Spark의 핵심은 무엇일까?

Resilient Distributed Datasets

A Fault-Tolerant Abstraction for In Memory Cluster Computing

하용호

yongho.ha@gmail.com


하
용
호

요즘 뜬다 하는

요즘 Spark이라고 하는 분산 프레임웍이 뜨고 있죠.
일부에서는 Hadoop은 이미 Legacy이다 라는 표현까지 나올 정도입니다.

Spark의 근간을 이루는 두 논문

Spark: Cluster Computing with Working Sets

Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica
University of California, Berkeley

Abstract

MapReduce and its variants have been highly successful in implementing large-scale data-intensive applications on commodity clusters. However, most of these systems are built around an acyclic data flow model that is not suitable for other popular applications. This paper focuses on one such class of applications: those that reuse a working set of data across multiple parallel operations. This includes many iterative machine learning algorithms, as well as interactive data analysis tools. We propose a new framework called Spark that supports these applications while retaining the scalability and fault tolerance of MapReduce. To achieve these goals, Spark introduces an abstraction called resilient distributed datasets (RDDs). An RDD is a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Spark can outperform Hadoop by 10x in iterative machine learning jobs, and can be used to interactively query a 39 GB dataset with sub-second response time.

1 Introduction

A new model of cluster computing has become widely popular, in which data-parallel computations are executed on clusters of unreliable machines by systems that automatically provide locality-aware scheduling, fault tolerance, and load balancing. MapReduce [11] pioneered this model, while systems like Dryad [17] and Map-Reduce-Merge [24] generalized the types of data flows supported. These systems achieve their scalability and fault tolerance by providing a programming model where the user creates acyclic data flow graphs to pass input data through a set of operators. This allows the underlying system to manage scheduling and to react to faults without user intervention.

While this data flow programming model is useful for a large class of applications, there are applications that cannot be expressed efficiently as acyclic data flows. In this paper, we focus on one such class of applications: those that reuse a *working set* of data across multiple parallel operations. This includes two use cases where we have seen Hadoop users report that MapReduce is deficient:

- **Iterative jobs**: Many common machine learning algorithms apply a function repeatedly to the same dataset to optimize a parameter (e.g., through gradient descent). While each iteration can be expressed as a MapReduce/Dryad job, each job must reload the data from disk, incurring a significant performance penalty.
- **Interactive analytics**: Hadoop is often used to run ad-hoc exploratory queries on large datasets, through SQL interfaces such as Pig [21] and Hive [1]. Ideally, a user would be able to load a dataset of interest into memory across a number of machines and query it repeatedly. However, with Hadoop, each query incurs significant latency (tens of seconds) because it runs as a separate MapReduce job and reads data from disk.

This paper presents a new cluster computing framework called Spark, which supports applications with working sets while providing similar scalability and fault tolerance properties to MapReduce.

The main abstraction in Spark is that of a *resilient distributed dataset* (RDD), which represents a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Users can explicitly cache an RDD in memory across machines and reuse it in multiple MapReduce-like *parallel operations*. RDDs achieve fault tolerance through a notion of *lineage*: if a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to be able to rebuild just that partition. Although RDDs are not a general shared memory abstraction, they represent a sweet-spot between expressivity on the one hand and scalability and reliability on the other hand, and we have found them well-suited for a variety of applications.

Spark is implemented in Scala [5], a statically typed high-level programming language for the Java VM, and exposes a functional programming interface similar to DryadLINQ [25]. In addition, Spark can be used interactively from a modified version of the Scala interpreter, which allows the user to define RDDs, functions, variables and classes and use them in parallel operations on a cluster. We believe that Spark is the first system to allow an efficient, general-purpose programming language to be used interactively to process large datasets on a cluster.

Although our implementation of Spark is still a prototype, early experience with the system is encouraging. We show that Spark can outperform Hadoop by 10x in iterative machine learning workloads and can be used interactively to scan a 39 GB dataset with sub-second latency.

This paper is organized as follows. Section 2 describes

Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica
University of California, Berkeley

Abstract

We present Resilient Distributed Datasets (RDDs), a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner. RDDs are motivated by two types of applications that current computing frameworks handle inefficiently: iterative algorithms and interactive data mining tools. In both cases, keeping data in memory can improve performance by an order of magnitude. To achieve fault tolerance efficiently, RDDs provide a restricted form of shared memory, based on coarse-grained transformations rather than fine-grained updates to shared state. However, we show that RDDs are expressive enough to capture a wide class of computations, including recent specialized programming models for iterative jobs, such as Pregel, and new applications that these models do not capture. We have implemented RDDs in a system called Spark, which we evaluate through a variety of user applications and benchmarks.

1 Introduction

Cluster computing frameworks like MapReduce [10] and Dryad [19] have been widely adopted for large-scale data analytics. These systems let users write parallel computations using a set of high-level operators, without having to worry about work distribution and fault tolerance.

Although current frameworks provide numerous abstractions for accessing a cluster's computational resources, they lack abstractions for leveraging distributed memory. This makes them inefficient for an important class of emerging applications: those that *reuse* intermediate results across multiple computations. Data reuse is common in many *iterative* machine learning and graph algorithms, including PageRank, K-means clustering, and logistic regression. Another compelling use case is *interactive* data mining, where a user runs multiple ad-hoc queries on the same subset of the data. Unfortunately, in most current frameworks, the only way to reuse data between computations (*e.g.*, between two MapReduce jobs) is to write it to an external stable storage system, *e.g.*, a distributed file system. This incurs substantial overheads due to data replication, disk I/O, and serialization, which can dominate application execution times.

Recognizing this problem, researchers have developed specialized frameworks for some applications that require data reuse. For example, Pregel [22] is a system for iterative graph computations that keeps intermediate data in memory, while HaLoop [7] offers an iterative MapReduce interface. However, these frameworks only support specific computation patterns (*e.g.*, looping a series of MapReduce steps), and perform data sharing implicitly for these patterns. They do not provide abstractions for more general reuse, *e.g.*, to let a user load several datasets into memory and run ad-hoc queries across them.

In this paper, we propose a new abstraction called *resilient distributed datasets (RDDs)* that enables efficient data reuse in a broad range of applications. RDDs are fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators.

The main challenge in designing RDDs is defining a programming interface that can provide fault tolerance *efficiently*. Existing abstractions for in-memory storage on clusters, such as distributed shared memory [24], key-value stores [25], databases, and Piccolo [27], offer an interface based on fine-grained updates to mutable state (*e.g.*, cells in a table). With this interface, the only ways to provide fault tolerance are to replicate the data across machines or to log updates across machines. Both approaches are expensive for data-intensive workloads, as they require copying large amounts of data over the cluster network, whose bandwidth is far lower than that of RAM, and they incur substantial storage overhead.

In contrast to these systems, RDDs provide an interface based on *coarse-grained* transformations (*e.g.*, map, filter and join) that apply the same operation to many data items. This allows them to efficiently provide fault tolerance by logging the transformations used to build a dataset (its *lineage*) rather than the actual data.[1] If a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to recompute

[1] Checkpointing the data in some RDDs may be useful when a lineage chain grows large, however, and we discuss how to do it in §5.4.

거칠게 말하자면

Spark = RDD + Interface

RDD를 알아야 Spark을 아는거다.

원전(논문)을 뒤져봅시다.

Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica
University of California, Berkeley

Abstract

We present Resilient Distributed Datasets (RDDs), a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner. RDDs are motivated by two types of applications that current computing frameworks handle inefficiently: iterative algorithms and interactive data mining tools. In both cases, keeping data in memory can improve performance by an order of magnitude. To achieve fault tolerance efficiently, RDDs provide a restricted form of shared memory, based on coarse-grained transformations rather than fine-grained updates to shared state. However, we show that RDDs are expressive enough to capture a wide class of computations, including recent specialized programming models for iterative jobs, such as Pregel, and new applications that these models do not capture. We have implemented RDDs in a system called Spark, which we evaluate through a variety of user applications and benchmarks.

1 Introduction

Cluster computing frameworks like MapReduce [10] and Dryad [19] have been widely adopted for large-scale data analytics. These systems let users write parallel computations using a set of high-level operators, without having to worry about work distribution and fault tolerance.

Although current frameworks provide numerous abstractions for accessing a cluster's computational resources, they lack abstractions for leveraging distributed memory. This makes them inefficient for an important class of emerging applications: those that *reuse* intermediate results across multiple computations. Data reuse is common in many *iterative* machine learning and graph algorithms, including PageRank, K-means clustering, and logistic regression. Another compelling use case is *interactive* data mining, where a user runs multiple ad-hoc queries on the same subset of the data. Unfortunately, in most current frameworks, the only way to reuse data between computations (*e.g.*, between two MapReduce jobs) is to write it to an external stable storage system, *e.g.*, a distributed file system. This incurs substantial overheads due to data replication, disk I/O, and serialization, which can dominate application execution times.

Recognizing this problem, researchers have developed specialized frameworks for some applications that require data reuse. For example, Pregel [22] is a system for iterative graph computations that keeps intermediate data in memory, while HaLoop [7] offers an iterative MapReduce interface. However, these frameworks only support specific computation patterns (*e.g.*, looping a series of MapReduce steps), and perform data sharing implicitly for these patterns. They do not provide abstractions for more general reuse, *e.g.*, to let a user load several datasets into memory and run ad-hoc queries across them.

In this paper, we propose a new abstraction called *resilient distributed datasets (RDDs)* that enables efficient data reuse in a broad range of applications. RDDs are fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators.

The main challenge in designing RDDs is defining a programming interface that can provide fault tolerance *efficiently*. Existing abstractions for in-memory storage on clusters, such as distributed shared memory [24], key-value stores [25], databases, and Piccolo [27], offer an interface based on fine-grained updates to mutable state (*e.g.*, cells in a table). With this interface, the only ways to provide fault tolerance are to replicate the data across machines or to log updates across machines. Both approaches are expensive for data-intensive workloads, as they require copying large amounts of data over the cluster network, whose bandwidth is far lower than that of RAM, and they incur substantial storage overhead.

In contrast to these systems, RDDs provide an interface based on *coarse-grained* transformations (*e.g.*, map, filter and join) that apply the same operation to many data items. This allows them to efficiently provide fault tolerance by logging the transformations used to build a dataset (its *lineage*) rather than the actual data.[1] If a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to recompute

[1]Checkpointing the data in some RDDs may be useful when a lineage chain grows large, however, and we discuss how to do it in §5.4.

- 2012년 NSDI에 발표
- NSDI
  - USENIX Symposium on Networked Systems Design & Implementation
  - 통신과 대형 처리 시스템
  - 다른 곳보다 구현체에 후함.
  - 올해 2014년 – 본류인 네트워크 위주
  - 당시 2012년 – 빅데이터 섹션이 흥함
- RDD -〉 2012 NSDI Best Paper

덤) 올해의 NSDI 재미난 것들

How speedy is SPDY?

Xiao (Sophia) Wang, Aruna Balasubramanian, Arvind Krishnamurthy, David Wetherall

University of Washington

High Throughput Data Center Topology Design

Ankit Singla, P. Brighten Godfrey, Alexandra Kolla

Microsoft Research

Microsoft

FaRM: Fast Remote Memory

Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, Miguel Castro

MICA: A Holistic Approach to Fast In-Memory Key-Value Storage

Hyeontaek Lim[1]

Dongsu Han,[2] David G. Andersen,[1] Michael Kaminsky[3]

[1]Carnegie Mellon University

[2]KAIST, [3]Intel Labs

OK! RDD를 읽어보자.

지식의 원전인 RDD 논문을 읽어봅시다.
Spark홈페이지 가보면 친철하게 링크도 되어 있습니다.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

14장 짜리 페이퍼

생각보다 술술술 읽힘

- 비슷한 계열의 Google File System논문과 비교하면 천지차이
  - GFS는 사골같이 쓰여져 있어 읽을수록 새로운 맛
  - RDD는 라면 같아서 한번에 후루룩 잘 읽힘
- 논문 쓰는 스타일의 차이도 있음
  - GFS = 스탠포드 = 정석
  - RDD = 버클리 = 개념원리

논문 구성은 심플

1. Introduction – 왜 이런 요상한 것을 만들었는가에 대한 설명

2. Resilient Distributed Datasets – RDD라는게 어떤 특성을 가지고 있고, 왜 좋은지에 대해 설명

3. Spark Programming Interface – RDD로 일반 문제를 풀 수 있음을 예제로 보임

4. Representing RDD – RDD는 어떻게 표현되고, Lineage는 또 어떠한가?

5. Implementation – 잡스케쥴링, 인터프리터 통합, 메모리 관리, 체크포인팅등을 어찌 구현?

6. Evaluation – 성능 평가. 특히 메모리에 자료를 올려놓은 Hadoop과도 비교함

7. Discussion – RDD는 제한된 세트인데도 왜 대부분의 프로그램 모델이 표현될까?

8. Related Work – 홀로 태어난 것이 아니라, 이것이 나오기 까지 영향 준 다른 프로젝트들

9. Conclusion

하지만 논문 구성대로라면
재미가 없으므로
제가 재구성 하였습니다.

Motivation

MapReduce가 대형 데이터 분석을 쉽게 만들어주긴 했어용

근데 뭔가 좀 부족해요. 예를 들어 아래의 것들 잘 못함

- 더 복잡하고, multi-stage 한 처리 (머신러닝, 그래프)

- interactive 하고 ad-hoc한 쿼리

이런거 해결할려고, pregel 같은 특수목적 분산 프레임웍들은 나왔지만 뭔가 근본적인 접근이 없을까?

뭐가 빠진 걸가?

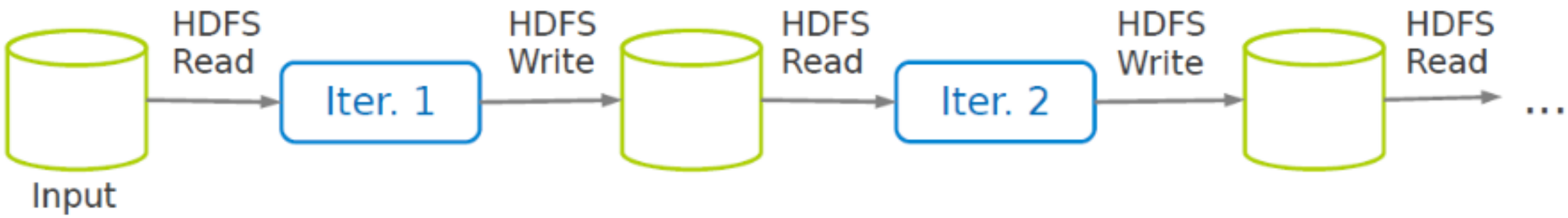
효율적인 Data Sharing 도구

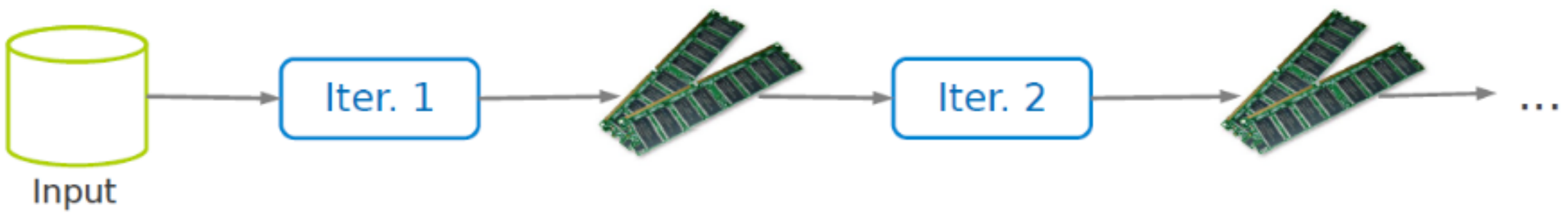
MR이 iteration에서 빡센 이유는

각 iteration을 돌때마다 스테이지간 자료 공유가

HDFS를 거치기 때문이지 않을까?

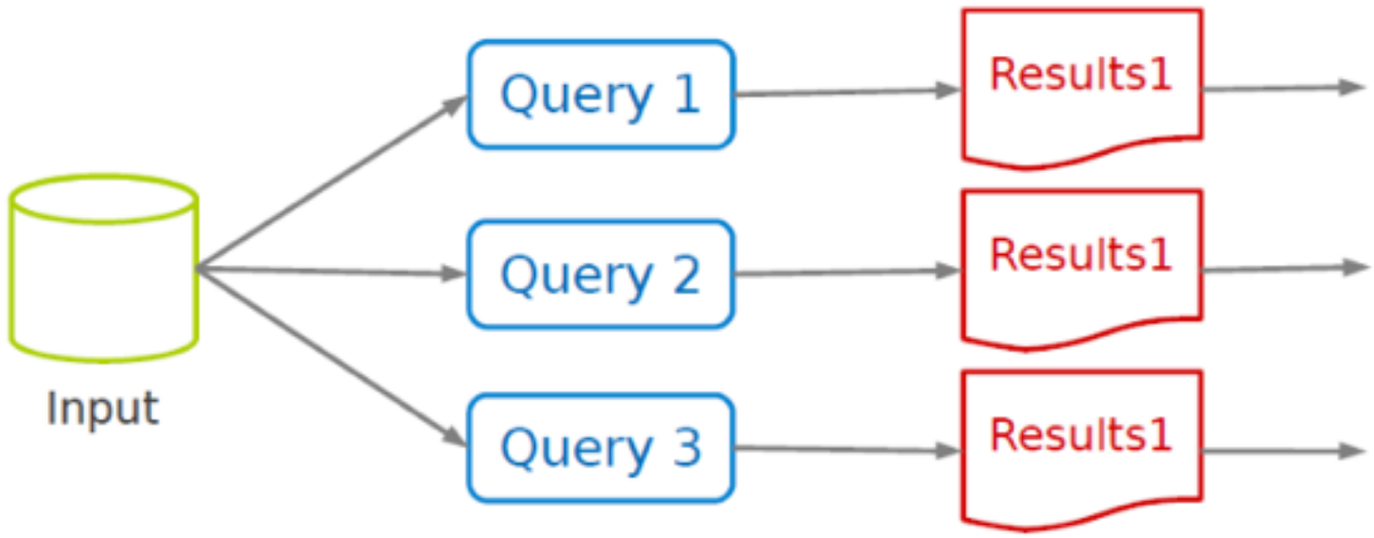
그걸 넘쳐나는 RAM으로 하자!

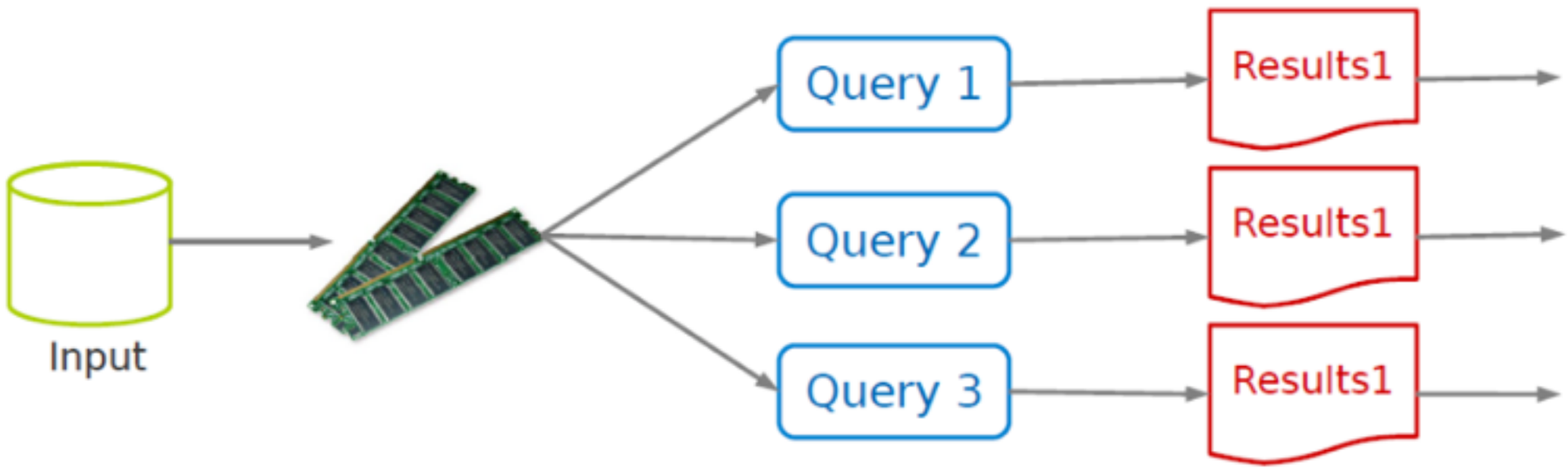




빠르겠지? 후훗?

위쪽의 단계마다 HDFS쓰는 삽질에서, 아래쪽의 램에 저장해 놓고 계속 쓰면 빠르겠지?





빠르겠지? 후훗?

쿼리할 때 마다, 처음부터 읽어오기보단, 한번 램에 올려 놓고, 그다음에 쿼리쿼리쿼리!

문제는 이거임

중간에 뽀개지면(fault)어떡하나?

어떻게 하면 fault-tolerant & efficient한 램스토리지를 만들수 있지?

(이쁘고 착한 여친인건가?)

어떤 이들에겐 '그냥 여자친구'도 넘기 힘든 장벽이지. 힘내라 사나이들.

기존의 RAM을 쓰던 패러다임

RAM은 RAM이니까. update는 되야 되지 않겠는감?

그래서 fine-grained update가 지원되는 RAMCloud, Piccolo등의 접근이 있었다.

근데 fault-tolerance위해 개판됨. replicating & checkpointing의 빡셈

한놈이 부셔져도, 무사하려면? -> 복사해 둬야겠군 -> replicating
replicating하는 동안은 멈춰야하겠지? -> 멈춘다고? -> 느려!
replicating으로도 불안하면 disk에 써야겠지? -> checkpointing -> 엄청 느려!

그런데 GFS(HDFS)가
어떻게 Breakthrough를 일으켰는지
생각해보자.

Modifiy가 안되는 파일 시스템
무조건 쓰며 달리는 파일 시스템

그게 뭔 파일 시스템이야 스럽지만
그걸로 많은 것을 단순화 시켜
큰 시스템을 만들 수 있었다.

때론 상식을 뒤집을 때 혁신이 일어난다.

RAM도 read-only로만 써볼까?

뭔가 일이 착착 풀리기 시작한다.

이것이 RDD!
Resilient Distributed Datasets

Resilient Distributed Datasets

어떤 특성을 가지나?

- Immutable, partitioned collections of records
- 스토리지 -〉 RDD 로 변환하거나, RDD -〉 RDD만 가능

그럼 뭐가 좋나?

- 어 만들어진 이래 고쳐진 적이 없다고? (immutable = read-only)
- 그럼 어떻게 만들어졌는지만 기록해두면 또 만들 수 있겠네?
- 부모로 부터 어찌 만들어졌는지 계보(lineage)만 기록해도 fault-tolerant

그렇게 게임을 했는데 리니지가 원래 뭔뜻인지는 이번에 알았다!

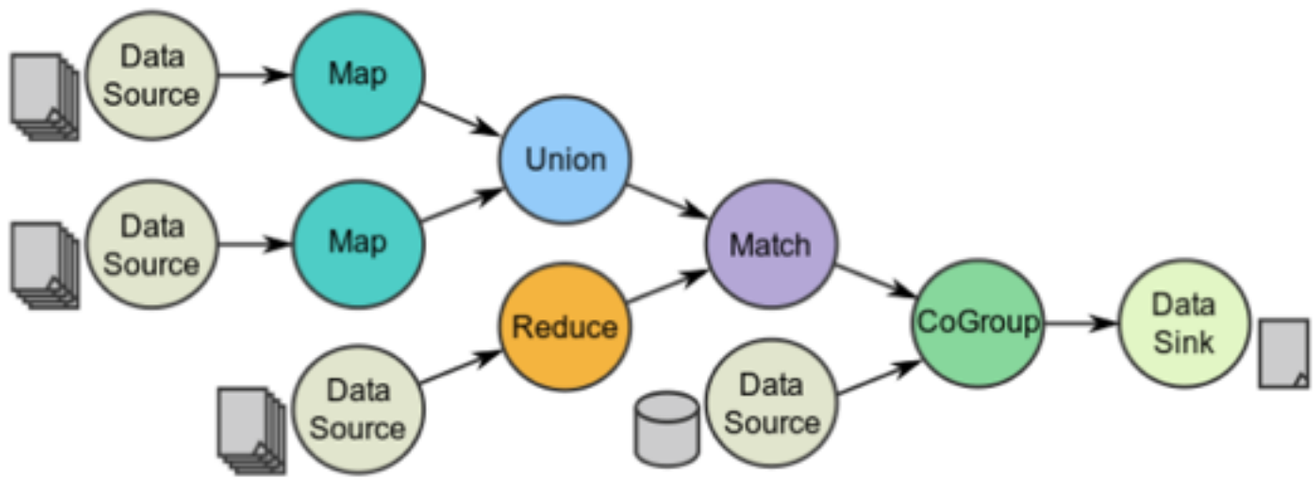
코딩은 어찌 하는가?

DryadLINQ-like API in the Scala language

코딩을 하는 것은 실제로 계산 작업이 되는게 아니라,

점점 더 나아가며 lineage 계보를,

directed acyclic graph(DAG)로 디자인 해 나가는 것



자료가 어떻게 변해갈지를 그리는 와중(transformation)에는 실제 계산은 일어나지 않는다.

그래서 2가지 RDD operator가 있다.

transformations & actions

Transformations	*map*(*f* : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
	filter(*f* : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
	flatMap(*f* : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(*f* : (V, V) ⇒ V)	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	union()	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	join()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	crossProduct()	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	mapValues(*f* : V ⇒ W)	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(*c* : Comparator[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(*p* : Partitioner[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	*count*()	:	RDD[T] ⇒ Long
	collect()	:	RDD[T] ⇒ Seq[T]
	reduce(*f* : (T, T) ⇒ T)	:	RDD[T] ⇒ T
	lookup(*k* : K)	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path* : String)	:	Outputs RDD to a storage system, *e.g.*, HDFS

Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

transformation은 (map, reduce, join 등등) 데이터의 이합집산, 지지고 볶는 흐름
단순히 map, reduce만 있던 MR보다 명령어가 풍부하다.
Actions는 실제로 말미에 '자 모든 transformation된 결과를 내놓아라~' 하는 명령
trans->trans->...->trans->action 요런 구성이 된다.

lazy-execution

인터프리터에서 룰루랄라, transformation들로 코딩하고 있으면 **아무일 안생김**

리니지만 신나게 생성되어 감

action들은 실제로 “야 내놔”하는 작업들

action에 해당하는 명령이 불리면 그제서야 쌓였던 것 실행 (lazy-execution)

근데 lazy가 대박이다.

어떻게 만들어지는지
계보를 다 그려놓은 상태에서
즉 대강의 Execution Plan이
다 만들어진 상태에서
뒤에 실행하므로,

자원이 배치된, 배치될 상황을
미리 고려해서
최적의 코스로 돌 수 있다.

상사가 나한테 똑같이 일 10개를 시킬 때, 하나하나씩 시키고 확인하는 거보다,
차라리 10개 리스트를 한번에 주면, 내가 알아서 순서 정해서 잘 일할 수 있는 것과
같은 원리. (물론 애초에 일을 안시키면 더 좋겠다?)

두가지 type의 dependency

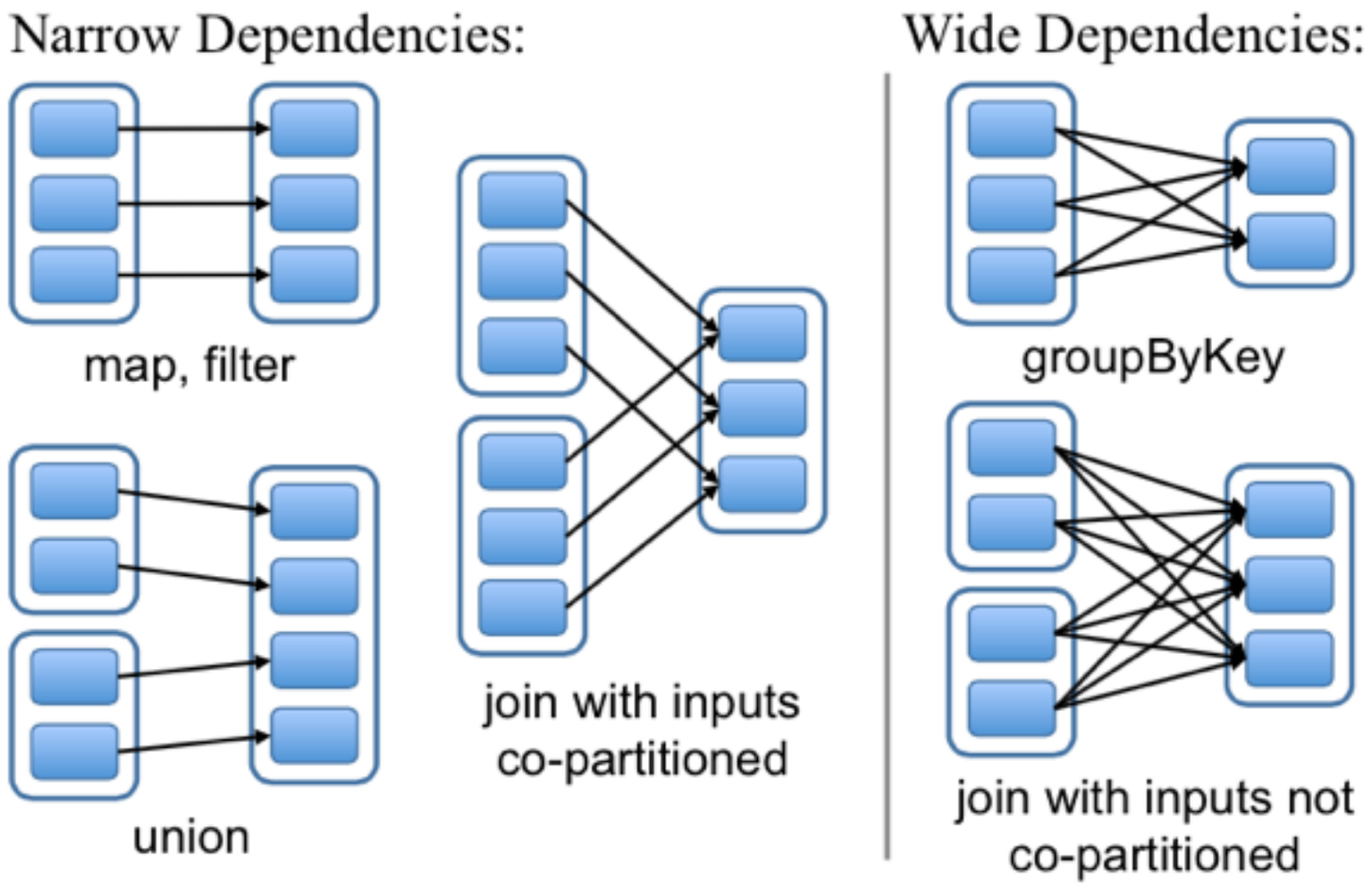
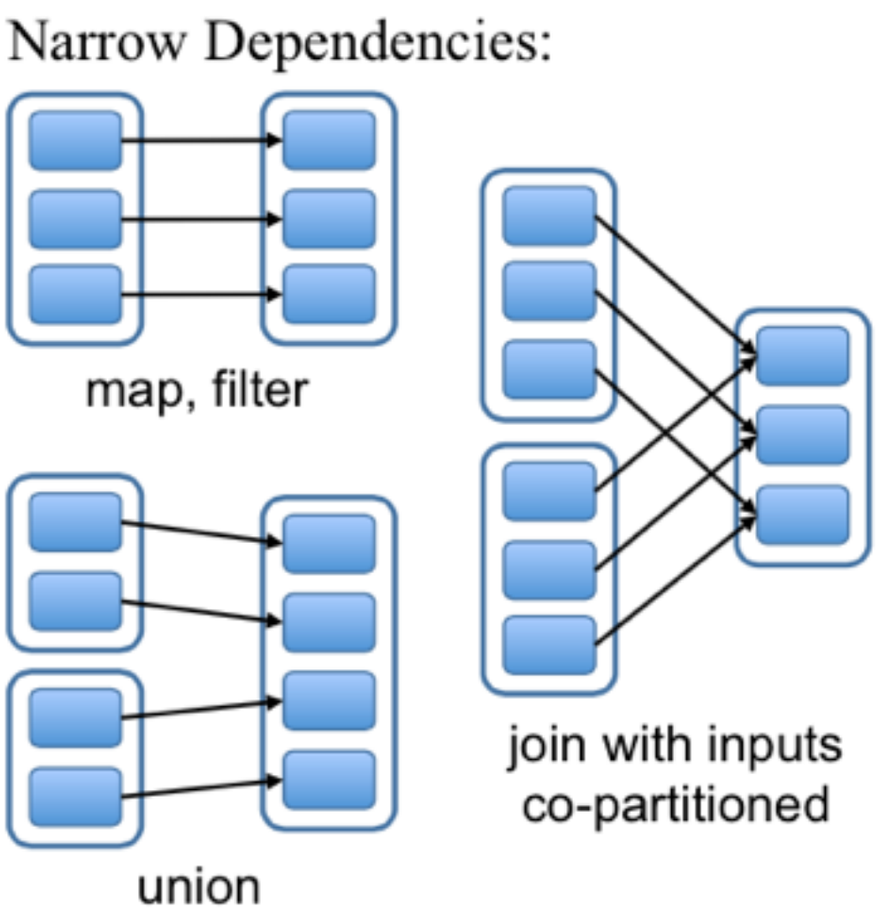


Figure 4: Examples of narrow and wide dependencies. Each box is an RDD, with partitions shown as shaded rectangles.

책상 한자리에서 다 처리할 수 있는 일은 모아서 하는게 좋다. 그런일이 narrow dependecy
이 자료부터 저 자료까지 모든 책상에 있는 자료 훑어와야 하는 건 느리다. 그런게 wide

narrow dependecy를 잘써라!



narrow dependency의 경우,
해당 작업이 다 한 노드에서 돌 수 있다.

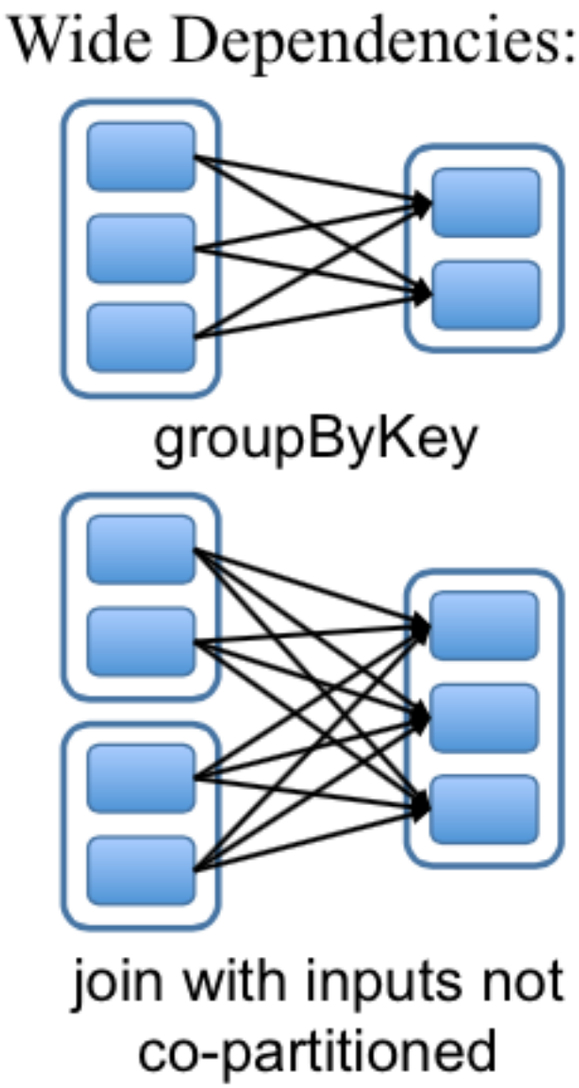
네트워크 안탄다는 소리다

엄청 빠르다.

설사 가지고 있는 파티션이 부셔져도,
그 노드에서 다 복원가능하다.

narrow는 한 책상 (한 노드 안에서) 다 처리 가능하다. -〉 메모리의 속도로 동작한다. 왕빠름
wide는 책상들을 돌아다니며 모은다. (여러 노드) -〉 네트웍의 속도로 동작한다. 좀 느림

wide dependency



쉽게 말해,
셔플이 일어나야 하는 애들이다.

네트워크 타야 한다.

부셔지만 망한다. (계산비용 비쌈)

그래서 이런 애들은 추가로
checkpointing해주는게 좋을수도.

job scheduling

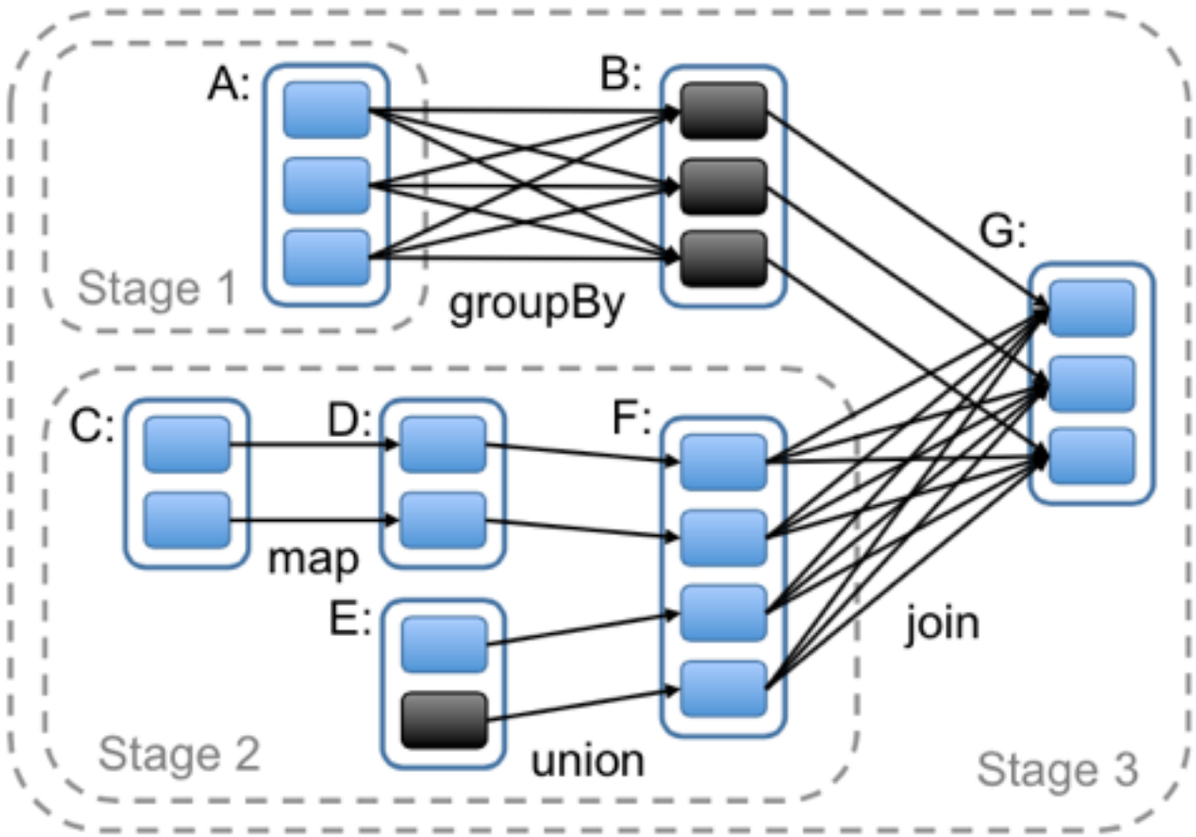


Figure 5: Example of how Spark computes job stages. Boxes with solid outlines are RDDs. Partitions are shaded rectangles, in black if they are already in memory. To run an action on RDD G, we build build stages at wide dependencies and pipeline narrow transformations inside each stage. In this case, stage 1's output RDD is already in RAM, so we run stage 2 and then 3.

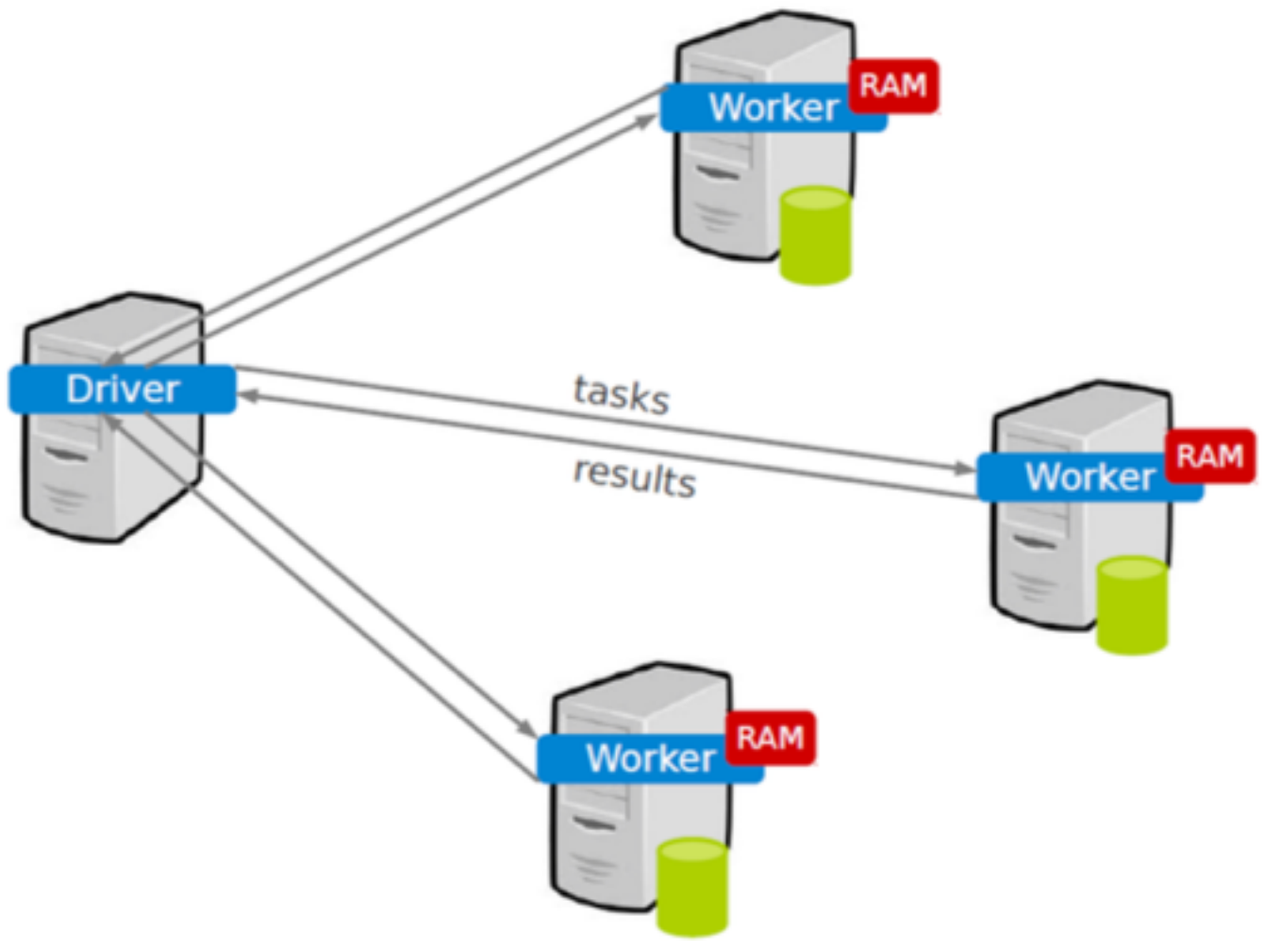
DAG에 따라 계산해 나간다.

이미 계산해놓은 파티션은 패스

필요한데 없는 파티션을 lineage를 이용해 만들어가며 결과를 내놓는다.

파티션이 수행될 노드는 data-locality 를 고려해 결정된다. (특히 HDFS)

워커들 도는 모습은 기존과 비슷



데이터 덩어리는 파티션으로 쪼개어 져서, 머신들에 배분되어 있습니다.
드라이버에서 내려온 transform들이 각 파티션마다 수행된다.
필요에 따라(shuffle류) 파티션 내용이 다른 머신으로 쏘아집니다.

여기서 궁금.

수행중 메모리가 모자라면 어찌 되나요?

LRU로 안쓰는 파티션 날림

- 캐시랑 하는짓이 유사
- 원본을 다시 가져올 수 있다는 점도 캐시랑 유사하니
- 당당히 캐시랑 비슷하게 동작

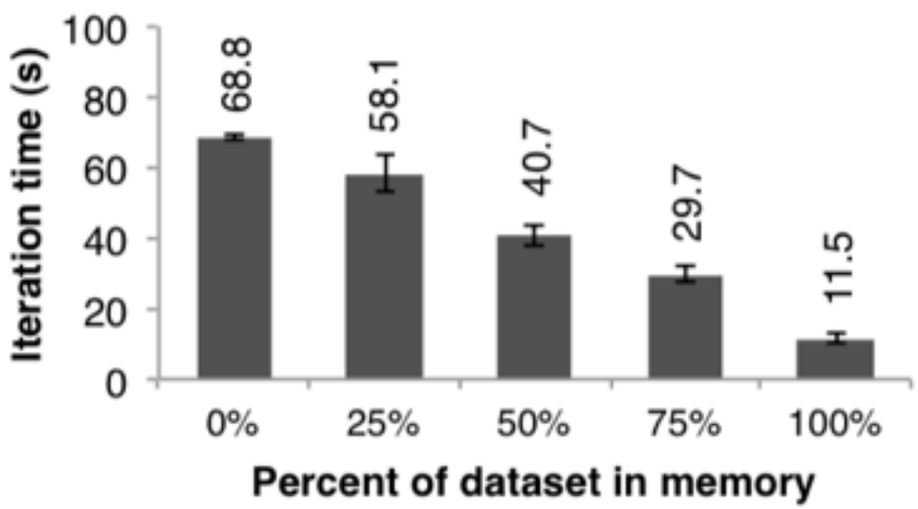


Figure 12: Performance of logistic regression using 100 GB data on 25 machines with varying amounts of data in memory.

여기서 궁금.

수행중 Fault가 나면 Recovery 영향은?

lineage는 용량이 작기때문에
잘 로깅해 놓고 있다가(부담없다)
특정 파티션에 문제 생기면
다른 노드서 땡겨서 실행

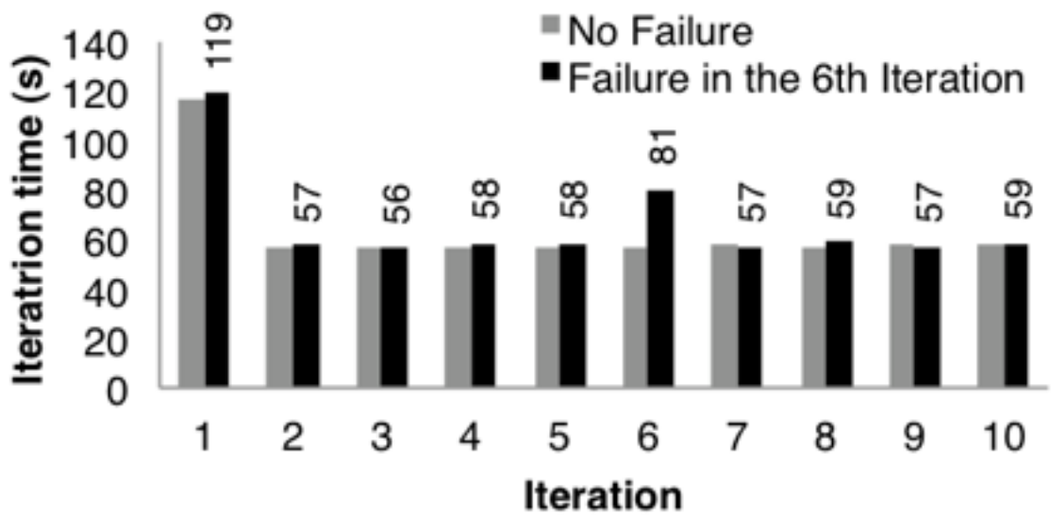


Figure 11: Iteration times for k-means in presence of a failure. One machine was killed at the start of the 6th iteration, resulting in partial reconstruction of an RDD using lineage.

그래도 체크포인팅을 한다면?

뽀개져도 lineage로 복원해 낼 수 있지만

wide dependency의 경우 모든 파티션을 다 훑어야 하므로 역시 느리다.

때로는 재계산을 통한 구축 비용이, 디스크에 checkpointing해 놓았다 걍 읽는 것보다 더 오래 걸릴 수도 있다.

이게 의심된다면 해놓자. checkpoint.

여기서 read-only가 또 활약. 다른 시스템들 처럼 멈출 필요 없이 병렬로 백그라운드에서 async로 돌리면 됨. 왜냐면 쓰는동안 안바뀌니까.

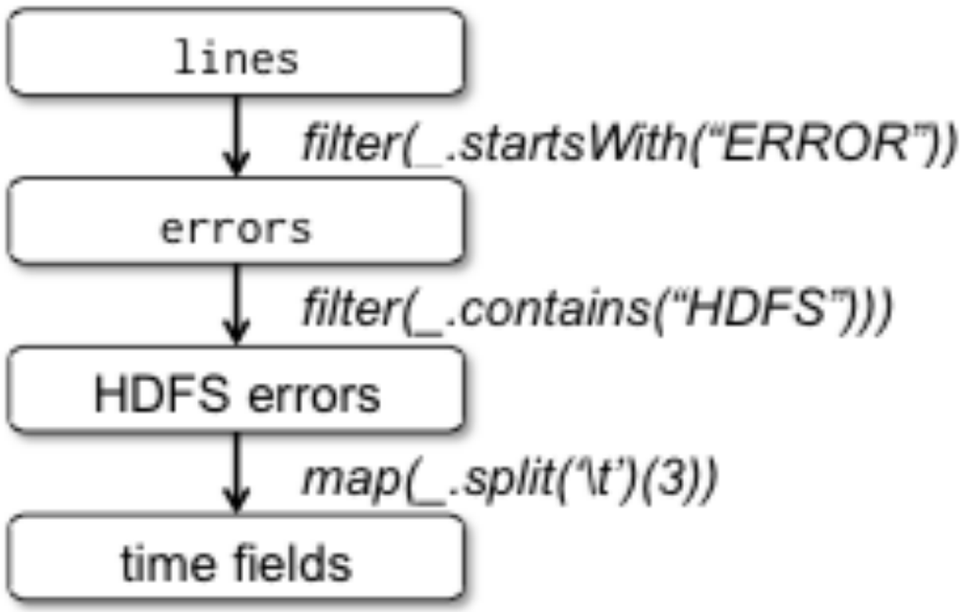


Figure 1: Lineage graph for the third query in our examp
Boxes represent RDDs and arrows represent transformations

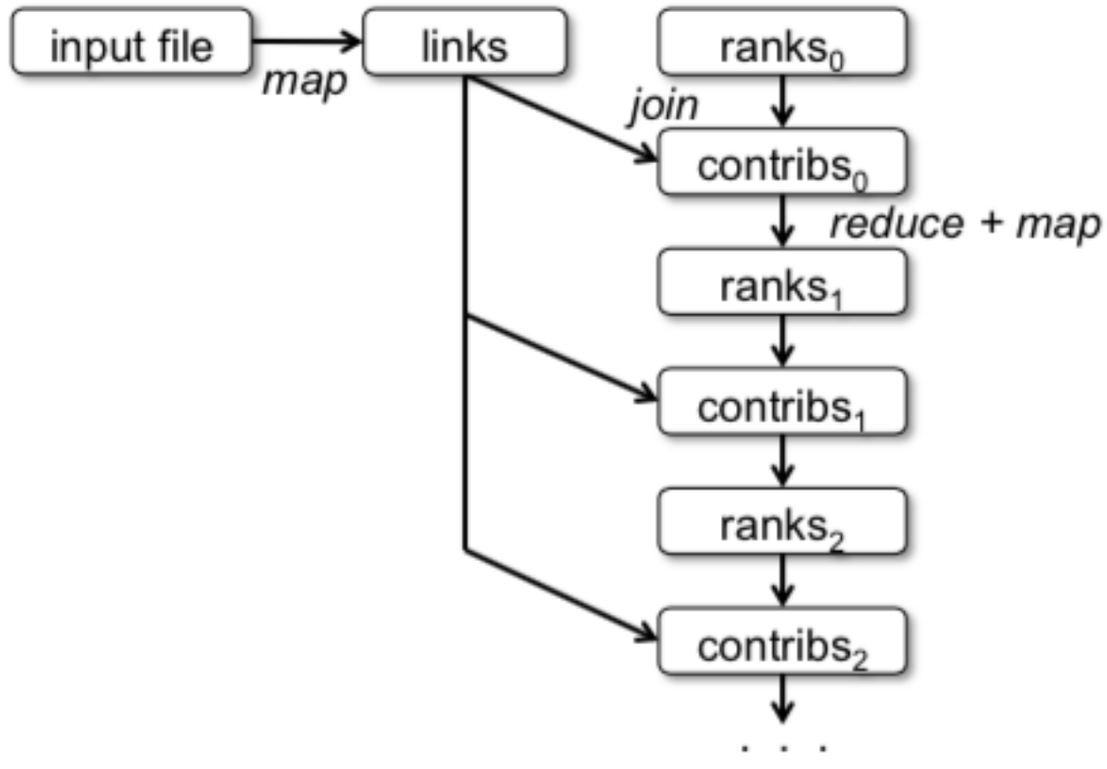


Figure 3: Lineage graph for datasets in PageRank.

이놈은 뭐 부서진 파티션이 있어도 금방 해당 파티션 만들 수 있음

이런 애들의 결과물은 반복된 shuffle(join)의 결과물이기에 checkpointing해두는 게 좋음

흥 네가 성능이 그리 좋아?

좋다.

IO Intensive CPU Intensive

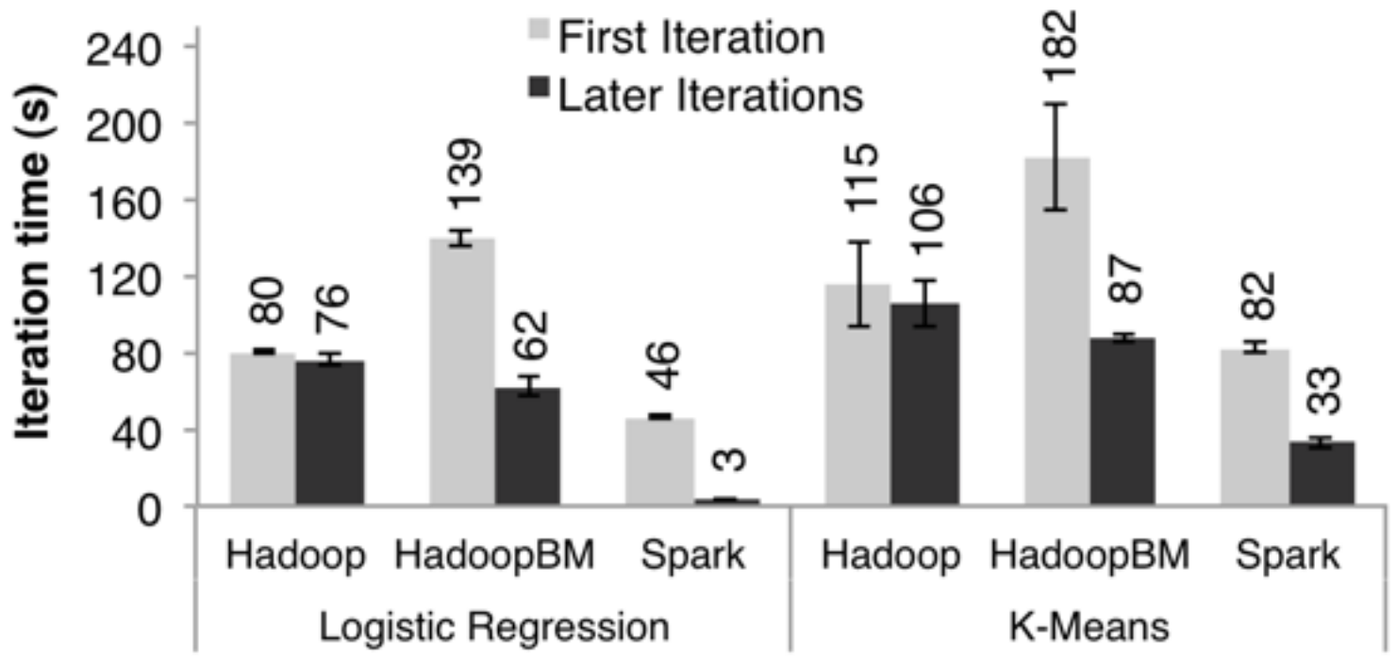
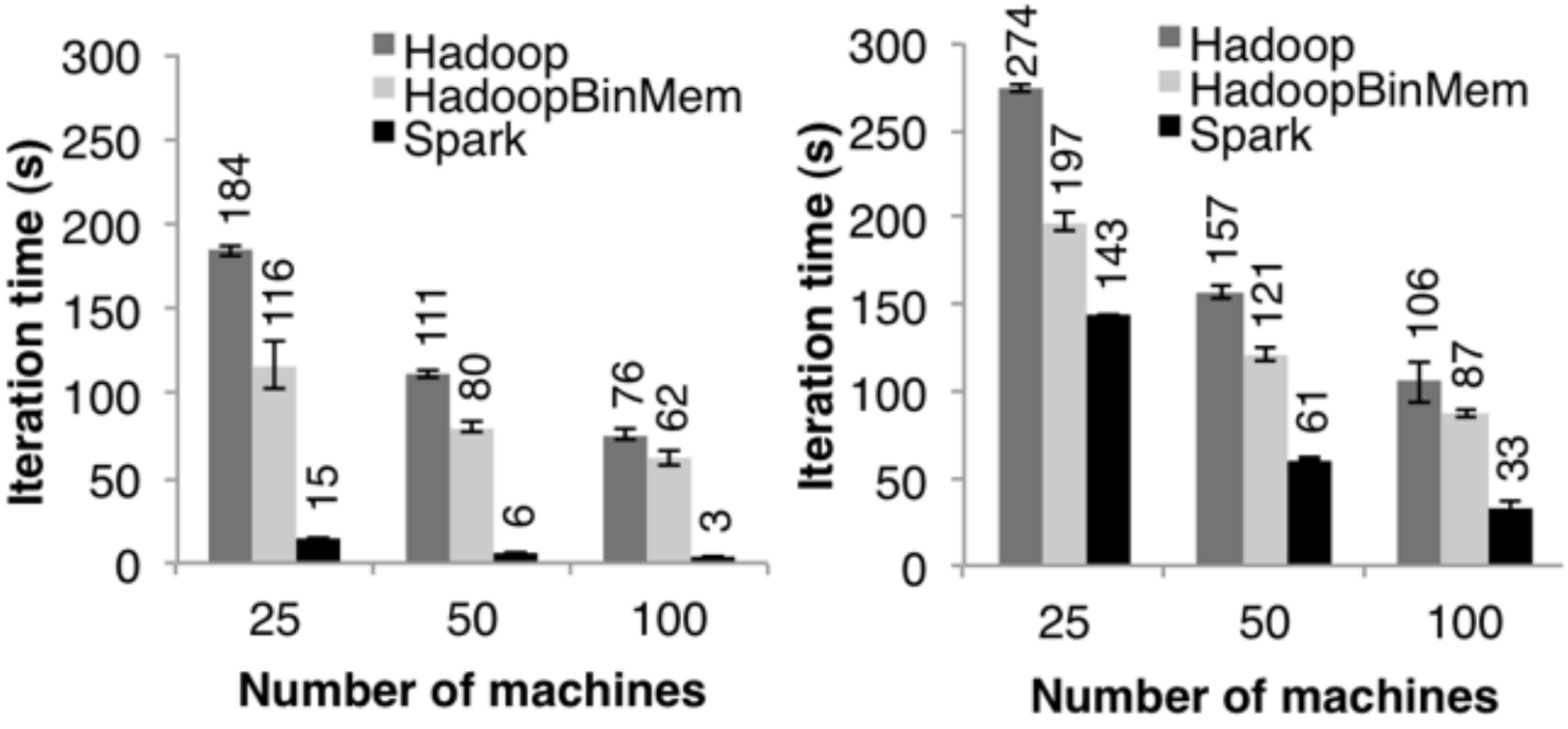


Figure 7: Duration of the first and later iterations in Hadoop, HadoopBinMem and Spark for logistic regression and k-means using 100 GB of data on a 100-node cluster.

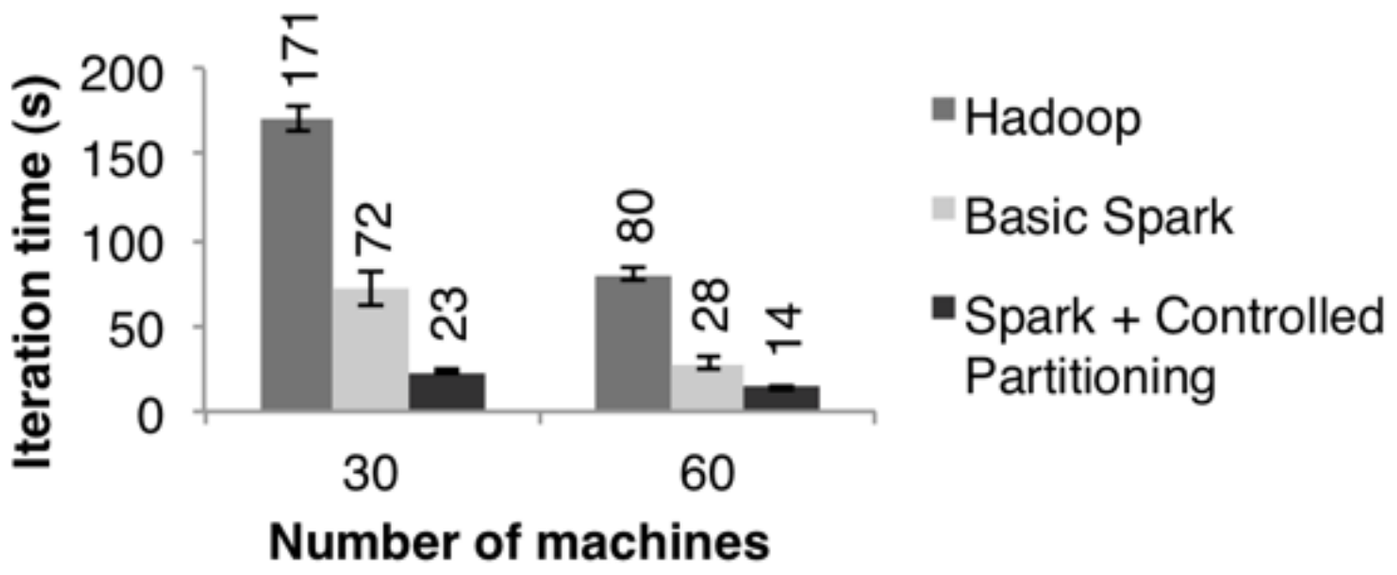
첫번째 Iteration은 다들 텍스트 파싱, 이건 대강 넘기시고
HadoopBM은 파일을 디스크 HDFS가 아니라 in-memory HDFS에 쓴다.
Hadoop이라도 하드를 쓰지 않고 메모리만 써가며 한번 도전해 본다.
두번째 이후가 순수 계산 대결인데(까만색) Spark이 압도적



(a) Logistic Regression

(b) K-Means

머신이 늘어도
그러한가?



그러하다.

Figure 10: Performance of PageRank on Hadoop and Spark.

Spark VS in-memory hadoop

RAM쓰니까 당연히 빠르겠거니 했지만

왜 in-memory HDFS를 쓰는 HadoopBinMem보다 빠른가?

- Hadoop의 소프트웨어 스택이 너무 복잡해서 기본요금이 비쌈
- HDFS라는 파일 구조체가 다루는데 비쌈
- Bin을 계산을 위해 Java object로 또 돌리는데 계산 비용
  - (spark은 java object상태로 계속 활용한다. 물론 용량큼)

결국 RDD가 Spark이 짱이란 말이 하고 싶었나보다.
풀업저글링(in-memory 로 전부 처리하는 hadoop)도 안되더라.
하긴 아무리 메모리라도 애초에 작업 반복할때마다 전부 썼다 읽었다 하는데 될리가;;;

그래. 빠르다. 근데 표현영역은?

아무래도 RDD라는 제한된 형태를 쓰다보니, 모던한 요구사항들을 다 커버할 수 있는지 궁금하다?

논문에 쓰여 있다. 거의 다 된단다.

MapReduce	Pregel
DryadLINQ	Iterative MapReduce
SQL	Batched Stream Processing

하긴 우린 Map, Reduce만 가지고도 지지고 볶고 어떻게든 잘 써왔다.
명령어 갯수가 더 많은데 당삼 더 잘되겠지..

결론

RAM을 ROM처럼 써보기로 했다.

(RAM이지만 한번쓰고 다신 안고쳐)

fault-tolerant & efficient한 램스토리지를 만들었다!

(이쁘고 착한 여친을 만났다!)

그리고 이것을 기반으로 Spark의 전설이 시작되었다.

부록) RDD에 영향을 준 패러다임들

- 관련된 작업들
  - 클러스터 프로그래밍 모델
    - data flow model
    - high-level programming interface
      - DryadLINQ
      - FlumeJava
    - high-level interface for specific application
      - Pregel
        - iterative graph
        - 좋긴 한데 메모리 사용을 유저 인터페이스에서 직접 정의하지 않고, 프레임웍이 알아서 한다.
          - 때문에 공유되는 데이터가 무엇인지
          - 쿼리간의 공유 데이터의 결정등도 프레임웍이 알아서
          - 심플하게 말하면, pregel에서는 유저가 "이걸 메모리에 올린 다음, 재활용해서 Query해야지"를 정할 수 없다. 정해주는 대로
      - BUT RDD
        - RDD는 명시적으로 분산 메모리 사용을 정의한다.
          - 이런 특성 때문에 interactive data mining같은 어플리케이션이 가능하지
    - Distributed shared memeory
      - Piccolo
        - DHT에 분배된 메모리 셀에 update가 가능함
      - RAMCloud
        - key-value store
      - BUT RDD
        - 위의 것들은 메모리 모델에 가까워서 API가 set, get, update 정도이다
          - 이에 반해 RDD는 map, sort, join등등 다채로워
        - 위의 것들은 fail-over를 위해서 checkpoint와 rollback을 쓰는데, 이거 비싸다.
          - lineage 베이스의 immutable은 요게 싸게 먹힘
        - 요거 RDD에 영감을 많이 주었음
  - 캐싱 시스템
    - Nectar
      - DryadLINQ job들 사이에 중간 결과를 재활용 할 수 있게 시도함
        - 쉽게 말해 MR의 intermediate데이터에 태그를 달아 또 활용할 수 있게 했었던 거임
        - 근데 nectar는 그걸 메모리에 저장안하고 분산파일 시스템에 저장했음
        - 그래서 요걸 메모리로 해야겠다~~ 생각함
    - Ciel, FlumeJava
    - BUT RDD
      - 위의 것들은 재활용이라는 의도는 가지고 있었는데, 대체로 in-memory캐싱이 아니었고, 명시적으로 요거 캐시해라 같은 명령어 셋이 없었음
    - Ananthanarayanan
      - DFS에 대한 in-memory캐시 레이어에 대한 논문을 내긴 했는데, 이미 디스크에 있는 것들에 대한 빠른 억세스이지, 다음 스텝 밟고 나면 의미 없음
  - Lineage
    - 사실 요거는 scientific computing에서 전통적인 주제임
  - Relational DB
    - RDD는 어찌보면 RDB의 view와 비슷하다
      - 어떻게 만들어진다는 형식을 정의하고
        - 때때로 materialize된다는 점에서
    - 물론 통짜 단위로 작업이 이루어진다는 것이 좀 다르다
      - 레코드 업데이트가 아니라
      - 테이블 단위로 작업하는 DB같은 느낌이지