1. 请分析大数据相关特征，包括数据体量大(Volume)、数据流动性高(Velocity)以及数据种类繁多(Variety)分别对大数据处理带来了什么具体挑战，并简要阐述相应措施。

数据体量大（Volume）：

- 挑战：大数据的体量庞大，传统的存储和处理技术无法有效地处理和分析这么大规模的数据。
- 措施：采用**分布式存储和处理技术**，如 Hadoop 和 Spark 等，可以将数据分割成小块并分布在多台计算机上进行并行处理。此外，采用压缩、索引和分区等技术可以优化数据存储和访问效率。

数据流动性高（Velocity）：

- 挑战：大数据通常以高速产生和流动，**处理速度跟不上数据的生成和传输速度**，可能导致数据堆积和延迟。
- 措施：**实时数据处理和流式处理技术**可以帮助实时捕获、处理和分析数据流。采用高吞吐量的存储系统和并行处理框架，如 Apache Kafka 和 Apache Flink 等，可以实现数据的快速接收、处理和传输。

数据种类繁多（Variety）：

- 挑战：大数据涵盖**多种类型的数据，包括结构化数据、非结构化数据**、文本、图像、音频等，这些数据具有不同的格式和特性，难以统一处理和分析。
- 措施：采用数据整合和集成技术，如 ETL（抽取、转换和加载）工具和数据湖架构，可以将不同格式的数据整合到一个统一的存储库中。利用机器学习和自然语言处理等技术，可以对非结构化数据进行自动分类、标注和分析。

2. 简述复制技术为什么会带来数据一致性问题，并分析如何保证多副本的一致性。

复制技术是指在分布式系统中将数据副本存储在不同的节点上，以提高系统的可用性和容错性。然而，复制技术也会引入数据一致性的问题，因为**多个副本之间的数据可能会发生不一致**的情况。

复制技术可能导致数据一致性问题的原因主要有两个：

- 副本之间的**延迟**：当数据被复制到不同的副本时，由于网络延迟或其他因素，副本之间的同步可能存在一定的延迟。在这段延迟期间，如果对其中一个副本进行了修改，而其他副本还未同步更新，就会导致副本之间的数据不一致。
- **并发更新**：如果多个客户端同时对不同的副本进行写入操作，由于复制操作的时间差，可能会出现并发更新的情况。这会导致不同副本上的数据发生冲突，无法保证一致性

为了保证多副本的一致性，可以采取以下措施：

- 同步复制：在进行数据复制时，要求所有副本必须同步更新。一种常见的同步复制方式是主从复制，其中一个主节点负责接收和处理写操作，然后将更新同步到所有的从节点。主节点确认写操作成功后，从节点才能响应读请求，确保数据一致性。
- 一致性协议：使用一致性协议来保证复制副本之间的一致性。例如，分布式系统中常用的一致性协议是 Paxos 和 Raft。这些协议通过选举和多阶段提交等机制，确保副本之间达成一致的数据状态。
- 冲突解决策略：当出现并发操作冲突时，需要采取合适的策略来解决冲突并确保数据一致。例如，可以使用锁机制或乐观并发控制（Optimistic Concurrency Control）来避免数据冲突，或者采用冲突检测和解决算法来处理冲突情况。
- 增量更新和日志记录：通过记录数据更新的日志或者增量更新，可以在副本之间传输和应用这些更新，从而保持数据的一致性。如果出现问题，可以根据日志进行回滚或修复。
- 数据一致性检查：定期进行数据一致性检查，确保所有副本的数据保持一致。可以使用一致性哈希等算法来检查副本之间的数据一致性，并进行修复和同步。

3. 副本的放置需要遵循哪些原则？简述在集群环境下的多副本放置策略。

   负载均衡，容错性，数据一致性，可扩展性

   副本放置策略：

   - 第一个副本在本节点。
   - 第二个副本在远端机架的节点。

   第三个副本看之前的两个副本是否在同一机架，如果是则选择其他机架，否则选择和第一个副本相同机架的不同节点，第四个及以上，随机选择副本存放位置

4. 简述分布式环境下 CAP 原理的内涵。

CAP 原则是 NoSQL（not only SQL）数据库的理论基础。其三要素涵义

1) 一致性：要求系统保持数据一致
2) 可用性：指系统处于可用状态
3) 分区容忍性：在存在多个数据分区或数据分区发生变化的情况下，分布式系统仍然能够满足一致性和可用性
4) 一致性（Consistency）：一致性要求系统的所有副本在同一时间具有相同的数据值。当一个节点更新了数据后，所有其他节点应该能够读取到最新的数据。在分布式系统中，实现一致性通常需要进行同步操作，对性能和延迟有一定影响。
5) 可用性（Availability）：可用性要求系统在任何时候都能够处理客户端的请求并返回合理的响应，即系统具备高可用性。在可用性方面，分布式系统应该能够继续工作，即使其中的某些节点或组件出现故障。
6) 分区容错性（Partition Tolerance）：分区容错性指的是系统在面对网络分区（节点之间的通信中断）时能够继续运行，即系统能够容忍节点之间的通信故障或延迟。在分布式系统中，网络分区是不可避免的，例如由于网络故障或节点故障而导致的通信中断。

一个分布式系统不可能完全同时满足 C，A，P 三个要素，设计分布式系统时必须取舍。

5. 向量钟和 Merkle 树分别是什么？简述他们的作用

向量钟是一种用于跟踪**事件顺序和发生关系的逻辑时钟**。它用于**解决分布式系统中的一致性问题**，特别是在分布式存储系统中用于解决冲突和并发操作的排序问题。每个节点或进程都维护一个向量钟，其中每个维度代表一个节点或进程，并记录了该节点或进程对其他节点或进程的事件顺序。向量钟的更新和比较规则使得可以判断事件的因果关系，从而实现事件的全局排序和一致性。

Merkle 树是一种**哈希树的数据结构，用于验证和保证数据的完整性**。

Merkle 树的基本思想是通过逐级哈希计算将数据分成不断缩小的数据块，直到最终形成一个根哈希值。每个数据块和非叶子节点都通过哈希函数计算得到一个哈希值，这些哈希值构成了树形结构。根哈希值作为整个数据集的摘要，可以用于快速验证和检测数

据是否被篡改。

6. NVM 有什么特性？为什么要用 DRAM 和 NVM 一起构成异构内存系统？

持久性/非易失性：NVM 是一种非易失性存储器，即使在断电或重启后也可以保持数据的存储

高密度：NVM 具有较高的存储密度，因为它可以在较小的物理空间内存储更多的数据。

低功耗：相对于 DRAM，NVM 通常具有较低的功耗，这是因为它不需要持续的电源供应来保持数据。

尽管 NVM 具有许多优点，但它也有一些限制，例如较慢的写入速度和有限的写入寿命。与此相比，DRAM 具有高速的读写操作和无限的写入寿命，但它是易失性的，断电后数据会丢失。

将 DRAM 和 NVM 组合在一起构成异构内存系统可以充分发挥它们各自的优势：

- 提高性能：将常用的数据和热数据存储在 DRAM 中，可以实现低延迟的高速访问，从而提高系统的性能。同时，将较少访问的数据和冷数据存储在 NVM 中，可以利用其高密度和非易失性的特性。

- 扩展容量：由于 NVM 具有较高的存储密度，将其用作扩展内存可以增加系统的总容量。这对于需要处理大规模数据集的应用程序非常有用。

- 数据持久性：将关键的持久性数据存储在 NVM 中可以确保数据在断电或系统崩溃后的安全性和可靠性。这对于需要持久性存储的应用程序非常重要，例如数据库系统或文件系统。

- 能耗优化：将常用数据存储在功耗较低的 NVM 中可以降低系统的能耗，从而提高能源效率。同时，DRAM 的高速读写操作可以满足对性能要求较高的任务。

7. 内存计算模式是什么？为什么内存系统对大数据处理很重要？

内存计算模式（In-Memory Computing）是一种计算模式，其中计算任务直接在内存中进行，而不是将数据从存储器（如硬盘或固态驱动器）加载到内存中进行处理。它利用内存的高速读写能力和低延迟特性，将数据存储在内存中，并在内存中进行计算操作，以加快数据处理速度和提高系统性能。

内存系统对大数据处理非常重要的原因如下：

快速数据访问

实时数据处理

大规模并行处理

降低存储成本

8. Dynamo 如何对数据进行分区？

Dynamo 使用了改进的一致性哈希算法：每个物理节点根据其性能的差异分配多个 token，每个 token 对应一个“虚拟节点”。每个虚拟节点的处理能力基本相当，并随机分布在哈希空间中。存储时，数据按照哈希值落到某个虚拟节点负责的区域，然后被存储在该虚拟节点所对应的物理节点中。

9. 什么是函数式编程？它有哪些特点？

函数式编程（Functional Programming）是一种编程范式，它强调将计算视为数学函数的求值，避免状态和变量。

特点

纯函数：函数式编程鼓励编写纯函数，即没有副作用和依赖于外部状态的函数。纯函数的输出仅由输入决定，不会修改传入的参数或对外部状态进行修改。

不可变数据（Immutable Data）：函数式编程倾向于使用不可变数据结构，即数据一旦创建就不能被修改。

函数组合：函数式编程鼓励将函数组合起来创建更复杂的功能。函数可以作为参数传递给其他函数，也可以作为返回值，这种高阶函数的特性使得函数可以灵活地组合和重用。

递归（Recursion）：函数式编程常常使用递归来处理循环和迭代的操作。

惰性求值（Lazy Evaluation）：函数式编程支持惰性求值，即仅在需要时才计算表达式的值。这种延迟计算的方式可以提高性能，避免不必要的计算

引用透明（Referential Transparency）：函数式编程追求引用透明性，即在程序中可以用表达式的结果替换该表达式本身，而不影响程序的行为。

10. 简述引用透明性、高阶函数、函数柯里化、惰性求值的内涵

引用透明性（Referential Transparency）：引用透明性是函数式编程中的一个重要概念，指的是在程序中可以用表达式的结果替换该表达式本身，而不会改变程序的行为。

高阶函数（Higher-order Functions）：高阶函数是指能够接受一个或多个函数作为参数，并/或返回一个函数的函数。

函数柯里化（Function Currying）：函数柯里化是一种将接受多个参数的函数转化为接受单个参数的函数序列的技术。把接受多个参数的函数变成接受一个单一参数的函数，并且返回接受余下参数且返回结果的新函数

惰性求值（Lazy Evaluation）：惰性求值是一种计算策略，即仅在需要时才计算表达式的值。

11. 简述 Google MapReduce 和 Hadoop 的异同点。

Google MapReduce 和 Hadoop 都是用于处理大规模数据的分布式计算框架，它们有一些相似之处，但也存在一些差异。

相同点：

1. 大规模数据处理：Google MapReduce 和 Hadoop 都专注于处理大规模数据集。它们能够**将数据分成小块进行并行处理**，从而加快计算速度。
2. 分布式计算：两者都基于分布式计算的概念，将数据和计算任务分发到多台计算机上进行并行处理。这种分布式架构使得处理大规模数据集变得可行且高效。
3. 容错性：Google MapReduce 和 Hadoop 都具备容错性，即在计算过程中能够处理节点故障和数据丢失等问题，保证计算的正确性和可靠性。

不同点：

1. 来源和开源性：Google MapReduce 是由 **Google 公司**开发的用于**内部数据处理**的框架，而 Hadoop 是 **Apache 基金会开发的开源框架**，可供广大用户免费使用和定制。
2. 生态系统和组件：Hadoop 生态系统更为庞大，提供了丰富的组件和工具，如 Hadoop Distributed File System（HDFS）作为分布式文件系统，YARN 作为资源管理器。而 Google MapReduce 相对较为封闭，用户接口相对简单，仅关注 Map 和 Reduce 两个阶段。
3. 数据复制策略：Hadoop 的 HDFS 采用了**数据复制**的策略，将数据复制到多个节点上以提供容错性和高可用性。而 Google MapReduce 则依赖 Google File System（**GFS**）来管理数据的存储和复制，具有类似的目标。
4. 编程模型：在编程模型上，Google MapReduce 使用的是基于**函数式**编程的模型，用户需要实现 Map 和 Reduce 函数来指定计算逻辑。而 Hadoop 采用的是基于 **Java** 的编程模型，用户需要编写 Java 代码来定义 Map 和 Reduce 任务。

12. 假设有 1 百万个均匀分布的数，试给出抽取其中前 100 个最大数的 MapReduce 方案，并给出 Map 和 Reduce 的伪代码。

```
Map(data):
    top100 = []
    for num in data:
        if len(top100) < 100:
            top100.append(num)
```

```
            else:
                min_num = min(top100)
                if num > min_num:
                    top100.remove(min_num)
                    top100.append(num)
    emit("top100", top100)

    Reduce(key, values):
        all_top100 = []
        for value in values:
            all_top100.extend(value)
        all_top100.sort(reverse=True)
        emit("result", all_top100[:100])
```

在 Map 阶段，每个 Mapper 接收一部分数据，并维护一个大小为 100 的列表，记录当前最大的 100 个数。如果新的数比列表中的最小值大，则替换最小值。最后，每个 Mapper 将自己的 top100 列表作为中间结果进行输出。

在 Reduce 阶段，所有 Mapper 的输出会被合并到一个 Reducer 中。Reducer 会将所有 top100 列表进行合并，并按降序排序。最后，Reducer 输出前 100 个最大数作为最终结果。

13. 请简述 at most once、at least once 和 exactly once 这三种语义保障。

at most once（至多一次）：该语义保障确保消息最多被传递一次，即消息可能会丢失，但不会被重复传递。在这种语义中，系统不保证消息的可靠传递，可能会因为网络故障、节点故障或其他原因导致消息丢失，但不会发生重复传递的情况。

at least once（至少一次）：该语义保障确保消息至少被传递一次，即消息不会丢失，但可能会被重复传递。在这种语义中，系统保证了消息的可靠传递，但由于网络延迟、节点重启或其他原因，可能会导致消息被重复传递。

exactly once（正好一次）：该语义保障确保消息被精确地传递一次，即消息不会丢失，也不会被重复传递。在这种语义中，系统保证了消息的可靠传递，并采取了一些机制来避免重复传递。确保正好一次语义是最理想的，因为它保证了消息不会丢失，同时避免了重复处理的问题。

14. 简述 Apache Storm 和 Spark Streaming 的特点和区别。

Apache Storm 和 Spark Streaming 都是流式数据处理框架，但它们有一些不同的特点和区别。

Apache Storm 的特点和优势：

低延迟：Apache Storm 专注于**实时数据**处理，并且具有**低延迟**的特点。它可以在毫秒级别内处理和传递数据，适用于对实时性要求较高的应用场景。

容错性：Storm 具有**高度容错**的特性，能够自动进行**故障检测和故障恢复**。它通过在集群中运行多个任务实例来实现容错，并且能够在节点故障时自动重新分配任务。

基于消息流的处理模型：Storm 采用流式处理模型，数据以**消息流**的形式在拓扑结构中流动。这种模型适用于处理连续的、无限流的数据，并能够进行实时的转换和聚合操作。

Spark Streaming 的特点和优势：

批处理和流处理的统一：Spark Streaming 提供了一种统一的编程模型，将流处理和批处理结合在一起。它将实时数据流划分为小的批次，并以**微批处理**的方式对每个批次进行处理，从而在一定程度上实现了流处理功能。

高级 API 和丰富的生态系统：Spark Streaming 构建在 Apache Spark 之上，继承了 Spark 的高级 API 和丰富的生态系统。它可以与 Spark 的机器学习、图处理和批处理等功能无缝集成，提供了更广泛的数据处理能力。

高吞吐量和数据处理效率：Spark Streaming 通过批处理的方式进行数据处理，利用 Spark 的优化引擎和内存计算能力，能够实现**高吞吐量的数据处理**，并提供更高的计算效率。

区别：

数据处理模型：Storm 采用流式处理模型，数据以消息流的形式在拓扑结构中流动；而 Spark Streaming 采用微批处理模型，将实时数据流划分为小的批次进行处理。

延迟和吞吐量：由于 Storm 专注于低延迟的实时处理，适用于对实时响应要求较高的场景；而 Spark Streaming 通过批处理的方式，虽然牺牲了一定的延迟，但可以提供更高的吞吐量。

生态系统和功能扩展：Spark Streaming 构建在 Spark 之上，能够充分利用 Spark 的生态系统和丰富的功能模块，提供更广泛的数据处理能力；而 Storm 相对较为轻量级，专注于实时流处理，生态系统相对较小。

15. 简述几种主要的图计算编程模型，并分析它们的优点和缺点。

以点为中心的编程模型

将计算任务分解为每个顶点上的计算和消息传递，易于理解和实现。

存在大量随机访问和预处理操作 时间开销大

以边为中心的编程模型

降低了时间开销

以路径为中心的编程模型

能够处理大规模图数据 简化了迭代计算步骤，加快了算法执行速度

以子图为中心的编程模型

加快算法收敛速度

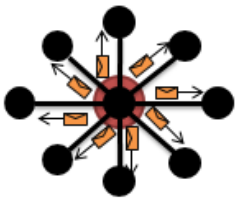
存在子图划分问题

16. 采用 Scatter-Gather 编程模型，写出单源点最短连接算法 SSSP 的伪代码。

17. 对于幂律图中的高维度节点（例如 Twitter 中具有大量粉丝的美国总统），有以下几种处理方式：

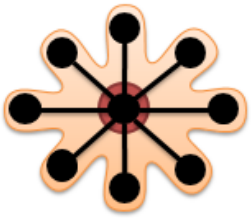
Sequentially process edges

顺序处理



Sends many messages (Pregel)

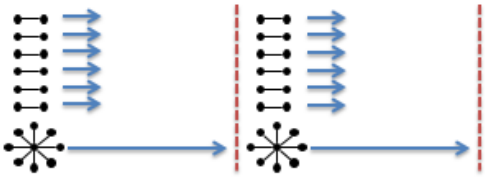
发送消息



Touches a large fraction of graph (GraphLab)

接触大部分图

共享状态：异步执行



共享状态：同步执行

请写出以上几种方法的缺陷。

对于幂律图中的高维度节点，以下是以上几种处理方式的缺陷：

顺序处理：

缺陷：顺序处理意味着**逐个处理每个节点，当遇到高维度节点时，处理时间会显著增加。这将导致整体处理时间的延长**，因为高维度节点需要更多的计算资源和时间，同时也会导致其他节点的处理受阻。

发送消息：

缺陷：在发送消息的方法中，**高维度节点可能会收到大量的消息，这可能引发消息堆积和延迟。**处理这些消息需要额外的计算资源和时间，并可能导致网络拥塞。此外，如果高维度节点无法及时处理收到的消息，可能会影响整个处理过程的效率和性能。

接触大部分图：

缺陷：**对于高维度节点，接触大部分图意味着需要处理大量的边和节点。这将导致更高的计算复杂度和存储需求**。同时，由于幂律图中高维度节点的数量相对较少，接触大部分图的方法可能会导致大量的**冗余计算和资源浪费**。

共享状态：异步执行：

缺陷：在异步执行中，高维度节点可能在不同的时间点完成计算并更新共享状态。这**可能导致状态不一致的问题**，因为其他节点可能仍在使用过时的状态进行计算。处理这种状态不一致性需要额外的同步机制和冲突解决策略，增加了复杂性和开销。

共享状态：同步执行：

缺陷：在同步执行中，**所有节点都必须等待高维度节点完成计算和状态更新。这将导致整体处理时间的延长**，因为其他节点需要等待高维度节点的完成。此外，如果高维度节点的计算时间较长，可能会导致其他节点的性能下降和资源闲置。

观察多层划分算法的伪代码，尝试理解操作 Coarsening 和 Uncoarsening 的意义

Multilevel Graph Partition (Graph G)

① Take the weighted undirected graph as input

② While ($|V|$> coarsening threshold) perform

Coarsening (G_i to G_i +1)

③ For all v in V

If vertex degree (v) is high perform

Balanced partitioning

Else

Perform Greedy partitioning

④ Simulated annealing(initial partition)

⑤ While (G_i!= G) perform

Uncoarsening (G_i to G_i-1)

18. 比较 Raft 协议和 Paxos 协议的区别与联系。

Raft 协议和 Paxos 协议是两种常见的一致性协议，用于解决分布式系统中的一致性问题。

区别

可理解性和易用性：Raft 协议相对于 Paxos 协议来说更易理解和实现。

领导者选举：在 Raft 协议中，领导者选举是一个重要的组件，它通过心跳机制和超时来实现。只有领导者才能接收客户端请求并将其复制到其他节点。而在 Paxos 协议中，没有明确的领导者角色，所有节点都有相同的角色，需要通过投票和多个阶段的消息交换来达成一致。

日志复制：在 Raft 协议中，日志复制是通过领导者将日志项发送给其他节点进行复制。领导者会等待大多数节点的确认后再提交日志项。而在 Paxos 协议中，通过提议和接受阶段来实现日志复制，每个节点都可以提议和接受日志项。

联系：

一致性保证：无论是 Raft 协议还是 Paxos 协议，它们的目标**都是实现分布式系统中的一致性**。它们通过协议规定的消息交换和状态机复制来保证所有节点在最终达成一致的值或状态。

多数派原则：Raft 协议和 Paxos 协议**都依赖于多数派原则来保证一致性**。在 Raft 中，大多数节点的确认是提交决策的条件；在 Paxos 中，多数派节点的接受是达成一致的条件。

故障容错：无论是 Raft 协议还是 Paxos 协议，它们**都考虑了节点故障和网络分区等故障情况**。它们都通过选举新的领导者或重新协商达成一致性，以保持系统的可用性和一致性。

19. 计算机系统有 9 个 CPU 和 18 GB 内存，用户 A 的每个任务都请求（1CPU，4GB）资源；用户 B 的每个任务都请求（3CPU，1GB），根据主资源公平分配算法，写出资源的分配过程。
20. 集群中总的资源量为 16 CPU，36 GB 内存，3GB/s 网络。现在有 A、B、C 三个 MapReduce 作业，A 有 16 个 map 任务和 3 个 reduce 任务，B 和 C 都有 8 个 map 任务和 3 个 reduce 任务。A 的每个 map 任务的资源需求是<1 CPU, 2GB 内存>，B 和 C 每个 map 任务的资源需求都是<2CPU, 1GB 内存>。A、B、C 的 reduce 任务都只需要 1GB/s 的网络带宽。请分别绘出用主资源公平调度算法和多资源打包调度算法的示意图。
21. 并行任务在分布式缓存系统中调度的基本原则是？存储优化中解决小文件问题的方法有哪些？

负载均衡：**将任务合理分配到不同的节点,避免某些节点过载而其他节点闲置。这可以提高整体系统的吞吐量。**

亲和性：**将相关的任务分配到同一个节点或集群,减少跨节点或跨集群的通信开销。**

数据本地性：**尽可能将任务分配到数据所在的节点,减少数据传输时间。**

容错性：**当某个节点出现故障时,能够快速重新调度任务到其他节点,避免影响整体系统的可靠性。**