大数据

> *大数据是什么*

大数据本质也是数据，但是在数据量相对较大的情况下，再使用传统的手段来处理文件就会导致效率变得比较低，因此需要采取更好的手段来提高处理效率。当然，随着大数据的发展，围绕着大数据也衍生出来了越来越多的产业，例如BI(商业智能)等。

> *大数据的特征*

大数据的特征主要有6个:

1. **Volume**：数据量大，包括采集、存储和计算的量都非常大。大数据的起始计量单位至少是T、P（1024个T）、E（100万个T）或Z（10亿个T）。到目前为止，人类所生产的所有印刷材料（书本、杂志、报刊等）的数据量大约是200PB，而历史上全人类总共说过的话的数据量大约是5EB。当前，典型个人计算机硬盘的容量为TB量级，而一些大企业的数据量已经接近或者达到EB量级。
2. **Variety**：种类样式和来源多样化。数据样式包括结构化数据(数据本身有结构并且数据解析之后能够用一张或者几张固定的表来进行存储)；半结构化数据(数据本身有结构但是解析之后无法用一张或者几张固定的表来进行存储，例如json，xml等)；非结构化数据(数据本身没有结构并且解析之后无法用几张固定的表来进行存储)，具体表现为网络日志、音频、视频、图片、地理位置信息等等，多类型的数据对数据的处理能力提出了更高的要求。
3. **Value**：数据价值密度相对较低，或者说是浪里淘沙却又弥足珍贵。价值密度指的是想要的数据在总的数据量中的占比。随着互联网以及物联网的广泛应用，信息感知无处不在，信息海量，但价值密度较低，如何结合业务逻辑并通过强大的机器算法来挖掘数据价值是大数据时代最需要解决的问题。随着网络的发展，价值密度越来越低，但是不意味着获取到的数据越来越少，恰恰相反，获取到的数据是在变多的。只是想要的数据的增长速度比不上样本总量的增长速度
4. **Velocity**：数据增长速度快，处理速度也快，时效性要求比较高。这是大数据区分于传统数据挖掘的最显著特征。根据IDC（Internet Data Center，互联网数据中心）的“数字宇宙”的报告，到2020年，全球数据使用量将达到35.2ZB。在如此海量的数据面前，处理数据的效率就是企业的生命。
5. **Veracity**：数据的准确性和可信赖度，即数据的质量随着信息的爆炸以及网络的发展，信息的来源良莠不齐，导致数据的可信赖度变得不同。
6. **Valence**：大数据之间的连通性。随着大数据的发展，衍生出来了很多的技术、模块和产业，这个时候，就不得不考虑这些模块、技术和产业之间的关系。

当然，近年来，随着大数据的发展，又延伸出来其他的一些技术，例如：Vitality(动态性)、Visualization(可视化)、Validity(合法性)等

> *大数据的应用*

大数据现在和我们的生活紧密联系，密不可分。例如风险控制、精准营销、推荐系统、用户画像等很多方面都有大数据的身影

Hadoop

> *Hadoop集群的运行模式*

Hadoop集群一共有三种运行模式：

1. 单机模式：在一台服务器上安装Hadoop，只能启动Hadoop的MapReduce模块
2. 伪分布式：在一台服务器上安装Hadoop，利用多个进程模拟集群环境，能够启动Hadoop的大部分功能
3. 完全分布式：在集群中安装Hadoop，能够启动Hadoop的所有功能

> *伪分布式中需要注意什么*

伪分布式（Pseudo）适用于开发和测试环境，在这个模式中，所有守护进程都在同一台机器上运行

> *Hadoop各个进程的端口号*

Hadoop进程的端口号如下表所示

进程	属性	HADOOP2.X	HADOOP3
NameNode	dfs.namenode.https-address	50470	9871
	dfs.namenode.http-address	50070	9870
	fs.defaultFS	8020	9820
SecondaryNameNode	dfs.namenode.secondary.https-address	50091	9869
	dfs.namenode.secondary.http-address	50090	9868
DataNode	dfs.datanode.ipc.address	50020	9867
	dfs.datanode.address	50010	9866
	dfs.datanode.https.address	50475	9865

进程	属性	HADOOP2.X	HADOOP3
	dfs.datanode.http.address	50075	9864
ResourceManager	yarn.resourcemanager.webapp.address	8088	8088
NodeMangager	X	X	X

> *DataNode如何检查数据是否损坏*

DataNode保证数据完整性的方法如下：

1. 当DataNode读取Block的时候，它会计算CheckSum
2. 如果计算后的CheckSum，与Block创建时值不一样，说明Block已经损坏
3. Client读取其他DataNode上的Block
4. DataNode在其文件创建后周期验证CheckSum

> *MapReduce的优缺点*

优缺点如下：

1. 优点：
    a. MapReduce易于编程：用户只需要简单的实现MapReduce提供的一些接口，就可以完成一个分布式程序，这个分布式程序可以分布到大量廉价的PC机器上运行
    b. 具有良好的扩展性：当当前的集群的计算资源不能得到满足的时候，可以通过简单的增加机器来扩展它的计算能力
    c. 高容错性：MapReduce设计的初衷就是使程序能够部署在廉价的PC机器上，这就要求它具有很高的容错性。例如，如果集群中某一台服务器宕机，那么MapReduce可以把上面的计算任务转移到另外一个节点上运行，不至于这个任务运行失败，而且这个过程不需要人工参与，而完全是由Hadoop内部完成的
    d. 适合PB级以上海量数据的离线处理：可以实现上千台服务器集群并发工作，提供数据处理能力
2. 缺点：
    a. 不擅长实时计算：MapReduce的运行速度相对比较低，一般在毫秒或者秒级内返回结果，因此不适合于实时分析的场景
    b. 不擅长流式计算：流式计算的输入数据是动态的，而MapReduce要求输入的数据集是静态的，不能动态变化。这是因为MapReduce自身的设计特点决定了数据源必须是静态的
    c. 不擅长DAG（有向图）计算：多个应用程序存在依赖关系，后一个应用程序的输入为前一个的输出。在这种情况下，MapReduce并不是不能做，而是使用后，每个MapReduce作业的输出结果都会写入到磁盘，会造成大量的磁盘IO，导致性能非常的低下

> *FileInputFormat的切片问题*

常见切片问题如下：

1. 如果当前文件是一个空文件，则整个文件作为一个切片进行处理
2. 文件存在可切与不可切的问题。默认情况下，文件是可切的；但是如果是压缩文件，那么不一定可切
3. 如果文件不可切，则整个文件无论多大都会作为一个切片进行处理
4. 如果不指定，则一个Split的大小默认和Block大小是一致的
5. 如果需要调小Split，那么需要调小maxSize；如果需要调大Split，那么可以调大minSize
6. 在切片过程中需要注意阈值SPLIT_SLOP=1.1的问题。即当剩余的字节数/splitSize>1.1才会继续切片

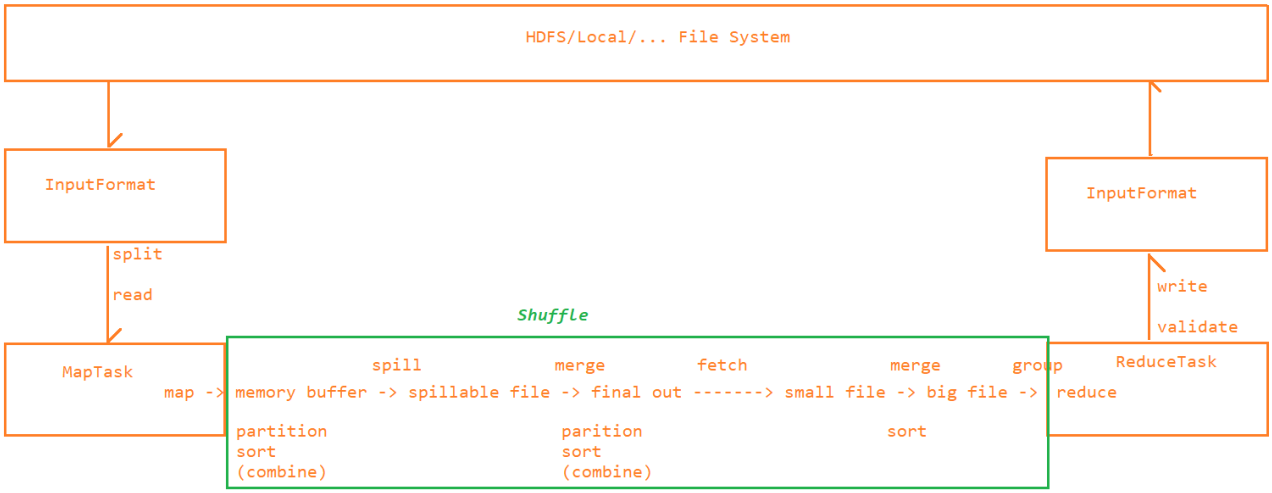
> *TextInputFormat读取数据的问题*

TextInputFormat读取数据的注意问题如下：

1. 在读取数据之前，TextInputFormat先获取文件的压缩编码，判断文件是否是一个压缩文件。如果不是压缩文件，则默认数据是可切的；如果不是压缩文件，则判断是否是一个可切的压缩文件。在Hadoop中，默认只有BZip2(文件后缀是.bz2)这一种压缩格式是可切的
2. 在读取数据的时候，除了第一个MapTask和最后一个MapTask，其他的MapTask都是从当前切片的第二行开始，读取到下一个切片的第一行。其中第一个MapTask要多读取一行数据，最后一个MapTask要少读取一行数据。这样做的目的是为了保证数据的完整性

> *TextInputFormat读取数据的问题*

工作流程图如下



> *YARN的工作机制*

步骤如下：

1. MR程序提交到客户端所在的节点
2. YarnRunner向ResourceManager申请一个Application
3. RM将该应用程序的资源路径返回给YarnRunner
4. 该程序将运行所需资源提交到HDFS上
5. 程序资源提交完毕后，申请运行mrAppMaster
6. RM将用户的请求初始化成一个Task
7. 其中一个NodeManager领取到Task任务
8. 该NodeManager创建容器Container，并产生MRAppmaster
9. Container从HDFS上拷贝资源到本地
10. MRAppmaster向RM 申请运行MapTask资源
11. RM将运行MapTask任务分配给另外两个NodeManager，另两个NodeManager分别领取任务并创建容器
12. MR向两个接收到任务的NodeManager发送程序启动脚本，这两个NodeManager分别启动MapTask，MapTask对数据分区排序
13. MrAppMaster等待所有MapTask运行完毕后，向RM申请容器，运行ReduceTask
14. ReduceTask向MapTask获取相应分区的数据
15. 程序运行完毕后，MR会向RM申请注销自己

> *Hadoop运行速度慢的原因*

排除硬件原因之后，那么其他的原因可能是：

1. 产生了数据倾斜
2. 小文件数量过多
3. 大量不可切分的文件
4. Spill次数过多
5. Merge次数过多

> *Hadoop常见优化方案*

Hadoop的优化主要是针对Shuffle过程的优化，例如

1. 增大缓冲区，可以考虑调节成250M~400M之间
2. 增加Combiner过程，但是注意的是并不是所有的场景都适合于使用Combiner
3. 考虑数据压缩，但是解压同样需要花费时间
4. 适当的增加fetch线程的数量

> *数据倾斜的解决方案*

常见解决方案如下

1. 对数据进行抽样，利用样本代替整体
2. 合理定义分区
3. 对数据进行Combine，减少Reduce的输入数据量
4. 尽量在Map端进行join而不是Reduce端进行join

> *小文件优化*

市面上针对小文件的处理手段无非两种：合并或者打包，而我们要考虑的是发生在哪个阶段，例如

1. 在收集数据的时候，就将小文件收集到一个文件中，合并成一个大文件
2. 在MapReduce处理数据之前，将数据合并或者打包
3. 在MapReduce处理数据的时候，考虑使用CombineTextInputFormat对数据进行合并

> *当前结构中HDFS的弊端*

一般在HDFS架构中，只有1个NameNode是对外开发接收操作的，其他NameNode一般是作为备份来使用，除非Active NameNode当即，那么Standy NameNode才会切换为Active状态，此时弊端如下：

1. NameNode会将元数据维系在内存中。实际开发中，一台服务器大概能腾出50G左右的内存给NameNode来使用，也就意味着一台服务器大概能存储3亿~4亿条元数据，经过计算，意味着NameNode所管理的集群大概能够存储12~15PB的数据。但是在现在的开发中，很多大型企业的数据量已经超过上百PB，原始的NameNode架构就不能满足这个需求
2. NameNode无法做到程序的隔离。所有的元数据都维系在一个NameNode上，意味着如果某一个任务占用的资源比较多，那么就会影响其他在进行的任务
3. 所有的请求都只能访问这唯一的一个NameNode，此时NameNode的并发量就成了整个HDFS的并发瓶颈

> *Hadoop宕机*

常见方案如下：

1. 如果是MapReduce任务过重造成系统宕机，那么此时要控制Yarn同时运行的任务数，和每个任务申请的最大内存。可以通过参数`yarn.scheduler.maximum-allocation-mb`来调节
2. 如果是因为写入文件过量造成NameNode宕机，那么此时需要调高Kafka的存储大小，控制从Kafka到HDFS的写入速度。高峰期的时候用Kafka进行缓存，高峰期过去数据同步会自动跟上

Flume

> *如何利用Flume实时收集日志中新产生的数据*

可以使用Exec source，利用command=tail -F XXX来监听，例如

```
a1.sources = s1
a1.channels = c1
a1.sinks = k1

# 配置Exec Source
# 类型必须是exec
a1.sources.s1.type = exec
# 要执行的命令
a1.sources.s1.command = tail -F /home/software/hive/logs/hive.log
# Shell执行的类型
a1.sources.s1.shell = /bin/bash -c

a1.channels.c1.type = memory

a1.sinks.k1.type = logger

a1.sources.s1.channels = c1
a1.sinks.k1.channel = c1
```

> *如何利用Flume实时监听指定目录的变化*

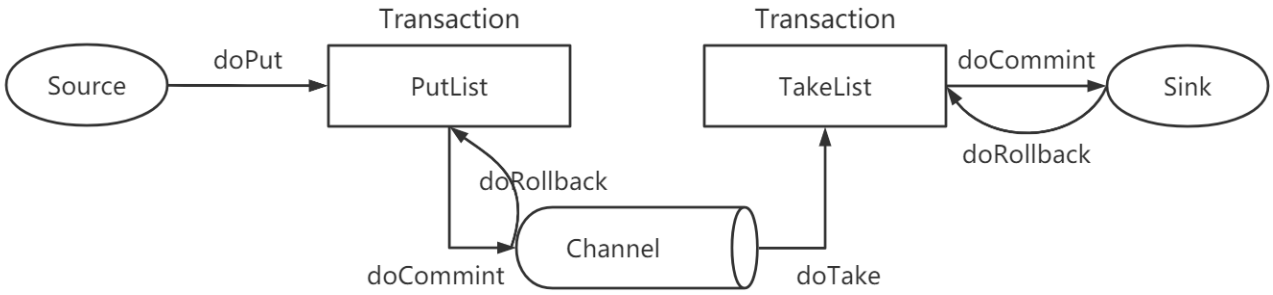
可以利用Spooling Directory Source来监听，例如

```
a1.sources = s1
a1.channels = c1
a1.sinks = k1

# 配置Spooling Directory Source
# 类型必须是spooldir
a1.sources.s1.type = spooldir
# 要监听的目录
a1.sources.s1.spoolDir = /home/flumedata

a1.channels.c1.type = memory
```

```
a1.sinks.k1.type = logger

a1.sources.s1.channels = c1
a1.sinks.k1.channel = c1
```

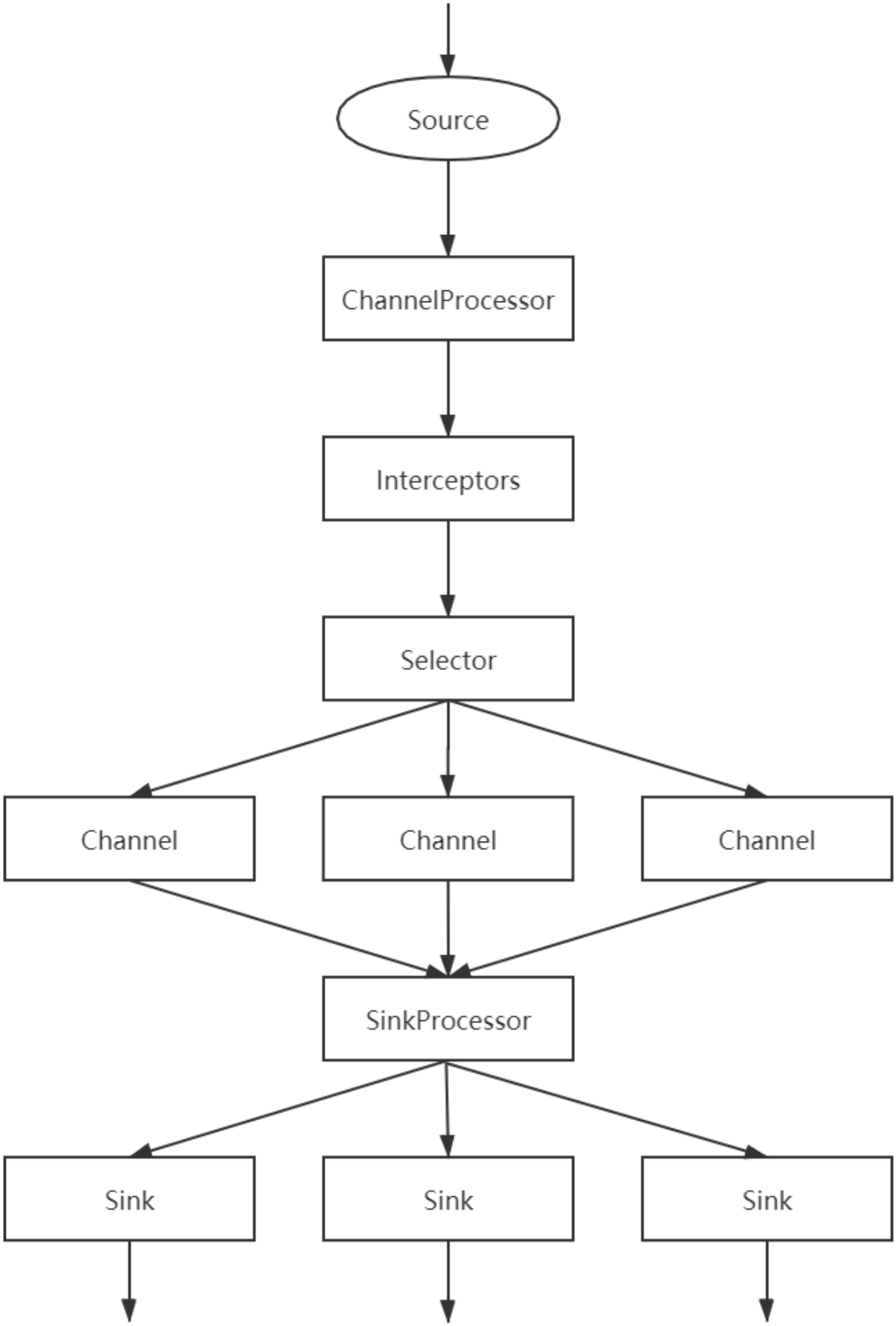
> *Flume的事务机制*

Flume的事务机制（类似数据库的事务机制）：Flume使用两个独立的事务分别负责从Soucrce到Channel，以及从Channel到Sink的事件传递。比如spooling directory source 为文件的每一行创建一个事件，一旦事务中所有的事件全部传递到Channel且提交成功，那么Soucrce就将该文件标记为完成。同理，事务以类似的方式处理从Channel到Sink的传递过程，如果因为某种原因使得事件无法记录，那么事务将会回滚。且所有的事件都会保持到Channel中，等待重新传递。如下图所示：



> *Flume的执行流程*

如下图所示



Flume采集数据会丢失吗

根据Flume的架构原理，Flume是不可能丢失数据的，其内部有完善的事务机制，Source到Channel是事务性的，Channel到Sink是事务性的，因此这两个环节不会出现数据的丢失，唯一可能丢失数据的情况是Channel的类型是memory，agent宕机导致数据丢失，或者Channel存储数据已满，导致Source不再写入，未写入的数据丢失。虽然Flume不会丢失数据，但是有可能造成数据的重复，例如数据已经成功由Sink发出，但是没有接收到响应，Sink会再次发送数据，此时可能会导致数据的重复。

Hive

> *Hive的优点和缺点*

Hive的优点和缺点如下

1. 优点：
    a. 操作接口采用类SQL语法，提供快速开发的能力(简单、容易上手)
    b. 避免了去写MapReduce，减少开发人员的学习成本
    c. Hive的执行延迟比较高，因此Hive常用于数据分析，对实时性要求不高的场合
    d. Hive优势在于处理大数据，对于处理小数据没有优势，因为Hive的执行延迟比较高
    e. 支持用户自定义函数，用户可以根据自己的需求来实现自己的函数
2. 缺点
    a. Hive的HQL表达能力有限：迭代式算法无法表达；不擅长数据挖掘，由于MapReduce数据处理流程的限制，效率更高的算法却无法实现
    b. Hive的效率比较低:Hive自动生成的MapReduce作业，通常情况下不够智能化；Hive调优比较困难，粒度较粗

> *Hive和数据库的区别*

区别如下

1. 查询语言：由于SQL的易学性，因此Hive在设计的时候采用了类SQL(HQL)语言，使得熟悉SQL开发的开发者可以很方便的使用Hive进行开发。但是大家在使用的过程中也会发现，HQL并不是完全和SQL一致，例如row format delimited fields terminated by之类的，是Hive提供的解析文本的格式，但是SQL中并不存在
2. 数据存储位置：Hive 是基于Hadoop来构建的，因此 Hive 中管理的数据都是落地在 HDFS上的；而数据库则可以将数据保存在块设备或者本地文件系统中
3. 数据更新：由于Hive是针对历史数据/已有数据进行处理，而这类数据的特点是读多写少。因此，Hive中不建议对数据进行修改，所有的数据在加载的时候都已经

确定好。而数据库中的数据往往是实时捕获的数据，通常是需要经常进行修改的，因此可支持完整的CRUD操作

4. 索引：由于数据本身的一些特性(例如日志中的数据可能无法做到唯一、非空之类的)，所以Hive在加载数据的过程中不会对数据进行任何处理，甚至不会对数据进行扫描，因此也不会针对某一列数据建立索引。Hive要访问数据中满足条件的特定值时，需要暴力扫描整个数据，因此访问延迟较高。由于 MapReduce 的引入，Hive 可以并行访问数据，因此即使没有索引，对于大数据量的访问，Hive 仍然可以体现出优势。数据库中，通常会针对一个或者几个列建立索引，因此对于少量的特定条件的数据的访问，数据库可以有很高的效率，较低的延迟
5. 执行：Hive中大多数查询的执行是通过 Hadoop 提供的 MapReduce 来实现的。而数据库通常有自己的执行引擎
6. 执行延迟：Hive 在查询数据的时候，由于没有索引，需要扫描整个表，因此延迟较高。另外一个导致 Hive 执行延迟高的因素是 MapReduce框架。由于 MapReduce 本身具有较高的延迟，因此在利用MapReduce 执行Hive查询时，也会有较高的延迟。相对的，数据库的执行延迟较低。当然，这个低是有条件的，即数据规模较小，当数据规模大到超过数据库的处理能力的时候，Hive的并行计算显然能体现出优势
7. 可扩展性：由于Hive是建立在Hadoop之上的，因此Hive的可扩展性是和Hadoop的可扩展性是一致的（世界上最大的Hadoop 集群在 Yahoo!，2009年的规模在4000台节点左右）。而数据库由于 ACID 语义的严格限制，扩展行非常有限。目前最先进的并行数据库 Oracle 在理论上的扩展能力也只有100台左右。
8. 数据规模：由于Hive建立在集群上并可以利用MapReduce进行并行计算，因此可以支持很大规模的数据；对应的，数据库可以支持的数据规模较小

> *Hive中的数据类型*

Hive中主要包含两种数据类型：基本类型和复杂(集合)类型

1. 基本类型：tinyint，smallint，int，bigint，float，double，boolean，string，binary和timestamp
2. 复杂(集合)类型：array，map和struct

> *order by和sort by的区别*

区别如下

1. order by是对所有的数据进行整体的排序，无论设置了多少个ReduceTask，都只会产生一个结果文件
2. sort by是在每一个MapReduce内部对数据进行排序，常常结合distribute by来使用。当sort by和distribute by相同的时候，可以使用cluster by

> *Hive中的文件存储格式*

Hive中经常使用的文件格式有四种：textfile，sequencefile，orc，parquet

1. textFile格式：默认格式，数据不做压缩，磁盘开销大，数据解析开销大。可结合Gzip、Bzip2使用，但使用Gzip这种方式，hive不会对数据进行切分，从而无法对数据进行并行操作
2. orc格式：每个Orc文件由1个或多个stripe组成，每个stripe一般为HDFS的块大小，每一个stripe包含多条记录，这些记录按照列进行独立存储，对应到Parquet中的row group的概念。每个Stripe里有三部分组成，分别是Index Data，Row Data，Stripe Footer。每个文件有一个File Footer，这里面存的是每个Stripe的行数，每个Column的数据类型信息等；每个文件的尾部是一个PostScript，这里面记录了整个文件的压缩类型以及FileFooter的长度信息等。在读取文件时，会seek到文件尾部读PostScript，从里面解析到File Footer长度，再读FileFooter，从里面解析到各个Stripe信息，再读各个Stripe，即从后往前读
3. Parquet格式：以二进制方式存储，所以是不可以直接读取的，文件中包括该文件的数据和元数据，因此Parquet格式文件是自解析的。通常情况下，在存储Parquet数据的时候会按照Block大小设置行组的大小，由于一般情况下每一个Mapper任务处理数据的最小单位是一个Block，这样可以把每一个行组由一个Mapper任务处理，增大任务执行并行度。一个Parquet文件可以存储多个行组，文件的首位都是该文件的Magic Code，用于校验它是否是一个Parquet文件，Footer length记录了文件元数据的大小，通过该值和文件长度可以计算出元数据的偏移量，文件的元数据中包括每一个行组的元数据信息和该文件存储数据的Schema信息。除了文件中每一个行组的元数据，每一页的开始都会存储该页的元数据，在Parquet中，有三种类型的页：数据页、字典页和索引页。数据页用于存储当前行组中该列的值，字典页存储该列值的编码字典，每一个列块中最多包含一个字典页，索引页用来存储当前行组下该列的索引，目前Parquet中还不支持索引页

> *窗口函数*

Hive中窗口函数如下：

1. RANK() 排序相同时会重复，总数不会变
2. DENSE_RANK() 排序相同时会重复，总数会减少
3. ROW_NUMBER() 会根据顺序计算
4. OVER()：指定分析函数工作的数据窗口大小，这个数据窗口大小可能会随着行的变而变化
    a. CURRENT ROW：当前行
    b. n PRECEDING：往前n行数据
    c. n FOLLOWING：往后n行数据
    d. UNBOUNDED：起点，UNBOUNDED PRECEDING 表示从前面的起点，UNBOUNDED FOLLOWING表示到后面的终点
    e. LAG(col,n)：往前第n行数据
    f. LEAD(col,n)：往后第n行数据

g. NTILE(n)：把有序分区中的行分发到指定数据的组中，各个组有编号，编号从1开始，对于每一行，NTILE返回此行所属的组的编号。注意：n必须为int类型

> *窗口函数 常见优化方案如下*

1. MapJoin。如果不指定MapJoin或者不符合MapJoin的条件，那么Hive解析器会将Join操作转换成Common Join，即：在Reduce阶段完成join。容易发生数据倾斜。可以用MapJoin把小表全部加载到内存在map端进行join，避免reducer处理
2. 进行行列过滤
3. 对数据进行分区
4. 合理设置Map数。在数据结构相对复杂或者处理逻辑相对复杂的前提下，可以通过增加切片数量来增加MapTask的数量以提高执行效率
5. 对小文件进行合并
6. 开启map端combiner：`set hive.map.aggr=true;`
7. 开启JVM重用

> *动态分区 静态分区与动态分区的主要区别在于静态分区是手动指定，而动态分区是通过数据来进行判断。 详细来说，静态分区的列实在编译时期，通过用户传递来决定的；动态分区只有在 SQL 执行时才能决定；动态分区是基于查询参数的位置去推断分区的名称，从而建立分区*

> *Sqoop导入导出Null存储一致性问题Hive中的Null在底层是以`"\N"`来存储，而MySQL中的Null在底层就是Null，为了保证数据两端的一致性。在导出数据时采用--input-null-string和--input-null-non-string两个参数。导入数据时采用--null-string和--null-non-string。*

HBase

> *HBase的写流程*

流程如下

1. Client先访问zookeeper，获取hbase:meta表位于哪个Region Server
2. 访问对应的Region Server，获取hbase:meta表，根据读请求的namespace:table/rowkey，查询出目标数据位于哪个Region Server中的哪个Region中。并将该table的region信息以及meta表的位置信息缓存在客户端的meta cache，方便下次访问
3. 与目标Region Server进行通讯
4. 将数据顺序写入（追加）到WAL

5. 将数据写入对应的MemStore，数据会在MemStore进行排序
6. 向客户端发送ack
7. 等达到MemStore的刷写时机后，将数据刷写到HFile

> *HBase的读流程*

流程如下

1. Client先访问zookeeper，获取hbase:meta表位于哪个Region Server
2. 访问对应的Region Server，获取hbase:meta表，根据读请求的namespace:table/rowkey，查询出目标数据位于哪个Region Server中的哪个Region中。并将该table的region信息以及meta表的位置信息缓存在客户端的meta cache，方便下次访问
3. 与目标Region Server进行通讯
4. 分别在Block Cache（读缓存），MemStore和Store File（HFile）中查询目标数据，并将查到的所有数据进行合并。此处所有数据是指同一条数据的不同版本（time stamp）或者不同的类型（Put/Delete）
5. 将从文件中查询到的数据块（Block，HFile数据存储单元，默认大小为64KB）缓存到Block Cache
6. 将合并后的最终结果返回给客户端

> *HBase的优化*

常见优化方案如下

1. 允许在HDFS的文件中追加内容。开启HDFS追加同步，可以配合HBase的数据同步和持久化。通过属性`dfs.support.append`来调节，默认值为true
2. 优化DataNode允许的最大文件打开数。HBase一般都会同一时间操作大量的文件，实际过程中可以根据集群的数量和规模以及数据动作来确定该值。可以通过属性`dfs.datanode.max.transfer.threads`来修改，默认值为4096
3. 优化延迟高的数据操作的等待时间。如果对于某一次数据操作来讲，延迟非常高，socket需要等待更长的时间，建议把该值设置为更大的值，以确保socket不会被timeout掉。可以通过属性`dfs.image.transfer.timeout`来修改，默认值为60000ms
4. 设置RPC监听数量。指定RPC监听的数量，可以根据客户端的请求数进行调整，读写请求较多时，适当增加监听数。通过属性`hbase.regionserver.handler.count`来调节，默认值为30
5. 优化HStore文件大小。如果需要运行HBase的MapReduce任务，可以减小HStore的大小，因为一个HRegion对应一个Map任务，如果单个HRegion过大，会导致Map任务执行时间过长。通过属性`hbase.hregion.max.filesize`；来调节，默认值为10737418240B(10GB)
6. 优化HBase客户端缓存。指定Hbase客户端缓存，适当增大缓存大小可以减少RPC调用次数，但是会消耗更多内存，反之则会增加RPC次数，但是会减少内存的消

耗。通过属性`hbase.client.write.buffer`来指定
7. 指定scan.next扫描HBase所获取的行数。指定scan.next方法获取的默认行数，值越大，消耗内存越大。可以通过属性`hbase.client.scanner.caching`来调节

> *行键设计原则*

RowKey的设计原则如下

1. 要保证唯一性
2. 行键要尽量散列，可以考虑采用随机数、哈希值等方式
3. 行键要尽量定长
4. 行键要设计的有意义

Kafka

> *Kafka中的ISR、AR、HW、LEO表示什么意思*

分别表示

1. ISR：与leader保持同步的follower集合
2. AR：分区的所有副本
3. LEO：没个副本的最后条消息的offset
4. HW：一个分区中所有副本最小的offset

> *Kafka中如何体现消息顺序性*

每个分区内，每条消息都有一个offset，故只能保证分区内有序

> *那些情景会造成消息漏消费*

先提交offset，后消费，有可能造成数据的重复

> *当使用kafka-topics.sh创建（删除）了一个topic之后，Kafka背后会执行什么逻辑*

逻辑如下

1. 会在zookeeper中的/brokers/topics节点下创建一个新的topic节点，如：/brokers/topics/first
2. 触发Controller的监听程序
3. kafka Controller 负责topic的创建工作，并更新metadata cache

> *topic的分区数可不可以减少？如果可以怎么减少？如果不可以，那又是为什么*

不可以减少，被删除的分区数据难以处理

> *Kafka的ISR副本同步队列*

ISR(In-Sync Replicas)，副本同步队列。ISR中包括Leader和Follower。如果Leader进程挂掉，会在ISR队列中选择一个服务作为新的Leader。通过`replica.lag.max.messages`(延迟条数)和`replica.lag.time.max.ms`(延迟时间)两个参数决定一台服务是否可以加入ISR副本队列，在0.10版本移除了`replica.lag.max.messages`参数，防止服务频繁的进去队列。任意一个维度超过阈值都会把Follower剔除出ISR，存入OSR（Outof-Sync Replicas）列表，新加入的Follower也会先存放在OSR中

> *KafKa的分区分配策略*

在 Kafka内部存在两种默认的分区分配策略：Range和 RoundRobin其中默认策略是Range策略。Range是对每个Topic而言的（即一个Topic一个Topic分），首先对同一个Topic里面的分区按照序号进行排序，并对消费者按照字母顺序进行排序。然后用Partitions分区的个数除以消费者线程的总数来决定每个消费者线程消费几个分区。如果除不尽，那么前面几个消费者线程将会多消费一个分区

> *Kafka消息数据积压，Kafka消费能力不足怎么处理*

常见处理方案如下

1. 如果是Kafka消费能力不足，则可以考虑增加Topic的分区数，并且同时提升消费组的消费者数量，消费者数=分区数
2. 如果是下游的数据处理不及时：提高每批次拉取的数量。批次拉取数据过少(拉取数据/处理时间<生产速度)，使处理的数据小于生产的数据，也会造成数据积压

> *Kafka幂等性*

Producer的幂等性指的是当发送同一条消息时，数据在Server端只会被存储一次，数据不丢不重，但是这里的幂等性是有条件的：

1. 只能保证Producer在单个会话内不丢不重，如果Producer出现意外挂掉再重启是无法保证的(幂等性情况下，是无法获取之前的状态信息，因此是无法做到跨会话级别的不丢不重)
2. 幂等性不能跨多个Topic-Partition，只能保证单个Partition内的幂等性，当涉及多个 Topic-Partition时，这中间的状态并没有同步

> *Kafka事务*

Kafka从0.11版本开始引入了事务支持。事务可以保证Kafka在Exactly Once语义的基础上，生产和消费可以跨分区和会话，要么全部成功，要么全部失败

1. Producer事务：为了实现跨分区跨会话的事务，需要引入一个全局唯一的Transaction ID，并将Producer获得的PID和Transaction ID绑定。这样当Producer重启后就可以通过正在进行的Transaction ID获得原来的PID。 为了管理Transaction，Kafka引入了一个新的组件Transaction Coordinator。Producer就是通过和Transaction Coordinator交互获得Transaction ID对应的任务状态。Transaction Coordinator还负责将事务所有写入Kafka的一个内部Topic，这样即使整个服务重启，由于事务状态得到保存，进行中的事务状态可以得到恢复，从而继续进行
2. Consumer事务：上述事务机制主要是从Producer方面考虑，对于Consumer而言，事务的保证就会相对较弱，尤其时无法保证Commit的信息被精确消费。这是由于Consumer可以通过offset访问任意信息，而且不同的Segment File生命周期不同，同一事务的消息可能会出现重启后被删除的情况

> *Kafka过期数据清理*

在保证数据没有被引用的前提下，Kafka中的日志清理保存的策略只有delete和compact两种：

1. `log.cleanup.policy=delete`启用删除策略
2. `log.cleanup.policy=compact`启用压缩策略

Spark

> *Spark有几种部署方式？请分别简要论述*

Spark的部署方式如下

1. Local:运行在一台机器上，通常用于测试环境
2. Standalone:构建一个基于Master+Slaves的资源调度集群，Spark任务提交给Master运行。是Spark自身的一个调度系统
3. Yarn: Spark客户端直接连接Yarn，不需要额外构建Spark集群。有yarn-client和yarn-cluster两种模式，主要区别在于Driver程序的运行节点
4. Mesos：国内大环境比较少用

> *如何理解Spark中的血统概念(RDD)*

RDD在Lineage依赖方面分为两种Narrow Dependencies与Wide Dependencies用来解决数据容错时的高效性以及划分任务时候起到重要作用

> *描述Spark的宽窄依赖，以及Spark如何划分stage，每个stage又根据什么决定task个数*

答案如下

1. Stage：根据RDD之间的依赖关系的不同将Job划分成不同的Stage，遇到一个宽依赖则划分一个Stage
2. Task：Stage是一个TaskSet，将Stage根据分区数划分成一个个的Task

> *列举Spark的transformation算子，并简述功能*

答案如下

1. map(func)：返回一个新的RDD，该RDD由每一个输入元素经过func函数转换后组成
2. flatMap(func)：类似于map，但是每一个输入元素，会被映射为0个或多个输出元素，(因此，func函数的返回值是一个seq，而不是单一元素)
3. mapPartitions(func)：类似于map，但独立地在RDD的每一个分片上运行，因此在类型为T的RD上运行时，func的函数类型必须是Iterator[T] => Iterator[U]。假设有N个元素，有M个分区，那么map的函数的将被调用N次,而mapPartitions被调用M次，一个函数一次处理所有分区
4. reduceByKey(func，[numTask])：在一个(K,V)的RDD上调用，返回一个(K,V)的RDD，使用定的reduce函数，将相同key的值聚合到一起，reduce任务的个数可以通过第二个可选的参数来设置
5. filter(func)：返回一个新的数据集，由经过func函数处理后返回值为true的原元素组成
6. pipe(command,[envVars])：通过管道的方式对RDD的每个分区使用shell命令进行操作，返回对应的结果
7. union(otherDataSet)：返回一个新的数据集，由原数据集合参数联合而成
8. intersection(otherDataset)：求两个RDD的交集
9. distinct([numtasks])：返回一个包含源数据集中所有不重复元素的i新数据集
10. aggregateByKey (zeroValue:U,[partitioner: Partitioner]) (seqOp: (U, V) => U,combOp: (U, U) => U: 在kv对的RDD中，，按key将value进行分组合并，合并时，将每个value和初始值作为seq函数的参数，进行计算，返回的结果作为一个新的kv对，然后再将结果按照key进行合并，最后将每个分组的value传递给combine函数进行计算（先将前两个value进行计算，将返回结果和下一个value传给combine函数，以此类推），将key与计算结果作为一个新的kv对输出
11. combineByKey(createCombiner: V=>C, mergeValue: (C, V) =>C, mergeCombiners: (C, C) =>C): 对相同K，把V合并成一个集合

12. createCombiner: combineByKey() 会遍历分区中的所有元素，因此每个元素的键要么还没有遇到过，要么就和之前的某个元素的键相同。如果这是一个新的元素,combineByKey()会使用一个叫作createCombiner()的函数来创建那个键对应的累加器的初始值
13. mergeValue: 如果这是一个在处理当前分区之前已经遇到的键，它会使用mergeValue()方法将该键的累加器对应的当前值与这个新的值进行合并
14. mergeCombiners: 由于每个分区都是独立处理的， 因此对于同一个键可以有多个累加器。如果有两个或者更多的分区都有对应同一个键的累加器， 就需要使用用户提供的 mergeCombiners() 方法将各个分区的结果进行合并
15. groupByKey([numtasks]) ：在一个由(K,v)对组成的数据集上调用，返回一个(K,Seq[V])对组成的数据集。默认情况下，输出结果的并行度依赖于父RDD的分区数目，如果想要对key进行聚合的话，使用reduceByKey或者combineByKey会有更好的性能
16. reduceByKey(func,[numTasks]) ：在一个(K,V)对的数据集上使用，返回一个(K,V)对的数据集，key相同的值，都被使用指定的reduce函数聚合到一起，reduce任务的个数是可以通过第二个可选参数来配置的 。推荐使用
17. sortByKey([ascending],[numTasks]) ：在类型为(K,V)的数据集上调用，返回以K为键进行排序的(K,V)对数据集，升序或者降序有boolean型的ascending参数决定
18. join(otherDataset,[numTasks]) ：在类型为(K,V)和(K,W)类型的数据集上调用，返回一个(K,(V,W))对，每个key中的所有元素都在一起的数据集
19. cogroup(otherDataset,[numTasks]) ：在类型为(K,V)和(K,W)类型的数据集上调用，返回一个数据集，组成元素为(K,Iterable[V],Iterable[W]) tuples
20. cartesian(otherDataset) ：笛卡尔积，但在数据集T和U上调用时，返回一个(T,U)对的数据集，所有元素交互进行笛卡尔积
21. coalesce(numPartitions) ：对RDD中的分区减少指定的数目，通常在过滤完一个大的数据集之后进行此操作
22. repartition(numpartitions) ：将RDD中所有records平均划分到numparitions个partition中

根据自身情况选择比较熟悉的算子加以介绍

> *列举Spark的action算子，并简述功能*

常见action算子如下

1. reduce(func) ：通过函数func聚集数据集中的所有元素，这个函数必须是关联性的，确保可以被正确的并发执行
2. collect() ：在driver的程序中，以数组的形式，返回数据集的所有元素，这通常会在使用filter或者其它操作后，返回一个足够小的数据子集再使用
3. count() ：返回数据集的元素个数
4. first() ：返回数据集的第一个元素(类似于take(1))
5. take(n) ：返回一个数组，由数据集的前n个元素组成。注意此操作目前并非并行执行的，而是driver程序所在机器

6. takeSample(withReplacement,num,seed)：返回一个数组，在数据集中随机采样num个元素组成，可以选择是否用随机数替换不足的部分，seed用于指定的随机数生成器种子
7. saveAsTextFile(path)：将数据集的元素，以textfile的形式保存到本地文件系统hdfs或者任何其他Hadoop支持的文件系统，spark将会调用每个元素的toString方法，并将它转换为文件中的一行文本
8. takeOrderd(n,[ordering])：排序后的limit(n)
9. saveAsSequenceFile(path)：将数据集的元素，以sequencefile的格式保存到指定的目录下，本地系统，hdfs或者任何其他hadoop支持的文件系统，RDD的元素必须由key-value对组成。并都实现了hadoop的writable接口或隐式可以转换为writable
10. saveAsObjectFile(path)：使用Java的序列化方法保存到本地文件，可以被sparkContext.objectFile()加载
11. countByKey()：对(K,V)类型的RDD有效，返回一个(K,Int)对的map，表示每一个可以对应的元素个数
12. foreache(func)：在数据集的每一个元素上，运行函数func,t通常用于更新一个累加器变量，或者和外部存储系统做交互

根据自身情况选择比较熟悉的算子加以介绍

> *Spark常用算子reduceByKey与groupByKey的区别，哪一种更具优势*

reduceByKey：按照key进行聚合，在shuffle之前有combine(预聚合)操作，返回结果是RDD[k,v]；groupByKey：按照key进行分组，直接进行shuffle

> *Repartition和Coalesce关系与区别*

二者的关系与区别如下

1. 关系：两者都是用来改变RDD的partition数量的，repartition底层调用的就是coalesce方法：coalesce(numPartitions, shuffle = true)
2. 区别：repartition一定会发生shuffle，coalesce根据传入的参数来判断是否发生shuffle。一般情况下增大rdd的partition数量使用repartition，减少partition数量时使用coalesce。

> *分别简述Spark中的缓存机制（cache和persist）与checkpoint机制*

答案如下

1. cache:内存，不会截断血缘关系，使用计算过程中的数据缓存
2. checkpoint：磁盘，截断血缘关系，在ck之前必须没有任何任务提交才会生效，ck过程会额外提交一次任务

> *简述Spark中共享变量(广播变量和累加器)的基本原理与用途*

累加器（accumulator）是Spark中提供的一种分布式的变量机制，其原理类似于mapreduce，即分布式的改变，然后聚合这些改变。累加器的一个常见用途是在调试时对作业执行过程中的事件进行计数。而广播变量用来高效分发较大的对象。

共享变量出现的原因：通常在向 Spark 传递函数时，比如使用 map() 函数或者用 filter() 传条件时，可以使用驱动器程序中定义的变量，但是集群中运行的每个任务都会得到这些变量的一份新的副本，更新这些副本的值也不会影响驱动器中的对应变量。Spark的两个共享变量，累加器与广播变量，分别为结果聚合与广播这两种常见的通信模式突破了这一限制

> *当Spark涉及到数据库的操作时，如何减少Spark运行中的数据库连接数*

使用foreachPartition代替foreach，在foreachPartition内获取数据库的连接

> *简述SparkSQL中RDD、DataFrame、DataSet三者的区别与联系*

答案如下

1. RDD
    a. 优点:
        i. 编译时类型安全
        ii. 编译时就能检查出类型错误
        iii. 面向对象的编程风格
        iv. 直接通过类名点的方式来操作数据
    b. 缺点
        i. 序列化和反序列化的性能开销
        ii. 无论是集群间的通信, 还是IO操作都需要对对象的结构和数据进行序列化和反序列化
        iii. GC的性能开销，频繁的创建和销毁对象, 势必会增加GC
2. DataFrame
    a. DataFrame引入了schema和off-heap
    b. schema : RDD每一行的数据, 结构都是一样的，这个结构就存储在schema中。 Spark通过schema就能够读懂数据, 因此在通信和IO时就只需要序列化和反序列化数据, 而结构的部分就可以省略了
3. DataSet
    a. DataSet结合了RDD和DataFrame的优点，并带来的一个新的概念Encoder
    b. 当序列化数据时，Encoder产生字节码与off-heap进行交互，能够达到按需访问数据的效果，而不用反序列化整个对象。Spark还没有提供自定义Encoder的API，但是未来会加入

> *append和overwrite的区别*

append在原有分区上进行追加，overwrite在原有分区上进行全量刷新

> *coalesce和repartition的区别*

coalesce和repartition都用于改变分区，coalesce用于缩小分区且不会进行shuffle，repartition用于增大分区（提供并行度）会进行shuffle,在spark中减少文件个数会使用coalesce来减少分区来到这个目的。但是如果数据量过大，分区数过少会出现OOM所以coalesce缩小分区个数也需合理

> *cache缓存级别*

DataFrame的cache默认采用 MEMORY_AND_DISK 这和RDD 的默认方式不一样RDD cache默认采用MEMORY_ONLY

> *Spark Shuffle默认并行度*

由参数`spark.sql.shuffle.partitions`决定， 默认值为200

> *kryo序列化*

kryo序列化比java序列化更快更紧凑，但spark默认的序列化是java序列化并不是spark序列化，因为spark并不支持所有序列化类型，而且每次使用都必须进行注册。注册只针对于RDD。在DataFrames和DataSet当中自动实现了kryo序列化

> *创建临时表和全局临时表*

DataFrame.createTempView() 创建普通临时表

DataFrame.createGlobalTempView() DataFrame.createOrReplaceTempView() 创建全局临时表

> *BroadCast join 广播join*

原理：先将小表数据查询出来聚合到driver端，再广播到各个executor端，使得表与表join时是在进行本地join，避免进行网络传输产生shuffle

使用场景：大表join小表 只能广播小表

> *控制Spark reduce缓存 调优shuffle*

可以通过参数`spark.reducer.maxSizeInFilght`来调节。此参数为ReduceTask能够拉取的数据量大小。默认48MB，当集群资源足够时，可以考虑增大这个值来减少reduce拉取数据量的次数，从而达到优化shuffle的效果，一般调大为96MB

参数`spark.shuffle.file.buffer`表示为每个shuffle文件输出流的内存缓冲区大小，适当的调大这个参数的值可以减少在创建shuffle文件时进行磁盘搜索和系统调用的次数，默认参数为32k 一般调大为64k

> *注册UDF函数*

SparkSession.udf.register 方法进行注册

> *SparkSQL中join操作与left join操作的区别*

join和sql中的inner join操作很相似，返回结果是前面一个集合和后面一个集合中匹配成功的，过滤掉关联不上的。

left join类似于SQL中的左外关联left outer join，返回结果以第一个RDD为主，关联不上的记录为空，部分场景下可以使用left semi join替代left join

因为 left semi join 是 in(keySet) 的关系，遇到右表重复记录，左表会跳过,性能更高，而 left join 则会一直遍历。但是left semi join 中最后 select 的结果中只许出现左表中的列名，因为右表只有 join key 参与关联计算了

> *Spark Streaming如何做到精准一次消费*

方案如下：

1. 手动维护偏移量
2. 处理完业务数据后，再进行提交偏移量操作
3. 极端情况下，如在提交偏移量时断网或停电会造成spark程序第二次启动时重复消费问题，所以在涉及到金额或精确性非常高的场景会使用事物保证精准一次消费

> *Spark Streaming中怎么控制每秒消费数据的速度*

通过spark.streaming.kafka.maxRatePerPartition参数来设置Spark Streaming从kafka分区每秒拉取的条数

> *Spark Streaming背压机制*

把spark.streaming.backpressure.enabled 参数设置为ture,开启背压机制后Spark Streaming会根据延迟动态去kafka消费数据,上限由spark.streaming.kafka.maxRatePerPartition参数控制，所以两个参数一般会一起使用

> *Spark Streaming 默认分区个数*

Spark Streaming默认分区个数与所对接的kafka topic分区个数一致，Spark Streaming里一般不会使用repartition算子增大分区，因为repartition会进行shuffle增加耗时

> *SparkStreaming有哪几种方式消费Kafka中的数据，它们之间的区别是什么*

1. 基于Receiver的方式：这种方式使用Receiver来获取数据。Receiver是使用Kafka的高层次Consumer API来实现的。receiver从Kafka中获取的数据都是存储在Spark Executor的内存中的（如果突然数据暴增，大量batch堆积，很容易出现内存溢出的问题），然后Spark Streaming启动的job会去处理那些数据。 然而，在默认的配置下，这种方式可能会因为底层的失败而丢失数据。如果要启用高可靠机制，让数据零丢失，就必须启用Spark Streaming的预写日志机制（Write Ahead Log，WAL）。该机制会同步地将接收到的Kafka数据写入分布式文件系统（比如HDFS）上的预写日志中。所以，即使底层节点出现了失败，也可以使用预写日志中的数据进行恢复
2. 基于Direct的方式：这种新的不基于Receiver的直接方式，是在Spark 1.3中引入的，从而能够确保更加健壮的机制。替代掉使用Receiver来接收数据后，这种方式会周期性地查询Kafka，来获得每个topic+partition的最新的offset，从而定义每个batch的offset的范围。当处理数据的job启动时，就会使用Kafka的简单consumer api来获取Kafka指定offset范围的数据。 优点是：
    a. 简化并行读取：如果要读取多个partition，不需要创建多个输入DStream然后对它们进行union操作。Spark会创建跟Kafka partition一样多的RDD partition，并且会并行从Kafka中读取数据。所以在Kafka partition和RDD partition之间，有一个一对一的映射关系
    b. 高性能：如果要保证零数据丢失，在基于receiver的方式中，需要开启WAL机制。这种方式其实效率低下，因为数据实际上被复制了两份，Kafka自己本身就有高可靠的机制，会对数据复制一份，而这里又会复制一份到WAL中。而基于direct的方式，不依赖Receiver，不需要开启WAL机制，只要Kafka中作了数据的复制，那么就可以通过Kafka的副本进行恢复
    c. 一次且仅一次的事务机制
3. 对比
    a. 基于receiver的方式，是使用Kafka的高阶API来在ZooKeeper中保存消费过的offset的。这是消费Kafka数据的传统方式。这种方式配合着WAL机制可以保证数据零丢失的高可靠性，但是却无法保证数据被处理一次且仅一次，可能会处理两次。因为Spark和ZooKeeper之间可能是不同步的
    b. 基于direct的方式，使用kafka的简单api，Spark Streaming自己就负责追踪消费的offset，并保存在checkpoint中。Spark自己一定是同步的，因此可以保证数据是消费一次且仅消费一次
    c. 在实际生产环境中大都用Direct方式

> *简述SparkStreaming窗口函数的原理*

窗口函数就是在原来定义的SparkStreaming计算批次大小的基础上再次进行封装，每次计算多个批次的数据，同时还需要传递一个滑动步长的参数，用来设置当次计算任务完成之后下一次从什么地方开始计算

> *Spark中的数据倾斜*

Spark中的数据倾斜，包括Spark Streaming和Spark Sql，表现主要有下面几种：

1. Executor lost，OOM，Shuffle过程出错
2. Driver OOM；
3. 单个Executor执行时间特别久，整体任务卡在某个阶段不能结束；
4. 正常运行的任务突然失败；

> *什么是Flink*

Flink 是一个框架和分布式处理引擎，用于对无界和有界数据流进行有状态计算。并且 Flink 提供了数据分布、容错机制以及资源管理等核心功能。Flink提供了诸多高抽象层的API以便用户编写分布式任务：

1. DataSet API， 对静态数据进行批处理操作，将静态数据抽象成分布式的数据集，用户可以方便地使用Flink提供的各种操作符对分布式数据集进行处理，支持Java、Scala和Python
2. DataStream API，对数据流进行流处理操作，将流式的数据抽象成分布式的数据流，用户可以方便地对分布式数据流进行各种操作，支持Java和Scala
3. Table API，对结构化数据进行查询操作，将结构化数据抽象成关系表，并通过类SQL的DSL对关系表进行各种查询操作，支持Java和Scala
4. 此外，Flink 还针对特定的应用领域提供了领域库，例如： Flink ML，Flink 的机器学习库，提供了机器学习Pipelines API并实现了多种机器学习算法。 Gelly，Flink 的图计算库，提供了图计算的相关API及多种图计算算法实现

> *Flink和Spark Streaming的区别*

两个框架的不同点非常之多。但是在面试时有非常重要的一点一定要回答出来：Flink 是标准的实时处理引擎，基于事件驱动。而 Spark Streaming 是微批（Micro-Batch）的模型。两个框架的主要区别：

1. 架构模型Spark Streaming 在运行时的主要角色包括：Master、Worker、Driver、Executor，Flink 在运行时主要包含：Jobmanager、Taskmanager和Slot
2. 任务调度Spark Streaming 连续不断的生成微小的数据批次，构建有向无环图DAG，Spark Streaming 会依次创建 DStreamGraph、JobGenerator、JobScheduler。Flink 根据用户提交的代码生成 StreamGraph，经过优化生成

JobGraph，然后提交给 JobManager进行处理，JobManager 会根据 JobGraph 生成 ExecutionGraph，ExecutionGraph 是 Flink 调度最核心的数据结构，JobManager 根据 ExecutionGraph 对 Job 进行调度

3. 时间机制Spark Streaming 支持的时间机制有限，只支持处理时间。 Flink 支持了流处理程序在时间上的三个定义：处理时间、事件时间、注入时间。同时也支持 watermark 机制来处理滞后数据
4. 容错机制对于 Spark Streaming 任务，我们可以设置 checkpoint，然后假如发生故障并重启，我们可以从上次 checkpoint 之处恢复，但是这个行为只能使得数据不丢失，可能会重复处理，不能做到恰好一次处理语义。Flink 则使用两阶段提交协议来解决这个问题

> *Flink的组件栈*

根据 Flink 官网描述，Flink 是一个分层架构的系统，每一层所包含的组件都提供了特定的抽象，用来服务于上层组件

自下而上，每一层分别代表：Deploy 层：该层主要涉及了Flink的部署模式，在上图中我们可以看出，Flink 支持包括local、Standalone、Cluster、Cloud等多种部署模式。Runtime 层：Runtime层提供了支持 Flink 计算的核心实现，比如：支持分布式 Stream 处理、JobGraph到ExecutionGraph的映射、调度等等，为上层API层提供基础服务。API层：API 层主要实现了面向流（Stream）处理和批（Batch）处理API，其中面向流处理对应 DataStream API，面向批处理对应DataSet API，后续版本，Flink有计划将DataStream和 DataSet API进行统一。Libraries层：该层称为Flink应用框架层，根据API层的划分，在API 层之上构建的满足特定应用的实现计算框架，也分别对应于面向流处理和面向批处理两类。面向流处理支持：CEP（复杂事件处理）、基于SQL-like的操作（基于Table的关系操作）；面向批处理支持：FlinkML（机器学习库）、Gelly（图处理）

> *Flink 的运行必须依赖Hadoop组件吗*

Flink可以完全独立于Hadoop，在不依赖Hadoop组件下运行。但是做为大数据的基础设施，Hadoop体系是任何大数据框架都绕不过去的。Flink可以集成众多Hadooop 组件，例如 Yarn、Hbase、HDFS等等。例如，Flink可以和Yarn集成做资源调度，也可以读写HDFS，或者利用HDFS做检查点

> *Flink的基础编程模型*

Flink 程序的基本构建是数据输入来自一个 Source，Source 代表数据的输入端，经过 Transformation 进行转换，然后在一个或者多个Sink接收器中结束。数据流（stream）就是一组永远不会停止的数据记录流，而转换（transformation）是将一个或多个流作为输入，并生成一个或多个输出流的操作。执行时，Flink程序映射到 streaming dataflows，由流（streams）和转换操作（transformation operators）组成

> *Flink集群有哪些角色？各自有什么作用？*

Flink 程序在运行时主要有 TaskManager，JobManager，Client三种角色。其中JobManager扮演着集群中的管理者Master的角色，它是整个集群的协调者，负责接收Flink Job，协调检查点，Failover 故障恢复等，同时管理Flink集群中从节点TaskManager。TaskManager是实际负责执行计算的Worker，在其上执行Flink Job的一组Task，每个TaskManager负责管理其所在节点上的资源信息，如内存、磁盘、网络，在启动的时候将资源的状态向JobManager汇报。Client是Flink程序提交的客户端，当用户提交一个Flink程序时，会首先创建一个Client，该Client首先会对用户提交的Flink程序进行预处理，并提交到Flink集群中处理，所以Client需要从用户提交的Flink程序配置中获取JobManager的地址，并建立到JobManager的连接，将Flink Job提交给JobManager

> *Flink 资源管理中 Task Slot 的概念*

在Flink架构角色中我们提到，TaskManager是实际负责执行计算的Worker，TaskManager是一个 JVM 进程，并会以独立的线程来执行一个task或多个subtask。为了控制一个TaskManager 能接受多少个 task，Flink 提出了 Task Slot 的概念。简单的说，TaskManager会将自己节点上管理的资源分为不同的Slot：固定大小的资源子集。这样就避免了不同Job的Task互相竞争内存资源，但是需要主要的是，Slot只会做内存的隔离。没有做CPU的隔离

> *说说 Flink 的常用算子*

常用算子举例如下：

1. map：DataStream --> DataStream：输入一个参数产生一个参数，map的功能是对输入的参数进行转换操作2
2. flatMap：DataStream --> DataStream：输入一个参数，产生0、1或者多个输出，这个多用于拆分操作
3. filter：DataStream --> DataStream：结算每个元素的布尔值，并返回为true的元素
4. keyBy：DataSteam --> DataStream：逻辑地将一个流拆分成不相交的分区，每个分区包含具有相同key的元素，在内部以hash的形式实现的。以key来分组
5. reduce：KeyedStream --> DataStream：滚动和并操作，合并当前元素和上一次合并的元素结果
6. fold：KeyedStream --> DataStream：用一个初始的一个值，与其每个元素进行滚动合并操作
7. aggregation：KeyedStream --> DataStream：分组流数据的滚动聚合操作：min和minBy的区别是min返回的是一个最小值，而minBy返回的是其字段中包含的最小值的元素（同样元原理适用于max和maxBy）
8. window：KeyedStream --> DataStream：windows是在一个分区的KeyedStreams中定义的，windows根据某些特性将每个key的数据进行分组（例如：在5s内到达的数据）
9. windowAll：DataStream --> AllWindowedStream：Windows可以在一个常规的DataStream中定义，Windows根据某些特性对所有的流（例如：5s内到达的数

据）

10. window apply：WindowedStream --> DataStream，AllWindowedStream --> DataStream：将一个通用的函数作为一个整体传递给window
11. window reduce：WindowedStream --> DataStream：给窗口赋予一个reduce的功能，并返回一个reduce的结果
12. window fold：WindowedStream --> DataStream：给窗口赋予一个fold的功能，并返回一个fold后的结果
13. aggregation on windows：WindowedStream --> DataStream：对window的元素做聚合操作，min和minBy的区别是min返回的是最小值，而minBy返回的是包含最小值字段的元素。（同样原理适用于max和maxBy）
14. union：DataStream --> DataStream：对两个或两个以上的DataStream做union操作，产生一个包含所有的DataStream元素的新DataStream。。注意：如果将一个DataStream和自己做union操作，在新的DataStream中，将看到每个元素重复两次
15. window join：DataStream，DataStream --> DataStream：根据给定的key和window对两个DataStream做join操作
16. window coGroup：DataStream，DataStream --> DataStream：根据一个给定的key和window对两个DataStream做CoGroups操作
17. connect：DataStream，DataStream --> ConnectedStreams：连接两个保持她们类型的数据流
18. coMap、coFlatMap：ConnectedStreams --> DataStream：作用于connected数据流上，功能与map和flatMap一样
19. split：DataStream --> SplitStream：根据某些特征把一个DataStream拆分成两个或多个DataStream
20. select：SplitStream --> DataStream：从一个SplitStream中获取一个或多个DataStream
21. iterate：DataStream --> IterativeStream --> DataStream：在流程中创建一个反馈循环，将一个操作的输出重定向到之前的操作，这对于定义持续更新模型的算法来说很有意义的
22. extract timestamps：DataStream --> DataStream：提取记录中的时间戳来跟需要事件时间的window一起发挥作用

> *说说Flink分区策略*

分区策略是用来决定数据如何发送至下游。目前 Flink 支持了8中分区策略的实现：

1. GlobalPartitioner： 数据会被分发到下游算子的第一个实例中进行处理
2. ShufflePartitioner ：数据会被随机分发到下游算子的每一个实例中进行
3. RebalancePartitioner： 数据会被循环发送到下游的每一个实例中进行处理
4. RescalePartitioner ：这种分区器会根据上下游算子的并行度，循环的方式输出到下游算子的每个实例。这里有点难以理解，假设上游并行度为 2，编号为 A 和 B。下游并行度为 4，编号为 1，2，3，4。那么 A 则把数据循环发送给 1 和 2，B 则把

数据循环发送给 3 和 4。假设上游并行度为 4，编号为 A，B，C，D。下游并行度为 2，编号为 1，2。那么 A 和 B 则把数据发送给 1，C 和 D 则把数据发送给 2
5. BroadcastPartitioner ：广播分区会将上游数据输出到下游算子的每个实例中。适合于大数据集和小数据集做Jion的场景
6. ForwardPartitioner：用于将记录输出到下游本地的算子实例。它要求上下游算子并行度一样。简单的说，ForwardPartitioner用来做数据的控制台打印
7. KeyGroupStreamPartitioner ：Hash 分区器。会将数据按Key的Hash值输出到下游算子实例中
8. CustomPartitionerWrapper：用户自定义分区器。需要用户自己实现 Partitioner 接口，来定义自己的分区逻辑

> *Flink的并行度设置是怎样的*

Flink中的任务被分为多个并行任务来执行，其中每个并行的实例处理一部分数据。这些并行实例的数量被称为并行度。我们在实际生产环境中可以从四个不同层面设置并行度：

1. 操作算子层面(Operator Level)
2. 执行环境层面(Execution Environment Level)
3. 客户端层面(Client Level)
4. 系统层面(System Level)

需要注意的优先级：算子层面>环境层面>客户端层面>系统层面。

> *Flink的Slot和parallelism有什么区别*

slot是指taskmanager的并发执行能力，假设我们将 taskmanager.numberOfTaskSlots 配置为3 那么每一个 taskmanager 中分配3个 TaskSlot, 3个 taskmanager 一共有9个TaskSlot

parallelism是指taskmanager实际使用的并发能力。假设我们把 parallelism.default 设置为1，那么9个 TaskSlot 只能用1个，有8个空闲

> *Flink的重启策略*

Flink 实现了多种重启策略:

1. 固定延迟重启策略（Fixed Delay Restart Strategy）
2. 故障率重启策略（Failure Rate Restart Strategy）
3. 没有重启策略（No Restart Strategy）
4. Fallback重启策略（Fallback Restart Strategy）

> *Flink中的分布式缓存*

Flink实现的分布式缓存和Hadoop有异曲同工之妙。目的是在本地读取文件，并把他放在taskmanager 节点中，防止task重复拉取

> *Flink中的广播变量，使用时需要注意什么*

我们知道Flink是并行的，计算过程可能不在一个 Slot 中进行，那么有一种情况即：当我们需要访问同一份数据。那么Flink中的广播变量就是为了解决这种情况。我们可以把广播变量理解为是一个公共的共享变量，我们可以把一个dataset 数据集广播出去，然后不同的task在节点上都能够获取到，这个数据在每个节点上只会存在一份

> *Flink中的窗口*

Flink 支持两种划分窗口的方式，按照time和count。如果根据时间划分窗口，那么它就是一个time-window 如果根据数据划分窗口，那么它就是一个count-window。flink支持窗口的两个重要属性（size和interval）如果size=interval,那么就会形成tumbling-window(无重叠数据) 如果size>interval,那么就会形成sliding-window(有重叠数据) 如果size< interval, 那么这种窗口将会丢失数据。比如每5秒钟，统计过去3秒的通过路口汽车的数据，将会漏掉2秒钟的数据。通过组合可以得出四种基本窗口：

1. time-tumbling-window 无重叠数据的时间窗口，设置方式举例：timeWindow(Time.seconds(5))
2. time-sliding-window 有重叠数据的时间窗口，设置方式举例：timeWindow(Time.seconds(5), Time.seconds(3))
3. count-tumbling-window无重叠数据的数量窗口，设置方式举例：countWindow(5)
4. count-sliding-window 有重叠数据的数量窗口，设置方式举例：countWindow(5,3)

> *Flink中的状态存储*

Flink在做计算的过程中经常需要存储中间状态，来避免数据丢失和状态恢复。选择的状态存储策略不同，会影响状态持久化如何和 checkpoint 交互。Flink提供了三种状态存储方式：MemoryStateBackend、FsStateBackend、RocksDBStateBackend

> *Flink中的时间有哪几类*

Flink 中的时间和其他流式计算系统的时间一样分为三类：事件时间，摄入时间，处理时间三种。如果以 EventTime 为基准来定义时间窗口将形成EventTimeWindow,要求消息本身就应该携带EventTime。如果以 IngesingtTime 为基准来定义时间窗口将形成IngestingTimeWindow,以 source 的systemTime为准。如果以 ProcessingTime 基准来定义时间窗口将形成 ProcessingTimeWindow，以 operator 的systemTime 为准

> *Flink 中水印是什么概念，起到什么作用*

Watermark 是 Apache Flink 为了处理 EventTime 窗口计算提出的一种机制, 本质上是一种时间戳。 一般来讲Watermark经常和Window一起被用来处理乱序事件

> *Flink Table & SQL 熟悉吗？TableEnvironment这个类有什么作用*

TableEnvironment是Table API和SQL集成的核心概念。这个类主要用来：

1. 在内部catalog中注册表
2. 注册外部catalog
3. 执行SQL查询
4. 注册用户定义（标量，表或聚合）函数
5. 将DataStream或DataSet转换为表
6. 持有对ExecutionEnvironment或StreamExecutionEnvironment的引用

> *Flink SQL的实现原理是什么？ 是如何实现 SQL 解析的呢？*

Flink 的SQL解析是基于Apache Calcite这个开源框架。基于此，一次完整的SQL解析过程如下：

1. 用户使用对外提供Stream SQL的语法开发业务应用
2. 用calcite对StreamSQL进行语法检验，语法检验通过后，转换成calcite的逻辑树节点；最终形成calcite的逻辑计划
3. 采用Flink自定义的优化规则和calcite火山模型、启发式模型共同对逻辑树进行优化，生成最优的Flink物理计划
4. 对物理计划采用janino codegen生成代码，生成用低阶API DataStream 描述的流应用，提交到Flink平台执行

> *Flink是如何做到高效的数据交换的*

在一个Flink Job中，数据需要在不同的task中进行交换，整个数据交换是有 TaskManager 负责的，TaskManager 的网络组件首先从缓冲buffer中收集records，然后再发送。Records 并不是一个一个被发送的，二是积累一个批次再发送，batch 技术可以更加高效的利用网络资源

> *Flink是如何做容错的*

Flink 实现容错主要靠强大的CheckPoint机制和State机制。Checkpoint 负责定时制作分布式快照、对程序中的状态进行备份；State 用来存储计算过程中的中间状态

> *Flink 分布式快照的原理*

Flink的分布式快照是根据Chandy-Lamport算法量身定做的。简单来说就是持续创建分布式数据流及其状态的一致快照，核心思想是在 input source 端插入 barrier，控制 barrier 的同步来实现 snapshot 的备份和 exactly-once 语义

> *Flink是如何保证Exactly-once语义的*

Flink通过实现两阶段提交和状态保存来实现端到端的一致性语义。 分为以下几个步骤：

1. 开始事务（beginTransaction）创建一个临时文件夹，来写把数据写入到这个文件夹里面
2. 预提交（preCommit）将内存中缓存的数据写入文件并关闭
3. 正式提交（commit）将之前写完的临时文件放入目标目录下。这代表着最终的数据会有一些延迟
4. 丢弃（abort）丢弃临时文件
5. 若失败发生在预提交成功后，正式提交前。可以根据状态来提交预提交的数据，也可删除预提交的数据

> *Flink 的 kafka 连接器有什么特别的地方*

Flink源码中有一个独立的connector模块，所有的其他connector都依赖于此模块，Flink 在1.9版本发布的全新kafka连接器，摒弃了之前连接不同版本的kafka集群需要依赖不同版本的connector这种做法，只需要依赖一个connector即可

> *Flink的内存管理*

Flink 并不是将大量对象存在堆上，而是将对象都序列化到一个预分配的内存块上。此外，Flink大量的使用了堆外内存。如果需要处理的数据超出了内存限制，则会将部分数据存储到硬盘上。Flink 为了直接操作二进制数据实现了自己的序列化框架。理论上Flink的内存管理分为三部分：

1. Network Buffers：这个是在TaskManager启动的时候分配的，这是一组用于缓存网络数据的内存，每个块是32K，默认分配2048个，可以通过`taskmanager.network.numberOfBuffers`修改
2. Memory Manage pool：大量的Memory Segment块，用于运行时的算法（Sort/Join/Shuffle等），这部分启动的时候就会分配。下面这段代码，根据配置文件中的各种参数来计算内存的分配方法。（heap or off-heap，这个放到下节谈），内存的分配支持预分配和lazy load，默认懒加载的方式
3. User Code，这部分是除了Memory Manager之外的内存用于User code和TaskManager本身的数据结构

> *Flink的序列化*

Java本身自带的序列化和反序列化的功能，但是辅助信息占用空间比较大，在序列化对象时记录了过多的类信息。Apache Flink摒弃了Java原生的序列化方法，以独特的方式处理数据类型和序列化，包含自己的类型描述符，泛型类型提取和类型序列化框架。
TypeInformation 是所有类型描述符的基类。它揭示了该类型的一些基本属性，并且可以生成序列化器。

ypeInformation 支持以下几种类型：

1. BasicTypeInfo: 任意Java 基本类型或 String 类型
2. BasicArrayTypeInfo: 任意Java基本类型数组或 String 数组
3. WritableTypeInfo: 任意 Hadoop Writable 接口的实现类
4. TupleTypeInfo: 任意的 Flink Tuple 类型(支持Tuple1 to Tuple25)。Flink tuples 是固定长度固定类型的Java Tuple实现
5. CaseClassTypeInfo: 任意的 Scala CaseClass(包括 Scala tuples)
6. PojoTypeInfo: 任意的 POJO (Java or Scala)，例如，Java对象的所有成员变量，要么是 public 修饰符定义，要么有 getter/setter 方法
7. GenericTypeInfo: 任意无法匹配之前几种类型的类

针对前六种类型数据集，Flink皆可以自动生成对应的TypeSerializer，能非常高效地对数据集进行序列化和反序列化

> *Flink中的Window出现了数据倾斜怎么解决*

window产生数据倾斜指的是数据在不同的窗口内堆积的数据量相差过多。本质上产生这种情况的原因是数据源头发送的数据量速度不同导致的。出现这种情况一般通过两种方式来解决：

1. 在数据进入窗口前做预聚合
2. 重新设计窗口聚合的key

> *Flink任务延迟高的优化*

在Flink的后台任务管理中，我们可以看到Flink的哪个算子和task出现了反压。最主要的手段是资源调优和算子调优。资源调优即是对作业中的Operator的并发数（parallelism）、CPU（core）、堆内存（heap_memory）等参数进行调优。作业参数调优包括：并行度的设置，State的设置，checkpoint的设置

> *Flink是如何处理反压的*

Flink 内部是基于 producer-consumer 模型来进行消息传递的，Flink的反压设计也是基于这个模型。Flink 使用了高效有界的分布式阻塞队列，就像 Java 通用的阻塞队列（BlockingQueue）一样。下游消费者消费变慢，上游就会受到阻塞

> *Operator Chains（算子链）*

为了更高效地分布式执行，Flink会尽可能地将operator的subtask链接（chain）在一起形成task。每个task在一个线程中执行。将operators链接成task是非常有效的优化：它能减少线程之间的切换，减少消息的序列化/反序列化，减少数据在缓冲区的交换，减少了延迟的同时提高整体的吞吐量。这就是我们所说的算子链

> *Flink什么情况下才会把Operator chain在一起形成算子链*

两个operator chain在一起的的条件：

1. 上下游的并行度一致
2. 下游节点的入度为1 （也就是说下游节点没有来自其他节点的输入）
3. 上下游节点都在同一个 slot group 中（下面会解释 slot group）
4. 下游节点的 chain 策略为 ALWAYS（可以与上下游链接，map、flatmap、filter等默认是ALWAYS）
5. 上游节点的 chain 策略为 ALWAYS 或 HEAD（只能与下游链接，不能与上游链接，Source默认是HEAD）
6. 两个节点间数据分区方式是 forward（参考理解数据流的分区）
7. 用户没有禁用 chain

> *Flink Job的提交流程*

用户提交的Flink Job会被转化成一个DAG任务运行，分别是：StreamGraph、JobGraph、ExecutionGraph，Flink中JobManager与TaskManager，JobManager与Client的交互是基于Akka工具包的，是通过消息驱动。整个Flink Job的提交还包含着ActorSystem的创建，JobManager的启动，TaskManager的启动和注册

> *Flink所谓"三层图"结构是哪几个"图"*

一个Flink任务的DAG生成计算图大致经历以下三个过程：

1. StreamGraph 最接近代码所表达的逻辑层面的计算拓扑结构，按照用户代码的执行顺序向StreamExecutionEnvironment添加StreamTransformation构成流式图。
2. JobGraph 从StreamGraph生成，将可以串联合并的节点进行合并，设置节点之间的边，安排资源共享slot槽位和放置相关联的节点，上传任务所需的文件，设置检查点配置等。相当于经过部分初始化和优化处理的任务图。 ExecutionGraph 由JobGraph转换而来，包含了任务具体执行所需的内容，是最贴近底层实现的执行图

> *JobManger在集群中扮演了什么角色*

JobManager 负责整个 Flink 集群任务的调度以及资源的管理，从客户端中获取提交的应用，然后根据集群中 TaskManager 上 TaskSlot 的使用情况，为提交的应用分配相应的 TaskSlot 资源并命令 TaskManager 启动从客户端中获取的应用。JobManager 相当于整个集群的 Master 节点，且整个集群有且只有一个活跃的 JobManager ，负责整个集群的任务管理和资源管理。JobManager 和 TaskManager 之间通过 Actor System 进行通信，获取任务执行的情况并通过 Actor System 将应用的任务执行情况发送给客户端。同时在任务执行的过程中，Flink JobManager 会触发 Checkpoint 操作，每个 TaskManager 节点 收到 Checkpoint 触发指令后，完成 Checkpoint 操作，所有的 Checkpoint 协调过程都是在 Fink JobManager 中完成。当任务完成后，Flink 会将任务执行的信息反馈给客户端，并且释放掉 TaskManager 中的资源以供下一次提交任务使用

> *JobManger在集群启动过程中起到什么作用*

JobManager的职责主要是接收Flink作业，调度Task，收集作业状态和管理TaskManager。它包含一个Actor，并且做如下操作：

1. RegisterTaskManager: 它由想要注册到JobManager的TaskManager发送。注册成功会通过AcknowledgeRegistration消息进行Ack
2. SubmitJob: 由提交作业到系统的Client发送。提交的信息是JobGraph形式的作业描述信息
3. CancelJob: 请求取消指定id的作业。成功会返回CancellationSuccess，否则返回CancellationFailure。 UpdateTaskExecutionState: 由TaskManager发送，用来更新执行节点(ExecutionVertex)的状态。成功则返回true，否则返回false
4. RequestNextInputSplit: TaskManager上的Task请求下一个输入split，成功则返回NextInputSplit，否则返回null
5. JobStatusChanged： 它意味着作业的状态(RUNNING, CANCELING, FINISHED, 等)发生变化。这个消息由ExecutionGraph发送

> *TaskManager在集群中扮演了什么角色？*

TaskManager 相当于整个集群的 Slave 节点，负责具体的任务执行和对应任务在每个节点上的资源申请和管理。客户端通过将编写好的 Flink 应用编译打包，提交到 JobManager，然后 JobManager 会根据已注册在 JobManager 中 TaskManager 的资源情况，将任务分配给有资源的 TaskManager节点，然后启动并运行任务。TaskManager 从 JobManager 接收需要部署的任务，然后使用 Slot 资源启动 Task，建立数据接入的网络连接，接收数据并开始数据处理。同时 TaskManager 之间的数据交互都是通过数据流的方式进行的。可以看出，Flink 的任务运行其实是采用多线程的方式，这和 MapReduce 多 JVM 进行的方式有很大的区别，Flink 能够极大提高 CPU 使用效率，在多个任务和 Task 之间通过 TaskSlot 方式共享系统资源，每个 TaskManager 中通过管理多个 TaskSlot 资源池进行对资源进行有效管理

> *TaskManager在集群启动过程中起到什么作用*

TaskManager的启动流程较为简单： 启动类：
org.apache.flink.runtime.taskmanager.TaskManager 核心启动方法 ：
selectNetworkInterfaceAndRunTaskManager 启动后直接向JobManager注册自己，注册完成后，进行部分模块的初始化

> *Flink 计算资源的调度是如何实现的*

TaskManager中最细粒度的资源是Task slot，代表了一个固定大小的资源子集，每个TaskManager会将其所占有的资源平分给它的slot。通过调整 task slot 的数量，用户可以定义task之间是如何相互隔离的。每个 TaskManager 有一个slot，也就意味着每个task运行在独立的 JVM 中。每个 TaskManager 有多个slot的话，也就是说多个task运行在同一个JVM中。 而在同一个JVM进程中的task，可以共享TCP连接（基于多路复用）和心跳消息，可以减少数据的网络传输，也能共享一些数据结构，一定程度上减少了每个task的消耗。 每个slot可以接受单个task，也可以接受多个连续task组成的pipeline，FlatMap函数占用一个taskslot，而key Agg函数和sink函数共用一个taskslot

> *简述Flink的数据抽象及数据交换过程*

Flink 为了避免JVM的固有缺陷例如java对象存储密度低，FGC影响吞吐和响应等，实现了自主管理内存。MemorySegment就是Flink的内存抽象。默认情况下，一个MemorySegment可以被看做是一个32kb大的内存块的抽象。这块内存既可以是JVM里的一个byte[]，也可以是堆外内存（DirectByteBuffer）。在MemorySegment这个抽象之上，Flink在数据从operator内的数据对象在向TaskManager上转移，预备被发给下个节点的过程中，使用的抽象或者说内存对象是Buffer。对接从Java对象转为Buffer的中间对象是另一个抽象StreamRecord

> *Flink 中的分布式快照机制是如何实现的*

Flink的容错机制的核心部分是制作分布式数据流和操作算子状态的一致性快照。 这些快照充当一致性checkpoint，系统可以在发生故障时回滚。 Flink用于制作这些快照的机制在“分布式数据流的轻量级异步快照”中进行了描述。 它受到分布式快照的标准Chandy-Lamport算法的启发，专门针对Flink的执行模型而定制

barriers在数据流源处被注入并行数据流中。快照n的barriers被插入的位置（我们称之为Sn）是快照所包含的数据在数据源中最大位置。例如，在Apache Kafka中，此位置将是分区中最后一条记录的偏移量。 将该位置Sn报告给checkpoint协调器（Flink的JobManager）。然后barriers向下游流动。当一个中间操作算子从其所有输入流中收到快照n的barriers时，它会为快照n发出barriers进入其所有输出流中。 一旦sink操作算子（流式DAG的末端）从其所有输入流接收到barriers n，它就向checkpoint协调器确认快照n完成。在所有sink确认快照后，意味快照着已完成。一旦完成快照n，job将永远不再向数据源请求Sn之前的记录，因为此时这些记录（及其后续记录）将已经通过整个数据流拓扑，也即是已经被处理结束

> *FlinkSQL是如何实现的*

Flink 将 SQL 校验、SQL 解析以及 SQL 优化交给了Apache Calcite。Calcite 在其他很多开源项目里也都应用到了，譬如 Apache Hive, Apache Drill, Apache Kylin, Cascading。Calcite 在新的架构中处于核心的地位，构建抽象语法树的事情交给了 Calcite 去做。SQL query 会经过 Calcite 解析器转变成 SQL 节点树，通过验证后构建成 Calcite 的抽象语法树（也就是图中的 Logical Plan）。另一边，Table API 上的调用会构建成 Table API 的抽象语法树，并通过 Calcite 提供的 RelBuilder 转变成 Calcite 的抽象语法树。然后依次被转换成逻辑执行计划和物理执行计划。在提交任务后会分发到各个 TaskManager 中运行，在运行时会使用 Janino 编译器编译代码后运行