MapReduce Algorithms

Quick Review of MapReduce
The All-Pairs Problem
The Theory of MapReduce Algorithms
Some-Pairs Problems

Cloud and Big Data Summer School, Stockholm, Aug., 2015
Jeffrey D. Ullman

What Does MapReduce Give You?

1. Easy parallel programming.
2. Invisible management of hardware and software failures.
3. Easy management of very-large-scale data.

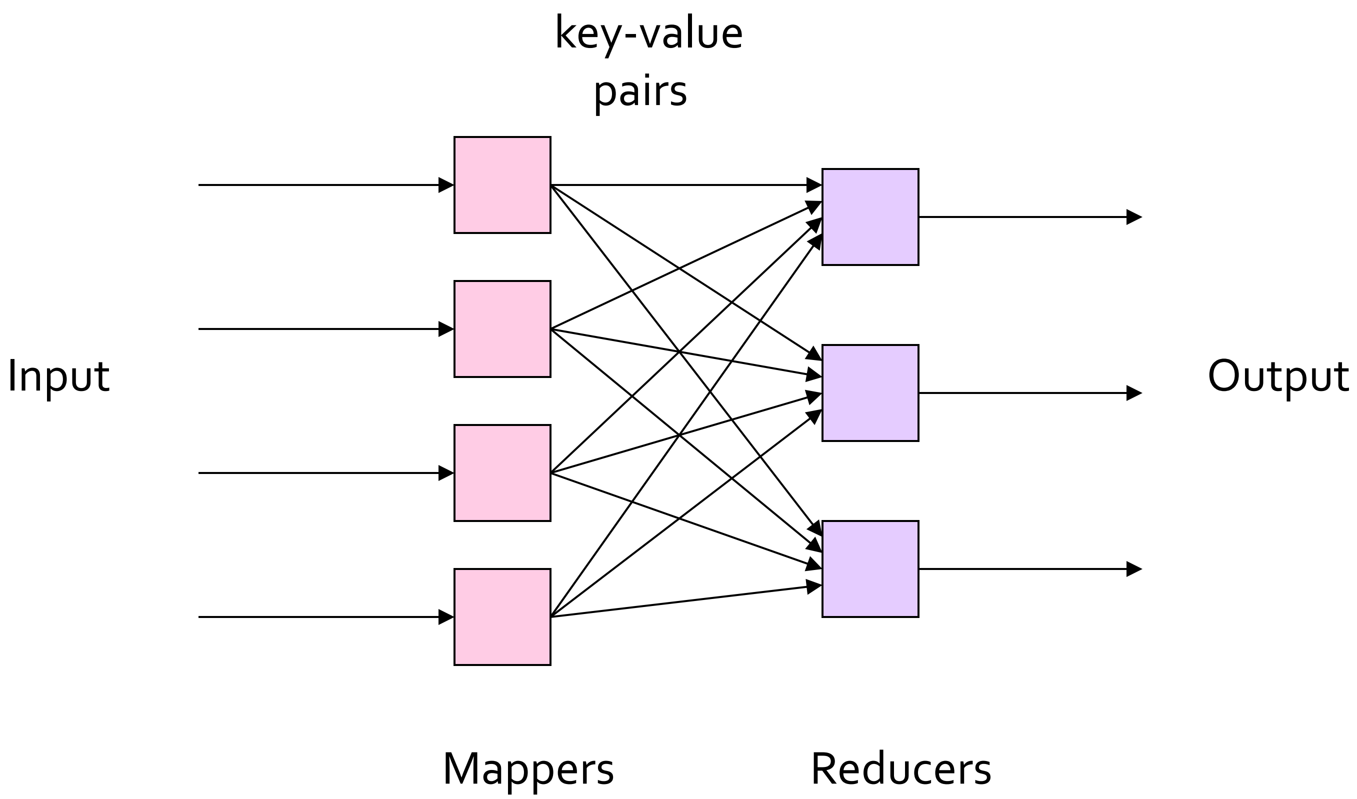
MapReduce in a Nutshell

- A MapReduce job starts with a collection of inputs of a single type.
- Apply a user-written *Map function* to each input, in parallel.
  - *Mapper* = application of the Map function to a single input.
    - Usually many mappers are grouped into a *Map Task*.
- The output of the Map function is a set of 0, 1, or more *key-value pairs*.
- The system sorts all the key-value pairs by key, forming key-(list of values) pairs.

In a Nutshell – (2)

- Another user-written function, the *Reduce function*, is applied to each key-(list of values).
  - Application of the Reduce function to one key and its list of values is a *reducer*.
    - Often, many reducers are grouped into a *Reduce Task*.
- Each reducer produces some output, and the output of the entire job is the union of what is produced by each reducer.

MapReduce Pattern



Example: Word Count

- We have a large file of documents, which are sequences of words.
- Count the number of times each distinct word appears in the file.

Word Count Using MapReduce

```
map(key, value):
// key: document ID; value: text of document
    FOR (each word w in value)
        emit(w, 1);

reduce(key, value-list):
// key: a word; value-list: a list of integers
        result = 0;
        FOR (each integer v on value-list)
                result += v;
        emit(result);
```

Expect to be all 1's, but "combiners" allow local summing of integers with the same key before passing to reducers.

Coping With Failures

- MapReduce is designed to deal with compute nodes failing to execute a Map task or Reduce task.
- Re-execute failed tasks, not whole jobs.
- Key point: MapReduce tasks have the *blocking property*: no output is used until task is complete.
- Thus, we can restart a Map task that failed without fear that a Reduce task has already used some output of the failed Map task.

Cost of a MapReduce Algorithm

1. Execution time of the mappers and reducers.
2. Communication cost of transmitting the output of the mappers to the location of the proper reducer.
   - Usually, many compute nodes handle both sorts of tasks in parallel, so there is little chance that the source and destination of a key-value pair are the same.
   - Often, communication cost dominates.

The All-Pairs Problem

Motivation: Drug Interactions
A Failed Attempt
Lowering the Communication

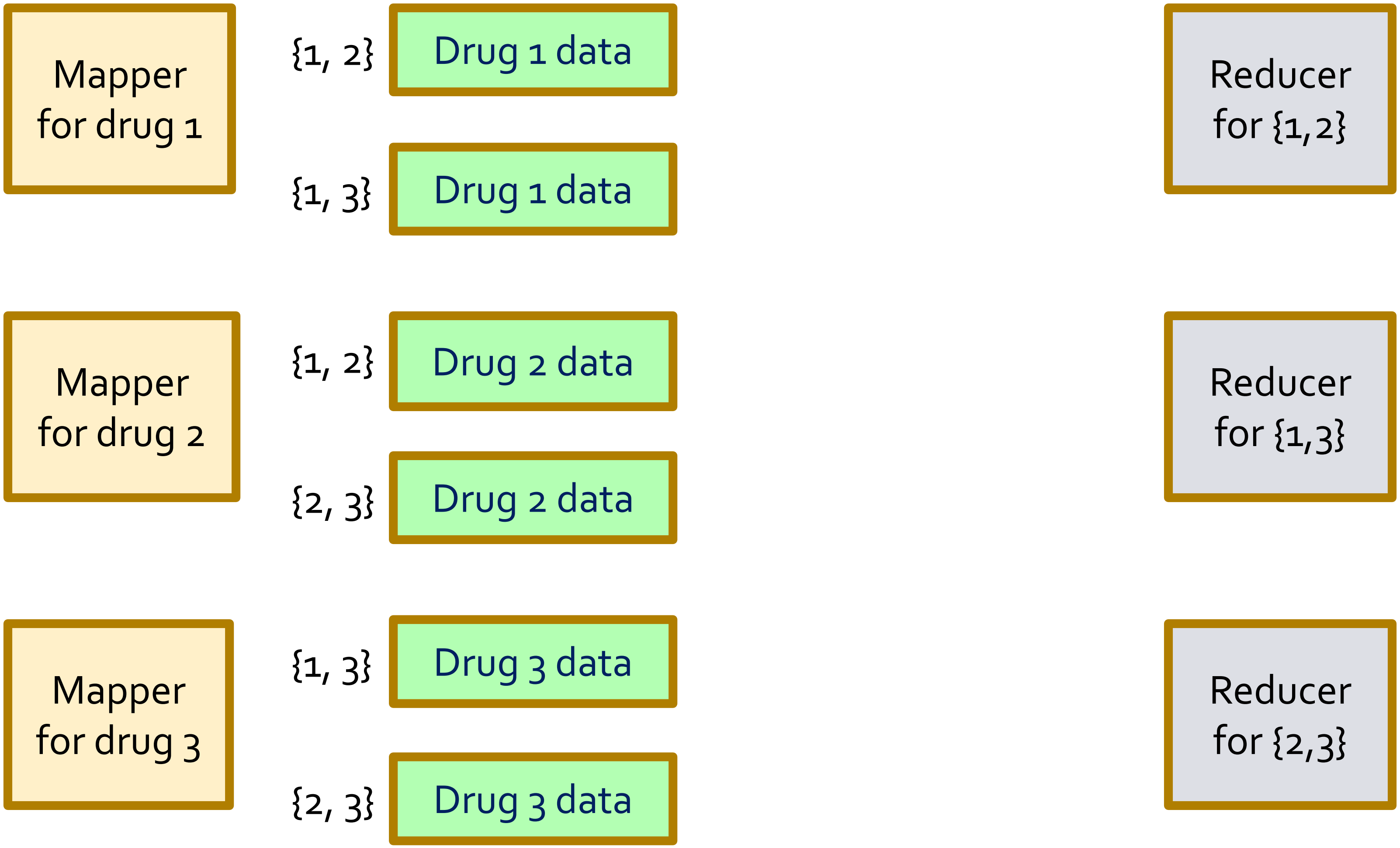
The Drug-Interaction Problem

- A real story from Stanford's CS341 data-mining project class.
- Data consisted of records for 3000 drugs.
  - List of patients taking, dates, diagnoses.
  - About 1M of data per drug.
- Problem was to find drug interactions.
  - Example: two drugs that when taken together increase the risk of heart attack.
- Must examine each pair of drugs and compare their data.

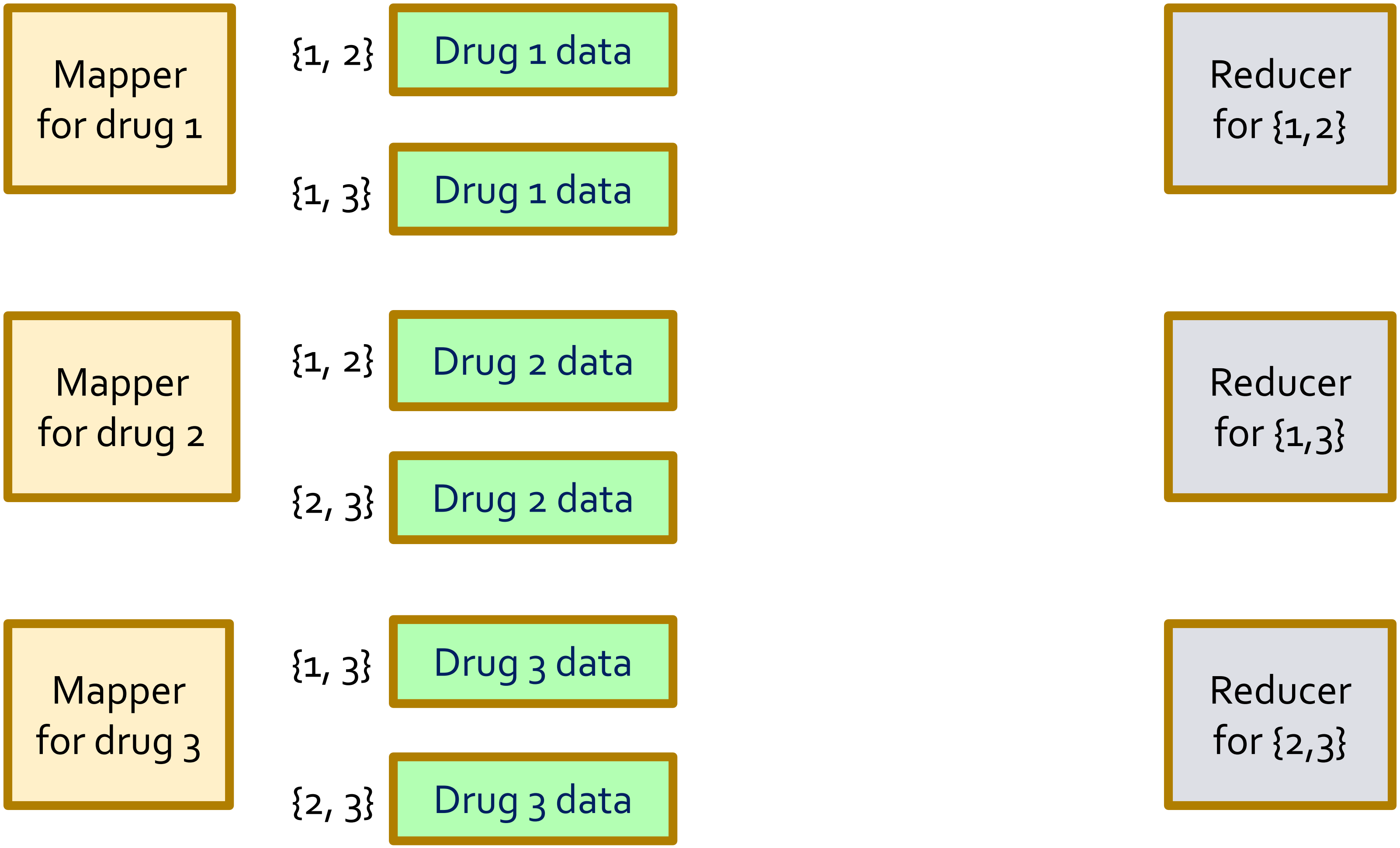
Initial Map-Reduce Algorithm

- The first attempt used the following plan:
  - Key = set of two drugs $\{i, j\}$.
  - Value = the record for one of these drugs.
- Given drug i and its record R_i, the mapper generates all key-value pairs $(\{i, j\}, R_i)$, where j is any other drug besides i.
- Each reducer receives its key and a list of the two records for that pair: $(\{i, j\}, [R_i, R_j])$.

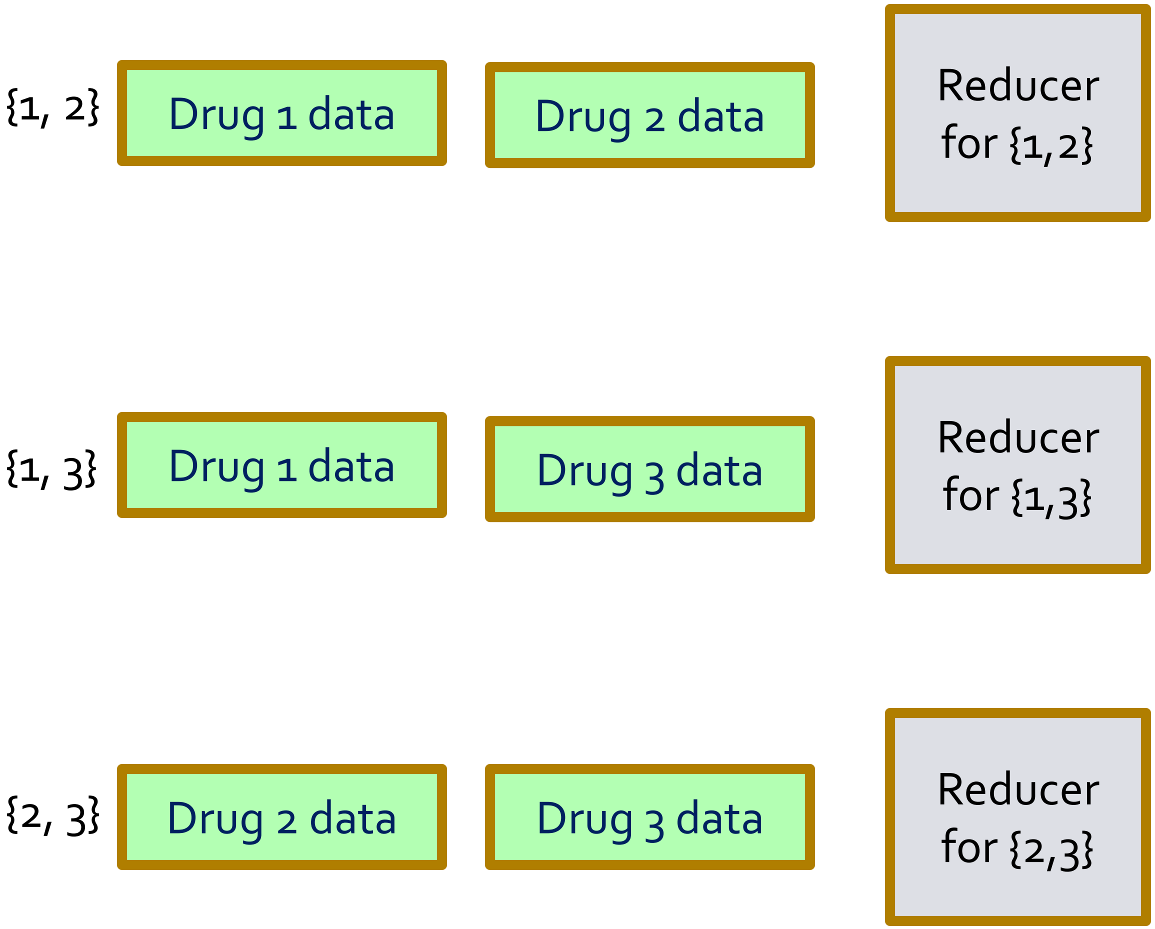
Example: Three Drugs



Example: Three Drugs



Example: Three Drugs



What Went Wrong?

- 3000 drugs
- times 2999 key-value pairs per drug
- times 1,000,000 bytes per key-value pair
- = 9 terabytes communicated over a 1Gb Ethernet
- = 90,000 seconds of network use.

The Improved Algorithm

- The team grouped the drugs into 30 groups of 100 drugs each.
  - Say G_1 = drugs 1-100, G_2 = drugs 101-200,..., G_{30} = drugs 2901-3000.
  - Let g(i) = the number of the group into which drug *i* goes.

The Map Function

- A key is a set of two group numbers.
- The mapper for drug *i* produces 29 key-value pairs.
  - Each key is the set containing g(i) and one of the other group numbers.
  - The value is a pair consisting of the drug number *i* and the megabyte-long record for drug *i*.

The Reduce Function

- The reducer for pair of groups {*m*, *n*} gets that key and a list of 200 drug records – the drugs belonging to groups *m* and *n*.
- Its job is to compare each record from group *m* with each record from group *n*.
  - Special case: also compare records in group *n* with each other, if $m = n+1$ or if $n = 30$ and $m = 1$.
- Notice each pair of records is compared at exactly one reducer, so the total computation is not increased.

The New Communication Cost

- The big difference is in the communication requirement.
- Now, each of 3000 drugs' 1MB records is replicated 29 times.
  - Communication cost = 87GB, vs. 9TB.

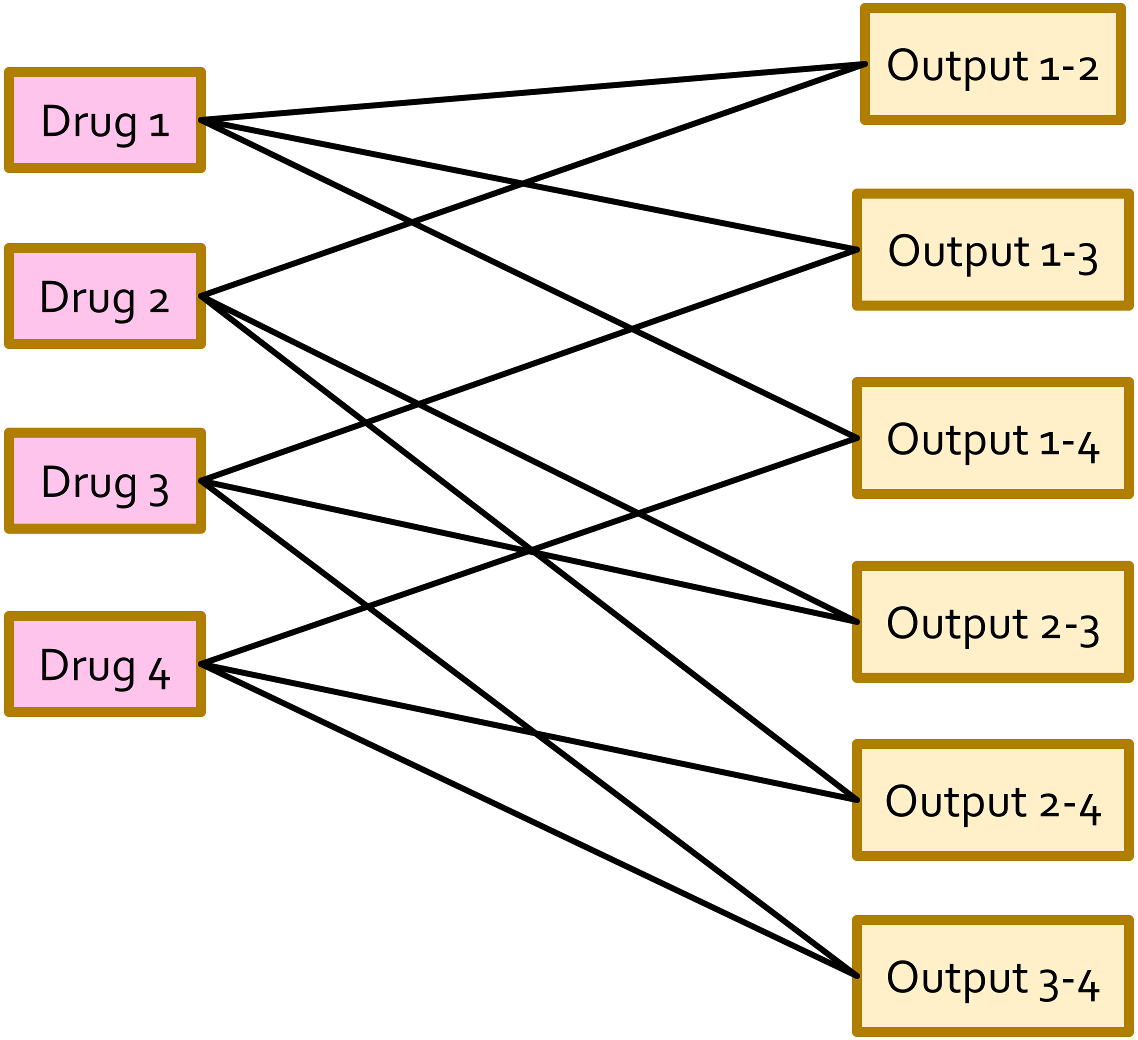
Outline of the Theory

- Work due to: Foto Afrati, Anish Das Sarma, Semih Salihoglu, U
- Reducer Size
- Replication Rate
- Mapping Schemas

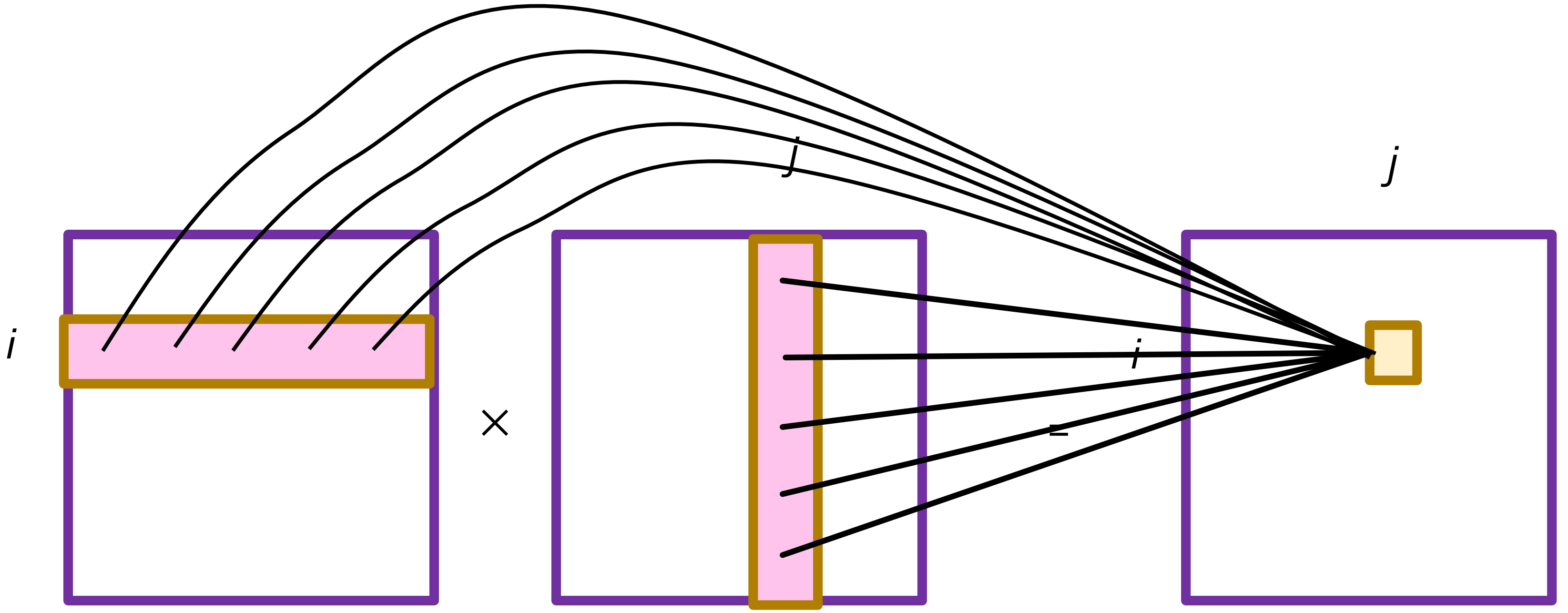
A Model for Map-Reduce Problems

1. A set of *inputs*.
   - Example: the drug records.
2. A set of *outputs*.
   - Example: one output for each pair of drugs, telling whether a statistically significant interaction was detected.
3. A many-many relationship between each output and the inputs needed to compute it.
   - Example: The output for the pair of drugs {*i*, *j*} is related to inputs *i* and *j*.

Example: Drug Inputs/Outputs



Example: Matrix Multiplication



Reducer Size

- *Reducer size*, denoted q, is the maximum number of inputs that a given reducer can have.
  - I.e., the length of the value list.
- Limit might be based on how many inputs can be handled in main memory.
- Or: make q low to force lots of parallelism.

Replication Rate

- The average number of key-value pairs created by each mapper is the *replication rate*.
  - Denoted r.
- Represents the communication cost per input.

Example: Drug Interaction

- Suppose we use g groups and d drugs.
- A reducer needs two groups, so $q = 2d/g$.
- Each of the d inputs is sent to g-1 reducers, or approximately $r = g$.
- Replace g by r in $q = 2d/g$ to get $r = 2d/q$.

Tradeoff!
The bigger the reducers,
the less communication.

Upper and Lower Bounds on r

- What we did gives an upper bound on r as a function of q.
- A solid investigation of MapReduce algorithms for a problem includes lower bounds.
  - Proofs that you cannot have lower r for a given q.

Proofs Need Mapping Schemas

- A *mapping schema* for a problem and a reducer size q is an assignment of inputs to sets of reducers, with two conditions:
  1. No reducer is assigned more than q inputs.
  2. For every output, there is some reducer that receives all of the inputs associated with that output.
     - Say the reducer *covers* the output.
     - If some output is not covered, we can't compute that output.

Mapping Schemas – (2)

- Every MapReduce algorithm has a mapping schema.
- The requirement that there be a mapping schema is what distinguishes MapReduce algorithms from general parallel algorithms.

Example: Drug Interactions

- d drugs, reducer size q.
- Each drug has to meet each of the d-1 other drugs at some reducer.
- If a drug is sent to a reducer, then at most q-1 other drugs are there.
- Thus, each drug is sent to at least $\lceil(d-1)/(q-1)\rceil$ reducers, and $r \geq \lceil(d-1)/(q-1)\rceil$.
  - Or approximately $r \geq d/q$.
- Half the r from the algorithm we described.
- Better algorithm gives $r = d/q + 1$, so lower bound is actually tight.

Some-Pairs Problems

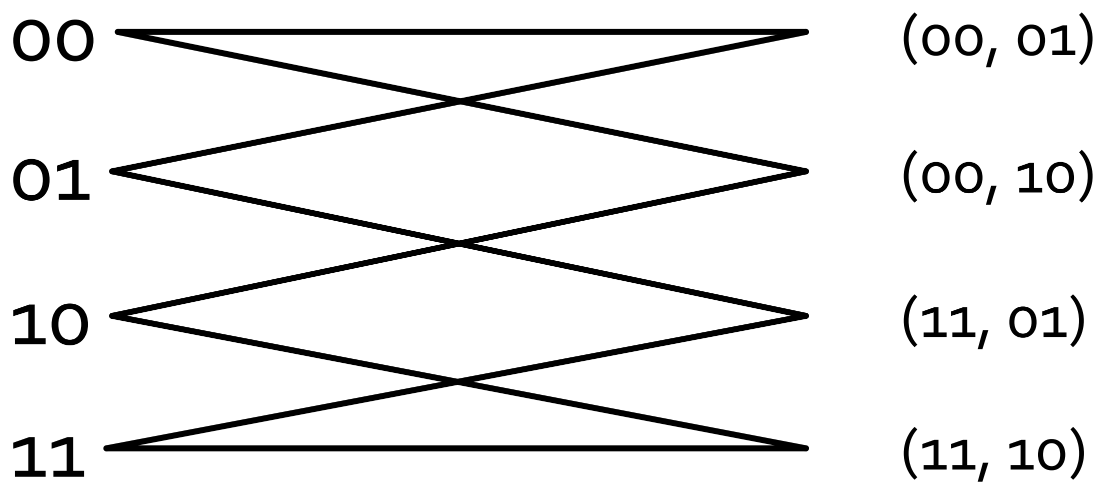
Work due to: Jonathan Ullman, U
Example: Hamming Distance
Two Obvious Approaches
Lower Bound

Some-Pairs Problems

- Outputs = a subset of the pairs of inputs.
- Example: HD1.
  - Inputs = bit strings of length b.
  - Outputs = all pairs of inputs that are at Hamming distance 1.
    - *Hamming distance* = number of positions in which strings differ.
- Known upper and lower bound: $r = b/\log_2 q$.

Example: HD1

Inputs



Outputs

Note pairs (00, 11) and (01, 10) are NOT outputs.

General-Purpose Algorithms

- Some particular some-pairs problems have really good solutions.
  - Example: HD1
- But we're looking for a single algorithm that solves any some-pairs problem and takes advantage of the fact that not all pairs of inputs are outputs.

Obvious Algorithm #1

- Let there be n inputs and m outputs.
- Assume all pairs are outputs, and use the all-pairs solution.
- Gives us $r = n/q$, independent of m.

Obvious Algorithm #2

- For each of the m outputs, create a reducer for only the two inputs associated with that output.
- Requires only $q = 2$.
- Gives us replication rate $r = 2m/n$.

Optimality of the Obvious

- Theorem: For any n, m, and q, there is a some-pairs problem whose replication rate is "almost" as large as min(m/n, n/q).
- More precisely, $r \geq \min\left(\varepsilon m/n, (n/q)^{1-\varepsilon}\right)$ for any $\varepsilon > 0$.
- Note: Most common problems will have better solutions; this lower bound only limits what a *general-purpose* algorithm can do.

Summary

- MapReduce is an important tool for failure-resistant parallel computation.
- The theory of algorithm design for MapReduce is in its infancy.
  - Involves the tradeoff between how much work to assign to a reducer and the amount of communication needed.
  - Many open questions remain, e.g., “Hamming-distance 2.”