Spark

1. 通常来说，Spark与MapReduce相比，Spark运行效率更高。请说明效率更高来源于Spark内置的哪些机制？

2. hadoop和spark使用场景？

Hadoop/MapReduce和Spark最适合的都是做离线型的数据分析，但Hadoop特别适合是单次分析的数据量“很大”的情景，而Spark则适用于数据量不是很大的情景。

一般情况下，对于中小互联网和企业级的大数据应用而言，单次分析的数量都不会“很大”，因此可以优先考虑使用Spark。

业务通常认为Spark更适用于机器学习之类的“迭代式”应用，80GB的压缩数据（解压后超过200GB），10个节点的集群规模，跑类似“sum+group-by”的应用，MapReduce花了5分钟，而spark只需要2分钟。

3. spark如何保证宕机迅速恢复?

适当增加spark standby master

编写shell脚本，定期检测master状态，出现宕机后对master进行重启操作

4. hadoop和spark的相同点和不同点？

Hadoop底层使用MapReduce计算架构，只有map和reduce两种操作，表达能力比较欠缺，而且在MR过程中会重复的读写hdfs，造成大量的磁盘io读写操作，所以适合高时延环境下批处理计算的应用；

Spark是基于内存的分布式计算架构，提供更加丰富的数据集操作类型，主要分成转化操作和行动操作，包括map、reduce、filter、flatmap、groupbykey、reducebykey、union和join等，数据分析更加快速，所以适合低时延环境下计算的应用；

spark与hadoop最大的区别在于迭代式计算模型。基于mapreduce框架的Hadoop主要分为map和reduce两个阶段，两个阶段完了就结束了，所以在一个job里面能做的处理很有限；spark计算模型是基于内存的迭代式计算模型，可以分为n个阶段，根据用户编写的RDD算子和程序，在处理完一个阶段后可以继续往下处理很多个阶段，而不只是两个阶段。所以spark相较于mapreduce，计算模型更加灵活，可以提供更强大的功能。

但是spark也有劣势，由于spark基于内存进行计算，虽然开发容易，但是真正面对大数据的时候，在没有进行调优的轻局昂下，可能会出现各种各样的问题，比如OOM内存溢出等情况，导致spark程序可能无法运行起来，而mapreduce虽然运行缓慢，但是至少可以慢慢运行完。

5. RDD持久化原理？

spark非常重要的一个功能特性就是可以将RDD持久化在内存中。

调用cache()和persist()方法即可。cache()和persist()的区别在于，cache()是persist()的一种简化方式，cache()的底层就是调用persist()的无参版本persist(MEMORY\_ONLY)，将数据持久化到内存中。

如果需要从内存中清除缓存，可以使用unpersist()方法。RDD持久化是可以手动选择不同的策略的。在调用persist()时传入对应的StorageLevel即可。

6. checkpoint检查点机制？

应用场景：当spark应用程序特别复杂，从初始的RDD开始到最后整个应用程序完成有很多的步骤，而且整个应用运行时间特别长，这种情况下就比较适合使用checkpoint功能。

原因：对于特别复杂的Spark应用，会出现某个反复使用的RDD，即使之前持久化过但由于节点的故障导致数据丢失了，没有容错机制，所以需要重新计算一次数据。

Checkpoint首先会调用SparkContext的setCheckPointDIR()方法，设置一个容错的文件系统的目录，比如说HDFS；然后对RDD调用checkpoint()方法。之后在RDD所处的job运行结束之后，会启动一个单独的job，来将checkpoint过的RDD数据写入之前设置的文件系统，进行高可用、容错的类持久化操作。

检查点机制是我们在spark streaming中用来保障容错性的主要机制，它可以使spark streaming阶段性的把应用数据存储到诸如HDFS等可靠存储系统中，以供恢复时使用。具体来说基于以下两个目的服务：

控制发生失败时需要重算的状态数。Spark streaming可以通过转化图的谱系图来重算状态，检查点机制则可以控制需要在转化图中回溯多远。

提供驱动器程序容错。如果流计算应用中的驱动器程序崩溃了，你可以重启驱动器程序并让驱动器程序从检查点恢复，这样spark streaming就可以读取之前运行的程序处理数据的进度，并从那里继续。

7. checkpoint和持久化机制的区别？

最主要的区别在于持久化只是将数据保存在BlockManager中，但是RDD的lineage(血缘关系，依赖关系)是不变的。但是checkpoint执行完之后，rdd已经没有之前所谓的依赖rdd了，而只有一个强行为其设置的checkpointRDD，checkpoint之后rdd的lineage就改变了。

持久化的数据丢失的可能性更大，因为节点的故障会导致磁盘、内存的数据丢失。但是checkpoint的数据通常是保存在高可用的文件系统中，比如HDFS中，所以数据丢失可能性比较低

8. RDD机制理解吗？

rdd分布式弹性数据集，简单的理解成一种数据结构，是spark框架上的通用货币。所有算子都是基于rdd来执行的，不同的场景会有不同的rdd实现类，但是都可以进行互相转换。rdd执行过程中会形成dag图，然后形成lineage保证容错性等。从物理的角度来看rdd存储的是block和node之间的映射。

RDD是spark提供的核心抽象，全称为弹性分布式数据集。

RDD在逻辑上是一个hdfs文件，在抽象上是一种元素集合，包含了数据。它是被分区的，分为多个分区，每个分区分布在集群中的不同结点上，从而让RDD中的数据可以被并行操作（分布式数据集）

比如有个RDD有90W数据，3个partition，则每个分区上有30W数据。RDD通常通过Hadoop上的文件，即HDFS或者HIVE表来创建，还可以通过应用程序中的集合来创建；RDD最重要的特性就是容错性，可以自动从节点失败中恢复过来。即如果某个结点上的RDD partition因为节点故障，导致数据丢失，那么RDD可以通过自己的数据来源重新计算该partition。这一切对使用者都是透明的。

RDD的数据默认存放在内存中，但是当内存资源不足时，spark会自动将RDD数据写入磁盘。比如某结点内存只能处理20W数据，那么这20W数据就会放入内存中计算，剩下10W放到磁盘中。RDD的弹性体现在于RDD上自动进行内存和磁盘之间权衡和切换的机制。

9. Spark streaming以及基本工作原理？

Spark streaming是spark core API的一种扩展，可以用于进行大规模、高吞吐量、容错的实时数据流的处理。

它支持从多种数据源读取数据，比如Kafka、Flume、Twitter和TCP Socket，并且能够使用算子比如map、reduce、join和window等来处理数据，处理后的数据可以保存到文件系统、数据库等存储中。

Spark streaming内部的基本工作原理是：接受实时输入数据流，然后将数据拆分成batch，比如每收集一秒的数据封装成一个batch，然后将每个batch交给spark的计算引擎进行处理，最后会生产处一个结果数据流，其中的数据也是一个一个的batch组成的。

10. DStream以及基本工作原理？

DStream是spark streaming提供的一种高级抽象，代表了一个持续不断的数据流。

DStream可以通过输入数据源来创建，比如Kafka、flume等，也可以通过其他DStream的高阶函数来创建，比如map、reduce、join和window等。

DStream内部其实不断产生RDD，每个RDD包含了一个时间段的数据。

Spark streaming一定是有一个输入的DStream接收数据，按照时间划分成一个一个的batch，并转化为一个RDD，RDD的数据是分散在各个子节点的partition中。

11. spark有哪些组件？

master：管理集群和节点，不参与计算。

worker：计算节点，进程本身不参与计算，和master汇报。

Driver：运行程序的main方法，创建spark context对象。

spark context：控制整个application的生命周期，包括dagsheduler和task scheduler等组件。

client：用户提交程序的入口。

12. spark工作机制？

用户在client端提交作业后，会由Driver运行main方法并创建spark context上下文。执行add算子，形成dag图输入dagscheduler，按照add之间的依赖关系划分stage输入task scheduler。task scheduler会将stage划分为task set分发到各个节点的executor中执行。

13. 说下宽依赖和窄依赖

宽依赖：

本质就是shuffle。父RDD的每一个partition中的数据，都可能会传输一部分到下一个子RDD的每一个partition中，此时会出现父RDD和子RDD的partition之间具有交互错综复杂的关系，这种情况就叫做两个RDD之间是宽依赖。

窄依赖：

父RDD和子RDD的partition之间的对应关系是一对一的。

14. Spark主备切换机制原理知道吗？

Master实际上可以配置两个，Spark原生的standalone模式是支持Master主备切换的。当Active Master节点挂掉以后，我们可以将Standby Master切换为Active Master。

Spark Master主备切换可以基于两种机制，一种是基于文件系统的，一种是基于ZooKeeper的。

基于文件系统的主备切换机制，需要在Active Master挂掉之后手动切换到Standby Master上；

而基于Zookeeper的主备切换机制，可以实现自动切换Master。

15. spark解决了hadoop的哪些问题？

MR：抽象层次低，需要使用手工代码来完成程序编写，使用上难以上手；

Spark：Spark采用RDD计算模型，简单容易上手。

MR：只提供map和reduce两个操作，表达能力欠缺；

Spark：Spark采用更加丰富的算子模型，包括map、flatmap、groupbykey、reducebykey等；

MR：一个job只能包含map和reduce两个阶段，复杂的任务需要包含很多个job，这些job之间的管理以来需要开发者自己进行管理；

Spark：Spark中一个job可以包含多个转换操作，在调度时可以生成多个stage，而且如果多个map操作的分区不变，是可以放在同一个task里面去执行；

MR：中间结果存放在hdfs中；

Spark：Spark的中间结果一般存在内存中，只有当内存不够了，才会存入本地磁盘，而不是hdfs；

MR：只有等到所有的map task执行完毕后才能执行reduce task；

Spark：Spark中分区相同的转换构成流水线在一个task中执行，分区不同的需要进行shuffle操作，被划分成不同的stage需要等待前面的stage执行完才能执行。

MR：只适合batch批处理，时延高，对于交互式处理和实时处理支持不够；

Spark：Spark streaming可以将流拆成时间间隔的batch进行处理，实时计算。

16. 数据倾斜的产生和解决办法？

数据倾斜以为着某一个或者某几个partition的数据特别大，导致这几个partition上的计算需要耗费相当长的时间。

在spark中同一个应用程序划分成多个stage，这些stage之间是串行执行的，而一个stage里面的多个task是可以并行执行，task数目由partition数目决定，如果一个partition的数目特别大，那么导致这个task执行时间很长，导致接下来的stage无法执行，从而导致整个job执行变慢。

避免数据倾斜，一般是要选用合适的key，或者自己定义相关的partitioner，通过加盐或者哈希值来拆分这些key，从而将这些数据分散到不同的partition去执行。

如下算子会导致shuffle操作，是导致数据倾斜可能发生的关键点所在：groupByKey；reduceByKey；aggregaByKey；join；cogroup；

17. 你用sparksql处理的时候， 处理过程中用的dataframe还是直接写的sql？为什么？

这个问题的宗旨是问你spark sql 中dataframe和sql的区别，从执行原理、操作方便程度和自定义程度来分析

这个问题。

18. 现场写一个笔试题

有hdfs文件，文件每行的格式为作品ID，用户id，用户性别。请用一个spark任务实现以下功能：

统计每个作品对应的用户（去重后）的性别分布。输出格式如下：作品ID，男性用户数量，女性用户数量

答案：

sc.textfile() .flatmap(.split(","))//分割成作品ID，用户id，用户性别

.map(((\_.1,\_.\_2),1))//((作品id,用户性别),1)

.reduceByKey(\_+\_)//((作品id,用户性别),n)

.map(\_.\_1.\_1,\_.\_1.\_2,\_.\_2)//(作品id,用户性别,n)

1

2

3

4

19. RDD中reduceBykey与groupByKey哪个性能好，为什么

reduceByKey：reduceByKey会在结果发送至reducer之前会对每个mapper在本地进行merge，有点类似于在MapReduce中的combiner。这样做的好处在于，在map端进行一次reduce之后，数据量会大幅度减小，从而减小传输，保证reduce端能够更快的进行结果计算。

groupByKey：groupByKey会对每一个RDD中的value值进行聚合形成一个序列(Iterator)，此操作发生在reduce端，所以势必会将所有的数据通过网络进行传输，造成不必要的浪费。同时如果数据量十分大，可能还会造成OutOfMemoryError。

所以在进行大量数据的reduce操作时候建议使用reduceByKey。不仅可以提高速度，还可以防止使用groupByKey造成的内存溢出问题。

20. Spark master HA主从切换过程不会影响到集群已有作业的运行，为什么

不会的。

因为程序在运行之前，已经申请过资源了，driver和Executors通讯，不需要和master进行通讯的。

21. spark master使用zookeeper进行ha，有哪些源数据保存到Zookeeper里面

spark通过这个参数spark.deploy.zookeeper.dir指定master元数据在zookeeper中保存的位置，包括Worker，Driver和Application以及Executors。standby节点要从zk中，获得元数据信息，恢复集群运行状态，才能对外继续提供服务，作业提交资源申请等，在恢复前是不能接受请求的。

注：Master切换需要注意2点：

1、在Master切换的过程中，所有的已经在运行的程序皆正常运行！

因为Spark Application在运行前就已经通过Cluster Manager获得了计算资源，所以在运行时Job本身的

调度和处理和Master是没有任何关系。

2、在Master的切换过程中唯一的影响是不能提交新的Job：一方面不能够提交新的应用程序给集群，

因为只有Active Master才能接受新的程序的提交请求；另外一方面，已经运行的程序中也不能够因

Action操作触发新的Job的提交请求。

Kafka

1. 为什么要使用 kafka？

缓冲和削峰：上游数据时有突发流量，下游可能扛不住，或者下游没有足够多的机器来保证冗余，kafka在中间可以起到一个缓冲的作用，把消息暂存在kafka中，下游服务就可以按照自己的节奏进行慢慢处理。

解耦和扩展性：项目开始的时候，并不能确定具体需求。消息队列可以作为一个接口层，解耦重要的业务流程。只需要遵守约定，针对数据编程即可获取扩展能力。

冗余：可以采用一对多的方式，一个生产者发布消息，可以被多个订阅topic的服务消费到，供多个毫无关联的业务使用。

健壮性：消息队列可以堆积请求，所以消费端业务即使短时间死掉，也不会影响主要业务的正常进行。

异步通信：很多时候，用户不想也不需要立即处理消息。消息队列提供了异步处理机制，允许用户把一个消息放入队列，但并不立即处理它。想向队列中放入多少消息就放多少，然后在需要的时候再去处理它们。

2. Kafka消费过的消息如何再消费？

kafka消费消息的offset是定义在zookeeper中的， 如果想重复消费kafka的消息，可以在redis中自己记录offset的checkpoint点（n个），当想重复消费消息时，通过读取redis中的checkpoint点进行zookeeper的offset重设，这样就可以达到重复消费消息的目的了

3. kafka的数据是放在磁盘上还是内存上，为什么速度会快？

kafka使用的是磁盘存储。

速度快是因为：

顺序写入：因为硬盘是机械结构，每次读写都会寻址->写入，其中寻址是一个“机械动作”，它是耗时的。所以硬盘 “讨厌”随机I/O， 喜欢顺序I/O。为了提高读写硬盘的速度，Kafka就是使用顺序I/O。

Memory Mapped Files（内存映射文件）：64位操作系统中一般可以表示20G的数据文件，它的工作原理是直接利用操作系统的Page来实现文件到物理内存的直接映射。完成映射之后你对物理内存的操作会被同步到硬盘上。

Kafka高效文件存储设计： Kafka把topic中一个parition大文件分成多个小文件段，通过多个小文件段，就容易定期清除或删除已经消费完文件，减少磁盘占用。通过索引信息可以快速定位

message和确定response的 大 小。通过index元数据全部映射到memory（内存映射文件），

可以避免segment file的IO磁盘操作。通过索引文件稀疏存储，可以大幅降低index文件元数据占用空间大小。

注：

Kafka解决查询效率的手段之一是将数据文件分段，比如有100条Message，它们的offset是从0到99。假设将数据文件分成5段，第一段为0-19，第二段为20-39，以此类推，每段放在一个单独的数据文件里面，数据文件以该段中 小的offset命名。这样在查找指定offset的

Message的时候，用二分查找就可以定位到该Message在哪个段中。

为数据文件建 索引数据文件分段 使得可以在一个较小的数据文件中查找对应offset的Message 了，但是这依然需要顺序扫描才能找到对应offset的Message。

为了进一步提高查找的效率，Kafka为每个分段后的数据文件建立了索引文件，文件名与数据文件的名字是一样的，只是文件扩展名为.index。

4. Kafka数据怎么保障不丢失？

分三个点说，一个是生产者端，一个消费者端，一个broker端。

生产者数据的不丢失

kafka的ack机制：在kafka发送数据的时候，每次发送消息都会有一个确认反馈机制，确保消息正常的能够被收到，其中状态有0，1，-1。

如果是同步模式：

ack设置为0，风险很大，一般不建议设置为0。即使设置为1，也会随着leader宕机丢失数据。所以如果要严格保证生产端数据不丢失，可设置为-1。

如果是异步模式：

也会考虑ack的状态，除此之外，异步模式下的有个buffer，通过buffer来进行控制数据的发送，有两个值来进行控制，时间阈值与消息的数量阈值，如果buffer满了数据还没有发送出去，有个选项是配置是否立即清空buffer。可以设置为-1，永久阻塞，也就数据不再生产。异步模式下，即使设置为-1。也可能因为程序员的不科学操作，操作数据丢失，比如kill -9，但这是特别的例外情况。

注：

ack=0：producer不等待broker同步完成的确认，继续发送下一条(批)信息。

ack=1（默认）：producer要等待leader成功收到数据并得到确认，才发送下一条message。

ack=-1：producer得到follwer确认，才发送下一条数据。

消费者数据的不丢失

通过offset commit 来保证数据的不丢失，kafka自己记录了每次消费的offset数值，下次继续消费的时候，会接着上次的offset进行消费。

而offset的信息在kafka0.8版本之前保存在zookeeper中，在0.8版本之后保存到topic中，即使消费者在运行过程中挂掉了，再次启动的时候会找到offset的值，找到之前消费消息的位置，接着消费，由于 offset 的信息写入的时候并不是每条消息消费完成后都写入的，所以这种情况有可能会造成重复消费，但是不会丢失消息。

唯一例外的情况是，我们在程序中给原本做不同功能的两个consumer组设置

KafkaSpoutConfig.bulider.setGroupid的时候设置成了一样的groupid，这种情况会导致这两个组共享同一份数据，就会产生组A消费partition1，partition2中的消息，组B消费partition3的消息，这样每个组消费的消息都会丢失，都是不完整的。 为了保证每个组都独享一份消息数据，groupid一定不要重复才行。

kafka集群中的broker的数据不丢失

每个broker中的partition我们一般都会设置有replication（副本）的个数，生产者写入的时候首先根据分发策略（有partition按partition，有key按key，都没有轮询）写入到leader中，follower（副本）再跟leader同步数据，这样有了备份，也可以保证消息数据的不丢失。

5. 采集数据为什么选择kafka？

采集层 主要可以使用Flume, Kafka等技术。

Flume：Flume 是管道流方式，提供了很多的默认实现，让用户通过参数部署，及扩展API.

Kafka：Kafka是一个可持久化的分布式的消息队列。 Kafka 是一个非常通用的系统。你可以有许多生产者和很多的消费者共享多个主题Topics。

相比之下,Flume是一个专用工具被设计为旨在往HDFS，HBase发送数据。它对HDFS有特殊的优化，并且集成了Hadoop的安全特性。

所以，Cloudera 建议如果数据被多个系统消费的话，使用kafka；如果数据被设计给Hadoop使用，使用Flume。

6. kafka 重启是否会导致数据丢失？

kafka是将数据写到磁盘的，一般数据不会丢失。

但是在重启kafka过程中，如果有消费者消费消息，那么kafka如果来不及提交offset，可能会造成数据的不准确（丢失或者重复消费）。

7. kafka 宕机了如何解决？

先考虑业务是否受到影响

kafka 宕机了，首先我们考虑的问题应该是所提供的服务是否因为宕机的机器而受到影响，如果服务提供没问题，如果实现做好了集群的容灾机制，那么这块就不用担心了。

节点排错与恢复

想要恢复集群的节点，主要的步骤就是通过日志分析来查看节点宕机的原因，从而解决，重新恢复节点。

8. 为什么Kafka不支持读写分离？

在 Kafka 中，生产者写入消息、消费者读取消息的操作都是与 leader 副本进行交互的，从 而实现的是一种主写主读的生产消费模型。

Kafka 并不支持主写从读，因为主写从读有 2 个很明显的缺点:

数据一致性问题：数据从主节点转到从节点必然会有一个延时的时间窗口，这个时间 窗口会导致主从节点之间的数据不一致。某一时刻，在主节点和从节点中 A 数据的值都为 X， 之后将主节点中 A 的值修改为 Y，那么在这个变更通知到从节点之前，应用读取从节点中的 A 数据的值并不为最新的 Y，由此便产生了数据不一致的问题。

延时问题：类似 Redis 这种组件，数据从写入主节点到同步至从节点中的过程需要经历 网络→主节点内存→网络→从节点内存 这几个阶段，整个过程会耗费一定的时间。而在 Kafka 中，主从同步会比 Redis 更加耗时，它需要经历 网络→主节点内存→主节点磁盘→网络→从节 点内存→从节点磁盘 这几个阶段。对延时敏感的应用而言，主写从读的功能并不太适用。

而kafka的主写主读的优点就很多了：

可以简化代码的实现逻辑，减少出错的可能;

将负载粒度细化均摊，与主写从读相比，不仅负载效能更好，而且对用户可控;

没有延时的影响;

在副本稳定的情况下，不会出现数据不一致的情况。

9. kafka数据分区和消费者的关系？

每个分区只能由同一个消费组内的一个消费者(consumer)来消费，可以由不同的消费组的消费者来消费，同组的消费者则起到并发的效果。

10. kafka的数据offset读取流程

连接ZK集群，从ZK中拿到对应topic的partition信息和partition的Leader的相关信息

连接到对应Leader对应的broker

consumer将⾃自⼰己保存的offset发送给Leader

Leader根据offset等信息定位到segment（索引⽂文件和⽇日志⽂文件）

根据索引⽂文件中的内容，定位到⽇日志⽂文件中该偏移量量对应的开始位置读取相应⻓长度的数据并返回给consumer

11. kafka内部如何保证顺序，结合外部组件如何保证消费者的顺序？

kafka只能保证partition内是有序的，但是partition间的有序是没办法的。爱奇艺的搜索架构，是从业务上把需要有序的打到同⼀个partition。

12. Kafka消息数据积压，Kafka消费能力不足怎么处理？

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。（两者缺一不可）

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。

13. Kafka单条日志传输大小

kafka对于消息体的大小默认为单条最大值是1M但是在我们应用场景中, 常常会出现一条消息大于1M，如果不对kafka进行配置。则会出现生产者无法将消息推送到kafka或消费者无法去消费kafka里面的数据, 这时我们就要对kafka进行以下配置：server.properties

replica.fetch.max.bytes: 1048576 broker可复制的消息的最大字节数, 默认为1M

message.max.bytes: 1000012 kafka 会接收单个消息size的最大限制， 默认为1M左右

1

2

注意：message.max.bytes必须小于等于replica.fetch.max.bytes，否则就会导致replica之间数据同步失败。

Hbase

1. Hbase是怎么写数据的？

Client写入 -> 存入MemStore，一直到MemStore满 -> Flush成一个StoreFile，直至增长到一定阈值 -> 触发Compact合并操作 -> 多个StoreFile合并成一个StoreFile，同时进行版本合并和数据删除 -> 当StoreFiles Compact后，逐步形成越来越大的StoreFile -> 单个StoreFile大小超过一定阈值后（默认10G），触发Split操作，把当前Region Split成2个Region，Region会下线，新Split出的2个孩子Region会被HMaster分配到相应的HRegionServer 上，使得原先1个Region的压力得以分流到2个Region上

由此过程可知，HBase只是增加数据，没有更新和删除操作，用户的更新和删除都是逻辑层面的，在物理层面，更新只是追加操作，删除只是标记操作。

用户写操作只需要进入到内存即可立即返回，从而保证I/O高性能。

2. HDFS和HBase各自使用场景

首先一点需要明白：Hbase是基于HDFS来存储的。

HDFS：

一次性写入，多次读取。

保证数据的一致性。

主要是可以部署在许多廉价机器中，通过多副本提高可靠性，提供了容错和恢复机制。

HBase：

瞬间写入量很大，数据库不好支撑或需要很高成本支撑的场景。

数据需要长久保存，且量会持久增长到比较大的场景。

HBase不适用与有 join，多级索引，表关系复杂的数据模型。

大数据量（100s TB级数据）且有快速随机访问的需求。如：淘宝的交易历史记录。数据量巨大无容置疑，面向普通用户的请求必然要即时响应。

业务场景简单，不需要关系数据库中很多特性（例如交叉列、交叉表，事务，连接等等）。

3. Hbase的存储结构

Hbase 中的每张表都通过行键(rowkey)按照一定的范围被分割成多个子表（HRegion），默认一个HRegion 超过256M 就要被分割成两个，由HRegionServer管理，管理哪些 HRegion 由 Hmaster 分配。 HRegion 存取一个子表时，会创建一个 HRegion 对象，然后对表的每个列族（Column Family）创建一个 store 实例， 每个 store 都会有 0 个或多个 StoreFile 与之对应，每个 StoreFile 都会对应一个HFile，HFile 就是实际的存储文件，一个 HRegion 还拥有一个 MemStore实例。

4. 热点现象（数据倾斜）怎么产生的，以及解决方法有哪些

热点现象：

某个小的时段内，对HBase的读写请求集中到极少数的Region上，导致这些region所在的RegionServer处理请求量骤增，负载量明显偏大，而其他的RgionServer明显空闲。

热点现象出现的原因：

HBase中的行是按照rowkey的字典顺序排序的，这种设计优化了scan操作，可以将相关的行以及会被一起读取的行存取在临近位置，便于scan。然而糟糕的rowkey设计是热点的源头。

热点发生在大量的client直接访问集群的一个或极少数个节点（访问可能是读，写或者其他操作）。大量访问会使热点region所在的单个机器超出自身承受能力，引起性能下降甚至region不可用，这也会影响同一个RegionServer上的其他region，由于主机无法服务其他region的请求。

热点现象解决办法：

为了避免写热点，设计rowkey使得不同行在同一个region，但是在更多数据情况下，数据应该被写入集群的多个region，而不是一个。常见的方法有以下这些：

加盐：在rowkey的前面增加随机数，使得它和之前的rowkey的开头不同。分配的前缀种类数量应该和你想使用数据分散到不同的region的数量一致。加盐之后的rowkey就会根据随机生成的前缀分散到各个region上，以避免热点。

哈希：哈希可以使负载分散到整个集群，但是读却是可以预测的。使用确定的哈希可以让客户端重构完整的rowkey，可以使用get操作准确获取某一个行数据

反转：第三种防止热点的方法时反转固定长度或者数字格式的rowkey。这样可以使得rowkey中经常改变的部分（最没有意义的部分）放在前面。这样可以有效的随机rowkey，但是牺牲了rowkey的有序性。反转rowkey的例子以手机号为rowkey，可以将手机号反转后的字符串作为rowkey，这样的就避免了以手机号那样比较固定开头导致热点问题

时间戳反转：一个常见的数据处理问题是快速获取数据的最近版本，使用反转的时间戳作为rowkey的一部分对这个问题十分有用，可以用 Long.Max\_Value - timestamp 追加到key的末尾，例如[key][reverse\_timestamp],[key]的最新值可以通过scan [key]获得[key]的第一条记录，因为HBase中rowkey是有序的，第一条记录是最后录入的数据。

比如需要保存一个用户的操作记录，按照操作时间倒序排序，在设计rowkey的时候，可以这样设计[userId反转]

[Long.Max\_Value - timestamp]，在查询用户的所有操作记录数据的时候，直接指定反转后的userId，startRow是[userId反转][000000000000],stopRow是[userId反转][Long.Max\_Value - timestamp]

如果需要查询某段时间的操作记录，startRow是[user反转][Long.Max\_Value - 起始时间]，stopRow是[userId反转][Long.Max\_Value - 结束时间]

HBase建表预分区：创建HBase表时，就预先根据可能的RowKey划分出多个region而不是默认的一个，从而可以将后续的读写操作负载均衡到不同的region上，避免热点现象。

5. HBase的 rowkey 设计原则

长度原则：100字节以内，8的倍数最好，可能的情况下越短越好。因为HFile是按照 keyvalue 存储的，过长的rowkey会影响存储效率；其次，过长的rowkey在memstore中较大，影响缓冲效果，降低检索效率。最后，操作系统大多为64位，8的倍数，充分利用操作系统的最佳性能。

散列原则：高位散列，低位时间字段。避免热点问题。

唯一原则：分利用这个排序的特点，将经常读取的数据存储到一块，将最近可能会被访问 的数据放到一块。

6. HBase的列簇设计

原则：在合理范围内能尽量少的减少列簇就尽量减少列簇，因为列簇是共享region的，每个列簇数据相差太大导致查询效率低下。

最优：将所有相关性很强的 key-value 都放在同一个列簇下，这样既能做到查询效率最高，也能保持尽可能少的访问不同的磁盘文件。以用户信息为例，可以将必须的基本信息存放在一个列族，而一些附加的额外信息可以放在另一列族。

7. HBase 中 compact 用途是什么，什么时候触发，分为哪两种，有什么区别

在 hbase 中每当有 memstore 数据 flush 到磁盘之后，就形成一个 storefile，当 storeFile的数量达到一定程度后，就需要将 storefile 文件来进行 compaction 操作。

Compact 的作用：

合并文件

清除过期，多余版本的数据

提高读写数据的效率

4

HBase 中实现了两种 compaction 的方式：minor and major. 这两种 compaction 方式的

区别是：

Minor 操作只用来做部分文件的合并操作以及包括 minVersion=0 并且设置 ttl 的过

期版本清理，不做任何删除数据、多版本数据的清理工作。

Major 操作是对 Region 下的 HStore 下的所有 StoreFile 执行合并操作，最终的结果

是整理合并出一个文件。

1. spark处理数据倾斜。

spark数据倾斜主要出现在shuffle阶段。

主要表现：1.大部分task都执行迅速，只有少数几个或一个task运行缓慢。

1. 大部分task都执行迅速，在执行有的少数task的时候突然报OOM，反复执行都是在有一个task报oom。

定位数据倾斜：

1. 查看shuffle算子，如：reduceBykey，countBykey，groupBykey，join等算子，根据代码逻辑查看此处是否出现数据倾斜。
2. 查看spark作业的日志文件，查看具体出现错误的stage，对应的shuffle算子是哪一个。

解决数据倾斜的方法：

1. 聚合原始数据：

为了避免数据倾斜，可以对原始数据进行处理，避免shuffle过程，也就从根本上避免了数据倾斜问题，比如来源于hive表的数据，可以对hive表中的数据进行聚合，按照key进行分组。

1. 过滤导致倾斜的key：

在允许丢弃少量数据的情况下，可以过滤掉导致数据倾斜的key。

1. 提高shuffle操作中的reduce并行度。

适用于较多key对应的数据量都比较大的情况。

1. 适用随机key实现双重聚合。

通过map算子给每个数据的key添加随机前缀，将key打散，使原先相同的key变得不一样。

1. 将reduce join转换为map join。

采用广播小RDD全量数据+map算子实现与join相同的效果。

**注意，RDD 是并不能进行广播的，只能将 RDD 内部的数据通过 collect 拉取**

**到 Driver 内存然后再进行广播。**

1. **sample采样对**倾斜的key单独join。

如果单个key的数量特别多，就可以考虑将这个key单独拿出来处理。

7.使用随机数以及扩容进行join。

如果RDD中有大量的key导致数据倾斜，可以考虑对其中一个RDD数据进行扩容，对另外的RDD进行稀释再join。

1. spark故障排除。

9.1**控制 reduce 端缓冲大小以避免 OOM**

首先将数据拉取到buffer缓冲区，reduce端的task会一边拉取一边计算，不一定每次都会拉满48m的数据，大多数时候就是拉取一部分就处理掉，48M-->12M，增加拉取次数，性能换执行成功。

9.2**JVM GC 导致的 shuffle 文件拉取失败**

在shuffle操作中，后面stage的task想要去前一个stage的task的executor拉取数据，次数对方正在GC，导致数据拉取失败。

增加拉取重试次数和拉取等待时间间隔。

9.3序列化导致的错误。

当 Spark 作业在运行过程中报错，而且报错信息中含有 Serializable 等类似词汇，那么

可能是序列化问题导致的报错。

序列化问题要注意以下三点：

1. 作为 RDD 的元素类型的自定义类，必须是可以序列化的；

2. 算子函数里可以使用的外部的自定义变量，必须是可以序列化的；

3. 不可以在 RDD 的元素类型、算子函数里使用第三方的不支持序列化的类型，例如

Connection

9.4**解决算子函数返回 NULL 导致的问题**

在一些算子函数里，需要我们有一个返回值，但是在一些情况下我们不希望有

返回值，此时我们如果直接返回 NULL，会报错，例如 Scala.Math(NULL)异常。

如果你遇到某些情况，不希望有返回值，那么可以通过下述方式解决：

1. 返回特殊值，不返回 NULL，例如“-1”；

2. 在通过算子获取到了一个 RDD 之后，可以对这个 RDD 执行 filter 操作，进

行数据过滤，将数值为-1 的数据给过滤掉；

3. 在使用完 filter 算子后，继续调用 coalesce 算子进行优化。

spark的checkpoint机制是为了弥补RDD持久化存储的不足性，RDD持久化在内存或者磁盘，这样的存储是不可靠的checkpoint可以将数据存储在HDFS这样的高可靠、高容错的介质上。较长的依赖关系。

宽窄依赖

宽：一个父RDD对应多个子RDD的分区