# 数据库基本概念

## ?数据库的基本组成

**什么情况下可以建立索引**，如果我的数据访问量都很大那怎么办

在访问量过大或是访问频繁或访问大表中一小部分的时候需要建立索引，以便提高效率可以对打表进行分区，减少I/O操作

Java的常用数据库对象包括？

DatabaseMetaData ResultSetMetaData

触发器（Trigger） [ˈtrɪgə(r)]、表（Table）、视图（View）、存储过程（StoredProcedure）、索引（Index）、缺省值（Default）、图表（Diagram）、用户（User）、规则（Rule）等几类。

## 索引

**索引的分类**

1.1、单列索引：一个索引只包含单个列，但一个表中可以有多个单列索引。 这里不要搞混淆了。

1.1.1、普通索引：MySQL中基本索引类型，没有什么限制，允许在定义索引的列中插入重复值和空值，纯粹为了查询数据更快一点。

1.1.2、唯一索引：索引列中的值必须是唯一的，但是允许为空值，

1.1.3、主键索引：是一种特殊的唯一索引，不允许有空值。（主键约束，就是一个主键索引）

组合索引

全文索引：全文索引，只有在MyISAM引擎上才能使用，只能在CHAR,VARCHAR,TEXT类型字段上使用全文索引

整个表变成了一个索引，也就是所谓的「聚集索引」

## SQL常用关键词和作用

create table view index procedure trigger select insert close

commit rollback set transaction declare explain open fetch prepare

## ？sql的行转列(PIVOT)与列转行(UNPIVOT)

## 关系型数据库为什么能活这么久？

**直观**：就是个表格；

**使用简单**：一般都只需要简单的SQL便可查出；

**对数据完整性的支持很好**；

**支持事务**；

**范式**：想要使用我们关系型数据库，必须得遵守一定的规则，这些规则就是“范式”。

第一范式是基本要求，即每个列都是不分割的数据项， 如果连这个都满足不了，还是洗洗睡吧。

第二范式要求实体属性要完全依赖主键，不能依赖部分主键。

第三范式就是一个表中不能包含其它表中已包含的非主关键字信息。不严谨地说就是这个表只包含其他表的ID。

做“数据的集成”

存在问题：高并发，大流量存在瓶颈。对分布式系统支持不好, 难于组成集群。

## 数据仓库

**数据仓库都有什么特点呢**

**1.主题性**

不同于传统数据库对应于某一个或多个项目，数据仓库根据使用者实际需求，将不同数据源的数据在一个较高的抽象层次上做整合，所有数据都围绕某一主题来组织。

这里的主题怎么来理解呢？比如对于滴滴出行，“司机行为分析”就是一个主题，对于链家网，“成交分析”就是一个主题。

**2.集成性**

数据仓库中存储的数据是来源于多个数据源的集成，原始数据来自不同的数据源，存储方式各不相同。要整合成为最终的数据集合，需要从数据源经过一系列抽取、清洗、转换的过程。

**3.稳定性**

数据仓库中保存的数据是一系列历史快照，不允许被修改。用户只能通过分析工具进行查询和分析。

**4.时变性**

数据仓库会定期接收新的集成数据，反应出最新的数据变化。这和特点并不矛盾。

## 什么是ETL？

ETL的英文全称是 Extract-Transform-Load 的缩写，用来描述将数据从来源迁移到目标的几个过程：

1.Extract，数据抽取，也就是把数据从数据源读出来。

2.Transform，数据转换，把原始数据转换成期望的格式和维度。如果用在数据仓库的场景下，Transform也包含数据清洗，清洗掉噪音数据。

3.Load 数据加载，把处理后的数据加载到目标处，比如数据仓库。

Hive是基于Hadoop的数据仓库工具，可以对存储在HDFS上的文件数据集进行查询和分析处理。

# 数据库锁

**首先对mysql锁进行划分：**

按照锁的粒度划分：行锁、表锁、页锁

按照锁的使用方式划分：共享锁、排它锁（悲观锁的一种实现）

还有两种思想上的锁：悲观锁、乐观锁。

InnoDB中有几种行级锁类型：Record Lock、Gap Lock、Next-key Lock

Record Lock：在索引记录上加锁

Gap Lock：间隙锁

Next-key Lock：Record Lock+Gap Lock

**1.行锁**

​行级锁是Mysql中锁定粒度最细的一种锁，表示只针对当前操作的行进行加锁。\*\*行级锁能大大减少数据库操作的冲突。其加锁粒度最小，但加锁的开销也最大。有可能会出现死锁的情况。\*\*行级锁按照使用方式分为共享锁和排他锁。

共享锁用法（S锁 读锁）：

若事务T对数据对象A加上S锁，则事务T可以读A但不能修改A，其他事务只能再对A加S锁，而不能加X锁，直到T释放A上的S锁。这保证了其他事务可以读A，但在T释放A上的S锁之前不能对A做任何修改。

select ... lock in share mode;

\*\*共享锁就是允许多个线程同时获取一个锁，一个锁可以同时被多个线程拥有。\*\*

排它锁用法（X 锁 写锁）：

​若事务T对数据对象A加上X锁，事务T可以读A也可以修改A，其他事务不能再对A加任何锁，直到T释放A上的锁。这保证了其他事务在T释放A上的锁之前不能再读取和修改A。

select ... for update

排它锁，也称作独占锁，一个锁在某一时刻只能被一个线程占有，其它线程必须等待锁被释放之后才可能获取到锁。

**2.表锁**

​表级锁是mysql锁中粒度最大的一种锁，表示当前的操作对整张表加锁，资源开销比行锁少，不会出现死锁的情况，但是发生锁冲突的概率很大。被大部分的mysql引擎支持，MyISAM和InnoDB都支持表级锁，但是InnoDB默认的是行级锁。

共享锁用法：

LOCK TABLE table\_name [ AS alias\_name ] READ

排它锁用法：

LOCK TABLE table\_name [AS alias\_name][ LOW\_PRIORITY ] WRITE

解锁用法：

unlock tables;

**3.页锁**

​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

**4.1悲观锁**

悲观锁，正如其名，它指的是对数据被外界（包括本系统当前的其他事务，以及来自外部系统的事务处理）修改持保守态度(悲观)，因此，在整个数据处理过程中，将数据处于锁定状态。 悲观锁的实现，往往依靠数据库提供的锁机制

悲观锁的具体流程：

在对任意记录进行修改前，先尝试为该记录加上排他锁（exclusive locking）

如果加锁失败，说明该记录正在被修改，那么当前查询可能要等待或者抛出异常。 具体响应方式由开发者根据实际需要决定。

如果成功加锁，那么就可以对记录做修改，事务完成后就会解锁了。

其间如果有其他对该记录做修改或加排他锁的操作，都会等待我们解锁或直接抛出异常。

在mysql/InnoDB中使用悲观锁：

首先我们得关闭mysql中的autocommit属性,begin;/begin work;/start transaction; (三者选一就可以)开始事务,commit;/commit work;提交事务

**4.2乐观锁**

相对于悲观锁，在对数据库进行处理的时候，乐观锁并不会使用数据库提供的锁机制。一般的实现乐观锁的方式就是记录数据版本。

实现数据版本有两种方式，第一种是使用版本号，第二种是使用时间戳。

乐观锁的优点和不足：

​ 乐观并发控制相信事务之间的数据竞争(data race)的概率是比较小的，因此尽可能直接做下去，直到提交的时候才去锁定，所以不会产生任何锁和死锁。但如果直接简单这么做，还是有可能会遇到不可预期的结果，例如两个事务都读取了数据库的某一行，经过修改以后写回数据库，这时就遇到了问题。

**5.MySQL/InnoDB中的行锁和表锁问题**

​首先我们知道InnoDB默认支持的是行锁，但这并不代表InnoDB不支持表锁。必须明白这一点在InnoDB中并不是在数据行上加锁，而是在对应的索引上加锁，这一点和oracle并不同，后者是在数据行上加锁的。这种实现的特点是：只有通过索引条件检索数据的时候加的是行锁，否则加表锁！

1.通过非索引项检索数据，加表锁！

mysql5.1InnoDB锁的特性

在不通过索引条件查询的时候，InnoDB使用的确实是表锁！

由于 MySQL 的行锁是针对索引加的锁,不是针对记录加的锁,所以虽然是访问不同行 的记录,但是如果是使用相同的索引键,是会出现锁冲突的。

当表有多个索引的时候,不同的事务可以使用不同的索引锁定不同的行,另外,不论 是使用主键索引、唯一索引或普通索引,InnoDB 都会使用行锁来对数据加锁。

即便在条件中使用了索引字段,但是否使用索引来检索数据是由 MySQL 通过判断不同 执行计划的代价来决定的,如果 MySQL 认为全表扫 效率更高,比如对一些很小的表,它 就不会使用索引,这种情况下 InnoDB 将使用表锁,而不是行锁（比如全表更新和多表查询时）。因此,在分析锁冲突时, 别忘了检查 SQL 的执行计划,以确认是否真正使用了索引。

2.使用相同索引值但是不同行引发的冲突

这个的主要原因还是由于Gap Lock

3.当使用索引检索数据时不同事务可以操作不同行数据

**表锁**

表锁的优势：开销小；加锁快；无死锁

表锁的劣势：锁粒度大，发生锁冲突的概率高，并发处理能力低

加锁的方式：自动加锁。查询操作（SELECT），会自动给涉及的所有表加读锁，更新操作（UPDATE、DELETE、INSERT），会自动给涉及的表加写锁。也可以显示加锁：

共享读锁：lock table tableName read;

独占写锁：lock table tableName write;

批量解锁：unlock tables;

MyISAM的表锁，读锁会阻塞写，不会阻塞读。而写锁则会把读写都阻塞。

**查看加锁情况**

show open tables; 1表示加锁，0表示未加锁。

分析表锁定

可以通过检查table\_locks\_waited 和 table\_locks\_immediate 状态变量分析系统上的表锁定：show status like ‘table\_locks%’

table\_locks\_immediate: 表示立即释放表锁数。

table\_locks\_waited: 表示需要等待的表锁数。此值越高则说明存在着越严重的表级锁争用情况。

此外，MyISAM的读写锁调度是写优先，这也是MyISAM不适合做写为主表的存储引擎。因为写锁后，其他线程不能做任何操作，大量的更新会使查询很难得到锁，从而造成永久阻塞。

**6.Record Lock、Gap Lock、Next-key Lock锁**

6.1.Record Lock

单条索引上加锁，record lock 永远锁的是索引，而非数据本身，如果innodb表中没有索引，那么会自动创建一个隐藏的聚集索引，锁住的就是这个聚集索引。所以说当一条sql没有走任何索引时，那么将会在每一条聚集索引后面加X锁，这个类似于表锁，但原理上和表锁应该是完全不同的。

**分析行锁定**

通过检查InnoDB\_row\_lock 状态变量分析系统上的行锁的争夺情况 show status like ‘innodb\_row\_lock%’

innodb\_row\_lock\_current\_waits: 当前正在等待锁定的数量

innodb\_row\_lock\_time: 从系统启动到现在锁定总时间长度；非常重要的参数

innodb\_row\_lock\_time\_avg: 每次等待所花平均时间；非常重要的参数，

innodb\_row\_lock\_time\_max: 从系统启动到现在等待最常的一次所花的时间；

innodb\_row\_lock\_waits: 系统启动后到现在总共等待的次数；非常重要的参数。直接决定优化的方向和策略。

**行锁优化**

1 尽可能让所有数据检索都通过索引来完成，避免无索引行或索引失效导致行锁升级为表锁。

2 尽可能避免间隙锁带来的性能下降，减少或使用合理的检索范围。

3 尽可能减少事务的粒度，比如控制事务大小，而从减少锁定资源量和时间长度，从而减少锁的竞争等，提供性能。

4 尽可能低级别事务隔离，隔离级别越高，并发的处理能力越低。

6.2.Gap Lock

间隙锁，是在索引的间隙之间加上锁，这是为什么Repeatable Read隔离级别下能防止幻读的主要原因。

幻读：事务A执行了一次读操作，此时事务B在事务A的影响区间内更新了一条数据，此时事务A在执行一次读操作时，会发现出现了不合理的数据。有关幻读的详细解释：https://blog.csdn.net/qq\_38238296/article/details/88363017

6.2.​1 什么叫间隙锁

​简单的说就是锁住了一个范围的数据

当我们用范围条件检索数据，并请求共享或排他锁时，InnoDB会给符合条件的已有数据记录的索引项加锁；对于键值在条件范围内但并不存在的记录，叫做”间隙(GAP)”。InnoDB也会对这个”间隙”加锁，这种锁机制就是所谓的间隙锁(Next-Key锁)。

危害(坑)：若执行的条件是范围过大，则InnoDB会将整个范围内所有的索引键值全部锁定，很容易对性能造成影响。

6.2.2 为什么说gap锁是RR隔离级别下防止幻读的主要原因。

快照读：简单的select操作，没有lock in share mode或for update

当前读：官方文档的术语叫locking read，也就是insert，update，delete,select…in share mode和select…for update，当前读会在所有扫描到的索引记录上加锁，不管它后面的where条件到底有没有命中对应的行记录。

首先了解到InnoDB索引的数据结构是B+树，其索引是有序性的，（具体原理可以看这篇文章：https://blog.csdn.net/qq\_38238296/article/details/88362635 ）如何保证两次当前读返回一致的记录，那就需要在第一次当前读与第二次当前读之间，其他的事务不会插入新的满足条件的记录并提交。注意是当前读。

根据索引的有序性，我们可以从上面的例子推断出满足where条件的数据，只能插入在num=（1,3]U[3,4)两个区间里面，只要我们将这两个区间锁住，那么就不会发生幻读。

6.3.Next-Key Lock

这个锁机制其实就是前面两个锁相结合的机制，行锁就是采用这个锁的机制用来防止幻读！

**7.死锁的原理及分析**

7.1. MVCC

​MySQL InnoDB存储引擎，实现的是基于多版本并发控制协议—MVCC(Multi-Version Concurrency Control) MVCC最大的好处，相信也是耳熟能详：读不加锁，读写不冲突。在读多写少的OLTP应用中，读写不冲突是非常重要的，极大的增加了系统的并发性能，这也是为什么现阶段，几乎所有的RDBMS，都支持了MVCC。

7.2. 2PL：Two-Phase Locking

​传统RDBMS（关系数据库管理系统）加锁的一个原则，就是2PL (二阶段锁)：Two-Phase Locking。相对而言，2PL比较容易理解，说的是锁操作分为两个阶段：加锁阶段与解锁阶段，并且保证加锁阶段与解锁阶段不相交。下面，仍旧以MySQL为例，来简单看看2PL在MySQL中的实现。

7.3 为什么会发生死锁

​MyISAM中是不会产生死锁的，因为MyISAM总是一次性获得所需的全部锁，要么全部满足，要么全部等待。而在InnoDB中，锁是逐步获得的，就造成了死锁的可能。（不过现在一般都是InnoDB引擎，关于MyISAM不做考虑）

在InnoDB中，行级锁并不是直接锁记录，而是锁索引。索引分为主键索引和非主键索引两种，如果一条sql语句操作了主键索引，MySQL就会锁定这条主键索引；如果一条语句操作了非主键索引，MySQL会先锁定该非主键索引，再锁定相关的主键索引。

当两个事务同时执行，一个锁住了主键索引，在等待其他相关索引。另一个锁定了非主键索引，在等待主键索引。这样就会发生死锁。

避免死锁，这里只介绍常见的三种

如果不同程序会并发存取多个表，尽量约定以相同的顺序访问表，可以大大降低死锁机会。

在同一个事务中，尽可能做到一次锁定所需要的所有资源，减少死锁产生概率；

对于非常容易产生死锁的业务部分，可以尝试使用升级锁定颗粒度，通过表级锁定来减少死锁产生的概率；

这篇文章关于mysql锁写的很有深度：http://hedengcheng.com/?p=771

# 事务

**事务的本质**

事务的核心是锁与并发

事务本质：独占某部分数据

事物单元之间的happen-before关系：读写 写读 读读 写写 只有这四种关系.

1、事务--排队法（一个队列） 优势：不需要冲突控制 劣势：慢速的设备。。。

2、事务--排他锁（多个队列）（Bob给Smith 100元；xiaoming给了xiaoyang100元同时进行）

3、事务--读写锁（对读读场景的优化）

4、事务--MVCC 本质来说就是copy on write

目前主流数据库能够做到写不阻塞读，能做到读读，读写，写读不冲突，只有写写控制不住。

基于数据版本记录机制，具体做法：数据库表增加一个"version"字段来实现，读取数据时，将版本号一同读出，之后更新，对版本号加1，将提交数据的版本数据与数据库对应记录的当前版本信息进行比对，如果提交的数据版本号大于数据库的数据，则予以更新，否则，被认为是过期数据。

**事务**：就是作为单个逻辑单元执行的一组操作，要么全成功，要么全失败。

分布式事务用于在分布式系统中保证不同节点之间的数据一致性。

事务常说一系列操作作为一个整体要么都成功要么都失败，主要特性acid，事务的的实现主要依赖两个log redo-log,undo-log,每次事务都会记录数据修改前的数据undo-log，修改后的数据放入redo-log,提出成功则使用redo-log 更新到磁盘，失败则使用undo-log将数据恢复到事务之前的数据

Java事务的类型有三种：JDBC事务、容器事务、JTA(Java Transaction API)事务。

不用事务的编程框架来管理事务，直接使用资源管理器来控制事务。典型的就是java.sql.Connection 中的 setAutoCommit、commit、rollback方法。本地事务比较简单，基本原理就是数据库的事务原理

**本地事务的优点**

支持严格的ACID属性

可靠

高效

状态可以只在资源管理器中维护

应用编程模型简单

**本地事务的局限**

不具备分布式事务处理能力

隔离的最小单位由资源管理器决定，如数据库中的一条记录

**全局事务的定义**：全局事务就是一个标准的分布式事务。全局事务是由资源管理器管理和协调的事务。

全局事务是一个DTP模型的事务，所谓DTP模型指的是X/Open DTP(X/Open Distributed Transaction Processing Reference Model)，是X/Open 这个组织定义的一套分布式事务的标准，也就是了定义了规范和API接口，由这个厂商进行具体的实现。

X/Open DTP 定义了三个组件：AP，TM，RM 和两个协议：XA、TX

AP(Application Program)：也就是应用程序，可以理解为使用DTP的程序

RM(Resource Manager)：资源管理器，这里可以理解为一个DBMS系统，或者消息服务器管理系统，应用程序通过资源管理器对资源进行控制。

TM(Transaction Manager)：事务管理器，负责协调和管理事务，提供给AP应用程序编程接口以及管理资源管理器。

XA协议：应用或应用服务器与事务管理之前通信的接口

TX协议：全局事务管理器与资源管理器之间通信的接口

事务管理器控制着全局事务，管理事务生命周期，并协调资源。资源管理器负责控制和管理实际资源

JTA(Java Transaction API):面向应用、应用服务器与资 源管理器的高层事务接口。

JTS(Java Transaction Service):JTA事务管理器的实现标 准,向上支持JTA,向下通过CORBA OTS实现跨事务域的互 操作性。

EJB:基于组件的应用编程模型,通过声明式事务管理进一步 简化事务应用的编程。

原文：https://mp.weixin.qq.com/s/QToXYvPzIBW36vAPqp6Ciw

## \*MySQL中事物ACID的原理

**Mysql怎么保证一致性的**

ACID嘛，原子性(Atomicity/,ætə'mɪsɪtɪ/)、一致性(Consistency/kən'sɪst(ə)nsɪ/)、隔离性(Isolation/aɪsə'leɪʃ(ə)n/)、持久性(Durability/djuərə'biləti/)

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。

**Mysql怎么保证原子性的**

OK，是利用Innodb的undo log

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。

undo log记录了这些回滚需要的信息，当事务执行失败或调用了rollback，导致事务需要回滚，便可以利用undo log中的信息将数据回滚到修改之前的样子。

**Mysql怎么保证持久性的**

OK，是利用Innodb的redo log。

正如之前说的，Mysql是先把磁盘上的数据加载到内存中，在内存中对数据进行修改，再刷回磁盘上。如果此时突然宕机，内存中的数据就会丢失。

怎么解决这个问题？简单啊，事务提交前直接把数据写入磁盘就行啊。但是这样太浪费资源了。于是，决定采用redo log解决上面的问题。当做数据修改的时候，不仅在内存中操作，还会在redo log中记录这次操作。当事务提交的时候，会将redo log日志进行刷盘(redo log一部分在内存中，一部分在磁盘上)。当数据库宕机重启的时候，会将redo log中的内容恢复到数据库中，再根据undo log和binlog内容决定回滚数据还是提交数据。

采用redo log的好处是进行刷盘比对数据页刷盘效率高，具体表现如下：redo log体积小，毕竟只记录了哪一页修改了啥，因此体积小，刷盘快。redo log是一直往末尾进行追加，属于顺序IO。效率显然比随机IO来的快。

**Mysql怎么保证隔离性的**

OK,利用的是锁和MVCC机制。

MVCC,即多版本并发控制(Multi Version Concurrency Control),一个行记录数据有多个版本对快照数据，这些快照数据在undo log中。

如果一个事务读取的行正在做DELELE或者UPDATE操作，读取操作不会等行上的锁释放，而是读取该行的快照版本。

由于MVCC机制在可重复读(Repeateable Read)和读已提交(Read Commited)的MVCC表现形式不同，就不赘述了。但是有一点说明一下，在事务隔离级别为读已提交(Read Commited)时，一个事务能够读到另一个事务已经提交的数据，是不满足隔离性的。但是当事务隔离级别为可重复读(Repeateable Read)中，是满足隔离性的。

## 事务隔离级别

**SQL 标准定义了四个隔离级别**：

•**READ-UNCOMMITTED(读取未提交)**： 最低的隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读

•**READ-COMMITTED(读取已提交):** 允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生

•**REPEATABLE-READ（可重读）**: 对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生。

•**SERIALIZABLE(可串行化)**: 最高的隔离级别，完全服从ACID的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，该级别可以防止脏读、不可重复读以及幻读。

**并发事务带来的问题**

•**脏读（Dirty read）**: 当一个事务正在访问数据并且对数据进行了修改，而这种修改还没有提交到数据库中，这时另外一个事务也访问了这个数据，然后使用了这个数据。因为这个数据是还没有提交的数据，那么另外一个事务读到的这个数据是“脏数据”，依据“脏数据”所做的操作可能是不正确的。

•**丢失修改（Lost to modify）**: 指在一个事务读取一个数据时，另外一个事务也访问了该数据，那么在第一个事务中修改了这个数据后，第二个事务也修改了这个数据。这样第一个事务内的修改结果就被丢失，因此称为丢失修改。 例如：事务1读取某表中的数据A=20，事务2也读取A=20，事务1修改A=A-1，事务2也修改A=A-1，最终结果A=19，事务1的修改被丢失。

•**不可重复读（Unrepeatableread）**: 指在一个事务内多次读同一数据。在这个事务还没有结束时，另一个事务也访问该数据。那么，在第一个事务中的两次读数据之间，由于第二个事务的修改导致第一个事务两次读取的数据可能不太一样。这就发生了在一个事务内两次读到的数据是不一样的情况，因此称为不可重复读。

•**幻读（Phantom read）**: 幻读与不可重复读类似。它发生在一个事务（T1）读取了几行数据，接着另一个并发事务（T2）插入了一些数据时。在随后的查询中，第一个事务（T1）就会发现多了一些原本不存在的记录，就好像发生了幻觉一样，所以称为幻读。

不可重复度和幻读区别：

不可重复读的重点是修改，幻读的重点在于新增或者删除。

例1（同样的条件, 你读取过的数据, 再次读取出来发现值不一样了 ）：事务1中的A先生读取自己的工资为 1000的操作还没完成，事务2中的B先生就修改了A的工资为2000，导 致A再读自己的工资时工资变为 2000；这就是不可重复读。

例2（同样的条件, 第1次和第2次读出来的记录数不一样 ）：假某工资单表中工资大于3000的有4人，事务1读取了所有工资大于3000的人，共查到4条记录，这时事务2 又插入了一条工资大于3000的记录，事务1再次读取时查到的记录就变为了5条，这样就导致了幻读。

MySQL InnoDB 存储引擎的默认支持的隔离级别是 REPEATABLE-READ（可重读）。我们可以通过SELECT @@tx\_isolation;命令来查看

# 分库分表技术演进&最佳实践

## 如何存储海量数据？目前比较普遍的方案有3个

分区；

分库分表；

NoSQL/NewSQL；

不使用NoSQL/NewSQL的原因：不成熟

不使用分区的原因：很多的资源都受到单机的限制，例如连接数，网络吞吐等！虽然每个分区可以独立存储，但是分区表的总入口还是一个MySQL示例。从而导致它的并发能力非常一般，远远达不到互联网高并发的要求！

NoSQL比较具有代表性的是MongoDB，es。NewSQL比较具有代表性的是TiDB。

## 分库分表中间件

阿里的TDDL，DRDS和cobar，

开源社区的sharding-jdbc（3.x已经更名为sharding-sphere）；

民间组织的MyCAT；

360的Atlas；

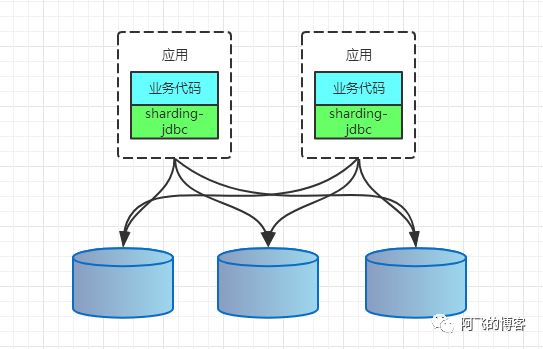
美团的zebra；

## 分库分表中间件全部可以归结为两大类型

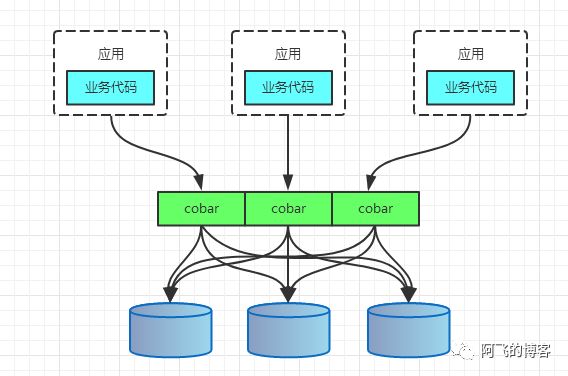
CLIENT模式；

PROXY模式；

CLIENT模式代表有阿里的TDDL，开源社区的sharding-jdbc（sharding-jdbc的3.x版本即sharding-sphere已经支持了proxy模式）。架构如下



PROXY模式代表有阿里的cobar，民间组织的MyCAT。架构如下：



但是，无论是CLIENT模式，还是PROXY模式。几个核心的步骤是一样的：SQL解析，重写，路由，执行，结果归并。

## 实战案例

分库分表第一步也是最重要的一步，即sharding column的选取，sharding column选择的好坏将直接决定整个分库分表方案最终是否成功。选择sharding column的方法最主要分析你的API流量，优先考虑流量大的API，将流量比较大的API对应的SQL提取出来，将这些SQL共同的条件作为sharding column。

这里列举分库分表的几种主要处理思路：

只选取一个sharding column进行分库分表 ；

多个sharding column多个分库分表；

sharding column分库分表 + es(es+HBase)；

# 优化

## \*优化现有MySQL数据库

**1.数据库设计和表创建时就要考虑性能**

简言之就是使用合适的数据类型,选择合适的索引

设计表时要注意的东西：表字段避免null值出现，null值很难查询优化且占用额外的索引空间；尽量使用INT而非BIGINT；使用枚举或整数代替字符串类型；单表不要有太多字段；

索引：要根据查询有针对性的创建，考虑在WHERE和ORDERBY命令上涉及的列建立索引；应尽量避免在WHERE子句中对字段进行NULL值判断；字符字段只建前缀索引；字符字段最好不要做主键；不用外键，由程序保证约束；使用多列索引时主意顺序和查询条件保持一致，同时删除不必要的单列索引；

**2.SQL的编写需要注意优化**

使用limit对查询结果的记录进行限定

避免select \*，将需要查找的字段列出来

使用连接（join）来代替子查询

拆分大的delete或insert语句

可通过开启慢查询日志来找出较慢的SQL

不做列运算：SELECT id WHERE age + 1 = 10，任何对列的操作都将导致表扫描，它包括数据库教程函数、计算表达式等等，查询时要尽可能将操作移至等号右边

SQL语句尽可能简单：一条SQL只能在一个cpu运算；大语句拆小语句，减少锁时间；一条大SQL可以堵死整个库

OR改写成IN：OR的效率是n级别，IN的效率是log(n)级别，in的个数建议控制在200以内

不用函数和触发器，在应用程序实现

避免%xxx式查询

少用JOIN

使用同类型进行比较，比如用'123'和'123'比，123和123比

尽量避免在WHERE子句中使用!=或<>操作符，否则将引擎放弃使用索引而进行全表扫描

对于连续数值，使用BETWEEN不用IN：SELECT id FROM t WHERE num BETWEEN 1 AND 5

列表数据不要拿全表，要使用LIMIT来分页，每页数量也不要太大

**3.分区**

MySQL在5.1版引入的分区是一种简单的水平拆分，用户需要在建表的时候加上分区参数，对应用是透明的无需修改代码

MySQL实现分区的方式也意味着索引也是按照分区的子表定义，没有全局索引

用户的SQL语句是需要针对分区表做优化，SQL条件中要带上分区条件的列，从而使查询定位到少量的分区上，否则就会扫描全部分区，可以通过EXPLAINPARTITIONS来查看某条SQL语句会落在那些分区上，从而进行SQL优化

分区的好处：

可以让单表存储更多的数据

分区表的数据更容易维护，可以通过清楚整个分区批量删除大量数据，也可以增加新的分区来支持新插入的数据。另外，还可以对一个独立分区进行优化、检查、修复等操作

部分查询能够从查询条件确定只落在少数分区上，速度会很快

分区表的数据还可以分布在不同的物理设备上，从而搞笑利用多个硬件设备

可以使用分区表赖避免某些特殊瓶颈，例如InnoDB单个索引的互斥访问、ext3文件系统的inode锁竞争

可以备份和恢复单个分区

分区的限制和缺点

一个表最多只能有1024个分区

如果分区字段中有主键或者唯一索引的列，那么所有主键列和唯一索引列都必须包含进来

分区表无法使用外键约束

NULL值会使分区过滤无效

所有分区必须使用相同的存储引擎

**4.分表**

分表分为垂直拆分和水平拆分，通常以某个字段做拆分项。

**5.分库**

把一个数据库分成多个，建议做个读写分离就行了，真正的做分库也会带来大量的开发成本

## \*Sql优化

我觉得，sql优化都应该遵循一条主线，就是避免全表扫描，并且将检索的行数降到最低。

怎么将检索的行数降到最低呢，

单表查询的时候，没什么好说的，加索引，但也要看一下表数据再加，有些列的重复数据太多了，加普通索引的话基本上没什么效果，必要时可以用一下联合索引。

多表查询的时候就要看情况了，我是一般遵循尽量早过滤的原则，比如一个表join另一个表，一般都是先join然后再用条件过滤出想要的数据，这个在表数量小的时候没什么问题，数据量大了就会很慢，这时可以先用子查询查询出符合的数据再做关联。当然，这也是基于索引上的，总结一点，其实sql优化也就是让查询最大限度的使用索引。

from后面的表关联，是自右向左解析的 ，而where条件的解析顺序是自下而上的。

些可以过滤掉最大数量记录的条件必须写在WHERE子句的末尾。

ORACLE 的解析器按照从右到左的顺序处理FROM子句中的表名，FROM子句中写在最后的表将被最先处理，这时就可以将记录最少的表放在最右边。还有就是表之间的连接条件必须写在其它Where条件之前，这样

1. 在表中建立索引，优先考虑where、group by使用到的字段。
2. 尽量避免使用select \*，返回无用的字段会降低查询效率。
3. 尽量避免使用in 和not in，会导致数据库引擎放弃索引进行全表扫描。
4. 尽量避免使用or，会导致数据库引擎放弃索引进行全表扫描。
5. 尽量避免在where条件中等号的左侧进行表达式、函数操作，会导致数据库引擎放弃索引进行全表扫描。
6. 尽量避免进行null值的判断，会导致数据库引擎放弃索引进行全表扫描。
7. 尽量避免在字段开头模糊查询，会导致数据库引擎放弃索引进行全表扫描。

Oracle的SQL语句优化方法：

**1.SQL语句尽量用大写的；**

**2.使用表的别名：**

**3.选择最有效率的表名顺序**(**只在基于规则的优化器(RBO)中有效**)：ORACLE 的解析器按照从右到左的顺序处理FROM子句中的表名，FROM子句中写在最后的表(基础表也称为驱动表,driving table)将被最先处理

**4.WHERE子句中的连接顺序：**ORACLE采用**自下而上的顺序解析**WHERE子句,根据这个原理,表之间的连接必须写在其他WHERE条件之前, 那些可以过滤掉最大数量记录的条件必须写在WHERE子句的末尾。

## ****\*索引实现原理****

[**数据库**](http://lib.csdn.net/base/14)**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B树及其变种B+树**。

在数据之外，数据库系统还维护着满足特定查找算法的数据结构，这些数据结构以某种方式引用（指向）数据，这样就可以在这些数据结构上实现高级查找算法。这种数据结构，就是索引。

## ?\*高并发下数据库设计

## 删除重复记录

（1）通过建立临时表来实现

SQL>create table temp\_emp as (select distinct \* from employee)

SQL> truncate table employee; (清空employee表的数据）

SQL> insert into employee select \* from temp\_emp; (再将临时表里的内容插回来）

( 2）通过唯一rowid实现删除重复记录.在Oracle中，每一条记录都有一个rowid，rowid在整个数据库中是唯一的，rowid确定了每条记录是在Oracle中的哪一个数据文件、块、行上。在重复的记录中，可能所有列的内容都相同，但rowid不会相同，所以只要确定出重复记录中那些具有最大或最小rowid的就可以了，其余全部删除。

SQL>delete from employee e2 where rowid not in (

select max(e1.rowid) from employee e1 where

e1.emp\_id=e2.emp\_id and e1.emp\_name=e2.emp\_name and e1.salary=e2.salary);--这里用min(rowid)也可以。

SQL>delete from employee e2 where rowid <(

select max(e1.rowid) from employee e1 where

e1.emp\_id=e2.emp\_id and e1.emp\_name=e2.emp\_name and

e1.salary=e2.salary);

（3）也是通过rowid，但效率更高。

SQL>delete from employee where rowid not in (

select max(t1.rowid) from employee t1 group by

t1.emp\_id,t1.emp\_name,t1.salary);--这里用min(rowid)也可以。

EMP\_ID EMP\_NAME SALARY

mysql删除重复数据

DELETE FROM USER WHERE id NOT IN (SELECT a.id FROM (SELECT MAX(id) id FROM USER GROUP BY age) a)

## 其它