

Spark/Shark

Daegeun Kim (dani.kim@geekple.com)

README

- 영어를 못하는 사람이 영문 자료를 보면서 작성되었습니다.
 - 논문, 프로젝트 페이지 등 (영어 못하는 사람이 저예요.)
- 논문이나 기타 문서에 있는 정보더라도 소스코드를 본 것 위주로 설명합니다.
 - 물론 소스코드를 잘못 해석했을 수도 있으니 참고!
- 단기간에 보고 작성된 것이라 오류가 있을 수 있습니다.

Agenda

- Spark
 - Granularity / 간단한 소개
 - RDD / Lineage / Fault-Tolerant
 - High-level Operators / Storage / Cache
- Shark
 - Architecture
 - Features / Cache

Spark

Spark

- 맵리듀스와 같은 클러스터 컴퓨팅 시스템이다.
- 병렬 연산을 위한 High-Level 명령 세트를 제공한다.
 - map, filter, join, ... (RDD bulk operation)
- 작업분배 및 Fault-Tolerance 에 대한 걱정을 줄인다.
- 대부분 코드는 스칼라(Scala)로 작성되었다.
- 퍼포먼스향상을 위해 인메모리로 데이터를 유지할 수 있다.
- 그 핵심에 RDD가 있다.

들어가기에 앞서 ...?

Granularity

Coarse-grained vs Fine-grained

상대적인 개념

Fine-grained

- 세밀하고 잘게 쪼갠 것을 의미

Coarse-grained

- Fine-grained 보다 적고 좀 더 큰 것을 의미
- 좀 더 뭉뚱그린 개념

Granularity

- 현 슬라이드 주제를 빗대어 비유하면
 - Coarse-grained : Shark랑 Spark에 대해 조사해~!
 - Fine-grained
 - Hive를 대체할 수 있는거임? / 하둡과 다른 점은 뭐임?
 - 백배 빠르다는데 그게 진짜임? / ...
- 또 다른 예?
 - Coarse: 이제해!, Fine: 출금!, 입금!

Hadoop을 대체할 수 있나?

- Spark의 목적은 Hadoop의 Map/Reduce와는 다르다.
- Iterative Algorithms에 더 중점을 뒀고
 - 반복적인 데이터마이닝에서 M/R보다 높은 성능을 지닌다.
 - 스팸 필터링, 자연어 분석, 도로 교통 예측 등
- 결론은 모든 부분을 대체할 수는 없다.
- 하지만 Spark는 DataFlow 처리 및 Iterative Algorithms 처리 등을 적절히 수용하고 있다.

하둡이 가지는 문제점

- Iterative Algorithms 관점!
- 각 이터레이션별 중간데이터는 HDFS에 쓰기와 읽기가 반복되고 복제되면서 디스크 I/O가 많이 발생한다.
 - 중간파일이 복제되면서 네트워크 I/O도 발생

RDD

- Resilient Distributed DataSet (탄력적인 분산 데이터셋)
- 효율적인 Fault-Tolerance 요구사항을 충족을 위해
 - 제한된 공유 메모리 형태를 가지며,
 - Coarse-grained Transformations! 기반으로 되어있다.
- read-only이며 스토리지에 있는 데이터를 통해 생성
 - 또는 다른 RDD를 기반으로 변형된 RDD를 생성한다.
 - High-Level 명령 세트를 제공 (map, filter, join 등)

Distributed Shared Memory

- 보통의 Key-Value 스토어
- 데이터 공유방식은 **Fine-grained updates** 방식으로
 - 임의의 데이터를 쓰기 혹은 읽기 등! (설명이 아리까리한데? :-(죄송...)
- RDD처럼 효율적이면서 Fault-Tolerance 구현이 어렵다.
 - 뒤에 Lineage에서 더 자세히 언급합니다. :-)

RDD Operators

- RDD를 이용한 연산에 사용되는 오퍼레이션 세트!
- 모든 오퍼레이션은 Bulk 오퍼레이션이다.
- RDD가 Read-only이므로 읽거나 또 다른 RDD로 변환만 가능하다.
- Operator는 새로운 RDD를 생성을 의미합니다.
 - 실제로 수행되지 않아요! (Lineage Graph 생성)
 - Action은 드라이버(여기서는 shell 같은 클라이언트 어플)에게 결과를 주거나 결과를 어디가 저장할 때 사용합니다.
 - 실제로 연산이 진행됩니다.

지원하는 High-Level Ops.

- map
 - filter
 - flatMap
 - sample
 - groupByKey
 - reduceByKey
 - union
 - join
 - cogroup
 - crossProduct
 - mapValues
 - sort
 - partitionBy
 - ...
- Actions
 - count
 - collect
 - reduce
 - lookup
 - save

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

Pig를 보신 분은 이해하기 쉽습니다. 표기법이 스칼라 문법이라서 헷갈릴 수 있는데 이 것까지 설명하면 너무 많....

Lineage

사전적 의미로 혈통, 핏줄, 일족, 가계와 같은 의미를 지닌다.

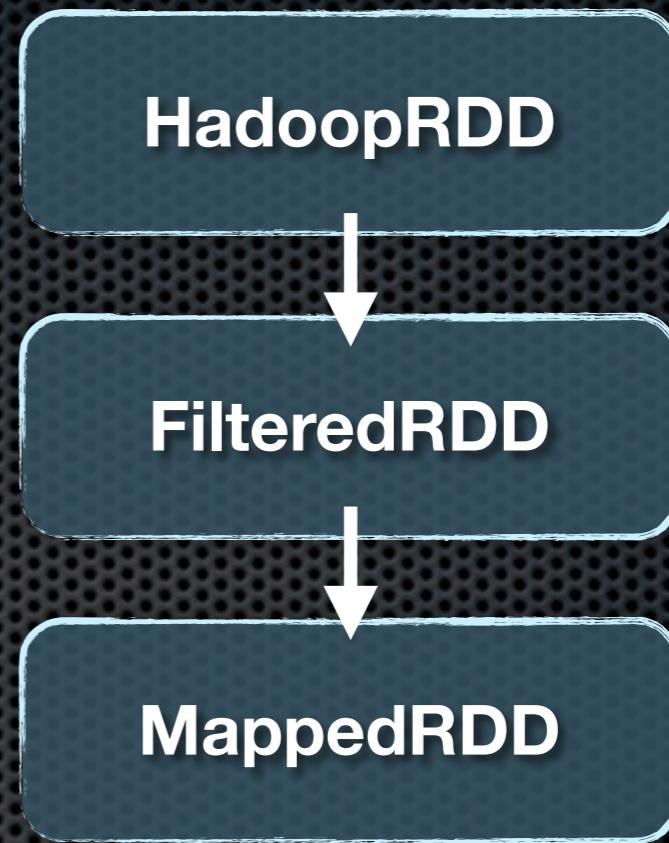
Lineage Graph

단순 예제.

```
val messages = spark.textFile("hdfs://...")  
val errors = messages.filter(_.contains("ERROR"))  
val category = errors.map(_.split(",")(3))
```

이해를 돋고자 보충 설명하면 (밑줄(_))은 scala의 placeholder 문법입니다.)

`_.contains("ERROR")` 는 func anonymousFunc1(String line) { return line.contains("ERROR"); } <= 이해를 위한 가짜코드
`_.split(",")(3)` 는 func anonymousFunc2(String line) { return line.split(",")[3]; } <= 이해를 위한 가짜코드



Fault-Tolerant

- Lineage Graph로 잃어버린 RDD를 재구성 가능하다.
 - 프로그램을 처음부터 다시 시작할 필요 없다.
 - 불필요한 체크포인트 등의 불필요한 오버헤드를 줄인다.
 - 복제와 같은...
- 물론, 안전하게 유지하도록 체크포인트 기능을 제공한다.
- StorageLevel 참고 및 checkpoint 메서드 참고.

물론 최초 소스는 안전한 스토리지에 있어야 한다. (이전 슬라이드에 언급했듯 RDD는 스토리지 및 또 다른 RDD에 의해 생성된다.)

Stage / Dependencies / Scheduler

- 이 부분은 생략합니다.

Storage

- RDD는 선택적으로 persist하게 유지할 수 있다.
 - In-memory Storage (Deserialized Java Objects)
 - In-memory Storage (Serialized data)
 - On-disk Storage
- (disk, memory, deserialize, replication)

Storage Level

- DISK_ONLY / DISK_ONLY_2
- MEMORY_ONLY / MEMORY_ONLY_2
- MEMORY_ONLY_SER / MEMORY_ONLY_SER_2
- MEMORY_AND_DISK / MEMORY_AND_DISK_2
- MEMORY_AND_DISK_SER
- MEMORY_AND_DISK_SER_2

_2 가 붙은 것은 2개 복제본을 유지를 의미하며, _ONLY는 다른 스토리지는 쓰지 않는 것을 의미
_SER 는 객체를 Serialize 한 후 저장하겠다는 것을 의미한다.

Deserialized Java Objects vs Serialize to Bytes

Deserialized Java Objects

- CPU연산이 적어 빠르다.
- 메모리 사용률에 있어 매우 비효율적이다.
- GC 오버헤드가 심하다.

Serialize to Bytes

- 메모리 효율이 매우 좋다.
- GC가 보다 효율적이나
- serialized/deserialize 과정에서 CPU 사용이 높다.

Java Objects vs Bytes

- 메모리 효율 = Bytes (x3)
- CPU 효율 = Java Objects (x5)
- GC 효율 = Bytes

Cache!

```
val messages = spark.textFile("hdfs://...")  
val cached = messages.cache() // or persist()  
val errors = cached.filter(_.contains("ERROR"))  
val info = cached.filter(_.contains("INFO"))
```

cache 메서드 또는 persist 메서드를 통해서 캐쉬할 수 있습니다. cache 메서드는 MEMORY_ONLY 스토리지 레벨이며 persist는 다른 레벨로 설정할 수 있다.

실습

- > val rows = sc.textFile("hdfs://tpch-1:9000/tpch/lineitem")
- > rows.count # took 3.80104688 s
- 11997996
- > val cached = rows.cached()
- > cached.count # took 3.19858059 s
- > cached.count # took 0.119807971 s
- > cached.filter(_.contains("TRUCK")).count # took 0.439822487 s
- 1714622

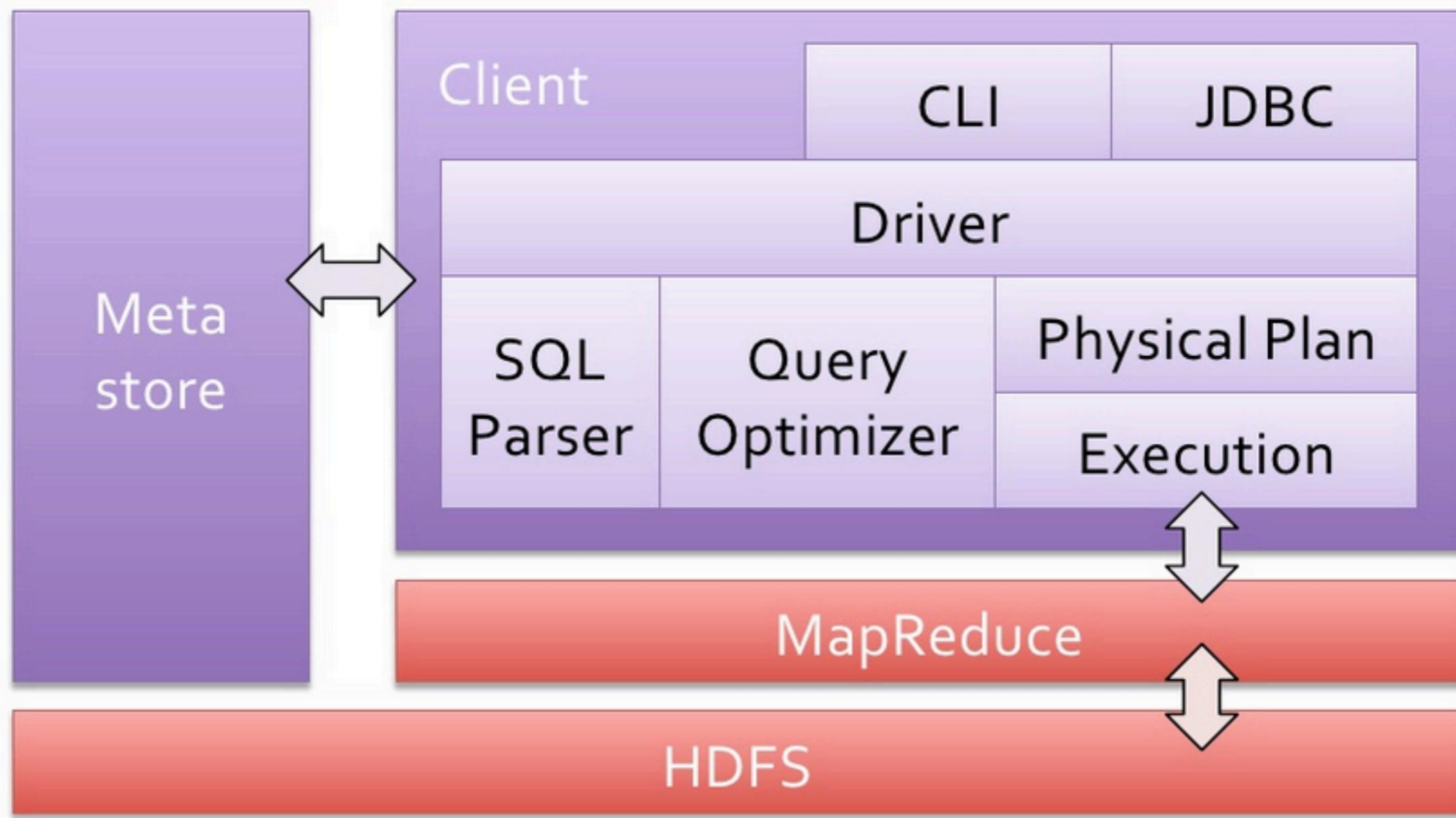
Shark

Shark

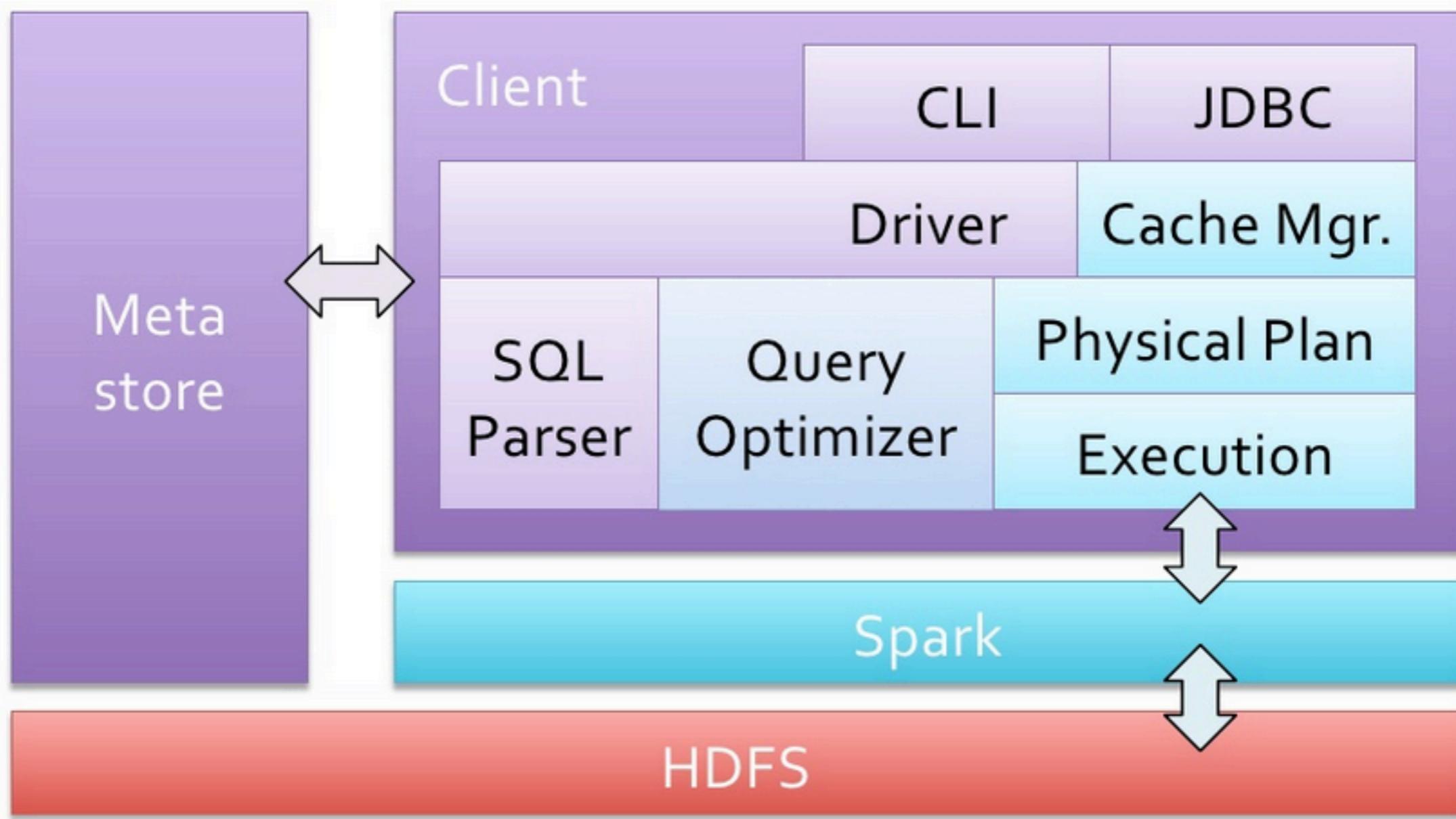
- Spark 상에서 Hive 구현 (물론 하이브 라이브러리 이용)
- Spark의 특징으로 가지는 특징
 - Apache M/R과는 다르게 Task 실행 오버헤드는 없다.
 - 각 스텝별 중간 데이터가 HDFS와 같은 스토어에 저장되지 않는다. 복제 및 Disk IO 등 오버헤드
 - 메모리로 커버 가능한 크기는 셔플 퍼포먼스 개선
 - 이건 코드를 못 찾아서 장담 못함

Architecture

Hive Architecture



Shark Architecture



Architecture

- 이용할 수 있을 만큼 하이브 것을 이용했다.
 - CLI, Thrift Server, JDBC, Metastore 등
- Thrift 또한 이용했음은 외부시스템은 클라이언트 라이브러리가 변경되지 않는다는 의미.
- Spark만큼 RDD 활용이 유연하지 않다.

Features

- Shark Feature는 곧 Hive Feature.
 - Hive UDF, UDAF, SerDe 등 가능. Impala는 안됨.
 - 대부분의 하이브 쿼리 지원
- Apache M/R기반이 아닌 Spark 상에서 동작.
 - In-memory RDD를 활용해서 성능향상을 꾀할 수 있음.
- CTAS (CREATE TABLE as SELECT ...)을 통해 캐싱
 - RDD의 persist를 호출하는 역할 (TableRDD)

Unsupported Features

- ADD FILE
- STREAMTABLE
- Outer join with non-equi join filter
- Table Statistics
- Table buckets
- Automatically convert joins to mapjoins
- Sort-merge mapjoins

예전 자료를 본 것이라 구현이 된 Feature가 있을 수 있습니다.

Unsupported Features

- Partitions with different input formats
- Unique joins
- Writing to two or more tables in a single SELECT
INSERT
- Archiving data
- Virtual columns (INPUT_FILE_NAME,
BLOCK_OFFSET_INSIDE_FILE)
- Merge small file from output

예전 자료를 본 것이라 구현이 된 Feature가 있을 수 있습니다.

Efficient In-Memory Storage

- Columnar Memory Storage 개발 (shark)
 - Spark의 Memory Storage 상에서 구동
- 공간절약과 읽기 성능 개선 효과 (shark)
 - Complex Types => single byte array (shark)
 - Other Types => java primitive type (shark)
- 대략 Data 270m => JavaObj: 971m, Serialized : 289m
- CPU-Efficient compression (shark)
 - dictionary encoding (Text 대상)
 - bit packing (byte, short, int)

ColumnarSerDe 구현을 참고하세요.

Shark Cache

- [Table name]_cached
- CTAS와 Table Properties를 이용한 방법
 - “shark.cache”: true or false
 - “shark.cache.storageLevel”: memory or disk
 - 이전 슬라이드 중 Storage Level을 참고하세요.
- 같은 테이블도 세션이 다르면 캐쉬된 테이블 접근 불가.
- 개인적으로 spark는 마음에 들지만 shark는 부족해 보임.

Shark Cache

- `create table lineitem_cached as select * from lineitem where l_shipmode = 'TRUCK';`
- `select count(*) from lineitem_cached;`
- `create table lineitem2 tblproperties("shark.cache" = "true", "shark.storageLevel" = "MEMORY_ONLY_2") as ...`
- `select count(*) from lineitem2;`

Shark Shell

- 제공 메서드
 - sql2rdd
 - sql2console
 - sql
- 이걸 이용하면 굳이 CTAS를 쓸 필요는 없다. 자체가 RDD이므로 persist 메서드 이용하면 됩니다. (TableRDD)

주의: 아직 결론을 내기엔 내공이 부족하여 소스분석 및 벤치마크 테스트가 더 진행되어야 합니다.

TPC-H Benchmark

외부에 공개하기엔 정확하지 않아 이 슬라이드에서는 빠집니다.

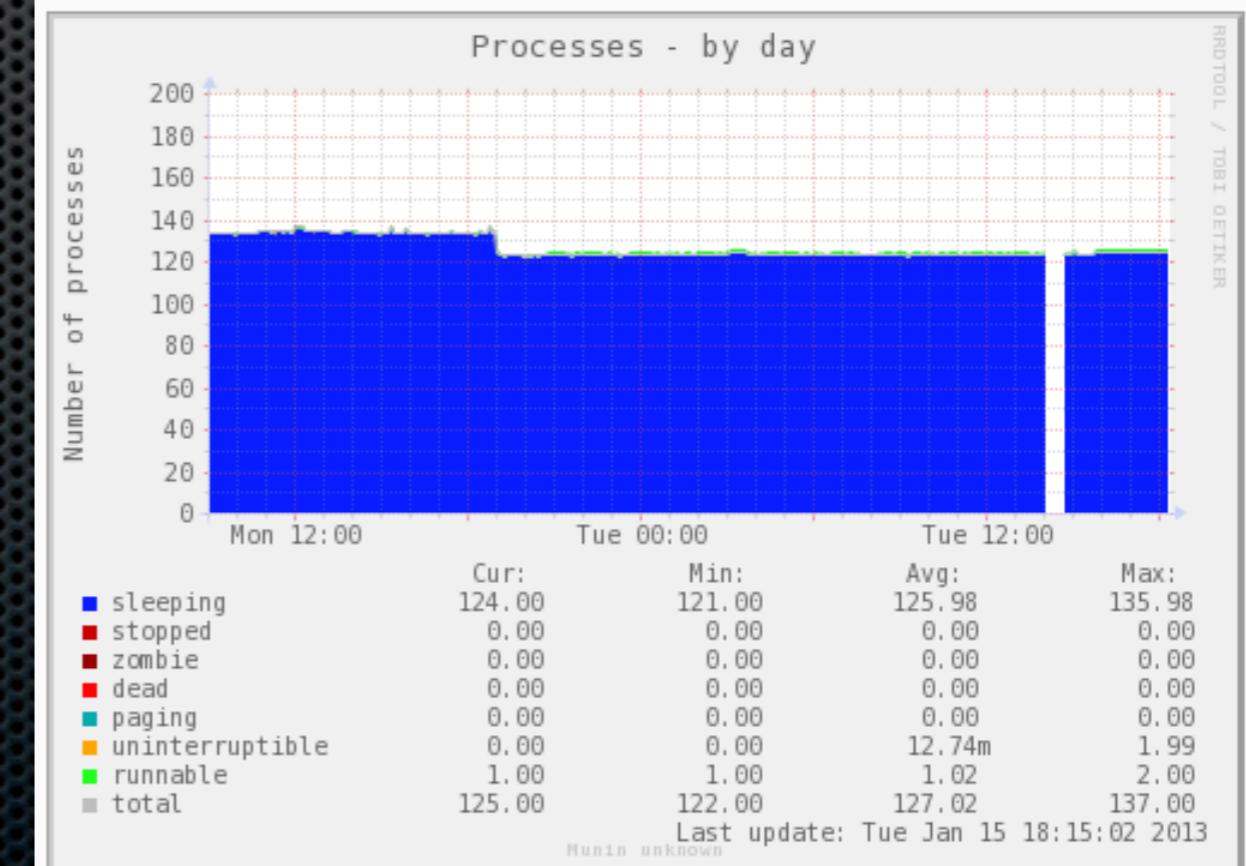
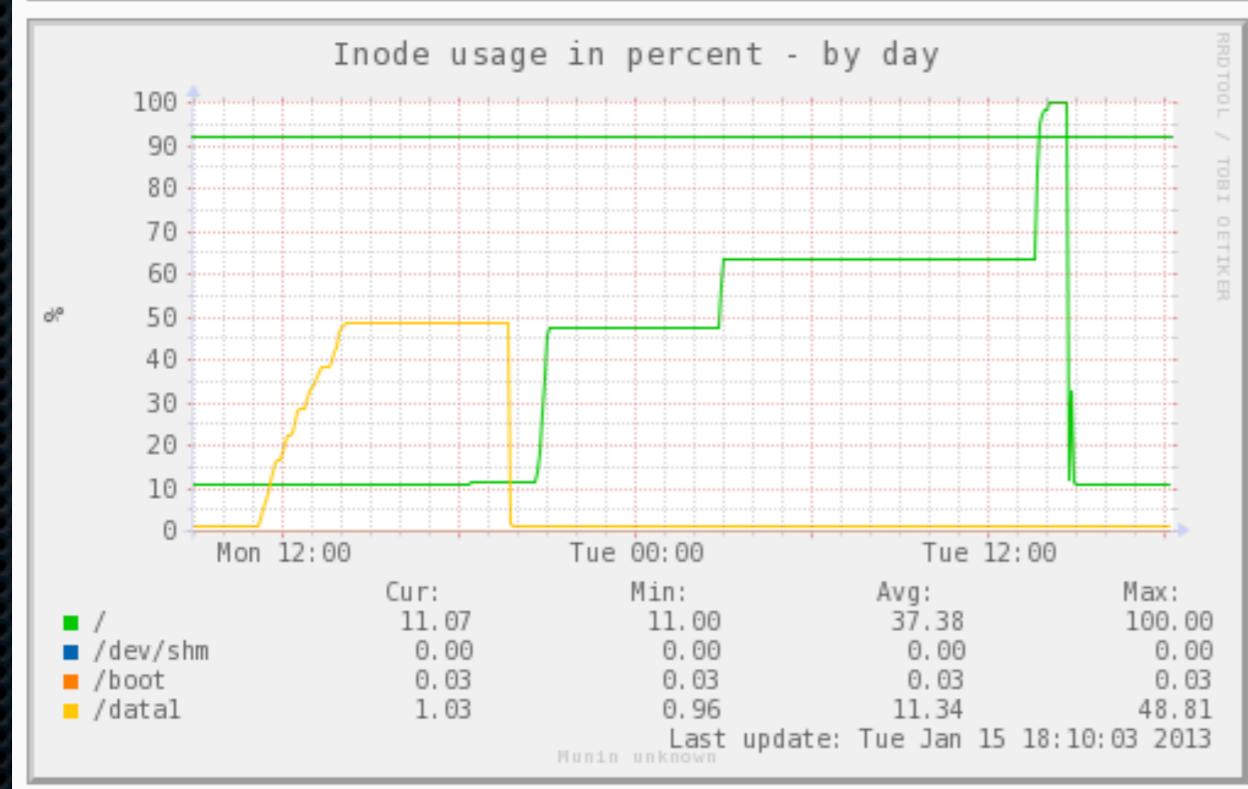
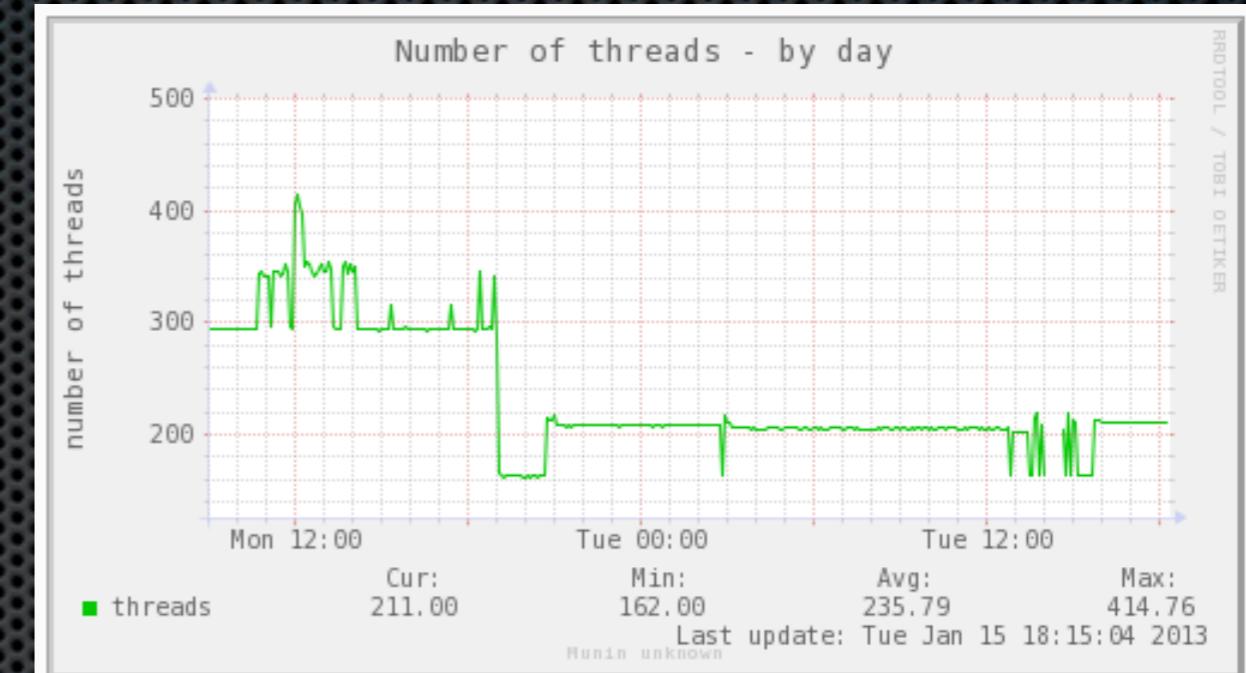
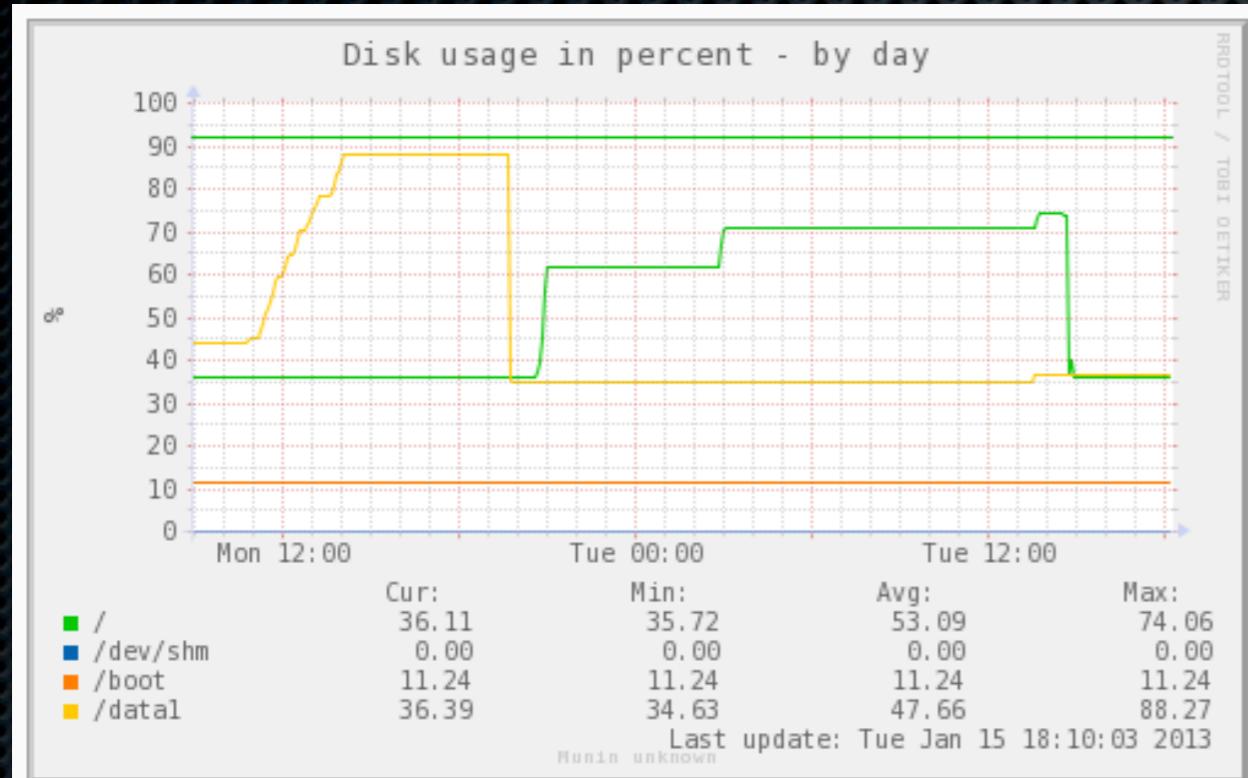
결론

- Shark가 Hive를 대체할 수 있는가에 대해서 고민에서 시작
 - 그래서 TPC-H 데이터로 벤치마크한 것
- 모니터링과 디버깅이 너무 힘들다. 이제 좀 적응 한 것 같다.
 - 멈추는 현상도 발생하고 가끔 워커가 죽기도 하고 리커버리가 안되기도 하고 다양한 케이스를 만났다.
- Shark는 추천하지 않지만 Spark는 추천
- 하이브는 좀 더 느려도 성공적으로 끝나지만 얘들은 뭔가 변수가 많다. (제 실력이 부족한 변수가 치명적)

결론

- RDD는 인메모리로 잘 관리하지만 부수적으로 생기는 파일이 많다. (이건 다음 슬라이드에 ...)
 - 디스크 사용량이 꽤 많은 것 같다.
 - 자동으로 끝나면 지워주면 좋으면만.
- ...

뭔가 막 이상함... 빵구남.



질의 응답!

한국
E