大量的 IT 组织如今都已自己的数据**架构**，因为都依赖于传统的数据架构。处理多数据源已不再新鲜；这些架构已经连接了多维度的数据源例如 CRM 系统，文件系统和其他商用系统。主要运行的关系型**数据库**有 **Oracle**, DB2 和 Microsoft SQL。

如今，一般的数据分析周期是运行一些周期性脚本直接从数据库提取和处理数据。这些主要由 ETL 工具如 Informatica 或者 Talend. 目标是将这些提炼的数据加载到数据仓库用于将来的分析。

不幸的是，这一方法在周期结束后可能不适合商务的需要了。这些数据流水线可能需要几个小时，几天甚至几周才能完成，但是商务决策的需求可能已经变了。除了处理时间，还有一些数据的自然改变使这些架构难于处理，例如 **数据结构**重构变化导致数据模型的重构或者数据容量导致的伸缩性考虑。

由于不是分布式系统，所以系统扩展比较困难。数据库需要高性能的 CPU，RAM 和存储方案，对于硬件的依赖使系统的扩展性部署非常昂贵。现在大多数 IT 组织已经切换到基于 **Hadoop** 的数据架构了。实际上，不仅是灵活性和技术成本，主要目标是一组商用主机分散处理负载，以及摄取海量的不同类型数据。

Figure 3-1 给出了这一架构的拓扑图。

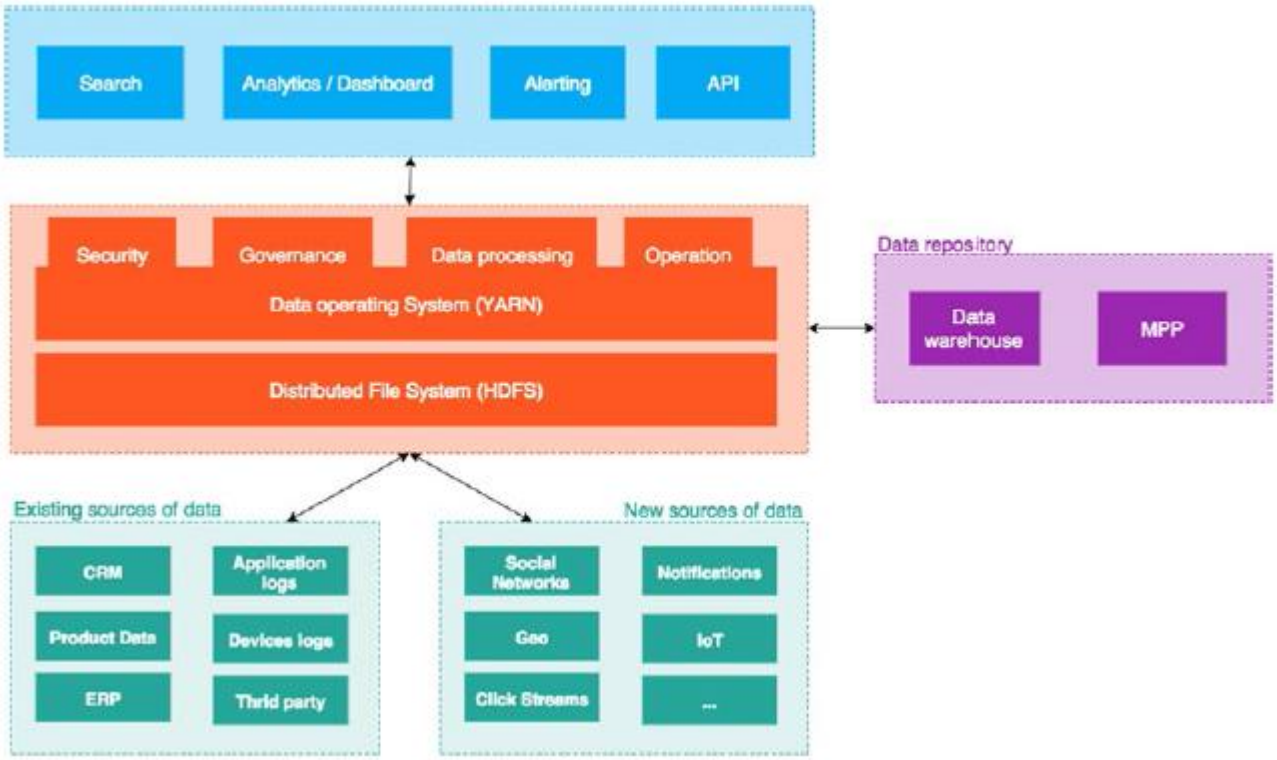


Figure 3-1. 基于 Hadoop 的数据架构

下面看一下数据流水线的涵盖范围，包含了哪些技术，以及这种类型架构的通用实践。

# 处理数据源

如 Figure 3-1 所示，数据可以来自各种内部或者外部的源，但是**大数据**还可以特殊地来自内部应用和设备的日志，例如社交网络，开放数据，甚至传感器。以社交网络为例，IT 组织感兴趣的信息数据会像洪水般流入，但是其中包含了大量无用的信息。

因此，第一是存储数据，然后对提取的重要信息进行处理。这些数据对销售非常有用，尤其是当运行情感分析的时候，可以感知整个社交系统对产品或品牌的感受。

依赖于提供商，数据可能是结构化的，半结构化，或者非结构化的。Listing 3-1 给出了一个半结构化消息的示例.

Listing 3-1. Example of the Semistructured Data of a Tweet

```
{
    "created_at": "Fri Sep 11 12:11:59 +0000 2015",
    "id": 642309610196598800,
    "id_str": "642309610196598785",
    "text": "After a period in silent mode, going back to tweet life",
    "source": "<a href="http://twitter.com/download/iphone" rel="nofollow">
    Twitter for iPhone</a>",
    "truncated": false,
    "in_reply_to_status_id": null,
    "in_reply_to_status_id_str": null,
    "in_reply_to_user_id": null,
    "in_reply_to_user_id_str": null,
    "in_reply_to_screen_name": null,
    "user": {
```

```
"id": 19450096,

"id_str": "19450096",

"name": "Bahaaldine",

"screen_name": "Bahaaldine",

"location": "Paris",

"description": "",

"url": null,

"entities": {

        "description": {

            "urls": []

    }

},

"protected": false,

"followers_count": 59,

"friends_count": 107,

"listed_count": 8,
```

```
        "created_at": "Sat Jan 24 15:32:11 +0000 2009",

        "favourites_count": 66,

        "utc_offset": null,

        "time_zone": null,

        "geo_enabled": true,

        "verified": false,

        "statuses_count": 253,

        "lang": "en",

        "contributors_enabled": false,

        "is_translator": false,

        "is_translation_enabled": false,

        "profile_background_color": "C0DEED",

        "profile_background_image_url": "http://pbs.twimg.com/profile_backgr
ound_

        images/454627542842896384/-n_C_Vzs.jpeg",

        "profile_background_image_url_https": "https://pbs.twimg.com/profile_
background_
```

```
images/454627542842896384/-n_C_Vzs.jpeg",

"profile_background_tile": false,

"profile_image_url": "http://pbs.twimg.com/profile_images/4489050
79673094144/

dz109X55_normal.jpeg",

"profile_image_url_https": "https://pbs.twimg.com/profile_images/448
9050079673094144/

dz109X55_normal.jpeg",

"profile_banner_url": "https://pbs.twimg.com/profile_banners/19450
096/1397226440",

"profile_link_color": "0084B4",

"profile_sidebar_border_color": "FFFFFF",

"profile_sidebar_fill_color": "DDEEF6",

"profile_text_color": "333333",

"profile_use_background_image": true,

"has_extended_profile": false,

"default_profile": false,
```

```
        "default_profile_image": false,
        "following": false,
        "follow_request_sent": false,
        "notifications": false
    },
    "geo": null,
    "coordinates": null,
    "place": null,
    "contributors": null,
    "is_quote_status": false,
    "retweet_count": 0,
    "favorite_count": 0,
    "entities": {
        "hashtags": [],
        "symbols": [],
        "user_mentions": [],
```

```
        "urls": []
    },
    "favorited": false,
    "retweeted": false,
    "lang": "en"
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14

- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29
- 30
- 31
- 32
- 33
- 34
- 35
- 36

- 37
- 38
- 39
- 40
- 41
- 42
- 43
- 44
- 45
- 46
- 47
- 48
- 49
- 50
- 51
- 52
- 53
- 54
- 55
- 56
- 57
- 58

- 59
- 60
- 61
- 62
- 63
- 64
- 65
- 66
- 67
- 68
- 69
- 70
- 71
- 72
- 73
- 74
- 75
- 76
- 77
- 78
- 79
- 80

- 81
- 82
- 83

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19

- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29
- 30
- 31
- 32
- 33
- 34
- 35
- 36
- 37
- 38
- 39
- 40
- 41

- 42
- 43
- 44
- 45
- 46
- 47
- 48
- 49
- 50
- 51
- 52
- 53
- 54
- 55
- 56
- 57
- 58
- 59
- 60
- 61
- 62
- 63

64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83

从例子中可以看到 ,这个文档是一个JSON ,有一组字段 ,其中字符串的元数据来描述tweet。但有些字段非常复杂；有点数组有时候是空的，有时候有包含了一个数据集合，也有纯文本

来表示 tweet 的内容。这就需要思考如何存储这样的数据。把数据放到 HDFS 是不足够的；必须在技术的顶层建立一个元数据结构来支持数据结构的复杂性。这就是有时需要使用 **Hive** 的原因。

当处理海量成分混杂数据的时候，社交网络是复杂性的代表。除了数据结构，还需要将数据分类成逻辑上的子集以便增强数据处理的效果。考虑以情绪分析的例子，从大数据集的非结构化数据中得到有价值信息的位置来组成数据。例如，通用的方法是对数据进行时间分片使数据处理更加聚焦，比方说一年数据中的某个特定周。

也必须注意到要安全地访问数据，多数采用象 Kerberos 或其他的认证提供者。但是如果数据平台涉及到新的使用场景，首先要处理的是多租户技术的安全性。然后，周期性地创建数据镜像以便故障发生时从中提取。所有这些考虑都是标准的，而且可以幸运地由大量供应商提供。这些开箱即用的软件可以保证，或帮助你实现或者配置管理这些概念。

# 处理数据

当从源到目标的纯粹传输时，数据传输时由 ETL 工具处理。这些工具有 Talend, Pentaho, Informatica, 或者 IBM Datastage ，这是大数据项目中最常用的软件。但还是不够的，还需要一些补充的工具例如 Sqoop 来简化数据导入或导出。在任何情况下，使用多种工具来摄取数据，通用的目标存储是 ：HDFS. HDFS 是一个 Hadoop 发布版的入口; 数据需要存储在这样的文件系统中，以便于高层应用和项目的处理。

当 HDFS 存储在数据中的时候，然后如何访问和处理它们呢？

作为一个例子可能是 Hive，它在 HDFS 中创建了一种数据结构，可以方便地访问这些文件。这个结构自身象一个数据表。例如, Listing 3-2 展示了一个处理 tweet 的结构示例。

Listing 3-2. Hive Tweet Structure

```
create table tweets (

    created_at string,

    entities struct <

        hashtags: array ,

        text: string>>,

        media: array ,

        media_url: string,

        media_url_https: string,

        sizes: array >,

        url: string>>,

        urls: array ,

        url: string>>,

        user_mentions: array ,

        name: string,

        screen_name: string>>>,
```

```
geo struct <

    coordinates: array ,

    type: string>,

id bigint,

id_str string,

in_reply_to_screen_name string,

in_reply_to_status_id bigint,

in_reply_to_status_id_str string,

in_reply_to_user_id int,

in_reply_to_user_id_str string,

retweeted_status struct <

    created_at: string,

    entities: struct <

    hashtags: array ,

    text: string>>,

    media: array ,
```

```
    media_url: string,
    media_url_https: string,
    sizes: array >,
    url: string>>,
    urls: array ,
    url: string>>,
    user_mentions: array ,
    name: string,
    screen_name: string>>>,
geo: struct <
    coordinates: array ,
    type: string>,
id: bigint,
id_str: string,
in_reply_to_screen_name: string,
in_reply_to_status_id: bigint,
```

```
in_reply_to_status_id_str: string,

in_reply_to_user_id: int,

in_reply_to_user_id_str: string,

source: string,

text: string,

user: struct <

    id: int,

    id_str: string,

    name: string,

    profile_image_url_https: string,

    protected: boolean,

    screen_name: string,

    verified: boolean>>,

source string,

text string,

user struct <
```

```
        id: int,
        id_str: binary,
        name: string,
        profile_image_url_https: string,
        protected: boolean,
        screen_name: string,
        verified: boolean>
)
PARTITIONED BY (datehour INT)
ROW FORMAT SERDE 'org.openx.data.jsonserde.JsonSerDe'
LOCATION '/user/username/tweets';
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7

- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29

- 30
- 31
- 32
- 33
- 34
- 35
- 36
- 37
- 38
- 39
- 40
- 41
- 42
- 43
- 44
- 45
- 46
- 47
- 48
- 49
- 50
- 51

- 52
- 53
- 54
- 55
- 56
- 57
- 58
- 59
- 60
- 61
- 62
- 63
- 64
- 65
- 66
- 67
- 68
- 69
- 70
- 71
- 72
- 73

- 74

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21

- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29
- 30
- 31
- 32
- 33
- 34
- 35
- 36
- 37
- 38
- 39
- 40
- 41
- 42
- 43

- 44
- 45
- 46
- 47
- 48
- 49
- 50
- 51
- 52
- 53
- 54
- 55
- 56
- 57
- 58
- 59
- 60
- 61
- 62
- 63
- 64
- 65

- 66
- 67
- 68
- 69
- 70
- 71
- 72
- 73
- 74

可以看到，tweets 是一个表中的结构，有一个子结构来描述非结构化文档数据源的复杂性。现在数据安全地存储在 HDFS 中了，由 Hive 来结构化,准备作为处理和查询流水线的一部分。作为一个例子, Listing 3-3 展示了所选数据哈希标签的分布.

Listing 3-3. Top Hashtags

```
SELECT

LOWER(hashtags.text),COUNT(*) AS hashtag_countFROM tweets

LATERAL VIEW EXPLODE(entities.hashtags) t1 AS hashtagsGROUP BY LO

WER(hashtags.text)ORDER BY hashtag_count DESC

LIMIT 15;
```

- 1

```
2
3
4
5
6
7
8
```

```
1
2
3
4
5
6
7
8
```

因为提供了类 SQL 的查询语言，通过 Hive 查询数据非常方便。问题就是查询时延；基本上等同于一个 MapReduce job 的时延. 实际上, Hive 查询被翻译成一个 MapReduce job 执行通用的处理流水线，这导致了长时处理。

当需要实时数据传输时，这就成为了一个问题，例如实时观察最多哈希标签的时候。对例如新兴的技术如 **Spark** 来说，实时处理海量数据不再神秘。不但可以实时处理而且实现简单。例如, Listing 3-4 展示了如何在 MapReduce 中实现一个单词计数功能。

Listing 3-4. MapReduce 的 Word Count (from www.dattamsha.com/2014/09/hadoop-mr-vs-spark-rdd-wordcount-program/)

```
package org.apache.hadoop.examples;


import java.io.IOException;

import java.util.StringTokenizer;


import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.Mapper;

import org.apache.hadoop.mapreduce.Reducer;

import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

import org.apache.hadoop.mapreduce.lib.input.FileSplit;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
```

```
import org.apache.hadoop.util.GenericOptionsParser;



public class WordCount {



    public static class TokenizerMapper extends

    Mapper<Object, Text, Text, IntWritable> {



        private final static IntWritable one = new IntWritable(1);

        private Text word = new Text();



        public void map(Object key, Text value, Context context)

        throws IOException, InterruptedException {

        StringTokenizer itr = new StringTokenizer(value.toString());

        while (itr.hasMoreTokens()) {

            word.set(itr.nextToken());

            context.write(word, one);
```

```
    }

    }

}


public static class IntSumReducer extends

Reducer<Text, IntWritable, Text, IntWritable> {



    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,

    Context context) throws IOException, InterruptedException {

        int sum = 0;

        for (IntWritable val : values) {

            sum += val.get();

        }

        result.set(sum);

        context.write(key, result);
```

```
    }

}


public static void main(String[] args) throws Exception {

    Configuration conf = new Configuration();

    String[] otherArgs = new GenericOptionsParser(conf, args)

    .getRemainingArgs();



    Job job = new Job(conf, "word count");



    job.setJarByClass(WordCount.class);



    job.setMapperClass(TokenizerMapper.class);

    job.setCombinerClass(IntSumReducer.class);

    job.setReducerClass(IntSumReducer.class);
```

```
        job.setOutputKeyClass(Text.class);

        job.setOutputValueClass(IntWritable.class);



        FileInputFormat.addInputPath(job, new Path(otherArgs[0]));

        FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));



        System.exit(job.waitForCompletion(true) ? 0 : 1);

    }

}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10

- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29
- 30
- 31
- 32

- 33
- 34
- 35
- 36
- 37
- 38
- 39
- 40
- 41
- 42
- 43
- 44
- 45
- 46
- 47
- 48
- 49
- 50
- 51
- 52
- 53
- 54

- 55
- 56
- 57
- 58
- 59
- 60
- 61
- 62
- 63
- 64
- 65
- 66
- 67
- 68
- 69
- 70
- 71
- 72
- 73

- 1
- 2
- 3

- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25

- 26
- 27
- 28
- 29
- 30
- 31
- 32
- 33
- 34
- 35
- 36
- 37
- 38
- 39
- 40
- 41
- 42
- 43
- 44
- 45
- 46
- 47

- 48
- 49
- 50
- 51
- 52
- 53
- 54
- 55
- 56
- 57
- 58
- 59
- 60
- 61
- 62
- 63
- 64
- 65
- 66
- 67
- 68
- 69

- 70
- 71
- 72
- 73

Listing 3-5 展示了 Spark 是如何做的（**Python**）.

Listing 3-5. Spark 的 Word Count

```
from pyspark import SparkContext

logFile = "hdfs://localhost:9000/user/bigdatavm/input"

sc = SparkContext("spark://bigdata-vm:7077", "WordCount")

textFile = sc.textFile(logFile)

wordCounts = textFile.flatMap(lambda line:line.split()).map(lambda word: (w
ord, 1)).reduceByKey(lambda a, b: a+b)

wordCounts.saveAsTextFile("hdfs://localhost:9000/user/bigdatavm/output")
```

- 1
- 2
- 3
- 4
- 5
- 6

- 1
- 2
- 3
- 4
- 5
- 6

这就需要分割架构成多个部分来处理特定的需求，一个是批处理，另一个是流处理。

# 架构分割

当处理海量数据的时候，Hadoop 带来了大量的解决方案，但也为资源分配和管理存储数据带来了挑战，我们总是希望在保持最小时延的同时而消减成本。和其他架构类似，数据架构满足了 SLA 驱动的需求。因此，每个 job 不应该均等地消耗每个资源，这就要求或者是可管理的，或者有一个优先级系统，或者有相互独立的架构，硬件，网络等等。

下面，将讨论现代数据架构如何按照 SLA 需要来分割不同的使用水平。为了方便解释，Figure 3-2.解释了重新分区。

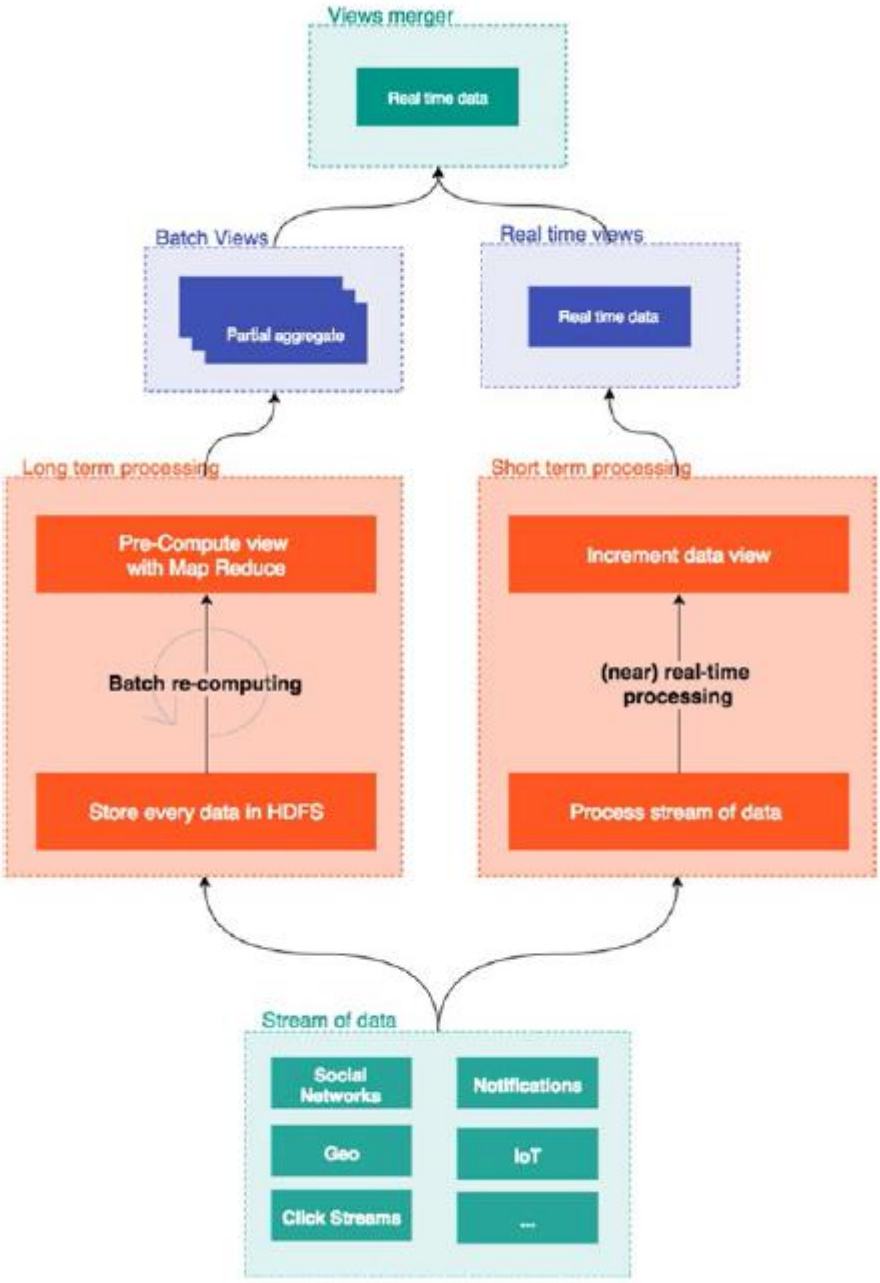


Figure 3-2. Modern data architecture

已经看到，架构分成如下部分：

- 

  长时处理部分

- 

- 

短时处理部分

- 
- 

视图融合部分

- 

看一下每个部分，并解释所影响的角色。

# 批处理

长时处理任务，或者批处理，是 Hadoop 的第一代实现，例如 MapReduce, Hive, Pig, 等等. 这些 jobs 趋向于处理海量数据，以及摄取数据或者聚合数据。使用 HDFS 作为数据的分布和调度，依赖所使用的发布版可以通过不同的工具来管控。

一般的, 这些任务的目标是保持数据的聚合计算结果，以及分析结果。已经说过，批处理是大数据开始实现时的头等公民，因为这是处理数据的自然方式：提取或采集数据，然后调度任务。批处理在完成聚合计算时要花费大量的时间。这些任务主要满足商用系统的处理需要而不是处理数据流。 批处理非常容易管理和监控，这是由于是单次运行，而流式系统需要连续监控。现在通过 YARN, 也可以管理批处理的资源分配。这种方式，使 IT 组织可以依赖每个批处理的 SLA 来分割批处理架构。

# 处理优先级

当对待批处理的时候, IT 组织希望对操作和处理进行总体控制，例如调度或优先处理某些任务。和大多数 IT 系统类似，一个数据平台同样开始于一个引导用例，而该用例可能影响到

其他组织的其他部分，又要增加更多的用例到这个平台上。一个简单的转化就是数据平台变成了多租户数据平台，依赖不同的使用场景有着很多 SLA。

在 Hadoop 2.0 和基于 Yarn 的架构中，多租户技术提供特性是允许用户访问同样的数据平台但在不同集群有着不同的处理能力。YARN 也允许运行非 MapReduce 应用，所以通过 ResourceManager 和 YARN 容量调度器,可以跨应用类型来提供任务的优先级。Hadoop 工作负载的分发由容量调度器完成。

这种配置优雅地安排可预测的集群资源，给我们安全和充分利用集群。在任务队列可以设置任务的使用百分比。Figure 3-3 解释了这一概念。


Figure 3-3. YARN job 队列

该示例解释了三种队列的不同优先级: 高,低 ， 和默认. 这可以翻译成简单的 YARN 容量调度器配置，见 Listing 3-6.

```
<property><name>yarn.scheduler.capacity.root.queues</name><value>default,highPriority,lowPriority</value></property><property><name>yarn.scheduler.capacity.root.highPriority.capacity</name><value>60</value></property><property><name>yarn.scheduler.capacity.root.lowPriority.capacity</name><value>20</value></property><property><name>yarn.scheduler.capacity.root.default.capacity</name><value>10</value></property>
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13

- 14
- 15
- 16

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16

每个队列有一个最小的集群容量，而且是弹性的。这意味着如果有空闲资源，这个队列可以被最小化执行。当然，有可能是最大容量

见 Listing 3-7.

Listing 3-7. 最大的 Queue 容量

```
<property><name>yarn.scheduler.capacity.root.lowPriority.maximum-capacit
y</name><value>50</value></property>
```

```
1
2
3
4
```

```
1
2
3
4
```

这一配置设置了容量，所以一个人提交了一个(例如，一个 MapReduce job),可以依赖所期望的需求提交到一个特殊队列，见 Listing 3-8.

Listing 3-8. 提交一个 MapReduce Job

```
Configuration priorityConf = new Configuration();

priorityConf.set("mapreduce.job.queuename", queueName);
```

- 1
- 2

- 1
- 2

通过 YARN 容量调度器，批处理在资源管理上非常高效，在工业界有着大量的应用，例如给推荐引擎分配比非重要需求的数据处理更多的资源。但是谈到了推荐，大多数 IT 系统 现在是一短时处理任务，以及依赖于流架构。

# 流处理

短时处理，或者叫流式处理，用于摄取高吞吐量数据. 流处理方案可以处理海量数据，而且是高分布，可伸缩，和容错的。这种架构解决了一系列的挑战。已经说过，一个主要目的是处理海量数据。尽管以前已经有各种流式技术，但现在是高可用，弹性和高性能的。高性能是应对数据容量，复杂性和大小的增长。

如果数据容量增长了，这些架构能够无缝集成各种数据源和应用，例如数据仓库，文件，数据历史，社交网络，应用日志等等。这需要提供一致性的**敏捷** API，面向客户端的 API，以及能够将信息输出到各种渠道，例如通知引擎，搜索引擎，和第三方应用。基本上，这样的技术有更多实时响应的约束。

最后，从流式架构中，用户最想得到的就是实时分析，需求很清楚，组成如下：实时发现数据，更容易地查询数据，主动监控事件的阈值以通知用户和应用。

流架构首先用在金融领域，这里有着高吞吐量交易的使用场景，但是已经扩展到大量其他的

使用场景，主要是电子商务，电信，防伪监测，和分析。从而诞生了两个主要技术：**Apache Spark** 和 Apache Storm.

这里选择了 Spark，有很好的社区支持，见 Figure 3-4 .

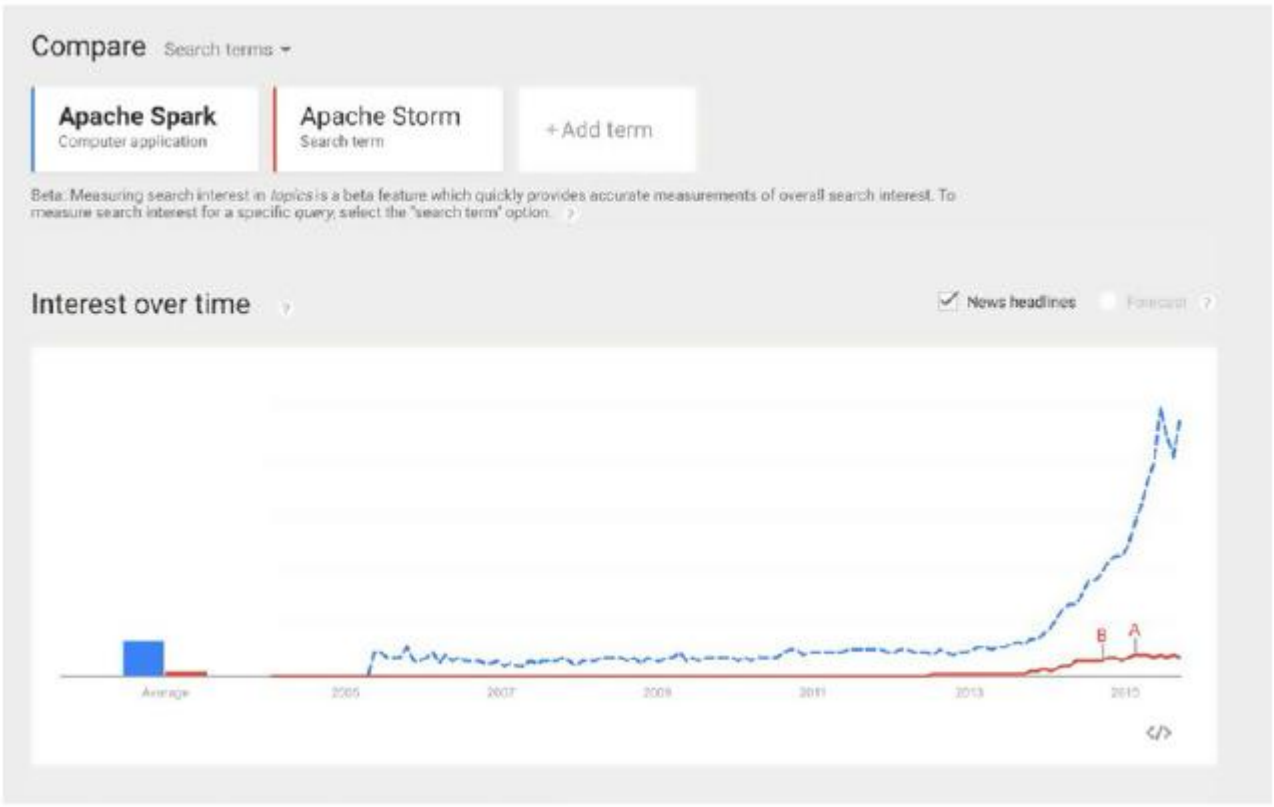


Figure 3-4. Apache Spark 和 Apache Storm 的 Google 趋势

有专门的章节来描述如何将不同的技术结合起来，包括 Spark 的实时流处理和搜索分析。

# Lambda 架构的概念

前面谈到将数据架构分成三个部分：批处理,流处理和服务架构. 尽管批处理还是现存 IT 组织中数据架构的通用实践，当还不能满足大多数流式数据的真正需求，如果需要的话，需要将数据存储在一个面向批处理的文件系统中。部署一个流式架构不像 IT 组织批处理架构那

么简单，与之对应，流处理架构带来了更多的操作复杂性，架构必须要设计成吸收无用突发数据以维持较低的响应时间。

当感到 Hadoop 发布版部署麻烦的时候，开始的方法是简化流架构以便有相同的处理 API 等等。

甚至如果 SLA 不能满足以及不希望以数秒或数分钟来获取数据，需要消减部署的繁琐性。在我看来，一个流式架构是现代数据架构的自然演进。它消减了软件的复杂性，就像第一代大数据架构消减了硬件那样。我们这一架构的选择可能不是通用的，但确实是广泛使用的，这一技术栈应该可以适应 90%的使用场景。

lambda 架构是两个世界中的最好品种。数据是暂态的，处理是实时的，可以重新计算和创建在批处理层的部分聚合数据，最后在服务层融化服务。为例实现这一范式，选择以下技术:

- Logstash-数据摄取和转发
- Apache Kafka 分发数据
- Logtash agent 处理日志
- Apache Spark 流处理
- Elasticsearch 作为服务层

Figure 3-5 介绍了这一架构.

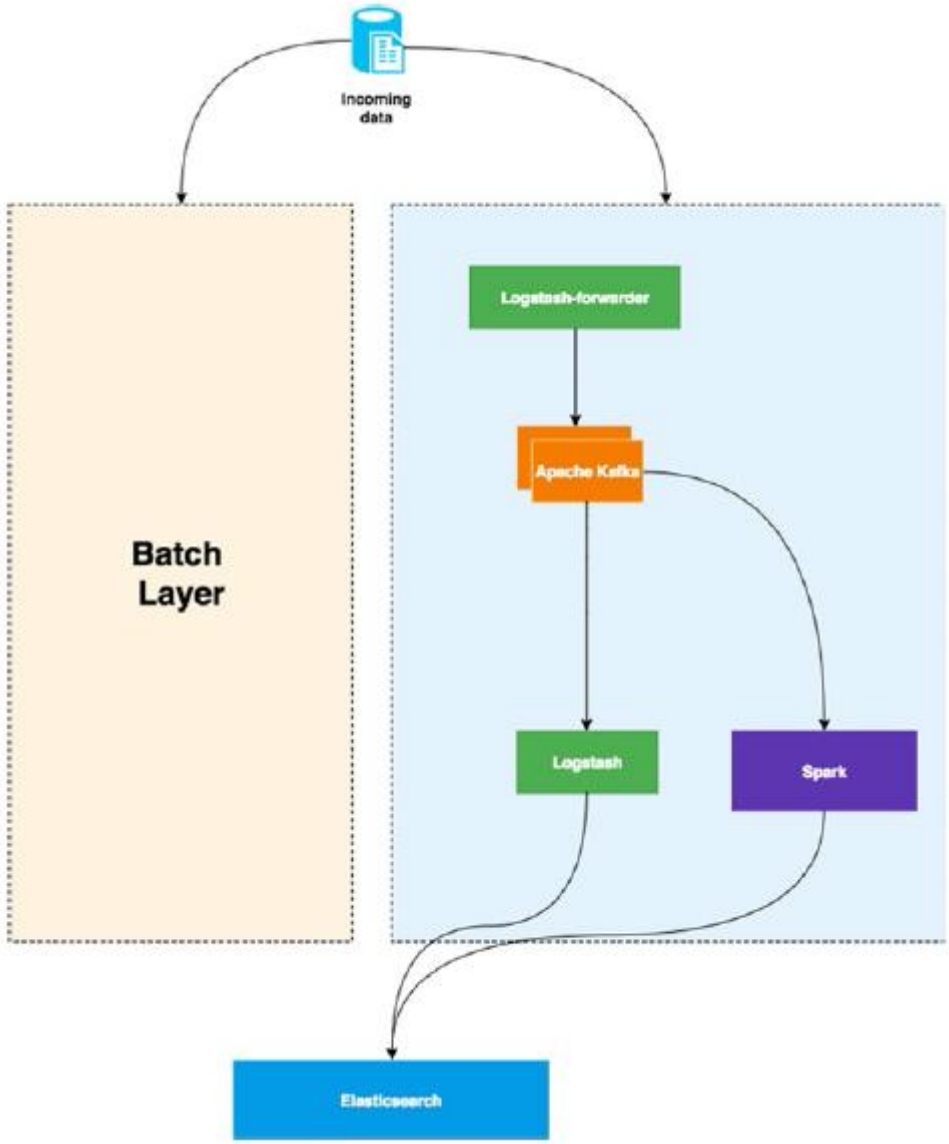


Figure 3-5. Lambda 架构的加速和服务层

lambda 架构通常用于电子商务网站来实现推荐或者安全分析等不同的目的。例如点击流数据，可以从中提取多重有意义的见解。

- 

一方面,使用长时处理层，处理点击流，聚合数据，与其他数据源交叉使用来建立推荐引擎。这个例子中 ,利用 点击流数据与其他数据源包含的人口统计信息的相关性，在 ElasticSearch 中构建索引视图。

- 
- 

另一方面，同样的数据被用来点检测。实际上，大多数电子商务应用都会面对安全上的威胁 ,一种方式是通过流处理层分析用户的点击行为而实时地将 IP 地址放入黑名单。参见 Figure 3-5，可以使用 Spark 处理复杂的互相关性 ,或者运行**机器学习**进程在 ElasticSearch 索引前来提取数据。

- 