



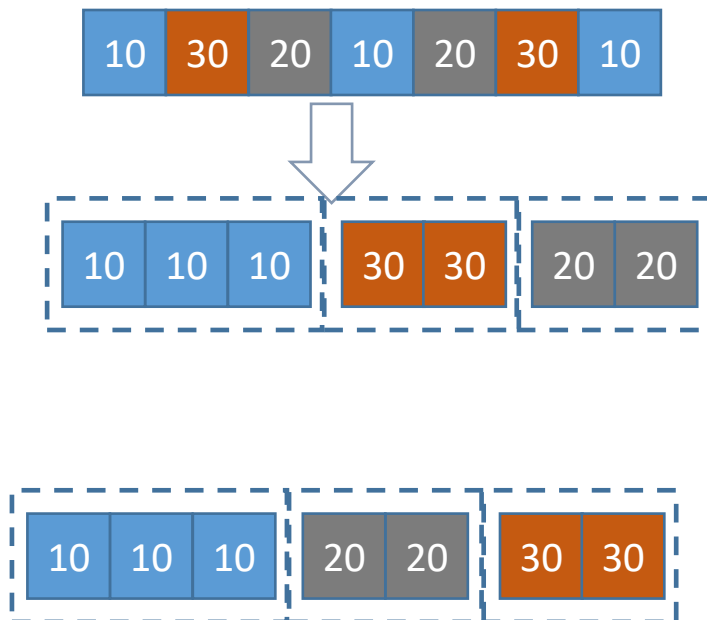
# MapReduce faux quiz (5 min, any 2):



- 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 3<sup>rd</sup> 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  $\rightarrow$  key

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



```
foreach(T elem in PF(ints))  
{  
    key    = KeyLambda(elem);  
  
    group = GetGroup(key)   
  
    group.Add(elem)   
}
```

**Note: sorting is VERY similar**

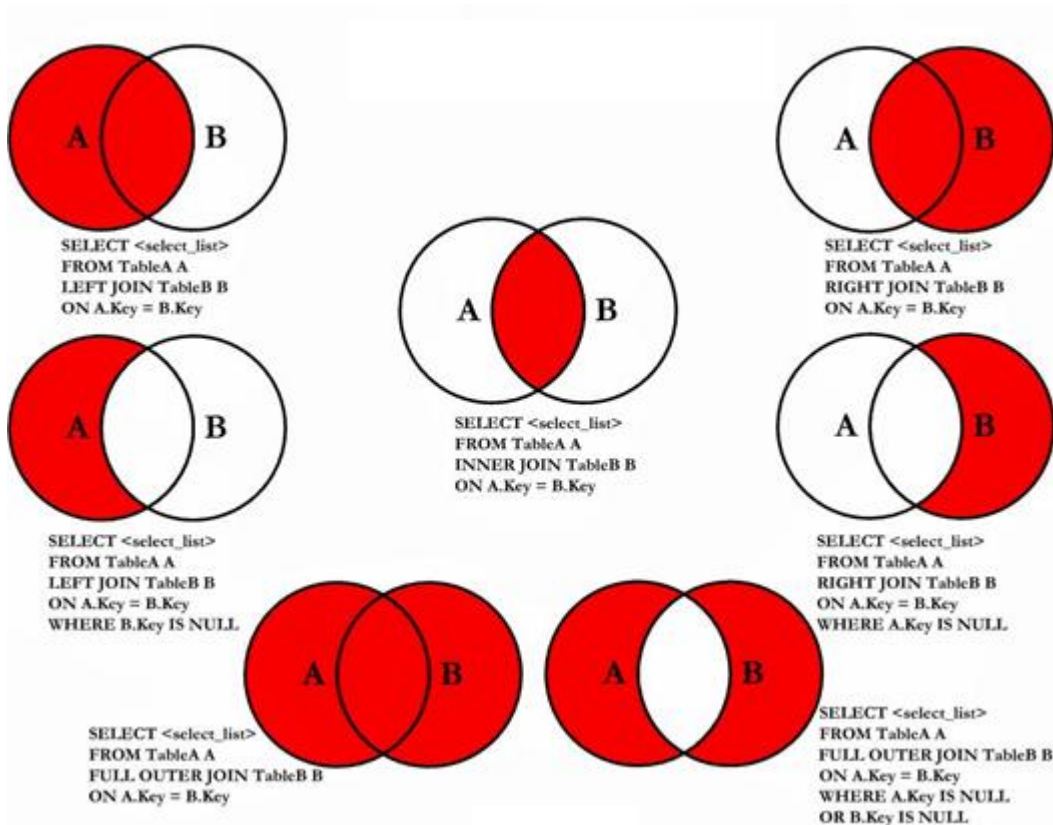
# GroupBy example

```
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);  
}
```



- 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

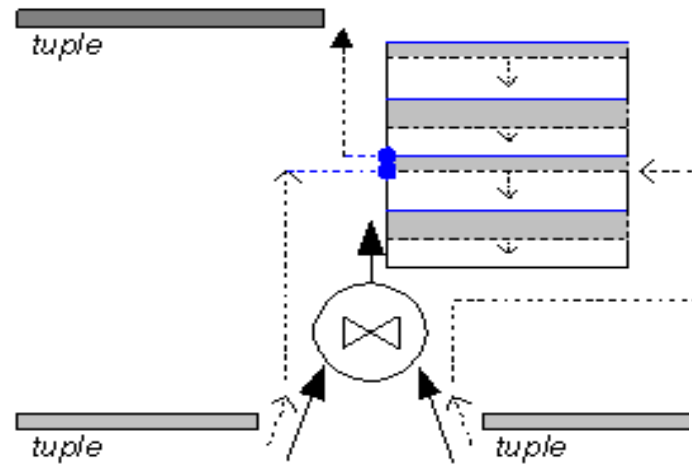
RESULT

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

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

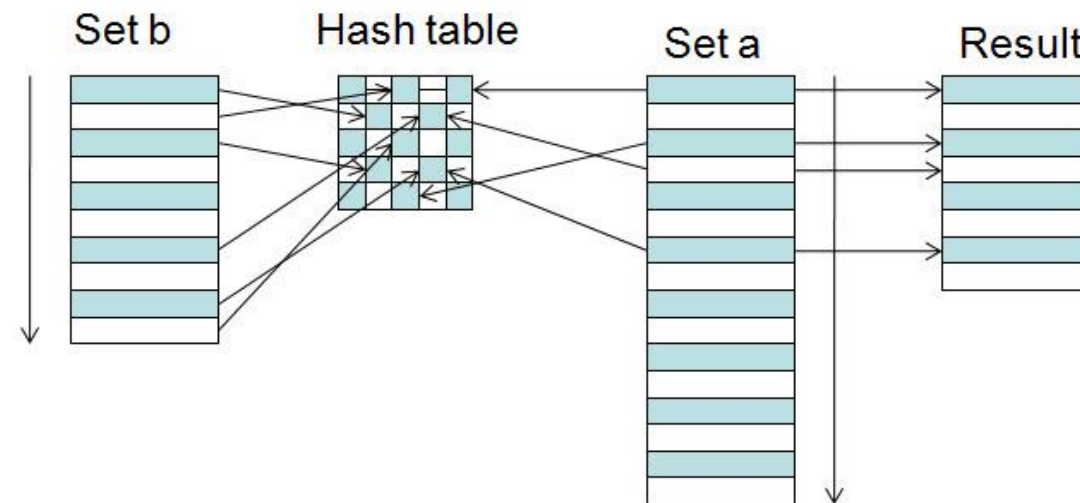
\*customerdemo database

# 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

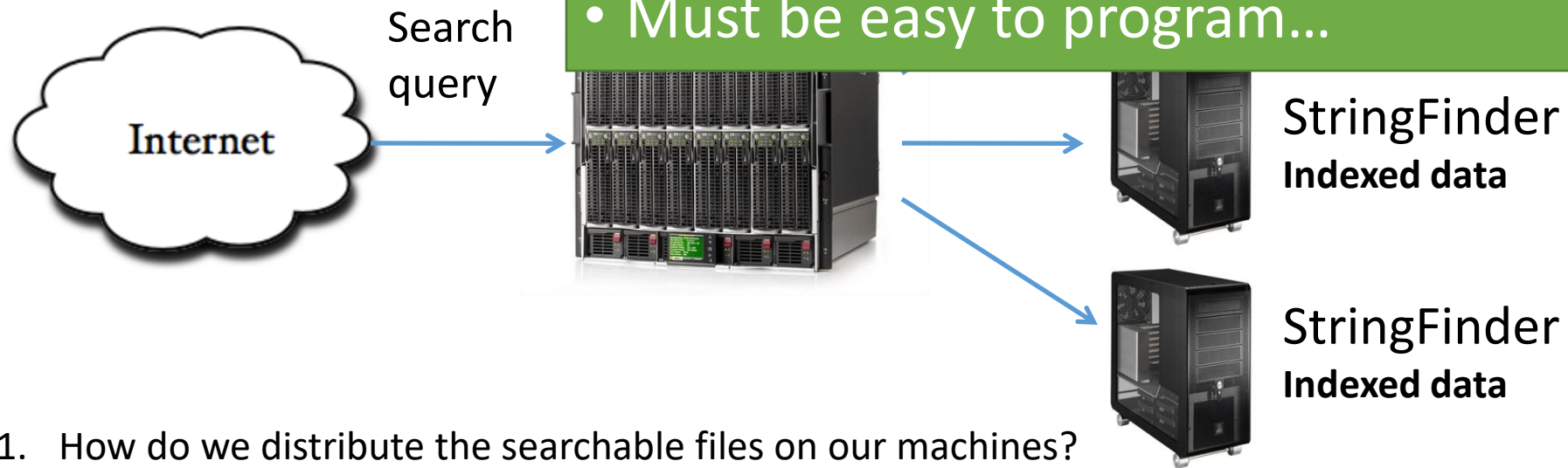
```
public class StringFinder {  
    int main(...) {  
        foreach(File f in getInputFiles()) {  
            if(f.contains(searchTerm))  
                results.add(f.getFileName());  
        }  
    }  
    System.out.println("Files:" + results.toString());  
}
```

# Infrastructure is hard

*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

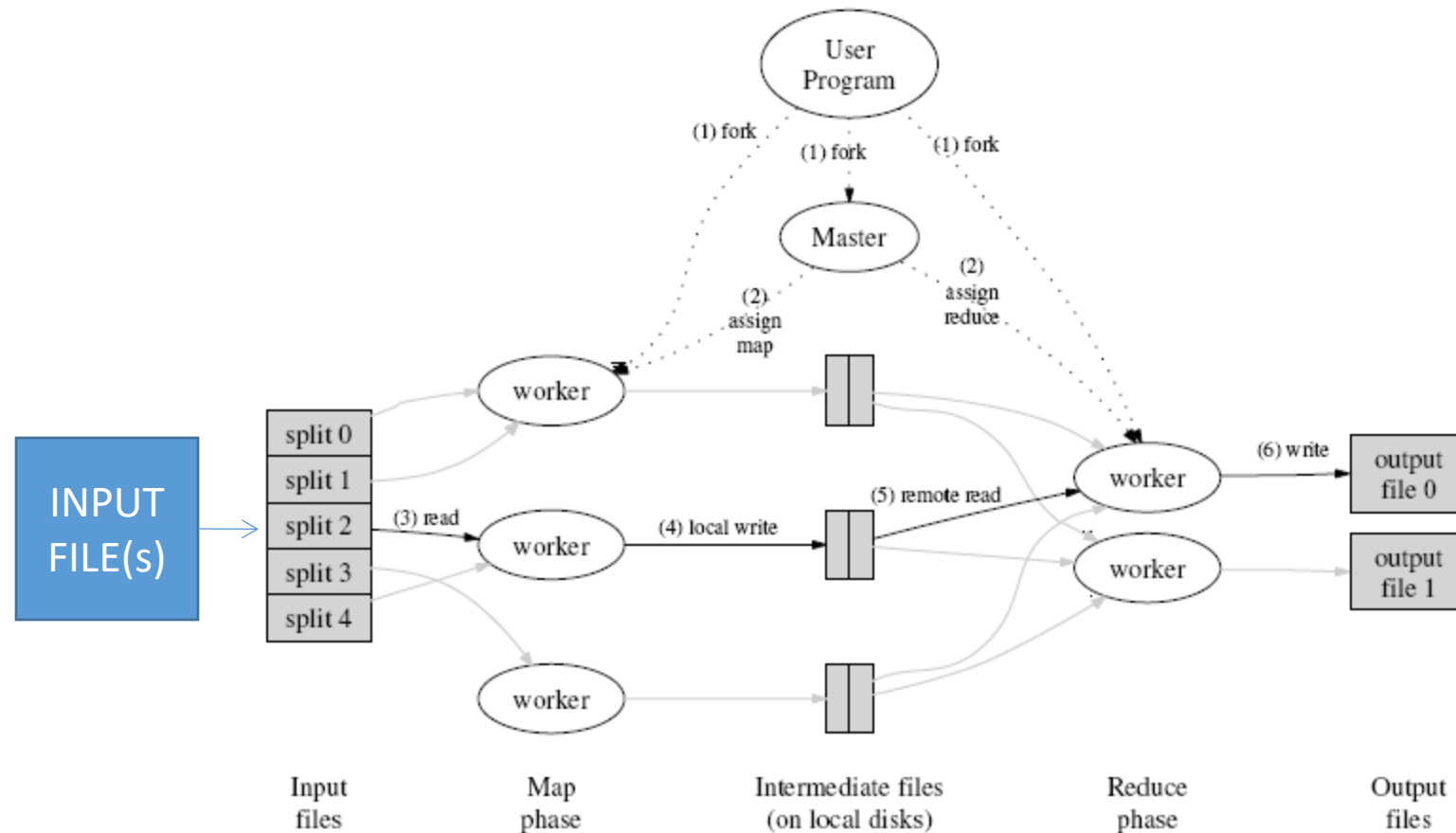
```
public void reduce() {  
    int sum = 0;  
    while(values.hasNext()) {  
        sum += values.next().get();  
    }  
    output.collect(key, sum);  
}
```

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?

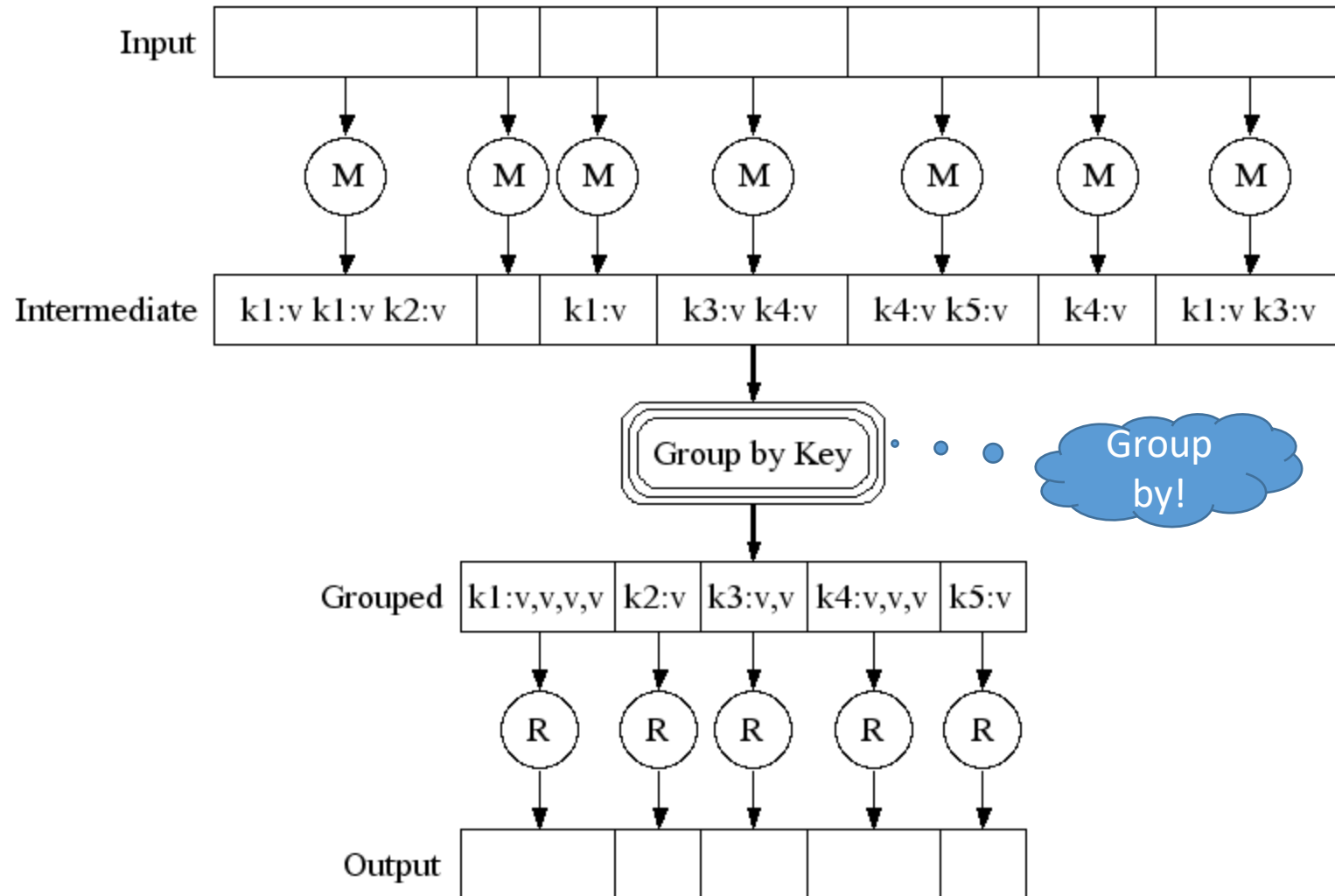




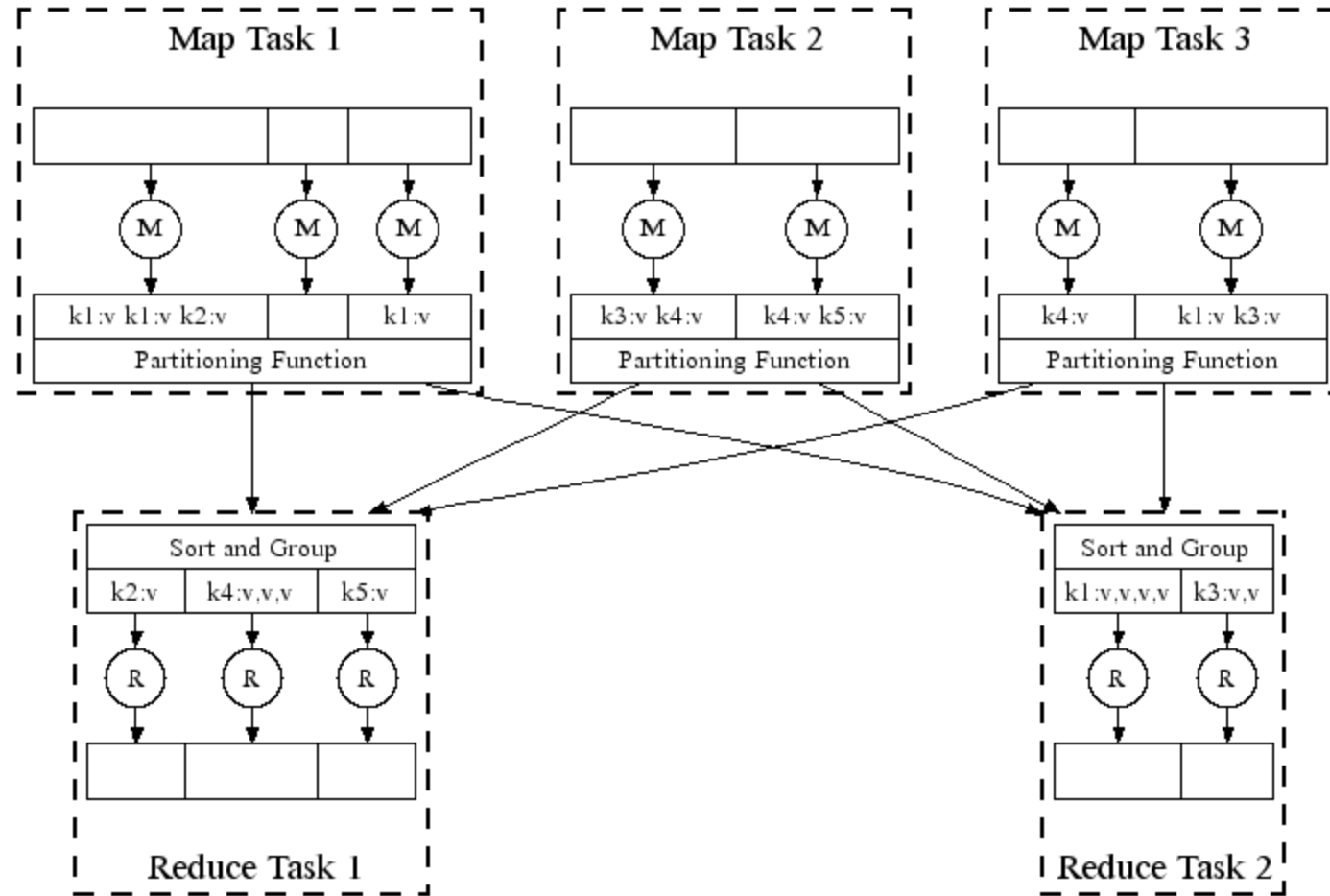
# Implementation

- 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



# Parallel Execution



# 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

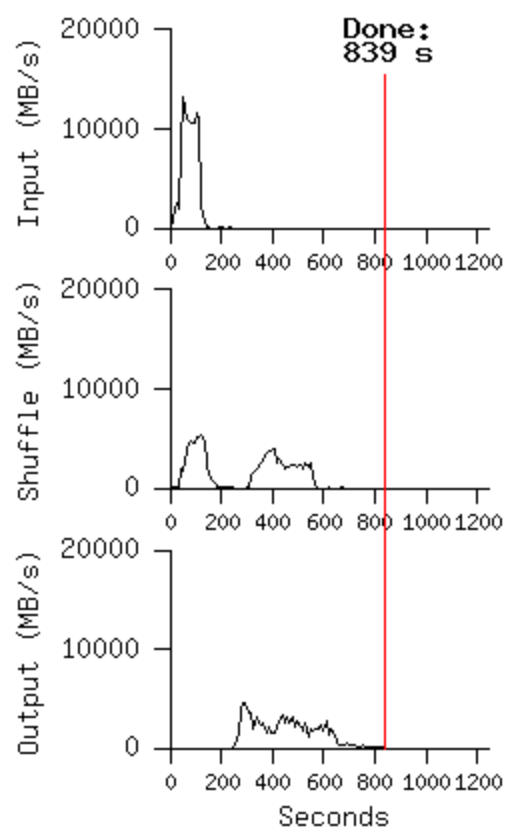
- What failures to handle?
  - How to detect failures?
  - How to respond?
    - For workers?
    - For master?
  - How to know tasks complete?
- Worker failures:
    - Detect via heartbeat
    - Re-execute completed, in-progress map
    - *Re-execute in-progress reducers (why?)*
  - Master failures: re-execute all!
  - Task completion committed through master

# Redundant Execution (straggler mitigation)

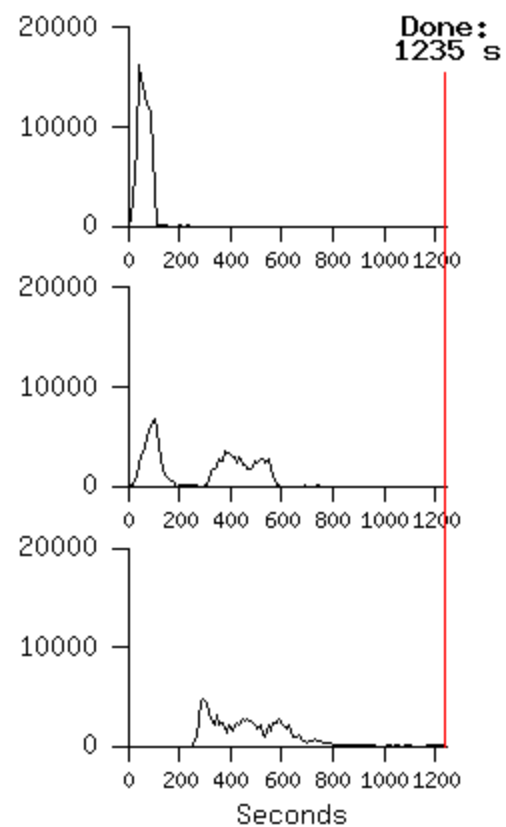
- Slow worker can throttle performance: why?
- What makes a worker slow?
- Solution:

# Redundancy performance

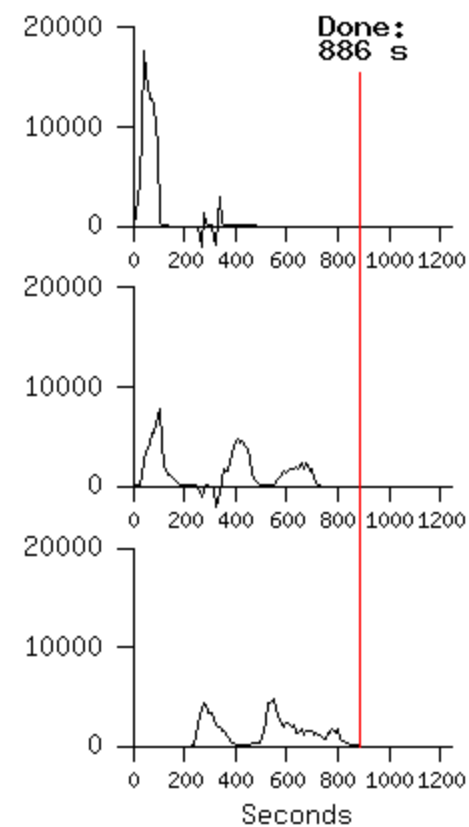
Normal



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 3<sup>rd</sup> 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?

# Engineers vote with their feet

## MapReduce Usage Statistics Over Time

	Aug, '04	Mar, '06	Sep, '07	May, '10
Number of jobs	29K	171K	2,217K	4,474K
Average completion time (secs)	634	874	395	748
Machine years used	217	2,002	11,081	39,121
Input data read (TB)	3,288	52,254	403,152	946,460
Intermediate data (TB)	758	6,743	34,774	132,960
Output data written (TB)	193	2,970	14,018	45,720
Average worker machines	157	268	394	368

# 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?

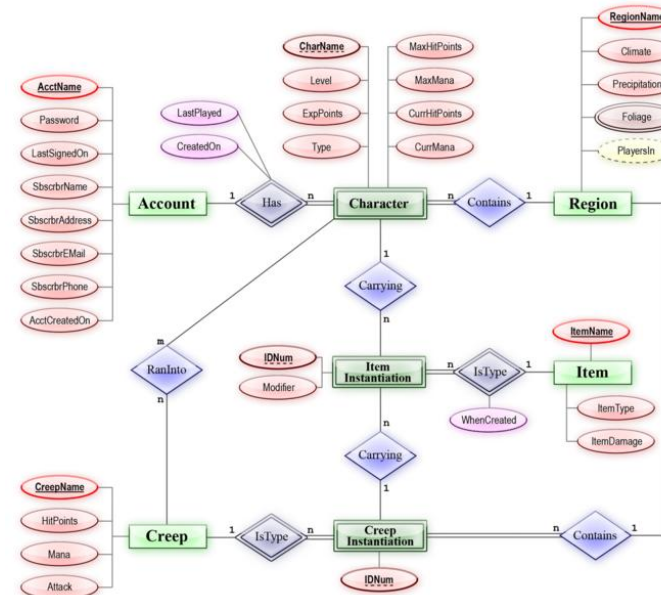
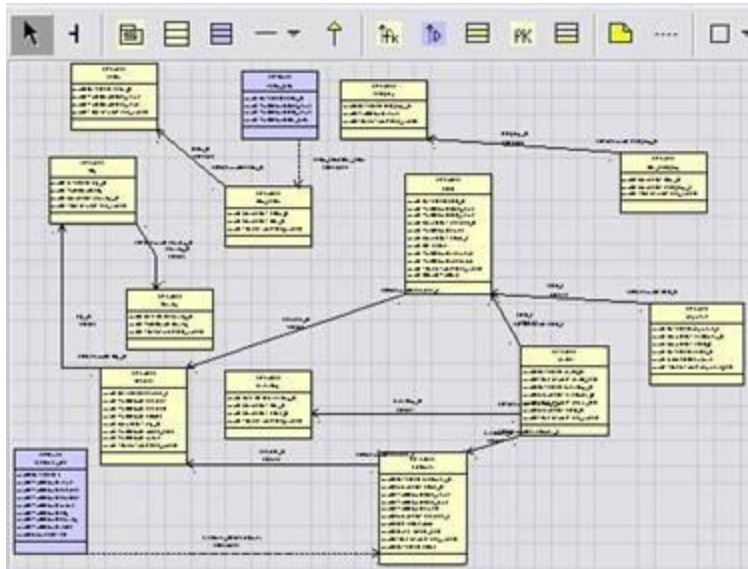
- Map == group-by
- Reduce == aggregate

```
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  $\rightarrow$   $M=500$  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

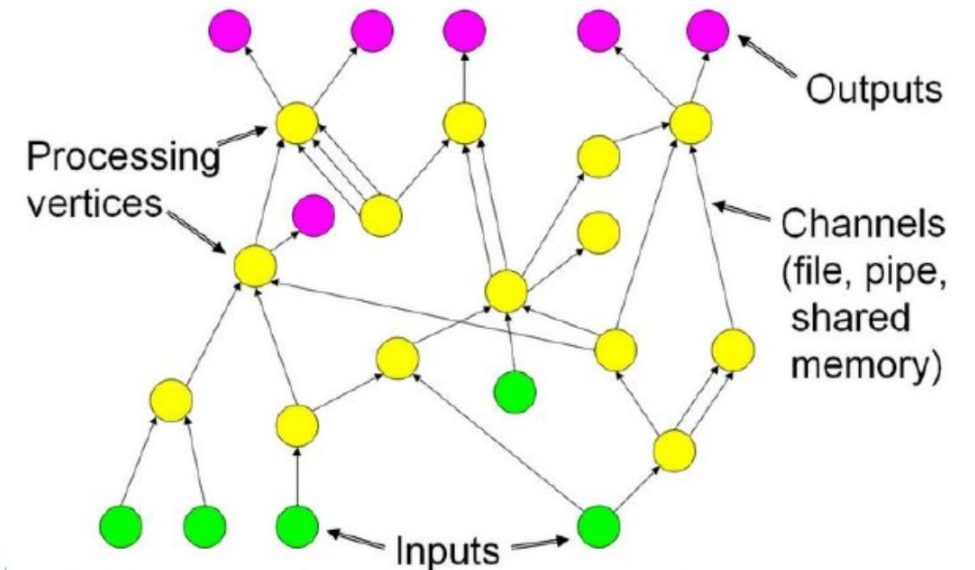
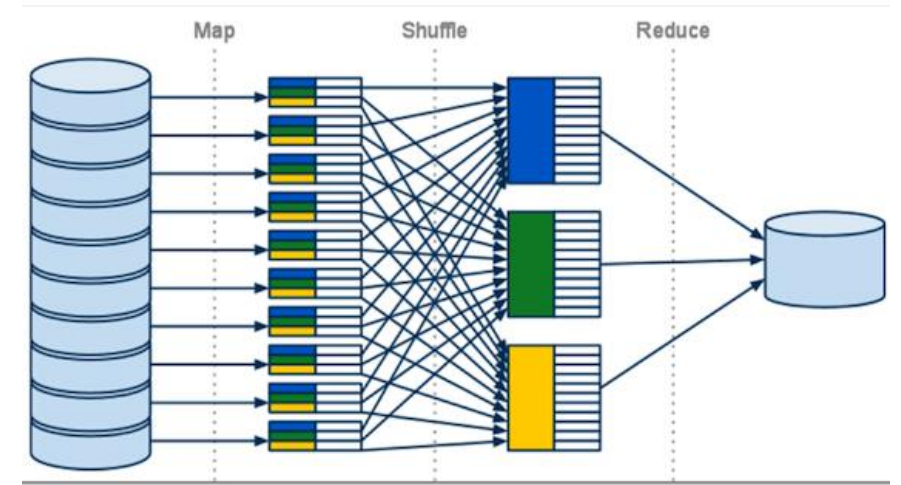
- Report writers
- Business intelligence tools
- Data-mining tools
- Replication tools
- Design tools (UML, embarcadero)

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



# Know your domain

- MapReduce excels at handling semi-structured data
  - ETL tasks, Extract, Transform, Load
  - Computations that do not require transactions
- Databases excel at complex queries and data sanitation
  - Query planners are amazing (requires talented admins)
  - Keeping data clean is helpful (yay Schemas)
  - Transactions are powerful
- Systems research often progresses finding weaker semantics that support important workloads