Spark RDD 운영 원리: Stage 구분과 실행 과정

Slide 1: 阶段的划分

Slide 1: 阶段的划分

Slide 2: Stage 구분의 기준

그럼 Spark는 무엇을 기준으로 Stage를 나눌까요?

답은 의존 관계입니다.

구체적으로는 RDD 간의 의존 관계가 다음 중 무엇인지를 봅니다:

- **Narrow Dependency (좁은 의존)**
- **Wide Dependency (넓은 의존)**

Slide 2: 划分阶段的依据

那么，Spark 到底是根据什么来划分阶段的呢？

答案就在这里：依赖关系。

具体来说，就是看 RDD 之间的依赖是窄依赖还是宽依赖(**zhǎi yīlài háishì kuān yīlài)**。"

Slide 3: DAG와 Stage

Spark는 프로그램 내의 변환 연산(Transformation)을 기반으로 **DAG(Directed Acyclic Graph, 방향성 비순환 그래프)**를 생성합니다.

그런 다음 앞서 말한 의존 관계를 기준으로 이 큰 그래프를 여러 개의 **Stage**로 나누게 됩니다.

Slide 3: DAG与阶段

Spark 会根据程序里的转换操作生成一个叫 DAG 的有向无环图。

然后呢，就会根据刚才说的依赖关系，把这个大图切分成一个一个的阶段。"

Slide 4: 최적화 원리 (Fork/Join)

왜 이렇게 구분할까요?

최적화가 주된 목적입니다.

Narrow Dependency

- 파이프라이닝 최적화가 가능합니다
- 공장 컨베이어 벨트처럼 차례로 빠르게 처리할 수 있습니다

Wide Dependency

- 모든 데이터가 준비될 때까지 기다려야 합니다
- 마치 교통 정체와 같아서 이런 최적화를 할 수 없습니다

Slide 4: 优化原理 (Fork/Join)

为什么要这么划分呢？

主要是为了优化。

- 窄依赖可以进行流水线优化，就像工厂的流水线一样，一个接一个快速处理。
- 而宽依赖不行，它需要等所有数据都准备好，有点像交通堵塞(**dǔ sè)**，所以不能进行这种优化。"

Slide 5-6: Fork/Join 메커니즘 도해

이면의 최적화 아이디어는 **Fork/Join 메커니즘**과 유사합니다.

예시:

- RDD1에 3개의 파티션이 있습니다
- 여러 작업을 'Fork'하여 데이터를 병렬로 처리합니다
- RDD2의 파티션을 생성합니다
- 최종적으로 결과를 얻기 위해 'Join' 합니다

결과: 계산 속도를 크게 높일 수 있습니다.

Slide 5-6: Fork/Join 机制示意图

这背后的优化思想类似于 Fork/Join 机制。

大家看这个图，

① 比如 RDD1 有三个分区，

② 它可以'fork'出多个任务，

③ 让这些任务并行地去处理数据，生成 RDD2 的分区，

④ 最后再'join'在一起得到结果。

这样可以大大提高计算速度。"

Slide 7: 전교 학생 비유

이해를 돕기 위한 비유

'전교 학생'이 데이터라고 가정해봅시다. 북경과 하문의 학생들을 각각 목적지로 보내야 합니다.

원시적인 '비효율적인 방법'

상황: 전교 학생이 베이징에서 샤먼으로 이동해야 합니다.

방식: 모든 사람이 한 줄로 서서 한 명씩 차례로 이동합니다.

문제점: 속도가 매우 느리고 효율이 극히 낮습니다. 마치 프로그램을 작성할 때 단일 스레드만 사용하여 모든 작업을 순차적으로 실행하는 것과 같습니다.

Slide 7: 全校学生示例 (比喻)

我打个比方帮助大家理解。

假设'全校学生'是数据。我们需要把北京和厦门的学生分别送到目的地。

原始的"笨办法"

场景：全校学生要从北京去厦门。

做法：所有人排成一队，一个一个地走。

问题：速度极慢，效率极低。就像我们写程序时，只用了一个线程顺序执行所有任务。

Slide 8: 초기 최적화 - "병렬" 처리 도입

시나리오: 동일한 작업이지만, 이번에는 '반장'과 '조직위원' 두 담당자를 동원했습니다.

수행 방식:

1. **Fork (작업 분할):** 전체 학생(모든 데이터)을 '남학생'과 '여학생' 두 그룹(두 데이터 파티션)으로 나눕니다.
2. **병렬 실행:** 반장은 남학생을, 조직위원은 여학생을 담당하여 **동시에 출발**시켜, 각각 상하이와 샤먼을 경유하게 합니다.

개선점: 작업이 분할되어 병렬로 실행됨에 따라 **총 소요 시간이 단축**되었습니다.

새로운 문제점: 그림에 "1시간 대기"가 표시되어 있습니다. 이는 두 병렬 작업의 진행 속도가 일치하지 않아, 하나의 작업이 끝난 후 다른 작업이 끝날 때까지 기다려야 하기 때문에 발생합니다. Spark에서 이러한 현상은 일반적으로 **데이터 치우침** 또는 **작업 스케줄링**이 충분히 최적화되지 않았을 때 발생합니다.

Slide 8: 初步优化 - 引入"并行"

场景：同样的任务，但这次我们动用了"班长"和"组织委员"两个负责人。

做法：

Fork（分任务）：把全校学生（全部数据）分成"男生"和"女生"两组（两个数据分区）。

并行执行：班长带男生，组织委员带女生，同时出发，分别经停上海和厦门。

进步：任务被拆分并行执行了，总时间变短了。

新问题：图中出现了"等待1小时"，这是因为两个并行任务的进度不一致，其中一个完成后必须等待另一个。在 Spark 中，这通常是因为 数据倾斜 或 任务调度 不够优化导致的。

Slide 9:**최종 최적화 - 완벽한 "병렬" 실행과 "결합"**

시나리오: 최적화된 실행 계획입니다.

수행 방식:

1. **Fork (작업 분할):** 동일하게 작업을 두 부분으로 나눕니다.
2. **병렬 실행:** 하지만 이번에는 두 작업이 각자 독립적으로, 중단 없이 전체 경로(5시간 비행)를 완료합니다.
3. **Join (결과 통합):** 최종적으로, 두 팀이 목적지(하문)에서 완벽하게 합류하여 최종 결과를 만들어냅니다.

결과: 모든 불필요한 대기가 제거되고, 리소스가 최대한 활용되며, **총 실행 시간이 가장 짧아집니다.** 이는 Spark 엔진이 최적화를 통해 가장 합리적인 Stage(단계)를 설계함으로써, 각 작업이 모두各自的 CPU 코어에서 방해받지 않고 끝까지 실행되도록 하는 것과 같습니다.

Slide 9:**最终优化 - 完美的"并行"与"合并"**

场景：最优化的执行方案。

做法：

① Fork（分任务）：同样将任务分成两部分。

② 并行执行：但这次，两个任务各自独立地、不间断地完成全部行程（飞5小时），中间没有任何停顿点。

③ Join（合结果）：最终，两支队伍在目的地（厦门）完美汇合，形成最终结果。

结果：消除了所有不必要的等待，资源被充分利用，总执行时间最短。这就像是 Spark 引擎经过优化后，规划出了最合理的 Stage（阶段），使得每个任务都能在各自的 CPU 核心上毫无阻塞地运行到底。

Slide 10: Stage 구분 실제 예시

RDD Stage 구분 과정

1. **데이터 읽기:** HDFS에서 데이터 읽기
2. **변환 연산:** map, groupByKey, join, union 등
3. **결과 저장:** Action 연산으로 HDFS에 쓰기

Stage 구분 방법

Spark의 역방향 추적:

- 마지막 Action 연산에서 시작
- DAG 그래프를 역방향으로 추적

경계 설정:

- **Wide Dependency 발견 시** (groupByKey, join 등 - Shuffle 필요)
    - 해당 위치에 선을 그어 Stage의 경계로 설정
- **Narrow Dependency의 경우** (map, union 등)
    - 동일한 Stage 내에 포함
    - 파이프라이닝 최적화 가능

Slide 10: 阶段划分实例

现在我们看一个真正的 RDD 阶段划分例子。

我们从HDFS读取数据，经过像map、groupByKey、join、union这些转换操作，最后通过Action操作把结果写回HDFS。

Spark会从最后的Action操作开始，反向回溯整(**sù zhěng**)个DAG图。当遇到宽依赖（比如groupByKey和join，它们通常需要shuffle），就在这里画一条线，作为一个阶段的边界。

像map和union这种窄依赖，则会被划分在同一个阶段内，可以进行流水线优化。”

Slide 11: Stage 내부 세부사항

Stage 내부 구조

예시: Stage 2

- union 연산 포함
- union은 Narrow Dependency
- 따라서 파티션들을 동일한 Stage 안에서 함께 처리 가능

Slide 12: Shuffle과 디스크

매우 중요한 포인트

Shuffle 발생 = Wide Dependency

Shuffle이 발생하면:

- 데이터는 **반드시 디스크에 기록**되어야 합니다

이유:

- 데이터를 재배치(shuffle)하고 혼합해야 함
- 다음 Stage를 위한 준비 과정
- 디스크는 이러한 중간 결과를 안정적으로 저장 가능

Slide 13:

Slide 11: 阶段内细节

我们仔细看看Stage内部。

像这个Stage 2，里面包含了union操作，这是一个窄依赖(**zhǎi yīlài)**，所以这些分区可以在同一个阶段里一起被处理。

Slide 12: Shuffle与磁盘

这里有一个非常重要的点：

只要发生Shuffle，就意味着有宽依赖，数据就必须被写到磁盘上。

这是为什么？

因为需要把数据重新洗牌(**xǐ pái)**、混合，为下一个阶段做好准备，

而磁盘可以可靠地存储这些中间结果。"

Slide 13: 부모가 하나이면 좁은의존관계, 둘이상이면 넓은 의존관계로 shuffle 필요

Slide 14: 최적화 비교 예시

학생 비유로 다시 살펴보기

왼쪽 - 최적화된 방안 (Narrow Dependency 사상)

- 대기 없음
- 병렬 실행

Slide 15:오른쪽 - Shuffle 도입 (Wide Dependency)

- 모든 팀이 상해에 한번 모여야 함
- 그 후 다시 출발
- **결과:** 추가적인 오버헤드와 시간 발생

Slide 16: Stage 구분 총정리

Stage 구분 과정 요약

Spark의 Stage 구분 방법:

1. **시작점:** DAG 그래프의 말단(Action)에서 시작
2. **역방향 분석:** 거꾸로 추적하며 분석
3. **Narrow Dependency 처리:**
    - 해당 연산을 현재 Stage에 추가
4. **Wide Dependency 처리:**
    - 새로운 Stage 생성

비유: Wide Dependency를 가위처럼 사용해 DAG 그래프를 여러 개의 Stage로 자르는 것과 같습니다.

Slide 14: 优化对比示例

我们再回头看那个学生的例子。

左边是优化后的方案（窄依赖思想），没有等待，并行执行。

Slide 15:右边如果引入了Shuffle（宽依赖），就像突然要求所有队伍都先到上海集合一次，然后再出发，这会增加额外的开销和时间。"(상해에서 1반과 2반으로 다시 나눔 즉 shuffle 발생)

Slide 16: 阶段划分总结

"所以，总结一下阶段的划分过程：

Spark从DAG图的末端开始反向解析，

① 遇到窄依赖就把操作加入当前阶段，

② 遇到宽依赖就生成一个新的阶段。

这样就像用宽依赖作为剪刀，把DAG图剪成了一截一截的阶段。（jiǎn chéng le yī jié yī jié de jiēduàn)"

Slide 17:

Slide 18: Spark 작업 스케줄링

슬라이드 보고 2개의 scheduler가 있다. 한 개는 DAG Scheduler, 다른 한 개는 TaskScheduler

Slide 19: Spark 작업 스케줄링

작업 실행 흐름

1. SparkContext

- 사용자 코드를 받아 DAGScheduler에 전달

2. DAGScheduler

- DAG 그래프를 Stage로 분할
- 각 Stage를 작업 집합으로 구성

3. TaskScheduler

- Stage별 작업 집합을 받음
- 클러스터의 Worker 노드에 분배

4. Executor

- Worker 노드의 프로세스
- 내부 스레드가 실제 작업 수행

실제 작업 수행: Executor 내의 스레드들이 실질적으로 작업을 처리합니다.

Slide 20: 마무리

이상으로 RDD 운영 원리에 관한 주요 내용이었습니다.

감사합니다!

Slide 17: 슬라이드 보고 동작 원리 설명

Slide 18: Spark 작업 스케줄링

슬라이드 보고 2개의 scheduler가 있다. 한 개는 DAG Scheduler, 다른 한 개는 TaskScheduler

Slide 18-19: Spark任务调度

"最后我们看一下任务是怎么调度执行的。

① SparkContext把我们的代码交给DAGScheduler，DAGScheduler负责把DAG图分成阶段。
② 然后TaskScheduler接收这些阶段，并把每个阶段的任务集合分发到集群的
③ Worker节点上的Executor进程中去执行。

真正干活的的是Executor里的线程。"

Slide 20: 结束页

"以上就是关于RDD运行原理的主要内容。谢谢大家！"

핵심 요약

주요 개념

1. **의존 관계 기반 Stage 구분**
    - Narrow Dependency: 파이프라이닝 최적화 가능
    - Wide Dependency: Shuffle 필요, 디스크 I/O 발생
2. **DAG와 Stage**
    - DAG 생성 → Stage 구분 → Task 실행
3. **최적화 원리**
    - Fork/Join 메커니즘으로 병렬 처리
    - Shuffle 최소화로 성능 향상
4. **작업 스케줄링**
    - SparkContext → DAGScheduler → TaskScheduler → Executor