聊聊wiredtiger架构上篇

fundamentals

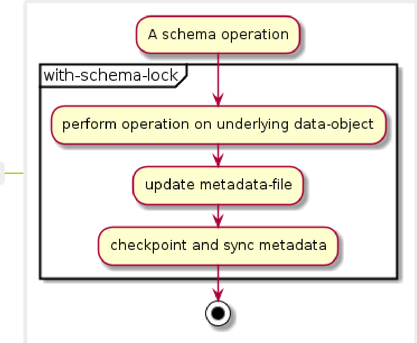
- WT_CONNECTION
  - 定义 — wt存储引擎实例处理实例,任何时刻针对wt存储引擎只会有一个处理实例
  - 初始化
    - WT_CONNECTION是通wiredtiger_open接口初始化，通过一些列的配置参数改变wt引擎的处理逻辑
    - wiredtiger_open传入"create"会创建数据库同时创建对应的turtle文件和元数据文件
    - wt引擎成功open后会初化内部的工作线程分别处理statistics/loggine/eviction/checkpoint/cache management的逻辑
  - 运行时 — wt引擎运行期间可以通过reconfigure函数动态配置的
- WT_SESSION
  - 定义
    - wt的连接初始化后用户的请求发送是通过WT_SESSION，open_session用来初始化WT_SESSION
    - WT_SESSION分为两种类型的会话，一种是用户会话(用户端);另外一种是内部会话(内部线程使用)
    - wt中所有的cursor会关联一个WT_SESSION
  - 初始化 — WT_SESSION初始化是通过open_session函数进行初始化
- Cursor
  - 类型
    - WT_CURSOR
    - WT_CURSOR_BACKUP — 用来管理backup的游标(包括全量和增量备份)
    - WT_CURSOR_BTREE — 用来访问btree的游标
    - WT_CURSOR_BULK
    - WT_CURSOR_DATA_SOURCE
    - WT_CURSOR_DUMP — 被wt dump工具内部使用的游标
    - WT_CURSOR_INDEX
    - WT_CURSOR_LOG
    - WT_CURSOR_METADATA
    - WT_CURSOR_STAT
  - 定义 — cursor用来读取和更改数据的，通过open_cursor来初始化，访问的数据类型包括btree的kv对/索引/statictics/log file/metadata
- Transactions
  - 定义 — WT_TXN
  - 说明
    - wt引擎执行(除了schema操作外)读写都会在一个事务内。如果没有显示的申请begin申明事务，WT会自动为用户操作创建事务
    - 事务完美支持ACID，支持read uncommit/read commit/snapshot隔离级别,事务没有提交前都会在内存中，提交后会持久化到存储介质。
    - 事务的回滚会在内存中，如果有部分持久化,evication和checkpoint线程进行可见性检查
    - prepared transactions 核心目的是为了分布式的两阶段锁而设计
  - 说明
    - 事务快照是WT引擎提供隔离的基础，提供数据版本可见性的基础
    - 事务内一个快照标识特定版本是否可以被访问的基础
    - snapshot隔离级别中snapshot被创建是在事务开始和贯穿整个事务的周期，这个隔离级别下读取只能读取事务开始之前的数据
    - read-committed隔离级别可以读取在事务开始后已经提交的数据(不可重复读)
  - 实现原理
    - snapshot实现是在snapshot创建时候通过捕获全局事务状态实现，维护在每个session的事务上下文中(WT_TXN)
    - Maximum transaction ID是创建snapshot时候全局最大的事务ID,可读的数据的事务一定是小于这个全局最大的事务ID
    - Concurrent transaction IDs是创建snapshot时候所有的active事务的列表，用于读取数据时候的版本可见性检查用的
    - Minimum transaction ID是创建snapshot时候最小的事务ID
  - 可见性规则(__wt_txn_visible_all)
    - 规则1：对于事务ID>=Maximum transaction ID，数据不可见(说明这个事务在创建snapshot之后的新事务)
    - 规则2：对于事务ID<=Minimum transaction ID的事务，只要事务提交了数据都可见
    - 规则3：对于事务ID存在于Concurrent transaction ID列表中，如果事务没提交数据是不可见的
    - 规则4：其他的情况数据都可见
- Checkpoint — 说明 — 检查点事需要snapshot来把内存数据刷到磁盘，检查点有自己私有的全局事务状态和snapshot状态

internal threads

- statistics log server线程(__wt_statlog_create)
- tiered server线程(__wt_tiered_storage_create)
- log server线程(__wt_logmgr_open)
- txn recover线程(__wt_txn_recover)
- eviction server线程(__wt_evict_create)
- handle sweep server线程(__wt_sweep_create)
- capacity server线程(__wt_capacity_server_create)
- checkpoint server线程(__wt_checkpoint_server_create)

Data Organization

- Schema
  - 说明 — Schema定义某种格式的数据集合的表
  - Schema Integrity



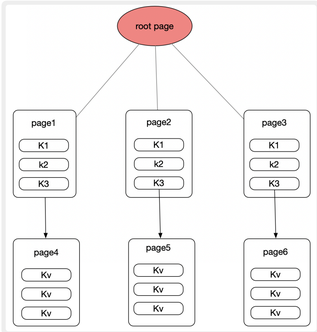
  - Data Files
    - 基于行存储的表都是以{table name}.wt作为数据文件存储
    - 基于列存储的表是以{table name}_{column name}.wt多个文件方式存储
    - 表索引文件都是以{table name}_{index name}.wti单独文件形式存储
    - LSM树存储是以{table name}-{chunk id>.lsm文件存储
    - LSM树的bloom filters是以<table name>-<chunk id>.bf文件形式存储
  - Format Types
    - 设置 — 通过key_format和value_format来设定存储格式
    - 存储格式

Format	C Type	Python type	Notes
x	N/A	N/A	pad byte, no associated value
b	int8_t	int	signed byte
B	uint8_t	int	unsigned byte
h	int16_t	int	signed 16-bit
H	uint16_t	int	unsigned 16-bit
i	int32_t	int	signed 32-bit
I	uint32_t	int	unsigned 32-bit
l	int32_t	int	signed 32-bit
L	uint32_t	int	unsigned 32-bit
q	int64_t	int	signed 64-bit
Q	uint64_t	int	unsigned 64-bit
r	uint64_t	int	record number
s	char[]	str	fixed-length string
S	char[]	str	NUL-terminated string
t	uint8_t	int	fixed-length bit field
u	WT_ITEM *	bytes	raw byte array

- Metadata
  - 说明 — Metadata元数据是以uri存储，key/value配置数据表
  - Metadata Table
    - 存储文件或者表的核心信息，其中包括configurations、checkpoints.
    - WiredTiger.wt是最核心的元数据表
  - Storing Checkpoint Metadata
    - 每个btree的checkpoint信息存储是以metadata entry存储.
    - checkpoint的元数据是为了保证btree表崩溃后从哪里进行恢复
    - 元数据表中btree的checkpoint元数据包括checkpoint(文件的checkpoint)、checkpoint_lsm(上一次checkpoint lsn)、checkpoint_backup_info(增量备份的信息)
  - Turtle file — WiredTiger.wt是存储用户表的元数据表，WiredTiger.turtle是WiredTiger.wt这个元数据表的元数据信息存储
  - Reading Metadata
    - 元数据信息读取通过设置uri="metadata:" 来读取
    - 通过wt list -v命令来dump出WiredTiger.wt中的数据

Data Sources

- B-Trees
  - 结构示意图



  - 说明
    - WT中的表是以B-Tree数据结构组织，其结构包含了root/internal/leaf page,internal page存储key以及key所在leaf page数据位置;leaf page存储了完整的key和value
    - 每个btree对应一个Dhandles用来操作page
    - WT_BTREE包含一个key/value对的缓存、从文件读写数据函数
    - 当内存不足以存储整个btree,在WT_TREE中会针对每个WT_PAGE设置一个WT_REF来判断page是否被加载到内存中
    - insert/modify来更新values时候使用WT_INSERT结构来插入;WT_UPDATE来进行modify或者delete操作
  - 数据结构
    - WT_BTREE
    - WT_PAGE
- Data Handles
  - 说明
    - dhandle (WT_DATA_HANDLE)是数据源操作的封装.可以访问任何数据源。对于btree操作可以直接(WT_BTREE *)dhandle->handle得到这个dhandle
    - session会Cache主dhandle，如果没有发现就直接检索全局dhanle的列表然后加读锁
    - 全局dhandle列表维护是通过sweep server间隔close_scan_interval时间内长期没有被使用的dhanle的回收(session_inuse=0)
    - sweep server会检查任何session中使用dhanle的使用情况.如果session_ref中的count大于0，是不能从全局dhandle列表中移除
  - 生命周期 — Dhandles是被connection，在sessions中共享。WiredTiger在全局dhanle列表中维护(R/W锁保护)
  - 数据结构 — WT_DHANDLE
- Row Store and Column Store
  - 说明
    - WiredTiger中行存储和列存储都是基于B-Tree.列存有固定长度和边长的存储模式
    - row-stores 和 column-stores都是基于WT_BTREE数据结构实现
  - 内部用途

	Row Store	Column Store - Variable Length	Column Store - Fixed Length
Characteristics			
Internal representation	B-Tree (WT_BTREE)	B-Tree (WT_BTREE)	B-Tree (WT_BTREE)
B-Tree type	Generalized (BTREE_ROW)	Specialized (BTREE_COL_VAR)	Specialized (BTREE_COL_FIX)
Leaf pages	Key count equal to half the number of physical entries (unless all empty values flag is set where key count is equal to the number of physical entries)	Only the first key is stored	Only the first key is stored
Record key	Anything (byte-string) prefix compressed or overflow objects	Record id (64 bit unsigned integer)	Record id (64 bit unsigned integer)
Record value	Variable byte string length	Variable byte string length	Fixed bit string length (up to 8 bits)
Features			
Random cursor	Yes	No	No
Block compression	Yes	Yes	Yes
Dictionary compression	Yes	Yes	No
Fast truncate	Yes	No	No
Huffman encoding	Yes	Yes	No
Prefix and suffix compression	Yes	No	No
RLE compression	No	Yes	No

In Memory Concepts

- Cache
  - 说明
    - wt内部B-TRee数据结构共享
    - Cache缓存最近访问或者修改的数据，WT读取page数据时候会缓存。当缓存内存使用过多时候会把缓存数据清理，对于修改的数据则是通知checkpoint或者eviction线程把数据flush到磁盘
    - 不论是读还是写操作,内存的page和磁盘page之间需要转换
    - WT加载磁盘page到Cache，内部构建table来缓存磁盘page
    - wt中记录数据被删除时候会更新tombstone值
    - Cache大小是通过cache_size控制，Cache只能缓存基于Btree数据，其中包括索引、插入链表、更新链。
  - 数据结构
    - WT_CACHE — 缓存行为控制和淘汰策略设置，每个connection一个WT_CACHE
    - WT_CACHE_POOL
    - WT_COL — 可变列存数据缓存
    - WT_COL_RLE
    - WT_INSERT — 当有新插入的数据时候通过skiplist呈现的WT_INSERT表示
    - WT_PAGE — 内存形式的page包含磁盘page的镜像
    - WT_PAGE_MODIFY — 首次插入或者更新leaf page时候WT_PAGE_MODIFY会和cache中的WT_PAGE关联
    - WT_REF — 对于缓存中每个page的访问是通过WT_REF结构
    - WT_ROW — row数据缓存
    - WT_UPDATE — 对于row存储的更改WT_PAGE_MODIFY会使用一个WT_UPDATE数组来tracks变化，对于一个值多个更新是通过list来链到一起
  - Shared caches
    - WiredTiger支持单个进程在不同db之间的缓存共享
    - 不同数据库的connection都有单独大小的缓存大小设置共享后会针对不同connection划分cache
    - 设置共享cache后cache pool server线程会管理整个share cache.它会申请WT_CACHE_POOL针对share cache来全局设置和统计统计信息
    - cache pool server会定期唤醒和适配每个connection的缓存大小
- Eviction
  - 说明
    - 从Cache中处理或者删除旧数据同时把数据flush到磁盘
    - Eviction服务是通过WT_EVICT_QUEUE作为淘汰队列，队列每个元素是WT_EVICT_ENTRY
    - Eviction是由eviction server、0或者对个eviction worker线程组成
    - eviction server目标是查找需要淘汰的page,它会遍历tree查找可被淘汰的page,目前是采用LRU算法策略来进行淘汰
    - 目前通过wiredtiger_open或者WT_CONNECTION::reconfigure设置eviction_target来触发
  - 数据结构
    - WT_EVICT_ENTRY
    - WT_EVICT_QUEUE