

第 14 章 索引

郑新、徐鹏飞、李健

2025 年 12 月 30 日

本节内容概览

- 1 引入、基本概念
- 2 线性索引、静态索引
- 3 倒排索引
- 4 动态索引
- 5 位索引
- 6 总结

数据库与索引

- CRUD/增删改查：增 (Create)、删 (Delete)、改 (Update)、查 (Read)
- 在远超内存容量的海量数据中，实现可靠的持久化存储与极速检索。
- 想象一个拥有千万级订单的电商系统，数据必须存储在磁盘上，
- 而索引，负责建立“逻辑标识”与“物理地址”的映射关系。

思考问题：在这种大规模场景下，如果我们不做索引而直接从头开始遍历磁盘文件（全表扫描），你认为对用户来说最直观的性能灾难是什么？

从内存到外存

- 排序 → 外排序
- 查找 → 外查找（建索引是关键）
- 共同点：从内存到外存、最小化外存的 I/O 次数、

从内存到外存

- 排序 → 外排序
- 查找 → 外查找（建索引是关键）
- 共同点：从内存到外存、最小化外存的 I/O 次数、二叉 → 多叉

将“二叉”改为“多叉”，每个节点容纳更多关键词对减少 I/O 有什么具体帮助？

从内存到外存

- 排序 → 外排序
- 查找 → 外查找（建索引是关键）
- 共同点：从内存到外存、最小化外存的 I/O 次数、二叉 → 多叉

将“二叉”改为“多叉”，每个节点容纳更多关键词对减少 I/O 有什么具体帮助？

就像一栋大楼，

- ① 如果每层只有 2 户人家，找第 100 户需要爬很多层；
- ② 但如果每层有 100 户，你可能只需要爬 1 层就能找到目标。

但是，

从内存到外存

- 排序 → 外排序
- 查找 → 外查找（建索引是关键）
- 共同点：从内存到外存、最小化外存的 I/O 次数、二叉 → 多叉

将“二叉”改为“多叉”，每个节点容纳更多关键词对减少 I/O 有什么具体帮助？

就像一栋大楼，

- ① 如果每层只有 2 户人家，找第 100 户需要爬很多层；
- ② 但如果每层有 100 户，你可能只需要爬 1 层就能找到目标。

但是，如果一个节点里塞了太多的关键词，查找特定关键词的开销会变大。

索引的定义和基本概念

主码（primary key）用于标识

主码是数据库中用于唯一标识每条记录的关键字段，确保每条记录的唯一性。

辅码（secondary key）用于检索

辅码是数据库中允许存在重复值的字段，它通过辅码索引与记录的主码值相关联，使得即使存在重复值，也能通过辅码实现快速灵活的检索。

以学生信息表为例

- 应该用什么作为主码，什么作为辅码？
- 能否设置多个辅码来支持更多样、灵活的查询？

数据库索引：主码与辅码的应用

- **主码索引 (Primary Index)**

- **字段选择**：学号 (Student ID)
- **特性**：唯一且非空，直接定位物理记录
- **效率**：查询特定学生速度最快

- **辅码索引 (Secondary Index)**

- **字段选择**：专业、姓名、入校年份
- **逻辑**：支持灵活的分类检索（如“查找所有计算机专业学生”）
- **维护**：存储主码值而非物理地址，平衡了空间与更新成本

设计原则

为了减少磁盘 I/O，索引通常采用“矮胖”的多路平衡树结构。

索引查询流程：回表机制

辅码索引 (例如：专业 = CS)

第一步：从辅码映射到主码

查找主码

主码索引 (获取学号)

第二步：从主码映射到位置

定位物理地址

物理记录
(磁盘上的完整数据)

核心概念辨析：索引、索引文件与索引技术

索引 (Index)

一种**逻辑结构**，建立关键码 (Key) 到记录存储地址 (Address) 的映射关系。其本质是实现“快速定位”的导航表。

索引文件 (Index File)

索引项的集合在外部存储器（如磁盘）上的物理表现形式。它通常独立于原始数据文件存在，通过频繁的 I/O 操作进行访问。

索引技术 (Indexing)

指设计和实现索引的**一系列算法与组织方式**。它决定了索引的结构（如线性、树型、散列）以及在增删改查时的维护策略。

三者关系

使用**索引技术**创建**索引**，并将其以**索引文件**的形式保存在磁盘中。

本节内容概览

- 1 引入、基本概念
- 2 线性索引、静态索引**
- 3 倒排索引
- 4 动态索引
- 5 位索引
- 6 总结

线性索引：稠密索引 (Dense Index)

- **定义：**为原始数据表中的每一个记录都建立一个索引项。
- **索引项结构：**〈**关键码**, **指针**〉
 - **关键码：**如“学号”，用于排序和查找。
 - **指针：**指向该记录在磁盘上的物理起始地址。

| 索引项 (学号) | 物理地址 (指针) |
|----------|-----------|
| 2023001 | 0x001A |
| 2023002 | 0x0C2B |
| 2023003 | 0x00FF |

表：稠密索引示例：索引有序，数据可无序

- ✓ 查找效率高（可使用二分查找， $O(\log n)$ ）
- ✗ 但空间开销大，且数据更新时索引维护成本高。

线性索引：稀疏索引 (Sparse Index)

- **核心思想**：只为每一个数据块 (Block) 建立一个索引项。
- **前提**：原始数据文件必须按关键码有序排列 (块内可以无序)。

| 索引关键码 | 块指针 | 包含的学号范围 |
|---------|---------|-----------------|
| 2023001 | Block 1 | 2023001 2023100 |
| 2023101 | Block 2 | 2023101 2023200 |
| 2023201 | Block 3 | 2023201 2023300 |

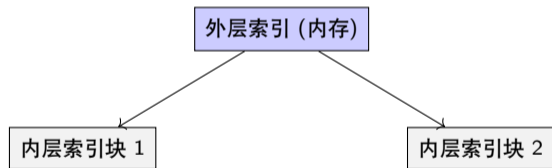
优缺点分析

优点：极大节省索引空间，减少磁盘 I/O。

缺点：定位到块后，仍需在块内进行顺序查找。

二级线性索引 (Two-level Linear Index)

- **背景：**当一级稠密索引太大，无法完全装入内存时使用。
- **逻辑结构：**
 - **外层索引：**记录内层索引块的起始范围（常驻内存）。
 - **内层索引：**具体的关键码与数据地址映射（按需从磁盘调入）。



查找效率

通过两次 I/O 即可定位：一次读入内层索引块，一次读入实际数据记录。

多级线性索引 (Multi-level Linear Index)

- 外层索引 (Index Directory): 存储在内存中, 记录每个索引块的范围。
- 内层索引块 (Index Blocks): 存储在磁盘上, 包含具体的 $\langle \text{Key}, \text{Pointer} \rangle$ 。
-

在这种多级结构下, 如果我们删除了一半的学生, 你是倾向于:

- ① 重新调整整个多级索引的线性布局?
- ② 还是允许索引块里留出一些空位, 以减少挪动数据的代价?

多级线性索引 (Multi-level Linear Index)

- 外层索引 (Index Directory): 存储在内存中, 记录每个索引块的范围。
- 内层索引块 (Index Blocks): 存储在磁盘上, 包含具体的 $\langle \text{Key}, \text{Pointer} \rangle$ 。
- 留白策略 (Padding): 每个块内部预留空间, 处理插入操作。

在这种多级结构下, 如果我们删除了一半的学生, 你是倾向于:

- ① 重新调整整个多级索引的线性布局?
- ② 还是允许索引块里留出一些空位, 以减少挪动数据的代价? ✓

性能权衡

虽然留出空位会浪费一点磁盘空间, 但它避免了频繁的全局数据迁移, 显著降低了写操作的延迟。

从静态到动态：索引块的分裂与链接

- **触发条件：**当新数据插入导致某个内层索引块（Index Block）溢出时。
- **核心操作：**
 - ① **分裂 (Split)：**将一个满块拆分为两个半满的块。
 - ② **向上生长：**在上层索引中插入一个新的条目，指向新分裂出的块。
- **优势：**
 - 避免了全文件的“线性移动代价”。
 - 保证了每个块都有一定的空位（填充因子），为后续插入留出余地。

从静态到动态：索引块的分裂与链接

- 触发条件：当新数据插入导致某个内层索引块（Index Block）溢出时。
- 核心操作：
 - ① 分裂 (Split)：将一个满块拆分为两个半满的块。
 - ② 向上生长：在上层索引中插入一个新的条目，指向新分裂出的块。
- 优势：
 - 避免了全文件的“线性移动代价”。
 - 保证了每个块都有一定的空位（填充因子），为后续插入留出余地。

这种“分而治之”的递归结构，最终演变成了经典的 $B/B+$ 树 结构。

课堂练习 1：稀疏索引 (Sparse Index) 的脆弱性

● 情境描述：

- 原始学生档案按“学号”有序存放。
- 数据库引擎执行了磁盘碎片整理，将物理块 A 的数据整体移动到了新位置 B。

互动提问

在这种情况下，即使学生的“学号”和“逻辑顺序”都没有发生任何变化，我们是否必须修改稀疏索引表？

- A. 不需要，因为学号顺序没变。
- B. 必须修改，因为索引项包含物理指针。
- C. 只需修改主码索引，不需要修改线性稀疏索引。

练习 1 解析：物理地址的“硬绑定”

- 正确答案：B
- 深度解析：
 - 稀疏索引的本质：它是 $\langle Key, Pointer \rangle$ 的集合。*Pointer* 是指向磁盘物理块的“门牌号”。
 - 物理关联：即使逻辑内容（学号）不变，一旦物理位置变动，原有的“门牌号”就失效了。
 - 教学重点：这是线性索引的一大痛点——维护成本随物理变动而增加。

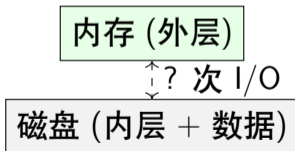
课堂练习 2：二级线性索引的 I/O 代价

- 背景设定：

- 外层索引：已调入 内存。
- 内层索引块与原始数据：均在 磁盘。

计算挑战

要从 10 万条记录中通过“学号”精确查找一名学生，最少需要进行几次磁盘 I/O 操作？



练习 2 解析：追踪磁盘访问路径

- 正确答案：2 次 I/O
- 步骤拆解：
 - ① 第 1 次：根据内存地址，从磁盘读取包含目标学号的那个“内层索引块”。
 - ② 第 2 次：从内层索引块获得物理指针，从磁盘读取“学生原始档案”。

课堂练习 3：二级线性索引的结构与规模

- 结构设定：

- 外层索引：包含 100 个条目。
- 内层索引块：每个块包含 500 个具体的学号索引项。

计算挑战

在这种二级线性索引结构下，该系统最多可以支持索引多少名学生的档案？

- Ⓐ. 600 名
- Ⓑ. 5,000 名
- Ⓒ. 50,000 名
- Ⓓ. 500,000 名

练习 3 解析：层级结构的乘法效应

- 正确答案：C (50,000 名)

- 计算公式：

总容量 = 外层条目数 × 每个内层块的条目数

$$100 \times 500 = 50,000$$

- 深度理解：

- 外层索引就像是“索引的索引”。
- 这种结构允许我们将巨大的索引表拆分成多个小块，从而实现“按需调入内存”，解决了内存装不下超大索引表的问题。

课堂练习 4：静态索引 (Static Index) 的维护

- **背景：**线性索引通常被视为一种静态索引结构。
- **操作：**学校数据库在学期中频繁发生“学生转入”和“新生成籍”等插入操作。

问题

静态索引在面对这些频繁的插入操作时，主要面临的挑战是什么？

- Ⓐ 查找算法会从二分查找退化为顺序查找。
- Ⓑ 必须定期重建索引以维持其物理或逻辑的有序性。
- Ⓒ 索引文件会因为数据增加而自动缩小。

练习 4 解析：静态与动态的博弈

- 正确答案：B

- 解析：

- **有序性是核心**：线性索引依赖有序排列来实现快速查找（如二分查找）。
- **维护困境**：在中间插入新项会导致后续所有项后移。为了效率，通常先将新数据放在溢出区，但这会导致查找性能下降。
- **最终手段**：必须通过“定期重建（Reorganize）”来重新排序索引，恢复最佳性能。

典型案例：ISAM，索引顺序访问方法

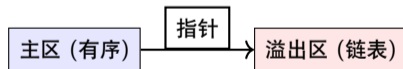
- **诞生背景：**由 IBM 在 1960 年代为大型机（Mainframe）开发，是磁盘取代磁带成为主流存储后的首个重要索引标准。
- **设计目标：**
 - **顺序处理：**像磁带一样，支持高效的批量顺序扫描（如打印全体工资单）。
 - **随机访问：**利用磁盘特性，支持通过键值快速定位单条记录。
- **硬件耦合：**其结构紧密贴合当时磁盘的物理构造（柱面 Cylinder、磁道 Track）。

ISAM 核心设计：三级静态索引结构

- **柱面索引**：最高层索引，记录每个柱面的最大关键码。通常可全部装入内存。
- **磁道索引**：位于每个柱面的起始磁道，记录该柱面内各磁道的关键码范围。
- **特性：静态结构**
 - 索引块在文件创建时一次性分配。
 - 即便数据量增减，索引层级和结构保持不变。

ISAM 核心设计：主存储区与溢出区

- **主存储区**：数据按键值顺序紧密排列。
- **溢出区**：当主区某磁道因插入而满员时，新记录被移至此处。
- **溢出链表**：
 - 主区记录通过指针指向溢出区。
 - 溢出区内部使用链表维持逻辑顺序。



量化分析：静态索引 (ISAM) 的性能崩塌

- 实验场景：一个固定 3 层索引的 ISAM 文件。
- 查询成本： $Cost = \text{索引访问} + \text{主区 I/O} + \text{溢出链表 I/O}$

| 溢出率 (%) | 平均链表长度 | 平均 I/O 次数 | 性能评价 |
|---------|--------|-----------|-------------|
| 0% | 0 | 3.0 | 极快 (理论上限) |
| 10% | 1.2 | 4.2 | 轻微下降 |
| 50% | 5.5 | 8.5 | 显著变慢 |
| 80% | 15.2 | 18.2 | 崩塌 (慢 6 倍!) |

结论

当溢出区数据过多时，原本有序的线性查找退化为多个离散块的随机 I/O。

逻辑演进：从“修补”到“重构”

静态索引 (ISAM) 的思路

- **策略**：索引不动，数据去“排队”（溢出链表）。
- **结果**：结构僵化，越用越慢。
- **维护**：必须定期整体重建。

动态索引 (B+ Tree) 的思路

- **策略**：数据变动，索引跟着“分裂”。
- **结果**：自平衡，始终保持 $O(\log n)$ 。
- **维护**：实时自我调整，无需停机。

本节内容概览

- 1 引入、基本概念
- 2 线性索引、静态索引
- 3 倒排索引
 - 针对文本数据的倒排索引
 - 基于属性的倒排索引（适合半结构化数据）
- 4 动态索引
- 5 位索引

倒排索引：从图书附录到搜索引擎

- **起源：**源于传统图书的关键词索引，是人类处理知识的最古老逻辑。
- **技术转折点：**
 - **1950s：**H.P. Luhn 提出词频统计，奠定自动索引基础。
 - **1990s：**互联网爆发，要求系统在毫秒级检索数千亿网页。
- **核心地位：**现代搜索引擎（Elastic-Search, Lucene, Google）的核心灵魂。

“如果 $B+$ 树是关系型数据库的基石，那么倒排索引就是信息检索的基石。”

为什么说倒排索引在早期是“静态”的？

- **构建代价**：构建倒排索引涉及对海量文档的“分词-排序-合并”，计算开销巨大。
- **不可变性**：为了压缩空间和极致性能，倒排列表通常是压缩且连续存储的。
- **更新策略**：
 - 无法像 B+ 树那样进行实时插入。
 - **增量处理**：新文档进入“内存缓冲区”，达到一定规模后，以“只读段 (Segment)”的形式写出，最终通过定时后台任务进行“段合并 (Merge)”。

案例分析：为什么新发的微博不能“秒搜”到？

- 操作流：用户发布微博 → “我爱学习数据库”。
- 后台逻辑：
 - ① 分词：[我, 爱, 学习, 数据库]
 - ② 索引状态：主索引（只读）中并没有这条新数据。
 - ③ 增量缓冲：系统先将新词放入内存的 Buffer 中。

主倒排索引（静态/只读）

| 关键词 | 文档 ID 列表 |
|-----|------------|
| 数据库 | 1, 45, 102 |
| 学习 | 3, 12, 45 |

增量段（临时存储）

| 关键词 | 新文档 ID |
|-----|----------|
| 数据库 | 2024 (新) |
| 学习 | 2024 (新) |

- 性能权衡：为了查询效率，系统不会每发一条微博就重排 TB 级的索引。
- 合并 (Merge)：通常在数秒或数分钟后，系统将增量段合并进主索引。

架构思考：实时性 vs. 查询性能

| 维度 | 信息检索 (如：微博/网页) | 事务处理 (如：银行转账) |
|-------|----------------|----------------|
| 实时性 | 允许 Lazy (准实时) | 必须即时 (强一致性) |
| 查询复杂度 | 极高 (多关键词、模糊匹配) | 较低 (基于主键/精确查找) |
| 更新频率 | 极高 (海量写入) | 中/高 (要求写成功即见) |
| 首选索引 | 倒排索引 + 异步合并 | B+ 树 (动态平衡) |

没有完美的索引，只有最合适的权衡

为什么倒排索引可以“等一等”，而银行索引必须“秒成”？

- 容错性：搜不到刚发的微博不会丢钱，但刚存的钱搜不到会引发恐慌。
- 物理结构：紧凑的静态结构利于全量搜索，灵活的动态结构利于单点更新。

索引设计的“不可能三角” (RUM Conjecture)

- R (Read Overhead): 读取开销, 希望查询越快越好。
- U (Update Overhead): 更新开销, 希望写入即见, 且不费力。
- M (Memory/Storage Overhead): 空间开销, 希望索引越小越好。



经典索引的权衡分析

- 倒排索引 / ISAM: 牺牲 U (更新慢), 追求极佳的 R (读) 和 M (压缩空间)。
- B+ 树: 牺牲 M (预留空间/空位), 追求 R 与 U 的动态平衡。
- LSM 树 (日志结构): 牺牲 R (读放大), 追求极致的 U (写性能)。

概念辨析：什么是半结构化数据 (Semi-structured Data) ?

- **结构化 (Structured)**：如传统 SQL 表。模式 (Schema) 固定，每行必须有相同的列，字段长度固定。
- **半结构化 (Semi-structured)**：数据带有“自我描述”性。
 - 常见格式：JSON, XML, YAML。
 - 特点 1：模式灵活性。同一集合中，记录 A 有“年龄”，记录 B 可以没有，而记录 C 可能多出一个“兴趣爱好”数组。
 - 特点 2：嵌套性。属性可以是另一个对象或列表。

JSON 示例

```
{ "UID": 101, "name": " 张三", "skills": ["Java", "SQL"], "extra": { "hobby": " 摄影" } }
```

倒排索引与数据格式的关系

核心结论

基于属性的倒排索引不一定针对半结构化数据，它只是一种“查询优化”手段。

● 场景 A：结构化数据中

- 在标准的 SQL 数据库中，我们也可以为“城市”列建立倒排索引（位图索引）。
- 目的：加速低基数字段的过滤，而不是为了处理灵活模式。

● 场景 B：半结构化数据中（天然契合）

- 面对 JSON 这种属性不确定的数据，B+ 树很难预先定义索引列。
- 倒排索引的优势：它可以为每一个出现的 Key:Value 对动态生成倒排项，完美适配“模式自由”的特性。

核心概念：什么是低基数字段？

- 定义：基数（Cardinality）是指某一列中不同取值（Unique Values）的个数。
- 低基数（Low Cardinality）：相对于总行数，该列的唯一值非常少。
- 高基数（High Cardinality）：该列的取值几乎每一行都不同。

高基数案例 (High)

- 身份证号：几乎不重复。
- 用户 ID：全球唯一。
- 精确时间戳：毫秒级唯一。

低基数案例 (Low)

- 性别：男、女、未知。
- 支付状态：待支付、已支付、退款。
- 省份：只有 34 个可能。

架构选择：基数如何影响索引？

- **高基数字段 → 选 B+ 树**

- 每次查询能过滤掉绝大多数数据。
- 路径清晰，检索效率极高。

- **低基数字段 → 选位图索引 (Bitmap/Inverted)**

- 若用 B+ 树，搜索结果会包含数百万行，导致频繁的磁盘随机 I/O，性能极差。
- 若用位图，只需对几个 Bit 数组做 CPU 位运算 (AND/OR)，速度极快。

经验准则 (Rule of Thumb)

如果一个字段的唯一值占总行数的比例小于 1%，通常被视为低基数。

基于属性的倒排索引案例：专家库

研究方向是“多值属性”。一位专家可以同时属于“人工智能”和“生物医疗”。

原始数据：专家信息表 (Forward Index)

| EID | 姓名 | 研究方向 (Attributes) |
|-----|-----|-------------------|
| E01 | 王教授 | 人工智能, 自动驾驶 |
| E02 | 李教授 | 生物医疗, 数据挖掘 |
| E03 | 张教授 | 人工智能, 知识图谱 |
| E04 | 陈教授 | 自动驾驶, 智慧交通 |

生成的属性倒排表 (Inverted Index)

- 方向: 人工智能 $\rightarrow \{E01, E03\}$
- 方向: 自动驾驶 $\rightarrow \{E01, E04\}$
- 方向: 生物医疗 $\rightarrow \{E02\}$

多维检索：寻找“人工智能 + 交通”的复合人才

- **查询需求：**寻找既懂“人工智能”又懂“智慧交通”的专家进行项目评审。
- **执行过程：**
 - ① **检索 1：**从倒排表取“人工智能”列表 $\rightarrow L_1 : \{E01, E03\}$
 - ② **检索 2：**从倒排表取“智慧交通”列表 $\rightarrow L_2 : \{E04\}$
 - ③ **检索 3：**从倒排表取“自动驾驶”列表 $\rightarrow L_3 : \{E01, E04\}$

逻辑运算 (Intersection / Union)

如果我们找的是 (人工智能 OR 自动驾驶) **AND** 智慧交通：

- $(L_1 \cup L_3) \cap L_2 = \{E01, E03, E04\} \cap \{E04\} = \{E04\}$

这种多对多的关系中，倒排索引避免了昂贵的表连接 (Join)，
将查询转化为高效的列表合并运算。

总结：我们为 B+ 树清理出的舞台

- **静态索引 (Static Index/ISAM):**

- **核心：**物理结构与索引固定，通过溢出区处理变动。
- **结论：**读得极快，但面对频繁写入会因“溢出链表”而性能崩塌。

- **倒排索引 (Inverted Index):**

- **核心：**单词到文档的映射，利用位图和跳跃表优化。
- **结论：**非结构化/低基数数据的王者，但维护成本高，实时性较弱。

- **架构公理 (RUM Conjecture):**

- **结论：**没有免费的午餐。索引设计必须在“读、写、空间”之间做残酷抉择。

转场

我们看过了 *ISAM* 因为不愿改变结构而导致的性能泥潭；
我们看过了倒排索引为了查询极致而忍受的异步延迟。

那么，有没有一种结构，它既能像 *ISAM* 一样利用磁盘的页特性，
又能像生物细胞一样，在压力过大时通过分裂来自我调节？

它不追求倒排索引那种极致的压缩，也不追求二叉树那种逻辑的简约。
它追求的是在真实世界复杂的读写洪流中，永远保持那一两毫秒的稳定。

现在，让我们正式揭开 *B+* 树的神秘面纱。

寻找“六边形战士”：理想索引的准则

在通用的数据库场景中，我们需要一个索引能够：

- ① **动态自适应**：数据增加时，索引能自动长高、变宽，无需停机重组。
- ② **磁盘友好**：树必须足够“矮胖”，将 100 万行数据的检索控制在 3-4 次 I/O 内。
- ③ **读写平衡**：既要支持高频的实时插入，又要支持秒级的范围扫描。
- ④ **空间可控**：利用率虽然不满，但不能随插入而产生无限长的溢出链。

主角登场：B+ Tree



准备好进入动态分裂与平衡的世界吗？

本节内容概览

- 1 引入、基本概念
- 2 线性索引、静态索引
- 3 倒排索引
- 4 动态索引**
- 5 位索引
- 6 总结

为什么二叉树 (BST) 不适合做数据库索引？

- **逻辑问题：树高失控**

- 100 万个节点，二叉树高度约为 $\log_2(10^6) \approx 20$ 层。
- 每一层都可能是一次磁盘随机 I/O，20 次 I/O 需要约 0.2 秒，太慢了！

- **物理问题：不符合预读特性**

- 磁盘读取是以“页 (Page)”为单位的（通常 4KB 或 16KB）。
- 二叉树一个节点只存一个 Key，浪费了整页的存储空间，I/O 利用率极低。

结论

我们需要一种“矮胖”的树，让每个节点正好填满一个磁盘页，从而在单次 I/O 中读入尽可能多的 Key。

物理基础：磁盘预读与页存储

- **空间局部性**：当读取某字节时，计算机认为很可能紧接着读取它附近的字节。
- **磁盘页 (Page/Block)**：数据库从磁盘读数据不是按字节读，而是按“页”读。
- **B+ 树的适配性**：
 - B+ 树的一个节点大小通常设为系统页大小的整数倍。
 - “一读即一簇”：一次 I/O 就能把成百上千个索引项拉入内存。

B+ 树的三大进化

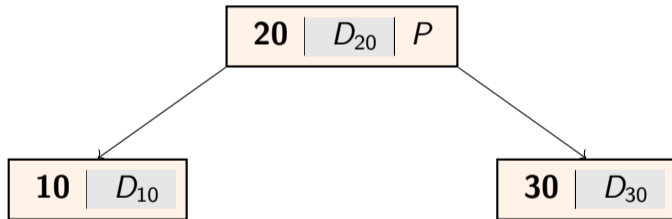
- 进化 1：从“静态”到“细胞分裂”
 - 痛点回溯：ISAM 靠“溢出链表”苟延残喘。
 - B+ 树方案：节点满了就“一分为二”，索引层级向上自动增长。
- 进化 2：从“瘦高”到“矮胖”
 - 痛点回溯：二叉树太高，磁盘 I/O 次数多到无法接受。
 - B+ 树方案：多叉平衡，让 100 万行数据的高度只有 3 层。
- 进化 3：从“树状”到“梳状”
 - 痛点回溯：B 树范围查询需要在层级间反复“横跳”。
 - B+ 树方案：数据全部沉降至底层，并用链表横向贯通。

核心结论

B+ 树不是一种纯粹的算法，它是针对磁盘物理特性和读写平衡的最优工程解。

B 树 (B-Tree): 一种平衡多路查找树

- **核心特征:** 每个节点既充当“索引”，也直接存储“数据记录”。
- **查询逻辑:** 如果在内部节点找到匹配的 Key，查询结束，无需下钻到叶子。



数据 D 分散在所有层级

深度对比：为什么数据库更爱 B+ 树？

| 特性 | B 树 (B-Tree) | B+ 树 (B+ Tree) |
|------|--------------|----------------|
| 数据存储 | 遍布所有节点 | 仅在叶子节点 |
| 扇出度 | 较低（数据占用空间大） | 极高（内部节点纯索引） |
| 查询路径 | 路径长度不固定 | 路径长度固定（稳定） |
| 范围查询 | 效率低（需跨层遍历） | 效率极高（链表扫描） |

结论：磁盘 I/O 是核心

由于 B+ 树内部节点不存数据，同样的磁盘页可以存放更多索引，从而使树更“矮”，大幅减少磁盘 I/O 次数。

B 树的“主战场”在哪？

- 1. 非关系型文件系统：如某些早期的文件系统分配表。
- 2. 频繁访问的高频 Key：
 - 如果某个数据被极频繁访问，B 树将其放在根节点，查找只需 1 次 I/O。
 - B+ 树则无论如何都需要下钻到最底层。
- 3. 内存数据库：
 - 在内存中，I/O 代价忽略不计，B 树的“快速返回”特性在特定算法中更有优势。

B 树与 B+ 树的差异

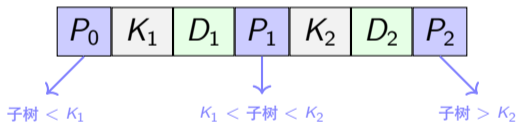
| 维度 | B 树 (B-Tree) | B+ 树 (B+ Tree) |
|------|------------------------|-----------------------|
| 结点结构 | 隔板模式: n 键对应 $n+1$ 指针 | 目录模式: n 键对应 n 指针 |
| 数据位置 | 任意结点 (找到即停) | 仅在叶子层 (必须到底部) |
| 键值重复 | 同一键值全树仅出现一次 | 同一键值可能在索引层重复出现 |
| 叶子关联 | 各叶子结点相互独立 | 双向链表连接所有叶子 |
| 适用场景 | 点查询 (Point Query) 频繁时 | 范围查询 (Range Scan) 频繁时 |

一句话概括

B 树是“数据在路边”的搜索树；B+ 树是“索引在云端，数据在脚下”的索引树。

结点结构：隔板模式

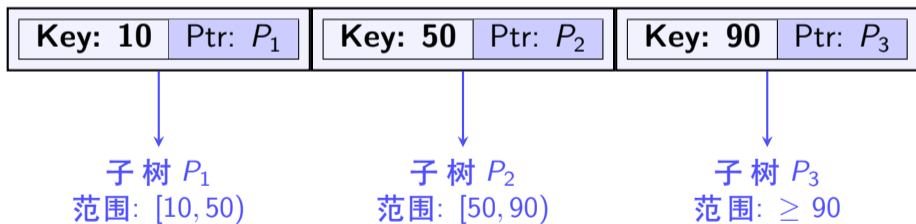
- $\{K_1, D_1\}$ 和 $\{K_2, D_2\}$ 可以看作是两个隔板
- 隔开了 3 颗子树



结点结构：目录模式

- $\{K_1, P_1\}, \{K_2, P_2\}$ 和 $\{K_3, P_3\}$ 是三个条目
- 每个条目代表一颗子树的键值下界及地址

内部节点 = [左边界, 子树指针] 数组



B+ 树的规格：阶数 (Order) 与填充因子

- **阶数 (m)**：一个节点最多拥有的子节点数量。
- **内部节点规则**：
 - 每个节点最多有 m 个子节点，最多 m 个键 (Key)。
 - 除根节点外，每个节点至少有 $\lceil m/2 \rceil$ 个子节点（保证空间利用率）。
- **叶子节点规则**：
 - 位于同一层，包含实际数据。
 - 节点间通过双向指针相连，形成一个有序链表。

为什么 m 很大？

若物理页为 16KB，Key 为 8B，指针为 6B，则 m 可达 1000 以上。
这意味着 3 层树就能支撑十亿级数据。

B+ 树的查找：单向向下，终点见分晓

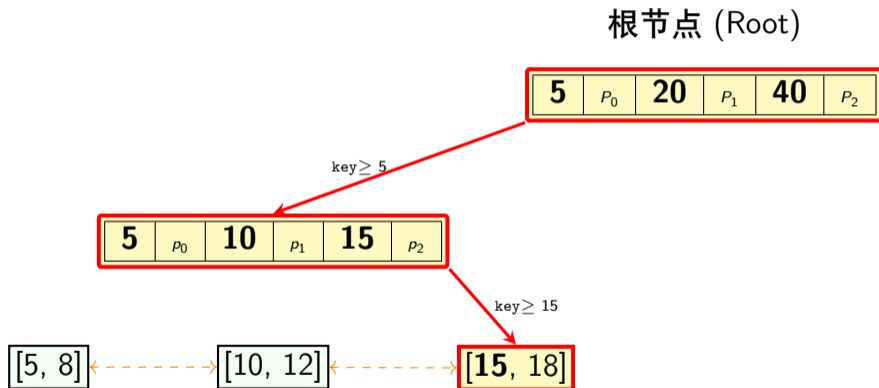
- 逻辑步骤：

- ① 根节点出发：比较目标值 K 与节点内的键。
- ② 区间判定：若 $K_i \leq K < K_{i+1}$ ，则沿指针 P_i 进入下一层。
- ③ 落叶归根：重复直到叶子节点。

- 特点：

- ① 路径等长：所有叶子都在同一层，查询性能极其稳定。
- ② 无中途拦截：即便内部节点有匹配的 Key，也必须走到叶子层（因为只有叶子有数据）。

B+ 树查找演示：寻找 Key = 15



B+ 树的杀手锏：范围查询

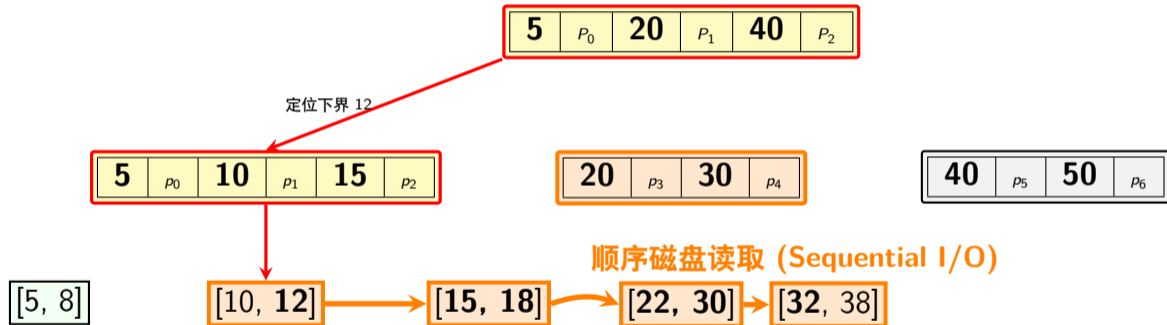
场景：查找所有 Age 介于 [20, 25] 的数据

- ① 索引查找：先通过树结构找到 Age=20 所在的叶子节点。
- ② 水平扫描：从该叶子节点开始，沿着底层的 Next 指针向右遍历。
- ③ 停止判定：直到遇到 Age > 25 的记录。

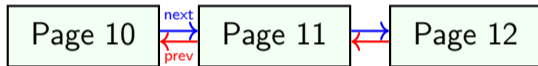
性能优势

- 只需要一次树的垂直下钻（随机 I/O）。
- 剩下的全是底层的顺序扫描（顺序 I/O），极大地利用了磁盘预读。

B+ 树范围查询演示：寻找 $12 \leq \text{Key} \leq 35$



为什么需要双向链表？



场景 A：倒序查找 *SELECT * FROM t ORDER BY id DESC*

场景 B：并发定位确保在页面分裂/合并时指针的一致性

B+ 树：数据库索引的“最终”方案

1. 结构化解耦 (Separation of Concerns)

- 内部结点：仅存 Key + Pointer，极致提高“扇出度 (Fan-out)”。
- 叶子结点：存储 Data，是数据的唯一终点。

2. 磁盘友好性 (Disk-Oriented Design)

- 结点大小与磁盘页 (Page) 对齐，将千万级数据的 I/O 压制在 3-4 次内。

3. 搜索全能型 (Universal Search)

- 点查询： $O(\log_m N)$ 稳定路径，所有记录检索成本相等。
- 范围查询：利用 Next 指针实现高速顺序扫描。
- 倒序查询：利用 Prev 指针无需排序直接反向遍历。

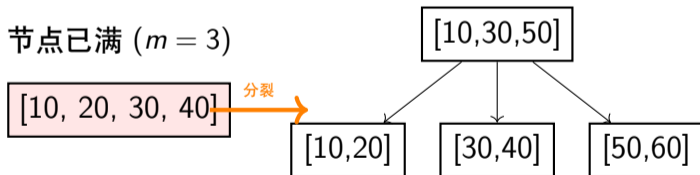
核心性能公式

树高 $h \approx \log_{\text{扇出度}}(\text{总记录数})$ 。

增大扇出度（让内部结点不存 Data）是 B+ 树比 B 树“更矮”的数学原因。

B+ 树的生长魔法：节点分裂 (Split)

- 规则：若插入后键值数 $> m$ ，则触发分裂。
- 分裂步骤：
 - ① 取出节点中位记录 (Median)。
 - ② 以中位为界，将原节点拆分为左右两个新节点。
 - ③ 向上拉升：将中位键 (或其副本) 推入父节点作为新的导航。



B+ 树的“细胞分裂”：从叶子到根

- **层级递归**：如果父节点也满了，父节点会继续分裂，直到根节点。
- **根节点分裂**：当根节点分裂时，会产生一个新的根，树的高度因此 +1。

生长特性总结

- **绝对平衡**：由于是从底部向上顶，所有叶子节点永远保持在同一层。
- **写代价**：单条数据的插入可能引发一系列的磁盘页分裂（这是为了读稳定性付出的成本）。

B+ 树的删除逻辑：保持紧凑

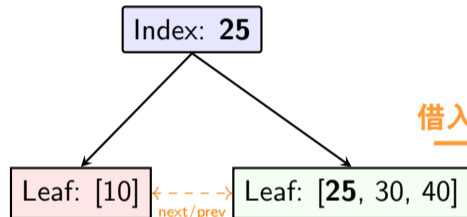
第一步（直接删除）：在叶子节点找到并移除该键。

第二步（判定阈值）：如果该节点中剩余键值数 $< \lceil m/2 \rceil$ ：

- ① **方案 A：旋转（向邻居借）：**如果左或右兄弟节点较富余，则借一个键过来，并更新父节点的索引。
- ② **方案 B：合并（Merge）：**如果邻居也很穷，则将当前节点与邻居合并为一个，并从父节点中删除对应的索引项。

B+ 树动态调整：旋转（向右兄弟借键）

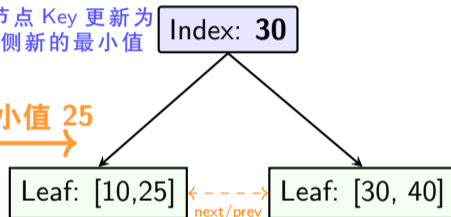
旋转前：删除后下溢 ($m = 3$)



旋转后：恢复紧凑

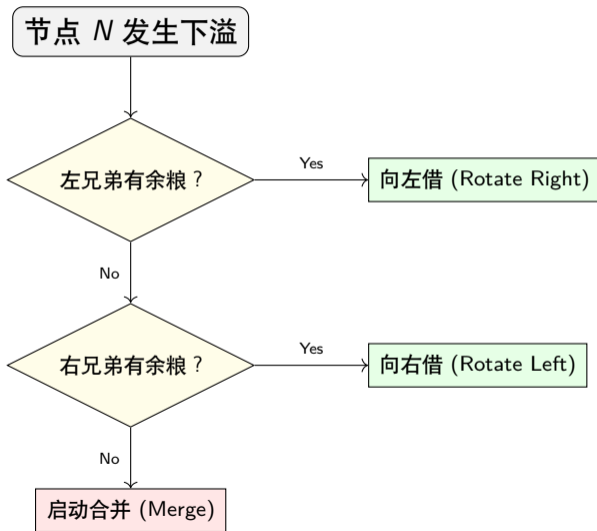
父节点 Key 更新为
右侧新的最小值

借入右侧最小值 25



- **局部调整**：旋转操作仅涉及相邻兄弟，不需要改变树的高度，开销极小。
- **一致性**：左叶子获得了 25，因此父节点必须将分隔值从 25 更新为 30。

决策逻辑：借左、借右还是合并？



工程细节：为什么“左邻右舍”也有优先级？

1. 局部性原理 (Locality)

- 左侧节点通常在索引扫描中已被加载到内存，借用成本低。

2. 减少抖动 (Anti-Thrashing)

- 避免与正在频繁分裂的右侧“增长点”发生冲突。

3. 并发控制 (Concurrency Control)

- 在多线程环境下，锁定左侧节点的顺序（通常按 Page ID 或键值顺序加锁）可以有效防止死锁。

结论

理论上虽然对称，但工程上倾向于“向稳定区域借，不向动荡区域并”。

B+ 树动态调整：合并 (Merge)

- 触发场景：

- ① 节点 N 发生下溢 (Key 数量 $< \lceil m/2 \rceil$)。
- ② 且左、右邻居节点也都处于“半满”下限，无法借出键值。

- 核心动作：

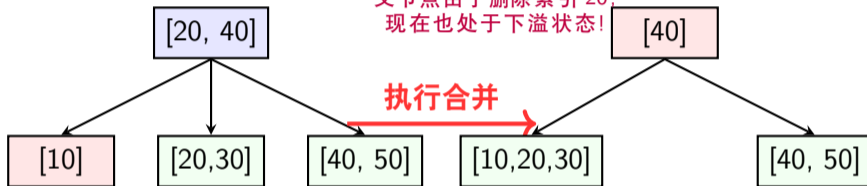
- 将节点 N 与其中一个邻居合并。
- 关键点：必须从父节点中删除指向被合并节点的索引项。

本质区别

“旋转”是水平方向的平衡，不改变树高；
“合并”是垂直方向的坍缩，可能导致树高降低。

B+ 树动态调整：合并 (Merge)

1. 删除后左侧下溢 ($m = 3$)



2. 合并完成

邻居只有 2 个键，借不到 (Borrow Fail)

- 精确状态：对于 $m = 3$ 树，键值数 < 2 触发下溢。邻居数量 = 2 则无法借出。
- 结构坍缩：父节点原来指向空节点的指针消失，导致父节点 Key 减少。

连锁反应：级联合并 (Cascading Merge)

- 向上蔓延：

- ① 叶子合并导致父节点删除一个键值。
- ② 如果父节点也因此低于半满，也可能需要与其兄弟合并。
- ③ 这一过程可能一直递归到根节点。

- 树高降低 (Shrinking)：

- 当根节点的最后两个子节点合并时，旧的根节点消失，新合并的节点成为新的根。
- 结果：树的高度减 1。

工程提示

在实际数据库中，为了减少这种昂贵的级联合并，通常会采用“延迟合并”策略：
允许页面的填充率暂时低于 50%，直到下探到某个更低的阈值（如 25%）

总结：B+ 树动态平衡操作对比

| 操作 | 触发条件 | 影响范围 | 树高变化 |
|-------------|-----------------|------------|------|
| 分裂 (Split) | 节点已满 (Overfull) | 向上蔓延 | 可能增加 |
| 旋转 (Rotate) | 下溢且邻居富余 | 仅父子节点 (局部) | 不变 |
| 合并 (Merge) | 下溢且邻居均穷 | 向上蔓延 | 可能降低 |

设计哲学：代价权衡

- 旋转是首选：代价极低，仅涉及指针和少量 Key 更新。
- 合并/分裂是次选：涉及 Page 的创建/销毁，可能引发级联反应。

工业界实现（如 InnoDB）

为减少磁盘 I/O 抖动，通常采用“非紧凑”策略：

- 允许节点填充率暂时低于 50%。
- 频繁操作时，优先标记删除而非立即合并。

总结：为什么 B+ 树成为数据库索引的首选？

- **动态平衡**：通过节点的 **分裂 (Split)** 与 **合并 (Merge)**，树的高度始终保持平衡，查询性能稳定在 $O(\log_m n)$ 。
- **彻底告别溢出区**：不像 ISAM 需要维护臃肿的链表，B+ 树通过改变树结构来吸收数据增长，无需定期物理重组。
- **磁盘 I/O 效率**：
 - 非叶子节点只存键值，不存数据，使得单个节点（页）能容纳更多分支（高扇出）。
 - 树的高度极低（通常 3-4 层即可支撑千万级数据）。

索引技术横向对比表

| 特性 | 静态线性索引 | ISAM | B+ 树 |
|-------|-----------|-----------|------------|
| 索引结构 | 固定排列 | 静态柱面/磁道级 | 动态树形结构 |
| 插入处理 | 整体移动数据 | 使用溢出区链表 | 节点自动分裂 |
| 查询稳定性 | 高 (若不常更新) | 随溢出增加而下降 | 极高 (始终平衡) |
| 维护成本 | 极高 (需重排) | 高 (需定期重组) | 低 (自动维护) |
| 典型应用 | 小规模只读数据 | 早期大型机系统 | 现代主流关系型数据库 |

B+ 树的挑战者：LSM 树 (Log-Structured Merge-Tree)

- **B+ 树的弱点**：每次插入都可能引发节点分裂和随机磁盘写入，写入性能受限。
- **LSM 树的思路**：
 - **完全顺序写**：所有更新先写内存 (MemTable)，达到一定大小后顺序刷新到磁盘。
 - **分层合并**：后台异步进行合并 (Compaction)，消除旧数据。
- **典型应用**：NoSQL 数据库 (LevelDB, RocksDB, Cassandra) 及 TiDB 底层存储。

结论

在“写多读少”的分布式场景下，LSM 树已经取代了 B+ 树。

内存数据库中的变革

- **瓶颈转移**：磁盘 I/O \rightarrow CPU 缓存失效 (Cache Miss) 和锁竞争。
- **替代结构**：
 - **Radix Tree (基数树)**：通过路径压缩减少指针跳转，提高缓存效率。
 - **Skip List (跳表)**：Redis 的核心，在高并发下比平衡树更容易实现无锁化。
 - **BW-Tree**：专门为多核 CPU 优化的无锁 B 树变种。

新一代需求：向量索引 (Vector Index)

- **局限：**B+ 树只能处理一维有序数据，无法处理成百上千维的向量。
- **新技术方向：**
 - **HNSW (分层导航小世界图)：**基于图论的高维检索。
 - **IVF (倒排文件索引)：**结合聚类分析的大规模向量索引。

当前格局

关系型查询看 B+ 树，大数据写入看 LSM 树，AI 语义检索看向量索引。

课堂练习 7: B+ 树的“高扇出”计算

- 已知参数:

- 磁盘块 (Page) 大小: $B = 4\text{ KB} = 4096\text{ Byte}$
- 索引关键码 (Key) 大小: $k = 8\text{ Byte}$
- 子节点指针 (Pointer) 大小: $p = 8\text{ Byte}$

- 计算目标:

- 一个非叶子节点 (内节点) 大约能容纳多少个分支?

请计算

请根据上述参数, 推导该 B+ 树的“分支因子” (Fan-out):

- Ⓐ 约 128 个
- Ⓑ 约 250 个
- Ⓒ 约 512 个
- Ⓓ 约 1024 个

练习 7 解析：为什么 B+ 树这么“矮”？

- 正确答案：B (约 250 个)
- 计算过程：
 - ① 每个索引单元大小 = $Key + Pointer = 8 + 8 = 16 \text{ Byte}$ 。
 - ② 最大条目数 $n = \lfloor 4096/16 \rfloor = 256$ 。
 - ③ 扣除页头等元数据开销，实际分支数通常约为 250。
- 性能推演（惊人的结论）：
 - 如果树高 $h = 3$ ，总记录数 $\approx 250^3 \approx 15,625,000$ 条。
 - 这意味着在千万级的数据库中，只需 3 次磁盘 I/O 就能定位到任何一条数据。
 - 这就是 B+ 树统治磁盘索引的物理奥秘！

课堂练习 8: B+ 树并发插入与锁耦合

- **背景**: 在高并发环境下, 多个线程同时插入数据。
- **挑战**: 如果对整棵树加锁, 性能会极差; 如果不加锁, 节点分裂时会破坏结构。

技术选择: 锁耦合 (Latch Crabbing)

在使用锁耦合技术进行插入操作时, 释放父节点锁的时机是:

- Ⓐ 一旦获取了子节点的锁, 立即释放父节点锁。
- Ⓑ 必须等到叶子节点修改完成并提交事务后。
- Ⓒ 获取子节点锁并确认该子节点“安全”(即插入不会导致分裂)后。
- Ⓓ 永远不释放, 直到搜索到达叶子节点。

练习 8 解析：攀岩式加锁

- 正确答案：C
- 工程逻辑：
 - 为什么叫 Crabbing？像螃蟹走路或爬山，手抓稳下一个支点且确认安全，才放开上一个。
 - “安全”的定义：如果子节点目前只有 $n < Max$ 个元素，那么插入新项不会触发分裂，也就不会改动父节点。此时释放父锁，其他线程就能访问父节点。
 - 核心价值：极大地提高了 B+ 树高层的并发吞吐量。

课堂练习 9: B+ 树的“懒惰”合并策略

- **场景**: 某系统执行了大量随机删除, 导致许多 B+ 树节点的填充率跌破了 50% (半满以下)。
- **现象**: 数据库并没有立即触发节点合并 (Merge) 操作。

深度思考

这种允许“半满”节点长期存在的策略, 主要是为了解决什么问题?

- Ⓐ 节省磁盘存储空间。
- Ⓑ 降低查询时的磁盘 I/O。
- Ⓒ 防止“抖动” (Thrashing) 现象, 即频繁合并又立即分裂。
- Ⓓ 强制索引转为稀疏索引。

练习 9 解析：平衡稳定性与代价

- 正确答案：C

- 工程逻辑：

- 抖动风险：如果 50% 是硬性红线，那么在临界点附近的一次插入和一次删除就会导致昂贵的“合并-再分裂”循环。
- 迟滞效应：工业界实现（如 InnoDB）通常允许节点填充率更低，甚至只在页面完全为空时才回收，以牺牲空间换取系统的运行平稳度。

课堂练习 10：前缀压缩 (Prefix Compression)

- **技术**：在非叶子节点中，不存储完整的字符串键值，只存足以区分分支的最短前缀。
- **目标**：显著增加分支因子 (Fan-out)，让树更矮。

代价分析

这种优化技术最显著的副作用 (Side Effect) 是什么？

- Ⓐ 导致树的高度增加。
- Ⓑ 增加了单次查询在节点内部对比时的 CPU 负载。
- Ⓒ 导致无法进行范围扫描。
- Ⓓ 索引会丢失原本的有序性。

练习 10 解析：空间与计算的博弈

- 正确答案：B
- 工程逻辑：
 - CPU vs I/O：数据库设计的永恒主题。前缀压缩减少了磁盘 I/O（树更矮、节点存更多项），但增加了 CPU 解析变长前缀的逻辑负担。
 - 结论：由于磁盘 I/O 比 CPU 计算慢好几个数量级，这个交换在绝大多数场景下是极其划算的。

课堂练习 11: B+ 树与 SSD 存储特性

- **背景**: SSD 不支持直接覆盖写 (Overwrite), 覆盖前必须擦除。
- **挑战**: B+ 树传统的“原地更新”会导致 SSD 写放大和寿命缩短。

架构演进

现代存储引擎 (如 WiredTiger) 如何改进 B+ 树以适应 SSD ?

- Ⓐ 强制使用全表扫描。
- Ⓑ 采用写时复制 (Copy-on-write), 将改动写到新块而非覆盖旧块。
- Ⓒ 禁止所有插入操作。
- Ⓓ 将索引项大小扩大到 4KB。

练习 11 解析：顺应硬件趋势

- 正确答案：B
- 工程逻辑：
 - 影子分页：修改后的页面被写到磁盘的空闲块，然后更新父节点指针。
 - 优势：避免了昂贵的“擦除-写入”周期，且天然支持快照隔离（Snapshot Isolation），因为旧版本数据在被回收前依然存在。

课堂练习 12：分布式数据库的索引选择

- 场景：一个分布式数据库集群有 100 台服务器。

架构抉择

为什么通常不使用一棵覆盖全集群的巨大 B+ 树，而选择分片局部索引？

- Ⓐ 因为网络延迟无法实现二分查找。
- Ⓑ 因为 B+ 树不支持分布式数据类型。
- Ⓒ 因为根节点（Root）会成为访问瓶颈和单点故障。
- Ⓓ 因为分布式环境下无法分裂节点。

练习 12 解析：去中心化的必然

- **正确答案：C**
- **工程逻辑：**
 - **根节点压力：**在 B+ 树中，无论查什么，第一步永远是访问根节点。在分布式场景下，这意味着全球的流量都会打在存根节点的那一台机器上。
 - **现代方案：**采用一致性哈希或 Range 分片，每台机器维护自己的局部 B+ 树，顶层通过全局元数据路由，分散压力。

本节内容概览

- 1 引入、基本概念
- 2 线性索引、静态索引
- 3 倒排索引
- 4 动态索引
- 5 位索引**
- 6 总结

位索引 (Bitmap Index)

- 适用场景：低基数列 (Low Cardinality)，即取值重复率极高的字段。
- 存储结构：每一个取值对应一个二进制位串 (Bit Vector)。

| 行号 | 性别 | 男 (Bit) | 女 (Bit) | 已缴费 (Bit) |
|----|----|---------|---------|-----------|
| 1 | 男 | 1 | 0 | 1 |
| 2 | 女 | 0 | 1 | 1 |
| 3 | 男 | 1 | 0 | 0 |
| 4 | 男 | 1 | 0 | 1 |

- 多条件查询：查找“已缴费的男性”？
- 位运算： $[1011]$ (男) **AND** $[1101]$ (已缴费) = $[1001]$ → 第 1, 4 行命中。

位索引的优缺点分析

- **核心优势：**

- ① **空间效率：**对于低基数数据，压缩后的位向量占用空间极小。
- ② **计算神速：**现代 CPU 对位运算有硬件级优化，处理速度远超 B+ 树的逐行对比。
- ③ **复杂组合：**处理多个 'AND/OR' 条件时，优势极其明显。

- **致命弱点：**

- **锁冲突：**插入/更新某一行时，往往需要锁定整个位向量块，不适合频繁写入 (OLTP) 场景。
- **高基数灾难：**如果是“身份证号”，位索引的大小会爆炸。

典型应用

数据仓库 (OLAP)、报表系统、分析型数据库 (如 Oracle, ClickHouse)。

A=Analytics, T=Transaction

位索引进阶：Roaring Bitmaps 压缩技术

- **挑战**：当数据稀疏时，位图中存在大量连续的 0，造成空间浪费。
- **Roaring Bitmap 方案**：将数据空间划分为 2^{16} 大小的分桶（Container）。
 - **Bitmap Container**：数据密集时，直接存 8KB 的位图。
 - **Array Container**：数据稀疏时，只存具体的行号列表（有序数组）。
 - **Run Container**：存在长段连续 1 时，存起始位置和长度。

结果

在保持 $O(1)$ 查找和高效位运算的同时，空间占用通常比 B+ 树小 10-100 倍。

工业级 OLAP 与位图索引 (Bitmap)

- ClickHouse (国际标杆)

- Roaring Bitmaps: 高性能位图压缩库的深度集成。
- 功能: 擅长海量 ID 的去重计数 (Count Distinct) 与集合运算。

- StarRocks / Doris (国产之光)

- 显式位图索引: 针对低基数维度字段自动构建。
- Runtime Filter: 在 Shuffle 过程中利用位图过滤无关数据。

为什么它们不选 B+ 树?

OLAP 面对的是千万量级数据的“扫射”而非“点射”。B+ 树在面对 *AND* 组合查询时需要多次回表，而位图索引只需在内存中做位运算，效率高出 10-100 倍。

位索引进阶：位片索引 (Bit-sliced Index)

- 原理：不按值分类，而是按“二进制位”进行分解存储。
- 示例：存储数值范围 0-7（需 3 个位片 B_2, B_1, B_0 ）。

| 行号 | 数值 | 位片 B_2 (2^2) | 位片 B_1 (2^1) | 位片 B_0 (2^0) |
|----|---------|--------------------|--------------------|--------------------|
| 1 | 5 (101) | 1 | 0 | 1 |
| 2 | 3 (011) | 0 | 1 | 1 |

- 神奇的聚合：计算总和 $Sum = 4 \times count(B_2) + 2 \times count(B_1) + 1 \times count(B_0)$ 。
- 优势：在大规模统计查询中，通过统计位图中 1 的个数 (Population Count) 即可完成，无需访问原始记录。

位片索引 (Bit-sliced Index): 数值运算的加速器

- **核心思想**: 将数值字段按二进制位 (Bit) 拆分, 每一位权重建立一个位图。
- **OLAP 优势**:
 - **聚合加速**: 利用 *PopCount* 指令快速计算 SUM/AVG。
 - **谓词下推**: 数值比较 (如 $X > k$) 转化为多层位图 AND/OR 运算。
- **国产应用案例**:
 - **Apache Doris / StarRocks**: 在处理海量数值指标的聚合查询时, 利用类似思想优化列式存储的读取过程。

场景对比

位图索引 → “我们要找所有的‘经理’。” (分类属性)

位片索引 → “我们要计算所有员工的‘薪水总和’。” (数值属性)

本节内容概览

- 1 引入、基本概念
- 2 线性索引、静态索引
- 3 倒排索引
- 4 动态索引
- 5 位索引
- 6 总结**

1.1 稀疏索引与物理地址

- **核心原理**：稀疏索引记录的是数据块的物理起始地址。
- **脆弱性分析**：一旦进行磁盘碎片整理或数据迁移，索引项必须更新。

结论

稀疏索引与物理存储“强耦合”，维护成本随物理变动而增加。

1.2 二级线性索引的 I/O 代价

- **查询路径：**外层索引 (内存) → 内层索引块 (磁盘) → 实际数据 (磁盘)。
- **计算：**对于二级结构，即使外层在内存中，最少也需要 2 次磁盘 I/O。

2.1 ISAM 的核心设计

- 分级结构：柱面索引 + 磁道索引。
- 溢出机制：主存储区 + 溢出区链表。
- 局限：静态索引无法随数据增长而改变，溢出链表过长会导致性能直线下降。

3.1 B+ 树的物理优势：高扇出

- **高扇出 (High Fan-out)**：内节点不存数据，仅存键值和指针。
- **典型计算**：4KB 页大小下，单节点可容纳约 250 个分支。
- **结论**：3 层高度即可支撑千万级数据，极大减少磁盘寻道。

3.2 并发与高性能：锁耦合 (Latch Crabbing)

- **原理：**获取子锁且确认“安全”后才释放父锁。
- **价值：**通过细粒度加锁，解决了高并发插入时的树结构竞争瓶颈。

3.3 现代挑战：SSD 与 LSM 树

- **B+ 树弱点**：随机原地更新导致 SSD 写放大。
- **LSM 树方案**：写时复制 (COW) 与顺序追加写，顺应闪存物理特性。

4.1 位索引与位运算

- 适用场景：低基数列（性别、缴费状态）。
- 黑科技：Roaring Bitmap（混合压缩）与位片索引（Bit-sliced）。
- 优势：利用硬件级位运算实现极速聚合查询。

索引选择指南

- OLTP (交易): 首选 B+ 树, 追求平衡与稳定。
- OLAP (分析): 首选 位索引 / 列存储, 追求批量计算速度。
- 高频写 / 日志: 首选 LSM 树, 追求顺序写吞吐。

欢迎提问！

你的疑惑，我的动力