直播将于 19:30 准时开始，请耐心等待 ~

需求&问题收集

关注 SelectDB

数据写入能力优化
数据同步更快更稳定

廖　鑫 - 飞轮科技数据库研发工程师、Apache Doris Committer
赵长乐 - 数据库研发工程师
梅　祎 - 数据库研发工程师

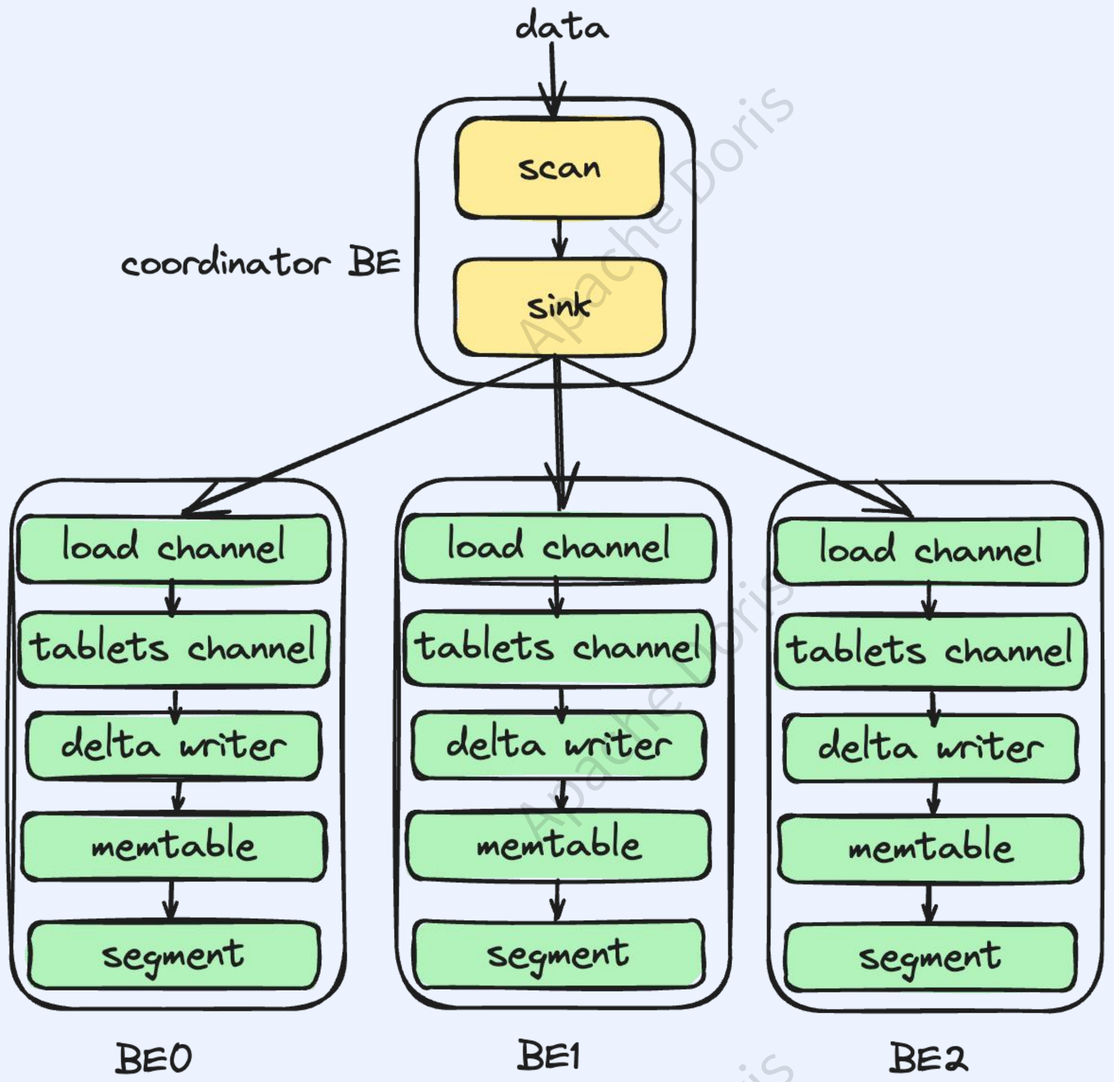
1 Memtable 前移，解锁极致数据传输速度

目录

Doris 现有导入实现

Coordinator BE：

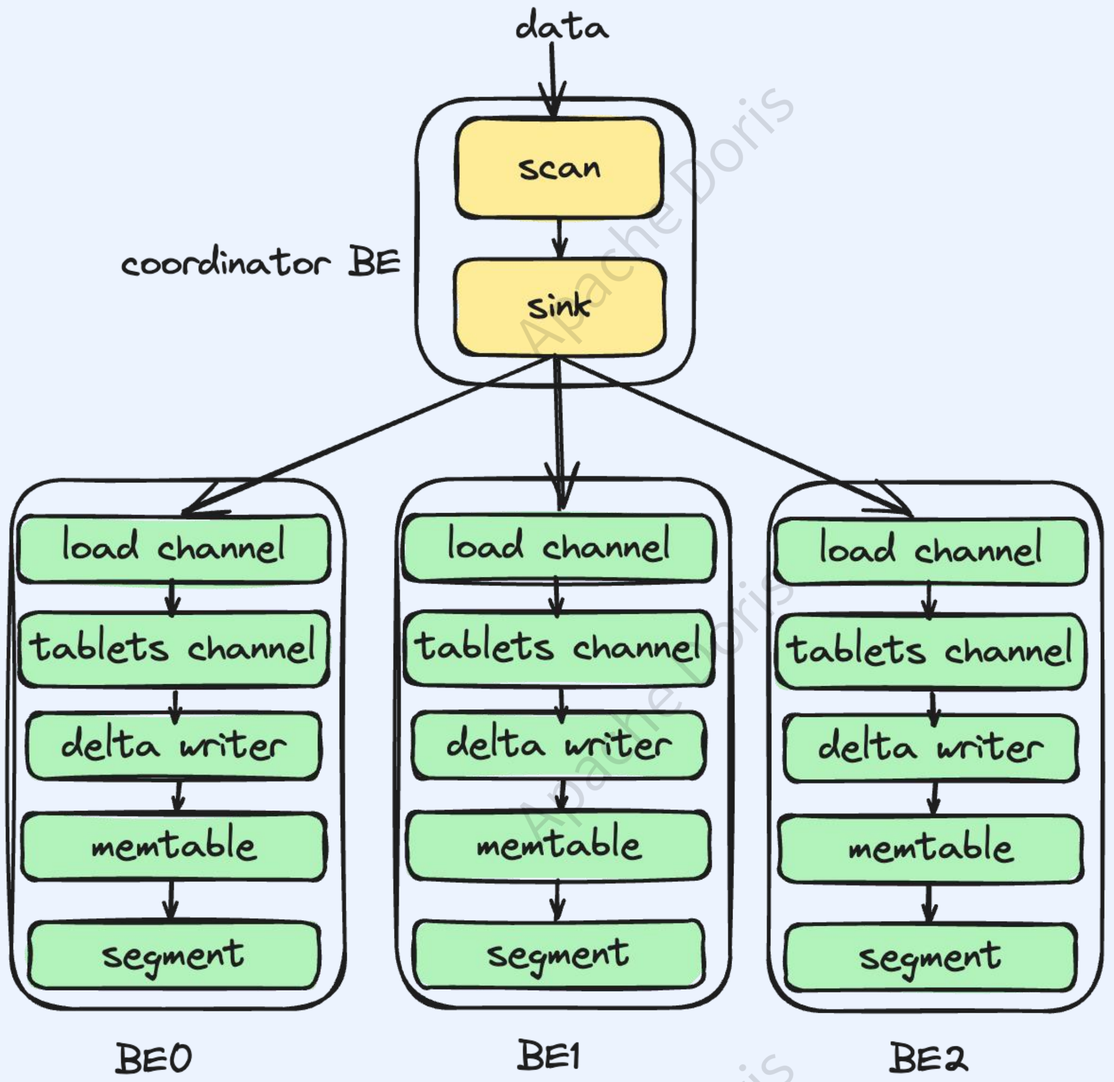
- 用户通过 Http 向发起请求，FE随机选出一个 BE 作为 Coordinator BE，并将用户请求重定向到该 BE
- Coordinaror BE 负责接收用户的请求，请求 FE 生成执行计划，并对导入事务进行管理
- Coordinator BE 调度执行导入计划
- Coordinator BE 对客户端的数据进行 scan，数据类型转换、清洗，通过 sink 发送给下游节点



Doris 现有导入实现

下游BE:

- 每个导入任务会建立一个 load channel，通过 load channel mgr 来管理
- Load channel 会创建 tablets channel 来执行具体的导入操作
- Tablets Channel 将数据分发给对应 Tablet，再由 DeltaWriter 将数据写入到 Tablet
- DeltaWriter 将数据先写到 Memtable
- Memtable对数据按key列进行排序
- Memtable 数据写满后，异步 flush 生成一个 Segment



现有导入存在的问题

计算和内存资源的冗余消耗

- 多个副本都要排序、压缩、编码
- rpc 序列化开销

Pingpong 模式 rpc，数据传输效率低

- 数据发送需要保证顺序，只能串行发送数据
- 每次发送数据必须等待一次网络 RTT + 处理数据时延

代码路径冗长

- LoadchannelMgr->LoadChanel->TabletsChannel...
- 复杂的 cancel 逻辑

导入负载控制复杂

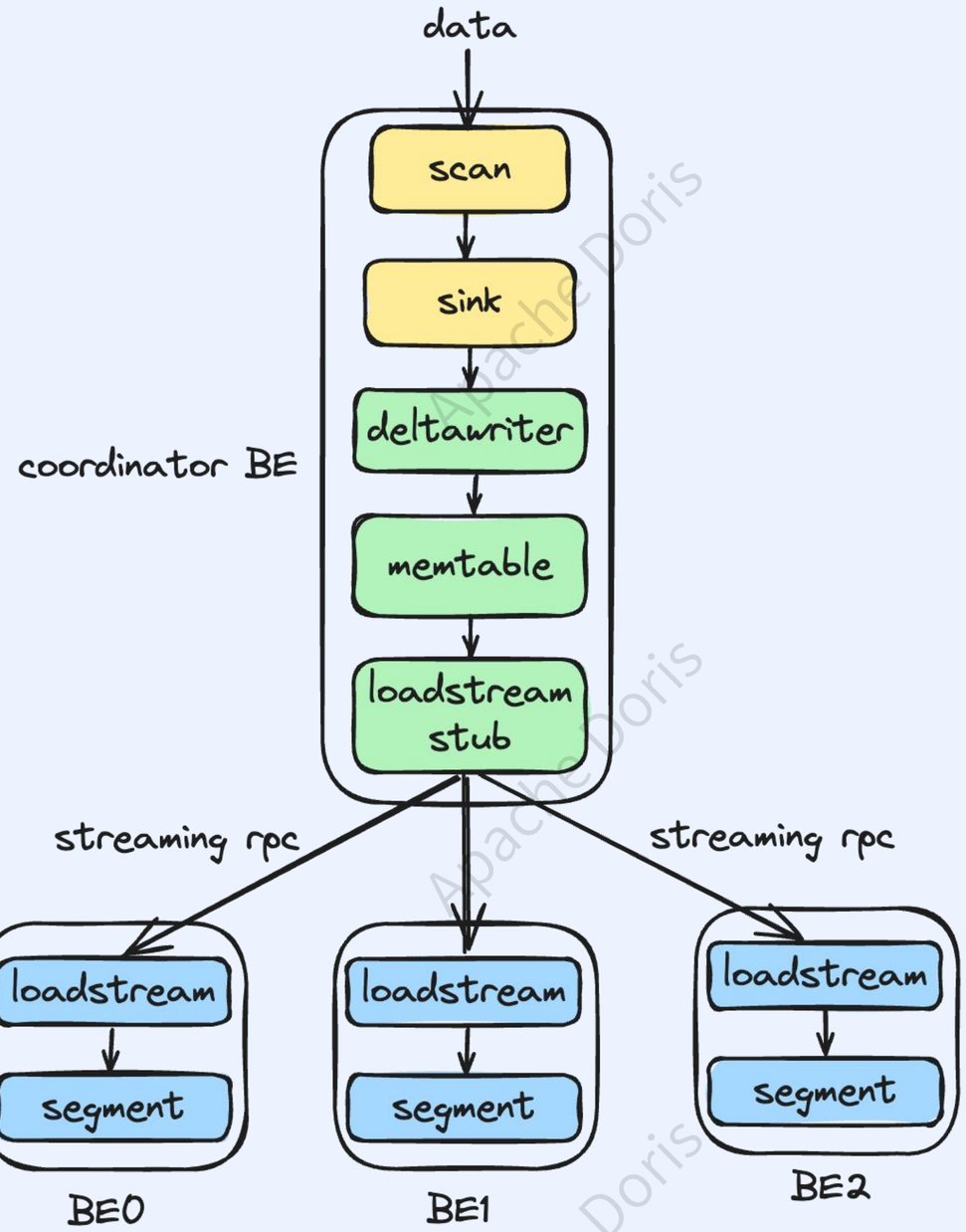
- 查询和计算不在同一节点，资源隔离不好做
- 内存控制复杂，反压实现难度大

目录

Memtable 前移架构

整体流程

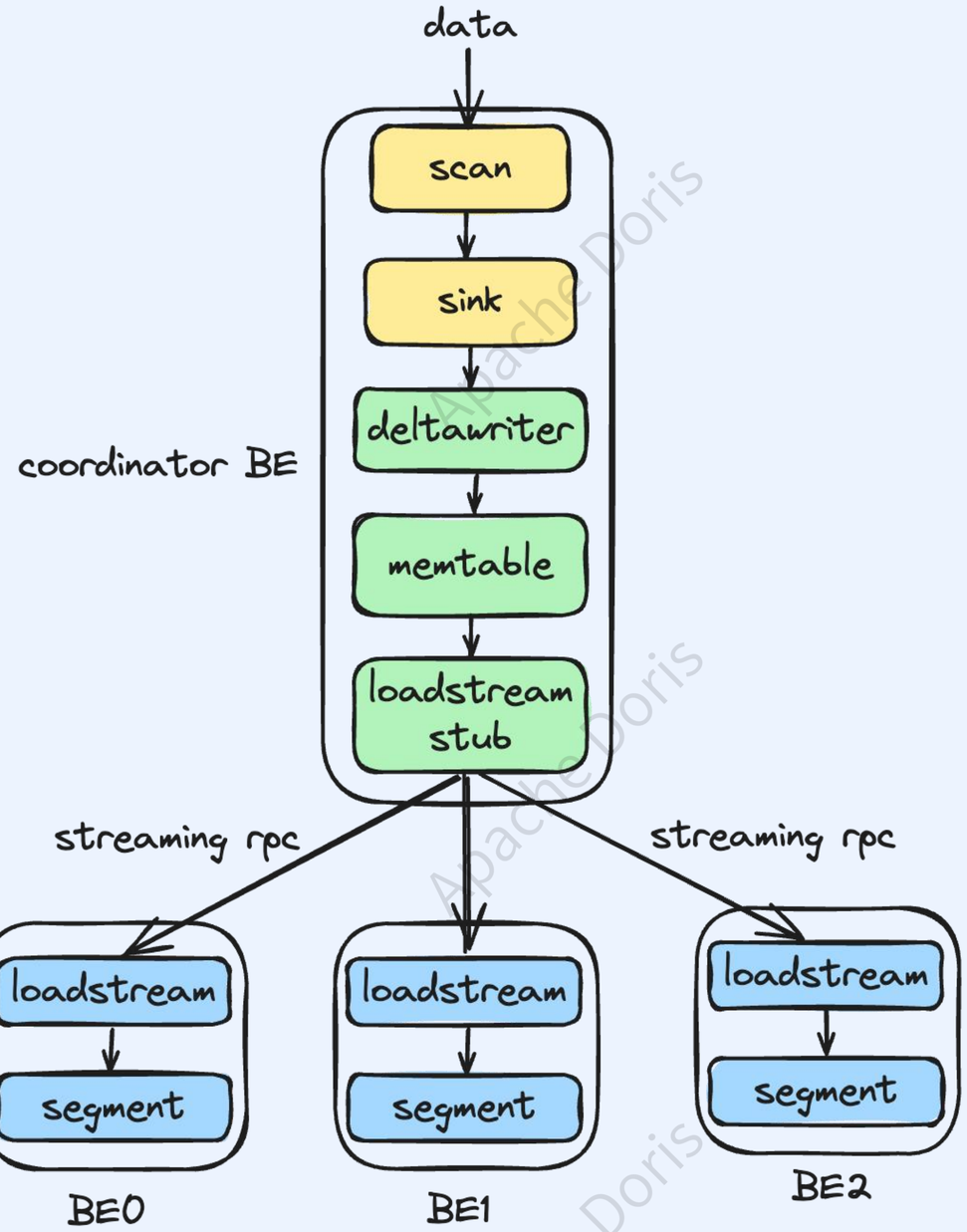
- Coordinator BE 接收用户的数据，调度执行导入计划
- 将 Memtable 从下游节点移动到 Coordinator BE 的 Sink node
- Sink node 将每个 tablet 的数据通过 delta writer 写到 Memtable，并进行排序
- memtable 写满后，生成 segment 文件，直接将 segment 文件发送给下游节点
- 下游节点接收 segment 文件，写入到磁盘



Memtable 前移实现

Streaming rpc

- 不再采用 pingpong 模型，使用 brpc 提供的 streaming rpc
- 上游通过 open_load_stream rpc 建立 streaming
- 通过 streaming 发送 segment 文件
- 下游收到 CLOSE_LOAD 消息，flush segment 文件，并完成 rowset 构建
- Report Commit 信息给上游节点
- 上游节点搜集完所有节点的 report 信息，开始 commit 事务



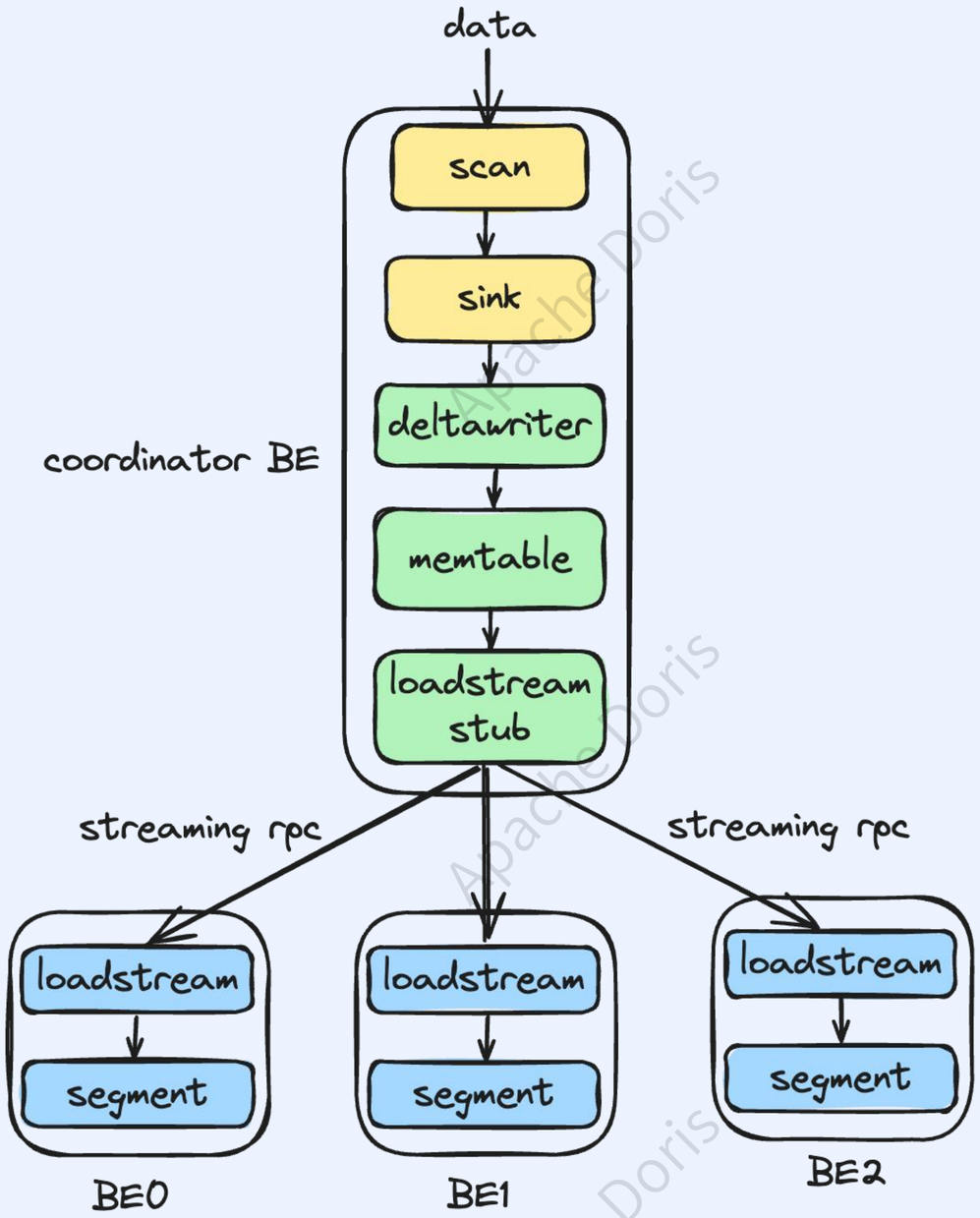
Memtable 前移实现

共享 delta writer 和 stream

- 同一个节点多个 sink node 共享 delta writer 和 stream
- 多个 sink node 的数据汇聚在一起，避免产生小 segment

流控/反压

- Streaming RPC 支持流控
- 调用 StreamWait 判断对端有没有堆积，如果有则进行等待



Memtable 前移的优势

更少的计算和内存资源

- 只有1个副本需要排序、压缩、编码
- 直接发送 segment 文件，减小了 rpc 序列化开销

Streaming rpc，更高效的数据传送

- 流式数据传输，减小 rpc pingpong 模式的等待时间
- Streaming rpc 天然保证了数据的顺序
- 传输的是压缩后的 segment 文件，传输效率更高

代码路径更简洁

- 没有 LoadchannelMgr -> LoadChanel -> TabletsChannel... 层层逻辑
- Cancel 流程更简单

导入负载控制简单

- 查询和计算在同一节点，方便 workload group 实现
- 内控控制不用分前后端，易于实现更加精准的反压控制

目录

Memtable 前移性能表现

测试环境：1 FE + 3 BE、每个节点 16C 64G、3 块高性能云盘

测试数据：tpch lineitem 表

测试方法：INSERT INTO lineitem_dst SELECT * FROM lineitem_src

性能表现：**导入性能提升超过 100%**

INSERT INTO SELECT	非前移	前移
lineitem 表 1 副本（3 亿行）	30.2s	11.1s
lineitem 表 3 副本（3 亿行）	47.4s	25.4s

目录

Memtable前移使用方式

Session 变量

Mysql 命令行执行：

set enable_memtable_on_sink_node = true

适用于：

- INSERT INTO
- S3 load
- Routine load

2.1 版本已默认打开该 session 变量

Stream load

通过 Header 配置：

curl --location-trusted -u root: -H "memtable_on_sink_node :true" -H "column_separator:," -H "columns:id,balance,last_access_time" -T /tmp/test.csv http://127.0.0.1:48037/api/db1/tbl1/_stream_load

通过 fe.conf 配置（默认 false）：

stream_load_default_memtable_on_sink_node = true;

目录

未来规划

- memtable 前移对倒排索引支持
- memtable 前移存算分离版本支持
- 持续性能优化，内存优化（数据湖，多 partition 多 bucket）

2 Auto partition 自动分区应用场景与性能比较

手动创建分区

业务数据，常见于日志、交易记录等明细表

分区数量多，需要经常手动变更

```
ALTER TABLE `DAILY_TRADE_VALUE`
ADD PARTITION p2022
VALUES LESS THAN ("2023-01-01");
```

```
CREATE TABLE `DAILY_TRADE_VALUE`
(
    `TRADE_DATE`              datev2 NOT NULL COMMENT '交易日期',
    `TRADE_ID`                varchar(40) NOT NULL COMMENT '交易编号',
    ......
)
UNIQUE KEY(`TRADE_DATE`, `TRADE_ID`)
PARTITION BY RANGE(`TRADE_DATE`)
(
    PARTITION p_2000 VALUES [('2000-01-01'), ('2001-01-01')),
    PARTITION p_2001 VALUES [('2001-01-01'), ('2002-01-01')),
    PARTITION p_2002 VALUES [('2002-01-01'), ('2003-01-01')),
    PARTITION p_2003 VALUES [('2003-01-01'), ('2004-01-01')),
    PARTITION p_2004 VALUES [('2004-01-01'), ('2005-01-01')),
    PARTITION p_2005 VALUES [('2005-01-01'), ('2006-01-01')),
    PARTITION p_2006 VALUES [('2006-01-01'), ('2007-01-01')),
    PARTITION p_2007 VALUES [('2007-01-01'), ('2008-01-01')),
    PARTITION p_2008 VALUES [('2008-01-01'), ('2009-01-01')),
    PARTITION p_2009 VALUES [('2009-01-01'), ('2010-01-01')),
    PARTITION p_2010 VALUES [('2010-01-01'), ('2011-01-01')),
    PARTITION p_2011 VALUES [('2011-01-01'), ('2012-01-01')),
    PARTITION p_2012 VALUES [('2012-01-01'), ('2013-01-01')),
    PARTITION p_2013 VALUES [('2013-01-01'), ('2014-01-01')),
    PARTITION p_2014 VALUES [('2014-01-01'), ('2015-01-01')),
    PARTITION p_2015 VALUES [('2015-01-01'), ('2016-01-01')),
    PARTITION p_2016 VALUES [('2016-01-01'), ('2017-01-01')),
    PARTITION p_2017 VALUES [('2017-01-01'), ('2018-01-01')),
    PARTITION p_2018 VALUES [('2018-01-01'), ('2019-01-01')),
    PARTITION p_2019 VALUES [('2019-01-01'), ('2020-01-01')),
    PARTITION p_2020 VALUES [('2020-01-01'), ('2021-01-01')),
    PARTITION p_2021 VALUES [('2021-01-01'), ('2022-01-01'))
)
DISTRIBUTED BY HASH(`TRADE_DATE`) BUCKETS 10
PROPERTIES (
  "replication_num" = "1"
);
```

动态分区解决方案

动态分区
Dynamic Partition

- 定期创建、定期回收
- 特定范围内的分区
- 一般用于实时数据收集

```
CREATE TABLE DAILY_TRADE_VALUE
(
`TRADE_DATE`    DATEV2 NOT NULL COMMENT '交易日期',
`TRADE_ID`      VARCHAR(40) NOT NULL COMMENT '交易编号',
......
)
PARTITION BY RANGE(`TRADE_DATE`)
DISTRIBUTED BY HASH(`TRADE_DATE`) BUCKETS 10
PROPERTIES
(
    "dynamic_partition.enable" = "true",
    "dynamic_partition.time_unit" = "DAY",
    "dynamic_partition.start" = "-7",
    "dynamic_partition.end" = "3",
    "dynamic_partition.prefix" = "p",
    "dynamic_partition.buckets" = "10"
);
```

自动分区解决方案

自动分区
Auto
Partition

- 根据插入数据自动创建
- 任意数据分布
- 适用于各类数据

```
CREATE TABLE DAILY_TRADE_VALUE
(
`TRADE_DATE`    DATEV2 NOT NULL COMMENT '交易日期',
`TRADE_ID`      VARCHAR(40) NOT NULL COMMENT '交易编号',
.....
)
AUTO PARTITION BY RANGE
(DATE_TRUNC(`TRADE_DATE`, 'DAY'))
()
DISTRIBUTED BY HASH(`TRADE_DATE`) BUCKETS 10
PROPERTIES
(
    .....
);
```

自动创建分区

```
mysql [test]>CREATE TABLE `DAILY_TRADE_VALUE`
    -> (
    ->     `TRADE_DATE`              datev2 NOT NULL COMMENT '交易日期',
    ->     `TRADE_ID`                varchar(40) NOT NULL COMMENT '交易编号'
    -> )
    -> UNIQUE KEY(`TRADE_DATE`, `TRADE_ID`)
    -> AUTO PARTITION BY RANGE (date_trunc(`TRADE_DATE`, 'year'))
    -> (
    -> )
    -> DISTRIBUTED BY HASH(`TRADE_DATE`) BUCKETS 10
    -> PROPERTIES (
    ->   "replication_num" = "1"
    -> );
Query OK, 0 rows affected (0.00 sec)

mysql [test]>show partitions from DAILY_TRADE_VALUE;
Empty set (0.00 sec)
```

```
mysql [test]>insert into `DAILY_TRADE_VALUE` values ('2012-12-13', 1), ('2008-02-03', 2), ('2014-11-11', 3);
Query OK, 3 rows affected (0.16 sec)
{'label':'label_f0a038acc7644776_802cb55a6e0b3f24', 'status':'VISIBLE', 'txnId':'42045'}

mysql [test]>show partitions from DAILY_TRADE_VALUE;
+-------------+-----------------+----------------+---------------------+--------+--------------+----------------------------------------------------------------------------
--+-----------------+---------+----------------+---------------+---------------------+--------------------+--------------------------+----------+------------+-------------
-----------+-----------+--------------------+--------------+
| PartitionId | PartitionName   | VisibleVersion | VisibleVersionTime  | State  | PartitionKey | Range
  | DistributionKey | Buckets | ReplicationNum | StorageMedium | CooldownTime        | RemoteStoragePolicy | LastConsistencyCheckTime | DataSize | IsInMemory | ReplicaAllocat
ion        | IsMutable | SyncWithBaseTables | UnsyncTables |
+-------------+-----------------+----------------+---------------------+--------+--------------+----------------------------------------------------------------------------
--+-----------------+---------+----------------+---------------+---------------------+--------------------+--------------------------+----------+------------+-------------
-----------+-----------+--------------------+--------------+
| 619214      | p20080101000000 | 2              | 2024-04-14 18:13:42 | NORMAL | TRADE_DATE   | [types: [DATEV2]; keys: [2008-01-01]; ..types: [DATEV2]; keys: [2009-01-01];
) | TRADE_DATE      | 10      | 1              | HDD           | 9999-12-31 23:59:59 |                    | NULL                     | 0.000    | false      | tag.location.d
efault: 1 | true      | true               | NULL         |
| 619193      | p20120101000000 | 2              | 2024-04-14 18:13:42 | NORMAL | TRADE_DATE   | [types: [DATEV2]; keys: [2012-01-01]; ..types: [DATEV2]; keys: [2013-01-01];
) | TRADE_DATE      | 10      | 1              | HDD           | 9999-12-31 23:59:59 |                    | NULL                     | 0.000    | false      | tag.location.d
efault: 1 | true      | true               | NULL         |
| 619172      | p20140101000000 | 2              | 2024-04-14 18:13:42 | NORMAL | TRADE_DATE   | [types: [DATEV2]; keys: [2014-01-01]; ..types: [DATEV2]; keys: [2015-01-01];
) | TRADE_DATE      | 10      | 1              | HDD           | 9999-12-31 23:59:59 |                    | NULL                     | 0.000    | false      | tag.location.d
efault: 1 | true      | true               | NULL         |
+-------------+-----------------+----------------+---------------------+--------+--------------+----------------------------------------------------------------------------
--+-----------------+---------+----------------+---------------+---------------------+--------------------+--------------------------+----------+------------+-------------
-----------+-----------+--------------------+--------------+
3 rows in set (0.00 sec)
```

LIST 自动分区

```
mysql> CREATE TABLE `str_table` (
    ->     `city` VARCHAR NOT NULL
    -> )
    -> DUPLICATE KEY(`city`)
    -> AUTO PARTITION BY LIST (`city`)
    -> ()
    -> DISTRIBUTED BY HASH(`city`) BUCKETS 10
    -> PROPERTIES (
    -> "replication_allocation" = "tag.location.default: 1"
    -> );
Query OK, 0 rows affected (0.09 sec)

mysql> insert into str_table values ("Beijing"), ("Shanghai"),
("Los_Angeles");
Query OK, 3 rows affected (0.25 sec)
```

```
mysql> show partitions from str_table;
+-------------+-----------------+----------------+
| PartitionId | PartitionName   | VisibleVersion |
+-------------+-----------------+----------------+
| 619550      | pBeijing7       | 2              |
| 619508      | pLos5fAngeles11 | 2              |
| 619529      | pShanghai8      | 2              |
+-------------+-----------------+----------------+
3 rows in set (0.12 sec)
```

自动分区功能总结

语法支持

1. AUTO RANGE PARTITION

```
AUTO PARTITION BY RANGE (FUNC_CALL_EXPR)
()
FUNC_CALL_EXPR ::= date_trunc ( <partition_column>, '<interval>' )
```

2. AUTO LIST PARTITION

```
AUTO PARTITION BY LIST (`partition_col`)
()
```

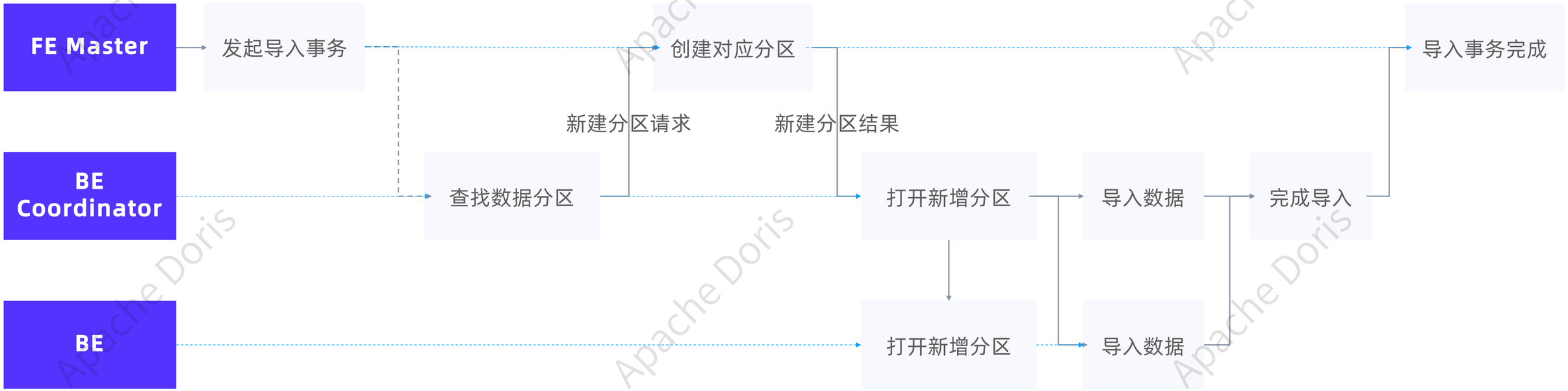
限制条件

1. LIST 自动分区**仅支持一个分区列**，每个自动创建的分区仅包含一个分区值，分区名长度不能超过 50
2. RANGE 自动分区支持多个分区列，**分区列类型必须为 DATE 或 DATETIME**
3. LIST 自动分区支持 NULLABLE 分区列和实际插入 NULL 值；RANGE 自动分区不支持 NULLABLE 分区列

自动分区和动态分区对比

	自动分区	动态分区
创建方式	根据导入数据在分区列的值自动创建	定期创建特定范围的分区
支持类型	LIST 和 RANGE（日期）分区	RANGE（日期）分区
灵活性	好	较差
性能影响	影响较小	无影响
删除方式	手动删除	定期回收

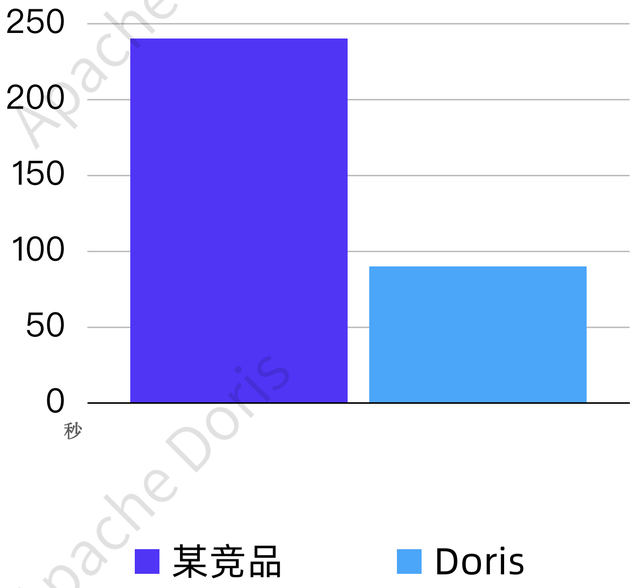
自动分区导入流程



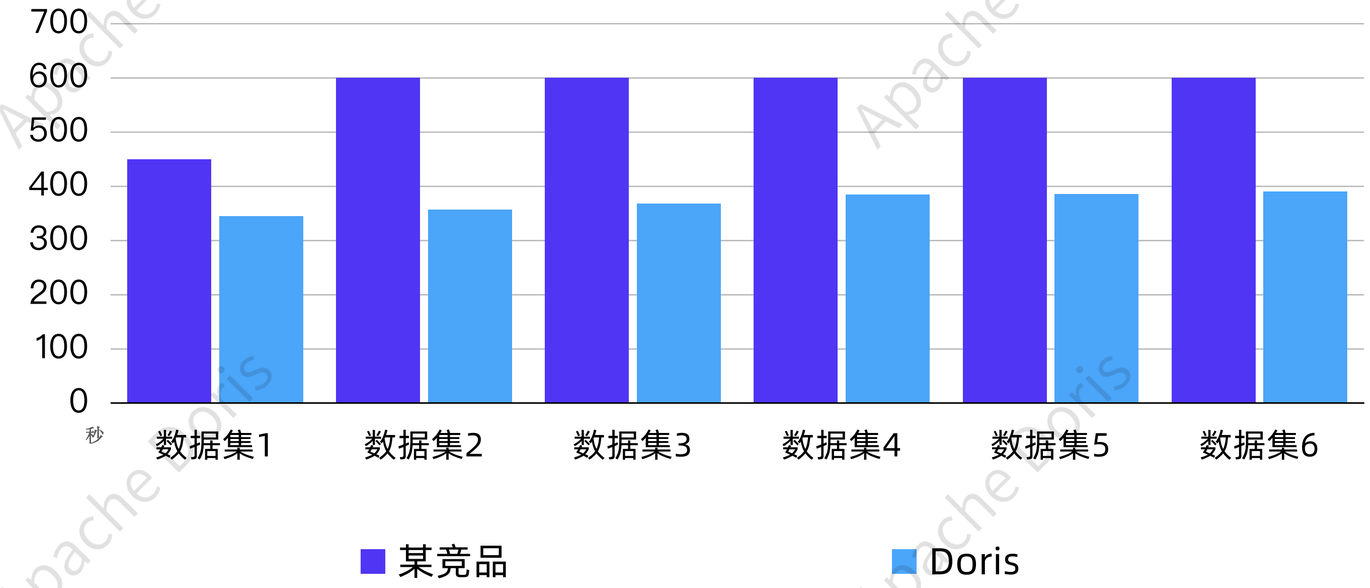
注意

目前，FE Master 会在“创建对应分区”时完成整个对应分区的创建事务，对应分区可见。如果导入流程最终失败，所创建的分区依然存在，不会回滚。

自动分区性能表现



场景1



场景2

场景描述

场景 1：1FE+1BE 环境，数字 1-1000 各重复 10000 次，生成 1 亿条数据并导入

场景 2：1FE+3BE 环境，每个数据集 1 亿行，涉及 2000 个分区，6 个数据集并行导入

3 Group Commit 服务端攒批：高频实时导入

背景

问题：

小数据量高频写入 Doris 时，经常遇到版本太多或事务超限的问题；

使用 JDBC 高频写入，FE 压力较大且执行耗时；

解决方法：

Group Commit（服务端攒批）

Group Commit 模式

async_mode(异步模式)

- 将数据写入 Write Ahead Log，导入立即返回。
- Doris异步提交数据，提交之后数据可见。

sync_mode(同步模式)

- Doris 将多个导入在一个事务提交，事务提交后导入返回。
- 适用于高并发写入场景，且在导入完成后要求数据立即可见。

off_mode(关闭模式)

不开启 Group Commit，保持以上三种导入方式的默认行为。

INSERT INTO values、StreamLoad、HttpStream 三种导入方式，可以结合 GroupCommit 使用

async_mode 异步

```
# 配置session变量开启 group commit (默认为off_mode),开启异步模式
mysql> set group_commit = async_mode;

# 这里返回的label是 group_commit 开头的，可以区分出是否使用了 group commit
mysql> insert into dt values(1, 'Bob', 90), (2, 'Alice', 99);
Query OK, 2 rows affected (0.05 sec)
{'label':'group_commit_a145ce07f1c972fc-bd2c54597052a9ad', 'status':'PREPARE', 'txnId':'181508'}

# 可以看出这个 label，txn_id 和上一个相同，说明是攒到了同一个导入任务中
mysql> insert into dt(id, name) values(3, 'John');
Query OK, 1 row affected (0.01 sec)
{'label':'group_commit_a145ce07f1c972fc-bd2c54597052a9ad', 'status':'PREPARE', 'txnId':'181508'}

# 不能立刻查询到
mysql> select * from dt;
Empty set (0.01 sec)

# 10秒后可以查询到，可以通过表属性 group_commit_interval 控制数据可见延迟。
mysql> select * from dt;
+------+-------+-------+
| id   | name  | score |
+------+-------+-------+
|    1 | Bob   |    90 |
|    2 | Alice |    99 |
|    3 | John  |  NULL |
+------+-------+-------+
3 rows in set (0.02 sec)
```

async_mode 异步

```
# 导入时在header中增加"group_commit:async_mode"配置

curl --location-trusted -u {user}:{passwd} -T data.csv -H "group_commit:async_mode"
{
    "TxnId": 7009,
    "Label": "group_commit_c84d2099208436ab_96e33fda01eddba8",
    "Comment": "",
    "GroupCommit": true,
    "Status": "Success",
    "Message": "OK",
    "NumberTotalRows": 2,
    "NumberLoadedRows": 2,
    "NumberFilteredRows": 0,
    "NumberUnselectedRows": 0,
    "LoadBytes": 19,
    "LoadTimeMs": 35,
    "StreamLoadPutTimeMs": 5,
    "ReadDataTimeMs": 0,
    "WriteDataTimeMs": 26
}

# 返回的GroupCommit为true，说明进入了group commit的流程
# 返回的Label是group_commit开头的，是真正消费数据的导入关联的label
```

sync_mode 同步

```
# 配置session变量开启 group commit (默认为off_mode),开启同步模式
mysql> set group_commit = sync_mode;

# 这里返回的 label 是 group_commit 开头的，可以区分出是否谁用了 grou
mysql> insert into dt values(4, 'Bob', 90), (5, 'Alice', 99);
Query OK, 2 rows affected (10.06 sec)
{'label':'group_commit_d84ab96c09b60587_ec455a33cb0e9e87', 's

# 数据可以立刻读出
mysql> select * from dt;
+------+-------+-------+
| id   | name  | score |
+------+-------+-------+
|    1 | Bob   |    90 |
|    2 | Alice |    99 |
|    3 | John  |  NULL |
|    4 | Bob   |    90 |
|    5 | Alice |    99 |
+------+-------+-------+
5 rows in set (0.03 sec)
```

```
# 导入时在header中增加"group_commit:sync_mode"配置

curl --location-trusted -u {user}:{passwd} -T data.csv -H "group_commit:sync_mode"
{
    "TxnId": 3009,
    "Label": "group_commit_d941bf17f6efcc80_ccf4afdde9881293",
    "Comment": "",
    "GroupCommit": true,
    "Status": "Success",
    "Message": "OK",
    "NumberTotalRows": 2,
    "NumberLoadedRows": 2,
    "NumberFilteredRows": 0,
    "NumberUnselectedRows": 0,
    "LoadBytes": 19,
    "LoadTimeMs": 10044,
    "StreamLoadPutTimeMs": 4,
    "ReadDataTimeMs": 0,
    "WriteDataTimeMs": 10038
}

# 返回的GroupCommit为true，说明进入了group commit的流程
# 返回的Label是group_commit开头的，是真正消费数据的导入关联的label
```

使用 JDBC

- **使用 Prepare 协议**

```
"?useLocalSessionState=true"
+ "&useServerPrepStmts=true"
+ "&rewriteBatchedStatements=true"
+ "&cachePrepStmts=true"
+ "&prepStmtCacheSqlLimit=10000"
+ "&prepStmtCacheSize=50"
```

- **使用 GroupCommit**

```
"&sessionVariables=group_commit=async_mode"
```

observer fe 也可写入

```
private static void groupCommitInsertBatch() throws Exception {
    Class.forName(JDBC_DRIVER);
    // add rewriteBatchedStatements=true and cachePrepStmts=true in JDBC url
    // set session variables by sessionVariables=group_commit=async_mode in
    try (Connection conn = DriverManager.getConnection(
            String.format(URL_PATTERN + "&rewriteBatchedStatements=true&cach

        String query = "insert into " + TBL + " values(?, ?, ?)";
        try (PreparedStatement stmt = conn.prepareStatement(query)) {
            for (int j = 0; j < 5; j++) {
                // 10 rows per insert
                for (int i = 0; i < INSERT_BATCH_SIZE; i++) {
                    stmt.setInt(1, i);
                    stmt.setString(2, "name" + i);
                    stmt.setInt(3, i + 10);
                    stmt.addBatch();
                }
                int[] result = stmt.executeBatch();
            }
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

提交参数

当满足时间间隔 (默认为 10 秒) 或数据量 (默认为 128 MB) 其中一个条件时，会自动提交数据。

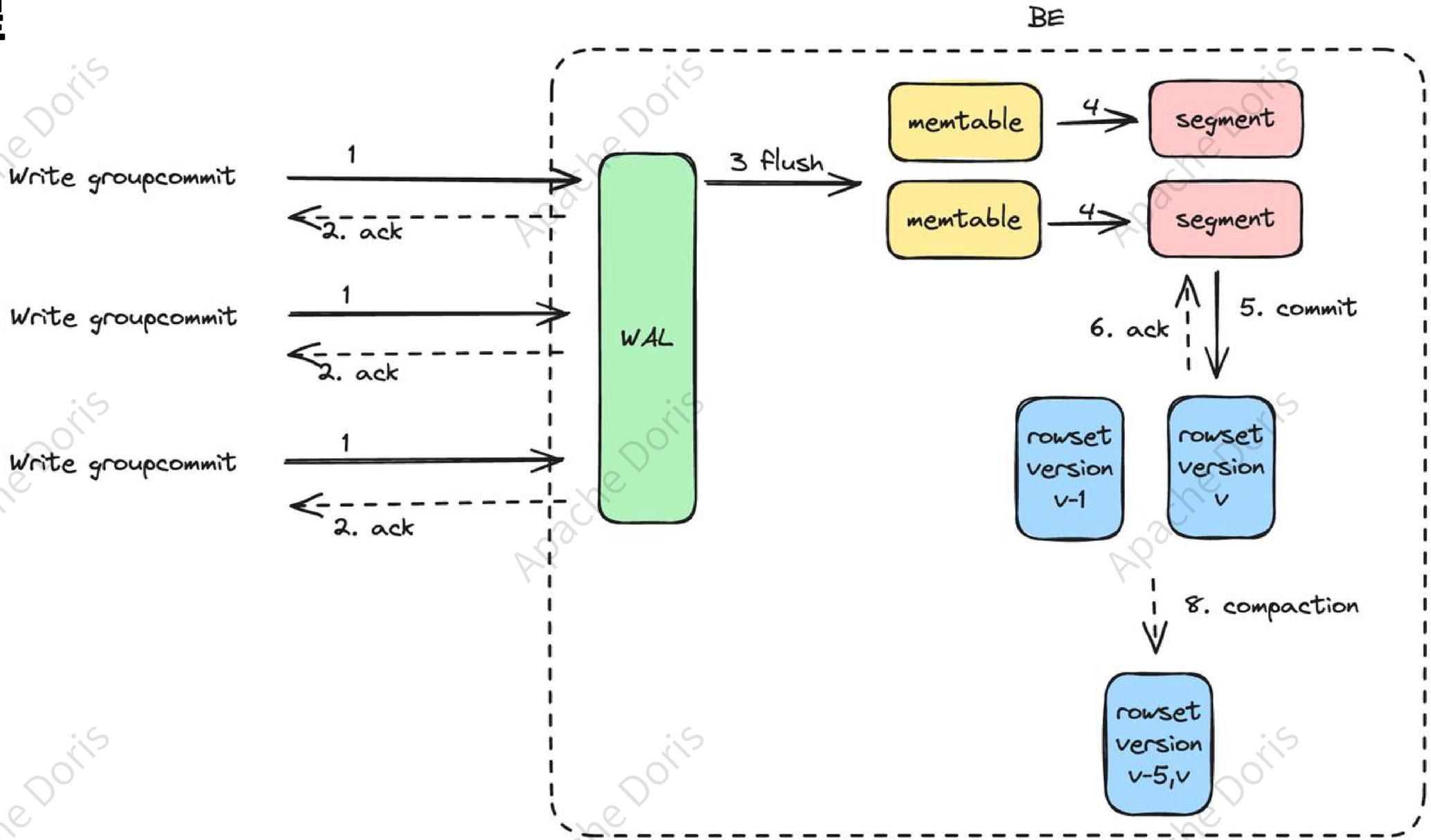
修改提交间隔

```
# 修改提交间隔为 2 秒
ALTER TABLE dt SET ("group_commit_interval_ms" = "2000");
```

修改提交数据量

```
# 修改提交数据量为 128MB
ALTER TABLE dt SET ("group_commit_data_bytes" = "134217728");
```

流程


BE
Write groupcommit
1
2. ack
Write groupcommit
1
2. ack
Write groupcommit
1
2. ack
WAL
3 flush
memtable
4
segment
memtable
4
segment
6. ack
5. commit
rowset version v-1
rowset version v
8. compaction
rowset version v-5,v

StreamLoad 日志场景

- **1 FE**

8核，16GB 内存

1 * 200GB SSD 云磁盘

- **3 BE**

16核，64GB 内存

1 * 2TB SSD 云磁盘

- **1 客户端**

16核，64GB 内存

1 * 100GB SSD 云磁盘

导入方式	单并发数据量	并发数	耗时(秒)	导入速率(行/秒)	导入吞吐(MB/秒)
`group_commit`	10 KB	10	3707	66,697	8.56
`group_commit`	10 KB	30	3385	73,042	9.38
`group_commit`	100 KB	10	473	522,725	67.11
`group_commit`	100 KB	30	390	633,972	81.39
`group_commit`	500 KB	10	323	765,477	98.28
`group_commit`	500 KB	30	309	800,158	102.56
`group_commit`	1 MB	10	304	813,319	104.24
`group_commit`	1 MB	30	286	864,507	110.88
`group_commit`	10 MB	10	290	852,583	109.28
`非group_commit`	1 MB	10	导入报错-235		
`非group_commit`	10 MB	10	519	476,395	61.12
`非group_commit`	10 MB	30	导入报错-235		

在上面的`group_commit`测试中，BE的CPU使用率在10-40%之间。

JDBC 场景

- **1 FE**

8核，16GB 内存，1 * 200GB SSD 云磁盘

- **1 BE**

16核，64GB 内存，1 * 2TB SSD 云磁盘

- **1 测试客户端**

16核，64GB 内存，1 * 100GB SSD 云磁盘

单个insert的行数	并发数	导入速率(行/秒)	导入吞吐(MB/秒)
100	20	106931	11.46

在上面的测试中，FE 的 CPU使用率在60-70%左右，BE 的 CPU使用率在10-20%左右。

Group Commit 限制条件

1、不支持部分 INSERT INTO values 写入

- 列更新写入、事务写入、指定 label、表没有开启 light schema change、values 中有表达式；

2、不支持部分 StreamLoad 或 HttpStream 写入

- 两阶段提交、指定 label、列更新写入、表没有开启 light schema change；

3、unique 模型

- 由于 group commit 不能保证提交顺序，用户可以配合 sequence 列使用来保证数据一致性；

4、支持了一定程度的 max_filter_ratio 语义

- 当导入的总行数不高于 group_commit_memory_rows_for_max_filter_ratio (配置在 be.conf 中，默认为 10000行)，max_filter_ratio 工作；

5、async 模式下 WAL 的限制

- 目前 WAL 文件只存储在一个 BE 上，如果这个 BE 磁盘损坏或文件误删等，可能导入丢失部分数据；
- 当下线 BE 节点时，请使用 DECOMMISSION 命令，安全下线节点。

欢迎加入 Apache Doris

加入社区用户微信群

扫码添加 Doris 小助手，备注“加群”

提问赢好礼

提问被选中回复的小伙伴，请添加小助手微信领取 Doris 精美周边，数量有限，先到先得 ~

Doris 问答论坛

地址：ask.selectdb.com

Thanks !