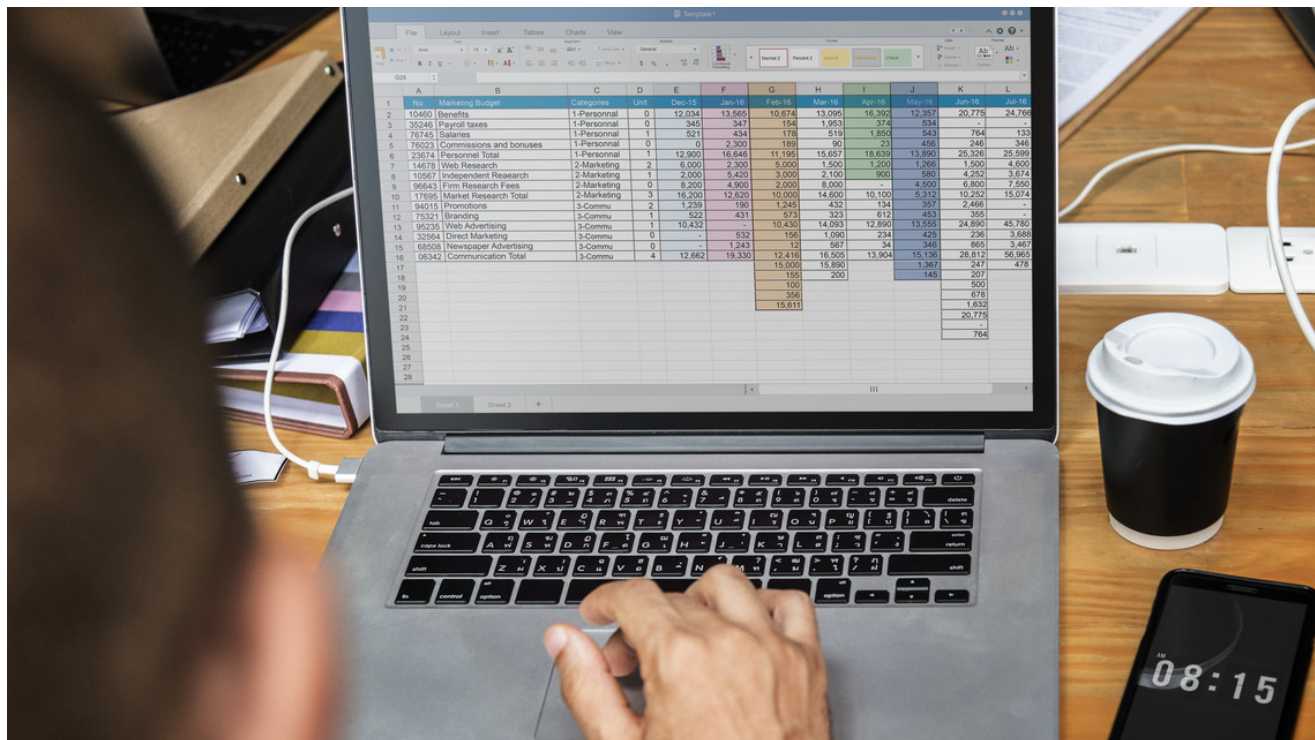


## 38 | 都说InnoDB好，那还要不要使用Memory引擎？

[time.geekbang.org/column/article/80495](http://time.geekbang.org/column/article/80495)



我在上一篇文章末尾留给你的问题是：两个 group by 语句都用了 order by null，为什么使用内存临时表得到的语句结果里，0 这个值在最后一行；而使用磁盘临时表得到的结果里，0 这个值在第一行？

今天我们就来看看，出现这个问题的原因吧。

### 内存表的数据组织结构

为了便于分析，我来把这个问题简化一下，假设有以下的两张表 t1 和 t2，其中表 t1 使用 Memory 引擎，表 t2 使用 InnoDB 引擎。

```
create table t1(id int primary key, c int) engine=Memory;
```

```
create table t2(id int primary key, c int) engine=innodb;
```

```
insert into t1 values(1,1),(2,2),(3,3),(4,4),(5,5),(6,6),(7,7),(8,8),(9,9),(0,0);
```

```
insert into t2 values(1,1),(2,2),(3,3),(4,4),(5,5),(6,6),(7,7),(8,8),(9,9),(0,0);
```

然后，我分别执行 select \* from t1 和 select \* from t2。

```
mysql> select * from t1;
+----+-----+
| id | c     |
+----+-----+
| 1  | 1     |
| 2  | 2     |
| 3  | 3     |
| 4  | 4     |
| 5  | 5     |
| 6  | 6     |
| 7  | 7     |
| 8  | 8     |
| 9  | 9     |
| 0  | 0     |
+----+-----+
10 rows in set (0.00 sec)

mysql> select * from t2;
+----+-----+
| id | c     |
+----+-----+
| 0  | 0     |
| 1  | 1     |
| 2  | 2     |
| 3  | 3     |
| 4  | 4     |
| 5  | 5     |
| 6  | 6     |
| 7  | 7     |
| 8  | 8     |
| 9  | 9     |
+----+-----+
10 rows in set (0.00 sec)
```

图 1 两个查询结果 -0 的位置

可以看到，内存表 t1 的返回结果里面 0 在最后一行，而 InnoDB 表 t2 的返回结果里 0 在第一行。

出现这个区别的原因，要从这两个引擎的主键索引的组织方式说起。

表 t2 用的是 InnoDB 引擎，它的主键索引 id 的组织方式，你已经很熟悉了：InnoDB 表的数据就放在主键索引树上，主键索引是 B+ 树。所以表 t2 的数据组织方式如下图所示：

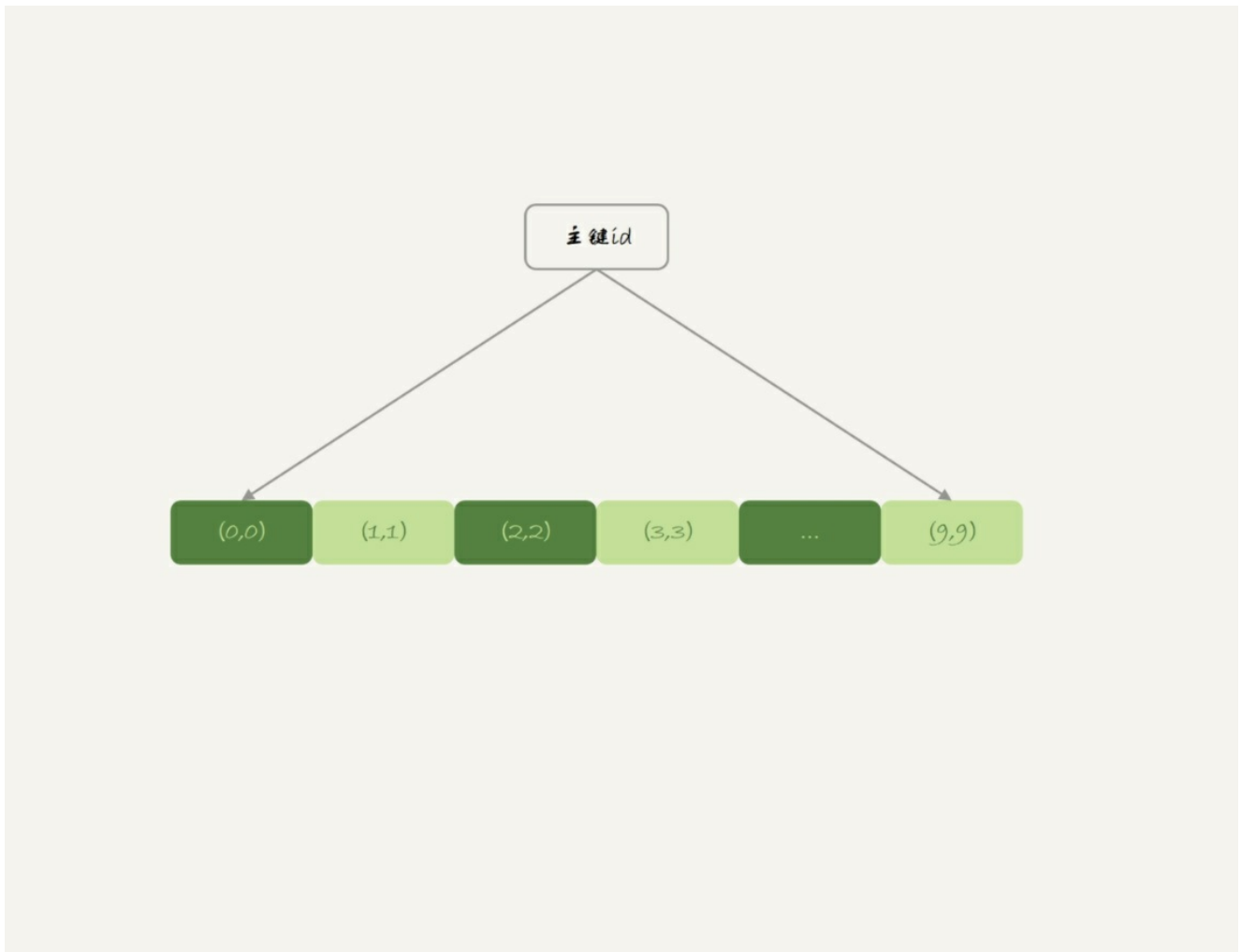


图 2 表 t2 的数据组织

主键索引上的值是有序存储的。在执行 `select *` 的时候，就会按照叶子节点从左到右扫描，所以得到的结果里，0 就出现在第一行。

与 InnoDB 引擎不同，Memory 引擎的数据和索引是分开的。我们来看一下表 t1 中的数据内容。

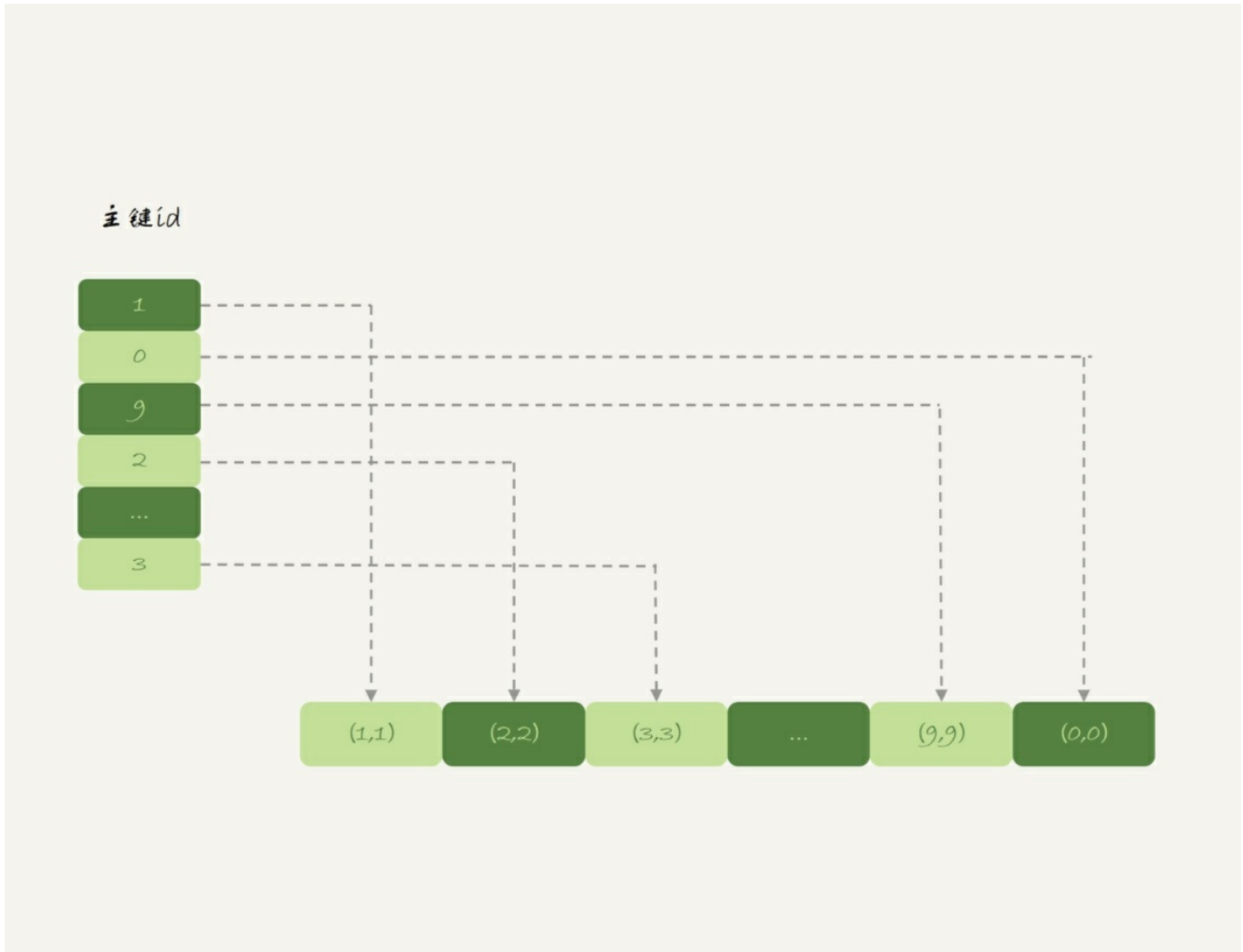


图 3 表 t1 的数据组织

可以看到，内存表的数据部分以数组的方式单独存放，而主键 id 索引里，存的是每个数据的位置。主键 id 是 hash 索引，可以看到索引上的 key 并不是有序的。

在内存表 t1 中，当我执行 `select *` 的时候，走的是全表扫描，也就是顺序扫描这个数组。因此，0 就是最后一个被读到，并放入结果集的数据。

可见，InnoDB 和 Memory 引擎的数据组织方式是不同的：

InnoDB 引擎把数据放在主键索引上，其他索引上保存的是主键 id。这种方式，我们称之为索引组织表（Index Organized Table）。

而 Memory 引擎采用的是把数据单独存放，索引上保存数据位置的数据组织形式，我们称之为堆组织表（Heap Organized Table）。

从中我们可以看出，这两个引擎的一些典型不同：

InnoDB 表的数据总是有序存放的，而内存表的数据就是按照写入顺序存放的；

当数据文件有空洞的时候，InnoDB 表在插入新数据的时候，为了保证数据有序性，只能在固定的位置写入新值，而内存表找到空位就可以插入新值；

数据位置发生变化的时候，InnoDB 表只需要修改主键索引，而内存表需要修改所有索引；

InnoDB 表用主键索引查询时需要走一次索引查找，用普通索引查询的时候，需要走两次索引查找。而内存表没有这个区别，所有索引的“地位”都是相同的。

InnoDB 支持变长数据类型，不同记录的长度可能不同；内存表不支持 Blob 和 Text 字段，并且即使定义了 varchar(N)，实际也当作 char(N)，也就是固定长度字符串来存储，因此内存表的每行数据长度相同。

由于内存表的这些特性，每个数据行被删除以后，空出的这个位置都可以被接下来要插入的数据复用。比如，如果要在表 t1 中执行：

```
delete from t1 where id=5;
```

```
insert into t1 values(10,10);
```

```
select * from t1;
```

就会看到返回结果里，id=10 这一行出现在 id=4 之后，也就是原来 id=5 这行数据的位置。

需要指出的是，表 t1 的这个主键索引是哈希索引，因此如果执行范围查询，比如

```
select * from t1 where id<5;
```

是用不上主键索引的，需要走全表扫描。你可以借此再回顾下[第 4 篇文章](#)的内容。那如果要想让内存表支持范围扫描，应该怎么办呢？

## hash 索引和 B-Tree 索引

实际上，内存表也是支持 B-Tree 索引的。在 id 列上创建一个 B-Tree 索引，SQL 语句可以这么写：

```
alter table t1 add index a_btree_index using btree (id);
```

这时，表 t1 的数据组织形式就变成了这样：

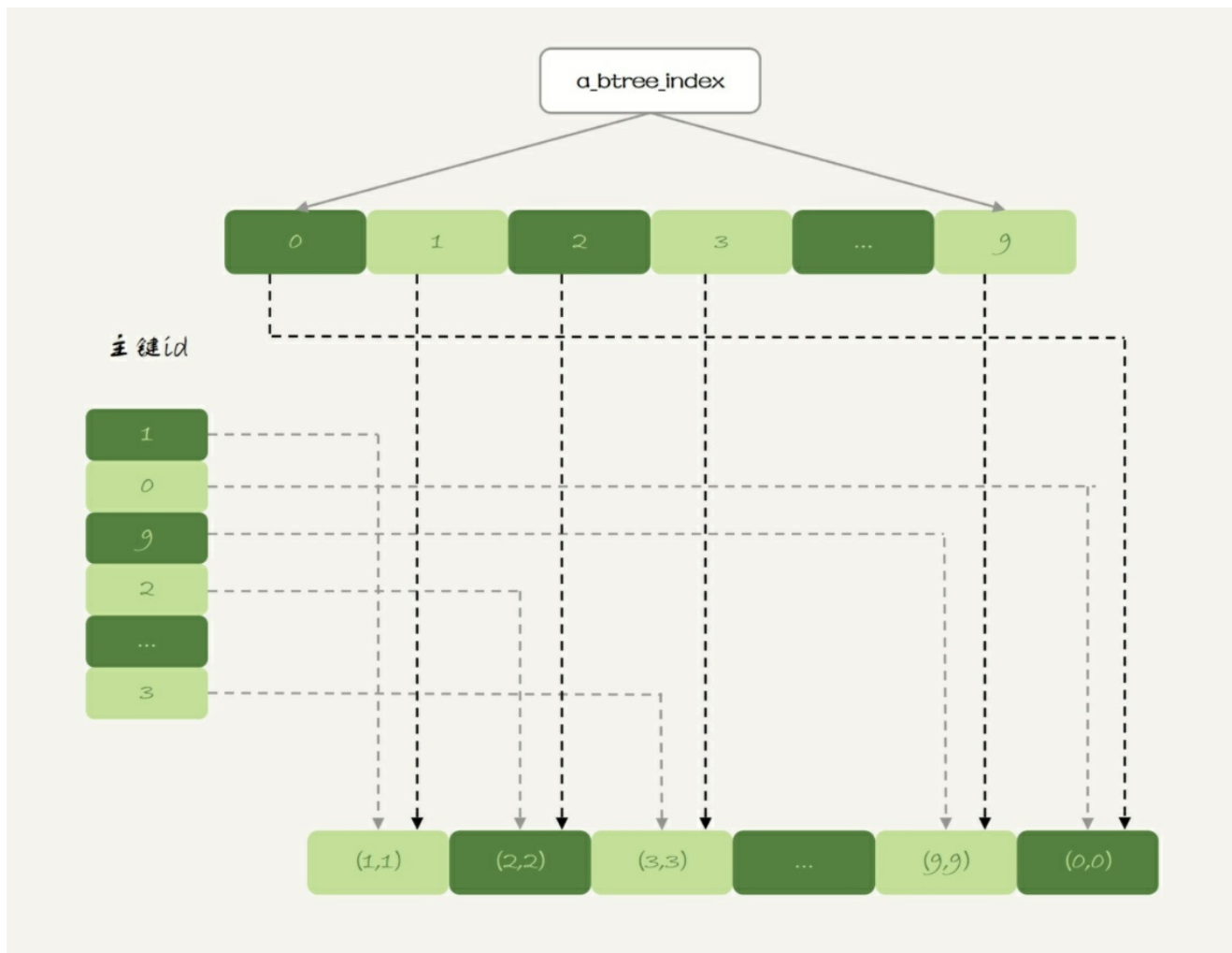


图 4 表 t1 的数据组织 -- 增加 B-Tree 索引

新增的这个 B-Tree 索引你看着就眼熟了，这跟 InnoDB 的 b+ 树索引组织形式类似。

作为对比，你可以看一下这下面这两个语句的输出：

```
mysql> select * from t1 where id<5;
+----+-----+
| id | c      |
+----+-----+
| 0  | 0      |
| 1  | 1      |
| 2  | 2      |
| 3  | 3      |
| 4  | 4      |
+----+-----+
5 rows in set (0.00 sec)
```

```
mysql> select * from t1 force index(primary) where id<5;
+----+-----+
| id | c      |
+----+-----+
| 1  | 1      |
| 2  | 2      |
| 3  | 3      |
| 4  | 4      |
| 0  | 0      |
+----+-----+
5 rows in set (0.00 sec)
```

图 5 使用 B-Tree 和 hash 索引查询返回结果对比

可以看到，执行 `select * from t1 where id<5` 的时候，优化器会选择 B-Tree 索引，所以返回结果是 0 到 4。使用 `force index` 强行使用主键 `id` 这个索引，`id=0` 这一行就在结果集的最末尾了。

其实，一般在我们的印象中，内存表的优势是速度快，其中的一个原因就是 Memory 引擎支持 hash 索引。当然，更重要的原因是，内存表的所有数据都保存在内存，而内存的读写速度总是比磁盘快。

但是，接下来我要跟你说明，为什么我不建议你在生产环境上使用内存表。这里的原因主要包括两个方面：

锁粒度问题；

数据持久化问题。

## 内存表的锁

我们先来说说内存表的锁粒度问题。

内存表不支持行锁，只支持表锁。因此，一张表只要有更新，就会堵住其他所有在这个表上的读写操作。

需要注意的是，这里的表锁跟之前我们介绍过的 MDL 锁不同，但都是表级的锁。接下来，我通过下面这个场景，跟你模拟一下内存表的表级锁。

session A	session B	session C
update t1 set id=sleep(50) where id=1;		
	select * from t1 where id=2; (wait 50s)	
		show processlist;

图 6 内存表的表锁 -- 复现步骤

在这个执行序列里，session A 的 update 语句要执行 50 秒，在这个语句执行期间 session B 的查询会进入锁等待状态。session C 的 show processlist 结果输出如下：

```
mysql> show processlist;
```

Id	User	Host	db	Command	Time	State	Info
4	root	localhost:28350	test	Query	3	User sleep	update t1 set id=sleep(50) where id=1
5	root	localhost:28452	test	Query	1	Waiting for table level lock	select * from t1 where id=2
6	root	localhost:28498	test	Query	0	starting	show processlist

图 7 内存表的表锁 -- 结果

跟行锁比起来，表锁对并发访问的支持不够好。所以，内存表的锁粒度问题，决定了它在处理并发事务的时候，性能也不会太好。

## 数据持久性问题

接下来，我们再看看数据持久性的问题。

数据放在内存中，是内存表的优势，但也是一个劣势。因为，数据库重启的时候，所有的内存表都会被清空。

你可能会说，如果数据库异常重启，内存表被清空也就清空了，不会有什么问题啊。但是，在高可用架构下，内存表的这个特点简直可以当做 bug 来看待了。为什么这么说呢？

我们先看看 M-S 架构下，使用内存表存在的问题。



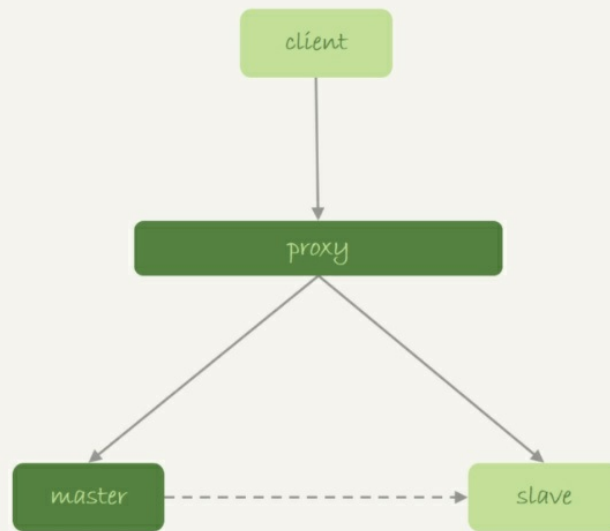


图 8 M-S 基本架构

我们来看一下下面这个时序：

业务正常访问主库；

备库硬件升级，备库重启，内存表 t1 内容被清空；

备库重启后，客户端发送一条 update 语句，修改表 t1 的数据行，这时备库应用线程就会报错“找不到要更新的行”。

这样就会导致主备同步停止。当然，如果这时候发生主备切换的话，客户端会看到，表 t1 的数据“丢失”了。

在图 8 中这种有 proxy 的架构里，大家默认主备切换的逻辑是由数据库系统自己维护的。这样对客户端来说，就是“网络断开，重连之后，发现内存表数据丢失了”。

你可能说这还好啊，毕竟主备发生切换，连接会断开，业务端能够感知到异常。

但是，接下来内存表的这个特性就会让使用现象显得更“诡异”了。由于 MySQL 知道重启之后，内存表的数据会丢失。所以，担心主库重启之后，出现主备不一致，MySQL 在实现上做了这样一件事儿：在数据库重启之后，往 binlog 里面写入一行 DELETE FROM t1。

如果你使用的是如图 9 所示的双 M 结构的话：

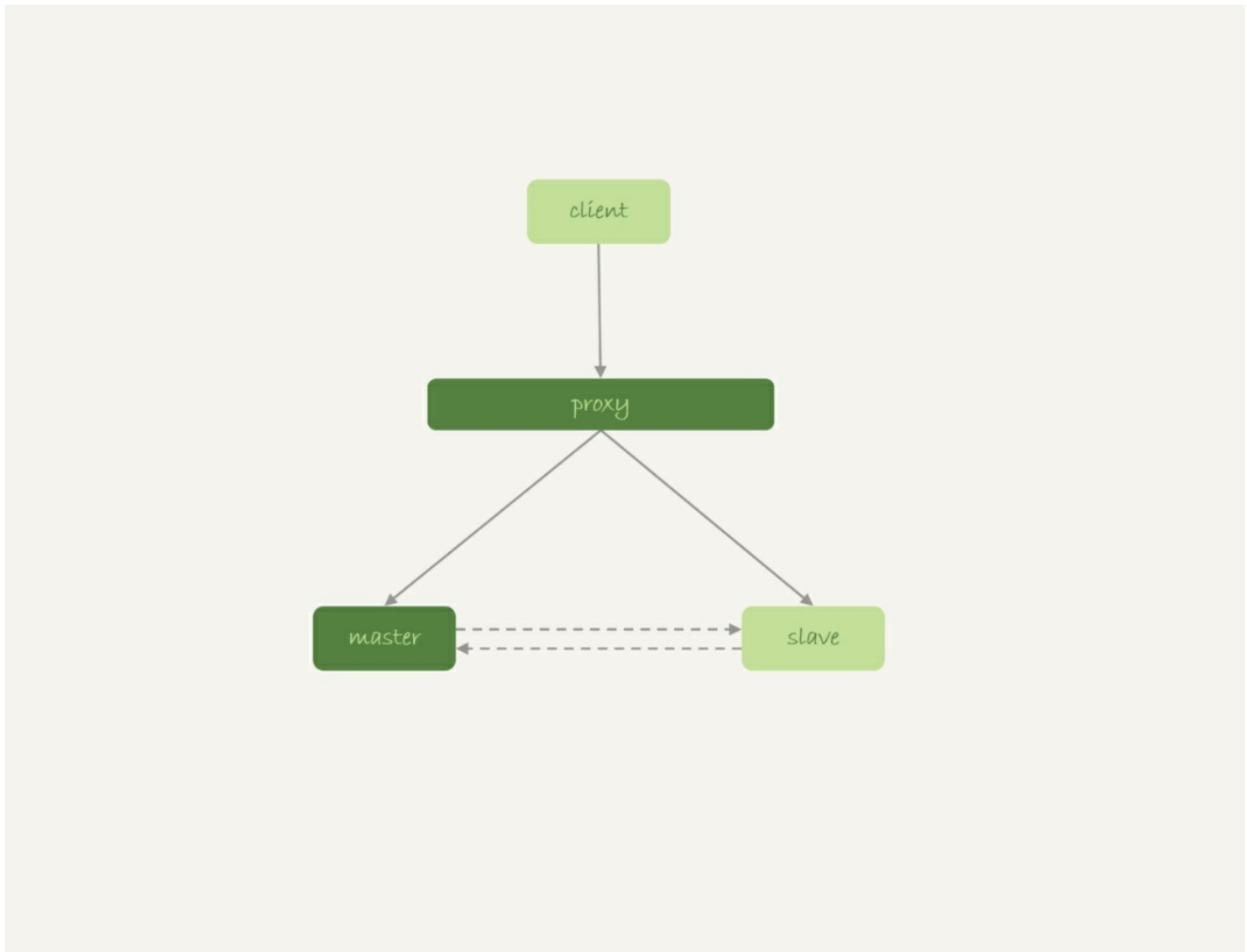


图 9 双 M 结构

在备库重启的时候，备库 binlog 里的 delete 语句就会传到主库，然后把主库内存表的内容删除。这样你在使用的时候就会发现，主库的内存表数据突然被清空了。

基于上面的分析，你可以看到，内存表并不适合在生产环境上作为普通数据表使用。

有同学会说，但是内存表执行速度快呀。这个问题，其实你可以这么分析：

如果你的表更新量大，那么并发度是一个很重要的参考指标，InnoDB 支持行锁，并发度比内存表好；

能放到内存表的数据量都不大。如果你考虑的是读的性能，一个读 QPS 很高并且数据量不大的表，即使是使用 InnoDB，数据也是都会缓存在 InnoDB Buffer Pool 里的。因此，使用 InnoDB 表的读性能也不会差。

所以，我建议你普通内存表都用 InnoDB 表来代替。但是，有一个场景却是例外的。

这个场景就是，我们在第 35 和 36 篇说到的用户临时表。在数据量可控，不会耗费过多内存的情况下，你可以考虑使用内存表。

内存临时表刚好可以无视内存表的两个不足，主要是下面的三个原因：

临时表不会被其他线程访问，没有并发性的问题；

临时表重启后也是需要删除的，清空数据这个问题不存在；

备库的临时表也不会影响主库的用户线程。

现在，我们回过头再看一下第 35 篇 join 语句优化的例子，当时我建议的是创建一个 InnoDB 临时表，使用的语句序列是：

```
create temporary table temp_t(id int primary key, a int, b int, index(b))engine=innodb;
```

```
insert into temp_t select * from t2 where b>=1 and b<=2000;
```

```
select * from t1 join temp_t on (t1.b=temp_t.b);
```

了解了内存表的特性，你就知道了，其实这里使用内存临时表的效果更好，原因有三个：

相比于 InnoDB 表，使用内存表不需要写磁盘，往表 temp\_t 的写数据的速度更快；

索引 b 使用 hash 索引，查找的速度比 B-Tree 索引快；

临时表数据只有 2000 行，占用的内存有限。

因此，你可以对第 35 篇文章的语句序列做一个改写，将临时表 temp\_t 改成内存临时表，并且在字段 b 上创建一个 hash 索引。

```
create temporary table temp_t(id int primary key, a int, b int, index (b))engine=memory;
```

```
insert into temp_t select * from t2 where b>=1 and b<=2000;
```

```
select * from t1 join temp_t on (t1.b=temp_t.b);
```

```
mysql> create temporary table temp_t(id int primary key, a int, b int, index (b))engine=memory;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into temp_t select * from t2 where b>=1 and b<=2000;
Query OK, 2000 rows affected (0.88 sec)
Records: 2000 Duplicates: 0 Warnings: 0
```

995	6	995	995	995	995
996	5	996	996	996	996
997	4	997	997	997	997
998	3	998	998	998	998
999	2	999	999	999	999
1000	1	1000	1000	1000	1000

```
1000 rows in set (0.00 sec)
```

图 10 使用内存临时表的执行效果

可以看到，不论是导入数据的时间，还是执行 join 的时间，使用内存临时表的速度都比使用 InnoDB 临时表要更快一些。

## 小结

今天这篇文章，我从“要不要使用内存表”这个问题展开，和你介绍了 Memory 引擎的几个特性。

可以看到，由于重启会丢数据，如果一个备库重启，会导致主备同步线程停止；如果主库跟这个备库是双 M 架构，还可能导致主库的内存表数据被删掉。

因此，在生产上，我不建议你使用普通内存表。

如果你是 DBA，可以在建表的审核系统中增加这类规则，要求业务改用 InnoDB 表。我们在文中也分析了，其实 InnoDB 表性能还不错，而且数据安全也有保障。而内存表由于不支持行锁，更新语句会阻塞查询，性能也未必就如想象中那么好。

基于内存表的特性，我们还分析了它的一个适用场景，就是内存临时表。内存表支持 hash 索引，这个特性利用起来，对复杂查询的加速效果还是很不错的。

最后，我给你留一个问题吧。

假设你刚刚接手的一个数据库上，真的发现了一个内存表。备库重启之后肯定是会导致备库的内存表数据被清空，进而导致主备同步停止。这时，最好的做法是将它修改成 InnoDB 引擎表。

假设当时的业务场景暂时不允许你修改引擎，你可以加上什么自动化逻辑，来避免主备同步停止呢？

你可以把你的思考和分析写在评论区，我会在下一篇文章的末尾跟你讨论这个问题。感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。

## 上期问题时间

---

今天文章的正文内容，已经回答了我們上期的问题，这里就不再赘述了。

评论区留言点赞板：

@老杨同志、@poppy、@长杰 这三位同学给出了正确答案，春节期间还持续保持跟进学习，给你们点赞。