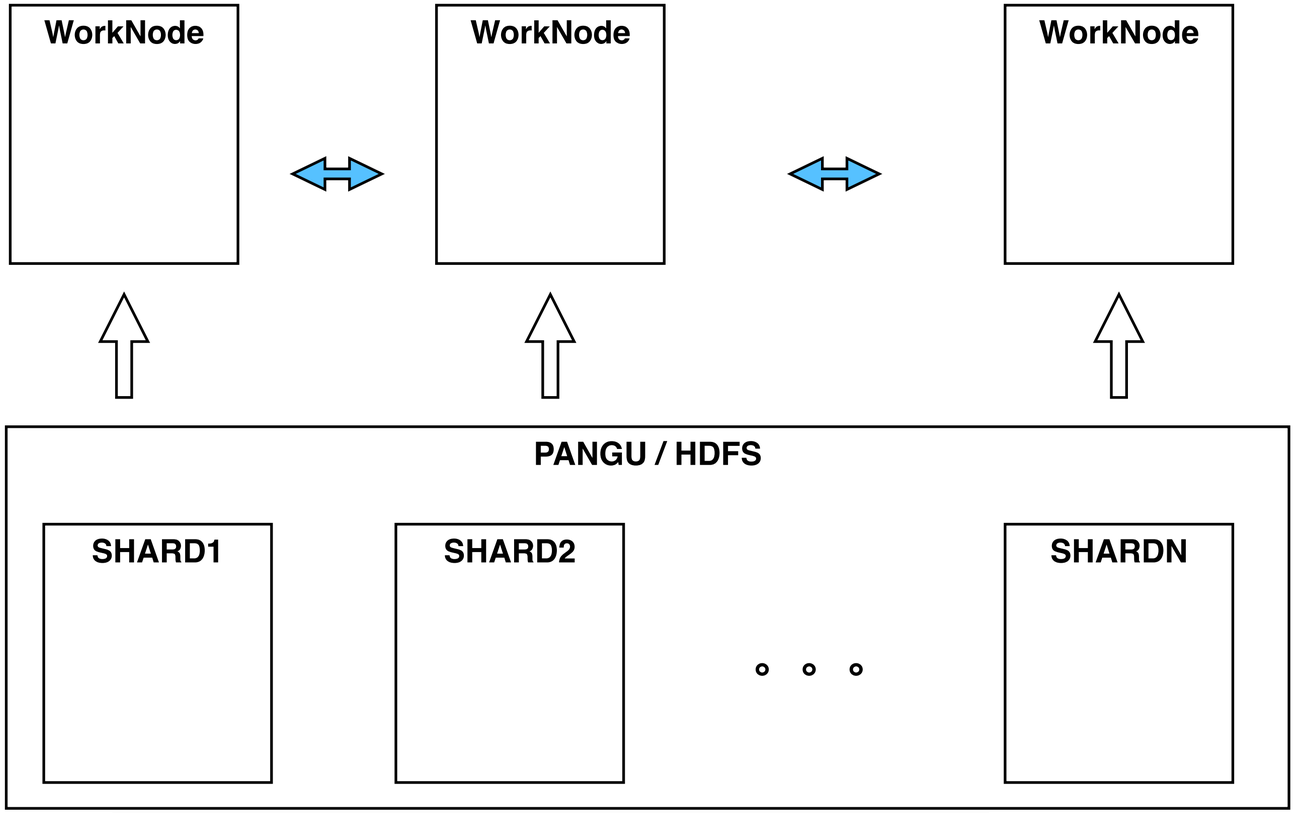
Hologres论文导读

《Alibaba Hologres- A Cloud-Native Service for Hybrid Serving/Analytical Processing》

PolarDB-O 兼容性组- 北侠

Hologres - 特点

- Holo存储层特点：
  - 底层存储共享；
  - 数据分区存储（必选）；
  - 行存/列存
- Holo计算层
  - 高并发
  - 执行引擎-MPP（必选）；
- 和Polar的对比
  - PolarDB + 分区表（可选）
  - PolarDB-O
    - 单机执行
    - 单机Parallel Query
    - MPP on shard-storage

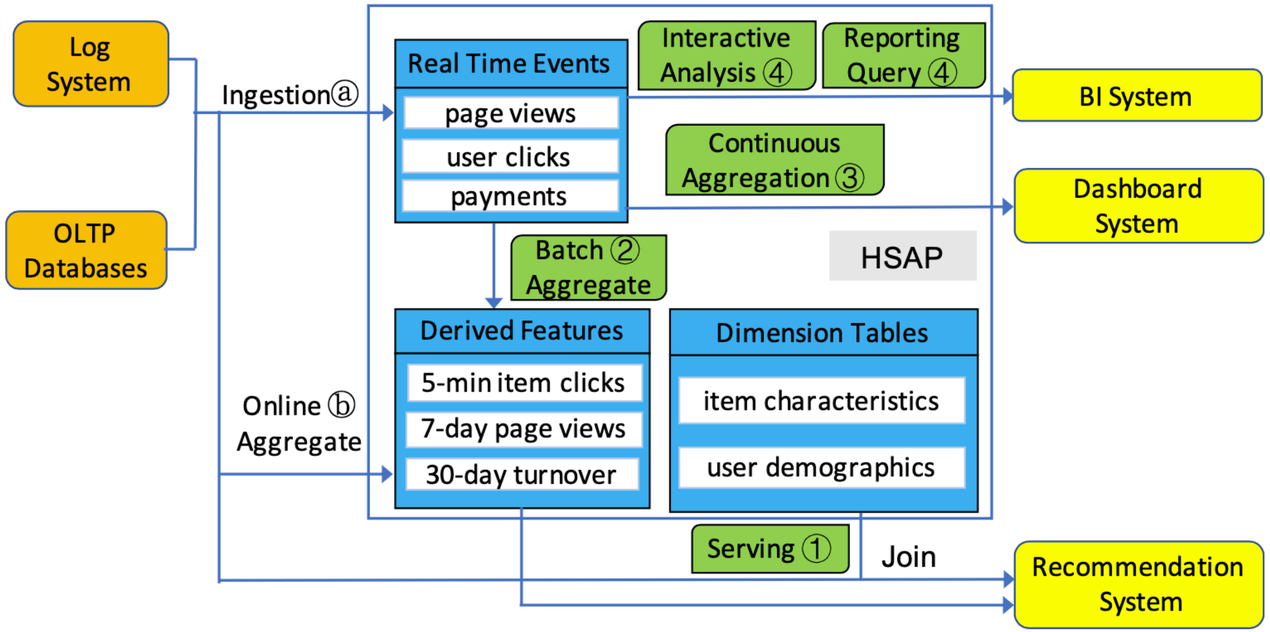



```
BEGIN;
    CREATE TABLE LINEITEM
    (
        L_ORDERKEY      BIGINT      NOT NULL,
        L_LINENUMBER    INT         NOT NULL,
        L_QUANTITY      DECIMAL(15,2) NOT NULL,
        L_SHIPDATE      TIMESTAMPTZ NOT NULL,
        L_SHIPINSTRUCT  TEXT        NOT NULL,
        PRIMARY KEY (L_ORDERKEY,L_LINENUMBER)
    );
    CALL SET_TABLE_PROPERTY('LINEITEM', 'orientation', 'column');
    CALL SET_TABLE_PROPERTY('LINEITEM', 'storage_format', 'orc');
    CALL SET_TABLE_PROPERTY('LINEITEM', 'clustering_key', 'L_SHIPDATE,L_ORDERKEY');
    CALL SET_TABLE_PROPERTY('LINEITEM', 'segment_key', 'L_SHIPDATE');
    CALL SET_TABLE_PROPERTY('LINEITEM', 'shard_count', '24');
    CALL SET_TABLE_PROPERTY('LINEITEM', 'distribution_key', 'L_ORDERKEY');
    CALL SET_TABLE_PROPERTY('LINEITEM', 'bitmap_columns', 'L_ORDERKEY, L_LINENUMBER, L_SHIPINSTRUCT');
    CALL SET_TABLE_PROPERTY('LINEITEM', 'dictionary_encoding_columns', 'L_SHIPINSTRUCT');
    CALL SET_TABLE_PROPERTY('LINEITEM', 'time_to_live_in_seconds', '31536000');
COMMIT;

BEGIN;
    CREATE TABLE ORDERS
    (
        O_ORDERKEY      BIGINT  NOT NULL PRIMARY KEY,
        O_CUSTKEY       INT         NOT NULL,
        O_ORDERSTATUS   TEXT        NOT NULL,
        O_TOTALPRICE    DECIMAL(15,2) NOT NULL,
        O_ORDERDATE     TIMESTAMPTZ NOT NULL
    );
    CALL SET_TABLE_PROPERTY('ORDERS', 'segment_key', 'O_ORDERDATE');
    CALL SET_TABLE_PROPERTY('ORDERS', 'distribution_key', 'O_ORDERKEY');
    CALL SET_TABLE_PROPERTY('ORDERS', 'colocate_with', 'LINEITEM');
    CALL SET_TABLE_PROPERTY('ORDERS', 'bitmap_columns', 'O_ORDERKEY,O_CUSTKEY,O_ORDERSTATUS');
    CALL SET_TABLE_PROPERTY('ORDERS', 'dictionary_encoding_columns', 'O_ORDERSTATUS');
    CALL SET_TABLE_PROPERTY('ORDERS', 'time_to_live_in_seconds', '31536000');
```

Hologres - 背景

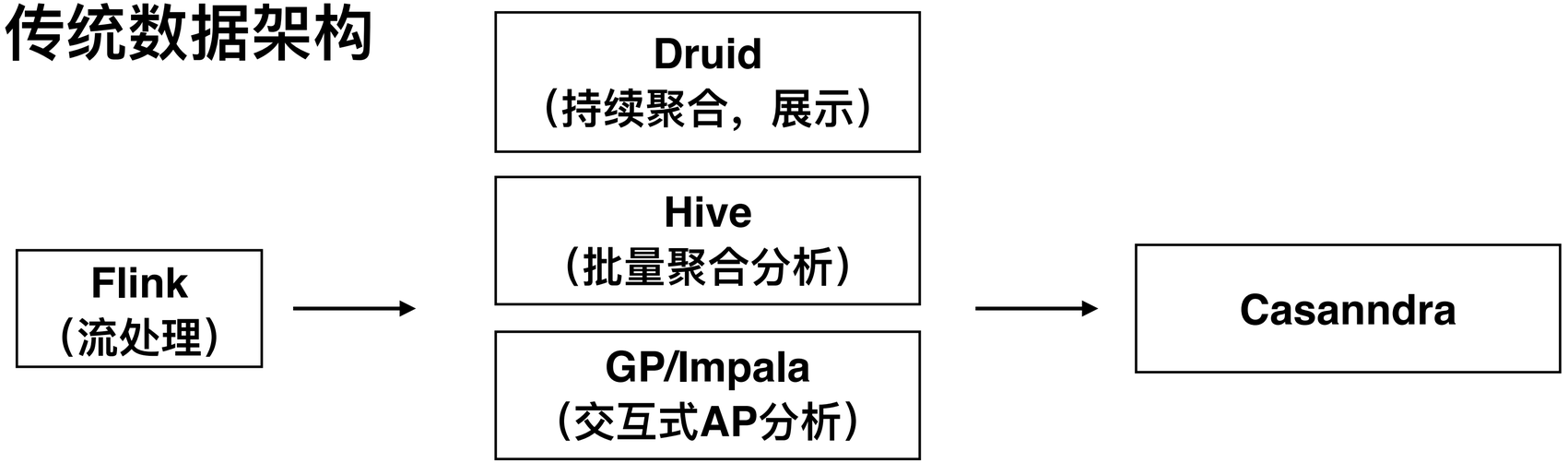
- morden bussiness（多维度数据，实时决策）（有好货）
  - 同时Analytical Processing and Serving
  - 同时Online and Offline Analysis（秒级别可见）
- 离线在先一体化
- 推荐系统举例
  - 实时特征：
    - 数据来自多个系统的日志（曝光，点击，支付）
    - 进入数据栈a，供后续BI和Dashbord使用；
    - 和其他维度信息（用户特征，商品特征）join，输出给推荐系统；
  - 聚合特征：
    - 实时数据进行批量聚合（5分钟点击量，7天曝光数量），进入数据栈b，做为推荐系统的维度数据；



- 每秒5.96亿实时写入；
- 单表2.5PB
- 99.99% 80ms

HSAP - Hybrid Serving/Analytical Processing

- 传统数据架构



- Hologres能力（all in one）

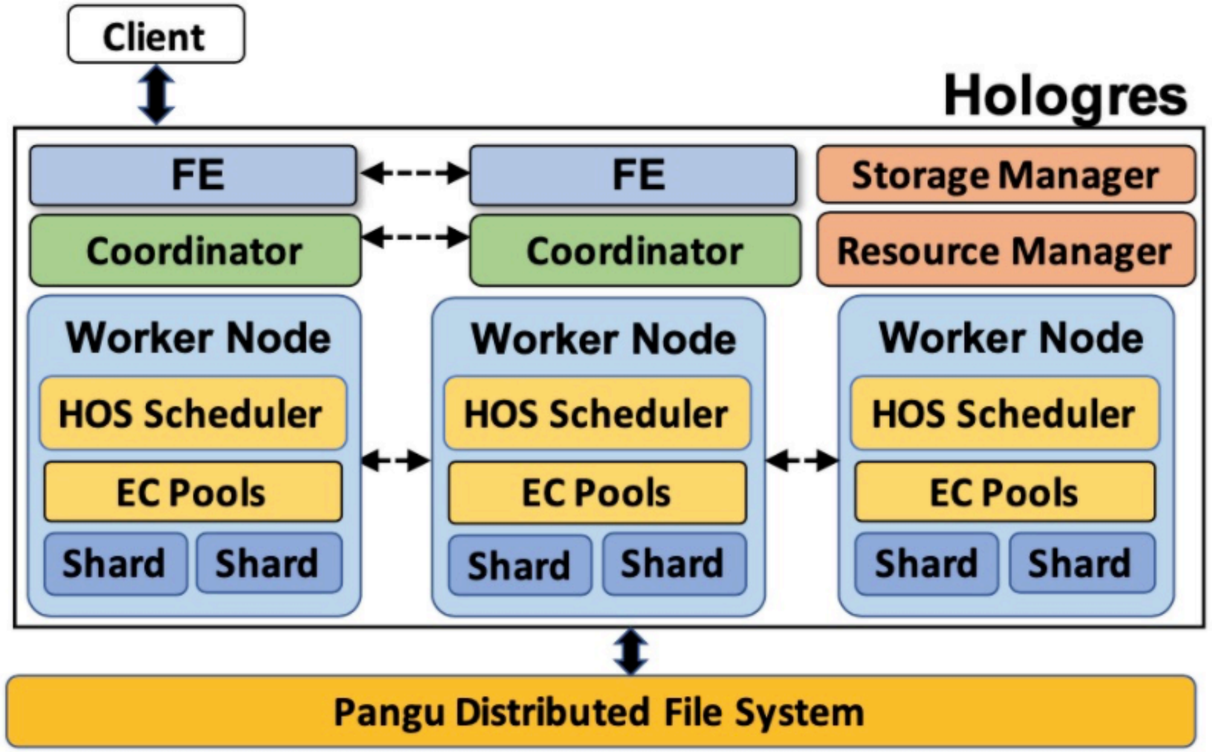
高并发混合查询 （比AP高5个量级）	高并发实时写入秒级可见 （HTAP）	高弹性和扩展性 （HTAP）

Hologres - 设计要点

- 存储引擎
  - 存储计算分离：盘古、HDFS；
  - tablet-base: 表和索引统一使用tablet表示；
    - 逻辑上table group：有业务关联表；
    - 物理组织：分多个shard存储；
  - LSM+多版本；
- 并行查询
  - HOS 用户态调度框架（类协程）
  - MPP执行过程（异步+火山模型）

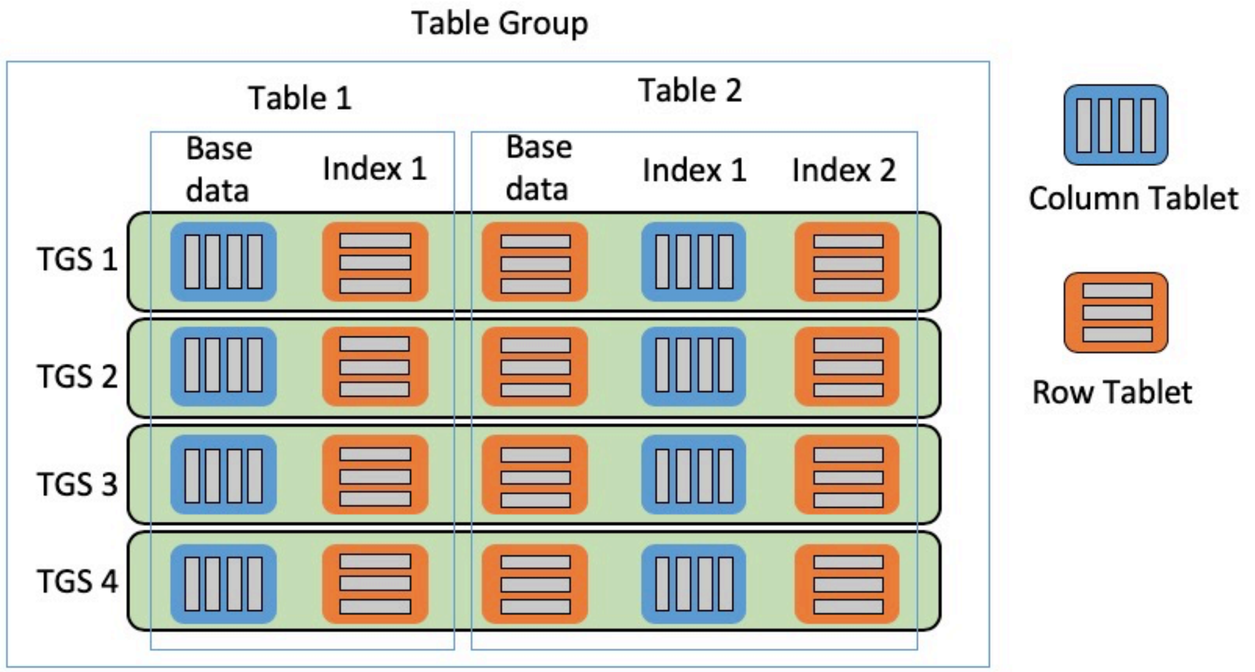
Hologres - 架构

- FE
  - 接收query；
  - 优化器生成计划树；
- Coordinator
  - 分发plan给特定的Coodinator；
- Work Node
  - 计算节点；
  - 管理一个多个shard；
  - EC：调度单元
- Resource manger
  - 管理work node上下线和分配；
- Storage Manager
  - 管理shard文件的路径和元信息（range范围）；
  - Coordinator缓存；
- 外部执行：全兼容PostgreSQL



Hologres - 存储引擎

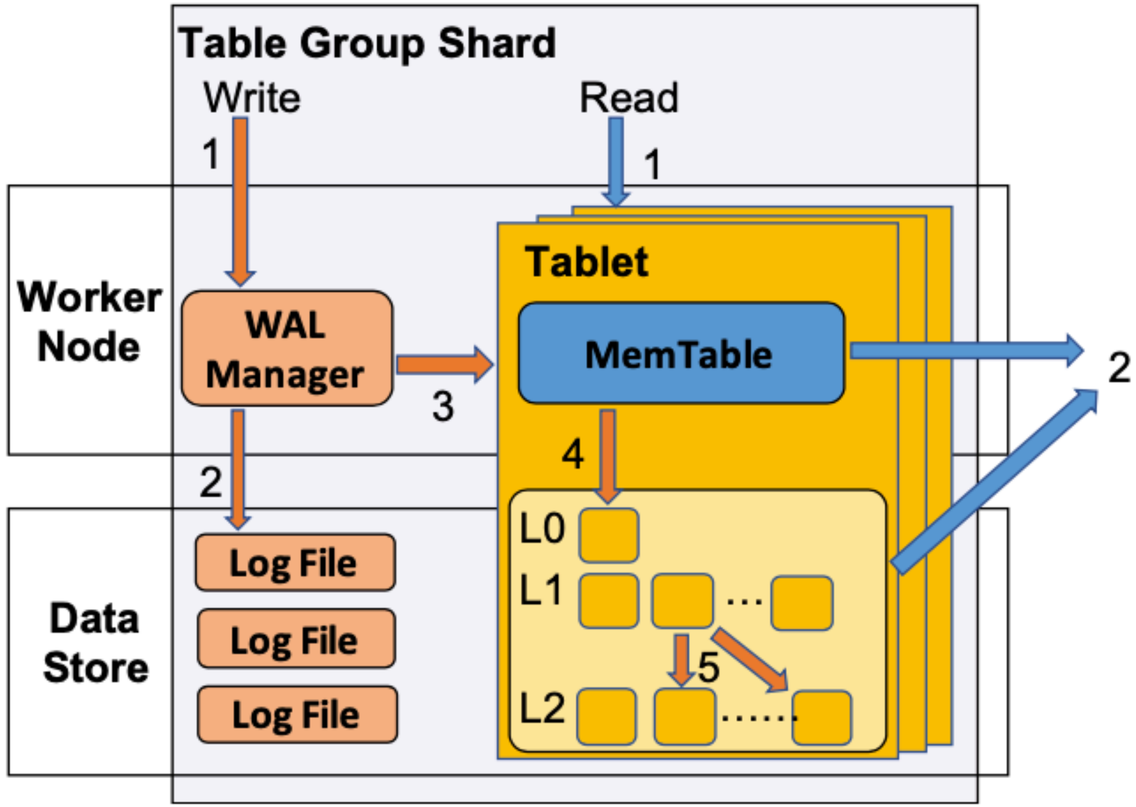
- 数据模型
  - cluster key：用户指定的多个key，LSM排序
  - unique row locator：行定位器
- TG：table group（业务关联的表）；
- TGS：table group shard 物理存储时分成shard；
- Tablet：每个shard管理一个表和对应索引的一个分片，每个分片都叫做tablet，tablet分为行存和列存；
- 唯一性key：
  - 数据tablet：就是row_locator
  - 索引tablet：
    - 如果索引唯一，则为索引key；
    - 否则为，索引key+row_loctor



Hologres - TGS结构

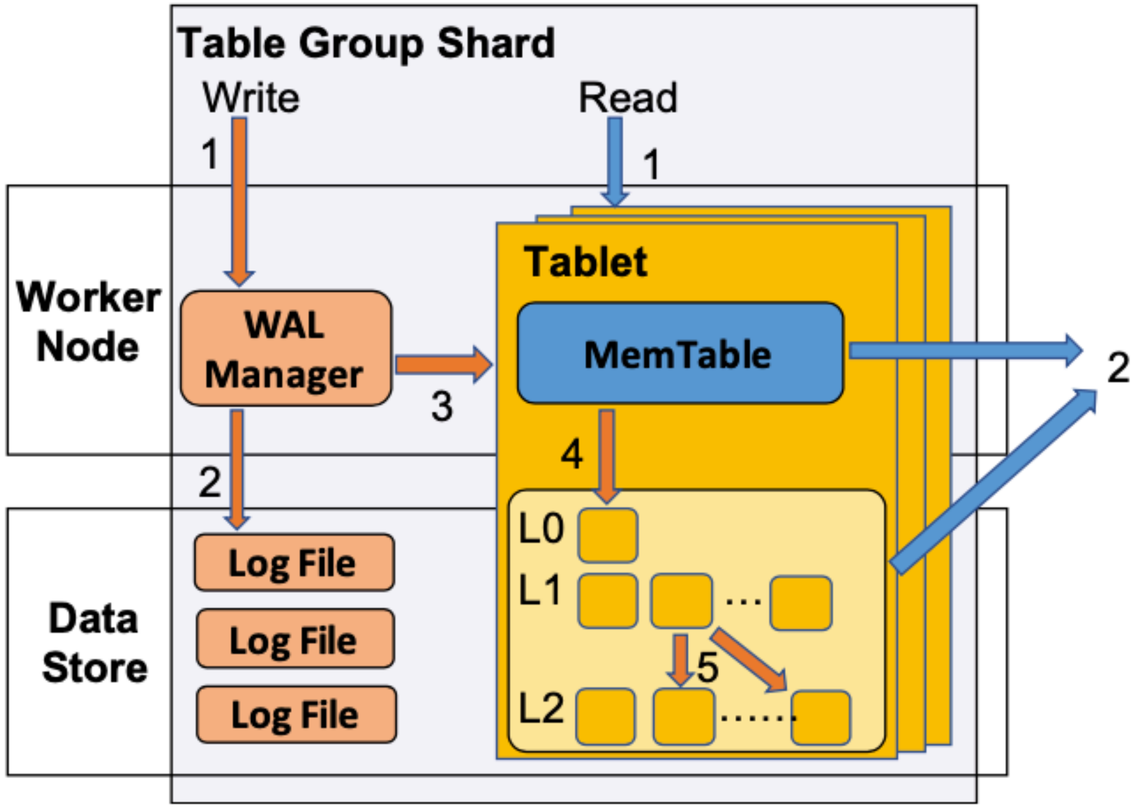
- TGS
  - 多个tablet（LSM） + WAL
  - 每个tablet：memtable + shard文件；
  - 每个tablet都有元数据（表示该tablet的状态）存在RoscksDB上；
- 一致性
  - 每个record都有version
  - atomic write
  - read-your-write

- TGS写流程
  - single shard write：写单个TGS，低延迟；
  - distribute shard write：load到多个TGS；



Hologres - TGS single write流程

- TGS Single Write 流程
  1. 给每个写入分配一个LSN = timestamp + 序列号；
  2. 组织成log entry写入存储，log entry写入wal后，该写入就自动commit了；
  3. 应用到内存中所有的tablet中，然后对读操作可见；
  4. memtable满了后，刷到shard file，并创建空的memtable；
  5. shard file后台异步的compact；
  6. 在compact之后，或者memtable刷到shard之后，更新metafile；



Hologres - TGS Batch write流程

- 2PC协议完成跨TGS的原子写。
  1. FE收到batch写请求；
  2. FE锁住所有相关联的TGS中涉及的tablet；
  3. TGS分配LSN；
  4. 刷memtable；
  5. 每个TGS的writer加载数据，过程和single shard类似（该过程可以把多个memtbale并行化）；
  6. 最终TGS给FE投票；
  7. FE决定commit还是rollback；
  8. TGS收到commit则记录commit日志；否则移除该次写的文件；
  9. 释放锁；

Hologres - TGS Read流程

- 不管是row tablet还是column tablet都支持version读，一致性级别是read-your-write。
  - 例如：client总是能读取都自己已经commit的数据。

- 每个读请求都有一个timestamp，构成LSN<read>，所有小于LSN<read>的都不可见；

- TGS需要维护LSN<ref>：需要维护的最老的数据版本，compact过程中：
  - 所有比LSN<ref>老的可以执行merge；
  - 所有大于LSN<ref>的都保持不动，因此存在client要读取这些version的数据；

Hologres - TGS Management

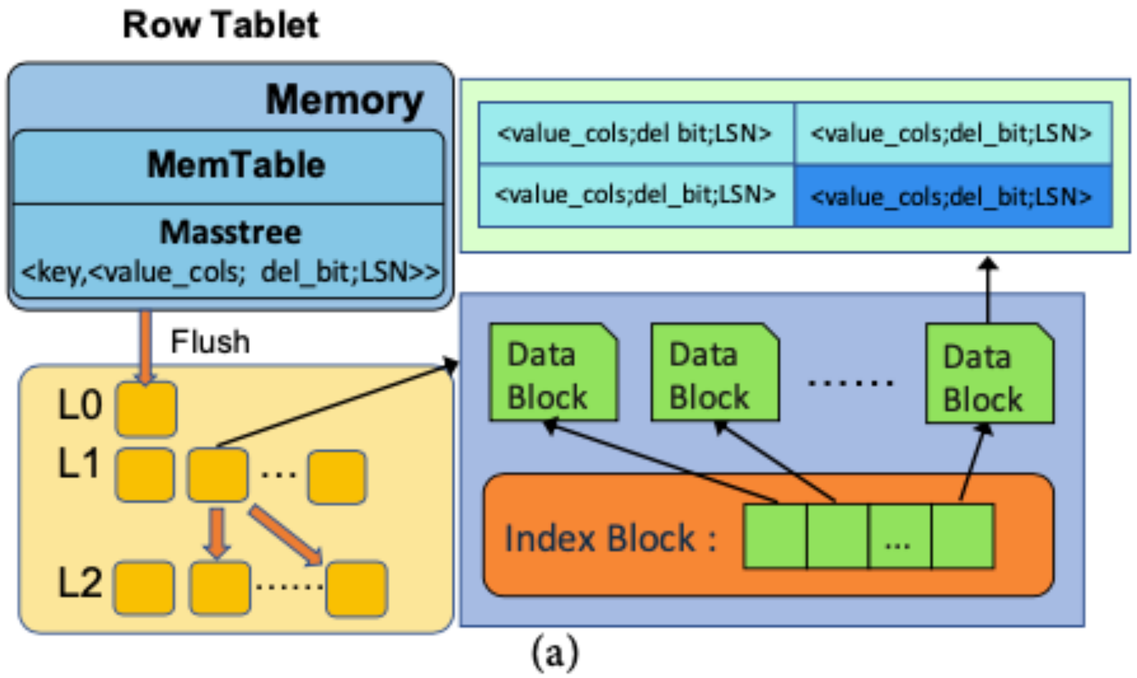
- TGS 热点
  - 迁移TGS到其他work node上
  - read-only副本
    - fully-syncd replica：最新的memtable + metadata；
    - partially-synced replica：最新的metadata，从存储读取数据；

- TGS failvoer
  1. storage manager请求resource manager分配一个可用的slot(记录slot的location)；
  2. 同时向所有的Coordinator广播TGS-fail消息；
  3. 得到slot后，这个work node开始replay日志，一直回放到meta中记录的上次刷memtable的位点；
  4. storage manager向Coordinator广播TGS-recovery消息，同只新TGS的location消息；
  5. 期间Coordinator会把请求排队；

Hologres - ROW Table的结构

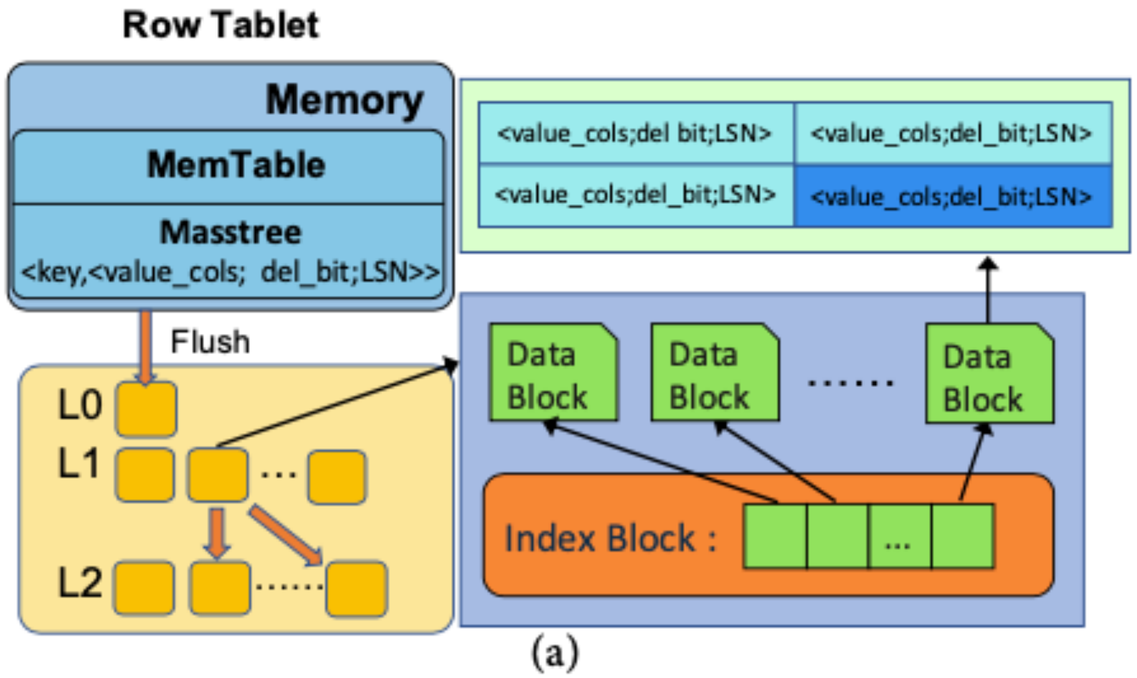
- LSM结构，内存是：MassTree
- shard文件：基于block管理
  - data block：根据key对record进行排序；
  - index block：记录每个data block的起始key以及offset：<key, block_offset>

- 为了支持multi-verisoned data，每个record扩展字段：<value_cols, del_bit，LSN>
  1. value_cols：存储非key的列的值；
  2. del_bit：标记是否被删除；
  3. LSN是写入时分配的LSN；



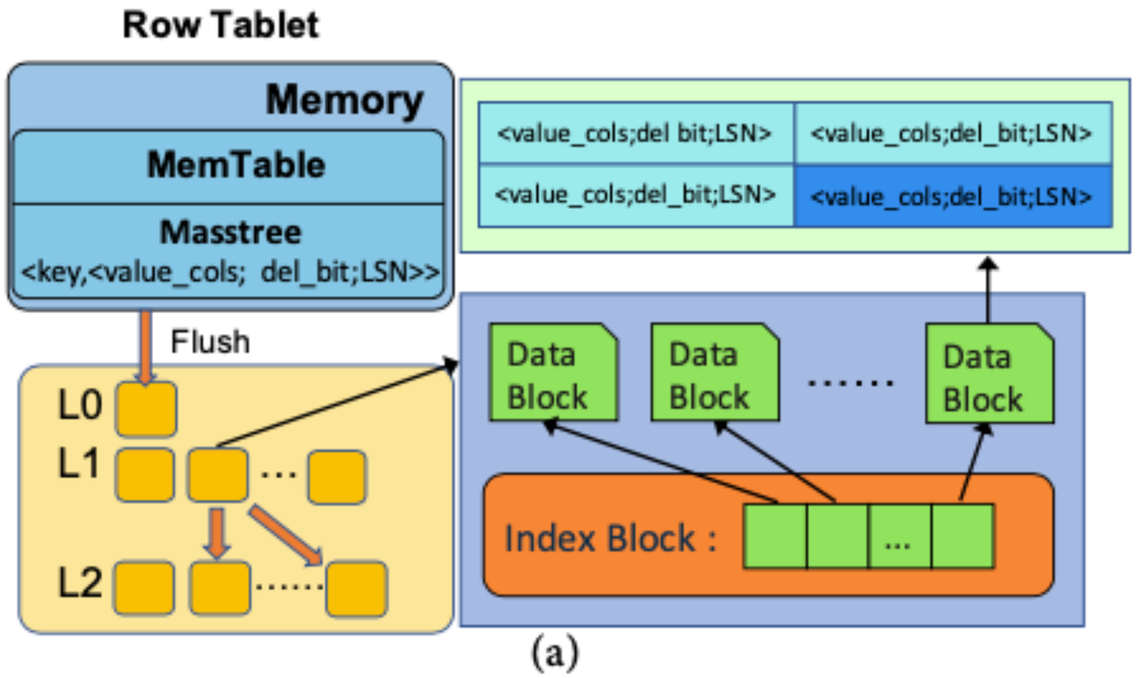
Hologres - ROW Table的Read流程

- 每个读请求构成：key和LSN<read>构成
- 并行查找memtable和shard file，仅查找和key有overlap的shard file；
- candidate record：
  - record包含这个key，同时LSN小于等于LSN<read>；
- 最终Result：所有candidate record被merge成一整行的result；



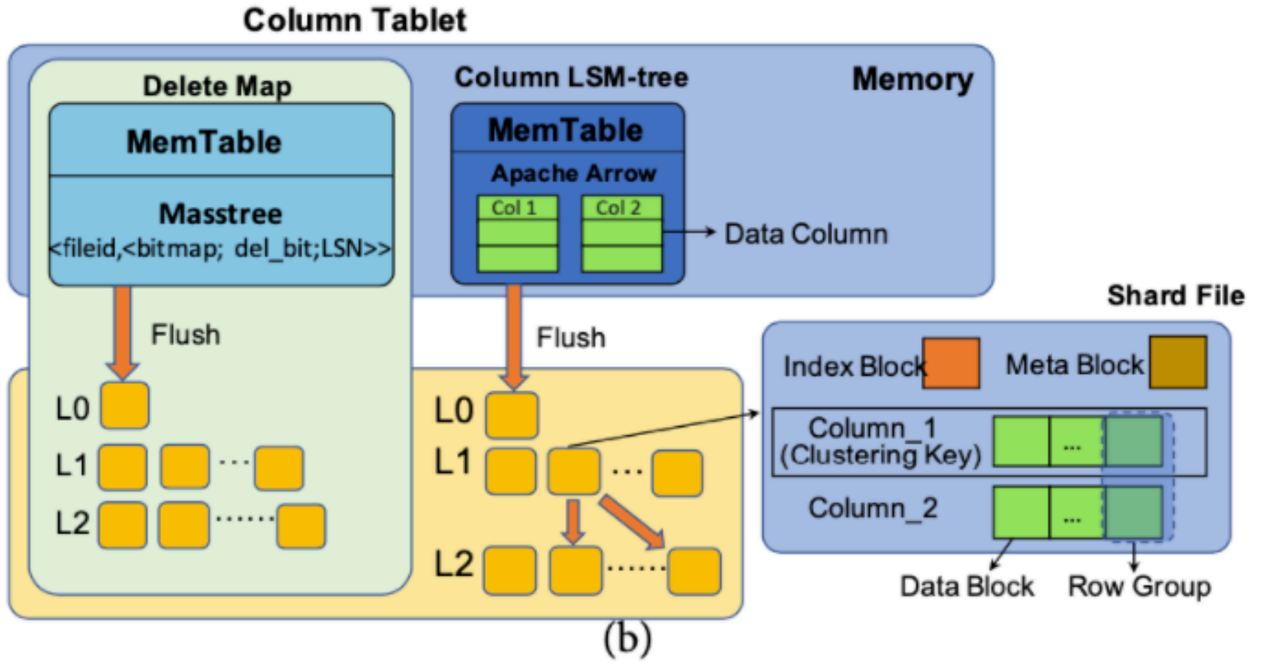
Hologres - ROW Table的Write流程

- insert/update请求构成：
  - key；
  - 更新的列值；
  - LSN<write>；
- delete请求构成：
  - key；
  - LSN<write>；
- 每个写都转换成了kv pair写入memtable，可能触发flush到level0，以及更高层的compaction；



Hologres - Column Table的结构

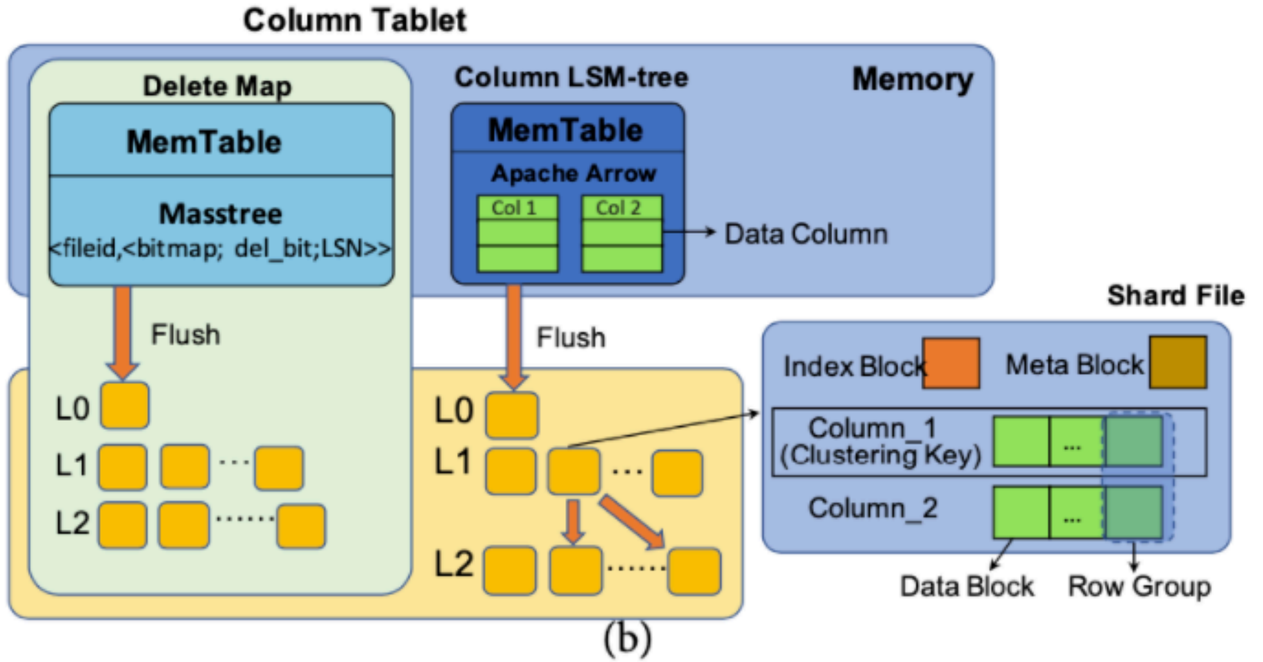
- 2个组件
- 组件1 - Column LSM-Tree：
  1. 格式<values_cols, LSN>；
  2. memtable用Apache Arrow格式，数据按照到来的顺序append；
  3. shard file按照key排序，逻辑上按照行范围分成多个group；
  4. 每个group中，不同的列存储在不同的data block中；
  5. 同一个shard文件中，同一个列连续存储，方便顺序扫描；
  6. meta维护每个列的元信息：
     1. 每个列在shard文件的offset，该列每个block的range，编码scheme；
     2. shard文件的压缩shceme，总row数目，LSN range，key range；
     3. 同样，索引块存储每个row group的第一个ke



- deletion map：
  - 也是一个tablet；
  - 行存格式；
  - <field, <bitmap, del_bit;LSN>>
    - field：是shard文件ID；
    - bitmap：标记哪些行被删除；

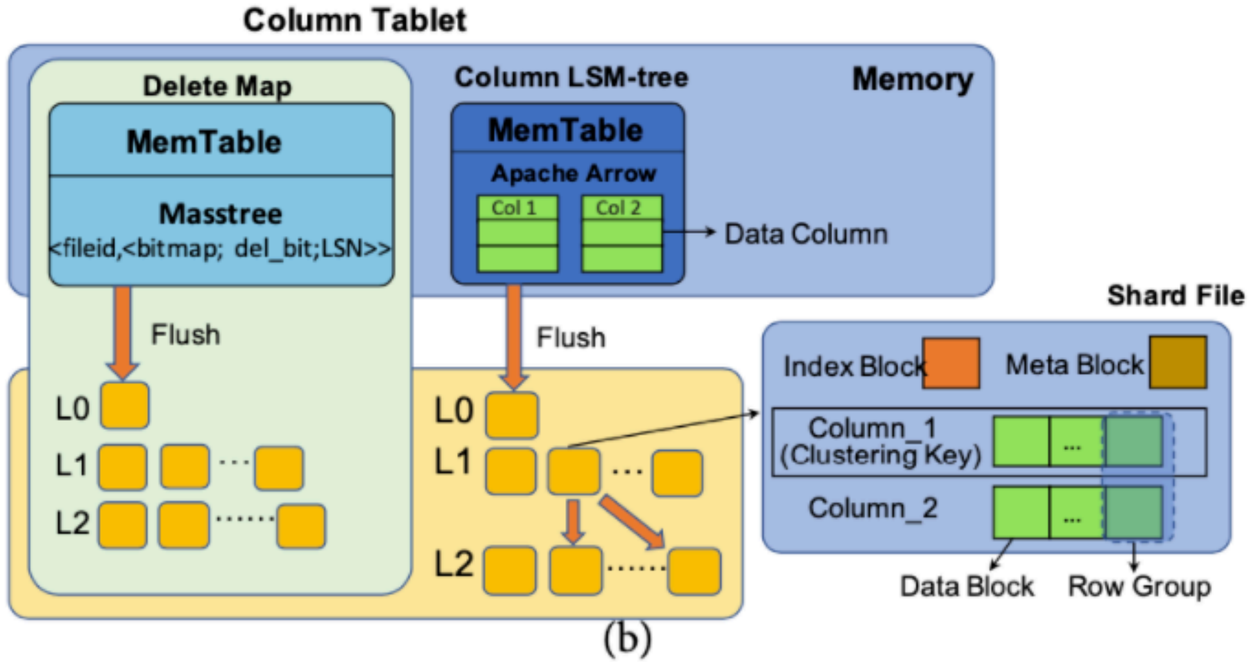
Hologres - Column Table的读流程

- 读Column文件：输入是column和LSN<read>
  1. 如果minimum LSN比LSN<read>大，该文件被跳过；
  2. 如果maxum LSN比LSN<read>小，该文件全部可见；
  3. 否则，部分可见，扫描LSN列存，扫描过程中生成一个LSN的bitmap，标记哪些对于当前LSN<read>是可见的；
- 读Delete文件：输入是上一步记录的shard文件ID
  1. 读取deletemap的行存格式，key为shard文件，LSN为LSN<read>；
  2. deletemap行存文件中的value存放的是：<bitmap, del_bit, LSN>；
  3. 最后merge所有candidate的bitmap，做为最终在LSN<read>下被读取到所有的bitmap；
- 上述两个bitmap做交集
- 由于delete文件是独立的，因此可以并行读取列存文件



Hologres - Column Table的写流程

- 通过meta和key可以快速的定位该key属于单个列存的shard文件ID，以及在该shard文件中的行号；
- 然后在deletemap的行存文件中插入一条记录：key是上面找到的shard文件ID，value是该文件中被删除的行号。

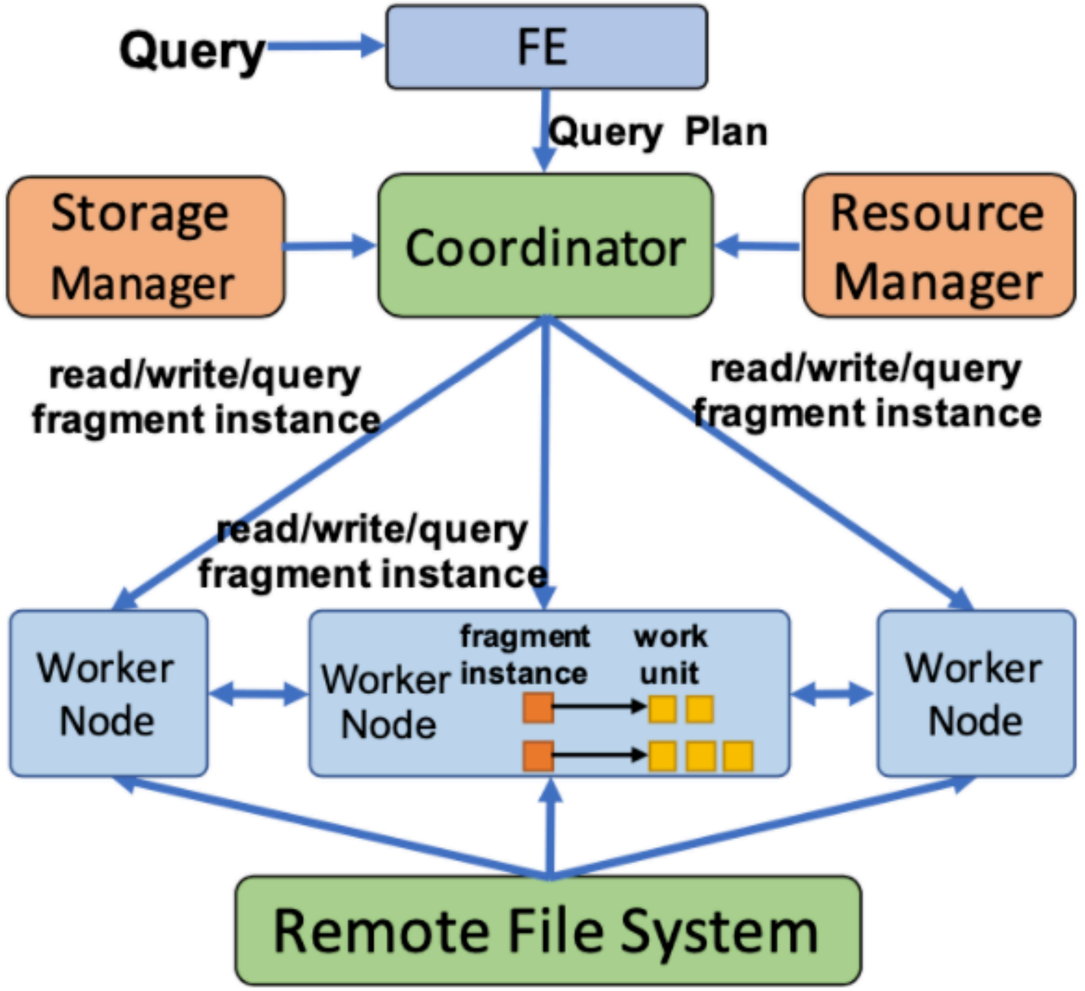


Hologres -分层缓存

- 3层级的cache：
    1. local disk cache缓存：从存储侧读取shard文件在本地的缓存；
    2. block cache：在内存中缓存最近读取的block，行存和列存分开；
    3. row cache：内存中缓存合并之后完整的一行数据；

Hologres - 执行引擎（流程）

1. FE产生分布式计划：
    1. 以shuffle为边界切分成fragments；
    2. 一个fragment给多个TGS来执行；
    3. 每个TGS上执行体称为fragment instance；
2. Coordinator
    1. 查找resource manager得知每个fragment需要到哪些TGS上执行；
    2. 分发plan给work node；
3. Work node把fragment instance映射成多个Work Unit，被HOS调度执行；

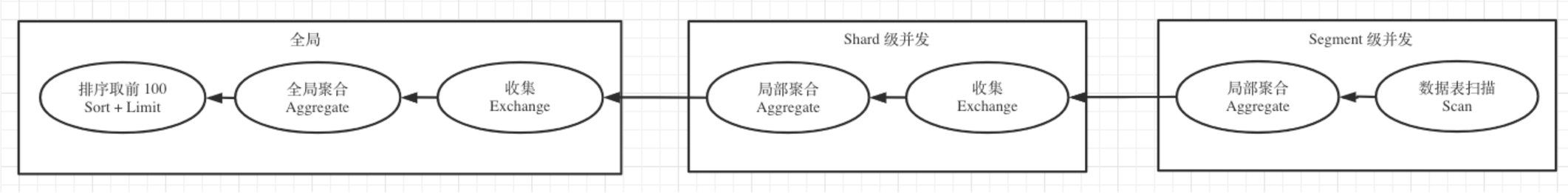


Hologres - 执行引擎（分布式Plan）

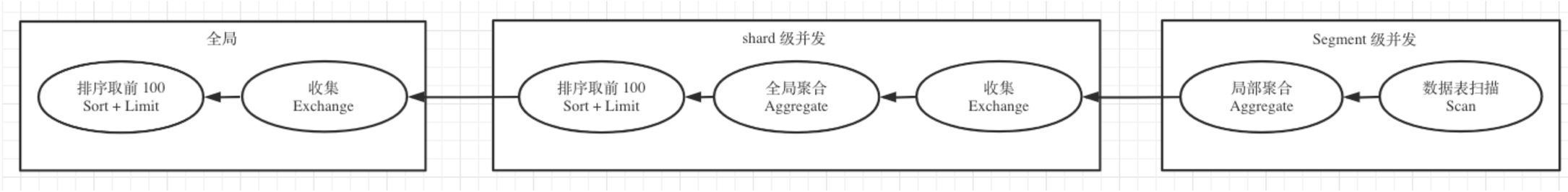
```
select key, COUNT(value) as total
from table1
group by key
order by total desc
limit 100
```

单机计划

Coordinator节点做全局聚合

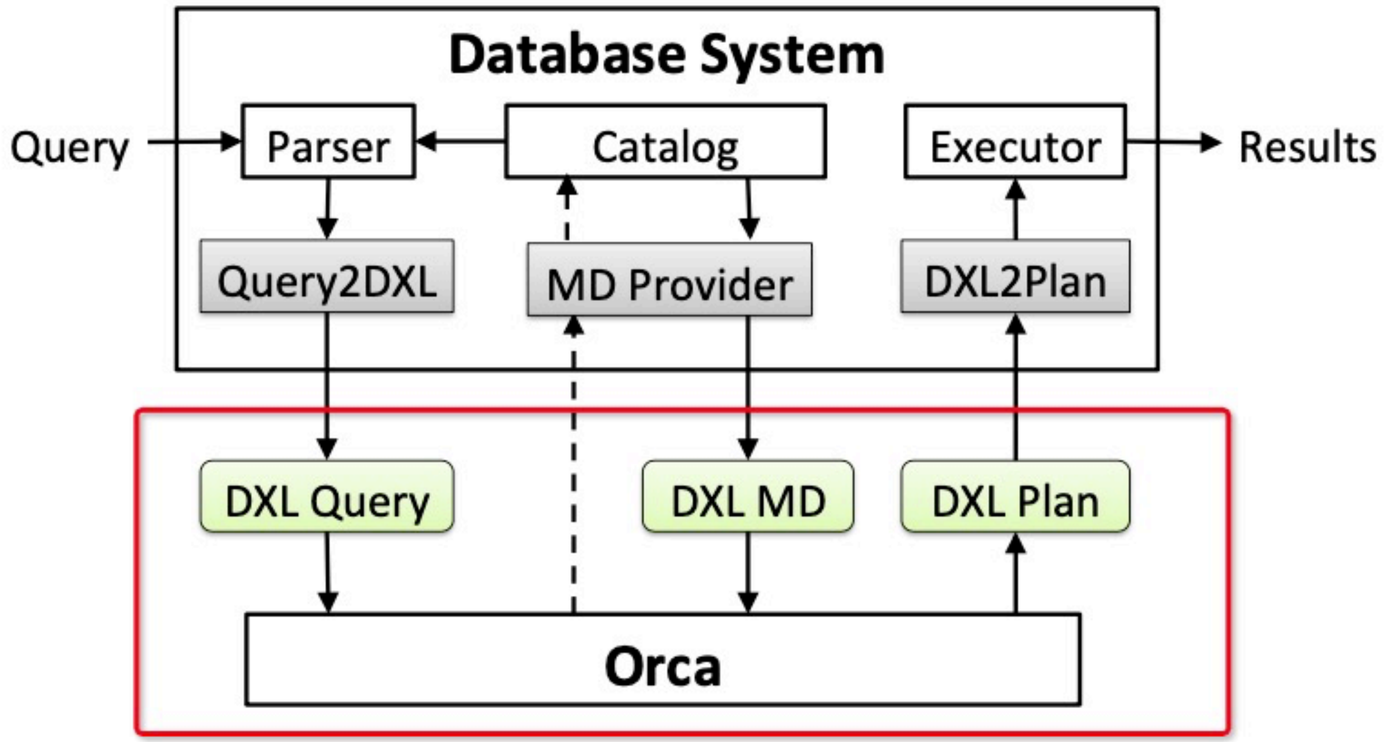


WorkNode节点做全局聚合：key就是distribute key



Hologres - 优化器（背景）

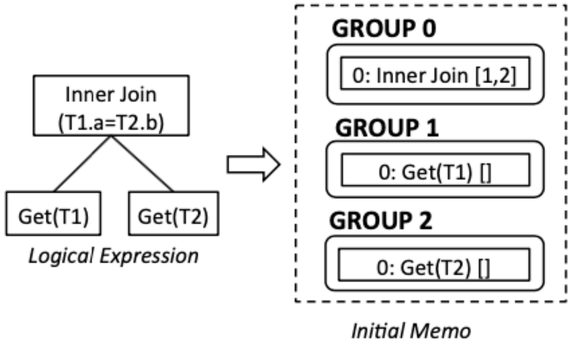
- Bottom-up优化器：
  - 自底向上的动态规划（计算下层最优解，层层向上）；
  - (PG，MYSQL）；
- Top-Down优化器：
  - 自上向下驱动；
  - 剪枝（根据Time，Cost）；
  - 易扩展（搜索规则解耦/模块化设计）；
  - calcite，gporca；
- Hologres：GPORCA：
  - Volcano/Cascades framework；
  - greenplum开源出来的实现；
  - 完善的Cascades Framework实现，完善的Property Enforce支持物理属性的扩展；



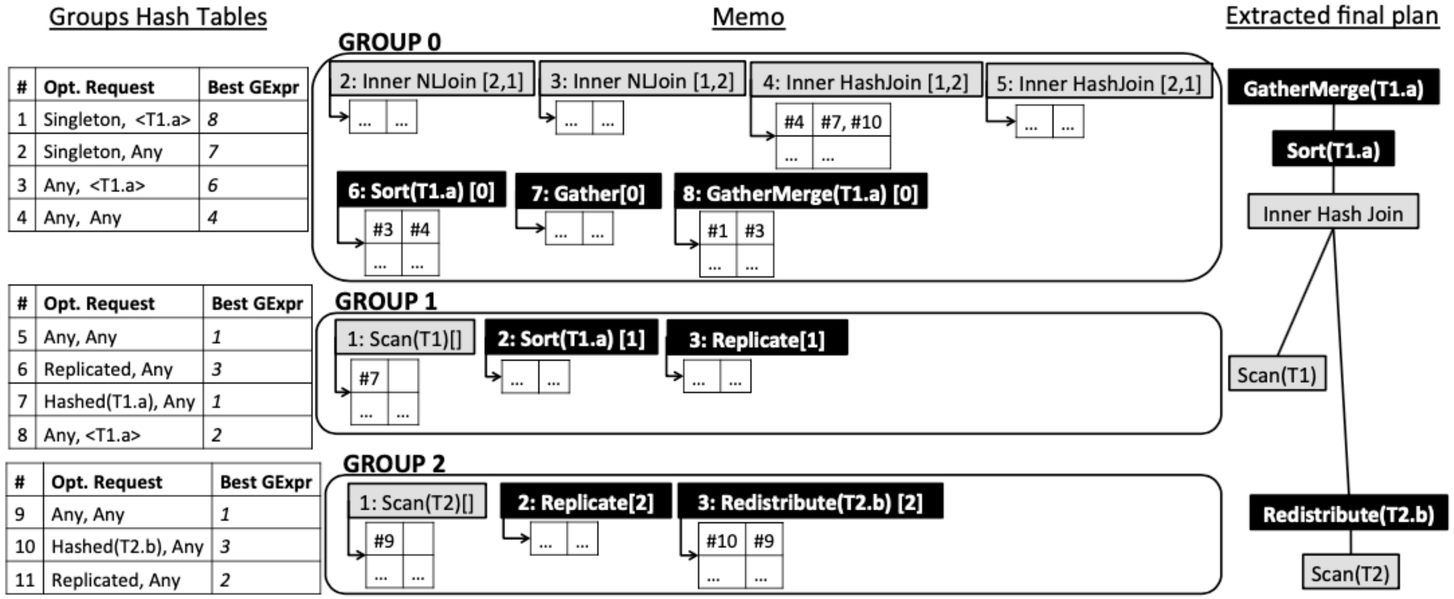
Hologres - 优化器（GPORCA）

GPORCA工作流程：

1. **Exporation：逻辑等价变换，join交换律；**
2. **Statistics Derivation：通过MD provider；**
3. **Implementation：物理转换，nljoin, hashjoin；**
4. **Optimization：Enforce，Cost计算；**



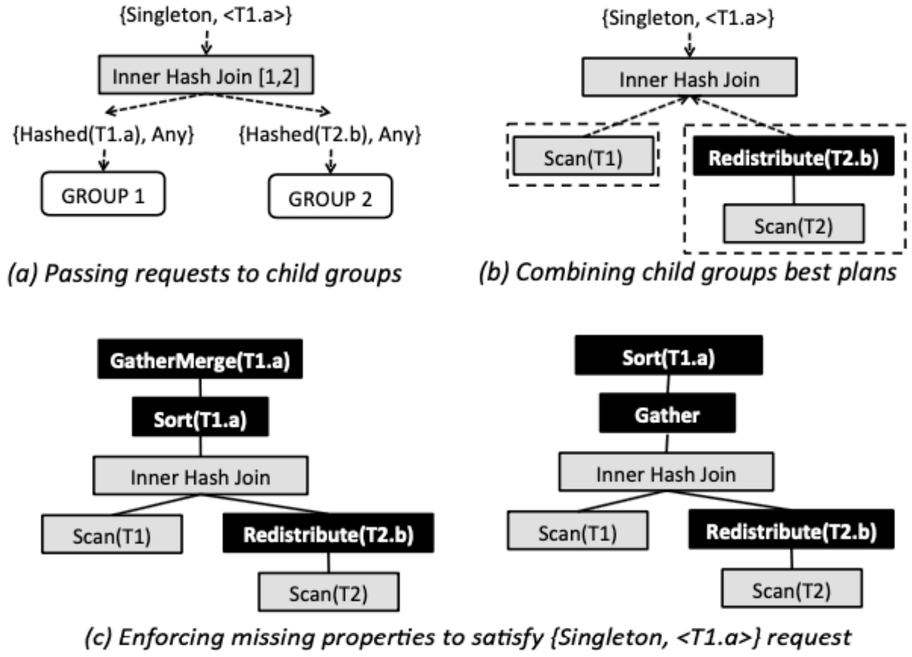
- **Enforcement:**
  - **输入：父节点的Req，自身的physical propter；**
  - **输出：对子节点的Req，**



#	Opt. Request	Best GExpr
1	Singleton, <T1.a>	8
2	Singleton, Any	7
3	Any, <T1.a>	6
4	Any, Any	4

#	Opt. Request	Best GExpr
5	Any, Any	1
6	Replicated, Any	3
7	Hashed(T1.a), Any	1
8	Any, <T1.a>	2

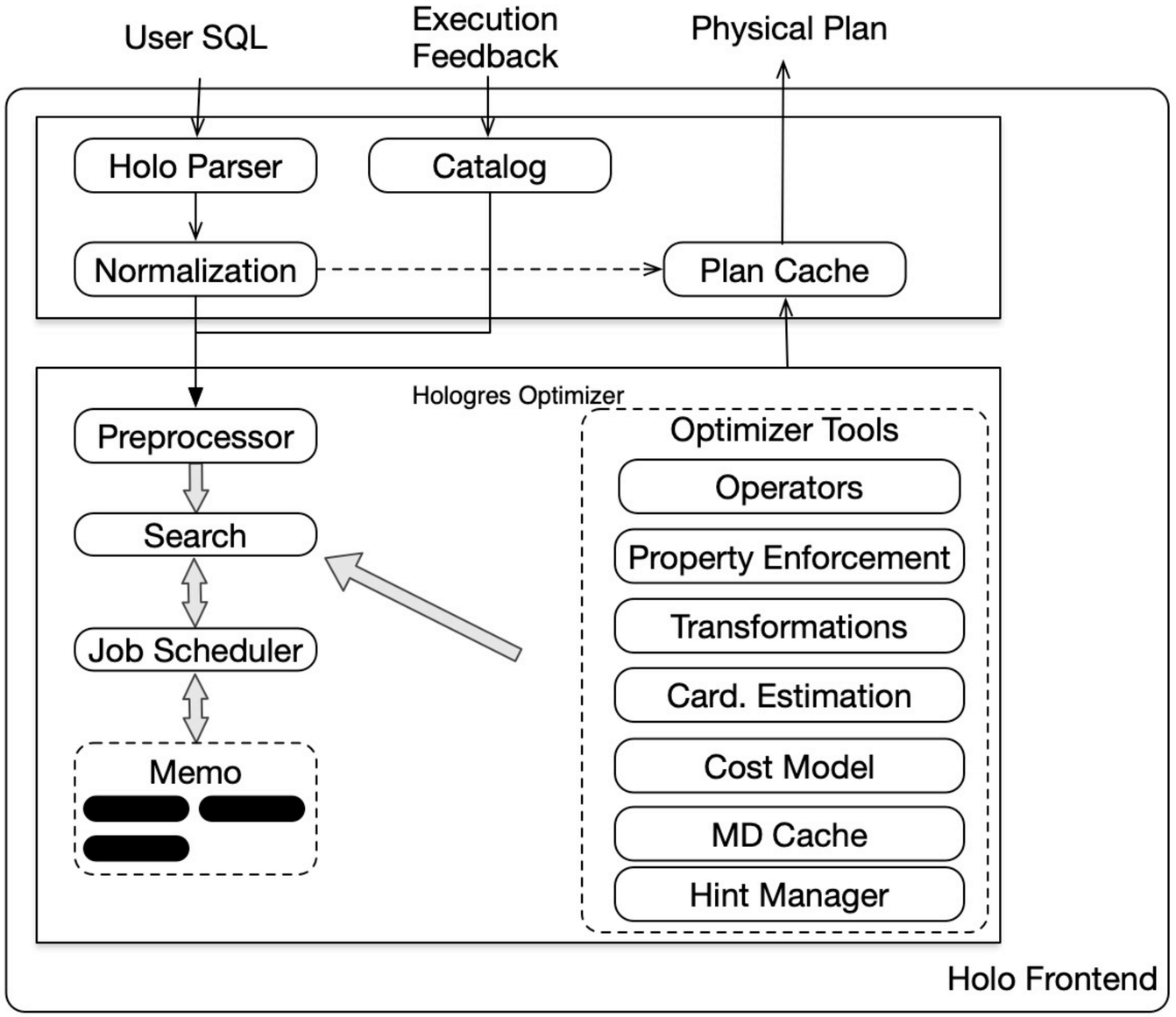
#	Opt. Request	Best GExpr
9	Any, Any	1
10	Hashed(T2.b), Any	3
11	Replicated, Any	2



Hologres - 优化器（Holo流程）

优化器执行流程：

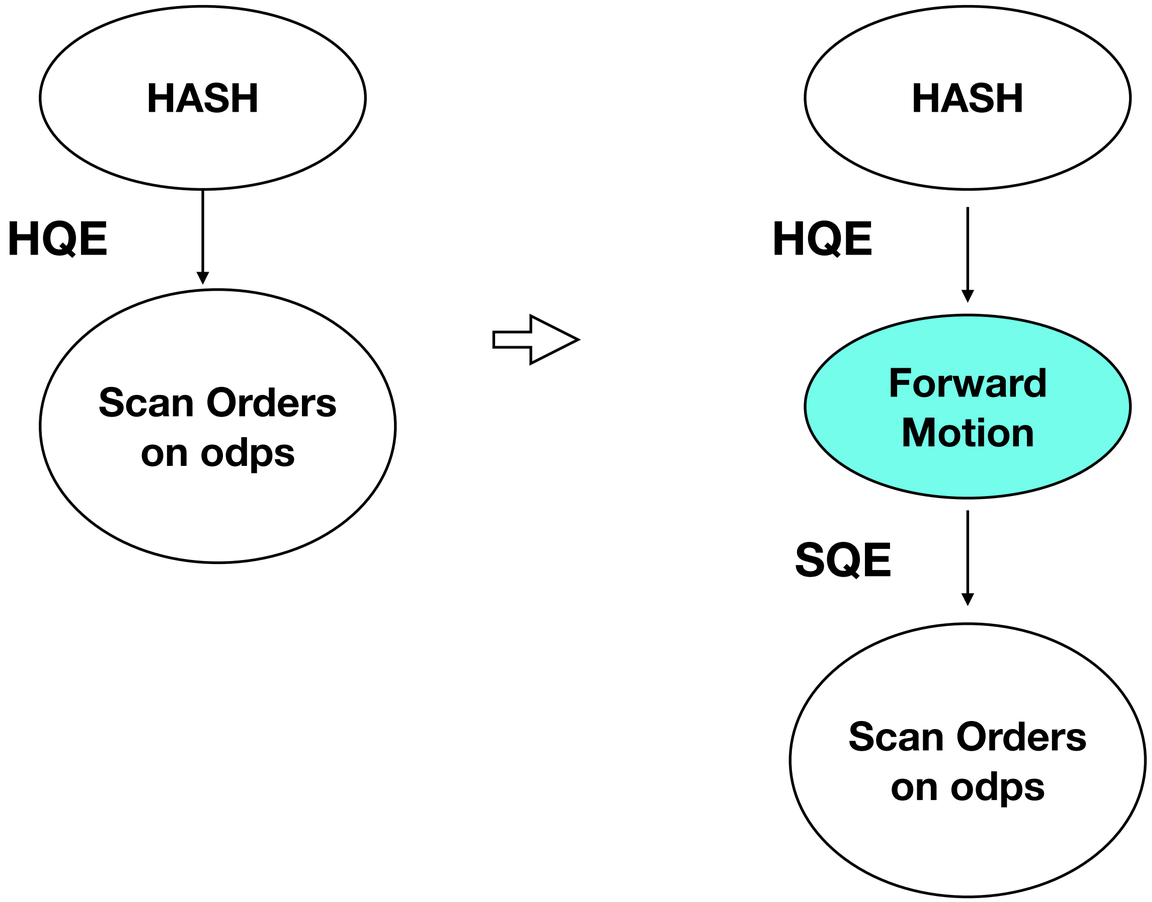
1. 经过解析器logical plan tree；
2. 对logical plan tree做normalization：
    1. 参数模糊化、必要的谓词下推等；
    2. 主要目的是为查找plan cache做一些准备工作；
3. 查找plan cache；
4. 把logical plan tyree转成dxl，传递给orca模块；
5. orca输出dxl，转成plantree；
6. 执行引擎执行计划树；
7. 执行时统计返回放置到Holo的catalog中，用于类似查询的再优化；



Hologres - 优化器（联邦查询）

- 背景：
  - Holo目标是全兼容PGSQL；暂时没有实现的SQL功能转发给PostgreSQL来执行；
  - 外表访问：Holo读取ODPS，Mysql等；

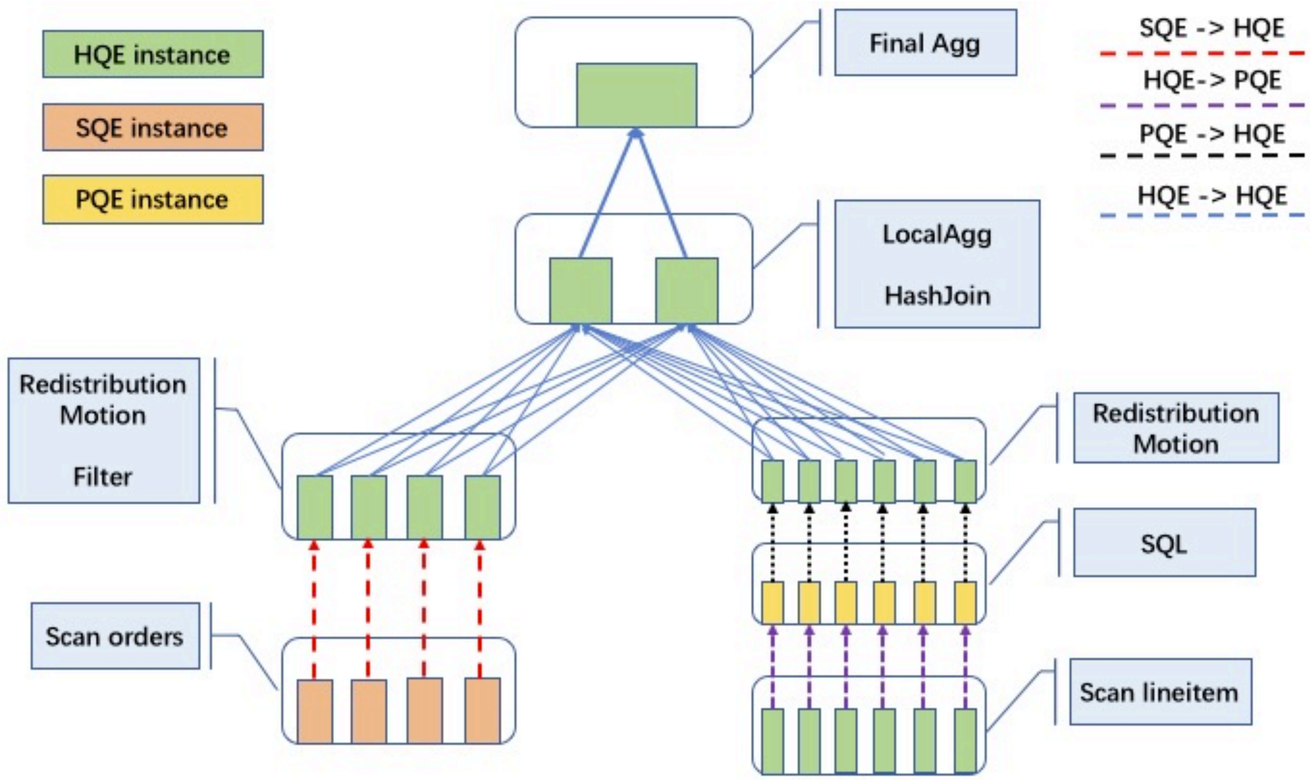
- 实现：
  - Property Enforce
  - 新增Property：Engine Type，表示单个算子需要在哪个执行引擎上被执行：
    - HQE: Hologres Query Engine;
    - PQE: Postgres Query Engine;
    - SQE: Foreign Query Engine



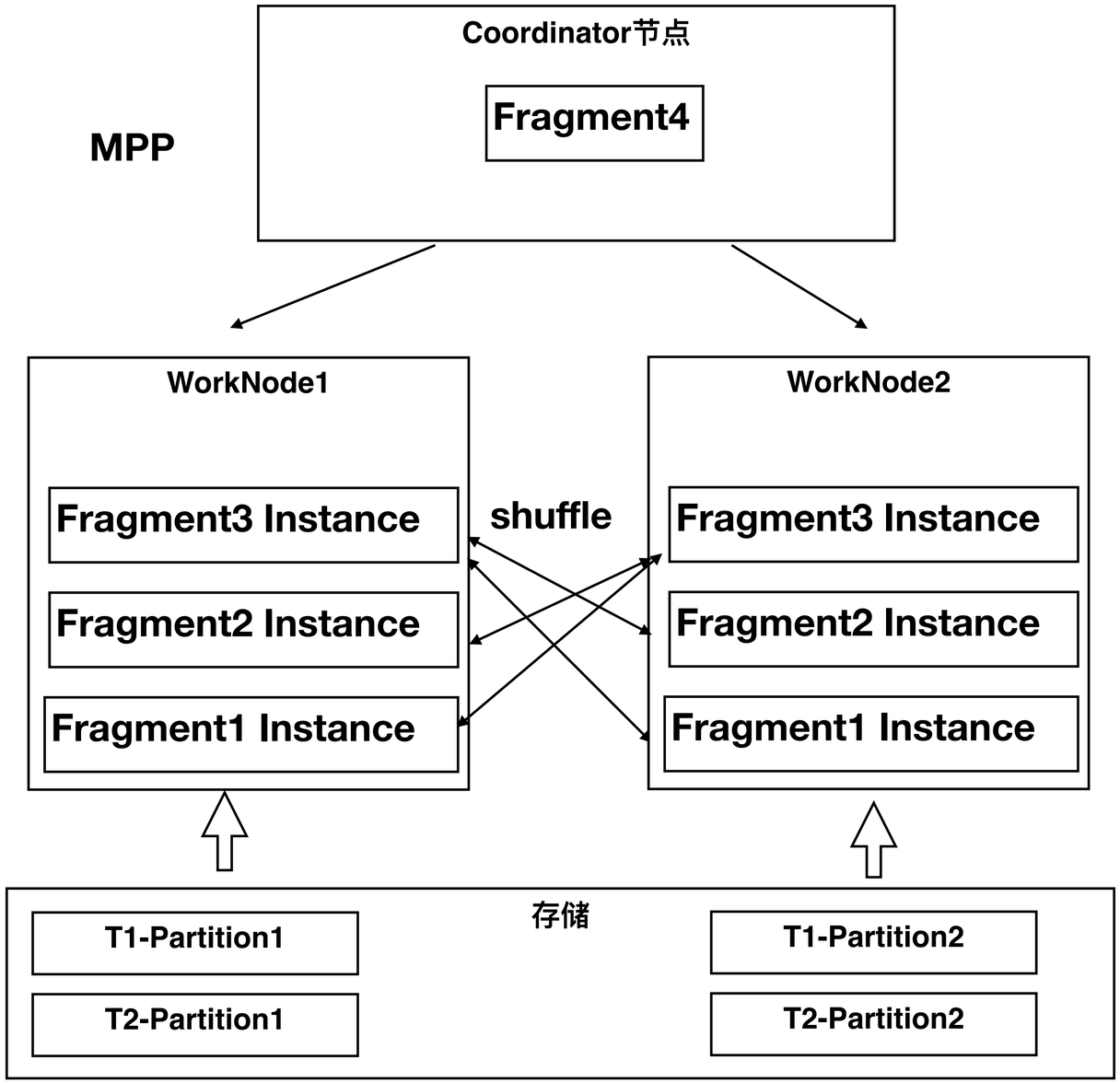
Hologres - 优化器（联邦查询）

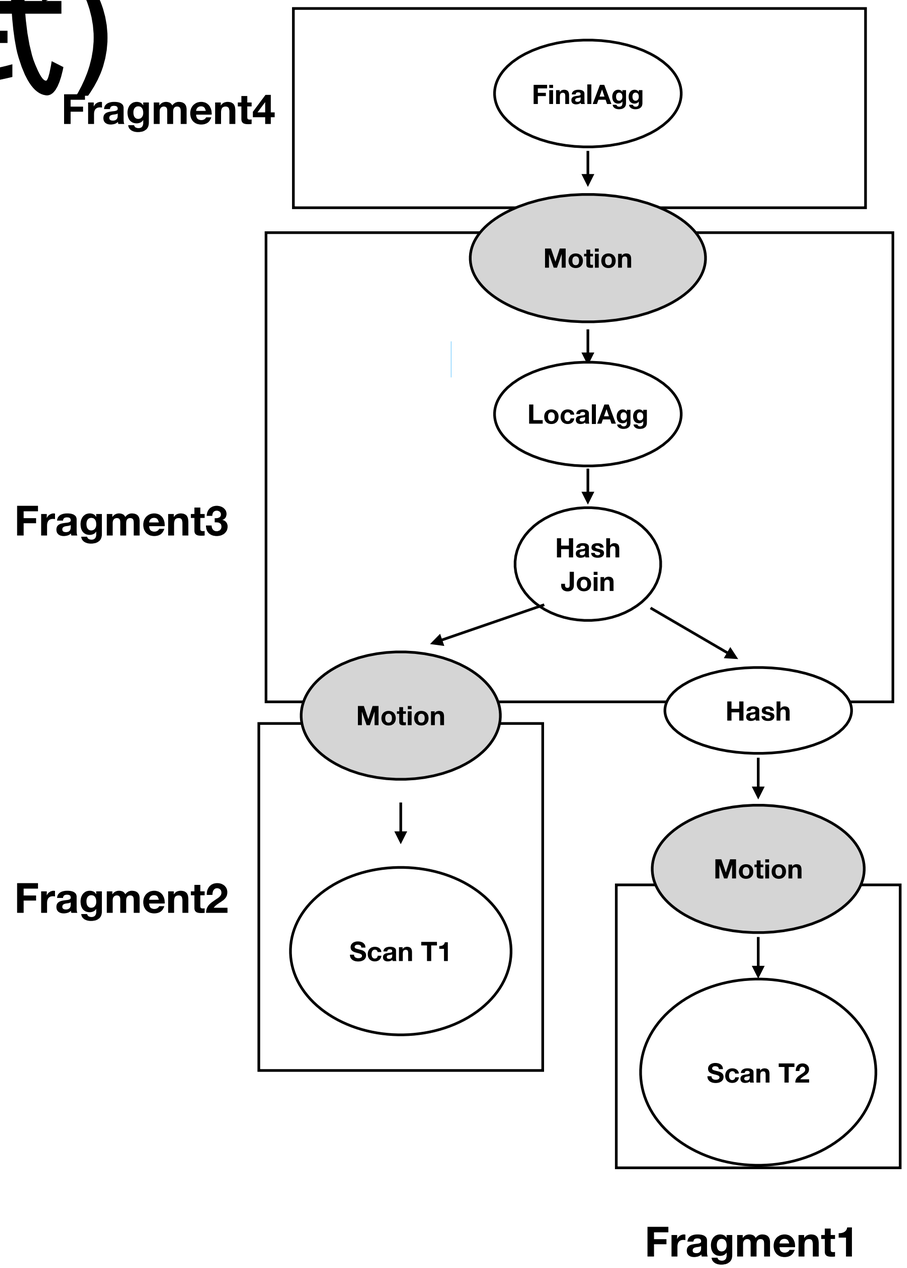
orders是ODPS上的表

```
SELECT COUNT(1)
 FROM
    lineitem t1
 JOIN
    orders t2
 ON
    t1.l_orderkey = t2.o_orderkey
 WHERE
    regexp_match(t1.l_linestatus, 'abc') && ARRAY['def', 'zxy']
    AND
    t2.o_orderstatus != 'abc';
```



Hologres - 执行器（执行方式）

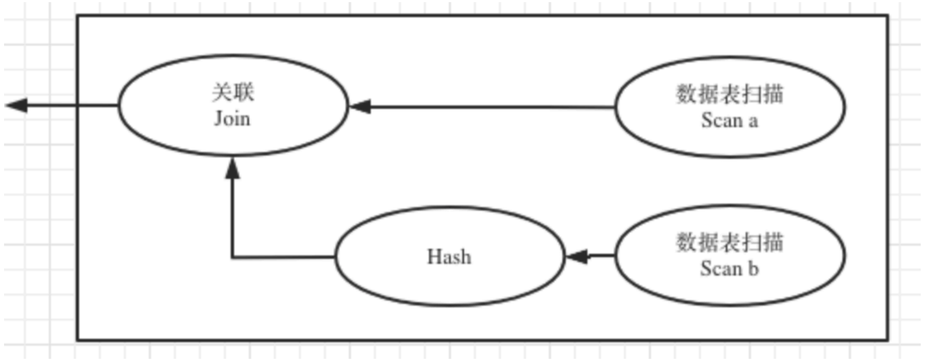




Hologres - 执行器（HOS）

- HOS背景：
  - 业务上有高并发查询
  - 存储计算分离延迟大，带宽大，需要高并发
  - 传统的线程/进程模型，阻塞IO

- Holo OS
  - 算子全异步 + 火山模型
  - （EC类似goroutine）
  - 基于seastar实现（future/promise/continuation）



- 如何基于异步算子增大读IO并发度
  - 传统火山模型：
    - 逐个算子执行；
    - prefetch和增大fragment Instance来优化
  - Holo根据运行时需要向B算子发送多个GetNext请求

```
future<> Open(const SeekParameters& parameters, ...)
future<RecordBatchPtr, bool> GetNext(...)
future<> Close(...)
```

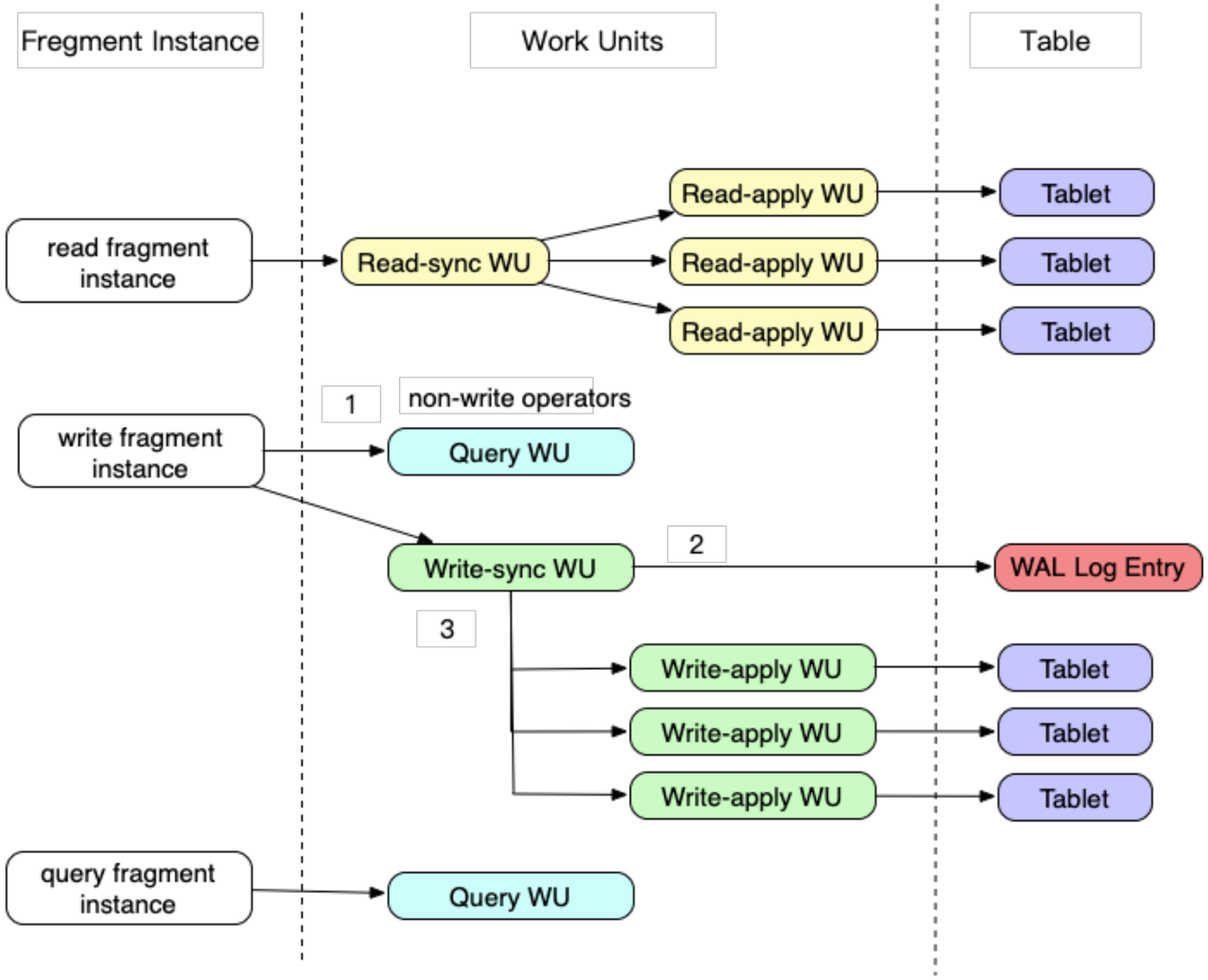
- 反压
  - 根据内存使用限制GetNext的个数；

Hologres - 执行器（Fragment，EC和WU）

- HOS3类EC Pools
  - Data-bound EC Pool:
    - WAL EC：执行write-sync WUs
    - Tablet EC：执行read-sync WUs和write-apply WUs
  - Query EC pool；query WU或read-apply WU
  - Background EC pool：compaction

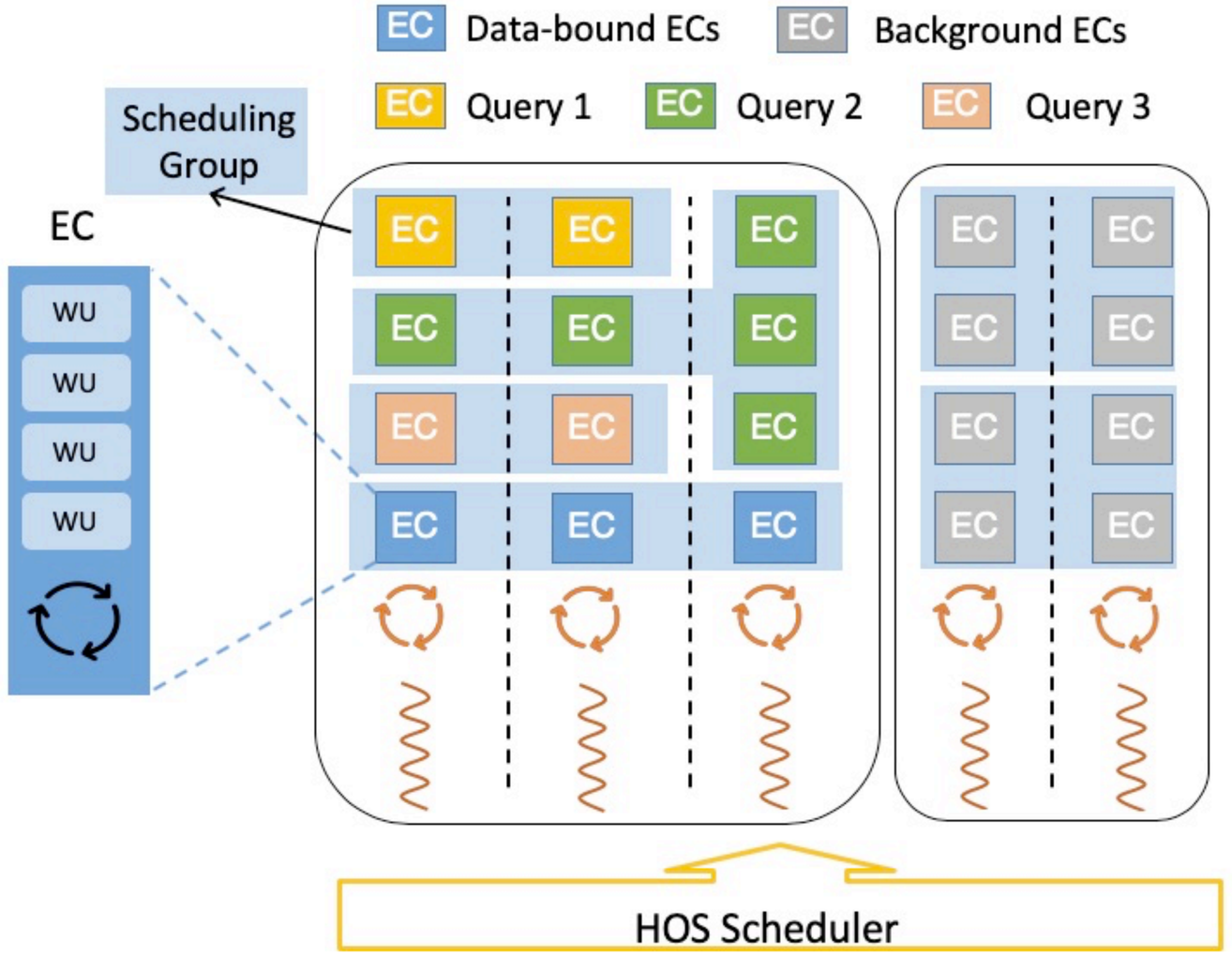
- EC两个队列（减少锁争抢）：
  - Internal Queue：自己产生的WU无锁；
  - Submit Queue：其他EC提交WU；

- Fragment instance被映射多个WU；



Hologres - 执行器（负载均衡）

- 迁移热点TGS
  - 标记fail；触发resource manager恢复机制；
- EC的均衡
  - 支持混合负载：大查询，高qps低延迟点查；
  - Scheduling Group调度组：
    - data-bound EC和query ECs的集合
    - 每个SG一个share值，代表被分配资源数量；
    - data-bound ECs: 分到单独SG，大share，确保，负责写入，负责read-sync；
    - query ECs: 每个查询一个SG，share想用；
- EC调度算法
  - ECPool调度EC，一个SG的EC分散在不同的ECPool上；
  - SG资源利用率：SG真实使用的share / SG分配的share；
  - EC的vruntime：单位share有效CPU使用；
  - 调度vruntime最小的EC；



$$EC\_share\_avg_i = EC\_share_i * \frac{\Delta T_{run}}{\Delta T_{run} + \Delta T_{spd} + \Delta T_{blk}}$$

$$SG\_share\_avg_i = \sum_{j=1}^{N} EC\_share\_avg_j$$

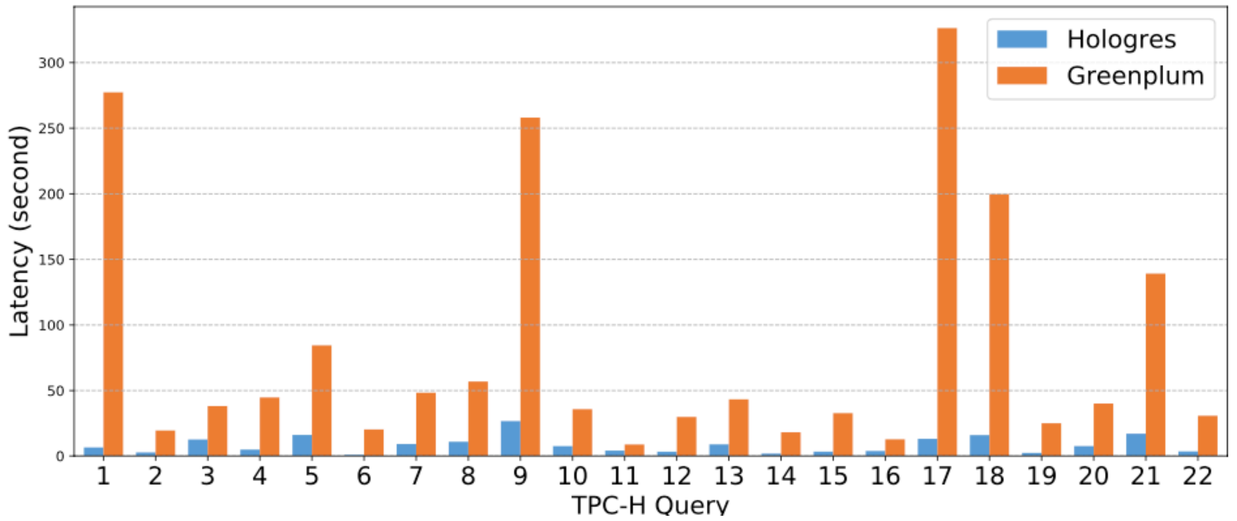
$$EC\_vshare_i = \frac{EC\_share_i * SG\_share_j}{SG\_share\_avg_j};$$

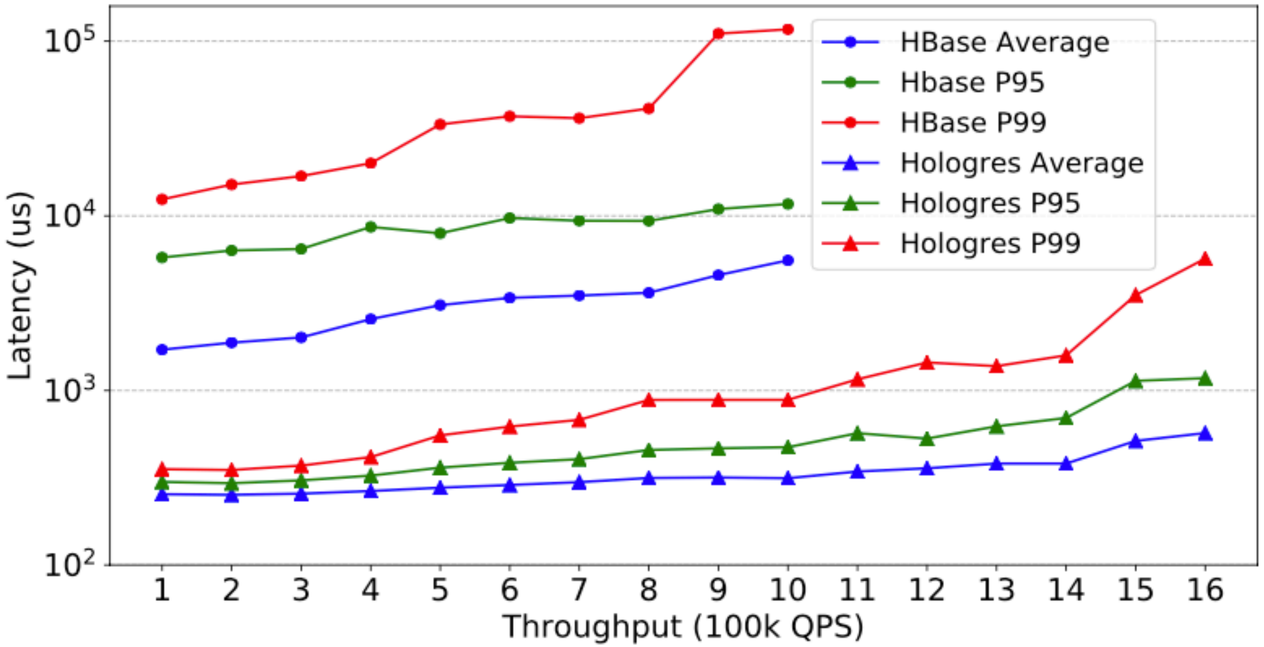
$$\Delta vruntime_i = \frac{\Delta CPU\_time_i}{EC\_vshare_i}$$

Hologres - 性能数据

- 环境
  - 8个物理机配置：24core，192GB内存，6TB本地盘；
  - Greenplum：8物理机，48segemnts，本地盘，列存；
  - HBase：8 region server，HDFS，本地盘；
  - Hologres：8 worknode行存+列存（6TB）；

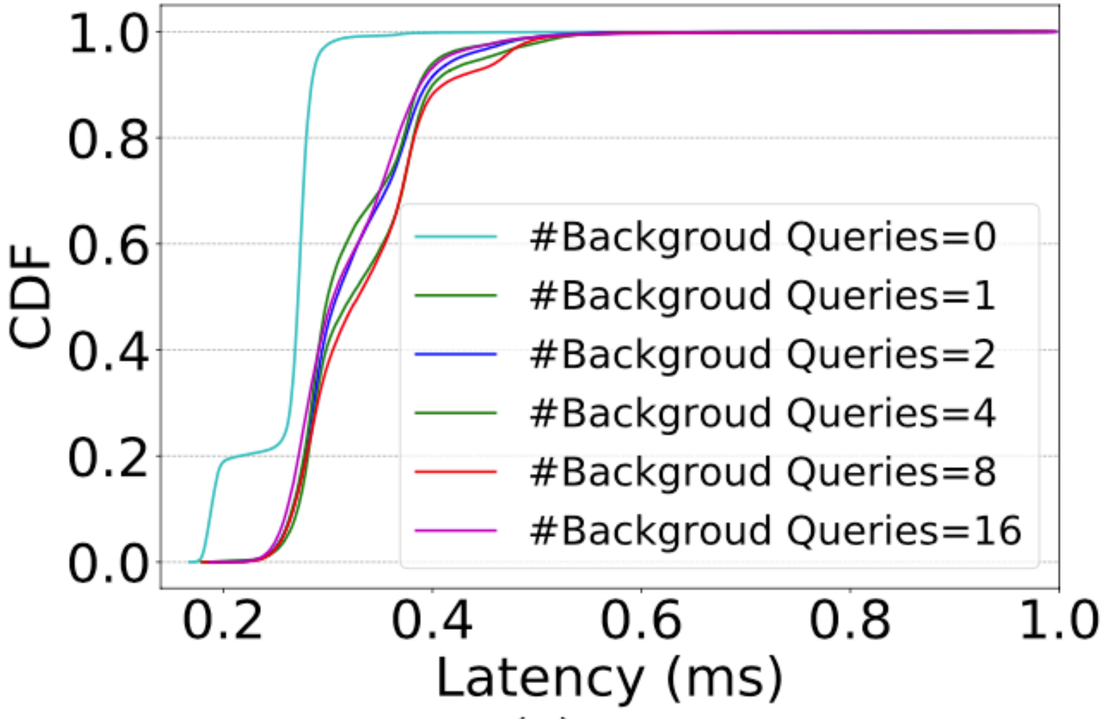
- TPCH
  - 10倍提升；
  - 原因：
    - 架构：HOS并发高，并发扫多个文件；GP的并发度 = segment个数；
    - 存储引擎：列存格式数据节凑；
    - 执行器：向量化，批量；
    - 优化器：dynamic filter；
- YCBS
  - 160K并发到1600K并发区间
  - HBase：延迟1000K以上大量超时；
  - Holo：99% 6ms以内，吞吐10~20倍；
  - 原因：
    - Hbase是线程模型，Holo是EC；



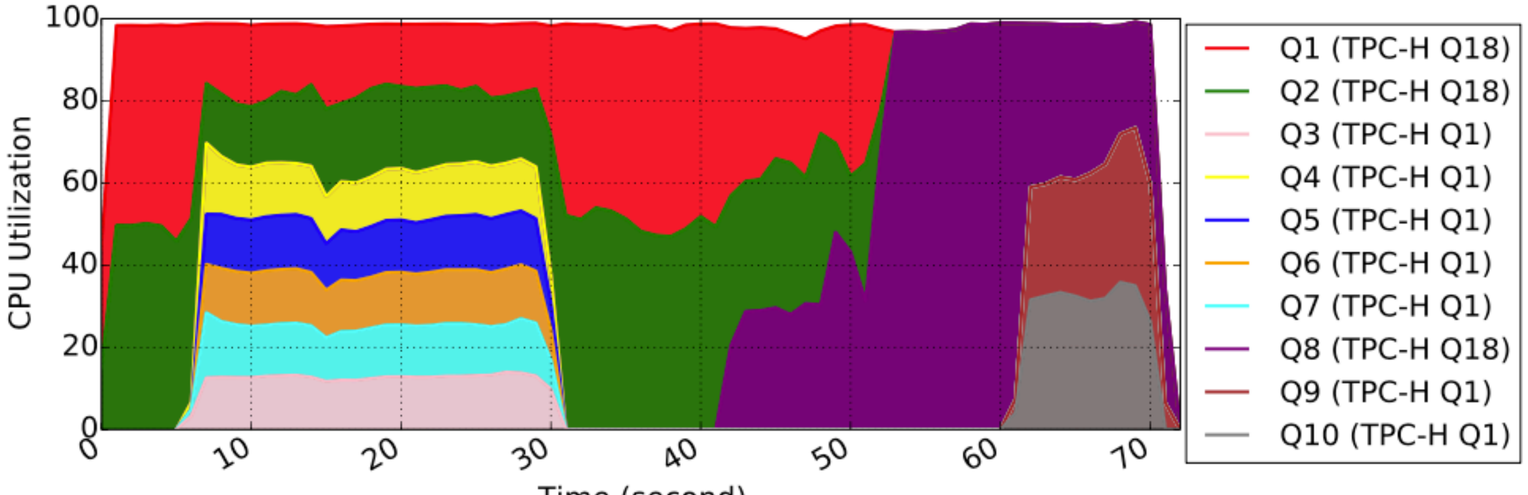


Hologres - HOS的调度

- 混合负载：增加tpch的同时，测试点查，1ms以内；

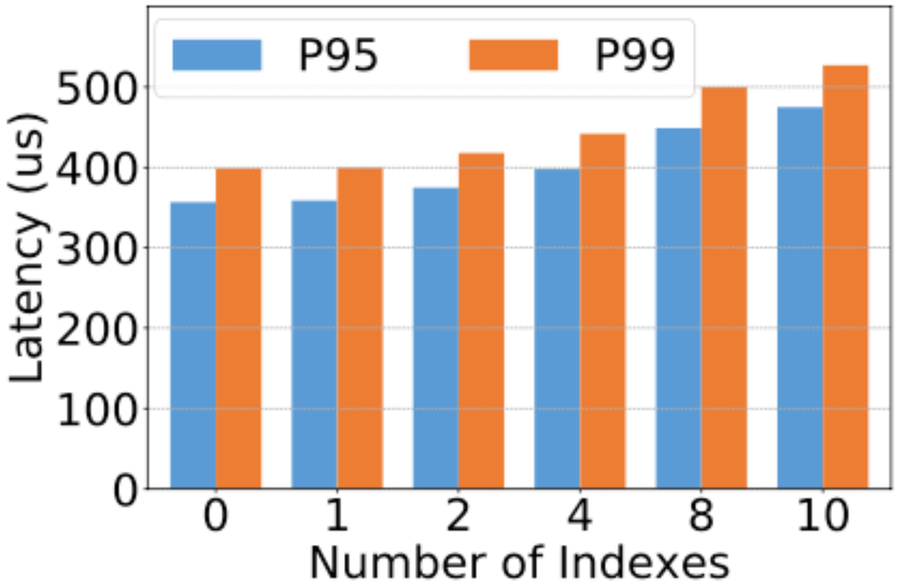


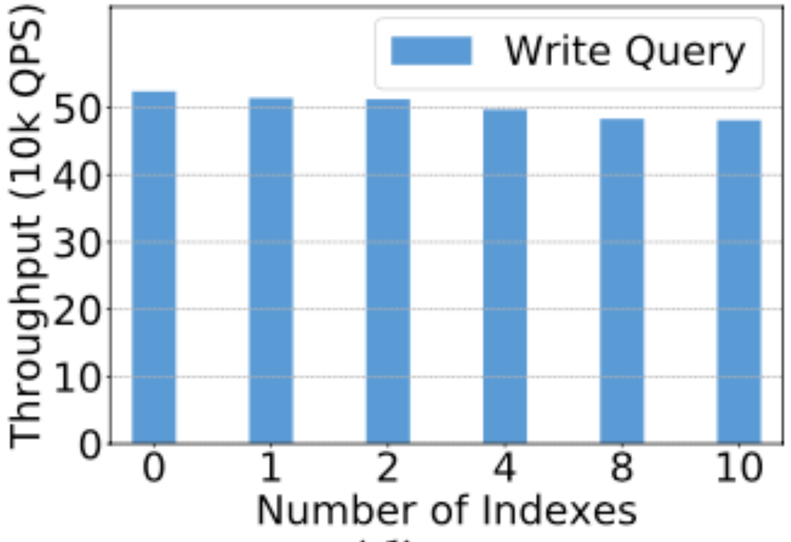
流量波动：在不同时间，加大压力，HOS能均摊压力



Hologres - 索引对写入的影响

- 索引数目增多，写入不受影响
  - 同一个TGS内的索引共享一个WAL；
  - 多个索引被WU并发更新；
  - backgroup EC负责compaction；





谢谢~

Hologres - 总结

- 存储引擎：
  - Holo支持行存+列存混合，适应TP点查+AP；

- 和传统扩展性：
  - GP需要reshard；
  - Holo增加Work Node + 迁移TGS；

- 混合负载：高并发+分析型

- 弱一致性：atomic write，read-your-write

seastar代码：

```
[1]  return conn->read_exactly(4).then(temporary_buffer<char> buf) {
[2]      int id = buf_to_id(buf);
[3]      return smp::submit_to(other_core, [id] {
[4]          return lookup(id);
[5]      });
[6]  }).then([this] (sstring result) {
[7]      return conn->write(result);
[8]   }
```