DTCC 2010-2022

第十三届中国数据库技术大会

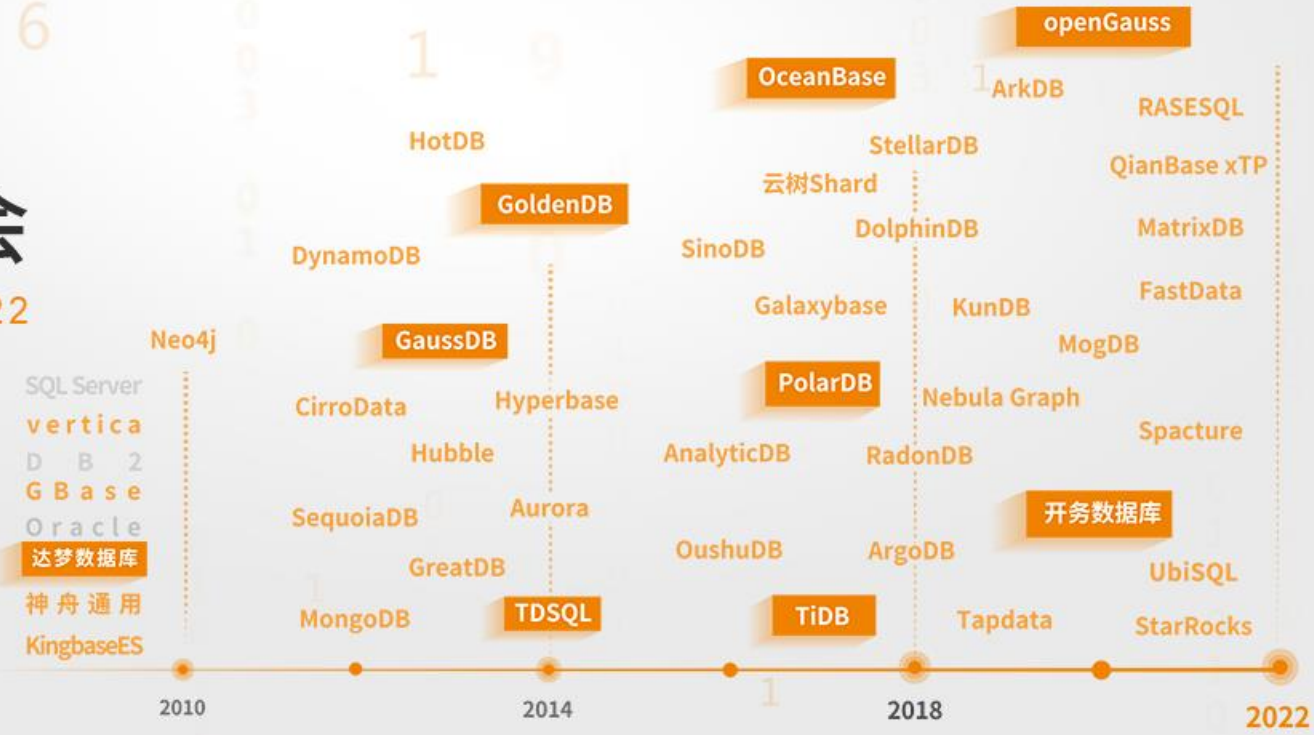
DATABASE TECHNOLOGY CONFERENCE CHINA 2022

数据智能 价值创新

IT168.com ChinaUnix.net ITPUB

线上直播 | 2022/12/14-16

数据来源：数据库产品上市商用时间



字节跳动图数据库架构演进

—— 索引和执行优化

陈超 字节跳动 研发工程师

简介

架构

关键问题

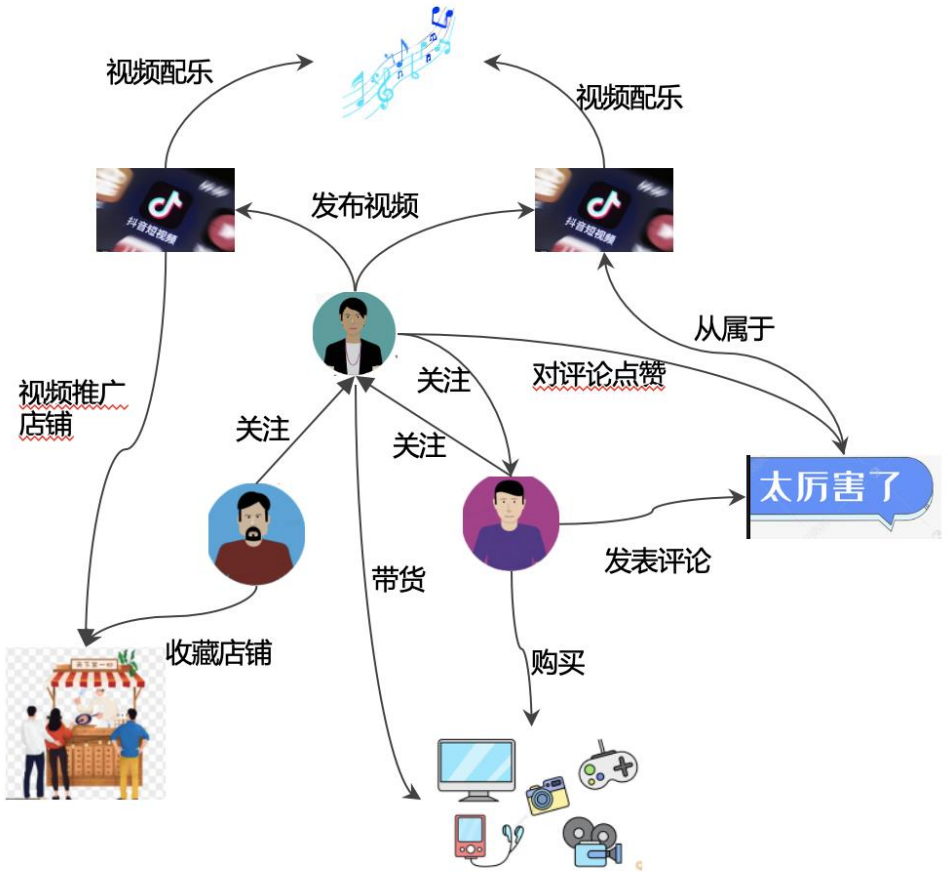
1.1 ByteGraph 可以做什么

- 字节有哪些业务数据呢?
  - ✓ 用户信息、用户关系
  - ✓ 内容（视频、文章、广告等）
  - ✓ 用户和内容联系（点赞、评论、转发、点击）

- 使用图表达业务场景的优势
  - ✓ 建模直观简洁
  - ✓ 挖掘数据关联

- ByteGraph 特点
  - ✓ 高吞吐
  - ✓ 低延迟
  - ✓ 最终一致
  - ✓ 兼容 Gremlin

- ByteGraph 学术论文已被 VLDB-2022 收录



1.2 Gremlin 查询接口

- Gremlin 简介
  - ✓ Gremlin 是一种图灵完备的图遍历语言
    - ✓ 相较 cypher 等查询语言，功能更全面，上手较为容易，使用更加广泛
  - ✓ 主流云厂商的图数据库都提供了对 Gremlin 的支持，ByteGraph 目前支持**一个子集**

- 数据模型
  - **有向属性图**
  - 点和边上都可以携带**多属性**，支持动态加减属性列

- 举例
  - ✓ 用户A所有一跳好友中满足粉丝数量大于100的子集
    - ✓ g.V(vertex(A.id, A.type)).out('好友'). where(in('粉丝关注').count().is(gt(100))).toList()
  - ✓ 搜索知识图谱：求中国出生的、配偶是日本人的女明星
    - ✓ g.V().has（‘出生地’，‘中国’).has（‘性别’，‘女’)and(out（‘配偶’).has（‘出生地’，‘日本’), out（‘职业’，‘明星’))
  - ✓ 电商风控图谱：求昨天内 uid=111 用户的订单中，给 id 为 222 的店的订单数
    - ✓ g.V().has（‘uid’, 111).out（‘订单’).groupCount().by（‘id').unfold().where(select(keys).is(222)).select(values)

1.2 Gremlin 查询接口举例-UGC场景

- 基于发文点赞关注的场景构图，查询举例
  - 用户C关注的作者今年收到了多少点赞

```
g.V(vertex(C, 用户)).
  out('关注').                          // 查询C关注的作者
  out('发文').                          // 查询作者的所有发文
  inE('点赞').has('tsUs', gt( 20190712))  // 查询每个发文被点赞的边，并按照时间tsUs属性过滤
  count()                               // 计数点赞边的总数量
```

  - 用户C关注的大V作者，按照粉丝数量倒排，取top10

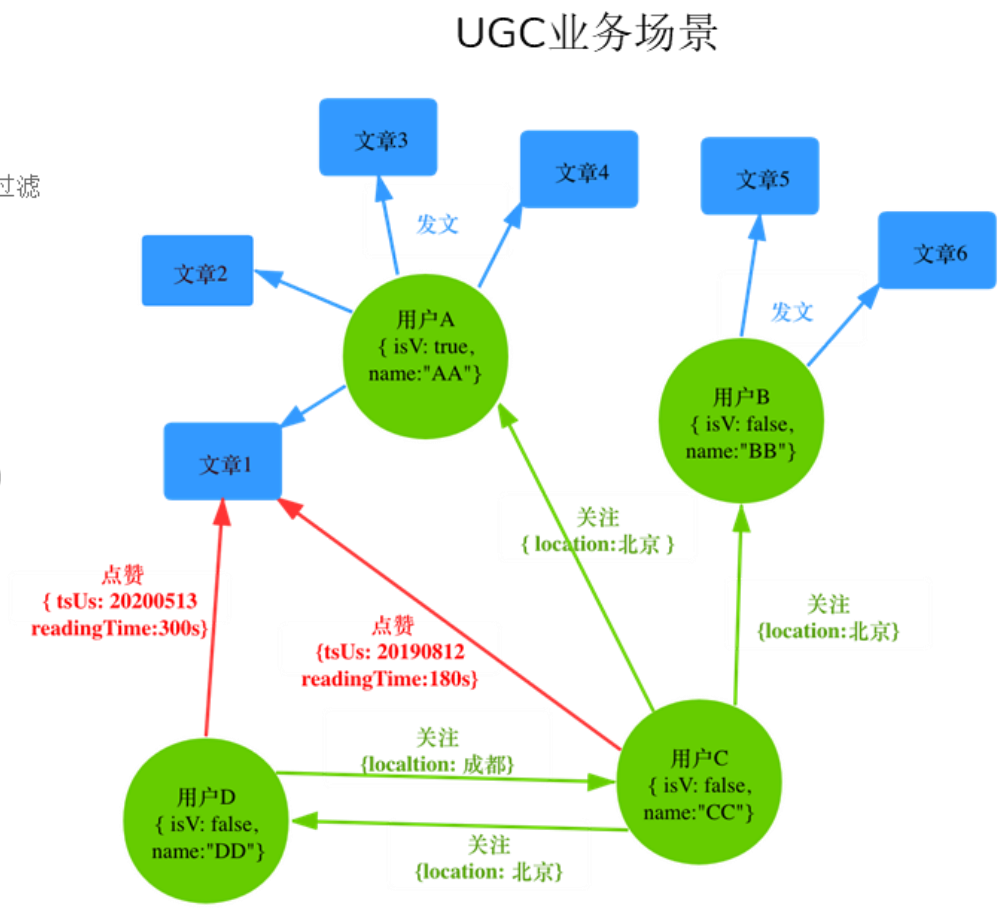
```
g.V(vertex(C, 用户)).
  out('关注').has('isV', true).          // 查询C关注的作者，并且筛选其中的大V作者
 order().by(inE('关注').count(), desc).  // 按照作者的粉丝数量，对上面的大V作者倒排序
 limit(10).                             // 取排序后的 top 10 大V作者
 values("name")                         // 返回大V作者的姓名name
```

  - 用户C在2020年关注的大V作者，按照*关注时间*倒排，取top10

```
g.V(vertex(C, 用户)).
  outE('关注').                          // 查询用户C关注的作者（边）
  has('tsUs', gte(20200101)).           // 过滤，只选取关注时间在2020年的关注边
  where(otherV().has('isV', true)).     // 过滤，只选取关注的大V作者
  order().by('tsUs', desc).             // 按照边上的关注时间tsUs，做倒排
  limit(10).                            // 倒排后 取top10
  otherV().values('name')               //
```

  - 用户C在2020年关注的大V作者，按照*关注地点*倒排

```
g.V(vertex(C, 用户)).
  outE('关注').                          // 查询用户C关注的作者（边）
  has('tsUs', gte(20200101)).           // 过滤，只选取关注时间在2020年的关注边
  where(otherV().has('isV', true)).     // 过滤，只选取关注的大V作者
  order().by('location'', desc).        // 按照边上的关注时间tsUs，做倒排
  otherV().values('name')
```



1.2 Gremlin 查询接口举例-UGC场景

- 支持跨集群、跨表查询：withTable() 用来指明数据源

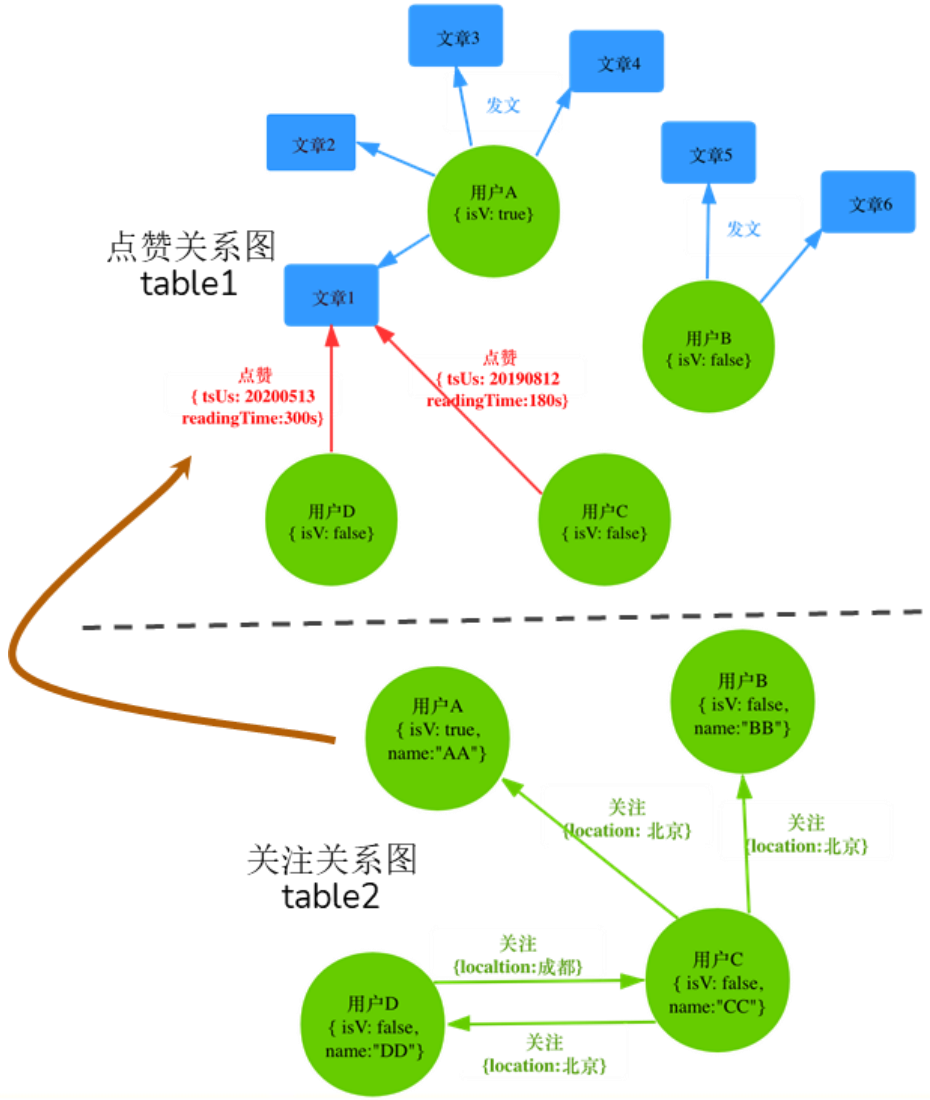
  假设点赞关系图和关注关系图分别存储在两个表table1和table2中

  - 用户C的好友（互相关注）喜欢的文章列表

```
g.V(vertex(C, 用户)).
  withTable('table2').                  // 切换到table2中
  double('关注').                       // 在table2 中使用双向边语法double查询好友
  withTable('table1').                  // 切换到table1中
  out('点赞').                          // 在table1 中查询好友点赞过的文章
  toList()
```

  - 用户C和C的好友都点赞过的文章列表

```
g.V(vertex(C, 用户)).
  withTable('table1').
  out('点赞').store('articals').        // 查询C点赞过的文章，并存储在集合articals中
  count().local(
    g.V().has('id', C).has('type', 用户).
    withTable('table2').                // 切换到table2中
    double('关注')                      // 在table2中查询C的好友
    withTable('table1').                // 切换到table1中
    out('点赞').                        // 在table1中查询好友点赞过的文章
    where(within('articals'))           // 上步查询到的文章中，只保留articals中出现过的文章
  ).toList()
```



1.3 ByteGraph 业务介绍

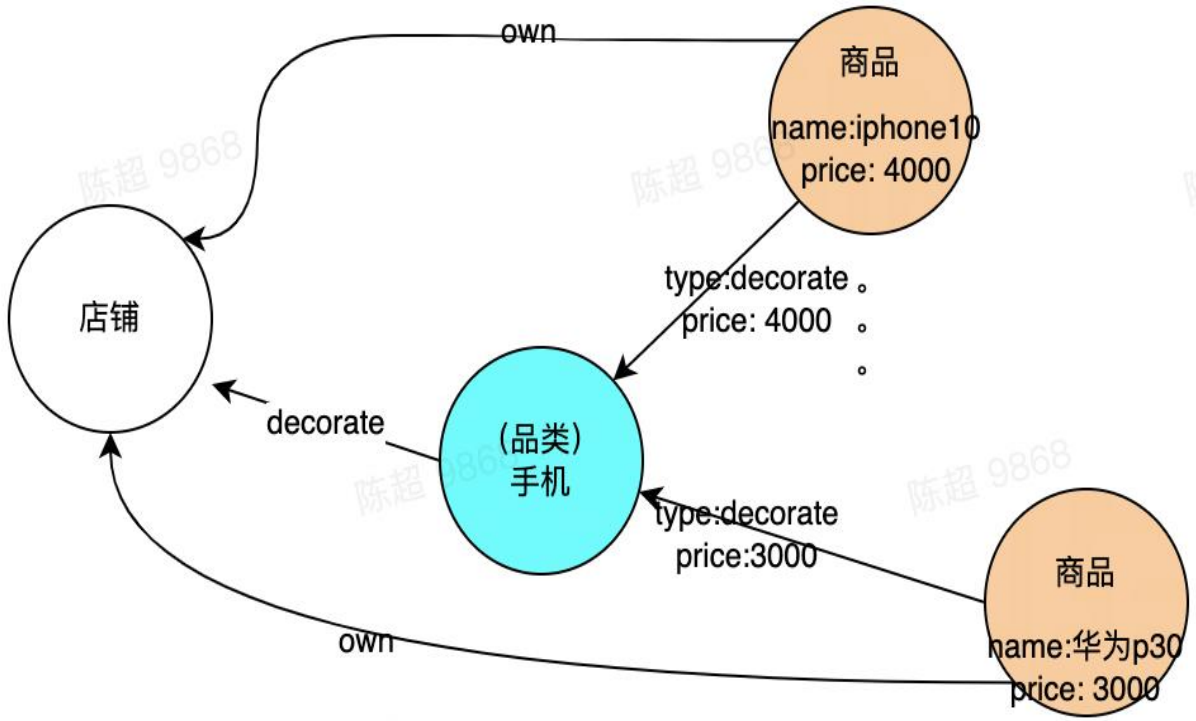
1000+ 业务集群

业务场景分类	分类	图模型	查询举例：
抖音用户关系的服务端在线存储	社交网络关系	点：用户 边：用户之间关系	关注/粉丝列表，关注关系判断
抖音推荐： 推人、推视频	社交推荐	点：用户、视频 边：用户关系(多种）、用户发文	好友的好友等多度查询
知识图谱： 搜索百科、教育、电商	知识图谱	点: 各种实体（课程、知识点，商品） 边：实体之间逻辑关系（报名课程、掌握知识点、收藏商品）	实体推荐 某个人在某个商铺昨天的订单数等
lib库、项目、线上服务之间的网状关系	IT系统	点: lib库、repo、线上服务 边：点之间依赖关系	给定某个库的确定版本，求所有依赖这个版本的库

1.3 ByteGraph 业务介绍-店铺/商品关系

- 构图
  - ✓ 店铺拥有的商品，店铺拥有的品类，商品所属的品类

- 查询
  - ✓ 查询店铺下有哪些品类
  - ✓ 筛选某个品类下价格处于xx-xxx之间的某种商品。商品具有“价格属性”。但是为了加快查询，在商品到品类之间等边上冗余了商品点 price 属性

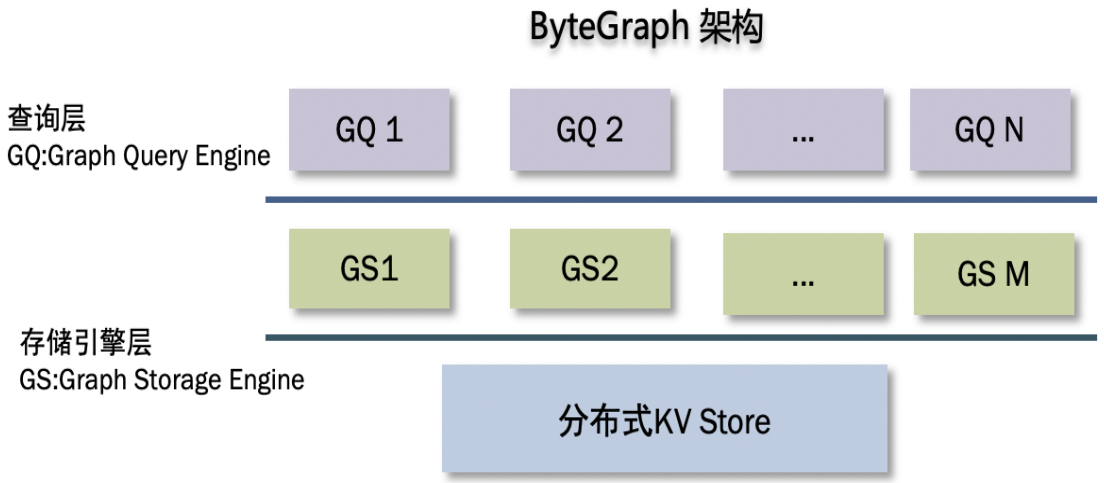


简介

架构

关键问题

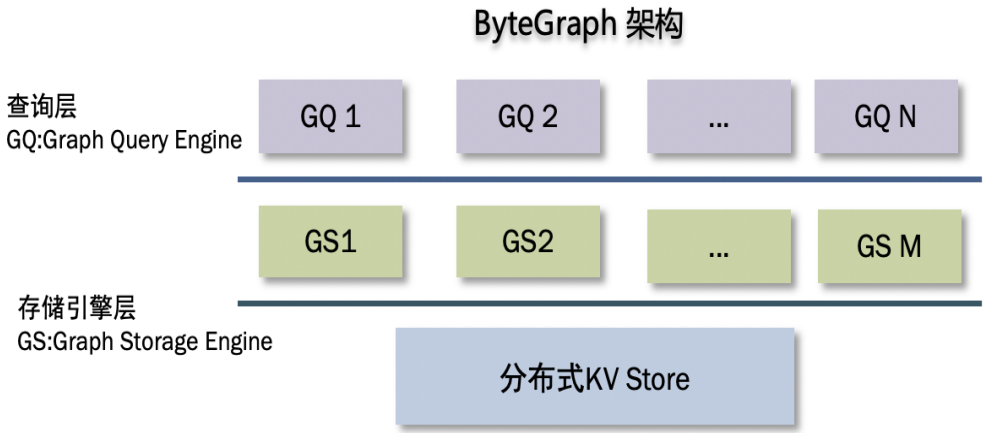
02. ByteGraph 架构-整体架构



- 整体分为三层，每层由多个进程实例组成集群
- 查询层和内存存储层可以混合部署或独立部署
- 分布式 KV 为可插拔

02. ByteGraph 整体架构

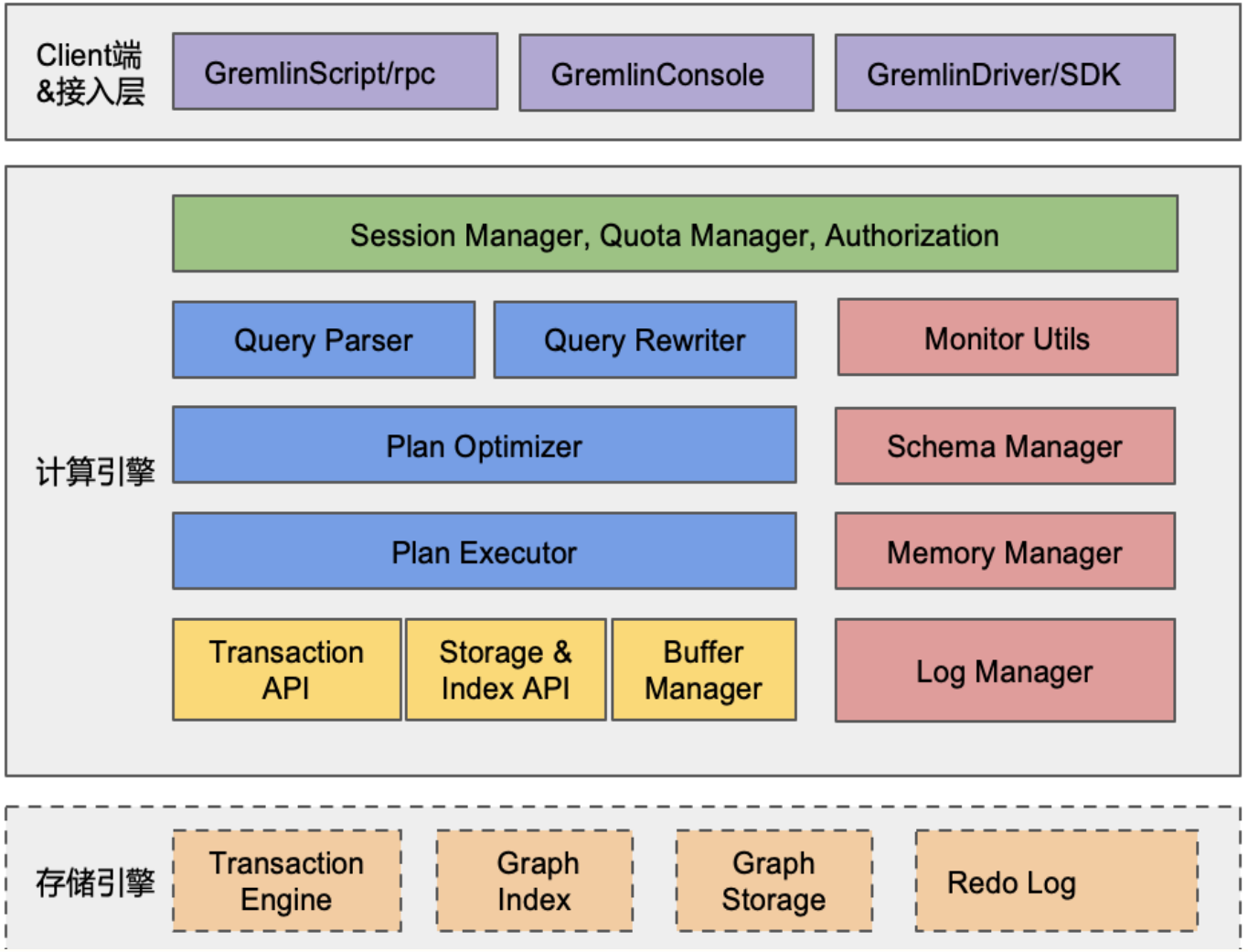
- 查询引擎层（GQ）
  - ✓ 用户session管理，服务的proxy 层
  - ✓ gremlin 查询语言的parser
  - ✓ 分布式的数据库执行器executor
  - ✓ GS层数据分布路由模块
  - ✓ go语言实现
- 存储引擎层（GS）
  - ✓ 负责把全图数据切分成子图（partititon），完成partition的存储和缓存
  - ✓ 负责partition分布策略
  - ✓ 负责partition的内存组织和磁盘组织方式
  - ✓ 实现WAL，支持事务
  - ✓ C++编写，追求极致性能



- 磁盘存储层
  - 负责管理磁盘资源
  - 目前依赖第三方的分布式 KV 系统，下个版本会自研图原生存储

2.1 ByteGraph 架构 - 查询引擎

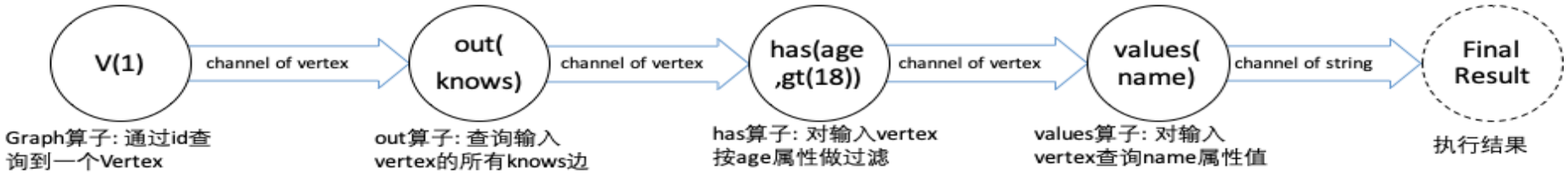
- Client端 & 接入层设计
  - ✓ Go、C++、Python、Java SDK
- Query Parser & Rewriter
  - ✓ 将 Gremlin 解析成语法树
  - ✓ 将语法树改写为执行计划树
- Plan Optimizer
  - ✓ 基于规则的优化（RBO）
  - ✓ 基于代价的优化（CBO）
- Plan Executor
  - ✓ Push 模式的 pipeline 驱动器
  - ✓ 支持行式 & 列式执行



2.1 ByteGraph 架构-查询引擎层 - 查询引擎

查询层(GQ)和MySQL的SQL层一样，主要工作是做查询的解析和处理；其中“处理”可以分为以下三个步骤：

- Parser 阶段:
  - ✓ 一个手写的递归下降解析器，将查询语言解析成一个查询语法树
- 生成查询计划：
  - ✓ 把步骤1中的查询语法树按照一定的查询优化策略（RBO & CBO)转换成执行计划
  - ✓ 为了减少解析和优化的开销，我们支持了查询计划缓存
- 执行查询计划：
  - ✓ 和Graph Storage 层(GS）交互，完成查询计划；需要理解存储层数据分Partition的逻辑，找到数据，下推算子，merge查询结果，完成查询
- 下图是一个查询的执行流水线
  - ✓ g.V().has(‘id’ , 1).has(‘type’ , person).out(‘knows’).has(‘age’ , gt(18)).values(‘name’)



2.1 ByteGraph 架构-查询引擎层 - 查询优化

- 基于规则的优化（rule based optimization）
  - ✓ Apache tinkerpop 的 gremlin 开源实现中包含了一些简单的优化规则
    - ✓ outE.inV => outV
  - ✓ 过滤器合并、算子下推，将尽量多的计算下推到底层，减少数据传输
    - ✓ outE.has.has => outEWithFilter
  - ✓ operator fusion，将部分 step 序列进行融合，成为更高效的单一 step
    - ✓ outE.count => edgeCount
  - ✓ 数据预取和子查询消除，来减少不必要的查询开销，以及提升查询并发度
    - ✓ out.where(in.count.is(gt)) => out.countprefetch.where(in.count.is(gt))
- 基于代价的优化（cost based optimization）
  - ✓ 统计信息：点的出度
  - ✓ 代价：网络通信成本 + 计算成本 + 磁盘读取成本
  - ✓ 举例：社交场景查询，查询 “A 关注的哪些人也关注了 B”
    - ✓ plan A: expand + expand
    - ✓ plan B: expand + broadcast join

2.1 ByteGraph 架构-查询引擎层 - 图分区算法

减少网络通信次数，具体选择的分区算法与 workload 强相关。

- brute force 哈希分区
  - ✓ 根据起点 + 边类型进行一致性哈希分区
  - ✓ 在大部分查询场景，尤其是一度查询场景下足够好
- 知识图谱场景
  - ✓ 边类型极多，每种类型的边数量相对小，单机可以容纳
  - ✓ 根据边类型进行哈希分区，将同种边类型数据分布在一个分区内
  - ✓ 大幅度降低查询中多度查询的扇出请求数量，降低延迟
- 社交场景
  - ✓ social hash 算法，由 facebook 2016 年论文提出，通过离线计算尽量将有关联的数据放置在同一个分片内
  - ✓ 例如：以点 1 为起点做二度查询，图 2 的分区方式就会比图 1 减少一次访问实例的开销。
  - ✓ 降低查询中多度查询的扇出请求数量，降低延迟

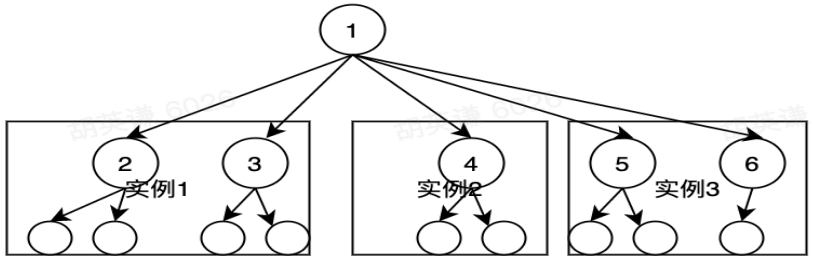


图1

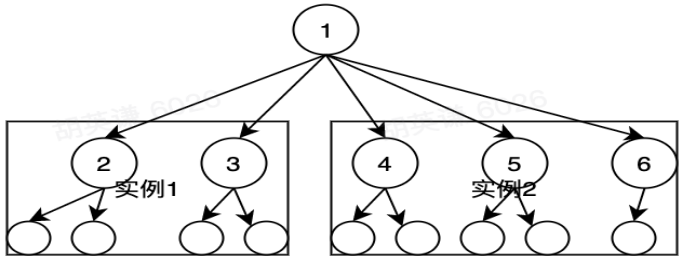
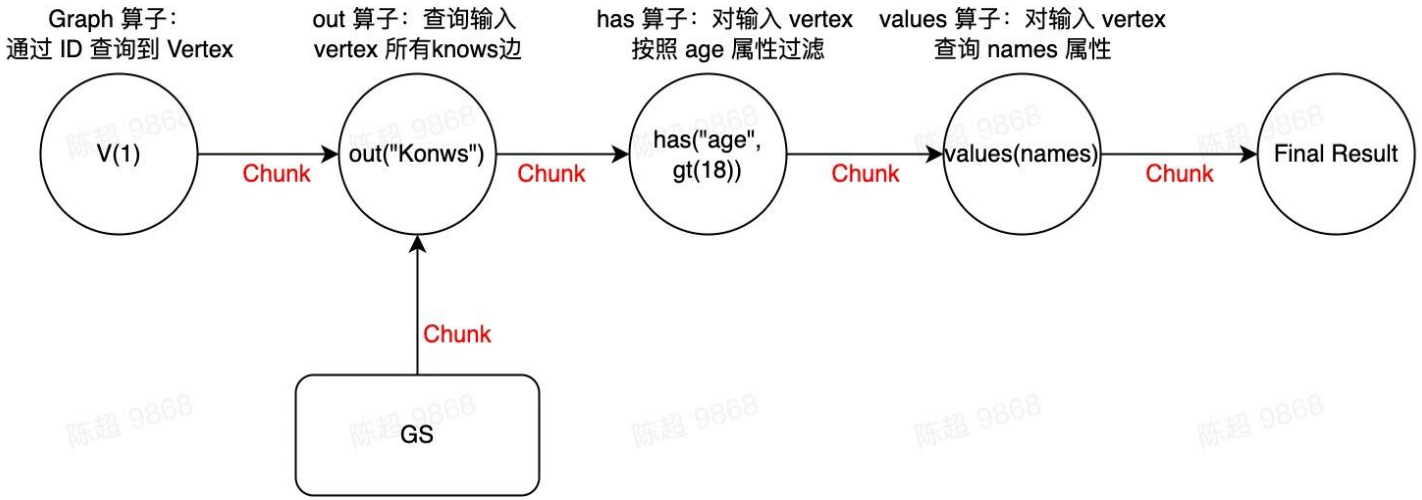


图2

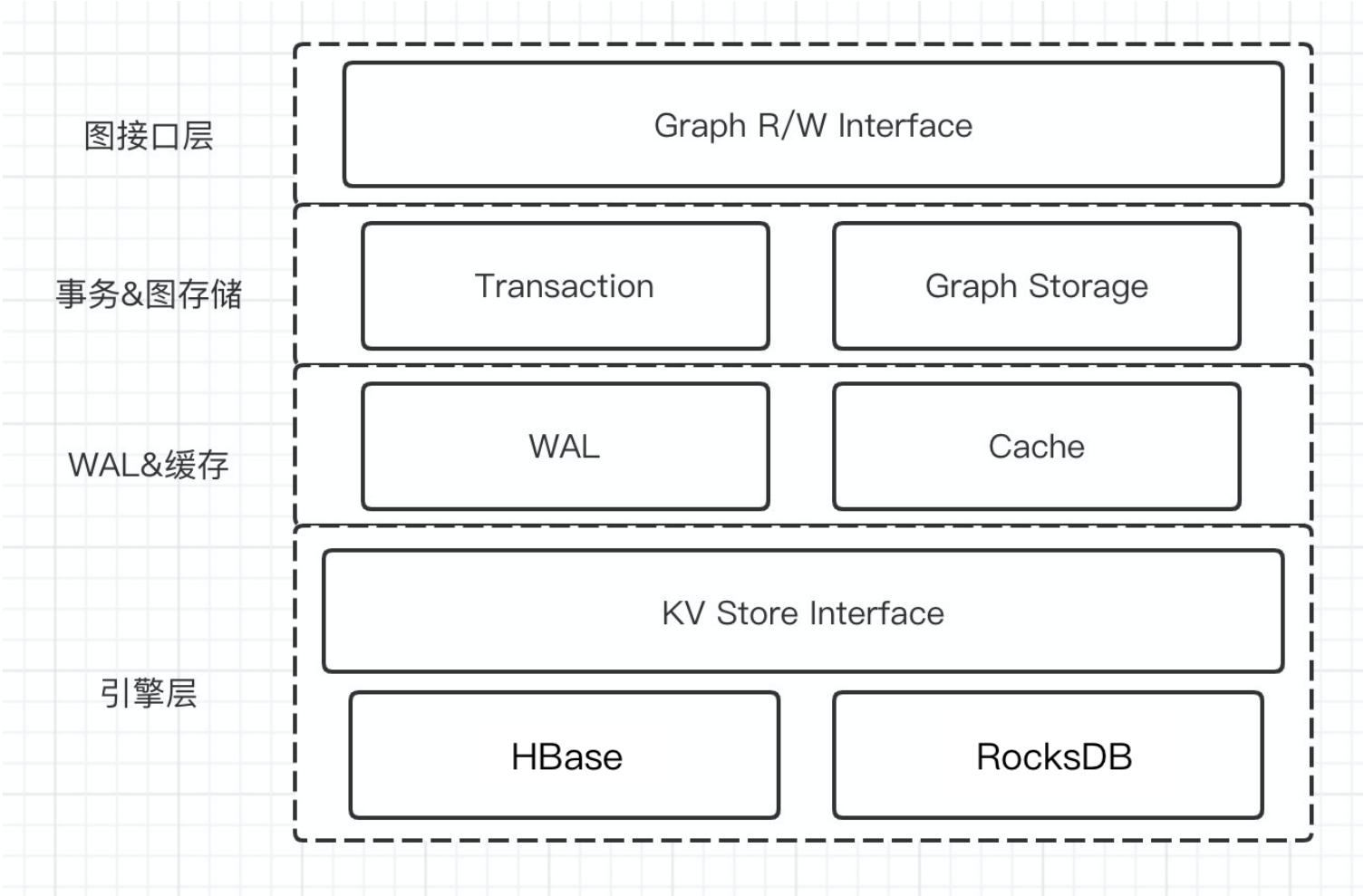
2.1 ByteGraph 架构-查询引擎层 - 列式计算

- **查询层列式计算**
  - ✓ 存储层按照列式返回结果
  - ✓ 算子支持列式计算
  - ✓ 算子间通过 Chunk 传输数据
- **性能**
  - ✓ 整体查询性能翻倍



2.2 ByteGraph 架构-存储引擎层

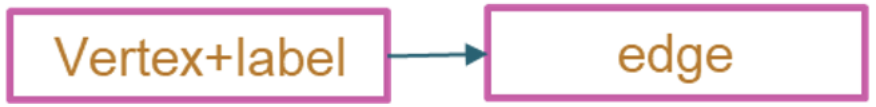
- 存储结构
- 多属性结构
- 日志管理
- 缓存实现
- 整体结构图



2.2 ByteGraph 架构-存储引擎层 - 存储结构

如何基于 KV 系统构建一个图结构?

- **一个 KV 对一条边:**
  - ✓ 实现简单
  - ✓ 写入放大（write amplification）较小，适合写入场景
  - ✓ 使用 kv Scan 实现一度邻居查询，某些场景下性能退化

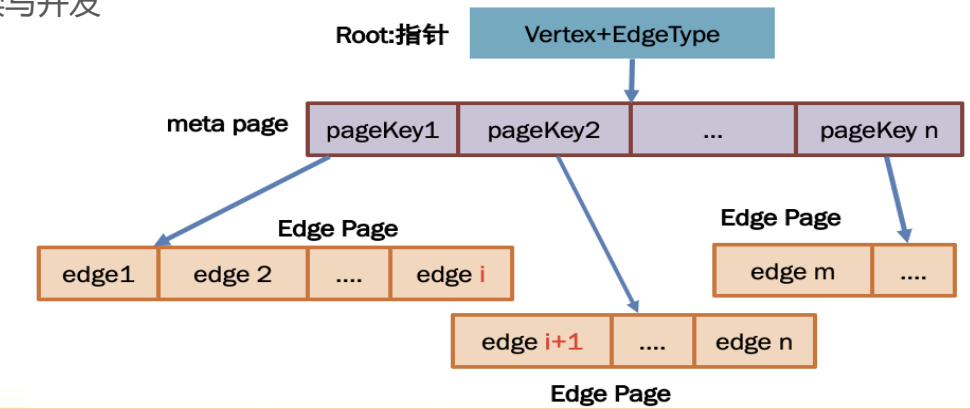


- **一个 KV 保存一个起点的所有边:**
  - ✓ 实现较为简单
  - ✓ 写入放大较大，读取一次寻址，适合重读场景
  - ✓ 无法处理超级顶点写入的问题

- **多个 KV 对组成B树等结构保存起点的所有边（ByteGraph 的选择）:**
  - ✓ 实现较为复杂
  - ✓ 可以根据配置来灵活平衡读放大和写放大
  - ✓ 可以解决超级顶点问题
    - ✓ 字节跳动内部场景大量存在超级顶点

2.2 ByteGraph 架构-存储引擎层 - 存储结构

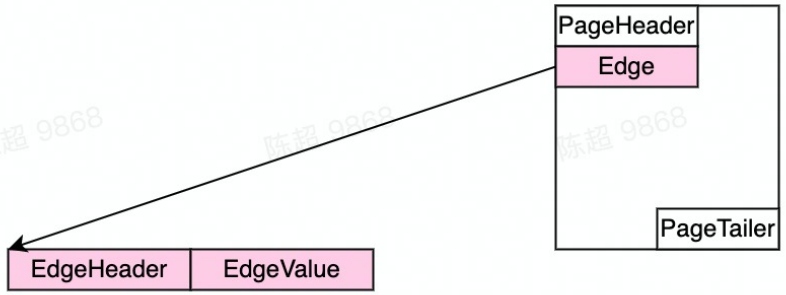
- B树细节结构:
  - 某起点的同一个边 type 的所有终点是一个存储单元
  - 一级存储（点的出度少于多少阈值）
    - ✓ 起点 ID + 起点 Type + 边 Type + 方向 作为 key
    - ✓ 同一起点相同 type、相同方向的所有边聚合成一个 value
  - 多级存储（点的出度超过阈值）
    - ✓ 所有边均匀切分成 EdgePage，并分配对应的 Partkey，所有 Partkey 组层 Meta 数据
    - ✓ MetaPage 整体作为 Value 存储，（点，边Type）-> (PartKey1, PartKeys2, ...)
    - ✓ EdgePage 的存储格式和一级存储类似，（PartKey）-> (Edge1, Edge2, ...)
    - MetaPage 可以有多级，和 EdgePage 整体组成B树，通过 COW 实现读写并发
  - 一级存储和多级存储之间可以动态转化

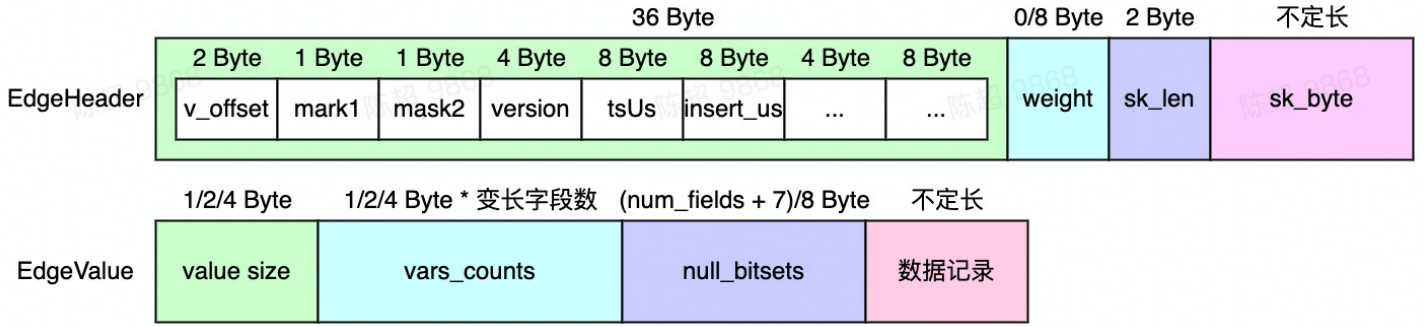


- 分布式集群存储:
  - ✓ 通过多种可选的图划分算法，将全局数据划分到多个 Shard 中

2.2 ByteGraph 架构-存储引擎层 - 多属性结构

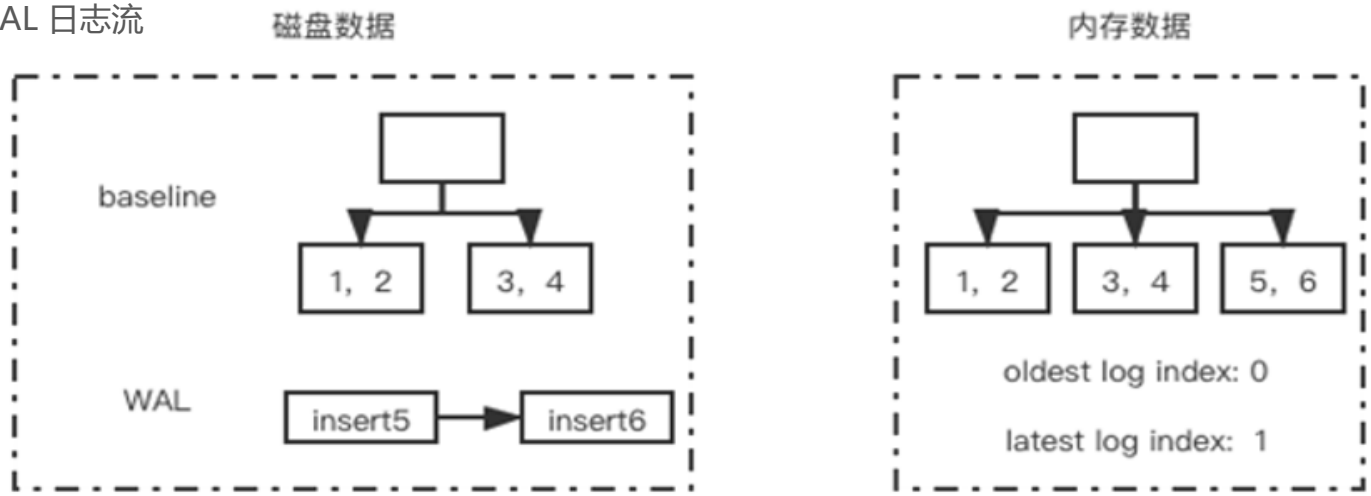
- 多属性数据结构
  - ✓ 连续紧凑，访问速度快
  - ✓ Header中保存schema版本：快速增加属性
  - ✓ 快速访问终点ID/Type、Weigh、tsUs 等边固有属性
  - ✓ 支持 Int/String 两种终点 ID/Type





2.2 ByteGraph 架构-存储引擎层 - 日志管理

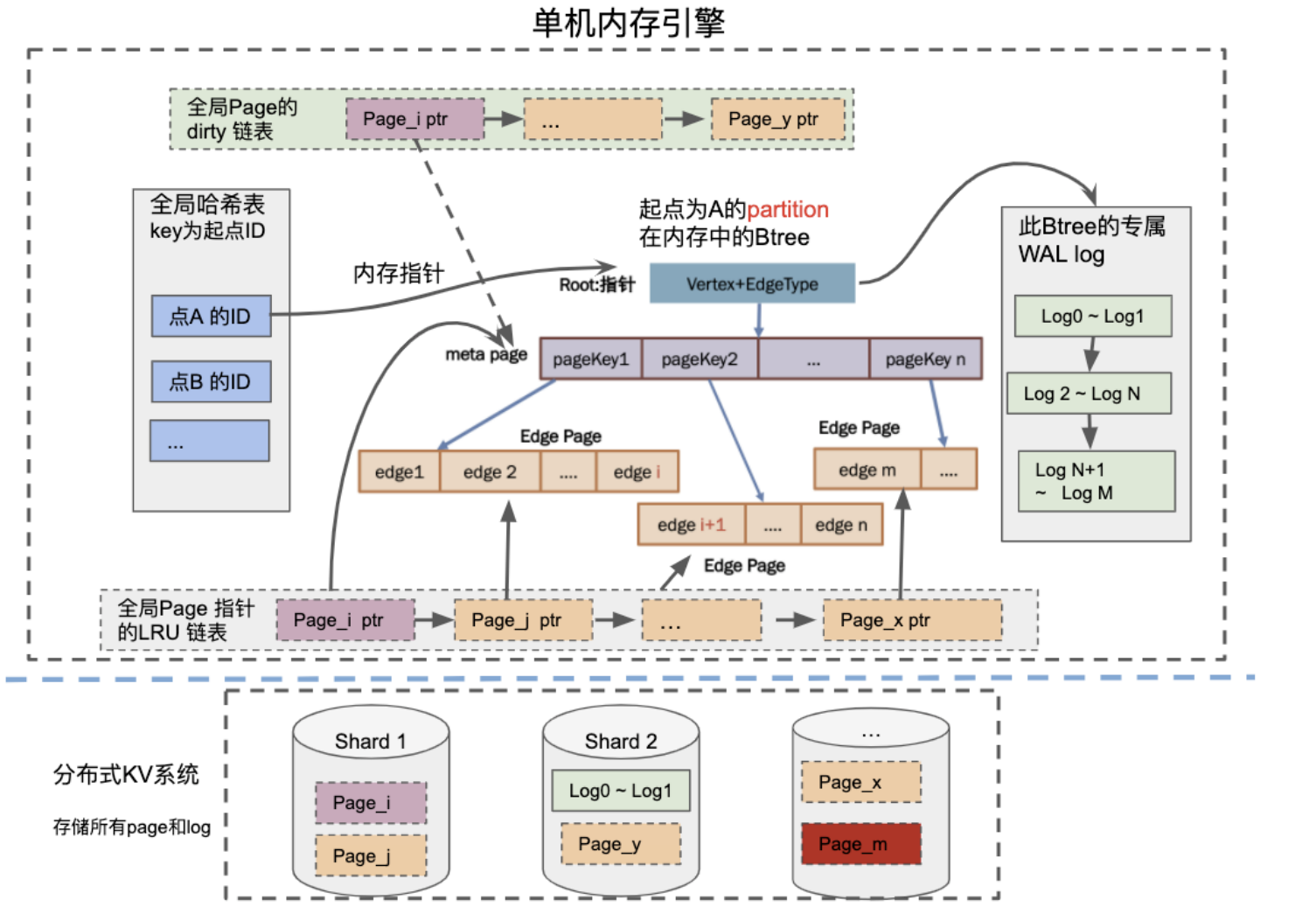
- 单个起点 + 边类型组成一颗 Btree:
  - ✓ Btree 每个节点是一个 KV 对
- Btree 完整性:
  - ✓ 由于一颗B树由多个 KV 对组成，且不假设底层 KV 系统支持事务，所以需要复杂的刷盘策略来确保B树结构的内部完整性
  - ✓ 每棵B树单一写入，防止并发写入导致不完善
- 缓解写放大问题:
  - ✓ B树的内部组成单元是 KV 对，每个 KV 对可能有多条边组成，存在写放大
  - ✓ 写入请求处理流程中只写入 WAL，并修改内存中数据，compaction 时再将数据落盘
  - ✓ 每棵B树有自己的 WAL 日志流



2.2 ByteGraph 架构-存储引擎层 - 缓存实现

- 图原生缓存，理解图的语义
  - ✓ “图原生”是指缓存层组织成图数据结构
  - ✓ 支持一度查询中部分计算下推功能
- 高性能 LRU Cache
  - ✓ 支持缓存逐出，逐出频率，逐出触发阈值可调
  - ✓ Numa aware、cpu cacheline aware，提高性能
- Write-through cache
  - ✓ 支持多种与底层存储同步数据的模式，可以每次写入落盘，也支持定时落盘
  - ✓ 支持定期与底层存储校验数据，防止数据过旧
  - ✓ 支持负缓存等常见优化策略
- 缓存与存储分离
  - ✓ 当数据规模不变，请求流量增大的情况下，缓存与存储分离的模式可以快速扩容缓存来提高服务能力

2.2 ByteGraph 架构-存储引擎层 - 整体结构



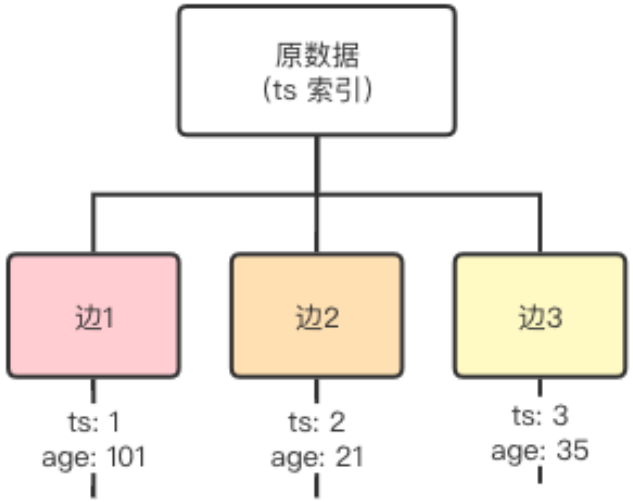
简介

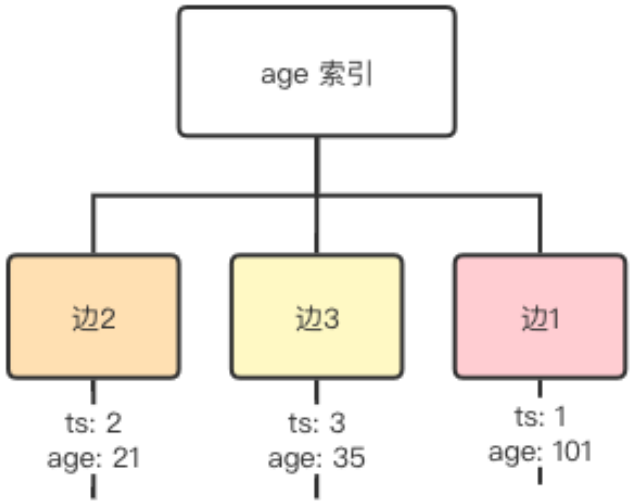
架构

关键问题

3.1 ByteGraph 架构-图索引 - 局部索引

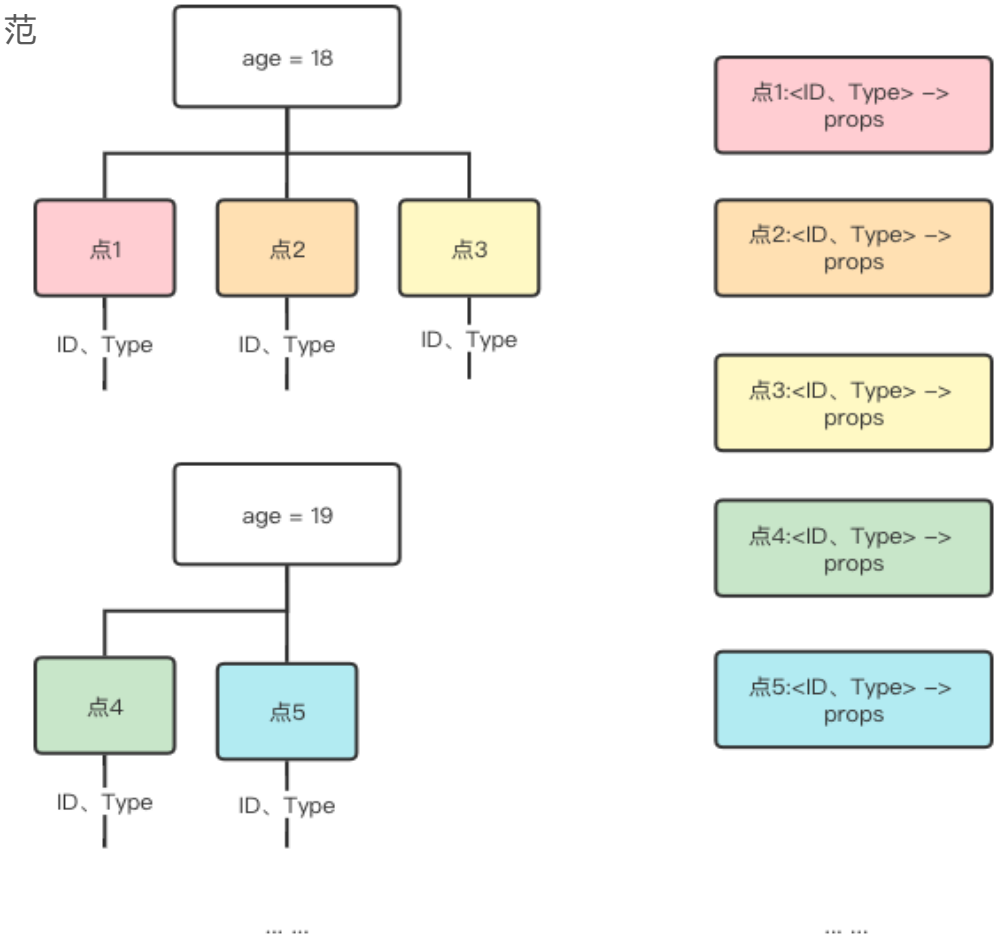
- 概念介绍
  - ✓ 给定一个起点和边类型之后，对边上属性构建的索引
- 使用场景：加速查询
  - ✓ 边属性过滤
  - ✓ 边属性排序
- 存储和构建实现方式
  - ✓ 边上的元素都可以作为索引键，例如终点、边属性
  - ✓ 会额外维护一份索引数据，切与对应的元数据使用同一条日志流来保证一致性
  - ✓ 存储
    - ✓ 采用B+ tree和page的方式存储索引，索引上的值为主键
    - ✓ 索引和原数据在同一个实例
  - ✓ **构建方式**
    - ✓ 同步构建：增删改时，同步修改元数据和索引
    - ✓ 惰性构建：根据查询代价构建（节省存储空间）





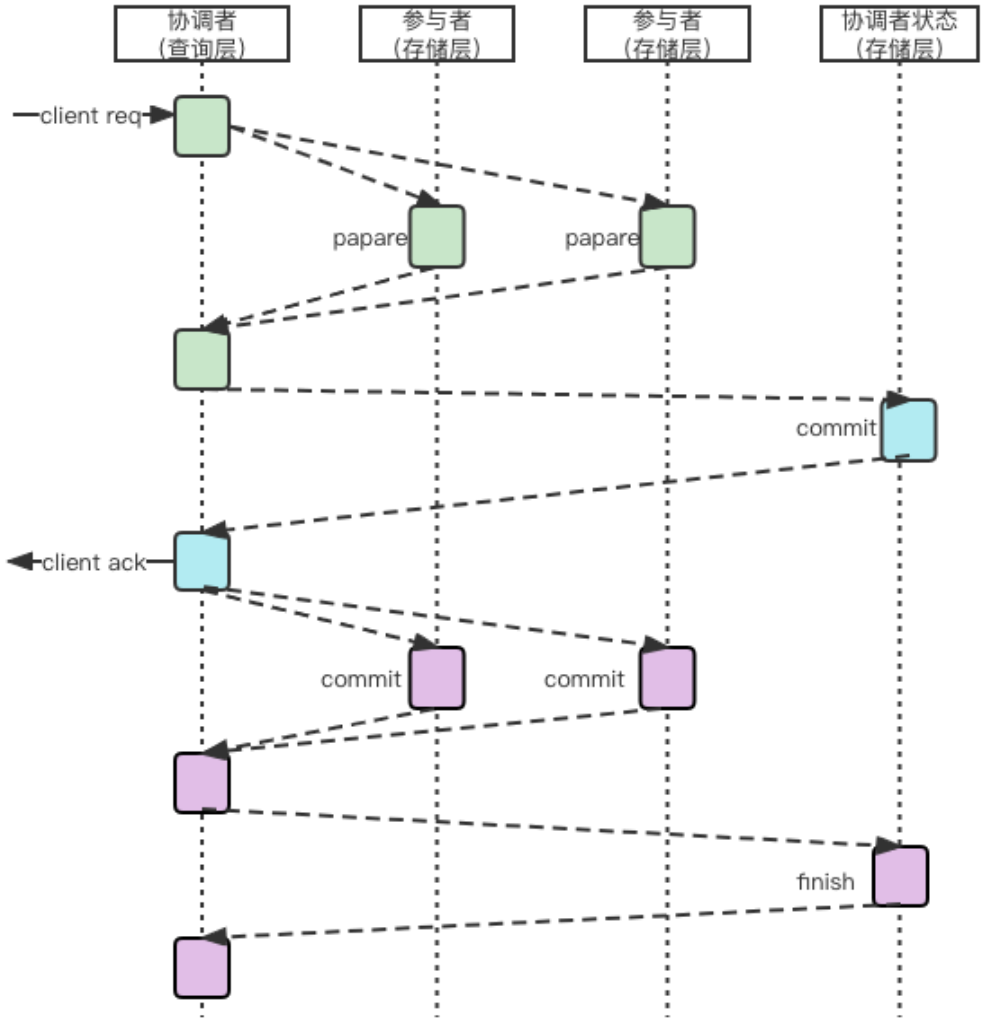
3.2 ByteGraph 架构-图索引 - 全局索引

- 概念介绍
  - ✓ 针对全图的点，对其单个或多个属性建立索引，实现对点的按属性等值查找、范围查找、排序需求（当前仅支持按属性等值查找）
- 使用场景
  - ✓ 查询所有年龄（属性名age，类型int64）为18的点
- 实现方式
  - ✓ 使用分布式事务维护数据和索引一致性
  - ✓ 查询通过索引找到点，再查点属性
- 存储和构建
  - 存储
    - ✓ 采用B+ tree和page的方式存储索引，索引上的值为点
    - ✓ 索引和原数据一般在不同实例
  - 构建方式
    - ✓ 同步构建：增删改时，同步修改元数据和索引
    - ✓ 不支持在存量的数据上构建



3.3 ByteGraph 架构-分布式事务

- 两阶段提交协议
  - ✓ 协调者：查询层
  - ✓ 参与者：存储层
  - ✓ 协调者状态：存储层
- Prepare 阶段
  - ✓ 协调者向所有参与者发起 Prepare 请求
- Commit 阶段
  - ✓ 协调者将事务状态修改为 Commited
  - ✓ 此时，可以向 client 返回该事务结果
- 后台异步阶段
  - ✓ 协调者向所有参与者发送 commit 请求
  - ✓ 协调者清理该事务状态
- 隔离级别和一致性
  - ✓ 读已提交
  - ✓ 最终一致



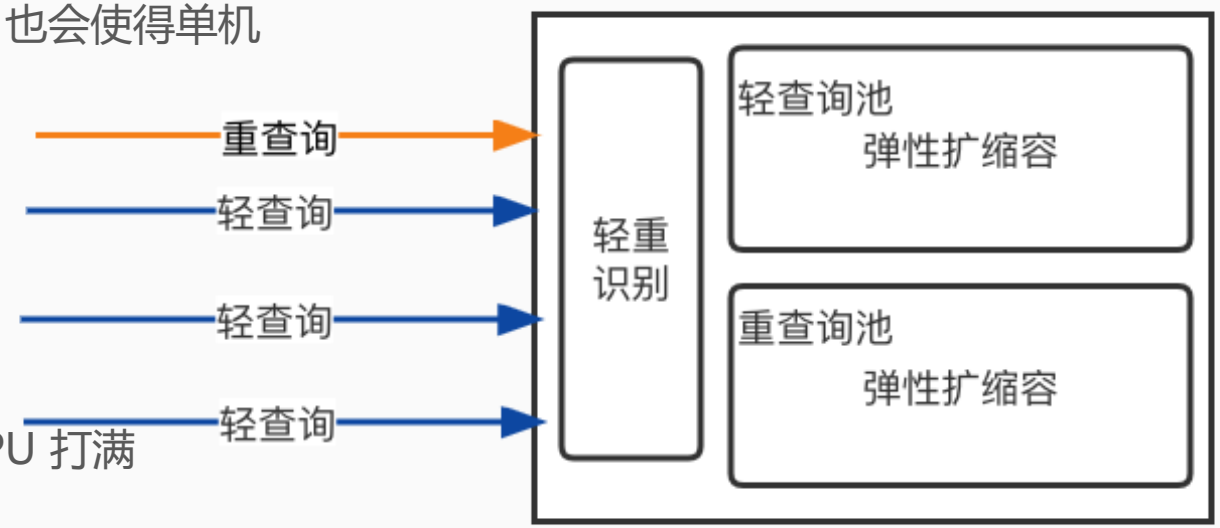
3.4 ByteGraph 架构-重查询-自适应限流

场景举例

- ✓ 超级节点：抖音中的网红大 V 会有千万或者上亿粉丝
- ✓ 查询：大 V点、边属性进行过滤，极低 QPS 也会使得单机 CPU 打满
- ✓ 后果：影响单机可用性

应对策略

- ✓ 识别查询代价
- ✓ 重查询线程池：限制服务能力，避免单机 CPU 打满
- ✓ 动态调整进入重查询线程池的阈值



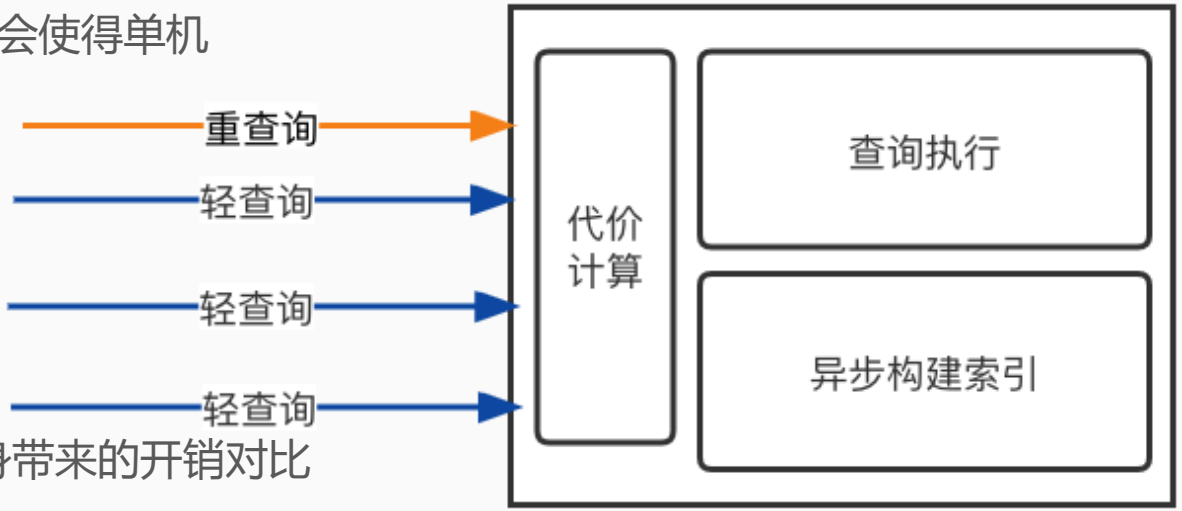
后续规划

- ✓ 增加重查询线程池弹性
- ✓ 根据查询优先级限流

3.4 ByteGraph 架构-重查询-自适应局部索引

场景举例

- ✓ 超级节点：抖音中的网红大 V 会有千万或者上亿粉丝
- ✓ 查询：大 V点、边属性进行过滤，极低 QPS 也会使得单机 CPU 打满
- ✓ 后果：影响单机可用性



应对策略

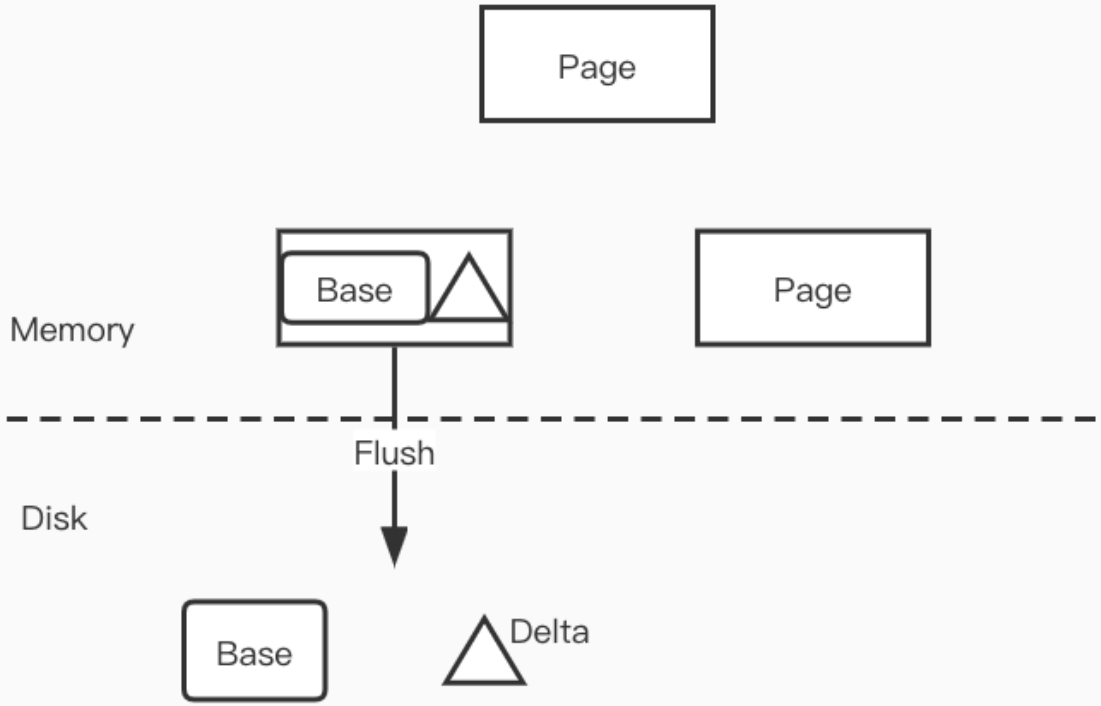
- ✓ 识别查询代价
- ✓ 是否构建索引：构建索引后执行代价和索引本身带来的开销对比
- ✓ 动态构建索引

后续规划

- ✓ 动态删除索引
- ✓ 更好地和限流结合？高优先级构建索引，低优先级限流

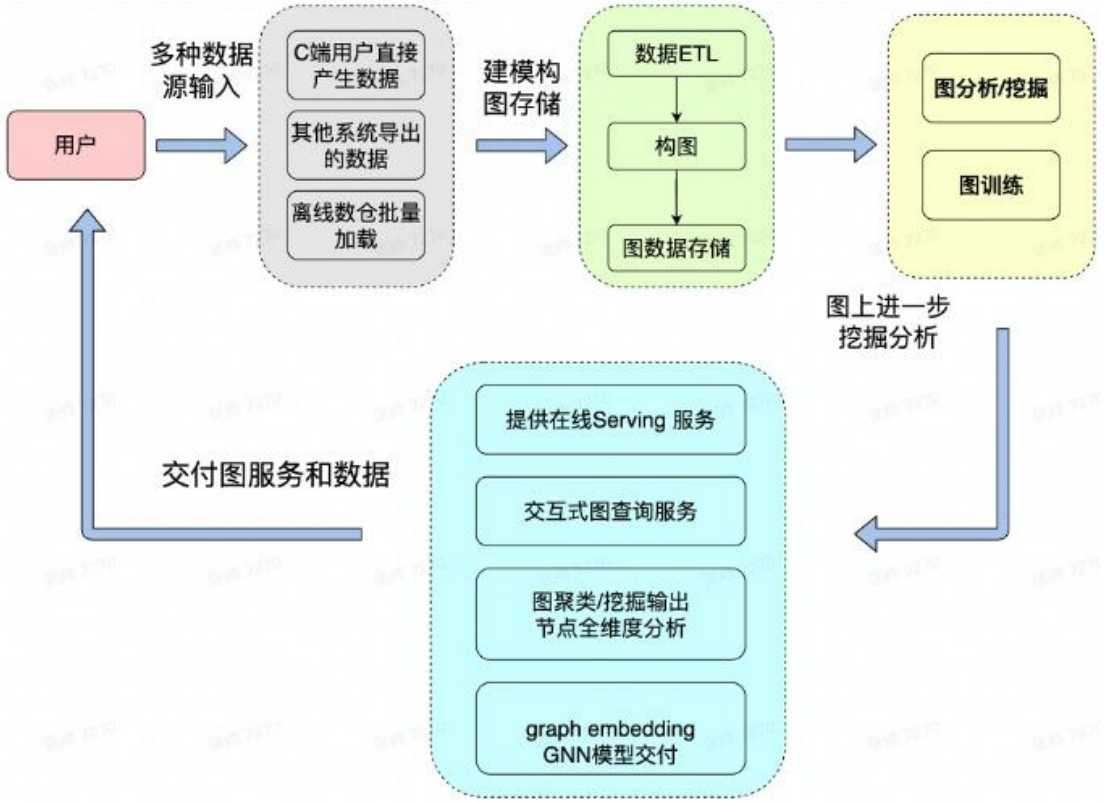
3.5 ByteGraph 架构-写入放大优化

- 场景举例
  - ✓ B+ 树写入会刷新脏 Page
  - ✓ 写入放大计算：脏 Page 大小 / tuple 大小
  - ✓ 结论：相对于 LSM，B+ 写入放大较大
- 应对策略
  - ✓ 策略：类似 BW-Tree
  - ✓ Page 写入：根据脏数据数量选择写入 Delta Page 和 Base Page
  - ✓ Page 读取：合并 Base Page 和 Delta Page



3.6 ByteGraph 架构-在离线生态

- 存量数据导入
  - ✓ 数据源：mysql /hive/redis/kv
  - ✓ 导入工具：
    - ✓ 数据量小：调用ByteGraph的写入rpc api；速度百万qps每秒
    - ✓ 数据量大：使用MapReduce计算存储格式，直接导入KV存储；数据500亿条边每小时
  - ✓ 在线数据实时写入
    - ✓ 直接调用ByteGraph写入rpc
    - ✓ ByteETL工具: kafka等多种消息队列消费时候调用写入rpc写入
- 在线数据天级快照
  - ✓ ByteGraph->Hive，集成在数据平台，页面一键完成配置，无需运维
- 离线数据分析
  - ✓ 基于hive，做图计算离线分析
  - ✓ 离线计算结果可以再导入ByteGraph在线访问



总结

- ByteGraph 介绍
  - ✓ ByteGraph 可以做什么
  - ✓ Gremlin 查询接口和举例
  - ✓ ByteGraph 业务介绍
- ByteGraph 架构
  - ✓ ByteGraph 查询引擎
  - ✓ ByteGraph 存储引擎
- 关键问题
  - ✓ 图局部索引和全局索引
  - ✓ 分布式事务
  - ✓ 重查询优化
  - ✓ 写放大优化
  - ✓ 在离线生态

THANKS!

今天的分享就到这里...

Q&A

我的邮箱: chenchao.chen@bytedance.com

欢迎后续交流讨论


openGauss
OceanBase
ArkDB
RASESQL
HotDB
StellarDB
QianBase xTP
云树Shard
GoldenDB
DolphinDB
MatrixDB
DynamoDB
SinoDB
FastData
Galaxybase
KunDB
SQL Server
vertica
D B 2
GBase
Oracle
达梦数据库
Spacture
开务数据库
UbiSQL
神舟通用
TDSQL
TiDB
Tapdata
StarRocks
KingbaseES
2010
2014
2018

THANKS