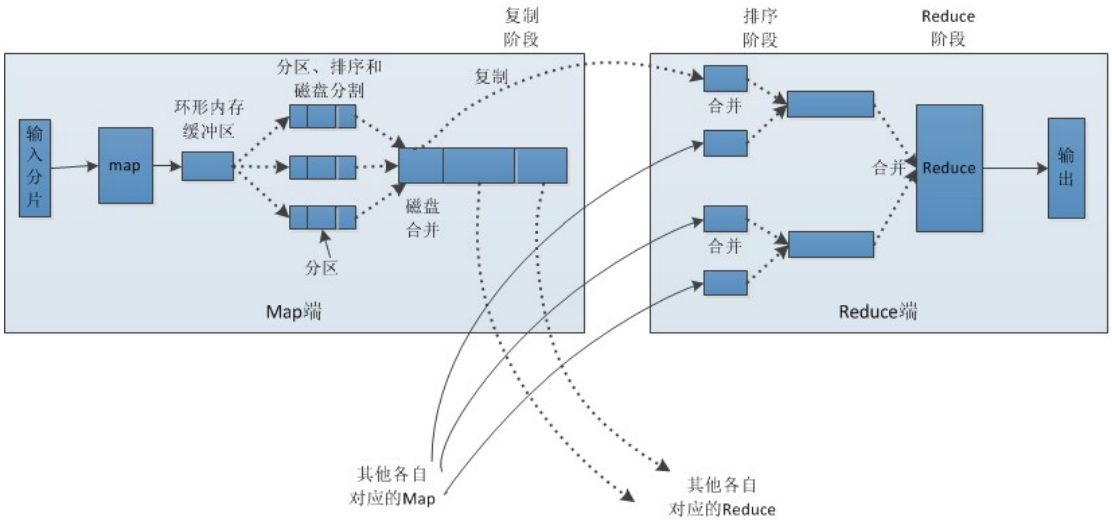
MR



Map 端：

1. 每个输入分片会让一个 map 任务来处理，默认情况下，以 HDFS 的一个块的大小（默认为 128M）为一个分片。map 输出的结果会暂且放在一个环形内存缓冲区中（该缓冲区的大小默认为 100M，由 io.sort.mb 属性控制），当该缓冲区快要溢出时（默认为缓冲区大小的 80%，由 io.sort.spill.percent 属性控制），会在本地文件系统中创建一个溢出文件，将该缓冲区中的数据写入这个文件。
2. 在写入磁盘之前，线程首先根据 reduce 任务的数目将数据划分为相同数目的分区，也就是一个 reduce 任务对应一个分区的数据。这样做是为了避免有些 reduce 任务分配到大量数据，而有些 reduce 任务却分到很少数据，甚至没有分到数据的尴尬局面。其实分区就是对数据进行 hash 的过程。然后对每个分区中的数据进行排序。如果此时设置了 combiner，将排序后的结果进行 combine 操作，combiner 也是一个 reduce，这样做的目的是让尽可能少的数据写入到磁盘。
3. 当 map 任务输出最后一个记录时，可能会有很多的溢出文件，这时需要将这些文件合并。合并的过程中会不断地进行排序和 combine 操作，目的有

两个：1.尽量减少每次写入磁盘的数据量；2.尽量减少下一复制阶段网络传输的数据量。最后合并成了一个已分区且已排序的文件。为了减少网络传输的数据量，这里可以将数据压缩，只要将 mapred.compress.map.out 设置为 true 就可以了。

4. 将分区中的数据拷贝给相对应的 reduce 任务。
   注：map 任务一直和其父 TaskTracker 保持联系，而 TaskTracker 又一直和 JobTracker 保持心跳。所以 JobTracker 中保存了整个集群中的宏观信息。reduce 任务向 JobTracker 获取对应的 map 输出位置。
5. shuffle：一个 map 产生的数据，结果通过 hash 过程分区却分配给了不同的 reduce 任务。

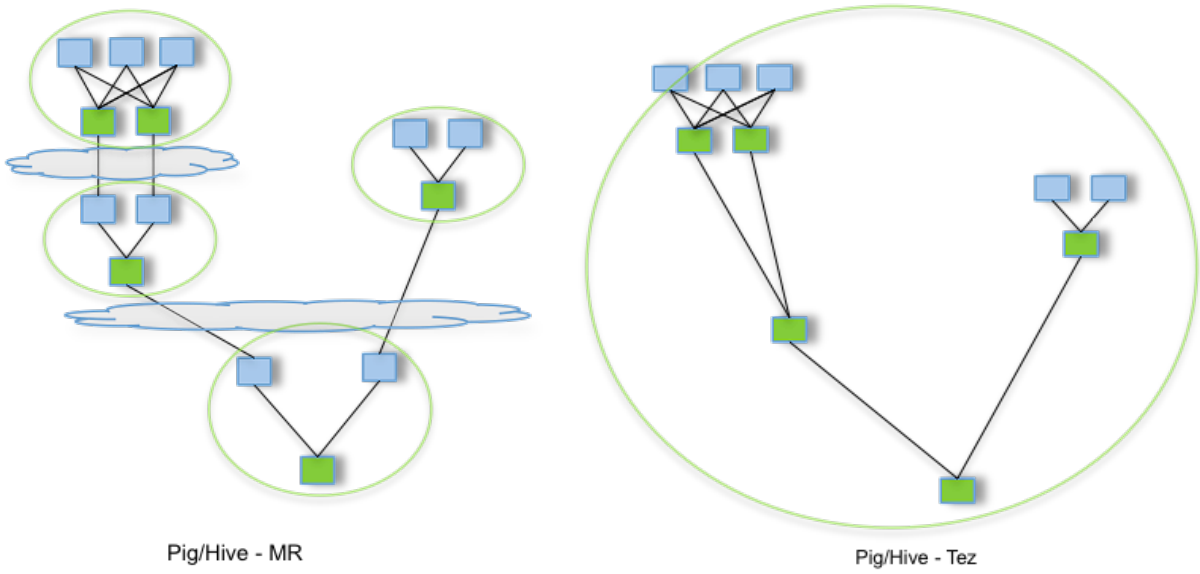
Reduce 端：

1. reduce 会接收到不同 map 任务传来的数据，并且每个 map 传来的数据都是有序的。如果 reduce 端接受的数据量相当小，则直接存储在内存中（缓冲区大小由 mapred.job.shuffle.input.buffer.percent 属性控制，表示用作此用途的堆空间的百分比），如果数据量超过了该缓冲区大小的一定比例（由 mapred.job.shuffle.merge.percent 决定），则对数据合并后溢写到磁盘中。
2. 随着溢写文件的增多，后台线程会将它们合并成一个更大的有序的文件，这样做是为了给后面的合并节省时间。其实不管在 map 端还是 reduce 端，MapReduce 都是反复地执行排序，合并操作。
3. 合并的过程中会产生许多的中间文件，但 MapReduce 会让写入磁盘的数据尽可能地少，并且最后一次合并的结果并没有写入磁盘，而是直接输入到 reduce 函数。

Tez

Apache Tez 将数据处理模型化为数据流图，图中的顶点代表数据的处理，边代表处理之间的数据移动。 因此，分析和修改数据的用户逻辑位于顶点。 边确定数据的使用者，数据如何传输以及生产者和消费者顶点之间的依赖关系。这

个模型简洁地捕捉了计算的逻辑定义。 当 Tez 作业在群集上执行时，它通过在顶点添加并行性来将此逻辑图展开为物理图形，以便缩放到正在处理的数据大小。 每个逻辑顶点创建多个任务并行执行计算。



第一个图表展示的流程包含多个 MR 任务，每个任务都将中间结果存储到 HDFS 上——前一个步骤中的 reducer 为下一个步骤中的 mapper 提供数据。第二个图表展示了使用 Tez 时的流程，仅在一个任务中就能完成同样的处理过程，任务之间不需要访问 HDFS。

更具体地说，数据处理以有向无环图（DAG）的形式表示。 处理从 DAG 的根顶点开始，并继续沿着有向边，直到达到叶顶点。 当 DAG 中的所有顶点都完成后，数据处理工作就完成了。 该图没有周期，因为 Tez 使用的容错机制是重新执行失败的任务。当任务的输入丢失时，输入的生成器任务将被重新执行，因此 Tez 需要能够遍历图边以定位重新开始计算的非失败任务。 图表中的循环可能会使此步行难以执行。 在某些情况下，可以通过展开循环来创建 DAG 来处理循环。

Tez 定义了一个简单的 Java API 来表达数据处理的 DAG。 该 API 有三个组件：

- DAG： 这定义了整个工作。 用户为每个数据处理作业创建一个 DAG 对象。
- 顶点： 这定义了用户逻辑和执行用户逻辑所需的资源和环境。 用户为作业中的每个步骤创建一个顶点对象，并将其添加到 DAG。
- 边： 这定义了生产者和消费者顶点之间的连接。 用户创建一个 Edge 对象并使用它来连接生产者和消费者顶点。

边缘属性

以下边缘属性使 Tez 能够实例化任务，配置其输入和输出，适当地调度它们并帮助在任务之间路由数据。 每个顶点的并行性是根据用户指导，数据大小和资源来确定的。

- 数据移动： 定义任务之间的数据路由
  - 一对一：来自第 i 个生产者任务的数据路由到第 i 个消费者任务。
  - 广播：从生产者任务路由到所有消费者任务的数据。
  - 分散聚集（Scatter-Gather）：生产者任务将数据分散到碎片中，消费者任务收集碎片。所有生产者任务的第 i 个碎片路由到第 i 个消费者任务。
- 调度 ： 定义消费者任务的安排时间
  - 顺序：消费者任务可以在生产者任务完成后安排。
  - 并发：消费者任务必须与生产者任务共同安排。
- 数据源： 定义任务输出的寿命/可靠性
  - 坚持：输出将在任务退出后可用。 输出可能会在稍后丢失。
  - 持久可靠：输出可靠地存储并始终可用
  - Ephemeral：输出仅在生产者任务正在运行时可用

Tez 调度程序

在决定如何分配任务的时候，Tez 调度程序考虑了很多方面，包括：任务位置需求、容器的兼容性、集群可利用资源的总量、等待任务请求的优先级、自动并行化、释放应用程序不再使用的资源（因为对它而言数据并不是本地的）等。它还维护着一个使用共享注册对象的预热 JVM 连接池。应用程序可以选择使用这些共享注册对象存储不同类型的预计算信息，这样之后再进行处理的时候就能重用它们而不需要重新计算了，同时这些共享的连接集合及容器池资源也能非常快地运行任务。

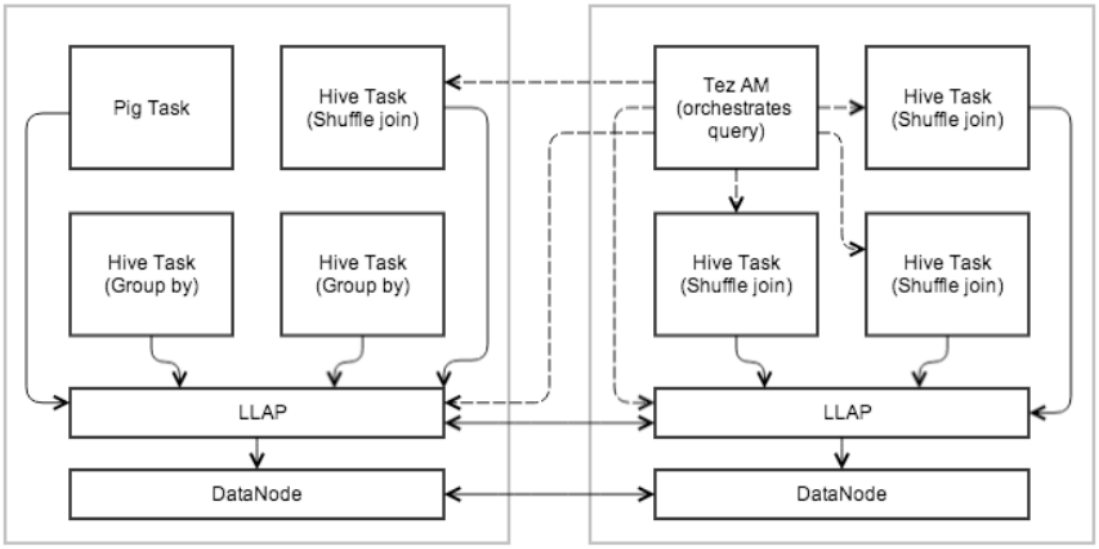
LLAP

LLAP 提供了一个高级的执行模式，它包括一个长久存活的守护程序去代替了和 HDFS datanode 的直接交互和一个紧密集成的 DAG 框架。这个守护程序中加入了缓存、预抓取、查询过程和访问控制等功能。短小的查询由守护程序执行，大的重的操作由 yarn 的 container 执行。

和 datanode 相似，llap 守护程序可以被其他程序使用，特别是一个以文件

为中的展示数据关系的视图。这个是守护程序也开发了 API 让其它程序来集成它。

下面这个图展示的了 LLAP 在 Tez 的执行。初始化阶段的查询放在了 LLAP，大的 shuffle 在他们的 container 中执行，多个查询和应用可以同时访问 LLAP.



持久的守护程序

为了实现缓存、JIT 优化和减少启动时间，我们要在集群的节点上运行守护进程，它将处理 IO、缓存、查询段执行。

- **这些节点是无状态的。**对 LLAP 节点的任何请求都包含数据位置和元数据。 它处理本地和远程位置; 地方是来电者的责任（YARN）。
- **恢复/弹性。** 故障和恢复被简化，因为任何数据节点仍然可以用来处理输入数据的任何片段。Tez AM 因此可以简单地在集群上重新运行失败的碎片。
- **节点之间的通信。** LLAP 节点能够共享数据（例如，获取分区，广播片段）。 这是用 Tez 中使用的相同机制来实现的。

执行引擎

LLAP 在现有的基于进程的 Hive 执行中工作，以保持 Hive 的可扩展性和多功能性。 它不会取代现有的执行模式，而是增强它。

- **守护进程是可选的。** Hive 可以在没有它们的情况下工作，即使部署和运行也可以绕过它们。 保持语言特征的功能均等性。
- **外部编排和执行引擎。** LLAP 不是一个执行引擎（如 MapReduce 或 Tez）。整个执行过程由现有的 Hive 执行引擎（如 Tez）在 LLAP 节点上以及常规容器上进行调度和监视。 显然，LLAP 的支持水平取决于每个单独的执行引擎（从 Tez 开始）。MapReduce 的支持没有计划，但其他

引擎可能会在稍后添加。 像 Pig 这样的其他框架也可以选择使用 LLAP 守护进程。

- **部分执行。** 由 LLAP 守护程序执行的工作的结果可以形成 Hive 查询结果的一部分，也可以传递给外部 Hive 任务，具体取决于查询。
- **资源管理。**YARN 仍然负责管理和分配资源。YARN 容器委托模型用于允许将分配的资源转移给 LLAP。为了避免 JVM 内存设置的限制，缓存的数据保存在堆外，以及处理的大缓冲区（例如，group by，joins）。这样，守护进程可以使用少量内存，并且会根据工作负载分配额外的资源（即 CPU 和内存）。

查询片段执行

- **并行执行。** LLAP 节点允许并行执行来自不同查询和会话的多个查询片段。
- **接口。**用户可以通过客户端 API 直接访问 LLAP 节点。他们能够指定关系转换并通过面向记录的流读取数据。

I / O

守护进程卸载 I / O 并将压缩格式转换为单独的线程。 数据在准备就绪时被传递到执行中，所以以前的批处理可以在下一个准备好的时候被处理。 数据以一个简单的 RLE 编码的列格式传递给执行，准备好进行矢量化处理; 这也是缓存格式，旨在尽量减少 I / O，缓存和执行之间的复制。

- **多种文件格式。** I / O 和缓存取决于底层文件格式的一些知识（特别是如果要有效地完成）。 因此，与矢量化工作类似，不同的文件格式将通过特定于每种格式的插件来支持（从 ORC 开始）。另外，可以添加一个支持任何 Hive 输入格式的通用效率较低的插件。 插件必须维护元数据并将原始数据转换为列块。
- **预测和开花过滤器。** 如果支持 SARG 和布隆过滤器，则会将其压入存储层。

高速缓存

守护进程缓存输入文件的元数据以及数据。 即使对于当前未缓存的数据，也可以缓存元数据和索引信息。元数据存储在 Java 对象中， 高速缓存的数据以 I / O 部分中描述的格式存储，并保存在堆外（请参阅资源管理 ）。

- **驱逐策略。**驱逐策略针对具有频繁（部分）表扫描的分析工作负载进行调整。 最初，使用像 LRFU 这样的简单策略。 该政策是可插入的。
- **缓存粒度。** 列块是缓存中数据的单位。 这实现了低开销处理和存储效

率之间的妥协。 块的粒度取决于特定的文件格式和执行引擎（Vectorized Row Batch Size，ORC stripe 等）。

自动创建布隆过滤器以提供动态运行时过滤。