索引

包括分类及优化方式，失效条件，底层结构

分类

单列索引，即一个索引只包含单个列，一个表可以有多个单列索引，但这不是组合索引。
组合索引，即一个索包含多个列。

- **普通索引**：这是最基本的索引，它没有任何限制。

```
CREATE INDEX indexName ON mytable(username(length));

-- 修改表结构
ALTER mytable ADD INDEX [indexName] ON (username(length))

-- 创建表的时候直接指定
CREATE TABLE mytable(
    ID INT NOT NULL,
    username VARCHAR(16) NOT NULL,
    INDEX [indexName] (username(length))
    );

-- 删除索引的语法：
DROP INDEX [indexName] ON mytable;
```

- **唯一索引**：

  它与前面的普通索引类似，不同的就是：索引列的值必须唯一，但允许有空值。如果是组合索引，则列值的组合必须唯一。它有以下几种创建方式：

```
CREATE UNIQUE INDEX indexName ON mytable(username(length))

-- 修改表结构
ALTER mytable ADD UNIQUE [indexName] ON (username(length))

-- 创建表的时候直接指定
CREATE TABLE mytable(
    ID INT NOT NULL,
    username VARCHAR(16) NOT NULL,
    UNIQUE [indexName] (username(length))
    );
```

- **主键索引**：

  它是一种特殊的唯一索引，不允许有空值。一般是在建表的时候同时创建主键索引：

```
CREATE TABLE mytable(
   ID INT NOT NULL,
   username VARCHAR(16) NOT NULL,
   PRIMARY KEY(ID)
   );

-- 修改表结构的时候（删除 & 添加）
alter table table_test drop primary key;
alter table table_test add primary key(ID);
```

- **组合索引**：

  为了进一步榨取MySQL的效率，就要考虑建立组合索引。就是将 name, city, age建到一个索引里：

```
ALTER TABLE mytable ADD INDEX name_city_age (name(10),city,age);

建立这样的组合索引，其实是相当于分别建立了下面三组组合索引：
· usernname,city,age
· usernname,city
· usernname
为什么没有 city，age这样的组合索引呢？这是因为MySQL组合索引“最左前缀”的结果。
简单的理解就是只从最左面的开始组合。

并不是只要包含这三列的查询都会用到该组合索引，下面的几个SQL就会用到这个组合索引：
SELECT * FROM mytable WHREE username="admin" AND city="郑州"
SELECT * FROM mytable WHREE username="admin"

而下面几个则不会用到：
SELECT * FROM mytable WHREE age=20 AND city="郑州"
SELECT * FROM mytable WHREE city="郑州"
```

索引的不足之处

1.虽然索引大大提高了查询速度，同时却会降低更新表的速度，如对表进行INSERT、UPDATE和DELETE。因为更新表时，MySQL不仅要保存数据，还要保存一下索引文件。

2.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。

使用索引的注意事项

1.索引不会包含有NULL值的列

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，那么这一列对于此复合索引就是无效的。所以我们在数据库设计时不要让字段的默认值为NULL。

2.使用短索引
对串列进行索引，如果可能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，多数值是惟一的，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。

3.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创建复合索引。

4.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，如何使用也是一个问题。like “%aaa%” 不会使用索引而like “aaa%”可以使用索引。

失效条件

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)

> 注意：要想使用or，又想让索引生效，只能将or条件中的每个列都加上索引

2.对于多列索引，不是使用的第一部分，则不会使用索引

3.like查询是以%开头

4.如果列类型是字符串，那一定要在条件中将数据使用引号引用起来,否则不使用索引

5.如果mysql估计使用全表扫描要比使用索引快,则不使用索引

6.对索引列进行运算.需要建立函数索引.

7.随着表的增长，where条件出来的数据太多，大于15%，使得索引失效
（会导致CBO计算走索引花费大于走全表）

8.没有查询条件，或者查询条件没有建立索引

优化

总纲只有一句话：建立必要的索引。

判断的最终标准是看这些索引是否对我们的数据库性能有所帮助。

1、表的主键、外键必须有索引；

2、数据量超过300的表应该有索引；

3、经常与其他表进行连接的表，在连接字段上应该建立索引；

4、经常出现在Where子句中的字段，特别是大表的字段，应该建立索引；

5、索引应该建在选择性高的字段上；

6、索引应该建在小字段上，对于大的文本字段甚至超长字段，不要建索引；

7、复合索引的建立需要进行仔细分析；尽量考虑用单字段索引代替：

```
A、正确选择复合索引中的主列字段，一般是选择性较好的字段；
B、复合索引的几个字段是否经常同时以AND方式出现在Where子句中？单字段查询是否极少甚至没有？
    如果是，则可以建立复合索引；否则考虑单字段索引；
C、如果复合索引中包含的字段经常单独出现在Where子句中，则分解为多个单字段索引；
D、如果复合索引所包含的字段超过3个，那么仔细考虑其必要性，考虑减少复合的字段；
E、如果既有单字段索引，又有这几个字段上的复合索引，一般可以删除复合索引；
```

8、频繁进行数据操作的表，不要建立太多的索引；

9、删除无用的索引，避免对执行计划造成负面影响；

- 避免对列的操作

```
select * from record where amount/30< 1000
改成 --->
select * from record where amount < 1000*30 (< 1秒)
```

- 避免不必要的类型转换

```
select col1,col2 from tab1 where col1>10
改成 --->
select col1,col2 from tab1 where col1>'10'
```

- 增加查询的范围限制，避免全范围的搜索。

```
select * from record where ActionTime < to_date ('20010301' ,'yyyymm')
改成 --->
select * from record where
ActionTime < to_date ('20010301' ,'yyyymm')
and
ActionTime > to_date ('20010101' ,'yyyymm')
```

- 尽量去掉"IN"、"OR"

- 尽量去掉 “<>”

- 去掉Where子句中的IS NULL和IS NOT NULL

…

SQL语法

```
join，union，子查询，having，group by，order by
```

join

```
参考资料：http://blog.csdn.net/xiao2004/article/details/6562435
用文氏图来解释：内连接、外连接、左连接、交叉集。
```

内连接：SELECT * FROM **TableA INNER JOIN TableB** ON TableA.name = TableB.name
外连接：SELECT * FROM **TableA FULL OUTER JOIN TableB** ON TableA.name = TableB.name
左连接：SELECT * FROM **TableA LEFT OUTER JOIN TableB** ON TableA.name = TableB.name
左连接（取部分）：
SELECT * FROM **TableA LEFT OUTER JOIN TableB** ON TableA.name = TableB.name WHERE TableB.id IS null

交叉集（笛卡尔积）： SELECT * FROM TableA CROSS JOIN TableB

union

UNION 操作符用于合并两个或多个 SELECT 语句的结果集。

```
请注意，UNION 内部的 SELECT 语句必须拥有相同数量的列。列也必须拥有相似的数据类型。
同时，每条 SELECT 语句中的列的顺序必须相同。
```

SELECT column*name(s) FROM table*name1
UNION
SELECT column*name(s) FROM table*name2

注释：默认地，UNION 操作符选取不同的值。如果允许重复的值，请使用 UNION ALL。

SELECT column*name(s) FROM table*name1
UNION ALL
SELECT column*name(s) FROM table*name2

注释：另外，UNION 结果集中的列名总是等于 UNION 中第一个 SELECT 语句中的列名。

子查询

分类：

单行子查询： 子查询的结果集是一行

多行子查询： 子查询的结果集是多行

```
注意事项：
子查询要包含在括号内。
将子查询放在比较条件的右侧。
单行操作符对应单行子查询，多行操作符对应多行子查询。

单行操作符：
  > < >= <= = <>
多行操作符：
  in(重点)、any（任意一个）、all（所有的）、not in
```

单行子查询：比较的对象是唯一的一个而不是一个范围集合

1.

```
SELECT *
FROM employees
WHERE salary > (
    SELECT salary
    FROM employees
    WHERE last_name = 'Abel'
);
```

2.

```
SELECT last_name 姓名,job_id ,salary 工资
FROM employees
WHERE job_id =(
   SELECT job_id
    FROM employees
    WHERE employee_id = 141
) AND salary>(
    SELECT salary
    FROM employees
    WHERE employee_id = 143
)
```

3.用上组函数

```
SELECT last_name,job_id,salary
```

```
FROM employees
WHERE salary=(
    //把如下语句获取的值作为salary比较的条件对象
    SELECT MIN(salary) FROM employees
);
```

4.用上having

```
SELECT MIN(salary) ,department_id
FROM employees
GROUP BY department_id
HAVING MIN(salary)>(
    SELECT MIN(salary)
    FROM employees
    WHERE department_id=50
);
```

非法使用单行子查询，其实就是 子查询结果集中返回的是多行
单行子查询的结果集中出现 null的问题
单行子查询的结果集多列的问题

多行子查询

```
SELECT employee_id,last_name,job_id,salary
    FROM employees
    WHERE salary<ANY(
        SELECT salary
        FROM employees
        WHERE job_id = 'IT_PROG'
    )

SELECT employee_id,last_name,job_id,salary
    FROM employees
    WHERE salary<ALL(
        SELECT salary
        FROM employees
        WHERE job_id = 'IT_PROG'
    )

SELECT employee_id,last_name,job_id,salary
    FROM employees
    WHERE salary NOT IN(
        SELECT salary
        FROM employees
```

*WHERE job*id = ‘IT_PROG’
)

having

在 SQL 中增加 HAVING 子句原因是，WHERE 关键字无法与合计函数一起使用。

1.

SELECT Customer,SUM(OrderPrice) FROM Orders
GROUP BY Customer
HAVING **SUM(OrderPrice)**<2000

2.

SELECT Customer,SUM(OrderPrice) FROM Orders
WHERE Customer=‘Bush’ OR Customer=‘Adams’
GROUP BY Customer
HAVING **SUM(OrderPrice)**>1500

group by

SELECT Customer, **SUM(OrderPrice)** FROM Orders
GROUP BY **Customer**

省略group by：
SELECT Customer,SUM(OrderPrice) FROM Orders
那么会返回，列的总计。

我们也可以对一个以上的列应用 GROUP BY 语句，就像这样：

SELECT Customer,OrderDate,SUM(OrderPrice) FROM Orders
GROUP BY Customer,OrderDate

返回的就是，某个客户在某个日期的订单总和。

order by

ORDER BY 语句用于根据指定的列对结果集进行排序。
ORDER BY 语句默认按照升序对记录进行排序。
如果您希望按照降序对记录进行排序，可以使用 DESC 关键字。（升序 ASC）

SELECT Company, OrderNumber FROM Orders **ORDER BY** Company

SELECT Company, OrderNumber FROM Orders ORDER BY **Company, OrderNumber**

SELECT Company, OrderNumber FROM Orders ORDER BY Company **DESC**

SELECT Company, OrderNumber FROM Orders ORDER BY **Company DESC, OrderNumber ASC**

引擎对比

```
InnoDB, MyISAM
```

MySQL数据库引擎取决于MySQL在安装的时候是如何被编译的。MySQL默认采用的是MyISAM。

1. MyISAM类型不支持事务处理等高级处理，而InnoDB类型支持。
2. MyISAM类型的表强调的是性能，其执行速度比InnoDB类型更快，但是不提供事务支持，而InnoDB提供事务支持以及外部键等高级数据库功能。
3. InnoDB不支持FULLTEXT类型的索引。全文索引是指对char、varchar和text中的每个词（停用词除外）建立倒排序索引。MyISAM的全文索引其实没啥用，因为它不支持中文分词，必须由使用者分词后加入空格再写到数据表里，而且少于4个汉字的词会和停用词一样被忽略掉。
4. 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。

没有where的count(*)使用MyISAM要比InnoDB快得多。因为MyISAM内置了一个计数器，count(*)时它直接从计数器中读，而InnoDB必须扫描全表。

1. InnoDB支持数据行锁定，MyISAM不支持行锁定，只支持锁定整个表。
2. InnoDB支持外键，MyISAM不支持。
3. InnoDB的主键范围更大，最大是MyISAM的2倍。
4. InnoDB是聚集索引，数据文件是和索引绑在一起的，必须要有主键，通过主键索引效率很高。

构成上的区别：

每个MyISAM在磁盘上存储成三个文件。第一个文件的名字以表的名字开始，扩展名指出文件类型。
.frm文件存储表定义。

数据文件的扩展名为.MYD (MYData)。

索引文件的扩展名是.MYI (MYIndex)。

基于磁盘的资源是InnoDB表空间数据文件和它的日志文件，InnoDB 表的大小只受限于操作系统文件的大小，一般为 2GB

SELECT UPDATE,INSERT，Delete操作

如果执行大量的SELECT，MyISAM是更好的选择,查询效率高一点。

1.如果你的数据执行大量的INSERT或UPDATE，出于性能方面的考虑，应该使用InnoDB表
2.DELETE FROM table时，InnoDB不会重新建立表，而是一行一行的删除。
3.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

精简

1、myisam查询效率更高，支持全文索引。innodb不支持全文索引，查询效率差myisam6-7倍。
2、innodb支持事务，行锁，外键。myisam不支持。

数据库的锁

```
行锁，表锁，页级锁，意向锁，读锁，写锁，悲观锁，乐观锁，以及加锁的select sql方式
```

数据的锁主要用来保证数据的一致性的。

从并发事务锁定的关系上看：共享锁、独占锁

- **共享锁**用于读取数据操作，它是非独占的，允许其他事务同时读取其锁定的资源，但不允许其他事务更新它。**（读锁）**
- **独占锁**也叫**排他锁**，适用于修改数据的场合。它所锁定的资源，其他事务不能读取也不能修改。**（写锁）**

读锁和写锁都是阻塞锁。

共享锁 (S) (S) (S) (S) (S) (S) (S)

1、加锁的条件：当一个事务执行select语句时，数据库系统会为这个事务分配一把共享锁，来锁定被查询的数据。

2、解锁的条件：在默认情况下，数据被读取后，数据库系统立即解除共享锁。例如，当一个事务执行查询“SELECT * FROM accounts”语句时，数据库系统首先锁定第一行，读取之后，解除对第一行的锁定，然后锁定第二行。这样，在一个事务读操作过程中，允许其他事务同时更新accounts表中未锁定的行。

3、与其他锁的兼容性：如果数据资源上放置了共享锁，还能再放置共享锁和更新锁。

4、并发性能：具有良好的并发性能，当数据被放置共享锁后，还可以再放置共享锁或更新锁。所以并发性能很好。

独占锁 （X）(X)（X）(X)（X）(X)（X）(X)

1、加锁的条件：当一个事务执行insert、update或delete语句时，数据库系统会自动对SQL语句操纵的数据资源使用独占锁。如果该数据资源已经有其他锁（任何锁）存在时，就无法对其再放置独占锁了。

2、解锁的条件：独占锁需要等到事务结束才能被解除。

3、兼容性：独占锁不能和其他锁兼容，如果数据资源上已经加了独占锁，就不能再放置其他的锁了。同样，如果数据资源上已经放置了其他锁，那么也就不能再放置独占锁了。

4、并发性能：不用说了，最差。只允许一个事务访问锁定的数据，如果其他事务也需要访问该数据，就必须等待，起到前一个事务结束，解除了独占锁，其他事务才有机会访问该数据。

行锁、表锁、页级锁

页级：引擎 BDB。
表级：引擎 MyISAM，理解为锁住整个表，可以同时读，写不行。
行级：引擎 INNODB，单独的一行记录加锁。

表级：直接锁定整张表，在你锁定期间，其它进程无法对该表进行写操作。如果你是写锁，则其它进程则读也不允许

> 开销小，加锁快；不会出现死锁；锁定粒度大，发出锁冲突的概率最高，并发度最低。

行级：仅对指定的记录进行加锁，这样其它进程还是可以对同一个表中的其它记录进行操作。

> 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。

页级：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。

> 开销和加锁时间界于表锁和行锁之间；会出现死锁；锁定粒度界于表锁和行锁之间，并发度一般

注意：

- InnoDB行锁是通过给索引上的索引项加锁来实现的，InnoDB这种行锁实现特点意味着：只有通过索引条件检索数据，InnoDB才使用行级锁，否则，InnoDB将使用表锁！
- 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。
- 在MySQL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。**在UPDATE、DELETE操作时，MySQL不仅锁定WHERE条件扫描过的所有索引记录，而且会锁定相邻的键值，即所谓的next-key locking。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，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。

1.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。
2.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；
3.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；

加锁/解锁方式

MyISAM 在执行查询语句(SELECT)前,会自动给涉及的所有表加读锁,在执行更新操作 (UPDATE、DELETE、INSERT 等)前，会自动给涉及的表加写锁，这个过程并不需要用户干预，因此，用户一般不需要直接用**LOCK TABLE**命令给MyISAM表显式加锁。

锁定表：LOCK TABLES tbl*name {READ | WRITE},[tbl*name {READ | WRITE},…]
解锁表：UNLOCK TABLES

在用 LOCK TABLES 给表显式加表锁时,必须同时取得所有涉及到表的锁。
在执行 LOCK TABLES 后，只能访问显式加锁的这些表，不能访问未加锁的表;

如果加的是读锁，那么只能执行查询操作，而不能执行更新操作。
在自动加锁的情况下也基本如此，MyISAM 总是一次获得 SQL 语句所需要的全部锁。这也正是 MyISAM 表不会出现死锁(Deadlock Free)的原因。

MyISAM的锁调度

前面讲过,MyISAM 存储引擎的读锁和写锁是互斥的,读写操作是串行的。那么,一个进程请求某个 MyISAM 表的读锁,同时另一个进程也请求同一表的写锁,MySQL 如何处理呢?

答案是写进程先获得锁。

不仅如此,即使读请求先到锁等待队列,写请求后到,写锁也会插到读锁请求之前!这是因为 MySQL 认为写请求一般比读请求要重要。这也正是 MyISAM 表不太适合于有大量更新操作和查询操作应用的原因,因为,大量的更新操作会造成查询操作很难获得读锁,从而可能永远阻塞。这种情况有时可能会变得非常糟糕!

```
通过指定启动参数low-priority-updates,使MyISAM引擎默认给予读请求以优先的权利。

通过执行命令SET LOW_PRIORITY_UPDATES=1,使该连接发出的更新请求优先级降低。

通过指定INSERT、UPDATE、DELETE语句的LOW_PRIORITY属性,降低该语句的优先级。
```

另外,MySQL也 供了一种折中的办法来调节读写冲突,即给系统参数max*write*lock_count 设置一个合适的值,当一个表的读锁达到这个值后,MySQL就暂时将写请求的优先级降低, 给读进程一定获得锁的机会。

意向锁 (IX/IS)(IX/IS)(IX/IS)(IX/IS)(IX/IS)(IX/IS)

意向锁是**表级锁**。不会和行级的X，S锁发生冲突。只会和表级的X，S发生冲突
意向锁是在添加行锁之前添加。
如果没有意向锁的话，则需要遍历所有整个表判断是否有行锁的存在，以免发生冲突
如果有了意向锁，只需要判断该意向锁与即将添加的表级锁是否兼容即可。因为意向锁的存在代表了，有行

级锁的存在或者即将有行级锁的存在。因而无需遍历整个表，即可获取结果

这两中类型的锁共存的问题考虑这个例子：

事务A锁住了表中的一行，让这一行只能读，不能写。
之后，事务B申请整个表的写锁。
如果事务B申请成功，那么理论上它就能修改表中的任意一行，这与A持有的行锁是冲突的。

数据库需要避免这种冲突，就是说要让B的申请被阻塞，直到A释放了行锁。数据库要怎么判断这个冲突呢?
step1：判断表是否已被其他事务用表锁锁表
step2：判断表中的每一行是否已被行锁锁住。

注意step2，这样的判断方法效率实在不高，因为需要遍历整个表。

于是就有了**意向锁**。在意向锁存在的情况下，事务A必须先申请表的意向共享锁，成功后再申请一行的行锁。在意向锁存在的情况下，上面的判断可以改成
step1：不变
step2：发现表上有意向共享锁，说明表中有些行被共享行锁锁住了，因此，事务B申请表的写锁会被阻塞。

注意：申请意向锁的动作是数据库完成的，就是说，事务A申请一行的行锁的时候，数据库会自动先开始申请表的意向锁，不需要我们程序员使用代码来申请。

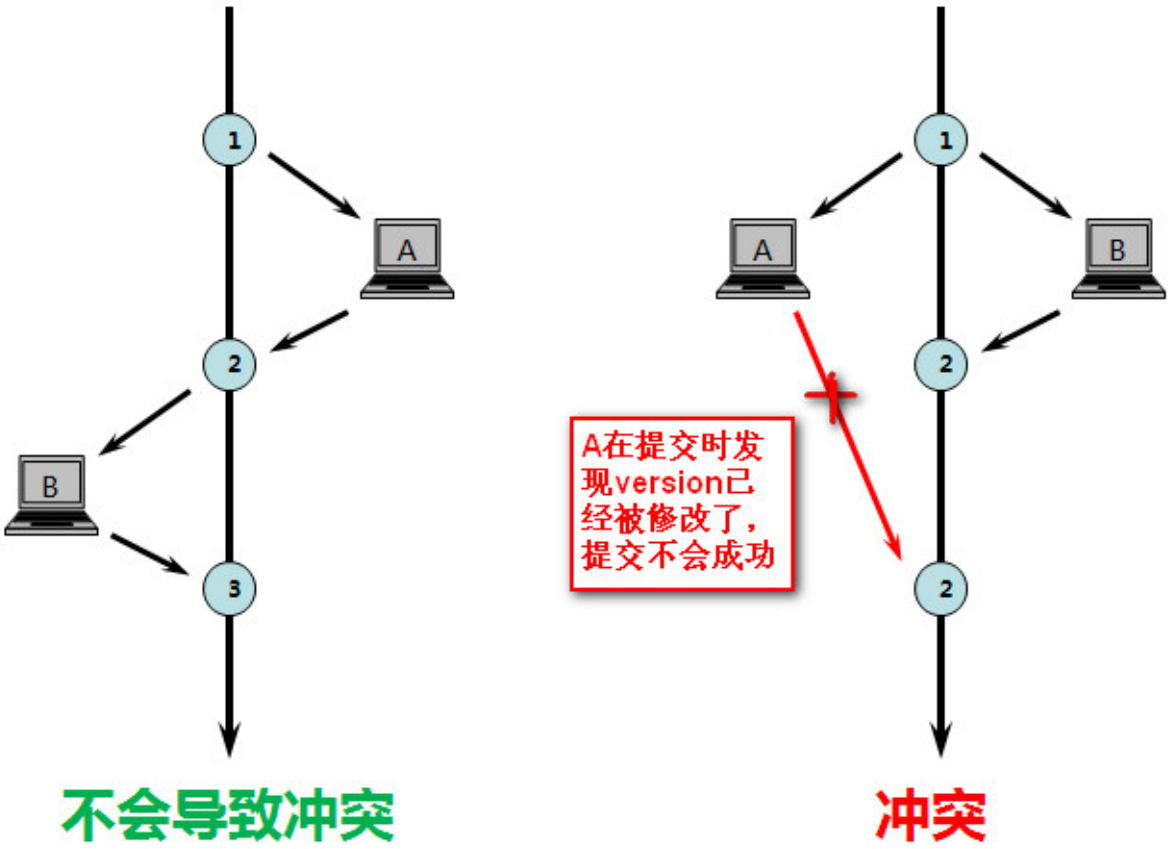
悲观锁，乐观锁

悲观锁(Pessimistic Lock), 顾名思义，就是很悲观，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会block直到它拿到锁。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。

> 大多数情况下依靠的是数据库的锁机制来实现。

乐观锁(Optimistic Lock), 顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。乐观锁适用于**多读**的应用类型，这样可以提高吞吐量，像数据库如果提供类似于write_condition机制的其实都是提供的乐观锁。

> （给表加一个版本号字段） 这个并不是乐观锁的定义，给表加版本号，是数据库实现乐观锁的一种方式。



两种锁各有优缺点，不可认为一种好于另一种，像乐观锁适用于**写比较少的情况**下，即冲突真的很少发生的时候，这样可以**省去了锁的开销**，加大了系统的整个吞吐量。但如果经常产生冲突，上层应用会不断的进行retry，这样反倒是降低了性能，所以这种情况下用悲观锁就比较合适。

隔离级别，依次解决的问题

> 脏读、不可重复读、幻读

√: 可能出现　×: 不会出现

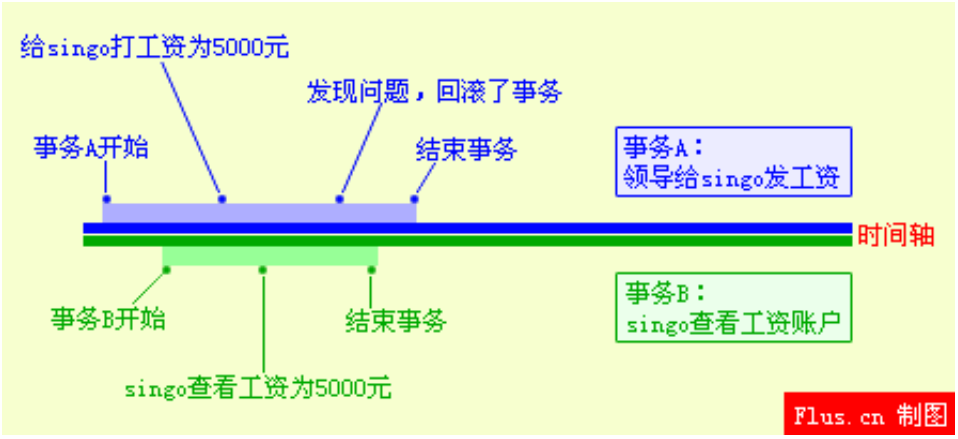
	脏读	不可重复读	幻读
Read uncommitted	√	√	√
Read committed	×	√	√
Repeatable read	×	×	√
Serializable	×	×	×

数据库事务的隔离级别有4个，由低到高依次为**Read uncommitted**、**Read committed**、**Repeatable read**、**Serializable**，这四个级别可以逐个解决**脏读**、**不可重复读**、**幻读**这几类问题。

注意：我们讨论隔离级别的场景，主要是在多个事务并发的情况下，因此，接下来的讲解都围绕事务并发。

Read uncommitted 读未提交

公司发工资了，领导把5000元打到singo的账号上，但是该事务并未提交，而singo正好去查看账户，发现工资已经到账，是5000元整，非常高兴。可是不幸的是，领导发现发给singo的工资金额不对，是2000元，于是迅速回滚了事务，修改金额后，将事务提交，最后singo实际的工资只有2000元，singo空欢喜一场。



出现上述情况，即我们所说的脏读，两个并发的事务，“事务A：领导给singo发工资”、“事务B：singo查询工资账户”，事务B读取了事务A尚未提交的数据。

当隔离级别设置为Read uncommitted时，就可能出现脏读，如何避免脏读，请看下一个隔离级别。

Read committed 读提交

singo拿着工资卡去消费，系统读取到卡里确实有2000元，而此时她的老婆也正好在网上转账，把singo工资卡的2000元转到另一账户，并在singo之前提交了事务，当singo扣款时，系统检查到singo的工资卡已经没有钱，扣款失败，singo十分纳闷，明明卡里有钱，为何……

出现上述情况，即我们所说的不可重复读，两个并发的事务，“事务A：singo消费”、“事务B：singo的老婆网上转账”，事务A事先读取了数据，事务B紧接了更新了数据，并提交了事务，而事务A再次读取该数据时，数据已经发生了改变。

当隔离级别设置为Read committed时，**避免了脏读**，但是可能会造成**不可重复读**。

大多数数据库的默认级别就是Read committed，比如**Sql Server , Oracle**。如何解决不可重复读这一问题，请看下一个隔离级别。

Repeatable read 重复读

当隔离级别设置为Repeatable read时，可以避免不可重复读。当singo拿着工资卡去消费时，一旦系统开始

读取工资卡信息（即事务开始），singo的老婆就不可能对该记录进行修改，也就是singo的老婆不能在此时转账。

虽然Repeatable read**避免了不可重复读**，但还有可能出现**幻读**。

singo的老婆工作在银行部门，她时常通过银行内部系统查看singo的信用卡消费记录。有一天，她正在查询到singo当月信用卡的总消费金额（select sum(amount) from transaction where month = 本月）为80元，而singo此时正好在外面胡吃海塞后在收银台买单，消费1000元，即新增了一条1000元的消费记录（insert transaction … ），并提交了事务，随后singo的老婆将singo当月信用卡消费的明细打印到A4纸上，却发现消费总额为1080元，singo的老婆很诧异，以为出现了幻觉，幻读就这样产生了。

注：**MySQL**的默认隔离级别就是Repeatable read。

Serializable 序列化

Serializable是最高的事务隔离级别，同时代价也花费最高，性能很低，一般很少使用，在该级别下，事务顺序执行，不仅可以避免脏读、不可重复读，还避免了幻像读。

> 数据库提供了四种事务隔离级别，不同的隔离级别采用不同的锁类开来实现.

1.Read uncommitted 读未提交数据：能解决第一类丢失更新的问题，但不能解决脏读的问题

实现原理是，读数据时候不加锁，写数据时候加行级别的共享锁，提交时释放锁。行级别的共享锁，不会对读产生影响，但是可以防止两个同时的写操作。

2.Read committed 读提交数据：能解决脏读的问题，但是不能解决不可重复读的问题：

实现原理是，事务读取数据（读到数据的时候）加行级共享锁，读完释放；事务写数据时候（写操作发生的瞬间）加行级独占锁，事务结束释放。由于事务写操作加上独占锁，因此事务写操作时，读操作也不能进行，因此，不能读到事务的未提交数据，避免了脏读的问题。**但是由于，读操作的锁加在读上面，而不是加在事务之上**，所以，在同一事务的两次读操作之间可以插入其他事务的写操作，所以可能发生不可重复读的问题。

3.Repeated Read 可重复读：顾名思义，可以解决不可重复读的问题，但是不能解决虚读问题：

实现原理，和读提交数据不同的是，**事务读取数据在读操作开始的瞬间就加上行级共享锁，而且在事务结束的时候才释放。**分析方法和读提交数据类似，本处不再赘述。但是，由于加锁只是加在行上，所以，仍然可能发生虚读的问题。

4. Serializable 串行化：可以解决以上所有的并发问题：

实现原理是，在读操作时，加表级共享锁，事务结束时释放；写操作时候，加表级独占锁，事务结束时释放。

事务的ACID

1. 原子性（Atomicity）
   原子性是指事务是一个不可分割的工作单位，事务中的操作要么都发生，要么都不发生。

2. 一致性（Consistency）
   事务必须使数据库从一个一致性状态变换到另外一个一致性状态。

3. 隔离性（Isolation）
   事务的隔离性是指一个事务的执行不能被其他事务干扰，即一个事务内部的操作及使用的数据对并发的其他事务是隔离的，并发执行的各个事务之间不能互相干扰。

4. 持久性（Durability）
   持久性是指一个事务一旦被提交，它对数据库中数据的改变就是永久性的，接下来的其他操作和数据库故障不应该对其有任何影响。

B树、B+树

B树

二分查找 –> 二叉查找树 –> 平衡查找树-2-3查找树 –> 红黑树 （AVL树，任意节点的俩子树高度差不大于1，删除插入后需要旋转）

```
参考资料：http://blog.csdn.net/yang_yulei/article/details/26066409
```

B树（B-tree）是一种树状数据结构，它能够存储数据、对其进行排序并允许以**O(log n)**的时间复杂度运行进行查找、顺序读取、插入和删除的数据结构。
B树，概括来说是一个节点可以拥有多于2个子节点的**二叉查找树**。与自平衡二叉查找树不同，B-树为系统最优化大块数据的读和写操作。B-tree算法减少定位记录时所经历的中间过程，从而加快存取速度。普遍运用在数据库和文件系统。**2-3树的扩展。**

- 根节点至少有两个子节点
- 每个节点有M-1个key，并且以升序排列
- 位于M-1和M key的子节点的值位于M-1 和M key对应的Value之间
- 其它节点至少有M/2个子节点

1. 关于B树中指针的表示。指针就是线索，是为了指示你找到目标。在内存中用内存的线性地址表示，在磁盘上，用磁盘的柱面和磁道号表示。
2. B树也是一种文件组织形式。它与OS文件系统的区别是，文件系统是面向磁盘上各种应用的文件的，所有文件的索引都被组织在一个系统文件表中。这样，一个相关应用的文件之间就没有体现有序性，我们对某组相关的文件进行查找，效率就会较低。 而B树是专门对某组相关的文件进行组织，使其之间相对有序，提高查找效率。 –尤其是对于需要频繁查找访问文件的操作。

通过自己建立的B树来索引文件（每次查找文件都通过该B树得到文件在磁盘上的位置）。B树是独立于OS的文件系统的，它中的每个文件都有相应的磁盘位置，而不仅是文件名。

```
    参考资料：http://blog.csdn.net/yang_yulei/article/details/26104921
    参考资料：http://www.cnblogs.com/yangecnu/archive/2014/03/29/Introduce-B-Tree-and-B-Plus-Tree.html
```

B+树

B+ tree：是应文件系统所需而产生的一种B-tree的变形树。
一棵m阶的B+树和m阶的B树的异同点在于：
1、有n棵子树的结点中含有n-1 个关键字； (与B 树n棵子树有n-1个关键字 保持一致，)
2、**所有的叶子结点中包含了全部关键字的信息，及指向含有这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大的顺序链接。便于查找和遍历**
3、所有的非终端结点可以看成是**索引部分**，结点中仅含有其子树根结点中最大（或最小）关键字。

【总结：最大的区别在于，B树是像2-3树那样把数据分散到所有的结点中，而B+树的数据都集中在叶结点，**上层结点只是数据的索引，并不包含数据信息】**

- 由于B+树在内部节点上不好含数据信息，因此在内存页中能够存放更多的key。 数据存放的更加紧密，具有更好的空间局部性。因此访问叶子几点上关联的数据也具有更好的缓存命中率。
- B+树的叶子结点都是相链的，因此对整棵树的便利只需要一次线性遍历叶子结点即可。而且由于数据顺序排列并且相连，所以便于区间查找和搜索。而B树则需要进行每一层的递归遍历。相邻的元素可能在内存中不相邻，所以缓存命中性没有B+树好。
- 但是B树也有优点，其优点在于，由于B树的每一个节点都包含key和value，因此经常访问的元素可能离根节点更近，因此访问也更迅速。有很多基于频率的搜索是选用B树，越频繁query的结点越往根上走。

range-query

1、为什么说B+-tree比B 树更适合实际应用中操作系统的文件索引和数据库索引?
数据库索引采用B+树的主要原因是 B树在提高了磁盘IO性能的同时并没有解决元素遍历的效率低下的问题。正是为了解决这个问题，B+树应运而生。
B+树只要遍历叶子节点就可以实现整棵树的遍历。而且在数据库中基于范围的查询是非常频繁的，而B树需要遍历整棵树，效率太低。

单词查找树（Ttri树）

优化

```
explain, 慢查询, show profile
```

MySQL 查询优化器有几个目标,但是其中最主要的目标是尽可能地使用索引,并且使用最严格的索引来消除尽

可能多的数据行。最终目标是提交 SELECT 语句查找数据行,而不是排除数据行。

我们经常会从职位描述上看到诸如“精通MySQL”、“SQL语句优化”、“了解数据库原理”等要求。我们知道一般的应用系统，读写比例在10:1左右，而且插入操作和一般的更新操作很少出现性能问题，遇到最多的，也是最容易出问题的，还是一些复杂的查询操作，所以查询语句的优化显然是重中之重。

explain

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。

- id：选定的执行计划中查询的序列号。
- selecttype：查询类型。
- table：输出行所引用的表
- type：显示连接使用的类型,按最优到最差的类型排序。
- possible_keys：指出 MySQL 能在该表中使用哪些索引有助于查询。如果为空,说明没有可用的索引。
- **key**：MySQL 实际从 possible_key 选择使用的索引。如果为 NULL,则没有使用索引。如果为 NULL,则没有使用索引。很少的情况 下,MYSQL 会选择优化不足的索引。
- **key_len**：使用的索引的长度。在不损失精确性的情况 下,长度越短越好。
- ref：显示索引的哪一列被使用了
- rows：MYSQL认为必须检查的用来返回请求数据的行数
- extra：Using filesort 、Using temporary，extra 中如果出现以上 2 项意味着 MYSQL 根本不能使用索引,效率会受到重大影响。应尽可能对此进行优化。

慢查询

- 分析MySQL语句查询性能的方法除了使用 EXPLAIN 输出执行计划，还可以让MySQL记录下查询超过指定时间的语句，我们将超过指定时间的SQL语句查询称为“慢查询”。
- 查看/设置“慢查询”的时间定义。
- 开启“慢查询”记录功能。（丢失了索引或索引没有得到最佳应用）

show profile

SHOW PROFIL命令是MySQL提供可以用来分析当前会话中语句执行的资源消耗情况。可以用于SQL的调优的测量。

数据库的范式

范式是关系数据库理论的基础，也是我们在设计数据库结构过程中所要遵循的规则和指导方法。

第一范式（1NF）：属性不可分。 × Tel –> 手机，座机

不满足第一范式的数据库，不是关系数据库！

第二范式（2NF）：符合1NF，并且，**非主属性完全依赖于码。**
另外包含两部分内容，一是表必须有一个主键；二是没有包含在主键中的列必须完全依赖于主键，而不能只依赖于主键的一部分。

```
学生   课程   老师   老师职称      教材   教室  上课时间
小明   一年级语文（上）      大宝   副教授  《小学语文1》  101  14：30

×××
```

第三范式（3NF）：符合2NF，并且，消除传递依赖
首先是 2NF，另外非主键列必须直接依赖于主键，不能存在传递依赖。即不能存在：非主键列 A 依赖于非主键列 B，非主键列 B 依赖于主键的情况。

第二范式（2NF）和第三范式（3NF）的概念很容易混淆，区分它们的关键点在于，2NF：非主键列是否完全依赖于主键，还是依赖于主键的一部分；3NF：非主键列是直接依赖于主键，还是直接依赖于非主键列。

分库分表，主从复制，读写分离。

分库分表

定义

从字面上简单理解，就是把原本存储于一个库的数据分块存储到多个库上，把原本存储于一个表的数据分块存储到多个表上。

原因

数据库中的数据量不一定是可控的，在未进行分库分表的情况下，随着时间和业务的发展，库中的表会越来越多，表中的数据量也会越来越大，相应地，数据操作，增删改查的开销也会越来越大；另外，由于无法进行分布式式部署，而一台服务器的资源（CPU、磁盘、内存、IO等）是有限的，最终数据库所能承载的数据量、数据处理能力都将遭遇瓶颈。

实施策略

分库分表有**垂直切分**和**水平切分**两种。

1. 何谓**垂直切分**，即将表按照*功能模块*、关系密切程度划分出来，部署到不同的库上。例如，我们会建立定义数据库workDB、商品数据库payDB、用户数据库userDB、日志数据库logDB等，分别用于存储项目数据定义表、商品定义表、用户数据表、日志数据表等。

2. 何谓**水平切分**，当一个表中的数据量过大时，我们可以把该表的数据按照某种规则，例如userID散列，

进行划分，然后存储到多个结构相同的表，和不同的库上。例如，我们的userDB中的用户数据表中，每一个表的数据量都很大，就可以把userDB切分为结构相同的多个userDB：part0DB、part1DB等，再将userDB上的用户数据表userTable，切分为很多userTable：userTable0、userTable1等，然后将这些表按照一定的规则存储到多个userDB上。

应该使用哪一种方式来实施数据库分库分表，这要看数据库中数据量的瓶颈所在，并综合项目的业务类型进行考虑。
如果数据库是因为表太多而造成海量数据，并且项目的各项业务逻辑划分清晰、低耦合，那么规则简单明了、容易实施的垂直切分必是首选。
而如果数据库中的表并不多，但单表的数据量很大、或数据热度很高，这种情况之下就应该选择水平切分，水平切分比垂直切分要复杂一些，它将原本逻辑上属于一体的数据进行了物理分割，除了在分割时要对分割的粒度做好评估，考虑数据平均和负载平均，后期也将对项目人员及应用程序产生额外的数据管理负担。

存在的问题

1. 事务问题。
2. 跨库跨表的join问题。需要多次查询。
3. 额外的数据管理负担和数据运算压力。

主从复制

mysql主从复制用途

1. 实时灾备，用于故障切换
2. 读写分离，提供查询服务
3. 备份，避免影响业务

MySQL内建的复制功能是构建大型、高性能应用程序的基础。将MySQL的数据分布到多个系统上去，这种分布式的机制，是通过将MySQL的某一台主机的数据，复制到其他的主机slaves上，并重新执行一边来实现。

复制过程中，一个服务器充当主服务器，而一个或多个其他服务器充当从服务器。主服务器将更新写入二进制日志文件，并维护文件的一个索引，以跟踪日志循环。这些日志可以记录发送到从服务器的更新。当一个从服务器连接到主服务器时，它通知主服务器，从服务器在日志中读取的最后一次成功更新的位置。从服务器接收从那时发生的任何更新，然后封锁并等待主服务器通知更新。

请注意，当你进行复制时，所有对复制中的表的更新，必须在主服务器上进行。否则，你必须要小心，以避免用户对主服务器上的表进行的更新与对从服务器的表进行的更新之间的冲突。

MySQL支持的复制的类型：

（1）：基于语句的复制：在服务器上执行的SQL语句，在从服务器上执行同样的语句。MySQL默认采用基于语句的复制，效率比较高。

（2）：基于行的复制：把改变的内容复制过去，而不是把命令在从服务器上执行一边。从MySQL5.0开始支

持。

（3）：混合类型的复制：默认采用基于语句的复制，一旦发现基于语句的无法精确复制时，就会采用基于行的复制。

MySQL复制技术有以下的特点：

(1) 数据的分布

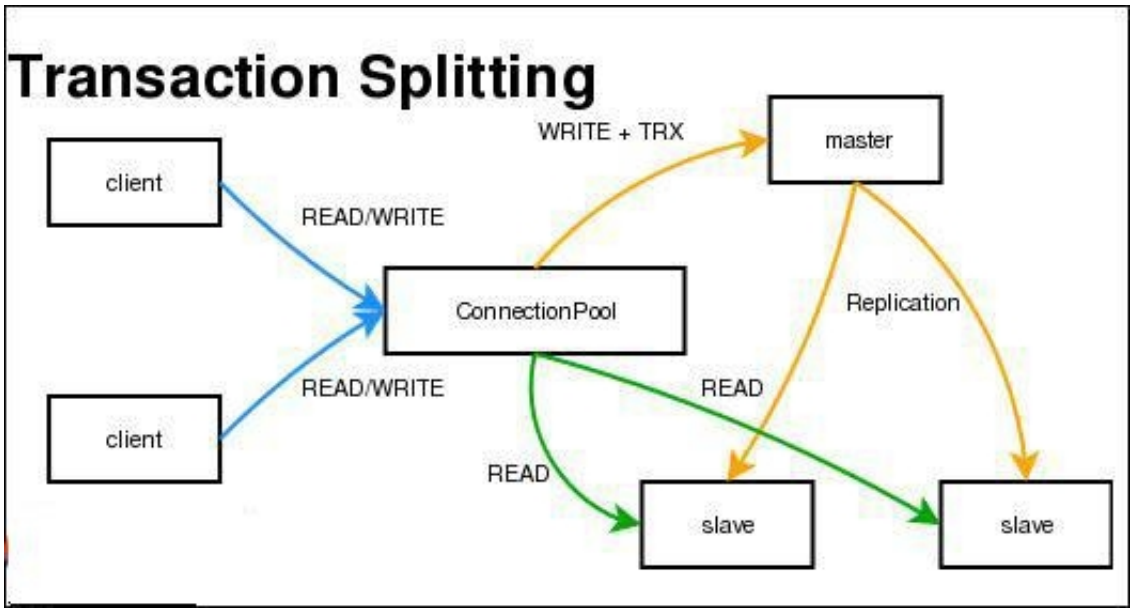
(2) 负载平衡

(3) 备份

(4) 备可用性和容错行

读写分离

MySQL Proxy最强大的一项功能是实现“读写分离(Read/Write Splitting)”。基本的原理是让主数据库处理事务性查询，而从数据库处理SELECT查询。数据库复制被用来把事务性查询导致的变更同步到集群中的从数据库。 当然，主服务器也可以提供查询服务。使用读写分离最大的作用无非是环境服务器压力。可以看下这张图：



读写分离的好处

1. 增加冗余
2. 增加了机器的处理能力
3. 对于读操作为主的应用，使用读写分离是最好的场景，因为可以确保写的服务器压力更小，而读又可以接受点时间上的延迟。

1.物理服务器增加，负荷增加
2.主从只负责各自的写和读，极大程度的缓解X锁和S锁争用
3.从库可配置myisam引擎，提升查询性能以及节约系统开销
4.从库同步主库的数据和主库直接写还是有区别的，通过主库发送来的binlog恢复数据，但是，最重要区别在于主库向从库发送binlog是异步的，从库恢复数据也是异步的
5.读写分离适用与读远大于写的场景，如果只有一台服务器，当select很多时，update和delete会被这些select访问中的数据堵塞，等待select结束，并发性能不高。 对于写和读比例相近的应用，应该部署双主相互复制
6.可以在从库启动是增加一些参数来提高其读的性能，例如–skip-innodb、–skip-bdb、–low-priority-updates以及–delay-key-write=ALL。
7.分摊读取。
8.MySQL复制另外一大功能是增加冗余，提高可用性，当一台数据库服务器宕机后能通过调整另外一台从库来以最快的速度恢复服务，因此不能光看性能，也就是说1主1从也是可以的。

NoSQL相关

```
redis和memcached区别之类的，如果你熟悉redis，redis还有一堆要问的
```

NoSQL，泛指非关系型的数据库。NoSQL数据库的产生就是为了解决大规模数据集合多重数据种类带来的挑战，尤其是大数据应用难题。
NoSQL被我们用得最多的当数key-value存储，当然还有其他的文档型的、列存储、图型数据库、xml数据库等。

NoSQL数据库的四大分类表格分析

分类	Examples举例	典型应用场景	数据模型	优点	缺点
键值（key-value）	Tokyo Cabinet/Tyrant, Redis, Voldemort, Oracle BDB	内容缓存，主要用于处理大量数据的高访问负载，也用于一些日志系统等等。	Key 指向 Value 的键值对，通常用hash table来实现	查找速度快	数据无结构化，通常只被当作字符串或者二进制数据
列存储数据库	Cassandra, HBase, Riak	分布式的文件系统	以列簇式存储，将同一列数据存在一起	查找速度快，可扩展性强，更容易进行分布式扩展	功能相对局限
文档型数据库	CouchDB, MongoDb	Web应用（与Key-Value类似，Value是结构化的，不同的是数据库能够了解Value的内容）	Key-Value对应的键值对，Value为结构化数据	数据结构要求不严格，表结构可变，不需要像关系型数据库一样需要预先定义表结构	查询性能不高，而且缺乏统一的查询语法。
图形(Graph)数据库	Neo4J, InfoGrid, Infinite Graph	社交网络，推荐系统等。专注于构建关系图谱	图结构	利用图结构相关算法。比如最短路径寻址，N度关系查找等	很多时候需要对整个图做计算才能得出需要的信息，而且这种结构不太好做分布式的集群方案。

1.NoSQL具有灵活的数据模型，可以处理非结构化/半结构化的大数据
2.NoSQL很容易实现可伸缩性（向上扩展与水平扩展）
3.动态模式
4.自动分片
5.复制

Memcached+MySQL

后来，随着访问量的上升，几乎大部分使用MySQL架构的网站在数据库上都开始出现了性能问题，web程序不再仅仅专注在功能上，同时也在追求性能。程序员们开始大量的使用缓存技术来缓解数据库的压力，优化数据库的结构和索引。开始比较流行的是通过文件缓存来缓解数据库压力，但是当访问量继续增大的时候，多台web机器通过文件缓存不能共享，大量的小文件缓存也带了了比较高的IO压力。在这个时候，Memcached就自然的成为一个非常时尚的技术产品。

Memcached作为一个独立的分布式的缓存服务器，为多个web服务器提供了一个共享的高性能缓存服务，在Memcached服务器上，又发展了根据hash算法来进行多台Memcached缓存服务的扩展，然后又出现了一致性hash来解决增加或减少缓存服务器导致重新hash带来的大量缓存失效的弊端。当时，如果你去面试，你说你有Memcached经验，肯定会加分的。

NOSQL的优势

- 易扩展

NoSQL数据库种类繁多，但是一个共同的特点都是去掉关系数据库的关系型特性。**数据之间无关系**，这样就非常容易扩展。也无形之间，在架构的层面上带来了可扩展的能力。

- 大数据量，高性能

NoSQL数据库都具有非常高的读写性能，尤其在大数据量下，同样表现优秀。**这得益于它的无关系性，数据库的结构简单。**一般MySQL使用Query Cache，每次表的更新Cache就失效，是一种大粒度的Cache，在针对web2.0的交互频繁的应用，Cache性能不高。而NoSQL的Cache是记录级的，是一种细粒度的Cache，所以NoSQL在这个层面上来说就要性能高很多了。

- 灵活的数据模型

NoSQL无需事先为要存储的数据建立字段，随时可以存储自定义的数据格式。而在关系数据库里，增删字段是一件非常麻烦的事情。如果是非常大数据量的表，增加字段简直就是一个噩梦。这点在大数据量的web2.0时代尤其明显。

- 高可用

NoSQL在不太影响性能的情况，就可以方便的实现高可用的架构。比如Cassandra，HBase模型，通过复制模型也能实现高可用。

Redis

一款内存高速缓存数据库。
Redis全称为：Remote Dictionary Server（远程数据服务），该软件使用C语言编写，Redis是一个key-value存储系统，它支持丰富的数据类型，如：string、list、set、zset(sorted set)、hash。

Redis以内存作为数据存储介质，所以读写数据的效率极高，远远超过数据库。

Redis跟memcache不同的是，储存在Redis中的数据是持久化的，断电或重启后，数据也不会丢失。因为Redis的存储分为内存存储、磁盘存储和log文件三部分，重启后，Redis可以从磁盘重新将数据加载到内存中，这些可以通过配置文件对其进行配置，正因为这样，Redis才能实现持久化。

Redis支持主从模式，可以配置集群，这样更利于支撑起大型的项目，这也是Redis的一大亮点。

其他

常见面试题

drop、delete与truncate分别在什么场景之下使用?

不再需要一张表的时候，用drop
想删除部分数据行时候，用delete，并且带上where子句
保留表而删除所有数据的时候用truncate

聚集索引 & 非聚集索引

聚集索引是指数据库表行中数据的物理顺序与键值的逻辑（索引）顺序相同。

1. 聚集索引一个表只能有一个，而非聚集索引一个表可以存在多个，这个跟没问题没差别，一般人都知道。

2. 聚集索引存储记录是物理上连续存在，而非聚集索引是逻辑上的连续，物理存储并不连续，这个大家也都知道。