第21章中讨论。

下面的代码演示了索引管理器如何使用，代码的功能是返回STUDENT表中的所有索引信息，然后遍历所有，打印出索引名和搜索的带来，最后，打开MajorID这个索引。

### 16.5.2 实现索引的元数据

`IndexInfo` 类的实现如下：

```

public class IndexInfo {
    private String idxName;
    private String fieldName;
    private Transaction tx;
    private TableInfo tableInfo;
    private StatInfo statInfo;

    /**
     * 创建一个索引
     *
     * @param idxName 索引名
     * @param tblName 被索引的表名
     * @param fieldName 被索引的字段名，当前只支持单字段索引
     * @param tx
     */
    public IndexInfo(String idxName, String tblName, String fieldName, Transaction tx) {
        this.idxName = idxName;
        this.fieldName = fieldName;
        this.tx = tx;
        tableInfo = SimpleDB.metadataMgr().getTableInfo(tblName);
        statInfo = SimpleDB.metadataMgr().getStatInfo(tblName);
    }

    /**
     * 搜索索引所需访问的块数。
     * <p>
     * 注意，在SimpleDB中只支持一个字段的索引
     * 不像在表记录文件中，索引不需要一个 EMPTY/INUSE flag
     *
     * @return
     */
    public int blocksAccessed() {
        // 被索引的记录，每条的长度
        int recordLen = tableInfo.recordLength();
        // 一个块可以存储多少条索引
        int recordNumPerBlock = BLOCK_SIZE / recordLen;
        // 总块数
        int numBlocks = statInfo.recordsOutput() / recordNumPerBlock;

        return BTREEINDEX.searchCost(numBlocks, recordNumPerBlock);
    }

    public int recordsOutput() {
        return statInfo.recordsOutput()
            / statInfo.distinctValues(fieldName);
    }

    public int distinctValues(String fieldName) {
}

```

```

        if (fieldName.equals(this.fieldName))
            return 1;
        else
            return Math.min(statInfo.distinctValues(fieldName),
                           1);
    }

    public Index open() {
        Schema sch = schema();
        return new BTreeIndex(idxName, sch, tx);
    }

    /**
     * 构造索引的schema, 如下:
     * <p>
     * (block, id, dataval)
     *
     * @return
     */
    private Schema schema() {
        Schema sch = new Schema();
        sch.addField("block");
        sch.addField("id");
        // 索引字段是 int 类型字段
        if (tableInfo.schema().type(fieldName) == Schema.INT_TYPE)
            sch.addField("dataval");
        // 索引字段是 string 类型字段
        else {
            int fieldLen = tableInfo.schema().length(fieldName);
            sch.addField("dataval", fieldLen);
        }
        return sch;
    }
}

```

`IndexInfo` 的构造函数接受索引名、被索引的表名、以及被索引的字段名。除了这些显式的元数据，一个 `IndexInfo` 对象也会持有被索引表的 `TableInfo` 对象和数据统计对象 `StatInfo`，这些元数据允许 `IndexInfo` 对象来构造索引的schema，并且估计索引文件的大小。

方法 `open()` 的功能是打开一个索引，传入索引名和该索引的schema，构造一个 `BTreeIndex` 对象，我们将会在第21章中讨论用B树来创建索引的内容。方法 `blocksAccessed()` 估计了索引的搜索代价，它首先会使用被索引表的 `TableInfo` 信息来判断每条被索引的记录的长度，并估计一个块中可以存放多少条索引记录(records per block, RPB)和索引文件的大小，并随后调用 `BTreeIndex.searchCost()` 来计算对于该类索引需要

的块访问次数。方法 `recordsOutput()` 方法则会估计每个search key 对映的索引记录数，方法 `distinctValues()` 返回的结果和被索引表的结果一样。

SimpleDB中的 `IndexMgr` 类实现了索引管理器。在simpleDB中，索引管理器会把信息存储在catalog表 `idxcat` 中，该表对每个索引都会包含一条相应的记录，并且包含3个字段：索引名、被索引的表名、和被索引的字段名。代码如下：

```

public class IndexMgr {

    private TableInfo tableInfo;

    /**
     * 创建IndexMgr对象。
     * <p>
     * IndexMgr会把索引的信息存储在表idxcat中，该表的schema如下：
     * (indexname, tablename, fieldname)
     * <p>
     * 分别代表索引名、被索引的表名、被索引的字段名
     *
     * @param isNew
     * @param tableMgr
     * @param tx
     */
    public IndexMgr(boolean isNew, TableMgr tableMgr, Transaction tx) {
        if (isNew) {
            Schema schema = new Schema();
            schema.addStringField("indexname", TABLE_AND_FIELDNAME);
            schema.addStringField("tablename", TABLE_AND_FIELDNAME);
            schema.addStringField("fieldname", TABLE_AND_FIELDNAME);
            tableMgr.createTable("idxcat", schema, tx);
        }
        tableInfo = tableMgr.getTableInfo("idxcat", tx);
    }

    /**
     * 创建一个索引
     *
     * @param idxName 索引名
     * @param tblName 被索引的表名
     * @param fldName 被索引的字段名
     * @param tx
     * @throws IOException
     */
    public void createIndex(String idxName, String tblName,
                           String fldName, Transaction tx) {
        RecordFile idxCatRecordFile = new RecordFile(tableInfo);
        idxCatRecordFile.insert(); // 找到一个插入位置
        idxCatRecordFile.setString("indexname", idxName);
        idxCatRecordFile.setString("tablename", tblName);
        idxCatRecordFile.setString("fieldname", fldName);
        idxCatRecordFile.close();
    }

}

```

```

    * 获取到指定表的所有索引信息
    *
    * @param tblName 被索引的表
    * @param tx
    * @return
    */
    public Map<String, IndexInfo> getIndexInfo(String tableName) {
        Map<String, IndexInfo> result = new HashMap<>();
        RecordFile idxCatRecordFile = new RecordFile(tableName);
        idxCatRecordFile.beforeFirst();
        while (idxCatRecordFile.next()) {
            if (idxCatRecordFile.getString("tablename").equals(tableName)) {
                String idxName = idxCatRecordFile.getString("idxname");
                String fldName = idxCatRecordFile.getString("fldname");

                IndexInfo indexInfo = new IndexInfo(idxName, fldName);
                result.put(tableName, indexInfo);
            }
        }
        idxCatRecordFile.close();

        return result;
    }
}

```

## 16.6 实现元数据管理器

SimpleDB中的元数据管理器是通过4种元数据管理器类 TableMgr , ViewMgr , StatMgr , IndexMgr 来实现的。 MetadataMgr 类隐藏了这种区别，因此客户端只可以从一个地方来获取到元数据，这就是在我们平时说到的 门面设计模式(facade pattern) 。它的构造函数创建了四个私有的不同的元数据管理器对象，它的各个方法只不过是复制了一下单个管理器的公共方法，当到时候客户端通过 MetadataMgr 对象来调用相关方法时，将调用适当的私有管理器来完成工作。代码如下：

```

public class MetadataMgr {
    private static TableMgr tableMgr;
    private static ViewMgr viewMgr;
    private static StatMgr statMgr;
    private static IndexMgr indexMgr;

    public MetadataMgr(boolean isNew, Transaction tx) throws
        tableMgr = new TableMgr(isNew, tx);
        viewMgr = new ViewMgr(isNew, tableMgr, tx);
        statMgr = new StatMgr(tableMgr, tx);
        indexMgr = new IndexMgr(isNew, tableMgr, tx);

    }

    public void createTable(String tblName, Schema schema,
        tableMgr.createTable(tblName, schema, tx);
    }

    public TableInfo getTableInfo(String tblName, Transaction tx)
        return tableMgr.getTableInfo(tblName, tx);
    }

    public void createView(String viewName, String viewDef,
        viewMgr.createView(viewName, viewDef, tx);
    }

    public String getViewDef(String viewName, Transaction tx)
        return viewMgr.getViewDef(viewName, tx);
    }

    public void createIndex(String indexName, String tblName,
        String fieldName, Transaction tx)
        indexMgr.createIndex(indexName, tableName, fieldName, tx);
    }

    public Map<String, IndexInfo> getIndexInfo(String tableName,
        return indexMgr.getIndexInfo(tableName, tx);
    }

    public StatInfo getStatInfo(String tableName, Transaction tx)
        return statMgr.getStatInfo(tableName, tx);
    }
}

```

## 16.7 章末总结

- 元数据是关于数据库的信息，但是不包括表中的具体数据内容。

- 元数据管理器负责存储和检测数据库的元数据。
- SimpleDB实现了4类元数据：
  1. 表的元数据，描述的是某张表的信息，例如该表每条记录的长度，每个字段的偏移量，类型等等。
  2. 视图的元数据，描述的是每个视图的属性，例如视图的名称和定义。
  3. 索引的元数据，描述的是每个索引的信息，包含索引名、被索引的表名、被索引的字段名。
  4. 数据统计的元数据，描述的是每张表的占用块数，已经各字段的值分布情况。
- 元数据管理器会把元数据存放在系统的catalog中，其实也就是几张表，这些表和普通表一样，也可以被查询。
- 表的元数据被存放在两张catalog表中，一张来存放表粒度的信息（例如记录长度），另一种表来存放表中各字段信息（例如字段名、字段长度、字段的类型）。
- 视图元数据也被存放在一张catalog表中，在simpleDB中，视图的定义最长字符数我们限定成了150。
- 数据统计元数据描述的是每张表的占用块数，已经各字段的值分布情况。很多商业数据库的元数据管理器会维持很多更细节、更有综合性的数据统计情况，例如表中各字段的值和范围的直方图统计情况，以及不同表之前字段值的相关性，等等。
- 最基本的数据统计信息包含下面这3个功能：
  1. 表T占用的块数
  2. 表T的记录条数
  3. 表T中字段F的不同值的数量。
- 数据统计元数据也可以存放在catalog表中，或者可以每次在系统启动时重新计算出。第一种方法避免了因为统计时间过长而影响系统启动时间，但是第一种方法可能会降低系统的并发性。
- 索引元数据，描述的是每个索引的信息，包含索引名、被索引的表名、被索引的字段名。

## 16.8 建议阅读

## 16.9 练习

## 第17章 查询处理和包 simpledb.query

译者注：这一章中有些名词不是很好翻译，例如scan，对于这些不好翻译的词，将直接采用英文。

在前面的两章中，我们已经看到了表中的记录是怎么用记录文件来实现的，也知道了表的元数据是怎么被组织起来的，从而方便以后的记录访问。接下来的三章，我们将会考虑一下数据库系统怎么支持对表的查询。

第4章中介绍了两种查询语言：关系代数(*relational algebra*) 和 SQL。关系代数相对来说比较容易实现，因为每个操作都是一个很小的、且定义好的小任务，然而SQL比较难直接去实现，因为一个SQL查询可以嵌入许多的任务和逻辑。事实上，我们在第4章中已经看到了，一个SQL查询语句可以被表达成一个关系代数查询树，这种对应关系提供了实现SQL的关键—数据库系统首先将SQL查询转换为关系代数，然后执行关系代数查询树。

在这章中，我们将研究一下关于如何执行关系代数的问题，而后续的两章将研究如何将SQL翻译成关系代数。

### 17.1 Scans

一个 Scan 就是代表一个关系代数查询的对象，Scan 在SimpleDB中是通过接口 Scan 来实现的，接口 Scan 的方法和 RecordFile 类中的方法很相似，客户端可以遍历一个scan，移动到下一条记录，并且检索相应的字段的值。代码如下：

```

public interface Scan {
    // 移动到第一条记录之前
    public void beforeFirst();

    // 移动到下一条记录, 如果没有下一条记录, 返回false
    public boolean next();

    // 关闭scan, 如果有subScan, 也会相应关闭
    public void close();

    // 获取当前记录指定字段的值, 被抽象成了一个Constant对象
    public Constant getVal(String fieldName);

    // 获取当前字段指定int字段的值
    public int getInt(String fieldName);

    // 获取当前字段指定string字段的值
    public String getString(String fieldName);

    // 判断是否有指定字段
    public boolean hasField(String fieldName);

}

```

下面的代码演示了如何使用一个scan, 方法 `printNameAndGradYear()` 会遍历输入的scan, 打印出每条记录的字段 `sname` 和 `gradyear` 取值。因为 `Scan` 是一个接口, 这个方法其实并不知道这个scan是怎么构造的或者说这个scan代表的具体是哪种查询, 凡是输出包含学生名字和毕业年份的查询, 其实都是可以成功执行的。

```

public static void printNameAndGradyear(Scan s) {
    while (s.next()) {
        String sname = s.getString("sname");
        String gradyr = s.getInt("gradyear");
        System.out.println(sname + "\t" + gradyr);
    }
    s.close();
}

```

`Scan` 接口和 `RecordFile` 类主要在下面的3个方面有所不同:

- `RecordFile` 类中包含读/写的方法, 而 `Scan` 中的方法只支持读, 只有Update Scan才有写方法, 我们将在17.2小节中讨论。
- `RecordFile` 类需要知道表的schema信息(`RecordFile` 类的构造函数需要提供一个 `TableInfo` 对象), 但是在 `Scan` 中不需要。相

反，在 Scan 中有一个 `hasField()` 方法来判断指定的字段是否被包含在 scan 的输出中。

- Scan 中除了 `getInt()` 和 `getString()` 之外还包含一个 `getVal()` 方法，其返回的是一个 Constant 类型的对象(详情见 17.7 小节)，这个类型是对 int 和 string 的抽象，这个方法的目的就是允许查询处理器在无需知道具体类型，就可以对字段的值进行比较。

每个 Scan 对象对应一个查询树中的一个节点，对于每一种关系代数中的操作，都对应有一个具体的 Scan 接口实现类，而对于查询树中的叶子节点，必定是对一个表的 scan，下面的代码中展示了 SimpleDB 中支持的 4 种不同的 Scan 以及它们的构造函数：

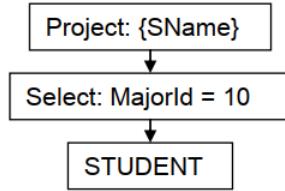
```
public SelectScan(Scan s, Predicate pred);
public ProjectScan(Scan s, Collection<String> fldlist);
public ProductScan(Scan s1, Scan s2);
public TableScan(TableInfo ti, Transaction tx);
```

考虑一下 `SelectScan` 类的构造函数，它接受两个输入，一个 Scan 对象和一个谓词对象（译者注：这里所的谓词，指的是离散数学中的谓词的概念，如果你有学过，可能会觉得很耳熟）。Scan 对象表示的是输入给 select 运算的表。回想一下，关系运算的输入可以是任何表或中间查询的结果，这刚好用一个类型为 Scan 的对象来表示。由于 Scan 是一个接口，因此，无论具体是一个实际存储的表还是其他查询的结果，`SelectScan` 对象可以对输入的表做同样的操作。

第二个谓词对象是一个 `Predicate` 类型的对象，我们将在 17.7 小节中讨论 SimpleDB 是怎么实现 select 运算中的 select 谓词的，在那之前，我们对谓词的讨论都会比较含糊，不过你可以暂时理解成一个高层次的抽象的概念，即对 scan 的结果进行比较筛选。

一个查询树可以用许多 Scan 对象来表示，树上的每个节点代表一个 Scan 对象。

图 17-4 和图 17-5 给出了两个示例，演示了查询树是怎么用 scan 对象组织起来的。



(a) A query tree to retrieve the names of students having major #10

```

SimpleDB.init("studentdb");
Transaction tx = new Transaction();

// the STUDENT node
TableInfo ti = SimpleDB.mdmgr().getTableInfo("student", tx);
Scan s1 = new TableScan(ti, tx);

// the Select node
Predicate pred = new Predicate(...); //majorid=10
Scan s2 = new SelectScan(s1, pred);

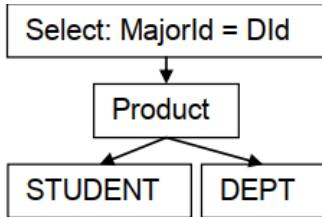
// the Project node
Collection<String> c = Arrays.asList("sname");
Scan s3 = new ProjectScan(s2, c);

while (s3.next())
    System.out.println(s3.getString("sname"));
s3.close();
  
```

(b) The SimpleDB code corresponding to the tree

Figure 17-4: Representing a query tree as a scan

图17-4 (a) 展示了一个查询树的情况，该查询的功能是返回所有专业id为10的学生姓名，图17-4 (b) 则给出了构造这个查询树的具体代码（暂时忽略select谓词的具体实现）。其中Scan类型的变量s1、s2和s3分别对应查询树中的一个节点，并且这个树是自底向上构建的（也就是说，s1对应的是STUDENT结点，s2对应的是Select结点，s3对应的是Project结点），s3是整个查询树的入口，遍历s3，则可以打印出每个专业id为10的学生姓名。



(a) A query tree to join the STUDENT and DEPT tables

```

SimpleDB.init("studentdb");
Transaction tx = new Transaction();
MetadataMgr mdMgr = SimpleDB.mdg();

// the STUDENT node
TableInfo sti = mdMgr.getTableInfo("student", tx);
Scan s1 = new TableScan(sti, tx);

// the DEPT node
TableInfo dti = mdMgr.getTableInfo("dept", tx);
Scan s2 = new TableScan(dti, tx);

// the Product node
Scan s3 = new ProductScan(s1, s2);

// the Select node
Predicate pred = new Predicate(..); //majorid=did
Scan s4 = new SelectScan(s3, pred);

while (s4.next())
    System.out.println(s4.getString("sname")
        + ", " + s4.getString("gradyear")
        + ", " + s4.getString("dname") );
s4.close();
  
```

(b) The SimpleDB code corresponding to the tree

Figure 17-5: Representing another query tree as a scan

图17-5 (a) 则展示了另一个查询树的情况，该查询的功能是返回所有的学生信息以及对应的学院。图17-5 (b) 给出了构造这个查询树的具体代码，和图17-4中的例子很相似。在这个例子中，代码中包含4个scan，因为这个查询树有4个节点，变量s4是整个查询树的入口，然后也和之前一样遍历打印结果。在这里为了节约篇幅，只打印了3个字段，如果你想的话，肯定也是可以打印出两张表join之后的所有字段信息的。

## 17.2 Update scan

一个query定义了一张虚拟表，而Scan接口给客户端提供了读取这些虚拟表的相关方法，但是无法修改虚拟表。在4.5.3小节中，我们已经讨论了更新一个虚拟表的问题，并且得出了这样的结论：只有对于那些可更新的查询 (updatable query) 所创建的虚拟表，更新虚拟表才是有意义的。这个结论在这里同样也成立。

当一个scan中的每条记录 $r$ 都在某个数据库表中存在一条对应的 $r'$ 记录时，我们称这个scan是可更新的。可更新的scan是通过接口 `UpdateScan` 来实现的，代码如下所示：

```
public interface UpdateScan extends Scan {
    public void setVal(String fieldName, Constant newVal);
    public void setInt(String fieldName, int newVal);
    public void setString(String fieldName, String newVal);
    public void insert();
    public void delete();

    public RID getRID();
    public void moveToRID(RID rid);
}
```

前5个方法对应的一些修改操作，而最后两个方法运行客户端获取到当前记录的rid或者将当前记录移动到指定rid对应的记录。（注意，rid对于非可更新的scan是没什么意义的，因为一个非可更新的scan中的一条记录可能对应的是底层多张表中的多个记录）

SimpleDB中实现了 `UpdateScan` 接口的两个类是 `TableScan` 和 `SelectScan`，至于如何使用它们，请看下面图17-7 中的例子，图17-7 (a) 中给出了一个SQL语句，该SQL语句的功能是把id为53的那堂课中的所有学生成绩全部改成C，图17-7 (b) 中给出了实现这个SQL的代码，代码首先创建了一个tableScan对象，随后创建了一个SelectScan对象，最后遍历该SelectScan对象，遍历的过程中会修改每个学生的成绩为C。

```
update ENROLL
set Grade = 'C'
where SectionId = 53
```

(a) An SQL statement to modify the grades of students in section #53

```
SimpleDB.init("studentdb");
Transaction tx = new Transaction();

TableInfo ti = SimpleDB.mdmgr().getTableInfo("student", tx);
Scan s1 = new TableScan(ti, tx);

Predicate pred = new Predicate(...); //SectionId=53
UpdateScan s2 = new SelectScan(s1, pred);

while (s2.next())
    s2.setString("grade", "C");
s2.close();
```

(b) The SimpleDB code corresponding to the update

Figure 17-7: Representing an SQL update statement as an update scan

注意，变量s2必须被声明成一个update scan，因为遍历的时候回调用 `setString()` 方法。此外，`SelectScan` 构造函数的第一个参数是 `Scan` 类型的，也就是说，`s1`不需要被声明为一个update scan；相反，因为有 `s2.setString()` 方法的存在，因此会将`s1`强转为一个 update scan，如果这个`s1`实际上不是一个实现了`UpdateScan`的对象，那么会抛出一个 `ClassCastException` 异常。

## 17.3 实现Scan接口

SimpleDB中包含4个实现了Scan接口的类，分别是：`TableScan`、`SelectScan`、`ProjectScan`和 `ProductScan`。下面我们将以此来讨论一下关于这些类的实现细节。

### 17.3.1 Table Scan

`TableScan` 类的代码如下所示，`TableScan`是可更新的scan，因此实现的是 `UpdateScan` 接口(`UpdateScan` 接口继承了 `Scan` 接口)。

```
public class TableScan implements UpdateScan {  
    private RecordFile recordFile;  
    private Schema schema;  
  
    public TableScan(TableInfo tableInfo, Transaction tx) {  
        recordFile = new RecordFile(tableInfo, tx);  
        schema = tableInfo.schema();  
    }  
    //=====Scan 接口中的方法实现=====  
    @Override  
    public void beforeFirst() {  
        recordFile.beforeFirst();  
    }  
  
    @Override  
    public boolean next() throws IOException {  
        return recordFile.next();  
    }  
  
    @Override  
    public void close() {  
        recordFile.close();  
    }  
  
    @Override  
    public Constant getVal(String fieldName) {  
        if (schema.type(fieldName)==Schema.VARCHAR)  
            return new StringConstant(recordFile.getString(fieldName));  
        else  
            return new IntConstant(recordFile.getInt(fieldName));  
    }  
  
    @Override  
    public int getInt(String fieldName) {  
        return recordFile.getInt(fieldName);  
    }  
  
    @Override  
    public String getString(String fieldName) {  
        return recordFile.getString(fieldName);  
    }  
  
    @Override  
    public boolean hasField(String fieldName) {  
        return schema.hasFiled(fieldName);  
    }  
    //=====UpdateScan 接口中额外的方法实现=====
```

```

@Override
public void setVal(String fieldName, Constant newVal) {
    if (schema.type(fieldName)==Schema.VARCHAR)
        recordFile.setString(fieldName,(String) newVal);
    else
        recordFile.setInt(fieldName,(Integer)newVal.asList().get(0));
}

@Override
public void setInt(String fieldName, int newVal) {
    recordFile.setInt(fieldName,newVal);
}

@Override
public void setString(String fieldName, String newVal)
    recordFile.setString(fieldName,newVal);
}

@Override
public void insert() throws IOException {
    recordFile.insert();
}

@Override
public void delete() {
    recordFile.delete();
}

@Override
public RID getRID() {
    return recordFile.currentRID();
}

@Override
public void moveToRID(RID rid) {
    recordFile.moveToRID(rid);
}
}

```

代码相对来说还是很直观的，构造函数会根据传入的 TableInfo 对象创造这张 RecordFile 对象，方法 getVal() 则会基于具体的字段，创建合适的 Constant 对象，方法 setVal() 则刚好执行相反的操作。方法 hasField() 则会查看一下表的 schema，其他的方法则都是通过 RecordFile 对象提供的方法来完成的。

### 17.3.2 Select Scan

SelectScan 类的代码如下，构造函数也会传入一个实现了Scan接口的对象，该对象可以是一个输入表（即TableScan），一个SelectScan对象的当前记录就是构造函数中传入的Scan对象的当前记录，这也就是说，SelectScan中许多方法的实现都可以通过直接调用传入Scan对象的相关方法来完成。

```
public class SelectScan implements UpdateScan {

    private Scan scan;
    private Predicate predicate;

    public SelectScan(Scan scan, Predicate predicate) {
        this.scan = scan;
        this.predicate = predicate;
    }

    //=====Scan 接口中的方法实现=====

    @Override
    public void beforeFirst() {
        scan.beforeFirst();
    }

    @Override
    public boolean next() throws IOException {
        while (scan.next())
            if (predicate.isSatisfied(scan))
                return true;
        return false;
    }

    @Override
    public void close() {
        scan.close();
    }

    @Override
    public Constant getVal(String fieldName) {
        return scan.getVal(fieldName);
    }

    @Override
    public int getInt(String fieldName) {
        return scan.getInt(fieldName);
    }

    @Override
    public String getString(String fieldName) {
        return scan.getString(fieldName);
    }

    @Override
    public boolean hasField(String fieldName) {
        return scan.hasField(fieldName);
    }
}
```

```
}

//=====UpdateScan 接口中额外的方法实现=====

@Override
public void setVal(String fieldName, Constant newVal) {
    // 这里必须把scan强转为UpdateScan类型
    // 也只有UpdateScan类型的对象才有setXXX()方法
    // 如果scan不是一个实现了UpdateScan接口的对象，运行时则会报错
    UpdateScan updateScan = (UpdateScan) scan;
    updateScan.setVal(fieldName,newVal);
}

@Override
public void setInt(String fieldName, int newVal) {
    UpdateScan updateScan = (UpdateScan) scan;
    updateScan.setInt(fieldName, newVal);
}

@Override
public void setString(String fieldName, String newVal)
    UpdateScan updateScan = (UpdateScan) scan;
    updateScan.setString(fieldName, newVal);
}

@Override
public void insert() throws IOException {
    UpdateScan updateScan = (UpdateScan) scan;
    updateScan.insert();
}

@Override
public void delete() {
    UpdateScan updateScan = (UpdateScan) scan;
    updateScan.delete();
}

@Override
public RID getRID() {
    UpdateScan updateScan = (UpdateScan) scan;
    return updateScan.getRID();
}

@Override
public void moveToRID(RID rid) {
    UpdateScan updateScan = (UpdateScan) scan;
    updateScan.moveToRID(rid);
```

```
    }  
}
```

唯一一个需要特别注意的地方就是 `next()` 方法， `next()` 方法的作用实际上是判断是否存在下一条记录， 而在这里我们又要加上对记录的谓词判断。在代码中，我们必须使用 `while(scan.next())` 而不是 `if(scan.next())`， 因为， 假如当前记录后面还存在3条记录r1、r2和r3， 而r1又不满足预定义的谓词，在`SelectScan`中的 `next()` 方法的功能是判断是否存在满足谓词的下一条记录， 因此必须找到r2并将之设置为当前记录， 然后返回true。

`Select scan`是`updatable`的，所以在 `SelectScan` 中有关 `UpdateScan` 接口中的方法，都会先将`scan`强转为 `UpdateScan` 类型的，然后再调用相关的方法。在`SimpleDB`中，`query planner`（后续章节中的内容）只会对`table scan`和`select scan`创建`updatable scan`对象，所以，一旦发现抛出了 `ClassCastException` 异常，肯定是代码中出现了 bug。

### 17.3.3 Project scan

`ProjectScan` 类的实现如下，想要输出的字段列表会通过构造函数传入，随后会判断这些字段是否存在，其他方法的实现也很直接，只要调用 `scan`对象的相关方法即可。

```
public class ProjectScan implements Scan {
    private Scan scan;
    private Collection<String> fieldList;

    public ProjectScan(Scan scan, Collection<String> fieldList) {
        this.scan = scan;
        this.fieldList = fieldList;
    }

    // =====Scan 接口中相关方法的实现=====

    @Override
    public void beforeFirst() {
        scan.beforeFirst();
    }

    @Override
    public boolean next() throws IOException {
        return scan.next();
    }

    @Override
    public void close() {
        scan.close();
    }

    @Override
    public Constant getVal(String fieldName) {
        if (hasField(fieldName))
            return scan.getVal(fieldName);
        else
            throw new RuntimeException("Field "+fieldName+" not found");
    }

    @Override
    public int getInt(String fieldName) {
        if (hasField(fieldName))
            return scan.getInt(fieldName);
        else
            throw new RuntimeException("Field "+fieldName+" not found");
    }

    @Override
    public String getString(String fieldName) {
        if (hasField(fieldName))
            return scan.getString(fieldName);
        else
            throw new RuntimeException("Field "+fieldName+" not found");
    }
}
```

```
}

@Override
public boolean hasField(String fieldName) {
    return fieldList.contains(fieldName);
}
}
```

注意，在SimpleDB中Project Scan被设计成了不可更新的（实际上，是可以设计成可更新的），因此也没必要也不能去实现 UpdateScan 接口，练习17.16有相关的讨论。

### 17.3.4 Product scan

ProductScan 类的实现代码如下：

```
public class ProductScan implements Scan {
    private Scan scan1,scan2;

    public ProductScan(Scan scan1, Scan scan2) throws IOException {
        this.scan1 = scan1;
        this.scan2 = scan2;
        scan1.next();
    }

    // =====Scan 接口中相关方法的实现=====

    @Override
    public void beforeFirst() throws IOException {
        scan1.beforeFirst();
        scan1.next();
        scan2.beforeFirst();
    }

    @Override
    public boolean next() throws IOException {
        if (scan2.next())
            return true;
        else {
            scan2.beforeFirst();
            return scan1.next() && scan2.next();
        }
    }

    @Override
    public void close() throws IOException {
        scan1.next();
        scan2.next();
    }

    @Override
    public Constant getVal(String fieldName) {
        if(scan1.hasField(fieldName))
            return scan1.getVal(fieldName);
        else
            return scan2.getVal(fieldName);
    }

    @Override
    public int getInt(String fieldName) {
        if(scan1.hasField(fieldName))
            return scan1.getInt(fieldName);
        else
            return scan2.getInt(fieldName);
    }
}
```

```

    }

    @Override
    public String getString(String fieldName) {
        if(scan1.hasField(fieldName))
            return scan1.getString(fieldName);
        else
            return scan2.getString(fieldName);
    }

    @Override
    public boolean hasField(String fieldName) {
        return scan1.hasField(fieldName) || scan2.hasField(fieldName);
    }
}

```

product操作对应的是大多数中文数据库书中的联结操作，也就是对两个表中的记录求一个笛卡尔积。在代码中，我们先开始遍历scan1中的所有记录作为笛卡尔积的左侧元素，然后再遍历scan2中所有的记录作为笛卡尔积的右侧元素；也就是类似一个嵌套的循环，scan1为外层循环，scan2为内层性循环。

`next()`方法是这样实现这个嵌套循环的：每次都调用内层scan2循环的`next()`，如果scan2有下一条记录，则返回true；如果scan2没有下一条记录，则肯定是内层循环到了底，此时必须调用外层循环的`next()`方法，于是返回scan1的下一条记录，并且把scan2移动到第一条记录，然后返回true；否则其他情况，返回false。

`getXXX()`方法则很简单了，只不过多了一个判断指定字段是来自于哪个scan的操作。

## 17.4 流水线式的查询处理

我们已经知道了simpleDB中几种基本的关系代数操作是如何实现的，实际上，这些实现被称为 流水线式的(pipelined)，它们有下面的两个特点。

- 按照客户端的需要，每次只会输出一条记录
- 实现中不会保存输出记录，也不会保存任何中间的计算结果。

我们在这里只分析 `TableScan` 和 `SelectScan` 的特点，`ProductScan` 和 `ProjectScan` 的分析则类似。

假设有一个 `TableScan` 类型的对象，它的实现中是维护了一个`record file`的，这个`record file`又是维护了一个`record page`的，这个`record page`又是持有了一个`buffer`的，`buffer`中又持有一个页，而这个页中包含的是对应块中的实际内容的。当前记录只是在这个页中的一个位置，每当客户端

请求访问一个字段的具体取值时，记录管理器只会提取出这个字段的值而已，随后返回这个值给客户端；而每次调用 `next()` 方法后则会将当前记录移动至下一条，这个操作有可能会引发一次缓冲区的换入换出（swapping）。

现在我们继续考虑有一个 `SelectScan` 类型的对象，每次调用该对象的 `next()` 方法实际上调用的是一个 `underlying` 的 `Scan` 类型对象的 `next()` 方法，直到找到一个满足谓词的记录。但是实际上，这个 `SelectScan` 类型的对象根本没有什么“当前记录”——如果这个对象的 `underlying` 的 `Scan` 类型对象是一个 `Table Scan`，那么这个当前记录只不过是在页中的一个位置而已了，对吧？也就是说，根本没有一个实际的变量保存了当前记录所有值。每次 `select scan` 调用 `next()` 方法，实际上都是从上一次的位置往下搜索的。

最后，很重要的一点，一个 `select scan` 不会追踪已经选择了的记录，也就是说，如果一个客户端需要连续 2 次请求同一个记录，`select scan` 必须遍历两次。

术语“流水线”代表的是查询树自顶向下的方法调用流，以及自底向上的结果返回流。举例来说，假如查询树中的某个结点调用了 `getInt()` 方法，那么这个结点会一直往下调用它的子结点的方法，直到访问到了叶子节点，叶子节点随后将提取出来的值又一直往它的父结点抛上去。在比如，假如调用 `next()` 方法，它可能会调用它的子节点的 `next()` 方法多次直到找到了满足条件的下一条记录，这个例子其实就是 `select scan` 对 `table scan` 的调用，随后也会同样地返回结果。

流水线式的实现是非常高效的，例如，请看图 17-12 中的查询树的例子，我们可以发现，`project` 结点和 `select` 结点的 `cost` 实际上是 0。

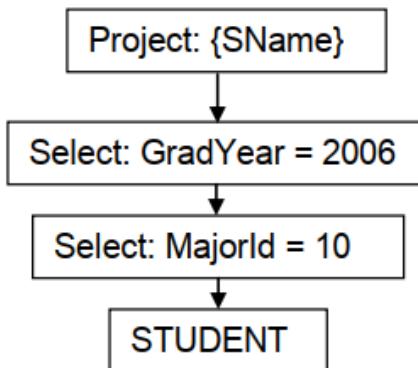


Figure 17-12: A query tree containing multiple `select` nodes

方法调用流程是这样的：

- 我们先考虑最顶层的 `project` 结点，每次调用 `next()` 方法，都会简单地调用第一个 `select` 结点（也就是 `Select: GradYear=2006` 结点）的 `next()` 方法；

- 第一个select结点又会继续往下调用第二个select结点的 `next()` 方法；
- 第二个select结点又会继续往下调用table scan结点的 `next()` 方法。

结果返回的流程又是这样的：

- table scan结点会遍历record file，判断是否存在下一条记录，并返回给第二个select结点（返回的是布尔值）；
- 如果子节点抛来的布尔值是true，则第二个select结点会判断当前这条记录是否满足谓词，随后把判断结果也往上抛；
- 如果第二个select结点往上抛出的布尔值是false，那肯定第一个select结点也无需再去做谓词判断了，随后继续把这个值抛给父结点；
- project结点也肯定会接受到这个false值，又重新执行下一轮调用流程了。

这么一分析，project结点和select结点的cost真的是0，只有叶子结点会执行具体的底层遍历逻辑。虽然说，这种流水线的实现方式在某些场合下很高效，但是在某些场合下也不是那么好（译者注：常言道，Every coin has two sides，但是既然有两面，那肯定有Edge，我们站在Edge上同时看两个sides就好了，对吧？）。这种不太好的场合的一个例子是，当一个select scan结点是在一个product scan结点的右边时，会执行很多轮的方法调用（很难用语言描述，看下代码分析一下就可以）。在这种情况下，为了避免一遍又一遍地遍历，选用一种可以将输出记录具体化（materialize）到一个临时表中的实现会更好。这中实现将在第22章中讨论。

## 17.5 估计Scan的代价

数据库系统的一个职责是对于一个给定的SQL查询，创建一个最高效的scan。于是系统必须去估计一个scan的执行用时大概是多少。由于一个scan所需执行的文件块访问数，是评估执行代价的最关键因素，因此估计所需的块访问次数就足够了。在本节中，我们将考虑如何估算一个scan的代价，而在第24章中，我们将了解数据库系统如何利用这些信息。

一个scan的代价可以用类似16.4小节中的数据统计信息术语来评估：

假设有某个scan，记为s，并记s中的某个字段为F。 $B(s)$ 表示构造s的输出所需的块访问数量。 $R(s)$ 表示s输出的记录条数。 $V(s, F)$ 表示s输出的所有记录中，F字段的所有可能取值数。

图17-13包含了每个关系代数操作的代价公式，这些公式的推导我们将一道来。

$s$	$B(s)$	$R(s)$	$V(s, F)$
TableScan( $T$ )	$B(T)$	$R(T)$	$V(T, F)$
SelectScan( $s_1, A = c$ )	$B(s_1)$	$R(s_1) / V(s_1, A)$	$\begin{cases} 1 & \text{if } F = A \\ \min\{R(s), V(s_1, F)\} & \text{if } F \neq A \end{cases}$
SelectScan( $s_1, A = B$ )	$B(s_1)$	$R(s_1) / \max\{V(s_1, A), V(s_1, B)\}$	$\begin{cases} \min\{V(s_1, A), V(s_1, B)\} & \text{if } F = A, B \\ \min\{R(s), V(s_1, F)\} & \text{if } F \neq A, B \end{cases}$
ProjectScan( $s_1, L$ )	$B(s_1)$	$R(s_1)$	$V(s_1, F)$
ProductScan( $s_1, s_2$ )	$B(s_1) + R(s_1) * B(s_2)$	$R(s_1) * R(s_2)$	$\begin{cases} V(s_1, F) & \text{if } F \text{ is in } s_1 \\ V(s_2, F) & \text{if } F \text{ is in } s_2 \end{cases}$

Figure 17-13: The statistical cost formulas for scans

### 17.5.1 Table scan的代价

每个table scan都维护了一个record file, record file又维护了一个当前的record page, record page实际上又是一个buffer中的page, page又实际对应了底层的一个文件块。当某个page上对应的记录全部被读取完毕后,该page所在的缓冲区会被取消固定(unpin), 并且记录文件中的下一个页(说准确点,应该是下一个块)又被读。因此, table scan只需一次遍历就可以访问所有的记录, 因此 $B(s)$ 、 $R(s)$ 和 $V(s, F)$ 则很容易计算, 和一个表的数据统计信息几乎一样, 如17-13中公式所示。

### 17.5.2 Select scan的代价

一个select scan有一个underlying的scan, 我们用 $s_1$ 来表示, 每次select scan调用 `next()` 方法, 则都会相应调用 $s_1$ 的 `next()` 方法一次或多次, 每次select scan调用 `getXXX()` 方法都会执行 $s_1$ 的相应get方法, 这些get方法是无需块访问的, 于是一个select scan的块访问数和它所拥有的underlying的scan的块访问数是一样的, 即 $B(s) = B(s_1)$ 。

至于 $R(s)$ 和 $V(s, F)$ 的计算, 则取决于相应具体的谓词了, 例如我们先分析一个常见的例子, 谓词的判断逻辑是一个字段的取值是一个具体的值, 或者一个字段的取值和另一字段的取值相等。

#### **Selection on a constant**

假设谓词是形如“ $A = c$ ”的形式, 如果我们假设A字段的所有取值全是均匀的分布(这个假设其实很常见), 那么匹配该谓词的所有记录数量则为

$$R(s_1) / V(s_1, A)$$

在均匀分布的假设下，也意味着输出中的其他字段的取值也是均匀分布的，因此有：

$$\begin{aligned} V(s, A) &= 1 \\ V(s, F) &= \min\{R(s), V(s_1, F)\}, \text{ for all other fields } F \end{aligned}$$

### **Selection on a field**

现在我们假设谓词的形式是“ $A = B$ ”，在这种情况下，字段A和字段B肯定是存在某种关系的，尤其是，如果我们假设B字段的所有可能取值数比A多（即 $V(s_1, A) < V(s_1, B)$ ），那么A字段的每个取值必然能在B字段中找到一个对应相等的值，这种情况其实就是B字段是某张表的主键，而字段A是另一张表中外键的情况了！在这种情况下，A字段的每个取值在匹配B字段中取值的概率为 $1/V(s_1, B)$ ；当然如果我们的假设刚好相反，即B字段的所有可能取值数比A少（即 $V(s_1, A) > V(s_1, B)$ ），概率值则为 $1/V(s_1, A)$ 。

无论是上述的哪一种情况，反正概率总是取值数多的那个字段的倒数，因此 $R(s) = R(s_1)/\max\{V(s_1, A), V(s_1, B)\}$ 。

在均匀分布的假设下，也意味着每个A字段的值也是有相同的概率匹配B字段的值，所以有：

$$\begin{aligned} V(s, F) &= \min\{V(s_1, A), V(s_1, B)\}, \text{ for } F=A \text{ or } B \\ V(s, F) &= \min\{R(s), V(s_1, F)\}, \text{ for all fields } F \text{ other than } A \end{aligned}$$

### **17.5.3 Project scan的代价**

和select scan一样，project scan也有一个underlying的scan（记作 $s_1$ ），这3个值可以很快地算出来，有：

$$\begin{aligned} B(s) &= B(s_1) \\ R(s) &= R(s_1) \\ V(s, F) &= V(s_1, F), \text{ for all fields } F \end{aligned}$$

### **17.5.4 Product scan的代价**

一个product scan有两个underlying的scan，记作 $s_1$ 和 $s_2$ ，和之前小节中分析的一样，遍历一次product scan，需要遍历一遍 $s_1$ ，而对于 $s_1$ 中的每一条记录，都要遍历一遍 $s_2$ ，因此有下列的公式：

```

B(s) = B(s1) + (R(s1) * B(s2))
R(s) = R(s1) * R(s2)
V(s, F) = V(s1, F) or V(s2, F), depending on which table F

```

这里非常有趣的一点是， $B(s)$ 这个公式不是关于 $s_1$ 和 $s_2$ 对称的，也就是说，从逻辑上讲，语句 `Scan s3 = new ProductScan(s1, s2);` 和 `Scan s3 = new ProductScan(s2, s1);` 的结果是一样的，但是访问代价是不一样的。

差了多少呢？我们现在定义 $RPB(s) = R(s)/B(s)$ ，即scan结果每个块上的记录数，于是上面的公式可以改写为

$B(s) = B(s_1) + (RPB(s_1) * B(s_1) * B(s_2))$ ，这个公式的关键项其实就是 $RPB(s_1) * B(s_1) * B(s_2)$ ，而决定该项大小的关键就是 $RPB(s_1)$ ，所以，把RPB小的那个scan放在product scan的左边是会让 $B(s)$ 整体取得最小！

举个例子来说，假设 $s_1$ 是学生表STUDENT， $s_1$ 是学院表DEPT，因为学生的记录数肯定远大于学院的记录数，并且一条学生的信息一般都比一条学院信息大，也就是说学生表的PRB更小，所以把学生表放在product的左边会更好。

译者注：其实我个人感觉这个推导不完全正确，因为还和 $B(s_1)$ 有关，如果你真的要仔细推导的话，你会发现除了RPB这个因素之外，其实还和 $R(s_1)$ 、 $R(s_2)$ 的具体取值有关，我只能说作者给出的上述结论在大多数情况下是成立的。

### 17.5.5 一个具体的例子

现在考虑一个SQL查询，其功能是返回所有数学专业的学生名字，图17-14 (a) 展示了这个query，图17-14 (b) 给出了相应的SimpleDB代码，图17-14 (c) 给出了这些scan的数据统计信息，我们假设底层实际表的存储和16章中的图16-7一样。

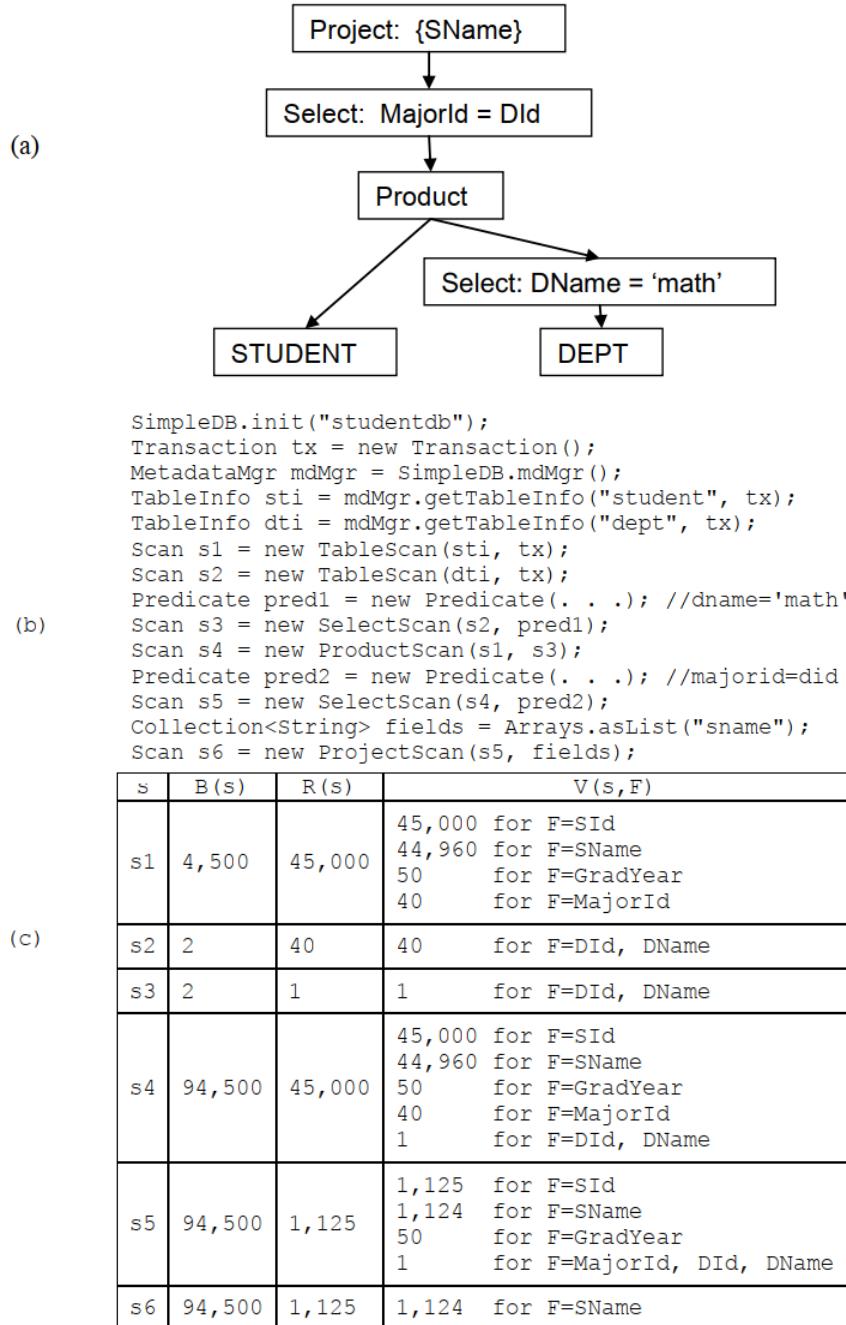


Figure 17-14: Calculating the cost of a scan

$s_1$ 和 $s_2$ 和学生表、学院表的数据统计信息一样，而 $s_3$ 这个select scan只返回1条记录，但需要访问的块数则是所有块； $s_4$ 这个product scan返回45000条学生记录和一条专业记录的笛卡尔积，输出自然也是45000条记录，但是这个笛卡尔积的操作会需要94500次块访问，因为每个学生记录都会遍历一遍DEPT表，这需要 $45000 \times 2 = 90000$ 次块访问，再加上STUDENT表也要遍历一遍，所以就是 $90000 + 4500 = 94500$ 次块访问。

而至于 $s_5$ 这个select scan的话会把笛卡尔积的45000条记录除以40（假设学生的专业是均匀分布的），得到1125条记录，但是需要访问的块数丝毫不会减少。project scan的话，不会改变任何一个值。

数据库系统重新计算数学学院这条记录45,000次，而且代价很高，这似乎很奇怪，我们查看一下STUDENT表和s3的RPB取值，我们会发现 $RPB(STUDENT) = 10$ ，而 $RPB(s3) = 0.5$ ，由于把较小RPB的scan放在product scan的左侧时效代价更低，因此将s4定义为以下会更有效：

```
s4 = new ProductScan(s3, STUDENT)
```

练习17.7要求您证明在这种情况下，该操作仅需要4502次块访问。

## 17.6 Plan

一个SQL查询可能对应多个等价的查询树，而每个查询树又对应一个不同的scan，因此又对应有不一样的执行代价。数据中的planner是负责创建最小代价scan的组件。为了实现这个功能，planner需要创建多个可能的查询树，并且分别计算出它们的代价，然而最后，只选择那个代价最小的scan。

为了比较执行代价而构造的查询树称为plan。

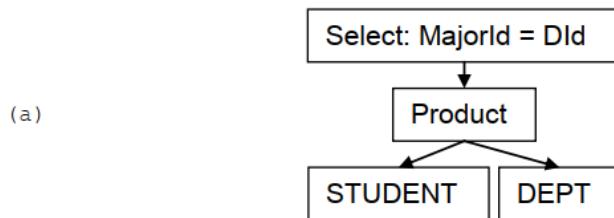
plan和scan在概念上相似，因为它们都表示查询树。不同之处在于，plan对象具有估算代价的方法，它访问数据库的元数据，但不访问实际数据，我们可以创建多个plan，而不会引起过度的磁盘访问开销。然而，scan对象则是为了访问数据库而创建的，创建scan对象后，将为查询中提到的每个表打开一个record file，该record file又将其记录文件的第一个块读入缓冲区，因为打开scan会导致磁盘访问，所以数据库系统在确定是执行查询的最佳方法之前不会创建scan。

一个plan的接口为 Plan，如下：

```
public interface Plan {
    public Scan open();
    public int blocksAccessed();
    public int recordsOutput();
    public int distinctValues(String fldName);
    public Schema schema();
}
```

解开了中支持方法 blocksAccessed()，recordsOutput() 和 distinctValues()，就是对应于一个plan的 $B(s), R(s), and V(s, F)$ 三个值。schema() 方法则会返回输出表的schema，query planner到时候可以使用这个schema来做类型验证，以及优化plan的工作；最后，每个plan对象也会有一个 open() 方法来创建一个对应的scan对象。

创建一个plan和创建一个scan类似，对于每个关系代数操作，都会有一个对应实现了 Plan 接口的具体类，TablePlan 类会处理存储的表。例如，考虑一下图17-16中的例子，这个例子和图17-4中的例子是对应一样的query，图17-4 (b) 和图17-16 (b) 是类似的，主要的区别就是这里创建了一个plan。



(b)

```

SimpleDB.init("studentdb");
Transaction tx = new Transaction();
Plan p1 = new TablePlan("student", tx);
Plan p2 = new TablePlan("dept", tx);
Plan p3 = new ProductPlan(p1, p2);
Predicate pred = new Predicate(...); //majorid=did
Plan p4 = new SelectPlan(p3, pred);

Scan s = p4.open();
while (s.next())
    System.out.println(s.getString("sname"));
s.close();

```

Figure 17-16: Creating and opening a query plan

下面将依次给出 TablePlan , SelectPlan , ProjectPlan 和 ProductPlan 的实现，其中 TablePlan 会直接从 StatMgr 类中估计代价，其他类则会按照17.5小节中的那张表格计算相应的代价。

```

public class TablePlan implements Plan {
    private Transaction tx;
    private TableInfo tableInfo;
    private StatInfo statInfo;

    public TablePlan(String tblName, Transaction tx) throws
        this.tx = tx;
        this.tableInfo = SimpleDB.metadataMgr().getTableInfo(tblName);
        this.statInfo = SimpleDB.metadataMgr().getStatInfo(tblName);
    }

    @Override
    public Scan open() throws IOException {
        return new TableScan(tableInfo, tx);
    }

    @Override
    public int blockAccessed() {
        return statInfo.blocksAccessed();
    }

    @Override
    public int recordsOutput() {
        return statInfo.recordsOutput();
    }

    @Override
    public int distinctValues(String fldName) {
        return statInfo.distinctValues(fldName);
    }

    @Override
    public Schema schema() {
        return tableInfo.schema();
    }
}

```

select plan的代价估计比其他操作复杂一些，因为具体的估计取决于谓词是什么。因此，select plan可以使用谓词的 reduceFactor() 和 equatesWithConstant() 方法。方法 recordsAccessed() 会调用 reductionFactor() 方法来计算这个谓词使得输入表记录减小的程度。 distinctValues() 会使用方法 equatesWithConstant() 来判断指定字段的值是否满足谓词中的常量。

```
public class SelectPlan implements Plan {  
    private Plan plan;  
    private Predicate predicate;  
  
    public SelectPlan(Plan plan, Predicate predicate) {  
        this.plan = plan;  
        this.predicate = predicate;  
    }  
  
    @Override  
    public Scan open() throws IOException {  
        Scan scan = plan.open();  
        return new SelectScan(scan, predicate);  
    }  
  
    @Override  
    public int blockAccessed() {  
        return plan.blockAccessed();  
    }  
  
    @Override  
    public int recordsOutput() {  
        // predicate.reductionFactor(plan)就是图17-13中的  
        // reductionFactor  
        return plan.recordsOutput() / predicate.reductionFactor(plan);  
    }  
  
    @Override  
    public int distinctValues(String fldName) {  
        if (null != predicate.equatesWithConstant(fldName))  
            return 1;  
        else {  
            String theOtherFldName = predicate.equatesWithConstant(theOtherFldName);  
            if (theOtherFldName != null)  
                return Math.min(plan.distinctValues(fldName),  
                               plan.distinctValues(theOtherFldName));  
            else  
                return Math.min(recordsOutput(), plan.distinctValues());  
        }  
    }  
  
    @Override  
    public Schema schema() {  
        return plan.schema();  
    }  
}
```

ProjectPlan 类和 ProductPlan 类会根据underlying的plan对象的 schema创建新的schema结果。具体来说， ProjectPlan 类会在 underlying plan对象已有schema中筛掉一些字段； ProductPlan 类则会取两个schema的并集。

```
public class ProjectPlan implements Plan {  
    private Plan plan;  
    private Schema schema = new Schema();  
  
    public ProjectPlan(Plan plan, Collection<String> fieldList)  
        this.plan=plan;  
        for (String fieldName : fieldList)  
            schema.add(fieldName, plan.schema());  
    }  
  
    @Override  
    public Scan open() throws IOException {  
        Scan scan=plan.open();  
        return new ProjectScan(scan,schema.fields());  
    }  
  
    @Override  
    public int blockAccessed() {  
        return plan.blockAccessed();  
    }  
  
    @Override  
    public int recordsOutput() {  
        return plan.recordsOutput();  
    }  
  
    @Override  
    public int distinctValues(String fldName) {  
        return plan.distinctValues(fldName);  
    }  
  
    @Override  
    public Schema schema() {  
        return schema;  
    }  
}
```

```
public class ProductPlan implements Plan {
    private Plan plan1, plan2;
    private Schema schema = new Schema();

    public ProductPlan(Plan plan1, Plan plan2) {
        this.plan1 = plan1;
        this.plan2 = plan2;
        this.schema.addAll(plan1.schema());
        this.schema.addAll(plan2.schema());
    }

    @Override
    public Scan open() throws IOException {
        Scan scan1 = plan1.open();
        Scan scan2 = plan2.open();
        return new ProductScan(scan1, scan2);
    }

    /**
     * 图17-13中的 B(s_1) + R(s_1) * B(s_2)
     *
     * @return
     */
    @Override
    public int blockAccessed() {
        return plan1.blockAccessed() +
            (plan1.recordsOutput() * plan2.blockAccessed());
    }

    /**
     * 图17-13中的 R(s_1) * R(s_2)
     * @return
     */
    @Override
    public int recordsOutput() {
        return plan1.recordsOutput() * plan2.recordsOutput();
    }

    @Override
    public int distinctValues(String fldName) {
        if(plan1.schema().hasFiled(fldName))
            return plan1.distinctValues(fldName);
        else
            return plan2.distinctValues(fldName);
    }

    @Override
    public Schema schema() {
```

```

        return schema;
    }
}

```

对于每个类中的 `open()` 方法则比较简单，通常来说，构造一个`plan`对应的`scan`需要2步：

- 首先，对于每个`plan`，都会先构造其`underlying plan`的`scan`（当然，`underlying plan`可能又有另外的`underlying plan`，这是一个递归的过程）；
- 随后，把这个构造好的 `Scan` 接口对象构造一个具体的`Scan`类型对象。

## 17.7 谓词

### 17.7.1 谓词定义及相关API

现在，我们来考虑一下谓词的结构以及它们在SimpleDB中的实现和使用方式。

从结构上讲，我们有以下定义：

一个谓词(predicate)是一系列项(term)的布尔组合；一个项(term)是两个表达式(expression)之间的比较；一个表达式(expression)包含对字段名和常数上的操作。

例如，考虑下面的这个谓词：

```
(GradYear = 2006 or GradYear = 2007) and MajorId = DId
```

该谓词包含三个项，前两个项将字段名称`GradYear`与常量进行比较，而第三项将两个字段名称进行比较；这些项中的每个表达式都不涉及任何运算，并且可以是字段名称或常量。通常，表达式可以具有运算符（例如算术运算符）或内置函数。

在SimpleDB中，表达式只能是常量或字段名；一个项只支持相等比较；谓词只支持项之间的主合取范式（离散数学中的概念）。在这里，我们只讨论简化的情况，练习17.11至17.13要求你考虑更一般的情况。

```

public class Predicate {
    // 供 Parser调用
    public void conjoinWith(Predicate predicate);

    // 供 SelectScan中的next()方法调用
    public boolean isSatisfied(Scan s);

    // 供SelectPlan中的recordsOutput()方法调用
    public int reductionFactor(Plan p);

    // 供 query planner调用
    public Predicate selectPred(Schema schema);
    public Predicate joinPred(Schema schema1, Schema schema2);
    public Constant equatesWithConstant(String fieldName);
    public String equatesWithField(String fieldName);

}

```

上面给出了类 `Predicate` 的API，这些方法解决了数据库系统中好几个部分的需求：

- parser会在处理SQL的where语句时构建一个谓词，那时会调用方法 `conjoinWith()` 来将一个当前这个谓词与另外一个谓词取AND运算。
- 一个select scan会调用 `isSatisfied()` 方法来判断一个谓词是否满足。
- 一个select plan会调用 `reductionFactor()` 和 `equatesWithConstant()` 方法来评估一个谓词的筛选性(selectivity)。
- query planner 需要分析谓词的范围并将其分解为较小的部分。方法 `selectPred()` 返回谓词的一部分（如果有的话），该部分是具有指定schema表的选择谓词。同样，方法 `joinPred()` 返回两个指定schema的join谓词（如果有的话）。`equatesWithConstant()` 方法会针对某个指定字段A查找形式为“`A = c`”的项；该项（term）的存在表明可能建立索引。出于类似的原因，方法 `equatesWithField()` 则会查找形式为“`A = B`”的项。

simpleDB对应的SQL语句也只支持两种常数：整型和字符串，而在真正的SQL中支持很多数据类型，但是无论有多少数据类型，query是不会具体去处理字段的类型的，而是用一个抽象的Constant类型来描述。考虑这样一个谓词的例子，`MajorId=DId`，这个谓词并不在意数据到底是什么类型的，谓词只关注这些值是否可以有意义地进行比较。

在SimpleDB中，接口 Constant 就是用来完成这个工作的，其中包含了一些常用的方法（如 equals()，hashCode()，compareTo() 等），方法 asJavaVal() 会转化为实际的Java类型，如下所示：

```
public interface Constant extends Comparable<Constant> {  
    public Object asJavaVal();  
}
```

对于每个具体的数据类型，都会有 Constant 接口的实现，在SimpleDB 中，也就有 IntConstant 和 StringConstant 类，实现代码如下所示：

```
// StringConstant.java
public class StringConstant implements Constant {
    private String val;

    public StringConstant(String val) {
        this.val = val;
    }

    @Override
    public int hashCode() {
        return val.hashCode();
    }

    @Override
    public boolean equals(Object obj) {
        StringConstant sc = (StringConstant) obj;
        return sc != null && val.equals(sc.val);
    }

    @Override
    public String toString() {
        return val.toString();
    }

    @Override
    public Object asJavaVal() {
        return val;
    }

    @Override
    public int compareTo(Constant o) {
        StringConstant sc = (StringConstant) o;
        return val.compareTo(sc.val);
    }
}

// IntConstant.java
public class IntConstant implements Constant {
    private Integer val;

    public IntConstant(Integer val) {
        this.val = val;
    }

    @Override
    public int hashCode()
```

```

        return val.hashCode();
    }

    @Override
    public boolean equals(Object obj) {
        IntConstant ic = (IntConstant) obj;
        return ic != null && val.equals(ic.val);
    }

    @Override
    public String toString() {
        return val.toString();
    }

    @Override
    public Object asJavaVal() {
        return val;
    }

    @Override
    public int compareTo(Constant o) {
        IntConstant ic = (IntConstant) o;
        return val.compareTo(ic.val);
    }
}

```

### 17.7.2 谓词的实现

SimpleDB中的表达式则是使用接口 `Expression` 来实现的，还记得表达式是什么吗？一个表达式(expression)包含对字段名和常数上的操作。代码如下：

```

public interface Expression {
    public boolean isConstant();
    public boolean isFieldName();
    public Constant asConstant();
    public String asFieldName();

    public Constant evaluate(Scan scan);
    public boolean appliesTo(Schema schema);
}

```

这个接口有2个具体的实现类，分别是形如“ $A=c$ ”和“ $A=B$ ”这样的表达式。方法 `isConstant()`、`isFieldName()`、`asConstant()`、`asFieldName()` 允许客户端获得表达式的内容，并且会被planner用来分析一个query的格式。方法 `appliesTo()` 告诉planner当前表达式是否涉及指定schema中的字段，方法 `evaluate()` 则会在返回某个scan的输出记录对应这个表达式的具体取值。如果这个表达式是一个形如“ $A=c$ ”的常数表达式，则你可以视为这个scan输入参数是无关的，并直接返回这个c；如果这个表达式是一个形如“ $A=B$ ”的字段表达式，则该方法会返回当前记录对应这个字段的取值。Talk is cheap, show you the code!

```
// ConstantExpression.java
public class ConstantExpression implements Expression {
    private Constant val;

    public ConstantExpression(Constant val) {
        this.val = val;
    }

    @Override
    public boolean isConstant() {
        return true;
    }

    @Override
    public boolean isFieldName() {
        return false;
    }

    @Override
    public Constant asConstant() {
        return val;
    }

    @Override
    public String asFieldName() {
        throw new ClassCastException();
    }

    @Override
    public Constant evaluate(Scan scan) {
        return val;
    }

    @Override
    public boolean appliesTo(Schema schema) {
        return true;
    }

    public String toString()
    {
        return val.toString();
    }
}

// FieldNameExpression.java
public class FieldNameExpression implements Expression {
    private String fldName;
```

```

public FieldNameExpression(String fldName) {
    this.fldName = fldName;
}

@Override
public boolean isConstant() {
    return false;
}

@Override
public boolean isFieldName() {
    return true;
}

@Override
public Constant asConstant() {
    throw new ClassCastException();
}

@Override
public String asFieldName() {
    return fldName;
}

@Override
public Constant evaluate(Scan scan) {
    return scan.getVal(fldName);
}

@Override
public boolean appliesTo(Schema schema) {
    return schema.hasFiled(fldName);
}

public String toString()
{
    return fldName;
}
}

```

SimpleDB的项是通过类 Term 来实现的，还记得项的定义吗？一个项(term)是两个表达式(expression)之间的比较，其代码如下所示，方法 reductionFactor() 是严格按照17.5.2小节中的公式实现的，这个方法有4种可能的形式：

- 表达式等号左右两边都是字段；
- 表达式等号左边是字段，等号右边是一个值；

- 表达式等号左边是一个值，等号右边是一个字段；
- 表达式等号左右两边都是值。

```

public class Term {
    private Expression lhs, rhs;

    public Term(Expression lhs, Expression rhs) {
        this.lhs = lhs;
        this.rhs = rhs;
    }

    /**
     * 一个谓词项使得记录数减少的因子。
     * <p>
     * 详情参见图 17-12
     *
     * @param plan
     * @return
     */
    public int reductionFactor(Plan plan) {
        String lhsName, rhsName;
        // 1. 左右两边都是字段
        // 图17-12 中的 max{V(s_1,A), V(s_1,B)}
        if (lhs.isFieldName() && rhs.isFieldName()) {
            lhsName = lhs.asFieldName();
            rhsName = rhs.asFieldName();
            return Math.max(plan.distinctValues(lhsName),
                plan.distinctValues(rhsName));
        }
        // 2. 左边是字段名, 右边是常量
        if (lhs.isFieldName()) {
            // 图17-12 中的 V(s_1,A)
            lhsName = lhs.asFieldName();
            return plan.distinctValues(lhsName);
        }
        // 3. 左边是常量, 右边是字段名
        if (rhs.isFieldName()) {
            // 图17-12 中的 V(s_1,A)
            rhsName = rhs.asFieldName();
            return plan.distinctValues(rhsName);
        }
        // 4. 左右两边全是常量
        if (lhs.asConstant().equals(rhs.asConstant()))
            return 1;
        else
            return Integer.MAX_VALUE;
    }

    public Constant equatesWithConstant(String fldName) {
        // 左边是字段名, 右边是常量
        if (lhs.isFieldName() && lhs.asFieldName().equals

```

```

        return rhs.asConstant();
        // 左边是常量，右边是字段名
    else if (rhs.isFieldName() && rhs.asFieldName().equals(lhs.asFieldName()))
        return lhs.asConstant();
    else
        return null;
}

public String equatesWithField(String fldName) {
    if (lhs.isFieldName() && lhs.asFieldName().equals(fldName))
        return rhs.asFieldName();
    } else if (rhs.isFieldName() && rhs.asFieldName().equals(lhs.asFieldName()))
        return lhs.asFieldName();
    } else
        return null;
}

public boolean appliesTo(Schema schema) {
    return lhs.appliesTo(schema) && rhs.appliesTo(schema);
}

/**
 * 判断一个项左右两边是否相等
 *
 * @param scan
 * @return
 */
public boolean isSatisfied(Scan scan) {
    Constant lhsVal = lhs.evaluate(scan);
    Constant rhsVal = rhs.evaluate(scan);
    return lhsVal.equals(rhsVal);
}

public String toString() {
    return lhs.toString() + " = " + rhs.toString();
}
}

```

谓词的实现类 `Predicate` 代码如下所示，一个谓词就是通过很多个项 (term) 组成的，谓词中会把各方法实现转发给相应的 `Term` 类中的方法。

```

public class Predicate {
    private List<Term> terms = new ArrayList<>();

    public Predicate() {
    }

    /**
     * Creates a predicate containing a single term.
     * @param term
     */
    public Predicate(Term term) {
        this.terms.add(term);
    }

    // 供 Parser调用
    public void conjoinWith(Predicate predicate) {
        terms.addAll(predicate.terms);
    }

    /**
     * 判断一个谓词是否成立。
     * <p>
     * 只有谓词中所有的项均为真，整个谓词才为真。
     * <p>
     * 该方法供SelectScan中的next()方法调用
     *
     * @param s
     * @return
     */
    public boolean isSatisfied(Scan s) {
        for (Term t : terms) {
            if (!t.isSatisfied(s))
                return false;
        }
        return true;
    }

    /**
     * 判断一个谓词使得输出记录数减少的因子。
     * <p>
     * 详情计算方法参见图 17-12。
     * <p>
     * 该方法供SelectPlan中的recordsOutput()方法调用
     *
     * @param p
     * @return
     */
    public int reductionFactor(Plan p) {

```

```

        int factor = 1;
        for (Term t : terms) {
            factor *= t.reductionFactor(p);
        }
        return factor;
    }

    /**
     * 返回满足指定schema的子谓词。
     * <p>
     * 该方法供 query planner调用
     *
     * @param schema
     * @return
     */
    public Predicate selectPred(Schema schema) {
        Predicate newPredicate = new Predicate();
        for (Term t : terms) {
            if (t.appliesTo(schema))
                newPredicate.terms.add(t);
        }
        if (newPredicate.terms.size() == 0)
            return null;
        else
            return newPredicate;
    }

    /**
     * 返回满足两个schema并集的子谓词。
     *
     * @param schema1
     * @param schema2
     * @return
     */
    public Predicate joinPred(Schema schema1, Schema schema2) {
        Predicate newPredicate = new Predicate();
        Schema newSchema = new Schema();
        newSchema.addAll(schema1);
        newSchema.addAll(schema2);
        for (Term t : terms) {
            if (!t.appliesTo(schema1) &&
                !t.appliesTo(schema2) &&
                t.appliesTo(newSchema))
                newPredicate.terms.add(t);
        }
        if (newPredicate.terms.size() == 0)
            return null;
        else
    }
}

```

```

        return newPredicate;
    }

    /**
     * 判断一个谓词中是否存在形如"A=c"的项,
     * 其中A是指定字段名, c是指定的常量。
     * <p>
     * 如果有, 返回该常量c; 否则返回null。
     *
     * @param fieldName
     * @return
     */
    public Constant equatesWithConstant(String fieldName)
        for (Term t : terms) {
            Constant result = t.equatesWithConstant(fieldName);
            if (result != null)
                return result;
        }
        return null;
    }

    /**
     * 判断一个谓词中是否存在形如"A=B"的项,
     * 其中A是指定字段名, B是另一个字段名
     * <p>
     * 如果有, 返回指定字段名; 否则返回null。
     *
     * @param fieldName 指定字段名A
     * @return 另一个字段名B
     */
    public String equatesWithField(String fieldName) {
        for (Term t : terms) {
            String result = t.equatesWithField(fieldName);
            if (result != null)
                return result;
        }
        return null;
    }

    public String toString() {
        Iterator<Term> iter = terms.iterator();
        if (!iter.hasNext())
            return "";
        String result = iter.next().toString();
        while (iter.hasNext())
            result += " and " + iter.next().toString();
        return result;
    }
}

```

```

    }
}
```

现在我们已经把谓词给实现了，看下怎么使用吧！想必你也肯定知道了。例如，考虑下面这个谓词：

```
SName = 'joe' and MajorId = DId
```

创建该谓词的客户端代码如下：

```

public class PredicateTest {
    public static void main(String[] args) {
        Expression lhs1=new FieldNameExpression("SName");
        Expression rhs1=new ConstantExpression(new String("joe"));
        Term term1=new Term(lhs1,rhs1);

        Expression lhs2=new FieldNameExpression("MajorId");
        Expression rhs2=new FieldNameExpression("DId");
        Term term2=new Term(lhs2,rhs2);

        Predicate pred1=new Predicate(term1);
        System.out.println(pred1);
        pred1.conjoinWith(new Predicate(term2));
        System.out.println(pred1);
    }
}
```

显然，这些代码对于人来说是乏味的，但是，parser用这种方式来解析谓词却是很自然的，并且parser就是负责构造谓词的，它知道怎么做。

## 17.8 章末总结

- 一个 scan 对象代表了一棵关系代数查询树。每一种关系代数操作都有一个 Scan 接口的具体实现，且每个操作在查询树中代表一个结点；对于涉及具体表的操作，则在树中是叶子结点。
- Scan 接口的方法和 RecordFile 类很相似，客户端遍历scan，当前记录不断向前移动，客户端可以提取相应的字段值。Scan 接口的具体实现类会负责移动指向当前记录的“游标”，移动到合适的位置，比较字段值（例如 SelectScan 类）。
- 当一个scan中的每条记录 $r$ 都在某个数据库表中存在一条对应的 $r'$ 记录时，我们称这个scan是可更新的。

- 每个具体的 Scan 接口实现类中，都会存在相应的关系代数操作实现：
  1. 一个select scan会检查underlying的scan对象中的每条记录，如果符合谓词，则返回这条记录；否则不断往下遍历。
  2. 一个product scan返回的是两个scan对象中所有记录的笛卡尔积。
  3. 一个table scan对象会打开指定表的Record File对象，而record file对象会将块固定缓冲区，而在读取相应值的时候又会获取相应的锁。
- 上述这些scan的实现方式被称为基于流水线式的实现，术语"流水线"代表的是查询树自顶向下的方法调用流，以及自底向上的结果返回流。一个查询树的非叶子结点会不断地向子结点forward方法调用，并且只有在叶子结点（也就是table scan）才会真正去访问RecordFile对象，涉及底层的页换入换出、块读写等操作；反过来，一旦叶子结点得到一个记录，就会不断向父结点backward回结果。
- 一条SQL语句可能对应多个查询树，为了构造最高效的查询树，数据库系统必须先估计一下遍历查询树（也就是scan）的代价。为了更好地估计某个scan  $s$ 的执行代价，有下面几个量需要着重关注：
  1.  $B(s)$ 代表遍历 $s$ 需要的块访问次数。
  2.  $R(s)$ 代表 $s$ 的输出记录条数。
  3.  $V(s,F)$ 代表 $s$ 的所有输出记录中， $F$ 字段所有可能取值的数量。
- 数据中planner组件负责创建执行代价最低的scan对象，为了比较各个输出结果等价的scan对象的执行代价，planner会创建多个plan对象。plan和scan在概念上几乎是一模一样的，它们都代表着一棵查询树，区别在于，plan中还有相关的评估函数，来评估执行代价，也就是上述的 $B(s)$ ,  $R(s)$ 和 $V(s,F)$ ,这些评估函数不会访问数据库表中的实际记录，而是访问表的元数据(metadata)。planner会最终选择评估代价最下的plan对象，并调用 `open()` 方法来创建一个对应的实际的scan对象。

## 17.9 建议阅读

## 17.10 练习

## 第18章 SQL解析和包 simpledb.parse

JDBC客户端将SQL语句作为字符串提交给服务器，数据库系统必须处理此字符串以创建可执行的scan。这个过程分为两个阶段：

- 基于语法的阶段，我们称为 解析(parsing) 阶段；
- 基于语义的阶段，即 计划(planning) 阶段。

译者注：我认为planning翻译成计划不是很恰当，下文中直接用planning替代。

我们将在本章讨论解析过程，而在下一章讨论planning。

### 18.1 语法 V.S. 语义

语言的 语法(syntax) 是一组描述字符串的规则，这些字符串可能是有意义的语句。

例如，考虑下面这样一个字符串：

```
select
from table T1 and T2
where b-3
```

很明显，这肯定是一个非法的SQL语句，因为它不满足SQL的语法，原因是：

- select语句必须包含具体的东西；
- 在from语句中指定的表名前面并不需要加关键字 table；
- 关键字 and 应该只是在谓词之间起连接作用，而不是在连接表的时候使用；
- 字符串 b-3 也不满足谓词的定义语法。

以上的这么多原因都充分地说明了这个SQL字符串注定不会是一个有意义的SQL语句，数据库也肯定没办法怎么处理这种“SQL”，就算你指定的T1，T2和b真的是表名和字段名。

语言的 语义(semantics) 指定了语法正确的字符串的实际含义。

现在，我们考虑一个语法正确的SQL语句，如下：

```
select a
from x,z
where b = 3
```

我们可以推断出，这个SQL语句是一个查询，其功能是从两张表的product结果中返回满足谓词 $b=3$ 的所有a字段的记录，这个SQL语句因此是有意义的。

一条语句是否有具体语义还取决于x, z, a和b的具体语义信息。其中，x和z必须是存在的表名，并且这两张表中至少有一个表存在字段a，并且存在一个int类型的字段b，这些语义信息倒是可以从数据库的元数据中获取到，解析器 parser 一点儿也不知道关于元数据的信息，因此也根本不知道也无法去估计一个SQL语句的具体含义；相反，检查具体元数据的工作是planner的责任，我们将会在第19章中讨论planner。

## 18.2 词法分析

parser为了解析一条SQL语句，首先要做的事情就是把输入的SQL字符串分割成一块一块的字符(token)，注意，这里说的字符并不是指单个ASCII码的字符，而是一个单词，例如 I love computer science中包含4个token，即I、love、computer和science。parser中负责将字符串分成token流的部分被称为 词法分析器(lexical analyzer)。

每个token都有一个类型(type)和一个值(value)，SimpleDB中的词法分析器支持5种类型的token，分别为：

- 单字符分隔符，例如，逗号；
- 整形常量，例如123；
- 字符串型常量，例如“joe”；
- 关键字，例如select、from、where等；
- 标识符，例如STUDENT,x, glop34a等。

空格字符（包括空格，制表符和换行符）通常不属于token；唯一的例外是在字符串常量内部的空格字符。空格的作用是用于增强SQL的可读性并使token之间彼此分离。

继续考虑之前的那个例子：

```
select a
from x,z
where b = 3
```

词法分析器会得到下面的10个token，如图18-1所示：

TYPE	VALUE
keyword	select
identifier	a
keyword	from
identifier	x
delimiter	,
identifier	z
keyword	where
identifier	b
delimiter	=
intconstant	3

当parser需要知道token流的时候，会调用词法分析器中的相应方法来得到token。词法分析器中可供parser调用的方法包含两类：

- 第一类是判断当前token是何种类型的方法；
- 第二类是获得当前token的值，并且移动到下一个token的方法，你可以看作好像是词法分析器在不断地“吃”token。

每种类型的token都会在词法分析器中有一对相应的方法，SimpleDB中的词法分析器的API如下所示，包含10个方法：

```
public class Lexer {
    public boolean matchDelim(char ch);
    public boolean matchIntConstant();
    public boolean matchStringConstant();
    public boolean matchKeyword(String w);
    public boolean matchIdentifier();

    public void eatDelim(char ch);
    public int eatIntConstant();
    public String eatStringConstant();
    public void eatKeyword(String w);
    public String eatIdentifier();
}
```

前5个方法返回的是当前token的信息，如果当前token和指定的分隔符相等，则方法 `matchDelim()` 会返回true；类似地，如果当前token和指定的关键字相等，则方法 `matchKeyword()` 会返回true；其他3个方法则会在当前token满足相关类型时返回true。

后5个方法则会不断地“吃掉”当前token，并移动到下一个token（如果有的话）。每个方法中会调用相应的 `matchXXX()` 方法，如果 `match` 方法返回 `false`，于是抛出一个异常；否则的话，下一个token变成当前的token；此外，`eatIntConstant()`、`eatStringConstant()` 和 `eatIdentifier()` 这3个方法还会返回当前token的具体取值。

## 18.3 实现词法分析器

概念上来说，词法分析器的实现倒是可以很简单粗暴——每次从字符串中读一个字符（注意，这里说的字符真的是一个字符），直到遇到了token之间的分隔符（也就是空格了）则停止，然后判断这个token是什么类型，具体token取值又是什么。因此，词法分析器的复杂度实际上是直接正比于token类型的数量的，token类型越多，这个实现越复杂。（译者注，举个例子，考虑一个SQL字符串 `select a from student`，词法分析器一直读读读，'s','e','l','e','c','t', 噢，到了一个空格，那前面的就是一个token了，再判断一下，哦这是 `select`，于是知道这是一个关键字 token；后续分析类似，因此试想一下，如果token的类型很多，那就会有很多的if判断，真是噩梦）。

好在Java语言中提供了两种不同的自带的 token解析器(`tokenizer`)，`tokenizer`是Java中的术语，无关紧要了，反正知道是用来解析token的就好了。这两种实现分别是 `StringTokenizer` 和 `StreamTokenizer`。

- `StringTokenizer` 用法很简单，但是它只支持两种类型的token：分隔符和单词(也就是两个分隔符之前的内容),稍加分析我们便可以知道，这个实际上不适合SQL，因为这个tokenizer到时候会不理解数值和带引号的字符串；
- 相反，`StreamTokenizer` 则支持广泛的token类型，包括SimpleDB中的5种token类型，因此SimpleDB中的 `Lexer` 类基于Java的 stream tokenizer来实现，具体代码如下所示：

```

public class Lexer {
    private Collection<String> keywords;
    private StreamTokenizer tokenizer;

    public Lexer(String s) {
        // 初始化关键字
        initKeywords();
        tokenizer = new StreamTokenizer(new StringReader(
            // 把'.'也视为一个字符，主要是为了到时候SQL中形如 stuTab
            // 识别成stuTable,'.'和name三个token，而不是视stuTab
            tokenizer.ordinaryChar('.'));
        // 使得关键字和标识符大小写不敏感
        tokenizer.lowerCaseMode(true);
        nextToken();
    }

    // ======判断token类型的相关方法=====

    /**
     * 判断当前token是否是指定的分隔符。
     * <p>
     * 在StreamTokenizer中，单字符token的ttype就是字符对应的ASCII
     * 
     * @param ch
     * @return
     */
    public boolean matchDelim(char ch) {
        return ch == (char) tokenizer.ttype;
    }

    public boolean matchIntConstant() {
        return tokenizer.ttype == StreamTokenizer.TT_NUMBER;
    }

    /**
     * 匹配String的单引号。
     * <p>
     * 注意，在SimpleDB中，字符串常量用单引号包围。
     * 
     * @return
     */
    public boolean matchStringConstant() {
        return '\'' == (char) tokenizer.ttype;
    }

    public boolean matchKeyword(String w) {
        return tokenizer.ttype == StreamTokenizer.TT_WORD
            tokenizer.sval.equals(w);
    }
}

```

```

    }

    /**
     * 判断是否是标识符
     * <p>
     * 除了关键字以外的word都视为标识符。
     *
     * @return
     */
    public boolean matchIdentifier() {
        return tokenizer.ttype == StreamTokenizer.TT_WORD
            !keywords.contains(tokenizer.sval);
    }

    // =====词法分析器不断“吃掉”当前token的相关方法=====
    public void eatDelim(char ch) {
        if (!matchDelim(ch))
            throw new BadSyntaxException();
        nextToken();
    }

    public int eatIntConstant() {
        if (!matchIntConstant())
            throw new BadSyntaxException();
        int i = (int) tokenizer.nval;
        nextToken();
        return i;
    }

    public String eatStringConstant() {
        if (!matchIntConstant())
            throw new BadSyntaxException();
        String str = tokenizer.sval;
        nextToken();
        return str;
    }

    public void eatKeyword(String w) {
        if (!matchKeyword(w))
            throw new BadSyntaxException();
        nextToken();
    }

    public String eatIdentifier() {
        if (!matchIdentifier())
            throw new BadSyntaxException();
        String str = tokenizer.sval;
        nextToken();
    }
}

```

```

        return str;
    }

    /**
     * 得到下一个token
     */
    private void nextToken() {
        try {
            tokenizer.nextToken();
        } catch (IOException e) {
            throw new BadSyntaxException();
        }
    }

    private void initKeywords() {
        keywords = Arrays.asList("select", "from", "where",
            "insert", "into", "values", "delete",
            "update", "set", "create", "table",
            "varchar", "int", "view", "as", "index",
        }
    }
}

```

Lexer 类的构造函数会创建一个stream tokenizer，其中对 tokenizer.ordinaryChar() 方法的调用作用是告诉tokenizer把'.'也看做是一个token，这样是为了到时候处理SQL中形如stuTable.name的字段时，识别成stuTable,'.'和name三个token，而不是视stuTable.name为一整个token（虽然说，在SimpleDB中，我们实际并不会用到点这个符号，但是在真正的SQL语句中必须把这个考虑到）；而方法调用 tokenizer.lowerCaseMode(true) 则会告诉tokenizer将字符串中所有的token全部转为小写的（但是被引号括起来的字符串不会），这样一来，SQL对于关键字和标识符大小写不敏感了。

在StreamTokenizer类中维护了一个public的实例变量，名为 ttype，表示的是当前token的类型；词法分析器简单地通过这个实例变量的值来判断token的类型，有一个看上去不太好理解的方法就是 matchStringConstant()，该方法判断的是一个字符串前的单引号字符。

这个只要你稍微看下StreamTokenizer类的文档你就知道是什么意思了，为了方便大家看，我只截出了关于ttype的文档描述，如下：

```

/**
 * After a call to the {@code nextToken} method, this field
 * contains the type of the token just read. For a single character
 * token, its value is the single character, converted to a
 * For a quoted string token, its value is the quote character.
 * Otherwise, its value is one of the following:
 * <ul>
 * <li>{@code TT_WORD} indicates that the token is a word.
 * <li>{@code TT_NUMBER} indicates that the token is a number.
 * <li>{@code TT_EOL} indicates that the end of line has been
 *       The field can only have this value if the
 *       {@code eolIsSignificant} method has been called with
 *       argument {@code true}.
 * <li>{@code TT_EOF} indicates that the end of the input stream
 *       has been reached.
 * </ul>
 * <p>
 * The initial value of this field is -4.
 *
 * @see     java.io.StreamTokenizer#eolIsSignificant(boolean)
 * @see     java.io.StreamTokenizer#nextToken()
 * @see     java.io.StreamTokenizer#quoteChar(int)
 * @see     java.io.StreamTokenizer#TT_EOF
 * @see     java.io.StreamTokenizer#TT_EOL
 * @see     java.io.StreamTokenizer#TT_NUMBER
 * @see     java.io.StreamTokenizer#TT_WORD
 */
public int ttype = TT NOTHING;

```

StreamTokenizer类中的 `.nextToken()` 方法可能会抛出一个 IOException, 我们的 Lexer 类中的私有方法 `nextToken()` 其实就是对上述方法的一个调用, 并且把这个IOException转化为一个 BadSyntaxException, 到时候这个BadSyntaxException可以传给客户端代码(最终又会变成一个SQLException, 我们将会在第20章中看到)。

方法 `initKeywords()` 则就是完成了初始化SimpleDB中SQL关键字的工作。

## 18.4 语法

译者注: 本节中会出现一些编译原理中的术语, 如果你有一点基础知识会帮助你更好的理解。

一个 语法(Grammar) 指的是一组规则, 这些规则定义了token之间合法的组合形式。

下面就是语法中的一条规则的示例:

```
<Field> := IdTok
```

一条语法规则的左边指明了 句法类别(syntactic category) , 句法类别指的是一种语言中的某个概念。 (译者注：这里我举个例子，比如说一句话 This apple is a banana,这个句子，在词法上 (syntax) 的确没错，但是在语义上 (semantic) ，是错的。那句法类别又是什么呢?是的，就是一个句子中的成分，主、谓、宾这些。) 于是在上面的例子中， `<Field>` 就是一个句法类别，表示的是一个表的字段名。

句法类别的具体格式是在一条语法规则的右边定义的，这个叫 模式 (pattern) .同样，在上述的例子中， `<Field>` 这个句法类别是所有标识符字符串的集合。每一个句法类别都可以看成是一个mini语言，如果一个字符串符合某个句法类别的模式，那么这个字符串就是这个语言的一个句子。 (译者注：说了这么多，一条语法规则其实就是编译原理中的一条产生式，语法规则的左边是非终结符，右边可以是终结符，也可以是非终结符) 。

例如，下面的两个字符串就满足 `<Field>` 这个句法类别的模式。

```
SName
Glop
```

记住，一个标识符没有谁规定了必须是有意义的字符串——它们只要是字符串就行了，因此Glop可以是一个标识符，但是select不是，因为它是关键字。

一条语法规则的右边可以是具体的token或者其他的句法类别（也就是我们说的，可以是终结符，也可以是非终结符，也可以是二者的组合）。那些具有众所周知取值的token (即关键字和定界符)，可以被直接写入语法规则中。其他类型的token (标识符，整数常量和字符串常量) 分别被记作为 `IdTok` , `IntTok` 和 `StrTok` 。我们还使用三个元字符 ( [ , ] 和 | ) 作为标点符号；这些字符并不是语言中的分隔符，我们只是用它们来帮助表达一条语法规则的模式。为了示例，请考虑以下这个包含了四条语法规则的一个mini语言：

```
<Constant>      := StrTok | IntTok
<Expression>    := <Field> | <Constant>
<Term>          := <Expression> = <Expression>
<Predicate>     := <Term> [ AND <Predicate> ]
```

- 第一条规则定义了句法类别 `<Constant>` ,它可以用来标识任何的常量——字符串常量和整形常量，元字符 | 表示或的意思，也就是说， `<Constant>` 可以匹配整形常量或者字符串常量；

- 第二条语法规则定义了句法类别 `<Expression>`，它表示的是一个谓词项中的一侧，该规则指明，一个表达式可以是一个字段名或者常量；
- 第三条语法规则定义了句法类别 `<Term>`，它表示的是两个表达式组成的项（和SimpleDB中的 `Term` 类是一个意思），下面的这4个字符串 `DeptId = DId`, `'math' = DName`, `SName = 123` 和 `65 = 'abc'` 都属于句法类别 `<Term>`。这里你尤其要注意一下最后两个字符串，你看，`SName`明显是个字符串类型，可是右边指明的却是一个整型常量，同理，`65='abc'`也肯定是无厘头的，但是，但是，语法检查可不管这么多具体的语义信息，这两个字符串仍然是属于句法类别 `<Term>` 的；
- 第四条语法规则电话已了句法类别 `<Predicate>`，它表示的是一个或多个term的布尔组合，和SimpleDB中的 `Predicate` 类差不多，`[]` 则是用来表示这是一个可选项，程序员都懂，这条语法规则就是说，一个 `<Predicate>` 是一个或多个 `<Term>` 的组合，并且只支持用AND组合起来，例如，下面的这3个字符串就都是属于 `<Predicate>` 的：

```
DName = 'math'
Id = 3 AND DName = 'math'
MajorId = DId AND Id = 3 AND DName = 'math'
```

如果一个字符串属于某个句法类别，我们可以画出一个对应的 解析树 (parse tree)，一个解析树中的中间结点就是一个句法类别，解析树中的叶子结点肯定是token。考虑图18-3所示的例子，它对应的是 `DName = 'math' AND GradYear = SName` 的解析树。

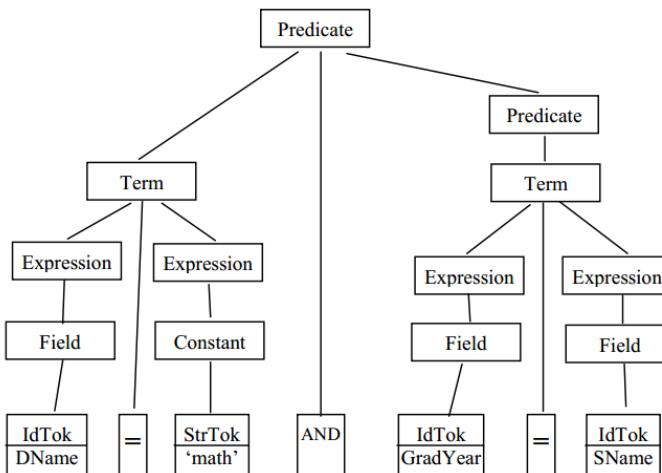


Figure 18-3: A parse tree for the string `DName = 'math' AND GradYear = SName`

上图就不多作解释了，根据上述的4条语法规则显然可以很容易推断得到。

下面我们给出SimpleDB中支持的部分SQL语句的整个语法，整个语法规则被分为9组，分别表示谓词、查询以及各种更新语句：

```
// SimpleDB支持的部分SQL语法
<Field>      := IdTok
<Constant>   := StrTok | IntTok
<Expression> := <Field> | <Constant>
<Term>       := <Expression> = <Expression>
<Predicate>  := <Term> [ AND <Predicate> ]

<Query>      := SELECT <SelectList> FROM <TableList> [ WHERE <Predicate> ]
<SelectList>  := <Field> [ , <SelectList> ]
<TableList>   := IdTok [ , <TableList> ]

<UpdateCmd>  := <Insert> | <Delete> | <Modify> | <Create>
<Create>      := <CreateTable> | <CreateView> | <CreateIndex>

<Insert>      := INSERT INTO IdTok ( <FieldList> ) VALUES ( <ConstList> )
<FieldList>   := <Field> [ , <FieldList> ]
<ConstList>   := <Constant> [ , <Constant> ]

<Delete>      := DELETE FROM IdTok [ WHERE <Predicate> ]

<Modify>      := UPDATE IdTok SET <Field> = <Expression>

<CreateTable>  := CREATE TABLE IdTok ( <FieldDefs> )
<FieldDefs>   := <FieldDef> [ , <FieldDefs> ]
<FieldDef>    := IdTok <TypeDef>
<TypeDef>     := INT | VARCHAR ( IntTok )

<CreateView>   := CREATE VIEW IdTok AS <Query>

<CreateIndex>  := CREATE INDEX IdTok ON IdTok ( <Field> )
```

在前面的小节中，我们已经知道了parser只是解析token的，并没有做类型的兼容性约束，也没有对那些列表项做的大小做约束。例如，一个insert SQL语句必须使得指定的字段名数量和具体字段取值数量保持相等，但是在上述的语法规则中，我们只定义了 `<FieldList>` 和 `<ConstList>`，而没有对二者具体的数量做约束。实际上，这个工作也不应该交给parser来完成，到时候planner会负责确保这些列表项的长度相等，因此，我们称这个SQL是类型兼容的。

## 18.5 递归下降解析器

一个解析树可以被看做是某个字符串词法合法的证明，但是，数据库系统怎么判断一个字符串是词法合法的呢？或者说，数据库系统怎么生成一个解析树并且做相应的合法判断呢？

编程语言的研究者已经提出了很多解析算法来完成这些工作，这些解析算法的复杂度也直接正比于语法的复杂程度（其实和词法分析器中是类似的）。幸运的是，SimpleDB中支持的SQL语法相对来说还是比较简单的，所以我们将使用最简单的解析算法，被称为 递归下降法(*recursive descent*)。（译者注：其实在真正的编程语言（例如C语言）编译器中，用的是自底向上的解析算法，并且需要精心设计语法规则，如满足LR(1)文法等，这都是编译原理中很重要的概念）。

在一个最基本的递归下降解析算法中，每个句法类别都会用一个void类型的方法来实现，每次调用该方法，都会“吃掉”当前的token并且构建相应的解析树，当这个token无法组成一个合法的解析树时，就会抛出一个异常。而这个递归下降解析器显然是会调用词法分析器中的相关方法的。

为了说明情况，我们先举一个只包含谓词部分的5条语法规则，这组成了SQL中的谓词，对应解析该语法的Java代码如下：

```

public class PredParser {
    private Lexer lexer;

    public PredParser(Lexer lexer) {
        this.lexer = lexer;
    }

    public void field() {
        lexer.eatIdentifier();
    }

    public void constant() {
        if (lexer.matchIntConstant())
            lexer.eatIntConstant();
        else
            lexer.eatStringConstant();
    }

    public void expression() {
        if (lexer.matchIdentifier())
            field();
        else
            constant();
    }

    public void term() {
        expression();
        lexer.eatDelim('=');
        expression();
    }

    public void predicate() {
        term();
        if (lexer.matchKeyword("and")) {
            lexer.eatKeyword("and");
            predicate();
        }
    }
}

```

递归下降解析法对于每条语法规则都有一个方法，并且每个方法都会调用语法规则右边词法类别的相应方法。

我们稍微分析一下上面的代码，考虑一下方法 `field()`，它会相应地调用词法分析器中的方法（在这里我们没有管返回值）。如果当前token是标识符，那么这个调用成功，并且“吃掉”这个token；如果当前token不是

标识符的话，那调用 `lexer.eatIdentifier()` 方法自然会抛出一个 `BadSyntaxException` 异常。相似地，其他方法也可以同样地分析。

文法规则的解析器其实也可以不用这种递归下降的方法，你可以直接用很多很多的if判断来完成，在这种实现策略下，我们必须根据if的条件来判断当前token是什么，并且决定下一步怎么做。我们也同样举个例子，考虑下 `constant()` 方法，如果当前token是一个整形的token话，那么就调用 `lexer.eatIntConstant()` 方法；否则的话，则调用 `lexer.eatStringConstant()` 方法；当然，如果既不是整形token也不是字符串型token的话，抛出一个异常就好了。其实 `constant()` 方法还是很简单的情况，我们现在考虑一下 `expression()` 方法，如果当前token是标识符，那么它必须找一个field，否则又必须找一个常量，这种情况下，显然会造成很多判断冗余！而且，你想下，代码里面全是if，真实噩梦！其实，还有更严重的问题，如果某两条规则的第一个token是一样的，那你怎么办呢？好，就算1个token相同你处理好了，2个、3个...甚至10个的相同token前缀的时候，你又怎么办呢？

`predicate()` 方法演示了一个递归的规则可以怎样实现。首先，调用 `term()` 方法“吃掉”一个term，然后判断当前token是不是AND关键字，如果是的话就递归调用；如果不是AND关键字，方法很自然地停止。

递归下降解析法的有趣之处在于，方法调用的顺序刚好和字符串的解析树构造顺序是一样的。练习18.4中会要求你修改每种方法中的代码以打印其名称（添加适当缩进）；结果将类似于分析树。

## 18.6 为解析器添加动作

上述递归下降法的最基础实现实际上并没有做任何事情，知识会在字符串语法合法时正常退出，但也不会返回具体的什么值。虽然说，上面的实现方式的确有点意思，可以构造想要的解析树，但是如果我真的项执行一个SQL语句的话，这种实现其实没有很大作用。于是，我们必须对代码作出一定修改，并且为planner返回一些信息，即为解析器parser添加一些动作(action)。

通常来说，SQL解析器需要提取出例如表名、字段名、谓词和常量部分，具体提取出什么信息也取决于具体的SQL语句：

- 对于一个query，字段名(select语句)、表名(from语句)和谓词(where语句)，这都是肯定要提取的信息；
- 对于一条插入语句，需要提取表名、所有字段名和各字段的取值；
- 对于一条删除语句，需要提取表名和谓词；
- 对于一条修改语句，需要提取表名、修改的字段名、一个包含修改后的字段取值的表达式和一个谓词；
- 对于一个创建表语句，需要提取表名和schema信息；
- 对于一个创建视图语句，需要提取表名和定义；

- 对于一个创建索引语句，需要提取索引名，表名，以及被索引的字段名。

所有的上述这些信息都会从token流中提取得到，这是通过调用 `Lexer` 类中的相关方法来完成的。话都说到这里，想必怎么修改parser肯定是很简单了吧？

下面给出 `Parser` 类的具体代码，这个类实现了SimpleDB中支持的所有SQL语法。

```

// 语法解析器类
public class Parser {
    private Lexer lexer;

    public Parser(Lexer lexer) {
        this.lexer = lexer;
    }

    //=====解析谓词、项、表达式相关的方法=====
    public String field() {
        return lexer.eatIdentifier();
    }

    public Constant constant() {
        if (lexer.matchIntConstant()) {
            int intVal = lexer.eatIntConstant();
            return new IntConstant(intVal);
        } else {
            String strVal = lexer.eatStringConstant();
            return new StringConstant(strVal);
        }
    }

    public Expression expression() {
        if (lexer.matchIdentifier()) {
            String fieldExtracted = field();
            return new FieldNameExpression(fieldExtracted);
        } else {
            Constant constExtracted = constant();
            return new ConstantExpression(constExtracted);
        }
    }

    public Term term() {
        Expression lhs = expression();
        lexer.eatDelim('=');
        Expression rhs = expression();
        return new Term(lhs, rhs);
    }

    public Predicate predicate() {
        Term t = term();
        Predicate pred = new Predicate(t);

        if (lexer.matchKeyword("and")) {
            lexer.eatKeyword("and");
            pred.conjoinWith(predicate());
        }
    }
}

```

```

        return pred;
    }

//=====解析Query相关的方法=====
private Collection<String> tableList() {
    Collection<String> list = new ArrayList<>();
    // 最少有一个表名
    list.add(lexer.eatIdentifier());
    if (lexer.matchDelim('，')) {
        lexer.eatDelim('，');
        list.addAll(tableList()); // 递归调用
    }
    return list;
}

private Collection<String> selectList() {
    Collection<String> list = new ArrayList<>();
    // 最少有一个字段名
    list.add(lexer.eatIdentifier());
    if (lexer.matchDelim('，')) {
        lexer.eatDelim('，');
        list.addAll(selectList()); // 递归调用
    }
    return list;
}

public QueryData query() {
    lexer.eatKeyword("select");
    Collection<String> fields = selectList();
    lexer.eatKeyword("from");
    Collection<String> tables = tableList();
    // 如果有谓词
    Predicate pred = new Predicate();
    if (lexer.matchKeyword("where")) {
        lexer.eatKeyword("where");
        pred = predicate();
    }

    return new QueryData(fields, tables, pred);
}

//=====解析各种update命令相关的方法=====
public Object updateCmd() {
    if (lexer.matchKeyword("insert"))
        return insert();
    else if (lexer.matchKeyword("delete"))
        return delete();
    else if (lexer.matchKeyword("update"))

```

```

        return modify();
    else
        return create();
}

private Object create() {
    lexer.eatKeyword("create");
    if (lexer.matchKeyword("table"))
        return createTable();
    else if (lexer.matchKeyword("view"))
        return createView();
    else
        return createIndex();
}

//=====解析insert相关的方法=====
public InsertData insert() {
    lexer.eatKeyword("insert");
    lexer.eatKeyword("into");

    String tblName = lexer.eatIdentifier();
    lexer.eatDelim('(');
    Collection<String> fields = fieldList();
    lexer.eatDelim(')');

    lexer.eatKeyword("values");
    lexer.eatDelim('(');
    Collection<Constant> vals = constList();
    lexer.eatDelim(')');

    return new InsertData(tblName, fields, vals);
}

private Collection<String> fieldList() {
    Collection<String> list = new ArrayList<>();
    list.add(lexer.eatIdentifier());
    if (lexer.matchDelim(',', ',')) {
        lexer.eatDelim(',', ',');
        list.addAll(fieldList());
    }
    return list;
}

private Collection<Constant> constList() {
    Collection<Constant> list = new ArrayList<>();
    list.add(constant());
    if (lexer.matchDelim(',', ',')) {
}

```

```

        lexer.eatDelim(',');
        list.addAll(constList());
    }
    return list;
}

//=====解析delete相关的方法=====
public DeleteData delete() {
    lexer.eatKeyword("delete");
    lexer.eatKeyword("from");
    String tblName = lexer.eatIdentifier();
    Predicate pred = new Predicate();
    // 如果存在where语句
    if (lexer.matchKeyword("where")) {
        lexer.eatKeyword("where");
        pred = predicate();
    }
    return new DeleteData(tblName, pred);
}

//=====解析modify相关的方法=====
public ModifyData modify() {
    lexer.eatKeyword("update");
    String tblName = lexer.eatIdentifier();

    lexer.eatKeyword("set");
    String fldName = field();

    lexer.eatDelim('=');
    Expression newVal = expression();

    Predicate pred = new Predicate();
    // 如果存在where语句
    if (lexer.matchKeyword("where")) {
        lexer.eatKeyword("where");
        pred = predicate();
    }
    return new ModifyData(tblName, fldName, newVal, pred);
}

//=====解析CreateTable相关的方法=====
public CreateTableData createTable() {
    lexer.eatKeyword("table");
    String tblName = lexer.eatIdentifier();
    lexer.eatDelim('(');
    Schema schema = fieldDefs();
    lexer.eatDelim(')');
}

```

```

        return new CreateTableData(tblName, schema);
    }

    private Schema fieldDefs() {
        Schema schema = fieldDef();
        if (lexer.matchDelim(',')) {
            lexer.eatDelim(',');
            schema.addAll(fieldDefs());
        }
        return schema;
    }

    private Schema fieldDef() {
        String fldName = lexer.eatIdentifier();
        return typeDef(fldName);
    }

    private Schema typeDef(String fldName) {
        Schema schema = new Schema();
        if (lexer.matchKeyword("int")) {
            lexer.eatKeyword("int");
            schema.addIntField(fldName);
        } else {
            lexer.eatKeyword("varchar");
            lexer.eatDelim('(');
            int strLen = lexer.eatIntConstant();
            lexer.eatDelim(')');
            schema.addStringField(fldName, strLen);
        }
        return schema;
    }

    //=====解析CreateView相关的方法=====
    public CreateViewData createView() {
        lexer.eatKeyword("view");
        String viewName = lexer.eatIdentifier();
        lexer.eatKeyword("as");
        QueryData qd = query();

        return new CreateViewData(viewName, qd);
    }

    //=====解析CreateIndex相关的方法=====
    public CreateIndexData createIndex() {
        lexer.eatKeyword("index");
        String indexName = lexer.eatIdentifier();
        lexer.eatKeyword("on");
        String tblName = lexer.eatIdentifier();
    }

```

```
lexer.eatDelim('(');
String fldName = field();
lexer.eatDelim(')');

return new CreateIndexData(indexName, tblName, fldName);
}
}
```

### 18.6.1 解析谓词和表达式

解析器会处理定义谓词和表达式的五条语法规则，因为它们用于解析几种不同类型的SQL语句。解析器中方法其实与之前的示例

类 PredParser 很相似，不同点在于它们现在包含动作和返回值。特别是，方法 field() 会从当前token中提取字段名称，然后将其返回。方法 constant()，expression()，term() 和 predicate() 的实现都几乎相似，各自返回的是Constant对象，Expression对象，Term对象和Predicate对象。

### 18.6.2 解析query

方法 query() 实现的是句法类别的内容，正如解析一个query的过程一样，该方法会继续获取到planner所需的三个信息——字段名称，表名称和谓词——并将它们保存在 QueryData 对象中，这些信息都可以通过 QueryData 类的 getXXX() 方法的访问到。QueryData 类还具有 toString() 方法，该方法可重建这条SQL字符串，这个方法十分重 要，因为在处理视图定义时，我们会需要此方法。具体的实现代码如下：

```
public class QueryData {

    private Collection<String> fields;
    private Collection<String> tables;
    private Predicate pred;

    public QueryData(Collection<String> fields,
                    Collection<String> tables,
                    Predicate pred) {
        this.fields = fields;
        this.tables = tables;
        this.pred = pred;
    }

    public Collection<String> getFields() {
        return fields;
    }

    public Collection<String> getTables() {
        return tables;
    }

    public Predicate getPred() {
        return pred;
    }

    /**
     * 重新构造形如 Select XXX from XXX where XXX的SQL字符串。
     *
     * @return
     */
    @Override
    public String toString() {
        StringBuilder result = new StringBuilder("select ");
        for (String f : fields)
            result.append(f).append(", ");
        // 移除循环最后添加的一个逗号
        result.substring(0, result.length() - 2);

        result.append(" from ");

        for (String tblName : tables)
            result.append(tblName).append(", ");
        // 再次移除循环最后添加的一个逗号
        result.substring(0, result.length() - 2);

        String predStr = pred.toString();
        if (!predStr.equals(""))
            result.append(" where " + pred);
    }
}
```

```
        result.append(" where ").append(predStr);

    return result.toString();
}
}
```

### 18.6.3 解析更新命令

方法 `updateCmd()` 实现了句法类别，该类别表示各种SQL更新语句的并集。在JDBC方法 `executeUpdate()` 执行期间，为了确定命令指示的具体更新类型，解析器调用 `updateCmd()` 方法。该方法会识别字符串的起始token从而识别具体的命令，然后将其分派到该命令相关的特定解析器方法。每种更新方法都有不同的返回类型，因为每种更新方法都需要从字符串中提取出不同种类的信息。因此，方法 `updateCmd()` 的返回类型是 `Object`。

### 18.6.4 解析插入

方法 `insert()` 实现的是句法类别。此方法提取三种信息：表名称，字段列表和值列表。`InsertData` 类对象保存这些值，并通过`get`方法访问实例参数。代码如下：

```
public class InsertData {

    private String tblName;
    private Collection<String> fields;
    private Collection<Constant> vals;

    public InsertData(String tblName,
                      Collection<String> fields,
                      Collection<Constant> vals) {
        this.tblName = tblName;
        this.fields = fields;
        this.vals = vals;
    }

    public String getTblName() {
        return tblName;
    }

    public Collection<String> getFields() {
        return fields;
    }

    public Collection<Constant> getVals() {
        return vals;
    }
}
```

## 18.6.5 解析删除

删除语句的解析则是通过 `delete()` 方法来实现的，并且返回一个 `DeleteData` 类对象，`DeleteData` 类的实现如下：

```
public class DeleteData {  
    private String tblName;  
    private Predicate pred;  
  
    public DeleteData(String tblName, Predicate pred) {  
        this.tblName = tblName;  
        this.pred = pred;  
    }  
  
    public String getTblName() {  
        return tblName;  
    }  
  
    public Predicate getPred() {  
        return pred;  
    }  
}
```

## 18.6.6 解析修改

删除语句的解析则是通过 `modify()` 方法来实现的，并且返回一个 `ModifyData` 类对象，这个类的实现和 `DeleteData` 很相似，区别在于 `ModifyData` 类包含了具体修改的字段和修改后字段的取值，实现如下：

```
public class ModifyData {  
    private String tblName;  
    private String fldName;  
    private Expression newVal;  
    private Predicate pred;  
  
    public ModifyData(String tblName, String fldName,  
                      Expression newVal, Predicate pred) {  
        this.tblName = tblName;  
        this.fldName = fldName;  
        this.newVal = newVal;  
        this.pred = pred;  
    }  
  
    public String getTblName() {  
        return tblName;  
    }  
  
    public String getFldName() {  
        return fldName;  
    }  
  
    public Expression getNewVal() {  
        return newVal;  
    }  
  
    public Predicate getPred() {  
        return pred;  
    }  
}
```

### 18.6.7 解析表创建、视图创建和索引创建

句法类型 `<Create>` 表明这个SQL是创建语句，而具体是创建表还是创建视图或者是创建索引则会有更加细节的句法类型，例如，创建表对应的是 `<CreateTable>` 这个句法类型，通过 `createTable()` 方法来实现，并且返回一个 `CreateTableData` 类对象；其他的实现则类似，下面依次给出XXXData类的代码：

```
// CreateTableData.java

public class CreateTableData {
    private String tblName;
    private Schema schema;
    public CreateTableData(String tblName, Schema schema) {
        this.tblName=tblName;
        this.schema=schema;
    }

    public String getTblName() {
        return tblName;
    }

    public Schema getSchema() {
        return schema;
    }
}

// CreateViewData.java
public class CreateViewData {
    private String viewName;
    private QueryData qd;

    public CreateViewData(String viewName, QueryData qd) {
        this.viewName = viewName;
        this.qd = qd;
    }

    public String getViewName() {
        return viewName;
    }

    public String getViewDef() {
        return qd.toString();
    }
}

// CreateIndexData.java
public class CreateIndexData {
    private String indexName;
    private String tblName;
    private String fldName;

    public CreateIndexData(String indexName, String tblName,
                          String fldName) {
        this.indexName = indexName;
        this.tblName = tblName;
```

```

        this.fldName = fldName;
    }

    public String getIndexName() {
        return indexName;
    }

    public String getTblName() {
        return tblName;
    }

    public String getFldName() {
        return fldName;
    }
}

```

索引是为了增加查询效率而引入的概念，我们将会在第21章具体介绍其实现。

## 18.7 章末总结

- 一个语言的 语法(syntax) 描述的是可能有意义的字符串语句的规则；
- 解析器(parser) 是负责确保某个字符串是语法正确的组件；
- 词法分析器(lexical analyzer) 其实也是解析器的一部分，它负责的是将输入字符串分成一系列的token流；
- 每个token都有一个类型和对应的值，SimpleDB的词法分析器支持五种类型的token：
  1. 单字符分隔符，例如，逗号；
  2. 整形常量，例如123；
  3. 字符串型常量，例如“joe”；
  4. 关键字，例如select、from、where等；
  5. 标识符，例如STUDENT,x, glop34a等。
- 每种token都有两个方法，一个判断当前token是否满足指定的类型，另一个则用来让词法分析器“吃掉”当前的token，并移动到下一个token；
- 一个语言的 语法(grammar) 描述的是一组各种token之间可以怎样合法组合的规则集：
  1. 一条语法规则的左边指明了 句法类型(syntactic category)，这只是一个语言中的一个概念而已，类似主谓宾这些。
  2. 一条语法规则的右边指明了句法类型的字符串 模式(patterns)。
- 某个字符串对应的 解析树(parsing tree) 中，中间结点对应的是句法类型，而叶子节点对应的是token。

- 解析算法(parsing algorithm) 为语法上合法的字符串构造一棵解析树。解析算法的复杂度与它支持的语法的复杂度成比例，一种简单的解析算法称为 递归下降解析法(recursive descent)。
- 递归下降解析法中，每条语法规则都对应有一个方法，每个方法又会调用语法规则右边的相应句法类型的方法。
- 递归下降解析器中的每个方法都会提取其读取的token的值，然后将其返回。一个SQL解析器应从SQL语句中提取必要的信息，例如表名，字段名，谓词和常量等，而具体提取什么内容取还是取决于它是哪种SQL语句：
  1. 对于一条插入语句，需要提取表名、所有字段名和各字段的取值；
  2. 对于一条删除语句，需要提取表名和谓词；
  3. 对于一条修改语句，需要提取表名、修改的字段名、一个包含修改后的字段取值的表达式和一个谓词；
  4. 对于一个创建表语句，需要提取表名和schema信息；
  5. 对于一个创建视图语句，需要提取表名和定义；
  6. 对于一个创建索引语句，需要提取索引名，表名，以及被索引的字段名。

## 18.8 建议阅读

## 18.9 练习

## 第19章 planning和 包 simpledb.planner

在上一章中，我们知道了解析器会解析一条SQL语句，解析完SQL语句后，我们需要做的下一步就是把SQL转化成一个关系代数查询树，这一步被称为 `planning`。在本章中，我们将研究一下基本的planning过程，尤其是，我们将重点考虑一下，为了验证一条SQL语句是否有意义，`planner`需要做的事情，并学习一些基本的构造plan的算法，并且看下 `update plan` 是怎么执行的。

在前面我们就曾说过，一条SQL语句可能有多条对应等效的查询树，并且这些查询树的查询代价往往都是不一样的。一个好的planning算法是一个数据库系统能否商业应用的关键，在大多数的商业数据库系统中，planning算法往往都很复杂，并且是精心fine-tune好的，为的就是能够创建出最高效的plan。在本章在，我们先暂时忽略掉高效这个问题，集中考虑一下`planner`是怎么工作的，而在第24章中，我们再捡起效率这个话题来，那时候再讨论高效的planning算法。

### 19.1 SimpleDB的planner

`planner`是一个数据库系统中负责将一条SQL语句转化为一个plan的组件。

为了更好地学习planning这个过程，我们将从SimpleDB的`planner`开始入手。

#### 19.1.1 使用planner

SimpleDB的`planner`是通过类 `Planner` 来实现的，它的API如下所示：

```
public class Planner {
    public Plan createQueryPlan(String query, Transaction tx);
    public int executeUpdate(String cmd, Transaction tx);
}
```

上述两个方法的第一个参数就是字符串类型的SQL语句，方法 `createQueryPlan()` 会创建一个plan并且将这个plan返回，而方法 `executeUpdate()` 执行的是一条命令SQL，返回的结果是受影响的记录条数（和JDBC中的 `executeUpdate()` 是一样的）。

一个客户端可以通过调用 `simpledb.server.SimpleDB` 类中的静态方法来持有一个 `Planner` 对象，下面的代码演示了如何使用`Planner`对象：

```

public class PlannerTest {

    public static void main(String[] args) throws IOException {
        SimpleDB.init("studentdb");
        Planner planner = SimpleDB.planner();
        Transaction tx = new Transaction();

        // 处理一个query
        String query = "select sname,gradyear from student";
        Plan queryPlan = planner.createQueryPlan(query, tx);
        Scan scan = queryPlan.open();
        while (scan.next()) {
            System.out.println(scan.getString("sname") + " "
                scan.getString("gradyear"));
        }
        scan.close();

        // 处理一个update query
        String cmd = "delete from STUDENT where MajorId=30";
        int numAffected = planner.executeUpdate(cmd, tx);
        System.out.println(numAffected + " records was affected");
    }
}

```

上述代码中的第一部分展示了一个SQL query语句是怎么被处理的，其中SQL语句被传入到 `createQueryPlan()` 方法，然后该方法返回一个对应的Plan对象，打开这个Plan对象则会创建一个具体的Scan，随后遍历这个Scan对象即可；代码中的第二部分展示了一个update query是怎样处理的，同样是先将SQL语句传入 `executeUpdate()` 函数，该函数中会执行相应的update动作，并且返回受影响的记录条数。

## 19.2 planner具体怎么做？

SimpleDB中的planner有两个方法：一个来处理query，一个来处理update。这两个方法其实都会对输入做类似的处理，下面列出了它们具体的执行步骤：

1. 解析SQL语句。在这一步会调用 `解析器(parser)` 中的方法，解析器会执行解析SQL语句的相关方法，并返回一个包含具体数据的对象（`QueryData`，`DeleteData` 等）。例如，对于一条query类型的SQL语句，则会返回一个 `QueryData` 对象；而对于插入SQL语句，则会返回一个 `InsertData` 对象。
2. 验证SQL语句。这一步则会检查上一步中得到的 `XXXData` 对象，从而判断这个SQL语句是否是有意义的。

3. 创建一个对应的plan。这一步中会执行相应的planning算法来生成一个对应的查询树，并且创建一个对应的Plan对象。  
 4a. 返回Plan对象  
 (对于 `createQueryPlan()` 方法而言)  
 4b. 执行plan(对于 `executeUpdate()` 方法而言)。这一步则会调用plan的 `open()` 方法得到一个scan，随后遍历这个scan，对遍历过程中的记录作出对应的修改，并且返回受影响的记录条数。

两个方法都会执行上述的1-3步骤，而不同点就在于它们各自创建的plan了，方法 `createQueryPlan()` 简单地将这个plan对象返回，而 `executeUpdate()` 方法则会打开并且执行这个plan。

方法1涉及了解析SQL的内容，这是我们上一章中讨论的东西，现在我们开始讨论一下剩下的几步。

## 19.2 验证SQL

planner的第一个职责就是判断一条SQL语句是否是有意义的，具体来说，需要考虑下面的这些信息：

- SQL语句中的表和字段名是否已经存在于catalog中；
- SQL语句中的字段是否有歧义；
- SQL语句中涉及字段的操作是否是类型匹配；
- SQL语句中涉及的常量是否和对应字段的类型及大小匹配。

上述所有的信息都可以通过检查表的schema信息来验证。例如，如果不存在相应的schema信息，则说明该表不存在，自然这个SQL语句也就是没意义的了；再如果一个表的schema信息中不存在SQL语句中涉及的字段名，则说明该字段不存在；再甚至，如果两张表的shcema信息中都存在相同的字段名，而某条SQL语句中同时涉及了这两张表和相应的字段名，这就是有歧义（译者注：在SimpleDB中，暂不支持tblName.fieldName的格式，而在实际数据库中这是支持的）；此外，planner还会检查schema中的具体类型和长度来做字段类型匹配。

类型匹配这个工作需要在谓词、修改赋值和插入SQL语句中完成。对于谓词，一个表达式中的每个操作符都必须是类型兼容的，并且每个项左右两边的表达式也必须是类型兼容的；对于一个修改SQL语句，表达式的实际取值也必须和字段的类型兼容；插入SQL语句也类似。

SimpleDB中的planner可以从元数据管理器中获取到表的schema信息，通过调用 `getTableInfo()` 方法。然而，**SimpleDB目前的实现中并没有执行任何的验证**，练习19.4-19.8则会要求你完成这些验证。

## 19.3 Query Planning

我们现在来考虑一下这个问题，在得到了parser返回的 `XXXData` 对象后，怎么构造一个对应的plan呢？也就是之前提到的planning算法问题。这一小节中，我们先考虑一下对于query，怎么创建对应的plan；而在下一小节再考虑update query 的情况。

### 19.3.1 SimpleDB的query planning算法

SimpleDB只支持SQL语句中的一部分，具体来说，SimpleDB不支持包含具体计算的SQL，不支持sort，不支持group by，不支持子查询，也不支持重命名。于是，除去上述的不支持的特性后，SimpleDB支持的SQL语句则可以只用 `select`，`project` 和 `product` 三种关系运算来构建一棵查询树，具体的算法如下：

#### SimpleDB的query planning算法

1. 为 `from` 语句中的每个表T创建一个plan;
  - a) 如果T是一个实际存储的表，则为该表创建一个的Table plan;
  - b) 如果T是一个视图，那么该视图对应的plan对象则是，递归调用定义该
2. 按照 `from` 语句中给定表名的顺序，依次求 `product` 操作后的结果；
3. 按照 `where` 语句中给定的谓词，选择满足谓词的记录；
4. 按照 `select` 语句中给定的字段名， `project` 出结果。

图19-5和图19-6中给出了两个对应上述query planning算法的图示，两个图中对应query的功能都是检索出那些上过Einstein教授的课并且成绩为'A'的同学的名字。

图19-5 (a) 中对应的SQL语句并没有使用视图，因此，它对应的查询树如图19-5 (b) 中所示，注意一下这个查询树中 `product` 操作的顺序，然后再执行相应的 `select` 和 `project` 操作。

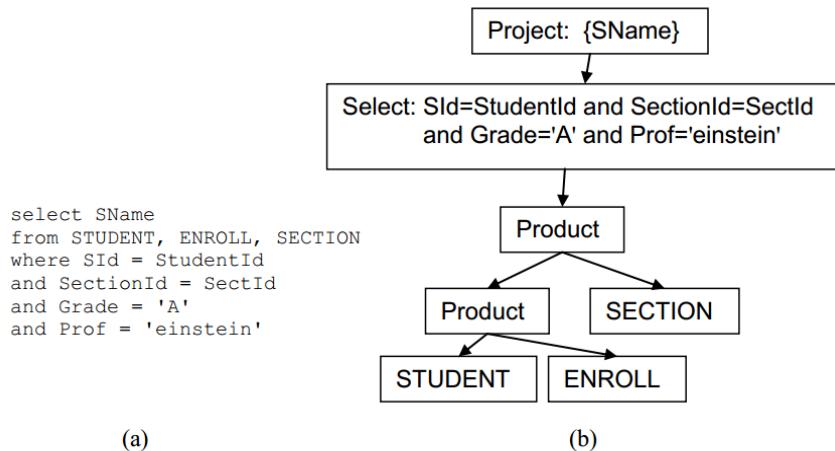


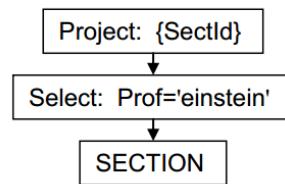
Figure 19-5: Applying the basic query planning algorithm to an SQL query

图19-6 (a) 中对应的SQL语句使用了视图, 图19-6 (b) 中展示的是递归调用该query planning算法对应视图的查询树, 而图19-6 (c) 中展示了最终的查询树。注意一下, 这种情况下, 会先对前两张表作product操作, 然后再将product的结果和view再次product, 随后再执行相应的select和project操作。也请注意一下图19-5(c)和图19-6 (c) 中的不同之处, 其实也就是, 之前的select中的一部分谓词的操作被“下降了”, 并且图19-6 (c) 中的查询树多了一个project操作。诸如此类的等效条件对于查询优化非常重要, 我们将在第24章中讨论这些细节。

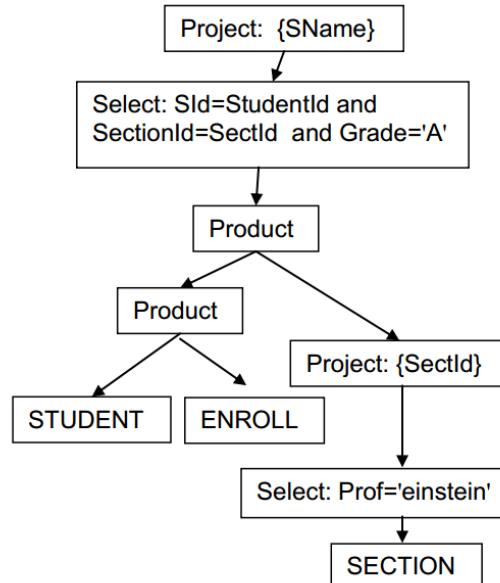
```
create view EINSTEIN as
select SectId
from SECTION
where Prof = 'einstein'

select SName
from STUDENT, ENROLL, EINSTEIN
where SId = StudentId
and SectionId = SectId
and Grade = 'A'
```

(a) The SQL query



(b) The tree for the view



(c) The tree for the entire query

Figure 19-6: Applying the basic query planning algorithm in the presence of views

这种最基本的planning算法相对来说十分朴素, 也就是简单地按照 `QueryData.tables` 中的表来执行product操作, 并且在这里的 product操作的顺序是无关紧要的 (译者注: 其实还是有一点影响的, 具体可以参考17.5.4小节中的讨论), 但是结果肯定是等效的。目前来说, 这种朴素的planning算法效率并不是很高, 因为我们并没有做出优化, 创建最低代价的plan。商业数据库系统中的planning算法远远比我们这里的算法复杂得多, 这些复杂的算法中不仅会分析各个等效的plan的代价, 而

且也实现了其他的关系代数操作，并且这些内容往往就是一个商业数据库系统能否超越同类型产品的关键所在，具体的优化技术将在本书的Part 4讨论。

SimpleDB中的类 BasicQueryPlanner 实现了当前这种朴素的query planning算法，代码中4个步骤都对应我们之前讨论的4步，代码如下：

```
// QueryPlanner.java

public interface QueryPlanner {
    public Plan createPlan(QueryData queryData, Transaction tx);
}

// BasicQueryPlanner.java
public class BasicQueryPlanner implements QueryPlanner {

    @Override
    public Plan createPlan(QueryData queryData, Transaction tx) {
        // 步骤1：对每个表或视图，创建一个plan
        List<Plan> plans = new ArrayList<>();
        for (String tblName : queryData.getTables()) {
            String viewDef = SimpleDB.metadataMgr().getViewDef(tblName);
            if (null != viewDef)
                plans.add(SimpleDB.planner().createQueryPlan(queryData, viewDef));
            else
                plans.add(new TablePlan(tblName, tx));
        }

        // 步骤2：依次做product操作
        Plan p = plans.remove(0);
        for (Plan nextPlan : plans) {
            p = new ProductPlan(p, nextPlan);
        }

        // 步骤3：根据where部分的谓词筛选
        p=new SelectPlan(p,queryData.getPred());

        // 步骤4：根据select部分做project操作
        p=new ProjectPlan(p,queryData.getFields());

        return p;
    }
}
```

### 19.3.2 标准SQL query的planning算法

SimpleDB中支持的SQL语句可以用一棵只包含3种关系代数操作的查询树来表示，但是在标准的SQL语句中，则包含更多的特性，自然也就包含更多的关系代数操作：

- 子查询需要使用 `semijoin` 和 `antijoin` 操作；
- 聚集(aggregations)操作需要使用 `groupby` 操作；
- 重命名和计算值需要使用 `extend` 操作；
- 外联结需要 `outerjoin` 操作；
- 求并集需要 `union` 操作；
- 排序需要 `sort` 操作；

SQL查询的语义要求运算符以特定顺序出现在查询树中（暂不考虑视图创建的子树）。对于SimpleDB查询，我们看到`product`操作位于查询树的底部，然后是单个`select`操作，最后是`project`操作。在标准SQL中，各操作的顺序如下，自底向上依次是：

- `product`操作；
- `outerjoin`操作；
- `select, semijoin`和`antijoin`操作；
- `extend`操作；
- `project`操作；
- `union`操作；
- `sort`操作；

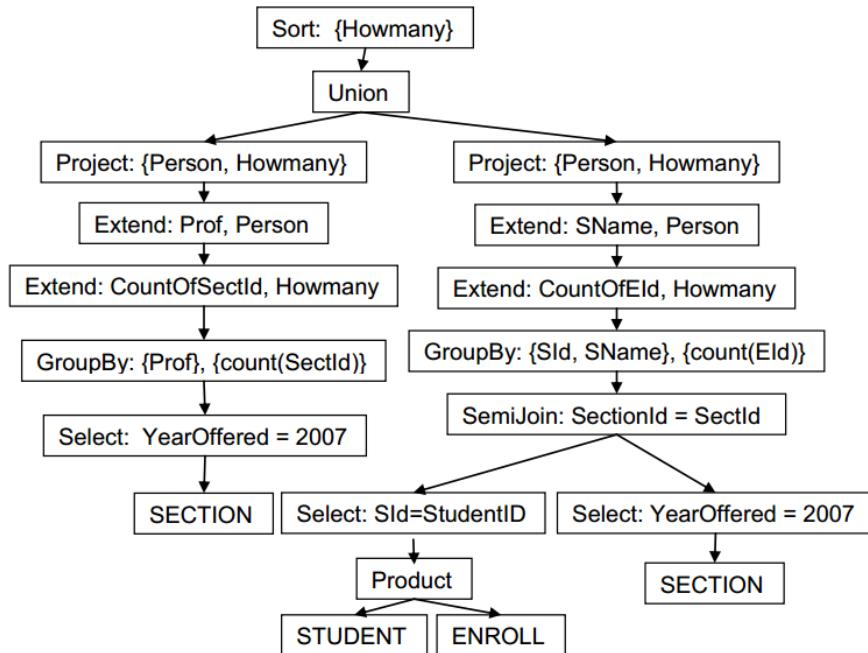
例如，考虑下面图19-8 (a) 中的SQL查询，其功能是返回所有在2007年教过课或者上过课的人的信息。`union`的第一部分返回的是所有在2007年上过课的学生，第二部分返回的是所有在2007年授过课的教授，并且`union`的结果会按照课程的编号排序。图19-8 (b) 中展示了这个复杂的SQL对应的查询树，这时还没有考虑任何有关效率的操作。

```

select SName as Person, count(EId) as Howmany
from STUDENT, ENROLL
where SID = StudentId
and SectionId in (select SectId
                  from SECTION
                  where YearOffered = 2007)
group by SID, SName
union
select Prof as Person, count(SectId) as Howmany
from SECTION
where YearOffered = 2007
group by Prof
order by Howmany

```

(a) A complex SQL query



(b) The query tree produced by the basic planning algorithm

Figure 19-8: Extending the basic query planner to include other operators

为了使SimpleDB能够生成这样的查询树，需要扩展SimpleDB中的解析器，以便从SQL中提取出必要的其他信息。其中一些附加信息来自SQL语句中的其他部分，例如order by和group by子句。在这种情况下，将需要修改的句法规则从而包括附加语法类别。相应地，也要修改 `Lexer` 类中的内容。

SimpleDB中的planner也将需要作出相应的扩展以使用parser提取的新信息。planning算法通过首先生成product节点，然后生成outerjoin节点等等，从而自底向上地构建一棵查询树。但是，在这种复杂的情况下，planner需要主动去搜索parser解析出来的信息。例如，select子句可以适用于groupby子句包含的信息，planner必须主动去判断是否存在groupby信息，以确定其中的哪一部分适用于哪个操作。

## 19.4 Update Planning

我们现在考虑一下对于update query, planner又应该怎么做。SimpleDB 中的 `BasicUpdatePlanner` 类也提供了一个最简单直接的实现，对于每种update操作（增、删、改），都包含一个相应的方法，具体的代码如下：

```

// UpdatePlanner.java
public interface UpdatePlanner {
    public int executeInsert(InsertData insertData, Transaction tx);
    public int executeDelete(DeleteData deleteData, Transaction tx);
    public int executeModify(ModifyData modifyData, Transaction tx);

    public int executeCreateTable(CreateTableData data, Transaction tx);
    public int executeCreateView(CreateViewData data, Transaction tx);
    public int executeCreateIndex(CreateIndexData data, Transaction tx);
}

// BasicUpdatePlanner.java
public class BasicUpdatePlanner implements UpdatePlanner {
    @Override
    public int executeInsert(InsertData insertData, Transaction tx) {
        Plan p = new TablePlan(insertData.getTblName(), tx);
        UpdateScan scan = (UpdateScan) p.open();
        scan.insert();
        // 插入的新记录各字段的取值
        Iterator<Constant> iter = insertData.getVals().iterator();
        for (String fieldName : insertData.getFields()) {
            Constant val = iter.next();
            scan.setVal(fieldName, val);
        }
        scan.close();
        return 1;
    }

    @Override
    public int executeDelete(DeleteData deleteData, Transaction tx) {
        Plan p = new TablePlan(deleteData.getTblName(), tx);
        p = new SelectPlan(p, deleteData.getPred());
        UpdateScan scan = (UpdateScan) p.open();
        int cnt = 0;
        while (scan.next()) {
            scan.delete();
            cnt++;
        }
        scan.close();
        return cnt;
    }

    @Override
    public int executeModify(ModifyData modifyData, Transaction tx) {
        Plan p = new TablePlan(modifyData.getTblName(), tx);
        p = new SelectPlan(p, modifyData.getPred());
    }
}

```

```

        UpdateScan scan = (UpdateScan) p.open();
        int cnt = 0;
        while (scan.next()) {
            scan.setVal(modifyData.getFldName(),
                        modifyData.getNewVal().asConstant());
            cnt++;
        }
        scan.close();
        return cnt;
    }

    @Override
    public int executeCreateTable(CreateTableData data, Transaction tx) {
        SimpleDB.metadataMgr().createTable(data.getTblName(),
                                            data.getSchema(),
                                            tx);
        return 0;
    }

    @Override
    public int executeCreateView(CreateViewData data, Transaction tx) {
        SimpleDB.metadataMgr().createView(data.getViewName(),
                                           data.getViewDef(),
                                           tx);
        return 0;
    }

    @Override
    public int executeCreateIndex(CreateIndexData data, Transaction tx) {
        SimpleDB.metadataMgr().createIndex(data.getIndexName(),
                                            data.getTblName(),
                                            data.getFldName(),
                                            tx);
        return 0;
    }
}

```

### 19.4.1 delete & modify planning

对于删除和修改SQL语句，它们对应的scan对象是select scan，它们会先检索到满足谓词的记录，然后再做出相应的删除或修改操作，例如：

```

update STUDENT
set MajorId = 20
where MajorId = 30 and GradYear = 2008

```

和

```
delete from STUDENT
where MajorId = 30 and GradYear = 2008
```

这两条SQL语句对应的scan对象其实是一样的，即都会找到那些专业ID为30并且在2008年毕业的学

生，`BasicUpdatePlanner.executeDelete()` 和 `BasicUpdatePlanner.executeModify()` 方法会相应地遍历scan对象，在遍历的过程中，可以执行对应的删除或修改操作。

仔细看下代码，我们会发现这两个方法都会创建相同的plan，两种方法也都会open这个scan并以类似的方式遍历scan。`executeDelete()` 方法对scan中的每个记录调用`delete`，而 `executeModify()` 对scan中的每条记录的目标字段执行 `setVal()` 操作，当然，因为这个scan对象是select scan，所以在遍历的时候实际上是在遍历满足谓词的记录。这两种方法还会返回受影响记录的条数。

### 19.4.2 Insert planning

插入SQL语句对应的scan对象其实就是一个简单的table scan，`executeInsert()` 方法最开始会调用 `insert()` 方法来找到一个合适的插入位置，注意，调用完这个方法后是还没有实际插入任何数据的。随后会遍历`inserData`的各个字段和对应的取值，真正将数据插入进去，最后返回1，表示插入了1条数据。

### 19.4.3 创建表、视图和索引的planning

方

法 `executeCreateTable()`，`executeCreateView()` 和 `executeCreateIndex()` 的代码很简单，因为它们不需要访问任何数据记录，因此不需要一个scan。他们使用parser得到的信息，简单地调用元数据管理器的方法 `createTable()`，`createView()` 和 `createIndex()` 来完成工作，它们的返回值为0，表示没有记录受到影响。

## 19.5 实现SimpleDB的planner

回顾一下19.1.1小节中类 `Planner` 的API，其实也就是两个方法 `createQueryPlan()` 和 `executeUpdate()` 方法，我们现在来考虑一下他们的实现，代码如下：

```

public class Planner {
    private QueryPlanner queryPlanner;
    private UpdatePlanner updatePlanner;

    public Planner(QueryPlanner queryPlanner, UpdatePlanner updatePlanner) {
        this.queryPlanner = queryPlanner;
        this.updatePlanner = updatePlanner;
    }

    public Plan createQueryPlan(String query, Transaction tx) {
        Parser parser=new Parser(new Lexer(query));
        QueryData queryData=parser.query();
        //===== TODO =====
        // 验证SQL语句语义正确性的代码
        //===== TODO =====
        return queryPlanner.createPlan(queryData,tx);
    }

    public int executeUpdate(String cmd, Transaction tx) {
        Parser parser=new Parser(new Lexer(cmd));
        Object obj=parser.updateCmd();
        //===== TODO =====
        // 验证update SQL语句语义正确性的代码
        //===== TODO =====
        if(obj instanceof InsertData)
            return updatePlanner.executeInsert((InsertData)obj);
        else if(obj instanceof DeleteData)
            return updatePlanner.executeDelete((DeleteData)obj);
        else if(obj instanceof ModifyData)
            return updatePlanner.executeModify((ModifyData)obj);
        else if(obj instanceof CreateTableData)
            return updatePlanner.executeCreateTable((CreateTableData)obj);
        else if(obj instanceof CreateViewData)
            return updatePlanner.executeCreateView((CreateViewData)obj);
        else if(obj instanceof CreateIndexData)
            return updatePlanner.executeCreateIndex((CreateIndexData)obj);
        else
            return 0;
    }
}

```

这些方法的实现也都很简单直接，执行的流程为，先将输入的字符串类型的SQL语句用Parser解析得到相应的 XXXData 对象（当然，parser也会相应地调用Lexer类中的方法），然后再调用QueryPlanner或者UpdatePlanner对象的相关方法即可。稍微注意一下 executeUpdate() 方法，因为Update类型的SQL包括增删改语句，所以parser解析的对象可能是多种类型的，需要将各个情况都分别处理一下。

译者注：还有一个地方需要注意一下，这也正是面向对象编程的一个最大的优势，设计时只要设计出接口，具体的实现可以多种多样，我想这也是为什么面向接口编程的思想大行其道的原因。我们现在的planning算法（包括query planning和update planning）是最基本的朴素实现，我们现在实际上没有作SQL语义验证和等效plan的代价分析，以后我们要加上这些特性的话，只要新建具体的实现了QueryPlanner接口和UpdatePlanner接口的类即可，而Planner类的代码无需做任何改变。（在第24章中，将实现一个HeuristicQueryPlanner类），接口的功能就是为了实现代码的可拔插性（plug-and-play capability）。

## 19.6 章末总结

- planner是数据库系统中负责将一个字符串类型的SQL语句转化为具体的plan的组件。
- planner需要通过检查下面的信息，来验证SQL语句的语义：
  1. SQL语句中的表和字段名是否已经存在于catalog中；
  2. SQL语句中的字段是否有歧义；
  3. SQL语句中涉及字段的操作是否是类型匹配；
  4. SQL语句中涉及的常量是否和对应字段的类型及大小匹配。
- 最朴素的query planning算法的流程如下：

1. 为from语句中的每个表 $\{ \% \text{ math } \% \} T \{ \% \text{ endmath } \% \}$ 创建一个plan;
  - a) 如果 $\{ \% \text{ math } \% \} T \{ \% \text{ endmath } \% \}$ 是一个实际存储的表，则为该表创建一个plan；
  - b) 如果 $\{ \% \text{ math } \% \} T \{ \% \text{ endmath } \% \}$ 是一个视图，那么该视图对应的plan就是从视图对应的表上创建的。
2. 按照from语句中给定表名的顺序，依次求product操作后的结果；
3. 按照where语句中给定的谓词，选择满足谓词的记录；
4. 按照select语句中给定的字段名，project出结果。

- 商业数据库系统中都存在分析各种等效plan执行代价的组件，这往往也是一个商业数据库系统的竞争力所在，我们将在第24章中讨论这些细节。
- SimpleDB中只支持SQL语句中的一小部分，不支持包含具体计算表达式的SQL，不支持sort，不支持group by，不支持子查询，也不支持重命名。于是，除去上述的不支持的特性后，SimpleDB支持的SQL语句则可以只用select, project 和 product三种关系运算来构建一棵查询树。
- 对应删除和修改记录方法的实现很类似，planner都会先创建一个select scan，然后遍历所有满足谓词的记录，并对遍历过程中的所有记录执行相应的删除或修改字段操作。
- 插入SQL语句的实现相对来说比较简答，planner只要创建一个underlying表的TableScan对象，找到一个插入位置，然后把各字段

的值插入即可。

- 创建表、视图和索引时不需要访问任何记录（这里值的是非catalog表的记录），因此不需要一个scan。只要使用parser解析得到的信息，然后简单地调用元数据管理器的方法 `createTable()`，`createView()` 和 `createIndex()` 来完成工作，它们的返回值为0，表示没有记录受到影响。

## 19.7 建议阅读

## 19.8 练习

## 第20章 数据库服务器和 包 simpledb.remote

在本章中，我们将会研究一下客户端程序访问数据库系统的两种不同的方式：

- 数据库系统可以被嵌入在客户端中；
- 可以作为一个服务端独立运行，并接受客户端的请求。

嵌入式的数据库策略不需要再写太多的新的代码，但是应用能力有限（译者注：可以试想一下，如果真的是这种嵌在一起的策略，为什么要当初要写并发的代码呢？）；然而，基于C/S架构的策略允许多个用户并发地共享数据库，但是这样就要求开发者去实现数据库服务端的代码，并且处理JDBC请求。我们接下来将会看到如何用Java的RMI来构建一个数据库服务端，并且实现一个JDBC driver。

### 20.1 服务型数据库 VS 嵌入型数据库

一个应用程序访问数据库的方式有两种：

- 通过服务器间接访问数据库；
- 直接访问数据库。

本书的第2部分主要关注的正是上述第一种方式。这种思路是说，应用程序首先调用drive中的相关方法来和数据库服务器来建立连接；然后发送一条或多条SQL语句给服务端从而访问数据库中的数据。

然而上述第二种方式，应用程序会直接调用数据库的方法，这种方法只适合于数据库系统暴露了API的情况。在这种情况下，我们说这个应用程序把一个数据库系统嵌入了其中。

一个嵌入型的数据库的例子如下所示，代码中创建了一个新的事务，调用planner的相关方法来为一个SQL查询创建一个plan对象，获得plan对象后，我们可以调用这个对象的open()方法来获得一个scan对象，再迭代这个scan对象，我们可以得到每条具体的数据，最后提交这个事务即可。

```

SimpleDB.init("studentdb");
Transaction tx = new Transaction();

Planner planner = SimpleDB.planner();
String qry = "select sname, gradyear from student";
Plan p = planner.createQueryPlan(qry, tx);
Scan s = p.open();

while (s.next())
    System.out.println(s.getString("sname") + " " +
                       s.getInt("gradyear"));

s.close();
tx.commit();

```

相反，一个服务型的数据库的示例代码则如下所示。

```

Driver d = new SimpleDriver();
String url = "jdbc:simpledb://localhost";
Connection conn = d.connect(url, null);

Statement stmt = conn.createStatement();
String qry = "select sname, gradyear from student";
ResultSet rs = stmt.executeQuery(qry);

while (rs.next())
    System.out.println(s.getString("sname") + " " +
                       s.getInt("gradyear"));

rs.close();
conn.commit();

```

我们可以看到，嵌入型的数据库代码中使用了SimpleDB中的5个类：SimpleDB，Transaction，Planner，Plan 和 Scan；而服务型的数据库代码中则使用了接口Driver，Connection，Statement 和 ResultSet。其实这些类和接口是存在着一种如图20-2所示的对应关系的，表中也展示了ResultSetMetaData 接口和 Schema 类之间的对应关系。

JDBC Interface	SimpleDB Class
Driver	SimpleDB
Connection	Transaction
Statement	Planner, Plan
ResultSet	Scan
ResultSetMetaData	Schema

Figure 20-2: The correspondence between JDBC interfaces and SimpleDB classes

上图表中的每一行都表示二者之间有一种共同的目的。例如， Connection 和 Transaction 都是用来管理事务的；而 Statement 和 Planner , Plan 都是来处理SQL语句的；而 ResultSet 和 Scan 都是来迭代查询结果的。通过这种对应关系，任何一个JDBC程序都可以被重构成等效的嵌入数据库的程序（当然，我们并不会这么做）。

使用嵌入型的数据库系统的程序不会（也不能）与其他程序共享其数据库。嵌入了数据库的应用程序对数据库是私有的。应用程序中没有其他用户，也没有其他线程（除非程序自己创建线程，如图14-30所示）。实际上，如果两个程序共享同一个嵌入式数据库，则没有什么方法可以阻止数据库被损坏——这些程序无法控制对方做出的修改，因为每个程序都管理着自己的 锁表(Lock Table)。

所以，这种嵌入型的数据库系统只对一些特定的应用有意义，这种系统一般是：

- 无法轻易地连接到数据库服务器，或者；
- 排斥其他用户访问数据(即只希望自己拥有数据)

对应上述第一条的最好的例子就是汽车的GPS导航系统了，一个GPS应用程序会访问数据库中的道路信息和位置信息。对于移动的汽车来说，连接到数据库服务器是很慢并且不稳定的，因此，一个嵌入型的数据库系统才更有可能运行应用程序顺利地操作并且可靠地访问数据。

而对应上述第二条的一个例子就是传感器控制系统，例如核反应堆中的程序。这种系统周期地从多个传感器中得到数据，并且将这些数据保存到数据库中，并进行相应的数据分析。这些传感器数据往往都是与其他的控制系统分离的，因此不需要设计一种服务型的数据库。

使用嵌入型数据库系统的一个后果是，应用程序必须在本地运行数据库代码，运行速度可能比服务型数据库慢得多。然而，嵌入型数据库系统可以进行自定义，从而允许用户关闭不需要的功能。例如，再次考虑GPS导航系统，该系统是只读的，不会运行并发线程，因此，它不需要恢复管理(recovery manager)，并发控制器(concurrency control)或update planner组件。通过从嵌入型数据库中删除这些功能，GPS系统变得更苗条，更快，甚至比在高端计算机上使用功能齐全的共享数据库服务器时还要快。

## 20.2 客户端-服务端通信

我们现在着重考虑如何实现服务型数据库，要实现数据库服务器，必须解决两个问题：

- 客户端和服务端之间怎样通信？
- 怎样实现JDBC接口？

在这个小节中，我们将解决第一个问题；而第二个问题将在20.3小节中讨论。

### 20.2.1 远程方法调用

SimpleDB中通过Java中的远程方法调用(Remote Method Invocation,RMI)来实现C/S间通信。为了使用RMI，系统必须定义一些接口，这些接口继承了JavaRMI中的接口 `Remote`，这些接口被称为远程接口(remote interface)。对于每个远程接口，都会对应有两个实现类：客户端上的 存根(stub) 类和服务端上的实现类。当客户端调用一个存根对象的相关方法时，这个方法调用会通过网络传递给服务端，服务端上的对应的实现类对象会执行相关方法，然后服务端会将执行方法返回的对象送回给客户端上的存根对象。关于更多Java RMI相关的知识，可以去网上查找相关资料，总而言之，一个远程方法是被客户端上的存根对象调用的，而实际是在服务端中的实现类对象完成的。

SimpleDB中定义了5个远程接口，分别是 `RemoteDriver`，  
`RemoteConnection`，`RemoteStatement`，  
`RemoteResultSet` 和 `RemoteMetaData`，定义这些接口的代码如下所示。这些远程接口反映的是其对应的JDBC接口，但是有两个区别：

- 只实现了基本的JDBC方法；
- 抛出的异常是`RemoteException`（这是RMI要求的），而不是抛出`SQLException`（这是JDBC的要求）

```

public interface RemoteDriver extends Remote {
    public RemoteConnection connect() throws RemoteException;
}

public interface RemoteConnection extends Remote {
    public RemoteStatement createStatement() throws RemoteException;
    public void close() throws RemoteException;
}

public interface RemoteStatement extends Remote {
    public RemoteResultSet executeQuery(String qry) throws RemoteException;
    public int executeUpdateCmd(String cmd) throws RemoteException;
}

public interface RemoteResultSet extends Remote {
    public boolean next() throws RemoteException;
    public int getInt(String fieldName) throws RemoteException;
    public String getString(String fieldName) throws RemoteException;
    public RemoteMetaData getMetaData() throws RemoteException;
    public void close() throws RemoteException;
}

public interface RemoteMetaData extends Remote {
    public int getColumnCount() throws RemoteException;
    public String getColumnName(int column) throws RemoteException;
    public int getColumnType(int column) throws RemoteException;
    public int getColumnDisplaySize(int column) throws RemoteException;
}

```

为了对RMI是如何工作的有个大概的了解，请考虑下面这样一段客户端的代码片段：

```

RemoteDriver rdvr = ...
RemoteConnection rconn = rdvr.connect();
RemoteStatement rstmt = rconn.createStatement();

```

上述代码片段中的每个变量都表示一个接口。但是，由于代码片段是运行在客户端上的，因此我们知道这些变量持有的实际对象来自存根类。请注意，该代码片段中未显示变量 `rdvr` 是如何获得其存根的。我们在这里先暂时搁置一下这个问题，直到后续讨论RMI注册时，我们再来讨论这个问题。

我们考虑一下 `rdvr.connect()` 这条语句，存根通过网络，将请求发送到服务器上相应的RemoteDriver实现类对象。远程实现类对象会在服务器端执行它自己的 `connect()` 方法，相应地，执行完 `connect()` 方法后

会返回一个新的 `RemoteConnection` 实现类对象，而此远程对象的存根则被发送回客户端，客户端将其存储为变量 `conn` 的值。

再考虑一下 `rconn.createStatement()` 这条语句。存根对象会将请求发送给服务器上与该存根对象对应的 `RemoteConnection` 实现类对象。该远程对象执行其 `createStatement()` 方法，并在服务器上创建一个 `RemoteStatement` 实现类对象，并再次将其存根返回给客户端。

## 20.2.2 RMI注册

客户端上的每个存根对象都包含了一个对应服务端远程实现对象的引用。一个客户端，一旦有了一个存根对象，就可以通过该存根对象来和服务端进行交互，交互的过程中有可能还会创建新的存根对象，以供客户端使用。但是，问题是，第一个存根对象是怎么获取到的呢？RMI中通过一个叫做RMI注册的程序来解决这个问题。

服务端通过RMI注册表来发布存根对象，客户端可以在RMI注册表中检索得到存根对象。

SimpleDB中的服务端只发布一个对象，这个对象是 `RemoteDriver` 类型的。通过执行 `simplesdb.server.Startup` 中的下面两行代码，服务器发布了一个存根对象：

```
RemoteDriver d = new RemoteDriverImpl();
Naming.rebind("simplesdb",d);
```

`Naming.rebind()` 方法会为远程实现对象 `d` 创建一个存根，并把这个存根保存在RMI注册表中，并且绑定的名称为 `simplesdb`。

一个客户端可以通过 `Naming.lookup()` 方法来请求获得注册表中的存根，在SimpleDB中，这个请求是通过 `SimpleDriver` 类（后面会讲这个类的实现）中下面两行代码来完成的：

```
String newUrl = url.replace("jdbc:simplesdb","rmi")+"/simplesdb";
RemoteDriver rdvr = (RemoteDriver) Naming.lookup(newurl);
```

举例来说，如果变量 `url` 的值为 `jdbc:simplesdb://localhost`，随后，`newUrl` 的值将会变为 `rmi://localhost/simplesdb`，这个新的字符串是为了符合RMI注册表的要求。紧接着，调用 `Naming.lookup()` 方法会根据指定的host(这里是localhost)去RMI注册表中寻找相应名为 `simplesdb` 的存根对象，在找到后把这个对象返回给调用者。

## 20.2.3 线程问题

在构建一个相对来说比较大型的Java程序的时候，如果我们能对程序中线程的执行情况有个清楚的了解，那肯定是再好不过了。在基于C/S架构的SimpleDB中，程序中会存在两类的线程：在客户端上的线程和在服务端上的线程。

每个客户端在其计算机上都有自己的线程。该线程在客户端执行期间持续存在，该线程调用所有客户端的存根对象。另一方面，服务器上的每个远程对象都在其自己的单独线程中执行。可以将服务器端远程对象视为“微型服务器”，它会一直等待其存根建立连接。建立连接后，远程对象将执行请求的工作，将返回值发送回客户端，并耐心等待另一个连接。

在 `simpledb.server.Startup` 中创建的 `RemoteDriver` 对象在一个线程中运行，该线程可以被视为“数据库服务”线程。

每当客户端进行远程方法调用时，客户端线程将会在服务器线程运行期间等待，并在服务器线程返回值时恢复运行。同样，服务器端线程将处于休眠状态，直到服务端的某个方法被请求调用，服务端也会在该方法执行完成后恢复休眠状态。因此，在任何给定时间，这些客户端和服务器线程中只有一个线程实际在做一些事情。打个不太准确的比方，请求了远程方法调用后，客户端线程似乎是在客户端和服务器之间来回移动。尽管这种想象可以帮助你更好地理解客户端-服务器应用程序中的控制流，但了解真正的实际情况也更重要。

区别客户端和服务端线程的一个办法就是打印一些东西，你可以用 `System.out.println` 来得知当前是哪个线程在执行。

## 20.3 实现远程接口

对于每个远程接口，都需要实现两个类：存根类和远程实现类。习惯上，远程实现类的类名为接口名加上 `Impl` 后缀，而存根类的类名则为接口名再加上 `Impl_Stub` 后缀。

幸运的是，对于所有的远程接口来说，客户端和服务器端对象之间的通信都是以一种相同的方式，这意味着所有通信代码都可以由RMI库类提供。应用程序员只需要提供每个接口的相关代码即可。换句话说，程序员根本不需要编写存根类的代码，而只需编写远程实现类的代码，也就是明确服务器在每个方法被调用时所执行的操作。在本节中，我们将描述 SimpleDB 中每个远程接口的实现。

### 20.3.1 `RemoteDriverImpl` 类

`RemoteDriverImpl` 类是服务端的入口点，其代码如下所示，只有一  
个 `RemoteDriverImpl` 类的对象会被创建，而创建的过程是  
在 `simpledb.server.Startup` 这个启动类中完成的，并且该远程实现  
对象对应的存根会被发布在RMI注册表中，这个存根对象也是唯一的。每  
次该远程实现对象的 `connect()` 方法被调用时（通过存根对象），将会

在服务端创建一个新的 `RemoteConnection` 远程对象，并且这个新的对象会在一个新的线程中执行。RMI会透明地创建对应的 `RemoteConnection` 存根对象，并返回给客户端。

```
public class RemoteDriverImpl extends UnicastRemoteObject
    implements RemoteDriver {

    public RemoteDriverImpl() throws RemoteException {
    }

    @Override
    public RemoteConnection connect() throws RemoteException {
        return new RemoteConnectionImpl();
    }
}
```

注意一下，代码十分简单，我们只需要关注服务端的对象创建即可，特别是，代码中不包括任何网络相关或者存根对象相关的代码，并且当需要创建新的远程对象时，服务端只要负责创建好远程实现对象就可以了，根本不需要管存根对象是怎么创建的。RMI中的 `UnicastRemoteObject` 类包含了其他有关连接和存根相关的所有代码。

### 20.3.2 `RemoteConnectionImpl` 类

`RemoteConnectionImpl` 类维护了客户端和服务端的连接，代码如下所示。对于每个客户端连接，都会有一个相关的 `RemoteConnectionImpl` 对象运行在服务器上，并且是在一个独立的线程中运行的。这个对象负责管理事务，也会负责为客户端创建查询语句，这个对象的大多数工作都是通过 `Transaction` 类的对象来完成的。

```

public class RemoteConnectionImpl extends UnicastRemoteObject
    implements RemoteConnection {
    private Transaction tx;

    public RemoteConnectionImpl() throws RemoteException {
        this.tx = new Transaction();
    }

    @Override
    public RemoteStatement createStatement() throws RemoteException {
        return new RemoteStatementImpl(this);
    }

    @Override
    public void close() throws RemoteException {
        tx.commit();
    }

    //=====以下方法都是在服务端中调用的=====
    Transaction getTrasnaction()
    {
        return tx;
    }
    void commit()
    {
        tx.commit();
        tx=new Transaction();
    }
    void rollback()
    {
        tx.rollback();
        tx=new Transaction();
    }
}

```

上面代码中包含了一些package-private的方法，即 `getTransaction()` , `commit()` , `rollback()` 方法，这些方法并不是远程接口中的方法，因此也不可能被客户端调用。这些方法是被服务端中的 `RemoteStatementImpl` 和 `RemoteResultSetImpl` 类来使用的。

### 20.3.3 RemoteStatementImpl 类

`RemoteStatementImpl` 类是负责执行SQL语句的，代码如下所示。方法 `executeQuery()` 会根据SQL语句从planner组件中获取到一个plan，并且把这个plan传递给 `RemoteResultSet` 对象去执行，`executeUpdate()` 方法执行的动作类似。

```

public class RemoteStatementImpl extends UnicastRemoteObject
    implements RemoteStatement {
    private RemoteConnectionImpl rconn;

    public RemoteStatementImpl(RemoteConnectionImpl rconn)
        this.rconn = rconn;
    }

    @Override
    public RemoteResultSet executeQuery(String qry) throws
        try {
            Transaction tx = rconn.getTrasnaction();
            Plan plan = SimpleDB.planner().createQueryPlan(qry);
            return new RemoteResultSetImpl(plan, rconn);
        } catch (IOException e) {
            rconn.rollback();
            System.out.println("Error in RemoteStatementImpl");
            return null;
        }
    }

    @Override
    public int executeUpdateCmd(String cmd) throws RemoteException {
        try {
            Transaction tx = rconn.getTrasnaction();
            int result = SimpleDB.planner().executeUpdate(cmd);
            rconn.commit();
            return result;
        } catch (IOException e) {
            rconn.rollback();
            System.out.println("Error in RemoteStatementImpl");
            return -1;
        }
    }
}

```

以上两个方法还负责了JDBC的自动提交，如果SQL语句正确执行，则必须commit，一旦SQL语句完成后，方法 executeUpdate() 会执行当前的连接的 commit() 方法。另一方面， executeQuery() 方法则无法立即 commit，因为其结果集仍会被使用，因此，事务的commit动作会被延迟到结果集close的时候。

当然，一旦SQL执行的过程中发生错误，连接必然会立马rollback事务。

#### 20.3.4 RemoteResultSetImpl 类

RemoteResultSetImpl 类包含了执行一个query plan的相关方法，代码如下所示。在构造函数中，会调用该query plan的 open() 方法并得到对应的scan对象。 RemoteResultSetImpl 中的 next() , getInt() , getString() 和 close() 方法都只是简单地调用了scan对象的相应方法。在 close() 方法中也会commit当前事务，从而满足JDBC中自动提交的要求。

```
public class RemoteResultSetImpl extends UnicastRemoteObject
    implements RemoteResultSet {
    private RemoteConnectionImpl rconn;
    private Scan s;
    private Schema sch;

    public RemoteResultSetImpl(Plan plan, RemoteConnectionImpl rconn)
        throws RemoteException {
        try {
            s = plan.open();
            sch = plan.schema();
            this.rconn = rconn;
        } catch (IOException e) {
            System.out.println("Error in RemoteResultSetImpl");
        }
    }

    @Override
    public boolean next() throws RemoteException {
        try {
            return s.next();
        } catch (IOException e) {
            rconn.rollback();
            System.out.println("Error in RemoteResultSetImpl");
        }
        return false;
    }

    @Override
    public int getInt(String fieldName) throws RemoteException {
        // 大小写不敏感
        fieldName=fieldName.toLowerCase();
        return s.getInt(fieldName);
    }

    @Override
    public String getString(String fieldName) throws RemoteException {
        // 大小写不敏感
        fieldName=fieldName.toLowerCase();
        return s.getString(fieldName);
    }

    @Override
    public RemoteMetaData getMetaData() throws RemoteException {
        return new RemoteMetaDataImpl(sch);
    }
```

```
@Override  
public void close() throws RemoteException {  
    try {  
        s.close();  
        rconn.commit();  
    } catch (IOException e) {  
        rconn.rollback();  
        System.out.println("Error in RemoteResultSetImpl");  
    }  
}
```

RemoteResultSetImpl 类中还包含了一个与当前plan对象对应  
的 Schma 对象，这个 Schma 对象存在的意义主要就是，为了供方  
法 getMetaData() 使用，从而可以创建一个 RemoteMetaDataImpl 对  
象。

### 20.3.5 RemoteMetaDataImpl 类

注意一下，这里的MetaData是关于ResultSet的元数据，而不是表的元数  
据。

RemoteMetaDataImpl 类中包含了一个 Schema 类对象，这个对象在构  
造函数中被传进来，代码如下所示。 RemoteMetaDataImpl 类中的方法  
其实和 Schema 类很类似，区别在于，前者是通过列号来获取列的信  
息，而后者是通过列名来检索的。

```

public class RemoteMetaDataImpl extends UnicastRemoteObject
    implements RemoteMetaData {
    private Schema schema;
    private Object[] fieldNames;

    public RemoteMetaDataImpl(Schema schema) throws RemoteException {
        this.schema = schema;
        fieldNames = schema.fields().toArray();
    }

    @Override
    public int getColumnCount() throws RemoteException {
        return fieldNames.length;
    }

    @Override
    public String getColumnName(int column) throws RemoteException {
        return (String) fieldNames[column];
    }

    @Override
    public int getColumnType(int column) throws RemoteException {
        String fieldName = getColumnName(column);
        return schema.type(fieldName);
    }

    @Override
    public int getColumnDisplaySize(int column) throws RemoteException {
        String fieldName = getColumnName(column);
        int fieldType = schema.type(fieldName);
        // 字符串长度
        if (fieldType == VARCHAR)
            return schema.length(fieldName);
        else
            return 6; // 6位数来显示一个整数 (可以不一样)
    }
}

```

## 20.4 实现JDBC接口

SimpleDB中的RMI远程实现类，实现了类似 `java.sql` 包中JDBC的功能，但是和JDBC还是有一定的区别，主要为以下两点：

- 即RMI方法不会抛出SQL异常；
- 基于RMI的实现类实现的是 `RemoteDriver` , `RemoteConnection` , `RemoteXXX` 等接口，并

没有实现JDBC中的 `Driver` , `Connection` 等接口。

这其实是一个在面向对象编程时常遇到的问题，解决办法就是在客户端那一边，将存根对象wrap一下。

为了演示wrapping是怎样工作的，我们先从包装类 `SimpleDriver` 开始讨论。为了更好地管理包装类，我们设计一个额外的抽象类 `DriverAdapter` ,这个适配器类适配了 `Driver` 中的接口，但是具体的实现并没有提供，具体的实现交给 `SimpleDriver` 就好了。`Driver` 类的代码如下：

```
public abstract class DriverAdapter implements Driver {  
    @Override  
    public Connection connect(String url, Properties info)  
        throw new SQLException("operation not implemented")  
    }  
  
    @Override  
    public boolean acceptsURL(String url) throws SQLException  
        throw new SQLException("operation not implemented")  
    }  
  
    @Override  
    public DriverPropertyInfo[] getPropertyInfo(String url,  
        return null;  
    }  
  
    @Override  
    public int getMajorVersion() {  
        return 0;  
    }  
  
    @Override  
    public int getMinorVersion() {  
        return 0;  
    }  
  
    @Override  
    public boolean jdbcCompliant() {  
        return false;  
    }  
  
    @Override  
    public Logger getParentLogger() throws SQLFeatureNotSupportedException  
        throw new SQLFeatureNotSupportedException("operation not implemented")  
    }  
}
```

`SimpleDriver` 类的实现也比较简单，我们这里只实现了 `connect()` 方法，其代码如下：

```
public class SimpleDriver extends DriverAdapter {

    /**
     * 客户端根据指定的url访问数据库，得到一个连接。
     * 首先，我们需要将JDBC格式的url换成RMI格式；
     * 然后，我们通过Naming绑定机制，在RMI注册表中得到RemoteDrive
     * 最后，通过调用RemoteDriver存根对象的相关方法来获取RemoteCo
     *
     * @param url
     * @param info
     * @return
     * @throws SQLException
     */
    @Override
    public Connection connect(String url, Properties info)
        try {
            // 把JDBC格式的url换成RMI格式
            String newURL = url.replace("jdbc:simpledb", "rmi://localhost:1099");
            RemoteDriver rdvr = (RemoteDriver) Naming.lookup(newURL);
            RemoteConnection rconn = rdvr.connect();

            return new SimpleConnection(rconn);
        } catch (Exception e) {
            throw new SQLException(e);
        }
    }
}
```

上述 `connect()` 方法需要返回一个 `Connection` 类型的对象，怎么获取呢？

- 首先，我们需要将JDBC格式的url换成RMI格式；
  - **解释：**因为JDBC要求URL的是形如 `jdbc:xxx` 的格式，其中 `xxx` 表示的是具体的数据库类型，我们这里当然就是 `simpledb` 了，因为SimpleDB的客户端/服务端通信是基于 RMI的方式，所以我们把 `jdbc:simpledb` 替换成 `rmi` 了，这也是RMI通信协议所要求的。
- 然后，我们通过Naming绑定机制，在RMI注册表中得到 `RemoteDriver` 对象的存根；
- 最后，通过调用 `RemoteDriver` 存根对象的相关方法来获取 `RemoteConnection` 对象的存根。

哎呀，这个时候得到的又是一个`RemoteConnection`类型的对象呀，可是`connect()`方法又要求我们返回一个`Connection`类型的对象，怎么办？再包装一下啊！这也是为什么我们在`connect()`方法的最后一行返回的是`new SimpleConnection(rconn)`。

那我们再来看下`SimpleConnection`类是怎么实现的吧！和之前的思路类似，为了更好地管理包装，我们再创建一个`ConnectionAdapter`适配器抽象类，具体的代码就不再贴了，它实现了`java.sql`中的`Connection`接口，并让所有的抽象方法写成`throw new SQLException("operation not implemented")`；即可，具体的实现工作交给`SimpleConnection`类完成，`SimpleConnection`的代码如下：

```
public class SimpleConnection extends ConnectionAdapter {
    RemoteConnection rconn;

    public SimpleConnection(RemoteConnection rconn) {
        this.rconn = rconn;
    }

    @Override
    public Statement createStatement() throws SQLException
        try {
            RemoteStatement rstmt = rconn.createStatement();
            return new SimpleStatement(rstmt);
        } catch (Exception e) {
            throw new SQLException(e);
        }
    }

    @Override
    public void close() throws SQLException {
        try {
            rconn.close();
        } catch (Exception e) {
            throw new SQLException(e);
        }
    }
}
```

其实也就是对`RemoteConnection`的相关方法进行了包装。注意，`createStatement()`方法要求我们返回一个`Statement`类型的对象，而`RemoteConnection`中的`createStatement()`返回的又是`RemoteStatement`类型的对象。所以，我们还要继续包装下去。想必读到这里，你也肯定知道接下来怎么做了！

是时候你自己去创建 StatementAdapter , ResultSetAdapter , MetadataAdapter 以及对应的 SimpleStatement , SimpleResultSet , SimpleMetadata 类了!

## 20.5 单元测试(译者添加)

JDBC的URL格式为 `jdbc:mysql://[host:port], [host:port].../[database]` , 而RMI的URL格式为 `rmi://host:port/repository_name` 。

为了方便对 `remote` 这个包进行单元测试, 我设计了如下的测试用例:

- 假设服务端在RMI registry中注册了一个名为 `simplesdb` 的 `RemoteDriver` 存根对象, 并且已有一个名为 `lzadb` 的数据库, 该数据库中已经存在一张 `student` 表, 其定义为 `student (sid int, sname varchar(5), age int)` , 并且我们已经插入了一条 `(88, 'andy', 22)` 的数据; (如果你忘了怎么创建一个这样的数据库, 请参阅 `simplesdb.planner.PlannerTest.java` 源文件)

```
public class Startup {
    public static void main(String args[]) throws Exception {
        // configure and initialize the database
        SimpleDB.init("lzadb");

        // 在指定端口创建RMI注册表
        Registry reg = LocateRegistry.createRegistry(1099);

        // 服务端访问入口
        RemoteDriver d = new RemoteDriverImpl();
        reg.rebind("simplesdb", d);

        System.out.println("database server ready");
    }
}
```

- 客户端根本不知道底层是基于RMI的通信协议, 完全用JDBC的格式来访问数据库, 但客户端知道数据库服务端端口号为1099 (RMI常见端口号) , 然后客户端去遍历 `student` 表中的所有记录 (其实只要一条)

```

public class ClientTest {
    public static void main(String[] args) throws RemoteException {
        // =====查看本机的rmi registry=====
        //     String host = "localhost";
        //     int port = 1099;
        //     Registry registry = LocateRegistry.getRegistry(host, port);
        //     for (String name : registry.list()) {
        //         System.out.println(name);
        //     }
        // =====查看本机的rmi registry=====

        SimpleDriver driver = new SimpleDriver();
        try {
            Connection conn = driver.connect("jdbc:simpledb");
            Statement stmt = conn.createStatement();

            String qryStr = "select sid,sname,age from student";
            ResultSet rs = stmt.executeQuery(qryStr);

            while (rs.next()) {
                System.out.println(rs.getInt("sid") + " "
                    + rs.getString("sname") + " " +
                    rs.getInt("age"));
            }
            rs.close();
        } catch (SQLException throwables) {
            throwables.printStackTrace();
        }
    }
}

```

先执行 StartUp.java 代码，输出如下：

```

new transaction: 1
recovering the existing database
transaction 1 committed
database server ready

```

再启动 ClientTest.java ，输出如下：

```
88 andy 22
```

完全正确。

此时我们也会发现，服务端也追加了输出，如下：

```
new transaction: 2
transaction 2 committed
new transaction: 3
```

即刚才的事务结束了，为下一个事务分配了新的id——3。你也可以试试别的测试用例，例如创建表

## 20.6 章末总结

- 大致上有两种访问数据库的方式：
  - 通过服务器，间接访问数据库；
  - 直接访问数据集，即嵌入型数据库系统。
- “嵌入型数据库系统”的策略更适合于例如GPS导航系统或传感器系统这样的场景，因为数据在这个系统中是私有的。
- SimpleDB通过Java中的RMI机制来实现客户端和服务端之间的通信。每个JDBC中的接口都存在一个对应的RMI接口，二者的区别主要在于后者抛出的是 `RemoteException` (RMI的规定)，而前者抛出的是 `SQLException` (JDBC的规定)。
- 服务端的每个远程实现类对象都在一个独自的线程中执行，等待客户端存根对象来联系。SimpleDB启动代码会创建一个 `RemoteDriver` 类型的远程实现对象，并把这个对象的存根对象注册到RMI的注册表中。当JDBC客户端想要连接到数据库系统时，客户端会先通过RMI注册表得到这个存根对象。
- `SimpleDriver` 类中的 `connect()` 方法会调用 `RemoteDriver` 类中的 `connect()` 方法来获得一个 `RemoteConnection` 类型的对象，并把它包装成 `Connection` 类型。
- JDBC的客户端不会直接使用存根对象，因为这些存根对象实现的是 `remote` 接口而不是JDBC接口，相反，客户端的对象是对存根对象的包装，例如，`SimpleConnection` 对应 `RemoteConnection`，`SimpleStatement` 对应 `RemoteStatement` 等。

## 20.7 建议阅读

## 20.8 练习