

002

2016-02-02

架構
ARCHNOTES
高 可 用 架 构

不一样的 数据库

数据库深度解析：从 NoSQL 的历史看未来

PostgreSQL HA 高可用架构实战

单表 60 亿记录等大数据场景的 MySQL 优化与运维之道

MySQL 5.7 新特性大全和未来展望

MongoDB 里程碑式的 WiredTiger 存储引擎

肖鹏：微博数据库那些事儿

SQL

不一样的数据库

- 1 数据库深度解析：从NoSQL的历史看未来
- 21 PostgreSQL HA高可用架构实战
- 41 单表60亿记录等大数据场景的MySQL优化与运维之道
- 70 MySQL 5.7 新特性大全和未来展望
- 89 MongoDB里程碑式的WiredTiger存储引擎
- 104 肖鹏：微博数据库那些事儿

不一样的数据库

数据库深度解析：

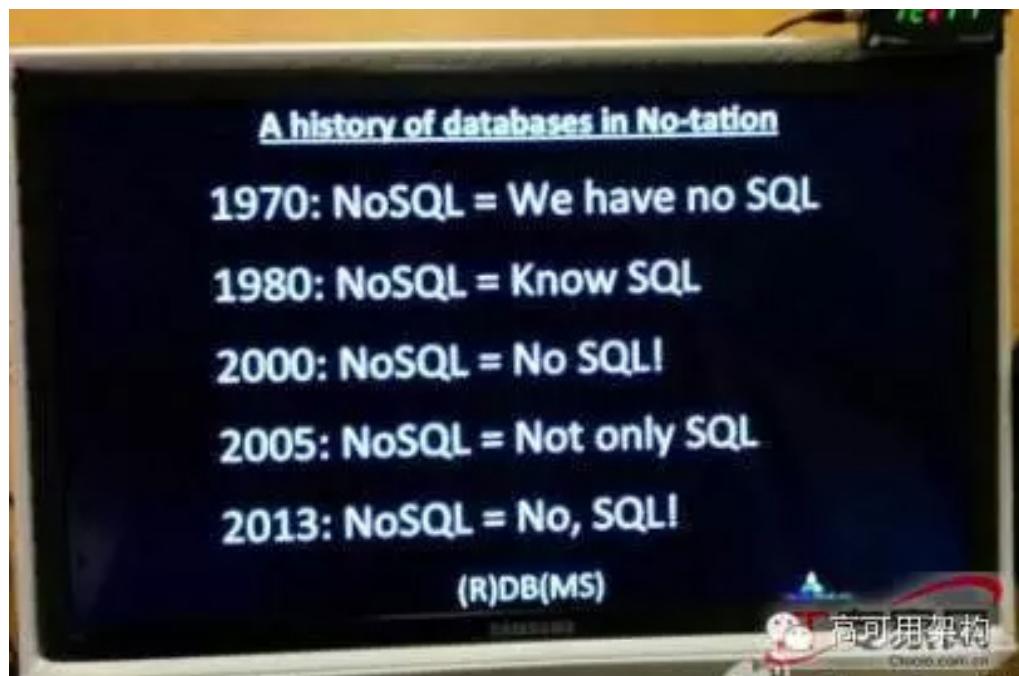
从 NoSQL 的历史看未来

作者 / 王晶昱

花名沈询，阿里资深技术专家，专注分布式数据库七年。毕业加入淘宝，在淘宝分布式数据层干了七年，参与了大部分的淘宝数据库业务架构设计工作。目前关注于分布式数据库 DRDS 和分布式消息系统 ONS 的研发工作。

前言

从 NoSQL 到 SQL，我之所以选择这个题目，其实就是因为看到了一张图：



看完以后我真的噗嗤就笑了，黑的漂亮。对一个对数据库历史有点了解的人来说，这张图真的是反映了我们在数据库存储领域螺旋上升式发展历程的最佳代表。这哥们真的是天赋异禀啊！为什么我会笑呢？希望读完本篇文章之后，大家也能跟我一样笑一下。

那么我们先来到第一章。

1970: We have no SQL

恩！我们没有SQL。

要介绍这个问题，我们就要先来看看什么叫数据库，以及数据库这个东西是怎么来的。程序员一般都会碰到类似这样的需求：用计算机表示一辆车子。这辆车呢，它有一个外壳，四扇玻璃，四个轮子。我应该如何用程序来表述它呢？首先能想到的一定是使用结构体（Java的话是Class）。

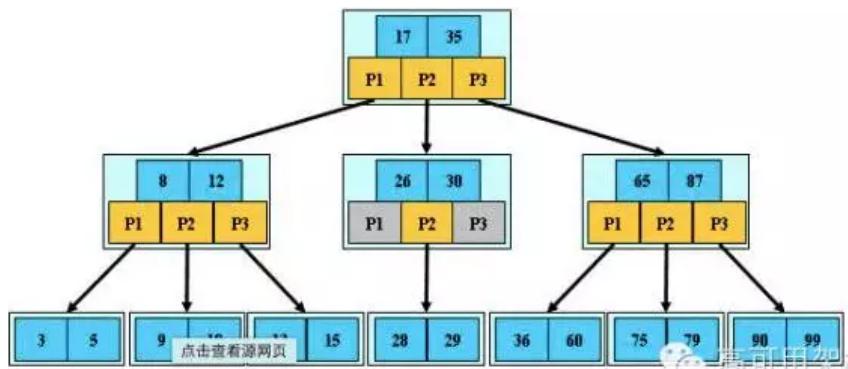
但当我们发现这个车子的项不够用了，比如我需要在车子上面装一对反光镜，怎么办呢？我们只能往里面增加一个新的属性来表示这个反光镜。如果这种需求越来越多，越来越多，我们就会发现，每次都改一下这个结构体、编译、发布，是个非常麻烦的事情。于是，就有了这种非常纯粹的需求：有没有可能把它弄成动态的。这时候，我们最常用到的一个数据结构就是“映射”。

对Java程序员来说，映射就是一个Map<Object, Object>。一般来说Map有两类实现：一类是Hash，一类是有序树。有了这个随需应变的集合，我们就可以把事情变成这样：

```
map.put("轮子", 轮子对象);  
map.put("镜子", 镜子对象) .....
```

有些时候我们又会担心数据丢失，对不对？所以还得想办法把这个对象以非常高效的方式持久化下来，放到磁盘上，这样就不容易丢失了。map.put/get操作其实都会有一次寻找的过程的，这个寻找过程对于磁盘来说会转变为一次随机寻道过程。有很多种方式能够用磁盘结构来存储类似Map这样的概念。我今天只介绍一种，就是B-tree，不知道大家对这个词儿是不是熟悉，反正我面试基本都会问问。

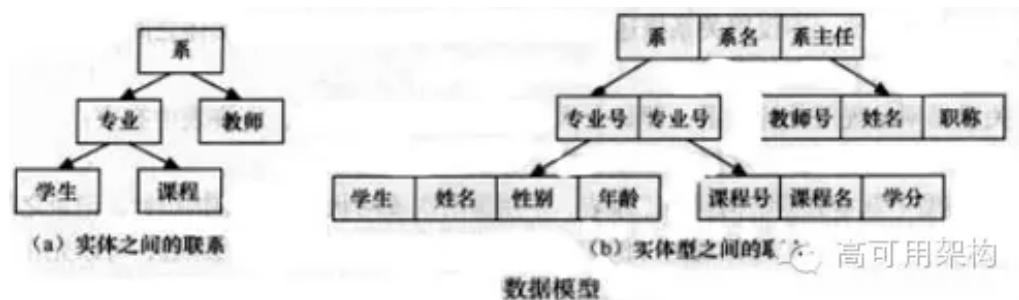
需要先提的一件事是B树 == B-树。所谓B-树，并不是B减树的意思，希望大家不要跟我一样土鳖。



这就是一个最简单的B树，观察一下就会发现，其实B树的出发点很简单：既然磁盘寻道时间很多，那就减少它，一次寻道能够从磁盘取更多数据就行了。所以，它是以“数组”为单位存储数据的（数组其实就是要连续并且有界的）。数组难以扩展，并且维护数组内元素有序也是有一定代价的。数组满了以后怎么办呢？这

就是B树会做的事情儿，分裂。如果这里大家能够联想到另外一个东西，那就算学明白了，HBase其实也就是棵巨大的、分布式的B树。其他容我最后吐槽~。

相信很多人都听过一个名词：层次数据库。这东西似乎就是在上古时代的神器，现在则不见了踪影。层次数据库到底是个什么玩意儿呢？我们来看一张图。



注：图片来自网络

抽象来看，层次模型其实就是这样的东西，我再用小汽车来表述一下：一个小车由四扇玻璃，四个轮子，两个反光镜组成。车有自己的属性，轮子有自己的属性，反光镜有自己的属性。

给大家两个例子相信大家应该立刻就能明白了，所谓的层次模型，如果用Java代码来写的话，就是Map套Map，每个Map有一些固定的属性，比如这个Map的名字是什么，这个Map的属性是什么，而这就是我们最开始在使用的数据库了。非常简单，一个Map结构搞定所有需求。看起来世界大同了。

1980: Know SQL

下面，我们就来到了1980年，Know SQL，知道SQL了。

为什么这么写？其实就是虽然关系数据库是上世纪70年代发明的，但是直到80年代，IBM发布了第一代全功能的关系数据库系统System R后，我们才正式进入到关系数据库模型。

相信很多人都觉得自己了解关系模型，似乎每个人提到它，都说“对对”。绝对对，因为这是有数学支持的，不应该被怀疑。可惜的是，如果大家了解科学发现的历史就会发现，自从爱因斯坦把牛顿那由完美数学保证的自洽理论踢出了神坛，数学自洽就再也不是真理的标准了。哪个的用户最多哪个就是真理。为什么关系模型最终赢得了比赛，而层次模型死掉了呢？很简单，因为人类都是蠢蛋和傻瓜啊。哪个简单易用，哪个就赢了。

下面，我们就以一个例子来看看关系模型易用在哪里。还是以车子为例，如果我要做这样的一个查询：把厂里生产的所有汽车里面，左轮子供应商是DRDS的轮胎都找出来。采用层次模型的代码是：

遍历每一辆车，从车对象中找到左面的轮子，查看轮子的属性，如果是DRDS，留下，不是则丢弃。

如果是关系模型呢？

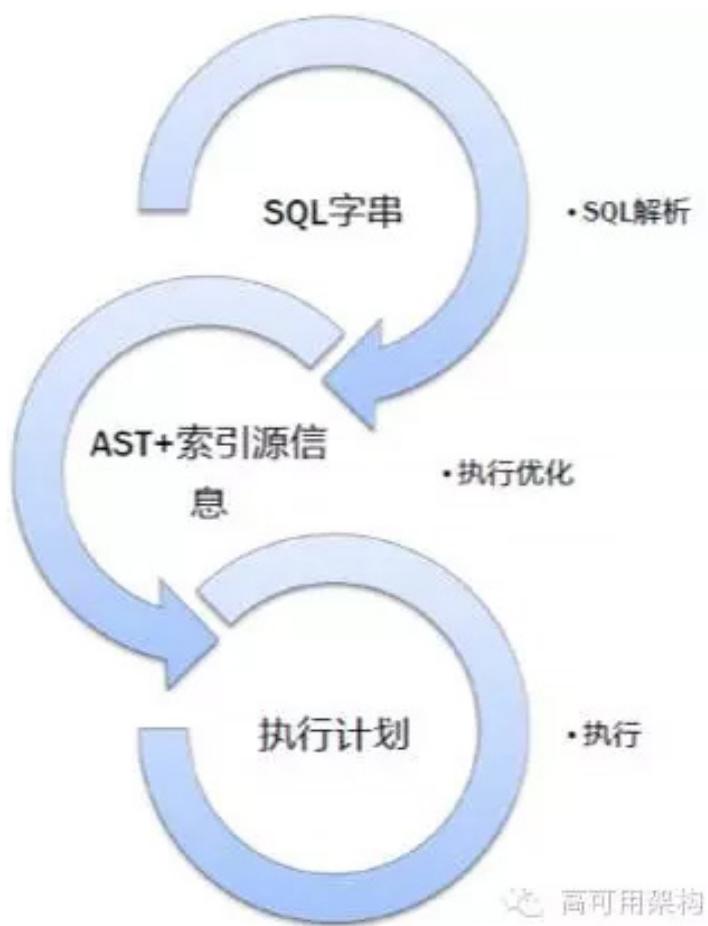
```
select * from 轮子表 where 轮子位置='左' and 轮子供应商='DRDS'  
完成。
```

我看了都觉得是个世界性的创举，不知道您是什么感觉？下一步，我们来看看关系模型将会怎么处理这条 SQL。



其实用这一张图就可以表示一个最简单的关系模型了。基本上所有的数据库都是这个组织形式：最上面的用户 API 就是执行的 SQL 和事务命令，中间的就是关系代数转换层和事务处理层、最后最底层是个 KV 存储。啊？ KV 这不是 NoSQL 的概念么，你凭什么盗用它？呵呵，谁盗用谁还不一定呢。

最底层的 KV 存储其实是我们一开始说过的“映射”结构，对应内存可能有 Hash 和有序树结构，对应磁盘则主要是 btree 树系和 LSM 树系。因为每个数据结构都有自己好玩的属性，讲起来太多了，这里就不展开了，大家可以看我博客。我们直接来聊聊关系代数引擎，这是数据库最关键的部分之一，但从功能目标来说却并不是很复杂。

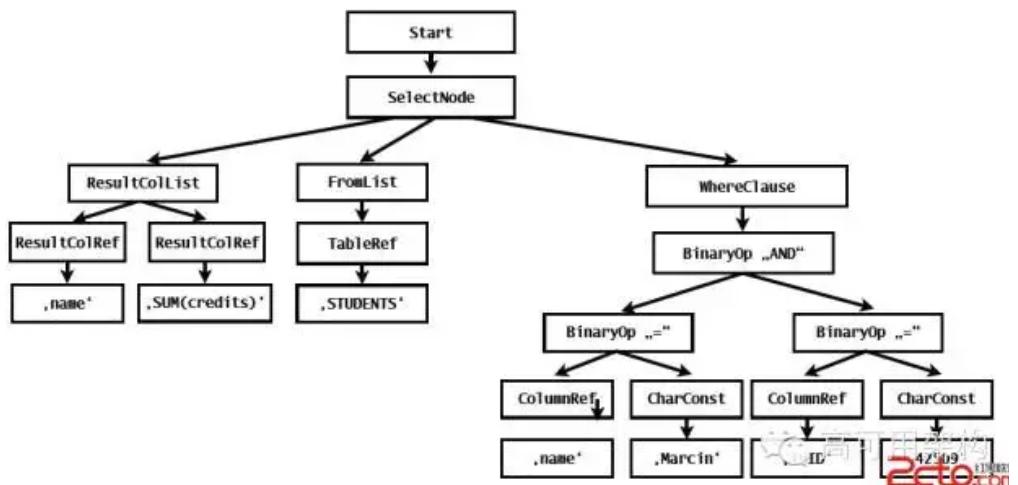


这张图就是整个关系代数引擎所经过的步骤：

最原始的是SQL字符串，类似 `select * from tab where id = 1`，它经过的过程叫SQL解析，会生成一个AST抽象语法树。

如图 6 所示：

```
SELECT name, SUM(credits) FROM STUDENTS  
WHERE name = 'Marcin' AND lvID = '42509'
```



select 被拆解为了 fromList/WhereClause 等细碎的字串。这个过程的主要作用是作为计算机编写代码而言，我们更容易识别这种结构化的数据，而文档属于非结构化数据；有了这棵树，下面就是执行优化，其入参是 AST 树 + 索引源信息。简单说来，AST 使得你可以很容易地通过在树中来回的跳跃来寻找所需的关键字信息，比如 where 条件是什么，返回哪些列等。索引源信息又是个什么鬼？要讲明白这个，得先看看关系模型和 Map 是怎么对应起来的，我用几张 PPT (很多人可能见过) 来说明：

- 如何按照key找到对应的数据

Primary Key: id	User_id	Name
0	0	袜子
1	0	鞋子
2	1	计算机
3	3	电池

–Select * from tab where id = ?

TAOBAO JM

高可用架构

第一个SQL是 select * from tab where id = ?, 上面的那个则是一个表格，如果我们用Map来表示，可以表示成这样：

Map:key -primaryKey, value -[pk,user_id,Name]

也就是以PK值作为Map的key，以一个包含了pk, userid, Name的值的结构作为Map的value (当然只包含 [userid,Name] 也OK)。有了这个Map，我们只需要从AST里面取出id = ? (假设id = 0)，通过map.get(0)拿到对应的user_id数据和Name数据，加上输入的id=0这个数据，拼成对象返回就可以了。

再来看另一个需求。

- 如何按照key找到对应的数据

Primary Key: id	User_id	Name
0	0	袜子
1	0	鞋子
2	1	计算机
3	3	电池

• Select * from tab where user_id = ?

我们来看看这个图，这里面我们的查询条件发生了变化，不是id了而是user_id。

我们刚才只有一个Map:

Map: key -> id , value -> [user_id,Name]

我应该如何利用这个Map去找到所有符合要求的结果呢？我能想到的第一种方式是遍历Map里面的每一个Entry，取出每一个Entry以后看看User_id是不是等于我要求的值，如果不等于就丢弃，等于的话返回即可。然而这种方式带来的问题是，如果我有1亿条记录，我就要做这件事1亿次。明显的O(N)效率太慢了。怎么加快一下？有需求就有人响应。于是我们可以用个空间换时间的法子：

- 如何按照key找到对应的数据

Primary Key: id	User_id	Name
0	0	袜子
1	0	鞋子
2	1	计算机
3	3	电池

- Select * from tab where user_id = ?

User_id	id
0	0
0	1
1	2
3	3

二级索引

高可用架构

看看上面的图，里面增加了一个新的Map:

Map: key -> user_id, value -> [id]

这个 Map 以 user_id 作为 key，于是我们又可以愉快而高效地用第二个 Map 的 get 接口来获取所有符合要求的 id 列表，然后再根据这个符合要求的 id 列表，去查第一个 Map，获得对应的数据了。刚才介绍的这块其实就是关系模型如何映射到 Map(也就是 KV 模型)的关键方法了。当然，还会有很多扩展性的方式和方法，不过这就不是今天的主题咯。这个数据比较小，只有三列，一个索引。如果我有十几个甚至几十个索引的时候我们又会面对另一个问题。

- 如何按照key找到对应的数据

Primary Key: id	User_id	Name
0	0	袜子
1	0	鞋子
2	1	计算机
3	3	电池

–Select ...where user_id = ? And name = 袜子

如果我有一个 user_id 的二级索引，又有一个 Name 的二级索引，我应该选择哪一个作为查询用的索引呢？是不是我需要有一种机制来选择那个最“便宜”的索引，这就是索引选择的过程。要进行索引选择就必然要知道每个索引的区分度高还是低（说白了就是一个 key 对应的 pklist.size() 少还是多），而索引的区分度高 | 低，就是所谓索引源信息的最简单模式。在真实的数据库中还有很多其他信息也是索引的源信息，不过为了方便大家理解简化了一下。有了索引源信息和 AST 树就可以生成执行计划了：

```

mysql> explain select * from t1, t2 where t1.id = t2.id and t2.col1 = 'ac';
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | t2 | ALL | PRIMARY | NULL | NULL | NULL |
| 1 | SIMPLE | t1 | eq_ref | PRIMARY | PRIMARY | 4 | Shared.t2.ID |
+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)

mysql> create index idx_col1_col2 on t2(col1,col2);
Query OK, 1001 rows affected (0.17 sec)
Records: 1001  Duplicates: 0  Warnings: 0

mysql> explain select * from t1, t2 where t1.id = t2.id and t2.col1 = 'ac';
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | t2 | ref | PRIMARY, idx_col1_col2 | idx_col1_col2 | 195 | const |
| 1 | SIMPLE | t1 | eq_ref | PRIMARY | PRIMARY | 4 | Shared.t2.ID |
+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)

```

这时候再尝试看看这个，相信大家就大概都能猜到这里的东西所表示的含义了。所以很多事儿呢不用去背，了解了背后的原理，优化就是信手拈来的事儿。这里我省略了事务步骤，这个更复杂，各位感兴趣可以看我的视频了解。

2000: No SQL!

No SQL! 豪言壮语，SQL数据库 has gone，新时代来临了！似乎一夜之间，这世界就翻了天，Facebook 开源了 Cassandra，Hadoop Hbase 横空出世，似乎这就是未来啊。

我们来看看知乎的问题：“互联网领域，传统的SQL很力不从心，一些更具有针对性的NoSQL会越来越火。以后会不会出来各种强力NoSQL？”，最雷我的一个问题大概是：“我们是一个新业务，想用NoSQL来提升开发效率，不知道Cassandra和HBase应该选哪个？”。

再见，看完以后我真的是觉得没办法好好做朋友了。

Digg采用Cassandra遭遇失败，工程副总裁离职，希望大家不是这个人。好啦，闲话扯完，回到正题。

No SQL是怎么来的？这还是得从事务说起，自从第一代产品化的数据库在上世纪80年代开发完成以后，我们的数据库主要演进模型里面只有几个有限的里程碑，我目前能记住的就这么几件事：

1. mvcc 多版本并发控制；
2. 存储过程；
3. 各类OLAP的分析类引擎。

实际上大家心里都知道，有一件事一定会发生，只是不知道什么时候会发生。这就是分布式系统。分布式系统能够具备无限的扩展能力，按需伸缩，只要有钱我们的系统就不会down，不会死。这种能力其实在上世纪80年代就早已深入人心了。还记得SUN公司提出的口号么？网络就是计算机。傻瓜都知道未来一定是分布式系统的天下，单机系统还有什么玩头？单机系统不就应该是那待宰的羔羊么？等着DRDS异军突起不就好了么？但是等啊等啊，30年过去了，却没等到自己寿终正寝的那一天，反而似乎活的越来越好了，这是为什么？理由很简单，技术没突破。

如果一个分布式系统做的跟单机系统一样方便，又能扩展，性能又好，那这世界上早就没有单机系统了。而且，从上世纪80年代到21世纪的前几年，我们实际上都不需要分布式系统。大部分的系统都是诸如“图书馆管理系统”，“客户关系管理系统”等等企业内部管理系统，不需要很高的并发，只需要容易操作就行了，而单机的关系数据库系统自然地最容易操作啊，所以单机系统大行其道。

然而，云计算和互联网的时代到来了，我们服务的对象从顶天了几千人一下就变成了十几亿人，计算机要管理的数据量呈指级别地飞速上涨，而我们却完全无法对用户数做出准确预估。这时候，扩展性、性能的要求就变得更为重要。不扩展的话业务就挂了，扩展的话开发难度少量上升，这两件事情做权衡，相信大家都能立刻知道哪个更重要。我们当然选扩展了！然而数据库却无法提供这样的扩展性，当年的淘宝也是用 Oracle 的，配置算不错的，也算是有小黑柜子。然而，今天不火的网站明天可能突然就火了，我们的用户数在一年内就会突破这个柜子的容量，折旧都来不及。很明显的，时代变了。

传统关系数据库，哪怕是 RAC 都不能满足我们对于数据库扩展性的追求了，这时候肯定有人在想：“这个有问题，我们就解决它啊”。这类技术就是 Oracle RAC 啊、MySQL Cluster 啊这类玩具，它们希望能够不改变用户行为来实现扩展性，可是做了好多年，发现玩不转。为了支撑更大的访问量和数据量，我们必然需要分布式数据库系统，然而分布式系统又必然会面对强一致性所带来的延迟提高的问题，因为网络通信本身比单机内通信代价高很多，这种通信的代价就会直接增加系统单次提交的延迟，延迟提高会导致数据库锁持有时间变长，使得高冲突条件下分布式事务的性能不升反降（具体可了解下 Amdahl 定律），甚至性能距离单机数据库有明显差距。

说了这么多我们可以发现问题的关键并不是分布式事务做不出来，而是做出来了却因为性能太差而没有什么卵用。数据库领域的高手们努力了 40 年，但至今仍然没有人能够很好地解决这个问题，Google Spanner 的开发负责人就经常在他的 Blog 上谈论延迟的问题，相信也是饱受这个问题的困扰。于是乎有一群人认为，既然

强一致性不怎么靠谱，那彻底绕开这个问题是不是更好的选择？他们发现确实有那么一些场景是不需要强一致事务的，甚至连SQL都可以不要。最典型的就是日志流水的记录与分析这类场景。去掉了事务和SQL，接口简单了，性能就更容易得到提升，扩展性也更容易实现，这就是NoSQL系统的起源。

他们喊出了非常响亮的口号：No SQL! No SQL标志着他们时代的到来。

2005: Not only SQL . 不仅仅是SQL

经过5年的忽悠，有很多人愿意相信NoSQL似乎确有其事。于是有一批先行者就开始探索各种玩法：<http://www.lampchina.net/mod-newsdo-showid-8272.html>

Digg采用Cassandra遭遇失败，工程副总裁离职：



Digg工程副总裁约翰·奎恩 (John Quinn)

玩着玩着，大家发现还是不靠谱。这不行啊，这东西不就是让我们每个业务都把关系数据库从新实现一把么？让我们退回到层次模型上去啊。对于人类这种懒惰而笨拙的动物，开历史的倒车明显是不受待见的。于是，有一批人站出来了，说No什么SQL，还是得有数据库。但NoSQL开发者们已经忽悠了那么多投资人的钱，总得有个交代啊，既然没办法颠覆，咱们就共存吧，什么NoSQL和SQL，大家一家人，各自发展就好了。这就是Not only SQL的来源。

NoSQL有哪些明确的场景呢？请注意前方高能。

比如HDFS比较火，于是就有人发现：“哎？我们如果学Google，也弄个分布式KV是不是也能火？”呵呵，我想这就是某个base最大的价值。不过在这个平缓期还是能看到一些创新性的想法的，他们帮助数据库领域往前走了不大的一步。MongoDB是个不错的思路（我个人觉得），json替代了臃肿的XML成为了一个小的标准，而在这个上面做很多索引，也是很聪明的做法，借鉴了数据库的核心思路，这也算是共存。其他的NoSQL也在往SQL上面努力，比如Cassandra的CQL、HBase的各类SQL引擎，其实都是对关系数据模型的一种妥协。毕竟，NoSQL还没有好到能够颠覆整个生态。

2013: No,SQL!

不，我们还是要关系数据库，这就是现在我们的感觉咯。经过了10年的折腾，我们还是发现关系模型目前来说是最方便表达数据存取的语言，比其他都要方便的多，所以还是妥协吧。于是所有的NoSQL都在想办法尝试支持关系数据库。然而回到初始，我们不就是因为关系数据库不能满足用户要求，所以才要做NoSQL的么。难道NoSQL弄个关系代数引擎，就能做出魔法么？其实，也不行，该有的限制一个都没少，最终大家殊途同归，还是回到了如何能够让关系数据库更具有扩展性，性能更好这条路上来，条条大路通罗马嘛，和气生财。这就是NewSQL的来源。

DRDS也是NewSQL的一员，其实说实话，我挺有感触的。作为这么多年来一直坚守在分布式数据库这个领域的人来说，能够坚持下来真的不容易，在外面有太多的诱惑，最火的时候，连DBA都去学了各种NoSQL的运维技术。然而，我们能够坚守，其实就是因为我们懂得历史也看得见现在。我们深刻地知道，科学就是承认自己并非无所不能，然后不断地往那无所不能的地方努力的一种精神。一直以来，我们都尽可能地协助用户保留关系数据库的方便性，然后想办法告知用户哪些地方目前还缺少技术突破，应该使用哪些工具来替代，所以也算是积累了非常多的经验和工具。

同时我们也在努力追求数据库领域的那个圣杯：更快的存取数据，可以按需扩缩以承载更大的访问量和更大的数据量，开发容易，硬件成本低。这是大家梦寐以求在追求的东西。也是我们在追求的。虽不能至，吾心向往之。

阿里的技术选择

最后，来聊聊阿里的技术选择。其实，所有大公司似乎都在释放各种信号，xxx在用什么系统了，xxx在用什么系统了。阿里可能不大一样，从内部来说，他也是个生态系统，用户选择什么实际上主要都是由用户自己决定的，所以阿里能够出现任何一种选择，只要能解决问题即可。TDDL DRDS这套体系，只能说是目前用的最广的一套，原因也很简单，改变行为习惯少。

双11对DRDS这套体系来说其实没什么压力，我前几年的双十一虽然都在核心作战室，不过我一般的做法都是到了那里：“您辛苦了，您也是，大家辛苦了”，然后吃吃吃。因为确实没什么好担心的啊，没有不平稳的理由，DRDS的能力更多地体现在双11开始和双11结束，我们需要在那之前机器扩容，以及那之后要机器缩容，这些才是DRDS的核心能力。上上次我确实是很紧张，因为新接了一个消息系统。哎！这种用来消峰填谷的系统才是压力山大，其余的时候基本上就没啥了，更多地跟大家一样，也都是普通的功能开发而已。■

Q&A

Q1：请问schema less/free你怎么看？

这俩似乎没见过拼一起哈，我分开。但本质是同一个东西，我个人觉得有市场。不过能有多大，不知道。

- 优势：业务模型更灵活
- 劣势：额外的空间占用

技术债也一定是要还的。我清楚地记得当年我维护的一个 CMS 系统，所有数据都是 map。结果最后有一些诡异的数据不知道什么时候被塞到里面。然后最后也没人知道是在哪里塞的。debug 都很难找到。所以一般来说，结合会更好一些。目前 pg/mysql 都开始支持 json 了。这东西其实只是工作量问题。没什么技术上的难度。

Q2：最近很多声音说不要再用 MongoDB，你怎么看？

这个就纯属个人意见了啊。我个人不喜欢 MongoDB 那帮人的嘴脸。MongoDB 核心贡献者：不是 MongoDB 不行，而是你不懂！

<http://www.cnblogs.com/shanyou/archive/2012/11/17/2774344.html>

然而，为什么会这样，还不是某些人为了骗分，默认配置特别激进么。而开发者不会告诉你，如果改成安全配置，他们的性能没比 SQL 强哪里去。

一个存储，最少10年才能稳定。前些天刚碰到一个游戏客户说某NoSQL数据文件损坏无法恢复，问我们有没有办法，我说，下次选择谨慎点，性能不是唯一，这次请节哀。

Q3：非结构化数据，如文本，树图等，这些SQL无法处理的，是否使用NoSQL更合理？

非结构化数据也是一个重要的门类，一直都存在，以后也会存在，但NoSQL为了宣传，把所有的东西都拉到自己阵营，这其实与其初衷已经违背。

Twitter的图数据库其实也是依托MySQL做的。你给我钱给我人，给我时间，我能用汇编写任何东西。

Q4：SQL模型已经那么多年了，数据库领域有其他可能更好使的语言模型方向么？

目前还没见到更好地抽象，所谓好使就是你发现他占据了更多江山。在历史上有很多次尝试，比如对象数据库等，但实际上也都是针对SQL问题的一些改善。不过目前还没有特别成功的。

不一样的数据库

PostgreSQL HA 高可用架构实战

作者 / 萧少聪

花名铁庵，广东中山人，阿里云 RDS for PostgreSQL/PPAS 云数据库产品经理。自 2006 年以来，长期从事 RedHat 及 SuSE Linux 的 HA 集群搭建及 PostgreSQL 数据库支持工作。2011 年开始组建 Postgres (数据库) 中国用户会。

PostgreSQL 背景介绍

有不少同学希望了解 PostgreSQL 的背景及它与 MySQL 的对比结果，所以在此啰唆两句，有兴趣的同学可以单独给我发 E-Mail，我可以分享详细的介绍及一些对比结果。

2015 年是 PostgreSQL 正式在中国起步的一年，我们看到越来越多的企业选择了 PostgreSQL。

- 中国移动主动使用 PostgreSQL 实现分布式数据库架构。
- 金融业方面平安集团明确表示将使用 PostgreSQL 作为新一代数据库的选型。
- 华为中兴纷纷加入 PostgreSQL 内核研究队伍。
- 阿里云正式提供 PostgreSQL 服务。

大部分人了解 MySQL 应该都是从 2005 年左右开始，那时在互联网带动下 LAMP 空前繁荣。而你所不知道的是，那时 PostgreSQL 已发展了近 30 年，至今已经超过 40 年。1973 年 Michael Stonebraker (2014 年图灵奖得主) 在伯克利分校研发了当前全球最重要的

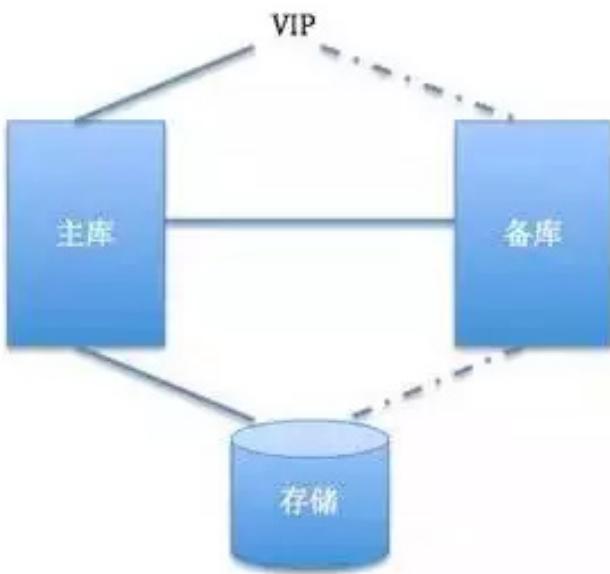
关系型数据库实现: Ingres。此后, 陆续改名为 Postgres、Postgres95, 直到现在的 PostgreSQL。PostgreSQL 有众多的衍生品牌产品, 就如同 Linux 有 RedHat、SUSE、Ubuntu 一样, 当前, 国内多个国产数据库都是基于 PostgreSQL 进行开发的, 同时, 国际知名的针对 OLAP 场景的 Greenplum 数据库, 及 EnterpriseDB 公司高度兼容 Oracle 语法的 PPAS 数据库也是基于 PostgreSQL 实现。

PostgreSQL 与 MySQL 相比功能更为完善, 同时, 在进行复杂 SQL 查询时 (特别是多表进行 JOIN 查询) 性能及稳定性也更为优秀, 是国外企业首选的应用于核心业务系统的开源 OLTP 业务关系型数据库引擎。PostgreSQL 被誉为全球最先进的开源数据库, 支持 NoSQL JSON 数据类型、地理信息处理 PostGIS、丰富的存储过程操作, 并可实现基于 Tuple (在 PostgreSQL 中此单位比 Block 还要小) 级别的 StreamingReplication 数据同步。

与 MySQL 不同, PostgreSQL 不支持多数据引擎。但支持 Extension 组件扩充, 以及通过名为 FDW 的技术将 Oracle、Hadoop、MongoDB、SQLServer、Excel、CSV 文件等作为外部表进行读写操作, 因此, 可以为大数据与关系型数据库提供良好对接。

在 PostgreSQL 下如何实现数据复制技术的 HA 高可用集群

业界大多数的数据库的 HA 实现都是基于共享存储方式的, 如下图。在这个方式下, 数据库 1 主 1 备, 使用一个共享存储保存数据。



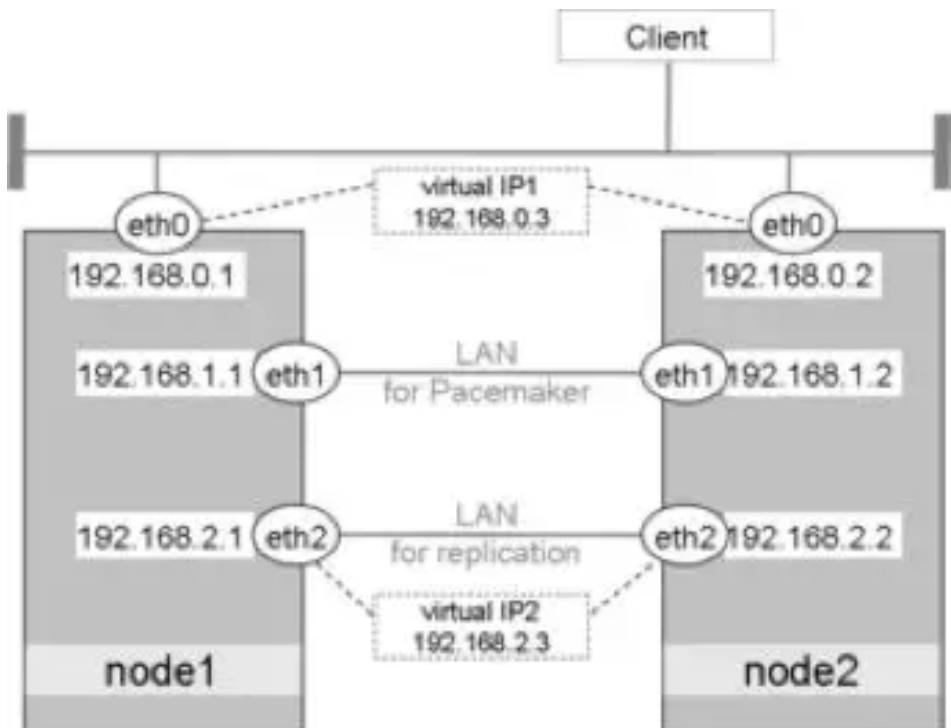
正常情况下主库连接存储及 VIP，进行数据业务处理。备库永远处于非运行状态，只有当主库出现故障后，备库才会进行存储及 VIP 的接管。但传统的企业中，这样的结构比比皆是，在我进入阿里云之前服务过的大多数企业都使用这样的架构（除了 Oracle RAC 及 DB2 的并行方案）。而当今，无论 Oracle、MySQL、SQLServer，还是今天我们用作说明案例的 Postgres，都已经支持基于数据库底层的 StreamingReplication 模式实现数据复制了，同时支持备库作为只读服务器提供业务服务。因此，备库资源对于企业来说是极大的浪费。

传统的 HA 方案在实现基于 Streaming Replication 方式时，往往需要通过大量人为控制的脚本进行判断和控制。2006 年到 2011 年，我为不同的客户及不同的数据库编写了多种特制的脚本，当中的安装配置及维护难度都有点让人望而却步。2011 年，我在 SUSE 系统的 HA 支持工作中接触到了 Corosync + Pacemaker 的 HA 结构。

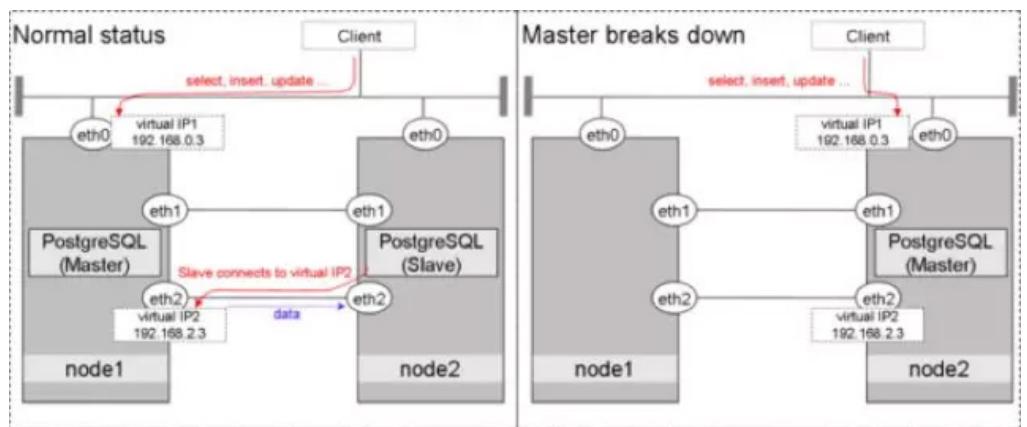
发现了“Master-Slave模式”。在这个模式下，系统支持 promote 及 demote，以解决数据库基于 Streaming Replication 主备模式的切换问题。

Corosync + Pacemaker MS 模式介绍

本次讲解主要针对架构及这个模式的处理原理。如果大家想要了解具体的配置方式，可以参考 <http://clusterlabs.org/wiki/PgSQLReplicatedCluster>。同时，当前最新的 Red Hat Enterprise Linux 7 及 SUSE Linux Enterprise Server 11/12 中的 HA 组件都基于此架构，你也可以通过厂商的官方文档或官方技术支持得到配置的详细说明。



上图有3个网段：0.x网段，用于数据库对外业务；1.X网段，用于Pacemaker心跳通讯；2.X网段，用于数据库的数据复制。同时提供主库读写服务VIP1 192.168.0.3和备库只读服务VIP2 192.168.2.3。用户的主应用程序可以通过VIP1进行读写操作，而只读处理可以通过VIP2实现。



上图中，左边是正常运行的模式：所有读写操作通过VIP1进入到Master节点；Slave节点的会连接到VIP2，通过此IP支持只读操作；Streaming Replication通过eth2进行两节点的数据同步。右边是Master故障时的模式：原Slave会promote成为Master节点；VIP1切换到node2继续提供服务；VIP2切换到node2继续提供服务。

在此处，系统从node1切换到node2有多种可能性。

1. Master 节点通过 pacemaker 控制人为进行 Switchover 切换。这种情况下主备模式会进行调换，并且过程中可以保证所

有 Master 节点中的数据会复制到 Slave 后再进行 node2 上的 promote 操作。因此，数据库中所有的事务都是完整的，且不会出现任何数据丢失。这种情况大多用于硬件需要进行主动维护时。

2. Master 节点意外出现故障时，将进行 Failover。由于 PostgreSQL 在双节点推荐使用的是 async 模式，因此如果 Master 节点故障时还有数据没来得及复制到 Slave。这些数据将丢失，但由于 PostgreSQL 的 Streaming Replication 是以事务为单位的，因此数据库的事务一致性是可以得到保障的，绝对不会出现备库中某个事务只恢复到一半的情况。

当前有一个比较严重的问题，就是如上图所示，切换后 node1 如果想要重新成为主节点，将需要重新进行全量的数据复制恢复。这是因为 Master 故障时如果有数据没复制到 Slave，Master 的最后一个事务时间将比 Slave 中的事务时间更新（如 Master 最后一个事务号为 1001，但 Slave 中的事务只恢复到 999）。此时 Slave 节点 promote 成为新的 Master 后，所有新的操作将由 999 号事务的结果为基础。也就是说原 Master 中的 1000 及 1001 事务所处理的数据将不可恢复。由于在当前设计中数据库中已经提交的事务不支持直接回退，所以，如果你的数据库到达 TB 级别，这将需要 6~7 小时。

但这个情况很快将会被改善。PostgreSQL 9.5 将为用户提供 pg_rewind 功能。当 Master 节点 Failover 后，原 Master 节点可以通过 pg_rewind 操作实现故障时间线的回退。回退后再从新的主库中获取最新的后续数据。因此，虽然之前没有提交的事务由于

ACID 原则无法重新使用，但原 Master 的数据无须进行重新全量初始化就可以继续进行 Streaming Replication，并作为新的 Slave 使用。

Corosync + Pacemaker M/S 环境配置

以下内容中截图来自于 <http://clusterlabs.org/wiki/PgSQLReplicatedCluster>。

Corosync + Pacemaker M/S 配置环境准备

- RA: Resource Agent 资源代理，PostgreSQL 最新的 RA 可以通过 <https://github.com/ClusterLabs/resource-agents/blob/master/heartbeat/pgsql> 下载。如果你发现这个 RA 不符合你的需求，也可以自行改写。
- 操作系统版本：Fedora 19 及以上、Red Hat Enterprise Linux 7 及以上、SUSE Linux Enterprise Server 11 SP3 及以上。
- 数据库要求：PostgreSQL 9.1 及以上，由于 PostgreSQL 9.1 以上才支持 Streaming Replication，因此，比这个版本低的数据库无法实现此功能。
- 两台服务器配置相同的 NTP 时间源及相同的时区。

通过 yum、zypper 安装 pacemaker（主要用于 HA 资源管理）、corosync（HA 心跳同步控制）、pcs3（HA 的命令行配置工具）。通过 yum、zypper 或任何其他方式安装 PostgreSQL 数据库，安

装时务必确认其 pg_ctl 命令、 psql 命令、 data 目录的存放位置，因为配置时要用到。

Packages (both nodes)

```
# yum -y install postgresql-server pacemaker corosync pcs
```

PostgreSQL Streaming Replication 配置 在 node1 中 初始化 PostgreSQL 数据库。

PostgreSQL (node1 only)

```
# su - postgres
$ mkdir /var/lib/pgsql/pg_archive
$ cd /var/lib/pgsql/data
$ initdb
```

对其 postgresql.conf 文件做如下修改。

```
listen_addresses = '*'
wal_level = hot_standby
synchronous_commit = on
archive_mode = on
archive_command = 'cp %p /var/lib/pgsql/pg_archive/%f'
max_wal_senders=5
wal_keep_segments = 32
hot_standby = on
restart_after_crash = off
replication_timeout = 5000
wal_receiver_status_interval = 2
max_standby_streaming_delay = -1
max_standby_archive_delay = -1
synchronous_commit = on
restart_after_crash = off
hot_standby_feedback = on
```

注意 wal_level = hot_standby, 使得日志支持 Streaming Replication; archive_mode = on, 启动归档模式; archive_command = 'xxx', 指定归档的保存方法; hot_standby = on, 备库启动为 standby 模式时可实现只读查询; 其他参数主要用于性能及延迟的设定。

将 data 目录下的 pg_hba.conf 文件做如下修改。

```
host    all            all      127.0.0.1/32      trust
host    all            all      192.168.0.0/16    trust
host    replication    all      192.168.0.0/16    trust
```

注意 这样配置后，所有 192.168.x.x 网段的 IP 都将可以无密码对此数据库进行访问，安全性可能会降低。因此，只作为练习使用，在生产环境中请严格控制 IP。如指定只 trust 某 IP 可以写成 192.168.100.123/32。

配置完成后，启动 node1 上的 PostgreSQL。

```
$ pg_ctl -D /var/lib/pgsql/data/ start
```

在 node2 进行数据初始化。

PostgreSQL (node2 only)

Copy data from node1 to node2

```
# su - postgres
$ rm -rf /var/lib/pgsql/data/*
$ pg_basebackup -h 192.168.2.1 -U postgres -D /var/lib/pgsql/data -X stream -P
$ mkdir /var/lib/pgsql/pg_archive
```

注意 通过pg_basebackup命令将从node1中将所有数据库中的数据都同步到/var/lib/pgsql/data去。

数据basebackup完成后，在node2中的data目录下建立recovery.conf文件并录入以下内容。

```
standby_mode = 'on'
primary_conninfo = 'host=192.168.2.1 port=5432 user=postgres application_name=node2'
restore_command = 'cp /var/lib/pgsql/pg_archive/%f %p'
recovery_target_timeline = 'latest'
```

注意 primary_conninfo指定了主服务器所在的位置、replicate所使用的用户名。由于我们在pg_hba.conf中使用trust方式，所以在此参数中不需要加入password。

配置完成后，启动node2上的PostgreSQL，准备检查同步效果。

```
$ pg_ctl -D /var/lib/pgsql/data/ start
```

如果在node1中通过psql命令登录数据库后可以得到以下信息，证明数据库端的Replication已运行正常。

```
# su - postgres
$ psql -c "select client_addr,sync_state from pg_stat_replication;"
 client_addr | sync_state
-----+-----
 192.168.2.2 | async
```

自此，PostgreSQL的Streaming Replication配置完成，两个数据库的数据将进行持续复制。

注意 以上两个服务器已经完成 Streaming Replication 配置，在配置 HA 前请将两个服务器上的 PostgreSQL 都停止。因为在 HA 架构中，所有资源都应该是由 HA 软件进行管理的，所以与此同时也请确认系统启动时 PostgreSQL 不会自动启动（你可以通过 chkconfig 检查）。

```
$ pg_ctl -D /var/lib/pgsql/data stop  
$ exit
```

Corosync + Pacemaker HA 基础配置

corosync 配置文件只有一个，/etc/corosync/corosync.conf。

```
quorum {  
    provider: corosync_votequorum  
    expected_votes: 2  
}  
aisexec {  
    user: root  
    group: root  
}  
service {  
    name: pacemaker  
    ver: 0  
}  
totem {  
    version: 2  
    secauth: off  
    interface {  
        ringnumber: 0  
        bindnetaddr: 192.168.1.0  
        mcastaddr: 239.255.1.1  
    }  
}  
logging {  
    to_syslog: yes  
}
```

我们可以看到，当前 quorum 中 expected_votes 为 2，这是因为我们使用 2 节点。totem 中有 bindnetaddr:192.168.1.0 及 mcastaddr: 239.255.1.1，这里说明 corosync 会使用本服务器上 192.168.1.X 网段的 IP 作为心跳。此处注意，不需要写明此 IP 的详细地址，系统会自动发现。通过 scp 命令将此文件复制到 node2 中相同的目录并保证其权限一致。

接下来就可以在两个节点中启动 corosync 了。以下是系统在 Fedora 19、RHEL7、SUSE12 后的服务启动命令。如果你使用的是低版本操作系统，请用 /etc/init.d/corosync start 或 service corosync start。

```
# systemctl start pacemaker.service
```

pacemaker 默认情况下是无污染的，但为了保证 HA 初始状态我们会进行以下操作。此操作会清空所有 HA 资源的配置。它在另外一些情况下也十分实用，如有时我们会发现两个节点 HA 启动时资源信息不同步。此时我们可以先择定一个可信的节点，然后将另一节点上的 cib 文件清空，然后进再启动 pacemaker，这样新节点就会自动同步现有节点的所有配置。

```
# crm_mon -Afr -l

Last updated: Mon Jul  8 09:46:27 2013
Last change: Mon Jul  8 09:46:27 2013 via crmd on node1
Stack: corosync
Current DC: node1 (402696384) - partition with quorum
Version: 1.1.9-3.fc19-781a388
2 Nodes configured, unknown expected votes
0 Resources configured.

Online: [ node1 node2 ]

Full list of resources:

Node Attributes:
* Node node1:
* Node node2:

Migration summary:
* Node node1:
* Node node2:
```

Pacemaker 资源配置

通过pcs命令行工具进行HA资源的配置。pcs命令行可以协助生成名为 config.pcs 的配置脚本，以进行最后的 HA 配置导入。首先，我们进行一个全局信息的配置，指明由于当前是 2 节点，所以忽略 no-quorum-policy；默认的 resource-stickiness 为 INFINITY，即任何资源默认都是与其他资源可共同运行的；默认的 migration-threshold 为 1，即任何情况下 migration 时都会重试一次。

```
pcs cluster cib pgsql_cfg

pcs -f pgsql_cfg property set no-quorum-policy="ignore"
pcs -f pgsql_cfg property set stonith-enabled="false"
pcs -f pgsql_cfg resource defaults resource-stickiness="INFINITY"
pcs -f pgsql_cfg resource defaults migration-threshold="1"
```

注 意 stonith-enabled="false" 表示不使用任何电源控制设备，这个情况不建议在生产中使用。熟悉 RHEL 集群的同学可以认为 Stonith 等同于 Fence 设备。

配置 VIP1 及 VIP2，以及 pgsql 资源。

```
pcs -f pgsql_cfg resource create vip-master IPAddr2 \
    ip="192.168.0.3" \
    nic="eth0" \
    cidr_netmask="24" \
    op start timeout="60s" interval="0s" on-fail="restart" \
    op monitor timeout="60s" interval="10s" on-fail="restart" \
    op stop timeout="60s" interval="0s" on-fail="block"

pcs -f pgsql_cfg resource create vip-rep IPAddr2 \
    ip="192.168.2.3" \
    nic="eth2" \
    cidr_netmask="24" \
    meta migration-threshold="0" \
    op start timeout="60s" interval="0s" on-fail="stop" \
    op monitor timeout="60s" interval="10s" on-fail="restart" \
    op stop timeout="60s" interval="0s" on-fail="ignore"
```

```
pcs -f pgsql_cfg resource create pgsql pgsql \
    pgctl="/usr/bin/pg_ctl" \
    psql="/usr/bin/psql" \
    pgdata="/var/lib/pgsql/data/" \
    rep_mode="sync" \
    node_list="node1 node2" \
    restore_command="cp /var/lib/pgsql/pg_archive/%f tp" \
    primary_conninfo_opt="keepalives_idle=60 keepalives_interval=5 keepalives_count=5" \
    master_ip="192.168.2.3" \
    restart_on_promote='true' \
    op start timeout="60s" interval="0s" on-fail="restart" \
    op monitor timeout="60s" interval="4s" on-fail="restart" \
    op monitor timeout="60s" interval="3s" on-fail="restart" role="Master" \
    op promote timeout="60s" interval="0s" on-fail="restart" \
    op demote timeout="60s" interval="0s" on-fail="stop" \
    op stop timeout="60s" interval="0s" on-fail="block" \
    op notify timeout="60s" interval="0s"
```

vip-master(VIP1) 及 vip-rep(VIP2) 相对比较好理解。而在 pgsql 资源中，如果大家有熟悉 Linux 集群的会发现，一般情况下 HA 中添加应用资源都会加入一个带有 start/stop/status 的脚本。而此处是通过一个 agent 实现，我们只要配置好 PostgreSQL 的

pgctl、psql、pgdata的文件或目录位置即可，处理十分方便。主要因为 PostgreSQL RA 已经包含 start|stop|status|monitor|promote|demote|notify 的操作脚本 (<https://github.com/ClusterLabs/resource-agents/blob/master/heartbeat/pgsql>)。感谢开源，感谢贡献者吧，这里头有 2070 行代码，相比我以前自己写的要精妙得多。

将以上的IP及pgsql资源进行关联，这也是pacemaker最精妙的地方。我们可以看到，首先，有一个“resource master”我们命名为 msPostgresql，pgsql 属于这个 Master 模式资源。这个模型的资源，基于 clone 模型，将会在两个节点同时启动。然后，建立了一个 master-group，将 vip-master 及 vip-rep 加到这个组中。接下来，constraintcolocation 指定了 master-group 中的资源 (vip-master 及 vip-rep) 倾向于与 msPostgresql 的 Master 节点运行在一起。最后，orderpromote 及 order demote 负责管理节点的启动顺序。

```
pcs -f pgsql.cfg resource master msPostgresql pgsql \
    master-max=1 master-node-max=1 clone-max=2 clone-node-max=1 notify=true
pcs -f pgsql.cfg resource group add master-group vip-master vip-rep
pcs -f pgsql.cfg constraint colocation add master-group with Master msPostgresql INFINITY
pcs -f pgsql.cfg constraint order promote msPostgresql then start master-group symmetrical=false score=INFINITY
pcs -f pgsql.cfg constraint order demote msPostgresql then stop master-group symmetrical=false score=0
pcs cluster push cib pgsql.cfg
```

注意 Master 节点会由 RA 自动识别；msPostgresql 进行 promote 以后才会进行 master-group 的 IP 挂接；同时，在进行 demote 时也是只有等 msPostgresql 完成停库后才进行 master-group 的 IP 断开处理。

```
# sh config.pcs
```

此处的 config.pcs 是由前面的 pcs 所生成，通过 crm_mon 你可以看到所有的资源情况。

```
# crm_mon -Afr -l

Last updated: Mon Jul  8 10:24:21 2013
Last change: Mon Jul  8 10:22:14 2013 via crm_attribute on node1
Stack: corosync
Current DC: node2 (419473600) - partition with quorum
Version: 1.1.9-3.fc19-781a388
2 Nodes configured, unknown expected votes
4 Resources configured.

Online: [ node1 node2 ]

Full list of resources:

Resource Group: master-group
    vip-master (ocf::heartbeat:IPAddr2):           Started node1
    vip-rep   (ocf::heartbeat:IPAddr2):           Started node2
Master/Slave Set: msPostgresql [pgsql]
    Masters: [ node1 ]
    Slaves: [ node2 ]

Node Attributes:
* Node node1:
    + master-pgsql                         : 1000
    + pgsql-data-status                     : LATEST
    + pgsql-master-baseline                : 0000000009000080
    + pgsql-status                          : PRI
* Node node2:
    + master-pgsql                         : 100
    + pgsql-data-status                   : STREAMING|SYNC
    + pgsql-status                        : HS:sync

Migration summary:
* Node node1:
* Node node2:
```

自此所有配置结束，你可以将node1中PostgreSQL的data目录mv到其他地方看看切换的效果，不再赘述。

关于排错，所有HA日志信息会保存在 /var/log/message 中。如果系统有问题，可以通过此日志进行分析。但有一点很重要，建议大家在配置HA前一定要确认NTP服务是否正常，保障两个服务器时间不要差距太大。不然排错会很麻烦，还会有可能导致集群的其他问题。

过去两年，这样的集群架构已经在很多企业使用，数据量多在20G到1T之间。PG是一个用于OLTP的系统，当前我所实施的企业大多是传统行业。大部份用户主要是从Oracle迁移来的。在大数据及分析方面，会先用Greenplum或Hadoop等进行处理，这时PostgreSQL也可以使用FDW功能进行对接。

PostgreSQL Sync 模式当前的问题

前面提到当前PostgreSQL在2节点情况下推荐使用的是async的模式，那sync是不是不支持？不是的，当前PostgreSQL支持sync模式，即使2节点也可以配置，但会有以下问题：

- 由于sync同步模式要求Master在Slave数据写入成功后才结束事务的Commit操作，因此性能会受到影响。
- 如果系统运行过程中slave出现故障，主节点也将受到影响使系统出现故障，在HA下这也会Failover。Pacemaker当前最新的PostgreSQLRA也还没有解决此问题。

- 如果需要使用 sync 模式的 Streaming Replication，我建议搭建 1 主 2 备的模型实现，而这个模型下 Pacemaker 还没有提供 3 节点的实现方案，尚待改进。



Q&A

Q1：排除年限，PG 相对 MySQL 和 Oracle 有啥优势？

PG 有很多 MySQL 没有的功能，就以 O2O 行业为例，PG 直接提供 PostGIS，可以有效地在数据库中通过 SQL 进行复杂的定位查询并与业务直接关联，更多功能欢迎线下交流。

Q2：金融企业中用 async 复制，怎么应对数据丢失？靠对账么？

首先，金融行业中如果要求 100% 数据不丢失，应该使用 sync 而不是 async，这个功能 PostgreSQL 是支持的，只是要用 3 节点方案。当前在这个方案下进行 HA 切换也是可以的，只是 Corosync+Pacemaker 没有直接支持，需要我们对 RA 进行附加的脚本控制。

Q3：MySQL 的 binlog 处理和 PG 的有什么区别？

MySQL 我只用过 4 及以下版本，对 binlog 不是十分了解，与公司 MySQL 大牛讨论中，我感觉这两个方式是很接近的，都是使用日

志进行恢复。但 PostgreSQL 的操作中会以 Tuple 为单位，这个可能是一个 row，甚至就是某个 row 中被修改过的 1 个字段的值。有一个讨论结果是 PG 的 StreamingReplication 粒库更细。

Q4：PG 的 HA 除了今天分享的还有其他方案吗？

PG 的 HA 还可以用 LiveKeeper、微软的 MSCS 等方案，对于 HA 来讲 PG 就只是一个服务，所以任何 HA 软件都可以与 PG 对接，但如果要进行 StreamingReplication 的切换就要自己写脚本了。

Q5：我感觉 PG 这个 HA 相比 MySQL 自带的主从没多大亮点。PG 有类似 mysql-proxy 这样的负载均衡中间件及 MMA 这类解决方案吗？之前听说 PG 在集群这块不是更好，方案复杂且性能损失大，是不是 PG 还是更适合单机？

当前 PG 业界可以通过 PGPool 实现 1 个 Master 进行读写，n 个 Slave 进行只读负载均衡的方案；PG 分布式集群当前方案的 Postgres-X2，确实复杂，我们这方面也正在努力；单机的这个问题上，我们有很多 PG 用户会选择通过应用程序自定义进行分库集群模型，毕竟在要求强一致性又没有 InfiniBand 或更高级的网络的情况下，事务和延迟都不好在传统集群中解决。

Q6：PG 有 sharding 功能吗？能否简单介绍一下。

最近 PG 出现了一个名为 pg_shard 的第三方组件，可以对数据库中的一个特定的表进行 sharding，可以扩展到 64 台服务器，但不保证此表的事务，一些分析场景可以考虑尝试。

Q7：能否简单对比一下PG的HA方案？

Corosync + Pacemaker，支持 Replication 模式，在 Linux 下这是我个为最推荐的方案，共享存储同样也没有问题；原 RHEL 中的 RHCS，配置简单，如果有共享存储，Linux 下这个方案最方便，但要注意 RHCS 是要求付费使用的；LiveKeeper，配置相对复杂一些，如果要支持 Replication 需要写比较复杂的脚本；微软 MSCS，Windows 平台必备，中国还真有几个用户是这样用的；VCS，跨 Windows 及 Linux 平台但同样只建议在有共享存储情况下使用。

Q8：PG 的表分区和 MySQL 的表分区差别在哪？各自的优点在哪？我印象中 PG 分区表对外会显示成单独的一个分区表，分区多了很难看。

PG 默认的表分区基于对象数据库结构的表集成，通过触发器进行数据调度，表大了以后性能很差，据说在 9.6 以后（当前 9.4）会改善。在此打个广告，分区表性能在阿里云的 PPAS 中已经得到解决，2~1000 个表分区性能表现恒定，不会因为表分区越来越多导致性能瓶颈。

不一样的数据库

单表60亿记录等大数据场景的 MySQL 优化与运维之道

作者 / 杨尚刚

美图公司数据库高级DBA，
负责美图后端数据存储平台
建设和架构设计。前新浪高
级数据库工程师，负责新浪
微博核心数据库架构改造优
化，以及数据库相关的服务
器存储选型设计。

前言

MySQL数据库大家应该都很熟悉，而且随着前几年的阿里的去IOE，MySQL逐渐引起更多人的重视。

MySQL历史

- 1979年，Monty Widenius写了最初的版本，96年发布1.0
- 1995-2000年，MySQL AB成立，引入BDB
- 2000年4月，集成MyISAM和replication
- 2001年，Heikki Tuuri向MySQL建议集成InnoDB
- 2003发布5.0，提供了视图、存储过程等功能
- 2008年，MySQL AB被Sun收购，09年推出5.1
- 2009年4月，Oracle收购Sun，2010年12月推出5.5
- 2013年2月推出5.6 GA，5.7开发中

MySQL的优点

- 使用简单
- 开源免费

- 扩展性“好”，在一定阶段扩展性好
- 社区活跃
- 性能可以满足互联网存储和性能需求，离不开硬件支持

上面这几个因素也是大多数公司选择考虑 MySQL 的原因。不过 MySQL 本身存在的问题和限制也很多，有些问题点也经常被其他数据库吐槽或鄙视

MySQL 存在的问题

- 优化器对复杂 SQL 支持不好
- 对 SQL 标准支持不好
- 大规模集群方案不成熟，主要指中间件
- ID 生成器，全局自增 ID
- 异步逻辑复制，数据安全性问题
- Online DDL
- HA 方案不完善
- 备份和恢复方案还是比较复杂，需要依赖外部组件
- 展现给用户信息过少，排查问题困难
- 众多分支，让人难以选择

看到了刚才讲的 MySQL 的优势和劣势，可以看到 MySQL 面临的问题还是远大于它的优势的，很多问题也是我们实际需要在运维中优化解决的，这也是 MySQL DBA 的一方面价值所在。并且 MySQL 的不断发展也离不开社区支持，比如 Google 最早提交的半同步 patch，后来也合并到官方主线。Facebook Twitter 等

也都开源了内部使用 MySQL 分支版本，包含了他们内部使用的 patch 和特性。

数据库开发规范

数据库开发规范定义：开发规范是针对内部开发的一系列建议或规则，由 DBA 制定（如果有 DBA 的话）。

开发规范本身也包含几部分：基本命名和约束规范，字段设计规范，索引规范，使用规范。

规范存在意义

- 保证线上数据库 schema 规范
- 减少出问题概率
- 方便自动化管理
- 规范需要长期坚持，对开发和 DBA 是一个双赢的事情

想想没有开发规范，有的开发写出各种全表扫描的 SQL 语句或者各种奇葩 SQL 语句，我们之前就看过开发写的 SQL 可以打印出好几张纸。这种造成业务本身不稳定，也会让 DBA 天天忙于各种救火。

基本命名和约束规范

- 表字符集选择 UTF8，如果需要存储 emoji 表情，需要使用 UTF8mb4 (MySQL 5.5.3 以后支持)
- 存储引擎使用 InnoDB
- 变长字符串尽量使用 varchar varbinary

- 不在数据库中存储图片、文件等
- 单表数据量控制在1亿以下
- 库名、表名、字段名不使用保留字
- 库名、表名、字段名、索引名使用小写字母，以下划线分割，需要见名知意
- 库表名不要设计过长，尽可能用最少的字符表达出表的用途

字段规范

- 所有字段均定义为 NOT NULL，除非你真的想存 Null
- 字段类型在满足需求条件下越小越好，使用 UNSIGNED 存储非负整数，实际使用时候存储负数场景不多
- 使用 TIMESTAMP 存储时间
- 使用 varchar 存储变长字符串，当然要注意 varchar(M) 里的 M 指的是字符数不是字节数；使
- UNSIGNED INT 存储 IPv4 地址而不是 CHAR(15)，这种方式只能存储 IPv4，存储不了 IPv6
- 使用 DECIMAL 存储精确浮点数，用 float 有的时候会有问题
- 少用 blob text

关于为什么定义不使用 Null 的原因：

1. 浪费存储空间，因为 InnoDB 需要有额外一个字节存储
2. 表内默认值 Null 过多会影响优化器选择执行计划

关于使用 datetime 和 timestamp，现在在 5.6.4 之后又有了

变化，使用二者存储在存储空间上大差距越来越小，并且本身 datatime 存储范围就比 timestamp 大很多，timestamp 只能存储到 2038 年

Data Type	Storage Required Before MySQL 5.6.4	Storage Required as of MySQL 5.6.4
<u>YEAR</u>	1 byte	1 byte
<u>DATE</u>	3 bytes	3 bytes
<u>TIME</u>	3 bytes	3 bytes + fractional seconds storage
<u>DATETIME</u>	8 bytes	5 bytes + fractional seconds storage
<u>TIMESTAMP</u>	4 bytes	4 bytes + fractional seconds storage

索引规范

- 单个索引字段数不超过 5，单表索引数量不超过 5，索引设计遵循 B+ Tree 索引最左前缀匹配原则
- 选择区分度高的列作为索引
- 建立的索引能覆盖 80% 主要的查询，不求全，解决问题的主要矛盾
- DML 和 order by 和 group by 字段要建立合适的索引
- 避免索引的隐式转换
- 避免冗余索引

关于索引规范，一定要记住索引这个东西是一把双刃剑，在加速读的同时也引入了很多额外的写入和锁，降低写入能力，这也是为什么要控制索引数原因。之前看到过不少人给表里每个字段都建了索引，其实对查询可能起不到什么作用。

冗余索引例子：

- idx_abc(a,b,c)
- idx_a(a) 冗余
- idx_ab(a,b) 冗余

隐式转换例子：

```
字段:remark varchar(50) NOT Null
MySQL>SELECT id, gift_code FROM gift WHERE deal_id = 640 AND remark=115127;
1 row in set (0.14 sec)
MySQL>SELECT id, gift_code FROM pool_gift WHERE deal_id = 640 AND
remark='115127'; 1 row in set (0.005 sec)
```

字段定义为 varchar，但传入的值是个 int，就会导致全表扫描，要求程序端要做好类型检查。

SQL类规范

- 尽量不使用存储过程、触发器、函数等
- 避免使用大表的 JOIN， MySQL优化器对join优化策略过于简单
- 避免在数据库中进行数学运算和其他大量计算任务
- SQL合并，主要是指的DML时候多个value合并，减少和数据库交互

- 合理的分页，尤其大分页
- UPDATE、DELETE语句不使用LIMIT，容易造成主从不一致

数据库运维规范

运维规范主要内容

- SQL审核，DDL审核和操作时间，尤其是OnlineDDL
- 高危操作检查，Drop前做好数据备份
- 权限控制和审计
- 日志分析，主要是指的MySQL慢日志和错误日志
- 高可用方案
- 数据备份方案

版本选择

- MySQL社区版，用户群体最大
- MySQL企业版，收费
- Percona Server版，新特性多
- MariaDB版，国内用户不多

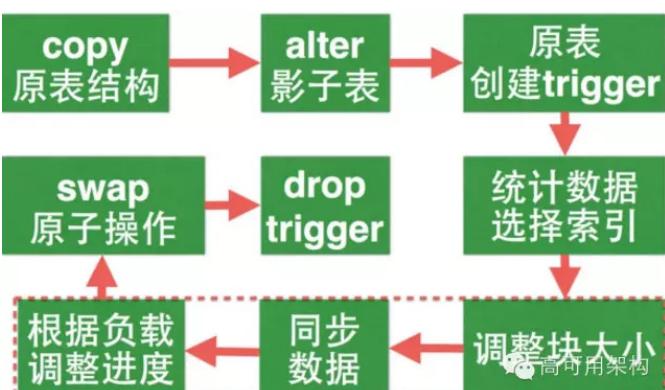
建议选择优先级为：MySQL社区版 > Percona Server > MariaDB > MySQL企业版。不过现在如果大家使用RDS服务，基本还以社区版为主。

Online DDL问题

原生MySQL执行DDL时需要锁表，且锁表期间业务是无法写入数据的，对服务影响很大，MySQL对这方面的支持是比较差的。大表做DDL对DBA来说是很痛苦的，相信很多人经历过。如何做到Online DDL呢，是不是就无解了呢？当然不是！

	Facebook OSC	5.6 OSC	Oracle
实现原理	触发器+change log	内部记录change log	存储格式支持
实现效果	不锁表		
优点	通用，支持多种DDL在线操作		快速，副作用少
缺点	性能有一定损失，加字段时间长 (CPU/IO负载增加20%， 4.8G+400insert/s)		不够通用 一定技术风险(存储格式支持)

上面表格里提到的 Facebook OSC 和 5.6 OSC 也是目前两种比较靠谱的方案。MySQL 5.6 的 OSC 方案还是解决不了 DDL 的时候到从库延时的问题，所以现在建议使用 Facebook OSC 这种思路更优雅。下图是 Facebook OSC 的思路：



后来 Percona 公司根据 Facebook OSC 思路，用 perl 重写了一版，就是我们现在用得很多的 pt-online-schema-change，软件本身非常成熟，支持目前主流版本。

使用 pt-online-schema-change 的优点有：

1. 无阻塞写入
2. 完善的条件检测和延时负载策略控制

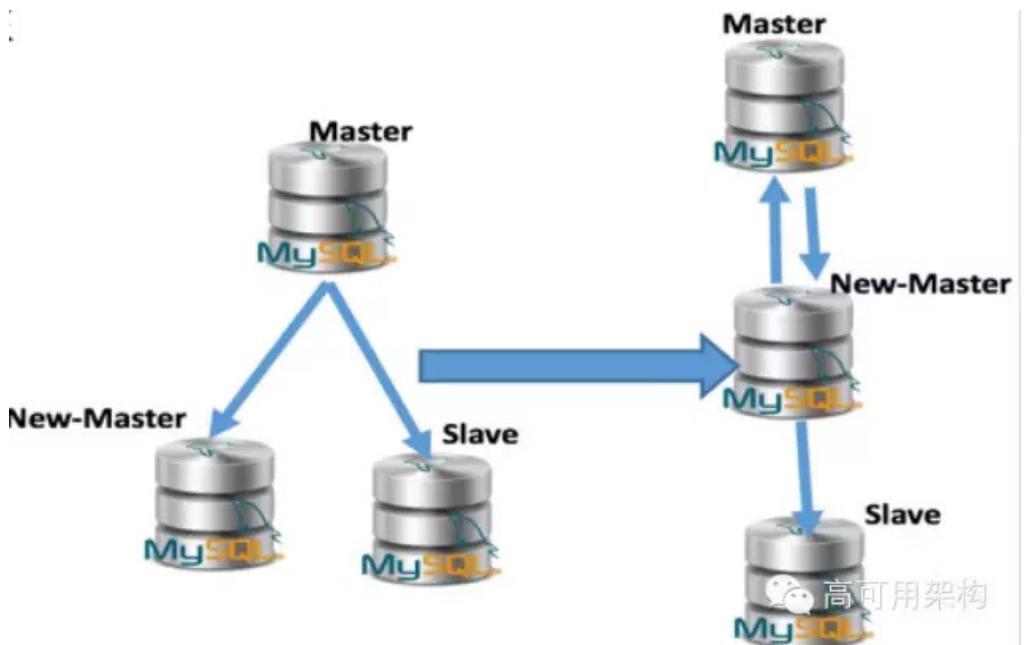
值得一提的是，腾讯互娱的 DBA 在内部分支上也实现了 Online DDL，之前测试过确实不错，速度快，原理是通过修改 InnoDB 存储格式来实现。

使用 pt-online-schema-change 的限制有：

- 改表时间会比较长（相比直接 alter table 改表）
- 修改的表需要有唯一键或主键
- 在同一端口上的并发修改不能太多

可用性

关于可用性，我们今天分享一种无缝切主库方案，可以用于日常切换，使用思路也比较简单。在正常条件下如何无缝去做主库切换，核心思路是让新主库和从库停在相同位置，主要依赖 slave start until 语句，结合双主结构，考虑自增问题。

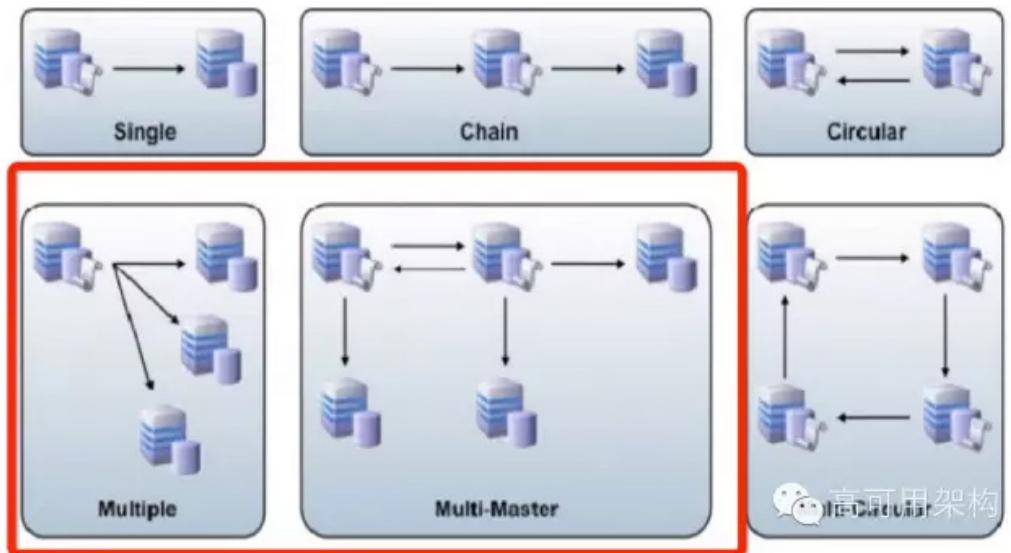


MySQL 集群方案：

- 集群方案主要是如何组织 MySQL 实例的方案
- 主流方案核心依然采用的是 MySQL 原生的复制方案
- 原生主从肯定存在着性能和安全性问题

MySQL 半同步复制（现在也有一些理论上可用性更高的其它方案）：

- Percona XtraDB Cluster(没有足够的把控力度，不建议上)
- MySQL Cluster(有官方支持，不过实际用的不多)

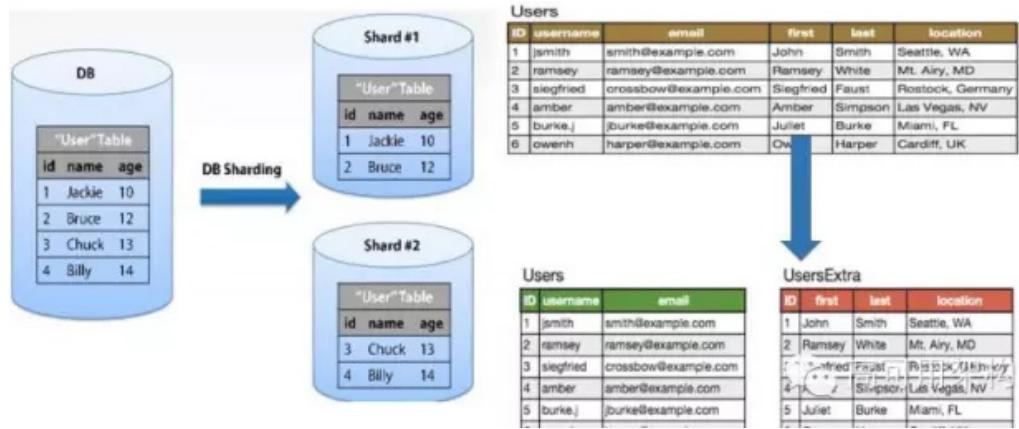


红框内是目前大家使用比较多的部署结构和方案。当然异常层面的 HA 也有很多第三方工具支持，比如 MHA、MMM 等，推荐使用 MHA。

sharding 拆分问题

- Sharding is very complex, so it's best not to shard until it's obvious that you will actually need to!
- sharding 是按照一定规则数据重新分布的方式
- 主要解决单机写入压力过大和容量问题
- 主要有垂直拆分和水平拆分两种方式
- 拆分要适度，切勿过度拆分
- 有中间层控制拆分逻辑最好，否则拆分过细管理成本会很高

曾经管理的单表最大 60 亿 +，单表数据文件大小 1TB +，人有时候就要懒一些。



上图是水平拆分和垂直拆分的示意图。

数据库备份

首先要保证的，最核心的是数据库数据安全性。数据安全都保障不了的情况下谈其他的指标(如性能等)，其实意义就不大了。

备份的意义是什么呢？

- 数据恢复！
- 数据恢复！
- 数据恢复！

目前备份方式的几个纬度：

- 全量备份 VS 增量备份
- 热备 VS 冷备
- 物理备份 VS 逻辑备份

- 延时备份
- 全量 binlog 备份

建议方式：

- 热备 + 物理备份
- 核心业务：延时备份 + 逻辑备份
- 全量 binlog 备份

借用一下某大型互联网公司做的备份系统数据：一年 7000 + 次扩容，一年 12 + 次数据恢复，日志量每天 3TB，数据总量 2PB，每天备份数据量百 TB 级，全年备份 36 万次，备份成功了 99.9%。

主要做的几点：

- 备份策略集中式调度管理
- xtrabackup 热备
- 备份结果统计分析
- 备份数据一致性校验
- 采用分布式文件系统存储备份

备份系统采用分布式文件系统原因：

- 解决存储分配的问题
- 解决存储 NFS 备份效率低下问题
- 存储集中式管理

- 数据可靠性更好

使用分布式文件系统优化点：

- Pbzip 压缩，降低多副本存储带来的存储成本，降低网络带宽消耗
- 元数据节点 HA，提高备份集群的可用性
- erasure code 方案调研

数据恢复方案

目前的 MySQL 数据恢复方案主要还是基于备份来恢复，可见备份的重要性。比如我今天下午 15 点删除了线上一张表，该如何恢复呢？首先确认删除语句，然后用备份扩容实例启动，假设备份时间点是凌晨 3 点，就还需要把凌晨 3 点到现在关于这个表的 binlog 导出来，然后应用到新扩容的实例上，确认好恢复的时间点，然后把删除表的数据导出来应用到线上。

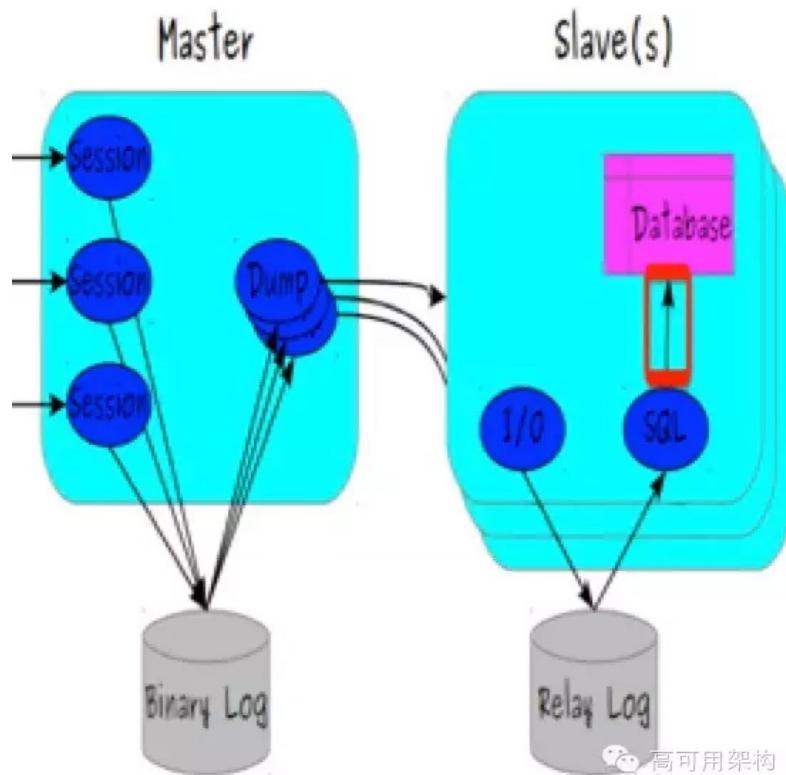
性能优化

复制优化

MySQL 复制：

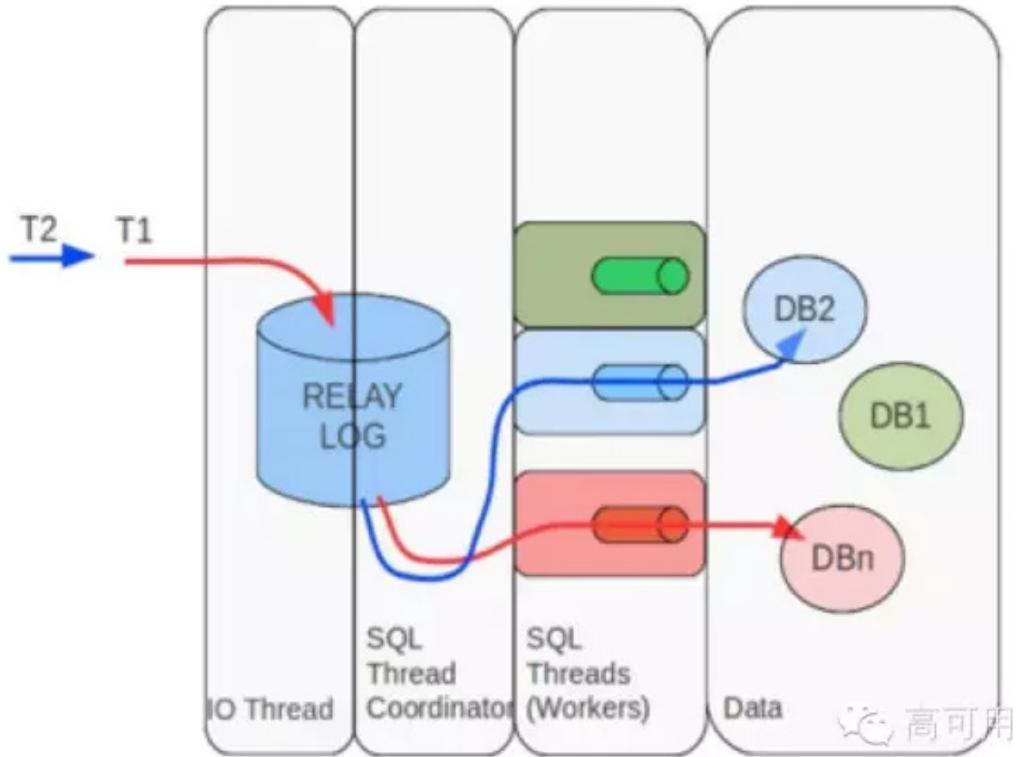
- 是 MySQL 应用得最普遍的应用技术，扩展成本低
- 逻辑复制
- 单线程问题，从库延时问题
- 可以做备份或读复制

问题很多，但是能解决基本问题。



上图是 MySQL 复制原理图，红框内就是 MySQL 一直被人诟病的单线程问题。单线程问题也是 MySQL 主从延时的一个重要原因，单线程解决方案：

- 官方 5.6+ 多线程方案
- Tungsten 为代表的第三方并行复制工具
- sharding

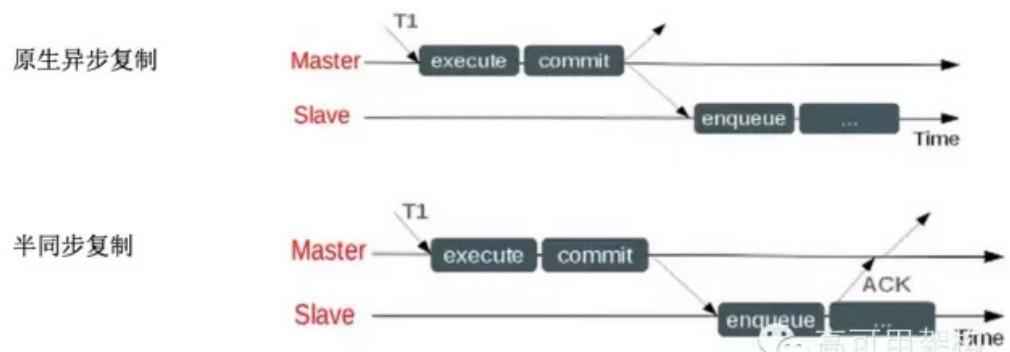


上图是MySQL5.6 目前实现的并行复制原理图，是基于库级别的复制，所以如果你只有一个库，使用这个意义不大。

当然MySQL也认识到5.6这种并行的瓶颈所在，所以在5.7引入了另外一种并行复制方式，基于logical timestamp的并行复制，平行复制不再受限于库的个数，效率会大大提升。

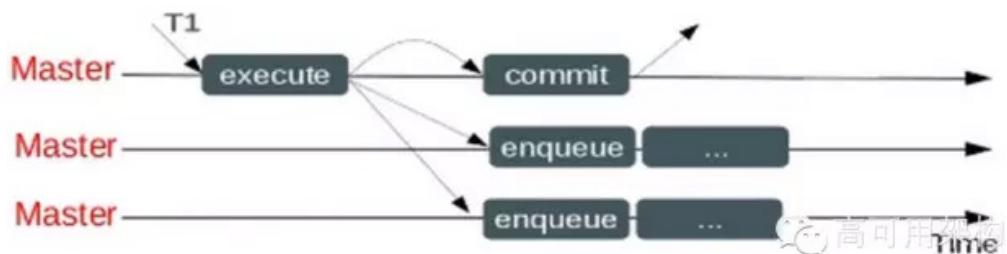


上图是 5.7 的 logical timestamp 的复制原理图。刚才我也提到 MySQL 原来只支持异步复制，这种数据安全性是非常差的，所以后来引入了半同步复制，从 5.5 开始支持。



上图是原生异步复制和半同步复制的区别。可以看到半同步通过从库返回 ACK 这种方式确认从库收到数据，数据安全性大大提高。

在 5.7 之后，半同步也可以配置你指定多个从库参与半同步复制，之前版本都是默认一个从库。对于半同步复制效率问题有一个小的优化，就是使用 5.6+ 的 mysqlbinlog 以 daemon 方式作为从库，同步效率会好很多。关于更安全的复制，MySQL 5.7 也是有方案的，方案名叫 Group replication 官方多主方案，基于 Corosync 实现。



主从延时问题

原因：一般都会做读写分离，其实从库压力反而比主库大／从库读写压力大非常容易导致延时。

解决方案：

- 首先定位延时瓶颈
- 如果是 IO 压力，可以通过升级硬件解决，比如替换 SSD 等
- 如果 IO 和 CPU 都不是瓶颈，非常有可能是 SQL 单线程问题，解决方案可以考虑刚才提到的并行复制方案
- 如果还有问题，可以考虑 sharding 拆分方案

提到延时不得不提到很坑人的 Seconds behind master，使用过 MySQL 的应该很熟悉。

这个值的源码里算法：

```
long time_diff= ((long)(time(0) - mi->rli.last_master_timestamp) - mi->clock_diff_with_master);
```

Secondsbehindmaster 来判断延时不可靠，在网络抖动或者一些特殊参数配置情况下，会造成这个值是 0 但其实延时很大了。通过 heartbeat 表插入时间戳这种机制判断延时是更靠谱的。

复制注意点：

- Binlog 格式，建议都采用 row 格式，数据一致性更好
- Replication filter 应用

主从数据一致性问题：

- row 格式下的数据恢复问题

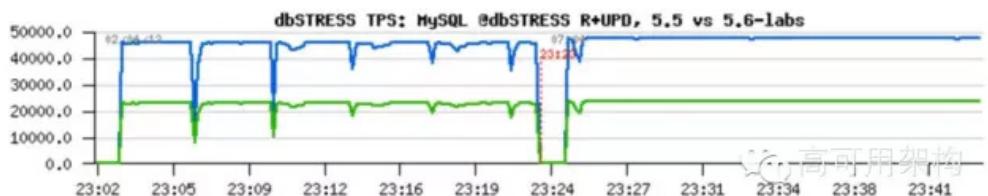
InnoDB 优化

成熟开源事务存储引擎，支持 ACID，支持事务四个隔离级别，更好的数据安全性，高性能高并发，MVCC，细粒度锁，支持 O_DIRECT。

主要优化参数：

- innodbfileper_table =1
- innodbbufferpool_size，根据数据量和内存合理设置

- innodbflushlog_attrxcommit= 0 1 2
- innodblogfile_size, 可以设置大一些
- innodbpagesize
- Innodbflushmethod = o_direct
- innodbundodirectory 放到高速设备 (5.6 +)
- innodbbufferpool_dump
- atshutdown , bufferpool dump (5.6+)



上图是 5.5 4G 的 redo log 和 5.6 设置大于 4G redo log 文件性能对比，可以看到稳定性更好了。innodblogfile_size 设置还是很有意义的。

InnoDB 比较好的特性：

- Bufferpool 预热和动态调整大小，动态调整大小需要 5.7 支持
- Page size 自定义调整，适应目前硬件
- InnoDB 压缩，大大降低数据容量，一般可以压缩 50%，节省存储空间和 IO，用 CPU 换空间
- Transportable tablespaces，迁移 ibd 文件，用于快速单表恢复
- Memcached API, full text, GIS 等

InnoDB在SSD上的优化:

- 在5.5以上，提高innodbwriteiothreads和innodbreadiothreads
- innodbiocapacity需要调大
- 日志文件和redo放到机械硬盘，undo放到SSD，建议这样，但必要性不大
- atomic write,不需要Double Write Buffer
- InnoDB压缩
- 单机多实例

也要搞清楚InnoDB哪些文件是顺序读写，哪些是随机读写。

随机读写:

- datadir
- innodbdata file_path
- innodbundo directory

顺序读写:

- innodbloggroupomedir
- log-bin

InnoDB VS MyISAM:

- 数据安全性至关重要，InnoDB完胜，曾经遇到过一次90G的MyISAM表repair，花了两天时间，如果在线上几乎不可忍受
- 并发度高

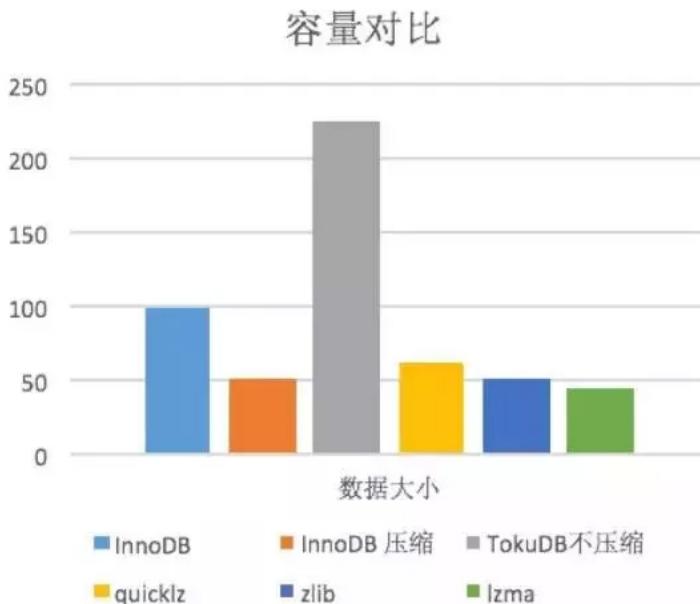
- MySQL 5.5默认引擎改为InnoDB，标志着MyISAM时代的落幕

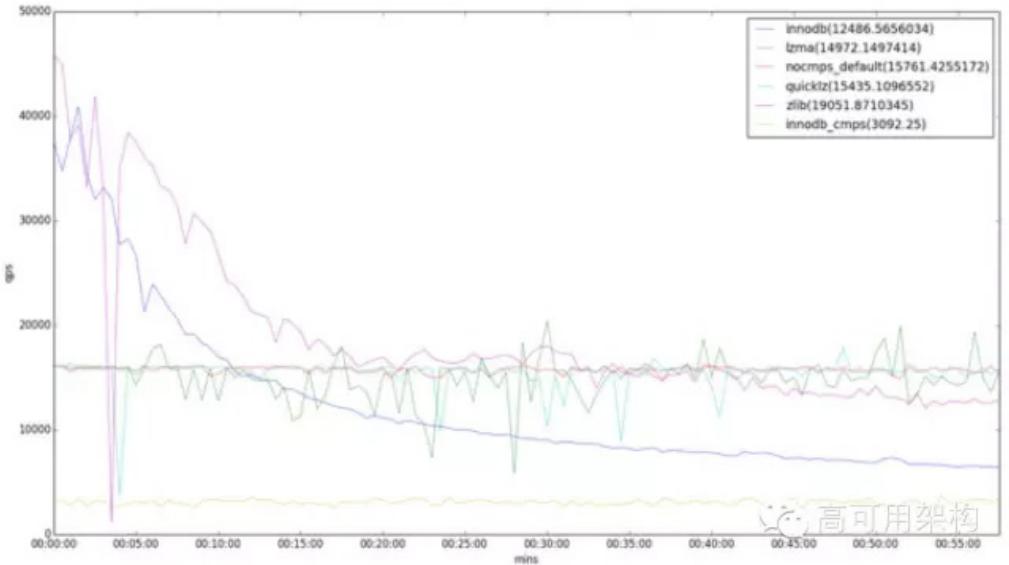
TokuDB:

- 支持事务 ACID 特性，支持多版本控制 (MVCC)
- 基于Fractal Tree Index，非常适合写入密集场景
- 高压缩比，原生支持Online DDL
- 主流分支都支持，收费转开源。目前可以和InnoDB媲美的存储引擎

目前主流使用TokuDB主要是看中了它的高压缩比，Tokudb有三种压缩方式：quicklz、 zlib、 lzma，压缩比依次更高。现在很多使用zabbix的后端数据表都采用的TokuDB，写入性能好，压缩比高。

下图是我之前做的测试对比和InnoDB：





上图是 sysbench 测试的和 InnoDB 性能对比图，可以看到 TokuDB 在测试过程中写入稳定性是非常好的。

tokudb 存在的问题：

- 官方分支还没很好的支持
- 热备方案问题，目前只有企业版才有
- 还是有 bug 的，版本更新比较快，不建议在核心业务上用

比如我们之前遇到过一个问题：TokuDB 的内部状态显示上一次完成的 checkpoint 时间是 “Jul 17 12:04:11 2014”，距离当时发现现在都快 5 个月了，结果堆积了大量的 redo log 不能删除，后来只能重启实例，结果重启还花了七八个小时

MySQL 优化相关的 case

Query cache, MySQL 内置的查询加速缓存，理念是好的，但设计不够合理，有点 out。锁的粒度非常大 MySQL 5.6 默认已经关闭。When the query cache helps, it can help a lot. When it hurts, it can hurt a lot. 明显前半句已经没有太大用处，在高并发下非常容易遇到瓶颈。

关于事务隔离级别，InnoDB 默认隔离级别是可重复读级别，当然 InnoDB 虽然是设置的可重复读，但是也是解决了幻读的，建议改成读已提交级别，可以满足大多数场景需求，有利于更高的并发，修改 transaction-isolation。

隔离级别	脏读	不可重复读	幻读
读未提交 (Read uncommitted)	✓	✓	✓
读已提交 (Read committed)	✗	✓	✓
可重复读 (Repeatable read)	✗	✗	✓
可串行化 (Serializable)	✗	✗	✗

	Session1	Session2	
1			
2	insert into t3 (id1) values(10);	insert into t3 (id1) values(10);	
3	ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction	insert into t3 (id1) values(9);	

```

测试版本5.5.31
CREATE TABLE `t3` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `id1` int(11) NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `id1` (`id1`)
) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=utf8

#高可用架构 REPEATABLE-READ
#锁信息
140714 11:21:28
*** (1) TRANSACTION:
TRANSACTION F08, ACTIVE 7 sec inserting
mysql tables in use 1, locked 1
LOCK WAIT 2 lock struct(s), heap size 376, 1 row lock(s), undo log entries 1
MySQL thread id 3, OS thread handle 0x4176f940, query id 76 localhost
myadmin update
insert into t3 (id1) values(10)
*** (1) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 8 page no 4 n bits 72 index 'id1' of table 'test'.`t3` trx
id F08 lock mode S waiting
Record lock, heap no 2 PHYSICAL RECORD: n_fields 2; compact format; info
bits 0
0: len 4; hex 8000000a; asc   ;;
1: len 4; hex 00000001; asc   ;;

*** (2) TRANSACTION:
TRANSACTION F07, ACTIVE 17 sec inserting
mysql tables in use 1, locked 1
3 lock struct(s), heap size 376, 2 row lock(s), undo log entries 2
MySQL thread id 2, OS thread handle 0x418e8940, query id 77 localhost
myadmin update
insert into t3 (id1) values(9)
*** (2) HOLDS THE LOCK(S):
RECORD LOCKS space id 8 page no 4 n bits 72 index 'id1' of table 'test'.`t3` trx
id F07 lock mode X locks rec but not gap
Record lock, heap no 2 PHYSICAL RECORD: n_fields 2; compact format; info
bits 0
0: len 4; hex 8000000a; asc   ;;
1: len 4; hex 00000001; asc   ;;

*** (2) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 8 page no 4 n bits 72 index 'id1' of table 'test'.`t3` trx
id F07 lock mode X locks gap before rec insert intention waiting
Record lock, heap no 2 PHYSICAL RECORD: n_fields 2; compact format; info
bits 0
0: len 4; hex 8000000a; asc   ;;
1: len 4; hex 00000001; asc   ;;

*** WE ROLL BACK TRANSACTION (1)

```



上图是一个比较经典的死锁 case，有兴趣可以测试下。

关于 SSD

关于 SSD，还是提一下吧。某知名大 V 说过“最近 10 年对数据库性能影响最大的是闪存”，稳定性和性能可靠性已经得到大规模验证，多块 SATA SSD 做 Raid5，推荐使用。采用 PCIe SSD，主流云平台都提供 SSD 云硬盘支持。

最后说一下大家关注的单表 60 亿记录问题，表里数据也是线上比较核心的。先说下当时情况，表结构比较简单，都是 bigint 这种整型，索引比较多，应该有 2-3 个，单表行数 60 亿 +，单表容量 1.2TB 左右，当然内部肯定是有碎片的。

形成原因：历史遗留问题，按照我们前面讲的开发规范，这个应该早拆分了，当然不拆有几个原因：

- 性能未遇到瓶颈，主要原因
- DBA 比较“懒”
- 想看看 InnoDB 的极限，挑战一下。不过风险也是很大的，想想如果在一个 1.2TB 表上加个字段加个索引，那感觉绝对酸爽。还有就是单表恢复的问题，恢复时间不可控。

我们后续做的优化，采用了刚才提到的 TokuDB，单表容量在 InnoDB 下 1TB+，使用 Tokudb 的 lzma 压缩到 80GB，压缩效果非常好。这样也解决了单表过大恢复时间问题，也支持 online DDL，基本达到我们预期。■

Q&A

Q1: `use schema;select * from table;` 和 `select * from schema.table;` 两种写法有什么不一样吗？会对主从同步有影响吗？

对于主从复制来说执行效率上差别不大，不过在使用replication filter时候这种情况需要小心，应该要使用ReplicateWildIgnoreTable这种参数，如果不使用带wildignore，第一种方式会有问题，过滤不起作用。

Q2：数据库规范已制定好，如何保证开发人员必须按照规范来开发？

关于数据库规范实施问题，也是有两个方面吧，第一、定期给开发培训开发规范，让开发能更了解。第二、还是在流程上规范，比如把我们日常通用的建表和字段策略固化到程序，做成自动化审核系统。这两方面结合 效果会比较好。

Q3：如何最大限度提高innodb的命中率？

这个问题前提是你的数据要有热点，读写热点要有交集，否则命中率很难提高。在有热点的前提下，也要求你的内存要足够大，能够存更多的热点数据。尽量不要做一些可能污染bufferpool的操作，比如全表扫描这种。

Q4：对超大表水平拆分，在使用MySQL中间件方面有什么建议和经验分享？

对于超大表水平拆分，在中间件上经验不是很多，早期人肉搞过几次。也使用过自己研发的数据库中间件，不过线上应用的规模不大。关于目前众多的开源中间件里，360的atlas是目前还不错的，他们公司内部应用的比较多。

Q5：我想问一下关于mysql线程池相关的问题，什么情况下适合使用线程池，相关的参数应该如何配置，老师有这方面的最佳实践没有？

线程池这个我也没测试过。从原理上来说，短链接更适合用线程池方式，减少建立连接的消耗。这个方面的最佳配置，我还没测试过，后面测试有进展可以再聊聊。

Q6：误删数据这种，数据恢复流程是怎么样的（从库也被同步删除的情况）？

看你删除数据的情况，如果只是一张表，单表在几GB或几十GB。如果能有延时备份，对于数据恢复速度是很有好处的。恢复流程可以参考我刚才分享的部分。目前的MySQL数据恢复方案主要还是基于备份来恢复，可见备份的重要性。比如我今天下午15点删除了线上一张表，该如何恢复呢。首先确认删除语句，然后用备份扩容实例启动，假设备份时间点是凌晨3点。就还需要把凌晨3点到现在关于这个表的binlog导出来，然后应用到新扩容的实例上。确认好恢复的时间点，然后把删除表的数据导出来应用到线上。

Q7：关于备份，binlog备份自然不用说了，物理备份有很多方式，有没有推荐的一种，逻辑备份在量大的时候恢复速度比较慢，一般用在什么场景？

物理备份采用xtrabackup热备方案比较好。逻辑备份一般用在单表恢复效果会非常好。比如你删了一个2G表，但你总数据量2T，用物理备份就会要慢了，逻辑备份就非常有用。

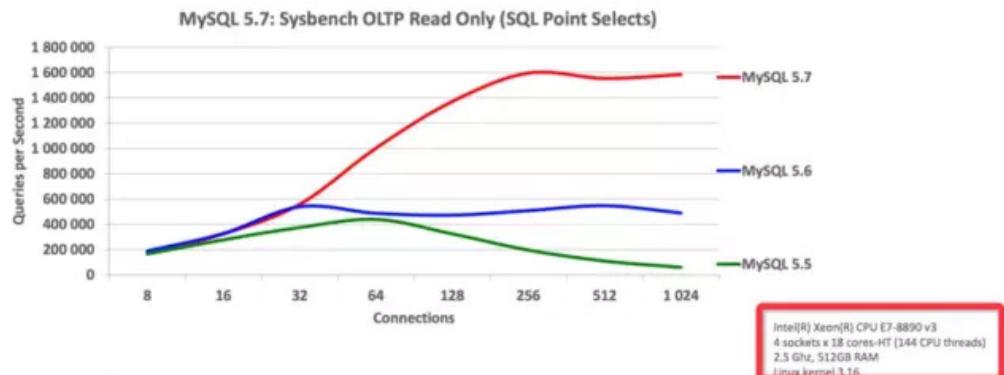
不一样的数据库

MySQL 5.7 新特性大全和未来展望

作者 / 杨尚刚

美图公司数据库高级DBA，负责美图后端数据存储平台建设和架构设计。前新浪微博核心数据库工程师，负责新浪微博核心数据库架构改造优化，以及数据库相关的服务器存储选型设计。之前在「高可用架构」发表的《单表60亿记录等大数据场景的MySQL 优化和运维之道》广受好评。

2015 年最重磅的当属 MySQL 5.7 GA 的发布，号称 160 万只读 QPS，大有赶超 NoSQL 趋势。



上面这个图是 Oracle 在只读场景下官方测试的结果，看上去 QPS 确实提升很大。不过官方的硬件测试环境是很高的，所以这个 160 万 QPS 对于大家测试来说，可能还比较遥远，所以实际测试的结果可能会失望。但是，至少我们看到了基于同样测试环境，MySQL 5.7 在性能上的改进，对于多核利用的改善。

提高运维效率的特性

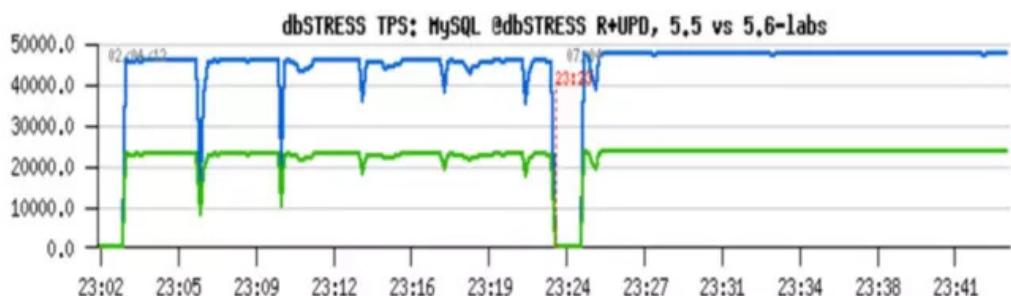
MySQL 5.7 动态修改 Buffer Pool

从 MySQL 5.7.5 开始可以在线动态调整，对运维更友好。很多人都经历过 Buffer Pool 过大或过小调整需要重启实例，运维成本非常高，尤其是主库或其他核心业务。

MySQL redo log 大小

5.5 <= 4G, 5.6 +<= 512G 当然这个也不是越大越好，但是提供了可以尝试的机会，越大的 redo log 理论会有更稳定的性能。当然带来的风险就是故障恢复时间会更长。

下图就是不同 redo 大小的性能对比，主要是看性能抖动情况。



innodb_file_per_table

默认值 Off <= 5.6.5 => On，独立表空间优点很明显。尤其可以使用 InnoDB Transportable tablespaces，可以像 MyISAM 一样快速迁移表。

query cache

$1 \leq 5.6.8 \leq 0$, 默认关闭。整体来说关闭 query cache 是利远大于弊，官方最终也选择了关闭。

SQL_Mode 变为 Strict mode

SQL 要求更严格, $version < 5.6.6$ sql_mode 为空, 最为宽松, 不够严谨。

$5.6.6 < version < 5.7.4$

- NO_ENGINE_SUBSTITUTION

$version > 5.7.9$

- ONLY_FULL_GROUP_BY
- STRICT_TRANS_TABLES
- NO_ZERO_IN_DATE
- NO_ZERO_DATE
- ERROR_FOR_DIVISION_BY_ZERO
- NO_AUTO_CREATE_USER NO_ENGINE_SUBSTITUTION

以 NO_ZERO_DATE 为例, 如果你原表里有 0000-00-00 这种数据, 在 MySQL 5.7 使用默认 SQL_Mode 时候改表时候就会报错了。

binlog_rows_query_log_events

默认关闭, 可选打开, 建议打开, 还是比较有用的。可以看到 row 格式下的 sql 语句, 方便排查问题和恢复数据。

```

# at 1354          end_log_pos 1408 CRC32 0x5e02d40f  Rows_query
#151216 10:24:50 server id          end_log_pos 1408 CRC32 0x5e02d40f  Rows_query
# update t1 set id=10 where id=1
# at 1408          end_log_pos 1453 CRC32 0x3f5eea22  Table_map: `t
#151216 10:24:50 server id 72920000  end_log_pos 1453 CRC32 0x3f5eea22  Table_map: `t
#151216 10:24:50 server id 72920000  end_log_pos 1499 CRC32 0x8f77c8f0  Update_rows:
### UPDATE `test`.`t1`
### WHERE
###   @l=1 /* INT meta=0 nullable=0 is_null=0 */
### SET
###   @l=10 /* INT meta=0 nullable=0 is_null=0 */

```

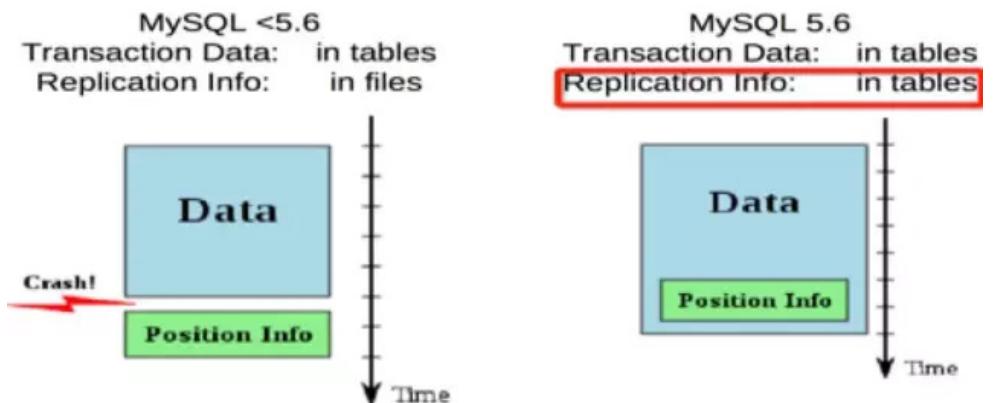
上图就是开启之后 binlog 解析出来的内容，可以看到正常 SQL。

max_execution_time

5.7.4 刚引入名字是 max_statement_time，后来改成 max_execution_time 单位是毫秒，SQL 语句的超时中断，自我保护的一种方案。只针对 select，也可以在 sql 里指定。如果 percona 版本的话，这个参数更暴力，对所有请求生效慎用。

replication info in tables

crash-safe slave 方案，最早 Google 做的方案是存储在事务日志里。单独存储更灵活，可以解决很多从库宕机后 duplicate key 问题。

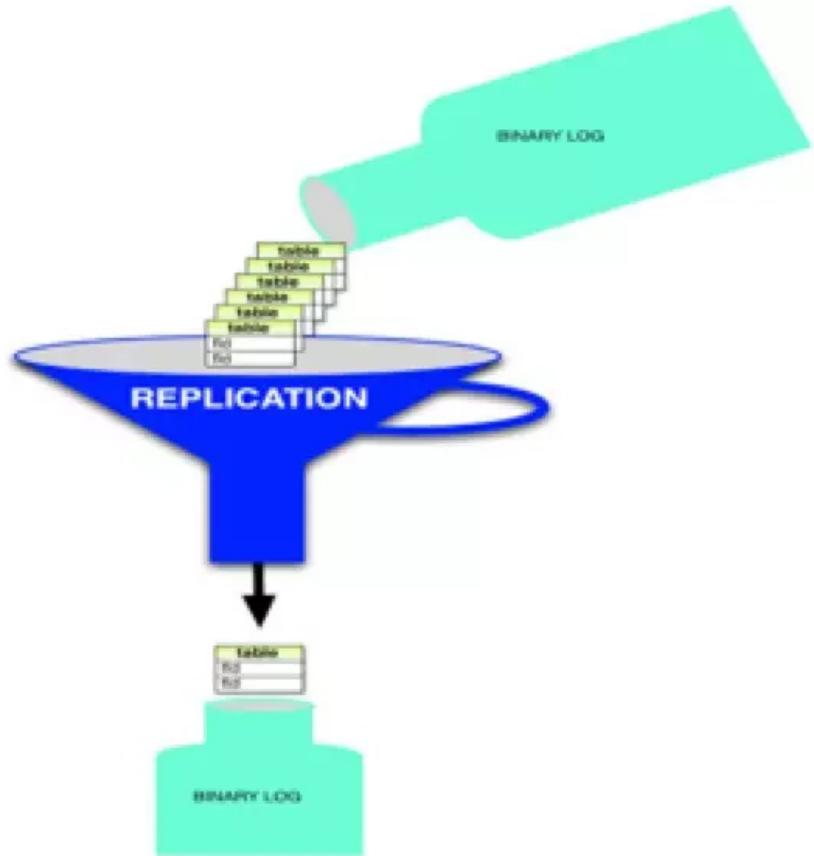


innodb_numa_interleave

建议关掉 numa。最早 Twitter 开源分支里有提供，也可以启动实例时候设置 numactl - interleave all，不过实际线上使用系统默认 numa 策略，并没有遇到过因为 numa 导致的 swap 问题。

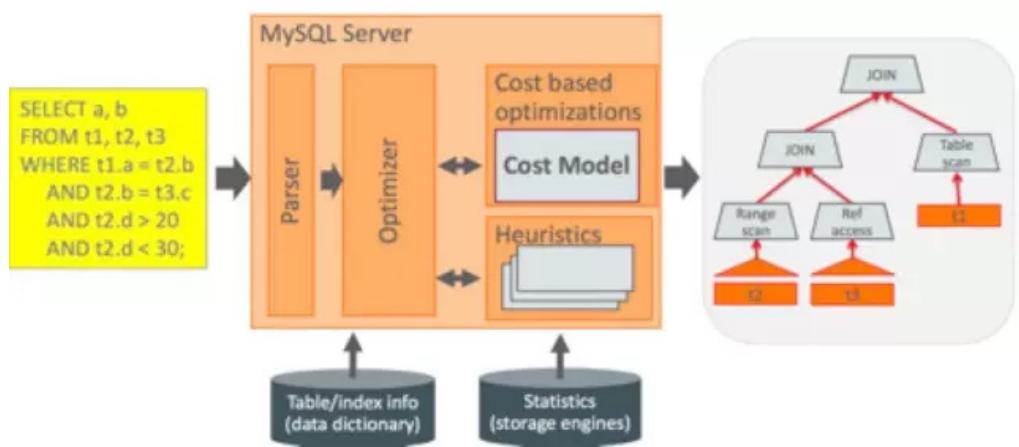
动态修改 replication filter

方便做拆分或做级连复制时候使用，可以通过 change 动态修改。如果用过 replication filter 应该清楚这个还是比较有用的。



优化器 Server 层改进

优化器主要还是基于 cost model 层面和给用户更多自主优化。



可配置 cost based optimizer, `mysql.server_cost` 和 `mysql.engine_cost`。

mysql> select * from server_cost;				
cost_name	cost_value	last_update	comment	
disk_tempetable_create_cost	NULL	2015-09-21 14:39:26	NULL	
disk_tempetable_row_cost	NULL	2015-09-21 14:39:26	NULL	
key_compare_cost	NULL	2015-09-21 14:39:26	NULL	
memory_tempetable_create_cost	NULL	2015-09-21 14:39:26	NULL	
memory_tempetable_row_cost	NULL	2015-09-21 14:39:26	NULL	
row_evaluate_cost	NULL	2015-09-21 14:39:26	NULL	

New JSON 数据类型和函数支持。当然 JSON 也可以存在 Text 或 VARCHAR 里用内置 json，更容易访问，方便修改。

```
CREATE TABLE employees (data JSON);
INSERT INTO employees VALUES ('{"id": 1, "name": "Jane"}');
INSERT INTO employees VALUES ('{"id": 2, "name": "Joe"}');
```

```
SELECT * FROM employees;
+-----+
| data |
+-----+
| {"id": 1, "name": "Jane"} |
| {"id": 2, "name": "Joe"} |
+-----+
2 rows in set (0,00 sec)
```

支持生成列（虚拟列），以及虚拟列上索引。

```
CREATE TABLE order_lines (orderno integer,
lineno integer,
price decimal(10,2),
qty integer,
sum_price decimal(10,2) GENERATED ALWAYS AS (qty * price) STORED );
```

简化查询，有 virtual 和 stored 两种情况，感觉这个功能还是比较小众。

下图是二者对比：

- | | |
|---|---|
| <ul style="list-style-type: none">• STORED<ul style="list-style-type: none">– Primary & secondary index– B-TREE, Full text search, R-TREE– Duplication of data in base table and index– Independent of SE– Online operation | <ul style="list-style-type: none">• VIRTUAL<ul style="list-style-type: none">– Secondary index only– B-TREE only– Less storage– Requires SE support– Online operation |
|---|---|

5.7 还对 explain 做了增强，对于当前正在运行查询 explain。

```
EXPLAIN [FORMAT=(JSON|TRADITIONAL)] FOR  
CONNECTION <id>;
```

InnoDB 层优化

InnoDB 层核心还是拆分各种锁，提高并发。只读事务优化就是其中一个例子不再使用只读事务 list，重构 MVCC 代码，不为只读事务分配事务 id，降低内存开销。

如何使用只读事务：

- start transaction read only
- 开启 autocommit 下的不加锁的 select 语句

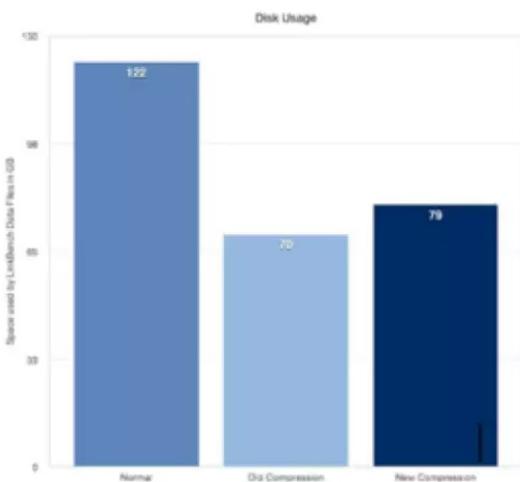
原生支持分区 Native Partitioning，之前版本分区表是放在 server 层管理的，现在是在引擎层面支持，更节省内存，分区越多，效果越明显。

atomic write, disable double write, MySQL 5.7 开始支持 atomic write，成本高，性价比不算高，需要底层存储硬件支持，感觉比较鸡肋。

- 支持 spatial index 空间索引
- 基于 R tree 实现
- 目前只支持 2D 数据类型
- 支持 GeoHash 和 GeoJson，提高数据查找效率

Transparent page compression

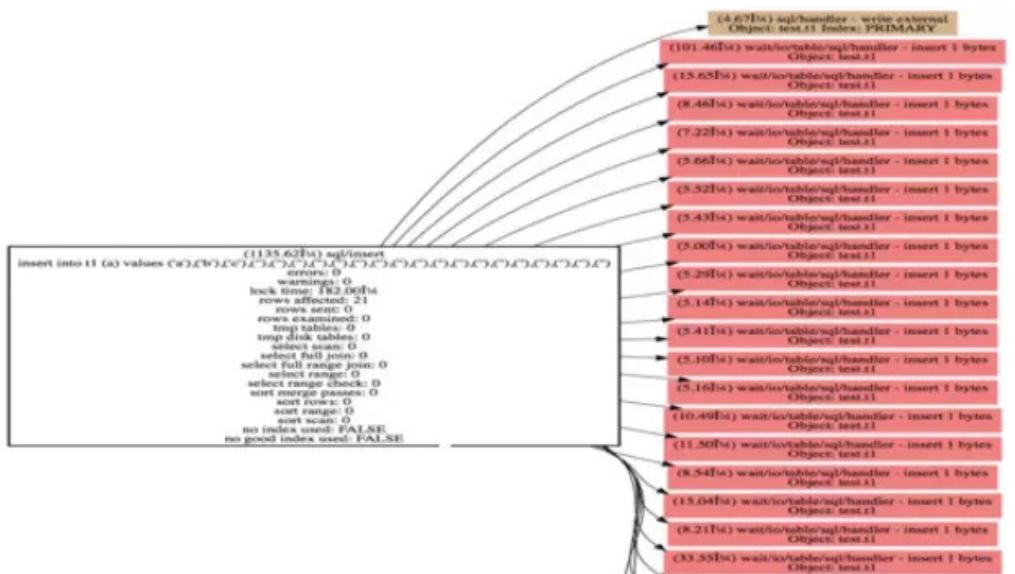
需要文件系统支持 PUNCH HOLE, ext4 和 xfs 都可以支持, 测试效果比目前压缩效果好一些。适配更多压缩算法, 支持 lz4 zlib, 功能上还不够稳定和成熟。



performance_schema 改进, 增加了很多统计信息表, metadata_locks, status_by_host , status_by_user, status_by_thread, 获 取 当前执行的慢查询 top10, 可以获取到更多状态信息。

THREAD_ID	VARIABLE_NAME	VARIABLE_VALUE
30	Bytes_received	6350
30	Bytes_sent	293718

上图是对单个 thread_id 吞吐量统计信息。新增加的 sys 数据库，相当于对 performance_schema 的数据整合。IO 信息和索引信息，相当于 Oracle 里的 V\$ Catalog，基于 ps_helper 实现。基于 performance schema 画的 SQL 执行时间分布图。



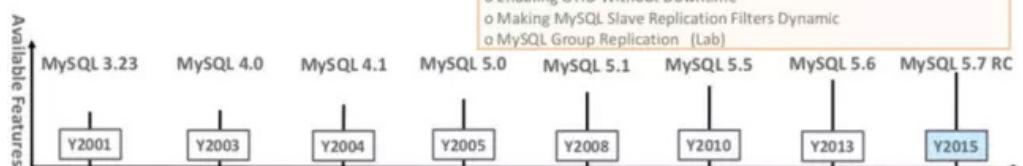
replication改进:

MySQL 3.23 - Generally Available, January 2001
 o MySQL Replication came to be [3.23.15 ~ May 2000]
 o Replication Filters
MySQL 4.0 - Generally Available, March 2003
 o Two Replication Threads instead of just one.
 o Slave Relay logs.
MySQL 4.1 - Generally Available, October 2004
 o Replication over SSL.
 o Disk synchronization options for binary log.

MySQL 5.0 - Generally Available, October 2005
 o Replication of Stored Routines and Triggers.
 o Slave retries transactions on transient errors.
MySQL 5.1 - Generally Available, November 2008
 o Row-based Replication (RBR).

MySQL 5.5 - Generally Available, December 2010
 o Semi-sync replication.
 o Replication Heartbeats.
 o RBR type conversion.
MySQL 5.6 - Generally Available, February 2013
 o Crash-safe Slaves.
 o Global Transaction IDs.
 o Replication Event Checksums.
 o Binary Log Group Commit.
 o Multi-threaded Slaves.
 o RBR enhanced.
 o MySQL Utilities 1.3, GA on August 2013

MySQL 5.7.7 RC, April 2015
 o Multi-Threaded Inter-Transactional Replication
 o Lossless Semi-Synchronous Replication
 o Multi-Source Replication
 o Enabling GTID Without Downtime
 o Making MySQL Slave Replication Filters Dynamic
 o MySQL Group Replication [Lab]



上图是 MySQL replication 的发展历史。最大的亮点 GTID 增强，支持在线调整 GTID。当然也不是简单的 SET @@GLOBAL.GTID_MODE = ON，步骤也是很复杂，不过至少不用停机，也是进步。从库可以不开启 log_slave_updates，通过引入 gtid_executed 表实现，对性能优一定帮助，大大简化切换流程。增强半同步复制，确保从库先收到，设置半同步从库个数。使用 mysqlbinlog 作为伪 slave 是个不错方案。

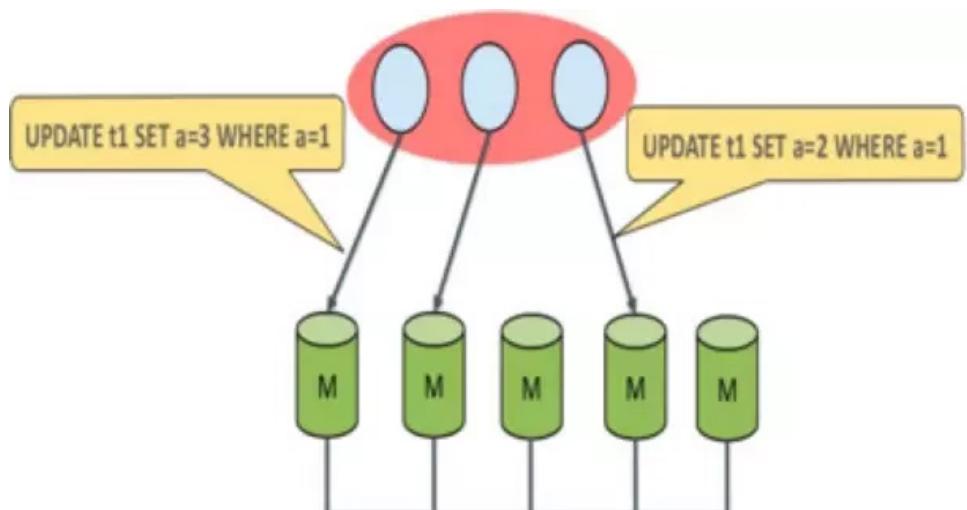


上图就是 loss-less 半同步和之前的区别。并行复制优化，Database 5.6 默认并行复制。logical-clock 5.7 引入，一个组提交内事务都可以并行，可以达到接近主库并发效果。

不同复制方案的可用性级别：

Simple replication	98-99.9%
Master-Master/MMM	99%
SAN	99.5-99.9%
DRBD, MHA, Tungsten Replicator	99.9%
NDBCluster, Galera Cluster	99.999%

5.7 引入的 group replication 也是为了提高可用性。多主复制，多点写入，内部检测冲突，保证一致性，自动探测。支持 GTID，共享 UUID，只支持 InnoDB，不支持并发 DDL。



安全方面密码自动过期，这个要注意，建议关闭。`default_password_lifetime` 控制过期默认一年。锁定用户，支持 SSL 访问，server 端利用 OpenSSL 加密。

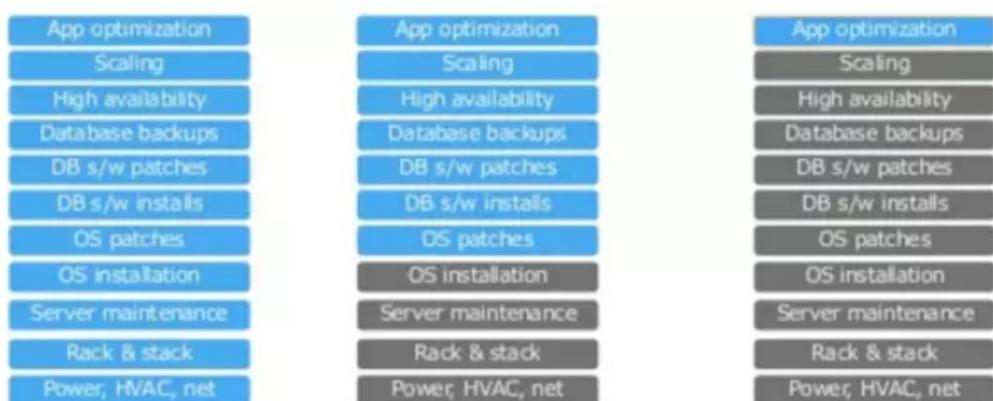
工具支持 mysqlpump，并行版 mysqldump，也是替换原生 mysqldump 和 mydumper 的。`--watch-progress` 查看 dump 进度，`--compress-output` 压缩。mysqldump 可以做为一个伪 slave 接受 binlog，做 binlog 备份的匪巢的方案，也支持 SSL。

从整体来说，MySQL 5.7 做的改进还是非常有吸引力的，不论是从运维角度还是性能优化上，当然真正在生产环境上遇到问题时在所难免的，要做好踩坑的准备。

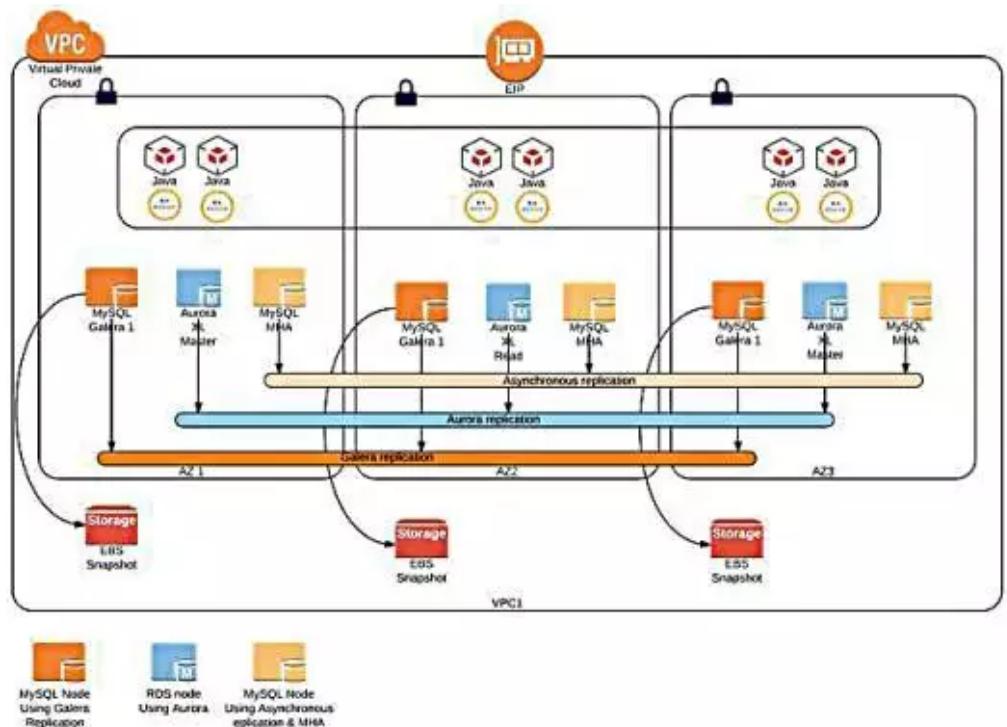
未来发展

RDS 服务

应该大家很多人都用过 RDS 服务，确实降低了使用成本。下图是各种架构区别，普通物理服务器，EC2，RDS，优势很明显。



阿里云 RDS MySQL，支持读写分离，多 zone，支持异地容灾，压缩（支持 TokuDB），阿里云宣称的一大亮点，代码控制能力强。老牌 RDS 服务 Amazon RDS，目前有 MySQL Aurora 和 MariaDB 这三种。Aurora 虽然目前来说，会有一些问题，但是方向还是不错的。HA 架构，从可用性来说 Galera > Aurora > MHA。



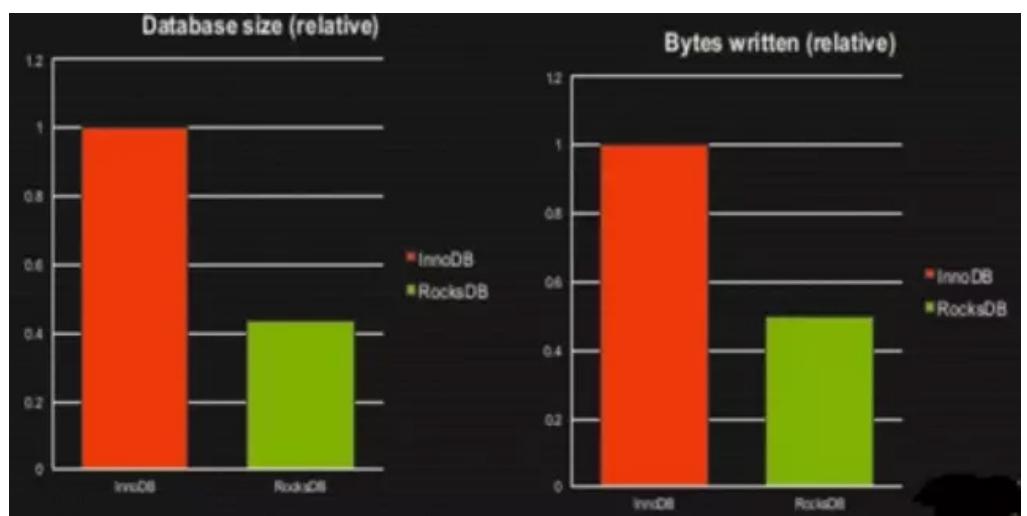
关于 RDS 服务的一些建议，数据库经验积累还是很重要的，面临过或解决的问题越多，提供的服务也相对越稳定。RDS 提供便利的同时，也存在数据安全的风险和 RDS 服务本身的 SLA 保证，是用户更关心的。

如何降低云服务商故障对业务影响，其实从 RDS 提供的性能指标考量，如果使用同等性能配置的物理服务器，RDS 成本还是偏高一些的。从功能上来说大部分 RDS 还算完备，具体哪些坑实际用的不多不好评判。

存储层的优化

LSM Tree : LevelDB, RocksDB, 适配高性能存储 SSD，更高的压缩比，更低的写入放大比例，缺点读性能差，适合写多读少场景。

MyRocks: MySQL + RocksDB:



系统层优化

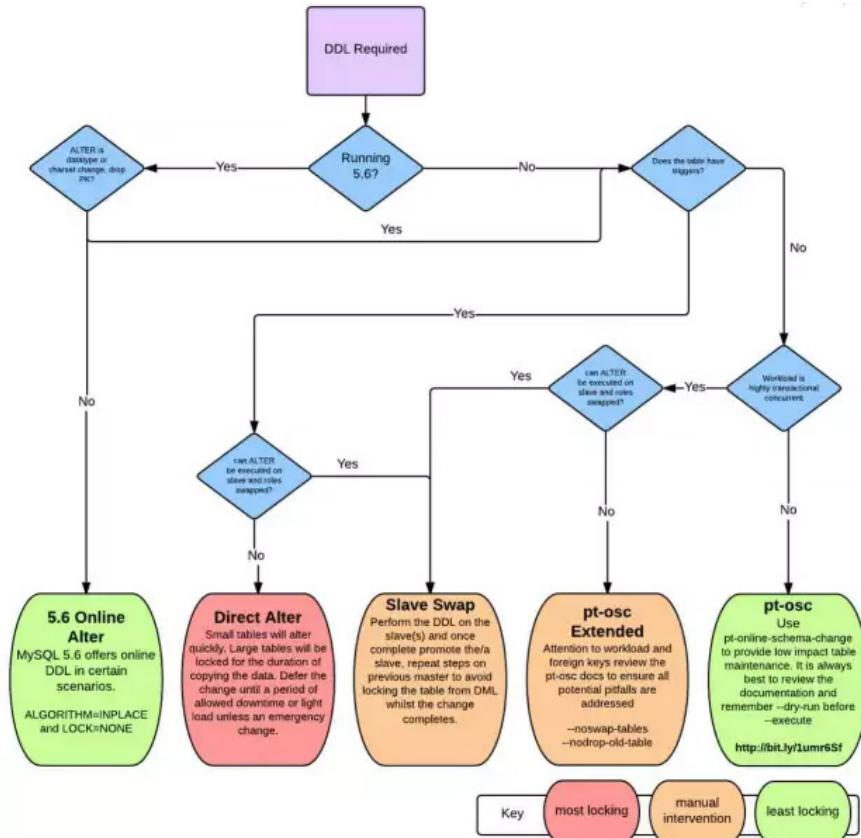
系统优化主要还是 IO 方面的，blk-mq、scsi-mq、IO 中断多队列、3D Xpoint 接近内存的访问速度和非易失存储，说不定以后整个数据库实例都可以放在这种介质上面，也是一场新的变革。

运维经验总结

数据恢复 备份 xtrabackup 物理为主，mydumper/mysqlpump 为辅，binlog 备份也是很重要的。恢复导出 SQL 文件正常恢复。

```
myloader  
xtrabackup  
InnoDB transportable space  
online ddl
```

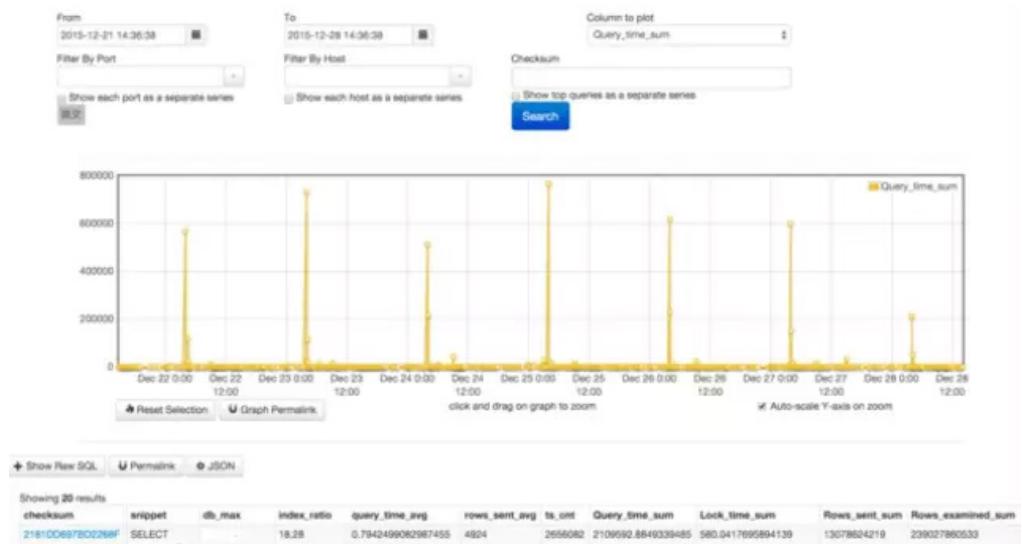
5.6 和 5.7 虽然一直在改善，但是在主从同步问题上依然有问题，下图是目前主流的 online DDL 方案。



总体使用 pt-osc 更通用一些，pt-osc 注意的一些坑，添加唯一键，导致数据丢失延时备份。行格式下，只在从库使用 OSC，丢数据。

MySQL 慢日志系统

基于 pt-query-digest logstash 和 Anemometer 实现，可以定期跟踪线上业务慢查询优化。



系统状态收集 基于某些特定条件触发，比如 MySQL 连接数增长到一定阈值，收集当前系统状态，方便后续问题排查。比如系统 top mpstat strace tcpdump 等，MySQL 的 processlist show engine innodb status 等。■

Q&A

Q1：请问 query cache 关闭的原因，MySQL 是否支持全同步？

query cache 访问需要获取一个全局锁，高并发时候争用很严重。更主要的是 query cache 缓存的效果并不好。原生 MySQL 的话，5.7 里的 group replication 是支持全同步的，还有目前的基于 galera 实现的 percona xtradb cluster 也是支持全同步的。

Q2：有没有 MySQL 的读写分离中间件？最好没有语言限制的，谢谢！

目前开源的中间件很多了，比如 mycat、atlas、vitess 等，你指的语言限制是指的对开发语言的兼容吧，我觉得每种都兼容的很好的不多，毕竟现在的中间件都是基于开发者本公司的现状开发的，在兼容性上不太能做到很完美。现在官方做了个 MySQL Fabric，现在应该也 GA 了，后面可以关注一下。

Q3：你们备份的策略(比如完整多久一次,增量多久一次,备份恢复测试多久一次)，备份对线上系统的影响如何控制都是选一个 slave 备份的么？

我们目前的策略是以周备+日备，结合 binlog 备份，理论可以恢复到一周内任意时间点。全部是全量备份，没有做增量。备份恢复测试，我们目前还没有做，基于 xtrabackup 备份数据可靠性还是

很高的，之前在新浪是有实现备份测试的，大概 2-3 天能对线上备份端口做一轮测试。目前备份是选择一个线上从库来做的，控制影响的话主要是通过对备份工具 xtrabackup 的并行度和 IO 来进行限制。

Q4：请问关于 DB 管理的问题，线上数据库是否可开放给业务方技术人员查询？开放到什么程度？有没有必要做 WEB 查询平台？

开发人员还是有必要开放的，如果完全禁止，很多业务数据查询的事情可能就需要 DBA 介入，其实效率是比较低的。当然在权限上可以限制，比如只开放读权限，禁止 dump 这种。对核心库限制要更严格一些。如果能做 Web 平台当然更好，可以在入口层做限制，后端控制数据访问频度和策略等。

Q5：MySQL 5.7 有没有对复合索引做优化，在违背 left most prefixing 时也能使用复合索引？

最左前缀的原则比较难突破，当然在 5.6 引入了 index condition pushdown 机制，可以在存储引擎层面做一些过滤，减少过滤行数，会有一定优化。

不一样的数据库

MongoDB 里程碑式的 WiredTiger 存储引擎

作者 / 毕洪宇

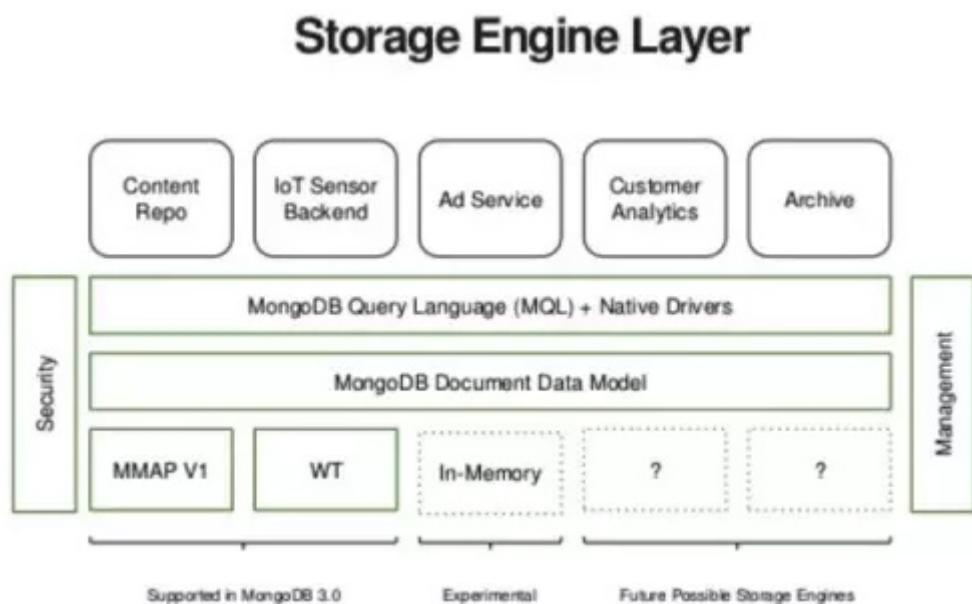
唯品会实时计算平台资深开发工程师。曾在 eBay、PPTV 任职 DBA。2012 年加入唯品会，历任 DBA、大数据平台工程师、实时平台工程师。专注数据库、大数据、分布式计算技术方向。

“2015 年 MongoDB 在年初发布了 3.0，标志一个新的里程碑，分离 Server 层和 Storage 层，并引入了 WiredTiger 存储引擎，基于此，我们也才敢再把 MongoDB 捡起来用。”

—— 毕洪宇

存储引擎的发展

MongoDB 拆分后的架构图如下：



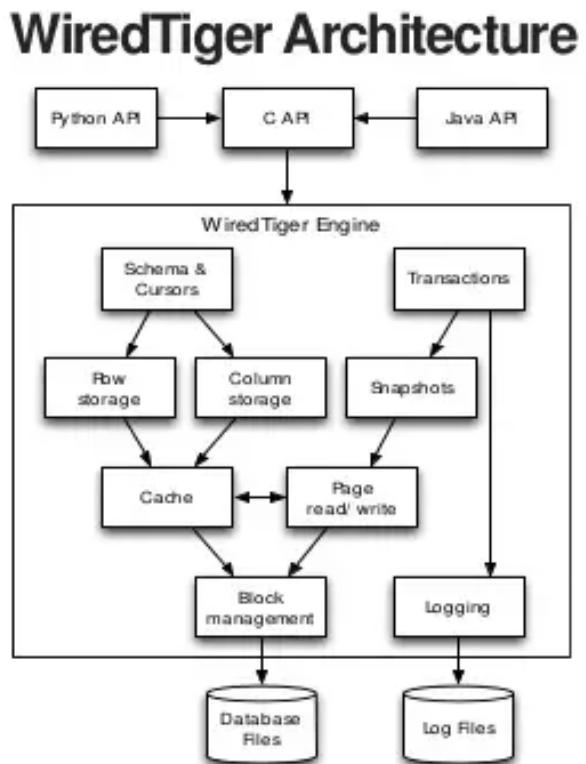
下面先从存储引擎层开始介绍一下一些改进点，现在已知支持的存储引擎包括：MMAP，WiredTiger，RocksDB，Memory 以及 Encryption 等。

MMAP

在 3.0.x 的时候依然还是默认的存储引擎，新的改进就是在并发方面，由原来的 database-level 锁转为 collection-level 锁，不过在 3.2 以后默认的存储引擎替换为 WiredTiger。

WiredTiger

应该是 MongoDB 2015 年最大的亮点了，内部架构图：



那么它有哪些比较突出的 feature 呢？

Document-level concurrency control

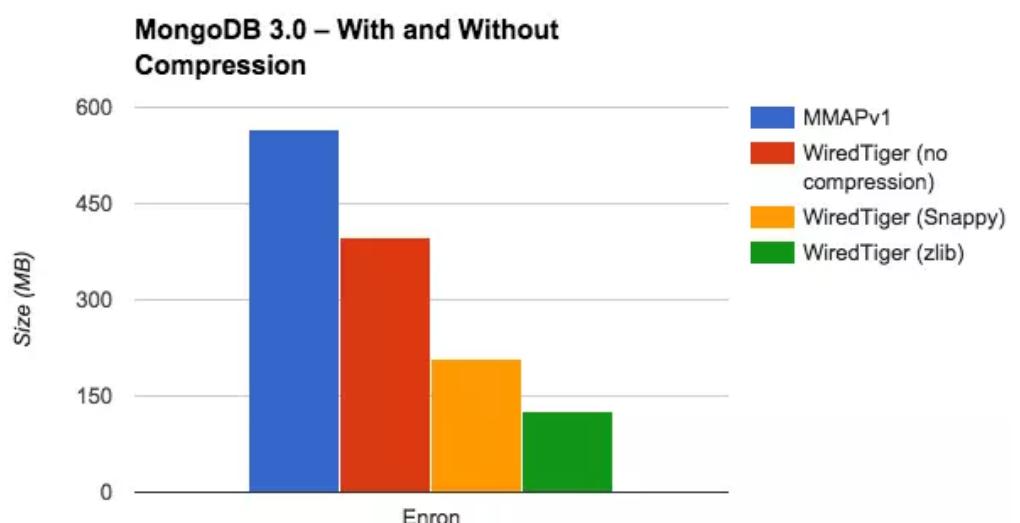
MongoDB 早期的锁粒度是简单粗暴的，instance-level, database-level 这也是一直被吐槽的地方；有一些场景为了适应这么粗粒度锁拆了一堆数据库出来，如果没有自动化管理配合的话，运维压力非常大。而 document-level 并发控制的引入使得 MongoDB 的并发能力得到极大的释放。

这里的数据也是存多版本的，有些类似 RDBMS 中 MVCC (multi version concurrency control)，来实现同一行的读写之间的并发访问，进一步提高系统的并发访问能力。

Compression

WiredTiger 支持两种压缩方式 snappy (默认) 和 zlib；

官方的测试效果：



我们一个业务场景之前在 2.6 上，1.18 Billion 记录数占用磁盘空间 3.12 TB，而升级到 3.0.x 后空间占用降到了 270 GB，使得 PCIE/SSD 的使用更划算。

我们在 MongoDB 3.0 正式生产之前使用 YCSB 进行了简单的压测：

Write-only 测试, standalone 模式

同样的 workload, enable 复制集

以及 read-only 场景

• 3.0.4 / snappy / replication / w=1 / enable journal	read:write=95:5
• [READ], AverageLatency(us), 394.409190121636	
• [READ], 99thPercentileLatency(ms), 0	
• [UPDATE], AverageLatency(us), 562.6090867394253	
• [UPDATE], 99thPercentileLatency(ms), 0	
insert query update delete getmore command % dirty % used flushes vsize res grn brw arw mchm netout conn	
+ 131554 7948 +# 480 4259# 8.3 92.6 8 2.35 2.85 8.8 39.1 17w 185m 66	
+ 131658 8016 +# 480 4341# 8.3 99.8 8 2.35 2.85 8.1 15.1 17w 185m 66	
+ 131762 7948 +# 480 4331# 7.4 92.8 8 2.35 2.85 20.1 12.1 17w 185m 66	
+ 131808 7924 +# 480 4256# 7.8 87.3 8 2.35 2.85 8.1 18.1 17w 187m 66	
+ 131862 8882 +# 396 4272# 7.7 81.1 8 2.35 2.85 22.1 3.1 17w 185m 66	
+ 146988 7683 +# 379 4871# 7.9 91.4 8 2.35 2.85 22.1 38.1 18.4 43 81 8 0 0.8 31.6 8.8 54122.6 8.5 1.8 9.3 0.85	
+ 159684 8829 +# 480 4380# 7.8 88.2 8 2.35 2.85 29.1 30.1 18.4 43 81 8 0 0.8 31.6 8.8 6879.3 8.8 0.8 9.3 0.85	
+ 159716 7888 +# 480 4370# 8.8 98.2 8 2.35 2.85 29.1 30.1 18.4 43 81 8 0 0.8 31.6 8.8 43279.9 8.8 2.1 9.3 0.85	
+ 160016 8146 +# 480 4331# 8.5 93.3 8 2.35 2.85 29.1 38.1 18.4 46 81 8 0 0.8 38.6 8.8 23976.7 8.3 2.8 8.4 0.85	
insert query update delete getmore command % dirty % used flushes vsize res grn brw arw mchm netout conn	
+ 148739 7566 +# 384 4244# 7.5 99.1 8 2.35 2.85 20.1 8.8 37.1 15.4 40 81 8 0 0.8 15.2 8.8 7852.5 8.8 0.8 9.3 0.84	
+ 148783 8897 +# 480 4331# 7.8 94.4 8 2.35 2.85 20.1 37.1 15.4 40 81 8 0 0.8 15.2 8.8 34122.6 8.5 1.8 9.3 0.84	
+ 148892 7566 +# 384 4244# 7.8 94.4 8 2.35 2.85 20.1 37.1 15.4 40 81 8 0 0.8 15.2 8.8 34122.6 8.5 1.8 9.3 0.84	
+ 152193 7948 +# 480 4221# 8.3 94.7 8 2.35 2.85 20.1 37.1 15.4 40 81 8 0 0.8 15.2 8.8 34122.6 8.5 1.8 9.3 0.84	
+ 149332 7997 +# 382 4310# 7.6 88.2 8 2.35 2.85 7.1 9.6 107 sys 20.1 low ts1 ts2 775 w/c w/n rxh/s w/b/s w/bet systh NetOut	
+ 159385 7963 +# 480 4311# 7.4 91.8 8 2.35 2.85 20.1 38.1 18.4 43 81 8 0 0.8 38.5 8.8 3223.4 8.8 0.9 8.5 0.84	
+ 159513 7913 +# 398 4468# 7.8 88.7 8 2.35 2.85 20.1 38.1 18.4 43 81 8 0 0.8 38.5 8.8 34122.6 8.5 1.8 9.3 0.84	
+ 151808 7923 +# 480 4311# 8.6 88.1 8 2.35 2.85 21.1 44.1 15.5 43 81 8 0 0.8 31.9 8.8 31852.0 8.8 2.4 8.3 0.84	
+ 148398 7547 +# 383 4311# 6.5 87.2 1 2.35 2.85 18.1 39.1 15.4 46 81 8 0 0.8 23.5 8.8 5827.1 8.8 3.2 8.3 0.84	
+ 143187 7539 +# 393 4461# 7.6 98.8 8 2.45 2.85 31.1 83.1 33.1 47 81 8 0 0.8 24.4 8.8 18274.7 8.3 2.1 8.4 0.84	
insert query update delete getmore command % dirty % used flushes vsize res grn brw arw mchm netout conn	
+ 182052 5226 +# 286 3881# 7.2 99.1 8 2.45 2.85 20.1 38.1 18.4 43 81 8 0 0.8 18.1 8.8 14042.6 8.8 0.9 8.5 0.84	
+ 182053 5226 +# 286 3881# 7.2 99.1 8 2.45 2.85 20.1 38.1 18.4 43 81 8 0 0.8 18.1 8.8 1733.0 8.8 0.9 8.5 0.84	
+ 182054 5226 +# 286 3881# 7.2 99.1 8 2.45 2.85 20.1 38.1 18.4 43 81 8 0 0.8 18.1 8.8 1733.0 8.8 0.9 8.5 0.84	
+ 182055 5226 +# 286 3881# 7.2 99.1 8 2.45 2.85 20.1 38.1 18.4 43 81 8 0 0.8 18.1 8.8 1733.0 8.8 0.9 8.5 0.84	
+ 35886 1802 +# 157 1751# 2.6 388.0 8 2.85 2.35 24.1 8.8 37.1 23.1 42 81 8 0 0.8 80385.0 8.8 338664.3 8.3 18.3 0.8 0.85	
+ 143864 7479 +# 388 4251# 6.4 98.6 8 2.85 2.35 3.1 1.1 22.1 34.1 31.1 8 0 0.8 3215.3 8.8 104632.8 167.3 51.3 8.3 18.8 0.85	
+ 151254 7881 +# 480 4343# 7.5 82.4 8 2.85 2.35 24.1 8.8 37.1 23.1 42 81 8 0 0.8 36585.9 8.8 139956.7 70.9 32.6 8.2 13.3 0.85	
+ 146599 7764 +# 388 4343# 7.1 82.3 8 2.85 2.35 24.1 39.1 35.1 47 81 8 0 0.8 41.6 8.8 139956.7 70.9 32.6 8.2 13.3 0.85	
+ 151378 7964 +# 480 4272# 7.7 98.5 8 2.85 2.35 34.1 77.1 35.1 47 81 8 0 0.8 280.2 8.8 63586.8 8.8 2.2 8.1 0.85	
+ 151813 7888 +# 480 4251# 8.8 91.3 8 2.85 2.35 34.1 77.1 35.1 47 81 8 0 0.8 280.2 8.8 63586.8 8.8 2.2 8.1 0.85	
+ 158798 7894 +# 480 4361# 7.7 92.8 8 2.85 2.35 34.1 77.1 35.1 47 81 8 0 0.8 34.6 8.8 33986.4 8.1 2.8 8.3 0.85	
+ 158799 7894 +# 480 4361# 7.7 92.8 8 2.85 2.35 34.1 77.1 35.1 47 81 8 0 0.8 15.3 8.8 7879.3 8.8 2.1 8.4 0.85	
+ 158800 7894 +# 480 4361# 7.7 92.8 8 2.85 2.35 34.1 77.1 35.1 47 81 8 0 0.8 35.7 8.8 54811.2 8.8 2.4 8.3 0.85	
+ 158801 7894 +# 480 4361# 7.7 92.8 8 2.85 2.35 34.1 77.1 35.1 47 81 8 0 0.8 18.1 8.8 139956.7 70.9 32.6 8.2 13.3 0.85	

在压测过程中，也遇到了一些问题，比如会有短暂的 stall，以及 replication enable 后 Primary 的写入吞吐量会降低非常多，这些都是 MongoDB 在和 WiredTiger 融合后存在的一个不足之处。

WiredTiger 在 eviction, checkpoint 和 capped collection 的处理算法还在持续改善中，感兴趣的可以在 JIRA (WT-1788, SERVER-16736) 上跟进。我们也在使用过程中也发现一些问题，也算是接下来对 MongoDB 2016 的一个展望。

另外，关于 WiredTiger 内部也有很多 internal 参数可以调整，这部分基本上在压测时没有调整，只是把 eviction worker min, max 都设置为 4。

RocksDB

该存储引擎是 facebook 开源的，写入强劲 NoSQL Storage (<http://rocksdb.org/>)，不过对于 MongoDB 来说是非官方 built-in 支持的存储引擎 (<https://github.com/mongodb-partners/mongo-rocks>)，有朋友在测试写入性能比较野性。

但是 WiredTiger 本身也支持 LSM option (默认是 btree)，可以通过 internal 的参数在创建表时指定，我简单测了一下 LSM 方式的写入也是很强，感兴趣的朋友可以深入 benchmark 下。

不过，关于 WiredTiger 的 LSM 特性和 MongoDB 的结合还在进行中，所以现在使用是修改 undocument parameter，并且也

存在比如 memory leak 的 Bug，这里只是 preview test，并没有生产环境使用。

Memory

企业版存储引擎，非开源，有些类似于 MySQL 的 memory engine，也是表级锁；这里简单提下感兴趣的朋友请参考 SERVER-1153。

复制集改进

Replication protocol 在 3.2 之后改进了 Replication Election/Consensus 算法，通过配置协议版本来指定。主要改进点：

- 引入 election ID 来加速 election progress，在这之前每轮选举无法给两个节点投票，并且每投票一次需要等待 30 秒才能进行下一轮，而引入 election ID 后可以加速这个进程，从而降低 MTTR(mean time to recovery)。简单来说 election ID 在每次“election attempt”的时候递增，用来区分每一轮的选举。
- 利用已有的复制通道完成心跳检测。在这之前复制集的每个节点默认每 2 秒会给集群中的每个节点发送心跳，这显然是不 scalable 的，因为这样会随着集群的增加导致“heartbeat storm”增加集群的 overhead。而采用新的算法每个节点只需要和上下游节点通过在 oplog 中写入额外的元数据来进行心跳检测，从而提高检测速度。

- 引入参数 election timeout。定义为：Node calls for an election when it hasn't heard from the primary within the election timeout。针对具体的网络环境来做 trade-off，如果网络环境比较差，可以提高该值减少 false detection，反之减小。

通过以上，可以进一步保证 MongoDB 的 MTTR。关于复制集另外一个改进，加入了 read concern。

Read Concern

在 3.2 之前复制集只支持 Write Concern，因此如果想做到强一致读的话只能将 Read_Preferanence 设置为 Primary，否则依然会导致 stale read。而 read concern 的引入弥补了这个不足。类似 CAP 中 $W + R > N$ 。

自动分片机制

对于自动分片的话，最大的一个改进是 Config server 支持复制集模式。Config Server 在 3.2 开始支持复制集模式，在这之前 Config Server 彼此是无感知的，不但可能会出现一致性问题，扩容和迁移都是非常痛苦，尤其是更换 hostname 的情况下。而在 3.2 之后 Config Server 完全可以利用复制集的特性了，Write Concern 是 Majority 同时 Read_Preference 是 Primary 来保证一致性和 HA。

其他新特性介绍

下面介绍一些其他特性。

Batch Index

在 MongoDB 中使用 `createIndex` 创建索引，如果想在一个表批量创建多个索引 MongoDB 提供了一个方便的命令：`createIndexes` (<https://docs.mongodb.org/manual/reference/command/createIndexes/#dbcmd.createIndexes>)。不过在 3.0 之前如果通过该命令给一个表创建多个索引的话，每个索引都需要进行一次 `FullTableScan` 这样显然是不高效的；在 3.0 之后，该命令只需要进行一次 `FullTableScan` 即可完成对所有索引的创建。

Partial Index

假设简单场景：一个流单系统表里包含两个字段 (`isDone default 0, add_time`)，已经处理过的 `isDone` 会被更新为 1，在流单取数的时候会跑如下 `query : {isDone : 1, add_time : {$ge : xx, $le : xx}}` 并且 `isDone = 1` 的记录总占比是小于 1% 的，因此创建索引 `{isDone : 1, add_time : 1}` 可以使得取数避免全表扫描，但是确浪费了存储大量 `isDone = 0` 的索引空间。在 3.2 之后 MongoDB 支持在创建索引时带一个过滤表达式 `partialFilterExpression` 来实现 partial index 的功能。在该场景创建索引如下：

```
createIndex (     {isDone : 1, add_time : 1},     { partialFilterExpression : {isDone : 1} } )
```

这样索引就只会索引 `isDone = 1` 的记录了。

Document Validator

MongoDB 的 schema free 是 MongoDB 刚推出时的一个卖点，不过估计已经被吐槽过无数次了。不一致的 schema，奇形怪状的数据等等，因此 data validation 的事情被推送到程序端来负责，使得这部分就变得复杂。

所以在 3.2 MongoDB 推出了 Validator。可以通过 `$type` (限制字段类型) 和 `$exists` (限制字段必须存在) 即可简单达成 schema 的限制，字段类型等。这个特性有些类似于 RDBMS 中的 check constraint。

文档链接：<https://docs.mongodb.org/manual/core/document-validation/>

Validator 带有 2 个参数 `validationAction` 和 `validationLevel` 来控制在违背约束时的行为。

		validationLevel		
		off	moderate	strict
ValidationAction	Warn	No checks	Warn on non-compliance for inserts & updates to existing valid documents. Updates to existing invalid docs OK.	Warn on any non-compliant document for any insert or update.
	error	No checks	Reject on non-compliant inserts & updates to existing valid documents. Updates to existing invalid docs OK.	Reject any non-compliant document for any insert or update. DEFAULT

Join

这个特性有一个小八卦，这个特性在 3.2 release 的时候是被宣布为 enterprise feature，然后社区就开始吐槽了：

How long until FindAndModify is moved to enterprise? I hope this is a joke.

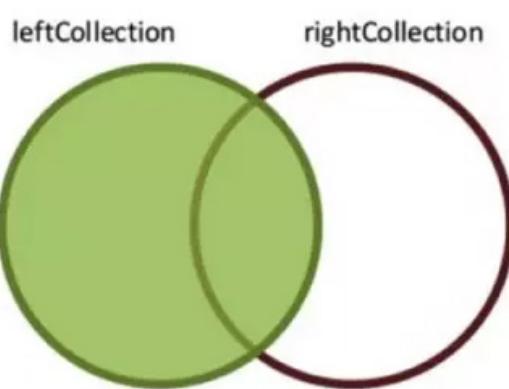
包括上面的 memory engine 被告知 enterprise feature 后也是有各种吐槽的，可以去看下 jira 和 mail list。

在 3.2 release 两周后，MongoDB CTO 宣布该 feature 开源，也算是比较折腾的了，闲言少叙功能描述如下：

Performs a left outer join to an unsharded collection in the same database to filter in documents from the “joined” collection for processing

本质就是 left outer join。

\$lookup



```
db.leftCollection.aggregate([{$lookup:{from: "rightCollection", localField: "leftVal", foreignField: "rightVal", as: "embeddedData"}}])
```

更详细的使用请查看文档: <https://docs.mongodb.org/manual/reference/operator/aggregation/lookup/>



Q&A

Q1：在一些涉及到地理位置的场景，例如陌陌附近的用户（基于地理位置和用户活跃时间排序），MongoDB 对这块也有支持，相较于 PostgreSQL 和 Solr 有什么优势么，该如何选择？

GIS 这块我个人是没有使用经验的，包括 MongoDB 的 GFS feature。我的看法还是团队对于哪个方案更熟悉，可控。网上

关于这些的对比很多，不过最终还是得多 benchmark，多看看 maillist 的吐槽，看看能不能接受副作用。我们对于 MongoDB 的使用主要是看重他在非事务场景下的性能和可用性。

Q2: MongoDB 相比 mysql 优势有哪些？未来可能剃掉关系型数据库吗？

相对优势有 bulit-in failover 支持、可配置的读写分离模式（这里主要指 read_preference，以及 tag-awareness，包括 read concern）、ddl-free（这里避免提 schema-free，因为很多场景是需要 schema 的），MySQL 的大表 DDL 还是问题，当然也是慢慢有解的、auto-sharding 也是比较方面的扩展。至于是否可以替代的话，其实今天我们回顾发现，都在不断的进化，比如 document validator/join 等 feature，包括 queryplan cache 等。而 MySQL 也会不断的改进自身的缺陷。所以我觉得不存在替换的可能。还是需要结合场景。说白了，还是 NoSQL，在事务方面想做到像 RDBMS 这样，还有很长的路要走。

Q3: MongoDB 做适合的场景是什么？2016 将会有哪些方向突破？

简单来说非常适合读多写少；因为如果想扩展写入的能力需要引入 auto-sharding，我个人觉得 MongoDB 在 auto-sharding 这块比较重。LVS->proxy->datanode 经典的三层架构运维比较痛。而复制集模式是比较好的用的，并且基本上 driver 都支持 topology auto discovery，所以从易用，易运维的角度我更推荐读多写少。

至于 2016 年的方向突破的话，也和 10gen 的聊过，他们有想去掉 mongos 的打算，做到类似 cassandra/redis-cluster 那种无中心的架构。我对此还是有些期待，起码在架构层面上简单一些，当然了，肯定一开始坑是不少的。另外就是 WiredTiger 和 MongoDB 的结合的稳定性还是有待提高的，我觉得性能还是会有更稳定的输出。

Q4：讲 MongoDB 和 Redis 同属 NoSQL，二者差异怎样，先选型上主要考虑哪些？

稳定的 latency 输出我会选择 Redis，MongoDB 当前还是会存在 spike 点。如果业务层面可以接受，或者通过架构层面来解决，那建议用 MongoDB。因为 MongoDB 天然支持 auto-failover 和读写分离，而 Redis 需要额外的工作。当然 redis-cluster 也可以支持 HA，不过生产使用还是慢慢在踩坑阶段，这方面还没有 MongoDB 成熟。

另外就是功能层面，MongoDB 支持的 API 和 Redis 支持的数据结构的差异。比如 mongodb 的 findAndModify，比如二级索引。而 Redis 支持的比如 HLL, zset，也是用 MongoDB 很难直接替代的。还有就是服务端的线程模型，Redis 是单线程而 MongoDB 是 one-thread-per-connection，这点也需要考虑进去，比如读热数据。

Q5：感觉这个新的存储引擎有些像关系数据库了，MongoDB 是怎么保证比关系数据库性能高的？

简单来说就是做的少，就做得快。我记得 Oracle 大牛 Tom 说过一句话翻译的大概意思是：如果想做一件事儿很快，最快的方法就是不做。当时的场景是讲 SQL parse 的。放到这样一样的道理。毕竟 MongoDB 相对 RDBMS 去掉了事务的支持，也就去掉太多事情了。快是应该的，但是现在和 WiredTiger 还是在不断磨合阶段，稳定输出有待提高。

Q6：有没有什么 MongoDB 和 ZooKeeper 结合的方案？

MongoDB 和 ZooKeeper 我们还没有结合在一起使用过。因为他的 failover 是基于内部协议的，没有通过 zk 来协调。比如 HBase 包括 Kafka 都在慢慢去掉对 zk 的依赖，使用内部的协议来完成可用性及拓扑发现的 feature。

不一样的数据库

肖鹏：微博数据库那些事儿



作者 / 图灵访谈



肖鹏，微博研发中心技术经理，主要负责微博数据库（MySQL/Reids/HBase/Memcached）相关的业务保障、性能优化、架构设计，以及周边的自动化系统建设。经历了微博数据库各个阶段的架构改造，包括服务保障及SLA体系建设、微博多机房部署、微博平台化改造等项目。10年互联网数据库架构和管理经验，专注于数据库的高性能和高可用技术保障方向。

问：您是如何和 MySQL 结缘，并成为数据库方面的专家的？

与 MySQL 结缘主要也是源于兴趣，第一份工作在一家小公司，各个领域的工作都会有接触，全都做下来发现还是对数据库最感兴趣，所以就一直从事和数据库相关的技术工作了。随着工作年限的增加，我在数据库方面积累的经验也逐步增加，越来越发现数据库管理员（DBA）是一个偏实践的工种，很多理论上的东西在现实中会有各种的变化，比如「反范式」设计等等。所以我建议大家：如果想成为数据库方面的专家，一定要挑选好环境，大平台很多时候会由于量变引发质变产生很多有挑战的问题，而解决这些问题成为技术专家的必经之路。

问：一路走来，微博的数据规模和业务场景都发生了很大的改变，请问新浪 MySQL 集群结构发展到今天都经历了哪些阶段？

微博到今天已经有 6 年了，非常有幸全程参与了这 6 年的变化。新浪的 MySQL 集群结构主要经历了 3 次重大的变化。

第一阶段：初创阶段

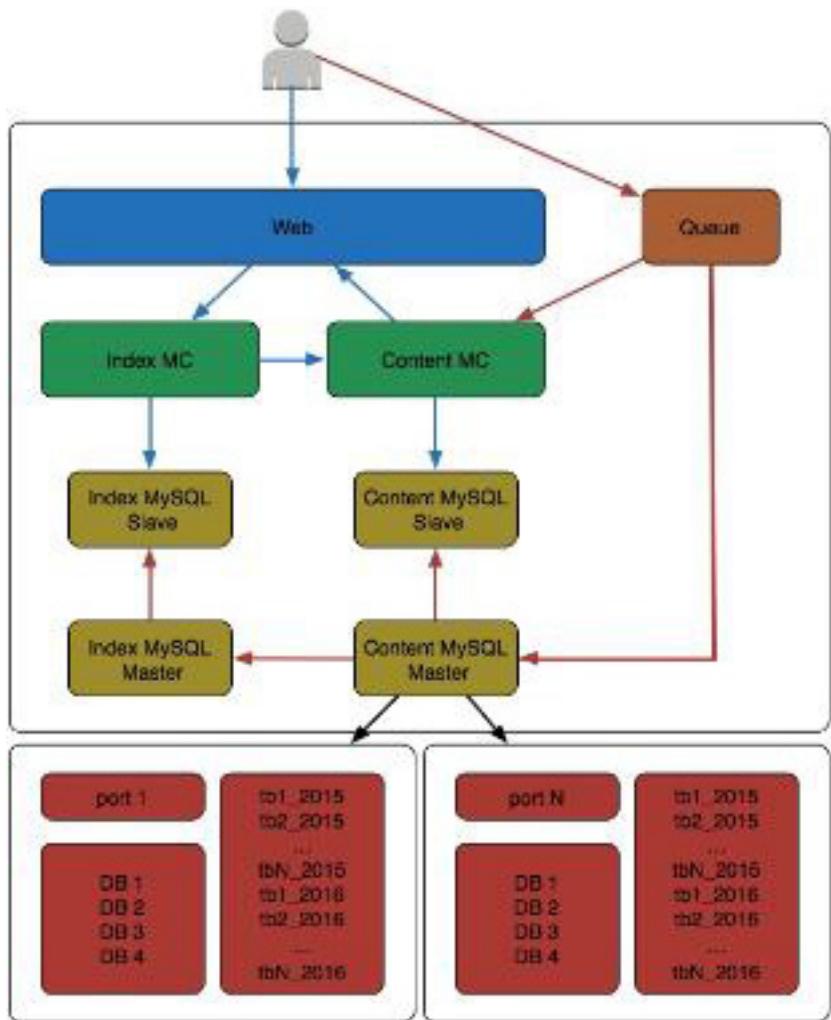
初期微博作为一个内部创新产品，功能比较简单，而数据库架构也采用的标准 1M2S1MB 结构，按照读写分离设计，主库承担写入，而从库承担访问。如果访问压力过大，可以通过扩容从库的数量获得 scale out 的能力。

第二阶段：爆发阶段

随着微博上线之后的用户活跃度增加，数据库的压力也与日俱增。我们首先通过采购高性能的硬件设备来对单机性能进行 scale up，以满足支撑业务的高速发展需求。然后，通过使用高性能设备争取来的时间对与微博进行整体上的业务垂直拆分，将用户、关系、博文、转发、评论等功能模块分别独立存储，并在垂直拆分的基础上，对一些预期会产生海量数据的业务模块再次进行了二次拆分。

以博文为例，博文是微博用户主要产生的内容，可以预见会随着时间维度不断增大，最终会变得非常巨大。如何在满足业务性能需求的情况下，尽可能使用较少的成本存储，这是我们面临的一个比较有挑战的问题。

- 首先我们将索引同内容进行了拆分，因为索引所需存储空间较少，而内容存储所需空间较大，且对这两者的使用需求也不尽相同。
- 然后，分别对索引和内容采用先 hash，然后在按照时间维度拆分的方式进行水平拆分，尽量保障每张表的容量在可控范围之内，以保障查询的性能指标。
- 最后，业务先通过索引获得实际所需内容 id，然后再通过内容库获得实际的内容，并通过部署 memcached 来加速整个过程。虽然看上去步骤变多，但是实际效果完全可以满足业务需求。



第三阶段：沉淀阶段

在上一个阶段，微博的数据库经历了很多的拆分改造，这也就直接造成了规模的成倍增长，而随着业务经历了高速增长之后，也开始趋于稳定。在这个阶段，我们开始着重进行自动化的建设。将之前在快速扩张期间积攒下来的经验转变为自动化工具，对外形成标准化和流程化的平台服务。我们相继建设改造了备份系统、监控系统、

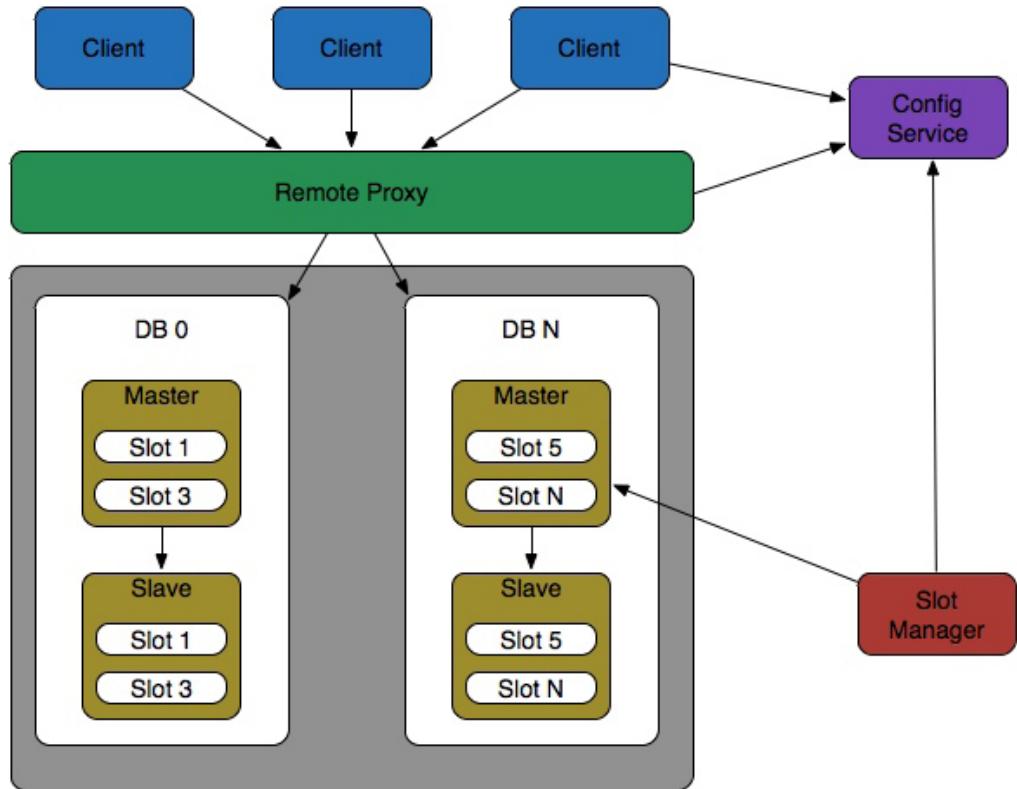
AutoDDL 系统、MHA 系统、巡检系统、慢查系统、maya 中间件系统，等等。并且为了提高业务使用效率和降低沟通成本，相对于内部管理系统，我们重新开发了 iDB 系统对数据库平台的用户使用。通过 iDB 系统，用户可以便捷地了解自己业务数据库的运行状态，并可以直接提交对数据库的 DDL 修改需求。DBA 仅需点击审核通过，即可交由 Robot 在线上执行，不但提高了工作效率，也提高了安全性和规范性。

问：在过去的 2015 年，新浪数据库平台都做了哪些重大的改进和优化？

数据库平台并不仅有 MySQL 还有 Redis、Memcached、HBase 等数据库服务，而在缓存为王的趋势下，我们 2015 年重点将研发精力投入在 Redis 上。

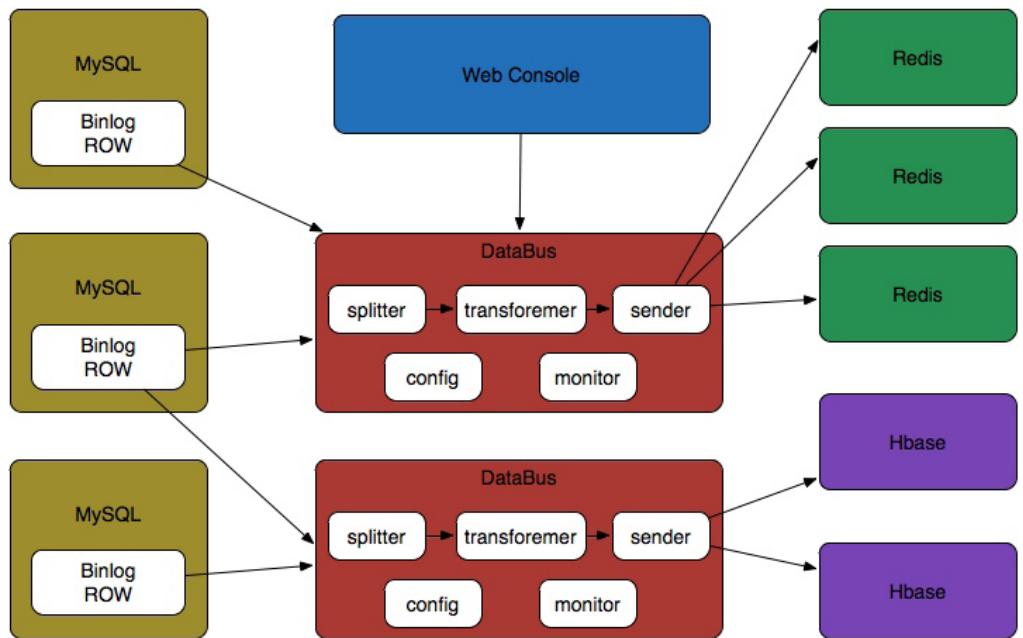
Redis 中间件

在 2015 年我们自研的 Redis 中间件 tribe 系统完成了开发和上线。tribe 采用有中心节点的 proxy 架构设计，通过 configer server 管理集群节点，并借鉴官方 Redis cluster 的 slot 分片的设计思路来完成数据存储。最终实现了路由、分片、自动迁移、fail over 等功能，并且预留了操作和监控的 API 接口，以便同其他的自动化运维系统对接。



Databus

由于我们先有MySQL后有的Redis和HBase等数据库，故存在一种场景就是目前数据已经写入到MySQL中，但是需要将这些数据同步到其他数据库软件中。所以我们为此开发了Databus，可以基于MySQL的binlog将数据同步到其他异构的数据库中，并且支持自定义业务逻辑。目前已经实现了MySQL到Redis和MySQL到HBase的数据流向，下一步计划开发Redis到MySQL的数据流向。



问：微博用户库设计采用了反范式设计，但是反范式设计也有自己的问题，比如在规模庞大时，数据冗余多，编码及维护的困难增加。请问你们是如何解决这些问题的？

反范式设计带来便利的同时确实也带来了一些问题，尤其是在数据规模变大之后，通常来说会有如下几种解决方案：

预拆分，在接到需求的时候提前针对于容量进行评估，并按照先垂直后水平进行拆分，如果可以按照时间维度设计，那就纳入归档机制。通过数据库的库表拆分，解决容量存储问题。

引入消息队列，利用队列的一写多读特性或多队列来满足冗余数据的多份写入需求，但仅能保障最终一致性，中间可能会出现数据延迟。

引入接口层，通过不同业务模块的接口将数据进行汇总之后再返回给应用层，降低应用层开发的编码复杂度。

问：微博平台当前在使用并维护着可能是世界上最大的 Redis 集群，在应用 Redis 的过程中，你们都产生了哪些具有独创性的解决方案？

微博使用 Redis 的时间较早，并且一开始量就很大，于是在实际使用过程中遇到了很多实际的问题，我们的内部分支版本都是针对这些实际问题进行优化的。比较有特点的有如下三个：

增加基于 pos 位同步功能

在 2.4 版本中，Redis 的同步一旦出现中断就会重新将主库的数据全部传输到从库上，这就会造成瞬时的网络带宽峰值，并且对于数据量较大的业务，从库恢复的时间较为缓慢。为此我们联合架构组的同学借鉴 MySQL 的主从同步复制机制，将 Redis 的 aof 改造为记录 pos 位，并让从库记录已经同步的 pos 位置。这样在网络出现波动的时候即使重传，也仅仅只是一部分数据，并不会影响到业务。

在线热升级

在使用初期，由于很多新功能的加入，Redis 版本不断升级，每次升级为了不影响业务都需要进行主库切换，对于运维带来了很大的挑战。于是我们开发了热升级机制，通过动态加载 libredis.so 来实现版本的改变，不再需要进行主库切换，极大地提升了运维的效率，也降低了变更带来的风险。

定制化改造

在使用 Redis 的后期，由于微博产品上技术类的需求非常多，所以我们为此专门开发了兼容 Redis 的 redisscounter 专门存储技术类的数据。通过使用 array 替换 hash table 极大降低了内存占用。而在此之后，我们开发了基于 bloom filter 的 phantom 解决判断类场景需求，

问：你在一次分享中曾经透露新浪数据库备份系统正计划结合水位系统实现智能扩容，请问现在实现到哪一步了？

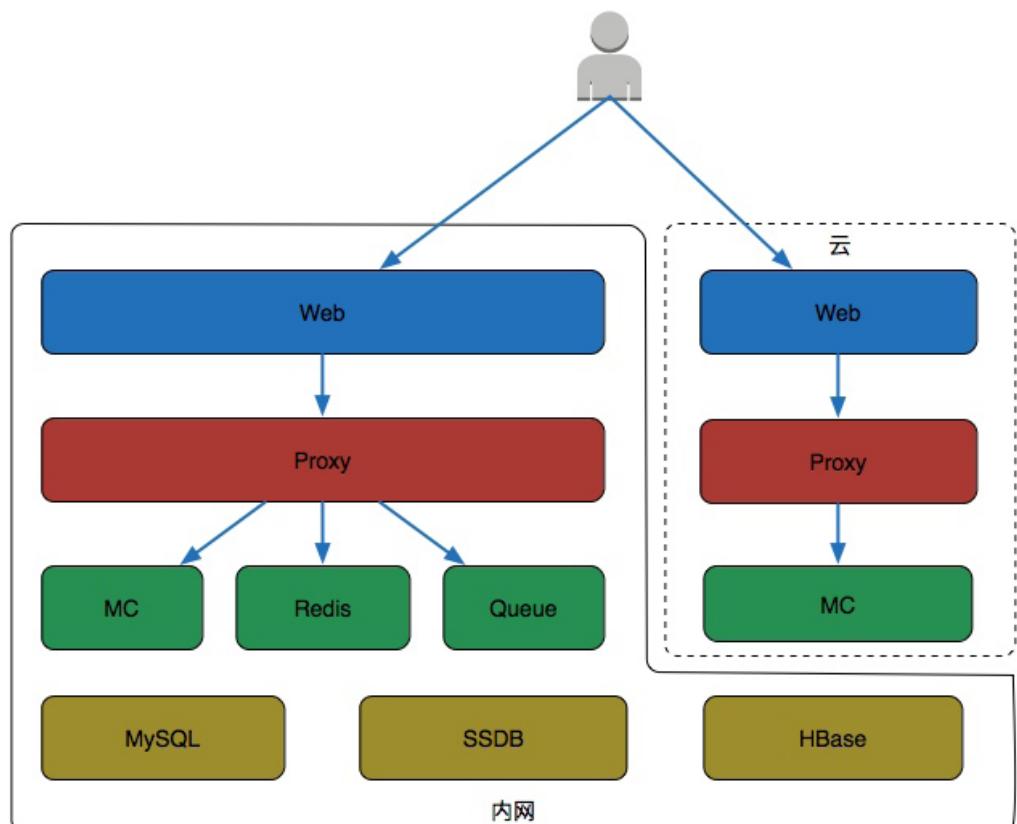
目前这个事情的进展不是很理想，主要我们发现 MySQL 的扩容速度跟不上业务的变化，有些时候扩容完毕之后业务的高峰已经过去了，接下来就需要做缩容，等于做了无用功。所以，目前我们的思路改为扩缩容 cache 层，首先实现 cache 层的自动扩缩容，然后同业务监控系统或者接入层的自动化系统进行联通，比如，如果计算节点扩容 100 个 node，那么我们的 cache 层就联动扩容 20node，以此来达到业务联动。

问：未来 5 年内新浪数据库还将做出什么样的改变？

随着业务的发展，会遇到越来越多的场景，我们希望可以引进最适合的数据库来解决场景问题，比如 PostgreSQL、SSDB 等。同时，利用 MySQL 新版本的特性（比如 5.7 的并行复制、GTID、动态调整 BP），不断优化现有服务的性能和稳定性。

另外，对于现有的NoSQL服务，推进服务化，通过使用proxy将存储节点进行组织之后对外提供服务。对外降低开发人员的开发复杂度和获取资源的时间，对内提高单机利用率并解决资源层横向扩展的瓶颈问题。

同时，尝试利用各大云计算的资源，实现cache层的动态扩缩容，充分利用云计算的弹性资源，解决业务访问波动的问题。



问：你如何看待新兴的NewSQL？

数据库圈子的变化确实很快，NoSQL还刚刚方兴未艾，NewSQL又开始你方唱罢我登场。我个人并不认为某种数据库会取代另一种数据库，就如同NoSQL刚刚兴起的时候很多声音说会它彻底取代MySQL，但是从实际情况看依然还是互依并存的关系。以我负责的集群来说，反倒是MySQL更多一些。我个人认为，每种数据库都有他最擅长的场景，在特定场景下它就是最佳的数据库，但是如果脱离了场景则很难说谁优谁劣。

问：能否请您横向对比一下MySQL、MongoDB，以及PostgreSQL？

我个人MySQL使用得较多，MongoDB和PostgreSQL都有过一些接触，MySQL作为LAMP中的一员，老实说在大部分场景都是合适的，尤其是在并发和数据库量并没有达到一个很大值的时候。但是，在某些场景下MongoDB和PostgreSQL确实更胜一筹。

比如我们在门户的新闻发布系统中使用了MongoDB，其schema less的设计模式和新闻非常贴合，而其sharding功能又解决了容量上的横向扩张问题，在这个场景下，MySQL并不具备什么优势。

而在LBS(基于地理位置信息服务)相关的方面，PostgreSQL和PostGIS更具有优势，利用其空间数据索引R-tree和实体类型点、线、线段、方形，以及特定的函数，可以很方便地实现空间计算需求。

就我个人来说，每种数据库都有其擅长的场景。如果没有特殊的架构需求，一般选择 MySQL 都不会出问题，而如果有特殊的架构需求，那么就需要根据需求的特点来选择不同的数据库了。

问：对于想要掌握 MySQL 的同学，您有哪些学习上的建议？

首先，多读书，至少将《High Performance MySQL》英文版通读一遍。

其次，有条件的话，最好找一些大平台历练一下，在很多情况下经验和能力等同于你解决过的问题的广度和深度，而环境决定你遇到的问题。

最后，有机会的话多做一些技术分享，很多知识点自己明白和能给别人讲明白是两个完全不同的境界。■

高可用架构部分分享讲师名单 (按姓名首字母排序)

- 陈飞, 新浪微博技术经理
- 常雷博士, Pivotal 中国研发中心研发总监, HAWQ 并行 Hadoop SQL 引擎创始人, Pivotal HAWQ 团队负责人
- 陈宗志, 奇虎 360 基础架构组高级存储研发工程师
- 杜传赢, Google 研发工程师
- 董西成, Hulu 网高级研发工程师, dongxicheng.org 博主
- 付海军, 时趣互动技术总监
- 冯磊, 新浪微博技术保障架构师
- 高磊, 雪球运维架构师
- 郭斯杰, Twitter 高级工程师
- 郭伟, 腾讯安全架构师
- 高永超, 宜信大数据创新中心云平台运维专家
- 黄东旭, PingCAP CTO, 开源项目 Codis co-author
- 霍泰稳, InfoQ、极客邦科技创始人兼 CEO
- 蒋海滔, 阿里巴巴国际事业部 高级技术专家
- 金自翔, 百度资深研发工程师
- 吕毅, 前百度资深研发工程师
- 马利超, 小米科技的系统研发与大数据工程师

- 马涛，前迅雷网络 CDN 系统研发工程师，前 EMC/Pivotal Hawq 研发工程师
- 彭哲夫，芒果 TV 平台部核心技术团队负责人
- 秦迪，新浪微博研发中心技术专家
- 沈剑，58 到家技术总监 / 技术委员会负责人
- 孙其瑞，得图技术总监
- 孙宇聪，Coding.net CTO，前 Google SRE
- 孙子荀，腾讯手机 QQ 公众号后台技术负责人
- 谭政，Hulu 网大数据基础平台研发工程师
- 唐福林，雪球首席架构师
- 田琪，京东云数据库技术负责人
- 王富平，1 号店搜索与精准化部门架构师
- 王劲，酷狗音乐大数据架构师
- 王晶昱，花名沈询，阿里资深技术专家
- 王康，奇虎 360 基础架构组资深工程师
- 王新春，大众点评网数据平台资深工程师
- 王晓伟，麦图科技
- 王渊命，Grouk 联合创始人及 CTO
- 王卫华，百姓网资深开发工程师及架构师
- 温铭，奇虎 360 企业安全服务端架构师，OpenResty 社区咨询委员会成员

- 萧少聪, 阿里云 RDS for PostgreSQL/PPAS 云数据库产品经理
- 许志雄, 腾讯云产品运营负责人
- 颜国平, 腾讯云天御系统研发负责人
- 闫国旗, 京东资深架构师, 京东架构技术委员会成员
- 杨保华, IBM 研究院高级研究员
- 杨尚刚, 美图公司数据库高级 DBA
- 尤勇, 大众点评网资深工程师, 开源监控系统 CAT 开发者
- 张开涛, 京东高级工程师
- 赵磊, Uber 高级工程师
- 张亮, 当当网架构师、当当技术委员会成员、消息中间件组负责人
- 张虔熙, Hulu 网 HBase contributor
- 赵星宇, 新浪微博 Android 高级研发工程师
- 周洋, 奇虎 360 手机助手技术经理及架构师



扫描二维码关注微信公众号
或搜索【ArchNotes】高可用架构

出品人 杨卫华

编辑/整理 李盼 刘伟 藏秀涛
启明 余长洪 刘芸
王杰 陈刚

设计 大胖

署名文章及插图版权归原作者所有。

商务及内容合作, 请邮件联系 iso1600@gmail.com