17 | 为什么不建议你使用自增主键?

2020-09-16 王磊

分布式数据库30讲

进入课程 >

讲述：王磊

时长 15:18　大小 14.02M

你好，我是王磊，你也可以叫我 Ivan。

有经验的数据库开发人员一定知道，数据库除了事务处理、查询引擎这些核心功能外，还会提供一些小特性。它们看上去不起眼，却对简化开发工作很有帮助。

不过，这些特性的设计往往是以单体数据库架构和适度的并发压力为前提的。随着业务规模扩大，在真正的海量并发下，这些特性就可能被削弱或者失效。在分布式架构下，是否要延续这些特性也存在不确定性，我们今天要聊的自增主键就是这样的小特性。

虽然，我对自增主键的态度和第 16 讲提到的存储过程一样，都不推荐你使用，但是原因各有不同。存储过程主要是工程方面的原因，而自增主键则是架构上的因素。好了，让我们进入正题吧。

自增主键的特性

自增主键在不同的数据库中的存在形式稍有差异。在 MySQL 中，你可以在建表时直接通过关键字 auto_increment 来定义自增主键，例如这样：

复制代码

```
create table ‘test’ (
  ‘id’  int(16) NOT NULL AUTO_INCREMENT,
  ‘name’  char(10) DEFAULT NULL,
  PRIMARY KEY(‘id’)
) ENGINE = InnoDB;
```

而在 Oracle 中则是先声明一个连续的序列，也就是 sequence，而后在 insert 语句中可以直接引用 sequence，例如下面这样：

复制代码

```
create sequence test_seq increment by 1 start with 1;
insert into test(id, name) values(test_seq.nextval, ' An example ');
```

自增主键给开发人员提供了很大的便利。因为，主键必须要保证唯一，而且多数设计规范都会要求，主键不要带有业务属性，所以如果数据库没有内置这个特性，应用开发人员就必须自己设计一套主键的生成逻辑。数据库原生提供的自增主键免去了这些工作量，而且似乎还能满足开发人员的更多的期待。

这些期待是什么呢？我总结了一下，大概有这么三层：

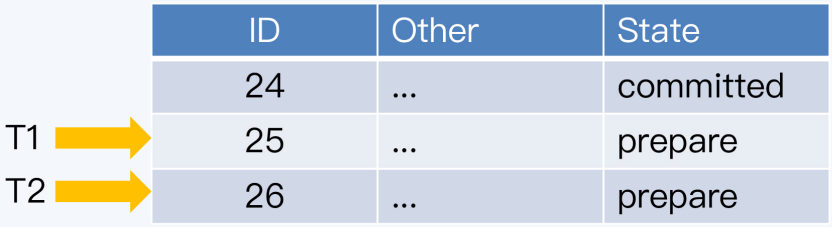
> 首先是唯一性，这是必须保证的，否则还能叫主键吗？
>
> 其次是单调递增，也就是后插入记录的自增主键值一定比先插入记录要大。
>
> 最后就是连续递增，自增主键每次加 1。有些应用系统甚至会基于自增主键的“连续递增”特性来设计业务逻辑。

单体数据库的自增主键

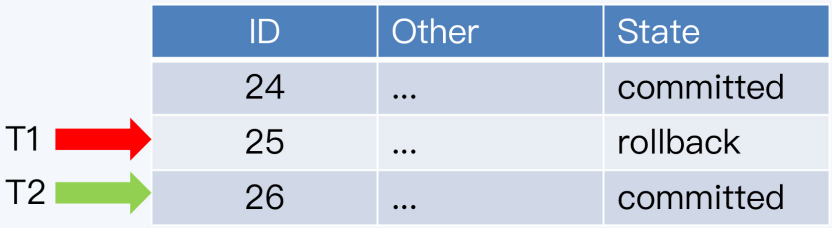
但是，我接下来的分析可能会让你失望，因为除了最基本的唯一性，另外的两层期待都是无法充分满足的。

无法连续递增

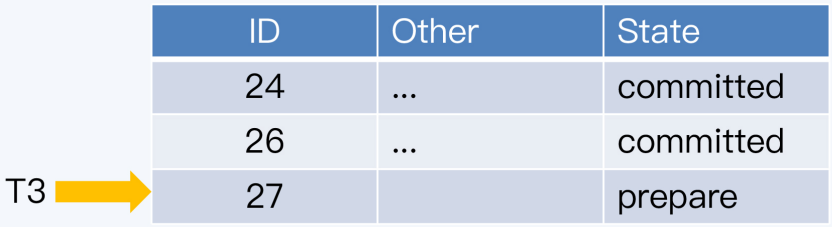
首先说连续递增。在多数情况下，自增主键确实表现为连续递增。但是当事务发生冲突时，主键就会跳跃，留下空洞。下面，我用一个例子简单介绍下 MySQL 的处理过程。



两个事务 T1 和 T2 都要在同一张表中插入记录，T1 先执行，得到的主键是 25，而 T2 后执行，得到是 26。



但是，T1 事务还要操作其他数据库表，结果不走运，出现了异常，T1 必须回滚。T2 事务则正常执行成功，完成了事务提交。



这样，在数据表中就缺少主键为 25 的记录，而当下一个事务 T3 再次申请主键时，得到的就是 27，那么 25 就成了永远的空洞。

为什么不支持连续递增呢？这是因为自增字段所依赖的计数器并不是和事务绑定的。如果要做到连续递增，就要保证计数器提供的每个主键都被使用。

怎么确保每个主键都被使用呢？那就要等待使用主键的事务都提交成功。这意味着，必须前一个事务提交后，计数器才能为后一个事务提供新的主键，这个计数器就变成了一个表级锁。

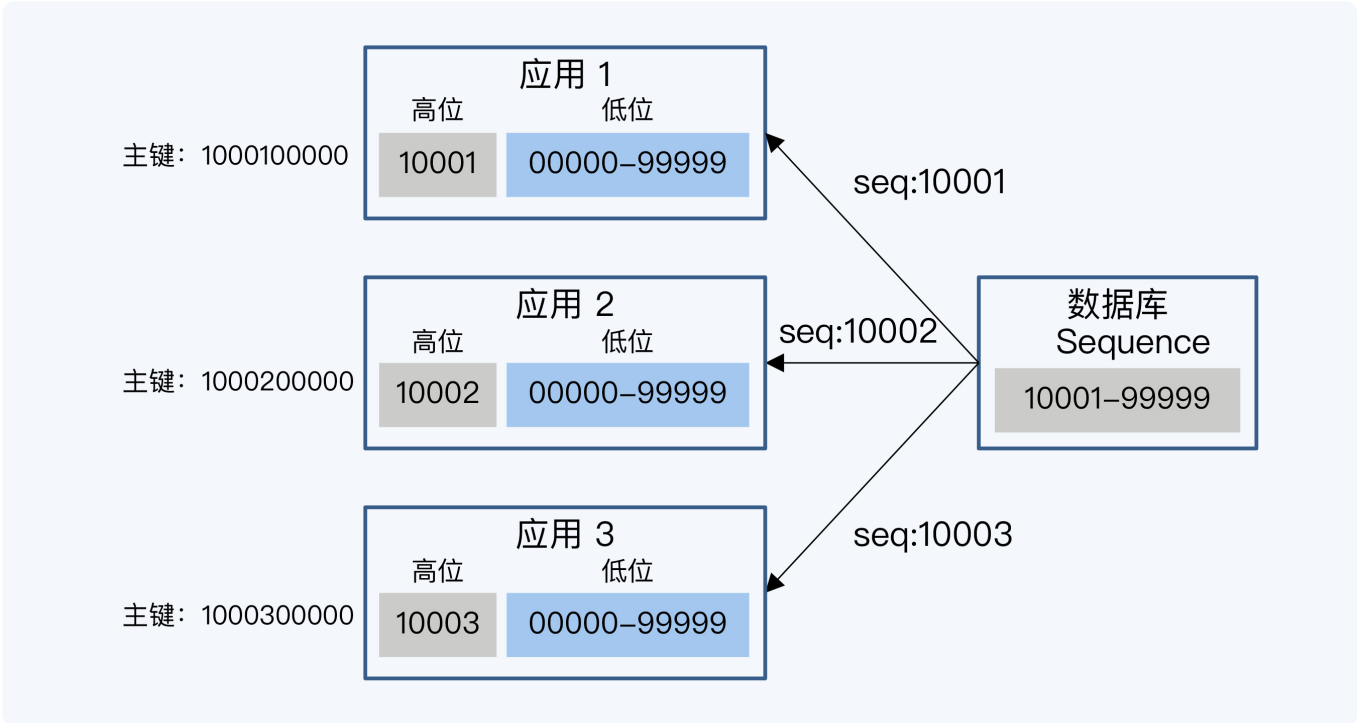
显然，如果存在这么大粒度的锁，性能肯定会很差，所以 MySQL 优先选择了性能，放弃了连续递增。至于那些因为事务冲突被跳过的数字呢，系统也不会再回收重用了，这是因为要保证自增主键的单调递增。

看到这里你可能会想， 虽然实现不了连续递增，但至少能保证单调递增，也不错。那么，我要再给你泼一盆冷水了，这个单调递增有时也是不能保证的。

无法单调递增

对于单体数据库自身来说，自增主键确实是单调递增的。但使用自增主键也是有前提的，那就是主键生成的速度要能够满足应用系统的并发需求。而在高并发量场景下，每个事务都要去申请主键，数据库如果无法及时处理，自增主键就会成为瓶颈。那么，这时只用自增主键已经不能解决问题了，往往还要在应用系统上做些优化。

比如，对于 Oracle 数据库，常见的优化方式就是由 Sequence 负责生成主键的高位，由应用服务器负责生成低位数字，拼接起来形成完整的主键。



图中展示这样的例子，数据库的 Sequence 是一个 5 位的整型数字，范围从 10001 到 99999。每个应用系统实例先拿到一个号，比如 10001，应用系统在使用这 5 位为作为高位，自己再去拼接 5 位的低位，这样得到一个 10 位长度的主键。这样，每个节点访问一次 Sequence 就可以处理 99999 次请求，处理过程是基于应用系统内存中的数据计算主键，没有磁盘 I/O 开销，而相对的 Sequence 递增时是要记录日志的，所以方案改进后性能有大幅度提升。

这个方案虽然使用了 Sequence，但也只能保证全局唯一，数据表中最终保存的主键不再是单调递增的了。

因为，几乎所有数据库中的自增字段或者自增序列都是要记录日志的，也就都会产生磁盘 I/O，也就都会面临这个性能瓶颈的问题。所以，我们可以得出一个结论：在一个海量并发场景下，即使借助单体数据库的自增主键特性，也不能实现单调递增的主键。

自增主键的问题

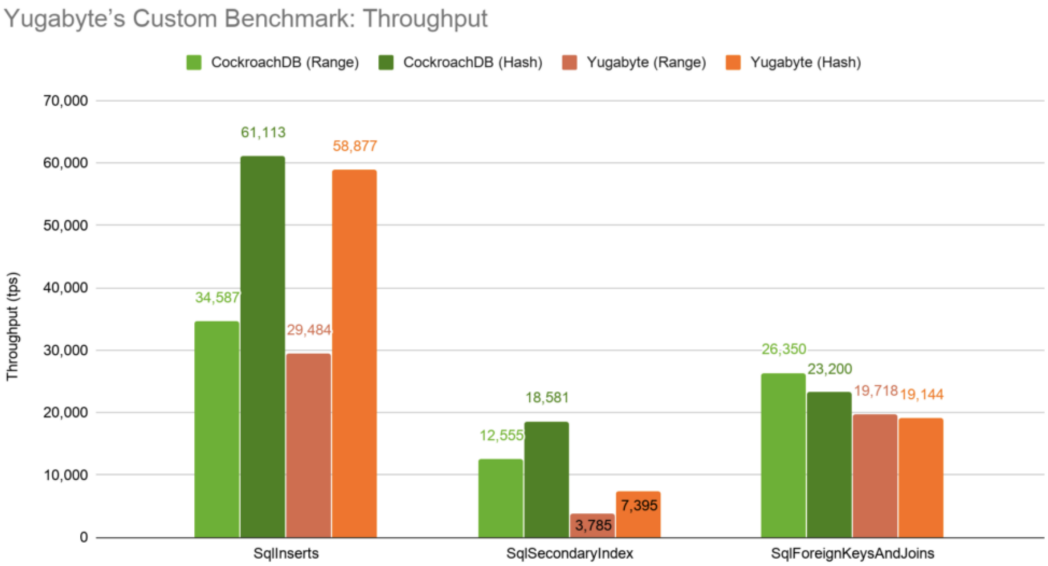
对于分布式数据库，自增主键带来的麻烦就更大了。具体来说是两个问题，一是在自增主键的产生环节，二是在自增主键的使用环节。

首先，产生自增主键难点就在单调递增。如果你已经学习过第 5 讲就会发现，单调递增这个要求和全局时钟中的 TSO 是很相似的。你现在已经知道，TSO 实现起来比较复杂，也容易成为系统的瓶颈，如果再用作主键的发生器，显然不大合适。

其次，使用单调递增的主键，也会给分布式数据库的写入带来问题。这个问题是在 Range 分片下发生的，我们通常将这个问题称为 “尾部热点” 。

尾部热点

我们先通过一组性能测试数据来看看尾部热点问题的现象，这些数据和图表来自 CockroachDB 官网。



这本身是一个 CockraochDB 与 YugabyteDB 的对比测试。测试环境使用亚马逊跨机房的三节点集群，执行 SQL insert 操作时，YugabyteDB 的 TPS 达到 58,877，而 CockroachDB 的 TPS 是 34,587。YugabyteDB 集群三个节点上的 CPU 都得到了充分使用，而 CockroachDB 集群中负载主要集中在一个节点上，另外两个节点的 CPU 多数情况都处于空闲状态。

为什么 CockroachDB 的节点负载这么不均衡呢？这是由于 CockroachDB 默认设置为 Range 分片，而测试程序的生成主键是单调递增的，所以新写入的数据往往集中在一个 Range 范围内，而 Range 又是数据调度的最小单位，只能存在于单节点，那么这时集群就退化成单机的写入性能，不能充分利用分布式读写的扩展优势了。当所有写操作都集中在集群的一个节点时，就出现了我们常说的数据访问热点（Hotspot）。

图中也体现了 CockroachDB 改为 Hash 分片时的情况，因为数据被分散到多个 Range，所以 TPS 一下提升到 61,113，性能达到原来的 1.77 倍。

现在性能问题的根因已经找到了，就是同时使用自增主键和 Range 分片。在第 6 讲我们已经介绍过了 Range 分片很多优势，这使得 Range 分片成为一个不能轻易放弃的选择。于是，主流产品的默认方案是保持 Range 分片，放弃自增主键，转而用随机主键来代替。

随机主键方案

随机主键的产生方式可以分为数据库内置和应用外置两种方式。当然对于应用开发者来说，内置方式使用起来会更加简便。

内置 UUID

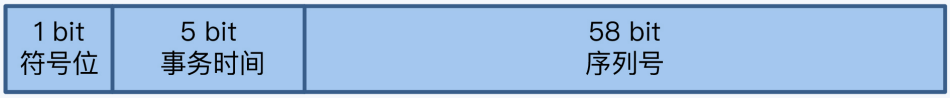
UUID（Universally Unique Identifier）可能是最经常使用的一种唯一 ID 算法，CockroachDB 也建议使用 UUID 作为主键，并且内置了同名的数据类型和函数。UUID 是由 32 个的 16 进制数字组成，所以每个 UUID 的长度是 128 位（16^32 = 2^128）。UUID 作为一种广泛使用标准，有多个实现版本，影响它的因素包括时间、网卡 MAC 地址、自定义 Namesapce 等等。

但是，UUID 的缺点很明显，那就是键值长度过长，达到了 128 位，因此存储和计算的代价都会增加。

内置 Radom ID

TiDB 默认是支持自增主键的，对未声明主键的表，会提供了一个隐式主键 _tidb_rowid，因为这个主键大体上是单调递增的，所以也会出现我们前面说的“尾部热点”问题。

TiDB 也提供了 UUID 函数，而且在 4.0 版本中还提供了另一种解决方案 AutoRandom。TiDB 模仿 MySQL 的 AutoIncrement，提供了 AutoRandom 关键字用于生成一个随机 ID 填充指定列。



这个随机 ID 是一个 64 位整型，分为三个部分。

第一部分的符号位没有实际作用。

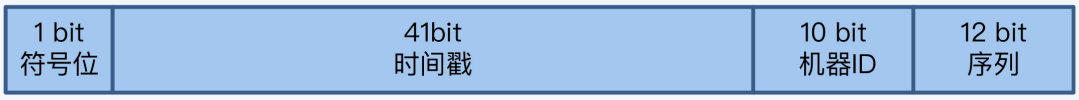
第二部分是事务开始时间，默认为 5 位，可以理解为事务时间戳的一种映射。

第三部分则是自增的序列号, 使用其余位。

AutoRandom 可以保证表内主键唯一，用户也不需要关注分片情况。

外置 Snowflake

雪花算法（Snowflake）是 Twitter 公司分布式项目采用的 ID 生成算法。



这个算法生成的 ID 是一个 64 位的长整型，由四个部分构成：

第一部分是 1 位的符号位，并没有实际用处，主要为了兼容长整型的格式。

第二部分是 41 位的时间戳用来记录本地的毫秒时间。

第三部分是机器 ID，这里说的机器就是生成 ID 的节点，用 10 位长度给机器做编码，那意味着最大规模可以达到 1024 个节点（2^10）。

最后是 12 位序列，序列的长度直接决定了一个节点 1 毫秒能够产生的 ID 数量，12 位就是 4096（2^12）。

这样，根据数据结构推算，雪花算法支持的 TPS 可以达到 419 万左右（2^22*1000），我相信对于绝大多数系统来说是足够了。

但实现雪花算法时，有个小问题往往被忽略，那就是要注意时间回拨带来的影响。机器时钟如果出现回拨，产生的 ID 就有可能重复，这需要在算法中特殊处理一下。

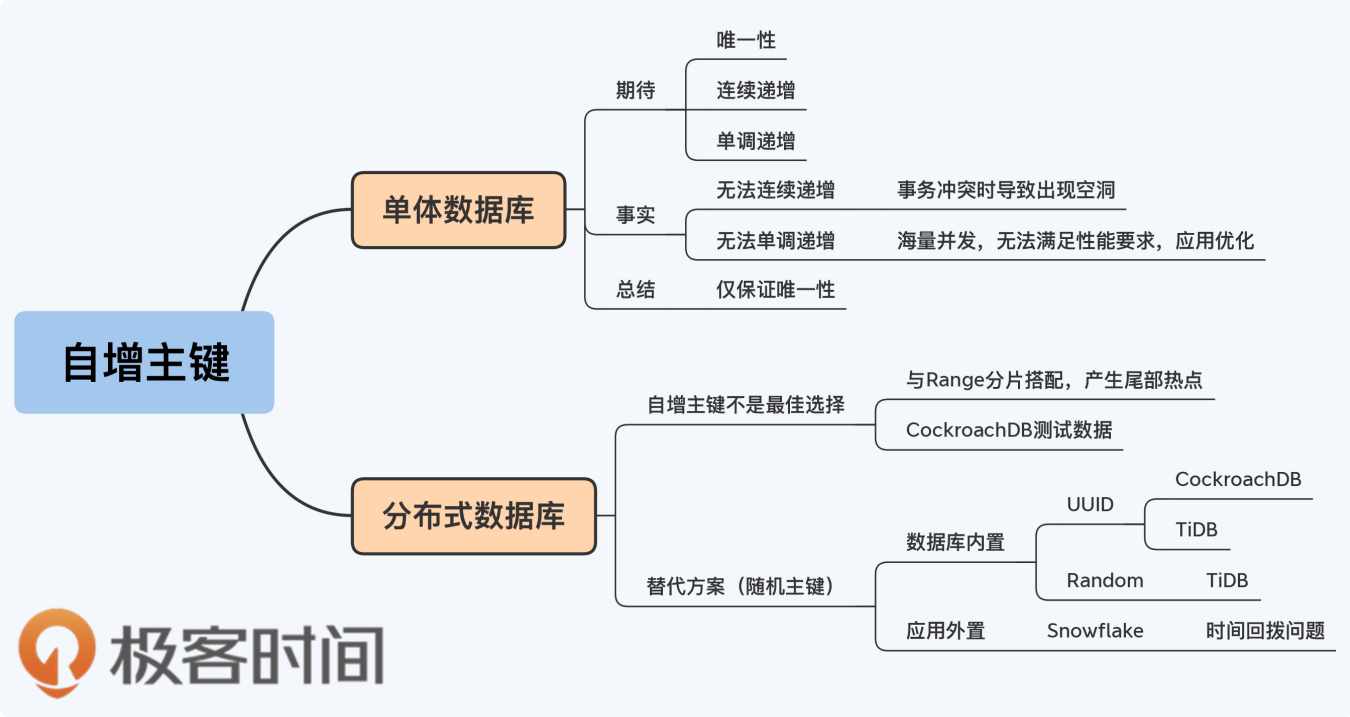
小结

那么，今天的课程就到这里了，让我们梳理一下这一讲的要点。

1. 单体数据库普遍提供了自增主键或序列等方式，自动产生主键。单体数据库的自增主键保证主键唯一、单调递增，但在发生事务冲突时，并不能做到连续递增。在海量并发场景下，通常不能直接使用数据库的自增主键，因为它的性能不能满足要求。解决方式是应用系统进行优化，有数据库控制高位，应用系统控制低位，提升性能。但使用这种方案，主键不再是单调递增的。
2. 分布式数据库在产生自增主键和使用自增主键两方面都有问题。生成自增主键时，要做到绝对的单调递增，其复杂度等同于 TSO 全局时钟，而且存在性能上限。使用自增主键

时，会导致写入数据集中在单个节点，出现“尾部热点”问题。

3. 由于自增主键的问题，有的分布式数据库，如 CockroachDB 更推荐使用随机主键的方式。随机主键的产生机制可以分为数据库内置和应用系统外置两种思路。内置的技术方案，我们介绍了 CockraochDB 的 UUID 和 TiDB 的 RadomID。外置技术方案，我们介绍了 Snowflake。



思考题

课程的最后，我们来看看今天的思考题。我们说如果分布式数据库使用 Range 分片的情况下，单调递增的主键会造成写入压力集中在单个节点上，出现“尾部热点”问题。因此，很多产品都用随机主键替换自增主键，分散写入热点。我的问题就是，你觉得使用随机主键是不是一定能避免出现“热点”问题呢？

欢迎你在评论区留言和我一起讨论，我会在答疑篇和你继续讨论这个问题。如果你身边的朋友也对分布式架构下如何设计主键这个话题感兴趣，你也可以把今天这一讲分享给他，我们一起讨论。

学习资料

CockroachDB: *Yugabyte vs CockroachDB: Unpacking Competitive Benchmark Claims*

提建议

上一篇 16 | 为什么不建议你使用存储过程?

下一篇 18 | HTAP是不是赢者通吃的游戏?

精选留言 (6)

写留言

Jxin
2020-09-16

课后题
1.第一次听说尾部热点，长见识。
2.不好说一定能避免出现“热点”。首先，随机主键替换自增主键，确实能分散写入热点。但如果这个写入"热点"超过db分配集群的容量,那么再怎么分散也没有意义。其次，既

然是随机，那么脸也很重要，非酋手全落到一个rang分片内，那么热点还是会出现。...

展开

2

真名不叫黄金

2020-09-17

随机主键也不一定能避免热点，因为索引也可能有热点:
1. 索引的列值可能是单调递增的，比如以(created_at)作为索引，那么这个索引的写入也会有尾部热点
2. 索引值的基数分布不均匀，比如以(user_id)作为索引，但是恰巧他是个大客户，数据库中20%都是同一个user_id的数据，那么也会有热点。

展开

Geek_0c1732

2020-09-17

oceanbase的自增字段只能保证在一个分区内的单调递增就是为了这个原因吧！印象中oceanbase好像不能使用自增字段做主键

展开

平风造雨

2020-09-16

随机主键的如果是64位的Long，再使用Range分区的情况下，某段时间内某个分区依然还是热点吧?

游弋云端

2020-09-16

个人认为自增主键本身在单体数据库中不是一个良好的设计，应该定义自己的主键或者流水号规则。分布式系统中，需要一个流水号分配中心，类似于Oracle的解决方案，分配一个号段先持久化，然后对外发放，异常后+X来避免重复分配，保障流水号唯一。

展开

左岸

2020-09-16

所以结论就是分布式下，做到全局唯一和趋势递增更简单，不要想着单调递增?