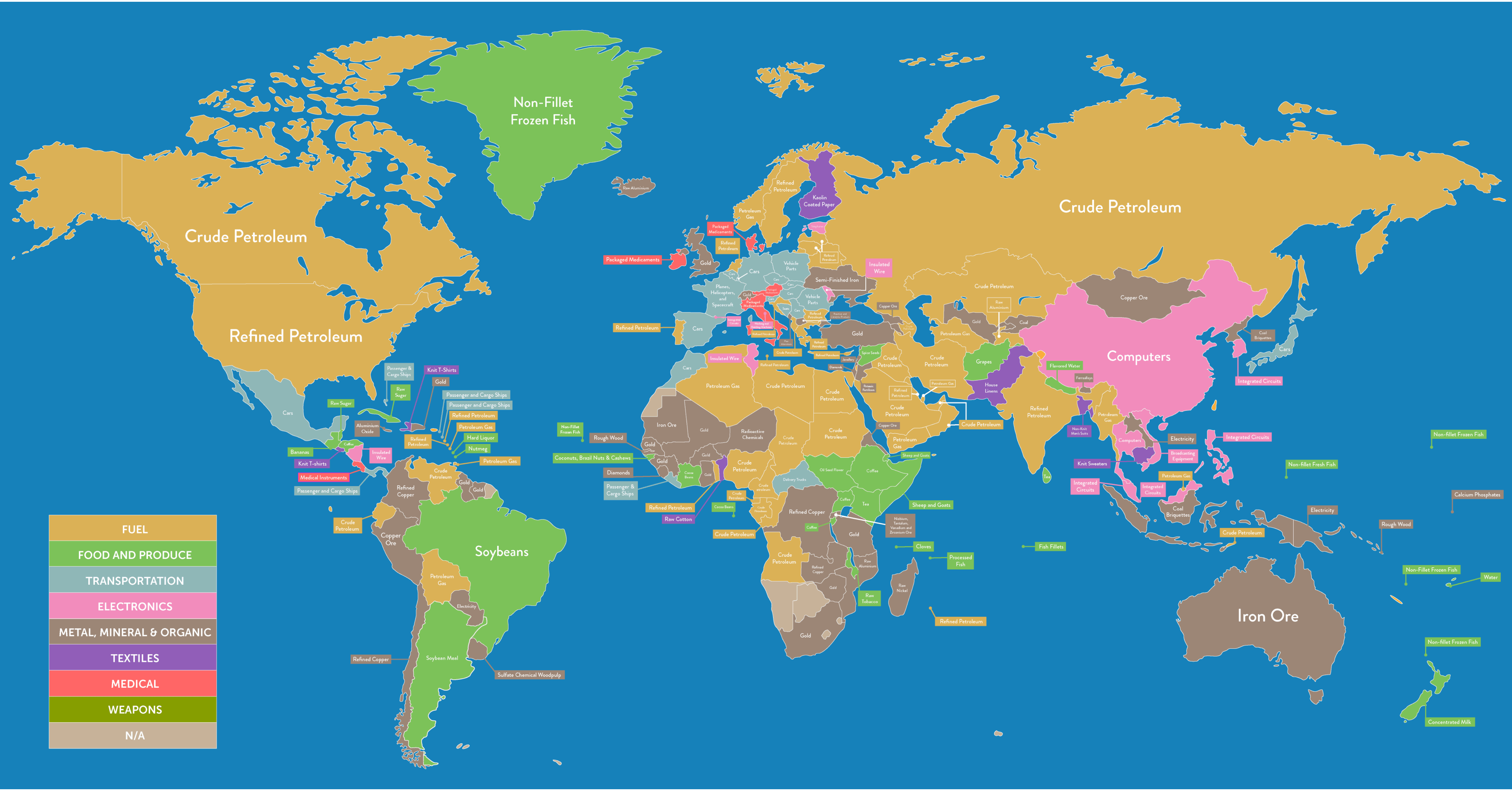
Data Centers: MapReduce

Emmett Witchel

CS380L


Non-Fillet Frozen Fish
Crude Petroleum
Refined Petroleum
Cars
Passenger & Cargo Ships
Raw Sugar
Knit T-Shirts
Gold
Passenger and Cargo Ships
Passenger and Cargo Ships
Refined Petroleum
Petroleum Gas
Hard Liquor
Nutmeg
Petroleum Gas
Raw Sugar
Aluminium Oxide
Refined Petroleum
Coffee
Bananas
Knit T-shirts
Insulated Wire
Medical Instruments
Passenger and Cargo Ships
Crude Petroleum
Refined Copper
Gold
Gold
Crude Petroleum
Copper Ore
Soybeans
Petroleum Gas
Electricity
Soybean Meal
Refined Copper
Sulfate Chemical Woodpulp
Raw Aluminium
Refined Petroleum
Petroleum Gas
Kaolin Coated Paper
Packaged Medicaments
Refined Petroleum
Packaged Medicaments
Gold
Telephones
Refined Petroleum
Cars
Vehicle Parts
Cars
Cars
Cars
Cars
Planes, Helicopters, and Spacecraft
Gold
Packaged Medicaments
Packaged Medicaments
Cars
Seats
Cars
Vehicle Parts
Insulated Wire
Semi-Finished Iron
Refined Petroleum
Reaction and Catalytic Products
Copper Ore
Crude Petroleum
Integrated Circuits
Washing and Bottling Machines
Refined Petroleum
Raw Aluminium
Crude Petroleum
Refined Petroleum
Refined Petroleum
Gold
Spice Seeds
Jewellery
Diamonds
Potassic Fertilizers
Refined Petroleum
Cars
Insulated Wire
Refined Petroleum
Petroleum Gas
Crude Petroleum
Crude Petroleum
Crude Petroleum
Crude Petroleum
Crude Petroleum
Refined Petroleum
Petroleum Gas
Crude Petroleum
Copper Ore
Petroleum Gas
Non-Fillet Frozen Fish
Rough Wood
Coconuts, Brazil Nuts & Cashews
Diamonds
Passenger & Cargo Ships
Iron Ore
Gold
Gold
Gold
Gold
Radioactive Chemicals
Crude Petroleum
Crude Petroleum
Cocoa Beans
Gold
Refined Petroleum
Raw Cotton
Crude Petroleum
Cocoa Beans
Crude Petroleum
Crude Petroleum
Crude Petroleum
Delivery Trucks
Oil Seed Flower
Coffee
Sheep and Goats
Coffee
Tea
Sheep and Goats
Refined Copper
Coffee
Niobium, Tantalum, Vanadium and Zirconium Ore
Gold
Cloves
Processed Fish
Crude Petroleum
Refined Copper
Raw Aluminium
Gold
Raw Tobacco
Raw Nickel
Refined Petroleum
Gold
Crude Petroleum
Raw Aluminium
Gold
Coal
Petroleum Gas
Grapes
House Linens
Flavored Water
Ferroalloys
Copper Ore
Computers
Coal Briquettes
Cars
Integrated Circuits
Refined Petroleum
Tea
Non-Knit Men's Suits
Petroleum Gas
Computers
Electricity
Broadcasting Equipment
Knit Sweaters
Integrated Circuits
Integrated Circuits
Integrated Circuits
Petroleum Gas
Coal Briquettes
Fish Fillets
Electricity
Crude Petroleum
Non-fillet Frozen Fish
Non-fillet Fresh Fish
Calcium Phosphates
Rough Wood
Non-Fillet Frozen Fish
Water
Iron Ore
Non-fillet Frozen Fish
Concentrated Milk
FUEL
FOOD AND PRODUCE
TRANSPORTATION
ELECTRONICS
METAL, MINERAL & ORGANIC
TEXTILES
MEDICAL
WEAPONS
N/A

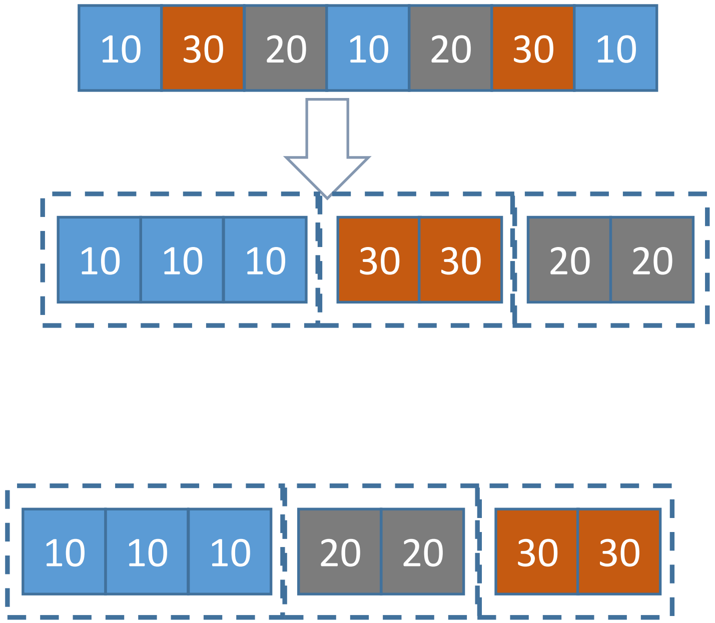
MapReduce faux quiz (5 min, any 2):

- What phenomena can slow down a map task?
- Do reducers wait for all their mappers before starting? Why/why not?
- What machine resources does the shuffle phase consume most?
- Is it safe to re-execute a failed map/reduce job sans cleanup? Why [not]?
- How does MR handle master failure? What are the alternatives?
- Why is[n’t] MR a “step backwards” relative to DBMSs?
- How does MR tolerate failures in 3rd party libraries?
- What is a straggler and how does MR deal with them?
- How are mappers scheduled onto cluster machines?
- In what ways does MR use sorting to improve efficiency?
- How would you design MR differently for a high bisection bandwidth cluster?
- List some aspects of GFS and MR that represent “mechanical sympathy” in design.
- What is a combiner? Why does it need to be associative and commutative? Provide an example.

What is GroupBy?

- Group a collection by key
- Lambda function maps elements → key

```
var res = ints.GroupBy(x => x);
```



10 30 20 10 20 30 10

10 10 10 | 30 30 | 20 20

10 10 10 | 20 20 | 30 30

```
foreach(T elem in PF(ints))
{
   key   = KeyLambda(elem);

   group = GetGroup(key)

   group.Add(elem);
}
```

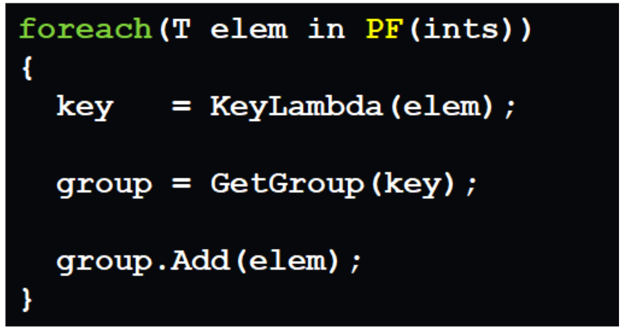
```
Note: sorting is VERY similar
```

GroupBy example

```
import pandas as pd
df = pd.DataFrame({'country': ['US', 'Canada', 'Mexico’],
                   'population': [100, 50, 100]})
df_grouped = df.groupby('country')
```

- df_grouped is an object that groups the data in a DataFrame by the country column
- **`df_grouped.mean()`** would compute the mean per-country

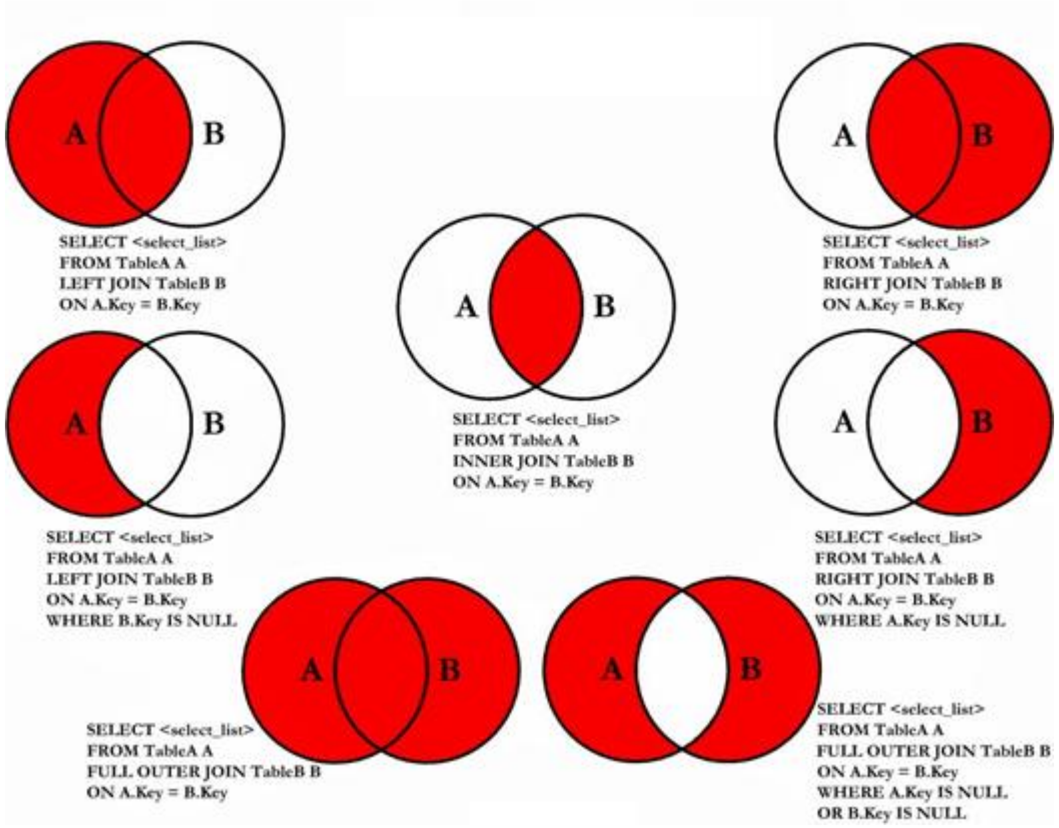
What is Join?



```
foreach(T elem in PF(ints))
{
  key   = KeyLambda(elem);

  group = GetGroup(key);

  group.Add(elem);
}
```



```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```

```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```

```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```

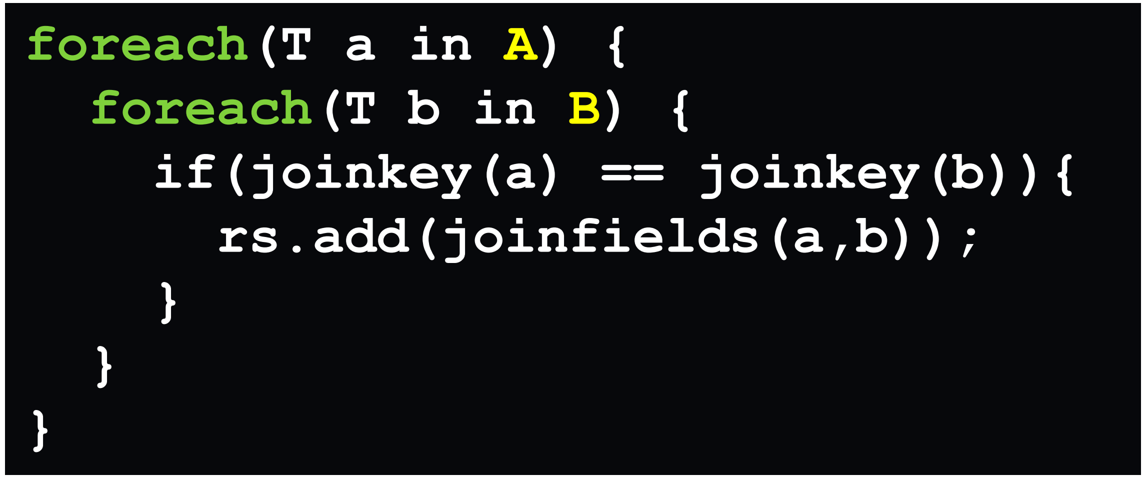
```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```

```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
```

```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
```

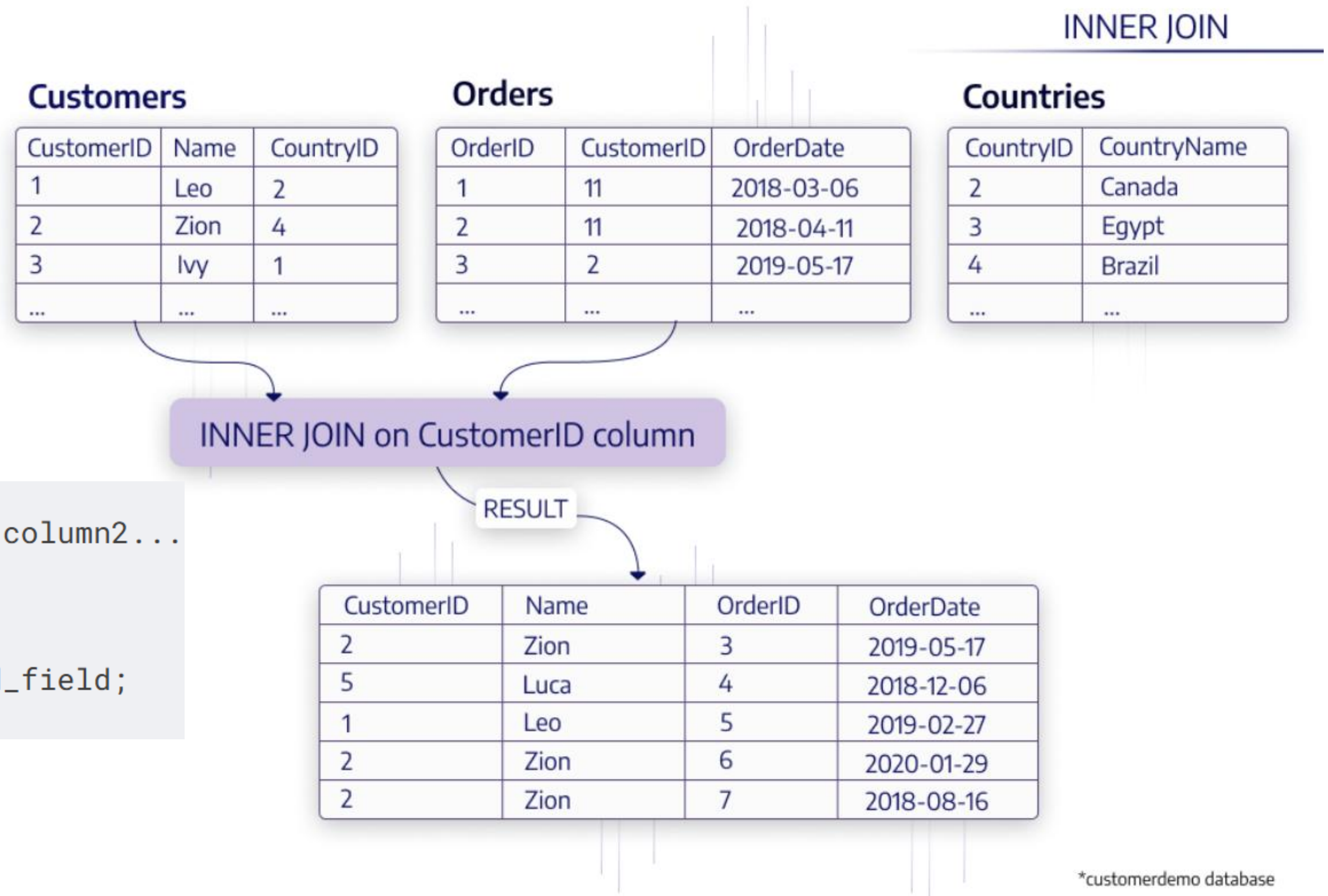
```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
OR B.Key IS NULL
```

- Equi-join / Inner-join: “workhorse”



```
foreach(T a in A) {
  foreach(T b in B) {
    if(joinkey(a) == joinkey(b)){
      rs.add(joinfields(a,b));
    }
  }
}
```

- Note similarity to GroupBy
- Lots of *implementations*
- *How to do this at scale?*



INNER JOIN

Customers

CustomerID	Name	CountryID
1	Leo	2
2	Zion	4
3	Ivy	1
...	...	...

Orders

OrderID	CustomerID	OrderDate
1	11	2018-03-06
2	11	2018-04-11
3	2	2019-05-17
...	...	...

Countries

CountryID	CountryName
2	Canada
3	Egypt
4	Brazil
...	...

INNER JOIN on CustomerID column

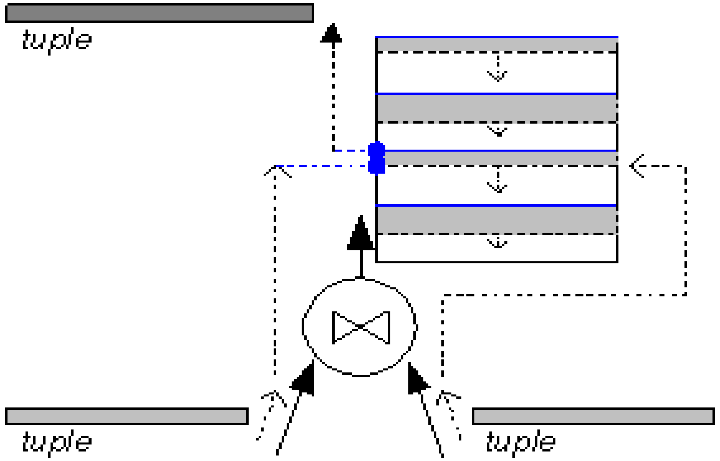
RESULT

CustomerID	Name	OrderID	OrderDate
2	Zion	3	2019-05-17
5	Luca	4	2018-12-06
1	Leo	5	2019-02-27
2	Zion	6	2020-01-29
2	Zion	7	2018-08-16

*customerdemo database

```
SELECT tableA.column1, tableB.column2...
FROM tableA
INNER JOIN tableB
ON tableA.id_field = tableB.id_field;
```

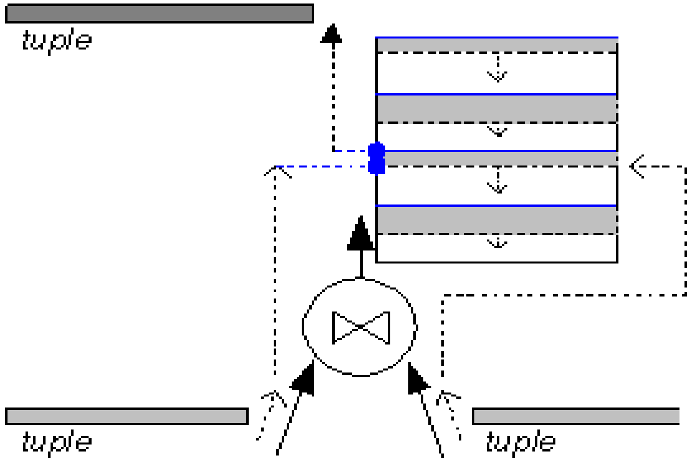
Hash Join



Read entire inner relation into hash table (join attributes as key)

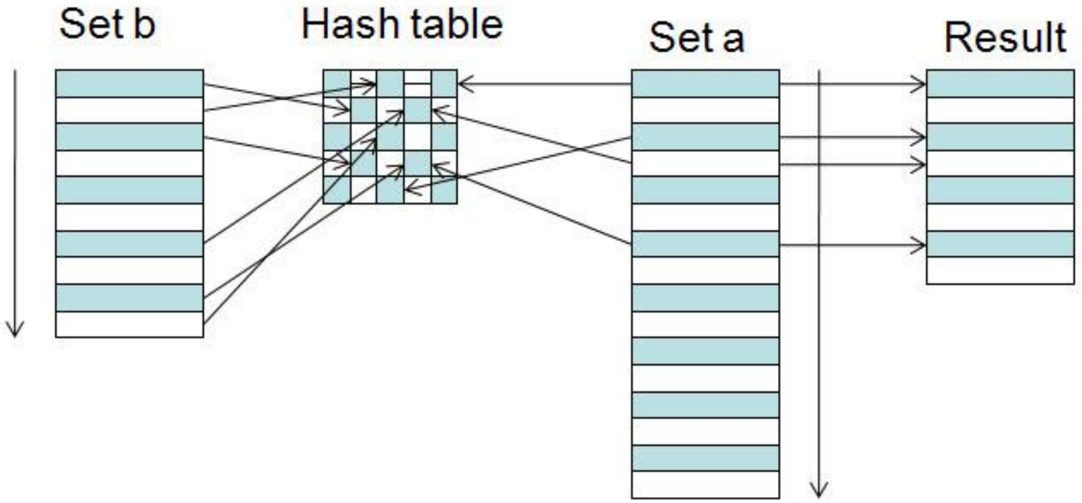
For each tuple from outer, look up in hash table & join

Hash Join



Read entire inner relation into hash table (join attributes as key)

For each tuple from outer, look up in hash table & join



Note:

- same idea hashes data onto cluster nodes
    - removes all:all data exchange
- Alternative for SORTED tables: merge join

You are an engineer at: Hare-brained-scheme.com

Your boss, comes to your office and says:

> "We're going to be hog-nasty rich! We just need a program to search for strings in text files..."

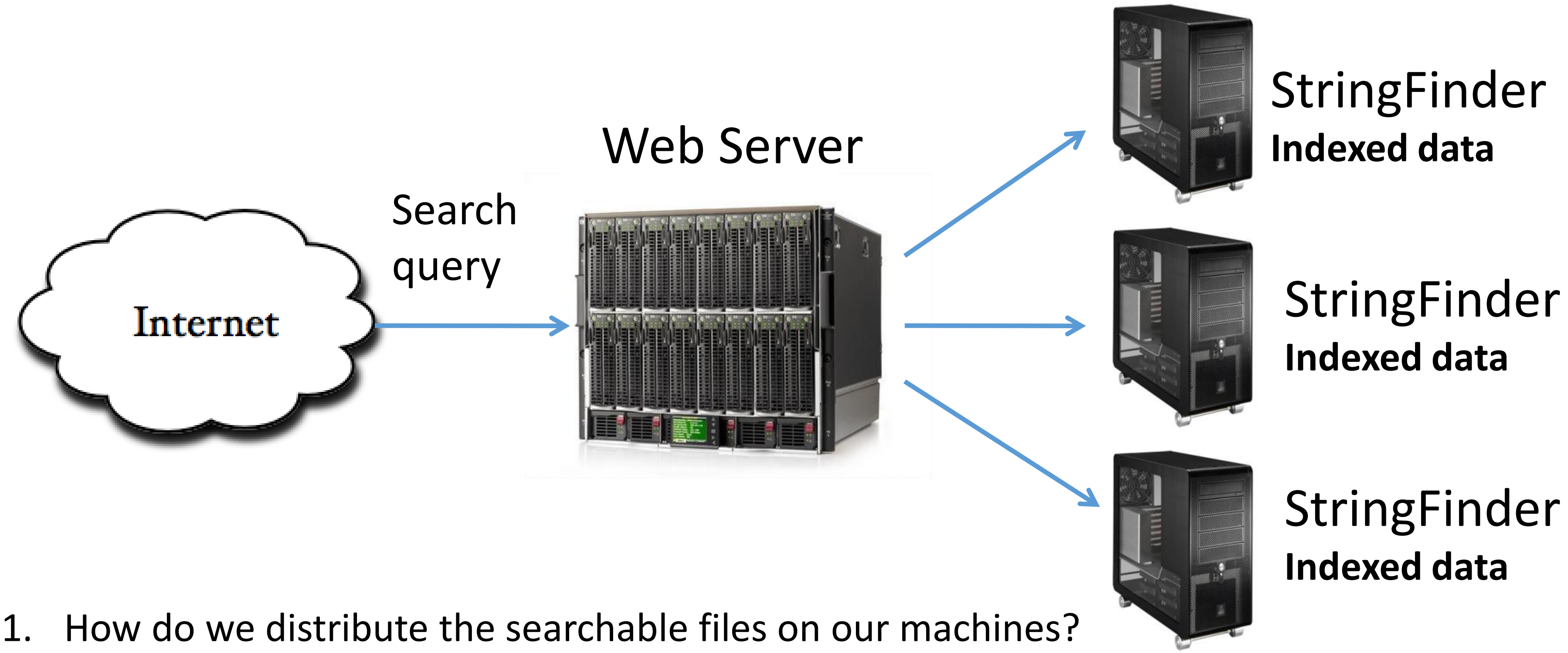
Input: <search_term>, <files>

Output: list of files containing <search_term>

One solution

```
public class StringFinder {
  int main(…) {
      foreach(File f in getInputFiles()) {
         if(f.contains(searchTerm))
           results.add(f.getFileName());
         }
      }
      System.out.println(“Files:” + results.toString());
  }
```

Infrastructure is hard to get right



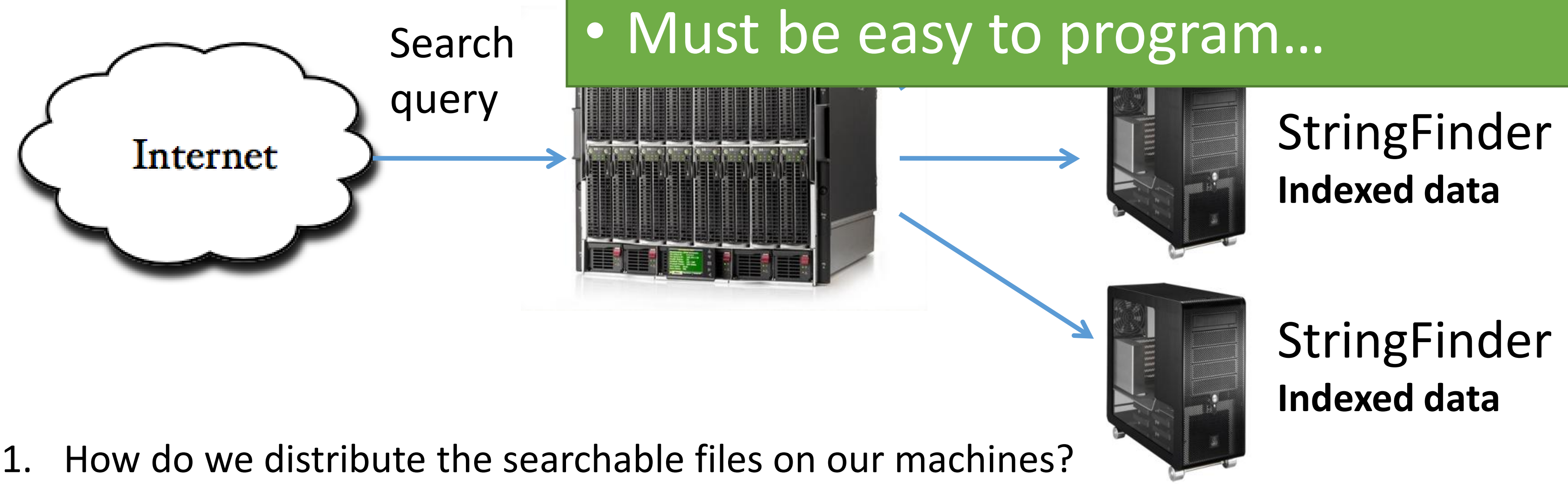
1. How do we distribute the searchable files on our machines?
2. What if our webserver goes down?
3. What if a StringFinder machine dies? How would you know it was dead?
4. **What if marketing comes and says, “well, we also want to show pictures of the earth from space too! Ooh..and the moon too!”**

Infrastructure is ha

StringFinder was the easy part!

You really need general infrastructure.

- Many different tasks
- Want to use hundreds or thousands of PC’s
- Continue to function if something breaks
- Must be easy to program…



1. How do we distribute the searchable files on our machines?
2. What if our webserver goes down?
3. What if a StringFinder machine dies? How would you know it was dead?
4. **What if marketing comes and says, “well, we also want to show pictures of the earth from space too! Ooh..and the moon too!”**

MapReduce

- Programming model + infrastructure
- Write programs that run on lots of machines
- Automatic parallelization and distribution
- Fault-tolerance
- I/O and jobs Scheduling
- Status and monitoring

MapReduce Programming Model

- Input & Output: sets of <key, value> pairs
- Programmer writes 2 functions:

```
map (in_key, in_value) -> list(out_key,
 intermediate_value)
```

  - Processes <k1,v1> pairs
  - Produces intermediate pairs: list(k2, v2)

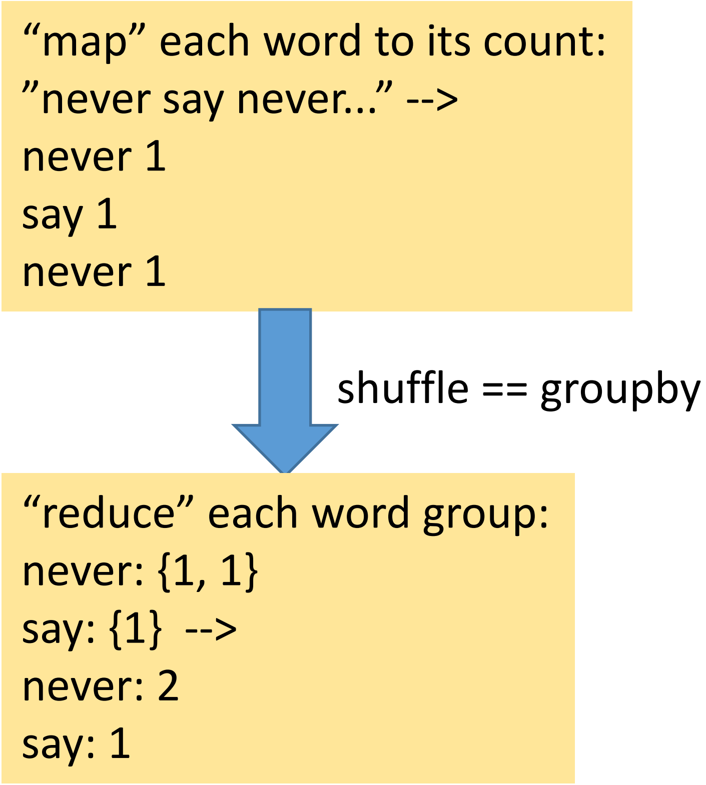
```
reduce (out_key, list(interm_val)) ->
  list(out_value)
```

  - list(k2, v2) -> list(v2)
  - Combines intermediate values for a key
  - Produces a merged set of outputs

Example: Counting Words...

```
map(String input_key, String input_value):
  // input_key: document name
  // input_value: document contents
  for each word w in input_value:
      EmitIntermediate(w, "1");

reduce(String output_key,
        Iterator intermediate_values):
  // output_key: a word
  // output_values: a list of counts
  int result = 0;
  for each v in intermediate_values:
       result += ParseInt(v);
  Emit(AsString(result));
```



“map” each word to its count:
”never say never...” -->
never 1
say 1
never 1

shuffle == groupby

“reduce” each word group:
never: {1, 1}
say: {1} -->
never: 2
say: 1

MapReduce handles all the other details!

Example (2): Indexing

```
public void map() {
  String line = value.toString();
  StringTokenizer itr = new StringTokenizer(line);
    if(itr.countTokens() >= N) {
      while(itr.hasMoreTokens()) {
        word = itr.nextToken()+“|”+key.getFileName();
        output.collect(word, 1);
    }
  }
}
```

Input: a line of text, e.g. “**mistakes were made**” from **myfile.txt**

Output:

mistakes|myfile.txt
were|myfile.txt
made|myfile.txt

Example (3): Indexing

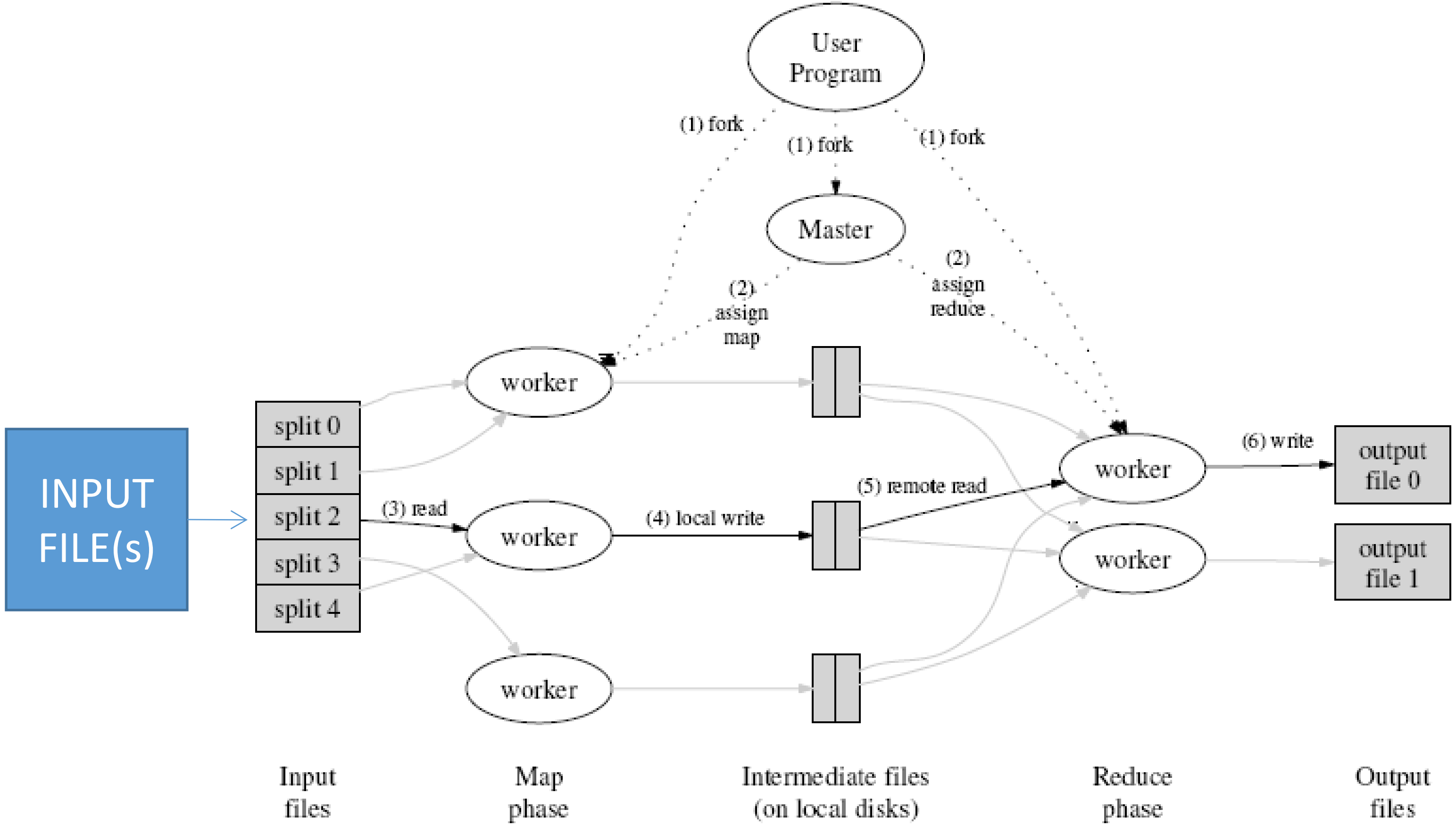
```
public void reduce() {
    int sum = 0;
    while(values.hasNext()) {
        sum += values.next().get();
    }
    output.collect(key, sum);
}
```

Input: a <term,filename> pair, list of occurrences (e.g. {1, 1,..1})

Output:

```
mistakes|myfile.txt   10
were|myfile.txt       45
made|myfile.txt        2
```

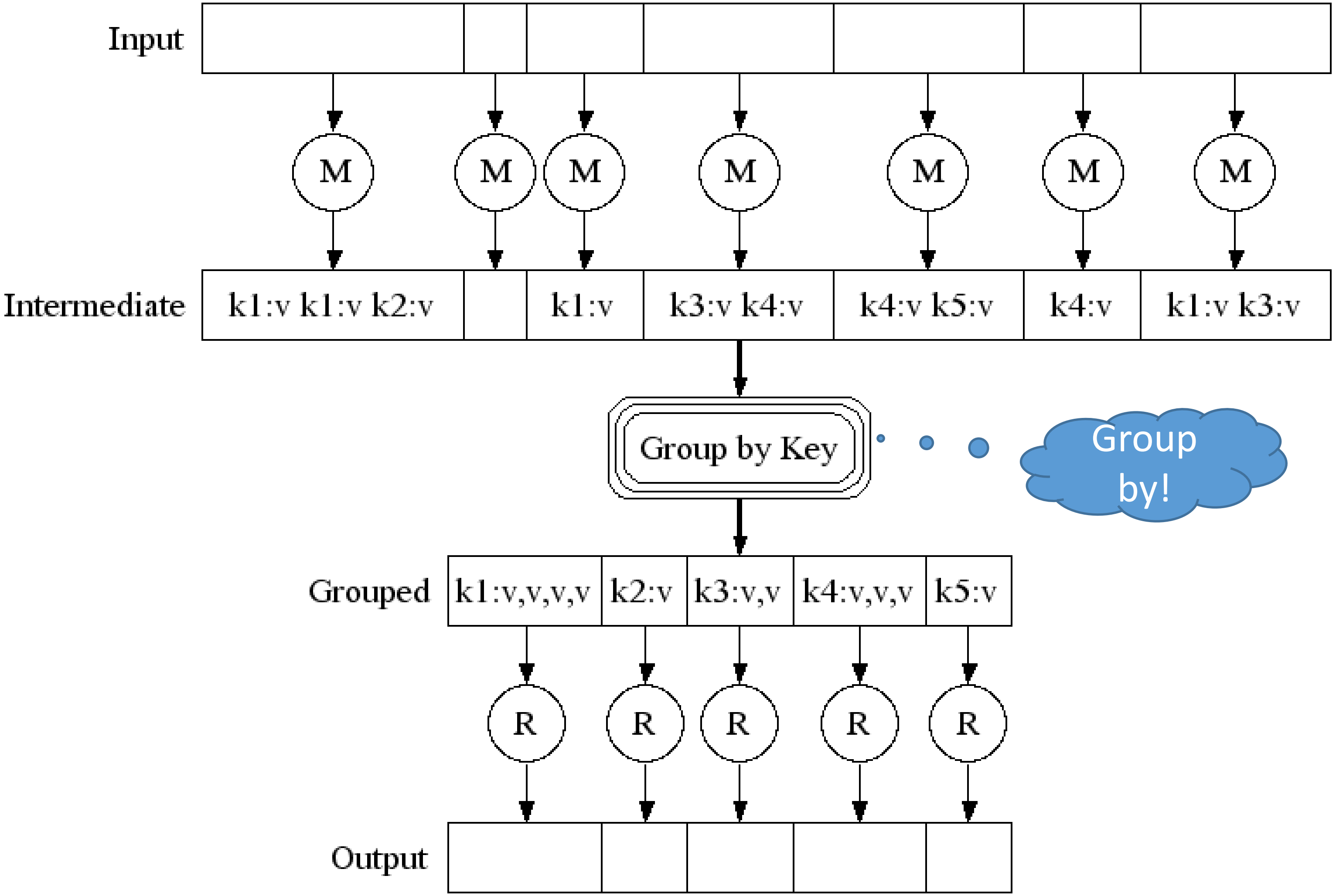
How does parallelization work?



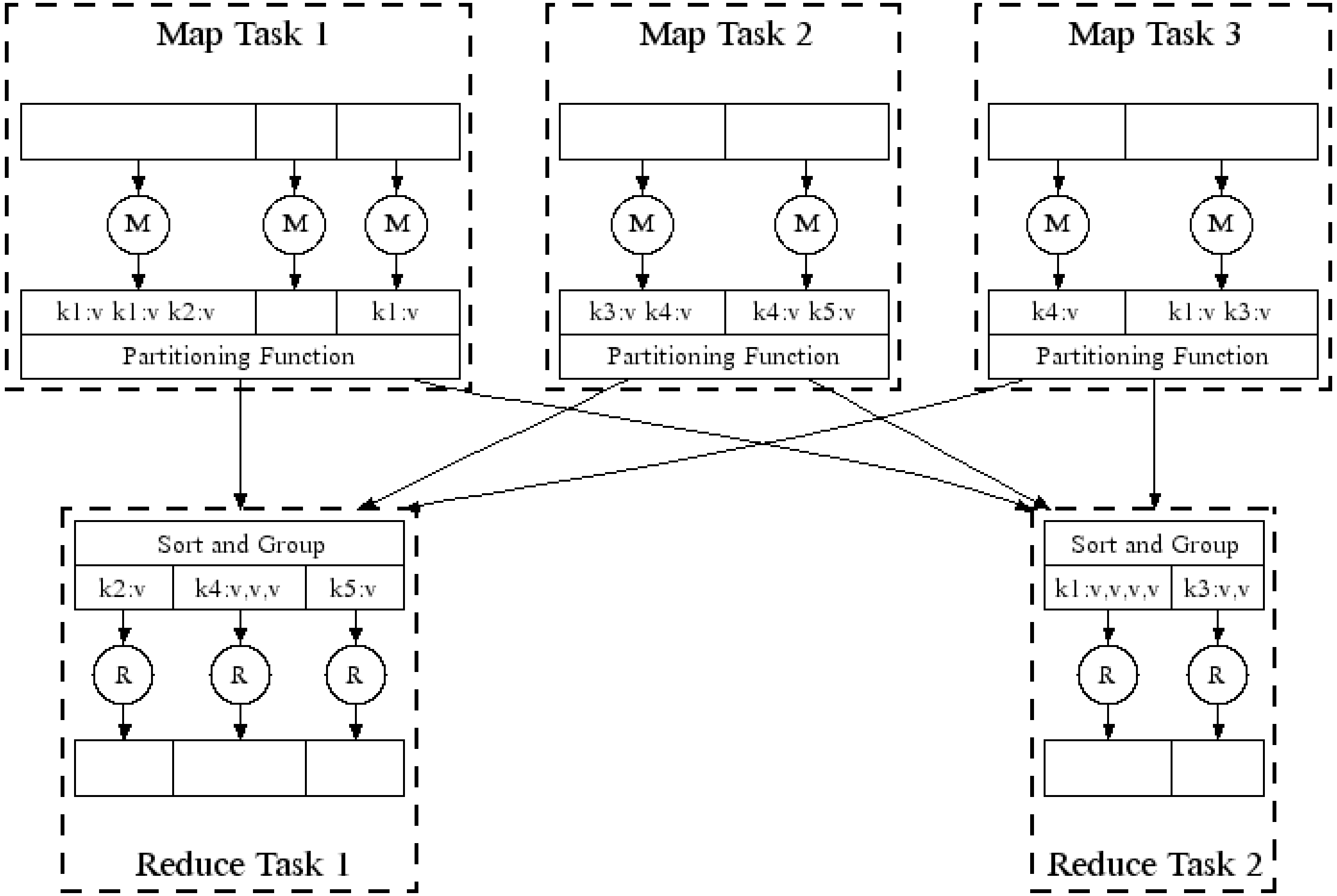
Implementation

- 1000s of 2 core x86, machines 2-4GB RAM
- Limited bisection bandwidth
  - What’s bisection bandwidth?
  - Why is it relevant?
- Local IDE disks + GFS
- Scheduling: job = set of task, scheduler assigns to machines

Execution


Input
M
M
M
M
M
M
M
Intermediate
k1:v k1:v k2:v
k1:v
k3:v k4:v
k4:v k5:v
k4:v
k1:v k3:v
Group by Key
Group by!
Grouped
k1:v,v,v,v
k2:v
k3:v,v
k4:v,v,v
k5:v
R
R
R
R
R
Output

Parallel Execution



Task Granularity And Pipelining

|map tasks| >> |machines| -- why?

Task Granularity And Pipelining

|map tasks| >> |machines| -- why?

- Minimize fault recovery time
- Pipeline map with other tasks
- Easier to load balance dynamically

- What is straggler mitigation (redundant execution)?
  - How much does it help? Why?
- How does MapReduce handle
  - Mapper failures
  - Reducer failures
  - Master failures
- What is the problem of data skew?
  - How does MapReduce deal with it?

Fault Tolerance

- What failures to handle?
- How to detect failures?
- How to respond?
  - For workers?
  - For master?
- How to know tasks complete?

Fault Tolerance

- What failures to handle?
- How to detect failures?
- How to respond?
  - For workers?
  - For master?
- How to know tasks complete?

- Worker failures:
  - Detect via heartbeat
  - Re-execute completed, in-progress map
  - *Re-execute in-progress reducers (why?)*

Fault Tolerance

- What failures to handle?
- How to detect failures?
- How to respond?
  - For workers?
  - For master?
- How to know tasks complete?

- Worker failures:
  - Detect via heartbeat
  - Re-execute completed, in-progress map
  - *Re-execute in-progress reducers (why?)*
- Master failures: re-execute all!

Fault Tolerance

- What failures to handle?
- How to detect failures?
- How to respond?
  - For workers?
  - For master?
- How to know tasks complete?

- Worker failures:
  - Detect via heartbeat
  - Re-execute completed, in-progress map
  - *Re-execute in-progress reducers (why?)*
- Master failures: re-execute all!
- Task completion committed through master

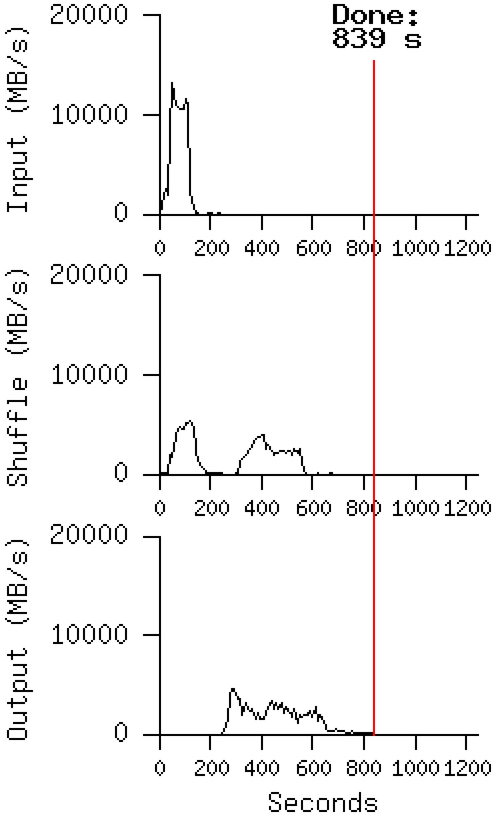
Redundant Execution (straggler mitigation)

- Slow worker can throttle performance: why?
- What makes a worker slow?

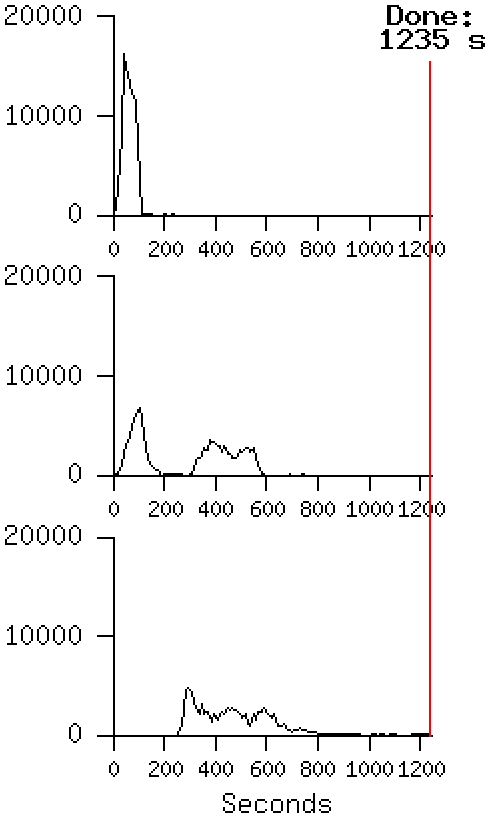
- Solution:

Redundancy performance

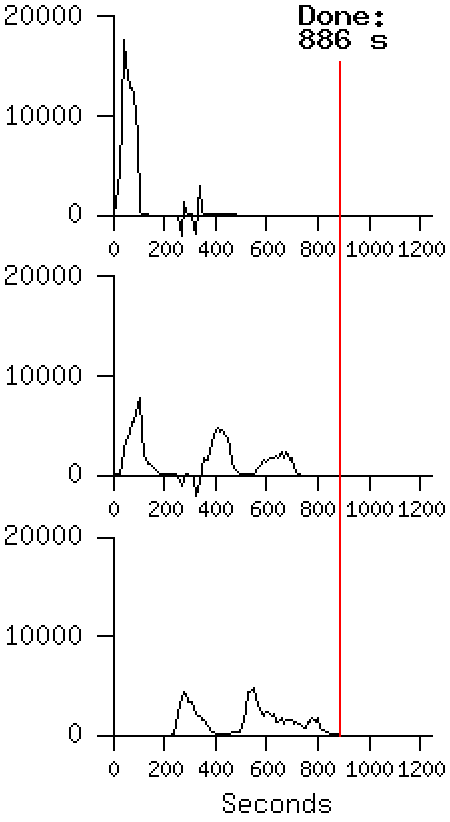
Normal



No backups



200 processes killed



Scheduling for Locality

Master policy:

- What does “locality” mean here?
- How to tailor for GFS?

- Ask GFS for locations of replicated input blocks
- Map task splits: 64MB == GFS block size
- Schedule so that input blocks are on local machine or local rack
- Otherwise rack switch becomes read rate bottleneck

Skipping Bad Records

For Failures on specific inputs

- Can’t always fix/debug
- Seg Fault:
  - Inform master with UDP packet
  - Include record identifier
  - If master sees multiple failures for a record, subsequent workers skip it
- Claim this tolerates bugs in 3rd party libraries
- Is correctness guaranteed?

Other cool stuff

- Sorting guaranteed in reduce partitions: why?
- Compression of intermediate data
- Combiners: what do they do?
- Local execution: anyone debugged an MR program?
- User-defined counters: what for?

The end of your career at:
Hare-brained-scheme.com

Your boss, comes to your office and says:

"I can't believe you used ***MapReduce!!!***
You're fired..."

Why might he say this?

Why is MapReduce backwards?

- Backwards step in programming paradigm
- Sub-optimal: brute force, no indexing
- Not novel: 35 year-old ideas from DBMS lit
- Missing most DBMS features
- Incompatible with most DBMS tools

What’s the problem with MR?

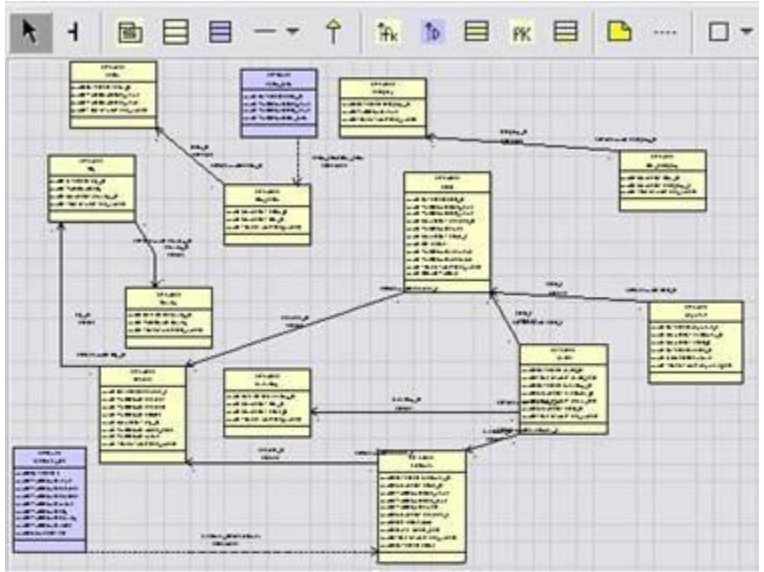
- Map == group-by
- Reduce == aggregate

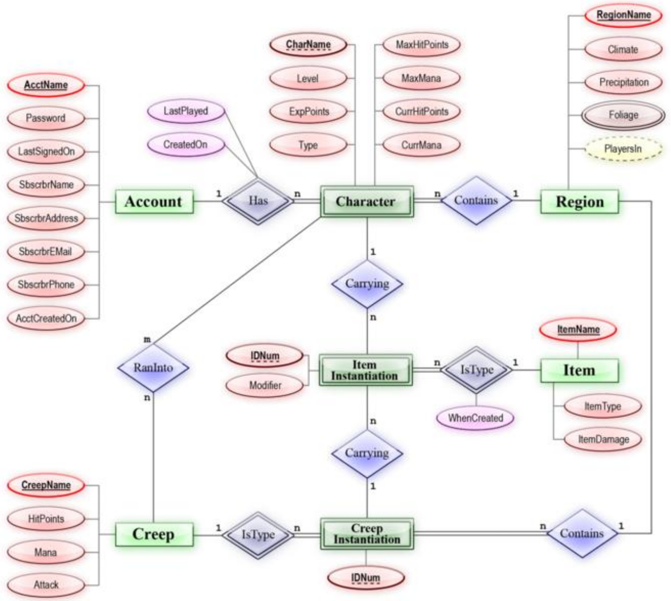
```
SELECT job, COUNT(*) as “numemps"
    FROM employees
    WHERE salary > 1000
    GROUP BY job;
```

- Where is the aggregate in this example?
- DBMS analog make sense? (hello, Lisp?)

Backwards programming model

- Schemas are good (what’s a schema?)
- Separation of schema from app is good (why?)
- High-level access languages are good (why?)





MapReduce is sub-optimal

- Modern DBMSs: hash + B-tree indexes to accelerate data access.
  - Indexes are user-defined
  - Could MR do this?
- No query optimizer! (oh my, terrible…but good for researchers! ☺)
- Skew: wide variance in distribution of keys
  - E.g. “the” more common than “zyzzyva”
- Materializing splits
  - N=1000 mappers → M=500 keys = 500,000 local files
  - 500 reducer instances “pull” these files
  - DBMSs push splits to sockets (no local temp files)

MapReduce: !novel

- Partitioning data sets (map) == Hash join
- Parallel aggregation == reduce
- User-supplied functions differentiates from SQL:
  - POSTGRES user functions, user aggregates
  - PL/SQL: Stored procedures
  - Object databases

MapReduce is feature-poor

Absent features:

- Bulk-loading
- Indexing
- Update operator
- Transactions
- Integrity constraints, referential integrity
- Views

Which of these are important?

Why is it OK for MR to elide them?

MapReduce incompatible with tools

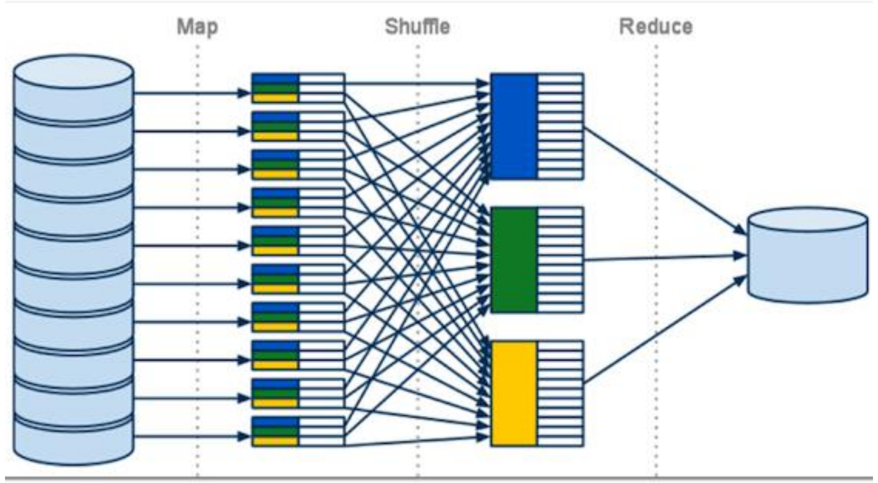
- Report writers
- Business intelligence tools
- Data-mining tools
- Replication tools
- Design tools (UML, embarcadero)

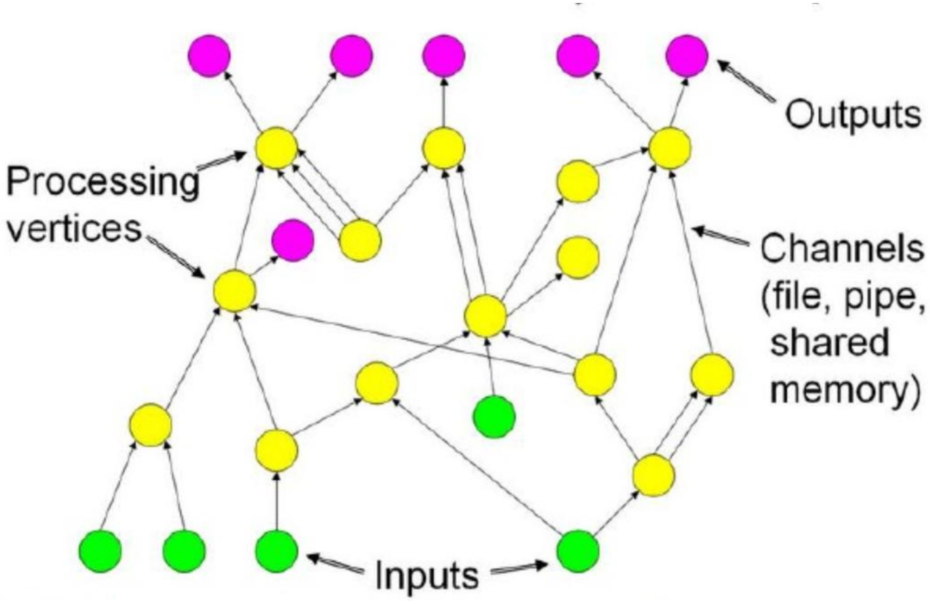
How important are these?

Are these accusations fair?

MapReduce and Dataflow

- MR is a ***dataflow*** engine
- Lots of others
  - Dryad
  - DryadLINQ
  - Dandelion
  - CIEL
  - GraphChi/PowerGraph/Pregel
  - Spark
- Keep this in mind over next few papers





Discussion Questions

(repeated from faux-quiz)

- What phenomena can slow down a map task?
- Do reducers wait for all their mappers before starting? Why/why not?
- What machine resources does the shuffle phase consume most?
- Is it safe to re-execute a failed map/reduce job sans cleanup? Why [not]?
- How does MR handle master failure? What are the alternatives?
- Why is[n't] MR a "step backwards" relative to DBMSs?
- How does MR tolerate failures in 3rd party libraries?
- What is a straggler and how does MR deal with them?
- How are mappers scheduled onto cluster machines?
- In what ways does MR use sorting to improve efficiency?
- How would you design MR differently for a high bisection bandwidth cluster?
- List some aspects of GFS and MR that represent "mechanical sympathy" in design.
- What is a combiner? Why does it need to be associative and commutative? Provide an example.