Chapter 3

索引结构及使用

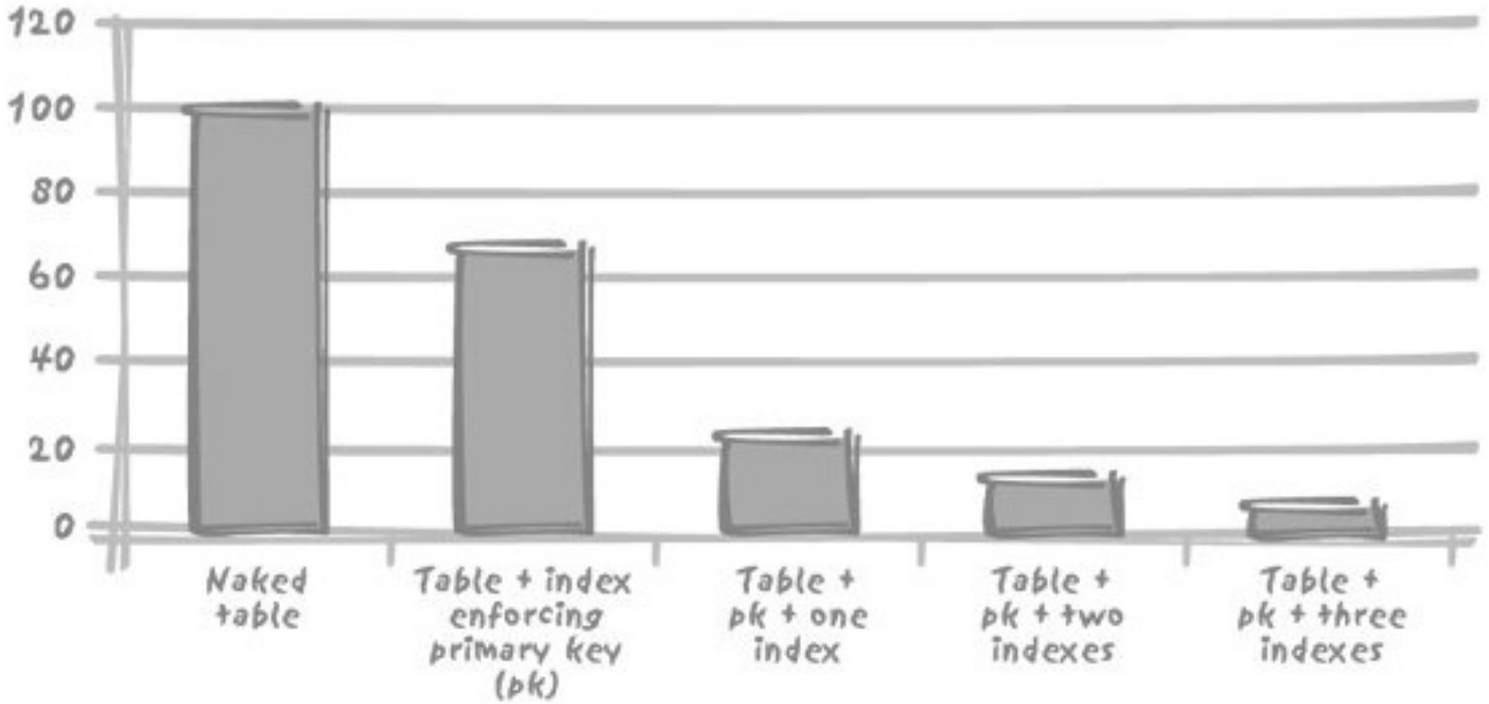
无论是多么复杂的ORM工具，在精妙和复杂的索引面前都是“浮云”

找到“切入点”

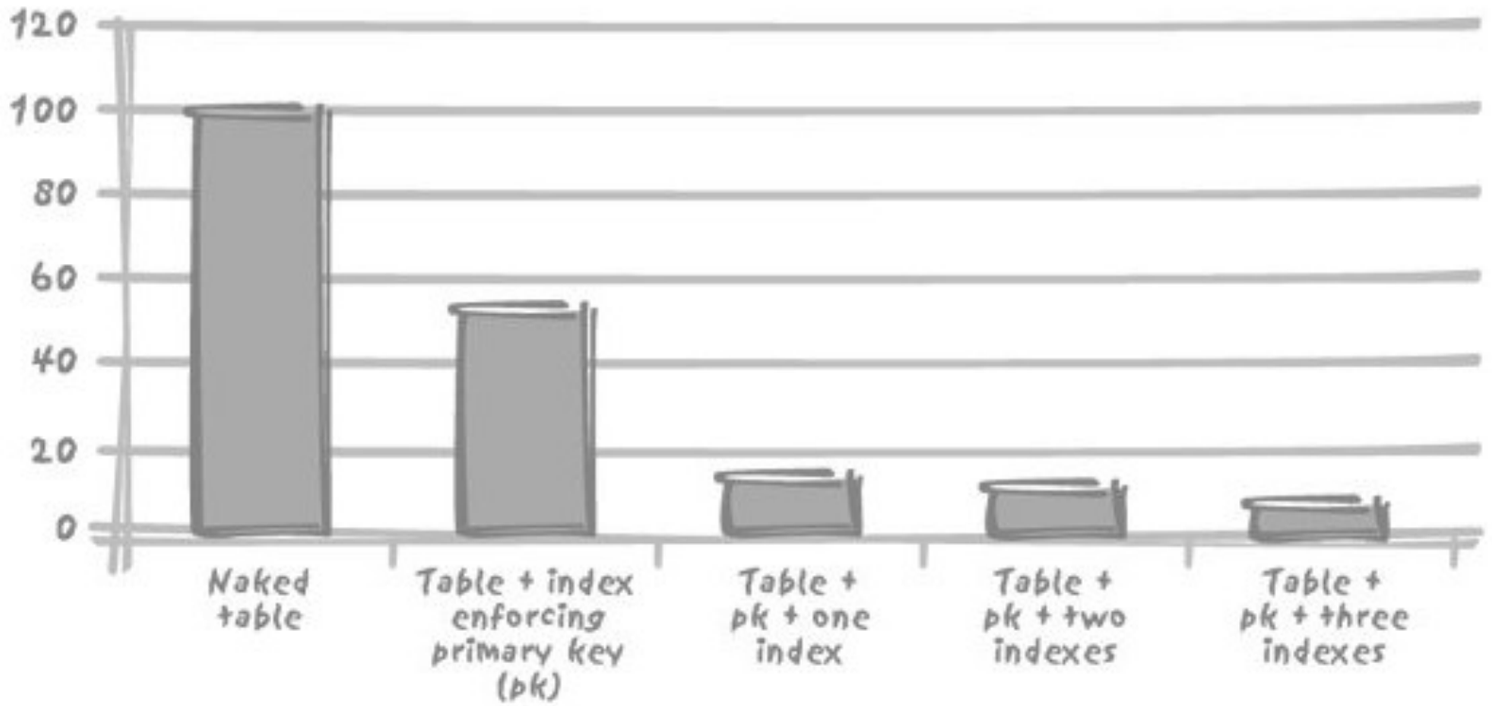
- 建立索引的基础
  - 检索条件、子集大小
- 索引不是万能的
  - 索引也有可能降低查询性能
- 索引的开销
  - 磁盘空间的开销、处理开销等

找到“切入点”（cont’）

- Oracle



- MySQL



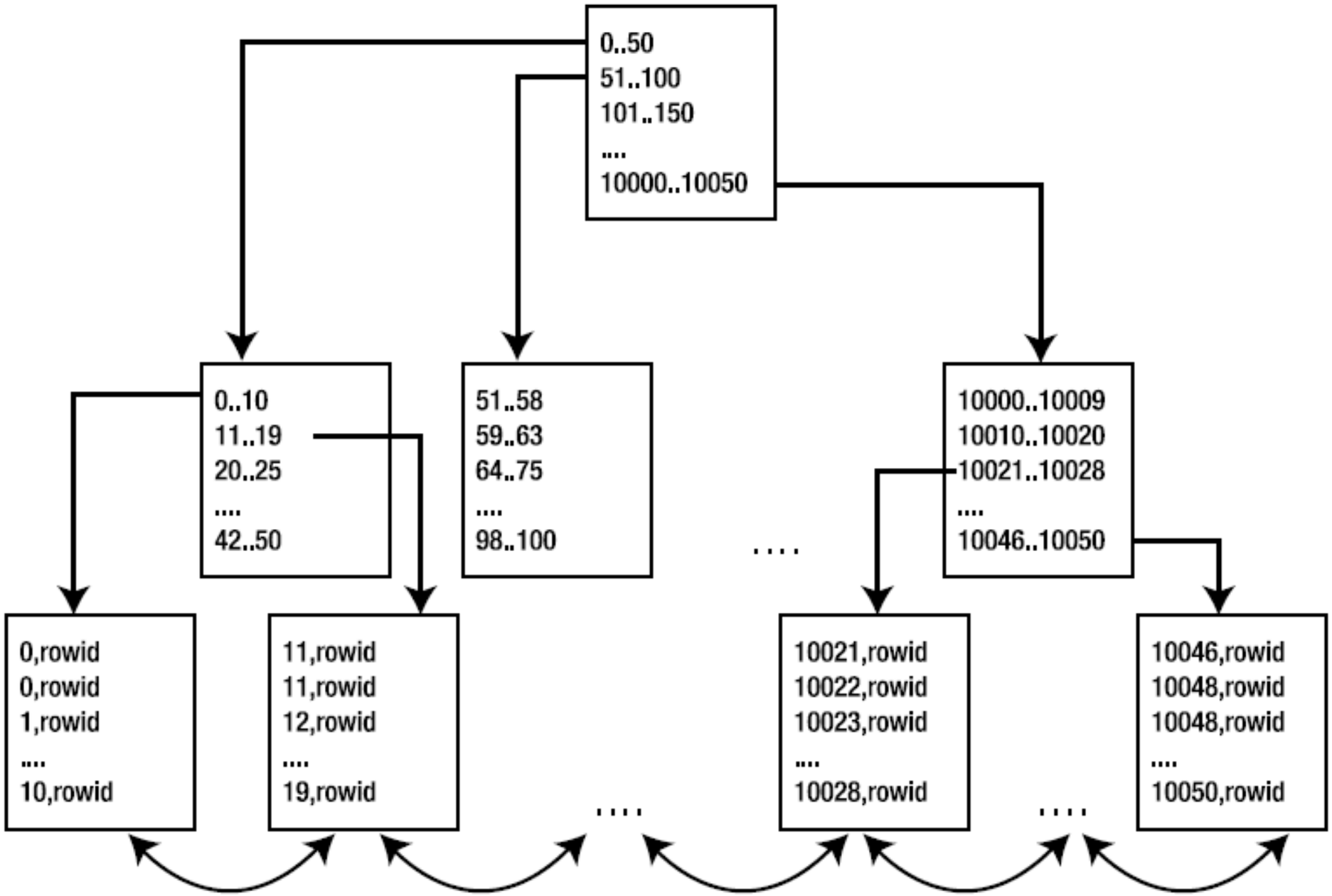
找到“切入点”（cont’）

- 索引始终是数据库中极重要的组成部分
  - 通用目的或事务处理型数据库系统
  - 决策支持系统
- 事务处理型数据库中 “太多索引≈设计不够稳定”

索引与目录

- 目录和索引是两种不同的机制
- 索引是一种以原子粒度访问数据的手段，而不是为了检索大量数据的

B-Tree（B+Tree）的结构



让索引发挥作用

- 索引的使用是否合理，首先取决于它是否有用。
- 长久以来，判断索引适用性的依据是检索比率（retrieval ratios）
- 什么时候应该使用B树索引
  - 仅当要通过索引访问表中很少一部分行
  - 如果要处理表中多行，而且可以使用索引而不用表

B树索引的适用范围

- 全值匹配
- 匹配最左前缀
- 匹配列前缀
- 匹配范围值
- 精确匹配某一列并范围匹配另外一列
- 只访问索引的查询

B树索引的限制

- 如果不是按照索引的最左列开始查找，则无法使用索引
- 不能跳过索引中的列
  - Key（last_name, first_name, dob）
    - 无法用于查找姓为Smith并且在某个特定日期出生的人
- 如果查询中有某个列的范围查询，则其右边所有列都无法使用索引优化查找
  - 例如：Where last_name=‘Smith’ And first_name Like ‘J%’ AND dob=‘1976-4-25’，这个查询只能使用索引的前两列，因为Like是一个范围条件
  - 可以使用多个等于条件代替范围条件
- 所以：索引列的顺序是最重要的关注点

让索引发挥作用（cont’）

- 如何使用索引的因素错综复杂
  - 磁盘访问
  - 内存访问
  - 记录存储
  - ……

哈希索引

- 在MySQL中，只有Memory引擎显性支持哈希索引
- 这也是Memory引擎的默认索引类型
- 与众不同的是，Memory引擎支持非唯一哈希索引

```
mysql> SELECT * FROM testhash;
+--------+-----------+
| fname  | lname     |
+--------+-----------+
| Arjen  | Lentz     |
| Baron  | Schwartz  |
| Peter  | Zaitsev   |
| Vadim  | Tkachenko |
+--------+-----------+
```

```
f('Arjen')= 2323
f('Baron')= 7437
f('Peter')= 8784
f('Vadim')= 2458
```

Slot	Value
2323	Pointer to row 1
2458	Pointer to row 4
7437	Pointer to row 2
8784	Pointer to row 3

哈希索引的限制

- 只包含哈希值和行指针，而不存储字段值
- 无法按照索引值排序
- 不支持部分索引列匹配查找
- 只支持等值比较查询，包括=、IN()、<=>
- 访问哈希索引的数据非常快，除非有很多哈希冲突
- 哈希冲突很多的时候，索引维护代价也很高

哈希索引的适用范围

- 只适用于特定场合，一旦适用，性能提升显著
- 数据仓库的星型schema
- InnoDB引擎有一个特殊功能叫“自适应哈希索引（adaptive hash index）”,当某些索引值被使用得非常频繁时，会在内存中基于B-tree索引之上再创建一个哈希索引

创建自定义哈希索引

- 只需要很小的索引就可以为超长的Key创建索引

```
mysql> SELECT id FROM url WHERE url="http://www.mysql.com";

mysql> SELECT id FROM url WHERE url="http://www.mysql.com"
    ->    AND url_crc=CRC32("http://www.mysql.com");
```

- 缺点是需要手动维护哈希值

```
CREATE TABLE pseudohash (
   id int unsigned NOT NULL auto_increment,
   url varchar(255) NOT NULL,
   url_crc int unsigned NOT NULL DEFAULT 0,
   PRIMARY KEY(id)
);
```

```
DELIMITER //

CREATE TRIGGER pseudohash_crc_ins BEFORE INSERT ON pseudohash FOR EACH ROW BEGIN
SET NEW.url_crc=crc32(NEW.url);
END;
//

CREATE TRIGGER pseudohash_crc_upd BEFORE UPDATE ON pseudohash FOR EACH ROW BEGIN
SET NEW.url_crc=crc32(NEW.url);
END;
//

DELIMITER ;
```

创建自定义哈希索引

- 记住不要使用SHA1(),MD5()作为哈希函数
- CRC()会出现大量的哈希冲突，可以考虑64位哈希函数

```
mysql> SELECT CONV(RIGHT(MD5('http://www.mysql.com/'), 16), 16, 10) AS HASH64;
+--------------------+
| HASH64             |
+--------------------+
| 9761173720318281581 |
+--------------------+
```

- 注意：必须在where子句中包含常量值

```
mysql> SELECT id FROM url WHERE url_crc=CRC32("http://www.mysql.com")
    ->    AND url="http://www.mysql.com";
```

- “生日悖论”CRC32() 索引93000条记录的时候冲突概率1%
- FNV64（）

其他索引类型

- 空间数据索引（R-Tree）： PostgreSQL的PostGIS
- 位图索引（Bitmap Index）： Oracle
- 位图联结索引（Bitmap join index）:Oracle
- 函数索引（function-based index）
- 全文索引（完全不是Where条件匹配，更像搜索引擎）

函数和类型转换对索引的影响

- Where f (indexed_col) = ‘ some value‘
- 这种检索条件会使索引无法发挥作用
  - 日期函数
  - 隐式类型转换
- 基于函数的索引（function-based index）
  - Functional index
  - Index extension
  - Index on computed column

索引与外键

- 系统地对表的外键加上索引的做法非常普遍
  - 但是为什么呢?
  - 有例外吗?

- 建立索引必须有理由，无论是对外键，或是其他字段都是如此

同一字段，多个索引

- 如果系统为外键自动增加索引，常常会导致同一字段属于多个索引的情况
  - Orders—Order_Details—Articles

- 为每个外键建立索引，可能会导致多余索引

索引合并策略

- MySQL老版本中，对各个字段分别构建索引是不好的措施

```
CREATE TABLE t (
  c1 INT,
  c2 INT,
  c3 INT,
  KEY(c1),
  KEY(c2),
  KEY(c3)
);
```

```
mysql> SELECT film_id, actor_id FROM sakila.film_actor
    -> WHERE actor_id = 1 OR film_id = 1;
```

```
mysql> SELECT film_id, actor_id FROM sakila.film_actor WHERE actor_id = 1
    -> UNION ALL
    -> SELECT film_id, actor_id FROM sakila.film_actor WHERE film_id = 1
    ->    AND actor_id <> 1;
```

- 虽然MySQL和Oracle都有索引合并的策略，但实际上出现索引合并只能证明索引构建的很糟糕
  - 消耗大量CPU和内存资源；不会计入查询成本中，造成选择错误

索引列顺序的选择

- 复合键索引的好处

- 复合键索引的关键是：将选择性最高的列放到索引最前列

```
SELECT * FROM payment WHERE staff_id = 2 AND customer_id = 584;
```

```
mysql> SELECT SUM(staff_id = 2), SUM(customer_id = 584) FROM payment\G
*************************** 1. row ***************************
     SUM(staff_id = 2): 7992
SUM(customer_id = 584): 30
```

```
mysql> SELECT SUM(staff_id = 2) FROM payment WHERE customer_id = 584\G
*************************** 1. row ***************************
SUM(staff_id = 2): 17
```

索引列顺序的选择

- 索引选择的结果非常依赖具体过滤值，但一般的情况该如何选择呢?

- 经验法则：考虑全局基数和选择性

```
mysql> SELECT COUNT(DISTINCT staff_id)/COUNT(*) AS staff_id_selectivity,
    > COUNT(DISTINCT customer_id)/COUNT(*) AS customer_id_selectivity,
    > COUNT(*)
    > FROM payment\G
*************************** 1. row ***************************
   staff_id_selectivity: 0.0001
customer_id_selectivity: 0.0373
              COUNT(*): 16049
```

索引列顺序选择的问题

- 特异值带来的差异（guest用户，导入数据的问题）

```
mysql> SELECT COUNT(DISTINCT threadId) AS COUNT_VALUE
    -> FROM Message
    -> WHERE (groupId = 10137) AND (userId = 1288826) AND (anonymous = 0)
    -> ORDER BY priority DESC, modifiedDate DESC
```

```
mysql> SELECT COUNT(*), SUM(groupId = 10137),
    -> SUM(userId = 1288826), SUM(anonymous = 0)
    -> FROM Message\G
*************************** 1. row ***************************
             count(*): 4142217
 sum(groupId = 10137): 4092654
sum(userId = 1288826): 1288496
   sum(anonymous = 0): 4141934
```

索引列选择需要注意的问题

- 一个复合键索引应付更多的状况
- 可选择性的列未必是唯一的指标，也许频繁性也是
- 将范围检索放在最后
- 避免出现多个范围检索
  - 用IN（）替代范围检索

系统生成键

- 系统生产序列号，远好于
  - 寻找当前最大值并加1
  - 用一个专用表保存”下一个值“且加锁更新
- 但如果插入并发性过高，在主键索引的创建操作上会发生十分严重的资源竞争
- 解决方案
  - 反向键索引或叫逆向索引（reverse index）
  - 哈希索引（hash indexing）

为什么没有使用我的索引

- 情况1：我们在使用B+树索引，而且谓词中没有使用索引的最前列
  - T，T(X,Y)上有索引，做SELECT * FROM T WHERE Y=5
- 跳跃式索引（仅CBO）

为什么没有使用我的索引？（cont’）

- 情况2：使用SELECT COUNT(*) FROM T，而且T上有索引，但是优化器仍然全表扫描

- 情况3：对于一个有索引的列作出函数查询
  - Select * from t where f(indexed_col) = value

- 情况4：隐形函数查询

为什么没有使用我的索引？（cont’）

- 情况5：此时如果用了索引，实际反而会更慢

- 情况6：没有正确的统计信息，造成CBO无法做出正确的选择

- 总结：归根到底，不使用索引的通常愿意就是“不能使用索引，使用索引会返回不正确的结果”，或者“不该使用索引，如果使用了索引就会变得更慢”

总结：索引访问的不同特点

- “查询使用了索引就万事大吉了”—误解啊～～
- 索引只是访问数据的一种方式
- “通过索引定位记录”只是查询工作的一部分
- 优化器有更多的选择权利
- 总结：索引不是万灵药。充分理解要处理的数据，做出合理的判断，才能获得高效方案