分布式数据库的进度管理：TiDB 备份恢复工具 PiTR 的原理与实践

PingCAP 2024-10-09 54 阅读9分钟

智能总结 复制 重新生成

本文深入解析了 TiDB 备份恢复工具 PiTR 的原理与实践。先介绍 PiTR 对数据恢复的重要性，然后阐述 TiKV 侧的备份流程，包括组件工作与备份操作。接着探讨从单个 Region 到全局的进度管理，引入 Checkpoint、Service Safepoint 和 Global Checkpoint 等概念。最后讲述 TiDB 侧负责进度管理的组件 CheckpointAdvancer 的工作及异常处理机制。

关联问题: PiTR 适用哪些场景 GC 如何影响备份 能否优化异常处理

基于该文章内容继续向AI提问

导读

对于一款企业级数据库产品而言，数据的安全性和可恢复性是至关重要的。PiTR（Point in Time Restore）作为 TiDB 备份工具的核心功能之一，提供了一种精细的数据恢复能力，允许用户将数据库集群恢复到过去的任意时间点。这种能力对于处理数据损坏、误操作或数据丢失等灾难性事件至关重要。

对于分布式系统而言，想实现精确的进度管理是十分复杂的，本文将深入解析 PiTR 在 TiDB 的分布式架构中的实现，包括其在 TiKV 层的备份流程，以及 TiDB 如何管理这些备份任务的进度。

希望本文能够帮助开发者和数据库管理员更好地理解 PiTR 的工作机制，有效地利用这一功能加固数据库基础设施。

PiTR 是 TiDB 备份工具中必不可少的一部分。如果说全量备份帮助我们获得了将集群回退到某个时间节点的能力，那么 PiTR 则更加精细地备份了集群的每一次写入，并且允许我们回到备份开始后的任意一个时点。

直觉上，当你启动一个 PiTR 任务，等于告诉集群：我需要知道从当前时间节点之后的全部变化。对于一个分布式数据库而言，这并不是一个简单的工作。

上图展示了目前 TiDB 的数据存储结构。用户以表和行的形式写入数据，每一行数据都会以一个键值对的形式存储在 TiKV 中，每一个 TiKV 又会被逻辑地划分为多个 region。

由于 TiDB 分布式写入的实质，各个 Region 的数据分布在不同宿主机上，也不存在一个确切统一的写入时点。所以我们需要找到一种方法分别管理每个 region 的写入工作，并且需要提供一个整体进度。在接下来的内容中，我们将详细展开 TiDB 的 PiTR 进度管理流程。从单个 TiKV 开始，逐步推进到整个集群。

TiKV 侧备份流程

如果我们希望管理备份工作的具体进度，首先需要了解的是，备份工作究竟是怎样完成的。在 TiDB 的实践中，PiTR 是一个分布式过程，每个 TiKV Server 自行记录备份数据，并将数据发送到远端储存，大致上按照下图所示的流程工作。

PingCAP LV.5
@http://pingcap.com
作者榜No.6 优秀作者
788 文章 548k 阅读 7.6k 粉丝
关注 私信

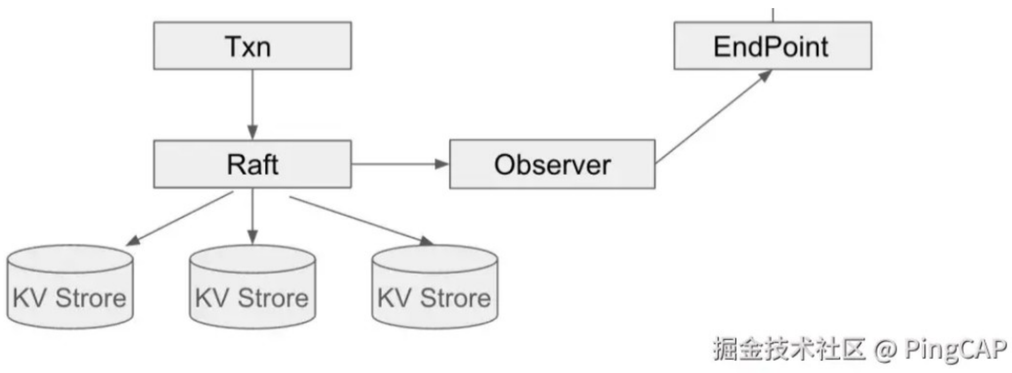
目录 收起

相关推荐

坚如磐石：TiDB 基于时间点的恢复（Pi...
1.6k阅读 · 7点赞
直击备份恢复的痛点：基于 TiDB Binlog...
1.6k阅读 · 0点赞
TiDB Hackathon 2021 — pCloud : 做数...
162阅读 · 0点赞
数据库误操作后悔药来了：AnalyticDB ...
534阅读 · 1点赞
TiDB Serverless 正式商用，全托管的云...
171阅读 · 0点赞

精选内容

Go语言中如何优雅实现单例模式
烛阴 · 78阅读 · 1点赞
Python计算TRC20地址
喵个咪 · 37阅读 · 0点赞
都2025年了，你还只懂用mybatis来做...
Y11_推特同名 · 304阅读 · 4点赞
十一. Redis 主从复制—>实现负载均衡(...
RainbowSea · 40阅读 · 1点赞
十. Redis 事务和 “锁机制”——> 并发秒...
RainbowSea · 48阅读 · 2点赞



在 TiKV server 初始化期间，会同时（先后）初始化 BackupStreamObserver[1] 和 Endpoint[2] 两个组件。它们共用了同一个 scheduler(backup_stream_scheduler[3]，通过向 scheduler 发送 Task 的方式进行互相沟通。

BackupStreamObserver 会实时监听 Raft 状态机的写入情况。其重点在于 on_flush_applied_cmd_batch()[4] 接口。这个接口会在 Raft 状态机 apply 时被调用，将 Raft 命令打包为 BatchEvent，然后作为一个任务发送给 scheduler。对于 PiTR 而言，这个任务被称为 Task::BatchEvent[5]。

```rust
pub struct CmdBatch {
    pub level: ObserveLevel,
    pub cdc_id: ObserveId,
    pub rts_id: ObserveId,
    pub pitr_id: ObserveId,
    pub region_id: u64,
    pub cmds: Vec<Cmd>,
}
```

可以看出，BatchEvent 的实质是一系列 Raft 命令的拷贝。PiTR 在备份时记录这些命令，并在恢复时重放，以实现日志备份功能。

而 Endpoint 负责沟通 TiKV Server 和外部储存。它会在启动之后进入一个循环，检查当前 scheduler 中是否包含新的任务，匹配并执行不同的函数。其中，我们需要关注的是 Task::BatchEvent，也就是从 Observer 发送来的写入数据。当 endpoint 匹配到 Task::BatchEvent，它会执行 backup_batch()[6] 函数开始备份这些键值对。

在这一步，Endpoint 先对 CmdBatch 进行简单检查，然后将它发往router.on_events()[7]，并开始异步地等待结果。

Router 的作用是将写入操作按照 range 拆分，以提高并发度。每个 range 的写入并不是即时的，我们会在内存中储存一个临时文件，用于暂时存储从 raft store 更新的信息。当内存中储存的临时文件大小超出上限，或者超过指定刷盘间隔，我们才会真正将储存在临时文件中的数据写入远端储存，并视为完成了一次（部份）备份。目前 BackupStreamConfig 的默认设置中，max_flush_interval 为 3 分钟。

```rust
impl Default for BackupStreamConfig {
    fn default() -> Self {
        // ...
        Self {
            min_ts_interval: ReadableDuration::secs(10),
            max_flush_interval: ReadableDuration::minutes(3),
        // ...
        }
    }
}
```

当满足刷盘条件后，我们会跳转到 endpoint.do_flush() [8] 函数。并在这里完成将备份文件刷盘的逻辑。当这个函数完成之后，备份数据已经被写入远端存储，可以认为备份到此告一段落。此处正是汇报备份进度的最佳时刻。在并不令人注意的角落，这个任务是由一个回调完成的：flush_ob.after() [9]。

```rust
        async fn after(&mut self, task: &str, _rts: u64) -> Result<()> {
         let flush_task = Task::RegionCheckpointsOp(RegionCheckpointOperation::FlushWith(
             std::mem::take(&mut self.checkpoints),
         )); //Update checkpoint
         try_send!(self.sched, flush_task);

         let global_checkpoint = self.get_checkpoint(task).await?;
         info!("getting global checkpoint from cache for updating."; "checkpoint" => ?global_checkpo
         self.baseline
             .after(task, global_checkpoint.ts.into_inner()) //update safepoint
             .await?;
         Ok(())
     }
```

这个回调函数做了两件事，更新 service safe point 和 store checkpoint。它们是什么，又有什么用呢?

从检查点（Checkpoint）到全局检查点（Global Checkpoint）

上文中我们阅读了 PiTR 备份流程的细节。现在，我们可以回到正题，反思整个流程。

首先我们已经明确，对于 TiDB 这样的分布式数据库，所有的数据都储存在一个个单独的 TiKV 节点上。在 PiTR 流程中，这些 TiKV 也是各自将数据打包成文件，发送到远端储存上。这引出了一个重要的问题：如何进行进度管理?

为了确保备份进度的有效管理，我们需要跟踪每个 TiKV 节点上的数据备份进度。对于单个 Region，可以通过记录已备份数据的时间戳来实现进度管理：当数据被刷盘时，记录当前时间戳，这个时间戳就是该 Region 完成备份的最小时间节点，即 Checkpoint。

同时，我们需要了解到，需要备份的数据并不会永恒的保留。由于 MVCC 机制，每次数据修改都会产生一个新版本并保留旧版本，旧版本可以用于历史查询和事务隔离。随着时间的推移，这些历史数据会不断累积，因此需要通过 GC 机制来回收和清理旧版本，释放存储空间并提高性能。

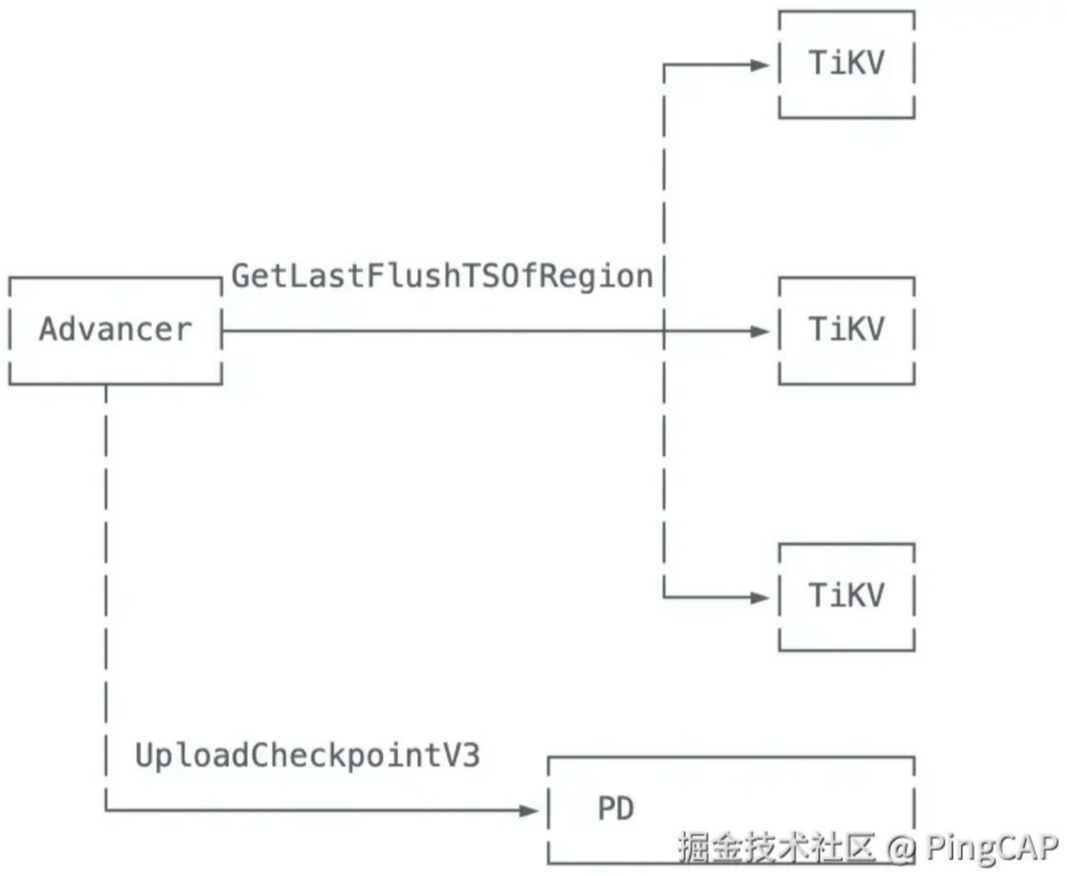
我们需要确保在备份（Flush）完成之前，备份数据不会被 GC 清除。所以此处引入一个指标，通知 GC 可以安全清除的数据时间戳。这就是Service Safepoint。

值得注意的是，以上的讨论只是单个 region 的进度管理，一个集群中会同时存在多个 region，所以我们需要设计一个指标便于管理整个集群的备份进度，它被称之为Global Checkpoint。

在实践中，Global Checkpoint 是所有 TiKV Checkpoint 的最小值[10]，这保证了所有 region 的进度都至少不小于这个时间节点。或者说，在这个时间节点之前，整个集群的数据都完成备份了。

而这个汇总所有 TiKV 进度并计算 Global Checkpoint 的工作，是在 TiDB 完成的。

TiDB 侧进度管理



既然我们了解了 TiKV 侧的备份进度管理流程。让我们转头看看 TiDB 的情况。

在 TiDB 侧，负责这项工作的组件被称为 CheckpointAdvancer [11]。它的本质是一个外挂在 TiDB 主程序上的守护进程，会随着时间执行一些周期性操作。它的工作主要包括两部分：

> 1. 订阅更新来自 TiKV 的 FlushTSO 更新。
> 2. 处理可能的错误并计算 Global Checkpoint。
> 3. 计算总体更新进度并汇报给 PD。

具体地，在 CheckPointAdvancer 中有一个名为 FlushSubscriber[12] 的字段，TiDB 就是通过它监听 TiKV 的刷盘操作和 checkpoint 推进。FlushSubscriber 维持一个 gRPC 流，持续监听[13] 不同 range 的 checkpoint 并将其记录下来。随后通过 channel 发送给 advancer。

而 advacner 接收到这些 checkpoint 之后，会将它们放置于 checkpoints[14] 字段中。当接收到来自 TiKV 的进度信息之后，advancer 会尝试开始更新 Global Checkpoint。作为一个守护进程，更新过程并不是实时的，而是随着主进程调用它的 tick()[15] 方法间歇性完成。

```go
func (c *CheckpointAdvancer) tick(ctx context.Context) error {
        //...
        var errs error
        cx, cancel := context.WithTimeout(ctx, c.Config().TickTimeout())
        defer cancel()
        err := c.optionalTick(cx)
        if err != nil {
            // ...
        }

        err = c.importantTick(ctx)
        if err != nil {
            // ...
        }

        return errs
}
```

这个过程实际上被分为了两个部分，optionalTick()[16] 和 importantTick() [17]。

optionalTick 主要负责与 FlushSubscriber 沟通，获取来自 TiKV 的进度更新。由于单个 TiKV 的 Checkpoint 并不一定会推进，所以取名为 optionalTick。一旦捕获到 TiKV FlushTSO 的更新，便会在这里记录并试图推进全局检查点。

而 importantTick 则负责管理全局进度。确认进度更新后，这里会产生新的 Global Checkpoint 和 Service Safepoint[18]。

这个行为是存在风险的。如果某个 TiKV 的 Checkpoint 因为种种原因一直没有成功推进，就会阻塞住 Global Checkpoint 的推进，进而可能阻塞住 GC，无法正确清除已经完成备份的冗余数据。在最糟糕的情况下，某个 TiKV 陷入了不可自动恢复的错误。它有可能会永远阻碍 GC 进度，造成对整体系统的更大破

坏。

因此，importantTick 会检查[19] checkpoint 距离上次更新的时间差。如果某个 Checkpoint 长时间没有推进，这个备份任务会被标记为异常状态[20]。随后，advancer 会自动暂停这个任务，等待管理员手工运维的介入。

```go
        isLagged, err := c.isCheckpointLagged(ctx)
        if err != nil {
                return errors.Annotate(err, "failed to check timestamp")
        }
        if isLagged {
                err := c.env.PauseTask(ctx, c.task.Name)
                if err != nil {
                        return errors.Annotate(err, "failed to pause task")
                }
                return errors.Annotate(errors.Errorf("check point lagged too large"), "check point
        }
```

此后，advancer 并不会停止，它只是跳过 [21] 了异常任务的 checkpoint 更新。如果 PD 恢复了这个任务，会向 advancer 发送信号[22]，advancer 便可以回到正常的 tick 流程中。

此处介绍的异常处理机制是完全防卫性质的。它只能识别异常状态的存在，却无法指出问题的原因，最终还需要管理员手动介入。或许在未来，我们能够实现 PiTR 的自动运维，当 checkpoint 恢复推进之后，可以自动重启这个任务。

参考资料

[1]

BackupStreamObserver: *github.com/tikv/tikv/b...*

[2]

Endpoint: *github.com/tikv/tikv/b...*

[3]

backup_stream_scheduler: *github.com/tikv/tikv/b...*

[4]

on_flush_applied_cmd_batch(): *github.com/tikv/tikv/b...*

[5]

Task::BatchEvent: *github.com/tikv/tikv/b...*

[6]

backup_batch(): *github.com/tikv/tikv/b...*

[7]

router.on_events(): *github.com/tikv/tikv/b...*

[8]

endpoint.do_flush(): *github.com/tikv/tikv/b...*

[9]

flush_ob.after(): *github.com/tikv/tikv/b...*

[10]

最小值: *github.com/pingcap/tid...*

[11]

CheckpointAdvancer: *github.com/pingcap/tid...*

[12]

FlushSubscriber: *github.com/pingcap/tid...*

[13]

持续监听: *github.com/pingcap/tid...*

[14]

checkpoints: *github.com/pingcap/tid...*

[15]

tick(): *github.com/pingcap/tid...*

[16]

optionalTick(): *github.com/pingcap/tid...*

[17]

importantTick(): *github.com/pingcap/tid...*

[18]

Service Safepoint: *github.com/pingcap/tid...*

[19]

检查: *github.com/pingcap/tid...*

[20]

标记为异常状态: *github.com/pingcap/tid...*

[21]

跳过: *github.com/pingcap/tid...*

[22]

发送信号: *github.com/pingcap/tid...*

标签： 数据库

评论 0

登录 / 注册 即可发布评论!

暂无评论数据

为你推荐

坚如磐石：TiDB 基于时间点的恢复（PiTR）特性优化之路 | 6.5 新特性解析
PingCAP | 1年前 | 1.6k | 7 | 评论 数据库

TiDB简述及TiKV的数据结构与存储 | 京东物流技术团队
京东云开发者 | 1年前 | 1.7k | 10 | 1 TiDB 数据结构 MySQL

TiKV 源码分析之 PointGet
vivo互联网技术 | 7月前 | 1.0k | 1 | 评论 数据库

分布式高可用存储系统--TiDB简介
fjian_fresh | 3年前 | 1.0k | 5 | 评论 Java

一文了解 PG PITR 即时恢复
青云技术社区 | 3年前 | 521 | 1 | 评论 云计算

Spring Boot集成tidb快速入门demo
HBLOG | 8月前 | 798 | 2 | 2 后端 Spring B... Java

分享 | 一文了解 PG PITR 即时恢复
RadonDB | 3年前 | 1.0k | 1 | 评论 数据库 后端

对比传统数据库，TiDB 强在哪？谈谈 TiDB 的适应场景和产品能力
威哥爱编程 | 7月前 | 401 | 2 | 评论 数据库 TiDB MySQL

CloudCanal x TiDB 数据迁移同步功能落地
ClouGence | 1年前 | 456 | 12 | 评论 后端 Java 数据库

TiDB 的列式存储引擎是如何实现的?
PingCAP | 4年前 | 1.1k | 4 | 评论 数据库

架构师必备--分布式数据库Tidb安装部署实践
一起随缘 | 1年前 | 1.9k | 10 | 6 架构 分布式 TiDB

一篇文章彻底搞懂 TiDB 集群各种容量计算方式
PingCAP | 11月前 | 529 | 点赞 | 评论 数据库

基于 TiDB 资源管控 + TiCDC 实现多业务融合容灾测试
PingCAP | 5月前 | 90 | 点赞 | 评论 TiDB


万字长文，深入浅出分布式数据库TiDB架构设计！
零壹技术栈 | 8月前 | 1.2k 14 1
数据库 分布式 TiDB
KubeSphere 部署 TiDB 云原生分布式数据库
PingCAP | 4年前 | 670 3 1
TiDB 后端