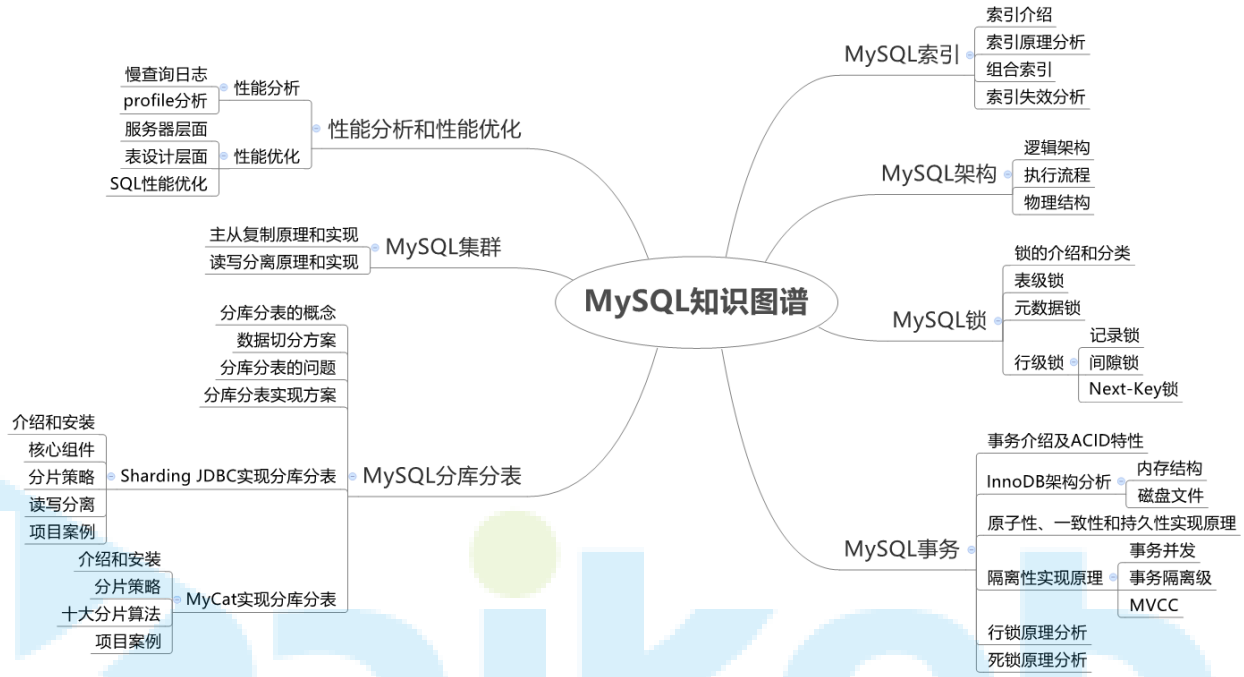
MySQL知识图谱



一、MySQL索引篇

索引介绍

索引是什么

- 官方介绍索引是帮助MySQL**高效获取数据**的**数据结构**。更通俗的说，数据库索引好比是一本书前面的目录，能**加快数据库的查询速度**。

  方便查找---检索

  索引查询内容---覆盖索引

  排序

- 一般来说索引本身也很大，不可能全部存储在内存中，因此**索引往往是存储在磁盘上的文件中的**（可能存储在单独的索引文件中，也可能和数据一起存储在数据文件中）。
- **我们通常所说的索引，包括聚集索引、覆盖索引、组合索引、前缀索引、唯一索引等，没有特别说明，默认都是使用B+树结构组织（多路搜索树，并不一定是二叉的）的索引。**

索引的优势和劣势

优势：

- **可以提高数据检索的效率，降低数据库的IO成本**，类似于书的目录。 -- 检索

- 通过**索引列对数据进行排序**，降低数据排序的成本，降低了CPU的消耗。 --排序
    - 被索引的列会自动进行排序，包括【单列索引】和【组合索引】，只是组合索引的排序要复杂一些。
    - 如果按照索引列的顺序进行排序，对应order by语句来说，效率就会提高很多。
    - where 索引列 在存储引擎层 处理
    - 覆盖索引，不需要回表查询

劣势：

- **索引会占据磁盘空间**
- **索引虽然会提高查询效率，但是会降低更新表的效率**。比如每次对表进行增删改操作，MySQL不仅要保存数据，还有保存或者更新对应的索引文件。

索引的分类

单列索引

- 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。 add index
- 唯一索引：索引列中的值必须是唯一的，但是允许为空值. add unique index
- 主键索引：是一种特殊的唯一索引，不允许有空值。 pk

组合索引

- 在表中的**多个字段组合上创建的索引** add index(col1,col2..)
- 组合索引的使用，需要遵循**最左前缀原则（最左匹配原则，后面高级篇讲解）**。
- 一般情况下，**建议使用组合索引代替单列索引**（主键索引除外，具体原因后面知识点讲解）。

全文索引

只有在**MyISAM引擎**、**InnoDB**（5.6以后）上才能使用，而且只能在CHAR,VARCHAR,TEXT类型字段上使用全文索引。 fulltext

优先级最高 先执行 不会执行其他索引

存储引擎 决定执行一个索引

空间索引

不做介绍，一般使用不到。

索引的使用

创建索引

- 单列索引之普通索引

```
CREATE INDEX index_name ON table(column(length)) ;
ALTER TABLE table_name ADD INDEX index_name (column(length)) ;
```

- 单列索引之唯一索引

```
CREATE UNIQUE INDEX index_name ON table(column(length)) ;
alter table table_name add unique index index_name(column);
```

- 单列索引之全文索引

```
CREATE FULLTEXT INDEX index_name ON table(column(length)) ;
alter table table_name add fulltext index_name(column)
```

- 组合索引

```
ALTER TABLE article ADD INDEX index_titme_time (title(50),time(10)) ;
```

删除索引

```
DROP INDEX index_name ON table
```

查看索引

```
SHOW INDEX FROM table_name \G
```

索引原理分析

索引的存储结构

索引存储结构

- 索引是在**存储引擎中实现**的，也就是说不同的存储引擎，会使用不同的索引
- **MyISAM和InnoDB存储引擎**：只支持**B+ TREE索引**， 也就是说**默认使用BTREE，不能够更换**
- MEMORY/HEAP存储引擎：支持HASH和BTREE索引

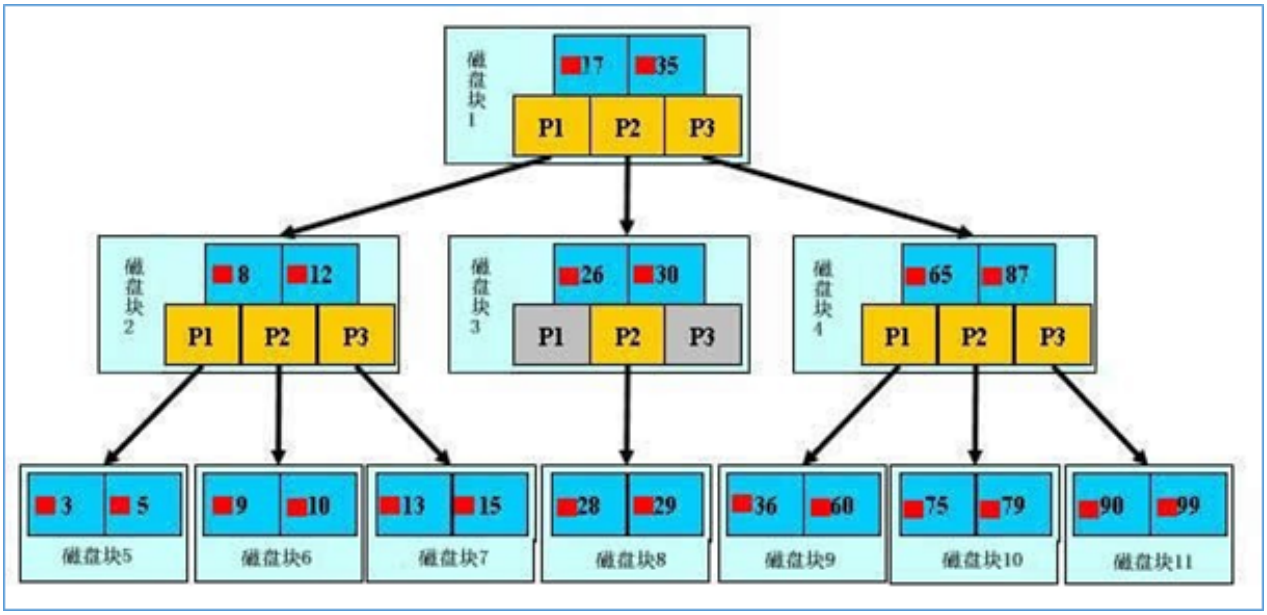
B树和B+树

数据结构示例网站：

https://www.cs.usfca.edu/~galles/visualization/Algorithms.html

B树图示

B树是为了磁盘或其它存储设备而设计的一种多叉（下面你会看到，相对于二叉，B树每个内结点有多个分支，即多叉）平衡查找树。 多叉平衡



- B树的高度一般都是在2-4这个高度，树的高度直接影响IO读写的次数。
- 如果是三层树结构---支撑的数据可以达到20G，如果是四层树结构---支撑的数据可以达到几十T

B树和B+树的区别

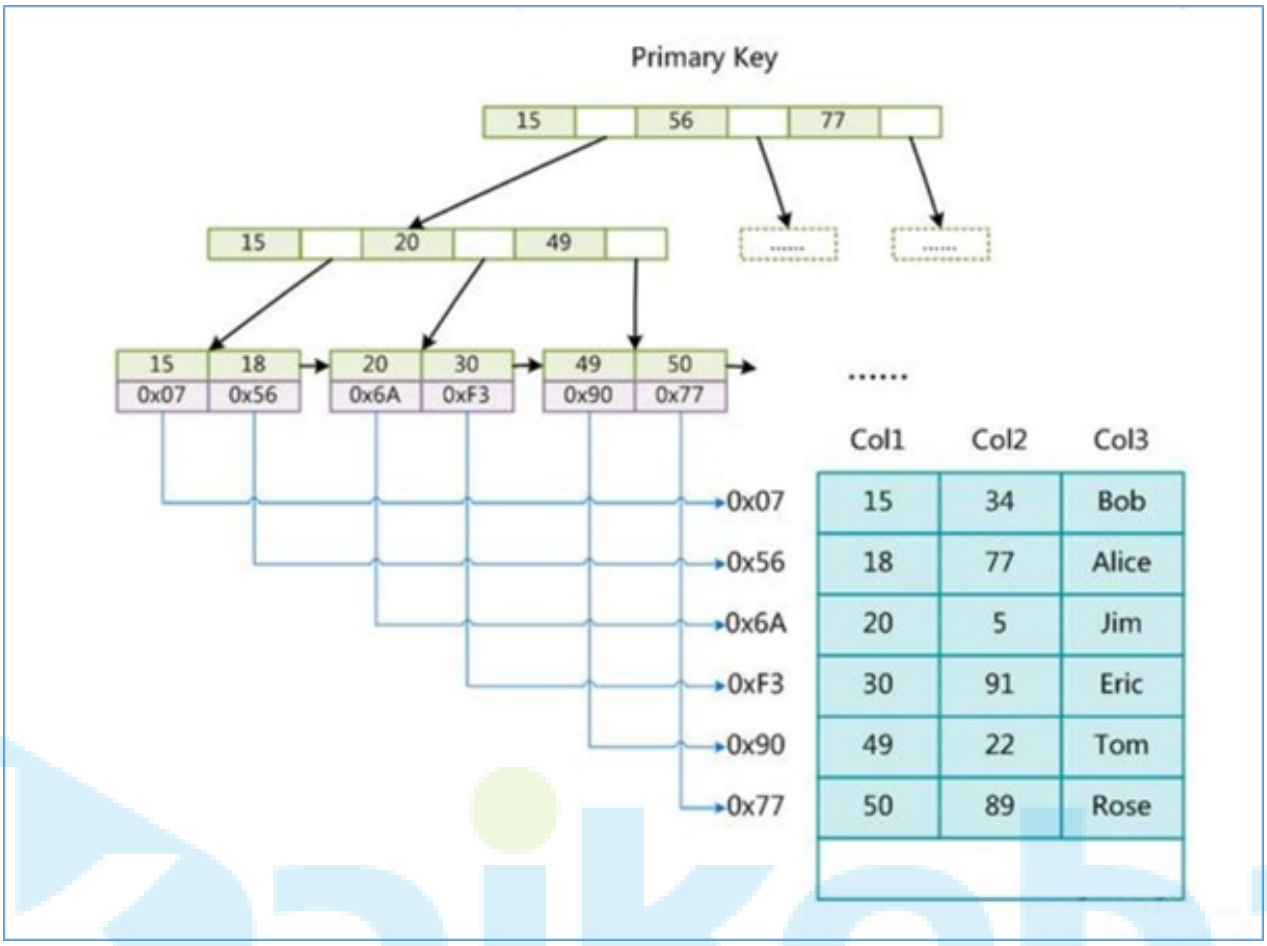
B树和B+树的最大区别在于**非叶子节点是否存储数据**的问题。

- B树是非叶子节点和叶子节点都会存储数据。
- B+树只有叶子节点才会存储数据，而且存储的数据都是在一行上，而且这些数据都是有指针指向的，也就是有顺序的。 索引列 `order by`

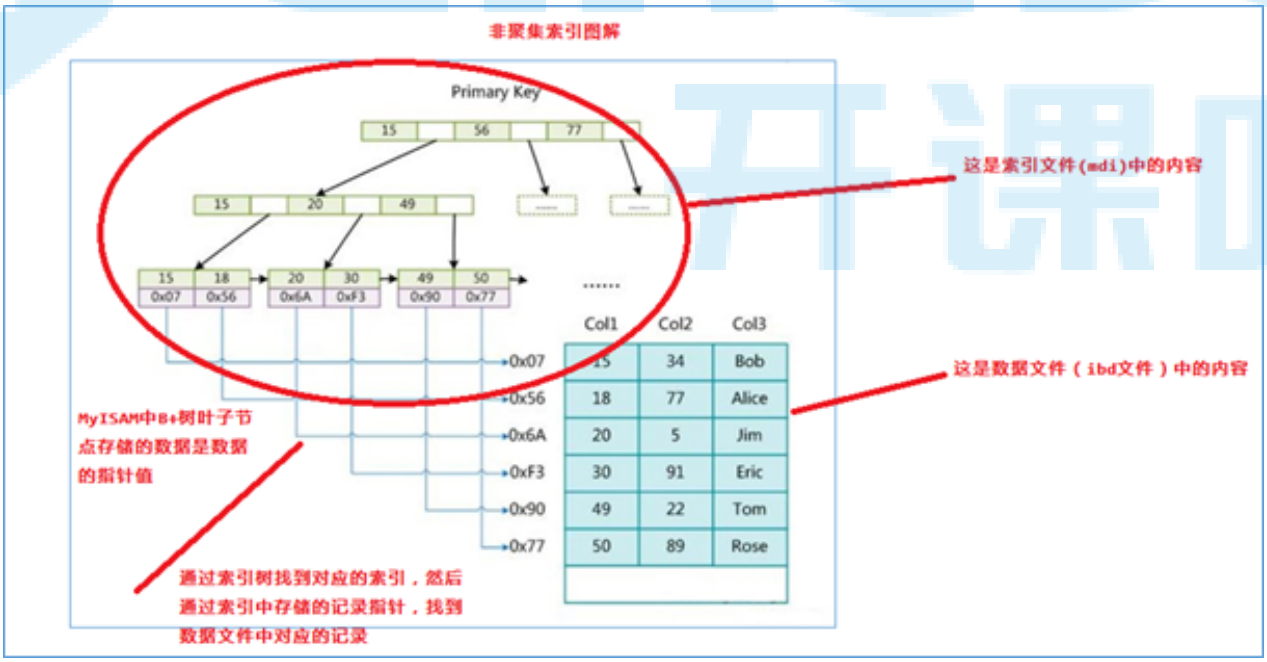
非聚集索引（MyISAM）

- B+树叶子节点只会存储数据行（数据文件）的指针，简单来说**数据和索引不在一起**，就是非聚集索引。
- **非聚集索引包含主键索引和辅助索引都会存储指针的值**

主键索引

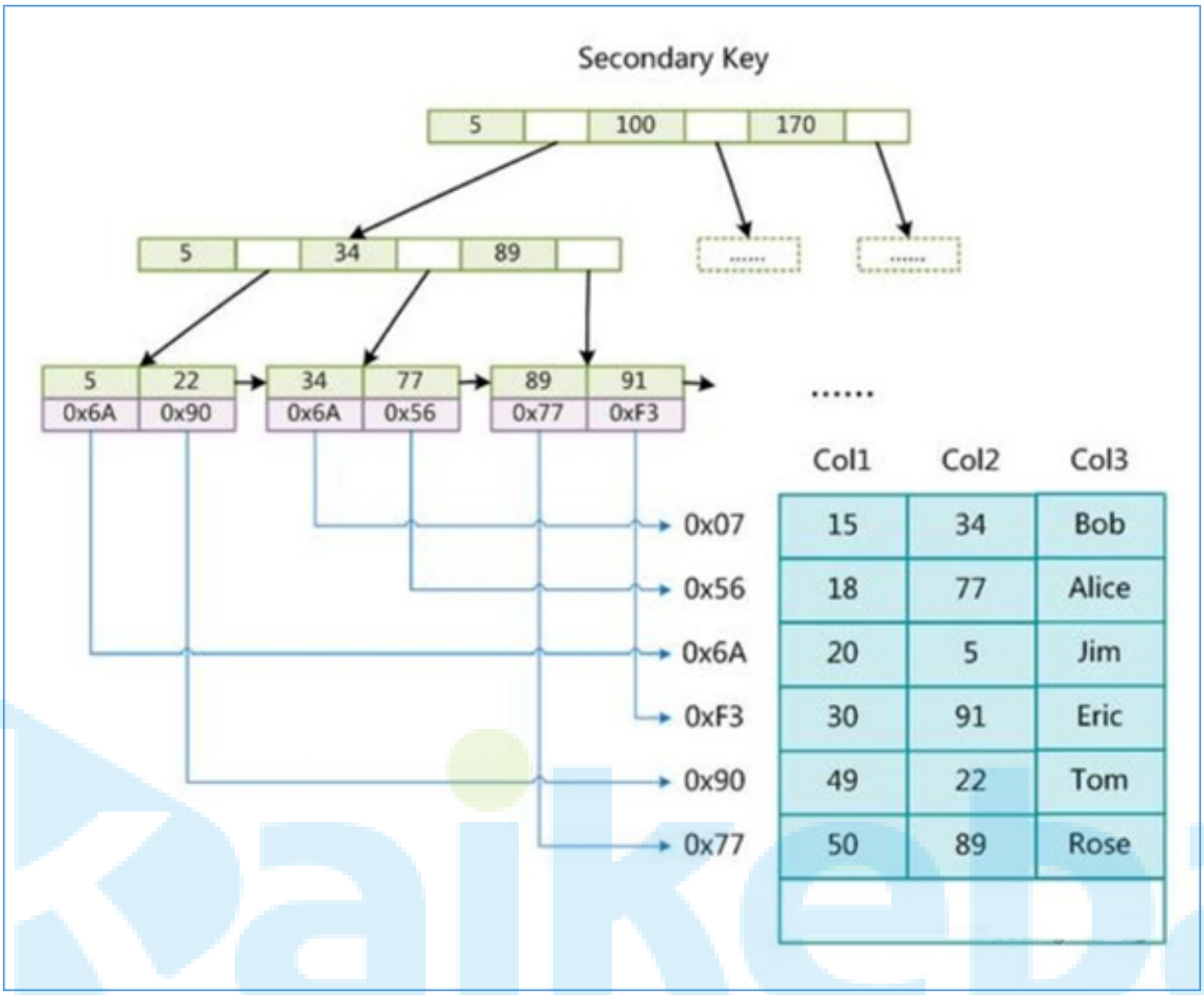


这里设表一共有三列,假设我们以 Col1 为主键,则上图是一个 MyISAM 表的主索引(Primary key)示意。可以看出 MyISAM 的索引文件仅仅保存数据记录的地址。



辅助索引（次要索引）

在 MyISAM 中,主索引和辅助索引(Secondary key)在结构上没有任何区别,只是主索引要求 key 是唯一的,而辅助索引的 key 可以重复。如果我们在 Col2 上建立一个辅助索引,则此索引的结构如下图所示



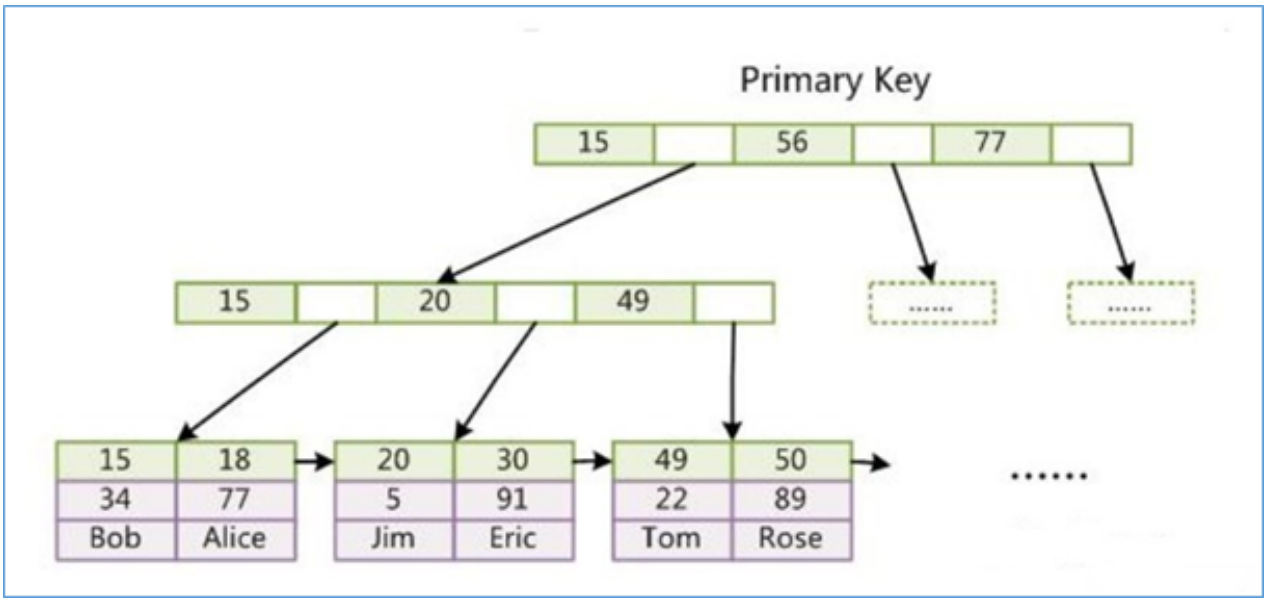
同样也是一颗 B+Tree,data 域保存数据记录的地址。因此,MyISAM 中索引检索的算法为首先按照 B+Tree 搜索算法搜索索引,如果指定的 Key 存在,则取出其data 域的值,然后以 data 域的值为地址,读取相应数据记录。

聚集索引（InnoDB）

- 主键索引（聚集索引）的叶子节点会存储数据行，**也就是说数据和索引是在一起，这就是聚集索引。**
- 辅助索引只会存储主键值
- 如果没有没有主键，则使用唯一索引建立聚集索引；如果没有唯一索引，MySQL会按照一定规则创建聚集索引。

主键索引

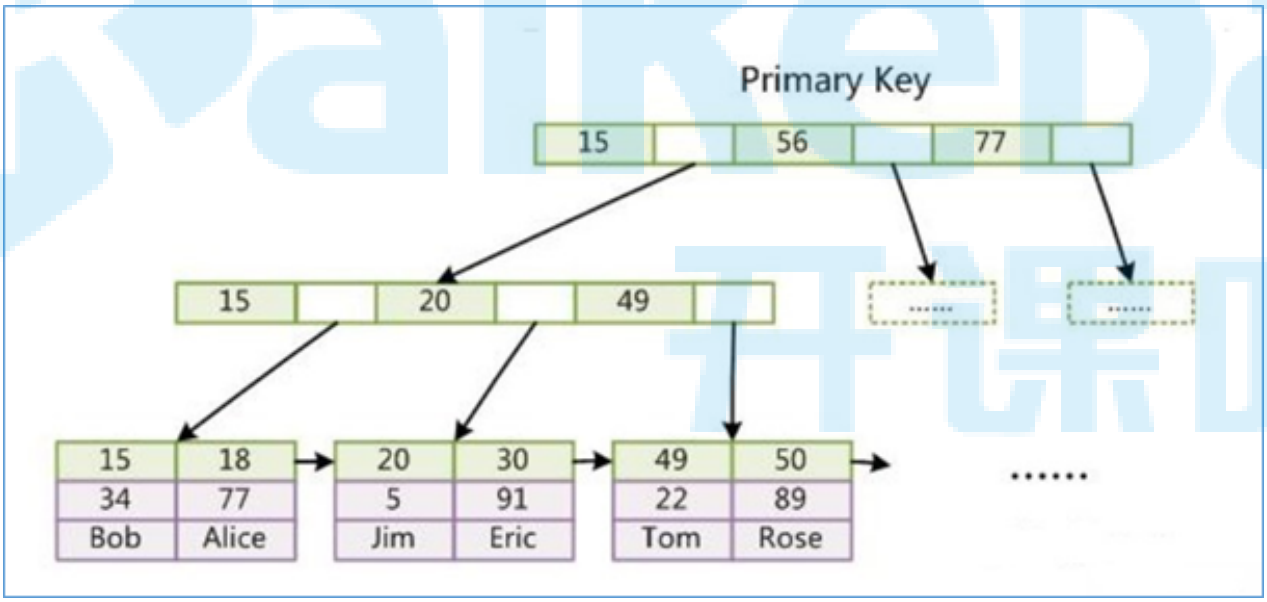
1.InnoDB 要求表必须有主键(MyISAM 可以没有),如果没有显式指定,则 MySQL系统会自动选择一个可以唯一标识数据记录的列作为主键,如果不存在这种列,则MySQL 自动为 InnoDB 表生成一个隐含字段作为主键,类型为长整形。



上图是 InnoDB 主索引(同时也是数据文件)的示意图,可以看到叶节点包含了完整的数据记录。这种索引叫做聚集索引。因为 InnoDB 的数据文件本身要按主键聚集,

辅助索引（次要索引）

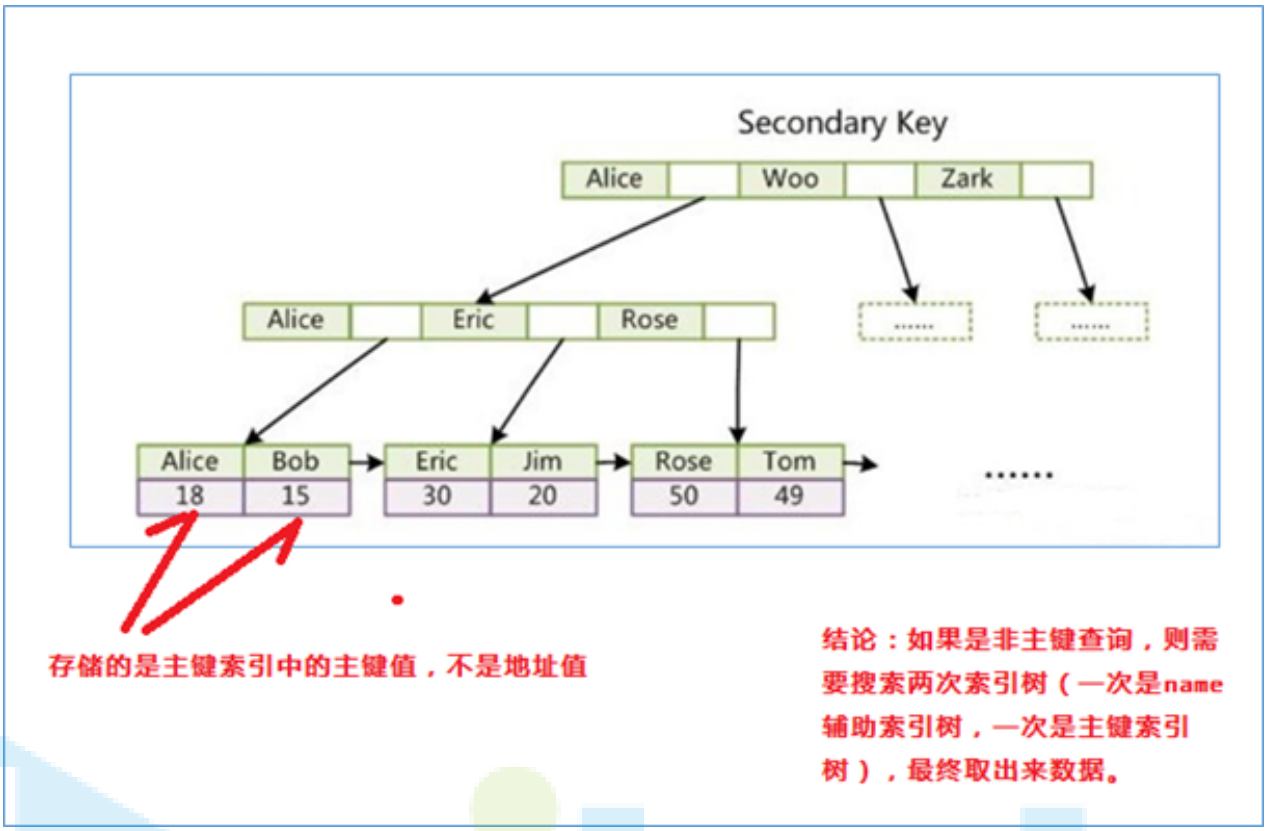
2.第二个与 MyISAM 索引的不同是 InnoDB 的辅助索引 data 域存储相应记录主键的值而不是地址。换句话说,InnoDB 的所有辅助索引都引用主键作为 data 域。



聚集索引这种实现方式使得按主键的搜索十分高效,但是辅助索引搜索需要检索两遍索引:首先检索辅助索引获得主键,然后用主键到主索引中检索获得记录。

select * from user where name='Alice' 回表查询 检索两次 非主键索引 --- pk---索引--->数据

select id,name from user where name='Alice' 不需要回表 在辅助索引树上就可以查询到了 覆盖索引(多用组合索引)



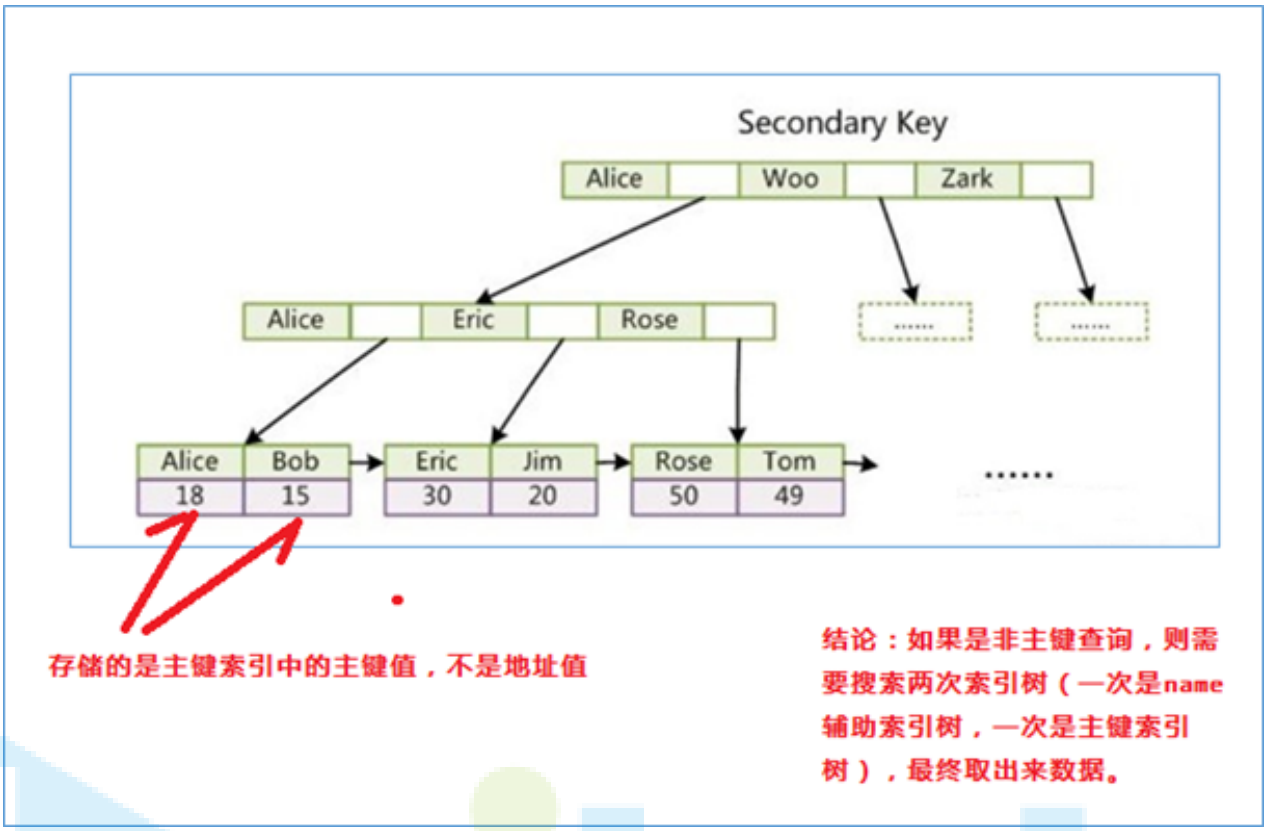
引申:为什么不建议使用过长的字段作为主键?

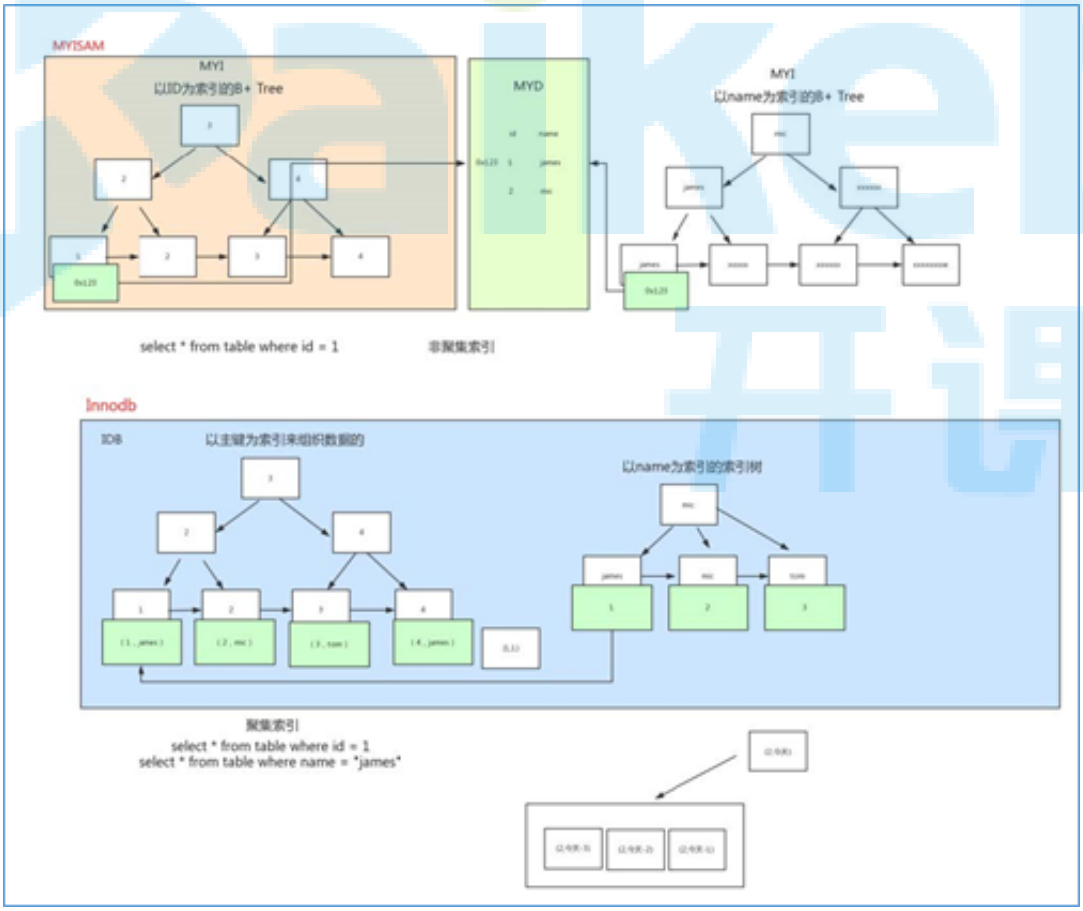
因为所有辅助索引都引用主索引,过长的主索引会令辅助索引变得过大。

同时,请尽量在 InnoDB 上采用自增字段做表的主键。

MyISAM 和 InnoDB的存储结构图示

为了更形象说明这两种索引的区别, 我们假想一个表如下图存储了 4 行数据。 **其中Id 作为主索引, Name 作为辅助索引。** 图示清晰的显示了聚簇索引和非聚簇索引的差异:





课堂主题

Mysql组合索引、索引失效分析、表级锁介绍

课堂目标

掌握索引使用场景

理解组合索引的结构和掌握使用原则

使用explain查看sql执行计划

掌握select_type、type、extra等参数意义

理解索引失效口诀

编写使用索引的sql

掌握MySQL的锁的分类

理解表级锁和元数据锁

知识要点

哪些情况需要创建索引

1. 主键自动建立唯一索引
2. 频繁作为查询条件的字段应该创建索引
3. 多表关联查询中，关联字段应该创建索引 on 两边都要创建索引
4. 查询中排序的字段，应该创建索引
5. 频繁查找字段 覆盖索引
6. 查询中统计或者分组字段，应该创建索引 group by

哪些情况不需要创建索引

1. 表记录太少
2. 经常进行增删改操作的表
3. 频繁更新的字段
4. where条件里使用频率不高的字段

为什么使用组合索引

mysql创建组合索引的规则是首先会对组合索引的最左边的，也就是第一个name字段的数据进行排序，在第一个字段的排序基础上，然后再对后面第二个的cid字段进行排序。其实就相当于实现了类似order by name cid这样一种排序规则。

为了节省mysql索引存储空间以及提升搜索性能，可建立组合索引（**能使用组合索引就不使用单列索引**）

例如：

创建组合索引（相当于建立了col1,col1 col2,col1 col2 col3三个索引）：

```
ALTER TABLE 'table_name' ADD INDEX index_name('col1','col2','col3')
```

一颗索引树上创建3个索引： 省空间

三颗索引树上分别创建1个索引

更容易实现覆盖索引

使用 遵循最左前缀原则

1、前缀索引 like a%

2、从左向右匹配直到遇到范围查询 > < between like

建立组合索引 (a,b,c,d)

where a=1 and b=1 and c>3 and d=1

到c>3停止了 所以d 用不到索引了

怎么办?

（a,b,d,c）

案例

```
mysql>  create table t1 (id int primary key ,a int ,b int,c int,d int);

mysql> alter table t1 add index idx_com(a,b,c,d);
mysql> drop index idx_com on t1;
mysql> create index idx_com on t1(a,b,d,c);
```

索引失效

查看执行计划

介绍

MySQL 提供了一个 **EXPLAIN 命令**, 它可以**对 SELECT 语句的执行计划进行分析**, 并输出 SELECT 执行的详细信息, 以供开发人员针对性优化.

使用explain这个命令来查看一个这些SQL语句的执行计划，查看该SQL语句有没有使用上了索引，有没有做全表扫描，这都可以通过explain命令来查看。

可以通过explain命令深入了解MySQL的基于开销的优化器，还可以获得很多可能被优化器考虑到的访问策略的细节，以及当运行SQL语句时哪种策略预计会被优化器采用。

EXPLAIN 命令用法十分简单, 在 SELECT 语句前加上 explain 就可以了, 例如:

```
mysql> explain select * from user;
+----+-------------+-------+------+---------------+------+---------+------+------+-------+
| id | select_type | table | type | possible_keys | key  | key_len | ref  | rows | Extra |
+----+-------------+-------+------+---------------+------+---------+------+------+-------+
|  1 | SIMPLE      | user  | ALL  | NULL          | NULL | NULL    | NULL |    7 | NULL  |
+----+-------------+-------+------+---------------+------+---------+------+------+-------+
1 row in set (0.00 sec)
```

参数说明

expain出来的信息有10列，分别是

```
id、select_type、table、type、possible_keys、key、key_len、ref、rows、Extra
```

案例表

```
--用户表
create table tuser(
id int primary key,
loginname varchar(100),
name varchar(100),
age int,
sex char(1),
dep int,
address varchar(100)
);
--部门表
create table tdep(
id int primary key,
name varchar(100)
);

--地址表
create table taddr(
id int primary key,
addr varchar(100)
);

--创建普通索引
mysql> alter table tuser add index idx_dep(dep);
--创建唯一索引
mysql> alter table tuser add unique index idx_loginname(loginname);
--创建组合索引
mysql> alter table tuser add index idx_name_age_sex(name,age,sex);
--创建全文索引
mysql> alter table taddr add fulltext ft_addr(addr);
```

id

- 每个 SELECT语句都会自动分配的一个唯一标识符.
- 表示查询中操作表的顺序，有三种情况：
  - id相同：执行顺序由上到下
  - id不同：如果是子查询，id号会自增，**id越大，优先级越高**。
  - id相同的不同的同时存在
- id列为null的就表示这是一个结果集，不需要使用它来进行查询。

select_type（重要）

查询类型，主要用于区别**普通查询、联合查询(union、union all)、子查询等复杂查询**。

simple

表示不需要union操作或者不包含子查询的简单select查询。有连接查询时，外层的查询为simple，且只有一个

```
mysql> explain select * from tuser;
+----+-------------+-------+------+---------------+------+---------+------+---
---+-------+
| id | select_type | table | type | possible_keys | key  | key_len | ref  |
rows | Extra |
+----+-------------+-------+------+---------------+------+---------+------+---
---+-------+
|  1 | SIMPLE      | tuser | ALL  | NULL          | NULL | NULL    | NULL |
 1 | NULL  |
+----+-------------+-------+------+---------------+------+---------+------+---
---+-------+
```

primary

一个需要union操作或者含有子查询的select，位于最外层的单位查询的select_type即为primary。且只有一个

```
mysql> explain select (select name from tuser) from tuser ;
+----+-------------+-------+-------+---------------+-----------------+-------
--+------+------+-------------+
| id | select_type | table | type  | possible_keys | key             |
key_len | ref  | rows | Extra       |
+----+-------------+-------+-------+---------------+-----------------+-------
--+------+------+-------------+
|  1 | PRIMARY     | tuser | index | NULL          | idx_dep         | 5
  | NULL |    1 | Using index |
|  2 | SUBQUERY    | tuser | index | NULL          | idx_name_age_sex | 312
  | NULL |    1 | Using index |
+----+-------------+-------+-------+---------------+-----------------+-------
--+------+------+-------------+
```

subquery

除了from字句中包含的子查询外，其他地方出现的子查询都可能是subquery

```
mysql> explain select * from tuser where  id = (select max(id) from tuser);
                                                +----+-------------+---
----+-------+---------------+---------+---------+-------+------+-------------
----------------+
| id | select_type | table | type  | possible_keys | key     | key_len | ref
| rows | Extra                       |
+----+-------------+-------+-------+---------------+---------+---------+------
-+------+-----------------------------+
|  1 | PRIMARY     | tuser | const | PRIMARY       | PRIMARY | 4       | const
|    1 | NULL                        |
|  2 | SUBQUERY    | NULL  | NULL  | NULL          | NULL    | NULL    | NULL
| NULL | Select tables optimized away |
+----+-------------+-------+-------+---------------+---------+---------+------
-+------+-----------------------------+
```

dependent subquery

与dependent union类似，表示这个subquery的查询要受到外部表查询的影响

```
mysql> explain select id,name,(select name from tdep a where a.id=b.dep) from
tuser b;
+----+--------------------+-------+--------+---------------+---------+--------
-+-------------+------+-------+
| id | select_type        | table | type   | possible_keys | key     | key_len
| ref         | rows | Extra |
+----+--------------------+-------+--------+---------------+---------+--------
-+-------------+------+-------+
|  1 | PRIMARY            | b     | ALL    | NULL          | NULL    | NULL
| NULL        |    2 | NULL  |
|  2 | DEPENDENT SUBQUERY | a     | eq_ref | PRIMARY       | PRIMARY | 4
| demo1.b.dep |    1 | NULL  |
+----+--------------------+-------+--------+---------------+---------+--------
-+-------------+------+-------+
```

union

union连接的两个select查询，第一个查询是PRIMARY，除了第一个表外，第二个以后的表select_type都是union

```
mysql> explain select * from tuser where sex='1'  union select * from tuser
where sex='2';
+----+--------------+------------+------+---------------+------+---------+----
--+------+-----------------+
| id | select_type  | table      | type | possible_keys | key  | key_len | ref
| rows | Extra           |
+----+--------------+------------+------+---------------+------+---------+----
--+------+-----------------+
|  1 | PRIMARY      | tuser      | ALL  | NULL          | NULL | NULL    |
NULL |    2 | Using where     |
|  2 | UNION        | tuser      | ALL  | NULL          | NULL | NULL    |
NULL |    2 | Using where     |
| NULL | UNION RESULT | <union1,2> | ALL  | NULL          | NULL | NULL    |
NULL | NULL | Using temporary |
+----+--------------+------------+------+---------------+------+---------+----
--+------+-----------------+
```

dependent union

与union一样，出现在union 或union all语句中，但是这个查询要受到外部查询的影响

```
mysql> explain select * from tuser where sex in (select sex from tuser where
sex='1' union select sex from tuser where sex='2');
+----+--------------------+------------+-------+---------------+-------------
----+---------+------+------+--------------------------+
| id | select_type        | table      | type  | possible_keys | key
   | key_len | ref  | rows | Extra                    |
+----+--------------------+------------+-------+---------------+-------------
----+---------+------+------+--------------------------+
|  1 | PRIMARY            | tuser      | ALL   | NULL          | NULL
    | NULL    | NULL |    2 | Using where              |
|  2 | DEPENDENT SUBQUERY | tuser      | index | NULL          |
idx_name_age_sex | 312     | NULL |    2 | Using where; Using index |
|  3 | DEPENDENT UNION    | tuser      | index | NULL          |
idx_name_age_sex | 312     | NULL |    2 | Using where; Using index |
| NULL | UNION RESULT       | <union2,3> | ALL   | NULL          | NULL
      | NULL    | NULL | NULL | Using temporary          |
+----+--------------------+------------+-------+---------------+-------------
----+---------+------+------+--------------------------+
```

union result

包含union的结果集，在union和union all语句中,因为它不需要参与查询，所以id字段为null

derived

from字句中出现的子查询，也叫做派生表，其他数据库中可能叫做内联视图或嵌套select

```
mysql> explain select * from (select * from tuser where sex='1') b;
+----+-------------+------------+------+---------------+------+---------+-----
-+------+-------------+
| id | select_type | table      | type | possible_keys | key  | key_len | ref
| rows | Extra       |
+----+-------------+------------+------+---------------+------+---------+-----
-+------+-------------+
|  1 | PRIMARY     | <derived2> | ALL  | NULL          | NULL | NULL    | NULL
|    2 | NULL        |
|  2 | DERIVED     | tuser      | ALL  | NULL          | NULL | NULL    | NULL
|    2 | Using where |
+----+-------------+------------+------+---------------+------+---------+-----
-+------+-------------+
```

table

- 显示的查询表名，如果查询使用了别名，那么这里显示的是别名
- 如果不涉及对数据表的操作，那么这显示为null
- 如果显示为尖括号括起来的就表示这个是临时表，后边的N就是执行计划中的id，表示结果来自于这个查询产生。

- 如果是尖括号括起来的**<union M,N>**，与类似，也是一个临时表，表示这个结果来自于union查询的id为M,N的结果集。

type（重要）

- 依次从好到差：

```
system, const, eq_ref, ref, fulltext, ref_or_null, unique_subquery,
index_subquery, range, index_merge, index, ALL
```

除了all之外，其他的type都可以使用到索引，除了index_merge之外，其他的type只可以用到一个索引

- **注意事项：**

```
最少要索引使用到range级别。
```

system

表中只有一行数据或者是空表。

```
mysql> explain select * from (select * from tuser where id=1) a;
+----+-------------+------------+--------+---------------+---------+---------
+-------+------+-------+
| id | select_type | table      | type   | possible_keys | key     | key_len |
ref   | rows | Extra |
+----+-------------+------------+--------+---------------+---------+---------
+-------+------+-------+
|  1 | PRIMARY     | <derived2> | system | NULL          | NULL    | NULL    |
NULL  |    1 | NULL  |
|  2 | DERIVED     | tuser      | const  | PRIMARY       | PRIMARY | 4       |
const |    1 | NULL  |
+----+-------------+------------+--------+---------------+---------+---------
+-------+------+-------+
```

const（重要）

使用**唯一索引或者主键**，返回记录一定是1行记录的等值where条件时，通常type是const。其他数据库也叫做唯一索引扫描

```
mysql> explain select * from tuser where id=1;
+----+-------------+-------+-------+---------------+---------+---------+------
-+------+-------+
| id | select_type | table | type  | possible_keys | key     | key_len | ref
| rows | Extra |
+----+-------------+-------+-------+---------------+---------+---------+------
-+------+-------+
|  1 | SIMPLE      | tuser | const | PRIMARY       | PRIMARY | 4       | const
|    1 | NULL  |
+----+-------------+-------+-------+---------------+---------+---------+------
-+------+-------+

mysql> explain select * from tuser where loginname = 'zy';
+----+-------------+-------+-------+---------------+---------------+---------
+-------+------+-------+
| id | select_type | table | type  | possible_keys | key           | key_len |
ref   | rows | Extra |
+----+-------------+-------+-------+---------------+---------------+---------
+-------+------+-------+
|  1 | SIMPLE      | tuser | const | idx_loginname | idx_loginname | 303     |
const |    1 | NULL  |
+----+-------------+-------+-------+---------------+---------------+---------
+-------+------+-------+
```

eq_ref（重要）

关键字:连接字段**主键或者唯一性索引**。

此类型通常出现在多表的 join 查询, 表示对于前表的每一个结果, **都只能匹配到后表的一行结果**. 并且查询的比较操作通常是 '=', 查询效率较高.

```
mysql> explain select a.id from tuser a left join tdep b on a.dep=b.id;
+----+-------------+-------+--------+---------------+---------+---------+-----
--------+------+-------------+
| id | select_type | table | type   | possible_keys | key     | key_len | ref
       | rows | Extra       |
+----+-------------+-------+--------+---------------+---------+---------+-----
--------+------+-------------+
|  1 | SIMPLE      | a     | index  | NULL          | idx_dep | 5       | NULL
       |    2 | Using index |
|  1 | SIMPLE      | b     | eq_ref | PRIMARY       | PRIMARY | 4       |
demo1.a.dep |    1 | Using index |
+----+-------------+-------+--------+---------------+---------+---------+-----
--------+------+-------------+
```

ref（重要）

针对非唯一性索引，使用等值（=）查询非主键。或者是使用了**最左前缀规则索引的查询。**

```
--非唯一索引
mysql>  explain select * from tuser where dep=1;
+----+-------------+-------+------+---------------+---------+---------+-------
+------+-------+
| id | select_type | table | type | possible_keys | key     | key_len | ref
| rows | Extra |
+----+-------------+-------+------+---------------+---------+---------+-------
+------+-------+
|  1 | SIMPLE      | tuser | ref  | idx_dep       | idx_dep | 5       | const
|    1 | NULL  |
+----+-------------+-------+------+---------------+---------+---------+-------
+------+-------+
--等值非主键连接
mysql> explain select a.id from tuser a left join tdep b on a.name=b.name;
+----+-------------+-------+-------+---------------+-----------------+-------
--+-------------+------+-------------------------+
| id | select_type | table | type  | possible_keys | key             |
key_len | ref         | rows | Extra                   |
+----+-------------+-------+-------+---------------+-----------------+-------
--+-------------+------+-------------------------+
|  1 | SIMPLE      | a     | index | NULL          | idx_name_age_sex | 312
  | NULL        |    2 | Using index             |
|  1 | SIMPLE      | b     | ref   | ind_name      | ind_name        | 72
| demo1.a.name |    1 | Using where; Using index |
+----+-------------+-------+-------+---------------+-----------------+-------
--+-------------+------+-------------------------+
--最左前缀
mysql> explain select * from tuser where name = 'zhaoyun';
+----+-------------+-------+------+-----------------+-----------------+-----
----+-------+------+-----------------------+
| id | select_type | table | type | possible_keys   | key             |
key_len | ref   | rows | Extra                 |
+----+-------------+-------+------+-----------------+-----------------+-----
----+-------+------+-----------------------+
|  1 | SIMPLE      | tuser | ref  | idx_name_age_sex | idx_name_age_sex | 303
    | const |    1 | Using index condition |
+----+-------------+-------+------+-----------------+-----------------+-----
----+-------+------+-----------------------+

思考：explain select * from tuser where sex = '1';
```

fulltext

全文索引检索，要注意，全文索引的优先级很高，若全文索引和普通索引同时存在时，mysql不管代价，优先选择使用全文索引

```
mysql> explain select * from taddr where match(addr) against('bei');
+----+-------------+-------+----------+---------------+---------+---------+---
---+------+-------------+
| id | select_type | table | type     | possible_keys | key     | key_len |
ref  | rows | Extra       |
+----+-------------+-------+----------+---------------+---------+---------+---
---+------+-------------+
|  1 | SIMPLE      | tuser | fulltext | ft_addr       | ft_addr | 0       |
NULL |    1 | Using where |
+----+-------------+-------+----------+---------------+---------+---------+---
---+------+-------------+
```

ref_or_null

与ref方法类似，只是增加了null值的比较。实际用的不多。

unique_subquery

用于where中的in形式子查询，子查询返回不重复值唯一值

index_subquery

用于in形式子查询使用到了辅助索引或者in常数列表，子查询可能返回重复值，可以使用索引将子查询去重。

range（重要）

索引范围扫描，常见于使用>,<,is null,between ,in ,like等运算符的查询中。

```
mysql> explain select id from tuser where id>1;
+----+-------------+-------+-------+---------------+---------+---------+------
+------+-------------+
| id | select_type | table | type  | possible_keys | key     | key_len | ref
| rows | Extra       |
+----+-------------+-------+-------+---------------+---------+---------+------
+------+-------------+
|  1 | SIMPLE      | tuser | range | PRIMARY       | PRIMARY | 4       | NULL
|    1 | Using where |
+----+-------------+-------+-------+---------------+---------+---------+------
+------+-------------+

--like 前缀索引
mysql> explain select * from tuser where name like 'zhao%';
+----+-------------+-------+-------+-----------------+-----------------+----
-----+------+------+----------------------+
| id | select_type | table | type  | possible_keys   | key             |
key_len | ref  | rows | Extra                |
```

```
+----+-------------+-------+-------+-----------------+-----------------+----
-----+------+------+-----------------------+
|  1 | SIMPLE      | tuser | range | idx_name_age_sex | idx_name_age_sex | 303
    | NULL |    1 | Using index condition |
+----+-------------+-------+-------+-----------------+-----------------+----
-----+------+------+-----------------------+

注:  like '%z'    不使用索引
```

index_merge

表示查询使用了两个以上的索引，最后取交集或者并集，常见and ，or的条件使用了不同的索引，官方排序这个在ref_or_null之后，但是实际上由于要读取所个索引，性能可能大部分时间都不如range

index（重要）

关键字：条件是出现在索引树中的节点的。可能没有完全匹配索引。

索引全表扫描，把索引从头到尾扫一遍，常见于使用索引列就可以处理不需要读取数据文件的查询、可以使用索引排序或者分组的查询。

```
--单索引
mysql> explain select loginname from tuser;
+----+-------------+-------+-------+---------------+---------------+---------
+------+------+-------------+
| id | select_type | table | type  | possible_keys | key           | key_len |
ref  | rows | Extra       |
+----+-------------+-------+-------+---------------+---------------+---------
+------+------+-------------+
|  1 | SIMPLE      | tuser | index | NULL          | idx_loginname | 303     |
NULL |    2 | Using index |
+----+-------------+-------+-------+---------------+---------------+---------
+------+------+-------------+
--组合索引
mysql> explain select age from tuser;
+----+-------------+-------+-------+---------------+-----------------+-------
--+------+------+-------------+
| id | select_type | table | type  | possible_keys | key             |
key_len | ref  | rows | Extra       |
+----+-------------+-------+-------+---------------+-----------------+-------
--+------+------+-------------+
|  1 | SIMPLE      | tuser | index | NULL          | idx_name_age_sex | 312
  | NULL |    2 | Using index |
+----+-------------+-------+-------+---------------+-----------------+-------
--+------+------+-------------+

思考: explain select loginname,age from tuser;
```

覆盖索引

all（重要）

这个就是全表扫描数据文件，然后再**在server层进行过滤**返回符合要求的记录。

```
mysql>  explain select * from tuser;
+----+-------------+-------+------+---------------+------+---------+------+---
---+-------+
| id | select_type | table | type | possible_keys | key  | key_len | ref  |
rows | Extra |
+----+-------------+-------+------+---------------+------+---------+------+---
---+-------+
|  1 | SIMPLE      | tuser | ALL  | NULL          | NULL | NULL    | NULL |
 2 | NULL  |
+----+-------------+-------+------+---------------+------+---------+------+---
---+-------+
```

回表查询

possible_keys

此次查询中可能选用的索引，一个或多个

key

查询真正使用到的索引，select_type为index_merge时，这里可能出现两个以上的索引，其他的select_type这里只会出现一个。

key_len

- 用于处理查询的索引长度，如果是单列索引，那就整个索引长度算进去，如果是多列索引，那么查询不一定都能使用到所有的列，具体使用到了多少个列的索引，这里就会计算进去，没有使用到的列，这里不会计算进去。
- 留意下这个列的值，算一下你的多列索引总长度就知道有没有使用到所有的列了。
- *另外，key_len只计算where条件用到的索引长度，而排序和分组就算用到了索引，也不会计算到key_len中。*

ref

- 如果是使用的常数等值查询，这里会显示const
- 如果是连接查询，被驱动表的执行计划这里会显示驱动表的关联字段
- 如果是条件使用了表达式或者函数，或者条件列发生了内部隐式转换，这里可能显示为func

rows

这里是执行计划中估算的扫描行数，不是精确值（InnoDB不是精确的值，MyISAM是精确的值，主要原因是InnoDB里面使用了MVCC并发机制）

extra（重要）

这个列包含不适合在其他列中显示单十分重要的额外的信息，这个列可以显示的信息非常多，有几十种，常用的有

using temporary

- 表示使用了临时表存储中间结果。
- MySQL在对查询结果**order by和group by**时使用临时表
- 临时表可以是内存临时表和磁盘临时表，执行计划中看不出来，需要查看status变量，used_tmp_table，used_tmp_disk_table才能看出来。

```
mysql> explain select distinct a.id from tuser a,tdep b where a.dep=b.id;

+----+-------------+-------+--------+-----------------------------------
-----------+---------+---------+------------+------+----------------------
---------------------+
| id | select_type | table | type   | possible_keys
          | key     | key_len | ref        | rows | Extra
                    |
+----+-------------+-------+--------+-----------------------------------
-----------+---------+---------+------------+------+----------------------
---------------------+
|  1 | SIMPLE      | a     | index  |
PRIMARY,idx_loginname,idx_name_age_sex,idx_dep | idx_dep | 5       | NULL
      |    2 | Using where; Using index; Using temporary |
|  1 | SIMPLE      | b     | eq_ref | PRIMARY
          | PRIMARY | 4       | kkb2.a.dep |    1 | Using index; Distinct
                    |
+----+-------------+-------+--------+-----------------------------------
-----------+---------+---------+------------+------+----------------------
---------------------+
```

no tables used

不带from字句的查询或者From dual查询

使用not in()形式子查询或not exists运算符的连接查询，这种叫做反连接

即，一般连接查询是先查询内表，再查询外表，反连接就是先查询外表，再查询内表。

using filesort（重要）

- 排序时无法使用到索引时，就会出现这个。常见于order by和group by语句中

- 说明MySQL会使用一个外部的索引排序，而不是按照索引顺序进行读取。
- MySQL中无法利用索引完成的排序操作称为“文件排序”

```
mysql> explain select * from tuser order by address;
+----+-------------+-------+------+---------------+------+---------+------+---
---+----------------+
| id | select_type | table | type | possible_keys | key  | key_len | ref  |
rows | Extra          |
+----+-------------+-------+------+---------------+------+---------+------+---
---+----------------+
|  1 | SIMPLE      | tuser | ALL  | NULL          | NULL | NULL    | NULL |
2 | Using filesort |
+----+-------------+-------+------+---------------+------+---------+------+---
---+----------------+
```

using index（重要）

查询时**不需要回表查询**，直接通过索引就可以获取查询的数据。

- 表示相应的SELECT查询中使用到了**覆盖索引（Covering Index）**，避免访问表的数据行，效率不错！
- 如果同时出现Using Where，说明索引被用来执行查找索引键值
- 如果没有同时出现Using Where，表明索引用来读取数据而非执行查找动作。

```
mysql>  explain select name,age,sex from tuser ;
+----+-------------+-------+-------+---------------+-----------------+-------
--+------+------+-------------+
| id | select_type | table | type  | possible_keys | key             |
key_len | ref  | rows | Extra       |
+----+-------------+-------+-------+---------------+-----------------+-------
--+------+------+-------------+
|  1 | SIMPLE      | tuser | index | NULL          | idx_name_age_sex | 312
  | NULL |    2 | Using index |
+----+-------------+-------+-------+---------------+-----------------+-------
--+------+------+-------------+
全值匹配 覆盖索引
```

using where（重要）

- 表示存储引擎返回的记录并不是所有的都满足查询条件，需要在server层进行过滤。

```
--查询条件无索引
mysql> explain select * from tuser where address='beijing';
```

```
+----+-------------+-------+------+---------------+------+---------+------+---
---+-------------+
| id | select_type | table | type | possible_keys | key  | key_len | ref  |
rows | Extra       |
+----+-------------+-------+------+---------------+------+---------+------+---
---+-------------+
|  1 | SIMPLE      | tuser | ALL  | NULL          | NULL | NULL    | NULL |
 2 | Using where |
+----+-------------+-------+------+---------------+------+---------+------+---
---+-------------+
--索引失效
mysql>  explain select * from tuser where age=1;
+----+-------------+-------+------+---------------+------+---------+------+---
---+-------------+
| id | select_type | table | type | possible_keys | key  | key_len | ref  |
rows | Extra       |
+----+-------------+-------+------+---------------+------+---------+------+---
---+-------------+
|  1 | SIMPLE      | tuser | ALL  | NULL          | NULL | NULL    | NULL |
 2 | Using where |
+----+-------------+-------+------+---------------+------+---------+------+---
---+-------------+
mysql> explain select * from tuser where  id in(1,2);
+----+-------------+-------+-------+---------------+---------+---------+------
+------+-------------+
| id | select_type | table | type  | possible_keys | key     | key_len | ref
| rows | Extra       |
+----+-------------+-------+-------+---------------+---------+---------+------
+------+-------------+
|  1 | SIMPLE      | tuser | range | PRIMARY       | PRIMARY | 4       | NULL
|    2 | Using where |
+----+-------------+-------+-------+---------------+---------+---------+------
+------+-------------+
```

- 查询条件中分为限制条件和检查条件，5.6之前，存储引擎只能根据限制条件扫描数据并返回，然后server层根据检查条件进行过滤再返回真正符合查询的数据。5.6.x之后支持**ICP**特性，可以把检查条件也下推到存储引擎层，不符合检查条件和限制条件的数据，直接不读取，这样就大大减少了存储引擎扫描的记录数量。extra列显示using index condition

```
mysql> explain select * from tuser where name='asd';
+----+-------------+-------+------+-----------------+-----------------+-----
----+-------+------+-----------------------+
| id | select_type | table | type | possible_keys    | key             |
key_len | ref   | rows | Extra                 |
+----+-------------+-------+------+-----------------+-----------------+-----
----+-------+------+-----------------------+
|  1 | SIMPLE      | tuser | ref  | idx_name_age_sex | idx_name_age_sex | 303
    | const |    1 | Using index condition |
+----+-------------+-------+------+-----------------+-----------------+-----
----+-------+------+-----------------------+
```

firstmatch(tb_name)

5.6.x开始引入的优化子查询的新特性之一，常见于where字句含有in()类型的子查询。如果内表的数据量比较大，就可能出现这个

loosescan(m..n)

5.6.x之后引入的优化子查询的新特性之一，在in()类型的子查询中，子查询返回的可能有重复记录时，就可能出现这个

- 除了这些之外，还有很多查询数据字典库，执行计划过程中就发现不可能存在结果的一些提示信息

filtered

使用explain extended时会出现这个列，5.7之后的版本默认就有这个字段，不需要使用explain extended了。这个字段表示存储引擎返回的数据在server层过滤后，剩下多少满足查询的记录数量的比例，注意是百分比，不是具体记录数。

参考网站

https://segmentfault.com/a/1190000008131735

https://blog.csdn.net/rewiner120/article/details/70598797

二、MySQL锁篇

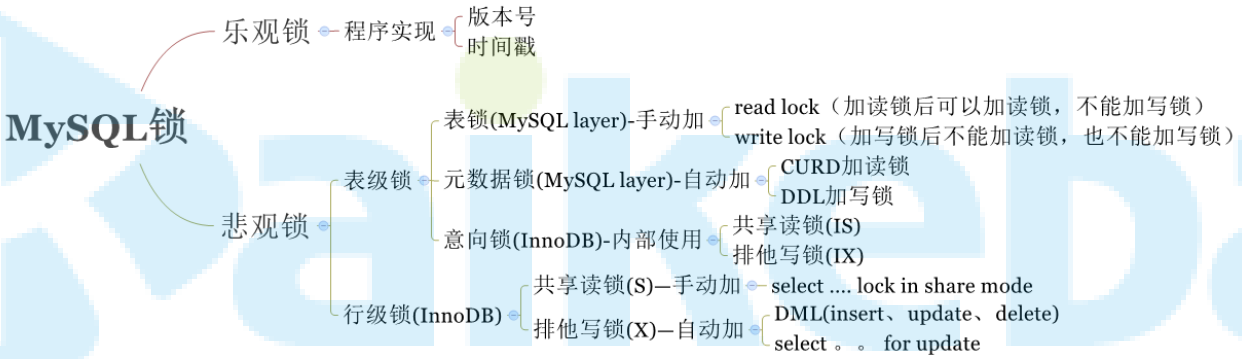
MySQL锁介绍

- 按照锁的粒度来说，MySQL主要包含三种类型（级别）的锁定机制：

```
- 全局锁：锁的是整个database。由MySQL的SQL layer层实现的

- 表级锁：锁的是某个table。由MySQL的SQL layer层实现的

- 行级锁：锁的是某行数据，也可能锁定行之间的间隙。由某些存储引擎实现，比如InnoDB。
```

- 按照锁的功能来说分为：**共享读锁**和**排他写锁**。
- 按照锁的实现方式分为：**悲观锁**和**乐观锁**（使用某一版本列或者唯一列进行逻辑控制）
- 表级锁和行级锁的区别：

```
表级锁：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；

行级锁：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；
```



MySQL表级锁

表级锁介绍

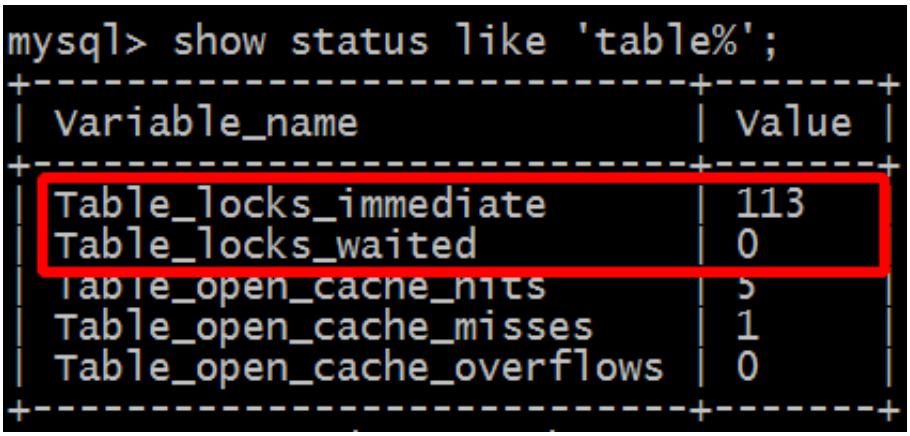
由MySQL SQL layer层实现

- MySQL的表级锁有两种：

```
一种是表锁。
一种是元数据锁（meta data lock，MDL）。
```

- MySQL 实现的表级锁定的争用状态变量：

```
mysql> show status like 'table%';
```




```
- table_locks_immediate：产生表级锁定的次数；
- table_locks_waited：出现表级锁定争用而发生等待的次数；
```

表锁介绍

- 表锁有两种表现形式：

```
表共享读锁（Table Read Lock）
表独占写锁（Table Write Lock）
```

- 手动增加表锁

```
lock table 表名称 read(write),表名称2 read(write)，其他；
```

- 查看表锁情况

```
show open tables;
```

- 删除表锁

```
unlock tables;
```

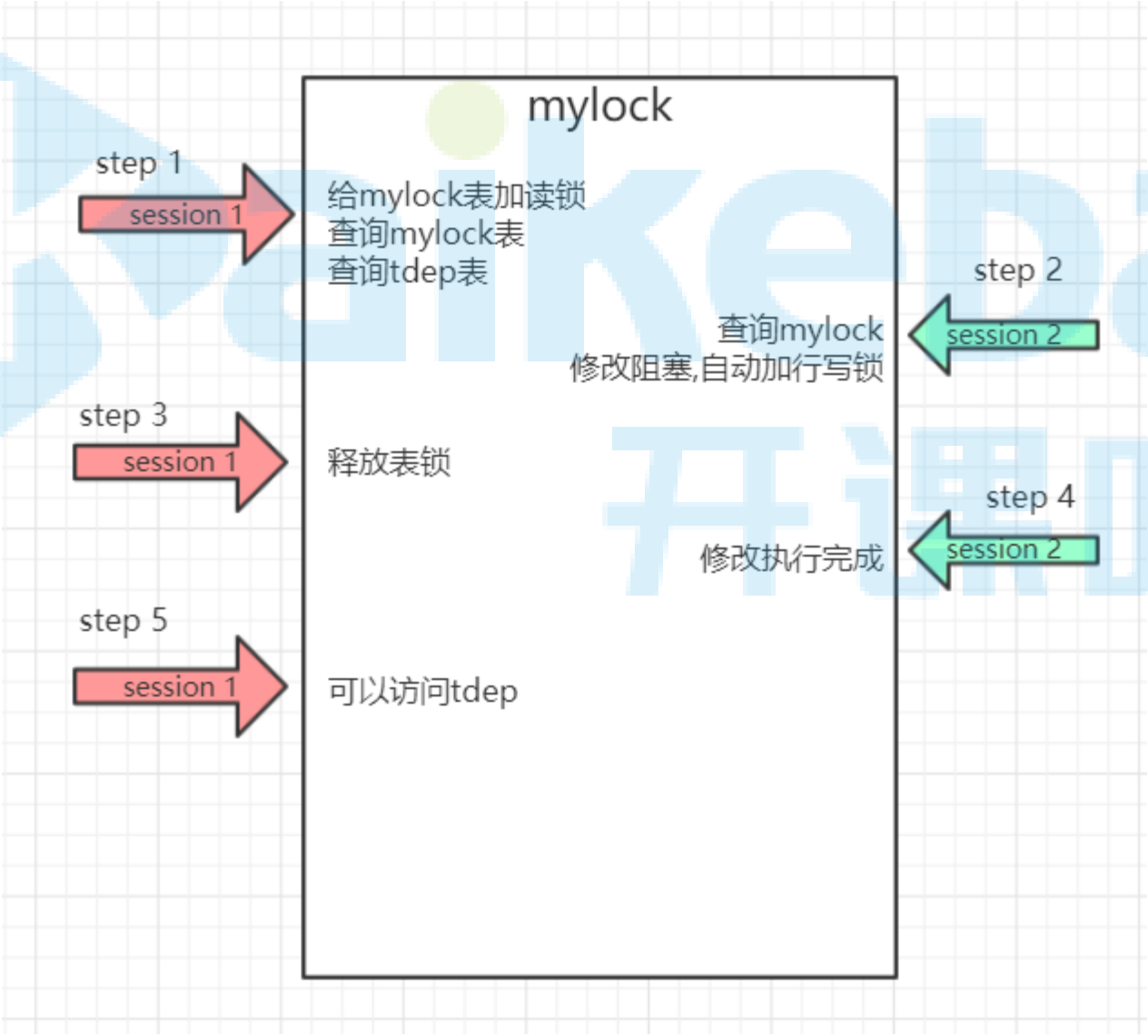
表锁演示

环境准备

```sql
--新建表
CREATE TABLE mylock (
id int(11) NOT NULL AUTO_INCREMENT,
NAME varchar(20) DEFAULT NULL,
PRIMARY KEY (id)
);
INSERT INTO mylock (id,NAME) VALUES (1, 'a');
INSERT INTO mylock (id,NAME) VALUES (2, 'b');
INSERT INTO mylock (id,NAME) VALUES (3, 'c');
INSERT INTO mylock (id,NAME) VALUES (4, 'd');
```

读锁演示

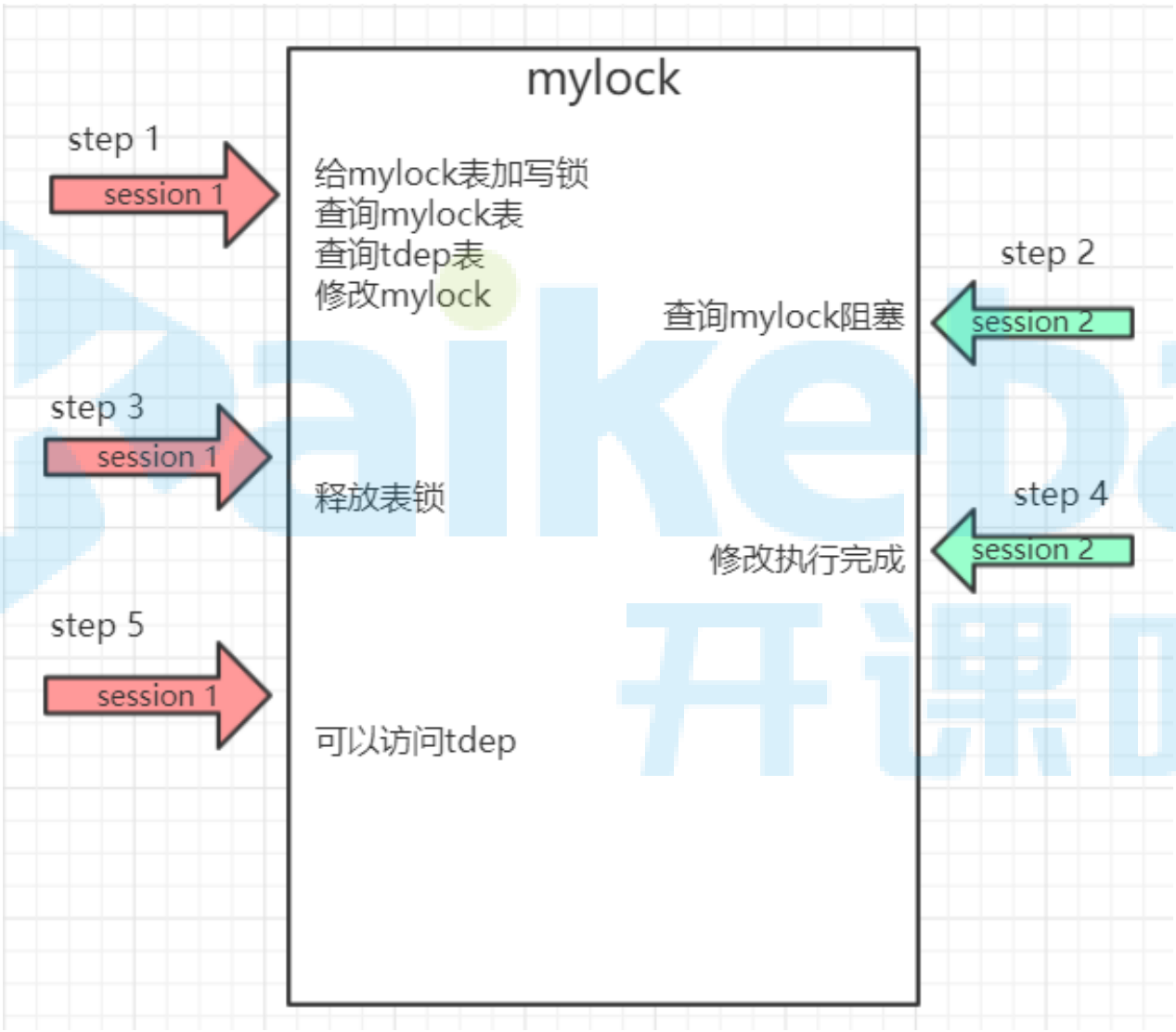
1、表读锁



session1（Navicat）、session2（mysql）

```
1、session1: lock table mylock read; -- 给mylock表加读锁
2、session1: select * from mylock; -- 可以查询
3、session1: select * from tdep; --不能访问非锁定表
4、session2: select * from mylock; -- 可以查询 没有锁
5、session2: update mylock set name='x' where id=2; -- 修改阻塞,自动加行写锁
6、session1: unlock tables; -- 释放表锁
7、session2: Rows matched: 1  Changed: 1  Warnings: 0 -- 修改执行完成
8、session1: select * from tdep; --可以访问
```

2、表写锁



session1（Navicat）、session2（mysql）

```
1、session1: lock table mylock write; -- 给mylock表加写锁
2、session1: select * from mylock; -- 可以查询
3、session1: select * from tdep; --不能访问非锁定表
4、session1: update mylock set name='y' where id=2; --可以执行
5、session2: select * from mylock; -- 查询阻塞
6、session1: unlock tables; -- 释放表锁
7、session2: 4 rows in set (22.57 sec) -- 查询执行完成
8、session1: select * from tdep; --可以访问
```

课堂主题

Mysql元数据锁、行锁、MySQL事务介绍、InnoDB架构

课堂目标

理解MySQL元数据锁的意义和使用场景

理解MySQL行锁的意义和使用场景

掌握MySQL记录锁和间隙锁的使用区别

掌握死锁的原理和死锁场景

理解事务的概念和四大特征（ACID）

掌握InnoDB的架构和组件作用

理解预写机制、双写机制、RedoLog和UndoLog的作用和日志落盘

知识要点

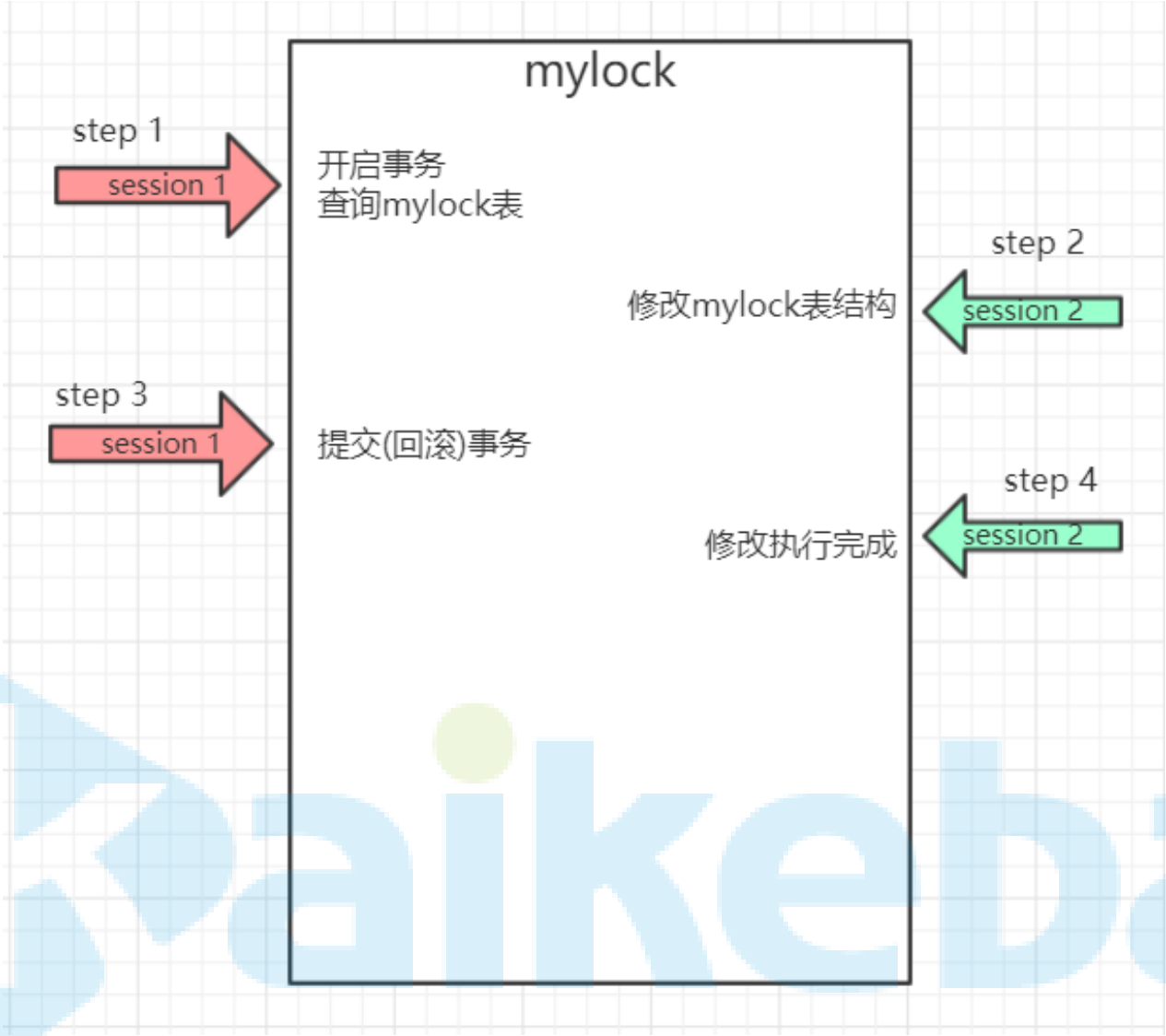
元数据锁介绍

MDL不需要显式使用，在访问一个表的时候会被自动加上。MDL的作用是，保证读写的正确性。你可以想象一下，如果一个查询正在遍历一个表中的数据，而执行期间另一个线程对这个表结构做变更，删了一列，那么查询线程拿到的结果跟表结构对不上，肯定是不行的。

因此，**在 MySQL 5.5 版本中引入了 MDL，当对一个表做增删改查操作的时候，加 MDL 读锁；当要对表做结构变更操作的时候，加 MDL 写锁。**

- 读锁之间不互斥，因此你可以有多个线程同时对一张表增删改查。
- 读写锁之间、写锁之间是互斥的，用来保证变更表结构操作的安全性。因此，如果有两个线程要同时给一个表加字段，其中一个要等另一个执行完才能开始执行。

元数据锁演示



session1（Navicat）、session2（mysql）

```
1、session1: begin;--开启事务
        select * from mylock;--加MDL读锁
2、session2: alter table mylock add f int; -- 修改阻塞
3、session1: commit; --提交事务 或者 rollback 释放读锁
4、session2: Query OK, 0 rows affected (38.67 sec)  --修改完成
        Records: 0  Duplicates: 0  Warnings: 0
```

我们可以看到 session A 先启动，这时候会对表 t 加一个 MDL 读锁。由于 session B 需要的也是 MDL 读锁，因此可以正常执行。

之后 session C 会被 blocked，是因为 session A 的 MDL 读锁还没有释放，而 session C 需要 MDL 写锁，因此只能被阻塞。

如果只有 session C 自己被阻塞还没什么关系，但是之后所有要在表 t 上新申请 MDL 读锁的请求也会被 session C 阻塞。前面我们说了，所有对表的增删改查操作都需要先申请 MDL 读锁，就都被锁住，等于这个表现在完全不可读写了。

你现在应该知道了，事务中的 MDL 锁，在语句执行开始时申请，但是语句结束后并不会马上释放，而会等到整个事务提交后再释放。

MySQL行级锁

行级锁介绍

MySQL的行级锁，是由存储引擎来实现的，利用存储引擎锁住索引项来实现的。这里我们主要讲解InnoDB的行级锁。

- InnoDB的行级锁，按照锁定范围来说，分为三种：

```
- 记录锁（Record Locks）:锁定索引中一条记录。 id=1

- 间隙锁（Gap Locks）:要么锁住索引记录中间的值，要么锁住第一个索引记录前面的值或者最后一个索引记录后面的值。

- Next-Key Locks:是索引记录上的记录锁和在索引记录之前的间隙锁的组合。
```

- InnoDB的行级锁，按照功能来说，分为两种： RR

```
 - 共享锁（S）：允许一个事务去读一行，阻止其他事务获得相同数据集的排他锁。
 - 排他锁（X）：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读锁（不是读）和排他写锁。
```

对于UPDATE、DELETE和INSERT语句，InnoDB会自动给涉及数据集加排他锁（X);

对于普通SELECT语句，InnoDB不会加任何锁，事务可以通过以下语句显示给记录集加共享锁或排他锁。 **手动添加共享锁（S）：**

```
SELECT * FROM table_name WHERE ... LOCK IN SHARE MODE
```

手动添加排他锁（x）：

```
SELECT * FROM table_name WHERE ... FOR UPDATE
```

- InnoDB也实现了表级锁，也就是意向锁，意向锁是mysql内部使用的，不需要用户干预。

- 意向共享锁（IS）：事务打算给数据行加行共享锁，事务在给一个数据行加共享锁前必须先取得该表的IS锁。
- 意向排他锁（IX）：事务打算给数据行加行排他锁，事务在给一个数据行加排他锁前必须先取得该表的IX锁。

- 意向锁和行锁可以共存，意向锁的主要作用是为了【全表更新数据】时的性能提升。否则在全表更新数据时，需要先检索该表是否某些记录上面有行锁。

	共享锁（S）	**排他锁（X）**	**意向共享锁（IS）**	**意向排他锁（IX）**
共享锁（S）	兼容	冲突	兼容	冲突
排他锁（X）	冲突	冲突	冲突	冲突
意向共享锁（IS）	兼容	冲突	兼容	兼容
意向排他锁（IX）	冲突	冲突	兼容	兼容

- **InnoDB行锁**是通过给索引上的**索引项加锁来实现的**，因此InnoDB这种行锁实现特点意味着：只有通过索引条件检索的数据，InnoDB才使用行级锁，否则，InnoDB将使用表锁！

- Innodb所使用的**行级锁定**争用状态查看：

```
mysql> show status like 'innodb_row_lock%';
```

```
mysql> show status like 'innodb_row_lock%';
+-------------------------------+-------+
| Variable_name                 | Value |
+-------------------------------+-------+
| Innodb_row_lock_current_waits | 0     |
| Innodb_row_lock_time          | 0     |
| Innodb_row_lock_time_avg      | 0     |
| Innodb_row_lock_time_max      | 0     |
| Innodb_row_lock_waits         | 0     |
+-------------------------------+-------+
5 rows in set (0.00 sec)
```

```
- 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；

- Innodb_row_lock_time_avg：每次等待所花平均时间；

- 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；
```

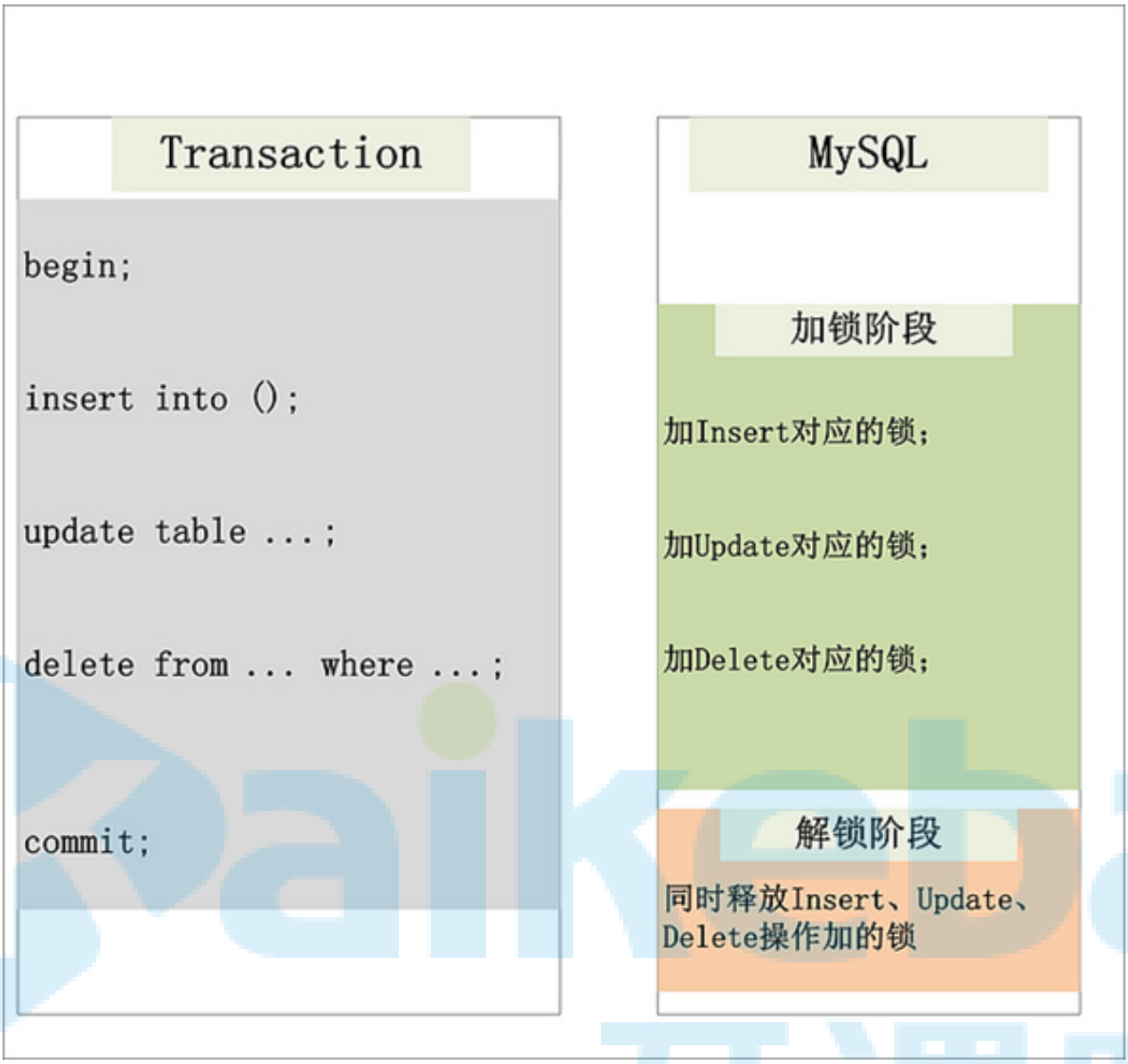
对于这5个状态变量，比较重要的主要是：

```
- Innodb_row_lock_time_avg（等待平均时长）

- Innodb_row_lock_waits（等待总次数）

- Innodb_row_lock_time（等待总时长）这三项。
```

尤其是当等待次数很高，而且每次等待时长也不小的时候，我们就需要分析系统中为什么会有如此多的等待，然后根据分析结果着手指定优化计划。

两阶段锁

传统RDBMS加锁的一个原则，就是2PL (Two-Phase Locking，二阶段锁)。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。



从上图可以看出，2PL就是将加锁/解锁分为两个完全不相交的阶段。

> 加锁阶段：只加锁，不放锁。
>
> 解锁阶段：只放锁，不加锁。

InnoDB行锁演示

行读锁

session1（Navicat）、session2（mysql）

```
查看行锁状态  show STATUS like 'innodb_row_lock%';
1、session1: begin;--开启事务未提交
          select * from  mylock  where ID=1 lock in share mode; --手动加id=1的行读锁,使用索引
2、session2: update mylock set name='y' where id=2; -- 未锁定该行可以修改
3、session2: update mylock set name='y' where id=1; -- 锁定该行修改阻塞
  ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction
 -- 锁定超时
4、session1: commit; --提交事务 或者 rollback 释放读锁
5、session2: update mylock set name='y' where id=1; --修改成功
             Query OK, 1 row affected (0.00 sec)
             Rows matched: 1  Changed: 1  Warnings: 0

注：使用索引加行锁 ，未锁定的行可以访问
```

行读锁升级为表锁

session1（Navicat）、session2（mysql）

```
1、session1: begin;--开启事务未提交
             --手动加name='c'的行读锁,未使用索引
             select * from  mylock  where name='c' lock in share mode;
2、session2: update mylock set name='y' where id=2; -- 修改阻塞 未用索引行锁升级为表锁
3、session1: commit; --提交事务 或者 rollback 释放读锁
4、session2: update mylock set name='y' where id=2; --修改成功
             Query OK, 1 row affected (0.00 sec)
             Rows matched: 1  Changed: 1  Warnings: 0

注：未使用索引行锁升级为表锁
```

行写锁

session1（Navicat）、session2（mysql）

```
1、session1: begin;--开启事务未提交
             --手动加id=1的行写锁,
             select * from  mylock  where id=1 for update;

2、session2: select * from  mylock  where id=2 ; -- 可以访问
3、session2: select * from  mylock  where id=1 ; -- 可以读 不加锁
    4、session2: select * from  mylock  where id=1 lock in share mode ; -- 加读锁被阻塞
5、session1: commit; -- 提交事务 或者 rollback 释放写锁
5、session2: 执行成功

主键索引产生记录锁
```

间隙锁

id(主键)	number（二级索引）
1	2
3	4
6	5
8	5
10	5
13	11

间隙锁有两种情况

1、防止插入间隙内的数据

2、防止已有数据更新为间隙内的数据

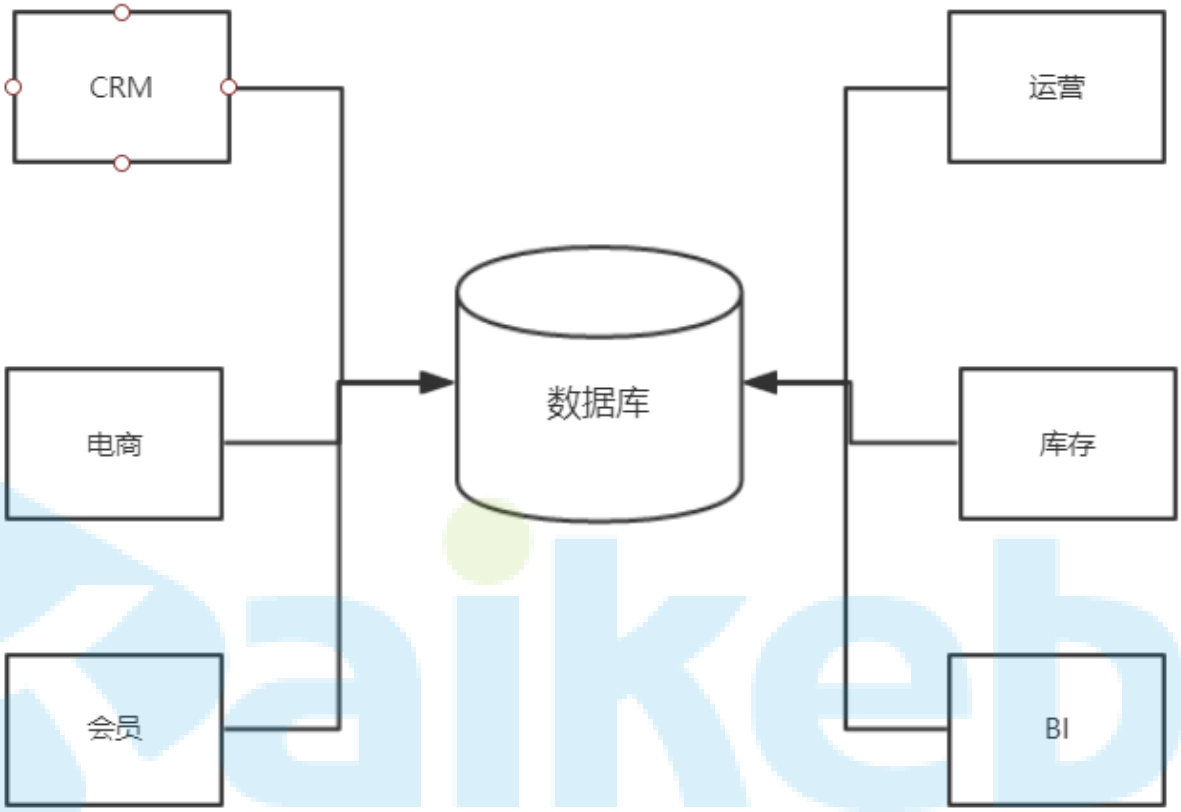
session1（Navicat）、session2（mysql）

案例演示：

```
mysql> create table news (id int, number int,primary key (id));
mysql> insert into news values(1,2);
......
--加非唯一索引
mysql> alter table news add index idx_num(number);
session 1:
start  transaction ;
select  * from news where number=4 for update ;

session 2:
start  transaction ;
insert into news value(2,4);#（阻塞）
insert into news value(2,2);#（阻塞）
insert into news value(4,4);#（阻塞）
insert into news value(4,5);#（阻塞）
insert into news value(7,5);#（执行成功）
insert into news value(9,5);#（执行成功）
insert into news value(11,5);#（执行成功）
````
```

注：id和number都在间隙内则阻塞。

```
session 1:
start  transaction ;
select  * from news where number=13 for update ;
 ( select * from news where id>1 and id < 8 for update;)
session 2:
start  transaction ;
insert into news value(11,5);#(执行成功)
insert into news value(12,11);#(执行成功)
insert into news value(14,11);#(阻塞)
insert into news value(15,12);#(阻塞)
```

检索条件number=13,向左取得最靠近的值11作为左区间，向右由于没有记录因此取得无穷大作为右区间，因此，session 1的间隙锁的范围（11，无穷大）

注：非主键索引产生间隙锁，主键范围
```

三、MySQL分库分表篇

传统项目结构



数据库性能瓶颈：

1、数据库连接数有限

MySQL数据库默认100个连接、单机最大1500连接。

2、表数据量

1、表数量多，成百上千

2、单表数据，千万级别

3、索引，命中率问题，索引存磁盘，占空间

3、硬件问题

性能指标：单表QPS、TPS

数据库性能优化

1、参数优化

2、缓存、索引

3、读写分离

4、分库分表

分库分表介绍

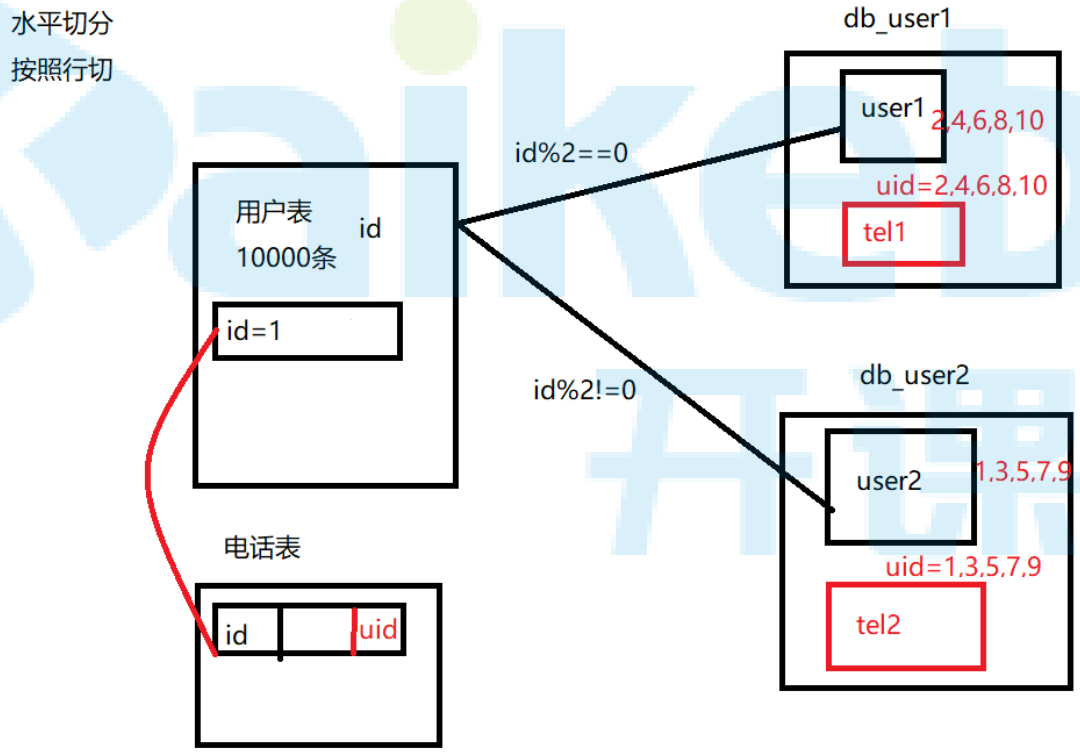
使用背景

- 当【表的数量】达到了几百上千张表时，众多的业务模块都访问这个数据库，压力会比较大，考虑对其进行分库。
- 当【表的数据】达到了几千万级别，在做很多操作都比较吃力,所以，考虑对其进行分库或者分表

数据切分（sharding）方案

数据的切分（Sharding）根据其切分规则的类型，可以分为两种切分模式：

- **垂直切分**：按照业务模块进行切分，将不同模块的表切分到不同的数据库中。
- **水平切分**：将一张大表按照一定的切分规则，按照**行**切分成不同的表或者切分到不同的库中。



水平切分规则

常用的切分规则有以下几种：

- **按照ID取模**：对ID进行取模，余数决定该行数据切分到哪个表或者库中
- **按照日期**：按照年月日，将数据切分到不同的表或者库中
- **按照范围**：可以对某一列按照范围进行切分，不同的范围切分到不同的表或者数据库中。

切分原则

- **第一原则**：能不切分尽量不要切分。

- **第二原则**：如果要切分一定要选择合适的切分规则，提前规划好。
- **第三原则**：数据切分尽量通过数据冗余或表分组（Table Group）来**降低跨库 Join** 的可能。
- **第四原则**：由于数据库中间件对数据 Join 实现的优劣难以把握，而且实现高性能难度极大，业务读取尽量少使用多表 Join。

分库分表需要解决的问题

分布式事务问题

```
本地事务：ACID
分布式事务：根据百度百科的定义，CAP定理又称CAP原则，指的是在一个分布式系统中，
Consistency（一致性）、 Availability（可用性）、Partition tolerance（分区容错性）。一
致性是强一致性。CAP理论最多只能同时满足两个。
BASE：基本可用+软状态+最终一致性
```

- 强一致性事务（同步）
- 最终一致性事务（异步思想） 利用记录日志等方式

分布式主键ID问题

- redis incr命令
- 数据库（生成主键）
- UUID
- snowflake算法（https://www.sohu.com/a/232008315_453160）

跨库join问题

- 通过业务分析，将不同库的join查询拆分成多个select
- 建立全局表（每个库都有一个相同的表）
- 冗余字段（不符合数据库三范式）
- E-R分片（将有ER关系的记录都存储到一个库中）
- 最多支持跨两张表跨库的join

跨库count、order by、group by问题

分库分表实现技术

- 阿里的TDDL、Cobar
- 基于阿里Cobar开发的Mycat
- 当当网的sharding-jdbc

课堂主题

MyCat介绍、分片规则、案例展示

课堂目标

理解MyCat的架构和原理

会安装和启动MyCat

会配置MyCat

熟悉常见分片规则和优劣

掌握mycat分库分表的方法

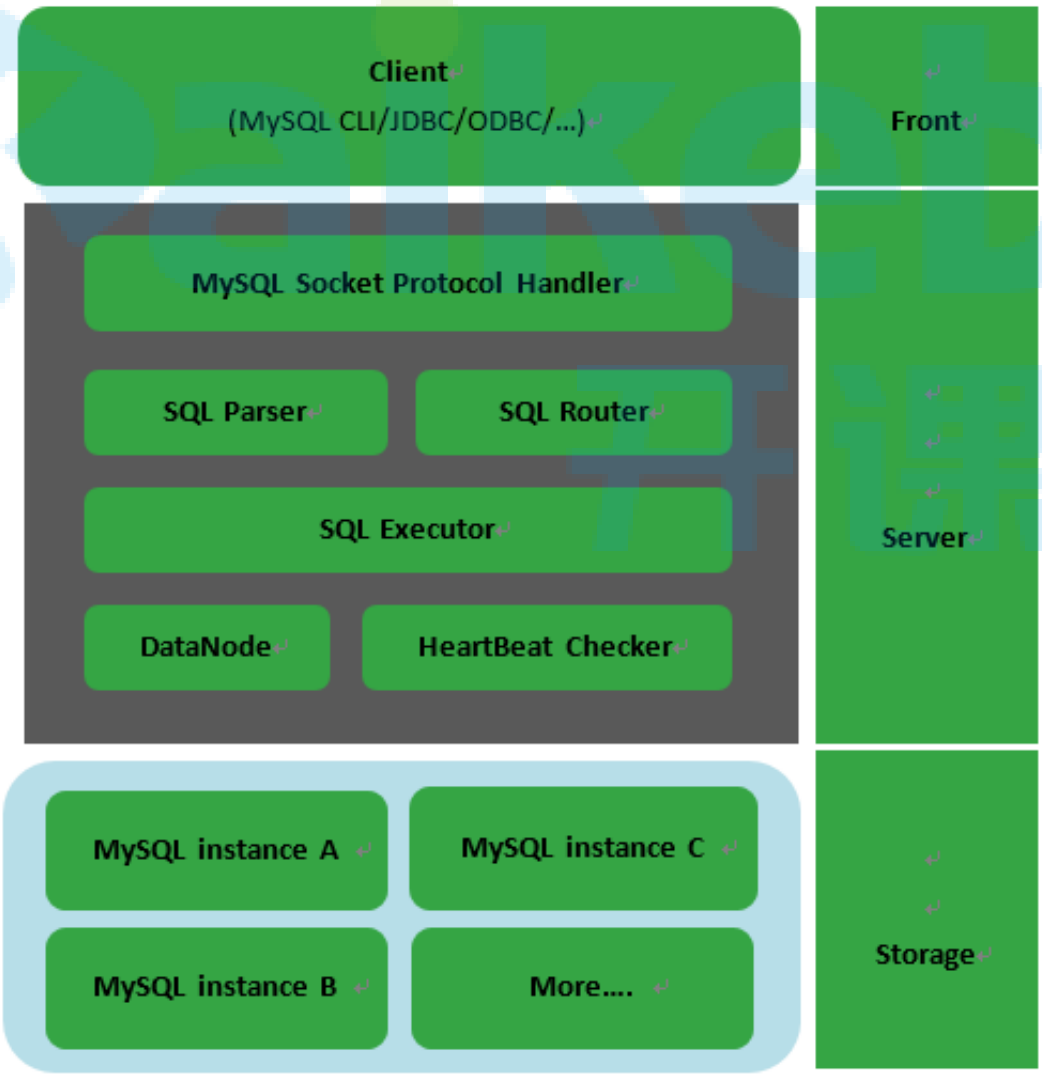
Mycat介绍

什么是Mycat?

官方网站：http://www.mycat.org.cn/ http://www.mycat.io/

db proxy Mycat

Mycat架构



Mycat核心概念

- **Schema**：由它指定逻辑数据库（相当于MySQL的database数据库）
- **Table**：逻辑表（相当于MySQL的table表）
- **DataNode**：真正存储数据的物理节点
- **DataHost**：存储节点所在的数据库主机（指定MySQL数据库的连接信息）
- **User**：MyCat的用户（类似于MySQL的用户，支持多用户）

Mycat主要解决的问题

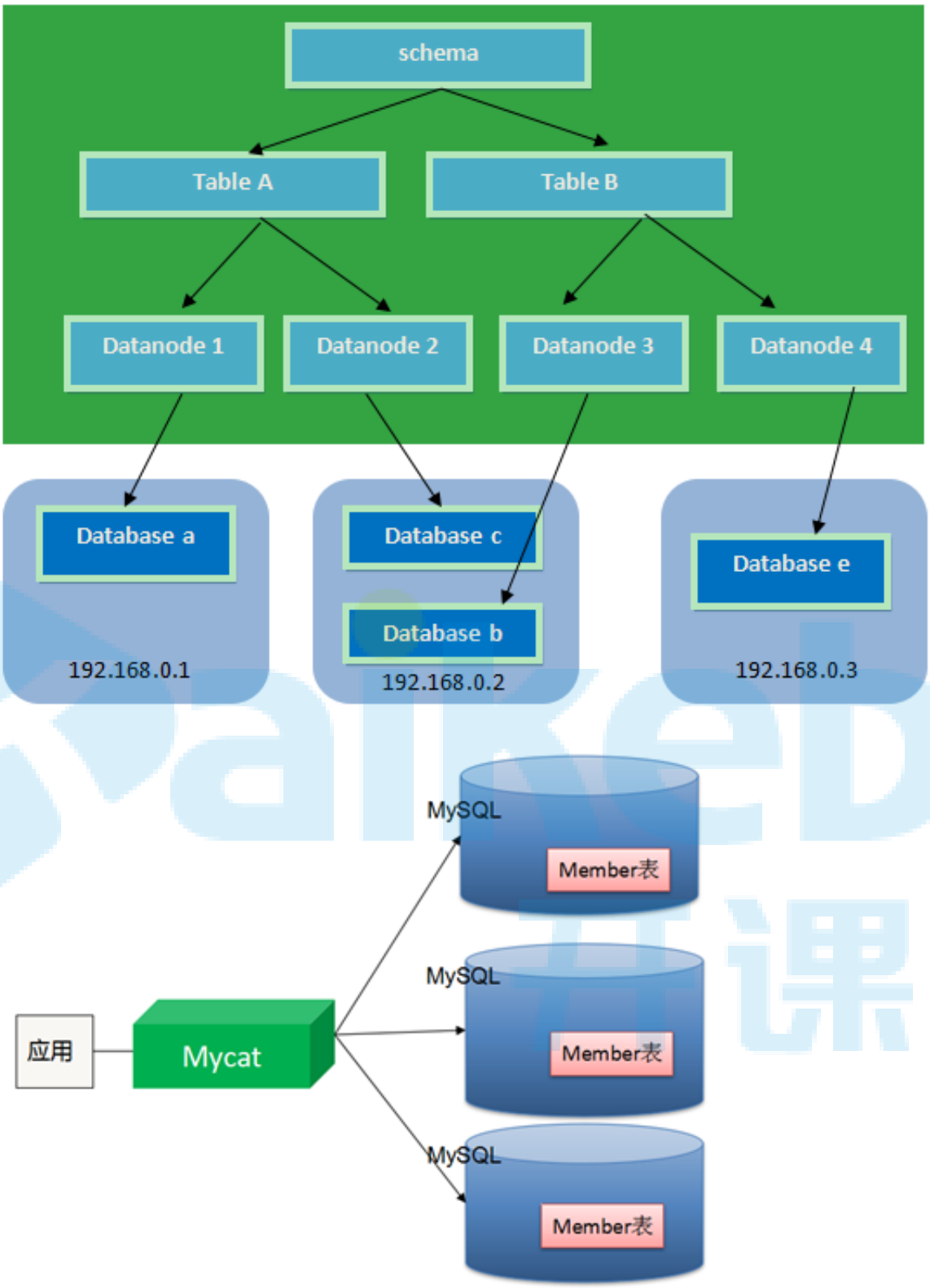
- **海量数据存储**
- **查询优化**

Mycat对多数据库的支持

Mycat分片策略

MyCAT支持水平分片与垂直分片：

- 水平分片：一个表格的数据分割到多个节点上，按照行分隔。
- 垂直分片：一个数据库中多个表格A，B，C，A存储到节点1上，B存储到节点2上，C存储到节点3上。



MyCAT通过定义表的分片规则来实现分片，每个表格可以捆绑一个分片规则，每个分片规则指定一个分片字段并绑定一个函数，来实现动态分片算法。

1.**Schema**：逻辑库，与MySQL中的Database（数据库）对应，一个逻辑库中定义了所包括的Table。

2.**Table**：表，即物理数据库中存储的某一张表，与传统数据库不同，这里的表格需要声明其所存储的逻辑数据节点DataNode。**在此可以指定表的分片规则。**

3.**DataNode**：MyCAT的逻辑数据节点，是存放table的具体物理节点，也称之为分片节点，通过DataHost来关联到后端某个具体数据库上

4.**DataHost**：定义某个物理库的访问地址，用于捆绑到Datanode上

Mycat安装

注意：需要先安装jdk（操作系统如果是64位，必须安装64位的JDK）

- 第一步：下载MyCat

```
wget http://dl.mycat.io/1.6-RELEASE/Mycat-server-1.6-RELEASE-20161028204710-linux.tar.gz
```

- 第二步：解压缩，得到mycat目录

```
tar -zxvf Mycat-server-1.6-RELEASE-20161028204710-linux.tar.gz
```

- 第三步：进入mycat/bin，启动MyCat

```
- 启动命令：./mycat start
- 停止命令：./mycat stop
- 重启命令：./mycat restart
- 查看状态：./mycat status
```

- 第四步：访问Mycat

```
使用mysql的客户端直接连接mycat服务。默认服务端口为【8066】
mysql -uroot -p123456 -h127.0.0.1 -P8066
```

Mycat分片

配置schema.xml

schema.xml介绍

schema.xml作为Mycat中重要的配置文件之一，**管理着Mycat的逻辑库、表、分片规则、DataNode以及DataHost之间的映射关系。**弄懂这些配置，是正确使用Mycat的前提。

- schema 标签用于定义MyCat实例中的逻辑库
- Table 标签定义了MyCat中的逻辑表
- dataNode 标签定义了MyCat中的数据节点，也就是我们通常说所的数据分片。
- dataHost标签在mycat逻辑库中也是作为最底层的标签存在，直接定义了具体的数据库实例、读写分离配置和心跳语句。

schema.xml配置

```
<?xml version="1.0"?>
<!DOCTYPE mycat:schema SYSTEM "schema.dtd">
<mycat:schema xmlns:mycat="http://io.mycat/">
  <!--
    schema : 逻辑库  name:逻辑库名称
    sqlMaxLimit：一次取多少条数据  要超过用limit xxx
    table:逻辑表
    dataNode:数据节点 对应datanode标签
    rule：分片规则，对应rule.xml
    subTables:子表
    primaryKey：分片主键 可缓存
  -->
  <schema name="TESTDB" checkSQLschema="false" sqlMaxLimit="100">
    <!-- auto sharding by id (long) -->
    <table name="item" dataNode="dn1,dn2,dn3" rule="mod-long"
primaryKey="ID"/>
  </schema>
  <!-- <dataNode name="dn1$0-743" dataHost="localhost1" database="db$0-743"
    /> -->
  <dataNode name="dn1" dataHost="localhost1" database="db1" />
  <dataNode name="dn2" dataHost="localhost1" database="db2" />
  <dataNode name="dn3" dataHost="localhost1" database="db3" />
  <!--
    dataHost : 数据主机（节点主机）
    balance：1 ：读写分离  0 ：  读写不分离
    writeType：0 第一个writeHost写， 1 随机writeHost写
    dbDriver：  数据库驱动 native：MySQL  JDBC：Oracle、SQLServer
    switchType：  是否主动读
    1、主从自动切换 -1 不切换 2 当从机延时超过slaveThreshold值时切换为主读
  -->
  <dataHost name="localhost1" maxCon="1000" minCon="10" balance="0"
        writeType="0" dbType="mysql" dbDriver="native" switchType="1"
 slaveThreshold="100">
    <heartbeat>select user()</heartbeat>

    <writeHost host="hostM1" url="192.168.24.129:3306" user="root"
          password="root" >
    </writeHost>
  </dataHost>

</mycat:schema>
```

配置Server.xml

server.xml介绍

server.xml几乎保存了所有mycat需要的系统配置信息。最常用的是在此配置用户名、密码及权限。

server.xml配置

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mycat:server SYSTEM "server.dtd">
<mycat:server xmlns:mycat="http://io.mycat/">
     <system>
          <property name="defaultSqlParser">druidparser</property>
     </system>
     <user name="mycat">
          <property name="password">mycat</property>
          <property name="schemas">TESTDB</property>
     </user>
</mycat:server>
```

配置rule.xml

rule.xml里面就定义了我们对表进行拆分所涉及到的规则定义。我们可以灵活的对表使用不同的分片算法，或者对表使用相同的算法但具体的参数不同。这个文件里面主要有tableRule和function这两个标签。在具体使用过程中可以按照需求添加tableRule和function。

此配置文件可以不用修改，使用默认即可。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mycat:rule SYSTEM "rule.dtd">
<mycat:rule xmlns:mycat="http://io.mycat/" >
    <tableRule name="sharding-by-intfile">
      <rule>
          <columns>sharding_id</columns>
          <algorithm>hash-int</algorithm>
      </rule>
    </tableRule>

    <function name="hash-int"
class="io.mycat.route.function.PartitionByFileMap">
        <property name="mapFile">partition-hash-int.txt</property>
     </function>
 </mycat:rule>
```

tableRule 标签配置说明：

- **name** 属性指定唯一的名字，用于标识不同的表规则
- **rule** 标签则指定对物理表中的哪一列进行拆分和使用什么路由算法。
- **columns** 内指定要拆分的列名字。
- **algorithm** 使用 function 标签中的 name 属性。连接表规则和具体路由算法。当然，多个表规则可以连接到同一个路由算法上。 table 标签内使用。让逻辑表使用这个规则进行分片。

function 标签配置说明：

- **name** 指定算法的名字。
- **class** 制定路由算法具体的类名字。
- **property** 为具体算法需要用到的一些属性。

路由算法的配置可以查看算法章节。

十个常用的分片规则

连续分片

一、日期列分区法

```
<!--按固定时间分片-->
<tableRule name="sharding-by-date">
  <rule>
    <columns>create_time</columns>
    <algorithm>sharding-by-date</algorithm>
  </rule>
</tableRule>
<function name="sharding-by-date"
  class="io.mycat.route.function..PartitionByDate">
  <property name="dateFormat">yyyy-MM-dd</property>
  <property name="sBeginDate">2014-01-01</property>
  <property name="sPartionDay">10</property>
</function>
<!--按自然月分片-->
<tableRule name="sharding-by-month">
  <rule>
    <columns>create_time</columns>
    <algorithm>sharding-by-month</algorithm>
  </rule>
</tableRule>
<function name="sharding-by-month"
  class="io.mycat.route.function..PartitionByMonth">
  <property name="dateFormat">yyyy-MM-dd</property>
  <property name="sBeginDate">2014-01-01</property>
</function>
<!--
  按单月小时分片
  适合做日志，每月末，手工清理
-->
<tableRule name="sharding-by-hour">
  <rule>
    <columns>create_time</columns>
    <algorithm>sharding-by-hour</algorithm>
  </rule>
</tableRule>
<function name="sharding-by-hour"
  class="io.mycat.route.function..LastestMonthPartition">
  <property name="splitOneDay">24</property>
```

```
</function>
```

配置说明：

- tableRule标签：

`columns`：标识将要分片的表字段

`algorithm`：指定分片函数

- function标签：

`dateFormat`：日期格式

`sBeginDate`：开始日期

`sPartionDay`：分区天数，即默认从开始日期算起，分隔10天一个分区

二、范围约定

```
<tableRule name="auto-sharding-long">
  <rule>
    <columns>user_id</columns>
    <algorithm>rang-long</algorithm>
  </rule>
</tableRule>

<function name="rang-long"
  class="io.mycat.route.function.AutoPartitionByLong">
  <property name="mapFile">autopartition-long.txt</property>
</function>
```

配置说明：

- tableRule标签：

`columns`：标识将要分片的表字段

`algorithm`：指定分片函数

- function标签：

`mapFile`：指定分片函数需要的配置文件名称

autopartition-long.txt文件内容：

所有的节点配置都是从0开始，及0代表节点1，此配置非常简单，即预先制定可能的id范围对应某个分片

```
# range start-end ,data node index
# K=1000,M=10000.
0-500M=0              0-100    0
500M-1000M=1          101-200 1
            201-300 2
1000M-1500M=2
default=0
# 或以下写法
# 0-10000000=0
# 10000001-20000000=1
```

优势：扩容无需迁移数据

缺点：热点数据，并发受限

离散分片

一、枚举法

```
<tableRule name="sharding-by-intfile">
  <rule>
    <columns>user_id</columns>
    <algorithm>hash-int</algorithm>
  </rule>
</tableRule>

<function name="hash-int"
  class="io.mycat.route.function.PartitionByFileMap">
  <property name="mapFile">partition-hash-int.txt</property>
  <property name="type">0</property>
  <property name="defaultNode">0</property>
</function>
```

配置说明：

- tableRule标签：

`columns`：标识将要分片的表字段

`algorithm`：指定分片函数

- function标签：

`mapFile`：指定分片函数需要的配置文件名称

`type`：默认值为0，0表示Integer，非零表示String

`defaultNode`：指定默认节点，小于0表示不设置默认节点，大于等于0表示设置默认节点，0代表节点1。

> - 默认节点的作用：枚举分片时，如果碰到不识别的枚举值，就让它路由到默认节点。
> - 如果不配置默认节点（defaultNode值小于0表示不配置默认节点），碰到不识别的枚举值就

> 会报错：
>
> can't find datanode for sharding column:column_name val:ffffffff

partition-hash-int.txt 配置：

```
10000=0    列等于10000  放第一个分片
10010=1
男=0
女=1
beijing=0
tianjin=1
zhanghai=2
```

二、求模法

此种配置非常明确，即根据id与count（你的结点数）进行求模运算，相比方式1，此种在批量插入时需要切换数据源，id不连续

```
<tableRule name="mod-long">
  <rule>
    <columns>user_id</columns>
    <algorithm>mod-long</algorithm>
  </rule>
</tableRule>
<function name="mod-long"
  class="io.mycat.route.function.PartitionByMod">
  <!-- how many data nodes -->
  <property name="count">3</property>
</function>
```

配置说明：

- tableRule标签：

`columns` ： 标识将要分片的表字段

`algorithm` ： 指定分片函数

- function标签：

`count` ： 节点数量

三、字符串拆分hash解析

```
<tableRule name="sharding-by-stringhash">
  <rule>
    <columns>user_id</columns>
    <algorithm>sharding-by-stringhash</algorithm>
  </rule>
</tableRule>
<function name="sharding-by-stringhash"
  class="io.mycat.route.function.PartitionByString">
  <property name="length">512</property> <!-- zero-based -->
  <property name="count">2</property>
  <property name="hashSlice">0:2</property>
</function>
```

配置说明：

- tableRule标签：

`columns`：标识将要分片的表字段

`algorithm`：指定分片函数

- function标签：

`length`：代表字符串hash求模基数

`count`：分区数

`hashSlice`：hash预算位，即根据子字符串 hash运算

> "2" -> (0,2)
>
> "1:2" -> (1,2
>
> "1:" -> (1,0)
>
> "-1:" -> (-1,0)
>
> ":-1" -> (0,-1)
>
> ":" -> (0,0)

```
public class PartitionByStringTest {

   @Test
   public void test() {
      PartitionByString rule = new PartitionByString();
      String idVal=null;
      rule.setPartitionLength("512");
      rule.setPartitionCount("2");
      rule.init();
```

```
        rule.setHashSlice("0:2");
//      idVal = "0";
//      Assert.assertEquals(true, 0 == rule.calculate(idVal));
//      idVal = "45a";
//      Assert.assertEquals(true, 1 == rule.calculate(idVal));



        //last 4
        rule = new PartitionByString();
        rule.setPartitionLength("512");
        rule.setPartitionCount("2");
        rule.init();
        //last 4 characters
        rule.setHashSlice("-4:0");
        idVal = "aaaabbb0000";
        Assert.assertEquals(true, 0 == rule.calculate(idVal));
        idVal = "aaaabbb2359";
        Assert.assertEquals(true, 0 == rule.calculate(idVal));
    }
```

四、固定分片hash算法

```
<tableRule name="rule1">
  <rule>
    <columns>user_id</columns>
    <algorithm>func1</algorithm>
  </rule>
</tableRule>

<function name="func1"
  class="io.mycat.route.function.PartitionByLong">
  <property name="partitionCount">2,1</property>
  <property name="partitionLength">256,512</property>
</function>
```

配置说明：

- tableRule标签：

`columns` ： 标识将要分片的表字段

`algorithm` ： 指定分片函数

- function标签：

`partitionCount`： 指定分片个数列表

`partitionLength`： 分片范围列表，分区长度:默认为最大2^n=1024 ,即最大支持1024分区

- 约束：

`count,length`两个数组的长度必须是一致的。 **1024 = sum((count[i]*length[i]))**

用法例子：

```
@Test
public void testPartition() {
    // 本例的分区策略：希望将数据水平分成3份，前两份各占25%，第三份占50%。（故本例非均匀分区）
    // |<---------------------1024------------------------>|
    // |<----256--->|<----256--->|<----------512---------->|
    // | partition0 | partition1 | partition2 |
    // | 共2份,故count[0]=2 | 共1份，故count[1]=1 |
    int[] count = new int[] { 2, 1 };
    int[] length = new int[] { 256, 512 };
    PartitionUtil pu = new PartitionUtil(count, length);

    // 下面代码演示分别以offerId字段或memberId字段根据上述分区策略拆分的分配结果
    int DEFAULT_STR_HEAD_LEN = 8; // cobar默认会配置为此值
    long offerId = 12345;
    String memberId = "qiushuo";

    // 若根据offerId分配，partNo1将等于0，即按照上述分区策略，offerId为12345时将会被分配到partition0中
    int partNo1 = pu.partition(offerId);

    // 若根据memberId分配，partNo2将等于2，即按照上述分区策略，memberId为qiushuo时将会被分到partition2中
    int partNo2 = pu.partition(memberId, 0, DEFAULT_STR_HEAD_LEN);

    Assert.assertEquals(0, partNo1);
    Assert.assertEquals(2, partNo2);
}
```

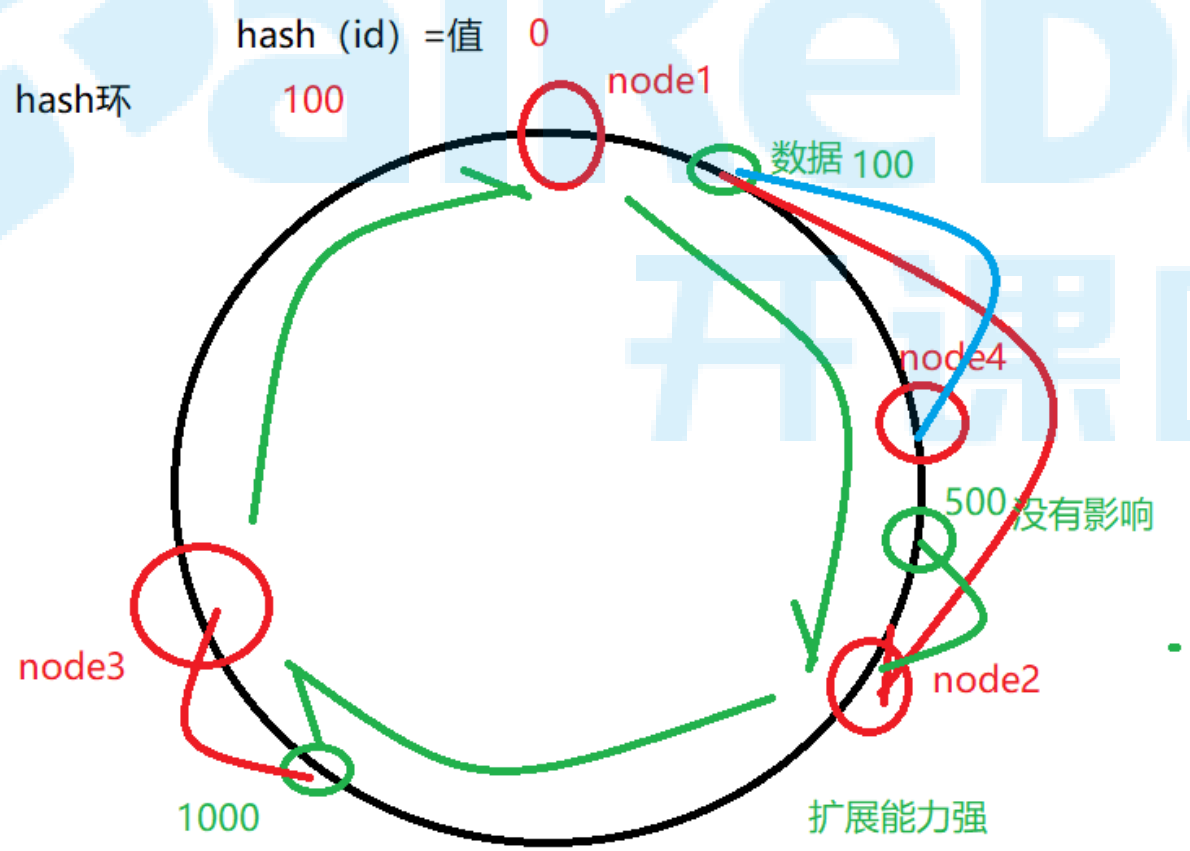
如果需要平均分配设置：平均分为4分片，partitionCount*partitionLength=1024

```
<function name="func1"
  class="org.opencloudb.route.function.PartitionByLong">
  <property name="partitionCount">4</property>
  <property name="partitionLength">256</property>
</function>
```

五、一致性hash

```
<tableRule name="sharding-by-murmur">
  <rule>
```

```
    <columns>user_id</columns>
    <algorithm>murmur</algorithm>
  </rule>
</tableRule>
<function name="murmur"
  class="io.mycat.route.function.PartitionByMurmurHash">
  <!-- 默认是0 -->
    <property name="seed">0</property>
    <!-- 要分片的数据库节点数量，必须指定，否则没法分片 -->
  <property name="count">2</property>
    <!-- 一个实际的数据库节点被映射为这么多虚拟节点，默认是160倍，也就是虚拟节点数是物理节点数的160倍 -->
  <property name="virtualBucketTimes">160</property>
  <!-- <property name="weightMapFile">weightMapFile</property> 节点的权重，没有指定权重的节点默认是1。以properties文件的格式填写，以从0开始到count-1的整数值也就是节点索引为key，以节点权重值为值。所有权重值必须是正整数，否则以1代替 -->
  <!-- <property name="bucketMapPath">/etc/mycat/bucketMapPath</property>
    用于测试时观察各物理节点与虚拟节点的分布情况，如果指定了这个属性，会把虚拟节点的murmur hash值与物理节点的映射按行输出到这个文件，没有默认值，如果不指定，就不会输出任何东西 -->
</function>
```



一致性hash预算有效解决了分布式数据的扩容问题，前1-9中id规则都多少存在数据扩容难题，而10规则解决了数据扩容难点

六、编程指定

```
<tableRule name="sharding-by-substring">
  <rule>
    <columns>user_id</columns>
    <algorithm>sharding-by-substring</algorithm>
  </rule>
</tableRule>
<function name="sharding-by-substring"
  class="io.mycat.route.function.PartitionDirectBySubString">
  <property name="startIndex">0</property> <!-- zero-based -->
  <property name="size">2</property>
  <property name="partitionCount">8</property>
  <property name="defaultPartition">0</property>
</function>
```

配置说明：

- tableRule标签：

`columns`：标识将要分片的表字段

`algorithm`：指定分片函数

- function标签：

`startIndex`：字符串截取的起始索引位置

`size`：截取的位数

`partitionCount`：分区数量

`defaultPartition`：默认分区

11010419800101

此方法为直接根据字符子串（必须是数字）计算分区号（由应用传递参数，显式指定分区号）。

例如id=05-100000002

在此配置中代表根据id中从startIndex=0，开始，截取siz=2位数字即05，05就是获取的分区，如果没传默认分配到defaultPartition

优点：数据分布均匀，并发能力强

缺点：移植性差、扩容性差

综合分片

一、通配取模

```
<tableRule name="sharding-by-pattern">
  <rule>
    <columns>user_id</columns>
    <algorithm>sharding-by-pattern</algorithm>
  </rule>
</tableRule>
<function name="sharding-by-pattern"
  class="io.mycat.route.function.PartitionByPattern">
  <property name="patternValue">256</property>
  <property name="defaultNode">2</property>
  <property name="mapFile">partition-pattern.txt</property>
</function>
```

配置说明：

- tableRule标签：

`columns` ： 标识将要分片的表字段

`algorithm` ： 指定分片函数

- function标签：

`patternValue`：求模基数

`defaultNode`：默认节点，如果不配置了默认，则默认是0即第一个结点

`mapFile`：配置文件路径

partition-pattern.txt文件内容：

配置文件中，`1-32` 即代表`id%256`后分布的范围，如果在1-32则在分区1，其他类推，如果id非数字数据，则会分配在defaultNode 默认节点

```
# id partition range start-end ,data node index
###### first host configuration
1-32=0
33-64=1
65-96=2
97-128=3
######## second host configuration
129-160=4
161-192=5
193-224=6
225-256=7
0-0=7
```

二、ASCII码求模通配

```
<tableRule name="sharding-by-prefixpattern">
  <rule>
    <columns>user_id</columns>
    <algorithm>sharding-by-prefixpattern</algorithm>
  </rule>
</tableRule>
<function name="sharding-by-pattern"
  class="io.mycat.route.function.PartitionByPrefixPattern">
  <property name="patternValue">256</property>
  <property name="prefixLength">5</property>
  <property name="mapFile">partition-pattern.txt</property>
</function>
```

配置说明：

- tableRule标签：

`columns`：标识将要分片的表字段

`algorithm`：指定分片函数

- function标签：

`patternValue`：求模基数

`prefixLength`：ASCII 截取的位数

`mapFile`：配置文件路径

partition-pattern.txt文件内容：

配置文件中，`1-32`即代表`id%256`后分布的范围，如果在1-32则在分区1，其他类推

此种方式类似方式6，只不过采取的是将列中前prefixLength位所有ASCII码的和与patternValue 进行求模，即`sum%patternValue`,获取的值在通配范围内的，即分片数。

> ASCII编码：
>
> - 48-57=0-9阿拉伯数字
> - 64、65-90=@、A-Z
> - 97-122=a-z

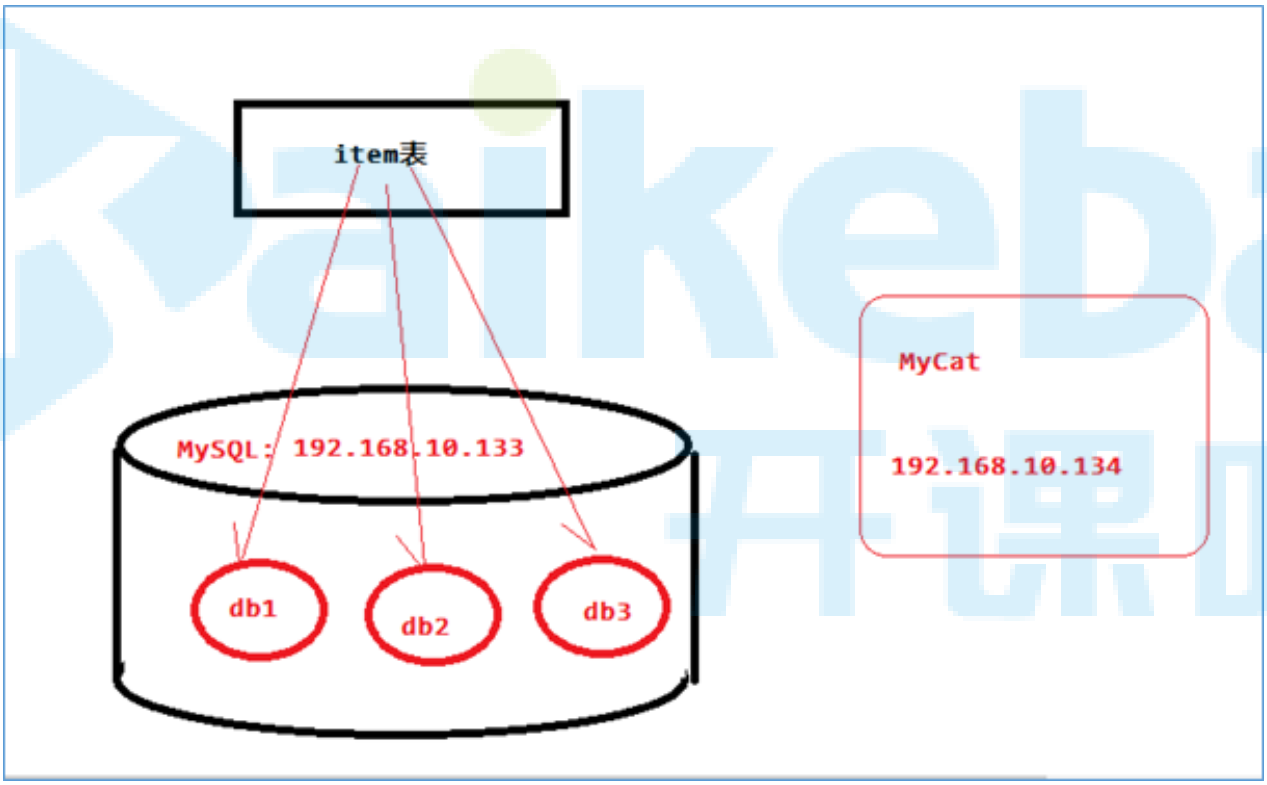
```
# range start-end ,data node index
# ASCII
# 48-57=0-9
# 64、65-90=@、A-Z
# 97-122=a-z
###### first host configuration
1-4=0
5-8=1
9-12=2
13-16=3
###### second host configuration
```

```
17-20=4
21-24=5
25-28=6
29-32=7
0-0=7
```

测试分片

需求

把商品表分片存储到三个数据节点上。



创建表

配置完毕后，重新启动mycat。使用mysql客户端连接mycat，创建表。

```
CREATE TABLE item (
  id int(11) NOT NULL,
  name varchar(20) DEFAULT NULL,
  PRIMARY KEY (id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

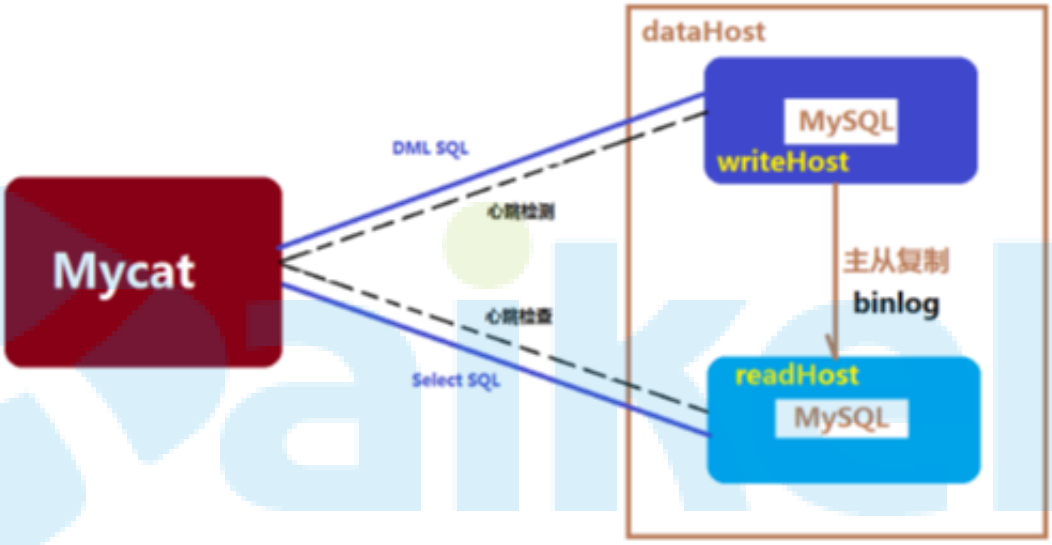
分片测试

- 分片策略指定为“auto-sharding-long”
- 分片规则指定为“mod-long”

Mycat读写分离

MyCat的读写分离是建立在**MySQL主从复制基础**之上实现的，所以必须先搭建MySQL的主从复制。

数据库读写分离对于大型系统或者访问量很高的互联网应用来说，是必不可少的一个重要功能。对于MySQL来说，标准的读写分离是主从模式，一个写节点Master后面跟着多个读节点，读节点的数量取决于系统的压力，通常是1-3个读节点的配置



Mycat实现的读写分离和自动切换机制，需要mysql的主从复制机制配合。

Mycat配置

Mycat 1.4 支持MySQL主从复制状态绑定的读写分离机制，让读更加安全可靠，配置如下：

```
  <dataNode name="dn1" dataHost="localhost1" database="db1" />
  <dataNode name="dn2" dataHost="localhost1" database="db2" />
  <dataNode name="dn3" dataHost="localhost1" database="db3" />

  <dataHost name="localhost1" maxCon="1000" minCon="10" balance="1"
    writeType="0" dbType="mysql" dbDriver="native" switchType="2"
slaveThreshold="100">
    <heartbeat>show slave status</heartbeat>
    <writeHost host="hostM" url="192.168.25.134:3306" user="root"
      password="root">
      <readHost host="hostS" url="192.168.25.166:3306" user="root"
      password="root" />
    </writeHost>
  </dataHost>
```

(1) 设置 balance="1"与writeType="0"

Balance参数设置：

1. balance=“0”, 所有读操作都发送到当前可用的writeHost上。
2. balance=“1”，所有读操作都随机的发送到readHost。
3. balance=“2”，所有读操作都随机的在writeHost、readhost上分发

WriteType参数设置：

1. writeType=“0”, 所有写操作都发送到可用的writeHost上。
2. writeType=“1”，所有写操作都随机的发送到readHost。
3. writeType=“2”，所有写操作都随机的在writeHost、readhost分上发。

“readHost是从属于writeHost的，即意味着它从那个writeHost获取同步数据，因此，当它所属的writeHost宕机了，则它也不会再参与到读写分离中来，即“不工作了”，这是因为此时，它的数据已经“不可靠”了。基于这个考虑，目前mycat 1.3和1.4版本中，若想支持MySQL一主一从的标准配置，并且在主节点宕机的情况下，从节点还能读取数据，则需要在Mycat里配置为两个writeHost并设置banlance=1。”

(2) 设置 switchType="2" 与slaveThreshold="100"

switchType 目前有三种选择：

-1：表示不自动切换

1：默认值，自动切换

2：基于MySQL主从同步的状态决定是否切换

Mycat心跳检查语句配置为 show slave status ，dataHost 上定义两个新属性： **switchType="2" 与slaveThreshold="100"**，此时意味着开启MySQL主从复制状态绑定的读写分离与切换机制。Mycat心跳机制通过检测 show slave status 中的 **"Seconds_Behind_Master", "Slave_IO_Running", "Slave_SQL_Running"** 三个字段来确定当前主从同步的状态以及Seconds_Behind_Master主从复制时延。