

法律声明

■ 本课件包括演示文稿、示例、代码、题库、视频和声音等内容，北风网和讲师拥有完全知识产权；只限于善意学习者在本课程使用，不得在课程范围外向任何第三方散播。任何其他人或者机构不得盗版、复制、仿造其中的创意和内容，我们保留一切通过法律手段追究违反者的权利。

■ 课程详情请咨询

◆ 微信公众号：北风教育

◆ 官方网址：<http://www.ibeifeng.com/>



大数据内存计算框架Spark



主讲人：Gerry

上海育创网络科技有限公司



课程要求

■ 课上课下 “九字” 真言

- ◆ 认真听，善摘录，勤思考
- ◆ **多温故，乐实践**，再发散

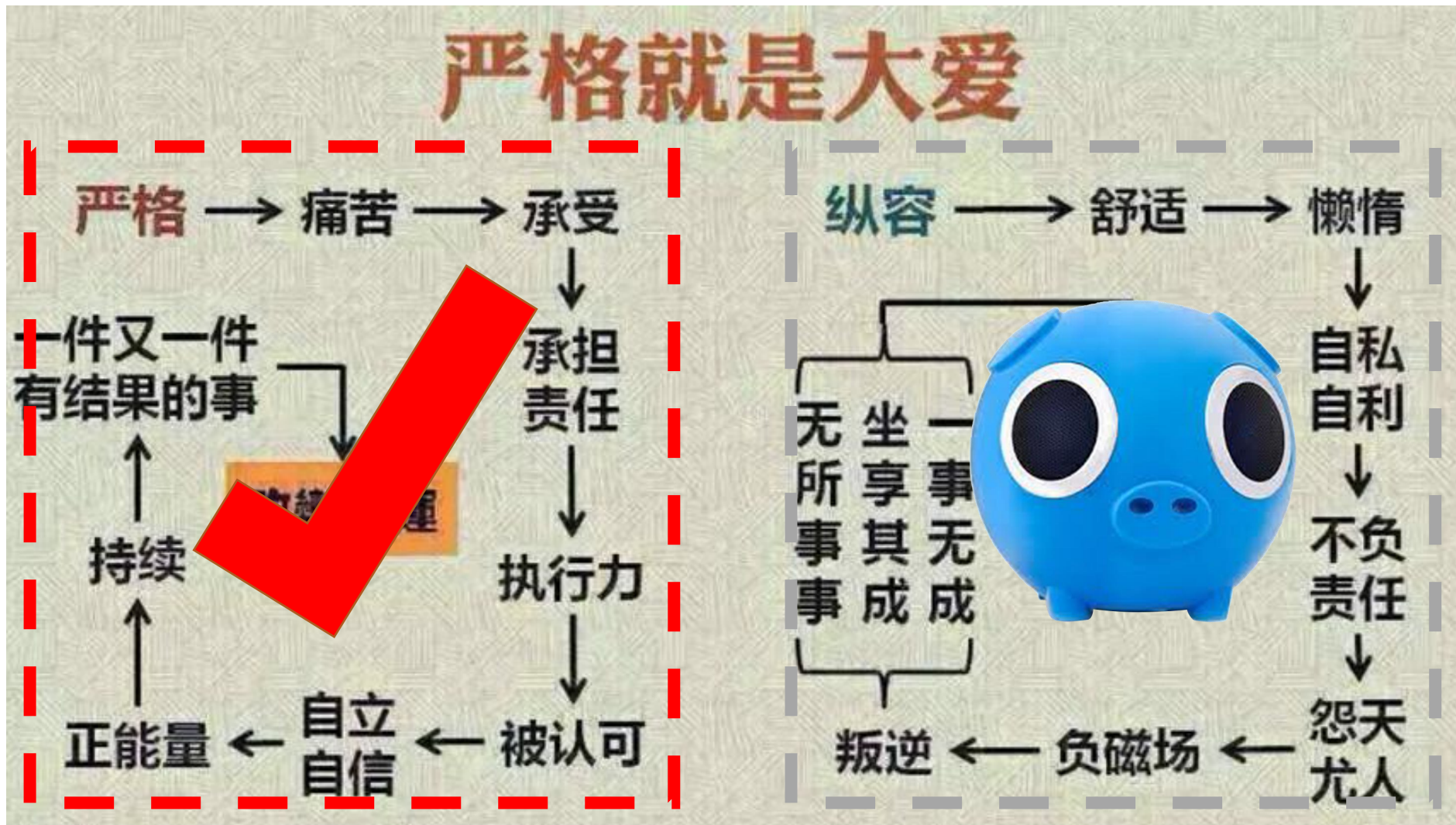
■ 四不原则

- ◆ **不懒散惰性，不迟到早退**
- ◆ **不请假旷课，不拖延作业**

■ 一点注意事项

- ◆ 违反 “四不原则”，不包就业和推荐就业

严格是大爱

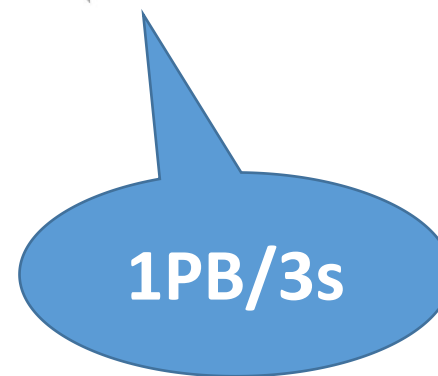
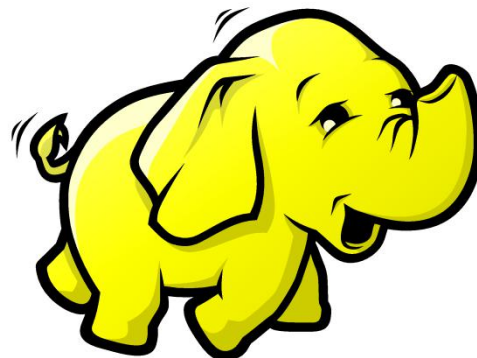


寄语

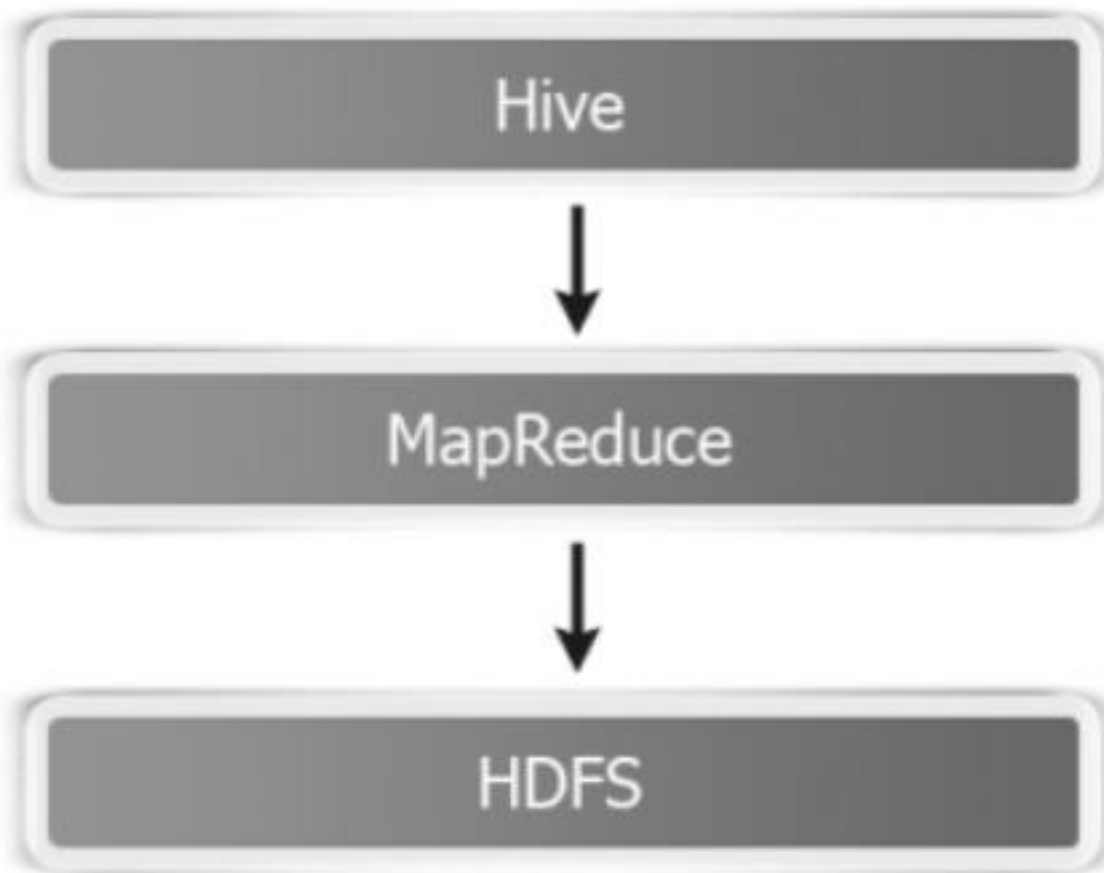


做别人不愿做的事，
做别人不敢做的事，
做别人做不到的事。

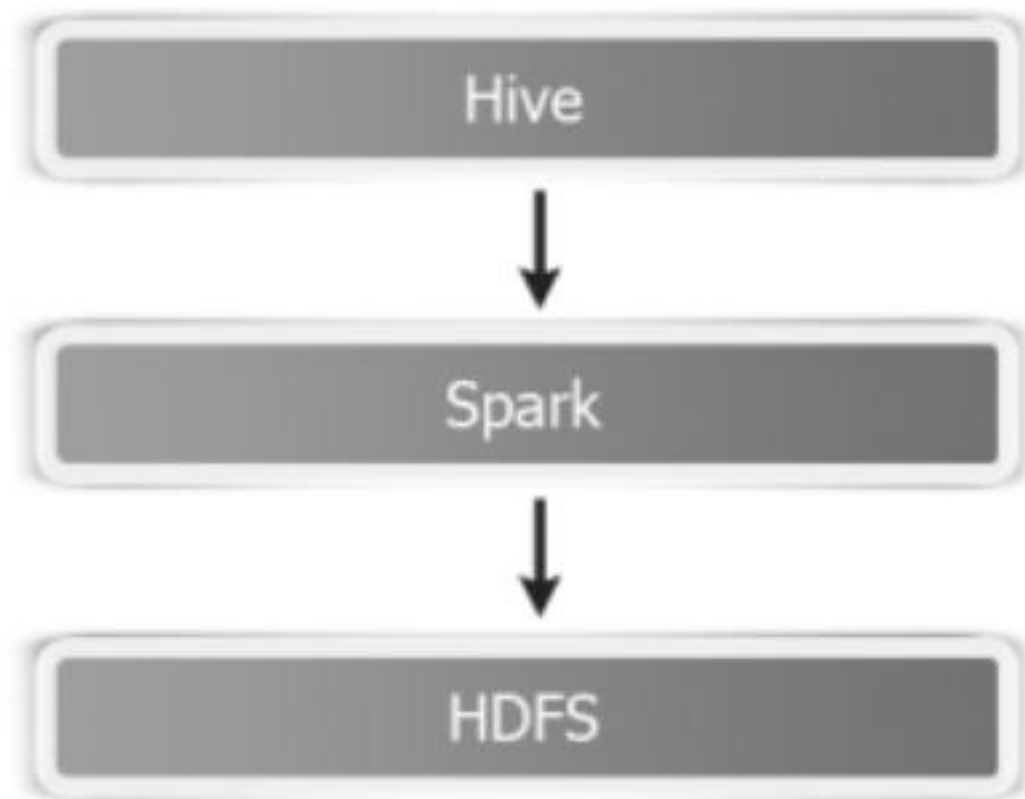
We want to use SQL in the Hadoop world



Hive



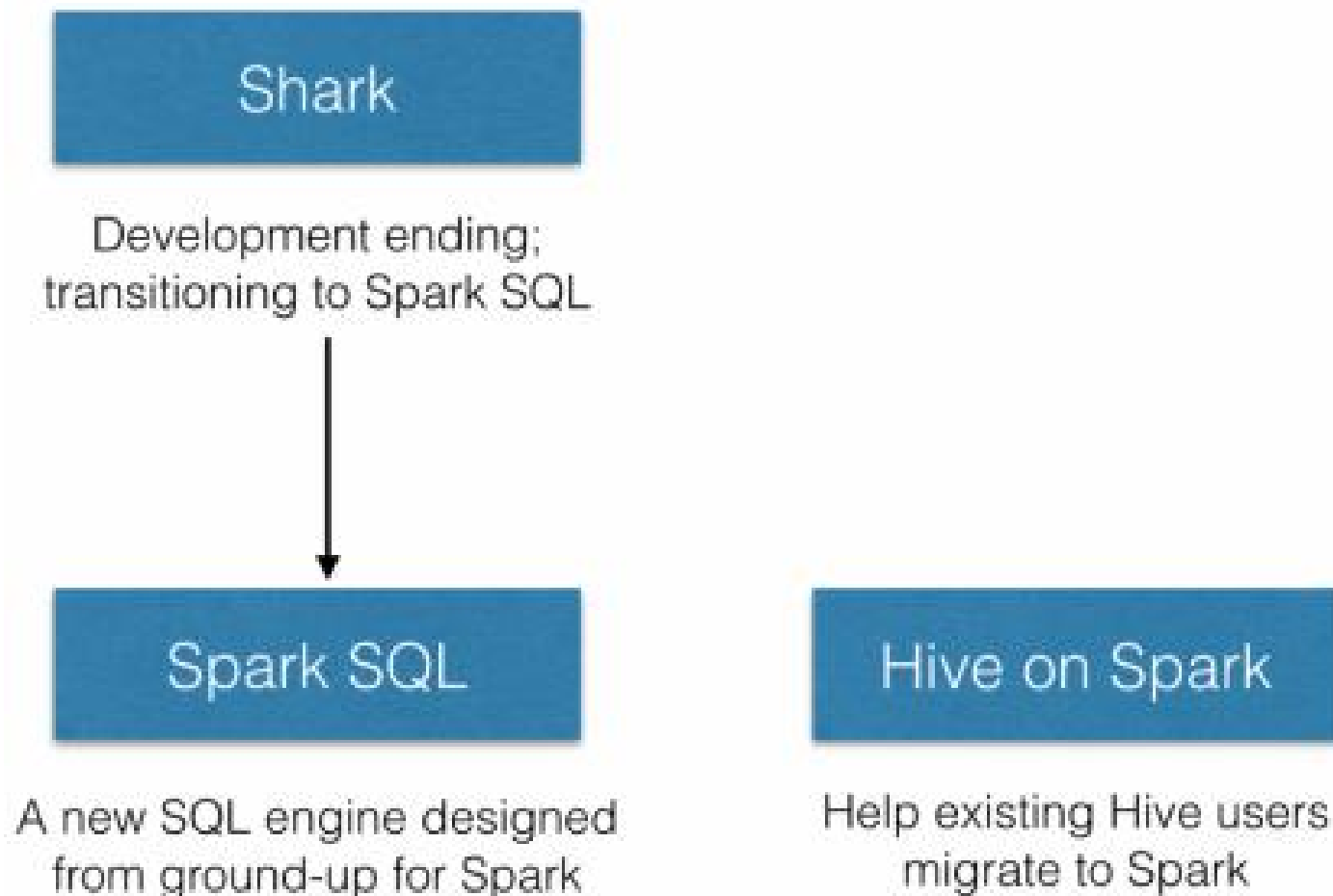
Shark



Shark

- 在三四年前，Hive可以说是SQL on Hadoop的唯一选择，**负责将SQL编译成可扩展的MapReduce作业**。鉴于Hive的性能以及与Spark的兼容，Shark项目由此而生。
- Shark即Hive on Spark，本质上是**通过Hive的HQL解析，把HQL翻译成Spark上的RDD操作，然后通过Hive的metadata获取数据库里的表信息，实际HDFS上的数据和文件，会由Shark获取并放到Spark上运算**。
- Shark的最大特性就是快和与Hive的完全兼容，且可以在shell模式下使用rdd2sql()这样的API，把HQL得到的结果集，继续在scala环境下运算，支持自己编写简单的机器学习或简单分析处理函数，对HQL结果进一步分析计算。
- <https://github.com/amplab/shark>

Spark SQL



- 在2014年7月1日的Spark Summit上，Databricks宣布终止对Shark的开发，将重点放到Spark SQL上。
- Databricks表示，**Spark SQL将涵盖Shark的所有特性，用户可以从Shark 0.9进行无缝的升级。**
- Databricks推广的Shark相关项目一共有两个，分别是Spark SQL和新的Hive on Spark (HIVE-7292)。

Shark, Spark SQL, Hive on Spark, and the future of SQL on Spark

<https://databricks.com/blog/2014/07/01/shark-spark-sql-hive-on-spark-and-the-future-of-sql-on-spark.html>

终止Shark的原因

- Databricks表示，Shark更多是对Hive的改造，替换了Hive的物理执行引擎，因此会有一个很快的速度。然而，不容忽视的是，Shark继承了大量的Hive代码，因此给优化和维护带来了大量的麻烦。

Shark built on the Hive codebase and achieved performance improvements by swapping out the physical execution engine part of Hive. While this approach enabled Shark users to speed up their Hive queries, Shark inherited a large, complicated code base from Hive that made it hard to optimize and maintain. As we moved to push the boundary of performance optimizations and integrating sophisticated analytics with SQL, we were constrained by the legacy that was designed for MapReduce.

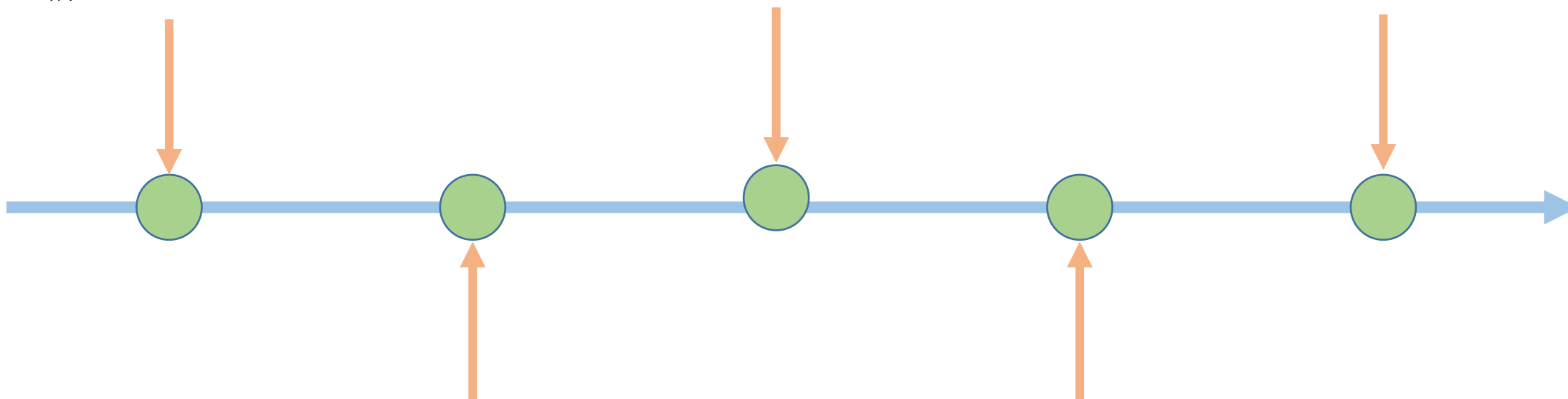
It is for this reason that we are ending development in Shark as a separate project and moving all our development resources to Spark SQL, a new component in Spark. We are applying what we learned in Shark to Spark SQL, designed from ground-up to leverage the power of Spark. This new approach enables us to innovate faster, and ultimately deliver much better experience and power to users.

Spark SQL Timeline

Shark: Hive on Spark, Spark1.0
发布之前

Spark SQL : **2015.4.13**, Spark 1.3.0
发布, **SparkSQL**为**Graduation**,
DataFrame诞生, 一统天下

Spark SQL: **2016.7.27**, Spark2.0发布,
Dataset=DataFrame[Row], 性能大幅度
提升, **Structure Streaming Process**诞生

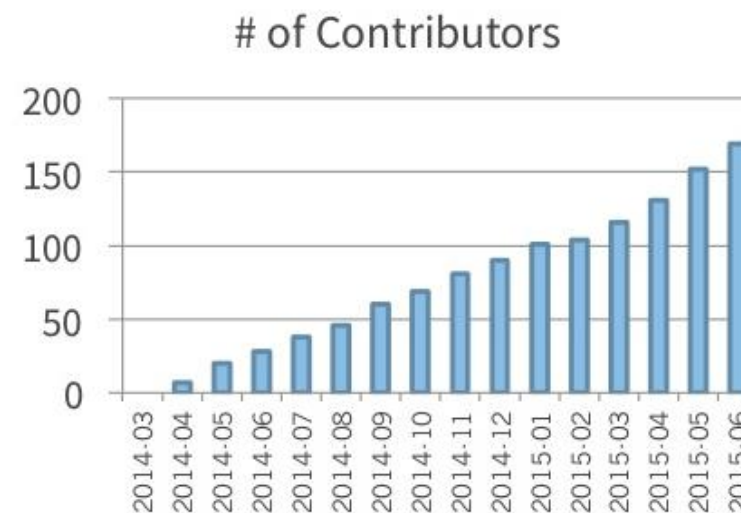
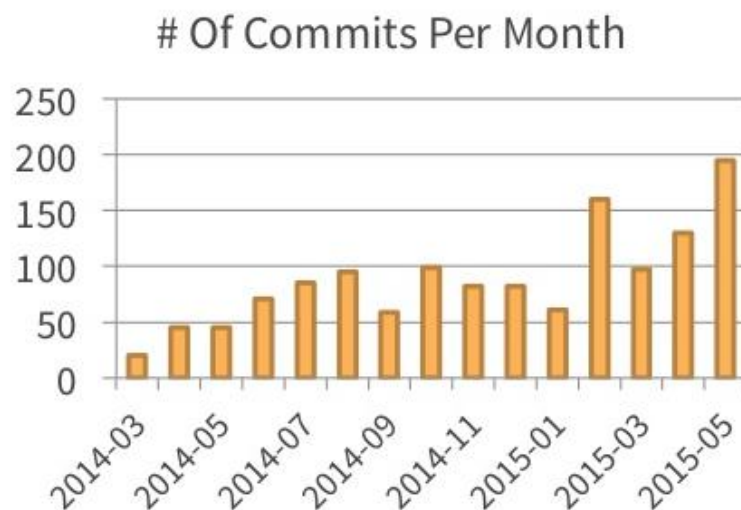


Spark SQL: **2014.5.30**, Spark1.0.0
发布, SchemaRDD诞生, **Shark** =
Spark SQL + Hive on Spark

Spark SQL: **2016.1.4**, Spark1.6.0发
布, Dataset出现, Tungsten功能第
一阶段和第二阶段功能实现, 主要
代码自动生成和内存管理。

Spark SQL

➤ 1. Part of the core distribution since Spark 1.0 (April 2014)



Graduated from Alpha in 1.3

Spark SQL

➤ 2. Runs SQL / HiveQL queries including UDFs UDAFs and SerDes.

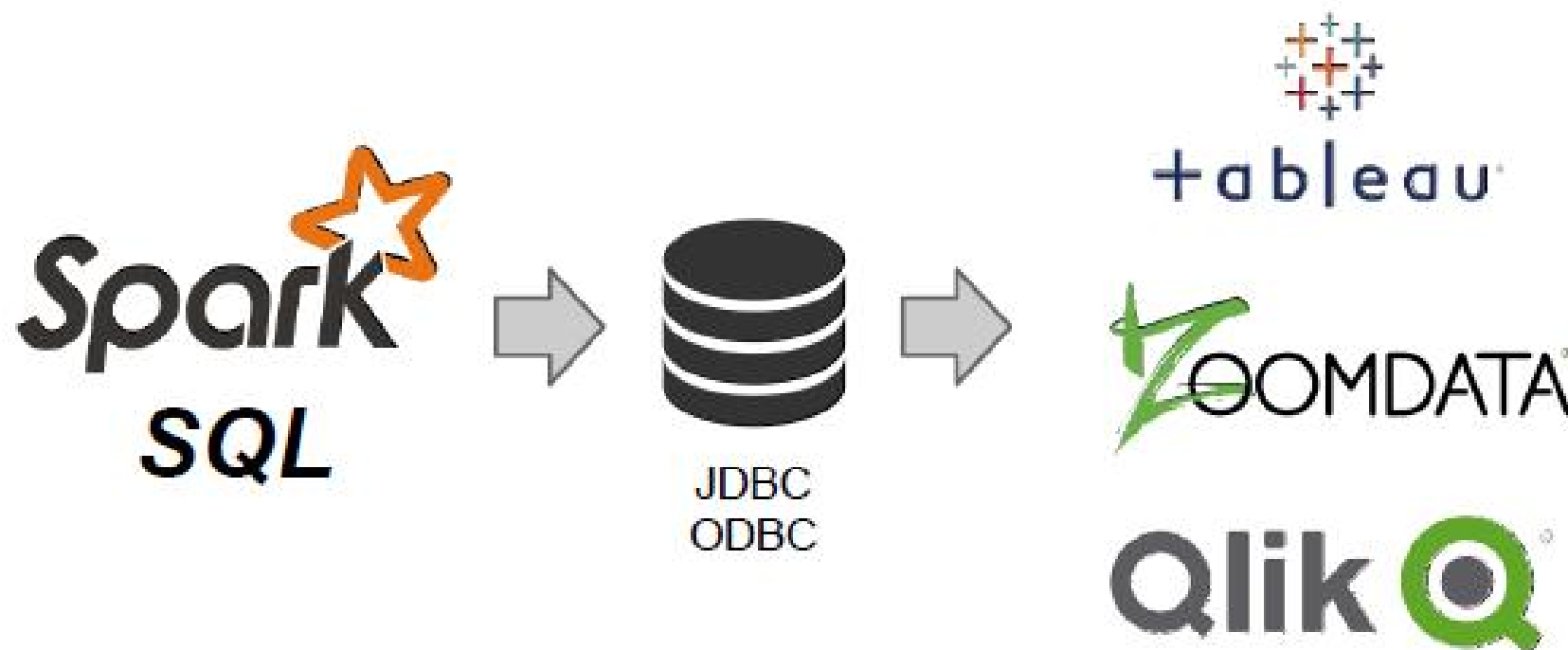


```
SELECT COUNT(*)
FROM hiveTable
WHERE hive_udf(data)
```

Spark SQL	Apache Hive
Library	Framework
Optional metastore	Mandatory metastore
Automatic schema inference	Explicit schema declaration using DDL
API - DataFrame DSL and SQL	HQL
Supports both Spark SQL and HQL	Only HQL
Hive Thrift server	Hive thrift server

Spark SQL

➤3. Connect existing BI tools to Spark through JDBC



Spark SQL

➤4. Bindings in Python, Scala, Java and R

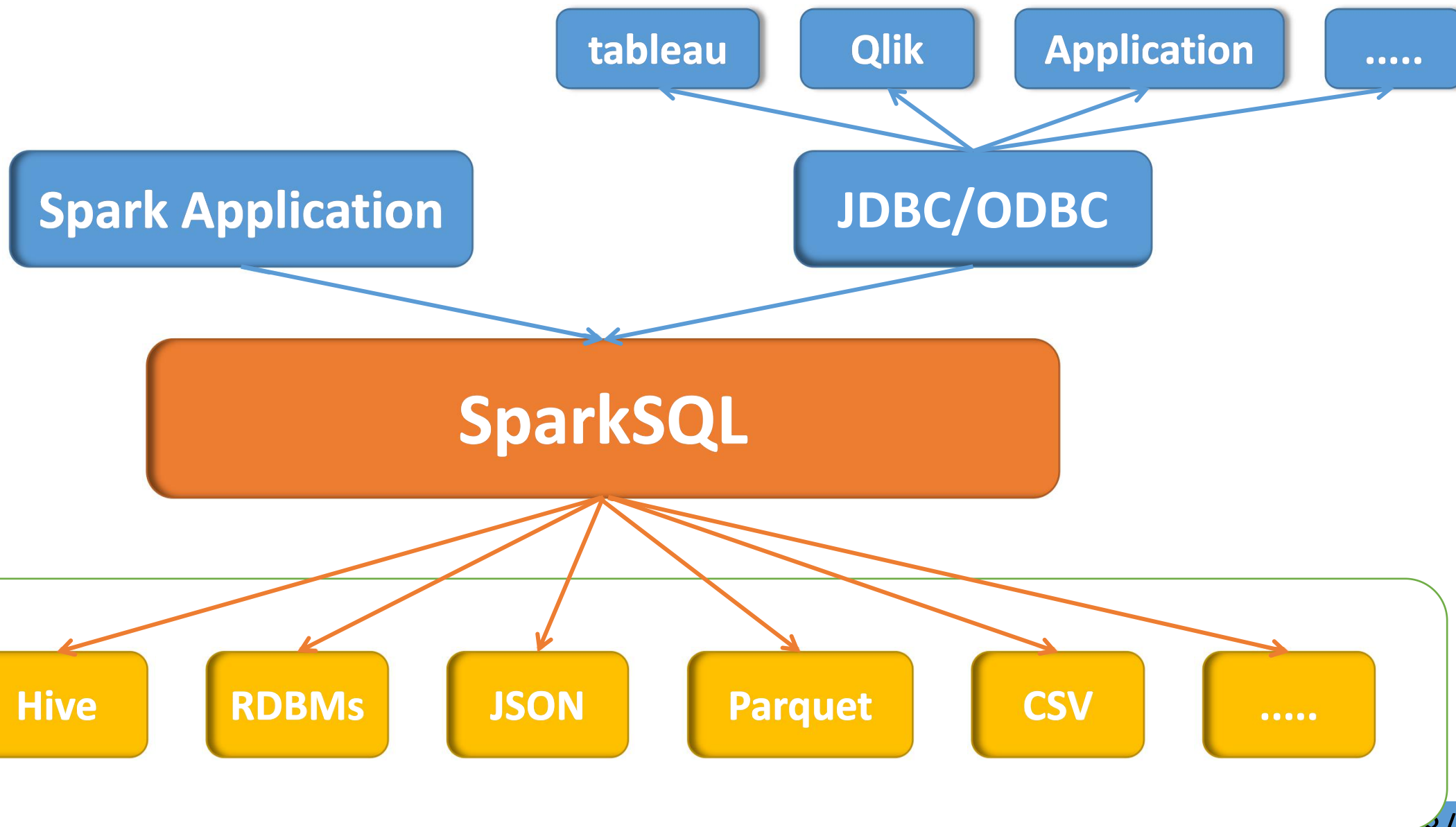


The not-so-secret truth...

Spark  *SQL*
is not about SQL.

Spark  *SQL*
is about more than SQL.

The not-so-secret truth...



Official Definition

- A Spark module for **structured data** processing(known set of fields for each record - schema) ;
- Provides DataFrames as an abstraction for distributed data processing ;
- Acts as a distributed SQL engine ;

The whole story

Creating and Running Spark Programs Faster:

- ◆ Write less code
- ◆ Read less data
- ◆ Let the optimizer do the hard work

Spark SQL编译

■ 编译Spark时支持Hive

Apache Hadoop 2.4.X with Hive 13 support

```
mvn -Pyarn -Phadoop-2.4 -Dhadoop.version=2.4.0 -Phive -Phive-thriftserver -DskipTests clean package
```

To enable Hive integration for Spark SQL along with its JDBC server and CLI, add the `-Phive` and `Phive-thriftserver` profiles to your existing build options. By default Spark will build with Hive 0.13.1 bindings. You can also build for Hive 0.12.0 using the `-Phive-0.12.0` profile.

```
# Apache Hadoop 2.4.X with Hive 13 support
```

```
mvn -Pyarn -Phadoop-2.4 -Dhadoop.version=2.4.0 -Phive -Phive-thriftserver -DskipTests clean package
```

```
# Apache Hadoop 2.4.X with Hive 12 support
```

```
mvn -Pyarn -Phadoop-2.4 -Dhadoop.version=2.4.0 -Phive -Phive-0.12.0 -Phive-thriftserver -DskipTests clean package
```

<http://spark.apache.org/docs/1.6.1/building-spark.html>

spark-sql/spark-shell

- hive-site.xml
- mysql driver lib

```
> > spark-sql
```

Cache Table

The `shark.cache table` property no longer exists, and tables whose name end with `_cached` are no longer automatically cached. Instead, we provide `CACHE TABLE` and `UNCACHE TABLE` statements to let user control table caching explicitly:

```
CACHE TABLE logs_last_month;
UNCACHE TABLE logs_last_month;
```

NOTE: `CACHE TABLE tbl` is now **eager** by default not **lazy**. Don't need to trigger cache materialization manually anymore.

Spark SQL newly introduced a statement to let user control table caching whether or not lazy since Spark 1.2.0:

```
CACHE [LAZY] TABLE [AS SELECT] ...
```

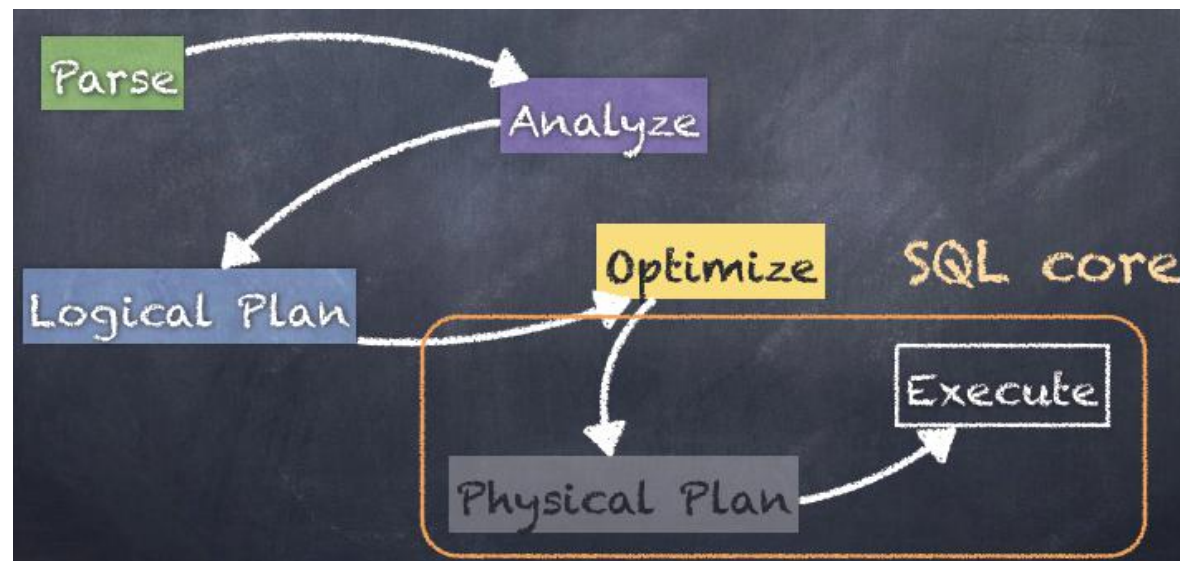
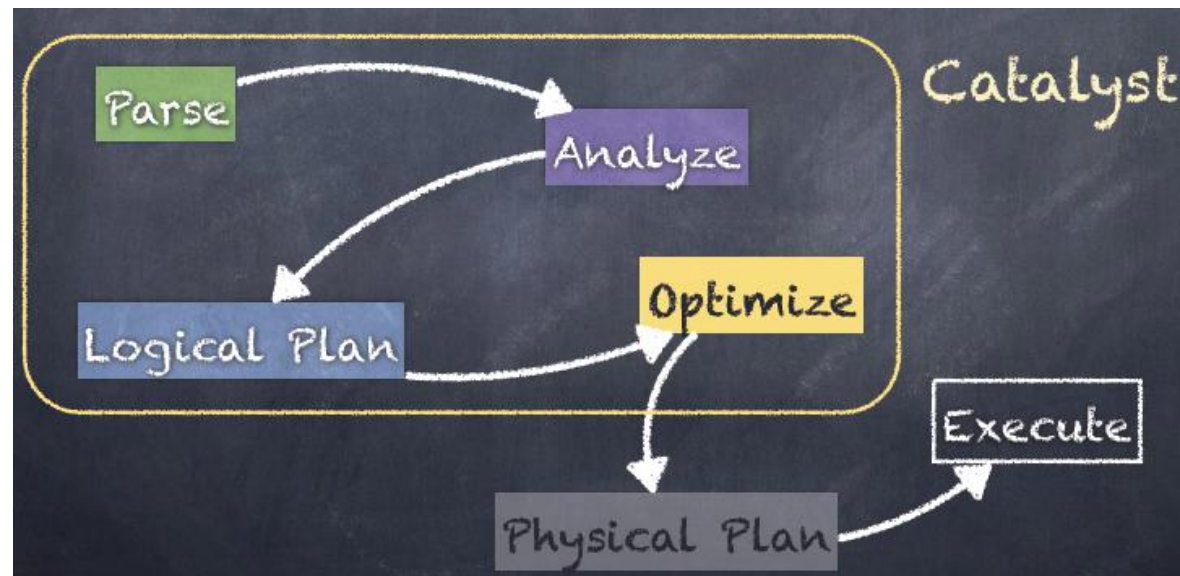
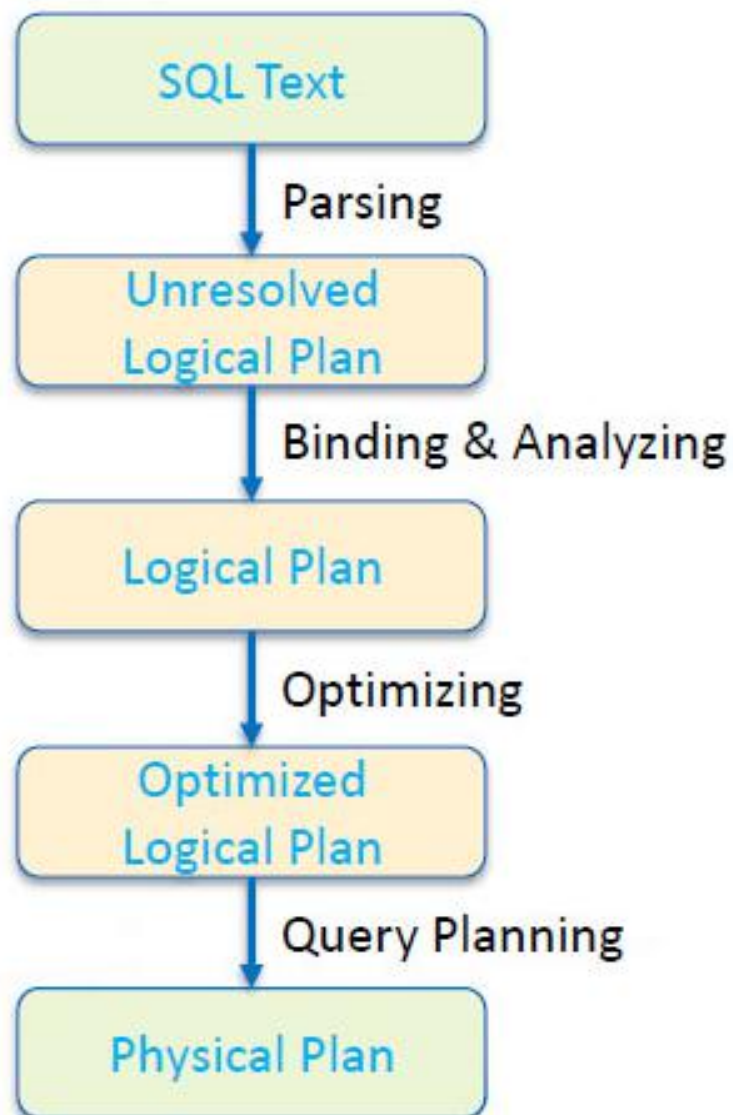
◆ `cache table emp ;`

◆ `uncache table emp ;`

Several caching related features are not supported yet:

- User defined partition level cache eviction policy
- RDD reloading
- In-memory cache write through policy

Catalyst



Examples

We execute the following commands on Spark SQL CLI:

```
CREATE TABLE T (key string, value string)
```

```
EXPLAIN EXTENDED SELECT a.key * (2 + 3),b.value FROM T a  
JOIN T b ON a.key=b.key AND a.key>3
```


Examples

== Parsed Logical Plan ==

```
Project [('a.key * (2 + 3)) AS c_0#24,'b.value]
  Join Inner, Some(((('a.key = 'b.key) && ('a.key > 3))))
    UnresolvedRelation None, T, Some(a)
    UnresolvedRelation None, T, Some(b)
```

== Analyzed Logical Plan ==

```
Project [(CAST(key#27, DoubleType) * CAST((2 + 3), DoubleType)) AS c_0#24,value#30]
  Join Inner, Some(((key#27 = key#29) && (CAST(key#27, DoubleType) > CAST(3, DoubleType))))
    MetastoreRelation default, T, Some(a)
    MetastoreRelation default, T, Some(b)
```

== Optimized Logical Plan ==

```
Project [(CAST(key#27, DoubleType) * 5.0) AS c_0#24,value#30]
  Join Inner, Some((key#27 = key#29))
    Project [key#27]
      Filter (CAST(key#27, DoubleType) > 3.0)
        MetastoreRelation default, T, Some(a)
    MetastoreRelation default, T, Some(b)
```

== Physical Plan ==

```
Project [(CAST(key#27, DoubleType) * 5.0) AS c_0#24,value#30]
  BroadcastHashJoin [key#27], [key#29], BuildLeft
    Filter (CAST(key#27, DoubleType) > 3.0)
      HiveTableScan [key#27], (MetastoreRelation default, T, Some(a)), None
    HiveTableScan [key#29,value#30], (MetastoreRelation default, T, Some(b)), None
```


SQLContext

- ◆ The **entry point** into all functionality in Spark SQL is **SQLContext** class, or one of its **descendants**.
- ◆ To create a basic SQLContext, all you need is a SparkContext:

```
val sc: SparkContext // An existing SparkContext.  
val sqlContext = new org.apache.spark.sql.SQLContext(sc)  
  
// this is used to implicitly convert an RDD to a DataFrame.  
import sqlContext.implicits._  
  
sqlContext.sql("select * from emp").show
```

HiveContext

When working with Hive one must construct a **HiveContext**, which inherits from SQLContext, and adds support for finding tables in the MetaStore and writing queries using HiveQL.

```
// sc is an existing SparkContext.
```

```
val sqlContext = new org.apache.spark.sql.hive.HiveContext(sc)
```

```
sqlContext.sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING)")
```

```
sqlContext.sql("LOAD DATA LOCAL INPATH 'examples/src/main/resources/kv1.txt' INTO TABLE src")
```

```
// Queries are expressed in HiveQL
```

```
sqlContext.sql("FROM src SELECT key, value").collect().foreach(println)
```

Thrift JDBC Server/beeline

- hive-site.xml
- mysql driver lib

```
> > start-thriftserver.sh
```

```
> > beeline -u jdbc:hive2://hostname:port/default -n hadoop
```

<http://spark.apache.org/docs/1.6.1/sql-programming-guide.html#running-the-thrift-jdbcodbc-server>

SparkSQL案例一

- 通过JDBC连接Spark thrift server服务，并测试一般的sql语句是否支持

```
<dependency>  
  <groupId>org.spark-project.hive</groupId>  
  <artifactId>hive-jdbc</artifactId>  
  <version>0.13.1</version>  
</dependency>
```

What is DataFrame ?

- A **distributed collection** of **rows organized into named columns**;(column name / column type / column value);
- It is conceptually equivalent to a **table in a relational database** or a data frame in R/Python, but with richer optimizations under the hood;
- An abstraction for **selecting, filtering, aggregation** and deal structured data (cf. R, Pandas);
- Archaic: Previously SchemaRDD (cf. Spark < 1.3);

What is DataFrame ?

- 在Spark中，**DataFrame**是一种以**RDD**为基础的分布式数据集，类似于**传统数据库中的二维表格**。
- DataFrame与RDD的主要区别在于，前者带有schema元信息，即DataFrame所表示的二维表数据集的每一列都带有**名称和类型**。
- 使得Spark SQL得以洞察更多的结构信息，从而对藏于DataFrame背后的数据源以及作用于DataFrame之上的变换进行了针对性的优化，最终达到大幅提升运行时效率的目标。
- 反观RDD，由于无从得知所存数据元素的具体内部结构，Spark Core只能在stage层面进行简单、通用的流水线优化。

DataFrame v.s. RDD[T]

Person
Person
Person

Person
Person
Person

RDD[Person]

Name	Age	Height
------	-----	--------

String	Int	Double
String	Int	Double
String	Int	Double

String	Int	Double
String	Int	Double
String	Int	Double

DataFrame


Write Less Code: Input & Output

Unified interface to **reading/writing** data in a variety of formats:

```
df = sqlContext.read \
    .format("json") \
    .option("samplingRatio", "0.1") \
    .load("/home/michael/data.json")
```

```
df.write \
    .format("parquet") \
    .mode("append") \
    .partitionBy("year") \
    .saveAsTable("fasterData")
```

load(...), save(...) or
saveAsTable(...)
functions create
new builders for
doing I/O



Write Less Code: Input & Output

ETL Using Custom Data Sources

```
sqlContext.read  
  .format("com.databricks.spark.git")  
  .option("url", "https://github.com/apache/spark.git")  
  .option("numPartitions", "100")  
  .option("branches", "master,branch-1.3,branch-1.2")  
  .load()  
  .repartition(1)  
  .write  
  .format("json")  
  .save("/home/michael/spark.json")
```

Write Less Code: Compute an Average



```
private IntWritable one =
    new IntWritable(1)
private IntWritable output =
    new IntWritable()
protected void map(
    LongWritable key,
    Text value,
    Context context) {
    String[] fields = value.split("\t")
    output.set(Integer.parseInt(fields[1]))
    context.write(one, output)
}

IntWritable one = new IntWritable(1)
DoubleWritable average = new DoubleWritable()

protected void reduce(
    IntWritable key,
    Iterable<IntWritable> values,
    Context context) {
    int sum = 0
    int count = 0
    for(IntWritable value : values) {
        sum += value.get()
        count++
    }
    average.set(sum / (double) count)
    context.write(key, average)
}
```



```
data = sc.textFile(...).split("\t")
data.map(lambda x: (x[0], [x[1], 1])) \
    .reduceByKey(lambda x, y: [x[0] + y[0], x[1] + y[1]]) \
    .map(lambda x: [x[0], x[1][0] / x[1][1]]) \
    .collect()
```

Write Less Code: Compute an Average

Using RDDs

```
data = sc.textFile(...).split("\t")
data.map(lambda x: (x[0], [int(x[1]), 1])) \
    .reduceByKey(lambda x, y: [x[0] + y[0], x[1] + y[1]]) \
    .map(lambda x: [x[0], x[1][0] / x[1][1]]) \
    .collect()
```

Using SQL

```
SELECT name, avg(age)
FROM people
GROUP BY name
```

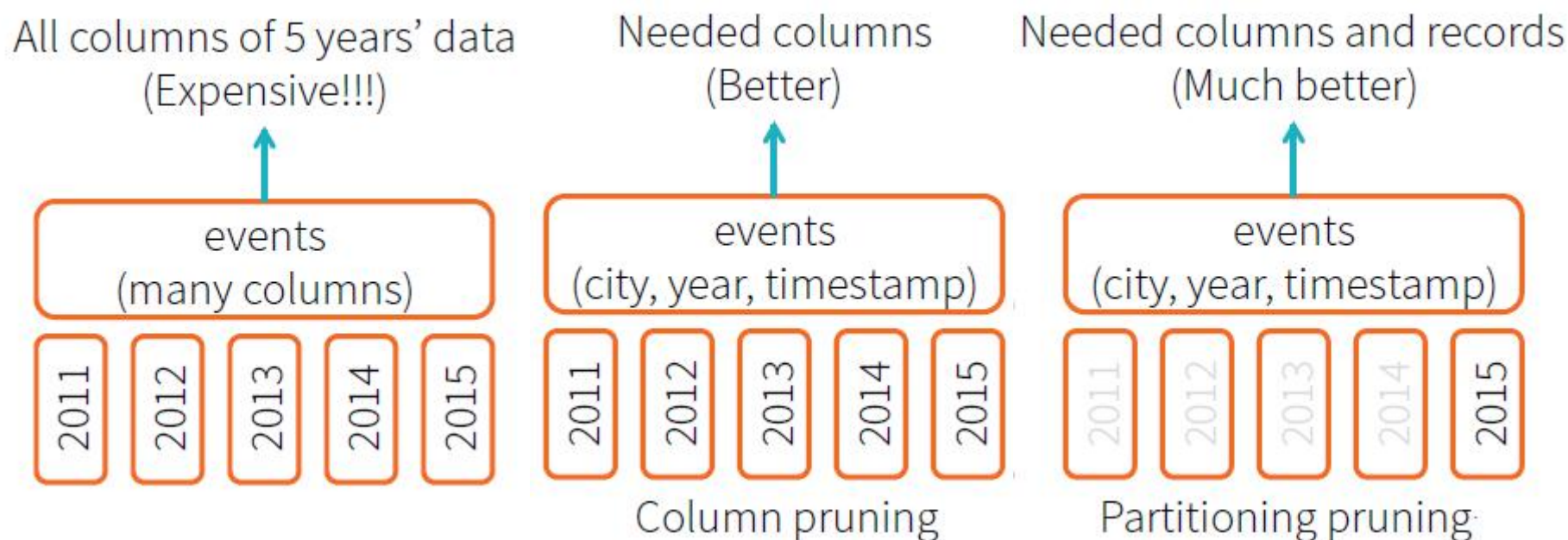
Using DataFrames

```
sqlCtx.table("people") \
    .groupBy("name") \
    .agg("name", avg("age")) \
    .collect()
```

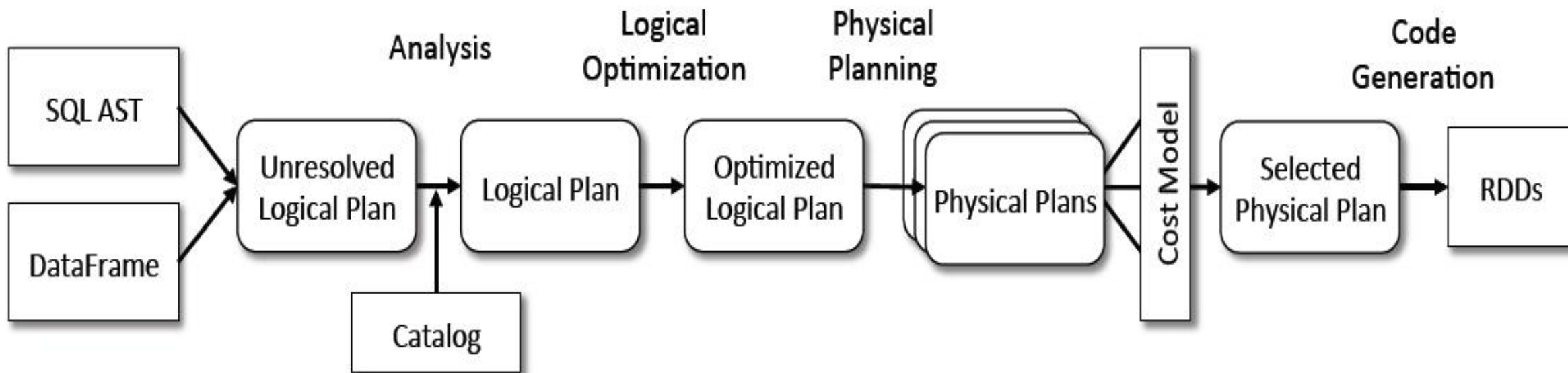

Let the optimizer do the hard work

```

events = sqlCtx.load("/data/events", "parquet")
training_data =
    events
        .where("city = 'New York' and year = 2015")
        .select("timestamp").collect()
    
```



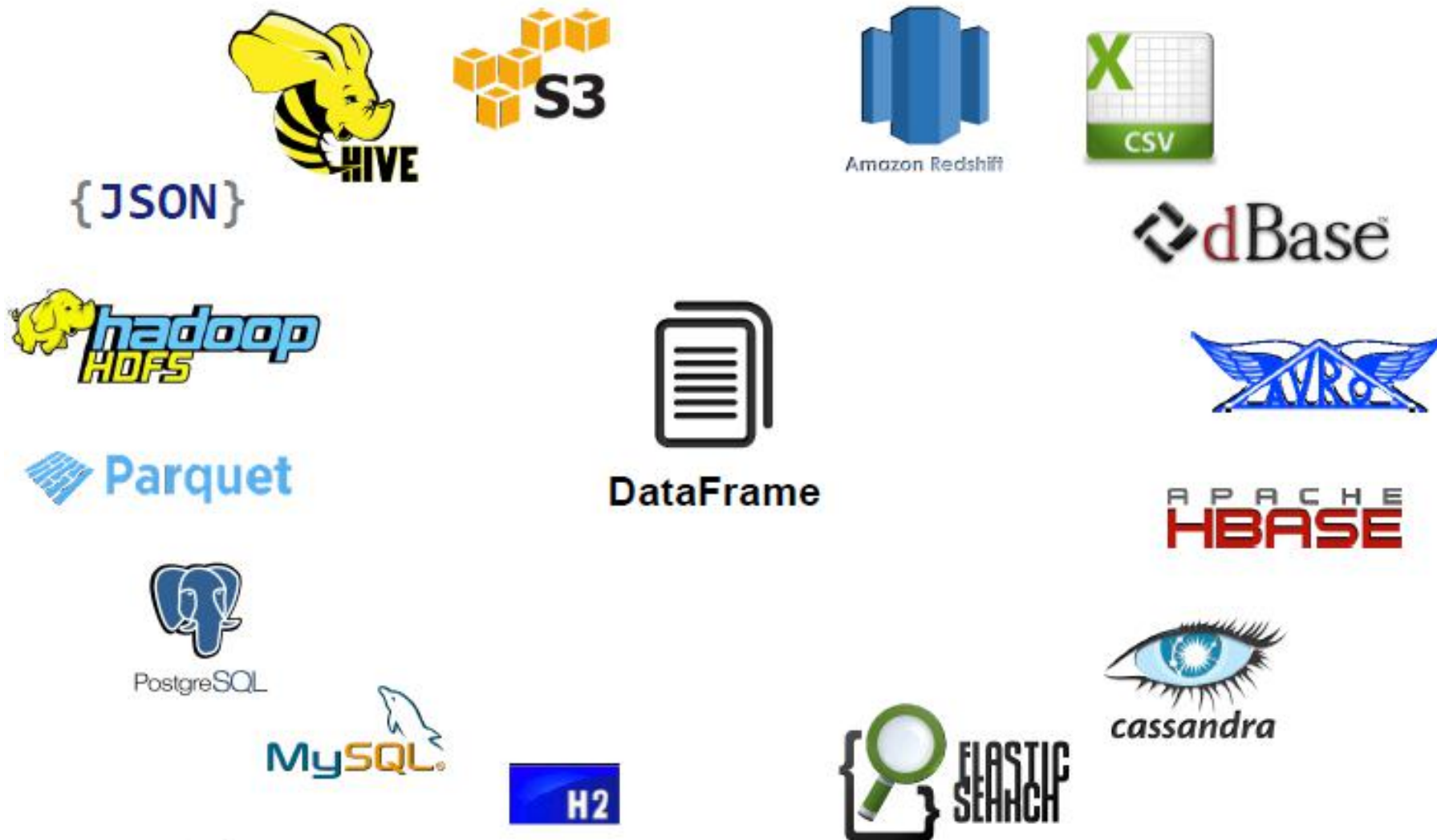
- Represented internally as a “logical plan” ;
- Execution is lazy, allowing it to be optimized by Catalyst;



DataFrames and SQL share the same optimization/execution pipeline

<http://databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html>

External Data Source API



Data Source Libraries

Users can use libraries based on Data Source API to **read/write** DataFrames **from/to** a variety of formats/systems.

Built-In Support



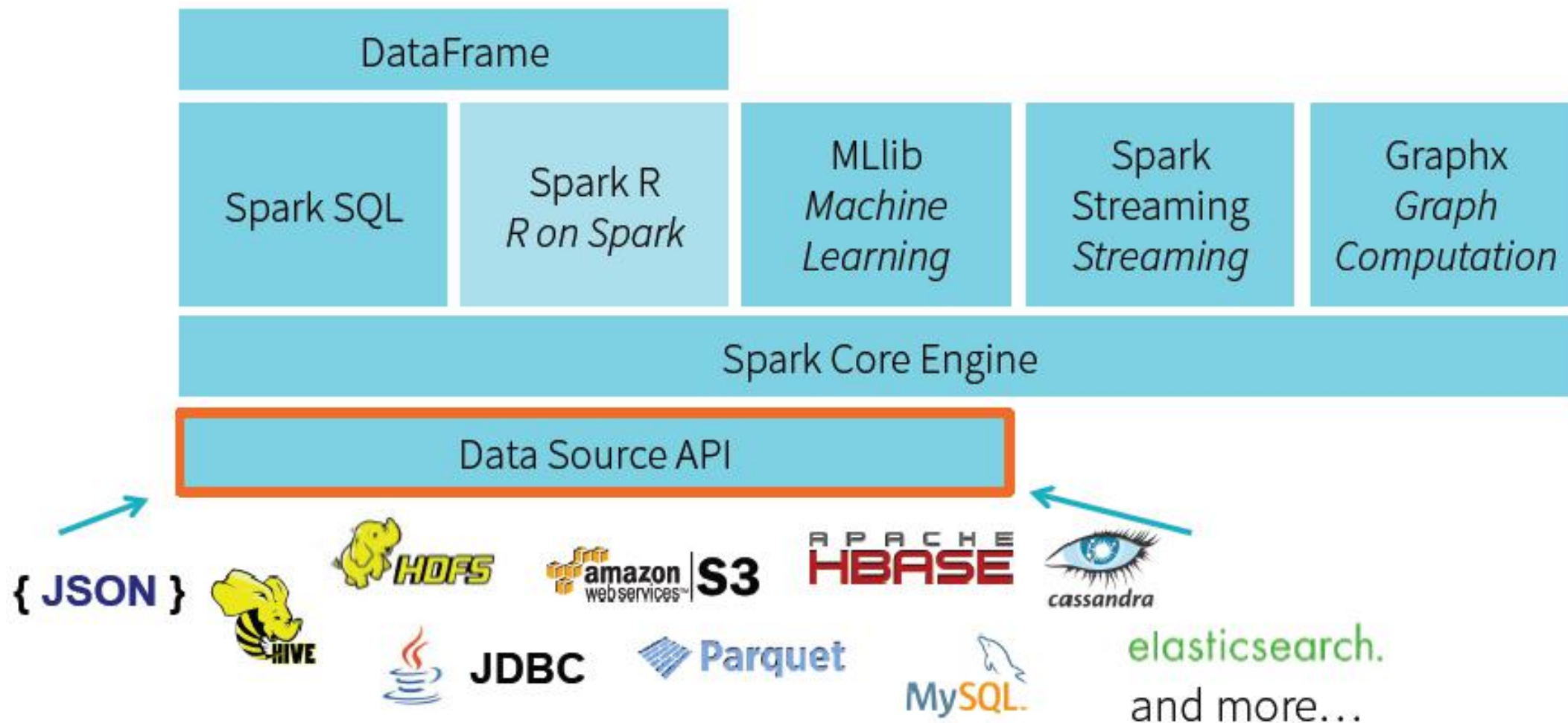
External Libraries



<https://github.com/databricks/>
<https://spark-packages.org/>

Data Source API

Alpha/Pre-alpha

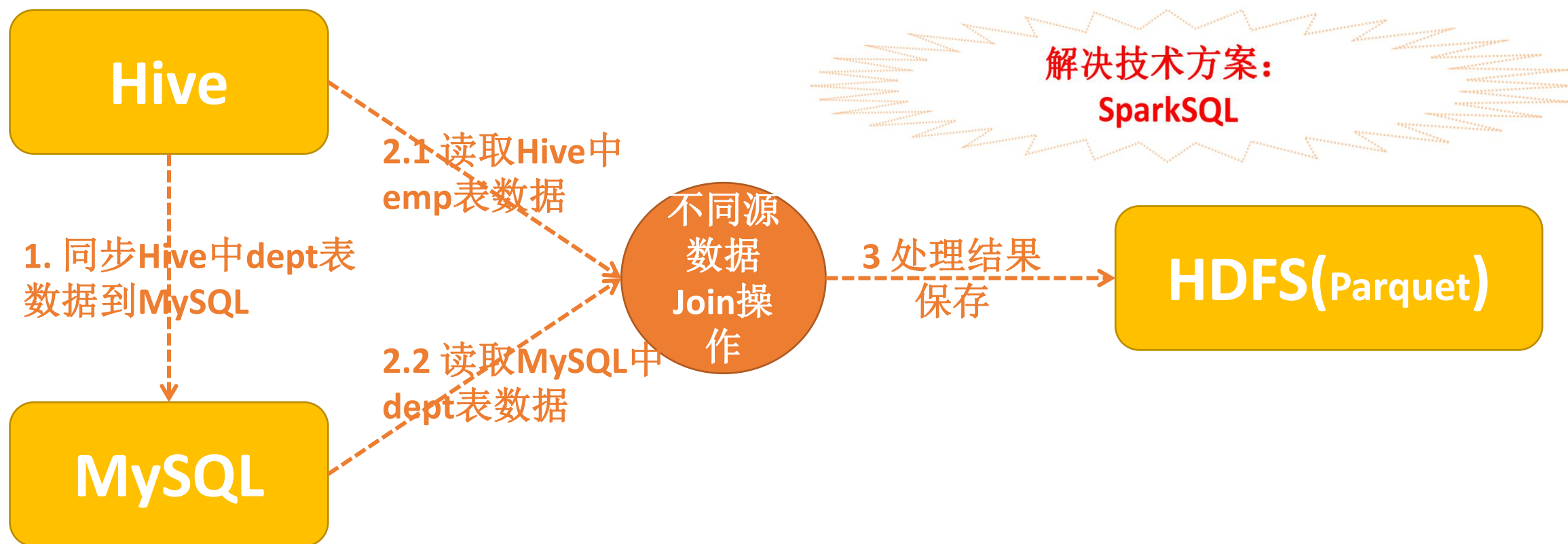


External Data Source API

Every Spark application starts
with loading data and ends with
saving data

Spark SQL案例（二）

- 案例：将Hive表数据输出到MySQL表中，将Hive表和MySQL表进行数据Join操作，并将最终结果保存为Parquet格式的数据，存储在HDFS中



RDD和DataFrame转换

- DataFrame ==> RDD: 直接调用DataFrame类提供的rdd方法即可将DataFrame转换为RDD数据类型
- RDD ==> DataFrame: 将RDD转换为DataFrame主要由两种方式，如下：
 - ◆ **Inferring the Schema Using Reflection**：利用case class类型的RDD进行数据反射创建DataFrame，要求RDD中的数据类型必须是case class类型。
 - ◆ **Programmatically Specifying the Schema**：主动给定DataFrame的数据类型来创建DataFrame，要求RDD的数据类型必须是Row，另外必须给定RDD中数据对应的数据类型，调用SQLContext的createDataFrame函数来创建DataFrame。

SparkSQL函数

- Hive支持的内置函数，SparkSQL基本上都支持，需要稍微注意一下的是：有一些Hive的函数的使用需要使用HiveContext对象，不能使用SQLContext对象来操作。
- SparkSQL支持两种自定义函数，分别是：**UDF**和**UDAF**，两种函数都是通过SQLContext的udf属性进行函数的注册使用的；SparkSQL不支持UDTF函数的自定义使用
 - ◆ UDF：一条数据输入，一条数据输出，一对一的函数，即普通函数
 - ◆ UDAF：多条数据输入，一条数据输出，多对一的函数，即聚合函数

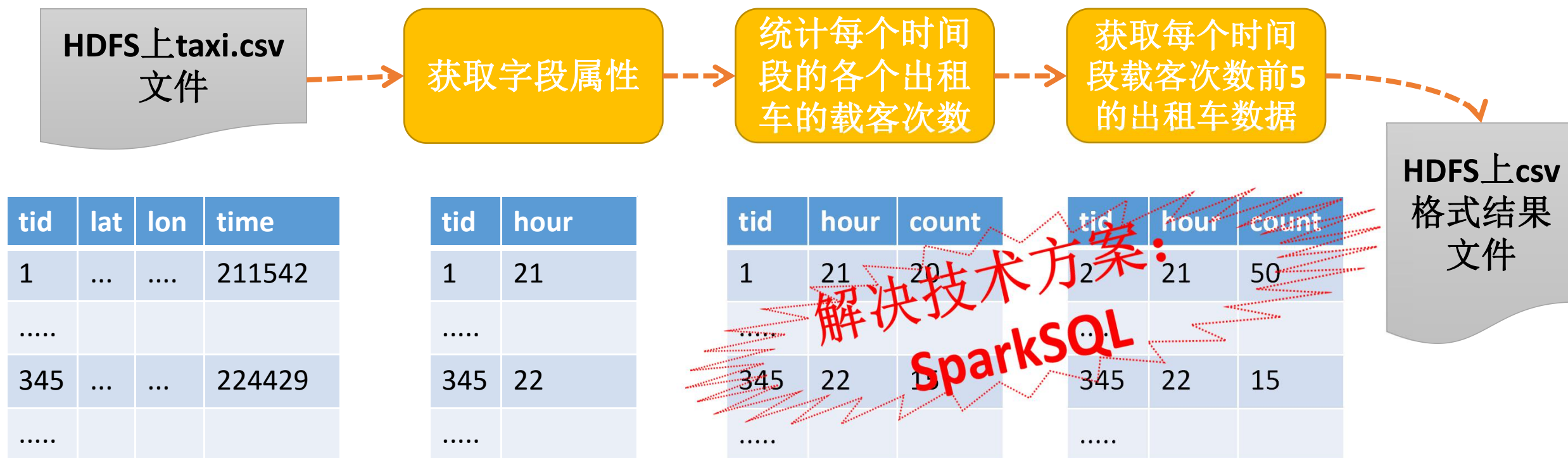
SparkSQL自定义函数案例

■ 案例：自定义SparkSQL函数

- ◆ UDF：Double类型的数据格式化，要求默认参数保留两位小数，可以给定具体保留小数的数量
- ◆ UDAF：自定义实现一个求平均值的函数

SparkSQL案例(三)

- 案例：通过SparkSQL读写CSV格式的数据，并实现给定需求
 - ◆ 需求：根据taxi.csv文件中的数据，求各个小时段Top5载客次数的出租车载客次数数据(假设一条数据就是一次载客记录)，并将结果保存为csv文件



SparkSQL DSL语法

- SparkSQL除了支持直接的HQL语句的查询外，还支持通过DSL语句/API进行数据的操作，主要DataFrame API列表如下：
 - ◆ select：类似于HQL语句中的select，获取需要的字段信息
 - ◆ where/filter：类似HQL语句中的where语句，根据给定条件过滤数据
 - ◆ sort/orderBy：全局数据排序功能，类似Hive中的order by语句，按照给定字段进行全部数据的排序
 - ◆ sortWithinPartitions：类似Hive的sort by语句，按照分区进行数据排序
 - ◆ groupBy：数据聚合操作
 - ◆ limit：获取前N条数据记录

Dateset

- Datasets are similar to RDDs, however, **instead of using Java Serialization or Kryo** they use **a specialized Encoder** to **serialize the objects for processing or transmitting over the network.**
- While both encoders and standard serialization are responsible for turning an object into bytes, encoders are code generated dynamically and use a format that **allows Spark to perform many operations like filtering, sorting and hashing without deserializing the bytes back into an object.**

Dataset

A `[[Dataset]]` is a strongly typed collection of objects that can be transformed in parallel using functional or relational operations.

A `[[Dataset]]` differs from an `[[RDD]]` in the following ways:

- Internally, a `[[Dataset]]` is represented by a Catalyst logical plan and the data is stored in the encoded form. This representation allows for additional logical operations and enables many operations (sorting, shuffling, etc.) to be performed without deserializing to an object.
- The creation of a `[[Dataset]]` requires the presence of an explicit `[[Encoder]]` that can be used to serialize the object into a binary format. Encoders are also capable of mapping the schema of a given object to the Spark SQL type system. In contrast, RDDs rely on runtime reflection based serialization. Operations that change the type of object stored in the dataset also need an encoder for the new type.

A `[[Dataset]]` can be thought of as a specialized `DataFrame`, where the elements map to a specific JVM object type, instead of to a generic `[[Row]]` container. A `DataFrame` can be transformed into specific `Dataset` by calling ``df.as[ElementType]``. Similarly you can transform a strongly-typed `[[Dataset]]` to a generic `DataFrame` by calling ``ds.toDF()``.

COMPATIBILITY NOTE: Long term we plan to make `[[DataFrame]]` extend ``Dataset[Row]``. However, making this change to the class hierarchy would break the function signatures for the existing functional operations (map, flatMap, etc). As such, this class should be considered a preview of the final API. Changes will be made to the interface after Spark 1.6.

Dataset

```
// Encoders for most common types are automatically provided by  
importing sqlContext.implicits._
```

```
val ds = Seq(1, 2, 3).toDS()
```

```
ds.map(_ + 1).collect() // Returns: Array(2, 3, 4)
```

```
// Encoders are also created for case classes.
```

```
case class Person(name: String, age: Long)
```

```
val ds = Seq(Person("Andy", 32)).toDS()
```

```
// DataFrames can be converted to a Dataset by providing a clas  
s. Mapping will be done by name.
```

```
val path = "examples/src/main/resources/people.json"
```

```
val people = sqlContext.read.json(path).as[Person]
```

Spark SQL案例(四)

- SparkSQL实现日志分析案例
 - ◆ 将用SparkCore实现的案例使用SparkSQL实现
 - ◆ 读取Hive中对应的表，然后通过HQL语句实现日志分析的四大案例
 - ◆ 开发工具：IDEA





THANK YOU

上海育创网络科技有限公司