# spark任务优化思路

# Spark 任务优化思路

> **引言**：大数据开发任务从 Hive 集群迁移至 Spark on K8s 集群后，由于两者引擎特性差异，直接迁移并不能充分发挥 Spark 的计算优势。优化应以 **SQL 代码优化为主、资源调参为辅**，切忌在 SQL 未优化的情况下盲目提高资源配置，徒增成本。

---

## 一、代码优化

Spark 的查询优化器 **Catalyst** 会自动执行列裁剪、谓词下推、常量折叠等优化，但良好的 SQL 写法能让 Catalyst 更好地发挥作用。

### 1.1 减少数据扫描量

避免 `SELECT *`，只选择必要的列。

> **重点**：在普通查询中，Catalyst 会自动进行列裁剪。但如果使用 `CACHE TABLE` 将视图物化到内存/磁盘，Catalyst **不会**对缓存内容做列裁剪，`SELECT *` 会导致所有列被缓存，浪费内存和磁盘。

```
-- 反例：缓存视图中使用 SELECT *，所有列都会被物化
CACHE TABLE temp_view1 AS
SELECT a.*, b.col1, b.col2
FROM tableA a
LEFT JOIN (SELECT * FROM tableB WHERE ...) b ON a.id = b.id;

-- 正例：只缓存需要的列
CACHE TABLE temp_view1 AS
SELECT a.col1, a.col2, a.coln, b.col1, b.col2
FROM tableA a
LEFT JOIN tableB b ON a.id = b.id;
```

## 1.2 合理使用过滤条件（ON vs WHERE）

`LEFT JOIN` 中，`ON` 和 `WHERE` 的过滤语义不同，需根据业务场景区分：

- **ON 中的条件**：仅影响关联匹配过程，不匹配时右表返回 NULL，左表数据保留
- **WHERE 中的条件**：在关联完成后过滤，对右表字段做非 NULL 过滤时，等价于将 `LEFT JOIN` 转换为 `INNER JOIN`

```
-- 优化前：LEFT JOIN + WHERE 过滤 b 表字段，语义上已等同于 INNER JOIN，写法易产生歧义
SELECT a.coln, b.coln
FROM tableA a
LEFT JOIN tableB b ON a.id = b.id
WHERE a.coln = '...' AND b.coln = '...';

-- 优化后：明确使用 INNER JOIN，语义清晰，Catalyst 可更直接地进行谓词下推和 Join 重排序
SELECT a.coln, b.coln
FROM tableA a
JOIN tableB b ON a.id = b.id AND b.coln = '...'
WHERE a.coln = '...';
```

> **原则**：如果对右表字段做了非 NULL 的 WHERE 过滤，说明不需要保留未匹配的左表行，应改用 `INNER JOIN`。虽然 Catalyst 在大多数情况下能自动识别这种模式并将 `LEFT JOIN` 转换为 `INNER JOIN`（Outer Join 消除），但明确写出 `INNER JOIN` 语义更清晰，也能避免 Catalyst 在复杂查询中遗漏优化的边界情况。

## 1.3 优化 JOIN 操作

### Broadcast Join（适用于大小表关联）

将小表广播到每个 Executor，执行 Map 端 Join，**无需 Shuffle 大表**，性能最优。

```
SELECT /*+ BROADCASTJOIN(b) */
    a.col1, a.col2, b.col2
FROM tableA a
LEFT JOIN tableB b ON a.col1 = b.col1;
```

- **适用场景**：小表数据量较小（通常 < 几十 MB \~ 几百 MB）
- **注意事项**：小表过大会占用大量 Executor 内存，甚至导致 OOM；两表都很小时，广播开销可能超过收益

### Sort Merge Join（适用于大表关联）

两张大表分别按 Join Key 排序后合并，不需要将整张表加载到内存。

```
SELECT /*+ SHUFFLE_MERGE(b) */
    a.col1, a.col2, b.col2
FROM tableA a
LEFT JOIN tableB b ON a.col1 = b.col1;
```

- **适用场景**：两张大表关联，或小表关联后会导致数据膨胀
- **注意事项**：两表都需要 Shuffle + 排序，网络和计算开销较大；数据分布不均时可能产生倾斜

### 通用原则

- 控制 JOIN 数量，减少多表关联的复杂度
- 为非等值关联条件（如 `CASE WHEN`）构建关联键字典，转化为等值 JOIN

## 1.4 临时视图与缓存策略

### 临时视图（Temp View）

临时视图只存储查询定义（逻辑计划），**不物化数据**。每次引用视图时，Spark 都会重新执行底层查询（懒加载）。

```
CREATE TEMP VIEW temp_viewA AS SELECT ...;
```

- 适用于：简化复杂查询、提高代码可读性
- 注意：多次引用同一视图时，会重复执行查询逻辑，效率低下

### 缓存视图（Cache Table）

当视图被多次引用时，应将其物化到内存（或溢写到磁盘），避免重复计算。

```
-- 方式一：先创建视图，再触发缓存
CREATE TEMP VIEW temp_viewA AS SELECT ...;
CACHE TABLE temp_viewA;

-- 方式二：直接缓存查询结果（效果与方式一相同）
CACHE TABLE temp_viewA AS SELECT ...;

-- 指定存储级别（序列化后存储到内存+磁盘，减少内存占用）
CACHE TABLE temp_viewA OPTIONS('storageLevel' 'MEMORY_AND_DISK_SER') AS SELECT ...;
```

### 缓存的内存风险

当缓存数据过多导致存储内存不足时，Spark 会按 **LRU（最近最少使用）策略驱逐**部分缓存分区（非无差别清除）。被驱逐的分区在下次访问时需要重新计算，如果此时内存仍然不足，会陷入"驱逐→重算→再驱逐"的恶性循环，任务执行时间大幅增长甚至无法完成。

> **最佳实践**：尽快对大数据量的缓存视图做聚合/裁剪后生成新的小视图，然后及时释放大视图。

```
-- 对大视图聚合后缓存为小视图
CACHE TABLE temp_viewB AS
SELECT col1, COUNT(col1) AS col1_cnt FROM temp_viewA GROUP BY col1;

-- 及时释放不再使用的大视图
UNCACHE TABLE temp_viewA;
```

### 缓存视图 vs Hive 中间表

| 特性 | Spark 缓存视图 | Hive 中间表 |
| --- | --- | --- |
| 存储位置 | Executor 本地内存/磁盘 | 分布式文件系统（HDFS/S3） |
| 文件格式 | Spark 内部格式 | 标准格式（ORC/Parquet） |
| 生命周期 | 随 SparkSession 结束而销毁 | 持久化存储 |
| 访问范围 | 当前 Session | 跨 Session / 跨应用 |
| 读取性能 | 本地读取，速度快 | 网络读取，相对较慢 |
| 管理方式 | 自动管理 | 需手动管理（建表/删表） |
| 故障恢复 | 可根据 Lineage 自动重建 | 需手动重跑 |

**结论**：同一会话内多次使用的中间数据 → 用缓存视图；跨会话或跨应用共享 → 用 Hive 中间表。

## 1.5 重新分区（Repartition）

通过 `REPARTITION` 将数据按指定字段重新分布到各分区，解决数据倾斜、提高下游关联效率。

```
CACHE TABLE temp_viewA_B AS
SELECT /*+ REPARTITION(5000, a.col1, b.col2) */
    a.col1, b.col2
FROM tableA a
JOIN tableB b ON a.col1 = b.col1;
```

- **原理**：按指定字段 Hash 分区，使相同 Key 的数据落在同一分区，后续按该字段 JOIN 或聚合时可减少甚至跳过 Shuffle
- **代价**：Repartition 本身会触发一次全量 Shuffle，需确保收益大于开销
- **适用场景**：缓存视图后续会被多次按同一字段 JOIN 或聚合

---

# 二、集群调参

> **原则**：平台默认配置可适用大部分场景，调参应保守渐进，切勿激进。

## 2.1 执行器（Executor）资源

| 参数 | 说明 | 示例值 |
| --- | --- | --- |
| spark.executor.instances | 固定 Executor 数量（Pod 数） | 200（默认） |
| spark.dynamicAllocation.initialExecutors | 动态分配的初始 Executor 数 | 10 |
| spark.dynamicAllocation.maxExecutors | 动态分配的最大 Executor 数 | 300 |
| spark.executor.cores | 每个 Executor 的 CPU 核心数 | 4（默认 3） |

> **注意**：`spark.executor.instances`（固定数量）和 `spark.dynamicAllocation.maxExecutors`（动态上限）二选一即可。开启动态分配后，Executor 数量由 Spark 按需伸缩；若未单独设置 `initialExecutors`，则 `spark.executor.instances` 的值会作为动态分配的初始 Executor 数。

**并行度公式**：

```
Stage 并发线程数 = min(该 Stage 的分区数, 存活 Executor 数 × 每个 Executor 的核心数)
```

## 2.2 内存配置

| 参数 | 说明 | 示例值 |
| --- | --- | --- |
| spark.executor.memory | 每个 Executor 的堆内存 | 16g（默认 10g） |
| spark.executor.memoryOverhead | 堆外内存（容器额外内存） | 16g |
| spark.memory.fraction | 执行+存储内存占堆内存的比例 | 0.5（Spark 默认 0.6） |
| spark.memory.storageFraction | 存储内存占（执行+存储内存）的比例 | 0.6（Spark 默认 0.5） |

**内存结构示意**（以 `spark.executor.memory=16g, fraction=0.5, storageFraction=0.6` 为例）：

```
堆内存 16g
├── 预留内存：300MB（固定）
├── 用户内存：(16g - 300MB) × (1 - 0.5) ≈ 7.85g
└── 统一内存：(16g - 300MB) × 0.5 ≈ 7.85g
    ├── 存储内存（初始）：7.85g × 0.6 ≈ 4.71g  ← 用于缓存
    └── 执行内存（初始）：7.85g × 0.4 ≈ 3.14g  ← 用于 Shuffle/排序/聚合
```

> **说明**：Spark 采用统一内存管理，执行内存和存储内存可以互相借用。当执行内存不足时可驱逐存储内存中的缓存；但执行内存一旦被占用，存储内存不能强行收回。因此 `storageFraction` 只是初始边界，不是硬限制。

**实际可用堆内存计算**（受 64MB 内存块对齐和 2% 安全缓冲影响）：

```
内存块数 = ceil(堆内存MB × fraction × 0.98 / 64)
实际堆内存 = round(内存块数 × 64 / 1024, 1) GB
```

## 2.3 Shuffle 与分区

| 参数 | 说明 | 示例值 |
| --- | --- | --- |
| spark.sql.shuffle.partitions | Shuffle 操作的初始分区数 | 200（默认） |

> **注意**：开启 AQE 后，此参数作为 Shuffle 分区的**初始上限**，AQE 会根据实际数据量自动合并小分区，无需精细手动调优。

## 2.4 AQE（自适应查询执行）

AQE 在运行时根据实际数据统计信息动态调整执行计划，包括自动合并小分区、优化 Join 策略、处理数据倾斜等，**Spark 3.2+ 默认开启**。

| 参数 | 说明 | 示例值 |
| --- | --- | --- |
| spark.sql.adaptive.enabled | AQE 总开关 | true |
| spark.sql.adaptive.coalescePartitions.enabled | 启用自动合并小分区 | true |
| spark.sql.adaptive.coalescePartitions.minPartitionNum | 合并后最少保留的分区数 | 200 |
| spark.sql.adaptive.coalescePartitions.minPartitionSize | 合并后单分区最小数据量 | 64MB |
| spark.sql.adaptive.coalescePartitions.initialPartitionNum | 合并前的初始分区数（并行度保底值） | 200 |
| spark.sql.adaptive.advisoryPartitionSizeInBytes | 合并后目标分区大小 | 1024MB |

**最终阶段（Final Stage）独立配置**：

最终阶段（写入输出表）的分区策略可单独配置，避免影响中间计算阶段：

| 参数 | 说明 | 示例值 |
| --- | --- | --- |
| spark.sql.finalStage.adaptive.coalescePartitions.enabled | 最终阶段启用合并 | true |
| spark.sql.optimizer.finalStageConfigIsolation.enabled | 最终阶段参数隔离 | true |
| spark.sql.finalStage.adaptive.coalescePartitions.minPartitionNum | 最终阶段最少分区数 | 200 |
| spark.sql.finalStage.adaptive.coalescePartitions.minPartitionSize | 最终阶段单分区最小大小 | 512MB |
| spark.sql.finalStage.adaptive.advisoryPartitionSizeInBytes | 最终阶段目标分区大小 | 1024MB |

## 2.5 JOIN 策略

| 参数 | 说明 | 示例值 |
| --- | --- | --- |
| spark.sql.autoBroadcastJoinThreshold | 自动广播的表大小阈值 | 52428800（50MB）或 -1（关闭） |

### 2.6 自定义参数

| 参数 | 说明 | 示例值 |
| --- | --- | --- |
| spark.custom.hiveToSpark.flag | 将老任务切换到 Spark 3.3.4 网关 | 1 |
| spark.custom.rss.service.enable | 开启远端 Shuffle 服务（RSS） | true |

> **RSS（Remote Shuffle Service）**：将 Shuffle 数据写入远端存储，避免 Executor 被回收时 Shuffle 数据丢失，需要重新计算。对动态分配场景尤其重要。

## 三、成本治理

### 3.1 优化优先级：先优化 SQL，再调资源

SQL 层面的优化（减少数据扫描、优化 JOIN、合理缓存）往往能带来数量级的性能提升，而单纯增加资源配置只是线性扩展，且成本线性增长。

**推荐顺序**：

1. 优化 SQL 查询逻辑（减少扫描量、优化 JOIN、减少 Shuffle）
2. 合理使用缓存策略（避免重复计算，但不过度缓存）
3. 最后才考虑调整资源配置

### 3.2 避免资源浪费

- **不要盲目加大 Executor 数量和内存**：未优化的 SQL 即使给再多资源也会因为数据倾斜、重复计算等问题导致效率低下
- **善用动态分配**：通过 `spark.dynamicAllocation` 让 Spark 按需申请和释放 Executor，避免资源闲置
- **开启 AQE**：让 Spark 自动合并小分区、优化执行计划，减少人工调参的盲目性
- **及时释放缓存**：大视图聚合裁剪后及时 `UNCACHE`，避免存储内存长期被占用

## 3.3 常见资源异常与排查

### 异常一：Executor 不足导致并行度低

**表现**：Task 数量较多的 Stage 执行时间异常长，并行线程数始终远低于设定上限，大量 Task 处于等待状态。

**原因**：集群可用资源不足，无法申请到足够的 Executor。

**解决思路**：

- 检查集群资源利用率，避开高峰期
- 适当降低单个 Executor 的资源请求（memory / cores），换取更多 Executor 数量
- 优化 SQL 减少数据量，从根本上降低资源需求

### 异常二：内存不足导致 Executor 被终止

**表现**：任务日志报内存不足，Executor 异常退出。常见退出码：

- **Exit Code 137**（SIGKILL）：K8s 因容器内存超出限制（`memory + memoryOverhead`）直接 OOM Kill，最常见的 OOM 场景
- **Exit Code 143**（SIGTERM）：Executor 被 Driver 主动关闭或 K8s Pod 驱逐，通常是资源不足时的优雅终止

**原因**：Executor 可用内存不足以容纳 Shuffle 中间数据，特别是在数据膨胀或倾斜的场景下。堆外内存溢出（如 Netty 传输缓冲、序列化对象）导致容器总内存超限。

**解决思路**：

- 优先检查 SQL 是否存在数据膨胀（如笛卡尔积、未过滤的 JOIN）
- 如果是 137：增加 `spark.executor.memoryOverhead`（堆外内存），使容器总内存限制更宽裕
- 如果是 143：检查集群资源是否充足，是否触发了 Pod 驱逐
- 对倾斜数据做 Repartition 或 Salting 处理

### 异常三：缓存数据丢失引发连锁问题

**表现**：Executor 被强制关闭后，其上缓存的视图分区数据一并丢失，后续查询触发重新计算，进一步加剧内存压力，形成恶性循环。

**解决思路**：

- 使用 `MEMORY_AND_DISK_SER` 存储级别，允许缓存溢写到磁盘
- 控制缓存视图的总数据量，及时释放不需要的缓存
- 开启 RSS，将 Shuffle 数据存储到远端，减少 Executor 丢失的影响

## 3.4 存储成本优化

存储成本往往是大数据平台中占比最高的长期支出。通过优化落表时的数据排序、文件数量和压缩格式，可以在不改变业务逻辑的前提下大幅降低存储费用。

### 策略一：SORT BY 最大化列式压缩效率

ORC/Parquet 等列式存储格式底层依赖 **RLE（Run-Length Encoding，游程编码）** 等压缩算法——连续相同的值越多，压缩率越高。通过在写入前对数据按低基数列排序，可以让同一列中相同值尽可能聚集在一起，显著提升压缩率。

```
-- 落表时按低基数字段排序，使相同值连续排列，最大化 RLE 压缩收益
INSERT OVERWRITE TABLE target_table PARTITION(dt = '2026-03-05')
SELECT col1, col2, col3, ...
FROM source_table
SORT BY site_tp, category_id, status;  -- 选择基数从低到高的字段排序
```

**选择排序字段的原则**：

- 优先选择**低基数**（distinct 值少）的字段，如状态码、国家、类目等
- 排序字段的顺序按基数**从低到高**排列，使最外层分组最大
- `SORT BY` 是分区内排序（每个 Reducer 内部排序），不会触发全局 Shuffle，开销可控
- 避免对高基数字段（如用户 ID、订单号）排序，收益极低且增加排序开销

**压缩效果参考**：对于枚举型字段（如 `status` 只有 5 个取值），排序后 RLE 压缩率可提升 **30%\~70%**，表越大收益越明显。

### 策略二：重新分区控制文件数量，避免小文件

落表时每个分区（Partition）生成的文件数等于写入该分区的 Task 数。如果 Task 过多或数据分布不均，会产生大量小文件，导致：

- HDFS/S3 NameNode 元数据压力增大
- 下游读取时产生大量小 IO，严重影响查询性能
- 存储碎片化，浪费存储空间

使用 `REPARTITION_BY_RANGE` Hint 按分区键做范围重分区，既能控制输出文件数量，又能保证数据有序（天然支持策略一的排序收益）。

```
-- 方式一：通过 Hint 控制输出文件数，同时按字段范围分布保证数据有序
INSERT OVERWRITE TABLE target_table PARTITION(dt = '2026-03-05')
SELECT /*+ REPARTITION_BY_RANGE(200, category_id, shop_id) */
    col1, col2, col3, ...
FROM source_table;

-- 方式二：结合 AQE 最终阶段合并（适用于不确定最终数据量的场景）
-- 通过 finalStage 参数让 AQE 自动合并最终阶段的小分区
SET spark.sql.finalStage.adaptive.advisoryPartitionSizeInBytes = 256MB;
SET spark.sql.finalStage.adaptive.coalescePartitions.enabled = true;
```

**REPARTITION vs REPARTITION_BY_RANGE 的区别**：

| 特性 | REPARTITION（Hash） | REPARTITION_BY_RANGE（Range） |
| --- | --- | --- |
| 分区方式 | 按字段 Hash 取模 | 按字段值范围划分 |
| 数据有序性 | 分区内无序 | 分区内按指定字段有序 |
| 适用场景 | 下游需要按 Key 关联/聚合 | 落表时控制文件数 + 排序压缩 |
| 与 SORT BY 的配合 | 需额外加 SORT BY | 自身已保证分区内有序 |

> **最佳实践**：落表场景优先用 `REPARTITION_BY_RANGE`，一步到位实现文件数量控制和数据排序，无需额外加 `SORT BY`。

## 策略三：ORC 转 Parquet + ZSTD 高压缩等级

Parquet + ZSTD 组合在压缩率和读取性能上通常优于 ORC + ZLIB（默认），尤其是提高 ZSTD 压缩等级后，可以在几乎不影响读取速度的前提下进一步压缩存储体积。

**相关配置参数**：

```
-- 设置 Parquet 压缩算法为 ZSTD
SET spark.sql.parquet.compression.codec = zstd;

-- 提高 ZSTD 压缩等级（默认 1，最大 22，建议 3~6 兼顾压缩率和写入速度）
SET spark.sql.parquet.compression.codec.zstd.level = 6;
```

**压缩格式对比**：

| 格式 + 压缩 | 压缩率 | 写入速度 | 读取速度 | 适用场景 |
| --- | --- | --- | --- | --- |
| ORC + ZLIB | 高 | 较慢 | 快 | Hive 生态默认方案 |
| ORC + SNAPPY | 中 | 快 | 快 | 对写入速度敏感 |
| Parquet + SNAPPY | 中 | 快 | 快 | Spark 生态默认方案 |
| Parquet + ZSTD(1) | 高 | 快 | 快 | 推荐基线方案 |
| **Parquet + ZSTD(3~6)** | **很高** | **中等** | **快** | **存储成本敏感的大表，推荐** |
| Parquet + ZSTD(10+) | 极高 | 慢 | 快 | 归档数据、极少写入 |

**建表示例**：

```
-- 新建 Parquet + ZSTD 表
CREATE TABLE IF NOT EXISTS dw.target_table (
    col1 STRING,
    col2 BIGINT,
    ...
)
PARTITIONED BY (dt STRING)
STORED AS PARQUET
TBLPROPERTIES (
    'parquet.compression' = 'ZSTD'
);

-- 已有 ORC 表迁移为 Parquet + ZSTD（通过 CTAS）
SET spark.sql.parquet.compression.codec = zstd;
SET spark.sql.parquet.compression.codec.zstd.level = 6;

CREATE TABLE dw.target_table_parquet
STORED AS PARQUET
AS SELECT * FROM dw.target_table_orc;
```

> **注意事项**：
>
> - ZSTD 压缩等级越高，写入越慢但压缩率越好，读取速度几乎不受影响（ZSTD 解压速度与等级无关）
> - 建议对**存储量大、读多写少**的表优先改造（如 DW 层宽表、历史归档表）
> - 迁移前后建议对比文件大小和下游查询性能，确认收益

## 三策略组合使用

三个策略可以叠加使用，达到最优存储效果：

```
SET spark.sql.parquet.compression.codec = zstd;
SET spark.sql.parquet.compression.codec.zstd.level = 6;

INSERT OVERWRITE TABLE dw.target_table PARTITION(dt = '2026-03-05')
SELECT /*+ REPARTITION_BY_RANGE(200, col1) */  -- 策略2：控制文件数 + 排序，对主键排序
    col1, col2, col3, ...
FROM source_table
SORT BY col2
    , col3
;
-- REPARTITION_BY_RANGE 可以保证指定字段有序，这时候RLE压缩已经有收益，所以sort by非必要     -- 策略1+
-- 表本身使用 Parquet + ZSTD(6) 存储                              -- 策略3：高压缩率
```

# 总结：优化手段成本收益一览

| 优化方向 | 具体手段 | 降低计算成本 | 降低存储成本 | 实施难度 | 优先级 |
|---|---|---|---|---|---|
| 代码优化 | 减少 SELECT *，只查必要列 | 中 | - | 低 | P0 |
| | LEFT JOIN → INNER JOIN（语义等价时） | 中 | - | 低 | P0 |
| | Broadcast Join（大小表关联） | 高 | - | 低 | P0 |
| | 缓存视图避免重复计算 | 高 | - | 中 | P0 |
| | 及时 UNCACHE 释放大视图 | 高 | - | 低 | P0 |
| | Repartition 预分区减少下游 Shuffle | 高 | - | 中 | P1 |
| | 构建关联键字典替代非等值 JOIN（特定场景） | 高 | - | 高 | P2 |
| 集群调参 | 动态分配（按需伸缩 Executor） | 高 | - | 低 | P0 |
| | 开启 AQE 自适应查询 | 中 | - | 低 | P0 |
| | 合理设置内存比例（fraction / storageFraction） | 中 | - | 中 | P1 |
| | 调整 shuffle.partitions | 中 | - | 中 | P1 |
| | 开启 RSS 远端 Shuffle 服务 | 中 | - | 低 | P1 |
| | Final Stage 独立分区合并 | 低 | 中 | 低 | P2 |
| 成本治理 | SORT BY 低基数列排序提升压缩率 | - | 高 | 低 | P0 |
| | REPARTITION_BY_RANGE 控制文件数 | 中 | 高 | 低 | P0 |
| | ORC → Parquet + ZSTD 高压缩等级 | - | 高 | 中 | P0 |
| | 异常排查（OOM / Executor 不足） | 高 | - | 中 | P1 |

**阅读说明**：「高/中/低」表示该手段对成本的降低幅度；「-」表示该维度无直接收益；优先级 P0 > P1 > P2。

**核心原则**：代码优化的 ROI 最高——零额外资源投入，效果可达数量级提升；集群调参是辅助手段，解决资源利用率问题；成本治理面向长期存储支出，收益随数据量增长持续放大。

# 附录：实际案例

以下是一个完整的优化案例，综合运用了缓存视图、Repartition、分步计算等策略：

```
-- 加载 UDF
ADD JAR s3://cep2prod1dataabc/data/common/jars/mysql-connector-java-5.1.40.jar;
ADD JAR s3://cep2prod1dataabc/data/common/jars/HiveUdfs.jar;
CREATE TEMPORARY FUNCTION MaxPartition AS 'org.hive.wh.udf.MaxPartition';

-- Step 1: 缓存基础数据，按后续关联字段预分区，减少下游 Shuffle
CACHE TABLE dm_pub_supplier_shop_repeat_tag_df_01 OPTIONS('storageLevel' 'MEMORY_AND_DISK_SE
SELECT /*+ REPARTITION(3000, shop_id, supplier_id) */
    goods_sn AS skc, shop_id, supplier_id
FROM dw.dw_pub_gds_info_td
WHERE site_tp = 'shein' AND onsale_flag = 1
GROUP BY goods_sn, shop_id, supplier_id;

-- Step 2: 相似 SKC 关联（利用 Step1 的缓存和分区）
CREATE TEMP VIEW dm_pub_supplier_shop_repeat_tag_df_02 AS
SELECT skcid_x, skcid_y, shop_id_x, shop_id_y, MAX(score) AS score
FROM (
    SELECT a.origin_goods_sn AS skcid_x, a.similar_goods_sn AS skcid_y,
           a.score, b.shop_id AS shop_id_x, c.shop_id AS shop_id_y
    FROM dmp.dmp_s15_s_s_multi_goods_sn_search_df_v2 a
    JOIN dm_pub_supplier_shop_repeat_tag_df_01 b ON a.origin_goods_sn = b.skc
    JOIN dm_pub_supplier_shop_repeat_tag_df_01 c ON a.similar_goods_sn = c.skc
    WHERE a.dt = SPLIT(MaxPartition('dmp.dmp_s15_s_s_multi_goods_sn_search_df_v2'), '/')[0]
      AND a.score >= 0.9
    UNION ALL
    SELECT a.similar_goods_sn AS skcid_x, a.origin_goods_sn AS skcid_y,
           a.score, c.shop_id AS shop_id_x, b.shop_id AS shop_id_y
    FROM dmp.dmp_s15_s_s_multi_goods_sn_search_df_v2 a
    JOIN dm_pub_supplier_shop_repeat_tag_df_01 b ON a.origin_goods_sn = b.skc
    JOIN dm_pub_supplier_shop_repeat_tag_df_01 c ON a.similar_goods_sn = c.skc
    WHERE a.dt = SPLIT(MaxPartition('dmp.dmp_s15_s_s_multi_goods_sn_search_df_v2'), '/')[0]
      AND a.score >= 0.9
) a
GROUP BY skcid_x, skcid_y, shop_id_x, shop_id_y;

-- Step 3: 计算店铺重合 SKC 数
CREATE TEMP VIEW dm_pub_supplier_shop_repeat_tag_df_03 AS
SELECT shop_id_x, shop_id_y, COUNT(shop_chonghe_skc) AS shop_chonghe_skc_cnt
FROM (
    SELECT a.shop_id AS shop_id_x, b.shop_id_y,
           CASE WHEN b.skcid_x IS NOT NULL THEN a.skc ELSE NULL END AS shop_chonghe_skc
    FROM dm_pub_supplier_shop_repeat_tag_df_01 a
    JOIN dm_pub_supplier_shop_repeat_tag_df_02 b ON a.skc = b.skcid_x
    GROUP BY 1, 2, 3
) a
GROUP BY shop_id_x, shop_id_y;

-- Step 4: 计算重合率指标
CREATE TEMP VIEW dm_pub_supplier_shop_repeat_tag_df_04 AS
SELECT shop_id_x,
    MAX(overlap_ratio) AS overlap_ratio_all,
    MAX(CASE WHEN shop_honghe_type = '单店铺重复铺货' THEN overlap_ratio END) AS overlap_ratio_
    MAX(CASE WHEN shop_honghe_type = '跨店重复铺货'   THEN overlap_ratio END) AS overlap_ratio
```

```
    MAX(CASE WHEN overlap_ratio >= 0.5 THEN 1 ELSE 0 END) AS if_label_all1,
    MAX(CASE WHEN shop_honghe_type = '单店铺重复铺货' AND overlap_ratio >= 0.5 THEN 1 ELSE 0 EN
    MAX(CASE WHEN shop_honghe_type = '跨店重复铺货'   AND overlap_ratio >= 0.5 THEN 1 ELSE 0 E
FROM (
    SELECT a.shop_id AS shop_id_x,
        COALESCE(1.0000 * b.shop_chonghe_skc_cnt / a.skc_cnt, 0) AS overlap_ratio,
        CASE WHEN b.shop_id_x IS NULL THEN '非店铺重复铺货' ELSE '店铺重复铺货' END AS if_shop_ho
        CASE
            WHEN a.shop_id = b.shop_id_y  THEN '单店铺重复铺货'
            WHEN a.shop_id <> b.shop_id_y THEN '跨店重复铺货'
        END AS shop_honghe_type
    FROM (
        SELECT shop_id, COUNT(DISTINCT skc) AS skc_cnt
        FROM dm_pub_supplier_shop_repeat_tag_df_01
        GROUP BY shop_id
    ) a
    LEFT JOIN dm_pub_supplier_shop_repeat_tag_df_03 b ON a.shop_id = b.shop_id_x
) a
GROUP BY shop_id_x;

-- Step 5: 输出最终结果表
DROP TABLE IF EXISTS tmp.dm_pub_supplier_shop_repeat_tag_df_10151238;
CREATE TABLE tmp.dm_pub_supplier_shop_repeat_tag_df_10151238 AS
SELECT a.*,
    CASE WHEN a.if_label_all1 = 1     THEN '重复铺货店铺'     ELSE '非重复铺货店铺'    END AS if_
    CASE WHEN a.if_label_dandian1 = 1 THEN '单店重复铺货店铺'  ELSE '非单店重复铺货店铺' END AS if
    CASE WHEN a.if_label_kuadian1 = 1 THEN '跨店重复铺货店铺'  ELSE '非跨店重复铺货店铺' END AS if
    b.skc_cnt AS usonsale_skc_cnt
FROM (
    SELECT a.supplier_id, a.shop_id,
        b.overlap_ratio_all, b.overlap_ratio_dandian, b.overlap_ratio_kuadian,
        b.if_label_all1, b.if_label_dandian1, b.if_label_kuadian1
    FROM (
        SELECT supplier_id, shop_id
        FROM dm_pub_supplier_shop_repeat_tag_df_01
        GROUP BY supplier_id, shop_id
    ) a
    JOIN dm_pub_supplier_shop_repeat_tag_df_04 b
    ON a.shop_id = b.shop_id_x AND b.overlap_ratio_dandian >= 0.8
) a
LEFT JOIN (
    SELECT shop_id, COUNT(DISTINCT skc) AS skc_cnt
    FROM dm_pub_supplier_shop_repeat_tag_df_01
    GROUP BY shop_id
) b ON a.shop_id = b.shop_id;
```