Java面试题-数据库优化

一.数据库索引的原理

数据库索引，是数据库管理系统中一个排序的数据结构，以协助快速查询、更新数据库表中数据。索引的实现通常使用 **BTree** 及其变种 **B+Tree**。

二.为什么要用 B-Tree

一般来说，索引本身也很大，不可能全部存储在内存中，因此索引往往以索引文件的形式存储的磁盘上。这样的话，索引查找过程中就要产生磁盘 I/O 消耗，相对于内存存取，I/O 存取的消耗要高几个数量级，所以评价一个数据结构作为索引的优劣最重要的指标就是在查找过程中磁盘 I/O 操作次数的渐进复杂度。换句话说，索引的结构组织要尽量减少查找过程中磁盘 I/O 的存取次数。

三.说说 SQL 优化方法

1.一些常见的 SQL 实践

- 负向条件查询不能使用索引

```
select from order where status!=0 and status!=1
```

```
not in/not exists # 都不是好习惯
```

可以优化为 `in` 查询：

```
select from order where status in(2,3)
```

- 前导模糊查询不能使用索引

```
select from order where desc like '%XX'
```

而非前导模糊查询则可以：

```
select from order where desc like 'XX%'
```

- 数据区分度不大的字段不宜使用索引

```
select from user where sex=1
```

原因：性别只有男，女，每次过滤掉的数据很少，不宜使用索引。

经验上，能过滤80%数据时就可以使用索引。对于订单状态，如果状态值很少，不宜使用索引，如果状态值很多，能够过滤大量数据，则应该建立索引。

- 在属性上进行计算不能命中索引

```
select from order where YEAR(date) < = '2017'
```

即使date上建立了索引，也会全表扫描，可优化为值计算：

```
select from order where date < = CURDATE()
```

或者：

```
select from order where date < = '2017-01-01'
```

2.并非周知的 SQL 实践

- 如果业务大部分是单条查询，使用Hash索引性能更好，例如用户中心

```
select from user where uid=?
select from user where login_name=?
```

原因：`B-Tree` 索引的时间复杂度是 `O(log(n))`；`Hash` 索引的时间复杂度是 `O(1)`

- 允许为 `null` 的列，查询有潜在大坑

单列索引不存 `null` 值，复合索引不存全为 `null` 的值，如果列允许为 `null`，可能会得到“不符合预期”的结果集

```
select from user where name != 'shenjian'
```

如果 `name` 允许为 `null`，索引不存储 `null` 值，结果集中不会包含这些记录。所以，请使用 `not null` 约束以及默认值。

- 复合索引最左前缀，并不是值 SQL 语句的 `where` 顺序要和复合索引一致

用户中心建立了(login_name, passwd)的复合索引

```
select from user where login_name=? and passwd=?
select from user where passwd=? and login_name=?
```

都能够命中索引

```
select from user where login_name=?
```

也能命中索引，满足复合索引最左前缀

```
select from user where passwd=?
```

不能命中索引，不满足复合索引最左前缀

- 使用 `ENUM` 而不是字符串

`ENUM` 保存的是 `TINYINT`，别在枚举中搞一些“中国”“北京”“技术部”这样的字符串，字符串空间又大，效率又低。

3.小众但有用的 SQL 实践

- 如果明确知道只有一条结果返回， `limit 1` 能够提高效率

```
select from user where login_name=?
```

可以优化为：

```
select from user where login_name=? limit 1
```

原因：你知道只有一条结果，但数据库并不知道，明确告诉它，让它主动停止游标移动

- 把计算放到业务层而不是数据库层，除了节省数据的 CPU，还有意想不到的查询缓存优化效果

```
select from order where date < = CURDATE()
```

这不是一个好的SQL实践，应该优化为：

```
$curDate = date('Y-m-d');
$res = mysqlquery('select from order where date < = $curDate');
```

原因：释放了数据库的 CPU，多次调用，传入的SQL相同，才可以利用查询缓存

- 强制类型转换会全表扫描

```
select from user where phone=13800001234
```

你以为会命中 phone 索引么？大错特错了，这个语句究竟要怎么改？

末了，再加一条，不要使用 `select *`（潜台词，文章的 SQL 都不合格 ==），只返回需要的列，能够大大的节省数据传输量，与数据库的内存使用量。

四.MySQL 优化

- 表关联查询时务必遵循 **小表驱动大表** 原则；
- 使用查询语句 `where` 条件时，不允许出现 **函数**，否则索引会失效；
- 使用单表查询时，相同字段尽量不要用 `OR`，因为可能导致索引失效，比如：`SELECT * FROM table WHERE name = '手机' OR name = '电脑'`，可以使用 `UNION` 替代；
- `LIKE` 语句不允许使用 `%` 开头，否则索引会失效；
- 组合索引一定要遵循 **从左到右** 原则，否则索引会失效；比如：`SELECT * FROM table WHERE name = '张三' AND age = 18`，那么该组合索引必须是

`name,age` 形式；
- 索引不宜过多，根据实际情况决定，尽量不要超过 10 个；
- 每张表都必须有 **主键**，达到加快查询效率的目的；
- 分表，可根据业务字段尾数中的个位或十位或百位（以此类推）做表名达到分表的目的；
- 分库，可根据业务字段尾数中的个位或十位或百位（以此类推）做库名达到分库的目的；
- 表分区，类似于硬盘分区，可以将某个时间段的数据放在分区里，加快查询速度，可以配合 **分表 + 表分区** 结合使用；

五.神器 `EXPLAIN` 语句

`EXPLAIN` 显示了 MySQL 如何使用索引来处理 `SELECT` 语句以及连接表。可以帮助选择更好的索引和写出更优化的查询语句。

使用方法，在 `SELECT` 语句前加上 `EXPLAIN` 即可，如：

```
EXPLAIN SELECT * FROM tb_item WHERE cid IN (SELECT id FROM tb_item_ca
t)
```

- **id**： SELECT 识别符。这是 SELECT 的查询序列号
- **select_type**： SELECT类型,可以为以下任何一种
  - SIMPLE: 简单 SELECT(不使用 UNION 或子查询)
  - PRIMARY: 最外面的 SELECT
  - UNION: UNION 中的第二个或后面的 SELECT 语句
  - DEPENDENT UNION: UNION 中的第二个或后面的 SELECT 语句,取决于外面的查询
  - UNION RESULT: UNION 的结果
  - SUBQUERY: 子查询中的第一个 SELECT
  - DEPENDENT SUBQUERY: 子查询中的第一个 SELECT,取决于外面的查询
  - DERIVED: 导出表的 SELECT(FROM 子句的子查询)
- **table**： 输出的行所引用的表
- **partitions**： 表分区
- **type**： 联接类型。下面给出各种联接类型，按照 **从最佳类型到最坏类型** 进

行排序
    - system: 表仅有一行(=系统表)。这是 const 联接类型的一个特例。
    - const: 表最多有一个匹配行,它将在查询开始时被读取。因为仅有一行,在这行的列值可被优化器剩余部分认为是常数。const 表很快,因为它们只读取一次!
    - eq_ref: 对于每个来自于前面的表的行组合, 从该表中读取一行。这可能是最好的联接类型, 除了 const 类型。
    - ref: 对于每个来自于前面的表的行组合, 所有有匹配索引值的行将从这张表中读取。
    - refornull: 该联接类型如同 ref,但是添加了 MySQL 可以专门搜索包含 NULL 值的行。
    - index_merge: 该联接类型表示使用了索引合并优化方法。
    - unique_subquery: 该类型替换了下面形式的 IN 子查询的 ref: `value IN (SELECT primary_key FROM single_table WHERE some_expr)` `unique_subquery` 是一个索引查找函数, 可以完全替换子查询, 效率更高。
    - indexsubquery: 该联接类型类似于 uniquesubquery。可以替换 IN 子查询, 但只适合下列形式的子查询中的非唯一索引: `value IN (SELECT key_column FROM single_table WHERE some_expr)`
    - range: 只检索给定范围的行,使用一个索引来选择行。
    - index: 该联接类型与 ALL 相同,除了只有索引树被扫描。这通常比 ALL 快,因为索引文件通常比数据文件小。
    - ALL: 对于每个来自于先前的表的行组合, 进行完整的表扫描。
- **possible_keys**： 指出 MySQL 能使用哪个索引在该表中找到行
- **key**： 显示 MySQL 实际决定使用的键(索引)。如果没有选择索引, 键是 NULL。
- **key_len**： 显示 MySQL 决定使用的键长度。如果键是 NULL, 则长度为 NULL。
- **ref**： 显示使用哪个列或常数与 key 一起从表中选择行。
- **rows**： 显示 MySQL 认为它执行查询时必须检查的行数。多行之间的数据相乘可以估算要处理的行数。
- **filtered**： 显示了通过条件过滤出的行数的百分比估计值。
- **Extra**： 该列包含 MySQL 解决查询的详细信息
    - Distinct: MySQL 发现第 1 个匹配行后,停止为当前的行组合搜索更多的行。
    - Not exists: MySQL 能够对查询进行 LEFT JOIN 优化, 发现 1 个匹配 LEFT JOIN 标准的行后, 不再为前面的的行组合在该表内检查更多的

行。

- range checked for each record (index map: #): MySQL 没有发现好的可以使用的索引, 但发现如果来自前面的表的列值已知, 可能部分索引可以使用。
- Using filesort: MySQL 需要额外的一次传递, 以找出如何按排序顺序检索行。
- Using index: 从只使用索引树中的信息而不需要进一步搜索读取实际的行来检索表中的列信息。
- Using temporary: 为了解决查询, MySQL 需要创建一个临时表来容纳结果。
- Using where: WHERE 子句用于限制哪一个行匹配下一个表或发送到客户。
- Using sortunion(...), Using union(...), Using intersect(...): 这些函数说明如何为 indexmerge 联接类型合并索引扫描。
- Using index for group-by: 类似于访问表的 Using index 方式,Using index for group-by 表示 MySQL 发现了一个索引,可以用来查询 GROUP BY 或 DISTINCT 查询的所有列, 而不要额外搜索硬盘访问实际的表。

六.MySQL 索引使用的注意事项

- 索引不会包含有 `NULL` 的列

只要列中包含有 `NULL` 值，都将不会被包含在索引中，复合索引中只要有一列含有 `NULL` 值，那么这一列对于此符合索引就是无效的。

- 使用短索引

对串列进行索引，如果可以就应该指定一个前缀长度。例如，如果有一个 `char(255)` 的列，如果在前 `10` 个或 `20` 个字符内，多数值是唯一的，那么就不要对整个列进行索引。短索引不仅可以提高查询速度而且可以节省磁盘空间和 `I/O` 操作。

- 索引列排序

MySql 查询只使用一个索引，因此如果 `where` 子句中已经使用了索引的话，那么 `order by` 中的列是不会使用索引的。因此数据库默认排序可以符合要求的情况下不要使用排序操作，尽量不要包含多个列的排序，如果需要最好给这些列建复合索引。

- `like` 语句操作

一般情况下不鼓励使用 `like` 操作，如果非使用不可，注意正确的使用方式。`like ‘%aaa%’` 不会使用索引，而 `like ‘aaa%’` 可以使用索引。

- 不要在列上进行运算

- 不使用 `NOT IN` 、 `<>` 、 `!=` 操作，但 `<` , `<=` , `=` , `>` , `>=` , `BETWEEN` , `IN` 是可以用到索引的

- 索引要建立在经常进行select操作的字段上
这是因为，如果这些列很少用到，那么有无索引并不能明显改变查询速度。相反，由于增加了索引，反而降低了系统的维护速度和增大了空间需求。

- 索引要建立在值比较唯一的字段上

- 对于那些定义为 `text` 、 `image` 和 `bit` 数据类型的列不应该增加索引。因为这些列的数据量要么相当大，要么取值很少

- 在 `where` 和 `join` 中出现的列需要建立索引

- `where` 的查询条件里有不等号 `(where column != …)`, MySql 将无法使用索引

- 如果 `where` 字句的查询条件里使用了函数(如： `where DAY(column)=…)`, MySql 将无法使用索引

- 在 `join` 操作中(需要从多个数据表提取数据时)，MySql 只有在主键和外键的数据类型相同时才能使用索引，否则及时建立了索引也不会使用

七.limit 20000 加载很慢怎么解决?

MySQL 的性能低是因为数据库要去扫描 `N + M` 条记录，然后又要放弃之前 `N` 条记录，开销很大

解决思路：

- 前端加缓存，或者其他方式，减少落到库的查询操作，例如某些系统中数据在搜索引擎中有备份的，可以用 es 等进行搜索
- 使用延迟关联，即先通过 limit 得到需要数据的索引字段，然后再通过原表和索引字段关联获得需要数据

```
select a.* from a,(select id from table_1 where is_deleted='N' limit
```

```
100000,20) b where a.id = b.id
```

- 从业务上实现，不分如此多页，例如只能分页前 100 页，后面的不允许再查了
- 不使用 limit N,M, 而是使用 limit N，即将 offset 转化为 where 条件。

八.MySQL有哪些存储引擎?

8种，默认innodb。

- myisam:不支持事务和外键，访问速度快，对事务完整性没要求的或以select，insert为 主的都可以用该引擎;
- innodb:默认的，支持事务和外键，行级锁;更新较多，支持事务，自动灾难恢复，外键约束，支持自增列值;
- memory:
  为了得到最快的响应速度...目标数据小，或者数据是临时的，或者数据丢失不会对服务产生实质的影响，可以用:该引擎;
- merge:
  是几个相同的myisam的组合;
- archive:
  归档，只支持最基本的插入和查询，5.5之后支持索引;

九.InnoDB存储引擎 与 MyiSAM存储引擎的区别

1. InnoDB 不支持 `FULLTEXT` 类型的索引。
2. InnoDB 中不保存表的具体行数，也就是说，执行 `select count() from table` 时，InnoDB 要扫描一遍整个表来计算有多少行，但是 MyISAM 只要简单的读出保存好的行数即可。注意的是，当 `count()` 语句包含 `where` 条件时，两种表的操作是一样的。
3. 对于 `AUTO_INCREMENT` 类型的字段，InnoDB 中必须包含只有该字段的索引，但是在 MyISAM 表中，可以和其他字段一起建立联合索引。

4. `DELETE FROM table` 时，InnoDB 不会重新建立表，而是一行一行的删除。
5. `LOAD TABLE FROM MASTER` 操作对 InnoDB 是不起作用的，解决方法是首先把 InnoDB 表改成 MyISAM 表，导入数据后再改成 InnoDB 表，但是对于使用的额外的 InnoDB 特性(例如外键)的表不适用。

另外，InnoDB 表的行锁也不是绝对的，假如在执行一个 SQL 语句时 MySQL 不能确定要扫描的范围，InnoDB 表同样会锁全表，例如 `update table set num=1 where name like “%aaa%”`

十.MySQL数据库设计规范

1.基础规范

- 表存储引擎必须使用 `InnoDB`
- 表字符集默认使用 `utf8`，必要时候使用 `utf8mb4`
  - 通用，无乱码风险，汉字 3 字节，英文 1 字节
  - `utf8mb4` 是 `utf8` 的超集，有存储 4 字节例如表情符号时，使用它
- 禁止使用存储过程，视图，触发器，Event
  - 对数据库性能影响较大，互联网业务，能让站点层和服务层干的事情，不要交到数据库层
  - 调试，排错，迁移都比较困难，扩展性较差
- 禁止在数据库中存储大文件，例如照片，可以将大文件存储在对象存储系统，数据库中存储路径
- 禁止在线上环境做数据库压力测试
- 测试，开发，线上数据库环境必须隔离

2.命名规范

- 库名，表名，列名必须用小写，采用下划线分隔
  - abc，Abc，ABC 都是给自己埋坑
- 库名，表名，列名必须见名知义，长度不要超过 32 字符
  - tmp，wushan 谁 TM 知道这些库是干嘛的
- 库备份必须以 bak 为前缀，以日期为后缀
- 从库必须以 `-s` 为后缀
- 备库必须以 `-ss` 为后缀

3.表设计规范

- 单实例表个数必须控制在 `2000` 个以内
- 单表分表个数必须控制在 `1024` 个以内
- 表必须有主键，推荐使用 `UNSIGNED` 整数为主键
  - 删除无主键的表，如果是 `row` 模式的主从架构，从库会挂住
- 禁止使用外键，如果要保证完整性，应由应用程式实现
  - 外键使得表之间相互耦合，影响 `update/delete` 等 SQL 性能，有可能造成死锁，高并发情况下容易成为数据库瓶颈
- 建议将大字段，访问频度低的字段拆分到单独的表中存储，分离冷热数据（具体参考：《如何实施数据库垂直拆分》）

4.列设计规范

- 根据业务区分使用 `tinyint/int/bigint`，分别会占用 `1/4/8` 字节
- 根据业务区分使用 `char/varchar`
  - 字段长度固定，或者长度近似的业务场景，适合使用 `char`，能够减少碎片，查询性能高
  - 字段长度相差较大，或者更新较少的业务场景，适合使用 `varchar`，能够减少空间
- 根据业务区分使用 `datetime/timestamp`
  - 前者占用 5 个字节，后者占用 4 个字节，存储年使用 `YEAR`，存储日期使用 `DATE`，存储时间使用 `datetime`
- 必须把字段定义为 `NOT NULL` 并设默认值
  - NULL 的列使用索引，索引统计，值都更加复杂，MySQL 更难优化
  - NULL 需要更多的存储空间
  - NULL 只能采用 `IS NULL` 或者 `IS NOT NULL`，而在 `=/!=/in/not in` 时有大坑
- 使用 `INT UNSIGNED` 存储 `IPv4`，不要用 `char(15)`
- 使用 `varchar(20)` 存储手机号，不要使用整数
  - 牵扯到国家代号，可能出现 `+/-/()` 等字符，例如 `+86`
  - 手机号不会用来做数学运算
  - `varchar` 可以模糊查询，例如 `like‘138%’`
- 使用 `TINYINT` 来代替 `ENUM`
  - `ENUM` 增加新值要进行 `DDL` 操作

5.索引规范

- 唯一索引使用 `uniq_[字段名]` 来命名
- 非唯一索引使用 `idx_[字段名]` 来命名

- 单张表索引数量建议控制在 5 个以内
  - 互联网高并发业务，太多索引会影响写性能
  - 生成执行计划时，如果索引太多，会降低性能，并可能导致 MySQL 选择不到最优索引
  - 异常复杂的查询需求，可以选择 `ES` 等更为适合的方式存储
- 组合索引字段数不建议超过 5 个
  - 如果 5 个字段还不能极大缩小 row 范围，八成是设计有问题
- 不建议在频繁更新的字段上建立索引
- 非必要不要进行 `JOIN` 查询，如果要进行 `JOIN` 查询，被 `JOIN` 的字段必须类型相同，并建立索引
  - 踩过因为 `JOIN` 字段类型不一致，而导致全表扫描的坑么?
- 理解组合索引最左前缀原则，避免重复建设索引，如果建立了(a,b,c)，相当于建立了(a), (a,b), (a,b,c)

6.SQL 规范

- 禁止使用 `select *`，只获取必要字段
  - `select *` 会增加 `cpu/io/内存/带宽` 的消耗
  - 指定字段能有效利用索引覆盖
  - 指定字段查询，在表结构变更时，能保证对应用程序无影响
- `insert` 必须指定字段，禁止使用 `insert into T values()`
  - 指定字段插入，在表结构变更时，能保证对应用程序无影响
- 隐式类型转换会使索引失效，导致全表扫描
- 禁止在 `where` 条件列使用函数或者表达式
  - 导致不能命中索引，全表扫描
- 禁止负向查询以及 `%` 开头的模糊查询
  - 导致不能命中索引，全表扫描
- 禁止大表 `JOIN` 和子查询
- 同一个字段上的 `OR` 必须改写问 `IN`，`IN` 的值必须少于 50 个
- 应用程序必须捕获 SQL 异常
  - 方便定位线上问题

说明

本规范适用于并发量大，数据量大的典型互联网业务.

十一.分库与分表带来的分布式困境与应对之

策

1.数据迁移与扩容问题

前面介绍到水平分表策略归纳总结为随机分表和连续分表两种情况。连续分表有可能存在数据热点的问题，有些表可能会被频繁地查询从而造成较大压力，热数据的表就成为了整个库的瓶颈，而有些表可能存的是历史数据，很少需要被查询到。连续分表的另外一个好处在于比较容易，不需要考虑迁移旧的数据，只需要添加分表就可以自动扩容。随机分表的数据相对比较均匀，不容易出现热点和并发访问的瓶颈。但是，分表扩展需要迁移旧的数据。

针对于水平分表的设计至关重要，需要评估中短期内业务的增长速度，对当前的数据量进行容量规划，综合成本因素，推算出大概需要多少分片。对于数据迁移的问题，一般做法是通过程序先读出数据，然后按照指定的分表策略再将数据写入到各个分表中。

2.表关联问题

在单库单表的情况下，联合查询是非常容易的。但是，随着分库与分表的演变，联合查询就遇到跨库关联和跨表关系问题。在设计之初就应该尽量避免联合查询，可以通过程序中进行拼装，或者通过反范式化设计进行规避。

3.分页与排序问题

一般情况下，列表分页时需要按照指定字段进行排序。在单库单表的情况下，分页和排序也是非常容易的。但是，随着分库与分表的演变，也会遇到跨库排序和跨表排序问题。为了最终结果的准确性，需要在不同的分表中将数据进行排序并返回，并将不同分表返回的结果集进行汇总和再次排序，最后再返回给用户。

4.分布式事务问题

随着分库与分表的演变，一定会遇到分布式事务问题，那么如何保证数据的一致性就成为一个必须面对的问题。目前，分布式事务并没有很好的解决方案，难以满足数据强一致性，一般情况下，使存储数据尽可能达到用户一致，保证系统经过一段较短的时间的自我恢复和修正，数据最终达到一致。

5.分布式全局唯一ID

在单库单表的情况下，直接使用数据库自增特性来生成主键ID，这样确实比较简

单。在分库分表的环境中，数据分布在不同的分表上，不能再借助数据库自增长特性。需要使用全局唯一 ID，例如 UUID、GUID等。关于如何选择合适的全局唯一 ID，我会在后面的章节中进行介绍。

十二.MySQL 遇到的死锁问题

产生死锁的四个必要条件：

- 互斥条件：一个资源每次只能被一个进程使用。

- 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。

- 不可剥夺条件：进程已获得的资源，在未使用完之前，不能强行剥夺。

- 循环等待条件：若干进程之间形成一种头尾相接的循环等待资源关系。

这四个条件是死锁的必要条件，只要系统发生死锁，这些条件必然成立，而只要上述条件之一不满足，就不会发生死锁。

下列方法有助于最大限度地降低死锁：

- 按同一顺序访问对象。
- 避免事务中的用户交互。
- 保持事务简短并在一个批处理中。
- 使用低隔离级别。
- 使用绑定连接。

十三.JDBC 流程

- 向 DriverManager 类注册驱动数据库驱动程序
- 调用 DriverManager.getConnection 方法， 通过 JDBC URL，用户名，密码取得数据库连接的 Connection 对象。
- 获取 Connection 后， 便可以通过 createStatement 创建 Statement 用以执行 SQL 语句。

- 有时候会得到查询结果，比如 select，得到查询结果，查询（SELECT）的结果存放于结果集（ResultSet）中。
- 关闭数据库语句，关闭数据库连接。