CSE544: Principles of Database Systems

Query Execution

Announcements

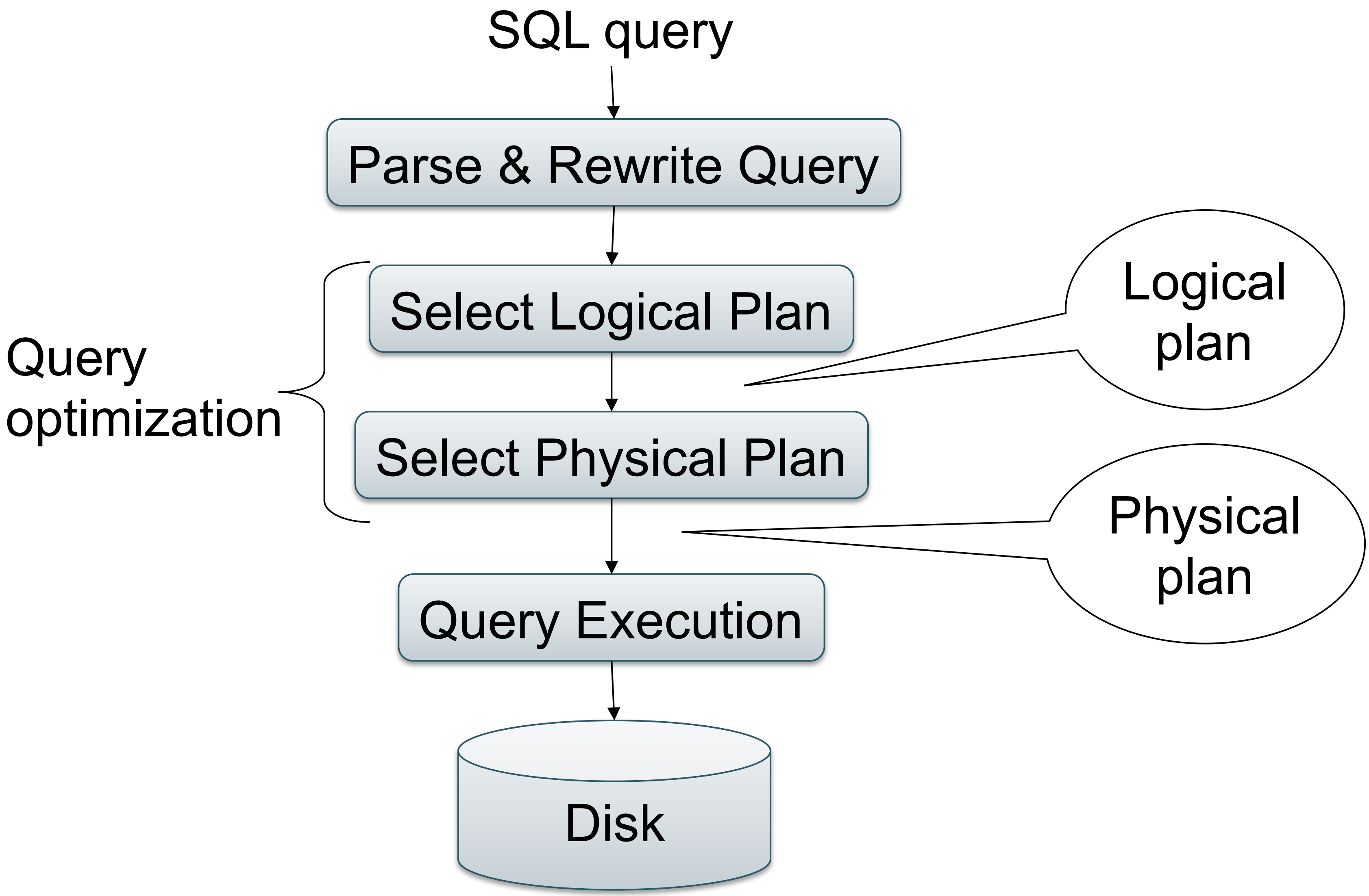
- Homework 2 is posted
  - Part A = SimpleDB (takes you a few days)
  - Part B = AWS, Hadoop (ditto)
  - Part C = a simple question (takes you 20')
  - Due on May 6 but start early!!
- Project M2 (Proposal) due April 22
  - Define clear, limited goals! Don't try too much
  - There is still time to switch

Outline

- Relational Algebra: Ch. 4.2
- Evaluating relational operators: Ch. 14 and Shapiro's paper

Relational Algebra

Steps of the Query Processor



SQL = WHAT

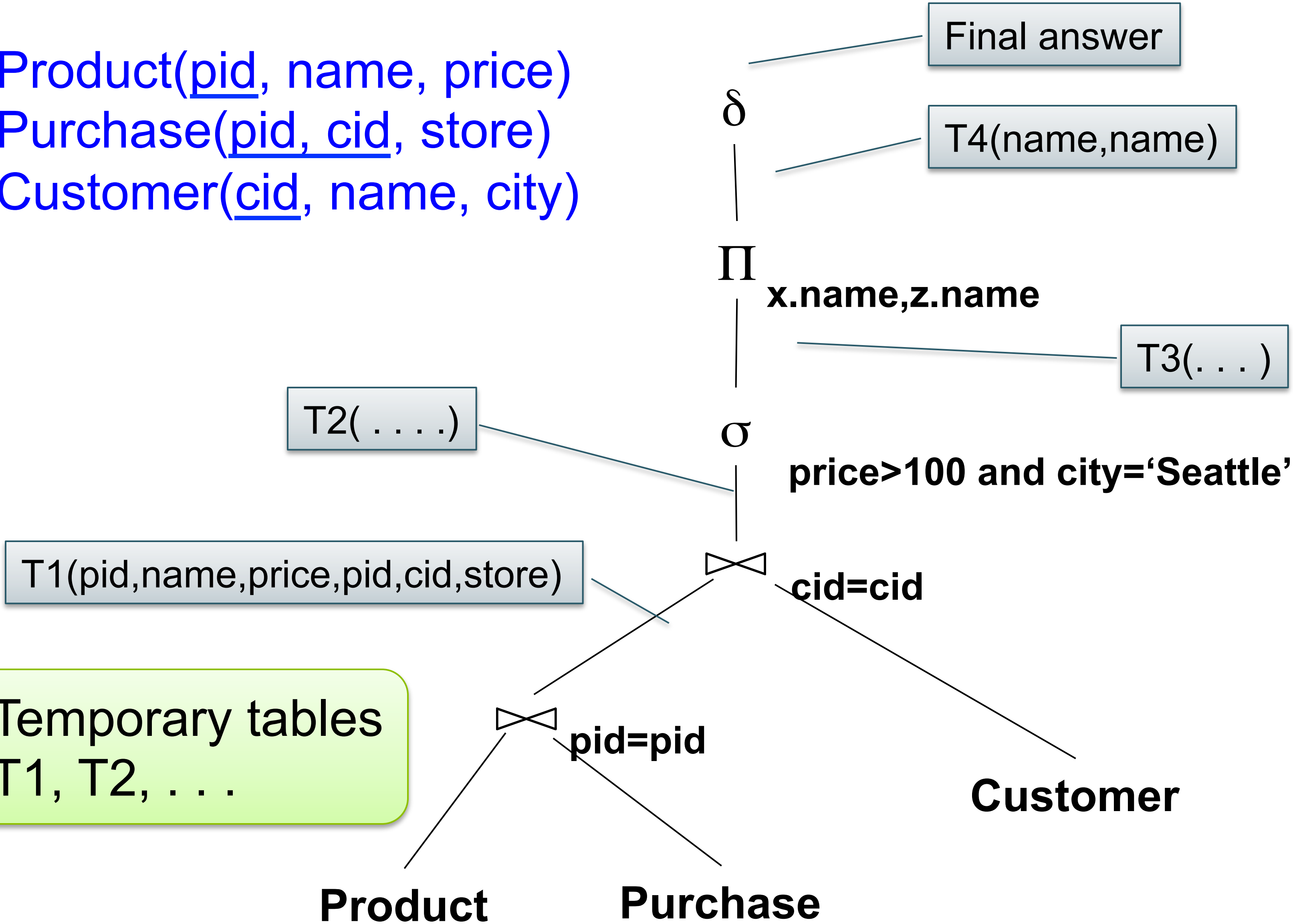
Product(pid, name, price)
Purchase(pid, cid, store)
Customer(cid, name, city)

```
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)



Temporary tables
T1, T2, . . .

Relational Algebra = HOW

The order is now clearly specified:

For each PRODUCT x
    Join with PURCHASE y
        Join with CUSTOMER z
            Select tuples with Price>100
                and City='Seattle'
Project on the columns x.name, z.name
Eliminate duplicates

Extended Algebra Operators

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

Relational Algebra

Extended Relational Algebra

Relational Algebra: Sets v.s. Bags Semantics

- 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

Union and Difference

$$R1 \cup R2$$
$$R1 - R2$$

What do they mean over bags ?

What about Intersection ?

- Derived operator using minus

$$\boxed{R1 \cap R2 = R1 - (R1 - R2)}$$

- Derived using join (will explain later)

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

What is the meaning under bag semantics?

Projection

- Eliminates columns

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

- Example:
  - $\Pi_{\text{SSN, Name}}$ (Employee)
  - Answer(SSN, Name)

Semantics differs over set or over bags

Employee

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

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

Name	Salary
John	20000
John	60000
John	20000

Bag semantics

Name	Salary
John	20000
John	60000

Set semantics

Which is more efficient?

Natural Join

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

- Meaning: $R1 \bowtie R2 = \Pi_A(\sigma(R1 \times R2))$
- Where:
  - σ checks equality of all common attributes
  - Π_A eliminates the duplicate attributes

Natural Join

R

A	B
X	Y
X	Z
Y	Z
Z	V

S

B	C
Z	U
V	W
Z	V

R ⋈ 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 schemas R(A, B, C, D), S(A, C, E), what is the schema of R ⋈ S ?

- Given R(A, B, C), S(D, E), what is R ⋈ S ?

- Given R(A, B), S(A, B), what is R ⋈ 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
  - Example band join: $R \bowtie_{R.A-5<S.B \,\wedge\, S.B<R.A+5} S$

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

Semijoin

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

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

$R \ltimes_C S$ returns tuples in R that join with some tuple in S

- Duplicates in R are preserved
- Duplicates in S don't matter

Note: the semijoin is an important notion; we will return to it

Operators on Bags

- Duplicate elimination $\delta(R)$ =

```
SELECT DISTINCT *
FROM R
```

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

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

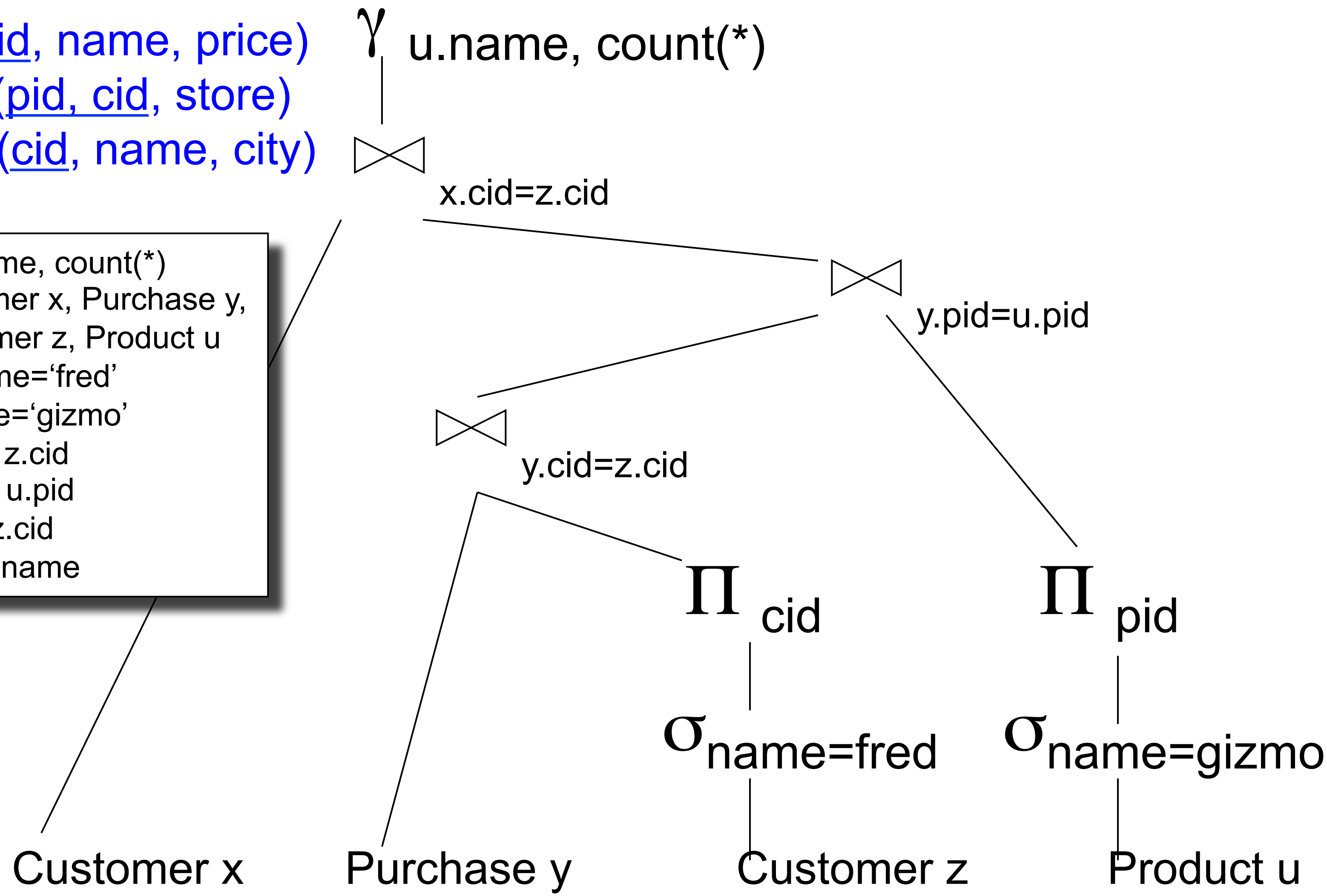
- Sorting $\tau_{A,B}$ (R)

```
SELECT  *
FROM R
ORDER BY A
```

Complex RA Expressions

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

```
SELECT u.name, count(*)
FROM Customer x, Purchase y,
         Customer z, Product u
WHERE z.name='fred'
    and u.name='gizmo'
    and y.cid = z.cid
    and y.pid = u.pid
    and x.cid=z.cid
GROUP BY u.name
```



Query Evaluation

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

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

Question in Class

Logical operator:

Product(pid,name,price) $\bowtie_{pid=pid}$ Purchase(pid,cid,store)

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

1.
2.
3.

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

Question in Class

Logical operator:

Product(pid,name,price) $\bowtie_{pid=pid}$ Purchase(pid,cid,store)

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

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

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

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

1. Nested Loop Join

```
for x in Product do {
   for  y in Purchase do {
        if (x.pid == y.pid) output(x,y);
   }
}
```

Product = *outer relation*
Purchase = *inner relation*
Note: sometimes terminology is switched

Would it be more efficient to choose Purchase=outer, Product=inner?
What if we had an index on Product.pid ?

Hash Tables

Separate chaining:

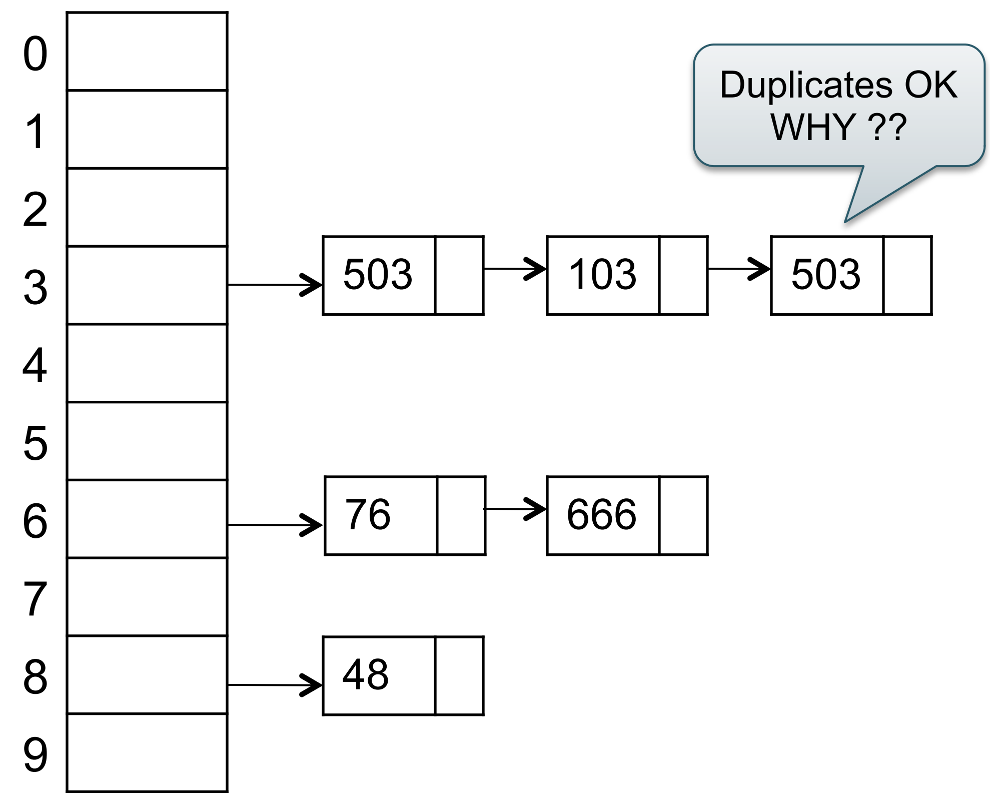
A (naïve) hash function:

$h(x) = x \bmod 10$

Operations on a hash table:

find(103) = ??

insert(488) = ??



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

Product(pid,name,price) $\bowtie_{pid=pid}$ Purchase(pid,cid,store)

2. "Classic Hash Join"

Build phase

```
for y in Purchase do  insert(y);

for x in Product do {
   ylist = find(x.pid);
   for y in ylist do { output(x,y); }
}
```

Probe phase

Product = *outer relation*
Purchase = *inner relation*

Better: make Product=inner, Purchse=outer (why?)

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

Product(pid,name,price) $\bowtie_{pid=pid}$ Purchase(pid,cid,store)

3. Merge Join (main memory)

```
Product1= sort(Product, pid);
Purchase1= sort(Purchase, pid);
x=Product1.get_next(); y=Purchase1.get_next();

While (x!=NULL and y!=NULL) {
    case:
      x.pid < y.pid:    x = Product1.get_next( );
      x.pid > y.pid:    y = Purchase1.get_next();
      x.pid == y.pid { output(x,y);
                         y = Purchase1.get_next();
                       }
}
```

Why ???

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 (used both in the book and by Shapiro):

- 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

- The operator *reads* the data from disk
- 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)

The Iterator Model

Each operator implements this interface

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

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

Product(pid,name,price) $\bowtie_{pid=pid}$ Purchase(pid,cid,store)

Main Memory Nested Loop Join

```
open( ) {
  Product.open( );
  Purchase.open( );
  x = Product.get_next( );
}
```

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

```
get_next( ) {
  repeat {
    y= Purchase.get_next( );
    if (y== NULL)
      { Purchase.close();
        x= Product.get_next( );
        if (x== NULL) return NULL;
        Purchase.open( );
        y= Purchase.get_next( );
      }
  until (x.cid == y.cid);
  return (x,y)
}
```

ALL operators need to be implemented this way !

Join Algorithms

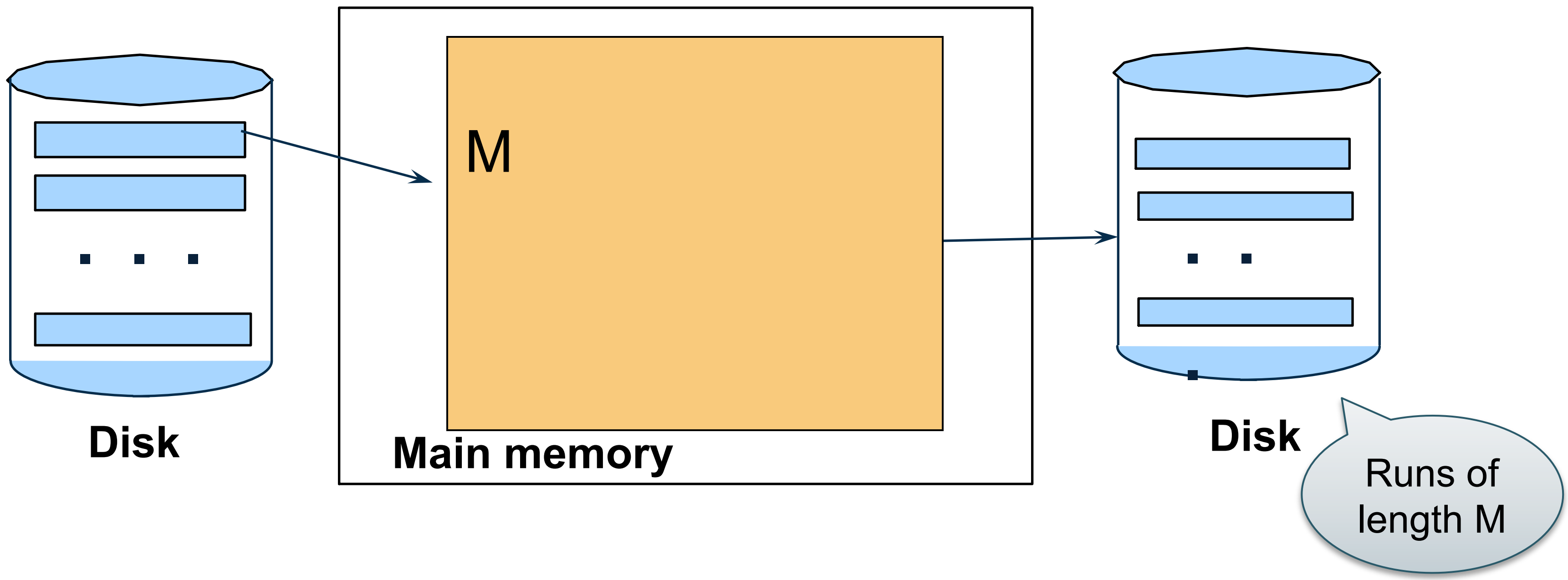
- Nested Loop Joins – have seen already
- Merge Join
- Hash join (and variations)

External Sorting

- Problem: sort a file R of size B(R) with memory M
- Will discuss only 2-pass sorting, when $B \leq M^2$

External Merge-Sort: Step 1

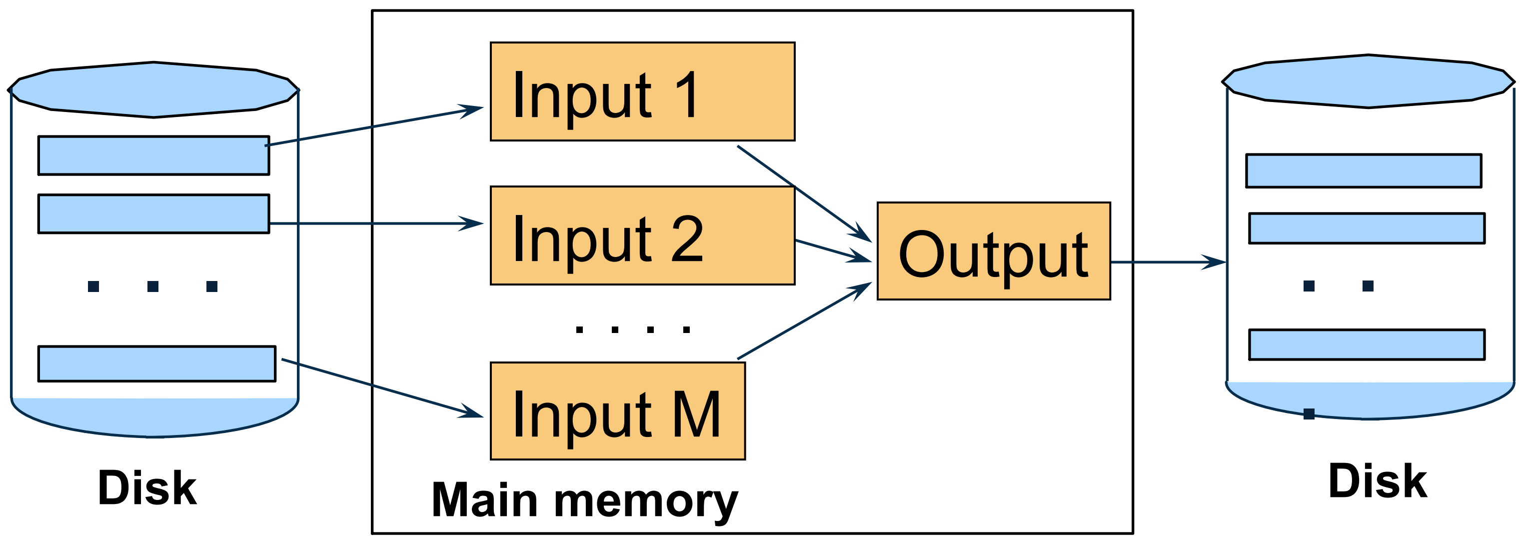
- Phase one: load M bytes in memory, sort



Can increase to length 2M using "replacement selection" (How?)

External Merge-Sort: Step 2

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



If B > M^2,
why not merge
more than M runs
in one step?

Assuming B ≤ M^2 then we are done

Cost of External Merge Sort

- Read+write+read = 3B(R)
  (we don't count the final write)

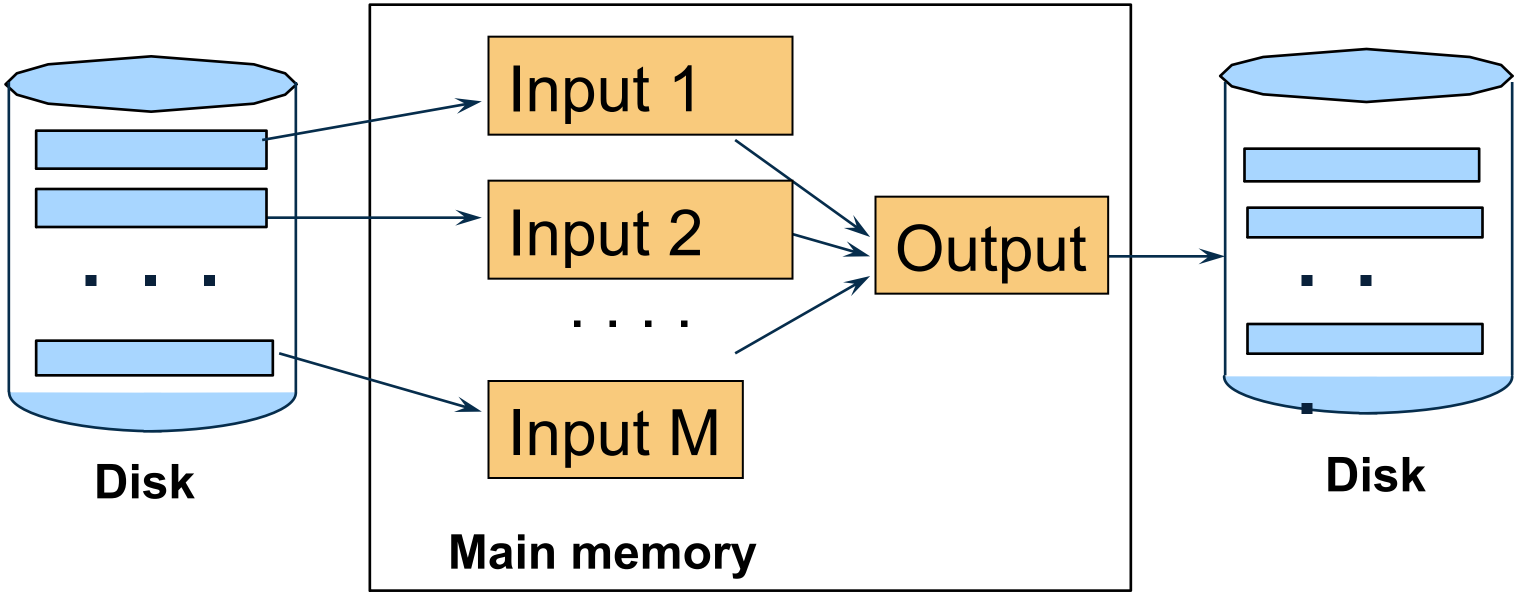
- Assumption: $B(R) <= M^2$

Application: Merge-Join

Join R ⋈ 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 $M_1 + M_2 \leq M$, or $B(R) + B(S) \leq M^2$

Partitioned Hash Join, or GRACE Join

$R \bowtie S$

How does it work?

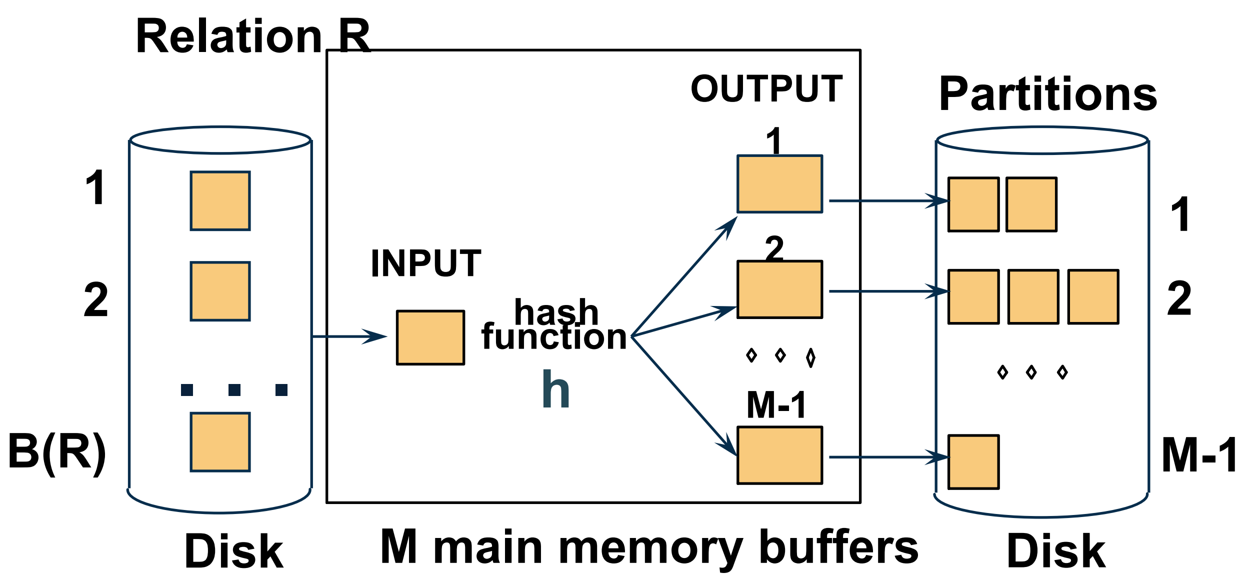
Partitioned Hash Join, or GRACE Join

R ⋈ 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

The Idea of Hash-Based Partitioning

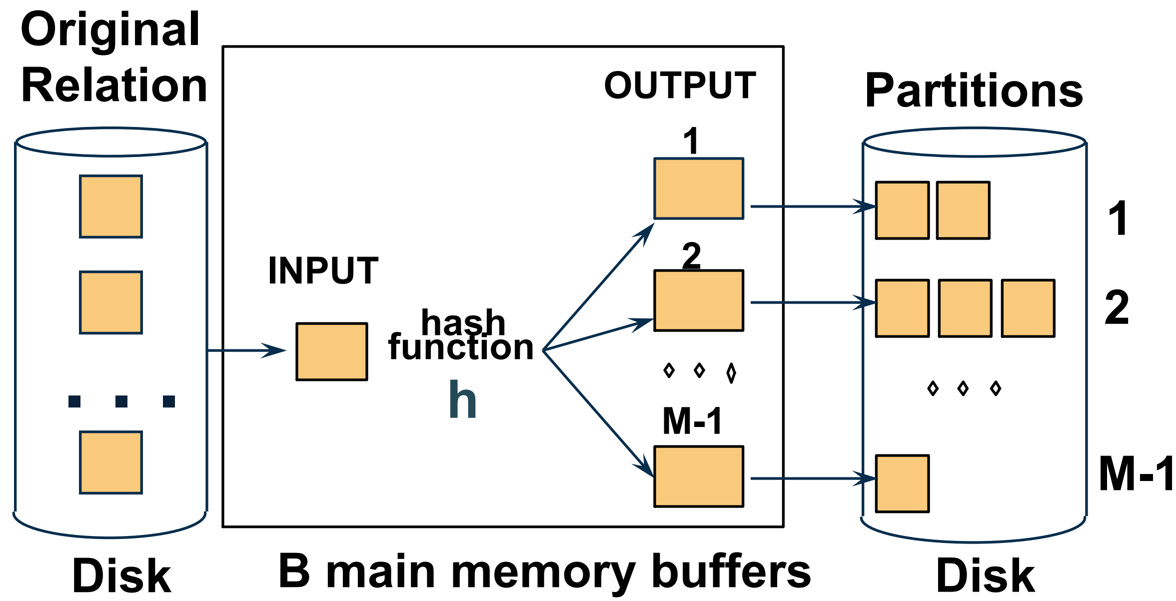
- Idea: partition a relation R into M-1 buckets, on disk
- Each bucket has size approx. B(R)/(M-1) ≈ B(R)/M



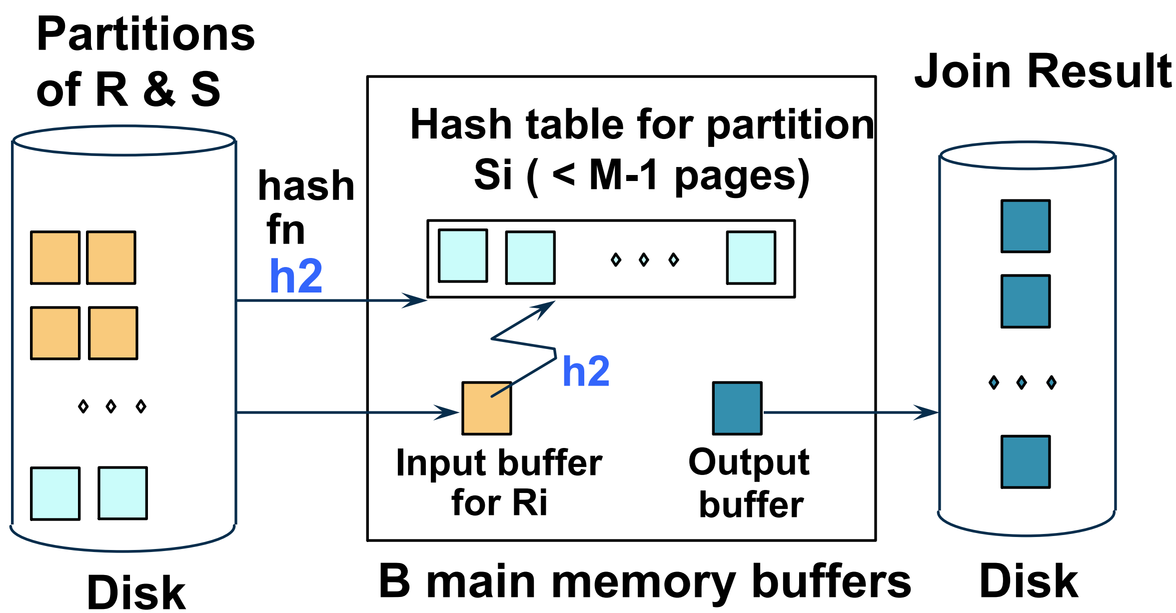
Assumption: B(R)/M <= M, i.e. $B(R) <= M^2$

Grace-Join

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

- What problem does it address?

Hybrid Hash Join

- What problem does it address?

- If $B(R) \leq M$ then we can use main memory hash-join: cost = $B(R) + B(S)$

- If $B(R) >\approx M$ then we must use Grace join: cost jumps to $3*B(R) + 3*B(S)$

Hybrid Hash Join

- How does it work?

Hybrid Hash Join

- How does it work?
- Use B(R)/M buckets
- Since B(R)/M << M, there is enough space left in main memory: use it to store a few buckets
- Fuzzy math to make this work, but best done adaptively:
  - Start by keeping *all* buckets in main memory
  - When the remaining memory (M - B(R)/M) fills up, spill one bucket to disk