

2019
版

治愈系 Java 工程 师面试指导课程

[Part3 · 数据库和事务]

[本教程梳理了 mysql 和事务部分的复习脉络，并且涵盖了 mysql 数据库的高频面试题；本章还介绍了不分事务的部分相关内容，更多关于分布式事务的知识点可参考分布式事务专题，建议大家将目录结构打开，对照目录结构做一个复习的、提纲准备面试。]

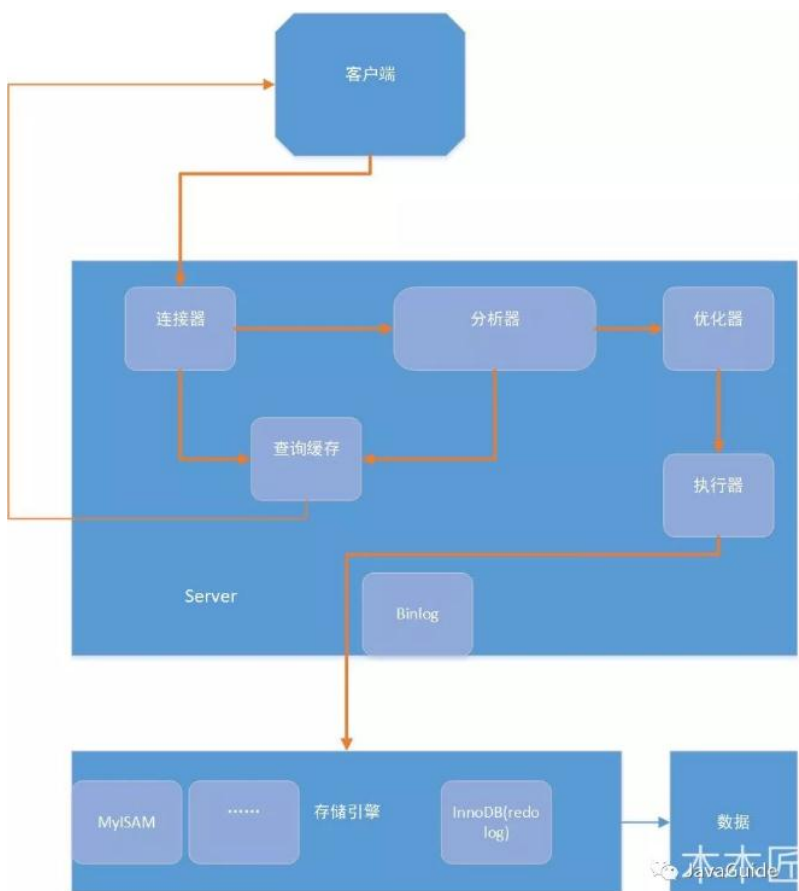


1 存储引擎

数据库存储引擎是数据库底层软件组织, 数据库管理系统 (DBMS) 使用数据引擎进行创建、查询、更新和删除数据。不同的存储引擎提供不同的存储机制、索引技巧、锁定水平等功能, 使用不同的存储引擎, 还可以获得特定的功能。现在许多不同的数据库管理系统都支持多种不同的数据引擎。存储引擎主要有: 1. MyIsam, 2. InnoDB, 3. Memory, 4. Archive, 5. Federated。

Mysql 在 V5.5.5 之前默认存储引擎是 MyISAM; 在此之后默认存储引擎是 InnoDB

1.1 Mysql 的架构图



- **Server 层:** 主要包括连接器、查询缓存、分析器、优化器、执行器等, 所有跨存储引擎的功能都在这一层实现, 比如存储过程、触发器、视图, 函数等, 还有一个通用的日志模块 binlog 日志模块。
- **存储引擎:** 主要负责数据的存储和读取, 采用可以替换的插件式架构, 支持 InnoDB、MyISAM、Memory 等多个存储引擎, 其中 InnoDB 引擎有自有的日志模块 redo log 模块。现在最常用的存储引擎是 InnoDB, 它从 MySQL 5.5.5 版本开始就被当做默认存储引擎了。

1.2 选择合适的存储引擎

- 当创建表时, 应根据表的应用场景选择适合的存储引擎。
- MyISAM 表最适合于大量的数据读而少量数据更新的混合操作。

MyISAM 表的另一种适用情形是使用压缩的只读表。

- 如果查询中包含较多的数据更新操作, 应使用 InnoDB。其行级锁机制和多版本的支持为数据读取和更新的混合操作提供了良好的并发机制。
- 可使用 MEMORY 存储引擎来存储非永久需要的数据, 或者是能够从基于磁盘的表中重新生成的数据。

1.3 InnoDB 和 MyISAM 的对比

1.3.1 两者的区别

1. count 运算上的区别: 因为 MyISAM 缓存有表 meta-data (行数等), 因此在做 COUNT(*) 时对于一个结构很好的查询是不需要消耗多少资源的。而对于 InnoDB 来说, 则没有这种缓存
2. 是否支持事务和崩溃后的安全恢复: MyISAM 强调的是性能, 每次查询具有原子性, 其执行速度比 InnoDB 类型更快, 但是不提供事务支持。但是 InnoDB 提供事务支持事务, 外部键等高级数据库功能。具有事务 (commit)、回滚 (rollback) 和崩溃修复能力 (crash recovery capabilities) 的事务安全 (transaction-safe (ACID compliant)) 型表。
3. 是否支持外键: MyISAM 不支持, 而 InnoDB 支持。
4. 是否支持行级锁: MyISAM 只有表级锁 (table-level locking), 而 InnoDB 支持行级锁 (row-level locking) 和表级锁, 默认为行级锁。

1.3.2 关于两者的总结

MyISAM 更适合读密集的表, 而 InnoDB 更适合写密集的的表。在数据库做主从分离的情况下, 经常选择 MyISAM 作为主库的存储引擎。

一般来说, 如果需要事务支持, 并且有较高的并发读取频率 (MyISAM 的表锁的粒度太大, 所以当该表写并发量较高时, 要等待的查询就会很多了), InnoDB 是不错的选择。如果你的数据量很大 (MyISAM 支持压缩特性可以减少磁盘的空间占用), 而且不需要支持事务时, MyISAM 是最好的选择。

2 事务的概念和 ACID 特性

2.1 概念

事务 (TRANSACTION) 表示一个完整的不可分割的业务, 批量的 DML 语句同时成功或者同时失败。原子性不可再分最基本单元, 可以看做一个完整的事件, 通常一个事务可以对应一个完整的业务流程。

可以保证多个操作原子性, 要么全成功, 要么全失败。对于数据库来说事务保证批量的 DML 要么全成功, 要么全失败。

事务中存在一些概念:

- a) 事务 (Transaction): 保证批量的 DML 同时成功或同时失败。一批操作 (一组 DML)
- b) 开启事务 (Start Transaction)
- c) 回滚事务 (rollback)
- d) 提交事务 (commit)

e) SET AUTOCOMMIT: 禁用或启用事务的自动提交模式

当执行 DML 语句是其实就是开启一个事务

关于事务的回滚需要注意: 只能回滚 insert、delete 和 update 语句, 不能回滚 select (回滚 select 没有任何意义), 对于 create、drop、alter 这些无法回滚.

事务只对 DML 有效果。

注意: rollback, 或者 commit 后事务就结束了。

2.2 ACID 特性

事务具有四个特征 ACID

2.2.1 原子性 (Atomicity)

- 整个事务中的所有操作, 必须作为一个单元全部完成 (或全部取消)。

undo log 名为回滚日志, 是实现原子性的关键, 当事务回滚时能够撤销所有已经成功执行的 sql 语句, 他需要记录你要回滚的相应日志信息。

2.2.2 一致性 (Consistency)

- 在事务开始之前与结束之后, 数据库都保持一致状态。

从数据库层面, 数据库通过原子性、隔离性、持久性来保证一致性。也就是说 ACID 四大特性之中, C(一致性)是目的, A(原子性)、I(隔离性)、D(持久性)是手段, 是为了保证一致性, 数据库提供的手段。数据库必须要实现 AID 三大特性, 才有可能实现一致性。例如, 原子性无法保证, 显然一致性也无法保证。

但是, 如果你在事务里故意写出违反约束的代码, 一致性还是无法保证的。例如, 你在转账的例子中, 你的代码里故意不给 B 账户加钱, 那一一致性还是无法保证。因此, 还必须从应用层角度考虑。从应用层面, 通过代码判断数据库数据是否有效, 然后决定回滚还是提交数据!

2.2.3 隔离性(Isolation)

- 一个事务不会影响其他事务的运行。

事务的隔离性是利用的是锁和 MVCC(Multi Version Concurrency Control)机制。

2.2.4 持久性(Durability)

- 在事务完成以后, 该事务对数据库所作的更改将持久地保存在数据库之中, 并不会被回滚。

事务的持久性是利用 Innodb 的 redo log。

3 事务隔离级别详解

事务的隔离级别决定了事务之间可见的级别。

3.1 四个隔离级别

- InnoDB 实现了四个隔离级别, 用以控制事务所做的修改, 并将修改通告至其它并发的任务:

- 读未提交 (READ UNCOMMITTED)

允许一个事务可以看到其他事务未提交的修改。

- 读已提交 (READ COMMITTED)

允许一个事务只能看到其他事务已经提交的修改, 未提交的修改是不可见的。

- 可重复读 (REPEATABLE READ)

确保如果在一个事务中执行两次相同的 SELECT 语句, 都能得到相同的结果, 不管其他事务是否提交这些修改。
(银行总账)

- 串行化 (SERIALIZABLE) 【序列化】

该隔离级别为 InnoDB 的缺省设置。

将一个事务与其他事务完全地隔离。

3.2 隔离级别与一致性问题

当多个客户端并发地访问同一个表时, 可能出现下面的一致性问题:

- 更新丢失: 两个并行操作, 后进行操作覆盖掉了先进行操作的操作结果, 被称作更新丢失。
- 脏读 (Dirty Read): 一个事务在提交之前, 在事务过程中修改的数据, 被其他事务读取到了。
- 不可重复读 (Non-repeatable Read): 一个事务在提交之前, 在事务过程中读取以前的数据却发现数据发生了改变。
- 幻读 (Phantom Read): 一个事务按照相同的条件重新读取以前检索过的数据时, 却发现了其他事务插入的新数据。

通用的解决思路是更新丢失通过应用程序完全避免。而其他的问题点则通过调整数据库事务隔离级别来解决。**事务的隔离机制的实现手段之一就是利用锁。**

隔离级别与一致性的关系

隔离级别	脏读取	不可重复读	幻像读
读未提交	可能	可能	可能
读已提交	不可能	可能	可能
可重复读	不可能	不可能	对 InnoDB 不可能
串行化	不可能	不可能	不可能

3.3 默认隔离级别的理解

与 SQL 标准不同的地方在于 InnoDB 存储引擎在 REPEATABLE-READ（可重读）事务隔离级别下使用的是 Next-Key Lock 锁算法，因此可以避免幻读的产生，这与其他数据库系统(如 SQL Server)是不同的。所以说 InnoDB 存储引擎的默认支持的隔离级别是 REPEATABLE-READ（可重读）已经可以完全保证事务的隔离性要求，即达到了 SQL 标准的 SERIALIZABLE(可串行化)隔离级别。

因为隔离级别越低，事务请求的锁越少，所以大部分数据库系统的隔离级别都是 READ-COMMITTED(读取提交内容);，但是你要知道的是 InnoDB 存储引擎默认使用 REPEATABLE-READ（可重读）并不会有任何性能损失。InnoDB 存储引擎在 分布式事务 的情况下一般会用到 SERIALIZABLE(可串行化)隔离级别。

4 分布式事务

4.1 什么是分布式事务

传统项目中用一个 mysql 数据库，mysql 数据库当中 innoDB 引擎是支持 ACID 事务特性的，能够满足我们的事务要求，但现在分布式项目中，我们会有多个微服务处理不同业务这些微服务都是独立的进程，数据库也都是相互独立，但是事务可能要横跨这些微服务，比如订单功能，如果我支付订单成功在订单微服务中我需要改变订单状态，在库存微服务中需要减对应的库存，在积分微服务中得加对应积分，这是一个整体的事务，这种事务就是分布式事务

4.2 CAP 定理和 Base 理论

CAP 定理是由加州大学伯克利分校 Eric Brewer 教授提出来的，他指出 WEB 服务无法同时满足以下三个属性

- **一致性(Consistency):** 客户端知道一系列的操作都会同时发生(生效)
- **可用性(Availability):** 每个操作都必须以可预期的响应结束
- **分区容错性(Partition tolerance):** 即使出现单个组件无法可用,操作依然可以完成

在分布式系统中,我们往往追求的是可用性,它的重要程度比一致性要高,那么如何实现高可用性呢?那就是另外一个理论 **BASE 理论**,它是用来对 CAP 定理进行进一步扩充的。BASE 理论指的是

- Basically Available (基本可用): 分布式系统在出现故障时,允许损失部分可用功能,保证核心功能可用。
- Soft state (软状态): 允许系统中存在中间状态,这个状态不影响系统可用性,这里指的是 CAP 中的不一致。
- Eventually consistent (最终一致性): 指经过一段时间后,所有节点数据都将会达到一致

BASE 理论是对 CAP 中的一致性和可用性进行一个权衡的结果,理论的核心思想就是:我们无法做到强一致,但每个应用都可以根据自身的业务特点,采用适当的方式来使系统达到最终一致性 (Eventual consistency)。

4.3 2PC 二阶段分步提交

其核心思想是将分布式事务拆分成本地事务进行处理

➤ 白话

各个服务在处理数据过程中,会进行预备提交操作,预备操作可以反映出各个节点是否都能够提交,预备操作如果成功的话,那么事务协调器会执行每个数据库的 commit 操作。如果在预提交过程当中有一个节点不能够提交,事务协调器会执行每个数据库的 roll back 操作

➤ 官腔

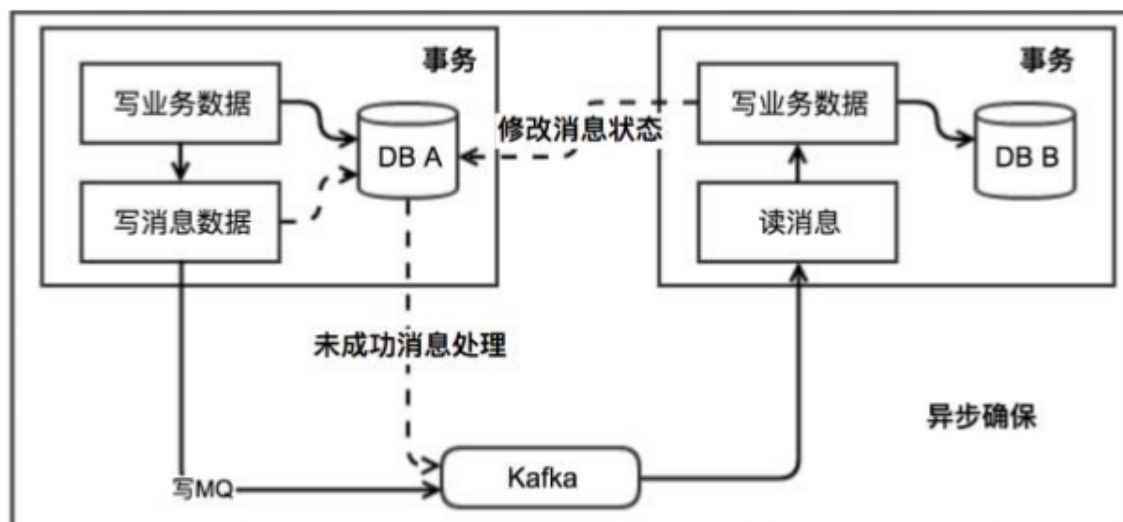
事务管理器要求每个涉及到事务的数据库预提交(precommit)此操作,并反映是否可以提交。事务协调器要求每个数据库提交数据,或者回滚数据。

➤ 实现的思路

消息生产方,需要额外建一个消息表,并记录消息发送状态。消息表和业务数据要在一个事务里提交,也就是说他们要在一个数据库里面。然后消息会经过 MQ 发送到消息的消费方。如果消息发送失败,会进行重试发送。

消息消费方,需要处理这个消息,并完成自己的业务逻辑。此时如果本地事务处理成功,表明已经处理成功了,如果处理失败,那么就会重试执行。如果是业务上面的失败,可以给生产方发送一个业务补偿消息,通知生产方进行回滚等操作。

生产方和消费方定时扫描本地消息表,把还没处理完成的消息或者失败的消息再发送一遍。如果有靠谱的自动对账补账逻辑,这种方案还是非常实用的。这种方案遵循 BASE 理论,采用的是最终一致性。



4.4 TCC (Try Confirm Cancel)

- **Try 阶段:** 尝试执行, 完成所有业务检查 (一致性), 预留必需业务资源 (准隔离性)。
- **Confirm 阶段:** 确认真正执行业务, 不作任何业务检查, 只使用 try 阶段预留的业务资源, Confirm 操作满足幂等性。要求具备幂等设计, Confirm 失败后需要进行重试。默认只要 Try 成功, Confirm 一定成功。
- **Cancel 阶段:** 取消执行, 释放 Try 阶段预留的业务资源, Cancel 操作满足幂等性。Cancel 阶段的异常和 Confirm 阶段异常处理方案基本上一致。
- **举个简单的例子:** 如果你用 100 元买了一瓶水, Try 阶段: 你需要向你的钱包检查是否够 100 元并锁住这 100 元, 水也是一样的。如果有一个失败, 则进行 Cancel(释放这 100 元和这一瓶水), 如果 Cancel 失败不论什么失败都进行重试 Cancel, 所以需要保持幂等。如果都成功, 则进行 Confirm, 确认这 100 元被扣, 和这一瓶水被卖, 如果 Confirm 失败无论什么失败则重试(会依靠活动日志进行重试)。

4.5 项目中的使用

使用开源框架 TX-LCN 分布式事务框架 5.0 版本对代码的侵入比较少, 而且提供了完整的 SpringCloud 和 dubbo 注解解决方案, 项目组引入该框架的成本非常低, 并且有专门的公司对这个开源项目进行维护。

该框架主要有两大核心部分 TM 事务的协调管理器需要独立运行 (在源码中有该工程, 并且整合了 docker 只需要改下配置文件中 mysql、redis 的地址即可运行)

- 所有参与分布式事务的 微服务项目 需要引入客户端依赖

```
<dependency>
  <groupId>com.codingapi.txlcn</groupId>
  <artifactId>txlcn-txmsg-netty</artifactId>
  <version>5.0.2.RELEASE</version>
</dependency>
```

- 在启动类当中, 使用@EnableDistributedTransaction 注解开启分布式事务
- 在配置文件中配置好 Tx-manager 的连接地址
- 在需要进行 事务管理的 service 方法中

如果要使用分布式事务加上 @LcnTransation; 如果要使用 TCC 的模式需要加上@TCCTransation 注解并且在 该注解指定 confirm 对应的方法以及 cancel 对应的方法

5 数据库锁和并发策略

<https://www.jianshu.com/p/69ed3b9d858c>

事务的隔离机制的实现手段之一就是利用锁

5.1 数据库锁的分类

- MyIsam 实现了表锁。表锁可以针对数据库表加锁, 在锁的灵活性上不如行锁。表锁分为两种锁: 读锁与写锁。
- InnoDB 存储引擎实现了行锁与表锁 (意向锁)。行锁可以以行为单位对数据集进行锁定。行锁也分为两种锁: 共享锁与排他锁。InnoDB 对于 Update、Delete、insert 语句会自动给涉及的数据集隐式的加上排他锁。对于 select 语句 InnoDB 不会加任何锁

- 共享锁: 允许一个事务读取一行, 阻止其他事务获得相同数据集的排他锁。但允许其他事务获取共享锁。
显示加锁 `select * from table where ... lock in share mode`
- 排他锁: 允许获得排他锁的事务更新数据, 阻止其他事务取得相同数据集的共享与排他锁。但是可以对获取了排他锁的数据集进行单纯的查询访问。
显示加锁 `sql select * from table where ... for update`
- InnoDB 的行锁的实现方式是基于索引项的。这意味着即使你尝试获取不同行的排他锁, 若使用了相同的索引键, 也可能造成锁冲突。

5.1.1 行级锁

行级锁是一种排他锁, 防止其他事务修改此行; 行锁的特点是开销大, 加锁慢; 会出现死锁; 锁定粒度最小, 发生锁冲突的概率最低, 并发度也最高

在使用以下语句时, Oracle 会自动应用行级锁:

1. INSERT、UPDATE、DELETE、SELECT ... FOR UPDATE [OF columns] [WAIT n] [NOWAIT];
2. SELECT ... FOR UPDATE 语句允许用户一次锁定多条记录进行更新
3. 使用 COMMIT 或 ROLLBACK 语句释放锁。

5.1.2 表级锁

表锁的特点是开销小, 加锁快; 不会出现死锁; 锁定粒度大, 发生锁冲突的概率最高, 并发度最低

表示对当前操作的整张表加锁, 它实现简单, 资源消耗较少, 被大部分 MySQL 引擎支持。最常使用的 MYISAM 与 INNODB 都支持表级锁定。

5.1.3 页级锁

页级锁是 MySQL 中锁定粒度介于行级锁和表级锁中间的一种锁。表级锁速度快, 但冲突多, 行级冲突少, 但速度慢。所以取了折衷的页级, 一次锁定相邻的一组记录。BDB 支持页级锁

5.2 数据库的并发策略

5.2.1 乐观锁

乐观锁认为一个用户读数据的时候, 别人不会去写自己所读的数据; 悲观锁就刚好相反, 觉得自己读数据库的时候, 别人可能刚好在写自己刚读的数据, 其实就是持一种比较保守的态度; 时间戳就是不加锁, 通过时间戳来控制并发出现的问题。

解决方案为: 时间戳就是在数据库表中单独加一列时间戳, 比如 “TimeStamp”, 每次读出来的时候, 把该字段也读出来, 当写回去的时候, 把该字段加 1, 提交之前, 跟数据库的该字段比较一次, 如果比数据库的值大的话, 就允许保存, 否则不允许保存, 这种处理方法虽然不使用数据库系统提供的锁机制, 但是这种方法可以大大提高数据库处理的并发量,

```
Select max(nub),version from biao
```

```
Update biao set nub=nub+1,version=vversion+1 where id=id and version =version
```

5.2.2 悲观锁

悲观锁就是在读取数据的时候, 为了不让别人修改自己读取的数据, 就会先对自己读取的数据加锁, 只有自己把数据读完了, 才允许别人修改那部分数据, 或者反过来说, 就是自己修改某条数据的时候, 不允许别人读取该数据, 只有等自己的整个事务提交了, 才释放自己加上的锁, 才允许其他用户访问那部分数据。

悲观锁所说的加“锁”, 其实分为几种锁, 分别是: 排它锁(写锁)和共享锁(读锁)

解决方案为 sql 语句后边加上 for update 例子: `select id,nam from biao for update`

5.2.3 锁表问题

- ◆ 锁表发生在 insert、update、delete 中
- ◆ 锁表的原理是数据库使用独占式锁机制, 当执行 insert、update 和 delete 的语句时, 对表进行锁住, 直到发生 commit 或者回滚或者退出数据库用户
- ◆ 锁表的原因: 当多个连接(数据库连接)同时对一个表的数据进行更新操作, 那么速度将会越来越慢, 持续一段时间后将出现数据表被锁的现象, 从而影响到其它的查询及更新。
 - A 程序执行了对 tableA 的 insert, 并还未 commit 时, B 程序也对 tableA 进行 insert 则此时会发生资源正忙的异常就是锁表。
 - 锁表常发生于并发而不是并行(并行时, 一个线程操作数据库时, 另一个线程是不能操作数据库的, cpu 和 i/o 分配原则)
- ◆ 减少锁表的概率
 - 减少 insert、update、delete 语句执行到 commit 之间的时间。具体点批量执行改为单个执行、优化 sql 自身的非执行速度
 - 如果异常对事物进行回滚

https://blog.csdn.net/fragrant_no1/article/details/79727263

6 存储过程和触发器

6.1 存储过程(特定功能的 SQL 语句集)

一组为了完成特定功能的 SQL 语句集, 存储在数据库中, 经过第一次编译后再次调用不需要再次编译, 用户通过指定存储过程的名字并给出参数(如果该存储过程带有参数)来执行它。存储过程是数据库中的一个重要对象。存储过程优化思路:

1. 尽量利用一些 sql 语句来替代一些小循环, 例如聚合函数, 求平均函数等。
2. 中间结果存放于临时表, 加索引。
3. 少使用游标。sql 是个集合语言, 对于集合运算具有较高性能。而 cursors 是过程运算。比如 对一个 100 万行的数据进行查询。游标需要读表 100 万次, 而不使用游标则只需要少量几次读取。
4. 事务越短越好。sqlserver 支持并发操作。如果事务过多过长, 或者隔离级别过高, 都会造成并发操作的阻塞, 死锁。导致查询极慢, cpu 占用率极高。

5. 使用 try-catch 处理错误异常。
6. 查找语句尽量不要放在循环内。

6.2 触发器(一段能自动执行的程序)

触发器是一段能自动执行的程序, 是一种特殊的存储过程, 触发器和普通的存储过程的区别是: 触发器是当对某一个表进行操作时触发。诸如: update、insert、delete 这些操作的时候, 系统会自动调用执行该表上对应的触发器。SQL Server 2005 中触发器可以分为两类: DML 触发器和 DDL 触发器, 其中 DDL 触发器它们会影响多种数据定义语言语句而激发, 这些语句有 create、alter、drop 语句。

7 数据库结构优化

一个好的数据库设计方案对于数据库的性能往往会起到事半功倍的效果。需要考虑数据冗余、查询和更新的速度、字段的数据类型是否合理等多方面的内容。

7.1 三大范式

第一范式: 有主键, 具有原子性, 字段不可分割

第二范式: 完全依赖, 表中非主键列不存在对主键的部分依赖。要求每个表只描述一 件事情。

第三范式: 没有传递依赖, 表中的列不存在对非主键列的传递依赖。

数据库设计尽量遵循三范式, 但是还是根据实际情况进行取舍, 有时可能会拿冗余换速度, 最终用目的要满足客户需求。

7.2 数据库命令规范

- 所有数据库对象名称必须使用小写字母并用下划线分割
- 所有数据库对象名称禁止使用 MySQL 保留关键字(如果表名中包含关键字查询时, 需要将其用单引号括起来)
- 数据库对象的命名要能做到见名识意, 并且最后不要超过 32 个字符
- 临时库表必须以 tmp 为前缀并以日期为后缀, 备份表必须以 bak 为前缀并以日期 (时间戳) 为后缀
- 所有存储相同数据的列名和列类型必须一致(一般作为关联列, 如果查询时关联列类型不一致会自动进行数据类型隐式转换, 会造成列上的索引失效, 导致查询效率降低)

7.3 建表时优化

7.3.1 列选择原则

1: 字段类型优先级 整型 > date,time > char,varchar > blob

2: 够用就行,不要慷慨 (如 smallint, varchar(N))

原因: 列的字段越大, 建立索引时所需要的空间也就越大, 这样一页中所能存储的索引节点的数量也就越少也越少, 在遍历时所需要的 IO 次数也就越多, 索引的性能也就越差。

3: 尽量避免用 NULL()

原因: 索引 NULL 列需要额外的空间来保存, 所以要占用更多的空间进行比较和计算时要对 NULL 值做特别的处理

4: 对于非负型的数据 (如自增 ID, 整型 IP) 来说, 要优先使用无符号整型来存储

原因: 无符号相对于有符号可以多出一倍的存储空间

SIGNED INT -2147483648~2147483647 UNSIGNED INT 0~4294967295

VARCHAR(N) 中的 N 代表的是字符数, 而不是字节数, 使用 UTF8 存储 255 个汉字 Varchar(255)=765 个字节。过大的长度会消耗更多的内存。

5: 避免使用 TEXT, BLOB 数据类型, 最常见的 TEXT 类型可以存储 64k 的数据

6: 同财务相关的金额类数据必须使用 decimal 类型

非精准浮点: float, double

精准浮点: decimal Decimal 类型为精准浮点数, 在计算时不会丢失精度

占用空间由定义的宽度决定, 每 4 个字节可以存储 9 位数字, 并且小数点要占用一个字节可用于存储比 bigint 更大的整型数据

7.3.2 主键的选择:

主键用来区分, 查找, 和关联数据, 非常重要.

1. 在 myisam 中, 字符串索引会被压缩, 用字符串做主键性能不如整型

2. 用递增的值, 不要用离散的值, 离散值会导致文件在磁盘的位置有间隔, 浪费空间且不易连续读取

3. UUID, 也是逐步增长的, 可以去掉 "-", 转换为整数

7.3.3 反范式设计表

反范式的目的--减少表的关联查询

常用办法:

冗余字段和冗余表

冗余字段: 表中某字段存储另一表的统计信息

冗余表: 表中统计或汇总其他表的信息, 又称汇总表

本质

1: 空间换时间

2: 大任务分成小任务, 分散执行

7.3.4 将字段很多的表分解成多个表

对于字段较多的表, 如果有些字段的使用频率很低, 可以将这些字段分离出来形成新表。

因为当一个表的数据量很大时, 会由于使用频率低的字段的存在而变慢。

举例: 商品表和商品描述表就是把描述字段拆分成一个单独的表

7.3.5 增加中间表

Note: 不是多对多的中间表

对于需要经常联合查询的表, 可以建立中间表以提高查询效率。

通过建立中间表, 将需要通过联合查询的数据插入到中间表中, 然后将原来的联合查询改为对中间表的查询。

举例:

7.3.6 增加冗余字段

设计数据表时应尽量遵循范式理论的规约, 尽可能的减少冗余字段, 让数据库设计看起来精致、优雅。但是, 合理的加入冗余字段可以提高查询速度。

表的规范化程度越高, 表和表之间的关系越多, 需要连接查询的情况也就越多, 性能也就越差。

注意:

冗余字段的值在一个表中修改了, 就要想办法在其他表中更新, 否则就会导致数据不一致的问题。

举例: 商品表里增加类目名称

8 查询语句优化

1: sql 语句的时间花在哪儿?

答: 等待时间, 执行时间.

这两个时间并非孤立的, 如果单条语句执行的快了, 对其他语句的锁定的也就少了
所以, 我们来分析如何降低执行时间.

2: sql 语句的执行时间, 又花在哪儿了?

答:

a: 查 ----> 沿着索引查, 甚至全表扫描

b: 取 ----> 查到行后, 把数据取出来(sending data)

3: sql 语句的优化思路?

答: 不查, 通过业务逻辑来计算,

比如论坛的注册会员数, 我们可以根据前 3 个月统计的每天注册数, 用程序来估算.

少查, 尽量精准数据,少取行. 我们观察新闻网站,评论内容等,一般一次性取列表 10-30 条左右.

必须要查,尽量走在索引上查询行.

取时, 取尽量少的列.

比如 `select * from tableA,` 就取出所有列, 不建议.

比如 `select * from tableA,tableB,` 取出 A,B 表的所有列.

治愈系Java工程师面试指导课程—samuel

4: 如果定量分析查的多少行,和是否沿着索引查?

答: 用 explain 来分析

8.1 数据库优化概述

2.1.1 什么是优化?

- 合理安排资源、调整系统参数使 MySQL 运行更快、更节省资源。
- 优化是多方面的, 包括查询、更新、服务器等。
- **原则: 减少系统瓶颈, 减少资源占用, 增加系统的反应速度。**

2.1.2 数据库性能参数

- 使用 SHOW STATUS 语句查看 MySQL 数据库的性能参数
 - SHOW STATUS LIKE 'value'
- 常用的参数:
 - Slow_queries 慢查询次数
 - Com_(CRUD) 操作的次数
 - Uptime 上线时间

8.2 常见的查询优化

SELECT 语句务必指明字段名称 (避免直接使用 select *)

SQL 语句要避免造成索引失效的写法

SQL 语句中 IN 包含的值不应过多

当只需要一条数据的时候, 使用 limit 1

如果排序字段没有用到索引, 就尽量少排序

如果限制条件中其他字段没有索引, 尽量少用 or

尽量用 union all 代替 union

避免在 where 子句中对字段进行 null 值判断

不建议使用 % 前缀模糊查询

避免在 where 子句中对字段进行表达式操作

Join 优化 能用 innerjoin 就不用 left join right join, 如必须使用 一定要以小表为驱动

8.2.1 缓存优化

为了提高查询速度, 我们可以通过不同的方式去缓存我们的结果从而提高响应效率。当我们的数据库打开了 Query Cache (简称 QC) 功能后, 数据库在执行 SELECT 语句时, 会将其结果放到 QC 中, 当下一次处理同样的 SELECT 请求时, 数据库就会从 QC 取得结果, 而不需要去数据表中查询。如果缓存命中率非常高的话, 有测试表明在极端情况下可以提高效率 238%。

8.2.2 读写分离

如果数据库的使用场景读的操作比较多的时候, 为了避免写的操作所造成的性能影响 可以采用读写分离的架构, 读写分离, 解决的是, 数据库的写入, 影响了查询的效率。读写分离的基本原理是让主数据库处理事务性增、改、删操作 (INSERT、UPDATE、DELETE), 而从数据库处理 SELECT 查询操作。数据库复制被用来把事务性操作导致的变更同步到集群中的从数据库。

8.2.3 mysql 的分库分表

- ◆ 数据量越来越大时, 单体数据库无法满足要求, 可以考虑分库分表
- ◆ 两种拆分方案:
 - 垂直拆分: (分库) 业务表太多? 将业务细化 不同的小业务专门用一个库来维护
 - 水平拆分: (分表) 单个表存的数据太多, 装不下了? 将该表查分成多个
- ◆ 分库分表常用工具: MyCat、Sharding-JDBC

8.3 EXPLAIN

在 MySQL 中可以使用 EXPLAIN 查看 SQL 执行计划, 用法: `EXPLAIN SELECT * FROM tb_item`

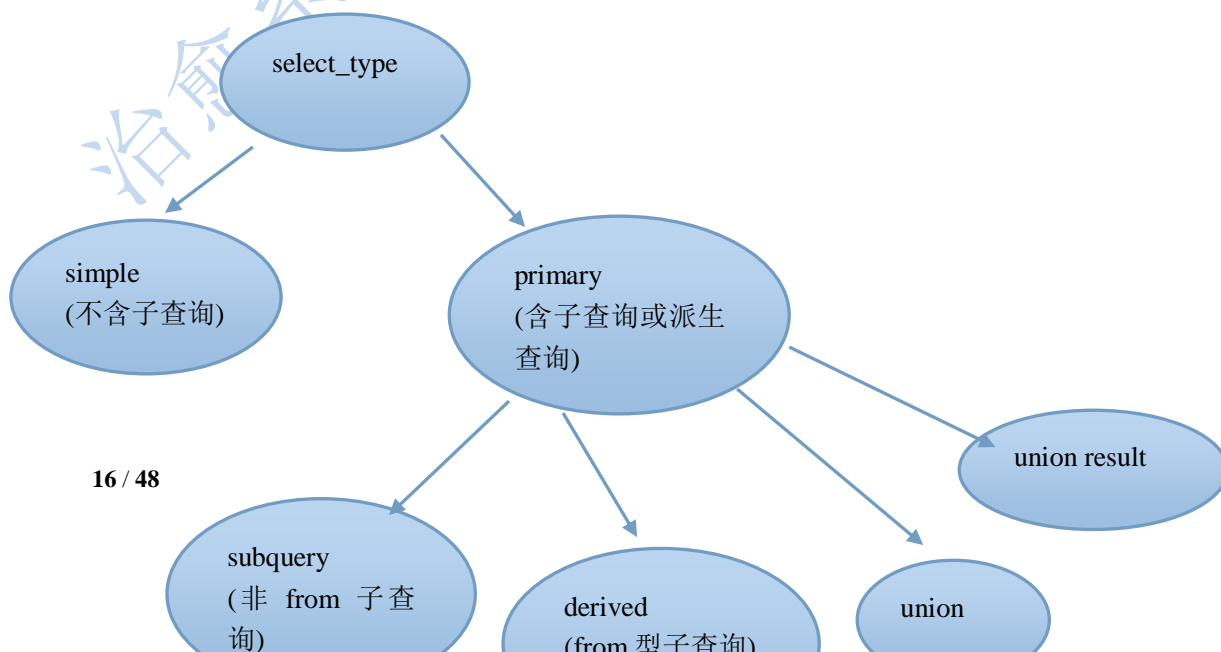
2.1.1 id

SELECT 识别符。这是 SELECT 查询序列号。这个不重要。

代表 select 语句的编号, 如果是连接查询, 表之间是平等关系, select 编号都是 1, 从 1 开始。如果某 select 中有子查询, 则编号递增。

2.1.2 select_type

表示 SELECT 语句的类型。有以下几种值:



1、SIMPLE

表示简单查询, 其中不包含连接查询和子查询。

2、PRIMARY

表示主查询, 或者是最外面的查询语句。

```
1 EXPLAIN SELECT * FROM (SELECT id FROM tb_item WHERE price = 5288) AS tmp
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	<derived2>	ALL	(Null)	(Null)	(Null)	(Null)	38	(Null)
2	DERIVED	tb_item	ALL	(Null)	(Null)	(Null)	(Null)	38	Using whe

3、UNION

表示连接查询的第 2 个或后面的查询语句。

```
1 EXPLAIN SELECT * FROM tb_order LIMIT 1 UNION SELECT * FROM tb_order LIMIT 2
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	tb_order	ALL	(Null)	(Null)	(Null)	(Null)	2	(Null)
2	UNION	tb_order	ALL	(Null)	(Null)	(Null)	(Null)	2	(Null)
(Null)	UNION RESULT	<union1,2>	ALL	(Null)	(Null)	(Null)	(Null)	(Null)	Using terr

4、DEPENDENT UNION

UNION 中的第二个或后面的 SELECT 语句, 取决于外面的查询。

5、UNION RESULT

连接查询的结果。

6、SUBQUERY

子查询中的第 1 个 SELECT 语句。

```
1 EXPLAIN SELECT * FROM tb_order WHERE user_id = (SELECT id FROM tb_user WHERE name = '张三')
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	tb_order	ref	FK_orders_1	FK_orde 8	const	1	1	Using whe
2	SUBQUERY	tb_user	ALL	(Null)	(Null)	(Null)	(Null)	7	Using whe

7、DEPENDENT SUBQUERY

子查询中的第 1 个 SELECT 语句, 取决于外面的查询。

8、DERIVED

SELECT(FROM 子句的子查询)。

2.1.3 table

表示查询的表。

有可能是

实际的表名 如 `select * from t1;`

表的别名 如 `select * from t2 as tmp;`

derived 如 from 型子查询时

null 直接计算得结果,不用走表

2.1.4 type (重要)

表示表的连接类型。

以下的连接类型的顺序是**从最佳类型到最差类型**:

1、system

表仅有一行,这是 `const` 类型的特列,平时不会出现,这个也可以忽略不计。

2、const

数据表最多只有一个匹配行,因为只匹配一行数据,所以很快,常用于 `PRIMARY KEY` 或者 `UNIQUE` 索引的查询,可理解为 **const 是最优化的**。

查询创建工具

查询编辑器

1

EXPLAIN SELECT * FROM tb_order WHERE id = 1

信息

结果1

概况

状态

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	tb_order	const	PRIMARY	PRIMARY	4	const	1	(Null)

3、eq_ref

mysql 手册是这样说的:"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。它用在索引的所有部分被联接使用并且索引是 `UNIQUE` 或 `PRIMARY KEY`"。eq_ref 可以用于使用=比较带索引的列。

1 EXPLAIN SELECT * FROM tb_order o,tb_user u WHERE u.id = o.user_id									
信息	结果1	概况	状态						
id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	o	ALL	FK_orders_1	(Null)	(Null)	(Null)	2	(Null)
1	SIMPLE	u	eq_ref	PRIMARY	PRIMAR	8	mybat	1	(Null)

4、ref

查询条件索引既不是 `UNIQUE` 也不是 `PRIMARY KEY` 的情况。ref 可用于=或<或>操作符的带索引的列。

```
1 EXPLAIN SELECT * FROM tb_order WHERE order_number = '20140921001'
```

信息	结果1	概况	状态						
id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	tb_order	ref	order_number	order_nt	62	const	1	Using index

5、ref_or_null

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。在解决子查询中经常使用该联接类型的优化。

上面这五种情况都是很理想的索引使用情况。

6、index_merge

该联接类型表示使用了索引合并优化方法。在这种情况下, **key** 列包含了使用的索引的清单, **key_len** 包含了使用的索引的最长的关键元素。

7、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。

8、index_subquery

该联接类型类似于 **unique_subquery**。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)**

9、range

只检索给定范围的行, 使用一个索引来选择行。

```
1 EXPLAIN SELECT * FROM tb_user WHERE age < 10
```

信息	结果1	概况	状态						
id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	tb_user	range	age	age	5	(Null)	1	Using index

10、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。

11、ALL

对于每个来自于先前的表的行组合, 进行完整的表扫描。(性能最差)

2.1.5 possible_keys

可能用到的索引

注意: 系统估计可能用的几个索引, 但最终, 只能用 1 个

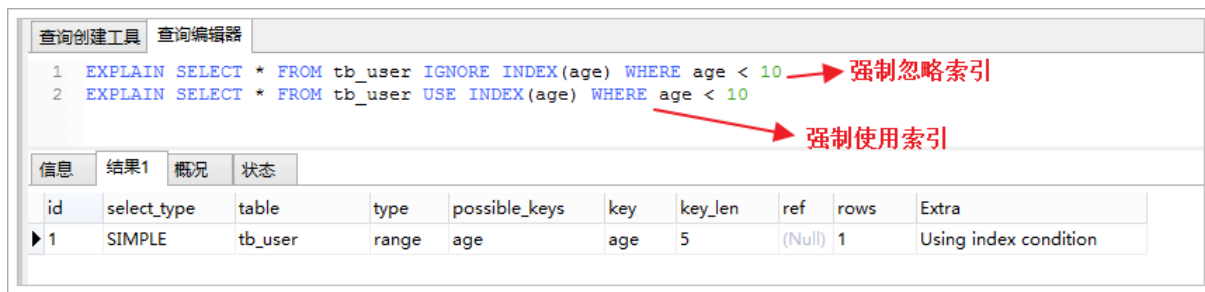
指出 MySQL 能使用哪个索引在该表中找到行。

如果该列为 **NULL**, 说明没有使用索引, 可以对该列创建索引来提供性能。

2.1.6 key

最终用的索引.显示 MySQL 实际决定使用的键(索引)。如果没有选择索引,键是 NULL。

可以强制使用索引或者忽略索引:



2.1.7 key_len

显示 MySQL 决定使用的键长度。如果键是 NULL,则长度为 NULL。

注意: key_len 是确定了 MySQL 将实际使用的索引长度。

使用的索引的最大长度

type 列: 是指查询的方式, 非常重要,是分析”查数据过程”的重要依据可能的值

all: 意味着从表的第 1 行,往后,逐行做全表扫描.,运气不好扫描到最后一行.

index: 比 all 性能稍好一点,

通俗的说: all 扫描所有的数据行,相当于 data_all index 扫描所有的索引节点,相当于 index_all

2.1.8 ref

显示使用哪个列或常数与 key 一起从表中选择行。

2.1.9 rows

显示 MySQL 认为它执行查询时必须检查的行数。

2.1.10 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 sort_union(...), Using union(...), Using intersect(...):**这些函数说明如何为 index_merge 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。

8.4 in 型子查询引出的陷阱

改进: 用连接查询来代替子查询

exists 子查询

优化 1: 在 group 时, 用带有索引的列来 group, 速度会稍快一些,另外, 用 int 型 比 char 型 分组,也要快一些.(见 37)

优化 2: 在 group 时, 我们假设只取了 A 表的内容,group by 的列,尽量用 A 表的列, 会比 B 表的列要快.

优化 3: 从语义上去优化

8.5 from 型子查询:

注意::内层 from 语句查到的临时表, 是没有索引的.

所以: from 的返回内容要尽量少.

奇技淫巧!

min/max 优化 在表中,一般都是经过优化的. 如下地区表

id	area	pid
1	中国	0
2	北京	1
...		
3115		3113

我们查 min(id), id 是主键,查 Min(id)非常快.

但是,pid 上没有索引, 现在要求查询 3113 地区的 min(id);

```
select min(id) from it_area where pid=69;
```

试想 id 是有顺序的,(默认索引是升序排列), 因此,如果我们沿着 id 的索引方向走, 那么 第 1 个 pid=69 的索引结点,他的 id 就正好是最小的 id.

```
select id from it_area use index(primary) where pid=69 limit 1;
```

```
| 12 | 0.00128100 | select min(id) from it_area where pid=69 |
| 13 | 0.00017000 | select id from it_area use index(primary) where pid=69 limit 1 |
```

改进后的速度虽然快,但语义已经非常不清晰,不建议这么做,仅仅是实验目的.

8.6 count() 优化

误区:

1:myisam 的 count()非常快

答: 是比较快,但仅限于查询表的”所有行”比较快, 因为 Myisam 对行数进行了存储.

一旦有条件的查询, 速度就不再快了.尤其是 where 条件的列上没有索引.

2: 假如,id<100 的商家都是我们内部测试的,我们想查查真实的商家有多少?

select count(*) from lx_com where id>=100; (1000 多万行用了 6.X 秒)

小技巧:

select count(*) from lx_com; 快

select count(*) from lx_com where id<100; 快

select count(*) from lx_com - select count(*) from lx_com where id<100; 快

select (select count(*) from lx_com) - (select count(*) from lx_com where id<100)

8.7 group by

注意:

1:分组用于统计,而不用于筛选数据.

比如: 统计平均分,最高分,适合, 但用于筛选重复数据,则不适合.

以及用索引来避免临时表和文件排序

2: 以 A,B 表连接为例 ,主要查询 A 表的列,

那么 group by ,order by 的列尽量相同,而且列应该显示声明为 A 的列

8.8 union 优化

注意: union all 不过滤 效率提高,如非必须,请用 union all

因为 union 去重的代价非常高, 放在程序里去重.

limit 及翻页优化

limit offset,N, 当 offset 非常大时, 效率极低,
原因是 mysql 并不是跳过 offset 行,然后单取 N 行,
而是取 offset+N 行,返回放弃前 offset 行,返回 N 行.
效率较低,当 offset 越大时,效率越低

优化办法:

1: 从业务上去解决

办法: 不允许翻过 100 页

以百度为例,一般翻页到 70 页左右.

1:不用 offset,用条件查询.

3: 非要物理删除,还要用 offset 精确查询,还不限制用户分页,怎么办?

分析: 优化思路是 不查,少查,查索引,少取.

我们现在必须要查,则只查索引,不查数据,得到 id.

再用 id 去查具体条目. 这种技巧就是延迟索引.

8.9 巧用变量

1:用变量排名

例: 以 ecshop 中的商品表为例,计算每个栏目下的商品数,并按商品数排名.

```
select cat_id,count(*) as cnt from goods group by cat_id order by cnt desc;
```

并按商品数计算这些栏目的名次

```
set @curr_cnt := 0,@prev_cnt := 0, @rank := 0;
```

```
select cat_id, (@curr_cnt := cnt) as cnt,
```

```
(@rank := if(@curr_cnt <> @prev_cnt,@rank+1,@rank)) as rank,
```

```
@prev_cnt := @curr_cnt
```

```
from ( select cat_id,count(*) as cnt from shop. goods group by shop. goods.cat_id order by cnt desc) as tmp;
```

2:用变量计算真正影响的行数

当插入多条,当主键重复时,则自动更新,这种效果,可以用 insert on duplication for update

要统计真正”新增”的条目, 如下图,我们想得到的值是”1”,即被更新的行数.

```
mysql> select * from user;
+-----+-----+-----+-----+-----+
| uid | uname | email | pass | status |
+-----+-----+-----+-----+-----+
| 1 | NXpHgfoA | 328268186@qq.com | | 1 |
| 4 | dsafd | | | 0 |
| 3 | lisi | | | 0 |
| 2 | cvadsfdf | | | 0 |
+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)

mysql> insert into user (uid,uname) values (4,'zhaoliu'),(5,'wangwu')
-> on duplicate key update uname=values(uname);
Query OK, 3 rows affected (0.00 sec)
Records: 2 Duplicates: 1 Warnings: 0
```

```
insert into user (uid,uname) values (4,'ds'),(5,'wanu'),(6,'safdsaf')
```

```
on duplicate key update uid=values(uid)+(0*(@x:=@x+1)) , uname=values(uname);
```

```
mysql> set @x:=0;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> insert into user (uid,uname) values (4,'ds'),(5,'wanu'),(6,'safdsaf')
-> on duplicate key update uid=values(uid)+(0*(@x:=@x+1)) , uname=values(uname);
Query OK, 5 rows affected (0.00 sec)
Records: 3 Duplicates: 2 Warnings: 0

mysql> select @x;
+-----+
| @x |
+-----+
| 2 |
+-----+
1 row in set (0.00 sec)
```

总影响行数-2*实际 update 数, 即新增的行数.

3: 简化 union

比如有新闻表,news , news_hot,

news_hot 是一张内存表,非常快,用来今天的热门新闻.

首页取新闻时,逻辑是这样的:

先取 hot, 没有 再取 news,为了省事,用一个 union 来完成.

select nid,title from news_hot where nid=xxx

union

select nid,title from news where nid=xxx;

```
mysql> select * from news;
+----+-----+
| id | content                |
+----+-----+
| 1  | 论酒后驾始的危害      |
| 2  | tomorrow is another day |
+----+-----+
2 rows in set (0.00 sec)

mysql> select * from news2;
+----+-----+
| id | content                |
+----+-----+
| 3  | class object fatal error |
+----+-----+
1 row in set (0.00 sec)

mysql> select * from news
-> where id=1
-> union
-> select * from news2 where id=1;
+----+-----+
| id | content                |
+----+-----+
| 1  | 论酒后驾始的危害      |
+----+-----+
1 row in set (0.00 sec)
```

后半句select
没必要执行

如何利用变量让后半句 select 不执行 ,

select id,content,(@find := 1) from news where id=1

union

select id,content,(@find :=1) from news2 where id=1 and (@find <= 0)

union 1,1,1 where (@find :=null) is not null;

3:小心变量的顺序

如下图:变量先在 where 中发挥作用,然后再是 select 操作.

如果 where 不成立,select 操作再不发生.

```
mysql> set Cnum := 0;
Query OK, 0 rows affected (0.00 sec)

mysql> select uid,Cnum := Cnum+1 from user where Cnum<=1;
+-----+-----+
| uid | Cnum := Cnum+1 |
+-----+-----+
| 1 | 1 |
| 2 | 2 |
+-----+-----+
2 rows in set (0.00 sec)
```

注意,where条件先发挥作用,然后再取值.

Eg;

```
mysql> explain select uid,uname,Cnum := Cnum+1 from user where Cnum<=1 order by
uid desc \G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: user
         type: ALL
possible_keys: NULL
         key: NULL
        key_len: NULL
         ref: NULL
         rows: 2
   Extra: Using where; Using filesort
1 row in set (0.00 sec)
```

在这个例子中,1,2 两行,先排好序,在内存中,就是这样的顺序 [2] [1]
再逐行 where 条件判断,取值.

```
mysql> select uid,uname,Cnum from user where (Cnum:=Cnum+1) <=1;
+-----+-----+-----+
| uid | uname | Cnum |
+-----+-----+-----+
| 1 | NXpHgfoA | 1 |
+-----+-----+-----+
```

```
mysql> select uid,uname,@num from user where (@num:=@num+1) <=3 order by uid desc;
+-----+-----+-----+
| uid | uname   | @num |
+-----+-----+-----+
| 3   | dfas    | 3     |
| 2   | cvadsfdf| 3     |
| 1   | NXpHgfoA| 3     |
+-----+-----+-----+
```

```
mysql> select uid,uname,@num:=@num+1 from user where @num <=3 order by uid desc;
+-----+-----+-----+
| uid | uname   | @num:=@num+1 |
+-----+-----+-----+
| 3   | dfas    | 1             |
| 2   | cvadsfdf| 2             |
| 1   | NXpHgfoA| 3             |
+-----+-----+-----+
```

对比这 2 张图,分析:

- 1: where 先发挥作用,把需要的行 都给找出
- 2: 然后再逐行 select

因此,前者,最终 select 时,select@num 变量,都是一个值
后者,不断 select,不断修改@num 的值,值不断变化.

同时: 使用变量,将会使 sql 语句的结果不缓存.

8.10 子查询优化

MySQL 从 4.1 版本开始支持子查询,使用子查询进行 SELECT 语句嵌套查询,可以一次完成很多逻辑上需要多个步骤才能完成的 SQL 操作。

子查询虽然很灵活,但是执行效率并不高。

执行子查询时,MYSQL 需要创建临时表,查询完毕后再删除这些临时表,所以,子查询的速度会受到一定的影响。

优化:

可以使用连接查询 (JOIN) 代替子查询,连接查询时不需要建立临时表,其速度比子查询快。

Rbac-----两种 SQL 语句

9 索引及优化

数据库索引的本质是数据结构, 这种数据结构能够帮助我们快速的获取数据库中的数据。

9.1 索引的作用

当表中的数据量越来越大时, 索引对于性能的影响愈发重要。索引优化应该是对查询性能优化最有效的手段了。索引能够轻易将查询性能提高好几个数量级。有了索引相当于我们给数据库的数据加了目录一样, 可以快速的找到数据, 如果不适用索引则需要一点一点去查找数据。简单来说提高数据查询的效率。

9.2 索引的分类

- ◆ 普通索引 `index`: 加速查找
- ◆ 唯一索引
 - 主键索引: `primary key`: 加速查找+约束 (不为空且唯一)
 - 唯一索引: `unique`: 加速查找+约束 (唯一)
- ◆ 联合索引 (组合索引)
 - `primary key(id,name)`: 联合主键索引
 - `unique(id,name)`: 联合唯一索引
 - `index(id,name)`: 联合普通索引
- ◆ 全文索引 `fulltext`: 用于搜索很长一篇文章的时候, 效果最好。
- ◆ 空间索引 `spatial`: 了解就好, 几乎不用

9.3 索引的优点

1. 可以通过建立唯一索引或者主键索引, 保证数据库表中每一行数据的唯一性.
2. 建立索引可以大大提高检索的数据, 以及减少表的检索行数
3. 在表连接的条件 可以加速表与表直接的相连
4. 在分组和排序字句进行数据检索, 可以减少查询时间中 分组 和 排序时所消耗的时间(数据库的记录会重新排序)
5. 建立索引, 在查询中使用索引 可以提高性能

9.4 索引的缺点

1. 在创建索引和维护索引 会耗费时间, 随着数据量的增加而增加
2. 索引文件会占用物理空间, 除了数据表需要占用物理空间之外, 每一个索引还会占用一定的物理空间
3. 当对表的数据进行 `INSERT, UPDATE, DELETE` 的时候, 索引也要动态的维护, 这样就会降低数据的维护速度, (建立索引会占用磁盘空间的索引文件。一般情况这个问题不太严重, 但如果你在一个大表上创建了多种组合索引, 索引文件的会膨胀很快)。

9.5 哪些情况或字段适合加索引

- 1.在经常需要搜索的列上,可以加快索引的速度
- 2.主键列上可以确保列的唯一性
- 3.在表与表的而连接条件上加上索引,可以加快连接查询的速度
- 4.在经常需要排序(order by),分组(group by)和的 distinct 列上加索引 可以加快排序查询的时间,

9.6 哪些情况不适合创建索引

- 1.查询中很少使用到的列 不应该创建索引,如果建立了索引然而还会降低 mysql 的性能和增大了空间需求.
- 2.很少数据的列也不应该建立索引,比如 一个性别字段 0 或者 1,在查询中,结果集的数据占了表中数据行的比例比较大,mysql 需要扫描的行数很多,增加索引,并不能提高效率
- 3.定义为 text 和 image 和 bit 数据类型的列不应该增加索引,
- 4.当表的修改(UPDATE,INSERT,DELETE)操作远远大于检索(SELECT)操作时不应该创建索引,这两个操作是互斥的关系。

9.7 哪些情况会造成索引失效

- 1.如果条件中有 or, 即使其中有条件带索引也不会使用(这也是为什么尽量少用 or 的原因)
- 2.索引字段的值不能有 null 值, 有 null 值会使该列索引失效
- 3.对于多列索引, 不是使用的第一部分, 则不会使用索引 (最左原则)
- 4.like 查询以%开头
- 5.如果列类型是字符串, 那一定要在条件中将数据使用单引号引用起来,否则不使用索引
- 6.在索引的列上使用表达式或者函数会使索引失效

例如: select * from users where YEAR(adddate) < 2007, 将在每个行上进行运算, 这将导致索引失效而进行全表扫描, 因此我们可以改成: select * from users where adddate < '2007-01-01'。

9.8 索引相关 sql

9.8.1 创建索引

```
mysql>ALTER TABLE 表名 ADD INDEX 索引名 列名;  
mysql>ALTER TABLE 表名 ADD UNIQUE 索引名 列名;  
mysql>ALTER TABLE 表名 ADD PRIMARY KEY 索引名 列名;  
mysql>CREATE INDEX 索引名 ON 表名 列名;  
mysql>CREATE UNIQUE INDEX 索引名 ON 表名 列名;
```

9.8.2 删除索引:

```
DROP INDEX IndexName ON TableName
```

9.8.3 查看索引

show index from tableName;

9.9 索引原理

- MySQL 的基本存储结构是页(记录都存在页里边):
- 各个数据页可以组成一个双向链表
- 每个数据页中的记录又可以组成一个单向链表
 - 每个数据页都会为存储在它里边儿的记录生成一个页目录, 在通过主键查找某条记录的时候可以在页目录中使用二分法快速定位到对应的槽, 然后再遍历该槽对应分组中的记录即可快速找到指定的记录
 - 以其他列(非主键)作为搜索条件: 只能从最小记录开始依次遍历单链表中的每条记录。

所以说, 如果我们写 `select * from user where indexname = 'xxx'` 这样没有进行任何优化的 sql 语句, 默认会这样做:

1. 定位到记录所在的页: 需要遍历双向链表, 找到所在的页
2. 从所在的页内中查找相应的记录: 由于不是根据主键查询, 只能遍历所在页的单链表了

很明显, 在数据量很大的情况下这样查找会很慢! 这样的时间复杂度为 $O(n)$ 。

- 使用索引之后

索引做了些什么可以让我们查询加快速度呢? 其实就是将无序的数据变成有序(相对):

很明显的是: 没有用索引我们是需要遍历双向链表来定位对应的页, 现在通过“目录”就可以很快地定位到对应的页上了! (二分查找, 时间复杂度近似为 $O(\log n)$)

其实底层结构就是 B+树, B+树作为树的一种实现, 能够让我们很快地查找出对应的记录。

索引被用来快速找出在一个列上用一特定值的行。没有索引, MySQL 不得不首先以第一条记录开始, 然后读整个表直到它找出相关的行。表越大, 花费时间越多。对于一个有序字段, 可以运用二分查找 (Binary Search), 这就是为什么性能能得到本质上的提高。

MYISAM 和 INNODB 都是用 B+Tree 作为索引结构

(主键, unique 都会默认地添加索引)

索引的实现本质上是为了让数据库能够快速查找数据, 而单独维护的数据结构。mysql 实现索引主要使用的两种数据结构: hash 和 B+树: 我们比较常用的 MyIsam 和 innorDB 引擎都是基于 B+树的。

- ◆ hash: (hash 索引在 mysql 比较少用) 他以把数据的索引以 hash 形式组织起来, 因此当查找某一条记录的时候, 速度非常快. 当时因为是 hash 结构, 每个键只对应一个值, 而且是散列的方式分布. 所以他并不支持范围查找和排序等功能.
- ◆ B+树: b+tree 是(mysql 使用最频繁的一个索引数据结构) 数据结构以平衡树的形式来组织, 因为是树型结构, 所以更适合用来处理排序, 范围查找等功能. 相对 hash 索引, B+树在查找单条记录的速度虽然比不上 hash 索引, 但是因为更适合排序等操作, 所以他更受用户的欢迎. 毕竟不可能只对数据库进行单条记录的操作.

9.9.1 BTree 索引

大的方面看, 都用的平衡树, 但具体的实现上, 各引擎稍有不同, 比如, 严格的说, NDB 引擎使用的是 T-tree
Myisam, innodb 中, 默认用 B-tree 索引

但抽象一下---B-tree 系统, 可理解为” 排好序的快速查找结构”。

BTree 是平衡搜索多叉树, 设树的度为 $2d$ ($d>1$), 高度为 h , 那么 BTree 要满足以下条件:

每个叶子结点的高度一样, 等于 h ;

每个非叶子结点由 $n-1$ 个 key 和 n 个指针 point 组成, 其中 $d \leq n \leq 2d$, key 和 point 相互间隔, 结点两端一定是 key;

叶子结点指针都为 null;

非叶子结点的 key 都是 [key,data] 二元组, 其中 key 表示作为索引的键, data 为键值所在行的数据;

9.9.2 B+Tree 索引

B+Tree 是 BTree 的一个变种, 设 d 为树的度数, h 为树的高度, B+Tree 和 BTree 的不同主要在于:

B+Tree 中的非叶子结点不存储数据, 只存储键值;

B+Tree 的叶子结点没有指针, 所有键值都会出现在叶子结点上, 且 key 存储的键值对应 data 数据的物理地址;

B+Tree 的每个非叶子节点由 n 个键值 key 和 n 个指针 point 组成;

9.9.1 hash 索引

在 memory 表里, 默认是 hash 索引, hash 的理论查询时间复杂度为 $O(1)$ 疑问

既然 hash 的查找如此高效, 为什么不都用 hash 索引?

答:

1: hash 函数计算后的结果, 是随机的, 如果是在磁盘上放置数据,

比主键为 id 为例, 那么随着 id 的增长, id 对应的行, 在磁盘上随机放置.

2: 无法对范围查询进行优化.

3: 无法利用前缀索引. 比如 在 btree 中, field 列的值 "helloworld", 并加索引

查询 `xx=helloworld`, 自然可以利用索引, `xx=hello`, 也可以利用索引. (左前缀索引)

因为 `hash('helloworld')`, 和 `hash('hello')`, 两者的关系仍为随机

4: 排序也无法优化.

5: 必须回行. 就是说 通过索引拿到数据位置, 必须回到表中取数据

9.9.1 btree 索引的常见误区

9.9.1.1 在 where 条件常用的列上都加上索引

例: `where cat_id=3 and price>100`; // 查询第 3 个栏目, 100 元以上的商品

误: `cat_id` 上, 和, `price` 上都加上索引.

错: 只能用上 `cat_id` 或 `Price` 索引, 因为是独立的索引, 同时只能用上 1 个.

9.9.1.2 在多列上建立索引后, 查询哪个列, 索引都将发挥作用

误: 多列索引上, 索引发挥作用, 需要满足左前缀要求.

以 `index(a,b,c)` 为例,

语句	索引是否发挥作用
Where a=3	是,只使用了 a 列
Where a=3 and b=5	是,使用了 a,b 列
Where a=3 and b=5 and c=4	是,使用了 abc
Where b=3 / where c=4	否
Where a=3 and c=4	a 列能发挥索引,c 不能
Where a=3 and b>10 and c=7	A 能利用,b 能利用, C 不能利用
同上,where a=3 and b like 'xxxx%' and c=7	A 能用,B 能用,C 不能用

多列索引经典题目:

<http://www.zixue.it/thread-9218-1-4.html>

9.9.1.3 面试题

有商品表, 有主键,goods_id, 栏目列 cat_id, 价格 price

说:在价格列上已经加了索引,但按价格查询还是很慢,

问可能是什么原因,怎么解决?

答: 在实际场景中,一个电商网站的商品分类很多,直接在所有商品中,按价格查商品,是极少的,一般客户都来到分类下,然后再查.

改正: 去掉单独的 Price 列的索引, 加 (cat_id,price)复合索引再查询.

9.10 索引优化策略

1. 选择唯一性索引
2. 唯一性索引的值是唯一的, 可以更快速的通过该索引来确定某条记录。
3. 为经常需要排序、分组和联合操作的字段建立索引:
4. 为常作为查询条件的字段建立索引。
5. 限制索引的数目: 越多的索引, 会使更新表变得很浪费时间。
6. 尽量使用数据量少的索引
7. 如果索引的值很长, 那么查询的速度会受到影响。
8. 尽量使用前缀来索引
9. 如果索引字段的值很长, 最好使用值的前缀来索引。
10. 删除不再使用或者很少使用的索引
11. 最左前缀匹配原则, 非常重要的原则。
12. 尽量选择区分度高的列作为索引
13. 区分度的公式是表示字段不重复的比例
14. 索引列不能参与计算, 保持列“干净”: 带函数的查询不参与索引。
15. 尽可能的扩展索引, 不要新建索引。

9.10.1最左前缀原则

MySQL 中的索引可以以一定顺序引用多列, 这种索引叫作联合索引。如 User 表的 name 和 city 加联合索引就是 (name,city), 而最左前缀原则指的是, 如果查询的时候查询条件精确匹配索引的左边连续一列或几列, 则此列就可以被用到。如下:

```
select * from user where name=xx and city=xx ; // 可以命中索引
select * from user where name=xx ; // 可以命中索引
select * from user where city=xx ; // 无法命中索引
```

这里需要注意的是, 查询的时候如果两个条件都用上了, 但是顺序不同, 如 city= xx and name =xx, 那么现在的查询引擎会自动优化为匹配联合索引的顺序, 这样是能够命中索引的。

由于最左前缀原则, 在创建联合索引时, 索引字段的顺序需要考虑字段值去重之后的个数, 较多的放前面。ORDER BY 子句也遵循此规则。

9.10.2注意避免冗余索引

冗余索引指的是索引的功能相同, 能够命中 就肯定能命中 , 那么 就是冗余索引如 (name,city) 和 (name) 这两个索引就是冗余索引, 能够命中后者的查询肯定是能够命中前者的 在大多数情况下, 都应该尽量扩展已有的索引而不是创建新索引。

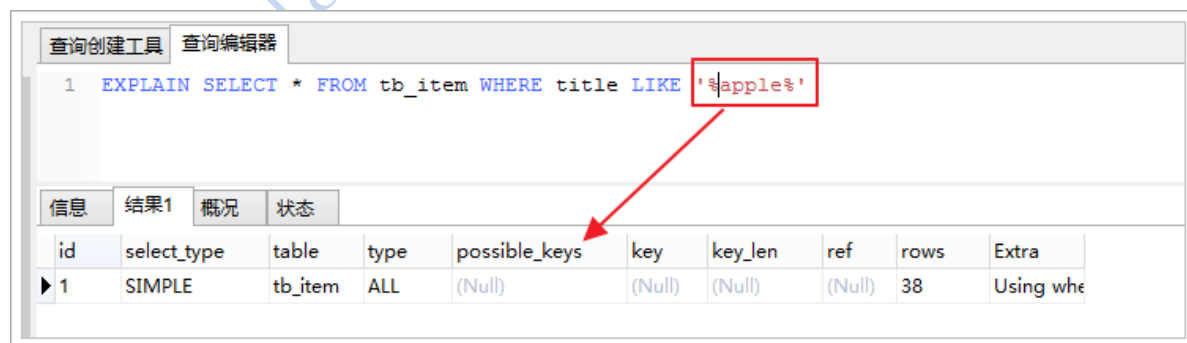
MySQL5.7 版本后, 可以通过查询 sys 库的 schema_redundant_indexes 表来查看冗余索引

9.10.3使用索引查询需要注意

索引可以提供查询的速度, 但并不是使用了带有索引的字段查询都会生效, 有些情况下是不生效的, 需要注意!

9.10.3.1 使用 LIKE 关键字的查询

在使用 LIKE 关键字进行查询的查询语句中, 如果匹配字符串的第一个字符为 “%”, 索引不起作用。只有 “%” 不在第一个位置, 索引才会生效。



id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	tb_item	ALL	(Null)	(Null)	(Null)	(Null)	38	Using where

查询创建工具

查询编辑器

```
1 EXPLAIN SELECT * FROM tb_item WHERE title LIKE 'appl%'
```

信息

结果1

概况

状态

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	tb_item	range	title	title	302	(Null)	1	Using index

9.10.3.2 使用联合索引的查询

MySQL 可以为多个字段创建索引，一个索引可以包括 16 个字段。对于联合索引，只有查询条件中使用了这些字段中第一个字段时，索引才会生效。

查询创建工具

查询编辑器

```
1 EXPLAIN SELECT * FROM tb_cart WHERE user_id = 1 AND item_id = 1
```

索引生效

信息

结果1

概况

状态

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	tb_cart	ref	userid_itemId	userid_itemId	18	const,cc 1		(Null)

查询创建工具

查询编辑器

1

EXPLAIN SELECT * FROM tb_cart WHERE user_id = 1

索引生效


信息	结果1	概况	状态																				
	<table><tr><th>id</th><th>select_type</th><th>table</th><th>type</th><th>possible_keys</th><th>key</th><th>key_len</th><th>ref</th><th>rows</th><th>Extra</th></tr><tr><td>1</td><td>SIMPLE</td><td>tb_cart</td><td>ref</td><td>userId_itemId</td><td>userId_itemId</td><td>9</td><td>const</td><td>1</td><td>(Null)</td></tr></table>	id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra	1	SIMPLE	tb_cart	ref	userId_itemId	userId_itemId	9	const	1	(Null)		
id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra														
1	SIMPLE	tb_cart	ref	userId_itemId	userId_itemId	9	const	1	(Null)														

查询创建工具

查询编辑器

```
1 EXPLAIN SELECT * FROM tb_cart WHERE item_id = 1
```

索引不生效，全表扫描



信息

结果1

概况

状态

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	tb_cart	ALL	(Null)	(Null)	(Null)	(Null)	1	Using where

9.10.3.3 使用 OR 关键字的查询

查询语句的查询条件中只有 OR 关键字，且 OR 前后的两个条件中的列都是索引时，索引才会生效，否则，索引不

生效。

查询创建工具 查询编辑器	
1 EXPLAIN SELECT * FROM tb_item WHERE id = 1 OR id = 2	
索引生效	
信息	结果1 概况 状态
id	select_type table type possible_keys key key_len ref rows Extra
1	SIMPLE tb_item range PRIMARY PRIMARY 8 (Null) 2 Using whe

查询创建工具 查询编辑器	
1 EXPLAIN SELECT * FROM tb_item WHERE id = 1 OR cid = 2	
索引生效	
信息	结果1 概况 状态
id	select_type table type possible_keys key key_len ref rows Extra
1	SIMPLE tb_item index_me PRIMARY,cid PRIMARY,cid 8,8 (Null) 2 Using unic

查询创建工具 查询编辑器	
1 EXPLAIN SELECT * FROM tb_item WHERE id = 1 OR price = 1000	
索引不生效	
信息	结果1 概况 状态
id	select_type table type possible_keys key key_len ref rows Extra
1	SIMPLE tb_item ALL PRIMARY (Null) (Null) (Null) 38 Using whe

9.10.4 聚簇索引与非聚簇索引【能说几句是几句】

Myisam 与 innodb 引擎,索引文件的异同

innodb 的主索引文件上,直接存放该行数据,称为聚簇索引,次索引指向对主键的引用

myisam 中,主索引和次索引,都指向物理行(磁盘位置).

注意: innodb 来说,

- 1: 主键索引 既存储索引值,又在叶子中存储行的数据
- 2: 如果没有主键, 则会 Unique key 做主键
- 3: 如果没有 unique,则系统生成一个内部的 rowid 做主键.
- 4: 像 innodb 中,主键的索引结构中,既存储了主键值,又存储了行数据,这种结构称为"聚簇索引"

聚簇索引

优势: 根据主键查询条目比较少时,不用回行(数据就在主键节点下)

劣势: 如果碰到不规则数据插入时,造成频繁的页分裂.

9.10.4.1 高性能索引策略

0:对于 innodb 而言,因为节点下有数据文件,因此节点的分裂将会比较慢.

对于 innodb 的主键,尽量用整型,而且是递增的整型.

如果是无规律的数据,将会产生的页的分裂,影响速度.

9.10.4.2 索引覆盖:

索引覆盖是指 如果查询的列恰好是索引的一部分,那么查询只需要在索引文件上进行,不需要回行到磁盘再找数据. 这种查询速度非常快,称为"索引覆盖"

9.10.4.3 理想的索引

1:查询频繁 2:区分度高 3:长度小 4: 尽量能覆盖常用查询字段.

1: 索引长度直接影响索引文件的大小,影响增删改的速度,并间接影响查询速度(占用内存多).

针对列中的值,从左往右截取部分,来建索引

1: 截的越短, 重复度越高,区分度越小, 索引效果越不好

2: 截的越长, 重复度越低,区分度越高, 索引效果越好,但带来的影响也越大--增删改变慢,并间接影响查询速度.

所以, 我们要在 区分度 + 长度 两者上,取得一个平衡.

对于一般的系统应用: 区别度能达到 0.1,索引的性能就可以接受.

9.10.5对于左前缀不易区分的列 ,建立索引的技巧

如 url 列

<http://www.baidu.com>

<http://www.zixue.it>

列的前 11 个字符都是一样的,不易区分,可以用如下 2 个办法来解决

1: 把列内容倒过来存储,并建立索引

Moc.udiab.www//:ptth

Ti.euxiz.www//:ptth

这样左前缀区分度大,

2: 伪 hash 索引效果

同时存 url_hash 列

9.10.6多列索引

多列索引的考虑因素--- 列的查询频率 , 列的区分度,

9.10.6.1 索引与排序

排序可能发生 2 种情况:

- 1: 对于覆盖索引,直接在索引上查询时,就是有顺序的, using index
- 2: 先取出数据,形成临时表做 filesort(文件排序,但文件可能在磁盘上,也可能在内存中)

我们的争取目标-----取出来的数据本身就是有序的! 利用索引来排序.

9.10.6.2 重复索引与冗余索引

重复索引: 是指 在同 1 个列(如 age), 或者 顺序相同的几个列(age,school), 建立了多个索引,称为重复索引, 重复索引没有任何帮助,只会增大索引文件,拖慢更新速度, 需要去掉.

冗余索引:

冗余索引是指 2 个索引所覆盖的列有重叠, 称为冗余索引

比如 x,m,列 , 加索引 index x(x), index xm(x,m)

x,xm 索引, 两者的 x 列重叠了, 这种情况,称为冗余索引.

甚至可以把 index mx(m,x) 索引也建立, mx, xm 也不是重复的,因为列的顺序不一样.

9.10.6.3 索引碎片与维护

在长期的数据更改过程中, 索引文件和数据文件,都将产生空洞,形成碎片.

我们可以通过一个 nop 操作(不产生对数据实质影响的操作), 来修改表.

比如: 表的引擎为 innodb, 可以 alter table xxx engine innodb

optimize table 表名, 也可以修复.

注意: 修复表的数据及索引碎片,就会把所有的数据文件重新整理一遍,使之对齐.

这个过程,如果表的行数比较大,也是非常耗费资源的操作.

所以,不能频繁的修复.

如果表的 Update 操作很频率,可以按周/月,来修复.

如果不频繁,可以更长的周期来做修复.

9.10.7 限制每张表上的索引数量(5 个)

建议单张表索引不超过 5 个;索引并不是越多越好! 索引可以提高效率同样可以降低效率。

索引可以增加查询效率, 但同样也会降低插入和更新的效率, 甚至有些情况下会降低查询效率。

因为 MySQL 优化器在选择如何优化查询时, 会根据统一信息, 对每一个可以用到的索引来进行评估, 以生成出一个最好的执行计划, 如果同时有很多个索引都可以用于查询, 就会增加 MySQL 优化器生成执行计划的时间, 同样会降低查询性能。

9.10.8 禁止给表中的每一列都建立单独的索引

5.6 版本之前, 一个 sql 只能使用到一个表中的一个索引, 5.6 以后, 虽然有了合并索引的优化方式, 但是还是远远没有使用一个联合索引的查询方式好。

9.10.9 每个 Innodb 表必须有个主键

Innodb 是一种索引组织表: 数据的存储的逻辑顺序和索引的顺序是相同的。每个表都可以有多个索引, 但是表的存储顺序只能有一种。

Innodb 是按照主键索引的顺序来组织表的

- 不要使用更新频繁的列作为主键, 不适用多列主键 (相当于联合索引)
- 不要使用 UUID, MD5, HASH, 字符串列作为主键 (无法保证数据的顺序增长)
- 主键建议使用自增 ID 值

9.10.10 常见索引列建议

- 出现在 SELECT、UPDATE、DELETE 语句的 WHERE 从句中的列
- 包含在 ORDER BY、GROUP BY、DISTINCT 中的字段
- 并不要将符合 1 和 2 中的字段的列都建立一个索引, 通常将 1、2 中的字段建立联合索引效果更好
- 多表 join 的关联列

9.10.11 如何选择索引列的顺序

建立索引的目的是: 希望通过索引进行数据查找, 减少随机 IO, 增加查询性能, 索引能过滤出越少的数据, 则从磁盘中读入的数据也就越少。

- 区分度最高的放在联合索引的最左侧 (区分度=列中不同值的数量/列的总行数)
- 尽量把字段长度小的列放在联合索引的最左侧 (因为字段长度越小, 一页能存储的数据量越大, IO 性能也就越好)
- 使用最频繁的列放到联合索引的左侧 (这样可以比较少的建立一些索引)

9.10.12 避免建立冗余索引和重复索引 (增加了查询优化器生成执行计划的时间)

- 重复索引示例: `primary key(id)`、`index(id)`、`unique index(id)`
- 冗余索引示例: `index(a,b,c)`、`index(a,b)`、`index(a)`

9.10.13 对于频繁的查询优先考虑使用覆盖索引

覆盖索引: 就是包含了所有查询字段 (`where`, `select`, `order by`, `group by` 包含的字段) 的索引

覆盖索引的好处:

- **避免 Innodb 表进行索引的二次查询:** Innodb 是以聚集索引的顺序来存储的, 对于 Innodb 来说, 二级索引在叶子节点中所保存的是行的主键信息, 如果是用二级索引查询数据的话, 在查找到相应的键值后, 还要通过主键进行二次查询才能获取我们真实所需要的数据。而在覆盖索引中, 二级索引的键值中可以获取所有的数据, 避免了对主键的二次查询, 减少了 IO 操作, 提升了查询效率。
- **可以把随机 IO 变成顺序 IO 加快查询效率:** 由于覆盖索引是按键值的顺序存储的, 对于 IO 密集型的范围查找来说, 对比随机从磁盘读取每一行的数据 IO 要少的多, 因此利用覆盖索引在访问时也可以把磁盘的随机读取的 IO 转变成索引查找的顺序 IO。

9.10.14 索引 SET 规范

尽量避免使用外键约束

- 不建议使用外键约束 (`foreign key`), 但一定要在表与表之间的关联键上建立索引
- 外键可用于保证数据的参照完整性, 但建议在业务端实现
- 外键会影响父表和子表的写操作从而降低性能

10 插入数据的优化

插入数据时, 影响插入速度的主要是索引、唯一性校验、一次插入的数据条数等。

插入数据的优化, 不同的存储引擎优化手段不一样, 在 MySQL 中常用的存储引擎有, MyISAM 和 InnoDB, 两者的区别:

<http://www.cnblogs.com/panfeng412/archive/2011/08/16/2140364.html>

MyISAM是MySQL的默认存储引擎, 基于传统的ISAM类型, 支持全文搜索, 但不是事务安全的, 而且不支持外键。每张MyISAM表存放在三个文件中: frm 文件存放表定义; 数据文件是MYD (MYData); 索引文件是MYI (MYIndex)。

InnoDB是事务型引擎, 支持回滚、崩溃恢复能力、多版本并发控制、ACID事务, 支持行级锁定 (InnoDB表的行锁不是绝对的, 如果在执行一个SQL语句时MySQL不能确定要扫描的范围, InnoDB表同样会锁全表, 如like操作时的SQL语句), 以及提供与Oracle类型一致的不加锁读取方式。InnoDB存储它的表和索引在一个表空间中, 表空间可以包含数个文件。

主要区别:

- MyISAM是非事务安全型的, 而InnoDB是事务安全型的。
- MyISAM锁的粒度是表级, 而InnoDB支持行级锁定。
- MyISAM支持全文类型索引, 而InnoDB不支持全文索引。
- MyISAM相对简单, 所以在效率上要优于InnoDB, 小型应用可以考虑使用MyISAM。
- MyISAM表是保存成文件的形式, 在跨平台的数据转移中使用MyISAM存储会省去不少的麻烦。
- InnoDB表比MyISAM表更安全, 可以在保证数据不会丢失的情况下, 切换非事务表到事务表 (alter table tablename type=innodb)。

应用场景:

- MyISAM管理非事务表。它提供高速存储和检索, 以及全文搜索能力。如果应用中需要执行大量的SELECT查询, 那么MyISAM是更好的选择。
- InnoDB用于事务处理应用程序, 具有众多特性, 包括ACID事务支持。如果应用中需要执行大量的INSERT或UPDATE操作, 则应该使用InnoDB, 这样可以提高多用户并发操作的性能。

10.1 MyISAM

10.1.1 禁用索引

对于非空表, 插入记录时, MySQL 会根据表的索引对插入的记录建立索引。如果插入大量数据, 建立索引会降低插入数据速度。

为了解决这个问题, 可以在批量插入数据之前禁用索引, 数据插入完成后再开启索引。

禁用索引的语句:

```
ALTER TABLE table_name DISABLE KEYS
```

开启索引语句:

```
ALTER TABLE table_name ENABLE KEYS
```

对于空表批量插入数据, 则不需要进行操作, 因为 MyISAM 引擎的表是在导入数据后才建立索引。

10.1.2 禁用唯一性检查

唯一性校验会降低插入记录的速度, 可以在插入记录之前禁用唯一性检查, 插入数据完成后再开启。

禁用唯一性检查的语句: `SET UNIQUE_CHECKS = 0;`

开启唯一性检查的语句: `SET UNIQUE_CHECKS = 1;`

10.1.3 批量插入数据

插入数据时, 可以使用一条 `INSERT` 语句插入一条数据, 也可以插入多条数据。

```
1  INSERT INTO tb_cart
2  VALUES
3  (
4      '40',
5      '6',
6      '36',
7      '苹果 (APPLE) iPhone 6 A1586 16G版 4G手机 (深空灰)',
8      'http://image.taobao.com/images/2014/10/23/201410230452400280709.jpg',
9      '5288000',
10     '1',
11     '2014-12-11 18:07:46',
12     '2014-12-12 16:52:55'
13 );
14 INSERT INTO tb_cart
15 VALUES
16 (
17     '41',
18     '6',
19     '36',
20     '苹果 (APPLE) iPhone 6 A1586 16G版 4G手机 (深空灰)',
21     'http://image.taobao.com/images/2014/10/23/201410230452400280709.jpg',
22     '5288000',
23     '1',
24     '2014-12-11 18:07:46',
25     '2014-12-12 16:52:55'
26 );
27
```

```
1  INSERT INTO tb_cart
2  VALUES
3  (
4      '40',
5      '6',
6      '36',
7      '苹果 (APPLE) iPhone 6 A1586 16G版 4G手机 (深空灰)',
8      'http://image.taobao.com/images/2014/10/23/201410230452400280709.jpg',
9      '5288000',
10     '1',
11     '2014-12-11 18:07:46',
12     '2014-12-12 16:52:55'
13 )
14 ,
15 (
16     '40',
17     '6',
18     '36',
19     '苹果 (APPLE) iPhone 6 A1586 16G版 4G手机 (深空灰)',
20     'http://image.taobao.com/images/2014/10/23/201410230452400280709.jpg',
21     '5288000',
22     '1',
23     '2014-12-11 18:07:46',
24     '2014-12-12 16:52:55'
25 );
26
```

第二种方式的插入速度比第一种方式快。

10.1.4 使用 LOAD DATA INFILE

当需要批量导入数据时, 使用 **LOAD DATA INFILE** 语句比 **INSERT** 语句插入速度快很多。

10.2 InnoDB

10.2.1 禁用唯一性检查

用法和 MyISAM 一样。

10.2.2 禁用外键检查

插入数据之前执行禁止对外键的检查, 数据插入完成后再恢复, 可以提供插入速度。

禁用: SET foreign_key_checks = 0;

开启: SET foreign_key_checks = 1;

10.2.3 禁止自动提交

插入数据之前执行禁止事务的自动提交, 数据插入完成后再恢复, 可以提高插入速度。

禁用: SET autocommit = 0;

开启: SET autocommit = 1;

11 服务器优化

11.1 优化服务器硬件

服务器的硬件性能直接决定着 MySQL 数据库的性能, 硬件的性能瓶颈, 直接决定 MySQL 数据库的运行速度和效率。

需要从以下几个方面考虑:

- 1、配置较大的内存。足够大的内存, 是提高 MySQL 数据库性能的方法之一。内存的 IO 比硬盘快的多, 可以增加系统的缓冲区容量, 使数据在内存停留的时间更长, 以减少磁盘的 IO。
- 2、配置高速磁盘, 比如 SSD。
- 3、合理分配磁盘 IO, 把磁盘 IO 分散到多个设备上, 以减少资源的竞争, 提高并行操作能力。
- 4、配置多核处理器, MySQL 是多线程的数据库, 多处理器可以提高同时执行多个线程的能力。

11.2 优化 MySQL 的参数

通过优化 MySQL 的参数可以提高资源利用率, 从而达到提高 MySQL 服务器性能的目的。

MySQL 的配置参数都在 `my.conf` 或者 `my.ini` 文件的 `[mysqld]` 组中, 常用的参数如下:

- `key_buffer_size`: 表示索引缓冲区的大小。索引缓冲区所有的线程共享。增加索引缓冲区可以得到更好处理的索引 (对所有读和多重写)。当然, 这个值也不是越大越好, 它的大小取决于内存的大小。如果这个值太大, 导致操作系统频繁换页, 也会降低系统性能。

- `table_cache`: 表示同时打开的表的个数。这个值越大, 能够同时打开的表的个数越多。这个值不是越大越好, 因为同时打开的表太多会影响操作系统的性能。
- `query_cache_size`: 表示查询缓冲区的大小。该参数需要和 `query_cache_type` 配合使用。当 `query_cache_type` 值是 0 时, 所有的查询都不使用查询缓冲区。但是 `query_cache_type=0` 并不会导致 MySQL 释放 `query_cache_size` 所配置的缓冲区内存。当 `query_cache_type=1` 时, 所有的查询都将使用查询缓冲区, 除非在查询语句中指定 `SQL_NO_CACHE`, 如 `SELECT SQL_NO_CACHE * FROM tbl_name`。当 `query_cache_type=2` 时, 只有在查询语句中使用 `SQL_CACHE` 关键字, 查询才会使用查询缓冲区。使用查询缓冲区可以提高查询的速度, 这种方式只适用于修改操作少且经常执行相同的查询操作的情况。
- `sort_buffer_size`: 表示排序缓存区的大小。这个值越大, 进行排序的速度越快。
- `read_buffer_size`: 表示每个线程连续扫描时为扫描的每个表分配的缓冲区的大小(字节)。当线程从表中连续读取记录时需要用到这个缓冲区。`SET SESSION read_buffer_size=n` 可以临时设置该参数的值。
- `read_rnd_buffer_size`: 表示为每个线程保留的缓冲区的大小, 与 `read_buffer_size` 相似。但主要用于存储按特定顺序读取出来的记录。也可以用 `SET SESSION read_rnd_buffer_size=n` 来临时设置该参数的值。如果频繁进行多次连续扫描, 可以增加该值。
- `innodb_buffer_pool_size`: 表示 InnoDB 类型的表和索引的最大缓存。这个值越大, 查询的速度就会越快。但是这个值太大会影响操作系统的性能。

- `max_connections`: 表示数据库的最大连接数。这个连接数不是越大越好, 因为这些连接会浪费内存的资源。过多的连接可能会导致 MySQL 服务器僵死。
- `innodb_flush_log_at_trx_commit`: 表示何时将缓冲区的数据写入日志文件, 并且将日志文件写入磁盘中。该参数对于 InnoDB 引擎非常重要。该参数有 3 个值, 分别为 0、1 和 2。值为 0 时表示每隔 1 秒将数据写入日志文件并将日志文件写入磁盘; 值为 1 时表示每次提交事务时将数据写入日志文件并将日志文件写入磁盘; 值为 2 时表示每次提交事务时将数据写入日志文件, 每隔 1 秒将日志文件写入磁盘。该参数的默认值为 1。默认值 1 安全性最高, 但是每次事务提交或事务外的指令都需要把日志写入 (flush) 硬盘, 是比较费时的; 0 值更快一点, 但安全方面比较差; 2 值日志仍然会每秒写入到硬盘, 所以即使出现故障, 一般也不会丢失超过 1~2 秒的更新。
- `back_log`: 表示在 mysql 暂时停止回答新请求之前的短时间内, 多少个请求可以被存在堆栈中。换句话说, 该值表示对到来的 Tcp/Ip 连接的侦听队列的大小。只有期望在一个短时间内有很多连接, 才需要增加该参数的值。操作系统在这个队列大小上也有限制。设定 `back_log` 高于操作系统的限制将是无效的。
- `interactive_timeout`: 表示服务器在关闭连接前等待行动的秒数。
- `sort_buffer_size`: 表示每个需要进行排序的线程分配的缓冲区的大小。增加这个参数的值可以提高 `ORDER BY` 或 `GROUP BY` 操作的速度。默认数值是 2 097 144 (2MB)。
- `thread_cache_size`: 表示可以复用的线程的数量。如果有很多新的线程, 为了提高性能可以增大该参数的值。
- `wait_timeout`: 表示服务器在关闭一个连接时等待行动的秒数。默认数值是 28 800。

要求: 必须记忆至少 3 个。

12 测试定位

高性能不是指"绝对性能"强悍,而是指业务能发挥出硬件的最大水平.性能强的服务器并非"设计"而来,而是不断改进,提升短板.测试,就是量化找出短板的过程.

12.1 测试指标

只有会测试,能把数据量化,才能进一步改进优化

- 1: 吞吐量:单位时间内的事务处理数,单位 tps(每秒事务数)
- 2: 响应时间:语句平均响应时间,一般截取某段时间内,95%范围内的平均时间
- 3: 并发性:线程同时执行
- 4: 可扩展性:资源增加,性能也能成正比增加

12.1 发现系统运行缓慢, 如何定位和分析查询慢的 sql 语句

- 1.开启 mysql 慢日志查询 定位查询较慢的 sql 语句 (200ms 500ms)
- 2.使用 EXPLAIN 关键字可以让你知道 MySQL 是如何处理你的 SQL 语句的。这可以帮你分析你的查询语句或是表结构的性能瓶颈。EXPLAIN 的查询结果还会告诉你你的索引主键被如何利用的, 你的数据表是如何被搜索和排序的.....等等, 等等
- 3.代码中 可以使用 AOP 的操作 对每个持久层的 service 方法 打印执行时间, 将所有执行时间较长的 sql 语句进行预警

12.2 测试工具

12.2.1mysqlslap

mysqlslap --options

--concurrency 代表并发数量, 多个可以用逗号隔开, concurrency=10,50,100, 并发连接线程数分别是 10、50、100 个并发。

--engines 代表要测试的引擎, 可以有多个, 用分隔符隔开。

--iterations 代表要运行这些测试多少次。

--auto-generate-sql 代表用系统自己生成的 SQL 脚本来测试。

--auto-generate-sql-load-type 代表要测试的是读还是写还是两者混合的 (read,write,update,mixed)

--number-of-queries 代表总共要运行多少次查询。每个客户运行的查询数量可以用查询总数/并发数来计算。

--debug-info 代表要额外输出 CPU 以及内存的相关信息。

```
mysqlslap -h 192.168.1.201 -uroot --auto-generate-sql --concurrency 20 --iterations 1 --create-schema=big_data  
--query='select * from dict limit 1'
```

12.2.2sysbench

测试 CPU 性能

测试 IO 性能

测试事务性能

12.3 查看 mysql 的进程状态

```
mysql -h 192.168.177.128 -u root -e 'show processlist' | grep State: | sort | uniq -c | sort -rn
```

5 State: Sending data

2 State: statistics

2 State: NULL

1 State: Updating

1 State: update

以下几种状态要注意:

converting HEAP to MyISAM 查询结果太大时,把结果放在磁盘

create tmp table 创建临时表(如 group 时储存中间结果)

Copying to tmp table on disk 把内存临时表复制到磁盘

locked 被其他查询锁住

logging slow query 记录慢查询