CSE6331: Cloud Computing

Leonidas Fegaras

University of Texas at Arlington

Map-Reduce Design Patterns

HDFS Specification

- All specs are in XML files in a configuration directory $HADOOP_CONF_DIR
- You may set a property programatically: conf.set("fs.defaultFS","hdfs://localhost/")
- The default is to run on local (standalone) mode
- The file "slaves" contains the names of the datanodes
- The most important specification is the file system (in core-site.xml):
  fs.defaultFS   hdfs://hadoop.uta.edu:9000/
- Default file system is the local file system:
  fs.defaultFS   hdfs://localhost/
- Other specifications are in hdfs-site.xml:

dfs.blocksize	256m
dfs.replication	3
dfs.namenode.name.dir	file:///home/dfs/namenode
dfs.datanode.data.dir	file:///home/dfs/datanode

Running Map-Reduce

- Running locally on test data (using the default configuration):

```
hadoop jar simple.jar edu.uta.cse6331.Simple input.txt output
```

- Pseudo-distributed mode: distributed mode where there is only one node in the cluster: your PC
- Running in pseudo- or fully-distributed mode, needs a special configuration directory

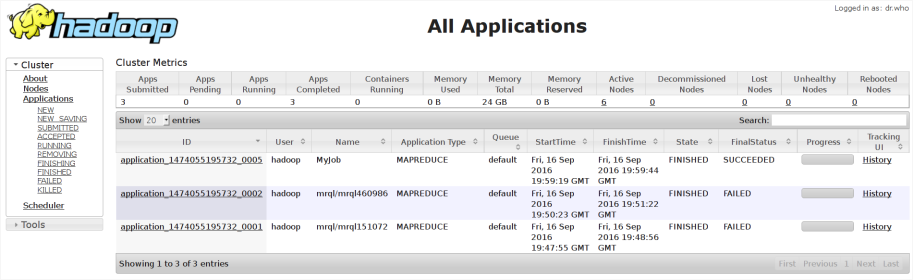
```
export HADOOP_CONF_DIR=/home/hadoop/conf
```

- ... or you may specify the conf directory in the hadoop command:

```
hadoop --config /home/hadoop/conf jar simple.jar \
     edu.uta.cse6331.Simple input.txt output
```

Monitoring Map-Reduce

Monitoring Yarn jobs on the Web using http://myhost:8088/, where myhost is the ResourceMananager hostname



Logged in as: dr.who

All Applications

- Cluster
  - About
  - Nodes
  - Applications
    - NEW
    - NEW_SAVING
    - SUBMITTED
    - ACCEPTED
    - RUNNING
    - REMOVING
    - FINISHING
    - FINISHED
    - FAILED
    - KILLED
  - Scheduler
- Tools

Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	Active Nodes	Decommissioned Nodes	Lost Nodes	Unhealthy Nodes	Rebooted Nodes
3	0	0	3	0	0 B	24 GB	0 B	6	0	0	0	0

Show 20 entries

Search:

ID	User	Name	Application Type	Queue	StartTime	FinishTime	State	FinalStatus	Progress	Tracking UI
application_1474055195732_0005	hadoop	MyJob	MAPREDUCE	default	Fri, 16 Sep 2016 19:59:19 GMT	Fri, 16 Sep 2016 19:59:44 GMT	FINISHED	SUCCEEDED		History
application_1474055195732_0002	hadoop	mrql/mrql460986	MAPREDUCE	default	Fri, 16 Sep 2016 19:50:23 GMT	Fri, 16 Sep 2016 19:51:22 GMT	FINISHED	FAILED		History
application_1474055195732_0001	hadoop	mrql/mrql151072	MAPREDUCE	default	Fri, 16 Sep 2016 19:47:55 GMT	Fri, 16 Sep 2016 19:48:56 GMT	FINISHED	FAILED		History

Showing 1 to 3 of 3 entries

First Previous 1 Next Last

HDFS Commands

- HDFS commands are similar to Unix commands:

```
hdfs dfs -ls /user/hadoop                    # list HDFS files
hdfs dfs -mkdir -p /user/hadoop              # create an HDFS directory
hdfs dfs -put a.txt /user/hadoop/a.txt       # copy from local to HDFS
hdfs dfs -get /user/hadoop/a.txt a.txt       # copy from HDFS to local
hdfs dfs -getmerge /user/hadoop/output output.txt   # -get with merge
hdfs dfs -rm -r /user/hadoop/output          # remove an HDFS directory
```

- Administrative commands:

```
hdfs namenode -format   # create an empty filesystem
start-dfs.sh            # start the dfs tasks (NameNode and DataNodes)
stop-dfs.sh             # stop all dfs tasks
hdfs fsck /             # check and repair the filesystem
hdfs dfsadmin -report
```

Yarn Specifications and Commands

- Specifications are in yarn-site.xml:

yarn.resourcemanager.hostname	hadoop.uta.edu
yarn.nodemanager.resource.memory-mb	4096
yarn.scheduler.minimum-allocation-mb	1024
yarn.scheduler.minimum-allocation-vcores	1
yarn.scheduler.maximum-allocation-vcores	2

- Administrative commands:

```
start-yarn.sh   # start yarn (the ResourceManager and the NodeManagers
stop-yarn.sh                  # stop all yarn tasks
yarn node -list               # list all nodes
yarn application -list        # list all running applications
yarn application -kill id     # kill the application with this id
```

Map-Reduce Specifications

- Must add shuffling support on Yarn in yarn-site.xml:
  yarn.nodemanager.aux-services mapreduce_shuffle
  yarn.nodemanager.aux-services.mapreduce_shuffle.class
  org.apache.hadoop.mapred.ShuffleHandler
- Other specifications are in mapreduce-site.xml:

mapreduce.framework.name	yarn
mapred.job.tracker	hadoop.uta.edu:9001
mapreduce.map.memory.mb	1024
mapreduce.reduce.memory.mb	1024
io.sort.mb	20

A SLURM Script to Run in Local Mode on Comet

```
#SBATCH --job-name="simple.local"
#SBATCH --output="simple.local.out"
#SBATCH --partition=shared
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --export=ALL
#SBATCH --time=10

module load hadoop/2.6.0

rm -rf output
hadoop --config $HOME jar simple.jar edu.uta.cse6331.Simple simple.txt  output
```

Run in Distributed Mode Using myhadoop

Allocates 3 nodes for the Hadoop cluster

```
#SBATCH --job-name="simple"
#SBATCH --output="simple.distr.out"
#SBATCH --partition=compute
#SBATCH --nodes=3
#SBATCH --ntasks-per-node=1
#SBATCH --export=ALL
#SBATCH --time=60

export HADOOP_CONF_DIR=/home/$USER/cometcluster
module load hadoop/2.6.0
myhadoop-configure.sh
start-dfs.sh
start-yarn.sh
hdfs dfs -mkdir -p /user/$USER
hdfs dfs -put simple.txt /user/$USER/simple.txt
hadoop jar simple.jar edu.uta.cse6331.Simple /user/$USER/simple.txt /user/$USER/output
rm -rf output
mkdir output
hdfs dfs -get /user/$USER/output/part* output
stop-yarn.sh
stop-dfs.sh
myhadoop-cleanup.sh
```

Debugging and Fine-Tuning Map-Reduce

- Each task produces a logfile called syslog, a file for data sent to stdout, and a file for stderr
- Stored in the userlogs subdirectory in the logs directory of the node
- When you print using System.out.print or System.err.print, it goes to logs
- You can use built-in or user-defined counters in map and reduce tasks

```
enum MyCounter { map_input, map_output }

context.getCounter(MyCounter.map_output).increment(1);
counters.findCounter(MyCounter.map_output).getValue();
```

- By default, each map and reduce task is allocated 1GB of memory and one virtual core.
- You may change this via the following properties:
  mapreduce.map.memory.mb mapreduce.reduce.memory.mb
  mapreduce.map.cpu.vcores mapreduce.reduce.cpu.vcores

Fine-Tuning Map-Reduce

- How many mappers?
  - Hard to control
  - Hadoop will use 1 map task per split, each split is at most 1 block
  - If there are fewer nodes than map tasks, each node gets multiple tasks
  - You can create multiple splits per block by setting mapreduce.input.fileinputformat.split.maxsize

    eg, for 256MB block, you can set it to 128MB to get 2 splits/block
  - How many splits/block? if you have n cores per node, you want n splits/block so that the block is processed locally by n cores
- If you have a large number of small files (each less than 1 block), Map-Reduce will be slow (too much bookkeeping overhead)
  - Use CombineFileInputFormat, instead of FileInputFormat
  - CombineFileInputFormat packs many files into each split so that each mapper has more to process

How Many Reducers?

- Default is 1 reducer
- Not good: the job will be very slow since all the intermediate data will flow through a single reduce task
- To use n reducers, use job.setNumReduceTasks(n)
- Increasing the number of reducers $\Rightarrow$ more parallelism
- ... but don't create too many small files
- Rule of thumb: each reducer should produce at least one block

Chaining Multiple Map-Reduce Jobs Together

```
public class MyClass extends Configured implements Tool {
    @Override
    public int run ( String[ ] args ) throws Exception {
        Configuration conf = getConf();
        Job job = Job.getInstance(conf,"Job1");
        ...
        job.waitForCompletion(true);
        Job job2 = Job.getInstance(conf,"Job2");
        ...
        job2.waitForCompletion(true);
        return 0;
    }

    public static void main ( String[ ] args ) throws Exception {
        ToolRunner.run(new Configuration(),new MyClass(),args);
    }
}
```

Need a Combiner for Good Performance

- It does partial reduction after map but before shuffling
- It reduces the amount of data passing through the shuffle
- Types:

$$\begin{aligned} map(k, v) &\rightarrow list(< k', v' >) \\ combine(k', list(v')) &\rightarrow list(< k', v' >) \\ reduce(k', list(v')) &\rightarrow list(< k'', v'' >) \end{aligned}$$

- But it requires an associative accumulator
- Consider $\text{avg}(S) = \text{sum}(S)/\text{count}(S)$

  $\text{avg}(S_1 \cup S_2)$ cannot be expressed in terms of $\text{avg}(S_1)$ and $\text{avg}(S_2)$
- Solution: use an associative accumulation for the combiner and do the final aggregation at the reducer
- For $\text{avg}(S)$, we use the pair $(\text{sum}(S), \text{count}(S))$
  - Accumulating $(\text{sum}(S), \text{count}(S))$ is done at the combiner

    $(\text{sum}(S_1 \cup S_2), \text{count}(S_1 \cup S_2)) = (\text{sum}(S_1) + \text{sum}(S_2), \text{count}(S_1) + \text{count}(S_2))$
  - Final accumulation and avg is done at the reducer

The Group-by Example Revisited

```
class MyMapper extends Mapper<Object,Text,IntWritable,DoubleWritable> {
    public void map ( Object key, Text value, Context context )
                throws IOException, InterruptedException {
        Scanner s = new Scanner(value.toString()).useDelimiter(",");
        int x = s.nextInt();
        double y = s.nextDouble();
        context.write(new IntWritable(x),new DoubleWritable(y));
        s.close();
    }
}
class MyReducer extends Reducer<IntWritable,DoubleWritable,IntWritable,DoubleWritable> {
    public void reduce ( IntWritable key, Iterable<DoubleWritable> values,
                         Context context )
                throws IOException, InterruptedException {
        double sum = 0.0;
        long count = 0;
        for (DoubleWritable v: values) {
            sum += v.get();
            count++;
        };
        context.write(key,new DoubleWritable(sum/count));
    }
}
```

The Group-by Example with a Combiner

```
class AvgPair implements Writable {
   public double sum;
   public long count;
   AvgPair( double sum, in count ) { this.sum = sum; this.count = count; }
   public void write ( DataOutput out ) throws IOException { ... }
   public void readFields ( DataInput in ) throws IOException { ... }
}
class MyMapper extends Mapper<Object,Text,IntWritable,AvgPair> {
    public void map ( Object key, Text value, Context context )
                throws IOException, InterruptedException {
        Scanner s = new Scanner(value.toString()).useDelimiter(",");
        int x = s.nextInt();
        double y = s.nextDouble();
        context.write(new IntWritable(x),new AvgPair(y,(long)1));
        s.close();
    }
}
```

The Group-by Example with a Combiner (cont.)

```
class MyCombiner extends Reducer<IntWritable,AvgPair,IntWritable,AvgPair> {
    public void reduce ( IntWritable key, Iterable <AvgPair> values, Context context )
                throws IOException, InterruptedException {
        double sum = 0.0;
        long count = 0;
        for (AvgPair v: values) {
            sum += v.sum;
            count += v.count;
        };
        context.write(key,new AvgPair(sum,count));
} }
class MyReducer extends Reducer<IntWritable,AvgPair,IntWritable,DoubleWritable> {
    public void reduce ( IntWritable key, Iterable <AvgPair> values, Context context )
                throws IOException, InterruptedException {
        double sum = 0.0;
        long count = 0;
        for (AvgPair v: values) {
            sum += v.sum;
            count += v.count;
        };
        context.write(key,new DoubleWritable(sum/count));
} }

job.setCombinerClass(MyCombiner.class); // inside the run method
```

In-Mapper Combining

Instead of using a combiner, one may combine values in the mapper using a static Hashtable H (you may flush it periodically)

```
@Override
protected void setup ( Context context ) throws IOException,InterruptedExcep
    H = new Hashtable<Key,Value>();
}
@Override
protected void cleanup ( Context context ) throws IOException,InterruptedExc
    for (Key key: H)
        context.write(key,H.get(key);
}
@Override
public void map ( Key key, Value value, Context context ) throws IOExceptio
   // ... calculate the new mapper output key2 and value2 from key and value
   if (H.get(key2) == null)
       H.put(key2,value2);
   else H.put(key2,acc(H[key2],value2));
}
```

How to Avoid Key Serialization/Deserialization

- Every time Map-Reduce compares keys in sorting/grouping, it deserializes the keys
- Expensive if the key is a complex object

```
class Pair implements WritableComparable {
    public int X;
    public int Y;
    public int compareTo ( Pair o ) {
        return (X == o.X) ? Y-o.Y : X-o.X;
    }
    public void write ( DataOutput out ) throws IOException { ... }
    public void readFields ( DataInput in ) throws IOException { ... }
}
```

- We want to compare keys without deserializing them

How to Avoid Key Serialization/Deserialization (cont.)

- A RawComparator compares two serialized keys (ie, as bytes sequences)

```
class PairComparator implements RawComparator<Pair> {
   public static int compare ( byte[ ] x, int xs, int xl,
                               byte[ ] y, int ys, int yl ) {
      int k = WritableComparator.readInt(x,xs)
              -WritableComparator.readInt(y,ys);
      if (k != 0)
         return k;
      return WritableComparator.readInt(x,xs+4)
             -WritableComparator.readInt(y,ys+4);
   }
}

job.setSortComparatorClass(PairComparator.class);
job.setGroupingComparatorClass(PairComparator.class);
```

Use a Custom Partitioner to Reduce Data Skew

- The default partitioner (can be overwritten):

```
class MyPartitioner extends Partitioner<KEY,VALUE> {
    public int getPartition ( KEY key, VALUE value, int numPartitions )
        return Math.abs(key.hashCode()) % numPartitions;
    }
}
job.setPartitionerClass(MyPartitioner.class); // inside the run method
```

- Another way: you overwrite the hashCode method
- Total sorting: you want to use many reducers but need to use a partitioner that respects the total order of the output ⇒ range partitioning

```
class MyPartitioner extends Partitioner<LongWritable,VALUE> {
    public int getPartition ( LongWritable key, VALUE value, int numPar
        int n = maxKeyValue % numPartitions + 1;
        return key/n;
    }
}
```

Custom InputFormat

- Normally, not needed
- Here is an example where you may need it
- You want to process many random nodes (without any input data)
- Example: generate random data for performance evaluation
- You want a generator that generates small input splits, one per worker node
- Generate num tiny files, where each one has 1 pair (start,length):

```
void generator ( String directory , long num, long length ) {
   for ( int i = 0; i < num; i++ ) {
      Path path = new Path(directory);
      SequenceFile.Writer writer
         = SequenceFile.createWriter(path.getFileSystem(conf),conf,path,
                                     LongWritable.class,LongWritable.class,
                                     SequenceFile.CompressionType.NONE);
      writer.append(new LongWritable(i*length),new LongWritable(length));
      writer.close();
   }
}
```

Custom InputFormat (cont.)

```
class GeneratorInputFormat
                extends FileInputFormat<LongWritable,LongWritable> {
public static class GeneratorRecordReader
                extends RecordReader<LongWritable,LongWritable> {
   final long offset; final long size; long index;
   public GeneratorRecordReader ( FileSplit split,
                        TaskAttemptContext context ) throws IOException {
      Configuration conf = context.getConfiguration();
      Path path = split.getPath();
      FileSystem fs = path.getFileSystem(conf);
      SequenceFile.Reader reader
          = new SequenceFile.Reader(path.getFileSystem(conf),path,conf);
      LongWritable key = new LongWritable();
      LongWritable value = new LongWritable();
      reader.next(key,value);
      offset = key.get();
      size = value.get();
      index = 0;
      reader.close();   }
```

Custom InputFormat (cont.)

```
public boolean nextKeyValue () throws IOException {
    return index++ < size;
}
public LongWritable getCurrentKey () throws IOException {
    return new LongWritable(index);
}
public LongWritable getCurrentValue () throws IOException {
    return new LongWritable(offset+index);
}
public float getProgress () throws IOException {
    return index / (float) size ;
}
}
public RecordReader<LongWritable,LongWritable>
        createRecordReader ( InputSplit split ,
                             TaskAttemptContext context ) throws IOException
    return new GeneratorRecordReader((FileSplit) split ,context );
}
```

Map-Backed Join

- Want to do a join between two datasets R and S:

```
select r.C, s.D
from R as r, S as s
where r.A = s.B
```

- Assume that one dataset, say R, can fit in the memory of every node
- The map-backed join algorithm:
  - Broadcast the R dataset to all worker nodes
  - Before a Map-Reduce job starts, create a built hash table from R
  - The Map-Reduce job needs a map stage only (no reduce stage)
  - A mapper joins its input split of S with the entire R hash table by probing the hash table
- In the Map-Reduce job:

```
static Hashtable<Key,Rvalue> built_table;
job.addCacheFile(new URI("R dataset path"));
```

Map-Backed Join

```
@Override
protected void setup ( Context context ) throws IOException,InterruptedExcep
   URI[ ] paths = context.getCacheFiles ();
   Configuration conf = context. getConfiguration ();
   built_table = new Hashtable<Key,RValue>();
   SequenceFile.Reader reader
         = new SequenceFile.Reader(conf,
                       SequenceFile.Reader. file (new Path(paths[0])));
   Key key = new Key();
   RValue value = new RValue();
   while (reader.next(key,value))
         built_table .put(key,value );
   reader. close ();
}
@Override
public void map ( Key key, SValue svalue, Context context ) throws IOExcept
   RValue rvalue = built_table .get(key);
   context. write(key,concatenate( rvalue , svalue ));
}
```

Reduce-Side Join

- If neither R nor S can fit in the memory of a node
- Map-Reduce now has two mappers:
  - a mapper for R that uses the join key $R.A$
  - a mapper for S that uses the join key $S.B$
- The mappers will send the R and S values with the same keys $R.A = S.B$ to the same reducer
- Must tag R tuples with 0 and S tuples with 1 so that the reducer can tell them apart

```
class Mapper1
    method map ( key, r )
        emit(r.A,(0,r));

class Mapper2
    method map ( key, s )
        emit(s.B,(1,s));

class Reducer
    method reduce ( key, values )
        for each (0, r) ∈ values
            for each (1, s) ∈ values
                emit(key,(r.C,s.D));
```

Reduce-Side Join Example

- A join between Employee and Department available at `http://lambda.uta.edu/cse6331/tests/Join.java`

```
class EmpDept implements Writable {
    public short tag;
    public Employee employee;
    public Department department;
     ...
}

MultipleInputs.addInputPath(job,new Path("e.txt"),TextInputFormat.class,
                            EmployeeMapper.class);
MultipleInputs.addInputPath(job,new Path("d.txt"),TextInputFormat.class,
                            DepartmentMapper.class);
job.setReducerClass(ResultReducer.class);
```

Reduce-Side Join (cont.)

```
class EmployeeMapper extends Mapper<Object,Text,IntWritable,EmpDept > {
    @Override
    public void map ( Object key, Text value, Context context ) throws IOExc
        Scanner s = new Scanner(value.toString()). useDelimiter(",");
        Employee e = new Employee(s.next(),s.nextInt(), s.next());
        context.write(new IntWritable(e.dno),new EmpDept(e));
        s.close();
    }
}
class DepartmentMapper extends Mapper<Object,Text,IntWritable,EmpDept >
    @Override
    public void map ( Object key, Text value, Context context ) throws IOExc
        Scanner s = new Scanner(value.toString()). useDelimiter(",");
        Department d = new Department(s.next(),s.nextInt());
        context.write(new IntWritable(d.dno),new EmpDept(d));
        s.close();
    }
}
```

Reduce-Side Join (cont.)

```
class ResultReducer extends Reducer<IntWritable,EmpDept,IntWritable,Text> {
    @Override
    public void reduce ( IntWritable key, Iterable<EmpDept> values,
                         Context context ) throws IOException, InterruptedExc
        Vector<Employee> emps = new Vector<Employee>();
        Vector<Department> depts = new Vector<Department>();
        for (EmpDept v: values)
            if (v.tag == 0)
                emps.add(v.employee);
            else depts.add(v.department);
        for ( Employee e: emps )
            for ( Department d: depts )
                context.write(key,new Text(e.name+" "+d.name));
    }
}
```

Application 1: k-Means Clustering

- Each point in 3D space is a record $< X, Y, Z >$
- Calculate one step of the k-means clustering algorithm by deriving k new centroids from the old
- For each point s, find the closest centroid c (minimum distance(c, s))
- Given the set of points closest to a centroid, the new centroid is the center of these points

```
select avg(s.X) as X, avg(s.Y) as Y, avg(s.Z) as Z
  from Points as s
 group by closest_centroid(s)
```

- where closest_centroid (s) is the centroid with the minimum distance from s:

```
(select * from Centroids as c order by distance(c,s))[0]
```

- Since Centroids is small, you can use a map-backed join

Map-Reduce Pseudo-Code for k-Means Clustering

```
setup ()
  read Centroids from distributed cache

map ( key, point ):
  closest ← null
  min = 10000
  for each c in Centroids
      if distance(c, point) < min
          closest ← c
          min = distance(c, point)
  emit( closest , point)

reduce ( centroid , points ):
  c = 0;   sx = sy = sz = 0.0
  for each p in points
      c++
      sx += p.X; sy += p.Y; sz += p.Z
  emit( centroid , new Point(sx/c,sy/c,sz/c))   // new centroid
```

Application 2: PageRank

- Assumption: if important pages are pointing to a page, then this page is important too
- Let $A_1, A_2, \ldots, A_n$ be the pages that point to the page A. Then the PageRank of A is

  $$PR(A) = (1 - d)/N + d * (PR(A_1)/C(A_1) + ... + PR(A_n)/C(A_n))$$

  where $C(A_i)$ is the number of outgoing links from A_i
  N is the total number of pages
- PR is the principal eigenvector of the link matrix of the web
  - can be computed as the fixpoint of the above equation
  - in practice, it is computed incrementally
  - Google computes the relevance of a page for a given search by first computing an TFxIDF relevance and then adjusting it by taking into account the PR of the top-ranked pages

PageRank using Map-Reduce

- A web graph G is represented as a set of links, where each link has a source id, a destination id, the number of outgoing links, and its current PageRank
- One step of the PageRank algorithm derives a new set of edges from the old set, changing only their rank:

  $< \text{source, dest, count, rank} >$
- SQL query:

```
select  m.source, m.dest, m.count, c.rank
  from (select n.dest, sum(n.rank/n.count) as rank
          from G as n
         group by n.dest) as c,
        G as m
 where m.source = c.dest
```

Map-Reduce Pseudo-Code for PageRank

- Type Link = < from: ID, to: ID, count: int, rank: double >
- We avoid the self-join by passing the entire graph to the reducer

```
map ( key, v ):                           // v is a Link
    emit( v.from, v )                     // pass the Link to the reducer
    emit( v.to, v.rank/v.count )          // an incoming PageRank contribution

reduce( id, values ):
    p ← 0.0
    vs ← ∅
    for v in values:
        if v is a pagerank value
            p ← p+v
        else vs ← vs ∪ {v}
    for a in vs:
        emit( < from: a.from, to: a.to, count: a.count, rank: p > )
```

Application 3: Matrix Multiplication

- A sparse matrix M is a collection of records $< V = v, I = i, J = j >$, for $v = M_{ij}$
- Matrix multiplication between two matrices X and Y is $\sum_k X_{ik} * Y_{kj}$
- Let X be an $N * K$ matrix and Y be an $K * M$ matrix
- Naive evaluation: equi-join followed by a group-by with aggregation

```
select  sum(x.V*y.V) as V, x.I,  y.J
  from X as x, Y as y
 where x.J = y.I
 group by x.I,  y.J
```

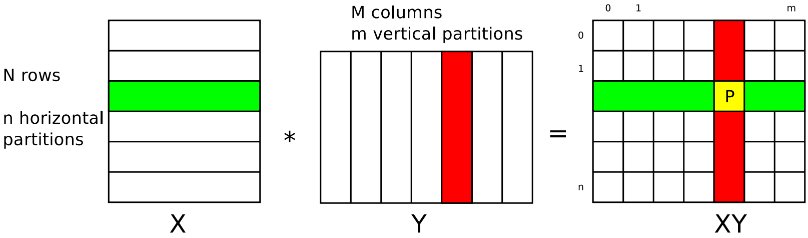
- The intermediate result of the join is of max size $N * K * M$

  n^3 for square matrices
- These data need to be shuffled to cluster nodes for the group-by operation

  very expensive

The SUMMA Algorithm

The SUMMA Algorithm [Geijn & Watts, 1997]



- X is an $N * K$ matrix and Y is an $K * M$ matrix
- distribute the data of X and Y as a grid of $m * n$ partitions
- each partition contains N/n full rows from X and M/m full columns from Y
- can be implemented using 1 map-reduce job

Map-Reduce Implementation of SUMMA

- One mapper for each input matrix, X and Y
- Each mapper emits pairs (key, (tag, data))
  - data is the input data
  - tag is the source number: 0 for X, 1 for Y
  - key is a triple (partition, joinkey, tag)
    - partition is one of the $n * m$ partitions
    - joinkey is the join key value, x.J or y.I
- A value $x \in X$ is sent to all row partitions (x.I mod n, *)
- A value $y \in Y$ is sent to all column partitions (*, y.J mod m)
- Custom partitioning, grouping, and sorting functions:
  - the partition function returns the partition value of the mapper key,
  - the grouping function returns the pair (partition, joinkey), and
  - the sorting is based on:
    - major order: partition
    - minor order: joinkey
    - sub-minor order: tag

Map-Reduce Pseudo-Code for SUMMA

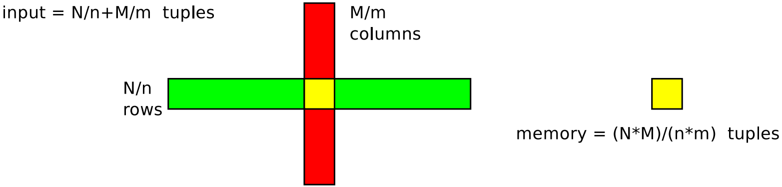
```
mapLeft ( key, x ):
  for each i in 0..m−1
      emit ( ((hashCode(x.I) % n)∗m+i, x.J, 0), (0,x) )

mapRight ( key, y ):
  for each i in 0..n−1
      emit ( ((hashCode(y.J) % m)+m∗i, y.I, 1), (1,y) )
```

```
reduce ( ( partition , joinkey , tag ), values ):
  if ( partition != current_partition )
     flush (H)
     current_partition ← partition
  for each leading (0,x) tuple in values
     insert x into xs
  for each (1,y) tuple in the rest of values
     for each x in xs
        key ← (x.I , y.J)
        if (H[key] is null )
           H[key] ← 0.0
        H[key] ← x.V∗y.V + H[key]
```

```
flush (H):
  for each (key, value) in H
     emit H(key,value)
  clear H
```

```
cleanup ( ):   flush (H)
```

Optimal Number of Partitions $n * m$



- Data replication is $N * K * m + K * M * n$ tuples

  $\Rightarrow$ we want to minimize $N/n + M/m$
- but ... the hash table H must have enough space for $(N * M)/(n * m)$ tuples
- Assuming each worker node has enough memory to fit $\mathcal{T}$ tuples
- Optimal solution: $N/n = M/m = \sqrt{\mathcal{T}}$
- Number of worker nodes $\leq n * m$
  - it can even be just 1: will process one partition at a time