数据库原理

一、事务

1、概念

事务指的是满足 ACID 特性的一组操作，可以通过 Commit 提交一个事务，也可以使用 Rollback 进行回滚。

2、ACID

2.1原子性

事务被视为不可分割的最小单元，事务的所有操作要么全部提交成功，要么全部失败回滚。

回滚可以用回滚日志来实现，回滚日志记录着事务所执行的修改操作，在回滚时反向执行这些修改操作即可。

2.2一致性

数据库在事务执行前后都保持一致性状态。在一致性状态下，所有事务对一个数据的读取结果都是相同的。

2.3隔离性

一个事务所做的修改在最终提交以前，对其它事务是不可见的。

2.4持久性

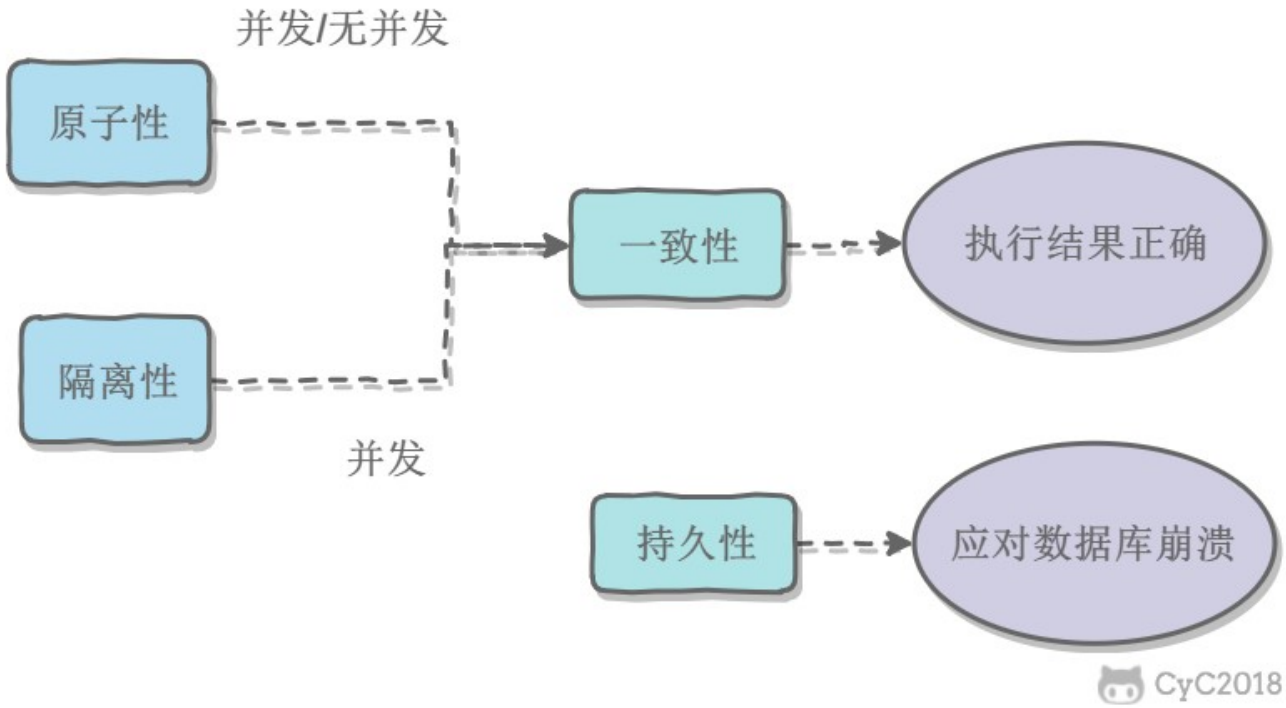
一旦事务提交，则其所做的修改将会永远保存到数据库中。即使系统发生崩溃，事务执行的结果也不能丢失。

使用重做日志来保证持久性。

2.5关系

这几个特性不是一种平级关系：

- 只有满足一致性，事务的执行结果才是正确的。
- 在无并发的情况下，事务串行执行，隔离性一定能够满足。此时只要能满足原子性，就一定能满足一致性。
- 在并发的情况下，多个事务并行执行，事务不仅要满足原子性，还需要满足隔离性，才能满足一致性。
- 事务满足持久化是为了能应对数据库崩溃的情况。



二、并发一致性问题

在并发环境下，事务的隔离性很难保证，因此会出现很多并发一致性问题。

1、丢失修改

T1 和 T2 两个事务都对一个数据进行修改，T1 先修改，T2 随后修改，T2 的修改覆盖了 T1 的修改。

旺财	小强
读取A的余额 100	
	读取A的余额 100
A=A+20	
写回 A = 120	
	A=A-50
	写回 A=50

很明显的看出，旺财对A添加的20块不翼而飞了，这就是“数据丢失”，对事务不加任何锁（不存在事务隔离），就会导致这种问题。

2、读脏数据

操作：写数据的时候添加一个**X锁（排他锁）**，也就是在写数据的时候不允许其他事务进行写操作，但是读不受限制，读不加锁。

旺财	小强
获取A的X锁	
读取A的值 100	获取A的X锁 **（失败）**
A=A+20	⋮
写回 A = 120	
释放A的X锁	
	获取A的X锁 **（成功）**
	读取A的值 120
	A=A-50
	写回 A=70
	释放A的X锁

这样就可以解决了多个人一起写数据而导致了“数据丢失”的问题，但是会引发新的问题——脏读。

脏读：读取了别人未提交的数据。

旺财	小强
获取A的X锁	
读取A的值100	
A=A+20	
写回 A = 120	
	读取A的值120
Rollback A 恢复为100	
释放A的X锁	
	尴尬了，小强的数据现在是错误的

3、不可重复读（读取了提交的新事物，指增删操作）

操作：写数据的时候加上X锁（排他锁），读数据的时候添加**S锁（共享锁）**，而且有约定：如果一个数据加了X锁就没法加S锁；同理如果加了S锁就没法加X锁，但是一个数据可以同时存在多个S锁（因为只是读数据），并且规定S锁读取数据，一旦读取完成就立刻释放S锁（不管后续是否还有很多其他的操作，只要是读取了S锁的数据后，就立刻释放S锁）。

旺财	小强
获取A的X锁	
读取A的值100	
A=A+20	
写回 A = 120	
	获取A的S锁 （失败）
Rollback A 恢复为100	⋮
释放A的X锁	
	获取A的S锁 （成功）
	读取A的值100 （正确）
	释放A的S锁

这样就解决了脏读的问题，但是又有新的问题出现——不可重复读。

不可重复读：同一个事务对数据的多次读取的结果不一致。

旺财	小强
获取A的S锁	
读取A的值100	
释放A的S锁	
获取B的S锁	
读取B的值50	
释放B的S锁	
C = A+B=150	
	获取B的X锁
	读取B的值50
	B = B-20=30
	写回B（30）
	释放B的X锁
获取A的S锁	
读取A的值100	
释放A的S锁	
获取B的S锁	
读取B的值30	
释放B的S锁	
C = A+B=**130**	

4、幻读（读取了提交的新事物，指增删操作）

操作：对S锁进行修改，之前的S锁是：读取了数据之后就立刻释放S锁，现在修改是：在读取数据的时候加上S锁，但是要直到事务准备提交了才释放该S锁，X锁还是一致。

旺财	小强
获取A的S锁	
获取B的S锁	
读取A的值100	
读取B的值50	
C = A+B=150	
做其他事情...	获取B的X锁 （失败）
读取A的值100	⋮
读取B的值50	
C = A+B=150	
提交事务	
释放A的S锁	
释放B的S锁	
	获取B的X锁 （成功）
	读取B的值50
	B = B-20=30
	写回B（30）
	释放B的X锁

这样就解决了不可重复读的问题了，但是又有新的问题出现——幻读。

例如：

有一次旺财对一个“学生表”进行操作，选取了年龄是18岁的所有行， 用X锁锁住， 并且做了修改。

改完以后旺财再次选择所有年龄是18岁的行， 想做一个确认， 没想到有一行竟然没有修改！

这是怎么回事？ 出了幻觉吗？

原来就在旺财查询并修改的的时候， 小强也对学生表进行操作， 他插入了一个新的行，其中的年龄也是18岁！ 虽然两个人的修改都没有问题， 互不影响， 但从最终效果看， 还是出了事。

5、读写锁

读写锁

- 排它锁（Exclusive），简写为 X 锁，又称写锁。
- 共享锁（Shared），简写为 S 锁，又称读锁。

有以下两个规定：

- 一个事务对数据对象 A 加了 X 锁，就可以对 A 进行读取和更新。加锁期间其它事务不能对 A 加任何锁。
- 一个事务对数据对象 A 加了 S 锁，可以对 A 进行读取操作，但是不能进行更新操作。加锁期间其它事务能对 A 加 S 锁，但是不能加 X 锁。

三、隔离级别

未提交读（READ UNCOMMITTED）（写加锁，读不加锁）
事务中的修改，即使没有提交，对其它事务也是可见的。

提交读（READ COMMITTED）（写加锁，读加锁）
一个事务只能读取已经提交的事务所做的修改。换句话说，一个事务所做的修改在提交之前对其它事务是不可见的。

可重复读（REPEATABLE READ）（写加锁，读加锁）
保证在同一个事务中多次读取同样数据的结果是一样的。
对于读操作加读锁到事务结束，其他事务的更新操作只能等到事务结束之后进行。和提交 读的区别在于，提交读的读操作是加读锁到本次读操作结束，可重复读的锁粒度更大。

可串行化（SERIALIZABLE）
强制事务串行执行。

mysql默认的隔离级别是：可重复读。

四、SQL

1、数据库创建

```
CREATE DATABASE test;
USE test;
```

2、创建表

```
CREATE TABLE mytable (
  # int 类型，不为空，自增
  id INT NOT NULL AUTO_INCREMENT,
```

```
  # int 类型，不可为空，默认值为 1，不为空
  col1 INT NOT NULL DEFAULT 1,
  # 变长字符串类型，最长为 45 个字符，可以为空
  col2 VARCHAR(45) NULL,
  # 日期类型，可为空
  col3 DATE NULL,
  # 设置主键为 id
  PRIMARY KEY (`id`));
```

3、修改表

添加列

```
ALTER TABLE mytable
ADD col CHAR(20);
```

删除列

```
ALTER TABLE mytable
DROP COLUMN col;
```

删除表

```
DROP TABLE mytable;
```

4、插入

普通插入

```
INSERT INTO mytable(col1, col2)
VALUES(val1, val2);
```

插入检索出来的数据

```
INSERT INTO mytable1(col1, col2)
SELECT col1, col2
FROM mytable2;
```

将一个表的内容插入到一个新表

```
CREATE TABLE newtable AS
SELECT * FROM mytable;
```

5、更新

```
UPDATE mytable
SET col = val
WHERE id = 1;
```

6、删除

```
DELETE FROM mytable
WHERE id = 1;
```

TRUNCATE TABLE 可以清空表，也就是删除所有行。

```
TRUNCATE TABLE mytable;
```

使用更新和删除操作时一定要用 WHERE 子句，不然会把整张表的数据都破坏。可以先用 SELECT 语句进行测试，防止错误删除。

7、查询

DISTINCT
相同值只会出现一次。它作用于所有列，也就是说所有列的值都相同才算相同。

```
SELECT DISTINCT col1, col2
FROM mytable;
```

LIMIT
限制返回的行数。可以有两个参数，第一个参数为起始行，从 0 开始；第二个参数为返回的总行数。

返回前 5 行：

```
SELECT *
FROM mytable
LIMIT 5;
```

```
SELECT *
FROM mytable
LIMIT 0, 5;
```

返回第 3 ~ 5 行：

```
SELECT *
FROM mytable
LIMIT 2, 3;
```

8、排序

- ASC ： 升序（默认）
- DESC ： 降序
  可以按多个列进行排序，并且为每个列指定不同的排序方式：

```
SELECT *
FROM mytable
ORDER BY col1 DESC, col2 ASC;
```

9、过滤

不进行过滤的数据非常大，导致通过网络传输了多余的数据，从而浪费了网络带宽。因此尽量使用 SQL 语句来过滤不必要的数据，而不是传输所有的数据到客户端中然后由客户端进行过滤。

```
SELECT *
FROM mytable
WHERE col IS NULL;
```

下表显示了 WHERE 子句可用的操作符

操作符	说明
=	等于
<	小于
>	大于
<> !=	不等于
<= !>	小于等于
>= !<	大于等于
BETWEEN	在两个值之间
IS NULL	为 NULL

应该注意到，NULL 与 0、空字符串都不同。

AND 和 **OR** 用于连接多个过滤条件。优先处理 AND，当一个过滤表达式涉及到多个 AND 和 OR 时，可以使用 () 来决定优先级，使得优先级关系更清晰。

IN操作符用于匹配一组值，其后也可以接一个 SELECT 子句，从而匹配子查询得到的一组值。

NOT 操作符用于否定一个条件。

10、通配符

通配符也是用在过滤语句中，但它只能用于文本字段。

- % 匹配 >=0 个任意字符；
- _ 匹配 ==1 个任意字符；
- [ ] 可以匹配集合内的字符，例如 [ab] 将匹配字符 a 或者 b。用脱字符 ^ 可以对其进行否定，也就是不匹配集合内的字符。

使用 Like 来进行通配符匹配。

```
SELECT *
FROM mytable
WHERE col LIKE '[^AB]%'; -- 不以 A 和 B 开头的任意文本
```

11、计算字段

在数据库服务器上完成数据的转换和格式化的工作往往比客户端上快得多，并且转换和格式化后的数据量更少的话可以减少网络通信量。

计算字段通常需要使用 AS 来取别名，否则输出的时候字段名为计算表达式。

```
SELECT col1 * col2 AS alias
FROM mytable;
```

CONCAT() 用于连接两个字段。许多数据库会使用空格把一个值填充为列宽，因此连接的结果会出现一些不必要的空格，使用 TRIM() 可以去除首尾空格。

```
SELECT CONCAT(TRIM(col1), '(', TRIM(col2), ')') AS concat_col
FROM mytable;
```

12、函数

以下主要是 MySQL 的函数。

函 数	说明
AVG()	返回某列的平均值
COUNT()	返回某列的行数
MAX()	返回某列的最大值
MIN()	返回某列的最小值
SUM()	返回某列值之和

AVG() 会忽略 NULL 行。

使用 DISTINCT 可以汇总不同的值。

```
SELECT AVG(DISTINCT col1) AS avg_col
FROM mytable;
```

13、文本处理

函数	说明
LEFT ()	左边的字符
RIGHT()	右边的字符
LOWER()	转换为小写字符
UPPER()	转换为大写字符
LTRIM()	去除左边的空格
RTRIM()	去除右边的空格
LENGTH()	长度
SOUNDEX()	转换为语音值

14、分组

把具有相同的数据值的行放在同一组中。

可以对同一分组数据使用汇总函数进行处理，例如求分组数据的平均值等。

指定的分组字段除了能按该字段进行分组，也会自动按该字段进行排序。

```
SELECT col, COUNT(*) AS num
FROM mytable
GROUP BY col;
```

GROUP BY 自动按分组字段进行排序，ORDER BY 也可以按汇总字段来进行排序。

```
SELECT col, COUNT(*) AS num
FROM mytable
GROUP BY col
ORDER BY num;
```

WHERE 过滤行，HAVING 过滤分组，行过滤应当先于分组过滤。

```
SELECT col, COUNT(*) AS num
FROM mytable
WHERE col > 2
GROUP BY col
HAVING num >= 2;
```

分组规定：

- GROUP BY 子句出现在 WHERE 子句之后，ORDER BY 子句之前；
- 除了汇总字段外，SELECT 语句中的每一字段都必须在 GROUP BY 子句中给出；
- NULL 的行会单独分为一组；
- 大多数 SQL 实现不支持 GROUP BY 列具有可变长度的数据类型。

15、子查询

子查询中只能返回一个字段的数据。

可以将子查询的结果作为 WHRER 语句的过滤条件：

```
SELECT *
FROM mytable1
WHERE col1 IN (SELECT col2
               FROM mytable2);
```

下面的语句可以检索出客户的订单数量，子查询语句会对第一个查询检索出的每个客户执行一次：

```
SELECT cust_name, (SELECT COUNT(*)
                   FROM Orders
                   WHERE Orders.cust_id = Customers.cust_id)
                   AS orders_num
```

```
FROM Customers
ORDER BY cust_name;
```

16、连接

连接用于连接多个表，使用 JOIN 关键字，并且条件语句使用 ON 而不是 WHERE。

连接可以替换子查询，并且比子查询的效率一般会更快。

可以用 AS 给列名、计算字段和表名取别名，给表名取别名是为了简化 SQL 语句以及连接相同表。

内连接
内连接又称等值连接，使用 INNER JOIN 关键字。

```
SELECT A.value, B.value
FROM tablea AS A INNER JOIN tableb AS B
ON A.key = B.key;
```

可以不明确使用 INNER JOIN，而使用普通查询并在 WHERE 中将两个表中要连接的列用等值方法连接起来。

```
SELECT A.value, B.value
FROM tablea AS A, tableb AS B
WHERE A.key = B.key;
```

自连接
自连接可以看成内连接的一种，只是连接的表是自身而已。

一张员工表，包含员工姓名和员工所属部门，要找出与 Jim 处在同一部门的所有员工姓名。

子查询版本

```
SELECT name
FROM employee
WHERE department = (
      SELECT department
      FROM employee
      WHERE name = "Jim");
```

自连接版本

```
SELECT e1.name
FROM employee AS e1 INNER JOIN employee AS e2
```

```
ON e1.department = e2.department
      AND e2.name = "Jim";
```

自然连接
自然连接是把同名列通过等值测试连接起来的，同名列可以有多个。

内连接和自然连接的区别：内连接提供连接的列，而自然连接自动连接所有同名列。

```
SELECT A.value, B.value
FROM tablea AS A NATURAL JOIN tableb AS B;
```

外连接
外连接保留了没有关联的那些行。分为左外连接，右外连接以及全外连接，左外连接就是保留左表没有关联的行。

检索所有顾客的订单信息，包括还没有订单信息的顾客。

```
SELECT Customers.cust_id, Orders.order_num
FROM Customers LEFT OUTER JOIN Orders
ON Customers.cust_id = Orders.cust_id;
```

17、组合查询

使用 UNION 来组合两个查询，如果第一个查询返回 M 行，第二个查询返回 N 行，那么组合查询的结果一般为 M+N 行。

每个查询必须包含相同的列、表达式和聚集函数。

默认会去除相同行，如果需要保留相同行，使用 UNION ALL。

只能包含一个 ORDER BY 子句，并且必须位于语句的最后。

```
SELECT col
FROM mytable
WHERE col = 1
UNION
SELECT col
FROM mytable
WHERE col =2;
```

18、视图

视图是虚拟的表，本身不包含数据，也就不能对其进行索引操作。

对视图的操作和对普通表的操作一样。

视图具有如下好处：

- 简化复杂的 SQL 操作，比如复杂的连接
- 只使用实际表的一部分数据；
- 通过只给用户访问视图的权限，保证数据的安全性；
- 更改数据格式和表示。

```
CREATE VIEW myview AS
SELECT Concat(col1, col2) AS concat_col, col3*col4 AS compute_col
FROM mytable
WHERE col5 = val;
```

18、存储过程

存储过程可以看成是对一系列 SQL 操作的批处理。

使用存储过程的好处：

- 代码封装，保证了一定的安全性；
- 代码复用；
- 由于是预先编译，因此具有很高的性能。

命令行中创建存储过程需要自定义分隔符，因为命令行是以 ; 为结束符，而存储过程中也包含了分号，因此会错误把这部分分号当成是结束符，造成语法错误。

包含 in、out 和 inout 三种参数。

给变量赋值都需要用 select into 语句。

每次只能给一个变量赋值，不支持集合的操作。

```
delimiter //

create procedure myprocedure( out ret int )
    begin
        declare y int;
        select sum(col1)
        from mytable
        into y;
        select y*y into ret;
    end //

delimiter ;
```

```
call myprocedure(@ret);
select @ret;
```

五、MySQL

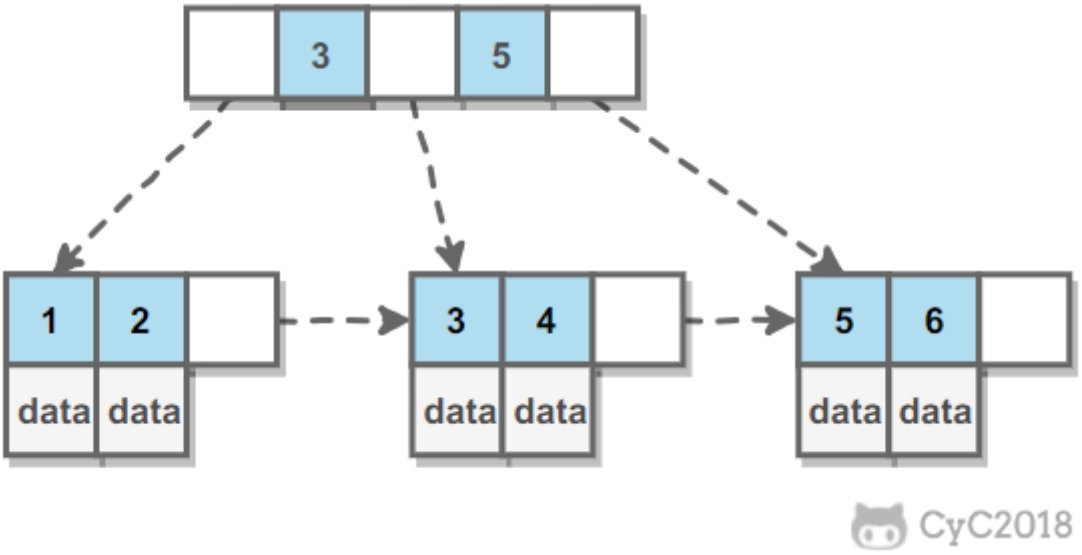
1、索引

1.1B+树原理

数据结构
B Tree 指的是 Balance Tree，也就是平衡树。平衡树是一颗查找树，并且所有叶子节点位于同一层。

B+ Tree 是基于 B Tree 和叶子节点顺序访问指针进行实现，它具有 B Tree 的平衡性，并且通过顺序访问指针来提高区间查询的性能。

在 B+ Tree 中，一个节点中的 key 从左到右非递减排列



操作
进行查找操作时，首先在根节点进行二分查找，找到一个 key 所在的指针，然后递归地在指针所指向的节点进行查找。直到查找到叶子节点，然后在叶子节点上进行二分查找，找出 key 所对应的 data。

插入删除操作会破坏平衡树的平衡性，因此在插入删除操作之后，需要对树进行一个分裂、合并、旋转等操作来维护平衡性。

与红黑树的比较
红黑树等平衡树也可以用来实现索引，但是文件系统及数据库系统普遍采用 B+ Tree 作为索引结构，主要有以下两个原因：

（一）更少的查找次数

平衡树查找操作的时间复杂度和树高 h 相关，O(h)=O(logdN)，其中 d 为每个节点的出度。

红黑树的出度为 2，而 B+ Tree 的出度一般都非常大，所以红黑树的树高 h 很明显比 B+ Tree 大非常多，查找的次数也就更多。

（二）利用磁盘预读特性

为了减少磁盘 I/O 操作，磁盘往往不是严格按需读取，而是每次都会预读。预读过程中，磁盘进行顺序读取，顺序读取不需要进行磁盘寻道，并且只需要很短的磁盘旋转时间，速度会非常快。

操作系统一般将内存和磁盘分割成固定大小的块，每一块称为一页，内存与磁盘以页为单位交换数据。数据库系统将索引的一个节点的大小设置为页的大小，使得一次 I/O 就能完全载入一个节点。并且可以利用预读特性，相邻的节点也能够被预先载入。

与B树的比较
B+树的磁盘读写代价更低：B+树的内部节点并没有指向关键字具体信息的指针，因此其内部节点相对B树更小，如果把所有同一内部节点的关键字存放在同一盘块中，那么盘块所能容纳的关键字数量也越多，一次性读入内存的需要查找的关键字也就越多，相对IO读写次数就降低了。

B+树的查询效率更加稳定：由于非终结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引。所以任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当。

由于B+树的数据都存储在叶子结点中，分支结点均为索引，方便扫库，只需要扫一遍叶子结点即可，但是B树因为其分支结点同样存储着数据，我们要找到具体的数据，需要进行一次中序遍历按序来扫，所以B+树更加适合在区间查询的情况，所以通常B+树用于数据库索引。

1.2MySQL索引

索引是在存储引擎层实现的，而不是在服务器层实现的，所以不同存储引擎具有不同的索引类型和实现。

1.2.1B+Tree索引

是大多数 MySQL 存储引擎的默认索引类型。

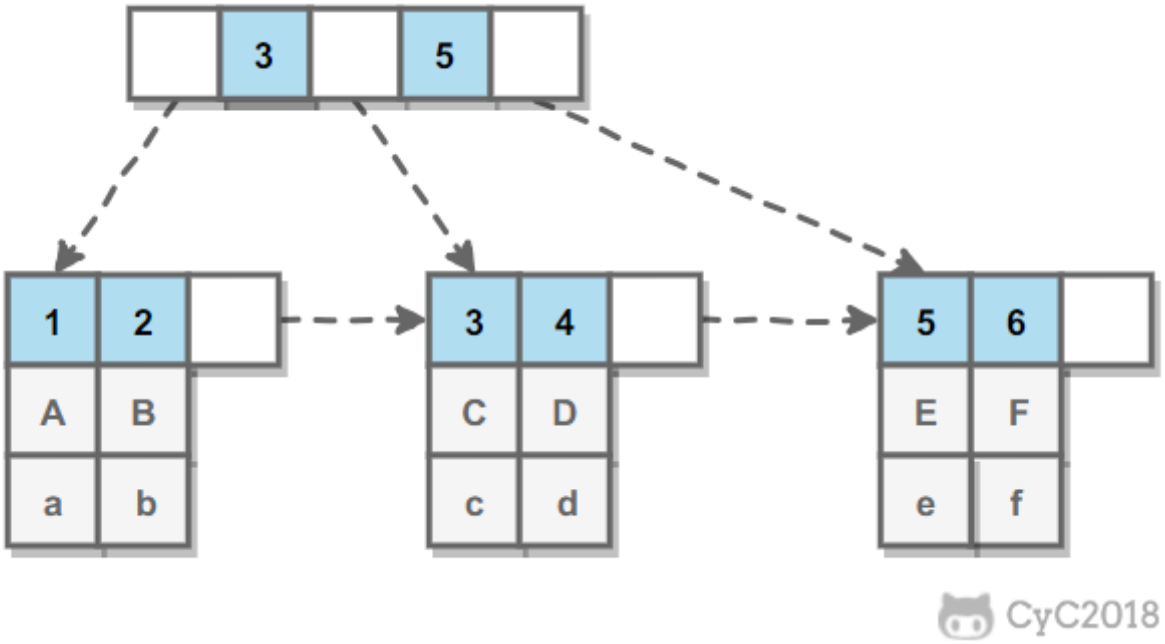
因为不再需要进行全表扫描，只需要对树进行搜索即可，所以查找速度快很多。

因为 B+ Tree 的有序性，所以除了用于查找，还可以用于排序和分组。

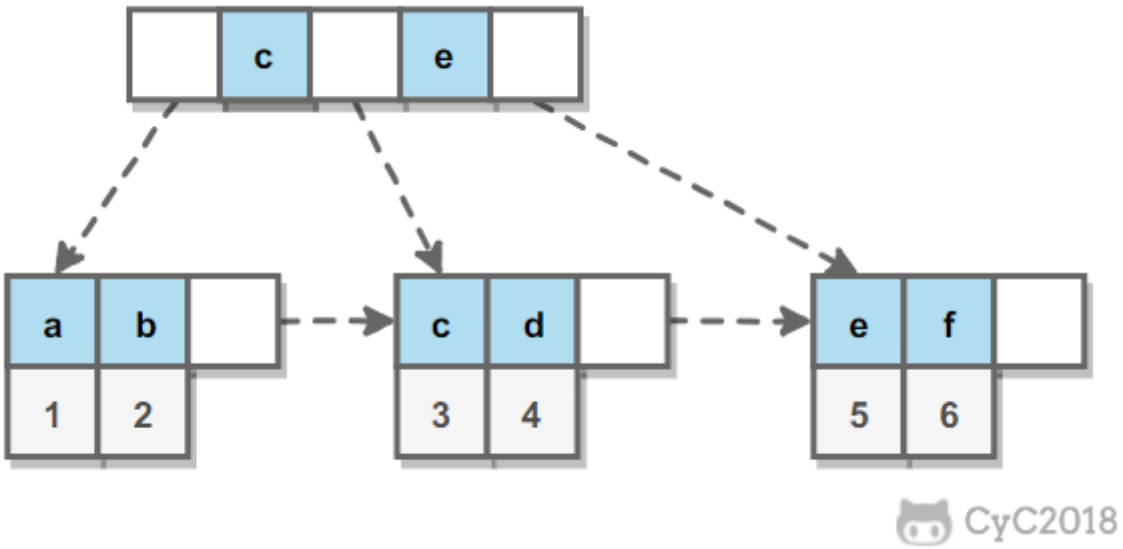
可以指定多个列作为索引列，多个索引列共同组成键。

适用于全键值、键值范围和键前缀查找，其中键前缀查找只适用于最左前缀查找。如果不是按照索引列的顺序进行查找，则无法使用索引。

InnoDB 的 B+Tree 索引分为主索引和辅助索引。主索引的叶子节点 data 域记录着完整的数据记录，这种索引方式被称为聚簇索引。因为无法把数据行存放在两个不同的地方，所以一个表只能有一个聚簇索引。



辅助索引的叶子节点的 data 域记录着主键的值，因此在使用辅助索引进行查找时，需要先查找到主键值，然后再到主索引中进行查找。



1.2.2哈希索引

哈希索引能以 O(1) 时间进行查找，但是失去了有序性：

- 无法用于排序与分组；
- 只支持精确查找，无法用于部分查找和范围查找。

InnoDB 存储引擎有一个特殊的功能叫“自适应哈希索引”，当某个索引值被使用的非常频繁时，会在 B+Tree 索引之上再创建一个哈希索引，这样就让 B+Tree 索引具有哈希索引的一些优点，比如

快速的哈希查找。

1.2.3全文索引

MyISAM 存储引擎支持全文索引，用于查找文本中的关键词，而不是直接比较是否相等。

查找条件使用 MATCH AGAINST，而不是普通的 WHERE。

全文索引使用倒排索引实现，它记录着关键词到其所在文档的映射。

InnoDB 存储引擎在 MySQL 5.6.4 版本中也开始支持全文索引。

1.2.4空间数据索引

MyISAM 存储引擎支持空间数据索引（R-Tree），可以用于地理数据存储。空间数据索引会从所有维度来索引数据，可以有效地使用任意维度来进行组合查询。

必须使用 GIS 相关的函数来维护数据。

1.3聚集索引和非聚集索引

聚集索引和非聚集索引的根本区别是表记录的**排列顺序和与索引的排列顺序是否一致**。

聚集索引
聚集索引表记录的排列顺序和索引的排列顺序一致，所以查询效率快，只要找到第一个索引值记录，其余就连续性的记录在物理也一样连续存放。聚集索引对应的缺点就是修改慢，因为为了保证表中记录的物理和索引顺序一致，在记录插入的时候，会对数据页重新排序。

非聚集索引
非聚集索引制定了表中记录的逻辑顺序，但是记录的物理和索引不一定一致，两种索引都采用B+树结构，非聚集索引的叶子层并不和实际数据页相重叠，而采用叶子层包含一个指向表中的记录在数据页中的指针方式。非聚集索引层次多，不会造成数据重排。

例子对比两种索引
聚集索引就类似新华字典中的拼音排序索引，都是按顺序进行，例如找到字典中的“爱”，就里面顺序执行找到“癌”。而非聚集索引则类似于笔画排序，索引顺序和物理顺序并不是按顺序存放的

2、存储引擎

InnoDB
是 MySQL 默认的事务型存储引擎，只有在需要它不支持的特性时，才考虑使用其它存储引擎。

实现了四个标准的隔离级别，默认级别是可重复读（REPEATABLE READ）。在可重复读隔离级别下，通过多版本并发控制（MVCC）+ Next-Key Locking 防止幻影读。

主索引是聚簇索引，在索引中保存了数据，从而避免直接读取磁盘，因此对查询性能有很大的提升。

内部做了很多优化，包括从磁盘读取数据时采用的可预测性读、能够加快读操作并且自动创建的自适应哈希索引、能够加速插入操作的插入缓冲区等。

支持真正的在线热备份。其它存储引擎不支持在线热备份，要获取一致性视图需要停止对所有表的写入，而在读写混合场景中，停止写入可能也意味着停止读取。

MyISAM
设计简单，数据以紧密格式存储。对于只读数据，或者表比较小、可以容忍修复操作，则依然可以使用它。

提供了大量的特性，包括压缩表、空间数据索引等。

不支持事务。

不支持行级锁，只能对整张表加锁，读取时会对需要读到的所有表加共享锁，写入时则对表加排它锁。但在表有读取操作的同时，也可以往表中插入新的记录，这被称为并发插入（CONCURRENT INSERT）。

可以手工或者自动执行检查和修复操作，但是和事务恢复以及崩溃恢复不同，可能导致一些数据丢失，而且修复操作是非常慢的。

如果指定了 DELAY_KEY_WRITE 选项，在每次修改执行完成时，不会立即将修改的索引数据写入磁盘，而是会写到内存中的键缓冲区，只有在清理键缓冲区或者关闭表的时候才会将对应的索引块写入磁盘。这种方式可以极大的提升写入性能，但是在数据库或者主机崩溃时会造成索引损坏，需要执行修复操作。

比较

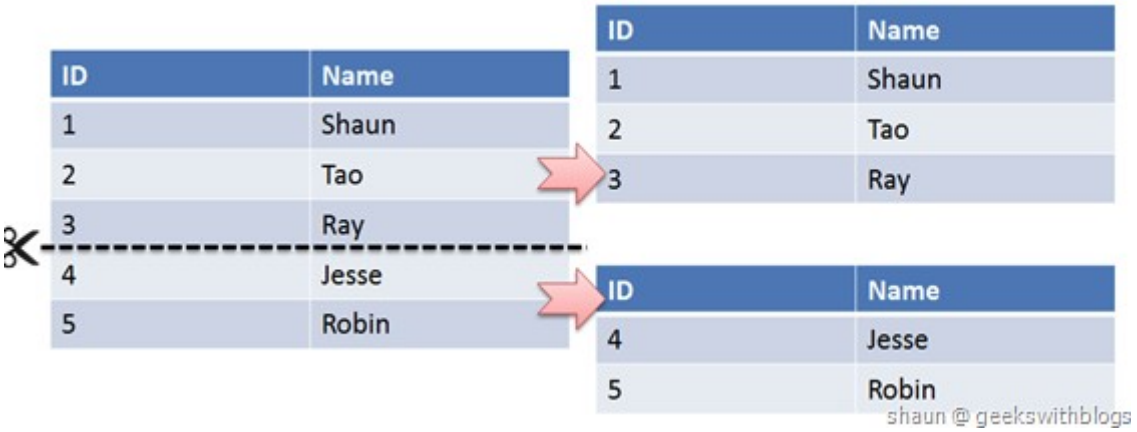
- 事务：InnoDB 是事务型的，可以使用 Commit 和 Rollback 语句。
- 并发：MyISAM 只支持表级锁，而 InnoDB 还支持行级锁。
- 外键：InnoDB 支持外键。
- 备份：InnoDB 支持在线热备份。
- 崩溃恢复：MyISAM 崩溃后发生损坏的概率比 InnoDB 高很多，而且恢复的速度也更慢。
- 其它特性：MyISAM 支持压缩表和空间数据索引。

3、切分

3.1水平切分

水平切分又称为 Sharding，它是将同一个表中的记录拆分到多个结构相同的表中。

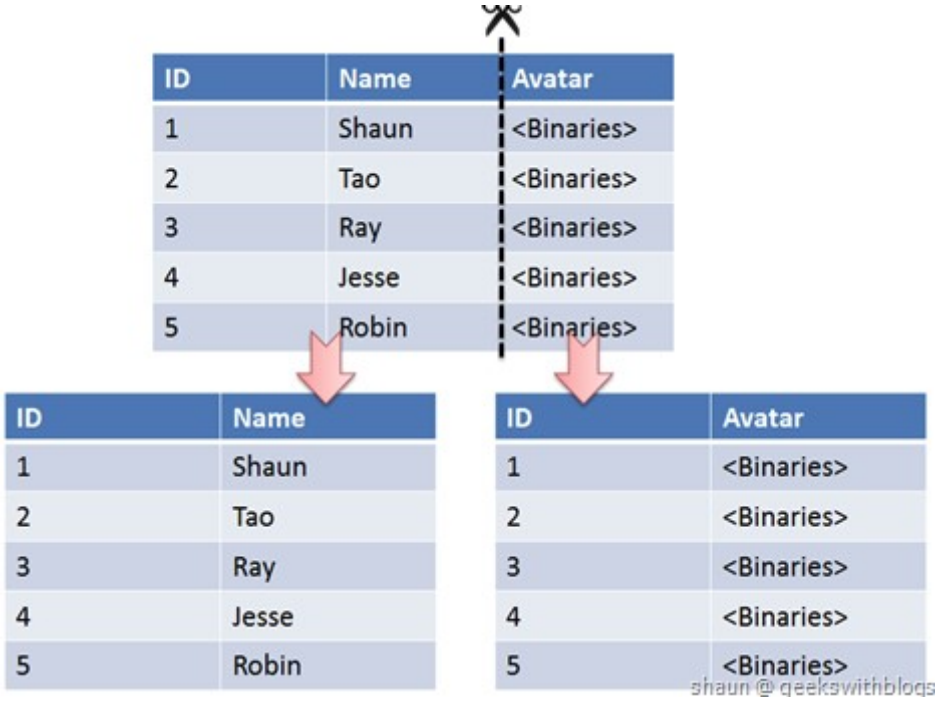
当一个表的数据不断增多时，Sharding 是必然的选择，它可以将数据分布到集群的不同节点上，从而缓存单个数据库的压力。



3.2垂直切分

垂直切分是将一张表按列切分成多个表，通常是按照列的关系密集程度进行切分，也可以利用垂直切分将经常被使用的列和不经常被使用的列切分到不同的表中。

在数据库的层面使用垂直切分将按数据库中表的密集程度部署到不同的库中，例如将原来的电商数据库垂直切分成商品数据库、用户数据库等。



4、复制

4.1主从复制

主要涉及三个线程：binlog 线程、I/O 线程和 SQL 线程。

- binlog 线程： 负责将主服务器上的数据更改写入二进制日志（Binary log）中。
- I/O 线程： 负责从主服务器上读取二进制日志，并写入从服务器的中继日志（Relay log）。
  SQL 线程： 负责读取中继日志，解析出主服务- - 器已经执行的数据更改并在从服务器中重放（Replay）。

4.2读写分离

主服务器处理写操作以及实时性要求比较高的读操作，而从服务器处理读操作。

读写分离能提高性能的原因在于：

- 主从服务器负责各自的读和写，极大程度缓解了锁的争用；
- 从服务器可以使用 MyISAM，提升查询性能以及节约系统开销；
- 增加冗余，提高可用性。

读写分离常用代理方式来实现，代理服务器接收应用层传来的读写请求，然后决定转发到哪个服务器。