LiveGraph

A Transactional Graph Storage System with Purely Sequential Adjacency List Scans

VLDB 2020

2001213077 陈俊达

2020年12月28日

目录

- 图数据存储相关技术以及LiveGraph的特点
- LiveGraph数据结构
- 单线程操作
- 多线程操作的并发控制
- 存储管理
- 性能测试
- 总结

目录

图相关计算任务的分类

交互式图数据管理

Transactional graph data management

- **更新**或者**查询**单个节点、边或者邻接表的信息
- 局部性、只更新或查询少量信息
- 要求**可交互**、操作**低延时**
- 要求存储系统拥有传统数据库系统的事务性特性
  - 并发控制
  - 持久性
- 举例：微博用户关注一个用户

图数据分析

Graph analytics

- 在一个图的某一个时间点的快照（snapshot）上对图进行分析任务
- 不会对图进行修改
- 分析时，图是只读的
- 性能高度依赖**邻接表扫描**操作
  - Adjacency list scans
- 举例：查询可能感兴趣的商品

现有数据结构

交互式图数据管理

- 传统DBMS使用的数据结构
  - B+树
  - Key-Value存储
- 支持动态数据修改
- 邻接表扫描速度慢
  - 大量使用指针，造成数据局部性不佳

图分析

- Compressed Sparse Rows（CSR）
- 邻接表线性存储，扫描速度快
- 是只读的，不支持低延时的数据修改操作

LiveGraph简介和特点

- 使用全新的数据结构：Transactional Edge Log（TEL）
- 同时支持**完全线性的邻接表扫描**和**低延迟的图更新操作**
  - 同时支持**交互式图数据管理**和**图数据分析**任务
- 支持了**并发控制**
  - 支持事务管理、恢复等
  - 支持多线程操作

目录

LiveGraph数据结构

- **VB**：存储点的属性（label）
- **TEL**：存储边的信息
- **Vertex Index**
  - 节点和最新VB的对应
- **Edge Index**
  - 节点和最新TEL对应
- 存储点和边的多个版本
- 每个VB指向之前的版本

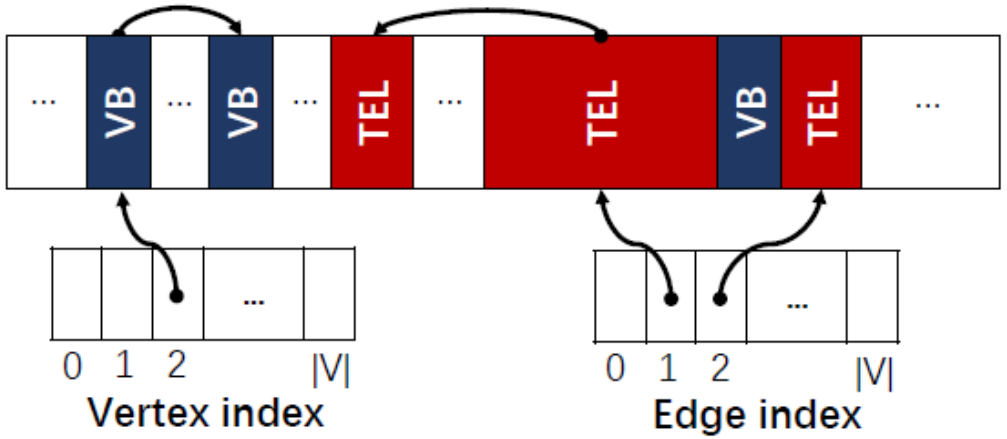
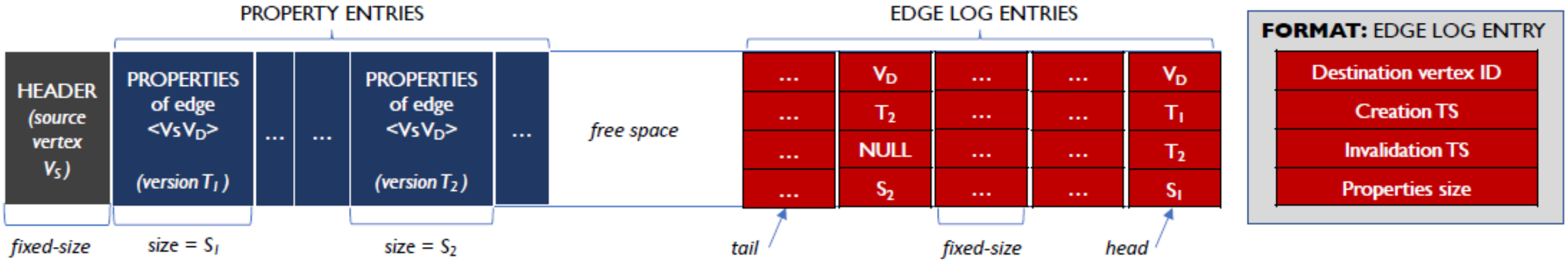


Figure 2: LiveGraph data layout. For simplicity, here we omit label index blocks and the vertex lock array.

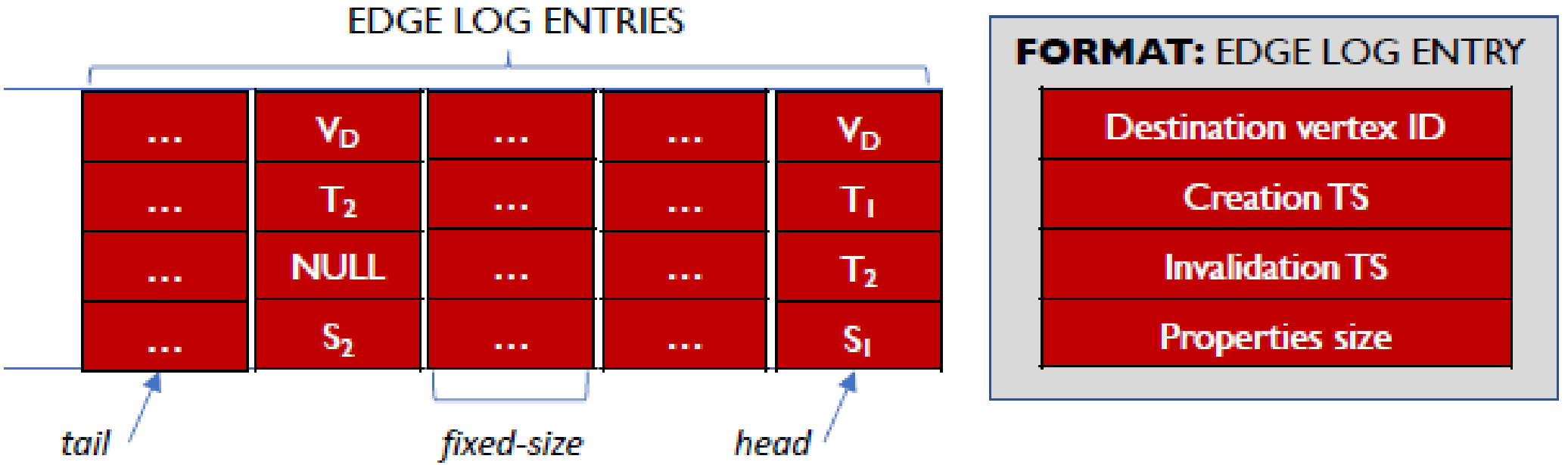
LiveGraph数据结构： TEL

- Transactional Edge Logs： TEL
- 一个TEL保存由**一个源节点**发出的边的信息
- 三个部分： **header**、**属性项**和**edge log**
- **属性项**存储边的label等信息



LiveGraph数据结构：Edge Log

- 每个log存储：**目标节点ID、创建时间、失效时间、属性项大小**
- Edge log连续存放，且大小与缓存对齐（可以放进一个缓存行）



目录

- 图数据存储相关技术以及LiveGraph的特点
- LiveGraph数据结构
- 单线程操作
  - 对节点、边的增删改查
- 多线程操作的并发控制
- 存储管理
- 性能测试
- 总结

单线程操作：对节点的操作

- **查询节点信息**：
- 1. 通过Vertex Index查找指向最新版本VB的指针
- 2. 通过指针访问最新版本（大多数情况）或者老版本的VB

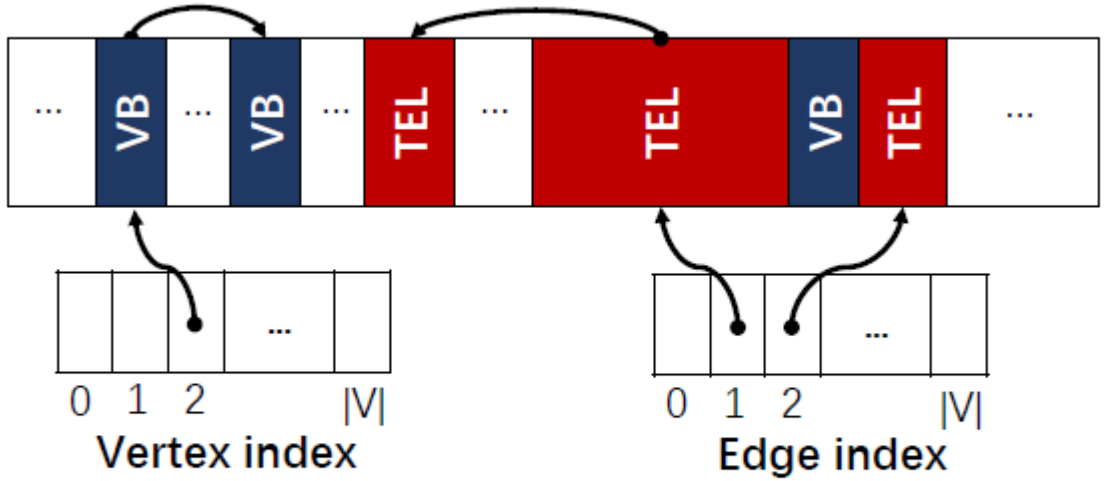


Figure 2: LiveGraph data layout. For simplicity, here we omit label index blocks and the vertex lock array.

单线程操作：对节点的操作

- **增加节点**：
- 获取新节点ID
- 增加vertex index、edge index、VB和TEL块

- **删除节点：**
- Since vertex deletions are rare, we leave the implementation of this mechanism to future work.

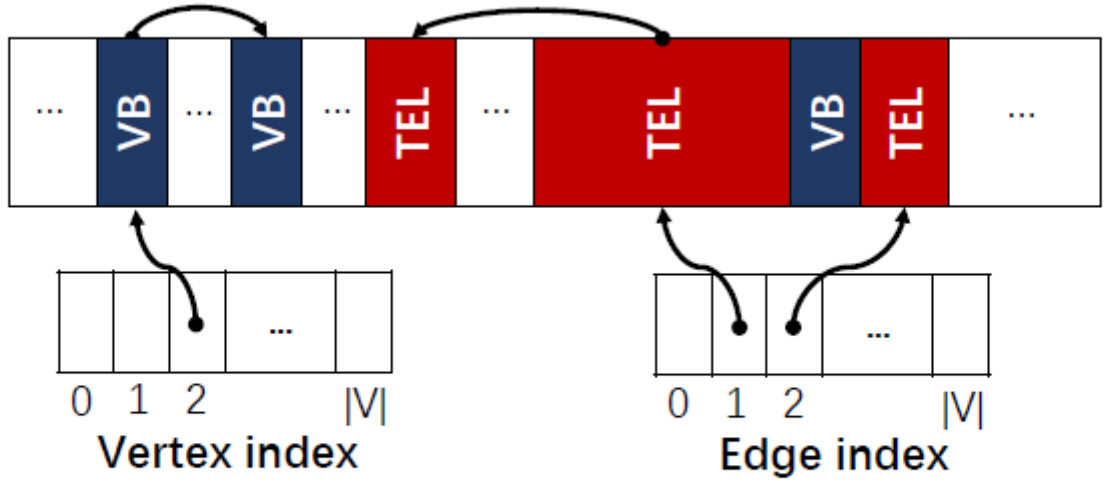
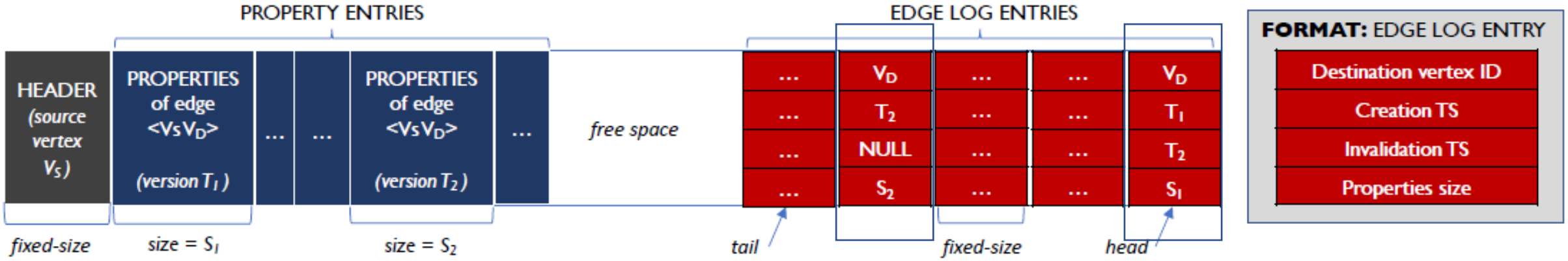


Figure 2: LiveGraph data layout. For simplicity, here we omit label index blocks and the vertex lock array.

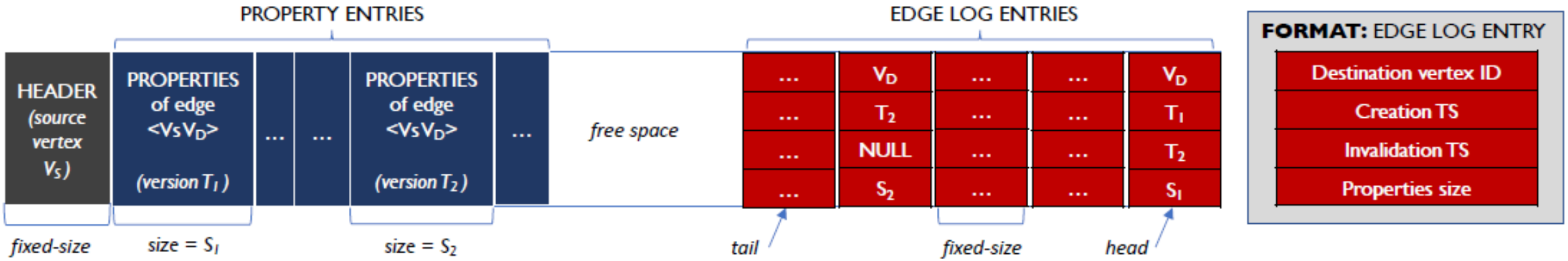
单线程操作：线性邻接表扫描

- Edge log中存储了节点的所有边信息
- 从后往前扫描一边edge logs就可以获取节点的所有边
- 如果要获取边的属性，再扫描一遍属性项即可
- 保存**多个版本**，使用**两个时间戳（创建时间、失效时间）**来O(1)地判断一个版本是否仍然有效



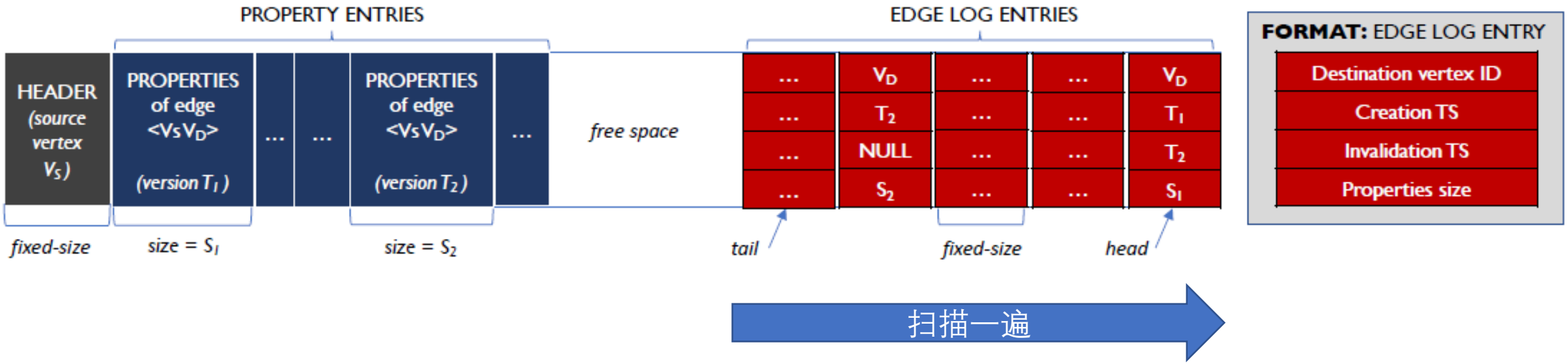
单线程操作：更新、删除、插入边

- 插入一个新的边：**追加**一个edge log即可：O(1)
- 更新边的属性、删除边：扫描一遍edge log并更新对应的时间戳
- 优化：header中有一个bloom filter，判断一次插入是否需要扫描
  - 如果目标节点是新的，bloom filter一定得出no，那就直接插一个log
  - 如果bloom filter给出yes，则需要扫描一遍edge log
- 边更新有时间局部性：越新插入的，越容易被修改



单线程操作：读取边

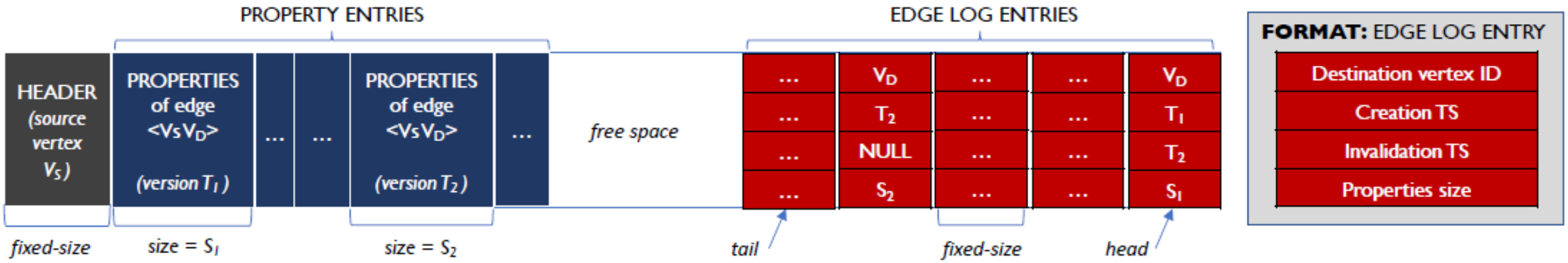
- 先查bloom filter
- 如果存在要读取的边，扫描一遍log获得边的信息
- 跳过失效的信息（当前时间大于失效时间）



目录

并发控制：特点

- 将并发控制所需要的信息（如时间戳）嵌入了数据结构中
  - 不需要其他数据结构辅助进行并发控制
- 多版本存储
- 读操作不需要获取锁，扫一遍即可获得当前时间点的数据的snapshot，不与其他读写操作冲突
  - snapshot isolation隔离等级

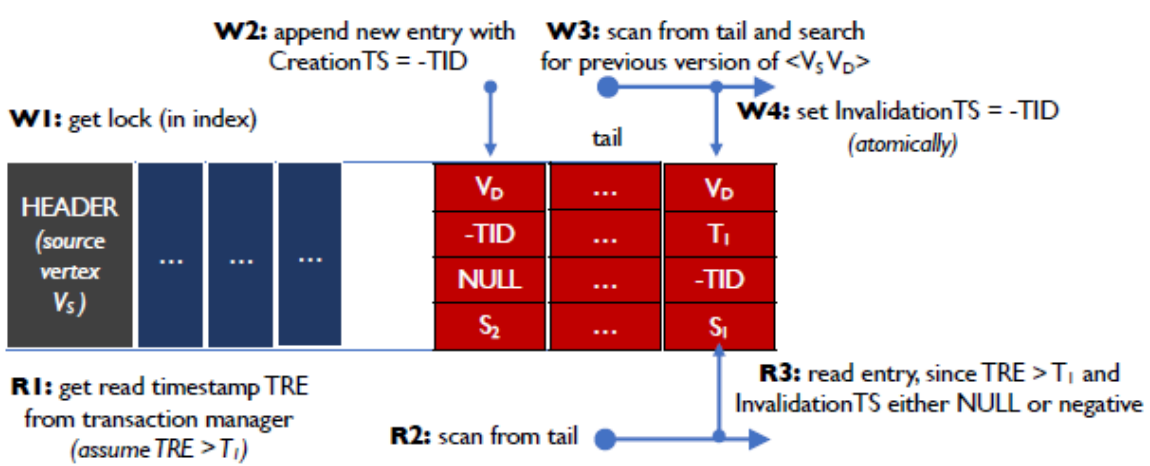


并发控制：事务处理

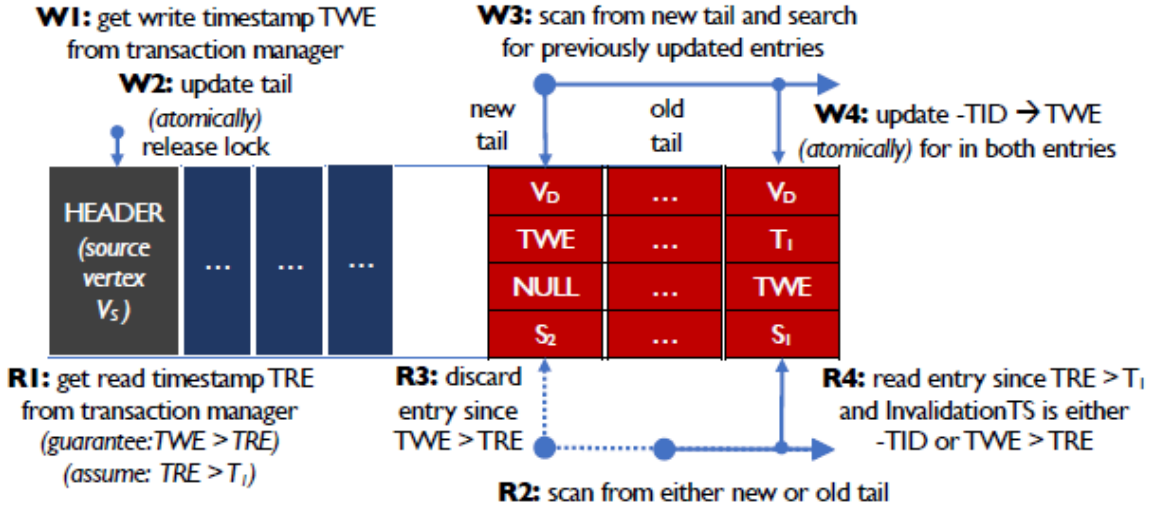
- 处理事务的线程（worker）池 + 一个事务管理线程
- 所有线程共享两个全局时间计数器（epoch counter），初始值为0:
  - **Global Read Epoch（GRE）**：读操作
  - **Global Write Epoch（GWE）**：写操作
- 每个线程具有局部时间计数器，确定事务相关的时间戳:
  - **Transaction-local Read Epoch（TRE）**：事务开始时设置为GRE
  - **Transaction-local Write Epoch（TWE）**：在commit时确定值
- 每个事务的ID：TID = 线程ID + 线程自己的事务计数器
  - 举例：线程001的第030个事务：001030

并发控制：写操作

- 写操作都是创建新TEL块，在提交之前新TEL块不能被其他事务看到
- **Work**：计算TID，获取锁，建新TEL块，创建时间和之前版本的失效时间设置为-TID
- **Persist：** GWE++，写WAL，把事务的TWE设置为GWE：确定事务提交的时间点
- **Apply：** 更新指针，释放锁，把所有块的-TID改成TWE，GRE++：使新事务可以看到新块



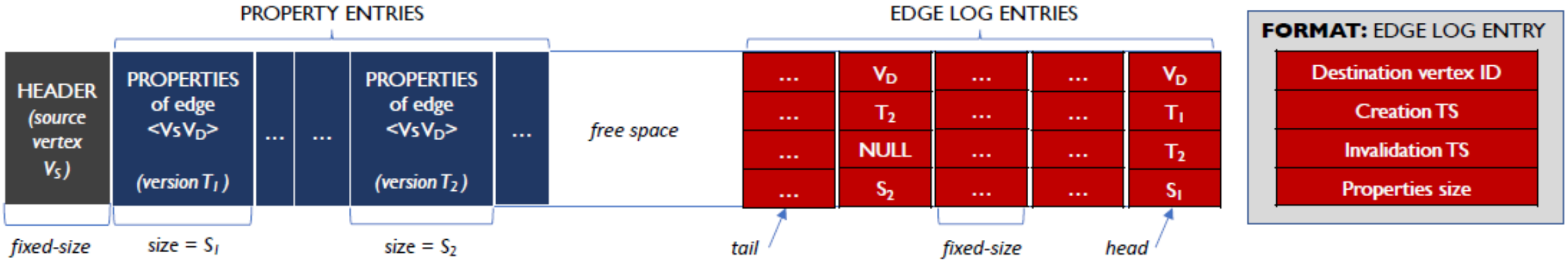
(a) Work phase



(b) Apply phase

并发控制：读取操作

- 从尾开始扫描TEL，只考虑满足下列条件中一个的log：
  - **(0 <= 创建时间 <= TRE) && (TRE < 失效时间 || 失效时间 < 0)**
    - 已经提交的且未失效的log
    - 失效时间<0：当前时间点没有失效，但是被还没提交的事务失效
  - **(创建时间 == -TID) && (失效时间 != -TID)**
    - 看到同一个事务之前创建的新log，忽略被当前事务失效的log



并发控制：锁、回滚和恢复

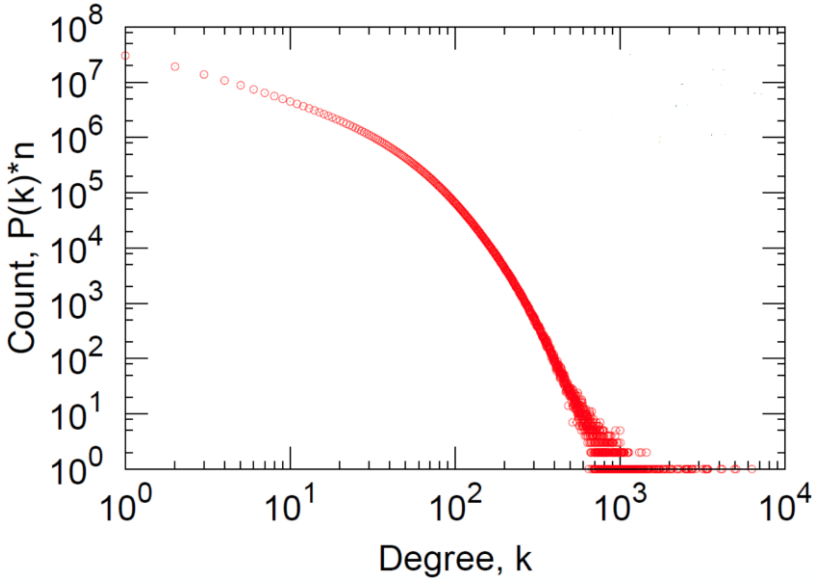
- 锁：
  - 锁的粒度：一个TEL（助教说是源节点）
  - 防止死锁是简单的timeout，超时就回滚
- 回滚
  - 把edge log的tail指针回拨一位：之后的新edge log将占据这个空间
  - 把-TID的时间改回NULL：恢复事务中设置为失效的log
  - 写操作设置失效时间时，如果遇到同一条边的、创建时间戳大于当前写操作的时间戳的块，则此事务需要回滚
    - 这个事务太慢了，之后进来的更新同一条边的事务已经commit了
- 恢复
  - 写WAL（Write ahead log）
  - 检查点
  - 周期性检查当前检查点的位置（哪个时间点之前的数据都已经写入了）并删除检查点之前的WAL

目录

存储管理：压缩

- 存储太多版本占用太多存储空间
- 需要周期性地压缩、删除历史版本：不会被使用的太旧的版本
- **太旧的版本**：当前所有事务的最小的TRE之前**失效**的版本
- 流程
  - 1. 执行压缩的线程访问所有线程的正在进行的事务的TRE，来确定哪些log是**现在以及以后都不可见的**
    - 最小的TRE就是最老的版本，比最小的TRE还小的log都没用了
  - 2. 分配新存储块，将所有**可见的块**移动到新存储块中
  - 3. 更新Vertex/Edge Index指针

存储管理：存储分配

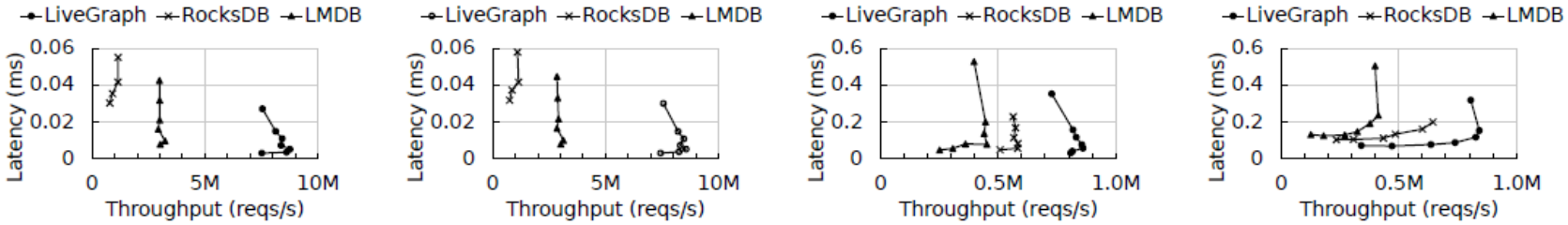


- TEL和VB存放在连续存储块中：**缓存友好性**
- 思想：
  - 真实生产数据，点的边的数量遵循指数分布
  - 借鉴Linux内存分配系统buddy算法
- 一个TEL块从64字节（36字节header + 28字节log）开始，存储一条边
- TEL的大小按两倍速度增大
- 一个数组L：L[i]存储(2^i) * 64字节大小的可用块的位置（i=0-57）
- 每个线程自己维护**只有自己能用的（private）**小块存储的的位置，全局维护大块的位置
  - 小块分配得多

目录

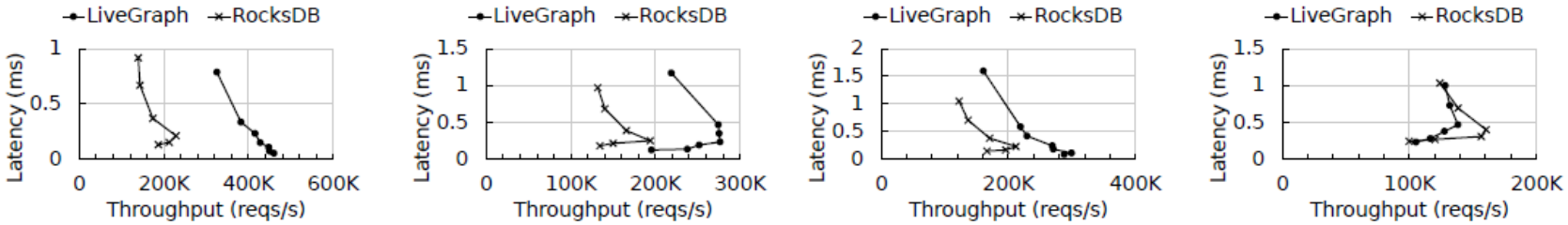
性能测试

- 吞吐量
- LiveBench： FB的数据
- TAO：99.8%读
- DFLT： 69%读



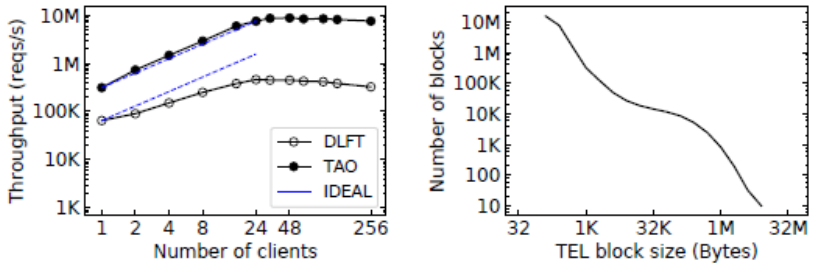
(a) Optane in memory (b) SSD in memory (c) Optane out of core (d) SSD out of core

Figure 5: TAO throughput and latency trends, with different number of clients and memory hardware



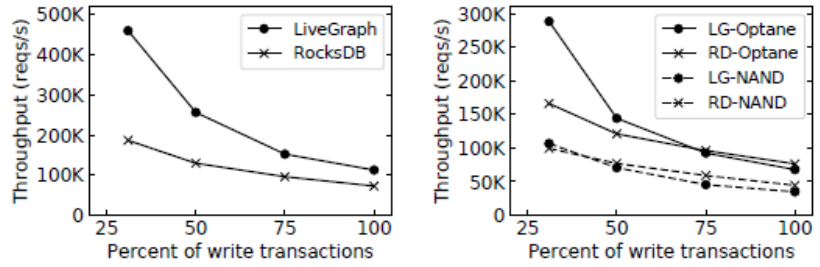
(a) Optane in memory (b) SSD in memory (c) Optane out of core (d) SSD out of core

Figure 6: DFLT throughput and latency trends, with different number of clients and memory hardware



(a) LiveGraph scalability (b) Block size distribution

Figure 7: LiveGraph scalability and block size distribution



(a) In memory (b) Out of core

Figure 8: LinkBench throughput with varying writing ratio

目录

- 现有图存储系统的分类、问题以及LiveGraph的特点
- LiveGraph数据结构
- 单线程操作
- 多线程操作的并发控制
- 存储管理
- 性能测试
- 总结

总结

- LiveGraph引入了Transaction Edge Log数据结构
- 同时支持**完全线性的邻接表扫描**和**低延迟的图更新操作**
- 性能测试证明了LiveGraph在多种图任务中都具有良好的性能
- 后续工作
  - 应用当前RDBMS所使用的对新硬件更友好的并发控制和存储方法
  - 当前LiveGraph只是单机的，支持分布式事务处理
  - 多版本存储使得LiveGraph可以用于temporal graph processing
    - 数据随着时间演变

谢谢！