深入浅出

杨子凡

Jul 18, 2025

数据库索引如同图书馆的目录系统，能避免「逐页查找书籍」式的全表扫描操作。其核心价值在于解决磁盘 **I/O** 瓶颈问题，通过建立辅助数据结构实现键值与数据行位置的映射关系。这种设计虽然会带来写操作开销增加和额外存储空间的代价，但对点查询和范围查询的性能提升往往是数量级的。本文旨在解析主流索引结构的内部机制，并提供经过实践验证的优化策略。

1　核心数据结构：索引的基石

1.1　B-Tree：关系型数据库的绝对主流

作为平衡多路搜索树，B-Tree 通过自平衡特性保证所有叶子节点位于同一层级。其节点包含键值（Key）和指向子节点或数据行的指针（Pointer）。当执行查询时，系统从根节点开始逐层比较键值，最终定位到目标叶子节点。插入操作可能引发节点分裂的连锁反应，例如当新值导致节点超出容量限制时，会分裂为两个节点并向父节点插入中间键值。B-Tree 的优势在于高效处理等值查询、范围查询和排序操作，但其随机插入可能导致频繁分裂影响写性能。

1.2　B+Tree：B-Tree 的优化变种

B+Tree 的核心革新在于数据仅存储在叶子节点，内部节点仅保留导航用的键值和指针。叶子节点通过双向链表连接，这使得范围查询只需遍历链表即可完成。在 MySQL InnoDB 的实现中，叶子节点存储的指针直接指向聚簇索引的数据行。其优势包括更稳定的查询路径长度（所有查询都必须到达叶子节点）和更高的缓存效率（内部节点更紧凑）。B+Tree 的查询时间复杂度为 $O(\log_b n)$，其中 b 为节点分支因子，n 为数据总量。

1.3　哈希索引

基于哈希表实现的索引通过对键值计算哈希值定位到哈希桶。每个桶内通过链表解决哈希冲突问题。哈希索引的等值查询时间复杂度接近 $O(1)$，典型实现如下：

```
-- MySQL MEMORY 引擎创建哈希索引
CREATE TABLE user_session (
   session_id CHAR(36) PRIMARY KEY,
   user_data JSON
) ENGINE=MEMORY;
```

此代码创建了基于内存的哈希索引，session_id 的哈希值直接映射到内存地址。但其致命缺陷是不支持范围查询和排序，且哈希冲突可能引发性能退化。

1.4 LSM-Tree：应对高写入负载

LSM-Tree 将随机写转换为顺序写以提升吞吐量。写入操作首先进入内存中的 **MemTable**（通常采用跳表实现），当达到阈值后冻结为 **Immutable MemTable** 并刷盘为有序的 **SSTable** 文件。磁盘上的 SSTable 分层存储，后台 **Compaction** 进程负责合并文件并清理过期数据。读取时需要从 MemTable 逐层向下搜索 SSTable，Bloom Filter 可加速判断键值是否存在。LSM-Tree 的写放大系数（Write Amplification Factor）可表示为：

$$WAF = \frac{\text{实际写入数据量}}{\text{逻辑写入数据量}}$$

通过优化 Compaction 策略可有效降低 WAF 值。

1.5 其他索引结构

位图索引为每个低基数列的唯一值创建位图向量，例如性别字段的位图可表示为 `male: 1010`，`female: 0101`。全文索引基于倒排索引实现，存储单词到文档列表的映射。空间索引如 R-Tree 使用最小边界矩形（MBR）组织空间对象，其查询复杂度为 $O(n^{1-1/d} + k)$，其中 d 为维度数，k 为结果数。

2 索引的内部实现关键点

2.1 聚簇索引与非聚簇索引

在 InnoDB 引擎中，聚簇索引的叶子节点直接存储数据行，表数据按主键物理排序。这解释了为何主键范围查询极快：

```
-- 聚簇索引范围查询
SELECT * FROM orders WHERE order_id BETWEEN 1000 AND 2000;
```

此查询只需遍历索引的连续叶子节点。相反，非聚簇索引的叶子节点仅存储主键值，查询需要二次查找（回表）：

```
-- 非聚簇索引引发回表
SELECT * FROM users WHERE email = 'user@example.com';
```

若 `email` 字段建有非聚簇索引，需先查索引获取主键，再通过主键获取数据行。

2.2 覆盖索引与复合索引

覆盖索引通过在索引中包含查询所需的所有列避免回表：

```
-- 创建覆盖索引
CREATE INDEX idx_cover ON orders (customer_id, order_date) INCLUDE (total_amount);

-- 查询可直接使用索引
```

```
SELECT customer_id, order_date, total_amount
FROM orders
WHERE customer_id = 123;
```

复合索引则需遵循最左前缀原则。索引 (A,B,C) 能优化 WHERE A=1 AND B>2 但无法优化 WHERE B=2。其排序规则满足：

$$\text{Key}_{\text{composite}} = \langle A, B, C \rangle \quad \text{按字典序排序}$$

2.3 索引键的选择性与基数

索引选择性计算公式为：

$$\text{Selectivity} = \frac{\text{COUNT(DISTINCT column)}}{\text{COUNT(*)}}$$

当选择性低于 0.03 时，全表扫描可能优于索引扫描。优化器使用直方图统计信息估算选择性，定期执行 ANALYZE TABLE 更新统计信息至关重要。

3 索引优化策略

3.1 设计原则与实践

索引设计必须基于实际查询模式。高频查询条件应作为索引前导列，避免创建超过 5 列的复合索引。主键设计推荐使用自增整数而非 UUIDv4，后者可能导致聚簇索引的页分裂率提升 30% 以上。覆盖索引应包含 SELECT 列表中的列：

```
-- 优化前：需要回表
SELECT username, email FROM users WHERE age > 30;

-- 创建覆盖索引后
CREATE INDEX idx_age_cover ON users (age) INCLUDE (username, email);
```

3.2 避免索引失效陷阱

常见失效场景包括：

- 隐式类型转换：WHERE user_id = '123'（user_id 为整型）
- 函数操作：WHERE YEAR(create_time) = 2023
- 前导通配符：WHERE name LIKE '%son'
- **OR** 条件未优化：应改写为 UNION ALL 结构

执行计划分析是优化的核心工具：

```
EXPLAIN SELECT * FROM products
WHERE category_id = 5 AND price > 100;
```

输出中的 type: range 表示范围索引扫描，Extra: Using where 说明进行了额外过滤。

3.3 索引维护与监控

索引重组（ALTER INDEX ... REORGANIZE）在线整理页碎片，而重建索引（ALTER INDEX ... REBUILD）需要锁表但效果更彻底。通过监控视图可识别无用索引：

```
-- PostgreSQL 查看索引使用统计
SELECT * FROM pg_stat_user_indexes;
```

B+Tree 在 OLTP 场景仍占主导地位，而 LSM-Tree 在写入密集型系统表现突出。自适应索引技术如 Oracle 的 Automatic Indexing 已能动态创建索引。索引下推（Index Condition Pushdown）将过滤条件提前到存储引擎层执行，减少 60% 以上的回表操作。实践建议始终遵循：基于 EXPLAIN 分析验证索引效果，定期清理使用率低于 1% 的索引，并深入理解特定数据库的索引实现差异。