Lecture 7:
Query Execution and Optimization

Tuesday, February 20, 2007

Outline

- Chapters 4, 12-15

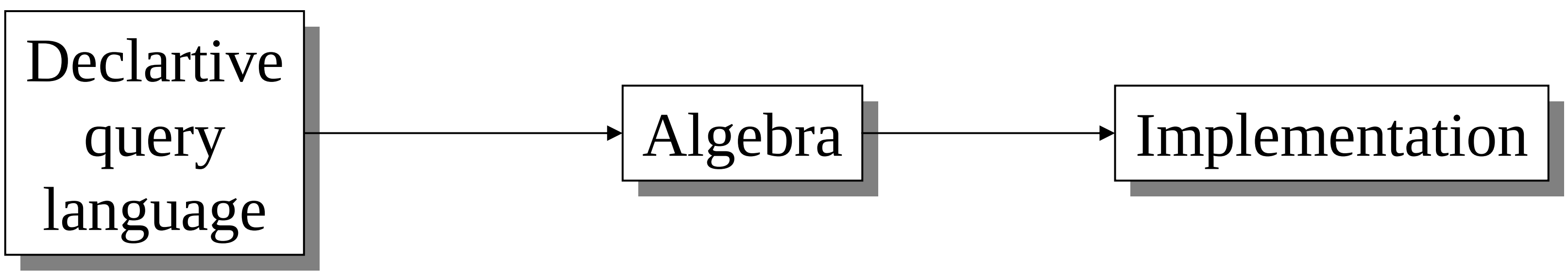
DBMS Architecture

How does a SQL engine work ?

- SQL query $\rightarrow$ relational algebra plan
- Relational algebra plan $\rightarrow$ Optimized plan
- Execute each operator of the plan

Relational Algebra

- Formalism for creating new relations from existing ones
- Its place in the big picture:



SQL,
relational calculus

Relational algebra
Relational bag algebra

Relational Algebra

- Five operators:
  - Union: $\cup$
  - Difference: -
  - Selection: σ
  - Projection: Π
  - Cartesian Product: $\times$
- Derived or auxiliary operators:
  - Intersection, complement
  - Joins (natural,equi-join, theta join, semi-join)
  - Renaming: ρ

1. Union and 2. Difference

- $R1 \cup R2$
- Example:
  - ActiveEmployees $\cup$ RetiredEmployees

- R1 – R2
- Example:
  - AllEmployees -- RetiredEmployees

What about Intersection ?

- It is a derived operator
- $R1 \cap R2 = R1 - (R1 - R2)$
- Also expressed as a join (will see later)
- Example
  - $\text{UnionizedEmployees} \cap \text{RetiredEmployees}$

3. Selection

- Returns all tuples which satisfy a condition
- Notation: $\sigma_c(R)$
- Examples
  - $\sigma_{\text{Salary > 40000}}$ (Employee)
  - $\sigma_{\text{name = “Smith”}}$ (Employee)
- The condition c can be $=, <, \leq, >, \geq, <>$

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

4. Projection

- Eliminates columns, then removes duplicates
- Notation: $\Pi_{A1,\ldots,An}(R)$
- Example: project social-security number and names:
  - $\Pi_{SSN,\ Name}$ (Employee)
  - Output schema: Answer(SSN, Name)

Note that there are two parts:
(1) Eliminate columns (easy)
(2) Remove duplicates (hard)
In the "extended" algebra we will separate them.

SSN	Name	Salary
1234545	John	200000
5423341	John	600000
4352342	John	200000

$\Pi_{\text{Name,Salary}}$ (Employee)

Name	Salary
John	20000
John	60000

5. Cartesian Product

- Each tuple in R1 with each tuple in R2
- Notation: R1 $\times$ R2
- Example:
  - Employee $\times$ Dependents
- Very rare in practice; mainly used to express joins

Cartesian Product Example

Employee

Name	SSN
John	999999999
Tony	777777777

Dependents

EmployeeSSN	Dname
999999999	Emily
777777777	Joe

Employee x Dependents

Name	SSN	EmployeeSSN	Dname
John	999999999	999999999	Emily
John	999999999	777777777	Joe
Tony	777777777	999999999	Emily
Tony	777777777	777777777	Joe

Relational Algebra

- Five operators:
  - Union: $\cup$
  - Difference: -
  - Selection: σ
  - Projection: Π
  - Cartesian Product: $\times$
- Derived or auxiliary operators:
  - Intersection, complement
  - Joins (natural,equi-join, theta join, semi-join)
  - Renaming: ρ

Renaming

- Changes the schema, not the instance
- Notation: $\rho_{B1,\ldots,Bn}$ (R)
- Example:
  - $\rho_{LastName,\ SocSocNo}$ (Employee)
  - Output schema: Answer(LastName, SocSocNo)

Renaming Example

Employee

Name	SSN
John	999999999
Tony	777777777

$$\rho_{LastName,\ SocSocNo}\ (\textbf{Employee})$$

LastName	SocSocNo
John	999999999
Tony	777777777

Natural Join

- Notation: R1 $|\times|$ R2
- Meaning: R1 $|\times|$ R2 $= \Pi_A(\sigma_C(R1 \times R2))$
- Where:
  - The selection σ_C checks equality of all common attributes
  - The projection eliminates the duplicate common attributes

Natural Join Example

Employee

Name	SSN
John	999999999
Tony	777777777

Dependents

SSN	Dname
999999999	Emily
777777777	Joe

Employee $\bowtie$ Dependents =

$\Pi_{\text{Name, SSN, Dname}}(\sigma_{\text{SSN=SSN2}}(\text{Employee} \times \rho_{\text{SSN2, Dname}}(\text{Dependents})))$

Name	SSN	Dname
John	999999999	Emily
Tony	777777777	Joe

Natural Join

- R=

A	B
X	Y
X	Z
Y	Z
Z	V

S=

B	C
Z	U
V	W
Z	V

- R |×| 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 $|\times|$ S ?

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

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

Theta Join

- A join that involves a predicate
- R1 $|\times|_{\theta}$ R2 $= \sigma_{\theta}(R1 \times R2)$
- Here θ can be any condition

Eq-join

- A theta join where θ is an equality
- $R1 \;|\times|_{A=B}\; R2 \;=\; \sigma_{A=B}\,(R1 \times R2)$
- Example:
  - Employee $|\times|_{SSN=SSN}$ Dependents

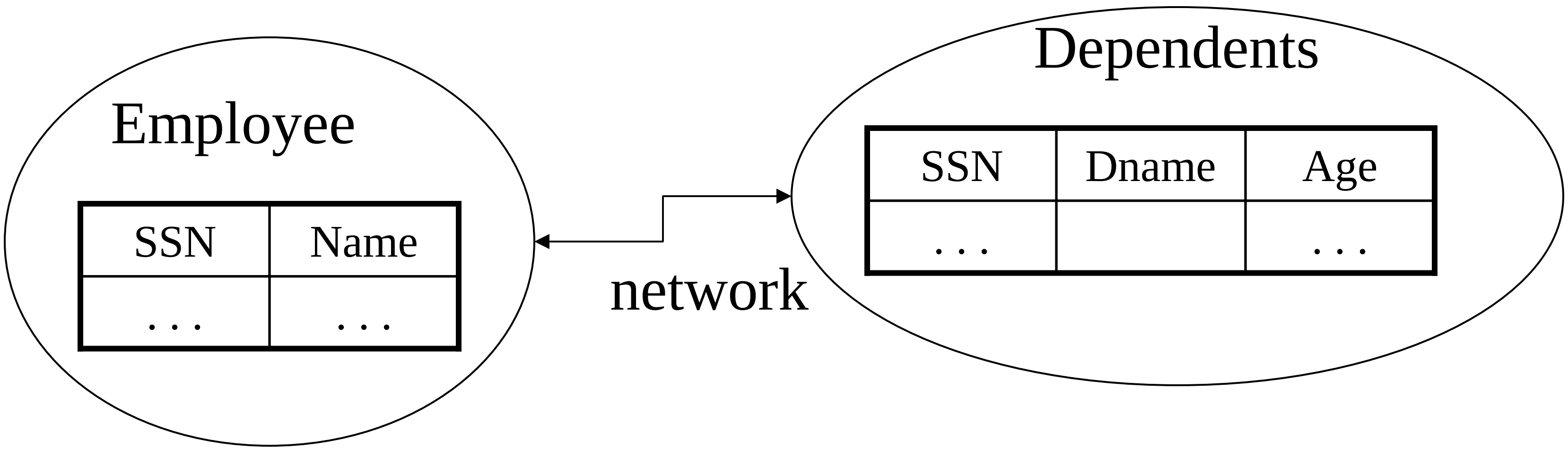
- Most useful join in practice

Semijoin

- $R \ltimes S = \Pi_{A1,\ldots,An} (R \bowtie S)$
- Where $A_1, \ldots, A_n$ are the attributes in R
- Example:
  - Employee $\ltimes$ Dependents

Semijoins in Distributed Databases

- Semijoins are used in distributed databases



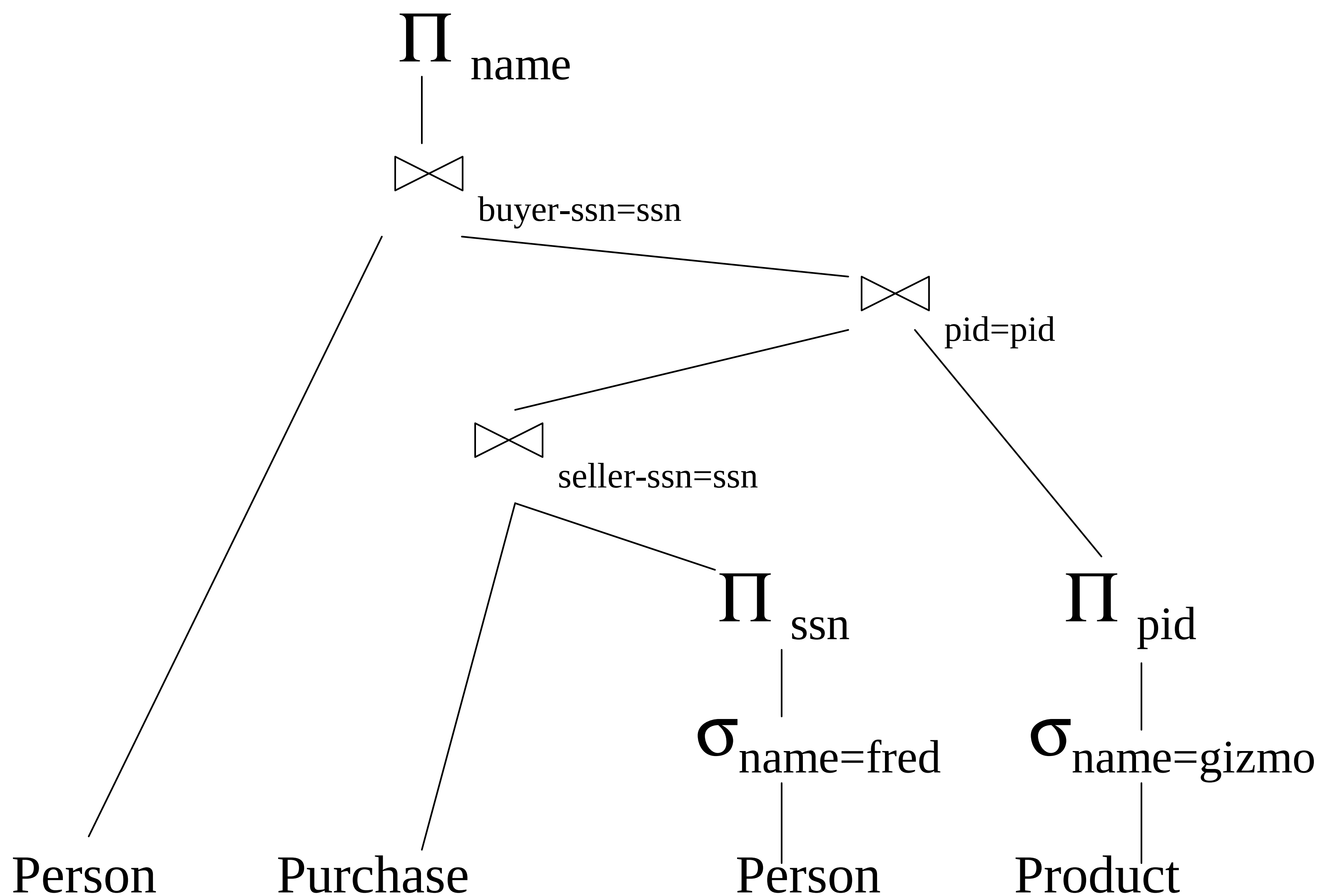
$$\text{Employee} \bowtie_{ssn=ssn} (\sigma_{age>71}(\text{Dependents}))$$

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

$$R = \text{Employee} \ltimes T$$

$$\text{Answer} = R \bowtie \text{Dependents}$$

Complex RA Expressions



Summary on the Relational Algebra

- A collection of 5 operators on relations
- Codd proved in 1970 that the relational algebra is equivalent to the relational calculus

Relational calculus/
First order logic/ SQL/
declarative language
=
WHAT

=

Relational algebra/
procedural language
=
HOW

Operations on Bags

A **bag** = a set with repeated elements

All operations need to be defined carefully on bags

- {a,b,b,c}$\cup${a,b,b,b,e,f,f}={a,a,b,b,b,b,b,c,e,f,f}
- {a,b,b,b,c,c} – {b,c,c,c,d} = {a,b,b,d}
- $\sigma_C(R)$: preserve the number of occurrences
- $\Pi_A(R)$: no duplicate elimination
- δ = explicit duplicate elimination
- Cartesian product, join: no duplicate elimination

Important ! Relational Engines work on bags, not sets !

Reading assignment: 5.3 – 5.4

Note: RA has Limitations !

- Cannot compute “transitive closure”

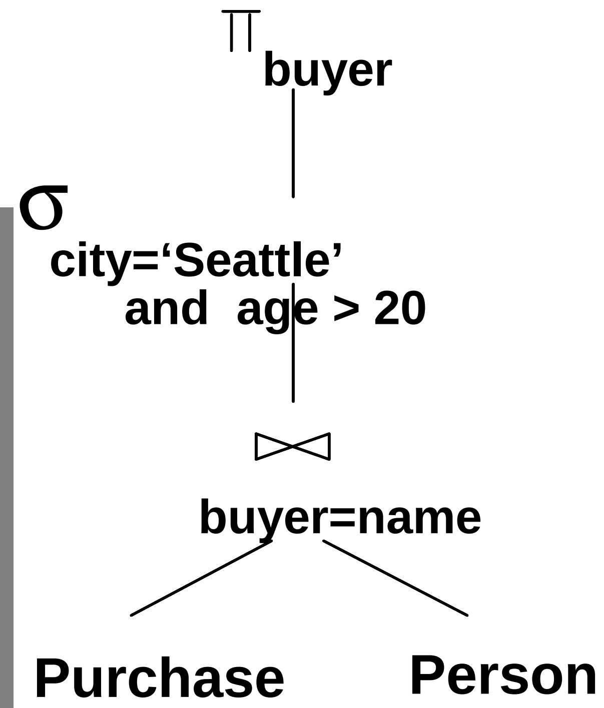
Name1	Name2	Relationship
Fred	Mary	Father
Mary	Joe	Cousin
Mary	Bill	Spouse
Nancy	Lou	Sister

- Find all direct and indirect relatives of Fred
- Cannot express in RA !!! Need to write C program

From SQL to RA

Purchase(buyer, product, city)
Person(name, age)

```
SELECT DISTINCT P.buyer
FROM Purchase P, Person Q
WHERE P.buyer=Q.name AND
      P.city='Seattle' AND
    Q.age > 20
```

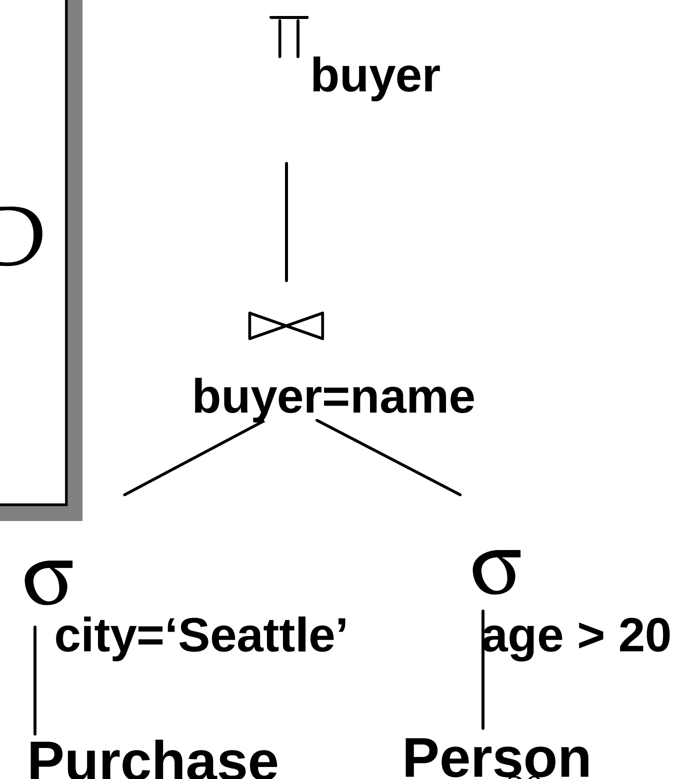


Also…

Purchase(buyer, product, city)

Person(name, age)

```
SELECT DISTINCT P.buyer
FROM Purchase P, Person Q
WHERE P.buyer=Q.name AND
      P.city='Seattle' AND
     Q.age > 20
```



Non-monontone Queries (in class)

Purchase(buyer, product, city)

Person(name, age)

```
SELECT DISTINCT P.product
FROM Purchase P
WHERE P.city='Seattle' AND
   not exists (select *
               from Purchase P2, Person Q
               where P2.product = P.product
                  and P2.buyer = Q.name
                  and Q.age > 20)
```

Extended Logical Algebra Operators (operate on Bags, not Sets)

- Union, intersection, difference
- Selection σ
- Projection Π
- Join |x|
- Duplicate elimination δ
- Grouping γ
- Sorting τ

Logical Query Plan

```
SELECT city, count(*)
FROM sales
GROUP BY city
HAVING sum(price) > 100
```

T3(city, c)

$\Pi_{\text{city, c}}$

T2(city,p,c)

$\sigma_{p > 100}$

T1(city,p,c)

$\gamma_{\text{city, sum(price)} \rightarrow \text{p, count(*)} \rightarrow \text{c}}$

sales(product, city, price)

T1, T2, T3 = temporary tables

Logical v.s. Physical Algebra

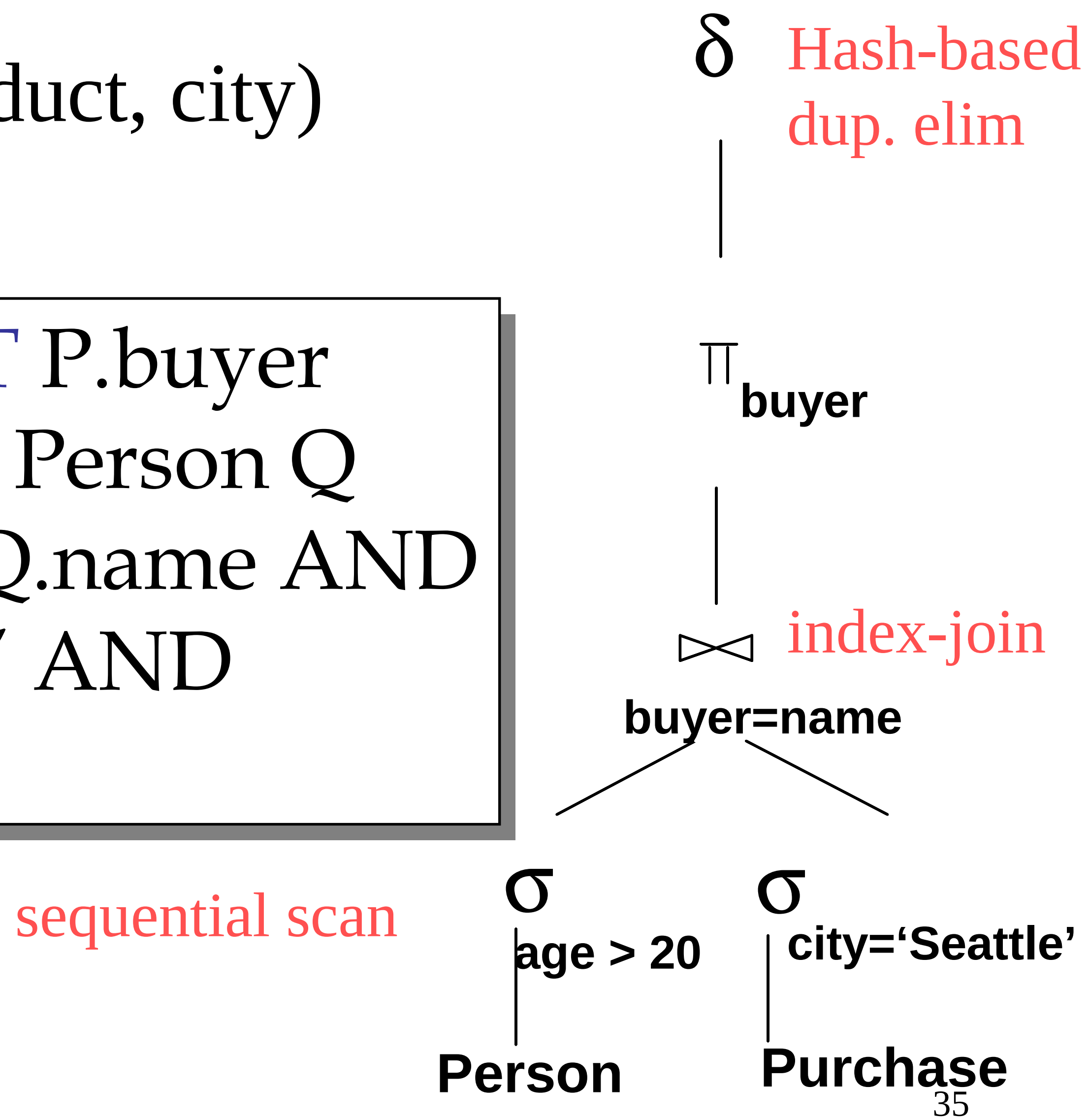
- We have seen the logical algebra so far:
  - Five basic operators, plus group-by, plus sort

- The Physical algebra refines each operator into a concrete algorithm

Physical Plan

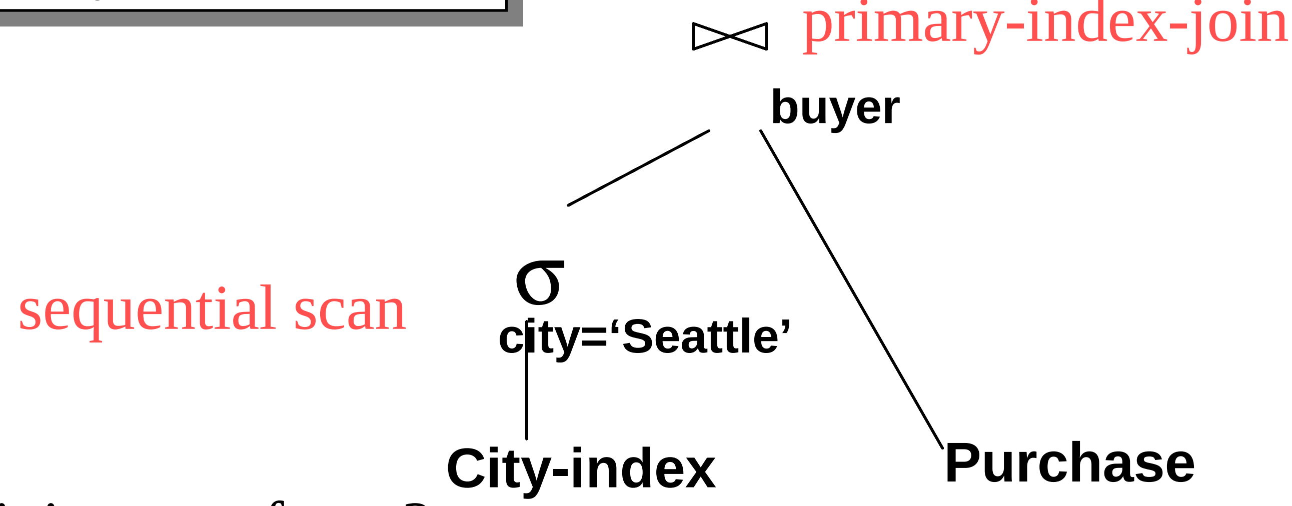
Purchase(buyer, product, city)
Person(name, age)

```
SELECT DISTINCT P.buyer
FROM Purchase P, Person Q
WHERE P.buyer=Q.name AND
      P.city='Seattle' AND
     Q.age > 20
```



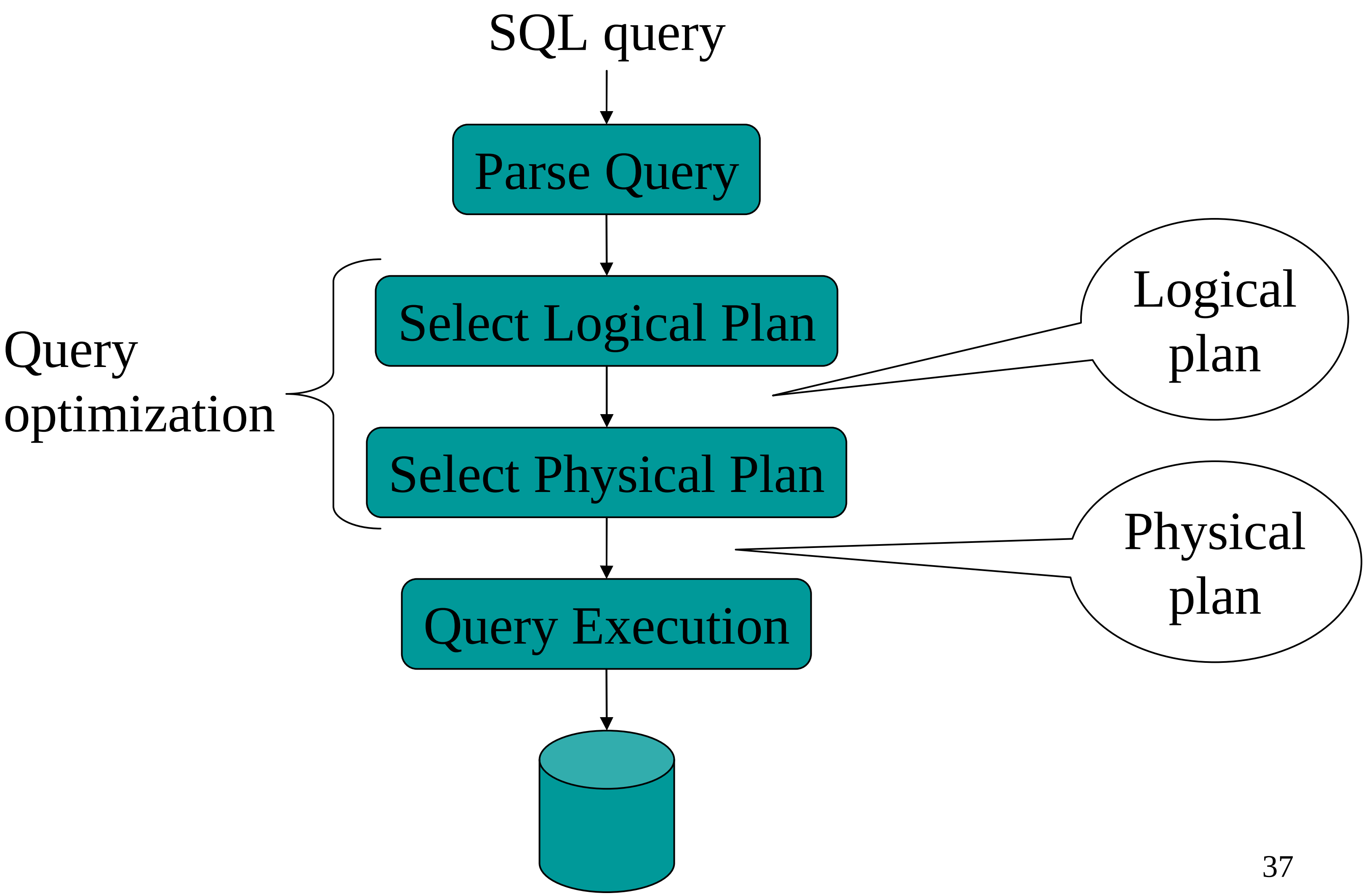
Physical Plans Can Be Subtle

```
SELECT *
FROM Purchase P
WHERE P.city='Seattle'
```



Where did the join come from ?

Architecture of a Database Engine



Question in Class

Logical operator:

Product(pname, cname) |×| Company(cname, city)

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

1.
2.
3.

Question in Class

Product(pname, cname) |x| Company(cname, city)

- 1000000 products
- 1000 companies

How much time do the following physical operators take if the data is **in main memory** ?

- Nested loop join time =
- Sort and merge = merge-join time =
- Hash join time =

Cost Parameters

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

result assumed to be delivered in main memory

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

NOTE: Book uses M for the number of blocks in R, and B for the number of blocks in main memory

Cost Parameters

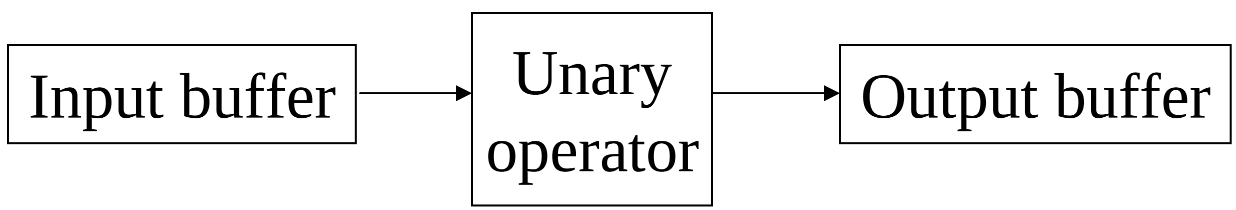
- *Clustered* table R:
  - Blocks consists only of records from this table
  - $B(R) << T(R)$
- *Unclustered* table R:
  - Its records are placed on blocks with other tables
  - $B(R) \approx T(R)$

- When a is a key, $V(R,a) = T(R)$
- When a is not a key, $V(R,a)$

Selection and Projection

Selection $\sigma(R)$, projection $\Pi(R)$

- Both are *tuple-at-a-time* algorithms
- Cost: B(R)

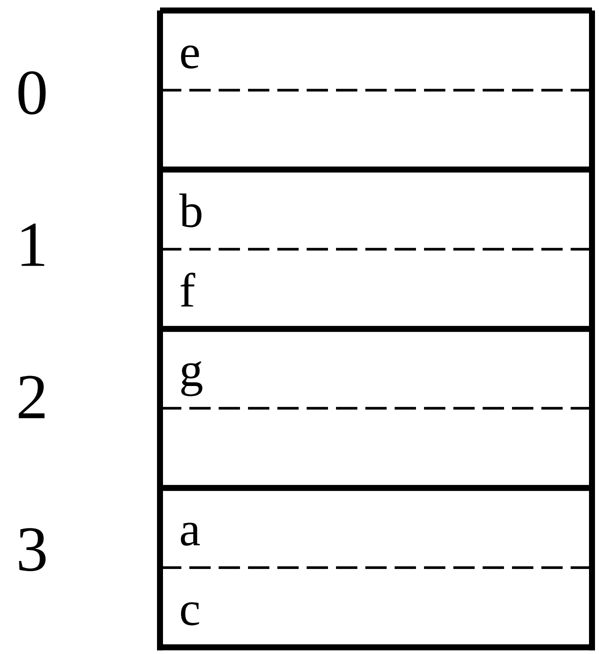


Hash Tables

- Key data structure used in many operators
- May also be used for indexes, as alternative to B+trees
- Recall basics:
  - There are n ***buckets***
  - A hash function f(k) maps a key k to {0, 1, …, n-1}
  - Store in bucket f(k) a pointer to record with key k
- Secondary storage: bucket = block, use overflow blocks when needed

Hash Table Example

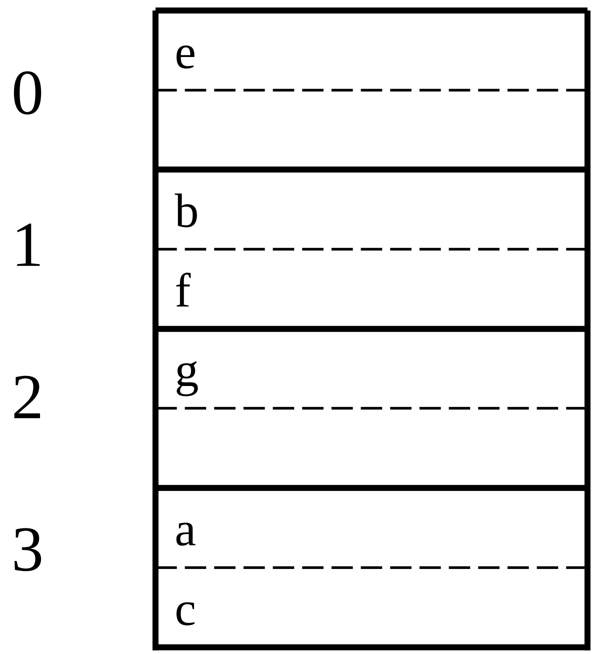
- Assume 1 bucket (block) stores 2 keys + pointers
- h(e)=0
- h(b)=h(f)=1
- h(g)=2
- h(a)=h(c)=3



Here: h(x) = x mod 4

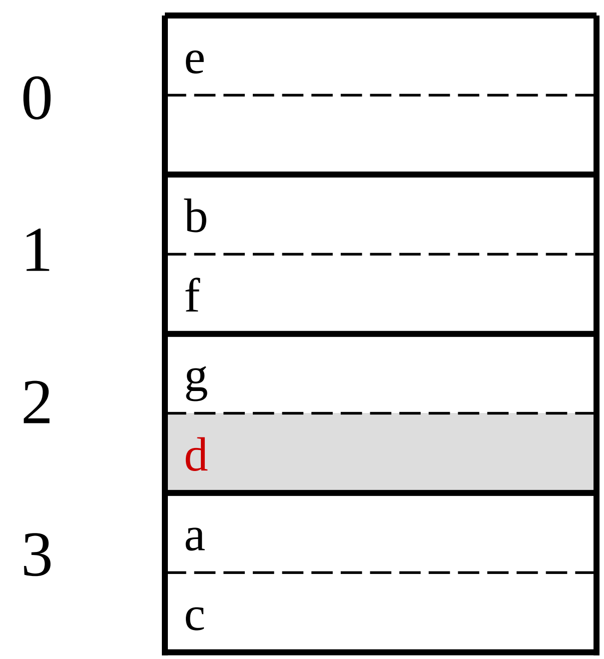
Searching in a Hash Table

- Search for a:
- Compute h(a)=3
- Read bucket 3
- 1 disk access



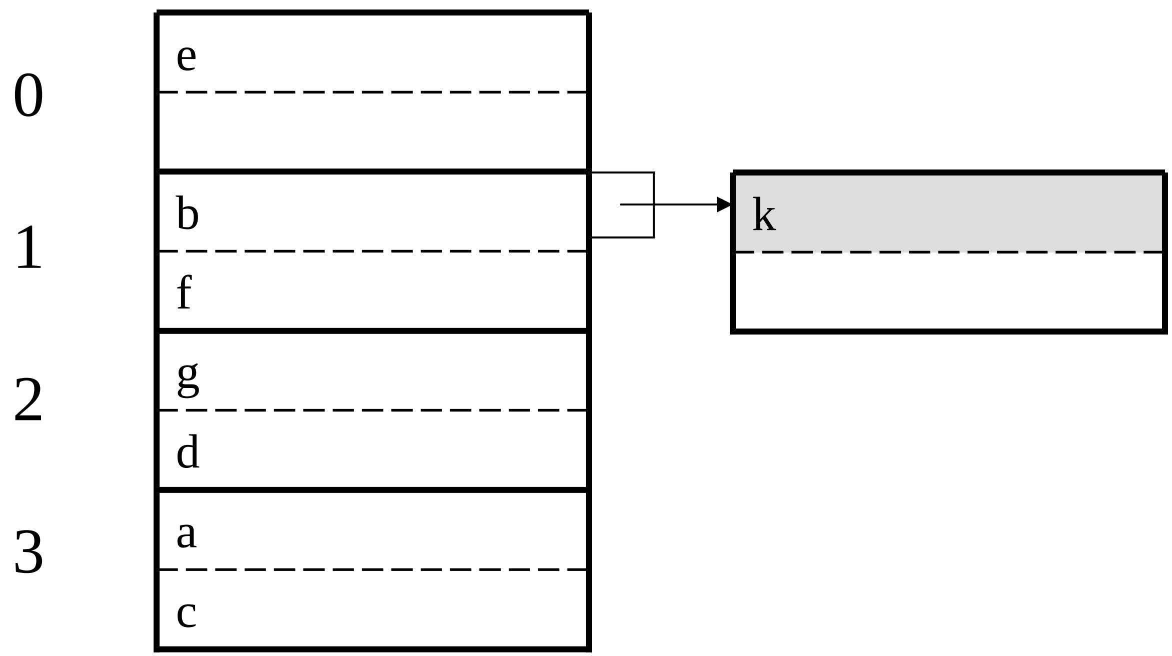
Insertion in Hash Table

- Place in right bucket, if space
- E.g. h(d)=2



Insertion in Hash Table

- Create overflow block, if no space
- E.g. h(k)=1



- More over-flow blocks may be needed

Hash Table Performance

- Excellent, if no overflow blocks
- Degrades considerably when number of keys exceeds the number of buckets (I.e. many overflow blocks).

Main Memory Hash Join

Hash join: R |x| S

- Scan S, build buckets in main memory
- Then scan R and join

- Cost: B(R) + B(S)
- Assumption: B(S) <= M

Main Memory Duplicate Elimination

Duplicate elimination $\delta(R)$

- Hash table in main memory

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

Main Memory Grouping

Grouping:

Product(name, department, quantity)

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

Main memory hash table

Question: How ?

Nested Loop Joins

- Tuple-based nested loop R ⋈ S

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

- Cost: T(R) B(S) when S is clustered
- Cost: T(R) T(S) when S is unclustered

Nested Loop Joins

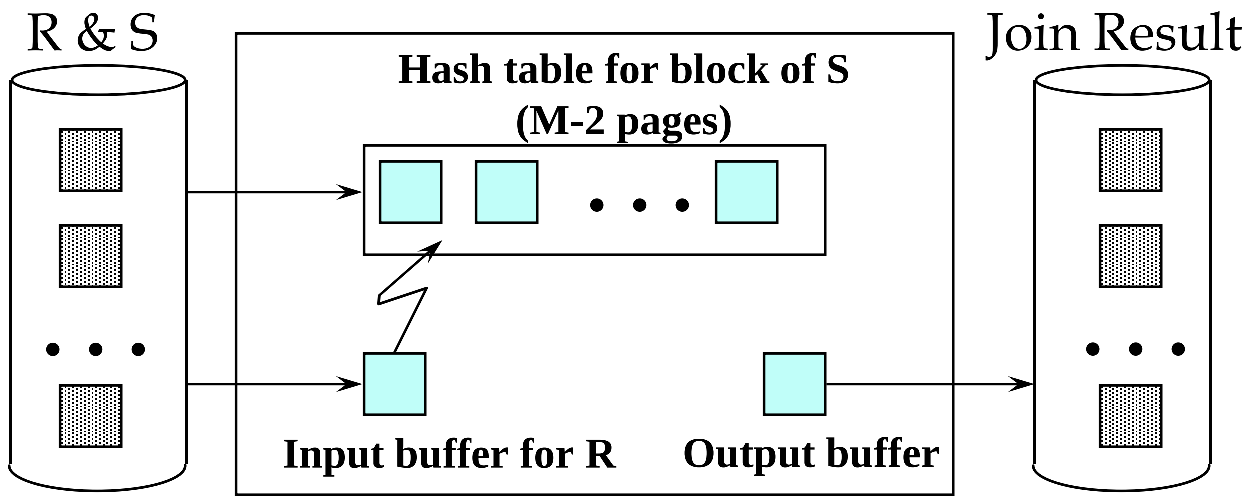
- We can be much more clever
- *Question*: how would you compute the join in the following cases ? What is the cost ?
  - B(R) = 1000, B(S) = 2, M = 4
  - B(R) = 1000, B(S) = 3, M = 4
  - B(R) = 1000, B(S) = 6, M = 4

Nested Loop Joins

- 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)
```

Nested Loop Joins



Nested Loop Joins

- Block-based Nested Loop Join
- Cost:
  - 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)
  - Total cost: B(S) + B(S)B(R)/(M-2)
- Notice: it is better to iterate over the smaller relation first
- R |x| S: R=outer relation, S=inner relation

Index Based Selection

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

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

- Unclustered index on a: cost T(R)/V(R,a)
  - We have seen that this is like a join

Index Based Selection

- Example:

$B(R) = 2000$
$T(R) = 100{,}000$
$V(R, a) = 20$

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

- Table scan (assuming R is clustered):
  - $B(R) = 2{,}000$ I/Os
- Index based selection:
  - If index is clustered: $B(R)/V(R,a) = 100$ I/Os
  - If index is unclustered: $T(R)/V(R,a) = 5{,}000$ I/Os

- Lesson: 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):

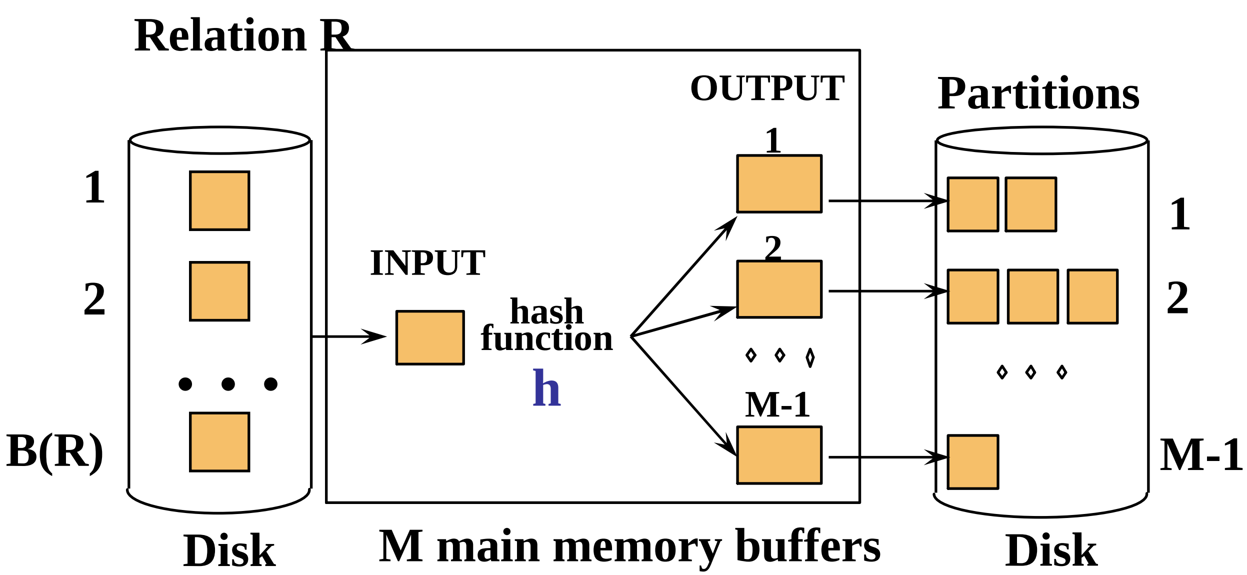
- If index is clustered: $B(R) + T(R)B(S)/V(S,a)$
- If index is unclustered: $B(R) + T(R)T(S)/V(S,a)$

Operations on Very Large Tables

- Partitioned hash algorithms
- Merge-sort algorithms

Partitioned Hash Algorithms

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



- Does each bucket fit in main memory ?
  - Yes if B(R)/M <= M,   i.e. B(R) <= M^2

Duplicate Elimination

- Recall: $\delta(R)$ = duplicate elimination
- Step 1. Partition R into buckets
- Step 2. Apply δ to each bucket (may read in main memory)

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

Grouping

- Recall: $\gamma(R)$ = 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: $B(R) <= M^2$

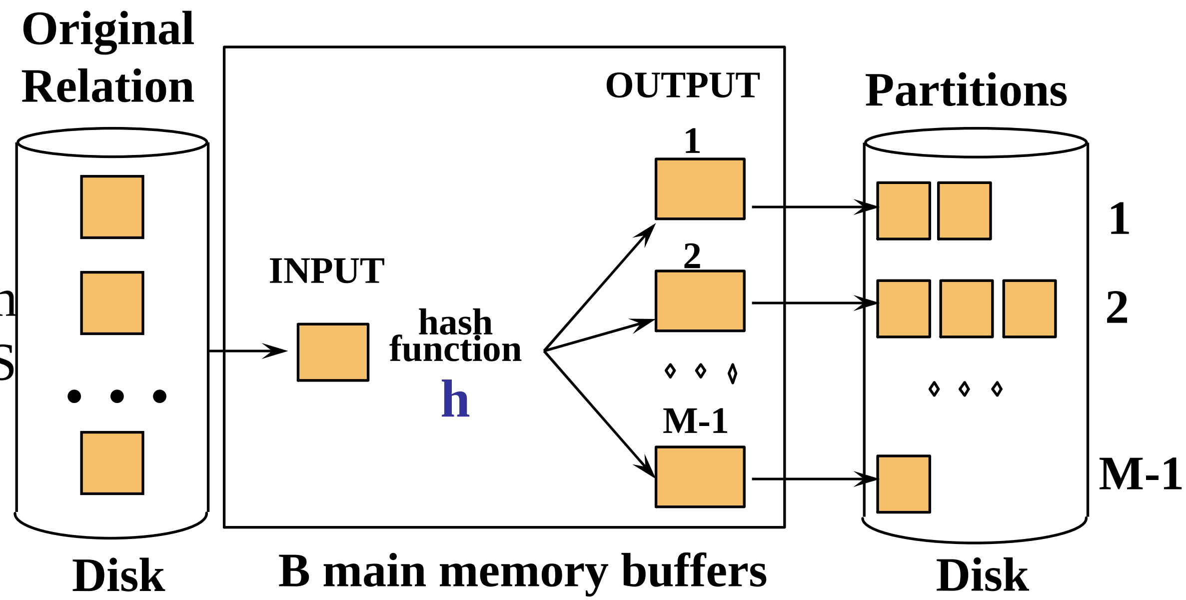
Partitioned Hash Join

R |x| S

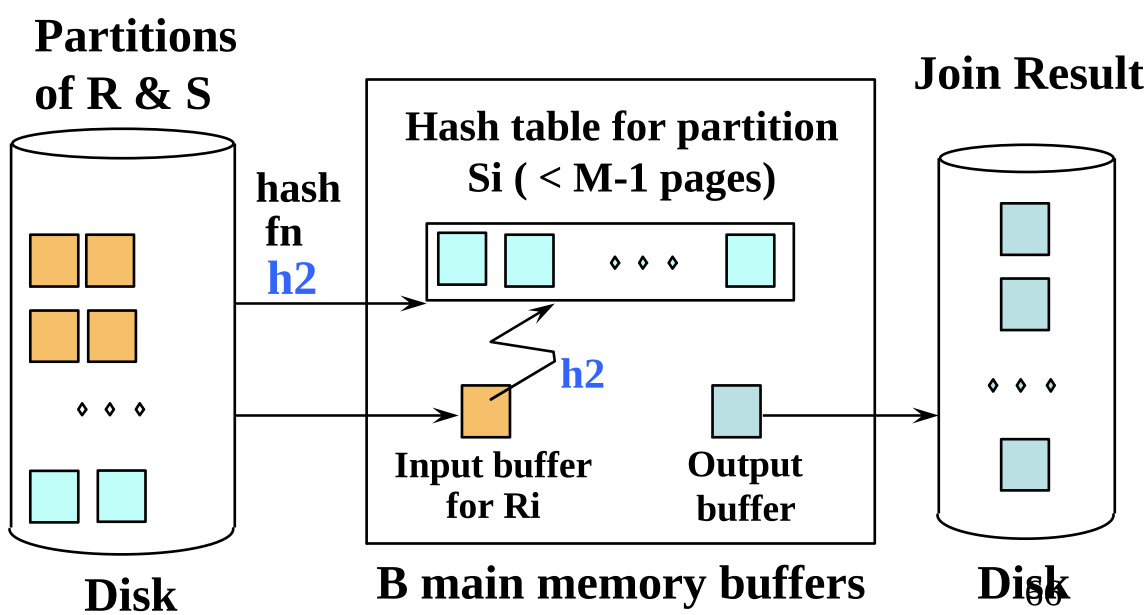
- Step 1:
  - Hash S into M buckets
  - send all buckets to disk
- Step 2
  - Hash R into M buckets
  - Send all buckets to disk
- Step 3
  - Join every pair of buckets

Hash-Join

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



- **Probe**: 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)) <= M^2

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 Join Algorithm

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

- Assuming $t/k * B(S) \gg k - t$: $t/k = M/B(S)$

Hybrid Join Algorithm

- How many I/Os ?
- Cost of partitioned hash join: 3B(R) + 3B(S)
- Hybrid join saves 2 I/Os for a t/k fraction of buckets
- Hybrid join 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

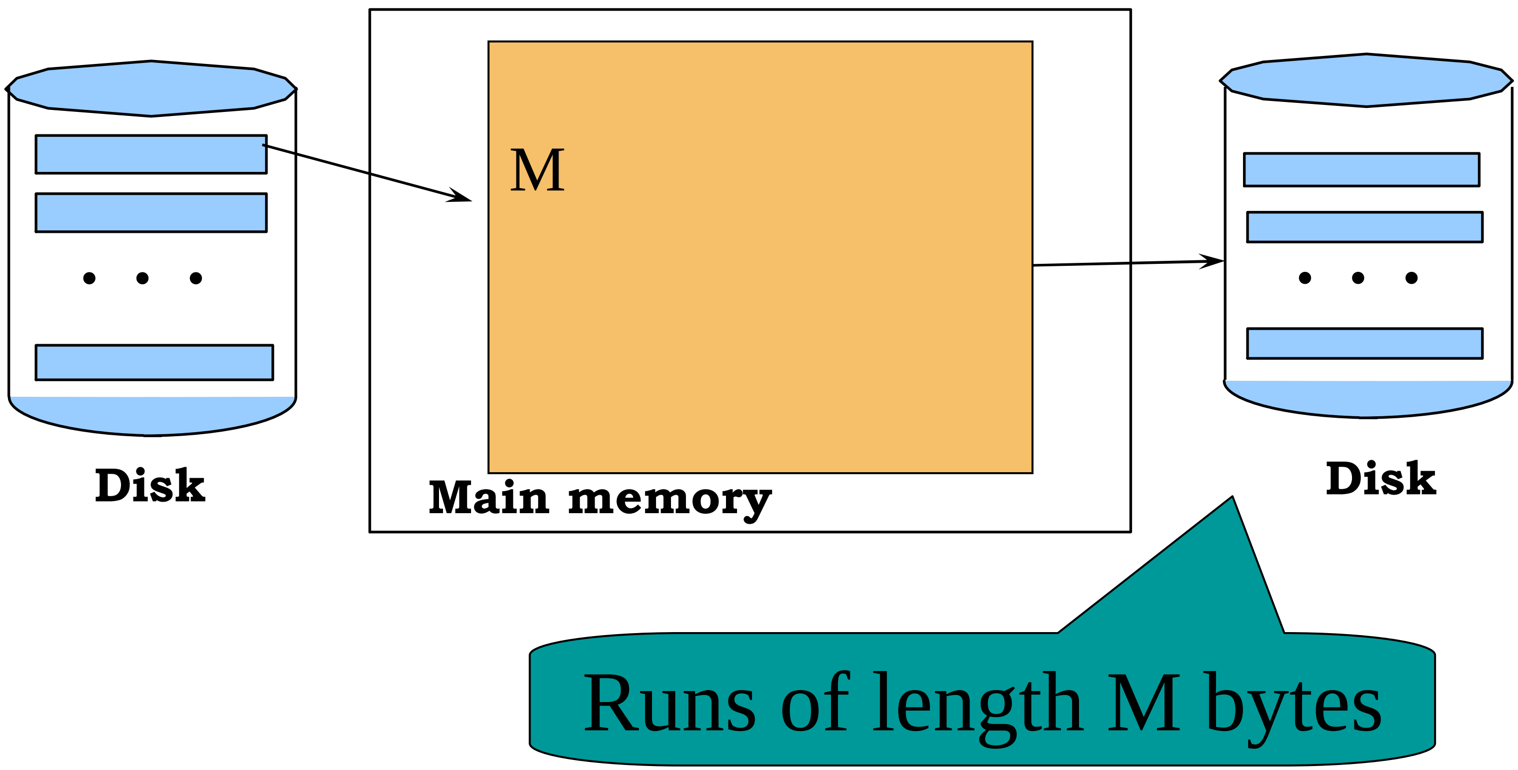
- Question in class: what is the real advantage of the hybrid algorithm ?

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, for when $B < M^2$

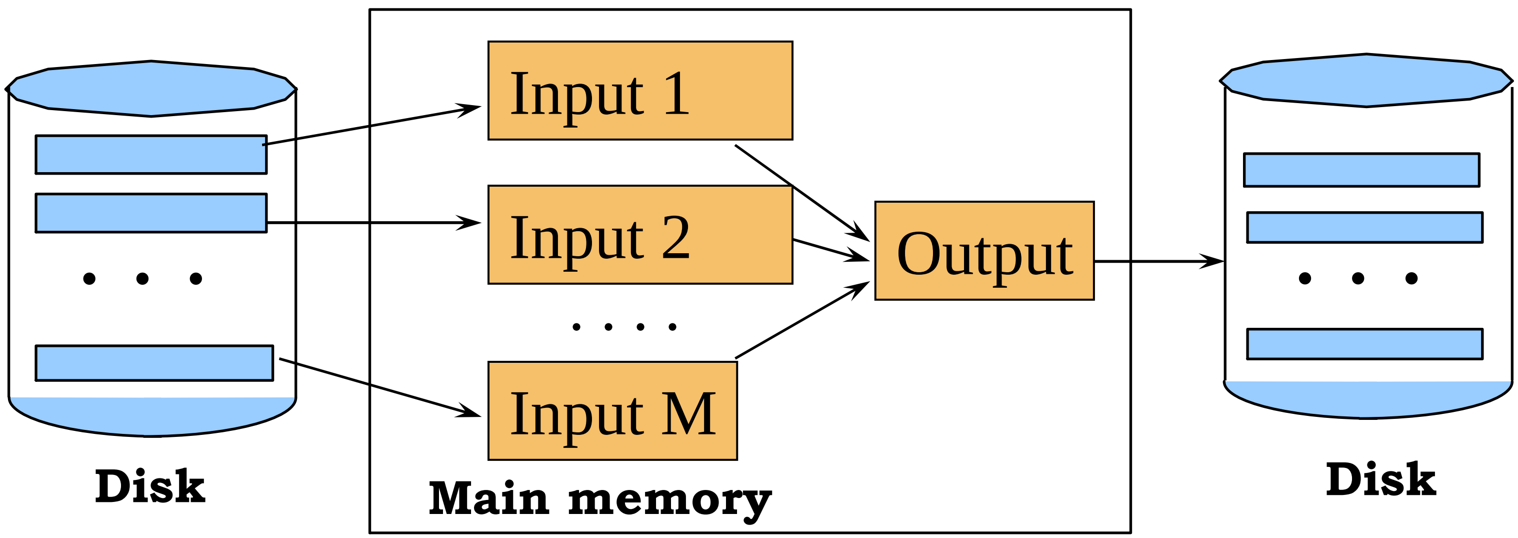
External Merge-Sort: Step 1

- 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) $\approx$ M^2



If B <= M^2 then we are done

Cost of External Merge Sort

- Read+write+read = 3B(R)

- Assumption: $B(R) \le M^2$

Extensions, Discussions

- Blocked I/O
  - Group b blocks and process them together
  - Same effect as increasing the block size by a factor b
- Double buffering:
  - Keep two buffers for each input or output stream
  - During regular merge on one set of buffers, perform the I/O on the other set of buffers
  - Decreases M to M/2

Extensions, Discussions

- Initial run formation (level 0-runs)
  - Main memory sort (usually Quicksort): results in initial runs of length M
  - Replacement selection: start by reading a chunk of file of size M, organize as heap, start to output the smallest elements in increasing order; as the buffer empties, read more data; *the new elements are added to the heap as long as they are > the last element output*. Expected run lengths turns out to be approx 2M

Duplicate Elimination

Duplicate elimination $\delta(R)$

- 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: $B(\delta(R)) <= M^2$

Grouping

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

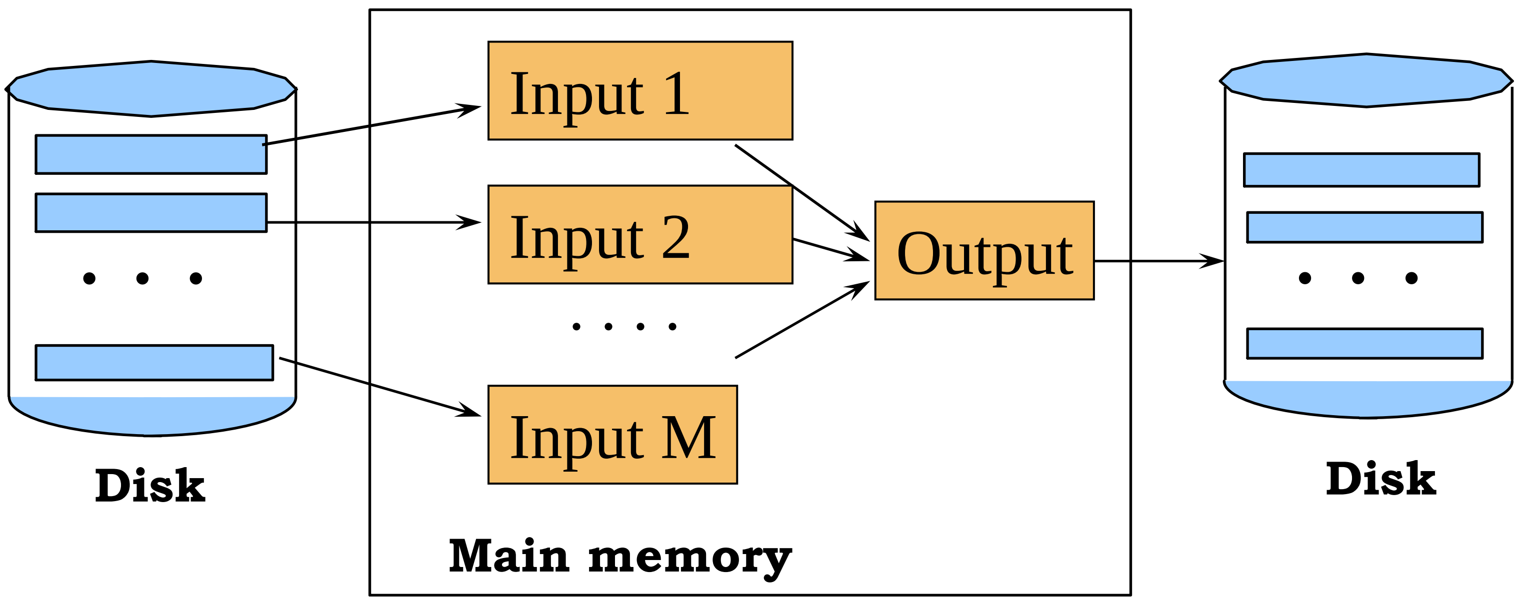
- Same as before: sort, then compute the sum(b) for each group of a's
- Total cost: 3B(R)
- Assumption: B(R) <= M^2

Merge-Join

Join R |x| 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

If B <= M^2 then we are done

Two-Pass Algorithms Based on Sorting

Join R |x| 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: $B(R) + B(S) <= M^2$

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
  - $B(R)+B(S) <= M^2$

Example

Product(<u>pname</u>, maker), **Company**(<u>cname</u>, city)

```
Select Product.pname
From  Product, Company
Where Product.maker=Company.cname
   and  Company.city = "Seattle"
```

- How do we execute this query ?

Example

Product(pname, maker), **Company**(cname, city)

Assume:

Clustered index: **Product**.pname, **Company**.cname

Unclustered index: **Product**.maker, **Company**.city

Logical Plan:

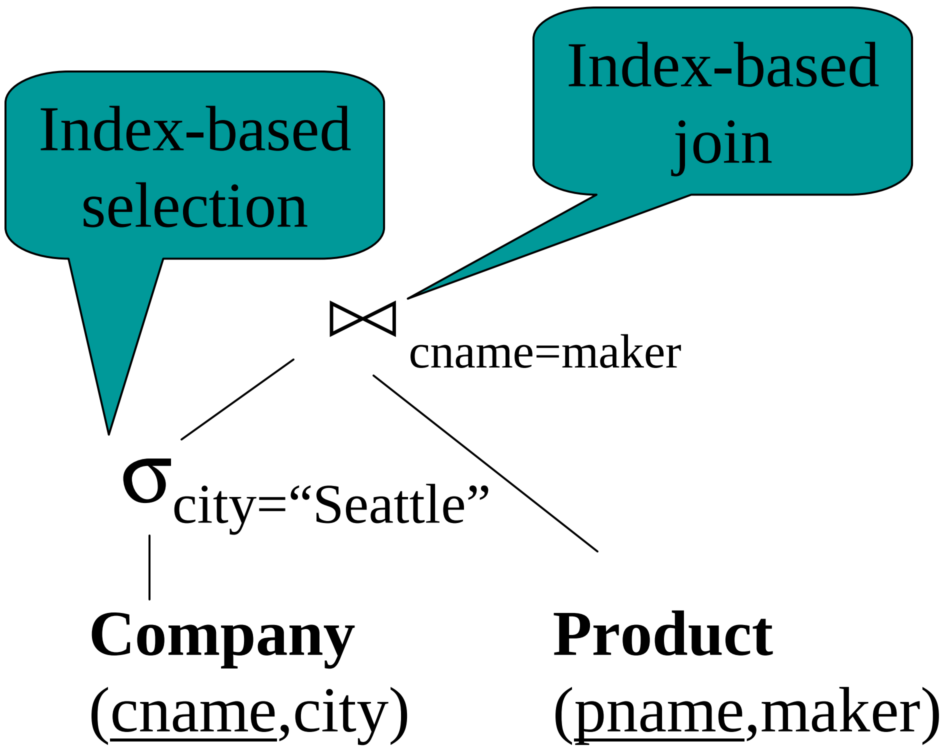
$\bowtie_{maker=cname}$

$\sigma_{city=\text{“Seattle”}}$

Product (<u>pname</u>,maker)

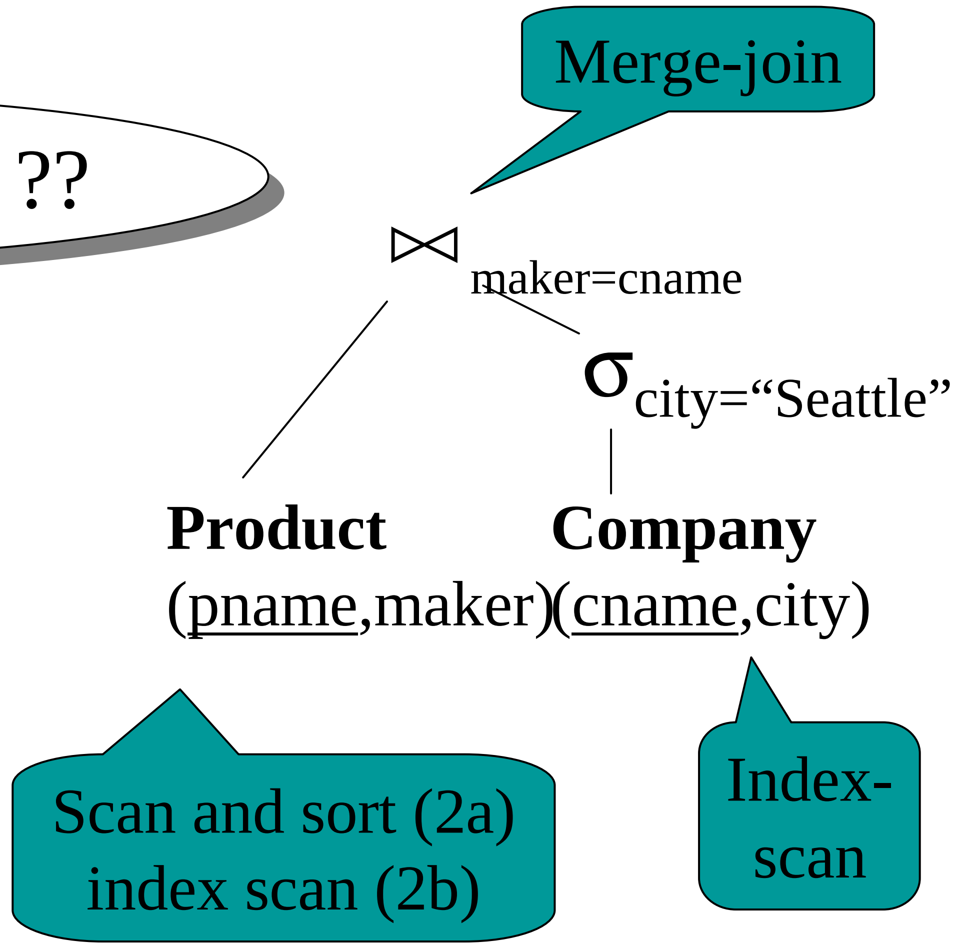
Company (<u>cname</u>,city)

Physical plan 1:

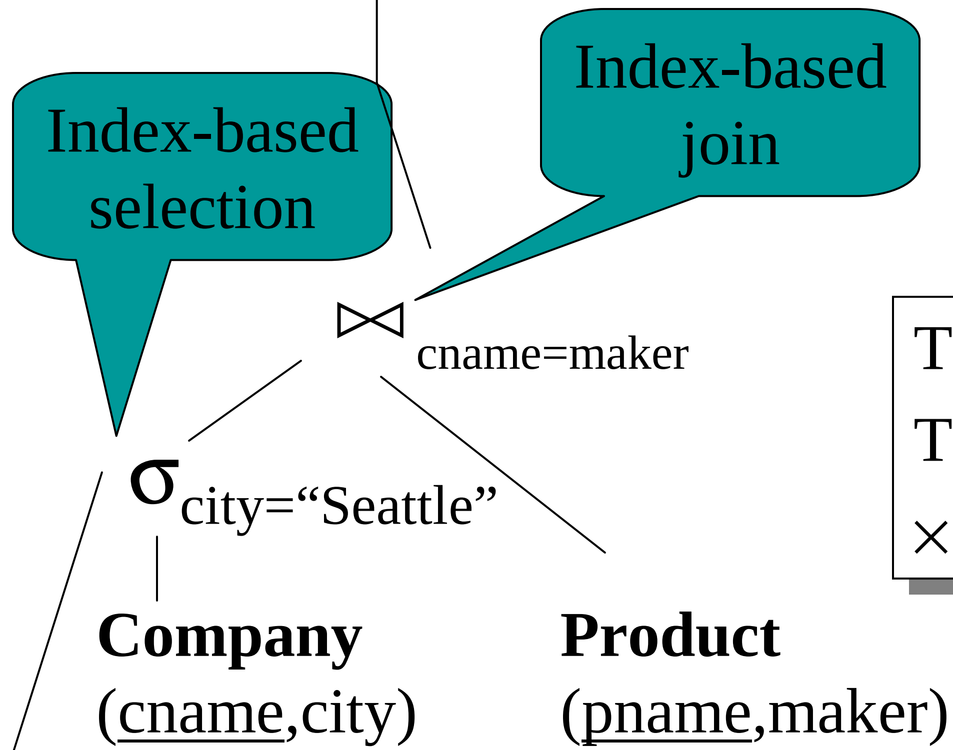


Physical plans 2a and 2b:

Which one is better ??



Physical plan 1:



$\times$ T(**Product**) / V(**Product**, maker)

Total cost:

T(**Company**) / V(**Company**, city)

$\times$ T(**Product**) / V(**Product**, maker)

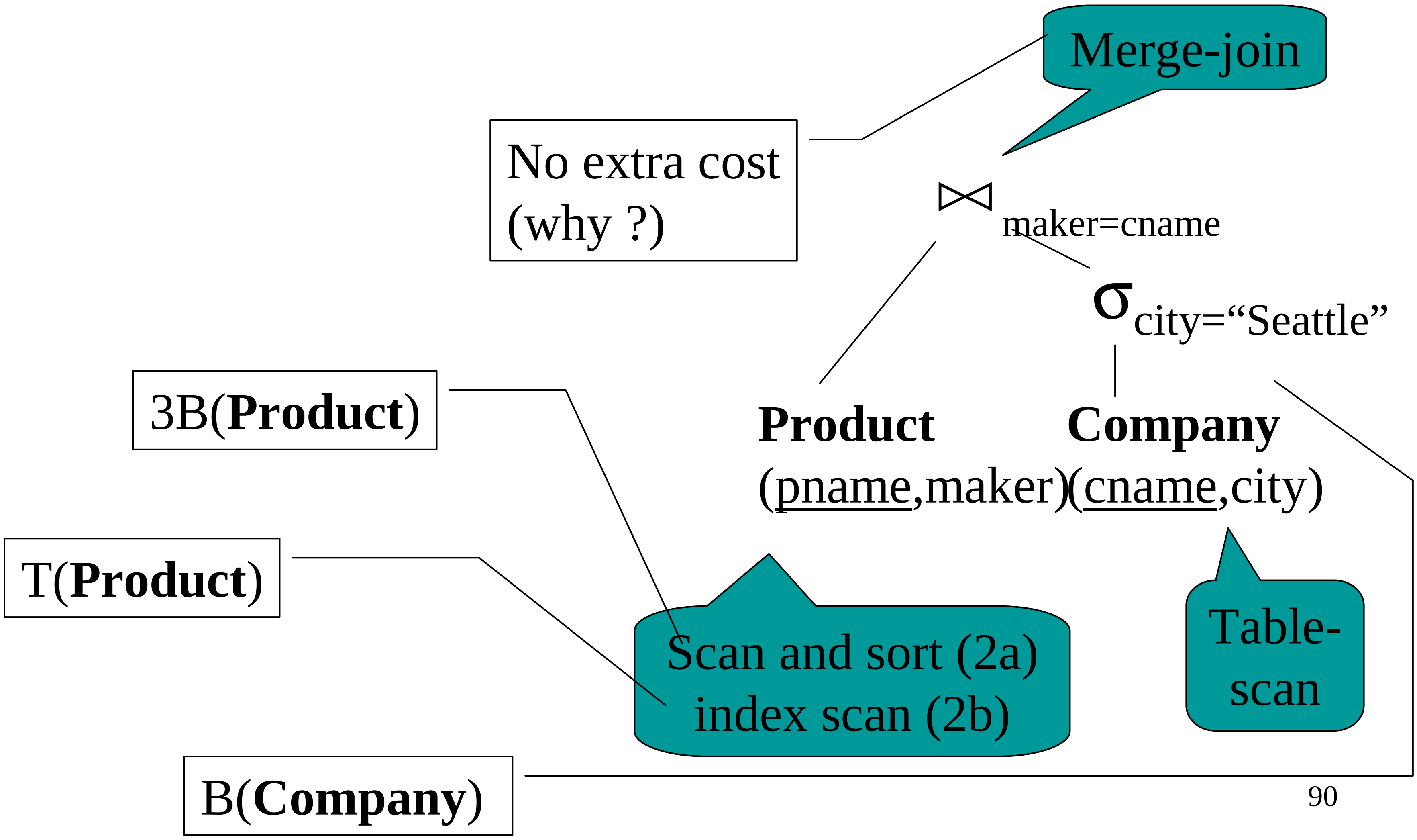
T(**Company**) / V(**Company**, city)

Total cost:

(2a): 3B(**Product**) + B(**Company**)

(2b): T(**Product**) + B(**Company**)

Physical plans 2a and 2b:



Plan 1: T(**Company**)/V(**Company**,city) $\times$ T(**Product**)/V(**Product**,maker)

Plan 2a: B(**Company**) + 3B(**Product**)

Plan 2b: B(**Company**) + T(**Product**)

Which one is better ??

It depends on the data !!

Example

T(**Company**) = 5,000   B(**Company**) = 500   M = 100
T(**Product**) = 100,000   B(**Product**) = 1,000

We may assume V(**Product**, maker) ≈ T(**Company**) (why ?)

- Case 1: V(**Company**, city) ≈ T(**Company**)

V(**Company**,city) = 2,000

- Case 2: V(**Company**, city) << T(**Company**)

V(**Company**,city) = 20

Which Plan is Best ?

Plan 1: T(**Company**)/V(**Company**,city) × T(**Product**)/V(**Product**,maker)
Plan 2a: B(**Company**) + 3B(**Product**)
Plan 2b: B(**Company**) + T(**Product**)

Case 1:

Case 2:

Lessons

- Need to consider several physical plan
  - even for one, simple logical plan
- No magic “best” plan: depends on the data
- In order to make the right choice
  - need to have ***statistics*** over the data
  - the B’s, the T’s, the V’s

Query Optimzation

- Have a SQL query Q
- Create a plan P

HOW ??

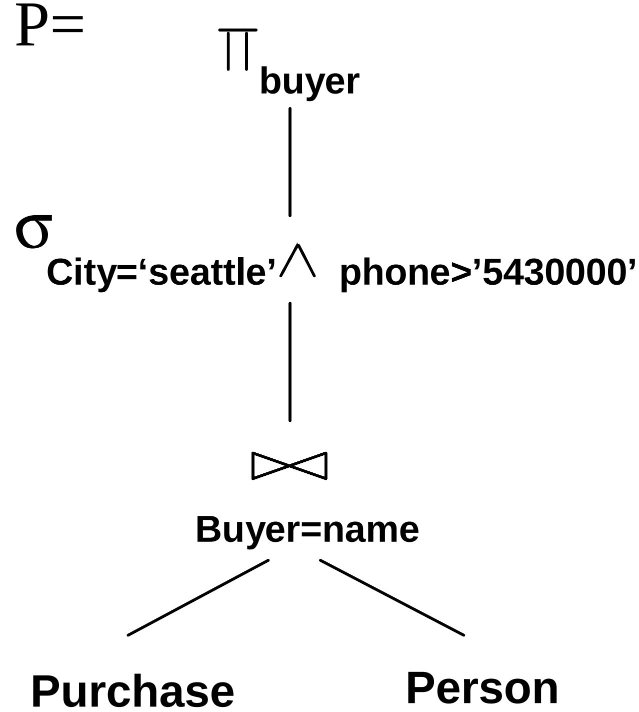
- Find equivalent plans P = P’ = P’’ = …
- Choose the “cheapest”.

Logical Query Plan

```
SELECT  P.buyer
FROM Purchase P, Person Q
WHERE P.buyer=Q.name AND
      P.city='seattle' AND
      Q.phone > '5430000'
```

Purchasse(buyer, city)
Person(name, phone)

P=



In class:
find a "better" plan P'

Logical Query Plan

Q=

```
SELECT city, sum(quantity)
FROM sales
GROUP BY city
HAVING sum(quantity) < 100
```

P=

T2(city,p)

$\sigma_{p < 100}$

T1(city,p)

$\gamma_{city,\ sum(quantity) \rightarrow p}$

sales(product, city, quantity)

In class:
find a “better” plan P’

Optimization

- Main idea: rewrite a logical query plan into an equivalent “more efficient” logical plan

The three components of an optimizer

We need three things in an optimizer:

- Algebraic laws
- An optimization algorithm
- A cost estimator

Algebraic Laws

- Commutative and Associative Laws

  $R \cup S = S \cup R$, $R \cup (S \cup T) = (R \cup S) \cup T$

  $R \mid\times\mid S = S \mid\times\mid R$, $R \mid\times\mid (S \mid\times\mid T) = (R \mid\times\mid S) \mid\times\mid T$

  $R \mid\times\mid S = S \mid\times\mid R$, $R \mid\times\mid (S \mid\times\mid T) = (R \mid\times\mid S) \mid\times\mid T$

- Distributive Laws

  $R \mid\times\mid (S \cup T) = (R \mid\times\mid S) \cup (R \mid\times\mid T)$

Algebraic Laws

- Laws involving selection:
  - $\sigma_{C \text{ AND } C'}(R) = \sigma_C(\sigma_{C'}(R)) = \sigma_C(R) \cap \sigma_{C'}(R)$
  - $\sigma_{C \text{ OR } C'}(R) = \sigma_C(R) \cup \sigma_{C'}(R)$
  - $\sigma_C(R \,|\times|\, S) = \sigma_C(R) \,|\times|\, S$
- When C involves only attributes of R
  - $\sigma_C(R - S) = \sigma_C(R) - S$
  - $\sigma_C(R \cup S) = \sigma_C(R) \cup \sigma_C(S)$
  - $\sigma_C(R \,|\times|\, S) = \sigma_C(R) \,|\times|\, S$

Algebraic Laws

- Example: R(A, B, C, D), S(E, F, G)
  - $\sigma_{F=3}(R \bowtie_{D=E} S) =$?
  - $\sigma_{A=5 \text{ AND } G=9}(R \bowtie_{D=E} S) =$?

Algebraic Laws

- Laws involving projections

  $\Pi_M(R \mid\times\mid S) = \Pi_M(\Pi_P(R) \mid\times\mid \Pi_Q(S))$

  $\Pi_M(\Pi_N(R)) = \Pi_{M,N}(R)$

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

  $\Pi_{A,B,G}(R \mid\times\mid_{D=E} S) = \Pi_{?}(\Pi_?(R) \mid\times\mid_{D=E} \Pi_?(S))$

Algebraic Laws

- Laws involving grouping and aggregation:

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

  $\gamma_{A,\ agg(B)}(\delta(R)) = \gamma_{A,\ agg(B)}(R)$ if agg is "duplicate insensitive"

- Which of the following are "duplicate insensitive" ? sum, count, avg, min, max

$\gamma_{A,\ agg(D)}(R(A,B) \,|\times|_{B=C}\, S(C,D)) =$
$\quad \gamma_{A,\ agg(D)}(R(A,B) \,|\times|_{B=C}\, (\gamma_{C,\ agg(D)}S(C,D)))$

Optimizations Based on Semijoins

THIS IS ADVANCED STUFF; NOT ON THE FINAL

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

Optimizations Based on Semijoins

Semijoins: a bit of theory (see [AHV])

- Given a query:

$$Q \text{ :- } \Pi \, (\sigma \, (R_1 \; |x| \; R_2 \; |x| \; . \; . \; . \; |x| \; R_n \,))$$

- A full reducer for Q is a program:

$$R_{i1} := R_{i1} \ltimes R_{j1}$$
$$R_{i2} := R_{i2} \ltimes R_{j2}$$
$$.....$$
$$R_{ip} := R_{ip} \ltimes R_{jp}$$

- Such that no dangling tuples remain in any relation

Optimizations Based on Semijoins

- Example:

  Q(A,E) :- R1(A,B) |x| R2(B,C) |x| R3(C,D,E)

- A full reducer is:

  R2(B,C) := R2(B,C) |x R1(A,B)

  R3(C,D,E) := R3(C,D,E) |x R2(B,C)

  R2(B,C) := R2(B,C) |x R3(C,D,E)

  R1(A,B) := R1(A,B) |x R2(B,C)

The new tables have only the tuples necessary to compute Q(E)

Optimizations Based on Semijoins

- Example:

$\boxed{\text{Q(E) :- R1(A,B) |x| R2(B,C) |x| R3(A,C, E)}}$

- Doesn’t have a full reducer (we can reduce forever)

Theorem a query has a full reducer iff it is “acyclic”

Optimizations Based on Semijoins

- Semijoins in [Chaudhuri'98]

```
CREATE VIEW DepAvgSal As (
          SELECT E.did, Avg(E.Sal) AS avgsal
          FROM Emp E
          GROUP BY E.did)

SELECT E.eid, E.sal
FROM Emp E, Dept D, DepAvgSal V
WHERE E.did = D.did AND E.did = V.did
          AND E.age < 30 AND D.budget > 100k
          AND E.sal > V.avgsal
```

Optimizations Based on Semijoins

- First idea:

```
CREATE VIEW LimitedAvgSal As (
        SELECT E.did, Avg(E.Sal) AS avgsal
        FROM Emp E, Dept D
         WHERE E.did = D.did AND D.buget > 100k
        GROUP BY E.did)

SELECT E.eid, E.sal
FROM Emp E, Dept D, LimitedAvgSal V
WHERE E.did = D.did AND E.did = V.did
        AND E.age < 30 AND D.budget > 100k
        AND E.sal > V.avgsal
```

Optimizations Based on Semijoins

- Better: full reducer

```
CREATE VIEW PartialResult AS
        (SELECT E.id, E.sal, E.did
        FROM Emp E, Dept D
        WHERE E.did=D.did AND E.age < 30
        AND D.budget > 100k)

CREATE VIEW Filter AS
        (SELECT DISTINCT P.did FROM PartialResult P)

CREATE VIEW LimitedAvgSal AS
        (SELECT E.did, Avg(E.Sal) AS avgsal
        FROM Emp E, Filter F
        WHERE E.did = F.did GROUP BY E.did)
```

Optimizations Based on Semijoins

```
SELECT P.eid, P.sal
FROM PartialResult P, LimitedDepAvgSal V
WHERE P.did = V.did AND P.sal > V.avgsal
```

Cost-based Optimizations

- Main idea: apply algebraic laws, until estimated cost is minimal
- Practically: start from partial plans, introduce operators one by one
  - Will see in a few slides
- Problem: there are too many ways to apply the laws, hence too many (partial) plans

Cost-based Optimizations

Approaches:

- **Top-down**: the partial plan is a top fragment of the logical plan

- **Bottom up**: the partial plan is a bottom fragment of the logical plan

Dynamic Programming

Originally proposed in System R

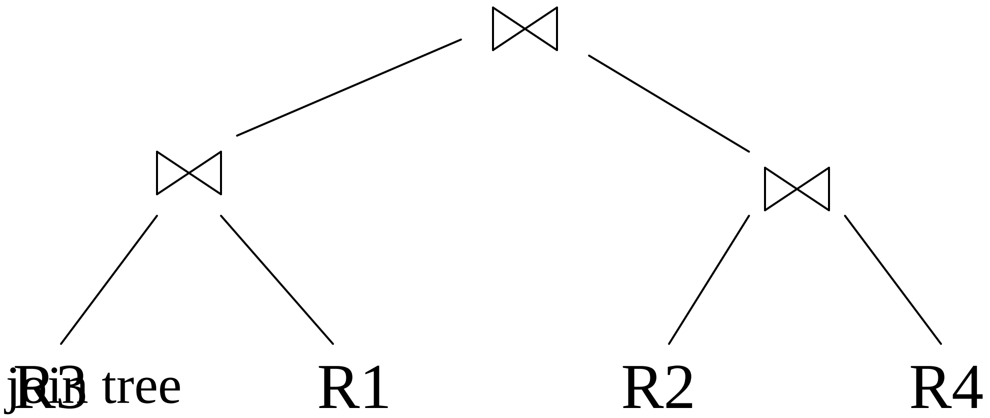
- Only handles single block queries:

```
SELECT list
FROM    list
WHERE cond1 AND cond2 AND . . . AND condk
```

- Heuristics: selections down, projections up
- Dynamic programming: ***join reordering***

Join Trees

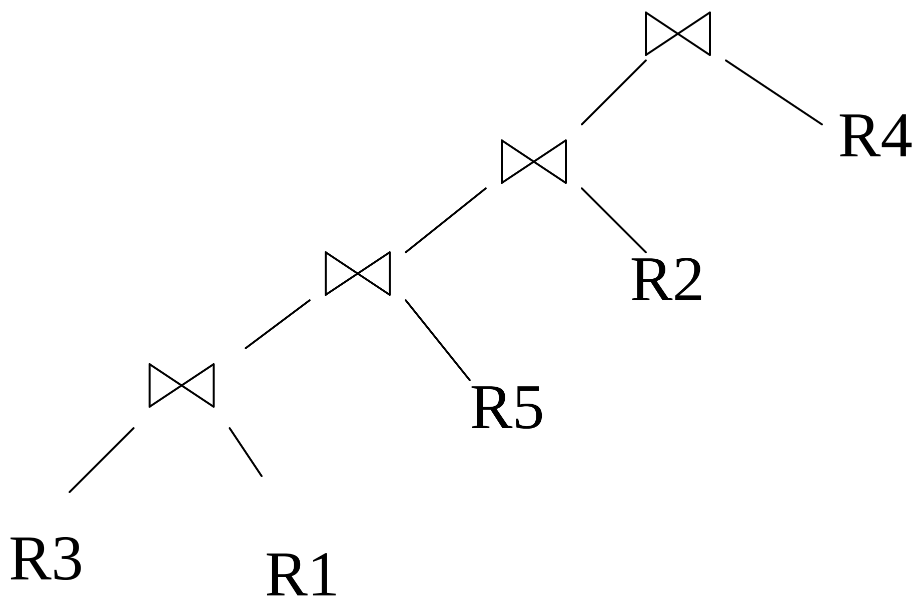
- R1 |×| R2 |×| …. |×| Rn
- Join tree:



- A plan = a join tree
- A partial plan = a subtree of a join tree

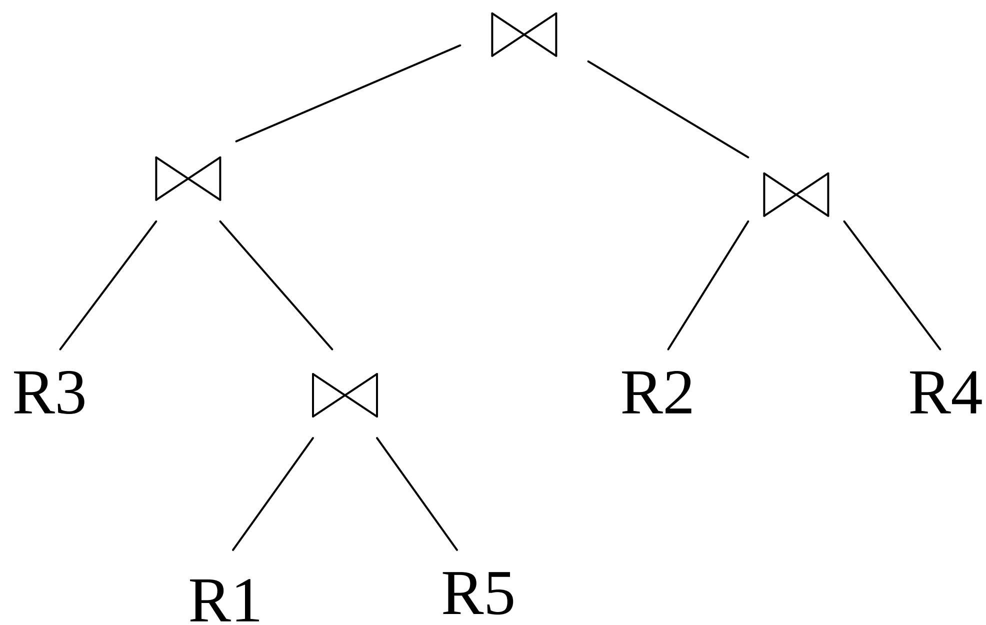
Types of Join Trees

- Left deep:



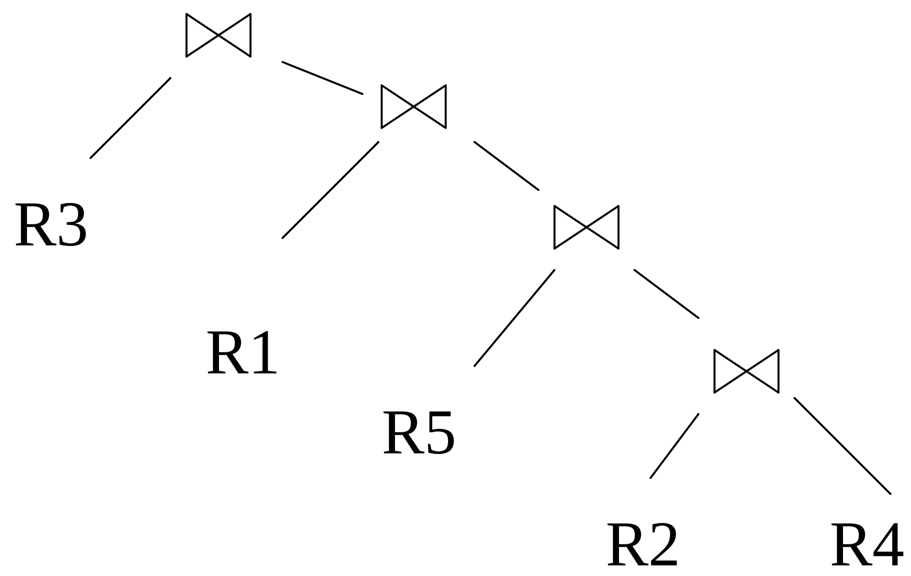
Types of Join Trees

- Bushy:



Types of Join Trees

- Right deep:



Dynamic Programming

- Given: a query $R1 \bowtie R2 \bowtie \ldots \bowtie Rn$
- Assume we have a function cost() that gives us the cost of every join tree
- Find the best join tree for the query

Dynamic Programming

- Idea: for each subset of {R1, …, Rn}, compute the best plan for that subset
- In increasing order of set cardinality:
  - Step 1: for {R1}, {R2}, …, {Rn}
  - Step 2: for {R1,R2}, {R1,R3}, …, {Rn-1, Rn}
  - …
  - Step n: for {R1, …, Rn}
- It is a bottom-up strategy
- A subset of {R1, …, Rn} is also called a *subquery*

Dynamic Programming

- For each subquery $Q \subseteq \{R1, \ldots, Rn\}$ compute the following:
  - Size(Q)
  - A best plan for Q: Plan(Q)
  - The cost of that plan: Cost(Q)

Dynamic Programming

- **Step 1**: For each $\{R_i\}$ do:
  - $\text{Size}(\{R_i\}) = B(R_i)$
  - $\text{Plan}(\{R_i\}) = R_i$
  - $\text{Cost}(\{R_i\}) =$ (cost of scanning R_i)

Dynamic Programming

- **Step i**: For each Q $\subseteq\{R_1, \ldots, R_n\}$ of cardinality i do:
  - Compute Size(Q) (later…)
  - For every pair of subqueries Q', Q''
    s.t. Q = Q' $\cup$ Q''
    compute cost(Plan(Q') $|\times|$ Plan(Q''))
  - Cost(Q) = the smallest such cost
  - Plan(Q) = the corresponding plan

Dynamic Programming

- Return Plan($\{R_1, \ldots, R_n\}$)

Dynamic Programming

To illustrate, we will make the following simplifications:

- $\text{Cost}(P_1 \bowtie P_2) = \text{Cost}(P_1) + \text{Cost}(P_2) +$ size(intermediate result(s))
- Intermediate results:
  - If P_1 = a join, then the size of the intermediate result is $\text{size}(P_1)$, otherwise the size is 0
  - Similarly for P_2
- Cost of a scan = 0

Dynamic Programming

- Example:
- Cost(R5 |×| R7) = 0 (no intermediate results)
- Cost((R2 |×| R1) |×| R7)
  = Cost(R2 |×| R1) + Cost(R7) + size(R2 |×| R1)
  = size(R2 |×| R1)

Dynamic Programming

- Relations: R, S, T, U
- Number of tuples: 2000, 5000, 3000, 1000
- Size estimation: T(A $|\times|$ B) = 0.01*T(A)*T(B)

Subquery	Size	Cost	Plan
RS			
RT			
RU			
ST			
SU			
TU			
RST			
RSU			
RTU			
STU			
RSTU			

Subquery	Size	Cost	Plan
RS	100k	0	RS
RT	60k	0	RT
RU	20k	0	RU
ST	150k	0	ST
SU	50k	0	SU
TU	30k	0	TU
RST	3M	60k	(RT)S
RSU	1M	20k	(RU)S
RTU	0.6M	20k	(RU)T
STU	1.5M	30k	(TU)S
RSTU	30M	60k+50k=110k	(RT)(SU)

Reducing the Search Space

- Left-linear trees v.s. Bushy trees

- Trees without cartesian product

Example: R(A,B) |×| S(B,C) |×| T(C,D)

Plan: (R(A,B) |×| T(C,D)) |×| S(B,C) has a cartesian product – most query optimizers will not consider it

Dynamic Programming: Summary

- Handles only join queries:
  - Selections are pushed down (i.e. early)
  - Projections are pulled up (i.e. late)

- Takes exponential time in general, BUT:
  - Left linear joins may reduce time
  - Non-cartesian products may reduce time further

Rule-Based Optimizers

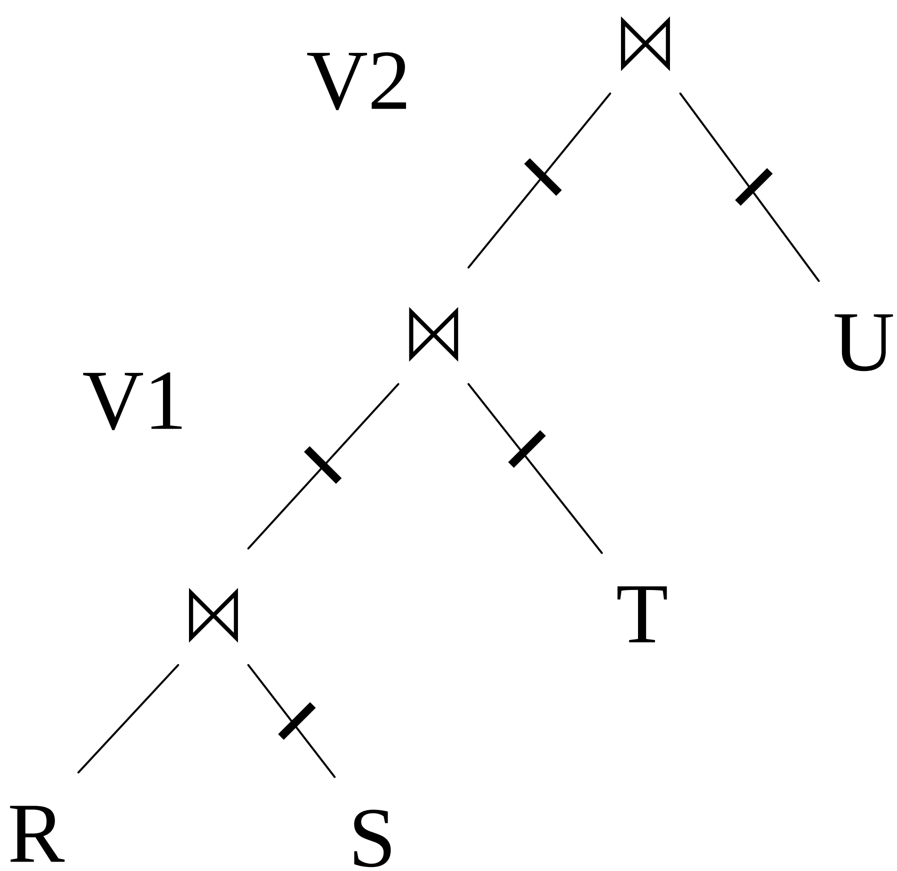
- ***Extensible*** collection of rules
  - Rule = Algebraic law with a direction
- Algorithm for firing these rules
  - Generate many alternative plans, in some order
  - Prune by cost

- Volcano (later SQL Sever)
- Starburst (later DB2)

Completing the Physical Query Plan

- Choose algorithm to implement each operator
  - Need to account for more than cost:
    - How much memory do we have ?
    - Are the input operand(s) sorted ?
- Decide for each intermediate result:
  - To materialize
  - To pipeline

Materialize Intermediate Results Between Operators




```
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)
```

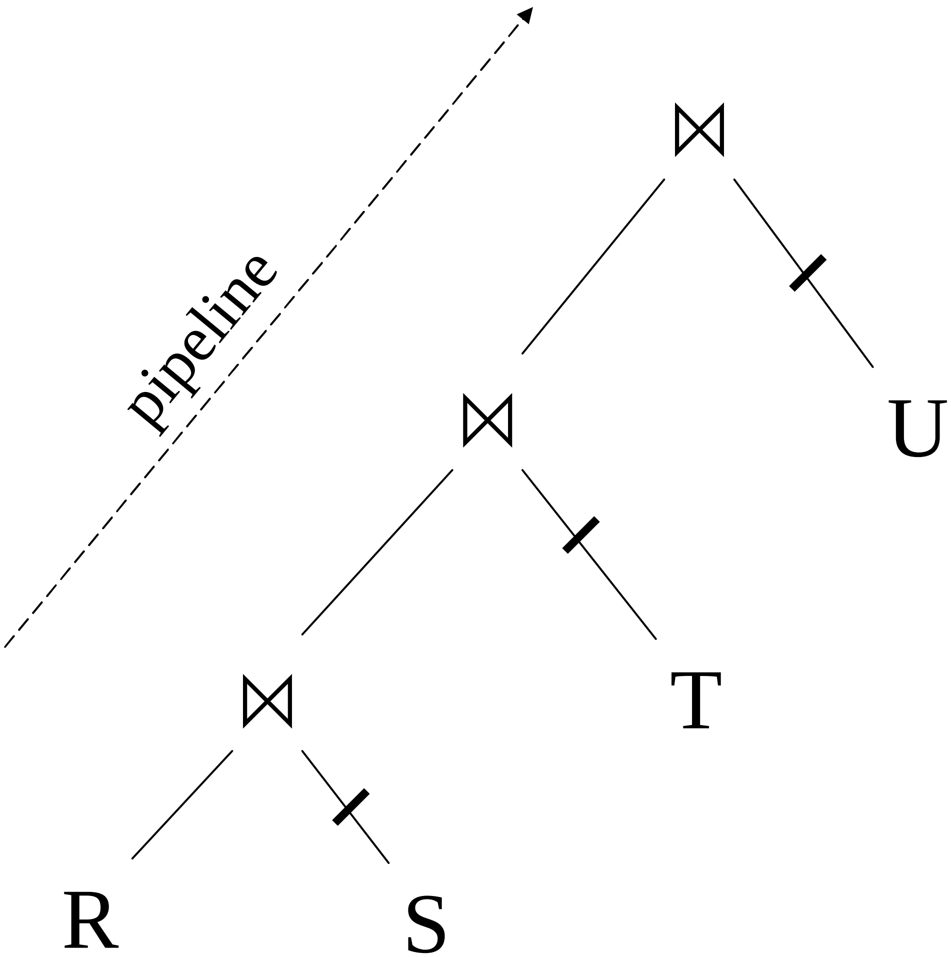
Materialize Intermediate Results Between Operators

Question in class

Given B(R), B(S), B(T), B(U)

- What is the total cost of the plan ?
  - Cost =
- How much main memory do we need ?
  - M =

Pipeline Between Operators



HashTable1 $\leftarrow$ S
HashTable2 $\leftarrow$ T
HashTable3 $\leftarrow$ U
repeat read(R, x)
y $\leftarrow$ join(HashTable1, x)
z $\leftarrow$ join(HashTable2, y)
u $\leftarrow$ join(HashTable3, z)
write(Answer, u)

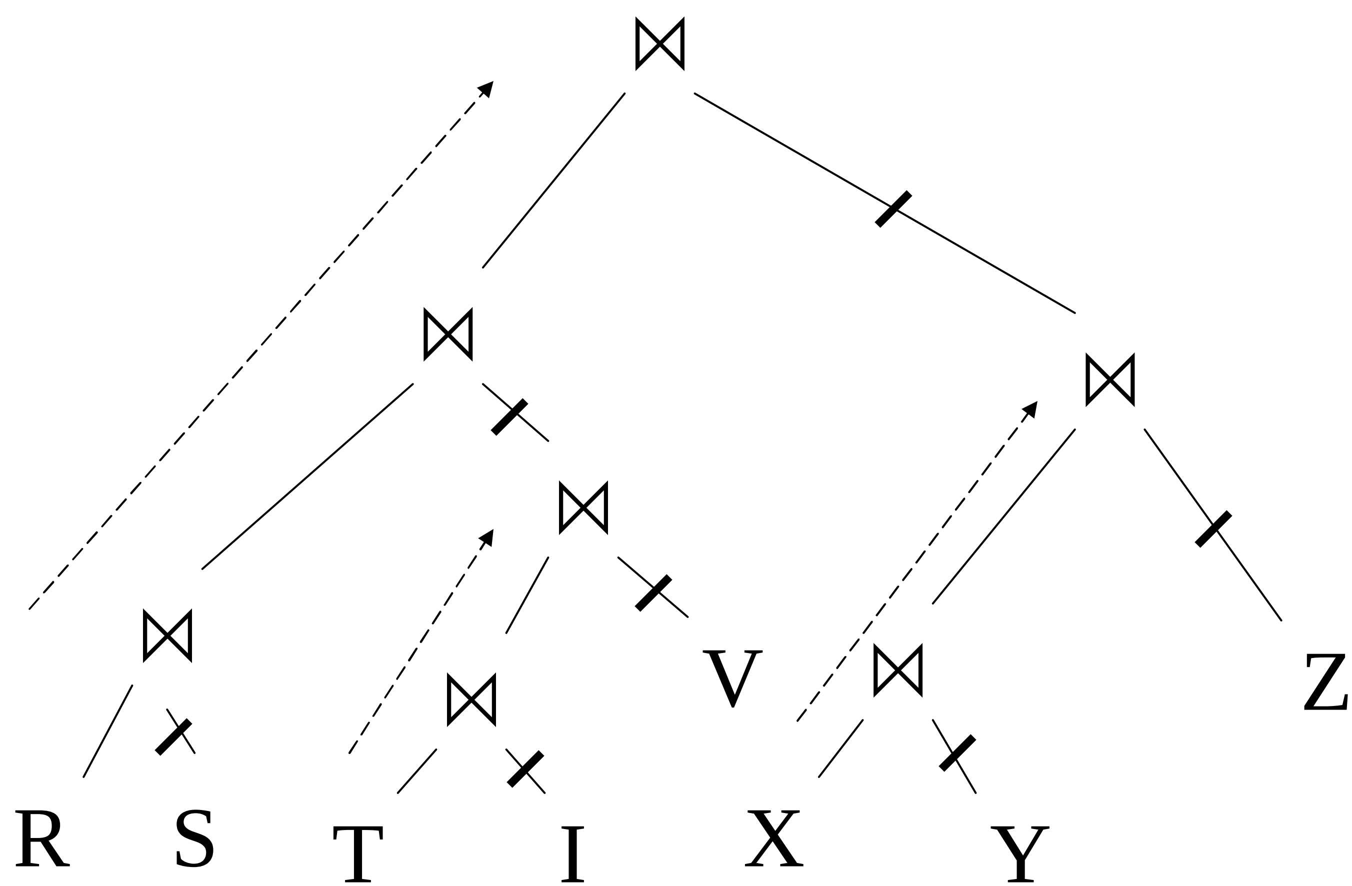
Pipeline Between Operators

Question in class

Given B(R), B(S), B(T), B(U)

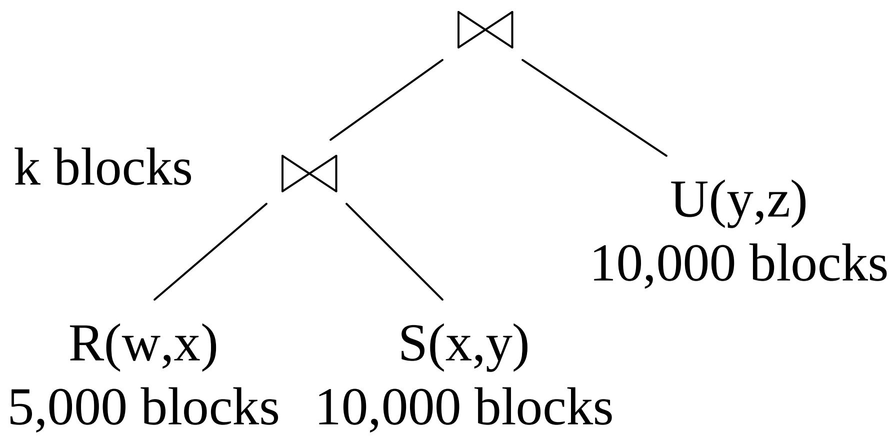
- What is the total cost of the plan ?
  - Cost =
- How much main memory do we need ?
  - M =

Pipeline in Bushy Trees



Example

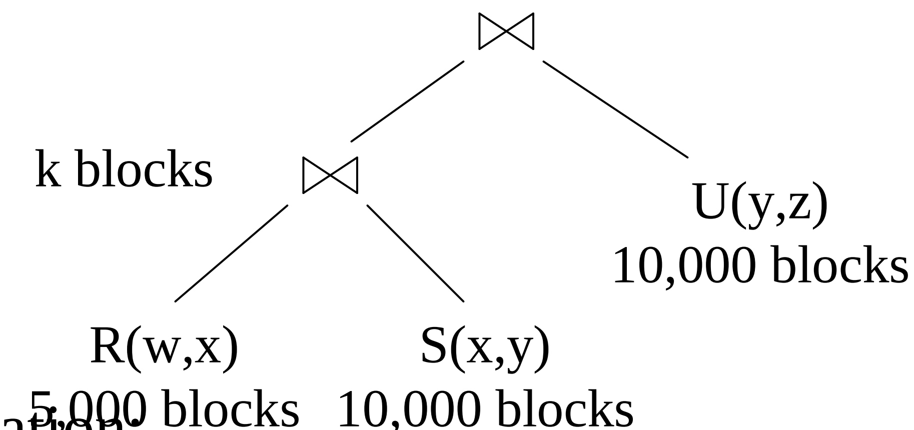
- Logical plan is:



- Main memory M = 101 buffers

Example

M = 101

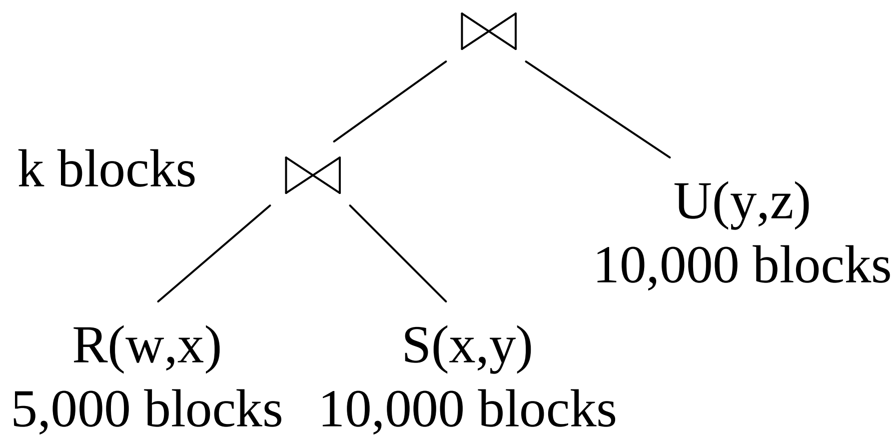


Naïve evaluation:

- 2 partitioned hash-joins
- Cost 3B(R) + 3B(S) + 4k + 3B(U) = 75000 + 4k

Example

M = 101

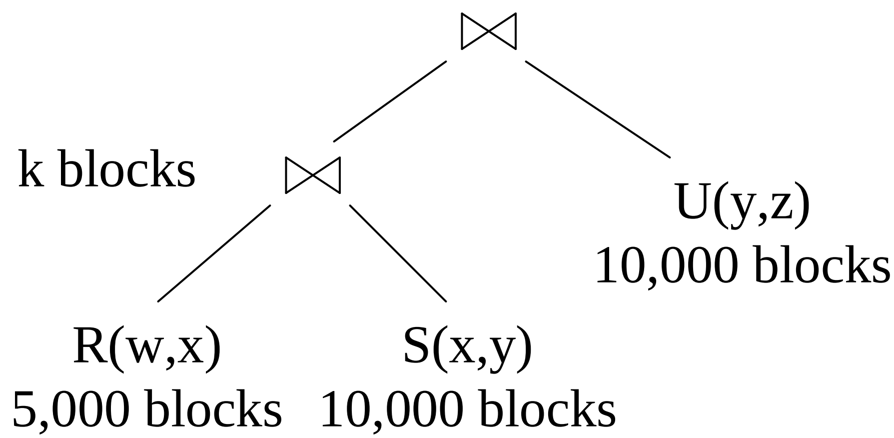


Smarter:

- Step 1: hash R on x into 100 buckets, each of 50 blocks; to disk
- Step 2: hash S on x into 100 buckets; to disk
- Step 3: read each R_i in memory (50 buffer) join with S_i (1 buffer); hash result on y into 50 buckets (50 buffers)   -- here we *pipeline*
- Cost so far: 3B(R) + 3B(S)

Example

M = 101

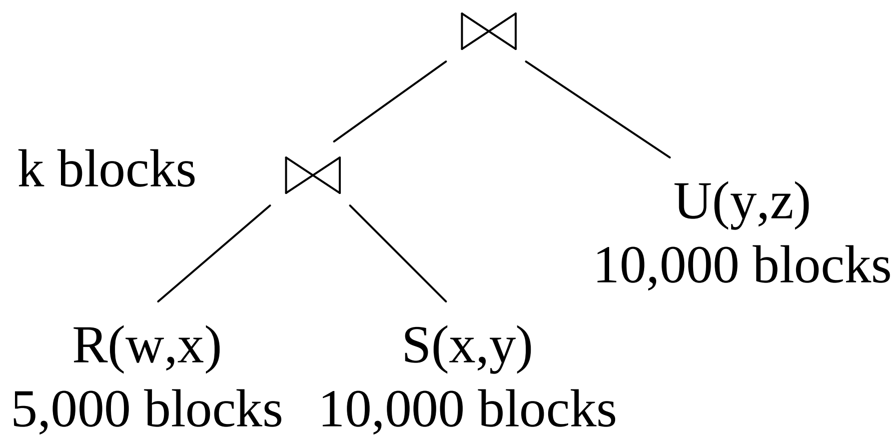


Continuing:

- How large are the 50 buckets on y ? Answer: k/50.
- If k <= 50 then keep all 50 buckets in Step 3 in memory, then:
- Step 4: read U from disk, hash on y and join with memory
- Total cost: 3B(R) + 3B(S) + B(U) = 55,000

Example

M = 101

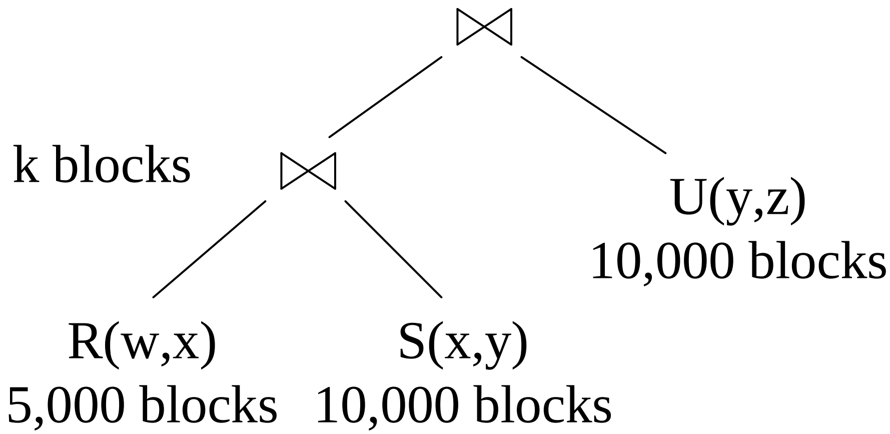


Continuing:

- If 50 < k <= 5000 then send the 50 buckets in Step 3 to disk
  - Each bucket has size k/50 <= 100
- Step 4: partition U into 50 buckets
- Step 5: read each partition and join in memory
- Total cost: 3B(R) + 3B(S) + 2k + 3B(U) = 75,000 + 2k

Example

M = 101



Continuing:

- If k > 5000 then materialize instead of pipeline
- 2 partitioned hash-joins
- Cost 3B(R) + 3B(S) + 4k + 3B(U) = 75000 + 4k

Example

Summary:

- If k <= 50, cost = 55,000
- If 50 < k <=5000, cost = 75,000 + 2k
- If k > 5000, cost = 75,000 + 4k

Size Estimation

The problem: Given an expression E, compute T(E) and V(E, A)

- This is hard without computing E
- Will 'estimate' them instead

Size Estimation

Estimating the size of a projection

- Easy: $T(\Pi_L(R)) = T(R)$
- This is because a projection doesn't eliminate duplicates

Size Estimation

Estimating the size of a selection

- $S = \sigma_{A=c}(R)$
  - T(S) san be anything from 0 to T(R) – V(R,A) + 1
  - Estimate: T(S) = T(R)/V(R,A)
  - When V(R,A) is not available, estimate T(S) = T(R)/10

- $S = \sigma_{A<c}(R)$
  - T(S) can be anything from 0 to T(R)
  - Estimate: T(S) = (c - Low(R, A))/(High(R,A) - Low(R,A))T(R)
  - When Low, High unavailable, estimate T(S) = T(R)/3

Size Estimation

Estimating the size of a natural join, $R \,|\times|_A\, S$

- When the set of A values are disjoint, then $T(R \,|\times|_A\, S) = 0$
- When A is a key in S and a foreign key in R, then $T(R \,|\times|_A\, S) = T(R)$
- When A has a unique value, the same in R and S, then $T(R \,|\times|_A\, S) = T(R)\, T(S)$

Size Estimation

Assumptions:

- *Containment of values*: if V(R,A) <= V(S,A), then the set of A values of R is included in the set of A values of S
  - Note: this indeed holds when A is a foreign key in R, and a key in S
- *Preservation of values*: for any other attribute B, $V(R \mid\times\mid_A S, B) = V(R, B)$ (or $V(S, B)$)

Size Estimation

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

- Then each tuple t in R joins *some* tuple(s) in S
  - How many ?
  - On average T(S)/V(S,A)
  - t will contribute T(S)/V(S,A) tuples in $R \,|\times|_A\, S$
- Hence $T(R \,|\times|_A\, S) = T(R)\,T(S) / V(S,A)$

In general: $T(R \,|\times|_A\, S) = T(R)\,T(S) / \max(V(R,A),V(S,A))$

Size Estimation

Example:

- $T(R) = 10000$, $T(S) = 20000$
- $V(R,A) = 100$, $V(S,A) = 200$
- How large is $R \,|\times|_A\, S$?

Answer: $T(R \,|\times|_A\, S) = 10000 \; 20000/200 = 1M$

Size Estimation

Joins on more than one attribute:

- $T(R \mid\times\mid_{A,B} S) =$

  $T(R)\ T(S)/(\max(V(R,A),V(S,A))*\max(V(R,B),V(S,B)))$

Histograms

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

Histograms

Employee(ssn, name, salary, phone)

- Maintain a histogram on salary:

Salary:	0..20k	20k..40k	40k..60k	60k..80k	80k..100k	> 100k
Tuples	200	800	5000	12000	6500	500

- T(Employee) = 25000, but now we know the distribution

Histograms

Ranks(rankName, salary)

- Estimate the size of Employee $|\times|_{\text{Salary}}$ Ranks

Employee	0..20k	20k..40k	40k..60k	60k..80k	80k..100k	> 100k
	200	800	5000	12000	6500	500

Ranks	0..20k	20k..40k	40k..60k	60k..80k	80k..100k	> 100k
	8	20	40	80	100	2

Histograms

- Eqwidth

0..20	20..40	40..60	60..80	80..100
2	104	9739	152	3

- Eqdepth

0..44	44..48	48..50	50..56	55..100
2000	2000	2000	2000	2000