基础和趋势®
数据库
卷。3, No.4 (2010) 203 - 402
Oc 2011 G.人名
DOI: 10.1561/1900000028

现代B树技术

Goetz格雷夫

内容

1.介绍204

1.1关于b型树的观点204
.21目的和范围206
1.3新的硬件207
1.4概述208

2.基本技术210

2.1数据结构213
2.2尺寸、树高等。215
2.3算法216
2.4数据库中的b棵树221
2.5b棵树与哈希索引为226
2.6总结230

3数据结构和算法231

3.1节点大小232
3.2插值搜索233
3.3可变长度的记录235
3.4标准化密钥237
3.5前缀B-trees 239
3.6 CPU缓存244

3.7键值246重复
3.8位图索引249
3.9数据压缩253
3.10空间管理256
3.11分裂节点258
3.12摘要259

4.交易技术260

4.1锁定和锁定265
4.2幽灵记录268
4.3钥匙范围锁定273
4.4锁定叶边界的关键范围280
.54分离器键范围锁定282
4.6眨眼-树283
4.7锁采集期间的锁286
4.8锁扣接头288
4.9生理学日志记录289
4.10未登录的页面操作293
4.11未被记录的索引创建295
4.12在线索引操作296
4.13事务处理隔离级别300
4.14总结304

5查询处理305

5.1磁盘顺序扫描309
5.2获取行312
5.3覆盖指数313
5.4索引到索引的导航317
5.5利用关键前缀324
5.6订单检索327
5.7单个表的多个索引329
5.8单个索引中的多个表333
5.9嵌套查询和嵌套迭代程序334
5.10更新计划337

5.11分区表和索引340
5.12摘要342

6b形树实用程序343

6.1索引创建344
6.2指数删除349
6.3指数重建350
6.4批量插入352
6.5批量删除357
6.6碎片整理工作359
6.7指数验证364
6.8摘要371

7高级关键结构372

7.1多维ub树373
7.2已分区的b型树375
7.3合并指数378
7.4列存储381
7.5大值385
7.6记录版本386
7.7总结390

8总结和结论392

致谢394年

引用395

基础和趋势®
数据库
卷。3, No.4 (2010) 203 – 402
Oc 2011 G.人名
DOI: 10.1561/1900000028

现代B树技术

Goetz格雷夫

惠普实验室，美国，戈茨。graefe@hp.com

摘要

b树索引发明于大约40年前，不到10年后就被称为无处不在，它已经被用于从手持设备到大型机和服务器场的各种计算系统中。多年来，为了提高效率或添加功能，许多技术被添加到基本设计中。示例包括对结构或内容的更新的分离、实用程序操作，如未记录的事务性索引创建，以及健壮的查询处理，如索引到索引导航期间的优雅退化。本调查回顾了数据库中b树和b树索引的基础知识、与b树相关的事务性技术和查询处理技术、数据库操作中必不可少的b树实用程序，以及许多优化和改进。它的目的是作为一个调查和作为一个参考，使研究人员能够比较索引创新与先进的b树技术，并使专业人员能够选择最适合其数据管理挑战的特性、功能和权衡。

1

介绍

在拜耳和麦克里怀特推出[7]树后不到10年，现在超过25年前，Comer称[7]树索引为无处不在的[27]。Gray和Reuter声称“b树是数据库和文件系统中最重要的访问路径结构”[59]。在数据库、信息检索和文件系统中使用了各种形式和变体的b型树。可以说，由于b型树，世界上的信息就在我们的指尖。

1.1b树的展望

图1.1显示了一个非常简单的b树，其中有一个根节点和四个叶节点。不显示节点内的单个记录和键。叶节点包含键范围不同的记录。根节点包含指向叶节点的指针和分隔叶中的键范围的分隔键。如果叶节点的数量超过了适合于根节点的指针和分隔符键的数量，则会引入一个“分支”节点的中间层。根节点中的分隔符键将分支节点（也称为内部、中间或内部节点）和分隔符分隔

1 .1.关于b型树的观点205

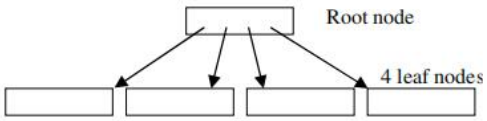


图1.1一个简单的带有根节点和四个叶节点的b树。

分支节点中的键划分叶中的键范围。对于非常大的数据收集，将使用带有多层分支节点的b-树。在用作数据库索引的b型树中，通常有一个或两个分支级别。

对b树的“数据结构视角”的补充是以下的“算法视角”。在排序数组中的二进制搜索允许具有鲁棒性能特征的高效搜索。例如，在10个搜索对象中进行一次搜索^{9}或230项目可以用
只有30个比较。但是，如果数据项数组大于内存，则需要某种形式的分页，通常依赖于虚拟内存或缓冲池。然而，它对于I/O是相当低效的，因为对于除了最后几个比较之外的所有比较，将获取包含数十个或数百个键的整个页面，但只检查一个键。因此，可能会引入一个包含在大数组中进行二进制搜索时最常用的键的缓存。这些是排序数组中的中位数键，每个结果的半数组的中位数，每个结果的四分之一数组的中位数，等等，直到缓存达到页面的大小。实际上，b树的根是这个缓存，它添加了一些灵活性，以启用非2的幂的数组大小，以及有效的插入和删除。如果根页中的键不能将原始的大数组划分为小于单个页面的子数组，则缓存每个子数组的键，在根页面和页面大小的子数组之间形成分支级别。

B-tree索引对于信息检索和数据库管理中需要的各种操作执行得非常好，即使其他索引结构对某些单个索引操作更快。也许它们的名称“B-树”中的“B”应该代表它们在查询、更新和实用程序之间的平衡性能。查询包括精确匹配查询（“=”和“in”谓词）、范围查询（“<”和“binter”谓词）和完整扫描，排序输出如果

必须的更新包括插入、删除、修改与特定键值相关联的现有数据，以及这些操作的“批量”变体，例如批量加载新信息和清除过期记录。实用程序包括创建和删除整个索引、碎片整理和一致性检查。对于所有这些操作，包括实用程序的增量式和在线式变体，b-树还可以实现有效的并发控制和恢复。

1.2目的和范围

许多学生、研究人员和专业人士都知道关于b树指数的基本事实。基本知识包括它们在一个根和许多叶等节点上的组织，根和叶之间的均匀距离，它们的对数高度和对数搜索努力，以及它们在插入和删除时的效率。本调查简要回顾了b树索引的基础知识，但假设读者对关于现代b树技术的更详细和更完整的信息感兴趣。

当涉及到并发控制和恢复等更深层次的主题时，或涉及到增量批量加载和结构一致性检查等实际主题时，通常持有的知识往往不足。b树帮助查询处理的许多方式也是如此。g.，在关系数据库中。这里的目标是使这些知识随时可作为一个调查和作为高级学生或专业人员的参考。

目前的调查在多种方面超越了“经典的”b树参考[7,8,27,59]。首先，介绍了最近的技术，包括研究思想和已验证的实现技术。虽然在这些参考文献中涵盖了b树改进的前二十年，但过去的20年却没有。其次，除了核心数据结构和算法外，本调查还讨论了它们的使用情况，例如在查询处理和有效的更新计划中。最后，介绍了辅助算法，例如碎片整理和一致性检查。

自其发明以来，b型树的基本设计在很多方面都得到了改进。这些改进与

到内存层次结构中的其他级别，如CPU缓存，到多维数据和多维查询，到并发控制技术，如多级别锁定和密钥范围锁定，到实用程序，如在线索引创建，以及到b树的更多方面。这里的另一个目标是将许多这些改进和技术收集在一个文档中。

本调查的重点和主要背景是数据库管理系统中的b树索引，主要是在关系数据库中。这反映在许多具体的解释、例子和论点中。尽管如此，许多技术都很容易适用，或者至少可以转移到b树的其他可能的应用领域，特别是信息检索[83]、文件系统[71]、“无SQL”数据库和最近为web服务和云计算[21,29]推广的关键值存储。

对技术的调查不能提供全面的绩效评估或立即的实施指导。读者仍然必须选择需要的技术或适合特定环境和需求的技术。需要考虑的问题包括预期的数据大小和工作负载、预期的硬件及其内存层次结构、预期的可靠性需求、并行程度和并发控制的需求、支持的数据模型和查询模式等。

.31个新硬件

闪存、闪存设备和其他固态存储技术将改变计算机系统的内存层次，特别是数据管理。例如，大多数当前的软件在内存层次结构中假设了两个级别，即RAM和磁盘，而任何其他级别，如CPU缓存和磁盘缓存，都被硬件及其嵌入式控制软件所隐藏。闪存也可能保持隐藏，可能作为大而快速的虚拟内存或快速磁盘存储。然而，更有可能的设计方法是为数据库而设计的，

似乎是对具有三个甚至更多层次的内存层次结构的显式建模。不仅是外部合并排序等算法，而且

此外，像b树索引这样的存储结构将需要重新设计，甚至可能需要重新实现。

在其他影响中，具有非常快的访问延迟的闪存设备将改变数据库查询处理。它们可能会将盈亏平衡点转向基于索引到索引导航的查询执行计划，而不是大型扫描和大型集合操作，如排序和哈希连接。通过更多的索引到索引导航，调整索引集，包括自动增量索引创建、增长、优化等。将成为未来数据库引擎更受关注的焦点。

正如固态存储将改变数据结构和访问算法的权衡和优化一样，多核处理器也将改变并发控制和恢复的权衡和优化。只有通过对一致状态和事务边界的适当定义，才能启用高度的并发性，并且针对单个事务和系统状态的恢复技术必须支持它们。必须为每种索引和数据结构定义这些一致的中间状态，b树可能是第一个索引结构，这些技术在可生产的数据库系统、文件系统和键值存储中实现。

尽管闪存设备和其他固态存储技术上的数据库和索引未来会发生变化，但目前的调查经常提到适合传统磁盘驱动器的权衡或设计选择，因为许多目前已知和实现的技术都是在这种背景下发明和设计的。其目标是为那些研究和实施适合于新型存储类型的技术的人提供关于b型树的全面的背景知识。

1.4概述

下一节（第2节）列出了它们可以在大学水平的教科书中找到的基本知识。以下部分将介绍针对成熟的数据库管理产品的实现技术。他们的主题是针对数据结构和算法的实现技术

（第3节）、事务性技术（第4节）、使用B树的查询处理（第5节）、特定于B树索引的实用程序操作（第6节），以及具有高级关键结构的B树（第7节）。这些部分可能更适合用于关于数据管理实现技术的高级课程，以及希望深入了解b树索引的专业开发人员。

2

基本技术

b树能够有效地检索按本地排序顺序排列的记录，因为在某种意义上，b树会捕获并保存排序操作的结果。此外，它们在能够容纳插入、删除和更新的表示法中保留了排序工作。b树和排序之间的关系可以通过多种方式加以利用；最常见的方法是，如果存在适当的b树，就可以避免排序操作，最有效的b树创建算法避免随机“插入”操作，而是为了有效的“附加”操作而支付初始排序的成本。

图2.1说明了B-树索引如何保存或缓存排序工作。通过排序操作的输出，b树具有根、叶节点、叶节点等。可以非常有效地创建。后续的扫描可以检索无需额外排序工作的排序数据。此外，

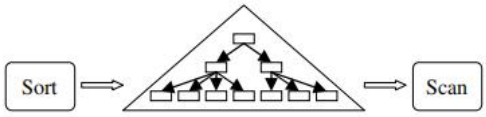


图2.1在b树中缓存排序工作。

为了在任意时间内保持排序工作，b树还允许高效的插入和删除，保留它们的本机排序顺序，并允许在任何时候以排序顺序进行高效扫描。

有序检索有助于许多数据库操作，特别是后续的连接和分组操作。如果后续操作中所需的排序键列表恰好等于或等于b树中排序键的前缀，则为真。然而，事实证明，在更多的情况下，b型树可以节省大量的精力。后面的部分将详细考虑b树索引和数据库查询操作之间的关系。

b树与二叉树有许多相同的特征，这就提出了一个问题，为什么二叉树通常用于内存内的数据结构，而b树通常用于磁盘上的数据。原因很简单：磁盘驱动器一直都是块访问设备，每次访问的开销都很高。b型树通过将节点大小与页面大小相匹配来利用磁盘页面。g.,4 KB.事实上，当今高带宽磁盘上的b树在多个页面的节点上表现最好。g.,64 KB或256 KB。由于通过CPU缓存及其缓存行存取时应该被视为块访问设备，内存中的b树也是有意义的。后面的部分将继续讨论内存层次结构及其对索引的最优数据结构和算法的影响。

b-树更类似于2-3-树，特别是由于这两个数据结构在一个节点中都有可变数量的键和子指针。事实上，b型树可以看作是2-3棵树的概括。有些书把它们都视为具有≥2和b≥2a-1 [92]的（a，b）-树的特殊情况。规范b树节点中的子指针数量在N和2N-1之间变化。对于一个小的页面大小和一个分区-
键的尺寸非常大，这可能确实是2到3之间的范围。通过连接两个二进制节点来表示2-3-树中的单个节点，在b-树中也有一个并行节点，稍后将讨论为闪烁树。

图2.2显示了一个2-3-树中的一个三元节点，它由两个二进制节点表示，其中一个指向三元节点的另一半，而不是一个子节点。一个父节点只有一个指向这个三元节点的指针，并且该节点有三个子指针。

在像b树这样的完美平衡树中，计算节点级别不是从根而是从叶子计算节点级别是有意义的。因此，叶子

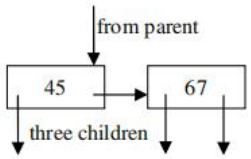


图2.2用二进制节点表示的2-3棵树中的一个三元节点。

有时被称为0级节点，它们是1级节点的子节点，等等。除了子指针的概念外，许多家族术语还被用于连接b型树：父母、祖父母、祖先、后代、兄弟姐妹和表亲。兄弟姐妹是属于同一父节点的子节点。表亲是位于同一b树级的节点，具有不同的父节点，但具有相同的祖父节点。如果第一个共同祖先是曾祖父母，则节点是第二个表亲，等等。然而，家族类比并没有完全使用。两个相邻的兄弟姐妹或表亲被称为邻居，因为在家庭中没有常用的术语来称呼这样的兄弟姐妹。两个相邻节点称为左右邻居；它们的键范围称为相邻的下键范围和上键范围。

在大多数关系数据库管理系统中，b树代码是存储层中的访问方法模块的一部分，该模块还包括缓冲区池管理、锁管理器、日志管理器等等。关系层依赖于存储层，并实现了查询优化、查询执行、目录等。排序和索引维护跨越了这两个层。例如，大型更新可能使用类似于查询执行计划的更新执行计划来尽可能有效地维护每个b树索引，但是单个b树修改以及预读和后写可能保留在存储层中。这种高级更新和预取策略的细节将在后面讨论。

简而言之：

- b型树是针对分页环境进行优化的索引。e.，存储器不支持字节访问。b树节点占据一个页面或一组连续的页面。访问单个记录需要一个字节寻址存储中的缓冲池，如RAM。

- b树是有序的；它们有效地保留了在索引创建期间用于排序的工作。与已排序的数组不同，b型树允许有效的插入和删除。
- 节点是叶子节点或分支节点。其中一个节点被区分为根节点。
- 其他需要知道的术语：父母、祖父母、祖先、孩子、后代、兄弟姐妹、表兄、邻居。
- 大多数实现在每个节点内保持排序顺序，包括叶节点和分支节点，以实现高效的二进制搜索。
- b树是平衡的，在根树搜索中具有统一的路径长度。这就保证了统一有效的搜索。

2.1数据结构

一般来说，b树有三种节点：一个根节点，许多叶节点，以及连接根和叶所需的许多分支节点。根包含至少一个键和至少两个子指针；所有其他节点始终都至少半满。通常所有的节点都有相同的大小，但这并不是真正必需的。

b型树的原始设计在所有节点中都有用户数据。现在更常用的设计只在叶节点中保存用户数据。根节点和分支节点只包含分隔键，以引导搜索算法到正确的叶节点。这些分隔符键可能等于当前或以前数据的键，但唯一的要求是它们可以指导搜索算法。

这种设计被称为B+-tree，但现在在讨论B-树时，它是默认的设计。这个设计的价值是，删除只能影响叶节点，而不影响分支节点，并且分支节点中的分隔键可以在适当的键范围内自由选择。如果后面讨论的支持可变长度记录，分隔键通常非常短。短分隔键增加节点扇形，i。e.，每个节点的子指针的数量，并减少b树的高度，i。e.，在根到叶搜索中访问的节点数。

叶节点中的记录包含一个搜索键和一些相关的信息。此信息可以是与关联的所有列

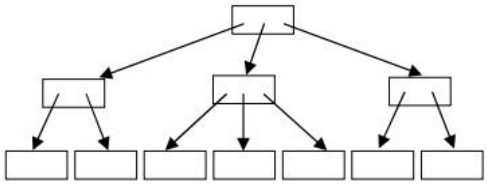


图2.3具有根、分支节点和叶的b型树。

数据库中的一个表，它可以是指向包含所有这些列的记录的指针，也可以是其他任何东西。在本调查的大部分内容中，这些信息的性质、内容和语义并不重要，也没有进一步讨论。

在分支节点和叶中，条目都按排序的顺序保持。其目的是在每个节点内启用快速搜索，通常使用二进制搜索。带有N个分隔键的分支节点包含N个+1子指针，相邻分隔符之间的每个键范围一个
键，一个用于键，低于最小分隔键，一个高于最大分隔键。

图2.3展示了一个比图1.1中更复杂的b型树，包括在叶子和根之间的一级分支节点。在图中，根节点和所有分支节点的扇形输出值为2或3。在存储在磁盘上的B树索引中，扇出输出由磁盘页、子指针和分隔符键的大小决定。在图2.3和下面的许多图中都省略了键，除非在手头的讨论中需要它们。

在所有可能的节点到节点的指针中，真正只需要子指针。许多实现还维护邻居指针，有时只在叶节点之间，有时只在一个方向上。一些罕见的实现也使用了父指针，e。g.，一个西门子的产品[80]。父指针的问题是，当父节点被移动或拆分时，它们会在许多子节点中强制进行更新。在基于磁盘的b树中，所有这些指针都被表示为页面标识符。

b树节点可能包含许多附加的字段，通常是在一个页头中。对于一致性检查，有表或索引标识符加上B树级别，对于叶页以0开始；对于空间

管理，有一个记录计数；对于具有可变大小记录的空间管理，有槽数、字节数和最低记录偏移；对于数据压缩，可能有共享密钥前缀，包括其大小加上其他压缩技术所需的信息；对于预写日志记录和恢复，通常有页面LSN（日志序列号）[95]；对于并发控制，特别是在共享内存系统中，可能有关于当前锁的信息；对于有效的键范围锁定、一致性检查和页面移动，如在碎片整理中，可能有栅栏键，i。e.，祖先页中的分隔符键的副本。下一节将讨论每个字段及其目的及其用途。

- 叶节点包含键值和一些相关的信息。在大多数b树中，包括根节点在内的分支节点包含分隔键和子指针，但没有相关信息。
- 儿童指针是必不可少的。兄弟姐妹指针通常是实现的，但并不是真正必需的。父指针很少被使用。
- b树节点通常包含一个固定格式的页面头、一个可变大小的固定大小的插槽数组和一个可变大小的数据区域。标头包含一个插槽计数器、与压缩和恢复有关的信息等等。插槽
  为可变大小的记录提供空间管理。

2.2尺寸、树高等。

在传统的数据库设计中，b树节点的典型大小为4-8KB。基于多重分析[57,86]，更大的b树节点对于今天的磁盘驱动器似乎更有效，但仍然很少在实践中使用。分隔符键的大小可以和记录一样大，但它也可以要小得多，正如在后面的前缀b树一节中所讨论的那样。因此，典型的扇形输出，i。e.，子指针或子指针的数量，有时只有数十个，通常是数百个，有时是数千个。

如果一个b树包含N个记录和L个记录，那么b树需要N/L个叶节点。如果每个父母的平均孩子数量

是F，分支级别的数量是日志吗$_F$(N/L).例如，图2.3中的b树有9个叶节点，一个扇形输出的F = 3，因此日志为39个=2个分支级别。根据惯例，b树的高度是2（叶子上方的高度）或3（包括叶子的水平）。为了反映根节点通常有不同的扇出事实，这个表达式被舍入。事实上，在一些随机插入和删除之后，节点中的空间利用率会因节点而有所不同。b型树的平均空间利用率通常为70%左右的[75]，但在实践中使用和稍后讨论的各种策略可能会导致更高的空间利用率。我们在这里的目标不是要是精确的，而是要显示关键的效果、基本的计算，以及各种选择和参数的数量级。

如果单个分支节点可以指向数百个子节点，那么根和叶之间的距离通常很小，并且所有b树节点的99%或更多是叶。换句话说，曾祖父母和更远的祖先在实践中是罕见的。因此，对于基于根到叶b树遍历的随机搜索的性能，只处理1%的b树索引，因此可能只有1%的数据库决定了大部分性能。例如，在内存中保持经常使用的b树索引的根目录有利于许多搜索，而对内存或缓存空间的成本很少。

- b树深度（沿根到叶路径的节点）的记录数量是对数的。它通常很小。
- 通常，b型树中99%以上的节点都是叶节点。
- b树页面在50%到100%之间被填充，允许插入和删除以及分割和合并节点。随机更新后的平均利用率约为70%。

2.3算法

b树最基本，也是最关键的算法是搜索。给定b树的搜索键或其前缀的特定值，其目标是尽可能正确和有效地找到b树中与搜索键匹配的所有条目。对于范围查询，搜索会找到满足谓词的最低键。

搜索需要一次从根到叶的传递。在每个分支节点中，搜索找到比搜索键更小或更大的相邻分隔键，然后继续跟踪这两个分隔键之间的子指针。

在一个叶中的L个记录之间的二进制搜索期间的比较次数是对数$_2$(L)，忽略舍入效应。类似地，在分支节点中的F个子指针之间的二进制搜索也需要进行日志记录$_2$(F)比较。有N个记录和L个记录的B树的叶节点数为N/L。b型树的深度是对数的$_F$（N/L），这也是在根到叶搜索中访问的分支节点的数量。在检查分支节点和叶节点的搜索中，比较的数量是日志$_F$（N/L）×日志$_2$(F) +日志$_2$(L).通过对数代数的初等规则，乘积项简化为对数$_2$（N/L），然后将整个表达式简化为日志$_2$(N).换句话说，节点大小和记录大小可能会在此计算中产生次级舍入效应，但记录计数是对b树中根到叶搜索中的比较数量的唯一主要影响。

图2.4显示了b树的一些关键值。对值31的搜索从根目录开始。键值7和89之间的指针在后面指向适当的分支节点。由于搜索键比该节点中的所有键都大，所以最右边的指针后面指向一个叶子。该节点内的搜索确定B树中不存在键值31。搜索键值23将导致图2.4中的中心节点，假设一个约定是一个分隔符键作为一个键范围的包含上界。
当在分支处找到值23时，搜索无法终止

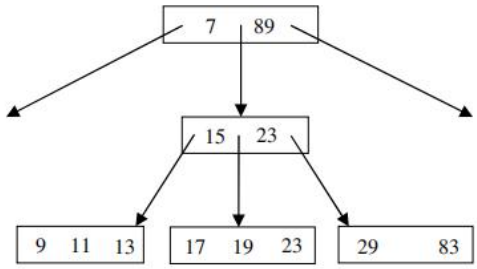


图2.4b型树与根到叶的搜索。

水平这是因为大多数b树搜索的目的是检索附加到每个键上的信息，并且信息内容在大多数b树实现中只存在于b树实现的叶子中。此外，从图2.4中的键值15可以看出，在有效叶条目中某个时间可能存在的键，即使删除了叶条目，仍可以继续作为非叶节点中的分隔键。

搜索完成后将完成精确匹配的查询，但是范围查询必须扫描从范围的低端到高端的叶节点。如果邻居指针存在于b树中，则扫描可以使用它们。否则，必须使用父节点和祖父节点及其子指针。为了利用多个异步请求，e。g.，对于存储在磁盘阵列或网络连接存储中的b树索引，需要父节点和祖父节点。依赖于邻居指针的范围扫描仅限于一次一个异步预取，因此不适合存储设备阵列或虚拟化存储。

插入以搜索正确的以放置新记录的叶子开始。如果该叶有所需的空闲空间，则插入已完成。否则，一个称为“溢出”的情况，需要将叶子分成两个叶子，并且必须插入一个新的分隔键插入父节点。如果父节点已满，则拆分父节点，并将分隔键插入相应的父节点。如果需要分割根节点，b树将再增加一层，i。e.，具有两个子节点和只有一个分隔符键的新根节点。换句话说，许多树的数据结构的深度生长在叶子处，而b型树的深度生长在根部。这就是保证b树中完美平衡的方法。在叶级，b-树只在宽度上增长，由每个父节点中可变数量的子节点启用。在b-树的一些实现中，旧的根页面变成了新的根页面，旧的根内容被分布到两个新分配的节点中。如果修改数据库目录中根节点的页面标识符代价昂贵，或者页面标识符缓存在编译的查询执行计划中，这是一种有价值的技术。

图2.5显示了图2.4中插入键22和由此产生的叶子分裂后的b树。注意，可以自由选择传播到父节点的分隔键；分隔的任何键值

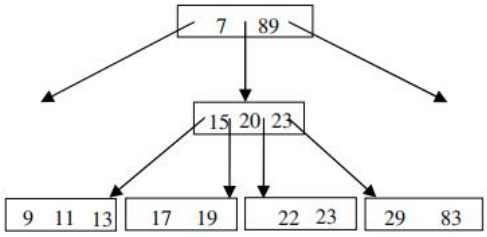


图2.5b型树，带有插入和叶片分裂。

由拆分产生的两个节点是可接受的。这对于可变长度的键特别有用：可以使用可能最短的分隔符键，以减少父节点中的空间需求。 一些b树的实现尽可能多地延迟分割，例如通过兄弟树之间的负载平衡，这样一个完整的节点就可以为插入腾出空间。这种设计提高了代码的复杂性，同时也提高了空间的利用率。如果从存储设备进行的数据传输是瓶颈，则高空间利用率可以实现高扫描率。此外，分裂而新的页面分配可能会迫使在扫描期间进行额外的搜索操作，这在基于磁盘的b树中是昂贵的。

删除还从搜索包含的正确叶子开始
适当的记录。如果该叶子最终只有不到一半的满，一种称为“下流”的情况，负载平衡或与兄弟节点合并都可以确保传统的b树不变，即除了根之外的所有节点至少都是半满的。合并两个兄弟节点可能会导致其父节点中的欠流。如果根节点的唯一两个子节点合并，生成的节点将成为根，旧的根将被删除。换句话说，b型树的深度既在根部生长又缩小。如果根节点的页面标识符如前面提到的那样被缓存，那么将所有内容移动到根节点并取消对根节点的两个子节点的分配可能是可行的。

图2.6显示了删除键值23后的图2.5中的b树。由于下流，两个叶子被合并了。请注意，分隔符键23没有被删除，因为它仍然服务于所需的功能。

然而，许多实现避免了负载平衡和合并的复杂性，并简单地让欠流持续存在。后续的插入或碎片整理可能会得到解决

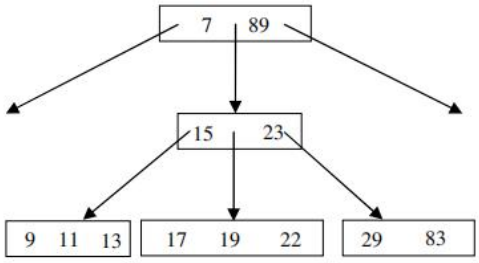


图2.6带有负载平衡的删除。

它稍后。最近一项关于b型树的最坏情况和平均情况行为的研究得出结论：“添加树的周期性重建，。”..数据结构。..在理论上在很多方面都优于标准的B-+树[和]。..在删除时进行重新平衡可能被认为是有害的“[116]”。

b树记录中的关键字段的更新通常需要在一个位置删除并插入b树中的另一个位置。正在进行非关键字固定长度字段的更新。如果记录包含可变长度的字段，则记录大小的更改可能会导致类似于插入或删除的溢出或下流。

最后的基本b树算法是创建b树。实际上，有两种算法，即随机插入和先验排序。一些数据库产品在最初的版本中使用了随机插入，但它们的客户发现创建大型索引的速度非常慢。在创建b树之前对未来的索引条目进行排序，可以提高许多效率，从大量的I/O节省到节省CPU工作的各种技术。随着未来索引大于可用缓冲池，越来越多的插入需要读取、更新和写入页面。数据库系统也可能还需要在恢复日志中记录每一个这样的更改，而现在大多数系统都使用非记录的索引创建，这将在后面讨论。最后，附加操作流还鼓励了磁盘上的b树布局，它允许使用最少的磁盘搜索数进行高效扫描。数据库系统的高效排序算法已经在[46]的其他地方讨论过。

- 如果在每个节点中使用二进制搜索，那么除了舍入效应外，搜索中的比较数量与记录和节点大小无关。

- B树同时支持平等（精确匹配）谓词和范围谓词。有序扫描可以利用邻居指针或祖先节点进行深度（多页）预读。
- 插入将使用现有的可用空间，或将一个完整的节点分割成两个半完整的节点。拆分需要向父节点添加一个分隔符键和一个子指针。如果根节点分裂，则需要一个新的根节点，并且b-树会增长一级。
- 删除可以合并半满的节点。许多实现忽略了这种情况，而是依赖于b树的后续插入或碎片整理（重组）。
- 通过重复随机插入加载b树非常缓慢；对未来的b树条目进行排序，可以有效地创建索引。

2.4B-数据库中的树

在回顾了b树作为数据结构的基础知识之后，还需要回顾b树作为索引的基础知识，例如在数据库系统中，b树索引已经是必不可少的和无处不在的几十年。数据库查询处理的最新发展主要集中在大型扫描的改进上。g.，通过在并发查询[33,132]之间共享扫描，通过在许多查询[17,121]中减少扫描量的柱状数据布局，或者通过特殊硬件的谓词评估，如FPGAs。数据库服务器中闪存设备的出现可能会导致数据库查询处理中更多的索引使用——它们的快速访问时间鼓励小的随机访问，而具有高容量和高带宽的传统磁盘驱动器支持大的顺序访问。作为大多数系统中b树索引的默认选择，数据库中b树索引的各种角色和使用模式值得关注。我们在这里关注关系数据库，因为它们的概念模型非常接近于所有数据库系统以及其他存储服务的存储层中使用的记录和字段。

在关系数据库中，所有数据都逻辑地组织在表中，由名称标识的列和组成表主键的列中的唯一值标识的行。表之间的关系在外键约束中被捕获。行之间的关系是

用外键列表示，其中包含其他地方的主键值的副本。其他形式的完整性约束包括一个或多个列的唯一性；唯一性约束通常使用b树索引强制执行。

数据库表的最简单表示是堆，它是一个保存记录的页面集合，尽管通常是按插入的顺序。通过页面标识符和插槽号（见下面的第3.3节）来识别和定位各个记录，其中页面标识符可能包括一个设备标识符。当一个记录由于更新而增长时，它可能需要移动到一个新的页面。在这种情况下，要么原始位置保留“转发”信息，要么保留对旧位置的所有引用，e。g.，在索引中，必须进行更新。在前一种情况下，所有未来的访问都会产生额外的开销，可能是磁盘读取的成本；在后一种情况下，一个看似简单的变化可能会导致不可预见和不可预测的重大成本。

图2.7显示了堆文件中的记录（实线）和页面（虚线）。记录的大小是不同的。对两个记录的修改改变了它们的大小，并强制将记录内容移动到另一个页面，并在原始位置留下转发指针（虚线）。如果索引指向此文件中的记录，则转发不会影响索引内容，但会影响后续查询中的访问时间。

如果使用b树结构而不是堆来存储表中的所有列，那么在这里将它称为主索引。其他常用的名称包括集群索引或由索引组织的表。在次级索引中，通常也称为非集群索引，每个条目必须包含对主索引中的行或记录的引用。此引用可以是主索引中的搜索键，也可以是包含页面标识符的记录标识符。下面“参考”一词经常用来指其中任何一个。在某些上下文中，对主索引中的记录的引用也被称为书签。

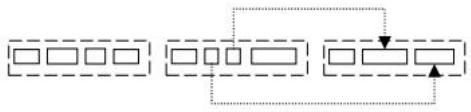


图2.7具有可变长度记录和转发指针的堆式文件。

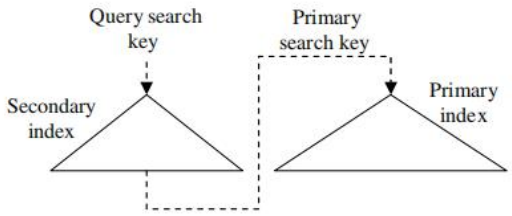


图2.8使用搜索键的索引导航。

这两种设计，即通过搜索键引用和通过记录标识符引用，都有优缺点[68]；没有完美的设计。前一种设计要求在主索引中进行每次搜索后，在主索引中进行根到叶搜索。图2.8说明了当表的主数据结构是主索引，而次索引中的引用使用主索引中的搜索键时的双索引搜索。从查询中提取的搜索键需要在辅助索引中进行索引搜索和从根到叶的遍历。与辅助索引中的键相关联的信息是针对主索引的搜索键。因此，在次索引中成功搜索之后，需要在主索引中进行另一个b树搜索，包括根到叶遍历。

后一种设计允许在搜索辅助索引后更快地访问主索引中的记录。然而，当主索引中的叶分裂时，此设计需要在所有相关的次索引中进行许多更新。由于许多I/O操作和b树搜索，这些更新非常昂贵，它们并不频繁，总是令人惊讶，它们足够频繁，会造成破坏，并且由于并发控制和日志记录，它们会造成巨大的惩罚。

也可以进行组合，使用页面标识符作为提示，使用搜索键作为后退键。这种设计有一些内在的困难。g.，当一个被引用的页面被取消分配，然后再重新分配给一个不同的数据结构时。最后，一些系统在堆文件上使用集群索引；他们的目标是在可能的情况下保持堆文件的排序，但仍然能够通过记录标识符进行快速的记录访问。

在数据库中，所有b树键必须是唯一的，即使用户定义的b树键列不能是唯一的。在主索引中，正确的检索需要唯一的键。例如，在辅助索引中找到的每个引用必须引导查询到主索引中的一条记录——因此，在搜索键引用设计中，主索引中的搜索键必须是明确的。如果用户定义的主索引的搜索键是一个人的姓氏，那么像“Smith”这样的值不太可能安全地识别主索引中的单个记录。

在辅助索引中，正确的删除需要唯一的键。否则，在删除主索引中的逻辑行及其记录之后，可能会删除非唯一次索引中的错误条目。例如，如果辅助索引中的用户定义搜索是一个人的名字，那么删除主索引中包含“BobSmith”的记录必须只删除辅助索引中的正确匹配记录，而不是所有具有搜索键“Bob”的记录或随机包含这样的记录。

如果在索引创建期间指定的搜索键由于唯一性限制而是唯一的，那么用户定义的搜索键就足够了。当然，一旦在索引结构中依赖于逻辑完整性约束，就必须防止删除完整性约束或随后进行索引重组。否则，必须向用户定义的索引键中添加一些人工字段。对于主索引，一些系统使用全局唯一的“数据库键”，如微秒时间戳，一些系统在表中使用唯一的整数值，而一些系统在用户定义的索引键中具有相同值的b树条目中使用唯一的整数值。对于辅助索引，大多数系统只是将引用添加到主索引的搜索键中。

b树条目按其整个唯一键进行排序。在一个主要索引中，这有助于有效的检索；在一个次要索引中，它有助于有效的删除。此外，每个唯一搜索键的排序引用列表支持有效的列表交集和并集。例如，对于一个查询谓词“a=5和B=15”，可以从列A和列B上的索引中获得已排序的引用列表，并通过一个简单的合并算法将它们的交集计算出来。

表和索引之间的关系不必像到目前为止所讨论的那样紧密和简单。一个表可能已经计算出了列

那些根本没有存储的，e。g.，两个日期（时间戳）列之间的差值（时间间隔）。另一方面，辅助索引可以组织在这样的索引上，在这种情况下，必须存储该列的副本。一个索引甚至可能包括有效地从另一个表中复制值的计算列。g.，如果存在适当的功能依赖项和外键约束，订单明细表可能包括客户标识符（从订单表）或客户名称（从客户表）。

另一个通常是固定的，但不需要固定的关系，是唯一性约束和索引之间的关系。许多系统在定义唯一性约束时自动创建一个索引，并在删除该约束时删除该索引。旧的系统根本不支持唯一性约束，而只支持唯一的索引。即使在同一列集上的索引已经存在，也会创建该索引，并且即使它在未来的查询中很有用，该索引也会被删除。另一种设计仅仅要求在唯一性约束激活时存在具有适当列集的索引。对于使用现有有用索引即时定义唯一性约束，可能的设计是在任何索引中每次插入和删除期间计算唯一键的数量。在像b树这样的排序索引中，应该为每个键前缀i维护一个计数。e.，仅对于第一个键字段，第一个和第二个键字段在一起，等等。所需的比较实际上是免费的，因为它们是搜索正确的插入或删除点的必要部分。如果唯一键值的计数等于索引中的记录计数，则会立即验证一个新的唯一性约束。

最后，表和索引可以水平划分（分成行集）或垂直划分（分成列集），后面将进行讨论。分区通常是不相交的，但这并不是真正必需的。水平分区可以应用于一个表，从而使该表的所有索引都遵循相同的分区规则，有时也称为“本地索引”。或者，分区可以单独应用于每个索引，辅助索引使用自己的分区规则与主索引不同，主索引有时被称为“全局索引”。一般来说，是物理数据库的设计或分离

逻辑表和物理索引仍然是一个机会和创新的领域。

- b型树在数据库和信息检索中普遍存在。
- 如果有多个b型树是相关的，则为e。g.，数据库表的主索引和辅助索引，指针可以是物理地址（记录标识符）或逻辑引用（主索引中的搜索键）。这两种选择都不是完美的，这两种选择都被使用过了。
- b树条目必须是唯一的，以确保正确的更新和删除。存在各种机制，通过添加一个人工键值来强制唯一性。
- 传统的数据库设计严格地连接表和b树，比真正需要的要严格得多。

2.5b棵树与哈希索引

令人惊讶的是，b树索引已经变得无处不在，而哈希索引却没有，至少在数据库系统中没有。有两种观点似乎强烈支持散列索引。首先，由于每次查找，哈希索引可以节省一个I/O，从而节省I/O成本，而b树每次搜索都需要一个完整的根到叶遍历。其次，由于有效的比较和地址计算，哈希索引和哈希值也应该节省CPU的工作量。然而，正如以下段落所解释的那样，这两种论点的有效性都非常有限。此外，与哈希索引相比，b-树在索引创建、范围谓词、排序检索、并发控制中的幻影保护等方面都比哈希索引具有实质性的优势。下面的段落也将讨论这些优点。这里提到的所有技术都将在后面的章节中进行更深入的解释。

关于I/O节省，事实证明，相当简单的实现技术可以使b树索引在这方面与哈希索引竞争。大多数b型树都有100秒或1000秒的扇形树。例如，对于8 KB的节点和20字节的记录，70%的利用率意味着每个父节点140个子节点。对于较大的节点大小

（比如64 KB）、良好的碎片整理（启用子指针的运行长度编码，比如平均2个字节）、使用前缀和后缀截断的键压缩（比如平均每个条目4个字节）、70%的利用率意味着每个父节点有5600个子节点。因此，从根到叶的路径很短，在一个b型树中，超过99%甚至99.9%的页面是叶节点。这些考虑必须与传统规则结合，即许多数据库服务器的内存大小等于存储大小的1-3%。今天和将来，这一比例可能会更高，内存数据库高达100%。换句话说，对于缓冲区池中为“温暖”的任何b树索引，所有分支节点都将出现在缓冲区池中。因此，每个b树搜索只需要一个I/O，即叶页。此外，还可以将分支节点提取到缓冲池中，以准备重复的查找操作，甚至可能固定在缓冲池中。如果它们被固定住，就可以应用进一步的优化，e。g.，将分隔符键扩展到一个单独的数组中，这样插值搜索是最有效的，用内存指针形式的子指针替换或扩充页面标识符形式的子指针，等等。

图2.9说明了该参数。除了叶节点之外，所有b树级别都很容易融入到RAM内存中的缓冲池中。对于页页，缓冲区池可能使用最近使用最少的（LRU）替换策略。因此，对于具有随机搜索键的搜索，只需要一个I/O操作，类似于在数据库系统中可用的哈希索引。

在CPU节省方面，b树索引可以使用一些简单的实现技术与哈希索引竞争。b树索引支持各种各样的搜索键，但它们也支持非常简单的搜索键

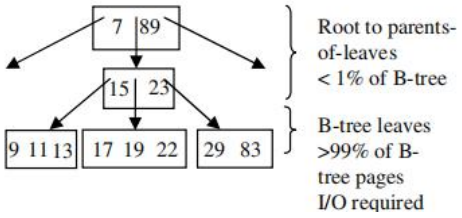


图2.9b级树级别和缓冲级别。

比如哈希值。当可以使用哈希索引时，哈希值上的b树也将提供足够的功能。在其他情况下，一个“穷人的标准化键”可以被使用，甚至是充分的，使所有额外的比较工作没有必要。后面的部分将讨论标准化键、穷人的规范化键，以及将穷人的标准化键缓存在“间接向量”中，这是可变大小的记录所必需的。总之，穷人的归一化键和间接向量可以表现得类似于哈希值和哈希桶。

b树也允许直接地址计算。具体来说，插值搜索可以比二值搜索更快地指导搜索。后面的部分讨论了插值搜索，包括通过在两个插值步骤后切换到二进制搜索来避免纯插值的异常行为，等等。

虽然b树索引可以与基于一些实现技术的哈希索引竞争，但b树也比哈希索引有明显的优势。例如，b型树中的空间管理就非常简单。在最简单的实现中，完整的节点被分成两半，空的节点被删除。为了使哈希索引优雅地增长，已经发明了多种方案，但没有一种方案似乎如此简单和健壮。哈希索引的优雅收缩算法并不广为人知。

b树对哈希索引最强的参数可能与多字段索引和关键值的不均匀分布有关。多个字段上的哈希索引需要对所有这些字段进行搜索键，以便能够计算出哈希值。另一方面，b树索引可以有效地支持对索引键i的前缀的精确匹配查询。e.，任意数量的领先索引字段。通过这种方式，带有N个搜索键的b树可以和N个哈希索引一样有用。实际上，b-树索引可以支持许多其他形式的查询；甚至不需要受限的字段是[82]-树的排序顺序中的前导字段。

对于关键值的不均匀（“偏态”）分布，想象一个带有10的表^{9}需要在10%的行中具有相同值的列上使用辅助索引的行。哈希索引需要引入溢出页面，并提供用于索引创建、插入、搜索、并发控制、恢复、一致性检查等的附加代码。

例如，当删除表中的一行时，在找到并删除辅助索引中的正确条目之前，需要进行昂贵的搜索，因此可能需要合并溢出页面。在b树中，条目总是唯一的，如果有必要，通过向前面讨论的搜索键添加一个字段。在散列索引中，附加的代码需要额外的执行时间以及额外的测试和维护工作。由于b树中定义良好的排序顺序，因此在任何索引函数中都不需要特殊的代码，也不需要额外的时间。

另一个支持b型树的有力论据是索引的创建。在提取未来的索引条目并对它们进行排序之后，b-树的创建非常简单、非常有效，即使对于最大的数据收集也是如此。在大多数管理大数据的系统中，都有一种有效的、通用的排序算法。如果可能的话，为哈希索引创建同样有效的索引将需要一个特殊目的的算法。通过重复的随机插入来创建索引对于b-树和哈希索引都是极其低效的。在线索引创建（带有并发数据库更新）的技术是众所周知的，并广泛应用于b树，但不是哈希索引。

与哈希索引相比，b型树的一个明显优势是支持有序扫描和范围谓词。有序扫描对于关键列和设置操作非常重要，例如合并连接和分组；对于非关键列，范围谓词通常更为重要。换句话说，b-树对于关系数据库中的关键列和非关键列都优于哈希索引，在在线分析处理中也被称为维度和度量。订购在并发控制方面也有优势，特别是通过键范围锁定（稍后详细介绍）而不是只锁定键值。

综上所述，这些参数更倾向于b型树，而不是哈希索引，它可以作为数据库和许多其他数据收集的一般索引技术。在哈希索引似乎有优势的地方，适当的b树实现技术会将其最小化。因此，很少有数据库实现团队在具有高效益和努力比率的机会或特性中找到散列索引，特别是如果在任何情况下都需要b树索引，以支持范围查询和有序扫描。

虽然10 KB的节点可能会导致具有多个级别分支节点级别的b树，但1 MB的节点可能不会。换句话说，上述考虑可能适用于闪存上的b树索引，但可能不适用于磁盘上。对于磁盘，最好是缓存内存中的所有分支节点，并使用相当小的叶节点，这样传输带宽和缓冲区空间都不会浪费在不需要的记录上。

- b树索引是普遍存在的，而哈希索引则不是，尽管哈希索引承诺通过在哈希目录中的直接地址计算和一个I/O进行精确匹配的查找。
- 如果需要，b树软件也可以提供类似的好处。此外，b树支持基于排序的高效索引创建，支持精确匹配谓词和部分谓词，在关键值之间出现重复或分布倾斜的情况下的优雅退化，以及有序扫描。

2.6总结

在本节关于基本数据结构的总结中，b-树是有序的，平衡的搜索树优化的块访问设备，如磁盘。它们保证了各种类型的搜索以及插入、删除和更新的良好性能。因此，它们特别适合于数据库，事实上在数据库中普遍存在了几十年。

随着时间的推移，许多技术已经被发明和实现，超越了基本的算法和数据结构。这些实际的改进将在下面的几节中介绍。

3

数据结构与算法

本节主要介绍在成熟的数据管理系统中发现的数据结构和算法，但通常不在大学级别的教科书中发现；后面的部分将介绍事务性技术、b树及其在数据库查询处理中的使用，以及b树实用程序。

虽然下面只有一个子部分被命名为“数据压缩”，但几乎所有的子部分都与某种形式的压缩有关：每条记录存储更少的字节，描述多个记录在一起，在每次搜索中比较更少的字节，在每次更新中修改更少的字节，避免碎片和浪费空间。在空间和时间上的效率是本节的主题。

下面的子部分使得第一组属于节点的大小和内部结构，下一组用于特定于b树的压缩，最后一组用于管理可用的自由空间。个别子部分中的大多数技术都独立于其他技术，尽管某些组合可能会简化它们的实现。例如，前缀和后缀截断需要详细的，而且可能是过多的记录保存，除非键值被规范化为二进制字符串。

3.1节点大小

甚至最早的关于b树的论文也讨论了磁盘[7]上b树的最优节点大小。它主要由访问延迟和传输带宽以及记录大小的控制。高延迟和高带宽都增加了最佳节点大小；因此，对于现代磁盘的最佳节点大小接近1 MB，而在flash设备上的最佳节点大小只有几个KB [50]。具有相同访问延迟和传输时间的节点大小是一种很有前途的启发式方法——它保证了至少一半理论最优的持续传输带宽，以及至少一半理论最优的I/O速率。它是通过乘以访问延迟和传输带宽来计算的。例如，对于具有5 ms访问延迟和200 MB/s传输带宽的磁盘，这将导致1 MB。估计的访问延迟为0.1 ms和100 MB/s的传输带宽导致10 KB作为flash设备上b树的一个很有前途的节点大小。

为了进行更精确的优化，目标是最大化每单位I/O时间的比较次数。这个计算的例子已经可以在原始的b树论文[7]中找到。这个优化假设的目标是优化根到叶搜索而不是大范围扫描，I/O时间而不是CPU努力是瓶颈，二进制搜索使用节点，和固定的比较根到叶b树搜索独立于上面讨论的节点大小。

图3.1显示了与[57]中类似的计算结果。它假设页面填充的70%是20个字节的记录，这在辅助索引中很典型。例如，在一个包含143条记录的4 KB的页面中，二进制文件

页面大小	记录	结点	输入输出时间	公用事业
1.2.1.1.2.1	/页	功用	[ms]	/时间
4	143	7.163	5.020	1.427
16	573	9.163	5.080	1.804
64	2,294	11.163	5.320	2.098
128	4,588	12.163	5.640	2.157
256	9,175	13.163	6.280	2.096
1,024	36,700	15.163	10.120	1.498
4,096	146,801	17.163	25.480	0.674

图3.1实用程序的页面大小为一个传统的磁盘。

搜索平均要进行7次以上的比较。比较的数量被称为节点对搜索索引的效用。图3.1中的I/O时间是假设5 ms访问时间和200 MB/s（突发）传输带宽计算的。上面的启发式方法建议页面大小为5 ms×200 MB/s = 1,000 KB。128 KB的b树节点能够进行与磁盘设备时间进行最多的比较（二进制搜索）。历史上，4 KB的普通磁盘页对于传统磁盘驱动器上的b树索引来说远不是最优的。不同的记录大小和不同的设备将导致b树索引的最佳页面大小不同。最重要的是，基于闪存设备的设备可以在没有显著不同传输带宽的情况下实现100倍的访问时间。最优的b树节点大小将会小得多，e。g.,2 KB [50].

- 节点大小应该根据底层存储的延迟和带宽进行优化。例如，对于传统磁盘和半导体存储器的最佳页面大小就会有所不同。

.23插值搜索

[1]与二进制搜索一样，插值搜索使用了剩余搜索间隔的概念，最初包括整个页面。插值搜索不像二进值搜索那样检查剩余间隔中心的键，而是估计所寻找的键值的位置，通常使用基于剩余间隔中的最低和最高键值的线性插值。对于一些键，e。g.，由业务操作中的发票编号等顺序过程生成的人工标识符值，插值搜索工作得非常好。

在最好的情况下，插值搜索实际上是不可战胜的。如果订单号和发票号是按顺序分配的，请考虑订单中订单号列上的索引。由于每个阶数只存在一次，因此在数百个阶数之间进行插值

[1] 本节的大部分内容都来自于[45]。

或者甚至是一个b树节点内的数千条记录，都会立即引导搜索到正确的记录。

然而，在最坏的情况下，由于键值的分布不均匀，纯插值搜索的性能等于线性搜索。在N个键[36,107]之间搜索的理论复杂度为0（log log N），对于实际页面大小为2到4个步骤。因此，如果在3或4个步骤之后还没有找到搜索的密钥，那么实际的密钥分布是不统一的，最好使用二进制搜索来执行剩余的搜索。

从纯插值搜索到纯二值搜索，一个渐进的过渡可能会得到回报。如果插值搜索将搜索引导到剩余间隔的一端，但不是直接引导到搜索的键值，则二进制搜索的剩余间隔可能非常小或非常大。因此，似乎建议偏向最后一个插值步骤，使所寻找的键很可能在较小的剩余间隔中。

初始插值计算可能使用页面中可能的最低和最高值、最低和最高的实际值，或基于所有当前值的回归线。后一种技术可以通过相关计算来增强，以引导初始搜索步骤走向插值或二进制搜索。快速得出回归和相关系数所需的和和计数可以很容易地在更新页面中的单个记录时进行增量维护。

图3.2显示了两个b树节点及其关键值。在上面的一个中，槽数和键值之间的相关性非常高（>0.998）。斜率和截距分别为3.1和5.9（插槽编号以0开头）。对键值12的插值搜索立即探测槽号（12-5.9）÷3.1=2（四舍五入），这就是键所在

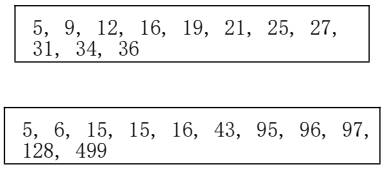


图3.2示例关键值。

.33个可变长度的记录，235个

值12确实可以找到。换句话说，如果位置和关键值之间的相关性非常强，那么插值搜索是很有前途的。在图3.2所示的下b树节点中，斜率和截距分别为-64和31。更重要的是，相关系数要低得多（<值为0.75）。毫不奇怪，对键值97的插值搜索开始在插槽（97-64）÷31 = 5上进行探测，而键值97的正确插槽数是8。因此，如果位置和关键值之间的相关性较弱，二进制搜索是更有前途的方法。

. 如果页面内的键值分布接近统一，插值搜索需要更少的比较，产生更少的缓存故障。阶数等人工标识符是插值搜索的理想情况。

. 对于键值分布不均匀的情况，各种技术都可以防止重复的错误插值。

3.3可变长度记录

虽然b树通常用于叶子中的固定长度记录和分支节点中的固定长度分隔键来解释，但几乎所有数据库系统中的b树都支持可变长度记录和可变长度分隔键。因此，b树节点内的空间管理并不简单。

在固定长度的页面中，包括b树和堆文件中的可变长度记录的标准设计，使用了一个具有固定大小的条目的间接向量（也称为插槽数组）。每个条目代表一条记录。一个条目必须包含记录的字节偏移量，并可能包含附加的信息，e。g.，记录的大小。

图3.3显示了数据库中磁盘页面中最重要的部分。页面标题，显示在页面的最左边，包含索引标识符、b树级别（用于一致性检查）、记录计数等。图3.3中后面是间接向量。在堆文件中，插槽在删除记录后仍保持未使用，以确保剩余的有效记录保留其记录标识符。在b树中，

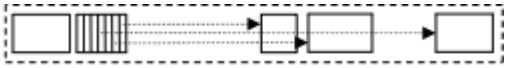


图3.3一个带有页面标题、间接向量和可变长度记录的数据库页面。

插入或删除需要移动一些插槽条目，以确保二进制搜索可以正确工作。（图4.2节中的图4.7显示了这种传统设计的替代方案，由于记录序列中有意存在的间隙，移动更少。）每个使用的槽都包含指向记录的指针（页面内字节偏移的形式）。在图中，间接向量从左到右增长，记录集从右到左增长。相反的设计也有可能。让两个数据结构相互增长，可以同样很好地实现许多小记录或更少的大记录。

为了进行有效的二进制搜索，将间接向量中的条目按它们的搜索键进行排序。不需要对条目的偏移量进行排序。e.，记录的放置位置。例如，图3.3左半部分的插槽序列与右半部分的记录序列不同。对于一致性检查和压缩或可用空间合并，只需要对偏移量的排序顺序，这可以通过记录插入、改变大小的记录更新或碎片整理实用程序调用。

记录插入需要记录和在间接向量中输入的空闲空间。在标准设计中，间接向量从页面的一端增长，记录所占用的数据空间从另一端增长。数据空间通常可以通过增加到中间的空闲空间来很快找到记录的空闲空间。条目的可用空间需要在已排序的间接向量中找到正确的位置，然后根据需要移动条目。平均而言，一半的间接方向必须移动一个位置。

记录删除的速度很快，因为它通常只是在数据空间中留下一个空白。但是，它必须保持间接向量的密集和排序，因此需要像插入一样移动。一些最近的设计需要更少的移动[12]。一些设计还在分支节点中分离分隔键和子指针，以实现更有效

压缩以及在每个分支节点内更有效的搜索。下面还将讨论这些技术。

- 可变大小的记录可以通过页面内的一个间接级别有效地得到支持。
- 间接向量中的移位操作可以通过间隙（无效条目）来最小化。

3.4规范化密钥

为了降低比较的成本，许多b树的实现将键转换为二进制字符串，这样简单的二进制比较就足以在索引创建期间对记录进行排序，并指导在b树中的搜索到正确的记录。原始键的排序顺序和二进制字符串的排序顺序是相同的，并且所有的比较都是等价的。这个二进制字符串可以编码多个列，它们的排序方向(e。g.，和排序规则，包括本地字符(e。g.，不区分大小写的德语)、字符串长度或字符串终止符等。

关键规范化是一种非常古老的技术。单例[118]已经没有提到过它，大概是因为它似乎是一个众所周知的或琐碎的概念：“整数比较被用来对标准化浮点数进行排序。”"

图3.4说明了基于一个整数列和两个字符串列的想法。初始单位（显示下划线）表示主键列是否包含有效值。使用0
对于空值，而1对于其他值，则确保空值“排序”
”低于所有其他值。如果整数列值不是空，则它存储在接下来的32位中。有符号的整数需要反转一些位，以确保正确的排序顺序，就像浮点值一样

整数第一个字符串	第二 细绳	归一化密钥
2“流”	错误“	1 0…0 0000 0000 0010_1 flow\0_1 error\0
3“花”	罕见”	1 0…0 0000 0000 0011_1 flower\0_1 rare\0
1024零		1 0…0 0100 0000 0000 0_1 brush\0
空“”		0_1 \0 0

图3.4归一化键。

刷
空的

需要适当地处理指数、尾数和两个符号位。图3.4假设第一列是无符号的。下面的单位（也显示下划线）表示第一个字符串列是否包含有效值。这个值在这里显示为文本，但实际上应该以适当的二进制格式存储。字符串终止符号（如图/0所示）标记了字符串的末尾。需要一个终止符号，以确保正确的排序顺序。例如，长度指示器会破坏规范化键的主要值，即使用简单的二进制比较进行排序。如果字符串终止符号可以作为某些字符串中的有效字符出现，则二进制表示必须提供比字母表所包含的符号多一个符号。注意差异
字符串列（第三行）中的缺失值和空字符串（第四行）之间的表示。

对于某些排序序列，“规范化键”会丢失信息。
一个典型的例子是一种将小写字母和大小写字母按不区分大小写的顺序进行排序和索引的语言。在这种情况下，两个不同的原始字符串可能映射到相同的规范化键，并且不可能从规范化键来决定使用了哪种原始样式。解决这个问题的一个方案是同时存储规范化密钥和原始字符串值。第二个解决方案是在规范化键上附加最小信息，以便精确恢复原始写入风格。第三个解决方案，特定于b树索引，是只在分支节点中使用规范化键；回想一下，分支节点中的键值只是引导搜索到正确的子节点，而不包含用户数据。

在许多操作系统中，都提供了适当的函数来从本地化的字符串值、日期值或时间值来计算标准化键。例如，此功能用于列出适合于本地语言的目录中的文件。为数字数据类型添加规范化相对简单，对多个规范化值的连接也是如此。但是，数据库代码不能依赖于这样的操作系统代码。依赖于操作系统对数据库索引的支持的问题是更新的频率。操作系统可能会由于代码或本地排序顺序的定义中的错误或扩展而更新其规范化代码；它是

但是，如果这样的更新悄无声地使现有的大型数据库索引不正确，这是不可接受的。

规范化键的另一个问题是，它们往往比原始字符串值更长，特别是对于某些语言以及它们的顺序、排序和索引查找的复杂规则。规范化键的压缩似乎是很有可能的，但在文献中似乎还缺少一个详细的描述。因此，规范化键目前主要用于内部b树节点，它们简化了前缀和后缀截断的实现，但从来不需要恢复原始键值。

- 规范化键可以通过传统的硬件指令进行比较，比对国际排序顺序、升序与降序排序等元数据的逐列插值要快得多。
- 规范化键可以比传统表示更长，但易于压缩。
- 有些系统在分支节点中使用规范化键，但在叶节点中没有。

.53前缀B树

一旦键被规范化为一个简单的二进制字符串，另一个Btree优化就会变得更容易实现，即前缀和后缀截断或压缩[10]。如果没有关键规范化，这些技术将需要相当多的簿记，即使它们只应用于整个关键字段，而不是单个字节；通过关键规范化，它们的实现相对简单。

前缀截断将分析b树节点中的键，并只存储该公共前缀一次，并将其从该节点中存储的所有键中进行截断。节省存储空间允许增加每个叶的记录数量和增加分支节点的扇形输出。此外，在搜索过程中进行比较时，不需要考虑被截断的键字节。

图3.5显示了不带有前缀截断和带有前缀截断的b树节点中相同的记录。很明显，后一种表示更有效。可以组合前缀

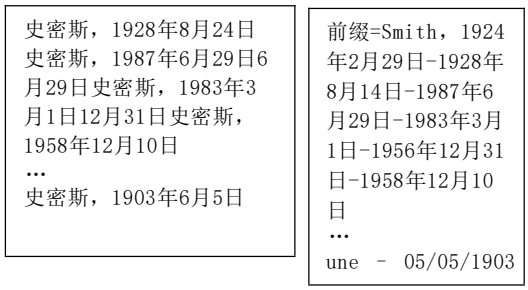


图3.5一个没有和没有前缀截断的b树节点。

截断与一些额外的压缩技术，e。g.，从指定的生日中删除符号。当然，它总是需要权衡运行时性能和存储效率的提高与包括测试工作在内的实现复杂性。

前缀截断可以应用于整个节点或节点内的某些键的子集。代码的简单性要求从b树节点中的所有条目中截断相同的前缀。此外，还可以基于当前在一个节点中保存的实际键，或基于父页（可能是其他祖先页）中的分隔键所定义的可能键范围，应用前缀截断。代码的简单性，特别是对于插入，主张基于最大可能的键范围的前缀截断，即使基于实际键的前缀截断可能产生更好的压缩[87]。如果前缀截断是基于实际的键，那么插入一个新键可能会强制重新格式化所有现有键。在极端情况下，一个新记录可能比b树页面中的空闲空间要小得多，但它的插入可能会迫使页面分裂。

b树页面的最大可能键范围可以通过在每个节点中保留两个栅栏键来捕获，即在拆分节点时张贴在父节点中的分隔键的副本。图4.11（在第4.4节中）说明了B树索引中多个节点中的栅栏键。栅栏键在b树实现中有多个好处，e。g.，用于钥匙范围锁定。关于前缀截断，页面的两个栅栏键所共享的前导字节通过页面中所有当前和未来的键值来定义字节。同时，前缀截断减少了栅栏键和后缀截断所造成的开销

（在拆分叶节点时应用）确保分隔键和栅栏键始终尽可能短。

前缀截断与插值搜索相互作用。特别是，如果插值计算使用固定的和有限的精度，截断公共前缀可以使更准确的插值。因此，规范化键、前缀截断、后缀截断和插值搜索在实现中是一个可能的组合。

对前缀截断技术的另一种非常不同的方法是偏移值编码[28]。它用于排序的高性能实现，特别是在排序运行和合并逻辑[72]中。在此表示中，每个字符串都与排序序列中的直接前缀进行比较，共享前缀被替换为其长度的指示。保留符号位以使指示器保持顺序，i。e.，短共享前缀排序晚于长共享前缀。结果与这个偏移量的数据相结合，这样单个机器指令可以比较偏移量和值。这种表示比统一应用于整个页面的前缀截断节省了更多的空间。它非常适合顺序扫描和合并，但不适合二值搜索或插值搜索。相反，trie表示可以尝试结合前缀截断和二进制搜索的优点，但它在很少的数据库系统中使用。可能的原因是代码的复杂性和更新开销。

即使前缀截断没有在b树及其页面格式中实现，它也可以被用来进行更快的比较，从而进行更快的搜索。以下技术可以称为动态前缀截断。在父节点中搜索正确的子指针时，将检查子指针两侧的两个键。如果它们在某些前导字节上达成一致，那么通过跟随子指针找到的所有键必须在相同的字节上达成一致，这可以在所有后续的比较中跳过。没有必要实际上对相邻的两个分隔键进行比较，因为通过对这些分隔键与搜索键进行必要的比较，可以很容易地获得所需的信息。换句话说，可以利用动态前缀截断，而不需要向b树中的根到叶搜索添加比较步骤。

例如，假设在图3.5左侧显示的b树节点内有一个二进制搜索，剩余的搜索间隔为

“史密斯，杰克”到“史密斯，杰森”。因此，搜索参数必须在这个范围内，并且也必须以“史密斯，Ja”开始。对于所有剩余的比较，可以假定这个前缀，因此在此搜索中的所有剩余比较中跳过。请注意，动态前缀截断也适用于使用前缀截断存储的b树节点。在本例中，在所有剩余的比较中，都可以跳过被截断的前缀“Smith，J”之外的字符串“a”。

前缀截断可以用于Btree中的所有节点，而后缀截断则专门用于分支节点[10]中的分隔符键。前缀截断在叶节点中最为有效，而后缀截断主要影响分支节点和根节点。当一个叶子分裂成两个相邻的叶子时，需要一个新的分隔键。分隔符不是选择左键的最高键或最低的键，而是作为叶子中分隔这两个键的最短字符串。

例如，假设图3.6中所示的关键值位于需要拆分的节点的中间。精确的中心靠近长箭头。分割节点的最小密钥需要至少9个字母，包括给定名称的第一个字母。另一方面，如果短箭头之间的分裂点是可以接受的，那么一个字母就足够了。对定义可接受分隔点范围的两个键进行一次比较可以确定最短的分隔键。例如，在图3.6中，“约翰逊，露西”和“史密斯，埃里克”之间的比较显示了它们在第一个字母中的第一个区别，这表明只有一个字母的分隔符键就足够了。任何字母

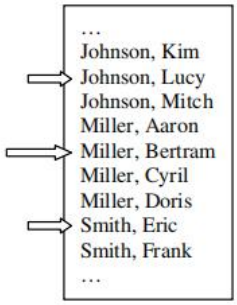


图3.6在分叶期间找到分隔符键。

可以选择大于J且不大于S的值。不需要字母实际出现在当前的键值中。

不仅在分割叶节点时，而且在分割分支节点时，都很容易应用后缀截短。然而，这个想法的问题是，祖父节点中的分隔符键不仅必须引导搜索到正确的父节点，而且还必须引导搜索到正确的叶。换句话说，再次应用后缀截断可能会引导搜索到左子树中的最高节点，而不是搜索到右子树中的最低节点，反之亦然。幸运的是，如果所有b树的99%节点是叶子，其余99%的节点是叶子的直接亲本，那么额外的截断最多可以占1%节点的1%。因此，这个有问题的想法，即使它能完美地工作，可能也不会对b树的大小或搜索性能产生实质性的影响。

图3.7说明了这个问题。上面b树中的分隔符键集被缩短的键“g”分割，但叶条目集则不是。因此，对“gh”键的从根到叶的搜索将被引导到正确的子树，从而失败，这显然是错误的。正确的解决方案是基于原始的分隔符键“gp”来指导搜索。换句话说，当分支节点被分割时，无需应用进一步的后缀截断。当拆分一个分支节点时，唯一的选择是拆分点和在那里找到的键。

- 一种简单的压缩技术是识别所有键值共享的前缀，并只存储一次前缀，这在叶页中尤其有效。

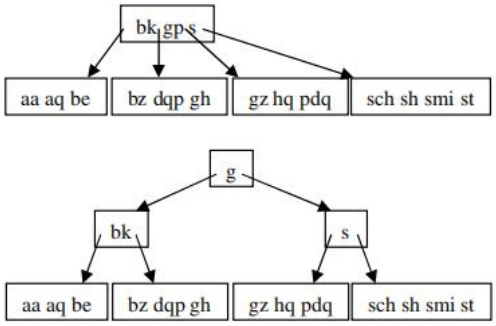


图3.7后缀截断错误。

- 另外，二进制搜索和插值搜索可以忽略剩余搜索间隔的下界和上界所共享的关键字节。在根叶搜索中，这种动态前缀截断从父项传递到子。
- 分支页面中的键值不需要是实际的键值。它们只需要指导从根到叶的搜索。当在分隔叶页时发布分隔键时，一个好的选择是在中间附近分隔的最短值。
- 偏移值编码将每个键值与其相邻键值进行比较，并截断共享前缀。它比页面前缀截断更好的压缩，但禁用了有效的二进制搜索和插值搜索。
- 归一化键显著降低了前缀和后缀截断以及偏移值编码的实现复杂度。

3.6个CPU缓存

缓存错误对搜索成本的影响贡献了很大的一部分
在b树页面中生成。如果需要用多个键搜索[128]树，并且可以修改搜索操作的序列，则可以利用时间局部性。否则，就需要优化数据结构。指令的缓存故障可以通过使用规范化键来减少——将单个字段与国际排序顺序、排序顺序等进行比较，再加上对模式信息的解释，可能需要大量的代码，而两个规范化键可以通过单个硬件指令进行比较。此外，规范化键不仅简化了前缀和后缀截断的实现，而且还简化了旨在减少数据访问的缓存故障的优化的实现。事实上，许多优化似乎只有在使用规范化键的情况下才实用。

在应用了前缀截断之后，二进制搜索中的许多比较都是由前几个字节决定的。即使在记录中没有使用规范化键，例如，在b树的叶子中，存储规范化键的一些字节也可以加快比较速度。要是只有这几个字节是就好了

存储，而不是整个标准化键，这样它们可以决定许多但不是所有的比较，它们被称为“穷人的标准化键”[41]。

为了启用键比较和搜索，而对数据记录没有缓存错误，穷人的规范化键可以是间接向量元素中的一个附加字段。该设计已成功地应用于字母排序[101]的实现中，在b树页面[87]中也同样有益。

另一方面，希望保持间接向量中的每个元素都较小。虽然传统设计通常包括图3.3中讨论中提到的间接向量元素中的记录大小，但如果没有访问相关记录，几乎无法访问记录长度。因此，指示记录长度的字段可以与主记录放置，而不是在间接向量中。

图3.8说明了这样一个b树页面，其键表示三个欧洲国家。左边是页眉和间接向量，右边是可变大小的记录。穷人的标准化键，在这里用一个字母表示，被保留在间接向量中。主记录包含记录的总大小和键的剩余字节。搜索“丹麦”可以通过穷人的标准化密钥消除所有记录，而不会导致主记录的缓存故障。另一方面，搜索“芬兰”可以依靠穷人的标准化键来进行二进制搜索，但最终必须访问“法国”的主记录。虽然图3.8中穷人的标准化键只包含一个字母，但2或4个字节似乎更合适，这取决于页面大小更大。例如，在一个为闪存及其快速访问延迟而优化的小型数据库页面中，2字节可能是最佳的；而在为传统磁盘及其快速传输带宽而优化的大型数据库页面中，4字节可能是最佳的。

另一种设计不是将间接向量组织为线性数组，而是作为缓存线的b树。其中每个节点的大小

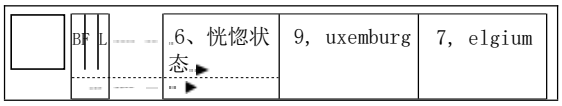


图3.8在间接向量中的穷人的规范化键。

b树等于单个缓存行或少量的[64]。这个b树中的根到叶导航可能使用指针或地址计算[110,87]。与没有为CPU缓存[24]进行优化的节点格式相比，b树页面中的搜索时间和缓存故障可能会被减少一半。一个补充的、更理论的缓存b树格式设计甚至更复杂，但独立于磁盘页面和缓存行[11]的大小实现最佳渐近性能。b树节点的两个组织，i。e.，如图3.8所示的线性数组和b树节点内的b树，可以从幽灵槽中获益。e.，具有有效键值但标记为无效的条目，稍后将稍后讨论。

- 高速缓存故障可能会浪费100秒的CPU周期。b树页面可以被优化以减少缓存故障，就像b树可以被优化（与二叉树相比）以减少页面故障一样。

3.7重复的密钥值

搜索键中的重复值相当常见的。重复的记录不太常见，但在某些数据库中确实会发生，即如果关系和表之间存在混淆，以及如果没有定义主键。对于重复的记录，标准表示是多个副本或带有计数器的单个副本。前一种方法实现起来更简单，因为后一种方法需要在查询操作期间维护计数器。g.，计算和和平均值的乘法，在重复消除时将计数器设为1，在连接中设置两个计数器的乘法。

b树索引中的重复键值是不可取的，因为它们可能导致歧义，例如在从辅助索引到主索引到主索引期间，或在删除与表中特定行相关的b树条目期间。因此，所有b树条目必须唯一。尽管如此，可以利用搜索键的主要字段中的重复值来减少存储空间和搜索工作。

存储非唯一键及其关联信息的最明显的方法是将每个键值与表示这些信息的数组相结合。在非唯一的辅助索引中，记录标识符为

与一个键相关联的信息，这是一种传统的格式。为了有效搜索，例如在删除期间，记录标识符列表保持排序。可以使用一些简单的压缩形式。其中一种方案使用最小的字节数来存储相邻值之间的差异，而不是存储完整的记录标识符。

上面已经讨论了一个类似的方案作为一种替代方案
用于前缀B树。为了实现有效的顺序搜索，可以适应偏移值编码[28]。

该方案的一个更复杂的变体允许显式控制只存储一次的键前缀和存储在数组中的记录剩余部分。例如，如果前键键字段很大，只有很少不同的值，而最终键字段很小，有许多不同的值，那么一次存储这些前键字段的值可以节省存储空间。

非唯一辅助索引的另一种表示使用位图。有各种不同的形式和变体。下面将讨论这些问题。

图3.9中的行显示了相同信息的替代表示： (a)显示了对每个与键的值相关联的不同记录标识符重复重复的键值，这是一个需要最多空间的简单方案。示例(b)显示了具有每个唯一键值的记录标识符列表，而(c)显示了这两种适合于破坏的技术的组合

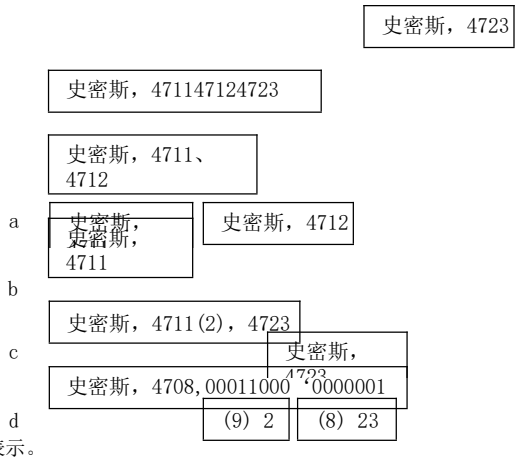


图3.9重复项的替代表示。

名单很长，e。g.，那些跨越多个页面的页面。示例(d)显示了一个基于共享前缀截断的简单压缩。例如，“(9)2”表示这个条目等于前9个字母中的前一个字母，或者“Smith，471”，后面跟着字符串“2”。请注意，这与前缀b树不同，前缀b树从页面或b树节点中的所有记录中截断相同的前缀。示例(e)显示了另一种基于运行长度编码的简单压缩方案。编码“4711(2)”表示一个连续的以4711开头的条系列。示例(f)显示了一个可能在位图索引中使用的位图。前导值4708表示由位图中的第一个位所表示的整数；位图中的“1”位表示值4711、4712和4723。位图本身通常使用运行长度的编码进行压缩。毫无疑问，许多读者可以设计出额外的变体和组合。

这些方案都有自己的优缺点。例如，(d)似乎结合了(a)的简单性和与(b)相当的空间效率，但它可能需要特殊的考虑，无论是使用二进值搜索还是插值搜索。换句话说，似乎没有一个完美的方案。也许原因是压缩技术关注的是顺序访问，而不是压缩数据结构中的随机访问。

这些方案可以扩展到多列b树密钥。例如，第一个字段中的每个不同值可以与第二个字段的值列表配对，并且每个值都有详细信息列表。在一个关于学生和课程的关系数据库中，作为一个具体的例子，一个多对多关系的索引可能对第一个外键有许多不同的值(e。g.，学生标识符)，每个值都有第二个外键的值列表(e。g.，课程编号)，以及关于一对键值之间的关系的附加属性(e。g.，学生参加这个课程的那个学期)。对于信息检索，全文索引可能有许多不同的关键字，每个关键字都有一个包含给定关键字的文档列表，每个文档条目都有一个随文档出现的关键字列表。忽略压缩，这是许多文本索引的基本格式。

重复的键值不仅适用于表示选择，还适用于关系数据库中的完整性约束。b树是

通常用于通过插入一个重复的键值来防止违反唯一的约束。另一种不常用的技术，使用现有的b树索引来即时创建和验证新定义的唯一性约束。在插入新的键值期间，搜索适当的插入位置可以指示与未来邻居键之一的最长共享前缀。所需的逻辑类似于动态前缀截断中的逻辑。基于这些共享前缀的长度，b树索引的元数据可能包含一个不同值的计数器。在多列b树中，可以维护多个计数器。当声明了唯一性约束时，这些计数器会立即指示是否已经违反了候选约束。

- 即使每个b树条目都是唯一的，键也可以被划分为前缀和后缀，这样每个前缀值就有许多后缀值。这使得许多压缩技术成为可能。
- 长列表可能需要分成分段，每个分段都小于一个页面。

3.8位图索引

术语位图索引是常用的，但它是相当模糊的，没有解释索引结构。位图既可以用于b型树，也可以用于哈希索引和其他形式的索引。如图3.9所示，位图是针对一组整数的一种或多种表示技术。只要有一组整数与每个索引键关联，该索引就可以是一个位图索引。然而，在下面，假设了一个非唯一的次要b树索引。

数据库索引中的位图是一个相当古老的想法[65,103]，它随着关系数据仓库的兴起而变得越来越重要。唯一的要求是与索引键和整数相关联的信息之间的一对一映射。e.，位图中的位的位置。例如，由设备号、页号和插槽号组成的记录标识符可以被解释为单个大整数，因此可以用位图和位图索引进行编码。

此外，位图也可以被分割和压缩。对于分割，可能的位位置的域被划分为范围。这些范围被编号，并为每个非空范围创建一个单独的位图。对每个线段重复使用搜索键，并通过范围编号进行扩展。图3.9显示了一个将列表分解为段的示例，尽管使用了引用列表，而不是使用位图。

具有2的线段大小^{15}位位置确保任何段的位图都很容易地适应数据库页面；带有2的段大小^{30}位位置确保了标准整数值可以通过运行长度编码用于压缩。将位图划分为2的段^{15}或^{230}位位置还支持合理有效的更新。例如，插入单个记录只需要对单个位图段进行解压缩和重新压缩，并且空间管理非常类似于更改传统b树记录的长度。

对于位图压缩，大多数方案主要依赖于运行长度编码。例如，WHA[126]将一个位图划分为31位的部分，并用一个计数替换多个相邻的部分。在压缩的图像中，一个32位的字包含一个指示符位加上一个31位的文字位图或一个常量值的运行。在每次运行中，一个30位的计数会留下一个位，以表示被替换的部分是包含“0”位还是“1”位。基于字节而不是单词的位图压缩方案往往可以实现更紧密的压缩，但需要更昂贵的操作[126]。如果运行长度是用可变长度的整数编码的，则尤其如此。

图3.10说明了这种压缩技术。示例(a)显示了一个类似于图3.10中的位图，尽管有不同的第三个值。示例(b)显示了WAH压缩。逗号表示压缩表示中的单词边界，下划线的位值表示单词使用。该位图以151个组开始

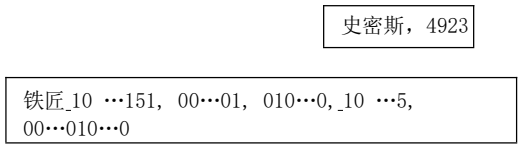
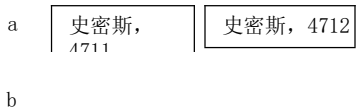


图3．10awah压缩位图。



31“0”位。下面两个字显示文字位图；需要两个位图，因为位位置4711和4712属于不同的31位位置组。另外五组31“0”位，然后向前跳过到位位置4923，这在决赛中显示为一个“1”位
31位的文字组。

在没有压缩的情况下，位图索引只对索引中很少有不同的关键值节省空间。在有效压缩条件下，位图索引的大小与将引用列表分成分段的传统索引的大小大致相同，如图3.10所示。例如，使用WAH压缩，每个引用最多需要运行一次“0”部分加上一个31位的位图。带有记录标识符的传统表示也可能需要每个引用64位。因此，位图索引对于稀疏和密集的位图都是有用的。e.，对于低基数和高基数属性[125,126]。

位图主要用于只读或大读数据，而不是用于更新密集型数据库和索引。这是由于可感知到的更新压缩位图的困难，e。g.，在诸如WAH等运行长度编码方案中插入一个新值。另一方面，使用数字差异压缩的记录标识符列表与运行长度编码中的计数器非常相似。在这两种压缩存储格式中，更新成本应该非常相似。

位图上的主要操作是创建、交集、并集、差值和扫描。位图创建发生在索引创建期间，当位图表示中间查询结果时，在查询执行期间。位图交集有助于连词（“和”）查询谓词，并连连词（“或”）谓词。注意，对整数键的范围查询通常可以转换为分离，e。g.，″...“在3到5之间”相当于“=3或”。.....=4或=5。”因此，即使大多数查询谓词都是写成连词而不是离连词，并集运算对于位图和引用列表也很重要。

对中间查询结果使用位图表示，隐式地对数据进行排序。当使用从辅助索引获得的引用从表中检索不可预测的行数时，这一点特别有用。在位图中收集引用，通常比按排序顺序获取所需的数据库行更多

然后高效地获取行而不进行排序。传统的排序操作可能比位图需要更多的内存和精力。

理论上，位图可以用于任何布尔属性。换句话说，位图中的一点表示某个记录是否具有感兴趣的属性。上面的讨论和图3.9中的示例隐式地假设这个属性与某个键值相等。因此，索引中的每个键值都有一个位图，指示具有这些键值的记录。另一种方案是基于模操作[112]。例如，如果要索引的列是一个32位整数，则有32个位图。位位置k的位图表示关键值模2k非零的记录。查询需要执行交集和联合操作。许多其他方案，如。g.，基于范围谓词，也可以进行设计。奥尼尔等人。[102]调查了许多设计选择。

通常，位图索引表示一对多的关系，e。g.，在键值和引用之间。在这些情况下，在索引中的一个位图中，一个特定的位位置被设置为“1”（假设有一行与该位位置对应的行）。然而，在某些情况下，一个位图索引可能表示一种多对多的关系。在这些情况下，相同的位位置可以在多个位图中设置为“1”。例如，如果一个表包含两个外键以捕获大量的关系，其中一个外键列可能提供辅助索引中的键值，另一个外键列由位图表示。作为一个更具体的例子，学生和课程之间的多种关系注册可以用学生标识符上的b树来表示。一个学生的课程可以在一个位图中被捕获。在许多位图中，代表特定课程的相同位位置被设置为“1”，即在参加该课程的所有学生的位图中。

- 位图需要值和位位置之间的一对一关系。
- 位图和压缩位图只是表示重复（前缀）值的另一种格式。
- 位图可以用于表示与不同前缀值关联的所有后缀值。

- 运行长度编码作为位图的一种压缩技术，类似于通过对列表进行排序和存储邻居之间的差异来压缩一个整数值的列表。基于这种相似性，这两种表示重复值的技术都可以具有类似的空间效率。

3.9数据压缩

数据压缩降低了购买存储设备的费用。它还降低了安装、连接、供电和冷却这些设备的成本。此外，它还可以提高有效的扫描带宽，以及诸如碎片整理、一致性检查、备份和恢复等实用程序的带宽。闪存设备由于其每单位存储空间的高成本，可能会增加人们对文件系统、数据库等数据压缩的兴趣。

b树是数据库中的主要数据结构，证明了专门为b树索引调整的压缩技术是合理的。b树索引中的压缩可以分为关键值的压缩、节点引用的压缩（主要是子指针）和重复项的表示。上面已经讨论过重复的内容；这里调查了另外两个主题。

对于键值，已经提到了前缀和后缀截断，非唯一键值的单个存储也是如此。规范化键的压缩也被提到，尽管这是一个没有发布技术的问题。另一种理想的压缩形式是零和空格的截断，并仔细注意在键[2]中的保持顺序的截断。

其他有序压缩方法在数据库系统中似乎很大程度上被忽略了，例如有序霍夫曼编码或算术编码。保持顺序的字典代码最初得到了关注[127]。它们在排序中的潜在用途，特别是数据库查询处理中的排序，在其他地方进行了调查；其中的许多考虑也适用于b树索引。

对于压缩和解压缩，保持顺序的霍夫曼码都依赖于二叉树。对于静态代码，该树类似于非顺序保持技术的树。建筑的

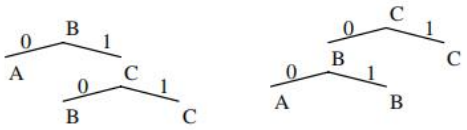


图3．11.自适应保序霍夫曼压缩中的树的旋转。

霍夫曼代码以每个单独的符号形成一个单例集，然后重复合并两组符号集。对于一个标准的霍夫曼代码，频率最低的两个集合被合并。对于一个保持顺序的霍夫曼码，选择了组合频率最低的近邻对。这两种技术都支持静态代码和自适应代码。自适应方法从为静态方法创建的树开始，但根据实际情况修改它，
未压缩流中符号的观察频率。每个这样的修改都会旋转二叉树中的节点。

图3.11，从[46]复制，显示了二叉树中心编码和解码的旋转保持霍夫曼压缩。叶节点表示符号，从根到叶的路径表示编码。左分支由a0编码，右分支由a1编码，符号“a”、“B”和“C”分别有编码“0”、“10”和“11”。树的分支节点包含分隔键，与b型树中的分隔键非常相似。图3.11中的左树是为相对频繁的“A”符号设计的。如果符号“C”特别频繁，则可以将编码树旋转到正确的树中，这样符号“A”、“B”和“C”分别有编码“00”、“01”和“1”。如果叶节点C的累积权重高于叶节点A、i，则从图3.11中的左树向右树的旋转是值得的。e.，如果有效的压缩对叶节点C比对叶节点A更重要。请注意，叶节点B的频率并不相关，其编码的大小不受旋转的影响，并且这种树变换不适合最小化到节点B的路径或B的表示。

b树子指针的压缩可能利用这样一个事实，即相邻节点很可能已经在相邻位置分配，而b树是从未来索引的排序流创建的

的复数形式在这种情况下，可以通过存储指针的绝对值，而是它们的数值差异，并将这些值存储在最少的[131]中。在极端情况下，可以使用一种运行长度编码的形式，它简单地指示起始节点的位置和连续分配的邻居节点的数量。由于b树节点的仔细布局可以提高扫描性能，因此这种b树节点的分配通常是使用适当的空间管理技术来创建和维护的。因此，这种压缩技术经常应用并用于产品。除了b树索引中的子指针外，还可以将一个变量应用于与非唯一辅助索引中的键值相关联的引用列表。

使用数字差异的压缩也是文档检索的主要技术，其中“反向索引”。..记录，对于每个不同的单词或术语，包含术语的文档列表，并根据支持的查询模式，也可能包含每个术语的频率和影响在每个文档，加上每个文档中的位置列表。为了进行有效的压缩，文档和位置号的列表通常会被排序，并转换为相邻值之间相应的差异（或间隙）序列。”[1].研究继续优化压缩效率，i。e.，值和长度指示器所需的位，以及解压缩带宽。例如，Anh和Moffat [1]评估方案，其中单一长度指标适用于在单个机器字中编码的所有差异。更多的想法和技术可以在专门的书籍和调查中找到。g.，[124，129].

- 对于分支节点中的分隔键和子指针，以及叶节点中的键值及其相关信息，存在各种数据压缩方案。
- 标准技术是截断空白空间和零，用它们与基值的差异来表示值，用它们的差异来表示已排序的数字列表。偏移值编码对于合并排序中的排序运行特别有效，但也可以在b树中使用。

•霍夫曼压缩、字典压缩和算术压缩。

3.10空间管理

有时有人说，与堆文件相比，b树具有内置记录的空间管理。另一方面，人们也可以说，即使多个页面有一些空闲空间，在b树中放置记录也没有选择；相反，新记录必须放置在键所在的地方，不能放置在其他地方。

然而，我们也有一些进行良好的空间管理的机会。首先，当插入由于适当节点中的空间不足而失败时，需要在压缩（回收页面内的可用空间）、压缩（重新编码键及其相关信息）、负载平衡（在兄弟节点之间）和拆分之间进行选择。由于简单和局部操作更可取，因此给出的顺序表明了最佳的方法。两个邻居之间的负载均衡很少实现；两个以上的邻居之间很难实现负载平衡
在任何时候但是，某些碎片整理实用程序可能只为特定的键范围调用，而不是针对整个b树。

其次，当需要分割和因此分配页面时，新页面的位置提供了一些优化的机会。如果大范围扫描和索引顺序扫描是频繁的，并且如果b树存储在具有昂贵的搜索操作的磁盘上，那么在现有页面附近分配新页面是很重要的。

第三，在删除过程中，也存在类似的选择。为了避免下流，在删除期间可能需要两个邻居之间的负载平衡，而这是一个可选的插入优化。在b树中删除的“教科书”设计的常用替代方法忽略了下流，在极端情况下，甚至允许b树中的空页面。空间回收将留给将来的插入或碎片整理实用程序。

为了避免或至少延迟节点分割，许多数据库系统允许在索引创建、批量加载和碎片整理过程中在每个页面中留出一些空闲空间。例如，留下10%

所有分支节点中的可用空间几乎不会影响它们的扇形输出或树的高度，但它减少了事务处理过程中节点分割的开销。此外，一些系统允许在磁盘上保留免费页面。例如，如果大扫描中的I/O单元包含多个b树节点，那么在每个这样的单元中保留几个页面可能是有利的。如果一个节点分裂，附近的页面很容易进行分配。在b树中的许多节点由于许多插入而被分割之前，扫描性能不会受到影响。

一种有趣的释放磁盘空间管理的方法依赖于b树的核心逻辑。O‘Neil的sbtree[104]在许多页面的大连续区域中分配磁盘空间，在索引创建和碎片整理期间在每个区域中留下一些空闲页面。当一个节点分裂时，将在同一范围内分配一个新节点。如果因为分配了整个范围，这是不可能的，那么范围将被分成两个范围，每个范围有一半是满的。这种分割非常类似于b型树中的节点分割。这一想法虽然简单而有希望，但尚未被广泛采用。这种“自相似”数据结构和算法的模式可以应用于内存层次结构的多个层次上。

图3.12显示了sb树中的两种节点。扩展区和页面都是节点，因为它们可能会溢出，然后被一分为二。第75.2页中的子指针包含页面标识符的值，因此易于压缩。例如，当第93.4页必须响应于一个插入进行分割时，整个第93页被分割为多个页，e。g.,93.3-93.5，转移到了一个新的程度。

- b型树根据它的排序键严格地放置一个新的记录，但可以优雅地处理空间管理，e。g.，通过邻居节点之间的负载平衡。

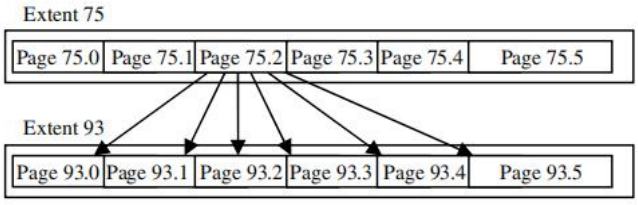


图3．一个sb树中的12个节点。

- b-树的概念不仅适用于页面中的记录的放置，也适用于存储介质上的连续页面集群中的页面的放置。

3.11分体节点

将叶节点分割为两个之后，必须在父节点中发布一个新的分隔键。这可能导致父节点溢出，因此父节点必须被分割为两个，并且必须在祖父节点中张贴一个分隔键。在极端情况下，必须将从叶到根的节点进行拆分，并且必须将一个新的根添加到b-树中。 最初的b树算法要求从叶到根的分裂，如前所述。但是，如果多个线程或事务共享一个B-树，那么自底而上（叶到根）在一个线程中的分割可能与另一个线程的自上而下（根到叶）搜索发生冲突。最早的设计依赖于一个“安全”节点的概念，i。e.，一个有更多插入的空间，并在根到叶搜索[9]期间保留来自最后一个安全节点的锁。一种更极端的方法限制了每个b型树一次只有一次结构变化，[93]。对这个问题使用了其他三个限制较少的解决方案。

首先，由于只有少数插入需要分割操作，因此可以强制这样的插入执行额外的根到叶遍历。第一次遍历确定了需要进行拆分的级别。第二次遍历在适当的级别上执行一个节点分割。如果它不能按需要发布分隔符键，它将停止，而是调用另一个根到叶传递，在下一个更高级别执行拆分。可以优化这个附加的从根到叶的遍历。为考试-

另外，如果上b树节点同时没有被改变，就不需要对已知的结果重复二进制搜索。

其次，插入操作的初始根到叶搜索可以验证所有访问的节点都有足够的空闲空间用于另一个分隔键。对没有足够自由空间的分支节点进行预防性的[99]分割。因此，单个从根到叶的搜索就可以执行所有的插入和节点分割。如果每个节点可以容纳数百个分隔键，那么比真正需要的更提前一点的分割不会对b树空间利用率、节点扇出或树高产生重大影响。

不幸的是，可变长度分隔符键出现了问题；要么拆分决策必须非常保守，要么在罕见的情况下需要第二次根到叶传递，如上一段描述的第一个解决方案。换句话说，在任何情况下都可能需要实现第一个解决方案。如果节点分割很少，添加带有自己的测试用例、回归测试等的启发式代码路径。可能不会提供有价值的、甚至是可衡量的性能增益。

第三，将一个b树节点并在节点的父节点中发布一个新的分隔键分为两个步骤[81]。在中间状态下，这可能会持续很长时间，但理想情况下不会，b-树节点看起来类似于2-3-树中的三元节点，如图2.2所示。换句话说，两个独立的步骤将一个完整的节点一分为二，并在父节点中发布一个分隔符键。在很短的一段时间内，新节点被链接到旧的邻居，而不是它的父节点，从而被称为闪烁树。一旦方便，e。g.，在下一次根到叶遍历期间，分隔符键和指针将从以前溢出的兄弟节点复制到父节点。

- 原始b树结构的一些变体使高并发性和有效的并发性控制成为可能。眨眼-树木似乎特别有前途，尽管它们似乎被产品中忽视了。

.123总结

总之，基本的b树设计，包括数据结构和算法，在几十年的研究和实现努力中，已经在许多方面得到了改进。许多工业实现都采用了迄今为止所回顾的许多技术。忽略甚至反驳这些技术的研究可能被视为与商业数据库管理产品无关。

4

事务性技术

前一节调查了b树数据结构和算法的优化；当前部分主要介绍并发控制和恢复技术。对实际系统的开发和测试工作的大部分时间都花在了磁盘上数据结构的并发性控制和恢复上，即主要是b型树。事务支持、查询处理和全套实用程序，i。e.，本节和以下部分的主题，将传统的数据库管理系统与现在在各种web服务及其实现[21,29]中使用的关键值存储区分开来

本节中隐含的一点是，b树结构不仅可以支持只读搜索，而且还可以同时更新现有记录，包括插入、删除和修改现有记录，包括键和非键字段。这里的重点是即时更新，而不是使用差异文件[117]等技术进行延迟更新。后面的部分将介绍在成熟的数据库管理系统中用于维护多个相关索引、实例化视图、完整性约束等的更新计划。

由于建立数据库的复杂性和费用通常只能通过在许多用户之间共享数据来证明，

许多应用程序，等等，通过并发事务和执行线程访问数据库，从一开始就处于数据库研究和开发的前沿，以及从软件或硬件故障中获得高可用性和快速、可靠的恢复。最近，多核处理器增加了对内存中数据结构的高度并发性的关注。事务性内存可能是解决方案的一部分，但需要理解适当的事务边界，因此需要选择一致的中间状态。

除了并发用户之外，还有一种使用异步、并行、联机和增量实用程序的趋势。这些操作可在永久存储器上执行可选的或强制性的任务。一个典型的强制性异步任务是巩固空闲空间，e。g.，在将表或b树从数据库中删除索引后，只需将其标记为过时，而无需将其页面添加到空闲空间中。一个典型的可选的异步任务是碎片整理，e。g.，b树叶之间的负载平衡和磁盘上布局的优化，以实现高效的范围查询和索引顺序扫描。还有许多其他的异步任务并不特别涉及到b树。g.，收集或更新统计信息，以便用于编译时查询优化。

图4.1列出了交易的四个“ACID”属性以及简要的说明。这些属性将在任何数据库教科书中进行更详细的讨论。在解释原子性时的“逻辑”一词值得通过一个具体的例子来进一步澄清。考虑一个用户事务，它试图将新行插入数据库表，分割b树节点以创建足够的可用空间，但随后失败。在事务回滚期间，必须撤消行插入，但回滚节点拆分并不严格要求，以确保正确的数据库内容。如果节点分割的影响仍在数据库中，则之后没有逻辑数据库更改

原子性一致性隔离的持久性	全部或全部：完全成功或没有（逻辑）数据库更改 一致的数据库状态将转换为新的一致的数据库状态事务输出和数据库更改，就像没有其他事务在活动中一样。一旦提交，数据库更改就会“通过火灾、洪水或暴动”持续存在

图4．1.事务处理的ACID属性。

即使有物理更改，事务也会回滚。“逻辑”在这里可能由“查询结果”定义，“物理”由数据库表示定义，如磁盘上的位。

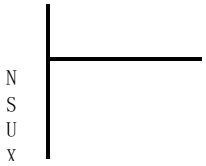
逻辑数据库和物理数据库，或数据库内容与数据库表示之间的区别，将渗透到下面的讨论中。一个特别有用的实现技术是“系统事务”的概念。e.，在数据库表示法中修改、记录和提交更改，但对数据库内容没有影响的事务。系统事务对于b型树中的节点分割、空间分配和整合等非常有用。在上面的示例中，用户事务调用执行、日志并提交节点拆分的系统事务；当用户事务回滚时，提交的节点分割仍然不变。系统事务通常非常简单，并且在单个线程中运行，通常是用户事务的线程，这样用户事务就会等待系统事务完成。如果用户事务在多个线程中运行，则每个线程都可以调用其自己的系统事务。

如果对大表和索引进行了分区，并将分区分配给分布式系统中的节点，那么通常的实现允许每个节点在需要时执行本地并发控制和恢复。如果单个站点使用多个恢复日志，则需要使用类似的技术。分布式事务、两阶段提交等。超出了本次b树索引调查的范围。

锁是并发性控制的通常机制。图4.2显示了一个基本的锁兼容性矩阵，它没有锁(N)、共享(S)、排他(X)和更新(U)模式。左列表示当前持有的锁，最上面的行表示请求的锁。矩阵中的空空格表示所请求的锁不能为

锁上拿	请求的锁定			
	S		U	X
S		U	X	
S	U			
U?				

图4.2基本的锁的兼容性矩阵。



假定如果当前没有任何活动的锁定，则可以授予任何锁定。两个共享锁是兼容的，这当然是共享的本质，而独占锁与任何其他锁都不兼容。共享锁也被称为读锁，独占锁也被称为写锁。

对于允许的锁定请求，该矩阵表示聚合锁定模式。它的目的是加快对新的锁请求的处理速度。即使许多事务对特定资源保持锁定，新的锁定请求也必须只针对聚合锁模式进行测试。不需要验证新的锁请求与已授予的每个锁的兼容性。.2换句话说，图4左列中现有的锁定模式是现有的聚合锁定模式。在图4.2中，锁定模式的聚合在大多数情况下都是微不足道的。后面具有附加锁定模式的示例包括新聚合锁模式不同于旧聚合锁模式和请求锁模式的情况。图4.2中没有反映出的一种特殊情况是将锁从更新模式降级到共享模式。由于只有一个事务可以保持更新锁，因此从更新模式降级到共享模式后的聚合锁定模式是共享锁。

更新锁是为在更新数据记录之前首先测试谓词的应用程序而设计的。从一开始就使用独占锁可以阻止其他事务处理相同的逻辑；仅仅使用一个共享锁允许两个锁在同一数据项上获得共享锁，但如果两者都试图升级到独占锁，就进入死锁。更新锁定一次只允许一个事务处于不确定的状态。在谓词计算之后，更新锁确实升级为独占锁，或者降级为共享锁。请注意，更新锁不授予共享锁的权利；它们的区别在于调度和死锁预防，而不是并发控制或允许的数据访问。因此，降级到共享锁是允许的。

更新锁也被称为升级锁。鉴于更新锁优先考虑升级锁，而不是更新数据项的权限，升级是一个更准确的名称。然而，更新锁似乎已经变得更常用了。Korth [79]深入探讨了派生锁等升级锁与共享锁和独占锁等基本锁之间的关系。

图4.2中的一个字段显示了一个问号。有些系统允许新的共享锁，而一个事务已经持有更新锁，有些则不允许。只有当事务请求独占锁时，前一组才会停止其他共享锁。最有可能的是，这是持有更新锁的事务，但不一定是。因此，后一种设计在防止死锁[59]方面更有效，即使它在锁矩阵中引入了不对称性。随后的锁矩阵的例子假设采用非对称设计。

提供故障原子性和持久性的主要方法是预写日志记录，这要求恢复日志在数据库发生任何就地更新之前描述更改。每种类型的更新都需要在初始处理过程中调用“do”方法，“重做”方法以确保数据库在失败或崩溃后仍反映更新，以及“撤销”方法以使数据库恢复到之前的状态。“do”方法还创建具有足够信息的日志记录，用于“重做”和“撤销”调用，并指示缓冲池保留脏数据页，直到这些日志记录安全到达“稳定存储”上。恢复是可靠的，因为稳定的存储是。镜像日志设备是一种常见的技术。日志页面一旦被写入，就不得被修改或覆盖。

在早期恢复技术中，“重做”和“撤销”动作必须是幂等的[56]，i。e.，重复应用相同的操作会导致与单个应用程序相同的状态。一个基本的假设是，恢复过程使恢复日志保持在只读模式，i。e.，从故障恢复期间没有日志记录。后来的技术，特别是ARIES [95]，通过记录“撤销”操作并在每个数据页中保留“page LSN”（日志序列号），可靠地应用“重做”和“撤销”操作，这表示最近应用的日志记录。此外，它们“补偿”更新，而不是物理上的更新。例如，删除补偿了一个插入，但是在b树索引中的叶子分裂后，删除可能发生在与插入不同的叶页中。中止一笔交易适用于补偿
正常更新然后提交，除了不需要立即强制提交记录到稳定存储。

- ACID属性的原子性、一致性、隔离性和持久性定义了事务。提前写入日志记录和

“做-再做-撤销”三重奏是恢复和可靠性的基石。锁存和锁定是并发控制的基石。

- B型树中的记录级锁定是键值锁定和键范围锁定。锁定的粒度(e。g.，记录，键)小于恢复的粒度(e。g.，页面)需要日志记录“撤销”操作和逻辑补偿，而不是调用未记录的幂等操作的严格物理恢复。
- 物理数据独立性将逻辑数据库内容及其物理表示分开。在数据库系统的关系层中，它创造了物理数据库设计的自由度，并强制需要自动查询优化。在数据库系统的存储层中，它在并发控制和恢复的实现中实现许多优化。
- 一个重要的优化方法是将查询或修改逻辑数据库内容的用户事务和只影响内容的物理表示的系统事务分开。说明系统事务的优点的典型示例是在b-树中分割一个节点。

.14锁定和锁定

1 B-树锁定，或锁定在b树索引中，意味着两件事。首先，它意味着在查询或修改数据库内容的并发数据库事务之间的并发控制。在此上下文中，主要关注的是逻辑数据库内容，它独立于它在b树索引等数据结构中的表示。其次，它意味着并发线程之间的并发控制，可以修改内存中的数据结构，包括缓冲池中基于磁盘的b树节点的特定图像。

这两个方面并不总是被清楚地分开。当单个数据库请求为时，它们之间的差异就会变得非常明显

[1] 本部分的大部分内容都是从[51]中复制过来的。

由多个并行线程处理。具体来说，同一事务中的两个线程必须“查看”相同的数据库内容、表中相同的行数，等等。这包括一个线程，“看到”由另一个线程代表同一个事务应用的更新。然而，当一个线程分割一个b树节点时，i。e.，修改特定数据结构中的数据库内容的表示，其他线程不能观察中间和不完整的数据结构。当单个执行线程服务于多个事务时，这种差异也会变得很明显。

这两个目的通常通过两种不同的机制来实现，锁和闩锁。不幸的是，关于操作系统和编程环境的文献通常使用术语锁来表示数据库系统中的机制称为锁，这可能会令人困惑。

图4.3总结了它们之间的差异。使用页面、B树键甚至键之间的间隔（打开间隔）的读写锁锁定单独的事务。后两种方法分别称为键值锁定和键范围锁定。键范围锁定是谓词锁定的一种形式，它使用B树中的实际键值和B树的排序顺序来定义谓词。默认情况下，锁定参与

	锁	门闩
单独。。。	用户事务处理	线程
保护。。。	数据库内容	内存中的数据结构是关键的部分
在……期间	整个事务处理[2]	
模式。。。	共享、独家、更新、意图、托管、模式等。	阅读，写作，（也许）更新
僵局	检测与分辨率	避免
通过	分析等待图、超时、事务中止、部分回滚、锁定降级	编码规程，即时超时请求，“锁定升级”[3]
保持状态。	锁定管理器的散列表	受保护的数据结构

图4.3锁和锁。

[2] 事务必须保留对事务提交的锁，以便与串行执行相等，也称为事务隔离级别“可序列化”。较弱的交易隔离允许更短的锁定时间。在许多数据库系统中，弱事务隔离是默认的，因此以正确和完全隔离并发事务为代价实现更高的并发性。

[3] 在这种技术中，会为任何锁存器分配一个级别。线程可能只请求比已经持有的最高锁存器级别更高的锁存器。

在死锁检测中，并一直保存到事务结束。锁还支持复杂的调度，e。g.，使用队列来进行挂起的锁请求和延迟新的锁获取，以支持锁转换，e。g.，对独占锁的现有共享锁。这种复杂的级别使得锁的获取和释放相当昂贵，通常是数百个CPU指令和数千个CPU周期，其中一些是由于锁管理器的散列表中的缓存故障造成的。

锁存访问b树页面的单独线程、缓冲区池的管理表，以及在多个线程之间共享的所有其他内存中的数据结构。由于锁管理器的哈希表是许多线程共享的数据结构之一，因此在检查或修改数据库系统的锁信息时需要闭锁器。对于共享的数据结构，如果一个线程需要写锁存器，即使是同一用户事务的线程也会发生冲突。锁存只在关键部分，i。e.，同时读取或更新数据结构。通过适当的编码规程，e，可以避免死锁。g.，要求按照精心设计的序列设置多个锁存器。死锁解决方案需要允许回滚之前的操作，而死锁避免则不会。因此，避免死锁更适合于锁存，其设计为最小的开销、最大的性能和可伸缩性。锁存器的获取和发布可能只需要几十条指令，通常没有额外的缓存故障，因为锁存器可以嵌入到它所保护的数据结构中。对于缓冲区池中的磁盘页面的图像，锁存器可以嵌入到描述符结构中，其中还包含页面标识符等。

由于锁与数据库内容有关，而不是它们的表示关系，因此包含叶节点中所有内容的b树不需要对b树的非叶级别进行锁。另一方面，所有页面都需要锁存器，这与它们在数据库中的角色无关。锁定和锁定之间的差异在次要索引的并发控制中也变得很明显。e.，指向非冗余存储结构的冗余索引。例如，在ARIES/IM [97]的仅数据锁定中，单个锁定覆盖了与逻辑行相关的所有记录和b树条目。辅助索引及其键不用于以更细的粒度分离事务，并允许更多的并发性。另一方面，锁闩是为了

多个并发线程触及的任何内存内数据结构，当然包括辅助索引的页面和节点。

- 锁定坐标线程以保护内存中的数据结构，包括缓冲池中的页面图像。锁定坐标事务以保护数据库内容。
- 死锁检测和解决通常用于事务和锁，但不用于线程和锁。闩锁的死锁避免需要编码规则和闩锁获取请求的失败，而不是等待。
- 锁存与临界部分密切相关，可以由硬件e来支持。g.，硬件事务性内存。

4.2幽灵记录

如果事务删除了b树中的记录，则它必须保留回滚的能力，直到事务提交为止。因此，事务必须确保在回滚过程中空间分配不会失败，并且另一个事务不能插入具有相同唯一b树键的新记录。满足这两个要求的一个简单技术是在b树中保留记录及其键，仅仅标记它无效。
记录及其密钥保持锁定，直到删除事务提交。另一个好处是，用户事务还延迟甚至避免了空间管理的一些工作。g.，在页面的间接数组中移动条目。此外，用户事务只需要锁定被删除的记录，而不是锁定记录的两个直接相邻键之间的整个密钥范围。

生成的记录被称为伪删除记录或幽灵记录。记录头中的单个位就足以指示记录的幽灵状态。因此，一个删除就变成了对幽灵位的修改。如果并发控制依赖于键范围锁定（下面讨论），则只需要锁定键本身，键之间的所有间隙可能保持解锁。

图4.4说明了一个带有幽灵记录的b树页面。e.，删除键为27的记录后立即出现的中间状态。显然，这是在幽灵清除和空间回收之前

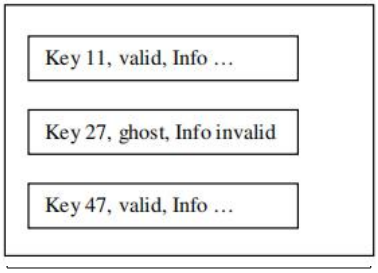


图4.4带有幽灵记录的b形树页面。

页面。有效的记录包含一些与键相关的信息，用椭圆表示，而鬼记录中的信息字段可能被保留，但没有意义。实现空间回收的第一步可以尽可能地缩短这些字段，尽管完全删除幽灵记录可能是一种选择的方法。

查询必须忽略（跳过）幽灵记录；因此，在带有幽灵记录的系统中的扫描总是有一个隐藏的谓词，尽管这个谓词的计算被编译到b树代码中，而不需要任何谓词解释器。空间回收留给后续事务，这可能是需要比页面中更多可用空间的插入、显式调用的页面压缩或b树碎片整理实用程序。

在锁定幽灵记录时，无法删除它。换句话说，至少在事务提交将有效记录转换为鬼记录之前，鬼记录仍然存在。随后，另一个事务可能会锁定一个幽灵记录，e。g.，以确保持续缺少关键值。锁定缺失对于可串行性至关重要；没有它，重复的“选择计数”(*)”在同一事务中的查询可能会返回不同的结果。

同一页面中可能同时存在多个幽灵记录，单个系统事务可能删除所有记录。将幽灵删除的日志记录与事务提交的日志记录合并，就不需要记录已删除记录的内容。合并这些日志记录会使事务不可能在幽灵删除和提交之间失败。因此，从来没有

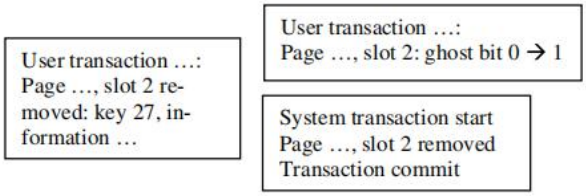


图4.5删除记录的日志记录。

可能需要回滚鬼魂删除，从而对恢复日志中的记录内容。换句话说，ghost记录不仅可以确保成功的事务回滚，而且通常会减少与删除相关的整个日志卷。

图4.5说明了无鬼记录和有鬼记录时删除记录的日志记录。在左侧，用户事务将删除该记录并记录其全部内容。如果需要，可以使用恢复日志中的信息重新插入该记录。在右边，用户事务仅仅是修改了幽灵位。在以后的某个时候，系统事务会创建一个日志记录，包括事务启动、幽灵删除和事务提交。没有办法从恢复日志中重新插入已删除的幽灵记录，但是没有必要这样做，因为删除已在记录时被提交。

如果在b树中使用与ghost记录相同的键插入新行，则可以重用旧记录。因此，插入可能会变成对鬼位的修改，在大多数情况下，还可能会变成记录中的一些其他非键字段。在删除期间，键范围锁定只需要锁定键值，而无需锁定插入新键的键范围。

虽然鬼影记录通常与b型树中的记录删除相关联，但它们也可以帮助插入新的键。拆分插入
分为两个步骤，可以减少事务所需的锁。首先，在锁存器的保护下，使用所需的键创建一个幽灵记录。此步骤不需要使用锁。其次，用户事务锁定和修改新记录。如果用户事务失败并回滚，则幽灵记录将保留。第二步需要锁定键值，但不需要插入新键的键范围。

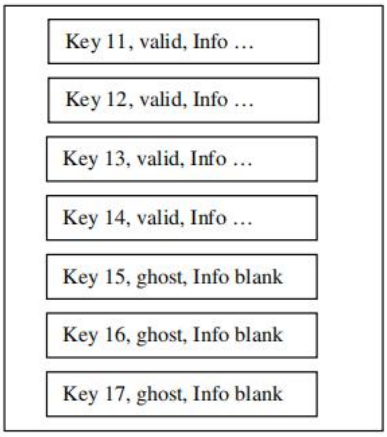


图4.6插入多个幽灵记录。

这个想法的另一个改进是创建多个可能带有未来键的幽灵记录。如果未来的密钥插入完全可预测，如订单号和发票号，这一点特别有用。即使未来键的精确值是不可预测的，这种幽灵记录可能有助于分离未来插入，从而在未来插入事务之间实现更多的并发性。例如，一个键可能由多个字段组成，但值仅对前导字段可预测，例如，每个顺序内的顺序号和行号。

图4.6说明了插入多个幽灵记录。在插入了键值11、12、13和14的有效记录之后，下一个操作很可能是插入了键值15、16、17等的记录。这些插入的性能可以通过预先分配已经填充这些键的适当的空间来提高这些插入的性能。用户事务节省了分配工作，并且只锁定了键值，既不是键之间的间隙，也不是现有最高键值与无穷大之间的间隙，这通常是具有这种插入的事务序列的瓶颈。

最后，在有效的b树记录中插入只包含键，而没有任何剩余的字段。将这些“幽灵槽”散布到有序的记录序列（或间接数组中的槽）中，可以实现有效的插入

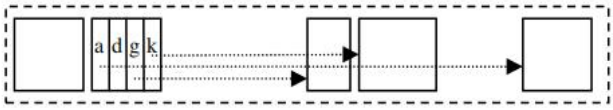


图4.7一个带有一个可供快速插入的鬼影插槽的b形树节点。

进入一个页面。在没有这种幽灵插槽的页面中，插入需要移动所有条目的一半，例如间接数组中的插槽。在带有ghost插槽的页面中，插入的复杂性不是O (N)，而是O（log N）[12]。例如，在一个二级索引每页成千上万的小记录，一个插入需要改变也许十而不是成千上万的条目间接向量，删除转移没有，只是留下一个鬼槽，和一个页面重组离开也许10%或20%的槽鬼槽。

图4.7是对图3.3的改进，显示了两个差异。

首先，间接向量中的条目包含键或实际上的键前缀。图中显示了字母，但实际的实现将使用穷人的规范化键，并将它们解释为整数值。第二，其中一个插槽是一个幽灵插槽，因为它包含一个键（“d”），但没有引用一个记录。这个插槽可以参与二进制搜索和键范围锁定。它可能是在页面重组期间被放在那里的，或者，同样可能是快速记录删除的结果，没有移动两个键“g”和“k”的槽。一旦它存在，它可以加快插入。例如，插入一个带有键“e”的新记录可以简单地修改当前包含“d”的插槽。当然，这要求键“d”当前没有被锁定，或者锁管理器允许进行适当的调整。

- 幽灵记录（也称为伪删除记录）通常用于减少删除期间的锁定要求，并简化删除的“撤消”。
- 幽灵记录对查询结果没有贡献，但参与了密钥范围锁定。
- 可以在插入或异步清理期间回收，但仅在未锁定的情况下。
- 幽灵记录也可以加速和简化插入。

4.3关键范围锁定

[4]术语键值锁定和键范围锁定通常可以互换使用。锁定键范围而不是只锁定键值的目的是保护一个事务不被另一个事务插入。例如，如果一个事务执行一个类型为“选择计数(*从…..在哪里..在…之间..和..,″ i.e.,具有索引列的范围谓词的查询，如果该查询在可序列化的事务隔离中运行，那么第二次执行相同的查询应该产生相同的计数。换句话说，除了保护查询范围内的现有b树条目不被删除之外，该事务获得和持有的锁还必须保护现有键值之间的差距，以防止插入具有新键值的新b树条目。换句话说，键范围锁定通过锁定现有键值之间的间隙来确保键值的持续缺失。弱于可序列化性的事务隔离级别并不能提供这种保证，但是许多应用程序开发人员无法掌握它们的精确语义及其对应用程序正确性的有害影响。

密钥范围锁定是谓词锁定[31]的一种特殊形式。在主要产品中既没有采用通用的谓词锁，也没有采用更实用的精确锁定[76]。在键范围锁定中，谓词由b树的排序顺序中的间隔来定义。间隔边界是b树中当前存在的关键值。通常的形式是半开间隔，包括两个相邻键和其中一个端点之间的间隙，“下一个键锁定”可能比“前一个键锁定”更常见。下一个键锁定需要具有锁定人工键值“+∞”的能力。优先键锁定可以通过锁定NULL值来实现，假设这是b-树的排序顺序中可能的最低值。

在最简单的钥匙范围锁定形式中，一个钥匙和与邻居的间隙被锁定为一个单个单元。任何形式的b树条目、其键或与其邻居的间隙的更新都需要一个独占锁，包括修改记录的非键字段，删除键，

[4] 本节的大部分内容都是从[51]复制过来的，它既不包括更新锁，也不包括聚合锁模式的概念。

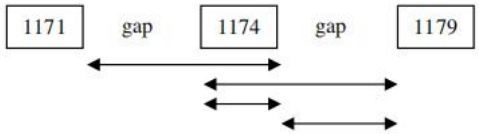


图4.8可能的锁定范围。

在间隙中插入一把新的钥匙，等等。删除钥匙需要锁定旧钥匙和它的邻居；后者需要确保在事务回滚时能够重新插入钥匙。

图4.8说明了单个键值上的键范围锁定可能保护的替代方案，使用包含三个键值在1170到1180之间的记录的叶子b。键值1174的锁定可能具有箭头指示的任何范围。第一个箭头说明了传统的下一个键锁定，i。e.，锁锁定了两个键值与以下记录及其键值之间的间隙。第二个箭头表示前键锁定。第三个箭头显示了一个仅限于键值的锁，而不覆盖相邻的任何一个间隙。因此，这个锁不能保证在事务的持续时间内没有钥匙。g.，键值为1176，因此不能保证可串行性。

第四个箭头显示一个锁，稍后将简要讨论，它补充了键值锁；它可以保证没有密钥，而不锁定现有的密钥。当一个交易按第四箭头所示锁定键值1174时，第二个交易可以用键值1174更新记录。更具体地说，第二个事务可以修改记录中的非键字段，但不能修改记录的键值。因此，在第一个事务释放其锁之前，第二个事务不能删除记录或密钥。另一方面，第二个事务可能会更新记录的幽灵位。例如，如果它发现该键值为1174的记录是一个有效的记录，那么它可以将其转换为一个幽灵记录，从而从未来的查询结果中排除该键值。相反，它可以将鬼记录转换为有效的记录，并更新记录中的所有非键字段，从而在b树中应用逻辑插入。图4.8还可以显示第五个锁定范围，它覆盖了锁定键之前的间隙；这个箭头被省略了，因为它可能会混淆下面的讨论。

键距锁定是商业系统中常用的锁定方法。aris/KVL（“键值锁定”）[93]和aris/IM(“索引管理”）[97]都是键范围锁定的形式。这两种技术都不会锁定单个索引中的各个条目。ARIES/KVL在各个索引中锁定唯一的键值。在非唯一的辅助索引中，单个锁覆盖了具有相同键值的整个记录列表，再加上对先前唯一键值的打开间隔。在可序列化的事务隔离中插入这样的开放间隔需要这样的锁，即使只有即时持续时间。如果一个并发事务持有一个相互冲突的锁，则e。g.，由于对列表中的一条记录的读取访问，插入失败或延迟。读取具有一个键值的记录和插入具有不同键值的记录之间没有实际冲突；设计中锁范围的选择只会使它看起来好像存在。也许正是由于这种人为的冲突，许多数据库安装运行的事务隔离级别低于可序列化性，而且许多供应商的软件都具有默认的较弱的隔离级别。

ARIES/IM中的锁定覆盖了表中的一行，包括它的所有索引项，以及每个索引中之前的打开间隔（“仅数据锁定”，稍后在DB2中“type-1索引”）。在非唯一索引中，此打开间隔可能被具有相同索引键但具有不同记录标识符的另一个条目所限定。在ARIES/IM页锁定中，锁定覆盖了数据页中的所有行、它们的索引项以及适当索引中的键之间的适当打开间隔。“结构修改操作”获得一个特定于索引树的X锁存器，只读操作不会获得，即使在S模式下也不会，除非这样的操作遇到带有“结构修改位”集的页面。由于这两种方法都相当复杂，因此鼓励读者阅读原始论文，而不是依赖次要来源，如这个调查。希望首先阅读这份调查能让你更少地阅读原始的论文。

微软的SQL Server产品采用了基于Lomet设计的[85]的密钥范围锁定，它建立在绵羊/IM和绵羊/KVL[93,97]之上。和白羊座一样，这种设计也需要“即时锁”。e.，持续时间极短的锁。此外，它还需要“插入锁”，

i.e., 一种新的锁定模式，仅适用于b树索引中的两个键值之间的打开间隔。然而，在为SQL Server发布的锁矩阵中，插入锁与排他锁非常相似，以至于不清楚为什么这种区分是必需的或有用的。最近的设计既不需要即时锁，也不需要插入锁，但它允许比Lomet的[48]设计有更多的并发性。

为了启用各种锁定范围，键范围锁定是基于分层锁定[58]。在锁定一个或多个小粒度锁定的项目之前，首先需要对大粒度的锁定进行适当的意图锁定。一个典型的用例是搜索锁定在S模式下的文件，并在X模式下更新几个记录，这除了在锁定X模式之外，还需要使用IX模式（意图获得独占锁）。冲突在文件级别检测，特别是通过之间的冲突
S和IX锁。

图4.9显示了分层锁定的锁兼容性矩阵。标记为“a”的组合表示锁由于更新锁模式的不对称而不兼容。这个矩阵通过添加聚合锁模式而超越了传统的锁兼容性矩阵。例如，如果一个资源已经被多个事务锁定在IS模式下（而没有其他模式），则聚合锁定模式也是IS。IX锁的请求可以基于聚合锁模式授予，而不检查先前事务持有的单个锁，新的聚合锁模式变成IX。

在图4.9中很容易看到，两个意图锁总是相互兼容的，因为任何实际冲突都将在较小的锁定粒度下受到检测。否则，意图锁和绝对锁就像传统的绝对锁一样完全兼容

	是	*ix*	*S*	*U X*	*六*
是 ixs ux6 吗	是 ix S a 六	ix ix	S S a	U U	六

图4.9分层锁定中的锁定兼容性。

头发组合模式S + IX与那些与S模式和IX模式兼容的锁模式兼容。

图4.9显示了更新(U)锁，但不是有意更新（IU和SIU）锁，遵循Gray和Reuter [59]。应获取意图写（IX和SIX）锁。一个大粒度锁定的IX锁覆盖了一个小粒度锁定的U型锁。

在基于分层锁定的键范围锁定中，锁定的大粒度是半开间隔；锁定的小粒度要么是键值，要么是开间隔。这个简单的层次结构允许非常精确的适合每个事务的需求的锁。这种设计的缺点是锁定键（或打开间隔）需要两次调用锁定管理器，一个用于半打开间隔的意图锁定，一个用于键值的绝对锁定。

假设所有三个锁（键值、打开间隔以及它们在半打开间隔内的组合）都是由键值标识的，那么可以在锁定模式的数量和锁定管理器调用的数量之间进行权衡。另外，人工锁定模式可以描述半开间隔、键值和打开间隔上的锁定组合。因此，对于半开间隔、密钥值和打开间隔使用分层锁定的系统不需要比只锁定半开间隔的系统更多的锁定管理工作。在不需要额外的运行时工作的情况下，这样的系统允许在分别锁定密钥值和打开间隔的事务之间进行额外的并发性。g.，以确保在打开的时间间隔中没有键值，并更新记录的非键属性。一个记录的非键属性包括该属性，无论该记录是有效记录还是幽灵记录；因此，当另一个事务锁定相邻的打开间隔时，甚至逻辑插入和删除也是可能的。

具体来说，半开间隔可以锁定在S、X、IS、IX模式下。不需要六个模式，因为有了精确的两个资源，更精确的锁定模式很容易实现。键值和打开间隔都可以锁定在S或X模式下。新的锁定模式必须精确地涵盖两种资源的S、X或N（无锁定）模式的所有可能组合，即键值和打开间隔。意图锁定IS和IX可以保持隐含。例如，如果

键值锁定在X模式，半开间隔隐含锁定在IX模式；如果键值锁定在S模式，开间隔锁定在X模式，则包含两者的半开间隔上的隐含锁定为IX模式。锁可以很容易地使用两种锁模式来识别，一种用于键值，另一种用于打开间隔。假设键前锁定，SN锁定在S模式下保护键值，并解锁以下打开间隔。NS锁定会使钥匙解锁，但会锁定打开的时间间隔。这种锁定模式可用于真正的幻影保护。

图4.10显示了锁的兼容性矩阵。它可以简单地通过检查第一个组件和第二个组件的兼容性来导出。例如，XS与NS兼容，因为X与N兼容，而S与S兼容。单字母锁相当于使用同一个字母两次，但引入超过绝对必要的锁模式并没有任何好处。

图4.10包括了新的聚合锁模式不同于先前的聚合锁模式和请求的锁模式的示例。SN和NS结合到S，但更有趣的是，SN和NX不仅兼容，而且结合到在方案中完全导出和解释的锁模式，而不需要特定于聚合锁模式的新的锁模式。

	ns	*nu*	*nx*	*sn*	*S*	*苏*	*sx*	*联合国*	*美国*	*U*	*ux*	*xn*	*xs*	*XU X*
ns nu nx sn S 苏 sx 联合国 美国 U ux xn xs 徐 X	ns a S S a 美国 美国 a xs xs a	nu 苏 U 徐	 sx ux X	S 苏 sx sn S 苏 sx a a a a	S S a a a a	苏 苏 苏 a	 sx a	美国 U ux 美国 美国 U ux	美国 美国 美国 美国	U U U	 ux	xs 徐 X	xs	徐

图4．10具有组合锁定模式的锁定表。

如果辅助索引中的条目不是唯一的，则可能会有多个行标识符与搜索键的每个值相关联。由于单个频繁的键值或很少有不同值的属性，每个键值甚至可能有数千个记录标识符。在非唯一索引中，密钥值锁定可以锁定每个值（及其整个行标识符集群），也可以锁定每个唯一的值对和行标识符。前者在搜索查询中保存锁请求，而后者可能在更新期间允许更高的并发性。对于前一种设计中的高并发性，意图锁可以应用于值。根据设计的详细信息，如果各个行标识符已经锁定在辅助索引所属的表中，则可能不需要锁定这些行标识符。

除了传统的读写锁、共享锁和独占锁外，还研究了其他锁模式。最值得注意的是“增量”锁。增量锁使并发事务能够增加和减少和和计数。这在详细表中很少需要，但可能是摘要视图中的并发瓶颈。在为这类实体化视图定义的B树索引中，鬼记录、键范围锁定和增量锁的组合，即使影响摘要记录和索引项的存在，也具有高并发性。键范围锁定可以很容易地扩展到包括增量锁，包括在b树中现有键值之间的间隔上的增量锁。更多的细节可以在其他地方找到[51,55,79] O‘Neil 1987。

高插入率可以在与时间相关的属性上的b树索引的“右边”处创建一个热点。通过下一个键锁定，一个解决方案验证在+∞（无限）上获得锁定的能力，但实际上并没有保留它。这种“即时锁”违反了两阶段锁定，但如果一次获取页面锁存器可以保护锁的验证和创建页面上的新密钥，那么它就可以正常工作。另一种解决方案依赖于系统事务来插入幽灵记录，让用户事务将它们转换为有效的记录而不相互干扰。系统事务不需要任何锁，因为它不修改逻辑数据库内容，而且后续的用户事务只需要对受影响的b树条目进行键值锁。如果未来b树条目的关键值为

可预测的。g.，订单号，单个系统事务可以插入多个幽灵记录，从而为多个用户事务做好准备。

- 键范围锁定会锁定键值和键值之间的间隙。它是一种特殊而实用的谓词锁定形式。设计的简单性在于它们和支持的并发性。
- 锁定现有键值之间的间隙，i。e.，锁定没有新密钥，是可序列化，i。e.，等同于串行执行的并发事务的真正隔离。

4.4叶片边界的关键范围

在传统的密钥范围锁定中，复杂性和效率低下的另一个来源是跨越相邻叶节点之间边界的范围锁定。例如，为了插入一个新的b树条目，其键高于叶中的所有现有键值，下一个键锁定需要在下一个叶中找到最低的键值。在叶中插入新的低压键值时，优先键锁定也存在同样的问题。为了高效地访问下一个叶，许多系统在每个b树节点中都包含一个下一个邻居指针，至少在叶节点中是这样的。另一种解决方案避免了邻居指针，而是在每个b树节点中使用两个栅栏键。它们定义了将来可能被插入到该节点中的键的范围。其中一个栅栏是包含边界，另一个是排他边界，这取决于当父节点中的分隔符键精确等于搜索键时的决定。

在初始的空b树中，有一个节点同时是根和叶，负无穷和正无穷用特殊的栅栏值表示。所有其他栅栏键值都是在分割叶节点时建立的分隔键的精确副本。当b树节点（叶或分支节点）溢出并被拆分时，将在父节点中安装的键也会保留在拆分后的两个页面中。

一个栅栏可能是一个有效的b树记录，但它不一定是。具体来说，作为包含边界的栅栏键可以是有效的

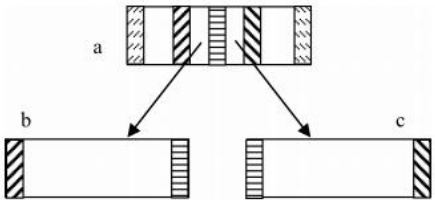


图4．11棵b形树与栅栏钥匙。

有时是数据记录，但另一个栅栏键（专属绑定）总是无效（幽灵记录）。如果删除了作为栅栏的有效记录，则其键必须作为幽灵记录保留在该页页中。事实上，幽灵记录是围栏的首选实现技术，但与传统的幽灵记录不同，围栏不能通过插入需要在叶子中使用空闲空间的记录或清理工具来删除。然而，当插入一个新的b树条目和一个与栅栏键精确相同的键时，作为包含栅栏的幽灵记录可以再次转换为一个有效的记录。

图4.11显示了一个在叶树节点和非叶节点（根）中都有栅栏键的B树。由于栅栏键定义了页面中可能的键范围，因此永远不需要锁定相邻叶子中的键值。当一个栅栏从一个幽灵记录变成一个有效的记录时，我。e.，在插入一个键值精确等于栅栏键的新b树条目时，不需要锁定键的范围。只能锁定键值，因为插入是通过修改现有的b树条目而不是创建一个新的条目来执行的。

- 传统的设计通过访问一个邻居节点来锁定存储在相邻b树叶子上的两个键值之间的间隙，即使该邻居节点不能对查询或更新作出其他贡献。
- 栅栏键是在分割叶节点时发布的分隔符键的副本。在每一片叶子中，都有一个栅栏键(e。g.，上面的栅栏)总是一个幽灵记录，一个栅栏键可以是有效的或一个幽灵。围栏密钥参与密钥范围锁定，从而避免了访问相邻叶节点进行并发控制。

4.5分离键键范围锁定

在大多数商业数据库系统中，锁定的粒度是一个整个索引、一个索引叶（页）或一个单个键（具有如上面所述的键值的子层次结构和键之间的开放间隔，如上所述）。锁定物理页面和逻辑键范围可能会令人困惑，特别是在页面分割、碎片整理等时。必须考虑。另一种模型依赖于叶子[48]上方的b树级的分隔键范围锁定。这不同于在b型树的叶子级别上锁定栅栏键，即使使用了相同的键值。每个这样的锁的范围类似于页面锁，但是分隔符键上的锁是谓词锁，其方式与b-树叶中的键范围锁相同。在叶页分割期间的锁定管理可以依赖于闪烁的中间状态-树或通过复制锁从一个分隔符键到一个新张贴的分隔符。

然而，非常大的数据库表及其索引可能需要数百万页页，迫使许多事务获得成千上万个锁或锁定比它们访问的更多的数据。已经提出了介于索引锁定和页面锁定之间的中间级别的锁层次结构，尽管尚未在商业系统中使用。

其中一个建议是[48]采用了b树结构，增加了在b树上层的分隔键上的键范围锁定到叶键上的键范围锁定。在这个建议中，锁标识符不仅包括一个密钥值，而且还包括b树索引中的级别(e。g.，第0级是叶子）。这种技术承诺自然地适应倾斜的密钥分布，就像分隔键集也适应实际的密钥分布一样。

[48]的另一个建议侧重于b树键，从化合物(i。e.，多列)键，如“姓氏，名字”。这种方法的优点是，它承诺在查询和数据库应用程序中匹配谓词，从而可以最小化所需的锁的数量。串联的“通用锁定”是一种刚性形式，使用键中固定数量的引导字节来定义键范围锁定的范围。[5]

[5] 萨拉科和Bontempo [113]描述了串联的通用锁定如下：“除了能够锁定一个行或一个表的分区，不停止的SQL/MP支持的概念

这两个关于大钥匙范围上的锁的建议都需要制定许多细节，其中许多只有在第一次工业实力实施期间才会变得明显。这种方法的一种变体[4]被用于XML存储，其中节点标识符遵循层次方案，这样祖先的标识符总是其后代的前缀。

- 大型索引需要在锁定键值和锁定整个索引之间进行中间粒度的锁定。
- 传统的设计除了锁定（或代替）锁定键值外，还包括叶页上的锁。索引中的页面数以及查询中的页面锁数可能远远超过锁管理器升级到更大的粒度锁定的阈值，这通常是几千个锁。
- 或者，键范围锁定可以应用于b树中某些或所有分支节点中的分隔键。该设计使传统的层次锁定适应于b树及其层次组织。

4 B.6链接树

[6]在b型树的原始设计中，分割一个溢出节点至少会更新三个节点：溢出节点、新分配的节点和它们的父节点。在最坏的情况下，必须分裂多个祖先。为了防止其他线程或事务读取甚至更新数据结构，需要对所有受影响的节点进行锁存。持有许多b树上的锁存器的一个线程

针对按键排序的表的通用锁。通用锁定通常会影响某个密钥范围内的多个行。受影响的行数可能小于、等于或大于单个页面。在创建表时，数据库设计器可以指定要应用于主键的“锁长度”参数。此参数确定表的最佳锁粒度级别。想象一个以10个字符的ID列作为其主键的保险单表。如果为锁长度参数设置了值“3”，系统将锁定ID列的前三个字节与查询中用户定义的搜索参数相匹配的所有行。请注意，Gray和Reuter [1993]将键范围锁定解释为锁定一个键前缀，而不一定是整个键。

[6] 本部分的大部分内容都是从[51]中复制过来的。

节点明显地限制了并发性、可扩展性，从而限制了系统性能。正确的b树的定义需要一些放松，而不是削弱线程的分离，从而冒着不一致的b树的风险。其中一种设计将一个节点分成两个独立的步骤，i。e.，拆分这些节点，并在父节点中发布一个新的分隔符键。在第一步之后，溢出的节点需要一个分隔符键和一个指向其右邻居的指针，因此需要名称眨眼树[81]。

在第二步之前，节点的父节点中尚未引用正确的邻居。换句话说，父节点及其关联的子指针中的单个键范围实际上指向两个子节点。根据这个指针，根到叶搜索必须首先将搜索的键与子节点的高栅栏进行比较，如果搜索的键更高，则继续搜索到正确的邻居。为了确保有效的对数搜索行为，这个状态只是短暂的，并在第一个机会结束。拆分节点的第一步定义了分隔键，创建一个新的右邻居节点，确保两个节点中正确的栅栏键，并在旧节点中保留新节点的高栅栏键。最后一个操作并不需要在b树中进行正确的搜索，但它可以实现对b树的有效一致性检查，即使是在某些节点处于这种瞬态状态的情况下。在这种短暂状态下，旧节点可以被称为新节点的“养父节点”。

第二个独立的步骤是将分隔符键放在父步骤中。第二步可以使任何未来的副作用根叶遍历，应该尽快发生，但可能会推迟超出系统重启甚至崩溃及其恢复没有数据丢失或磁盘数据结构的不一致（见图6.11更多细节允许状态和不变量）。

眨眼的优势-树是指新节点的分配并将其初始引入b树是一个本地步骤，只影响一个已存在的节点，并且只需要在溢出的节点上有一个锁存器。缺点是，在瞬态状态期间搜索的效率可能有点低，需要一个解决方案来防止在高插入速率期间出现长列表的邻居节点，并且验证b树的结构一致性更复杂，也可能效率更低。

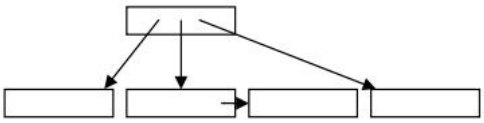


图4．12.闪烁树中的中间状态。

图4.12说明了一个在标准b树中不可能出现的状态，但在blink树中是一个正确的中间状态。这里的“正确”意味着搜索和更新算法必须处理这种状态，验证磁盘上数据结构的数据库实用程序不能报告错误。在原始状态下，父节点有三个子节点。请注意，这些子节点可能是叶节点或分支节点，而父节点可能是b树的根节点或分支节点。第一步是分割一个子节点，从而得到图4.12所示的中间状态。接下来的第二步将第四个子指针放入父指针，并放弃邻居指针，除非在b树的特定实现中需要邻居指针。请注意与2-3-树中的三元节点的相似性，如图2.2所示。

在大多数情况下，在父节点中发布分隔符键（上面的第二步）可以是由其调用的非常快速的系统事务
下一个从根到叶的遍历。此线程不是更新事务的一部分，因为b树结构中的任何更改都将是系统事务的一部分，而不是用户事务。当一个线程在父节点和子节点上都保持锁存时，它可以检查是否存在尚未发布的分隔符键。如果是，它将其锁存升级为独占锁存，在父节点中分配一个新条目，并将分隔键从子项移动到父项。如果另一个线程持有一个共享的锁存器，则该操作将被放弃，并留给后续的根到叶搜索。如果父节点不能容纳其他分隔键，则分配一个新的溢出节点，填充并链接到父节点。拆分父节点应该是一个单独的系统事务。如果根到叶搜索发现根节点有一个链接的溢出节点，则树应该按其他级别增长。如果不能立即获取任何所需的锁存器，那么系统事务可能会中止，并将其留给以后的b树遍历，以便在父节点中发布分隔符键。

在父节点中发布分隔键之前，节点必须再次分割的不太可能的情况下，多个溢出节点可以形成一个链表。可以通过将拆分操作限制在适当的父节点指向的节点上，来防止多次拆分导致的长链表。这些和进一步的细节上的眨眼-树最近在一篇详细的论文[73]中被描述过了。

眨眼的分裂过程-树可以反转，以使去除b树节点[88]。第一步创建一个邻居指针，并从父节点中删除子指针，然后第二步将删除的受害者与其邻居节点合并。这个闪烁瞬态-树甚至可能对兄弟节点之间的负载平衡和b-树的碎片整理很有用，尽管这种想法尚未在研究原型或工业实现中尝试过。

- 眨眼-树放松了严格的b树结构，以使更多的并发性。拆分一个节点并在父节点中发布一个新的分隔符键是两个独立的步骤。
- 每个步骤都可以是一个系统事务，它提交以使其更改对其他线程和其他事务可见。
- 在这两个步骤之间的瞬态状态中，旧节点是新节点的“养父节点”。瞬态应该是短暂的，但如果第二步被延迟，则可能会持续存在。g.，由于并发性冲突。
- 眨眼-树及其瞬态可能对b树的其他结构变化有用。g.，删除一个节点（合并两个节点的键范围）和负载平衡
  在两个节点之间（替换分隔符键）。

4.7锁采集期间的闩锁

如果锁是由B树中的实际键值定义的，则必须小心管理锁。具体来说，当事务试图获取密钥范围锁定时，它的线程必须在缓冲池中的数据结构上保持锁存，这样密钥值就不能被其他线程删除。另一方面，如果不能立即授予锁，线程在事务等待时不应该持有锁。事实上，

语句可能更一般：线程不能在持有锁时等待。否则，多个线程可能会相互死锁。回想一下，死锁检测和解决通常只提供锁，而不提供锁锁。

有几种设计来解决这个潜在的问题。在一个解决方案中，当检测到冲突时，锁管理器调用时，只将锁请求排队，然后返回。这使得线程在第二次调用锁管理器并等待锁之前释放适当的锁可用。仅仅在第一次调用中使锁请求失败是不够的。在锁请求插入到锁管理器的数据结构之前，需要缓冲池中数据结构的锁存以确保密钥值的存在。

另一个解决方案将一个函数和适当的函数参数传递给要在等待之前调用的锁管理器。这个回调函数可以释放缓冲池中数据结构上的锁存器，以便在等待和获取锁之后重新获取。在任何一种情况下，锁定请求期间的操作序列不仅需要表明成功与失败，还需要表明即时与延迟的成功。

当事务等待键值锁而不保持缓冲池中数据结构上的锁存时，其他事务可能会通过分割、合并、负载平衡或页面移动来改变b树结构。g.，在b树碎片整理期间，或在RAID或闪存[44]上的写优化b树中。因此，在等待密钥值锁定之后，事务必须重复其对密钥的根到叶的搜索。为了最小化此重复搜索的成本，可以在等待之前保留沿着根到叶路径的日志页面序列数，并在等待之后进行验证。或者，可以使用计数器（结构修改）来快速决定b树结构是否可能改变了[88]。这些计数器可以是系统状态的一部分，i。e.，这不是数据库状态的一部分，并且在系统崩溃后不需要恢复它们以前的值。

- 在获取键值锁时，数据页必须保持锁存状态，以防止删除键值，但在等待锁时，不能保留数据页上的锁存状态。

- 解决方案需要回拨或重复呼叫锁管理器，一个是将锁插入等待队列，另一个是等待锁获取。

4.8锁扣联轴器

当根到叶的遍历从一个b树节点前进到它的一个子节点时，在读取指针值（子节点的页面标识符）和访问子节点之间存在一个简短的漏洞窗口。在最坏的情况下，另一个线程会在此期间从b树中删除子页面，甚至可能开始在另一个b树中使用该页面。如果子页面存在于缓冲池中，则概率很低，但不能忽略它。如果忽略或未正确实现，很难将此漏洞确定为数据库损坏的原因。如果缓冲池中不在子页面，并且需要I/O，则需要其他考虑，这将在下一个小节中讨论。

一种叫做锁存耦合的技术避免了这个问题。根叶搜索保留父页面上的锁存，从而保护页面不受更新，直到它获得子页面上的锁存。一旦子页面在缓冲池中被定位、固定并锁定，父页面上的锁定器就会被释放。如果子页面在缓冲区池中很容易，父页面和子页面的锁存只会在很短的时间内重叠。

锁存耦合是在b树[9]的早期发明的。对于只读查询，一次最多需要锁定（锁定两个节点），两者都处于共享模式。在最初的插入设计中，独占锁保留在根到叶路径的所有节点上，直到找到一个节点有足够的空闲空间允许分割子节点并发布一个分隔键。不幸的是，可变大小的记录和键可能会迫使用户做出非常保守的决定。相反，更新的设计依靠眨眼-树（临时邻居指针，直到可以显示分隔键）或重复从根到叶的过程。初始的根-叶传递在根节点和分支节点上使用共享锁存器，即使预期的操作将通过插入或删除来修改叶节点。

依赖于邻居指针来实现有效的游标、扫描和键范围锁定的系统。e.，没有利用栅栏键的实现，也在邻居节点之间使用锁存耦合。在这些系统中，多个线程可能会试图从不同的方向锁定一个叶页，这可能会导致死锁。回想一下，锁存器通常不支持死锁避免或检测。因此，锁存器获取必须包括故障快速无等待模式，并且b树代码必须处理失败的锁存器获取。

大多数从根到叶的遍历一次最多有两个b树节点，i。e.，有一个父母和一个孩子。拆分操作需要包含三个b树锁存器，其中一个用于新分配的节点。此外，它还需要锁定空闲空间信息。在眨眼-树，分割操作一次只需要两个锁存器。即使是将分隔符键和指针从子节点移动到父节点的最终操作，也只需要两个锁存器；不需要锁存新节点。另一方面，闪烁树中的完整分割序列需要两个具有独占锁存的周期，即使最后的操作可以延迟到适当的锁存随时可用。

- 在从一个b树节点导航到另一个b树节点时，指针必须保持有效。通常的实现保持源被锁定，直到目标被锁定。
- 如果需要I/O，则应释放锁扣。b树导航可能需要重复，可能从根节点开始。
- 眨眼-树一次最多锁定两个节点，即使在分割一个节点和发布一个分隔键。

4.9生理记录

除了锁定之外，支持事务所需的其他基本技术是日志记录。e.，编写足够的关于数据库更改的冗余信息，以处理事务故障、媒体故障和系统故障[56,59]。必须将描述更新操作的日志记录写入可靠的存储器，然后才能将修改后的数据页面从缓冲池写入到数据库中的位置，

激励着“提前写日志”这个名字。日志的主要优化是减少日志体积。

数据库页面中的每一个更改都必须可以恢复，在事务失败或媒体失败时，必须采用“撤消”和“重做”模式。在传统的物理日志记录方案中，必须详细地记录这些操作。如果只有一条记录受到更改的影响，那么将该记录的“前像”和“后像”复制到恢复日志就足够了。在最早的方案中，整个页面的两个图像都被记录了下来。换句话说，在一个8 KB的页面中更改20个字节需要将16 KB写入恢复日志，以及适当的记录头，这对于日志记录[59]来说是相当大的。

在逻辑日志记录方案中，只记录插入和删除，包括适当的记录内容，而不引用更改的特定物理位置。该方案的问题是没有记录对自由空间和数据结构的修改。例如，拆分b树节点在恢复日志中不会留下任何跟踪。因此，一些康复病例变得相当复杂和缓慢。例如，如果单个b树存储在多个磁盘上，而其中一个磁盘发生故障，则必须通过恢复早期的备份副本和“重播”整个设备集的记录历史记录来恢复所有它们。如果检查点，除非永久关闭，否则系统崩溃后的恢复成本会更加昂贵，这与今天的高性能检查点技术相矛盾，如二机会检查点、模糊检查点和检查点间隔。

第三种选择结合了物理日志和逻辑日志，通过记录页面的每个内容更改，但仅通过它们的槽号而不是字节位置[59]来引用页面内的记录。在这种“生理”日志记录中，[7]恢复单个媒体甚至单个页面是可能的和有效的，但通常可以避免记录整个页面的副本，剩下的日志记录可以简化或缩短日志记录。特别是，页面内空间管理的更改。

[7] 这个名字是“物理”和“逻辑”的结合；它不是指医学术语“生理学”，这可能会令人困惑。

	页面压缩	*记录删除*
物理日志记录	在之前和之后的全页图像	删除的记录
生理记录	鬼魂移除	幽灵位中的变化
逻辑日志记录	没有什么	行删除

图4．13个日志记录方案和b形树操作。

图4.13总结了b树节点中两个操作的物理、逻辑和生理日志记录。物理日志记录很简单，但代价昂贵。逻辑日志记录意味着复杂的恢复。生理测井是为了达到良好的平衡，在现代数据库中很常用。

Gray和Reuter [59]将生理日志描述为“页面的物理日志，页面中的逻辑日志”。生理日志记录的逻辑方面有时与物理“撤销”操作和操作[95]的逻辑补偿之间的差异相混淆。例如，在b树叶中插入一个新记录可能需要在不同的页面中进行逻辑补偿，特别是如果新插入的b树条目在由于相同的事务或另一个事务的另一个插入而导致相关的叶节点被拆分后移动。换句话说，生理日志可以容忍页面内的表示变化，e。g.，由于压缩或可用空间压缩，但它本身并不能通过锁定和精细粒度的现代恢复方案所要求的补偿来实现逻辑“撤销”。

增加记录大小的插入、删除和更新可能需要重新组织b树节点。这种压缩操作包括删除鬼影记录，巩固空闲空间，也许还会改进压缩。在传统的物理日志记录方案中，必须详细地记录这些操作，通常是通过将整个页面的前映像和后映像同时复制到恢复日志中。在生理日志中，没有详细记录空闲空间的巩固和页面内需要移动的记录；事实上，根本不需要记录这样的移动。另一方面，必须记录删除鬼记录，因为鬼记录占据了间接向量中的槽，因此删除它们会影响其他有效记录的槽号。如果没有记录幽灵删除，后续的日志记录引用特定的页面和

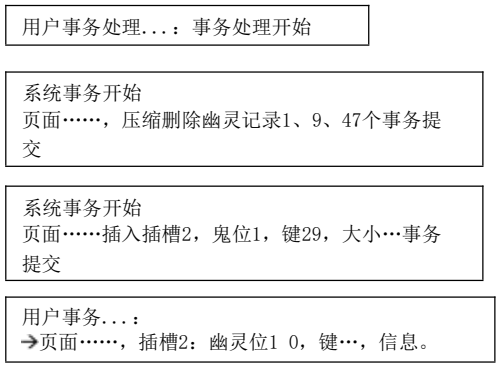


图4．14插入键值为29的复杂记录的日志记录。

在恢复期间，槽号可能被应用到错误的记录。但是，请记住，如果此日志记录还包含对删除事务的提交，则可以使用一个短的日志记录来记录删除一个或多个幽灵记录，从而省略记录内容。

图4.14显示了一个可能的日志记录序列，最终，将一个新的记录放入一个b树节点中。在b树中的搜索确定了正确的叶子，发现它有足够的未使用空间，但它在页面中不是连续的。因此，可以调用页面压缩，并且生成的日志信息非常少，远远少于页面的两个完整图像（之前和之后）。即使使用压缩文件，完整的页面图像也可能相当大。作为系统事务运行，页面压缩不需要锁，只需要锁存，并且即使用户事务失败，其效果仍然有效。另一个系统事务创建具有所需大小和键值的幽灵记录；这还确定页面内适当的插槽号。此事务也不获取锁，但它需要验证是否存在保护另一个事务的可序列化性的锁。最后，用户事务将鬼影记录转换为有效记录，并填充与键值相关联的信息。这两个系统事务可以合并为一个。

- 生理日志与医学或生理学无关；它意味着记录“物理到页面，逻辑”

在一个页面中。”换句话说，页面由它们的物理地址（页面标识符）引用，页面内的记录由它们的插槽号或键值引用，而不是它们的字节地址。

- 页面内的空间碎片整理不需要任何日志记录。如果其他日志记录按槽号（而不是键值）引用B树条目，则插入或删除幽灵需要一个日志记录。

4.10未被记录的页面操作

另一个日志优化涉及结构b树操作，i。e.，分割一个节点，合并相邻节点，并平衡相邻节点之间的负载。至于页面内压缩，可以避免详细的日志记录，因为这些操作不会改变b树的内容，只会改变它的表示。然而，与页面内压缩不同的是，这些操作涉及到多个页面，而且单个页面的内容确实发生了变化。

所考虑的操作实际上反映在恢复日志中；在这个意义上说，常用的术语“未记录”在字面上并不准确。一个更好的描述性名称可能是“仅分配日志记录”。尽管如此，节省还是相当可观的。例如，在严格的物理日志记录中，分割一个8 KB的节点可能会生成24 KB的日志卷加上日志记录头、一些用于页面分配的短日志记录和事务提交，而优化的实现可能只需要这几个短日志记录。

关键的见解是，旧的页面内容，e。g.，拆分前的完整页面，可以用来确保拆分后两个页面的可恢复性。因此，必须保护旧的内容，直到移动的内容被安全写入。例如，在将完整页面加载到缓冲池中并发现需要拆分后，需要执行多个步骤：

1. 在磁盘上分配了一个新的页面，并记录了此分配，
2. 此新磁盘页面的新页面帧被分配在缓冲池中，

3．一半的页面内容被移动到缓冲池中的新页面；这个移动用一个短日志记录记录，不包括移动记录的内容，但可能包括记录计数，
4．新的页面将被写入数据存储区，并且
5．旧页面被写入数据存储中，只剩下一半的原始内容，覆盖旧页面内容，从而丢失移动到另一个页面的一半。

在步骤4和步骤5中小心的写顺序是至关重要的。此操作列表不包括在完整节点及其新兄弟节点的父节点中发布新的分隔符键。进一步的优化是可能的，尤其是眨眼树[73]。上面列表中的日志记录可以合并到一个日志记录中，以便节省记录头的空间。上述列表的关键方面是，在前一个操作完成之前，不能尝试最后一个操作。前三个操作和后两个操作之间的延迟可以任意长，而不会危及可靠性或可恢复性。

这种技术的变体也适用于其他结构B-tree操作，特别是合并相邻节点，平衡相邻节点之间的负载，和移动项在相邻的叶子或分支节点为了重建所需的分数的空闲空间的快速扫描和快速插入之间的最佳权衡。在所有这些情况下，如上所述的仅分配日志记录可以保存物理日志记录中所需的大部分日志卷。关于非记录页面操作的更多细节将在第6.6节中讨论。

- “未记录”应视为“未记录页面内容”。另一个名称是“仅分配日志记录”或“最小日志记录”。
- 当将记录从一个页面移动到另一个页面时（在拆分、负载平衡或碎片整理期间），旧的页面可以用作备份。在将目标页面保存在存储器上之前，必须先保护它。

4.11未被记录的索引创建

术语“非记录的索引创建”似乎很常用，尽管它并不完全准确。将记录数据库目录和空闲空间管理信息中的更改。但是，b形树页面的内容并没有被记录下来。因此，与已记录的索引创建相比，非已记录的索引创建可能节省了99%的日志卷。

所有新分配的b树页面，包括叶子和分支节点，都必须在提交操作之前从缓冲池转移到数据库中的永久存储中。当然，b树节点的图像可能会保留在缓冲池中，这取决于缓冲池中的可用空间和替换策略。磁盘上的页面分配进行了优化，允许在最初写入b树时进行大顺序写入，以及在未来索引扫描期间进行大顺序读取。

图4.15比较了已记录索引创建和未记录索引创建中的日志卷。不记录大量的操作，特别是单个记录插入或完整的b树页面。例如，只记录了数百万条页面分配，而不是数百万条记录。如果在加载处理期间允许页面在缓冲池中逗留，则提交处理会很慢。但是，就像表扫描与缓冲区池中的LRU替换发生严重的交互一样，在加载处理过程中填充的页面应该尽快从缓冲区池中弹出。

恢复未记录的索引创建需要精确地重复原始索引创建，特别是空间分配操作，因为后续的用户事务及其日志记录可能引用特定数据库页面中的特定键，e。g.，在行删除。当恢复这些事务时，他们必须在该页面中找到该键。因此，在恢复期间，数据库页面的节点分割和分配必须精确地重复原始执行。

行动	*记录索引创建非*	
页面分配	记录分配表中的更改	同一的
记录插入	每个页页1个日志记录	强制页页页
叶片分裂	2-4个日志记录	强制分支节点
分支节点拆分	2-4个日志记录	强制所有节点

图4．15记录和未记录索引创建中的记录详细信息。

- 由于索引可能非常大，因此记录新索引的整个内容可能会超过可用的日志空间。大多数系统都有用来创建非记录的辅助索引的工具。
- 在完成后，新的索引将被强制进行存储。
- 事务日志的备份必须包含新索引；否则，即使包含在事务日志和日志备份中，也不能保证对新索引的后续更新。

4.12在线索引操作

索引创建的另一个重要优化是在线索引创建。如果不创建联机索引，其他事务可以基于已存在的索引查询表；创建联机索引，并发事务还可以更新表，包括插入和删除，在创建索引之前正确应用更新到索引。

这里描述的传统技术对于小的更新已经足够了，但是仍然不能在同时修改物理数据库设计并创建和删除索引时执行批量插入或删除。有两种主要设计：要么将并发更新应用于仍在构建的结构，要么将这些更新在其他地方捕获，并在主索引创建活动完成后应用。这些设计被称为“无边文件”和“侧文件”[98]。恢复日志可以作为“侧边文件”。

斯里尼瓦桑和Carey [119]进一步划分了索引创建的在线算法，特别是“侧边文件”方法。在他们的比较研究中，所有并发的更新都被捕获在一个列表或一个索引中。它们不考虑在恢复日志或目标索引（“无边文件”方法）中捕获更新。它们的各种算法允许在整个索引创建过程中或仅在其扫描阶段进行并发更新。他们的一些算法对并发更新的列表进行排序，甚至将其与由索引构建器扫描和排序的候选索引条目合并。他们的总体建议是使用一个并发更新的列表（一个侧边文件），并将其与索引构建器的候选索引条目合并。

4．12在线索引操作297

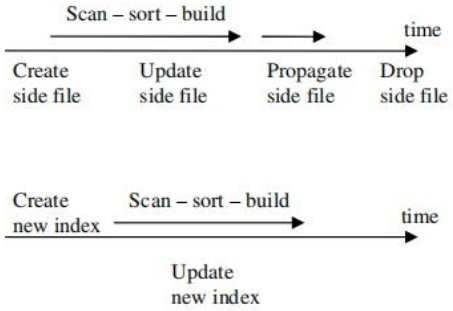


图4．16在线索引创建有和没有边文件。

图4.16说明了使用或不使用“边文件”的联机索引创建的数据流。上部操作从创建空边文件开始（除非恢复日志作为边文件）。并发事务在那里缓冲了它们的更新，并且在扫描、排序和b树加载完成后传播侧文件的全部内容。较低的操作从创建新的索引开始，尽管此时它完全为空。并发事务捕获新索引中的插入和删除，甚至在b树加载之前和加载期间也是如此。

“侧文件”设计允许索引创建继续进行，而不考虑并发更新。这个索引创建过程应该构建初始索引和离线索引创建一样快。基于“侧文件”的最终“追赶”阶段要么需要静止的同步更新活动，要么需要在捕获更新并将它们应用到新索引之间的竞争。一些系统执行固定数量的追赶阶段，第一个追赶阶段应用在索引创建期间捕获的更新，第二个追赶阶段应用在第一个追赶阶段捕获的更新，最后的追赶阶段应用剩余的更新并防止新的更新。

“无边文件”设计要求在开始时创建未来索引为空，并发更新修改未来索引，索引创建过程围绕由并发更新事务插入的未来索引中的记录工作。一个问题是，索引创建过程可能无法实现类似于离线索引创建的写带宽。另一个问题是并发更新

事务可能需要删除索引创建过程尚未插入的键范围中的键。例如，索引的创建可能仍然是在对要插入到新索引中的记录进行排序。

这种删除可以用负记录或“反物质”记录来表示。当索引创建过程遇到反物质记录时，相应的记录将被抑制，而不会插入到新的索引中。当时，反物质记录已经发挥了它的作用，并被从b树中删除。当索引创建过程插入其所有记录时，必须将所有反物质记录从B树索引中删除。

反物质记录和鬼唱片是完全不同的。幽灵记录表示一个完整的删除，而反物质记录表示一个不完整的删除。换句话说，反物质记录表明索引创建过程必须抑制一个看似有效的记录。如果给记录权重，有效记录的权重为+1，幽灵记录的权重为0，反物质记录的权重为-1。

第二个并发事务有可能通过留下一个反物质记录来插入一个以前被删除的键。在这种情况下，需要一个带有抑制标记的有效记录。抑制标记表示第一个事务执行了删除；有效记录的其余部分包含由第二个事务插入到数据库中的信息。第三个并发事务可能会再次删除此键。因此，抑制标记和鬼记录是完全正交的，除了带有抑制标记的鬼记录不能像其他鬼记录一样被删除，因为这样会丢失抑制信息。

图4.17说明了在没有侧文件的在线索引创建期间使用鬼位和反物质位的情况。在索引创建过程加载索引项之前，键47和11都会在索引中进行更新。这个批量负荷显示在图4.17的最后两个条目中。键值47的历史记录从一个插入开始；因此，它从不设置反物质位。键值11的历史记录从删除开始，删除必须引用索引创建过程要加载的未来索引项；因此，这个键值保留其反物质位，直到对加载流中的记录被取消为止。

行动	*鬼反物质*	
插入键47	不	不
删除密钥11	是	是
删除键47	是	不
插入键11	不	是
删除密钥11	是	是
加载键11	是	不
加载键47	不	不

图4．17反物质在在线索引创建没有侧文件。

键值11的最终结果可以是一个无效的（幽灵）记录，或者根本没有记录。

然而，在物化摘要（“按”分组）视图中，幽灵标记和抑制标记可以统一为一个计数器，该计数器提供类似于引用计数[55]的角色。换句话说，如果它的引用计数为零，则摘要记录为幽灵记录；如果它的引用计数为负数，则摘要记录意味着在联机索引创建期间的抑制语义。在包含对每个唯一键值的引用列表的非唯一索引中，对于单独的键值和引用对需要一个反物质位。唯一键值的引用计数可以使用类似于鬼位，i。e.，当且仅当计数为零时，才可以删除一个键值。

维护有效性可疑的索引不仅适用于在线索引的创建，也适用于索引的删除。e.，从数据库中删除一个索引，还有两个注意事项。首先，如果在较大的事务中删除索引，则并发事务必须继续进行标准索引维护。只要包括索引删除在内的事务仍然可以被中止，这就是必需的。其次，数据库页面的实际去分配可以是异步的。在提交B-tee删除后，必须停止并发事务处理的更新。此时，一个异步实用程序可以扫描整个b树结构，并使用空闲空间信息将页面插入到数据结构中。这个过程可以分为多个步骤，这些步骤可以同时发生，或者在步骤之间出现暂停。

最后，如上所述的在线索引的创建和删除很容易被数据库用户认为仅仅是第一步。这个

上述技术在开始和结束时需要一两个短暂的静止时间。在表或索引的相应数据库目录上需要使用独占锁。根据应用程序、其事务大小和响应时间的要求，这些静止期可能会造成令人痛苦的破坏性。“完全在线”索引操作的实现可能需要对数据库目录和预编译的查询执行计划的缓存进行多值并发控制。在文献中还没有描述过这样的实现。

- 在线索引操作允许通过并发事务进行更新，同时未来的索引条目将被提取、排序并插入到新的索引中。当新的索引在数据库目录中插入时，以及在最终的事务提交期间，大多数实现都会锁定受影响的表及其模式。
- 并发事务的更新可以立即应用于新索引（“无边文件”）或在初始索引创建完成后（“边文件”）。前者要求“反物质”记录来反映索引中的关键值的历史记录是由删除开始的；后者需要基于更新日志的“追赶”操作。

4.13事务处理隔离级别

因此，低于序列化性的事务隔离级别允许不正确的查询结果[59]。在运行查询以计算更改集的更新语句中，较弱的事务隔离级别也可能导致对数据库的错误更新。如果并发控制应用于单个索引，例如在主和次b树中使用键范围锁定，可能的结果通常不能很好地理解。商业系统、命令集和文档在隔离级别的定义、名称和语义方面有所不同，这并没有帮助。例如，MicroftSQLServer中的“可重复读取”保证记录一旦读取，就可以被同一事务再次读取。尽管如此，一个事务还是可以插入满足另一个事务的查询谓词的记录，以便在同一事务中第二次执行相同的查询

产生额外的结果，即所谓的“幻影”行。在IBM DB2 LUW中，“可重复读取”保证了完全的可序列化性。e.，它可以保护幽灵。在b树中的锁定方面，SQL Server中的“可重复读取”会锁定键值，而DB2中的“可重复读取”也会锁定键值之间的间隙，i。e.，它应用键范围锁定为
在4.3节中介绍。

由于事务隔离级别在[58,59]别处定义并解释了，这里有两个例子就足够了。这两个问题都可能出现在“读取提交”（SQL Server）和“游标稳定性”（DB2）隔离级别中，这是多个产品中的默认隔离级别。第一个示例显示了一个丢失的更新，这是一个关于弱事务隔离级别影响的标准教科书示例。两个经交流电
版本T1和T2读取相同的数据库记录和相同的属性
值，比如10。此后，事务T1将该值增加1到11，而事务T2将该值增加2到12。在两个事务提交之后，最终值是11或12，这取决于哪个事务写入最终值。如果两个增量操作都被串行应用，或者与可序列化事务隔离并发应用，最终值将是13。

第二个示例说明了由于单个索引中的锁定获取（和释放）而导致的单个结果行中的不一致性。如果在索引交集或索引连接中使用了单个表的多个索引，则可能会对不一致的记录计算多个谓词子句。在极端情况下，一行可能会在基于数据库中从未存在过的行的查询结果中包含（或排除）。例如，假设一个查询谓词“其中x = 1和y=2”针对一个在x和y上有单独索引的表，以及一个查询执行计划，首先探测x上的索引，然后是y上的索引，并使用哈希连接算法将两者结合起来。一行可能以满足查询谓词的值开始，但它的y值可能在扫描y上的索引之前已经改变；另一行可能以满足查询谓词的值结束，但它的x值只有在扫描x上的索引后才会更新；等等。

图4.18说明了这个例子。数据库从未包含满足查询谓词的有效已提交行。尽管如此，由于较弱的事务隔离级别和交错执行

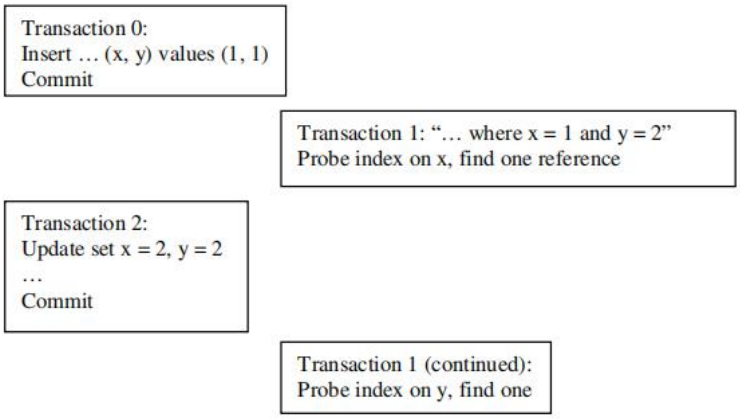


图4．18由于事务隔离级别较弱而导致的查询结果错误。

对于事务1和事务2，查询返回一行。如果这两个索引不能覆盖查询和所需的列，则事务处理1中的最终操作将从一个主索引中获取所有所需的列。在这种情况下，结果行包含两列的值2，它不满足查询谓词。另一方面，如果索引覆盖了查询，则结果行包含值1和值2，这些值满足谓词，但从未同时存在于数据库中。

此外，索引扫描可以对在索引交集完成时不再存在的行产生引用。在传统的查询执行中，索引扫描和获取行之间的延迟很小；如果对引用进行排序或从多个辅助索引组合了引用列表，则延迟可能会很大，并且更容易使并发更新事务干扰查询执行。

在理解问题的情况下，一种常见的方法是在组装完表中的整个行之后，重新计算表中的所有谓词子句。不幸的是，这似乎只是解决了这个问题，但它并没有真正解决这个问题。该技术可以确保没有单个结果行明显违反查询谓词，但它不能确保生成的查询结果是完整的。它也不跨数据库中的多个表扩展任何保证。例如，查询优化可能会基于外键约束删除冗余连接；如果查询执行计划与弱查询执行计划一起运行

但是，在事务隔离级别上，省略或包含看似冗余的连接可能会导致不同的查询结果。

如果查询优化可以在实体化视图（及其索引）和基表之间自由选择，则存在同样的问题。如果查询执行计划包含实体化视图与其底层表之间的连接，则问题会更严重。例如，实例化视图及其索引可以开始选择，然后从数据库中的基本表中获取详细信息。

另一种方法承诺了更多可预测的结果和执行语义，但它更复杂和严格。它依赖于一个查询执行期间的临时序列化性。一旦一个计划的执行完成，锁可以被适当地降级或释放。上面列出的索引交集、基于外键约束的半连接删除、从实例化视图到基表的“反向连接”等并发控制问题。可以用这个方法来解决。但是，如果开发人员从未使用程序状态或临时数据库表将信息从一个查询传输到下一个查询，那么这种方法才有效。例如，如果查询优化不能可靠地选择一个好的计划，那么数据库应用程序开发人员通常会使用临时表将一个复杂的查询分解为多个语句。例如，如果一个语句计算了一组主键值，那么下一个语句可能依赖于数据库中存在的所有主键值。当然，这是早期连接和反向连接问题的一个变体。

也许最可靠的解决方案是建议用户避免较弱的隔离级别或支持具有较强的事务隔离级别的嵌套事务。嵌套事务可以隐式地用于单个查询执行计划，也可以由用户显式地用于其脚本中的单个语句或适当的语句组。在创建或部署新数据库时，将序列化性设置为默认隔离级别是很好的第一步，因为它将确保用户只有在积极地“选择加入”这个复杂的问题之后，才会出现错误的查询结果和错误的更新。

- 弱事务隔离级别（可重复读取、读取提交等）避免了正确的和完全的隔离

并发事务，以获得并发性、性能和可伸缩性。

- 许多数据库系统使用弱事务隔离级别作为默认值。许多用户和应用程序开发人员可能并不完全理解它对应用程序正确性的影响。

4.14总结

总之，有许多发明已经改进了基于b树索引的数据库的并发控制、日志记录和恢复性能。锁定和锁定、数据库内容和内存中数据结构的分离，与在删除和可能的插入期间由幽灵记录辅助的密钥范围锁定一样重要。在大型索引实用程序期间，减少日志卷，特别是未记录（或仅分配记录）索引创建，防止了几乎与数据库一样大的日志空间，但它引入了从缓冲区池中强制脏页面的需要。最后，弱事务隔离级别对于增加并发性似乎是个好主意，但它们可能会引入错误的查询结果，并且在计算数据库更改的更新中使用时，会导致对数据库的错误更新。

也许未来最迫切需要的发展方向是简化。用于并发控制和恢复的功能和代码过于复杂，无法进行设计、实现、测试、调试、调优、解释和维护。消除特殊情况而性能没有严重下降
或者所有数据库开发和测试团队都具有可伸缩性。

5

查询处理

b树索引的作用只有利用它们的查询执行技术和考虑这些查询执行计划的查询优化技术的补充。因此，本节总结了通常与b树索引一起使用的查询执行技术。在关于查询处理的b树技术的个别小节之前，简要介绍了查询处理及其twp主组件、查询优化和查询执行。对自动查询优化的需求和机会来自于基于SQL等非程序性数据库语言的用户界面和物理数据独立性。

术语物理数据独立性描述了表中的逻辑数据组织和（实体化）视图与堆文件和索引中的物理数据组织之间的分离。表包含由名称标识的列和由唯一主键标识的行；行包含列值。文件和索引包含由堆中的记录标识符或索引中的唯一搜索键标识的记录；记录包含字段。当然，行和记录之间以及列和字段之间存在关系，但是物理数据独立性允许这种关系相当松散。

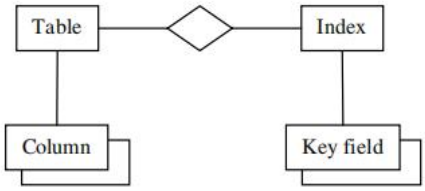


图5．1入口关系图，包括表、索引和b树。

许多优化机会都来自于利用这些松散的关系。在上面使用的术语中，表和索引之间以及逻辑行和物理记录之间可能存在多对多的关系。有趣的是，可以将适当模式概念之间的实体和关系建模得与用户数据库中的实体和关系非常相似，并且可以使用标准技术导出所需的模式表。当然，不仅逻辑数据库设计，物理数据库设计也适用于目录表，包括为目录信息将数据放置在内存缓存中。

图5.1显示了实体类型表和索引，以及表列和索引字段的设置值属性。在大多数数据库管理系统中，表和索引之间的关系是一对多的关系，i。e.，一个表可以有多个索引，但一个索引只属于一个表。多对多的关系将表示支持连接索引[123]。索引和b树之间的关系可以是一个简单的一对一关系，无需在ER图中进一步阐述，如果每个索引可以被划分为多个b树，则为一对多关系，或者主细节聚类的多对一关系，e。g.，使用合并的b型树[49]。

根据软件系统提供的设施，物理数据库设计可能被限制为索引调优，或者它也可以利用水平和垂直分区、面向行或列的存储格式、压缩和位图编码、页面和索引中的空闲空间、排序顺序和主细节集群，以及许多其他选项[38]。鉴于自动化的趋势，似乎关于自动化的选择也应该包括在物理数据库设计中。g.，启用自动创建和维护统计信息

（直方图）和软约束[60]，索引的自动创建和优化，以及具有它们自己的索引、统计数据和软约束的实体化视图。

由于数据库查询指定了表，但查询执行计划要访问索引，因此需要进行映射。物理数据独立性支持并需要在此映射中进行选择。查询优化可以是资源密集型的，传统的数据库管理软件设计是将编译时查询优化和运行时查询执行分开的。除了访问路径选择外，查询优化还选择了执行算法(e。g.，以及处理序列(e。g.，连接顺序）。

这些选择将包含在查询执行计划中。查询执行计划通常是一个树，但在常见的子表达式的情况下，它可能是一个dag（有向无环图）。请注意，在查询优化过程中可能会引入常见的子表达式，e。g.，用于具有多个聚合的查询，包括SQL中具有“不同”关键字的查询，具有垂直或水平分区的表，以及大型更新。

图5.2显示了一个简单的查询执行计划，该数据结构由编译时查询优化构建，并由运行时查询执行进行解释。这些节点指定了算法和所需的定制设置，e。g.，谓词、排序子句和投影列表。理想情况下，这些代码被编译成机器代码，尽管特殊用途的字节码和运行时解释在今天似乎更常见。

在大多数查询执行体系结构中，控制从消费者流到生产者，数据从生产者流到消费者

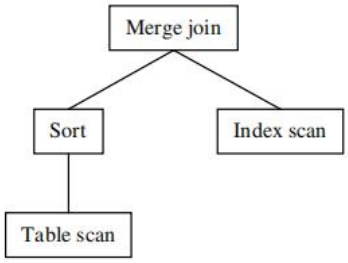


图5.2一个查询执行计划。

每个查询执行计划。数据流的单位可以是一个记录、一个“值包”[78]、一个页面，或任何其他被认为是复杂性和性能之间的良好权衡的单元。树中的控制流通常使用迭代器[39]来实现，从而产生一个自上而下的控制流。对于具有多个使用者的共享（公共）子计划，自顶向下的控制流的例外情况是可取的，并且需要具有嵌套迭代的查询执行计划，i。e.，嵌套的SQL表达式，由于其复杂性或基于索引搜索的高效执行策略，在查询优化过程中没有“扁平”。如果查询执行在一个管道中使用了多个线程，那么自底向上的控制通常似乎更可取。管道中的生产者和消费者之间的流动控制使这种区别实际上是沉默的。底部线程启动似乎与自上而下的迭代器相矛盾，但实际上不是[40]。

有些操作，最明显的是排序，具有不重叠的输入和输出阶段，在许多情况下还有一个中间阶段。这些操作员阶段描绘了计划阶段，e。g.，从一个排序操作（在输出阶段）通过合并连接到另一个排序操作（在输入阶段）的管道。这些操作被称为走走停停的算法、管道中断操作和类似的名称；它们在查询执行计划中的出现会明显地影响资源管理，如内存分配。

图5.3将计划阶段添加到图5.2的查询执行计划中，如排序表扫描结果的启停操作所示。中间的计划阶段是可选的：如果内存分配

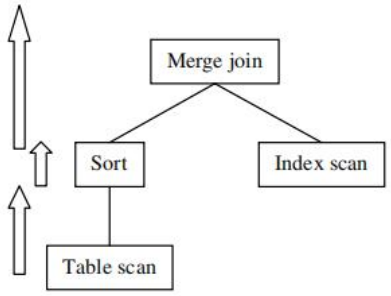


图5.3计划阶段。

而排序操作的输入大小只需要一个合并步骤，从而省略中间阶段。

在大多数传统系统中，查询执行计划是相当严格的：在查询优化过程中做出选择，而查询执行的作用是遵循这些选择。在查询执行过程中存在各种选择的技术，从资源管理(e。g.，排序和哈希连接的工作空间的大小)到计划[84]的有限突变和记录[3]的自由路由。在查询执行期间的选择可能受益于在查询优化过程中使用的假设和估计的运行时验证，e。g.，在中间结果中记录计数和数据分布。

b树索引与其他索引的特征之一是它们支持有序检索。索引条目的排序支持多列索引中的范围谓词和许多谓词。对中间查询结果的排序有助于从索引到索引的导航、从磁盘检索、设置操作，如索引交集、合并连接、基于顺序的聚合和嵌套迭代。

- 由于非程序性查询语言和物理数据独立性，编译时查询优化选择了一个查询执行计划，即由查询执行操作组成的数据流图，基于基数估计和替代的查询表达式和执行计划的成本计算。
- b树索引在查询执行计划中进行检索(e。g.，在查询文本中查找文字)和进行有序扫描。
- 查询优化还可以改进更新的执行情况（索引维护）。

5.1磁盘顺序扫描

现在，我们转向特定的b树技术，以便在大型数据存储中进行有效的查询处理。

大多数b树扫描将由b树结构引导，输出以与索引相同的排序顺序（“索引顺序扫描”）。深度，多页的预读可以由父母的信息和

父节点。如果查询必须扫描b树中的所有叶子，则扫描可以由b树的分配信息进行引导。这些信息被保存在自由空间管理环境中的许多系统中，通常以位图的形式存在。基于这种分配位图的扫描可以在存储设备上产生较少的搜索操作（“磁盘顺序扫描”）。在这两种扫描中，b树的分支节点都必须被读取到除了叶子之外的分支节点，所以这些扫描在传输量上几乎没有区别。如果不需要排序输出，则磁盘顺序扫描通常会更快。

根据b树的碎片化以及索引顺序扫描中所需的查找操作的数量，磁盘顺序扫描可能会更快，即使需要少于所有的b树叶。在极端碎片化的情况下，磁盘顺序扫描和显式排序操作可能比索引顺序扫描更快。

图5.4显示了一个小的、严重破碎的b树。从根到叶的搜索不受碎片化的影响，但是一个大范围的查询或完整的索引顺序扫描必须频繁地搜索，而不是读取连续的磁盘段。由分配信息引导的磁盘顺序扫描可以读取15个连续的页面，甚至可以读取（然后忽略）两个未使用的页面（第一行中心和最右边的第二行）。

另一种加速扫描的技术通常与磁盘顺序扫描相关，尽管不总是，是协调扫描[33,132]。这种并行扫描的优化现在在一些数据库系统中被利用。当新扫描启动时，系统首先检查新扫描是否可以按任何顺序消耗项目，以及是否

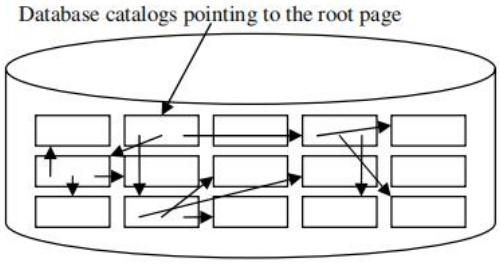


图5.4一个严重破碎的b树。

另一个扫描已激活同一对象。如果是这样，扫描被链接，新的扫描从先前扫描的当前位置开始。在以前的扫描完成后，新的扫描必须重新启动扫描，以获取以前跳过的项目。这种技术适用于任意数量的扫描，有时也被称为“旋转木马”扫描，因为每个数据使用者都要进行一次连续活动的扫描。这种类型的扫描的潜在问题是并发控制（如果所有扫描器锁定整个表、索引或等效值，这最好工作）和带宽。例如，如果两个查询以非常不同的速度处理扫描的项目，e。g.，由于具有用户定义函数的谓词，这两个扫描应该被取消链接。

协调扫描更复杂；它们的初始化方式是利用保留在缓冲池中的页面，它们可能会多次链接和断开链接，并优化整个系统的未来I/O吞吐量，这是基于共享的数量，每个查询中剩余工作的数量，以及查询饥饿的危险。为了减少管理工作，这些考虑事项被应用于“块”或页面组，而不是单个页面。

另一方面，智能预读和预取技术可以提高碎片索引中的索引顺序扫描的性能。这些技术优化了在一个分支节点中引用的许多子节点之间的读取序列，以最小化存储设备中的查找操作的数量。

随着内存大小的不断增加，以及半导体存储上的数据越来越多，如闪存、共享和协调扫描以及智能预取，可能会失去b树索引和数据库查询处理的重要性。然而，为了充分利用共享大型CPU缓存的多个核，将来也有可能需要这些技术。

- 如果选择索引是由于它的列集，而不是由于它的排序顺序或支持谓词，并且如果索引是碎片化的，则由分配信息引导的磁盘顺序扫描可能比索引顺序扫描更快。

5.2提取行

如果逻辑表及其行直接映射到堆文件或主索引及其记录，则许多查询执行计划将从辅助索引获取引用（记录标识符或主索引中的键），然后为每行获取其他列。可以使用索引嵌套循环连接轻松实现获取行，即使该算法比获取更通用，因为它允许为每个外部输入记录提供任意数量的内部结果记录。

对于最中执行，这是传统的策略，仍然很常见，这可能导致大量的随机I/O操作。因此，次要索引似乎只对极具选择性的查询有价值。最近许多关于数据库查询执行的研究都集中在在没有辅助索引的情况下扫描大型表上，例如使用协调扫描[33,132]、数据压缩[69,111]、柱状存储[122]和谓词评估的硬件支持，e。g.，gpu或FPGAs [37]。以下技术可能会使盈亏平衡阈值偏向于次级指标。

图5.5说明了一个简单操作的三个竞争计划的执行成本，即从一个表中选择一组行。在左侧，表中的所有行都会被查询谓词拒绝；在右侧，所有行都满足该谓词。扫描表（或包含所有行和列的数据结构）的成本几乎相同，但与结果大小无关。它要求每个页面或每个范围（连续页面序列）的顺序I/O。在传统的数据库管理系统中，如果输出的基数估计不可靠，这是最稳健的计划，这通常是一种情况。这个

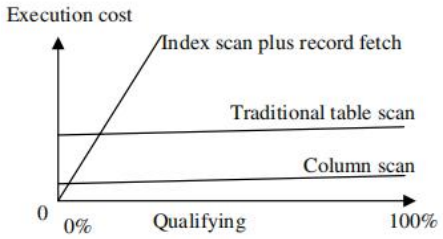


图5.5竞争计划的成本图。

其他传统计划使用辅助索引，通常是b树，以获取满足查询谓词的行的记录标识符。这个计划对于小的结果大小要优越得多，通常比表格扫描快1000倍。然而，对于较大的结果大小，传统的执行技术非常昂贵，因为每个限定行都需要一个随机的I/O。柱存储和柱扫描目前被认为优于传统的计划。它们非常健壮，但比表扫描速度快，它等于完整行的大小除以给定查询真正需要的列的组合大小。这个因素通常是一个数量级或更多的，特别是如果柱状存储格式受益于压缩，而传统的表格式则不能。

当然，理想的情况是一种技术，它对具有很少限定记录的谓词进行二次索引搜索，对具有许多限定记录的谓词进行列扫描，以及在两者之间的整个范围内优雅退化。

- 辅助索引可以比表扫描甚至列扫描更快地回答选择性查询。
- 除非在编译时查询优化过程中的基数估计非常准确，否则具有健壮性能的计划可能比具有更好的预期性能的计划更可取。

5.3覆盖索引

如果辅助索引可以提供查询中所需的所有列，或者提供特定表中的所有列，则不需要从堆或主索引中获取记录。这其中常用的术语是“仅限索引的检索”或“覆盖索引”。后一个术语可能会令人困惑，因为这种影响并不仅仅是由于索引的属性，而是由于索引和查询的组合。

图5.6显示了一个没有覆盖列的表到表导航的查询执行计划。查询导航多对多关系，在这种情况下，在课程和以注册为中间表的学生之间。具体来说，查询是

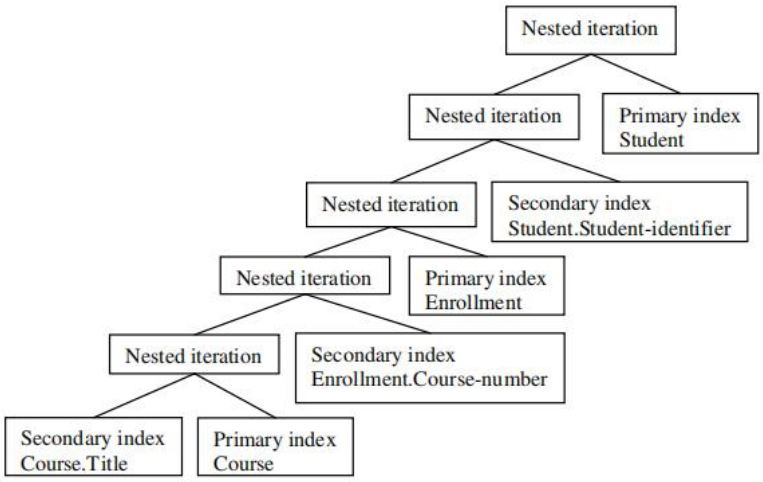


图5.6查询执行缺少覆盖索引。

从学生中选择学生名为学生，注册为学生，课程为学校。标题=为“生物学101”和enr。课程编号= co。课程编号和enr。学生标识符= st。学生标识符。”

这个三表查询需要两个双表连接。每个连接可能需要两个索引检索：首先，在辅助索引中的搜索生成一个行标识符；其次，在主索引中的搜索会产生额外需要的列值。在查询执行计划中，每个这些索引搜索都显示为嵌套的迭代。它的左侧输入提供外部输入；在嵌套迭代中，外部循环迭代来自左侧输入的项目。来自左侧输入的字段值被绑定为右侧输入的参数。绑定相关参数是查询执行计划中数据流到根目录的一个例外；这是在实现嵌套迭代时遇到的主要困难之一，特别是在并行查询执行[42]中。正确的子计划对每个不同的参数绑定执行一次。嵌套迭代操作中的内部循环遍历其正确输入的结果。

在图5.6中的每个实例中，正确的子计划是一个单个节点，一个索引搜索。第一个（最低的）嵌套迭代实现了对特定课程的搜索（“生物学101”）和所有必需的课程属性，特别是课程编号；其余四个嵌套迭代和索引搜索的实例实现了三表连接使用

非覆盖次级索引。图5.6显示了最坏的情况：学生表可能有一个主索引主要关键学生凭证，保存一个连接操作，和注册表可能有50%的机会，因为两个外键形成表的主键。

为了允许在更多的查询中只进行索引检索，一些系统允许向非搜索键一部分的索引定义添加列。仅使用索引的检索所导致的性能提高必须与额外的存储空间、扫描期间的带宽需求和更新期间的开销相平衡。主键和外键似乎是最有希望添加的列，因为它们往往在具有聚合和多表连接的查询中大量使用，使用小数据类型（如整数而不是字符串），并且稳定（使用存储值的罕见更新）。在针对商业智能的数据库中，向索引中添加日期列也可能很有用，因为时间在商业智能中通常是必不可少的。另一方面，出于同样的原因，许多查询在日期上都是有选择性的，因此更倾向于使用日期列作为搜索键而不是作为添加的列的索引。

图5.7显示了与图5和数据库相同的查询的查询执行计划，但具有覆盖索引的有益效果，i。.6e.，添加到每个辅助索引中的关键字列。这个计划代表了最好的情况，与学生标识符包括在sec-
注册时的边界索引，并将学生姓名添加到学生标识符上的索引中。

如果没有一个索引覆盖一个给定的查询，那么多个辅助索引可能会放在一起。通过在公共引用列上连接两个或多个这样的索引，可以使用查询所需的所有列来组装行。如果辅助索引中的记录大小之和为

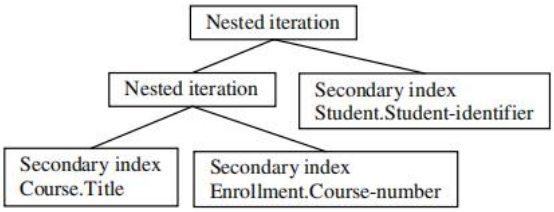


图5.7利用覆盖索引进行的查询执行。

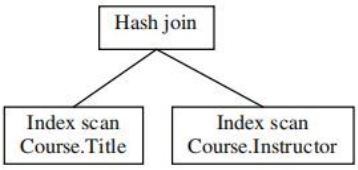


图5.8使用索引连接，以覆盖一个查询。

小于主索引中的记录大小，可以通过连接多个次级索引来减少I/O卷。如果连接算法消耗记录的速度就像扫描产生的记录一样快，那么e。g.，作为一个内存散列连接，这种技术需要的I/O比主索引的扫描要少，但牺牲了大量的内存需求。如果查询谓词减少了一个或多个次索引[52]的扫描，这种情况尤其可能。

.8图5显示了查询执行计划的一个片段，其中同一表的两个索引一起覆盖了一个查询。这里的查询是“选择标题，讲师从课程中的标题像‘生物学*’。”该计划利用了两个辅助索引，其中一个都不包含查询所需的所有列。连接谓词链接了这两个引用列。由于两个索引都按键值排序，而不是引用列，哈希连接是最快的连接方法，特别是由于从索引扫描到连接输出[52]的两个索引扫描产生非常不同的数据卷。如果有两个以上的索引可以覆盖查询，所有这些索引都连接在引用列上，那么“有趣的排序”就适用于基于排序的连接算法[115]，以及基于哈希的连接算法[52]中的“团队”。

如果多个索引处于相同的排序顺序，则一个简化的合并连接就足够了。这是柱状数据库的标准技术，每列按引用的顺序存储；引用在数据页上被压缩甚至省略。在传统的二级索引和柱状存储中，都可以压缩重复的列值，e。g.，通过运行长度编码。

早期的关系数据库管理系统没有利用覆盖索引；相反，它们总是从堆或主索引中获取记录。目前的一些系统仍然在做同样的事情。例如，

Postgres依赖于多版本的并发性控制。由于多版本并发控制的空间开销，e。g.，对于所有记录的时间戳，版本控制只在堆中实现，而不在辅助索引中实现。因此，为了进行并发控制，在辅助索引中的任何搜索之后都必须获取记录。

- 如果辅助索引包含查询所需的所有列，则不需要访问表的主存储结构（仅使用索引进行检索）。
- 在某些系统中，辅助索引可能包括既不是索引键也不引用主索引的列。包含在次要索引中的主要候选项是主键列和外键列。进一步的候选项包括在谓词中经常使用但很少更新的列，e。g.，日期

5.4索引到索引导航

如果仅索引检索不够，各种技术可以基于从另一个索引中提取的值加快获取b树索引中的记录。e.，用于从一个b树索引“导航”到另一个索引，以及从一个索引中的一个记录导航”到另一个索引中的另一个记录。这些技术主要基于两种方法，异步预取和排序。这两种方法都单独或一起使用，并且在许多变体中。

异步预取可以应用于所有单独的b树页面，包括叶页，或者只能应用于内部页面。例如，在索引嵌套循环连接之前，b型树的上层可能会被读入到缓冲池中。回想一下，上层通常占b树的不到1%，低于数据库服务器中内存大小和磁盘大小的2-3%。页面可以被固定在缓冲池中以防止它们被替换，或者根据LRU等标准缓冲池替换策略保留或替换。小表及其索引，e。g.，带有星形模式[77]的关系数据仓库中的维度表，甚至可以被映射到红砖产品[33]中的虚拟内存中。在这种情况下，可以使用

内存地址，而不是需要在缓冲区池中进行搜索的页面标识符。

通过异步预取，可以减少等待叶页的等待。有些系统填充一小组不完整的搜索，然后将它们传递给存储层[35]；其他系统在[34]时刻保持固定数量的不完整搜索。如果大部分甚至所有预取提示导致缓冲池中的命中，则可以抑制进一步的提示，以避免处理无用提示的开销。

除了异步预取外，对中间搜索信息进行排序通常是值得的。e.，按照与b树相同的顺序对搜索键集进行排序。在极端情况下，它可以折叠对同一叶页的预取请求。在许多情况下，对未解决的引用进行排序使存储层能够将许多小的读操作转换为更少、更大、从而更高效的读操作。

图5.9说明了在一个索引（通常是次要索引）中获得的引用如何在另一个索引（通常是主索引）中解析之前进行预处理。在查询执行计划中，引用集是一个中间结果——图5.9显示了一些可能是主索引中的关键值的数字。在次级索引中，以最方便或最有效的方式获得参考值，而没有有助于下一个处理步骤的特定排序顺序。对这些引用进行排序可以使用任何常用的算法和性能技术。压缩可能会影响精度；例如，当前三个特定的引用被折叠到一个范围内时，特定的信息将丢失，并且在获取它们之后，必须对此范围内的所有记录重复谓词计算。在实现中，排序和压缩可以以各种方式交错，就像重复消除可以集成到运行生成和合并到外部合并排序[16,39]中一样。

	中间结果
未排序排序 压缩	34，42，98，26，43，57，29 26，29，34，42，43，57，98 26-34，42-43，57，98

图5.9预处理预取提示。

一组经过排序的搜索键允许在b树中有效地导航。最重要的是，每个页面只需要一次，分页持续时间较短，分支节点持续时间较长。一旦搜索移动到具有更高键的节点邻居，节点可以在缓冲池中丢弃；不需要使用LRU等启发式替换策略来保留它。因此，缓冲区池只需要为每个b树级提供一个页面，以及适当的预存帧来进行预取和预读。考虑到大多数b树都有一个大的扇形，因此有一个非常小的高度，将当前根到叶路径上的所有页面固定在缓冲池中是合理的。此外，如果b树的每个级别中到目前为止读取的最高键被缓存（通常与b树被“打开”以通过查询执行计划访问时创建的数据结构一起），则不需要每个单独的搜索都从根页面开始。相反，每个搜索都可以按叶到根的顺序尝试b树页面。关于每个页面最近更新的信息，e。g.，Ariss日志序列号[95]，可以确保并发更新不会干扰有效的b树检索。b树节点[44]中的栅栏键可以确保范围比较的准确性。程等人。[25]描述了一种稍微一般一些的技术，用于利用磁盘页中的父指针，而不是固定在缓冲池中。

图5.10说明了图2.4中的B树的部分内容，以及在下次搜索过程中能够在B树中进行高效导航的缓存。图5.10的右半部分显示了缓冲区池中的数据页；图5.10的左半部分显示了与扫描或查询操作相关联的部分状态。如果最后一个搜索键是17，则缓存包含指向所显示的三个节点的缓冲池的指针和

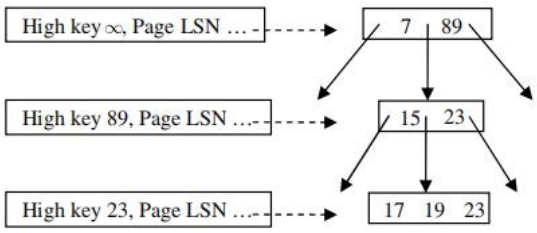


图5．10在预排序的索引抓取中的缓存状态。

高压键和每个节点的页面LSN。如果下一个搜索与键值19有关，则缓存立即有用，最好是在叶级开始搜索。如果下一个搜索与键值47有关，则必须放弃当前叶，但它的直接父叶仍然有用。父节点的高键是89，这是从根节点中的分隔符键学习到的。如果下一个搜索键与键值99有关，则必须同时放弃叶和它的父节点，并且只有这个b树的缓存根节点才有用。每个新搜索的最佳起始级别可以通过每个b树级别的单一比较来找到。

如果对b树结构的修改会导致b树页面的去分配，这有几个困难。例如，如果所显示的叶节点由于删除和下流而合并到相邻的叶中，则必须确保搜索不依赖于缓冲区池中的无效页面。这个问题可以通过多种方式来解决，例如通过验证沿着根到叶路径的所有页面lsn。

为了启用这些优化，搜索键可能需要在b树查找之前进行显式排序操作。在一种搜索键中，昂贵的临时文件可以通过三种方式来避免。第一种方法是进行较大的内存分配。第二种方法是只使用部分排序的搜索键来执行运行生成和调用搜索。这可能被称为机会主义排序，因为它会机会主义地利用现成的内存，并在其输出中创建尽可能多的排序。

第三种方法使用了一个位图来进行压缩和排序。该方法只适用于在位图中的搜索参数和位置之间存在足够密集的双向映射。此外，这个映射必须保持顺序，这样扫描位图就会产生已排序的搜索键。最后，这适用于从辅助索引到同一表的主索引的导航，但如果必须保存在早期查询操作中获得的任何信息并在查询执行计划中传播，那么它就不起作用。

图5.11使用位图继续使用图5.9。为了简洁起见，该插图采用了十六进制编码，用每个4位数字或16位的块的前缀0x表示。鉴于位图通常包含许多内容

	中间结果
压缩的 位图（十六进制） 压缩位图混合表示	26-34, 42-43, 57, 98 0x0000, 0x0020, 0x2030, … 3×0x00, 2×0x20, 1×0x30, … 3×0x00, 0x2020, 0x30, 57, 98

图5．11.针对预取提示的位图压缩。

零，相似字节的运行长度编码是一种标准技术，至少对于包含所有零的字节是这样的。最后，最压缩和最简洁的无损表示法将运行长度编码、显式位图和显式值列表交织起来。此外，为了避免外部排序，表示可以使压缩技术无信息丢失。

各种压缩方法可以结合起来。其目标是减少大小，以便可以将整个引用集保留并在内存中进行排序，而无需进行对存储层次结构中较低级别的昂贵访问。如果压缩为某些关键范围引入了任何信息丢失，则在b树访问期间必须扫描该范围内的所有b树条目，包括重新评估适当的查询谓词。换句话说，在受影响的键范围内重复b树搜索变成了范围扫描。图5.9显示了一个示例。如果存在分布倾斜的问题，那么直方图（通常用于查询优化）可以帮助选择最有希望进行压缩的搜索键。如果除了引用之外的信息还必须在查询执行计划中随身携带，那么i。e.，如果索引的目的是覆盖给定的查询，那么只适用无损压缩技术。

对搜索键集进行排序可以提高查询执行性能[30,96]，也可以提高性能[52]的健壮性。如果查询优化过程中的基数估计非常不准确，甚至可能是几个数量级，那么将搜索键排序到一个排序流中，并优化b树导航，确保索引索引导航的性能不会不比表或索引扫描差。

图5.12显示了图5.5中所示的相同的两个传统计划的性能，以及一个稳健计划的成本。对于较小的查询结果，对从辅助索引获得的引用进行排序几乎不会影响执行成本。对于大型查询结果，请对其进行排序

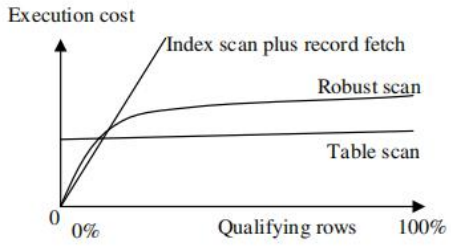


图5．12索引扫描和表扫描之间的鲁棒转换。

引用确保成本不会超过表扫描的成本。不同之处在于从辅助索引中获取和排序引用。如果次索引中的记录大小是主索引中的记录大小的一部分，则次索引中的扫描成本是主索引的扫描成本的相同比例。适当的压缩确保排序操作可以依赖于内部排序算法，并且不会对总体查询执行成本贡献I/O操作。

因此，图5.12说明了一个健壮的索引-索引导航的原型示例。类似的技术也适用于连接多个表的复杂查询，而不仅仅是在同一表的多个索引之间的导航。

如果使用一个次要索引来提供一个“有趣的排序”，那么[115]，e。g.，对于后续的合并连接，如果辅助索引不是表和查询的覆盖索引，则合并连接必须在获取完整记录之前进行合并连接，或者获取操作不能对引用进行排序，从而放弃有效检索，或者获取的记录必须排序回从辅助索引获得的顺序。后一种方法基本上需要另一种排序操作。

加快第二种排序操作的一个想法是在第一次排序之前用序列号标记记录，这样第二种排序就可以有效地进行，并且独立于类型、排序序列等。在原始的次要索引中。另一方面，这个排序操作必须管理比连接之前的排序操作更大的记录，并且它不能从压缩中获益，特别是从压缩中获益

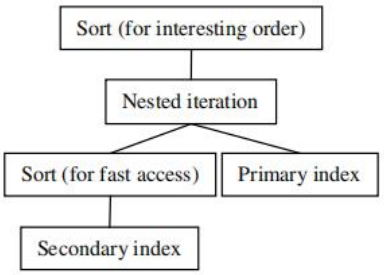


图5．13为一个强大的连接和“返回”排序。

压缩与信息丢失。因此，即使初始排序操作不是外部排序，此排序操作也可能是外部排序。

图5.13显示了查询执行计划的一个片段，其中这种技术可能很有益。辅助索引以“有趣”的排序顺序提供记录，例如合并连接（图5.13中没有显示），然而，对于从主索引进行健壮的检索，需要另一种排序顺序。通过适当的技术，可以通过利用辅助索引中的排序顺序来改进最终排序的性能。

为了支持多个表之间的索引到索引导航，一个很有前途的启发式方法是在所有辅助索引中包含所有关键列，包括主键列和外键列。图5.6和图5.7中的查询执行计划之间的差异说明了其有益的效果。该技术可以通过包含基于函数依赖关系和外键约束的其他表的列来进行扩展。例如，订单详细信息表的日期列上的索引可能包含客户标识符；订单详细信息表通常不包括此列，但在订单表中存在适当的功能依赖关系。

- 如果需要用许多搜索键来探测一个索引，那么对这些键进行排序可以避免重复搜索，并可能提高缓冲池的有效性，从而提高I/O成本。
- 索引搜索可能在前面搜索结束的地方恢复，导致b树索引的向上和向下导航。

- 这些优化支持了健壮的查询执行计划，从仅基于辅助索引获取少量项到完全扫描主索引，从而降低了在编译时查询优化过程中由于错误的基数估计而导致的惊人性能。
- 优化扫描之间的盈亏平衡点(e。g.，在压缩的列存储中)，优化的索引检索可能会从传统的高延迟磁盘存储转向半导体存储，如闪存。

5.5利用关键前缀

到目前为止，大多数讨论都假设在b树中的搜索从一个完整的键i开始。e.，B树键中所有字段的特定值。搜索的输出是与一个键，e相关联的信息。g.，指向主索引或堆文件中的记录的指针。由于所有的b树都是唯一的，如果有必要，通过向用户定义的键添加一些东西，这样的b树搜索最多会产生一个匹配的记录。虽然一些索引只支持使用整个索引键的精确值进行搜索，特别是哈希索引，但b-树要一般得多。

最重要的是，b树支持对关键值在给定的下界和上界之间的记录的范围查询。这不仅适用于键中的整个字段，还适用于列值中的前缀，特别是字符串值。例如，如果一个b树键字段包含像“Smith，John”这样的名称，那么对姓氏为“史密斯”的所有人的查询实际上是一个范围查询，对以“Sm”或“史密斯，J”开头的名称的查询也是如此。

人们通常认为，只有当查询中的搜索键指定了b树键的严格前缀时，b树才有用，但事实并非如此。例如，在按邮政编码（邮政编码）和（族）名称组织的b树中，查找所有具有特定名称的人需要枚举b树中所有不同的邮政编码值，并为每个邮政编码搜索指定的名称一次。邮政编码的枚举可以在搜索之前，或者两者可以交错[82]。此外，在一个邮政编码内的搜索完成后，将进行下一步操作

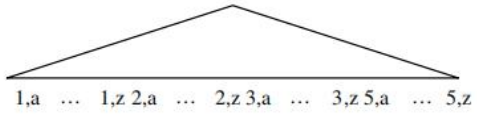


图5．14双柱b树索引。

可以使用b树索引搜索下一个邮政编码，或者它可以简单地添加一个到之前的邮政编码，并尝试找到递增的邮政编码和所需的名称的组合。如果由于增加的邮政编码不存在而尝试失败，那么b树中的根到叶搜索将导致在邮政编码列中包含下一个实际值的记录。

图5.14显示了一个带有两列键的B-树索引。对于只指定字母列而不指定数字列的查询，e。g.，“其中字母= ‘q’，”枚举数值是必需的。b树的第一个根到叶遍历找到数字1或实际上是组合键“1，a”；第一个搜索搜索一个组合键“1，q”。下一个从根到叶的遍历可以通过查找数字大于1的b树条目来查找组合键“2，a”中的数字2。在改进的方案中，数字2不是在搜索中找到的，而是从之前的值1计算出来的，下一个根到叶搜索立即集中于组合键“2，q”。这很适合于数值2和3。在搜索组合键“4，q”时，从根到叶的遍历没有找到适当的记录，但它会找到一个数值为5的记录。因此，下一个搜索可能会直接搜索组合键“5，q”。

如果不同值的数量相当小（上面示例中的邮政编码或图5.14中的数字值），这种技术似乎最有希望。这是因为每个不同的前导b树键值都需要一个随机的I/O来获取一个b树的叶子。然而，不同值的数量并不是真正的标准。另一种查询执行计划通常是使用大的顺序I/O操作扫描整个索引。如果每个不同的领先b树键值的数据量太大，以致于扫描比单个探针需要的时间更长，则探测计划比扫描计划快。请注意，这种效率比较不仅必须包括I/O，还必须包括谓词评估的努力。

除了任何邮政编码区域中的特定名称的前导b树键外，前导列还可能受到范围谓词而不是特定值的限制。例如，邮政编码可能被限制为“537”的开头，而不考虑最后两个数字。此外，该技术还适用于两个以上的列：第一列可以限制为特定值，第二列未指定，第三列限制为范围，第四列未指定，第五列再次限制为特定值。对这些谓词的分析可能是复杂的，但b树似乎比其他索引结构更支持这些谓词。也许最复杂的方面是在全索引扫描、范围扫描和选择性探针之间进行基于成本的编译时决策所需的成本分析。动态运行时策略可能对基数估计误差、成本估计误差、数据偏差等是最有效和鲁棒的。

在具有由多个字段组成的键的b树索引中，单个字段可以被认为是形成一个层次结构或表示多维空间中的维度。因此，一个多列b树可以看作是一种多维访问方法（MDAM）[82]。如果在查询谓词中指定，则必须枚举其可能的值。如果维数在整数坐标中测量，并且领先维数中不同值的数量，则该策略很有效。通过按照b树的排序顺序将查询谓词转换为适当的范围查询，即使是复杂的分离和连接也可以相当有效地处理[82]。

图5.10中所示的技术支持了从根到叶的遍历探测b树和按索引顺序扫描b树之间的动态和自动转换。在一些最近的请求可以满足当前叶或它的直接邻居，异步

基于父节点和祖父节点的后续叶节点的预读取似乎很有希望。如果叶节点的重用很少，但父节点及其近邻居在连续的搜索请求中很有用，那么叶节点可能会迅速从缓冲池中逐出，但异步预读可能会应用于父节点。

- 化合物(i。e., 多列)b树索引支持其列的任何前缀上的相等谓词。
- 通过对谓词的巧妙分析，b树还支持其列的子集上的范围和相等式谓词的混合，甚至包括对前导列没有限制的情况。

5.6订购检索

就像b树可以提供比通常实现的更多类型的搜索类型一样，它们也可以在扫描的输出中支持更多类型的排序顺序。回到之前将b树作为排序操作的缓存状态的想法，最明显的排序顺序是将b树键作为排序键。在大多数实现中，升序和降序都同样得到了良好的支持。还支持b树键前缀上的任何排序顺序。

如果所需的排序键开始B树的前缀，但也包括B树列但不是在b树键的一个有用的方式，那么它可以有利于执行排序操作为每个不同的值的有用的前缀列B树。例如，如果一个表中有列A，B，C，。..，X、Y、Z存储在列A、B、C的主要索引，如果一个查询需要输出排序a、B、P、Q，那么可以利用A，B的排序顺序和P，Q的A，B的不同值记录。理想情况下，这个小排序操作序列可以比单个大型排序操作快，例如，因为这些小排序操作可以在内存中执行，而单个大型排序需要在磁盘上运行文件。在这种情况下，这个执行策略需要（大部分）大型的运行生成工作，而不是其合并工作。

图5.15使用虚线括号说明了这些单独的小型排序操作。单个段的大小可能会有很大的不同，因此一些但不是所有的段可能适合可用的内存分配。

另一方面，如果所需的排序顺序省略了前缀，但与b树键匹配，则省略的不同值

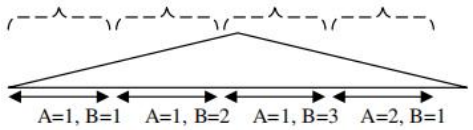


图5．15对不同的A，B排序一次。

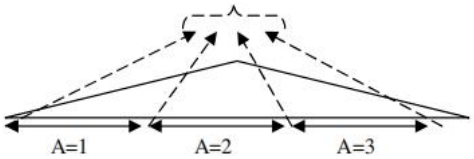


图5．16通过合并由A的不同值标识的运行来进行排序。

前缀列可以被认为是外部合并排序中运行的标识符，并且可以通过合并这些运行来产生所需的排序顺序。使用相同的示例表和主索引，如果一个查询需要在BC上排序的输出，那么B树可以被认为是在BC上排序的许多运行，而a的不同值可以识别这些运行。这种执行策略需要大规模的合并工作，但不需要其运行生成工作。

图5.16说明了合并步骤，即为A的每个单独值扫描A、B、C上的主索引一次，然后合并由A的不同值标识的运行。如果A的不同值的计数小于最大合并扇入值（由内存分配和I/O的单位决定），则单个合并步骤就足够了。因此，所有记录都被读取一次，在经过合并逻辑后，为下一步运行生成、在磁盘上运行文件等做好准备。

这两种技术可以结合起来。例如，如果一个查询请求输出按A、C排序，那么A的不同值将启用多个小排序操作，每个操作合并运行在C上排序，并由B的不同值标识。请注意，B的不同值的数量可能会随着A的不同值而有所不同。因此，小的排序操作可能需要不同的合并策略。因此，合并规划和优化比在标准排序操作需要更多的工作。这两种技术的更复杂的组合也有可能实现。

排序输出不仅需要在查询文本中的显式请求下，还需要许多连接、设置操作如交集、分组聚合等。对于最重要、最紧急或最相关数据项的“顶级”查询受益于上面描述的排序和合并策略，因为第一个输出比大型排序操作更快，大型排序操作在产生第一个输出之前消耗整个输入。如果内部查询使用基于顺序的算法，如合并连接，则类似的考虑也适用于具有“存在”子句的查询。

- 除了任何前缀之外，b树还可以使用合并或分段执行在其列列表的许多其他变体上产生排序输出。
- 这些技术帮助许多基于顺序的查询评估算法，并应用于任何预排序的数据来源，而不仅仅是b树。

5.7单个表的多个索引

上面讨论的大多数查询执行策略对每个表最多利用一个索引。此外，还有许多策略利用多个索引，例如针对多个谓词。

对于两个或多个谓词子句的连接，每个谓词子句匹配一个不同的索引，标准策略从每个索引中获得一个引用列表，然后与这些列表相交。对于分离，需要进行列表的结合。被否定的子句要求进行差异操作。

图5.17显示了一个典型的查询执行计划，它是通过交叉引用列表执行的

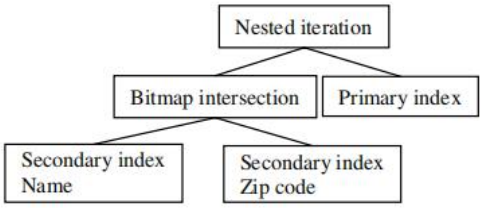


图5．17个连词谓词的索引交集。

从同一个表的多个索引中获得。如果索引不是位图索引，则每个辅助索引的结果列表将转换为位图，计算它们的交集，并使用索引嵌套循环连接为表获取完整的行。

交集等设置操作可以用标准连接算法如哈希连接或位图来处理。如果位图没有被压缩，则可以利用“二进制和和”等硬件指令。如果位图使用运行长度编码或其变体进行压缩，适当的合并逻辑使设置操作无需解压。或者，可以使用一个位图来处理分离和否定，使用多个索引扫描设置或清除位。

另一种类型的多索引策略利用同一个表上的多个索引来实现覆盖索引的性能优势，即使不存在单一覆盖索引的情况。用少列扫描和连接两个或多个辅助索引，因此短记录可以比扫描有长记录的主索引或堆更快，这取决于连接操作的效率和足够内存[52]的可用性。图5.3节中的图5.8显示了一个包含两个索引的查询执行计划。它与图5.7中的查询执行计划不同，因为它避免了嵌套的迭代来查找行和连接，而不是通过交集操作。

在某些连接查询中，最佳的计划会为每个表使用多个索引。例如，辅助指标可以用于半连接缩减[14]。其目标是减少不仅要通过单表选择谓词以及通过多表连接谓词获取的行列表。因此，首先根据查询中的连接谓词连接来自不同表的辅助索引。当然，这是假定这些辅助索引中的列覆盖了连接谓词。然后从两个表的主索引中获取行的成本只针对真正满足连接谓词的行发生。

图5.18显示了使用半连接缩减连接两个表的查询执行计划。图5.18中的查询执行计划的基本方面是，在从表中获取行之前，将计算连接谓词。表（或其索引）所在的排列顺序

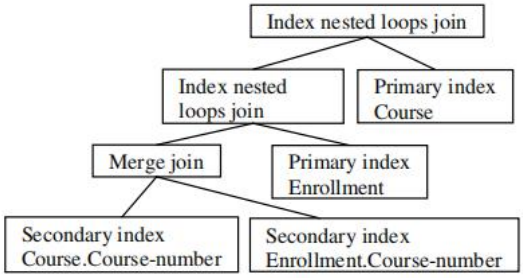


图5．18双表半连接减少。

连接的方式与访问表以获取缺失的列值的顺序无关。在这个具体的例子中，可以在数据库中定义参考完整性约束（课程仅存在于注册学生中，注册学生必须参考现有的课程）。如果是这样，半连接减少可能是无效的，除非图中没有显示附加谓词，e。g.，对注册表中的课程名称或成绩，使参考完整性约束无效。

该技术适用于具有多个表的复杂连接，并且已被证明对具有一个特别大的表的复杂连接特别有价值，该表也被称为关系数据仓库中的“星形模式”中的中心“事实表”。在此类数据库上的典型“星形查询”中，谓词通常应用于事实表周围的“维度表”，但查询的执行成本主要由对大型事实表的访问决定。在这种“星连接”的标准查询优化技术中，维度表首先与事实表的辅助索引连接，事实表中的行标识符列表合并成一个满足查询中所有谓词的行列表，只将事实表中的最小行集作为查询执行计划的最后一步。

图5.19显示了同时使用索引交集和半联接缩减的星型联接的查询执行计划。原始查询包括对两个维度“客户”的谓词。g.，″城市=大天空”）和“部分”（e。g.，″部分名称为=，名为“船锚”）。请注意，星号查询通常涉及的相关性远远超出了数据库统计数据和元数据的语义能力。例如，很少有客户

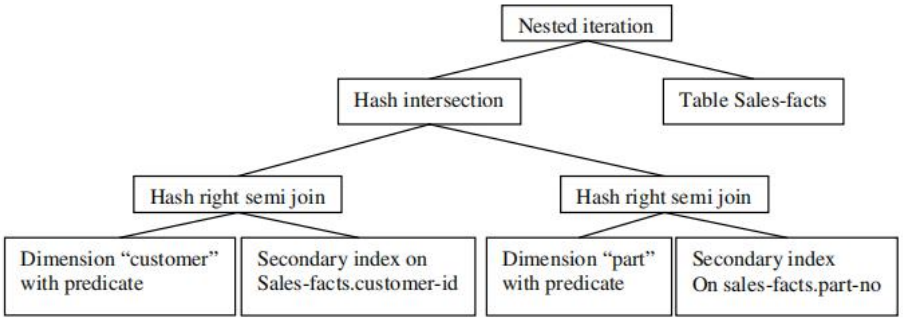


图5．19个星形连接使用交叉点和半连接减少。

将购买高山上的船锚。在图5.19的查询执行计划中，两个初始连接操作中的每一个都在维度表和事实表的适当辅助索引之间执行半连接。结果是事实表中的一组引用。交集操作会在事实表中为满足两个维度上的谓词的行生成引用。最后的右半连接产生这些事实行。如果希望保留维表中的列，则适当的初始半连接和最终的半连接必须是内部连接。

一个通常既不被理解也不被考虑的相关问题是同一表的多个索引之间的并发控制，特别是在选择了弱事务隔离级别时。如果在多个索引中执行谓词评估，但并发控制无法确保多个索引中匹配的索引条目一致，则查询结果可能包括无法满足整个查询谓词的行，也可能无法包含确实满足谓词的行。

此问题也适用于跨多个表的谓词，但可能依赖于弱事务隔离级别的用户更有可能期望表之间的弱一致性，而不是在单个表的单个行内的弱一致性。实体化视图及其底层表之间也存在此问题，即使增量视图维护是每个更新事务的一部分。

- 索引交集可以加速连接谓词，是常用的。索引联合可以支持分离和分离

指数差异可以支持“而不是”谓词。位图可以能够有效地实现这些操作。

- 在仅使用索引检索的特殊情况下，多个索引一起可能覆盖一个查询。
- 在具有星型或雪花型模式的数据仓库中，可能需要使用复杂的索引，以获得最佳性能。

.85个单个索引中的多个表

由于连接操作传统上是最昂贵的数据库查询操作之一，而且大多数连接都是使用外键和主键上的相同相等谓词指定的，因此有两种技术建议自己将多个表组合在单个索引中。

首先，连接索引将连接列的值映射到两个（或更多）表[66,123]中的引用。这些引用可以用列表或位图[105]来表示。扫描连接索引所产生的结果类似于在各自的连接列上连接两个辅助索引的结果。

.20图5说明了Hrder提出的3个表的组合图像，包括一个键值、每个表的出现次数计数器，以及表中记录标识符的适当数量。不要求每个表中出现每个键值；计数器可能为零。

一个“星形索引”的[33]非常类似于组合的图像和连接索引。主要的区别在于，星形索引包含了维度表中的记录引用，而不是键值。换句话说，星形索引映射适当维度表中的记录引用，以记录事实表中的引用。如果星型索引是b树，则星型索引记录中的维度表的顺序很重要，其中所述维度表的聚类效果最好

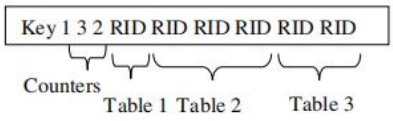


图5.20由Hrder提出的组合图像。

第一如果所有维度表中的记录引用都是列值，则e。g.，对于维度表的主键或聚类键，则星形索引实际上与事实表中相应的外键值上的多列b树索引相同。

第二，合并后的索引将在第7.3节中进行讨论。

- 连接索引和星形索引将多个辅助索引组合成一个b树，以加速大型连接。

.95个嵌套的查询和嵌套的迭代操作

[1]数据库和磁盘大小继续快速增长，而磁盘访问性能和磁盘带宽的提高要慢得多。如果没有其他原因，对数据库查询处理的研究必须重新关注那些不是与数据库大小线性增长，而是与查询结果大小线性增长的算法。这些算法和查询执行计划非常依赖于索引导航，i。e.，它们从查询谓词中给定的一个常量开始，在索引中找到一些记录，从这些记录中提取进一步的列值，在另一个索引中找到更多的记录，以此类推。这种类型的查询计划的成本与所涉及的记录数量呈线性增长，这很可能意味着该成本有效地独立于数据库大小。实际上，传统b树索引的索引查找成本随着数据库大小呈对数增长，这意味着当表从1000到1000000条记录增长时，成本会翻倍，从100000000000000条记录会翻倍。成本几乎没有从100万条变化到200万条记录，而排序或散列操作的成本翻了一番。　　此外，众所周知，扫描可以“清除”所有有用页面的I/O缓冲区，除非替换策略被编程为识别扫描[26,91,120]。然而，CPU缓存及其替换策略识别扫描并不可能；因此，大型扫描将反复清除所有CPU缓存，甚至是多兆字节的2级和3级缓存。基于这些行为，根据经济增长的增长率磁盘大小和磁盘带宽，以及最近在主流关系数据库系统中添加的物化和索引视图，

[1] 本节中的一部分来自于[42]。

我们应该期待基于索引的查询执行会强劲复苏，因此研究对严重依赖嵌套迭代的执行计划的兴趣。访问延迟比传统磁盘驱动器快100倍，将加速这一趋势。

在在线事务处理（OLTP）和在线分析处理（OLAP）中，交互式响应时间的关键是确保查询结果被获取，而不是搜索和计算。例如，如果OLAP产品能够缓存以前和将来可能出现的查询的结果，那么它们的性能就会很好。在关系数据库系统中，直接获取查询结果意味着索引搜索。如果一个结果需要来自多个索引的记录，那么索引嵌套循环连接，或者更一般地说，嵌套迭代是选择[63]的算法。

图5.21显示了三个表的简单连接的查询评估计划，嵌套迭代的“外部”输入显示为左侧输入。嵌套迭代是简单的笛卡尔积，因为连接谓词已经被下推到内部查询计划中。如果过滤器操作实际上是索引搜索，并且如果表T0相对于T1和T2很小，那么这个计划可以比任何使用合并连接或哈希连接的计划更有效。备选方案使用嵌套迭代的多个分支，而不是图5.21中的多个级别。当然，复杂的计划可以在任何层次上结合多个层次和多个分支。

嵌套循环连接是嵌套迭代的最简单形式。内部查询的复杂性没有限制。它可能需要搜索一个辅助索引，然后从主节点中获取

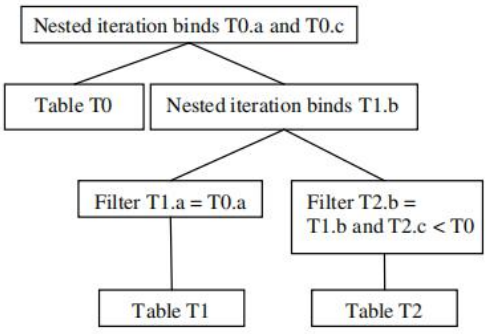


图5.21具有嵌套迭代的查询计划。

表；使用位图、排序或哈希操作相交两个或多个辅助索引；每个表都有自己的访问路径；多个层次的嵌套迭代，每个层次有多个子分支；等等。除了最琐碎的案例外，所有问题都有复杂的问题需要在商业产品和完整的研究原型中解决。这些问题存在于策略和机制的级别上，包括避免I/O或利用异步I/O，以及管理内存和线程等资源。

除了前面描述的异步预取、在抓取行之前对引用进行排序等技术之外，b树索引还可以在缓存内部查询的结果方面发挥重要作用。从概念上讲，人们可能希望使用两个索引，一个用于以前遇到的外部绑定值（内部查询的参数），另一个用于内部查询的结果。后者可能包含许多外部绑定的许多记录，因此可能会变得相当大。前者还可能包括结果大小以及使用频率或最近时间。关于空结果的信息可能会防止重复无效的搜索，而关于实际结果大小和使用情况的信息可能会指导替换决策，e。g.，为了在CPU缓存或缓冲池中保留最有价值的内部结果。这个索引甚至可以用来指导下一步处理哪个外部绑定，e。g.，立即为具有空的内部结果的外部绑定生成结果。注意，这两个索引可能保留在一个合并的b树中，这样将缓存划分为两个索引的开销就会最小化。

- 在基于扫描的查询处理中，查询执行时间随着表或数据库大小的增加而增长。在基于索引到索引导航的查询处理中，查询执行时间随着中间结果大小的增加而增长，并且几乎不受数据库或表大小的影响。
- 嵌套循环连接是嵌套迭代的最简单形式。通常，内部查询执行计划可以是复杂的，包括多个嵌套迭代。
- 对于一般嵌套迭代，排序与对于索引嵌套循环连接一样有帮助。

- 如果内部查询访问的表非常小或有索引，那么嵌套迭代可以非常有效。

5.10更新计划

为了启用这些各种基于索引的查询执行策略，索引必须存在，并且必须是最新的。因此，高效的索引创建和索引维护的问题是对查询处理的必要补充。在下面的讨论中，更新包括对现有记录、插入和删除的修改。

已经设计了各种有效的索引维护策略。这些更改包括在修改b树中的记录和页面之前对更改进行排序，将现有记录的每次修改分割为删除和插入，以及在逐行和逐索引更新之间进行选择。逐行技术是传统的算法。当一个具有多个索引的表中的多行发生更改时，此策略将计算增量行（包括旧值和新值），然后逐个应用它们。对于每个增量行，在处理下一个增量行之前，将更新所有索引。

逐索引维护策略首先将所有应用于主索引的更改。增量行可以在与主索引相同的列上进行排序。所期望的效果类似于在从索引到索引的导航过程中对一组搜索键进行排序。当然，只有在逐个索引维护中才能对每个索引的增量行分别排序。

如果每个索引中有比页面更多的更改，这种策略特别有益。对每个索引的更改进行排序可以确保每个索引页最多需要读写一次。与只读查询一样，可以使用单个页面的预取或大型预读功能；此外，更新也受益于后写。如果预读和后写将单个页面的大序列作为单个操作传输，那么如果在索引扫描期间更改的数量超过此类传输的数量，则此策略是有益的。

图5。22显示了查询执行计划的部分，特别是在更新语句中与二级索引维护相关的部分。下面没有显示的滑轴操作是用来计算的查询计划

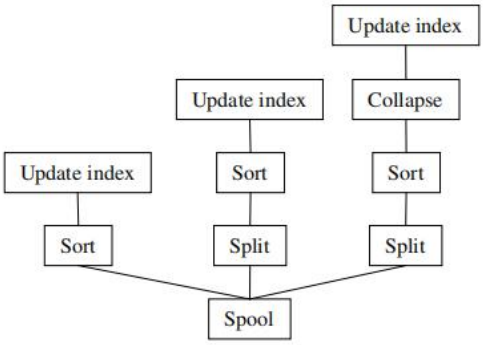


图5.22、优化后的指标维护计划。

要应用于表及其索引的增量。在左侧分支中，不修改索引的搜索键中的列。因此，只要优化将更改应用于现有索引条目的顺序就足够了。在中心分支中，将修改搜索键中的一个或多个列。因此，索引条目可以在索引内移动，或者，将更新分割为删除和插入操作。在右侧分支中，将更新唯一索引中的搜索键列。因此，索引中每个搜索键最多可以有一个删除和一个插入，匹配的删除和插入项可以折叠到单个更新项中，这可能保存根到叶的b树遍历和日志卷。尽管索引之间存在差异，以及它们如何受到更新语句的影响，但它们的维护受益于排序，理想的是数据驱动的排序操作。

按索引进行索引维护的实现可能使用带有多个操作的计划，这与查询执行计划非常相似。辅助索引的维护操作可以形成一个长链，更改行从一个传递到下一个，或者更改行可以是通常使用中间存储实现的共享中间结果。这两个更新计划都会不断地修改辅助索引。长链可能看起来更容易调度，但它要求早期的排序操作携带后续索引更新中所需的所有字段。在图5.22所示的更新计划中，避免了这种情况，但代价是编写和读取包含线轴操作中包含所有列的更改集。

为了提高效率，可以避免滑阀操作，或者更准确地说，可以集成到排序操作中。具体来说，在这个计划片段底部的单个排序操作可以接收输入数据，将这些记录保存在其工作区中，然后为所有三个分支生成运行。对于每个分支，排序操作必须创建一个数组，每个记录在工作区中有一个或两个代理。对于在图5.22中显示拆分操作的分支，需要两个代理。这个多功能排序操作写入与图5.22中的多重排序操作在磁盘上运行的完全相同的记录。节省下来的费用是编写和读取卷轴操作中的所有列，并将数据复制到多个排序操作的工作区中；缺点是，在运行生成期间，所有列都出现在工作区中，导致更多、更小的运行，并可能导致额外的合并步骤。

在这个更新计划中，排序操作争夺资源，特别是内存，这些操作计算更改集并准备它，可能是对它进行排序，以便更新主索引。因此，如图5.22所示的排序操作中的中间运行可能比集成的排序操作更有效。如果内存中的工作区足以对整个更改集进行排序，而不需要临时运行，那么该算法的另一种变体似乎是可取的。因此，数据库系统在为索引表和实例化视图的大型更新而实现的各种优化方面存在差异也就不足为奇了。

另一种通过索引到索引维护来实现的技术是针对万圣节问题[90]的永久索引保护。一般来说，在同一计划阶段搜索和更新索引可能会导致不正确的更新，甚至是无限循环。由于排序操作创建了单独的计划阶段，因此可以利用一个或多个辅助索引来计算更改行，并可以将这些更改立即应用到主索引中。

最后，逐索引维护和更新计划的基本思想也适用于外键约束、物化视图及其索引，甚至是触发器。具体来说，外键约束可以使用更新计划中的分支来验证，该分支非常类似于对单个次级索引的维护，以及对的信息的级联

外键约束和实体化视图可以在其他分支中实现。在复杂数据库中创建计划本身就是一项挑战；为了对b树索引进行调查，我们足以认识到在一些数据库管理系统中已经实现了各种非常有效的索引维护技术。

- 在只读查询执行期间，索引维护可以像索引到索引导航一样得到优化。主要的选择是逐行索引维护或索引索引。类似的选择也适用于检查完整性约束。
- 对每个索引的排序更新可以加快索引的维护速度。由于万圣节问题的分离，排序也可以提供保护。

5.11个已分区的表和索引

分区将一个表划分为多个数据结构。在分区的基本形式中，水平分区定义了行或记录的子集，而垂直分区定义了列或字段的子集。在每种形式的划分中，这些子集通常是不相交的。换句话说，分区不会引入冗余。

在分区表或索引中，每个分区都保存在自己的数据结构中，通常是b树。每个分区及其b树都在数据库目录中注册。由于分区的创建和删除必须记录在数据库目录中，因此在目录中需要适当的权限和并发性控制。相比之下，一个分区的[43]树会在单个[43]树中保留多个水平分区。b树条目属于每个记录中由人工前导关键字字段标识的分区。只要插入或删除适当的记录及其键值，就可以创建和删除分区。

水平分区可用于可管理性和作为索引的一种粗糙形式。可管理性可以通过对一个时间属性进行分区来提高，以便加载和清除操作

在滚动和推出时，请始终修改整个分区。作为一种索引形式，分区将检索定向到表的子集。如果大多数查询需要扫描与分区相同的数据，那么分区可以替代索引。在某些情况下，多维分区可能比单维索引更有效。

垂直分区，也称为柱状存储，当没有适当的索引来指导搜索时，可以进行高效扫描。有些人声称，只有一列和数据类型的数据结构允许更有效的压缩，因此也允许更快的扫描；其他人认为，在行存储和列存储[108]中都可以进行同样有效的压缩。除了压缩之外，扫描效率还受益于只传输查询执行计划中真正需要的那些列。另一方面，结果行必须从列扫描组装起来，利用共享或协调扫描可能更加复杂。

查询执行计划通常必须处理分区列和分区标识符，这与主索引中的搜索键非常相似。在索引到索引导航中对搜索键进行排序时，分区标识符应该是次要排序键；如果分区标识符是主要排序键，则将逐个搜索分区，而不是并行搜索。根据键分布的不同，在某些情况下可能需要从引用到排序键的更复杂的映射。

许多这样的考虑事项和技术也适用于更新。
在更新分区键时，索引条目可能需要从一个分区移动到另一个分区；这需要将更新分割为删除和插入。换句话说，在非分区索引中的性能优化可能是分区索引中的要求。

- 表和索引可以水平（按行）或垂直（按列）划分。
- 如果辅助索引以与表的主数据结构相同的方式进行水平划分，则该索引是本地的。从辅助索引到主数据结构的引用不像在全局索引中那样需要分区标识符。

- 大多数在查询执行期间的索引到索引导航策略和索引维护的策略都需要对分区表和索引进行一些调整。

.125总结

总之，在查询执行计划和更新执行计划中，b树索引可以优化使用。查询优化或查询执行的中阶或不完全实现降低了数据库中b树索引的值。似乎所有的实际实现在一个方面或另一个方面都有缺陷。因此，索引的创建、维护、扫描、搜索和连接仍然是一个创新和竞争的领域。由于数据库存储在闪存设备上，而不是传统的磁盘驱动器，数百倍的访问时间将增加对索引数据结构和算法的关注。

6

B树实用程序

除了利用数据库中的b树索引的各种高效的事务技术和查询处理方法之外，也许确实是大量现有的实用工具将b树与其他索引技术区分开来。例如，数据库或某些备份介质上的b树索引结构可以以任意顺序对数据页面进行一次扫描，并在内存或磁盘上保留有限数量的临时信息。即使是为大型数据集创建有效的索引也需要数年时间来开发新的索引结构。例如，在很长一段时间里，与简单地高效构建b树索引的排序相比，[23,62]的策略越来越有效，这也适用于适应于多维度[6,109]的b树索引。用于替代索引结构的可比技术通常不是对新索引技术的最初建议的一部分。

索引实用程序通常是设置操作，无论是通过准备一组未来索引项来创建索引，还是通过从b树等索引中处理一组索引事实来进行结构验证。因此，许多传统的查询处理技术

可用于索引实用程序，包括查询优化(e。g.，选择连接同一表的两个辅助索引或扫描表的主索引)、分区和并行性、用于准入控制和调度的工作负载管理，以及用于内存和临时磁盘空间或磁盘带宽的资源管理。类似地，可以在上面已经讨论过的索引实用程序中使用许多事务性技术。因此，下面的讨论并没有涉及与空间管理、分区、非记录操作、具有并发更新的在线操作等相关的实现问题。；相反，下面讨论的主要焦点是尚未涵盖的方面，但与一般索引的数据库实用程序，特别是b树的方面相关。

- 实用程序对于数据库及其应用程序的高效运行至关重要。b树的技术和实现比其他任何索引格式都要多。
- 实用程序经常会影响整个数据库、表或索引。它们经常运行很长时间。一些系统采用了来自查询优化和查询执行的技术和代码。

6.1指数创建

虽然一些产品最初依赖于重复插入来创建索引，但通过首先对未来的索引条目进行排序，索引创建的性能得到了很大的提高。因此，有效创建索引的技术可以分为快速排序的技术、从已排序的流构建b树的技术、支持并行构建索引的技术和创建索引的事务性技术。

在来自已排序流的b树构建过程中，未来b树的整个“右边”可能会一直被保留，甚至固定在缓冲池中，以避免在缓冲池中进行冗余搜索。类似地，在在线索引创建过程中，“大”锁[48]可能会被尽可能多地保留，并且只有在响应一个相互冲突的锁请求时才会被释放。或者，可以完全避免事务锁，因为索引创建不会修改逻辑数据库内容，只修改它们的表示。

通常会创建一个新的索引，并带有一些空闲空间，用于将来的插入和更新。每个叶节点内的空闲空间允许没有叶分割的插入，分支节点内的空闲空间允许在下一个较低的b树级分割，而分配单元内的空闲页面（如区段或磁盘柱）允许在分割期间没有昂贵的搜索操作，更重要的是，在所有后续范围查询和索引搜索期间。并不是所有的系统都能对这些形式的自由空间进行明确的控制，因为很难预测在未来的索引使用过程中，哪些参数值将会是最优的。此外，控制参数可能无法被精确地遵循。例如，如果每个叶子中所需的自由空间是10%，前缀b-树[10]可以选择相邻叶子的键范围，这样分隔键很短，并且左节点包含0%到20%的自由空间。

图6.1显示了在索引创建后立即显示的具有可变大小记录的固定大小的页面。所有的页面都是公平的，但并不一定是完全完整的。在一些数量的页面（这里是3）之后，一个空页面仍然可以随时可用于将来的页面分割。

在具有非唯一键值的索引中，排序顺序应该包含足够的引用信息，以使条目为唯一，如前所述。这将有助于并发控制、日志记录和恢复，以及最终删除条目。例如，当删除表中的逻辑行时，必须删除所有索引中与该行相关的所有记录。在非唯一的辅助索引中，此排序顺序可以找到可以有效地删除的正确条目。

如果一个次要索引是“非常非唯一的”，则i。e.，每个唯一的键值都有大量的引用，各种压缩方法可以减少索引的大小以及将初始索引写入磁盘、在查询处理期间扫描索引或在复制或备份操作期间复制索引的时间。最传统的表示方式将一个计数器和一个对每个唯一项的引用列表关联起来

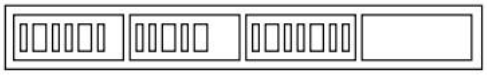


图6．1在索引创建后立即释放空间。

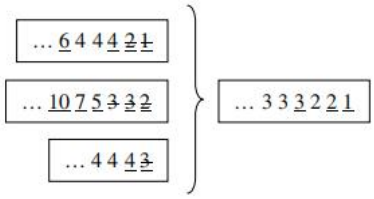


图6.2避免没有重复消除的比较。

键值[66]。在许多情况下，相邻引用之间的数值差异可以比完整引用值更紧凑地表示。适当的排序可以有效地构建这些差异列表；事实上，这样做的列表的构建可以被建模为一个聚合函数，从而可以减少在外部合并排序中写入临时存储的数据量。也可以使用位图。尽可能早地基于相同的键值将未来的索引条目分组，不仅可以进行压缩，从而在排序期间I/O更少，而且在合并步骤中比较更少，因为在条目分组后，只有一个具有代表性的键值需要参与合并逻辑。

图6.2（来自[46]）说明了三方合并的这一点。带下划线的键表示合并输入和合并输出中的一个组。值1、2和3在合并输入中被删除，因为它们已经通过了合并逻辑。在输入中，值2的两个副本都被标记为其运行中的一个组的代表。在输出中，只有第一个副本被标记，而第二个副本没有，将在下一个合并级别中被利用。对于值3，输入中的一个副本已经没有被标记出来，因此没有参与当前合并步骤的合并逻辑。在下一个合并级别中，值3的两个副本将不会参与合并逻辑。对于值4，节省的钱承诺更大：6个副本中只有2个将参与当前步骤的合并逻辑，而在下一个合并级别中只有六分之一。

在创建大型索引期间，另一个可能出现的问题是它们需要临时空间来保存运行文件。请注意，一旦合并进程消耗了运行文件，甚至运行文件中的单个页面可能会被“回收”。一些商业数据库系统

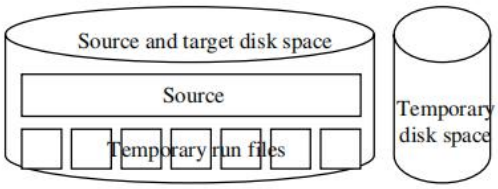


图6.3在目标空间中的排序。

因此，将运行存储在为最终索引指定的磁盘空间中，默认存储或作为选项。在最终合并期间，页面创建的索引。如果目标空间是唯一可用的磁盘空间，则在运行中使用它没有其他选择，尽管此选择的一个明显问题是目标空间通常位于镜像磁盘或冗余RAID磁盘上。此外，在目标空间中的排序可能会导致最终的索引相当碎片化，因为页面可以从合并输入中循环使用，从而以随机顺序有效地合并输出。因此，对结果索引的索引顺序扫描，例如大范围查询，会导致许多磁盘搜索。

图6.3说明了排序操作的数据源大于临时空间的情况。因此，不能将临时运行文件放置在临时数据的标准位置。相反，运行文件被放置在目标空间中。由于合并步骤会消耗运行文件，因此必须立即释放磁盘空间，以便为合并步骤的输出创建可用空间。

有两种可能的解决方案。首先，最终的合并可以将页面释放到可用页面的全局池，最终创建索引尝试从那里分配大的连续磁盘空间。然而，除非分配算法对连续自由空间的搜索非常有效，否则大多数分配将在相同的小规模内，在合并中回收空间。其次，空间可以从初始运行到中间运行，中间运行，以及更大单元的最终索引，通常是I/O单元的倍数。例如，如果这个倍数为8，则可以保留不超过8倍的磁盘空间，这通常是创建大索引时可接受的开销。

其好处是，一个全索引顺序扫描或一个大范围的扫描
在大型有序扫描中，完成的索引需要的搜索次数少了8倍。如果索引创建
的排序操作使用了最终的b树空间
对于临时运行，从系统或介质故障中恢复必须非常精确地重复原始排序操作。否则，恢复可能会将b树项与原始执行不同，并且描述b树更新的后续日志记录不能应用于恢复的b树索引。具体来说，初始运行的大小、合并步骤的顺序、合并扇入、合并输入的选择等等。所有这些都必须记录或包含创建索引的一些信息，e。g.，授予该排序操作的内存分配。因此，在恢复期间，必须为排序提供与原始执行期间相同的内存。为排序提供数据的扫描也必须精确地重复，而不排列输入页面或记录，e。g.，由于异步的预读。如果在与原始执行相同的不同硬件上调用恢复，这可能是一个问题。g.，在发生灾难性的硬件故障后，如洪水或火灾。对精确重复执行的需要也可能会抑制索引创建过程中的自适应内存分配。e.，内存分配，初始运行大小，合并风扇在所有在索引创建期间响应内存争用的波动。

不仅可以使用分组和聚合，还可以使用从查询处理中获得的各种技术来创建索引，包括查询优化和查询执行技术。例如，创建新的辅助索引的标准技术是扫描表的基本结构；但是，如果存在两个或多个包含所有必需列的辅助索引，并且可以比基本结构更快地扫描，查询优化可能会选择一个计划来扫描这些现有索引并将结果连接起来，以构建新索引的条目。查询优化在视图物化过程中扮演更大的角色，在某些系统中，它被建模为为视图而不是表的索引创建。

对于并行索引的创建，可以使用标准的并行查询执行技术，以所需的排序顺序生成未来的索引条目。剩下的问题是并行插入到新的b树数据结构中。一种方法是创建多个单独的方法

具有不相交键范围的b树，并将它们与单个“叶-根传递”和负载平衡“缝合”在一起。

- 高效的b树创建依赖于高效的排序和提供事务保证，而不记录新的索引内容。
- 用于创建索引的命令通常有很多选项，e。g.，关于压缩、关于为将来的更新留出空闲空间，以及关于为将来的索引条目进行排序的临时空间。

6.2索引去除

由于各种原因，删除指数可能看起来相当微不足道，但也可能不是。例如，如果索引删除可以是较大事务的一部分，该事务是否阻止所有其他事务访问表，即使索引删除事务可能中止？可以是在线的索引删除，i。e.，是否可以为该表启用并发查询和更新？对于另一个例子，如果一个表同时有一个主（非冗余）索引和一些辅助索引（通过搜索键指向主索引中的记录），那么在删除主索引时需要多少努力？也许主索引的叶子会成为堆，仅仅释放分支节点，但是重建辅助索引需要多长时间呢？同样，索引删除可以在线进行吗？

最后，一个索引可能非常大，并更新了分配信息(e。g.，自由空间地图)可能需要相当长的时间。在这种情况下，“即时”删除索引可能仅仅声明索引在适当的目录记录中过时。这有点类似于鬼记录，除了鬼指示器只适用于发生它的记录，而这个过时指示器适用于目录记录所表示的整个索引。此外，尽管一个幽灵记录可能在其创建后很长时间就被删除，但一个被删除的索引的空间应该尽快被释放，因为可能涉及大量的存储空间。即使在释放此空间之前或过程中发生了系统崩溃，该过程也应该在成功重新启动后快速继续。合适的日志记录在文件中

需要恢复日志，精心设计可以减少日志记录量，但也可以确保在重复尝试恢复时发生崩溃时成功。

作为目录记录中的过时指示器的一个替代方案，内存中的数据结构可以表示延迟的工作。请注意，此数据结构是服务器状态（在内存中）的一部分，而不是数据库状态（在磁盘上）。因此，除非服务器在完成延迟工作之前崩溃，否则这个数据结构工作得很好。对于这种情况，数据结构的创建和最终删除都应该记录在恢复日志中。因此，这种替代设计并不会节省日志记录工作。此外，这两种设计都要求在正常处理和可能的崩溃和随后恢复后的恢复期间都支持中间状态。

- 索引删除可能是复杂的，特别是如果必须创建一些结构作为响应。
- 索引删除可以立即延迟用于空闲空间管理的数据结构的更新。

许多其他实用程序也可以使用这个执行模型，但索引删除似乎是最明显的候选工具。

6.3索引重建

重建现有索引的原因有多种多样，有些系统需要在高效碎片整理更合适时重新构建索引，特别是如果索引重建可以在线或增量索引。

如果主索引中的搜索键不是唯一的，并且需要在人工字段中获得新的值，则可能需要重新构建主索引。移动一个带有物理记录标识符的表同样会修改所有引用。请注意，这两个操作都要求重新构建所有辅助索引，以反映修改后的引用值。

当主索引更改时，也需要重新构建所有辅助索引。e.，当主索引中的键列集发生更改时。如果仅仅是它们的序列更改，则不严格要求分配新的引用并重建辅助索引。

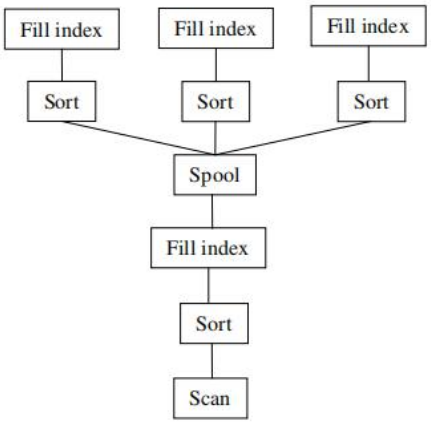


图6.4重建主要指标和次要指标。

更新所有现有的辅助索引可能比重建索引要慢，部分原因是更新需要恢复日志中的完整信息，而重建索引可以使用非记录的索引创建（仅限分配的日志记录）。

图6.4说明了重建表的主索引以及随后的三个辅助索引的查询执行计划。该扫描将从当前的主索引或堆文件中捕获数据。排序将准备填充新的主索引。滑轴操作只保留在临时存储中的那些辅助索引所需的列。如果重复扫描新的主索引比写入和重新读取线轴数据更便宜，则可以省略线轴操作。或者，线轴操作之后的各个排序操作可以满足线轴操作的目的，如前面关于图6.4所讨论的那样。

除了未记录的索引创建之外，索引重建操作还可以使用索引创建中的其他技术，如分区、并行执行、使用不相交的键范围缝合b树等。对于在线索引重建操作，使用同样的技术进行锁定，反物质等。申请创建在线索引。

- 在损坏后（由于软件或硬件故障）或需要进行碎片整理但删除和重建索引更快时，可以重新构建索引。

- 当重新构建主索引时，通常也必须重新构建次索引。各种优化适用于此操作，包括一些通常不用于标准查询处理的优化。

6.4批量插入

批量插入，也称为增量加载、滚入或信息捕获，在许多数据库中是一种非常常见的操作，特别是数据仓库、数据集市和其他保存有关事件或活动（如销售交易）信息的数据库，而不是状态（如帐户余额）。批量插入的性能和可伸缩性有时会在竞争供应商之间决定何时进行夜间数据库维护或初始概念验证实现的时间窗口较短。

任何对批量插入的性能或带宽的分析都必须区分即时带宽和持续带宽，以及在线和离线负载操作。第一个区别是由于物化视图、索引、直方图等统计数据的延迟维护。例如，分区的B树[43]启用高即时负载带宽（基本上，以磁盘写入速度附加到B树）。但是，最终，随着每个b树中增加附加分区，查询性能会下降，需要通过合并分区进行重组；在确定可以无限持续的负载带宽时，必须考虑这种重组。

第二个区别集中在在加载操作期间为应用程序提供查询和更新服务的能力。例如，某些版本的一些数据库供应商建议在大批量插入之前删除所有索引，例如大于现有表大小的1%的插入。这是由于索引插入的性能不佳；为之前表大小的101%重建所有索引可以比等于表大小的1%的插入更快。

为有效地批量插入到b型树而优化的技术可以分为两组。两组都依靠某种形式的缓冲来延迟b树的维护，并获得一些规模经济。第一组主要关注b型树的结构和缓冲区插入

分支节点[74]。因此，b树节点非常大，被限制在一个小的扇形输出，或者需要额外的存储在“侧面”。第二组利用b树而不修改它们的结构，要么使用多个[100]树[100]，要么通过人工领先关键字段[43]在单个b树中创建分区。在所有情况下，具有活动插入的页面或分区都会保留在缓冲区池中。各种方法的相对性能，特别是在持续带宽方面，尚未进行实验研究。下面的一些简单的计算突出显示了对负载带宽的主要影响。

图6.5说明了一个缓冲插入的b树节点，e。g.，一个根节点或一个分支节点。有两个分隔符键（11和47），三个指向同一b树中的子节点的指针，以及带有每个子指针的一组缓冲插入。在次要索引中，索引条目包含一个键值和对表的主索引中的记录的引用，这里用“ref”表示。换句话说，每个被缓冲的插入都是一个未来的叶项。中间子的缓冲插入集比左子的缓冲插入集小得多，这可能是由于工作负载的倾斜或最近插入向中间子的的传播。为右子项缓冲的更改集不仅包括插入，还包括删除（键值72)。缓冲删除仅在辅助索引中可行，因为在先前更新主索引之后，已确保要删除的值确实必须存在于辅助索引中。

分区b树的本质是通过人工引导关键字段来在单个b树中维护分区，并有效地使用外部合并排序中已知的合并步骤来在线重组和优化这样的b树。这个关键字段可能应该是这样的

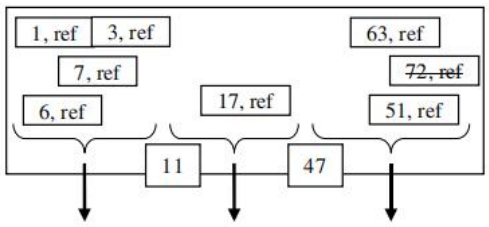


图6.5在分支节点中的缓冲区。

是一个包含2个或4个字节的整数。默认情况下，相同的单个值出现在b树中的所有记录中，并且大多数分区b树的技术依赖于利用多个替代值，在大多数情况下是暂时的，对于少数技术是永久的。如果关系数据库中的一个表或视图有多个索引，则每个索引都有自己的人工引导键字段。这些字段中的值不会在各个索引之间进行协调或传播。换句话说，每个人工领先的关键字段都是一个b树的内部，这样每个b树都可以独立于所有其他b树进行重组和优化。如果一个表或索引是水平分区并用多个b树表示，则应该为每个分区分别定义人工引导键字段。

图6.6说明了人工引导关键字字段如何将b树中的记录划分为分区。在每个分区中，记录可以通过用户定义的键进行排序、索引和搜索，就像在标准的b树中一样。在这个示例中，分区0可能是主分区，而分区3和分区4包含最近的插入，它们在内存中排序后作为新的分区附加到b树中。最后一个分区可能会保留在缓冲池中，在那里它可以非常有效地吸收随机插入。当其大小超过可用缓冲池时，将启动一个新分区，并将前一个分区从缓冲池写入磁盘，或者在缓冲池中的标准页面替换期间根据缓冲池请求写入磁盘。或者，显式排序操作可以对大量插入进行排序，然后附加一个或多个分区。显式排序实际上只执行运行生成，将运行作为分区附加到已分区的b树中。

初始加载操作应该将新插入的记录或新填充的页面复制到恢复日志中，以便创建新的数据库

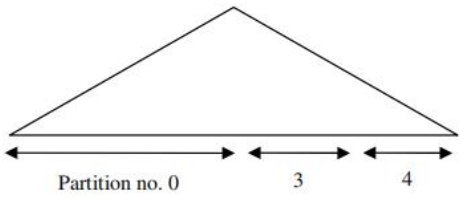


图6.6b树中的分区。

即使在媒体或系统出现故障时，也可以保证提供内容。b树的重组可以避免记录内容，从而通过仔细的写顺序只记录b树索引中的结构变化。具体来说，删除记录或指针的页面只有在新位置写入记录的新副本后，才会在旧位置覆盖其早期版本。对于其他形式的b树重组[44,89,130]，已经描述了通过仔细写顺序启用的最小日志记录，但它也适用于通过附加新分区批量插入后的分区b树中的合并。

对于一些带宽计算的例子，考虑批量插入到一个具有主索引和三个辅助索引的表中，所有这些都存储在一个磁盘上，支持每秒200个读写操作（100个读写对）和100个MB/s读写带宽（假设I/O单位很大，因此访问延迟可以忽略不计）。在这个示例计算中，记录大小在主索引中为1 KB，在次要索引中为0.02 KB，包括页面和记录头的开销、可用空间等。为简单起见，让我们假设一个温暖的缓冲池，这样只有叶子页面需要I/O。基线计划依赖于随机插入到4个索引中，每个索引需要一个读操作和一个写操作。每插入行8个I/O，每秒插入25行。持续插入带宽为每个磁盘驱动器25×1 KB = 25 KB/s = 0.025 MB/s。这是即时的和持续的插入带宽。

对于依赖于索引删除和插入后重新创建的计划，假设索引删除实际上是即时的。在没有索引的情况下，即时插入带宽等于磁盘写入带宽。在将表大小增长1%之后，索引创建必须扫描101倍插入卷，然后写入该数据卷的106%以运行4个索引文件（排序辅助索引时共享运行生成），最后将这些运行合并到索引中：对于给定的数据量，I/O卷为1 + 101×（1 + 3×1.06）=的423倍；在100 MB/s时，这允许100/423 MB/s = 0.236 MB/s持续插入带宽。虽然与100 MB/s相比，这可能看起来很差，但它比随机插入快10倍，所以供应商推荐这个方案也就不足为奇了。

356B树实用程序

对于在每个分支节点上缓冲插入的b树实现，假设每个节点中的缓冲空间的插入比节点中的子指针多10倍。溢出时的传播侧重于具有最多未决插入的子项；让我们假设平均可以传播20条记录。因此，只有每20个记录插入强制读写一个b树叶，或者每个记录插入的1/20个读写对。另一方面，带有缓冲区的b树节点要大得多，因此每个记录插入可能需要读写一个叶节点及其父节点，在这种情况下，每个记录插入强制一个读写对的2/20 = 1/10。在包含一个主索引和三个次级索引的示例表中，i。e.，总共有4棵b树，这些假设导致每个插入的记录有4/10的读写对。假设的每秒有100个读写对的磁盘硬件因此支持每秒250个记录插入或0.250 MB/s的持续插入带宽。与之前的方法相比，除了有轻微的带宽改进外，该技术还保留并维护了原始的b树索引，并允许在整个加载过程中进行查询处理。

使用假设的磁盘硬件，分区的b树允许100 MB/s的即时插入带宽，i。e.，使用快速排序或替换选择，纯附加在内存工作区中排序的新分区。b树优化，i。e.，在分区b树中，可以处理读写分区的单个合并级别50 MB/s .如果在添加的分区达到主分区大小的33%时调用重组，那么添加给定数量的数据需要相同数量的初始写入加上4倍的初始数量来进行重组（读取和写入）。每量数据的9个I/O量为单个b树产生11 MB/s的持续插入带宽。对于带有主索引和三个辅助索引的示例表，该带宽必须在所有索引中划分，给出11 MB/s÷（1+3×0.02）KB = 10 MB/s持续插入带宽。这比其他批量装载技术快一个数量级。此外，在整个新信息的初始捕获过程中，以及在b树重组期间，查询处理仍然是可能的。一个多级的合并方案可能会进一步增加这个带宽，因为主分区

重组的次数较少，但现有分区的数量可以保持很少。

除了传统的数据库之外，b-树索引也可以用于数据流。如果在索引中只保留最近的数据项，则需要批量插入技术和批量删除技术。因此，索引流将在下一节中进行讨论。

- 批量插入（也称为加载、滚动或信息捕获）的效率对于数据库操作至关重要。
- 在某些实现中，索引维护的效率非常差，以至于在较大的负载操作之前就会删除索引。已经发布并实现了各种技术，以加快对b树索引的批量插入。它们的持续插入带宽相差不同数量级。

6.5批量删除

批量删除，也被称为清除、推出或信息去分期，可以使用一些为批量插入而发明的技术。例如，一种删除技术只是简单地使用最快的批量插入技术插入反物质记录，并将其留给查询或随后的重组，以删除记录和回收存储空间。

分区的b型树允许在实际删除之前进行重组。在第一步和准备步骤中，要删除的记录从主“源”分区移动到专用的“受害者”分区。在第二步，也是最后一步，这个专用分区被非常有效地删除，主要是简单地对叶节点的去分配和对内部b树节点的适当修复。请注意，第一步可以是增量式的，完全只依赖于系统事务，并且可以在信息真正应该从数据库中消失之前运行。

图6.7显示了在准备批量删除后的分区b树的中间状态。要删除的b树条目都已从主分区移动到一个单独的分区中，这样它们的实际删除和删除就可以取消整个分配

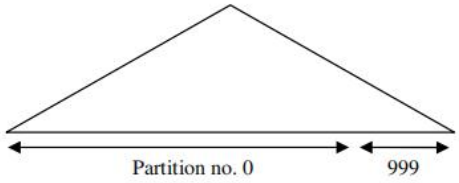


图6.7为批量删除而准备的分区b树。

页页而不是删除分布在所有页页中的单个记录。如果可以预期未来会有多个删除，则e。g.，每天清除过时的信息，可以同时填充多个受害者分区。

在分区b树中批量删除期间的日志体积可以通过多种方式进行优化。首先，初始重组（“不合并”）到一个或多个受害者分区可以使用与基于仔细的写顺序的合并相同的技术。其次，在将受害者分区被写入回数据库之后，将多个有效记录转换为幽灵记录可以在一个短日志记录中进行描述。第三，如前所述，如果合并了要删除和提交的日志记录，则删除鬼记录不需要内容日志记录。第四，对整个b树节点（页面）的去分配可以使用类似的技术，将分隔符键转换为幽灵记录。如果受害者分区非常小，可以保留在缓冲池中直到最终删除，那么受害者分区中的页面删除允许写入源分区中的脏页面。

批量插入和批量删除的技术一起实现了数据流的索引。数据流和近实时的数据处理可以受益于许多数据库技术，也许已经适应了，e。g.，从需求驱动的执行到数据驱动的执行[22]。然而，大多数流管理系统并不为流内容提供持久的索引，因为中殿或传统的索引维护技术会大大减慢处理速度。

使用高带宽插入（附加在内存中排序的分区）、索引优化（合并运行）、分区分割（根据预期的删除日期）和删除（通过删除整个分区），甚至可以永久维护流

储藏例如，如果一个磁盘驱动器可以以100 MB/s的速度移动数据，那么就可以附加新数据，合并最近的分区，从主分区分离出来的立即过时的分区，以及持续切割约20 MB/s的真正过时的分区。如果初始分区或中间分区被放置在特别有效的存储器上，则为e。g.，闪存设备或非易失性RAM，或者如果设备排列在阵列中，系统带宽可能会高得多。

具有多个独立索引的流能够有效地插入新数据并同时删除过时的数据，即使多个索引需要持续维护。在这种情况下，同步所有所需的活动会增加一些开销。尽管如此，示例磁盘驱动器可以在20 MB/s下吸收和清除索引项（所有索引一起）。

类似的技术允许在存储层次结构的多个层次中分段数据。g.，内存存储、闪存设备、性能优化的“企业”磁盘和容量优化的“消费者”磁盘。磁盘存储可能不仅在驱动器技术上有所不同，而且在处理冗余性和故障恢复能力的方法上也有所不同。例如，性能通过RAID-1“镜像”配置进行优化，而每容量的成本通过RAID-5“条带冗余”配置或RAID-6“双冗余”配置进行优化。请注意，RAID-5和-6在每容量成本上可以相等，因为后者可以容忍双重故障，因此可以用于更大的磁盘阵列。

- 批量删除没有批量插入重要；尽管如此，各种优化可能会影响带宽的数量级。
- 索引数据流需要来自批量插入和批量删除的技术。

6.6碎片整理

[1] 文件系统中的碎片整理通常意味着将属于同一文件的块在物理上放在一起；在数据库b树中，

[1] 本节中的许多材料都是从[55]中复制过来的。

碎片整理还包括其他一些考虑事项。这些考虑事项适用于单个b树节点或页面、b树结构和分隔符键。在许多情况下，由于正常的工作负载处理，当部分或所有受影响的页面在缓冲区池中时，可以调用碎片整理逻辑，从而导致增量和在线碎片整理或重组[130]。

对于每个节点，碎片整理包括每个页面内的空闲空间整合，以便于有效的未来插入、删除幽灵记录（除非当前被用户事务锁定）和优化pige数据压缩(e。g.，字段值的重复数据消除）。b树结构可以通过碎片整理进行优化，以实现平衡的空间利用、如上面在b树创建上下文中讨论的空闲空间、更短的分隔键（后缀截断）以及每个页面上更好的前缀截断。

b树碎片整理可以按键顺序或独立的键范围进行，这也为并行性创造了机会。每个任务的关键范围可以先验地或动态地确定。例如，当系统负载增加时，碎片整理任务可以立即提交其更改、暂停并稍后恢复。请注意，碎片整理不会改变b树的内容，只是改变它的表示。因此，碎片整理任务不需要获取锁。当然，它必须获取锁存器，以保护内存中的数据结构，如缓冲池中的页面图像。

在传统的b树结构中移动一个节点是相当昂贵的，原因有几个。首先，页面内容可以从缓冲池中的一个页面帧复制到另一个页面帧。虽然这样做的成本是适中的，但“重命名”一个缓冲区页面可能会更快，i。e.，同时为新旧位置和新旧位置分配和锁存缓冲区描述符，然后将页面帧从一个描述符转移到另一个描述符。因此，页面应该在缓冲池中“通过引用”而不是“按值”迁移。如果每个页面都包含其预期的磁盘位置，以帮助进行数据库一致性检查，则此时必须更新此字段。如果可能有一个取消分配的页面徘徊在缓冲池中，e。g.，在创建、写入、读取和删除临时表之后，此优化的缓冲区操作必须首先从缓冲区的哈希表中删除任何带有新页面标识符的前一个页面。

或者，这两个缓冲区描述符可以简单地交换它们的两个页面帧。

其次，移动一个页面可能很昂贵，因为每个b树节点都参与一个指针web。当移动叶页时，必须更新父页以及前一页和后一页。因此，周围的所有三个页面必须出现在缓冲池中，它们的更改记录在恢复日志中，以及在下一个检查点之前或期间写入磁盘的修改页面。同时移动多个叶页通常是有利的，这样每个叶页只读写一次。尽管如此，每个单页移动操作都可以是一个单一的系统事务，这样锁就可以经常为分配信息(e。g.，一个分配位图)和用于正在重新组织的索引。

如果每个级别内的b树节点没有通过物理页面标识符形成一个链，则i。e.，如果每个b树节点只由其父节点指向，而不是由邻居节点指向，那么页面迁移和碎片整理的成本要低得多。具体来说，当页面移动时，只有b树节点的父节点需要更新。它的兄弟代和子代都不受影响；在页面迁移期间不需要在内存中，它们不需要I/O、更改或日志记录等。

页面迁移可能非常昂贵的第三个原因是日志记录。e.，写入恢复日志的信息量。在碎片整理期间，记录页面迁移的标准“完全记录”方法是记录页面内容，作为分配和格式化新页面的一部分。从系统崩溃或媒体故障中恢复，无条件地将页面内容从日志记录复制到磁盘上的页面，就像它对所有其他页面分配所做的那样。

然而，记录整个页面内容只是使迁移持久性的几种方法之一。第二种“强制写”方法是用一个小日志记录记录迁移本身，该记录包含新旧页面位置，但不包含页面内容，并在提交页面迁移之前将数据页面强制到新位置的磁盘上。在日志记录和恢复[67]的理论和实践中，在事务提交之前将更新的数据页强制转到磁盘是很好的建立。从系统崩溃中恢复可以安全地假定a

已提交的迁移将反映在磁盘上。另一方面，介质恢复必须重复页面迁移，并且能够这样做，因为在日志驱动的重做期间，旧的页面位置此时仍然包含正确的内容。这同样适用于日志传输和数据库镜像。e.，通过从主站点连续发送恢复日志并在辅助站点上运行连续重做恢复，保持第二个（通常是远程）数据库准备好进行即时故障转移的技术。

最雄心勃勃、最高效的碎片整理方法既不会记录页面内容，也不会将其强制存储到新位置的磁盘上。相反，这种“非记录”的页面迁移依赖于旧的页面位置，以保存可以基于其进行恢复的页面映像。在系统恢复期间，会检查旧页面的位置。如果它包含一个低于迁移日志记录的日志序列号，则必须重复迁移，i。e.，在旧页面恢复到迁移时间之后，必须在缓冲池中重新重命名该页面，然后可以向新页面应用其他日志记录。为了保证能够从故障中恢复，必须将旧页面图像保存在旧位置，直到新图像写入新位置。即使在迁移事务提交之后，单独的事务为新目的分配旧位置，在迁移页面成功写入新位置之前，也不能在磁盘上覆盖旧位置。因此，如果系统恢复在旧页面位置找到一个新的日志序列号，它可以安全地假设迁移的页面内容在新位置可用，并且不需要进一步的恢复操作。

一些可恢复的b树维护方法已经使用了缓冲池中的数据页之间的这种写依赖关系，以及众所周知的预写日志记录的写依赖关系。要使用标准技术实现这种依赖项，新旧页面都必须在缓冲区管理器中表示。与通常的写依赖关系不同，旧位置可以被迁移事务标记干净。e.，它不需要将任何东西写回磁盘上的旧位置。请注意，迁移事务的重做恢复必须重新创建此写依赖关系，e。g.，在媒体恢复和日志运输中。

这第三种方法的潜在缺点是备份和恢复操作，特别是如果备份是“联机的”，i。e.，在系统主动处理用户事务时进行，并且备份不包含整个数据库，而只包含当前分配给某些表或索引的页面。此外，备份过程和页面迁移的详细操作必须以一种特别不幸的方式相互交织。在这种情况下，备份可能不包含旧位置上的页面映像，因为它已被取消分配。因此，在备份日志以补充在线数据库备份时，迁移事务必须由新的页面映像来补充。实际上，在联机数据库备份及其相应的恢复操作中，日志记录和恢复行为从未记录的页面迁移改变为完全记录的页面迁移。在还原操作期间应用此日志时，必须检索添加到迁移日志记录中的页面内容，并将其写入其新位置。如果页面也反映了在页面迁移后发生的后续更改，那么由于页面上的日志序列号，恢复将正确处理这些更改。同样，这与现有的机制非常相似，在这种情况下，是由一些商业数据库管理系统支持的“非记录”索引创建的备份和恢复。

当迁移事务将页面从旧位置移动到新位置时，用户事务锁定b树节点中的键是可以接受的。但是，任何这样的用户事务都必须再次搜索B树节点，并重新将搜索从B树根传递到叶子，以便获得新的页面标识符，并正确地记录进一步的内容更改，如果有的话。这与b树节点的拆分和合并操作非常相似，这也会使用户事务可能暂时保留的页面标识符的知识失效。最后，如果用户事务必须回滚，则它必须补偿其在新位置的操作，这同样与在不同事务拆分或合并b树节点后补偿用户事务非常相似。

- 大多数b树的实现（与其他存储结构一样）都需要偶尔进行碎片整理（重组），以确保连续性（在扫描期间进行更少的搜索）、可用空间等。

- 通过使用栅栏键而不是邻居指针（参见第3.5和4.4节）和仔细的写入顺序（参见第4.10节），可以降低页面移动的成本。
- 碎片整理（重组、压缩）可以在许多小的系统事务中进行，它可以“暂停和恢复”而不浪费工作。

6.7指标验证

[2]显然，有许多高效的数据结构技术和b树索引的算法。随着更多的技术在一个特定的软件系统中被发明或实现，遗漏或错误就会出现，并且必须被发现。许多这些错误都表现在不满足预期不变量的数据结构中。因此，作为软件开发和改进过程中严格的回归测试的一部分，b-树的验证是一个至关重要的必要性。

许多这些遗漏和错误需要大量的b树、高更新和查询负载，以及频繁的验证，以便在软件开发过程中及时发现。因此，效率在b树验证中很重要。

当然，在部署后，还需要对b型树进行验证。硬件缺陷发生在DRAM、闪存设备和磁盘设备中。软件缺陷不仅可以在数据库管理系统中发现，也可以在设备驱动程序、文件系统代码等中发现。[5, 61].虽然在许多硬件和软件层中都存在一些自检，但数据库管理系统的供应商建议定期验证数据库。对备份媒体的验证也很有价值，因为它增强了人们对这些媒体及其内容的信任和信心。

例如，Mohan描述了由于SCSI标准[94]实现中的性能优化而导致的部分写入的危险。他的重点是使用适当的页面修改来预防问题，每次阅读操作后的页面验证，日志记录，

[2] 本节的大部分内容都来自于[54]。

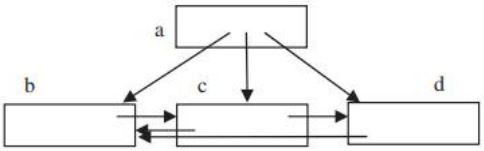


图6.8一个不完全的叶分裂。

日志分析、恢复逻辑等。这些技术的复杂性，加上对这些性能模块的不断改进，强化了我们的信念，即b树结构的完整、可靠和有效的验证是一种必要的防御措施。

例如，图6.8显示了叶节点不正确分割的结果。当分割叶节点b并创建叶节点c时，后续节点d中的向后指针错误地保持不变。后续的叶级（降序）扫描将产生错误的查询结果，随后的拆分和合并操作将造成进一步的破坏。该问题可能在不完整执行、不完整恢复或不完整复制之后出现。原因可能是数据库软件的缺陷。g.，在缓冲池管理中，或在存储管理软件中，e。g.，在快照或版本管理中。换句话说，有成千上万行代码可能包含一个缺陷，从而导致如图6.8所示的情况。

由于b树是复杂的数据结构，对所有不变量的有效验证一直难以捉摸，包括页面内不变量、父不变量
子指针和邻居指针，以及键关系，i。e.，叶节点中分隔键和键的正确顺序。后一个问题不仅涉及父子关系，而且涉及所有的祖先-后代关系。例如，B树的根节点中的分隔符键不仅必须对根的直接子节点中的键进行排序，还必须对所有B树级别上的键进行排序。与多个b-树相关的不变量，e。g.，表及其辅助索引的主索引或物化视图及其底层表和视图，通常可以使用适当的连接进行处理。如果将数据库验证的所有方面都建模为查询处理问题，那么就可以利用许多查询处理技术，从资源管理到并行执行。

一旦页面在缓冲区池中，页面内不变量就很容易进行验证，但详尽的验证需要检查所有不变量的所有实例，e。g.，所有相邻叶节点之间的键范围关系。跨页面不变量很容易通过对整个键范围的索引顺序扫描来验证。但是，如果由于并行执行或由于磁带等备份介质的限制而不需要索引顺序扫描，则可以使用基于聚合的算法来验证结构不变量。在按任何顺序扫描b树页面时，会从每个页面中提取所需的信息，并与来自其他页面的信息相匹配。例如，如果页面x将页面y作为后继者，页面y将页面x作为前继者，则邻居指针匹配。键范围必须包含在提取的信息中，以确保键范围是不相交的，并通过适当的祖先节点中的分隔符键正确区分。如果两个叶节点共享一个父节点，则此测试非常简单；如果最低的共同祖先在b树的更前面，如果要避免传递操作，则必须在b树节点中保留一些额外的信息。

图6.9显示了一个b树，相邻的叶节点没有共享的父节点，而是一个共享的祖父节点i。e.，表亲节点d和e。着色区域表示记录及其键；如果两个键的着色值不同（相等），则它们就会有所不同（相等）。在传统的b树实现中，对表亲节点d和表亲节点e之间的键和指针的有效验证并没有一个直接或明显的有效解决方案。潜在的问题是，没有简单的方法来验证所有的密钥

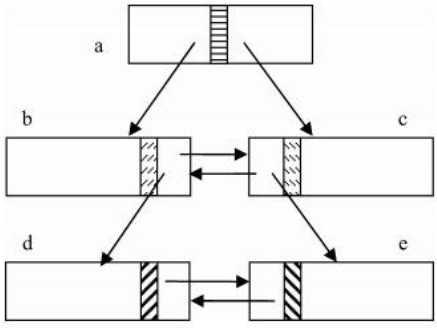


图6.9b树验证中的表亲问题。

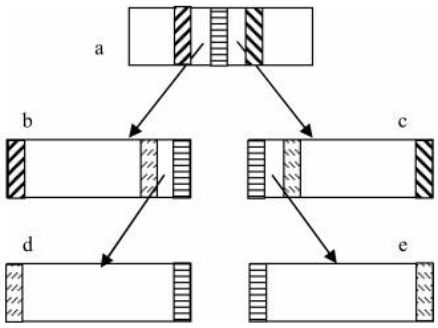


图6．10个栅栏键和表亲节点。

在叶页d中，d确实比根页a中的分隔键小，而且叶页e中的所有键确实比根页a中的分隔键大。正确的邻居之间的键关系（-和-）和父母与孩子之间的键关系（-，-，-，-）不能保证跳过级别的正确键关系（-，-）的键关系。

图6.10显示了栅栏键如何为b树验证中的表亲问题提供一个简单的解决方案，即使栅栏键最初是由写优化的b树[44]驱动的。与传统的b树设计的本质区别在于，页面拆分不仅向父页面发布了一个分隔键，而且在两个拆分后的兄弟页面中保留了这个分隔键的高和低的“栅栏键”。请注意，由于前缀和后缀截断[10]，分隔符和栅栏键可能非常短。这些栅栏键充当了兄弟姐妹指针的角色，用搜索键替换了传统的页面标识符。栅栏键通过删除在页面移动时必须更新的所有页面标识符，即父节点中的子指针，来加快碎片整理速度。栅栏键还有助于锁定键范围，因为它们是可以锁定的键值。从这个意义上说，它们与传统的幽灵记录相似，只是栅栏键不受幽灵清理的影响。

这里的重要好处是，验证是被简化的，表亲问题可以很容易地解决，包括“二表亲”、“第三表亲”等。在b型树中有额外的层次。在图6.10中，可以导出关于标记的键的以下四对事实

通过水平着色，每对都独立于两页。

1. 从a页开始，b是一个一级的页面，以及它的高栅栏键
2. 从a页开始，c是一个一级页面，以及它的低栅栏键
3. 从b页开始，b是一个一级页面，以及它的高栅栏键；这与上面的事实1相匹配
4. 从b页开始，d是一个叶子页，以及它的高栅栏键
5. 从c页开始，c是一个一级页面，以及它的低栅栏键；这与上面的事实2相匹配
6. 从c页开始，e是一个叶页，以及它的低栅栏键
7. 从d页开始，d是一个叶页，以及它的高栅栏键；这与上面的事实4相匹配
8. 从e页开始，e是一个叶页，以及它的低栅栏键；这与上面的事实6相匹配

表亲页d和e之间不需要匹配。由于其他比较中的传递性，它们的栅栏键是相等的。事实上，从页面d和页面e中得到的匹配事实不能包含页面标识符，因为这些页面不包含其他人的页面标识符。在最好的情况下，可以得出以下事实，尽管它们被上述事实所暗示，因此对b树验证的质量没有贡献：

9. 从第b页开始，第1级页面有一个特定的高栅栏键
10. 从c页开始，一级页面有一个特定的低栅栏键；匹配上面的事实9
11. 从第d页开始，一个叶子页有一个特定的高栅栏键
12. 从第e页开始，一个叶页有一个特定的低栅栏键；来匹配上面的事实11

根的分隔键沿相邻节点的整个接缝一直复制到叶级。平等和一致性沿整个接缝检查，并通过传递性检查。因此，栅栏键也解决了二表亲、三表亲等问题。在b型树中有额外的层次。

这些事实可以以任何顺序推导出来；因此，b树验证可以使用从磁盘顺序扫描甚至从备份介质中获得的数据库页面。这些事实可以使用内存中的散列表（可能散列表溢出到磁盘上的分区文件），也可以用于切换位图中的位。前一种方法需要更多的内存和更多的CPU工作量，但可以立即识别任何错误；后一种方法更快，需要更少的内存，但如果某些事实与来自其他页面的相同事实不匹配，则需要第二次通过数据库。此外，位图方法无法检测到两个相互掩盖的错误的概率很小。

栅栏键还扩展了本地在线验证技术[80]。在传统的系统中，邻居指针可以验证在root-toleaf导航只为兄弟姐妹，因为兄弟姐妹的身份信息共享的父节点但验证表指针需要I/O操作获取表的父节点（也为表妹的祖父母节点，等等）。因此，早期的技术[80]不能验证b树中的所有正确性约束，无论有多少搜索操作执行验证。另一方面，栅栏键在整个b树接缝上都相等，从叶级到祖先节点，其中键值作为分隔键。可以利用栅栏键值在b树的每个级别进行在线验证，在查询和更新处理过程中，普通的根到叶b树下降不仅可以验证具有共享父本的兄弟姐妹，还可以验证表亲、表亲等。对相邻叶子中的键进行两个搜索操作，验证所有b树约束，即使叶子是表亲节点，而接触所有叶子节点的搜索操作，验证整个b树中的所有正确性约束。

例如，图6.10中所示的索引中的两个从根到叶的搜索可能以叶节点d和e结束。假设这两个从根到叶的传递发生在单独的事务中。这两个搜索可以验证整个接缝上正确的围栏键。在B树中

采用邻居指针而不是栅栏键如图6.9所示，相同的两个根叶搜索可以验证条目叶节点d和e确实更小和大于根符键，但他们不能验证表节点之间的指针d和e是相互的和一致的。

通过提取和匹配事实的B树验证不仅适用于传统的B-树，也要眨眼-树木和它们的过渡状态。在节点分裂后，父节点还没有更新在眨眼-树，从而产生上面的事实。新分配的页面是一个普通页面，也生成上面的事实。

最近分割的页面是唯一一个具有特殊信息的页面，即一个邻居指针。因此，一个带有邻居指针指示最近的分割的页面必须触发一些特殊事实的派生。由于这个旧节点为新节点提供了适当的“父事实”，如果想继续父、子父、祖先等的比喻，旧节点可以被称为“养父”。

图6.11说明了这种情况，节点b最近被分裂了。关于节点d的低栅栏键的事实不能从（未来）父节点a推导出。来自节点d的事实必须与来自节点b的事实相匹配。因此，节点b不仅在搜索逻辑上，而且在验证b树结构时，节点b就像新节点d的临时父节点。请注意，图6.11中的中间状态也可以在从b树中删除节点时使用，同样能够在任何时间、任何节点序列中，从而在任何媒体上执行完整和正确的b树验证。

除了验证一个b树的结构外，每个单独的页面都必须在提取事实之前进行验证，并且可能需要相互匹配多个b树的索引，e。g.，针对主索引中的适当标识符的辅助索引。页内

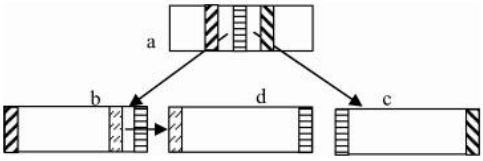


图6．11.在眨眼树中的栅栏键和验证。

验证是相当简单的，尽管有多少细节值得验证可能会令人惊讶。相互匹配多个索引非常类似于连接操作，并且可以使用所有标准的连接算法。或者，可以使用位图，两个错误相互掩盖的概率很小，如果位图表明存在错误，则需要第二次传递。　　对b树指标的自动修复研究不足。技术可能依赖于删除和重建整个索引，只为一个页面的恢复日志，或者调整一个页面以匹配相关页面页对修复算法、其能力及其性能的系统研究将对整个行业都很有用。

- b树指标的验证可以防止软件和硬件故障。所有的商业数据库系统都提供这样的实用程序。
- b树可以通过单一索引顺序扫描来验证，这可能由于碎片而昂贵。
- 基于磁盘顺序扫描的验证需要聚合从页面中提取的事实。位向量滤波器可以加快过程的速度，但如果发现不一致，则不能准确地识别不一致（由于可能的哈希冲突）。
- 如果节点携带栅栏键而不是邻居指针，查询执行可能（作为一个副作用）验证所有b树不变量。

6.8总结

总之，公用事业在数据库系统的可用性和总拥有成本中扮演着重要的角色。b树在索引结构中是独一无二的，因为大量的高效工具的技术是众所周知的和广泛实现的。新提出的索引结构不仅在查询处理和更新期间，而且在从索引创建到碎片整理和索引验证的实用程序操作过程中，都必须与b树的性能和可伸缩性进行竞争。

7

高级关键结构

正如前面几节所演示的，为了完全支持b树等数据库索引结构，需要大量的规划和编码。没有其他索引结构受到数据库研究人员和软件开发人员的广泛关注。然而，通过仔细和创造性地构建b树键，可以在核心b树代码中通过很少的修改来启用额外的索引功能。本节调查了其中的几个问题。

本节不考虑计算列上的索引，i。e.，从同一表中的其他列派生出来的值，并在数据库模式中指定了一个名称。这些列将根据需要进行计算，并且不需要存储在表的主数据结构中。但是，如果在这样的列上有一个索引，则适当的键值将存储在此辅助索引中。

同样地，本节不考虑部分索引，i。e.，基于选择谓词的条目少于底层表中的行的索引。一个典型的谓词可以确保只对除Null以外的值进行索引。这两个主题，即计算列和部分索引，都与b树中的高级关键结构正交。

前面的讨论（第2.5节）给出了一个不寻常的b树键的例子，即在哈希值上实现为b树的哈希索引。B-tree代码中的一些小的调整模拟了传统上与哈希索引相关联的主要性能好处，即在最坏的情况下是单个I/O，在哈希目录中直接计算地址，以及有效的键值比较。第一个可以通过在缓冲池中固定一个非常大的根页面来模拟（非常类似于一个大的散列目录）；另外两个好处可以通过适当的键值来反映，包括穷人的规范化键。

- 具有高级键结构的b树保留了b树的所有优势，例如，键范围锁定的理论和实现、日志记录和恢复的优化、高效的索引创建以及其他用于高效数据库操作的实用工具。
- 与传统哈希值的b树相比，哈希值具有许多优点。

.17多维UB树

根据它们的本质，b型树只支持一个单一的排序顺序。如果使用多个键列，它们可能形成主排序键、次要排序键等的层次结构。在这种情况下，限制前导键列的查询比不限制前导键列的查询性能更好。但是，如果关键列表示空间中的维度，则e。g.，在几何空间中，查询只能通过范围谓词来限制前导列，或者根本不限制前导列。莱斯利等人。[82]描述了在这种情况下访问b型树的复杂算法。

另一种方法是基于空间填充曲线将多维数据投影到单维上。空间填充曲线设计的主要权衡一方面是概念性和计算上的简单性，另一方面是保持局部性，从而提高搜索效率。空间填充曲线的最简单构造首先将每个维度映射到一个无符号整数，然后从这些整数中插入单独的位。当在二维空间中绘制成一条线时，这条空间填充曲线类似于嵌套的Z形状，这就是为什么它也被称为z阶。这是设计的基础设计

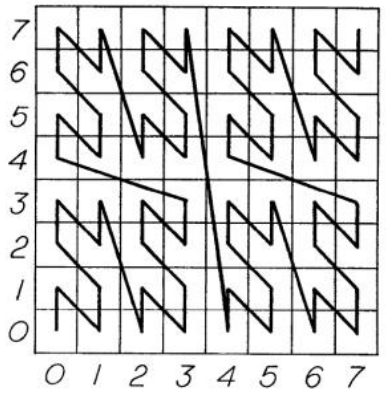


图7.1z阶曲线。

探针[106]和跨基[109]中的多维索引和查询处理。这个莫顿曲线的替代品包括希尔伯特曲线和其他曲线。

图7.1（从[106]复制）显示了z阶曲线。原始的x和y坐标有3个位，因此有8个不同的值；交错的z值包含6个位，因此包含64个点。“Z”形状在3个尺度上重复。这种技术适用于任意数量的
尺寸，而不仅仅是如图7.1所示的二维空间。

ub树是这种z值上的b树索引。针对原始维度的每个点和范围查询都被映射到沿着z曲线的适当间隔。该映射的关键组件是确定z曲线进入和退出由查询谓词定义的多维范围时的z值。已知的算法在原始维数上是线性的，其分辨率(i。e.，z值中的位数)，以及入口和退出点的数量[109]。

除了多维空间中的点外，空间填充曲线、z阶映射和ub树还可以通过将起始点和结束点作为单独的维度来为多维矩形（方框）提供索引。换句话说，ub树不仅可以索引有关点的信息，还可以索引有关区间和矩形的信息。空间和时间（包括时间间隔）都可以用这种方式进行索引。对于移动的对象，位置和速度（在每个空间维度中）可以被视为

单独的尺寸。最后，如果需要的话，甚至在位置或速度上的精度也可以被索引。不幸的是，基于空间填充曲线的索引会随着维数的增加而失效，就像传统的b树包含许多列，但在查询谓词中只指定了少数列时，b树应用程序的性能会受到影响一样。

- z值（或其他空间填充曲线）提供了一些具有b树的所有优势的多维索引。
- 专门化的多维索引的查询性能可能较好，但b型树的加载和更新性能并不容易匹配。

.27分区B树

如前面关于批量插入的部分所讨论的

在图6.6中，分割的b型树的本质是[43][1]是通过一个人工的领先关键字段来在单个b树中维护分区。对数据库用户隐藏分区和人工引导键字段。它们的存在是为了加快对b型树的大型操作，而不是为了携带任何信息。使用外部合并排序中著名的合并步骤对分区进行优化。默认情况下，相同的单个值会出现在b树中的所有记录中，并且大多数特定的技术依赖于利用多个替代值，但只是暂时的。如果一个表或索引用多个b树表示，则应该为每个b树分别定义人工引导键字段。

领先的人工键列有效地定义了单个b树中的分区。每个现有的不同值都隐式地定义了一个分区，当插入和删除记录时，分区会自动出现和消失。这种设计与传统的水平分区不同，其重要之处在于为每个分区使用单独的b树：设计的大部分优势依赖于非常动态地创建和删除的分区（或领先的人工键列中的不同值）。在传统的分区实现中(使用

[1] 本小节中的文本是改编自本参考文献中的。

多个b树)，创建或删除一个分区是更改表的模式和目录条目，这需要锁定表的模式或目录条目，因此排除并发或长时间运行的用户对表的访问，以及强制重新编译缓存的查询和更新计划。如果在单个b树中创建和删除分区，那么平滑的连续操作相对容易实现。令人惊讶的是，这种简单的技术在数据管理软件及其现实世界的使用中可以帮助解决多少问题。

首先，它允许将外部合并排序中的所有运行放到单个b树中（将运行号作为人工的领先关键字段），这反过来又允许改进异步预读和自适应内存使用。在SAN和NAS环境中，通过利用异步预读取来隐藏延迟非常重要。对于条带化磁盘，预测多个I/O操作是很重要的。最后，在非常大的在线数据库，动态增长和收缩资源用于一个操作是非常重要的，和提出的变化允许这样做甚至暂停的极端操作，让一个操作使用机器的整个内存和整个处理器在否则空闲的批处理窗口。虽然排序用于有效地构建b树索引，b树用于避免排序费用，减少查询处理过程中的搜索费用，但排序和b树之间的互利关系可以走得更远。

其次，分区的b树可以大大减少新创建的索引可用于查询回答之前的等待时间，至少减少两倍。虽然索引的初始形式不执行以及最后，完全优化索引或传统索引，至少它是可用的查询和允许替换表扫描索引搜索，导致更好的查询响应时间以及较小的“锁定足迹”，从而减少死锁的可能性。此外，该索引可以从初始形式逐步改进到最终的完全优化形式，这与传统索引创建后的最终形式非常相似。因此，最终的索引在性能上与离线或使用传统的在线方法创建的索引非常相似；主要的区别是减少了一半（或更好）

在创建新索引的决定和它对查询处理的第一个有益影响之间的延迟。

第三，到目前为止，向一个完全索引的大型数据仓库添加大量数据造成了删除和重新构建所有索引或一次更新所有索引之间的困境，这意味着随机插入、性能差、大日志量和大的增量备份。分区b树在没有特殊的新数据结构的大多数情况下解决了这种困境。加载操作简单地向每个受影响的索引添加多个新分区；这些分区的大小由加载操作期间内存中运行生成的内存分配控制。更新（插入和删除）可以附加到一个或多个新分区中的现有b树中，以便在最早方便的时候集成到主分区中，此时可以将删除应用于适当的旧记录。当然，附加分区是差分文件[117]主题的另一个变体。分区b树的批量维护减少了总体更新时间；此外，如果主分区的页面完全充满压缩记录，可以提高整体空间需求；如果主分区保持未分割，页面优化，则会减少查询执行时间。g.，插值寻找

虽然分区的b树实际上包含多个分区，但任何查询都必须搜索所有分区。用户查询不太可能将自己限制为分区的子集甚至单个分区（查询语法甚至不允许）。另一方面，即使已经有较新的分区可用，历史查询或“作为”查询也可能映射到单个分区。但是，通常情况下，必须搜索所有现有的分区。由于分区是在其他标准的b树实现中使用人工的前导关键字段实现的，这相当于查询没有限制传统的多列b树索引中的前导列。针对这种情况的有效技术是已知的，在这里不再进一步讨论，[82]。

分区可以保持最初保存的状态，也可以被合并。合并可能是渴望的。g.，一旦分区的数量达到一个阈值，就会立即合并。g.，当有空闲时间时进行合并。g.，合并回答实际问题所需的键范围

问题后者被称为自适应合并[53]。合并不是在准备查询处理时合并分区，而是可以集成在查询执行中。e.，这是查询执行的一个副作用。因此，即使键范围留在查询谓词中的参数中，这种技术也只合并实际查询的键范围。所有其他关键字范围都保留在初始分区中。当工作负载和访问模式随着时间的变化时，它们已经准备好执行查询和索引优化。

- 在分区的b树中，分区由一个人工的领先关键字段来识别。分区只通过插入和删除具有适当键值的b树条目来出现和消失，而不更新目录。
- 分区的b树对于有效的排序很有用。g.，深度预读)，索引创建(e。g.，早期的查询处理）、批量插入（内存中的数据捕获）和批量删除（受害者准备）。
- 一旦所有分区都被合并，查询性能就等于传统的b型树，这是默认状态。

.37合并索引

正如其他人所观察到的，“减少物理I/o数量的优化技术通常比那些提高执行I/O[70]效率的优化技术更有效。”人们普遍认为，聚类相关的记录需要在记录之间使用指针。具有记录集群的关系数据库管理系统的一个示例是星爆[20]，它在相关记录之间使用隐藏的指针，并在插入、删除和更新期间影响它们的自动维护。该技术只服务于表及其主存储结构，而不服务于次索引，而且它需要使用外键完整性约束定义的多对一关系。

聚类次要指标的可取性很容易体现在多对多的关系中，如“招生”，如“课程”和“学生”之间的多对多关系。为了支持从学生到课程、索引到索引、记录到记录和从课程到学生的导航，注册

表需要至少两个索引，其中只有一个可以是主要索引。然而，为了在两个方向上有效地访问数据，最好将一个注册索引与学生记录和一个注册索引与课程记录聚集起来。

合并索引[49][2]是包含多个传统索引的b型树，并根据一个共同的排序顺序交错排列它们的记录。在关系数据库中，合并索引实现了相关记录的“主细节聚类”。g.，订单和订单的细节。因此，合并后的索引将去规范化从表和行的逻辑级别转移到索引和记录的物理级别，这是一个更合适的位置。对于面向对象的应用程序，与传统索引相比，集群可以将相关表中连接行的I/O成本降低到一部分，并对缓冲池需求产生额外的有益影响。

图7.2显示了这样一个b树中的记录的排序顺序。排序顺序保持相关记录的位置；记录之间不需要额外的指针。在其最有限的形式中，主细节聚类结合了两个次要索引，e。g.，将两个行标识符列表与每个键值相关联。或者，主详细集群可以合并两个主索引，但不允许任何次要索引。合并索引的设计适应了单个b树中主索引和次索引的任何组合，从而支持对整个复杂对象进行聚类。此外，表、视图和索引集可以不受限制地发展。集群列集也可以自由演化。一个关系查询处理器可以搜索和

…
订单第4711号，客户“史密斯”，……。
订单4711，第1行，数量3，...。
订单4711号，第2行，数量1号，……。
订单4711，第3行，数量9，……。
订单4712号，客户“琼斯”，……。
订单4712，第1行，数量1，……。
…

图7.2在合并索引中的记录顺序。

[2] 本小节的文本改编自本参考文献。

就像更新传统索引一样更新索引记录。有了这些能力，所提出的设计可能最终将一般的主细节聚类带到传统数据库及其在性能和成本上的优势。

为了简化合并索引的设计和实现，关键的第一步是将b-树结构的实现从其内容中分离出来。一种技术是使用标准化键，图3.4前面讨论和说明，这样b树结构只管理二进制记录和二进制键。在合并索引中，b树中从多列键到二进制搜索键的映射比传统索引要复杂一些，特别是如果需要在任何时候添加和删除任何索引，以及单个索引可能有不同的键列。因此，必须设计一个从索引中的键到b-树中的字节字符串的灵活映射。如图7.3所示，指示关键列的域并在实际关键字段之前的标记可以很容易地实现这一点。换句话说，当为合并的索引构造规范化键时，域标记和字段值交替出现到并包含索引的标识符。

在实践中，与图7.3中所示不同的是，域标记将是一个小数字，而不是一个字符串。可以将域标记与Null指示符（在图7.3中省略）结合起来，从而实现所需的排序顺序，但实际值将存储在字节边界上。类似地，索引标识符将是一个数字，而不是一个字符串。

场值	**字段类型**
“客户标识符”	域标签
123	数据值
“订单号”	域标签
4711	数据值
“索引标识符”	域标签
圣职订单密钥”	标识符值
“2006/12/20”	数据值
“紧急”	数据值
…	数据值

图7.3合并后的索引中的b形树记录。

B-树记录中的所有字段都不需要域标记。它们只用于关键列，更具体地说，只用于合并索引中集群所需的主要关键列。在这些主要的键列之后是一个特殊的标记和记录所属的单个索引的标识符。例如，在图7.3中，键值和索引标识符只有两个域标记。如果不需要对顺序细节的行号进行集群，那么只有到达顺序号的关键字段才需要域标记。因此，合并索引的每个记录存储开销很小，并且可能确实隐藏在字段与字边界的对齐中，以实现快速内存处理。每个记录的2-4个单字节域标记的开销在实践中可能是典型的。

- 将多个索引合并到一个b树中，可以提供具有b树的所有优点的主细节聚类。单个b树可以包含任意数量的表的任意数量的主索引和辅助索引。
- b树键替换域标记和值，并包括索引标识符。
- 合并后的索引允许传统标准形式的表具有自由去正规化的性能。
- 合并后的索引在具有深度存储层次结构的系统中特别有价值。

7.4柱式存储

[3]柱存储被提议作为大型扫描的性能增强，因此用于特殊查询和数据挖掘可能找不到适当索引的关系数据仓库。缺少索引可能是由于查询谓词中复杂的算术表达式，或不可接受的更新和加载性能。柱状存储的基本思想是存储一个不采用基于行的传统格式的关系表，这样扫描单个列就可以充分利用从磁盘获取的页面或从内存获取的缓存行中的所有数据字节。

[3] 本节中的某些文本是从[47]中复制过来的。

如果每个列按其包含的值排序，则必须使用某种逻辑行标识符标记值。组装整个行需要连接操作，这可能太慢和太昂贵。为了避免此费用，表中的列必须以相同的顺序存储。[4]这个顺序可以称为表中行的顺序，因为没有一个索引来确定它，而b-树可以使用几乎为零附加空间的标记来实现列存储。

这些标记在很多方面都与行标识符相似，但是这些标记和传统的行标识符之间有一个重要的区别：标记值不是物理上的，而是逻辑上的。换句话说，它们不捕获或表示一个物理地址，如一个页面标识符，也没有办法从一个标记值来计算一个页面标识符。如果存在将标记值映射到行地址和返回的计算，则此计算必须假定可变长度列的最大长度。因此，存储空间将被浪费在部分或全部的垂直分区中，这将与柱状存储的目标相矛盾，即非常快的扫描。

由于大多数数据库管理系统的大部分或所有索引都依赖于b树，因此对传统存储结构的重用和适应主要意味着对b树的适应，包括它们的空间管理和对搜索键的依赖。为了确保行及其列在所有b型树中以相同的顺序出现，所有索引中的搜索键必须相同。此外，为了实现目标，搜索键的存储需求实际上为零，这似乎相当违反直觉。

所需技术的本质是非常简单。行按照添加到表中的顺序按顺序编号来分配标记值。请注意，标记值标识表中的行，而不是单个分区或单个索引中的记录。每个标记值在每个索引i中恰好出现一次。e.，它为表中的每一列与一个值配对。所有的垂直分区都以b树格式存储，标记值作为主键。重要的方面是如何将这个主要密钥的存储减少为零。

[4] 请注意，实例化视图可能以不同的排序顺序存储。如果是这样，那么行在实体化视图中的位置当然对于检索基表中的其他信息并没有用处。

每个b树页面中的页面头存储该页面上所有条目中最低的标记值。每个b树条目的实际标签值是通过添加这个值和页面内条目的槽号来计算的。不需要将标记值存储在单个B树条目中；每个页只需要一个标记值。如果一个页面包含数十个、数百个甚至数千个b树条目，那么对于每个单独的记录，存储最小标签值的开销实际上为零。如果行标识符的大小为4或8字节，而b树节点的大小为8 KB，则每页行标识符会造成0.1%或更少的开销。

如果页面中的所有记录都有连续的标记值，这种方法不仅解决了存储问题，而且还将索引中对特定键值的“搜索”减少为一点点算术，然后直接访问所需的b树条目。因此，在这些b型树的叶页中实现的访问性能甚至可以比在插值搜索或哈希索引中实现的访问性能更好。

图7.4说明了一个包含2列和3行和列存储的表。圆括号中的值表示行标识符或标记。图的右侧显示了两个磁盘页，每列对应一个。每个页面的列标题（虚线）显示行数和页面中的最低标记。

到目前为止，这些考虑事项只覆盖了b型树的叶页。当然，也需要考虑上面的索引页面。幸运的是，它们只引入了适度的额外存储需求。分支节点中的存储需求由键大小、指针大小和可变长度条目的任何开销决定。在这种情况下，键的大小等于行标识符的大小，通常是4或8个字节。指针的大小等于一个页面标识符，通常也是4或8个字节。管理变量函数条目的开销，尽管下面的b树索引并不严格需要

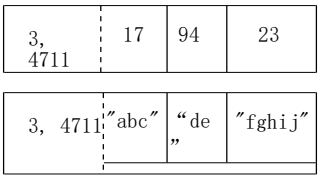
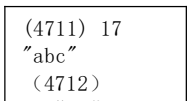


图7.4表格和柱状存储。



考虑到，通常是4个字节的一个字节偏移量和一个长度指示符。因此，每个分隔符条目的的存储需要为8到20字节。例如，如果节点大小为8 KB，且平均利用率为70%，则平均b树扇出率为280到700。因此，所有上b树页面一起需要的磁盘空间小于或等于0.3%的磁盘空间，这在实践中是可以忽略不计的。

与其他存储垂直分区的方案相比，所描述的方法允许跨多个分区以相同的顺序非常有效地存储可变长度的值。因此，使用多路合并连接来组装表中的整个行是非常有效的。此外，单个行的组装也非常有效，因为每个分区都在行标识符上建立了索引。所有传统的b树索引优化都适用，e。g.，非常大的b树节点和插值搜索。请注意，在均匀的数据分布之间的插值搜索实际上是即时的。

图7.5说明了柱状存储的b-树的值，特别是当列值的大小可以自然变化或由于压缩而变化时。字母字符串是实际值；虚线框表示带有记录计数和最低标记值的页头。b型树的上层表示它们各自的子树中的最低标记值。每个页面具有不同记录计数的叶页可以很容易地进行管理，并且通过查找标签来组装单个行可以非常有效。根据键值的分布及其大小，进一步的压缩可能是可能的，并且经常用于具有柱状存储的关系数据库管理系统。

- 通过适当的压缩，使运行长度编码适应于一系列的行标识符，柱状存储可以基于b-树。

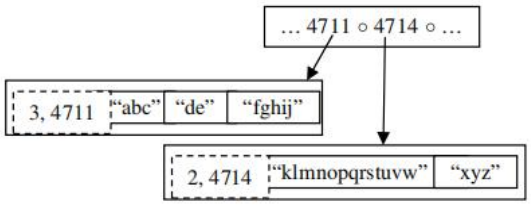


图7.5使用b型树的柱状存储。

7.5大值

除了包含许多记录的b树外，每个记录都小于单个叶页，b树还可以表示大型二进制对象或具有多个字节的字节字符串。在这种情况下，叶节点包含数据字节，而分支节点包含大小或偏移量。叶节点中的数据字节可以像传统的b树索引一样被划分为记录，也可以不需要任何额外的结构，i。e.，字节字符串。在后一种情况下，大部分或所有的大小信息都保存在分支节点中。大小或偏移量可作为分支节点中的分隔符键。为了最小化更新操作的工作量和范围，特别是插入和删除单个字节或子字符串的操作，大小和偏移量将在本地计算，i。e.，在节点及其子对象中，而不是在整个大型二进制对象中。

改编自[18,19]的图7.6说明了这些想法。在这个
例如，该对象的总大小为900字节。叶级别上的树节点表示字节范围。这些值只显示在叶节点中，以便在这里进行说明；相反，叶节点应该包含实际的数据字节，并且可能还包含有效字节的本地计数。树的分支节点表示大对象内的大小和偏移量。在图的左半部分和根节点中的键值相当明显。这棵树中最有趣的条目是右父节点中的键值。它们表示其子节点中的有效字节的计数；它们并不表示这些字节在整个对象中的位置。为了确定绝对位置，需要将从根添加到叶的关键值。例如，在最右的叶节点中，最左边的字节的绝对位置为
421 + 365 = 786.

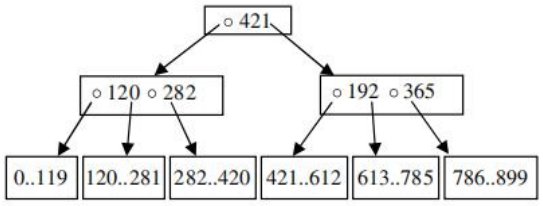


图7.6作为小子字符串的树的大字符串。

类似地，搜索可以在节点内使用二进制搜索（甚至是插值搜索），但必须对上节点内的键值进行调整。例如，在搜索字节698时的根到叶遍历可以在右父节点中使用二进制搜索，但仅在从搜索键（698）中减去根（421）中的键值之后。e.，在右父节点内搜索键值698-421 = 277，并找到192到365之间的间隔。对于该叶，本地字节位置277-192 = 85对应于全局字节位置698。

在某些叶节点中插入或删除某些字节只影响沿一个根到叶路径的分支节点。例如，在位置30处删除10个字节会减少图7.6中的值120、282和421。尽管这样的删除改变了右子树中数据字节的绝对位置，但右父节点及其子节点保持不变。类似地，插入或删除整个叶节点及其数据字节只影响单个根到叶路径。沿路径对关键值的维护可以是搜索受影响叶子的初始根到叶遍历的一部分，也可以遵循叶节点中的数据字节的维护。使用与传统b树非常相似的算法，所有节点都可以保持50-100%的完整。兄弟节点之间的积极的负载平衡可以延迟节点的分割。表示大型对象的b树比标准的b树更支持这种拆分前合并策略，因为父树包含足够的信息来决定兄弟叶是否有希望成为负载平衡的候选对象。

. 使用相对字节偏移量作为键值，b-树可以用于存储跨越多个页面的大型对象，甚至允许有效地插入和删除字节范围。

7.6记录版本

许多应用程序需要“事务时间”和“真实世界时间”的概念。e.，关于一个事实何时被插入数据库以及该事实在现实世界中何时有效的信息。这两个时间的概念都允许有时被称为“时间旅行”，包括“这个查询昨天会产生什么结果？”和“什么是什么”

你现在已经知道昨天的情况了吗？”这两种类型的查询及其结果都可能具有法律重要性。[5]

前一种查询类型也用于并发性控制。在这些方案中，每个事务的同步点都是它的开始时间。换句话说，事务可以在可序列化的事务隔离中运行，但等效的串行调度按事务的开始时间排序，而不是像普通锁定技术那样按结束时间排序。对于长时间运行的事务，它可能需要提供一个过时的数据库状态。这通常是通过保留更新记录的旧版本来实现的。因此，该技术的名称是多版本并发控制[15]。与之密切相关的是快照隔离[13,32]的概念。

由于大多数应用程序中的大多数事务都需要最新的状态，因此一种实现技术会更新数据库记录，如果旧事务需要，则使用缓冲池中的第二个副本回滚数据页。回滚逻辑与事务回滚的逻辑非常相似，只是将它应用于数据页面的副本。事务回滚依赖于每个事务的日志记录链；数据页的有效回滚需要与每个数据页相关的日志记录链，i。e.，每个日志记录都包含一个指向同一事务的先前日志记录的指针，以及另一个指向与同一数据页相关的先前日志记录的指针。

另一种设计依赖于每个逻辑记录的多个实际记录，i。e.，版本的记录。版本控制可以仅应用于表的主数据结构e中进行管理和管理。g.，主索引，或者它可以在每个数据结构中进行管理，i。e.，每个辅助索引、每个物化视图等。如果一个设计在空间或努力方面增加了大量的开销，那么前一种选择可能更合适。为了使数据结构和算法最简单和统一，似乎希望减少开销，以便版本控制可以应用于每个数据结构，e。g.，数据库中的每个b树索引。

[5] 迈克尔·凯里用来解释需要在数据库中编辑大型对象与以下玩1970年代的美国总统政治：“假设你有一个音频对象代表记录电话对话，你觉得有必要删除18分钟。. . 在玩弄20世纪80年代的美国总统政治时，人们可能会说：“数据库知道什么，他是什么时候知道的？”

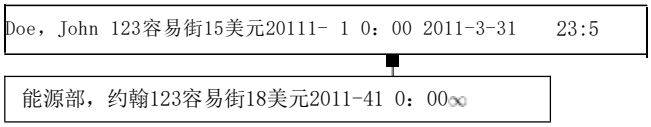


图7.7版本记录、开始时间、结结束时间和指针。

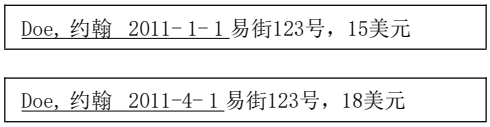


图7.8以开始时间作为键后缀的版本记录。

图7.7说明了对记录版本控制的一些设计如何标记每个版本记录的开始时间、结束时间以及指向版本链中下一个记录的指针。在本例中，更改单个小字段以反映工人的小时工资增长，需要一个包含所有字段和标签的全新记录。在每个索引条目中很少有字段的辅助索引中，另外三个字段会造成很高的开销。但是，通过对b树键进行适当的修改，就可以避免这三个字段中的两个。此外，新版本可能需要的空间比完成版本化记录的新副本要少得多。

具体来说，如果开始时间提供了b-树键中最不重要的部分，那么相同的逻辑记录（具有相同的用户定义的键值）的所有版本都是记录序列中的邻居。不需要指针或版本链，因为版本的序列只是b-树条目的序列。如果一个版本的开始时间被解释为先前版本的结束时间，则可以省略结束时间。在删除一个逻辑记录后，需要一个具有适当的开始时间的幽灵记录。只要此幽灵记录包含有关逻辑记录的历史记录和最终删除的信息，它就必须得到保护。

图7.8说明了该设计。记录键上有下划线。开始时间是版本记录中唯一需要的附加字段，避免了对带有时间戳的版本记录的最简单设计所需的3个附加字段中的2个，并且自然地确保了版本记录所需的位置。

开始时间可以通过在每个b形树的叶子中存储一个等于页面中最古老的记录版本的基本时间来进行压缩。在这种情况下，每个记录中的开始时间用与基础时间的差值表示，希望这是一个小值。换句话说，附加到b树键的附加键字段可以使用少量字节的记录版本控制，甚至可能是单个字节。

此外，还可以通过显式地只存储版本与其前任版本之间的差异来压缩记录内容。为了最快地检索和汇编最新版本，版本记录应该存储版本与其后继版本之间的差异。在这种情况下，检索旧版本需要多个记录，这有点类似于日志记录的“撤销”。或者，也可以使用实际的日志记录，从而导致类似于基于页面回滚的设计，但也应用于单个记录。

图7.9说明了这些技术。页眉中的一个字段表示当前页面上最古老的版本记录的时间。
每个记录都存储了与这个基本时间之间的差异
比完整的时间戳。此外，不变的字段值不会重复。版本记录的顺序是这样的，即当前的记录是最容易获得的，而较旧的版本可以通过正向的本地扫描来构建。在图中，显示了先验值的绝对值，尽管对于许多数据类型，可以使用一个相对值，e。g.，″-$3.″进一步的优化和压缩，e。g.，前缀截断，可酌情使用。

如果数据库中的“时间旅行”可以被限制，例如在过去的一年内，所有超过这个间隔的版本记录都可以被解释为幽灵记录。因此，它们就像传统的幽灵记录一样，会被清除和空间回收，具有所有的规则

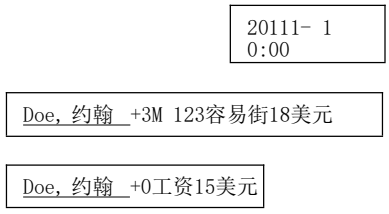


图7.9带有压缩功能的版本记录。

以及在幽灵删除期间对锁定和日志记录的优化。当这样删除了逻辑记录的所有其他版本时，也可以删除指示删除逻辑记录的鬼影记录，并可以回收其空间。

如果辅助索引中的版本控制与表主索引中的版本控制无关，则辅助索引中的指针只能引用主索引中相应的相应逻辑记录（唯一的用户定义的键值）。事务上下文必须为主索引i中的剩余关键字段提供一个值。e.，需要使用来自主索引的记录的时间值。例如，由于两天前的更新，次索引可能包含两个版本，而主索引可能包含三个版本，因为一天前的非索引字段只更新了三个版本。事务可能查询四天前的索引，并确定旧索引条目满足查询谓词；从辅助索引到主索引导致所有三个版本记录，其中事务选择四天前有效的版本记录。如果大多数事务需要最新的记录版本，如果正向扫描比向后扫描更有效，那么可能是
在所有版本中首先存储这个记录很有用。e.，通过减少开始时间来对版本记录进行排序，如图7.9所示。

- 向每个键值附加一个版本号，并尽可能地压缩相邻的记录，从而使b树变成一个在空间（存储）和时间（查询和更新）上高效的版本存储。

7.7总结

总之，b型树可以解决各种各样的索引、数据移动和数据放置问题。并不是每个问题都需要更改索引结构；通常，一个精心选择的键结构可以在b树索引中实现新的功能。具有新设计的关键结构的b树保留了b树的传统操作优势。g.，通过排序、密钥范围锁定、生理日志记录等方式创建索引。因此，当需要新的功能时，可以启用

通过b树索引的新键结构实现这个功能可能比定义和实现新的索引结构更容易。

高级键结构可以以各种方式从用户定义的键中派生出来。前面的讨论包括添加人工前缀（分区的B树）或后缀（记录版本）、交错用户键（UB树）或人工关键组件（合并索引）。虽然这个替代关键增强列表看起来很详尽，但未来可能会发明新的关键结构，以扩展索引的能力，而不反映在研究、开发和测试方面已经花费在b树上的努力。

显然，上面讨论的许多技术都可以结合起来。
例如，合并的索引可以通过将对象或属性标识符与单个对象内的偏移量组合来容纳多个复杂对象。此外，还可以在ub树或合并索引中添加一个人工的领先关键字域，从而将高效的加载和增量索引优化与多维索引或主细节聚类结合起来。类似地，合并的索引不仅可以包含（并因此包含集群）传统记录（来自各种索引），还可以包含位图或大型字段。进行组合的机会似乎是无穷无尽的。

8

总结和结论

综上所述，b型树的核心设计在40年来一直保持不变：平衡树、页面或其他I/O单元作为节点，高效的根到叶搜索、分割和合并节点等。另一方面，大量的研究和开发改进了b树的各个方面，包括多维数据等数据内容、多维查询等访问算法、每个节点内的数据组织如压缩和缓存优化、闭锁分离等并发控制、多级恢复等恢复等。

对b树索引最重要的技术之一似乎是：

- 使用排序和仅附加b树维护来高效地创建索引
- 具有可变大小记录的节点内的空间管理
- 归一化键
- 前缀和后缀截断
- 数据压缩，包括保持顺序的压缩
- 围栏钥匙

- 逻辑内容和物理表示的分离——用户事务与系统事务、锁定与锁定等。
- 真正同步原子性的键范围锁定（可序列化性）
- 眨眼-带有临时“养父母”的树
- 用于删除和插入的幽灵记录
- 未记录（但具有事务性）的索引操作，特别是索引创建
- 覆盖查询处理过程中的索引和索引交集。
- 在重复搜索之前对搜索键进行排序(e。g., 在索引嵌套循环连接)的性能和可伸缩性
- 优化的更新计划，按索引排列的更新
- 增量加载的批量插入（和删除）
- B树验证

Gray和路透社认为“b树是数据库和文件系统中最重要的访问路径结构”[59]。这一说法在今天似乎仍然正确。由于闪存的出现，b树索引很可能在关系数据库中获得新的重要性。与传统的磁盘存储相比，快速访问延迟允许更多的随机I/O操作，从而在全带宽扫描和b树索引搜索之间转移盈亏平衡点，即使该扫描具有柱状数据库存储的好处。我们希望本教程和b树技术的参考将能够、组织和促进b树的索引技术的研究和发展。

确认信息

吉姆·格雷、史蒂文·林德尔和许多其他行业的同事激发了我对存储层概念的兴趣，并逐渐以急需的耐心教育了我。如果没有他们，这份调查就不会被写出来。迈克尔·凯里对两份草稿给出了反馈，敦促更加强调基本知识，刺激部分比较Btrees和哈希索引，并迫使澄清许多单独的观点和问题。鲁道夫·拜耳建议包含空间索引和ub树，这导致包含了关于高级关键结构的第7节。塞巴斯蒂安·布希尔和迈克尔·凯里都询问了唱片的版本控制，这导致了第7.6节的加入。a数据库中的基础和趋势的匿名审稿人提出了许多改进。阿纳斯塔西娅·艾拉玛基给出了建议和鼓励。巴布·彼得斯和库野春美对文本提出了许多改进。

参考文献

[1] V.N.Anh和A。莫法特，“使用64位单词的索引压缩”，“软件：实践与经验”，第1卷。40岁，没有。2，pp.131 - 147，2010.

[2] G.Antoshenkov，D.B.Lomet和J。“保持秩序的压缩”，国际数据工程国际会议，第3页。655 - 663，1996.

[3] R.Avnur和J。M.赫勒斯坦，“涡流：连续自适应查询处理”，数据管理特殊兴趣小组，页。261 - 272，2000.

[4] S.ăBchle和T。ăHrder，“XML锁协议背后的真正性能驱动因素”，DEXA，pp。38 - 52，2009.

[5] L.N.Bairavasundaram，M.伦格塔，N。阿格拉瓦尔。C.阿帕西杜索，R。H.阿帕西-杜索和M。M.“分析磁盘指针损坏的影响”，《可靠系统与网络》，第3页。502 - 511，2008.

[6] R.拜耳，“多维索引的通用b树：一般概念”，全球计算及其应用，pp。198 - 209，1997.

[7] R.拜耳和E。M.“大型有序指数的组织与维护”，签名研讨会，页。107 - 141，1970.

[8] R.拜耳和E。M.“大型有序索引的组织与维护”，《信息学报》，第1卷。1，pp.173 - 189，1972.

[9] R.拜耳和M。“b树操作的并发”，《信息学》，卷。9，pp.1 - 21，1977.

[10] R.拜耳和K。“前缀b树”，数据库系统事务，卷。2、没有。1，pp.11 - 26，1977.

[11] M.A.本德，E。D.DemaineM。法拉赫-科尔顿，“缓存无关的b树”，SIAM计算杂志（SiAmomcomp），卷。35岁，没有。2，pp.341 - 358，2005.

[12] M. A.本德和H。胡，“自适应打包内存阵列”，数据库系统上的ACM事务，卷。32岁，没有。4, 2007.

[13] H.Berenson, P.A.伯恩斯坦，J。灰色，J。梅尔顿，E。J.奥尼尔和P。E.奥尼尔，“对ANSI SQL隔离水平的批判”，数据管理特别兴趣小组，第3页。1 - 10, 1995.

[14] P. A.伯恩斯坦和DM。.-W.邱，“使用半连接解决关系查询”，ACM期刊，第1卷。28日，没有。1, pp. 25 - 40, 1981.

[15] P. A.伯恩斯坦，五。哈兹拉科斯和N。《数据库系统中的并发控制和恢复》。艾迪生-韦斯利，1987年。

[16] D. 比顿和D。J.德维特，“大数据文件中的重复记录消除”，数据库系统上的ACM事务，第1卷。8、没有。2, pp. 255 - 265, 1983.

[17] P. A.Boncz, M.L.Kersten和S。“打破MonetDB中的记忆墙”，ACM的通信，卷。51岁，没有。12, pp. 77 - 85, 2008.

[18] M. J.凯里，D。J.德威特，J。E.理查森和E。J.张建民，“外流可扩展数据库系统中的对象与文件管理”，国际大型数据库杂志，页。91 - 100, 1986.

[19] M. J.凯里，D。J.德威特，J。E.理查森和E。J.“面向对象迁移中的对象的存储管理”，参见面向对象的概念、数据库和应用程序。金和F。H.Lochovsky, eds.),ACM出版社和艾迪生-韦斯利出版社，1989年。

[20] M. J.凯里，E。J.Shekita, G.拉皮斯，B。G.林赛和J。麦克弗森，“星爆的增量连接附件》，《国际超大数据库杂志》，页。662 - 673, 1990.

[21] F. 张，J。院长，S。Ghemawat。C.谢谢，D。A.瓦拉赫，M。洞穴，T。钱德拉。菲克斯和R。E.“大表：结构化数据的分布式存储系统”，中国理论计算机科学，第1卷。26日，没有。2, 2008.

[22] J. 陈，D。J.德威特，F。田和Y。王：“互联网数据库的可扩展连续查询系统”，数据管理特别兴趣小组，第页。379 - 390, 2000.

[23] L. 陈，R。乔比和E。A.“合并r树：局部批量插入的有效策略”，第1卷。6、没有。1, pp. 7 - 34,
2002.

[24] S.陈，P。B.吉本斯，T。C.Mowry和G。“分形预取B”$^{+}$-树：优化缓存和磁盘性能，“数据管理特别兴趣小组，页。157 - 168, 2002.

[25] J. 程，D。Haderle, R.对冲，B。R.Iyer，T。梅辛格，C。莫汉和Y。王，“一个高效的混合连接算法：一个DB2原型”，数据工程国际会议，pp。171 - 180, 1991.

[26] HT..-周和D。J.德威特出版社，《关系数据库系统的缓冲区管理策略的评估》，《国际超大数据库杂志》，第3页。127 - 141, 1985.

[27] D. “无处不在的b树”，ACM计算调查，vol。11日，没有。2, pp. 121 - 137, 1979.

[28] W.M.“偏移值编码”，IBM技术披露公告，第1卷。20岁，没有。7, pp. 2832 - 2837, 1977.

[29] G.DeCandia, D.Hastorun, M.Jampani, G.卡库拉帕蒂。拉克什曼，A。Pilchin, S.西瓦苏布拉曼，P。Vossall和W。沃格尔斯，“发电机：亚马逊的关键价值商店”，操作系统原理研讨会，页。205 - 220, 2007.

[30] D.J.德威特，J。F.诺顿和J。汉堡，“嵌套循环”，并行和分布式信息系统，页。230 - 242, 1993.

[31] K.P.Eswaran, J.灰色，R。A.洛里和我。L.《数据库系统中的一致性和谓词锁的概念》，《ACM通讯》，第1卷。19日，没有。11, pp.624 - 633, 1976.

[32] A.Fekete，D。Liarokapis, E.J.O'Neil, P.E.奥尼尔和D。“使快照隔离可序列化”，《数据库系统上的ACM事务处理》，第1卷。30岁，没有。2, pp.492 - 528, 2005.

[33] P.M.费尔南德斯，“红砖仓库：主要阅读开放SMP平台的RDBMS”，数据管理特别兴趣小组，p。492, 1994.

[34] C.弗里德曼的博客，2008年10月07日，检索到2011年8月16日，在http:// blogs。msdn.com/craigfr/archive/2008/10/07/random-prefetching.aspx.

[35] P.加斯纳，G。M.洛曼，K。B.Spier和Y。王，“IBM DB2系列中的查询优化”，IEEE数据工程公告，第1卷。16日，没有。4, pp.4 - 18, 1993.

[36] G.H.Gonnet, L.D.罗杰斯和J。A.“插值搜索的算法与复杂性分析》，《信息学报》，第1卷。13, pp.39 - 52, 1980.

[37] N.K.乔文达拉朱，J。灰色，R。库马尔和D。“数据排序：用于大型数据库管理的高性能图形协同处理器排序”，数据管理特别兴趣小组，第页。325 - 336, 2006.

[38] G.“物理数据库设计的选择”，资料记录管理特别兴趣小组，第1卷。22日，没有。3, pp.76 - 83, 1993.

[39] G.“大型数据库的查询评估技术”，ACM计算机计算调查，第1卷。25日，没有。2, pp.73 - 170, 1993.

[40] G.“迭代器、调度器和分布式内存并行性”，《软件：实践与经验》，第1卷。26日，没有。4, pp.427 - 452, 1996.

[41] G.Graefe，“每一个°数据工程国际会议，页。349 - 358, 2001.

[42] G.“执行嵌套查询”，商业、技术和网络数据库系统，页。58 - 77, 2003.

[43] G.Graefe，“用分区b树排序和索引”，无类Inter域路由，2003年。

[44] G.“写优化b树”，非常大型数据库国际杂志，页。672 - 683, 2004.

[45] G.Graefe，“b树索引，插值搜索，和倾斜”，DaMoN，p。5, 2006.

[46] G.“在数据库系统中实现排序”，美国计算机管理管理计算调查，第1卷。38岁，没有。3, 2006.

[47] G.“b树的高效柱状存储”，数据记录管理特别兴趣小组，第1卷。36岁，没有。1, pp.3 - 6, 2007.

[48] G.“b树索引中的层次锁定”，面向商业，技术和网络的数据库系统，第3页。18 - 42, 2007.

[49] G.“使用合并索引的主细节集群”，关于信息的研究，第1卷。21日。3 - 4，pp.127 - 145，2007.

[50] G.“20年后的五分钟规则和闪存如何改变规则”，《ACM的通讯》，第1卷。52岁，没有。7，pp.48 - 59，2009.

[51] G.他，“b树锁定技术的调查”，关于数据库系统的ACM事务，v1。35岁，没有。3，2010.

[52] G.格雷夫，R。Bunker和S。库珀，“哈希加入并讨论了微软SQL服务器上的团队”，关于非常大型数据库的国际杂志，pp。86 - 97，1998.

[53] G.格雷夫和H。A.国野，《自我选择，自我调整，增量优化索引》，《扩展数据库技术》，第3页。371 - 381，2010.

[54] G.格雷夫和R。“b树完整性的有效验证”，《商业、技术与网络的数据库系统》，第3页。27 - 46，2009.

[55] G.格雷夫和M。J.“索引视图的交易支持”，数据管理特别利益小组，第3页。323 - 334，2004.（扩展版：惠普实验室技术报告HPL-2011-16）。

[56] J.灰色，“关于数据库操作系统的笔记”，在操作系统-一个高级课程。《计算机科学》第60期的课堂讲稿。拜耳。M.格雷厄姆和G。üSeegmller，eds.),柏林海德堡纽约：春天春天，1978年。

[57] J.灰色和G。“十年后，五分钟规则和其他计算机存储经验规则”，数据记录管理特别利益小组，第1卷。26日，没有。4，pp.63 - 68，1997.

[58] J.灰色，R。A.洛里，G。R.Putzolu和我。L.“共享数据库中锁的粒度和一致性度”，在IFIP数据库管理系统建模工作会议上，pp。365 - 394，1976.

[59] J.灰色和A。《事务处理：概念和技术》。摩根·考夫曼，1993年。

[60] J.Gryz，K.B.斯皮耶，J。郑和C。他说，“DB2中检查约束的发现与应用”，中国数据工程国际会议，第3页。551 - 556，2001.

[61] H.S.Gunawi，C.卢比奥-冈兹雷斯，A。áC.阿帕西杜索，R。H.arpitusseau和B。Liblit，“EIO：错误处理偶尔是正确的，”快速，pp。207 - 222，2008.

[62] A.“r树：空间搜索的动态索引结构”，数据管理特别兴趣小组，第3页。47 - 57，1984.

[63] L.M.哈斯，M。J.凯里，M。利夫尼和A。舒克拉，“寻求关于特别加入成本的真相”，VLDB期刊，第1卷。6、没有。3，pp.241 - 256，1997.

[64] R.A.汉金斯和J。M.“节点大小对高速缓存意识B性能的影响”$^{+}$-树，“签名，pp。283 - 294，2003.

[65] T.ä王建民，国立台湾大学医学研究所硕士论文，第3页。379 - 393，1975.

[66] T.ä“为关系数据库系统实现广义访问路径结构”，《数据库系统上的ACM事务，第1卷。3、没有。3，pp.285 - 298，1978.

[67] T.äHrder和A。“面向事务的数据库恢复原理”，ACM计算调查，第1卷。15日，没有。4，pp.287 - 317，1983.

[68] G.举行和M。“b树重新检查”，ACM通讯，第1卷。21日。2, pp.139 - 143, 1978.
[69] A.L.霍洛威，五。拉曼，G。斯瓦特和D。J.德威特，“如何用时间交换位：数据库扫描的压缩和带宽权衡”，数据管理特别兴趣小组，页。389 – 400, 2007.
[70] W.W.徐和A。J.“I/O优化和磁盘改进的性能影响”，IBM研究与开发杂志，第1卷。48岁，没有。2, pp.255 – 289, 2004.
[71]http://en.维基百科。org/wiki/Btrfs，于2009年12月6日检索。
[72] B.R.Iyer，“IBM的DB2DBMS中的硬件辅助排序”，COMAD，2005。海得拉巴
[73] I.Jaluta, S.Sippu和E。“平衡b链树的并发控制与恢复”，国际期刊，第1卷。14日，没有。2, pp.257 – 277, 2005.
[74] C.杰梅因。达塔和E。“支持大量数据仓库插入的新索引”，《超大数据库国际期刊》，第3页。235 – 246, 1999.
[75] T.约翰逊和D。沙莎，《插入、删除和修改”，《数据库系统原理》，页。235 – 246, 1989.
[76] J.R.J.乔丹。班纳吉和R。B.《蝙蝠侠》，《精密锁》，数据管理特别兴趣小组，第3页。143 - 147, 1981.
[77] R.《数据仓库工具包：构建的实用技术》
尺寸数据仓库。约翰·威利，1996年。
[78] R.“关系数据库中查询的优化”，Ph。D.论文集，凯斯西储大学，1980年。
[79] H.F.“锁定数据库系统中的原语”，《ACM杂志》，第1卷。30岁，没有。1, pp.55 – 79, 1983.
[80] K.克斯伯特，《数据银行系统人的研究”，第3卷。施普林格99号
1985.
[81] P.L.雷曼兄弟和S。B.Yao，“b树上并发操作的有效锁定”，数据库系统上的ACM事务，vol。6、没有。4, pp.650 – 670,
1981.
[82] H.莱斯利，R。Jain，D。伯兹尔和H。八脉大学，“多维b树的高效搜索”，国际超大数据库杂志，页。710 – 719, 1995.
[83] N.莱斯特。莫法特和J。Zobel，“文本数据库的高效在线索引构建”，《在数据库系统上的ACM事务处理，vol。33日，没有。3,
2008.
[84] Q.李，M。邵，五。马克尔，K。S.拜尔，L。S.科尔比和G。M.罗曼，“查询执行过程中的自适应重新排序连接”，数据工程国际会议，第3页。26 – 35, 2007.
[85] D.B.Lomet，“改进并发性的关键范围锁定策略”，《非常大型数据库的国际期刊》，第页。655 – 664, 1993.
[86] D.B.Lomet，“考虑缓存时的b树页面大小”，关于数据记录管理的特别兴趣组，第1卷。27日，没有。3, pp.28 – 32, 1998.