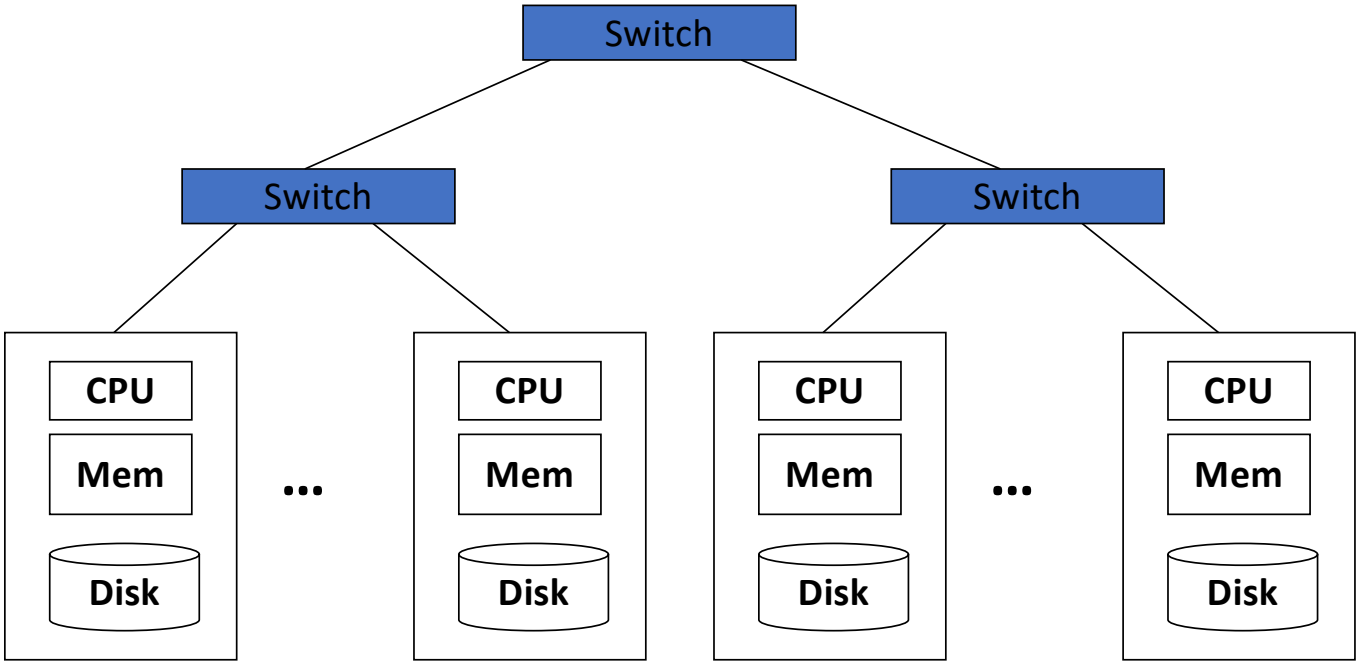
Computational Tools for Data Science

Week 3 Lecture: MapReduce

The New Software Stack

- Modern computing often makes use of massive data/files
- Just reading such files from disk takes a prohibitively long time
- Must make use of parallelism
- Cluster Computing Model

Cluster Computing Architecture



Each rack contains 16-64 nodes

Challenge: Machines fail

- One server may stay up 3 years (1,000 days)
- If you have 1,000 servers, expect to lose 1/day
- People estimated Google had approximately 1M machines in 2011
  - 1,000 machines fail every day!

Solution: Distributed file system.

Distributed File System

- **Provides global namespace, redundancy, and availability**

- **Examples:**
  - Google File System (GFS)
  - Hadoop File System (HDFS)

- **Typical usage pattern:**
  - Huge files
  - Data rarely updated in place
  - Reads and appends are common

Distributed File System

- **Chunk servers**
  - Files split into “chunks”
  - Typically 16-64 MB
  - Each chunk replicated 2x or 3x, preferably on different racks
- **Master/Name node**
  - Stores metadata about where files are stored
  - Might be replicated
- **Client library for file access**
  - Talks to master to find chunks
  - Connects directly to chunk servers to access data

Other Challenges

- **Network Bottlenecks**
  - Network bandwidth e.g. 40 Gbps
  - Moving 400 TB takes approximately 1 day

Idea: move computation to the data

- **Distributed programming is hard!**
  - Need a simple model that hides most of the complexity

Solution to both: **MapReduce!**

MapReduce

- **Dataflow**
  - **Split:** Partition data into chunks and distribute to different machines.
  - **Map:** Map data items to list of <key, value> pairs.
  - **Shuffle:** Group data with the same key and send to the same machine.
  - **Reduce:** Takes list of values with the same key <key, [$value_1$, ..., $value_k$]> and outputs list of new data items.

- You only write the Map and Reduce functions.

- Technically all inputs to Map tasks and outputs from Reduce tasks should have <key, value> form to allow for composition.

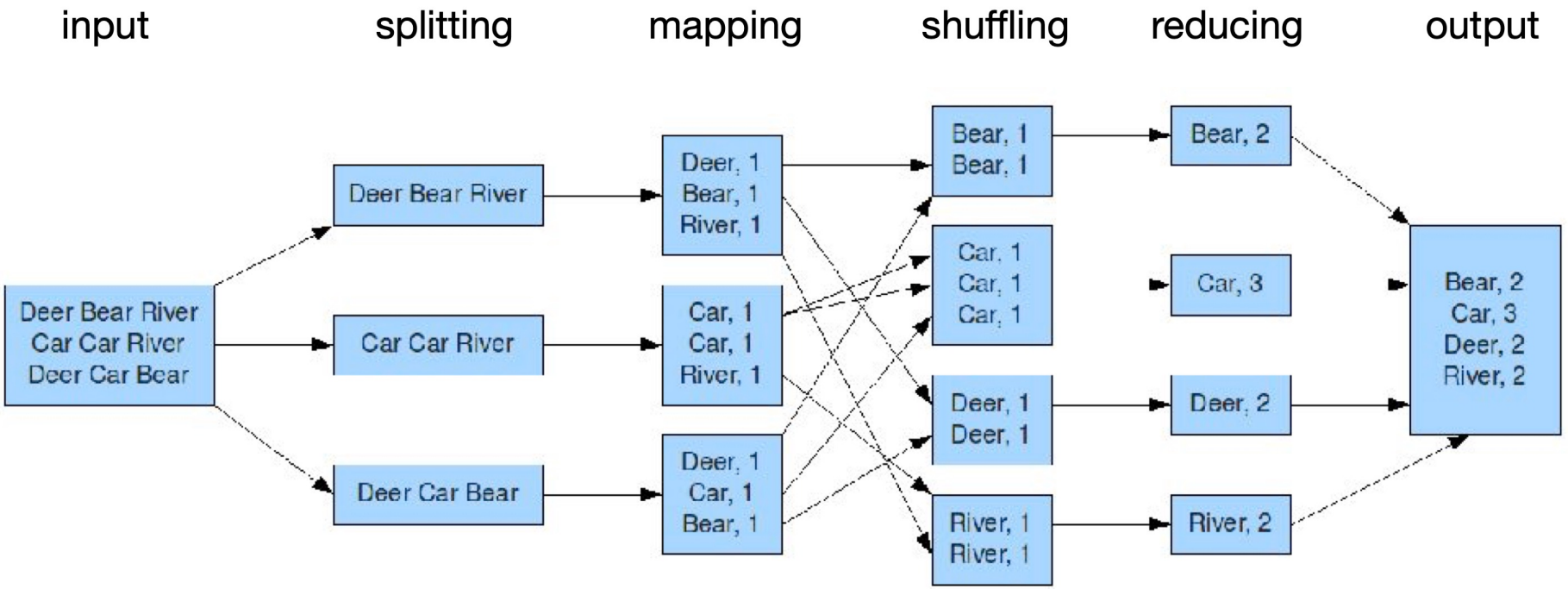
MapReduce: Word Counting

- **Input:** Document of words
- **Output:** Frequency of each word

- Document: “Deer Bear River Car Car River Deer Car Bear.”

- (Bear, 2), (Car, 3), (Deer, 2), (River, 2)

MapReduce: Word Counting



map(word) → <word, 1>

reduce(word, [1, 1, .., 1]) → <word, number of 1's>

MapReduce more formally

- **Input:** a set of key-value pairs <k, v>

- **map**$(k, v) \rightarrow$ <k_1, v_1>, <k_2, v_2>, …, <k_n, vn>
  - Takes a key-value pair and outputs a set of key-value pairs (possibly zero)
  - One map call for every input <k,v> pair

- **reduce**$(k, [v_1, \ldots, v_r]) \rightarrow$ <k, v'_1>, …, <k, v'_s>
  - All values v_i with the same key k are reduced together
  - There is one **reduce** call for each *unique* key k produced by **map**

MapReduce: Inverted Index

- **Input:** Set of documents
- **Output:** List of documents that contain each word

- Document 1: “Deer Bear River Car Car River Deer Car Bear.”
- Document 2: "Deer Antilope Stream River Stream"

- (Bear, [1]), (Car, [1]), (Deer, [1,2]), (River, [1,2]), (Antilope, [2]), (Stream, [2])

MapReduce: Inverted Index

- map(word in document i) → <word, i>

- reduce(word, [i_1, i_2, …, i_k]) → <word, [j_1, j_2, …, $j_{k'}$]>

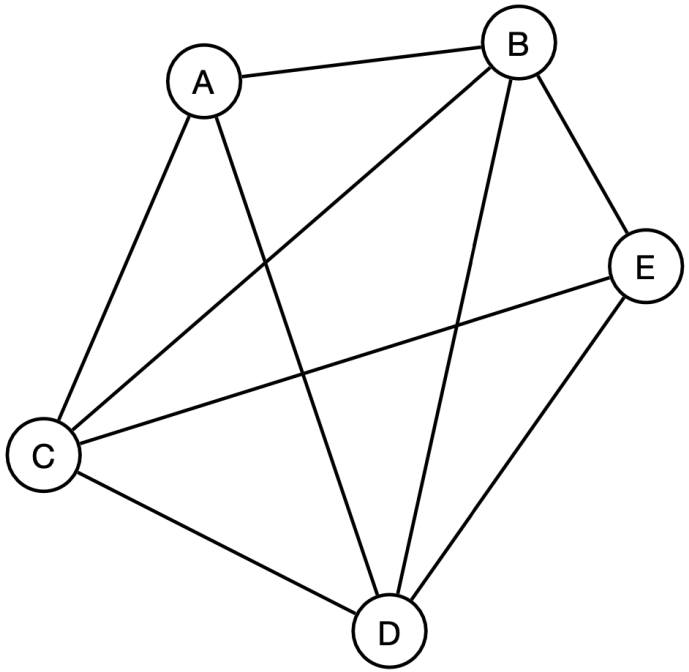
↑
Unique
(sorted)

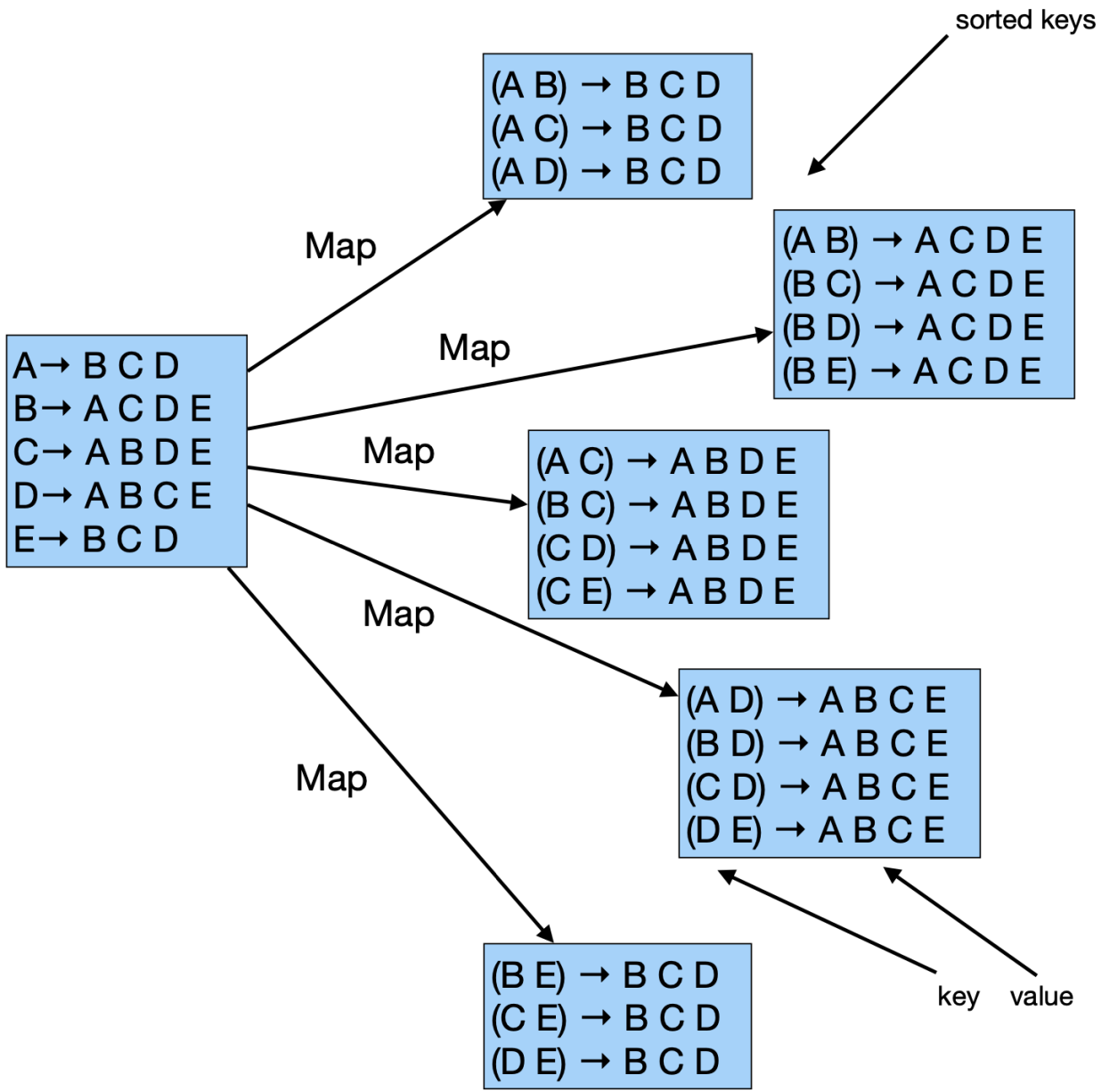
MapReduce: Common Friends

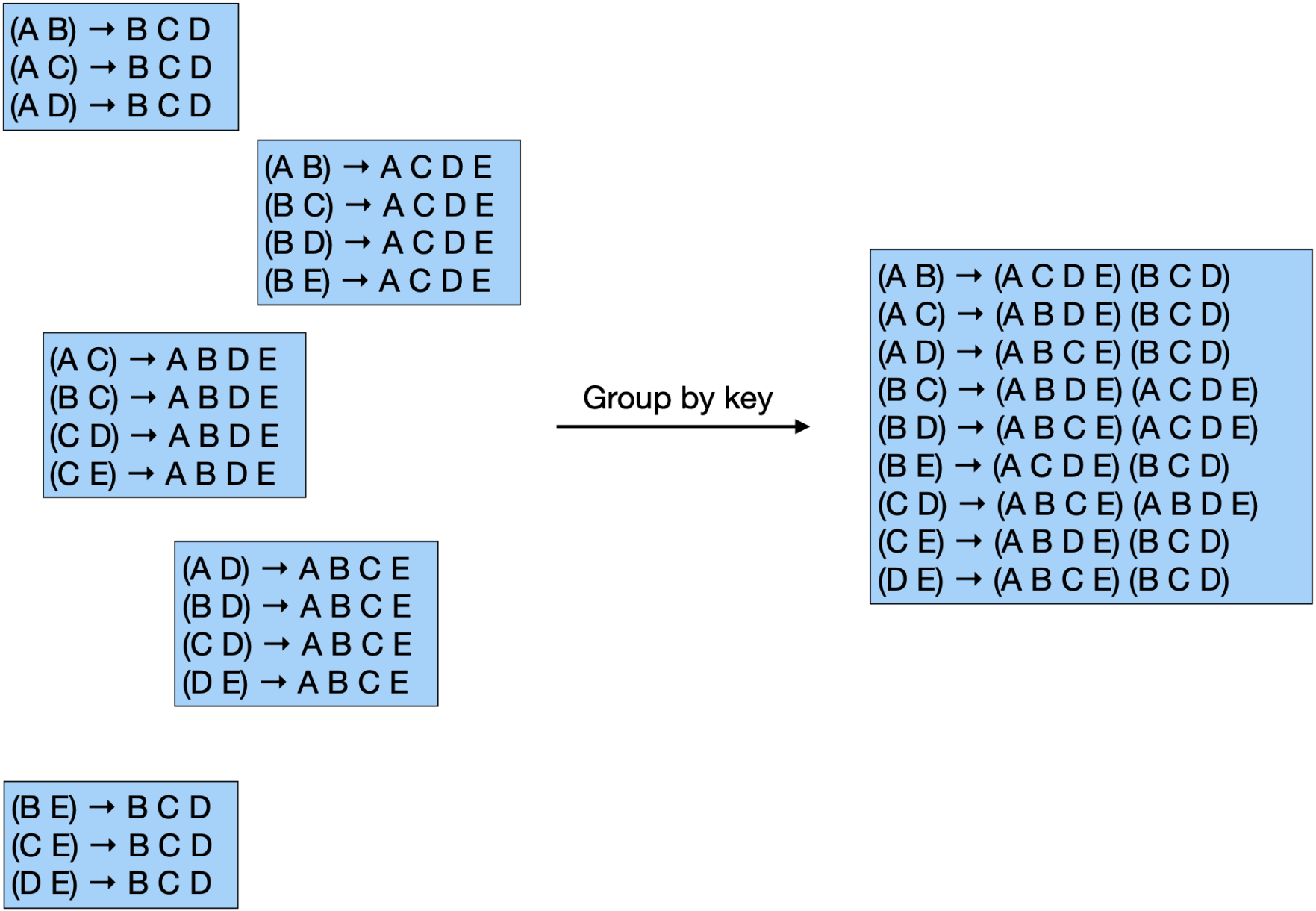
- **Input:** Friend lists
- **Output:** For pairs of friends, a list of common friends

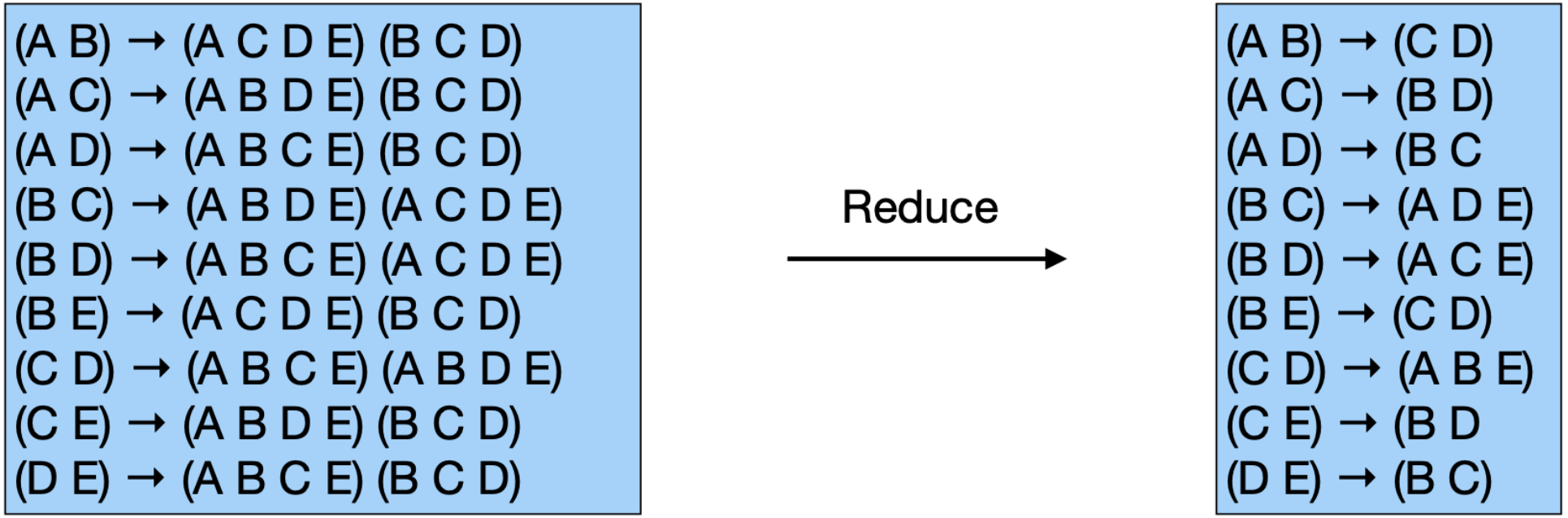
```
A→ B C D
B→ A C D E
C→ A B D E
D→ A B C E
E→ B C D
```

```
(A B) → (C D)
(A C) → (B D)
(A D) → (B C
(B C) → (A D E)
(B D) → (A C E)
(B E) → (C D)
(C D) → (A B E)
(C E) → (B D
(D E) → (B C)
```




sorted keys
(A B) → B C D
(A C) → B C D
(A D) → B C D
Map
(A B) → A C D E
(B C) → A C D E
(B D) → A C D E
(B E) → A C D E
A→ B C D
B→ A C D E
C→ A B D E
D→ A B C E
E→ B C D
Map
Map
(A C) → A B D E
(B C) → A B D E
(C D) → A B D E
(C E) → A B D E
Map
(A D) → A B C E
(B D) → A B C E
(C D) → A B C E
(D E) → A B C E
Map
(B E) → B C D
(C E) → B C D
(D E) → B C D
key
value


(A B) → B C D
(A C) → B C D
(A D) → B C D
(A B) → A C D E
(B C) → A C D E
(B D) → A C D E
(B E) → A C D E
(A C) → A B D E
(B C) → A B D E
(C D) → A B D E
(C E) → A B D E
(A D) → A B C E
(B D) → A B C E
(C D) → A B C E
(D E) → A B C E
(B E) → B C D
(C E) → B C D
(D E) → B C D
Group by key
(A B) → (A C D E) (B C D)
(A C) → (A B D E) (B C D)
(A D) → (A B C E) (B C D)
(B C) → (A B D E) (A C D E)
(B D) → (A B C E) (A C D E)
(B E) → (A C D E) (B C D)
(C D) → (A B C E) (A B D E)
(C E) → (A B D E) (B C D)
(D E) → (A B C E) (B C D)


(A B) → (A C D E) (B C D)
(A C) → (A B D E) (B C D)
(A D) → (A B C E) (B C D)
(B C) → (A B D E) (A C D E)
(B D) → (A B C E) (A C D E)
(B E) → (A C D E) (B C D)
(C D) → (A B C E) (A B D E)
(C E) → (A B D E) (B C D)
(D E) → (A B C E) (B C D)
Reduce
(A B) → (C D)
(A C) → (B D)
(A D) → (B C
(B C) → (A D E)
(B D) → (A C E)
(B E) → (C D)
(C D) → (A B E)
(C E) → (B D
(D E) → (B C)

Matrix Vector Multiplication

- **Input:** $r \times n$ matrix M, vector v of length n
  - $m_{ij} =$ (i, j) entry of M
  - $v_j = j^{\text{th}}$ entry of v

- **Output:** vector $x = Mv$ of length r with entries

$$x_i = \sum_{j=1}^{n} m_{ij} v_j$$

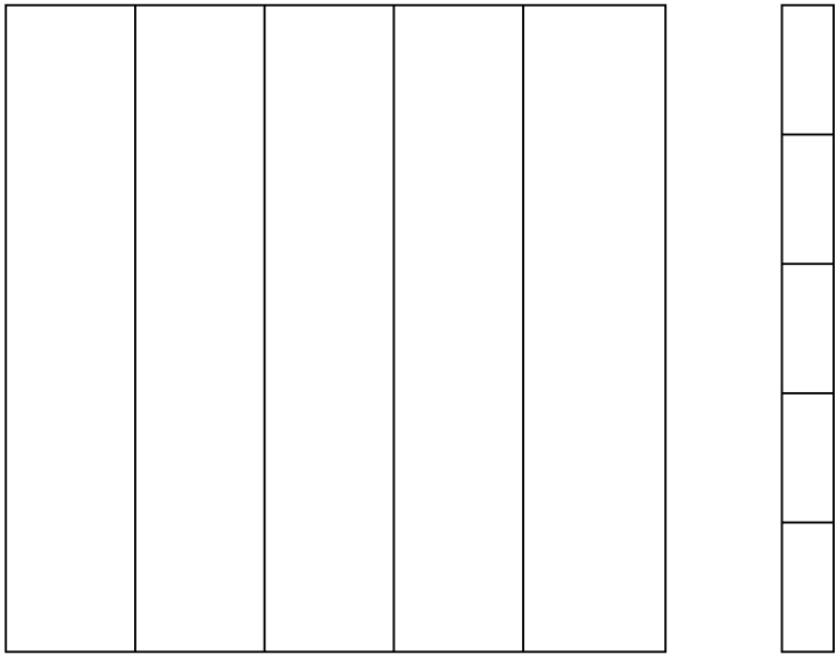
Matrix Vector Multiplication

$$x_i = \sum_{j=1}^{n} m_{ij} v_j$$

- map(m_{ij}) $\rightarrow$ <$i, m_{ij} v_j$>
  - So all terms making up x_i have the same key

- reduce: simply sum up all values associated to a key i to get <i, x_i>

What if v is too big for main memory?



Matrix M Vector v

Relations

- A **relation** is a table with column headers called **attributes**, e.g.

From	*To*
url1	url2
url1	url3
url2	url3
url2	url4
...	...

- Rows of the table are called **tuples**

- Write $R(A_1, A_2, \ldots, A_n)$ to say that the relation name is R and its attributes are $A_1, A_2, \ldots, A_n$

- Used in databases

Operations on relations

- **Selection:** Apply a condition C to each tuple in the relation and produce as output only those tuple that satisfy C. Denoted $\sigma_C(R)$.

- **Projection:** For some subset S of attributes, produce from each tuple only the components for the attributes in S. The result of this projection is denoted $\pi_S(R)$.

- **Union, Intersection, and Difference:** Used for two relations with the same schema.

Operations on relations (cont.)

- **Natural Join:** Given two relations R and S, compare each pair of tuples, one from each relation. If the tuples agree on all the attributes that are common to the two schemas, then produce a tuple that has components for each of the attributes in either schema and agrees with the two tuples on each attribute. If the tuples disagree on one or more shared attributes, then produce nothing from this pair of tuples. Denoted $R \bowtie S$.

Operations on relations (cont.)

- **Grouping and Aggregation:**
  - Given a relation R, partition its tuples according to their values in one set of attributes G, called the *grouping attributes*.
  - For each group, aggregate the values in certain other attributes.
  - The normally permitted aggregations are SUM, COUNT, AVG, MIN, and MAX.
  - The result of this operation is one tuple for each group. That tuple has a component for each of the grouping attributes, with the value common to tuples of that group.
  - It also has a component for each aggregation, with the aggregated value for that group.

Selections using MapReduce

- **map:** for each tuple t in relation R, test if it satisfies C.
  - If it does, produce <t, t>
  - Otherwise produce nothing

- **reduce:** the identity function
  - The values (or keys) form the relation $\sigma_C(R)$

Projections using MapReduce

- **map:** For each tuple t in R, construct a tuple t' by eliminating from t those components whose attributes are not in S. Output the key-value pair <t', t'>.

- **reduce:** <t', $[t', t', \ldots, t']$> $\rightarrow$ <t', t'>

Operations using MapReduce

- **Union:**
  - map(t) → <t, t>
  - Reduce: <$t, [t]$> → <t, t> and <$t, [t, t]$> → <t, t>
- **Intersection:**
  - map(t) → <t, t>
  - reduce: <$t, [t, t]$> → <t, t> and <$t, [t]$> → nothing
- **Difference $\boldsymbol{R - S}$:**
  - map: t → <t, R> for t in R and t → <t, S> for t in S
  - reduce: produce <t, t> if value list is $[R]$, otherwise produce nothing

Natural Join using MapReduce

- Consider joining $R(A, B)$ and S(B, C)

- **map:** $(a, b) \rightarrow$ <b, (R, a)> for (a, b) in R
  $(b, c) \rightarrow$ <b, (S, c)> for (b, c) in S

- **reduce:**
  - each key b will be associated to a list of pairs each of the form (R, a) or (S, c)
  - Construct all triples (a, b, c) where (R, a) and (S, c) are associated to key b
  - These triples are your values and the keys are irrelevant

Group and Aggregation using MapReduce

- Consider $R(A, B, C)$ where A will be a grouping attribute, B an aggregated attribute, and C neither grouped nor aggregated.
- θ is the aggregation function, e.g., MAX, MIN, COUNT, etc.

- map$(a, b, c) \rightarrow <a, b>$

- reduce$(a, [b_1, \ldots, b_k]) \rightarrow (a, \theta([b_1, \ldots, b_k]))$

Matrix Multiplication $P = MN$

$$p_{ik} = \sum_j m_{ij} n_{jk}$$

- Can think of M and N as relations:
  - $M(I, J, V)$ with tuples (i, j, mij)
  - $M(J, K, W)$ with tuples (j, k, n_{jk})

- Good for sparse matrices (only need to record nonzero entries)

- Natural join: (i, j, k, v, w) for each (i, j, v) in M and (j, k, w) in N
  - This represents the pair (m_{ij}, n_{jk})
  - Want $(i, j, k, v \times w)$

Matrix Multiplication (cont.)

$$p_{ik} = \sum_j m_{ij} n_{jk}$$

- **map:** for m_{ij} produce <$j, (M, i, m_{ij})$>, for n_{jk} produce <$j, (N, k, n_{ik})$>
- **reduce:** for each value (M, i, m_{ij}) and each value (N, k, n_{ik}) associated to key j, produce <$(i, k), m_{ij} n_{jk}$>

- **map:** identity
- **reduce:** for each key (i, k) compute the sum of the associated values

Matrix multiplication in one step

- **map:**
  - for each m_{ij} produce <(i,k), (M,j,mij)> for all k
  - for each n_{jk} produce <(i,k), (N,j,n_{jk})> for all i

- **reduce:**
  - each key (i,k) will have an associated list with all the values (M,j,mij) and (N,j,n_{jk}) for all j
  - Must match up (M,j,mij) and (N,j,n_{jk}) for each j (sort by j)
  - Multiply the third components and sum these up to obtain p_{ik}
  - Output <(i,k),p_{ik}>

- Not necessarily faster