*/\*\* Gets or computes an RDD partition. Used by RDD.iterator() when an RDD is cached. \*/*

**def** getOrCompute[T](  
 rdd: RDD[T],  
 partition: Partition,  
 context: TaskContext,  
 storageLevel: StorageLevel): Iterator[T] = {  
  
 **val** key = *RDDBlockId*(rdd.*id*, partition.index)  
 logDebug(s"Looking for partition **$**key")  
 blockManager.get(key) **match** {  
 **case** *Some*(blockResult) =>  
 // Partition is already materialized, so just return its values  
 **val** existingMetrics = context.taskMetrics  
 .getInputMetricsForReadMethod(blockResult.readMethod)  
 existingMetrics.incBytesRead(blockResult.bytes)  
  
 **val** iter = blockResult.data.asInstanceOf[Iterator[T]]  
 **new** InterruptibleIterator[T](context, iter) {  
 **override def** next(): T = {  
 existingMetrics.incRecordsRead(1)  
 delegate.next()  
 }  
 }  
 **case** None =>  
 // Acquire a lock for loading this partition  
 // If another thread already holds the lock, wait for it to finish return its results  
 **val** storedValues = acquireLockForPartition[T](key)  
 **if** (storedValues.isDefined) {  
 **return new** InterruptibleIterator[T](context, storedValues.get)  
 }  
  
 // Otherwise, we have to load the partition ourselves  
 **try** {  
 logInfo(s"Partition **$**key not found, computing it")  
 **val** computedValues = rdd.computeOrReadCheckpoint(partition, context)  
  
 // If the task is running locally, do not persist the result  
 **if** (context.isRunningLocally) {  
 **return** computedValues  
 }  
  
 // Otherwise, cache the values and keep track of any updates in block statuses  
 **val** updatedBlocks = **new** ArrayBuffer[(BlockId, BlockStatus)]  
 **val** cachedValues = putInBlockManager(key, computedValues, storageLevel, updatedBlocks)  
 **val** metrics = context.taskMetrics  
 **val** lastUpdatedBlocks = metrics.*updatedBlocks*.getOrElse(*Seq*[(BlockId, BlockStatus)]())  
 metrics.*updatedBlocks* = *Some*(lastUpdatedBlocks ++ updatedBlocks.toSeq)  
 **new** InterruptibleIterator(context, cachedValues)  
  
 } **finally** {  
 *loading*.synchronized {  
 *loading*.remove(key)  
 *loading*.notifyAll()  
 }  
 }  
 }  
}