Lecture 9: 索引与哈希

Database

Author: Forliage

Email: masterforliage@gmail.com

Date: June 4, 2025

College: 计算机科学与技术学院

Abstract

本讲笔记聚焦数据库系统中索引与哈希技术的基本原理与实现方法，旨在介绍如何通过多种索引结构加速数据访问，并探讨针对不同应用场景的优化策略。首先，在“基本概念”部分阐述了索引的定义、搜索键与指针机制，以及评估索引性能的关键指标，包括访问时间、插入时间、删除时间和空间开销等。接着，“有序索引”章节深入分析了稠密索引与稀疏索引的设计与维护，以及二级索引与多级索引的构建方式；同时说明了索引在插入与删除操作时的更新流程，并区分了主索引与辅助索引在顺序扫描与随机访问中的优势与代价。

随后，“B+树索引文件”部分详细介绍了 B+ 树的结构特性，包括节点布局、非叶子节点与叶子节点的指针与键值组织，以及查找、插入与删除操作的具体算法与再平衡（分裂、合并、重分配）过程，并讨论了处理重复键的策略。紧接着，“写优化索引”部分着重讲解了日志结构合并（LSM）树的分层存储模型与分级归并机制，阐述了插入、查询、删除/更新的工作流程，以及 LSM 树在顺序写入性能与查询延迟之间的权衡；此外还概述了分步合并（Stepped-Merge）和缓冲树（Buffer Tree）等变种技术，以应对海量写入场景下的 I/O 优化需求。

最后，“SQL 中的索引定义”简要演示了使用标准 SQL 语句创建与删除索引的方法，并说明了唯一索引与非唯一索引的约束作用。通过本讲笔记的学习，读者能够系统掌握关系型数据库中常用的索引与哈希技术原理，为后续查询优化与存储管理的研究奠定坚实基础。

（该Abstract由ChatGPT-o4-mini-high生成）

Contents

1 基本概念

1.1 基本概念

为什么我们需要索引？索引机制用于加快对所需数据的访问速度，例如，图书馆中的作者目录。

搜索键：用于在文件中查找记录的属性或属性集。

索引文件由以下形式的记录(称为索引条目)组成： Search key + pointer

索引文件通常比原始文件小的多。

两种基本的索引类型（index文件中索引记录如何组织？取决于索引类型）：

- 有序索引：搜索键（索引条目）按排序顺序存储顺序
- 哈希索引：搜索键（索引条目）均匀分布，使用“哈希函数”跨“桶”进行操作。

1.2 索引评估指标

可高效支持的访问类型，例如：

- 属性中具有指定值的记录
- 或者属性值落在指定值范围内的记录。
- 如：哈希索引不适合'Between'查询条件，但有序索引适用

访问时间、插入时间、删除时间、空间开销

时间效率和空间效率是衡量索引技术的最主要指标，也是数据库系统组织和管理技术关注的焦点之一。

2 有序索引

2.1 基本概念

1.在有序索引中，索引条目按搜索键值排序存储，例如图书馆中的作者目录。

2.顺序排序文件：文件（数据文件）中的记录按搜索键排序。

3.主索引：一种索引，其搜索键等于创建该索引的顺序排序数据文件的搜索键（与对应的数据文件本身的排列顺序相同的索引称为主索引）。也称为聚集索引，主索引的搜索键通常是但并非一定是主码。

以下是一个例子：假设有一张学生信息表`Student`，其定义为：

```sql
CREATE TABLE Student (
    StuID CHAR(10) PRIMARY KEY,
    Name VARCHAR(50),
    Major VARCHAR(30),
    Grade INT
);
```

其中StuID是表的主码，且在逻辑上是唯一的。但在物理存储层面，我们并不一定要按照主码来排序。比如为了经常按学生姓名检索，我们可以让数据文件按Name排序。

4.非顺序文件没有主索引，但关系可以有主码。索引顺序文件：带有主索引的顺序有序文件。

5.辅助索引：一种索引，其搜索键指定的顺序与文件的顺序不同。

2.2 稠密索引文件

稠密索引：文件中每个搜索键值都有对应的索引项。

例如，教师关系的ID属性上的索引：

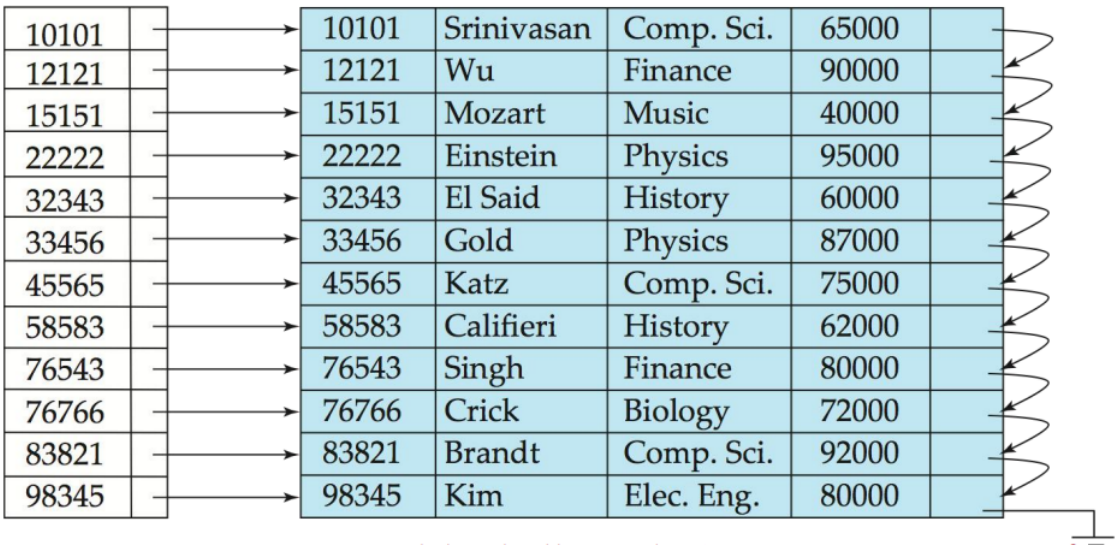

Figure 1. 例1

基于dept_name的密集索引，教师文件按depy_name排序：

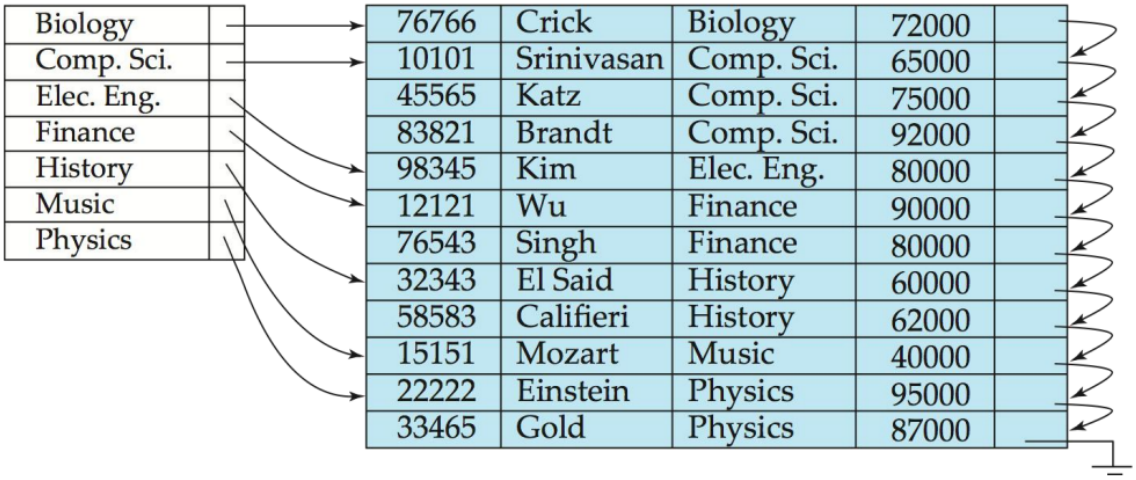

Figure 2. 例2

2.3 稀疏索引文件

稀疏索引：仅包含部分搜索键值的索引项（通常，一个数据块对应一个索引项，一个块包含多个有序的数据记录）。仅适用于数据文件记录按搜索键值顺序排列的情况。

要查找搜索键值为K的记录：

- 步骤1：找到搜索键值最大为<K的索引记录
- 步骤2：从索引项所指向的记录开始顺序搜索文件

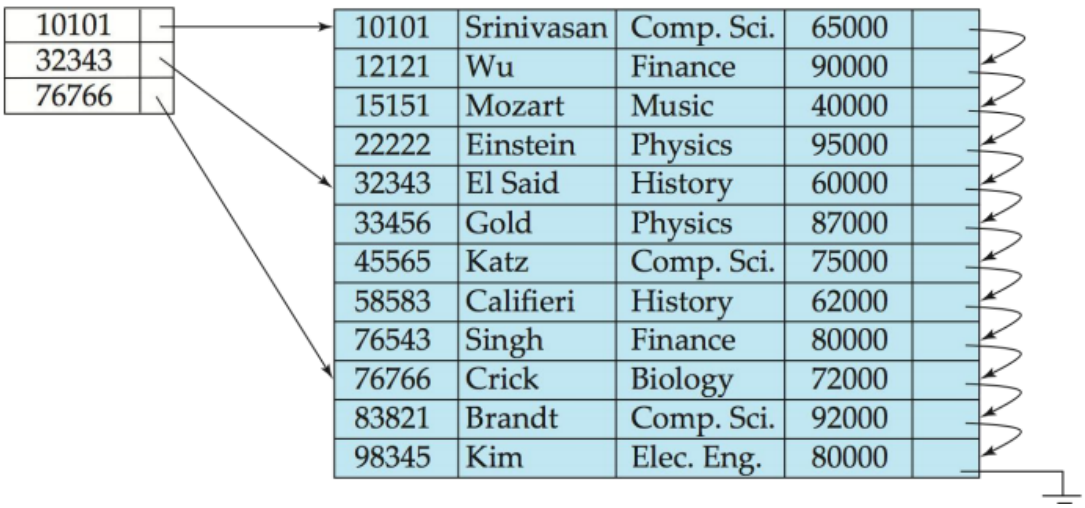

Figure 3. 例

与密集索引相比：插入和删除操作占用的空间更少，维护开销更低；通常在定位记录方面比密集索引慢。

不错的权衡：为文件中的每个块设置一个索引项的稀疏索引，对应于块中的最小搜索键值（一个块中通常包含多个数据记录，每块中最小的搜索键值放到索引项中）。

稀疏索引只能用于顺序文件，而密集索引可用于顺序和非顺序文件，如构成索引无序文件。

2.4 二级索引

通常，人们希望找到某个字段值满足特定条件的所有记录，而该字段并非主索引的搜索键。（实际应用中常有多种属性作为查询条件）

示例：在按账号顺序存储的账户数据库中，我们可能希望找到具有指定余额或余额范围的所有账户。

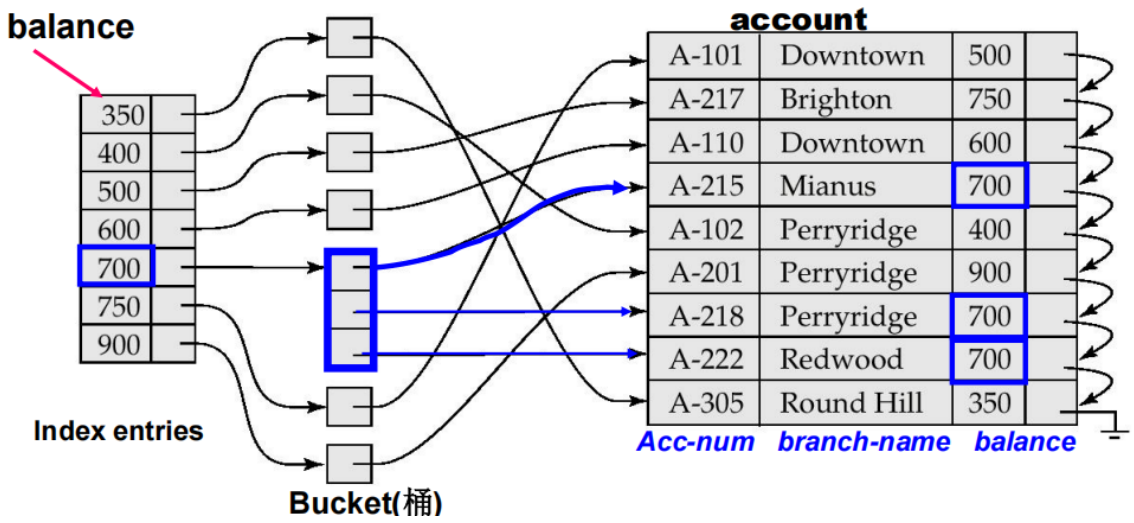

Figure 4. 例

我们可以为每个搜索键值创建一个二级索引记录；索引记录指向一个桶，该桶包含指向所有具有该特定搜索键值的实际记录的指针。

（辅助索引不能使用稀疏索引，每条记录都必须有指针指向。但search key常存在重复项—如700，而index entry不能有重复，否则查找算法复杂化，为此，使用bucket结构）

2.5 多级索引

如果主索引太大而无法装入内容，访问成本会很高。

例如：1,000,000条记录/每块10条记录=100,000块=100,000个索引项（稀疏索引）/每块100项=1000块（稀疏索引文件的大小）二分查找：$\lceil \log_2(1000) \rceil = 9$次块读取，$10 * 15ms = 150ms$

为减少对索引记录的磁盘访问次数，将保存在磁盘上的主索引视为顺序文件，并在其上构建稀疏索引。（把内层索引文件看作顺序数据文件一样，在其上建立外层的稀疏索引）

- 外层索引—主索引的稀疏索引
- 内部索引—住索引文件

如果即使外部索引也不太大而无法装入主内存，则可以创建另一级别的索引，以此类推（可以推广到任意多层索引）

对文件进行插入或删除操作时，必须更新所有级别的索引。

2.6 索引更新：删除

步骤1：系统在数据文件中找到该记录，然后删除

步骤2：更新索引文件：（针对单层索引删除）

情况1：密集索引。

如果被删除的记录是具有其特定搜索键值的唯一记录，则从索引文件中删除相应的索引条目（单记录，几搜索键具有唯一性），否则，（多条记录，即搜索键不具有唯一性）如果有多个指针指向具有相同搜索键值的所有记录，则从索引条目中删除指向已删除记录的指针（对应辅助索引的情况），否则，（对应主索引）如果被删除的记录是第一个被指向的记录，则将指针更改为指向下一条记录，否则，无需对索引条目进行任何操作。

情况2：稀疏索引。

如果已删除记录的搜索键值未出现在索引中，则无需对索引进行任何操作。否则，如果索引文件中存在该搜索键的索引条目，则通过用数据文件中（按搜索键顺序）的下一个搜索键值替换该条目来删除它。如果下一个搜索键值已经有索引条目，则直接删除该索引条目，而不是进行替换。

对于多级索引：由底层逐级向上扩展，每一层的处理过程与上述单层索引情况下类似。

2.7 索引更新：插入操作

使用待插入记录中出现的搜索键值进行查找。（利用索引找到插入位置，在数据文件中插入记录，然后分别根据情况来修改索引）

单层索引插入：

密集索引：如果搜索键值未出现在索引中，则插入一个包含该搜索键值的索引项。否则，如果有多个指针，则添加一个指向新记录的指针（在索引项中），否则，不处理该索引项。

稀疏索引：（假设每个块有一个索引项）如果创建了一个新块，则将新块中的第一个搜索键值插入到索引中，如果新纪录在其所在块中具有最小的搜索键值，则更新索引项；否则，索引不做更改。

多级插入（以及删除）算法是单级算法的简单扩展。

2.8 主索引和辅助索引

在搜索记录时，索引能带来显著的好处。

但是，更新索引会给数据库修改带来额外开销—当文件被修改时，文件上每个索引都必须更新。

使用主索引进行顺序扫描效率较高，但使用辅助索引进行顺序扫描成本较高：

- 每次记录访问可能会从磁盘读取一个新的数据块
- 数据块读取大约需要5-10ms，而内存访问大约需要100ns

3 B+树索引文件

3.1 B+树索引文件

B+树索引是索引顺序文件的一种替代方案。

- 索引顺序文件的缺点：随着文件增长性能下降，因为会创建许多溢出块；需要定期对整个文件进行重组。
- B+树索引文件的优点：面对插入和删除操作时，能通过小规模的局部更改自动进行自我重组；无需对整个文件进行重组即可保持性能。
- （次要）B+树的缺点：额外的插入和删除开销；空间开销。
- B+树的优点大于缺点：被广泛应用

B+树是一种满足以下性质的有根树：

- 从根节点到叶节点是所有路径相同—平衡树
- 每个非根节点和非叶节点有介于$\lceil n/2 \rceil$到n个孩子节点
- 叶节点有介于$\lceil (n-1)/2 \rceil$到$n-1$个值
- 特殊情况：
  - 如果根节点不是叶节点，它至少有2个孩子节点
  - 如果根节点是叶子节点（即树中没有其它节点），那么它可以有0到$n-1$个值。

3.2 B+树节点结构

3种典型节点：

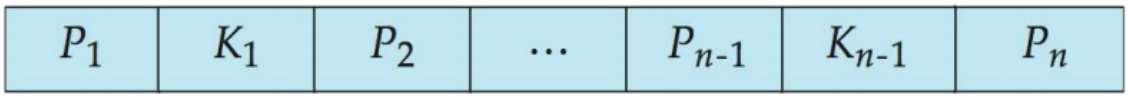

Figure 5. B+树节点

- K_i是搜索键值
- P_i是指向子节点的指针（对于非叶节点）或指向记录或记录桶的指针（对于叶节点）
- 通常，一个节点对应一个block

节点中的搜索键是有序的：

$$K_1 < K_2 < ... < K_{n-1}$$

(最初假设没有重复键，稍后处理重复问题)

对于$i = 1, 2, ..., n-1$，指针P_i要么指向具有搜索键值K_i的文件记录，要么指向指向文件记录的指针桶，每个记录都具有搜索键值K_j。仅当搜索键不构成主键时才需要桶结构（类似于密集索引，每个搜索键都出现在叶节点中）

如果L_i, L_j是叶节点且$i < j$，L_i中的所有搜索键都小于L_j的搜索键值。（叶节点间的搜索键不重叠，且所有左节点中的搜索键值一定效于右节点中的搜索键值）

必须有介于$\lceil (n-1)/2 \rceil$和$n-1$之间的搜索键。

P_n按搜索键顺序指向下一个叶节点，这便于对文件进行顺序处理。

3.3 B+树中的非叶子节点

在$\lceil n/2 \rceil$和n之间有指针（子树）。扇出数=节点中的指针数量

非叶子几点在叶子节点上形成多级稀疏索引。对于具有m个指针的非叶子节点：

- P_1所指子树中的所有搜索键都小于K_1.(P_1所指的子树中的所有search keys都小于K_1)
- 对于$2 \leq i \leq n-1$，P_i所指的子树中的所有搜索键的值都大于或等于K_{i-1}且小于K_i
- P_n所指的子树中的所有搜索键的值都大于或等于K_{n-1}

3.4 B+树查询

查找搜索键值为V的记录。

1.C =根节点

2.当C不是叶节点是：

1. 令i为满足$V \leq K_i$的最小值
2. 如果不存在这样的值，将C设置为C中最后一个非空指针
3. 否则如果$(V = K_i)$ 令$C = P_{i+1}$，否则令$C = P_i$

3.令i为满足$K_i = V$的最小值

4.如果存在这样的值i，则通过指针P_i找到所需的记录。

5.否则，不存在搜索键值为k的记录。

处理重复项

- 带有重复的搜索键：
  - 在叶节点和内部节点中，我们无法保证$K_1 < K_2 < ... < K_{n-1}$，但可以保证$K_1 \leq K_2 \leq ... \leq K_{n-1}$
  - P_i所指向的子树中的搜索键，是$\leq K_i$，但不一定是$< K_i$。为了明白原因，假设相同的搜索键值V存在两个叶节点L_i和L_{i+1}中。那么在父节点K_i中必须等于V
- 我们按如下方式修改查找过程:
  - 遍历P_i，即使$V = K_i$
  - 一旦我们到达叶节点C，在检查C是否包含V之前，设置$C = C$的右兄弟节点
- printAll过程
  - 使用修改后的查找过程来查找V的首此出现位置
  - 遍历连续的叶子节点以查找V的所有出现位置

如果文件中有K个搜索键值，那么树的高度不超过$\lceil \log_{\lceil n/2 \rceil}(K) \rceil$

一个节点的大小通常与一个磁盘块相同，一般为4千字节

3.5 B+树更新：插入操作

在B+树中，每个内部节点（非叶子节点）最多有n个关键字(keys)和$n+1$个指针；每个叶节点最多也可以容纳n条(键，指针)对（具体数目取决于B 树所定义的阶数）。当我们要向B 树中插入一个新的记录（对应一个新的搜索键 key）时，主要有以下步骤：

1. 找到要插入的叶节点。从根节点开始，根据要插入的搜索键和内部节点中存储的分隔键（separator keys），不断向下选择合适的子树，直到定位到叶节点（Leaf Node）。示例：如果当前树只有一个根节点而该节点也是叶节点，那么直接选择它；若树已经有多层内部节点，则依次比较分隔键来选择正确的分支，直到到达某个叶节点。
2. 判断该叶节点中是否已经存在相同的搜索键。
   - 如果叶节点中已经存在该搜索键（假设B 树允许重复键，则可能需要再建立bucket指针等，不过我们这里简化为不允许重复——若允许重复，可将新记录挂到bucket中而不拆分节点）。
   - 如果叶节点中不存在该搜索键，则下一步要尝试将其插入该叶节点。
3. 在叶节点中插入或分裂:
   - 若叶节点有空闲空间，即可简单地将 (key, pointer) 插入，并对叶节点中的所有键保持升序排列，插入结束。
   - 若叶节点已满（满载条数为假设的n条，此时要插入第n+1条），则必须“分裂”该叶节点：

(a) 将原来的n个(键，指针)对（假设排序后为$k_1, k_2, ..., k_n$）与新插入的键（记为k_{new}）合并，一共变成$n+1$个(键，指针)对（具体数目取决于B 树所定义的阶数）。
(b) 对这$n+1$个键进行排序，取前$\lceil (n+1)/2 \rceil$个放回到原来的叶节点，剩余$\lfloor (n+1)/2 \rfloor$个放入到一个新建的叶节点中
(c) 记新叶节点中最小的键为k_{split}，则在“分裂之后”的父节点（其指向原叶节点的指针所在的内部节点）中插入一个新的分隔键$(k_{split}, pointer_to_new_leaf)$
(d) 如果父节点也已满，则递归地对该内部节点做“分裂-向上传递”操作，直到找到一个未满的内部节点或者最终分裂到根节点。若分裂到根节点，则根节点也要分裂并创建新的根，树的高度+1.

4. 在内部节点插入或分裂（如果需要）：当从下面向上回溯到内部节点时，如果要插入的新分隔键刚好使该内部节点关键字数目达到n+1（即满载），则该内部节点也要进行同样的“分裂”：

- 将内部节点原有的n 个分隔键加上新插入的分隔键，共n+1 个，放到一个“临时内存区域”M中，其中需要存储n+1 个关键字和n+2 个指针（因为内部节点有#keys+1 条指针）。
- 对M 中的所有 (pointer, key) 成员根据 key 排序后，将前$\lceil (n+1)/2 \rceil - 1$个关键字及对应的$\lceil (n+1)/2 \rceil$个指针留在原节点；将剩下的$\lfloor (n+1)/2 \rfloor - 1$个关键字及对应的$\lfloor (n+1)/2 \rfloor$个指针放到新分配的内部节点N
- 再把 M 中的中间那个关键字 K_{mid}（第$\lceil (n+1)/2 \rceil$个关键字）“提升”插入到父节点，指向新内部节点 N′ 。
- 如果父节点也已满，就继续向上分裂。

3.6 B+树更新：删除操作

1.查找要删除的记录：

- 首先，从根节点开始，根据要删除的搜索键（Key）一路向下沿着子树找到目标叶节点（Leaf Node）。
- 如果B 树允许重复键且在叶节点中有bucket指针，则先从bucket中删除对应的记录（Record）；如果bucket本身空了，则需要从叶节点移除对应 (Key, Pointer) 对。

2.从叶节点中移除(Key, Pointer):

- 在叶节点找到该键后直接删去对应的 (Key, Pointer)。
- 删除后要检查该叶节点是否满足“下界”要求。通常，一个阶数为 n 的B 树，对叶节点的最小占用数要求是：最小键数=$\lceil n/2 \rceil$（即向上取整）。

3.处理叶节点过少的情况：

- 若删除后该叶节点中剩余的键≥ $\lceil n/2 \rceil$，则直接结束，不需要上溯。
- 若删除后该叶节点中剩余的键< $\lceil n/2 \rceil$，则需要与相邻兄弟节点（Sibling）进行如下两种操作之一：
  1. 兄弟节点合并：如果该叶节点和某一相邻兄弟（通常选择左兄弟或者右兄弟）合并后，总条目数≤ n，则可以将两个节点中的所有键都放入到一个节点中，另一个节点被删除。从父节点中删除与被删除兄弟节点对应的分隔键(K_{i-1}, P_i)，如果因此导致父节点条目过少，则继续向上处理。
  2. 兄弟节点重分配：如果相邻兄弟节点有多余的键，可以从兄弟节点借一个键过来，使得当前叶节点和兄弟节点都能满足最小键数要求。然后需要在父节点中更新对应的分隔键，使其反映兄弟节点与本节点之间新的“最小键”分隔值。
- 叶节点的合并或重分配可能会导致父节点关键字数量发生变化，从而继续导致父节点下溢，需要“递归”向上执行合并或重分配。

4.内部节点(Non-Leaf Node)的合并/重分配

- 如果在子节点合并后，需要从父节点删除分隔键，若父节点剩余关键字数量仍≥ $\lceil n/2 \rceil - 1$，则结束
- 否则，父节点也发生下溢，需要与兄弟内部节点合并或重分配；
- 继续检查父节点是否“下溢”，若下溢则向上递归，直到碰到根节点或者节点满足最小键数。

5.根节点的特殊处理：

- 如果根节点删除分隔键后只剩下 1 条指针（即只剩 1 棵子树），则删掉这个根节点，让它的唯一子指针所指的节点成为新的根，树的高度减一。
- 如果根节点删分隔键后仍有 ≥2 条指针，则根节点仍然有效，不需进一步调整。

3.7 非唯一搜索键

之前描述方案的替代方案

- 单独块上的桶（糟糕的主意）
- 每个键对应的元组指针列表
  - 处理长列表的额外代码
  - 如果搜索键上有许多重复项，删除一个元组的代价可能很高
  - 空间开销低，查询无额外成本
- 通过添加记录标识符使搜索键唯一
  - 键的额外存储开销

– 插入/删除代码更简单
– 广泛使用

由于记录比指针占 更多空间，因此良好的空间利 率很重要。

为提高空间利 率，在分裂和合并期间让更多兄弟节点参与重新分配

4 写优化索引

4.1 日志结构合并(LSM)树

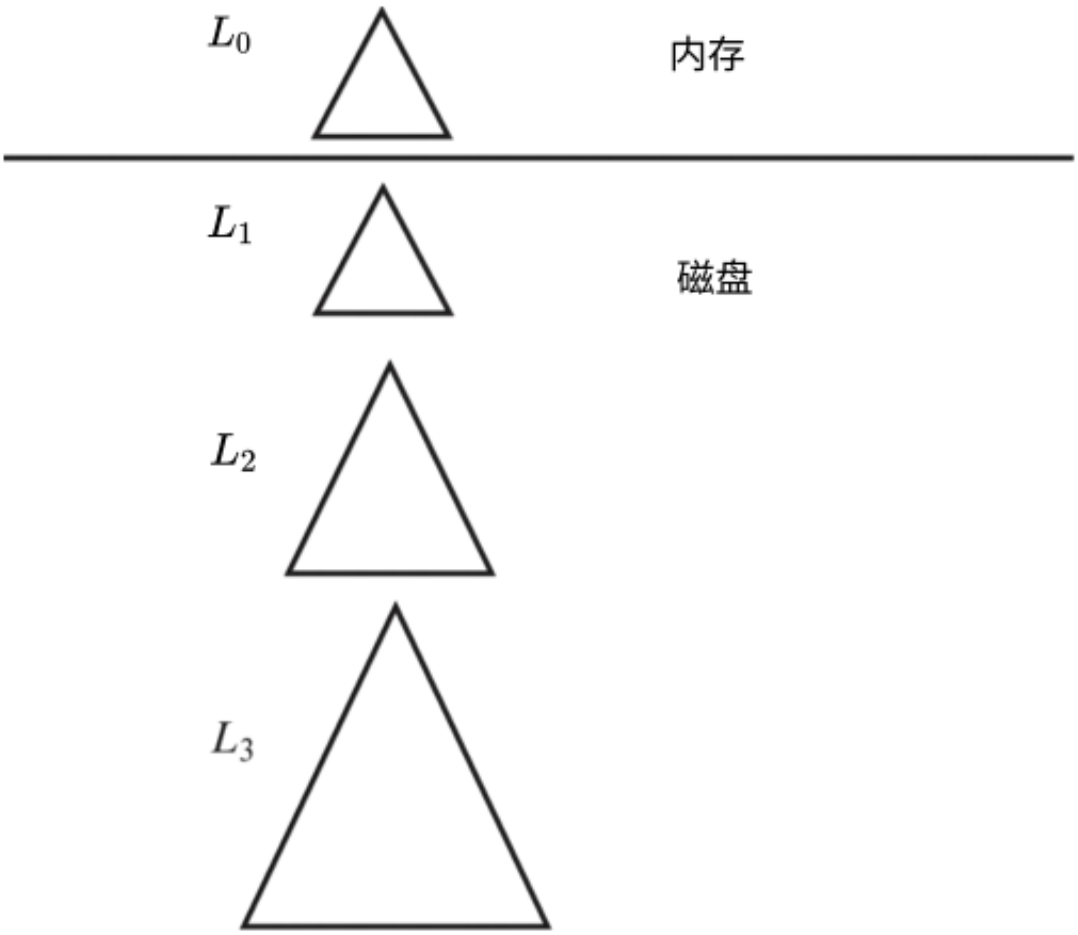

Figure 6. LSM树

4.1.1 什么是LSM树

LSM树是一种专门为高吞吐量写入优化的索引结构，常用于需要频繁插入（insert）和查询（query）的场景，尤其对磁盘或闪存（SSD/Flash）后端极为友好。其核心思想是：

1. 写请求先写到内存中的小结构（MemTable），避免磁盘小量随机写。
2. 当内存结构满后，以批量顺序写的方式「刷新」（flush）到磁盘上的一个较小层次。
3. 随后当磁盘上该层数据积累到一定阈值，再与下一级更大的层次进行“合并”（merge），形成更大规模的有序文件。
4. 通过多级、批量、顺序 I/O（而非随机写），显著减少磁盘写入的开销。

L_0：维护一个“可排序的内存结构”（通常是跳表、红黑树或MemTable），写速度非常快。

$L_1, L_2, ...$（磁盘）：每一层都是一个或多个“有序文件” 或“B+树结构”，它们依次满足“容量逐层增大”的策略。

当L_0满时，会以批量方式将数据合并到L_1；当L_1达到阈值时，再和L_2做合并，以此类推。每次合并都是“顺序读旧数据 + 顺序写新数据”，大幅度减少对磁盘的随机写操作。

4.1.2 LSM树的层次结构与阈值策略

假设我们将L_0看作内存树，$L_1, L_2, L_3, ...$都是磁盘上的有序结构（例如某种底层可拆分的SSTable或多路归并后生成的B+树文件），并且通常约定：

- L_0的容量比较小，例如C_0个键后就出发flush
- L_1的容量上限为$C_1 = k \times C_0$个键
- L_2的容量上限为$C_2 = k \times C_1 = k^2 \times C_0$
- 以此类推：$C_{i+1} = k \times C_i$，其中k是一个常数

因此，当L_0内存结构积累到C_0键时，就会“批量写”到磁盘L_1；当L_1上积累到C_1键时，就会和L_2做一次更大批量的merge，生成新的L_2文件；以此类推。

这种逐层“分级合并（tiered merging）” 或“大小归并（size-tiered compaction）” 的方式正是LSM树的核心。

4.1.3 INSERT流程

下面用一段文字示例来说明 LSM 树插入的具体过程，以便对比其与传统 B+树的不同之处。

假设我们当前的各层状态如下：

内存层L_0还没有满，已存在若干键：$L_0 : [< a, 1 >, < b, 2 >, < d, 4 >]$（以跳表或红黑树形式存储，因此是有序的）

磁盘层L_1现有一个有序文件（SSTable或B+树文件）包含键：$L_1 : [< e, 5 >, < h, 8 >]$

磁盘层L_2为空或容量尚未到达阈值。

1.写入"<c,3>"：将其插入到内存层L_0。$L_0 : [< a, 1 >, < b, 2 >, < c, 3 >, < d, 4 >]$。此时$L_0$键数=4.假设$C_0 = 4$（达到内存阈值），需要flush L_0到L_1.

2.将L_0 flush到L_1:

步骤2.1：先把L_0中所有键“顺序写”到磁盘，形成一个临时有序文件（SSTable），比如我们得到一个文件F_0包含：$F_0 : [< a, 1 >, < b, 2 >, < c, 3 >, < d, 4 >]$

步骤2.2：将现有的L_1文件$[< e, 5 >, < h, 8 >]$与新生成的F_0做多路归并，得到新的L_1（容量$C_1 = 10$例如）。多路归并后：$L_1' = [< a, 1 >, < b, 2 >, < c, 3 >, < d, 4 >, < e, 5 >, < h, 8 >]$

步骤2.3：清空内存层L_0（再建一个空的跳表，让应用继续写）。

3.持续插入并触发$L_1 \rightarrow L_2$合并：后续若插入$< f, 6 >, < g, 7 >, < i, 9 >$ ...均进入L_0，直到L_0满再flush，当L_1中键数累计到达阈值C_1（假设为6）时，就要触发L_1与L_2合并：

先将L_1当前文件（如[a,b,c,d,e,h]）与最新的L_1 flush结果做归并，生成更大的一个临时文件。

归并后得到新的L_2文件（容量$C_2 = k \times C_1$，例如60）

此后，清空L_1（或保留少量增量，以“分层归并”而不是“一次性全量归并”），保持L_2总是有序的。

4.1.4 QUERY操作流程

当我们要查询某个键(Key="X")时，LSM树必须分层搜索：

1.先到L_0（内存层）中查找。如果在这儿找到了“<X, value>” 且没有 tombstone（删除标记），直接返回结果。

2.如果L_0中没有，再去l_1（最上层磁盘）查找；

3.若L_1中也没有，再去$L_2, L_3, ...$依次查找。

4.如果在某一层发现了“删除标记（tombstone）”条目，则说明该 Key 已被删除，即使在更低层存在旧值，也当作“已删除”而返回空。

由于查询要向下搜索多层，会产生一定的查询延迟。但为了缩短查询路径，可以在每个层的文件中维护一个布隆过滤器（Bloom Filter），先检查布隆过滤器，看该Key是否“可能存在”再决定是否真正读磁盘。这样可以避免绝大多数无意义的盘上查找。

4.1.5 DELETE与UPDATE操作

LSM 树中的删除与更新并不直接在磁盘层做随机删除，而是通过“打墓碑（tombstone）”的方式：

1.DELETE:

- 当应用请求删除某个键<X>时，首先向L_0（内存）中插入一条特殊的“删除条目”(tombstone)，格式可以是<X, DELETE_MARK>。
- 后续的查询如果先在L_0发现了<X, DELETE>，就判定为已删除，不往下层查旧值；

- 当L_0 flush到L_1时，这个tombstone也会连同正常条目一并合并。若在更低层(L_1, L_2)存在旧的<X,old_value>，则在归并过程中一旦发现delete entry，就将对应的旧值直接过滤掉，保留delete tombstone；
- 如果tombstone也在更下层归并时被清理（因为它本身同样过时或者到了更低层合并时过期），就从索引中彻底删除该键。

2.UPDATE:

更新其实等同于一次“删除旧值 + 插入新值”两个步骤：

1.向L_0插入<X,DELETE>标记（将旧值逻辑删除）；

2.再向L_0插入<X,new_value>（像正常插入一样排序）；

在后续的归并过程中，归并算法会发现“先有 delete tombstone，再有新值”，则保留最新 <X, new_value> 同时丢弃 tombstone；如果是相反次序，也会丢弃 tombstone并保留最新值。

4.1.6 LSM 树的优缺点

优点

1. 写入仅做顺序 I/O:不像B+树那样会频繁产生随机写，LSM树只会“先写内存，再批量顺序写磁盘”，有效减少磁盘写开销。
2. 磁盘层文件始终满载:每次都进行合并，使得最终持久化到磁盘的文件几乎都是满页，避免空洞和空间浪费。
3. 插入/删除吞吐量高:由于大多数插入都集中在内存，再通过合并在后台清理陈旧或删除条目，插入性能相比传统B+树更优。

缺点：

1. 查询需要同时在多层查找:最坏情况下要查到最底层，如果每层都在磁盘上，就会导致多次随机读，查询延迟增加。可以通过布隆过滤器（Bloom Filter）降低多层查找次数（如果 Bloom Filter 判定不存在，就不去对应层做实际查找）。
2. 数据被重复复制:每一次合并，都会把整层数据读出然后写入到下一层，导致重复 I/O。虽然是顺序I/O，但如果合并频繁，仍会带来开销。

4.1.7 Stepped-Merge Index（分级归并变种）

为了进一步降低磁盘写成本，有时会在每一层不是只保留一个有序文件，而是保留多个 (SSTable) 文件，形成“分步合并（stepped-merge）” 的策略：

- 每一层L_i允许存在最多T_i个小文件（SSTable）。

- 当超出阈值时，不是一次性把L_i所有文件合并到L_{i+1}，而是先将L_i中几个文件有选择地合并成一个新文件，放回L_i本层；
- 这样可以减少每层向下合并的 I/O 规模，但查询时就要在更多文件中定位某个 Key，需要额外索引或 Bloom Filter 来过滤文件。

许多大数据存储系统（如 Google Bigtable、Apache Cassandra、MongoDB、LevelDB、MyRocks 等）都使用了这种多文件分级合并变种。

4.2 缓冲树

LSM树的替代方案。

核心思想：B+树的每个内部节点都有一个用于存储插入操作的缓冲区：当缓冲区满时，插入操作会被移到更低层次；若缓冲区较大，每次会有许多记录被移到更低层次；相应的，每条记录的I/O操作减少。

优势：查询开销更小，可与任何树索引结构一起使用。

缺点：比LSM树有更多随机I/O。

5 SQL中的索引定义

创建一个索引：

```
create index <index-name> on <table-name>(<attribute-list>);
```

例如：

```
create index b-index on branch(branch-name);
create index cust-strt-city-index on customer(customer-city,
   customer-street);
```

使用`create unique index`间接指定并强制要求搜索键为候选键

例如：`create unique index uni-acnt-index on account(account-number);`。如果支持SQL唯一完整性约束，则并非真正必须。

删除索引：`drop index <index-name>`