

事务执行方法、装置、计算设备及存储介质

本申请要求于 2021 年 10 月 19 日提交的申请号为 202111214946.4、发明名称为“事务执行方法、装置、计算设备及存储介质”的中国专利申请的优先权，其全部内容通过引用结合在本申请中。

技术领域

本申请涉及数据库技术领域，特别涉及一种事务执行方法、装置、计算设备及存储介质。

背景技术

随着数据库技术的发展，SQL（Structured Query Language，结构化查询语言）是一种数据库查询和程序设计语言，用于存取数据以及查询、更新和管理关系数据库系统。在 SQL 命令中涉及一类 DDL（Data Definition Language，数据定义语言）语句，DDL 语句是用于修改数据库中的对象（如表、索引、列、触发器等）的定义的语句。在对数据库中的某个对象执行 DDL 语句时，通常需要对该对象加锁，阻塞在该 DDL 语句执行期间其他业务事务对该对象中数据的修改，以保证 DDL 语句所操作的对象中所存储数据的一致性。

发明内容

本申请实施例提供了一种事务执行方法、装置、计算设备及存储介质。

一方面，提供了一种事务执行方法，该方法包括：响应于执行中断的数据定义语言 DDL 事务符合恢复条件，确定所述 DDL 事务在执行中断前处理完毕的最后一个数据块；从所述最后一个数据块的下一个数据块开始，继续执行所述 DDL 事务；响应于对所述 DDL 事务操作的各个数据块均处理完毕，提交所述 DDL 事务。

一方面，提供了一种事务执行装置，该装置包括：确定模块，用于响应于执行中断的数据定义语言 DDL 事务符合恢复条件，确定所述 DDL 事务在执行中断前处理完毕的最后一个数据块；执行模块，用于从所述最后一个数据块的下一个数据块开始，继续执行所述 DDL 事务；提交模块，用于响应于对所述 DDL 事务操作的各个数据块均处理完毕，提交所述 DDL 事务。

一方面，提供了一种计算设备，该计算设备包括一个或多个处理器和一个或多个存储器，该一个或多个存储器中存储有至少一条计算机程序，该至少一条计算机程序由该一个或多个处理器加载并执行以实现如上述事务执行方法。

一方面，提供了一种存储介质，该存储介质中存储有至少一条计算机程序，该至少一条计算机程序由处理器加载并执行以实现如上述事务执行方法。

一方面，提供一种计算机程序产品或计算机程序，所述计算机程序产品或所述计算机程序包括一条或多条程序代码，所述一条或多条程序代码存储在计算机可读存储介质中。计算设备的一个或多个处理器能够从计算机可读存储介质中读取所述一条或多条程序代码，所述一个或多个处理器执行所述一条或多条程序代码，使得计算设备能够执行上述事务执行方法。

附图说明

图 1 是本申请实施例提供的一种事务执行方法的实施环境示意图；

图 2 是本申请实施例提供的一种事务执行方法的流程图；

图 3 是本申请实施例提供的一种事务执行方法的流程图；

图 4 是本申请实施例提供的一种事务执行方法的流程图；

图 5 是本申请实施例提供的一种在线增加索引的正常执行流程图；

图 6 是本申请实施例提供的一种在线增加索引的异常中断后恢复流程图；

图 7 是本申请实施例提供的一种 block 分裂的原理性示意图；
图 8 是本申请实施例提供的一种 block 分裂的原理性示意图；
图 9 是本申请实施例提供的一种离线增加索引的正常执行流程图；
图 10 是本申请实施例提供的一种离线增加索引的异常中断后恢复流程图；
图 11 是本申请实施例提供的一种在线增加索引的正常并行执行流程图；
图 12 是本申请实施例提供的一种在线增加索引的异常中断后并行恢复流程图；
图 13 是本申请实施例提供的一种事务执行装置的结构示意图；
图 14 是本申请实施例提供的一种终端的结构示意图；
图 15 是本申请实施例提供的一种计算设备的结构示意图。

具体实施方式

为使本申请的目的、技术方案和优点更加清楚，下面将结合附图对本申请实施方式作进一步地详细描述。

本申请中术语“第一”“第二”等字样用于对作用和功能基本相同的相同项或相似项进行区分，应理解，“第一”、“第二”、“第 n”之间不具有逻辑或时序上的依赖关系，也不对数量和执行顺序进行限定。

本申请中术语“至少一个”是指一个或多个，“多个”的含义是指两个或两个以上，例如，多个第一位置是指两个或两个以上的第一位置。

在介绍本申请实施例之前，需要引入一些云技术领域内的基本概念。

云技术（Cloud Technology）：是指在广域网或局域网内将硬件、软件、网络等系列资源统一起来，实现数据的计算、储存、处理和共享的一种托管技术，也即是基于云计算商业模式应用的网络技术、信息技术、整合技术、管理平台技术、应用技术等的总称，通过组成资源池，按需所用，灵活便利。云计算技术将变成云技术领域的重要支撑。技术网络系统的后台服务需要大量的计算、存储资源，如视频网站、图片类网站或更多的门户网站。伴随着互联网行业的高度发展和应用，将来每个物品都有可能存在自己的识别标志，都需要传输到后台系统进行逻辑处理，不同程度级别的数据将会分开处理，各类行业数据皆需要强大的系统后盾支撑，均能通过云计算来实现。

云存储（Cloud Storage）：是在云计算概念上延伸和发展出来的一个新的概念，分布式云存储系统（以下简称存储系统）是指通过集群应用、网格技术以及分布存储文件系统等功能，将网络中大量各种不同类型的存储设备（存储设备也称之为存储节点）通过应用软件或应用接口集合起来协同工作，共同对外提供数据存储和业务访问功能的一个存储系统。

数据库（Database）：简而言之能够视为一种电子化的文件柜——存储电子文件的处所，支持用户对文件中的数据进行新增、查询、更新、删除等操作。所谓“数据库”是以一定方式储存在一起、能与多个用户共享、具有尽可能小的冗余度、与应用程序彼此独立的数据集合。

以下，对本申请实施例所涉及的术语进行解释说明。

DDL（Data Definition Language，数据定义语言）语句：即 DDL 语句操作，指用来修改数据库中的对象（如：表、索引、列、触发器等）的定义的语句。

Online DDL（在线 DDL）：在线 DDL 是相对于离线 DDL 来说的，通常对数据库中的某个对象执行一个 DDL 语句时，都会对该对象加上锁，再执行变更操作。上述加锁是用来阻塞在该 DDL 语句执行期间其他业务事务（如 DML 事务）对该对象中数据的修改，以保证 DDL 语句所操作的对象中所存储数据的一致性，其中 DML 是指 Data Manipulate Language（数据操纵语言）。换言之，在 DDL 语句执行期间，用户发起的业务操作（业务事务）会因为该对象被锁定而阻塞。随着用户对于业务的高可用要求越来越高，各家数据库厂商都提出在线 DDL 的实现，即在对某个对象做 DDL 变更的时候，实现对于用户业务不阻塞或者只阻塞一个很小的时间段。

Thomas Write (托马斯写): 在计算机科学领域, 特别是数据库领域, 托马斯写是一种基于时间戳的并发控制机制, 被总结为忽略掉已经过时的写。例如: 事务 1 在 T1 时刻开始修改数据 A, 事务 2 在 T2 时刻开始也修改数据 A, $T1 < T2$, 因为某种原因事务 1 和事务 2 同时向数据库提出提交申请, 在乐观事务提交机制下, 数据库会选择先达到的提交申请的事务进行提交, 其他冲突事务回滚, 但是在托马斯写规则下, 则数据库需要确保事务 2 的提交一定会成功, 因为事务 1 的时间戳在托马斯写规则下被认为是已经过时的写, 发生冲突的情况下需要被回滚掉。需要说明的是, 悲观事务模型下会因为要修改的某个数据已经被锁定, 导致其他事务等待, 因此不会出现同时有两个事务同时修改同一数据, 并同时提交的情况。

Tuple (数据记录): 通常是指关系型数据库中的数据表中的某一行数据记录, 这个数据记录存储了表定义中所有列的实例化信息, 并按照列定义的顺序排列, 组成一个连续的内容, 也即是说, 这段连续的内容被称为数据表的一条数据记录, 即 Tuple。

数据字典: 数据库中用来存放数据库中的对象的定义的地方。其中, 数据库中的对象包括: 数据库、表空间、数据表、数据列、数据索引、分区信息、用户、函数、角色等。

并行度: 并行处理中将会启动多少并行处理任务来将原来需要处理的任务进行分解; 通常在计算机处理中, 就是将原来需要处理的全量数据, 划分为若干子任务, 启动并行度指定数量的线程, 取得子任务, 并按照相应的处理逻辑进行处理, 理想情况下并行执行能够达到并行度倍数的性能提升。

以下, 对传统数据库中 DDL 语句的离线执行模式进行说明。

对于传统单机数据库系统, 通过对元数据加锁的方式来阻塞有冲突的操作的执行, 保证 Schema (数据库对象的集合) 变更的正确执行, 即保证 DDL 语句的正确执行。

对于传统集群数据库系统, 集群数据库系统作为单机数据库系统的一个扩展, 集群内包括多个计算节点 (即计算设备), 每个计算节点的数据库中存储有多个数据表, 每个数据表中存储有一条或多条数据记录 (即数据行), 每一条数据记录由一组按照相同位置索引排列的字段集组成, 即数据字段列, 计算节点的数据库为任一类型的集群数据库, 包括关系型数据库或者非关系型数据库中至少一项, 例如 SQL 数据库、MySQL、NoSQL、NewSQL (泛指各种新式的可拓展/高性能数据库) 等, 在本申请实施例中对数据库的类型不作具体限定。

针对上述集群数据库系统, DDL 语句的执行模式划分为 Share Memory (共享内存) 模式和 Share Nothing (零共享) 模式:

A) Share Memory (共享内存) 模式, 通过在集群内的一个中央协调节点来控制各个计算节点对于某一个 Schema 对象的访问控制逻辑;

B) Share Nothing (零共享) 模式, 通常所有计算节点都有一个控制节点来确定哪些计算节点需要参与 DDL 语句的执行, 然后 DDL 语句再分发到相应的计算节点执行, 在每个计算节点中视为在一个单机数据库系统上执行, 因此与单机数据库基本同理。

对于计算存储分离的云原生数据库架构, 目前都只实现了单写, 与 MySQL 单实例一致, 在实现多写集群方案中, 需要在多个写节点同步 DDL 锁以保证数据在 DDL 变更前后以及变更过程中的数据一致性。

综上所述, 不管是传统单机数据库、传统集群分布式数据库还是计算存储分离的数据库架构, 如果在 DDL 语句执行期间, 由于某种故障导致 DDL 语句的执行线程崩溃, 在 DDL 语句异常中断后, 均无法提供一种在中断处继续执行的能力, 这是因为要么 DDL 语句会被回滚掉, 要么通过重做日志 (Redo Log) 回滚到 DDL 语句开始时的状态后, 从头开始对该 DDL 语句进行重做, 针对 DDL 语句操作的对象所存储的数据量较大 (例如存储了海量数据的表) 的情况, 数据库系统重做该 DDL 语句的代价较高, 换言之, DDL 事务的成功率低, 数据库资源利用率低。

有鉴于此, 本申请实施例提供一种事务执行方法, 应用于传统数据库、分布式数据库等各类数据库系统, 针对 DDL 事务提供在异常中断点继续恢复执行的方案, 在 DDL 事务执行发生异常中断后, 后台线程或进程能够在上次中断位置继续完成 DDL 的实现, 从而对于操作

比较大的对象（例如存储了海量数据的表）的 DDL 事务，能够提高 DDL 事务的成功率，并且由于能够保留异常中断之前已经完成的部分处理结果，使得整体的数据库资源利用率得到了提升。

在一些实施例中，本申请实施例还应用于一种基于区块链技术的数据库系统（以下简称为“区块链系统”），区块链系统在本质上属于一种去中心化式的分布式数据库系统，采用共识算法保持区块链上不同计算设备所记载的账本数据一致，通过密码算法保证不同计算设备之间账本数据的加密传送以及不可篡改，通过脚本系统来拓展账本功能，通过网络路由来进行不同计算设备之间的相互连接。

在区块链系统中包括一条或多条区块链，区块链是一串使用密码学方法相关联产生的数据块，每一个数据块中包含了一批次网络交易的信息，用于验证其信息的有效性（防伪）和生成下一个区块。

区块链系统中计算设备之间组成点对点（Peer To Peer，P2P）网络，P2P 协议是一个运行在传输控制协议（Transmission Control Protocol，TCP）协议之上的应用层协议。在区块链系统中，计算设备具备如下功能：1）路由，计算设备具有的基本功能，用于支持计算设备之间的通信；2）应用，用于部署在区块链中，根据实际业务需求而实现特定业务，记录实现功能相关的数据形成账本数据，在账本数据中携带数字签名以表示数据来源，将账本数据发送至区块链系统中的其他计算设备，供其他计算设备在验证账本数据来源以及完整性成功时，将账本数据添加至临时区块中，其中，应用实现的业务包括钱包、共享账本、智能合约等；3）区块链，包括一系列按照先后的时间顺序相互接续的区块，新区块一旦加入到区块链中就不会再被移除，区块中记录了区块链系统中计算设备提交的账本数据。

在一些实施例中，每个区块中包括本区块存储交易记录的哈希值（本区块的哈希值）以及前一区块的哈希值，各区块通过哈希值连接形成区块链，另，区块中还包括有区块生成时的时间戳等信息。

以下，对本申请实施例的系统架构进行说明。

图 1 是本申请实施例提供的一种事务执行方法的实施环境示意图。参见图 1，本申请实施例适用于单机数据库、集群数据库、计算存储分离的云原生数据库架构等各类数据库系统，下面以分布式集群数据库为例进行说明。分布式集群数据库系统中包括应用客户端 101、网关服务器 102 以及分布式存储集群 103，在分布式存储集群 103 中包括一个或多个计算设备（即计算节点）。

应用客户端 101 是指用户侧的终端上安装和运行的能够发起数据请求的客户端，该数据请求是 DDL 请求或者 DML 请求等，本申请实施例对此不进行具体限定。可选地，应用客户端 101 的类型包括：支付应用、社交应用、音视频应用、直播应用、购物应用、外卖应用或者打车应用等，本申请实施例不对应用客户端 101 的类型进行具体限定。在一些实施例中，用户侧的终端也称为用户设备、终端设备、用户终端、移动终端、智能终端、通信设备等。终端的设备类型包括：智能手机、平板电脑、笔记本电脑、台式计算机、智能音箱、智能手表、车载终端、智能家电、智能语音交互设备等，但并不局限于此。

应用客户端 101 以及网关服务器 102 通过有线或无线通信方式进行直接或间接地连接，本申请在此不做限制。

网关服务器 102 用于接收外部的数据请求，并将数据请求对应的读写事务分发至分布式存储集群 103，示意性地，用户在终端上登录应用客户端 101，触发应用客户端 101 生成 DDL 请求，例如，DDL 请求为修改数据表 A 的表名，应用客户端 101 调用分布式集群数据库系统提供的 API（Application Programming Interface，应用程序编程接口），将该 DDL 请求发送至网关服务器 102，比如，该 API 是 MySQL API（一种关系型数据库系统提供的 API）。又例如，在智慧交通场景下，新增停车位的描述信息请求为 DDL 请求，而查询已有停车位的请求为 DML 请求。

在一些实施例中，该网关服务器 102 与分布式存储集群 103 中的任一个计算设备合并

同一个物理机上，也即是，让某个计算设备充当网关服务器 102。

分布式存储集群 103 包括一个或多个计算设备，本申请实施例不对分布式存储集群 103 中计算设备的数量进行具体限定，例如，计算设备的数量为 m 个， m 为大于或等于 1 的整数。可选地，每个计算设备采用主备结构（一主多备集群），如图 1 所示，以计算设备为一主两备集群为例进行示意，每个计算设备中包括一个主机和两个备机，可选地，每个主机或备机都配置有代理（Agent）设备，代理设备与主机或备机是物理独立的，或者，代理设备作为主机或备机上的一个代理模块，以计算设备 1 为例，计算设备 1 包括一个主数据库及代理设备（主 database+agent，简称主 DB+agent），此外还包括两备数据库及代理设备（备 database+agent，简称备 DB+agent）。

在一个示例性场景中，每个计算设备的主机或备机的数据库实例集合称为一个 SET（集合），例如，假设某一计算设备为单机设备，那么该计算设备的 SET 仅为该单机设备的数据库实例，假设某一计算设备为一主两备集群，那么该计算设备的 SET 为主机数据库实例以及两个备机数据库实例的集合。

在一些实施例中，上述网关服务器 102 以及分布式存储集群 103 所构成的分布式集群数据库系统，视为一种向用户终端提供数据服务的服务器，该服务器是独立的物理服务器，或者是多个物理服务器构成的服务器集群或者分布式系统，或者是提供云服务、云数据库、云计算、云函数、云存储、网络服务、云通信、中间件服务、域名服务、安全服务、CDN（Content Delivery Network，内容分发网络）以及大数据和人工智能平台等基础云计算服务的云服务器。

以下，对本申请实施例涉及的数据结构进行说明。

为了做到能够让 DDL 事务实现从中断处恢复执行，需要将 DDL 事务所修改数据表中的数据划分为多个小的逻辑或者物理单元，并以这些最小单元作为这一数据表中这一数据行中的断点位置。通常情况下，数据库的对象会以底层文件系统或存储方式来组织对象中的数据，将对象中的数据映射到文件系统或者存储系统大致会涉及如下情况：

a) heap（堆）表数据管理模式

heap 表数据管理模式，数据库以 heap 表方式来管理数据表中的数据记录，即以一定大小的 page（页）或者 block（块）来组织和存放数据表中的数据记录。换言之，数据记录都是存储在一个一个固定大小的数据块中，通常称为 page 或者 block，例如，数据块是 $2^n k$ 大小的 page，同时也以 $2^n k$ 大小的 page 存储实际数据。

b) B+树数据管理模式

选取一个固定大小的 block 来代表 B+树的非叶子结点和叶子结点，实际数据会存放在叶子结点当中，是和上述 a) 的 heap 表数据管理模式同理的存储数据方式。

c) LSM（Log-Structured Merge Tree，结构化合并树）数据管理模式

LSM 数据管理模式即 Key-Value（键-值）数据管理模式，为了提升读写效率，通常会以 block 模式来组织一定数量的 Key-Value 对（键值对），在一些数据库中，为了做到存储能够水平扩展，引入了 region（域）的定义，region 是一种一个或多个物理 block 的逻辑映射，通过增加逻辑映射能够屏蔽掉 region 关联的数据真实存储的位置。换言之，LSM 存储方式下一定大小的数据会组成 block 或者 region 的概念，region 或者 block 是有一定数据范围的。因此，LSM 数据管理模式也被认为是以 block 方式存储数据的。

通过上面的分析，对于各种数据库中关于表数据存储的实现，均能够在存储数据这个粒度上找到一个 block 来作为本申请实施例的 DDL 事务在进行数据处理时的最小单元。

在上述以 block 作为 DDL 事务在进行数据处理时的最小单元的基础上，本申请实施例将对事务执行方法的异常中断恢复执行流程进行说明。本申请实施例适用于单机数据库系统、集群数据库系统、分布式数据库系统、计算存储分离的云原生数据库架构、区块链系统等，在此不做具体限定。

图 2 是本申请实施例提供的一种事务执行方法的流程图。参见图 2，该实施例由数据库系统中的任一计算设备执行，该实施例包括下述步骤：

201、计算设备响应于执行中断的数据定义语言 DDL 事务符合恢复条件，确定该 DDL 事务在执行中断前处理完毕的最后一个数据块。

在一些实施例中，用户在终端上登录应用客户端，触发应用客户端生成 DDL 请求，例如，DDL 请求为向数据表 A 增加一个索引 (Index)，或者，DDL 请求为修改数据表 A 的表名等，本申请实施例不对 DDL 请求的类型进行具体限定。应用客户端在生成 DDL 请求之后，调用 API 将该 DDL 请求发送至计算设备。

在一些实施例中，计算设备接收应用客户端的任一数据请求，解析该数据请求的头字段，当该头字段指示该数据请求为 DDL 请求时，为该 DDL 请求创建一个新的 DDL 进程或线程，或者复用已创建的某一 DDL 进程或线程，并通过该 DDL 进程或线程执行该 DDL 请求关联的 DDL 事务。

在一些实施例中，计算设备串行执行该 DDL 事务，或者，数据库系统内的一个或多个计算设备并行执行该 DDL 事务，例如，单个计算设备上的多个 DDL 线程并行执行该 DDL 事务，或者，多个计算设备上各自的多个 DDL 线程并行执行该 DDL 事务，本申请实施例不在此进行具体限定。

在一些实施例中，在 DDL 事务的执行过程中，由于某种原因（例如，进程或线程崩溃、闪退等）导致 DDL 事务执行中断，在数据库系统重启后，如果后台的 DDL 执行检测器检测到某一个执行中断的 DDL 事务，在该 DDL 事务符合恢复条件的情况下，由于以数据块作为事务处理的最小单元，因此确定出该 DDL 事务在执行中断前处理完毕的最后一个数据块，并执行下述步骤 202。其中，该 DDL 执行检测器用于周期性地检测数据库当前是否有 DDL 事务发生了执行异常情况。

在一些实施例中，每个 DDL 事务划分为两阶段执行，在第一阶段中，向 DDL 事务操作的数据表的数据字典中修改该 DDL 事务操作的对象的定义，例如，如果 DDL 事务是增加一个索引，那么第一阶段下需要修改数据表的数据字典以增加索引的定义，以完成索引结构在表对象中的增加，并将索引置为 Delete Only（仅允许删除）状态，此时表的结构中增加了索引的定义，但该索引还不能被优化器选中，也不能在查询中使用新增的该索引来查询数据；在第二阶段中，需要扫描 DDL 事务操作的全表数据，以处理该 DDL 事务操作的对象涉及的每一条数据记录，例如，如果 DDL 事务是增加一个索引，那么第二阶段下需要同步或者异步执行填充索引数据的任务，即扫描全表数据，为每条数据记录 (Tuple) 创建该索引的 Key 值（键值），此时需要将索引置为 Write Only（仅允许写入）状态，在 Write Only 状态之前开始的修改表数据的事务，如果该事务没有提交，那么在 Write Only 状态设置成功之后，在提交时数据库引擎会拒绝该事务提交以使得该事务被回滚掉。

在一些实施例中，在该第二阶段中，将索引设置成 Write Only 状态的同时，还获取或申请一个事务标识 (Identification, ID)，以该事务标识来指示从设置 Write Only 状态之后最早开始的事务（即 DDL 请求所关联的 DDL 事务），因此，将第一阶段视为 DDL 事务的初始化阶段，将第二阶段视为 DDL 事务的执行阶段，其中，事务标识随着时间戳单调递增。可选地，在单机数据库中基于预先设定的规则来分配事务标识，在集群数据库中向一个用于生成全局事务标识的设备（如协调节点）来申请事务标识，本申请实施例不对事务标识的获取方式进行具体限定。

在一些实施例中，在该第二阶段中，基于该 DDL 事务开始扫描全表数据，例如，如果 DDL 事务是增加一个索引，采用托马斯写规则来填充每条数据记录的索引 Key 值。另外，在将索引设置成 Write Only 状态之后，所有对于该数据表的修改事务，同时也要更新相应的索引，但由于 Write Only 状态的索引还没有完成数据填充，因此在第二阶段下新增的索引也不能被优化器使用来查询数据。根据托马斯写规则，针对在扫描全表、填充索引的 DDL 事务之后开始并修改了索引的用户操作（如 DML 事务），由于 DML 事务的时间戳都比进行扫描的 DDL 事务的时间戳新，导致 DDL 事务在填充索引 Key 值的时候，如果发现有冲突（即索引中已经包含某一 Key 值，说明已经有用户对 Key 值字段做了修改并完成了索引 Key 值的同

步), DDL 填充事务直接丢弃当前 Key 值, 继续后续记录扫描即可。

本申请实施例适用于在线 DDL 场景, 也适用于离线 DDL 场景, 在不同场景下 DDL 事务的恢复条件是不尽相同的, 下面进行说明。

针对在线 DDL 场景, 若该 DDL 事务为在线 DDL 事务, 有两种情况下 DDL 事务的执行中断没有办法继续执行, 即满足下述情况 (I) 或 (II) 时, 代表不符合该恢复条件, 数据库引擎回滚该 DDL 事务:

情况 (I)、DDL 事务尚未完成第一阶段, 由于第一阶段是指在 DDL 事务操作的数据表的数据字典中修改该 DDL 事务操作的对象的定义, 尚未完成第一阶段就代表: 在 DDL 事务操作的数据表的数据字典中不包含该 DDL 事务操作的对象的定义。

例如, DDL 事务增加一个索引, 第一阶段需要修改数据表的数据字典以增加索引的定义, 并将索引置为 Delete Only (仅允许删除) 状态, 如果该 DDL 事务尚未完成第一阶段, 则该数据表的数据字典中没有关于该索引的定义, 比如在该数据表的数据字典中查询不到索引的信息, 那么无法从异常中断点继续执行 DDL 事务, 该 DDL 事务会被回滚掉。

情况 (II)、DDL 事务出现硬件问题, 导致最后一个 block 的下一个 block 以及该下一个 block 的所有副本都不可用。

换言之, DDL 事务在对该最后一个 block 处理完毕之后, 由于某种硬件故障, 导致下一个 block 以及下一个 block 的所有副本都读不出来, 这种原因所引发的异常中断即使在系统重启后也仍然无法恢复。需要说明的是, 在保证 block 数据的一致性能力的情况下, 通常在数据库中每一个 block 都会存储有多个副本 (一主两备、一主多备等), 来确保即使出现硬件磁盘故障, 数据也不会丢失, 因此情况 (II) 只会在极端情况下发生, 即所有机器上的所有副本都不可用, 但这种极端情况的发生概率极低, 几乎为 0。

上述 (I) 和 (II) 介绍了在线 DDL 场景下不满足恢复条件的两种情况, 那么不属于上述 (I) 和 (II) 的其余情况均满足恢复条件, 也即是说, 在线 DDL 场景下的恢复条件包括: 该 DDL 事务操作的数据表的数据字典中包含该 DDL 事务操作的对象的定义, 且该 DDL 事务操作的数据块或该数据块的至少一个副本可读。

针对离线 DDL 场景, 若该 DDL 事务为离线 DDL 事务, 离线 DDL 事务的执行流程相对来说比较简单, 通常都是在 DDL 事务执行后创建一个临时数据表, 将原数据表中的数据记录复制到临时数据表中, 并对临时数据表中的对象完成相应的处理之后, 修改原数据表的表名, 将临时数据表的表名修改为原数据表的表名, 再异步删除原数据表即可。因此, 在离线 DDL 场景下的恢复条件包括: 该 DDL 事务操作的原数据表和该 DDL 事务创建的临时数据表均存在, 此时说明对原数据表中的数据记录尚未扫描完毕, 需要从执行中断点恢复扫描。

需要说明的是, 在离线 DDL 场景下, 如果不满足恢复条件, 说明原数据表或者临时数据表中至少一项不存在, 此时视情况判断是回滚 DDL 事务还是继续推进 DDL 事务。如果原数据表存在, 但临时数据表还没有创建, 在满足这种情况时回滚 DDL 事务, 否则, 可分为如下三种情况 (a) 至 (c) 进行讨论, 在满足下述情况 (a) 至 (c) 中任一项时, 代表执行中断前全表已经扫描完毕, 无需回滚 DDL 事务, 只需要继续推进 DDL 事务即可:

(a) 原数据表改名成功, 临时数据表存在, 说明对原数据表扫描完毕, 继续推进 DDL 事务即可, 换言之, 继续将临时数据表的表名修改为原数据表的表名, 再异步删除原数据表;

(b) 临时数据表改名为原数据表, 原数据表改名后形成的另一临时数据表也存在, 说明对原数据表扫描完毕, 继续推进 DDL 事务即可, 换言之, 继续从数据字典中删除该另一临时数据表, 并异步完成删除该另一临时数据表中的数据;

(c) 临时数据表改名为原数据表, 原数据表改名后形成的另一临时数据表不存在, 继续推进 DDL 事务即可, 换言之, 继续确认异步完成删除该另一临时数据表中的数据。

在上述过程中, 分别介绍了在线 DDL 场景和离线 DDL 场景下, 对 DDL 事务是否符合恢复条件的判定过程, 在确认执行中断的 DDL 事务符合所属场景下的恢复条件时, 计算设备确定该 DDL 事务在执行中断前处理完毕的最后一个 block。在本申请实施例之前已经介绍过,

不管是针对 heap 表数据管理模式、B+树数据管理模式、LSM 数据管理模式，都能够以 block 作为处理 DDL 事务的最小单元，因此，针对每个 DDL 事务增加一个管理数据，该管理数据用于记录该 DDL 事务处理完毕的各个 block，每当 DDL 事务对一个 block 中的每条数据记录对完成相应的处理，则将处理完毕的 block 记录到该管理数据中。可选地，为节约存储空间仅在管理数据中记录各个 block 的 block ID，该管理数据亦称为 block 管理结构。

在一些实施例中，计算设备获取该 DDL 事务的管理数据，从该管理数据中，查询得到该 DDL 事务在执行中断前处理完毕的最后一个 block。由于以 block 作为处理 DDL 事务的最小单元，且对每个 DDL 事务都维护了管理数据，在管理数据中会记录已处理完毕的各个 block，使得 DDL 事务在执行中断后，能够方便地从管理数据中定位到异常中断点，并从异常中断点开始继续执行 DDL 事务，而无需从头开始进行重做或者整体回滚，大大提高了 DDL 事务的成功率。

202、计算设备从该最后一个数据块的下一个数据块开始，继续执行该 DDL 事务。

在一些实施例中，由于各个 block 都是按序存放在数据表中的，因此在定位到执行中断前处理完毕的最后一个 block 之后，说明该最后一个 block 之前的所有 block 都是处理完毕的，无需重复处理，只需要从数据表中定位到该最后一个 block 的下一个 block，并从该下一个 block 开始继续执行 DDL 事务即可，例如，DDL 事务是增加一个索引，在第二阶段中扫描到数据表的 block10 时异常中断了，此时 block10 尚未完成处理因此管理数据中不会记录 block10，而是记录了已处理完毕的 block1~9，确定出异常中断前处理完毕的最后一个 block 是 block9，因此，计算设备从 block9 的下一个 block10 开始，继续执行该 DDL 事务，这样能够避免将 DDL 事务整体回滚，并且也无需从 block1 开始进行重做，最多会产生一个 block 的冗余工作量（即处理到 block10 的最后一条数据记录时发生中断，平均增加半个 block 的冗余工作量），大大提高了数据库的资源利用率。

203、计算设备响应于对该 DDL 事务操作的各个数据块均处理完毕，提交该 DDL 事务。

在一些实施例中，计算设备以 block 为单位来处理该 DDL 事务，即每次读取一个 block，然后对读取的该 block 中的每条数据记录进行处理，上述处理是依据 DDL 事务的类型不同而定的，例如，DDL 事务是增加一个索引，那么对每条数据记录都需要填充索引的 Key 值。在对本 block 的所有数据记录处理完毕后，读取下一个 block，并对下一个 block 中的每条数据记录执行相同的处理，以此类推，迭代执行上述操作，直到不存在需要处理的 block，即代表对所有 block 均处理完毕，此时提交该 DDL 事务，例如，DDL 事务是增加一个索引，在提交 DDL 事务时，将索引置为 Public（发布）状态，并将 DDL 事务进入 Commit（提交）阶段，标记 DDL 任务结束。

上述所有可选技术方案，能够采用任意结合形成本公开的可选实施例，在此不再一一赘述。

本申请实施例提供的技术方案，以数据块作为处理 DDL 事务的最小单元，在 DDL 事务执行中断时，能够方便地定位到执行中断前处理完毕的最后一个数据块，无需整体回滚该 DDL 事务，而是从下一个数据块开始继续执行该 DDL 事务，避免了从头开始重做 DDL 事务的冗余工作量，提高了 DDL 语句所指示的 DDL 事务的成功率，提高了数据库的资源利用率。

在上述实施例中，简略介绍了 DDL 事务在执行中断后，如何定位异常中断点并继续执行 DDL 事务，由于本申请实施例既支持对 DDL 事务进行串行处理，也支持对 DDL 事务进行并行处理，因此，在本申请实施例中，将详细介绍 DDL 事务的串行处理流程，并在下一实施例中详细介绍 DDL 事务的并行处理流程。

图 3 是本申请实施例提供的一种事务执行方法的流程图。参见图 3，该实施例由数据库系统中的任一计算设备执行，该数据库系统包括：单机数据库系统、集群数据库系统、分布式数据库系统、计算存储分离的云原生数据库架构、区块链系统等，该实施例包括下述步骤：

301、计算设备响应于执行中断的 DDL 事务符合恢复条件，获取该 DDL 事务的管理数据，该管理数据用于记录该 DDL 事务处理完毕的各个 block。

关于在线 DDL 场景和离线 DDL 场景下判断该 DDL 事务是否符合恢复条件的过程,已在上一实施例中介绍过,这里不做赘述。

计算设备在检测到执行中断的 DDL 事务符合该恢复条件的情况下,获取该 DDL 事务的管理数据,由于该管理数据用于记录该 DDL 事务处理完毕的各个 block,因此当 DDL 事务在对每一个 block 中存储的各条数据记录均处理完毕的情况下,将处理完毕的该 block 的 block ID (数据块标识) 记录到该管理数据中。

可选地,仅在管理数据中记录各个 block 的 block ID,该管理数据亦称为 block 管理结构,从而能够节约管理数据所占用的存储数据量。示意性地,该管理数据为动态数组,该动态数组中存储有该 DDL 事务处理完毕的各个 block 的 block ID,动态数组中的最后一个元素存放的是最后一个处理完毕的 block 的 block ID。

需要说明的是,动态数组仅仅是管理数据的一种示例性说明,该管理数据还被提供为哈希表、集合、队列、堆栈等其他数据结构,本申请实施例对此不进行具体限定。

在一些实施例中,计算设备在进入 DDL 事务的第二阶段时,获取该 DDL 事务的事务 ID,对该 DDL 事务创建管理数据,此后随着 DDL 事务对数据表的处理过程,实时维护该管理数据,也即是说,随着 DDL 事务扫描全表数据,每对数据表中的一个 block 扫描完毕,则将最新扫描完毕的 block 记录到管理数据中,例如,将最新扫描完毕的 block 的 block ID 存储到动态数组的最后一个元素中。某一时刻,DDL 事务执行中断,DDL 执行检测器发现该 DDL 事务符合恢复条件,从缓存区中获取该 DDL 事务的管理数据,例如,从缓存区中获取该 DDL 事务的动态数组。

302、计算设备从该管理数据中,查询得到该 DDL 事务在执行中断前处理完毕的最后一个 block。

在一些实施例中,由于通常是按照 block 从小到大的顺序来处理数据表中的各个 block,因此在查询时,计算设备从该管理数据中,读取该 DDL 事务在执行中断前处理完毕的各个 block 的 block ID,将 block ID 最大的 block 确定为执行中断前处理完毕的最后一个 block。

在一些实施例中,该管理数据为动态数组的情况下,由于动态数组中的最后一个元素存放的是最后一个处理完毕的 block 的 block ID,因此在查询时,计算设备将该动态数组中最后一个元素内存储的 block ID 所指示的 block,确定为执行中断前处理完毕的最后一个 block。

在上述步骤 301-302 中,提供了确定该 DDL 事务在执行中断前处理完毕的最后一个 block 的一种可能实施方式,由于以 block 作为处理 DDL 事务的最小单元,且对每个 DDL 事务都维护了管理数据,在管理数据中会记录已处理完毕的各个 block,使得 DDL 事务在执行中断后,能够方便地从管理数据中定位到异常中断点,并从异常中断点开始继续执行 DDL 事务,而无需从头开始进行重做或者整体回滚,大大提高了 DDL 事务的成功率。

303、计算设备对该最后一个 block 的下一个 block 或该下一个 block 之后的任一个 block,读取该 block,处理该 block 中存储的每条数据记录。

在一些实施例中,由于各个 block 都是按序存放在数据表中的,因此在定位到执行中断前处理完毕的最后一个 block 之后,说明该最后一个 block 之前的所有 block 都是处理完毕的,无需重复处理,只需要从数据表中定位到该最后一个 block 的下一个 block,并从该最后一个 block 的下一个 block 开始,继续执行该 DDL 事务即可。

例如,DDL 事务是增加一个索引,在第二阶段中扫描到数据表的 block10 时异常中断了,此时 block10 尚未完成处理因此管理数据中不会记录 block10,而是记录了已处理完毕的 block1~9,确定出异常中断前处理完毕的最后一个 block 是 block9,因此,计算设备从 block9 的下一个 block10 开始,继续执行该 DDL 事务,这样能够避免将 DDL 事务整体回滚,并且也无需从 block1 开始进行重做,最多会产生一个 block 的冗余工作量(即处理到 block10 的最后一条数据记录时发生中断),大大提高了数据库的资源利用率。

在从该最后一个 block 的下一个 block 开始继续执行该 DDL 事务时,对该最后一个 block 之后的任一个 block (包括该下一个 block,或者该下一个 block 之后的任一个 block),由于以

block 为单位来处理 DDL 事务，因此计算设备需要以 block 为单位读取数据，即，计算设备读取当前 block，并处理当前 block 中存储的每条数据记录，上述处理是依据 DDL 事务的类型不同而定的，例如，DDL 事务是增加一个索引，对每条数据记录都需要填充索引的 Key 值。

在对当前 block 的所有数据记录处理完毕后，读取当前 block 的下一个 block，并对下一个 block 中的每条数据记录执行相同的处理，以此类推，迭代执行上述操作，直到不存在需要处理的 block，代表对所有 block 均处理完毕，执行下述步骤 306。

304、计算设备响应于对该 DDL 事务的进度查询指令，基于该管理数据和该 DDL 事务所需处理的数据块总数，获取该 DDL 事务的数据进度。

在本申请实施例，提供一种对外可实时查询 DDL 事务的数据进度的功能，由于以 block 作为处理 DDL 事务的最小单位，比较 DDL 事务当前处理完毕的 block 数量（即，管理数据中记录的 block 数量）与 DDL 事务所需处理的 block 总数，即获取到 DDL 事务的数据进度。

在一些实施例中，计算设备将该管理数据中记录的 block 数量除以该 block 总数所得的数值，确定为该 DDL 事务的数据进度。可选地，将该数值转换成百分比之后再确定为该 DDL 事务的数据进度，或者，进行其他的线性或非线性映射，本申请实施例对此不进行具体限定。

305、计算设备向触发该进度查询指令的设备返回该 DDL 事务的数据进度。

在一些实施例中，如果是单机数据库系统，用户在计算设备上输入进度查询指令，数据库引擎从底层查询并返回 DDL 事务的数据进度，此时由计算设备自身对该数据进度进行可视化显示。如果是集群数据库系统，用户输入进度查询指令的设备和处理 DDL 事务的计算设备通常不同，此时计算设备基于进度查询指令向用户侧的设备返回 DDL 事务的数据进度，由用户侧的设备来对该数据进度进行可视化显示。

可选地，以文本形式直接显示该数据进度的数值，或者，以进度条形式实时更新该数据进度，或者，以动画形式动态显示该数据进度的变化等，本申请实施例不对该数据进度的显示方式进行具体限定。其中，该进度条包括：条形进度条、扇形进度条、环形进度条等。

示意性地，用户可输入如下代码，以在输入 DDL 语句的同时，附带显示出 DDL 事务的数据进度，以 DDL 事务为创建一张新的数据表为例：

```
MySQL [test]> show create table t1;
```

```
+-----+-----+
| Table | Create Table |
+-----+-----+
| t1    | CREATE TABLE `t1` (
  `a` int NOT NULL,
  `b` int DEFAULT NULL,
  PRIMARY KEY (`a`),      /* public */
  KEY `idx1` (`b`)        /* write only 66%, parallel degree 24*/
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb3 |
+-----+-----+
1 row in set (0.00 sec)
```

其中，* public *代表主键索引当前状态为 Public，即正常状态，通常在 DDL 事务执行完成后，DDL 事务所操作的对象会显示为 Public 状态，说明该对象当前处于能够正常为用户提供服务的阶段。

其中，* write only 66%, parallel degree 24*代表二级索引 idx1 当前状态为 Write Only，即处于索引创建过程中扫描表数据以生成索引数据的阶段，同时显示当前创建索引的数据进度为 66%，并且以 24 的并行度来并行处理数据。

306、计算设备响应于对该 DDL 事务操作的各个数据块均处理完毕，提交该 DDL 事务。上述步骤 306 与上述步骤 203 同理，这里不做赘述。

上述所有可选技术方案，能够采用任意结合形成本公开的可选实施例，在此不再一一赘述。

本申请实施例提供的技术方案，以数据块作为处理 DDL 事务的最小单元，在 DDL 事务执行中断时，能够方便地定位到执行中断前处理完毕的最后一个数据块，无需整体回滚该 DDL 事务，而是从下一个数据块开始继续执行该 DDL 事务，避免了从头开始重做 DDL 事务的冗余工作量，提高了 DDL 语句所指示的 DDL 事务的成功率，提高了数据库的资源利用率。

在上述实施例中，详细介绍了对执行中断的 DDL 事务的串行恢复处理流程，由于除了支持串行处理之外，还支持并行处理，因此在本申请实施例中将详细介绍对执行中断的 DDL 事务的并行恢复处理流程。

图 4 是本申请实施例提供的一种事务执行方法的流程图。参见图 4，该实施例由数据库系统中的任一计算设备执行，该数据库系统包括：单机数据库系统、集群数据库系统、分布式数据库系统、计算存储分离的云原生数据库架构、区块链系统等，该实施例包括下述步骤：

401、计算设备响应于执行中断的 DDL 事务符合恢复条件，获取该 DDL 事务的管理数据，该管理数据用于记录该 DDL 事务处理完毕的各个 block。

上述步骤 401 与上述步骤 301 同理，这里不做赘述。

402、计算设备从该管理数据中，查询得到该 DDL 事务在执行中断前处理完毕的最后一个 block。

上述步骤 402 与上述步骤 302 同理，这里不做赘述。

403、计算设备基于该 DDL 事务的并行度，从该最后一个 block 的下一个 block 开始，对该 DDL 事务尚未处理的各个 block 进行并行处理。

并行度是指：并行处理中将会启动多少并行处理任务来将原来需要处理的任务进行分解。通常在计算机处理中，就是将原来需要处理的全量数据，划分为若干子任务，启动并行度指定数量的线程，取得子任务，并按照相应的处理逻辑进行处理，理想情况下并行执行能够达到并行度倍数的性能提升，例如，串行处理模式下，由单个 DDL 线程扫描全表数据，并行处理模式下，假设并行度为 10，则会由 10 个 DDL 线程并行扫描全表数据。

在一些实施例中，该并行度是由数据库引擎预先设置的，例如，预先设置某个默认值（如并行度为 5），又例如，预先设置多个并行度，并对每个并行度建立与之关联的数据量区间，形成并行度与数据量区间的映射关系，根据 DDL 事务所操作的对象涉及的数据量，选择与该数据量所位于的数据量区间具有映射关系的并行度，又例如，默认继承该 DDL 事务在执行中断之前所采用的并行度，或者继承上一个 DDL 事务所采用的并行度，又例如，该并行度是在执行中断之后由技术人员进行设定的，本申请实施例不对该并行度的获取方式进行具体限定。

上述步骤 403 中，从该最后一个 block 的下一个 block 开始，继续执行该 DDL 事务，与上述步骤 303 不同的是，将串行处理转换为并行处理。在并行处理模式下，基于该 DDL 事务的并行度（不同的 DDL 事务具有相同或不同的并行度，这里不做限定），可将原本单个的 DDL 事务划分成多个 DDL 子事务，其中 DDL 子事务的数量等于该 DDL 事务的并行度，并分别由并行度数量的进程或线程来并行处理该多个 DDL 子事务，此外，为避免冗余的工作量，不同的 DDL 子事务中不存在任何相同的 block，即避免在并行处理中对某一 block 重复扫描多次所带来的冗余工作量。其中，并行处理模式下每个 DDL 线程或进程处理分配至自身的 DDL 子事务时，与串行处理模式下 DDL 线程或进程处理 DDL 事务的方式同理，这里不做赘述。

需要说明的是，在单机数据库中，这些进程或线程可使用多线程并行处理能力来实现，在集群数据库中，这些进程或线程可分布在多个计算设备上，以达到分布式并行处理的效果，可缓解单个节点的计算压力。

在一些实施例中，在基于并行度将 DDL 事务划分为多个 DDL 子事务时，可根据 DDL 事务所需处理的 block 总数进行等分，使得不同进程或线程上分配的任务较为负载均衡。

在另一些实施例中，在基于并行度将 DDL 事务划分为多个 DDL 子事务时，由于 DDL 子事务各自的进程或线程可能并非位于同一计算设备上，而 DDL 子事务所位于的计算设备当前可用的计算资源也不尽相同，因此无需根据 DDL 事务所需处理的 block 总数进行等分，而是根据 DDL 子事务所位于的计算设备当前可用的计算资源进行灵活分配，例如，当前可用的

计算资源大于预设阈值时,分配的DDL子事务中待扫描的block较多,当前可用的计算资源小于或等于预设阈值时,分配的DDL子事务中待扫描的block较少,其中,该预设阈值为固定值,或者为百分比,本申请实施例对此不进行具体限定。

在另一些实施例中,在基于并行度将DDL事务划分为多个DDL子事务时,以分布式场景为例,由于不同的block可能分布式地存储在不同的计算设备上,因此可依据不同的block所位于的计算设备,对每个计算设备有针对性地划分DDL子事务,例如,先确定该DDL事务所操作的对象涉及到哪些计算设备,再针对每个计算设备,分配存储于该计算设备上的数据的DDL子事务。

404、计算设备在并行处理完毕时,获取该DDL事务的第一block集合和第二block集合,该第一block集合用于记录该DDL事务在开始执行时所需操作的数据表中的各个block,该第二数据块集合用于记录该DDL事务在并行处理完毕时所需操作的数据表中的各个block。

由于在线DDL执行过程中,用户业务是能够修改数据的(采用托马斯写规则),因此在DDL事务扫描全表的过程中,有可能block会产生分裂,即原本的某个block由于插入了数据记录导致分裂成了多个block,此时并行扫描下有可能会漏掉分裂出来的新的block,从而导致数据不一致。因此,在并行处理完毕后,通过执行上述步骤404,获取第一block集合和第二block集合,来解决由于block分裂所导致的不一致问题。

其中,第一block集合是指DDL事务在第二阶段中开始并行扫描时,收集所需扫描的数据表中所有block ID所形成的集合,记作blockIdInit,作为并行扫描的基线。

其中,第二block集合是指DDL事务在第二阶段中并行扫描结束后,重新收集一遍所需扫描的数据表中所有block ID所形成的集合,记作blockIdCur,通过比较第一block集合和第二block集合,只要发生了block分裂现象,那么第二block集合一定会比第一block集合的block ID数量要多,从而能够避免遗漏掉部分分裂产生的block。

在一些实施例中,在开始并行扫描时(通常在执行中断之前)会存储一遍第一block集合,在执行中断之后恢复执行时,一旦并行扫描完毕,则从缓存中读取该第一block集合,并再次收集以获取到第二block集合,比较第一block集合和第二block集合的大小,即比较第一block集合和第二block集合中存储的各个block ID(即集合元素),如果第一block集合和第二block集合一致,代表DDL执行期间本数据表中没有发生block分裂,执行下述步骤405,直接提交DDL事务;如果第一block集合和第二block集合不一致,代表DDL执行期间本数据表中发生了block分裂,即第二block集合中比第一block集合中新增了block,则计算设备确定该第二block集合相较于该第一block集合中新增的block,在对该新增的block处理完毕后,提交该DDL事务。

在一些实施例中,对上述新增的block采取串行处理,即执行与上一实施例中步骤303同理的操作,或者,对上述新增的block采取并行处理,即返回上述步骤403(可切换至另一并行度,也可采用原本的并行度),本申请实施例对此不进行具体限定。

405、计算设备在该第一block集合和该第二block集合一致的情况下,提交该DDL事务。

上述步骤405中提交DDL事务的过程与上述步骤306中提交DDL事务的过程同理,这里不做赘述。

需要说明的是,本申请实施例中,也可执行与上述步骤304-305同理的操作,以在并行处理模式下向用户侧的设备获取并返回DDL事务的数据进度,可选地,可获取并返回DDL事务的总数据进度,此外,还支持对每个DDL子事务分别获取并返回各自的子数据进度,并且还支持对用户侧的设备返回设置的并行度,以提高人机交互效率。

上述所有可选技术方案,能够采用任意结合形成本公开的可选实施例,在此不再一一赘述。

本申请实施例提供的技术方案,以数据块作为处理DDL事务的最小单元,在DDL事务执行中断时,能够方便地定位到执行中断前处理完毕的最后一个数据块,无需整体回滚该DDL事务,而是从下一个数据块开始继续执行该DDL事务,避免了从头开始重做DDL事务

的冗余工作量,提高了 DDL 语句所指示的 DDL 事务的成功率,提高了数据库的资源利用率。

在上述两个实施例,详细介绍了在执行中断后,串行恢复处理 DDL 事务的流程,和并行恢复处理 DDL 事务的流程,在本申请实施例中,将以 DDL 事务为新增一个索引为例,详细介绍在线 DDL 场景下的串行处理流程。

通常在线 DDL 事务又分为两种:一种是只需要修改数据字典中表的定义就能够完成的 DDL 事务,例如,在数据表中追加一个数据列,在线 DDL 执行流程中只需要修改表定义即可;另外一种不但需要修改数据字典中表的定义,同时还需要完成基于表的数据创建对应对象的数据的过程,例如,为数据表增加一个索引,需要修改索引定义,同时需要扫描全表数据并使用对应索引涉及的列信息创建索引对应的 Key 信息内容(即 Key 值)。

本申请实施例主要针对上述后一种情况实现中断续做,因为前一种情况本身就不需要扫描全表数据,如果发生异常即便回滚,下次重新执行也只需要完成修改数据字典中表的定义即可,重做的代价也不大。

图 5 是本申请实施例提供的一种在线增加索引的正常执行流程图,请参考图 5,该实施例由数据库系统中的任一计算设备执行,以 DDL 事务为在线增加索引为例进行说明。

步骤 1、开始一个增加索引(Add Index)的 DDL 任务。

步骤 2、修改数据表对应的数据字典,在表结构上增加一个索引,并设置索引当前状态为 Delete Only 状态,Delete Only 状态下只允许对索引进行删除操作。

步骤 3、提交事务,保存新的表结构到数据字典当中。

步骤 4、开始第二阶段 DDL 执行。

步骤 5、在获得相应权限之后,将索引状态修改为 Write Only 状态,同时获得一个事务 ID。

其中,获取权限的步骤是可选的,以提高 DDL 事务的安全性,但并非一定要获取权限后才能修改索引状态,也支持不获取权限直接修改索引状态,以提高 DDL 事务的执行效率,本申请实施例对此不进行具体限定。

其中,将索引修改成 Write Only 状态之后,后续对于这张表的修改都会同步到索引当中,并且,在设置 Write Only 状态成功时,如果还有未提交的对于表的修改事务,这些修改事务将会被回滚掉,能够保证创建索引的正确性。

步骤 6、开始全表扫描,完成如下任务:

步骤 6.1、以步骤 5 获得的事务 ID 为扫描和建立二级索引数据事务的事务 ID;

步骤 6.2、收集表所包含的 block 总数,并登记到 DDL 任务当中,用来作为 DDL 任务的总体工作量值 total blocks。

换言之,在步骤 6 中,开始扫描全表数据,并记录扫描全表数据所需处理的 block 总数(即总体工作量 total blocks)到 DDL 任务中。

步骤 7、开始扫描,并在 DDL 任务中记录上述事务 ID。

步骤 8、获取第一个 block。

步骤 9、处理 block 中的每一条数据记录,提取对应的数据字段用来创建二级索引的 Key,使用托马斯写规则,写入索引结构当中。由于事务 ID 是在索引状态变为 Write Only 的时候申请的,因此根据托马斯写规则,之前未提交的事务都被回滚了;此外,上述扫描并创建索引的事务,比后续任何用户对于表修改的事务都要老,因此如果两者之间有冲突的话,则只需要保留后续用户对于表对应数据的修改即可。

步骤 10、判断是否处理完 block 中的数据记录?如果是,即处理完 block 中的数据记录,转到步骤 11;如果否,即尚未处理完 block 中的数据记录,返回步骤 9 继续下一条数据记录的处理。

步骤 11、在 DDL 的管理数据(如 Map 动态数组结构)中登记 block 已经扫描完毕,并更新已经完成扫描 block 的总数 processed block num。

换言之,在步骤 11 中,登记当前 block 到 DDL 的 Map 结构中,更新当前已经处理完毕

的 block 数（即 processed block num）。

其中,DDL 事务的数据进度等于 $\text{processed block num} / \text{total blocks}$, 在用户使用 show create table 语句的时候, 计算得到当前 DDL 事务的执行进度, 以进行前台可视化显示。

步骤 12、获得下一个 block。

步骤 13、判断是否还有待处理的 block? 如果是, 即仍有待处理的 block, 返回步骤 9 继续处理; 如果否, 即没有待处理的 block, 说明创建索引过程结束, 转到步骤 14。

步骤 14、修改索引状态为 Public (表明索引创建完成, 能够被正常使用)。

步骤 15、标记 DDL 任务结束。

需要说明的是, 上述步骤 6-13 中的扫描全表创建二级索引的过程, 也能够采取并行实现的方式来加快索引创建的速度, 将在后续的实施例详细描述并行适配的流程, 不做赘述。

图 6 是本申请实施例提供的一种在线增加索引的异常中断后恢复流程图, 请参考图 6, 该实施例由数据库系统中的任一计算设备执行, 以 DDL 事务为在线增加索引为例进行说明。

步骤 1、后台线程发现一个异常中断的增加索引 (Add Index) 任务。

步骤 2、判断是否能够继续执行? 如果否, 即不能继续执行, 进入步骤 3; 如果是, 即能够继续执行, 进入步骤 4。

其中, 有两种情况的执行中断没有办法继续:

情况 (I)、如果该 DDL 还没有完成第一阶段, 则表结构中没有关于该索引的信息, DDL 事务无法继续, 该 DDL 事务会被回滚掉。

情况 (II)、如果出现硬件问题导致 block 数据读不出来, DDL 事务无法继续, 该 DDL 事务会被回滚掉。由于目前部分数据库已经实现了 block 数据的一致性能力, 通常在数据库中每一个 block 都存在有多个副本, 来确保即使出现硬件磁盘故障, 数据也不会丢失, 所以情况 (II) 只会在极端情况下发生, 即所有机器上的所有副本都不可用, 这种概率几乎为 0。

步骤 3、回滚 DDL 事务, 进入步骤 13。

步骤 4、通过 DDL 任务中记录的管理数据 (如 Map 动态数组结构) 中找到处理完毕的最后一个 block。

步骤 5、通过步骤 4 找到的最后一个 block, 找到下一个 block。

可选地, 如果是 heap 表结构, 通过文件定位到该最后一个 block, 再顺序移动到下一个 block 即可。

可选地, 如果是 B+树结构, 通过最后一个 block (叶子 block) 能够链接到下一个 block。

可选地, 如果采用 region 作为数据的逻辑组织方式, 则能够通过 region 找到对应的下一个 region 继续扫描。

步骤 6、判断是否找到下一个 block? 如果是, 即找到了下一个 block, 进入步骤 7; 如果否, 即没有找到下一个 block, 进入步骤 12。

步骤 7、处理 block 中的每一条数据记录, 提取对应的数据字段用来创建二级索引的 Key, 使用托马斯写规则, 写入索引结构当中。由于事务 ID 是在索引状态变为 Write Only 的时候申请的, 因此根据托马斯写规则, 之前未提交的事务都被回滚了; 此外, 上述扫描并创建索引的事务, 比后续任何用户对于表修改的事务都要老, 因此如果两者之间有冲突的话, 则只需要保留后续用户对于表对应数据的修改即可。

步骤 8、判断是否处理完 block 中的数据记录? 如果是, 即处理完 block 中的数据记录, 转到步骤 9; 如果否, 即尚未处理完 block 中的数据记录, 返回步骤 7 继续下一条数据记录的处理。

步骤 9、在 DDL 的管理数据 (如 Map 动态数组结构) 中登记 block 已经扫描完毕, 并更新已经完成扫描 block 的总数 processed block num。

换言之, 在步骤 9 中, 登记当前 block 到 DDL 的 Map 结构中, 更新当前已经处理完毕的 block 数 (即 processed block num)。

其中,DDL 事务的数据进度等于 $\text{processed block num} / \text{total blocks}$, 在用户使用 show create

table 语句的时候, 计算得到当前 DDL 事务的执行进度, 以进行前台可视化显示。

步骤 10、获得下一个 block。

步骤 11、判断是否还有待处理的 block? 如果是, 即仍有待处理的 block, 返回步骤 7 继续处理; 如果否, 即没有待处理的 block, 说明创建索引过程结束, 转到步骤 12。

步骤 12、修改索引状态为 Public (表明索引创建完成, 能够被正常使用)。

步骤 13、标记 DDL 任务结束。

下面针对串行处理模式下, 在线 DDL 事务在执行过程中, 如果发生了中断后恢复, 且 block 发生分裂或合并现象进行分析, 以验证上述中断后恢复流程的正确性。

场景一、分裂或者合并的 block 已经扫描处理完毕。

图 7 是本申请实施例提供的一种 block 分裂的原理性示意图, 假设 block n 为中断断点, 针对 block n 之前已经扫描处理完毕的 block m ($m < n$), 上述 block m 分裂为 block m 和 block m'。通常 block 分裂的原因是 block m 中增加了数据记录, 或者修改其中数据记录的字段导致超过了 block 所能够容纳的大小, 从而分裂为两个 block。因为开始扫描的时候索引状态为 Write Only, 所有对于 block m 中记录的新增或者修改都会反映到新建的索引当中, 因此这部分数据的变化会同步到索引当中, 图 7 示出了 block m 向右边分裂的情况, 如果 block m 向左边分裂, 也不会有任何影响。

针对合并 block, 能够看作是分裂 block 的反向操作, 上图 7 从下往上的变化过程, 即 block m 和 block m' 合并为 block m, 一般合并是因为 block m 或者 block m' 中删除了数据记录或者修改数据记录, 导致 block m 或者 block m' 中的空间使用量或空间使用率小于某个阈值, 为了更好的利用空间会将这两个 block 合并, 合并后成为上图的 block m, 和上面分裂的情况一样, 删除数据记录或者修改数据记录也会更新到索引当中, 因此索引的数据也是正确的。

综上所述, 针对场景一, 只需要从 block n 的下一个 block 继续扫描并完成索引填写即可, 因为索引变为 Write Only 状态之后的所有修改, 都会同步到索引当中, 从而能够保证数据的一致性。

场景二、分裂或者合并的 block, 就是当前扫描处理的 block。

上述场景二也分为两种情况, 其一, 当前处理的数据记录在分裂后的 block m 中, 其二, 当前处理的数据记录在分裂后的 block m' 中。

图 8 是本申请实施例提供的一种 block 分裂的原理性示意图, 假设当前正在处理 block m 时 DDL 事务发生中断, 在中断之后重启任务, 且 block m 分裂为 block m 和 block m'。此时找到 block m 的前一个完成的 block m-1, 并从 block m-1 的下一个 block m 开始重新扫描, 扫描的数据记录如果之前已经扫描并填充到索引中, 那么多次扫描会因为托马斯写规则被丢弃掉, 因此会产生最多一个 block 所容纳的数据记录的冗余工作量, 即假设 block m 原来已经扫描处理到最后一条数据记录了, 中断后就相当于重新处理了一遍 block m 然后丢弃了。

在上述两个实施例中, 详细介绍了在线 DDL 场景下, DDL 事务的正常串行处理流程和中断后恢复流程, 而在本申请实施例和下一实施例中, 将详细介绍离线 DDL 场景下, DDL 事务的正常串行处理流程和中断后恢复流程。

图 9 是本申请实施例提供的一种离线增加索引的正常执行流程图, 请参考图 9, 该实施例由数据库系统中的任一计算设备执行, 以 DDL 事务为离线增加索引为例进行说明。

步骤 1、开始一个离线 (非 Online) 增加索引的 DDL 任务。

步骤 2、创建一个 DDL 执行后的表结构的临时数据表, 即创建一张新定义的临时数据表。

步骤 3、申请数据表对应的权限 (申请锁)。

需要说明的是, 对于分布式数据库, 需要申请分布式的锁。

步骤 4、将表状态设置为 No Write (禁止写) 状态。

其中, No Write 状态一种特殊的离线 DDL 场景下的表状态, 当表定义设置为 No Write 状态, 在分布式数据库中的其他节点对于表的修改事务都不允许进行或者提交, 即使表状态在进入 No Write 状态之前开始的事务, 在提交的时候发现表状态变为了 No Write, 那么数据

库引擎也会拒绝事务提交。

步骤 5、开始全表扫描，并收集表所包含的 block 总数，并登记到 DDL 任务当中，用来作为 DDL 任务的总体工作量值 total blocks。

换言之，在步骤 5 中，开始扫描全表数据，并记录扫描全表数据所需处理的 block 总数（即总体工作量 total blocks）到 DDL 任务中。

步骤 6、获取第一个 block。

步骤 7、处理 block 中的每一条数据记录，将原数据表中的每一条数据记录按照新数据表结构组合成为新的数据记录，并插入步骤 2 创建的临时数据表，即按照新的表结构修改后插入步骤 2 创建的临时数据表中。

步骤 8、判断是否处理完 block 中的数据记录？如果是，即处理完 block 中的数据记录，转到步骤 9；如果否，即尚未处理完 block 中的数据记录，返回步骤 7 继续下一条数据记录的处理。

步骤 9、在 DDL 的管理数据（如 Map 动态数组结构）中登记 block 已经扫描完毕，并更新已经完成扫描 block 的总数 processed block num。

换言之，在步骤 9 中，登记当前 block 到 DDL 的 Map 结构中，更新当前已经处理完毕的 block 数（即 processed block num）。

其中，DDL 事务的数据进度等于 $\text{processed block num} / \text{total blocks}$ ，在用户使用 show create table 语句的时候，计算得到当前 DDL 事务的执行进度，以进行前台可视化显示。

步骤 10、获得下一个 block。

步骤 11、判断是否还有待处理的 block？如果是，即仍有待处理的 block，返回步骤 7 继续处理；如果否，即没有待处理的 block，说明创建索引过程结束，转到步骤 12。

步骤 12、修改原数据表的表名，将步骤 2 创建的临时数据表的表名修改为原数据表的表名，异步删除原数据表。

步骤 13、标记 DDL 任务结束。

需要说明的是，上述步骤 5-11 中的扫描全表创建二级索引的过程，也能够采取并行实现的方式来加快索引创建的速度，将在后续的实施例中进行详细描述并行适配的流程，不做赘述。

图 10 是本申请实施例提供的一种离线增加索引的异常中断后恢复流程图，请参考图 10，该实施例由数据库系统中的任一计算设备执行，以 DDL 事务为离线增加索引为例进行说明。

步骤 1、后台线程发现一个异常中断的 DDL 任务。

步骤 2、判断是否能够继续执行？如果是，即原数据表和临时数据表都存在，需要从扫描表过程的中断点恢复扫描，进入步骤 4；如果否，即不满足上述情况，即原数据表或临时数据表中至少一项不存在，进入步骤 3。

步骤 3、判断是否回滚 DDL 事务？如果是，进入步骤 13，即只有原数据表存在，临时数据表还没有创建，回滚 DDL 事务；如果否，则分为三种情况：a) 原数据表改名成功，临时数据表也在，扫描全表已经结束，继续推进；b) 临时数据表改名为原数据表，原数据表改名后的临时数据表也存在，继续推进，从数据字典中删除临时数据表，异步完成删除临时数据表中的数据记录；c) 临时数据表改名为原数据表，原数据表改名后的临时数据表不存在，继续推进，确认异步完成删除临时数据表中的数据记录，满足 a) -c) 中任一种则进入步骤 12。

步骤 4、申请表相应的修改权限（申请锁），此时表状态还是 No Write 状态。

步骤 5、通过 DDL 任务中记录的管理数据（如 Map 动态数组结构）中找到处理完毕的最后一个 block。

步骤 6、判断是否找到下一个 block？如果是，即找到了下一个 block，进入步骤 7；如果否，即没有找到下一个 block，进入步骤 12。

步骤 7、处理 block 中的每一条数据记录，将原数据表中的每一条数据记录按照新数据表结构组合成为新的数据记录，并插入步骤 2 创建的临时数据表，即按照新的表结构修改后插入步骤 2 创建的临时数据表中。

步骤 8、判断是否处理完 block 中的数据记录？如果是，即处理完 block 中的数据记录，转到步骤 9；如果否，即尚未处理完 block 中的数据记录，返回步骤 7 继续下一条数据记录的处理。

步骤 9、在 DDL 的管理数据（如 Map 动态数组结构）中登记 block 已经扫描完毕，并更新已经完成扫描 block 的总数 processed block num。

换言之，在步骤 9 中，登记当前 block 到 DDL 的 Map 结构中，更新当前已经处理完毕的 block 数（即 processed block num）。

步骤 10、获得下一个 block。

步骤 11、判断是否还有待处理的 block？如果是，即仍有待处理的 block，返回步骤 7 继续处理；如果否，即没有待处理的 block，说明创建索引过程结束，转到步骤 12。

步骤 12、修改原数据表的表名，将步骤 2 创建的临时数据表的表名修改为原数据表的表名，异步删除原数据表。

步骤 13、标记 DDL 任务结束。

在上述实施例中，在离线即非 Online DDL 执行的过程中，通过将表的状态设置为 No Write，就保证了在 DDL 事务完成或者回滚前，数据表对应的 block 不会发生变化。

上述各个实施例中，以 DDL 事务为增加索引为例，介绍了在线 DDL、离线 DDL 场景下各自的正常执行流程和异常中断恢复流程，而在本申请实施例中，将介绍上述各类应用场景下，如何与并行处理模式进行适配。由于是以 block 为单位进行并行处理，因此需要增加用于并行处理的 2 个数据结构：第一 block 集合和第二 block 集合。

第一 block 集合，记作 blockIdInit，是指在开始扫描全表时收集到当前所有的 block ID 构成的集合，用来作为并行扫描的基线。

第二 block 集合，记作 blockIdCur，是指在并行扫描结束后重新收集一遍所有 block ID 构成的集合，用来与基线对比，以检查出是否有因为并行扫描漏掉的 block。

需要说明的是，只有在线 DDL 并采用并行处理模式的情况下，才有可能漏掉 block，因为在线 DDL 执行过程中，用户业务是能够修改数据记录的，因此在扫描全表过程中 block 会产生分裂，并行扫描因为需要初始化记录 block ID，因此会出现可能后续新分裂出来的 block ID 被漏掉，即新分裂出来的 block 仍然需要保证被处理完毕，才能够保证数据一致性。对于离线 DDL 并采用并行处理模式，由于用户业务被阻塞，block 在 DDL 执行过程中不会发生变化，因此并行执行扫描的完成并不会遗漏某些 block。另外，只有 block 分裂才有可能造成遗漏，block 合并是不影响的，因为合并的话会将 2 个 block 合并为一个，这样 block 在初始化的时候都在 blockIdInit 中有记录，因此不会漏掉数据。

图 11 是本申请实施例提供的一种在线增加索引的正常并行执行流程图，请参考图 11 该实施例由数据库系统中的任一计算设备执行，以 DDL 事务为在线增加索引为例进行说明。

步骤 1、开始一个增加索引（Add Index）的 DDL 任务。

步骤 2、修改数据表对应的数据字典，在表结构上增加一个索引，并设置索引当前状态为 Delete Only 状态，Delete Only 状态下只允许对索引进行删除操作。

步骤 3、提交事务，保存新的表结构到数据字典当中。

步骤 4、开始第二阶段 DDL 执行。

步骤 5、在获得相应权限之后，将索引状态修改为 Write Only 状态，同时获得一个事务 ID。

其中，获取权限的步骤是可选的，以提高 DDL 事务的安全性，但并非一定要获取权限后才能修改索引状态，也支持不获取权限直接修改索引状态，以提高 DDL 事务的执行效率，本申请实施例对此不进行具体限定。

其中，将索引修改成 Write Only 状态之后，后续对于这张表的修改都会同步到索引当中，并且，在设置 Write Only 状态成功时，如果还有未提交的对于表的修改事务，这些修改事务将会被回滚掉，能够保证创建索引的正确性。

步骤 6、开始扫描全表数据，并以步骤 5 获得的事务 ID 为扫描和建立二级索引数据事务的事务 ID。

步骤 7、收集需要扫描的表的 block ID 集合（第一 block 集合），收集表中所包含的 block 总数，并登记到 DDL 任务当中，用来作为 DDL 任务的总体工作量值 total blocks。

换言之，在步骤 7 中，记录全表中需要扫描的 block ID 集合（第一 block 集合）和扫描全表数据所需处理的 block 总数（即总体工作量 total blocks）到 DDL 任务中。

步骤 8、选择合适的并行度，根据该 block ID 集合，初始化并行任务，启动并行执行流程（步骤 9-15 为并行任务的执行主流程）。

步骤 9、开始扫描，并在 DDL 任务中记录上述事务 ID。

步骤 10、获取第一个 block。

步骤 11、处理 block 中的每一条数据记录，提取对应的数据字段用来创建二级索引的 Key，使用托马斯写规则，写入索引结构当中。由于事务 ID 是在索引状态变为 Write Only 的时候申请的，因此根据托马斯写规则，之前未提交的事务都被回滚了；此外，上述扫描并创建索引的事务，比后续任何用户对于表修改的事务都要老，因此如果两者之间有冲突的话，则只需要保留后续用户对于表对应数据的修改即可。

步骤 12、判断是否处理完 block 中的数据记录？如果是，即处理完 block 中的数据记录，转到步骤 13；如果否，即尚未处理完 block 中的数据记录，返回步骤 11 继续下一条数据记录的处理。

步骤 13、在 DDL 的管理数据（如 Map 动态数组结构）中登记 block 已经扫描完毕，并更新已经完成扫描 block 的总数 processed block num。

换言之，在步骤 13 中，登记当前 block 到 DDL 的 Map 结构中，更新当前已经处理完毕的 block 数（即 processed block num）。

其中，DDL 事务的数据进度等于 $\text{processed block num} / \text{total blocks}$ ，在用户使用 show create table 语句的时候，计算得到当前 DDL 事务的执行进度，以进行前台可视化显示。

其中，并行任务修改同一变量实现由数据并发编程变量控制实现。

步骤 14、获得下一个 block。

步骤 15、判断是否还有待处理的 block？如果是，即仍有待处理的 block，返回步骤 11 继续处理；如果否，即没有待处理的 block，说明创建索引过程结束，转到步骤 16。

步骤 16、判断是否并行执行？如果是，即处于并行处理模式，进入步骤 17；如果否，即处于串行处理模式，进入步骤 21。

步骤 17、确定所有并行处理任务结束。

步骤 18、再次获取表的 block ID 集合（第二 block 集合）。

步骤 19、将步骤 18 获取的 block ID 集合和步骤 7 初始化的 block ID 集合比较，是否有新增的（由于 block 分裂导致未处理的）block？如果有新增的 block，说明执行过程中发生了分裂，进入步骤 20；如果没有新增的 block，说明所有 block 处理完毕，进入步骤 21。

步骤 20、判断是否需要并行执行？如果是，即对新增的 block 仍需要并行执行（例如新生成了很多 block，依然选择并行执行），返回步骤 8，对新增的 block 构成的集合选择并行度，初始化并行任务，启动并行执行流程；如果否，即对新增的 block 不需要并行执行，返回步骤 9，进行串行处理新增的 block，这种串行执行方式在结束后不需要再判断过程中是否新增了 block。

步骤 21、修改索引状态为 Public（表明索引创建完成，能够被正常使用），同时标记 DDL 任务为已完成。

图 12 本申请实施例提供的一种在线增加索引的异常中断后并行恢复流程图，请参考图 12，该实施例由数据库系统中的任一计算设备执行，以 DDL 事务为在线增加索引为例说明。

步骤 1、后台线程发现一个异常中断的增加索引（Add Index）任务。

步骤 2、判断是否能够继续执行？如果否，即不能继续执行，进入步骤 3；如果是，即能

够继续执行，进入步骤 4。

其中，有两种情况的执行中断没有办法继续：

情况（I）、如果该 DDL 还没有完成第一阶段，则表结构中没有关于该索引的信息，DDL 事务无法继续，该 DDL 事务会被回滚掉。

情况（II）、如果出现硬件问题导致 block 数据读不出来，DDL 事务无法继续，该 DDL 事务会被回滚掉。由于目前部分数据库已经实现了 block 数据的一致性能力，通常在数据库中每一个 block 都存在有多个副本，来确保即使出现硬件磁盘故障，数据也不会丢失，所以情况（II）只会在极端情况下发生，即所有机器上的所有副本都不可用，这种概率几乎为 0。

步骤 3、回滚 DDL 事务，进入步骤 18。

步骤 4、通过 DDL 任务中记录的管理数据（如 Map 动态数组结构）找到处理完毕的最后一个 block，并查找最后一个 block 的下一个 block。

步骤 5、对剩下的 block 选择合适的并行度，初始化并行任务，启动并行执行。

步骤 6、判断是否找到下一个 block？如果是，即找到了下一个 block，进入步骤 7；如果否，即没有找到下一个 block，进入步骤 17。

步骤 7、处理 block 中的每一条数据记录，提取对应的数据字段用来创建二级索引的 Key，使用托马斯写规则，写入索引结构当中。由于事务 ID 是在索引状态变为 Write Only 的时候申请的，因此根据托马斯写规则，之前未提交的事务都被回滚了；此外，上述扫描并创建索引的事务，比后续任何用户对于表修改的事务都要老，因此如果两者之间有冲突的话，则只需要保留后续用户对于表对应数据的修改即可。

步骤 8、判断是否处理完 block 中的数据记录？如果是，即处理完 block 中的数据记录，转到步骤 9；如果否，即尚未处理完 block 中的数据记录，返回步骤 7 继续下一条数据记录的处理。

步骤 9、在 DDL 的管理数据（如 Map 动态数组结构）中登记 block 已经扫描完毕，并更新已经完成扫描 block 的总数 processed block num。

换言之，在步骤 9 中，登记当前 block 到 DDL 的 Map 结构中，更新当前已经处理完毕的 block 数（即 processed block num）。

其中，DDL 事务的数据进度等于 $\text{processed block num} / \text{total blocks}$ ，在用户使用 show create table 语句的时候，计算得到当前 DDL 事务的执行进度，以进行前台可视化显示。

步骤 10、获得下一个 block。

步骤 11、判断是否还有待处理的 block？如果是，即仍有待处理的 block，返回步骤 7 继续处理；如果否，即没有待处理的 block，说明创建索引过程结束，转到步骤 12。

步骤 12、判断是否并行执行？如果是，即处于并行处理模式，进入步骤 13；如果否，即处于串行处理模式，进入步骤 17。

步骤 13、确定所有并行处理任务结束。

步骤 14、再次获取表的 block ID 集合（第二 block 集合）。

步骤 15、将步骤 14 获取的 block ID 集合和 DDL 事务初始化的 block ID 集合比较，是否有新增的（由于 block 分裂导致未处理的）block？如果有新增的 block，说明执行过程中发生了分裂，进入步骤 16；如果没有新增的 block，说明所有 block 处理完毕，进入步骤 17。

步骤 16、判断是否需要并行执行？如果是，即对新增的 block 仍需要并行执行（例如新生成了很多 block，依然选择并行执行），返回步骤 5，对新增的 block 构成的集合选择并行度，初始化并行任务，启动并行执行流程；如果否，即对新增的 block 不需要并行执行，返回步骤 6，进行串行处理新增的 block，这种串行执行方式在结束后不需要再判断过程中是否新增了 block。

步骤 17、修改索引状态为 Public（表明索引创建完成，能够被正常使用）。

步骤 18、标记 DDL 任务结束。

通过上述两个实施例，能够看出，在并行扫描数据情况下，增加了一个新增 block 集合

的检测与处理的过程,即比较第一 block 集合和第二 block 集合中是否发现新增 block,以避免有分裂产生的新的 block 被漏掉。需要说明的是,串行执行是不会出现漏扫描的,因为串行的执行不需要提前采集 block 集合,顺序扫描所有的 block,因此不会出现漏掉过程中新分裂的 block 中有未扫描到的数据。

上述两个实施例所提供的并行执行流程,能够大大的提升数据大表的扫描速度,相对于增加的检测过程来说,数据库系统的整体性能仍然能够保证大幅度提升。

在上述各个实施例中,详细介绍了在各类应用场景下,从 DDL 事务的数据处理中断点,恢复继续执行的技术方案,针对涉及到需要扫描全表完成数据处理的每条 DDL 事务,理论上只需要扫描一次全表数据即可完成 DDL 事务,从而大大提高了 DDL 事务的成功率;此外,由于能够从中断处继续执行,大大节约 DDL 执行期间占用的集群计算和存储资源,使得之前计算所使用的资源不会因为 DDL 事务的中断,最终语句被回滚而被浪费掉,换言之,大大的节约了由于 DDL 事务失败带来的资源重复使用的浪费,使得分布式数据库资源利用率更高。

进一步地,由于能够很方便地返回 DDL 事务的数据进度,使得用户能够清楚的知道目前 DDL 执行的情况,提升了人机交互效率。此外,由于以 block 为单位来处理 DDL 事务,很容易实现并行扫描处理数据,从而大大提升了 DDL 执行的速度,通过合理地并行处理数据,能够做到真正做到迅速完成 DDL 语句。

图 13 是本申请实施例提供的一种事务执行装置的结构示意图,如图 13 所示,该装置包括:

确定模块 1301,用于在执行中断的数据定义语言 DDL 事务符合恢复条件的情况下,确定该 DDL 事务在执行中断前处理完毕的最后一个数据块;

执行模块 1302,用于从该最后一个数据块的下一个数据块开始,继续执行该 DDL 事务;

提交模块 1303,用于在对该 DDL 事务操作的各个数据块均处理完毕的情况下,提交该 DDL 事务。

本申请实施例提供的技术方案,以数据块作为处理 DDL 事务的最小单元,在 DDL 事务执行中断时,能够方便地定位到执行中断前处理完毕的最后一个数据块,无需整体回滚该 DDL 事务,而是从下一个数据块开始继续执行该 DDL 事务,避免了从头开始重做 DDL 事务的冗余工作量,提高了 DDL 语句所指示的 DDL 事务的成功率,提高了数据库的资源利用率。

在一些实施例中,基于图 13 的装置组成,该确定模块 1301 包括:获取单元,用于获取该 DDL 事务的管理数据,该管理数据用于记录该 DDL 事务处理完毕的各个数据块;查询单元,用于从该管理数据中,查询得到该最后一个数据块。

在一些实施例中,该管理数据为动态数组,该动态数组中存储有该 DDL 事务处理完毕的各个数据块的数据块标识;该查询单元用于:将该动态数组中最后一个元素内存储的数据块标识所指示的数据块,确定为该最后一个数据块。

在一些实施例中,在对每一个数据块中存储的各条数据记录均处理完毕的情况下,将该数据块的数据块标识记录在该管理数据中。

在一些实施例中,基于图 13 的装置组成,该装置还包括:第一获取模块,用于响应于对该 DDL 事务的进度查询指令,基于该管理数据和该 DDL 事务所需处理的数据块总数,获取该 DDL 事务的数据进度;发送模块,用于向触发该进度查询指令的设备返回该 DDL 事务的数据进度。

在一些实施例中,该第一获取模块用于:将该管理数据中记录的数据块数量除以该数据块总数所得的数值,确定为该 DDL 事务的数据进度。

在一些实施例中,该执行模块 1302 用于:对该下一个数据块或该下一个数据块之后的任一个数据块,读取该数据块,处理该数据块中存储的每条数据记录。

在一些实施例中,该执行模块 1302 用于:基于该 DDL 事务的并行度,从该最后一个数据块的下一个数据块开始,对该 DDL 事务尚未处理的各个数据块进行并行处理。

在一些实施例中,基于图 13 的装置组成,该装置还包括:第二获取模块,用于在并行处

理完毕的情况下,获取该 DDL 事务的第一数据块集合和第二数据块集合,该第一数据块集合用于记录该 DDL 事务在开始执行时所需操作的数据表中的各个数据块,该第二数据块集合用于记录该 DDL 事务在并行处理完毕时所需操作的数据表中的各个数据块;该提交模块 1303,还用于在该第一数据块集合和该第二数据块集合一致的情况下,提交该 DDL 事务;该执行模块 1302,还用于在该第一数据块集合和该第二数据块集合不一致的情况下,确定该第二数据块集合相较于该第一数据块集合中新增的数据块;该提交模块 1303,还用于在对该新增的数据块处理完毕后,提交该 DDL 事务。

在一些实施例中,在该 DDL 事务为在线 DDL 事务的情况下,该恢复条件包括:该 DDL 事务操作的数据表的数据字典中包含该 DDL 事务操作的对象的定义,且该 DDL 事务操作的数据块或该数据块的至少一个副本可读。

在一些实施例中,在该 DDL 事务为离线 DDL 事务的情况下,该恢复条件包括:该 DDL 事务操作的原数据表和该 DDL 事务创建的临时数据表均存在。

上述所有可选技术方案,能够采用任意结合形成本公开的可选实施例,在此不再一一赘述。

需要说明的是:上述实施例提供的事务执行装置在执行 DDL 事务时,仅以上述各功能模块的划分进行举例说明,实际应用中,能够根据需要而将上述功能分配由不同的功能模块完成,即将计算设备的内部结构划分成不同的功能模块,以完成以上描述的全部或者部分功能。另外,上述实施例提供的事务执行装置与事务执行方法实施例属于同一构思,其具体实现过程详见事务执行方法实施例,这里不再赘述。

图 14 是本申请实施例提供的一种终端的结构示意图,终端 1400 是计算设备的一种示例性说明。可选地,该终端 1400 的设备类型包括:智能手机、平板电脑、笔记本电脑或台式电脑。终端 1400 还可能被称为用户设备、便携式终端、膝上型终端、台式终端等其他名称。

通常,终端 1400 包括有:处理器 1401 和存储器 1402。

可选地,处理器 1401 包括一个或多个处理核心,比如 4 核心处理器、8 核心处理器等。可选地,处理器 1401 采用 DSP (Digital Signal Processing, 数字信号处理)、FPGA (Field-Programmable Gate Array, 现场可编程门阵列)、PLA (Programmable Logic Array, 可编程逻辑阵列) 中的至少一种硬件形式来实现。在一些实施例中,处理器 1401 包括主处理器和协处理器,主处理器是用于对在唤醒状态下的数据进行处理的处理单元,也称 CPU (Central Processing Unit, 中央处理器);协处理器是用于对在待机状态下的数据进行处理的低功耗处理器。在一些实施例中,处理器 1401 集成有 GPU (Graphics Processing Unit, 图像处理器),GPU 用于负责显示屏所需要显示的内容的渲染和绘制。

在一些实施例中,存储器 1402 包括一个或多个计算机可读存储介质,可选地,该计算机可读存储介质是非暂态的。可选地,存储器 1402 还包括高速随机存取存储器,以及非易失性存储器,比如一个或多个磁盘存储设备、闪存存储设备。在一些实施例中,存储器 1402 中的非暂态的计算机可读存储介质用于存储至少一个程序代码,该至少一个程序代码用于被处理器 1401 所执行以实现本申请中各个实施例提供的事务执行方法。

在一些实施例中,终端 1400 还可选包括有:外围设备接口 1403 和至少一个外围设备。处理器 1401、存储器 1402 和外围设备接口 1403 之间能够通过总线或信号线相连。各个外围设备能够通过总线、信号线或电路板与外围设备接口 1403 相连。外围设备包括:显示屏 1405。

外围设备接口 1403 可被用于将 I/O (Input /Output, 输入/输出) 相关的至少一个外围设备连接到处理器 1401 和存储器 1402。在一些实施例中,处理器 1401、存储器 1402 和外围设备接口 1403 被集成在同一芯片或电路板上;在一些其他实施例中,处理器 1401、存储器 1402 和外围设备接口 1403 中的任意一个或两个在单独的芯片或电路板上实现。

显示屏 1405 用于显示 UI (User Interface, 用户界面)。可选地,该 UI 包括图形、文本、图标、视频及其它们的任意组合。当显示屏 1405 是触摸显示屏时,显示屏 1405 还具有采集在显示屏 1405 的表面或表面上方的触摸信号的能力。该触摸信号能够作为控制信号输入至处

理器 1401 进行处理。可选地，显示屏 1405 还用于提供虚拟按钮和/或虚拟键盘，也称软按钮和/或软键盘。可选地，显示屏 1405 采用 LCD (Liquid Crystal Display, 液晶显示屏)、OLED (Organic Light-Emitting Diode, 有机发光二极管) 等材质制备。

本领域技术人员能够理解，图 14 中示出的结构并不构成对终端 1400 的限定，能够包括比图示更多或更少的组件，或者组合某些组件，或者采用不同的组件布置。

图 15 是本申请实施例提供的一种计算设备的结构示意图，该计算设备 1500 可因配置或性能不同而产生比较大的差异，该计算设备 1500 包括一个或一个以上处理器 (Central Processing Units, CPU) 1501 和一个或一个以上的存储器 1502，其中，该存储器 1502 中存储有至少一条计算机程序，该至少一条计算机程序由该一个或一个以上处理器 1501 加载并执行以实现上述各个实施例提供的事务执行方法。可选地，该计算设备 1500 还具有有线或无线网络接口、键盘以及输入输出接口等部件，以便进行输入输出，该计算设备 1500 还包括其他用于实现设备功能的部件，在此不做赘述。

在示例性实施例中，还提供了一种计算机可读存储介质，例如包括至少一条计算机程序的存储器，上述至少一条计算机程序可由终端中的处理器执行以完成上述各个实施例中的事务执行方法。例如，该计算机可读存储介质包括 ROM (Read-Only Memory, 只读存储器)、RAM (Random-Access Memory, 随机存取存储器)、CD-ROM (Compact Disc Read-Only Memory, 只读光盘)、磁带、软盘和光数据存储设备等。

在示例性实施例中，还提供了一种计算机程序产品或计算机程序，包括一条或多条程序代码，该一条或多条程序代码存储在计算机可读存储介质中。计算设备的一个或多个处理器能够从计算机可读存储介质中读取该一条或多条程序代码，该一个或多个处理器执行该一条或多条程序代码，使得计算设备能够执行以完成上述实施例中的事务执行方法。

本领域普通技术人员能够理解实现上述实施例的全部或部分步骤能够通过硬件来完成，也能够通过程序来指令相关的硬件完成，可选地，该程序存储于一种计算机可读存储介质中，可选地，上述提到的存储介质是只读存储器、磁盘或光盘等。

以上所述仅为本申请的可选实施例，并不用以限制本申请，凡在本申请的精神和原则之内，所作的任何修改、等同替换、改进等，均应包含在本申请的保护范围之内。

权利要求书

- 1.一种事务执行方法，由计算设备执行，所述方法包括：
在执行中断的数据定义语言 DDL 事务符合恢复条件的情况下，确定所述 DDL 事务在执行中断前处理完毕的最后一个数据块；
从所述最后一个数据块的下一个数据块开始，继续执行所述 DDL 事务；
在对所述 DDL 事务操作的各个数据块均处理完毕的情况下，提交所述 DDL 事务。
- 2.根据权利要求 1 所述的方法，所述确定所述 DDL 事务在执行中断前处理完毕的最后一个数据块包括：
获取所述 DDL 事务的管理数据，所述管理数据用于记录所述 DDL 事务处理完毕的各个数据块；
从所述管理数据中，查询得到所述最后一个数据块。
- 3.根据权利要求 2 所述的方法，所述管理数据为动态数组，所述动态数组中存储有所述 DDL 事务处理完毕的各个数据块的数据块标识；
所述从所述管理数据中，查询得到所述最后一个数据块包括：
将所述动态数组中最后一个元素内存储的数据块标识所指示的数据块，确定为所述最后一个数据块。
- 4.根据权利要求 2 或 3 所述的方法，在对每一个数据块中存储的各条数据记录均处理完毕的情况下，将所述数据块的数据块标识记录在所述管理数据中。
- 5.根据权利要求 2 或 3 所述的方法，所述方法还包括：
响应于对所述 DDL 事务的进度查询指令，基于所述管理数据和所述 DDL 事务所需处理的数据块总数，获取所述 DDL 事务的数据进度；
向触发所述进度查询指令的设备返回所述 DDL 事务的数据进度。
- 6.根据权利要求 5 所述的方法，所述基于所述管理数据和所述 DDL 事务所需处理的数据块总数，获取所述 DDL 事务的数据进度包括：
将所述管理数据中记录的数据块数量除以所述数据块总数所得的数值，确定为所述 DDL 事务的数据进度。
- 7.根据权利要求 1 所述的方法，所述从所述最后一个数据块的下一个数据块开始，继续执行所述 DDL 事务包括：
对所述下一个数据块或所述下一个数据块之后的任一个数据块，读取所述数据块，处理所述数据块中存储的每条数据记录。
- 8.根据权利要求 1 或 7 所述的方法，所述从所述最后一个数据块的下一个数据块开始，继续执行所述 DDL 事务包括：
基于所述 DDL 事务的并行度，从所述最后一个数据块的下一个数据块开始，对所述 DDL 事务尚未处理的各个数据块进行并行处理。
- 9.根据权利要求 8 所述的方法，所述方法还包括：
在并行处理完毕的情况下，获取所述 DDL 事务的第一数据块集合和第二数据块集合，所述第一数据块集合用于记录所述 DDL 事务在开始执行时所需操作的数据表中的各个数据块，所述第二数据块集合用于记录所述 DDL 事务在并行处理完毕时所需操作的数据表中的各个数据块；
在所述第一数据块集合和所述第二数据块集合一致的情况下，提交所述 DDL 事务；
在所述第一数据块集合和所述第二数据块集合不一致的情况下，确定所述第二数据块集合相较于所述第一数据块集合中新增的数据块，在对所述新增的数据块处理完毕后，提交所述 DDL 事务。
- 10.根据权利要求 1 所述的方法，在所述 DDL 事务为在线 DDL 事务的情况下，所述恢复

条件包括:所述 DDL 事务操作的数据表的数据字典中包含所述 DDL 事务操作的对象定义,且所述 DDL 事务操作的数据块或所述数据块的至少一个副本可读。

11.根据权利要求 1 所述的方法,在所述 DDL 事务为离线 DDL 事务的情况下,所述恢复条件包括:所述 DDL 事务操作的原数据表和所述 DDL 事务创建的临时数据表均存在。

12.一种事务执行装置,所述装置包括:

确定模块,用于在执行中断的数据定义语言 DDL 事务符合恢复条件的情况下,确定所述 DDL 事务在执行中断前处理完毕的最后一个数据块;

执行模块,用于从所述最后一个数据块的下一个数据块开始,继续执行所述 DDL 事务;

提交模块,用于在对所述 DDL 事务操作的各个数据块均处理完毕的情况下,提交所述 DDL 事务。

13.一种计算设备,所述计算设备包括一个或多个处理器和一个或多个存储器,所述一个或多个存储器中存储有至少一条计算机程序,所述至少一条计算机程序由所述一个或多个处理器加载并执行以实现如权利要求 1 至权利要求 11 任一项所述的事务执行方法。

14.一种存储介质,所述存储介质中存储有至少一条计算机程序,所述至少一条计算机程序由处理器加载并执行以实现如权利要求 1 至权利要求 11 任一项所述的事务执行方法。

15.一种计算机程序产品,所述计算机程序产品包括至少一条计算机程序,所述至少一条计算机程序由处理器加载并执行以实现如权利要求 1 至权利要求 11 任一项所述的事务执行方法。