Spark/HDFS data locality optimization

Overview

This document researches how exactly HDFS data locality is broken in Kubernetes. And also discusses how to fix it. See “How to fix” for the ideas.

Links:

- See this umbrella issue discussing HDFS data locality for other related points.
- A prototype in Kubernetes-HDFS repo that puts HDFS namenode/datanode daemons in Kubernetes (Kubernetes-HDFS repo issue)

HDFS locality, broken in Kubernetes

HDFS data locality relies on matching executor host names against datanode host names:

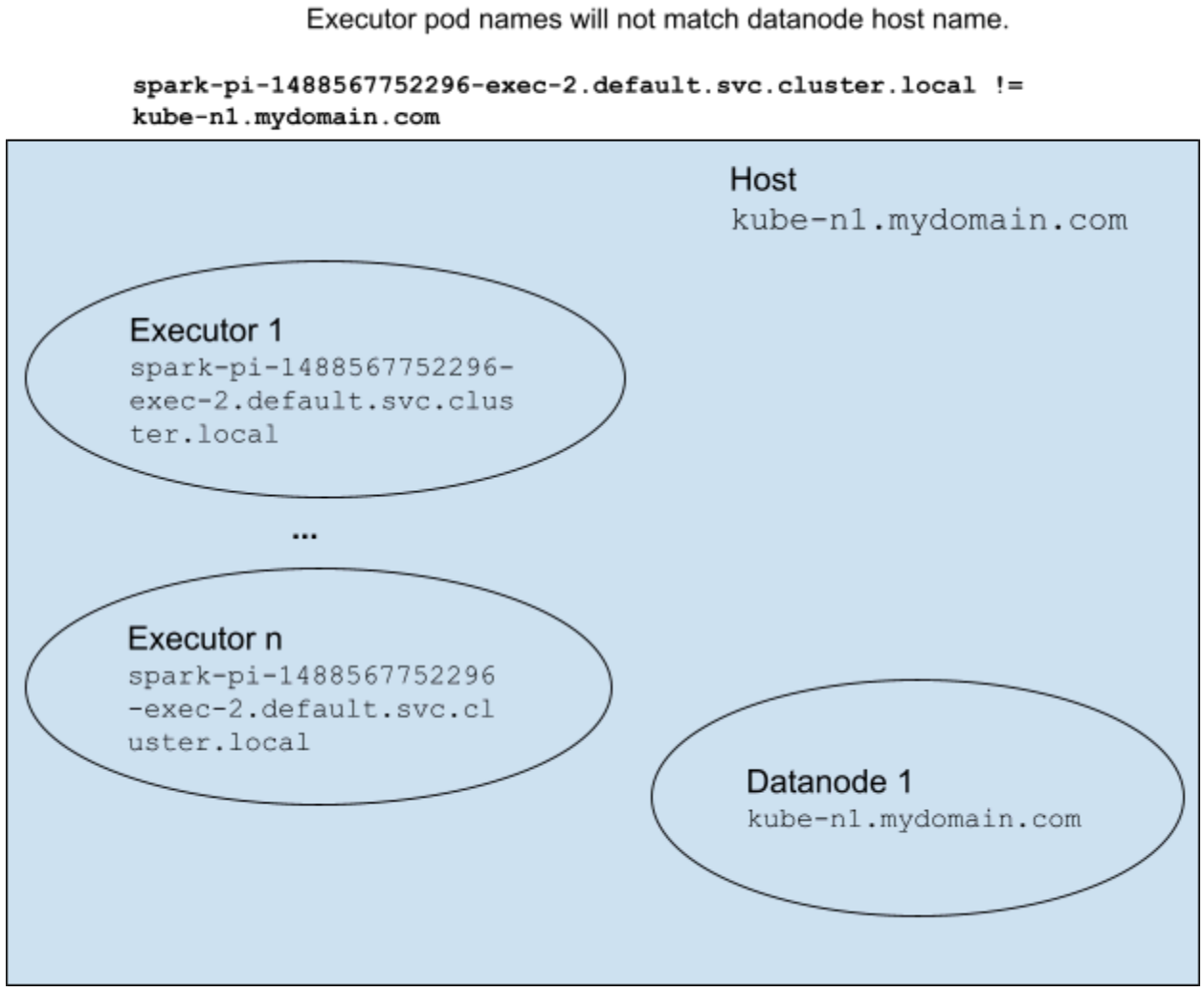
A. Node locality: If the host name of an executor matches that of a given datanode, it means the executor can read the data from the local disks of the datanode.
B. Rack locality:
    a. The namenode has topology information which is a list of host names with rack names as satellite values. If the namenode can retrieve an entry in its topology information using the executor host name as the key, then we can determine the rack that the executor resides. This means the rack locality will work. I.e. The

executor can read data from datanodes in the same rack. The namenode is provided with the topology information by a topology plugin.

b. Spark driver also accesses the same topology information using the same topology plugin mechanism. Note the driver loads own instance of plugin into its JVM, as opposed to sending RPC requests to the namenode.

The node locality (A) is broken In Kubernetes. Each executor pod is assigned a virtual IP and thus virtual host name. This will not match datanodes' physical host names.

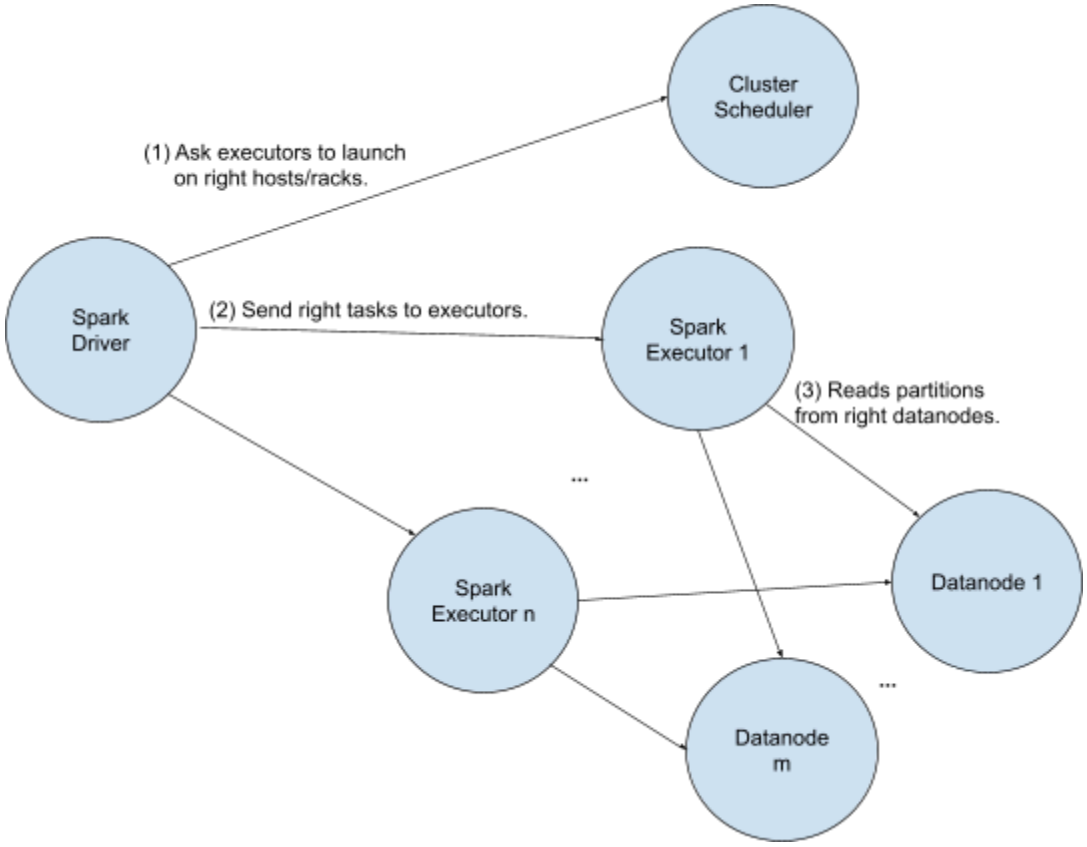
Similarly, rack locality (B) is broken. The executor host name may fail to match any topology entry in the namenode if the namenode or driver see only the virtual pod IP addresses for executors.



Locality-aware layers

Here, we look at existing locality-aware layers.

When Spark reads data from HDFS, it can increase the read throughput by sending tasks to the executors that can access the needed disk data on the same node or another node on the same rack. This locality-aware execution is implemented in three different layers:



1. Executor scheduling: When Spark Driver launches executors, it may suggest the cluster scheduler to consider a list of candidate hosts and racks that it prefers. The Driver gets this list by asking namenode which datanode hosts have the input data of the application. (In YARN, the optimization in this layer is triggered only when Spark dynamic allocation is enabled). See Appendix A for the detailed code snippets.
2. Task dispatching: Once it gets executors, Spark Driver will dispatch tasks to executors. When an executor is ready for more tasks, the Driver tries to send tasks that have the input data on the very executor host or on other hosts in the same rack. For this, the Driver builds the hosts-to-tasks and racks-to-tasks mapping with the datanode info from

the namenode, and later looks up the maps using executor host names or rack names. See Appendix B for details.

3. Reading HDFS partitions: When a task actually runs on an executor, it will read its partition block from a datanode host. The HDFS read library asks the namenode to return multiple candidate datanode hosts each of which has a copy of the data. The namenode sorts the result in the order of the proximity to the client host. The client picks the first one in the returned list. See Appendix C for details.

Layer (1) is not necessarily broken for k8s because we can probably use the k8s node selection, in particular the new affinity can express the preference as soft requirement. However the correct implementation only exists in YARN-related code. We'll need to generalize and reuse the code in the right way for k8s.

Layer (2) is broken. Spark Driver will build the mapping using the data node names. Then it will look up the maps using the executor pod host names, which will never match.

Layer (3) may be broken. So far we saw this only in Google Container Engine GKE, which uses the native kubenet network provider (See http://tiny.pepperdata.com/hdfs-k8s-gke for details) When this layer is broken, the namenode will retrieve the executor pod IP for the client. And compare the pod IP against datanode host IPs to sort the datanode list. The resulting list will not be in the correct order.

However, we observed that many network providers rewrite packets to conduct NAT so the namenode sees the physical IP of the node that pods run on.

- By design, overlay networks such as weave and flannel conduct NAT for any pod packet heading outside a local pod network. This means packets coming to a node IP also does NAT. (In overlay, pod packets heading to another pod in a different node puts back the pod IPs once they got inside the destination node)
- In EC2, the standard tool kops can provision k8s clusters using the same native kubenet that GKE uses. Unlike GKE, it turns out kubenet in EC2 does NAT between pod subnet to host network. This is because kops sets option **--non-masquerade-cidr=100.64.0.0/10** to cover only pod IP subnet. Traffic to IPs ouside this range will do NAT. In EC2, cluster hosts like 172.20.47.241 sits outside this CIDR. This means pod packets heading to node IPs will do masquerading. (Note GKE kubenet uses the default value of --non-masquerade-cidr, 10.0.0.0/8, which covers both pod IP and node IP subnets. GKE does not expose any way to override this value)
- Calico is a popular non-overlay network plugin. It turns out Calico can be also configured to do NAT between pod subnet and node subnet thanks to the nat-outgoing option. The option is enabled by default. We expect that if the nat-outgoing is turned off by config, this layer (3) issue will manifest.

How to fix

Based on weekly SIG discussion on Mar 29 2017:

Layer (1) can be probably fixed using the k8s node selection to express the preference. The new node affinity mechanism can express this as soft requirement. This is being addressed by PR #316.

Layer (2) can be an easy fix, if we translate executor pod IPs to physical cluster node names and use the physical names for looking up the hosts-to-tasks map. (Implemented by PR #216) We can use similar approach for racks-to-tasks map by getting the rack name of the physical cluster node name. (Implemented by PR #350)

Fixing layer (3), if broken like the GKE kubenet, is hard. We can think about three approaches:

I. Extend the namenode RPC protocol to explicitly specify, as the proximity ordering key, the physical cluster name that executor runs on. This requires changing both HDFS namenode and client library code in upstream Hadoop.
II. Change HDFS client library code to re-sort the datanode list on the client side. Inform the client library code of the physical cluster node name that the executor is running on, using an environment variable or JVM property. Most of the work will be subset of with (I). But this still requires changing upstream Hadoop code.
III. Write a topology plugin for the namenode and let the topology plugin aware of the executor pods. Somehow make the namenode retrieve the right topology entry for executor pods and do the ordering in the right way. Many knowns. And the topology update might be too frequent. But this can be the least intrusive way. See this HDFS doc for the topology plugin details.

We are leaning toward the topology plugin approach. kubernetes-HDFS #PR 11 implements the namenode plugin for the GKE kubenet plugin.

Open questions

1. How about HDFS write? Does it concern locality?

Appendix A: Executor launch code

In YARN, Spark Driver starts by launching the Application Master (AM). Afterward, it gets help from the Application Master further for launching executors. As such, the locality-aware code is implemented in a few helper classes that the AM is interacting closely with.

Static Allocation

Once launched, the AM registers itself with the Driver. While doing so, it creates a helper object YarnAllocator and tell it to ask YARN scheduler to launch executors.

```
private def registerAM(
     _sparkConf: SparkConf,
     _rpcEnv: RpcEnv,
     driverRef: RpcEndpointRef,
     uiAddress: Option[String],
     securityMgr: SecurityManager) = {

   ...
   allocator = client.register(driverUrl,
     driverRef,
     yarnConf,
     _sparkConf,
     uiAddress,
     historyAddress,
     securityMgr,
     localResources)

   allocator.allocateResources()
   ...
 }
```

When created, YarnAllocator gets its initial target number of executors from the spark config:

```
   @volatile private var targetNumExecutors =
   YarnSparkHadoopUtil.getInitialTargetExecutorNumber(sparkConf)
```

Then, it allocates executors. Note inside updateResourceRequests below, it consults with locality-aware data structures. They are empty for the static allocation, but will be populated properly from stage information when the dynamic allocation is enabled.

```
def allocateResources(): Unit = synchronized {
   updateResourceRequests()
   ...
   val allocateResponse = amClient.allocate(progressIndicator)
   ...
 }

 def updateResourceRequests(): Unit = {
   val pendingAllocate = getPendingAllocate
   val numPendingAllocate = pendingAllocate.size
```

```
    val missing = targetNumExecutors - numPendingAllocate - numExecutorsRunning

    if (missing > 0) {
      ...
      // Split the pending container request into three groups: locality matched list,
locality
      // unmatched list and non-locality list. Take the locality matched container request
into
      // consideration of container placement, treat as allocated containers.
      // For locality unmatched and locality free container requests, cancel these container
      // requests, since required locality preference has been changed, recalculating using
      // container placement strategy.
      val (localRequests, staleRequests, anyHostRequests) =
splitPendingAllocationsByLocality(
        hostToLocalTaskCounts, pendingAllocate)
      ...
      val containerLocalityPreferences =
containerPlacementStrategy.localityOfRequestedContainers(
        potentialContainers, numLocalityAwareTasks, hostToLocalTaskCounts,
          allocatedHostToContainersMap, localRequests)
    ...
  }
```

Dynamic Allocation

When the dynamic allocation is enabled, another helper object ExecutorAllocationManager is activated. It hooks a listener on to the onStageSubmitted event. In case a given stage uses HDFS input data, the listener extracts the locality information and updates internal maps inside ExecutorAllocationManager.

```
   override def onStageSubmitted(stageSubmitted: SparkListenerStageSubmitted): Unit = {
       ...
        stageSubmitted.stageInfo.taskLocalityPreferences.foreach { locality =>
          if (!locality.isEmpty) {
            numTasksPending += 1
            locality.foreach { location =>
              val count = hostToLocalTaskCountPerStage.getOrElse(location.host, 0) + 1
              hostToLocalTaskCountPerStage(location.host) = count
            }
          }
        }
        stageIdToExecutorPlacementHints.put(stageId,
          (numTasksPending, hostToLocalTaskCountPerStage.toMap))

        // Update the executor placement hints
        updateExecutorPlacementHints()
      }
    }
```

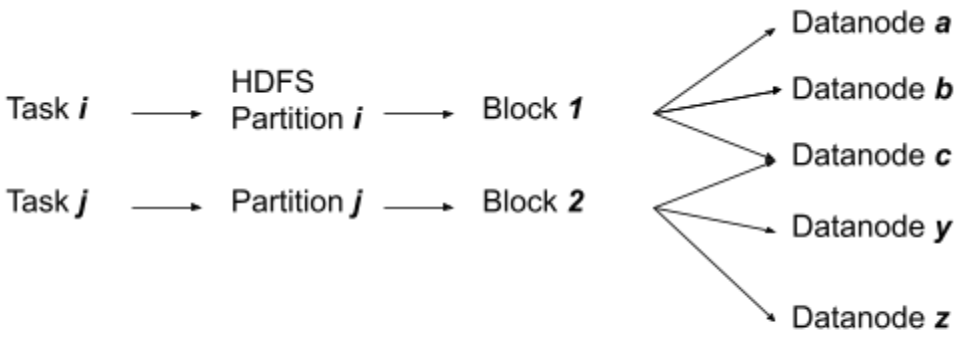
```
  def updateExecutorPlacementHints(): Unit = {
      var localityAwareTasks = 0
      val localityToCount = new mutable.HashMap[String, Int]()
      stageIdToExecutorPlacementHints.values.foreach { case (numTasksPending, localities) =>
        localityAwareTasks += numTasksPending
        localities.foreach { case (hostname, count) =>
          val updatedCount = localityToCount.getOrElse(hostname, 0) + count
          localityToCount(hostname) = updatedCount
        }
      }

      allocationManager.localityAwareTasks = localityAwareTasks
      allocationManager.hostToLocalTaskCount = localityToCount.toMap
    }
  }
```

These internal maps are referred to by the AM when it gets a request to allocate more executors from the Spark Driver. The AM simply tells YarnAllocator to get more using the locality data structure.

```
    override def receiveAndReply(context: RpcCallContext): PartialFunction[Any, Unit] = {
     case r: RequestExecutors =>
       Option(allocator) match {
         case Some(a) =>
           if (a.requestTotalExecutorsWithPreferredLocalities(r.requestedTotal,
             r.localityAwareTasks, r.hostToLocalTaskCount, r.nodeBlacklist)) {
             resetAllocatorInterval()
           }
           context.reply(true)
```

Appendix B: Task dispatch code

The spark driver maintains mapping from executors/hosts/racks to a list of tasks that they prefer running locally. For HDFS, it updates these maps with the data locality information. It gets datanode locations per HDFS partition by sending RPC to namenode. For a given partition, there is one task ID in charge. So it can update those internal maps with hosts/racks of datanode locations and the task ID. These maps are looked up later when idle executors want to take new tasks. i.e. For each idle executor on a given host, check if the internal map has a host/rack entry marked with preferred task IDs. If yes, send a preferred task to the executor.



pendingTasksForHosts[***a***] = Task ***i***
…[***b***] = Task ***i***
…[***c***] = Task ***i***, Task ***j***
…[***y***] = Task ***j***
…[***z***] = Task ***j***

When the spark driver is about to execute a stage, it computes how to divide the work up into tasks by computing partitions of the RDDs in the stage. (There is one-to-one mapping between tasks and partitions.) When the stage has an input data stored in HDFS, the driver uses NewHadoopRDD. When an instance of NewHadoopRDD is created, it will computes the partitions and gets the list of datanode locations for each partition.

Then the driver creates a TaskSet and hands it over to TaskSetManager. TaskSetManager maintains a number of maps. Keyed by executor, host or rack names, each entry in the map contains a list of task IDs that we prefer sending there.

```
// Set of pending tasks for each executor.
private val pendingTasksForExecutor = new HashMap[String, ArrayBuffer[Int]]

// Set of pending tasks for each host. Similar to pendingTasksForExecutor,
// but at host level.
private val pendingTasksForHost = new HashMap[String, ArrayBuffer[Int]]

// Set of pending tasks for each rack -- similar to the above.
private val pendingTasksForRack = new HashMap[String, ArrayBuffer[Int]]
```

TaskSetManager updates these maps with the datanode locations (tasks(index).preferredLocations below) per partition (Partition ID is task ID, because there is one to one correspondence)

```
  /** Add a task to all the pending-task lists that it should be on. */
  private def addPendingTask(index: Int) {
    for (loc <- tasks(index).preferredLocations) {
      loc match {
        case e: ExecutorCacheTaskLocation =>
          pendingTasksForExecutor.getOrElseUpdate(e.executorId, new ArrayBuffer) += index
        case e: HDFSCacheTaskLocation =>
          val exe = sched.getExecutorsAliveOnHost(loc.host)
          exe match {
            case Some(set) =>
              for (e <- set) {
                pendingTasksForExecutor.getOrElseUpdate(e, new ArrayBuffer) += index
              }
              logInfo(s"Pending task $index has a cached location at ${e.host} " +
                ", where there are executors " + set.mkString(","))
            case None => logDebug(s"Pending task $index has a cached location at ${e.host} "
+
                ", but there are no executors alive there.")
          }
        case _ =>
      }
      pendingTasksForHost.getOrElseUpdate(loc.host, new ArrayBuffer) += index
      for (rack <- sched.getRackForHost(loc.host)) {
        pendingTasksForRack.getOrElseUpdate(rack, new ArrayBuffer) += index
      }
    }

    if (tasks(index).preferredLocations == Nil) {
      pendingTasksWithNoPrefs += index
    }

    allPendingTasks += index  // No point scanning this whole list to find the old task
there
  }
```

Note that sched.getRackForHost is only implemented for YARN currently. The code is in YarnScheduler, which is subclass of TaskSchedulerImpl. RackResolver is a Hadoop utility class:

```
// By default, rack is unknown
override def getRackForHost(hostPort: String): Option[String] = {
  val host = Utils.parseHostPort(hostPort)._1
  Option(RackResolver.resolve(sc.hadoopConfiguration, host).getNetworkLocation)
}
```

Then, these maps are consulted with when the driver finds an idle executor and is about to dispatch tasks there:

```
  /**
   * Respond to an offer of a single executor from the scheduler by finding a task
```

```
   *
   * NOTE: this function is either called with a maxLocality which
   * would be adjusted by delay scheduling algorithm or it will be with a special
   * NO_PREF locality which will be not modified
   *
   * @param execId the executor Id of the offered resource
   * @param host  the host Id of the offered resource
   * @param maxLocality the maximum locality we want to schedule the tasks at
   */
  @throws[TaskNotSerializableException]
  def resourceOffer(
      execId: String,
      host: String,
      maxLocality: TaskLocality.TaskLocality)
    : Option[TaskDescription] =
  {
      ...
      dequeueTask(execId, host, allowedLocality).map { case ((index, taskLocality,
speculative)) =>
      ...
  }



  /**
   * Dequeue a pending task for a given node and return its index and locality level.
   * Only search for tasks matching the given locality constraint.
   *
   * @return An option containing (task index within the task set, locality, is
speculative?)
   */
  private def dequeueTask(execId: String, host: String, maxLocality: TaskLocality.Value)
    : Option[(Int, TaskLocality.Value, Boolean)] =
  {
    for (index <- dequeueTaskFromList(execId, host, getPendingTasksForExecutor(execId))) {
      return Some((index, TaskLocality.PROCESS_LOCAL, false))
    }

    if (TaskLocality.isAllowed(maxLocality, TaskLocality.NODE_LOCAL)) {
      for (index <- dequeueTaskFromList(execId, host, getPendingTasksForHost(host))) {
        return Some((index, TaskLocality.NODE_LOCAL, false))
      }
    }

    if (TaskLocality.isAllowed(maxLocality, TaskLocality.NO_PREF)) {
      // Look for noPref tasks after NODE_LOCAL for minimize cross-rack traffic
      for (index <- dequeueTaskFromList(execId, host, pendingTasksWithNoPrefs)) {
        return Some((index, TaskLocality.PROCESS_LOCAL, false))
      }
    }
```

```
    if (TaskLocality.isAllowed(maxLocality, TaskLocality.RACK_LOCAL)) {
      for {
        rack <- sched.getRackForHost(host)
        index <- dequeueTaskFromList(execId, host, getPendingTasksForRack(rack))
      } {
        return Some((index, TaskLocality.RACK_LOCAL, false))
      }
    }

    if (TaskLocality.isAllowed(maxLocality, TaskLocality.ANY)) {
      for (index <- dequeueTaskFromList(execId, host, allPendingTasks)) {
        return Some((index, TaskLocality.ANY, false))
      }
    }

    // find a speculative task if all others tasks have been scheduled
    dequeueSpeculativeTask(execId, host, maxLocality).map {
      case (taskIndex, allowedLocality) => (taskIndex, allowedLocality, true)}
  }
```

The TaskSetManager code above is called by TaskSchedulerImpl, when the driver got a list of idle executors. Note TaskSchedulerImpl below calls the TaskSetManager code above multiple times, each time with different preferred locality level. It does so because it wants to schedule as many tasks as possible across all the idle executors with the best locality first, then only then move down to the next best locality level. I.e. PROCESS_LOCAL, NODE_LOCAL, NO_PREF, RACK_LOCAL, ANY. (There is also related work for delaying-scheduling that balances locality and fairness.)

```
  def resourceOffers(offers: IndexedSeq[WorkerOffer]): Seq[Seq[TaskDescription]] =
synchronized {
    ....
    val sortedTaskSets = rootPool.getSortedTaskSetQueue
    ...

    // Take each TaskSet in our scheduling order, and then offer it each node in increasing
order
    // of locality levels so that it gets a chance to launch local tasks on all of them.
    // NOTE: the preferredLocality order: PROCESS_LOCAL, NODE_LOCAL, NO_PREF, RACK_LOCAL,
ANY
    for (taskSet <- sortedTaskSets) {
      var launchedAnyTask = false
      var launchedTaskAtCurrentMaxLocality = false
      for (currentMaxLocality <- taskSet.myLocalityLevels) {
        do {
          launchedTaskAtCurrentMaxLocality = resourceOfferSingleTaskSet(
            taskSet, currentMaxLocality, shuffledOffers, availableCpus, tasks)
```

```
          launchedAnyTask |= launchedTaskAtCurrentMaxLocality
        } while (launchedTaskAtCurrentMaxLocality)
      }
      if (!launchedAnyTask) {
        taskSet.abortIfCompletelyBlacklisted(hostToExecutors)
      }
    }

    if (tasks.size > 0) {
      hasLaunchedTask = true
    }
    return tasks
  }

...

  private def resourceOfferSingleTaskSet(
      taskSet: TaskSetManager,
      maxLocality: TaskLocality,
      shuffledOffers: Seq[WorkerOffer],
      availableCpus: Array[Int],
      tasks: IndexedSeq[ArrayBuffer[TaskDescription]]) : Boolean = {
    var launchedTask = false
    // nodes and executors that are blacklisted for the entire application have already been
    // filtered out by this point
    for (i <- 0 until shuffledOffers.size) {
      val execId = shuffledOffers(i).executorId
      val host = shuffledOffers(i).host
      if (availableCpus(i) >= CPUS_PER_TASK) {
        try {
          for (task <- taskSet.resourceOffer(execId, host, maxLocality)) {
            tasks(i) += task
            val tid = task.taskId
            taskIdToTaskSetManager(tid) = taskSet
            taskIdToExecutorId(tid) = execId
            executorIdToRunningTaskIds(execId).add(tid)
            availableCpus(i) -= CPUS_PER_TASK
            assert(availableCpus(i) >= 0)
            launchedTask = true
          }
        } catch {
          ...
        }
      }
    }
    return launchedTask
  }
```

Appendix C: HDFS client code

A HDFS client such as a task on a Spark Executor wants to read a block. It sends a RPC request to the namenode asking for the list of datanode locations that has the block. The RPC request has the client host name too. The namenode response has the datanode list sorted by the proximity with the closest datanode first. The client then reads data from the first datanode in the list.

When an app reads a HDFS file, it creates an instance DFSInputStream, which is a stream for reading HDFS data.
The app will then issue read calls specifying the byte offset range to read:

```
public int read(long position, byte[] buffer, int offset, int length)
```

The byte offset range maps to HDFS blocks. The read method will find the mapping by asking namenode. Below, dfsClient.getLocatedBlocks() is the client code sending requests to namenode:

```
/**
  * Get blocks in the specified range.
  * Includes only the complete blocks.
  * Fetch them from the namenode if not cached.
  */
 private List<LocatedBlock> getFinalizedBlockRange(
     long offset, long length) throws IOException {
   ...
     while(remaining > 0) {
       LocatedBlock blk = fetchBlockAt(curOff, remaining, true);
       ...
     }
     return blockRange;
   }
 }
```

```
/** Fetch a block from namenode and cache it */
private LocatedBlock fetchBlockAt(long offset, long length, boolean useCache)
    throws IOException {
  synchronized(infoLock) {
    int targetBlockIdx = locatedBlocks.findBlock(offset);
    if (targetBlockIdx < 0) { // block is not cached
      targetBlockIdx = LocatedBlocks.getInsertIndex(targetBlockIdx);
      useCache = false;
    }
    if (!useCache) { // fetch blocks
      final LocatedBlocks newBlocks = (length == 0)
          ? dfsClient.getLocatedBlocks(src, offset)
          : dfsClient.getLocatedBlocks(src, offset, length);
      if (newBlocks == null || newBlocks.locatedBlockCount() == 0) {
        throw new EOFException("Could not find target position " + offset);
      }
      locatedBlocks.insertRange(targetBlockIdx, newBlocks.getLocatedBlocks());
    }
    return locatedBlocks.get(targetBlockIdx);
  }
}
```

Then, read will retrieve one block at a time and fill out the output buffer with the block data. A given HDFS block may have multiple copies, each residing on a different datanode. LocatedBlock above from namenode has the list of datanodes. When the request is sent to datanode, the request also has the hostname of the client. namenode takes this information into account and sorts the datanode list in the proximity order, with the closest datanodes first in the list. So read simply picks the best datanode from the head of the list and uses it. Below, block.getLocations returns the sorted list of datanode info entries.

```
/**
  * Get the best node from which to stream the data.
  * @param block LocatedBlock, containing nodes in priority order.
  * @param ignoredNodes Do not choose nodes in this array (may be null)
  * @return The DNAddrPair of the best node. Null if no node can be chosen.
  */
protected DNAddrPair getBestNodeDNAddrPair(LocatedBlock block,
    Collection<DatanodeInfo> ignoredNodes) {
  DatanodeInfo[] nodes = block.getLocations();
  StorageType[] storageTypes = block.getStorageTypes();
  DatanodeInfo chosenNode = null;
  StorageType storageType = null;
  if (nodes != null) {
    for (int i = 0; i < nodes.length; i++) {
      if (!deadNodes.containsKey(nodes[i])
          && (ignoredNodes == null || !ignoredNodes.contains(nodes[i]))) {
        chosenNode = nodes[i];
        // Storage types are ordered to correspond with nodes, so use the same
```

```
        // index to get storage type.
        if (storageTypes != null && i < storageTypes.length) {
          storageType = storageTypes[i];
        }
        break;
      }
    }
  }
  if (chosenNode == null) {
    reportLostBlock(block, ignoredNodes);
    return null;
  }
  final String dnAddr =
      chosenNode.getXferAddr(dfsClient.getConf().isConnectToDnViaHostname());
  DFSClient.LOG.debug("Connecting to datanode {}", dnAddr);
  InetSocketAddress targetAddr = NetUtils.createSocketAddr(dnAddr);
  return new DNAddrPair(chosenNode, targetAddr, storageType);
}
```

Here's the DFSClient code for getLocatedBlocks above.

```
public LocatedBlocks getLocatedBlocks(String src, long start, long length)
    throws IOException {
  try (TraceScope ignored = newPathTraceScope("getBlockLocations", src)) {
    return callGetBlockLocations(namenode, src, start, length);
  }
}

/**
 * @see ClientProtocol#getBlockLocations(String, long, long)
 */
static LocatedBlocks callGetBlockLocations(ClientProtocol namenode,
    String src, long start, long length)
    throws IOException {
  try {
    return namenode.getBlockLocations(src, start, length);
  } catch(RemoteException re) {
    throw re.unwrapRemoteException(AccessControlException.class,
        FileNotFoundException.class,
        UnresolvedPathException.class);
  }
}
```

Here's the protocol definition of the request at ClientProtocol.java:

```
/**
  * Get locations of the blocks of the specified file
  * within the specified range.
  * DataNode locations for each block are sorted by
  * the proximity to the client.
```

```
   * <p>
   * Return {@link LocatedBlocks} which contains
   * file length, blocks and their locations.
   * DataNode locations for each block are sorted by
   * the distance to the client's address.
   * <p>
   * The client will then have to contact
   * one of the indicated DataNodes to obtain the actual data.
   *
   * @param src file name
   * @param offset range start offset
   * @param length range length
   ...
   */
  @Idempotent
  LocatedBlocks getBlockLocations(String src, long offset, long length)
      throws IOException;
```

Here's the namenode side code FSNamesystem handling getBlockLocations request. Notice the very first argument, which is inserted by the server code automatically, is clientMachine, the host name of the client. (See getClientMachine in NameNodeRpcServer. getClientMachine may return IP instead of the host name. We may have to check both scenario when we get down to details) Also note that the routine ends by sorting the located blocks:

```
/**
 * Get block locations within the specified range.
 * @see ClientProtocol#getBlockLocations(String, long, long)
 */
LocatedBlocks getBlockLocations(String clientMachine, String srcArg,
    long offset, long length) throws IOException {
  ...
  LocatedBlocks blocks = res.blocks;
  sortLocatedBlocks(clientMachine, blocks);
  return blocks;
}

private void sortLocatedBlocks(String clientMachine, LocatedBlocks blocks) {
  if (blocks != null) {
    List<LocatedBlock> blkList = blocks.getLocatedBlocks();
    if (blkList == null || blkList.size() == 0) {
      // simply return, block list is empty
      return;
    }
    blockManager.getDatanodeManager().sortLocatedBlocks(clientMachine,
        blkList);

    // lastBlock is not part of getLocatedBlocks(), might need to sort it too
    LocatedBlock lastBlock = blocks.getLastLocatedBlock();
```

```
      if (lastBlock != null) {
        ArrayList<LocatedBlock> lastBlockList = Lists.newArrayList(lastBlock);
        blockManager.getDatanodeManager().sortLocatedBlocks(clientMachine,
            lastBlockList);
      }
    }
  }
```

sortLocatedBlocks in DatanodeManager in turn just calls sortByDistance in NetworkTopology:

```
  private void sortLocatedBlock(final LocatedBlock lb, String targetHost,
      Comparator<DatanodeInfo> comparator) {
    // As it is possible for the separation of node manager and datanode,
    // here we should get node but not datanode only .
    boolean nonDatanodeReader = false;
    Node client = getDatanodeByHost(targetHost);
    if (client == null) {
      nonDatanodeReader = true;
      ...
    }
    if(nonDatanodeReader) {
      networktopology.sortByDistanceUsingNetworkLocation(client,
          lb.getLocations(), activeLen);
    } else {
      networktopology.sortByDistance(client, lb.getLocations(), activeLen);
    }
    // must update cache since we modified locations array
    lb.updateCachedStorageInfo();
  }

  /** @return the datanode descriptor for the host. */
  public DatanodeDescriptor getDatanodeByHost(final String host) {
    return host2DatanodeMap.getDatanodeByHost(host);
  }
```

sortByDistance in NetworkTopology will compute weights that is a proxmity score between reader and datanode, and sort the list using weights:

```
/**
   * Sort nodes array by network distance to <i>reader</i>.
   * <p/>
   * As an additional twist, we also randomize the nodes at each network
   * distance. This helps with load balancing when there is data skew.
   *
   * @param reader    Node where data will be read
   * @param nodes     Available replicas with the requested data
   * @param activeLen Number of active nodes at the front of the array
   * @param nonDataNodeReader True if the reader is not a datanode
   */
```

```
  private void sortByDistance(Node reader, Node[] nodes, int activeLen,
      boolean nonDataNodeReader) {
    /** Sort weights for the nodes array */
    int[] weights = new int[activeLen];
    for (int i=0; i<activeLen; i++) {
      if(nonDataNodeReader) {
        weights[i] = getWeightUsingNetworkLocation(reader, nodes[i]);
      } else {
        weights[i] = getWeight(reader, nodes[i]);
      }
    }
    // Add weight/node pairs to a TreeMap to sort
    TreeMap<Integer, List<Node>> tree = new TreeMap<Integer, List<Node>>();
    for (int i=0; i<activeLen; i++) {
      int weight = weights[i];
      Node node = nodes[i];
      List<Node> list = tree.get(weight);
      if (list == null) {
        list = Lists.newArrayListWithExpectedSize(1);
        tree.put(weight, list);
      }
      list.add(node);
    }

    int idx = 0;
    for (List<Node> list: tree.values()) {
      if (list != null) {
        Collections.shuffle(list, r);
        for (Node n: list) {
          nodes[idx] = n;
          idx++;
        }
      }
    }
    Preconditions.checkState(idx == activeLen,
        "Sorted the wrong number of nodes!");
  }
}
```

getWeight computes weight by finding the common network address component, from node name to rack name, etc.

```
  /**
    * Returns an integer weight which specifies how far away {node} is away from
    * {reader}. A lower value signifies that a node is closer.
    *
    * @param reader Node where data will be read
    * @param node Replica of data
    * @return weight
    */
```

```
protected int getWeight(Node reader, Node node) {
  // 0 is local, 2 is same rack, and each level on each node increases the
  //weight by 1
  //Start off by initializing to Integer.MAX_VALUE
  int weight = Integer.MAX_VALUE;
  if (reader != null && node != null) {
    if(reader.equals(node)) {
      return 0;
    }
    int maxReaderLevel = reader.getLevel();
    int maxNodeLevel = node.getLevel();
    int currentLevelToCompare = maxReaderLevel > maxNodeLevel ?
        maxNodeLevel : maxReaderLevel;
    Node r = reader;
    Node n = node;
    weight = 0;
    while(r != null && r.getLevel() > currentLevelToCompare) {
      r = r.getParent();
      weight++;
    }
    while(n != null && n.getLevel() > currentLevelToCompare) {
      n = n.getParent();
      weight++;
    }
    while(r != null && n != null && !r.equals(n)) {
      r = r.getParent();
      n = n.getParent();
      weight+=2;
    }
  }
  return weight;
}
```