尚硅谷大数据项目之Flink优化

(作者：尚硅谷大数据研发部)

版本：V2.0

# 第1章 资源配置调优

Flink性能调优的第一步，就是为任务分配合适的资源，在一定范围内，增加资源的分配与性能的提升是成正比的，实现了最优的资源配置后，在此基础上再考虑进行后面论述的性能调优策略。

提交方式主要是yarn-per-job，资源的分配在使用脚本提交Flink任务时进行指定。

标准的Flink任务提交脚本（Generic CLI 模式），从1.11开始，增加了通用客户端模式，参数使用-D <property=value>指定

bin/flink run \

-t yarn-per-job \

-d \

-p 5 \ 指定并行度

-Dyarn.application.queue=test \ 指定yarn队列

-Djobmanager.memory.process.size=1024mb \ 指定JM的总进程大小

-Dtaskmanager.memory.process.size=1024mb \ 指定每个TM的总进程大小

-Dtaskmanager.numberOfTaskSlots=2 \ 指定每个TM的slot数

-c com.atguigu.app.dwd.LogBaseApp \

/opt/module/gmall-flink/gmall-realtime-1.0-SNAPSHOT-jar-with-dependencies.jar

参数列表：

<https://ci.apache.org/projects/flink/flink-docs-release-1.12/deployment/config.html>

## 1.1 内存设置（1CPU配置4G内存）

生产资源配置：

bin/flink run \

-t yarn-per-job \

-d \

-p 5 \ 指定并行度

-Dyarn.application.queue=test \ 指定yarn队列

-Djobmanager.memory.process.size=2048mb \ JM2~4G足够

-Dtaskmanager.memory.process.size=6144mb \ 单个TM2~8G足够

-Dtaskmanager.numberOfTaskSlots=2 \ 与容器核数1core：1slot或1core：2slot

-c com.atguigu.app.dwd.LogBaseApp \

/opt/module/gmall-flink/gmall-realtime-1.0-SNAPSHOT-jar-with-dependencies.jar

Flink是实时流处理，关键在于资源情况能不能抗住高峰时期每秒的数据量，通常用QPS/TPS来描述数据情况。

## 1.2 并行度设置

### 1.2.1 最优并行度计算

开发完成后，先进行压测。任务并行度给10以下，测试单个并行度的处理上限。然后 总QPS/单并行度的处理能力 = 并行度

不能只从QPS去得出并行度，因为有些字段少、逻辑简单的任务，单并行度一秒处理几万条数据。而有些数据字段多，处理逻辑复杂，单并行度一秒只能处理1000条数据。

最好根据高峰期的QPS压测，并行度\*1.2倍，富余一些资源。

### 1.2.2 Source 端并行度的配置

数据源端是 Kafka，Source的并行度设置为Kafka对应Topic的分区数。

如果已经等于 Kafka 的分区数，消费速度仍跟不上数据生产速度，考虑下Kafka 要扩大分区，同时调大并行度等于分区数。

Flink 的一个并行度可以处理一至多个分区的数据，如果并行度多于 Kafka 的分区数，那么就会造成有的并行度空闲，浪费资源。

### 1.2.3 Transform端并行度的配置

* Keyby之前的算子

一般不会做太重的操作，都是比如map、filter、flatmap等处理较快的算子，并行度可以和source保持一致。

* Keyby之后的算子（KeyGroup 最小值为128）

如果并发较大，建议设置并行度为 2 的整数次幂，例如：128、256、512；

小并发任务的并行度不一定需要设置成 2 的整数次幂；

大并发任务如果没有 KeyBy，并行度也无需设置为 2 的整数次幂；

### 1.2.4 Sink 端并行度的配置

Sink 端是数据流向下游的地方，可以根据 Sink 端的数据量及下游的服务抗压能力进行评估。

如果Sink端是Kafka，可以设为Kafka对应Topic的分区数。

Sink 端的数据量小，比较常见的就是监控告警的场景，并行度可以设置的小一些。

Source 端的数据量是最小的，拿到 Source 端流过来的数据后做了细粒度的拆分，数据量不断的增加，到 Sink 端的数据量就非常大。那么在 Sink 到下游的存储中间件的时候就需要提高并行度。

另外 Sink 端要与下游的服务进行交互，并行度还得根据下游的服务抗压能力来设置，如果在 Flink Sink 这端的数据量过大的话，且 Sink 处并行度也设置的很大，但下游的服务完全撑不住这么大的并发写入，可能会造成下游服务直接被写挂，所以最终还是要在 Sink 处的并行度做一定的权衡。

## 1.3 RocksDB大状态调优

RocksDB 是基于 LSM Tree 实现的（类似HBase），写数据都是先缓存到内存中，所以RocksDB 的写请求效率比较高。RocksDB 使用内存结合磁盘的方式来存储数据，每次获取数据时，先从内存中 blockcache 中查找，如果内存中没有再去磁盘中查询。优化后差不多单并行度 TPS 5000 record/s，性能瓶颈主要在于 RocksDB 对磁盘的读请求，所以当处理性能不够时，仅需要横向扩展并行度即可提高整个Job 的吞吐量。以下几个调优参数：

* **设置本地 RocksDB 多目录**

在flink-conf.yaml 中配置：

state.backend.rocksdb.localdir: /data1/flink/rocksdb,/data2/flink/rocksdb,/data3/flink/rocksdb

注意：不要配置单块磁盘的多个目录，务必将目录配置到多块不同的磁盘上，让多块磁盘来分担压力。当设置多个 RocksDB 本地磁盘目录时，Flink 会随机选择要使用的目录，所以就可能存在三个并行度共用同一目录的情况。如果服务器磁盘数较多，一般不会出现该情况，但是如果任务重启后吞吐量较低，可以检查是否发生了多个并行度共用同一块磁盘的情况。

当一个 TaskManager 包含 3 个 slot 时，那么单个服务器上的三个并行度都对磁盘造成频繁读写，从而导致三个并行度的之间相互争抢同一个磁盘 io，这样务必导致三个并行度的吞吐量都会下降。设置多目录实现三个并行度使用不同的硬盘从而减少资源竞争。

如下所示是测试过程中磁盘的 IO 使用率，可以看出三个大状态算子的并行度分别对应了三块磁盘，这三块磁盘的 IO 平均使用率都保持在 45% 左右，IO 最高使用率几乎都是 100%，而其他磁盘的 IO 平均使用率相对低很多。由此可见使用 RocksDB 做为状态后端且有大状态的频繁读取时， 对磁盘IO性能消耗确实比较大。



如下图所示，其中两个并行度共用了 sdb 磁盘，一个并行度使用 sdj磁盘。可以看到 sdb 磁盘的 IO 使用率已经达到了 91.6%，就会导致 sdb 磁盘对应的两个并行度吞吐量大大降低，从而使得整个 Flink 任务吞吐量降低。如果每个服务器上有一两块 SSD，强烈建议将 RocksDB 的本地磁盘目录配置到 SSD 的目录下，从 HDD 改为 SSD 对于性能的提升可能比配置 10 个优化参数更有效。

* **state.backend.incremental：**开启增量检查点，默认false，改为true。
* **state.backend.rocksdb.predefined-options：**SPINNING\_DISK\_OPTIMIZED\_HIGH\_MEM设置为机械硬盘+内存模式，有条件上SSD，指定为FLASH\_SSD\_OPTIMIZED
* **state.backend.rocksdb.block.cache-size**: 整个 RocksDB 共享一个 block cache，读数据时内存的 cache 大小，该参数越大读数据时缓存命中率越高，默认大小为 8 MB，建议设置到 64 ~ 256 MB。
* **state.backend.rocksdb.thread.num**: 用于后台 flush 和合并 sst 文件的线程数，默认为 1，建议调大，机械硬盘用户可以改为 4 等更大的值。
* **state.backend.rocksdb.writebuffer.size**: RocksDB 中，每个 State 使用一个 Column Family，每个 Column Family 使用独占的 write buffer，建议调大，例如：32M
* **state.backend.rocksdb.writebuffer.count**: 每个 Column Family 对应的 writebuffer 数目，默认值是 2，对于机械磁盘来说，如果内存⾜够大，可以调大到 5 左右
* **state.backend.rocksdb.writebuffer.number-to-merge**: 将数据从 writebuffer 中 flush 到磁盘时，需要合并的 writebuffer 数量，默认值为 1，可以调成3。
* **state.backend.local-recovery**: 设置本地恢复，当 Flink 任务失败时，可以基于本地的状态信息进行恢复任务，可能不需要从 hdfs 拉取数据

## 1.4 Checkpoint设置

一般我们的 Checkpoint 时间间隔可以设置为分钟级别，例如 1 分钟、3 分钟，对于状态很大的任务每次 Checkpoint 访问 HDFS 比较耗时，可以设置为 5~10 分钟一次Checkpoint，并且调大两次 Checkpoint 之间的暂停间隔，例如设置两次Checkpoint 之间至少暂停 4或8 分钟。

如果 Checkpoint 语义配置为 EXACTLY\_ONCE，那么在 Checkpoint 过程中还会存在 barrier 对齐的过程，可以通过 Flink Web UI 的 Checkpoint 选项卡来查看 Checkpoint 过程中各阶段的耗时情况，从而确定到底是哪个阶段导致 Checkpoint 时间过长然后针对性的解决问题。

RocksDB相关参数在1.3中已说明，可以在flink-conf.yaml指定，也可以在Job的代码中调用API单独指定，这里不再列出。

// 使⽤ RocksDBStateBackend 做为状态后端，并开启增量 Checkpoint

RocksDBStateBackend rocksDBStateBackend = new RocksDBStateBackend("hdfs://hadoop102:8020/flink/checkpoints", true);

env.setStateBackend(rocksDBStateBackend);

// 开启Checkpoint，间隔为 3 分钟

env.enableCheckpointing(TimeUnit.MINUTES.toMillis(3));

// 配置 Checkpoint

CheckpointConfig checkpointConf = env.getCheckpointConfig();

checkpointConf.setCheckpointingMode(CheckpointingMode.EXACTLY\_ONCE)

// 最小间隔 4分钟

checkpointConf.setMinPauseBetweenCheckpoints(TimeUnit.MINUTES.toMillis(4))

// 超时时间 10分钟

checkpointConf.setCheckpointTimeout(TimeUnit.MINUTES.toMillis(10));

// 保存checkpoint

checkpointConf.enableExternalizedCheckpoints(

CheckpointConfig.ExternalizedCheckpointCleanup.RETAIN\_ON\_CANCELLATION);

## 1.5 使用 Flink ParameterTool 读取配置

在实际开发中，有各种环境（开发、测试、预发、生产），作业也有很多的配置：算子的并行度配置、Kafka 数据源的配置（broker 地址、topic 名、group.id）、Checkpoint 是否开启、状态后端存储路径、数据库地址、用户名和密码等各种各样的配置，可能每个环境的这些配置对应的值都是不一样的。

如果你是直接在代码⾥⾯写死的配置，每次换个环境去运行测试作业，都要重新去修改代码中的配置，然后编译打包，提交运行，这样就要花费很多时间在这些重复的劳动力上了。在 Flink 中可以通过使用 ParameterTool 类读取配置，它可以读取环境变量、运行参数、配置文件。

ParameterTool 是可序列化的，所以你可以将它当作参数进行传递给算子的自定义函数类。

### 1.5.1 读取运行参数

我们可以在Flink的提交脚本添加运行参数，格式：

* --参数名 参数值
* -参数名 参数值

在 Flink 程序中可以直接使用 ParameterTool.fromArgs(args) 获取到所有的参数，也可以通过 parameterTool.get("username") 方法获取某个参数对应的值。

举例：通过运行参数指定jobname

bin/flink run \

-t yarn-per-job \

-d \

-p 5 \ 指定并行度

-Dyarn.application.queue=test \ 指定yarn队列

-Djobmanager.memory.process.size=1024mb \ 指定JM的总进程大小

-Dtaskmanager.memory.process.size=1024mb \ 指定每个TM的总进程大小

-Dtaskmanager.numberOfTaskSlots=2 \ 指定每个TM的slot数

-c com.atguigu.app.dwd.LogBaseApp \

/opt/module/gmall-flink/gmall-realtime-1.0-SNAPSHOT-jar-with-dependencies.jar \

--jobname dwd-LogBaseApp //参数名自己随便起，代码里对应上即可

在代码里获取参数值：

ParameterTool parameterTool = ParameterTool.fromArgs(args);

String myJobname = parameterTool.get("jobname"); //参数名对应

env.execute(myJobname);

### 1.5.2 读取系统属性

ParameterTool 还⽀持通过 ParameterTool.fromSystemProperties() 方法读取系统属性。做个打印：

ParameterTool parameterTool = ParameterTool.fromSystemProperties();  
System.out.println(parameterTool.toMap().toString());

可以得到全面的系统属性，部分结果：



### 1.5.3 读取配置文件

可以使用ParameterTool.fromPropertiesFile("/application.properties") 读取 properties 配置文件。可以将所有要配置的地方（比如并行度和一些 Kafka、MySQL 等配置）都写成可配置的，然后其对应的 key 和 value 值都写在配置文件中，最后通过 ParameterTool 去读取配置文件获取对应的值。

### 1.5.4 注册全局参数

在 ExecutionConfig 中可以将 ParameterTool 注册为全作业参数的参数，这样就可以被 JobManager 的web 端以及用户⾃定义函数中以配置值的形式访问。

StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment(); env.getConfig().setGlobalJobParameters(ParameterTool.fromArgs(args));

可以不用将ParameterTool当作参数传递给算子的自定义函数，直接在用户⾃定义的 Rich 函数中直接获取到参数值了。

env.addSource(new RichSourceFunction() {

@Override

public void run(SourceContext sourceContext) throws Exception {

while (true) {

ParameterTool parameterTool = (ParameterTool)getRuntimeContext().getExecutionConfig().getGlobalJobParameters();

}

}

@Override

public void cancel() {

}

})

## 1.6 压测方式

压测的方式很简单，先在kafka中积压数据，之后开启Flink任务，出现反压，就是处理瓶颈。相当于水库先积水，一下子泄洪。

数据可以是自己造的模拟数据，也可以是生产中的部分数据。

# 第2章 反压处理

反压（BackPressure）通常产生于这样的场景：短时间的负载高峰导致系统接收数据的速率远高于它处理数据的速率。许多日常问题都会导致反压，例如，垃圾回收停顿可能会导致流入的数据快速堆积，或遇到大促、秒杀活动导致流量陡增。反压如果不能得到正确的处理，可能会导致资源耗尽甚至系统崩溃。

反压机制是指系统能够自己检测到被阻塞的 Operator，然后自适应地降低源头或上游数据的发送速率，从而维持整个系统的稳定。Flink 任务一般运行在多个节点上，数据从上游算子发送到下游算子需要网络传输，若系统在反压时想要降低数据源头或上游算子数据的发送速率，那么肯定也需要网络传输。所以下面先来了解一下 Flink 的网络流控（Flink 对网络数据流量的控制）机制。

## 2.1 反压现象及定位

Flink 的反压太过于天然了，导致无法简单地通过监控 BufferPool 的使用情况来判断反压状态。Flink 通过对运行中的任务进行采样来确定其反压，如果一个 Task 因为反压导致处理速度降低了，那么它肯定会卡在向 LocalBufferPool 申请内存块上。那么该 Task 的 stack trace 应该是这样：

java.lang.Object.wait(Native Method)

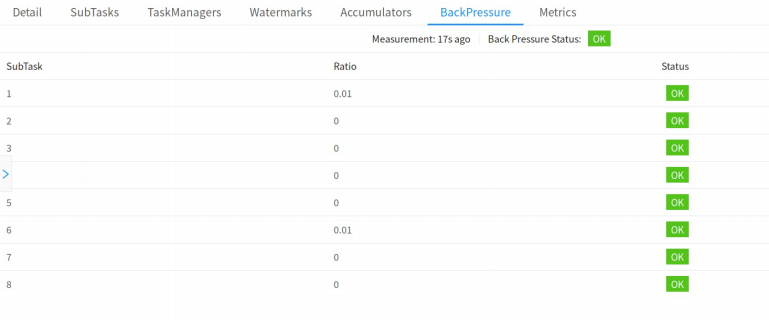
o.a.f.[...].LocalBufferPool.requestBuffer(LocalBufferPool.java:163) o.a.f.[...].LocalBufferPool.requestBufferBlocking(LocalBufferPool.java:133) [...]

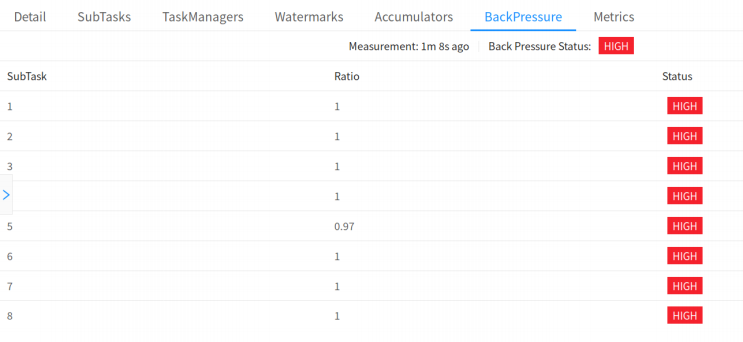
监控对正常的任务运行有一定影响，因此只有当Web页面切换到 Job 的 BackPressure 页面时，JobManager 才会对该 Job 触发反压监控。默认情况下，JobManager 会触发 100 次 stack trace 采样，每次间隔 50ms 来确定反压。Web 界面看到的比率表示在内部方法调用中有多少 stack trace 被卡在LocalBufferPool.requestBufferBlocking()，例如: 0.01 表示在 100 个采样中只有 1 个被卡在LocalBufferPool.requestBufferBlocking()。采样得到的比例与反压状态的对应关系如下：

* OK: 0 <= 比例 <= 0.10
* LOW: 0.10 < 比例 <= 0.5
* HIGH: 0.5 < 比例 <= 1

Task 的状态为 OK 表示没有反压，HIGH 表示这个 Task 被反压。

### 2.1.1 利用 Flink Web UI 定位产生反压的位置

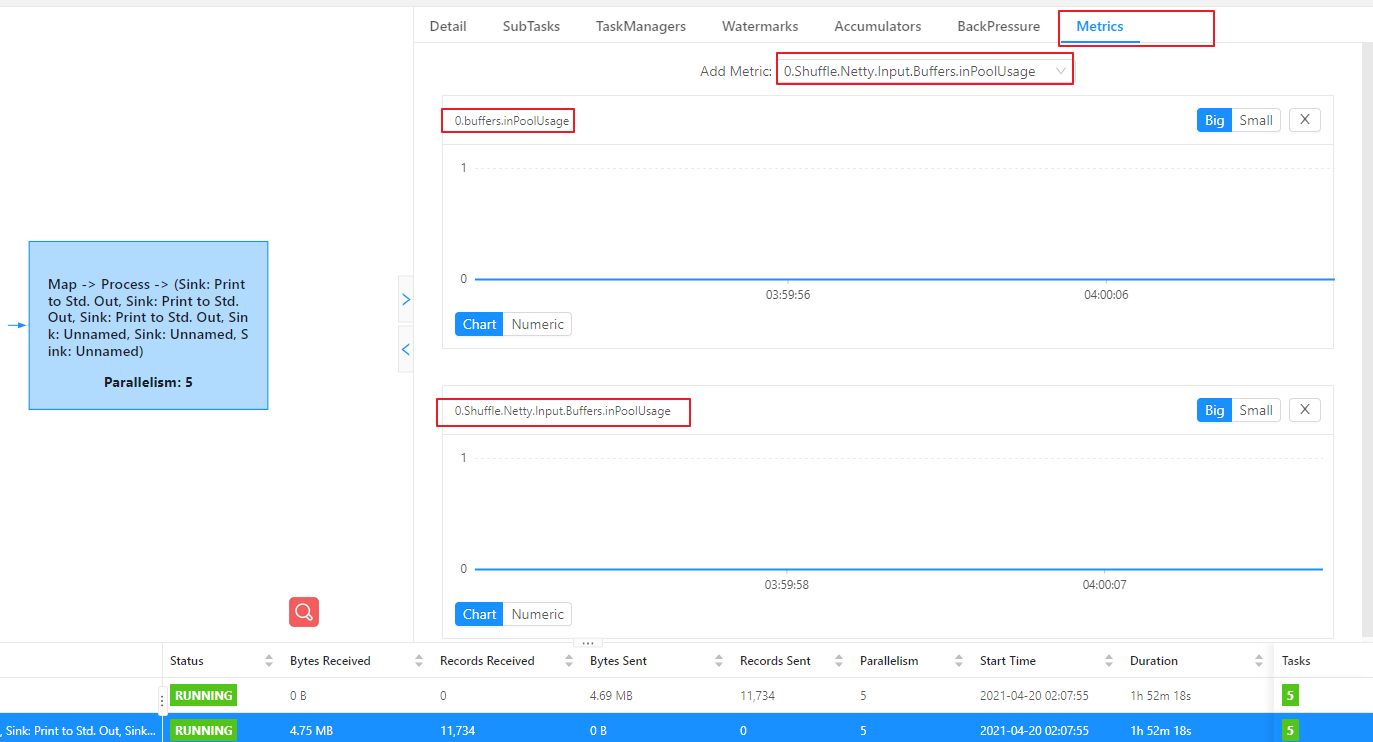
在 Flink Web UI 中有 BackPressure 的页面，通过该页面可以查看任务中 subtask 的反压状态，如下两图所示，分别展示了状态是 OK 和 HIGH 的场景。排查的时候，先把operator chain禁用，方便定位。



### 2.1.2 利用Metrics定位反压位置

当某个 Task 吞吐量下降时，基于 Credit 的反压机制，上游不会给该 Task 发送数据，所以该 Task 不会频繁卡在向 Buffer Pool 去申请 Buffer。反压监控实现原理就是监控 Task 是否卡在申请 buffer 这一步，所以遇到瓶颈的 Task 对应的反压⻚⾯必然会显示 OK，即表示没有受到反压。

如果该 Task 吞吐量下降，造成该Task 上游的 Task 出现反压时，必然会存在：该 Task 对应的 InputChannel 变满，已经申请不到可用的Buffer 空间。如果该 Task 的 InputChannel 还能申请到可用 Buffer，那么上游就可以给该 Task 发送数据，上游 Task 也就不会被反压了，所以说遇到瓶颈且导致上游 Task 受到反压的 Task 对应的 InputChannel 必然是满的（这⾥不考虑⽹络遇到瓶颈的情况）。从这个思路出发，可以对该 Task 的 InputChannel 的使用情况进行监控，如果 InputChannel 使用率 100%，那么该 Task 就是我们要找的反压源。Flink 1.9 及以上版本inPoolUsage 表示 inputFloatingBuffersUsage 和inputExclusiveBuffersUsage 的总和。



反压时，可以看到遇到瓶颈的该Task的inPoolUage为1。

## 2.2 反压的原因及处理

先检查基本原因，然后再深入研究更复杂的原因，最后找出导致瓶颈的原因。下面列出从最基本到比较复杂的一些反压潜在原因。

注意：反压可能是暂时的，可能是由于负载高峰、CheckPoint 或作业重启引起的数据积压而导致反压。如果反压是暂时的，应该忽略它。另外，请记住，断断续续的反压会影响我们分析和解决问题。

### 2.2.1 系统资源

检查涉及服务器基本资源的使用情况，如CPU、网络或磁盘I/O，目前 Flink 任务使用最主要的还是内存和 CPU 资源，本地磁盘、依赖的外部存储资源以及网卡资源一般都不会是瓶颈。如果某些资源被充分利用或大量使用，可以借助分析工具，分析性能瓶颈（JVM Profiler+ FlameGraph生成火焰图）。

如何生成火焰图：<http://www.54tianzhisheng.cn/2020/10/05/flink-jvm-profiler/>

如何读懂火焰图：<https://zhuanlan.zhihu.com/p/29952444>

* 针对特定的资源调优Flink
* 通过增加并行度或增加集群中的服务器数量来横向扩展
* 减少瓶颈算子上游的并行度，从而减少瓶颈算子接收的数据量（不建议，可能造成整个Job数据延迟增大）

### 2.2.2 垃圾收集（GC）

长时间GC暂停会导致性能问题。可以通过打印调试GC日志（通过-XX:+PrintGCDetails）或使用某些内存或 GC 分析器（GCViewer工具）来验证是否处于这种情况。

* 在Flink提交脚本中,设置JVM参数，打印GC日志：

bin/flink run \

-t yarn-per-job \

-d \

-p 5 \ 指定并行度

-Dyarn.application.queue=test \ 指定yarn队列

-Djobmanager.memory.process.size=1024mb \ 指定JM的总进程大小

-Dtaskmanager.memory.process.size=1024mb \ 指定每个TM的总进程大小

-Dtaskmanager.numberOfTaskSlots=2 \ 指定每个TM的slot数

-Denv.java.opts="-XX:+PrintGCDetails -XX:+PrintGCDateStamps"

-c com.atguigu.app.dwd.LogBaseApp \

/opt/module/gmall-flink/gmall-realtime-1.0-SNAPSHOT-jar-with-dependencies.jar

* 下载GC日志的方式：

因为是on yarn模式，运行的节点一个一个找比较麻烦。可以打开WebUI，选择JobManager或者TaskManager，点击Stdout，即可看到GC日志，点击下载按钮即可将GC日志通过HTTP的方式下载下来。



* 分析GC日志：

通过 GC 日志分析出单个 Flink Taskmanager 堆总大小、年轻代、老年代分配的内存空间、Full GC 后老年代剩余大小等，相关指标定义可以去 Github 具体查看。

GCViewer地址：<https://github.com/chewiebug/GCViewer>

*扩展：最重要的指标是Full GC 后，老年代剩余大小这个指标，按照《Java 性能优化权威指南》这本书 Java 堆大小计算法则，设 Full GC 后老年代剩余大小空间为 M，那么堆的大小建议 3 ~ 4倍 M，新生代为 1 ~ 1.5 倍 M，老年代应为 2 ~ 3 倍 M。*

### 2.2.3 CPU/线程瓶颈

有时，一个或几个线程导致 CPU 瓶颈，而整个机器的CPU使用率仍然相对较低，则可能无法看到 CPU 瓶颈。例如，48核的服务器上，单个 CPU 瓶颈的线程仅占用 2％的 CPU 使用率，就算单个线程发生了 CPU 瓶颈，我们也看不出来。可以考虑使用2.2.1提到的分析工具，它们可以显示每个线程的 CPU 使用情况来识别热线程。

### 2.2.4 线程竞争

与上⾯的 CPU/线程瓶颈问题类似，subtask 可能会因为共享资源上高负载线程的竞争而成为瓶颈。同样，可以考虑使用2.2.1提到的分析工具，考虑在用户代码中查找同步开销、锁竞争，尽管避免在用户代码中添加同步。

### 2.2.5 负载不平衡

如果瓶颈是由数据倾斜引起的，可以尝试通过将数据分区的 key 进行加盐或通过实现本地预聚合来减轻数据倾斜的影响。（关于数据倾斜的详细解决方案，会在下一章节详细讨论）

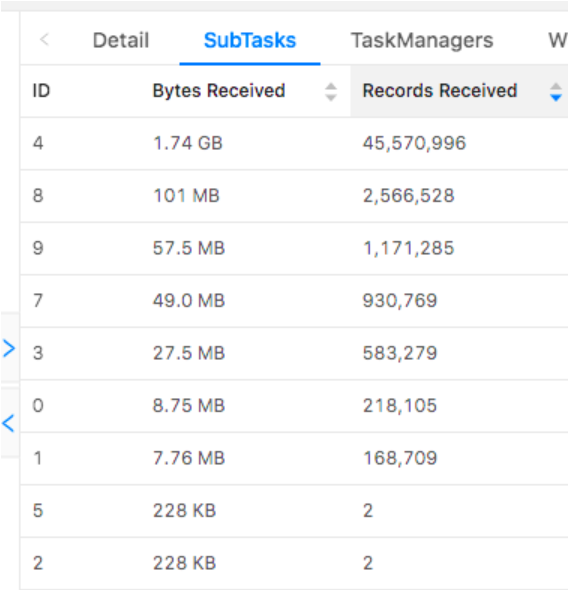
### 2.2.6 外部依赖

如果发现我们的 Source 端数据读取性能比较低或者 Sink 端写入性能较差，需要检查第三方组件是否遇到瓶颈。例如，Kafka 集群是否需要扩容，Kafka 连接器是否并行度较低，HBase 的 rowkey 是否遇到热点问题。关于第三方组件的性能问题，需要结合具体的组件来分析。

# 第3章 数据倾斜

## 3.1 判断是否存在数据倾斜

相同 Task 的多个 Subtask 中，个别Subtask 接收到的数据量明显大于其他 Subtask 接收到的数据量，通过 Flink Web UI 可以精确地看到每个 Subtask 处理了多少数据，即可判断出 Flink 任务是否存在数据倾斜。通常，数据倾斜也会引起反压。



## 3.2 数据倾斜的解决

### 3.2.1 keyBy 之前发生数据倾斜

如果 keyBy 之前就存在数据倾斜，上游算子的某些实例可能处理的数据较多，某些实例可能处理的数据较少，产生该情况可能是因为数据源的数据本身就不均匀，例如由于某些原因 Kafka 的 topic 中某些 partition 的数据量较大，某些 partition 的数据量较少。对于不存在 keyBy 的 Flink 任务也会出现该情况。

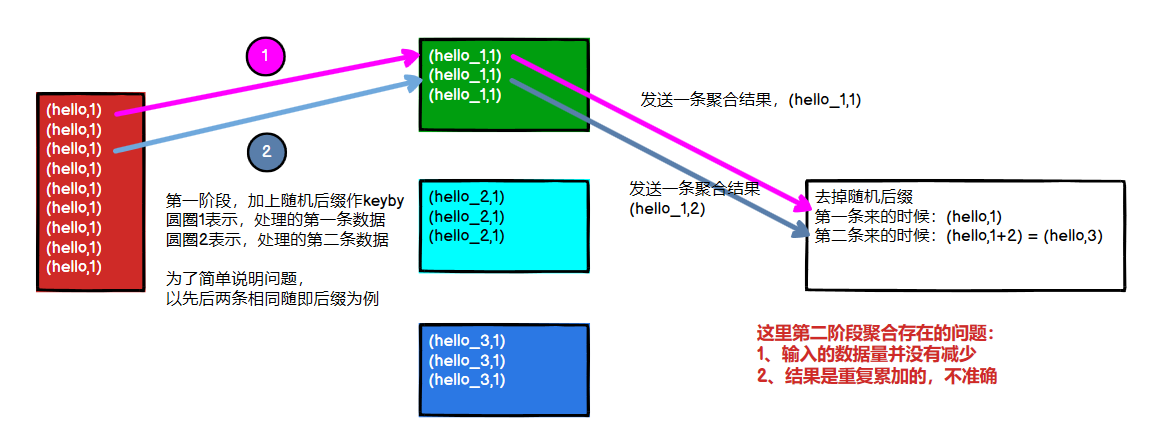
这种情况，需要让 Flink 任务强制进行shuffle。使用shuffle、rebalance 或 rescale算子即可将数据均匀分配，从而解决数据倾斜的问题。

### 3.2.2 keyBy 后的聚合操作存在数据倾斜

使用LocalKeyBy的思想：在 keyBy 上游算子数据发送之前，首先在上游算子的本地

对数据进行聚合后再发送到下游，使下游接收到的数据量大大减少，从而使得 keyBy 之后的聚合操作不再是任务的瓶颈。类似MapReduce 中 Combiner 的思想，但是这要求聚合操作必须是多条数据或者一批数据才能聚合，单条数据没有办法通过聚合来减少数据量。从Flink LocalKeyBy 实现原理来讲，必然会存在一个积攒批次的过程，在上游算子中必须攒够一定的数据量，对这些数据聚合后再发送到下游。

**注意：**Flink是实时流处理，如果keyby之后的聚合操作存在数据倾斜，且没有开窗口的情况下，简单的认为使用两阶段聚合，是不能解决问题的。因为这个时候Flink是来一条处理一条，且向下游发送一条结果，对于原来keyby的维度（第二阶段聚合）来讲，数据量并没有减少，且结果重复计算（非FlinkSQL，未使用回撤流），如下图所示：



* **实现方式：**以计算PV为例，keyby之前，使用flatMap实现LocalKeyby

class LocalKeyByFlatMap extends RichFlatMapFunction<String, Tuple2<String,

//Checkpoint 时为了保证 Exactly Once，将 buffer 中的数据保存到该 ListState 中

private ListState<Tuple2<String, Long>> localPvStatListState;

//本地 buffer，存放 local 端缓存的 app 的 pv 信息

private HashMap<String, Long> localPvStat;

//缓存的数据量大小，即：缓存多少数据再向下游发送

private int batchSize;

//计数器，获取当前批次接收的数据量

private AtomicInteger currentSize;

//构造器，批次大小传参

LocalKeyByFlatMap(int batchSize){

this.batchSize = batchSize;

}

@Override

public void flatMap(String in, Collector collector) throws Exception {

// 将新来的数据添加到 buffer 中

Long pv = localPvStat.getOrDefault(in, 0L);

localPvStat.put(in, pv + 1);

// 如果到达设定的批次，则将 buffer 中的数据发送到下游

if(currentSize.incrementAndGet() >= batchSize){

// 遍历 Buffer 中数据，发送到下游

for(Map.Entry<String, Long> appIdPv: localPvStat.entrySet()) {

collector.collect(Tuple2.of(appIdPv.getKey(), appIdPv.getValue()

}

// Buffer 清空，计数器清零

localPvStat.clear();

currentSize.set(0);

}

}

@Override

public void snapshotState(FunctionSnapshotContext functionSnapshotConte

// 将 buffer 中的数据保存到状态中，来保证 Exactly Once

localPvStatListState.clear();

for(Map.Entry<String, Long> appIdPv: localPvStat.entrySet()) {

localPvStatListState.add(Tuple2.of(appIdPv.getKey(), appIdPv.ge

}

}

@Override

public void initializeState(FunctionInitializationContext context) {

// 从状态中恢复 buffer 中的数据

localPvStatListState = context.getOperatorStateStore().getListState

new ListStateDescriptor<>("localPvStat",

TypeInformation.of(new TypeHint<Tuple2<String, Long>>})));

localPvStat = new HashMap();

if(context.isRestored()) {

// 从状态中恢复数据到 localPvStat 中

for(Tuple2<String, Long> appIdPv: localPvStatListState.get()){

long pv = localPvStat.getOrDefault(appIdPv.f0, 0L);

// 如果出现 pv != 0,说明改变了并行度，

// ListState 中的数据会被均匀分发到新的 subtask中

// 所以单个 subtask 恢复的状态中可能包含两个相同的 app 的数据

localPvStat.put(appIdPv.f0, pv + appIdPv.f1);

}

// 从状态恢复时，默认认为 buffer 中数据量达到了 batchSize，需要向下游发

currentSize = new AtomicInteger(batchSize);

} else {

currentSize = new AtomicInteger(0);

}

}

}

### 3.2.3 keyBy 后的窗口聚合操作存在数据倾斜

因为使用了窗口，变成了有界数据的处理（3.2.2已分析过），窗口默认是触发时才会输出一条结果发往下游，所以可以使用两阶段聚合的方式：

**实现思路：**

* 第一阶段聚合：key拼接随机数前缀或后缀，进行keyby、开窗、聚合

*注意：聚合完不再是WindowedStream，要获取WindowEnd作为窗口标记作为第二阶段分组依据，避免不同窗口的结果聚合到一起）*

* 第二阶段聚合：去掉随机数前缀或后缀，按照原来的key及windowEnd作keyby、聚合

# 第4章 KafkaSource调优

## 4.1 动态发现分区

当 FlinkKafkaConsumer 初始化时，每个 subtask 会订阅一批 partition，但是当 Flink 任务运行过程中，如果被订阅的 topic 创建了新的 partition，FlinkKafkaConsumer 如何实现动态发现新创建的 partition 并消费呢？

在使用 FlinkKafkaConsumer 时，可以开启 partition 的动态发现。通过 Properties指定参数开启（单位是毫秒）：

FlinkKafkaConsumerBase.KEY\_PARTITION\_DISCOVERY\_INTERVAL\_MILLIS

该参数表示间隔多久检测一次是否有新创建的 partition。默认值是Long的最小值，表示不开启，大于0表示开启。开启时会启动一个线程根据传入的interval定期获取Kafka最新的元数据，新 partition 对应的那一个 subtask 会自动发现并从earliest 位置开始消费，新创建的 partition 对其他 subtask 并不会产生影响。

代码如下所示：

properties.setProperty(FlinkKafkaConsumerBase.KEY\_PARTITION\_DISCOVERY\_INTERVAL\_MILLIS, 30 \* 1000 + "");

## 4.2 从Kafka数据源生成watermark

Kafka单分区内有序，多分区间无序。在这种情况下，可以使用 Flink 中可识别 Kafka 分区的 watermark 生成机制。使用此特性，将在 Kafka 消费端内部针对每个 Kafka 分区生成 watermark，并且不同分区 watermark 的合并方式与在数据流 shuffle 时的合并方式相同。

在单分区内有序的情况下，使用时间戳单调递增按分区生成的 watermark 将生成完美的全局 watermark。

可以不使用 TimestampAssigner，直接用 Kafka 记录自身的时间戳：

StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

Properties properties = new Properties();

properties.setProperty("bootstrap.servers", "hadoop1:9092,hadoop2:9092,hadoop3:9092");

properties.setProperty("group.id", "fffffffffff");

FlinkKafkaConsumer<String> kafkaSourceFunction = new FlinkKafkaConsumer<>(

"flinktest",

new SimpleStringSchema(),

properties

);

kafkaSourceFunction.assignTimestampsAndWatermarks(

WatermarkStrategy

.forBoundedOutOfOrderness(Duration.ofMinutes(2))

);

env.addSource(kafkaSourceFunction)

## 4.3 设置空闲等待

如果数据源中的某一个分区/分片在一段时间内未发送事件数据，则意味着 WatermarkGenerator 也不会获得任何新数据去生成 watermark。我们称这类数据源为空闲输入或空闲源。在这种情况下，当某些其他分区仍然发送事件数据的时候就会出现问题。比如Kafka的Topic中，由于某些原因，造成个别Partition一直没有新的数据。

由于下游算子 watermark 的计算方式是取所有不同的上游并行数据源 watermark 的最小值，则其 watermark 将不会发生变化，导致窗口、定时器等不会被触发。

为了解决这个问题，你可以使用 WatermarkStrategy 来检测空闲输入并将其标记为空闲状态。

StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

Properties properties = new Properties();

properties.setProperty("bootstrap.servers", "hadoop1:9092,hadoop2:9092,hadoop3:9092");

properties.setProperty("group.id", "fffffffffff");

FlinkKafkaConsumer<String> kafkaSourceFunction = new FlinkKafkaConsumer<>(

"flinktest",

new SimpleStringSchema(),

properties

);

kafkaSourceFunction.assignTimestampsAndWatermarks(

WatermarkStrategy

.forBoundedOutOfOrderness(Duration.ofMinutes(2))

.withIdleness(Duration.ofMinutes(5))

);

env.addSource(kafkaSourceFunction)

## 4.4 Kafka的offset消费策略

FlinkKafkaConsumer可以调用以下API，注意与”auto.offset.reset”区分开：

* setStartFromGroupOffsets()：默认消费策略，默认读取上次保存的offset信息，如果是应用第一次启动，读取不到上次的offset信息，则会根据这个参数auto.offset.reset的值来进行消费数据。建议使用这个。
* setStartFromEarliest()：从最早的数据开始进行消费，忽略存储的offset信息
* setStartFromLatest()：从最新的数据进行消费，忽略存储的offset信息
* setStartFromSpecificOffsets(Map)：从指定位置进行消费
* setStartFromTimestamp(long)：从topic中指定的时间点开始消费，指定时间点之前的数据忽略
* 当checkpoint机制开启的时候，KafkaConsumer会定期把kafka的offset信息还有其他operator的状态信息一块保存起来。当job失败重启的时候，Flink会从最近一次的checkpoint中进行恢复数据，重新从保存的offset消费kafka中的数据（也就是说，上面几种策略，只有第一次启动的时候起作用）。
* 为了能够使用支持容错的kafka Consumer，需要开启checkpoint

# 第5章 FlinkSQL调优

FlinkSQL官网配置参数：

<https://ci.apache.org/projects/flink/flink-docs-release-1.12/dev/table/config.html>

## 5.1 Group Aggregate优化

### 5.1.1 开启MiniBatch（提升吞吐）

MiniBatch是微批处理，原理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。

* **MiniBatch默认关闭，开启方式如下:**

// 初始化table environment

TableEnvironment tEnv = ...

// 获取 tableEnv的配置对象

Configuration configuration = tEnv.getConfig().getConfiguration();

// 设置参数：

// 开启miniBatch

configuration.setString("table.exec.mini-batch.enabled", "true");

// 批量输出的间隔时间

configuration.setString("table.exec.mini-batch.allow-latency", "5s");

// 防止OOM设置每个批次最多缓存数据的条数，可以设为2万条

configuration.setString("table.exec.mini-batch.size", "20000");

* FlinkSQL参数配置列表：

<https://ci.apache.org/projects/flink/flink-docs-release-1.12/dev/table/config.html>

* 适用场景

微批处理通过增加延迟换取高吞吐，如果有超低延迟的要求，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。

* **注意事项：**

1）目前，key-value 配置项仅被 Blink planner 支持。

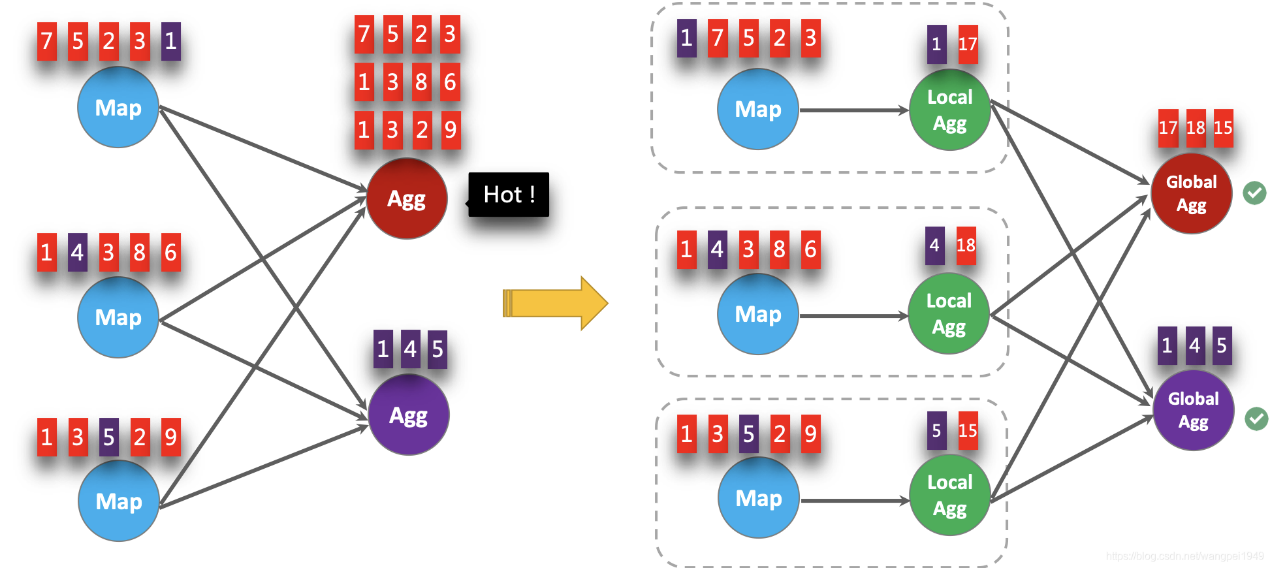
2）1.12之前的版本有bug，开启miniBatch，不会清理过期状态，也就是说如果设置状态的TTL，无法清理过期状态。1.12版本才修复这个问题。

参考ISSUE：<https://issues.apache.org/jira/browse/FLINK-17096>

### 5.1.2 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。结合下图理解LocalGlobal如何解决数据倾斜的问题。



由上图可知：

* 未开启LocalGlobal优化，由于流中的数据倾斜，Key为红色的聚合算子实例需要处理更多的记录，这就导致了热点问题。
* 开启LocalGlobal优化后，先进行本地聚合，再进行全局聚合。可大大减少GlobalAgg的热点，提高性能。
* **LocalGlobal开启方式：**

1）LocalGlobal优化需要先开启MiniBatch，依赖于MiniBatch的参数。

2）table.optimizer.agg-phase-strategy: 聚合策略。默认AUTO，支持参数AUTO、TWO\_PHASE(使用LocalGlobal两阶段聚合)、ONE\_PHASE(仅使用Global一阶段聚合)。

// 初始化table environment

TableEnvironment tEnv = ...

// 获取 tableEnv的配置对象

Configuration configuration = tEnv.getConfig().getConfiguration();

// 设置参数：

// 开启miniBatch

configuration.setString("table.exec.mini-batch.enabled", "true");

// 批量输出的间隔时间

configuration.setString("table.exec.mini-batch.allow-latency", "5 s");

// 防止OOM设置每个批次最多缓存数据的条数，可以设为2万条

configuration.setString("table.exec.mini-batch.size", "20000");

// 开启LocalGlobal

configuration.setString("table.optimizer.agg-phase-strategy", "TWO\_PHASE");

* 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。

* 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。

* **注意事项：**

1）需要先开启MiniBatch

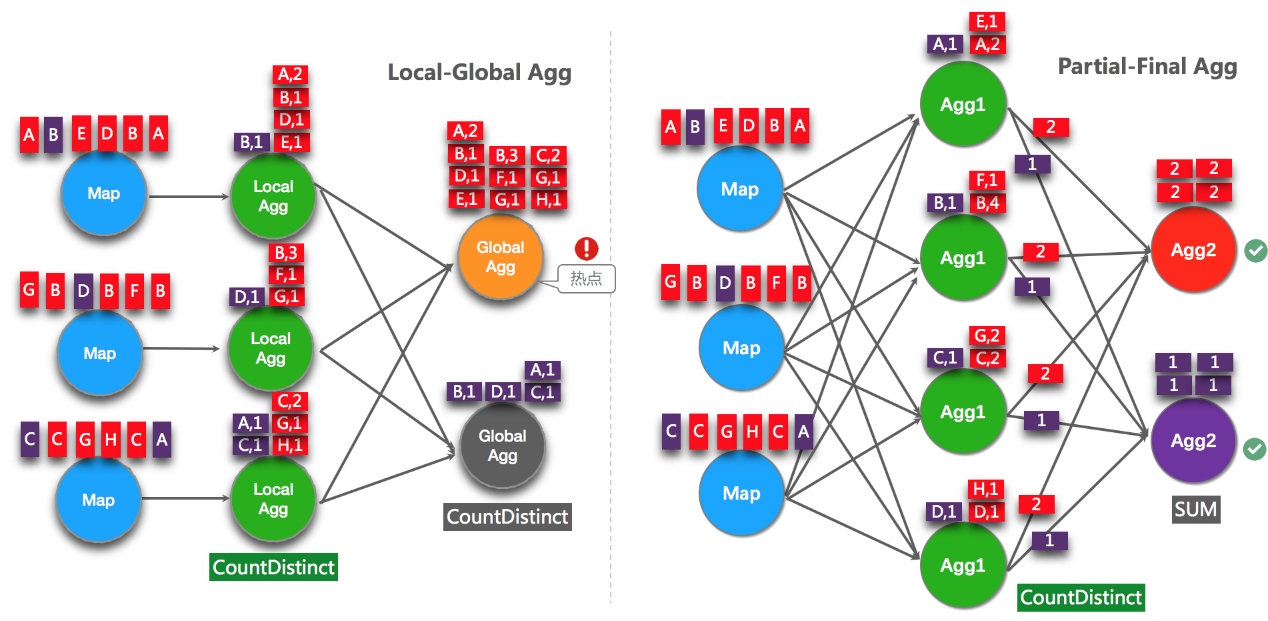
2）开启LocalGlobal需要UDAF实现Merge方法。

### 5.1.3 开启Split Distinct（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）。

从Flink1.9.0版本开始，提供了COUNT DISTINCT自动打散功能，不需要手动重写。Split Distinct和LocalGlobal的原理对比参见下图。



**举例：统计一天的UV**

SELECT day, COUNT(DISTINCT user\_id)

FROM T

GROUP BY day

**如果手动实现两阶段聚合：**

SELECT day, SUM(cnt)

FROM (

SELECT day, COUNT(DISTINCT user\_id) as cnt

FROM T

GROUP BY day, MOD(HASH\_CODE(user\_id), 1024)

)

GROUP BY day

第一层聚合: 将Distinct Key打散求COUNT DISTINCT。

第二层聚合: 对打散去重后的数据进行SUM汇总。

* **Split Distinct开启方式**

默认不开启，使用参数显式开启：

* table.optimizer.distinct-agg.split.enabled: true，默认false。
* table.optimizer.distinct-agg.split.bucket-num: Split Distinct优化在第一层聚合中，被打散的bucket数目。默认1024。

// 初始化table environment

TableEnvironment tEnv = ...

// 获取 tableEnv的配置对象

Configuration configuration = tEnv.getConfig().getConfiguration();

// 设置参数：

// 开启Split Distinct

configuration.setString("table.optimizer.distinct-agg.split.enabled", "true");

// 第一层打散的bucket数目

configuration.setString("table.optimizer.distinct-agg.split.bucket-num", "1024");

* 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。

* 适用场景

使用COUNT DISTINCT，但无法满足聚合节点性能要求。

* **注意事项：**

1）目前不能在包含UDAF的Flink SQL中使用Split Distinct优化方法。

2）拆分出来的两个GROUP聚合还可参与LocalGlobal优化。

3）从Flink1.9.0版本开始，提供了COUNT DISTINCT自动打散功能，不需要手动重写（不用像上面的例子去手动实现）。

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

在某些场景下，可能需要从不同维度来统计UV，如Android中的UV，iPhone中的UV，Web中的UV和总UV，这时，可能会使用如下CASE WHEN语法。

SELECT

day,

COUNT(DISTINCT user\_id) AS total\_uv,

COUNT(DISTINCT CASE WHEN flag IN ('android', 'iphone') THEN user\_id ELSE NULL END) AS app\_uv,

COUNT(DISTINCT CASE WHEN flag IN ('wap', 'other') THEN user\_id ELSE NULL END) AS web\_uv

FROM T

GROUP BY day

在这种情况下，建议使用FILTER语法, 目前的Flink SQL优化器可以识别同一唯一键上的不同FILTER参数。如，在上面的示例中，三个COUNT DISTINCT都作用在user\_id列上。此时，经过优化器识别后，Flink可以只使用一个共享状态实例，而不是三个状态实例，可减少状态的大小和对状态的访问。

**将上边的CASE WHEN替换成FILTER后**，如下所示:

SELECT

day,

COUNT(DISTINCT user\_id) AS total\_uv,

COUNT(DISTINCT user\_id) FILTER (WHERE flag IN ('android', 'iphone')) AS app\_uv,

COUNT(DISTINCT user\_id) FILTER (WHERE flag IN ('wap', 'other')) AS web\_uv

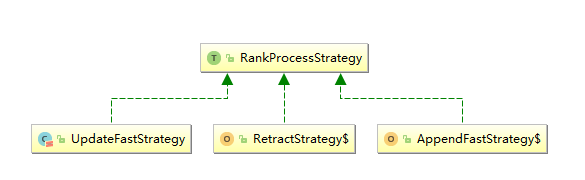
FROM T

GROUP BY day

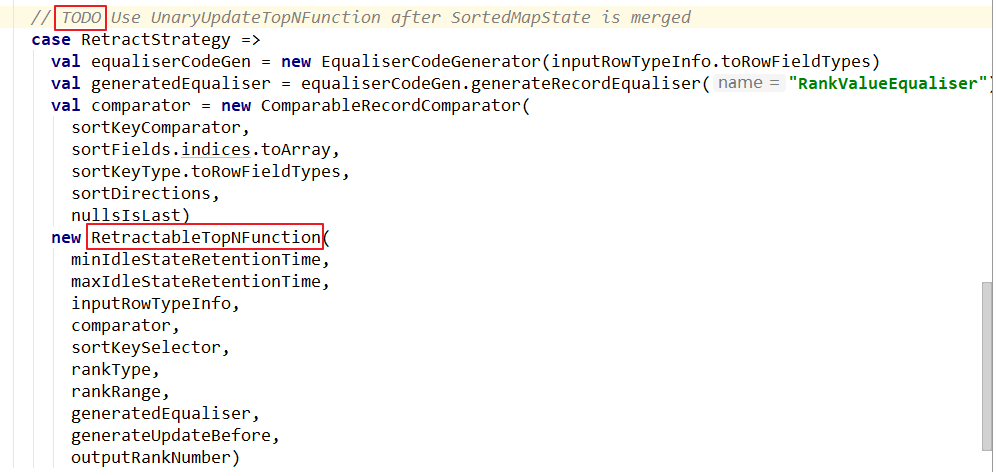
## 5.2 TopN优化

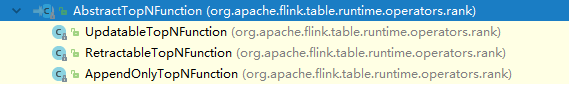
### 5.2.1 使用最优算法

当TopN的输出是非更新流（例如Source），TopN只有一种算法AppendRank。当TopN的输出是更新流时（例如经过了AGG/JOIN计算），TopN有2种算法，性能从高到低分别是：UpdateFastRank 和RetractRank。算法名字会显示在拓扑图的节点名字上。



注意：apache社区版的Flink1.12目前还没有UnaryUpdateRank，阿里云实时计算版Flink才有





* **UpdateFastRank ：最优算法**

需要具备2个条件：

1）输入流有PK（Primary Key）信息，例如Group BY AVG。

2）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。

如果要获取到优化Plan，则您需要在使用ORDER BY SUM DESC时，添加SUM为正数的过滤条件。

* **AppendFast：结果只追加，不更新**
* **RetractRank：普通算法，性能差**

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UpdateFastRank优化。

### 5.2.2 无排名优化（解决数据膨胀问题）

* **TopN语法：**

SELECT \*

FROM (

SELECT \*,

ROW\_NUMBER() OVER ([PARTITION BY col1[, col2..]]

ORDER BY col1 [asc|desc][, col2 [asc|desc]...]) AS rownum

FROM table\_name)

WHERE rownum <= N [AND conditions]

* **数据膨胀问题：**

根据TopN的语法，rownum字段会作为结果表的主键字段之一写入结果表。但是这可能导致数据膨胀的问题。例如，收到一条原排名9的更新数据，更新后排名上升到1，则从1到9的数据排名都发生变化了，需要将这些数据作为更新都写入结果表。这样就产生了数据膨胀，导致结果表因为收到了太多的数据而降低更新速度。

* **使用方式**

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。只需要在外层查询中将rownum字段裁剪掉即可

// 最外层的字段，不写 rownum

SELECT col1, col2, col3

FROM (

SELECT col1, col2, col3

ROW\_NUMBER() OVER ([PARTITION BY col1[, col2..]]

ORDER BY col1 [asc|desc][, col2 [asc|desc]...]) AS rownum

FROM table\_name)

WHERE rownum <= N [AND conditions]

在无rownum的场景中，对于结果表主键的定义需要特别小心。如果定义有误，会直接导致TopN结果的不正确。 无rownum场景中，主键应为TopN上游GROUP BY节点的KEY列表。

### 5.2.3 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。

* **使用方式**

// 初始化table environment

TableEnvironment tEnv = ...

// 获取 tableEnv的配置对象

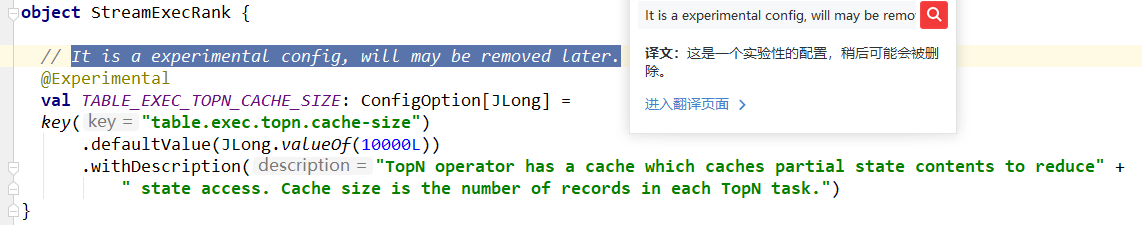
Configuration configuration = tEnv.getConfig().getConfiguration();

// 设置参数：

// 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%

configuration.setString("table.exec.topn.cache-size", "200000");

注意：目前源码中标记为实验项，官网中未列出该参数



### 5.2.4 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。

### 5.2.5 优化后的SQL示例

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（无排名优化）

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（分区字段优化）

ORDER

BY pay\_ord\_amt DESC --根据上游sum结果排序。排序字段的更新是单调的，且单调方向与排序方向相反（走最优算法）

) as rownum

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能使用优化算法，只获取前100个数据（走最优算法）

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

);

## 5.3 高效去重方案

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。

### 5.3.1 保留首行的去重策略（Deduplicate Keep FirstRow）

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;

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。

### 5.3.2 保留末行的去重策略（Deduplicate Keep LastRow）

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;

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。

## 5.4 高效的内置函数

### 5.4.1 使用内置函数替换自定义函数

Flink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。使用内置函数好处：

1）优化数据序列化和反序列化的耗时。

2）新增直接对字节单位进行操作的功能。

支持的系统内置函数：

<https://ci.apache.org/projects/flink/flink-docs-release-1.12/dev/table/functions/systemFunctions.html>

### 5.4.2 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。

### 5.4.3 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

## 5.5 指定时区

本地时区定义了当前会话时区id。当本地时区的时间戳进行转换时使用。在内部，带有本地时区的时间戳总是以UTC时区表示。但是，当转换为不包含时区的数据类型时(例如TIMESTAMP, TIME或简单的STRING)，会话时区在转换期间被使用。为了避免时区错乱的问题，可以参数指定时区。

// 初始化table environment

TableEnvironment tEnv = ...

// 获取 tableEnv的配置对象

Configuration configuration = tEnv.getConfig().getConfiguration();

// 设置参数：

// 指定时区

configuration.setString("table.local-time-zone", "Asia/Shanghai");

## 5.6 设置参数总结

总结以上的调优参数，代码如下：

// 初始化table environment

TableEnvironment tEnv = ...

// 获取 tableEnv的配置对象

Configuration configuration = tEnv.getConfig().getConfiguration();

// 设置参数：

// 开启miniBatch

configuration.setString("table.exec.mini-batch.enabled", "true");

// 批量输出的间隔时间

configuration.setString("table.exec.mini-batch.allow-latency", "5 s");

// 防止OOM设置每个批次最多缓存数据的条数，可以设为2万条

configuration.setString("table.exec.mini-batch.size", "20000");

// 开启LocalGlobal

configuration.setString("table.optimizer.agg-phase-strategy", "TWO\_PHASE");

// 开启Split Distinct

configuration.setString("table.optimizer.distinct-agg.split.enabled", "true");

// 第一层打散的bucket数目

configuration.setString("table.optimizer.distinct-agg.split.bucket-num", "1024");

// TopN 的缓存条数

configuration.setString("table.exec.topn.cache-size", "200000");

// 指定时区

configuration.setString("table.local-time-zone", "Asia/Shanghai");