23 | 数据库中间件：传统数据库向分布式数据库的过渡

上一讲我们讨论了传统单机数据库向分布式数据库的转型尝试。今天这一讲，我们就来聊聊传统数据库构造为分布式数据库的帮手，同时也是分布式数据库演化的重要一环：数据库中间件。这里说的中间件一般是具有分片功能的数据库中间层。

关系型数据库本身比较容易成为系统性能瓶颈，单机存储容量、连接数、处理能力等都很有限，数据库本身的“有状态性”导致了它并不像 Web 和应用服务器那么容易扩展。在互联网行业海量数据和高并发访问的考验下，应用服务技术人员提出了分片技术（或称为 Sharding、分库分表）。同时，流行的分布式系统数据库，特别是我们上一讲介绍的从传统数据库过渡而来的分布式数据库，本身都友好地支持 Sharding，其原理和思想都是大同小异的。

成功的数据库中间件除了支持分片外，还需要全局唯一主键、跨分片查询、分布式事务等功能的支持，才能在分片场景下保障数据是可用的。下面我就为你一一介绍这些技术。

全局唯一主键

在单机数据库中，我们往往直接使用数据库自增特性来生成主键 ID，这样确实比较简单。而在分库分表的环境中，数据分布在不同的分片上，不能再借助数据库自增长特性直接生成，否则会造成不同分片上的数据表主键重复。

下面我简单介绍下使用和了解过的几种 ID 生成算法：

1. Twitter 的 Snowflake（又名“雪花算法”）
2. UUID/GUID（一般应用程序和数据库均支持）
3. MongoDB ObjectID（类似 UUID 的方式）

其中，Twitter 的 Snowflake 算法是我近几年在分布式系统项目中使用最多的，未发现重复或并发的问题。该算法生成的是 64 位唯一 ID（由 41 位的 timestamp + 10 位自定义的机器码 + 13 位累加计数器组成）。我在“03 | 数据分片：如何存储超大规模的数据”中介绍过 ShardingShpere 实现 Snowflake 的细节，你可以再回顾一下。

那么解决了全局唯一主键，我们就可以对数据进行分片了。下面为你介绍常用的分片策略。

分片策略

我介绍过的分片模式有：范围分片和哈希分片。

当需要使用分片字段进行范围查找时，范围分片可以快速定位分片进行高效查询，大多数情况下可以有效避免跨分片查询的问题。后期如果想对整个分片集群扩容时，只需要添加节点即可，无须对其他分片的数据进行迁移。

但是，范围分片也有可能存在数据热点的问题，有些节点可能会被频繁查询，压力较大，热数据节点就成了整个集群的瓶颈。而有些节点可能存的是历史数据，很少需要被查询到。

哈希分片我们采用 Hash 函数取模的方式进行分片拆分。哈希分片的数据相对比较均匀，不容易出现热点和并发访问的瓶颈。

但是，后期分片集群扩容起来需要迁移旧的数据。使用一致性 Hash 算法能够很大程度地避免这个问题，所以很多中间件的分片集群都会采用一致性 Hash 算法。离散分片也很容易面临跨分片查询的复杂问题。

很少有项目会在初期就开始考虑分片设计的，一般都是在业务高速发展面临性能和存储的瓶颈时才会提前准备。因此，不可避免地就需要考虑历史数据迁移的问题。一般做法就是通过程序先读出历史数据，然后按照指定的分片规则再将数据写入到各个分片节点中。我们介绍过 ShardingShpere 的弹性伸缩正是解决这个问题的有力武器。

此外，我们需要根据当前的数据量和 QPS 等进行容量规划，综合成本因素，推算出大概需要多少分片（一般建议单个分片上的单表数据量不要超过 1000W）。

数据分散到不同的数据库、不同的数据表上，此时如果查询跨越多个分片，必然会带来一些麻烦。下面我将介绍几种针对分片查询不同的策略。

跨分片查询

中间件跨分片查询，本质上讲原本由数据库承担的数据聚合过程转变到了中间件层。而下面介绍的几种方案，其原理都来源于存储引擎层面。

分页查询

一般来讲，分页时需要按照指定字段进行排序。当排序字段就是分片字段的时候，我们通过分片规则可以比较容易定位到指定的分片，而当排序字段非分片字段的时候，情况就会变得比较复杂了。为了最终结果的准确性，我们需要在不同的分片节点中将数据进行排序并返回，并将不同分片返回的结果集进行汇总和再次排序，最后再返回给用户。

在分布式的场景中，将“LIMIT 10000000，10”改写为“LIMIT 0，10000010”，才能保证其数据的正确性。为什么这样呢？你可以仔细想想。结果就是此种模式会将大量无用数据加载到内存中，从而给内存带来极大的压力。一般解决的手段是避免使用 LIMIT 关键字，而是直接用如下的模式。

```
SELECT * FROM t_order WHERE id > 100000 AND id <= 100010 ORDER BY id;
```

而在翻页时，通过记录上一页最后一条数据的位置，从而减少数据的加载量。

聚合函数

在使用 Max、Min、Sum、Count 和 Avg 之类的函数进行统计和计算的时候，需要先在每个分片数据源上执行相应的函数处理，然后再将各个结果集进行二次处理，最终再将处理结果返回。这里要注意 Avg 函数的实现比较特殊，需要借助 Sum 和 Count 两个函数的实现逻辑进行配合。

跨分片 Join

Join 是关系型数据库中最常用的特性，但是在分片集群中，Join 也变得非常复杂，我们应该尽量避免跨分片的 Join查询（这种场景比上面的跨分片分页更加复杂，而且对性能的影响很大）。

通常有以下两种方式来对其进行优化。

1. 全局表。全局表的基本思想就是把一些类似数据字典又可能会产生 Join 查询的表信息放到各分片中，从而避免跨分片的 Join。
2. ER 分片。在关系型数据库中，表之间往往存在一些关联的关系。如果我们可以先确定好关联关系，并将那些存在关联关系的表记录存放在同一个分片上，那么就能很好地避免跨分片 Join 问题。在一对多关系的情况下，我们通常会选择按照数据较多的那一方进行拆分。

以上就是分布式中间件实现跨分片查询的一些细节。下面我要为你介绍的是中间件面临的最大的挑战——分布式事务。

分布式事务

此处的分布式事务与上一讲的传统数据库发展而来的分布式数据库面临的困难是类似的。那就是，中间件只能与数据库节点进行交互，而无法影响底层数据结构。从而只能从比较高的层次去解决问题，所以下面要介绍的众多方案都有各自的缺点。

客户端一阶段

这是通过客户端发起的一种事务方案，它去掉了两阶段中的 Prepare 过程。典型的实现为：在一个业务线程中，遍历所有的数据库连接，依次做 Commit 或者 Rollback。这种方案对数据库有一种假设，那就是底层数据库事务是做“前向检测”（模块二事务）的，也就是 SQL 执行阶段就可以发现冲突。在客户端进行 Commit 时，失败的概率是非常低的，从而可以推断事务整体失败概率很低。阅文集团早期采用该方案，SLA 可达两个 9。

这种方案相比下面介绍的其他方案来说，性能损耗低，但在事务提交的执行过程中，若出现网络故障、数据库宕机等预期之外的异常现象，将会造成数据不一致，且无法进行回滚。

XA 两阶段

二阶段提交是 XA 的标准实现。让我们复习一下两阶段提交。它将分布式事务的提交拆分为两个阶段：Prepare 和 Commit/Rollback。

开启 XA 全局事务后，所有子事务会按照本地默认的隔离级别锁定资源，并记录 undo 和 redo 日志，然后由 TM 发起 Prepare 投票，询问所有的子事务是否可以进行提交。当所有子事务反馈的结果为“Yes”时，TM 再发起 Commit；若其中任何一个子事务反馈的结果为“No”，TM 则发起 Rollback；如果在 Prepare 阶段的反馈结果为 Yes，而 Commit 的过程中出现宕机等异常时，则在节点服务重启后，可根据 XA Recover 再次进行 Commit 补偿，以保证数据的一致性。

2PC 模型中，在 Prepare 阶段需要等待所有参与子事务的反馈，因此可能造成数据库资源锁定时间过长，不适合并发高以及子事务生命周期较长的业务场景。

ShardingSphere 支持基于 XA 的强一致性事务解决方案，可以通过 SPI 注入不同的第三方组件作为事务管理器实现 XA 协议，如 Atomikos。

最大努力送达

最大努力送达，是针对客户端一阶段的一种补偿策略。它采用事务表记录所有的事务操作 SQL，如果子事务提交成功，将会删除事务日志；如果执行失败，则会按照配置的重试次数，尝试再次提交，即最大努力地进行提交，尽量保证数据的一致性。这里可以根据不同的业务场景，平衡 C 和 A，采用同步重试或异步重试。这与 TiDB 实现 Percolator 事务中重试的思路有相似之处。

这种策略的优点是无锁定资源时间，性能损耗小。缺点是尝试多次提交失败后，无法回滚，它仅适用于事务最终一定能够成功的业务场景。因此**最大努力送达是通过对事务回滚功能上的妥协，来换取性能的提升**。

TCC

TCC 模型是把锁的粒度完全交给业务处理，它需要每个子事务业务都实现 Try-Confirm/Cancel 接口。

- Try：尝试执行业务。完成所有业务检查，并预留必需业务资源。
- Confirm：
  确认执行业务。真正执行业务，不做任何业务检查。只使用 Try 阶段预留的业务资源。Confirm 操作满足幂等性。
- Cancel：取消执行业务。释放 Try 阶段预留的业务资源。Cancel 操作满足幂等性。

这三个阶段都会按本地事务的方式执行，不同于 XA 的 Prepare，TCC 无须将 XA 投票期间的所有资源挂起，因此极大地提高了吞吐量。但是它的缺点是需要实现 Cancel 操作，这不仅给实现带来了很多麻烦，同时有一些操作是无法 Cancel 的。

Saga

Saga 起源于 1987 年 Hector & Kenneth 发表的论文《Sagas》。

Saga 模型把一个分布式事务拆分为多个本地事务，每个本地事务都有相应的执行模块和补偿模块（TCC 中的 Confirm 和 Cancel）。当 Saga 事务中任意一个本地事务出错时，可以通过调用相关的补偿方法恢复之前的事务，达到事务最终的一致性。

它与 TCC 的差别是，Saga 是以数据库事务维度进行操作的，而 TCC 是以服务维度操作的。

当每个 Saga 子事务“T1，T2，…，Tn”都有对应的补偿定义“C1，C2，…，Cn-1”，那么 Saga 系统可以保证子事务序列“T1，T2，…，Tn”得以完成（最佳情况）或者序列“T1，T2，…，Tj，Cj，…，C2，C1”得以完成，也就是取消了所有的事务操作。

由于 Saga 模型中没有 Prepare 阶段，因此事务间不能保证隔离性，当多个 Saga 事务操作同一资源时，就会产生更新丢失、脏数据读取等问题，这时需要在业务层控制并发，例如：在应用层面加锁、应用层面预先冻结资源。

Saga 支持向前和向后恢复。

- 向后恢复：如果任一子事务失败，补偿所有已完成的事务。
- 向前恢复：假设每个子事务最终都会成功，重试失败的事务。

显然，向前恢复没有必要提供补偿事务，如果你的业务中，子事务最终总会成功，或补偿事务难以定义或不可能，向前恢复会更符合你的需求。理论上补偿事务永不失败，然而，在分布式世界中，服务器可能会宕机、网络可能会失败，甚至数据中心也可能会停电，这时需要提供故障恢复后回退的机制，比如人工干预。

总的来说，**TCC 是以应用服务的层次进行分布式事务的处理，而 XA、Bed、Saga 则是以数据库为层次进行分布式处理，故中间件一般倾向于采用后者来实现更细粒度的控制**。

Apache ShardingShpere 的分布式事务变迁

ShardingShpere 在 3.0 之前实现了客户端一阶段（弱 XA），最大努力送达和 TCC。其中最大努力送达需要配合调度任务异步的执行。而弱 XA 作为默认的实现模式，此种组合是实用性与实现难度之间的平衡，但是在分布式失败模型描述的场景下会产生不一致的问题。

在 3.0 后，团队梳理了事务模型。实现了 XA 两阶段和 Saga。这两种事务都是面向数据库层面的，同时有完整的理论支撑，更加符合现代分布式数据库的设计风格。同时事务模块也如其他模块一样支持 SPI，也就是可以实现第三方的事务模型。而京东 JDTX 事务引擎就是通过 SPI 集成到 ShardingShpere 的。下一讲我会介绍 JDTX 的相关内容。

总结

这一讲我们探讨了实现数据库中间件的几种技术，包括全局唯一主键、分片策略和跨分片查询。其中最重要的就是分布式事务。

不同于分布式数据库，中间件的分布式事务多了很多应用服务的特色，比如客户单一阶段、TCC。它们更偏向于服务层面，从而揭示了中间件大部分是由应用研发或应用架构团队开发迭代的产物。而随着中间件的发展，它们不可避免地向分布式数据演进，如阿里云的 DRDS 和 PolarDB-X 就是由中间件 TDDL 演化而成。

数据库中间件是一个过渡产品，随着近几年技术的发展，越来越多原生 NewSQL 出现在我们面前。下一讲我就为你介绍几种典型的 NewSQL 数据库，看看它们都具备怎样的特点。

希望下一讲准时与你相见，谢谢。