常规优化策略

MySQL查询优化

1.关于MySQL的查询优化目的和目标：

优化的目的是让资源发挥价值；

SQL和索引是调优的关键，往往可以起到“四两拨千斤”的效果。

2.关于优化的流程和思路：

充分了解核心指标，并构建完备的监控体系，这是优化工作的前提；

SQL优化的原则是减少数据访问及计算；

常用的优化方法主要是调整索引、改写SQL、干预执行计划。

3.关于MySQL的核心概念及原理：

Innodb的表是典型的IOT，数据本身是B+ tree索引的叶节点；

扫描二级索引可以直接获取数据，或者返回主键ID；

优化器是数据库的大脑，我们要了解优化器，并观测以及干预MySQL的行为。

MySQL 开发规约实战

作者：芦火

前言

语句规范要建立在结构规范的基础上。

（一）字符集

1.统一字符集，建议UTF8mb4

常用的字符集包括：Latin1、gbk、utf8、utf8mb4。

常用字符集	描述	默认校对规则	最大长度	备注
latin1	cp1252 West European	latin1_swedish_ci	1	早期官方默认字符集
gbk	GBK Simplified Chinese	gbk_chinese_ci	2	非国际标准
utf8	UTF-8 Unicode	utf8_general_ci	3	alias for utf8mb3
utf8mb4	UTF-8 Unicode	utf8mb4_0900_ai_ci	4	官方8.0默认字符集

2.统一排序规则

目前互联网上以UTF8mb4字符集为主，是官方8.0默认字符集。在之前的5.5、5.6、5.7版本是建表使用的Utf8，排序规则是默认“utf8_general_ci”。在8.0之前UTF8mb4独有的默认排序方式是“utf8_general_ci”,在8.0之后默认规则为“utf8mb4_0900_ai_ci”，所以有时会出现不同版本间字符集排序规则不兼容的问题，需要注意。

（二）字段

1. 统一字段名/类型

统一字段名是为了解决**业务歧义问题**。MySQL内部系统Information_schema下边的Tables表，假设按照不规范的命名如“table_name”字段可能就会命名成“name”。如果这时Columns表中的COLUMN_NAME字段也命名为“Name”字段，在查询的时候，可能会导致意义上的混乱。统一类型是为了**解决隐式转换问题**，包括表的连接、查询都会存在隐式转换问题。

2.字段长度 varchar(255)

常见的问题是字段长度都配置为varchar(255)，在不知道业务将来存多少长度的情况下，先设成255，**在开发阶段可能比较方便，但存在性能隐患**。比如索引评估，在一个255列长的字段上建索引，实际索引评估会考虑列长，如果默认255长

度，会导致索引使用时评估不准确。

再比如字段，如果字段有2个255或3个255要做复合索引时，虽然真实的值可能每个字段长度只存了10或20，在默认参数配置下会发现索引由于太长建不出来，对线上维护与后续业务开发都有影响。

3.定义 id int primary key

PK　在业务中建议强制必须建立。可以保证主从架构下的数据一致性以及避免复制时性能问题。另一个是主键要如果采用数值型建议使用无符号类型，一般来说在一个表里ID肯定是自增的，不存在负值，如果定义一个有符号Int，会导致Int可用的值少一半。因为Int最大的值在有符号情况下是21亿，如果定义成无符号最大可以到42亿。说明数据快速增长时，有符号类型导致ID或某个自增长满的问题。

4. 禁止Null值

Null & Null =?。比如在排序场景下，两个行按某个允许null的列值做排序，如果不存储有意义的值，默认为null值的情况下，会导致一个随机的顺序，实际上就是业务上的乱序。 又比如无主键表情况下，会导致复制数据不一致的问题，所以要禁止空值。

（三）索引

0%的语句性能问题都可以靠索引解决，但索引有几个问题：

第一，**单列索引要充分评估**，比如有20个列，每个列上都有1个单列的情况，会造成对写入的影响很大，同时单列索引的建议一定要评估可选择性。

第二，**定期Review索引有效性**，索引是不是在业务中真正使用在MySQL里相对不好定位，失效索引在业务快速发展频繁变更的场景下会很常见，随着新业务新添加很多新索引，这时要看新的索引是不是已经覆盖之前的旧索引，此时旧索引实际上是没有用的。维护无效索引要多一份IO成本，删除除重复索引保留有效的即可。

第三，**不要走极端**，包括两点：复合索引所有列与所有列都建单列索引。

比如一张表有七八个列，只在单列有索引，因为索引有回表不回表的区别，所以直接建立一个包含所有列复合索引，这个方法不可取，虽然提升了查询的效率，但等于又另外维护了一张所有字段都要排序的表。

所有列都建单列索引，实际上跟是复合索引所有列是一样的。主要消耗会出现索引维护上。

索引有关内容，请关注【MySQL表和索引优化实战】课程。

SQL语句编写规范

（一）规范语法

不兼容语法：

Select * from sbtest.sbtest1 group by id;

Select id,count(*) from sbtest.sbtest1 group by id desc;

MySQL是一个相对成熟的产品，但它支持的一些语法并不标准，比如“Select * from sbtest.sbtest1 group by id;”在传统的数据库如Oracle和其他关系型数据库里中是非法的语法，系统不支持，而“Select id,count(*) from sbtest.sbtest1 group by id desc;”在 8.0版本已经淘汰。随着MySQL语法越来越规范化，在版本升级后，这种不兼容语法可能会带来应用或语句报错，因此在实际环境下不建议使用。

（二）别名

Select id,count(*) id_count from sbtest.sbtest1 group by id;

所有返回列要给有意义的命名，与列名原则一致，强制AS关键词，防止造成语意不清。

（三）执行顺序

执行顺序如下：

1 .FROM,（-including JOIN ）

2. WHERE

3. GROUP BY

4. HAVING

5. WINDOW functions

6. SELECT

7. DISTINCT

8. UNION

9. ORDER BY

10. LIMIT and OFFSET

· 语句性能应注意两个方面：

1）数据流的流向；

2）order by limit场景。

从执行顺序上看，在SELECT之前的所有子语都是在做数据筛选，SELECT以后开始执行运算，用户应注意数据流的流向。order by limit一般是在最后运行，如果在一开始运行，有时候会造成返回数据量过大，进而导致执行时间过长。

· 数据返回逻辑应注意两个方面：

1）数据的筛选机制；

2）left join where场景。

许多用户的概念中都是先做WHERE再做JOIN，这是错误的。例如在做Left Join时，先获得所有数据，再通过Where筛选数据。用户应注意梳理流向，才能最优地输出数据。

（四）如何判断语句是否已最优:explain

```
+----+-------------+-------+------------+-------+---------------+---------+---------+-----------+------+----------+-----------------------------------------------------+
| id | select_type | table | partitions | type  | possible_keys | key     | key_len | ref       | rows | filtered | Extra                                               |
+----+-------------+-------+------------+-------+---------------+---------+---------+-----------+------+----------+-----------------------------------------------------+
|  1 | SIMPLE      | c     | NULL       | range | idx_pk        | idx_pk  | 100     | NULL      | 1460 |     1.11 | Using index condition; Using where; Using MRR |
|  1 | SIMPLE      | b     | NULL       | ref   | i_test        | i_test  | 302     | b.c.KeyNo |    1 |   100.00 | Using where; Using index                            |
+----+-------------+-------+------------+-------+---------------+---------+---------+-----------+------+----------+-----------------------------------------------------+
```

如上图所示，用户可以通过Explain判断语句是否已最优，其中Type与Extra的主要类型与含义如下：

· Type

1）ALL：Full Table Scan 全表扫描；

2）index: Full Index Scan，索引扫描；

3）range:范围扫描；

4）ref: 表示连接匹配条件；

5）eq_ref: 类似ref，区别就在使用的索引是唯一索引；

6）const: 常量查询，比如pk等值；

7）system是Const类型的特例；当查询的表只有一行的情况下，使用system。

从性能角度来看，从上往下性能越来越高，根据《开发手册》，此处最低要求是到Range范围扫描。

· Extra

1）Using filesort 排序；

2）Using index 使用索引可以返回请求列；

3）Using index condition 通过索引初步过滤；回表再过滤其它条件；

4）Using temporary 临时表；

5）Using where 单独出现时；一般代表表上出现全表扫描过滤；

6）Using index & Using where 使用索引返回数据；同时通过索引过滤。

Extra反映了执行计划的真实执行情况。

结合上图执行计划分析，C表是外部驱动表，索引方式为idx_pk，Type是Range，Extra有Using index condition、Using where以及Using MRR，表示进行全表扫描，通过索引初步过滤，回表B再过滤其他条件。B表是从外表取数据做内循环，索引方式为i_text，扫描的列为c.b.KeyNo，这种情况说明这个执行计划相对完善。

（五）禁止与建议

SQL的语句编写包含一些禁止项与建议项语句，用户深入了解与熟练掌握这些内容能够更好地开展业务。

1.禁止项

1）select *，返回无用数据，过多IO消耗，以及Schema 变更问题；

2）Insert语句指定具体字段名称，不要写成insert into t1 values(…)，道理同上；

3）禁止不带WHERE，导致全表扫描以及误操作；

4）Where条件里等号左右字段类型必须一致，否则可能 会产生隐式转换，无法利用索引；

5）索引列不要使用函数或表达式，否则无法利用索引。

如where length(name)=‘Admin’或where user_id+2=5；

6）Replace into，会导致主备不一致；

7）业务语句中带有DDL操作，特别是Truncate。

2.建议项

1）减小三表以上Join；

2）用Union all 替代Union；

3）使用Join 替代子查询；

4）不要使用 like ‘%abc%’，可以使用 like ‘abc%’；

5）Order by /distinct /group by 都可以利用索引有序性；

6）减少使用event/存储过程，通过业务逻辑实现；

7）减小where in() 条件数据量；

8）减少过于复杂的查询和拼串写法。

（六）用数据库的思维考虑SQL

我们提倡用户用数据库的思维考虑SQL，由于数据库要处理的是数据集而非单行数据，因此与开发的逻辑不太一样。

在开发逻辑中，有时候会希望通过用一个语句解决所有问题，但这在数据库中会导致SQL语句过大甚至上万行，过于复杂的查询使得执行计划不稳定。因此我们倡导少即是美，每一层结果集都要最大限度地减小。

数据库中无法用开发应用的逻辑写语句，而应把所有的运算、判断应用逻辑都放到SQL实现。存储过程使用过重的话，会导致难以调适、定位问题。同时，应减少单条数据集处理，减少数据访问与扫描。

对于新Feature，在未经过充分测试的情况下，应谨慎考虑使用到生产中，防止造成Bug或存在性能上的问题。

（七）SQL改写

SQL有一些编写规范，这些规范是在优化日常问题时总结而来，下面举例说明。

1.SQL改写-join

select count(a.id) from sbtest1 a left join sbtest2 b on a.id=b.id

```
mysql>  select count(a.id) from sbtest1 a left join sbtest2  b on a.id=b.id;
+--------------+
| count(a.id) |
+--------------+
|    20000000 |
+--------------+
1 row in set (19.34 sec)
```

如上图所示，请注意Join键为PK，也就是左表右表应该是1对1的关系，在Left Join的情况下，可以理解成返回的数据全部是左边的数据，也就是“a”表的数据，执行时间大概为20秒。

```
mysql> explain select count(*) from sbtest1 a left join sbtest2  b on a.id=b.id;
+----+-------------+-------+------------+--------+---------------+---------+---------+-------------+----------+----------+-------------+
| id | select_type | table | partitions | type   | possible_keys | key     | key_len | ref         | rows     | filtered | Extra       |
+----+-------------+-------+------------+--------+---------------+---------+---------+-------------+----------+----------+-------------+
|  1 | SIMPLE      | a     | NULL       | index  | NULL          | k_1     | 4       | NULL        | 19728432 |   100.00 | Using index |
|  1 | SIMPLE      | b     | NULL       | eq_ref | PRIMARY       | PRIMARY | 4       | sbtest.a.id |        1 |   100.00 | Using index |
+----+-------------+-------+------------+--------+---------------+---------+---------+-------------+----------+----------+-------------+
2 rows in set, 1 warning (0.00 sec)

mysql> explain  select count(a.id) from sbtest1 a ;
+----+-------------+-------+------------+-------+---------------+------+---------+------+----------+----------+-------------+
| id | select_type | table | partitions | type  | possible_keys | key  | key_len | ref  | rows     | filtered | Extra       |
+----+-------------+-------+------------+-------+---------------+------+---------+------+----------+----------+-------------+
|  1 | SIMPLE      | a     | NULL       | index | NULL          | k_1  | 4       | NULL | 19728432 |   100.00 | Using index |
+----+-------------+-------+------------+-------+---------------+------+---------+------+----------+----------+-------------+
1 row in set, 1 warning (0.00 sec)

mysql> select count(a.id) from sbtest1 a ;
+-------------+
| count(a.id) |
+-------------+
|    20000000 |
+-------------+
1 row in set (3.32 sec)
```


如上图说是，可以将“select count(a.id) from sbtest1 a left join sbtest2 b on a.id=b.id;”简化为“select count(a.id) from sbtest1 a”，执行时间缩减到3秒左右，大幅提升执行效率。

SQL改写一般会出现在复杂查询的Join场景中，除去显式Join（left join与right join），还包括半连接（exists,in）和反连接（not exists,not in）。

此类查询过慢时，请参考执行计划，考虑是否可通过SQL改写优化。

2.SQL改写-分页统计

分页统计是一种常见的业务逻辑，例如我们现在有一条分页语句：

select a.id from sbtest1 a left join sbtest2 b on a.id=b.id limit 200,20;

取总数据量：

select count(*) from

(select a.id from sbtest1 a left join sbtest2 b on a.id=b.id) as a;

```
+----+-------------+-------+------------+-------+---------------+---------+---------+-------------+----------+----------+-------------+
| id | select_type | table | partitions | type  | possible_keys | key     | key_len | ref         | rows     | filtered | Extra       |
+----+-------------+-------+------------+-------+---------------+---------+---------+-------------+----------+----------+-------------+
|  1 | SIMPLE      | a     | NULL       | index | NULL          | k_1     | 4       | NULL        | 19728432 |   100.00 | Using index |
|  1 | SIMPLE      | b     | NULL       | eq_ref | PRIMARY      | PRIMARY | 4       | sbtest.a.id |        1 |   100.00 | Using index |
+----+-------------+-------+------------+-------+---------------+---------+---------+-------------+----------+----------+-------------+
```

分页统计是一种常见的业务逻辑，比如有1万条数据需要分页，常见的处理方法是把以上所有的语句逻辑框起来，在外面加“Count”，这种做法会导致语句冗余，且执行时间长。改写的方法有：

改写1：

select count(a.id) from sbtest1 a left join sbtest2 b on a.id=b.id;

改写2:

select count(a.id) from sbtest1 a;

```
mysql> select count(a.id) from sbtest1 a left join sbtest2  b on a.id=b.id;
+--------------+
| count(a.id) |
+--------------+
|    20000000 |
+--------------+
1 row in set (15.06 sec)

mysql> select count(a.id)  from sbtest1 a;
+--------------+
| count(a.id) |
+--------------+
|    20000000 |
+--------------+
1 row in set (0.25 sec)
```

如上图所示，两种改写方式的执行计划与最初写法的语义逻辑上无本质区别，第一种改写后执行时间为15秒，第二种改写后执行时间为0.25秒，且语句更加简单。

· 此类改写目的：

1）精简语句，简化语句逻辑；

2）进一步寻找优化空间。

事务的使用与优化

（一）事务是什么?

事务是指访问并可能更新数据库中各种数据项的一个程序执行单元(Unit)。

事务应该具有四个属性：原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Durability），这四个属性通常称为ACID特性。

目前生产环境所用的隔离级别较多，主要有以下四种：

1）Read Uncommitted

2）Read Committed（一般采用）

3）Repeatable Read （官方默认）

4）Serializable

关于事务需要强调一点：**大事务不等于长事务**。

例如：

1）Insert table batch

它是个大事务，但它可能并不长。

2）Begin

insert single data

sleep(3600)

Commit

这是个长事务，但不是大事务。

同时要说一下，有些DDL本身是原子性的，包括加列、建索引，事务可能大且长。

（二）事务的问题

长事务和大事务可能存在以下问题：

1.Undo 异常增长

导致Ibdata空间问题，增加存储成本，也会使得Hitory List过长，导致严重的性能问题。

2.Binlog 异常增长

由于单个事务不拆分存放，会导致某一个或者某一些Binlog非常的大，做复制或主从时会产生一定问题。

3.Slave延迟

DDL类，写入等，DDL是语句级回放，Slave要等到执行结束后再继续。

4.锁问题

死锁、阻塞。

针对大事务与长事务做出优化：

· 大事务

1）大事务拆分为小事务；

2）DDL拆分（无锁变更）。

· 长事务

1）合并为大事务（特别合适应用于写入场景，对写提升很大，而且数据不会特别长）；

2）事务分解（不必要的请求摘除）；

3）应用侧保证一致性。

事务使用基本原则：在保证业务逻辑的前提下，尽可能缩短事务长度。

（三）事务问题定位

1.长事务问题定位

定位命令：Information_schema.innodb_trx

例如：

SELECT trx.trx_id, trx.trx_started,

trx.trx_mysql_thread_id FROM

INFORMATION_SCHEMA.INNODB_TRX trx WHERE

trx.trx_started < CURRENT_TIMESTAMP - INTERVAL 1

SECOND。

2.锁问题定位

8.0以前：information_schema.innodb_lock_waits、innodb_locks。

8.0及以后：performance_schema. data_lock_waits、data_locks。

```
+---------------------------------+---------------------+------+-----+---------+-------+
| Field                           | Type                | Null | Key | Default | Extra |
+---------------------------------+---------------------+------+-----+---------+-------+
| ENGINE                          | varchar(32)         | NO   |     | NULL    |       |
| REQUESTING_ENGINE_LOCK_ID       | varchar(128)        | NO   | MUL | NULL    |       |
| REQUESTING_ENGINE_TRANSACTION_ID | bigint(20) unsigned | YES  | MUL | NULL    |       |
| REQUESTING_THREAD_ID            | bigint(20) unsigned | YES  | MUL | NULL    |       |
| REQUESTING_EVENT_ID             | bigint(20) unsigned | YES  |     | NULL    |       |
| REQUESTING_OBJECT_INSTANCE_BEGIN | bigint(20) unsigned | NO   |     | NULL    |       |
| BLOCKING_ENGINE_LOCK_ID         | varchar(128)        | NO   | MUL | NULL    |       |
| BLOCKING_ENGINE_TRANSACTION_ID  | bigint(20) unsigned | YES  | MUL | NULL    |       |
| BLOCKING_THREAD_ID              | bigint(20) unsigned | YES  | MUL | NULL    |       |
| BLOCKING_EVENT_ID               | bigint(20) unsigned | YES  |     | NULL    |       |
| BLOCKING_OBJECT_INSTANCE_BEGIN  | bigint(20) unsigned | NO   |     | NULL    |       |
+---------------------------------+---------------------+------+-----+---------+-------+
```

开发中常见问题与最佳实践

（一）分页问题

分页的传统写法：select * from sbtest1 order by id limit M,N。

它存在的问题点：需要扫描大量无效数据后，再返回请求数据，成本很高。

根据业务需求，有以下三种解决方法：

1）select * from sbtest1 where id > #max_id# order by id limit n;

适用顺序翻页的场景，每次记录上一页#max_id#带入下一次查询中。

2）select * from sbtest1 as a inner join (select id from sbtest1 order by id limit m, n) as b

on a.id = b.id order by a.id;

适用只按照id进行分页，无Where条件。

3）select * from sbtest1 as a

inner join (select id from sbtest1where col=xxxx order by id limit m, n) as b

on a.id = b.id order by a.id;

适用于带Where条件，同时按照ID顺序分页。此时，需要在Where条件上创建二级索引。

（二）大表数据清理

1.数据清理场景

一般的数据清理场景为历史数据清理，例如数据归档、Delete等。

这里经常存在的问题有：

1）单次Delete行数过多，容易导致锁堵塞、主从复制延迟、影响线上业务；

2）易失败，死锁、超时等。

解决的建议方案：

1）伪代码

Select min(id),max(id) from t where gmt_create<$date

For I in “max(id)-min(id)/1000

Delete from t where id>=min(id) and id<min(id)+1000 and gmt_create<$date

……

2）定期Optimize Table回收碎片。

2.全表数据清理

常用的方法：用Truncate删掉整张表的数据。

存在问题：大表（如：>100G），Truncate期间会造成IO持续抖动。

解决方案：硬连接方式后Truncate，异步Trim文件。

（三）隐式转换问题

隐式转换问题发生在比较值类型不一致的场景下，除去一些规定情况，所有的比较最终都是转换为浮点数进行。

Create table testtb(id varchar(10) primary key);

Select * from testtb where id=1;

此类问题在编写sql语句时很难发现，上线可能会导致严重的性能问题。

（四）循环

开发环境中的循环分为外部循环与内部循环。

1.外部循环

外部循环在应用侧实现，主要问题来自每次请求的RT。

例如：

```
for i=0;i++;i<500
  insert（db 交互）
  next
rt=single rt* total count
```

建议Batch一次写入。

2.内部循环

内部循环常用在存储过程，事务无法保证。

例如：

```
While do
  insert;
  Commit;
end while
```

存在频繁Commit问题，造成IO上的冲击。

或:

```
Begin tran
While do
  insert;
end while
Commit
```

无法保证数据一致性，以及事务过长。

（五）存储过程中的事务处理

```
create procedure insertTest(IN num int)
BEGIN
   DECLARE errno int;
   declare i int;
   declare continue HANDLER for sqlexception set errno=1;
start transaction;
set i=0;
while i<num do
   INSERT testfor VALUES(i);
   set i=i+1;
end while;
if errno=1 then
   rollback;
else
   commit;
end if;
end;
```

以上方为例，在BEGIN Train后，下面最终有一个Commit。如果这里是一个重复键值，但前面已经插了10条数据，这10条数据是不回滚的，所以这个事务要直接在这里声明捕捉错误，然后回滚整个事务，才能完成这整个存储过程的回滚。

（六）常见问题

1. Where 后面的列顺序是不是要符合最左原则?

Where a=1 and b=2 等价于 Where b=2 and a=1

最左原则指的是索引顺序，不是谓词顺序，以上两个条件都匹配(a,b) 复合索引。

2. Join 的顺序是否指定左边为驱动表?

Inner Join场景下，在执行计划中按统计信息的预估自动选中驱动表，Left Join ,Right Join 时左右写的顺序才有显式意义。

3.业务上有随机返回的需求，能否用order by rand()

一般不建议，如果结果集非常小，勉强可用，但结果集大时由于随机数排序，会产生Sort操作甚至溢出到磁盘，有很大性能损耗，此类需求可以考虑伪随机算法。

4.Delete数据之后，为什么磁盘空间占用反而大了?

Delete数据并不能清理数据文件空间，反而会导致Undo,Binlog文件的增长，使用Optimize收缩。

5.Binlog是否一定要Row格式?

在主从场景下，Binlog使用Row格式是为了保证主从数据一致性。

单机场景下，Binlog做为增长数据备份使用，同时也包括一些语句级数据恢复的功能。

6.死锁、阻塞的区别

通常说的阻塞，主要是由于锁获取不到，产生的请求被阻塞，一般需要手动解锁(Kill或等待)。

死锁不等于阻塞，虽然死锁中阻塞是必现的，但是会自动回滚事务解锁，不用手动处理，但需要业务判断语句逻辑。

以上两种情况都是由于业务侧逻辑出现，并非内核原因。

7.做DDL时是否会锁表

所有的DDL都需要锁表，只是操作顺序和操作获取时间的问题。如下图所示，允许并发DDL是No，就证明对业务有一些阻塞。

Operation	In Place	Rebuilds Table	Permits Concurrent DML	Only Modifies Metadata
Creating or adding a secondary index	Yes	No	Yes	No
Dropping an index	Yes	No	Yes	Yes
Adding a `FULLTEXT` index	Yes*	No*	No	No
Changing the index type	Yes	No	Yes	Yes

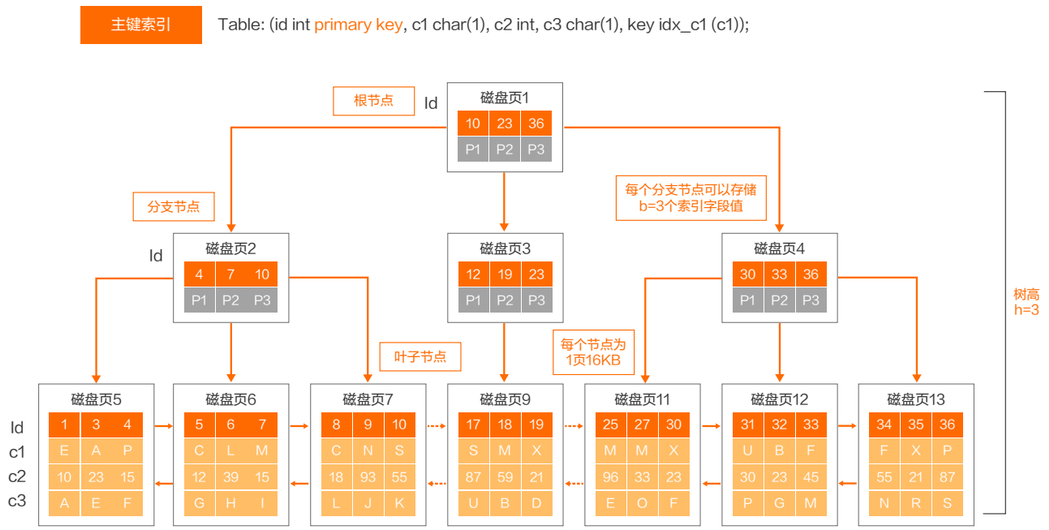
详情参考官方文档：https://dev.mysql.com/doc/refman/5.6/en/innodb-online-ddl-operations.html

RDS for MySQL表和索引优化实战

作者：田杰

主键索引

作为数据库的使用者，每天都需要跟数据库打交道，避免不了接触两个概念，一个是表，一个是索引。日常思维中，表是用来存储表中的数据，索引是用来加速查询访问。下面来看一下RDS for MySQL在InnoDB引擎下面，数据的物理组织是如何组织?



如上图所示这张表，它的主键是1个名为 id 的整型字段、4个字节、id作为主键，后面跟着一个单字符的c1，然后还有一个int类型的c2，然后单字符的c3，同时在 c1字段上有索引，这是很简单的一张表。这个数据是如何组织?

在InnoDB引擎下，数据是存储在主键中，就是指数据是通过主建进行物理组织，跟Oracle本身默认的堆表不一样，Oracle本身默认创建的表如果不指定的话是一个堆表，真的有一个对象、物理结构、数据结构，以堆的数据结构来存储数据，同时主建是另外一个数据结构，是两份数据。对于 MySQL在InnoDB引擎下面，本身数据是存储在主键的叶子节点中，如下图所示，“c1、c2、c3” 3列数据都存储在主键的叶子节点。

整个主键的数据结构是B+-Tree，B指的是Balance Tree多路平衡树，而不是Binary tree二叉树。多路平衡树和二叉树之间区别在于: