Lecture 4: Query Execution

Tuesday, January 28, 2014

Announcemenents

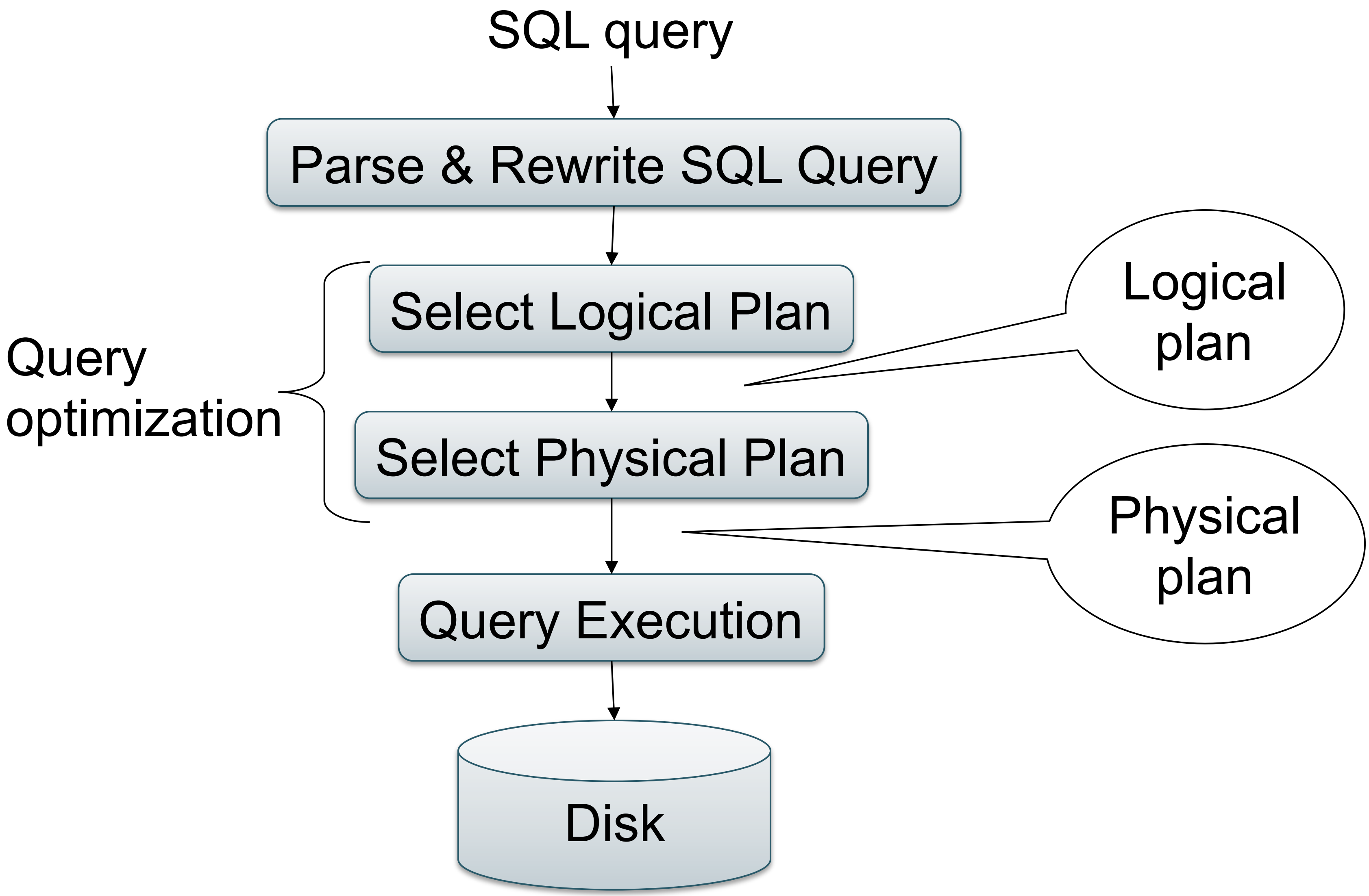
- Homework 2 was due last night
- Paper review (Shapiro) was due today
- Homework 3 is posted
  - You have received a token (=$100@AWS)
  - You need to write 4 simple queries
  - Data is huge: last query ≈ 4-7 hours
  - Learn PigLatin on your own (easy)
  - Plan a lot of time for setup

Where We Are

Query execution!

- We have seen:
  - Disk organization = set of blocks(pages)
  - The buffer pool
  - How records are organized in pages
  - Indexes, in particular B+ -trees
- Today: rest of query execution, optimization

Steps of the Query Processor



Steps in Query Evaluation

- **Step 0: Admission control**
  - User connects to the db with username, password
  - User sends query in text format
- **Step 1: Query parsing**
  - Parses query into an internal format
  - Performs various checks using catalog
    - Correctness, authorization, integrity constraints
- **Step 2: Query rewrite**
  - View rewriting, flattening, etc.

Continue with Query Evaluation

- **Step 3: Query optimization**
  - Find an efficient query plan for executing the query

- A **query plan** is
  - **Logical query plan**: an extended relational algebra tree
  - **Physical query plan**: with additional annotations at each node
    - Access method to use for each relation
    - Implementation to use for each relational operator

Final Step in Query Processing

- **Step 4: Query execution**
  - Each operator has several implementation algorithms

- Synchronization techniques:
  - Pipelined execution
  - Materialized relations for intermediate results
- Passing data between operators:
  - Iterator interface
  - One thread per operator

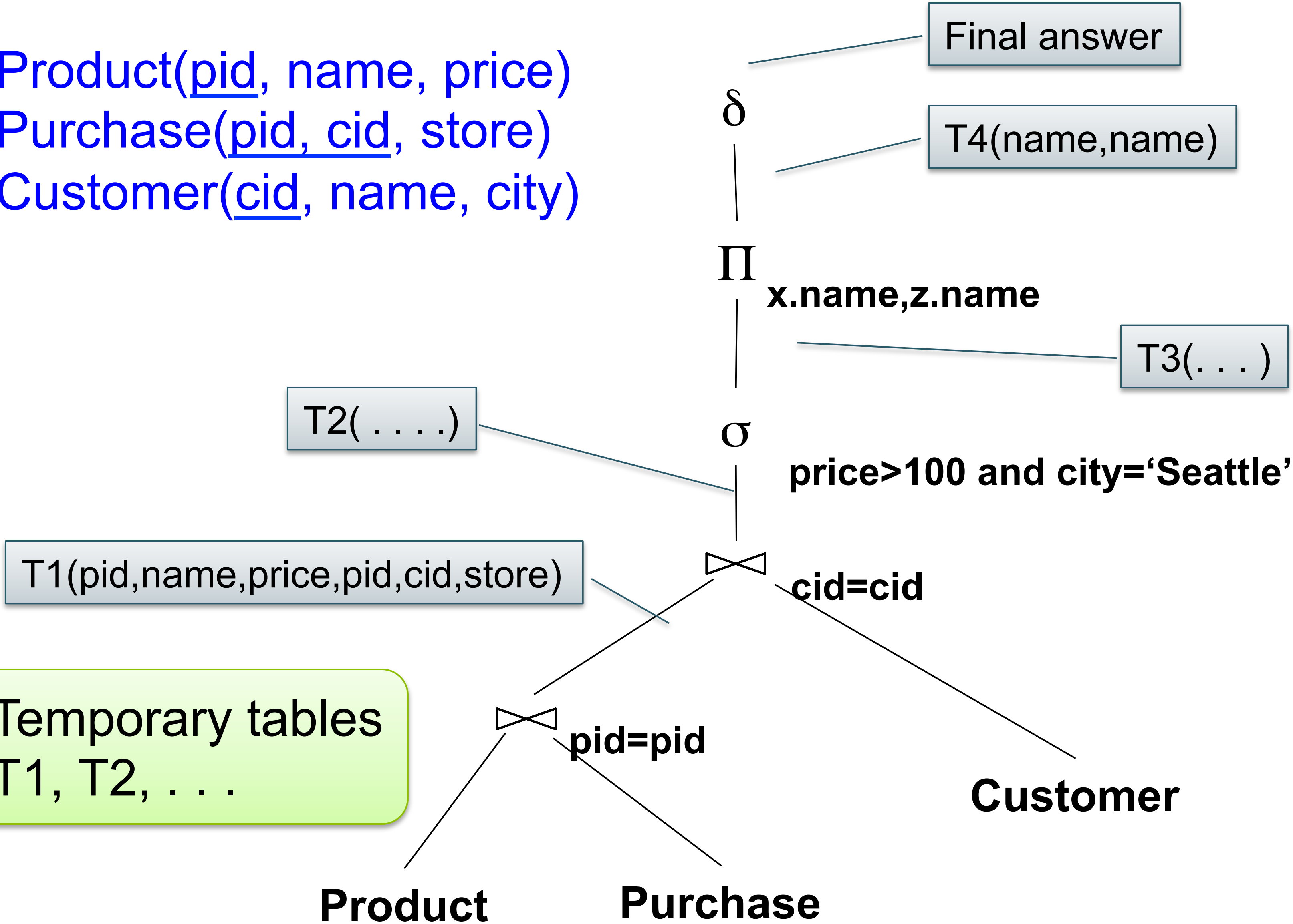
SQL Query

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

```
SELECT DISTINCT x.name, z.name
FROM Product x, Purchase y, Customer z
WHERE x.pid = y.pid and y.cid = y.cid and
        x.price > 100 and z.city = 'Seattle'
```

Logical Plan

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



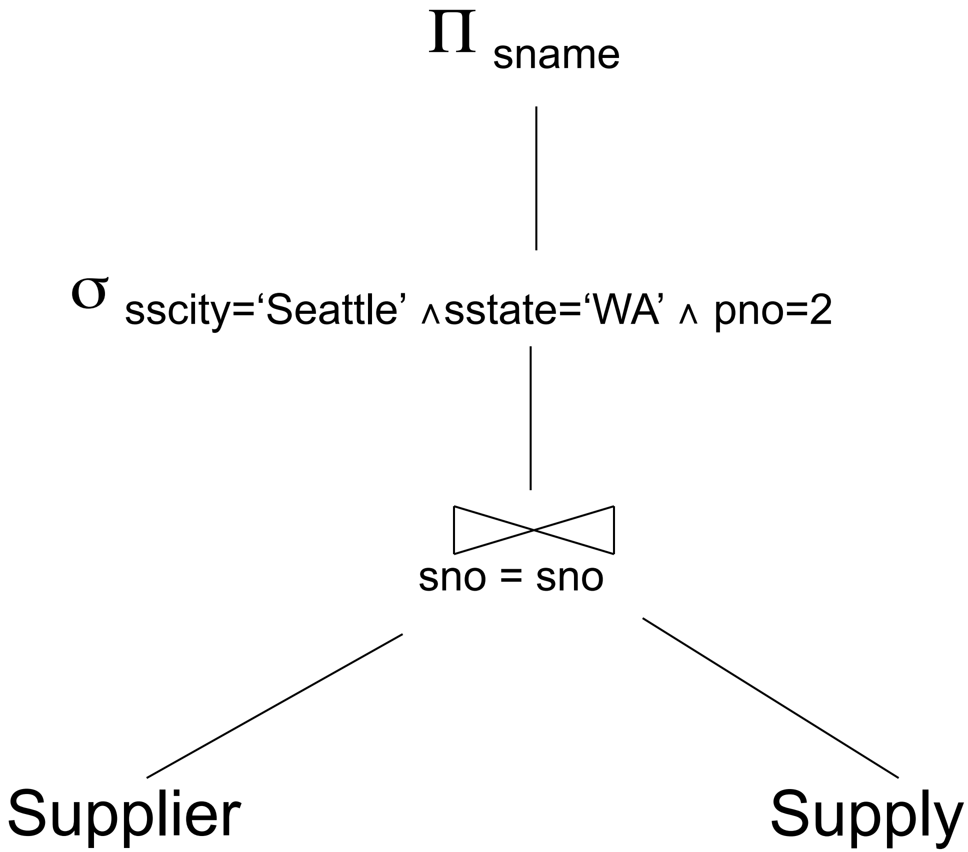
Temporary tables
T1, T2, . . .

Logical v.s. Physical Plan

- Physical plan = Logical plan plus annotations
- Access path selection for each relation
  - Use a file scan or use an index
- Implementation choice for each operator
- Scheduling decisions for operators

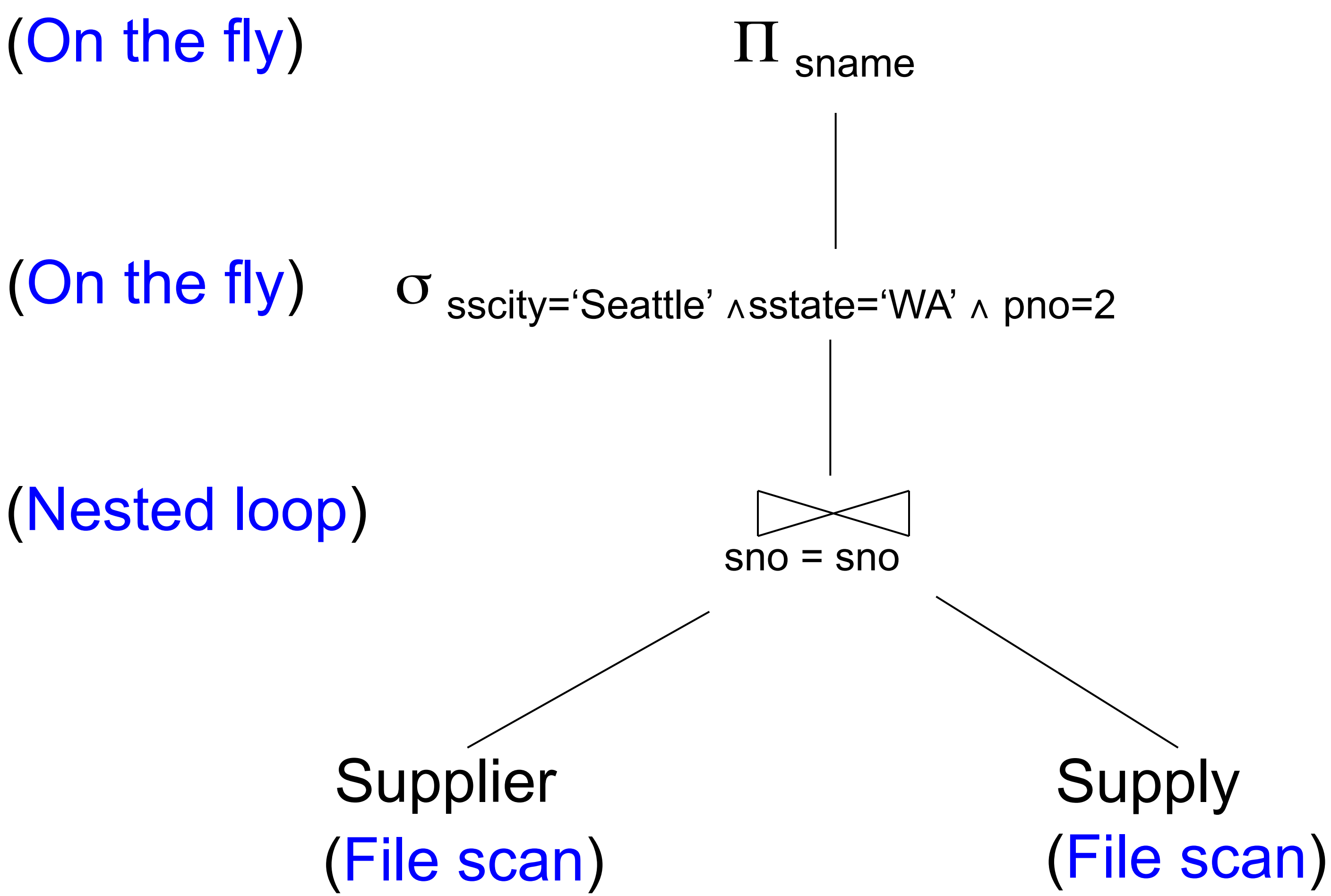
Supplier(sno, sname, scity, sstate)
Supply(sno, pno, price)

Logical Query Plan



Supplier(sno, sname, scity, sstate)
Supply(sno, pno, price)

Physical Query Plan



Outline of the Lecture

- Physical operators: join, group-by
- Query execution: pipeline, iterator model
- Query optimization
- Database statistics

Extended Algebra Operators

- Union $\cup$, difference $-$
- Selection σ
- Projection Π
- Join $\bowtie$ -- also: semi-join, anti-semi-join
- Rename ρ
- Duplicate elimination δ
- Grouping and aggregation γ
- Sorting τ

Basic RA

ExtendedRA

Sets v.s. Bags

- Sets: {a,b,c}, {a,d,e,f}, { }, . . .
- Bags: {a, a, b, c}, {b, b, b, b, b}, . . .

Relational Algebra has two semantics:

- Set semantics (paper “Three languages…”)
- Bag semantics

Physical Operators

Each of the logical operators may have one or more implementations = physical operators

Will discuss several basic physical operators, with a focus on join

Supply(<u>sno, pno</u>, price)
Part(<u>pno</u>, pname, psize, pcolor)

Question in Class

Logical operator:

Supply(sno,pno,price) $\bowtie_{pno=pno}$ **Part(pno,pname,psize,pcolor)**

Propose three physical operators for the join, assuming the tables are in main memory:

1.
2.
3.

Supply(sno, pno, price)
Part(pno, pname, psize, pcolor)

Question in Class

Logical operator:

Supply(sno,pno,price) $\bowtie_{pno=pno}$ **Part(pno,pname,psize,pcolor)**

Propose three physical operators for the join, assuming the tables are in main memory:

1. Nested Loop Join
2. Merge join
3. Hash join

BRIEF Review of Hash Tables

A (naïve) hash function:

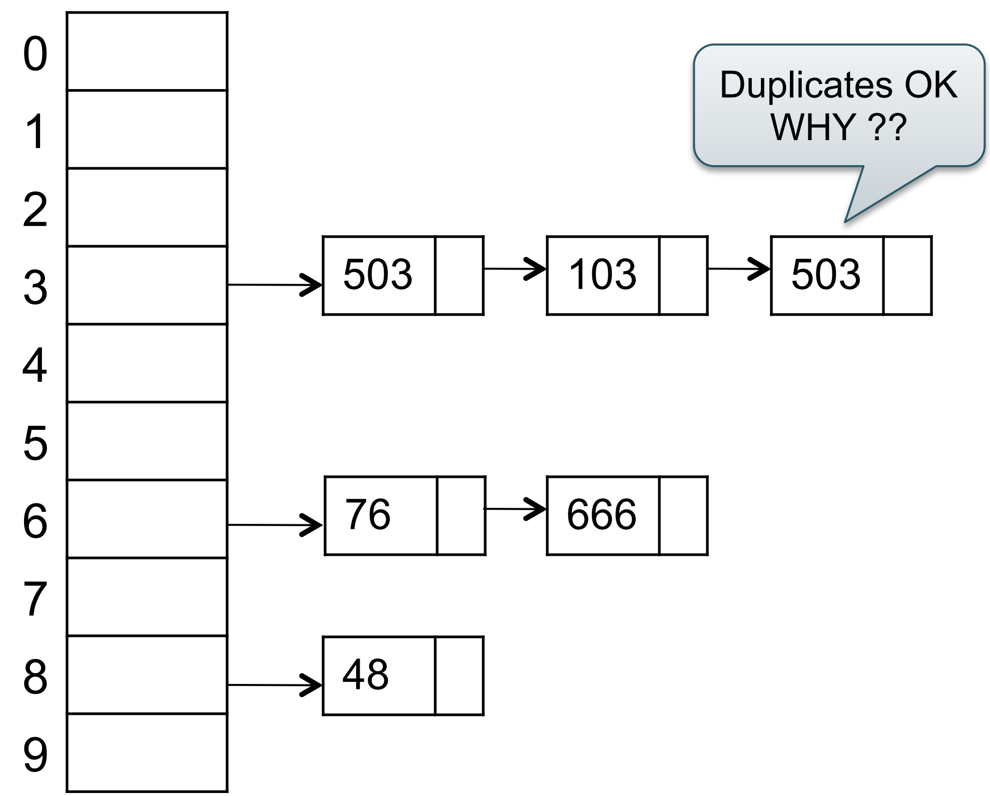
$$h(x) = x \bmod 10$$

Operations:

find(103) = ??

insert(488) = ??

Separate chaining:



BRIEF Review of Hash Tables

- insert(k, v) = inserts a key k with value v
- Many values for one key
  - Hence, duplicate k's are OK
- find(k) = returns the ***list*** of all values v associated to the key k

Cost Parameters

The *cost* of an operation = total number of I/Os

Cost parameters (used both in the book and by Shapiro):

- B(R) = number of blocks for relation R (Shapiro: |R|)
- T(R) = number of tuples in relation R
- V(R, a) = number of distinct values of attribute a
- M = size of main memory buffer pool, in blocks

Facts: (1) B(R) << T(R):
(2) When a is a key, V(R,a) = T(R)
When a is not a key, V(R,a) << T(R)

Cost of an Operator

Assumption: runtime dominated by # of disk I/O's; will ignore the main memory part of the runtime

- If R (and S) fit in main memory, then we use a main-memory algorithm
- If R (or S) does not fit in main memory, then we use an external memory algorithm

Ad-hoc Convention

- The operator *reads* the data from disk
  - Note: different from Shapiro
- The operator *does not write* the data back to disk (e.g.: pipelining)
- Thus:

Any main memory join algorithms for R $\bowtie$ S: Cost = B(R)+B(S)

Any main memory grouping γ(R): Cost = B(R)

Nested Loop Joins

- Tuple-based nested loop $R \bowtie S$

```
for each tuple r in R do
  for each tuple s in S do
    if r and s join then output (r,s)
```

R=outer relation
S=inner relation

- Cost: T(R) B(S)

Examples

M = 4

- Example 1:
  - B(R) = 1000, T(R) = 10000
  - B(S) = 2, T(S) = 20
  - Cost = ?

Can you do better with nested loops?

- Example 2:
  - B(R) = 1000, T(R) = 10000
  - B(S) = 4, T(S) = 40
  - Cost = ?

Block-Based Nested-loop Join

```
for each (M-2) blocks bs of S do
   for each block br of R do
        for each tuple s in bs
             for each tuple r in br do
                  if "r and s join" then output(r,s)
```

Terminology alert: sometimes S is called S the *inner* relation

Block-Based Nested-loop Join

Why not M ?

```
for each (M-2) blocks bs of S do
   for each block br of R do
      for each tuple s in bs
         for each tuple r in br do
            if "r and s join" then output(r,s)
```

Terminology alert: sometimes S is called S the *inner* relation

Block-Based Nested-loop Join

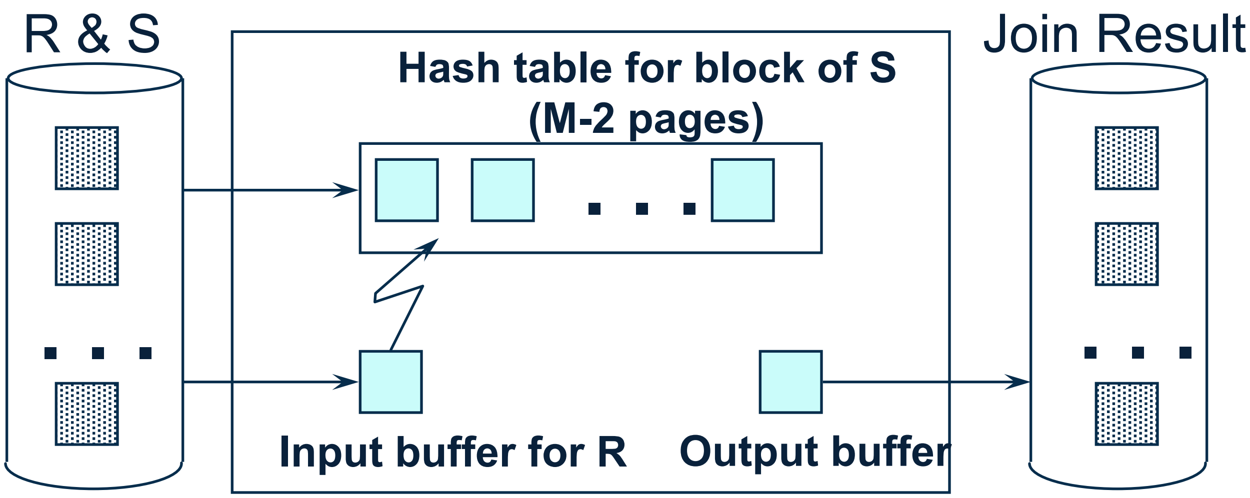
Why not M ?

<u>for</u> each (M-2) blocks **bs** of **S** <u>do</u>
    <u>for</u> each block **br** of **R** <u>do</u>
        <u>for</u> each tuple **s** in **bs**
            <u>for</u> each tuple **r** in **br** <u>do</u>
                <u>if</u> "**r** and **s** join" <u>then</u> output(**r**,**s**)

Better: main memory hash join

Terminology alert: sometimes S is called S the *inner* relation

Block Nested-loop Join



Examples

M = 4

- Example 1:
  - B(R) = 1000, T(R) = 10000
  - B(S) = 2, T(S) = 20
  - Cost = B(S) + B(R) = 1002

- Example 2:
  - B(R) = 1000, T(R) = 10000
  - B(S) = 4, T(S) = 40
  - Cost = B(S) + 2B(R) = 2004

Note: T(R) and T(S) are irrelevant here.

Cost of Block Nested-loop Join

- Read S once: cost B(S)
- Outer loop runs B(S)/(M-2) times, and each time need to read R: costs B(S)B(R)/(M-2)

Cost = B(S) + B(S)B(R)/(M-2)

Index Based Selection

Recall IMDB; assume indexes on Movie.id, Movie.year

```
SELET *
FROM Movie
WHERE id = ‘12345’
```

```
SELET *
FROM Movie
WHERE year = ‘1995’
```

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

What is your estimate of the I/O cost ?

Index Based Selection

Selection on equality: $\sigma_{a=v}(R)$

- Clustered index on a: cost ?
- Unclustered index : cost ?

Index Based Selection

Selection on equality: $\sigma_{a=v}(R)$

- Clustered index on a: cost B(R)/V(R,a)
- Unclustered index : cost T(R)/V(R,a)

Index Based Selection

Selection on equality: $\sigma_{a=v}(R)$

- Clustered index on a: cost B(R)/V(R,a)
- Unclustered index : cost T(R)/V(R,a)

Note: we assume that the cost of reading the index = 0
Why?

Index Based Selection

- Example:

$B(R) = 10k$
$T(R) = 1M$
$V(R, a) = 100$

cost of $\sigma_{a=v}(R)$ = ?

- Table scan:
  - B(R) = 10k I/Os
- Index based selection:
  - If index is clustered: B(R)/V(R,a) = 100 I/Os
  - If index is unclustered: T(R)/V(R,a) = 10000 I/Os

Rule of thumb:
don't build unclustered indexes when V(R,a) is small !

Index Based Join

- $R \bowtie S$
- Assume S has an index on the join attribute

```
for each tuple r in R do
  lookup the tuple(s) s in S using the index
  output (r,s)
```

Index Based Join

Cost:

- If index is clustered:
- If unclustered:

Index Based Join

Cost:

- If index is clustered: $B(R) + T(R)B(S)/V(S,a)$
- If unclustered: $B(R) + T(R)T(S)/V(S,a)$

Operations on Very Large Tables

- Compute R $\bowtie$ S when each is larger than main memory
- Two methods:
  - Partitioned hash join (many variants)
  - Merge-join
- Similar for grouping

External Sorting

- Problem:
- Sort a file of size B with memory M
- Where we need this:
  - ORDER BY in SQL queries
  - Several physical operators
  - Bulk loading of B+-tree indexes.
- Will discuss only 2-pass sorting, when $B < M^2$

Basic Terminology

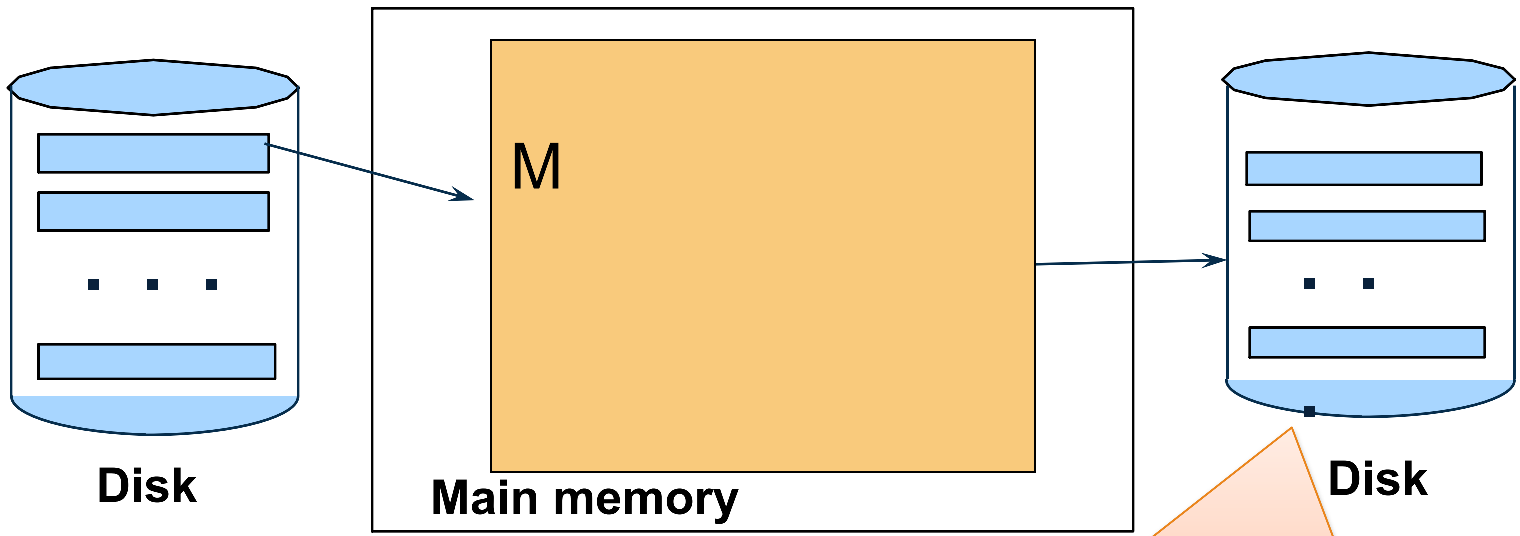
- A run in a sequence is an increasing subsequence

- What are the runs?

  2, 4, 99, 103, 88, 77, 3, 79, 100, 2, 50

External Merge-Sort: Step 1

- Phase one: load M bytes in memory, sort



Runs of length M bytes

Can increase to length 2M using "replacement selection"

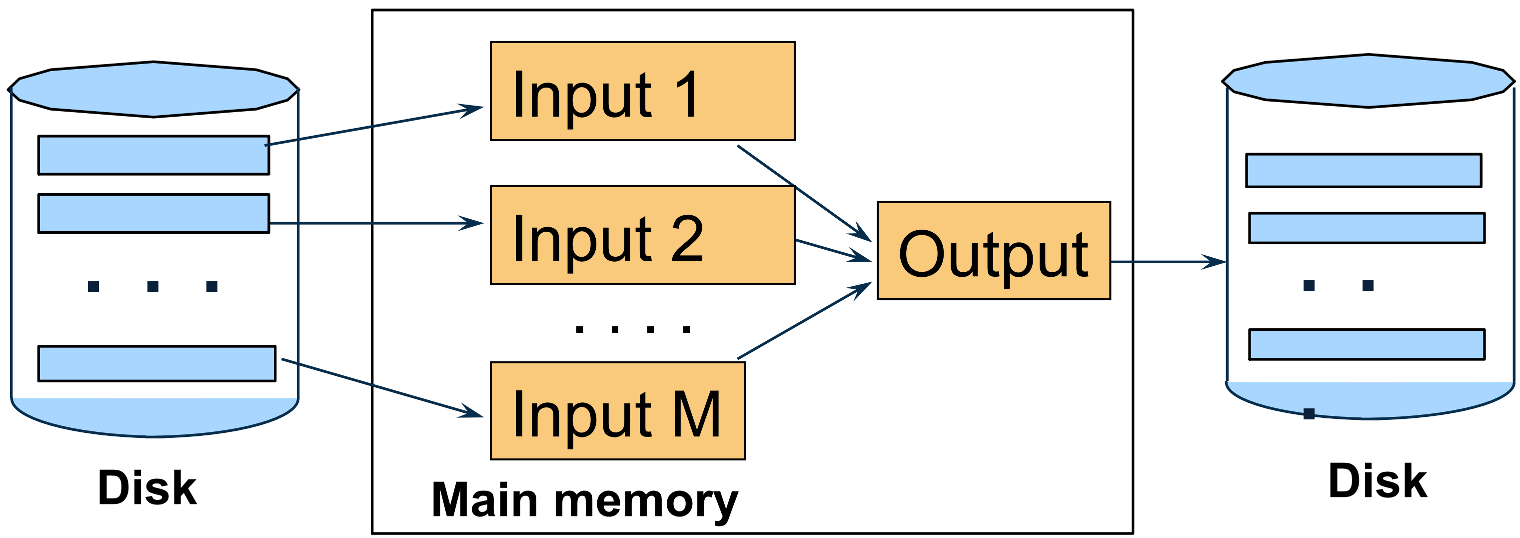
Basic Terminology

- Merging multiple runs to produce a longer run:
  0, **14, 33, 88, 92, 192, 322**
  2, 4, 7, **43, 78, 103, 523**
  1, 6, **9, 12, 33, 52, 88, 320**

  Output:
  0, 1, 2, 4, 6, 7, ?

External Merge-Sort: Step 2

- Merge M – 1 runs into a new run
- Result: runs of length M (M – 1)$\approx$ M^2



If B <= M^2 then we are done

Cost of External Merge Sort

- Read+write+read = 3B(R)
- Assumption: B(R) <= M^2

Group-by

Group-by: $\gamma_{a,\ sum(b)}$ (R)

- Idea: do a two step merge sort, but change one of the steps

- Question in class: which step needs to be changed and how ?

Cost = 3B(R)
Assumption: B(δ(R)) <= M^2

Merge-Join

Join R ⋈ S

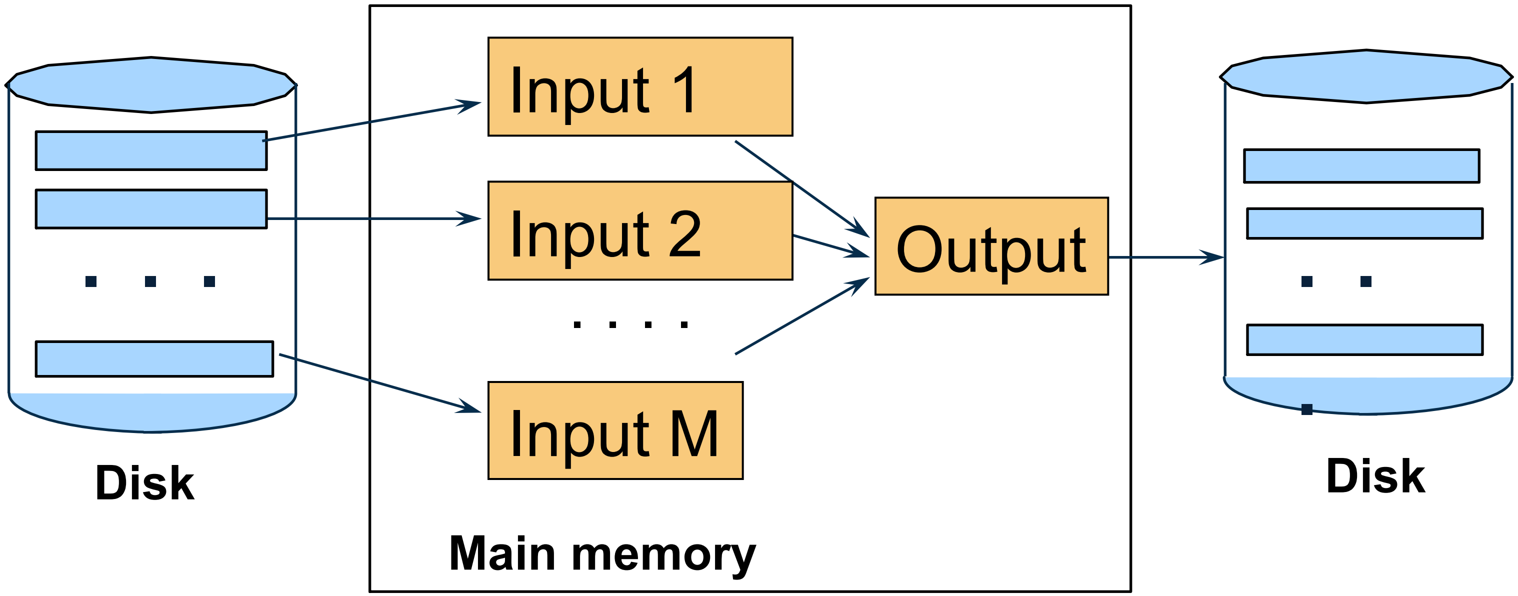
- How?....

Merge-Join

Join R $\bowtie$ S

- Step 1a: initial runs for R
- Step 1b: initial runs for S
- Step 2: merge and join

Merge-Join



$M_1 = B(R)/M$ runs for R

$M_2 = B(S)/M$ runs for S

Merge-join $M_1 + M_2$ runs;

need $M_1 + M_2 <= M$

Partitioned Hash Algorithms

Idea:

- If $B(R) > M$, then partition it into smaller files:
  $R1, R2, R3, \ldots, Rk$

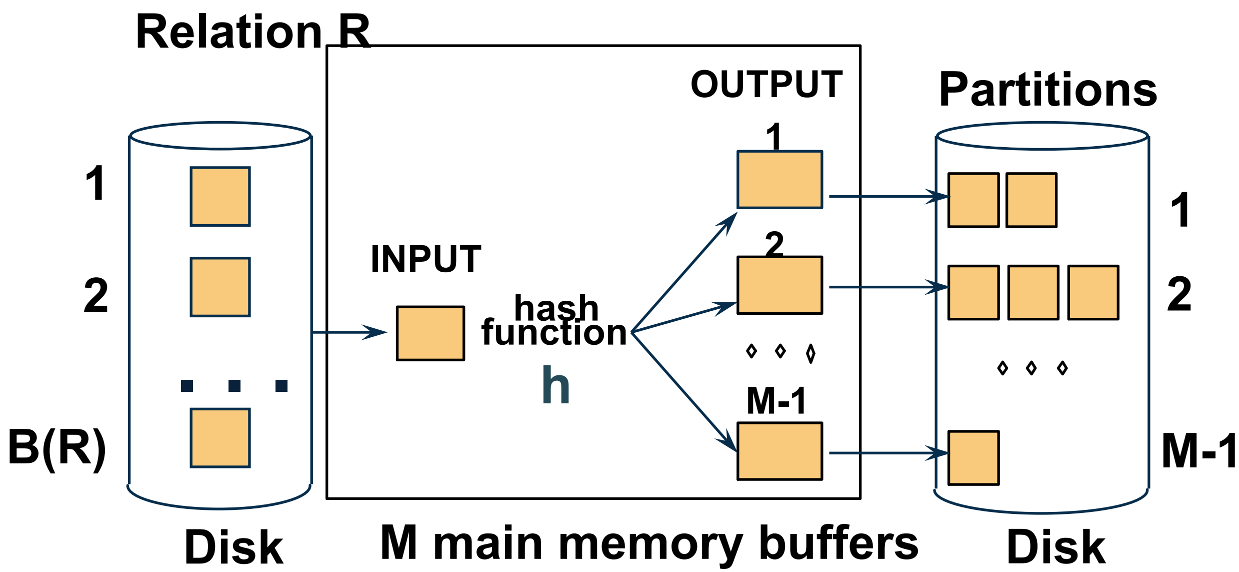
- Assuming $B(R1)=B(R2)=\ldots= B(Rk)$, we have
  $B(Ri) = B(R)/k$

- Goal: each Ri should fit in main memory:
  $B(Ri) \leq M$

How big can k be ?

Partitioned Hash Algorithms

- Idea: partition a relation R into M-1 buckets, on disk
- Each bucket has size approx. B(R)/(M-1) ≈ B(R)/M



Assumption: B(R)/M ≤ M, i.e. B(R) ≤ M^2

Grouping

- γ(R) = grouping and aggregation
- Step 1. Partition R into buckets
- Step 2. Apply γ to each bucket (may read in main memory)

- Cost: 3B(R)
- Assumption: $B(R) \leq M^2$

Grace-Join

R ⋈ S

Note: grace-join is
also called
partitioned hash-join

Grace-Join

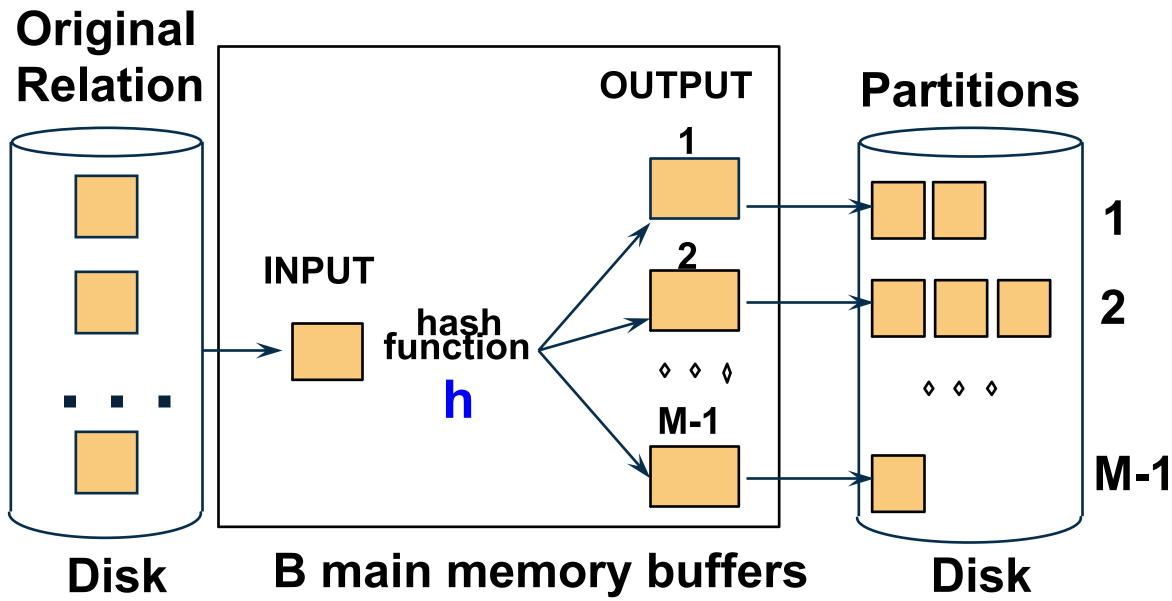
$R \bowtie S$

- Step 1:
  - Hash S into M buckets
  - send all buckets to disk
- Step 2
  - Hash R into M buckets
  - Send all buckets to disk
- Step 3
  - Join every pair of buckets

Note: grace-join is also called *partitioned hash-join*

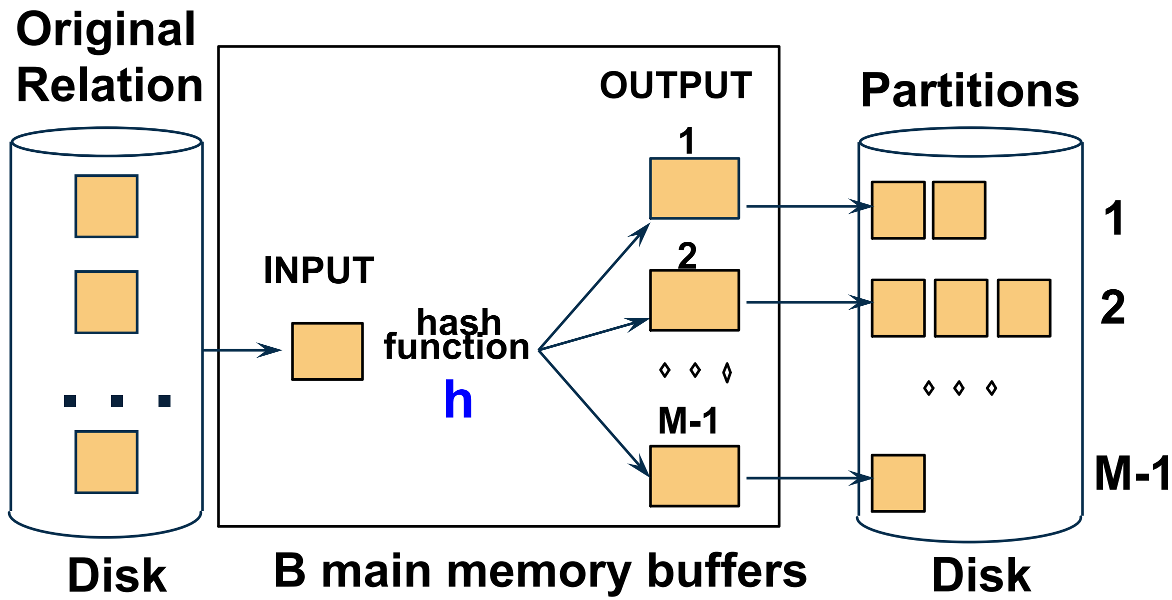
Grace-Join

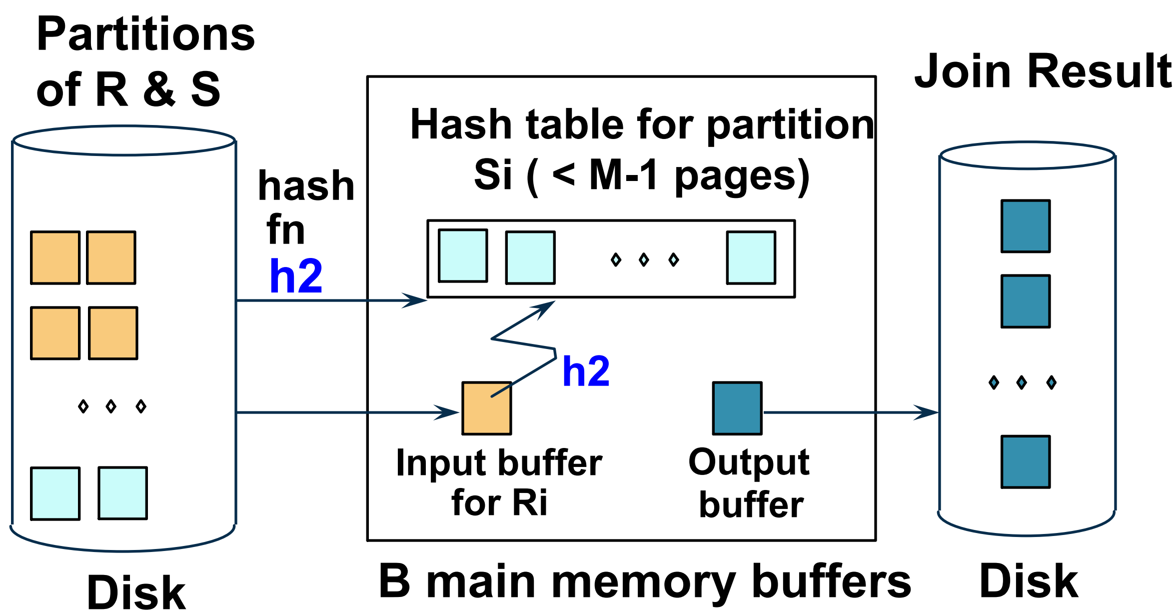
- Partition both relations using hash fn **h**: R tuples in partition i will only match S tuples in partition i.



Grace-Join

- Partition both relations using hash fn **h**: R tuples in partition i will only match S tuples in partition i.





- Read in a partition of R, hash it using **h2 (<> h!)**. Scan matching partition of S, search for matches.

Grace Join

- Cost: 3B(R) + 3B(S)
- Assumption: min(B(R), B(S)) <= M^2

Hybrid Hash Join Algorithm

- How does it work?

Hybrid Hash Join Algorithm

- Partition S into k buckets

  t buckets S_1 , …, S_t stay in memory

  k-t buckets S_{t+1}, …, S_k to disk
- Partition R into k buckets
  - First t buckets join immediately with S
  - Rest k-t buckets go to disk
- Finally, join k-t pairs of buckets:

  (R_{t+1},S_{t+1}), (R_{t+2},S_{t+2}), …, (R_k,S_k)

Hybrid Hash Join Algorithm

- Partition S into k buckets

  t buckets S_1 , …, S_t stay in memory

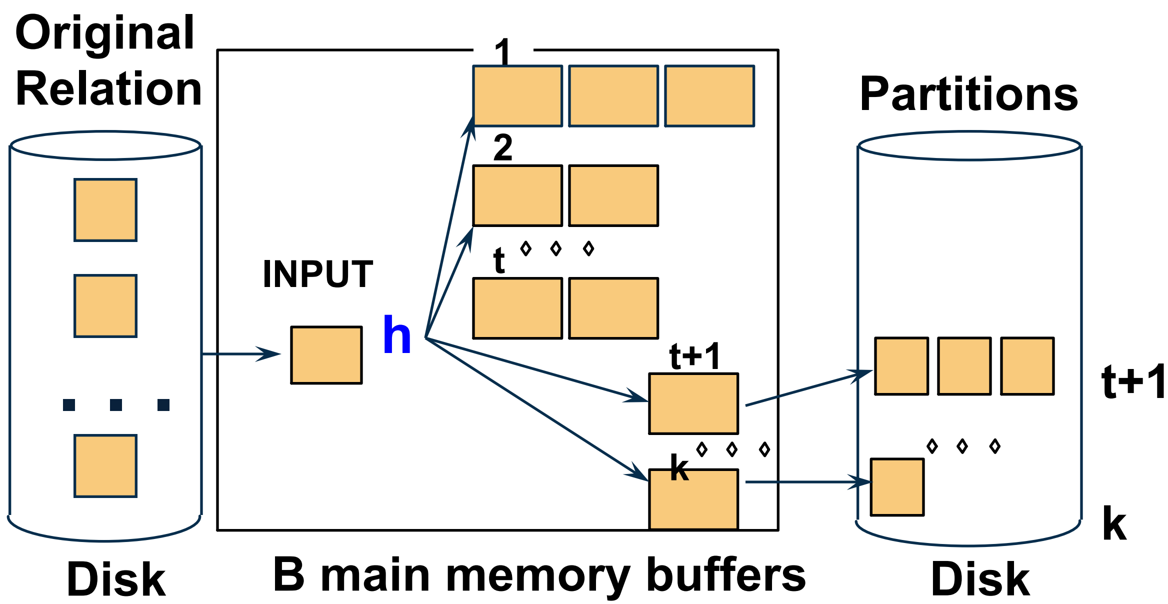
  k-t buckets S_{t+1}, …, S_k to disk

- Partition R into k buckets
  - First t buckets join immediately with S
  - Rest k-t buckets go to disk
- Finally, join k-t pairs of buckets:

  (R_{t+1},S_{t+1}), (R_{t+2},S_{t+2}), …, (R_k,S_k)

Shapiro's notation:
1/(B+1) = t/k in main memory
B/(B+1) = (k-t)/k go to disk

Hybrid Hash Join Algorithm



Hybrid Join Algorithm

- How to choose k and t ?
  - Choose k large but s.t. k <= M
  - Choose t/k large but s.t. t/k * B(S) <= M
  - Moreover: t/k * B(S) + k-t <= M

- Assuming t/k * B(S) >> k-t: t/k = M/B(S)

Hybrid Join Algorithm

Cost of Hybrid Join:

- Grace join: 3B(R) + 3B(S)
- Hybrid join:
  - Saves 2 I/Os for t/k fraction of buckets
  - Saves 2t/k(B(R) + B(S)) I/Os
  - Cost:
    (3-2t/k)(B(R) + B(S)) = (3-2M/B(S))(B(R) + B(S))

Hybrid Join Algorithm

- Question in class: what is the advantage of the hybrid algorithm ?

Summary of External Join Algorithms

- Block Nested Loop: B(S) + B(R)*B(S)/M
- Index Join: B(R) + T(R)B(S)/V(S,a)
- Partitioned Hash: 3B(R)+3B(S);
  - min(B(R),B(S)) <= M^2
- Merge Join: 3B(R)+3B(S)
  - B(R)+B(S) <= M^2

Other Operators

- Selection, projection
- Duplicate elimination
- Semi-join
- Anti-semijoin

Selections, Projections

- Selection = easy, check condition on each tuple at a time
- Projection = easy (assuming no duplicate elimination), remove extraneous attributes from each tuple

Duplicate Elimination IS Group By

Duplicate elimination $\delta(R)$ is *the same* as group by $\gamma(R)$ WHY ???

- Hash table in main memory

- Cost: B(R)
- Assumption: $B(\delta(R)) <= M$

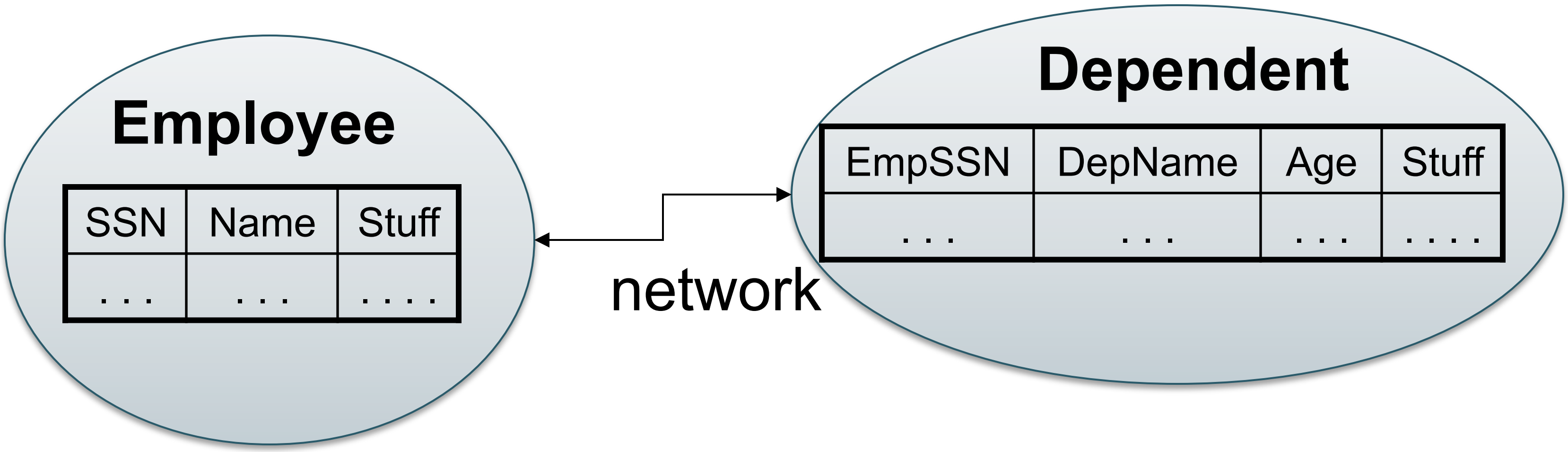
Semijoin

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

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

Formally, $R \ltimes_C S$ 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

Semijoins in Distributed Databases



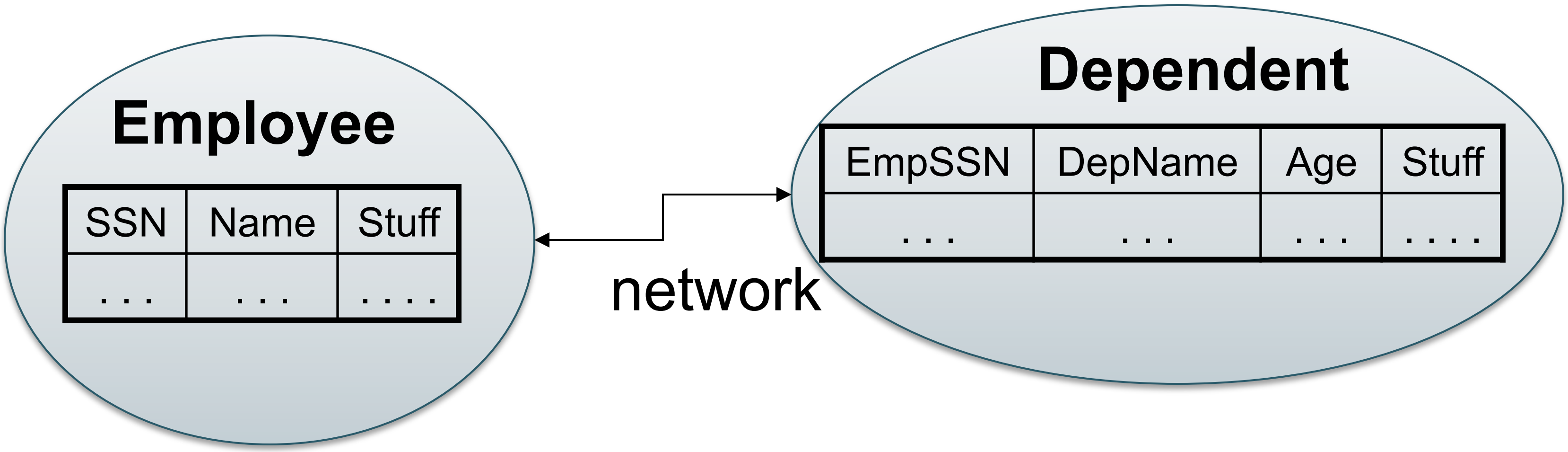
$$\text{Employee} \bowtie_{\text{SSN=EmpSSN}} (\sigma_{\text{age}>71} (\text{Dependent}))$$

Assumptions

- Very few dependents have age > 71.
- “Stuff” is big.

Task: compute the query with minimum amount of data transfer

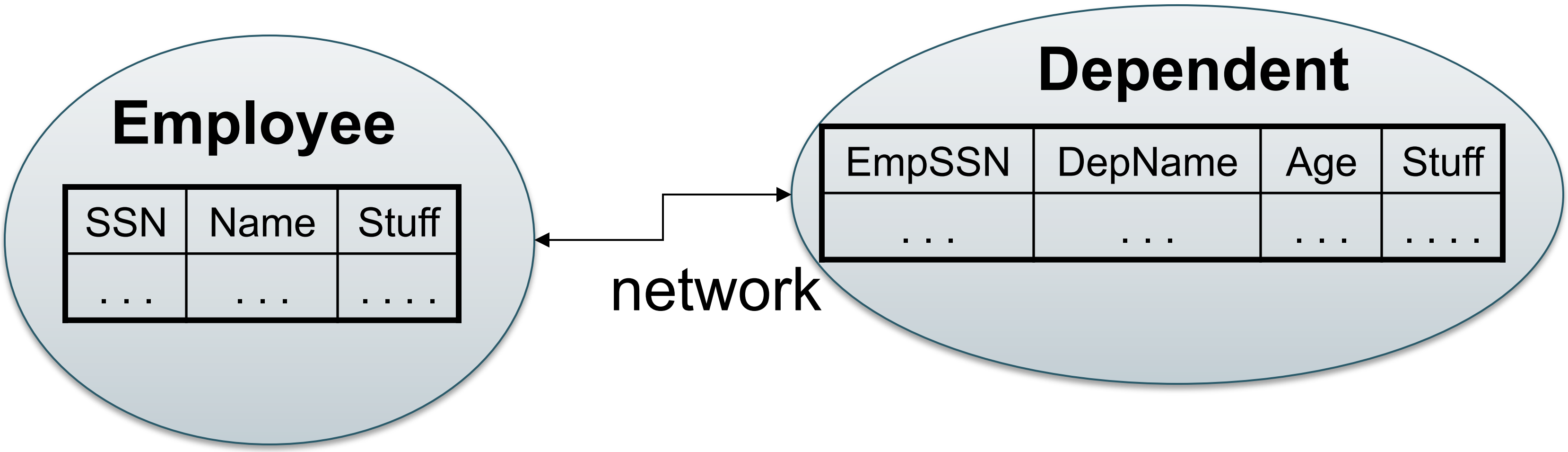
Semijoins in Distributed Databases



$$\text{Employee} \ltimes_{SSN=EmpSSN} (\sigma_{age>71}(\text{Dependent}))$$

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

Semijoins in Distributed Databases

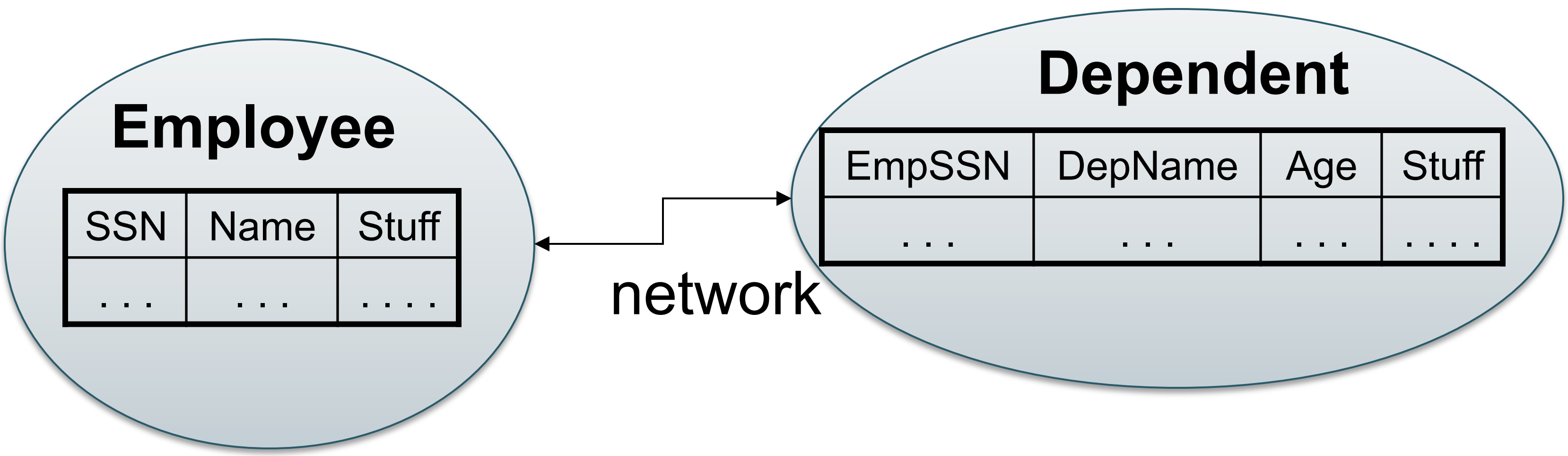


$$\text{Employee} \bowtie_{SSN=EmpSSN} (\sigma_{age>71} (\text{Dependent}))$$

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

$$R = \text{Employee} \bowtie_{SSN=EmpSSN} T$$
$$= \text{Employee} \ltimes_{SSN=EmpSSN} (\sigma_{age>71} (\text{Dependents}))$$

Semijoins in Distributed Databases



$$\text{Employee} \bowtie_{SSN=EmpSSN} (\sigma_{age>71} (\text{Dependent}))$$

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

$$R = \text{Employee} \ltimes_{SSN=EmpSSN} T$$

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

Anti-Semi-Join

- Notation: R ▷ 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=

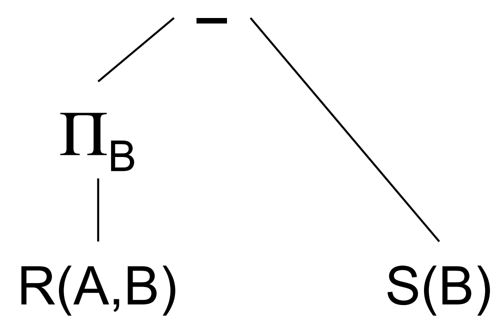
```
SELECT DISTINCT *
FROM R
WHERE not exists  (SELECT *
                   FROM S
                   WHERE R.B=S.B)
```

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=



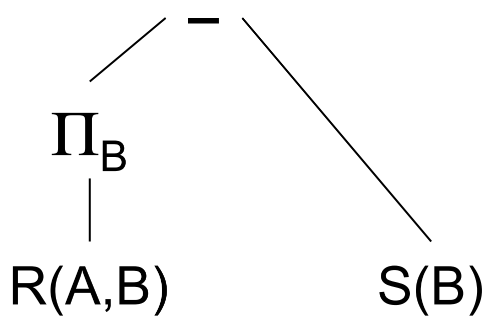

```
SELECT DISTINCT *
FROM R
WHERE not exists  (SELECT *
                   FROM S
                   WHERE R.B=S.B)
```

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=

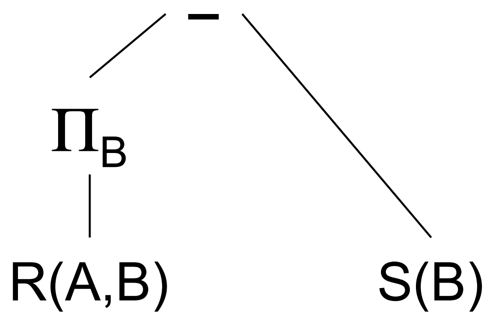
```
SELECT DISTINCT *
FROM R
WHERE not exists  (SELECT *
                   FROM S
                   WHERE R.B=S.B)
```

R(A,B)
S(B)

Set Difference v.s. Anti-semijoin

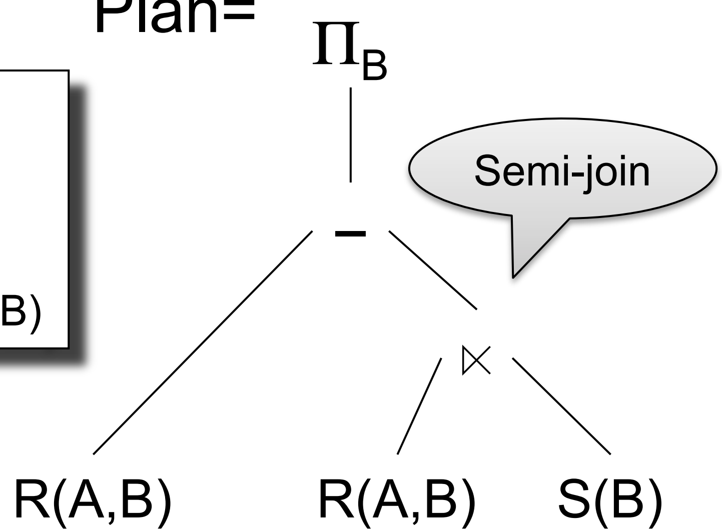
```
SELECT DISTINCT R.B
FROM R
WHERE not exists  (SELECT *
                   FROM S
                   WHERE R.B=S.B)
```

Plan=




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

Plan=

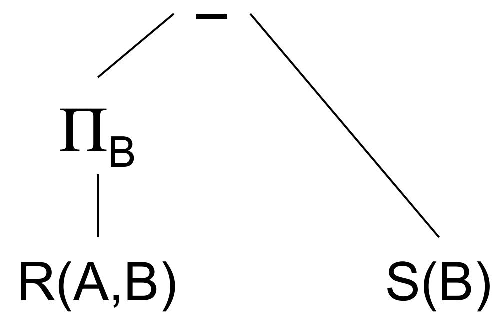


R(A,B)
S(B)

Set Difference v.s. Anti-semijoin

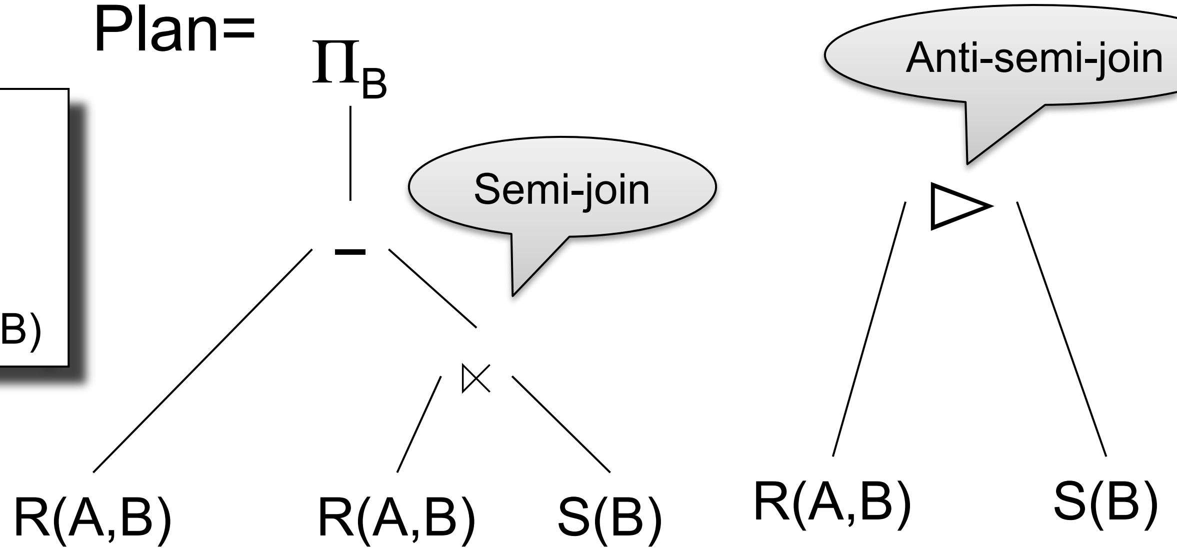
```
SELECT DISTINCT R.B
FROM R
WHERE not exists  (SELECT *
                   FROM S
                   WHERE R.B=S.B)
```

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

Outline of the Lecture

- Physical operators: join, group-by
- Query execution: pipeline, iterator model
- Query optimization
- Database statistics

Iterator Interface

- Each **operator implements this interface**
- Interface has only three methods
- **open()**
  - Initializes operator state
  - Sets parameters such as selection condition
- **get_next()**
  - Operator invokes get_next() recursively on its inputs
  - Performs processing and produces an output tuple
- **close()**: cleans-up state

Supplier(<u>sno</u>, sname, scity, sstate)
Supply(<u>sno, pno</u>, price)
Part(<u>pno</u>, pname, psize, pcolor)

1. Nested Loop Join

```
for S in Supply do {
   for  P in Part do {
        if (S.pno == P.pno) output(S,P);
   }
}
```

Supply = *outer relation*
Part = *inner relation*
Note: sometimes terminology is switched

Would it be more efficient to choose Part=outer, Supply=inner?
What if we had an index on Part.pno ?

It's more complicated…

- Each **operator implements this interface**
- **open()**
- **get_next()**
- **close()**

Supplier(<u>sno</u>, sname, scity, sstate)
Supply(<u>sno, pno</u>, price)
Part(<u>pno</u>, pname, psize, pcolor)

Main Memory Nested Loop Join

```
open ( ) {
   Supply.open( );
   Part.open( );
   S = Supply.get_next( );
}
```

```
close ( ) {
   Supply.close ( );
   Part.close ( );
}
```

```
get_next( ) {
   repeat {
      P= Part.get_next( );
      if (P== NULL)
         { Part.close();
            S= Supply.get_next( );
            if (S== NULL) return NULL;
            Part.open( );
            P= Part.get_next( );
         }
   until (S.pno == P.pno);
   return (S, P)
}
```

ALL operators need to be implemented this way !

Supplier(<u>sno</u>, sname, scity, sstate)
Supply(<u>sno, pno</u>, price)
Part(<u>pno</u>, pname, psize, pcolor)

2. Hash Join (main memory)

Build phase

```
for S in Supply do  insert(S.pno, S);

for P in Part do {
   LS = find(P.pno);
   for S in LS do { output(S, P); }
}
```

Probing

Supply=outer
Part=inner

Recall: need to rewrite as open, get_next, close

Supplier(sno, sname, scity, sstate)
Supply(sno, pno, price)
Part(pno, pname, psize, pcolor)

3. Merge Join (main memory)

```
Part1 = sort(Part, pno);
Supply1 = sort(Supply,pno);
P=Part1.get_next(); S=Supply1.get_next();

While (P!=NULL and S!=NULL) {
    case:
      P.pno < S.pno:    P = Part1.get_next( );
      P.pno > S.pno:    S = Supply1.get_next();
      P.pno == S.pno { output(P,S);
                       S = Supply1.get_next();
                     }
}
```

Why ???

Supplier(<u>sno</u>, sname, scity, sstate)
Supply(<u>sno, pno</u>, price)

Pipelined Execution

(On the fly) Π_{sname}

(On the fly) $\sigma_{\text{sscity='Seattle'} \wedge \text{sstate='WA'} \wedge \text{pno=2}}$

(Nested loop) $\bowtie_{\text{sno = sno}}$

Supplier (File scan)

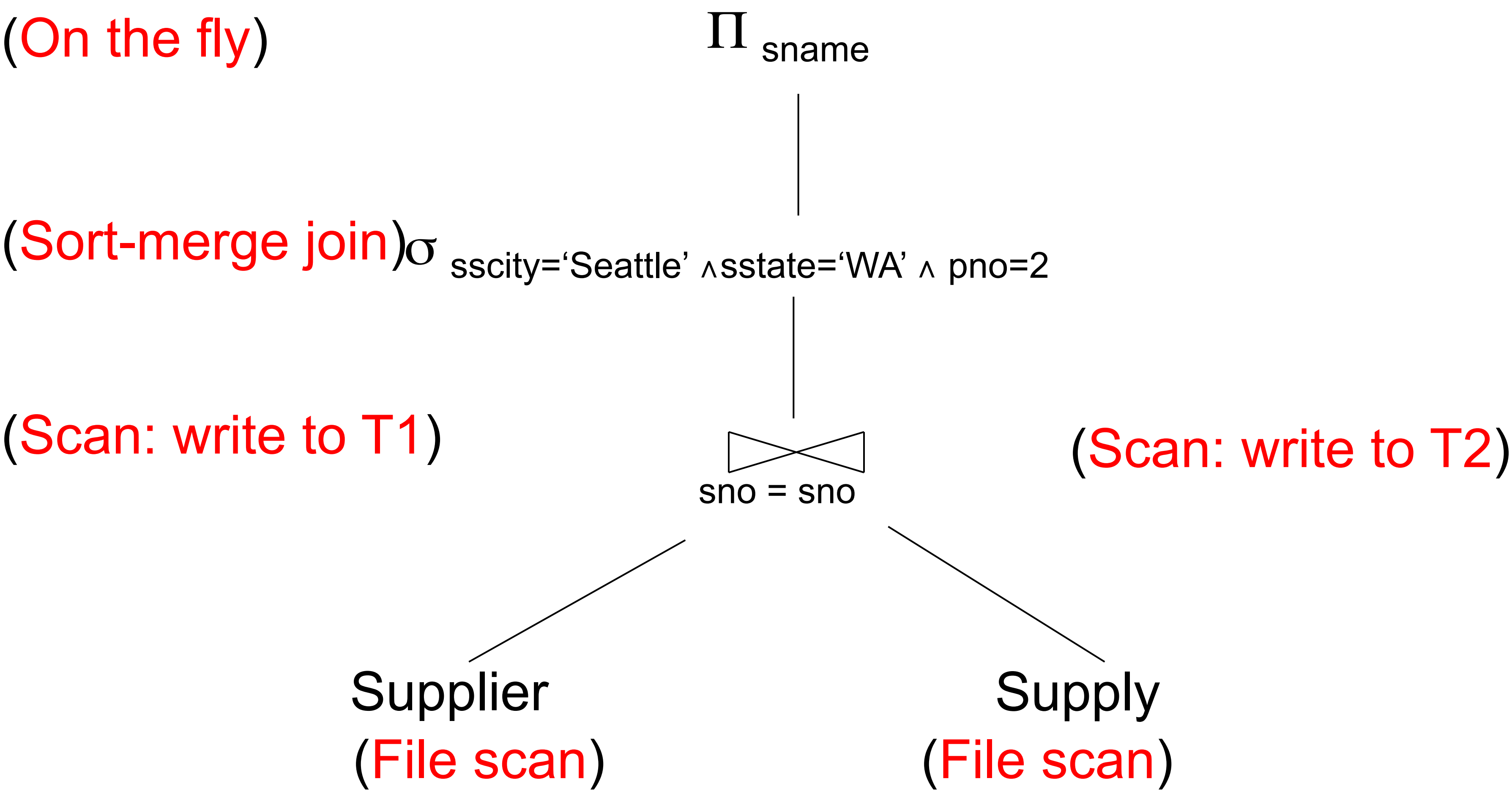
Supply (File scan)

Pipelined Execution

- Applies parent operator to tuples directly as they are produced by child operators
- Benefits
  - No operator synchronization issues
  - Saves cost of writing intermediate data to disk
  - Saves cost of reading intermediate data from disk
  - Good resource utilizations on single processor
- This approach is used whenever possible

Supplier(sno, sname, scity, sstate)
Supply(sno, pno, price)

Intermediate Tuple Materialization



Intermediate Tuple Materialization

- Writes the results of an operator to an intermediate table on disk

- No direct benefit but
- Necessary data is larger than main memory
- Necessary when operator needs to examine the same tuples multiple times

Outline of the Lecture

- Physical operators: join, group-by
- Query execution: pipeline, iterator model
- Query optimization
- Database statistics

Query Optimization

- Search space = set of all physical query plans that are equivalent to the SQL query
  - Defined by *algebraic laws* and restrictions on the *set of plans* used by the optimizer
- 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)

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

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)

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

Laws Involving Constraints

Foreign key

Product(<u>pid</u>, pname, price, cid)
Company(<u>cid</u>, 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: $R \ltimes S = \Pi_{A1,\ldots,An} (R \bowtie S)$

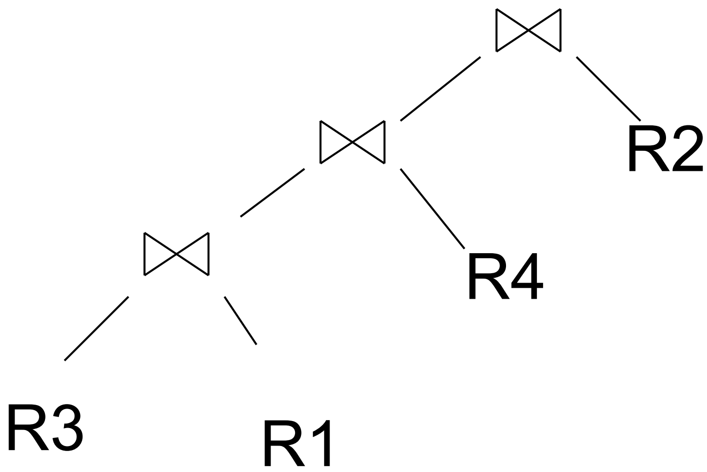
- The law of semijoins is:

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

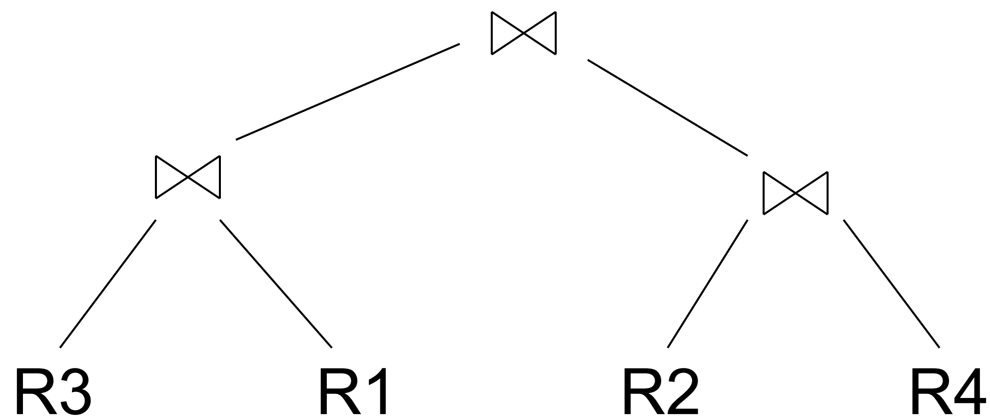
Laws with Semijoins

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

Left-Deep Plans and Bushy Plans



Left-deep plan



Bushy plan

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

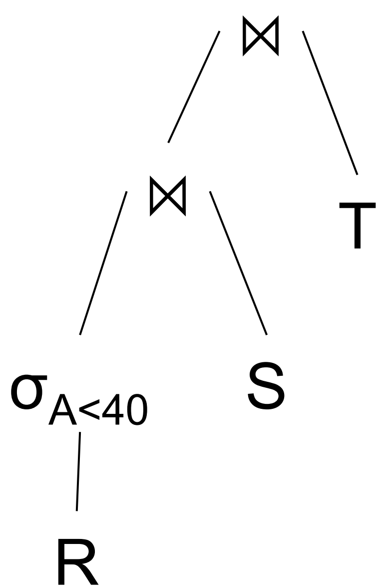
Search Algorithms

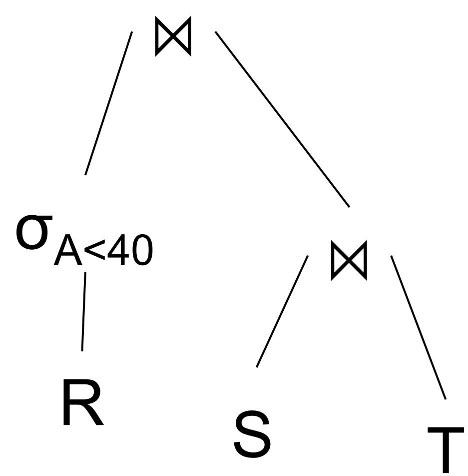
- Dynamic programming
  - Pioneered by System R for computing optimal join order, used today by all advanced optimizers
- Search space pruning
  - Enumerate partial plans, drop unpromising partial plans
  - Bottom-up v.s. top-down plans
- Access path selection
  - Refers to the plan for accessing a single 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
```





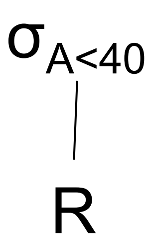
If the algorithm enumerates *complete* plans, then it is difficult to prune out unpromising sets of plans.

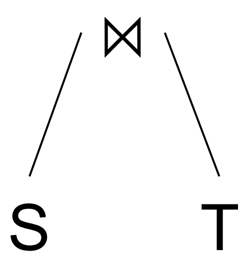
Bottom-up Partial Plans

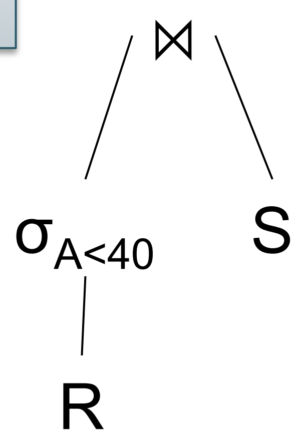
R(A,B)
S(B,C)
T(C,D)

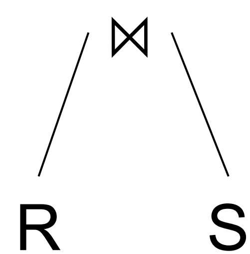
```
SELECT *
FROM R, S, T
WHERE R.B=S.B and S.C=T.C and R.A<40
```

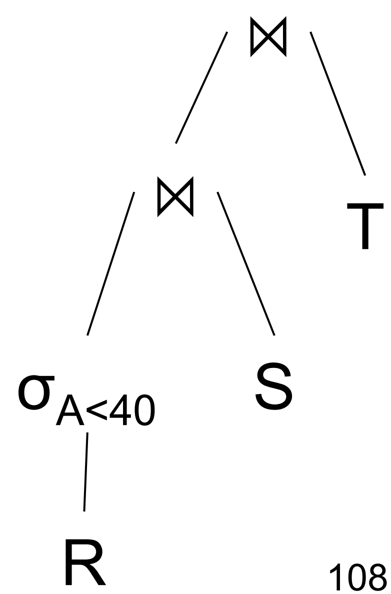
If the algorithm enumerates partial bottom-up plans, then pruning can be done more efficiently











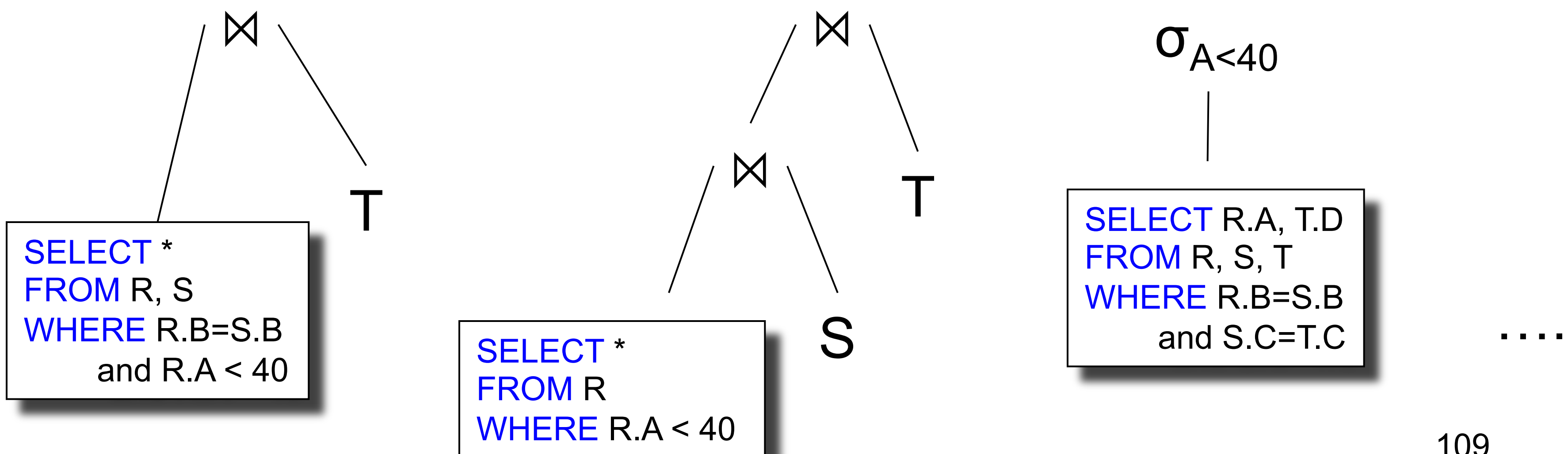
.....

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)

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

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

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

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

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)

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

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

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

Outline of the Lecture

- Physical operators: join, group-by
- Query execution: pipeline, iterator model
- Query optimization
- Database statistics

Database Statistics

- Collect statistical summaries of stored data
- Estimate size (=cardinality) in a bottom-up fashion
  - This is the most difficult part, and still inadequate in today's query optimizers
- Estimate cost by using the estimated size
  - Hand-written formulas, similar to those we used for computing the cost of each physical operator

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 (hard)

- 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

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)

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 $R \bowtie_{A=B} S$

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

Size Estimation for Join

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(<u>ssn</u>, 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) = ?

Estimate = 25000 / 50 = 500   Estimate = 25000 * 6 / 50 = 3000

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 rarely used
  - WHY ?

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