Lecture 8:
Query Execution

Wednesday, November 17, 2010

Outline

- Relational Algebra: Ch. 4.2
- Overview of query evaluation: Ch. 12
- Evaluating relational operators: Ch. 14

The WHAT and the HOW

- In SQL we write WHAT we want to get form the data
- The database system needs to figure out HOW to get the data we want
- The passage from WHAT to HOW goes through the Relational Algebra

Data Independence

SQL = WHAT

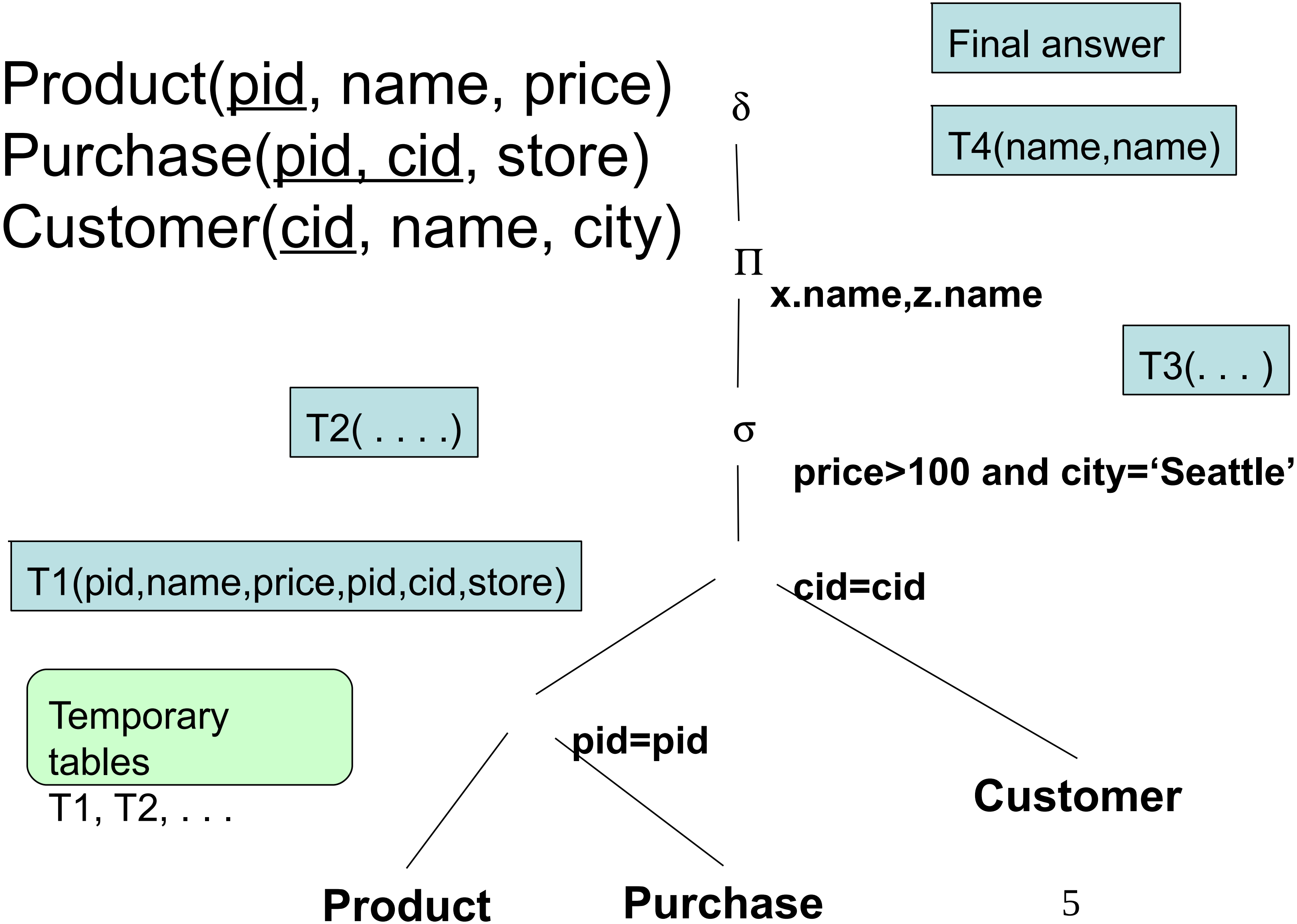
Product(<u>pid</u>, name, price)
Purchase(<u>pid, cid</u>, store)
Customer(<u>cid</u>, name, city)

```
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'
```

It's clear WHAT we want, unclear HOW to get it

Relational Algebra = HOW

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



Relational Algebra = HOW

The order is now clearly specified:

Iterate over PRODUCT…
…join with PURCHASE…
…join with CUSTOMER…
…select tuples with Price>100 and City='Seattle'…
…eliminate duplicates…
…and that's the final answer !

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
- Bag semantics

Extended Algebra Operators

- Union ᄉ , intersection ㅂ , difference -
- Selection σ
- Projection Π
- Join $\bowtie$
- Rename ρ
- Duplicate elimination δ
- Grouping and aggregation γ
- Sorting τ

Relational Algebra (1/3)

The Basic Five operators:

- Union: ˆ
- Difference: -
- Selection: σ
- Projection: Π
- Join: $\bowtie$

Relational Algebra (2/3)

Derived or auxiliary operators:

- Renaming: ρ
- Intersection, complement
- Variations of joins
  - natural, equi-join, theta join, semi-join, cartesian product

Relational Algebra (3/3)

Extensions for bags:

- Duplicate elimination: δ
- Group by: γ
- Sorting: τ

Union and Difference

R1 ˆ R2
R1 – R2

What do they mean over bags ?

What about Intersection ?

- Derived operator using minus

$$R1 \cap R2 = R1 - (R1 - R2)$$

- Derived using join (will explain later)

$$R1 \cap R2 = R1 \bowtie R2$$

Selection

- Returns all tuples which satisfy a condition

$$\sigma_c(R)$$

- Examples
  - $\sigma_{Salary > 40000}$ (Employee)
  - $\sigma_{name = \text{"Smith"}}$ (Employee)
- The condition c can be =, <, ≤, >, ≥, <>

Employee

SSN	Name	Salary
1234545	John	200000
5423341	Smith	600000
4352342	Fred	500000

$\sigma_{\text{Salary} > 40000}$ (Employee)

SSN	Name	Salary
5423341	Smith	600000
4352342	Fred	500000

Projection

- Eliminates columns

$$\Pi_{A1,\ldots,An}(R)$$

- Example: project social-security number and names:
  - $\Pi_{SSN,\ Name}(Employee)$
  - Answer(SSN, Name)

Semantics differs over set or over bags

Employee

SSN	Name	Salary
1234545	John	20000
5423341	John	60000
4352342	John	20000

Π Name,Salary (Employee)

Name	Salary
John	20000
John	60000
John	20000

Bag semantics

Name	Salary
John	20000
John	60000

Set semantics

Which is more efficient to implement ?

Cartesian Product

- Each tuple in R1 with each tuple in R2

$$\boxed{R1 \times R2}$$

- Very rare in practice; mainly used to express joins

Employee

Name	SSN
John	999999999
Tony	777777777

Dependent

EmpSSN	DepName
999999999	Emily
777777777	Joe

Employee ✕ Dependent

Name	SSN	EmpSSN	DepName
John	999999999	999999999	Emily
John	999999999	777777777	Joe
Tony	777777777	999999999	Emily
Tony	777777777	777777777	Joe

Renaming

- Changes the schema, not the instance

$$\boxed{\rho\ B1,\ldots,Bn\ (R)}$$

- Example:
  - $\rho N, S(\text{Employee}) \rightarrow \text{Answer}(N, S)$

Natural Join

$$\boxed{R1 \bowtie R2}$$

- Meaning: $R1 \bowtie R2 = \Pi A(\sigma(R1 \times R2))$
- Where:
  - The selection σ checks equality of all common attributes
  - The projection eliminates the duplicate common attributes

Natural Join

R

A	**B**
X	Y
X	Z
Y	Z
Z	V

S

B	**C**
Z	U
V	W
Z	V

R $\bowtie$ **S** =
$\Pi_{ABC}(\sigma_{R.B=S.B}(R \times S))$

A	**B**	**C**
X	Z	U
X	Z	V
Y	Z	U
Y	Z	V
Z	V	W

Natural Join

- Given the schemas R(A, B, C, D), S(A, C, E), what is the schema of R $\bowtie$ S ?

- Given R(A, B, C), S(D, E), what is R $\bowtie$ S ?

- Given R(A, B), S(A, B), what is R $\bowtie$ S ?

Theta Join

- A join that involves a predicate

$$\boxed{R1 \bowtie_{\theta} R2 \quad = \quad \sigma_{\theta}\,(R1 \times R2)}$$

- Here θ can be any condition

Eq-join

- A theta join where θ is an equality

$$\boxed{R1 \bowtie_{A=B} R2 \quad = \quad \sigma_{A=B} (R1 \times R2)}$$

- This is by far the most used variant of join in practice

So Which Join Is It ?

- When we write R $\bowtie$ S we usually mean an eq-join, but we often omit the equality predicate when it is clear from the context

Semijoin

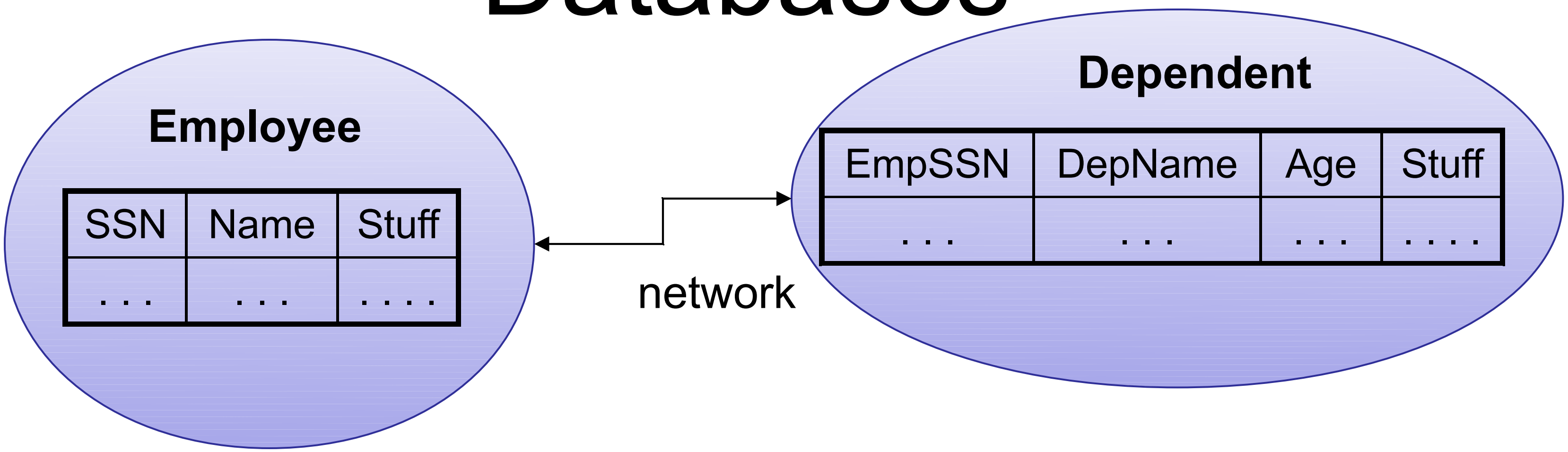
$$R \ltimes_C S = \Pi_{A1,\ldots,An} (R \bowtie_C S)$$

- Where A1, …, An 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

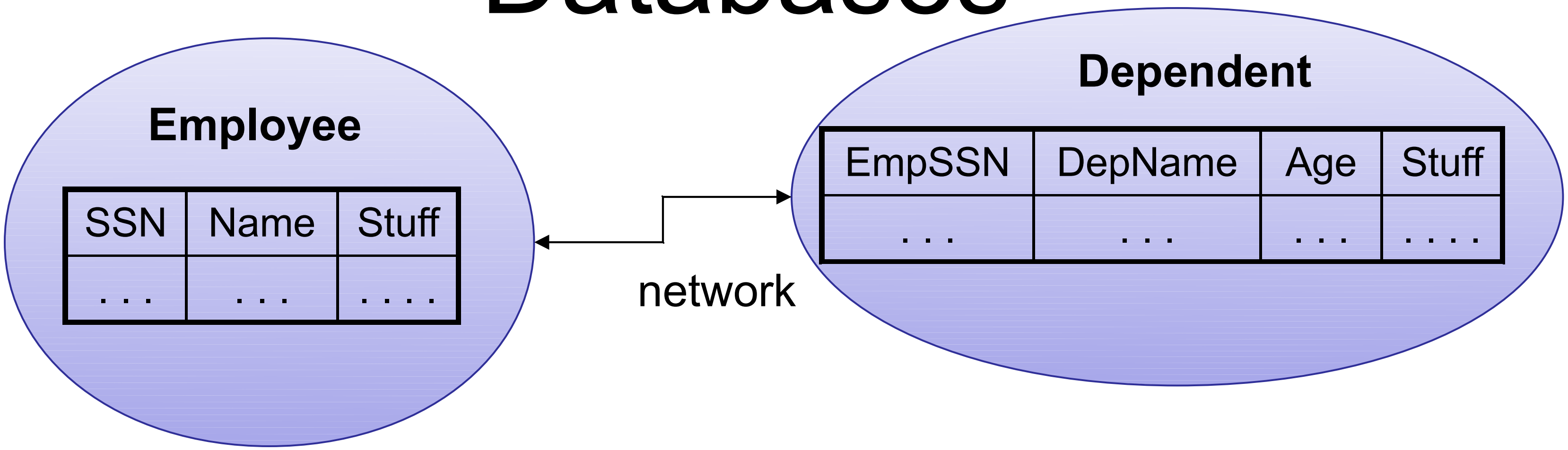


$\text{Employee} \bowtie_{\text{SSN}=\text{EmpSSN}} (\sigma_{\text{age}>71} (\text{Depender}$

Assumptions: Very few Employees have dependents.
Very few dependents have age > 71.
"Stuff" is big.

Task: compute the query with minimum amount of data transfer

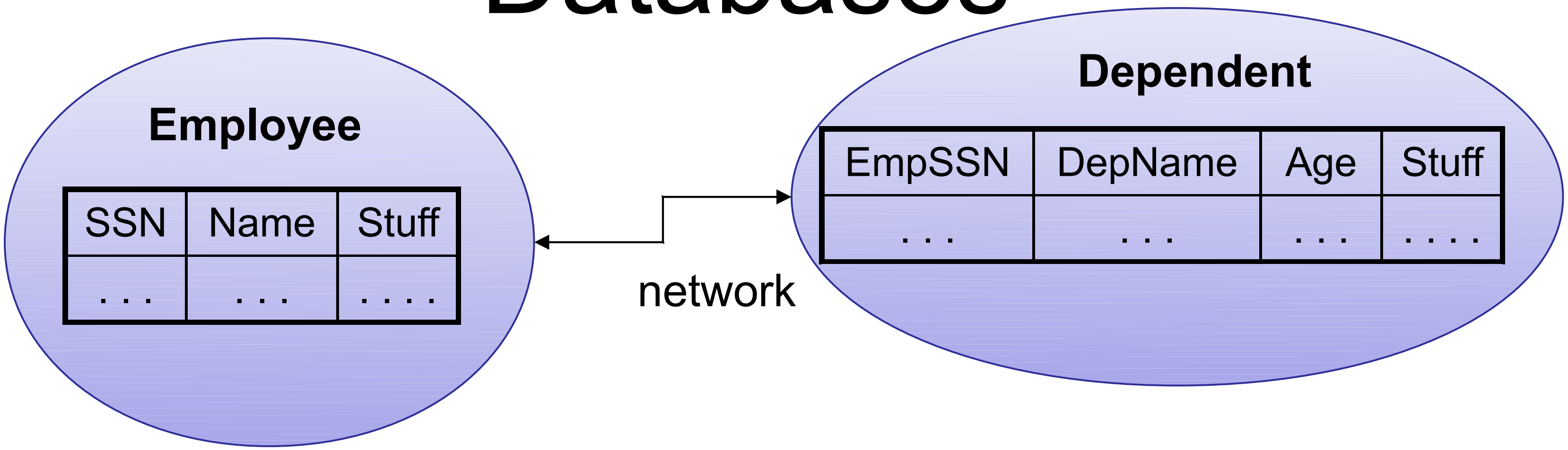
Semijoins in Distributed Databases



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

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

Semijoins in Distributed Databases

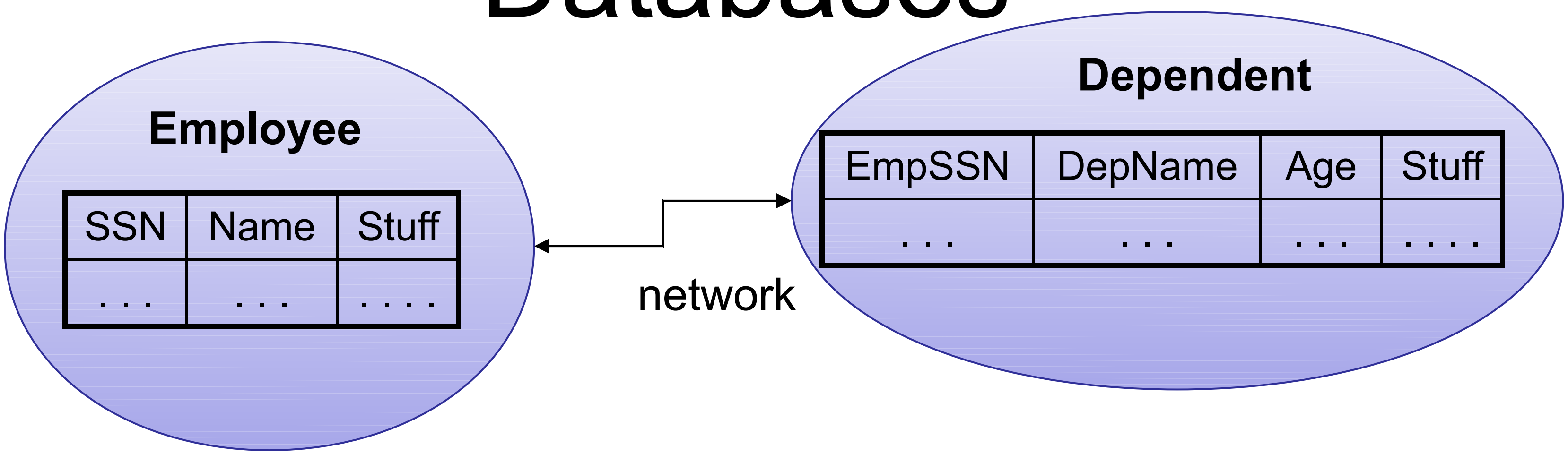


$$\text{Employee} \bowtie_{\text{SSN=EmpSSN}} (\sigma_{\text{age>71}} (\text{Dependen}$$

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

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

Semijoins in Distributed Databases



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

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

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

$\text{Answer} = R \bowtie_{SSN=EmpSSN} \text{Dependents}$

Joins R US

- The join operation in all its variants (eq-join, natural join, semi-join, outer-join) is at the *heart* of relational database systems

- WHY ?

Operators on Bags

- Duplicate elimination δ

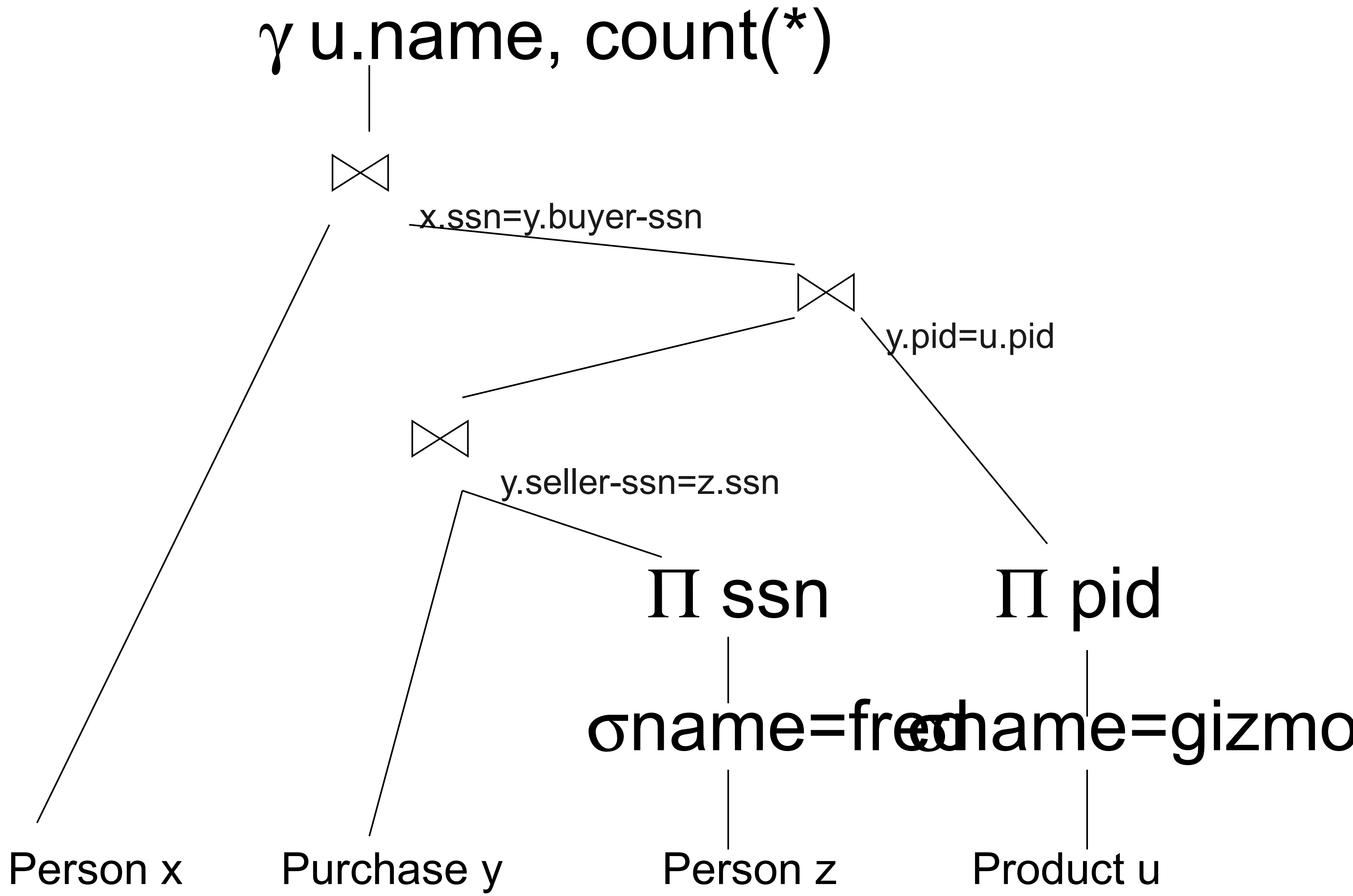
$\delta(R)$ = select distinct * from R

- Grouping γ

$\gamma A,sum(B)$ (R) = select A,sum(B) from R group by A

- Sorting τ

Complex RA Expressions



RA = Dataflow Program

- Several operations, plus strictly specified order
- In RDBMS the dataflow graph is always a tree
- Novel applications (s.a. PIG), dataflow graph may be a DAG

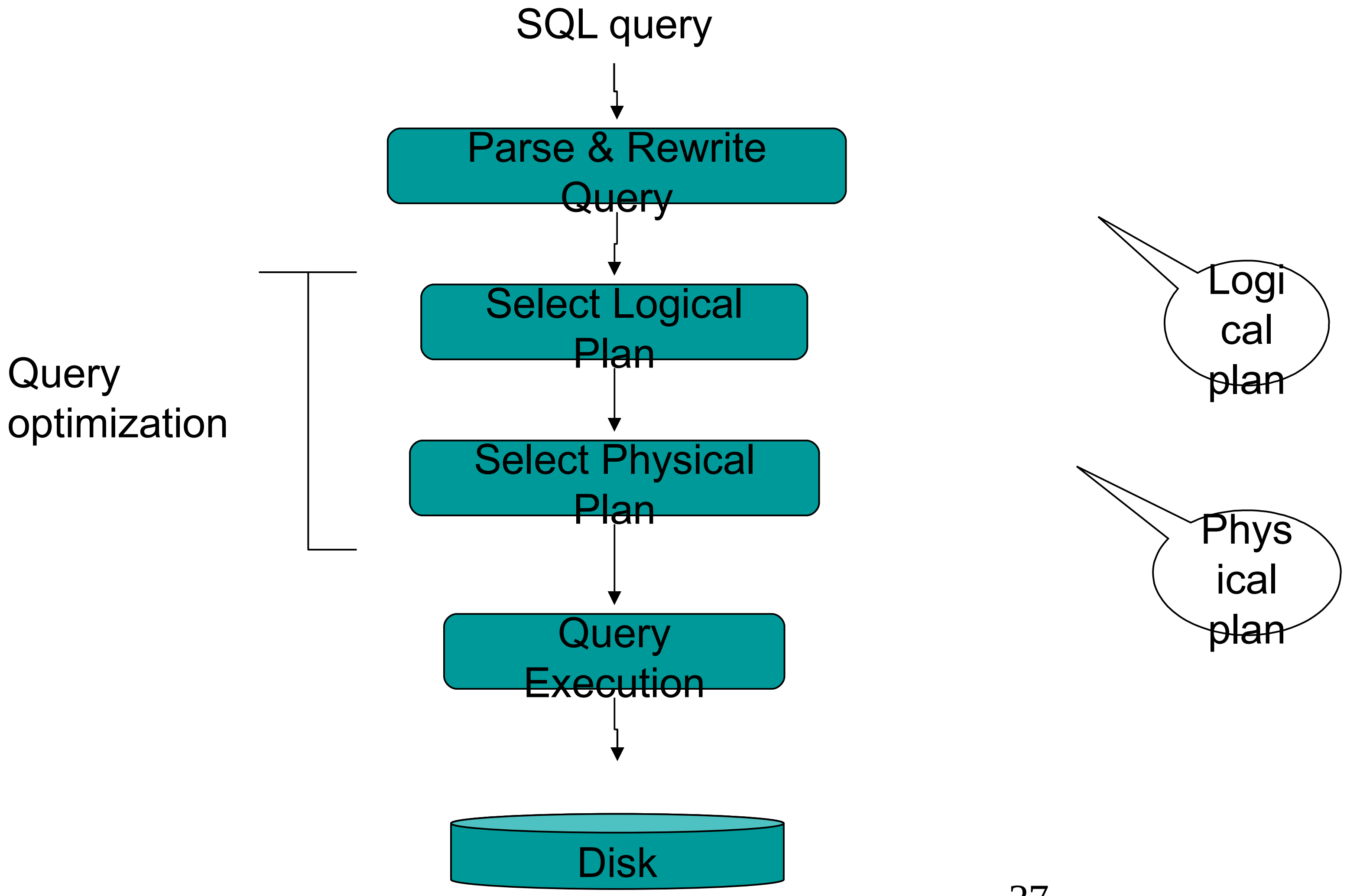
Limitations of RA

- Cannot compute "transitive closure"

Name1	Name2	Relationship
Fred	Mary	Father
Mary	Joe	Cousin
Mary	Bill	Spouse
Nancy	Lou	Sister

- Find all direct and indirect relatives of Fred
- Cannot express in RA !!! Need to write Java program
- Remember *the Bacon number* ? Needs TC too !

Steps of the Query Processor



Example Database Schema

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

View: Suppliers in Seattle

```
CREATE VIEW NearbySupp AS
SELECT sno, sname
FROM Supplier
WHERE scity='Seattle' AND sstate='WA'
```

Example Query

Find the names of all suppliers in Seattle who supply part number 2

```
SELECT sname FROM NearbySupp
WHERE sno IN ( SELECT sno
              FROM Supplies
              WHERE pno = 2 )
```

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.

Rewritten Version of Our Query

Original query:

```
SELECT sname
FROM NearbySupp
WHERE sno IN ( SELECT sno
               FROM Supplies
               WHERE pno = 2 )
```

Rewritten query:

```
SELECT S.sname
FROM Supplier S, Supplies U
WHERE S.scity='Seattle' AND S.sstate='WA’
AND S.sno = U.sno
AND U.pno = 2;
```

Continue with Query Evaluation

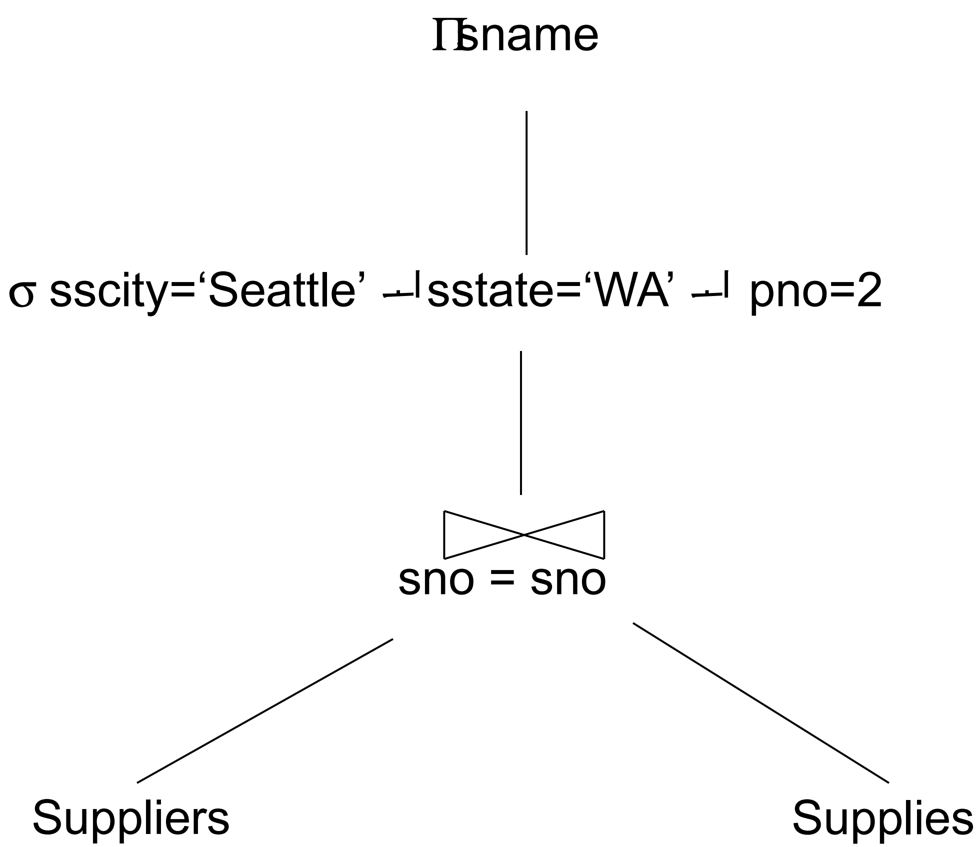
- **Step 3: Query optimization**
  - Find an efficient query plan for executing the query

- 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

Extended Algebra Operators

- Union $\cup$, intersection $\cap$, difference -
- Selection σ
- Projection Π
- Join $\bowtie$
- Duplicate elimination δ
- Grouping and aggregation γ
- Sorting τ
- Rename ρ

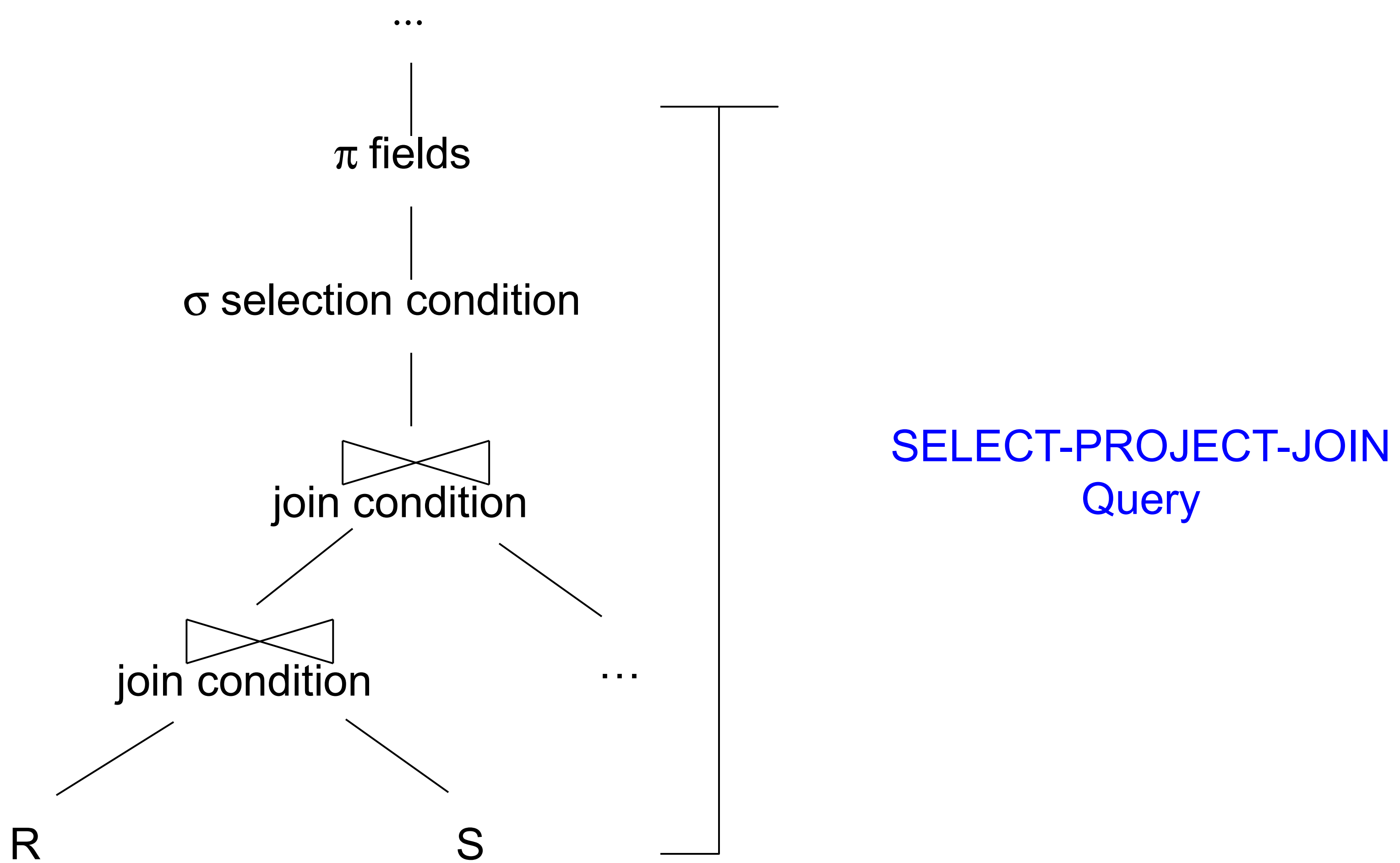
Logical Query Plan



Query Block

- Most optimizers operate on individual query blocks
- A query block is an SQL query with **no nesting**
  - **Exactly one**
    - SELECT clause
    - FROM clause
  - **At most one**
    - WHERE clause
    - GROUP BY clause
    - HAVING clause

Typical Plan for Block (1/2)



Typical Plan For Block (2/2)

σhaving-ondition

γ fields, sum/count/min/max(fields)

σ selection condition

$\bowtie$ join condition

… …

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

How about Subqueries?

```
SELECT  Q.sno
FROM Supplier Q
WHERE  Q.sstate = 'WA'
   and not exists
      SELECT *
      FROM Supply P
      WHERE P.sno = Q.sno
            and P.price > 100
```

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

How about Subqueries?

```
SELECT  Q.sno
FROM Supplier Q
WHERE  Q.sstate = 'WA'
    and not exists
       SELECT *
       FROM Supply P
       WHERE P.sno = Q.sno
             and P.price > 100
```

Correlation !

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

How about Subqueries?

```
SELECT  Q.sno
FROM Supplier Q
WHERE  Q.sstate = 'WA'
    and not exists
       SELECT *
       FROM Supply P
       WHERE P.sno = Q.sno
             and P.price > 100
```

De-Correlation

```
SELECT  Q.sno
FROM Supplier Q
WHERE  Q.sstate = 'WA'
    and Q.sno not in
       SELECT P.sno
       FROM Supply P
       WHERE P.price > 100
```

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

How about Subqueries?

Un-nesting

```
(SELECT  Q.sno
 FROM Supplier Q
 WHERE  Q.sstate = 'WA')
   EXCEPT
 (SELECT P.sno
   FROM Supply P
   WHERE P.price > 100)
```

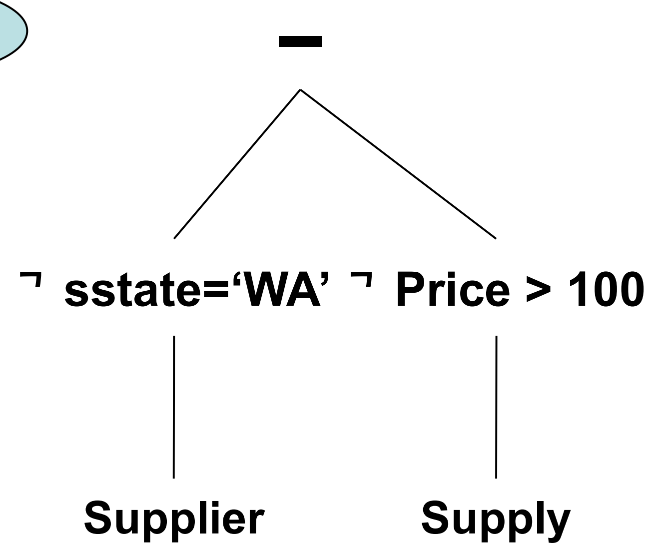
```
SELECT  Q.sno
FROM Supplier Q
WHERE  Q.sstate = 'WA'
    and Q.sno not in
      SELECT P.sno
      FROM Supply P
      WHERE P.price > 100
```

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

How about Subqueries?

```
(SELECT  Q.sno
 FROM Supplier Q
 WHERE  Q.sstate = 'WA')
   EXCEPT
 (SELECT P.sno
  FROM Supply P
  WHERE P.price > 100)
```

Finall
y…

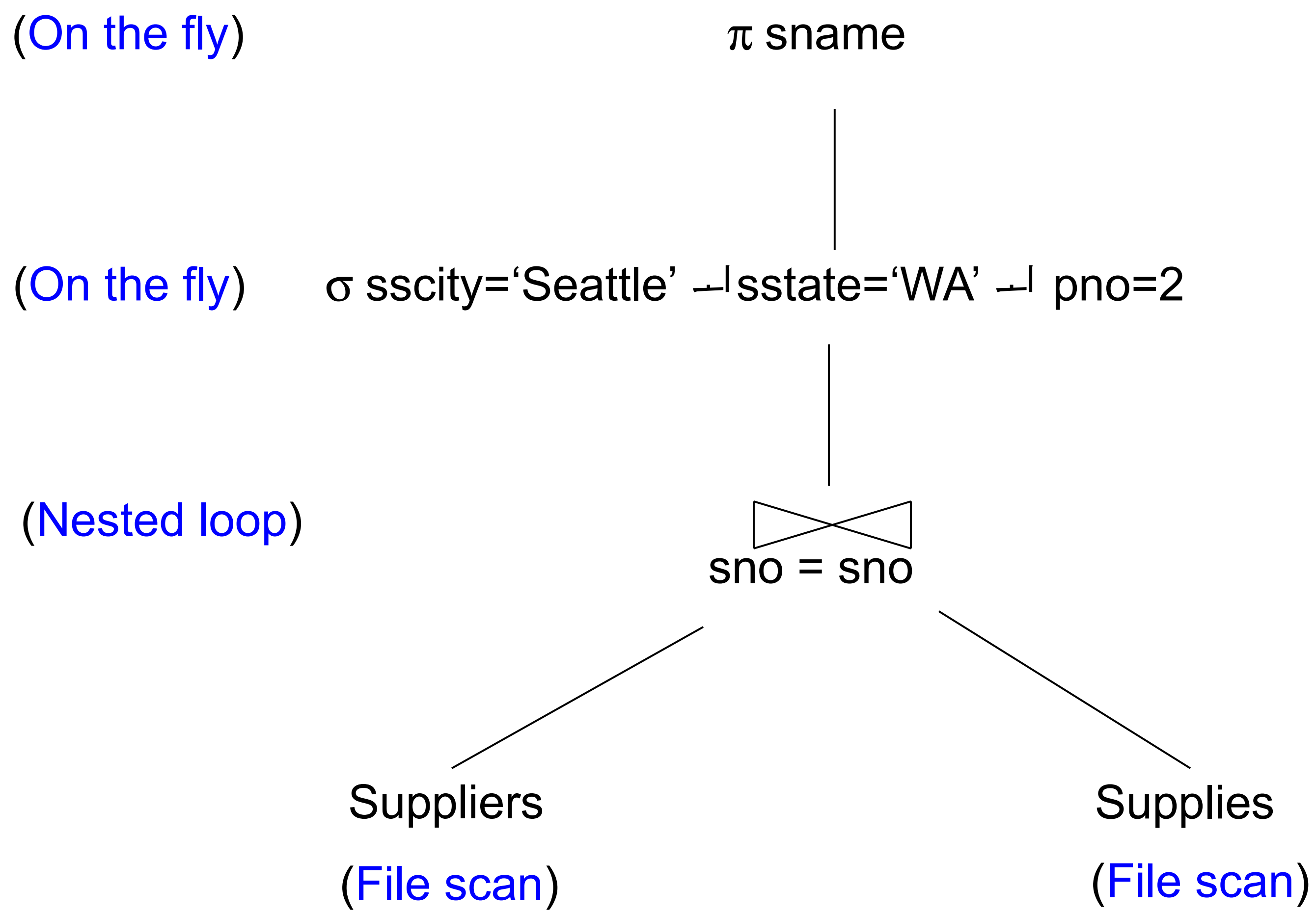


Physical Query Plan

- Logical query plan with extra 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)
Part(pno,pname,psize,pcolor)
Supply(sno,pno,price)

Physical Query Plan



Final Step in Query Processing

- **Step 4: Query execution**
  - How to synchronize operators?
  - How to pass data between operators?

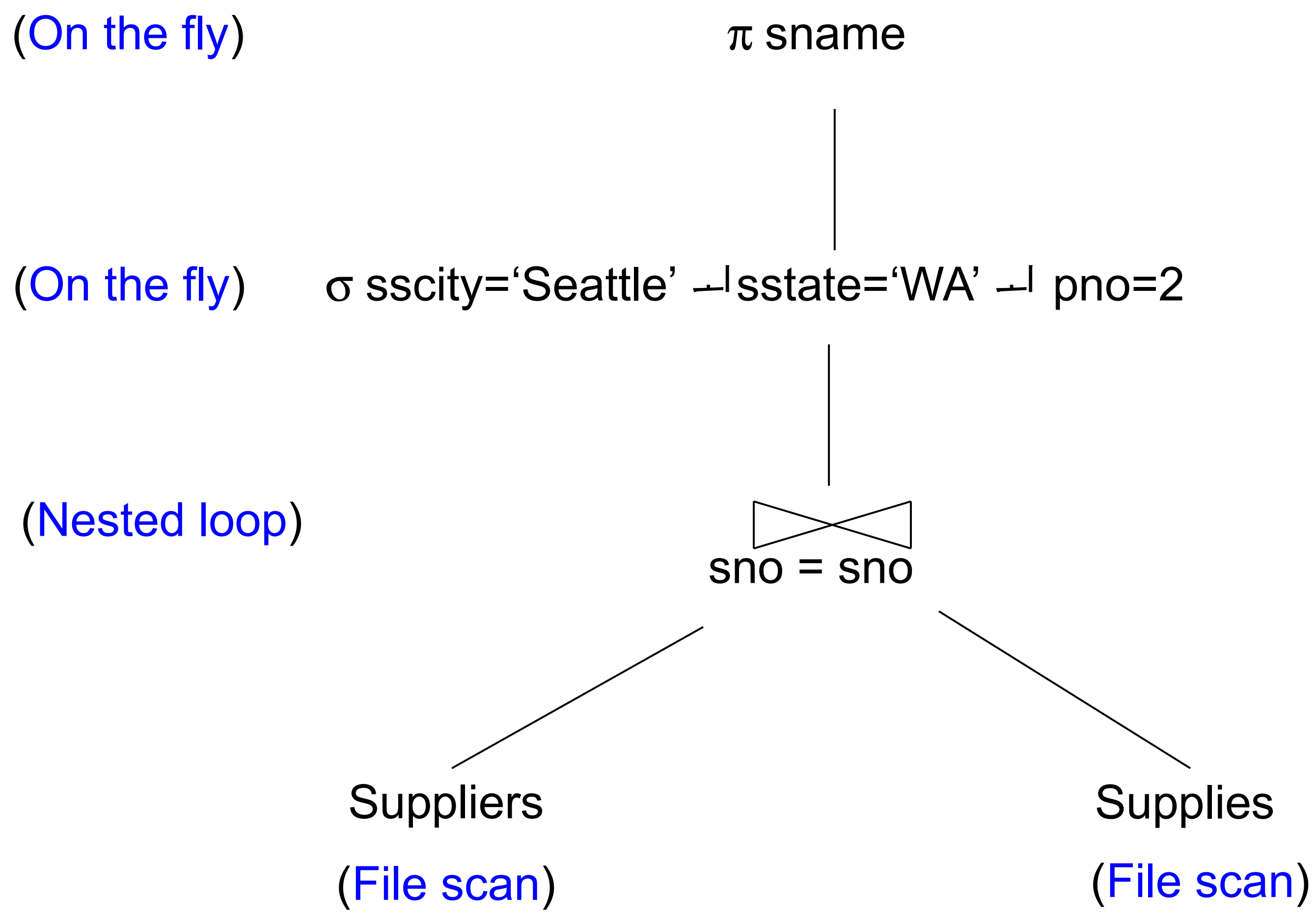
- What techniques are possible?
  - One thread per query
  - Iterator interface
  - Pipelined execution
  - Intermediate result materialization

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

Pipelined Execution

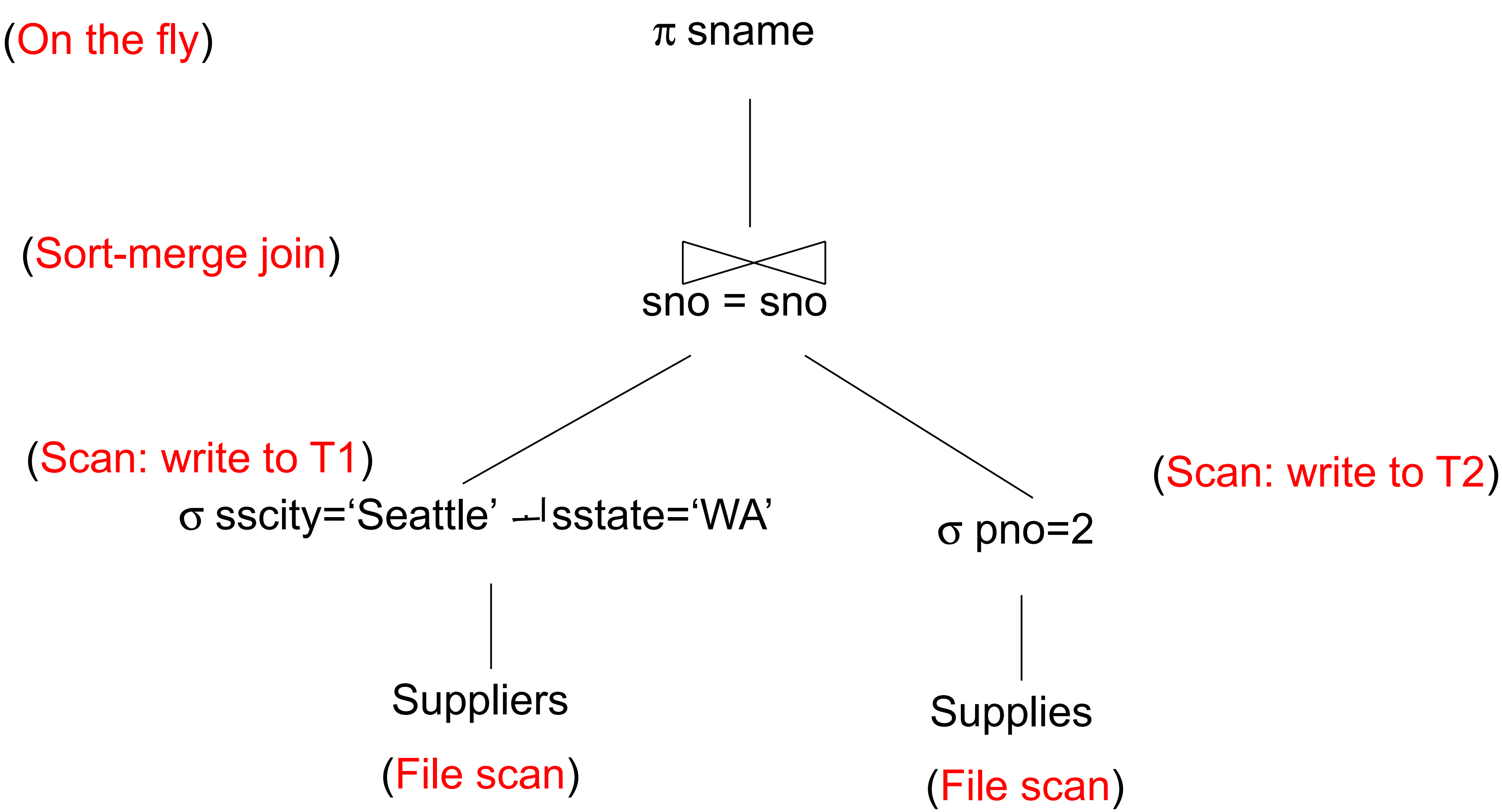


Pipelined Execution

- 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)
Part(pno,pname,psize,pcolor)
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

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

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

Question in Class

Logical operator:

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

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

1.

2.

3.

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

Question in Class

Logical operator:

Supply(sno,pno,price) 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

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

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=inner, Supply=outer ?
What if we had an index on Part.pno ?

It's more complicated…

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

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

Main Memory Nested Loop Join Revisited

```
pen ( ) {
  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 !

BRIEF Review of Hash Tables

A (naïve) hash function:

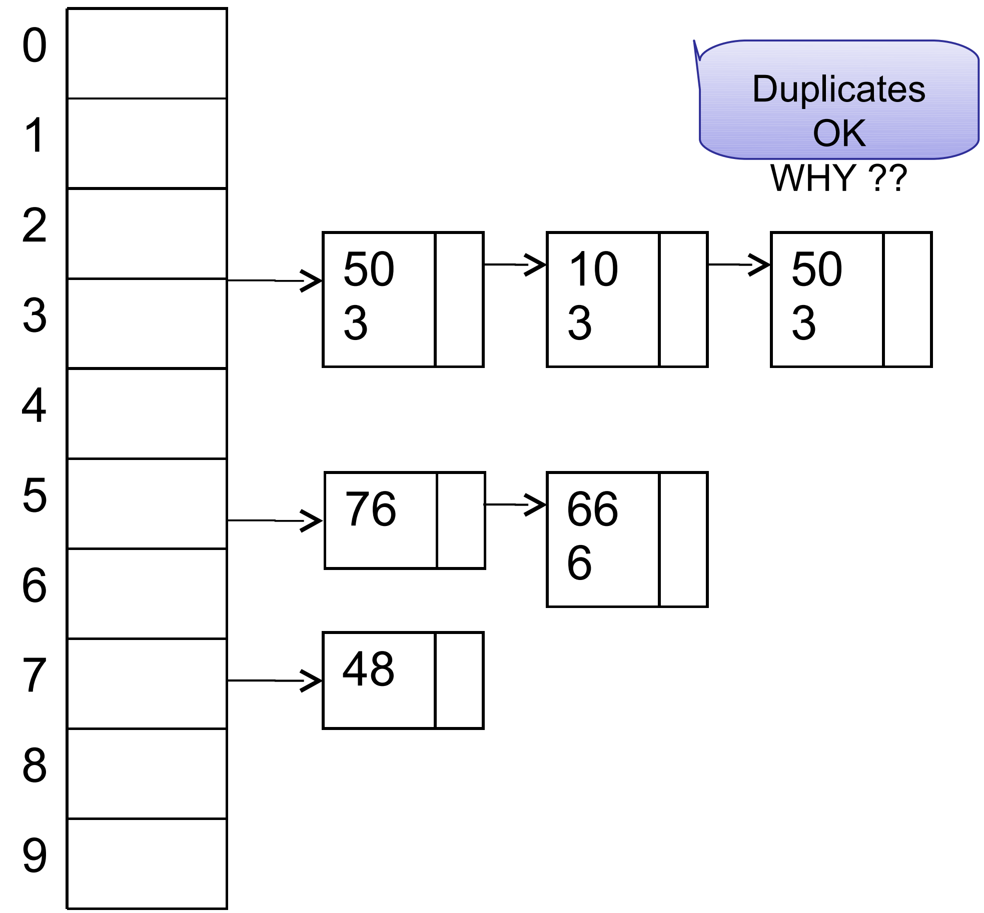
$$h(x) = x \bmod 10$$

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 ***<u>list</u>*** of all values v associated to the key k

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

2. Hash Join (main memory)

Buil
d
pha
se

```
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); }
}
```

Prob
ing

Supply=
outer
Part=inn
er

Recall: need to rewrite as open, get_next, close

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

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 ???

Main Memory Group By

Grouping:

Product(name, department, quantity)

$\gamma_{\text{department, sum(quantity)}}$ (Product)

Answer(department, sum)

Main memory hash table

Question: How ?

Duplicate Elimination IS Group By

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

- Hash table in main memory

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

Selections, Projections

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

Review (1/2)

- Each **operator implements this interface**
- **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

Review (2/2)

- Three algorithms for main memory join:
  - Nested loop join
  - Hash join
  - Merge join

If $|R| = m$ and $|S| = n$, what is the asymptotic complexity for computing $R \bowtie S$?

- Algorithms for selection, projection, group-by

External Memory Algorithms

- Data is too large to fit in main memory
- Issue: disk access is 3-4 orders of magnitude slower than memory access
- Assumption: runtime dominated by # of disk I/O's; will ignore the main memory part of the runtime

Cost Parameters

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

Cost parameters:

- B(R) = number of blocks for relation 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)

Ad-hoc Convention

- We assume that the operator *reads* the data from disk
- We assume that 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)

Sequential Scan of a Table R

- When R is *clustered*
  - Blocks consists only of records from this table
  - B(R) << T(R)
  - Cost = B(R)

- When R is *unclustered*
  - Its records are placed on blocks with other tables
  - B(R) ≈ T(R)
  - Cost = T(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) when S is clustered
- Cost: T(R) T(S) when S is unclustered

Examples

M = 4;   R, S are clustered

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

Can you do better ?

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

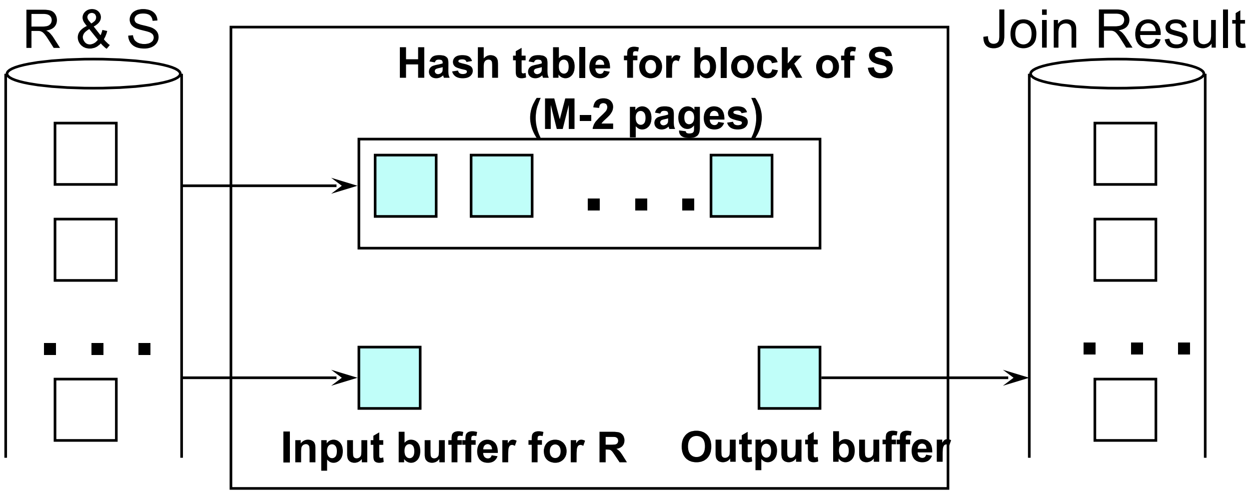
Block-Based Nested-loop Join

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: book calls S the *inner* relation

Block Nested-loop Join



Examples

M = 4;   R, S are clustered

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

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

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

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

Index Based Selection

- Example:

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

cost of $\sigma a=v(R)$ = ?

- Table scan (assuming R is clustered):
  - 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 (Assuming R is clustered):

- 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 ⋈ S when each is larger than main memory
- Two methods:
  - Partitioned hash join (many variants)
  - Merge-join
- Similar for grouping

Partitioned Hash-based Algorithms

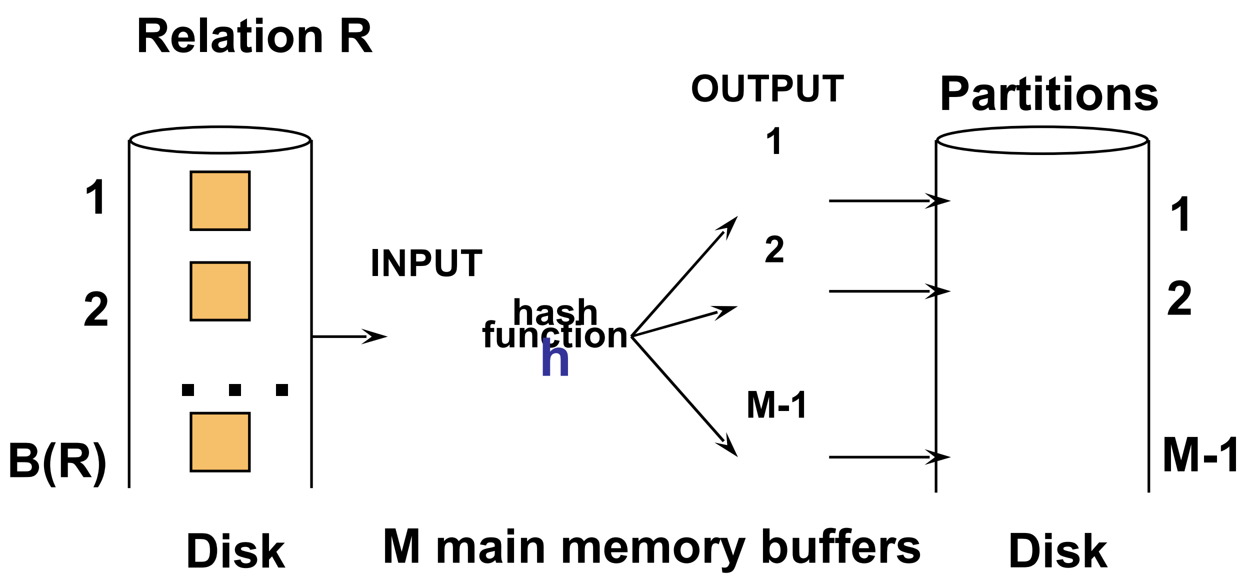
Idea:

- 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) <= M2

Grouping

- $\gamma(R) =$ 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) <= M2

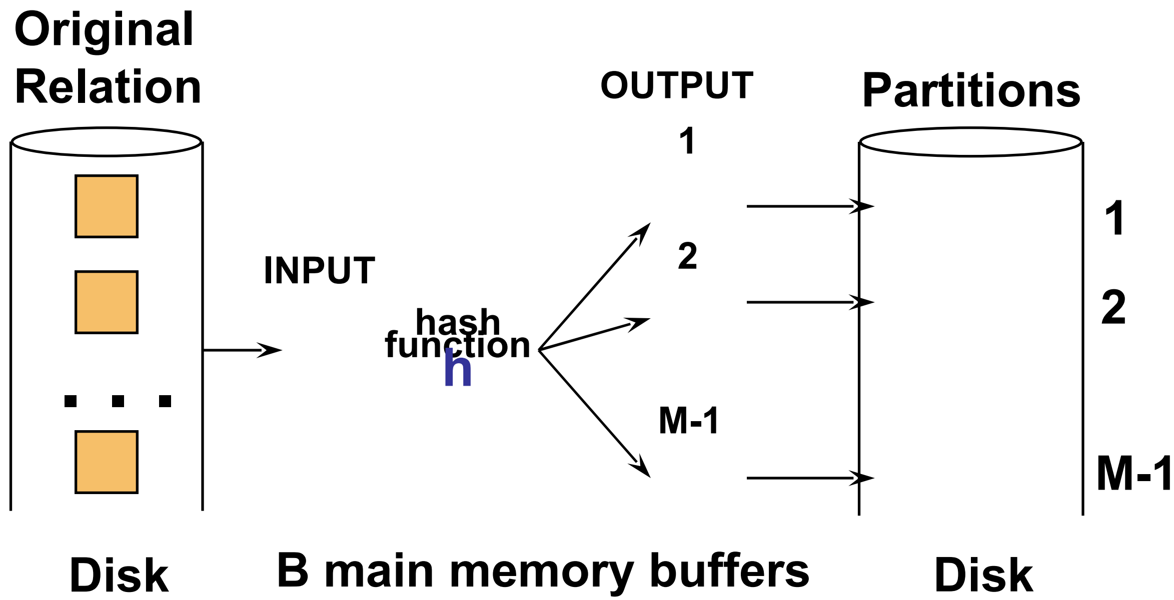
Partitioned Hash Join

$R \bowtie S$

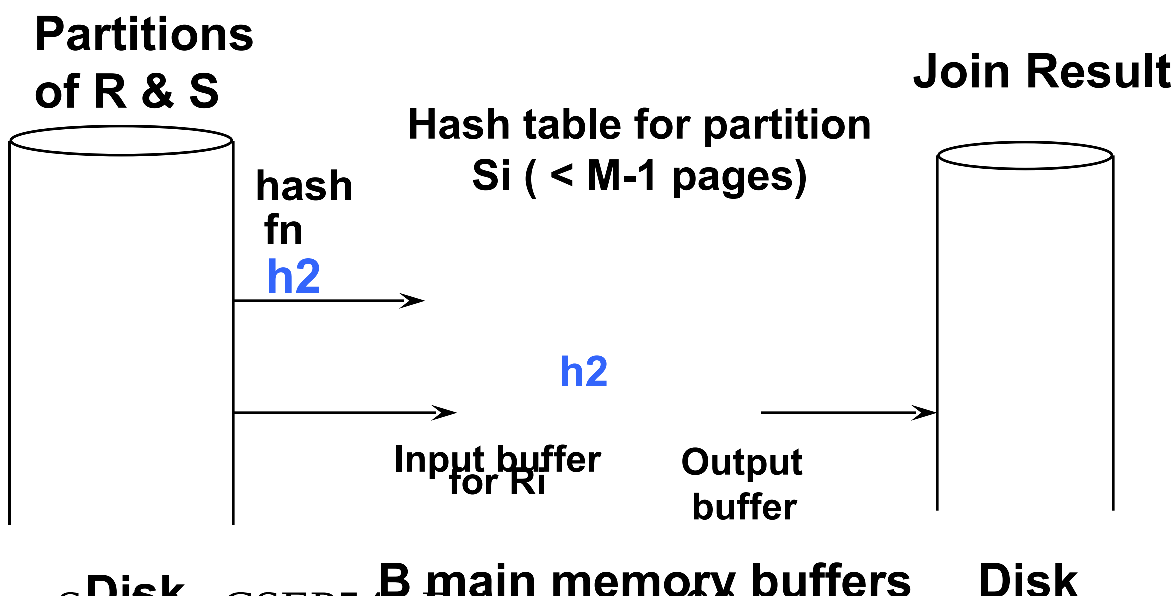
- Step 1:
  - Hash S into M buckets
  - send all buckets to disk
- Step 2
  - Hash R into M buckets
  - Send all buckets to disk
- Step 3
  - Join every pair of buckets

Hash-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.



Partitioned Hash Join

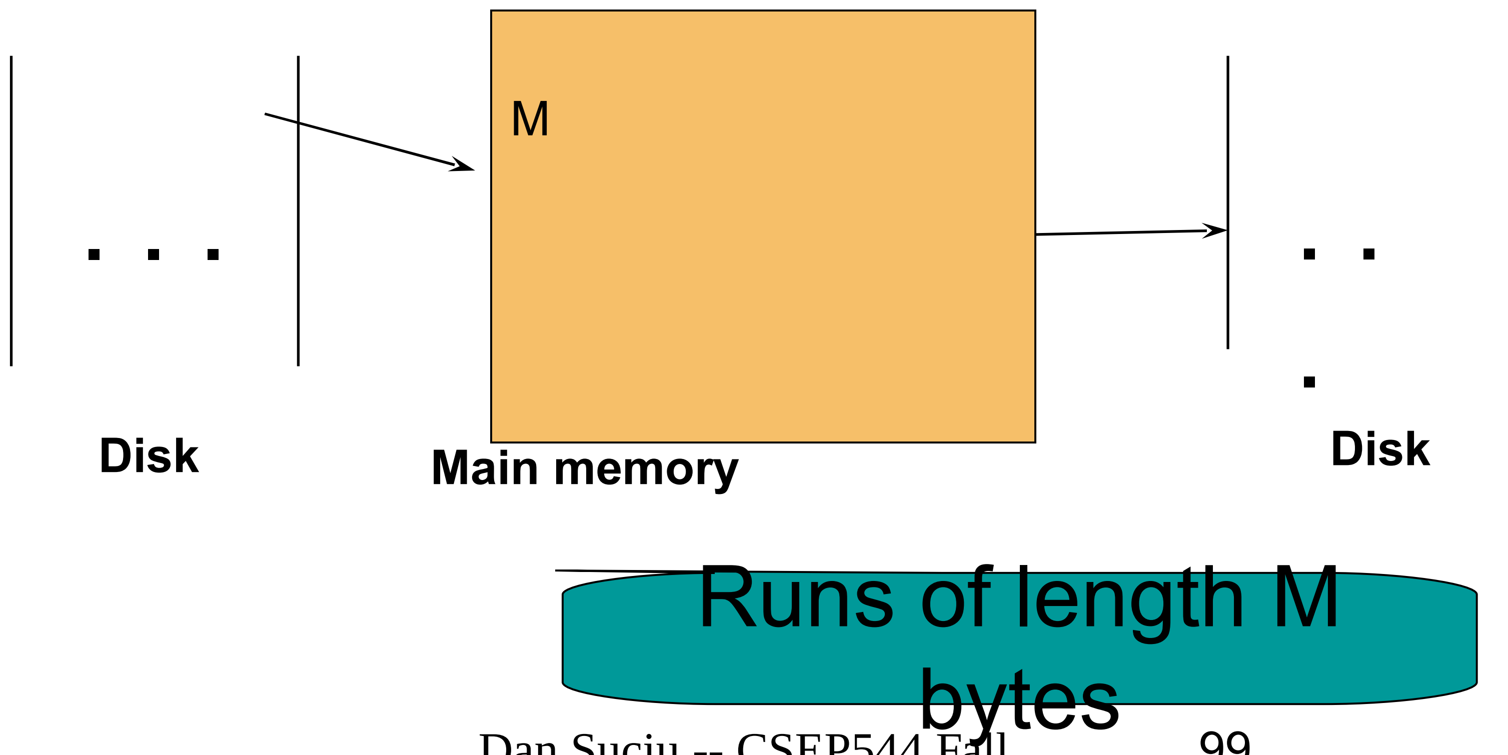
- Cost: 3B(R) + 3B(S)
- Assumption: min(B(R), B(S)) <= M2

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 < M2

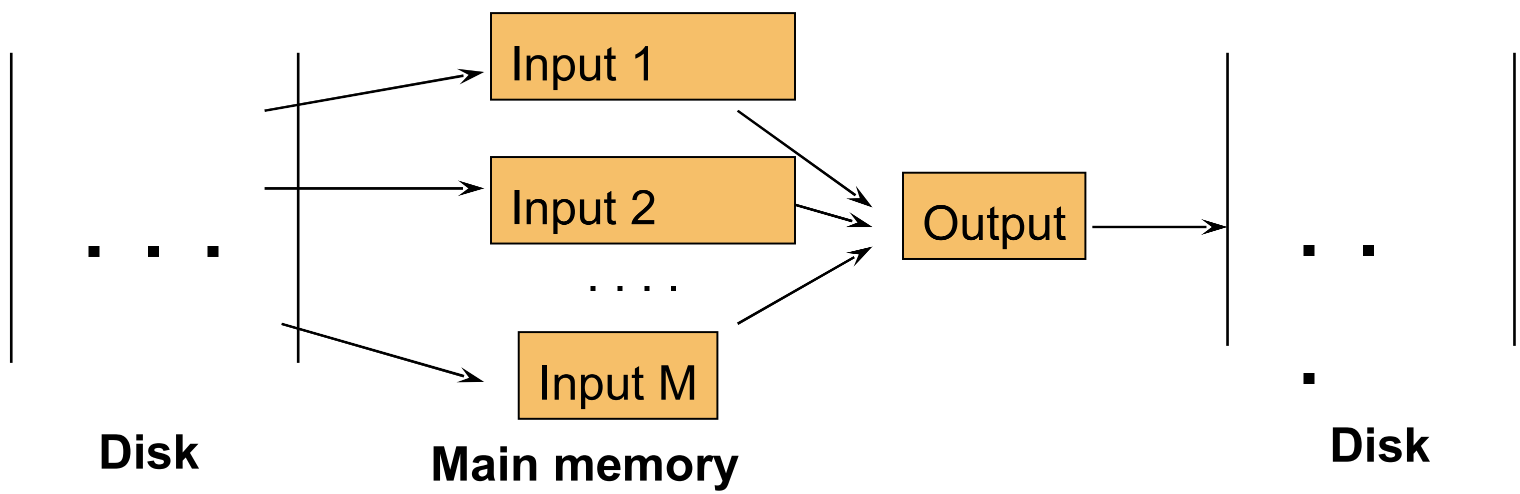
External Merge-Sort: Step 1

- Phase one: load M bytes in memory, sort



External Merge-Sort: Step 2

- Merge M – 1 runs into a new run
- Result: runs of length M (M – 1) ≈ M2



If B <= M2 then we are done

Cost of External Merge Sort

- Read+write+read = 3B(R)
- Assumption: B(R) <= M2

Grouping

Grouping: $\gamma_{a,\ sum(b)}$ (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 ?

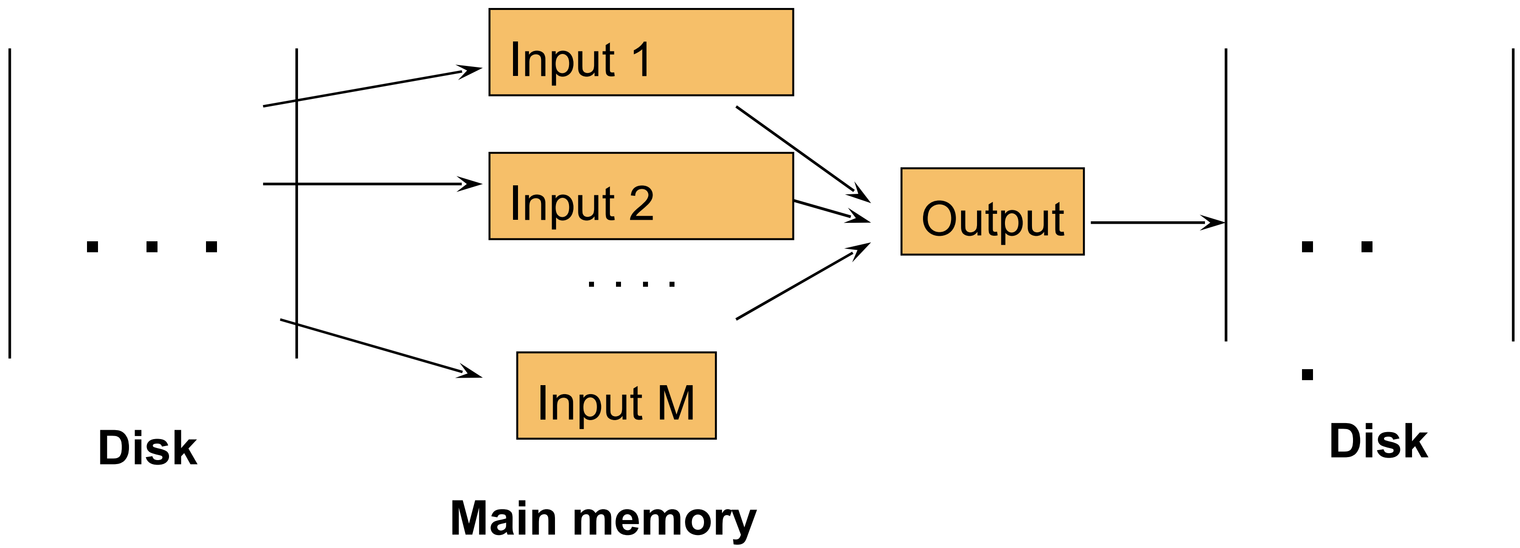
Cost = 3B(R)
Assumption: $B(\delta(R)) \le M^2$

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



M1 = B(R)/M runs for R

M2 = B(S)/M runs for S

Merge-join M1 + M2 runs;

need M1 + M2 <= M

Two-Pass Algorithms Based on Sorting

Join R $\bowtie$ S

- If the number of tuples in R matching those in S is small (or vice versa) we can compute the join during the merge phase
- Total cost: 3B(R)+3B(S)
- Assumption: B(R) + B(S) <= M2

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)) <= M2
- Merge Join: 3B(R)+3B(S)
  - B(R)+B(S) <= M2