1. 特征：数据体量大、数据流动性高、数据种类繁多、价值密度低、数据准确度低

（1）数据体量大：

**挑战：**体量的大幅增长意味着存储和处理数据需要**更多的资源**。传统的数据处理系统和存储设备**无法满足**大规模数据集的处理需求，导致处理**速度**变慢、**存储**成本增加以及数据**管理**复杂化。

**措施**：**分布式**存储系统，如HDFS、GFS、BigTable。**并行**处理技术，如MapReduce模型。用数据**压缩**、归约和有效的数据组织方式来减少存储需求。

（2）数据流动性高：

**挑战：**许多数据的价值在于**时效性**，数据的高速**流动**和快速**动态变化**要求系统能够**实时**或近实时地**处理**和**分析**数据。这就需要系统具备高效的数据摄取能力，以及快速的数据处理和分析能力。

**措施**：采用**流处理**技术，如Apache Storm、Spark Streaming、Apache Flink等。实现对实时生成的数据进行快速处理。还可以通过优化**数据传输**和**处理流程**，以及使用高效的**数据缓存**策略来提高数据处理速度。

（3）数据种类繁多：

**挑战：**数据种类的多样性带来了数据的异构性，包括**结构化**数据、**半结构化**数据和**非结构化**数据。这增加了存储和处理的难度，对数据的处理能力也提出了更高的要求。

**措施**：采用更灵活的数据模型，如**NoSQL**数据库，从而实现存储和处理各种格式的数据。此外，使用数据融合技术整合来自不同来源的数据，以及利用数据挖掘和机器学习技术来分析和提取有价值的信息。

2.

**原因**：数据一致性问题主要源于副本之间的**更新同步**。当系统中的数据副本分布在不同的节点上时，如果一个节点上的数据副本被更新，系统需要确保其他节点上的副本也能及时更新，以保持数据的**一致性**。如果更新**不能及时同步到所有副本**，那么系统可能面临数据不一致的风险，即不同的用户或进程可能读取到不同版本的数据，这可能导致错误的数据读取或数据过时。

**措施：**

**（1）复制协议：**定义**强同步复制**（**所有副本上同步完成**才确认更新成功）和**异步复制**（立即返回成功，然后再**逐步同步**到其他副本），适应不同场景需求。

**（2）读写一致性模型**：定义不同的一致性模型，如**强一致性、弱一致性、最终一致性**等，以适应不同的应用场景需求。

**（3）选举协议**：当主副本的节点出现故障时，由**分布式协议**选举出一个新的主副本，如**Paxos**协议，通过多数派投票机制确保副本之间的一致性。

**（4）HDFS**实例：一致性是在**写数据阶段完成**，采用“一次写入，多次读取”的一致性模型。

**（5）Dynamo**实例：提供**最终的一致性**，即允许更新异步地传播到所有副本中，引入**向量时钟**解决冲突，捕捉同一对象不同版本之间的因果关系。

**（6）其他技术**：如**版本控制、日志、检查点、智能的数据同步策略**等。

3. **原则：**

**放置策略应能提高数据可靠性、可用性，提高网络带宽利用率。**

（1）**跨机架放置**：为了提高容错性，同一文件块的不同副本的存储于**不同机架和不同节点**上，以减少机架故障对数据可用性的影响。

（2）**数据均匀分布**：为了实现负载均衡，数据应该均匀地分布在集群的各个节点上，避免某些节点过载而其他节点资源闲置。

（3）**容错和数据可靠性**：通过在不同的节点上存储数据的多个副本，可以提高数据的可靠性和容错能力。即使某个节点发生故障，数据也不会丢失。

（4）**网络带宽优化**：在设计副本放置策略时，应考虑网络拓扑结构，以减少网络拥塞和提高数据传输效率。

**放置策略：**

（1）**任何数据节点都不包含任多块的多个副本**

（2）机架充足时：创建一个新文件块时，HDFS将一个副本放在一个节点，第二个和第三个副本放在不同的机架的不同节点，其余更多副本放在另一个节点

（3）机架不足时：第二和第三个副本会放在同一机架的不同节点上。

4. CAP原理指出一个分布式系统不可能同时满足以下三个特性：

（1）**一致性**：分布式系统涉及多个分布在不同节点上的数据副本，一致性要求分布的各数据**副本保持一致**。

（2）**可用性**：指系统始终处于**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.

在分布式系统中，分区容错性**（P）是必选项**。因此在分布式环境中，CAP理论更合适的描述是：**在满足分区容错的前提下，没有算法能同时满足数据一致性和服务可用性。**

（这意味着系统设计者需要根据具体的业务需求和场景，选择牺牲某特性以满足另一个特性。例如，一些系统可能更重视数据的一致性，而另一些则可能更注重系统的可用性和响应速度。）

5. 向量钟用于处理Dynamo的数据版本**冲突**，保证**数据一致性**；Merkle树作为**容错机制**，用于处理Dynamo**永久故障**时的数据同步和故障恢复。

**向量钟**是一种用于捕捉分布式系统中同一对象不同版本之间**因果关系**的机制。它是一个（节点，计数值）的链表，关联每个对象的每个版本。每个节点在进行写操作时会更新计数值，并将其与自己的标识符一起加入到向量钟中。当发生数据更新时，向量钟会随着更新一起传播到其他节点，从而帮助系统跟踪不同数据版本之间的先后顺序和依赖关系。向量钟的主要作用是解决分布式系统中的数据版本控制问题，通过比较向量钟，系统可以判断出不同数据副本的更新顺序，从而解决更新冲突，并保证数据的最终一致性。

**Merkle树**是一种哈希树的数据结构，用于验证分布式系统中的数据**完整性和一致性**。在Merkle树中，每个叶节点为其文件内容的哈希值，非叶节点则为其所有子节点的哈希值，任何一个数据块的更改都会导致从该数据块到根节点的哈希值发生变化。Merkle树提供了一种高效的方式来检查分布式存储中的数据是否被篡改，同时它也支持对数据的增量更新和同步。Dynamo出现永久故障时就需要借助Merkle树机制从其他副本进行数据同步。

6.

**特性**：**非易失**性、存储**密度高**、字节可寻址、可原位更新、能耗低、读写不对称、寿命有限。

**原因**：NVM访问速度低，寿命短，但非易失，存储密度高，成本低；DRAM易失，存储密度小，可扩展性低，成本高，但访问速度快，寿命长。因此设计DRAM+NVM的异构内存系统，既能利用NVM扩大主存容量，又能依靠DRAM降低访问速度与使用寿命上的损失。利用局部性特征和优化手段，理想情况下能够构建出一个近似DRAM的性能与NVM容量的内存系统。

7.

**内存计算**是（以大数据为中心，依托计算机硬件的发展、依靠新型的软件体系结构，通过对计算机系统结构、系统软件、编程模型等进行重大革新），将数据**装入内存**中处理，进行**避免I/O操作**的一种新型的以数据为中心的并行计算模式。

大数据处理以数据为中心，需要极高的数据传输及处理速率，数据的存储与传输取代了计算任务成为新的核心。内存计算减少了磁盘数据访问，降低了网络延迟于磁盘I/O的影响，大幅提升数据吞吐量与数据处理速度，成为海量数据分析的利器。

8.

Dynamo依赖**一致性哈希**算法进行数据分区。哈希函数的输出为一个连续的循环空间或“环”，每个节点在环上被分配一个随机的哈希值，代表其在环上的位置。数据项通过哈希键来确定它在环上的位置，然后系统会沿着环顺时针寻找第一个位置值大于数据项哈希值的节点，该节点即为数据项负责的节点。

为了解决数据和负载分布不均匀的问题，以及忽略节点性能异质性的问题，Dynamo采用了**改良**的一致性哈希算法，引入了“**虚拟节点**”的概念。每个物理节点可以拥有多个虚拟节点，这些虚拟节点在环上有不同的位置。当一个节点加入系统时，它会被分配多个虚拟节点，从而在环上拥有多个位置，一个物理节点可以管理多个虚拟节点，增加了系统的灵活性和可扩展性。

9.

**定义**：函数式编程是一种编程模型，它将计算看作是数学中函数的计算，并且避免了状态和变量的概念。函数式编程语言是一类与命令式编程语言风格完全不同的编程语言，具有更强的数学表达性。它将计算视为函数的计算，由函数定义和调用构成计算程序，其理论基础是λ演算，而且λ演算的函数可以接受函数当作输入和输出。

**特点：**透明性，高阶函数，惰性求值与并行，递归调用及其优化，模式匹配

10.

**透明性**：程序可自由的将（子）表达式替换为它的值而不改变程序（表达式）。

**高阶函数**：函数可以作为参数传递给其他函数，或作为结果返回，使得函数能够操作其他函数。

**函数柯里化**：是指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**：调用函数时，不是盲目地计算所有实参的值然后再进入函数体，而是先进入函数体，只有当需要实参值时才计算所需的实参值，其目的是尽可能避免无用计算。

11.

**相同点**：

（1）**分布并行**：都是分布式计算框架，支持并行处理大量数据，通过将任务分配到多个计算节点上以提高处理速度，适用于大规模数据处理。

（2）都采用了MapReduce编程模型，包含Map（处理输入）和Reduce（汇总）两个阶段

（3）**简化模型**：都简化了编程模型，使得开发者能够专注于数据处理逻辑，而不必处理底层的分布式计算细节。

（4）**容错**：都设计有容错机制，能够在节点故障时重新分配任务，确保计算过程的连续性。

（5）**扩展性**：它们都能够在从几十到几千个节点的集群上运行，有高扩展性。

**不同点**：

（1）**实现语言**：Google MapReduce是用C++实现的，而Hadoop是用Java实现的。

（2）**开源性**：Hadoop是开源的，可以运行在多种硬件和操作系统平台；Google MapReduce闭源，只能运行在专有硬件和软件平台上。Hadoop的开源性质使得它得到了更广泛的社区支持和应用，拥有一个更为丰富的生态系统，如HDFS、HBase、Hive等，构成了一个完整的数据处理平台。而Google MapReduce主要作为一个计算框架存在。

（3）**API和编程模型**：Hadoop提供了更丰富的API和编程模型，支持Java以外的其他语言（如Scala、Python等）；而Google MapReduce主要支持C++，API和功能封闭，不利于外部拓展。

（4）**存储系统**：Hadoop采用HDFS分布式文件系统；Google的采用GFS分布式文件系统。

12.

def mapper(key, value):

# 初始化大小为100的优先队列

min\_heap = MinHeap(size=100)

for number in value:

# 如果堆未满，或当前数大于堆顶元素，则加入堆中

if min\_heap.size() < 100 or number > min\_heap.peek():

if min\_heap.size() == 100:

# 如果堆已满，移除堆顶元素

min\_heap.pop()

min\_heap.push(number)

# 输出局部最大的100个数

for number in min\_heap.get\_elements():

yield None, number

def reducer(key, values):

# 初始化大小为100的优先队列

min\_heap = MinHeap(size=100)

for number in values:

# 如果堆未满，或当前数大于堆顶元素，则加入堆中

if min\_heap.size() < 100 or number > min\_heap.peek():

if min\_heap.size() == 100:

# 如果堆已满，移除堆顶元素

min\_heap.pop()

min\_heap.push(number)

# 输出全局最大的100个数

result = []

while not min\_heap.is\_empty():

result.append(min\_heap.pop())

13.

**至多一次**：指系统能够保障在处理数据时，任意元组**要么刚好被处理一次，要么被丢弃**，不会出现对同一元组的重复处理（统计分析）

**至少一次**：指系统能够保障处理数据时，任意元组都**至少被处理一次不能丢失，但是允许同一元组被重复处理**（安全监测）

**恰好一次**：指系统能够保障数据处理时，所有元组**都恰好被处理一次**，不存在元组被重复处理或者被丢弃的情况（金融交易）

14.

**Apache Storm**的特点：

**实时性**：实时计算系统，能够保证数据的低延迟处理。

**容错性**：具有很好的容错性，能够在节点故障时重新分配任务。

**简单性**：API相对简单，易于编程和实现流处理应用。

**通用性**：支持多种编程语言，易于部署和管理；可以应用于多种场景，包括实时分析、在线机器学习、连续计算等。

**高扩展、保障数据被处理**。

**Spark Streaming**的特点：

**微批处理**：基于Spark的微批处理模型，将数据流分割成一系列小批次进行处理。

**高吞吐量**：能够提供高吞吐量的数据处理能力。

**易用性**：提供了丰富的API，包括复杂的转换操作，使得开发更加灵活和方便。

**集成性**：作为Spark生态系统的一部分，Spark Streaming能够与Spark的其他组件（如Spark SQL、MLlib）无缝集成。

**\*容错性**：通过RDD的 lineage信息实现容错，能够从失败中恢复并重新计算。

**可扩展、支持多种数据源、多种输出、应用场景丰富**

**区别**：

**处理模型**：Storm是一个纯粹的流处理系统，而Spark Streaming是基于微批处理的流处理系统。

**延迟**：Storm的延迟可以更低，因为它是实时处理数据；Spark Streaming的延迟相对较高，因为它需要等待数据积累到一定量后再进行处理。

**吞吐量**：尽管两者都设计为高吞吐量系统，但Spark Streaming由于其微批处理的特性，可能在某些场景下提供更高的吞吐量。

**编程模型**：Storm使用spout和bolt的概念来构建拓扑，而Spark Streaming使用Spark的RDD和DAG模型。

**容错机制**：Storm通过ack机制来保证消息处理的可靠性，Spark Streaming则依赖于RDD的 lineage信息来恢复失败的任务。

15.

**以点为中心**的编程模型：

**优点**：每个顶点独立计算，容易实现**扩展**和**细粒度**的并行；容易**表达和实现**图算法，如**PageRank**和各种基于顶点属性计算的算法。

**缺点**：存在大量的**随机访存**操作，导致性能问题，时间开销大；需要对原始的图数据进行排序等**预处理**操作，这也增大总的时间开销。

**以边为中心**的编程模型：

**优点**：通过边来传递信息，可以实现**负载均衡**和**细粒度**的并行；**顺序访问**边，减少了随机访存带来的开销；不需要对原始的图数据进行排序等预处理操作，从而降低了时间开销。

**缺点**：可能存在边的**冗余**计算，尤其是在稀疏图中；存在对点的大量随机访问，激活点比较少时适合以点为中心

**以路径为中心**的编程模型：

优点：采用**顺序访问**，访存效率高，**IO负载低**，图局部更新**收敛性好**；适合顺序处理大规模图数据；**简化了迭代**计算的步骤，加快算法的执行速度

缺点：实现复杂，需要维护路径信息；对于非路径相关的算法可能不够高效。

**以子图为中心**的编程模型：

优点：通过局部更新子图来**促进全局收敛**，减少迭代次数；适合处理**大规模**图数据，可以有效地处理子图级别的任务

缺点：并不适用于所有算法；需要对图进行划分，这可能增加**预处理**的复杂性和通信开销。

16.采

**初始化**：

对于每个顶点v ∈ V：

如果v是源点：

distance[v] = 0

否则：

distance[v] = ∞

**Scatter**阶段：

对于每个顶点v ∈ V：

如果distance[v] < ∞：

对于每个邻接点u ∈ Adj[v]：

如果distance[u] > distance[v] + weight(v, u)：

distance[u] = distance[v] + weight(v, u) 发送消息

**Gather**阶段：

对于每个顶点v ∈ V：

接收消息，更新distance[v]为所有收到的新距离值中的最小值

**重复**Scatter和Gather阶段直到所有顶点的距离不再变化。

最后一部分课本表述：

whie not done

for all vertices v that need to scatter updates

vertex\_scatter(v)

for all vertices v that have updates

vertex\_gather(v)

17.

**顺序处理**：对于高维度节点，顺序处理可能会非常低效，因为它需要遍历所有的边。这在节点度数非常高时尤其成问题，会导致计算瓶颈。

**发送消息**：处理高维度节点时，单个工作节点可能需要向大量邻居节点发送消息，这会导致网络拥塞和延迟。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**：可能需要访问图的大部分来获取状态信息，这在大规模图中会导致性能问题，尤其是当图存储在单个机器上时，边的元数据可能过大。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** ：可能需要大量的**锁**来**同步状态**，这可能导致**竞争条件和一致性问题**。

**共享状态**：**同步执行**：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。

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

**Coarsening（粗化）**:

意义: Coarsening是将图逐渐简化的过程，通过合并节点来减少图的大小，从而简化问题的复杂度。在多层次图划分中，Coarsening有助于创建一个更小、更易于管理的图，以便进行初始划分。

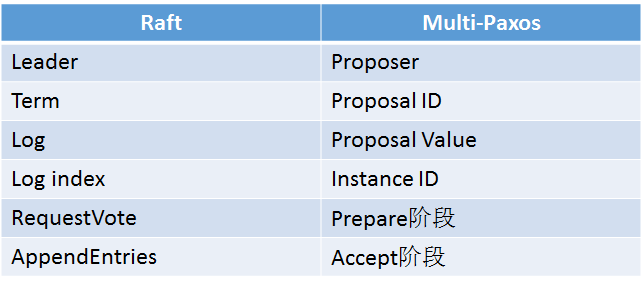
**Uncoarsening（细化）**:

意义: Uncoarsening是Coarsening的逆过程，它将简化后的图逐步恢复到原始大小。在多层次图划分中，Uncoarsening有助于细化初始划分的结果，通过迭代优化来改进图的划分质量。

18. 区别：（这部分答案不一定对，请自行对照课本完善）



联系：都是基于领导者的一致性算法。拥有许多相似的概念，比如Raft中的Leader与Paxos中的Proposer

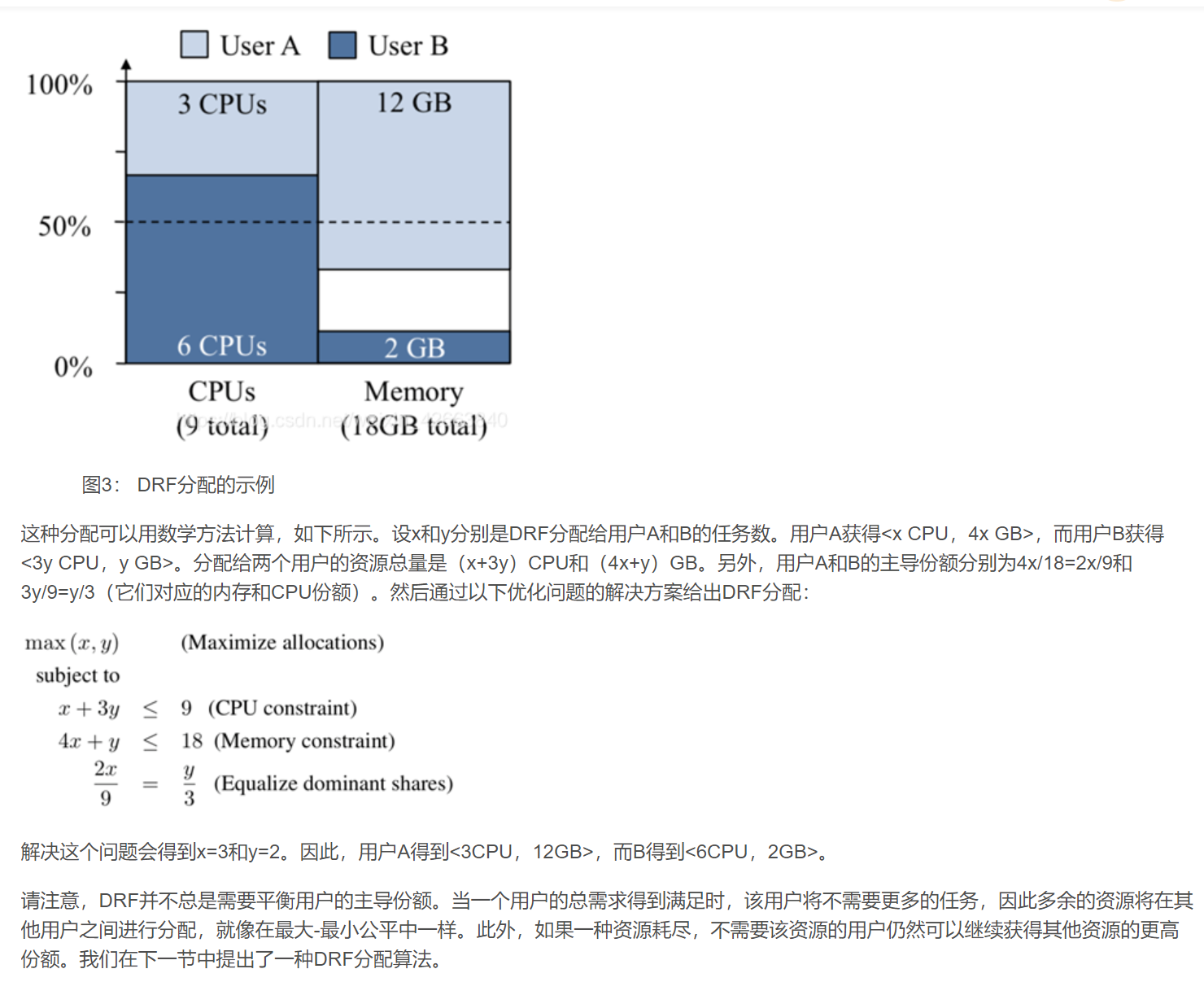


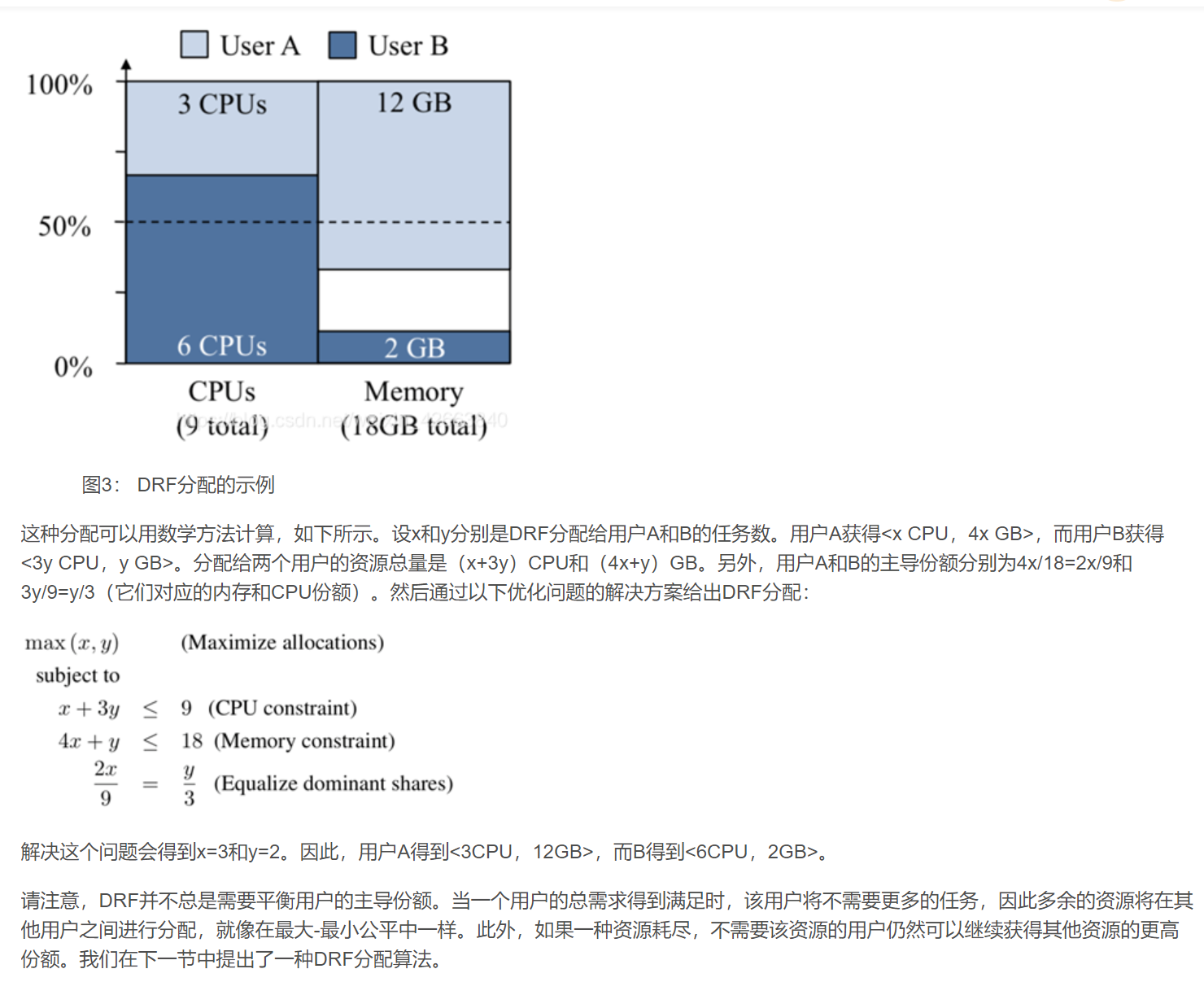
19.

主资源公平分配算法：DRF计算分配给该用户的每个资源的份额。用**户所有份额中的最大值称为该用户的主导份额**，与主导份额相对应的资源称为主导资源。

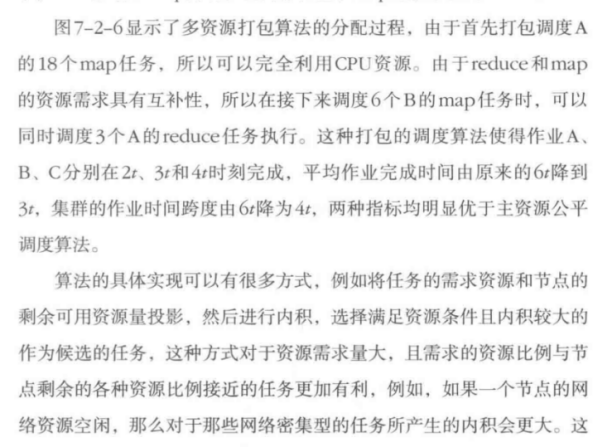
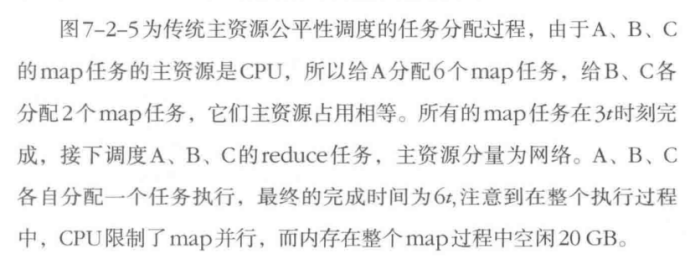
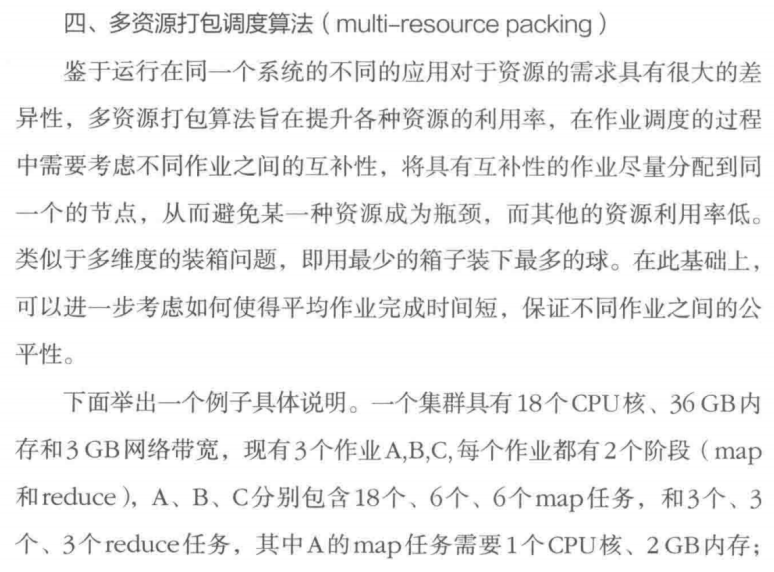
首先确定主资源，对A而言 CPU 1/9 <4/18内存 所以对A而言内存为主资源

对B而言 CPU 3/9>1/18内存，所以对B而言CPU为主资源





20.



21.

**数据本地化**：通过将任务调度到输入数据所在的计算节点，尽量让数据在本地计算，减少任务执行中的数据网络传输开销。

**负载均衡**：通过被动解决或主动预防，解决数据倾斜和计算倾斜导致的复杂均衡问题。

**Straggler（拖后腿问题）**优化：为短板任务启动推测任务，推测任务通常分发到其他节点上，让该备份任务与原始任务同时处理一份数据，将先运行完的结果作为最终结果，并在运行完成后杀死掉另一个任务，解决拖后腿问题。

方法：

（1）针对数据自身的组织方式，使用**分层式**文件存储格式， 如Hadoop archive（HAR文件），将一组小文件合并到单个物理文件中存储，减少系统中物理文件的数量。

（2）针对存储系统自身的设计和实现，使用**多个主控节点**来分担元数据的存储和服务负载，提升系统服务扩展能力。

（3）采用**序列文件**格式，将小文件存入序列文件中，并在序列文件上以流方式处理，降低主控节点的计算负载，但小文件的逻辑存储层对于应用不透明。