高性能Flink SQL优化技巧

本文为您介绍提升性能的Flink SQL推荐写法、配置及函数。

Group Aggregate优化技巧

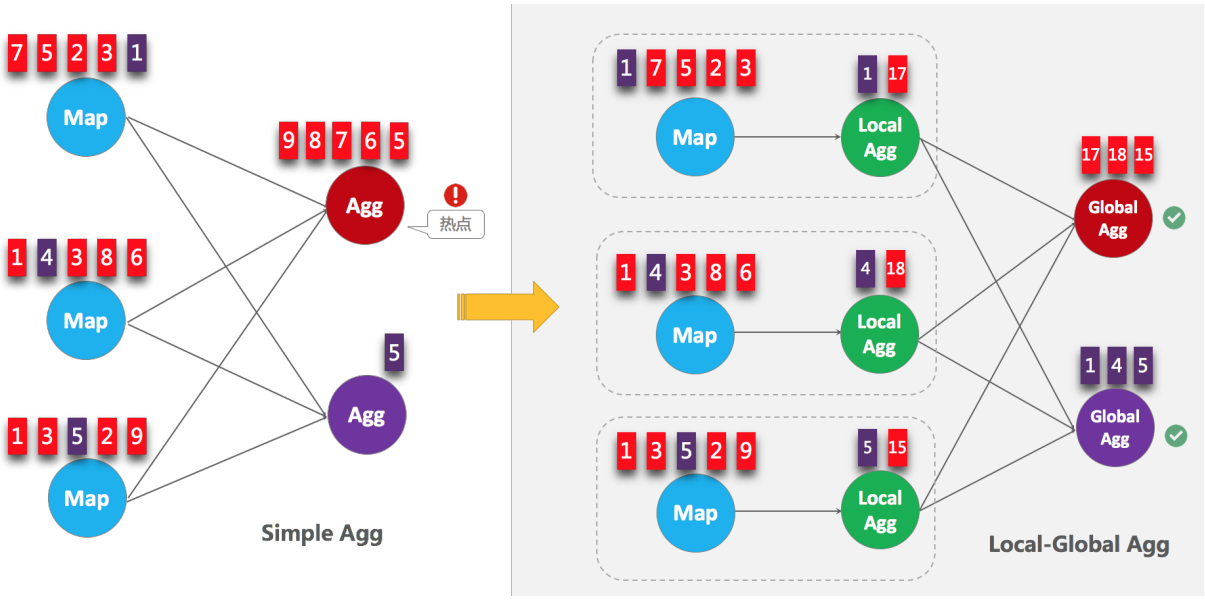
- 开启MicroBatch或MiniBatch（提升吞吐）
  MicroBatch和MiniBatch都是微批处理，只是微批的触发机制略有不同。原理同样是缓存一定的数据后再触发处理，以减少对State的访问，从而提升吞吐并减少数据的输出量。
  MiniBatch主要依靠在每个Task上注册的Timer线程来触发微批，需要消耗一定的线程调度性能。MicroBatch是MiniBatch的升级版，主要基于事件消息来触发微批，事件消息会按您指定的时间间隔在源头插入。MicroBatch在元素序列化效率、反压表现、吞吐和延迟性能上都要优于MiniBatch。
  - 适用场景
    微批处理通过增加延迟换取高吞吐，如果您有超低延迟的要求，不建议开启微批处理。通常对于聚合的场景，微批处理可以显著的提升系统性能，建议开启。

    > **说明** MicroBatch模式也能解决两级聚合数据抖动问题。

  - 开启方式
    MicroBatch和MiniBatch默认关闭，开启方式如下。

    ```
    # 3.2及以上版本开启Window miniBatch方法（3.2及以上版本默认不开启Window miniBatch）。
    sql.exec.mini-batch.window.enabled=true
    # 批量输出的间隔时间，在使用microBatch策略时，需要增加该配置，且建议和blink.miniBatch.a
    blink.microBatch.allowLatencyMs=5000
    # 在使用microBatch时，需要保留以下两个miniBatch配置。
    blink.miniBatch.allowLatencyMs=5000
    # 防止OOM设置每个批次最多缓存数据的条数。
    blink.miniBatch.size=20000
    ```

- 开启LocalGlobal（解决常见数据热点问题）
  LocalGlobal优化将原先的Aggregate分成Local+Global两阶段聚合，即MapReduce模型中的Combine+Reduce处理模式。第一阶段在上游节点本地攒一批数据进行聚合（localAgg），并输出这次微批的增量值（Accumulator）。第二阶段再将收到的Accumulator合并（Merge），得到最终的结果（GlobalAgg）。
  LocalGlobal本质上能够靠LocalAgg的聚合筛除部分倾斜数据，从而降低GlobalAgg的热点，提升性能。您可以结合下图理解LocalGlobal如何解决数据倾斜的问题。

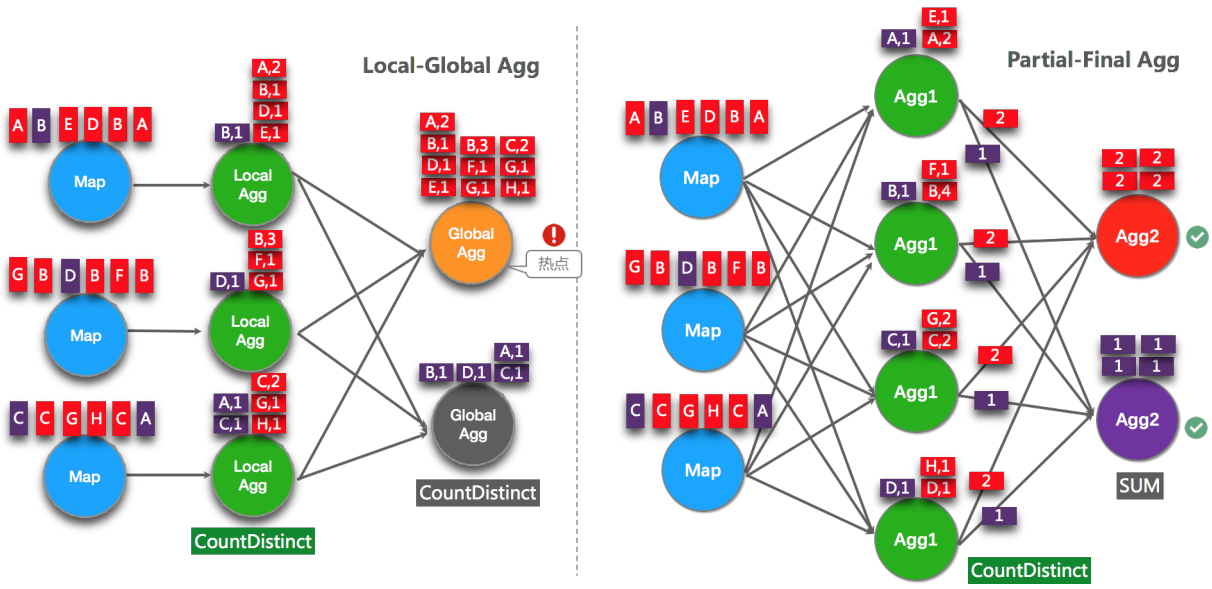


- 适用场景
  LocalGlobal适用于提升如SUM、COUNT、MAX、MIN和AVG等普通聚合的性能，以及解决这些场景下的数据热点问题。

  > **说明** 开启LocalGlobal需要UDAF实现`Merge`方法。

- 开启方式
  实时计算2.0版本开始，LocalGlobal是默认开启的，参数是**`blink.localAgg.enabled=true`**，但是需要在**`microbatch`**或**`minibatch`**开启的前提下才能生效。
- 判断是否生效
  观察最终生成的拓扑图的节点名字中是否包含**GlobalGroupAggregate**或**LocalGroupAggregate**。

- 开启PartialFinal（解决COUNT DISTINCT热点问题）
  LocalGlobal优化针对普通聚合（例如SUM、COUNT、MAX、MIN和AVG）有较好的效果，对于COUNT DISTINCT收效不明显，因为COUNT DISTINCT在Local聚合时，对于DISTINCT KEY的去重率不高，导致在Global节点仍然存在热点。
  之前，为了解决COUNT DISTINCT的热点问题，通常需要手动改写为两层聚合（增加按Distinct Key取模的打散层）。自`2.2.0`版本开始，实时计算提供了COUNT DISTINCT自动打散，即PartialFinal优化，您无需自行改写为两层聚合。PartialFinal和LocalGlobal的原理对比参见下图。



- 适用场景
  使用COUNT DISTINCT，但无法满足聚合节点性能要求。

> **说明**
> - 不能在包含UDAF的Flink SQL中使用PartialFinal优化方法。
> - 数据量不大的情况下，不建议使用PartialFinal优化方法。PartialFinal优化会自动打散成两层聚合，引入额外的网络Shuffle，在数据量不大的情况下，浪费资源。

  - 开启方式
    默认不开启，使用参数显式开启 `blink.partialAgg.enabled=true` 。
  - 判断是否生效
    观察最终生成的拓扑图的节点名中是否包含**Expand**节点，或者原来一层的聚合变成了两层的聚合。

- 改写为AGG WITH FILTER语法（提升大量COUNT DISTINCT场景性能）

> **说明** 仅实时计算2.2.2及以上版本支持AGG WITH FILTER语法。

统计作业需要计算各种维度的UV，例如全网UV、来自手机客户端的UV、来自PC的UV等等。建议使用标准的AGG WITH FILTER语法来代替CASE WHEN实现多维度统计的功能。实时计算目前的SQL优化器能分析出Filter参数，从而同一个字段上计算不同条件下的COUNT DISTINCT能共享State，减少对State的读写操作。性能测试中，使用AGG WITH FILTER语法来代替CASE WHEN能够使性能提升1倍。

  - 适用场景
    建议您将AGG WITH CASE WHEN的语法都替换成AGG WITH FILTER的语法，尤其是对同一个字段上计算不同条件下的COUNT DISTINCT结果，性能提升很大。
  - 原始写法

```
COUNT(distinct visitor_id) as UV1 , COUNT(distinct case when is_wireless='y' t
```

  - 优化写法

```
COUNT(distinct visitor_id) as UV1 , COUNT(distinct visitor_id) filter (where
```

TopN优化技巧

- TopN算法
  当TopN的输入是非更新流（例如Source），TopN只有一种算法AppendRank。当TopN的输入是更新流时（例如经过了AGG/JOIN计算），TopN有3种算法，性能从高到低分别是：UpdateFastRank 、 UnaryUpdateRank和RetractRank。算法名字会显示在拓扑图的节点名字上。
  - UpdateFastRank： 最优算法。
    需要具备2个条件：
    - 输入流有PK（Primary Key）信息，例如ORDER BY AVG。
    - 排序字段的更新是单调的，且单调方向与排序方向相反。例如，ORDER BY COUNT/COUNT_DISTINCT/SUM（正数）DESC（仅实时计算2.2.2及以上版本支持）。
      如果您要获取到优化Plan，则您需要在使用ORDER BY SUM DESC时，添加SUM为正数的过滤条件，确保**total_fee**为正数。

```
insert
  into print_test
SELECT
  cate_id,
  seller_id,
  stat_date,
  pay_ord_amt  --不输出rownum字段，能减小结果表的输出量。
FROM (
    SELECT
```

```
      *,
      ROW_NUMBER () OVER (
        PARTITION BY cate_id,
        stat_date  --注意要有时间字段，否则state过期会导致数据错乱。
        ORDER BY pay_ord_amt DESC
      ) as rownum  --根据上游sum结果排序。
    FROM (
        SELECT
          cate_id,
          seller_id,
          stat_date,
          --重点。声明Sum的参数都是正数，所以Sum的结果是单调递增的，因此TopN能使用优化
          sum (total_fee) filter (
            where
              total_fee >= 0
          ) as pay_ord_amt
        FROM
          random_test
        WHERE
          total_fee >= 0
        GROUP BY cate_name,
          seller_id,
          stat_date
      ) a
    WHERE
      rownum <= 100
  );
```

- UnaryUpdateRank：仅次于UpdateFastRank的算法。需要具备1个条件：输入流中存在PK信息。
- RetractRank：普通算法，性能最差，不建议在生产环境使用该算法。请检查输入流是否存在PK信息，如果存在，则可进行UnaryUpdateRank或UpdateFastRank优化。

- TopN优化方法
  - 无排名优化
    TopN的输出结果无需要显示rownum值，仅需在最终前端显式时进行1次排序，极大地减少输入结果表的数据量。无排名优化方法详情请参见TopN语句。
  - 增加TopN的Cache大小
    TopN为了提升性能有一个State Cache层，Cache层能提升对State的访问效率。TopN的Cache命中率的计算公式为。

    ```
    cache_hit = cache_size*parallelism/top_n/partition_key_num
    ```

    例如，Top100配置缓存10000条，并发50，当您的PatitionBy的key维度较大时，例如10万级别时，Cache命中率只有10000*50/100/100000=5%，命中率会很低，导致大量的请求都会击中State（磁盘），性能会大幅下降。因此当PartitionKey维度特别大时，可以适当加大TopN的CacheS ize，相对应的也建议适当加大TopN节点的Heap Memory（请参见手动配置调优）。

    ```
    ##默认10000条，调整TopN cahce到20万，那么理论命中率能达200000*50/100/100000 = 100%。
    blink.topn.cache.size=200000
    ```

  - PartitionBy的字段中要有时间类字段
    例如每天的排名，要带上Day字段。否则TopN的结果到最后会由于State ttl有错乱。

高效去重方案

> **说明** 仅实时计算3.2.1及以上版本支持高效去重方案。

实时计算的源数据在部分场景中存在重复数据，去重成为了用户经常反馈的需求。实时计算有保留第一条（Deduplicate Keep FirstRow）和保留最后一条（Deduplicate Keep LastRow）2种去重方案。

- 语法
  由于SQL上没有直接支持去重的语法，还要灵活的保留第一条或保留最后一条。因此我们使用了SQL的ROW_NUMBER OVER WINDOW功能来实现去重语法。去重本质上是一种特殊的TopN。

```
SELECT *
FROM (
   SELECT *,
    ROW_NUMBER() OVER ([PARTITION BY col1[, col2..]
     ORDER BY timeAttributeCol [asc|desc]) AS rownum
   FROM table_name)
WHERE rownum = 1
```

参数	说明
ROW_NUMBER()	计算行号的OVER窗口函数。行号从1开始计算。
PARTITION BY col1[, col2..]	可选。指定分区的列，即去重的KEYS。
ORDER BY timeAttributeCol [asc|desc])	指定排序的列，必须是一个时间属性的字段（即Proctim或Rowtime）。可以指定顺序（Keep FirstRow）或者倒序 （Keep LastRow）。
rownum	仅支持`rownum=1`或`rownum<=1`。

如上语法所示，去重需要两层Query：

i. 使用`ROW_NUMBER()`窗口函数来对数据根据时间属性列进行排序并标上排名。

> **说明**
> - 当排序字段是Proctime列时，Flink就会按照系统时间去重，其每次运行的结果是不确定的。
> - 当排序字段是Rowtime列时，Flink就会按照业务时间去重，其每次运行的结果是确定的。

ii. 对排名进行过滤，只取第一条，达到了去重的目的。

> **说明** 排序方向可以是按照时间列的顺序，也可以是倒序：
> - Deduplicate Keep FirstRow：顺序并取第一条行数据。
> - Deduplicate Keep LastRow：倒序并取第一条行数据。

- Deduplicate Keep FirstRow
  保留首行的去重策略：保留KEY下第一条出现的数据，之后出现该KEY下的数据会被丢弃掉。因为STATE中只存储了KEY数据，所以性能较优，示例如下。

```
SELECT *
FROM (
  SELECT *,
```

```
    ROW_NUMBER() OVER (PARTITION BY b ORDER BY proctime) as rowNum
  FROM T
)
WHERE rowNum = 1
```

> **说明** 以上示例是将T表按照b字段进行去重，并按照系统时间保留第一条数据。Proctime在这里是源表T中的一个具有Processing Time属性的字段。如果您按照系统时间去重，也可以将Proctime字段简化`PROCTIME()`函数调用，可以省略Proctime字段的声明。

- Deduplicate Keep LastRow
  保留末行的去重策略：保留KEY下最后一条出现的数据。保留末行的去重策略性能略优于LAST_VALUE函数，示例如下。

```
SELECT *
FROM (
  SELECT *,
    ROW_NUMBER() OVER (PARTITION BY b, d ORDER BY rowtime DESC) as rowNum
  FROM T
)
WHERE rowNum = 1
```

> **说明** 以上示例是将T表按照b和d字段进行去重，并按照业务时间保留最后一条数据。Rowtime在这里是源表T中的一个具有Event Time属性的字段。

高效的内置函数

- 使用内置函数替换自定义函数
  实时计算的内置函数在持续的优化当中，请尽量使用内部函数替换自定义函数。实时计算2.0版本对内置函数主要进行了如下优化：
  - 优化数据序列化和反序列化的耗时。
  - 新增直接对字节单位进行操作的功能。
- KEY VALUE函数使用单字符的分隔符
  KEY VALUE 的签名： `KEYVALUE(content, keyValueSplit, keySplit, keyName)`，当keyValueSplit和KeySplit是单字符（例如，冒号（:）、逗号（,））时，系统会使用优化算法，在二进制数据上直接寻找所需的keyName 的值，而不会将整个content做切分。性能约提升30%。
- 多KEY VALUE场景使用MULTI_KEYVALUE

> **说明** 仅实时计算`2.2.2`及以上版本支持MULTI_KEYVALUE。

在Query中对同一个Content进行大量KEY VALUE的操作，会对性能产生很大影响。例如Content中包含10个Key-Value对，如果您希望把10个Value的值都取出来作为字段，您就需要写10个KEY VALUE函数，则系统就会对Content进行10次解析，导致性能降低。
在这种情况下，建议您使用MULTI_KEYVALUE表值函数，该函数可以对Content只进行一次Split解析，性能约能提升50%~100%。

- LIKE操作注意事项
  - 如果需要进行StartWith操作，使用`LIKE 'xxx%'`。
  - 如果需要进行EndWith操作，使用`LIKE '%xxx'`。
  - 如果需要进行Contains操作，使用`LIKE '%xxx%'`。

- 如果需要进行Equals操作，使用`LIKE 'xxx'`，等价于`str = 'xxx'`。
- 如果需要匹配 `_` 字符，请注意要完成转义`LIKE '%seller/id%' ESCAPE '/'`。_在SQL中属于单字符通配符，能匹配任何字符。如果声明为 `LIKE '%seller_id%'`，则不单会匹配`seller_id`还会匹配`seller#id`、`sellerxid`或`seller1id` 等，导致结果错误。

- 慎用正则函数（REGEXP）
  正则表达式是非常耗时的操作，对比加减乘除通常有百倍的性能开销，而且正则表达式在某些极端情况下可能会进入无限循环，导致作业阻塞。建议使用LIKE。正则函数包括：
  - REGEXP
  - REGEXP_EXTRACT
  - REGEXP_REPLACE

网络传输的优化

目前常见的Partitioner策略包括：

- KeyGroup/Hash：根据指定的Key分配。
- Rebalance：轮询分配给各个Channel。
- Dynamic-Rebalance：根据下游负载情况动态选择分配给负载较低的Channel。
- Forward：未Chain一起时，同Rebalance。Chain一起时是一对一分配。
- Rescale：上游与下游一对多或多对一。

- 使用Dynamic-Rebalance替代Rebalance
  Dynamic-Rebalance可以根据当前各Subpartition中堆积的Buffer的数量，选择负载较轻的Subpartition进行写入，从而实现动态的负载均衡。相比于静态的Rebalance策略，在下游各任务计算能力不均衡时，可以使各任务相对负载更加均衡，从而提高整个作业的性能。例如，在使用Rebalance时，发现下游各个并发负载不均衡时，可以考虑使用Dynamic-Rebalance。参数： `task.dynamic.rebalance.enabled=true`， 默认关闭。
- 使用Rescale替代Rebalance

  > **说明** 仅实时计算2.2.2及以上版本支持Rescale。

  例如，上游是5个并发，下游是10个并发。当使用Rebalance时，上游每个并发会轮询发给下游10个并发。当使用Rescale时，上游每个并发只需轮询发给下游2个并发。因为Channel个数变少了，Subpartition的Buffer填充速度能变快，能提高网络效率。当上游的数据比较均匀时，且上下游的并发数成比例时，可以使用Rescale替换Rebalance。参数： `enable.rescale.shuffling=true`，默认关闭。

推荐的优化配置方案

综上所述，作业建议使用如下的推荐配置。

```
# EXACTLY_ONCE语义。
blink.checkpoint.mode=EXACTLY_ONCE
# checkpoint间隔时间，单位毫秒。
blink.checkpoint.interval.ms=180000
blink.checkpoint.timeout.ms=600000
# 2.x使用niagara作为statebackend，以及设定state数据生命周期，单位毫秒。
state.backend.type=niagara
state.backend.niagara.ttl.ms=129600000
# 2.x开启5秒的microbatch。
blink.microBatch.allowLatencyMs=5000
# 整个Job允许的延迟。
blink.miniBatch.allowLatencyMs=5000
```

```
# 单个batch的size。
blink.miniBatch.size=20000
# local 优化，2.x默认已经开启，1.6.4需手动开启。
blink.localAgg.enabled=true
# 2.x开启PartialFina优化，解决COUNT DISTINCT热点。
blink.partialAgg.enabled=true
# union all优化。
blink.forbid.unionall.as.breakpoint.in.subsection.optimization=true
# object reuse优化，默认已开启。
#blink.object.reuse=true
# GC优化（SLS做源表不能设置该参数）。
blink.job.option=-yD heartbeat.timeout=180000 -yD env.java.opts='-verbose:gc -XX:N
# 时区设置。
blink.job.timeZone=Asia/Shanghai
```