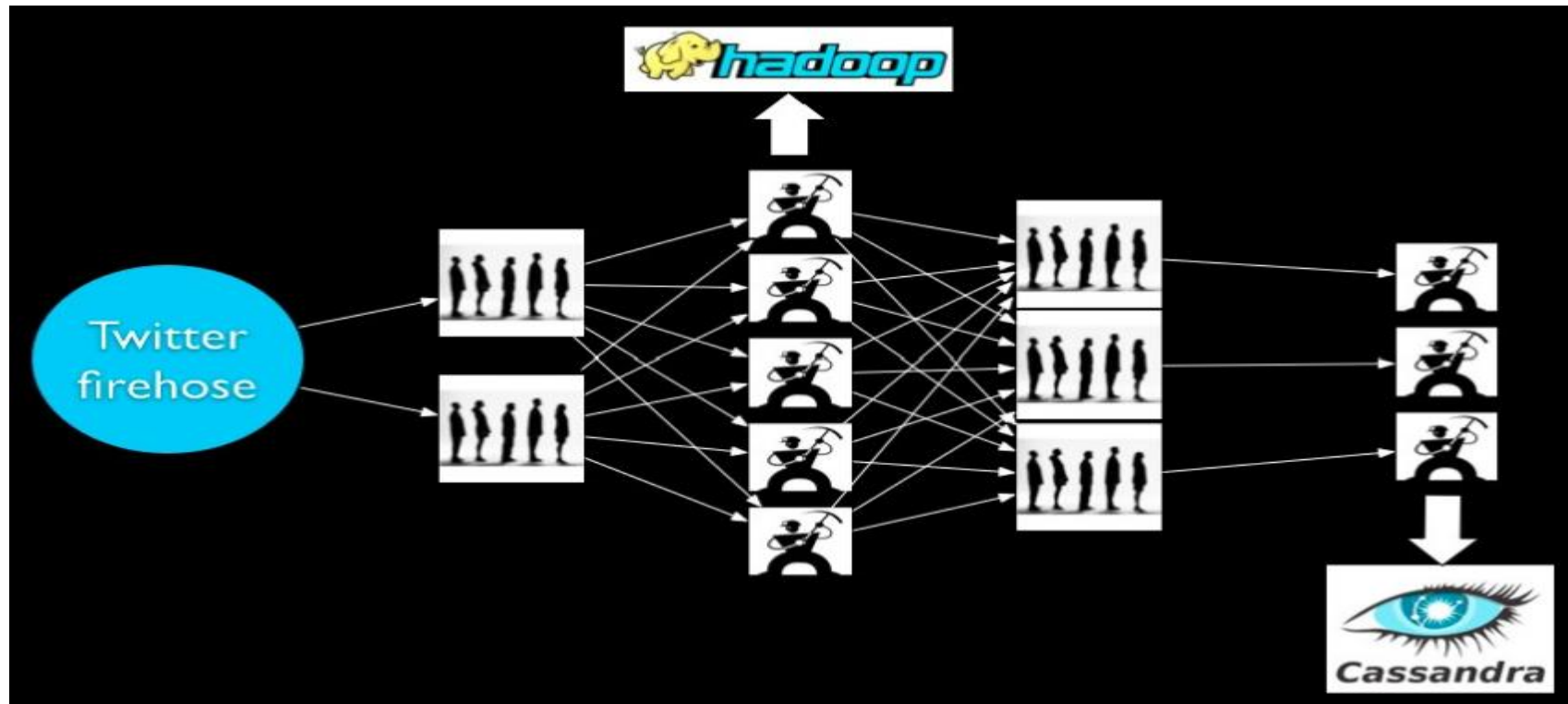


数据分析与挖掘

Storm篇



在Storm之前



MR问题

- 规模繁重
- 容错性差-Hadoop是有状态的
- 编码乏味
- 启动时间长。
- 多采用pull模型，需要手动维护一个由消息队列（Queues）和消息处理者（Workers）所组成的实时处理网络，消息处理者从消息队列取出一个消息进行处理，更新数据库，发送消息给其它队列进行进一步处理，这种计算方式的局限性太大、复杂、不健壮且扩展性差。
- 没有JVM缓存池
- 调度开销大
- 中间数据写磁盘
- 批处理
 - 长等待
 - 非实时

Storm出现

- 可扩展性和鲁棒性
- 没有持久层
- 保证无数据丢失：ack消息追踪记录
- 容错：模块无状态，随时可重启
- 与编程语言无关
- 案例：流处理，分布式RPC，持续计算

Storm集群

Storm是一个**分布式实时流式**计算平台
分布式

水平扩展：通过加机器、提高并发数就提高处理能力

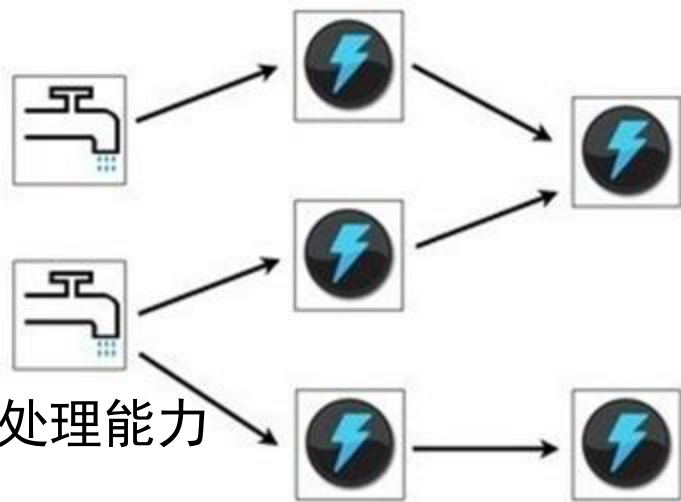
自动容错：自动处理进程、机器、网络异常

实时：数据不写磁盘，延迟低（毫秒级）

流式：不断有数据流入、处理、流出，水管不停的产生数据，流向中间螺栓(处理逻辑)

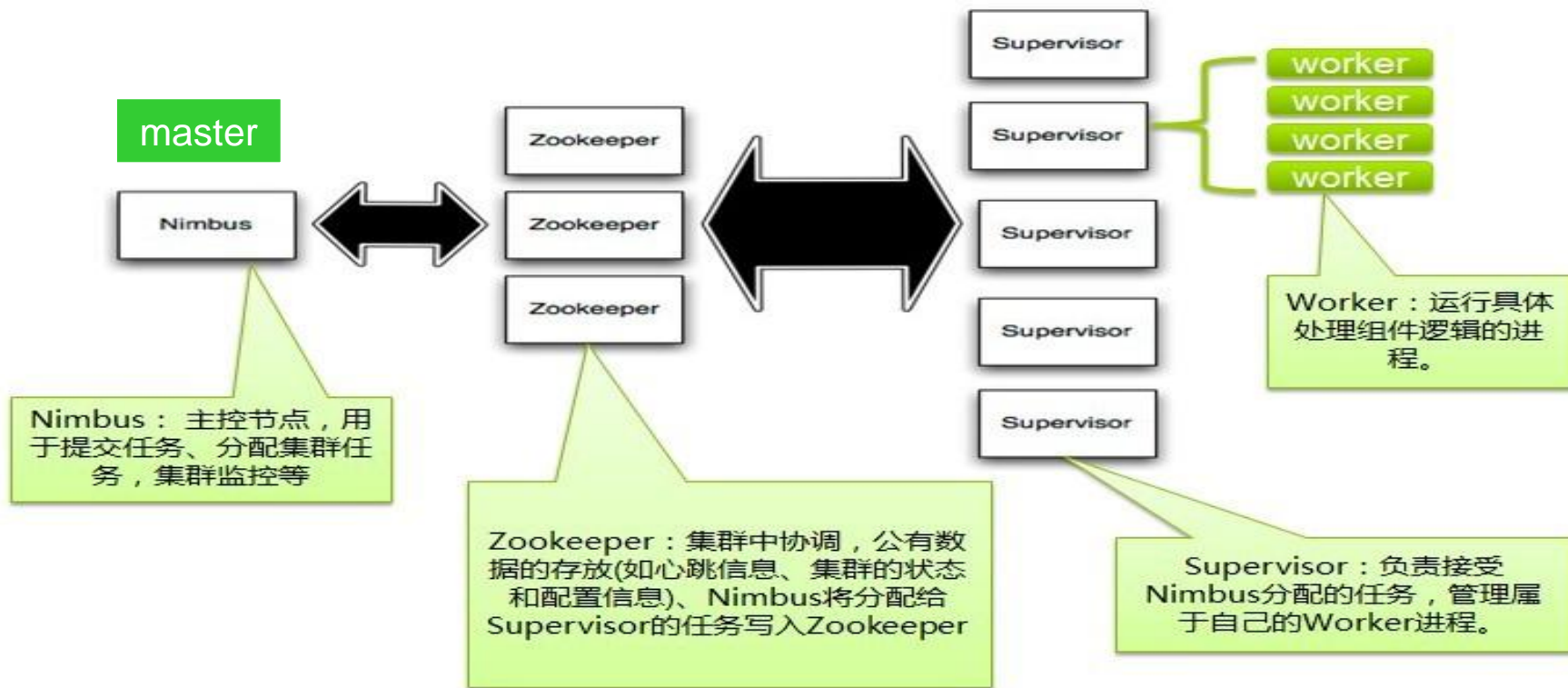
持续计算：storm拓扑不终止，除非被杀死，它一直运行

开源：twitter开源，社区很活跃



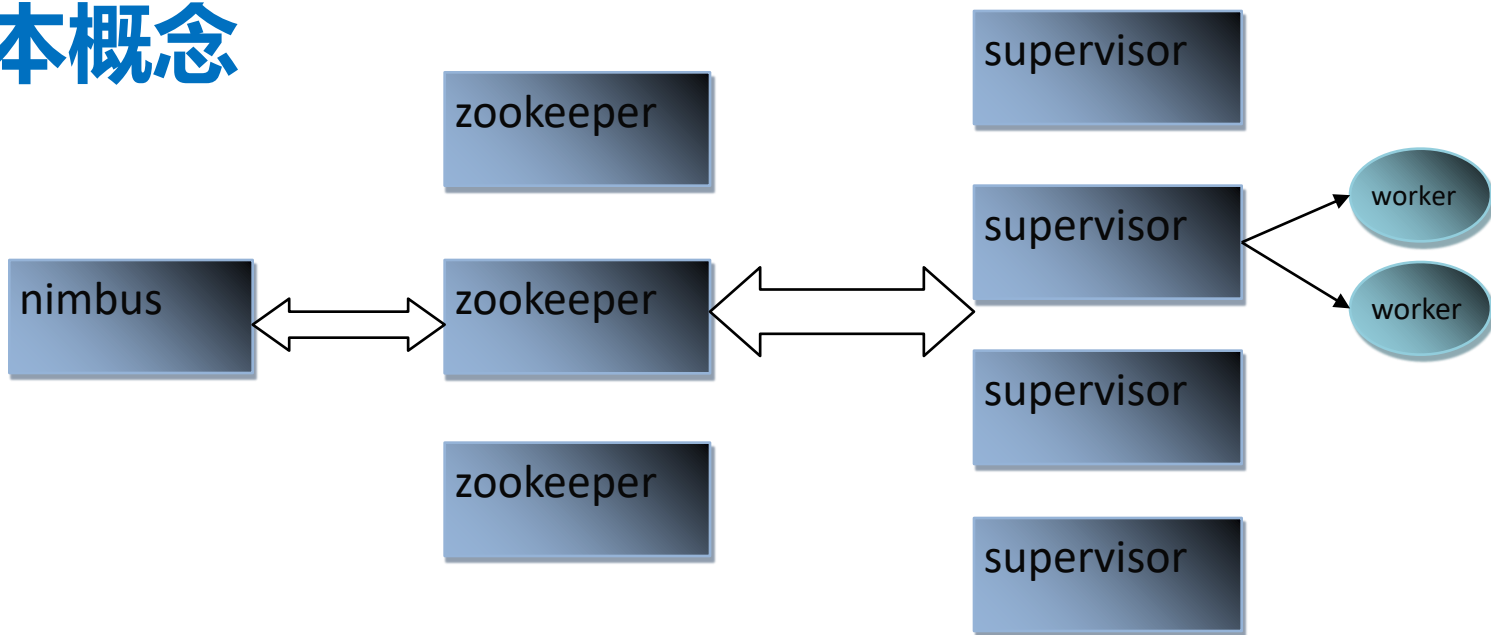
Storm集群

- **Nimbus**
 - Like JobTracker in hadoop
- **Supervisor**
 - Manage workers
- **Zookeeper**



Storm基本概念

- Nimbus
- Zookeeper
- Supervisor
- Worker
- Task
- Tuple
- Streams
- Spouts
- Bolts
- Topologies
- Groupings



nimbus: 集群master, 负责管理supervisor、调度topology

zookeeper: 负责存储以上模块的状态, 做到高可用

supervisor: 负责运行topology的worker

worker: 负责实际的计算和网络通信

Storm基本概念

	Hadoop	Storm
系统角色	RM	Nimbus
	AM	Supervisor
	NM	Worker
应用名称	Job	Topology
组件接口	Mapper/Reducer	Spout/Bolt



集群master，负责管理supervisor、调度topology

Zookeeper

分布式系统，用于存储元数据。是Storm重点依赖的外部资源。Nimbus和Supervisor是fail-fast机制和无状态的，所有状态都保存在Zookeeper。Nimbus和Supervisor甚至实际运行的Worker都是把心跳保存在Zookeeper上的。

Nimbus根据Zookeeper上的心跳和任务运行状况，进行调度和任务分配的。

Nimbus和Supervisor之间的所有沟通都是通过Zookeeper进行

在具有 $2k + 1$ 个zookeeper节点的集群上，当 k 个节点失败时，系统可以恢复。

Zookeeper

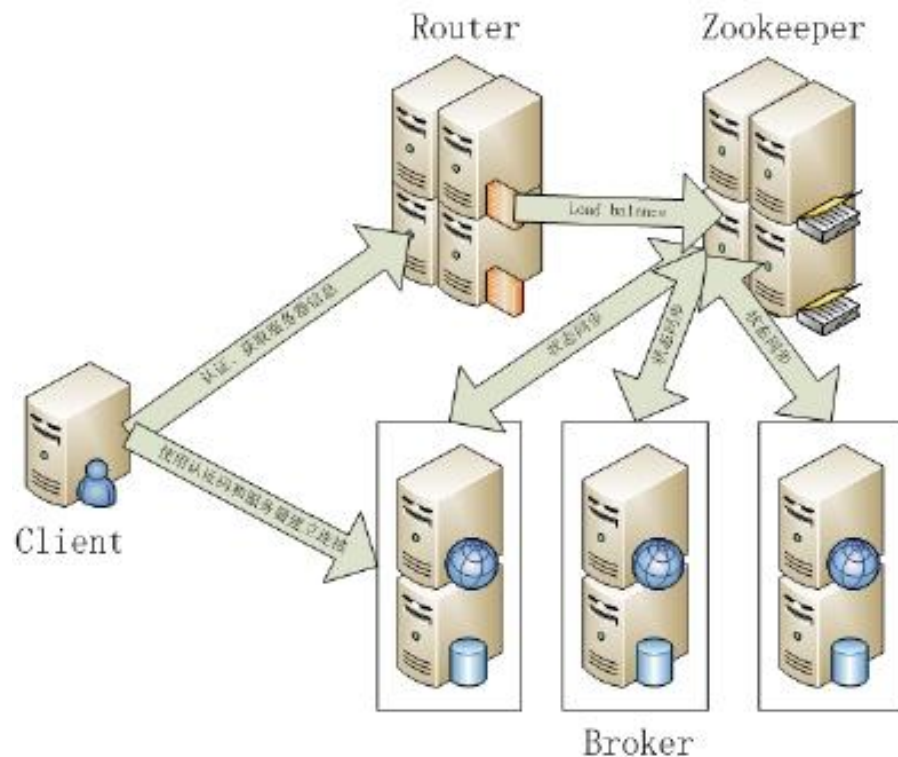
Hadoop的正式子项目，针对大型分布式系统的可靠协调系统，提供的功能包括：配置维护、名字服务、分布式同步、组服务等。

ZooKeeper的目标就是封装好复杂易出错的关键服务，将简单易用的接口和性能高效、功能稳定的系统提供给用户。

用于协调分布式系统上的各种服务。例如确认消息是否准确到达，防止单点失效，处理负载均衡等

应用场景：Hbase（用zookeeper实现数据库节点之间协调的问题），实现Namenode自动切换。

工作原理：领导者，跟随者以及选举过程



Supervisor

负责接受nimbus分配的任务，启动和停止属于自己管理的worker进程。

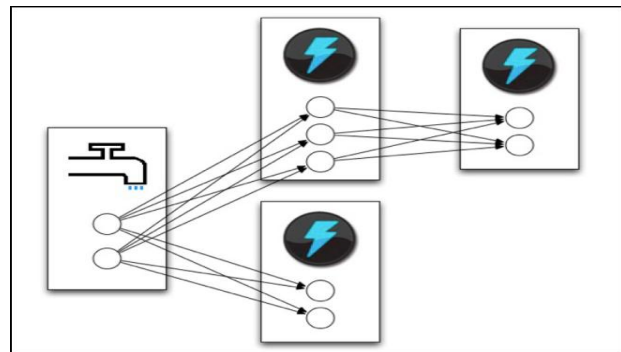
监听分配给它那台机器的工作，根据需要启动/关闭工作进程Worker。

Worker

运行具体处理组件逻辑的进程

Task

- 每个Spout和bolt都作为很多task在集群中运行
- 每个task对应一个线程，worker中每一个spout/bolt的线程称为一个task。
- 同一个spout/bolt的task可能会共享一个物理线程，该线程称为executor。
- Stream groupings定义如何把tuples从一组task发向另一组task



Tuples

表示流中一个消息传递的基本处理单元。

如一条cookie日志，包括多个field，每个field表示一个属性。

由于各个组件间传递的tuple的字段名称已经事先定义好，tuple中只要按序填入各个value

- (“user”, “link”, “event”, “10/3/12 17:50”)

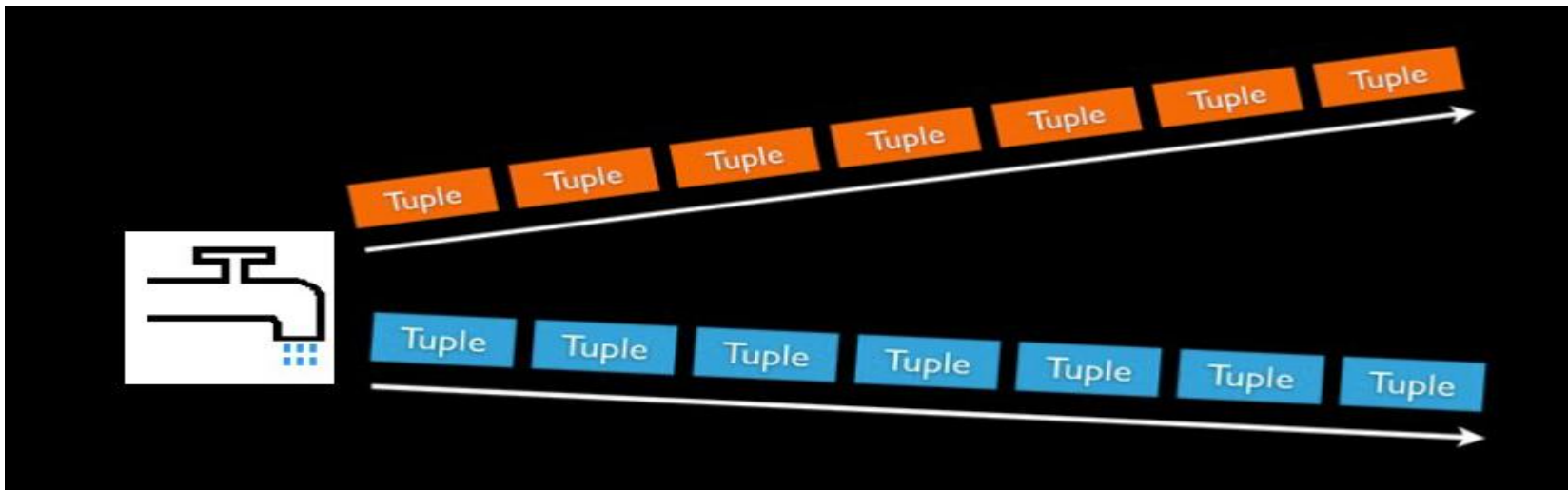


Streams

- 源源不断传递的tuple组成了stream。
- Stream是Storm中的一个核心概念，Storm将输入的数据看成流，以tuple为单位组成的一条有向无界的数据流。



Spout



Spout是一个stream的源头。

spout会从外部数据源读取数据并发送tuple到stream。

Spout会从外部数据源中读取数据，然后转化为topology内部的源数据

Spout是一个主动的角色，Storm框架会不停的调用nextTuple()函数，用户只要在其中生成源数据即可。例如：logs, API calls, event data, queues, ...

Bolts



处理输入的流并产生新的输出流。可以用来做简单的stream转换，复杂的流处理/转换一般会分解为多步完成，所以会使用多个bolt级联起来，每个bolt完成一些较简单的功能。

Bolts

一个bolt可以产生多个输出流。在一个topology中接受数据然后执行处理的组件，处理输入的流并产生新的输出流。

Bolt可以执行过滤、函数操作、合并、写数据库等操作。

Bolt是一个被动的角色，有一个叫execute(Tuple input)的函数，在接受到消息后调用此函数，用户再其中执行自己想要的操作。

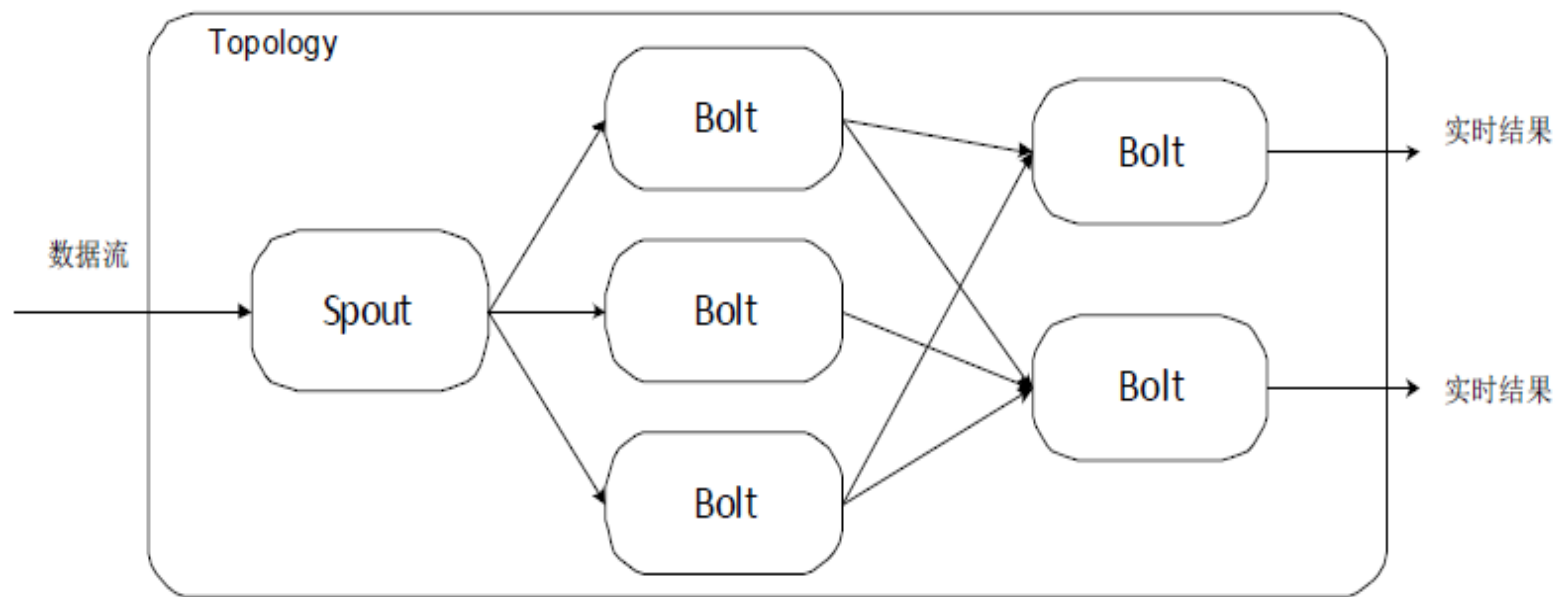
Bolt可以：

- Filtering
- Functions
- Aggregations
- Joins
- talking to databases

Topology

- 由spout和bolt构成的网状图
- 实时处理程序在逻辑上构成一个storm的拓扑
- Storm 拓扑与传统任务的区别： storm拓扑不终止的，除非被杀死，它一直运行
- storm中运行的一个实时应用程序，因为各个组件间的消息形成逻辑上的一个拓扑结构而得其名， topology处理的最小的消息单位是一个Tuple，由Spout和Bolt构成。

Topology



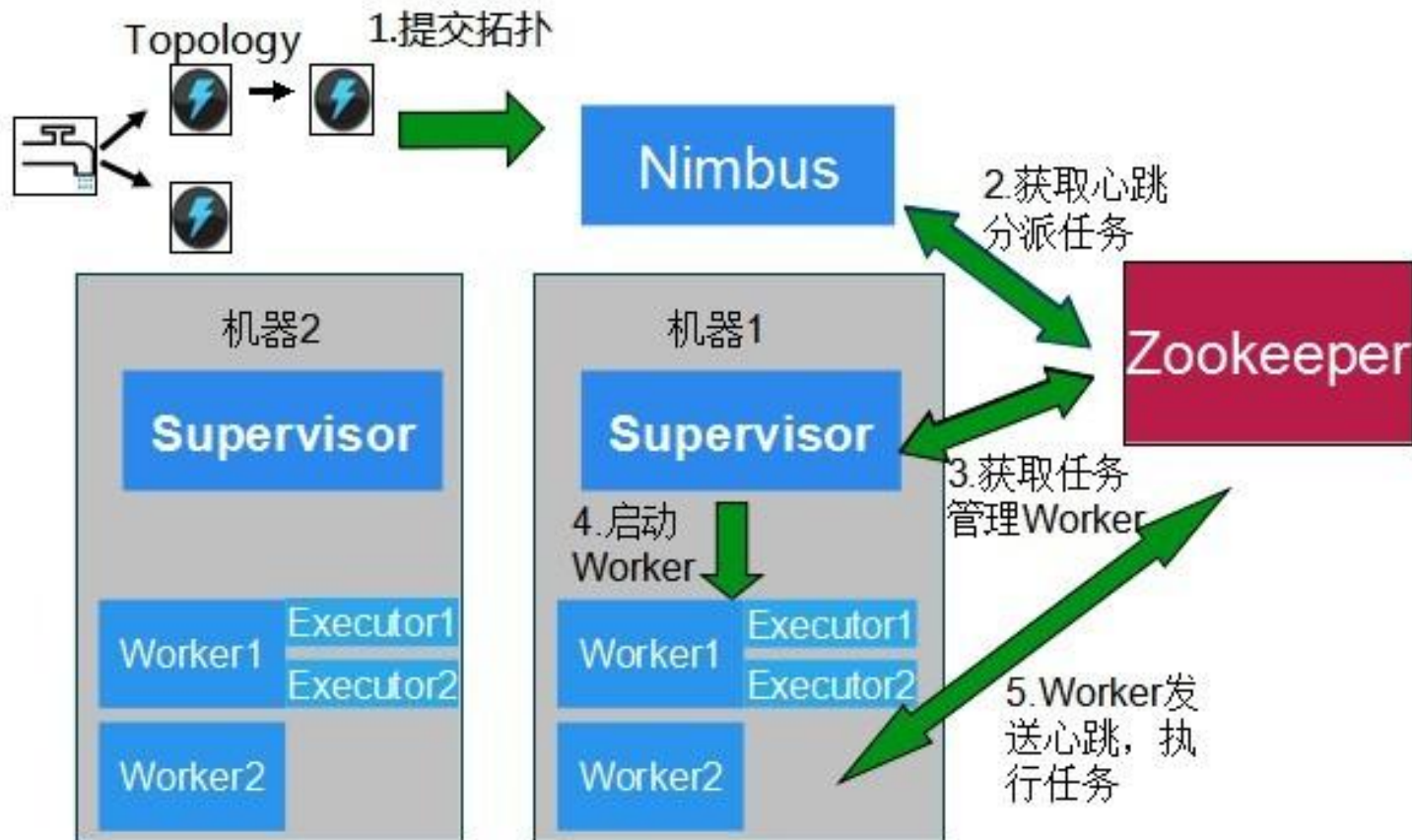
transactional topology

- 为了满足对消息处理有着极其严格的场景，如实时计算某个用户的成交笔数，要求结果完全精确。
- 完全基于底层的spout/bolt/acker原语实现的，通过一层巧妙的封装得出一个优雅的实现。
- 将消息分为一个个的批，同一批内的消息以及批与批之间的消息可以并行处理，另一方面，用户可以设置某些bolt为committer，storm可以保证committer的finishBatch()操作是按严格不降序的顺序执行的。用户可以利用这个特性通过简单的编程技巧实现消息处理的精确。

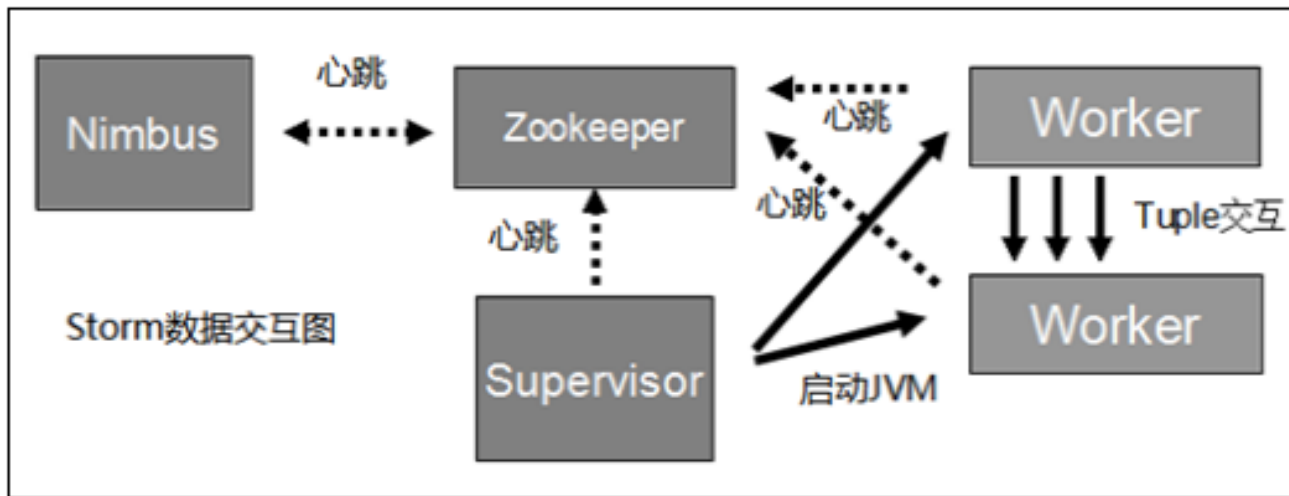
Groupings

- 消息的分组方式。控制着在event在Topology中如何流动。当一个tuple被发送时，如何确定将它发送到那个（些）task
- shuffle: 随机选择一个task发送
- fields hash: 根据tuple的一部分做一致性hash，相同的tuple被发送到相同的task
- all: 发送到所有的task.
- global: 由系统自行选择，一般是选择task id最低的tasks发送.
- none: 不关心tuple发送到哪个task，等价于shuffle grouping
- direct: 直接将tuple发送到指定的task
- localOrShuffle等

Storm一个计算任务的启动过程



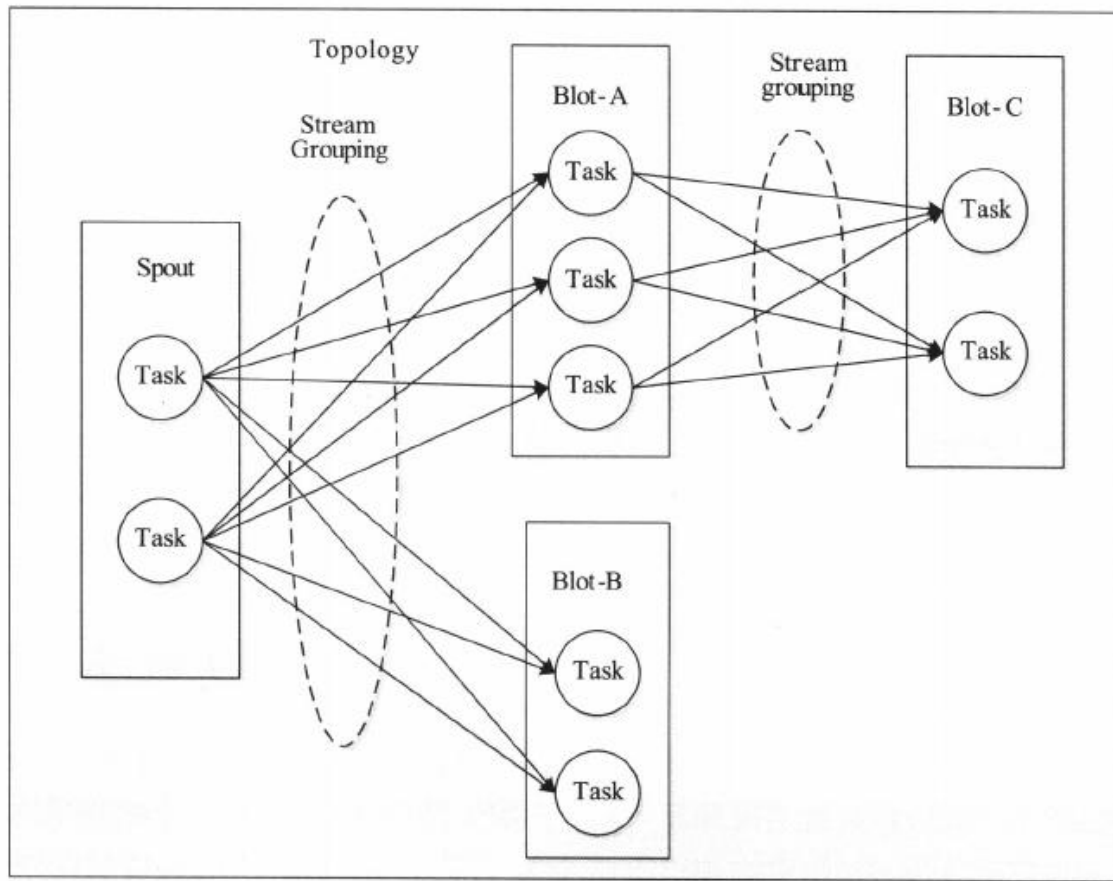
Storm数据交互



Nimbus和Supervisor之间没有直接交互。状态都是保存在Zookeeper上。Worker之间通过ZeroMQ传送数据。

Storm编程模型

- Tuple: 数据表示模型, 数据库中的一行记录, 可以为integer、long, 也可以自定义的序列化
- Stream: 消息流。每个Tuple认为是一个消息, 消息流就是Tuple队列。
- Topology: 应用程序处理逻辑, 不会终止的MR作业。
- Spout: 消息源
- Bolt: 消息处理逻辑。多个Bolt之间有依赖关系, DAG组织。
- Task: Spout和Bolt可以被并行化拆分为多个处理单元, 每个单元为一个Task Stream
- Grouping: 消息分发策略, 7种: 随机、按字段、广播等。



Storm编程例子

- wordcount为例，wordcount分为1个Spout和2个Bolt
- 流程：RandomSentenceSpout→SplitSentence→WordCount

创建TopologyBuilder，设置Spout、bolt，然后提交此拓扑。

```
public static void main(String[] args) throws Exception {  
    TopologyBuilder builder = new TopologyBuilder();  
    builder.setSpout("spout", new RandomSentenceSpout(), 5);  
    //5为并发消息源任务数  
    builder.setBolt("split", new SplitSentence(), 8).shuffleGrouping("spout");  
    //8为Split并发任务数；shuffleGrouping指定了从Spout到SplitBolt的消息分发策略：随机  
    builder.setBolt("count", new WordCount(), 12).fieldsGrouping("split", new Fields("word")); ...  
    cluster.submitTopology("word-count", conf, builder.createTopology());  
    //12为计数并发任务书；fieldsGrouping指定了从SplitBolt到WordCount Bolt的消息分发策略：按字段分组  
    , 保证同一单词分配到同一task
```

Storm编程例子

Spout的作用就是源源不断的产生数据，形象的描述“水龙头”。**Spout**在**open**中先定义了一个随机数生成器，之后**Storm**框架会不断的调用**nextTuple**，每次随机从5条字符串中选取一条作为**Tuple**送到后面的**Bolt**。

```
public class RandomSentenceSpout extends BaseRichSpout {  
    @Override  
    public void open(Map conf, TopologyContext context, SpoutOutputCollector collector) {  
        _collector = collector;  
        _rand = new Random();  
    }  
  
    @Override  
    public void nextTuple() {  
        Utils.sleep(100);  
        String[] sentences = new String[]{ "the cow jumped over the moon", "an apple a day keeps the  
doctor away", "four score and seven years ago", "snow white and the seven dwarfs", "i am at two  
with nature" };  
        String sentence = sentences[_rand.nextInt(sentences.length)]; //随机抽取一条字符串  
        _collector.emit(new Values(sentence));  
    }  
}
```

Storm编程例子

Spolt丢出来的**Tuple**消息是一个多个单词组成的字符串，**SplitBolt**会先把它**Split**为多个单词

```
public static class SplitSentence extends ShellBolt implements IRichBolt {
```

```
    public SplitSentence() { //调用python脚本来拆字符串
        super("python", "splitsentence.py");
    }
```

脚本将字符串**Split**以后再**emit**（发射）出去给下一个**bolt**。注意每次发射的是单个单词。此脚本的路径是./multilang/resources/splitsentence.py

```
import storm
```

```
class SplitSentenceBolt(storm.BasicBolt):
    def process(self, tup):
        words = tup.values[0].split(" ")
        for word in words:
            storm.emit([word])
```

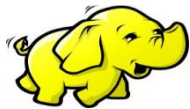
```
SplitSentenceBolt().run()
```

Storm编程例子

WordCount bolt将前面**bolt**发射出来的单词汇总起来，建立单词与词频的映射关系。由于采用了**Field Grouping**策略，**WordCount bolt**只要写入**Map**即可。

```
public static class WordCount extends BaseBasicBolt {
    Map<String, Integer> counts = new HashMap<String, Integer>();

    @Override
    public void execute(Tuple tuple, BasicOutputCollector collector) {
        String word = tuple.getString(0);
        Integer count = counts.get(word);
        if (count == null)
            count = 0;
        count++;
        counts.put(word, count); //写入Map表
        collector.emit(new Values(word, count)); //继续向后发射
    }
}
```



MR

VS

STORM

- 批处理
 - 作业完成
 - 有状态节点
 - 可扩展
-
- 保证没有数据丢失
 - 开源

- 实时处理
 - 拓扑一直运行
 - 无状态节点
 - 可扩展
-
- 保证没有数据丢失
 - 开源

Storm库

- STORM uses a lot of libraries. The most prominent are
- Clojure a new lisp programming language. Crash-course follows
- Jetty an embedded webserver. Used to host the UI of Nimbus.
- Kryo a fast serializer, used when sending tuples
- Thrift a framework to build services. Nimbus is a thrift daemon
- ZeroMQ a very fast transportation layer
- Zookeeper a distributed system for storing metadata



Storm安装

- 1 配置zookeeper集群
- storm通过zookeeper来协调整个集群。zookeeper不是用来做消息传递，因此storm不会给zookeeper带来很大的压力。把zookeeper运行在一个监督进程之下是非常关键的，因为zookeeper是一个fail-fast的进程，当它遇到任何错误的时候都会自动退出，定时的去压缩和转移zookeeper数据也是非常关键的，因为zookeeper不具备压缩和清楚数据机制，如果不设置一个cron管理这些数据，zookeeper产生的数据会很快的占满磁盘，如果zookeeper启动失败，查看一下它bin目录下的zookeeper.out文件，配置一下它的myid试试。
- 2 安装依赖到nimbus和worker节点
- storm需要依赖的是：Java 6, Python 2.6.6
- 需要注意，storm对大多数版本的依赖都做了测试，但是storm并不保证对任何版本的依赖都能正常工作。
- 3 下载解压storm发布版本到nimbus和worker节点
- 下载解压storm压缩文件到每一台机器。

Storm安装

- 4 配置storm.yaml文件
- Storm配置文件conf/storm.yaml。storm.yaml中的配置会覆盖掉default.yaml中的配置。下面配置一个集群必须修改的配置
- 1) storm.zookeeper.servers: 配置zookeeper集群的列表
- storm.zookeeper.servers:
 - – "111.222.333.444"
 - – "555.666.777.888"
- 如果zookeeper集群使用的端口不是默认端口, 配置storm.zookeeper.port。
- 2) storm.local.dir: storm的nimbus和work进程需要一个目录来存放一小部分状态数据, 比如jars、confs等等。在每台机器上创建这个目录并且赋予其相应的权限。
- storm.local.dir: "/mnt/storm"

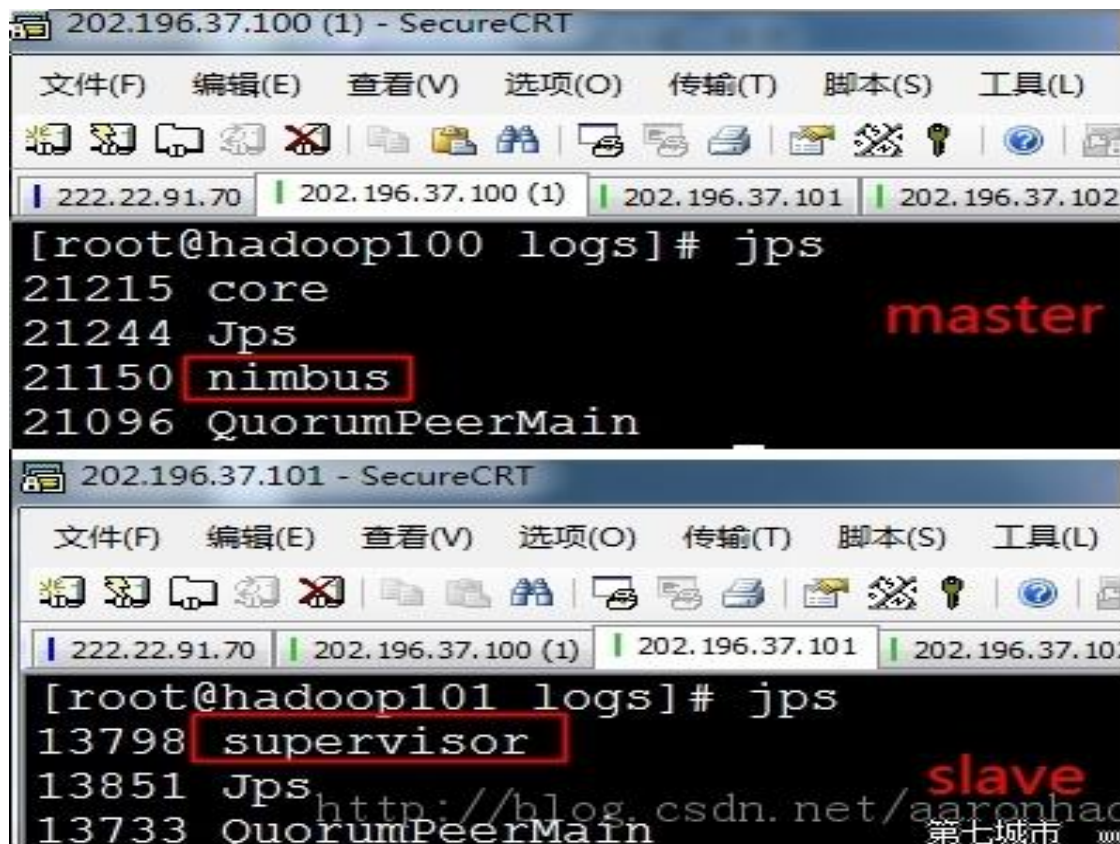
Storm安装

- 4 配置storm.yaml文件
- 3) nimbus.host: worker节点需要知道哪个机器是master节点, 以便自己从master节点下载jars和confs。
- nimbus.host: "111.222.333.44"
- 4) supervisor.slots.ports: 对于每一台worker机器, 它决定了这台机器一共可以运行多少个worker进程。每个worker进程会独占一个端口来接收消息, 这个参数就是配置了哪些端口会分配给worker进程。如果你在这配置了5个端口, 那么storm将能分配5个worker进程给这台机器, 如果配置3个端口, 那么storm也只能分配3个worker进程。storm默认分配4个worker进程到6700, 6701, 6702, 6703端口。比如:
- supervisor.slots.ports:
 - - 6700
 - - 6701
 - - 6702
 - - 6703

Storm安装

- 5 通过storm命令运行storm相关的守护进程
- 启动所有storm相关守护进程。当然，把这些进程都纳入到监督进程管理之下是很有必要的。storm也是一个fail-fast系统，这就意味着这些进程一旦遇到异常就会终止。storm之所以会这么设计，是为了它可以在任何时候安全的终止和在进程重启的时候恢复。这就是storm为不在进程中保存相关状态的原因，如果nimbus或supervisor节点重启，运行着的topoloies不会受到任何影响。下面就是启动storm相关进程的命令：
- Nimbus：在master节点运行 “bin/storm nimbus”
- Supervisor：在每一台worker节点运行 “bin/storm supervisor” , supervisor进程负责在worker节点上启动和停止相应的worker进程
- UI：运行 “bin/storm ui” ，一个通过页面管理和展示storm集群运行状态的工具，可以通过 “http://nimbus host:8080” 来访问。
- 启动storm服务进程相当简单直接，storm产生的log会保存在各台机器的storm/logs目录中，storm通过logback管理它的日志，我们可以通过修改其logback.xml文件来改变其log的目录及内容。

Storm



The image displays two terminal windows from SecureCRT, showing the output of the 'jps' command on different nodes in a Storm cluster.

Top Window: 202.196.37.100 (1) - SecureCRT

```
[root@hadoop100 logs]# jps
21215 core
21244 Jps
21150 nimbus
21096 QuorumPeerMain
```

The output shows the processes running on the master node. The word "master" is written in red on the right side of the terminal. The process "nimbus" is highlighted with a red box.

Bottom Window: 202.196.37.101 - SecureCRT

```
[root@hadoop101 logs]# jps
13798 supervisor
13851 Jps
13733 QuorumPeerMain
```

The output shows the processes running on the slave node. The word "slave" is written in red on the right side of the terminal. The process "supervisor" is highlighted with a red box. A watermark "http://blog.csdn.net/aaronhad" and "第七城市" are visible at the bottom of the terminal window.

Storm

192.168.80.100:18080

Storm UI

storm集群配置启动成功..... QQ男主角献

Cluster Summary

Version	Nimbus uptime	Supervisors	Used slots	Free slots	Total slots	Executors	Tasks
0.8.2	4m 49s	3	0	12	12	0	0

Topology summary

Name	Id	Status	Uptime	Num workers	Num executors	Num tasks
------	----	--------	--------	-------------	---------------	-----------

Supervisor summary

Id	Host	Uptime	Slots	Used slots
5148d741-6719-4e19-ad81-6418975de41e	storm102	2m 49s	4	0
5edec771-f088-4859-a645-cdb4b9839a05	storm101	3m 37s	4	0
ff9a86f4-f979-4f58-b6c4-cec175a1bc4f	storm100	54s	4	0

Nimbus Configuration

Key	Value
dev.zookeeper.path	/tmp/dev-storm-zookeeper

Storm发展趋势

Storm从纯计算框架演变成了存储和计算的实时计算框架
Storm利用Trident，提供更加友好的接口，同时可定制
Storm scheduler特性针对不同的应用场景基于实时QL进行
优化

Storm一直在朝着融入mesos框架方向努力

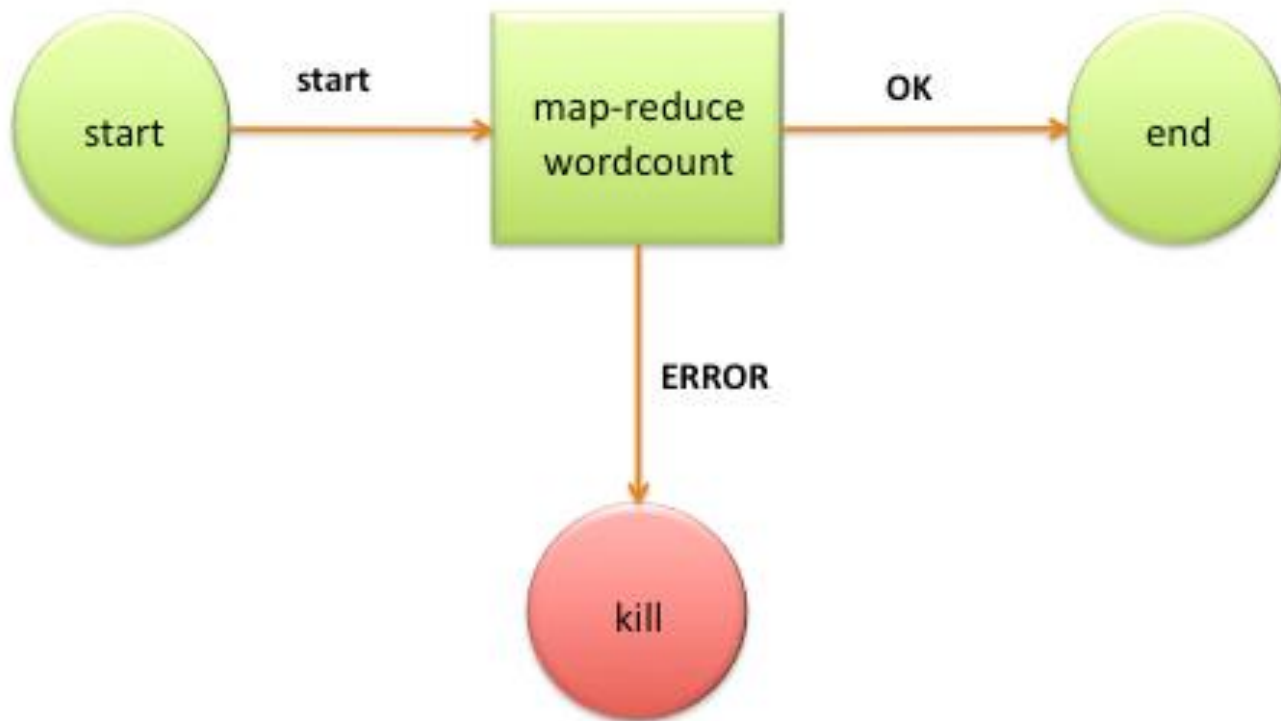
Storm在实现细节上不断地优化，使用很多优秀开源产品，
包括kryo, Disruptor, curator等。

OOIZE

- 大多数统计分析任务不是由一个map-reduce完成的，需要多个map-reduce组合完成，或者还需要其他程序配合。
- 各map-reduce或其他任务间有依赖关系
- oozie的翻译是驯象人。是管理hadoop jobs的工作流调度系统，是使用HPDL语言（一种XML流程定义语言）描述工作流程图（DAG有向无环图）。
- 使用数据库来存储以下内容：
 - 1) 工作流定义。
 - 2) 当前运行的工作流实例，包括实例的状态和变量。
- 需要组合不同任务：MR、HADOOP命令、SSH、JAVA、Sqoop、MAIL

OOIZE是hadoop的工作流软件，就是为了这个需求产生的

OOIZE



OOIZE支持的动作

- Email 动作——支持发送邮件
- Shell 动作——支持操作系统shell命令
- Hive动作——支持hive脚本
- Sqoop动作——支持Sqoop命令
- Ssh动作——支持ssh命令
- Distcp动作——支持两个hadoop集群间拷贝数据
- 自定义动作

OOZIE安装

- 安装基础环境
 - Maven-3.9.0
 - Hadoop-2.7.1
- Server 安装
 - 解压oozie-4.3.0.tar.gz
 - 将oozie的安装包解压到/opt/bigdata/oozie目录下

OOZIE安装

- 修改pom.xml
- `<targetJavaVersion>1.8</targetJavaVersion>`
- `<sourceJavaVersion>1.8</sourceJavaVersion>`
- `<minJavaVersion>1.7</minJavaVersion>`
- `<hadoop.version>2.7.1</hadoop.version>`
- mvn编译
- `/opt/bigdata/oozie/bin/mkdistro.sh -DskipTests -Dhadoop.version=2.7.1`
- 最后会在/opt/bigdata/oozie/distro/target目录生成编译好的oozie-4.3.0-distro.tar.gz压缩包。将其解压到/data目录下：
- `/data/oozie-4.3.0`
- 并配置好环境变量：
- `export OOZIE_HOME=/data/oozie-4.3.0`

OOIZE安装

- 添加相关jar包
- oozie server需要用到一个js库，需要下载ext-2.2.zip这个文件放到libext文件夹里。否则就看不到了oozie的web控制台。（打war包的时候会报：INFO: Oozie webconsole disabled, ExtJS library not specified）
- mkdir libext
- 下载ext-2.2.zip放到libext目录下面
- 把hadoop的一些jar包也放到这个libext文件夹内：
- cp \${HADOOP_HOME}/share/hadoop/*.jar libext/
- cp \${HADOOP_HOME}/share/hadoop/*/lib/*.jar libext/
- 注意：oozie server默认使用tomcat 6.0.41，而hadoop也有内置的server，如果按照上面两个命令把hadoop依赖的jar包都拷贝过去，有可能出现冲突，这两个server使用的servlet、jsp版本很可能不一样。这里需要把这几个jar包删除，不要放到libext中
- jsp-api-2.1.jar
- oozie server还需要依赖个数据库，常用的是mysql，所以要把mysql的驱动jar包也放到libext中。 mysql-connector-java-5.1.36.jar

OOIZE安装

- `core-site.xml`
- 修改/data/oozie-4.3.0/conf/hadoop-conf/core-site.xml
- 注意：因为Namenode配置了HA，所以需要配置fs.defaultFS，并且把hdfs-site.xml拷贝到/data/oozie-4.3.0/conf/hadoop-conf目录下
- 打war包
- 可以生产server的war包了：
- `oozie-setup.sh prepare-war`
- 打印日志：
- `setting CATALINA_OPTS=" $CATALINA_OPTS -Xmx1024m"`
- `INFO: Adding extension: /data/oozie-4.3.0/libext/activation-1.1.jar`
- ...这里省略很多jar包
- `INFO: Adding extension: /data/oozie-4.3.0/libext/commons-codec-1.4.jar`
- `New Oozie WAR file with added 'ExtJS library, JARs' at /data/oozie-4.3.0/oozie-server/webapps/oozie.war`
- war包在/data/oozie-4.3.0/oozie-server/webapps/下

OOIZE安装

- 共享HDFS配置
- 在启动之前，还需要把执行下面的命令：
- `/data/oozie-4.3.0/bin/oozie-setup.sh sharelib create -fs hdfs://appcluster:8020`
- 这把hadoop的一些jar包传到hdfs上，后面会用到。
- 成功后提示：the destination path for sharelib is:
`/user/root/share/lib/lib_20161214170752`
- 启动与停止
- `/data/oozie-4.3.0/bin/oozied.sh start`
- `/data/oozie-4.3.0/bin/oozied.sh stop`

OOIZE例子wordcount

```
<workflow-app name='wordcount-wf' xmlns="uri:oozie:workflow:0.1">
```

```
  <start to='wordcount'/>
```

```
    <action name='wordcount'>
```

```
      <map-reduce>
```

```
        <job-tracker>${jobTracker}</job-tracker>
```

```
        <name-node>${nameNode}</name-node>
```

```
        <configuration>
```

```
          <property>
```

```
            <name>mapred.mapper.class</name>
```

```
            <value>org.myorg.WordCount.Map</value>
```

```
          </property>
```

```
          <property>
```

```
            <name>mapred.reducer.class</name>
```

```
            <value>org.myorg.WordCount.Reduce</value>
```

```
          </property>
```

OOIZE例子wordcount

```
<property>
```

```
    <name>mapred.input.dir</name>
```

```
    <value>${inputDir}</value>
```

```
</property>
```

```
<property>
```

```
    <name>mapred.output.dir</name>
```

```
    <value>${outputDir}</value>
```

```
</property>
```

```
</configuration>
```

```
</map-reduce>
```

```
<ok to='end'/>
```

```
<error to='end'/>
```

```
</action>
```

```
<kill name='kill'> <message>Something went wrong: ${wf:errorCode('wordcount')}</message> </kill/> <end name='end'/>
```

```
</workflow-app>
```

OOIIZE例子wordcount

```
<kill name='kill'>
```

```
<message>Something went wrong: ${wf:errorCode('wordcount')}</message>
```

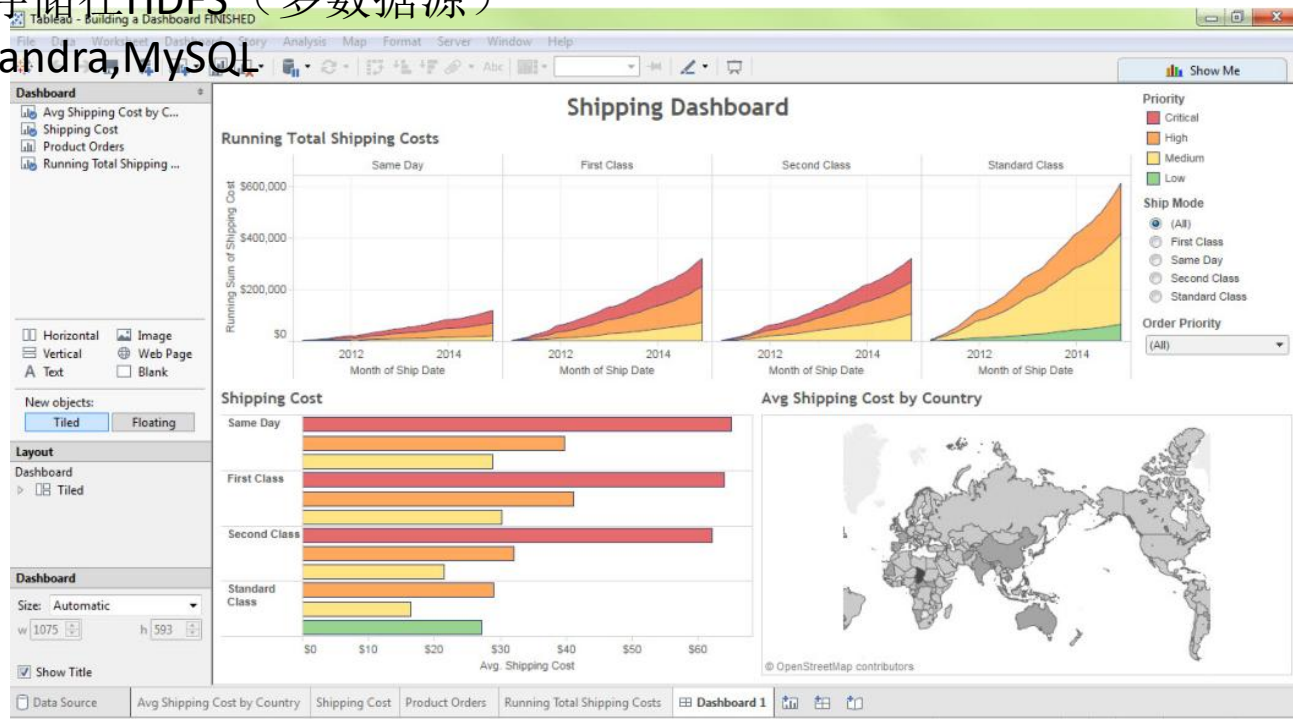
```
</kill/>
```

```
<end name='end'/>
```

```
</workflow-app>
```

Presto是什么?

- ◆ Dashboard, BI工具, 数据分析师 (低延迟的大数据交互式查询引擎)
 - Hive太慢
 - MySQL, PostgreSQL很难处理大数据
- ◆ 有一些数据并不存储在HDFS (多数据源)
 - HBase, Cassandra, MySQL



Presto是什么?

- Facebook开源的
- 基于内存的，分布式SQL交互式查询引擎
- Massively parallel processing(MPP)
- 支持任意数据源
 - ✓ 扩展式Connector组件
 - ✓ 数据规模GB~PB级

facebook

airbnb

GROUPON™

twitter

NETFLIX

Dropbox

UBER

LinkedIn

Presto是什么?

➤ Facebook

✓ 部署了多个生产环境（超过100个节点）

- 超过300PB数据
- 大量MySQL实例
- 大量使用SSD

✓ 每天超过1000个用户

✓ 10-100个并发查询

➤ Netflix

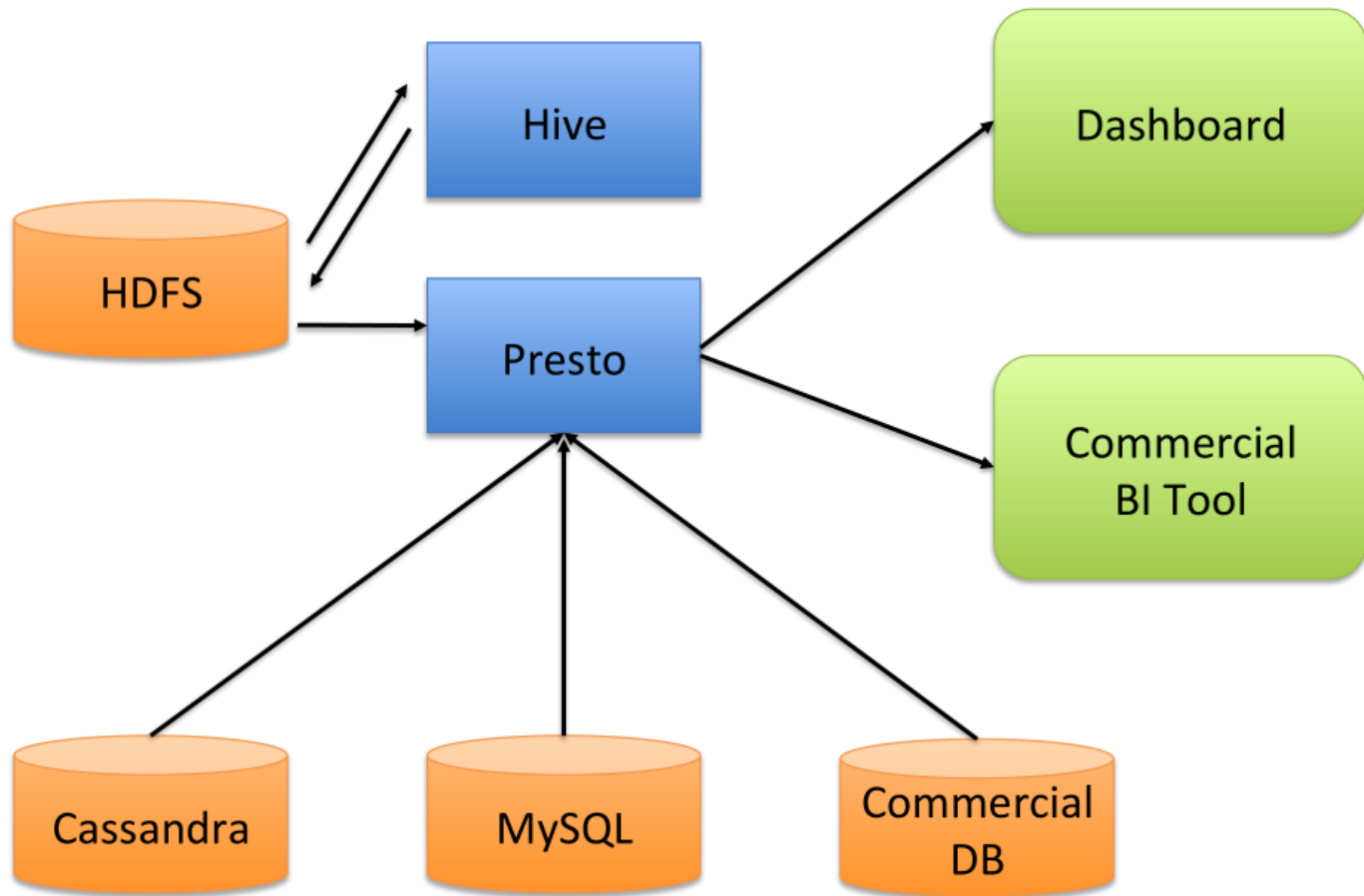
✓ 部署了超过200个结节点

✓ 在S3上存储了超过25PB数据（Parquet格式）

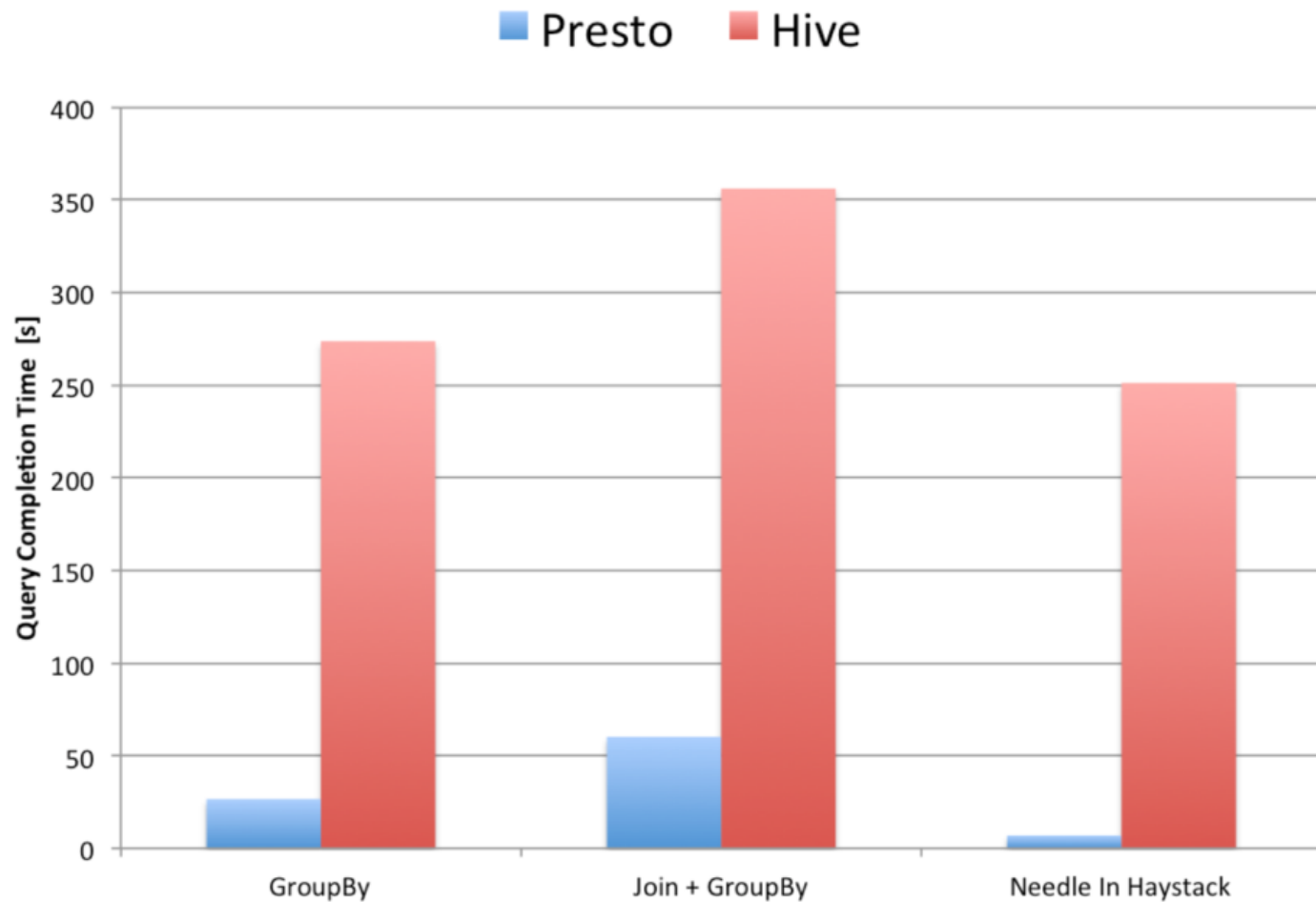
✓ 每天超过350个活跃用户，超过3000个查询

- 完全基于内存的并行计算
- MPP架构，过个节点管道式执行
- 向量化计算
- 多线程处理
- 动态编译执行计划
- 优化的ORC和Parquet Reader
- 类BlinkDB的近似查询

Presto是什么?



Presto是什么?



Presto可以做什么？

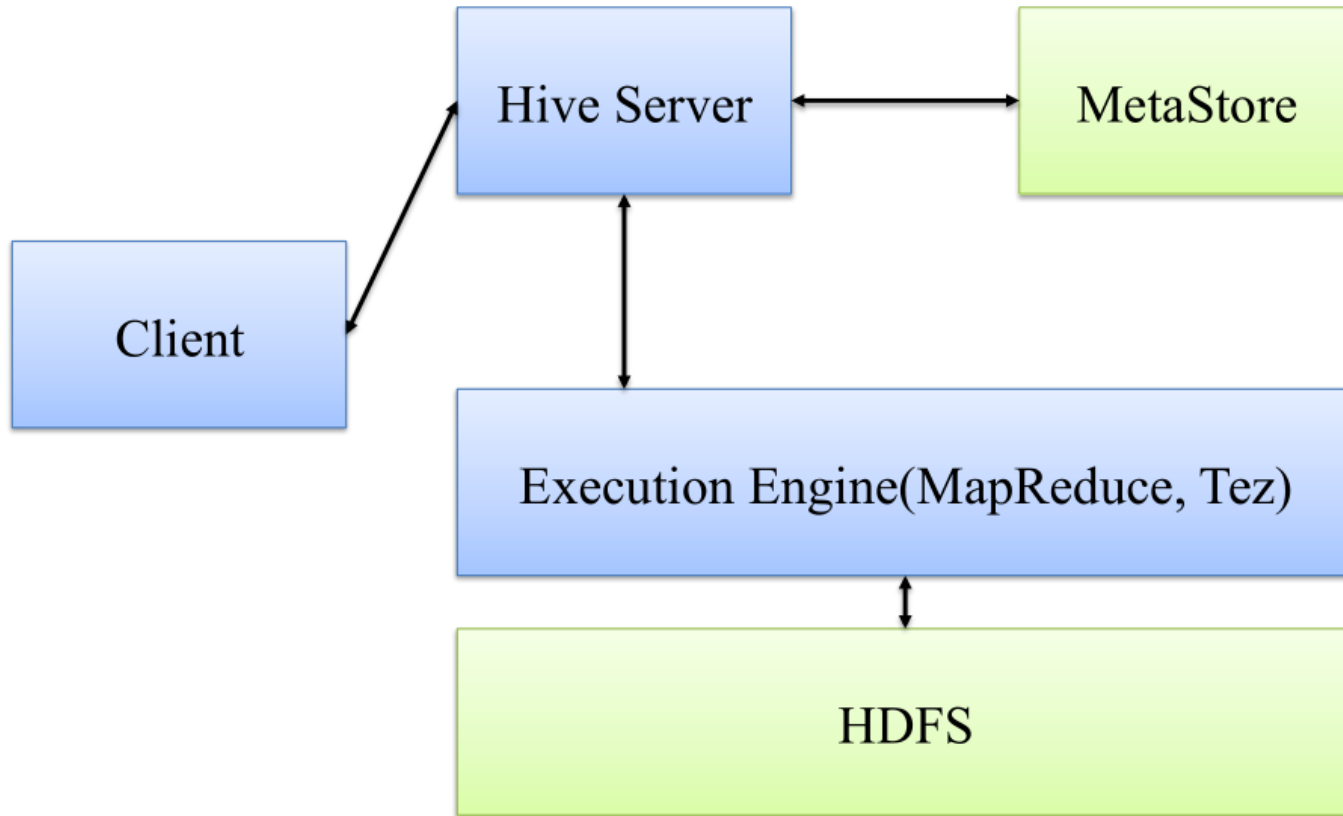
◆ 适合做什么

- PB级海量数据复杂分析
- 交互式SQL查询
- ANSI SQL(not HQL)
- 支持跨数据源查询

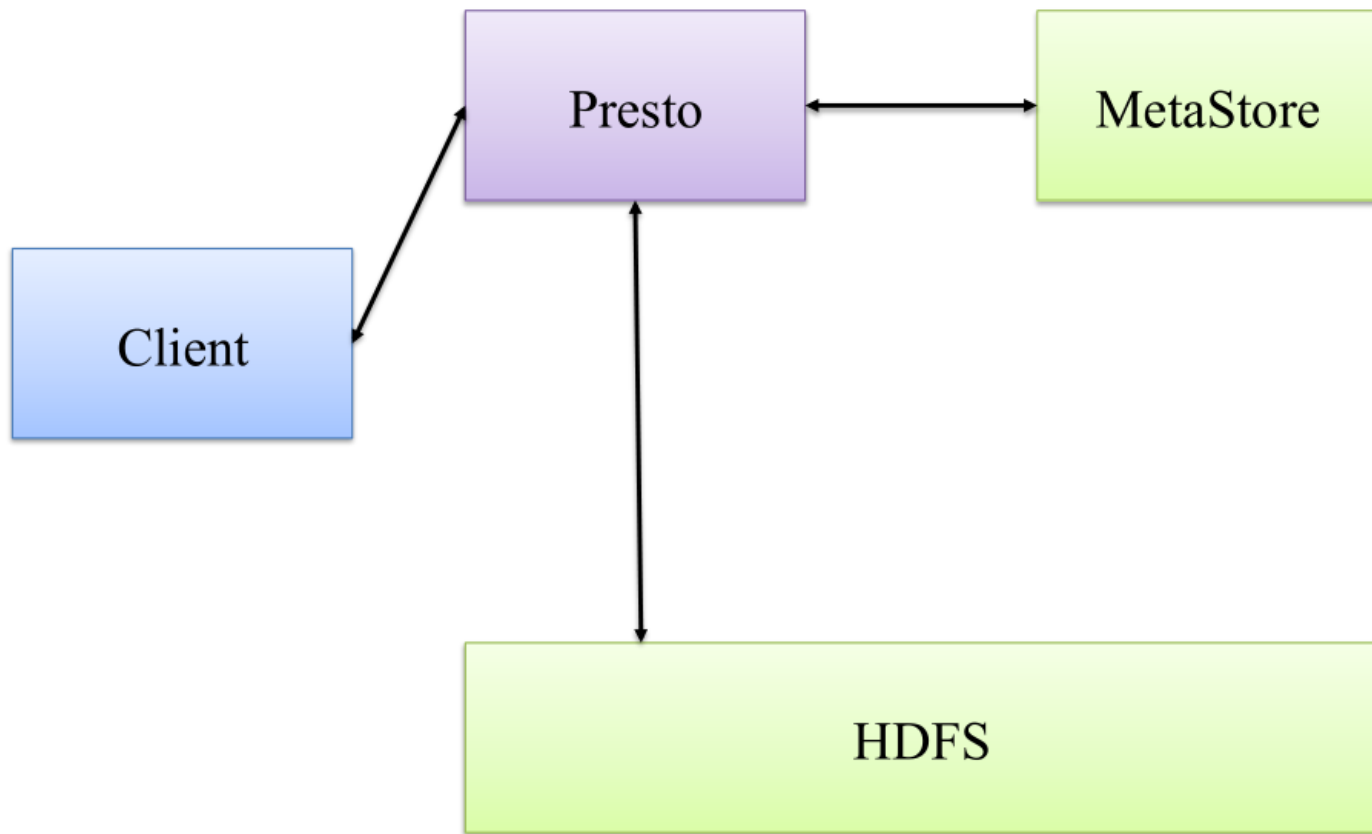
◆ 不适合做什么

- 多个大表的join操作

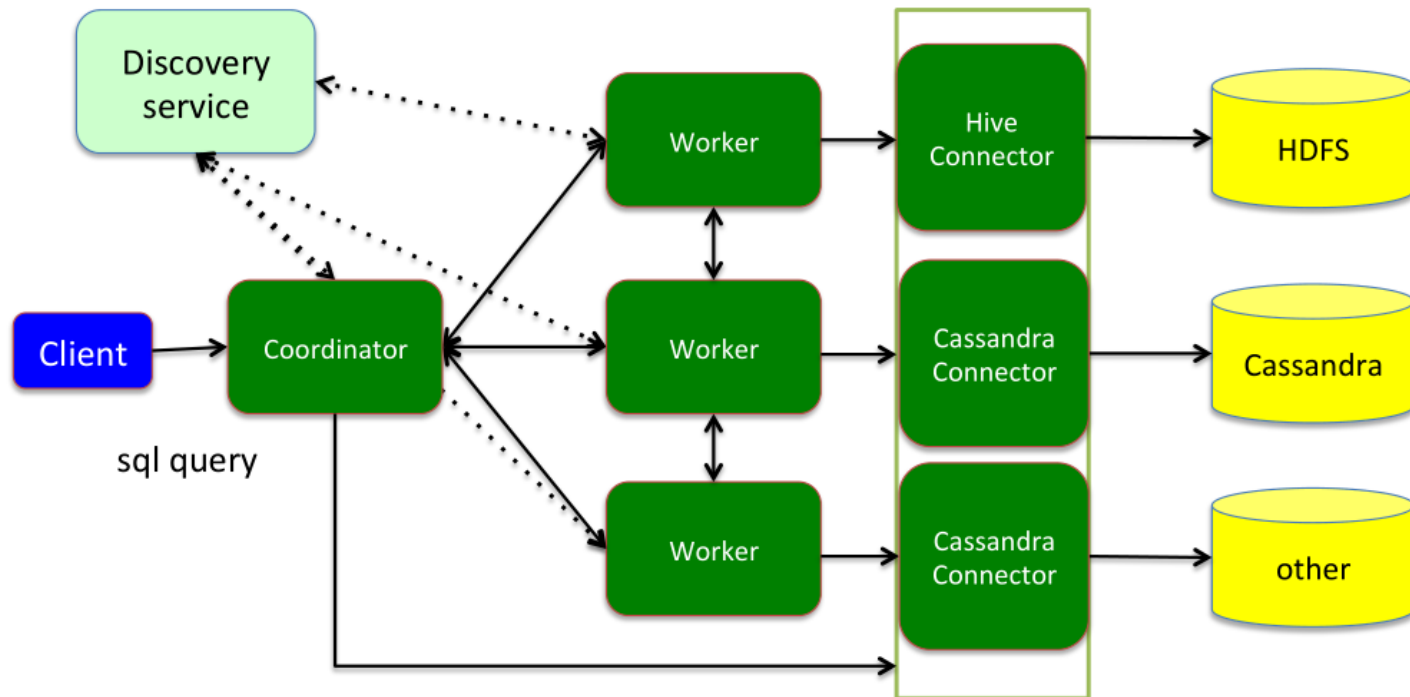
Hive架构



Presto架构



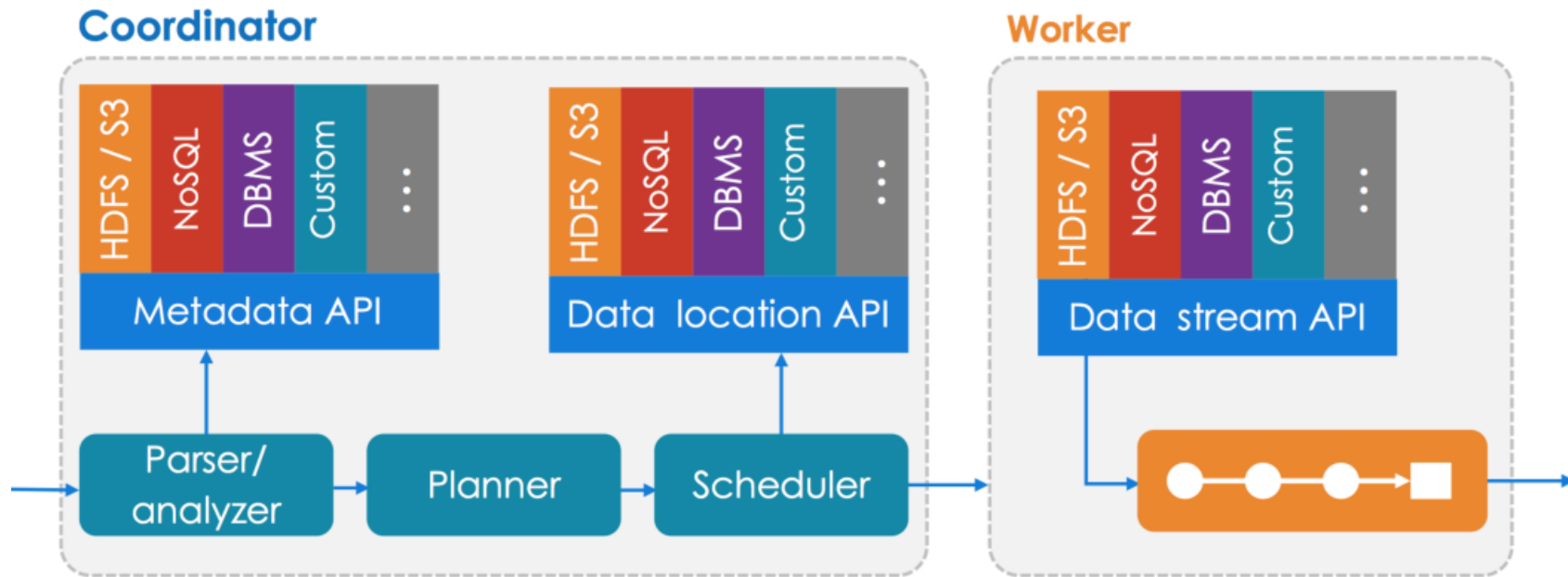
Presto架构



Presto架构

- 三种服务
 - ✓ Coordinator, Worker, Discovery service
- 通过connector plugin获取数据和元信息
 - ✓ Presto不是一个数据
 - ✓ Presto为其他数据存储系统提供了SQL能力
- 客户端协议: HTTP+JSON
 - ✓ 支持的语言包括: Ruby, Python, PHP, Java (JDBC)
 - Coordinator
 - ✓ 解析SQL语句
 - ✓ 生成执行计划
 - ✓ 分发执行任务给worker节点执行
 - Worker
 - ✓ 负责实际执行查询任务
 - Discovery service
 - ✓ Worker节点启动后向Discovery Server服务注册
 - ✓ Coordinator从Discovery Server获得Worker节点

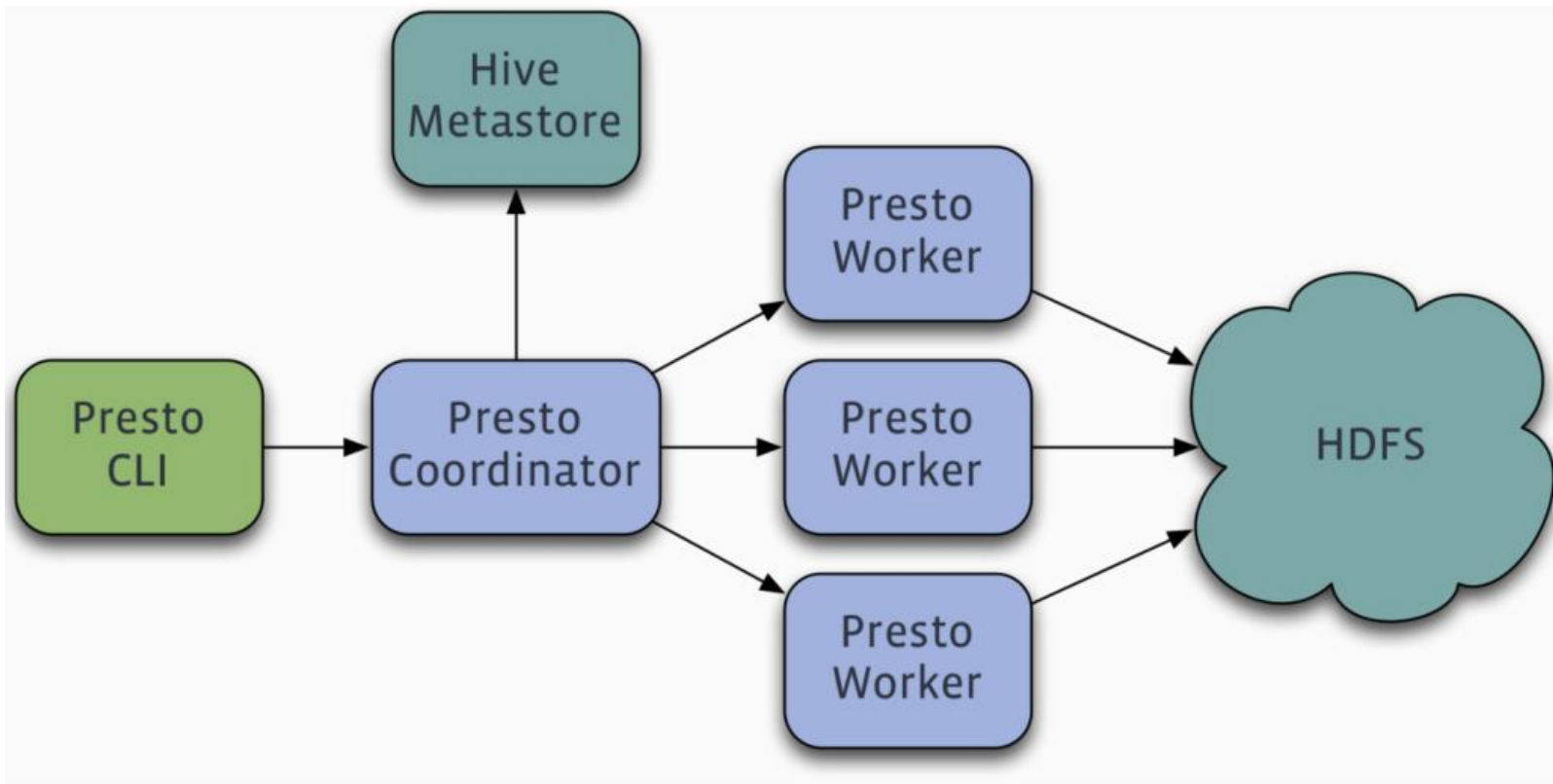
Presto扩展: Connector



Presto支持数据源、存储格式、部署

- Hadoop/Hive connector与存储格式
 - ✓ HDFS
 - ✓ ORC, RCFILE, Parquet, SequenceFile, Text
- 开源数据存储系统
 - ✓ MySQL&PostgreSQL
 - ✓ Cassandra
 - ✓ Kafka
 - ✓ Redis
- 其他
 - ✓ MongoDB
 - ✓ ElasticSearch
 - ✓ HBase
- 系统需要
 - ✓ HDFS Linux or Mac OS X
 - ✓ Java8,64-bit
 - ✓ Python 2.4+
- Hive Connector
 - ✓ 版本
 - Apache Hadoop 1.x
 - Apache Hadoop 2.x
 - Cloudera CDH4
 - Cloudera CDH5
 - ✓ Hive MetaStore
 - ✓ HDFS

Presto部署



Presto部署

- 下载[Presto安装包](#)
- 解压缩后创建etc文件夹进行配置
 - ✓ Catalog Properties
 - ✓ Config PROPERTIES
 - ✓ JVM Config
 - ✓ Node Properties

```
[bigdata@bigdata presto-server-0.157]$ ls etc
catalog  config.properties  jvm.config  node.properties
```

Presto部署

➤ Catalog Properties

- ✓ 配置Connector
- ✓ etc/catalog/hive.properties

```
[bigdata@bigdata presto-server-0.157]$ ls -l etc/catalog
total 4
-rw-r--r--. 1 bigdata bigdata 195 Jan  8 02:56 hive.properties
```

```
connector.name=hive-hadoop2
hive.metastore.uri=thrift://bigdata:9083
hive.config.resources=/home/bigdata/hadoop-2.7.3/etc/hadoop/core-site.xml,/home/bigdata/hadoop-2.7.3/etc/hadoop/hdfs-site.xml
```

Presto部署

➤ Config Properties

- ✓ 配置Presto Server
- ✓ etc/config.properties

```
coordinator=true
node-scheduler.include-coordinator=true
http-server.http.port=8080
query.max-memory=512MB
query.max-memory-per-node=512MB
discovery-server.enabled=true
discovery.uri=http://bigdata:8080
```

Presto部署

➤ JVM Properties

- ✓ 配置JVM
- ✓ jvm.properties

```
-server  
-Xmx1G  
-XX:+UseG1GC  
-XX:G1HeapRegionSize=32M  
-XX:+UseGCOverheadLimit  
-XX:+ExplicitGCInvokesConcurrent  
-XX:+HeapDumpOnOutOfMemoryError  
-XX:OnOutOfMemoryError=kill -9 %p
```

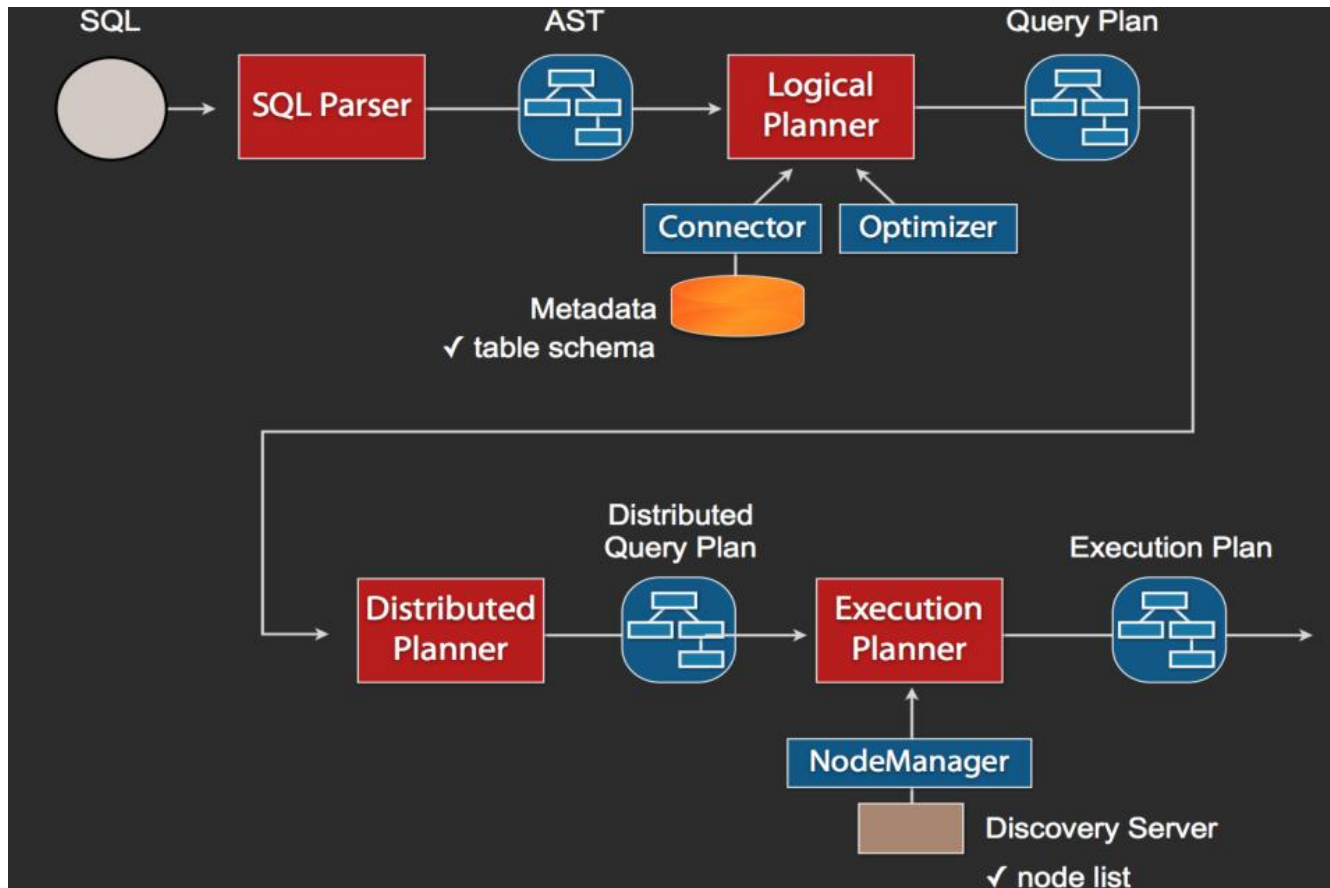
Presto部署

➤ Node Properties

- ✓ 配置每一个节点
- ✓ node.properties

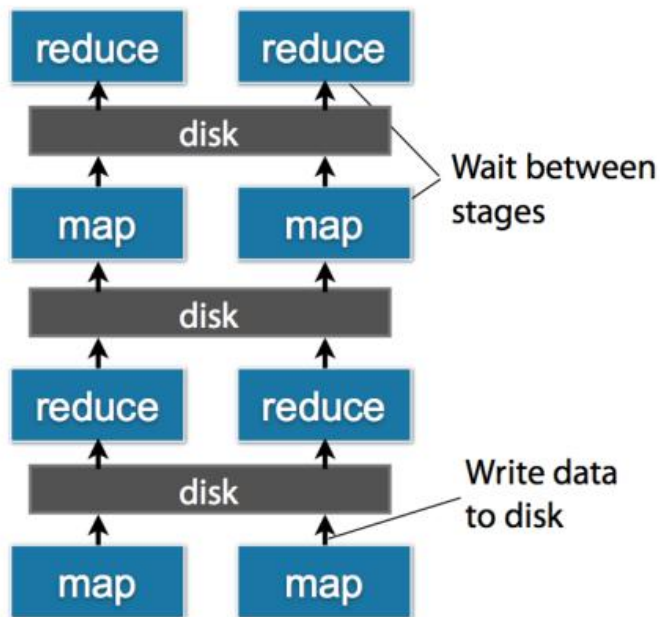
```
node.environment=production  
node.id=bigdata  
node.data-dir=/home/bigdata/presto-server-0.157/presto_data
```

Presto部署中SQL运行过程：整体流程

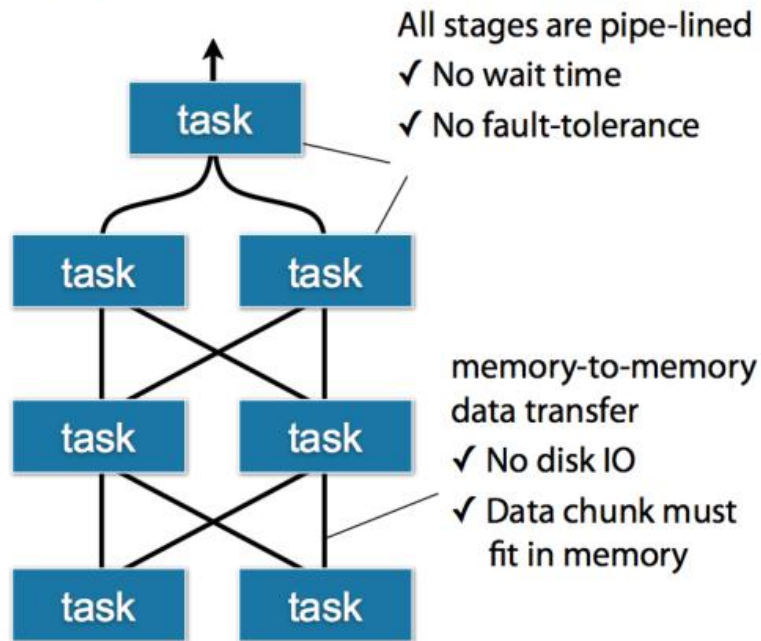


Presto部署中SQL运行过程

MapReduce



Presto



Presto部署中SQL运行过程：查询计划

SQL

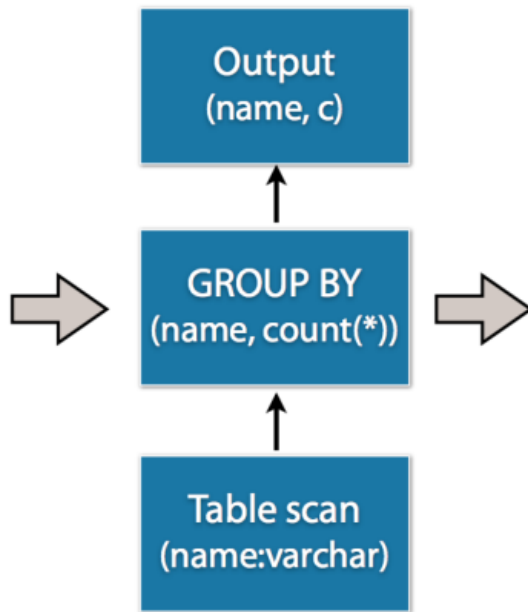
```
SELECT
  name,
  count(*) AS c
FROM impressions
GROUP BY name
```

+

Table schema

```
impressions (
  name varchar
  time bigint
)
```

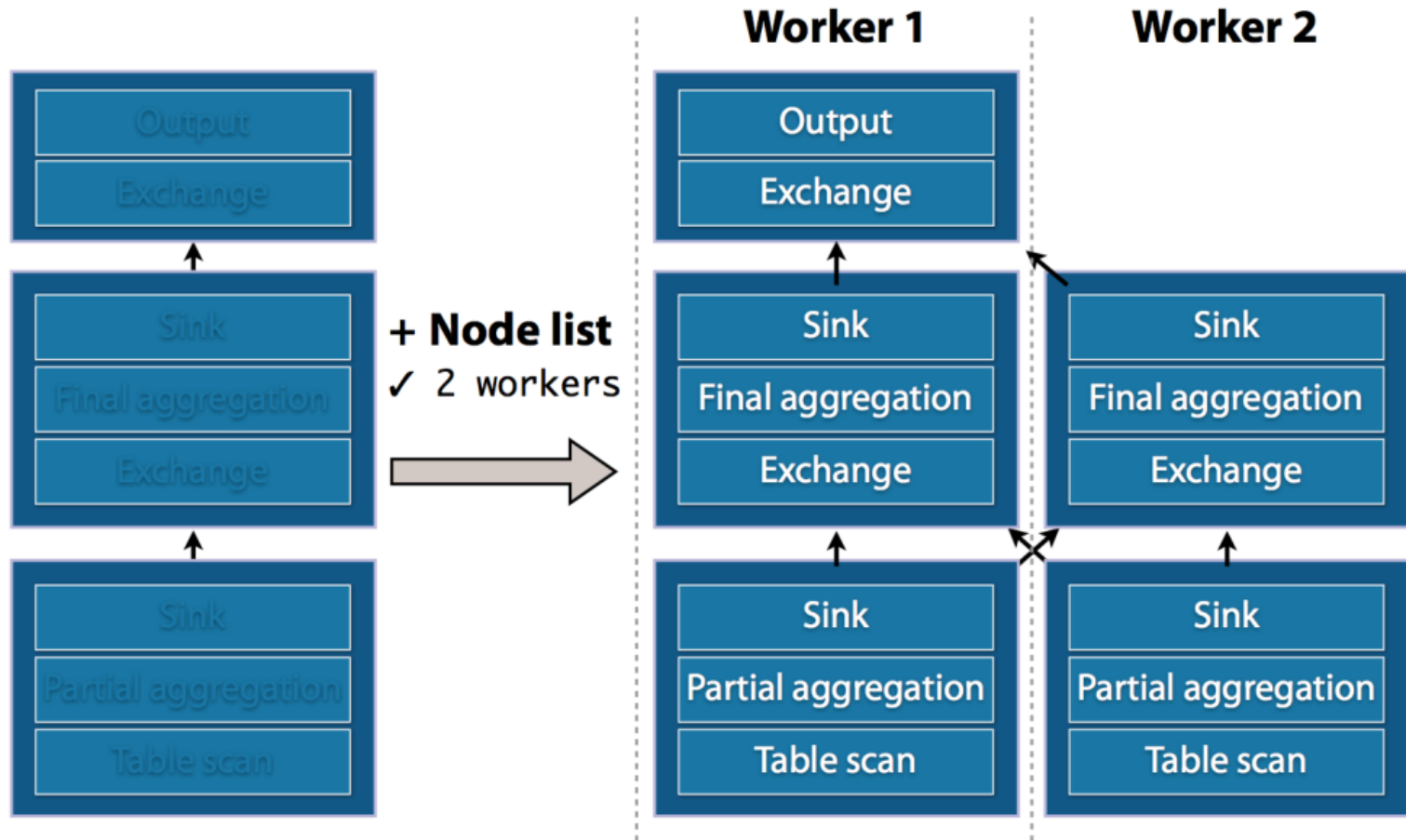
Logical query plan



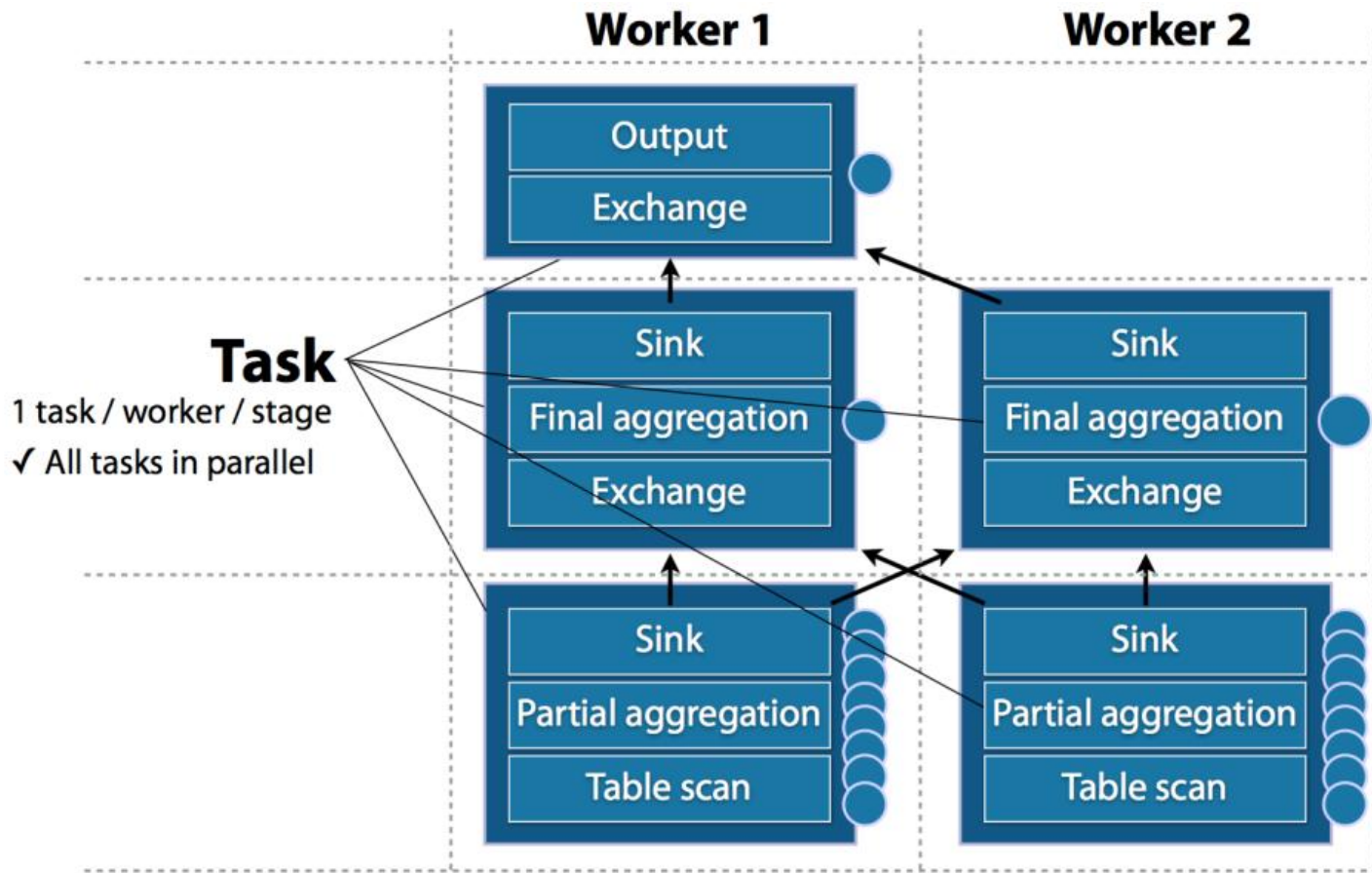
Distributed query plan



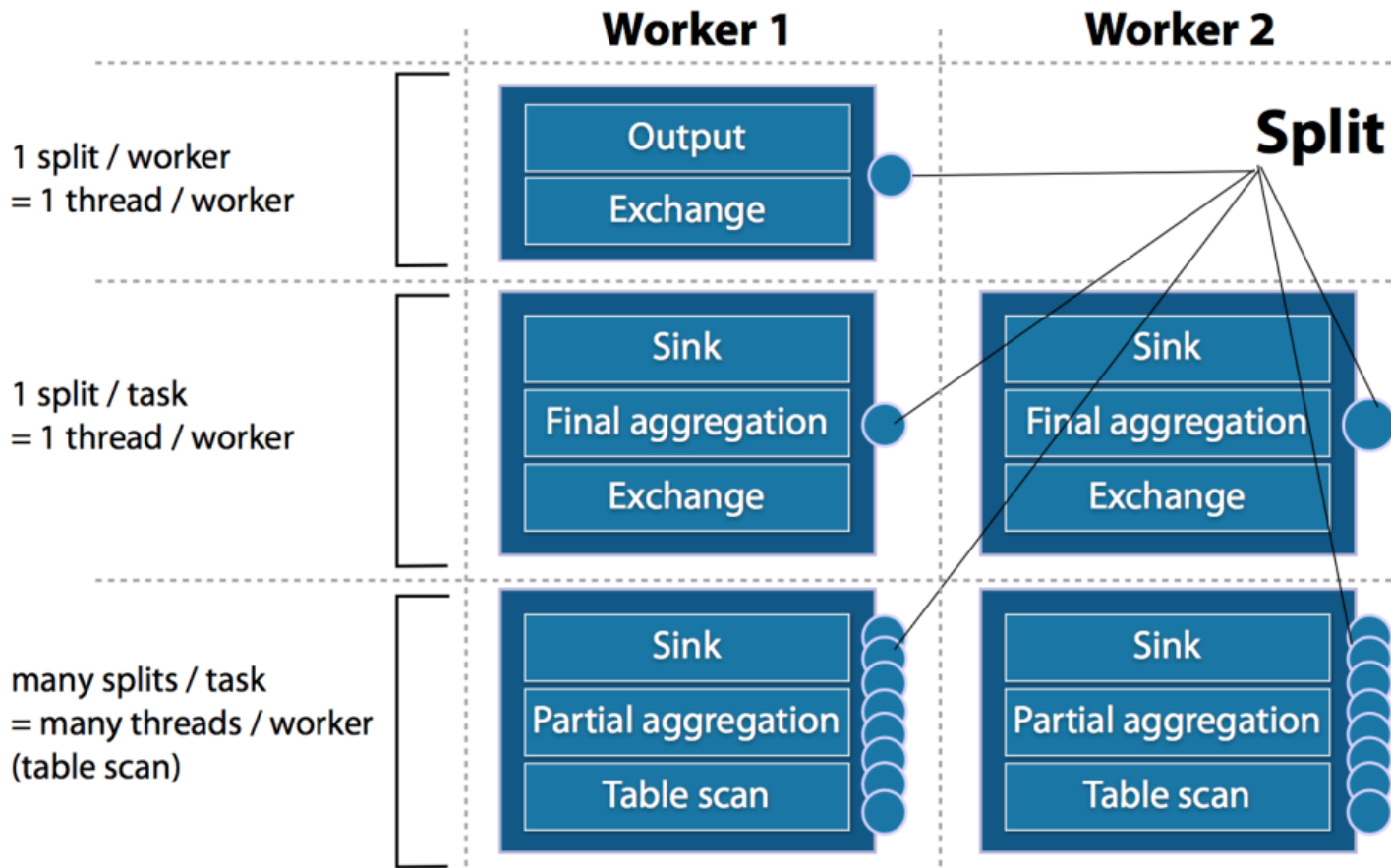
Presto部署中SQL运行过程：执行计划



Presto部署中SQL运行过程：执行计划



Presto部署中SQL运行过程：执行计划



访问Presto

- Presto CLI（命令行）

Presto—server localhost:8080 --catalog hive --schema default

<https://prestodb.io/docs/current/installation/cli.html>

- JDBC

<https://prestodb.io/docs/current/installation/jdbc.html>

- ODBC

<https://github.com/prestodb/presto-odbc>

- PyHive(python)

<https://github.com/dropbox/PyHive>

Presto对SQL的支持

```
[ WITH with_query [, ...] ]  
SELECT [ ALL | DISTINCT ] select_expr [, ...]  
[ FROM table1 [[ INNER | OUTER ] JOIN table2 ON (...)]  
[ WHERE condition]  
[ GROUP BY expression [, ...] ]  
[ HAVING condition]  
[ UNION [ ALL | DISTINCT ] select]  
[ ORDER BY expression [ ASC | DESC ] [, ...] ]  
[ LIMIT [ count | ALL ] ]
```

- 此外, 还支持
 - ✓ 窗口函数
 - ✓ 估算函数(比如distinct)
 - ✓ 复杂子查询
 - ✓ ROLLUP, CUBE等

Presto监控和配置： 监控

- WebUI
 - ✓ Query基本状态的查询
- JMX HTTP API
 - ✓ GET/v1/Jmx/mbean[/{objectName}]
 - com.facebook.presto.execution:name=TaskManager
 - com.facebook.presto.execution:name=QueryManager
 - com.facebook.presto.execution:name=NodeScheduler
- 事件通知
 - ✓ EventListener
 - query start, query complete

Presto监控和配置：配置

- 执行计划 (Coordinator)
 - ✓ `node-scheduler.include-coordinator`
 - 是否让coordinator运行task
 - ✓ `query.initia1-hash-partitions`
 - 每个GROUPBY操作使用的hash bucket(=tasks)最大数目(default: 8)
 - ✓ `node-schedulerm1n-candidates`
 - 每个stage并发运行过程中可使用的最大worker数目 (default: 10)
 - ✓ `node-schedulerinclude-coordinator`
 - 是否让coordinator运行task
 - ✓ `queryschedule-split-batch-size`
 - 每个stage一次运行的split(=task)个数

Presto监控和配置：配置

➤ 任务执行 (worker)

- ✓ `query.max-memory` (default: 20GB)
 - 一个查询可以使用的最大集群内存
 - 控制集群资源使用,防止一个大查询占住集群所有资源
 - 使用`resource_overcommit`可以突破限制
- ✓ `query.max-memory-per-node` (default: 1GB)
 - 一个查询在一个节点上可以使用的最大内存
- ✓ 举例
 - Presto集群配置: 120G*40
 - `query.max-memory=1TB`
 - `query.max-memory=1TB`

Presto监控和配置：配置

➤ 任务执行(worker)

- ✓ `query.max-run-time` (default: 100d)
 - 一个查询可以运行的最大时间
 - 防止用户提交一个长时间查询阻塞其他查询
- ✓ `task.max-worker-threads` (default: Node CPUs*4)
 - 每个worker同时运行的split个数
 - 每个worker同时运行的split个数

➤ 队列(Queue)

- ✓ 资源隔离,查询可以提交到相应队列中
- ✓ 每个队列可以配置AcL(权限)
- ✓ 每个队列可以配置Quota
 - 可以并发运行查询的数量
 - 排队的最大数

Presto监控和配置：配置

➤ 队列（Queue）

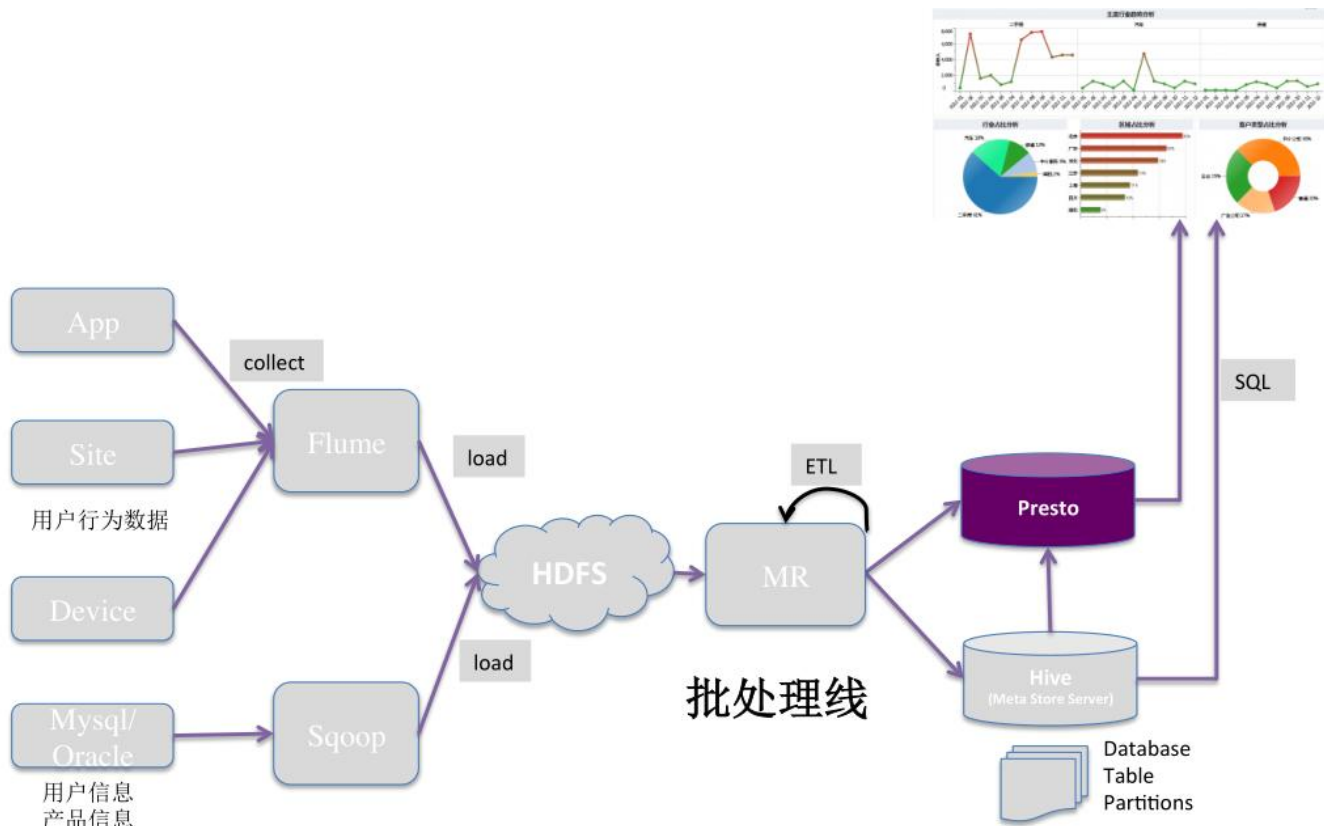
```
{
  "queues": {
    "user.${USER}": {
      "maxConcurrent": 5,
      "maxQueued": 20
    },
    "user_pipeline.${USER}": {
      "maxConcurrent": 1,
      "maxQueued": 10
    },
    "pipeline": {
      "maxConcurrent": 10,
      "maxQueued": 100
    },
    "admin": {
      "maxConcurrent": 100,
      "maxQueued": 100
    },
    "global": {
      "maxConcurrent": 100,
      "maxQueued": 1000
    }
  },
}
```

```
"rules": [
  {
    "user": "bob",
    "queues": ["admin"]
  },
  {
    "source": ".*pipeline.*",
    "queues": [
      "user_pipeline.${USER}",
      "pipeline",
      "global"
    ]
  },
  {
    "queues": [
      "user.${USER}",
      "global"
    ]
  }
]
```

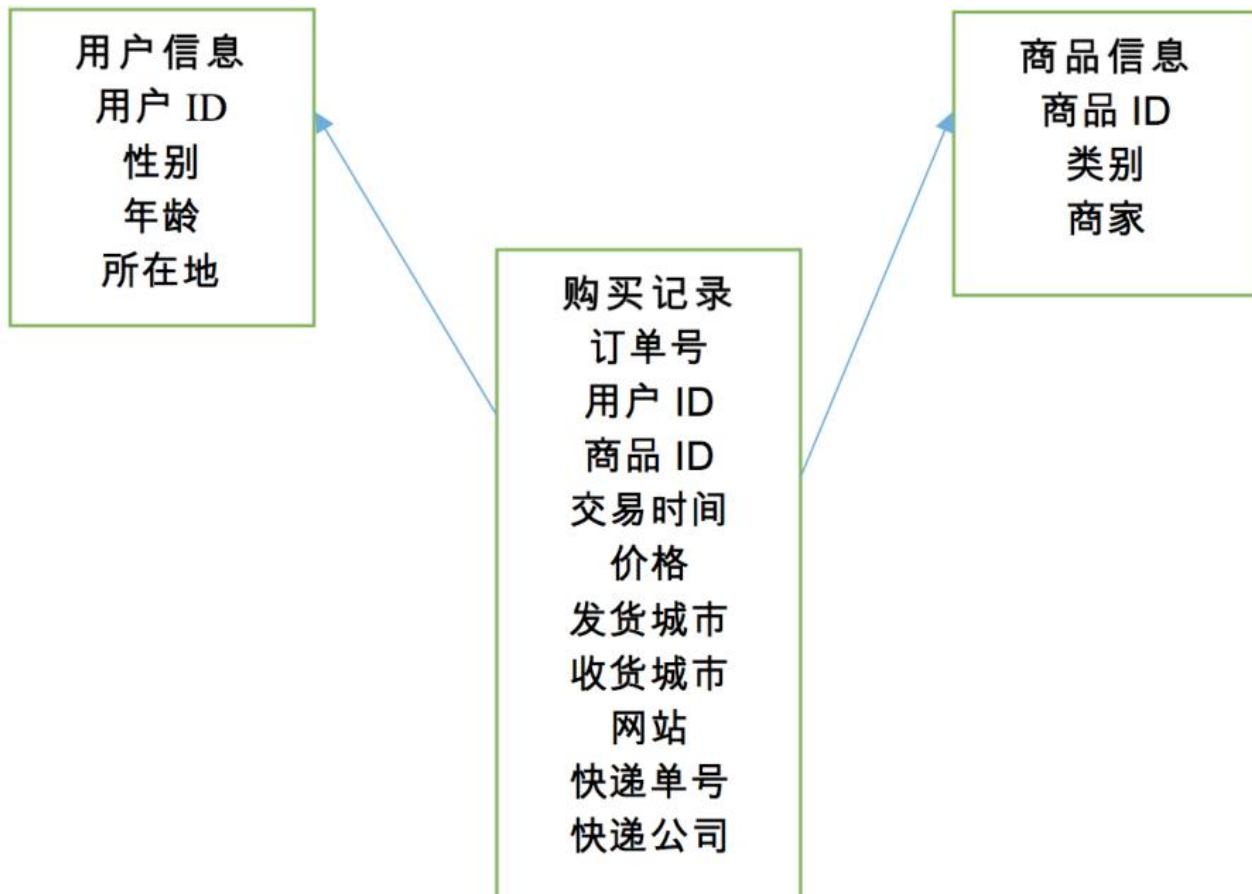
大数据OLAP引擎对比

- Presto
内存计算
- Druid
时序，数据放内存，索引
- Spark Core
基于Spark SQL
- Kylin
Cube预计算

分布式日志分析系统



分布式日志分析系统：表组织结构



分布式日志分析系统：数据查询模块

统计不同性别喜欢购买的品牌排行

```
select gender, brand, count(*) as purchase_count  
from record_orc join user_dimension_orc on record_orc.uid=user_dimension_orc.uid  
join brand_dimension_orc on record_orc.bid=brand_dimension_orc.bid  
group by gender, brand  
order by gender, purchase_count DESC
```

Hive	68 seconds
Presto	6 seconds

分布式日志分析系统：数据查询模块

统计各年龄段用户消费总额

```
select cast((year(CURRENT_DATE)-year(birth)) as integer) as age,sum(price) as totalPrice  
from record join user_dimension on record.uid=user_dimension.uid  
group by cast((year(CURRENT_DATE)-year(birth)) as integer)  
order by totalPrice desc
```

查询各品牌销售总额

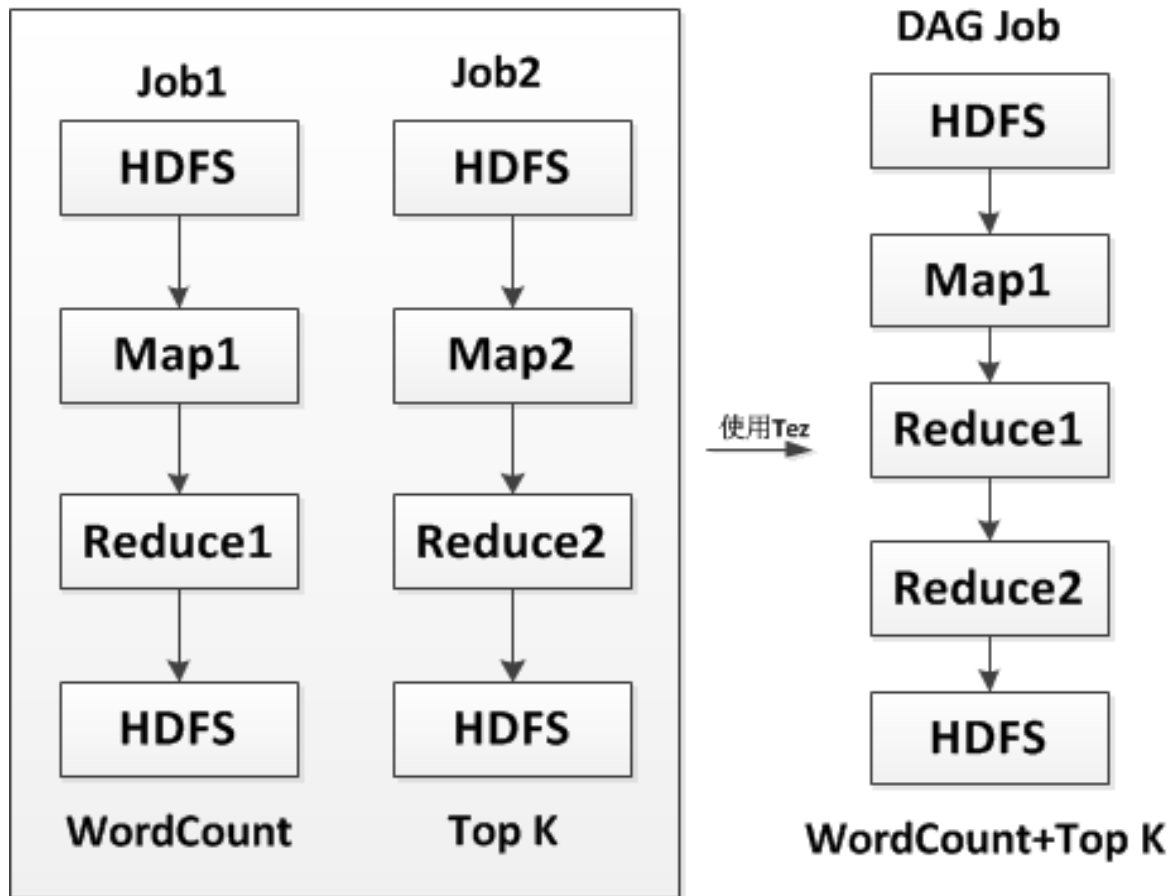
```
select brand,sum(price) as totalPrice  
from record join brand_dimension on record.bid=brand_dimension.bid  
group by brand_dimension.brand  
order by totalPrice desc
```

Tez

- 是Apache最新开源的支持DAG作业(有向无环图)计算框架
- 直接源于MapReduce框架, 由Hadoop2开发团队打造。
- 运行在HADOOP2 YARN平台之上。
- tez与hive结合后, 提高45倍。

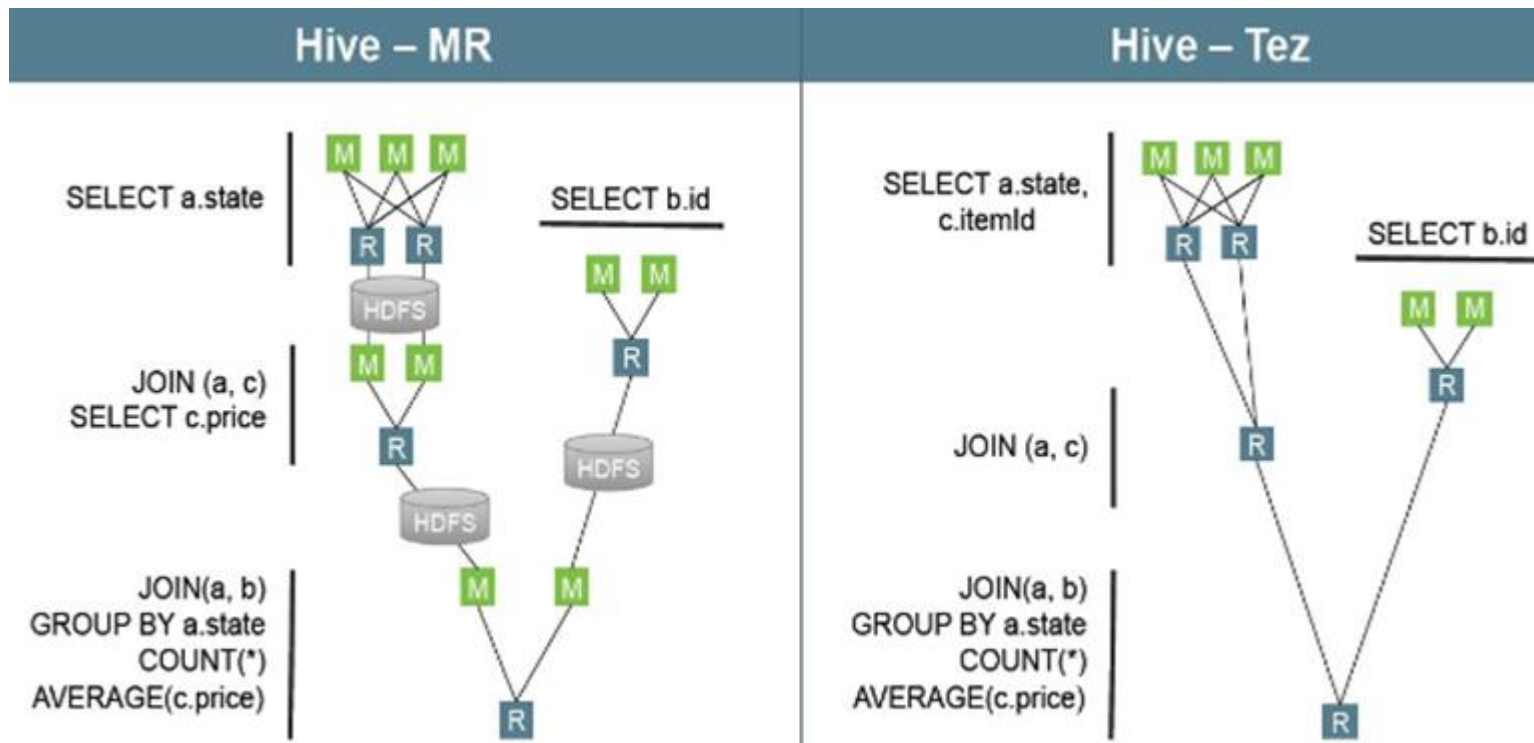
TEZ的典型使用场景—MR场景

分别使用
MapReduce和
Tez统计最热门
的k个查询词，
即Top k问题。



TEZ的典型使用场景—HIVE场景

Hive SQL会翻译成四个MR作业，Tez生成一个DAG作业，大大减少磁盘IO：



```
SELECT a.state, COUNT(*), AVERAGE(c.price) FROM a JOIN b ON(a.id = b.id)  
JOIN c ON(a.itemId = c.itemId) GROUP BY a.state
```

R语言

- R是用于统计分析、统计绘图的语言和操作环境。R提供一些集成的统计工具，但更大量的是它提供各种数学计算、统计计算的函数，从而使使用者能灵活机动的进行数据分析，甚至创造出符合需要的新的统计计算方法。
- R语言之父：Ross Ihaka



R+Hadoop

- Hadoop家族的强大之处在于对大数据的处理，让原来的不可能（TB, PB数据量计算），成为了可能。
- R语言的强大之处在于统计分析，在没有Hadoop之前，对于大数据的处理，要取样本，假设检验，做回归，长久以来R语言都是统计学家专属的工具。
- hadoop重点是全量数据分析，而R语言重点是样本数据分析。两种技术放在一起，刚好是最长补短！

Rhadoop计算场景

- 对1PB的新闻网站访问日志做分析，预测未来流量变化
- 用R语言，通过分析少量数据，对业务目标建回归建模，并定义指标
- 用Hadoop从海量日志数据中，提取指标数据
- 用R语言模型，对指标数据进行测试和调优
- 用Hadoop分步式算法，重写R语言的模型，部署上线
- 这个场景中，R和Hadoop分别都起着非常重要的作用。以计算机开发人员的思路，所有事情都用Hadoop去做，没有数据建模和证明，“预测的结果”一定是有问题的。以统计人员的思路，所有的事情都用R去做，以抽样方式，得到的“预测的结果”也一定是有问题的。所以让二者结合，是产界业的必然的导向，也是产界业和学术界的交集，同时也为交叉学科的人才提供了无限广阔的想象空间。

Rhadoop计算场景

- RHadoop是RevolutionAnalytics的工程的项目，开源实现代码在GitHub社区可以找到。
- RHadoop包含三个R包（rmr，rhdfs，rhbase），分别是对应Hadoop系统架构中的MapReduce，HDFS，HBase三部分。

Rhadoop安装

- 操作系统： centos7.2
- Hadoop环境使用的是2.7.3
- Java使用的是1.8

Rhadoop安装

- 必要的包
- `install.packages("rJava")`
- `install.packages("reshape2")`
- `install.packages("Rcpp")`
- `install.packages("iterators")`
- `install.packages("itertools")`
- `install.packages("digest")`
- `install.packages("RJSONIO")`
- `install.packages("functional")`
- `install.packages("caTools")`

Rhadoop安装

- 环境变量的设置
- HADOOP_CMD环境变量的设置(易错):
- 把hadoop的bin下的hadoop赋给HADOOP_CMD
- HADOOP_CMD=/opt/hadoop-2.7.3/bin/hadoop
- HADOOP_STREAMING环境变量的设置 (rmr需要):
- export HADOOP_STREAMING=/opt/hadoop-2.7.3/share/hadoop/tools/lib/hadoop-streaming-2.7.3.jar

Rhadoop安装

- rhdfs的安装(只需要在user-client上安装即可)
- R CMD INSTALL rhdfs_1.0.8.tar.gz
- rmr2的安装(每个节点都需要安装)
- R CMD INSTALL rmr2_3.3.1.tar.gz

Rhadoop安装测试

Rhdfs

```
>library("rhdfs")
```

```
> hdfs.init()
```

```
16/08/01 15:55:35 WARN util.NativeCodeLoader: Unable to load native-hadoop library for  
your platform... using builtin-java classes where applicable
```

```
> hdfs.ls("/")
```

	permission	owner	group	size	modtime	file
1	drwxr-xr-x	root	supergroup	0	2016-07-31 14:53	/library
2	drwxr-xr-x	Administrator	supergroup	0	2016-07-31 16:37	/user

Rhadoop各个包在集群中的安装情况

Package	Where to Install
plyrmr	On every node in the cluster
ravro	Only on the node that runs the R client
rhbase	Only on the node that runs the R client
rhdfs	Only on the node that runs the R client
rmr2	On every node in the cluster

实例

- 查看hdfs文件目录
- `hadoop fs -ls /user` `hadoop`
- `hdfs.ls("/user/")` `R`
- 查看hadoop数据文件
- `hadoop fs -cat /user/hdfs/o_same_school/part-m-00000`
- `hdfs.cat("/user/hdfs/o_same_school/part-m-00000")`

rmr2包启动

```
> library(rmr2)
```

```
Loading required package: Rcpp
```

```
Loading required package: RJSONIO
```

```
Loading required package: digest
```

```
Loading required package: functional
```

```
Loading required package: stringr
```

```
Loading required package: plyr
```

```
Loading required package: reshape2
```

注： rhdfs和rmr2包之间没有依赖关系

rmr实现MapReduce算法

MapReduce算法

普通的R语言程序：

```
> small.ints = 1:10  
> sapply(small.ints, function(x) x^2)
```

结果：

```
[1] 1 4 9 16 25 36 49 64 81 100
```

注：因为MapReduce只能访问HDFS文件系统，先用to.dfs()

把数据存储到HDFS文件系统里。

MapReduce的运算结果再用

from.dfs()函数从HDFS文件系统中取出。

MapReduce的R语言程序：

```
> small.ints = to.dfs(1:10)  
> mapreduce(input = small.ints, map = function(k, v) cbind(v, v^2))  
> from.dfs("/tmp/RtmpWnzxl4/file5deb791fcbd5")
```

结果：

```
$key  
NULL  
  
$val v  
[1,] 1 1  
[2,] 2 4  
[3,] 3 9  
  
[4,] 4 16  
[5,] 5 25  
[6,] 6 36  
[7,] 7 49  
[8,] 8 64  
[9,] 9 81  
[10,] 10 100
```

rmr对文件中的单词计数

```
> input <- '/user/hdfs/o_same_school/part-m-00000'
> wordcount = function(input, output = NULL, pattern = " "){
  wc.map = function(., lines) {
    keyval(unlist( strsplit( x = lines, split = pattern)), 1)
  }
  wc.reduce = function(word, counts) {
    keyval(word, sum(counts))
  }
  mapreduce(input = input, output = output, input.format = "text",
            map = wc.map, reduce = wc.reduce, combine = T)
}
> wordcount(input)
```


rhbase安装

- 配置HBase, Thrift
- 启动Hadoop, Hbase, Thrift Server
- 安装rhbase
- ~ R CMD INSTALL /root/R/rhbase_1.1.1.tar.gz

Rhbase函数

- hb.compact.table
- hb.describe.table
- hb.insert
- hb.regions.table
- hb.defaults
- hb.get
- hb.insert.data.frame
- hb.scan
- hb.delete
- hb.delete
- hb.get.data.frame
- hb.list.tables
- hb.scan.ex
- hb.delete.table
- hb.init
- hb.new.table
- hb.set.table.mode

Rhbase操作

- 建表
- HBASE:create 'student_shell','info'
- RHBASE: hb.new.table("student_rhbase","info")
- 列出所有表
- HBASE:list
- RHBASE: hb.list.tables()
- 显示表结构
- HBASE:describe 'student_shell'
- RHBASE: hb.describe.table("student_rhbase")
- 插入一条数据
- HBASE:put 'student_shell','mary','info:age','19'
- RHBASE: hb.insert("student_rhbase",list(list("mary","info:age", "24")))
- 读取数据
- HBASE:get 'student_shell','mary'
- RHBASE: hb.get('student_rhbase','mary')
- 删除表(HBASE需要两条命令, rhbase仅是一个操作)
- HBASE:disable 'student_shell'
- HBASE:drop 'student_shell'
- RHBASE: hb.delete.table('student_rhbase')

谢谢！

