Spark Web UI 排查问题思路分享

定位问题：通过一个案例，介绍 Spark Web UI ，并快速定位任务报错或性能瓶颈的具体环节。

成本治理：针对高内存消耗、长耗时任务提供排查思路。

解决方案：输出具体的调优手段，降低资源浪费，保障任务稳定性。

以任务 **3361191** 为例子介绍 Spark Web UI

【Executor内存】

出现以下报错信息，可以定位是内存不足的问题：

dw_pfc_dly_wave_chnl_choose_record_di_pre

任务：新中央 **3325872**

现象：

- 任务报错 OOM（内存不足），Spark Executor 内存被打满
- 临时将 Executor 自定义内存调至 **64G** 后勉强跑通
- 实际 Executor 内存使用高达 **58G**，资源消耗巨大，带来显著成本压力

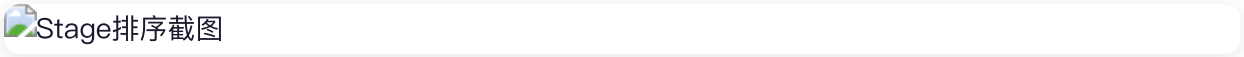
打开semr日志，观察本次任务实例的配置内存与实际用到的峰值内存

实际使用内存58g，非常大，需要通过Spark Web UI定位资源消耗大的地方。目前Spark Web UI没有地方可以直接看出各个环节消耗的内存大小。

在 Spark Web UI 中，我们无法直接看到"哪行代码消耗了多少内存"，但可以通过 **Stage 详情** 推导资源消耗点。

排查路径：进入 Spark Web UI → 点击 `Stage` 标签页 → 按耗时或外部输入输出数据量或shuffle数据量做倒序排列

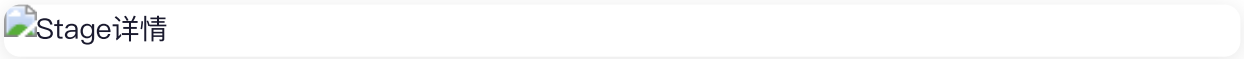
排查经验思路：点击stage进入查看各个stage的情况，排序stage，查看task数较多的且处理数据量较大的，就是消耗内存较大的stage。



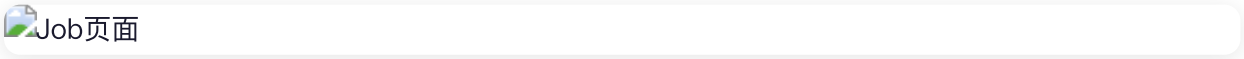
关键指标解读

指标	含义	排查要点
Tasks: Succeeded / Total	任务进度与并行度	**Total** 代表分区数（并行度）。若 **Succeeded** 长时间卡顿在某个数字（例如 199/200），高度怀疑**数据倾斜**。
Input	外部存储读取量	对应 SQL 的 SCAN 操作。若某 Stage Input 远超预期，通常意味着**全表扫描**。
Output	写入外部存储量	对应 INSERT 或 CREATE TABLE 的最终写出量。
Shuffle Read	Shuffle 拉取量	下游 Stage 从上游拉取的数据量，用于 Join/Aggregation。
Shuffle Write	Shuffle 写出量	上游 Stage 处理后写盘的数据量。

点击对应 stage 的 description，跳转到 stage detail 明细页面：



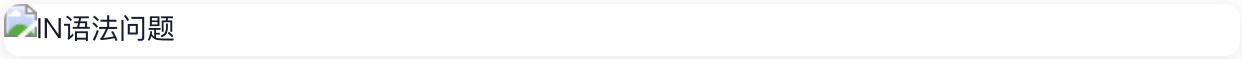
再点击关联到对应的job页面：



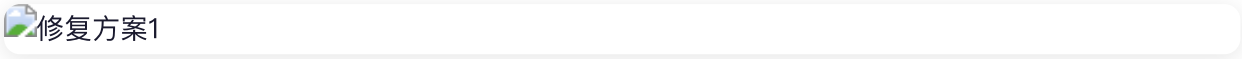
最后点击SQL Query跳转到sql页面：

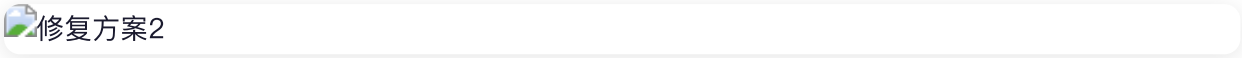
拿到上面排查出来的stage8，通过sql分析查看，有一处增量分区表做了全表扫描的操作，直接读取了全表近150亿数据，sql里的分区裁剪没有生效

分析sql代码逻辑，发现sparksql的in语法会使得分区裁剪不生效

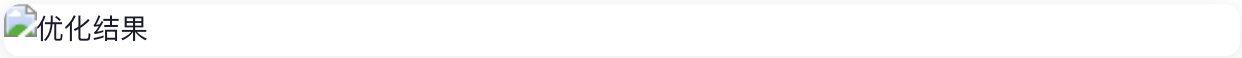


修复方案：利用变量语法，查出最小有修改的分区，显式声明分区裁剪





查看最新webui，发现裁剪成功，自定义内存可以由原来的64g降低成16g



排查总结

1. **观察 SEMR 日志**：确认实际内存峰值高达 58GB。
2. **定位 Stage**：在 Web UI 中发现 **Stage 8** 的 **Input** 高达 **1436.2 GiB**。
3. **定位SQL**：该 Input 对应一张增量分区表。虽然 SQL 中有过滤逻辑，但 Input 大小显示其进行了全表扫描（读取了近 150 亿条数据）。
4. **根因锁定**：Spark SQL 的 `IN` 子句语法在特定场景下导致**分区裁剪失效**。
5. **优化方案**：利用变量语法，显式查出最小修改分区，强制声明分区裁剪范围，避免全表扫描。
6. **优化效果**：分区裁剪生效，Input 数据量大幅下降。Executor 内存由 **64GB** 降至 **16GB**，任务运行平稳。

【并行度】

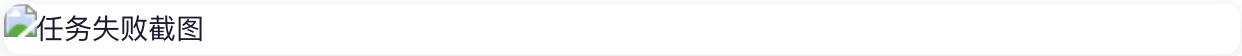
并行度如果太小，导致每个task分到的数据量太大，也会出现内存溢出。

当出现任务运行时长非常长的现象，可初步定位是数据倾斜或者并行度太小的问题，需要具体分析。

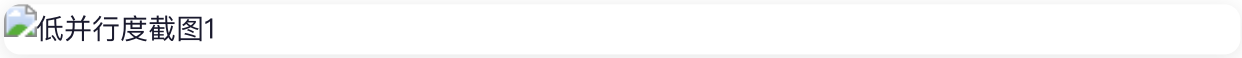
dw_pcm_qac_qmc_size_rule_v2_match_res_di

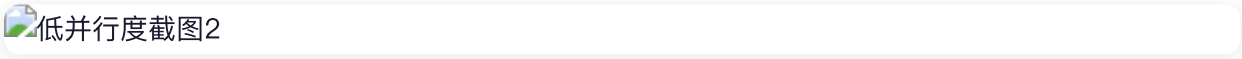
任务：新中央 3321521

现象：任务运行极慢，经常自动失败，且并行度异常低。


任务失败截图

排查 Spark Web UI 后发现，某个stage运行时间非常长，且观察到并行度很低，只有22，继续往下找出并行度这么小的原因


低并行度截图1


低并行度截图2

原因排查：

- AQE 检测到读取源数据较小，只有一万多条，智能地合并了分区。
- AQE 未能预判 `explode` 操作会产生海量数据，导致用极少的 Task 处理千万级别数据，引发内存溢出或超时。

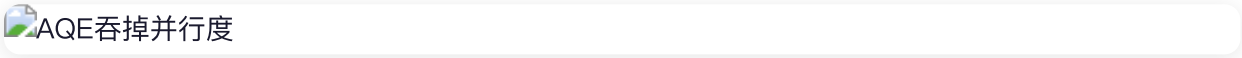
代码：炸裂取数

```
--部位与规则信息
create temporary table dw.dw_pcm_qac_qmc_size_rule_v2_match_tmp03 as
select
    a.biz_data.skc  as skc,
    a.biz_data.trace_id as trace_id,
    a.biz_data.area_match_multi_rule_log_list as
area_match_multi_rule_log_list,
    t.area_rule_json.area_id as area_id,
    t.area_rule_json.is_match_white_list as is_match_white_list,
    t.area_rule_json.rule_info_list as rule_info_list,
    t.area_rule_json.basic_size as basic_size,
    t2.rule_info.rule_id as rule_id,
    t2.rule_info.version as version,
```

```
    t2.rule_info.match_rule_type as match_rule_type
from(
  select * from dw.dw_pcm_qac_qmc_size_rule_v2_match_tmp00
) a
lateral view outer explode(a.biz_data.area_match_multi_rule_log_list) t
as area_rule_json
lateral view outer explode(t.area_rule_json.rule_info_list) t2 as
rule_info
```

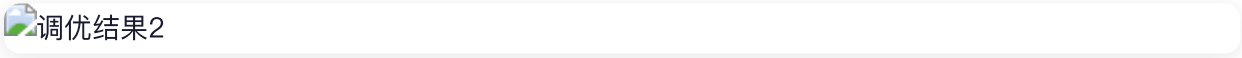
第一步思想，通过加大并行度的方案解决，尝试直接调参 `spark.sql.shuffle.partitions`，后续发现该参数会**影响全局任务**的shuffle并行度，目前只有这一部分需要加大并行度，希望不影响其他stage的并行度，因此利用sparksql的hint语法，`/* REPARTITION(1200) */`，针对某一段加大并行度

尝试后结果发现另一个问题，spark的 **aqe将调参加大的并行度吞掉**了，因为aqe发现读取的数据量很少，默认减少了处理的并行度，调的参数不起作用



希望不关闭全局aqe，其他stage也能享受到aqe的优化，而只针对有问题的这部分代码关闭aqe。

调优结果：



注意加大并行度处理后，再开一个REPARTITION合并小文件问题，REPARTITION的分区数要按照实际情况调整

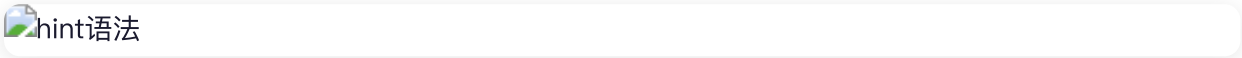
最终优化代码（仅供参考）：

```
--部位与规则信息
set spark.sql.adaptive.coalescePartitions.enabled = false;
create temporary table dw.dw_pcm_qac_qmc_size_rule_v2_match_tmp03 as
select
    a.biz_data.skc  as skc,
    a.biz_data.trace_id as trace_id,
    ...
from(
  select
  /*+ REPARTITION(1200) */
    *
  from dw.dw_pcm_qac_qmc_size_rule_v2_match_tmp00
) a
```

```
lateral view outer explode(a.biz_data.area_match_multi_rule_log_list) t
as area_rule_json
lateral view outer explode(t.area_rule_json.rule_info_list) t2 as
rule_info
;
set spark.sql.adaptive.coalescePartitions.enabled = true;
```

调优策略：局部关闭 AQE + 显式重分区 + 处理小文件 + 恢复 AQE。

遇到任务报错内存溢出或者无法广播大表时，也可以借助sparksql hint语法，强制使用shuffle_merge，不要强行全局关闭广播 `spark.sql.autoBroadcastJoinThreshold = false`



总结调整并行度的方法

场景一（粗粒度调整）：没有精准定位哪一部分代码跑不过去，只是任务报错OOM，将executor内存调大后能跑过。现在希望通过调大并发同时减少内存来优化。

AQE开启的情况：

```
spark.sql.adaptive.coalescePartitions.initialPartitionNum = 3000
spark.sql.adaptive.coalescePartitions.minPartitionNum = 3000
spark.sql.adaptive.coalescePartitions.parallelismFirst = True
```

AQE关闭的情况：

```
spark.sql.adaptive.coalescePartitions.enabled = false
spark.sql.shuffle.partitions = 3000
```

以上操作保持高并发度，对大表与大表join需要调高分区的场景有效，适用于**逻辑简单**任务，属于全局参数，会影响所有shuffle操作

场景二（精细化调整）：已经通过 Web UI 排查出某一段sql并行度不足，需要针对某一段sql调整并行度，不影响其他的部分代码享受aqe的分区优化

方案1：局部代码手动关闭aqe，利用spark.sql.shuffle.partitions设置整一段的并行度，再重新开启aqe

```
...
SET spark.sql.adaptive.coalescePartitions.enabled = false; --关闭aqe
SET spark.sql.shuffle.partitions = 1800; --以下代码涉及shuffle操作，分区数改为
1800
```

```
----需要调整并行度的部分
select xxx
from (select xxx from table_a) a
  left join (select xxx from table_b) b on ...
  left join (select xxx from table_c) c on ...
----启动aqe
SET spark.sql.adaptive.coalescePartitions.enabled = true;
...
```

方案二：利用sparksql的hint语法，更细致的控制某一段代码的并行度

```
SET spark.sql.adaptive.coalescePartitions.enabled = false; --关闭aqe
select xxx
from (
    select /*+ REPARTITION(3000,skc) */ xxx from table_a_b
) a
left join (
    select /*+ REPARTITION(3000,goods_sn) */ xxx from table_c
) c on a.skc = c.goods_sn
SET spark.sql.adaptive.coalescePartitions.enabled = true;
```

小文件问题处理：调整完并行度之后，由于关闭了aqe，会有小文件问题。解决方案，在走完shuffle之后，再次利用REPARTITION合并小文件：

```
SET spark.sql.adaptive.coalescePartitions.enabled = false;
insert overwrite table xxx
select
    /*+ REPARTITION(200,skc) */    --解决小文件问题
    xxx
from (
    select /*+ REPARTITION(3000,skc) */ xxx from table_a_b
) a
left join (
    select /*+ REPARTITION(3000,goods_sn) */ xxx from table_c
) c on a.skc = c.goods_sn
SET spark.sql.adaptive.coalescePartitions.enabled = true;
```

【数据倾斜】

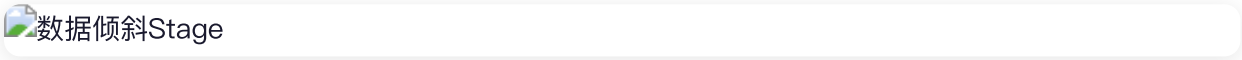
介绍如何通过 Web UI 的 Task 耗时分布图识别倾斜，以及如何定位是具体某一段sql导致的倾斜，利用 Salt Key 打散的方式解决。

dim_qmc_skc_material_info_d

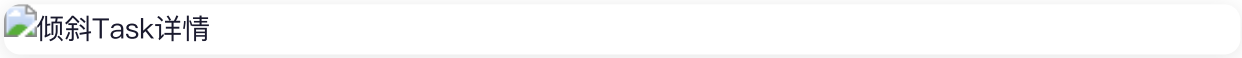
任务：新中央 3236722

现象：拖慢整体进度，99% 的 Task 都在 1 分钟内跑完，剩下的 1% 可能要跑 1 小时，出现长尾效应，个别 Task 内存溢出，导致任务反复重试甚至失败，需要调大executor内存才能跑过。

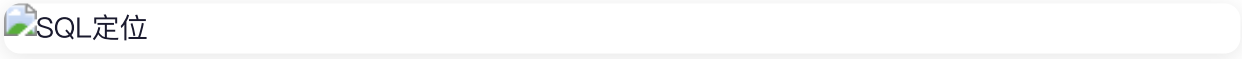
进入 Web UI 查看耗时最长的stage：



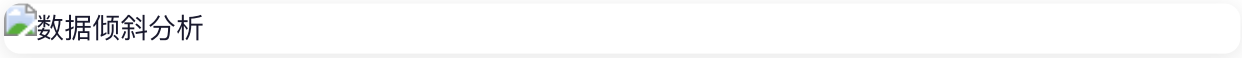
查看有一个task数据量极大，且非常耗时：



拿到stage id = 136，定位具体sql逻辑：



回到一站式任务，查看识别出来第321行的关联导致数据倾斜，对主表做数据探查，对 Join Key 进行 `GROUP BY count(1)` 统计，发现数据分布极不均匀。(Platform、Brazil、null)数据量比较大的值，对倾斜key拉出来单独处理。



治理结果：平均运行时长从 **1 小时** 缩短至 **10 分钟**，移除了自定义的大内存配置，显著降低了计算资源消耗。