关系型数据库的工作原理

作者：Christophe

发表: August 19, 2015

译者：Franklin Yang（杨伟华）

提起关系型数据库，我就想起一些必须说一说的事情。关系型数据库使用很广泛，而且各种各样：从小巧灵活的 SQLite 到强大威猛的 Teradata。但是，却很少有文章解释数据库的工作原理。你可以自己 google 搜索“how does a relational database work”，会发现几乎没有什么结果，就算有也很简短。如果寻找新潮技术的资料，比如 Big Data、NoSQL 或者 JavaScript，可以找到很多资料而且都有不错的技术深度。

难道因为关系型数据库太老太无聊，除了高校和研究报告，其他人都没兴趣吗？

作为看一个开发者，我讨厌使用我不理解的东西；另外，作为一个在过去四十年持续发挥价值的东西，数据库必然有它独特的价值。过去好几年，我话了许多时间去研究这个一直被当作黑盒的东西。**关系型数据库**非常有意思，因为他们基于一些很有价值并且很重要的理论．如果想理解数据库这个东西，但是却没有时间和愿望去深入研究它，那么这个文章应该对你有点价值。

这个文档的标题很明白，重点不在于解释怎么使用数据库。所以，假设你已经知道了怎么执行简单的连接查询和 CRUD 操作（**译者注：Create、Retrieve、Update、Delete**）；否则的话，你应该理解不了这个文档。这一点你必须知道，其他的事情后面再解释。

我会从计算机科学的一些基本概念，比如时间复杂度出发。我知道有些人讨厌概念和理论，但是没有理论和概念，不可能理解数据库中包含的智慧。因为这是一个大题目，我会把重点放在我认为重要的和根本的：数据库是如何处理一个 SQL 查询的？我这里只探讨数据的基本理论，所以看完本文你应该能够揭开数据库的面纱然后对它们有全面的了解。

毕竟，这是一个不短的技术文章，其中涉及许多算法和数据结构，所以读下来应该要费点时间。有些概念和理论不容易理解，可以跳过去，重点是领会整体的思想。

另外，还有说明一下，这个文章基本可以分为 3 部分：

- 低级和高级数据库组件概述
- 查询优化进程概述
- 事务和 buffer pool 管理概述

目录

译者致歉

我爱好技术，不过工作却跟技术结合得不太紧密（至少主流观点认为是这样）------搞了多年软件测试工作，目前混迹互联网行业，主要搞 Web 方面的测试。

虽然喜欢技术，但是对数据库却不甚了解；回想起七八年前在书店看到那种“InnoDB 存储引擎”之类的书，只能仰望然后找 Linux 之类的东西看看。

囿于技术浅薄，虽然费时费力但是翻译质量必定不好；如果同行有批判和建议，尽管说清道明，我自当洗耳恭听、感激不尽。

五年前的在“某大公司”的一项工作，其中涉及 Oracle 数据库的备份恢复、表空间、表、datafile、flashback、redo log、archived log 等概念；而在那之前我只不过听说过 Oracle 这个“高大上”数据库，幸亏当时领导对我要求很低还能忍受我的无能。刚开始跑去书店站着蹭看“最容易找到的”Oracle 专家“eagle”（大家应该都听说过）的书，发现虽然是中文的，但是前后文的深度和难度没有控制在一个水平上，所以理解起来有点“踉跄”，其他 Oracle 方面的中文书也是这样，明显不适合我这个新手。后来下载了 Oracle 的官方文档，有一个特别初级的文档（2 Day DBA），英文原版就是好，表达很清晰我喜欢看；两三个星期后看完，大约知道了一点点东西，结合公司的测试 Oracle 环境，边想边看边动手。看完了“2 Day”之后，感觉应该再看一个“内容稍微丰富”的东西，然后就下载了“Administrator’s Guide”（Oracle 管理员手册），内容确实丰富，涉及 Oracle 数据库的方方面面，也是边想边看边动手，看完了 3/4 左右吧（后面没有继续，因为公司叫我去搞别的工作了）。其实现在想来，“手册”介绍的知识虽然复杂，不过复杂性更多在于 Oracle 对数据库的实现，其中并没有多少纯粹关于数据库的知识；所以我对数据库这个东西，依然可以说一知半解。

直到去年，工作中涉及一点点 SQL 语句，发现开发同学给我的 SQL 使用了 join，卧槽，闻所未闻！然后学习了两个星期，基本掌握了一些简单的东西。不过，还是觉得这些东西不够深入，除了使用 join（换掉之前的子查询）把 SQL 语句写得很长，其他方面没有差别。

前两周，有幸在微博上看到有人转发这个文档，一看就觉得：这是我必须要看的东西。看完之后，突发奇想：实在太好了，我想翻译！然后费时两个星期的工作间隙时间，基本完成了这个“粗糙劣质”的翻译。

虽然通过这篇文章“给自己扫盲”了，但是工作中依然用不到，不过即使用不到，心里还是踏实了一点点，就一点点！

翻译的不好的地方很多：

0， 有些地方我觉得不应该翻译，比如 Cache、Merge、Access Path、Buffer 和 Log 等。

1， 比如 query executor 翻译成“查询执行”，想过翻译成“查询执行器”，不知所措；另外还有 query rewriter 等。

2， 还有一些算法，其实我还没明白，所以表达肯定不清楚（这种地方我有注释说明）。

3， 文中的图，没有重新画，直接使用原文的图，其中的英文没有翻译（我懒）。

4， 还有自己感觉翻译不地道的地方，保留了翻译和原文（原文使用括号）。

5， 诸如此类。

总而言之，翻译质量保证不了，必须先写这个“致歉”，欢迎拍砖！

如果有人因为看了这个“劣质”翻译，以至于错过了原文，我的罪过大了！

1 基础回顾

很久以前 (就像宇宙那么久远。。。)开发者必须很清楚地了解自己的代码的运算次数，算法和数据结构要牢记在心里，他们得保证自己的代码没有浪费 CPU 和 Memory，因为他们那个年代的电脑真的很慢。

在这部分，我必须给大家介绍一些必须明白的理论知识，关于数据库的非常重要的概念。另外，也会介绍数据库索引。

1.1 O(1) vs O(n^2)

现代，很多开发者根本不关心时间复杂度。。。而且，其实他们没错！

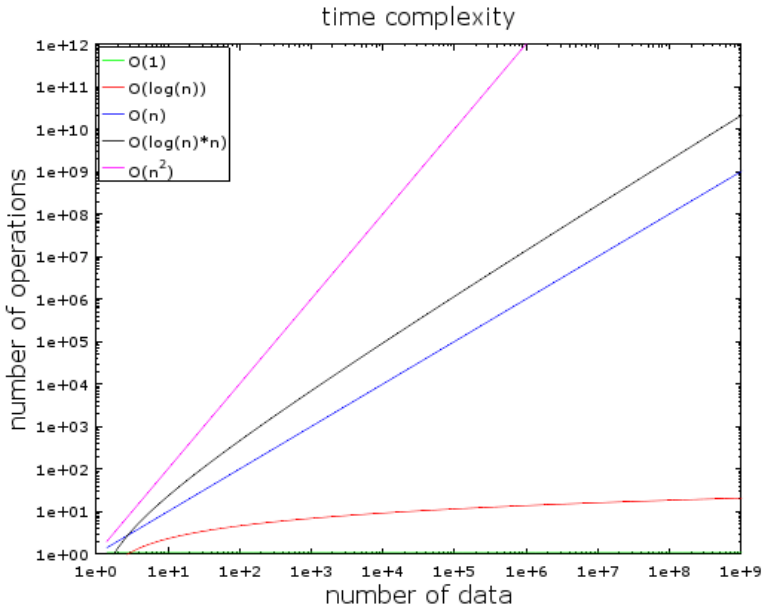
但是当我们面对巨大的数据量（我说的不是千）或者对性能有很高的要求（哪怕 ms 也要争取）时，这些概念就非常重要了。想象一下，数据库必须面对两个情况！我说这个不为别的，就是要大家意识到概念很重要；这个概念可以帮助大家理解 **cost based optimization（基于成本的优化）**理论。

概念

时间复杂度，用来描述某一个算法在给定数据量情况下消耗的时间。表达时间复杂度，计算机科学家使用数学符号 O 表示。这个符号再加一个函数，就可以描述在给定数据量的情况下，某个算法需要完成运算的数量。

比如，当我说“这个算法是 O(F())”时，意味着对于一个特定数量 N 的数据，这个算法需要 F(N) 次运算才能完成。

我们的关注点不应该是数据量，而是数据量增长时，算法运算次数相应增加的方式。时间复杂度不给提供确切的运算次数，而是一个好的想法。



如图所示， 可以看到不同复杂度的变化趋势，这里使用对数增长来衡量它们。换句话说，如果数据量快速地从 1 增加到 1M。我们可以看到：

- 时间复杂度 O(1)或者常数复杂度一直保持同一个值（否则的话它也不能被称作常数复杂度).
- 时间复杂度 **O(log(n))**在 1M 数据量时依然保持在一个较低的值。
- 最坏的时间复杂度是 **$O(n^2)$**，数据量增加时，复杂度瞬间爆表。
- 另外 2 种算法的时间复杂度也是快速增加。

举例

当时数据量不大时，时间复杂度 O(1)和 $O(n^2)$看不出多大差别。比如假设你要使用一个算法处理 2000 个数据：

- 时间复杂度 O(1)的算法，需要 1 次运算
- 时间复杂度 O(log(n)) 的算法，需要 7 次运算
- 时间复杂度 O(n) 的算法，需要 2K 次运算
- 时间复杂度 O(n*log(n)) 的算法，需要 14K 次运算
- 时间复杂度 $O(n^2)$ 的算，需要 4M 次运算

时间复杂度 O(1)和 $O(n^2)$看起来差了 4M 呢，但是时间上只不过多了 2ms，就是你眨眼的差别。但是，实际上，近几年的处理器的运算能力大大提高都能达到 hundreds of millions of operations per second。这就解释了为什么对于很多 IT 项目，性能和优化并不是啥大事。

然并卵，当面对巨量数据要处理时，时间复杂度依然是我们不得不去关注的最重要概念！如果这一次，你要处理 1000000 个数据(其实并不算数据库的数据量级别)：

- 时间复杂度 O(1)的算法，需要 1 次运算
- 时间复杂度 O(log(n)) 的算法，需要 14 次运算
- 时间复杂度 O(n) 的算法，需要 1M 次运算
- 时间复杂度 O(n*log(n)) 的算法，需要 14M 次运算
- 时间复杂度 $O(n^2)$ 的算法，需要 1G 次运算

我没有精确计算，但是粗略估计一下，时间复杂度 $O(n^2)$的算法应该需要喝一杯咖啡的时间。如果数据量增加 10 倍，那你可以去睡一觉咯。

继续深入

总结一些想法：

- 表现良好的哈希表可以让查找的时间复杂度到 O(1)
- 平衡树可以查找的让时间复杂度到 O(log(n))
- 数组的查找，时间复杂度是 O(n)
- 最好的排序算法的时间复杂度可以到 O(n*log(n))
- 最坏的排序算法的时间复杂度是 $O(n^2)$

注意：下一节，我们会探讨一下这些算法和数据结构。

根据具体情况，一个算法在时间复杂度上可以有不同的表现：

- 平均情况
- 最好情况
- 最坏情况

时间复杂度通常表示最坏情况。

这里我们只讨论时间复杂度，其实复杂度还有其他方面：

- 算法的内存消耗
- 算法的 I/O 消耗

不过呢，其实时间复杂度还有更坏的，比 n^2 还要坏：

- n^4：他妈的！我见过这种时间复杂度的算法

- 3^n：额滴神啊！这篇文章中会介绍一种算法，时间复杂度就是这个（而且很多数据库都使用了）
- factorial n：当数据量比较大时，你这辈子估计等不到计算结果
- n^n：如果你实现了一个算法，时间复杂度这样，你应该考虑自己是否适合搞 IT，建议去建筑工地找个活。。。

注意：我不会尝试精确地定义 O 这个术语，而是着重说明一个思想。你可以在这里 Wikipedia 找到精确的定义。

1.2 归并排序

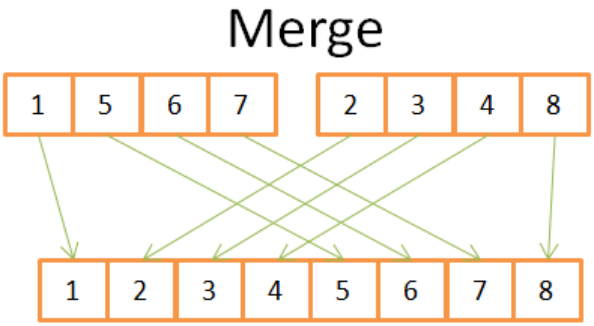
想要给一坨数据搞一下排序，你打算怎么做？什么？调用函数 sort()。。。好吧，回答不错。。。但是，想要理解数据库，首先必须理解函数 sort()的原理。

对于排序问题，有好几种算法；不过这里只介绍最重要的那个：归并排序。也许你现在并不理解排序算法有什么用，不过必须记住随后我们要讨论数据库查询的优化。而理解归并排序可以帮助我们更好地理解一个常见的数据库连接操作 Merge Join（**译者注：Merge Join 是数据库的物理连接方式，不是左连接，也不是右连接，更不是内连接或者全连接**）。

Merge

就像其他各种算法一样，归并排序也有它的技巧：把两个已经排序长度 N/2 的数组合并起来形成一个长度 N 的数组，运算次数是 N。这个过程，称作 **merge**

举个例子说明一下：



从上面的图可以看到，构造最终的完成排序的 8 元素数组，只需要 8 次运算；其中 2 个 4 元素的数组都只需要遍历一次。前提是，2 个 4 元素的数组都已经完成排序：

- 1）对比 2 个数组的当前元素（第一次运算时当前元素就是第 1 个）
- 2）然后，把最小（较小的）的放入 8 元素数组
- 3）接着，执行下一个元素（上一步最小的那个元素所在的数组），
- 接下去，就是重复 1，2，3 直到某个数组的最后一个元素
- 另一数组剩下的元素，全都放在 8 元素数组最后

这个算法之所以有效果，是因为 2 个 4 元素的数组都已经完成排序；因此你用不着给这个数组再来次遍历。

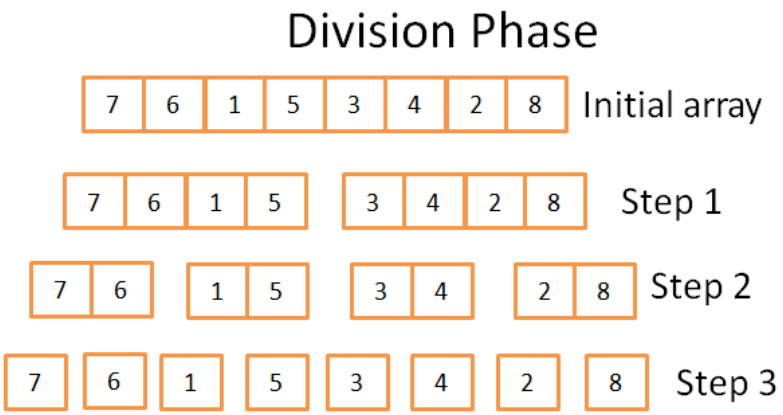
现在，我们理解了归并排序的技巧，下面是我的伪代码：

```
array mergeSort(array a)
   if(length(a)==1)
      return a[0];
   end if

   //recursive calls
   [left_array right_array] := split_into_2_equally_sized_arrays(a);
   array new_left_array := mergeSort(left_array);
   array new_right_array := mergeSort(right_array);

   //merging the 2 small ordered arrays into a big one
   array result := merge(new_left_array,new_right_array);
   return result;
```

归并排序的思想，把一个问题分解为小问题，解决了多个小问题就解决了整个问题。（注意：这种算法通称分治算法）。如果你没有理解这个算法，没关系；我第一次看到时，也没理解。下面的解释或许有助于理解，我把整个算法看成 2 个阶段：

- 第 1 阶段是分解，把整个大数组分解为多个小数组
- 第 2 阶段是排序，几个小数组被（按顺序）合并起来（使用 merge）构成大数组

第一阶段：分解

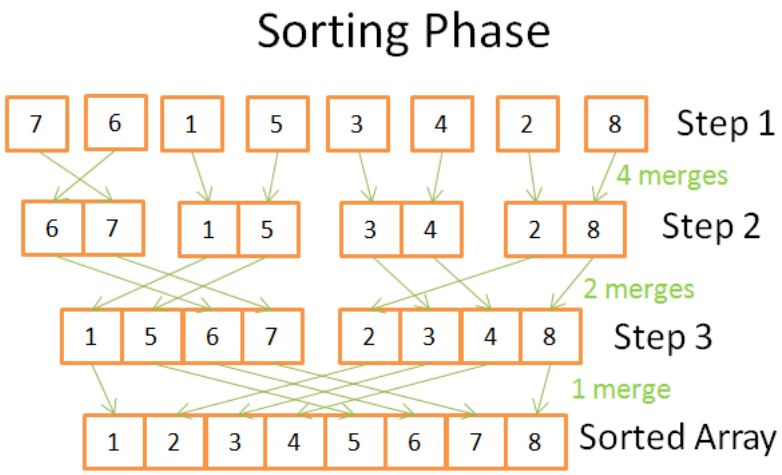


在第 1 阶段，8 元素的数组经过 3 次被分解程多个一元数组。分解的次数是 log(N) （因为 N=8，log(N) = 3)。

我怎么知道的？

~~我是天才嘛～~~ 或者说：数学。思想是，每一次分解，都是把某个数组分为 2 个。所以分解的次数，肯定是这样：数学上的对数（底数是 2)。

第 2 阶段：排序



排序阶段，从上一步分解出来的许多一元数组开始。每一步都使用 merge，最终的运算次次数是 N=8：

- 第一步，执行 4 次 merge，每个 merge 需要 2 次运算
- 第二步，执行 2 次 merge，每个 merge 需要 4 次运算
- 第三部，执行 1 次 merge，这个 merge 需要 8 次运算

因为需要 log(N)个步骤 ，所以整体的运算次数就是 **N*log(N)**。

归并排序的威力

为什么归并排序这个牛逼？

因为：

- 可以通过某些修改减少内存的占用，比如 merge 时不必创建新数组，而是直接修改

注意：这种算法，称作 in-place。

- 有一类算法可以通过某些修改，减少了磁盘占用和内存占用，同时不会带来磁盘 I/O 的压力。思想是，内存中只加载正在处理的部分数据，而不是全部。这类算法在某些场景可以大显身手：比如要处理一个好几 G 数据的表，但是内存却需要控制在 100M 左右。

注意：这类算法称作 external sorting.

- 可以通过某些修改，做到多进程/多线程/多机器执行上并发执行。

比如，分布式归并排序（distributed merge sort）是 Hadoop 关键组件(而 Hadoop 是 Dig Data 的框架)。

- 这个算法可以变废为宝点石成金（真相啊～)。

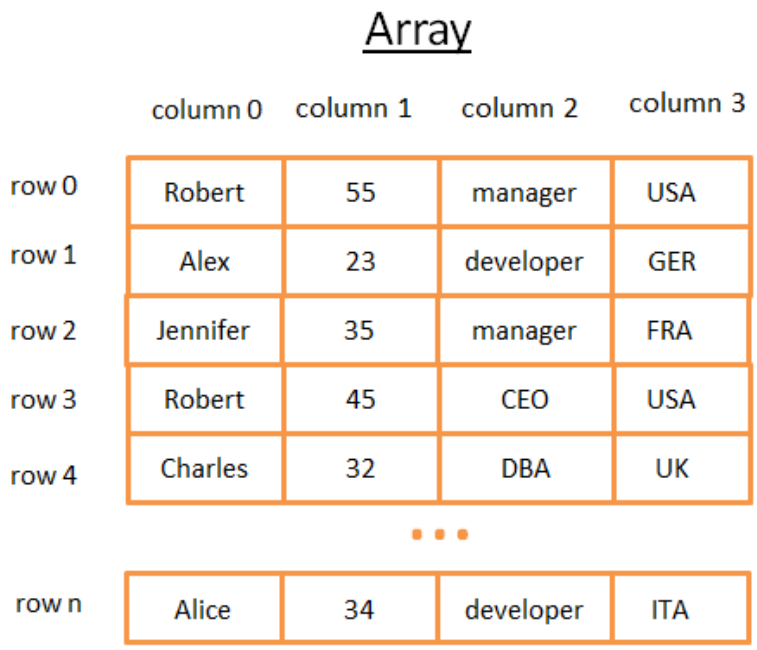
归并排序算法应用在绝大多数（虽然不是全部）数据库中，所以肯定还有其他的排序算法。如果想了解更多，可以参阅这个研究论文 research paper，其中对应用在数据库中的各种排序算法做了淋漓尽致的评头品足。

1.3 数组，树和哈希表

现在，我们已经了解了时间复杂度和排序背后的思想，我还要说 3 个数据结构。他们是现代数据库的骨干，非常重要。另外还会介绍一个术语：数据库索引。

数组

二维数组是最简单的数据结构。数据库的表，可以理解为一个二维数组，比如：



二维数组是一个有行有列的表：

- 每一行表示一个对象
- 列，都是描述这个对象的各个属性
- 每一列，保存某一个类型的数据(integer，string，date ...)

虽然二维数组能搞定数据的存储和抽象，而且看起来还不错；但是当你想要查询某个特定数据时，操蛋了。

比如，想找到所有在 UK 工作的小伙伴，你必须查询每一行，检查这一行工作地那一列是否匹配 UK。这个查询需要 N 次运算 (N 就是行数)，其实这个方法还不坏，但是有更好的方法吗？那必须是树咯！

注意：现代数据库提供了更高级的数组，用来存储表，比如堆表（heap-organized tables）和索引表（index-organized tables）。但是，这并没有解决多个（或一个）列有条件约束时的快速查找问题。

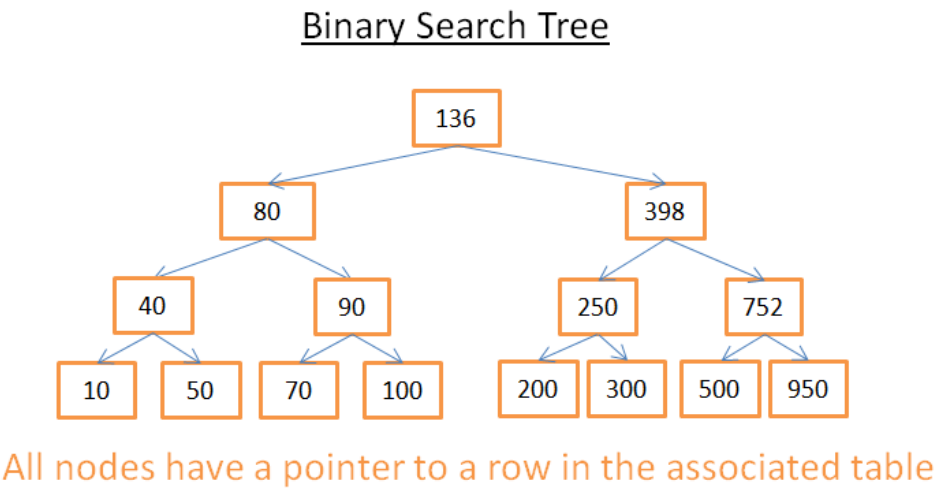
索引是数据库索引

二叉树，是一种特殊的树，关键在于每个节点都必须：

- 每个节点左边的所有节点都要比这个节点小
- 每个节点右边的所有节点都要比这个节点大

我们画个图，更直观，看看这是什么意思

思想



*我故意搞了一个错误，在节点 70，如果不是 70 而是 85，就正确了。

上图所示的 B 树（二叉树），有 N=15 个元素。我们试试查询节点 208：

- 从根节点 136 开始；由于 136<208，所以查询根节点的右子树。
- 398>208，所以，查询节点 398 的左子树
- 250>208，所以，查询节点 250 的左子树
- 200<208，所以，查询节点 200 的右子树。但是节点 200 没有右子树，所以 208 不存在！(因为如果 208 真的存在的话，它肯定在节点 200 的右子树里)

接下来，我们尝试查询 40

- 从根节点 136 开始；由于 136>40，所以查询根节点 136 的左子树
- 80>40，所以，查询节点 80 的左子树
- 40=40，所以，40 存在。这里我把表的行号当作树上 node 的 ID (上图没有画)，然后根据行号查询表
- 知道了行号，就可以知道数据在数据库表中的准确位置，然后，立刻马上就可以得到它。

最后，上面的两个例子，需要的运算次数取决于树的深度。如果前面归并排序部分你看得够仔细，应该还记得，树的深度就是 log(N)。所以，这个查询的时间复杂度是 log(N)，不错哦！

回到我们的问题

这里说的都太抽象了，我们需要结合具体问题。如果之前的表里，不再是傻傻的整形数值，而是字符串，字符串表示某人的“国籍”。假设，有一个树表示了这个表的“国籍”列：

- 如果你想知道哪些小伙伴在 UK 工作

- 你可以查询这个树，获取到节点 UK
- 在节点 UK 上，可以找到那些工作的 UK 的小伙伴的行号（一行或者多行）

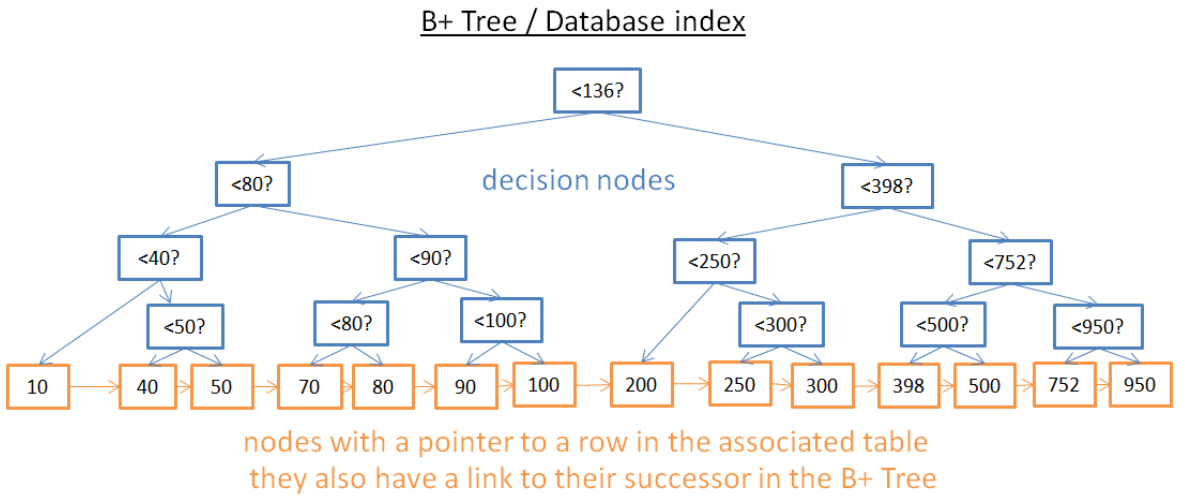
这个操作，只需要 log(N)次运算，而不是“遍历数组的”N 次！这个，就是数据库的索引。

你可以在任意类型（a string，an integer，2 strings，an integer and a string，a date…）的列上建立树结构索引，只要你有函数对比这种数据类型（比如多个列合起来的这种）；所以还可以给这些数据类型（可以是数据库中的任意基本类型）建立一个优先级.

B+树索引

虽然，树可以很好的解决查找某个特定值的问题，但是还有一个更大的问题：找到某个取值范围内的全部元素。如果遍历树上的每个节点检查它是否在 2 个值之间（比如中序遍历一个树），时间复杂度是 O(N)。而且，这个做法还会带来磁盘 I/O 负载，因为你要加载整个树。要更好的解决这个范围搜索问题，我们得想别的办法。答案在这里，现代数据库都使用了一个修改版的 B 树：B+树。B+树规则是：

- 只有最末端的节点（叶子节点）才存储数据，(比如，某个表的行号)
- 其他的节点（非叶子节点）只有路由功能------根据查询找到正确的节点



*同样地，我这里搞了一个错误，在节点 70

你可以看到，这个树比 B 树多了很多节点(几乎两倍甚至更多)。实际上，这里有许多附加的节点，“决策节点”就是用来实现路由功能，帮助找到正确的节点(真正存储数据的节点)。但是查询的时间复杂度依然是 O(log(N)) (深度最多增加 1)。B+树与 B 树最大的不同在于，每个叶子节点都指向它们的后续节点。

使用 B+树，如果我们想找到介于 40 和 100 的值：

- 首先只需要找到 40（如果 40 不存在就找最接近 40 的），就像在 B 树中的那样

- 然后收集节点 40 的后续节点（这里很方便，因为每个节点都执行自己的后续节点），直到 100。

假设，我们在一个 N 节点的树上找到了 M 个后续节点。首先查询到某一个节点，运算次数是 log(N)，就像 B 树一样。然后通过这个节点再找到（每个节点指向它的后续节点）M 个后续节点。这个运算的运算次数应该是 M + log(N)(而在 B 树上是 N)；而且还不需要加载整个树(只需要加载 M + log(N) 个节点)，这意味着更少的磁盘占用。如果 M 很小（与 200）而 N 很大（1000000）时，优势就很明显了！

不过，还有一个问题要解决(别怕～)。如果对数据库的某个表执行增删操作（那么这个表相关的 B+ 树也要相应增删)：

- 必须保证 B+树上各个节点排序正确，否则查询必定失败
- 必须保证整个 B+树的深度尽可能最小，否则时间复杂度会变坏，理想的 O(log(N))变成 O(N)

换句话说，B+树必须保持 **self-ordered** 和 **self-balanced**。幸好，如果使用智能删除和智能插入就可以解决这个问题，但是带了一些损耗：B+树上的插入和删除操作的时间复杂度也是 O(log(N))。这就是为什么有人说“索引太多不见得就好”。很明显，这样会拖慢很多表操作（插入，更新和删除），因为数据库同时需要去更新表上的索引，而每个索引的更新操作的时间复杂度是 O(log(N)) 。所以，增加索引意味着事务管理器更大的负荷（本文章后面会介绍事务管理器)。

想要了解更多，你可以参阅 Wikipedia article about B+Tree（关于 B+树的文章）。如果你想看到一个数据库里使用 B+树的真实案例，可以参阅这篇 this article 和这篇 this article（MySQL 的核心工程师写的）。这 2 篇文章都重点阐述了 InnoDB (MySQL 数据库的某个引擎) 处理不同索引的原理。

哈希表

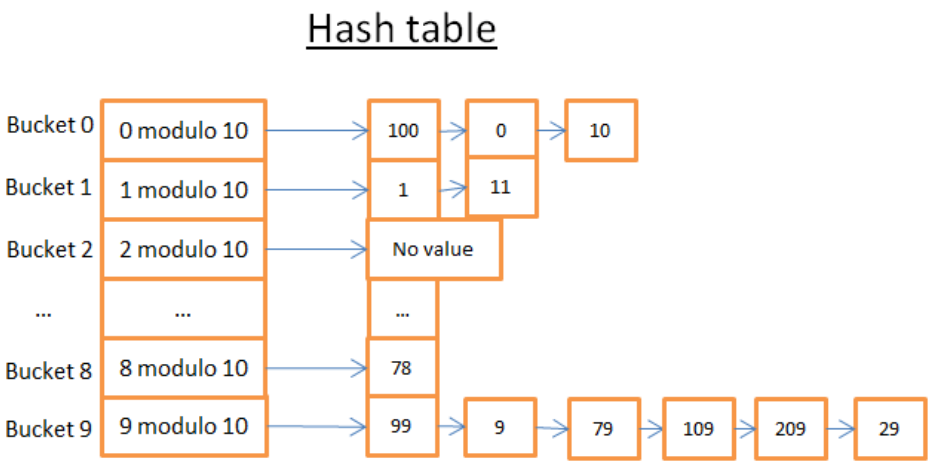
最后一个重要的数据结构，是哈希表。查询操作时，哈希表的实现速度更快。所以，理解哈希表，有助于我们更好的理解一个常见的数据库连接运算 Hash Join（**译者注：Hash Join 是数据库的物理连接方式，不是左连接，也不是右连接，更不是内连接或者全连接**）。这个数据结构还会用在数据库中存储一些内部数据（比如 **lock table 和 buffer pool，后面会介绍这两个概念**）。

The hash ta 哈希表，是一种可以通过元素的 key 快速查询到元素的数据结构。建立一个哈希表，需要，需要一些关键：

- 定义各个元素的 key（**译者注：通过一种方法从元素中提取出的关键码**）
- 定义一个哈希函数：可以通过 key 计算出元素所在位置 （位置可以称为 bucket）
- 定义一个“key 比较函数”：实现各个 key 的比较。如果找到了某元素所在的 bucket，还要通过这个函数在 bucket 内的众多元素中找到这个元素。

一个简单的例子

为了更直观，画个图：



上图所示的哈希表应该有 10 个 buckets。因为我懒得画，所以就只有 5 个了，其余 5 个以你的聪明才智应该想象得到。哈希函数很简单，元素的 key 模 10 运算。换句话说，太简单了，通过某元素的 key 的最后一个数字，就能找到这个元素所在的 bucket：

- 如果某元素的 key 通过哈希函数后得到 0，那么这个元素所在位置就是 bucket 0
- 如果某元素的 key 通过哈希函数后得到 1，那么这个元素所在位置就是 bucket 1
- 如果某元素的 key 通过哈希函数后得到 2，那么这个元素所在位置就是 bucket 2
- 。。。。。。

比较函数也很简单，对比 2 个数字是否相等。

我们一起演示一下，怎么找到 78：

- 78 的 key 通过哈希函数计算得到 8
- 看起来 78 应该在 bucket 8，然后先检查其中的第 1 个元素的 key，是否等于 78
- 通过“key 比较函数”得到结果：等于，那就找到了
- **这个查询运算的只需要 2 次运算**（1 次 key 通过哈希函数计算，1 次在 bucket 中找到某个元素）

现在，我们试试怎么找到 59：

- 59 的 key 通过哈希函数得到 9
- 看起来，59 应该在 bucket 9，其中第 1 个元素是 99；明显 99!=59，所以元素 99 不是我们想要的
- 使用相同的逻辑，通过“key 比较函数”检查第 2 个元素 9，然后第 3 个元素 79，...，最后一个元素 29

- 这个元素不存在哦
- 这个查询需要 7 次运算

适合的哈希函数

如上所述，因为查询的元素不同，时间复杂度也不同！

现在我们改变一下哈希函数，元素的 key 模 1 000 000 运算（或者说，去最后 6 为数字）。这样的话，第 2 次查询就只需要 1 次运算，因为 bucket 000059 里面没有元素。所以，最重要的是，找到一个适合的哈希函数------每个 buckets 里存储最少的元素。

对于上面的例子来说，找到一个适合的哈希函数比较容易，因为这个例子本身就很简单。但是在某些复杂情况下，找到适合的哈希函数很困难，比如元素的 key 是这样：

- 一个字符串（比如某个人的姓）
- 两个字符串（比如某个人的姓和名）
- 两个字符串加一个日期 （比如，某人的姓、名和生日）
- 。。。

如果哈希函数选择得足够好，那么查询的时间复杂度可以是 O(1)。

数组 vs 哈希表

有人问，为什么不使用数组呢？

嗯， 这个问题值得思考。

- 哈希表可以部分加载进内存，如果不需要其他 buckets，还可以不加载
- 使用数组的话，必须在内存上分配连续的内存资源。而如果数组很大的话，又要很大又要连续，这个太困难
- 使用哈希表的话，可以灵活地定义元素的 key（比如，某人的“国籍”加“姓”作为 key）

如果想了解更多，可以参阅这个关于 Java HashMap 的文档，其中介绍了一个高效的哈希表实现；而且其实不依赖 Java 语言本身，不懂 Java 也不影响理解其中的原理和概念。

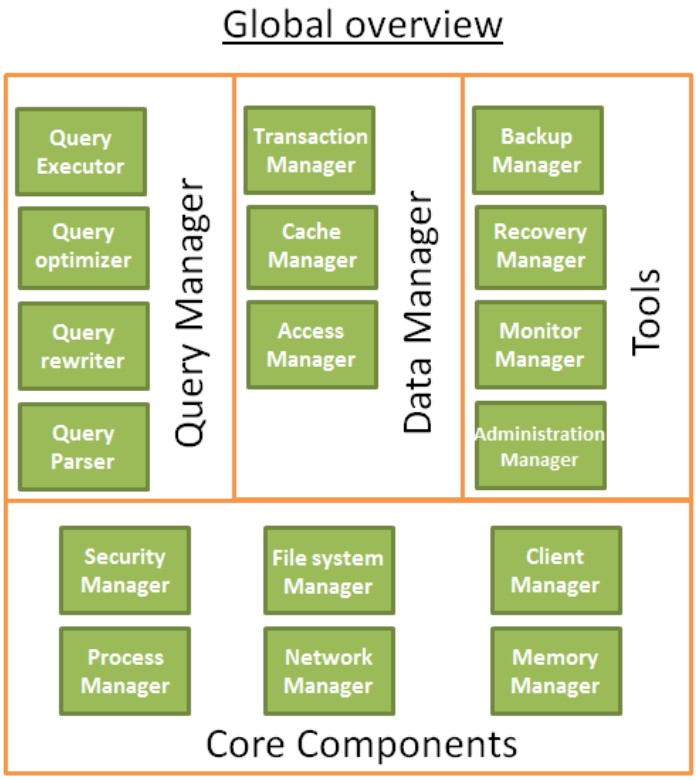
2 全局概要

我们刚刚概括地了解了数据库的基础组成部分，接着我们的认识还要继续扩展。

数据库实现对数据和信息的存储和整理，可以方便地存取和修改。其实，把一批文件整理在一起，也可以达到实现这个效果啊。事实上，最简单的数据库，比如 SQLite 就是这样：一堆文件。但是 SQLite 不是普通的一堆文件，而是饱含心思设计精巧的一批文件，它的特性有：

- 事务，保证数据存取时安全和一致
- 大量数据时的依然可以快速处理

通常来说，一个数据库可以认为包括以下组成部分：



在写这部分内容前，我阅读了许多论文和书籍，几乎每种资料都对数据库有不同的表达和实现。所以呢，我这里不太关心数据库的体系架构，也不关心进程的名称，因为我更倾向于讲清楚我想说的原理。为什么数据库由这些组件组成呢？大体来说，一个数据由多个组件组成，通过组件之间互动和协作实现数据库的完整功能。

核心组件：

- **进程管理器：**很多数据库需要许多进程/线程同时都运行，所以搞了一个进程/线程池来管理这些进程/线程。比如，有些数据库为了得到纳秒（nanoseconds）信息，不直接使用操作系统的线程，而是通过自己创建的线程来实现
- **网络管理器：**网络 I/O 是一个大问题，尤其是分布式数据库。所以，许多数据库有自己的网络管理器

- **文件系统管理器：**磁盘 I/O 是数据库的最常见的瓶颈所在。数据库有自己的文件系统管理器，可以更好的解决数据库与操作系统的文件系统之间数据交换，甚至替换操作习惯的文件系统。
- **内存管理器：**为了避免磁盘 I/O 的效率低下的问题，大量的内存的使用肯定很有价值。但是，如果操作大量内存时，必须一个独立的高效的内存管理器。特别是，同一时间有很多查询都在存取内存时，显得尤为必要
- **安全管理器：** 用户身份的鉴定和权限管理
- **客户端管理器：**管理多个客户端连接
- 。。。。。。

工具：

- **备份管理器：** 备份和恢复一个数据库
- **恢复管理器：** 数据库发生崩溃后重启，需要把数据库的所有数据做到一致状态
- **监控管理器：** 使用 log 记录数据库的所有行为，并且提供工具和信息用以监控数据库
- **管理管理器（译者注：翻译的不好，保留原文 Administration manager）：**恢复数据库的原数据（比如某个表的表名和结构），提供工具和信息用以管理数据库、Schemas（模式）和表空间。。。
- ...

查询管理器：

- **查询解析：**检查一个查询是否有效
- **查询重写：**预优化一个查询
- **查询优化：**优化一个查询
- **查询执行：**编译以及执行一个查询

数据管理器：

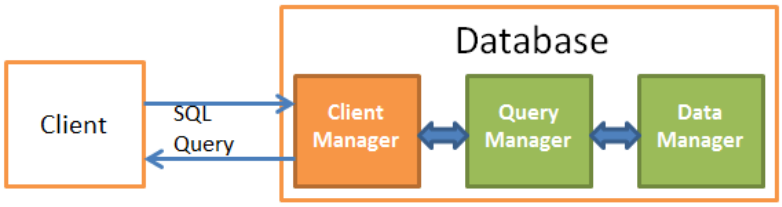
- **事务管理器：**处理事务
- **Cache 管理器：**在数据被使用前先加载到内存，保存将要写入数据库的数据
- **数据存取管理器：** 存取磁盘上的数据

本文剩下的内容，主要说明数据库如何通过以下三个步骤处理一个 SQL 查询：

- 客户端管理器
- 查询管理器
- 事务管理器

3 客户端管理器

Client manager



数据库的客户端管理器，负责处理来自客户的连接。客户，可以是一个服务器，或 Web 服务器，或者一个终端用户和终端软件。客户端管理器实现了各种不同的访问数据库的接口，比如众所周知的：JDBC，ODBC，OLE-DB。。。

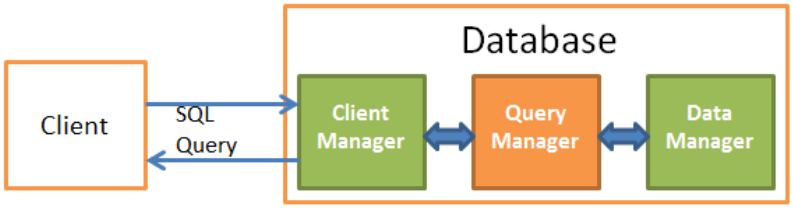
当然，也实现了一些专用的数据库访问接口。

如何处理一个数据库连接：

- 首先，客户端管理器检查用户的身份验证信息（用户名和密码），还检查用户是否有访问数据的权限（这类权限有数据库的 DBA 分配给用户）
- 然后，是否有可以处理这个连接的进程/线程资源可用
- 然后，此时此刻数据库的负载如何
- 如果需要等待一下才能继续处理。那么等待时间有超时机制，如果超时了应该关闭连接再给一个友好可读的错误信息
- 如果连接可以处理，查询会被交给查询管理器，查询管理器会继续处理
- 由于这个处理是一种孤注一掷的事情（要么有，要么没有，不可能半途而废），所以只要得到了数据，立刻就会进入缓存然后再分段返回数据给用户
- 在某些意外状况下，客户端管理器会断开连接，然后给用户一段可读的说明文字，并且释放数据库资源。

4 查询管理器

Query manager



这部分是数据库之所以神秘的原因所在。在这部分，哪些奇奇怪怪的查询代码，会被转换成可（快速）执行的代码。然后，这个代码会被数据库执行，并且客户端管理器会得到查询结果，这个过程可以分为以下几个步骤：

- 查询先被解析，检查是否合法
- 然后被重写，去掉一些冗余的运算，执行预优化
- 接着查询被优化，以期提高性能；再转换为可执行的数据存取计划
- 编译存取计划
- 最后，执行

这部分，我们不讨论最后 2 个步骤，因为前 3 个步骤才是重点。

前面这段文字，如果能轻松理解的话，建议再看看这些：

- The initial research paper (1979) on cost based optimization: Access Path Selection in a Relational Database Management System. This article is only 12 pages and understandable with an average level in computer science.
- A very good and in-depth presentation on how DB2 9.X optimizes queries here
- A very good presentation on how PostgreSQL optimizes queries here. It's the most accessible document since it's more a presentation on "let's see what query plans PostgreSQL gives in these situations" than a "let's see the algorithms used by PostgreSQL".
- The official SQLite documentation about optimization. It's "easy" to read because SQLite uses simple rules. Moreover, it's the only official documentation that really explains how it works.
- A good presentation on how SQL Server 2005 optimizes queries here
- A white paper about optimization in Oracle 12c here
- 2 theoretical courses on query optimization from the authors of the book "*DATABASE SYSTEM CONCEPTS*" here and there. A good read that focuses on disk I/O cost but a good level in CS is required.
- Another theoretical course that I find more accessible but that only focuses on join operators and disk I/O.

（译者注：上面的引用就不翻译了，谢谢）

4.1 查询解析

每一个 SQL 语句都会传给“查询解析”，“查询解析”要检查 SQL 的语法。如果 SQL 中有语法错误，那么拒绝。比如，如果以把 “SELECT …”写成“SLECT …”，基本就到此为止了。

比语法检查深入一点呢？还有，检查关键字的顺序。比如，关键字 WHERE 在关键字 SELECT 前面，也是不能通过检查！

然后，SQL 语句中涉及的表和列会被分析，“查询解析”会通过数据库的原数据去检查：

- 这个表是否存在
- 这个表的这个列是否存在
- 对列的不同类型的运算是否有效（比如，不能拿一个数字和一个字符串进行对比运算，不能对一个数字执行 substring()函数）

再然后，“查询解析”检查用户是否有权限读（或者写）这个表。说明一下，对表的操作权限，取决于数据库 DBA 的设定。

解析过程中，SQL 查询语句会被转换为一种内部表示方式（通常是一个数结构）。

如果通过了“查询解析”的检查，那么这个内部表示方式会被传递到“查询重写”。

4.2 查询重写

这一步，处理从上一步传来的内部表示方式开始。“查询重写”的目的在于：

- 对查询做预优化
- 去掉冗余的不需要的运算
- 提供信息协助“查询优化”找到最优的查询计划

“查询重写”会根据一系列普遍有效的规则对查询进行处理，如果查询 SQL 模式匹配到某条规则，那这条规则就生效了，然后查询 SQL 会被重写一下。下面是部分（非全部）规则，：

- **视图（View）插入：**如果查询 SQL 中有使用到某个视图，那么视图展开的 SQL 语句会插入到查询语句中
- **子查询平坦化：**查询 SQL 中有子查询时，优化比较困难。所以“查询重写”会尝试使用一个易于优化的子查询替换原来的子查询。

举例

```
SELECT PERSON.*
FROM PERSON
```

```
WHERE PERSON.person_key IN
(SELECT MAILS.person_key
FROM MAILS
WHERE MAILS.mail LIKE 'christophe%');
```

上面的查询 SQL 会被替换成：

```
SELECT PERSON.*
FROM PERSON, MAILS
WHERE PERSON.person_key = MAILS.person_key
and MAILS.mail LIKE 'christophe%';
```

- **删除冗余的运算：**比如，语句中使用了 DISTINCT，但是那一列上原来已经有 UNIQUE 索引约束了，所以关键字 DISTINCT 是冗余的会被删掉
- **多余连接消除：**如果重复使用了相同的连接条件，由于语句中有视图，而视图已经有这样的连接条件，或者由于其他传递性导致的无效连接，都会被消除掉
- **常量运算赋值：**如果你写了一个语句，其中需要一个运算，那这个运算也许会在“查询重写”中完成。比如，WHERE AGE > 10+2 会被转换成 WHERE AGE > 12，或者 TODATE(“某个日期”)会被转换成日期格式的数据
- **（高级） Partition Pruning:** 如果你正在操作一个分区表，“查询重写”这里会定位到正确的分区
- **（高级） Materialized view rewrite**：如果有一个物化视图，并且这个物化视图可以满足查询语句中的谓词，那么查询重写会检查这个物化视图是否更新到最新，然后再修改查询语句使用物化视图，避免使用本来的数据库表。
- **（高级） 定制规则：**如果有自定义的规则可以修改查询语句（就像 Oracle 的政策），查询重写会执行这些规则
- **（高级） Olap transformations**：分析函数和开窗函数、星型连接、RollUp 等，也都会被转换（不过我不确定这些转换由查询重写执行还是由查询优化执行，因为查询重写和查询优化这两个进程必须严格依赖数据库）

经过“查询重写”处理后，查询被发送给“查询优化”，然后好戏上场！

4.3 统计

在探讨数据库怎么优化一个查询前，我得先讲一下统计------因为没有统计，数据库就是 SB。如果你不指示数据库去分析它自己的数据，它不会自动去做，这样的话数据库会很傻很天真（不能对查询做出任何有效的优化）。

但是，数据库需要分析收集什么信息才能变聪明一点呢？

这里我必须大概说一下数据库和操作系统怎么存储数据。存储数据的最小单元，叫做 **a page 或 a block** （默认 4 或 8 KB）。这意味着，如果只有 1Kb 数据存储时，也需要占用一个完整的 page，如果 page 的大小是 8KB，那必然浪费 7KB 磁盘空间。

回到统计！当你指示数据库去收集一些统计信息，它会这么做：

- 一个表的行数和这个表占用的 page 数
- 对这个表的每一列：
  - 所有不同取值
  - 数据值的长度（最小程度，最大程度，平均长度）
  - 数据范围信息（最小值，最大值，平均值）
- 表上的全部索引

这些统计数据可以帮助数据库在优化某个查询时，判断磁盘 I/O、CPU 和内存使用。

每一列的统计信息非常重要。比如，某个表 Person 需要在 2 个列上作连接：LAST_NAME, FIRST_NAME。有了这些统计信息，数据库就知道列 FIRST_NAME 有 1 000 个不同的取值而 LAST_NAME 列有 1 000 000 个不同取值。因此，数据库会调整连接条件的顺序，是“LAST_NAME, FIRST_NAME”而不是“FIRST_NAME, LAST_NAME”，因为这样的调整减少了两个值的对比运算的次数，因为 LAST_NAME（有 1000000 个不同取值）大多数时间不大可能相同因此只需要对比前 2、3 个字符就够了。

除了这些基本的统计信息，你还可以让数据库去做一些高级统计，称作**柱状图**。通过柱状图，可以知道每一列的取值的分布情况。比如：

- 最高频的取值
- 不同取值的比例
- 。。。

这些高级统计获取到的信息，有助于数据库找到更优的查询计划。特别是，equality predicate (比如：WHERE AGE = 18) 或者 range predicates (比如：WHERE AGE > 10 and AGE <40)，因为这些 predicates 对行数敏感而数据库已经知道这些信息（注意：技术上有个专门术语叫做选择度）。

统计信息都存储在数据库的原数据中。比如，可以查看某个表（不是分区表）的统计信息：

- 在 USER/ALL/DBA_TABLES 和 USER/ALL/DBA_TAB_COLUMNS（Oracle 数据库）
- 在 *TABLES* 和 *SYSCAT.COLUMNS* （DB2 数据库）

这些统计信息，必须实时地更新到最新。没有什么事情比数据库认为某个表只有 500 行而实际上有 1M 行更坏的事情了！统计信息的唯一缺点在于，需要花时间费资源通过计算才能得到。这就是为什么大多数数据库上没有自动启用统计的原因。对于亿万级别的数据来说，计算统计信息真的有难度。在这个情况下，可以选择只计算基本的统计信息或者对数据库的抽样部分计算统计信息。

举例，有一个项目的数据库，数据量是千亿级别。我选择只统计其中 10%的数据，这样节省了很多时间。不过这样抽 10%的做法有一个风险，如果选择处理的那 10%（统计信息来自于某些表的某些列上）的数据，与整体的数据差别非常大（100M 行的表，不用担心发生这种事情）。这是一个失真（或者说错误）的统计信息，理想情况下 30s 能完成的查询可能会耗费 8 个小时；而且这个情况，要排查定位到根本原因，非常困难简直让人分分钟崩溃。这个例子应该可以让你知道统计有多么重要！

注意： 当然啦，每个数据库都有它自己的高级统计方式。如果想了解更多，可以预读这些数据库的文档。我曾经研究过统计信息的计算/使用的原理，感觉最好的官方文档是 one from PostgreSQL。

4.4 查询优化

所有现代数据库，都使用一种称为基于成本的优化（**Cost Based Optimization** 或 **CBO**）的方式去优化查询。这种方式的思路是，给每个运算设定一个成本，通过计算哪种运算顺序的总成本最小，得到最优的结果。

为了理解“成本优化”方式的原理，我觉得应该举个例子先感受一下这个方式的复杂度。下面这个部分，我会举例说明 3 种常用的表连接方式；然后我们会发现，即使最简单的连接查询，也是查询优化的噩梦！最后，我们还是要探讨数据库里解决这个问题的真实情况。

对于这些连接，我们把重点放在它们的时间复杂度上，而数据库怎么计算 CPU 消耗、磁盘 I/O 和内存占用先不关注。时间复杂度与 CPU 消耗之间的差别在于，时间复杂度可以近似计算，而 CPU 消耗就需要考虑每个运算（比如一个条件、一个 if 语句、一个并发或者一个循环），而且：

- 高级语言的每个运算，都对应很多个 CPU 运算
- 运算的消耗不是相同的（ 术语叫 CPU 周期），因为 Intel Core i7、Intel Pentium 4、AMD Opteron 是不同的。换句话说，取决于 CPU 架构。

使用时间复杂度更加容易（至少对我来说），而且我们依然可以研究 CBO 理论。有时候我也会提到磁盘 I/O 因为这是一个很重要的概念。要知道，大多数时候，瓶颈在磁盘 I/O 而不是 CPU 消耗。

索引

当我们说 B+树是，提到过索引。要记得，索引是排好序的。

仅供参考，其实还有其他多种索引，比如 **bitmap indexes**。不同的索引，在 CPU 消耗、磁盘 I/O 和内存占用上的表现与 B+树索引有所不同。

然而，很多现代数据库可以动态地创建临时索引，主要目的在于临时地实现某个查询的优化。

Access Path

在使用连接运算前，需要先（从表中）获取数据。下面介绍几种获取数据的方式。

注意：因为实际情况中，所有 access paths 的问题都在于磁盘 I/O，所以不必考虑时间复杂度了。

Full scan（全扫描）

如果你曾经读过某个查询计划，你一定看到过 **full scan**（或者 scan）。Full scan 很简单，就是数据库完整地读取一个表或者索引。**从磁盘 I/O 角度来看，对一个表的 full scan 的成本很明高于对一个索引的 full scan**。

Range Scan（范围扫描）

还有其他多种类型的扫描，比如 **index range scan**。如果你的 SQL 语句中使用了谓词"WHERE AGE > 20 AND AGE <40"，就用得到了。

当然，你的列 AGE 上得先有一个索引，这样才能 index range scan。

在第一节，我们已经知道了 range query 的时间复杂度大约是 log(N) +M，其中 N 是索引中的数据量，M 是范围内的行数。由于统计信息的缘故，N 和 M 都可以认为是已知的（注意：M 是对于谓词 AGE >20 AND AGE<40 的选择度）。然而，对于一个范围扫描，其实不必全部扫描整个索引，所以对比 full scan 有优势了：更低的磁盘 I/O 成本。

Unique Scan(唯一扫描)

如果只需要从索引中取出一个值，可以使用 **unique scan**。

通过行号存取

多数时候，如果数据库使用到了索引，那它必须找到与索引对应的行。怎么做到呢，使用行号。

举例子，做这样的事情：

```
SELECT LASTNAME, FIRSTNAME from PERSON WHERE AGE = 28
```

如果表 Person 的列 AGE 上有索引，优化时可以使用索引拿到全部年龄是 28 的人，再通过行号从表 Person 得到完整信息。为什么读了索引，还要再读表呢？因为索引上只有 AGE，其他信息（比如姓名）不在索引上，所以依然要读表。

但是，如果要做这样的事：

```
SELECT TYPE_PERSON.CATEGORY from PERSON ,TYPE_PERSON
WHERE PERSON.AGE = TYPE_PERSON.AGE
```

表 PERSON 上的索引，可以用来与 TYPE_PERSON 连接，但是表 PERSON 不必再读，因为这个查询没有要求表 PERSON 上的其他信息。

虽然，对于存取数据量不大时，这样做还算 OK。但是这个做法的真正弱点在于磁盘 I/O，如果运算时巨量很大时，效果相当于 full scan 了。

其他方式 Others paths

我这里不会列出所有的 access paths。如果有兴趣了解更多，可以参阅 Oracle documentation。各个数据库中 access path 的名字不能不太一样，但是原理是相同的。

（译者注）

Access Path	Heap-Organized Tables	B-Tree Indexes and IOTs	Bitmap Indexes	Table Clusters
Full Table Scans	x			
Table Access by Rowid	x			
Sample Table Scans	x			
Index Unique Scans		x		
Index Range Scans		x		
Index Full Scans		x		
Index Fast Full Scans		x		
Index Skip Scans		x		
Index Join Scans		x		
Bitmap Index Single Value			x	
Bitmap Index Range Scans			x	
Bitmap Merge			x	
Bitmap Index Range Scans			x	
Cluster Scans				x
Hash Scans				x

连接运算符

所以，我们现在知道了怎么获取数据，现在我们试试连接运算！

这里解释 3 个常见的连接方式：Merge Join，Hash Join 和 Nested Loop Join。首先，我要介绍 2 个新术语：**inner relation** 和 **outer relation**。一个关系可以是（关系型数据库的理论）：

- 一个表
- 一个索引
- 前面运算产生的中间结果（比如一个连接的结果）

当我们连接两个关系时，连接算法处理两个关系的方式并不相同！本章中，我们做如下约定：

- 连接运算符左边的关系称为 outer relation
- 连接运算符右边的关系称为 inner relation

比如，A JOIN B 是 A 和 B 的连接，A 作为 outer relation 而 B 作为 inner relation。

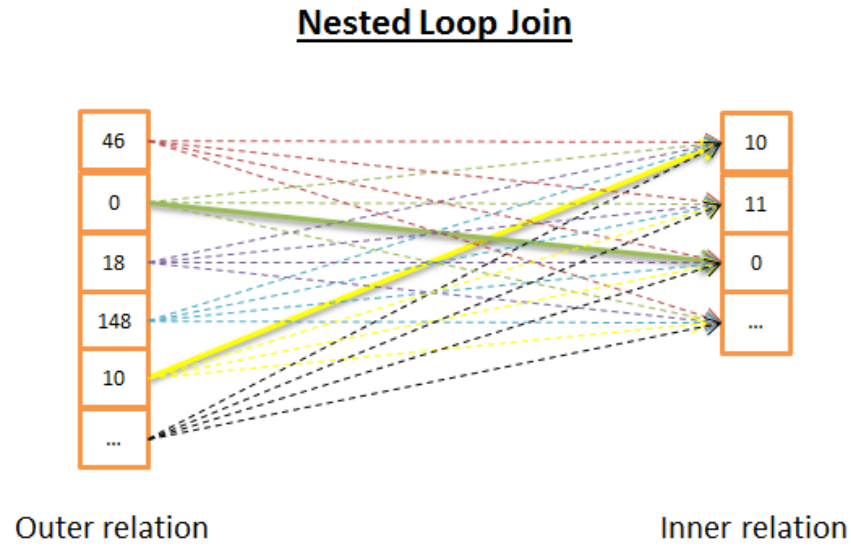
多数时候，**A JOIN B 与 B JOIN A 的成本并不相同。**

在这部分，我要假设，outer relation 有 N 个元素，inner relation 有 M 个元素。要记住，实际状况下，数据库可以功过统计信息得知 N 和 M。

注意： N 和 M 分别是两个关系的基数。

Nested loop join

Nested loop join 是最简单的连接方式



这里是基本思想：

- 对 outer relation 的每一行
- 检查 inner relation 中的每一行是否满足连接条件**（译者注：原文没有提到连接条件，译者根据自己的理解）**

下面是伪代码：

```
nested_loop_join(array outer, array inner)
  for each row a in outer
    for each row b in inner
      if (match_join_condition(a,b)
```

```
        write_result_in_output(a,b)
      end if
    end for
  end for
```

因为这里是一个双循环，所以时间复杂度是 **O(N*M)**

从磁盘 I/O 来看，外循环 N 次从 outer relation 中读取 N 行，内循环 M 次从 inner relation 读取 M 行数据，这个算法需要读 M + N*M 次。但是，inner relation 足够小的话，你会发现如果能把 inner relation 读出来缓存起来，这样运算读取次数就是 M +N。要想实现这个修改，需要 **inner relation** 真的够小，这样才能把它放在内存里。

从 CPU 消耗来看，没有差别；但是从磁盘 I/O 角度考虑，如果两个关系可以只读一次都放在内存里，那就更好啦。

当然，inner relation 可以是一个索引，那样的话，对磁盘 I/O 来说就好上加好了。（译者注：因为索引是表的一列或者几列，所以读取的数据当然更少）

因为这个算法真的非常简单，这里可以有一个改进版，对磁盘 I/O 更好；如果 inner relation 因为太大不能放进内存的话。这是基本思想：

- 不要逐行地读取两个关系
- 分组地从两个关系读取数据，然后都放进内存里（2 个关系各有一组数据在内存里）
- 对比两个组的行，保留满足连接条件的行**（译者注：原文没有提到连接条件，译者根据自己的理解）**
- 读完一组，继续读下一组放进内存，继续同样的操作
- 直到 2 个关系的数据都处理完

这里是大概的算法伪代码：

```
// improved version to reduce the disk I/O.
nested_loop_join_v2(file outer, file inner)
  for each bunch ba in outer
  // ba is now in memory
    for each bunch bb in inner
              // bb is now in memory
                    for each row a in ba
          for each row b in bb
            if (match_join_condition(a,b)
```

```
            write_result_in_output(a,b)
          end if
        end for
      end for
    end for
  end for
```

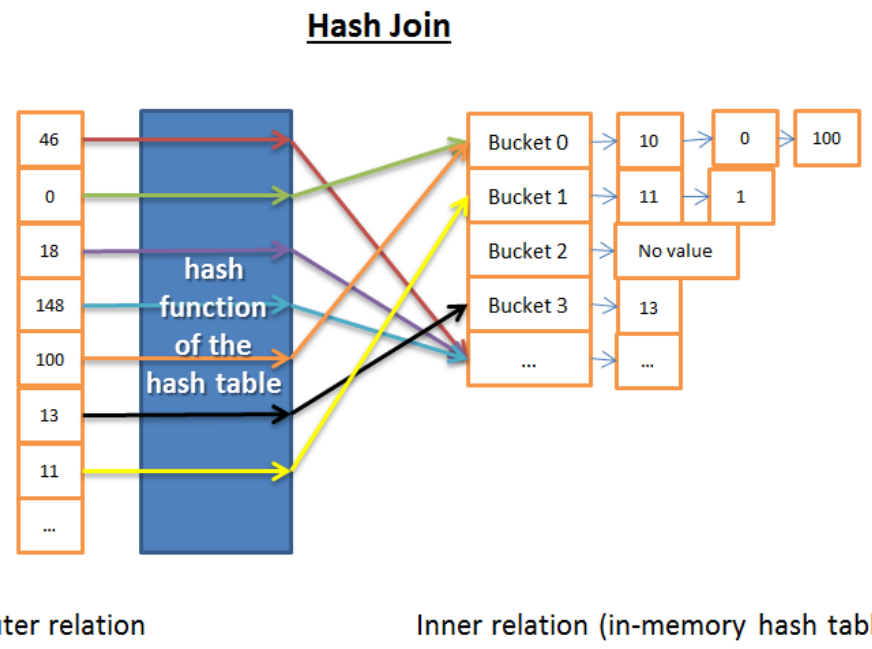
在这个版本，时间复杂度还是相同，但是磁盘 I/O 明细地减少了很多：

- 前一个版本，算法需要 N + N*M 次读取数据（每次一行）
- 新版本，读取数据的次数变为 number_of_bunches_for(outer)+ number_of_ bunches_for(outer) * number_of_ bunches_for(inner)。
- 如果增加分组的大小，每次读出更多，还能继续减少读取数据的次数。

注意：虽然每次读取的数据多一些可以减少读取的次数，然无卵用啊！因为数据从硬盘中读出，属于顺序操作（根本问题在于，机械硬盘的弱点------需要时间去寻道）。

Hash join

哈希连接是一个比较复杂的方式，但是成本比 nested loop join 更低（大多数情况都能更低）



哈希连接的思想是：

- 1） 读取 inner relation 的全部元素
- 2） 构建一个常驻内存的哈希表
- 3） 逐个从 outer relation 读取元素
- 4） 计算每个元素的哈希（哈希函数就是构建哈希表时的哈希函数），发现 inner relation 的相关 bucket
- 5） 在 bucket 中找到是否有元素与 outer relation 的元素相同

分析一下这个时间复杂度，我要先做些假设简化这个问题：

- inner relation 分为 X 个 buckets
- 哈希函数处理 2 个关系时可以保持一致，换句话说，各个 buckets 基本同样大小
- outer relation 的元素与某个 bucket 里的所有元素的对比的次数，基本取决于 bucket 中元素的数量

时间复杂度是 (M/X)*(N/X) + cost_to_create_hash_table(M) + cost_of_hash_function*N

如果哈希函数创建了一个够小的 buckets，那么时间复杂度就是 **O(M+N)。（译者注：bucket 更小意味着 X 更大，如果每个 bucket 里面只有 1 个元素，X==M，如果 M==N，就意味着时间复杂度成为 cost_to_create_hash_table(M) + cost_of_hash_function*N，就是 O(M+N)咯！）**

哈希连接也有另一个版本，占用内存更少而且磁盘 I/O 也更少，那就是：

- 1）给 inner relations 和 outer relations 都构建哈希表
- 2）然后（2 个哈希表都）放回到磁盘中
- 3）逐个 bucket 对比（其中一个 bucket 常驻内存，而另一个逐行读出）

Merge join

这个 merge join 连接方式是唯一可以输出排好序的结果的连接方式。

注意：一个简化的 merge join，既没有 inner relations 也没有 outer relations；两个关系的角色相同。但是真实实现中，两个关系有区别，比如处理 duplicates 时。

merge join 连接可以分为 2 个步骤：

1. （可选）连接排序运算：2 个关系都根据连接条件完成排序
2. Merge join 运算：2 个排好序的关系互相合并

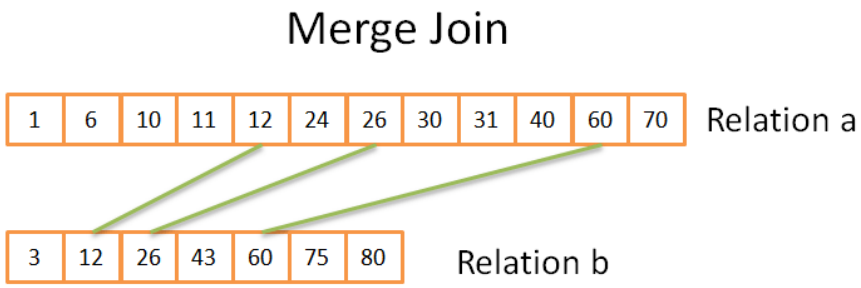
排序

我们之前说过归并排序，在这个例子中，归并排序算是一个很棒的算法（不过也不是最好的，如果内存够大或者不用考虑内存占用，有更好的算法哦）。

不过有时候连接的 2 个关系已经排好序了，比如：

- 一个表可以原本已经排好序，比如连接的某个表符合某种索引约束
- 或者连接的就是一个索引
- 或者连接的是前面运算的中间结果，而前面的运算已经给这个中间结果排好序。

Merge join



这部分的原理与我们之前看到的归并排序的原理很相似。但是这一次，我们不会从两个关系中逐个读取每一个元素，而是只读取那些满足连接条件（译者注：原文没有提到连接条件，译者根据自己的理解）的元素。思路是这样：

- 1） 比较 2 个关系的当前元素（第一次的当前元素就是第 1 个元素）
- 2） 如果相等，把 2 个关系的元素都放入结果中，然后处理下一个元素
- 3） 如果不相等，比较小的那个元素所在的关系进入下一个元素（因为下一个元素也许会相等）
- 4） 重复步骤 1、2、3，知道 2 个关系都处理到最后一个元素

因为处理的这 2 个关系，都是排好序的，所以从小到大、一路向前、不用“回头”、一次搞定！

上面介绍算是一个简化的版本，因为其中没有考虑 2 个关系中存在多个相同的元素（换句话说，多相匹配）。这个算法的真实情况会比较复杂，不适合用来说明原理，这也是我在这里简化的原因。

如果 2 个关系的已经排好序，那么时间复杂度就是 **O(N+M)**

如果 2 个关系需要我们先排序再连接，那么时间复杂度主要（这里 N+M 可以忽略）是 **O(N*Log(N) + M*Log(M))**

那些牛逼的计算机 geek，一定想实现这个算法（注意：这里是我的伪代码，不保证 100%正确哦）：

```
mergeJoin(relation a, relation b)
```

```
relation output
integer a_key:=0;
integer  b_key:=0;

while (a[a_key]!=null and b[b_key]!=null)
   if a[a_key] < b[b_key] a_key++; else if a[a_key] > b[b_key]
    b_key++;
   else //Join predicate satisfied
    write_result_in_output(a[a_key],b[b_key])
    //We need to be careful when we increase the pointers
    if (a[a_key+1] != b[b_key])
      b_key++;
    end if
    if (b[b_key+1] != a[a_key])
      a_key++;
    end if
    if (b[b_key+1] == a[a_key] && b[b_key] == a[a_key+1])
      b_key++;
      a_key++;
    end if
  end if
end while
```

上面 3 种连接方式哪种最好？

如果有最好的连接，那就不会有多种连接了。哪个最好，这个问题很难回答，因为需要考虑的因素太多：

- **内存的占用：**如果没有足够的内存，基本要告别强大的 hash join（至少也告别全内存 hash join）
- **2 个关系的数据量：**比如要连接的两个表，一个数据量特别巨大，一个又很小很小，这时候 nested loop join 的效果要比 hash join 好，因为 hash join 给那个数据量巨大的表创建 hash 表就很费事。如果两个表都有巨量的数据，nested loop join 连接方式的 CPU 负载会比较大。
- **索引的方式：**如果连接的两个关系都有 B+树索引，那肯定是 merge join 效果最好
- **结果是否需要排序：**如果希望这次连接得到一个排序的结果（这样就可以使用 merge join 方式实现下一个连接），或者查询本身（有 ORDER BY/GROUP BY/DISTINCT 运算符）要求的排序的结果；如果是这个情况，即使当前要连接的 2 个关系本身并没有排好序，依然建议选择稍微有点费事的 merge join（可以给出排序的结果）。
- **连接的 2 个关系本身已经排序：**这个情况，必须用 merge join 啊

- **连接类型**：如果是等值连接（比如：tableA.col1 = tableB.col2）？或者是内连接、外连接、笛卡尔积、自连接？有些连接方式（**译者注：上面的 3 中物理连接方式**）可能不能处理这些不同类型的连接
- **数据的分布**：如果连接条件的数据扭曲了（比如要连接表 PERSON 连接条件是列“姓”，但是意味的是很多人的姓是相同的），这个情况如果使用 hash join 一定会带来灾难，对吧？因为哈希函数计算后各个 buckets 上数据的分布肯定存在巨大的问题（**译者注：有些 bucket 很小，只有一两个元素；而有些 buckets 太大，好几千的元素**）。
- **多线程/多进程**：如果希望连接运算可以分布式执行

如果想了解更多，请参阅 DB2，ORACLE 和 SQL Server 文档。

简单的例子

我们刚刚探讨了 3 中连接方式

现在，我们尝试一下连接 5 个表获取某个人的全部信息，包括这些：

- 个人手机号码
- 个人邮箱
- 个人居住地址
- 个人银行帐号

再解释下一下，更快地处理下面这个查询：

```
SELECT * from PERSON, MOBILES, MAILS,ADRESSES, BANK_ACCOUNTS
WHERE
PERSON.PERSON_ID = MOBILES.PERSON_ID
AND PERSON.PERSON_ID = MAILS.PERSON_ID
AND PERSON.PERSON_ID = ADRESSES.PERSON_ID
AND PERSON.PERSON_ID = BANK_ACCOUNTS.PERSON_ID
```

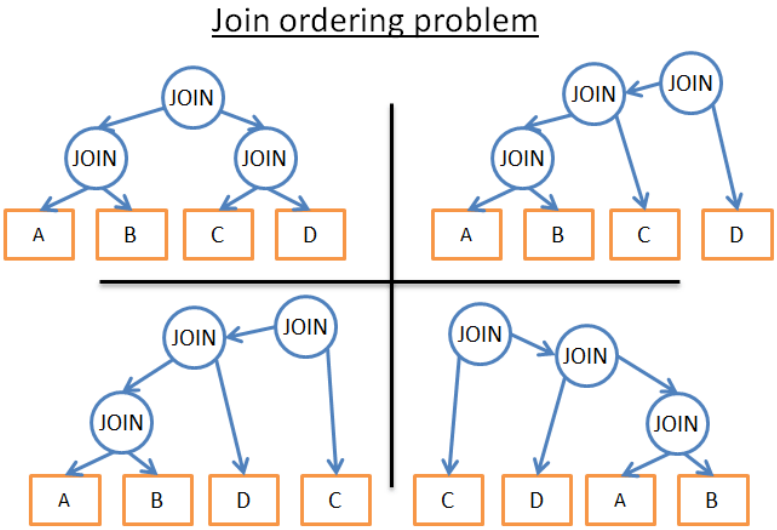
作为查询优化，必须尝试找到最优的查询计划。但是还有 2 个问题：

- 每次连接，我应该使用哪种连接方式（**译者注：这里的连接就上前面说过的 3 中物理连接**）？

有 3 种可选的方式（Hash Join，Merge Join，Nested Join），还有“没有索引”、“一个表有索引”和“2 个表有索引”等不同情况（另外索引还分不同类型，这里就不考虑了）。

- 我应该使用怎样的连接顺序？

比如，下面这个图展示了 4 个表 3 次连接的可能顺序：



所以，也许我可以这么做：

- 1） 使用暴力方式

根据数据库的统计信息，计算每一种查询计划的成本，得到最好的那个。但是，所有的查询计划的数量是很大的，对于每个连接顺序，其中的每个连接运算都有 3 种方式：HashJoin，MergeJoin，NestedJoin。所以每个连接顺序都有这么多查询计划，大约是 3^4。连接顺序是一个“二叉树遍历问题”permutation problem on a binary tree，可能性有这么多(2*4)!/(4+1)!。这样算下来，所有的查询计划有这么多，大约 3^4*(2*4)!/(4+1)!，而且这还是一个简单的例子。

通俗的讲，所有的查询计划有 27216 个。如果我们再考虑 merge join 时 B+树索引的差异（有几个表有 B+树），那么查询计划的数量会达到 210 000！还记得我刚刚说过这是一个很简单的例子吗？

- 2） 我崩溃了，老子不干了

看起来有模有样，实际上不可能得到结果嘛，我还要赚钱养家呢

- 3） 我只计算其中一部分查询计划，找到其中成本最小的

毕竟我不是超人，我不可能计算所有的查询计划。实际上，可以从全部可能性中任意地筛选出一部分，再从中找到一个成本最小的

- 4） 我使用一些智能的规则，从全部查询计划中选择一部分

这里是 2 个规则：

可以使用某些逻辑作为规则，可以帮助去掉那些成本明细很大的查询计划，但是这样的筛选粒度太粗。比如，这样的规则“使用 nested loop join 连接时 inner relation 必须是数据量更小的那个关系”。

为了找到最优的查询计划，我们换一个思维方式，通过更多的规则去消减更多（明显不好的）查询计划，剩下的应该都不错吧！比如，“如果某个关系数据库很小，使用 nested loop join 这种连接，别使用 merge join 也别使用 hash join”。

在这个“简单”的例子中，我说过所有的查询计划有很多，但是在真实的查询中，会更多！可以考虑其他的关系运算符，比如 OUTER JOIN，CROSS JOIN，GROUP BY，ORDER BY，PROJECTION，UNION，INTERSECT，DISTINCT。。。如果这些也考虑进去，你可以想象一下所有的查询计划会有多少，我简直不敢想。

然而，数据库到底怎么解决这个问题的呢？

动态规划，贪婪算法和启发式算法

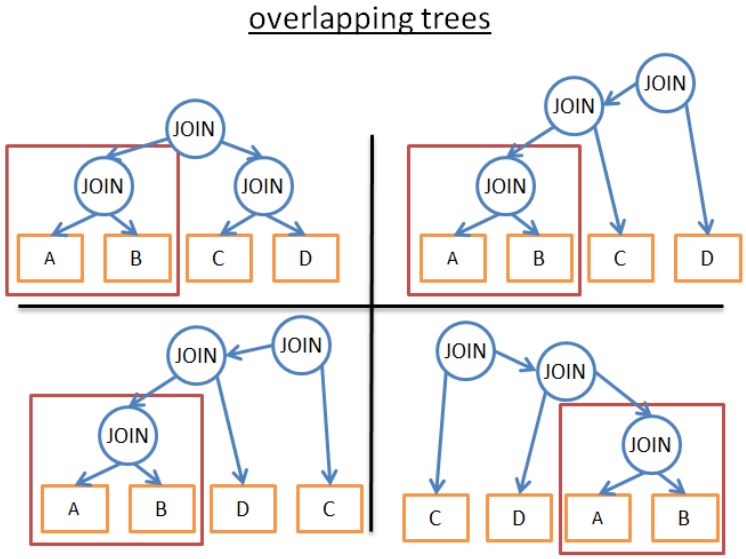
一个关系型数据库，需要尝试多种方式去实现优化。真正需要实现的，不是仅仅“找到最优的查询计划”，而是“在有限的时间内找到最优的查询计划”。

多数时候，数据库优化不能找到“最好”的查询计划，而是找到一个“还不错”的。

对于小型的查询，粗暴遍历所有的查询计划，是可行的。如果有一个方法可以避免多余的运算，这样对于中型的查询，其实也可以搞粗暴遍历。这个方法叫做“动态规划”。

动态规划

实施动态规划的背景是，有许多查询计划是相似的。比如，观察下面的查询计划：



它们有共同的子树（A JOIN B）。所以，这个子树的成本，不必每次都重新计算，而是只计算一次然后保存计算结果，下次看到是直接使用。正式地说，我们遇到了重叠问题。为了避免重叠部分的过度计算，我们需要使用“记忆”技术（memoization）。

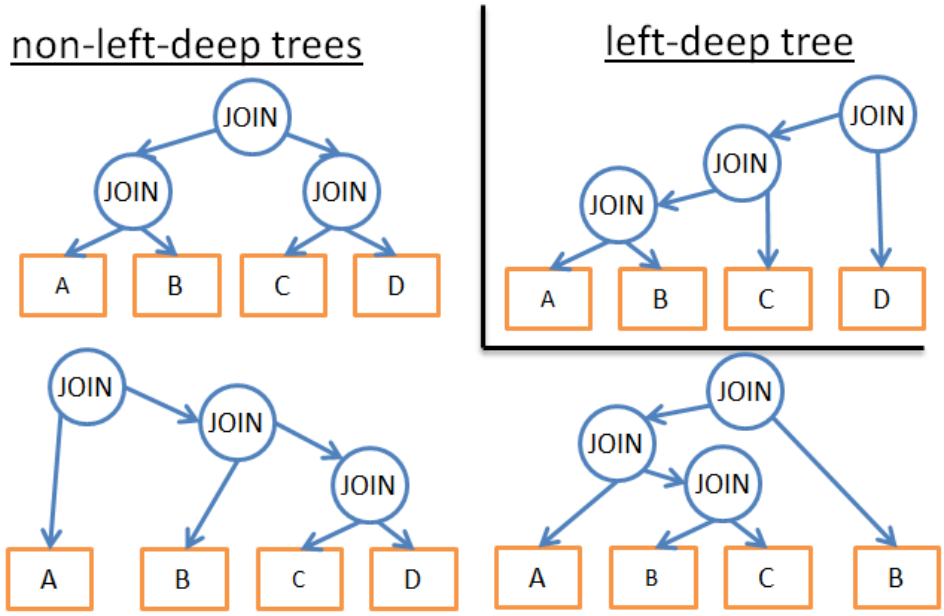
使用记忆技术，所有的查询计划（或者说时间复杂度）不再是(2*N)!/(N+1)!而是 3^N。拿我们之前的 4 个表连接的简单例子来说，所有的查询计划从 336 个减少到 81 个。如果现在要连接 8 个表（也不算太复杂），意味着所有的查询计划从 **57 657 600 减少到 6561**。

计算机 geeks 会对这个感兴趣，下面是我在 formal course I already gave you 发现的算法。我不解释这个算法，如果你懂得动态规划或者擅长算法，就自己看吧（我警告过你哦）：

```
procedure findbestplan(S)
if (bestplan[S].cost infinite)
   return bestplan[S]
// else bestplan[S] has not been computed earlier, compute it now
if (S contains only 1 relation)
         set bestplan[S].plan and bestplan[S].cost based on the best way
         of accessing S  /* Using selections on S and indices on S */
     else for each non-empty subset S1 of S such that S1 != S
   P1= findbestplan(S1)
   P2= findbestplan(S - S1)
   A = best algorithm for joining results of P1 and P2
   cost = P1.cost + P2.cost + cost of A
   if cost < bestplan[S].cost
       bestplan[S].cost = cost
      bestplan[S].plan =  “execute P1.plan; execute P2.plan;
                  join results of P1 and P2 using A”
return bestplan[S]
```

对于大型的查询，其实依然可以使用动态规划，不过需要一些附加规则（启发式）去掉多余的（那些明显不好的）查询计划：

- 如果我们分析某一种类型的查询计划（比如 left-deep trees，左叉树）**（译者注：这里翻译成左叉树，不知道是否准确）**，那么所有的查询计划再是 3^n 而是 $n*2^n$。



- 如果我们再加一个逻辑规则去匹配那些不合适的查询计划然后去掉（比如，“如果某个表的某列有索引，而且查询只需要索引列，这样不必尝试 merge join 这个表，而是索引”）这样可以显著减少可能查询计划的数量，而且不必担心（因为不大可能）错失“最好”的查询计划。
- 如果在处理过程逻辑上增加规则（比如，“连接运算符的优先级高于其他运算符”），这样又可以减少一大批（明显不好）的查询计划。
- 。。。

贪婪算法

对于大型查询或者需要更快响应（只是查询计划判断时间有限，并非查询整个运算时间有限）的情况，还有一种算法有用武之地，贪婪算法。

这个算法的思路是，迭代地使用某一个规则（或者 **heuristic，启发**）逼近最优的结果。有了某个规则，贪婪算法可以一步一步地得到最优结果。这个算法从连接运算符开始处理查询，然后每一步都使用相同的规则增加一个连接运算符。

举一个简单的例子，如果我们要在 5 个表（A，B，C，D 和 E）上做 4 次连接运算。为了简化这个问题，我们只考虑 nested join 不考虑其他（2 种）连接方式。我们尝试这个规则：选择成本最低的连接方式：

- 我们任意选择一个表作为第 1 个连接的表（那就表 A 吧）
- 计算其他的表与 A 连接的成本（表 A 可以是 inner relation 也可以是 outer relation）
- 我们发现 A JOIN B 这样连接的成本最低

- 然后我们再计算其他表与 A JOIN B 连接的成本（A JOIN B 可以是 inner relation 也可以是 outer relation）
- 我们发现 (A JOIN B) JOIN C 这样连接成本最低
- 然后，我们继续计算其他表与 (A JOIN B) JOIN C 连接的成本。。。
- 。。。
- 最后，我们得到“最优”的查询计划：(((A JOIN B) JOIN C) JOIN D) JOIN E)

因为我们任意选择了 A 作为第 1 个表，所以我们还可以继续这个算法，拿 B 作为第 1 个，再拿 C，再拿 D，再拿 E；最后我们肯定能得到真正最优的查询计划。

顺便说一下，这个算法的名字，叫做为 Nearest neighbor algorithm。

这里就不深究细节了， 总之只要有一个好的模型和时间复杂度 N*log(N) 的排序算法，这个问题不难解决。贪婪算法，如果完整地结合动态规划，时间复杂度（**译者注：所有的查询计划也可以当作这个算法的时间复杂度**）大约是 **O(N*log(N))而不是 O(3^N)。**如果有一个查询很复杂，需要连接 20 次，意味着时间复杂度是 26 而不是 3 486 784 401 那么大！

这个算法有一点不足，我们假设了两个表的最低成本连接方式在增加一个表连接时，依然保持成本最低。但是：

- 即使 A JOIN B 在三个表 A，B 和 C 连接时成本最低
- 然而有可能(A JOIN C) JOIN B 的成本比(A JOIN B) JOIN C 更低。

想要弥补这个不足，可以使用多重贪婪算法，分别使用不同的逻辑规则处理，这样就可以得到最优的查询计划。

其他算法

【如果你已经受够了这些枯燥的算法，可以调到下一节，没关系的。接下来要说的，其实已经不重要】

找到最优查询计划的问题，对于很多计算机领域的研究者来说，属于主动学习问题（an active research topic）。这个类型的问题通常都是针对问题和模式不断优化的寻求最优解。比如，

- 比如一个 star join（star join 是多连接查询的一种类型），有些数据库会使用特别的算法应对
- 比如针对并行查询（parallel query），有些数据库会使用特殊的算法应对
- 。。。

针对大型的查询，不断有研究者提出其他算法，尝试代替动态规划算法。而贪婪算法，属于另一个算法类：启发式算法。贪婪算法，按照某个规则（或者 heuristic），保留上一步的最优结果，基于上

一步的结果在这一步继续添加，然后找到这一步的最优结果。还有类似的算法，也是相同的思路，但是不是每一步都保留最优的一个结果。这些算法都可以成为启发式算法。

举个例子，遗传算法（**genetic algorithms**）也执行某个逻辑，但是不是每一步都保留上一步的一个最优结果：

- 一个运算结果是一个查询计划
- 但是这个结果不是一个，而是多个（也就是说每一步保留最优的几个结果，比如 P 个）
- 0）初始时，随机创建 P 个结果
- 1）执行规则后，这 P 个结果对应的 P 个最优结果产生了
- 2）这些结果，再一次执行规则，又产生 P 个最优结果
- 3）这 P 个结果会被随机修改
- 4）步骤 1，2，3 冲突 t 次
- 5）然后，从最终的 P 个结果中选择一个最优的结果

循环次数越多，结果越好

是不是很神奇？不，这本就是自然法则：适者生存！

仅供参考，遗传算法（genetic algorithms）在数据库 PostgreSQL 上实现了，但是我不确定是否默认打开。

当然，数据库实现中还有其他的启发式算法，比如 Simulated Annealing，Iterative Improvement，Two-Phase Optimization 等等。。。但是我不知道这些算法是否应用到企业数据库中了，有可能只是在研究阶段供测试数据库使用。

真正的优化

【可以跳过，因为这一节不重要】

毕竟，上面吧啦吧啦了一大堆，全都是理论。我是一个开发者，不是研究者，我喜欢实际项目中的案例。

我们看看这个数据 SQLite optimizer 的优化的原理。这是一个很轻量的数据库，所以它的优化也比较简单，基于贪婪算法（greedy algorithm），还有附加的逻辑规则可以筛选掉一部分（明显不好的）查询计划：

- SQLite 从不给 CROSS JOIN 运算的表进行重新排序
- **连接运算通过 nested joins 实现**
- 外连接通常就是连接时的顺序
- 。。。
- 在 3.8.0 版本之前，**SQLite 在寻找最优查询计划时使用“Nearest Neighbor”贪婪算法**

骚等一下。。。我找到这个算法了！太凑巧了！

- 从 3.8.0 版本有（2015 发布），SQLite 使用“N Nearest Neighbors”贪婪算法

我们现在看看另一个优化方式。IBM DB2 像全部企业数据库一样，但是重点不说这个，即使这是我转换到 Big Data 前使用的最后一个数据库。

如果哦我们查看一下官方文档 official documentation，可以了解到 DB2 的优化允许用户设置 7 个不同级别的优化：

- 连接时使用贪婪算法
  - 0 – 最低优化，使用索引扫描和 nested-loop join 连接方式，避免查询重写（Query Rewrite）
  - 1 – 低优化
  - 2 – 全优化
- 连接时使用动态规划算法
  - 3 – 现代优化和粗略近似
  - 5 – 全优化，使用启发式算法的所有技术
  - 7 – 全优化，类似 5，没有启发式算法
  - 9 – 最大优化，考虑所有的连接顺序包括笛卡尔积，需求最优的查询计划

我们看到，**DB2 使用到了贪婪算法和动态规划。**当然，他们不会把他们使用的启发式算法分享出来，毕竟查询优化是是一个数据库的核心关键部分。

仅供参考，默认级别是 **5。** 所以默认情况下，优化使用到如下特性：

- **所有可参考的统计信息**，包括频率-数值统计和柱状图统计
- **所有查询重新规则**（包括物化查询表路由）都会使用到，但是一些“计算强度”相关的规则不会使用到，除非在某些极少见的场景下。
- **动态规划算法连接遍历会使用到，不过有点要求：**
  - 限制的使用 inner relation 的混合运算
  - 限制的使用涉及遍历表的星型模式的笛卡尔积运算**（译者注：这里翻译不准确，抱歉）**
- **广泛地使用 access methods**，包括列表预取（注意：我们会看到这是什么意思）、索引 ANDing（一种特殊的对索引的运算）和物化查询表路由。

默认情况下，DB2 使用对连接顺序有启发式算法限制的动态规划算法（dynamic programming limited by heuristics for the join ordering）**（译者注：表达不好，迫不得已，保留原文）。**

其他的“条件”运算（GROUP BY，DISTINCT。。。），都可以通过简单的规则来处理。

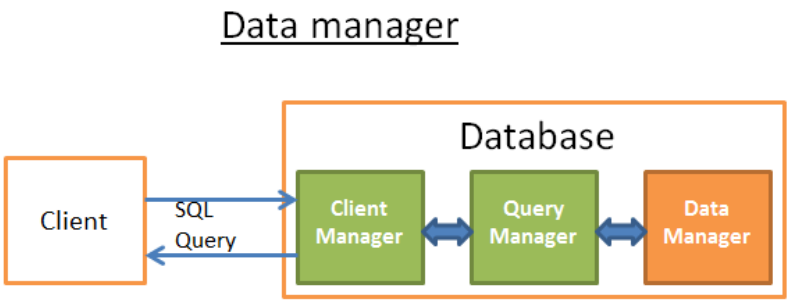
查询计划缓存

因为查询计划的创建需要时间，大多数数据库把查询计划缓存进查询计划缓存（**query plan cache**），这样可以避免相同查询的查询计划重新计算。这是一个的标题，因为数据库需要知道什么时候可以更新那些过期的缓存。基本思路是，设置一个阀值，如果某个表的统计信息的变更高于这个阀值，就刷新涉及这个表的所有查询计划在缓存中的记录。

4.5 查询执行

在这一阶段，我们已经有了一个优化过的查询计划了，这个查询计划会被编译，成为可执行代码。然后，如果有足够的计算资源（内存，CPU），可执行代码就会被查询执行处理。查询语句中的运算符（JOIN，SORT BY。。。）可以顺序执行，可以并行执行；这个取决于查询执行。对于写入或者读取数据，查询执行会与数据管理器交互，数据管理器在下一章介绍。

5 数据管理器



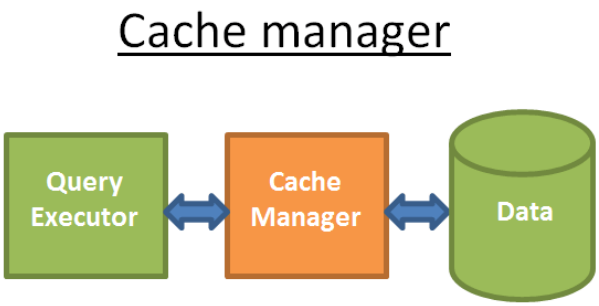
在这一步，查询执行时，需要获取表或者索引的数据。查询执行时，从数据管理器请求获取数据，下面是 2 个问题：

- 关系型数据库使用事务模型。所以不能随时或者任意数据，因为有可能这部分数据此时此刻正在被修改或者锁定。
- **数据检索时数据库里最费时的事情**，所以数据管理器必须足够智能，可以把“要使用”的数据提取加载进内存，然后缓存起来以备后用。

在这一章，我们会分析关系型数据库怎么解决这 2 个问题。我不会讨论数据管理器怎么获取数据，这不重要（而且相关的论文很多很多）。

5.1 Cache 管理

就像我说过的，数据库的瓶颈在磁盘 I/O。为了提升性能，现代数据库都使用了 Cache 管理器。



查询执行先请求 Cache 管理器获取数据，而不是直接从文件系统读取。Cache 管理器有一个常驻内存的缓存，叫 **buffer pool**。从内存里读取数据，牛逼哄哄地提升了数据库的处理速度。不过很难说提高了多少倍，因为速度不是由一个因素简单地决定，还取决于所执行的运算：

- 顺序存取（如：全扫描）与随机存取（如：通过行号存取），
- 读与写

和数据库所使用的磁盘的类型：

- 转速：7.2k/10k/15k rpm HDD
- 是否 SSD
- RAID 类型：RAID 1/5/...

不过可以肯定，内存的速度绝对是硬盘的 100 倍以上，甚至 10 万倍！

但是，有出项了另一个（数据库常见的）问题。Cache 管理器需要在查询执行前，提前把查询用到的数据加载进内存里；否则的话，查询管理器就得在查询执行时，焦急地等待磁盘慢速的处理。

数据预取

上面提到的问题，叫做数据预取。查询执行知道它想要用到的数据是那些，因为查询执行知道这个查询的整个过程同时也通过统计信息知道了磁盘上的数据。下面是一个思路：

- 当查询执行开始处理数据的第一块时
- 它可以请求 Cache 管理器预加载数据的第二块
- 当查询执行开始处理第二块数据时
- 它可以请求 Cache 管理器准备好数据的第三块，然后通知 Cache 管理器第一块数据不需要保留了
- 。。。

Cache 管理器保存了 buffer pool 里的所有数据。想知道某个数据是否还需要保留，Cache 管理器给缓存的数据都增加了一下附加的信息（叫做 **latch**）。

有时候查询执行不知道它需要哪些数据，有些数据库也没有提供这个 功能。实际上，它们用到了一个分析的数据预取方式（比如，如果查询执行请求到数据 1、3、5，这个方式会认为需要提前加载数据 7、9、11）或者随机数据预取方式（在这个情况下，Cache 管理器会简单地从磁盘中加载上一次请求的数据后面的数据）。

为了评测数据预取的效果如何，现代数据库都提供了一个度量方式，叫做 buffer/cache 命中率（**buffer/cache hit ratio**）。这个命中率意思是某个请求所需的数据在 buffer cache 中获取不必从磁盘获取的概率

注意：buffer/cache 命中率很低，并不表示 Cache 功能坏了。想要了解跟过的，可以参阅 Oracle documentation。

然后，buffer 是一段有限的内存，不能无限存储。所以呢，需要清理旧数据腾出空间加载新数据。加载和清除缓存，肯定会带来磁盘 I/O 和网络 I/O 上的成本。如果有一个查询经常执行，但是如果 Cache 管理器反复加载了这个查询的数据然后再清除，很明显这样效率很低。怎么应对怎么情况呢，现代数据库都有一个 buffer 置换策略。

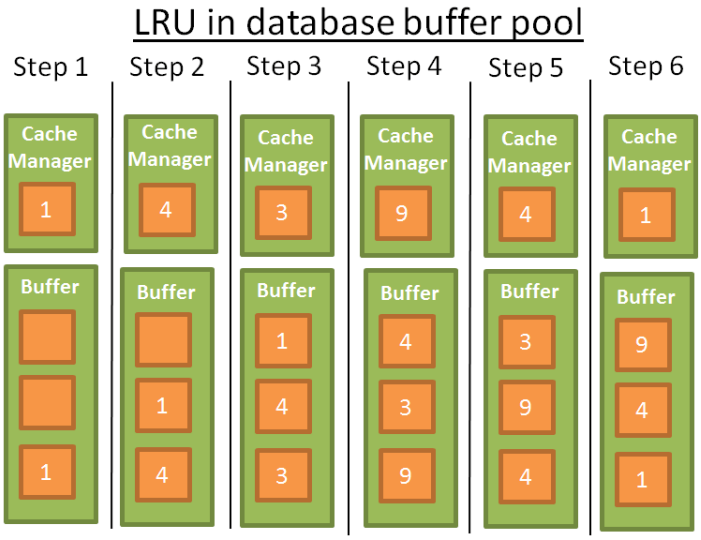
Buffer 置换策略

大多数现代数据库（至少 SQL Server，MySQL，Oracle 和 DB2）都会使用 LRU 算法。

LRU

LRU 全称是 **L**east **R**ecently **U**sed。这个算法的思路是，缓存中的数据，如果正在被使用的数据，多半还会再被使用。

这里是一个更直观的例子：



为了容易理解，我假定 buffer 中的数据都没有被（latches buffer，被锁定的可以清除）锁定。这个简单的例子中，buffer 的大小可以存储 3 个单位的数据：

- 1：Cache 管理器用到数据 1 时，把数据 1 载入空白的 empty buffer 中
- 2：Cache 管理器用到数据 4 时，把数据 4 载入已经占用的 buffer 中
- 3：Cache 管理器用到数据 3 时，把数据 3 载入已经占用的 buffer 中
- 4：Cache 管理器使用数据 9 时，buffer 已经满了，所以数据 1 被清除（因为数据 1 是最久没有被使用的），然后数据 9 载入 buffer
- 5：Cache 管理器使用数据 4 时，数据 4 已经在 buffer 中所以它又变成最近被使用的数据
- 6：Cache 管理器使用数据 1 时，buffer 是满的，所以 data 9 被清除（因为数据 9 是最久没有被使用的），然后数据 1 载入 buffer 中
- 。。。

这个算法运行效果不错哦，但是存在一些限制。给一个巨大的表做全扫描，怎么处理？换句话说，如果某个表或者索引的数据量大于 buffer 时，怎么办？使用这个算法，会清除掉 buffer 中之前的全部缓存数据，但是载入的数据其实只会用到一次。

改进一下

为了防止上面说的情况的发生，有些数据库会添加一个特定的规则。比如，这个文档 Oracle documentation 说明了这种规则：

> *"For very large tables, the database typically uses a direct path read, which loads blocks directly […], to avoid populating the buffer cache. For medium size tables, the database may use a direct read or a cache read. If it decides to use a cache read, then the database places the blocks at the end of the LRU list to prevent the scan from effectively cleaning out the buffer cache."*

（译者注：上面是引用，保留原文，同时翻译如下）

“对于超大的表，数据库通常使用 direst access 读取数据，这种方式直接加载数据块[...]，不会占用 buffer 产生缓存。对于中型的表，数据库可以使用直接读取或者缓存读取，如果想使用缓存读取，数据库会把数据块放在 LRU 列表的结尾，防止把 buffer 中的缓存数据清除干净”

当然也有其他类似做法，比如使用一个改进版的 LRU，叫做 LRU-K。举例，SQL Server 就使用了 LRU-K 并且 K =2。

这个算法的思路是，记录更多的缓存记录信息。简单的 LRU（也可以理解为 LRU-K 并且 K=1），这个算法只能记录每个缓存记录最近被使用的时间。如果是 LRU-K 算法：

- 这个算法还会记录每个缓存记录被累计使用的次数 K
- 除了次数 K，还会有一个权重值
- 如果新数据有一块载入缓存，旧缓存记录如果使用次数 K 比较高，就不会被清除（因为这个缓存记录的权重较高）
- 但是这个算法不会保留那些不再被使用的缓存记录
- 所以，如果缓存记录不被使用，它的权重会随着时间减少

缓存记录的权重需要计算成本，这也是 SQL Server 只使用 K=2。这个值运行情况良好，很多情况下都可以接受。

想了解更多更深入关于 LRU-K 的知识，可以参阅这篇最初始的研究论文（1993）：The LRU-K page replacement algorithm for database disk buffering。

其他算法

当然，有其他的算法解决缓存管理问题，比如：

- 2Q （一个类似 LRU-K 的算法）
- CLOCK （一个类似 LRU-K 的算法）
- MRU （目前使用最广泛，使用了与 LRU 相同的逻辑，但是另一种规则）
- LRFU （Least Recently and Frequently Used）
- 。。。

有些数据库实现了多种算法，默认其中一种。

写 buffer

前面我只说了读 buffer，查询时的数据预取。其实数据库中，还有写 buffer，用于存储数据然后一批批地写入到磁盘；（如果没有写 buffer 就只能）一个个地写入，这样会产生过量多余的磁盘存取操作，非常影响性能。

要记住的是，buffers 存储单位是 **pages** （最小的数据存储单元），不是行（行是数据的肉眼可读的逻辑形式）。如果 buffer 中的某个 page 被修改了但是还没有写入磁盘，被认为是脏数据（dirty page）。有许多算法用来判断脏数据写进磁盘的最佳时机，不过这个话题与事务有很大关联，是我们下一节要讨论的话题。

5.2 事务管理器

Last but not least（译者注：感觉保留原文更好，不过还是翻译一下：最后但并非最不重要），这一节关于事务管理器。我们会探讨事务管理器进程怎么保证每一次查询都在它所在的事务中处理。不过在这之前，我们需要理解 ACID 事务的相关理论。

我是 ACID

一个 ACID 事务，是一个具备如下特性的工组单元：

- **原子性（Atomicity）**：事务的性质就是“要么成功，要么什么都没发生”，即使这个事务持续处理 10 个小时。如果是一个事务崩溃，所有的东西回到事务开始前（这个叫做事务的回滚）。
- **隔离性（Isolation）**：如果 2 个事务 A 和 B 同时处理，A 和 B 的结果必须是相同的，不管哪个先完成。
- **持久性（Durability）**：只要某个事务提交完成，（比如，处理成功），数据就一定要保存在数据库中，不管发生了什么事（崩溃或者错误）。
- **一致性（Consistency）**：只有合法的数据（从关系型约束和函数型约束来看）才能写到数据库中。一致性与原子性和隔离性有关联。

在同一事务中，你可以使用多个 SQL 语句对数据执行 read、create、update 和 delete 操作。麻烦往往在两个事务同时操作同一个数据时发生，一个经典的例子是，帐户 A 转账到帐户 B。假设你 2 个事务：

- 事务 1 的功能是从帐户 A 取款 100RMB，给帐户 B
- 事务 2 的功能是，从帐户 A 取款 50RMB，给帐户 B

如果我们回顾一下 **ACID** 四要素：

- **原子性**保证不管事务 1 处理是发生什么（一个服务器崩溃，或者网络失败），你都不会停在“已经从帐户 A 中取款 100 但是没有转给帐户 B”这种狗血状态（这是一个不一致状态）。
- **隔离性**保证即使事务 1 和事务 2 同时处理，最后的结果肯定是“要么帐户 A 被取款 150RMB 并且帐户 B 得到 150RMB，要么什么事情都没发生”，比如帐户 A 被取款 150RMB 但是帐户 B 只得到了 50 元，这说明事务 2 部分地覆盖了事务 1 的操作（这还是一个不一致状态）。
- **持久性**保证事务 1 在提交后就不会凭空消失，即使数据库崩溃一次
- **一致性**保证了在系统中钱的总量是恒定的，不会多出了也不会少。

【你可以跳过下面这一节，如果你不爱看，这一节剩下的内容，其实都不重要】

大多数现代数据库都不能完全满足隔离性，因为实现这个特性会产生严重的性能损失。所以 SQL 规范定义了 4 个级别的隔离性：

- **Serializable**：最高级别的隔离性。两个同时处理的事务，实现 100%隔离，每个事务有它自己的“世界”。
- **Repeatable read**：每个事务有它自己的“世界”，允许一个意外情况。如果一个事务成功处理完成并且新增了一些数据，这些数据在别的事务或者其他正在处理的事务中是可见的。但是如果事务 A 修改了一些数据然后成功完成，这些数据的修改在其他正在处理的事务中是不可见的。所以，这种事务间对隔离性的突破只在新数据发生，已经存在数据不发生。

举个例子，如果事务 A 执行了“SELECT count(1) from TABLE_X”然后一个新的数据由事务 B 增加和提交，如果事务 A 再执行一次，两次 count(1)的结果不同。

这个叫做幻读（**phantom read**）

- **Read committed**：这是很多数据库（Oracle，MySQL，PostgreSQL 和 SQL Server）的默认行为。这是一种可重复的读，而且也是对隔离性的突破。如果事务 A 读取数据 D，然后这个数据被事务 B 提交并修改（或者删除），如果事务 A 再读取数据 D，会得到被事务 B 修改（或者删除）后的数据。

这个叫做不可重复读（**non-repeatable read**）

- **Read uncommitted**：最低级别的隔离性。这是一个提交的读，而且也是对隔离性的突破。如果一个事务 A 读取数据 D，然后这个数据被事务 B 提交并修改中（事务 B 正在处理中，还没

有完成），如果事务 A 再读取数据 D 会看到修改后的数据。如果事务 B 回滚了，如果事务 A 再读取数据 D，会得到最初的数据；因为数据 D 被“没有发生的”事务 B 修改了（因为它回滚了）。

这个叫做脏读（**dirty read**）

大多数数据库会增加它自己定义的隔离性级别（比如 PostgreSQL，Oracle 和 SQL Server 都实现的 snapshot 隔离）。然而，大多数数据库并没有实现全部级别的 SQL 规范（尤其是 read uncommitted 级别）。

隔离性的默认级别可以允许用户/开发者在连接时设置（只需要添加一行非常简单的代码）

并发控制（Concurrency Control）

真正保证隔离性、一致性和原子性的困难在于同一时间对同一数据的写操作(add，update 和 delete)：

- 如果所有的事务都只是读取数据，那这些事务可以同时处理，也不会互相影响。
- 如果（即使只有一个）事务在修改数据，而同一时间其他事务在读取这个数据，数据库需要提供一个机制屏蔽这个数据的修改对其他事务的影响。另外，还要保证这个被修改的数据不会被其他事务（其他事务看不到修改后的数据）擦除。

这个问题叫做**并发控制**。

解决这个问题最简单的方式是，事务一个一个处理（就是说 顺序地）。但是这个完全不能扩展，，而且在多处理器多核的服务器上只有一个核在工作，效率太低，资源太浪费。。。

一个理想的解决方式是，每次当有一个事务被创建或者取消时，都执行如下操作：

- 检查所有事务中的所有操作
- 检查是否有 2 个（或多个）事务中有部分冲突，比如都在读取或者修改相同的数据
- 涉及冲突的事务，重新安排其中的某些操作的执行顺序，这样可以减少冲突操作
- 按照一个特定的顺序（比如找到没有涉及冲突的事务正在处理的时间）执行冲突操作
- 有些不能安排处理的事务，取消这写事务然后做记录

严格来说，这是一个带有冲突调度的调度问题。精确地说，这是一个极具难度而且比较耗费 CPU 的优化问题。企业数据库不可能允许等待几个小时，然后才给每一个新增的事务找到最优的调度决策。因此呢，只能使用一个不太理想的方式，即使有可能会造成事务冲突时处理时间的浪费，那也没办法。

Lock 管理器

怎么处理这个问题呢，大多数数据库会使用锁机制和（或者）数据版本化（**data versioning**）。因为这是一个大题目，我重点说明一下锁机制，然后会简单说明一下数据版本化。

悲观锁机制（Pessimistic locking）

这个锁机制的思路是：

- 如果某个事务需要一个数据
- 它会给这个数据加锁
- 如果另一个事务也需要这个数据
- 另一个事务必须等待，直到第一个事务给这个数据解锁

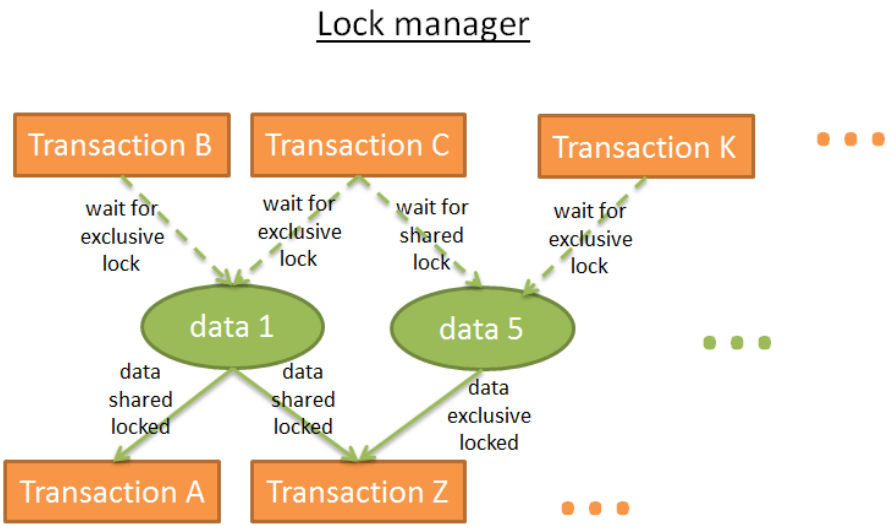
这个叫做互斥锁（**exclusive lock**）。

不过使用悲观锁，对于那些只需要读取数据的事务来说，纯粹是浪费时间，因为它只是想读取数据却被强制等待。**这就是为什么我们要使用另一种锁机制：共享锁**。

使用共享锁的思路：

- 如果一个事务需要读取数据 A
- 它给数据 A 加上“共享锁”，然后读取数据 A
- 如果第二个事务也要读（只是读，没有别的操作）数据 A
- 第二个事务也被数据 A 加上“共享锁”，然后读取数据 A
- 如果第三个事务想要修改数据 A
- 第三个事务给数据 A 加上“互斥锁”，但是要等到前两个事务都给数据 A 解除了它们的“共享锁”，这个“互斥锁”才会生效

然并卵，当某个数据被加上了“互斥锁”，那些只是读取数据的事务依然要浪费时间被强制等待，直到互斥锁被解除才能加上“共享锁”读取数据。

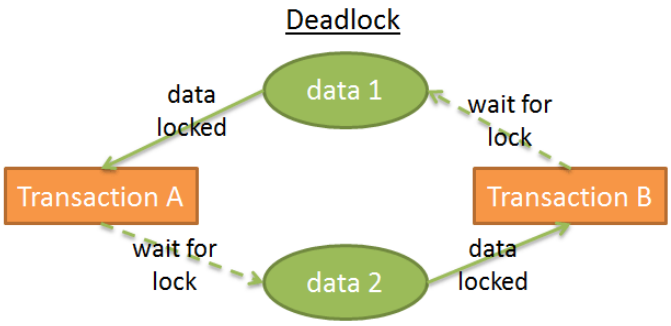


Lock 管理器是一个进程，实现加锁和解锁。从内部实现来看，锁存储在哈希表中（哈希表的 key 是锁对应的数据）并且知道每个数据对应的锁（译者注：这里翻译的表达不够准确，抱歉）：

- 某个数据被哪些事务加锁
- 某个数据有哪些事务等待

死锁（Deadlock）

锁的使用会导致一种特殊的情况，比如两个事务因为某一个数据而陷入无限等待：



如图所示：

- 事务 A 给数据 1 加了互斥锁，并且在等待读取数据 2
- 事务 B 给数据 2 加了互斥锁，并且在等待读取数据 1

这个叫做死锁（**deadlock**）。

在死锁状态中，Lock 管理器需要选择把某个事务取消掉，才能打破死锁状态。取消哪个事务，这个决策不容易：

- 是不是取消掉“修改最少数据的”事务更好（因为这样引发的回滚最少）？
- 是不是取消“最年轻的”事务更好（因为其他事务已经处理了很多）？
- 是不是取消“距离完成需时最少”的事务更好（and avoid a possible starvation）（译者注：不知道括号里面的内容应该怎么翻译，没看明白啊）？
- 如果回滚，有多少事务会受到这次回滚的影响？

在做出选择（取消某个事务）之前，得先判断一下是否存在死锁。

哈希表可以看作一个图（就像前面的图）。如果图中存在一个环路，就说明存在死锁。因为检查环路比较费时（毕竟很多锁组成的图，还是非常大的），所以通常使用一个简单的方式：超时。如果某个锁在一定时间内没有加上，就认为个事务存在死锁状态。

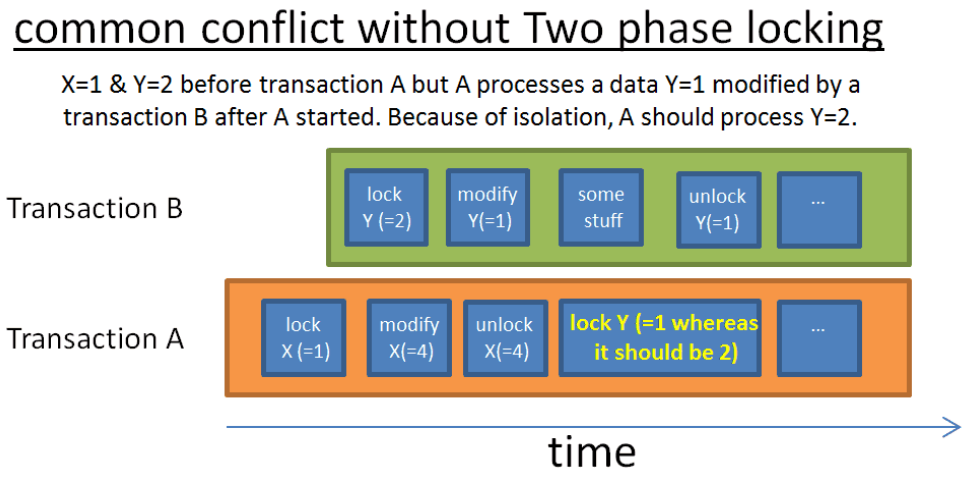
Lock 管理器也可以在加锁前检查一下是否会导致死锁，但是想做得彻底，计算的成本会很大。因此，就引入了一些基本规则来实现预检查。

双相锁机制（Two-phase locking）

保证完全隔离性的最简单的方法是在事务处理开始时加锁，事务处理完成时解锁。这意味着这个事务在处理开始前必须等待相关数据都加锁，然后知道这个事务完成前才给全部解锁。这个办法很有效，但是会导致很大的时间浪费，因为要等待给全部的数据加锁。

有一个更快的方法， **双相锁协议（Two-Phase Locking Protocol，**DB2 和 SQL Server 中都有使用），把一个事务分为两个阶段：

- 生长期（**growing phase**），事务可以加锁但是不能解锁
- 衰退期（**shrinking phase**），where 事务可以解锁（给已经处理过并且不会再处理的数据解锁），但是不能（给新数据）加锁



这个方法的，基于 2 个简单的规则：

- 通过给用不着的数据解锁，可以减少其他事务在这个锁上的等待时间
- 防止出现一种情况：事务处理中对同一个数据的前后多次读取，存在不一致（因为后面对该数据的读取获取到的是被其他事务修改后的数据）。

“双相锁协议”虽然大多数情况下都很棒，但是还是有点瑕疵：如果一个事务要修改某个数据然后解锁，但是这个事务被取消，然后回滚。结局也许会这样，另一个事务读取了这个修改后的数据，片刻之后这个数据被回滚。怎么应对这个问题呢，**要求所有的互斥锁都在事务完成前才能解锁。**

一点思考和总结

当然，真实的数据库会使用一个复杂精妙能够处理各种类型的锁（包括意图锁）的“锁处理系统”，以及不同级别和不同梯度（行级锁，页级锁，区级锁，表级锁和表空间级锁）的控制，但是核心思想是相同的。

上面完全是纯粹基于锁（解决并发控制问题）的方式，数据版本化是另一个可以解决该问题的方式。

数据版本化的思路是这样：

- 每个事务都可以同时修改同一个数据
- 每个事务都有这个数据的一个副本（或者说版本）
- 如果 2 个事务都修改了相同的数据，只有一个可以被真正接受，另一个会被拒绝，然后对应的事务会回滚（可能会再次处理）。

这个办法会有性能损失：

- **某个事务的读不会阻塞其他事务的写**
- **某个事务的写也不会阻塞其他事务的读**
- 也不会存在来自 Lock 管理器的瓶颈

看起来都很不错，除了两个情况让人不满意：2 个事务同时写同一个数据，太费磁盘了（因为要给每个数据搞副本，所以分分钟拖垮磁盘）。

数据版本化和锁机制，是两个不同的角度：**乐观锁（optimistic locking）与悲观锁（pessimistic locking）**。可以说这两种方式各有千秋；实际上取决是使用场景（数据读的多，还是写的多）。有一个 PPT，关于数据版本化的，我强烈推荐 this very good presentation，介绍数据库 PostgreSQL 上“多版本并发控制”的实现。

有些数据库，比如 DB2（DB2 9.7 前）和 SQL Server（除了一些 snapshot 隔离）都只是用了锁机制。其他数据，比如 PostgreSQL，MySQL 和 Oracle 使用了混合方式，既有锁机制也有数据版本化。我没听说哪个数据库只是用了数据版本化（如果你知道，务必告诉我）。

还是老套路，如果你想了解更多，可以参阅主流数据库的文档（比如 MySQL，PostgreSQL 和 Oracle）。

如果你读过了隔离性的多个不同级别，就可以知道提高隔离性的级别时，锁的数量肯定会相应增加，事务在等待中浪费的时间也会相应增加。这就是为什么大多数数据库模式不使用最高的隔离性级别。

Log 管理器

我们已经知道数据库可以通过把数据存储在内存 buffer 中提高性能。但是如果当某个事务处理中时数据库奔溃了，内存中的数据会丢失，这个情况下事务的持久性满足不了。

你可以对磁盘写入任何东西，但是当数据库奔溃后，结果是你写的数据只有一部分在磁盘里，这个情况下事务的原子性也满足不了。

事务产生的任何修改和写入都必须“要么不做要么完成”（undone or finished）（译者注：如果做了一半、不到一半或者一大半反悔了想不做，就得回滚，后面的“undone”都翻译成“回滚”）。

有 2 个方式可以解决这个问题：

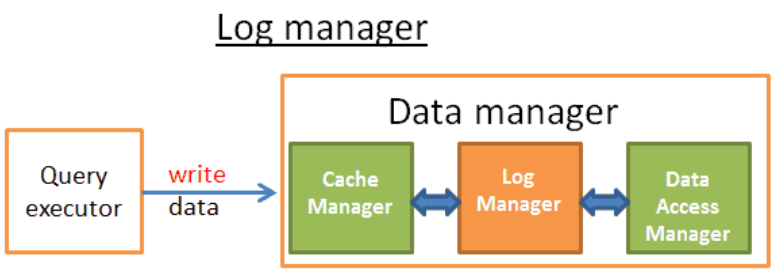
- **Shadow copies/pages**：每个事务都会创建它自己的数据库（或者数据库的一部分）副本，然后在这个副本上处理。如果有错误出现，副本被删除。如果成功了，数据库立刻马上“借助文件系统的诡计”切换到副本中的新数据并删掉旧数据。
- **事务 log（Transaction log）**：事务 log 是一个存储空间。在每一次写入磁盘前，数据库先在事务 log 中写一个记录，如果事务处理中发生了崩溃或者取消，数据库就知道怎么去回滚（或者完成）这个没完成（被崩溃影响）的事务了。

WAL

第 1 种方式 shadow copies/pages 需要占用觉得磁盘空间，尤其是数据库比较大又有很多事务同时处理时。这就是当今数据库都使用**事务 log** 的原因。事务 log 必须保存在一个稳定的存储设备上。这里不必深入讨论存储技术了，但是（至少）要使用 RAID 吧，这样才能保证磁盘故障时依然稳定。

大多数数据库（至少 Oracle，SQL Server，DB2，PostgreSQL，MySQL 和 SQLite）都使用 **Write-Ahead Logging protocol**（WAL）来处理事务 log。WAL 协议包括 3 条规则：

- 1）每一个对数据库的修改，都产生一个 log 记录，而且 log 记录必须在数据写入磁盘前写入事务 log 中
- 2）log 记录都必须按照顺序；log 记录 A 如果发生时间早于 log 记录 B，则必须在 log 记录 B 前面
- 3) When 当一个事务提交完成了，提交顺序也要在事务处理完成前记录在事务 log 中



记录事务 log 这个功能由 Log 管理器实现。让我们直观地看一看，Log 管理器位于 Cache 管理器和 data access manager（把数据写入磁盘）之间，Log 管理器把每一次操作（update/delete/create/commit/rollback）记录到事务 log，注意是在数据写入磁盘前完成。很简单，对吧？

错！就我们之前所探讨的，你应该知道跟数据库相关的任何事情，都与数据库性能有关系。严格来说，这个问题在于寻求一种优良的写 log 方式同时还能保证性能不受影响。如果在事务 log 中做记录的速度太慢，会发生什么，肯定会拖慢一切！

ARIES

早在遥远的 1992 年，IBM 研究员就“发明”了一个 WAL 的强化版本，叫做 ARIES。ARIES 被应用在当今大多数数据库中。ARIES 的设计思想影响非常广泛，虽然不同的实现在逻辑上略有差别。我给前面的“发明”两个字加上引号，是因为根据 MIT course，这个 IBM 的研究员实际上“只有一个好看的事务恢复的实际过程，其他什么也没做”。由于当 ARIES 论文发表时，我才 5 岁，所以我完全不关系当年其他研究员的七嘴八舌。实际上，我提到 ARIES 的目的仅仅是给大家一个提醒，为我们开始下一节做准备。我读过这些论文 research paper on ARIES 的绝大部分，很有意思的！ 在一节，我们只是大概地探讨一下 ARIES，但是我强烈建议大家，如果你也真的想学习知识，去读这些论文。

ARIES 的全称是 **A**lgorithms for **R**ecovery and **I**solation **E**xploiting **S**emantics。

这个技术的目标有 2 个：

- 1）写 log 是性能良好
- 2）恢复时又快又稳

由于种种原因，一个数据需要实现事务回滚：

- 用户可以取消一个事务
- 数据库或者网络故障
- 某个事务会破坏数据库的数据（举例，某个表的某个里上有 UNIQUE 索引约束，而某个事务要添加一个重复的记录）
- 死锁了

有时候（比如，网络故障时），数据库可以恢复某个事务。

怎么做到的呢？要回答这个问题，我们需要理解 log 记录中的信息。

Log

事务中的每一个操作（add/remove/modify）都会产生一条 log 记录。log 记录应该由如下字段构成：

- **LSN**：一个唯一的 **L**og **S**equence **N**umber。LSN 按照时间序列生成。意味着，如果操作 A 发生在操作 B 之前，那么 log A 的 LSN 必定小于 log B 的 LSN
- **TransID**：事务的 ID

- **PageID**：修改的数据的磁盘位置。磁盘存储的最小单位是 page，所以数据的位置，就是数据所在 page 的位置
- **PrevLSN**：同一个事务中某操作前面的操作的 log 记录的链接
- **UNDO**：去掉操作效果（回滚）的方式

举个例子，如果某个操作时 update，其中 UNDO 记录的是 update 之前那个数据的值和状态（value/state）（这是物理 UNDO），或者与一个可以回到之前状态的逆操作（这是逻辑 UNDO）。

- **REDO**：重做某个操作的方式

同样地，REDO 也有 2 个实现方式。可以记录操作后某个数觉得值和状态（value/state），也可以记录一个操作（就是这个操作，可以再来一次）

- 。。。：（仅供参考，一个 ARIES log 还有另外 2 个字段：UndoNxtLSN 和 Type）

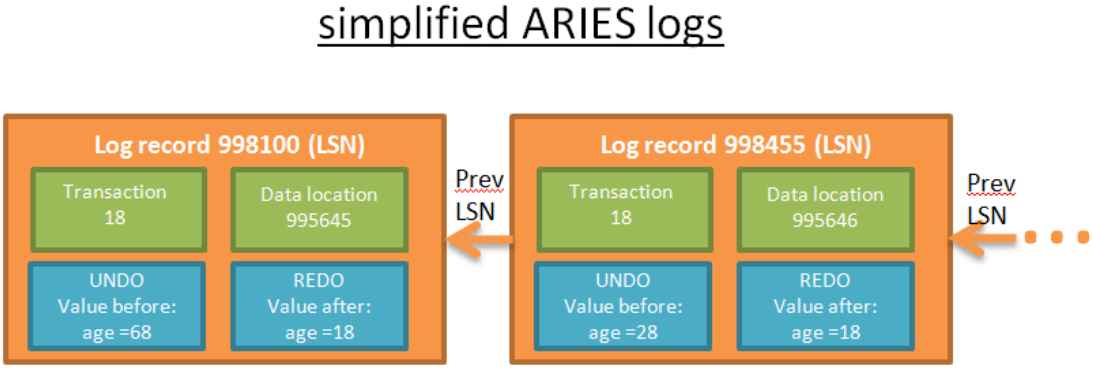
另外，每一个磁盘上用于存储数据（不存储 log）的 page 都在“修改数据的最后一个（或者说上一个）操作”的 log 记录（LSN）中有一个 id。**（译者注：这里的表达应该不太准确，如果有疑问可以参考原文）**

*生成的方式比较复杂，因为 LSN 与 log 存储的方式有关联。但是背后的思想是相同的。

**ARIES 只使用了逻辑 UNDO，因为处理物理 UNDO 实在太麻烦。

注意：依我的浅薄之见识，只有 PostgreSQL 没有使用 UNDO，而是使用一个垃圾收集器（garbage collector）服务。垃圾收集器服务会删掉数据的旧版本，这个与 PostgreSQL 上数据版本化的实现有关系。

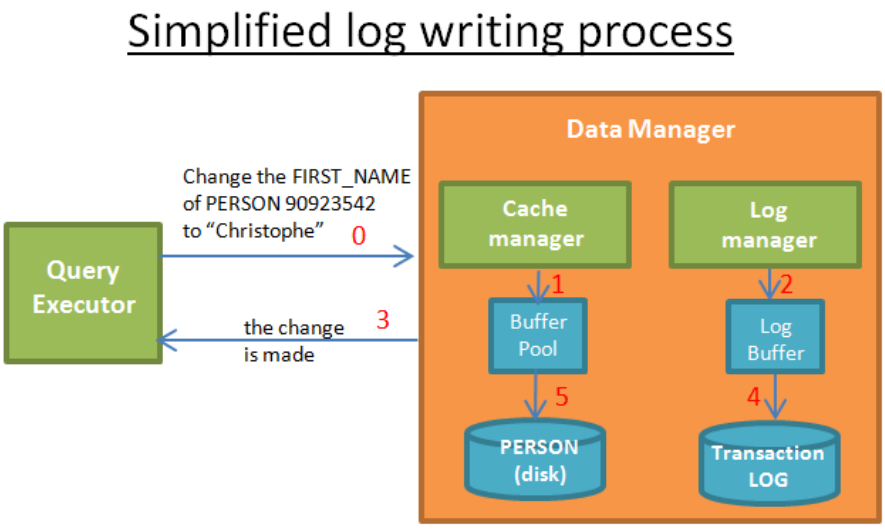
为了让大家更容易理解，下面图中是一个更直观跟简单的 log 记录，记录了查询操作“UPDATE FROM PERSON SET AGE = 18;”。我们分析一下这个查询在事务 18 中的执行。



每一条 log 记录都有独特的 LSN。属于同一个事务的所有 log 记录，这些 log 记录按照时间顺序组织（最后一条 log 记录，就是最后一个操作的对应 log 记录）。

Log Buffer

为了避免写 log 引起性能损失或者性能瓶颈，需要使用 **log buffer**。



当一个查询请求想执行数据修改：

- 1）首先 Cache 管理器会把数据的修改保存在它的 buffer
- 2）其次 Log 管理器在它的 buffer 中保存相关的 log
- 3）查询执行确认操作完成（然后可以处理其他数据修改了）
- 4）（最后）Log 管理器在事务 log 上写入 log。什么时候写 log 的决策由算法实现
- 5）（再最后）Cache 管理器在磁盘上写入数据修改。什么时候写入磁盘的决策由算法实现

当一个事务处理到提交完成，意味着事务中的每一个操作都完成了上面所说的 1、2、3、4、5 个步骤。在事务 log 上写入 log 记录，速度很快，因为仅仅是“在事务 log 的某个地方新增一个 log 记录”而已；而把数据写入磁盘，处理会比较复杂，因为要实现“选择一个读起来更快的方式把数据写入磁盘”。

STEAL 和 FORCE 政策

为了性能，第 5 步有可能是在 commit 之后再完成，因为数据库崩溃后还可以使用 REDO logs 实现事务的恢复。这个就是 **NO-FORCE policy**。

一个数据库可以现在一个 FORCE 政策（比如，第 5 不，必须在事务提交完成前结束），这样可以减少恢复时的工作量。

另一个问题是，要么选择“数据一步一步写入磁盘”（**STEAL policy**），或者选择“buffer 管理器等到提交指令后才一次性把数据写入磁盘”（NO-STEAL）。这两个政策 STEAL 和 NO-STEAL 的选择取决于你的目标：写入快但是使用 UNDO 恢复慢，或者恢复很快？

这里对恢复时不同政策的影响的总结：

- **STEAL/NO-FORCE 需要 UNDO 和 REDO**：性能最高，但是 log 复杂而且恢复也麻烦（像 ARIES）。这是大多数数据库的一致选择。注意：我查阅过许多分析这个政策的研究论文和课程，但是严格来说官方文档并没有类似这样明确的说法
- STEAL/ FORCE 只需要 UNDO
- NO-STEAL/NO-FORCE 只需要 REDO
- NO-STEAL/FORCE 什么都不需要：只是性能非常低下，而且需要浪费大量的内存

恢复部分

好了，现在我们有了很棒的 log，怎么使用呢？

假设一个新手把数据库搞崩溃了（最高指示：数据库崩溃永远是新手的错误）。你们重启数据库，然后开始恢复。

在崩溃后实施 ARIES 恢复，分 3 个步骤：

- **1）The Analysis pass**：恢复进程先读取完整的事务 log，根据这些 log 重建崩溃时的操作时间轴。然后决定哪些事务要回滚（所有没有提交指令的事务，都要回滚），还要决定哪些（崩溃时的）数据需要写入磁盘。
- **2）The Redo pass**：这一步从上一步决定的 log 记录开始，使用 REDO log 把数据库更新到奔溃前的状态。

在 Redo pass 这个步骤，REDO log 按照时间顺序（就是依据 LSN）处理。

对于每一条 log 记录，恢复进程读取“被修改的数据在磁盘上的 page”的 LSN。

如果 LSN(page_on_disk)>=LSN(log_record)，意味着这个数据在奔溃前已经被写入到磁盘（但是数据又被另一个后来的并且在奔溃前完成的操作覆盖了），所以**（译者注：已经是正常状态）**什么都不用做。

如果 LSN(page_on_disk)<LSN(log_record)，意味着这个数据被更新了**（译者注：这个情况表示数据没有写入磁盘，需要 REDO log 重做一下）**。

另外，REDO log 甚至可以用来实现事务的回滚，而且还简化了恢复进程（但是我肯定，当今的数据都没有这个做）。

- **3）The Undo pass**：这一步会回滚所有奔溃时处理还没完成的事务。这个回滚从每一个事务的最后一个 log 记录开始，大致也按照时间顺序处理 UNDO logs（使用 log 记录中的 PrevLSN）。

数据库恢复过程中，必须非常谨慎地确认那些由恢复进程的动作产生的事务 log，这样才可以执行写入磁盘的数据与事务 log 中的数据的同步。有一个办法：删除那些回滚（或者取消）的事务相关的 log 记录，但是这个办法有点麻烦。ARIES 会在事务 log 中记录一些补充 log，逻辑上可以是实现“删除了”那些“作废的”事务的 log 记录。

当某个事务被手动取消了，或者被 Lock 管理器（因为死锁问题）取消了，或者就是一个网络故障，那么第一步的 analysis pass 就用不着了。实际上，那些 REDO 和 UNDO 的信息，都可以从 2 个内存表中的到：

- **一个是 transaction table**（存储所有当前正在处理的事务的状态）
- **另一个是 dirty page table** （存储需要写入磁盘的数据）

这 2 个表由 Cache 管理器和事务管理器一起维护，每一个新事务都会产生对应的记录。因为这 2 个表是内存表，所以如果数据库奔溃，这 2 个表会随之消失。

数据库奔溃后恢复时，第一步 analysis pass 实现从事务 log 中提取信息重建这 2 个内存表。*想加速第一步 analysis pass 的处理，ARIES 提供了一个概念：检查点（**checkpoint**）。这个概念的思路是数据库在写入数据时，也随机地（有控制的随机）把 transaction table、dirty page table 和“最近的”LSN 写入磁盘，这样一来恢复时第一步 analysis pass 就只需要分析这个 LSN 之后的 log 记录。

6 总结

在写这篇文章之前，我知道这是一个大题目，我也知道在这个大题目下写一篇有深度的文章会更费时间。不过我很有信心，毕竟我做到了我完成了，花去的时间是我预期的两倍还多，不过我学到了很多。

如果你想好更广阔更深入地理解数据库，我建议你看看这一篇研究论文“Architecture of a Database System ”。这篇论文（110 页）很棒，详尽地介绍了数据库，而且非计算机专业的人士也可以看懂。这篇论文给了我很多帮助，提供了关于数据库的知识大纲和概要，其中较少关注数据结构和算法，主要是数据库的宏观的架构理论。

如果这篇文章，你阅读得够仔细，你应该能够领略数据库的强大。毕竟这是一篇比较长的文章，所以我还是带你会议一下我们探讨过的那些：

- B+树索引概要
- 数据库概览

- 基于成本的优化原理，还有对连接操作的重点分析
- 数据库 buffer pool 管理概要
- 事务管理概要

当然，数据库里中包含的智慧远远不止这些，比如还有一些棘手的问题我没有提到：

- 数据库集群中全局事务的管理
- 数据库运行时快照的实现
- 数据的高效存储和压缩
- 内存管理机制

所以呢，在健壮的关系型数据库与稚嫩的 NoSQL 数据库之间做选择时，希望你三思而后行。别被我舞蹈，其实有些 NoSQL 数据库还是很强大滴！但是毕竟他们还年轻，在一些对稳定性要求较低的专门的业务领域发光发热。

总而言之，如果有人问你数据库的工作原理，现在的你不必躲避这个问题，直接告诉他：

或者，你给他这个文档！