# 大数据处理综合实验

**一、题型**

填空（基本概念，不会偏）、简答（对系统的熟悉程度）和程序设计（思路+编程、注释、关键类名，关键函数，Driver 要求写，例如改写 inputformat 要在 Driver 里面设置）

考试内容不会脱离 PPT、课上讲课内容、实验内容

**第一章 介绍**

1. 提高计算机性能的主要手段
   a) 四个空：提高处理器字长；提升流水线等微体系结构技术；提高集成度；提高处理器频率
   b) 三个空：提高处理器字长和流水线等微体系结构技术；提高集成度；提高处理器频率
2. 为什么单核处理器性能提升接近极限：VLSI 集成度不可能无限制提高；处理器的指令级并行度提升接近极限；处理器速度和存储器速度差异越来越大；功耗和散热大幅增加超过芯片承受能力
3. 为什么需要并行计算？单核处理器性能提升接近极限；单处理器向多核、众核并行计算发展成为必然趋势；应用领域计算规模和复杂度大幅提高
4. **并行计算技术的分类**
   a) 数据和指令处理结构分类（弗林分类）：单指令单数据流 SISD、单指令多数据流 SIMD、多指令单数据流 MISD、多指令多数据流 MIMD
   b) 并行类型分类：位级并行、指令级并行、线程级并行（数据级并行和任务级并行）
   c) 存储访问结构分类：共享内存、分布共享存储体系结构、分布式内存
   d) 系统类型分类：多核/众核并行计算系统 MC、对称多处理系统 SMP、大规模并行处理 MPP、集群和网格（从前到后耦合度越来越松散，可拓展性和能耗越来越高）
   e) 计算特征：数据密集型并行计算（大规模 Web 信息检索）、计算密集型并行计算（3D 建模渲染、气象预报、科学计算）、数据密集与计算密集混合型并行计算（3D 电影渲染）
   f) 并行程序设计模型/方法分类：共享内存变量（pthread 和 OpenMP）、消息传递方式（MPI）、MapReduce 方式
5. 并行计算的主要技术问题：多核/多处理器网络互连结构技术、存储访问体系结构、分布式数据与文件管理、并行计算任务分解与算法设计、并行程序设计模型和方法、数据同步访问和通信控制、可靠性设计与容错技术、并行计算软件框架平台、系统性能评价和程序并行度评估。
   a) 程序并行度评估（要求掌握）：
      i. **Amdahl 定律**$S=\frac{1}{(1-P)+\frac{P}{N}}$，S 是加速比，P 是程序可并行比例，N 是处理器数目
      ii. 一个并行程序可加速程度是有限制的，并非可无限加速，并非处理器越多越好
6. MPI 的特点
   a) 灵活性好，适合于各种计算密集型的并行计算任务
   b) 独立于语言的编程规范，可移植性好
   c) 有很多开放机构或厂商实现并支持
7. MPI 的不足
   a) 无良好的数据和任务划分支持
   b) 缺少分布文件系统支持分布数据存储管理（MPI 只负责消息传递）

c) 通信开销大，当计算问题复杂、节点数量很大时，难以处理，性能大幅下降
d) 无节点失效恢复机制，一旦有节点失效，可能导致计算过程无效
e) 缺少良好的构架支撑，程序员需要考虑以上所有细节问题，程序设计较为复杂

8. 大数据的特点：4V+1C，Volume 大容量、Variety 多样性、Velocity 时效性、Veracity 准确性、Complexity 复杂性
9. 大数据的类型：
   a) 结构特征分类：结构化、非结构化、半结构化数据
   b) 获取和处理方式：动态/实时数据、静态/非实时数据
   c) 关联特征：无关联/简单关联数据（键值记录型数据）、复杂关联数据（图数据）
10. 大数据研究的基本途径：寻找新算法降低计算复杂度；寻找和采用降低数据尺度的算法；分而治之的并行化处理
11. 大数据研究的变化：从 MapReduce 等演变成 Hadoop 与 Spark 主流的并行计算模型与系统。
12. MapReduce 的定义（要求掌握）：**MapReduce 是面向大规模数据并行处理的基于集群的高性能并行计算平台、并行程序开发与运行框架、并行程序设计模型与方法**
    **a)** 对付大数据并行处理：**分而治之**
       i. Master：负责划分和分配任务；Worker：负责数据块计算
    b) 上升到抽象模型：用 **Map** 和 **Reduce** 两个函数提供了高层的并行编程抽象模型和接口
       i. Map：对一组数据元素进行某种重复处理；Reduce：对 Map 的中间结果进行某种进一步的结果整理
       ii. 注意：进行 Reduce 处理之前，必须等到所有的 Map 函数做完，要求进入 Reduce 之前有一个同步障（barrier）
    c) 上升到构架：以统一构架实现**自动并行化计算**，为程序员**隐藏**绝大多数**系统层细节**。完成以下系统底层相关处理：
       i. 计算任务的划分和调度；数据的分布存储和划分；处理数据与计算任务的同步；结果数据的收集整理(sorting, combining, partitioning)；系统通信、负载平衡、计算性能优化处理；处理系统节点出错检测和失效恢复
13. MapReduce 提供的主要功能：
    a) 五个空：数据划分和计算任务调度、数据/代码互定位、出错检测和恢复、分布式数据存储与文件管理、Combiner 与 Partitioner
    b) 四个空：数据划分和计算任务调度、数据/代码互定位、出错检测和恢复、系统优化
14. **MapReduce 的主要设计思想与特点：**
    a) **向“外”横向拓展，而非向“上”纵向拓展**：MapReduce 集群的构筑选用价格便宜、易于扩展的大量低端商用服务器，性价比高
    b) **失效被认为是常态**：MapReduce 并行计算软件框架使用了多种有效的机制，如节点自动重启技术，使集群和计算框架具有对付节点失效的健壮性，能有效处理失效节点的检测和恢复；
    c) **把处理向数据迁移**：MapReduce 采用了数据 代码互定位的技术方法，计算节点将首先将尽量负责计算其本地存储的数据 以发挥数据本地化特点( 仅当节点无法处理本地数据时，再采用就近原则寻找其它可用计算节点，并把数据传送到该可用计算节点。
    d) **顺序处理数据、避免随机访问数据**：MapReduce 设计为面向大数据集批处理的并

行计算系统，所有计算都被组织成很长的流式操作，以便能利用分布在集群中大量节点上磁盘集合的高传输带宽。

e) **为应用开发者隐藏系统层细节**：MapReduce 提供了一种抽象机制将程序员与系统层细节隔离开来，程序员仅需描述需要计算什么 (what to compute)，而具体怎么去做 (how to compute) 就交由系统的执行框架处理，这样程序员可从系统层细节中解放出来，而致力于其应用本身计算问题的算法设计

f) **平滑无缝的可扩展性（数据扩展性和系统规模扩展性）**：基于 MapReduce 的计算性能可随节点数目增长保持近似于线性的增长

15. 流式大数据问题的特征，ab：Map，de：Reduce
    a) 大量数据记录/元素进行重复处理
    b) 对每个数据记录/元素作感兴趣的处理，获取中间结果信息
    c) 排序和整理中间结果以利后续处理
    d) 收集整理中间结果
    e) 产生最终结果输出

**第二章 MapReduce、HDFS**

1. **Google MapReduce 的基本工作过程**
    a) **待处理的大数据，被划分为大小相同的数据块(如 64MB)，及与此相应的用户作业程序；**
    b) **系统中有一个负责调度的主节点（Master)，以及数据 Map 和 Reduce 的工作节点（Worker)；**
        i. **Hadoop MapReduce 中 Master 改成 JobTracker，Worker 改成 TaskTracker！**
    c) **用户作业程序提交给主节点；**
    d) **主节点为作业程序寻找和配备可用的 Map 节点，并将程序和作业传送给 Map 节点；**
    e) **主节点也为作业程序寻找和配备可用的 Reduce 节点，并将程序传送给 Reduce 节点；**
    f) **主节点启动每个 Map 节点执行程序，每个 Map 节点尽可能读取本地或本机架的数据进行计算**
    g) **每个 Map 节点处理读取的数据块,并做一些数据整理工作，将中间结果存放在本地，同时通知主节点计算任务完成并告知中间结果数据存储位置；**
    h) **主节点等所有的 Map 节点计算完成后，开始启动 Reduce 节点执行，Reduce 节点从主节点所掌握的中间结果数据位置信息，远程读取这些数据；**
    i) **Reduce 节点计算结果汇总输出到一个结果文件，即获得整个处理结果。**
2. 为什么中间结果要写到本地磁盘？因为 1. 为了缓冲，避免网络故障；2. Reduce 的时候如果 Worker 崩了，但本地磁盘仍然有备份，就不需要 Map 重新计算了。
3. **MapReduce 的失效处理与性能优化方面的主要措施：**
    a) **失效处理：**
        i. **MapReduce 在数据存储方面采用多副本存储设计，即同一个数据块的多个副本存放在不同节点上：这加快传输速度，可以帮助判断网络传输是否出错，还可以保证某个 DataNode 失效的情况下不会丢失数据；**
        ii. **主节点失效：主节点中会周期性地设置检查点(checkpoint)，检查整个计算作业的执行情况，一旦某个任务失效，可以从最近有效的检查点开始重新执行，避免从头开始计算的时间浪费；**
        iii. **工作节点失效：主节点会周期性地给工作节点发送心跳检测，如果工作节点没**

**有回应，这认为该工作节点失效，主节点将终止该工作节点的任务并把失效的任务重新调度到其它工作节点上重新执行；**

**b) 性能优化：**

i. **带宽优化：增加 Combiner 类，减少网络通信带宽；**

ii. **计算优化：把一个计算任务同时让多个 Map 节点做，取最快完成者的计算结果**

iii. **数据相关性问题（需要把属于同一个 Reduce 节点的数据归并到一起）：根据一定策略对 Map 输出的中间结果进行数据分区 Partition，减少传输到 Reduce 节点上的数据相关性。**

4. Google GFS 的基本设计原则：廉价本地磁盘分布存储、多数据自动备份解决可靠性、为上层的 MapReduce 计算框架提供支撑
5. GFS Master：保存命名空间（整个分布式文件系统的目录结构）、Chunk 与文件名的映射表、Chunk 副本的位置信息（每个 Chunk 默认有 3 个副本）三种元数据
   a) Master 失效时，只要 ChunkServer 数据保存完好，可迅速恢复 Master 上的元数据
6. GFS ChunkServer：用来保存大量实际数据的数据服务器
7. GFS 数据访问工作过程
   a) 程序运行前，数据已经存储在 GFS 文件系统中。程序实行时应用程序会告诉 GFS Server 所要访问的文件名或者数据块索引是什么；
   b) GFS Server 根据文件名会数据块索引在其文件目录空间中查找和定位该文件或数据块，并找数据块在具体哪些 ChunkServer 上；将这些位置信息回送给应用程序；
   c) 应用程序根据 GFS Server 返回的具体 Chunk 数据块位置信息，直接访问相应的 Chunk Server；
   d) 应用程序根据 GFS Server 返回的具体 Chunk 数据块位置信息直接读取指定位置的数据进行计算处理。
8. GFS 数据访问工作过程的特点：1. 应用程序访问具体数据时不需要经过 GFS Master。因此避免了 Master 成为访问瓶颈；2. 由于一个大数据会存储在不同的 ChunkServer 中，应用程序可实现并发访问
9. GFS 的系统管理技术：大规模集群安装技术、故障检测技术、节点动态加入技术、节能技术
10. BigTable：结构化数据存储和访问管理系统
    a) 主要动机：需要存储多种数据、海量的服务请求、商用数据库无法适用
    b) 主要设计目标：广泛的适用性、很强的拓展性、高吞吐量数据访问、高可用性和容错性、自动管理能力、简单性
11. BigTable 主要是一个分布式多维表，表中数据通过**行关键字、列关键字和时间戳**进行索引和查询定位；
12. BigTable 中的数据一律视为字节串；
    a) 行：字节串，按字节序排，水平方向上可分为多个子表
       i. URL 地址倒排的好处：1. 同一地址的网页将被存储在表中连续的位置,便于查找；2. 倒排便于数据压缩,可大幅提高数据压缩率
    b) 列：列关键字组织为列族，列关键字可表示为“族名：列名”
       a) 按列族存储的原因是不太可能跨列族访问，如个人基本信息(姓名、身份证号、身高)，成绩(语文、数学、英语)，和个人特质(兴趣、特长)可以分为 3 个列族存储
    c) 时间戳

13. BigTable 基本构架
    a) 主服务器：新子表分配、子表监控、元数据操作、负载均衡；
    b) 子表服务器：数据存储和访问操作
        i. 基本存储结构 SSTable，存储在 GFS 文件系统中。一个 SSTable 对应 GFS 一个 64MB 的数据块，一个 SSTable 里面可以划分 64KB 的子块，还有 Index 索引。
        ii. 子表寻址：3 级 B+树
14. **Hadoop 系统：**
    a) **NameNode（对应 MasterServer）：分布存储的主控节点，用以存储和管理分布式文件系统的元数据**
    b) **DataNode（对应 ChunkServer）：实际存储大规模数据的从节点**
    c) **JobTracker（对应 Master）：并行计算框架的主控节点，用以管理和调度作业执行**
    d) **TaskTracker（对应 Worker）：管理每个计算从节点上计算任务的执行**
15. 为了实现 Hadoop 系统中本地化计算的原则，DataNode 和 TaskTracker 将合并设置
16. 为什么需要在 TaskTracker 里面建立 child JVM？用户程序可能有 bug，这么做可以方便报错且维持了 TaskTracker 持续运行；
17. InputFormat：定义了数据文件如何分割和读取
    a) 选择文件或其它对象，用来作为输入
    b) 定义 InputSplits，将一个文件分为不同任务
        i. InputSplit 定义了输入到单个 Map 任务的输入数据
    c) 为 RecordReader 提供一个工厂，用来读取这个文件
        i. RecordReader 定义了如何将数据记录转化为一个(key, value)对的详细方法，并将数据记录传给 Mapper 类
18. 一个作业的 map 任务数量由 split 数量决定，split 数量则由 InputFormat 决定
19. **Combiner 作用：合并相同 key 的键值对，减少中间结果数据网络传输开销；**
    a) 执行时间结点：在 Map 过程处理结束后，发送给 Reduce 节点前
20. **Partitioner 作用：决定一个给定的(key, value)对传给哪个 Reduce 节点，消除数据传入 Reducer 后不必要的相关性；**
    a) 如何避免在某些 Reducer 上聚集过多的数据而拖慢了整个程序：划分均匀（预读一小部分数据采样排序后划分）
    b) 当有大量的 key 要分配到多个 partition（也就是 Reducer）时，如何高效地找到每个 Key 所属的 partition：查找快速（构建高效的划分模型，按字节比较的则构建 Trie 树，否则二分查找确定 key 的区间）
    c) 执行时间结点：在 Map 过程完成计算之后，输出中间结果之前
21. 程序执行的容错处理：将失败的任务再次执行，TaskTracker 汇报给 JobTracker，最终由 JobTracker 决定
22. 计算性能优化：自动重复执行同一个任务，以最先执行成功的为准
23. HDFS 基本特征（估计也是要背的！）
    a) 大规模数据分布存储能力：可以将数据保存到大量节点中；不仅可存储很大的单个文件，还可以支持在一个文件系统中存储高达数千万量级的文件数量
    b) 高并发访问能力：以多节点并发访问方式提供很高的数据访问带宽，并且可以把带宽大小等比例拓展到集群中的全部节点上；
    c) 强大的容错能力：能在经常有节点发生硬件故障的情况下正确检测病自动恢复；为此 HDFS 采用多副本数据块形式存储（默认 3），按照块的方式随机选择存储节点；
    d) 顺序式文件访问：对顺序读优化，支持大量数据的快速顺序读出

i. 代价是随机访问负载较高；

e) 简单的一致性模型：一次写多次读，不支持更新，但支持尾部写入（因为要改的话三个节点要一起改，开销很大，容错性很低）

f) 数据块存储模式：大粒度数据块的方式存储文件（一个数据块默认 64MB，目的是减少寻址开销时间）

i. 减少元数据的数量

ii. 有利于顺序读写

iii. 允许将这些数据块通过随机方式选择节点，分布存储在不同的地方。

24. NameNode 保存文件系统的三种元数据：命名空间、数据块与文件名的映射表、每个数据块副本的位置信息。

25. NameNode 获得所需访问数据块的存储位置后直接访问 DataNode 上的数据的好处？

a) 可以允许一个文件的数据能同时在不同的 DataNode 上并发访问，提高数据访问的速度；

b) 大大减少 NameNode 的负担，避免 NameNode 成为数据访问瓶颈；

26. HDFS 基本文件访问过程：

a) 用户的应用程序通过 HDFS 客户端程序将文件名发送到 NameNode；

b) NameNode 接收到文件名之后，在 HDFS 目录中检索文件名对应的数据块，再根据数据块信息找到保存数据块的 DataNode 地址，将这些地址回送给客户端；

c) 客户端接收到这些 DataNode 地址后，与这些 DataNode 并行地进行数据传输擦欧总，同时将操作结果的相关日志提交到 NameNode

**27. HDFS 可靠性设计**

a) **多副本存储设计**：1 或 2 个不能保证数据对错，默认 3 个；

b) **安全模式启动**：与 DataNode 通信获取信息，检查数据块，退出安全模式的时候才进行文件操作等；

c) **SecondaryNameNode**：周期性备份 NameNode 的元数据，本身不处理任何请求；

d) **心跳包和副本重新创建**：保证 NameNode 和各个 DataNode 的联系；

e) **数据一致性**：数据校验和（校验和文件和文件本身保存在同一空间中）；

f) **租约**：防止多人写入的租约，需要恢复机制和租约时间限制防止 NameNode 或 DataNode 的崩溃所造成的影响；

g) **回滚**：需要回滚到旧版本解决升级导致的 bug 或不兼容问题。

**第三章 HBase 和 Hive**

1. HBase 设计目标：针对 HDFS 缺少结构化半结构化数据存储访问能力的缺陷，和传统数据库在容量和数据格式上都难以适应半结构化/非结构化大数据的处理，提供分布式数据管理系统，解决大规模的结构化和半结构化数据存储访问问题；

2. HBase 技术特点：

a) 列式存储：将所有记录的同一个列族下的数据集中存放；

b) 表数据是稀疏的分布式多维映射表，表中数据通过一个行关键字，一个列关键字和一个时间戳进行索引和查询定位，时间戳允许数据有多个版本；

c) 读写的严格一致性；

d) 提供很高的数据读写速度，为写数据进行了特别优化；

e) 良好的线性可拓展性：增加集群规模线性提高 HBase 的吞吐量和存储容量，服务器能动态加入和删除，自动调整负载平衡；

f) 提供海量数据存储能力；

g) 数据会自动分片，也可由用户控制分片；

h) 自动的失效检测和恢复能力，保证数据不丢失；
i) 提供了方便的与 HDFS 和 MapReduce 集成的能力，提供 Java API 作为主要的编程接口。

3. 数据模型区分：
   a) MapReduce 的数据模型：<key, value>；
   b) HBase 的逻辑数据模型：分布式多维映射表；
   c) HBase 的物理数据模型：按列存储的稀疏行/列矩阵（格式上按逻辑模型的行分割，按照列存储，值为空的不存储）；
4. HBase 基本组成结构：
   a) HBase Master：HBase 集群的主控服务器，负责集群状态的管理维护；
   b) HBase Region Server：子表数据区服务器，是 HBase 具体对外提供服务的进程；
      i. 大表被分为很多子表 Region，每个子表存储在一个子表服务器 Region Server 上
         1. 不保证一个 Region 一定在某一个机器上，但我们知道怎么查它；
      ii. 每个子表的数据区 Region 由很多个数据存储块 Store 构成
      iii. 每个 Store 由存放在内存中的 memStore 和存放在文件中的 StoreFile 构成；
5. HBase 数据访问：
   a) 当客户端需要进行数据更新时，先查到子表服务器 然后向子表提交数据更新请求。提交的数据并不直接存储到磁盘上的数据文件中，而是添加到一个基于内存的子表数据对 memStore 中，当 memStore 中的数据达到一定大小时，系统将自动将数据写入到文件数据块 StoreFile 中。每个文件数据块 StoreFile 最后都写入底层基于 HDFS 的文件中
      iv. 为什么加入到 memStore：加入内存缓存可以提升读写数据的速度；
      v. 如果断电导致数据丢了？Hlog 可以用于恢复；
      vi. 为什么写 HDFS 文件系统？多副本备份；
   b) 需要查询数据时，子表先查 memStore。如果没有，则再查磁盘上的 StoreFile。每个 StoreFile 都有类似 B 树的结构，允许进行快速的数据查询。StoreFile 将定时压缩。两个小子表可以合并，子表大到超过某个阈值的时候，可以分割成两个新的子表。
6. HBase 主服务器 HServer：
   a) 使用主表服务器 HServer 管理所有子表服务器，主表服务器维护所有子表服务器任何时刻的状态；
   b) 当一个新子表服务器注册时，主服务器让新的子表服务器装载子表；
   c) 若主服务器与子表服务器连接超时，那么子表服务器将自动停止，并重新启动；而主服务器则假定该子表服务器已死机，将其上的数据转移至其它子表服务器，将其上的子表标注为空闲，并在重新启动后另行分配使用。
7. HBase 数据记录的查询定位：描述所有子表和子表中数据块的元数据都存放在专门的元数据表中，并存储在特殊的子表中。子表元数据会不断增长，因此会使用多个子表来保存。所有元数据子表的元数据都保存在根子表中。主服务器会扫描根子表，从而得到所有的元数据子表位置，再进一步扫描这些元数据子表即可获得所寻找子表的位置。
8. HBase 三层索引保存 Region 位置：
   a) 通过 zookeeper 的文件获得.META.表的 ROOT Region 的位置；
   b) 通过 ROOT Region(.META.表的第一个 Region)查找.META.表中相应 Region 的位置；
   c) 通过.META.表找到所要查询的用户表相应 Region 的位置；

vii. .META.表的全部 Region 数据都会保存在内存中，从而提高访问.META.表的性能；
viii. HBase 客户端维护一些 cache，将已经寻址过的 Region 放入 cache 中，避免重新寻址和定位 Region 的操作

d) 总结：根子表－>用户表的元数据表－>用户表

9. HBase 修改表需要在 disable 状态下进行
10. Hive 简介：Hive 包括一个高层语言的执行引擎，类似于 SQL 的执行引擎；它建立在 Hadoop 的其它组成部分之上，包括 Hive 依赖于 HDFS 进行数据保存，依赖于 MapReduce 完成查询操作；
11. Hive 组成模块：
    a) HiveQL：Hive 的数据查询语言。Hive 提供这个数据查询语言与用户的接口，包括命令行接口 CLI、数据库访问编程接口 JDBC 与 Web 界面的网络接口；
    b) Driver：执行驱动程序。用以将各个组成部分形成一个有机的执行系统，包括会话的处理，查询获取以及执行驱动；
    c) Compiler：编译器，将 HiveQL 语言编译程中间表示。包括对 HiveQL 语言的分析、执行计划生成和优化等工作；
    d) Execution Engine：执行引擎，在 Driver 的驱动下具体完成执行操作。包括 MapReduce 执行、HDFS 操作或者元数据操作；
    e) Metastore：用以储存元数据。包括存储操作的数据对象的格式信息，在 HDFS 中的存储位置信息以及其它的用于数据转换的信息 SerDe 等
12. Hive 数据模型：
    a) 表 Table
        i. 逻辑上有两部分，分别是真实数据和描述表格中数据形式的元数据；
        ii. 物理上，每个表的数据将存储在一个 HDFS 目录下，描述表格中数据形式的元数据将存储在关系型数据库中；
        iii. 列有类型，可以是 int、float、string 等，也可以是复合类型如 list: map；
        iv. 储存在/home/hive/warehouse 目录下
            1. 数据表存储在 warehouse 的子目录中，Partition 和 Bucket 都在数据表的子目录下
            2. 数据可以以任意一种形式存储，如：使用分隔符的文本文件，或 SequenceFile，或使用用户自定义的 SerDe 任意格式文件。
    b) 外部表 External Table
        i. 创建表时默认表内数据管理由 Hive 负责，但是可创建外部表。外部表是已经存储在 HDFS 文件中，并具有一定格式的数据。即外部表意味着表内数据不在 Hive 的数据仓库内，它会到仓库目录以外的位置访问数据
    c) 分区 Partition：是一种根据“分区列”的值对表进行粗略划分的机制
        i. 作用是对表进行合理的管理以及提高查询效率；
    d) 桶 Bucket：在一定的范围内的数据按照 Hash 的方式进行划分
        i. 组织为桶的目的
            1. 为了取样更高效：处理大规模数据集时在开发、测试阶段把所有数据全部处理一遍不太现实，这时取样必不可少；
            2. 为了获得更好的查询处理效率：桶为表提供了额外的结构，Hive 在某些查询的时候可以利用这个结构，从而提高查询效率。
13. Hive 的元数据存储 Metastore

a) 在 Hive 中由一系列的数据表格组成一个命名空间，关于这个命名空间的描述信息会保存在 Metastore 的空间中
b) 元数据使用 SQL 的形式存储在传统的关系数据库中，Apache 实现是 Derby 数据库
c) 原因：元数据不宜存放在 Hadoop 的 HDFS 文件系统中，否则会降低元数据的访问效率，进一步导致降低 Hive 的整体性能。

**第四章 高级技术**

1. 引入复合键的原因：大量小的键值对合并为大的键值对来减少网络数据通信开销、提高程序计算效率，同时，复合键可以让系统帮助我们完成对 value 的排序处理（需要实现新的 Partitioner 保证原来同一 key 的键值对最后分区到同一个 Reduce 节点上）；
2. 内置的文件输入格式：
   a) TextInputFormat：key：行偏移量，value：行内容
   b) KeyValueTextInputFormat：key：\tab 之前的内容，\tab 之后剩余的部分
3. 内置的文件输出格式：
   a) TextOutputFormat：”key\tvalue”格式输出
4. 用户自定义数据类型
   a) 第一个基本要求是需要实现 Writable 接口，以便改数据能被序列化后完成网络传输或文件输入输出；
   b) 第二个要求是，如果该数据需要作为主键 Key 使用或者需要比较数值大小时，需要重载的 WritableComparable 接口：write、readFields 和 compareTo
   c) compareTo
      i. return this-object：升序
      ii. return object-this：降序
      iii. str1.compareTo(str2)：str1>str2 返回 1，str1=str2 返回 0，str1<str2 返回-1
5. 迭代式 MapReduce 程序设计：PageRank
6. 组合式 MapReduce 程序设计
   a) 具有数据依赖关系的 MapReduce 子任务的执行
      i. Job 类除了维护 Conf 信息外，还维护 Job 间的依赖关系；
      ii. JobControl 类用来控制整个作业的执行;
   b) 前处理和后处理步骤的链式执行
      i. 目的是减少使用多个 MapReduce 作业的处理周期，减少 I/O 操作，提升处理效率；
      ii. 提供链式 Mapper（ChainMapper）和链式 Reducer（ChainReducer）
7. 多数据源的连接
   a) DataJoin 类实现 Reduce 端连接：
      i. 首先为不同数据源下的每个数据记录定义数据源标签 Tag（例如 Customers 和 Orders），然后需要为每个待连接的数据记录确定一个连接主键（GroupKey）
      ii. 程序员需要完成如下部分：
         1. DataJoinMapperBase：由程序员实现的 Mapper 类继承。该基类已实现了 map()方法用以完成标签化数据记录的生成和输出，因此程序员仅需实现产生数据源标签、GroupKey 和标签化记录所需要的抽象方法（generateInputTag, generateTaggedMapOutput, generateGroupKey）；
         2. DataJoinReducerBase：由程序员实现的 Reducer 类继承。该基类已实现了 reduce()方法用以完成多数据源记录的叉积组合生成。程序员仅需实现 combine()方法以便对每个叉积组合中的数据记录进行合并连接处理；

3. TaggedMapOutput：描述一个标签化数据记录，实现了 getTag()，setTag()方法，程序员需要继承并实现 Writable 接口以进行 I/O（TaggedRecordWritable），并实现抽象的 getData()方法用以读取记录数据。

b) 用文件复制方法实现 Map 端 Join：
   i. 引入原因：前述用 DataJoin 类实现的 Reduce 端 Join 方法中，Join 操作直到 Reduce 阶段才能处理，很多无效的连接数据组合在 Reduce 阶段才能去除，而这时这些数据已经通过网络从 Map 阶段传送到了 Reduce 阶段，占据了很多的通信带宽。因此这个方法的效率不是很高；
   ii. 什么是 Distributed Cache？当一个数据源的数据量较小、能够存放在单个节点的内存中时，我们可以使用一个称为“Replicated Join”的方法，把较小的数据源文件复制到每个 Map 节点 然后在 Map 阶段完成 Join 操作；Hadoop 提供了一个 Distributed Cache 机制用于将一个或多个文件复制到所有节点上。
   iii. 编程设置：
      1. Job 类中：addCacheFile：将一个文件存放到 Distributed Cache 文件中；
      2. Mapper 或 Reducer 的 context 类中：getLocalCacheFiles：获取设置在 DistributedCache 中的文件路径，以便将这些文件读入到每个节点中。
   iv. PPT 示例中，不再需要 Reducer
   v. 即使较小的数据源文件，也可能仍然无法全部存放在内存中处理。但如果计算问题本身仅需要使用较小数据源中的部分记录，可以实现一个方法，先根据一定条件过滤数据记录，并保存为一个临时文件。当这个临时文件可以存放在内存中时，即可使用 Replicated Join 方法进行处理

c) 带 Map 端过滤的 Reduce 端连接：过滤后数据仍然无法放在内存中处理
   i. 在 Map 端先生成一个仅包含连接主键的过滤文件，并存放在 Distributed Cache 中，然后在 Map 端先过滤不在这个列表中的所有两张表记录，然后再实现正常的 Reduce 端连接

d) 限制：数据源有多个不同的主键/外键连接时，需要多次 MapReduce 过程完成不同的主键/外键连接

8. 全局参数的传递与使用：Configuration
   a) 目的：为了能让用户灵活设置某些作业参数，避免用硬编码方式在程序中设置作业参数，一个 MapReduce 计算任务可能需要在执行时从命令行输入这些作业参数，并将这些参数传递给各个计算节点
   b) 方法：在命令行指定，然后该输入参数可以作为一个属性保存在 Configuration 对象中，并允许 Map 和 Reduce 节点从 Configuration 对象中获取和使用该属性值。
9. 全局数据文件的传递：Distributed Cache
   a) 作用：较小的并且需要复制到各个节点的数据文件，具体使用时为提供访问速度可以将这些较小的文件数据读入内存。
   b) 使用：job.addCacheFile(xxx)，context.getLocalCacheFiles()
10. 查询任务相关信息：可以通过 Configuration 对象，使用预定义的属性名称查询计算作业相关的信息。
11. 划分多个输出文件集合（例如按日期划分）：MultipleOutputFormat，需要实现其中的 generateFileNameForKeyValue()方法以根据当前的键值对和当前数据文件名由程序产生并返回一个输出文件路径
12. 关系数据库输入输出：提供基于 MapReduce 大规模数据并行处理的 OLAP（联机分析

处理）

a) 输入
   i. DBInputFormat（提供从数据库读取数据的格式）和 DBRecordReader（提供读取数据记录的接口）的处理效率不理想，不适合 OLAP 数据仓库大量数据的读取处理；
   ii. 读取大量数据记录一个更好的解决办法是，用数据库中的 Dump 工具将大量待分析数据输出为文本数据文件，并上载到 HDFS 中进行处理。
b) 输出
   i. 处理结果数据量一般不会太大，可以直接向数据库写入。Hadoop 接口有 DBOutputFormat（提供向数据库输出数据的格式），DBRecordWriter（提供向数据库写入数据记录的接口），DBConfiguration（提供数据库配置和创建连接的接口）

**第五章 编程**

1. PageRank
   a) 简化的 PageRank 面临的两个问题：
      i. 排名泄露：如果有网页没有出度链接，则经过多次迭代后所有网页的 PR 值趋向 0；
      ii. 排名下沉：如果有网页没有入度链接，则经过多次迭代后它的 PR 值会趋近 0；
   b) GraphBuilder、PageRankIter（两种 key, value 对）、PageRankViewer
2. 倒排索引
3. 二次排序
4. 数据库表 Join
5. K-Means
   a) 唯一的全局文件：ClusterID、Cluster Center 和属于该 Cluster Center 的数据点的个数
   b) <key, value>=<ClusterID, (pm, n)>，pm 是数据点或数据点的中心点
   c) 为什么输出 value 必须是(p,1)而非 p？因为有 Combiner，需要计数，并且因为 Reducer 的输入格式唯一，所以 Combiner 和 Mapper 的发射类型要一样。
6. KNN：训练样本数据放在 Distributed Cache 中供每个节点共享访问，测试样本分块后放在不同的结点上计算相似度，取相似度最大的 K 个节点的标签加权得到类别。
7. 朴素贝叶斯：用 MapReduce 扫描训练数据集，统计分类 Yi 和每个属性值 xj 在 Yi 中出现的频度；在分类预测时，根据从训练数据集计算出的 FxYij 进行求积得到 FXYi(即 P(X|Yi))，再乘以 FYi 即可得到 X 在各个 Yi 中出现的频度 P(X|Yi)P(Yi)，取得最大频度的 Yi 即为 X 所属的分类。
8. SVM、PSON 感觉了解一下大题思路即可。
9. 撰写算法思想的思路：
   a) Mapper？
   b) Reducer？
   c) Combiner？Partitioner？（如果有）
   d) 全局共享的数据有哪些？自定义的 Comparable 数据结构有什么？
   e) 算法主要分什么阶段（如果有迭代特性的话）

**第六章 Spark**

1. 为什么有 Spark？（关键：重复计算、资源共享、系统组合、内存计算）
   a) MapReduce 计算模式的缺陷，包括 I/O 性能低下，延时高、数据共享麻烦、没有

很好地利用内存、对复杂问题的表达能力差；

b) MapReduce 对非批处理大数据问题的局限性，而以内存计算为核心，集诸多计算模式之大成的 Spark 更好；
c) 速度：在内存计算上比 MapReduce 至多快 100 倍，磁盘上至多快 10 倍；
d) 易用性：可以用 Java、Scala、Python 等简洁快速编程；
e) 广泛性：结合 SQL、流处理和复杂分析；
f) 多处运行：可以在 Hadoop、云等地方运行；
g) 数据共享方便：利用内存加速计算

2. Spark 基本定义：一种基于内存的迭代式分布式计算框架，适合完成多种计算模式的大数据处理任务
3. RDD：弹性分布式数据集
   a) 定义：能横跨集群所有节点进行并行计算的分区元素集合；
   b) 创建方式：Hadoop 文件系统的一个文件创建而来，或者从一个已有的 Scala 集合转换得到；
   c) 只读，可分区，全部或部分可以缓存在内存中，在多次计算间重用
4. Spark 使用 RDD 以及对应的 Transform（惰性操作）/Action（立即计算）等操作算子执行分布式计算；
   a) 容错性（Lineage、CheckPoint）：Lineage 基于 RDD 之间的依赖关系组成 lineage，以及重计算和 checkpoint 等机制；
   b) 只读，可分区：数据集的全部或部分可以缓存在内存中，在多次计算中重用；
   c) 弹性：内存不够时可以与磁盘进行交换；
   d) 一组 RDD 形成可执行的有向无环图 DAG；
5. Spark 集群的基本结构（集群角度）
   a) Master Node：集群部署时的概念，是整个集群的控制器，负责整个集群的正常运行，管理 Worker Node；
   b) Worker Node：是计算节点，接收主节点命令与进行状态汇报；
      i. Worker Node 中有 Worker 和 Executor。Worker 负责管理本地资源，接收主节点命令，Executor 负责执行计算；
      ii. Master 和 Worker 都不负责计算，但都有负责管理资源的功能（Master 会管理集群资源，Worker 管理本地资源）；
   c) Executors：每个 Worker 上有一个 Executor，负责完成 Task 程序的执行。
6. Spark 系统的基本结构（应用程序角度）
   a) Driver：应用执行起点，负责作业调度；
   b) Worker：管理计算节点及创建并行处理任务；
   c) Cache：存储中间结果；
7. Spark 应用程序的基本结构
   a) Application：基于 Spark 的用户程序，包含了一个 Driver Program 和多个 Executor
   b) Job：某个 RDD 的 Action 算子生成或者提交的一个或者多个一系列的调度阶段，称为一个或者多个 Job，可包含多个 Task；
   c) Stage：每个 Job 被拆分成一系列任务的集合，每组任务被称为一个 Stage；
   d) Task：基本程序执行单元，在一个 Executor 上执行
   e) 总结：Application 包含 Driver Program 和多个 Executors；一个 Job 里有多组 Stage，每个 Stage 里面有多个 Task，每个 Task 在一个具体的 Executor 上执行
8. Spark Driver 的组成：

a) RDD：Spark 的基本计算单元，一组 RDD 形成可执行的 DAG，操作主要有 Transformation 和 Action

9. Spark 运行框架主节点
   a) Application：基于 Spark 的用户程序，包含了一个 Driver Program 和多个 Executor
   b) Driver Program：执行用户代码的 main()函数，并创建 SparkContext（一定要有，否则只是一个 Scala 程序）；
   c) Cluster Manager：集群当中的资源调度服务选取；
   d) **Job：**某个 RDD 的 Action 算子生成或者提交的一个或者多个一系列的调度阶段，称为一个或者多个 Job，它由一组 Transformation 操作和一个 Action 操作组成；
   e) **Spark Context：**SparkContext 由用户程序启动，是 Spark 运行的核心模块，它对一个 Spark 程序进行了必要的初始化过程
      i. 创建 SparkConf 类的实例：包含用户自定义参数信息和 Spark 配置文件中的一些信息；
      ii. 创建 SparkEnv 类的实例：包含了 Spark 执行时所需要的许多环境对象；
      iii. 创建调度类的实例：创建 TaskScheduler 和 DAGScheduler 两种 Spark 的调度；
10. Spark 运行框架的从节点：Executor、Stage、Task
    a) Stage 分两种，Shuffle Stage（宽依赖时触发，父 RDD 的一个 Partition 被子 RDD 多个 Parition 依赖）和 Final Stage（Action 操作触发）
    b) Task 作用单位是 Partition，针对同一个 Stage，分发到不同的 Partition 上执行。
11. Spark 程序执行过程：
    a) 用户编写的 Spark 程序提交到相应的 Spark 运行框架中；
    b) Spark 创建 SparkContext 作为本次程序的运行环境；
    c) SparkContext 连接相应的集群配置(Mesos/YARN),来确定程序的资源配置使用情况。；
    d) 连接集群资源成功后，Spark 获取当前集群上存在 Executor 的节点，即当前集群中 Spark 部署的子节点中处于活动并且可用状态的节点
    e) Spark 分发程序代码到各个节点。；
    f) 最终，SparkContext 发送 Tasks 到各个运行节点来执行。
12. RDD 在转换过程中，内部的 Partition 实际是存储管理在底层的 BlockManager 中的，在需要的时候由 BlockManger 提供对 RDD 数据的内存缓存；
13. Stage 切分方式：1. 最后一个是一个独立的 stage；2. 凡是遇到 Shuffle 的地方，就要一个新的 Stage，因为它们不可以并行算，必须等前面全部做完才能算
14. 构建 RDD 世系关系的优势：基于 RDD 世系的并行执行优化；更快速，更细粒度的容错
    a) 容错粒度是以每个 RDD 的 Partition 为基本单位的，只需重算所需 Partition 即可
15. Spark 的技术特点：
    a) RDD：弹性分布式数据集，指能横跨集群所有节点进行并行计算的分区元素集合，是 Spark 核心的分布式数据抽象；
    b) Transformation 和 Action：Spark 通过 RDD 的两种不同类型的运算实现了惰性计算，即在 RDD 的 Transformation 运算时，Spark 并没有进行作业的提交；而在 RDD 的 Action 操作时才会触发 SparkContext 提交作业
       i. 好处：更容易做全局优化，并且 RDD 的容错相比于 HDFS 的多副本存储设计而言粒度更细。
    c) Lineage：为了保证 RDD 中数据的鲁棒性，Spark 系统通过世系关系(lineage)来记录一个 RDD 是如何通过其他一个或者多个父类 RDD 转变过来的，当这个 RDD 的

数据丢失时，Spark 可以通过它父类的 RDD 重新计算；

d) Spark 调度:：Spark 采用了事件驱动的 Scala 库类 Akka 来完成任务的启动，通过复用线程池的方式来取代 MapReduce 进程或者线程启动和切换的开销。

e) API：Scala 更简洁，也支持 Java、Python 等语言；

f) Spark 生态：Spark SQL、Spark Streaming、GraphX 等等为 Spark 的应用提供了丰富的场景和模型，适合应用于不同的计算模式和计算任务；

g) Spark 部署：Spark 拥有 Standalone、Mesos、YARN 等多种部署方式，可以部署在多种底层平台上

16. 持久化 RDD 的三个存储策略：
    a) 未序列化的 Java 对象，存于内存中；
    b) 序列化的数据，存于内存中；
    c) 磁盘存储；
17. RDD 包含：一组 RDD 的 Partition、对父 RDD 的一组依赖（Lineage）、父 RDD 上执行何种计算的一个函数和描述分区模式和数据存放位置的元数据四类信息
18. Spark 的 map、flatMap 有什么区别？
    a) map 返回一个输出项
    b) flatMap 返回一个序列，序列中可以有 0 个或更多个输出项
    c) 总的来说，flatMap 会先执行 map 的操作，再将所有对象扁平化为一个序列对象
19. Spark 的 mapPartitions 和 mapParitionsWithIndex 区别？
    a) mapParitions：类似 map，但是只在 RDD 的每个 Parition 上独立运行，即 func 在 RDD 的 type 是 T 的情况下，形参必须是 Iterator<T>
    b) mapPartitionsWithIndex：类似 mapPartitions，但是 func 除了 Iterator<T>作为形参以外还需要 Partition 的索引（整型）作为形参
20. WordCount 的 Spark 代码
    a) val file = spark.textFile("hdfs://…")
    b) val counts = file.flatMap(" ").map((word=>(word,1)).reduce(_+_)
    c) counts.saveAsTextFile("hdfs://…")