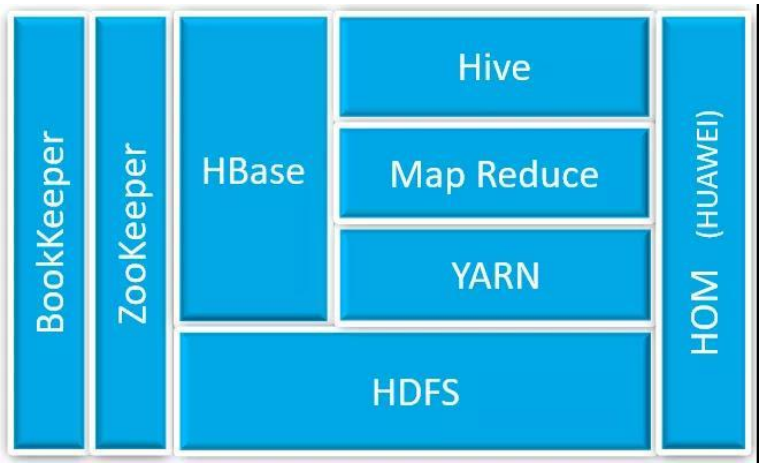
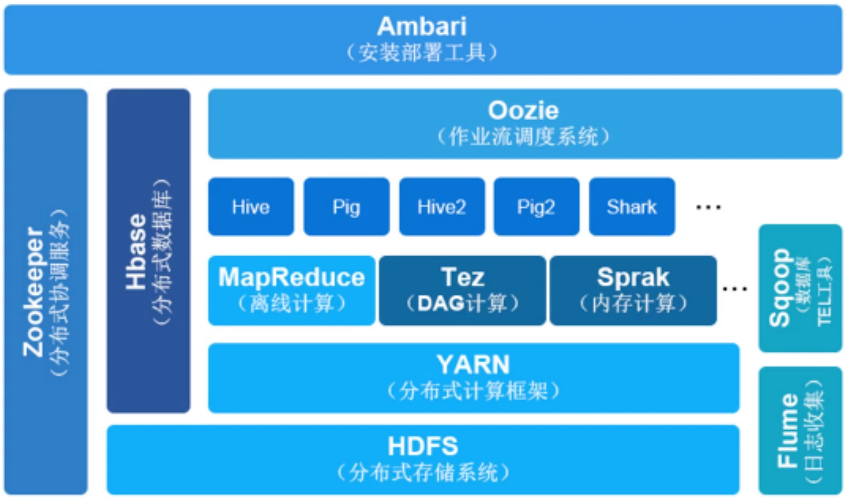
Hadoop 生态系统

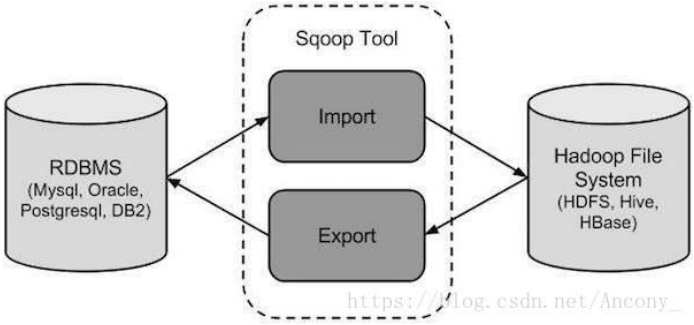

BookKeeper
ZooKeeper
HBase
Hive
Map Reduce
YARN
HDFS
HOM (HUAWEI)


Ambari
（安装部署工具）
Zookeeper
（分布式协调服务）
Hbase
（分布式数据库）
Oozie
（作业流调度系统）
Hive
Pig
Hive2
Pig2
Shark
…
MapReduce
（离线计算）
Tez
（DAG计算）
Sprak
（内存计算）
…
YARN
（分布式计算框架）
HDFS
（分布式存储系统）
Sqoop
（数据库
TEL工具）
Flume
（日志收集）

组件	功能
HDFS	分布式文件系统
MapReduce	分布式并行编程模型
YARN	资源管理和调度器
Tez	运行在YARN之上的下一代Hadoop查询处理框架
Hive	Hadoop上的数据仓库
HBase	Hadoop上的非关系型的分布式数据库
Pig	一个基于Hadoop的大规模数据分析平台，提供类似SQL的查询语言Pig Latin
Sqoop	用于在Hadoop与传统数据库之间进行数据传递
Oozie	Hadoop上的工作流管理系统
Zookeeper	提供分布式协调一致性服务
Storm	流计算框架
Flume	一个高可用的，高可靠的，分布式的海量日志采集、聚合和传输的系统
Ambari	Hadoop快速部署工具，支持Apache Hadoop集群的供应、管理和监控
Kafka	一种高吞吐量的分布式发布订阅消息系统，可以处理消费者规模的网站中的所有动作流数据
Spark	类似于Hadoop MapReduce的通用并行框架

组件	功能	解决Hadoop中存在的问题
Pig	处理大规模数据的脚本语言，用户只需要编写几条简单的语句，系统会自动转换为MapReduce作业	抽象层次低，需要手工编写大量代码
Spark	基于内存的分布式并行编程框架，具有较高的实时性，并且较好支持迭代计算	延迟高，而且不适合执行迭代计算
Oozie	工作流和协作服务引擎，协调Hadoop上运行的不同任务	没有提供作业（Job）之间依赖关系管理机制，需要用户自己处理作业之间依赖关系
Tez	支持DAG作业的计算框架，对作业的操作进行重新分解和组合，形成一个大的DAG作业，减少不必要操作	不同的MapReduce任务之间存在重复操作，降低了效率
Kafka	分布式发布订阅消息系统，一般作为企业大数据分析平台的数据交换枢纽，不同类型的分布式系统可以统一接入到Kafka，实现和Hadoop各个组件之间的不同类型数据的实时高效交换	Hadoop生态系统中各个组件和其他产品之间缺乏统一的、高效的数据交换中介

pig：提供 SQL Like 语法 Pig Latin，最终转化成 MapReduce 作业
Tez：优化
sqoop：用于传统关系型数据库与 Hadoop 文件系统之间的文件传输



分布式文件系统：分布式系统底层物理存储结构

HDFS：Hadoop Distribute File System

GFS：Google File System

底层存储结构：

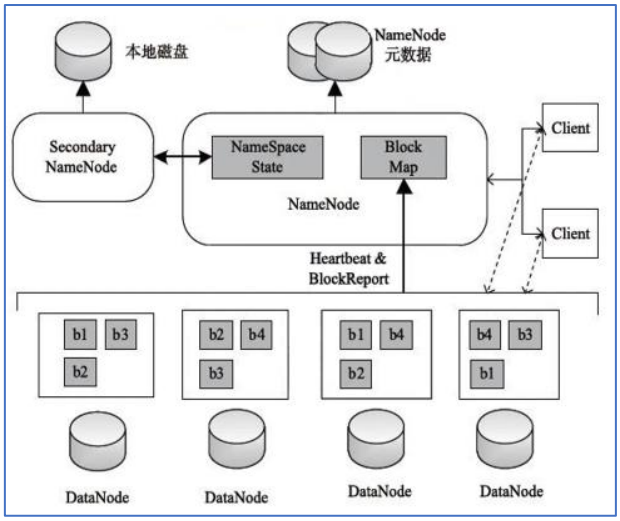
物理存储结构：HDFS

逻辑存储结构：HBase

HDFS 结构：

一个主节点：Master；多个从节点 slave

Secondary NameNode 主节点的备份



NoSQL：Not only SQL

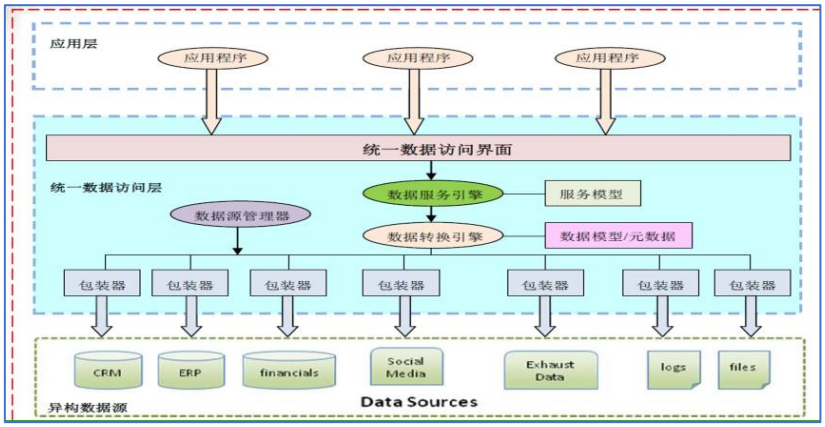
NoSQL数据库的4种类型

- 键值数据库 (key-value store database)
- 列存储数据库 (column family-oriented database)
- 文档数据库 (document-oriented database)
- 图形数据库 (graph-oriented database)

键值数据库：Redis、Memcached、SimpleDB

列族数据库：HBase
文档数据库（类似键值数据库）：MongoDB
图数据库：Neo4j

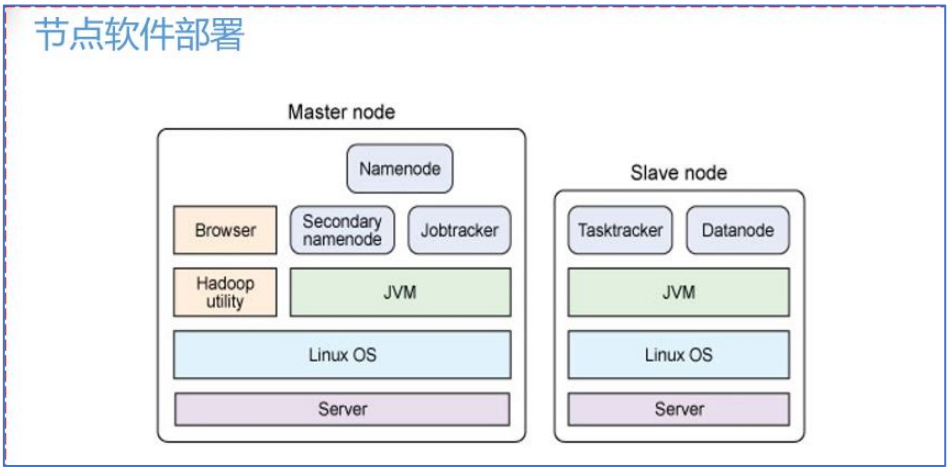
统一数据读写接口（DAL）：
ODBC：C++
JDBC：java 语言编写



主要计算模式：
离线批处理模式：MapReduce
图计算模式：BSP
流计算模式：流计算模型
内存计算模式：spark
大规模并行处理模式：NUMA

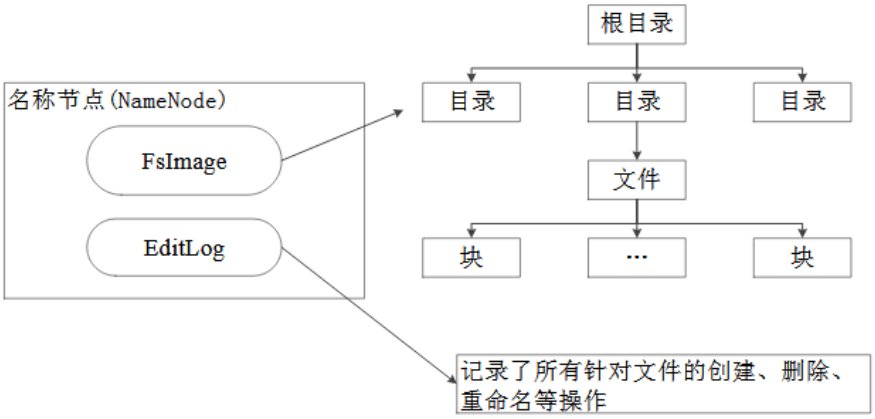
HDFS 分布式文件系统

主节点上运行程序：namenode、Jobtracker
从节点上运行程序：datanode、Tasktracker
全部运行在 Linux 操作系统和 JVM 之上

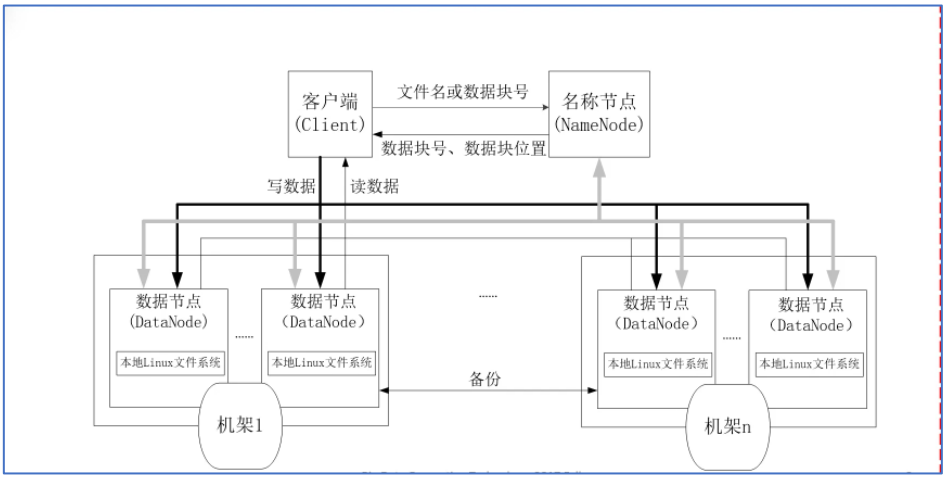


NameNode（名称节点）负责管理分布式文件系统的命名空间，包括两种核心数据结构，FsImage 和 EditLog

名称节点记录了每个文件中各个块所在的数据节点的位置信息

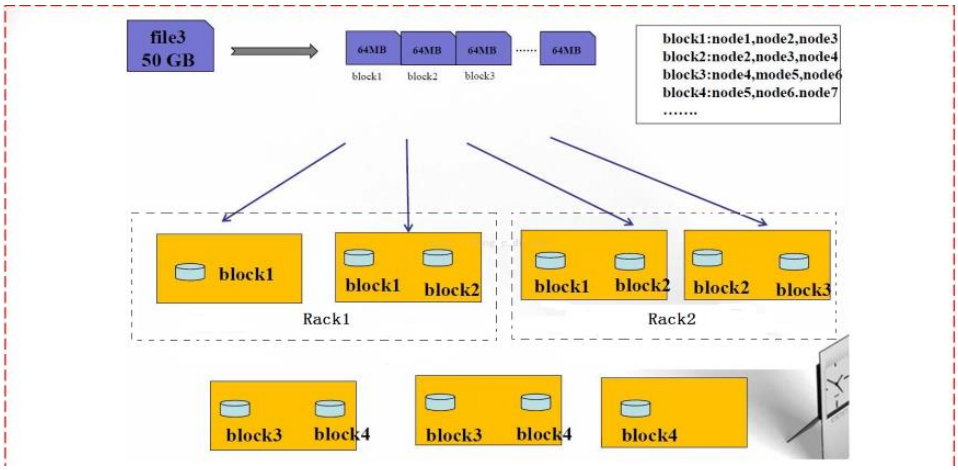


通过与 NameNode 交互获得数据位置，然后 Client 直接与 DataNode 交互，进行数据读写



备份（容错和恢复），存储在不同的 rack 和 DataNode 上

block size=64Mb



心跳包检测

NameNode 主节点与备份节点，当一段时间接受不到，备份节点自动取代主节点

NameNode 与 DataNode 之间的心跳检测

HDFS 文件读写机制：

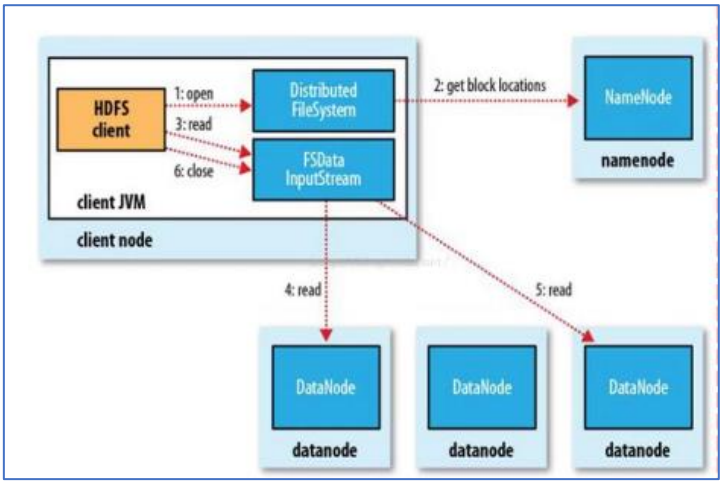
HDFS Shell 命令

HDFS Java API

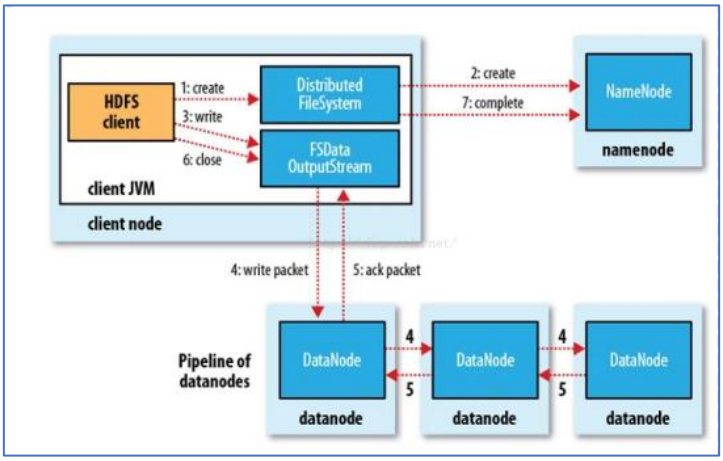
Hadoop 中有三种 Shell 命令方式：

- hadoop fs 适用于任何不同的文件系统，比如本地文件系统和 HDFS 文件系统
- hadoop dfs 只能适用于 HDFS 文件系统
- hdfs dfs 跟 hadoop dfs 的命令作用一样，也只能适用于 HDFS 文件系统

读取数据流程

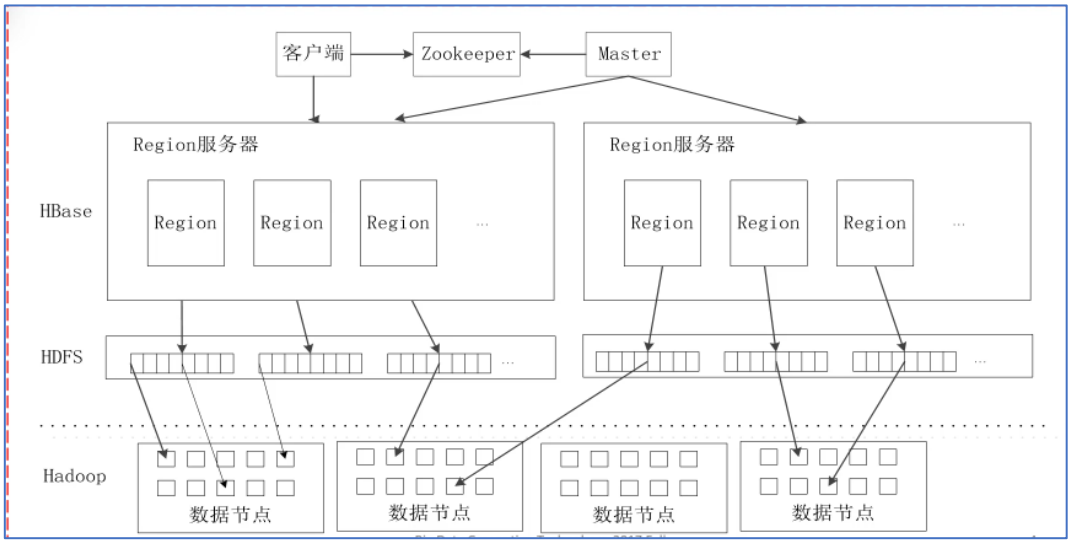


写入数据流程



HBase：Region

HBase 构建在底层 HDFS 分布式文件系统之上



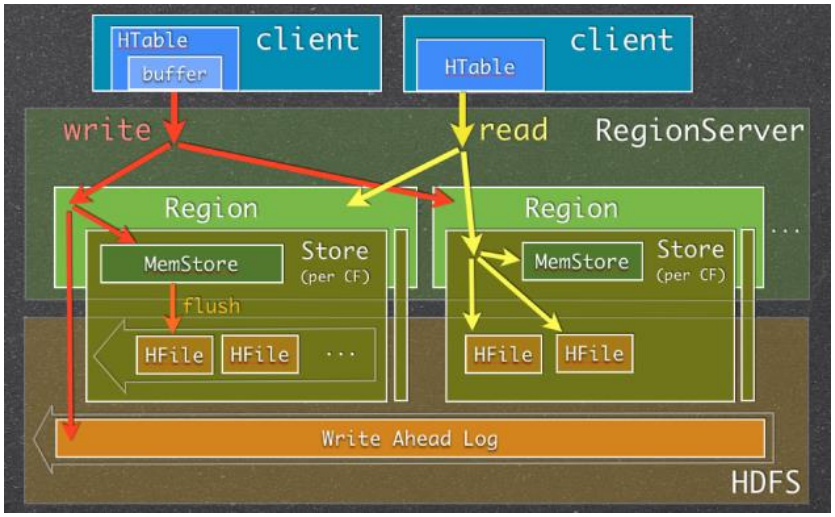
Region 由数据表按行拆分，Store 由 Region 按列拆分

memstore:内存缓冲区。数据首先放入MemStore,等MemStore满了之后再放入StoreFile

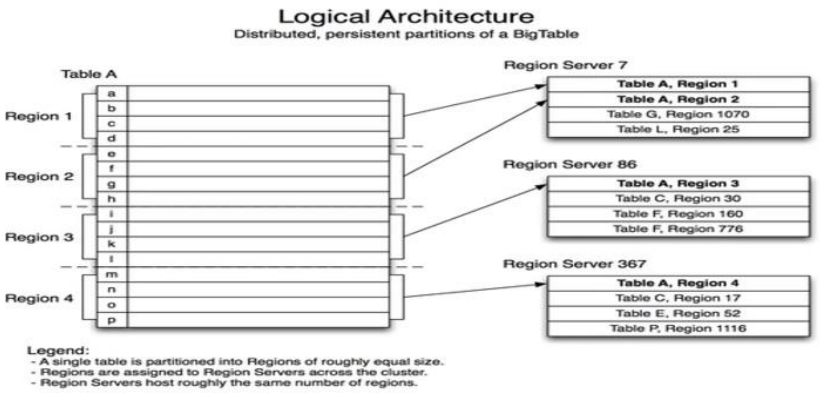
$$Region = HLog + m*Store$$

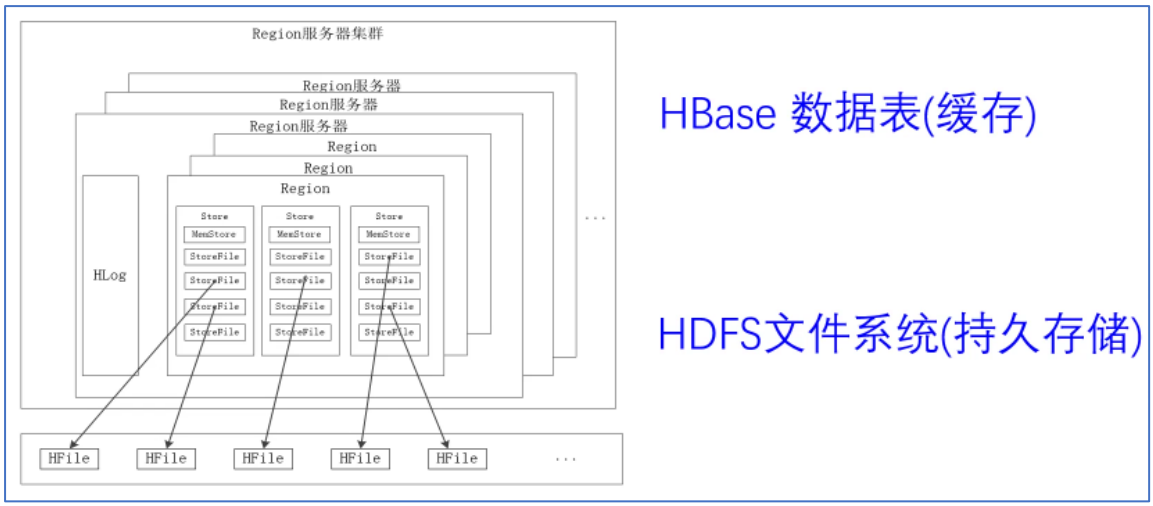
$$Store = MemStore + n*StoreFile$$

当 RS 处理写请求的时候，数据首先写入到 Memstore，然后当到达一定的阀值的时候，Memstore 中的数据会被刷到 HFile 中。



数据表的Region划分

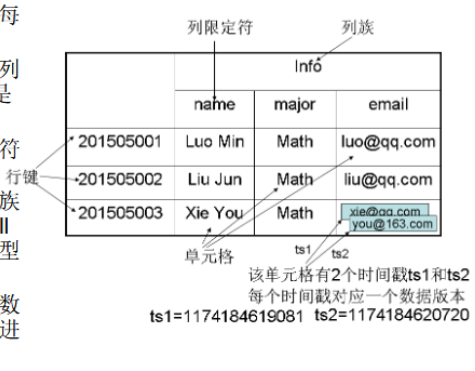




HBase 中执行更新操作时，并不会删除数据旧的版本，而是生成一个新的版本，旧有的版本仍然保留。

HBase 使用**四维坐标**来表示一个单元格：**[行键，列族，列限定符，时间戳]**

- 表：HBase采用表来组织数据，表由行和列组成，列划分为若干个列族
- 行：每个HBase表都由若干行组成，每个行由行键（row key）来标识。
- 列族：一个HBase表被分组成许多“列族”（Column Family）的集合，它是基本的访问控制单元
- 列限定符：列族里的数据通过列限定符（或列）来定位
- 单元格：在HBase表中，通过行、列族和列限定符确定一个“单元格”（cell），单元格中存储的数据没有数据类型，总被视为字节数组byte[]
- 时间戳：每个单元格都保存着同一份数据的多个版本，这些版本采用时间戳进行索引



Hbase 的概念视图——逻辑视图（用户开发面向的视图）：一个行键对应一行，有多个时间戳，对于每个时间戳，对应列族和列（稀疏）

表4-4 HBase数据的概念视图

行键	时间戳	列族contents	列族anchor
"com.cnn.www"	t5		anchor:cnnsi.com="CNN"
	t4		anchor:my.look.ca="CNN.com"
	t3	contents:html="<html>..."	
	t2	contents:html="<html>..."	
	t1	contents:html="<html>..."	

Hbase 物理视图：**按列存储**

表4-5 HBase数据的物理视图

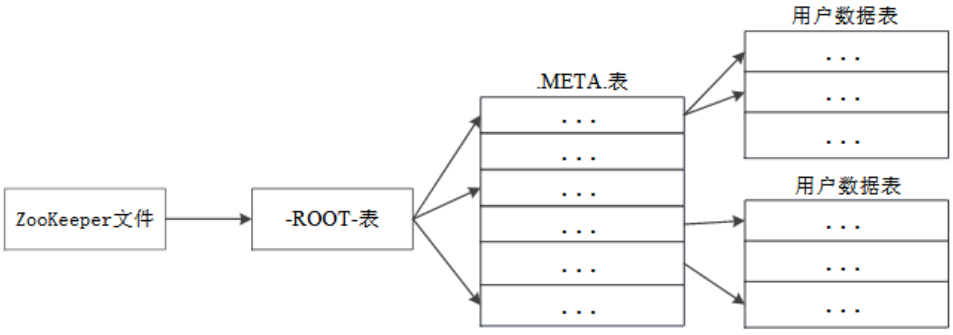
列族contents

行键	时间戳	列族contents
"com.cnn.www"	t3	contents:html="<html>..."
	t2	contents:html="<html>..."
	t1	contents:html="<html>..."

列族anchor

行键	时间戳	列族anchor
"com.cnn.www"	t5	anchor:cnnsi.com="CNN"
	t4	anchor:my.look.ca="CNN.com"

HBase 索引三层结构（如何通过 Region Id 查找到 Region 所在的 Region 服务器）：zookeeper 文件中保存-Root-表所在的路径，-Root-表索引.Meta 表，.Meta 表索引 Region id 与 Region 服务器对应的映射关系。（.Meta 表可能有很大，也会进行 Region 分表，所以需要-Root-表）



HBase 性能监视（查看 HBase 资源利用率等）：Master-status，Ganglia，OpenTSDB，Ambari 工具

HBase 访问接口：**除了利用最基本的 Java API 和 Shell 命令访问外，可以在 HBase 上层构建 Hive，利用类似 SQL 语言方式访问 HBase**（Phoenix 也可实现 Hive 类似的功能，在 Apache HBase 之上构建一个 SQL 中中间层，让开发者在 HBase 上执行 SQL 查询）

类型	特点	场合
Native Java API	最常规和高效的访问方式	适合Hadoop MapReduce作业并行批处理HBase表数据
HBase Shell	HBase的命令行工具，最简单的接口	适合HBase管理使用
Thrift Gateway	利用Thrift序列化技术，支持C++、PHP、Python等多种语言	适合其他异构系统在线访问HBase表数据
REST Gateway	解除了语言限制	支持REST风格的Http API访问HBase
Pig	使用Pig Latin流式编程语言来处理HBase中的数据	适合做数据统计
Hive	简单	当需要以类似SQL语言方式来访问HBase的时候

HBase Shell 常见命令：（四维坐标）

```
create tableName colFamily
put tableName rowKey, ‘colFamily’ ,value
get tableName,rowKey,{COLUMN=>’ colFamliy:col’ }

disable tableName
drop tableName
```

HMaster 宕机的时候,哪些操作还能正常工作

对表内数据的增删查改是可以正常进行的,因为 hbase client 访问数据只需要通过 **zookeeper 来找到 rowkey 的具体 region 位置**即可. 但是对于**创建表/删除表等（DDL）**的操作就无法进行了,因为这时候是需要 HMaster 介入, 并且 region 的拆分,合并,迁移等操作也都无法进行了

YARN

MapReduce1.0：既是计算框架，也是一个资源管理调度框架

MapReduce2.0：资源管理调度功能分离出来，形成 YARN；MapReduce2.0 是运行在 YARN 之上的一个纯粹的计算框架，不在自己负责资源调度管理服务，而是有 YARN 为其提供资源管理

调度服务。

YARN 以 Container（容器）作为最小单位

ResourceManager：包括 Scheduler（调度器）和 ApplicationManager

ApplicationMaster：

ApplicationMaster 本身也是应用程序，需要有自己独立运行的程序，因此最开始 ResourceMananger 需要分配容器给 ApplicationMaster，

针对不同的计算框架重新写 ApplicationMaster，服务于该计算框架.MapReduce 批处理架构有 ApplicationMaster，Strom 框架也有自己的 ApplicationMaster，因此 YARN 是一种专门的资源管理框架，用于底层，支撑各种计算架构

NodeManager：

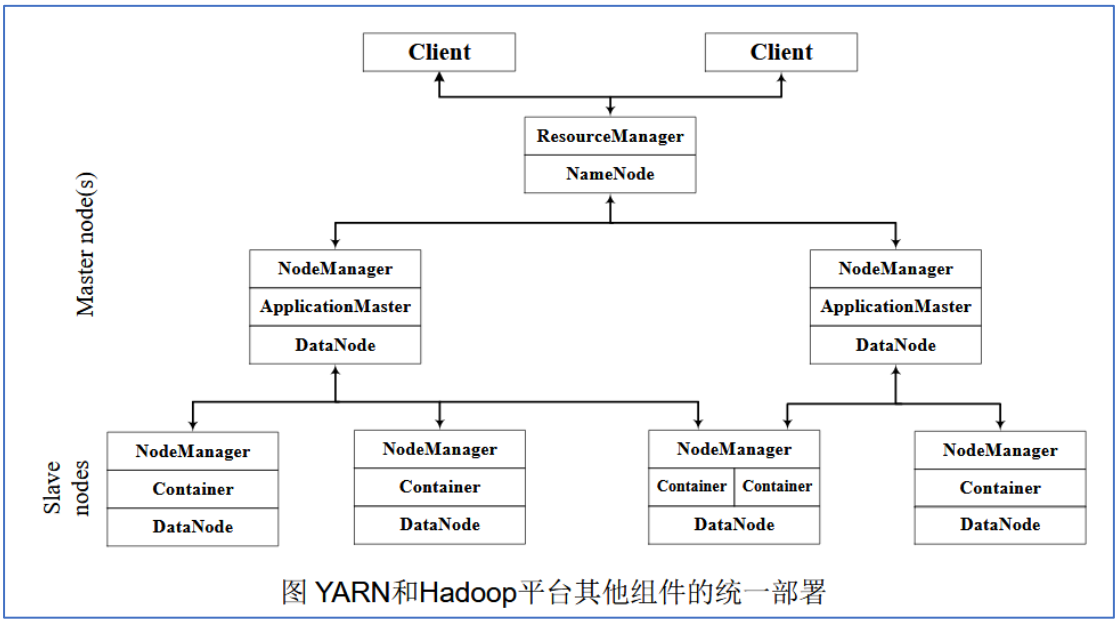
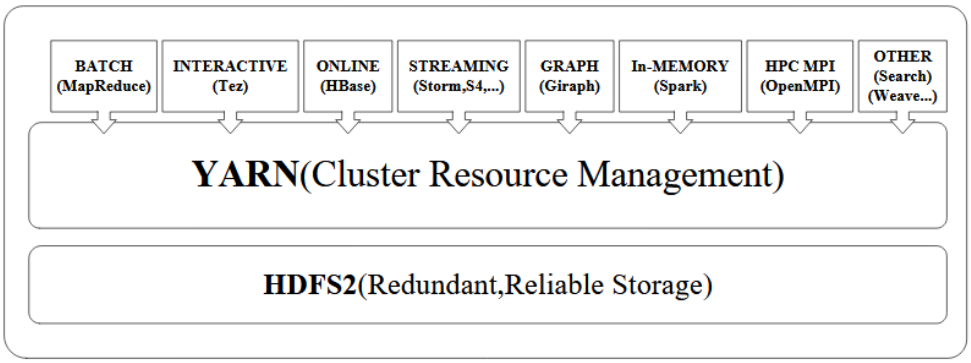


图 YARN和Hadoop平台其他组件的统一部署

NodeManager 主要负责管理抽象的容器，只处理与容器相关的事情，而不具体负责每个任务（Map 任务或 Reduce 任务）自身状态的管理，因为这些管理工作是由 ApplicationMaster 完成的，ApplicationMaster 会通过不断与 NodeManager 通信来掌握各个任务的执行状态

•YARN的目标就是实现“一个集群多个框架”，即在一个集群上部署一个统一的资源调度管理框架YARN，在YARN之上可以部署其他各种计算框架
•由YARN为这些计算框架提供统一的资源调度管理服务，并且能够根据各种计算框架的负载需求，调整各自占用的资源，实现集群资源共享和资源弹性收缩
•可以实现一个集群上的不同应用负载混搭，有效提高了集群的利用率
•不同计算框架可以共享底层存储，避免了数据集跨集群移动



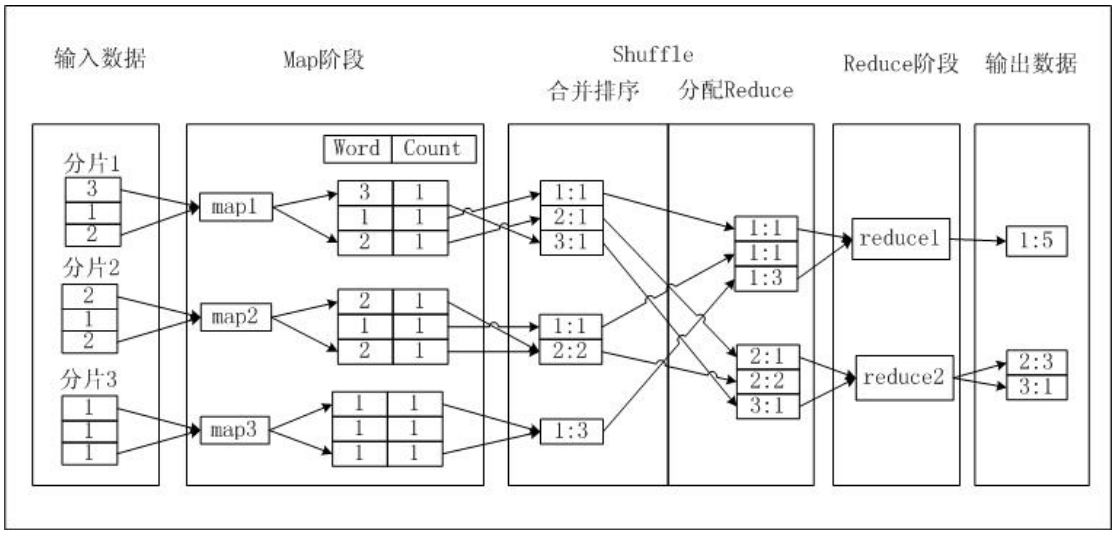
ZooKeeper

作用一：Master 选举可以说是 ZooKeeper 最典型的应用场景了。比如 HDFS 中 Active NameNode 的选举、YARN 中 Active ResourceManager 的选举和 HBase 中 Active HMaster 的选举等。

作用二：HBase 的索引结构存储位置

MapReduce

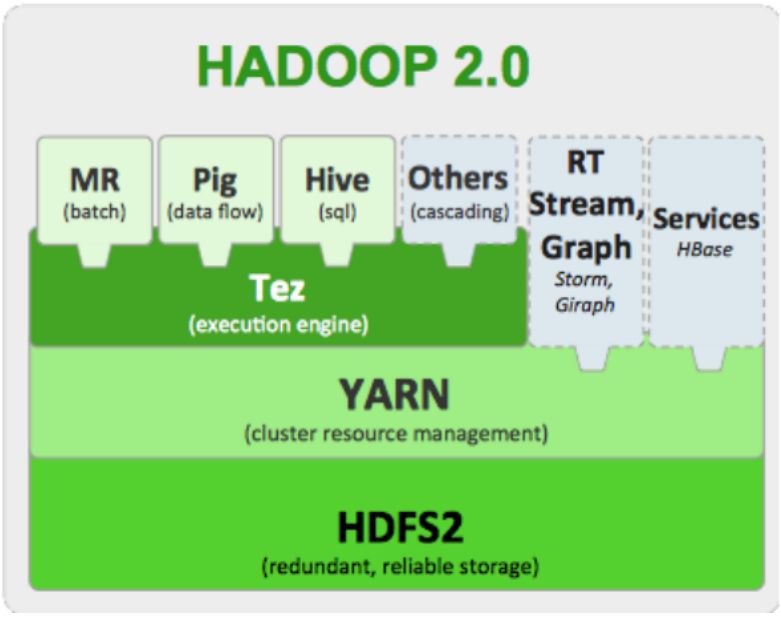
Pig 和 Hive 类似，都是将上层类似 SQL 语法的语句转化成 MapReduce 作业，区别在于：Pig 更多将采集到的数据处理之后存储到数据仓库 Hive，而 Hive 可能更多是对存储好的数据仓库 Hive 进行批处理分析



Tez 用于对 MapReduce 任务进行优化，通常 MapReduce 任务有大量的 MAP 和 Reduce 任务，以及耗时非常大的磁盘读写操作，Tez 用于优化上述过程

MapReduce、Hive、Pig 等计算框架，都需要最终以 MapReduce 任务的形式执行数据分析，因此，Tez 框架可以发挥重要的作用。

Impala 类似 Hive，采用 SQL 语法，底层同样基于 HDFS，不同的是 Impala 不是转化成 MapReduce 任务，而是有它自己的一套机制。速度不 Hive 快 30 倍



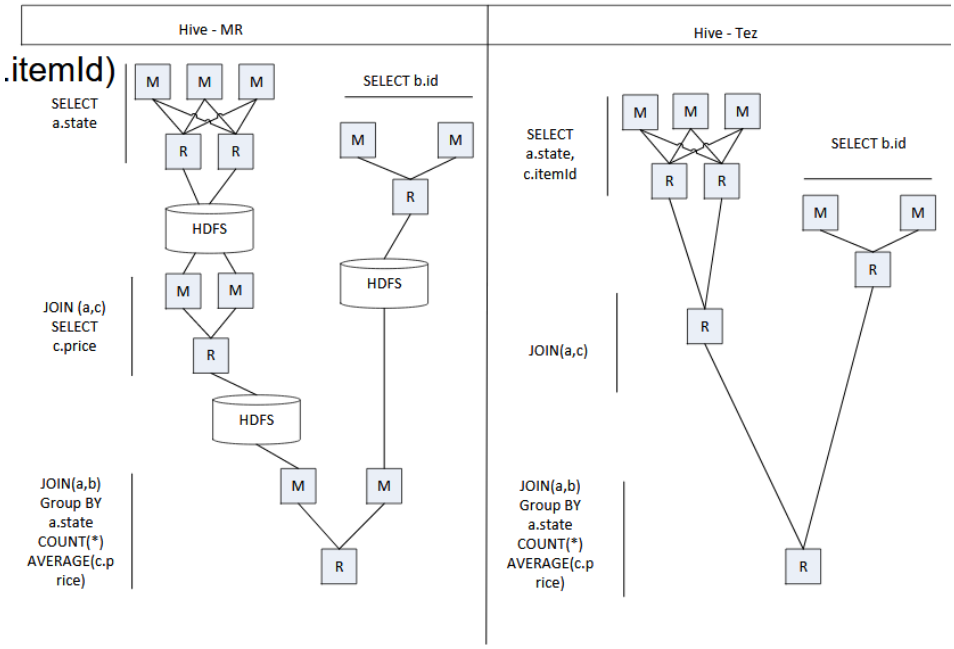
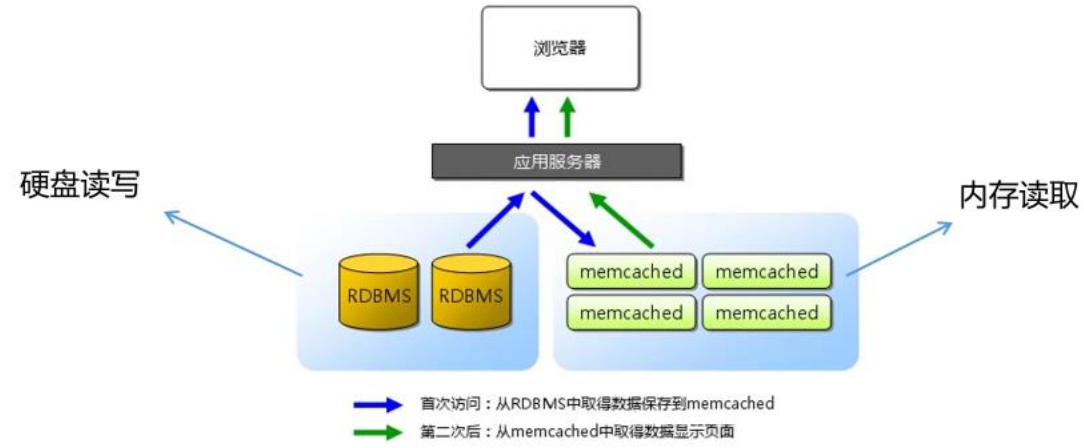


图 HiveQL语句在MapReduce和Tez中的执行情况对比

内存计算模型

Redis、Memcached



Memcached 服务器：

数据同步缓存

数据不同步缓存

数据编码压缩技术

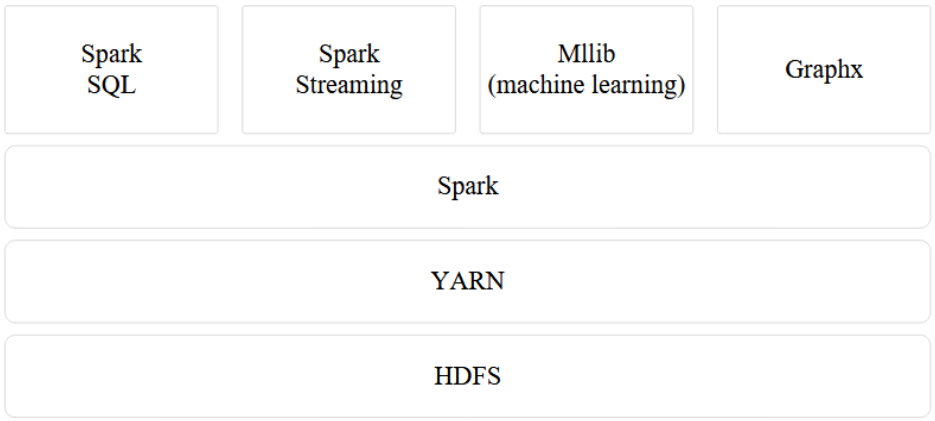
列存储技术

数据表分区技术

Spark

应用场景	时间跨度	其他框架	Spark生态系统中的组件
复杂的批量数据处理	小时级	MapReduce、Hive	Spark
基于历史数据的交互式查询	分钟级、秒级	Impala、Dremel、Drill	Spark SQL
基于实时数据流的数据处理	毫秒、秒级	Storm、S4	Spark Streaming
基于历史数据的数据挖掘	-	Mahout	MLlib
图结构数据的处理	-	Pregel、Hama	GraphX

spark 有自己的生态系统，拥有不同的组件：类似 MapReduce 的 Spark Core，类似 Hive、Impala 的 Spark SQL；类似 Storm 的 Spark Streaming，但是 Spark Streaming 做不好毫秒级；图计算领域，类似 Pregel 的 GraphX；机器学习领域，类似 Mahout 的 MLlib。



Spark 底层是 RDD，上层的 Spark SQL、Spark Streaming 等应用最终应该拆分成 RDD 操作。可以通过 shell、scala 等直接对 RDD 操作，也可以在上层对各个应用进行编程。通常企业实际使用过程中，忽略了底层实现过程，直接对最上层进行操作。

就像 Hadoop 生态系统一样，可以直接利用 Java API 和 Shell 对 HDFS 文件系统操作；也可以利用 Java API 和 Shell 直接对 MapReduce 层进行编程；同时，可以利用 SQL 语言对最上层的 Hive 进行 SQL 编程。

Spark 逻辑模型：RDD

RDD 不可修改，只读，只能转换成一个新的 RDD

RDD 算法：Transformation、Action

Transformation 是 RDD 到 RDD 之间的关系转换，在运算的中间状态；RDD 最终的结果需要通过 Action 操作才能转化成数字。

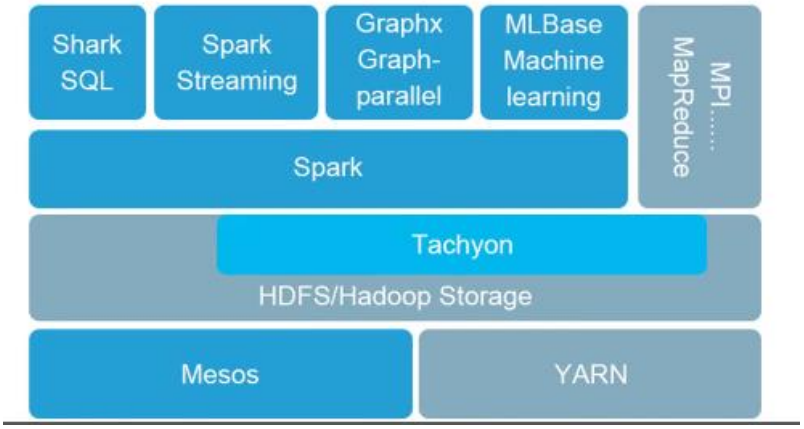
Transformation 常见操作：map、filter，GroupByIndex 等

Action 常见操作：count，collection 等

窄依赖：RDD 一对一的关系（map、filter）

宽依赖：多对一关系（GroupByIndex）

Spark 物理模型：作业模型（Application、Job、Stage、Task）
窄依赖位于同一个 stage，宽依赖位于不同的 stage，不同 stage 之间都是宽依赖关系（shuffle）



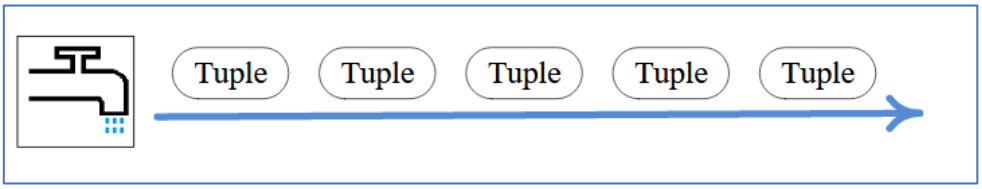
上图可以看出，spark 和 hadoop 最底层硬件存储结构都是 HDFS，spark 的所有应用都是建立在 Spark Core 之上，而 Hive、Pig 等 Hadoop 组件都是建立在 MapReduce 之上。

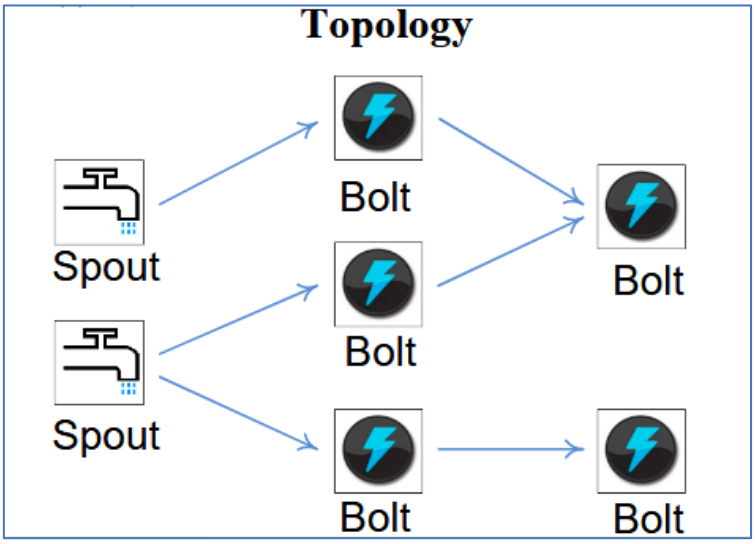
实时流计算

主流产品：

Twitter：storm、yahoo：S4、Apache：spark Streaming、阿里：StreamCompute
Storm 可实现毫秒级别时延，且开源，是目前最普遍的流计算架构

Storm 有以下几个组件：Streams、Spouts、Bolts、Topology 和 StreamGroupings
针对用户（client）来说，主要就是编写 Topology，包括 spout、bolt 的处理流程以及 StreamGrouping 的定义。
spout 和 bolt 的流出叫做 Stream，是一串源源不断的 Tuple，Tuple 可以是任意类型，在编写 spout 或者 bolt 流出是必须定义好 Tuple 每个字段的名称，DeclareOutputFiles 函数中即使定义 Tuple 各个字段名。
实现 bolt 类时，需要重写 execute 和 DeclareOutputFiles 方法






```java
// 对单词进行计数
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);
        collector.emit(new Values(word, count));
    }
    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word", "count"));
    }
}
```

Topology 里面的每个处理组件（Spout 或 Bolt）都包含处理逻辑，而组件之间的连接则表示数据流动的方向。当 Spout 或者 Bolt 发送元组时，它会把元组发送到每个订阅了该 Stream 的 Bolt 上进行处理

Topology 里面的每一个组件都是**并行**运行的，虽然后边的 bolt 需要等前面的 spout 或者 bolt 处理完才能进入下一步，但是就像水流一样，**当前 bolt 处理的 Tuple 是前面处理完的**，所以是并行的，就像流水线一样。

在 Topology 里面可以指定每个组件的并行度，Storm 会在集群里面分配那么多的线程来同时计算。

(1)worker:每个 worker 进程都属于一个特定的 Topology，每个 Supervisor 节点的 worker 可以有多个，每个 worker 对 Topology 中的每个组件（Spout 或 Bolt）运行一个或者多个 executor 线程来提供 task 的运行服务

(2)executor：executor 是产生于 worker 进程内部的线程，会执行同一个组件的一个或者多个 task。

(3)task:**实际的数据处理由 task 完成**，在 Topology 的生命周期中，每个组件的 task 数目是不会发生变化的，而 executor 的数目却不一定。executor 数目小于等于 task 的数目，默认情况下，二者是相等的。**下图中 RandomSentenceSpout 中的 5 表示并行的 Task 数**

目为 5.

```
// 创建一个Topology builder
TopologyBuilder builder = new TopologyBuilder();
// 使用builder.setSpout()方法对Spout数据源进行定义
builder.setSpout("sentences", new RandomSentenceSpout(), 5);
// 使用builder.setBolt()定义Bolt处理任务
builder.setBolt("split", new SplitSentence(), 8)
        .shuffleGrouping("sentences");
// Groupings()系列方法定义了Tuple发送方式
builder.setBolt("count", new WordCount(), 12)
        .fieldsGrouping("split", new Fields("word"));
```

上图中，名称为“split”的 bolt 有 8 个并行 task；名称为“cout”的 bolt 有 12 个并行 task。执行“fileGrouping”的 StreamingGrouping 方式后，根据 Tuple 中名称为“word”的字段进行分配（事实上，经过“split”bolt 后，Tuple 中只有一个字段，就是“word”，每个句子都已经被分成一个一个单词，独立形成 tuple），**相同的名称分配到 12 个“count”task 中同一个 task**。

Stream Groupings 方式

目前，Storm 中的 Stream Groupings 有如下几种方式：

(1)ShuffleGrouping：随机分组，随机分发 Stream 中的 Tuple，保证每个 Bolt 的 Task 接收 Tuple 数量大致一致

(2)FieldsGrouping：按照字段分组，保证相同字段的 Tuple 分配到同一个 Task 中

(3)AllGrouping：广播发送，每一个 Task 都会收到所有的 Tuple

(4)GlobalGrouping：全局分组，所有的 Tuple 都发送到同一个 Task 中

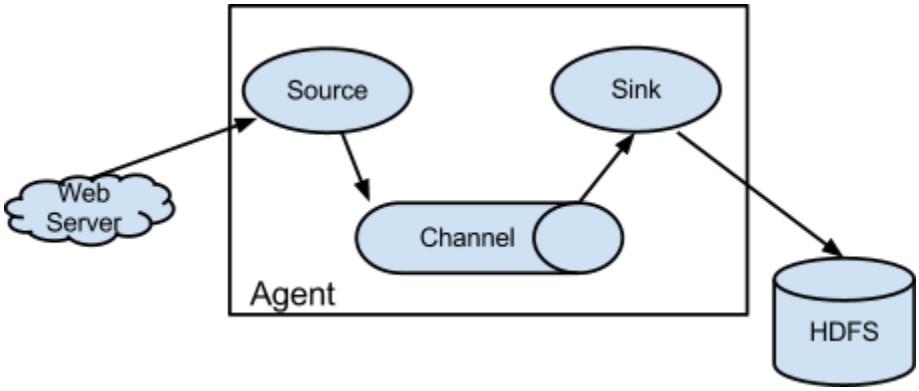
(5)NonGrouping：不分组，和 ShuffleGrouping 类似，当前 Task 的执行会和它的被订阅者在同一个线程中执行

(6)DirectGrouping：直接分组，直接指定由某个 Task 来执行 Tuple 的处理

Flume

Flume 是一种日志收集工具，Flume 由多个 agent 组成，每个 agent 包括 source、channel 和 sink，多个 agent 之间可以存在拓扑关系。

source 源数据通过“put”的 transaction（事务）向 putList 中传入数据，传输失败后数据 rollback，然后传入到 channel 中；channel 通过“take”的 transaction 向 takeList 中传入数据，传输失败数据返回 channel，保证事务的一致性和完整性。



Source

source 组件是专门用来收集数据的，可以处理各种类型、各种格式的日志数据,包括 avro、thrift、exec、jms、spooling directory、netcat、sequence generator、syslog、http、legacy

avro：序列化数据，当**两个 agent 对接**时，即为 avro 类型，需要定义 type、bind（主机名）、port（端口号）

exec：将一个**文件**中的内容远远不断的放入 flume 中，例如 Linux 命令：tail -F

spooling directory：将**文件夹**中的所有文件放到 flume 中

netcat：监听主机端口，将端口的内容源源不断放入到 flume

Channel

source 组件把数据收集来以后，临时存放在 channel 中，即 channel 组件在 agent 中是专门用来存放临时数据的——对采集到的数据进行简单的缓存，可以存放在 **memory、jdbc、file** 等等。

Sink

sink 组件是用于把数据发送到目的地的组件,目的地包括 hdfs、logger、avro、thrift、ipc、file、null、Hbase、solr、自定义。

hdfs：分布式文件系统

avro：两个 agent 对接

logger：console 控制台

File Roll Sink：本地文件系统

kfaka：kfaka 发布订阅系统中

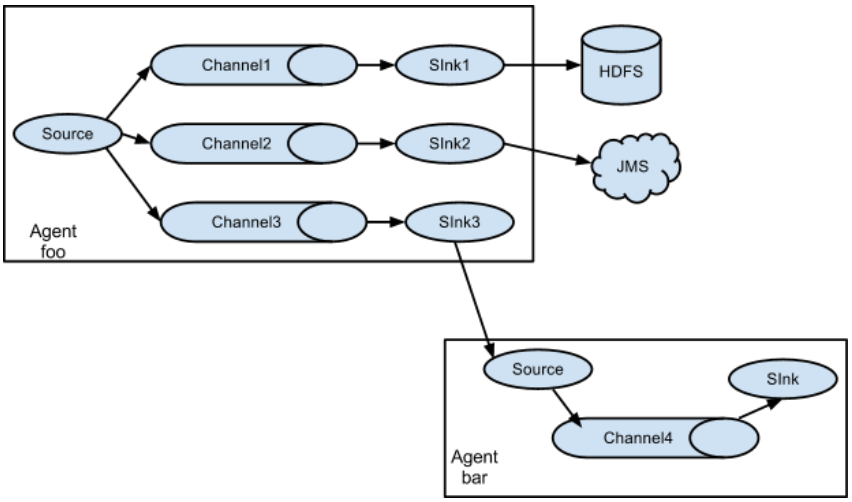
Event

传输单元，Flume 数据传输的基本单元，以事件的形式将数据从源头送至目的地。

几种典型应用场景：

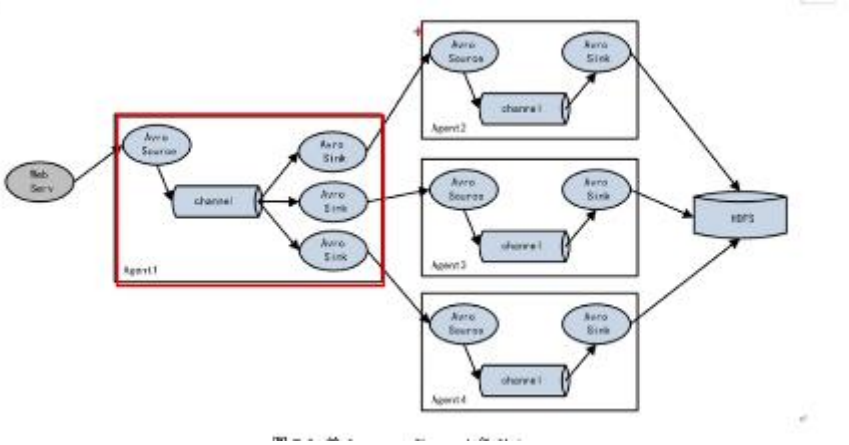
单 Flume 多 Channel、Sink：用于将原始数据传输到各个部门，每个部门看到的数据都是一样的

channel selector=**replicating**，表示将 channel 中的内容复制到所有 sink

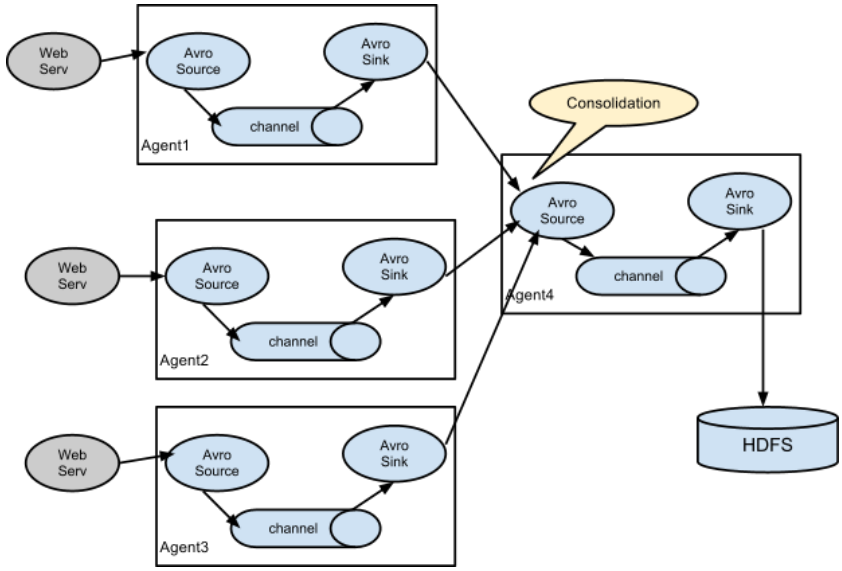


单数据源多出口（sink 组）：此处和上面的差别在于，channel 只有一个，而 sink 有多个。

此种设计存在两个好处：①此处通过设置第一个 agent 的 sink 组可以实现后续的**负载均衡**，Sink groups Processors type 设置为 **load balace**。②每个时间只向一个 agent 分发流量，当一个挂了的时候，另外的 agent 立马补上，保证系统高可用性（HA）。



多 Flume 汇总数据到单 Flume：源数据来源各个地方，最后都汇集到一个地方



Kafka

消息队列（MQ）：解耦、异步、削峰
两种模式：点对点模式、发布订阅模式
主流产品：ActiveMQ,RabbitMQ,RocketMQ,Kafka

kafka：通过 shell、Java 和 Python 都可以操作 Kafka，相对来说，python 可能操作更加便捷
常用架构：**sever→flume→kafka→HDFS/HBase**
flume 用于多数据源的汇集，收集各种日志数据；kafka 用于分发到多个消费者

Avro

数据序列化工具：Avro、protobuf
　　java 本身 jdk 自带序列化工具，但是 java 序列化后的文件只能使用 java 程序打开进行反序列化，同时文件很大。使用 avro 可以跨平台，同时生成的文件很小。

典型流程：
[1] 在 Python 中执行 easy_install avro 安装 avro
[2] 通过 Python 代码将数据序列化
[3] 在 java 中读取数据，进行反序列化

MySQL

MySQL 在 Windows 下数据库名、表名、列名、别名都不区分大小写。
MySQL 在 Linux 下数据库名、表名、列名、别名大小写规则：
　　1、数据库名与表名是严格区分大小写
　　2、表的别名是严格区分大小写
　　3、列名与列的别名在所有的情况下均是忽略大小写的
　　4、变量名也是严格区分大小写的

1.查询——select

```
SELECT [ALL|DISTINCT|DISTINCTROW|TOP]
{*|talbe.*|[table.]field1[AS alias1][,[table.]field2[AS alias2][,…]]}
[case when]
FROM tableName
[WHERE…]
[GROUP BY…]
[HAVING…]
[ORDER BY…]
```

[LIMIT]

顺序：先 where 筛选（行），然后根据 groupby 分组，（此处进行 select 筛选列），再通过 having **筛选分组（组）**，order by 排序，limit 限制输出行数

select 用法

select 后边每个语句都是一个列，该列并不是原有数据表中列，而是**重新生成的列**，对照[case when]可以发现。

使用 groupby 时，除了聚合函数外，select 语句中所有列都必须出现在 groupby 中，以保证每个分组只能有一列

case when 语法

[case when]跟在 select 之后，从原始表格中重新筛选出一个新的列。

典型案例：

数据表格 dataSet 中存在两列：type 和 value。type 存在三种类型，对应 0、1/2，分别表示水费、电费和燃气费。现在需要统计每种类型的电费总计是多少。

type，E_value，Water_value,Heat_value

```
    select
        sum(case when type = 0 then value else 0) as E_value
        sum(case when type = 1 then value else 0) as Water_value
        sum(case when type = 2 then value else 0) as Heat_value
    from dataSet
```

where 用法

where 在自带运算符表达式

类别	运算符	说明
比较运算符	=, <, >, <=, >=, <>	比较两个表达式
逻辑运算符	AND ,OR, NOT	组合两个表达式的运算结果或取反
范围运算符	BETWEEN,NOT BETWEEN	搜索值是否在范围内
列表运算符	IN,NOT IN	查询值是否属于列表值之一
字符匹配符	LIKE ,NOT LIKE	字符串是否匹配
未知值	IS NULL ,IS NOT NULL	查询值是否为 NULL

除此之外，where 还可以匹配**正则表达式**

特殊语法：where id is null

GroupBy 用法

函数名	功能
COUNT	求组中项数
SUM	求和
AVG	求平均值
MAX	求最大值
MIN	求最小值
ABS	求绝对值
ASCII	求 ASCII 码

函数名	功能
RAND	产生随机数

order by 用法

升序：asc

降序：desc

limit 用法

limit 初始位置，查询记录数量（index 从 0 开始）

limit 显示记录数（返回前几条记录）

多表查询——join

笛卡尔积

将两个表的所有行对应起来，保存所有的列。**M 行的 R 表与 N 行的 S 表进行笛卡尔积之后是 M*N 行的表，列数为 R 和 S 的列数之和**。笛卡尔积是下面所有 join 的基础

自然连接——natural join

在笛卡尔积中名称相同的属性，如果他们的内容也是一样就保存这行，同时将相同的两个属性只保留一个。

inner join=join

和 natural join 类似，在笛卡尔积之后，需要指定两列相同，即 p1.c1=p2.c2。通过上述 on 命令筛选行。

left join=left out join

R left join S,在笛卡尔积后，通过 on 命令，对于 R 中的每一行，在 S 中检索满足 on 条件的列。如果 S 中不存在一个，最后的结果也要保存 R 的一行，对应的 S 部分用 null 代替。

on 命令用于构建临时表，在笛卡尔积中筛选满足 on 条件的行，组成临时表。通过 from 联表查询建立的临时表，应该包括笛卡尔积后的所有列，然后通过 select 在临时表中筛选列，通过 where 在临时表中筛选行。

right join=right out join

R right join S，在笛卡尔积后，对于 S 中的每一行，在 R 中检索满足 on 条条件的列。如果 R 中不存在一个，最后的结果也也奥保存 S 中的一行，赌赢的 R 部分用 null 代替。

full join=full out join

在 MySQL 中没有 full join 语法，通过 union 实现

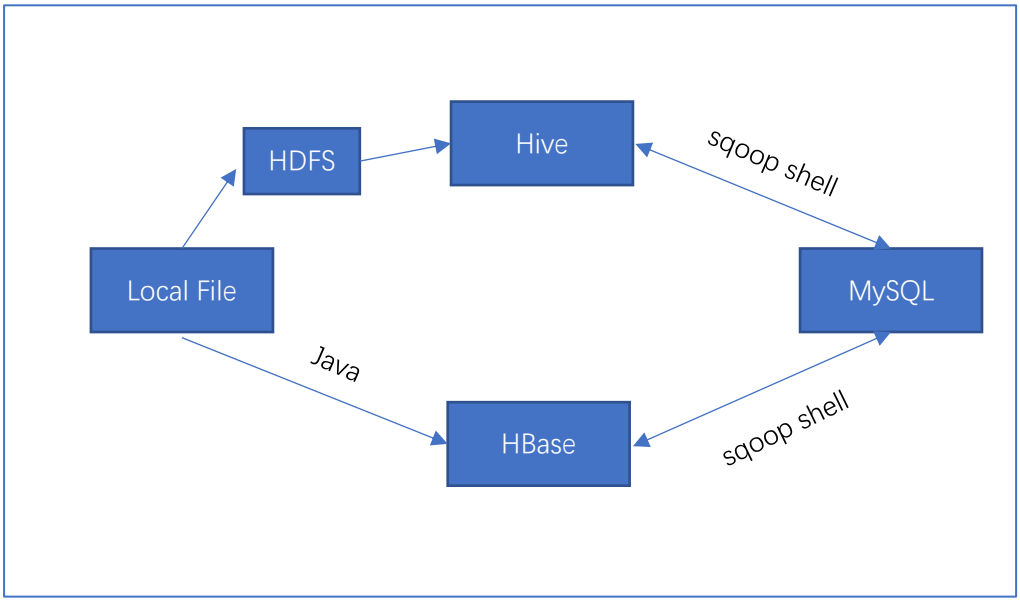
```
select R.c1,R.c2,S.c1,S.c3
from person as R
full join address as S
on R.personId=S.personId

select R.c1,R.c2,S.c1,S.c3
from person as R
left join address as S
on R.personId=S.personId
union
select R.c1,R.c2,S.c1,S.c3
from person as R
```

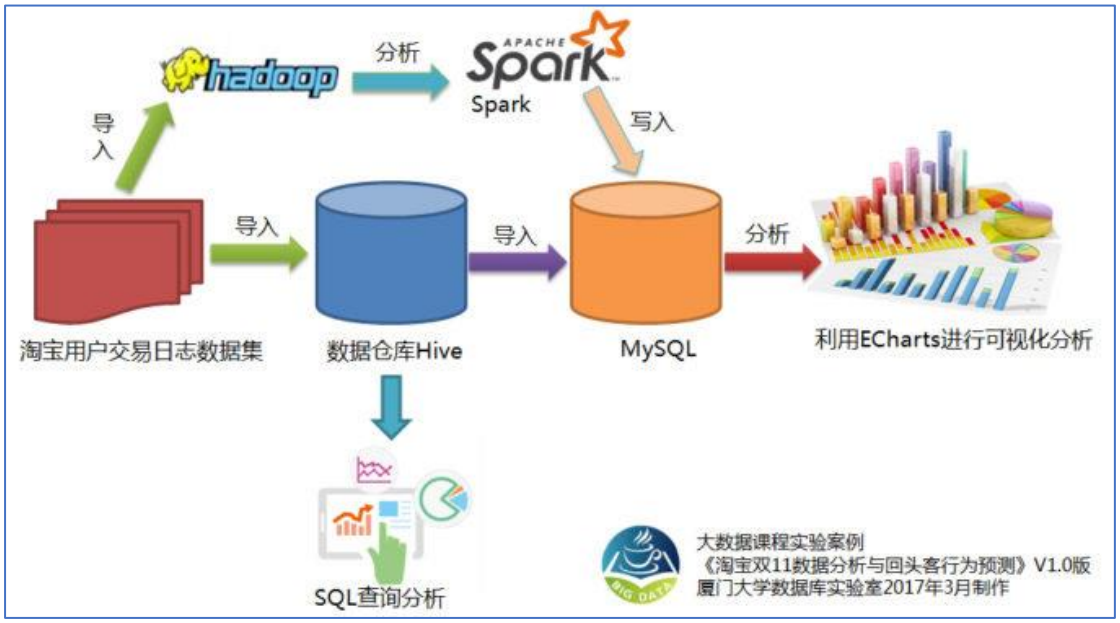
```
right join address as S
on R.personId=S.personId
```

Linux 命令

cd: 进入某个目录
pwd: 当前目录
mkdir: 在当前目录下创建文件夹
head:查看文件前几行
tail: 查看末尾几行
sed 删除文件某行
    sed -i ‘1d’ raw_user:删除文件 raw_user 第一行
ls: 显示文件夹下所有内容
chown: 给文件某权限
unzip: 解压文件
tar: 解压文件
    tar -xzvf filename.tar.gz
zip: 压缩文件
echo: 显示某变量内容
rm: 删除文件或者文件夹
rm: 移动文件或者重命名
vi、vim: 编辑 txt 文件
touch: 新建文件

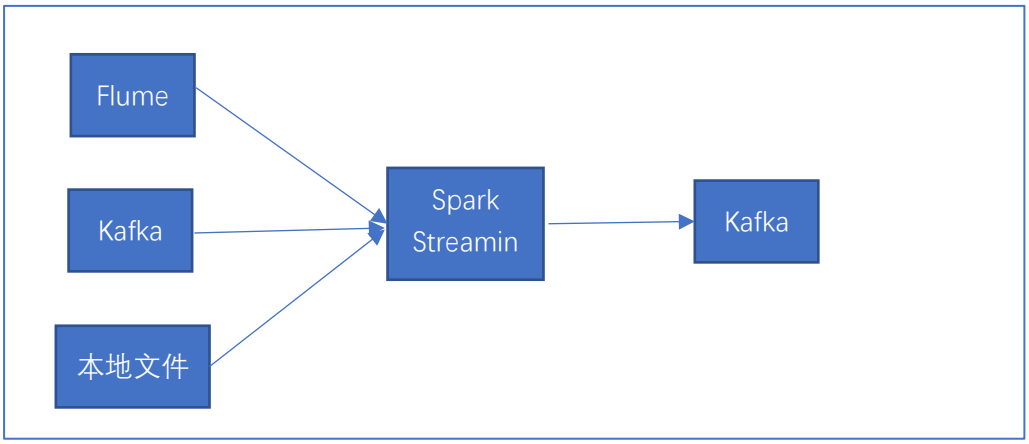
典型案例





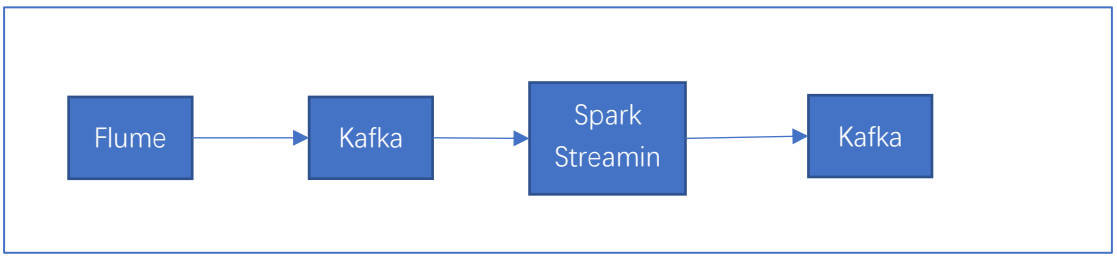
上图中，下路：本地文件上传到 HDFS 中，然后传输到 Hive 中，通过 SQL 语法对 Hive 进行数据分析；通过 sqoop 将数据从 Hive 传输到 MySQL 中。

上路：本地文件首先上传到 HDFS 文件系统中，spark 从 HDFS 文件系统中读取文件，然后通过 SpartML 进行分析，最后把结果存储到 MySQL 中。



spark Streaming 的源头可以来自各个地方，包括本地文件、netcat、flume 和 kafka。对于 python-spark-Streaming 来说，flume 使用 FlumeUtil 类实现 flume 流文件的读入，kafka 使用 KafkaUtil 类实现 kafka 流文件的读入，然后对读入的文件进行流操作处理。

在进行 flume 与 spark Streaming 对接时，flume 输出 sink 选择 avro，再选择某端口；spark 使用 FlumeUtil 通过读取中转端口实现数据的读取。

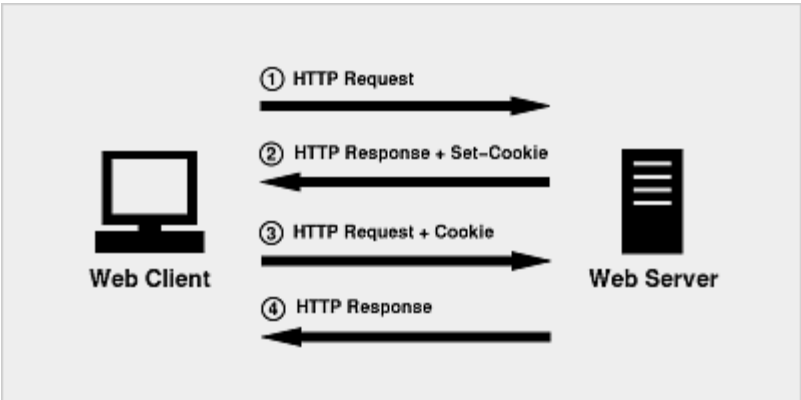


① Flume 端通过配置文件，配置 source 源头 netcat 的某端口，sink 端为 kafka，还需要要定义某个 kafka 的某个 topic（需要事先在 kafka 上定义 topic 名称）；

② python 编写 spark streaming 程序，对 kafka 某个 topic 进行读取，并进行业务流程处理，通过 KafkaProducer 定义某个 topic 输出到 Kafka 中

Web 基础

Cookie 技术：客户端技术——用户状态信息存储在客户端



第一次客户端向服务器请求时，服务器将个人信息转化成 cookie，并保存在**响应报文 Header 的 Set-Cookie 结构体**中；客户端接收到来自服务器的 cookie 之后，在未过期或者未删除情况下，后续对于服务器的请求都在**请求报文的 Header 中加入 Cookie**，服务器接收到来自客户端的 cookie 之后就知道客户端对应的身份信息。

Session：服务器端技术——用户状态信息存储在服务器端

Session 技术则是服务端的解决方案，它是通过服务器来保持状态的。由于 Session 这个词汇包含的语义很多，因此需要在这里明确一下 Session 的含义。首先，我们通常都会把 Session 翻译成会话，因此我们可以把客户端浏览器与服务器之间一系列交互的动作称为一个 Session。从这个语义出发，我们会提到 Session 持续的时间，会提到在 Session 过程中进行了什么操作等等；其次，Session 指的是服务器端为客户端所开辟的存储空间，在其中保存的信息就是用于保持状态。从这个语义出发，我们则会提到往 Session 中存放什么内容，如何根据键值从 Session 中获取匹配的内容等。要使用 Session，第一步当然是创建 Session 了。那么 Session 在何时创建呢？当然还是在服务器端程序运行的过程中创建的，不同语言实现的应用程序有不同创建 Session 的方法，而在 Java 中是通过调用 HttpServletRequest 的 getSession 方法（使用 true 作为参数）创建的。在创建了 Session 的同时，服务器会为该 Session 生成唯一的 Session id，而这个 Session id 在随后的请求中会被用来重新获得已经创建的 Session；在 Session 被创建之后，就可以调用 Session 相关的方法往 Session 中增加内容了，而这些内容只会保存在服务器中，发到客户端的只有 Session id；当客户端再次发送请求的时候，会将这个 Session id 带上，服务器接受到请求之后就会依据 Session id 找到相应的 Session，从而再次使用之。正式这样一个过程，用户的状态也就得以保持了。

每个来访者对应一个 Session 对象，所有该客户的状态信息都保存在这个 Session 对象里。

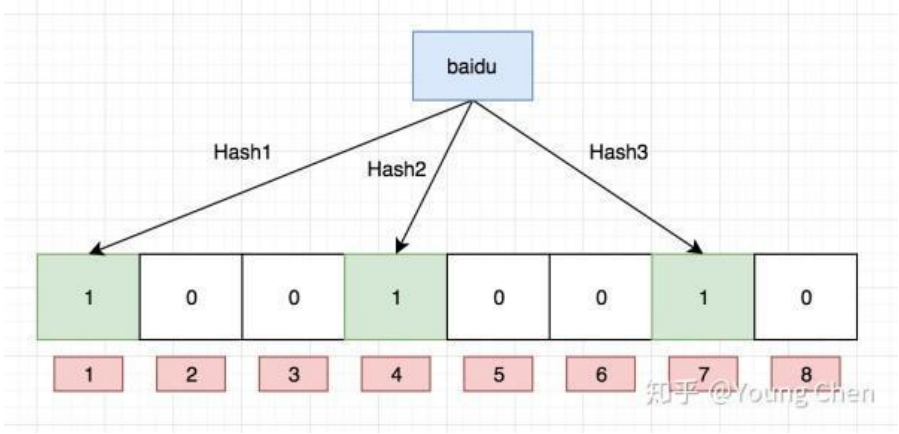
布隆过滤器

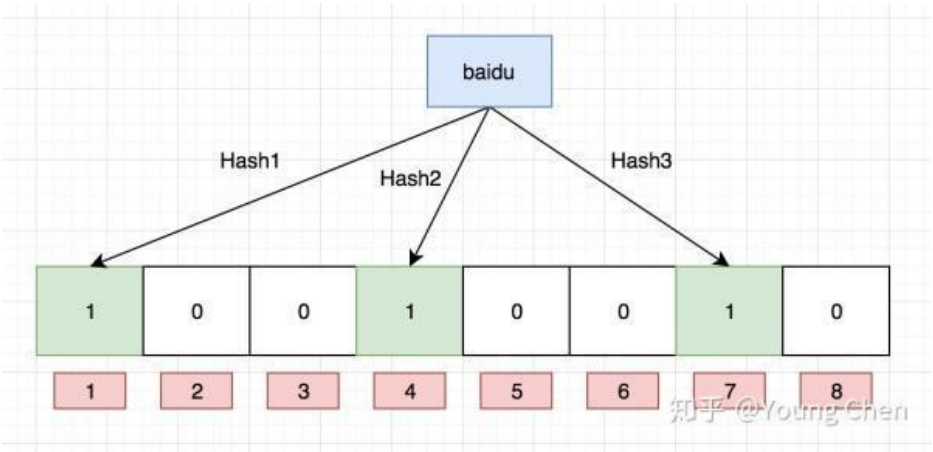
在一个集合中检索某个元素是否存在，可以采用平衡二叉树（O（logN）复杂度）和 hash（O(1)复杂度）。但是当集合数据量非常大时，无法把所有元素调入内容，即无法创建平衡二叉树和 hash 表，此时采用布隆过滤器。

首先选取 hash 的数目 K，和 hash 表的长度 m。初始时刻，hash 表所有元素都是 0。对于集合中的每个元素，分别计算 K 个 hash 值，然后放入 hash 表中，将对应为置为 1。

查询阶段：对于某个元素 x，分别计算 K 个 hash 值，在 hash 表中扫描 K 个值。如果某个值不存在，那么该元素一定不在集合中；如果 K 个元素都存在，那么久判定该元素在该集合中出现过（有一定出错的概率）。

最终整个算法由一定的出错率，该数值与 hash 的个数 K、hash 表长度 m 以及元素个数 N 都有关系。



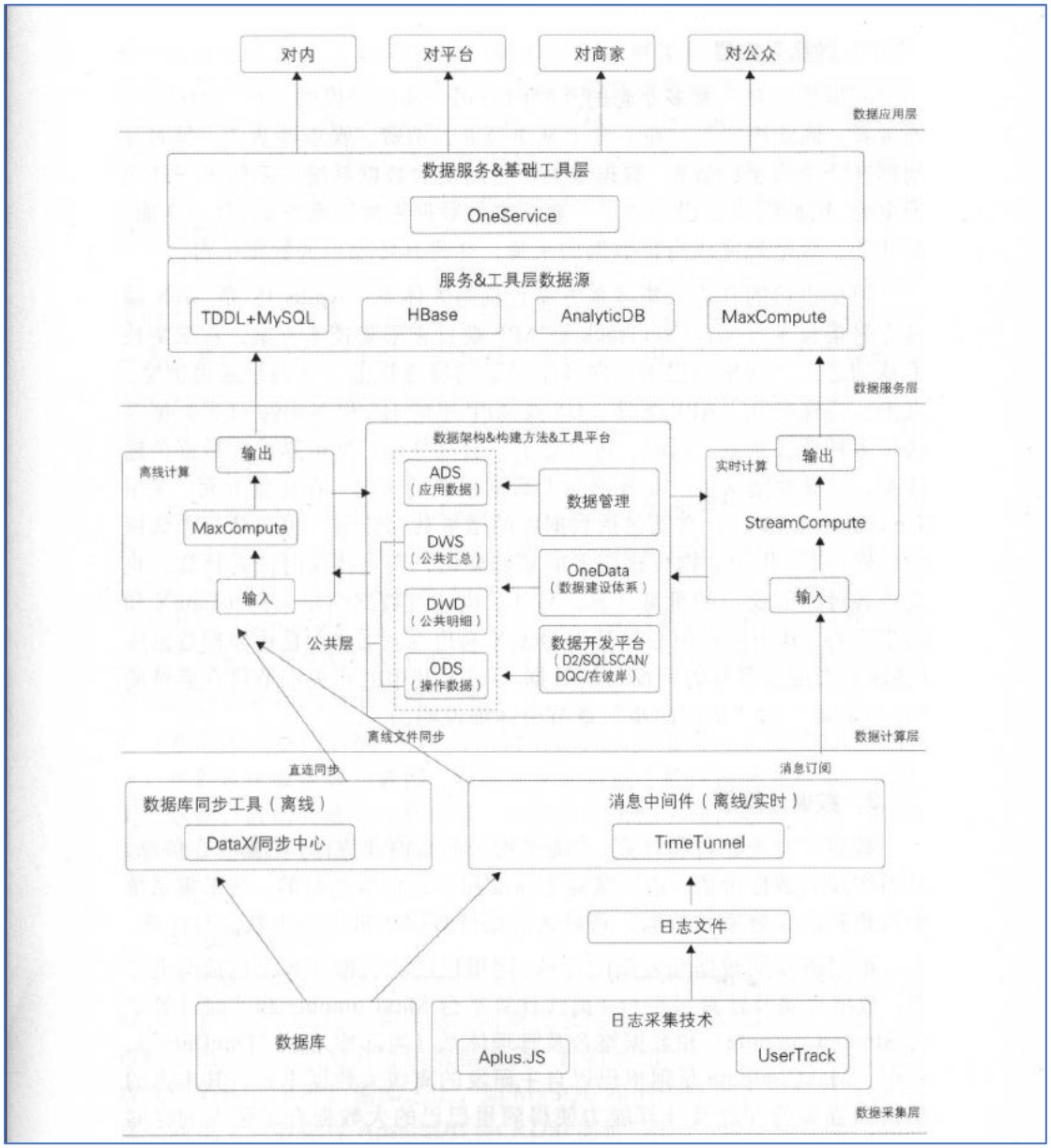


基数估计

针对和布隆过滤器类似的场景，当内容中无法加载所有的数据时，如何计算一个数组的基数（不重复元素的个数）。典型应用场景：**去除重复元素，统计某个店铺的 UV**（User View）数。

当数据量不大时，可以使用**基数排序**方法。

淘宝大数据



底层：飞天系统——HDFS
离线计算：MaxCompute——Hive、Pig
实时计算：StreamCompute——Storm
数据采集：Aplus.JS：web 端日志采集，UserTrack：App 端日志采集
数据同步：DataX——Sqoop
消息中间件：TimeTunnel——Kafka

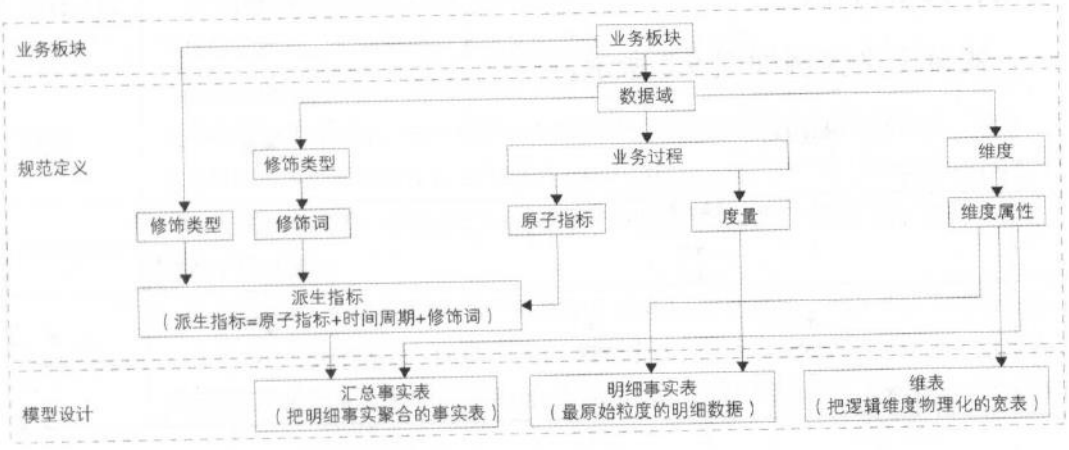


图 9.1　体系架构图

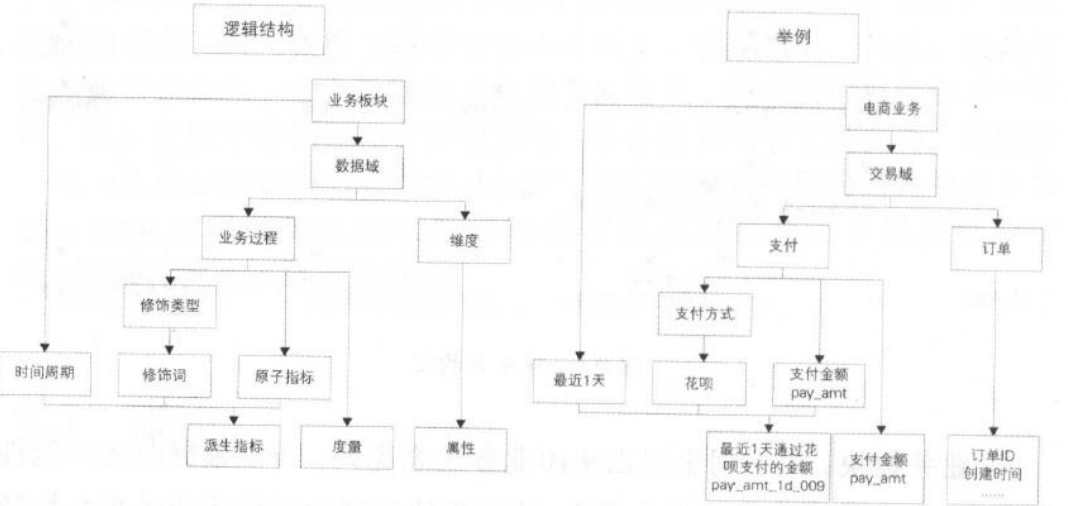


图 9.2　规范定义实例