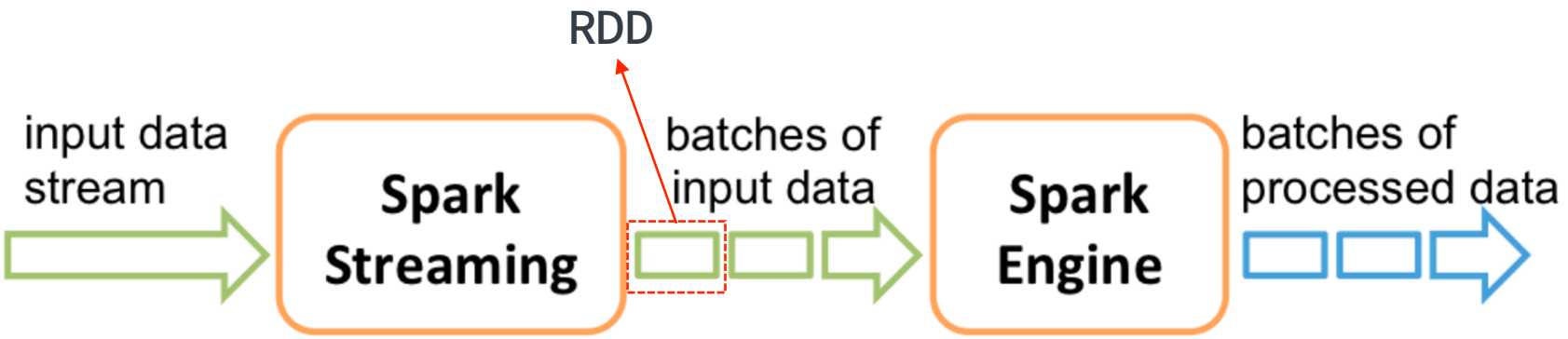
mobigen
Make The Best For Smart Business

Spark SQL 실시간 스트리밍 (Structured Streaming)

Spark Structured Streaming

Who am I?

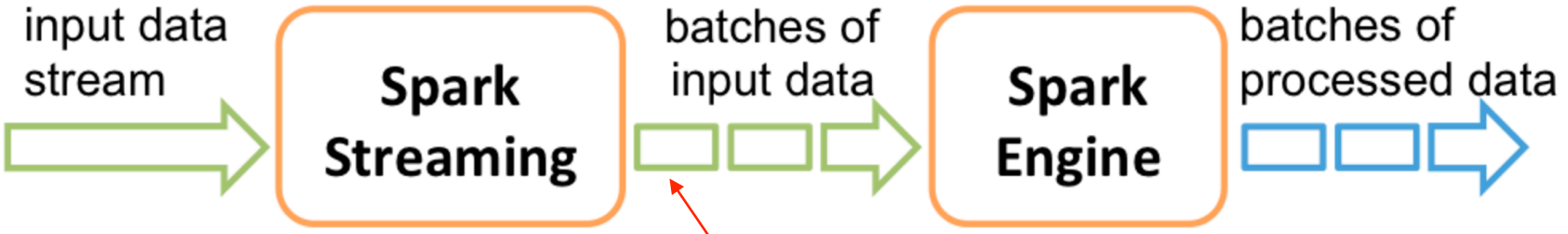
- 모비젠 선임 나부랭이
- 심각한 수포자
- 좁고 모난마음의 3년차 개발자
- 군인 (내년 2월까지 ㅠ_ㅠ)
- 열심히 Spark 관련하여 Github 에서 까부는중!



(Time-based) micro batch

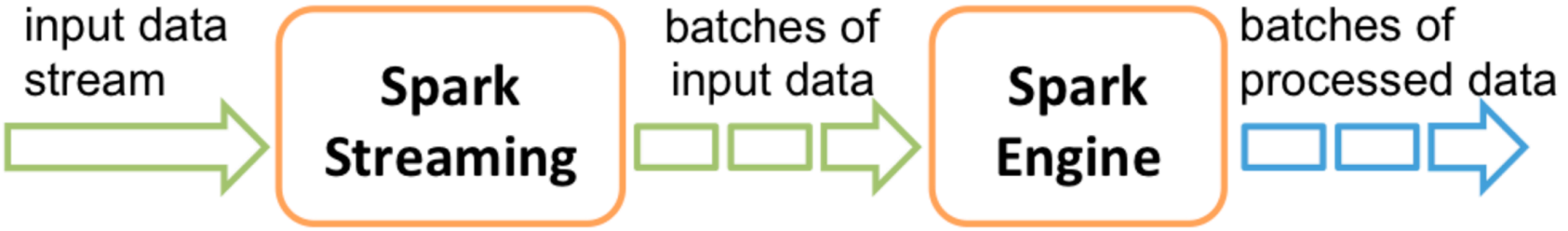
Record level streaming

http://www.slideshare.net/databricks/a-deep-dive-into-structured-streaming




```
// Create the context with a 1 second batch size
val sparkConf = new SparkConf().setAppName("NetworkWordCount")
val ssc = new StreamingContext(sparkConf, Seconds(1))

// Create a socket stream on target ip:port and count the
// words in input stream of \n delimited text (eg. generated by 'nc')
// Note that no duplication in storage level only for running locally.
// Replication necessary in distributed scenario for fault tolerance.
val lines = ssc.socketTextStream(args(0), args(1).toInt, StorageLevel.MEMORY_AND_DISK_SER)
val words = lines.flatMap(_.split(" "))
val wordCounts = words.map(x => (x, 1)).reduceByKey(_ + _)
wordCounts.print()
ssc.start()
ssc.awaitTermination()
```

http://www.slideshare.net/databricks/a-deep-dive-into-structured-streaming
https://github.com/apache/spark/blob/master/examples/src/main/scala/org/apache/spark/examples/streaming/NetworkWordCount.scala




```
// Create the context with a 1 second batch size
val sparkConf = new SparkConf().setAppName("NetworkWordCount")
val ssc = new StreamingContext(sparkConf, Seconds(1))

// Create a socket stream on target ip:port and count the
// words in input stream of \n delimited text (eg. generated by 'nc')
// Note that no duplication in storage level only for running locally.
// Replication necessary in distributed scenario for fault tolerance.
val lines = ssc.socketTextStream(args(0), args(1).toInt, StorageLevel.MEMORY_AND_DISK_SER)
val words = lines.flatMap(_.split(" "))
val wordCounts = words.map(x => (x, 1)).reduceByKey(_ + _)
wordCounts.print()
ssc.start()
ssc.awaitTermination()
```

http://www.slideshare.net/databricks/a-deep-dive-into-structured-streaming
https://github.com/apache/spark/blob/master/examples/src/main/scala/org/apache/spark/examples/streaming/NetworkWordCount.scala

```
bin/run-example org.apache.spark.examples.streaming.NetworkWordCount localhost 9999
```

```
-------------------------------------------
Time: 1476195953000 ms
-------------------------------------------
(Hyukjin,1)
(Hey,,1)
(I,3)
(love,1)
(it,1)
(really,1)
(Kwon,1)
(am,2)
(so,1)
(too,1)
...

-------------------------------------------
Time: 1476195954000 ms
-------------------------------------------
(Hello,1)
(what?,1)
(know,1)
(You,1)
(world!,1)
```

Terminal 1

```
nc -lk 9999
```

```
Hey, I am Hyukjin Kwon and I am so active on Spark and I really love it too :)
You know what? Hello world!
```

Terminal 2

```
bin/run-example org.apache.spark.examples.streaming.NetworkWordCount localhost 9999
```

```
-------------------------------------------
Time: 1476195953000 ms
-------------------------------------------
(Hyukjin,1)
(Hey,,1)
(I,3)
(love,1)
(it,1)
(really,1)
(Kwon,1)
(am,2)
(so,1)
(too,1)
...
```

```
-------------------------------------------
Time: 1476195954000 ms
-------------------------------------------
(Hello,1)
(what?,1)
(know,1)
(You,1)
(world!,1)
```

이 시점에서 과거 데이터를 바라보고 batch 처리 하기가 까다로워..

(StreamingContext.remember(...) 등으로 가능은 하지만..)

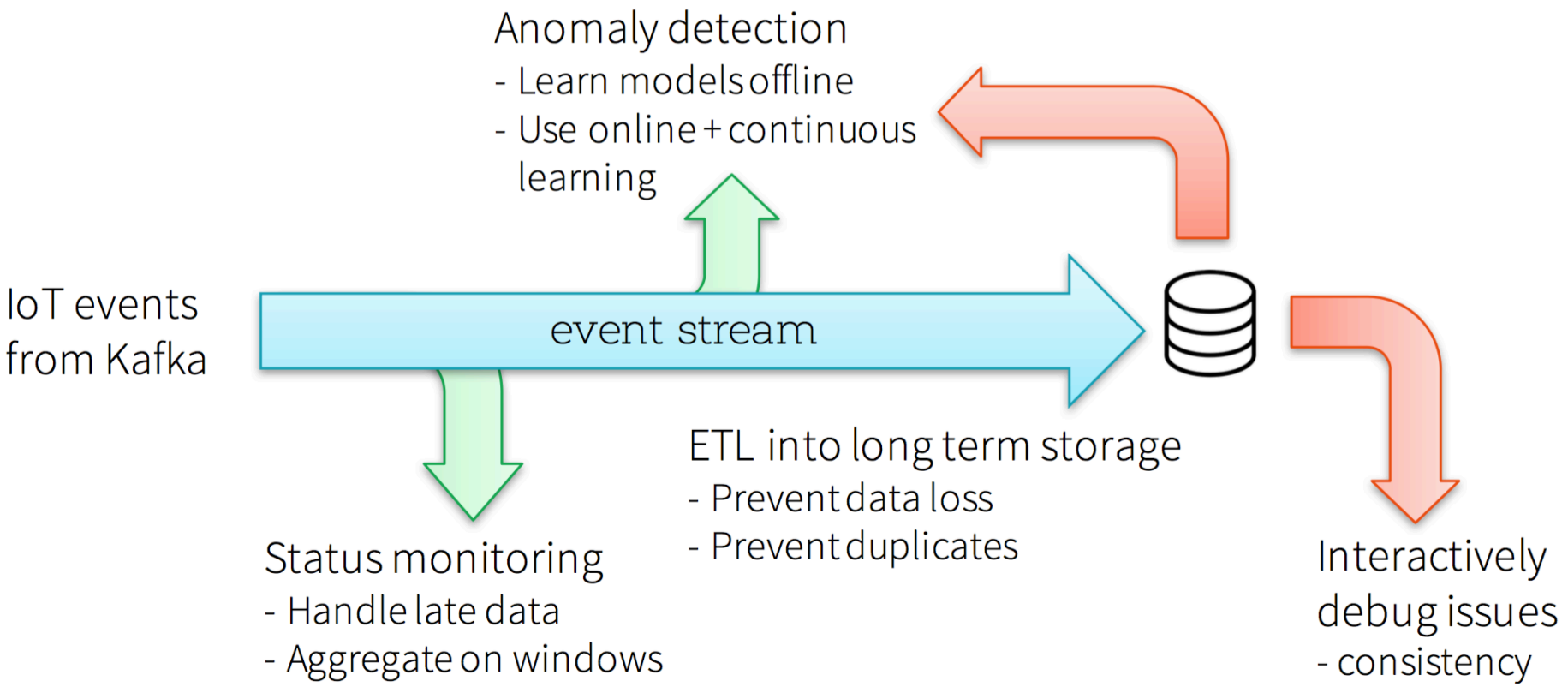
Terminal 1

```
nc -lk 9999
```

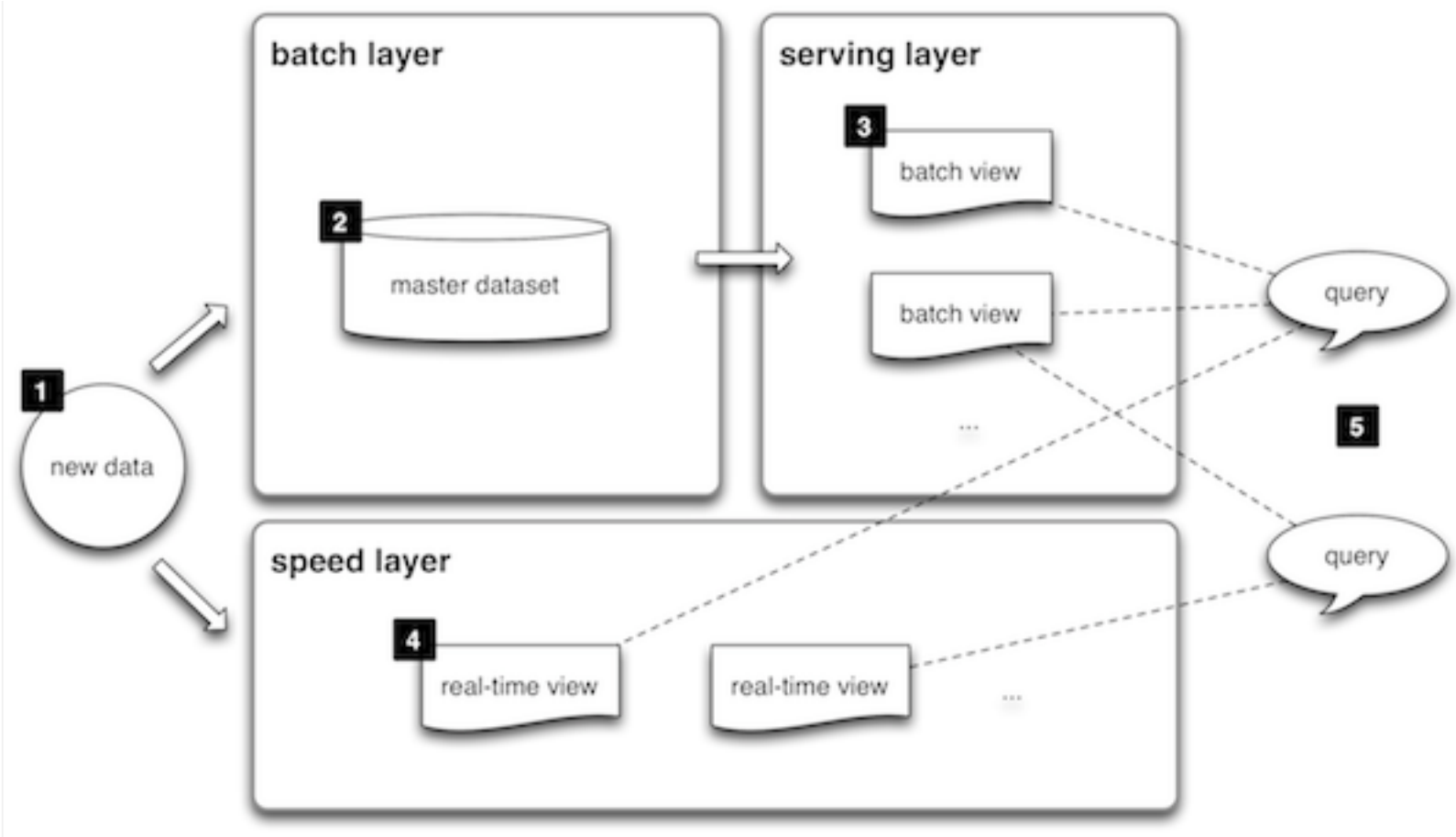
```
Hey, I am Hyukjin Kwon and I am so active on Spark and I really love it too :)
You know what? Hello world!
```

Terminal 2

Use case: IoT Device Monitoring



http://www.slideshare.net/databricks/a-deep-dive-into-structured-streaming



Lambda Architecture

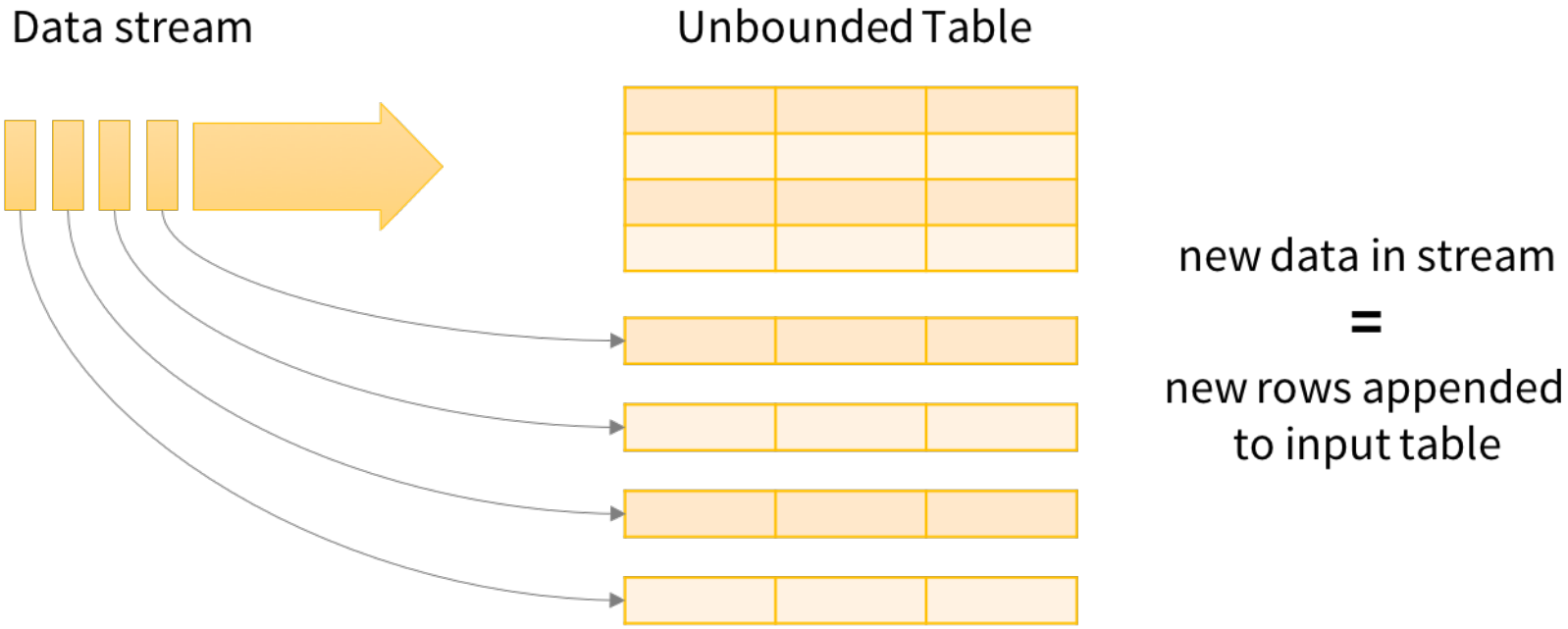
http://lambda-architecture.net/

그 외..

- Dstream과 함께 Dataframe/DataSet API 쓰기가 불편하다거나
- Computation optimisation이 힘들다거나..
- Fault tolerance 관리가 힘들다거나..
- Sink 만들기가 힘들다거나…

쓰다가 보면.. 부들부들!!

Structured Streaming Overview



Data stream as an unbounded Input Table

Batch 도 OK! Streaming 처리도 OK!

https://databricks.com/blog/2016/07/28/structured-streaming-in-apache-spark.html

```scala
import spark.implicits._

// Create DataFrame representing the stream of input lines from connection to host:port
val lines = spark.readStream
  .format("socket")
  .option("host", host)
  .option("port", port)
  .load()

// Split the lines into words
val words = lines.as[String].flatMap(_.split(" "))

// Generate running word count
val wordCounts = words.groupBy("value").count()

// Start running the query that prints the running counts to the console
val query = wordCounts.writeStream
  .outputMode("complete")
  .format("console")
  .start()

query.awaitTermination()
```

단순히 word count임

FYI - Processing interval은 default 로 0 초임... 이건 끝나면 바로실행하고 끝나면 바로 실행하고 를 의미함..

https://github.com/apache/spark/blob/master/examples/src/main/scala/org/apache/spark/examples/sql/streaming/StructuredNetworkWordCount.scala

```
./bin/run-example org.apache.spark.examples.sql.streaming.StructuredNetworkWordCount localhost 9999
```

```
-------------------
Batch: 0
-------------------
+-------+-----+
|  value|count|
+-------+-----+
|   love|    1|
|    too|    1|
|Hyukjin|    1|
|   Kwon|    1|
| active|    1|
|     on|    1|
|   Hey,|    1|
|     it|    1|
| really|    1|
|    and|    2|
|     :)|    1|
|      I|    3|
|  Spark|    1|
|     so|    1|
|     am|    2|
+-------+-----+
```

```
-------------------
Batch: 1
-------------------
+-------+-----+
|  value|count|
+-------+-----+
|  what?|    1|
|   love|    1|
|    too|    1|
|Hyukjin|    1|
|   Kwon|    1|
| world!|    1|
| active|    1|
|     on|    1|
|   Hey,|    1|
|     it|    1|
|    You|    1|
|  Hello|    1|
| really|    1|
|    and|    2|
|     :)|    1|
|      I|    3|
|  Spark|    1|
|     so|    1|
|     am|    2|
|   know|    1|
+-------+-----+
```

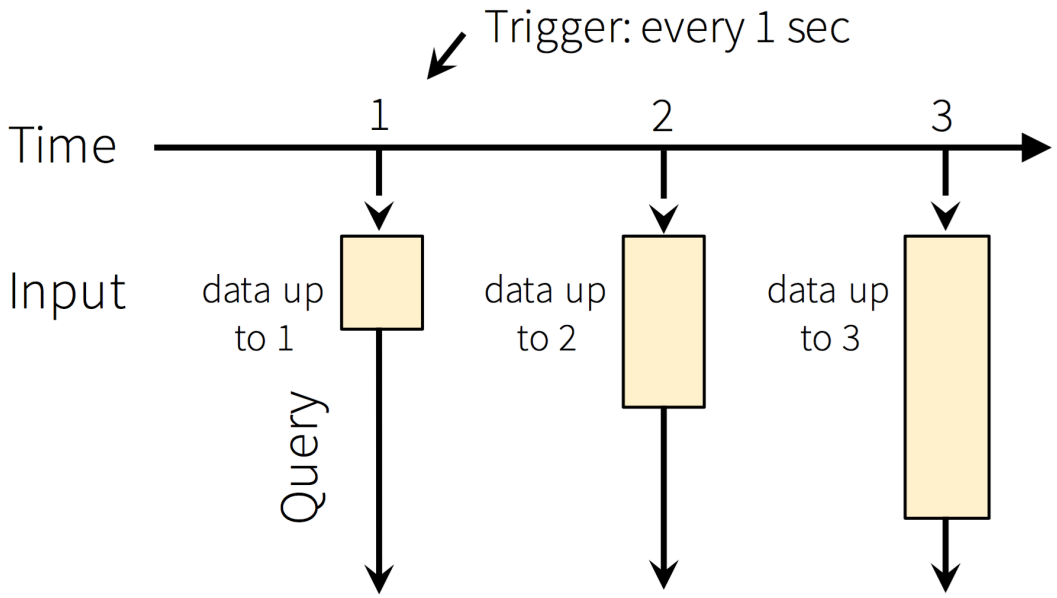
과거 데이터랑 합쳐서 batch로 처리됨 (굿굿)

Terminal 1

```
nc -lk 9999
```

```
Hey, I am Hyukjin Kwon and I am so active on Spark and I really love it too :)
You know what? Hello world!
```

Terminal 2

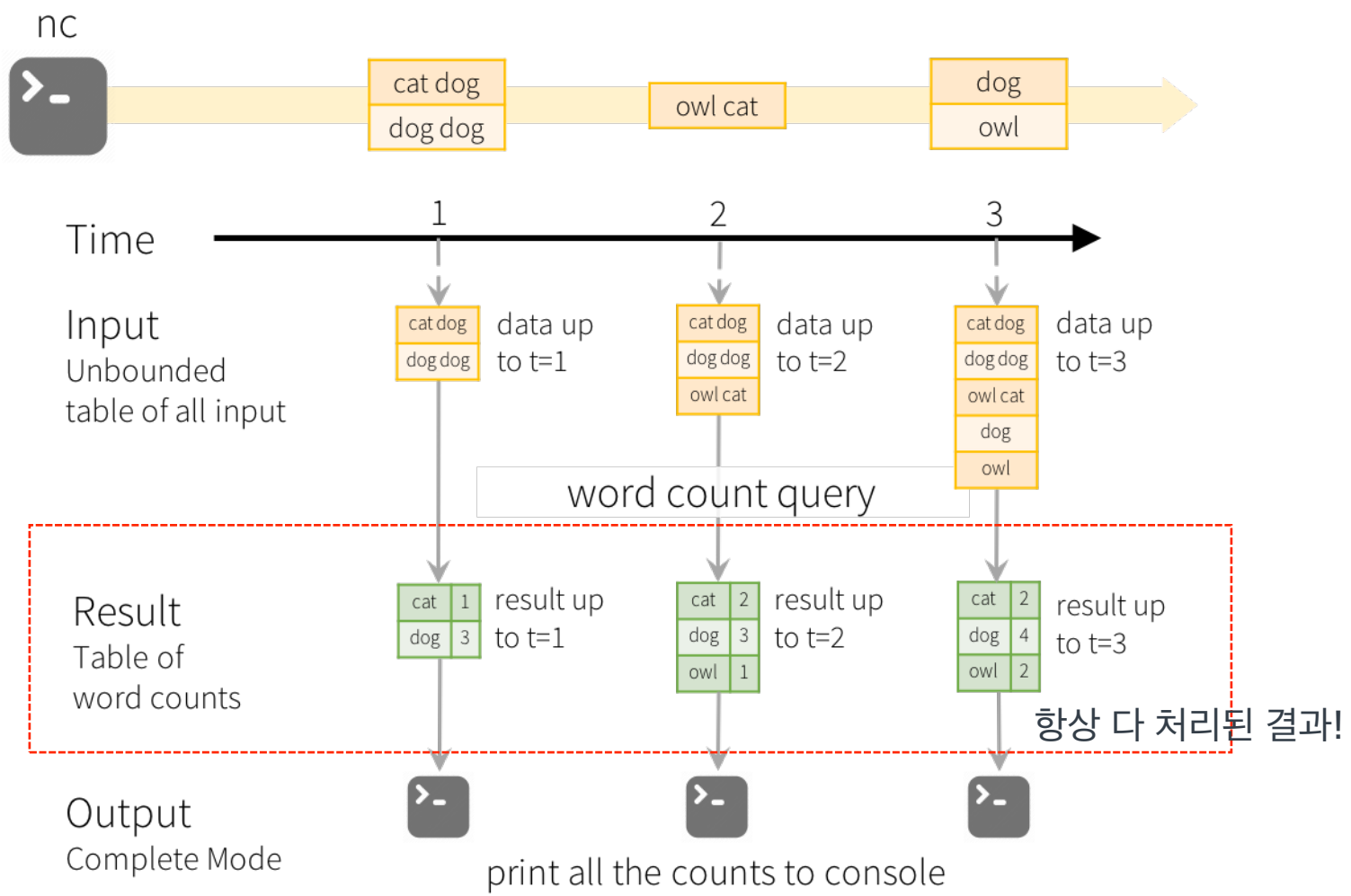


Complete Mode: 항상 업데이트 된 output의 전체

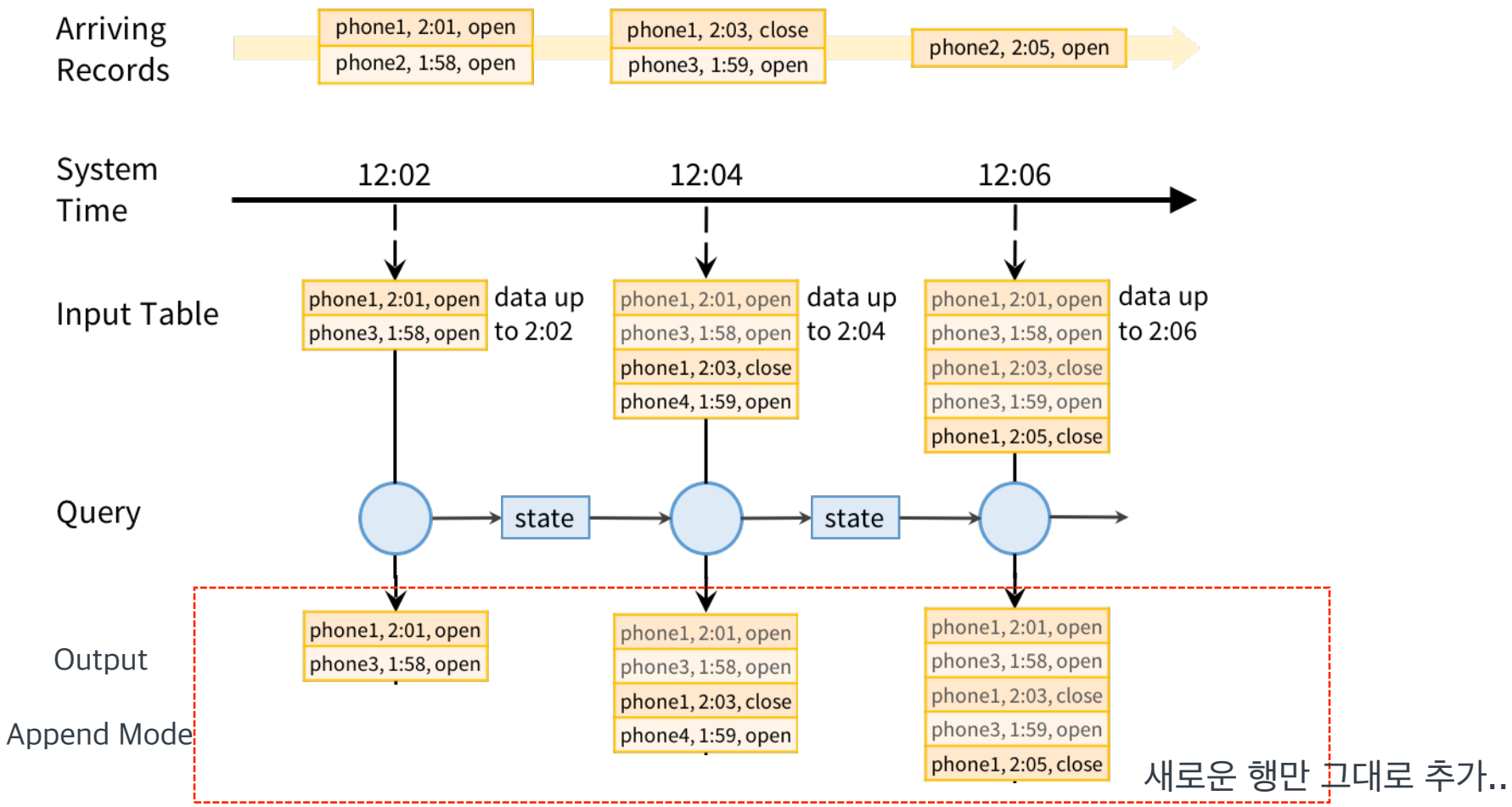
Append Mode: 추가되는 놈들만 update 없이..

Update Mode: 중복 되는 놈들은 업데이트 하면서..

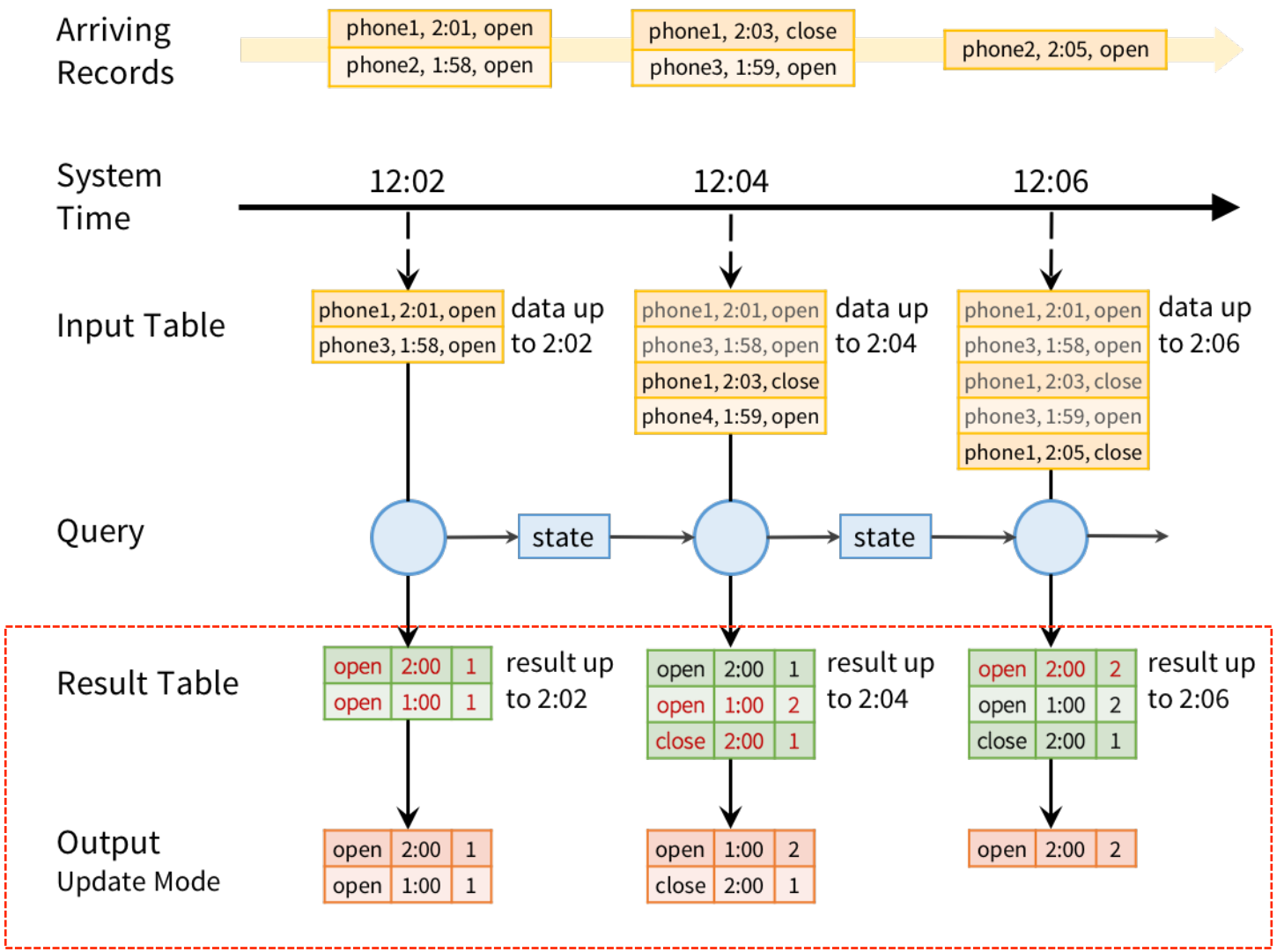
http://www.slideshare.net/databricks/a-deep-dive-into-structured-streaming



https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html



https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html



업데이트 된 놈들만!

https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

```
input = spark.read
     .format("json")
     .load("source-path")
```

Read from Json file

```
result = input
     .select("device", "signal")
     .where("signal > 15")
```

Select some devices

```
result.write
     .format("parquet")
     .save("dest-path")
```

Write to parquet file

기존 DataFrame load 하는 코드에서..

http://www.slideshare.net/databricks/a-deep-dive-into-structured-streaming

```
input = spark.read
      .format("json")
      .load("source-path")
```

Read from Json file

```
result = input
      .select("device", "signal")
      .where("signal > 15")
```

Select some devices

```
result.write
      .format("parquet")
      .save("dest-path")
```

Write to parquet file

이렇게만 바꿔주면 스트림을 읽음!

```
input = spark.read
      .format("json")
      .stream("source-path")
```

Read from Json file stream
Replace `load()` with `stream()`

```
result = input
      .select("device", "signal")
      .where("signal > 15")
```

Select some devices
Code does not change

```
result.write
      .format("parquet")
      .startStream("dest-path")
```

Write to Parquet file stream
Replace `save()` with `startStream()`

http://www.slideshare.net/databricks/a-deep-dive-into-structured-streaming

```
input.avg("signal")
```

```
input.groupBy("device-type")
     .avg("signal")
```

Aggregation OK!

```
input.groupBy(
        $"device-type",
        window($"event-time-col", "10 min"))
     .avg("signal")
```

Windows Aggregation OK! (기존 Dstream에서 안됐었음!)

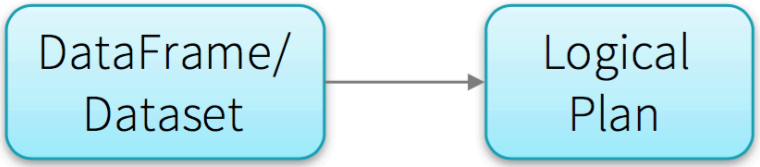
```
kafkaDataset = spark.read
 .kafka("iot-updates")
 .stream()

staticDataset = ctxt.read
 .jdbc("jdbc://", "iot-device-info")

joinedDataset =
 kafkaDataset.join(
   staticDataset, "device-type")
```

Stream과 Static table간의 조인도 가능!!

http://www.slideshare.net/databricks/a-deep-dive-into-structured-streaming

Structured Streaming Internals

Batch Execution on Spark SQL



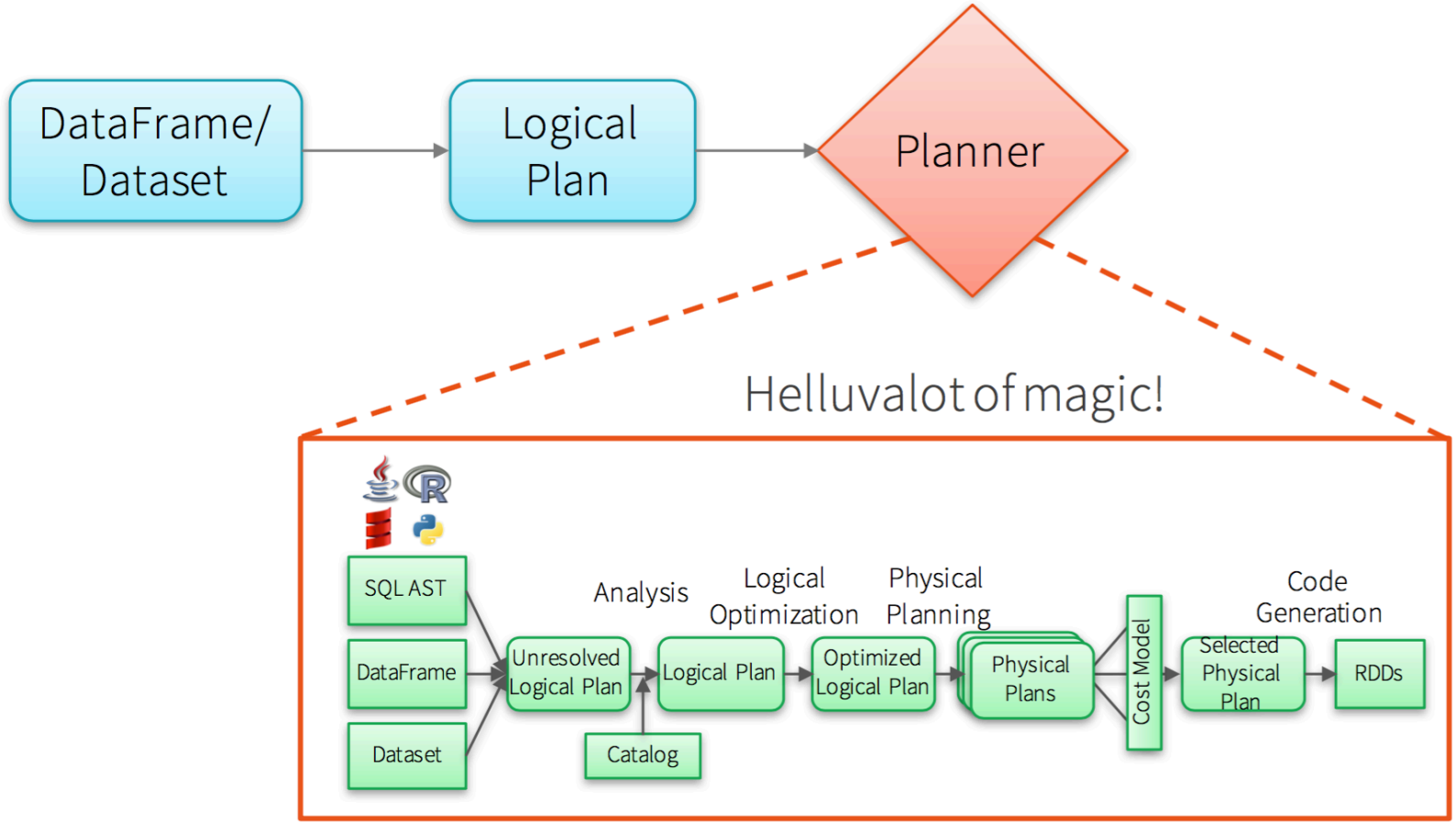
Abstract representation of query

기존 Dataframe과 마찬가지로 플랜 생성

```
== Parsed Logical Plan ==
Aggregate [value#8], [value#8, count(1) AS count#13L]
+- SerializeFromObject [staticinvoke(class org.apache.spark.unsafe.types.UTF8String, StringType, fromString, input[0, java.lang.
   +- MapPartitions <function1>, obj#7: java.lang.String
      +- DeserializeToObject cast(value#48 as string).toString, obj#6: java.lang.String
         +- Project [value#46 AS value#48]
            +- LocalRelation [value#46]
```

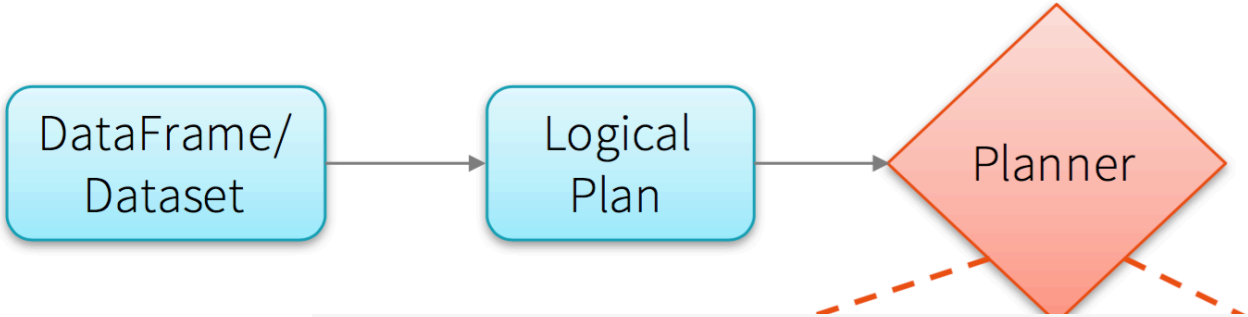
http://www.slideshare.net/databricks/a-deep-dive-into-structured-streaming

Batch Execution on Spark SQL



http://www.slideshare.net/databricks/a-deep-dive-into-structured-streaming

Batch Execution on Spark SQL



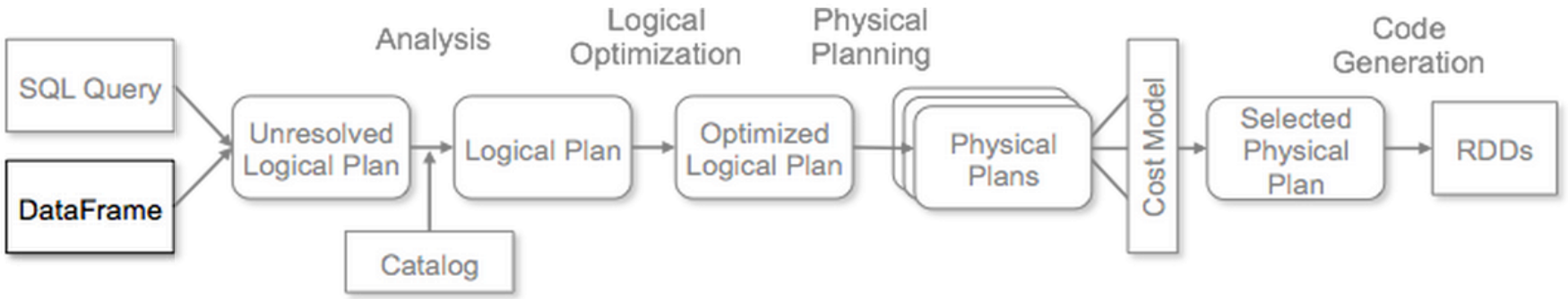

```
== Analyzed Logical Plan ==
value: string, count: bigint
Aggregate [value#8], [value#8, count(1) AS count#13L]
+- SerializeFromObject [staticinvoke(class org.apache.spark.unsafe.types.UTF8String, StringType, fromString, input[0, java.lang.
   +- MapPartitions <function1>, obj#7: java.lang.String
      +- DeserializeToObject cast(value#48 as string).toString, obj#6: java.lang.String
         +- Project [value#46 AS value#48]
            +- LocalRelation [value#46]

== Optimized Logical Plan ==
Aggregate [value#8], [value#8, count(1) AS count#13L]
+- SerializeFromObject [staticinvoke(class org.apache.spark.unsafe.types.UTF8String, StringType, fromString, input[0, java.lang.
   +- MapPartitions <function1>, obj#7: java.lang.String
      +- DeserializeToObject value#46.toString, obj#6: java.lang.String
         +- LocalRelation [value#46]

== Physical Plan ==
*HashAggregate(keys=[value#8], functions=[count(1)], output=[value#8, count#13L])
+- Exchange hashpartitioning(value#8, 200)
   +- *HashAggregate(keys=[value#8], functions=[partial_count(1)], output=[value#8, count#27L])
      +- *SerializeFromObject [staticinvoke(class org.apache.spark.unsafe.types.UTF8String, StringType, fromString, input[0, jav
         +- MapPartitions <function1>, obj#7: java.lang.String
            +- DeserializeToObject value#46.toString, obj#6: java.lang.String
               +- LocalTableScan [value#46]
```

http://www.slideshare.net/databricks/a-deep-dive-into-structured-streaming

Dataframe을 생성하는 단계에서 각 File의 Schema를 읽는다.

```
val region_df = sqlContext.read.load("hdfs://localhost:9000/user/vagrant/parquet/region")
region_df.registerTempTable("region")

val nation_df = sqlContext.read.load("hdfs://localhost:9000/user/vagrant/parquet/nation")
nation_df.registerTempTable("nation")
```

Read Schema



http://www.slideshare.net/databricks/building-a-modern-application-with-dataframes-52776940

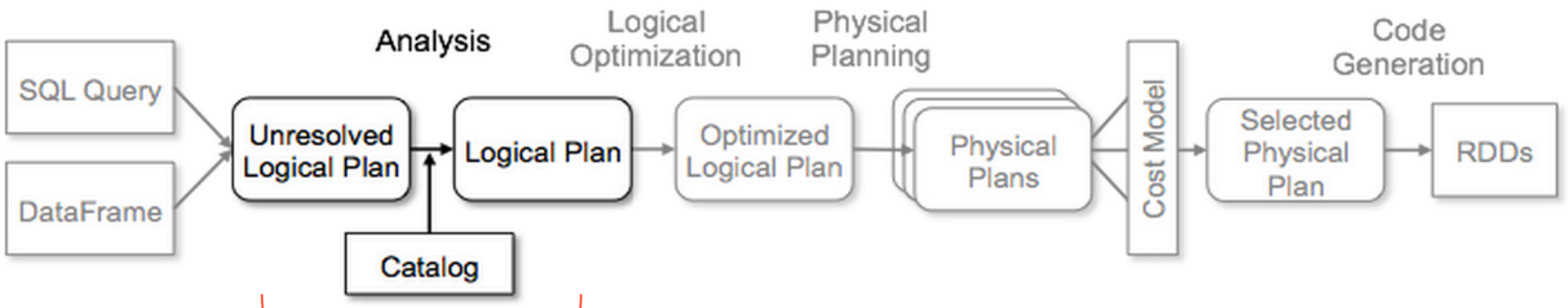
Register Table 에서 Catalog에 이 Relation과 Table 정보들을 저장한다.

```
val region_df = sqlContext.read.load("hdfs://localhost:9000/user/vagrant/parquet/region")
region_df.registerTempTable("region")

val nation_df = sqlContext.read.load("hdfs://localhost:9000/user/vagrant/parquet/nation")
nation_df.registerTempTable("nation")
```

Register Schema & Relation

```
== Parsed Logical Plan ==
Relation[r_regionkey#0,r_name#1,r_comment#2] org.apache.spark.sql.parquet.ParquetRelation2@a74ca45f
```

```
== Analyzed Logical Plan ==
r_regionkey: int, r_name: string, r_comment: string
Relation[r_regionkey#0,r_name#1,r_comment#2] org.apache.spark.sql.parquet.ParquetRelation2@a74ca45f
```



내 테이블 이름은 nation/region 임 그리고 스키마도있음

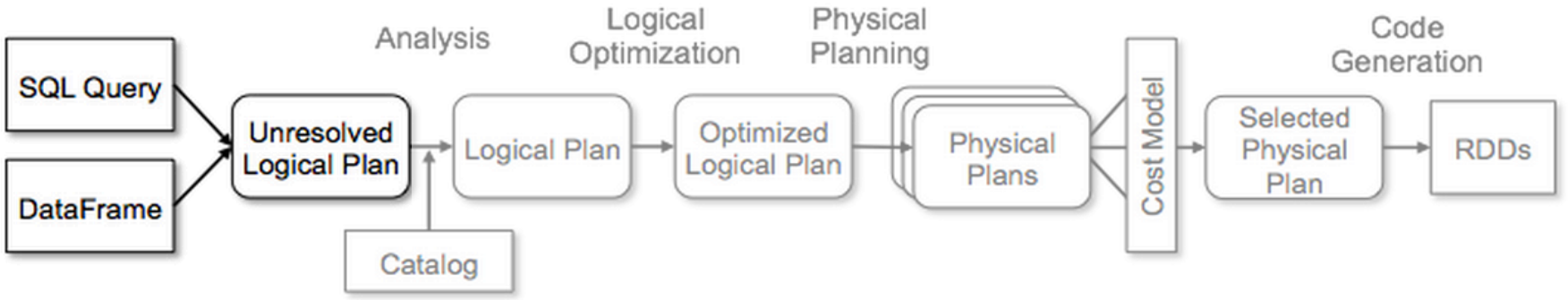
http://www.slideshare.net/databricks/building-a-modern-application-with-dataframes-52776940

해당된 Datafame에 Query를 할 때, Query를 먼저 Parsing 한다.

```
...
conf.set("spark.sql.parquet.filterPushdown", "true")
...
val result_df = sqlContext.sql(
  """
     SELECT n_name, n_nationkey
     FROM nation n join region r
     WHERE n.n_regionkey = r.r_regionkey and r.r_name = 'ASIA'
     ORDER BY n_name

  """
)
val result = result_df.collect()
```

Query Parsing

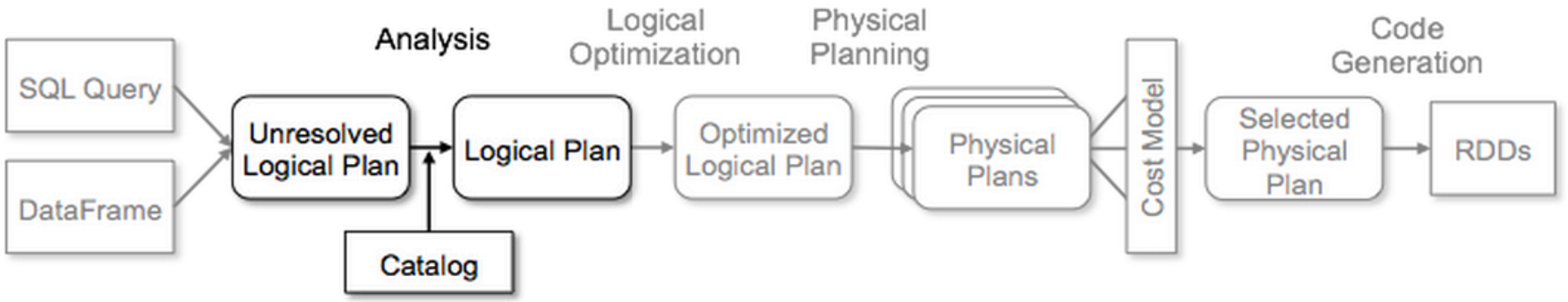
```
== Parsed Logical Plan ==
'Sort ['n_name ASC], true
 'Project ['n_name,'n_nationkey]
  'Filter (('n.n_regionkey = 'r.r_regionkey) && ('r.r_name = ASIA))
   'Join Inner, None
    'UnresolvedRelation [nation], Some(n)
    'UnresolvedRelation [region], Some(r)
```



http://www.slideshare.net/databricks/building-a-modern-application-with-dataframes-52776940

Analyze 단계에서 전단계에서 등록된 Catalog를 lookup후 Relation을 알아낸다.

```
== Parsed Logical Plan ==
'Sort ['n_name ASC], true
 'Project ['n_name,'n_nationkey]
  'Filter (('n.n_regionkey = 'r.r_regionkey) && ('r.r_name = ASIA))
   'Join Inner, None
    'UnresolvedRelation [nation], Some(n)
    'UnresolvedRelation [region], Some(r)

== Analyzed Logical Plan ==
n_name: string, n_nationkey: int
Sort [n_name#4 ASC], true
 Project [n_name#4,n_nationkey#3]
  Filter ((n_regionkey#5 = r_regionkey#0) && (r_name#1 = ASIA))
   Join Inner, None
    Subquery n
     Subquery nation
      Relation[n_nationkey#3,n_name#4,n_regionkey#5,n_comment#6] org.apache.spark.sql.parquet.ParquetRelation2@f3f0b73b
    Subquery r
     Subquery region
      Relation[r_regionkey#0,r_name#1,r_comment#2] org.apache.spark.sql.parquet.ParquetRelation2@a74ca45f
```



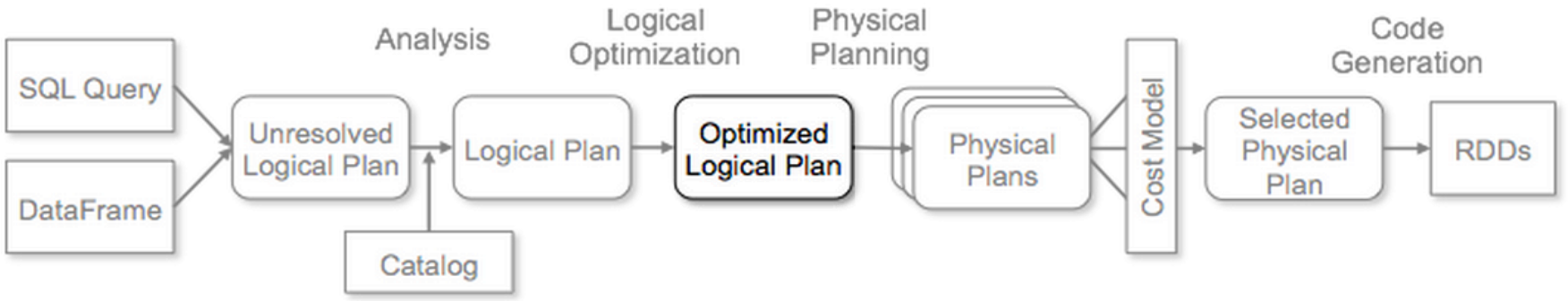
http://www.slideshare.net/databricks/building-a-modern-application-with-dataframes-52776940

Predicate push down 등등 optmization 과정을 거쳐 plan 최적화

```
== Optimized Logical Plan ==
Sort [n_name#4 ASC], true
 Project [n_name#4,n_nationkey#3]
  Join Inner, Some((n_regionkey#5 = r_regionkey#0))
   Project [n_name#4,n_nationkey#3,n_regionkey#5]
    Relation[n_nationkey#3,n_name#4,n_regionkey#5,n_comment#6] org.apache.spark.sql.parquet.ParquetRelation2@f3f0b73b
   Project [r_regionkey#0]
    Filter (r_name#1 = ASIA)
     Relation[r_regionkey#0,r_name#1,r_comment#2] org.apache.spark.sql.parquet.ParquetRelation2@a74ca45f
```

n_name, n_nationkey, n_regionkey

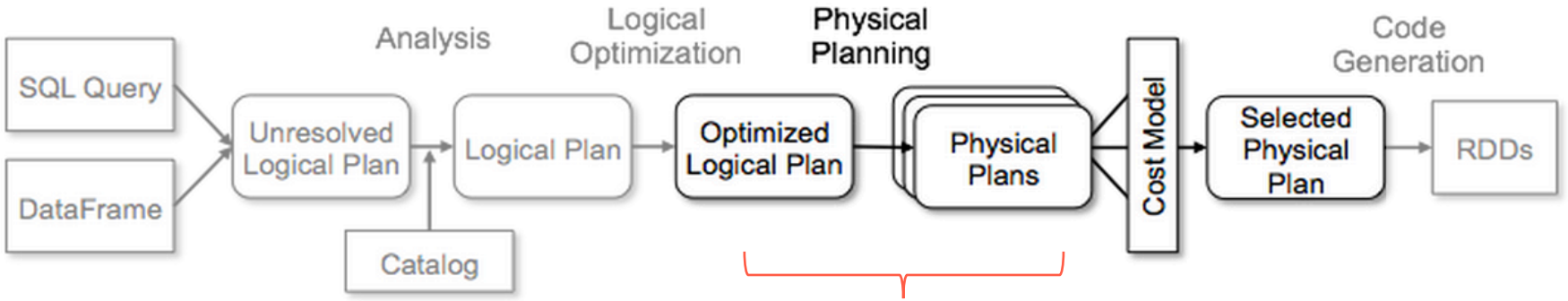
r_regionkey, r_name
r_name = 'ASIA'



http://www.slideshare.net/databricks/building-a-modern-application-with-dataframes-52776940

Relation이 Push down 된 Filter와 Project의 조건을 알게되면 그에 맞는 PhysicalRDD로 바꿔 준다. (현재 2.0 기준으로는 FileScanRDD 임..)

```
== Optimized Logical Plan ==
Sort [n_name#4 ASC], true
 Project [n_name#4,n_nationkey#3]
  Join Inner, Some((n_regionkey#5 = r_regionkey#0))
   Project [n_name#4,n_nationkey#3,n_regionkey#5]
    Relation[n_nationkey#3,n_name#4,n_regionkey#5,n_comment#6] org.apache.spark.sql.parquet.ParquetRelation2
   Project [r_regionkey#0]
    Filter (r_name#1 = ASIA)
     Relation[r_regionkey#0,r_name#1,r_comment#2] org.apache.spark.sql.parquet.ParquetRelation2

== Physical Plan ==
Sort [n_name#4 ASC], true
 Exchange (RangePartitioning 200)
  Project [n_name#4,n_nationkey#3]
   BroadcastHashJoin [n_regionkey#5], [r_regionkey#0], BuildRight
    PhysicalRDD [n_name#4,n_nationkey#3,n_regionkey#5], MapPartitionsRDD[1]
    Project [r_regionkey#0]
     Filter (r_name#1 = ASIA)
      PhysicalRDD [r_regionkey#0,r_name#1], MapPartitionsRDD[3]
```

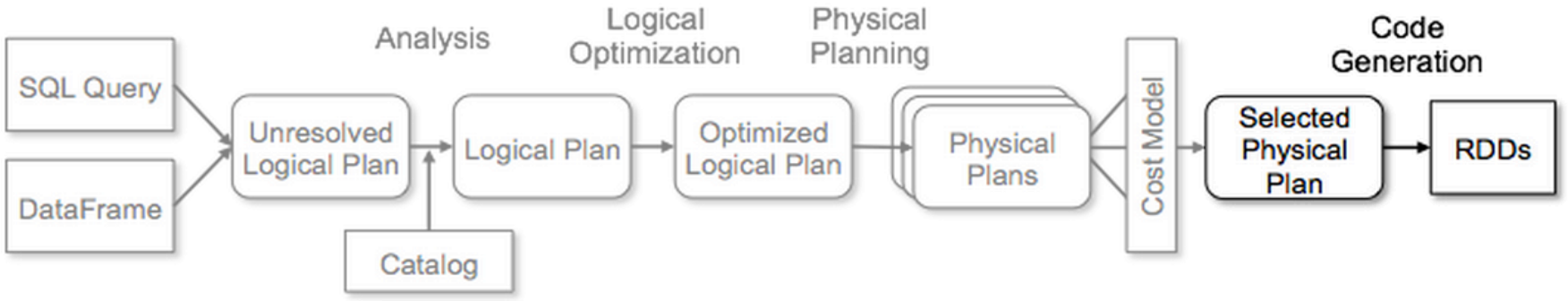


http://www.slideshare.net/databricks/building-a-modern-application-with-dataframes-52776940

실행! codegen/off-heap/encoding 등등..

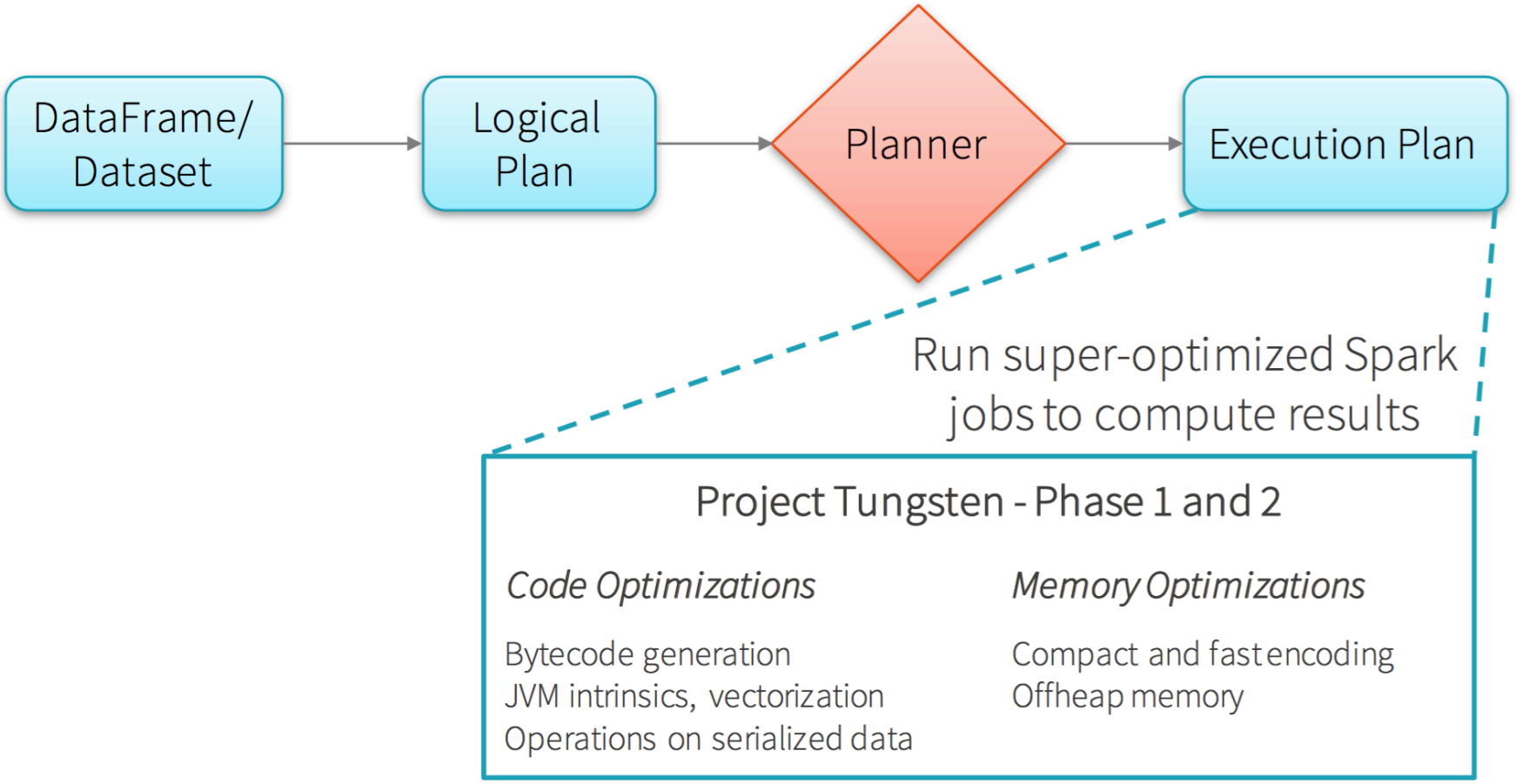
```
== Physical Plan ==
Sort [n_name#4 ASC], true
 Exchange (RangePartitioning 200)
  Project [n_name#4,n_nationkey#3]
   BroadcastHashJoin [n_regionkey#5], [r_regionkey#0], BuildRight
    PhysicalRDD [n_name#4,n_nationkey#3,n_regionkey#5], MapPartitionsRDD[1]
    Project [r_regionkey#0]
     Filter (r_name#1 = ASIA)
      PhysicalRDD [r_regionkey#0,r_name#1], MapPartitionsRDD[3]
```

```
long count = 0;
for (ss_item_sk in store_sales) {
  if (ss_item_sk == 1000) {
    count += 1;
  }
}
```



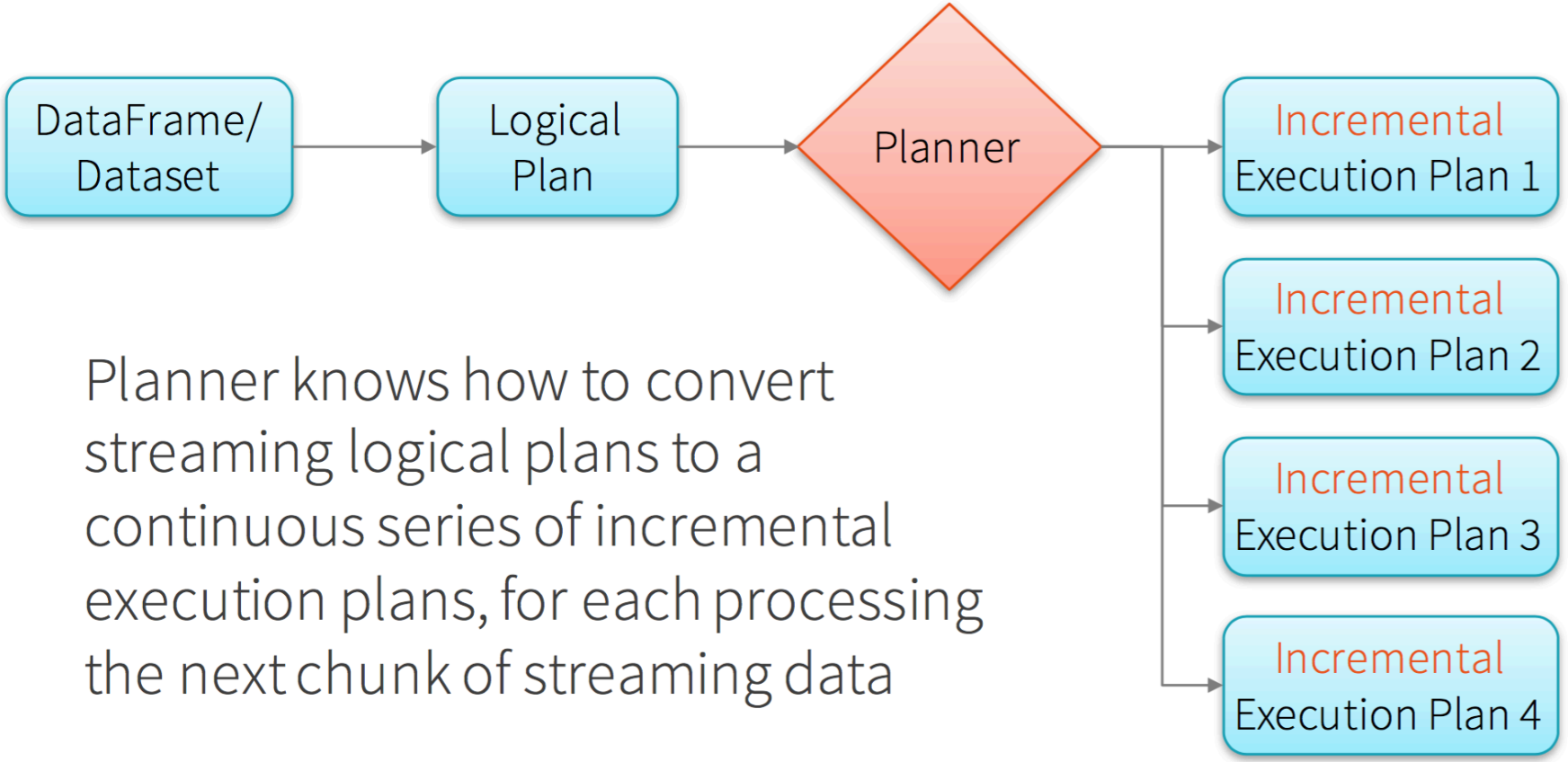
http://www.slideshare.net/databricks/building-a-modern-application-with-dataframes-52776940

Batch Execution on Spark SQL



http://www.slideshare.net/databricks/a-deep-dive-into-structured-streaming

Continuous Incremental Execution



Planner knows how to convert streaming logical plans to a continuous series of incremental execution plans, for each processing the next chunk of streaming data

http://www.slideshare.net/databricks/a-deep-dive-into-structured-streaming

ConsoleSink.scala

```
  override def addBatch(batchId: Long, data: DataFrame): Unit = synchronized {
    val batchIdStr = if (batchId <= lastBatchId) {
      s"Rerun batch: $batchId"
    } else {
      lastBatchId = batchId
      s"Batch: $batchId"
    }

    // scalastyle:off println
    println("-------------------------------------------")
    println(batchIdStr)
    println("-------------------------------------------")
    // scalastyle:off println
    data.sparkSession.createDataFrame(
      data.sparkSession.sparkContext.parallelize(data.collect()), data.schema)
      .show(numRowsToShow, isTruncated)
  }
```

사실 각 Batch가 그냥 DataFrame으로 들어와서 처리함..

```
class ConsoleSinkProvider extends StreamSinkProvider with DataSourceRegister {
  def createSink(
      sqlContext: SQLContext,
      parameters: Map[String, String],
      partitionColumns: Seq[String],
      outputMode: OutputMode): Sink = {
    new ConsoleSink(parameters)
  }

  def shortName(): String = "console"
}
```

사실상 Dstream을 한번 감싸놓고 이쁘게 사용 할 수 있게 해논거인듯..

```scala
// Convert RDDs of the words DStream to DataFrame and run SQL query
words.foreachRDD { (rdd: RDD[String], time: Time) =>
  // Get the singleton instance of SparkSession
  val spark = SparkSessionSingleton.getInstance(rdd.sparkContext.getConf)
  import spark.implicits._

  // Convert RDD[String] to RDD[case class] to DataFrame
  val wordsDataFrame = rdd.map(w => Record(w)).toDF()

  // Creates a temporary view using the DataFrame
  wordsDataFrame.createOrReplaceTempView("words")

  // Do word count on table using SQL and print it
  val wordCountsDataFrame =
    spark.sql("select word, count(*) as total from words group by word")
  println(s"========= $time =========")
  wordCountsDataFrame.show()
}
```

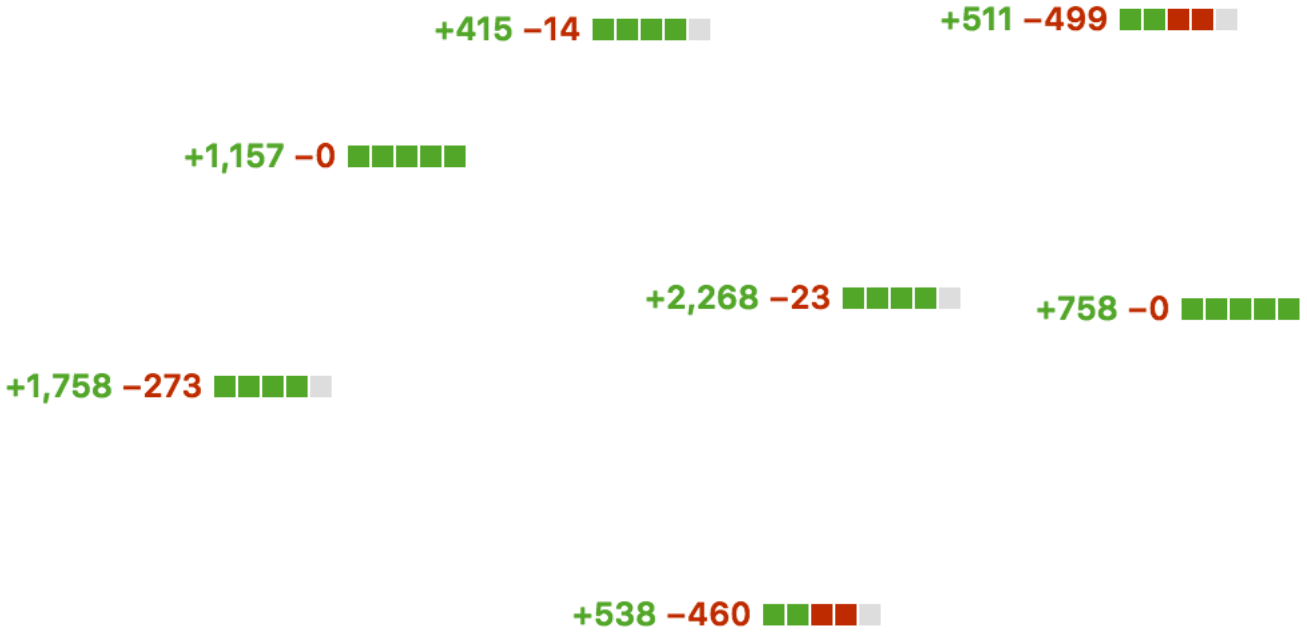
Dstream

```scala
// Generate running word count
val wordCounts = words.groupBy("value").count()

// Start running the query that prints the running counts to the console
val query = wordCounts.writeStream
  .outputMode("complete")
  .format("console")
  .start()

query.awaitTermination()
```

Structured Streaming

Can / Can't!

역변하는중..



Append mode with non-aggregation queries

```
input.select("device", "signal")
      .write
      .outputMode("append")
      .format("parquet")
      .startStream("dest-path")
```

Complete mode with aggregation queries

```
input.agg(count("*"))
      .write
      .outputMode("complete")
      .format("parquet")
      .startStream("dest-path")
```

Update Mode 는 엄슴.. 안됨... -_-..

Append mode랑 aggregation도 안됨.. flatMap도.. 암것도 안됨.. 부들부들..

```
org.apache.spark.sql.AnalysisException: Append output mode not supported when there are streaming aggregations
 on streaming DataFrames/DataSets;
    at org.apache.spark.sql.catalyst.analysis.UnsupportedOperationChecker$.org$apache$spark
       $sql$catalyst$analysis$UnsupportedOperationChecker$$throwError(UnsupportedOperationChecker.scala:173)
```

http://www.slideshare.net/databricks/a-deep-dive-into-structured-streaming

Sink	Supported Output Modes	Usage	Fault-tolerant
File Sink (only parquet in Spark 2.0)	Append	`writeStream .format("parquet") .start()`	Yes
Foreach Sink	All modes	`writeStream .foreach(...) .start()`	Depends on ForeachWriter implementation
Console Sink	Append, Complete	`writeStream .format("console") .start()`	No
Memory Sink	Append, Complete	`writeStream .format("memory") .queryName("table") .start()`	No

Parquet 으로 밖에 못 씀.. ㅠㅠ

https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

Spark Improvement Proposals

- Industry leaders on stage making fun of Spark's streaming model

- Users saying they chose Flink because it was technically superior and they couldn't get any answers on the Spark mailing lists

The way structured streaming has played out has shown that there are significant technical blind spots (myself included).

결론은 그닥 새련된 방법이 아니고.. 안전성이 글쌔...

But 정확하게 쉽게 scalability가 있듬

website now, recommends Apache Spark for most anything. For streaming in particular, there is a lot of confusion because many of the concepts aren't well-defined (e.g. what is "at least once", etc), and it's also a crowded space. But Spark Streaming prioritizes a few things that it does very well: correctness (you can easily tell what the app will do, and it does the same thing despite failures), ease of programming (which also requires correctness), and scalability. We should of course both

http://mail-archives.apache.org/mod_mbox/spark-dev/201610.mbox/%3CCAKWX9VXRXgLtsiRnx3Evxg-K5zEttJXLU8=OVwZsTBr34TASig@mail.gmail.com%3E

감사합니다.