# CSE544: Principles of Database Systems

Query Execution

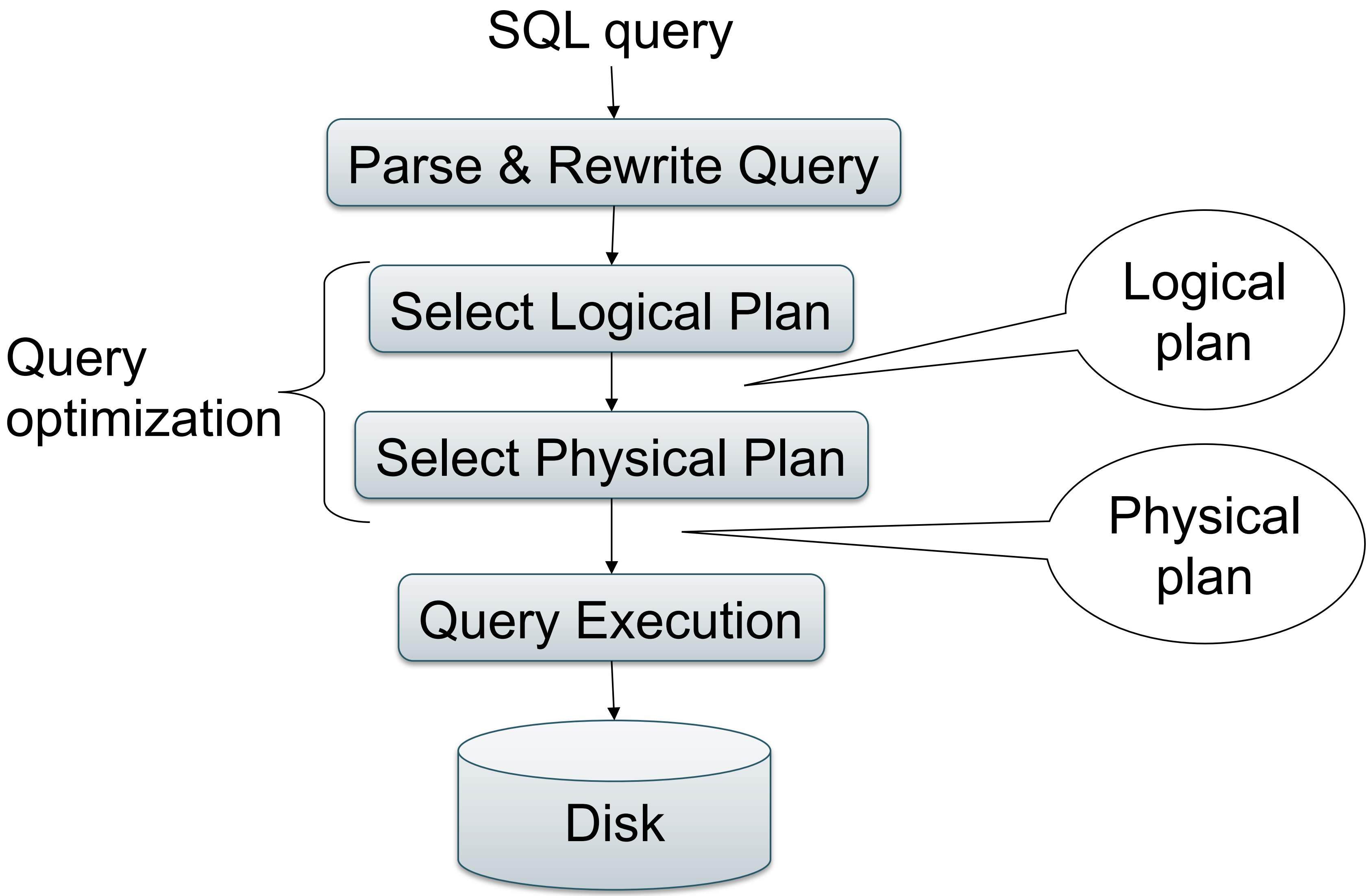
# Announcements

- Homework 2 is posted, due May 6
  – SimpleDB
  – Understand existing code PLUS write more code
  – Start early!!
- Review 3 (Selectivity estimation): due April 24
- Project meetings: tomorrow, April 24
- Project M2 (Proposal) due April 26
  – Please try to choose your project by Wednesday
  – Proposal: define clear, limited goals! Don’t try too much

# Outline

- Relational Algebra: Ch. 4.2
- Query Evaluation: Ch. 12-14

# Steps of the Query Processor



# SQL = WHAT

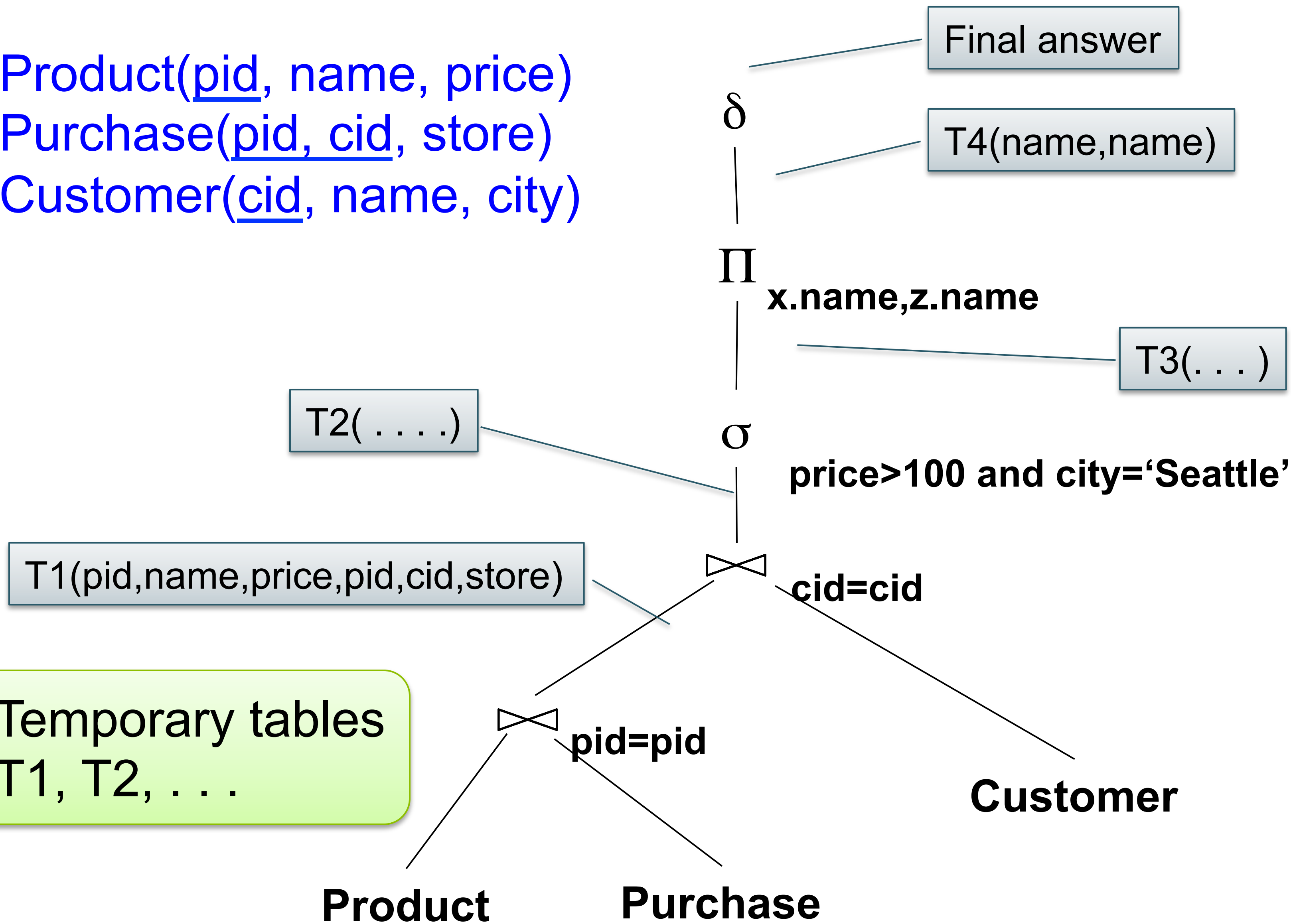
Product(pid, name, price)
Purchase(pid, cid, store)
Customer(cid, name, city)

```
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, . . .

# Extended Algebra Operators

- Union $\cup$,
- Difference $-$
- Selection $\sigma$
- Projection $\Pi$
- Join $\bowtie$
- Rename $\rho$
- Duplicate elimination $\delta$
- Grouping and aggregation $\gamma$
- Sorting $\tau$

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 of ∩ 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:
  - $\sigma$ checks equality of all common attributes
  - $\Pi_A$ 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

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

- Here $\theta$ 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 $\theta$ 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

Semijoin is ***important***; we will return to it

# Anti-Semi-Join

- Notation: R $\triangleright$ 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=

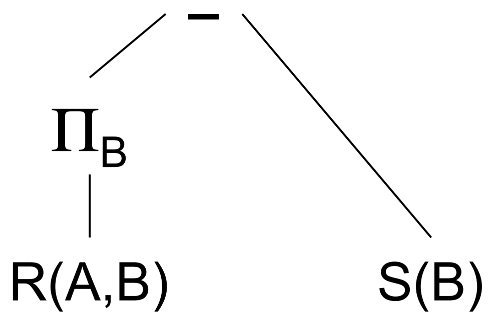
```
SELECT DISTINCT *
FROM R
WHERE not exists  (SELECT *
                   FROM S
                   WHERE R.B=S.B)
```

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=



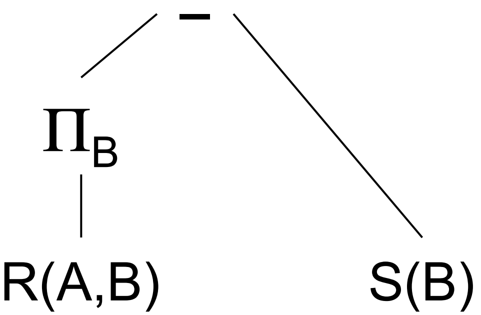

```
SELECT DISTINCT *
FROM R
WHERE not exists  (SELECT *
                   FROM S
                   WHERE R.B=S.B)
```

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=

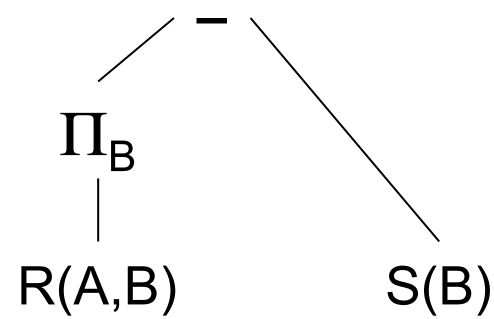
```
SELECT DISTINCT *
FROM R
WHERE not exists  (SELECT *
                   FROM S
                   WHERE R.B=S.B)
```

R(A,B)
S(B)

# Set Difference v.s. Anti-semijoin

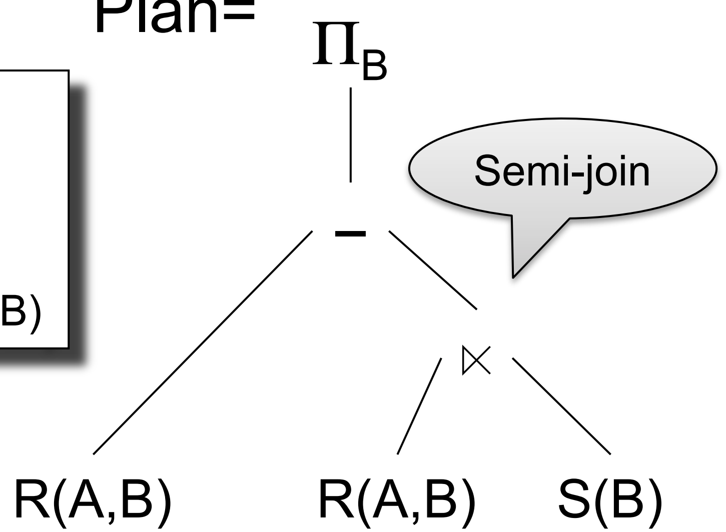
```
SELECT DISTINCT R.B
FROM R
WHERE not exists  (SELECT *
                   FROM S
                   WHERE R.B=S.B)
```

Plan=




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

Plan=

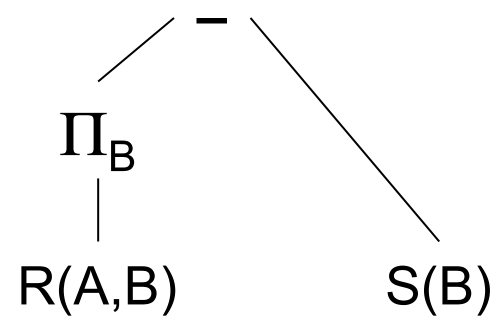


R(A,B)
S(B)

# Set Difference v.s. Anti-semijoin

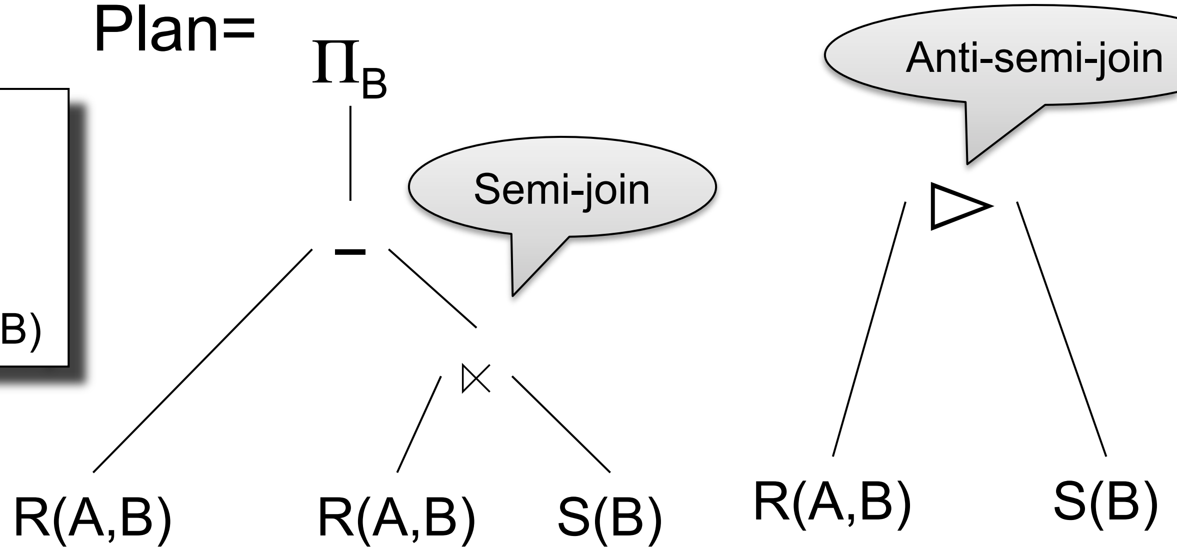
```
SELECT DISTINCT R.B
FROM R
WHERE not exists  (SELECT *
                   FROM S
                   WHERE R.B=S.B)
```

Plan=




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

Plan=



# Operators on Bags

- Duplicate elimination $\delta(R)$ =

```
SELECT DISTINCT *
FROM R
```

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

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

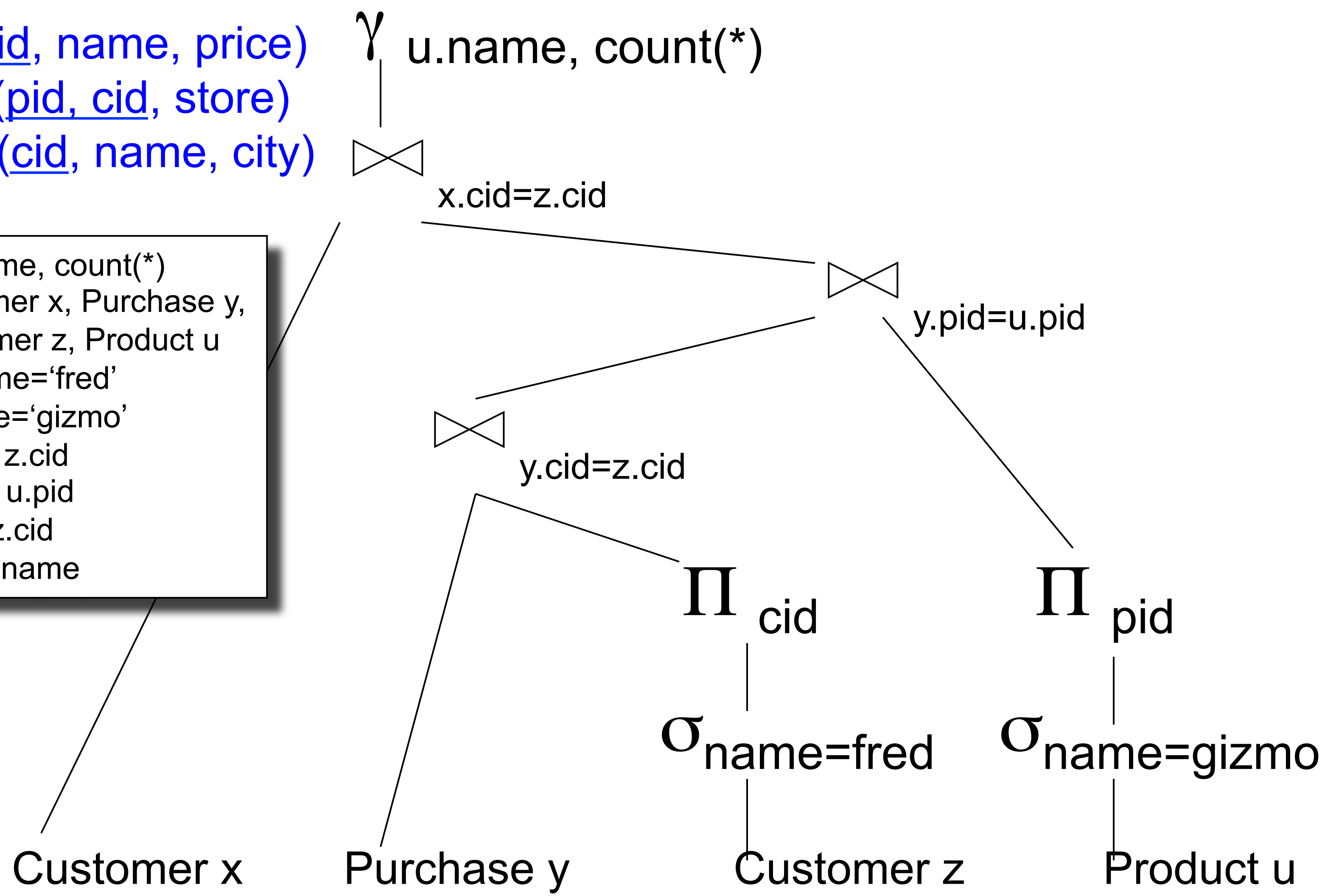
- Sorting $\tau_{A,B}$ (R)

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

# Complex RA Expressions

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

```sql
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

# Question in Class

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

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

1.
2.
3.

# Question in Class

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

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

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

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

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

# 1. Nested Loop Join

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

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

Discuss the possible use of an index Cusomer(cid)

# Hash Tables

A (naïve) hash function:

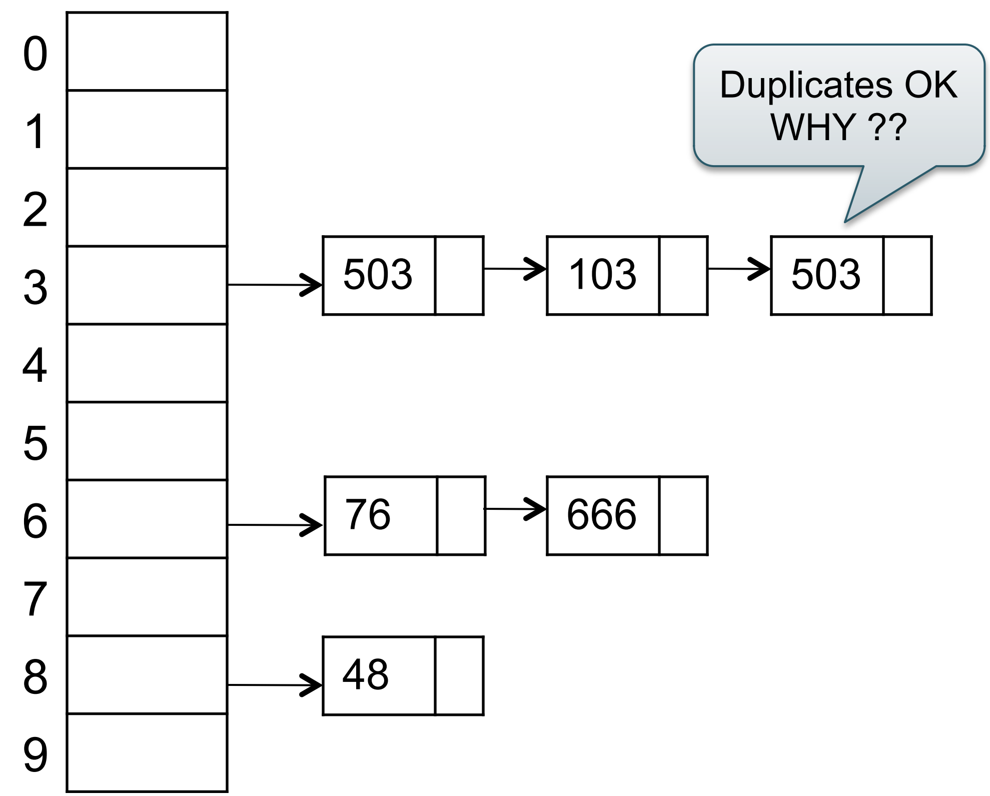
$$h(x) = x \bmod 10$$

Operations on a hash table:

find(103) = ??

insert(488) = ??

Separate chaining:



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

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

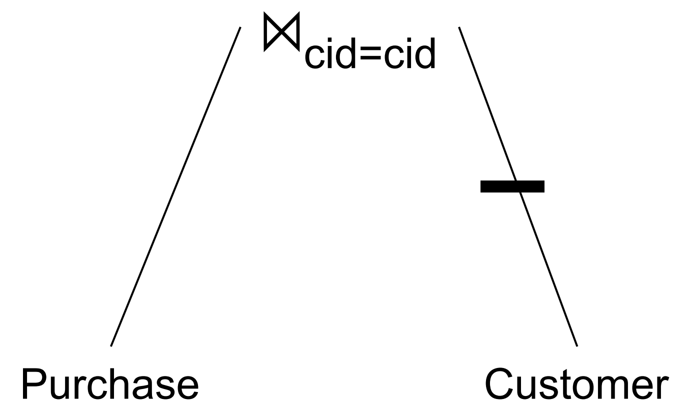
# 2. "Classic Hash Join"

Build phase

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

for x in Purchase do {
   y = find(x.cid);
    if (y != NULL)  { output(x,y); }
}
```

Probe phase




Purchase = *outer relation*
Customer = *inner relation*

What changes if the join attribute is not a key in the inner relation?

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

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

# 3. Merge Join (main memory)

```
Purchase1= sort(Purchase, cid);
Customer1=sort(Customer, cid)
x=Purchase1.get_next(); y=Customer1.get_next();

While (x!=NULL and y!=NULL) {
    case:
      x.cid < y.cid:    x = Purchase1.get_next( );
      x.cid > y.cid:    y = Customer1.get_next();
      x.cid == y.cid { output(x,y);
                         y = Purchase1.get_next();
                       }
}
```

Why ???

# The Iterator Model

Each operator implements this interface

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

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

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

# Main Memory Nested Loop Join

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

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

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

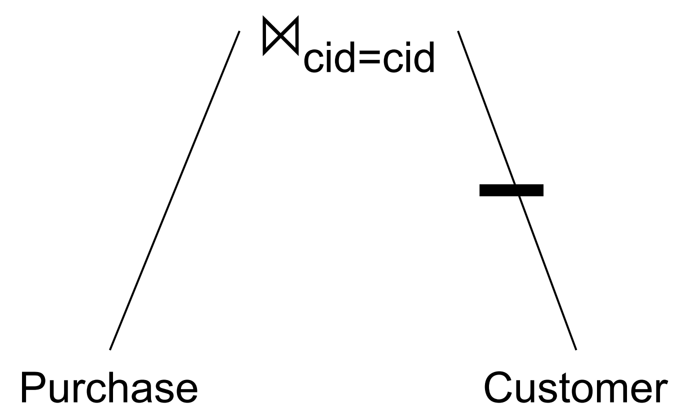
ALL operators need to be implemented this way !

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

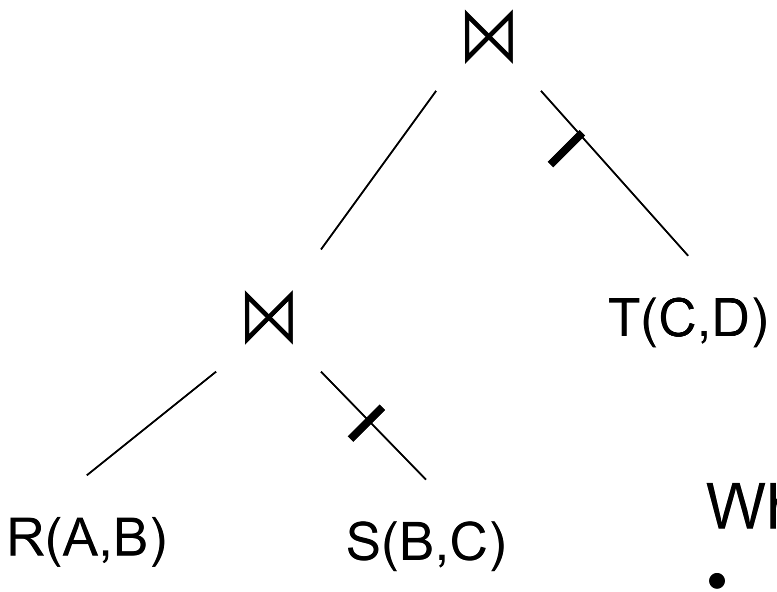
# Classic Hash Join

What do these operators do for the classic Hash Join?

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



# Discussion in class




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

What happens:

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

# 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)

# External Memory Join Algorithms

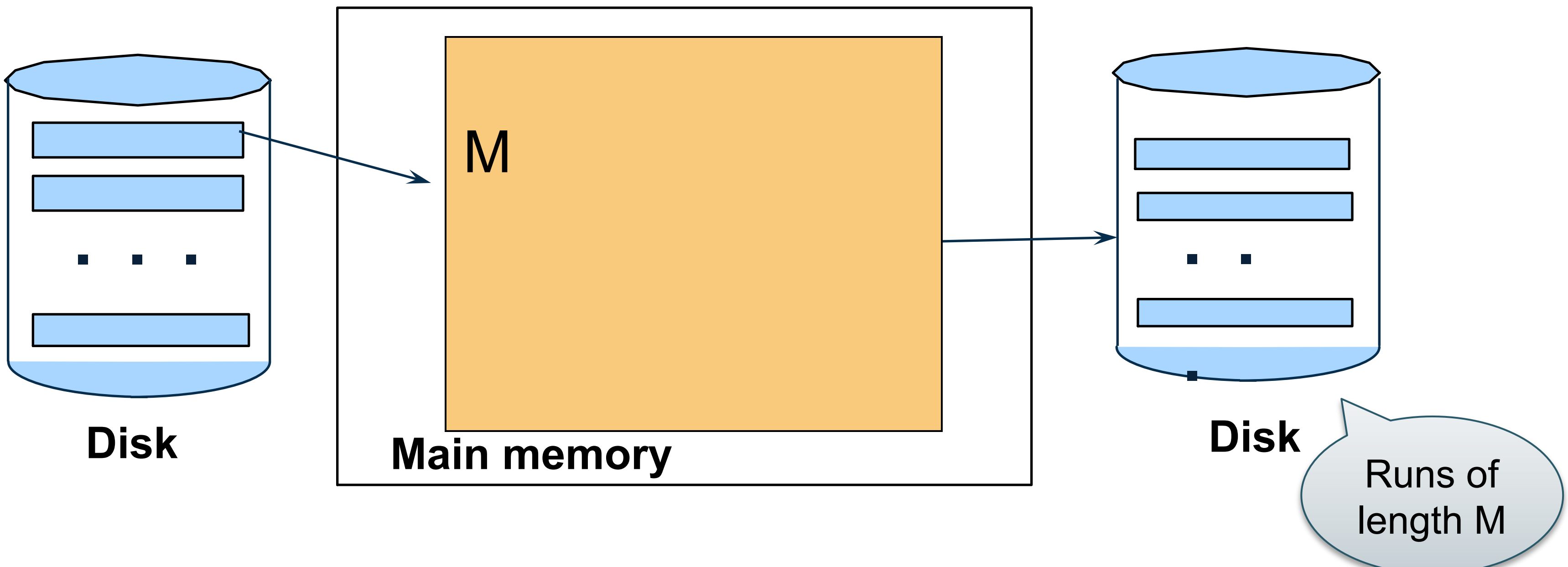
- Nested Loop Joins
- Merge Join
- Hash join: read paper, discuss next week

# External Sorting

- Problem: sort a file R of size B(R) with memory M
- Concrete:
  - Size of R is 100TB = $10^{14}$
  - Size of M is 1GB = $10^{9}$
  - Page size is 10KB = $10^{4}$

# External Merge-Sort: Step 1

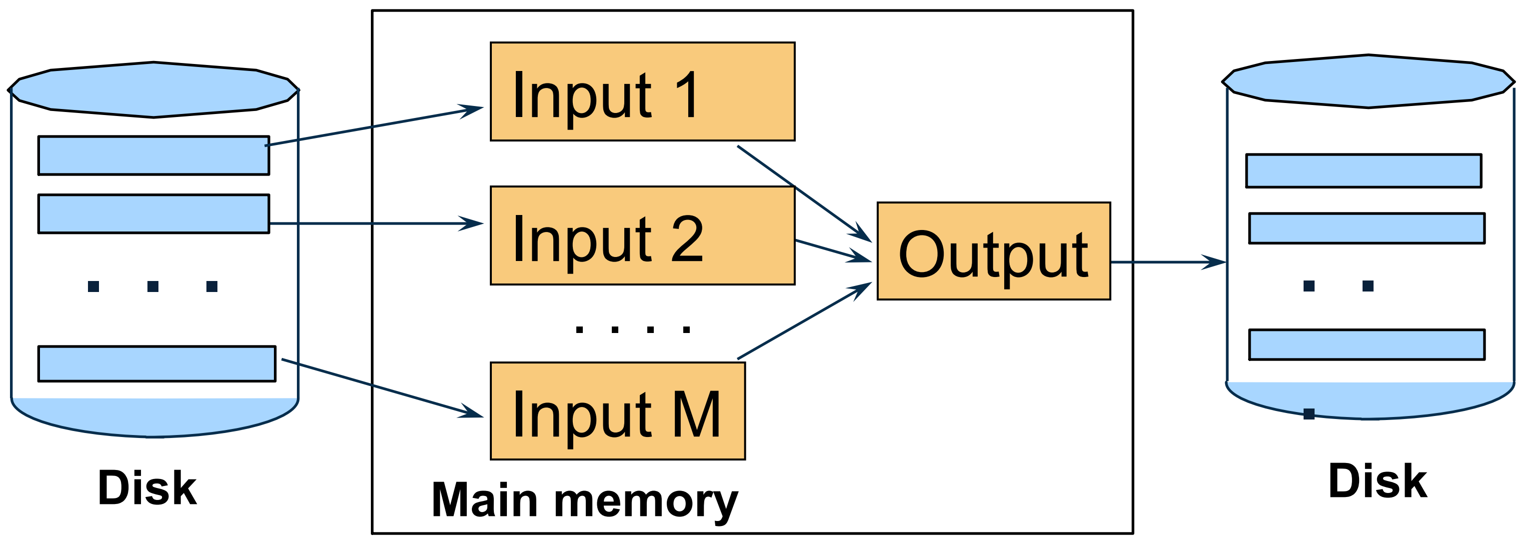
- Phase one: load M bytes in memory, sort




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) $\approx$ M$^2$




Assuming B ≤ M$^2$ then we are done

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

# Cost of External Merge Sort

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

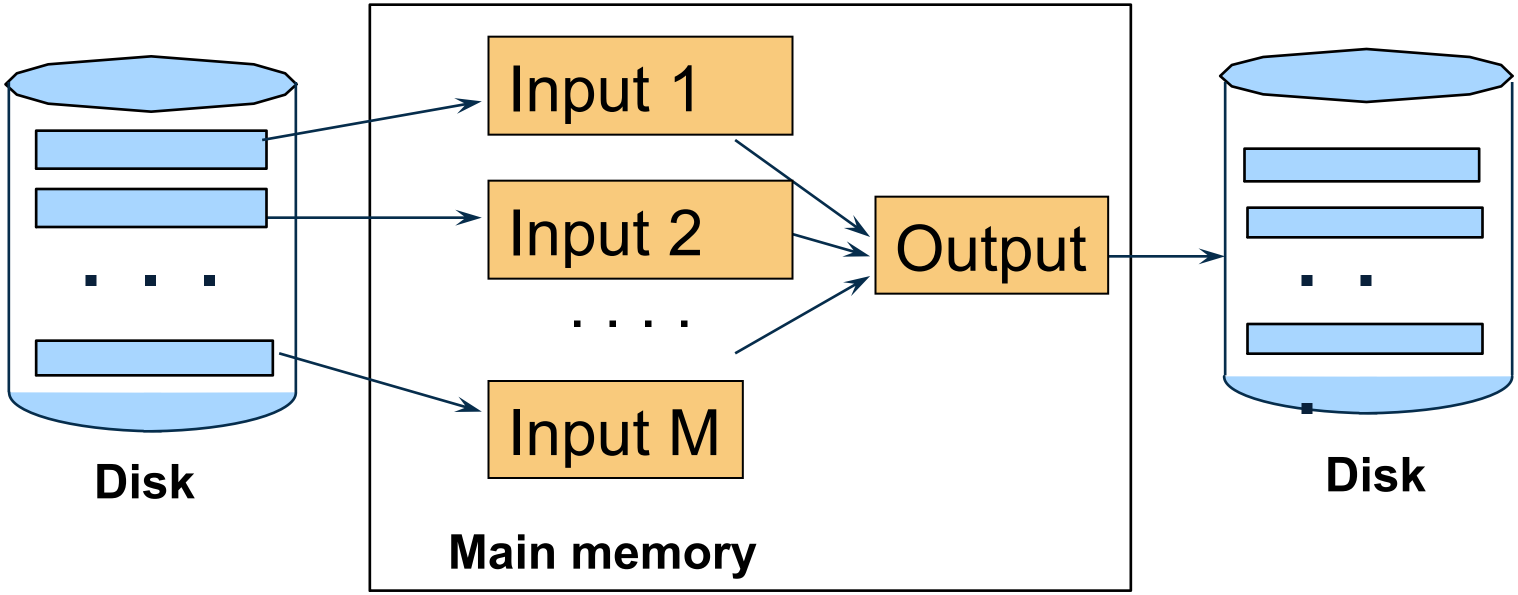
- Assumption: B(R) <= $M^2$

# Application: Merge-Join

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