**Jointly Optimizing** **Task Granularity and Concurrency for In-Memory MapReduce Frameworks**

accelerate workloads -> frequent data reuse

effective in-memory caching -> 대부분의 IO operation을 제거. 대신 producer와 consumer task 사이의 통신을 필요로함

memory footprint가 가능한 메모리 사이즈를 넘으면 과도한 spill과 grabage collection 때문에 이점이 무효화됨.

N\_partitions: data partition의 수가 task granularity를 결정

N\_threads: 각 execution당 task의 수가 execution의 parallelism degree를 결정

WASP - workload aware task scheduler and partitioner -> runtime에 N\_partitions랑 N\_threads를 jointly optimize함.

먼저 주어진 workload에 대한 DAG를 분석하고 analytical model로 모든 stage마다 두 파라미터의 optimal setting을 예측.

WASP는 hill climbing algorithm을 이용해 각 stage마다 optimal한(data spill과 GC를 최소화하는 동안 concurrency를 최대화하는) N\_threads를 찾아냄.

3개의 다른 parallel platform에서 6개의 workload로 실험.

WASP는 3.22배 효율. 40%만큼 클러스터의 계산 비용 절감, Spark Tuning Guideline(?)을 베이스로,,,, 그리고 shuffle-heavy & shuffle-light workload 둘 다에 좋음..

I. INTRODUCTION

spark frequent data use에 매우 빠름. 허나 메모리 부족에 performance가 크게 영향을 받음. Spark는 JVM에서 돌기 때문에 RDD도 heap의 캐시에 저장/ 메모리가 거의 다 차면 large number of disk spill이랑 GC를 야기함 -> in-memory의 빠르다는 장점을 없애는 요인!

특히 shuffle operation은 이런 slowdown에 크게 영향을 받음. -> in memory usage의 최적화가 중요~

N\_partitions: 한개의 RDD에서 몇개의 data partition이 생기는지를 나타냄. 이게 너무 낮으면(partition이 너무 적으면) 메모리 압력이 증가해서 과도한 spill과 GC를 발생시킴. 이게 너무 높으면 scheduling이랑 shuffle operation하는데에 significant overhead를 야기한다

N\_threads: 한개의 executor에 몇개의 thread가 배정되어있는지를 나타냄. (thread 한개가 시간에따라 task를 한개씩 처리할때) 이게 너무 낮으면 -> 주어진 element를 underutilization하게 되서 성능이 subobtimal 된다. 너무 높으면 -> 메모리 압력이 증가하고 resource 경합때문에 성능이 저하됨.

즉 두개 파라미터를 하드웨어 resource랑 workload 특징등을 고려해서 optimal 성능을 내기 위한 값을 잘 골라야된다

지금까지는 이런걸 해결할때 workload 특징을 고려하지 않거나, single parameter를 독립시켜 최적화시키거나 했다. 예를 들어 Spark Tuning Guildlines는 두 파라미터를 workload를 고려하지 않고 구하는 방식을 사용. 또한 한개의 프로그램 안에서도 RDD에 따라 optimal parameter값이 다름에도 불구하고 entire 프로그램에서 한 set의 파라미터들을 쓰도록 한다. 이것은 왁벽한 최적화가 아니고 최적의 성능이아님. 두 파라메터가 독립이 아니고 서로 영향을 주기때문~

그래서 이 논문에서는 runtime에 worklload 특징(RDD graph & input data)과 execution 환경을 고려해서 두 파라미터를 jointly optimize하겠다. -> WASP

WASP

1. 먼저 주어진 workload에 대한 DAG를 분석
2. analytical model을 이용해서 각 stage별로 두 파라미터의 optimal setting을 예측
3. 위의 두개를 input으로 해서 GC-aware task scheduler가 runtime에 individual task를 모니터하면서 N\_thread를 optimize.

->data spill과 GC를 최소화하면서 performance를 최적화!~

HiBench의 6개 workload사용. WASP는 Spark Tuning Guidelines에 비해 performance 최대 3.22배. geomean speedup은 4 node with 64 fat cores에서는 1.74배. 1 node 64 thin cores에서는 1.56배 좋다..

contributions

1. in-memory MapReduce frameworks에서 N\_partitions와 N\_thread를 최적화하는 effective analytical model 제시.
2. disk spill과 GC는 최소화하면서 task concurrency(CPU utilization)를 최대화하는 WASP scheduler를 Spark위에서 돌도록 디자인.
3. 두개의 parallel platform(4 node 64 fat core, 1 node 64 thin core)와 한개의 virtual platform(64 node 256 fat core)에서 실험돌려봄.

II. Background and Motivation

Spark

spark.memory.fraction = JVM heap space의 75%

나머지 25%는 user memory로 사용

spark memory = storage memory + execution memory (초기에는 반반씩)

storage memory -> RDD caching에 사용.

execution memory -> Spark computation

서로 덜사용되는 부분 있으면 땡겨서 사용할 수 있는 그런것..

spill -> Spark의 executor memory가 부족하면 일어남. spill이 일어나면 부분적으로 생긴 output RDD를 serialize하고 disk에 씀.

색깔이 진할수록 speedup이 많이 된 것.

terasort & bayes -> baseline configuration이 검은별, best performance가 흰별

두 workload의 특성이 다름에 따라 흰별의 위치가 바뀜,

Terasort의 경우 baseline configuration 사용하면 memory spill이 많이 일어남.

Terasort의 reduce stage(stage2)가 slowdown의 가장 큰 이유라고함.. 이거를 N\_partitions를 128~4096까지 바꿔가면서 어떻게 바뀌는지에 대한 그래프.

N\_partitions가 1024보다 작으면 execution memory가 부족해서 spill overhead가 execution time의 대부분이 됨. spill time의 대부분은 Full GC>spill to Disk>Young GC 순서.

spill operation은 serializaion & disk write 가 대부분의 시간을 차지하지만, spilled object는 JVM의 old-generation heap 영역에 있어서 spill operation이 완전히 끝날 때까지 free 되지 못하기 때문이다…Spark가 계속 작은 heap object를 만들고 그 중 몇은 young-generational heap 에서 old-generational heap으로 옮겨지고 frequent full GC를 야기하게 된다는 것...때문에 N\_partitions를 1024보다 크게 둬서 spill이 안일어나도록 하는것이 필요하다

또한 둘을 독립적으로 optimize하는 것은 suboptimal performance를 내게 된다. 사진에서 보면 terasort는 N\_partitions를 optimize하는게 낫고, bayes는 N\_threads를 optimize하는게 나음. 둘 중 한개만 optimize해서는 최적의 성능이 아니게된다

III. Workload-Aware task Schedulart and Partitioner

A. 오버뷰

첫번째 box는 workload와 platform parameter를 가지고 optimal한 N\_partitions랑 N\_threads를 예측하는 analytic model임. 새로운 job이 생길때마다 실행되고 task execution이 시작하기 전에 RDD graph에 예측된 값을 주석으로 남김.

두번째 box는 GC-aware task scheduler. N\_threads를 task execution을 하는 동안 optimize하는 역할. master node랑 worker node사이에 feedback loop 이 있어서 과도한 GC를 일으키지 않으면서 동시에 돌릴 수 있는 최대 worker thread의 수를 찾음.

B. analytic model

새로운 job이 생기면 Spark는 RDD graph를 만들고 WASP는 analytical model을 실행해서 매 stage마다 DAG scheduler를 실행하기 전에 N\_partitions와 N\_threads를 optimal하게 setting함.

2 pass approach

1. first pass

N\_partitions 와 N\_threads의 초기값을 세팅. second pass의 input으로 사용됨.

N\_partitions = total number of partitions

N\_threads = number of threads per executor

RDD size k = k번째 stage의 RDD graph의 lineage에서 가장 큰 RDD의 size

Execution memory = 한개의 executor에게 사용가능한 execution memory의 size

max의 앞에값은 각 task가 maximum RDD partition size만큼 execution memory가 필요하다고 생각할때 spill 없이 몇개의 task를 할당할 수 있는지를 계산하는것.

N threads(k)(init) = max(Execution memory/(RDDsizek/N\_partitions(k)(init)), 1)

MAF = Memory amplification factor; transformation function별로 input RDDsize에 비해 output RDDsize의 비율. map()이나 flatmap()의 경우 다른 function과 다르게 input size가 거의 비슷해도 output size가 매우 다양하게 나옴.

RDD size k -> 구하는 식

Input size k -> 구하는 식

runtime에 이걸 구하느라 드는 시간이; re-partitioning input RDD를 하는 오버해드보다 적음.

2. second pass

analytical model을 사용해 stage execution time동안 2차원 파라메터 공간을 gradient search algorithm으로 돌도록 함. first pass에서 구한 init얘네들이 starting point. second pass를 통해 각 stage마다의 optimal한 파라메터값을 얻게 됨.

(4a) Exectime은 3가지 값의 곱에 비례한다..

* RDDsizek/N\_partitions(k) = task의 granularity(입자성, 세분성)를 나타냄.. k번째 stage에서 partition당 size.
* 알파 k = spill의 cost. (5)보면 분자가 분모의 execution memory를 넘어서게 되면 data spill이 일어나 분자/분모 의 비율로 execution time이 증가하게 될 것.
* N\_partitions(k)/N\_threads(k) = 각 round가 N\_threads개의 task를 완성하는 것 (thread가 N\_threads개 있으니까 한round에 저만큼 가능)이라고 가정할 때, k번째 stage동안 몇번의 round가 필요한지.

(N partitions(k)(init) , N threads(k)(init) ) 에서 시작해서 8개의 이웃의 execution time을 비교해서 최소의 execution time인 애 주변 8개를 또 보는 그런....

(N partitions(k)(init) /2, N threads(k)(init) /2)부터 (N partitions(k)(init) ×2, N threads(k)(init) ×2) 까지 bounded

맨처음 시작했던 애의 exectime에 비해 10%이상 작은 넘을 찾아내면 멈춤.

N\_partitions가 커질수록 shuffle의 cost가 커지기 때문에 GC랑 spill이 없는 가장작은 N\_partitions를 고름. (5)번 식 보면 N\_partitions의 upper bound (RDD\*N\_thread/exec\_mem)을 둬서 너무 커지지 않도록 함.

C. Garabage Collection Aware Task Scheduler

stage별로 예측된 두 파라메터값들은 DAG scheduler로 넘겨짐. DAG scheduler -> Spark job을 task의 stage들로 partitioning하는 역할. 그러면 task set이 task scheduler로 넘겨지고 executor들이 parallel하게 여러 task를 실행하기 시작한다아

그러나! 앞에서까지 예측한 두 값들은 runtime 환경에 따라 optimal값들이 변하게 되어있슴. 이것을 처리해주자!

고러나 runtime에 N\_partitions를 바꾸는 것은 RDD 전체를 re-partitioning해야 되는 비용이 든다.. 그래서 N\_threads를 바꾸는게 낫다는 말인거같은데 맞나봄

WASP는 Spark에서 제공하는 thread pool을 이용해 thread의 수를 쉽게 조정할 수 있고 이는 concurrent task간에 data 의존성이 없기 때문에 가능.

아무튼 그래서 WASP는 master랑 worker node사이에 feedback loop을 둔다.. master node의 task scheduler는 GC operation에 걸리는 시간을 모니터한다. 만약 GC에 걸리는 시간이 threshold(total execution time의 20%) 보다 적으면 N\_threads는 upper-bound(core의 총 수의 2배)까지 2배씩 증가됨.

만약 GC에 걸리는 시간이 threshold보다 커지면(혹은 execution time이 직전보다 20%이상 늘어나면, straggler task - 엄청 오래걸리는 task - 를 detect하기 위해서라는데...) N\_thread는 다음 round에 1작아짐. GC가 threshold보다 낮아질때까지 계속 감소.

IV. Methodology

3가지 환경 사용. Native cluster, Knights landing(KNL), Amazon Web Services EC2..

1. Native cluster

4-node cluster with 64 fat cores.

single node는 2개의 core -> 8 physical cores, 16 logical cores(Hyper threading)

4 executor/node, 4 physical cores/executor

1. Knights landing(KNL)

64 thin cores.

64 physical cores, 256 logical cores(4 way Hyper threading)

single node, many core system.

1. Amazon Web Services EC2

64-node cluster.

1 executor/node -> total 64 executor, 4 thread/executor with 10GB of Spark memory

Intel의 HiBench suite 중에 6개 사용. 총 10개 중에 Join, Scan, Aggregate는 SqlContext대신 HiveContext를 SQL query를 실행하는데 사용하기 때문에 제외함. (이유 잘 모르게씀..)

또, Sleep은 real computation이 없는 micro-benchmark라서 제외. 나머지 6개.

Shuffled-heavy와 shuffled-light로 나눔!

KNL task failure…?(모르겠음)

WASP를 4가지 design에 따라 비교해보게씀

1. baseline

N\_threads는 가능한 physical cores를 전부 사용하도록 하고, N\_partitions는 2 tasks per 1 CPU core 로 설정..(total threads의 두배)

1. 2-D exhaustive N\_threads

N\_partitions는 baseline의 값으로 고정시키고, (stage, N\_threads)에 대해 완전탐색을 해봄. N\_threads는 독립적으로 이론적으로 max performance인 것이 찾아짐.

1. 2-D exhaustive N\_partitions

N\_threads는 baseline의 값으로 고정시키고, (stage, N\_partitions)에 대해 완전탐색을 해봄. N\_partitions는 독립적으로 이론적 max performance.

1. 3-D optimal

(stage, N\_partitions, N\_threads)에 대해 완전 탐색을 해봄.

V.Evaluation

1. Native cluster (4-node cluster with 64 fat cores)

5가지 design; baseline, 2-D N\_threads, 2-D N\_partitions, WASP, 3-D optimal

2-D 두개 보다 WASP의 perform이 나은 걸 보아 isolation한 optimization은 한계가 있다.

* shuffled heavy

stage사이에 data 교환량이 많은 workload. TeraSort랑 Sort랑 둘다 sortByKey라는 transformation하는데, 이건 전체 RDD를 shuffle하는 것임. PageRank의 경우는 flatmap이랑 map이라는 transformation을 하는데, 요게 shuffle이 선행되야 하는 것들이다.

큰 메모리를 사용하기 때문에 N\_partitions에 더 sensitive하다. 그래서 N\_threads만 optimize한 건 poor한 결과가 나옴.

geomean을 보면 N\_partition은 2.36배인것에 비해 cost가 높은 exhaustive탐색을 하지 않고도 WASP는 2.43배나옴.

그럼 머땜에 속도가 증가했을까아~ 첫번째 이유는 long GC의 대부분의 이유인 spilled data의 양이 줄기 때문! Terasort의 경우 GC time의 91%가 줄음. 또한 compute time도 37%정도 감소했는데, 이는 GC time을 최소로 하면서 N\_thread를 maximize해서 task 병렬성을 높였기 때문

* shuffled light

얘네한테는 shuffle에 대한 overhead가 별로 크지 않음. 이들은 spill overhead가 작으며 N\_threads에 더 sensitive한 애들임.

2-D N\_threads 1.21배 인데, WASP는 1.25배임. 3-D는 1.31배.

2. Knights landing(KNL) (1node 64 thin cores)

TeraSort와 Sort에서는 baseline(128)값을 그대로 쓰면 task failure가 나서 더 높은 N\_partitions값을 사용했다. Terasort-288/Sort-224

위에랑 똑같이 shuffled heavy에서는 N\_partitions가 더 sensitive하고 shuffled light에서는 N\_threads가 더 sensitive함.

3. Amazon Web Services EC2 (64-node cluster.)\

환경이 더 larger scale에서 돌리는것.

Amazon에서 charge된 bill로 improved 된 performance라는 걸 확인했다는 뜻인가?

암튼 그래서 speedup과 cost reduction을 해냈다..

4. Performance Analysis

**spill-aware scheduler vs GC-aware scheduler**

WASP는 둘중 GC-aware를 선택한 것이다. 과도한 GC가 high memory usage에서의 Spark application slow down의 가장 주된 요소이기 때문에, 또, 적은 양의 spill은 maximize task concurrency에서는 커버가 가능하기 때문에. spill-aware scheduler는 N\_threads에 너무 보수적(적당히 봐줘도 되는걸 제한한다는 뜻이네)이여서 performance를 완벽히 optimal하지 못한다.

spill-aware scheduler는 첫 spill이 발생하면 N\_threads값을 보수적으로 제한한다. WASP는 그와 다르게 spill이 일어나도 GC time의 overhead가 일정 threshold에 도달할 때까진 N\_threads값을 증가시킨다. 실험결과를 통해 GC time이 spill의 양보다 memory pressure를 추정하기에 나은 값이라는 것을 알 수 있다.

**Sensitivity on GC threshold**

GC threshold를 너무 낮게 잡으면 N\_threads값에 너무 보수적이게 되고, 너무 높게 잡으면 GC overhead가 너무 커지게 된다. 그러므로 GC threadhold값을 어케 선택할지에 대한 실험도 진행함. 10%~50%중에 실험해본 결과 GC threshold가 20%일때가 가장 적당한 값이다~

**Impact on CPU utilization and GC time**

terasort - baseline의 경우 CPU utilization이 높은 편인데, GC비율도 높았었다. WASP를 통해 GC 시간을 엄청 줄여서, CPU cycle을 GC가 아닌 useful work에 사용할 수 있도록 했다.

bayes - shuffled light workload로 GC가 매우 적은 편. 그러므로 이런 경우는 GC를 줄이는 것 보다 N\_threads를 optimize해서 CPU utilization을 높이는 것이 더 중요함. WASP는 CPU utilization을 50%->80%로 증가!

VI. Related Work

1. Guidelines for Spark performance tuning

Spark tuning guidline은 다 좋은데, workload랑 platform 특징을 고려를 안하고 parameter들을 결정했다.

- HPC(휴대용 PC) 환경에서 parameter setting이 Spark performance에 영향을 미치는 정도를 두 benchmark(Kmeans, sortByKey)에서 in-depth performance 분석을 진행.

- Shuffling, compression, serialization을 위한 tunable Spark parameter값들을 제시했다

어쨌든 둘다 WASP와는 다르게 runtime behavior(spill과 GC양)을 고려를 못했고, stage마다가 아닌 전체 application에 한개의 set을 제시했다.

1. Fine-grained tuning of N\_threads

각각의 stage마다 dynamically N\_threads를 결정하는 prediction-based dynamic SMT threading(PBDST) framework에 대한 연구… 그들의 N\_threads prediction model은 microarchitectural event(cache miss rate, branch miss rate 등)을 input으로 하지만, software-level event(GC, spill, straggler task등)을 고려하는데에 실패했다.

ㅇ와 이런거까지 알아야될까… 여긴안읽어도될꺼같아

1. Fine-grained tuning of N\_partitions
2. General-purpose N\_threads tuning frameworks

VII. Conclusion

WASP - workload aware task scheduler & jointly optimize N-threads and N-partitions

Spark job이 생성되면 analytic model을 실행해서 모든 stage에 대한 optimal한 N\_threads와 N\_partitions값을 예측함. GC-aware task scheduler는 이 값을 input으로 CPU utilization을 최대로 하면서 spill과 GC를 최소로 하는 최적의 N\_threads값을 업데이트 하도록한다. WASP는 여러 환경에서 baseline에 비해 좋은 성능을 낸다는 실험결과~