CSEP 544: Lecture 05

Query Optimization,
Parallel Databases,
MapReduce

Homework 3

- PigLatin (MapReduce) on AWS
- Go to http://aws.amazon.com/grants/ click on AWS Educate, get code for $100 credit for AWS
- Remember to turn off your instances!

Overview of Today's Lecture

- Query Execution/Optimization
  – Review two papers
- Parallel databases
- Map/Reduce
  – Next week: MR paper review
- Not in class: PigLatin
  – Read for HW3

Query Execution/Optimization

- Execution: logical/physical operators
  - Started last lecture, reviewed today
- Optimization: Query plans + rewrite rules
  - Today
- Size estimation: statistics + assumptions
  - Today

Will discuss in this order: 3, 1, 2

Database Statistics

- Collect statistical summaries of stored data
- Estimate <u>size</u> (=cardinality), bottom-up
- Estimate <u>cost</u> by using the estimated size

Database Statistics

- Number of tuples = cardinality
- Indexes: number of keys in the index
- Number of physical pages, clustering info
- Statistical information on attributes
  - Min value, max value, number distinct values
  - Histograms
- Correlations between columns

Collection approach: periodic, using sampling

Size Estimation Problem

```
S = SELECT list
     FROM   R1, …, Rn
     WHERE cond_1 AND cond_2 AND . . . AND cond_k
```

Given T(R1), T(R2), …, T(Rn)
Estimate T(S)

How can we do this ? Note: doesn't have to be exact.

Size Estimation Problem

```
S = SELECT list
     FROM   R1, …, Rn
     WHERE cond1 AND cond2 AND . . . AND condk
```

Remark: $T(S) \leq T(R1) \times T(R2) \times \ldots \times T(Rn)$

Selectivity Factor

- Each condition *cond* reduces the size by some factor called *selectivity factor*

- Assuming independence, multiply the selectivity factors

Example

R(A,B)
S(B,C)
T(C,D)

```
SELECT *
FROM R, S, T
WHERE R.B=S.B and S.C=T.C and R.A<40
```

T(R) = 30k, T(S) = 200k, T(T) = 10k

Selectivity of R.B = S.B is 1/3
Selectivity of S.C = T.C is 1/10
Selectivity of R.A < 40 is ½

What is the estimated size of the query output ?

Example

R(A,B)
S(B,C)
T(C,D)

```
SELECT *
FROM R, S, T
WHERE R.B=S.B and S.C=T.C and R.A<40
```

T(R) = 30k, T(S) = 200k, T(T) = 10k

Selectivity of R.B = S.B is 1/3
Selectivity of S.C = T.C is 1/10
Selectivity of R.A < 40 is ½

What is the estimated size of the query output ?

30k * 200k * 10k * 1/3 * 1/10 * ½
= 1TB

Discussion: Paper

What is the probability space?

S = SELECT list
FROM R_1 as x_1, …, R_k as x_k
WHERE Cond -- a conjunction of predicates

Discussion: Paper

What is the probability space?

S = SELECT list
FROM R_1 as x_1, …, R_k as x_k
WHERE Cond -- a conjunction of predicates

(x_1, x_2, …, x_k), drawn randomly, independently from R_1, ..., R_k

Pr(R_1.A = 40) = prob. that random tuple in R_1 has A=40

Descriptive attribute

Join indicator (in class…)

Pr(R_1.A = 40 and $J_{R1.B = R2.C}$ and R_2.D = 90) = prob. that …

E[|SELECT ... WHERE Cond|] = Pr(Cond) * T(R_1) * T(R_2) * ... * T(R_k)

Discussion: Paper

What is the probability space?

S = SELECT list
FROM R_1 as x_1, …, R_k as x_k
WHERE Cond -- a conjunction of predicates

What are the three simplifying assumptions?

Discussion: Paper

What is the probability space?

S = SELECT list
FROM R_1 as x_1, …, R_k as x_k
WHERE Cond -- a conjunction of predicates

What are the three simplifying assumptions?

Uniform: $Pr(R_1.A = \text{'a'}) = 1/V(R_1, A)$

Attribute Indep.: $Pr(R_1.A = \text{'a'} \text{ and } R_1.B = \text{'b'}) = Pr(R_1.A = \text{'a'})\, Pr(R_1.B = \text{'b'})$

Join Indep.: $Pr(R_1.A = \text{'a'} \text{ and } J_{R1.B = R2.C}) = Pr(R_1.A = \text{'a'})\, Pr(J_{R1.B = R2.C})$

Rule of Thumb

- If selectivities are unknown, then:
  selectivity factor = 1/10
  [System R, 1979]

Using Data Statistics

- Condition is $A = c$ /* value selection on R */
  - Selectivity = $1/V(R,A)$

- Condition is $A < c$ /* range selection on R */
  - Selectivity = $(c - Low(R, A))/(High(R,A) - Low(R,A))T(R)$

- Condition is $A = B$ /* $R \bowtie_{A=B} S$ */
  - Selectivity = $1 / \max(V(R,A),V(S,A))$
  - (will explain next)

Selectivity of Join Predicates

Assumptions:

- *Containment of values*: if $V(R,A) <= V(S,B)$, then the set of A values of R is included in the set of B values of S
  - Note: this indeed holds when A is a foreign key in R, and B is a key in S

- *Preservation of values*: for any other attribute C, $V(R \bowtie_{A=B} S, C) = V(R, C)$ (or $V(S, C)$)

Selectivity of Join Predicates

Assume $V(R,A) <= V(S,B)$

- Each tuple t in R joins with $T(S)/V(S,B)$ tuple(s) in S
- Hence $T(R \bowtie_{A=B} S) = T(R)\ T(S) / V(S,B)$

In general: $T(R \bowtie_{A=B} S) = T(R)\ T(S) / \max(V(R,A),V(S,B))$

Selectivity of Join Predicates

Example:

- $T(R) = 10000$, $T(S) = 20000$
- $V(R,A) = 100$, $V(S,B) = 200$
- How large is $R \bowtie_{A=B} S$?

Histograms

- Statistics on data maintained by the RDBMS
- Makes size estimation much more accurate (hence, cost estimations are more accurate)

Histograms

Employee(ssn, name, age)

T(Employee) = 25000, V(Empolyee, age) = 50
min(age) = 19, max(age) = 68

$\sigma_{age=48}$(Empolyee) = ? $\sigma_{age>28 \text{ and } age<35}$(Empolyee) = ?

Histograms

Employee(ssn, name, age)

T(Employee) = 25000, V(Empolyee, age) = 50
min(age) = 19, max(age) = 68

$\sigma_{age=48}$(Empolyee) = ? $\sigma_{age>28 \text{ and } age<35}$(Empolyee) = ?

Age:	0..20	20..29	30-39	40-49	50-59	> 60
Tuples	200	800	5000	12000	6500	500

Histograms

Employee(ssn, name, age)

T(Employee) = 25000, V(Empolyee, age) = 50
min(age) = 19, max(age) = 68

$\sigma_{age=48}$(Empolyee) = ? $\sigma_{age>28 \text{ and } age<35}$(Empolyee) = ?

Age:	0..20	20..29	30-39	40-49	50-59	> 60
Tuples	200	800	5000	12000	6500	500

Estimate = 1200 Estimate = 1*80 + 5*500 = 2580

Types of Histograms

- How should we determine the bucket boundaries in a histogram ?

Types of Histograms

- How should we determine the bucket boundaries in a histogram ?

- Eq-Width
- Eq-Depth
- Compressed
- V-Optimal histograms

Employee(ssn, name, age)

Histograms

Eq-width:

Age:	0..20	20..29	30-39	40-49	50-59	> 60
Tuples	200	800	5000	12000	6500	500

Eq-depth:

Age:	0..20	20..29	30-39	40-49	50-59	> 60
Tuples	1800	2000	2100	2200	1900	1800

Compressed: store separately highly frequent values: (48,1900)

V-Optimal Histograms

- Defines bucket boundaries in an optimal way, to minimize the error over all point queries
- Computed rather expensively, using dynamic programming
- Modern databases systems use V-optimal histograms or some variations

Difficult Questions on Histograms

- Small number of buckets
  - Hundreds, or thousands, but not more
  - WHY ?
- *Not* updated during database update, but recomputed periodically
  - WHY ?

Multidimensional Histograms

Classical example:

SQL query:

```
SELECT …  FROM   …
WHERE  Person.city = ‘Seattle’ …
```

User “optimizes” it to:

```
SELECT …  FROM   …
WHERE  Person.city = ‘Seattle’
       and Person.state = ‘WA’
```

Big problem! (Why?)

Multidimensional Histograms

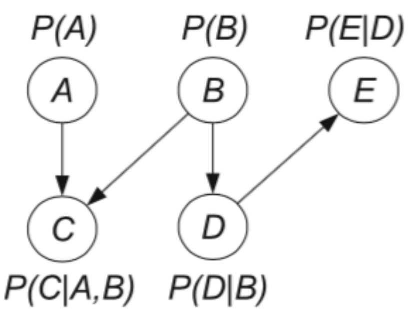
- Store distributions on two or more attributes
- Curse of dimensionality: space grows exponentially with dimension
- Paper: discusses using only two dimensional histograms

Paper: Bayesian Networks

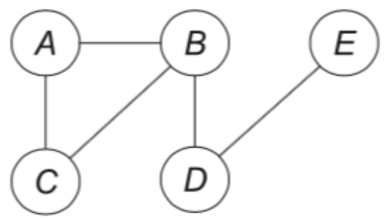
$P_{BN}(A, B, C, D, E) = P(E|D)P(D|B)P(C|A, B)\ P(A)P(B).$

Paper: Bayesian Networks

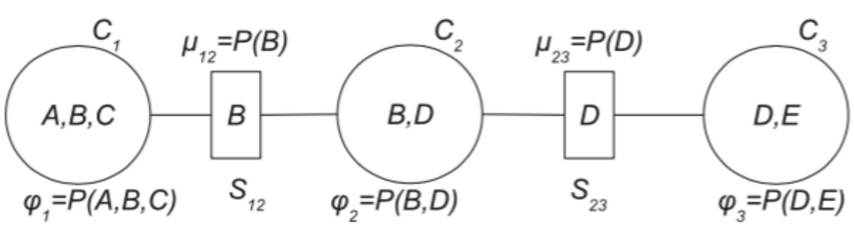
$P_{BN}(A, B, C, D, E) = P(E|D)P(D|B)P(C|A, B)\ P(A)P(B).$



(a)



(b)



(c)

a	b	c	$P(a,b,c)$
a_1	b_1	c_1	0.25
a_1	b_1	c_2	0.32
a_1	b_2	c_1	0.01
a_1	b_2	c_2	0.12
a_2	b_1	c_1	0.08
a_2	b_1	c_2	0.04
a_2	b_2	c_1	0.1
a_2	b_2	c_2	0.08

b	d	$P(b,d)$
b_1	d_1	0.4
b_1	d_2	0.3
b_2	d_1	0.15
b_2	d_2	0.15

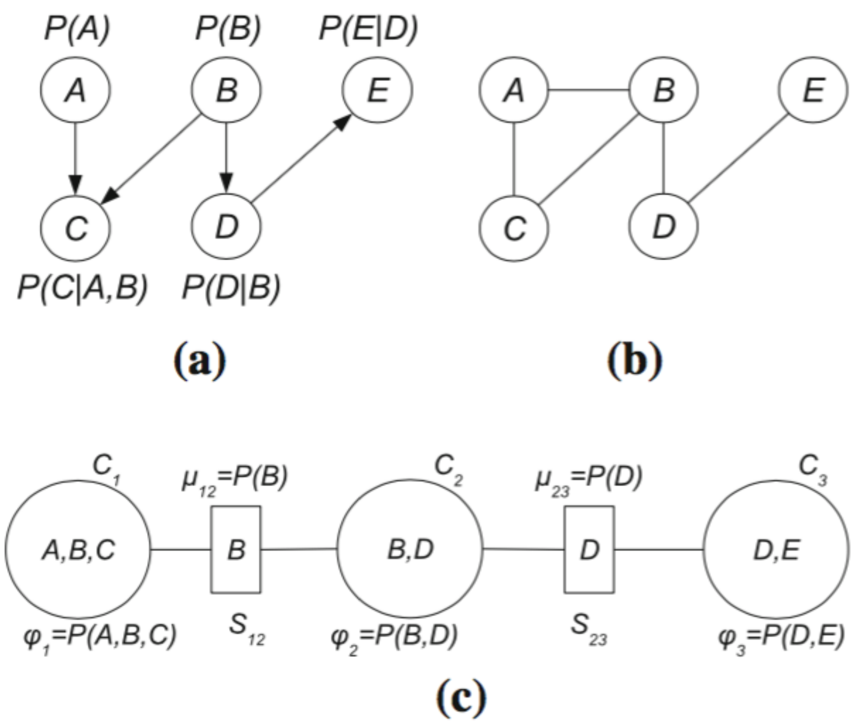
d	e	$P(d,e)$
d_1	e_1	0.7
d_1	e_2	0.1
d_2	e_1	0.05
d_2	e_2	0.15

(d)

Fig. 1 A small graphical model of five binary random variables A, B, C, D, E **a** Bayesian network. **b** Moral graph. **c** Junction tree. **d** Clique potentials

Paper: Bayesian Networks

$P_{BN}(A, B, C, D, E) = P(E|D)P(D|B)P(C|A, B)\, P(A)P(B).$



a	b	c	$P(a,b,c)$
a_1	b_1	c_1	0.25
a_1	b_1	c_2	0.32
a_1	b_2	c_1	0.01
a_1	b_2	c_2	0.12
a_2	b_1	c_1	0.08
a_2	b_1	c_2	0.04
a_2	b_2	c_1	0.1
a_2	b_2	c_2	0.08

b	d	$P(b,d)$
b_1	d_1	0.4
b_1	d_2	0.3
b_2	d_1	0.15
b_2	d_2	0.15

d	e	$P(d,e)$
d_1	e_1	0.7
d_1	e_2	0.1
d_2	e_1	0.05
d_2	e_2	0.15

(d)

Fig. 1 A small graphical model of five binary random variables A, B, C, D, E **a** Bayesian network. **b** Moral graph. **c** Junction tree. **d** Clique potentials

$$P(A, D) = \sum_{B,C} \frac{P(A, B, C)P(B, D)}{P(B)}.$$

Paper Highlights

- Universal table (what is it?)
- Acyclic v.s. Cyclic Schemas
- Within a table: tree-BN only
- Join indicator: two parents only
- Hence: acyclic schema → 2D-histograms only in the junction tree
- Simplifies construction, estimation

Summary of Size Estimation

- Critical, yet very difficult piece of a query optimizer
- *Selectivity estimation:* simple probability space (outcome = 1 tuple) to estimate a selection (includes joins)
- More complex estimations: much more difficult (e.g. estimate the size of DISTINCT)

Query Execution

- Logical operators:
  - Select/project/join/union/difference
  - Group-by/sort
- Physical operators:
  - Main memory (“in core”)
    e.g. hash-join, merge-join
  - External memory (“out of core”)
    index-join, partitioned hash join, merge join

Discussion: Shapiro's paper

- Describe the *merge-join* algorithm. How long are the initial runs?
- What is *classic hashing?*
- What is *simple hash-join?*
- What is *Grace-join?*
- What is *Hybrid hash-join?*

Advanced Stuff

- Semi-joins
- Anti-semi-joins

Semijoin

$$R \ltimes_C S = \Pi_{A1,\ldots,An} (R \bowtie_C S)$$

Where $A_1, \ldots, A_n$ are the attributes in R

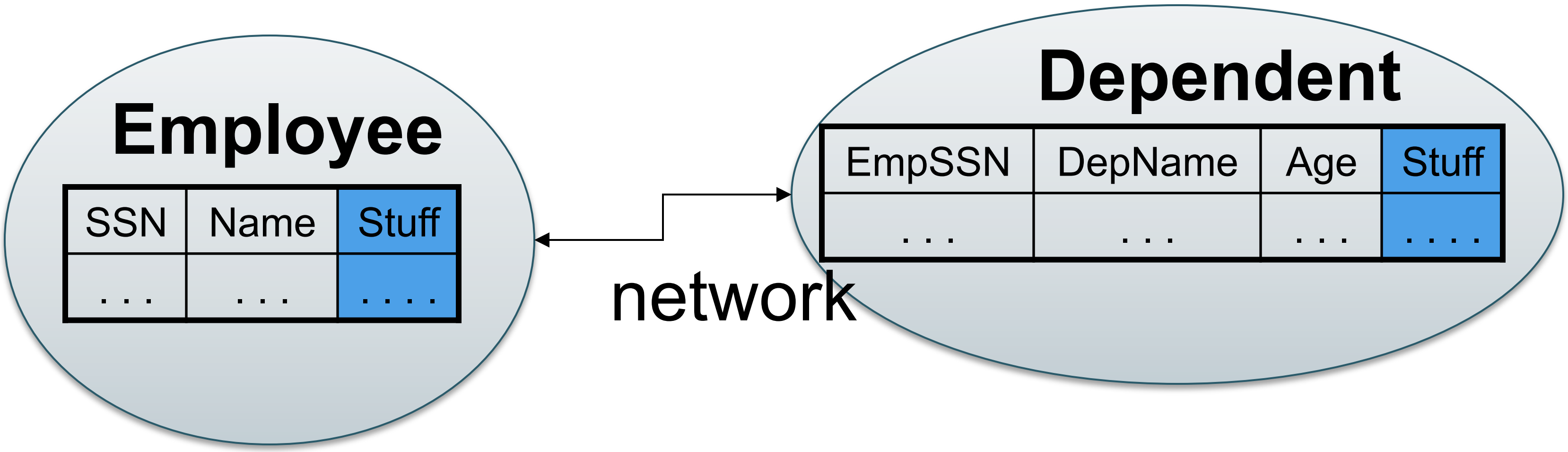
Formally, R ⋉C S means this: retain from R only those tuples that have some matching tuple in S

Duplicates in R are preserved

Duplicates in S don't matter

Applications: distributed query execution, standard query execution for complex queries

Semijoins in Distributed Databases

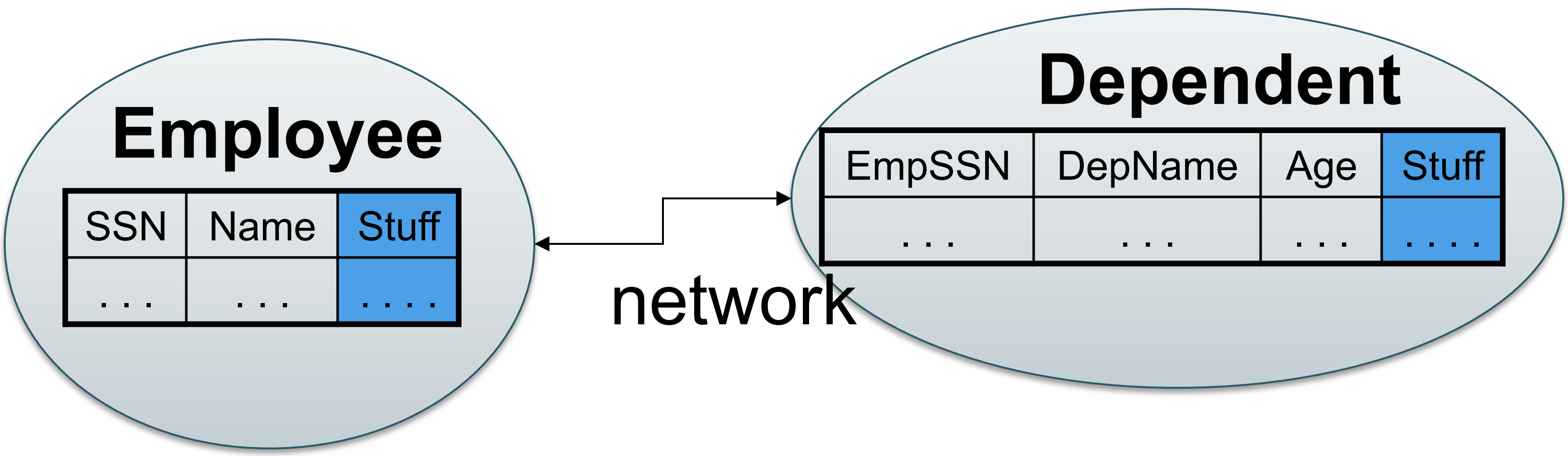


$$\sigma_{\text{name like 'M\%'}}(\text{Employee}) \bowtie_{\text{SSN=EmpSSN}} (\sigma_{\text{age>71}}(\text{Dependent}))$$

Assumptions

- Very few employees start with M
- Very few dependents have age > 71.
- "Stuff" is big.

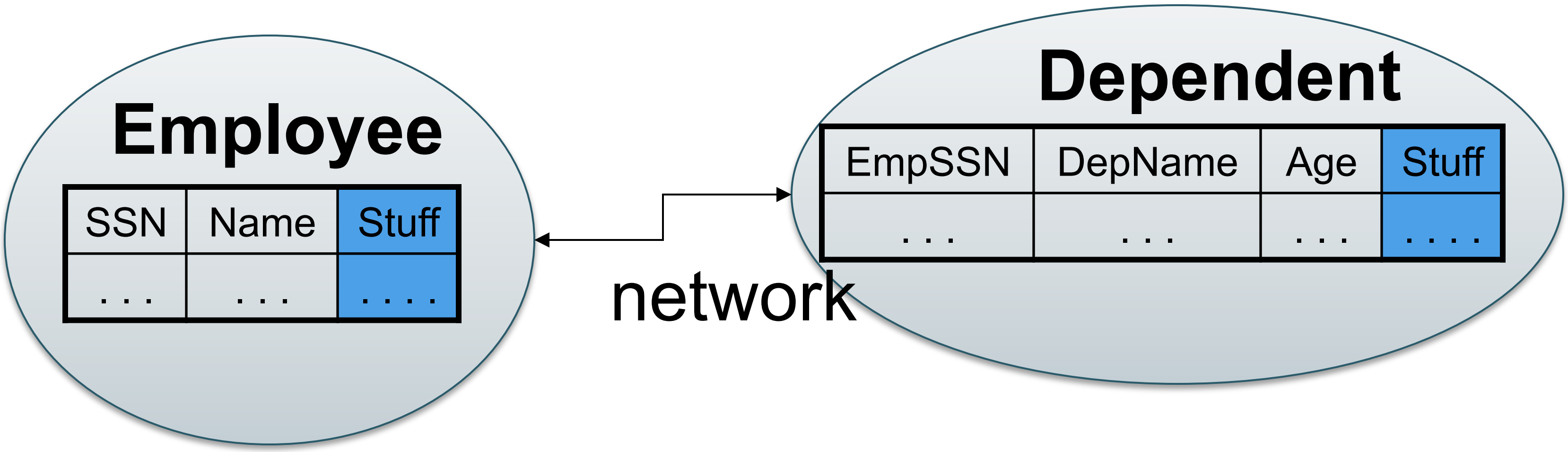
Semijoins in Distributed Databases



$$\sigma_{\text{name like 'M\%'}}(\text{Employee}) \bowtie_{\text{SSN=EmpSSN}} (\sigma_{\text{age>71}}(\text{Dependent}))$$

$$T = \Pi_{\text{EmpSSN}}\ \sigma_{\text{age>71}}(\text{Dependents})$$

Semijoins in Distributed Databases

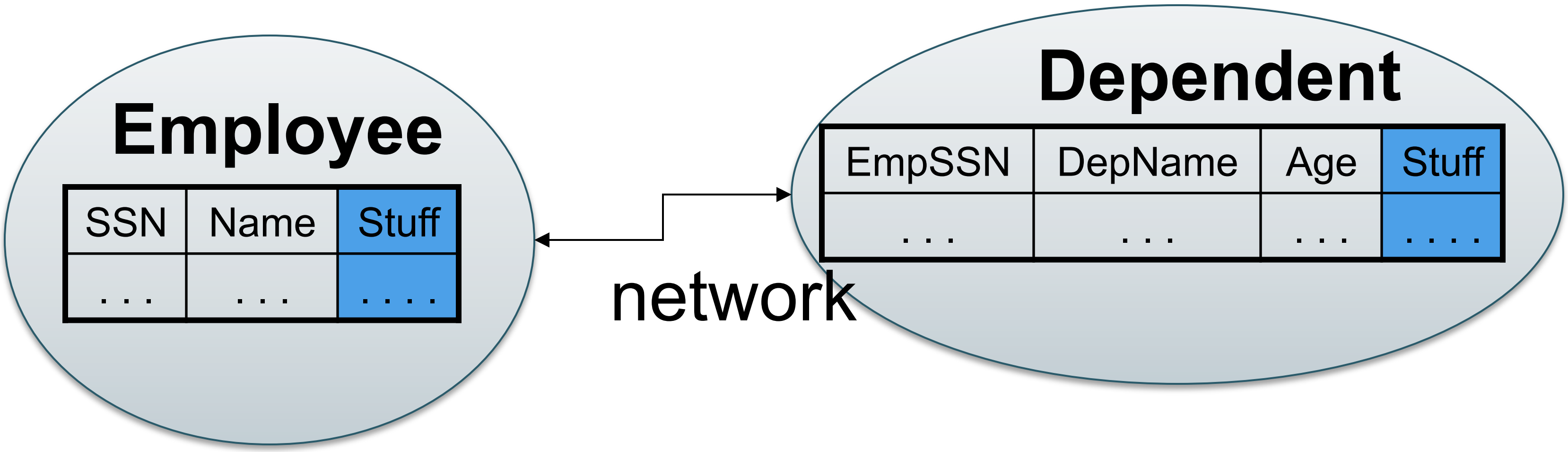


$$\sigma_{\text{name like 'M\%'}}(\text{Employee}) \bowtie_{\text{SSN=EmpSSN}} (\sigma_{\text{age>71}} (\text{Dependent}))$$

$$T = \Pi_{\text{EmpSSN}}\ \sigma_{\text{age>71}} (\text{Dependents})$$

$$R = \sigma_{\text{name like 'M\%'}}(\text{Employee}) \bowtie_{\text{SSN=EmpSSN}} T$$

Semijoins in Distributed Databases



$\sigma_{\text{name like 'M\%'}}(\text{Employee}) \bowtie_{\text{SSN=EmpSSN}} (\sigma_{\text{age>71}} (\text{Dependent}))$

$T = \Pi_{\text{EmpSSN}}\ \sigma_{\text{age>71}} (\text{Dependents})$

$R = \sigma_{\text{name like 'M\%'}}(\text{Employee}) \bowtie_{\text{SSN=EmpSSN}} T$

$\text{Answer} = R \bowtie_{\text{SSN=EmpSSN}} \sigma_{\text{age>71}}\ \text{Dependents`}$

Anti-Semi-Join

- Notation: $R \rhd S$
  - Warning: not a standard notation
- Meaning: all tuples in R that do NOT have a matching tuple in S

R(A,B)
S(B)

Set Difference v.s. Anti-semijoin

```
SELECT DISTINCT R.B
FROM R
WHERE not exists  (SELECT *
                   FROM S
                   WHERE R.B=S.B)
```

Plan=

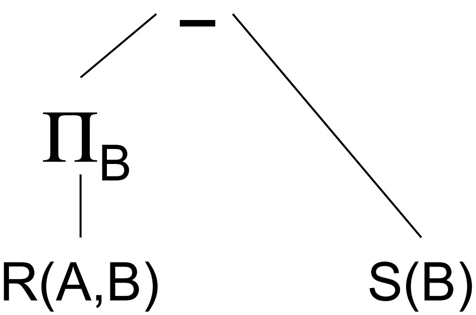
```
SELECT DISTINCT *
FROM R
WHERE not exists  (SELECT *
                   FROM S
                   WHERE R.B=S.B)
```

R(A,B)
S(B)

Set Difference v.s. Anti-semijoin

```
SELECT DISTINCT R.B
FROM R
WHERE not exists  (SELECT *
                   FROM S
                   WHERE R.B=S.B)
```

Plan=



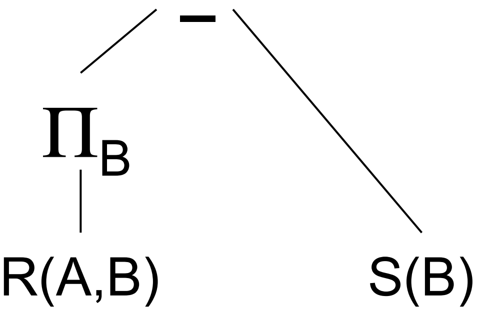

```
SELECT DISTINCT *
FROM R
WHERE not exists  (SELECT *
                   FROM S
                   WHERE R.B=S.B)
```

R(A,B)
S(B)

Set Difference v.s. Anti-semijoin

```
SELECT DISTINCT R.B
FROM R
WHERE not exists  (SELECT *
                   FROM S
                   WHERE R.B=S.B)
```

Plan=



Plan=

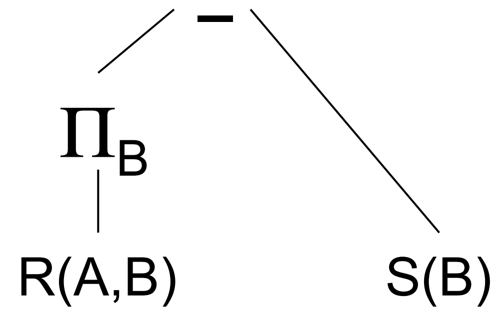
```
SELECT DISTINCT *
FROM R
WHERE not exists  (SELECT *
                   FROM S
                   WHERE R.B=S.B)
```

R(A,B)
S(B)

Set Difference v.s. Anti-semijoin

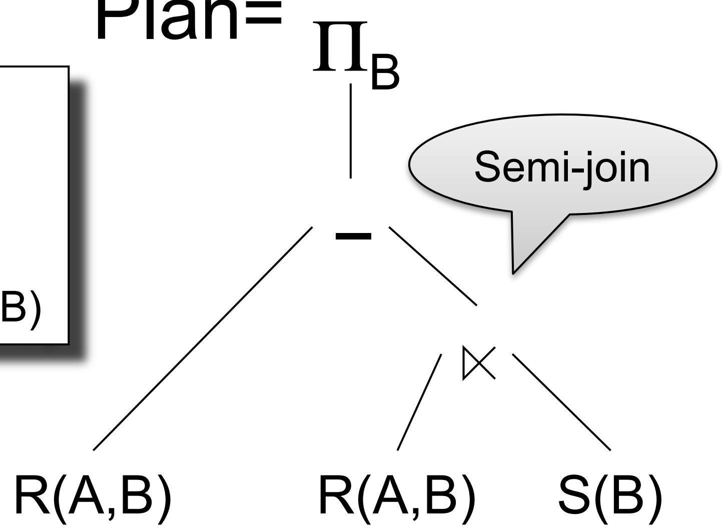
```
SELECT DISTINCT R.B
FROM R
WHERE not exists  (SELECT *
                   FROM S
                   WHERE R.B=S.B)
```

Plan=




```
SELECT DISTINCT *
FROM R
WHERE not exists  (SELECT *
                   FROM S
                   WHERE R.B=S.B)
```

Plan=

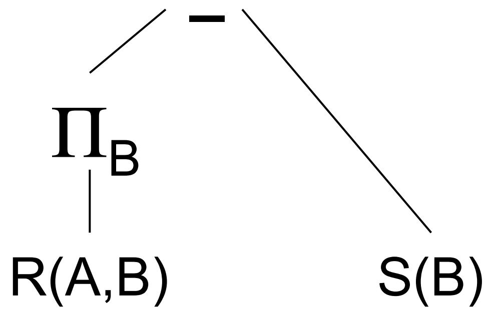


R(A,B)
S(B)

Set Difference v.s. Anti-semijoin

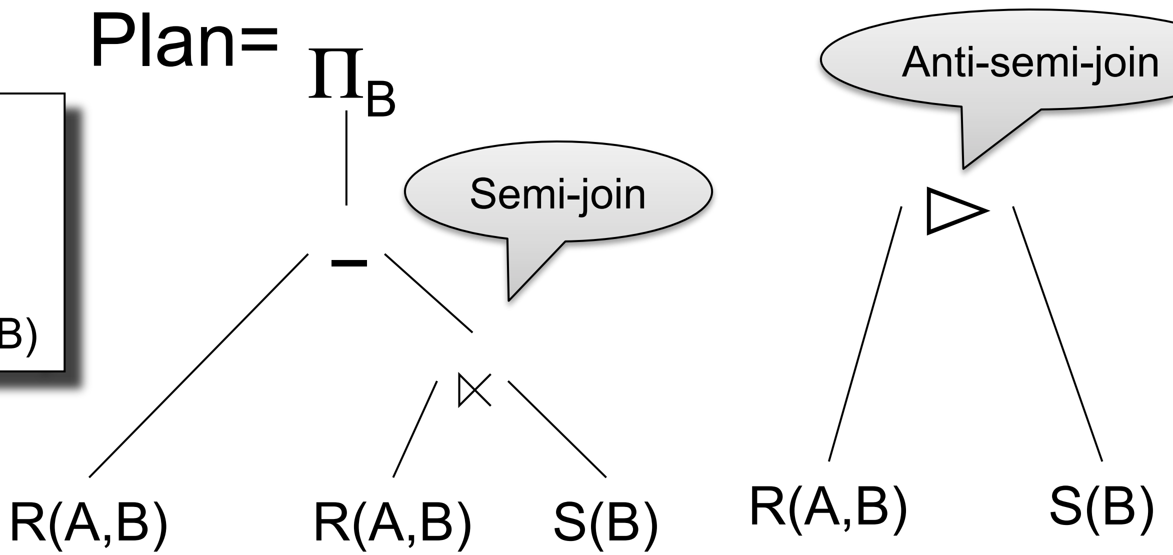
```
SELECT DISTINCT R.B
FROM R
WHERE not exists  (SELECT *
                   FROM S
                   WHERE R.B=S.B)
```

Plan=




```
SELECT DISTINCT *
FROM R
WHERE not exists  (SELECT *
                   FROM S
                   WHERE R.B=S.B)
```

Plan=



Operators on Bags

- Duplicate elimination δ

  $\delta(R)$ = SELECT DISTINCT * FROM R

- Grouping γ

  $\gamma_{A,sum(B)}(R)$ =

  SELECT A,sum(B) FROM R GROUP BY A

- Sorting τ

Query Optimization

- Search space = set of all physical query plans considered

- Search algorithm = a heuristics-based algorithm for searching the space and selecting an optimal plan

Relational Algebra Laws: Joins

Commutativity :	$R \bowtie S = S \bowtie R$
Associativity:	$R \bowtie (S \bowtie T) = (R \bowtie S) \bowtie T$
Distributivity:	$R \bowtie (S \cup T) = (R \bowtie S) \cup (R \bowtie T)$

Outer joins get more complicated

Relational Algebra Laws: Selections

R(A, B, C, D), S(E, F, G)

$\sigma_{F=3}(R \bowtie_{D=E} S) =$?

$\sigma_{A=5 \text{ AND } G=9}(R \bowtie_{D=E} S) =$?

Relational Algebra Laws: Selections

R(A, B, C, D), S(E, F, G)

$$\sigma_{F=3}(R \bowtie_{D=E} S) = R \bowtie_{D=E} (\sigma_{F=3}(S))$$
$$\sigma_{A=5 \text{ AND } G=9}(R \bowtie_{D=E} S) = \sigma_{A=5}(R) \bowtie_{D=E} \sigma_{G=9}(S)$$

Group-by and Join

R(A, B), S(C,D)

$$\gamma_{A,\ sum(D)}(R(A,B) \bowtie_{B=C} S(C,D)) = \quad ?$$

Group-by and Join

R(A, B), S(C,D)

$$\gamma_{A,\ sum(D)}(R(A,B) \bowtie_{B=C} S(C,D)) =$$
$$\gamma_{A,\ sum(D)}(R(A,B) \bowtie_{B=C} (\gamma_{C,\ sum(D)} S(C,D)))$$

These are very powerful laws.
They were introduced only in the 90's.

Laws Involving Constraints

Foreign key

Product(pid, pname, price, cid)
Company(cid, cname, city, state)

$$\Pi_{\text{pid, price}}(\text{Product} \bowtie_{\text{cid=cid}} \text{Company}) = \quad ?$$

Laws Involving Constraints

Foreign key

Product(pid, pname, price, cid)
Company(cid, cname, city, state)

$$\Pi_{pid,\ price}(\text{Product} \bowtie_{cid=cid} \text{Company}) = \Pi_{pid,\ price}(\text{Product})$$

Need a second constraint for this law to hold. Which ?

Why such queries occur

Foreign key

Product(pid, pname, price, cid)
Company(cid, cname, city, state)

```
CREATE VIEW CheapProductCompany
    SELECT *
    FROM Product x, Company y
    WHERE x.cid = y.cid and x.price < 100
```

```
SELECT pname, price
FROM CheapProductCompany
```

⇒

```
SELECT pname, price
FROM Product
WHERE price < 100
```

Law of Semijoins

- Input: R(A1,…An), S(B1,…,Bm)
- Output: T(A1,…,An)
- Semjoin is: $R \ltimes S = \Pi_{A1,\ldots,An}(R \bowtie S)$

- The law of semijoins is:

$$\boxed{R \bowtie S = (R \ltimes S) \bowtie S}$$

Laws with Semijoins

- Used in parallel/distributed databases
- Often combined with Bloom Filters
- Read pp. 747 in the textbook

The Iterator Model

Each operator implements this interface

- open()
- get_next()
- close()

See details on the slides of the previous lecture

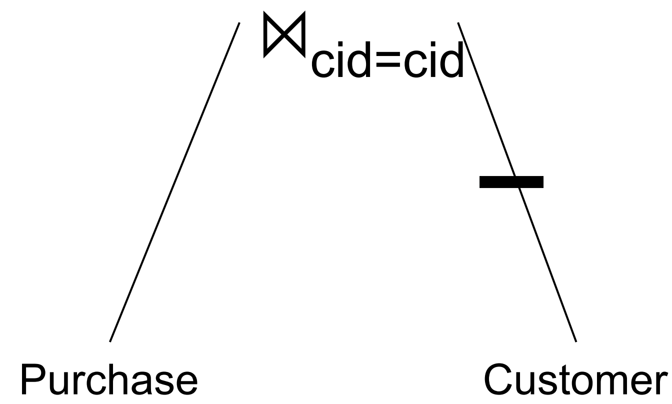
Purchase(<u>pid, cid</u>, store)
Customer(<u>cid</u>, name, city)

Purchase(<u>pid,cid</u>,store) $\bowtie_{cid=cid}$ Customer(<u>cid</u>, name, city)

Classic Hash Join

What do these operators do for the classic Hash Join?

- open()
- get_next()
- close()



Purchase(pid, cid, store)
Customer(cid, name, city)

Purchase(pid,cid,store) $\bowtie_{cid=cid}$ Customer(cid, name, city)

Main Memory Hash Join

```
open( ) {
   Customer.open( );
   while (c = Customer.get_next( ))
      hashTable.insert(c.cid, c);
   Customer.close();
   Purchase.open( );
}
```

```
get_next( ) {
   repeat {
      p = Purchase.get_next( );
      if (p == NULL)
         { c = hashTable.find(p.cid); }
   until (p == NULL or c != NULL);
   return (p,c)
}
```

```
close( ) {
   Purchase.close( );
}
```

Purchase(pid, cid, store)
Customer(cid, name, city)

Purchase(pid,cid,store) $\bowtie_{cid=cid}$ Customer(cid, name, city)

Main Memory Hash Join

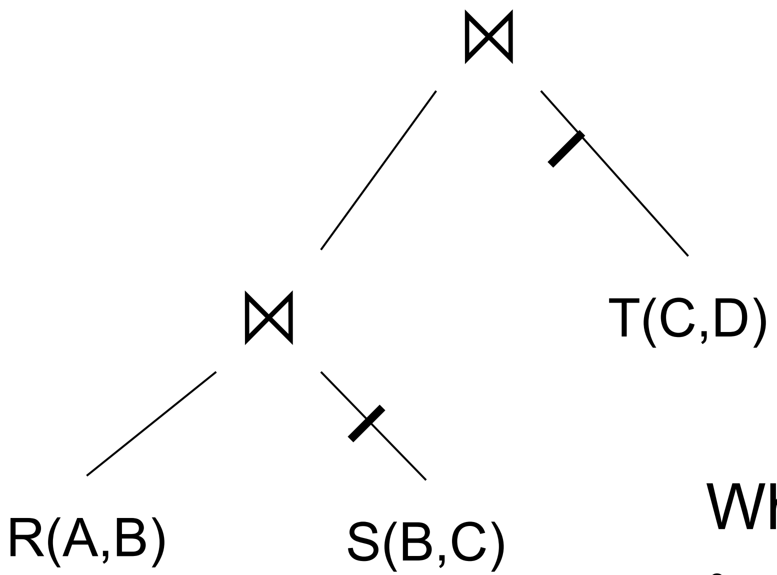
```
open( ) {
  Customer.open( );
  while (c = Customer.get_next( ))
     hashTable.insert(c.cid, c);
  Customer.close();
  Purchase.open( );
}
```

```
get_next( ) {
  repeat {
     p = Purchase.get_next( );
     if (p == NULL)
        { c = hashTable.find(p.cid); }
  until (p == NULL or c != NULL);
  return (p,c)
}
```

```
close( ) {
  Purchase.close( );
}
```

What changes if we don’t join on a key-foreign key?

Discussion in class

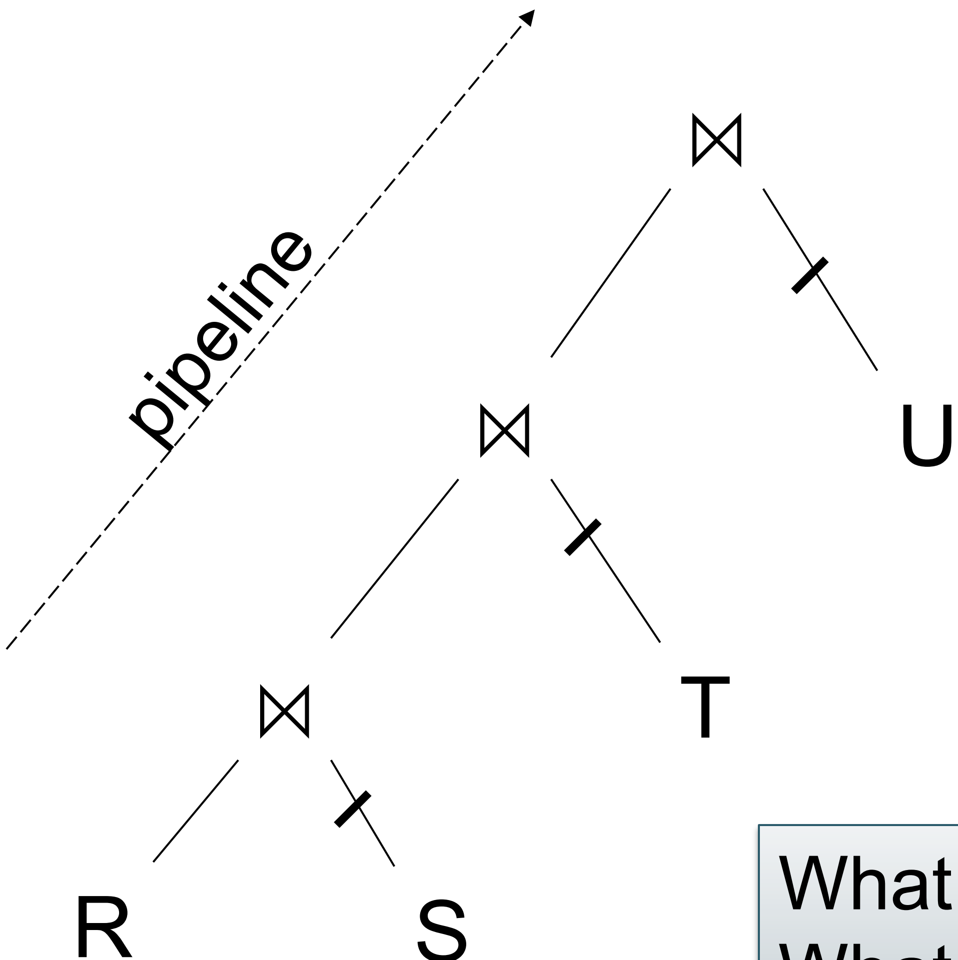


Every operator is a hash-join and implements the iterator model

What happens:

- When we call open() at the top?
- When we call get_next() at the top?

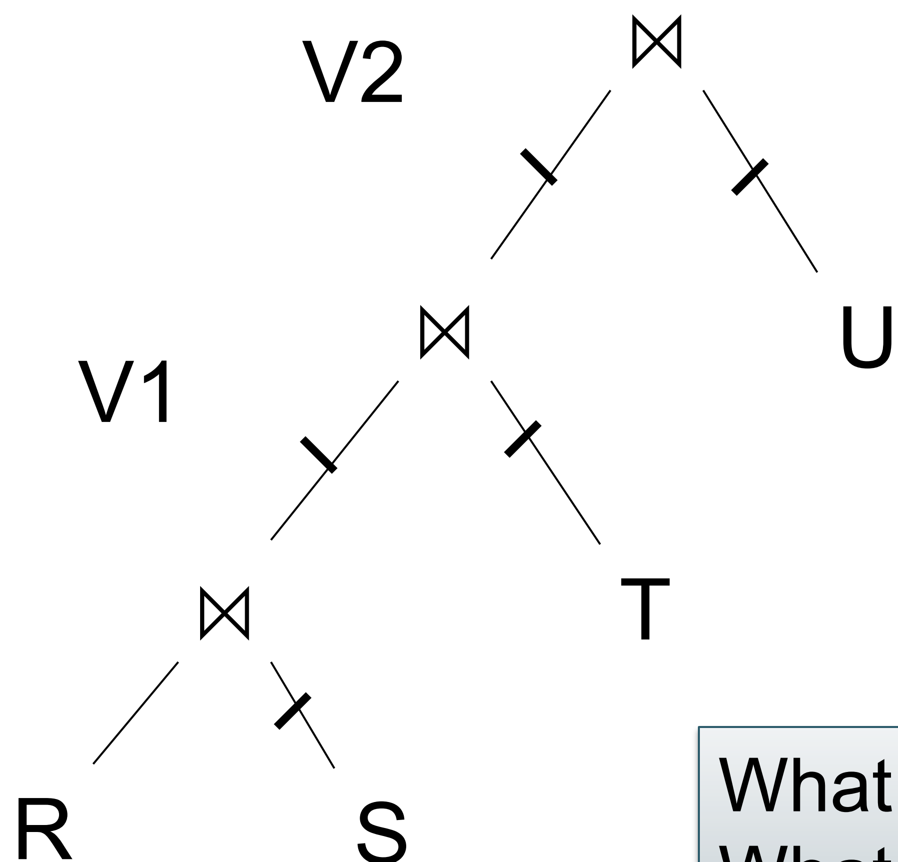
Discussion in class




```
HashTable1 ← S
HashTable2 ← T
HashTable3 ← U
repeat    read(R, x)
          y ← join(HashTable1, x)
          z ← join(HashTable2, y)
          u ← join(HashTable3, z)
          write(Answer, u)
```

What is the total cost?
What is the requirement on M?

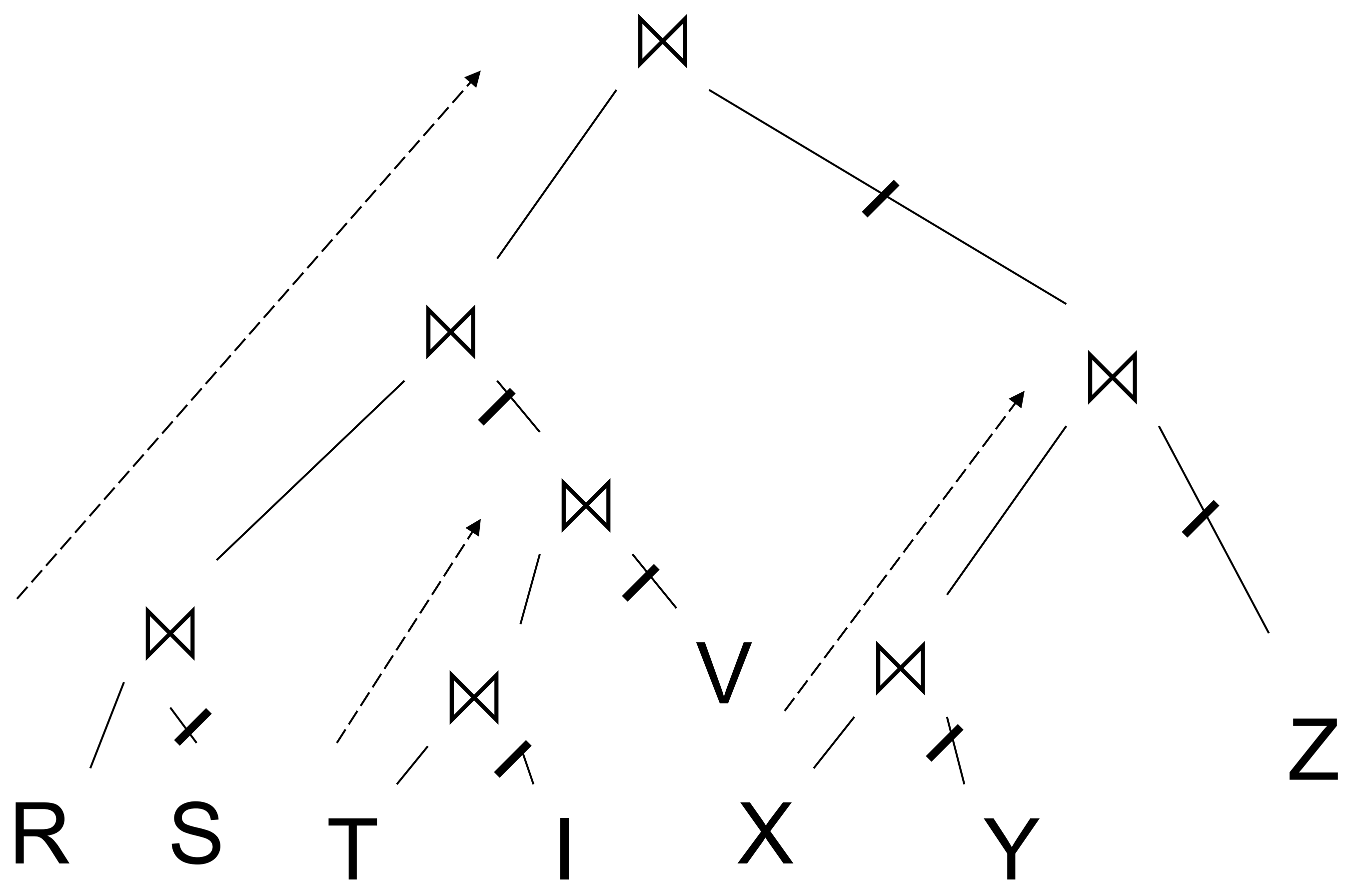
Discussion in class



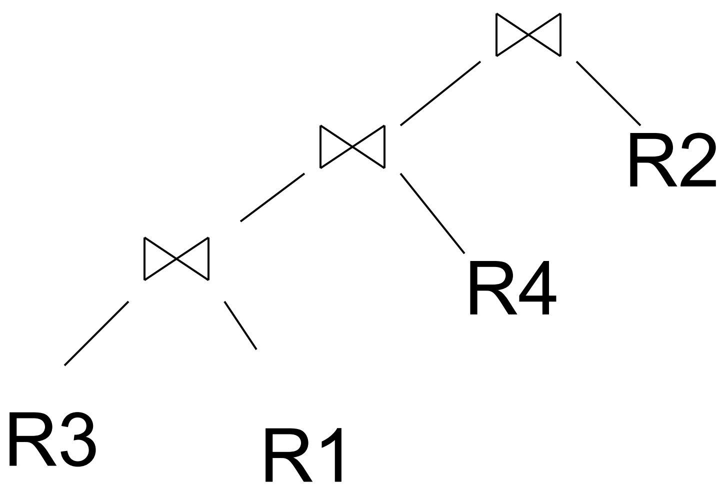

```
HashTable ← S
repeat    read(R, x)
          y ← join(HashTable, x)
          write(V1, y)

HashTable ← T
repeat    read(V1, y)
          z ← join(HashTable, y)
          write(V2, z)

HashTable ← U
repeat    read(V2, z)
          u ← join(HashTable, z)
          write(Answer, u)
```

What is the total cost?
What is the requirement on M?

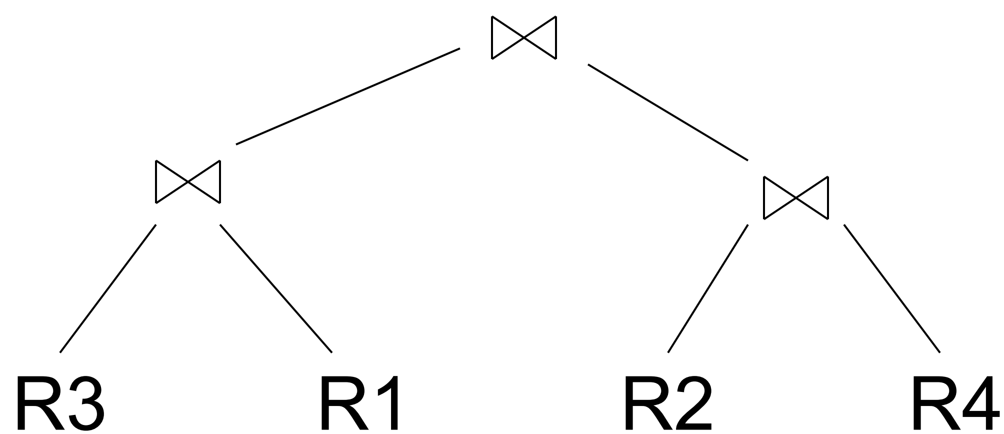
Discussion in class



Left-Deep Plans and Bushy Plans



Left-deep plan



Bushy plan

System R considered only left deep plans, and so do some optimizers today

Search Algorithms

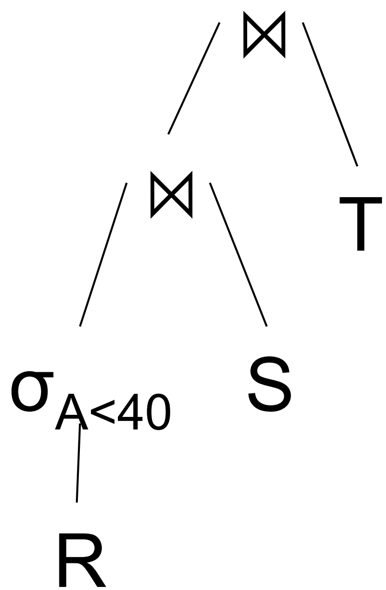
Goal: start with any query plan, find an equivalent plan with lowest estimated cost

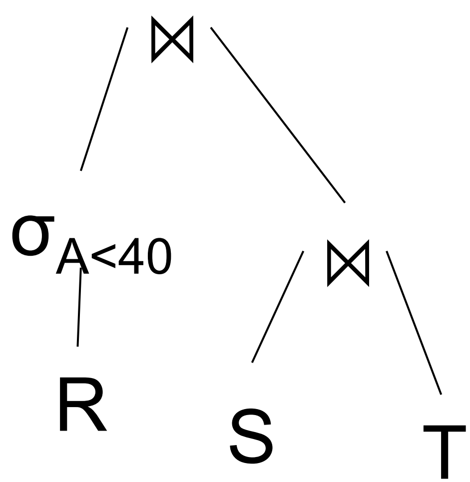
- Dynamic programming
  - Pioneered by System R for computing optimal join order
- Search space pruning
  - Drop unpromising partial plans; bottom-up v.s. top-down plans
- Access path selection
  - Refers to the plan for accessing a single table

Complete Plans

R(A,B)
S(B,C)
T(C,D)

```
SELECT *
FROM R, S, T
WHERE R.B=S.B and S.C=T.C and R.A<40
```





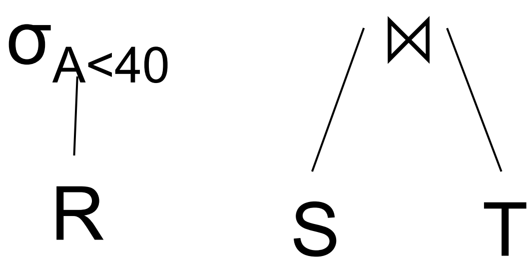
If the algorithm enumerates *complete* plans, then it is difficult to prune out unpromising sets of plans.

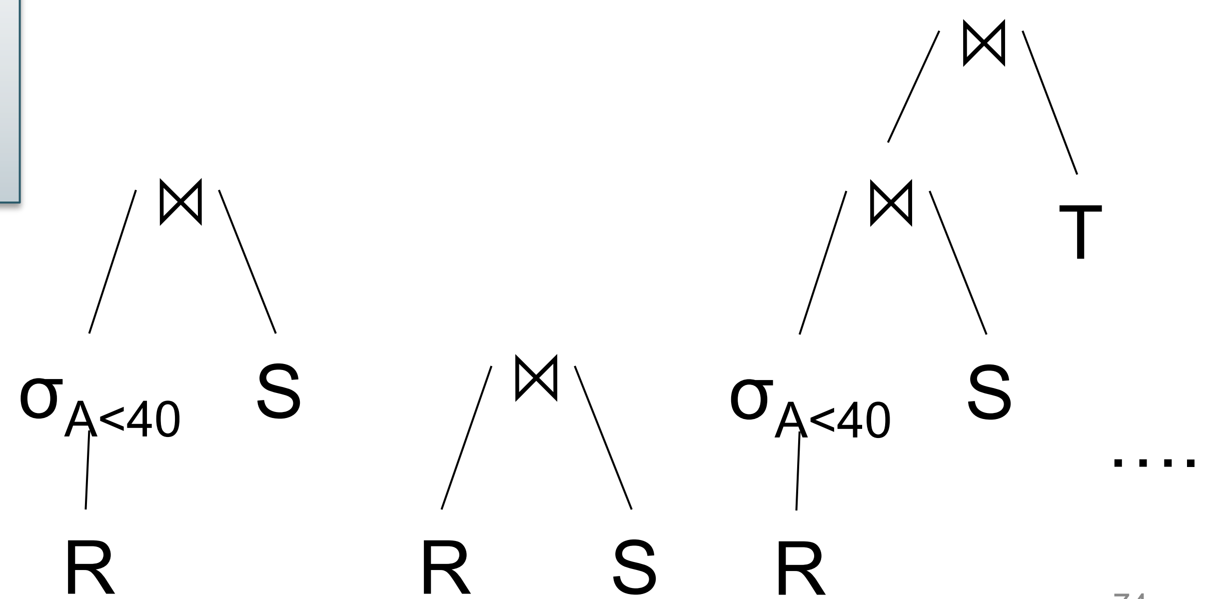
Bottom-up Partial Plans

R(A,B)
S(B,C)
T(C,D)

```
SELECT *
FROM R, S, T
WHERE R.B=S.B and S.C=T.C and R.A<40
```

If the algorithm enumerates partial bottom-up plans, then pruning can be done more efficiently



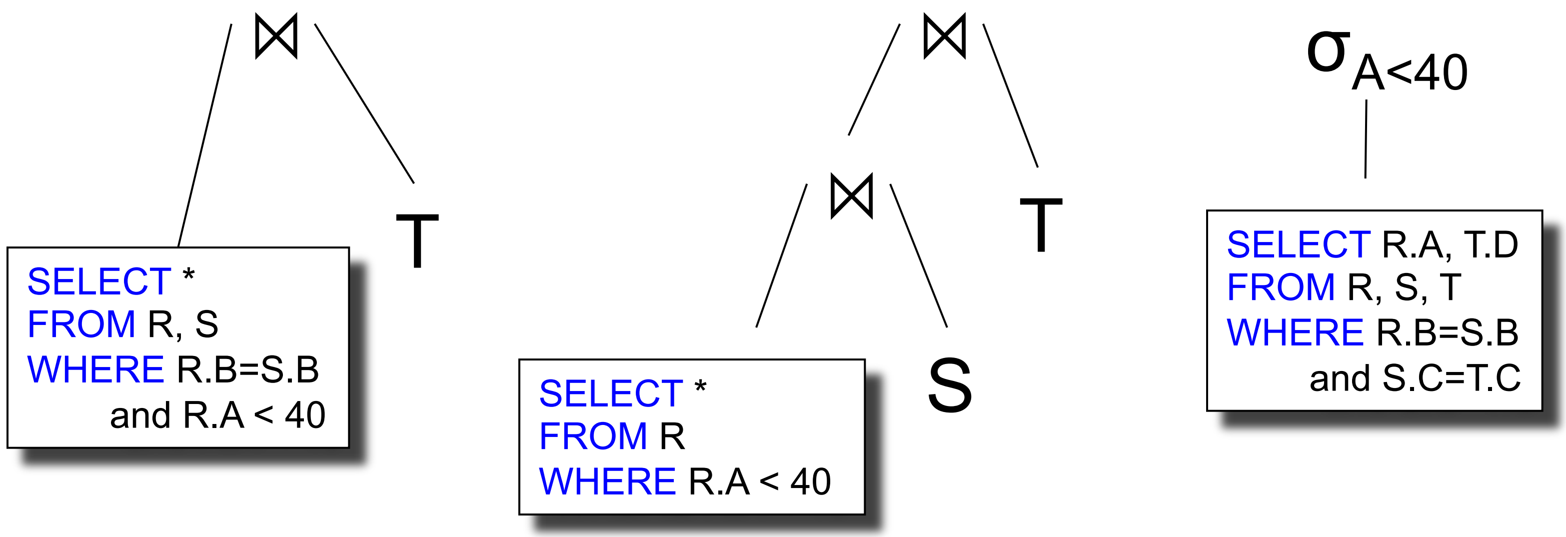


Top-down Partial Plans

R(A,B)
S(B,C)
T(C,D)

```
SELECT *
FROM R, S, T
WHERE R.B=S.B and S.C=T.C and R.A<40
```

Same here.



.....

Access Path Selection

Supplier(sid,sname,scategory,scity,sstate)

$\sigma_{\text{scategory = 'organic'} \wedge \text{scity='Seattle'}}$ (Supplier)

Clustered index on scity
Unclustered index on (scategory,scity)

B(Supplier) = 10k
T(Supplier) = 1M

V(Supplier,city) = 1000
V(Supplier,scategory)=100

Access plan options:

- Table scan: cost = ?
- Index scan on scity: cost = ?
- Index scan on scategory,scity: cost = ?

Access Path Selection

Supplier(sid,sname,scategory,scity,sstate)

$\sigma_{\text{scategory = 'organic'} \wedge \text{scity='Seattle'}}$ (Supplier)

B(Supplier) = 10k
T(Supplier) = 1M

V(Supplier,city) = 1000
V(Supplier,scategory)=100

Clustered index on scity
Unclustered index on (scategory,scity)

Access plan options:

- Table scan: cost = 10k = 10k
- Index scan on scity: cost = 10k/1000 = 10
- Index scan on scategory,scity: cost = 1M/1000*100 = 10

Summary of Query Optimization

- Three parts:
  - search space, algorithms, size/cost estimation
- Ideal goal: find optimal plan. But
  - Impossible to estimate accurately
  - Impossible to search the entire space
- Goal of today's optimizers:
  - Avoid very bad plans

Overview of Today's Lecture

- Query Execution/Optimization
- Parallel databases
  – Book: Ch. 22.1 – 22.10
- Map/Reduce
  – Next week: MR paper review
- Not in class: PigLatin
  – Read for HW3

Parallel Databases

Parallel Computation Today

Two Major Forces Pushing towards Parallel Computing:

- Change in Moore's law
- Cloud computing

Parallel Computation Today

1. Change in Moore's law* (exponential growth in transistors per chip density) no longer results in increased clock speeds
   – Increased hw performance available only through parallelism
   – Think multicore: 4 cores today, perhaps 64 in a few years

* Moore's law says that the number of transistors that can be placed inexpensively on an integrated circuit doubles approximately every two years [Intel co-founder Gordon E. Moore described the trend in his 1965 paper and predicted that it will last for at least 10 years]

Parallel Computation Today

2. Cloud computing commoditizes access to large clusters
   - Ten years ago, only Google could afford 1000 servers;
   - Today you can rent this from Amazon Web Services (AWS)

Jeff Dean, SOCC’2010:

Numbers Everyone Should Know

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K w/cheap compression algorithm	3,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

Google

Memory access

Communication

Jeff Dean, SOCC'2010:

Numbers Everyone Should Know

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K w/cheap compression algorithm	3,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

Google

Memory access

Communication

Jeff Dean, SOCC'2010:

Numbers Everyone Should Know

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K w/cheap compression algorithm	3,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

Google

Memory access

Communication

Jeff Dean, SOCC'2010:

Numbers Everyone Should Know

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K w/cheap compression algorithm	3,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

Google

Memory access

Communication

Jeff Dean, SOCC'2010:

Numbers Everyone Should Know

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K w/cheap compression algorithm	3,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

Google

Memory access

Local access is significantly faster than communication

Communication

Parallel DBMSs

- Goal
  - Improve performance by executing multiple operations in parallel

- Key benefit
  - Cheaper to scale than relying on a single increasingly more powerful processor

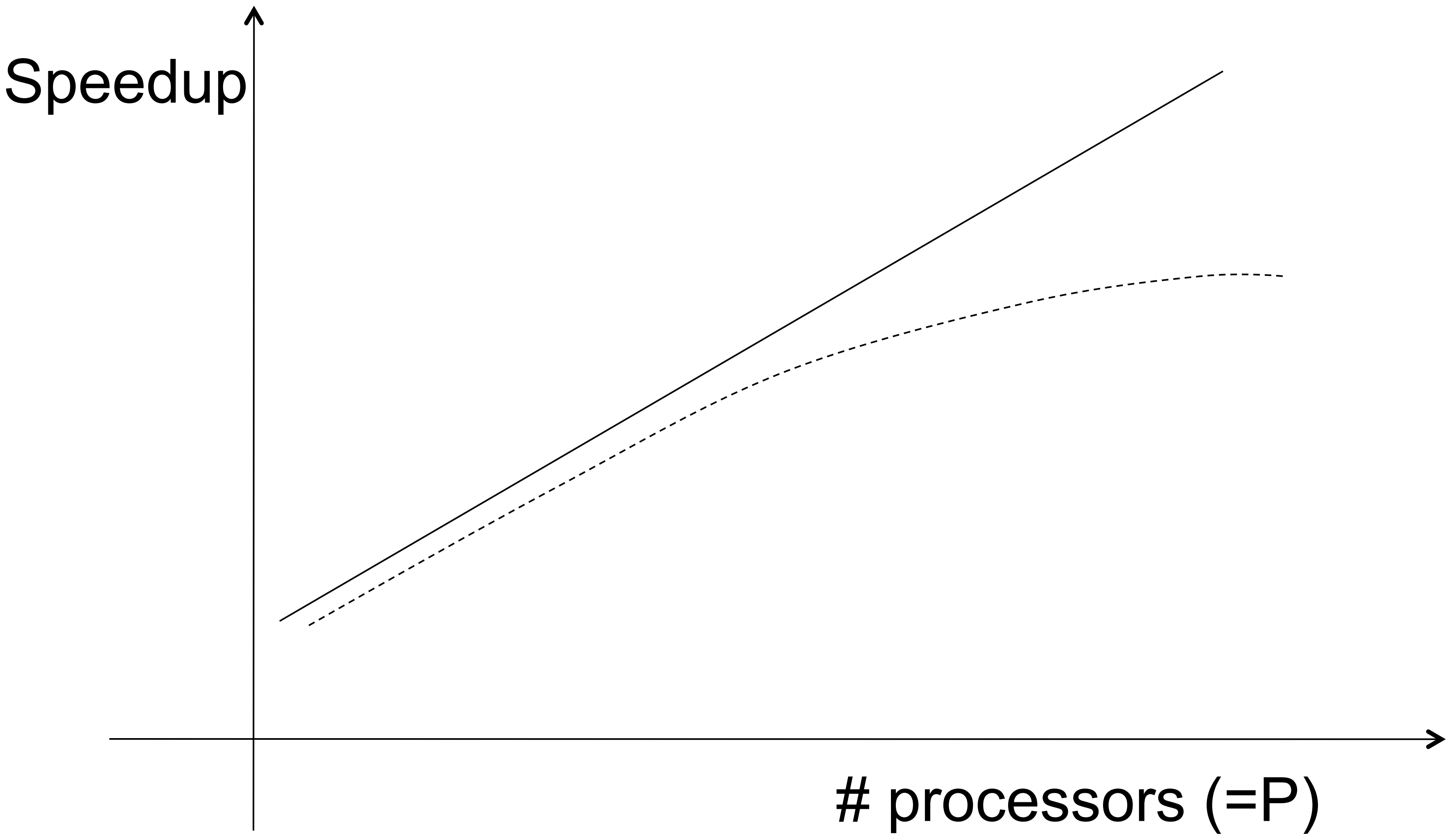
- Key challenge
  - Ensure overhead and contention do not kill performance

Performance Metrics for Parallel DBMSs

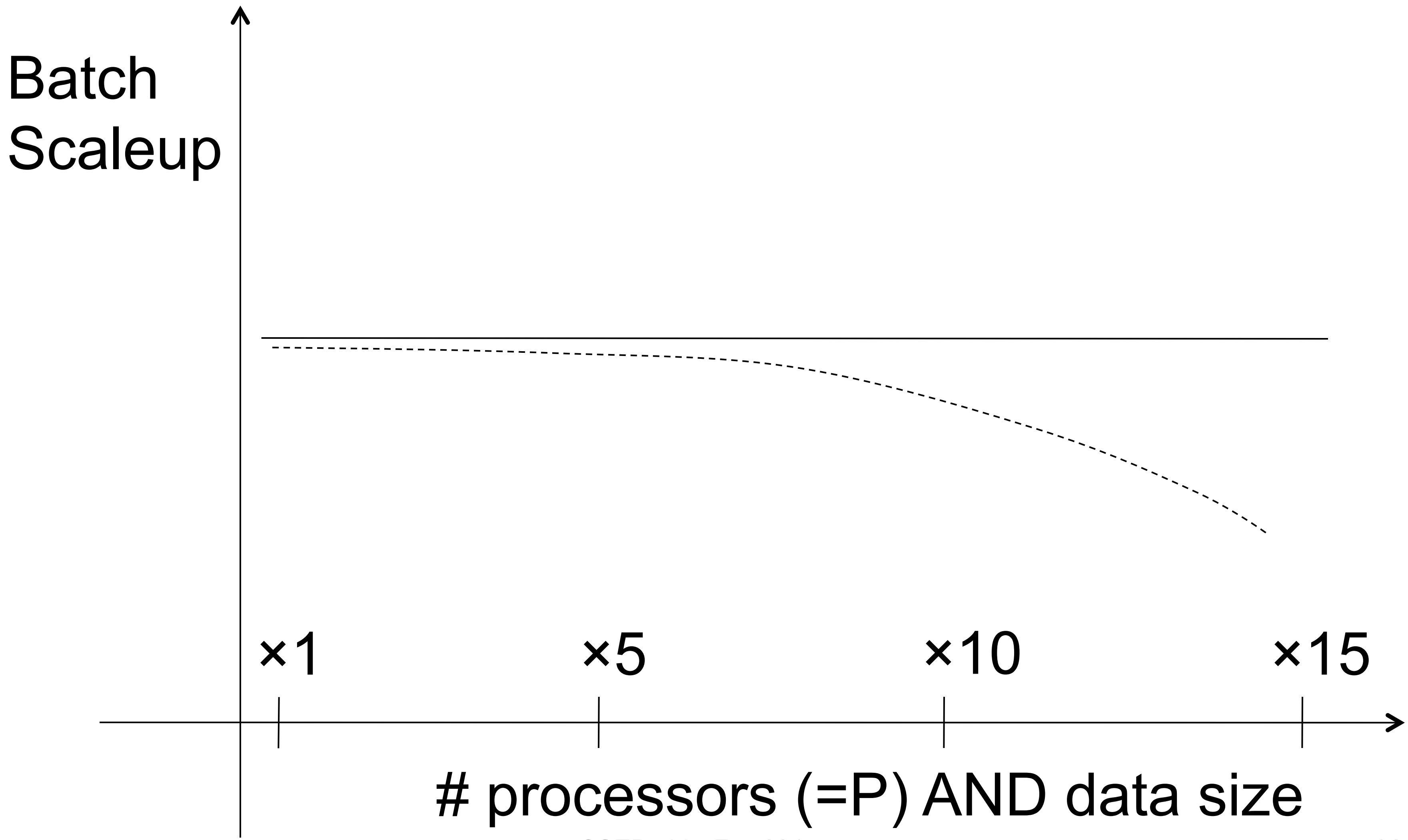
- Speedup
  - More processors → higher speed
  - Individual queries should run faster
  - Should do more transactions per second (TPS)

- Scaleup
  - More processors → can process more data
  - Batch scaleup
    - Same query on larger input data should take the same time
  - Transaction scaleup
    - N-times as many TPS on N-times larger database
    - But each transaction typically remains small

Linear v.s. Non-linear Speedup



Linear v.s. Non-linear Scaleup



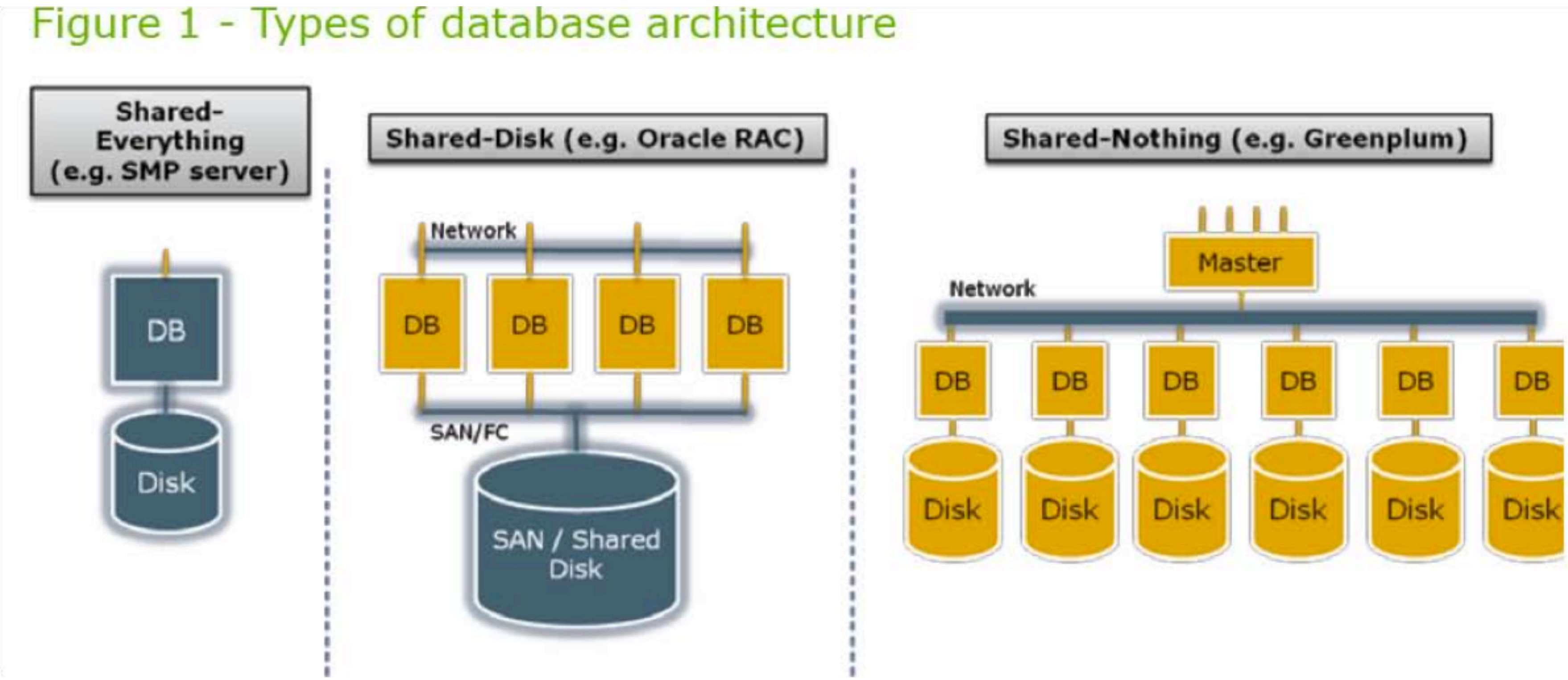
Challenges to Linear Speedup and Scaleup

- Startup cost
  - Cost of starting an operation on many processors
- Interference
  - Contention for resources between processors
- Skew
  - Slowest processor becomes the bottleneck

Architectures for Parallel Databases

- Shared memory
- Shared disk
- Shared nothing

Architectures for Parallel Databases



From: Greenplum Database Whitepaper

Shared Memory

- Nodes share both RAM and disk
- Dozens to hundreds of processors

Example: SQL Server runs on a single machine and can leverage many threads to get a query to run faster (see query plans)

- Easy to use and program
- But very expensive to scale: last remaining cash cows in the hardware industry

Shared Disk

- All nodes access the same disks
- Found in the largest "single-box" (non-cluster) multiprocessors

Oracle dominates this class of systems.

Characteristics:

- Also hard to scale past a certain point: existing deployments typically have fewer than 10 machines

Shared Nothing

- Cluster of machines on high-speed network
- Called "clusters" or "blade servers”
- Each machine has its own memory and disk: lowest contention.

NOTE: Because all machines today have many cores and many disks, then shared-nothing systems typically run many "nodes” on a single physical machine.

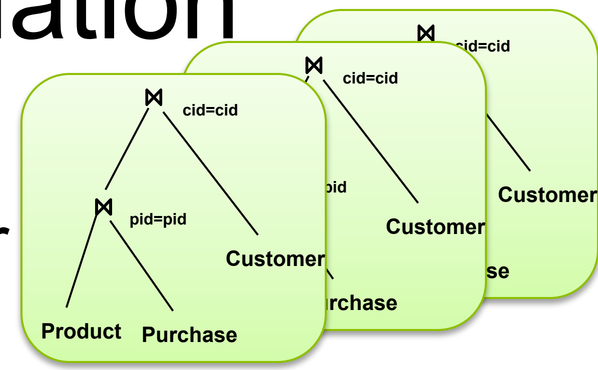
Characteristics:

- Today, this is the most scalable architecture.
- Most difficult to administer and tune.

We discuss only Shared Nothing in class

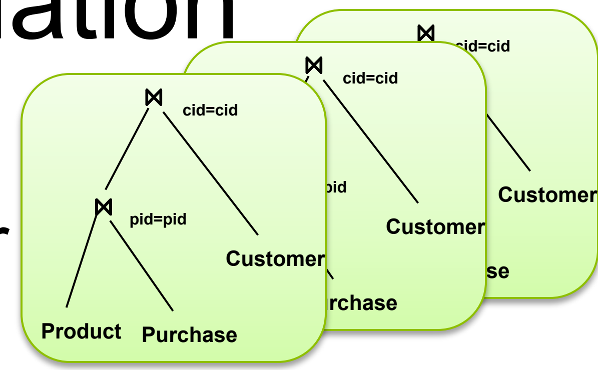
Taxonomy for Parallel Query Evaluation

- Inter-query parallelism
  - Each query runs on one processor

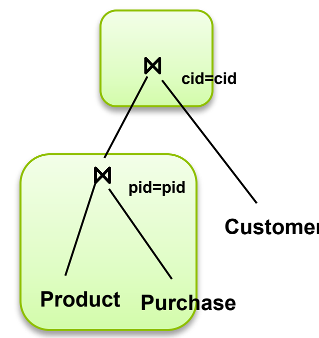


Taxonomy for Parallel Query Evaluation

- Inter-query parallelism
  - Each query runs on one processor



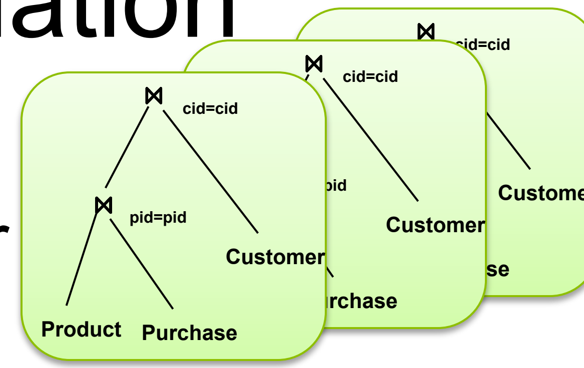
- Inter-operator parallelism
  - A query runs on multiple processors
  - An operator runs on one processor



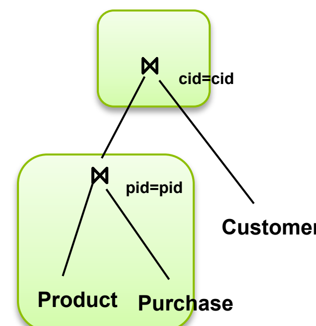
-
  -

Taxonomy for Parallel Query Evaluation

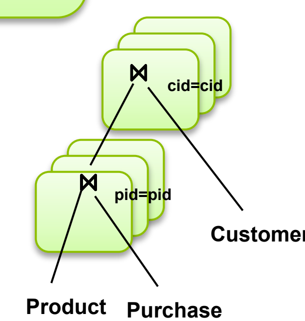
- Inter-query parallelism
  - Each query runs on one processor



- Inter-operator parallelism
  - A query runs on multiple processors
  - An operator runs on one processor

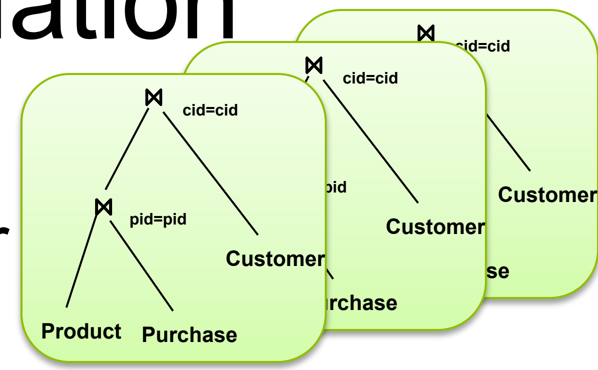


- Intra-operator parallelism
  - An operator runs on multiple processors

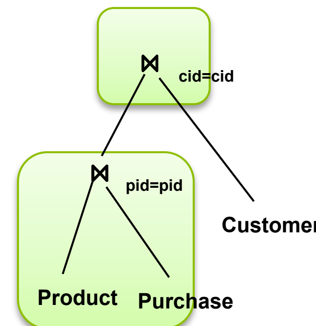


Taxonomy for Parallel Query Evaluation

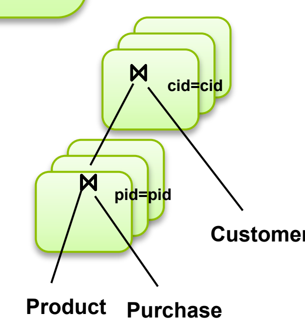
- Inter-query parallelism
  - Each query runs on one processor



- Inter-operator parallelism
  - A query runs on multiple processors
  - An operator runs on one processor



- Intra-operator parallelism
  - An operator runs on multiple processors



We study only intra-operator parallelism: most scalable

Parallel Query Processing

How do we compute these operations on a shared-nothing parallel db?

- Selection: $\sigma_{A=123}(R)$ (that's easy, won't discuss…)
- Group-by: $\gamma_{A,sum(B)}(R)$
- Join: $R \bowtie S$

Before we answer that: how do we store R (and S) on a shared-nothing parallel db?

Horizontal Data Partitioning

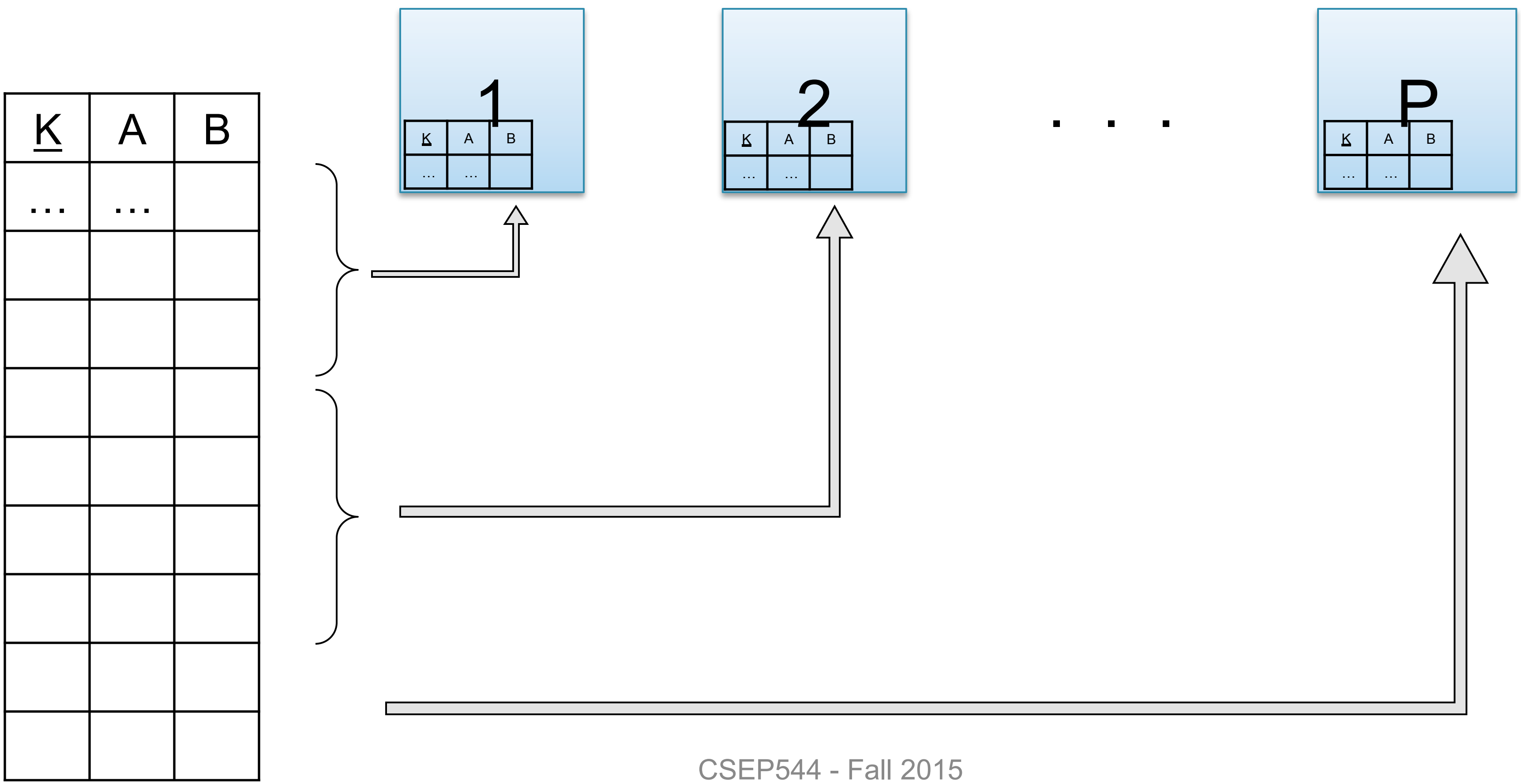
Data:

K	A	B
…	…	

Servers:

Horizontal Data Partitioning

Data: Servers:

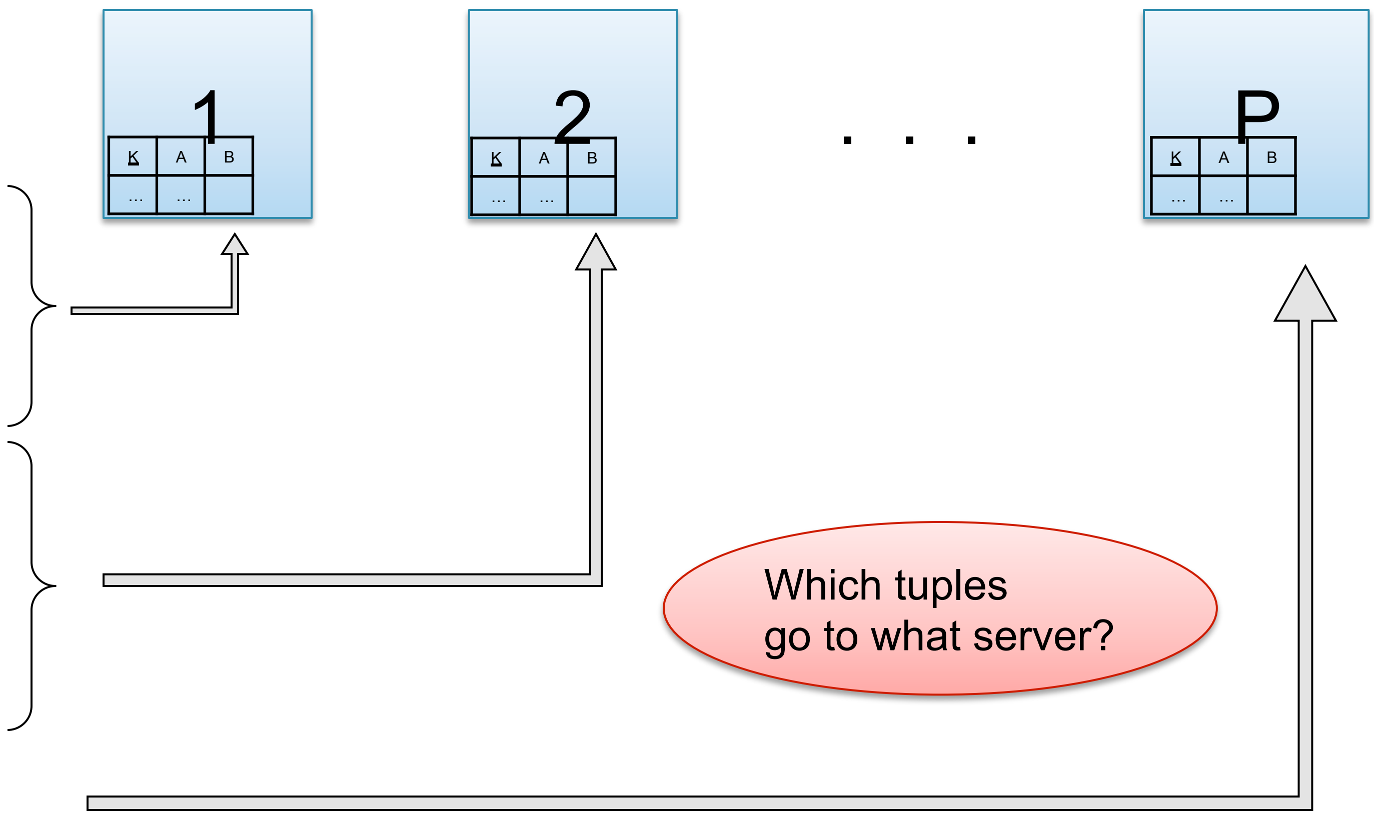


Horizontal Data Partitioning

Data:

K	A	B
…	…	

Servers:



Horizontal Data Partitioning

- Block Partition:
  - Partition tuples arbitrarily s.t. size(R_1)≈ … ≈ size(R_P)
- Hash partitioned on attribute A:
  - Tuple t goes to chunk i, where i = h(t.A) mod P + 1
- Range partitioned on attribute A:
  - Partition the range of A into $-\infty = v_0 < v_1 < \ldots < v_P = \infty$
  - Tuple t goes to chunk i, if $v_{i-1} < t.A < v_i$

Basic Parallel GroupBy

Data: R(K,A,B,C)
Query: $\gamma_{A,sum(C)}(R)$
Discuss in class how to compute in each case:

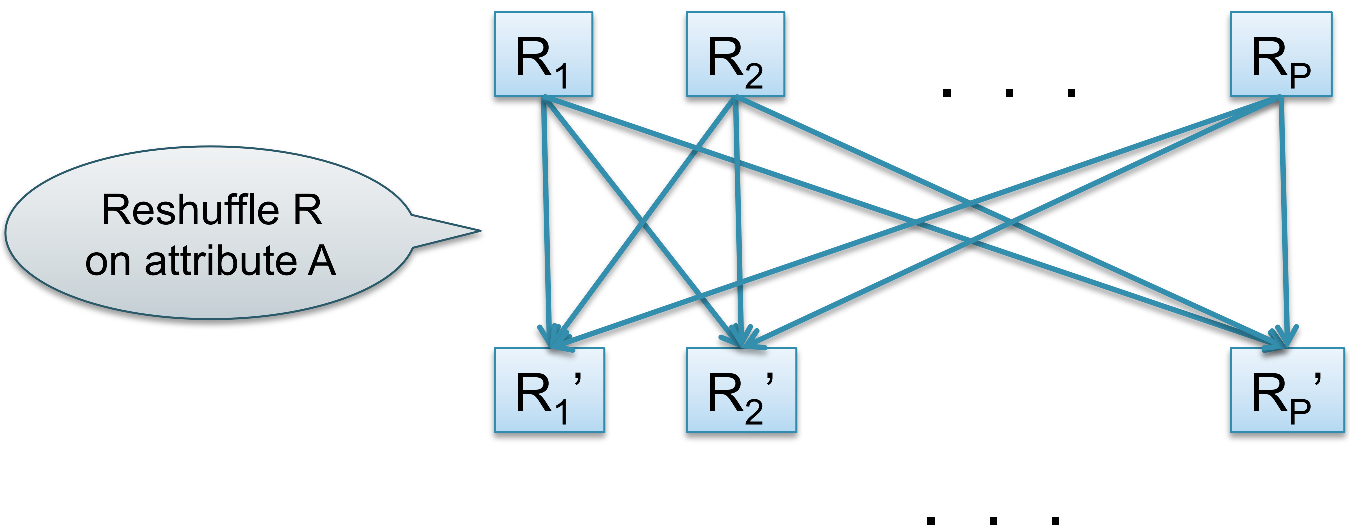
- R is hash-partitioned on A
- R is block-partitioned
- R is hash-partitioned on K

Basic Parallel GroupBy

Data: R(<u>K</u>,A,B,C)

Query: $\gamma_{A,sum(C)}(R)$

- R is block-partitioned or hash-partitioned on K



Basic Parallel Join

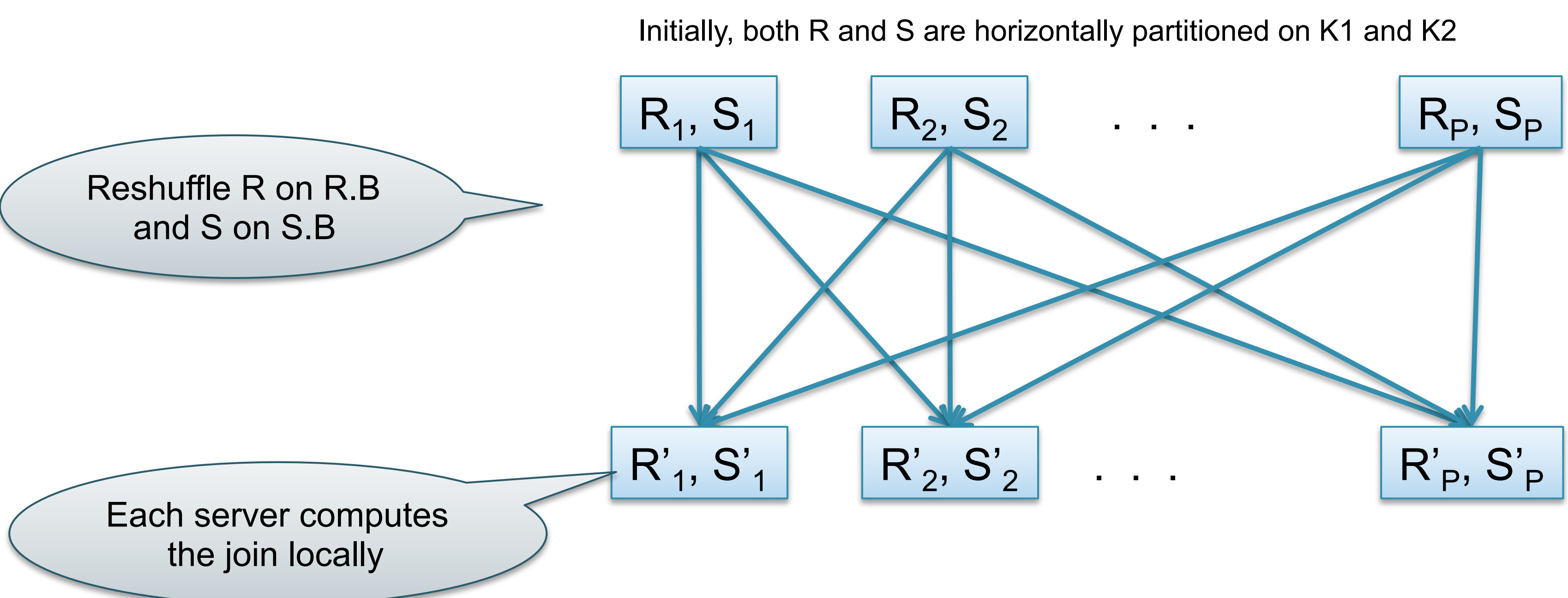
- **Data**: $R(\underline{K1}, A, B)$, $S(\underline{K2}, B, C)$
- **Query**: $R(\underline{K1},A,B) \bowtie S(\underline{K2},B,C)$

Initially, both R and S are horizontally partitioned on K1 and K2

R_1, S_1 R_2, S_2 R_P, S_P

Basic Parallel Join

- Data: R(<u>K1</u>,A, B), S(<u>K2</u>, B, C)
- Query: R(<u>K1</u>,A,B) ⋈ S(<u>K2</u>,B,C)



Speedup and Scaleup

- Consider:
  - Query: $\gamma_{A,sum(C)}(R)$
  - Runtime: dominated by reading chunks from disk
- If we double the number of nodes P, what is the new running time?

- If we double both P and the size of R, what is the new running time?

Speedup and Scaleup

- Consider:
  - Query: $\gamma_{A,sum(C)}(R)$
  - Runtime: dominated by reading chunks from disk
- If we double the number of nodes P, what is the new running time?
  - Half (each server holds ½ as many chunks)
- If we double both P and the size of R, what is the new running time?
  - Same (each server holds the same # of chunks)

Uniform Data v.s. Skewed Data

- Let R(K,A,B,C); which of the following partition methods may result in skewed partitions?

- Block partition

- Hash-partition
  - On the key K
  - On the attribute A

Uniform Data v.s. Skewed Data

- Let R(K,A,B,C); which of the following partition methods may result in skewed partitions?

- Block partition

Uniform

Assuming good hash function

- Hash-partition
  - On the key K
  - On the attribute A

Uniform

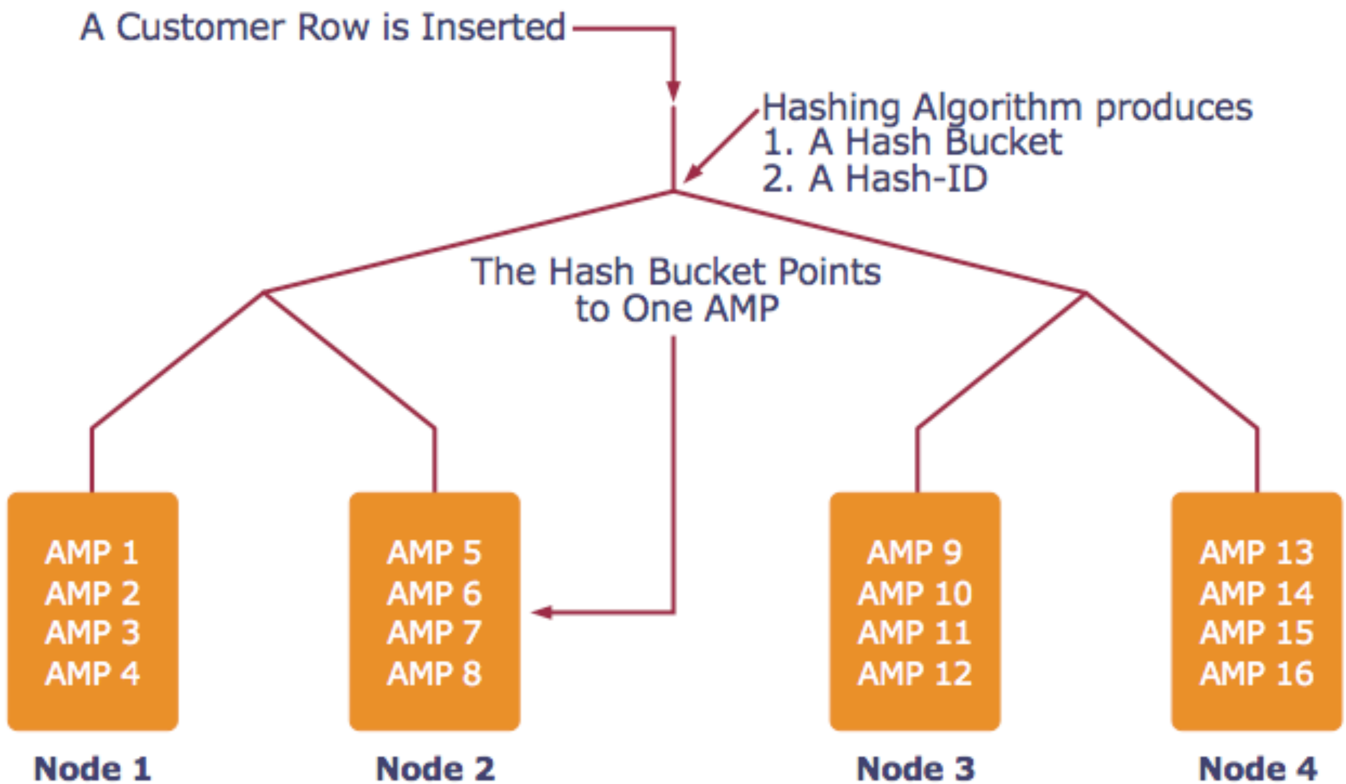
May be skewed

E.g. when all records have the same value of the attribute A, then all records end up in the same partition

Parallel DBMS

- Parallel query plan: tree of parallel operators
  Intra-operator parallelism
  - Data streams from one operator to the next
  - Typically all cluster nodes process all operators
- Can run multiple queries at the same time
  Inter-query parallelism
  - Queries will share the nodes in the cluster
- Notice that user does not need to know how his/her SQL query was processed

Example: Teradata – Loading

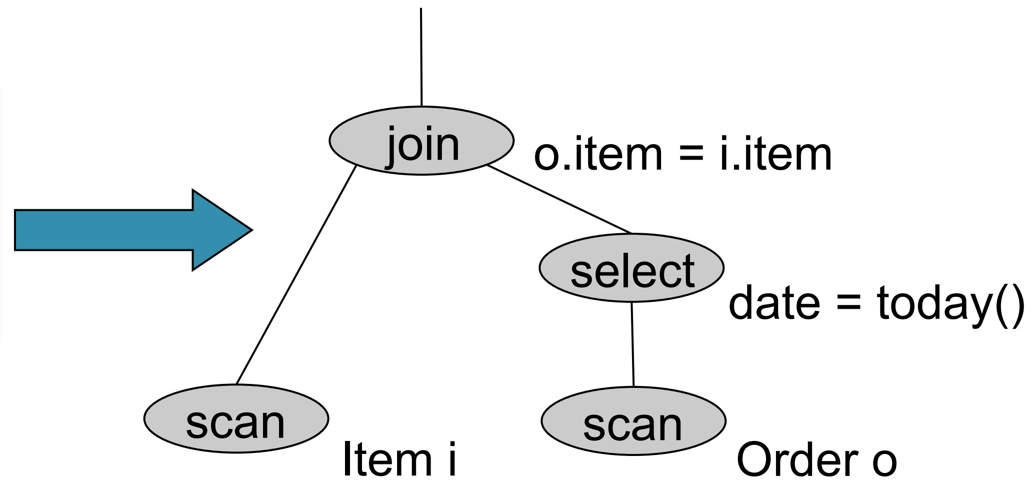


AMP = "Access Module Processor" = unit of parallelism

Example: Teradata – Query Execution

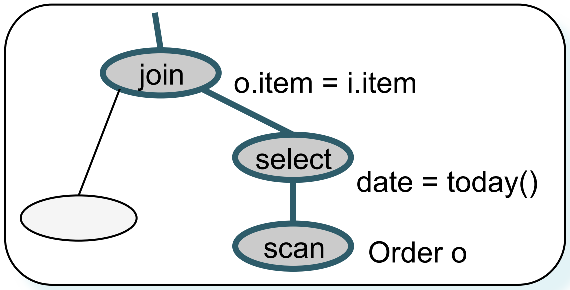
Find all orders from today, along with the items ordered

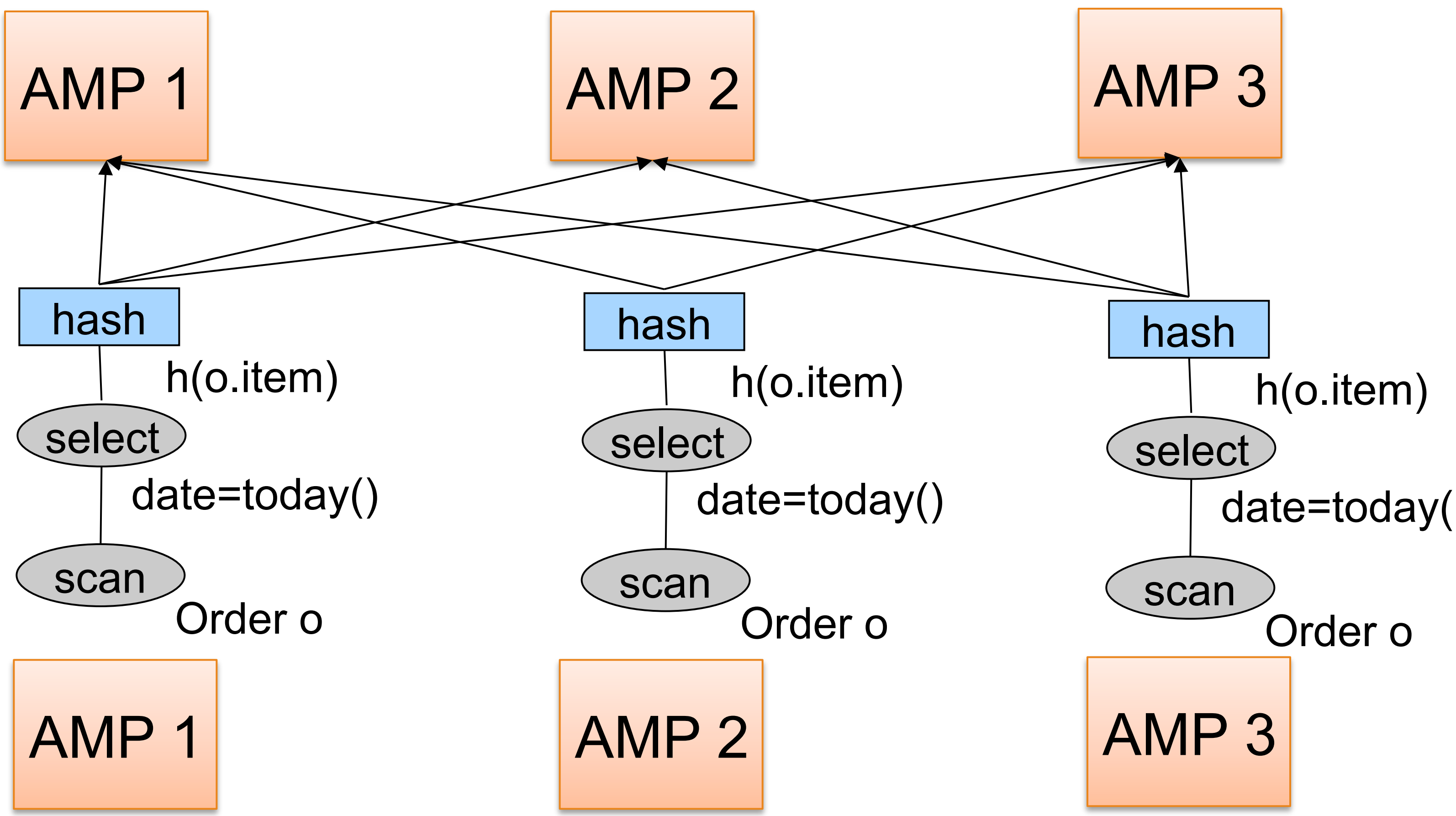
```
SELECT *
  FROM Order o, Line i
 WHERE o.item = i.item
   AND o.date = today()
```



Order(oid, item, date), Line(item, …)

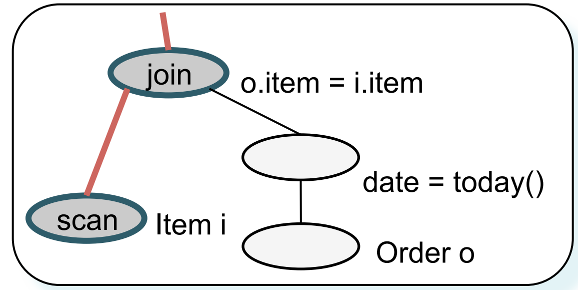
Query Execution

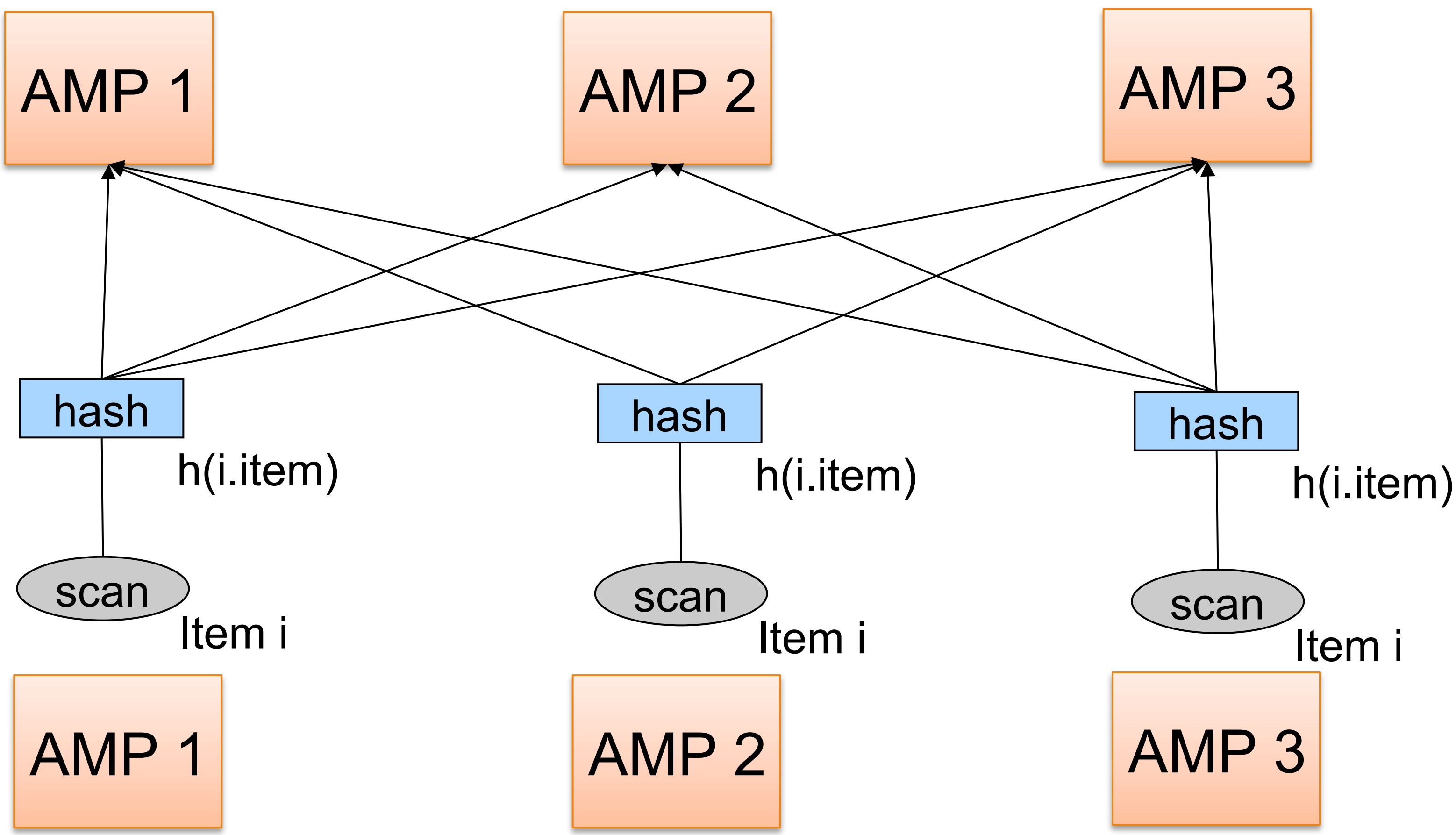




Order(oid, item, date), Line(item, …)

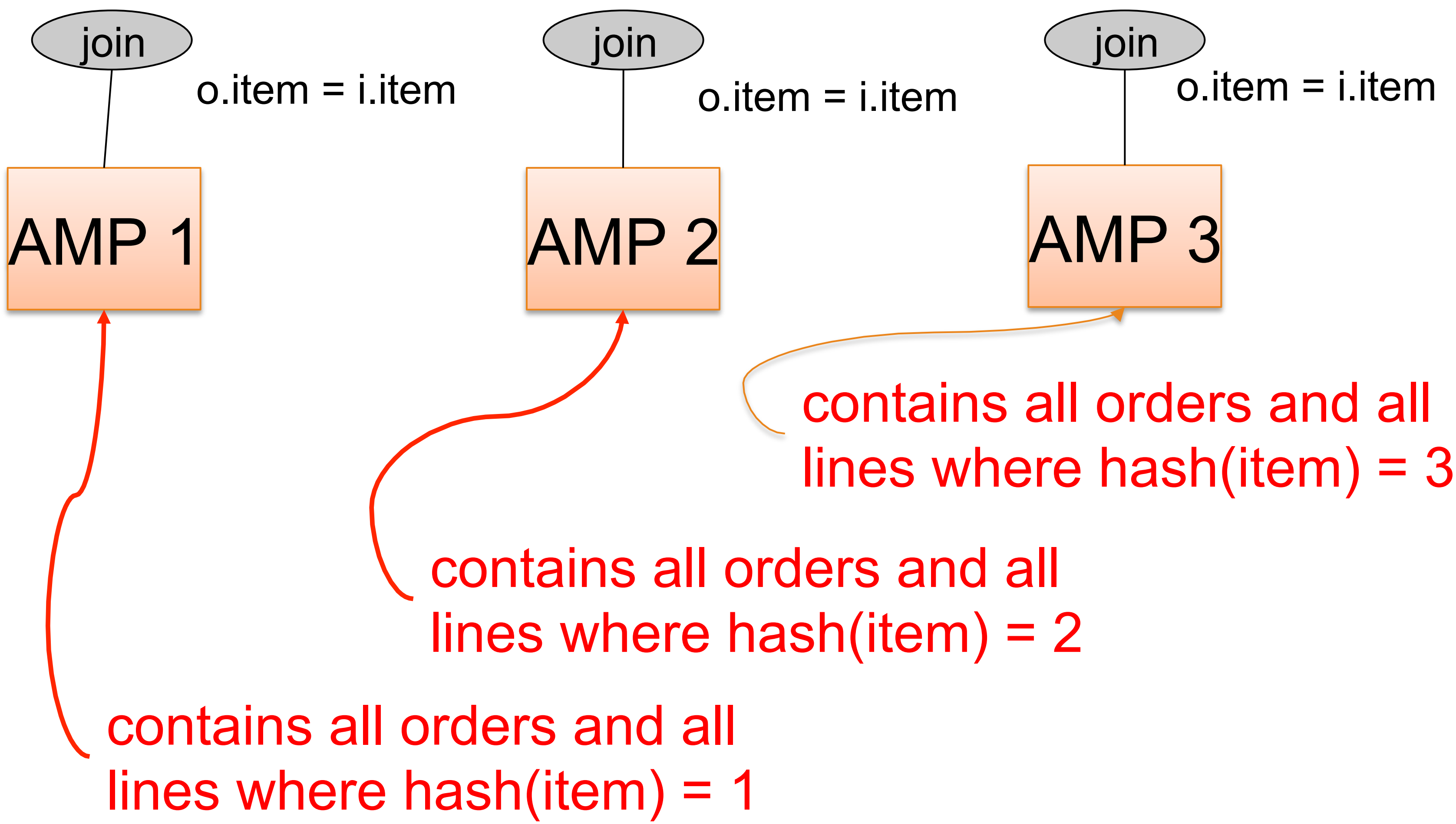
Query Execution





Order(oid, item, date), Line(item, …)

Query Execution



Overview of Today's Lecture

- Query Execution/Optimization
- Parallel databases
- Map/Reduce
  - Next week: MR paper review
- Not in class: PigLatin
  - Read for HW3

Cluster Computing

Cluster Computing

- Large number of commodity servers, connected by high speed, commodity network
- Rack: holds a small number of servers
- Data center: holds many racks

Cluster Computing

- Massive parallelism:
  - 100s, or 1000s, or 10000s servers
  - Many hours
- Failure:
  - If medium-time-between-failure is 1 year
  - Then 10000 servers have one failure / hour

Distributed File System (DFS)

- For very large files: TBs, PBs
- Each file is partitioned into *chunks*, typically 64MB
- Each chunk is replicated several times ($\geq$3), on different racks, for fault tolerance
- Implementations:
  - Google's DFS: GFS, proprietary
  - Hadoop's DFS: HDFS, open source

Map Reduce

- Google: paper published 2004
- Free variant: Hadoop

- Map-reduce = high-level programming model and implementation for large-scale parallel data processing

Data Model

Files !

A file = a bag of **`(key, value)`** pairs

A MapReduce program:

- Input: a bag of **`(inputkey, value)`** pairs
- Output: a bag of **`(outputkey, value)`** pairs

Step 1: the MAP Phase

User provides the MAP-function:

- Input: `(input key, value)`
- Ouput:
  bag of `(intermediate key, value)`

System applies the map function in parallel to all `(input key, value)` pairs in the input file

Step 2: the REDUCE Phase

User provides the REDUCE function:

- Input: `(intermediate key, bag of values)`
- Output: bag of output `(values)`

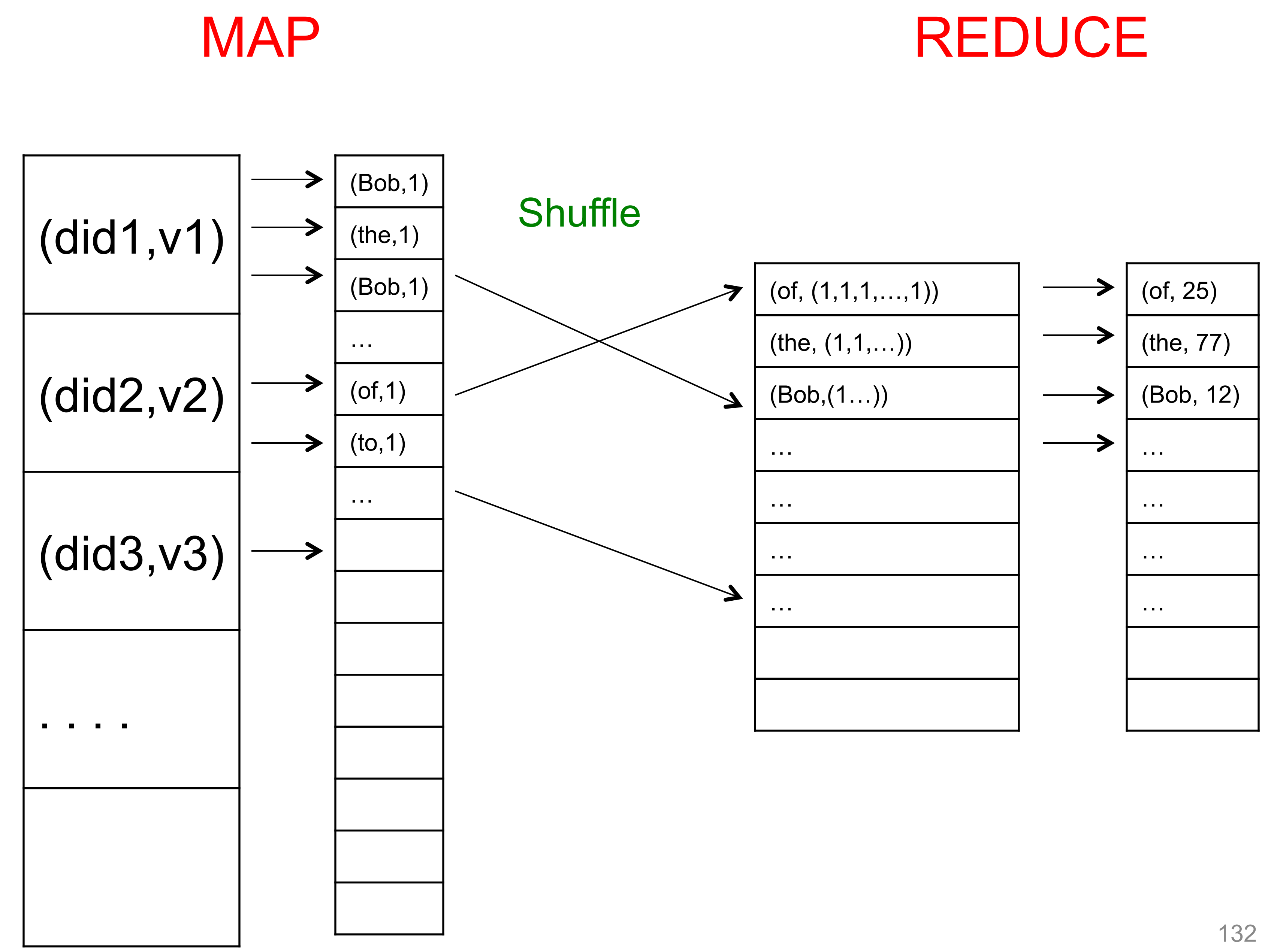
System groups all pairs with the same intermediate key, and passes the bag of values to the REDUCE function

Example

- Counting the number of occurrences of each word in a large collection of documents
- Each Document
  - The key = document id (did)
  - The value = set of words (word)

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

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

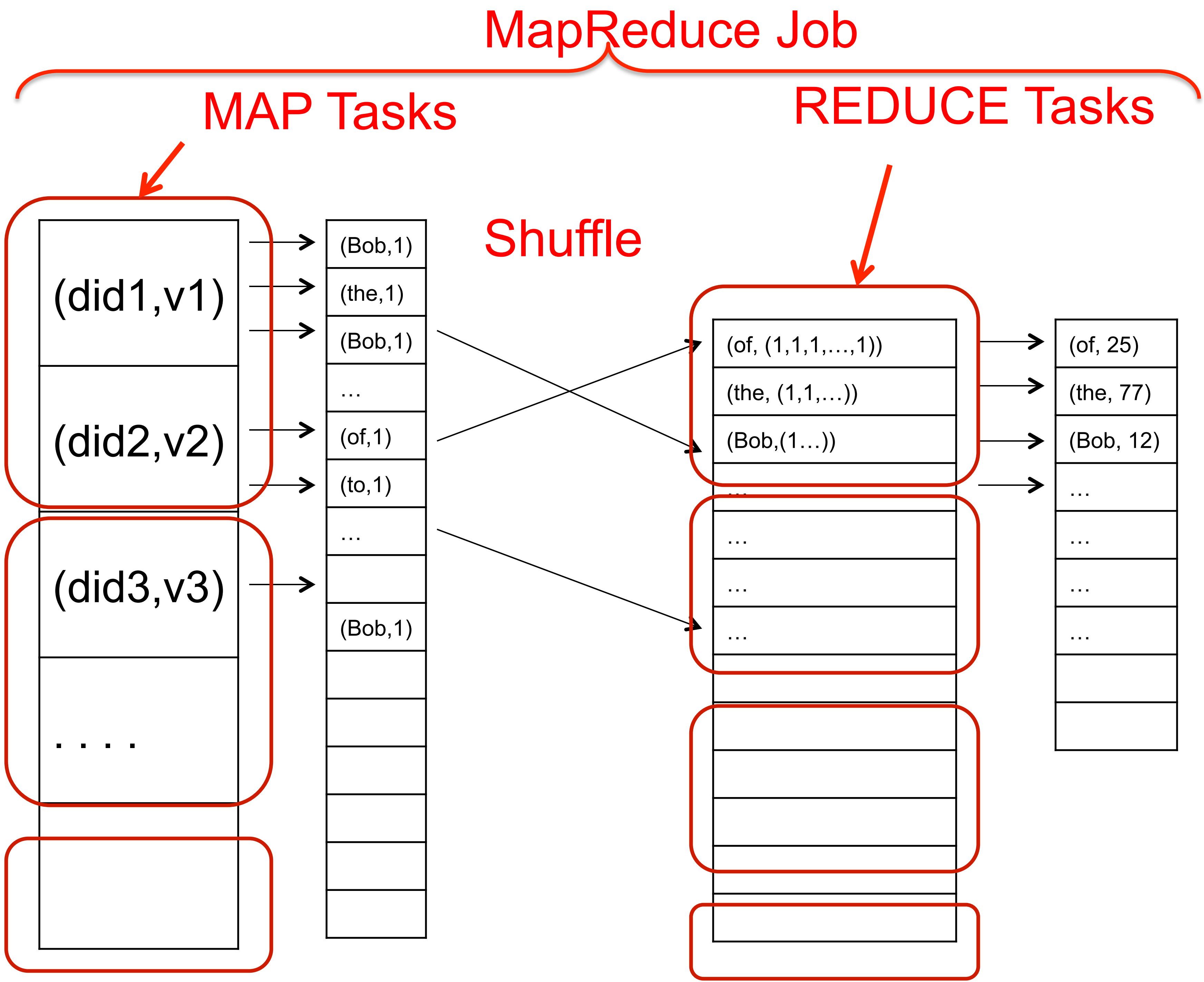

MAP
REDUCE
(did1,v1)
(did2,v2)
(did3,v3)
. . . .
(Bob,1)
(the,1)
(Bob,1)
…
(of,1)
(to,1)
…
Shuffle
(of, (1,1,1,…,1))
(the, (1,1,…))
(Bob,(1…))
…
…
…
…
(of, 25)
(the, 77)
(Bob, 12)
…
…
…
…

Jobs v.s. Tasks

- A MapReduce Job
  - One single “query”, e.g. count the words in all docs
  - More complex queries may consists of multiple jobs
- A Map Task, or a Reduce Task
  - A group of instantiations of the map-, or reduce-function, which are scheduled on a single worker

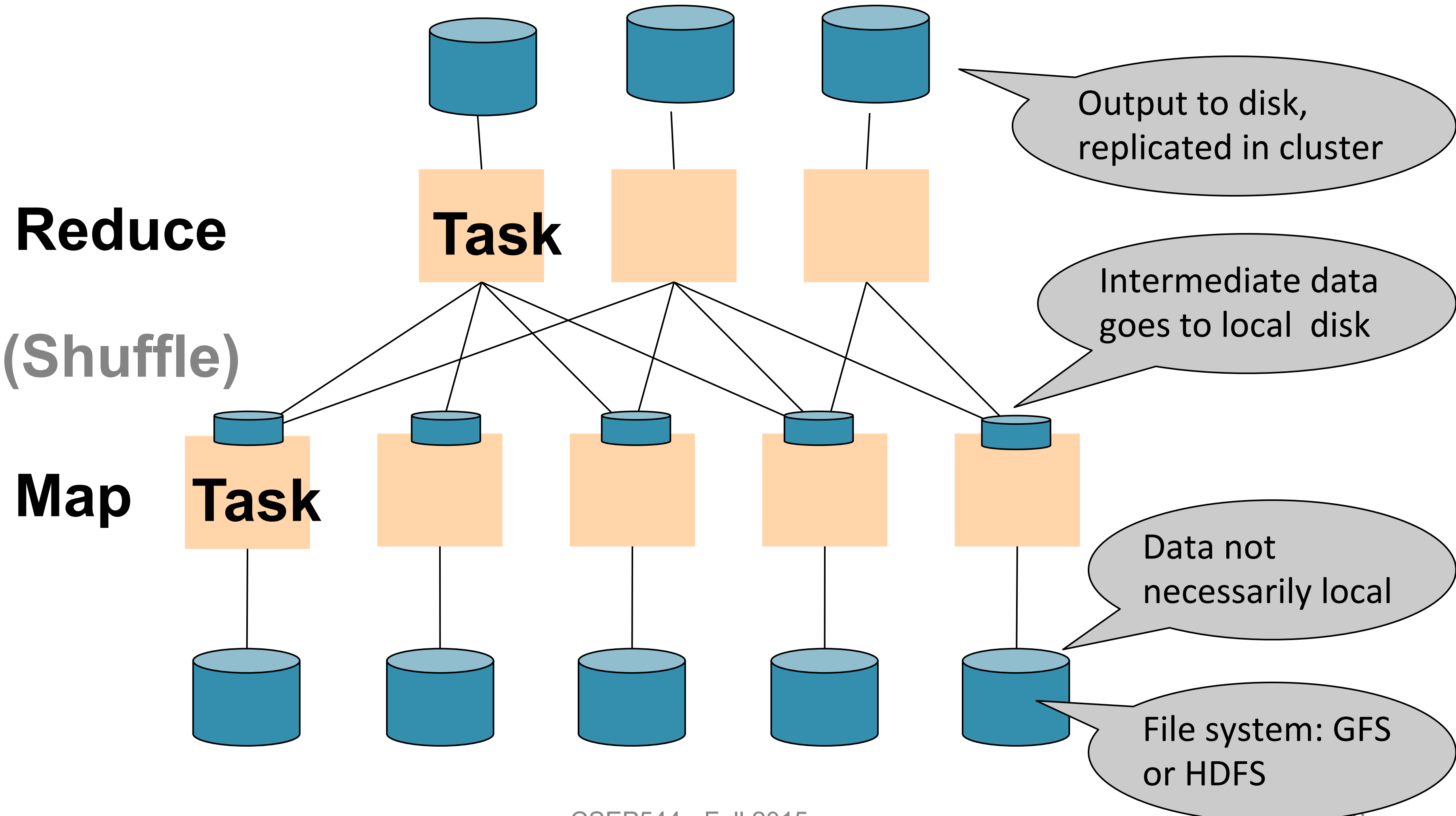
Workers

- A worker is a process that executes one task at a time
- Typically there is one worker per processor, hence 4 or 8 per node



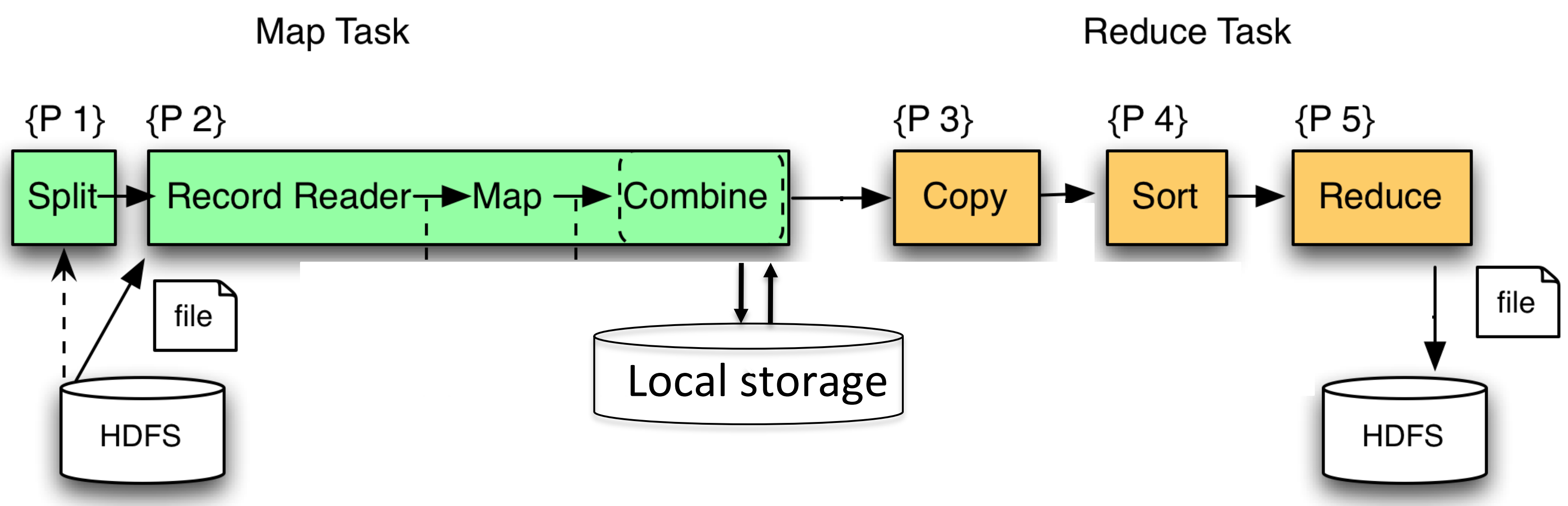
MapReduce Job
MAP Tasks
REDUCE Tasks
Shuffle
(did1,v1)
(did2,v2)
(did3,v3)
. . . . .
(Bob,1)
(the,1)
(Bob,1)
…
(of,1)
(to,1)
…
(Bob,1)
(of, (1,1,1,…,1))
(the, (1,1,…))
(Bob,(1…))
…
…
…
…
(of, 25)
(the, 77)
(Bob, 12)
…
…
…
…

MapReduce Execution Details



MR Phases

- Each Map and Reduce task has multiple phases:



Implementation

- There is one master node
- Master partitions input file into *M splits*, by key
- Master assigns *workers* (=servers) to the *M map tasks*, keeps track of their progress
- Workers write their output to local disk, partition into *R regions*
- Master assigns workers to the *R reduce tasks*
- Reduce workers read regions from the map workers' local disks

Interesting Implementation Details

Worker failure:

- Master pings workers periodically,
- If down then reassigns the task to another worker

Interesting Implementation Details

Backup tasks:

- *Straggler* = a machine that takes unusually long time to complete one of the last tasks. Eg:
  - Bad disk forces frequent correctable errors (30MB/s → 1MB/s)
  - The cluster scheduler has scheduled other tasks on that machine
- Stragglers are a main reason for slowdown
- Solution*: pre-emptive backup execution of the last few remaining in-progress tasks*

MapReduce Summary

- Hides scheduling and parallelization details
- However, very limited queries
  - Difficult to write more complex queries
  - Need multiple MapReduce jobs
- Solution: declarative query language

Declarative Languages on MR

- PIG Latin (Yahoo!)
  – New language, like Relational Algebra
  – Open source
- HiveQL (Facebook)
  – SQL-like language
  – Open source
- SQL / Dremmel / Tenzing (Google)
  – BigQuery – SQL in the cloud

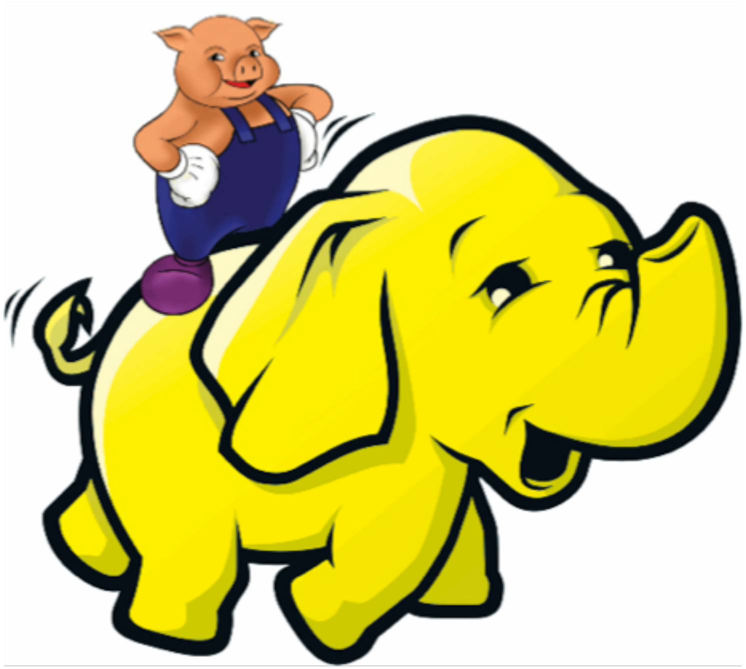
Overview of Today's Lecture

- Query Execution/Optimization
- Parallel databases
- Map/Reduce
  - Next week: MR paper review
- Not in class: PigLatin
  - Read for HW3

Pig Latin – Reference only (will not discuss in class)

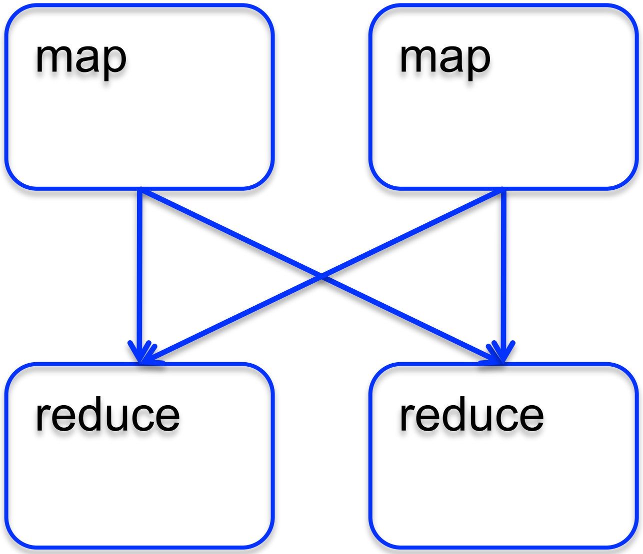
What is Pig?

- An engine for executing programs on top of Hadoop
- It provides a language, Pig Latin, to specify these programs
- An Apache open source project
  http://hadoop.apache.org/pig/



Credit: Alan Gates, Yahoo!

Map Reduce Illustrated

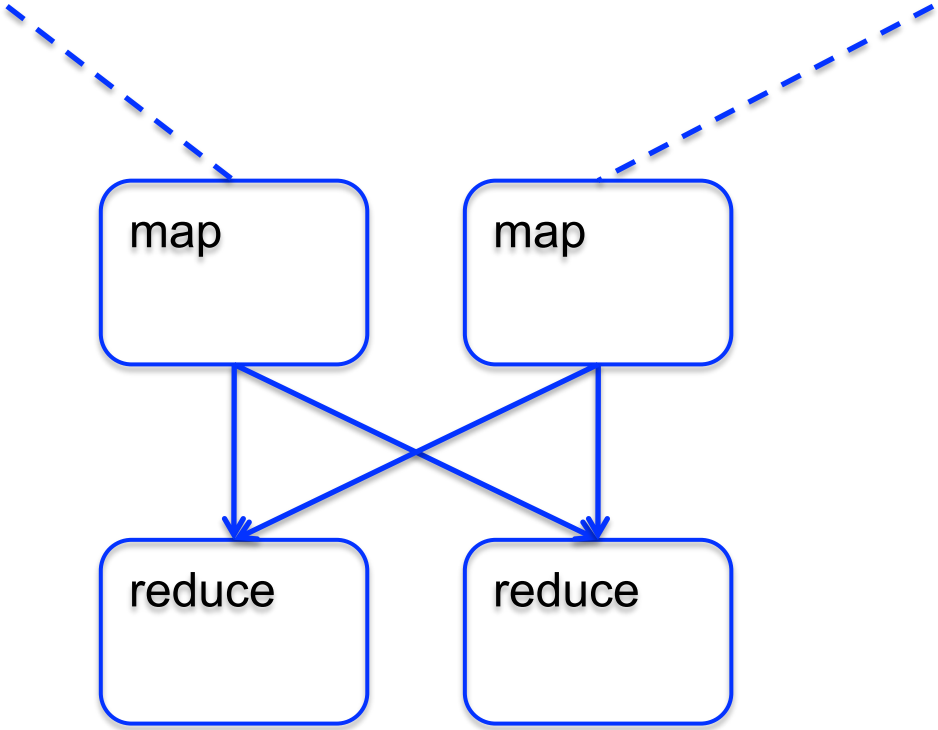


Credit: Alan Gates, Yahoo!

Map Reduce Illustrated

Romeo, Romeo, wherefore art thou Romeo?

What, art thou hurt?



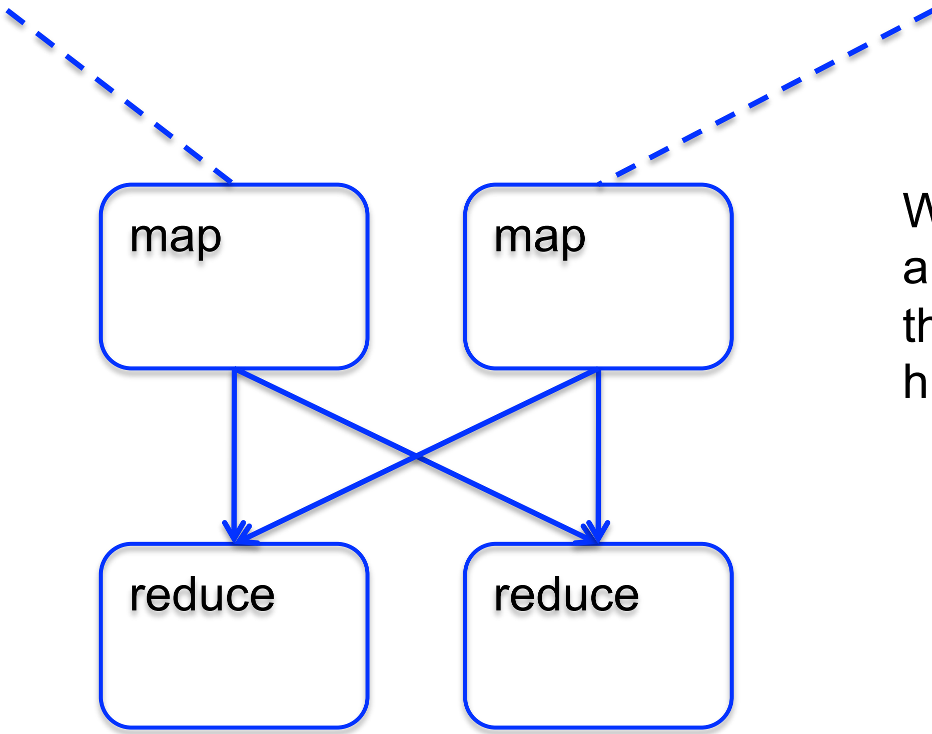
Credit: Alan Gates, Yahoo!

Map Reduce Illustrated

Romeo, Romeo, wherefore art thou Romeo?

What, art thou hurt?

Romeo, 1
Romeo, 1
wherefore, 1
art, 1
thou, 1
Romeo, 1



What, 1
art, 1
thou, 1
hurt, 1

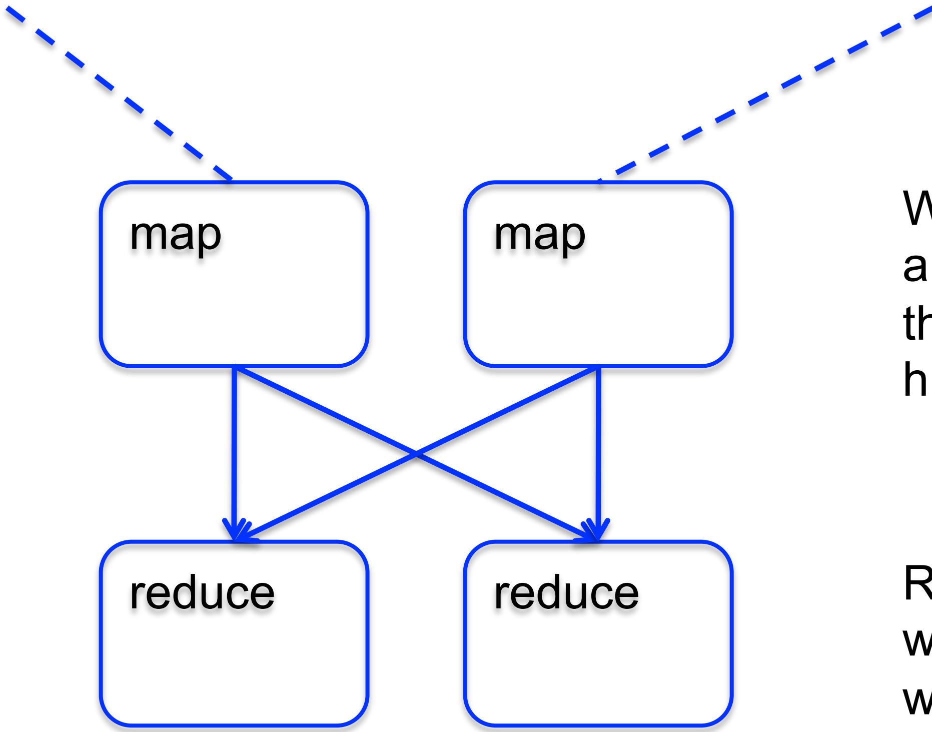
Credit: Alan Gates, Yahoo!

Map Reduce Illustrated

Romeo, Romeo, wherefore art thou Romeo?

What, art thou hurt?

Romeo, 1
Romeo, 1
wherefore, 1
art, 1
thou, 1
Romeo, 1



What, 1
art, 1
thou, 1
hurt, 1

art, (1, 1)
hurt (1),
thou (1, 1)

Romeo, (1, 1, 1)
wherefore, (1)
what, (1)

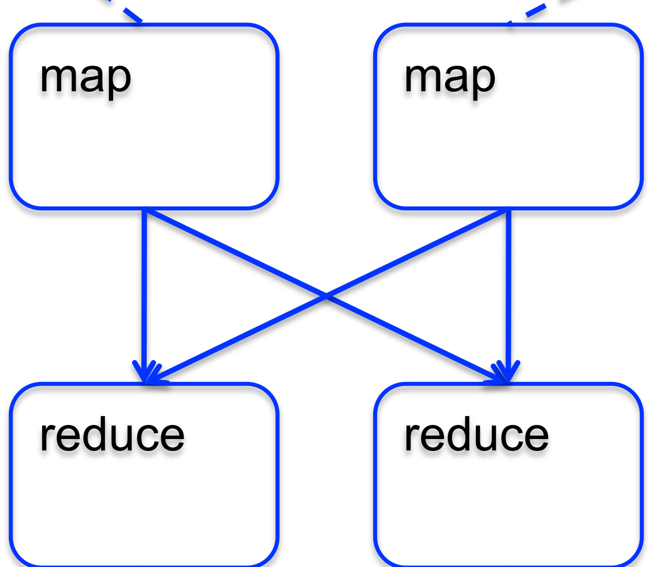
Map Reduce Illustrated

Romeo, Romeo, wherefore art thou Romeo?

What, art thou hurt?

Romeo, 1
Romeo, 1
wherefore, 1
art, 1
thou, 1
Romeo, 1

What, 1
art, 1
thou, 1
hurt, 1



art, (1, 1)
hurt (1),
thou (1, 1)

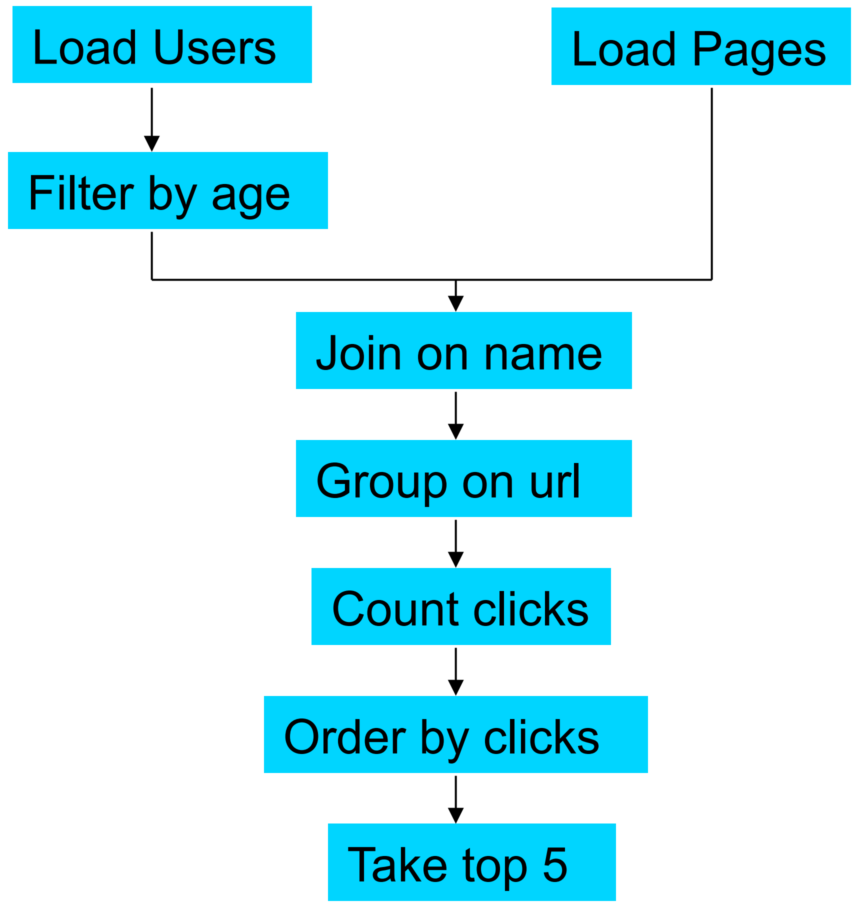
Romeo, (1, 1, 1)
wherefore, (1)
what, (1)

art, 2
hurt, 1
thou, 2

Romeo, 3
wherefore, 1
what, 1

Why use Pig?

Suppose you have user data in one file, website data in another, and you need to find the top 5 most visited sites by users aged 18 - 25.



In Map-Reduce

```java
import java.io.IOException;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.Writable;
import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.KeyValueTextInputFormat;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.RecordReader;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.mapred.SequenceFileInputFormat;
import org.apache.hadoop.mapred.SequenceFileOutputFormat;
import org.apache.hadoop.mapred.TextInputFormat;
import org.apache.hadoop.mapred.jobcontrol.Job;
import org.apache.hadoop.mapred.jobcontrol.JobControl;
import org.apache.hadoop.mapred.lib.IdentityMapper;

public class MRExample {
    public static class LoadPages extends MapReduceBase
        implements Mapper<LongWritable, Text, Text, Text> {

        public void map(LongWritable k, Text val,
                OutputCollector<Text, Text> oc,
                Reporter reporter) throws IOException {
            // Pull the key out
            String line = val.toString();
            int firstComma = line.indexOf(',');
            String key = line.substring(0, firstComma);
            String value = line.substring(firstComma + 1);
            Text outKey = new Text(key);
            // Prepend an index to the value so we know which file
            // it came from.
            Text outVal = new Text("1" + value);
            oc.collect(outKey, outVal);
        }
    }
    public static class LoadAndFilterUsers extends MapReduceBase
        implements Mapper<LongWritable, Text, Text, Text> {

        public void map(LongWritable k, Text val,
                OutputCollector<Text, Text> oc,
                Reporter reporter) throws IOException {
            // Pull the key out
            String line = val.toString();
            int firstComma = line.indexOf(',');
            String value = line.substring(firstComma + 1);
            int age = Integer.parseInt(value);
            if (age < 18 || age > 25) return;
            String key = line.substring(0, firstComma);
            Text outKey = new Text(key);
            // Prepend an index to the value so we know which file
            // it came from.
            Text outVal = new Text("2" + value);
            oc.collect(outKey, outVal);
        }
    }
    public static class Join extends MapReduceBase
        implements Reducer<Text, Text, Text, Text> {

        public void reduce(Text key,
                Iterator<Text> iter,
                OutputCollector<Text, Text> oc,
                Reporter reporter) throws IOException {
           // For each value, figure out which file it's from and
store  it
            // accordingly.
            List<String> first = new ArrayList<String>();
            List<String> second = new ArrayList<String>();

            while (iter.hasNext()) {
                Text t = iter.next();
                String value = t.toString();
               if (value.charAt(0) == '1')
first.add(value.substring(1));
                else second.add(value.substring(1));
                reporter.setStatus("OK");
            }

            // Do the cross product and collect the values
            for (String s1 : first) {
                for (String s2 : second) {
                    String outval = key + "," + s1 + "," + s2;
                    oc.collect(null, new Text(outval));
                    reporter.setStatus("OK");
                }
            }
        }
    }
    public static class LoadJoined extends MapReduceBase
        implements Mapper<Text, Text, Text, LongWritable> {

        public void map(
                Text k,
                Text val,
                OutputCollector<Text, LongWritable> oc,
                Reporter reporter) throws IOException {
            // Find the url
            String line = val.toString();
            int firstComma = line.indexOf(',');
            int secondComma = line.indexOf(',', firstComma);
            String key = line.substring(firstComma, secondComma);
            // drop the rest of the record, I don't need it anymore,
            // just pass a 1 for the combiner/reducer to sum instead.
            Text outKey = new Text(key);
            oc.collect(outKey, new LongWritable(1L));
        }
    }
    public static class ReduceUrls extends MapReduceBase
        implements Reducer<Text, LongWritable, WritableComparable,
Writable>  {

        public void reduce(
                Text key,
                Iterator<LongWritable> iter,
                OutputCollector<WritableComparable, Writable> oc,
                Reporter reporter) throws IOException {
            // Add up all the values we see

            long sum = 0;
            while (iter.hasNext()) {
                sum += iter.next().get();
                reporter.setStatus("OK");
            }

            oc.collect(key, new LongWritable(sum));
        }
    }
    public static class LoadClicks extends MapReduceBase
        implements Mapper<WritableComparable, Writable, LongWritable,
Text>  {

        public void map(
                WritableComparable key,
                Writable val,
                OutputCollector<LongWritable, Text> oc,
                Reporter reporter) throws IOException {
            oc.collect((LongWritable)val, (Text)key);
        }
    }
    public static class LimitClicks extends MapReduceBase
        implements Reducer<LongWritable, Text, LongWritable, Text> {

        int count = 0;
        public void reduce(
            LongWritable key,
            Iterator<Text> iter,
            OutputCollector<LongWritable, Text> oc,
            Reporter reporter) throws IOException {

            // Only output the first 100 records
            while (count < 100 && iter.hasNext()) {
                oc.collect(key, iter.next());
                count++;
            }
        }
    }
    public static void main(String[] args) throws IOException {
        JobConf lp = new JobConf(MRExample.class);
        lp.setJobName("Load Pages");
        lp.setInputFormat(TextInputFormat.class);
        lp.setOutputKeyClass(Text.class);
        lp.setOutputValueClass(Text.class);
        lp.setMapperClass(LoadPages.class);
       FileInputFormat.addInputPath(lp, new
Path("/user/gates/pages"));
        FileOutputFormat.setOutputPath(lp,
            new Path("/user/gates/tmp/indexed_pages"));
        lp.setNumReduceTasks(0);
        Job loadPages = new Job(lp);

        JobConf lfu = new JobConf(MRExample.class);
        lfu.setJobName("Load and Filter Users");
        lfu.setInputFormat(TextInputFormat.class);
        lfu.setOutputKeyClass(Text.class);
        lfu.setOutputValueClass(Text.class);
        lfu.setMapperClass(LoadAndFilterUsers.class);
       FileInputFormat.addInputPath(lfu, new
Path("/user/gates/users"));
        FileOutputFormat.setOutputPath(lfu,
            new Path("/user/gates/tmp/filtered_users"));
        lfu.setNumReduceTasks(0);
        Job loadUsers = new Job(lfu);

        JobConf join = new JobConf(MRExample.class);
        join.setJobName("Join Users and Pages");
        join.setInputFormat(KeyValueTextInputFormat.class);
        join.setOutputKeyClass(Text.class);
        join.setOutputValueClass(Text.class);
        join.setMapperClass(IdentityMapper.class);
        join.setReducerClass(Join.class);
       FileInputFormat.addInputPath(join, new
Path("/user/gates/tmp/indexed_pages"));
       FileInputFormat.addInputPath(join, new
Path("/user/gates/tmp/filtered_users"));
       FileOutputFormat.setOutputPath(join, new
Path("/user/gates/tmp/joined"));
        join.setNumReduceTasks(50);
        Job joinJob = new Job(join);
        joinJob.addDependingJob(loadPages);
        joinJob.addDependingJob(loadUsers);

        JobConf group = new JobConf(MRExample.class);
        group.setJobName("Group URLs");
        group.setInputFormat(KeyValueTextInputFormat.class);
        group.setOutputKeyClass(Text.class);
        group.setOutputValueClass(LongWritable.class);
        group.setOutputFormat(SequenceFileOutputFormat.class);
        group.setMapperClass(LoadJoined.class);
        group.setCombinerClass(ReduceUrls.class);
        group.setReducerClass(ReduceUrls.class);
       FileInputFormat.addInputPath(group, new
Path("/user/gates/tmp/joined"));
       FileOutputFormat.setOutputPath(group, new
Path("/user/gates/tmp/grouped"));
        group.setNumReduceTasks(50);
        Job groupJob = new Job(group);
        groupJob.addDependingJob(joinJob);

        JobConf top100 = new JobConf(MRExample.class);
        top100.setJobName("Top 100 sites");
        top100.setInputFormat(SequenceFileInputFormat.class);
        top100.setOutputKeyClass(LongWritable.class);
        top100.setOutputValueClass(Text.class);
        top100.setOutputFormat(SequenceFileOutputFormat.class);
        top100.setMapperClass(LoadClicks.class);
        top100.setCombinerClass(LimitClicks.class);
        top100.setReducerClass(LimitClicks.class);
       FileInputFormat.addInputPath(top100, new
Path("/user/gates/tmp/grouped"));
       FileOutputFormat.setOutputPath(top100, new
Path("/user/gates/top100sitesforusers18to25"));
        top100.setNumReduceTasks(1);
        Job limit = new Job(top100);
        limit.addDependingJob(groupJob);

       JobControl jc = new JobControl("Find top 100 sites for users
18  to 25");
        jc.addJob(loadPages);
        jc.addJob(loadUsers);
        jc.addJob(joinJob);
        jc.addJob(groupJob);
        jc.addJob(limit);
        jc.run();
    }
}
```

170 lines of code, 4 hours to write

Credit: Alan Gates, Yahoo!

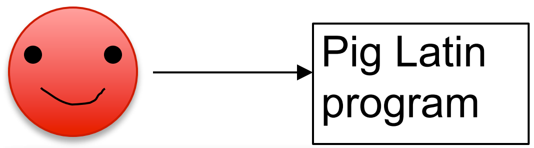
In Pig Latin

```
Users = load 'users' as (name, age);
Fltrd = filter Users by
           age >= 18 and age <= 25;
Pages = load 'pages' as (user, url);
Jnd = join Fltrd by name, Pages by user;
Grpd = group Jnd by url;
Smmd = foreach Grpd generate group,
          COUNT(Jnd) as clicks;
Srtd = order Smmd by clicks desc;
Top5 = limit Srtd 5;
store Top5 into 'top5sites';
```

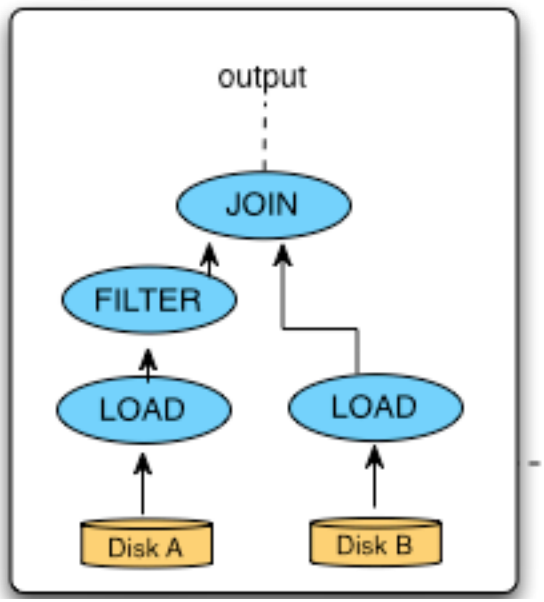
9 lines of code, 15 minutes to write

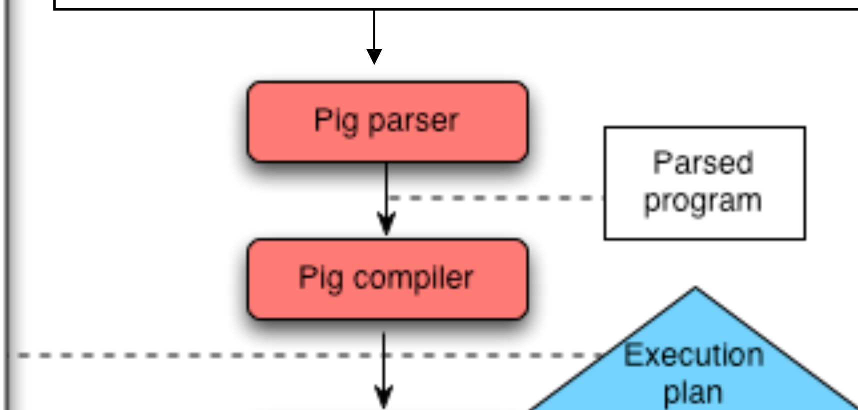
Credit: Alan Gates, Yahoo!

Background: Pig system

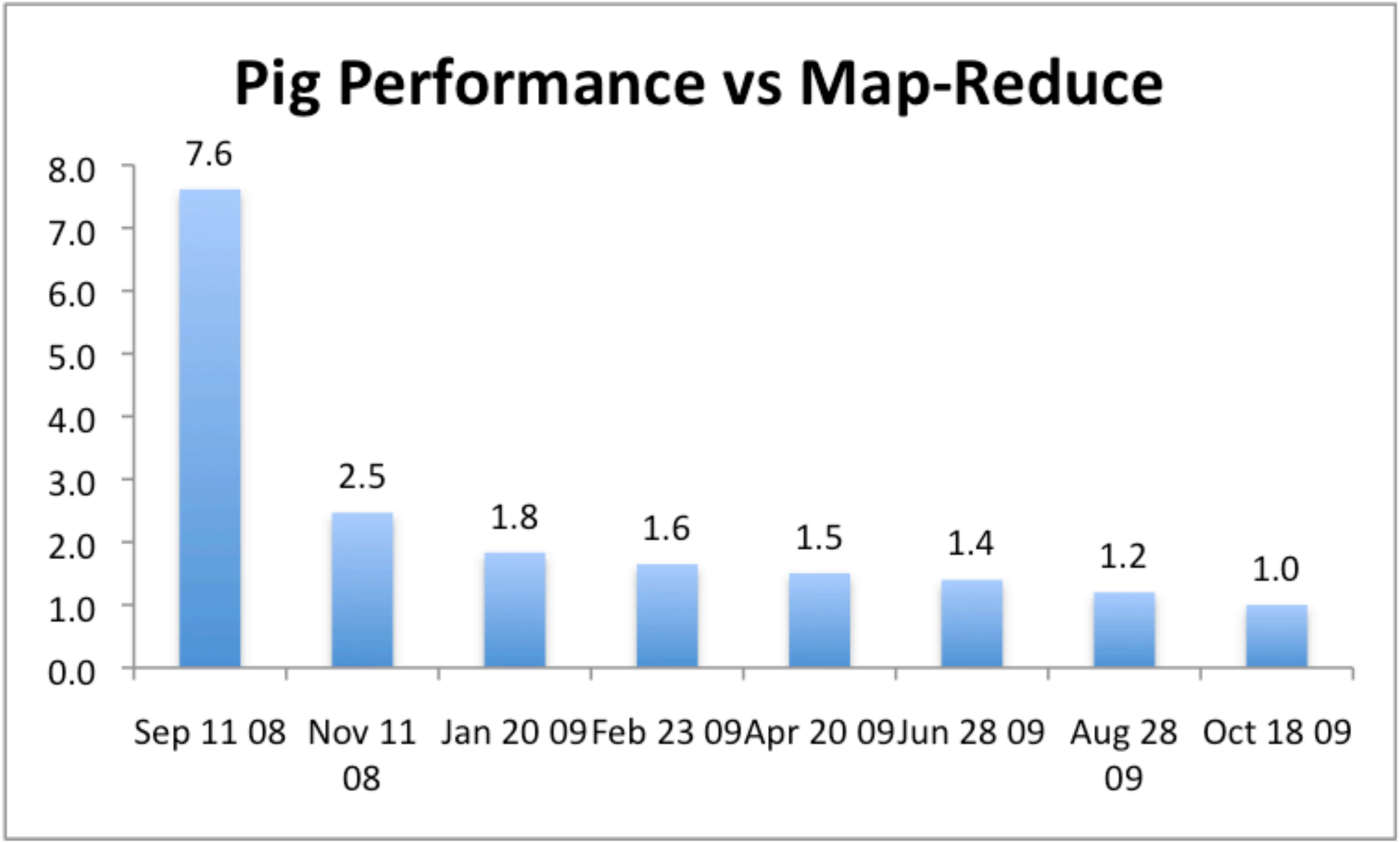



```
A = LOAD 'file1' AS (sid,pid,mass,px:double);
B = LOAD 'file2' AS (sid,pid,mass,px:double);
C = FILTER A BY px < 1.0;
D = JOIN C BY sid,
         B BY sid;
  STORE g INTO 'output.txt';
```





But can it fly?

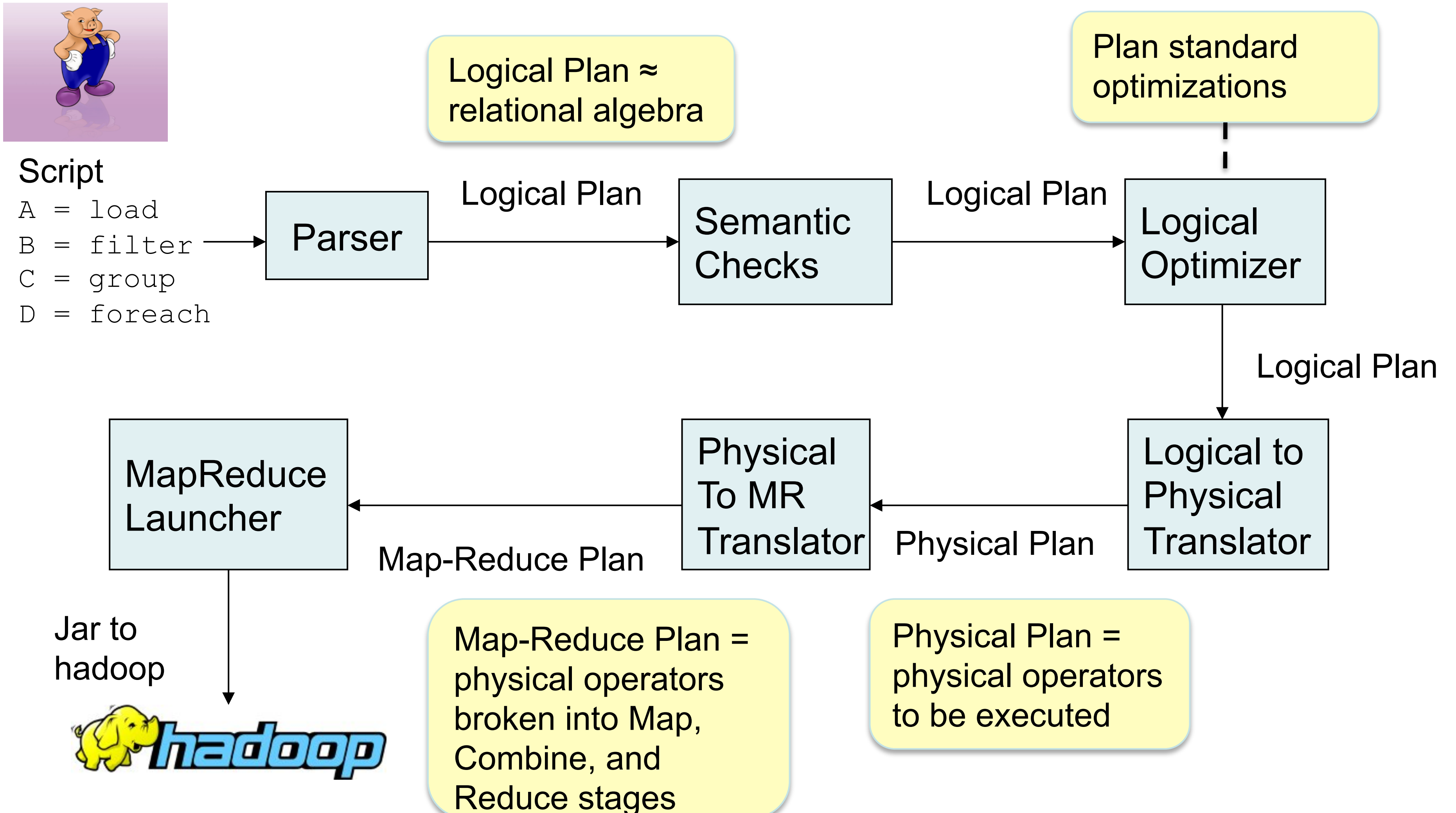


Credit: Alan Gates, Yahoo!

Essence of Pig

- Map-Reduce is too low a level to program, SQL too high
- Pig Latin, a language intended to sit between the two:
  - Imperative
  - Provides standard relational transforms (join, sort, etc.)
  - Schemas are optional, used when available, can be defined at runtime
  - User Defined Functions are first class citizens
  - Opportunities for advanced optimizer but optimizations by programmer also possible

How It Works



Script
A = load
B = filter
C = group
D = foreach
Parser
Logical Plan ≈ relational algebra
Logical Plan
Semantic Checks
Plan standard optimizations
Logical Plan
Logical Optimizer
Logical Plan
Logical to Physical Translator
Physical Plan
Physical To MR Translator
Map-Reduce Plan
MapReduce Launcher
Jar to hadoop
hadoop
Map-Reduce Plan = physical operators broken into Map, Combine, and Reduce stages
Physical Plan = physical operators to be executed

Tenzing

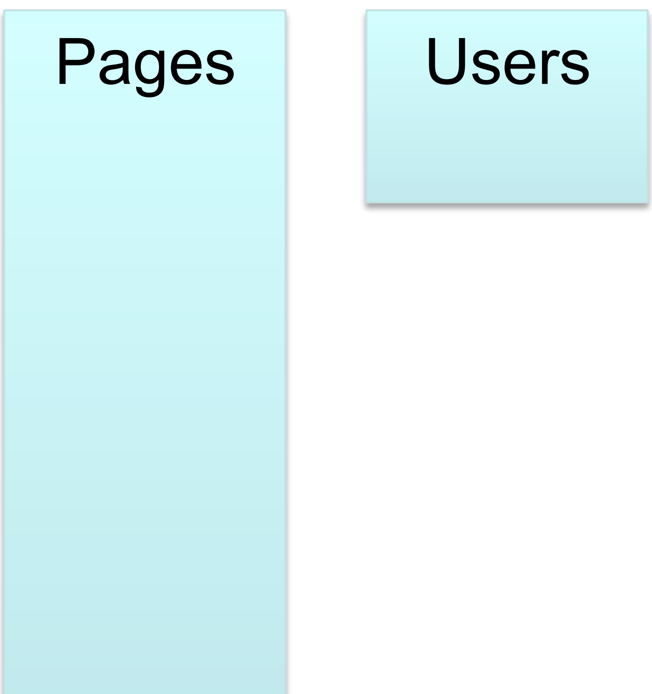
- Google’s implementation of SQL
- Supports full SQL92
- On top of google’s Map/Reduce
- Uses traditional query optimizer, plus optimizations to MR
- Widely adopted inside Google, especially by the non-engineering community

Join Algorithms on Map/Reduce

- Broadcast join
- Hash-join
- Skew join
- Merge join

Fragment Replicate Join

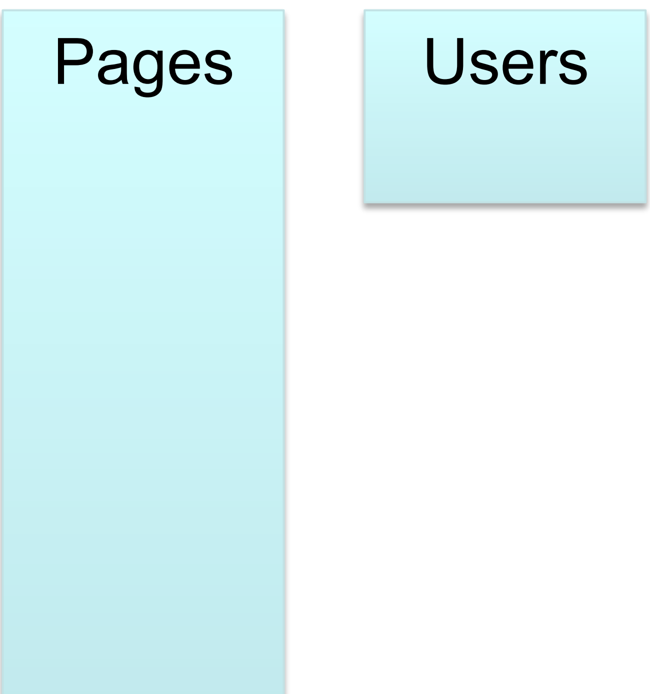
Aka
“Broakdcast Join”



Fragment Replicate Join

Aka
“Broakdcast Join”

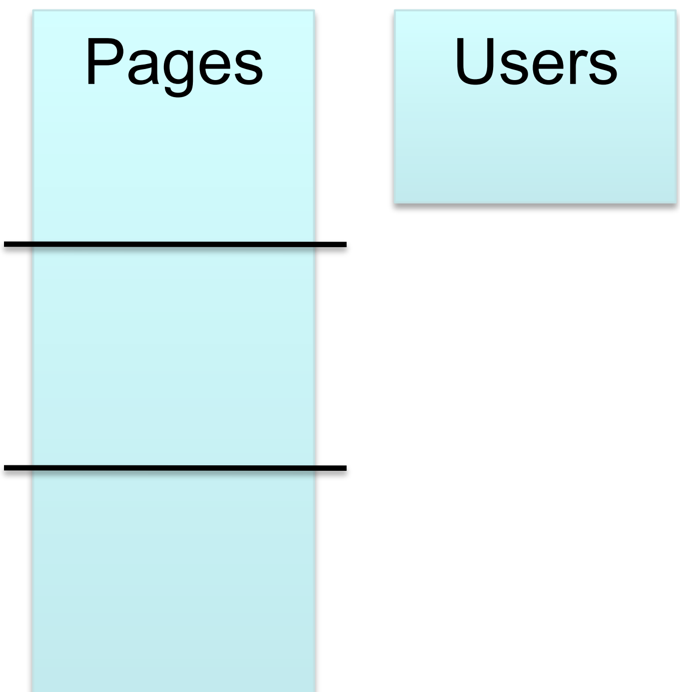
```
Users = load ‘users’ as (name, age);
Pages = load ‘pages’ as (user, url);
Jnd = join Pages by user, Users by name using “replicated”;
```



Fragment Replicate Join

Aka
“Broakdcast Join”

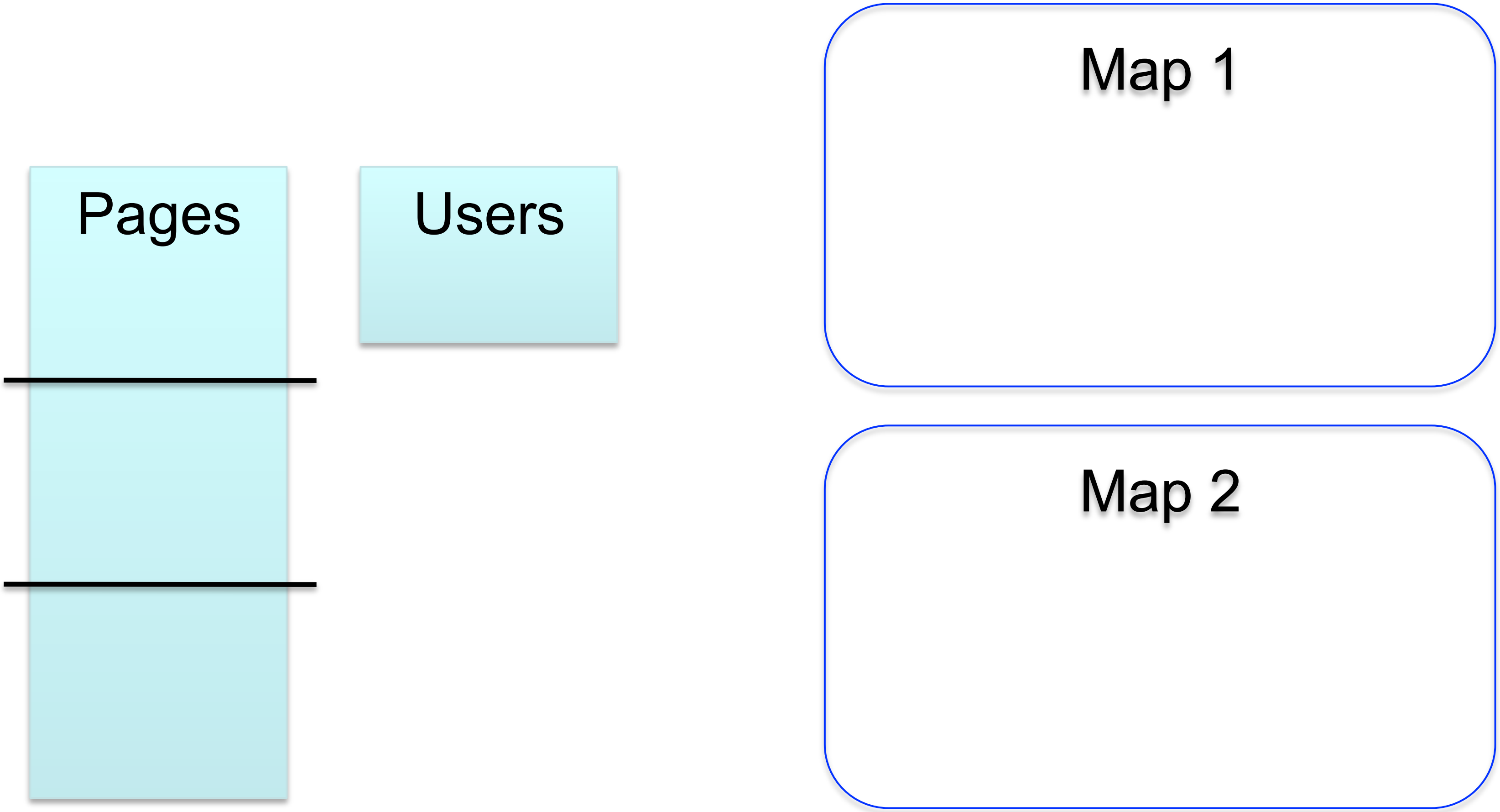
```
Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Pages by user, Users by name using "replicated";
```



Fragment Replicate Join

Aka
"Broakdcast Join"

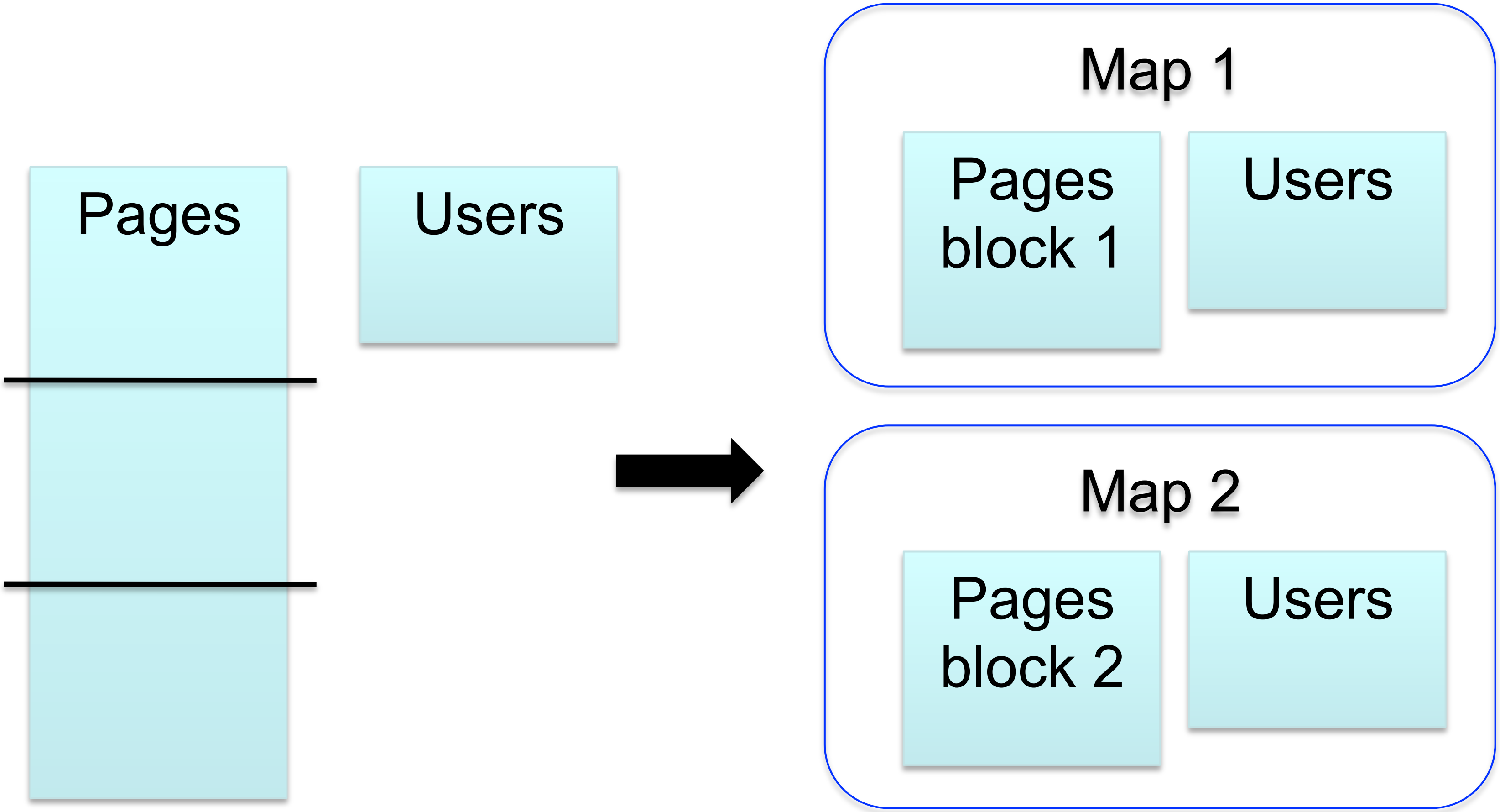
```
Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Pages by user, Users by name using "replicated";
```



Fragment Replicate Join

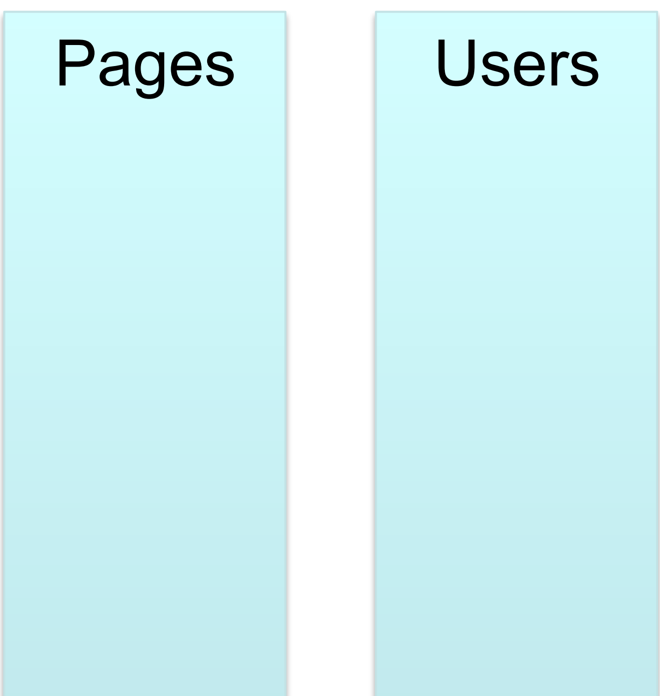
Aka
"Broakdcast Join"

```
Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Pages by user, Users by name using "replicated";
```



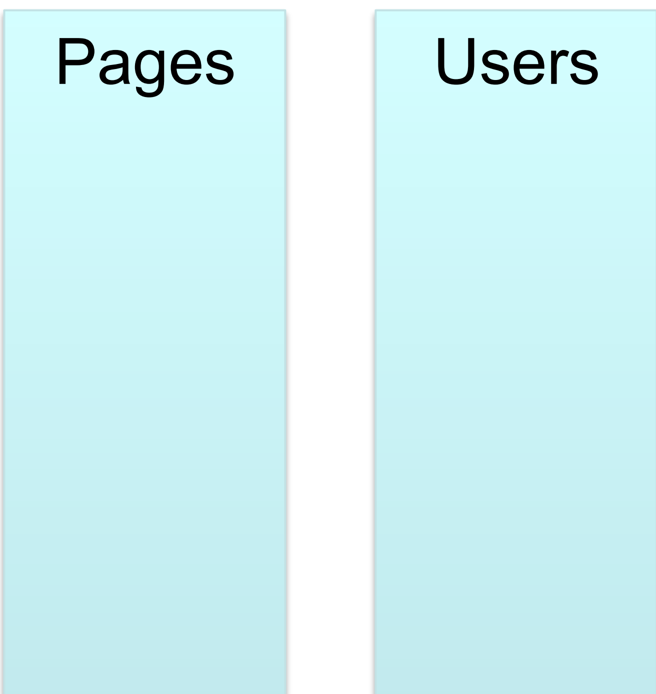
Credit: Alan Gates, Yahoo!

Hash Join



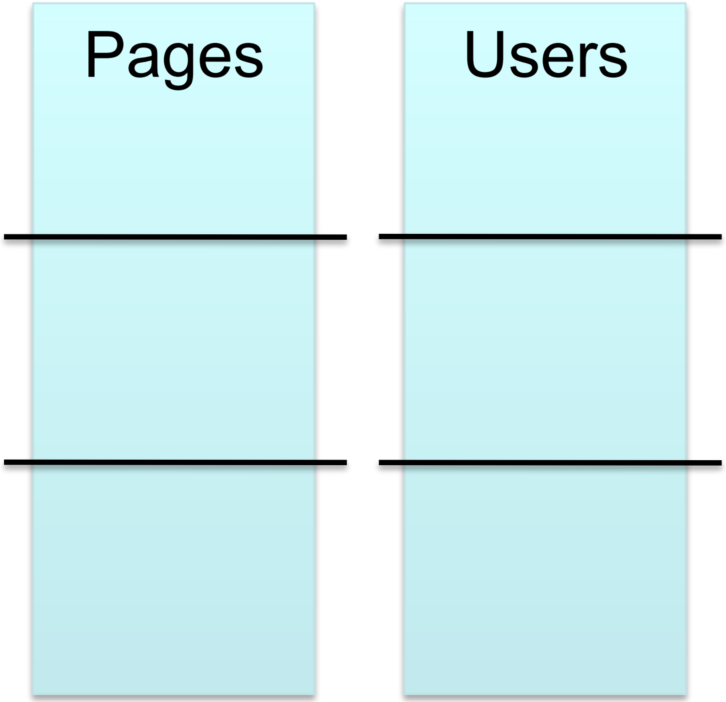
Hash Join

```
Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Users by name, Pages by user;
```



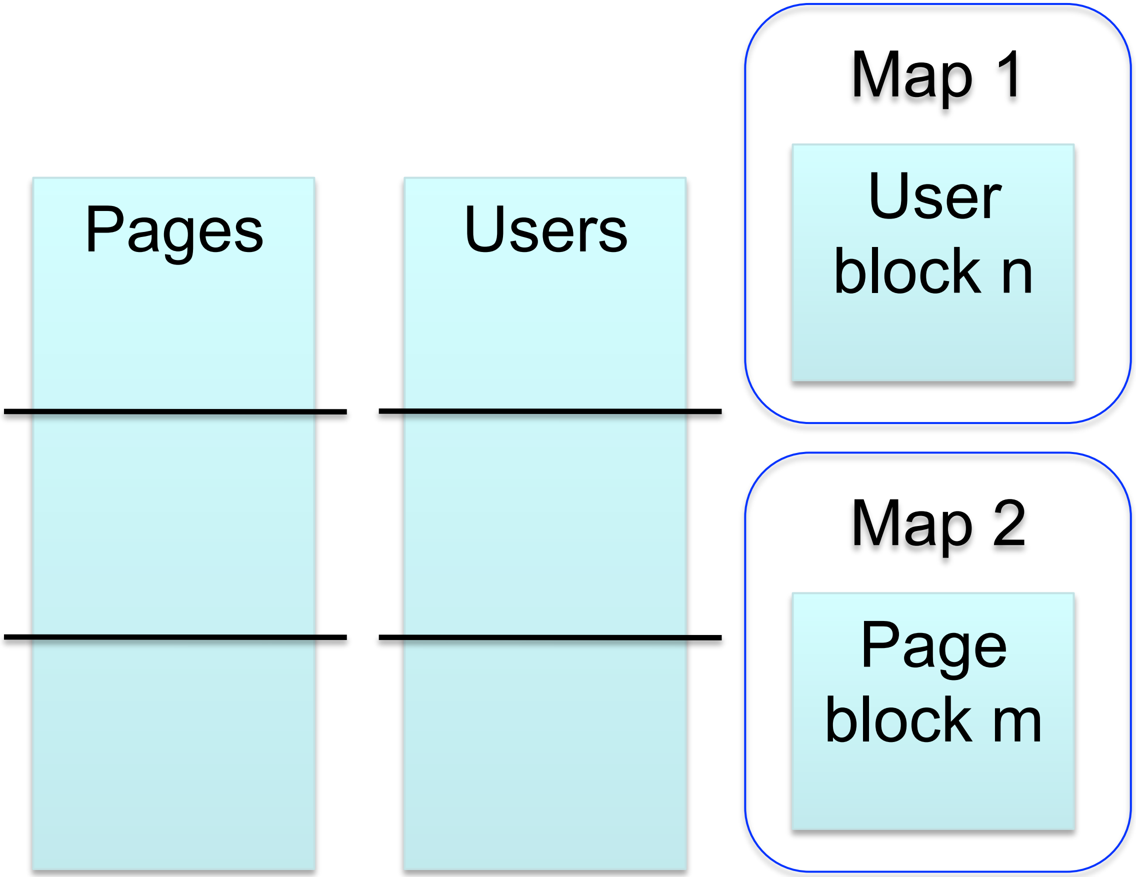
Hash Join

```
Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Users by name, Pages by user;
```



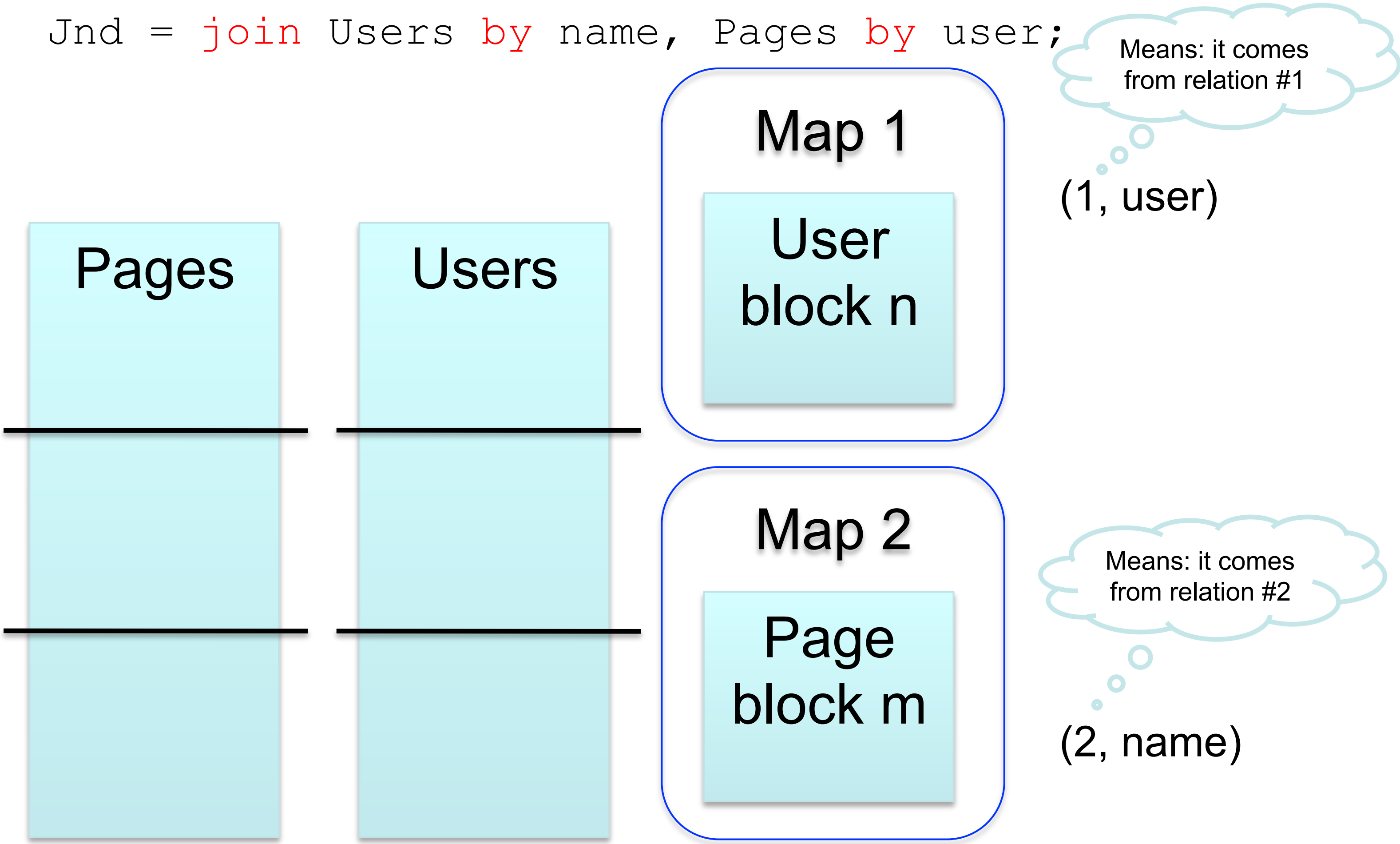
Hash Join

```
Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Users by name, Pages by user;
```



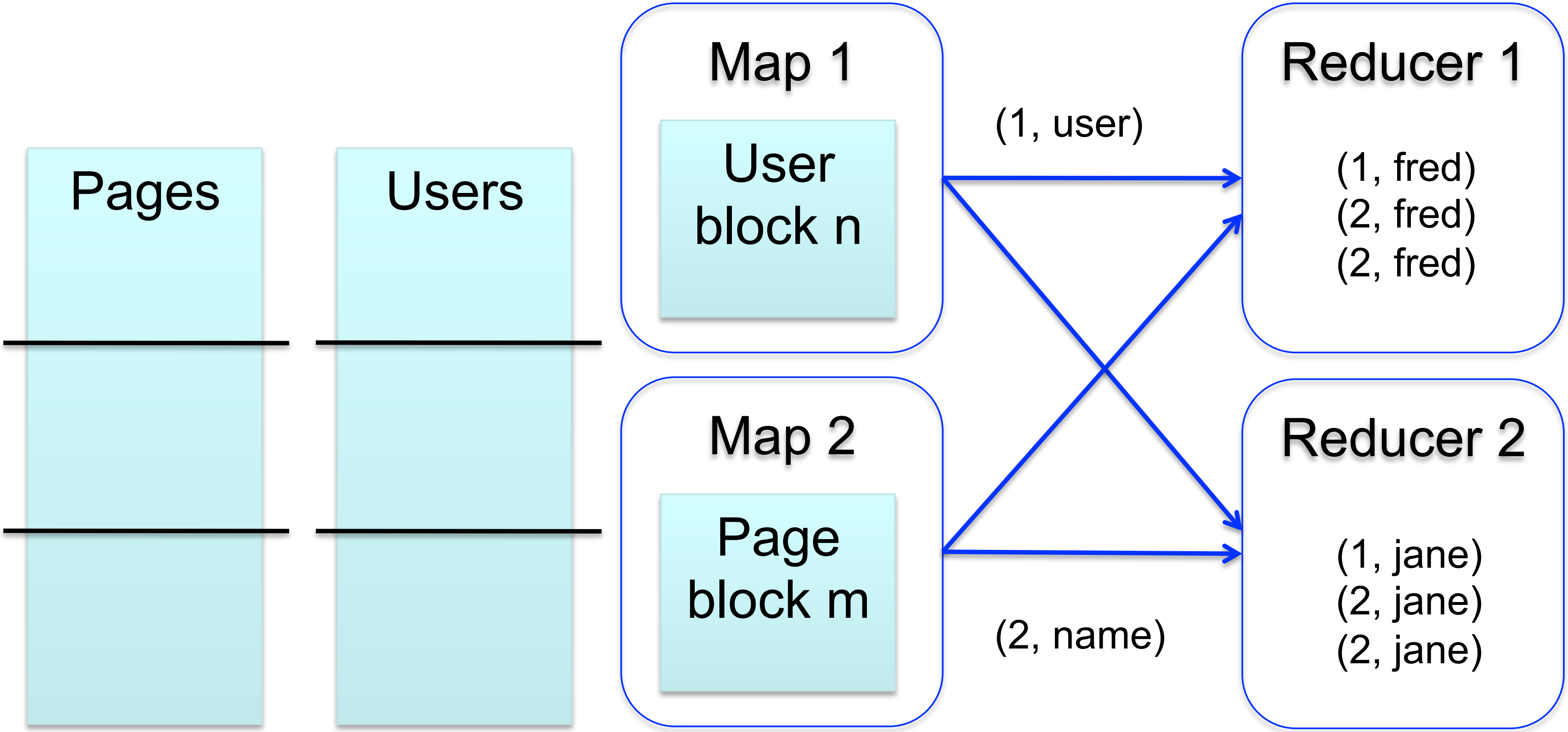
Hash Join

```
Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Users by name, Pages by user;
```

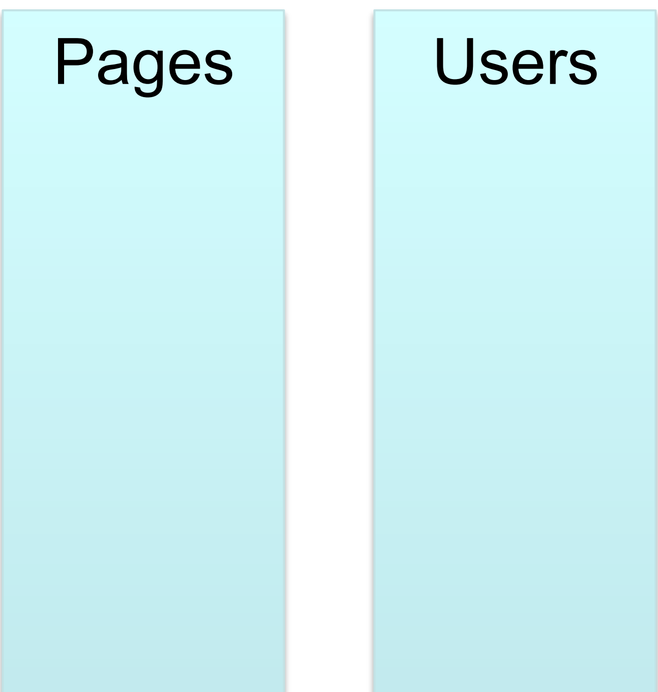


Hash Join

```
Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Users by name, Pages by user;
```

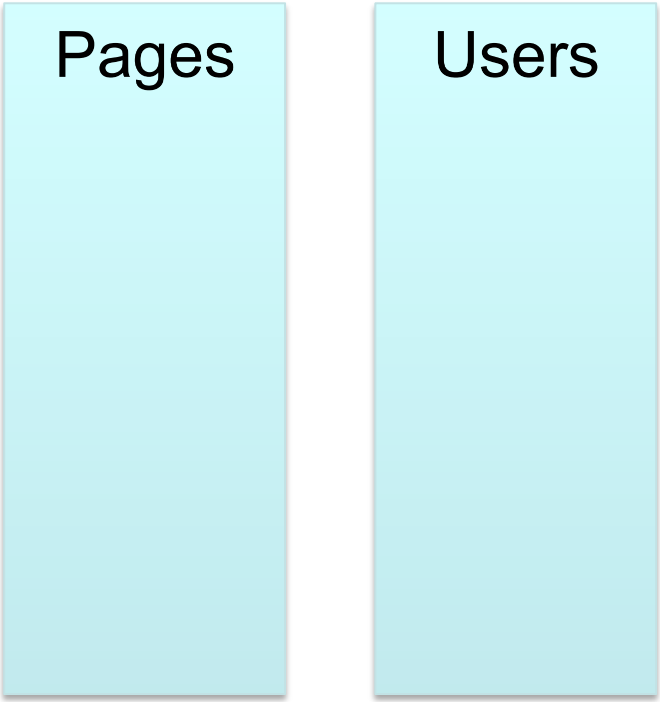


Skew Join



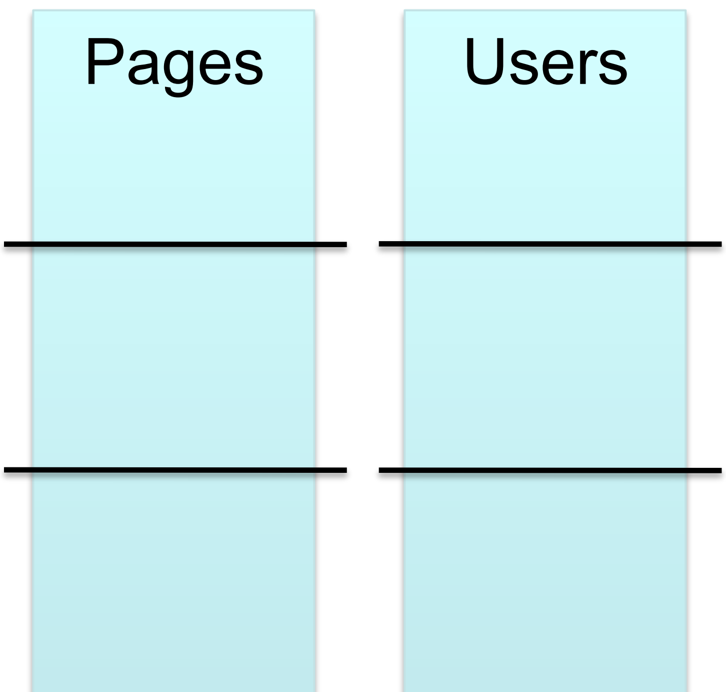
Skew Join

```
Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Pages by user, Users by name using "skewed";
```



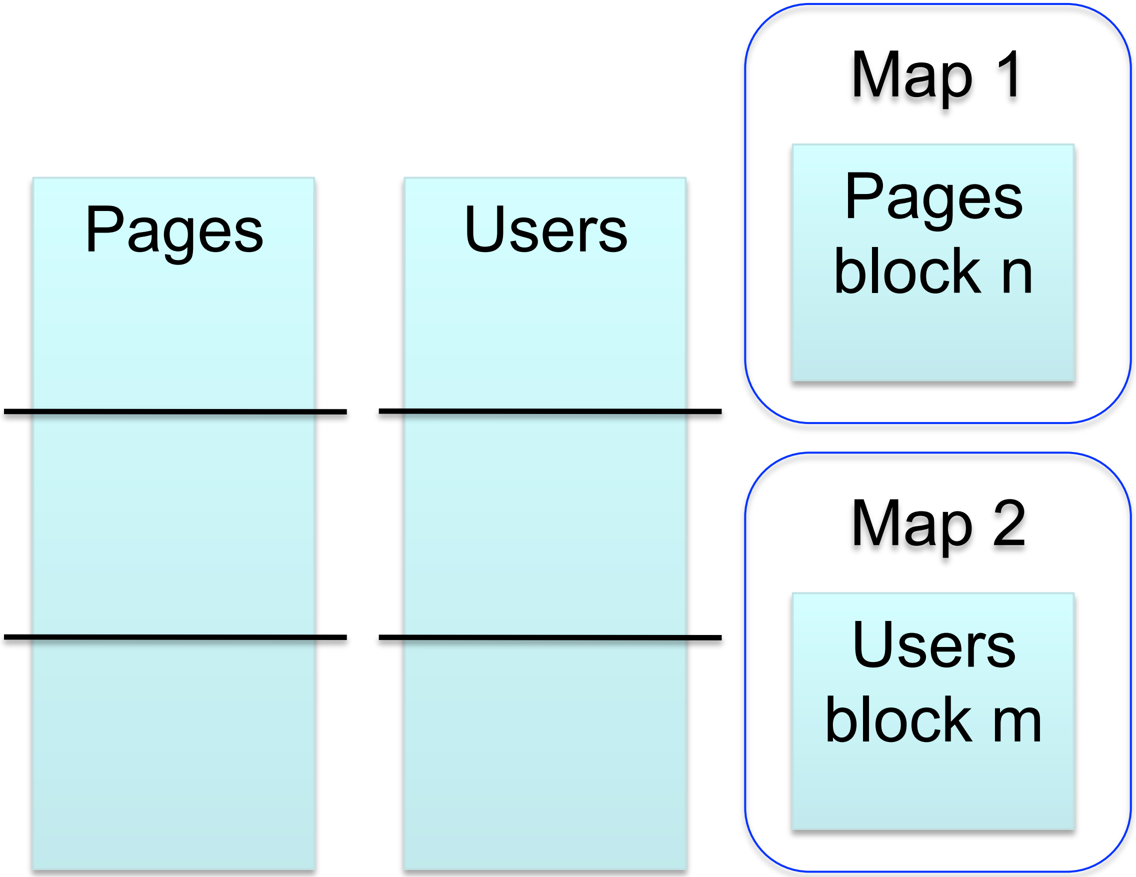
Skew Join

```
Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Pages by user, Users by name using "skewed";
```



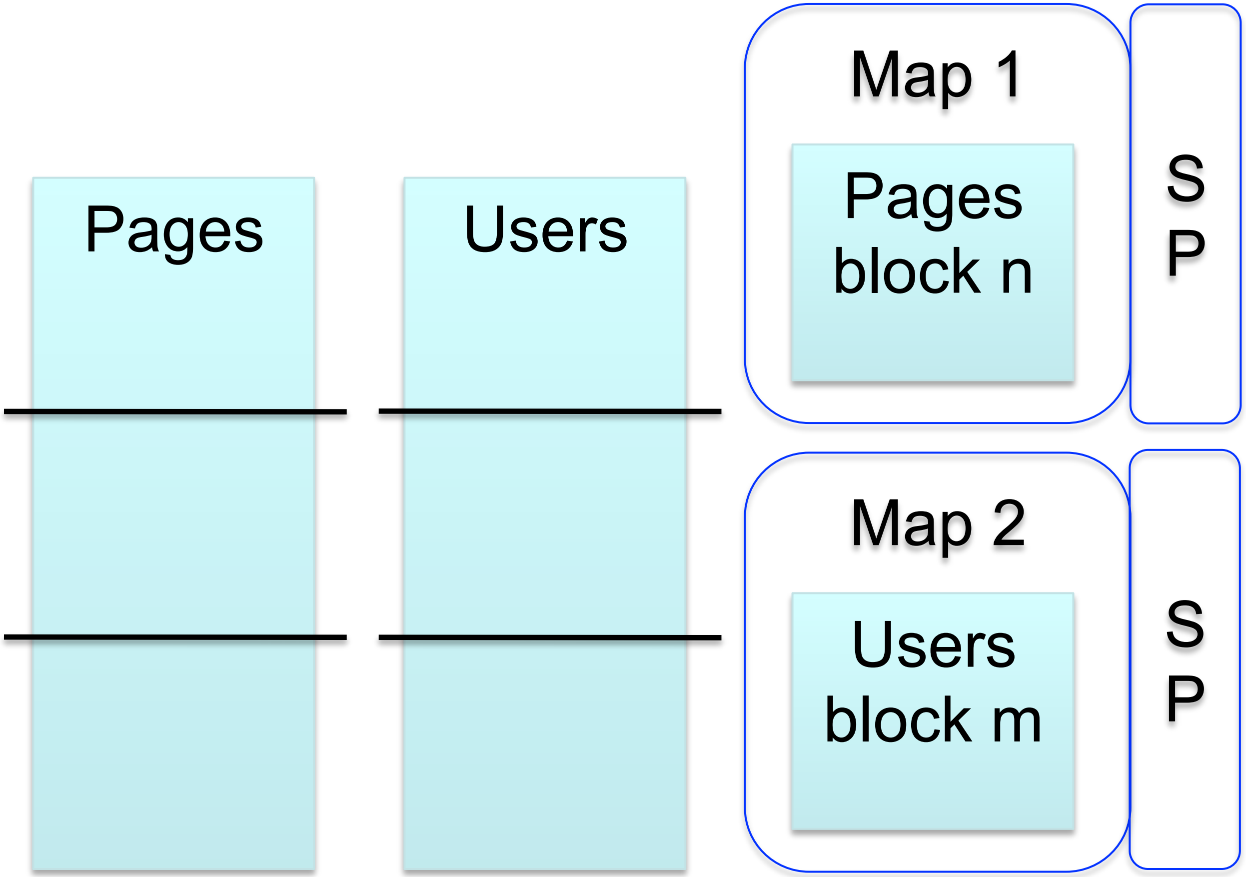
Skew Join

```
Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Pages by user, Users by name using "skewed";
```



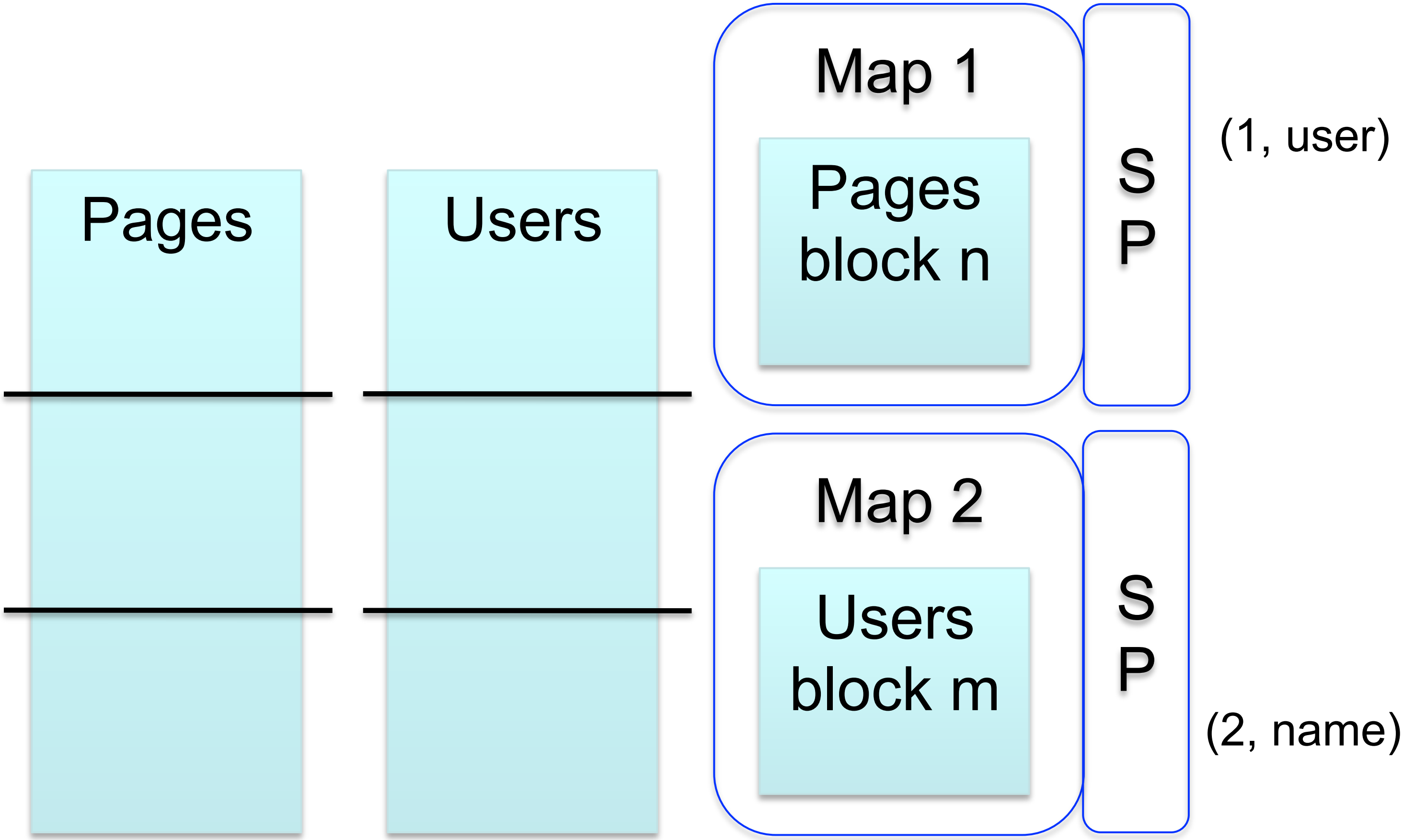
Skew Join

```
Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Pages by user, Users by name using "skewed";
```



Skew Join

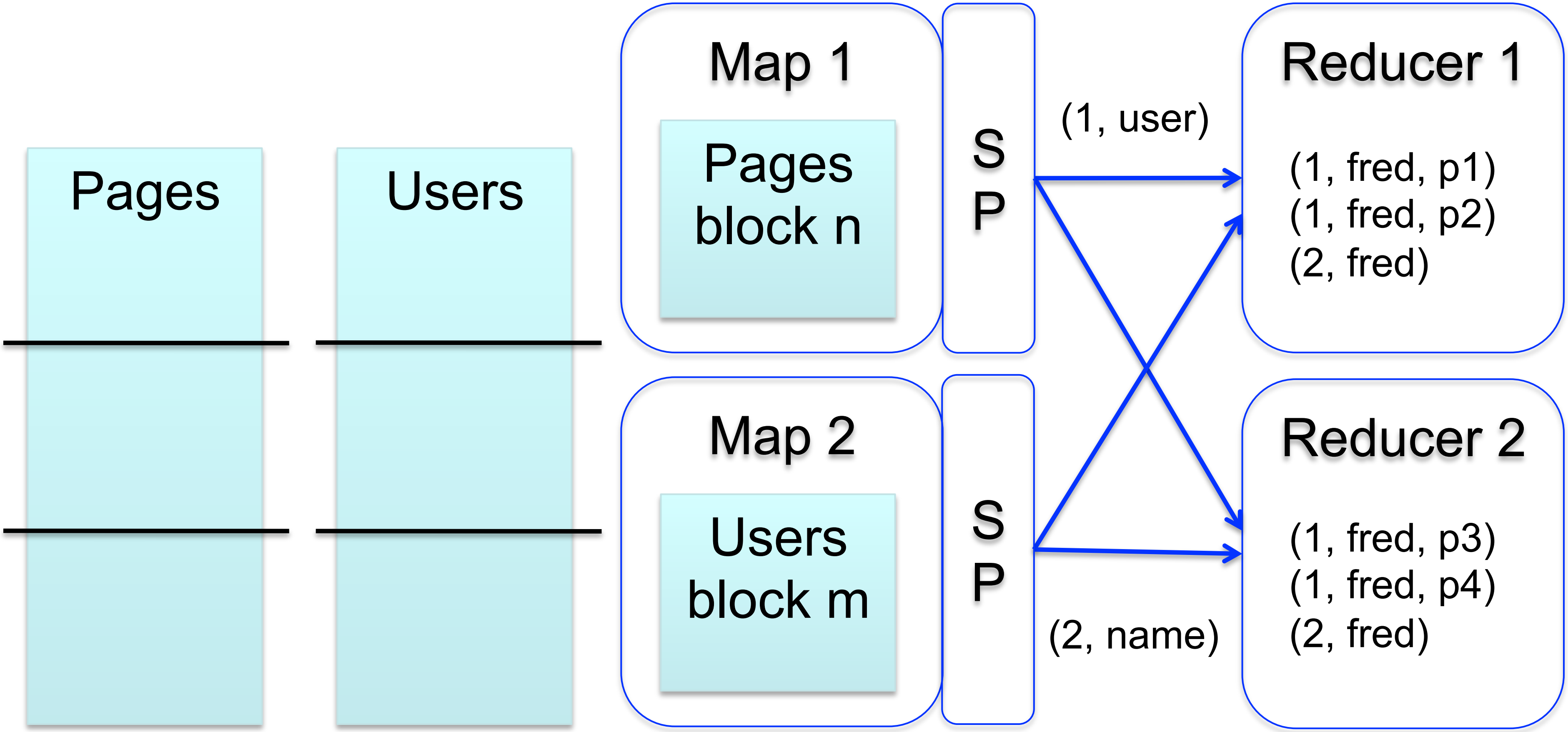
```
Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Pages by user, Users by name using "skewed";
```



Credit: Alan Gates, Yahoo!

Skew Join

```
Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Pages by user, Users by name using "skewed";
```



Merge Join

Pages	Users
aaron	aaron
.	.
.	.
.	.
.	.
.	.
.	.
.	.
.	.
zach	zach

Credit: Alan Gates, Yahoo!

Merge Join

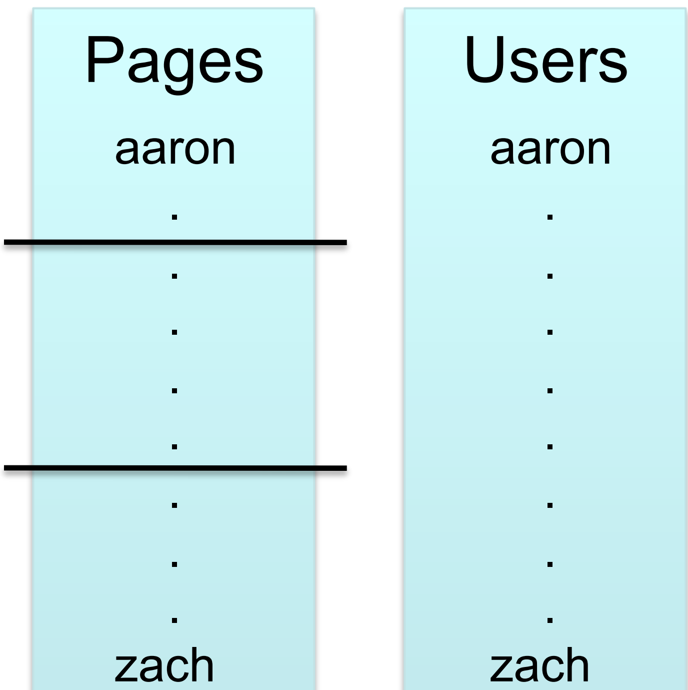
```
Users = load ‘users’ as (name, age);
Pages = load ‘pages’ as (user, url);
Jnd = join Pages by user, Users by name using “merge”;
```

Pages	Users
aaron	aaron
.	.
.	.
.	.
.	.
.	.
.	.
.	.
.	.
zach	zach

Credit: Alan Gates, Yahoo!

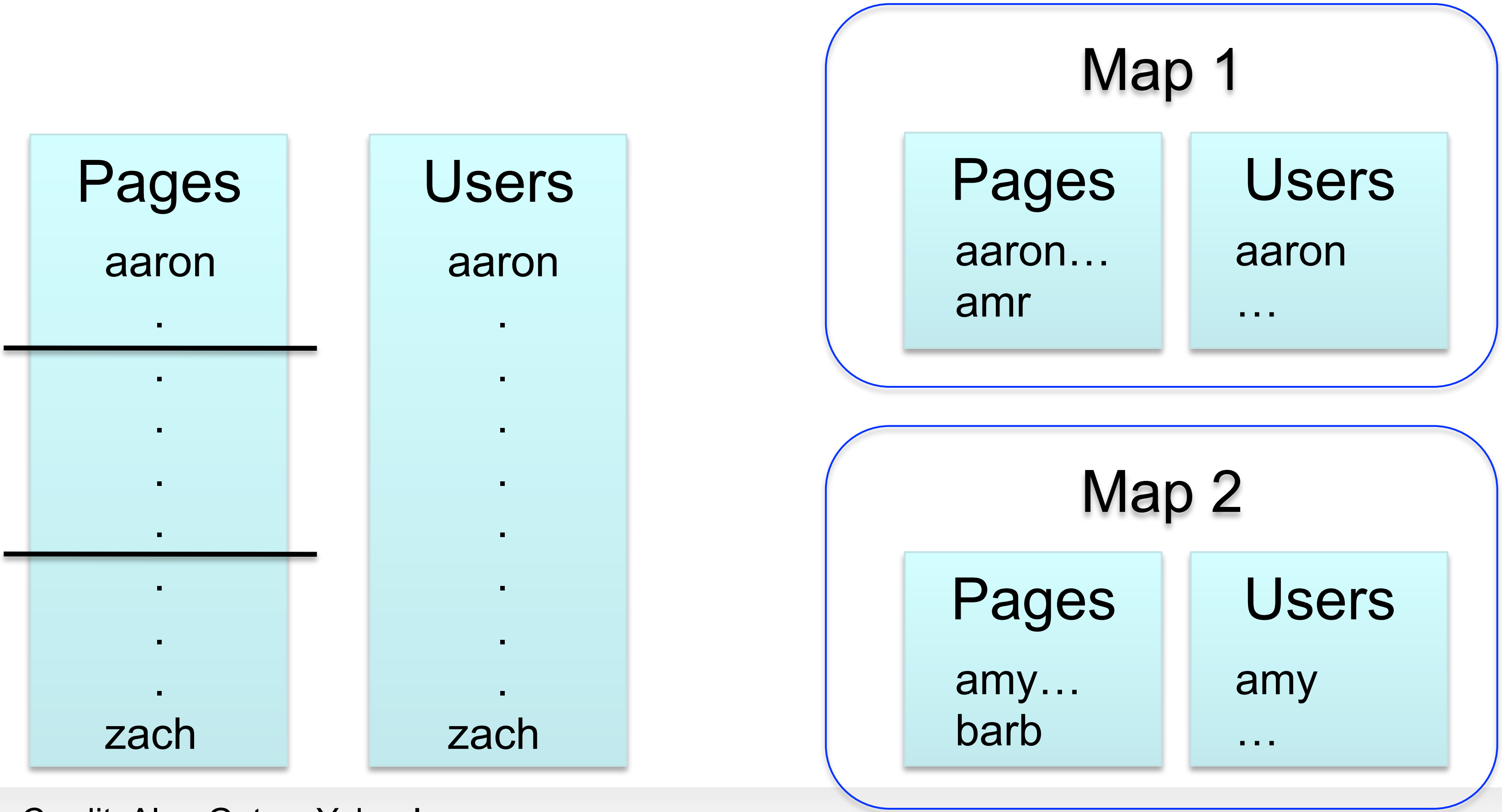
Merge Join

```
Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Pages by user, Users by name using "merge";
```



Merge Join

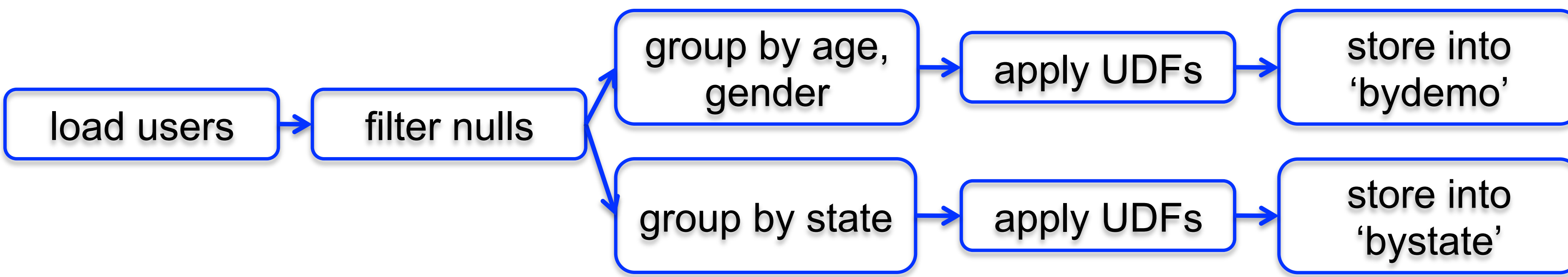
```
Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Pages by user, Users by name using "merge";
```



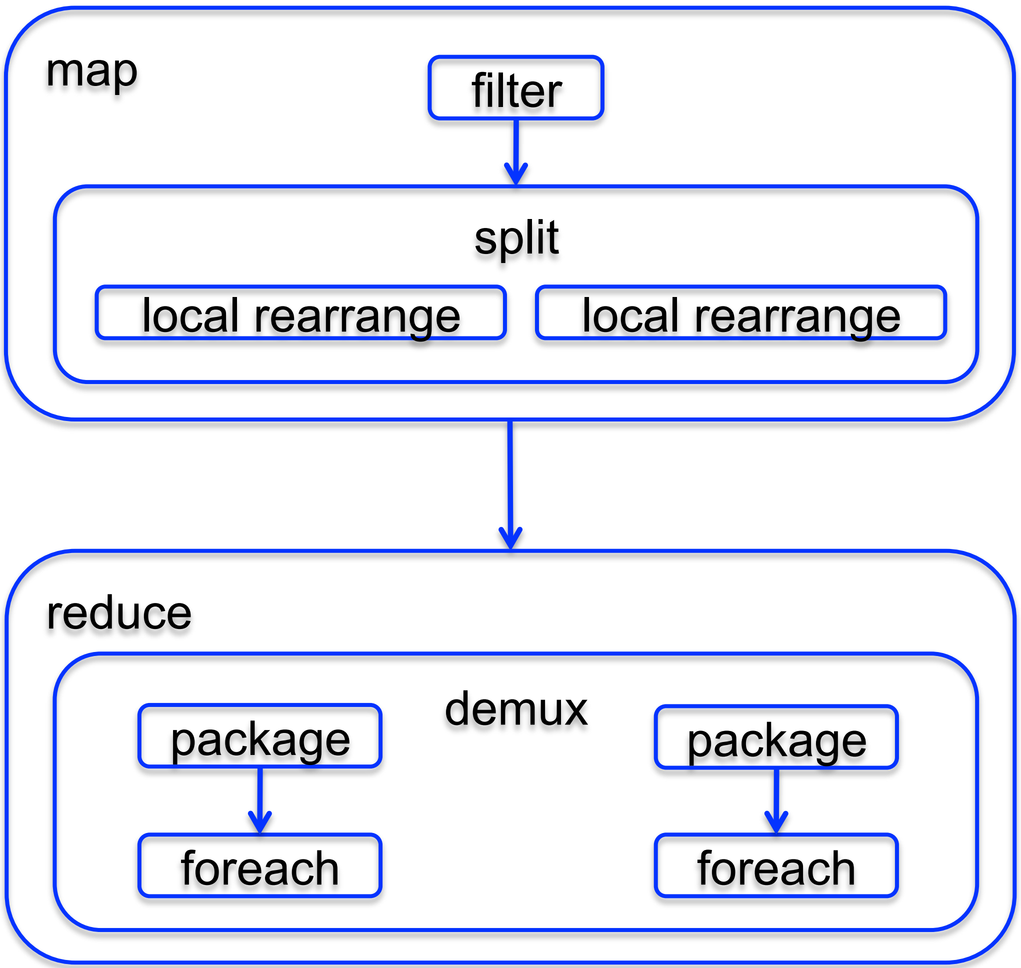
Credit: Alan Gates, Yahoo!

Multi-store script

```
A = load ‘users’ as (name, age, gender,
      city, state);
B = filter A by name is not null;
C1 = group B by age, gender;
D1 = foreach C1 generate group, COUNT(B);
store D into ‘bydemo’;
C2= group B by state;
D2 = foreach C2 generate group, COUNT(B);
store D2 into ‘bystate’;
```



Multi-Store Map-Reduce Plan



Other Optimizations in Tenzing

- Keep processes running: process pool
- Remove reducer-side sort for hash-based algorithms
  - Note: the data must fit in main memory, otherwise the task fails
- Pipelining
- Indexes

Final Thoughts

Challenging problems in MR jobs:

- Skew

- Fault tolerance

Balazinska, *A study of Skew*

Skew

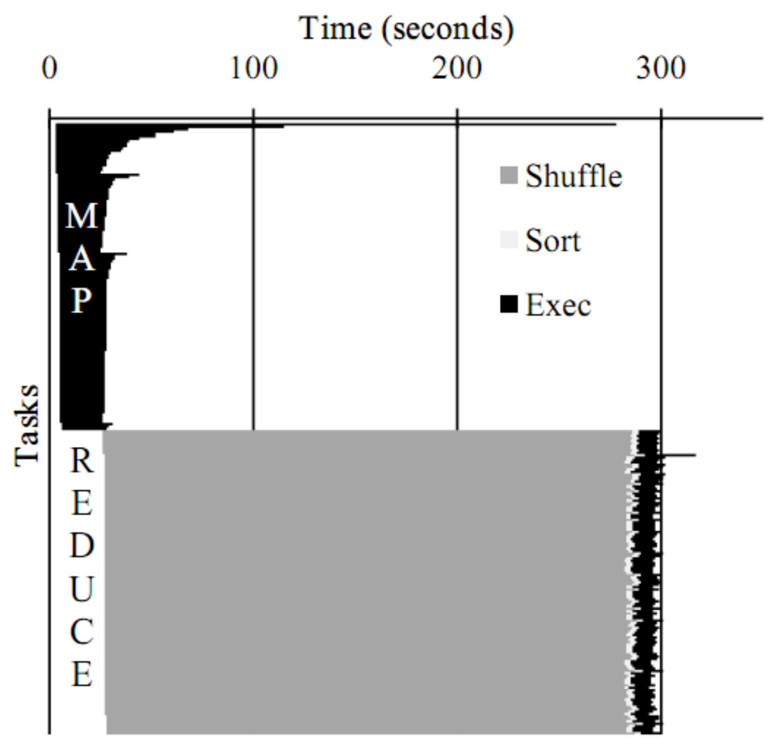
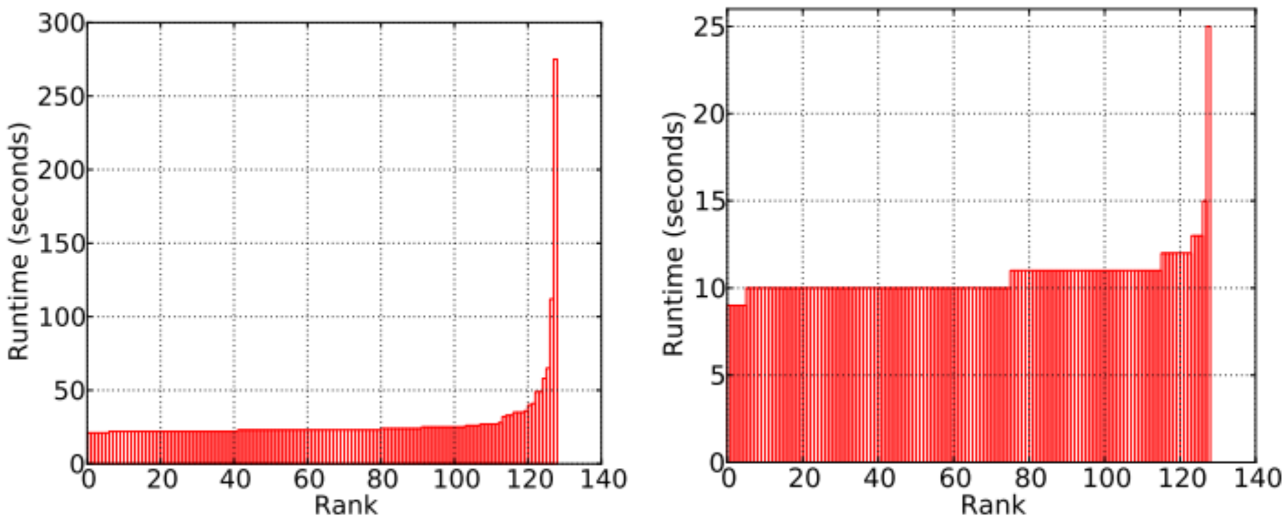


Fig. 1. A timing chart of a MapReduce job running the PageRank algorithm from Cloud 9 [5]. Exec represents the actual map and reduce operations. The slowest map task (first one from the top) takes more than twice as long to complete as the second slowest map task, which is still five times slower than the average. If all tasks took approximately the same amount of time, the job would have completed in less than half the time.

Balazinska, *A study of Skew*

Skew

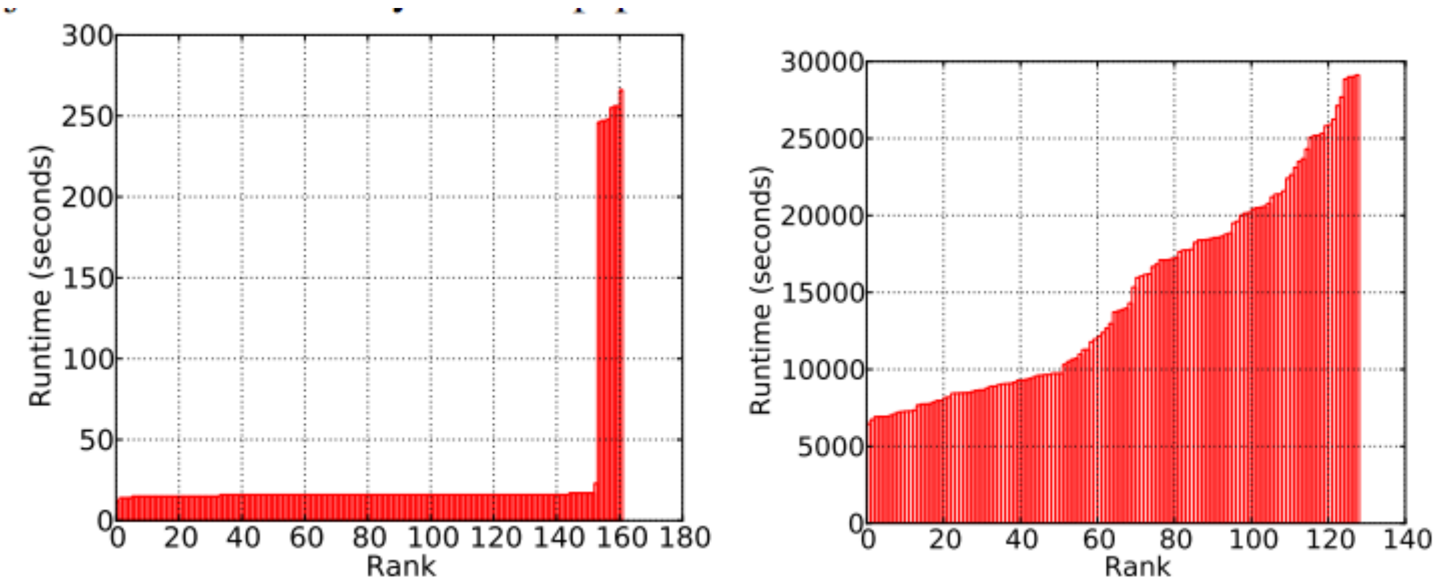


(a) Page Rank - Map (b) Page Rank - Reduce

Fig. 2. The distribution of task runtimes for PageRank with 128 map and 128 reduce tasks. A graph node with a large number of edges is much more expensive to process than many graph nodes with few edges. Skew arises in both the map and reduce phases, but the overall job is dominated by the map phase.

Balazinska, *A study of Skew*

Skew



(a) CloudBurst - Map (b) CloudBurst - Reduce

Fig. 3. Distribution of task runtime for CloudBurst. Total 162 map tasks, and 128 reduce tasks. The map phase exhibits a bimodal distribution. Each mode corresponds to map tasks processing a different input dataset. The reduce is computationally expensive and has a smooth runtime distribution, but there is a factor of five difference in runtime between the fastest and the slowest reduce tasks.

Fault Tolerance

- Fundamental tension:
- Materialize after each Map and each Reduce
  - This is what MR does
  - Ideal for fault tolerance
  - Very poor performance
- Pipeline between steps
  - This is what Parallel DBs usually do
  - Ideal for performance
  - Very poor fault tolerance

Pig Latin Mini-Tutorial

(will skip in class; please read in order to do homework 6)

Outline

Based entirely on *Pig Latin: A not-so-foreign language for data processing*, by Olston, Reed, Srivastava, Kumar, and Tomkins, 2008

Quiz section tomorrow: in CSE 403 (this is CSE, don't go to EE1)

Pig-Latin Overview

- Data model = loosely typed *nested relations*
- Query model = a sql-like, dataflow language

- Execution model:
  - Option 1: run locally on your machine
  - Option 2: compile into sequence of map/reduce, run on a cluster supporting Hadoop

- Main idea: use Opt1 to debug, Opt2 to execute

Example

- Input: a table of urls:
  (url, category, pagerank)
- Compute the average pagerank of all sufficiently high pageranks, for each category
- Return the answers only for categories with sufficiently many such pages

First in SQL…

```
SELECT category, AVG(pagerank)
FROM urls
WHERE pagerank > 0.2
GROUP By category
HAVING COUNT(*) > 10^6
```

…then in Pig-Latin

```
good_urls = FILTER urls BY pagerank > 0.2
groups = GROUP good_urls BY category
big_groups = FILTER groups
                BY COUNT(good_urls) > 10^6
output = FOREACH big_groups GENERATE
                category, AVG(good_urls.pagerank)
```

Types in Pig-Latin

- Atomic: string or number, e.g. ‘Alice’ or 55
- Tuple: (‘Alice’, 55, ‘salesperson’)
- Bag: {(‘Alice’, 55, ‘salesperson’),
  (‘Betty’,44, ‘manager’), …}
- Maps: we will try not to use these

Types in Pig-Latin

Bags can be nested !

- {(‘a’, {1,4,3}), (‘c’,{ }), (‘d’, {2,2,5,3,2})}

Tuple components can be referenced by number

- $0, $1, $2, …

$$\texttt{t} = \left(\text{`alice'}, \left\{\begin{array}{c}(\text{`lakers'}, 1)\\(\text{`iPod'}, 2)\end{array}\right\}, [\text{`age'} \rightarrow 20]\right)$$

Let fields of tuple `t` be called `f1, f2, f3`

Expression Type	Example	Value for `t`
Constant	`'bob'`	Independent of `t`
Field by position	`$0`	`'alice'`
Field by name	`f3`	$[\text{`age'} \rightarrow 20]$
Projection	`f2.$0`	$\left\{\begin{array}{c}(\text{`lakers'})\\(\text{`iPod'})\end{array}\right\}$
Map Lookup	`f3#'age'`	20
Function Evaluation	`SUM(f2.$1)`	`1 + 2 = 3`
Conditional Expression	`f3#'age'>18? 'adult':'minor'`	`'adult'`
Flattening	`FLATTEN(f2)`	`'lakers', 1` `'iPod', 2`

Loading data

- Input data = FILES !
  - Heard that before ?

- The LOAD command parses an input file into a bag of records
- Both parser (=“deserializer”) and output type are provided by user

Loading data

```
queries = LOAD ‘query_log.txt’
          USING myLoad( )
          AS (userID, queryString, timeStamp)
```

Loading data

- USING userfuction( ) -- is optional
  – Default deserializer expects tab-delimited file
- AS type – is optional
  – Default is a record with unnamed fields; refer to them as $0, $1, …
- The return value of LOAD is just a handle to a bag
  – The actual reading is done in pull mode, or parallelized

FOREACH

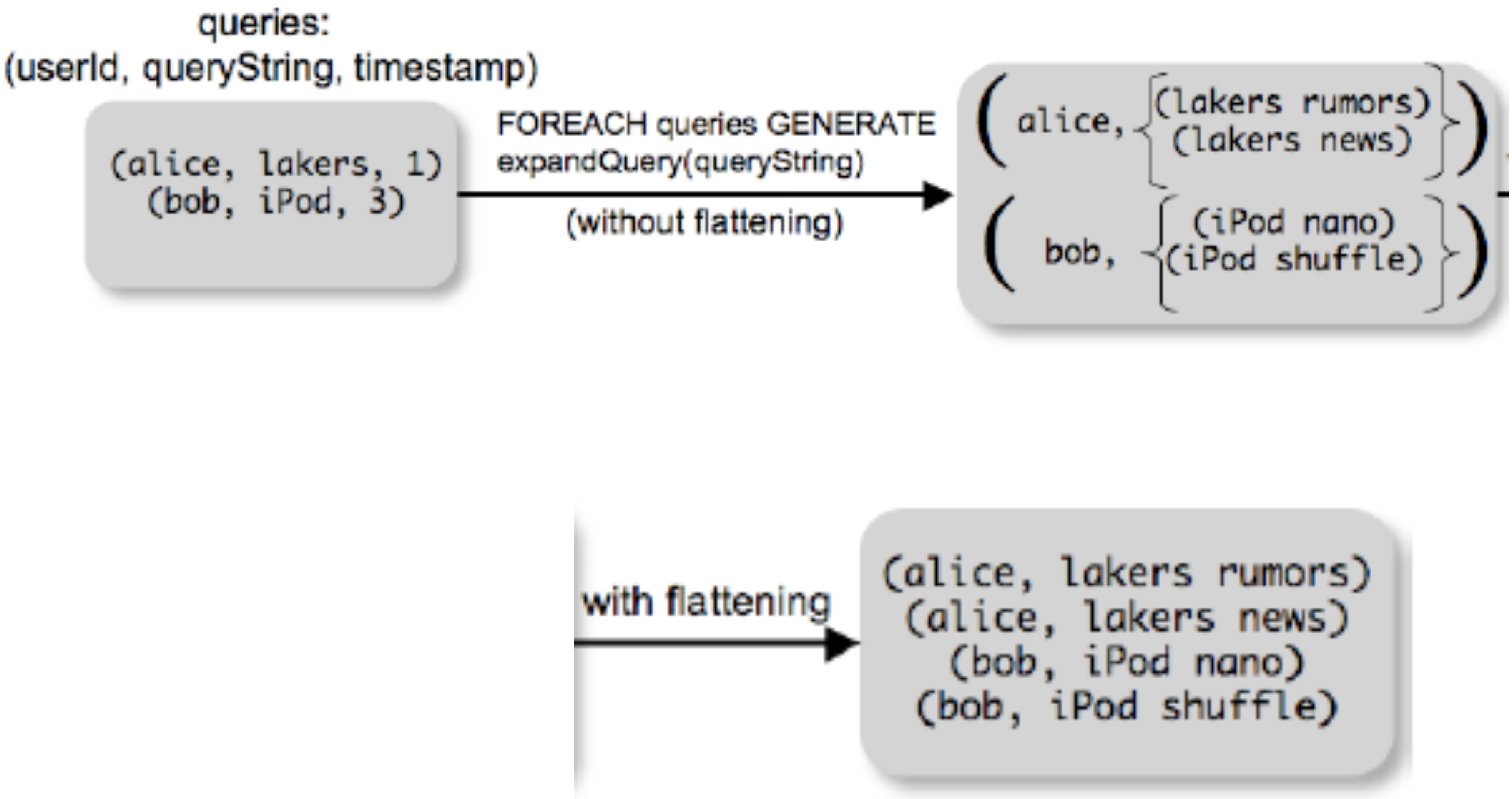
```
expanded_queries =
   FOREACH queries
   GENERATE userId, expandQuery(queryString)
```

expandQuery( ) is a UDF that produces likely expansions
Note: it returns a bag, hence expanded_queries is a nested bag

FOREACH

```
expanded_queries =
   FOREACH queries
   GENERATE userId,
                  flatten(expandQuery(queryString))
```

Now we get a flat collection


queries:
(userId, queryString, timestamp)
(alice, lakers, 1)
(bob, iPod, 3)
FOREACH queries GENERATE
expandQuery(queryString)
(without flattening)
alice,
(lakers rumors)
(lakers news)
bob,
(iPod nano)
(iPod shuffle)
with flattening
(alice, lakers rumors)
(alice, lakers news)
(bob, iPod nano)
(bob, iPod shuffle)

FLATTEN

Note that it is NOT a first class function !

(that’s one thing I don’t like about Pig-latin)

- First class FLATTEN:
  - FLATTEN({{2,3},{5},{},{4,5,6}}) = {2,3,5,4,5,6}
  - Type: {{T}} → {T}
- Pig-latin FLATTEN
  - FLATTEN({4,5,6}) = 4, 5, 6
  - Type: {T} → T, T, T, …, T ?????

FILTER

Remove all queries from Web bots:

```
real_queries =  FILTER queries BY userId neq ‘bot’
```

Better: use a complex UDF to detect Web bots:

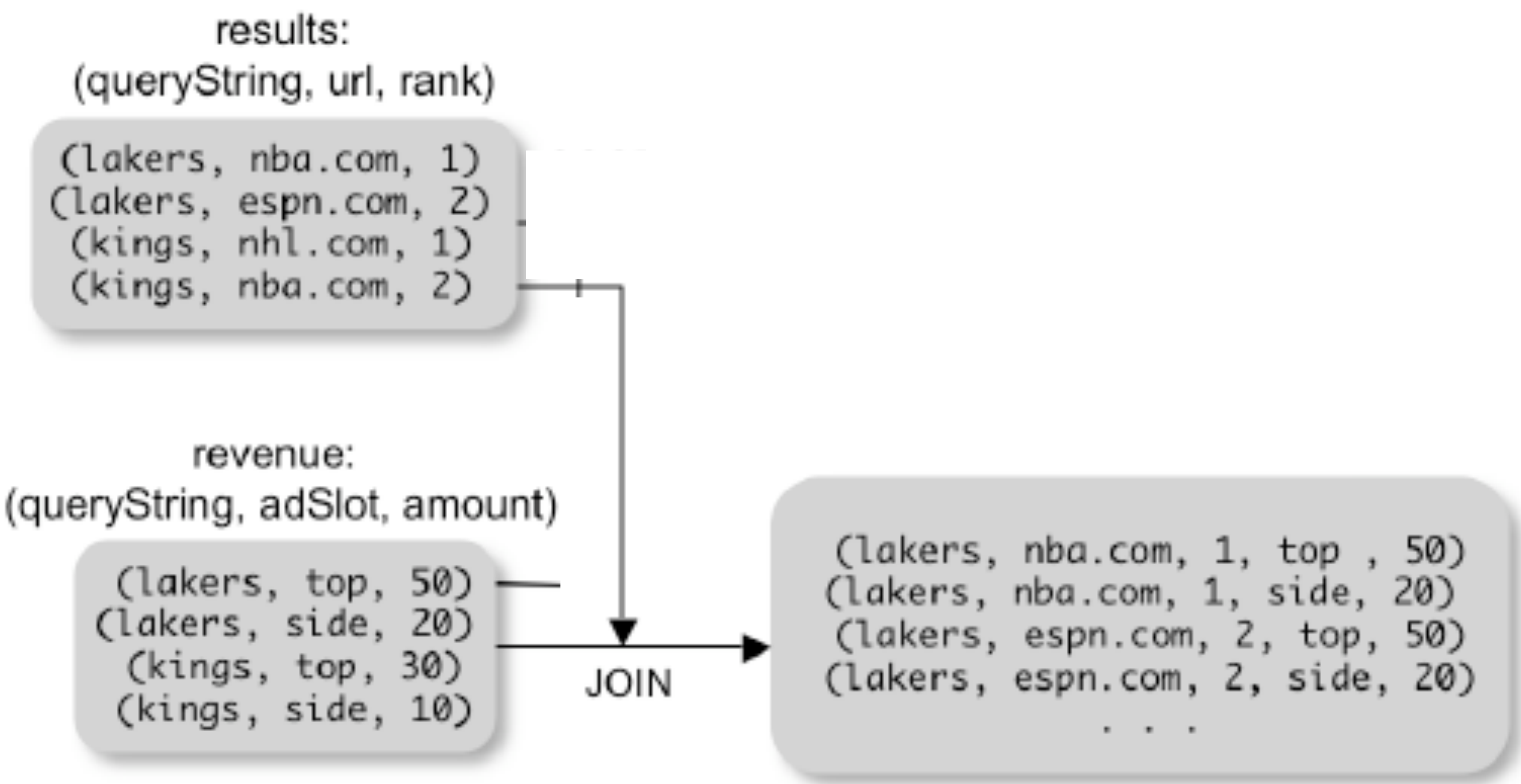
```
real_queries =  FILTER queries
                       BY NOT isBot(userId)
```

JOIN

results: {(queryString, url, position)}
revenue: {(queryString, adSlot, amount)}

```
join_result = JOIN results BY queryString
                   revenue BY queryString
```

join_result : {(queryString, url, position, adSlot, amount)}


results:
(queryString, url, rank)
(lakers, nba.com, 1)
(lakers, espn.com, 2)
(kings, nhl.com, 1)
(kings, nba.com, 2)
revenue:
(queryString, adSlot, amount)
(lakers, top, 50)
(lakers, side, 20)
(kings, top, 30)
(kings, side, 10)
JOIN
(lakers, nba.com, 1, top , 50)
(lakers, nba.com, 1, side, 20)
(lakers, espn.com, 2, top, 50)
(lakers, espn.com, 2, side, 20)
. . .

GROUP BY

revenue: {(queryString, adSlot, amount)}

```
grouped_revenue = GROUP revenue BY queryString
query_revenues =
    FOREACH grouped_revenue
    GENERATE queryString,
             SUM(revenue.amount) AS totalRevenue
```

grouped_revenue: {(queryString, {(adSlot, amount)})}
query_revenues: {(queryString, totalRevenue)}

Simple Map-Reduce

input : {(field1, field2, field3, ...)}

```
map_result = FOREACH input
                GENERATE FLATTEN(map(*))
key_groups = GROUP map_result BY $0
output = FOREACH key_groups
            GENERATE reduce($1)
```

map_result : {(a1, a2, a3, . . .)}
key_groups : {(a1, {(a2, a3, . . .)})}

Co-Group

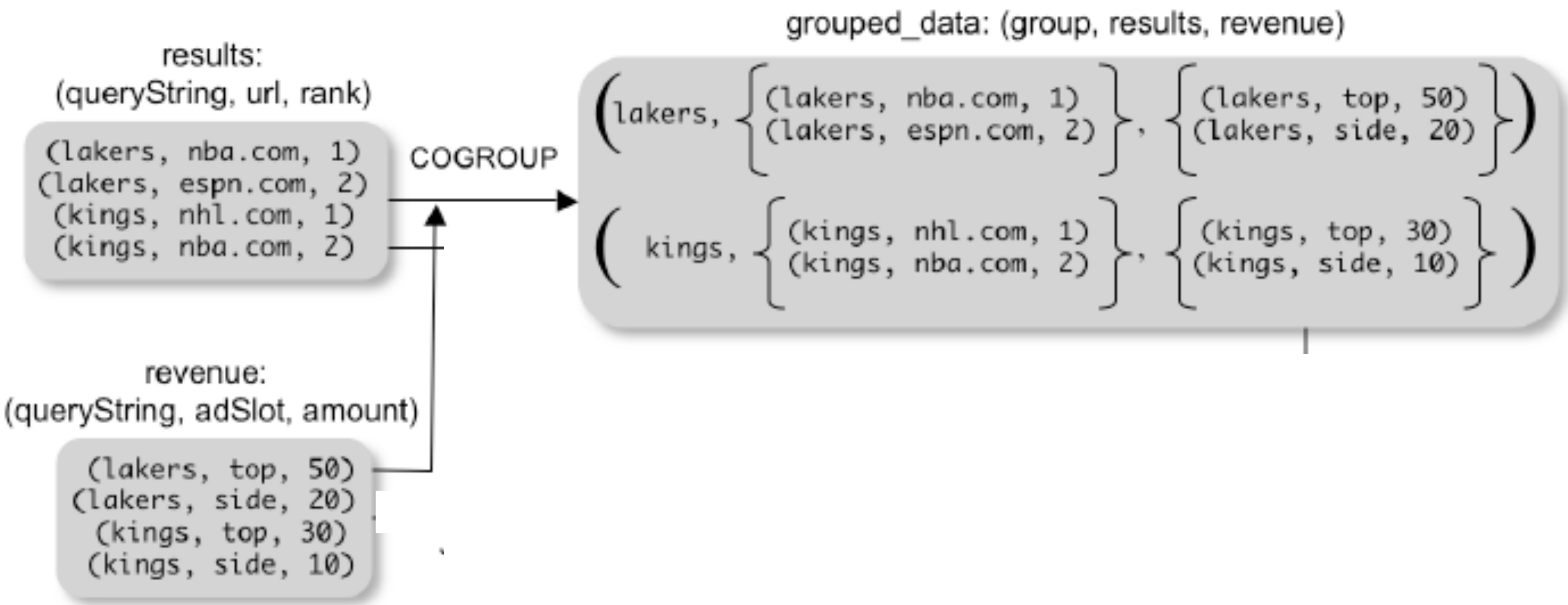
results: {(queryString, url, position)}
revenue: {(queryString, adSlot, amount)}

```
grouped_data =
      COGROUP results BY queryString,
                      revenue BY queryString;
```

grouped_data: {(queryString, results:{(url, position)},
revenue:{(adSlot, amount)})}

What is the output type in general ?

Co-Group



Is this an inner join, or an outer join ?

Co-Group

grouped_data: {(queryString, results:{(url, position)},
revenue:{(adSlot, amount)})}

```
url_revenues = FOREACH grouped_data
      GENERATE
            FLATTEN(distributeRevenue(results, revenue));
```

distributeRevenue is a UDF that accepts search results and revenue information for a query string at a time, and outputs a bag of urls and the revenue attributed to them.

Co-Group v.s. Join

grouped_data: {(queryString, results:{(url, position)},
revenue:{(adSlot, amount)})}

```
grouped_data = COGROUP results BY queryString,
                                revenue BY queryString;
join_result = FOREACH grouped_data
                   GENERATE FLATTEN(results),
                                   FLATTEN(revenue);
```

Result is the same as JOIN

Asking for Output: STORE

```
STORE query_revenues INTO `myoutput'
          USING myStore();
```

Meaning: write query_revenues to the file ‘myoutput’

Implementation

- Over Hadoop !
- Parse query:
  - Everything between LOAD and STORE → one logical plan
- Logical plan → sequence of Map/Reduce ops
- All statements between two (CO)GROUPs → one Map/Reduce op

Implementation