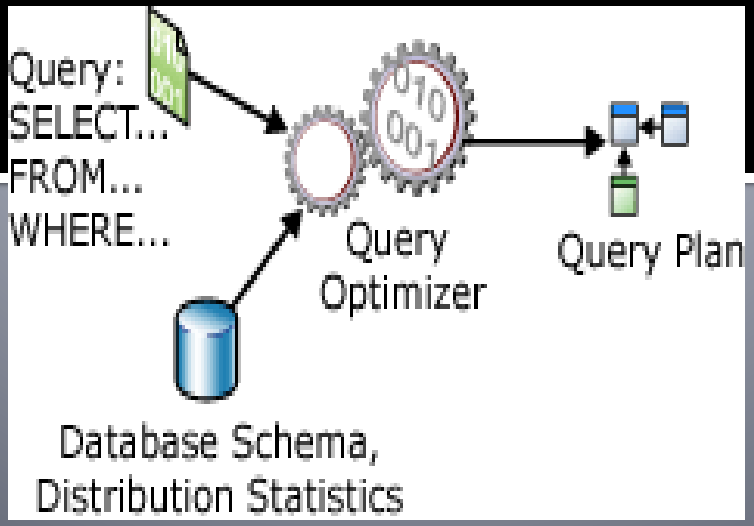
Query Processing and Optimization

CMPT 606

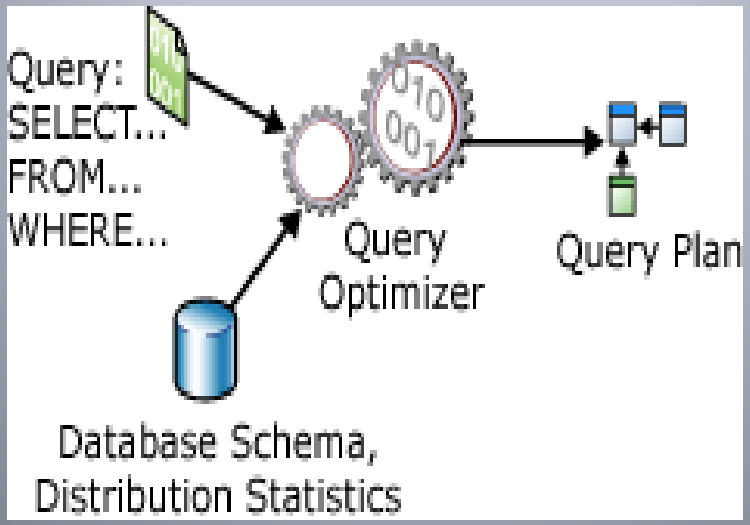


Read ch 6, 18 and 19

Outline

- Query Processing Overview
- Logical Query Plans
- Query Rewrite Rules
- Physical Query Plan
- Implementing Selection $\sigma_{(attr\ \text{op}\ value)}$
- Sort-Merge Algorithm
- Join Operator Implementation

Query Processing Overview



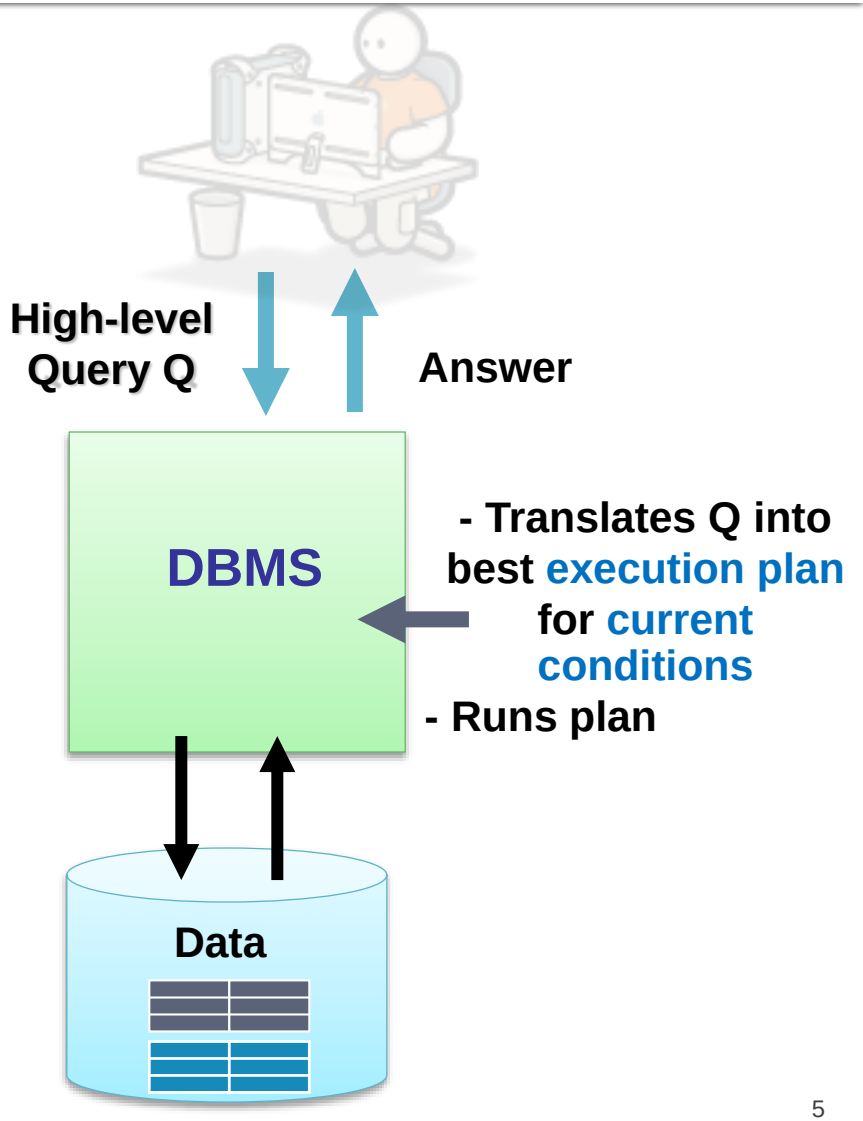
Why Study Query Processing and Optimization?

- Become familiar with the internals of the Query Optimizer and the execution plans it produces
  - Need to know Query Operator algorithms to understand query plans
  - Need to understand query plans to tune a DBMS
  - Take your query writing and performance diagnostic skills to the next level

=> Better placed to write queries that optimize well, and to debug plan-related performance problems

Query Processing

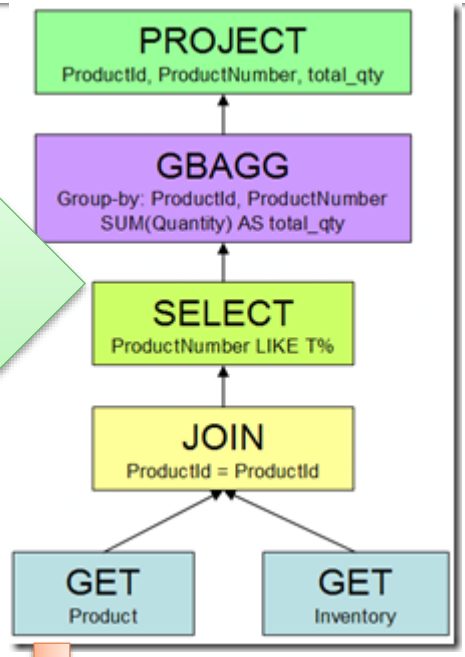
- **Query Processing**: Translate high level queries (e.g. SQL query) to low level operations (or algorithms) on the data files to retrieve the required data
  - SQL is a declarative language. It provides a way for the query writer to ***logically*** **describe the results required**
  - The Query Optimizer (QO) compiles and optimizes the SQL query to create an efficient **physical plan** that the Query Processor (QP) can execute

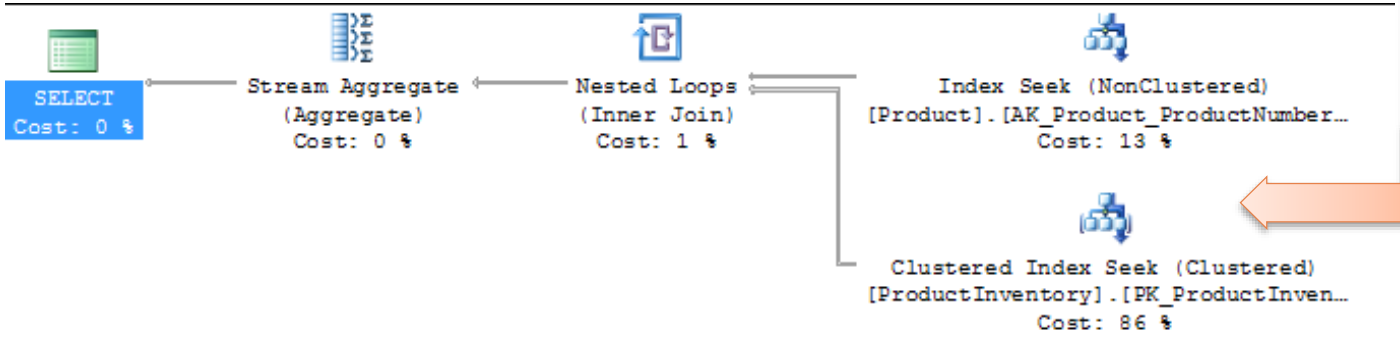


Query Processing Example

```
SELECT  P.ProductNumber, P.ProductID,
    SUM(I.Quantity) TotalQty
FROM  Production.Product P
      JOIN  Production.ProductInventory I
        ON  I.ProductID = P.ProductID
       WHERE  P.ProductNumber LIKE 'T%'
       GROUP BY P.ProductID, P.ProductNumber
```

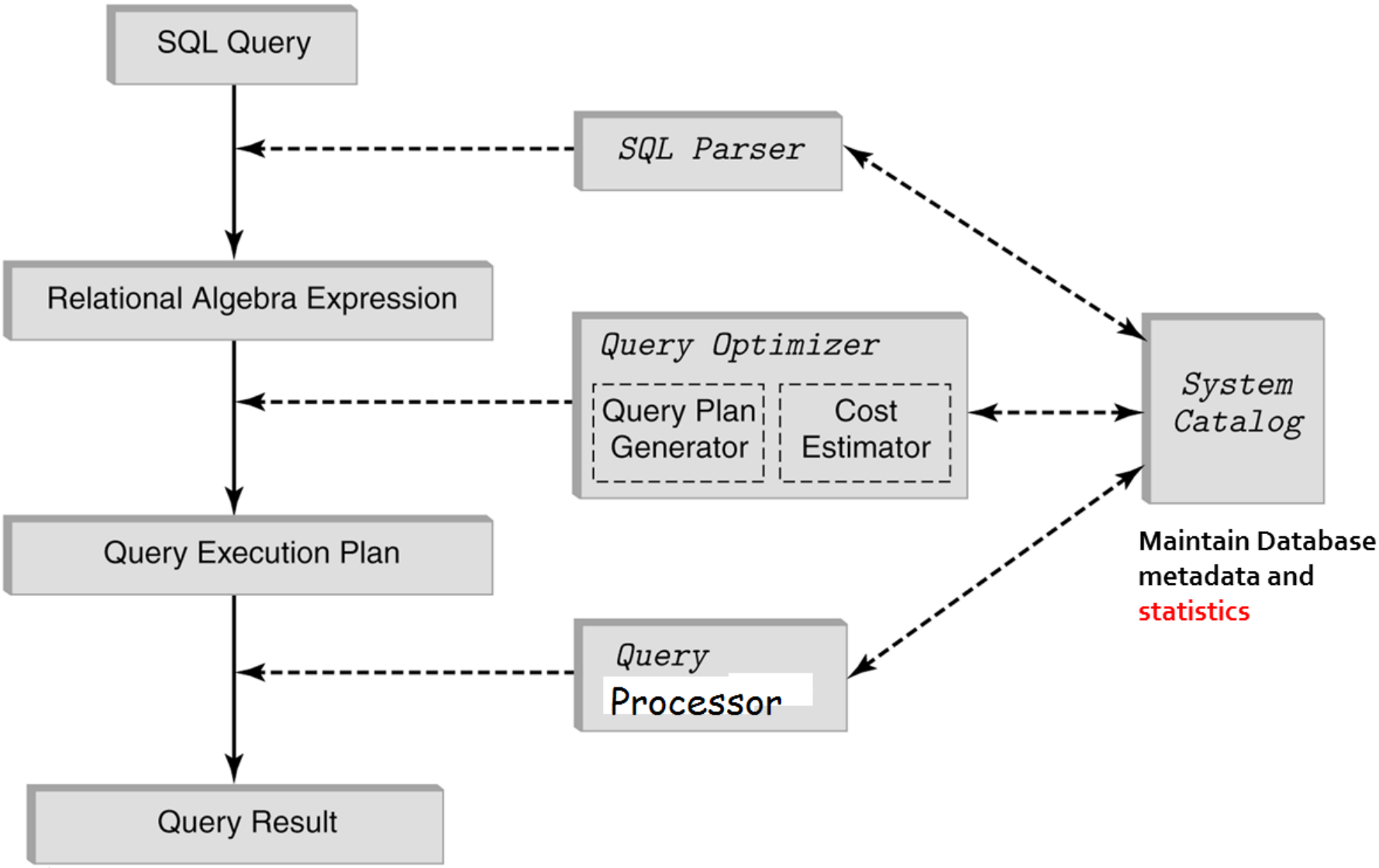
Transformed Into a Logical Plan (tree of logical relational operators)





Translate this into a physical plan that can be executed by the Query Processor

Query Processing Architecture



DBMS Components for Query Processing

- **Query Parser** – Verify validity of the SQL statement. Translate the query into a **parse tree** using Relational Algebra (**RA**)
- **Query Optimizer**:
  - Generate equivalent logical query plans
  - Estimate cost of query plans
  - Select the cheapest plan
  - Generate a physical plan = select **algorithms** for each of the operators in the query
- **Query processor** – Executes the physical query plan

Query Optimization

- **Query Optimization (QO)** = choosing an execution plan with lowest total **execution cost**
  - Execution cost includes **Disk I/Os** and **Processor time**
    - Cost is mainly determined by the **number of disk blocks read and written** during the query execution - because it is the most time consuming action-
- Selecting a good query plan entails deciding
  - Which plan leads to lowest total execution cost
  - Which **algorithm** should be used to implement each of the operators in the query
  - How data from one operation should be passed to the next
- These choices depend on database metadata
  - Size of relations
  - Indexes and data file organization
  - Attributes statistics, e.g., number of distinct values, min, max values

Cost Parameters

- Cost of an operation = number of disk I/Os to
  - Read the operands
  - Compute the result
- **Cost Parameters:**

– **B(R) = # of blocks for relation R**

– **T(R) = # of tuples in relation R**

– **V(R, a) = # of distinct values of attribute a**

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

Database Statistics

- Here are typical statistics kept
  - For a relation
    - Number of tuples, number of tuples per block, number of blocks
  - For an attribute
    - Number of distinct values, min, max values
    - Selection cardinality – avg number of records satisfying an equality condition
  - For an index
    - Number of levels, number of leaf blocks

Updating Statistics

- Success of estimation depends on amount and currency of statistical information DBMS holds
- Keeping statistics current can be problematic
  - If statistics updated every time tuple is changed, this would impact performance
  - DBMS could update statistics on a periodic basis, for example nightly, or whenever the system is idle

Logical Query Plans

Different Query Execution Strategies

Find all Managers that work at a Doha branch:

```
SELECT *
FROM staff s, branch b
WHERE s.bno = b.bno AND (s.position = 'Manager'
                AND b.city = 'Doha');
```

3 equivalent Relational Algerbra (RA) queries are:

$\sigma_{(position='Manager') \wedge (city='Doha') \wedge (staff.bno=branch.bno)} (Staff \times Branch)$

Cartesian Product

$\sigma_{(position='Manager') \wedge (city='Doha')} (Staff \bowtie Branch)$

Join

$(\sigma_{position='Manager'}(Staff)) \bowtie (\sigma_{city='Doha'}(Branch))$

Relational Operations

- **Basic Operators**
  - *Selection* (σ) Selects a subset of rows from relation

  e.g., $\sigma_{age > 60}$(Employee) returns all employees over 60
  - *Projection* (π) Selects desired columns from relation

  e.g., $\pi_{id, salary}$(Employee) returns all IDs, and salaries
- **Set Operations**
  - *Set-difference* ($-$) Tuples in R, but not in S
  - *Union* ($\cup$) Tuples which are either in R, or in S, or in both
  - *Intersection* ($\cap$) Tuples that R and S have in common
  - *Cartesian Product* (**X**) every tuple of R is matched with every tuple of S
  - *Aggregation* (SUM, MIN, etc.) and GROUP BY
- ***Join*** ($\bowtie$) Allows us to combine two relations
  - $R \bowtie S$ is the natural join of the relations *R* and *S*, the set of all combinations of tuples in *R* and *S* that are equal on their **common attribute names**

Assume:

- 1000 tuples in Staff; 50 tuples in Branch;
- 50 Managers; 5 Doha branches;
- No indexes or sort keys;
- Results of any intermediate operations stored on disk;
- Tuples are accessed one at a time

Cost are:

Number of Disk Reads/Writes

(1) (1000 + 50) + 2*(1000 * 50) = 101 050

(2) (1000 + 50) + 2*1000 = 3 050

(3) (1000 + 50) + 2* (50 + 5) = 1 160

- Cartesian product and join operations are much more expensive than selection

- (3) significantly reduces size of relations being joined together

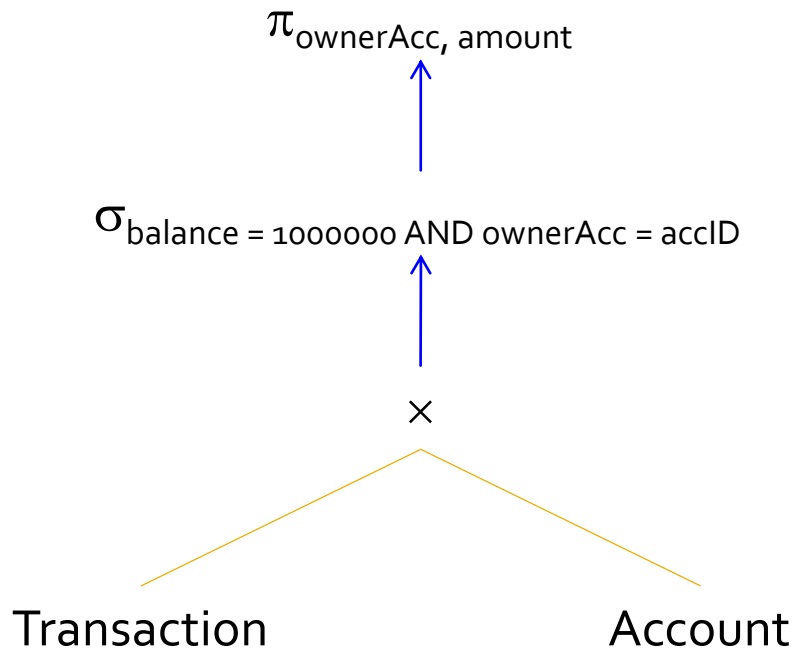
Example 1 - SQL To Algebraic Expression Tree

```
SELECT ownerAcc, amount
FROM Transaction, Account
WHERE balance = 1000000 AND ownerAcc = accID
```

Expression Tree =

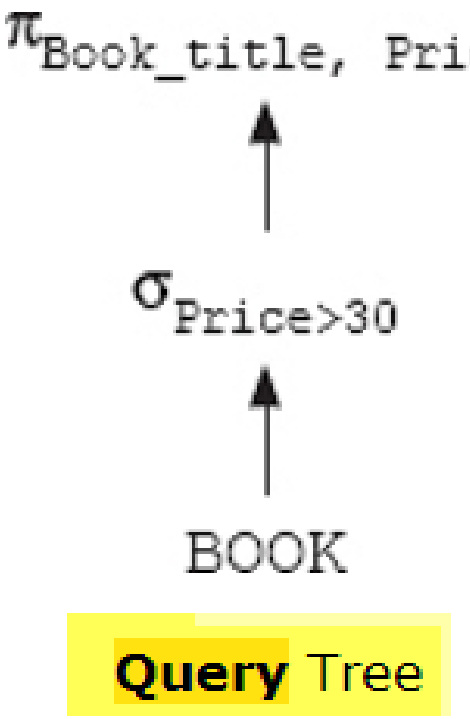
Query Tree =

Data-flow graph of relational algebra operators

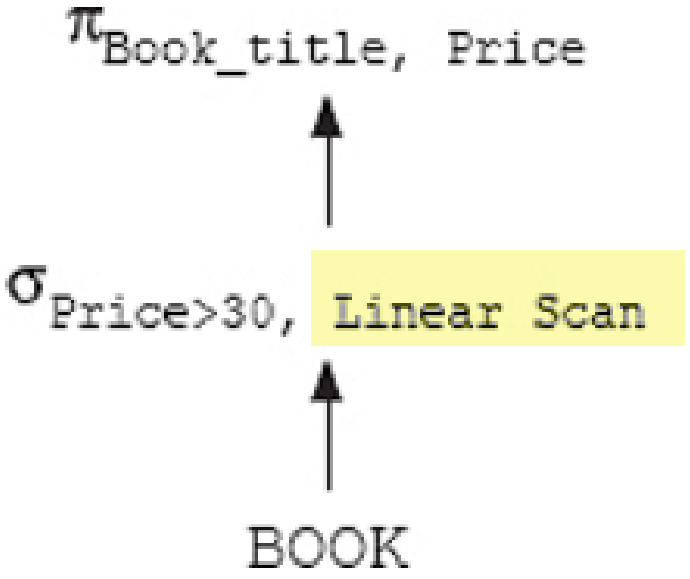


Query Tree vs. Query Plan

$$\pi_{\text{Book\_title, Price}}(\sigma_{\text{Price>30}}(\text{BOOK}))$$



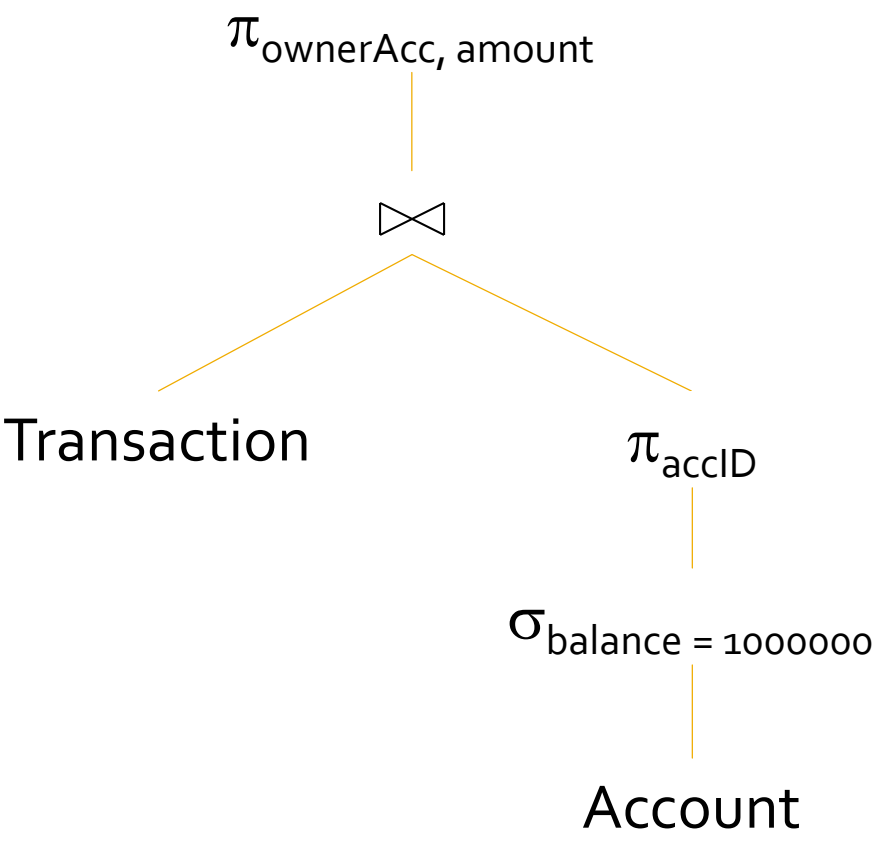
Query Tree



Query-evaluation Plan

Example 2 - SQL To Algebraic Expression Tree

```
SELECT ownerAcc, amount
FROM Transaction
WHERE ownerAcc IN(
        SELECT accID
        FROM Account
        WHERE balance = 1000000)
```



Select-Project-Join-Queries

We will focus on Select-Project-Join-Queries (SPJ)

Select <attribute list>
From <relation list>
Where <condition list>

Example Join Query over **R**(A,B,C) and **S**(C,D,E):

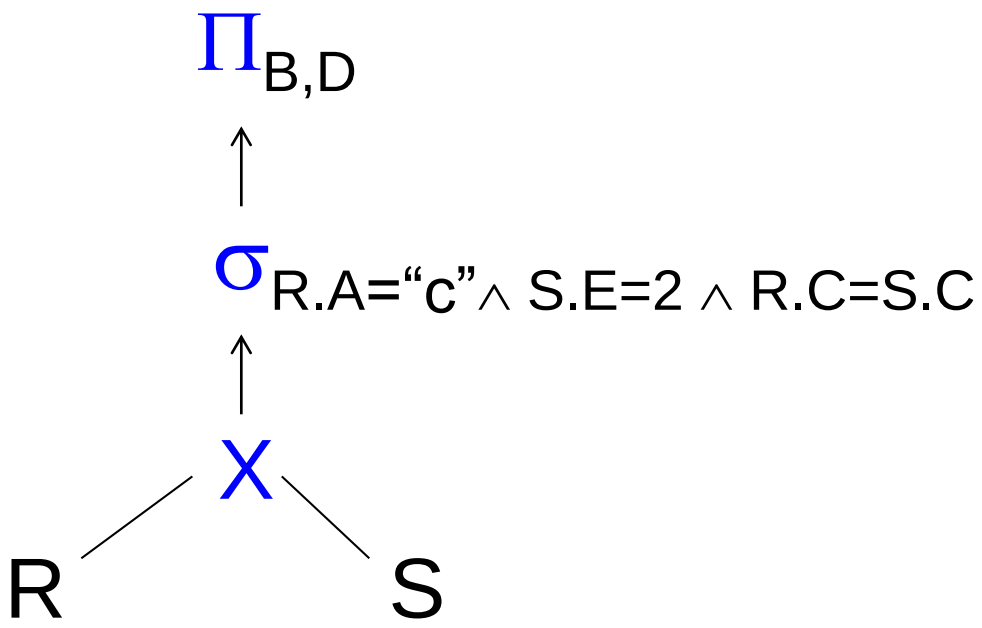
Select B, D
From **R**, **S**
Where R.A = "c" ∧ S.E = 2 ∧ R.C = S.C

Plan I

- Do Cartesian product
- Select tuples
- Do projection

Relational Algebra can be used to describe plans

$$\Pi_{B,D}\,[\,\sigma_{R.A=\text{“c”}\wedge\ S.E=2\ \wedge\ R.C\ =\ S.C}\ (RXS)]$$



Select B,D
From R,S
Where R.A = “c”
∧ S.E = 2 ∧
R.C=S.C

Bingo!

Got one...

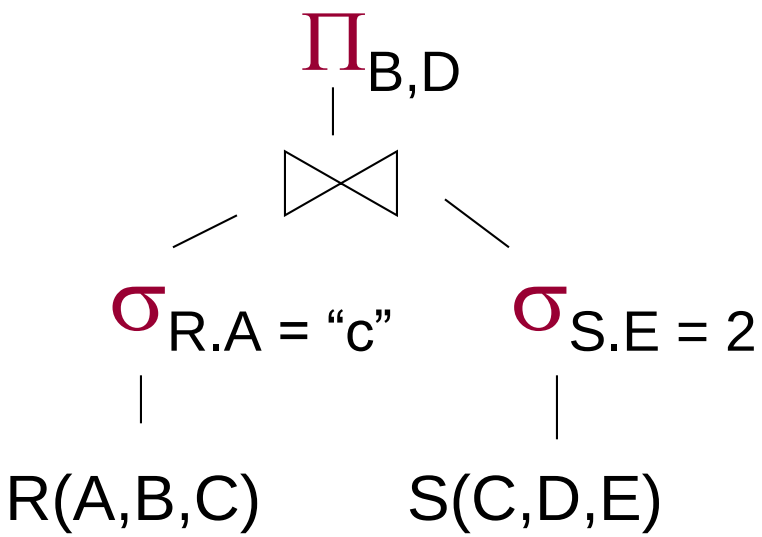
R X S	R.A	R.B	R.C	S.C	S.D	S.E
	a	1	10	10	x	2
	a	1	10	20	y	2
	.					
	.					
	c	2	10	10	x	2
	.					
	.					

Answer

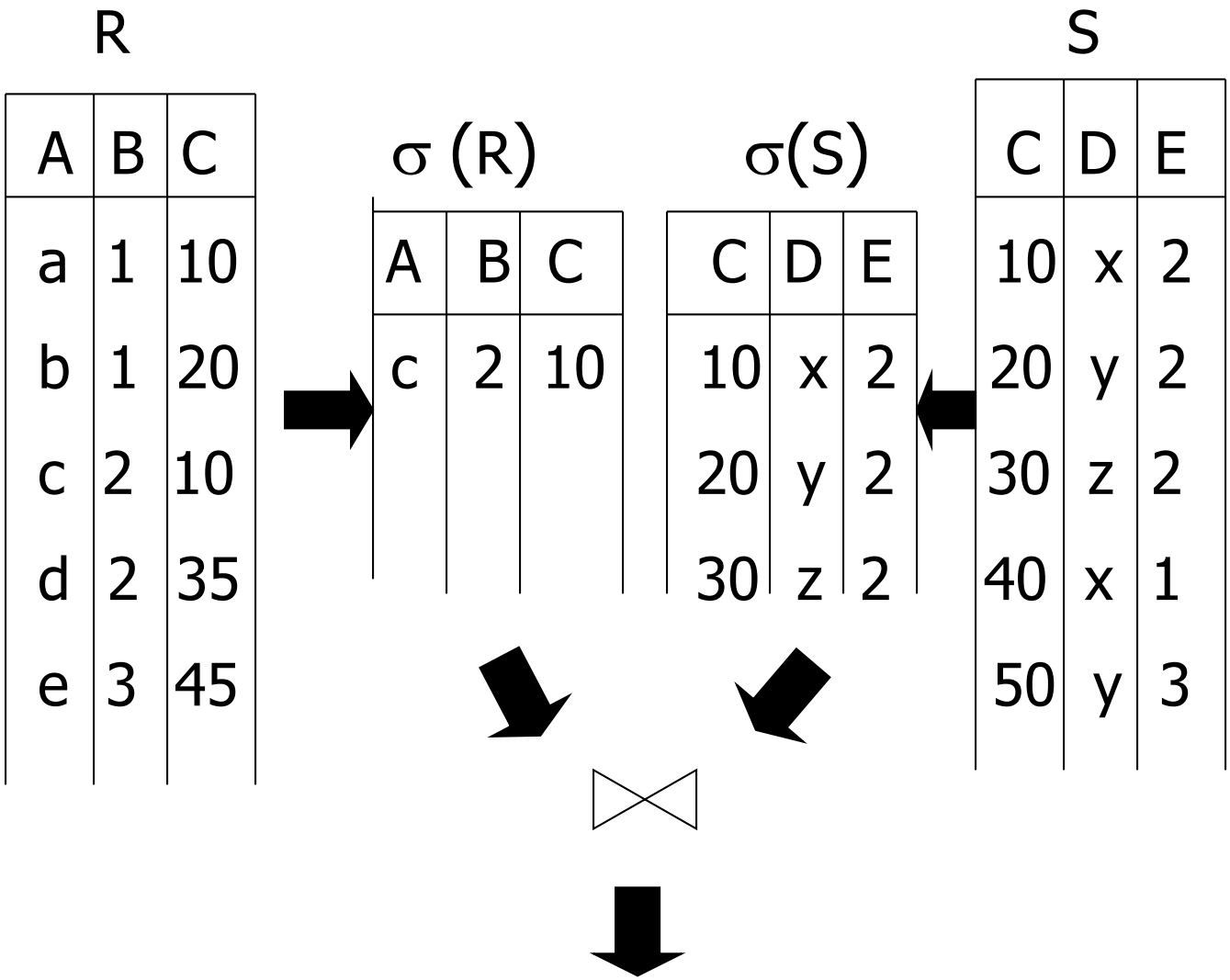
B	D
2	x

Plan II

- Select tuples
- Do natural join ⋈
- Do projection



Select B,D
From R,S
Where R.A = "c" ∧ S.E = 2 ∧ R.C=S.C



Answer

B	D
2	x

Select B,D

From R,S

Where R.A = “c” ∧ S.E = 2 ∧ R.C=S.C

Plan III

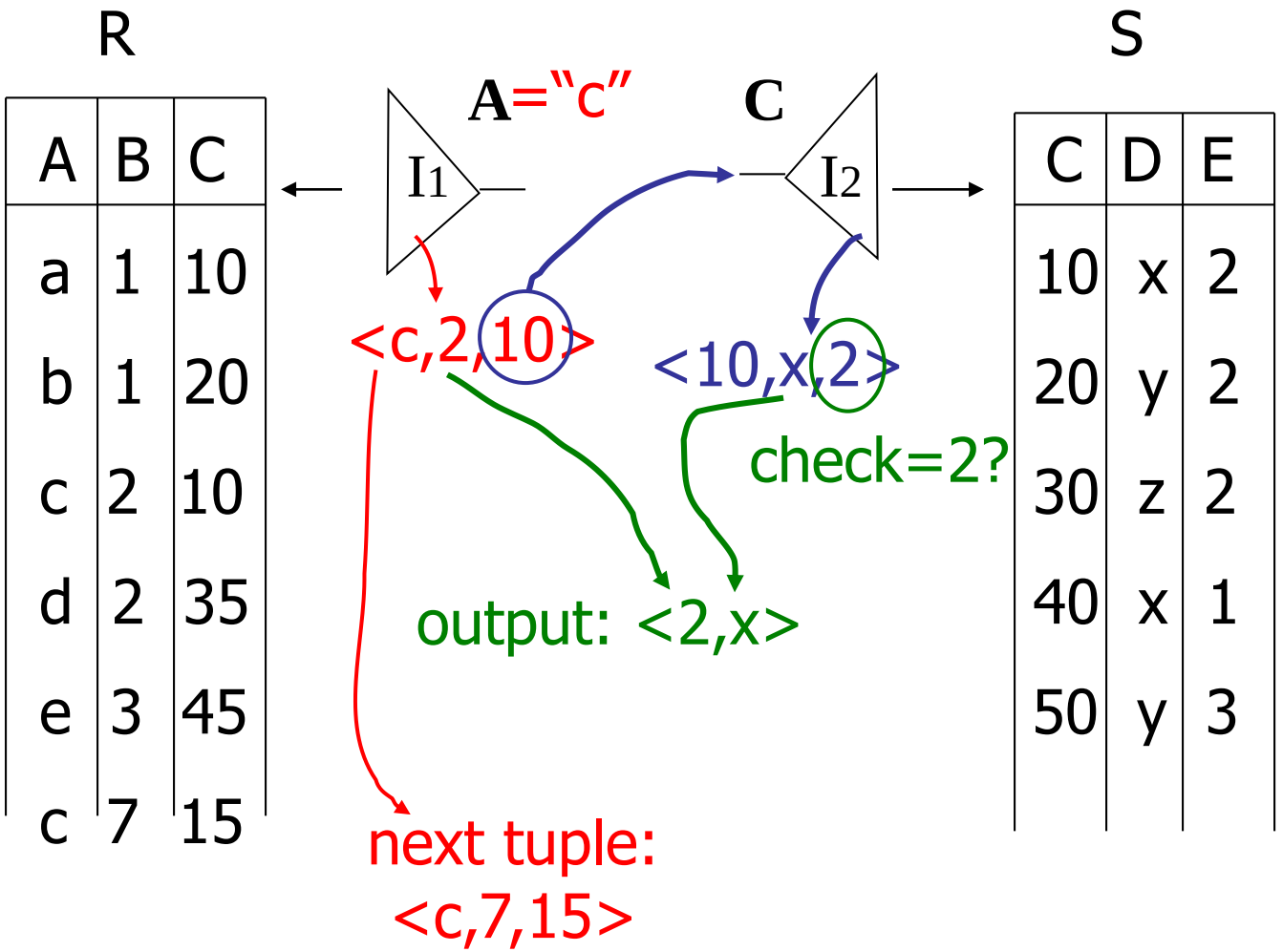
Use R.A and S.C Indexes

(1) Use R.A index to select R tuples with R.A = “c”

(2) For each R.C value found, use S.C index to find matching tuples

(3) Eliminate S tuples S.E $\neq$ 2

(4) Join matching R,S tuples, project B,D attributes, and place in result



Select B,D

From R,S

Where R.A = “c” ∧ S.E = 2 ∧ R.C=S.C

Query Rewrite Rules

Some Query Rewrite Rules

- Transform one logical plan into a more efficient one using equivalences in relational algebra
- Some common Query Rewrite Rules
  - Perform Selection operations as early as possible
    - Cut down the number of rows involved since rows can be discarded sooner
  - Perform Projection as early as possible.
    - Cut down the number of columns involved
  - Combine Selections and Cartesian products into equijoins
  - Compute common expressions once.
    - If common expression appears more than once then compute once and store the result and reuse it when required.
  - Some Subqueries can be converted to Joins

Equivalences in Relational Algebra

$R \bowtie S = S \bowtie R$ Commutativity

$(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$ Associativity

Joins are commutative and associative => Relations may therefore be joined in any order

Also holds for: Cartesian Product, Union, Intersection

$R \times S = S \times R$

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

$R \cup S = S \cup R$

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

Join Order

- The order in which joins and Cartesian products are made affects the size of intermediate relations
  - Which, in turn, affects the time taken to process a query
- Consider these three relations:
  - **Customer = {CustID, fn, ln, age}** – 1,000 records
  - **Account = {AcctID, type, balance}** – 1,200 records
  - **Owns = {CustID^{fkCustomer}, AcctID^{fkAccount}}** – 1,400 records
- **Owns $\bowtie$ (Customer $\bowtie$ Account)**
  - Intermediate relation – 1,000 * 1,200 = 1,200,000 records
- **(Owns $\bowtie$ Customer) $\bowtie$ Account**
  - Intermediate relation – 1,400 records

Rules: $\sigma + \bowtie$ combined

Let p = predicate with only R attributes

q = predicate with only S attributes

m = predicate with both R,S attributes

$$\sigma_p (R \bowtie S) = [\sigma_p (R)] \bowtie S$$

$$\sigma_q (R \bowtie S) = R \bowtie [\sigma_q (S)]$$

A selection can be *pushed through* a Join

Rules: σ + $\bowtie$ combined (continued)

$$\sigma_{p \wedge q}(R \bowtie S) = [\sigma_p(R)] \bowtie [\sigma_q(S)]$$

$$\sigma_{p \wedge q \wedge m}(R \bowtie S) =$$

$$\sigma_m \left[(\sigma_p R) \bowtie (\sigma_q S) \right]$$

$$\sigma_{p \vee q}(R \bowtie S) =$$

$$\left[(\sigma_p R) \bowtie S \right] \cup \left[R \bowtie (\sigma_q S) \right]$$

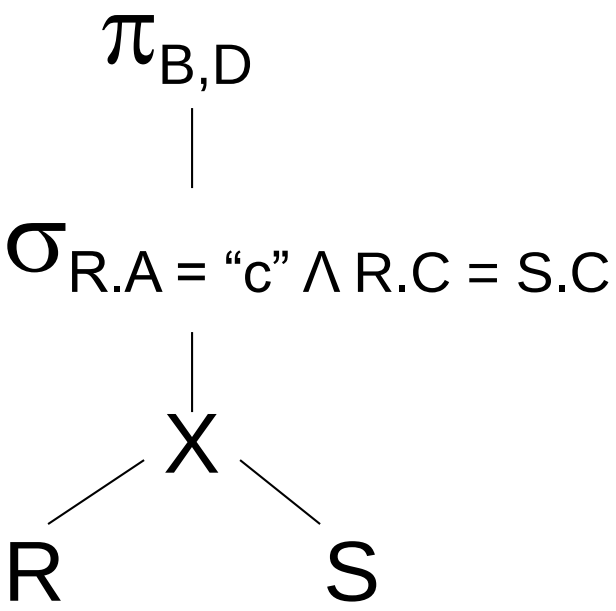
Example

$$\sigma_{\text{age>50} \wedge \text{opName="lobotomy"}}(\text{Patient} \bowtie \text{Operation}) \equiv$$

$$\sigma_{\text{age>50}}(\text{Patient}) \bowtie \sigma_{\text{opName="lobotomy"}}(\text{Operation}))$$

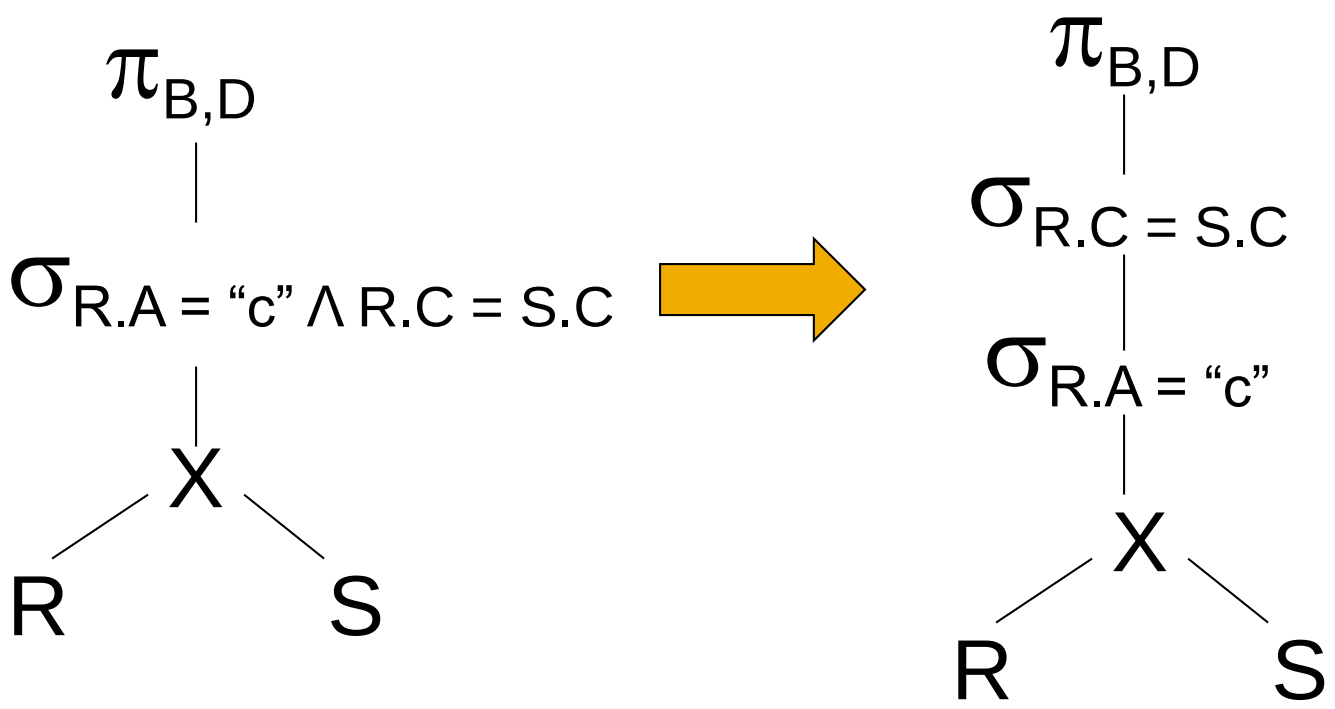
Example 1 - Initial Logical Plan

Select B,D
From R,S
Where R.A = “c” ∧ R.C=S.C



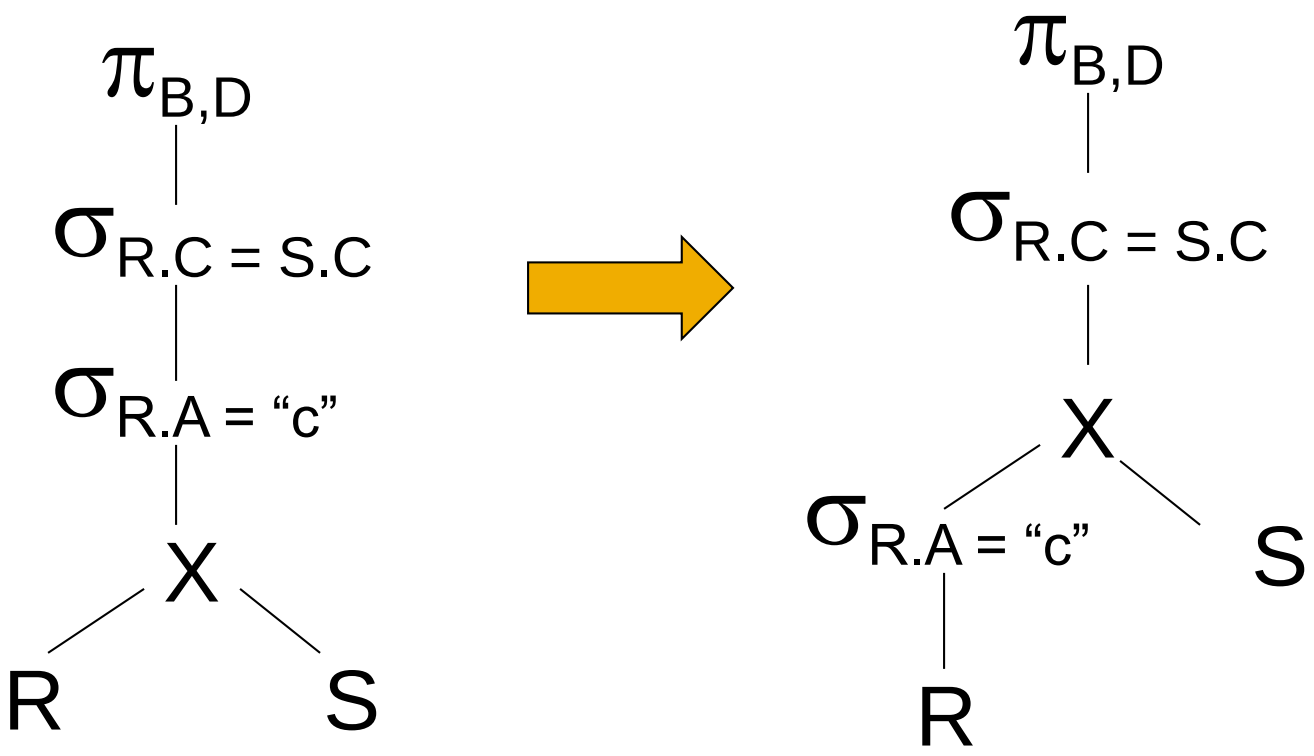
Relational Algebra: $\Pi_{B,D}[\sigma_{R.A=\text{“c”}\wedge R.C = S.C}(RXS)]$

Apply Rewrite Rule (1) – Break complex selection into simpler ones



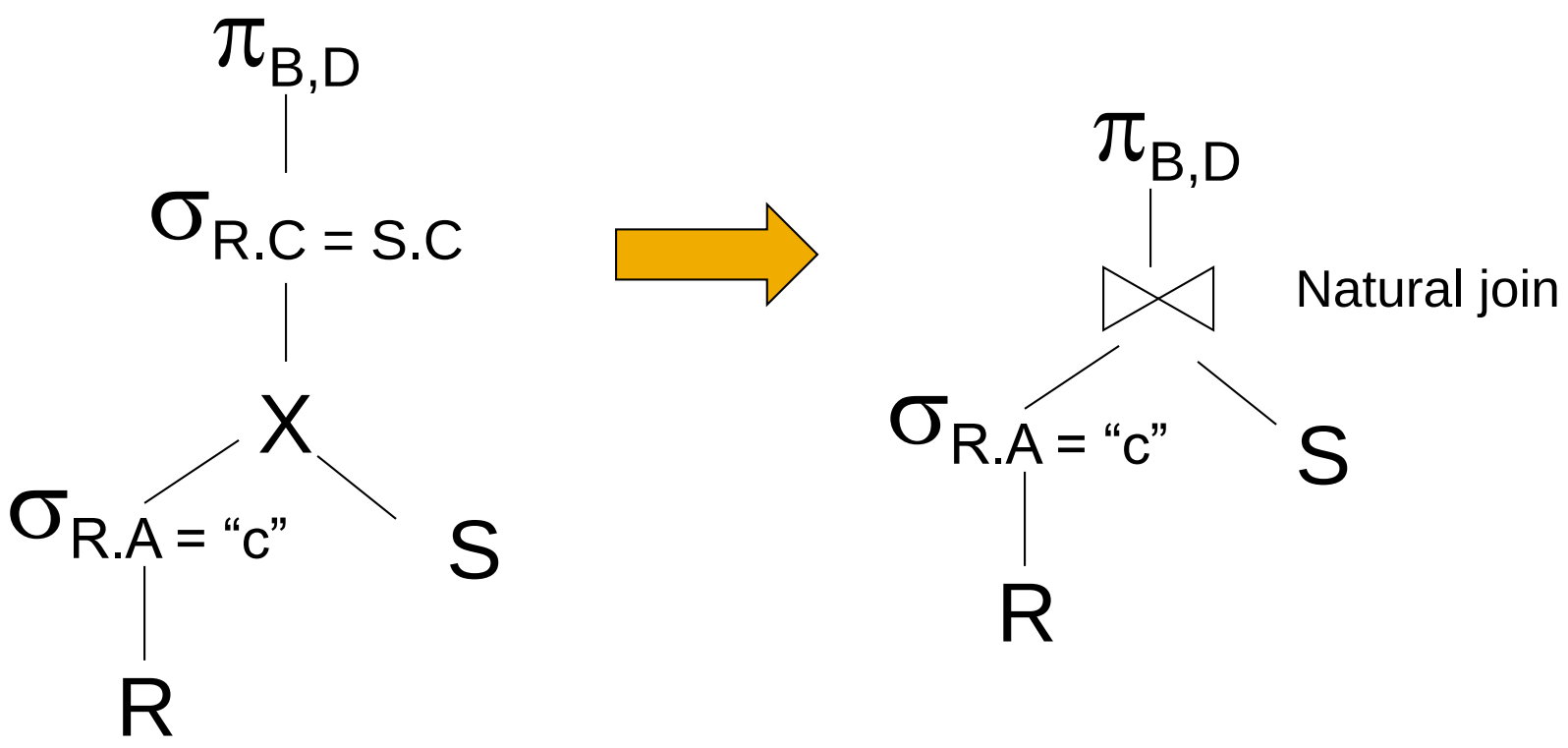
$$\Pi_{B,D}\,[\,\sigma_{R.C=S.C}\,[\sigma_{R.A=\text{“c”}}(R \times S)]]$$

Apply Rewrite Rule (2) – Push Selection down



$$\Pi_{B,D}\,[\,\sigma_{R.C=S.C}\,[\sigma_{R.A=\text{“c”}}(R)]\ X\ S]$$

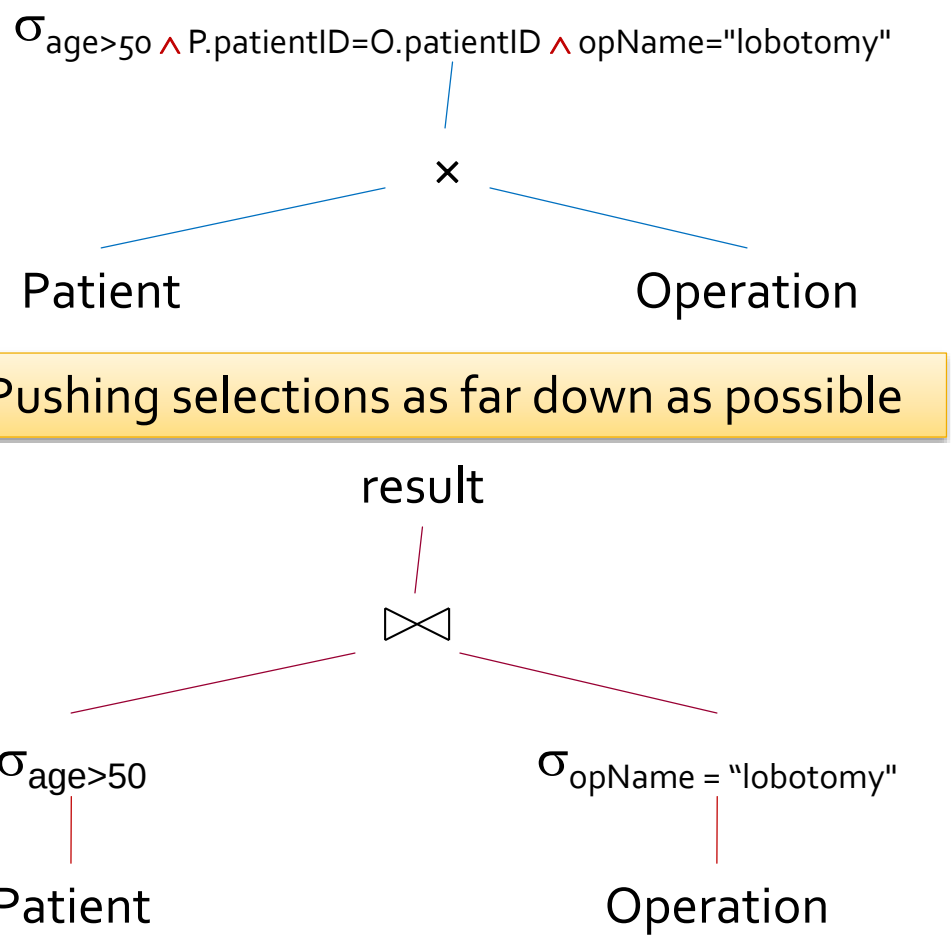
Apply Rewrite Rule (3) – Push Selection Down to replace Cross-Join with Natural Join



$$\Pi_{B,D}\,[[\sigma_{R.A=\text{“c”}}(R)] \bowtie S]$$

Example 2 - Pushing Selections down

$\sigma_{age>50 \wedge P.patientID=O.patientID \wedge opName = "lobotomy"}(Patient \times Operation)$



Do projects early

Example: R(A,B,C,D,E)

p condition: (A=3) ∧ (B="cat")

$$\pi_E [\sigma_p (R)] \equiv \pi_E [\sigma_p [\pi_{ABE}(R)]]$$

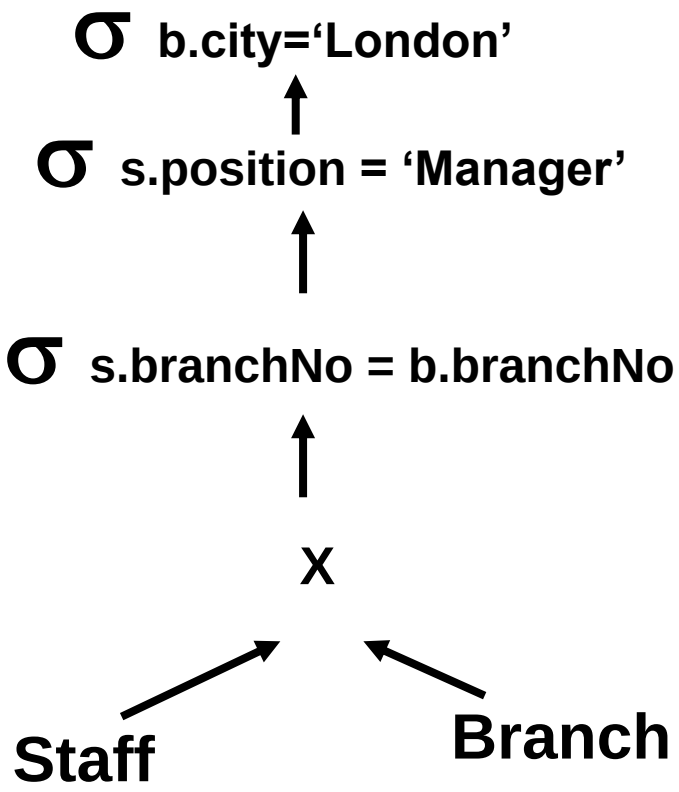
- **Projections can be pushed through joins**

$$\pi_{\text{patientID,age, opName}} [\sigma_{\text{age>50}}(\text{Patient}) \bowtie \text{Operation}] \equiv$$

$$\sigma_{\text{age>50}} [\pi_{\text{patientID,age}}(\text{Patient})] \bowtie \pi_{\text{patientID,opName}}(\text{Operation})$$

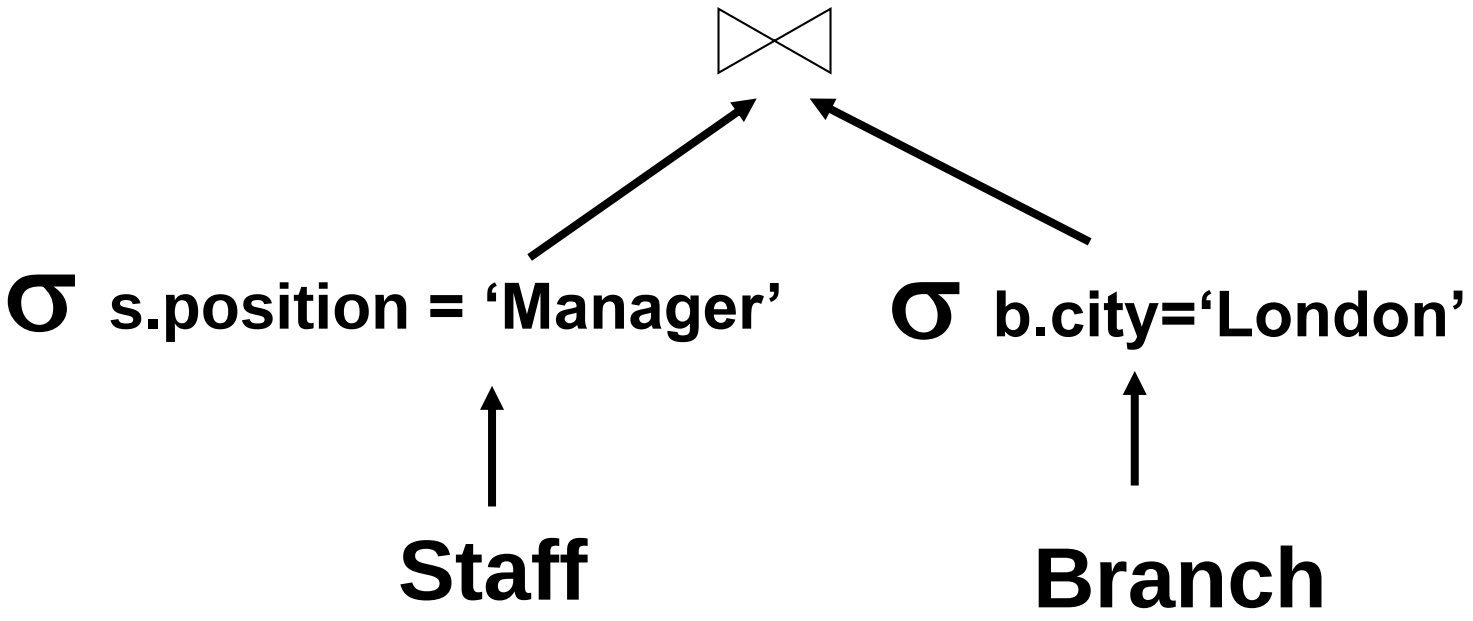
Example 3 – Initial relational algebra tree

```
SELECT * FROM Staff s, Branch b
WHERE s.branchNo = b.branchNo
AND s.position = 'Manager'  AND b.city='London' ;
```



Example 3 – Improved relational algebra tree

The selections have been done first to reduce the number of rows involved in the join

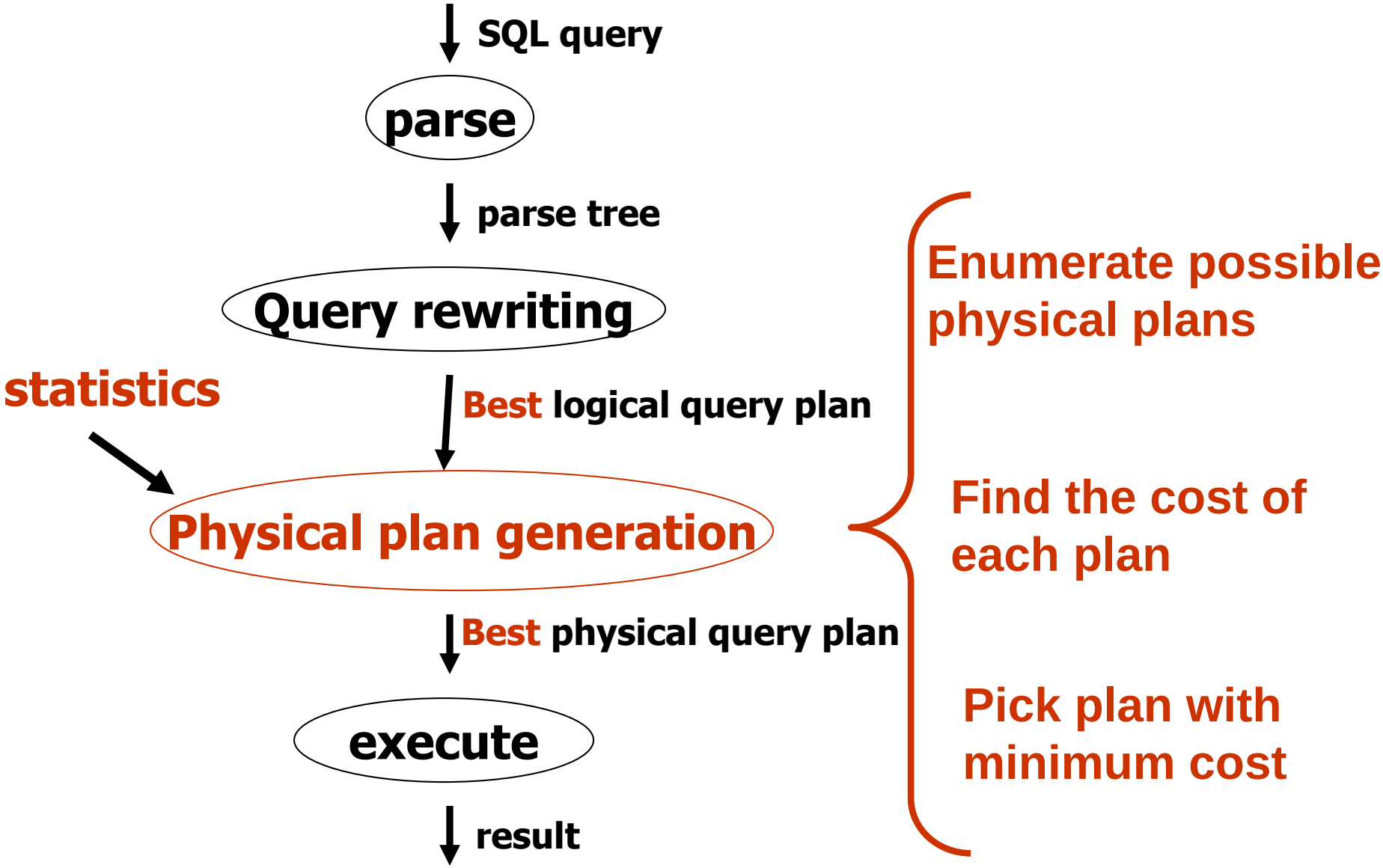


Summary of Optimization Rules

- Perform selection early (reduces number of tuples)
- Perform projection early (reduces number of attributes)
- Replace Cartesian Product by join whenever possible
- Pick a good join order, based on the **expected size** of intermediate results

Physical Query Plan

Generate Physical Plan

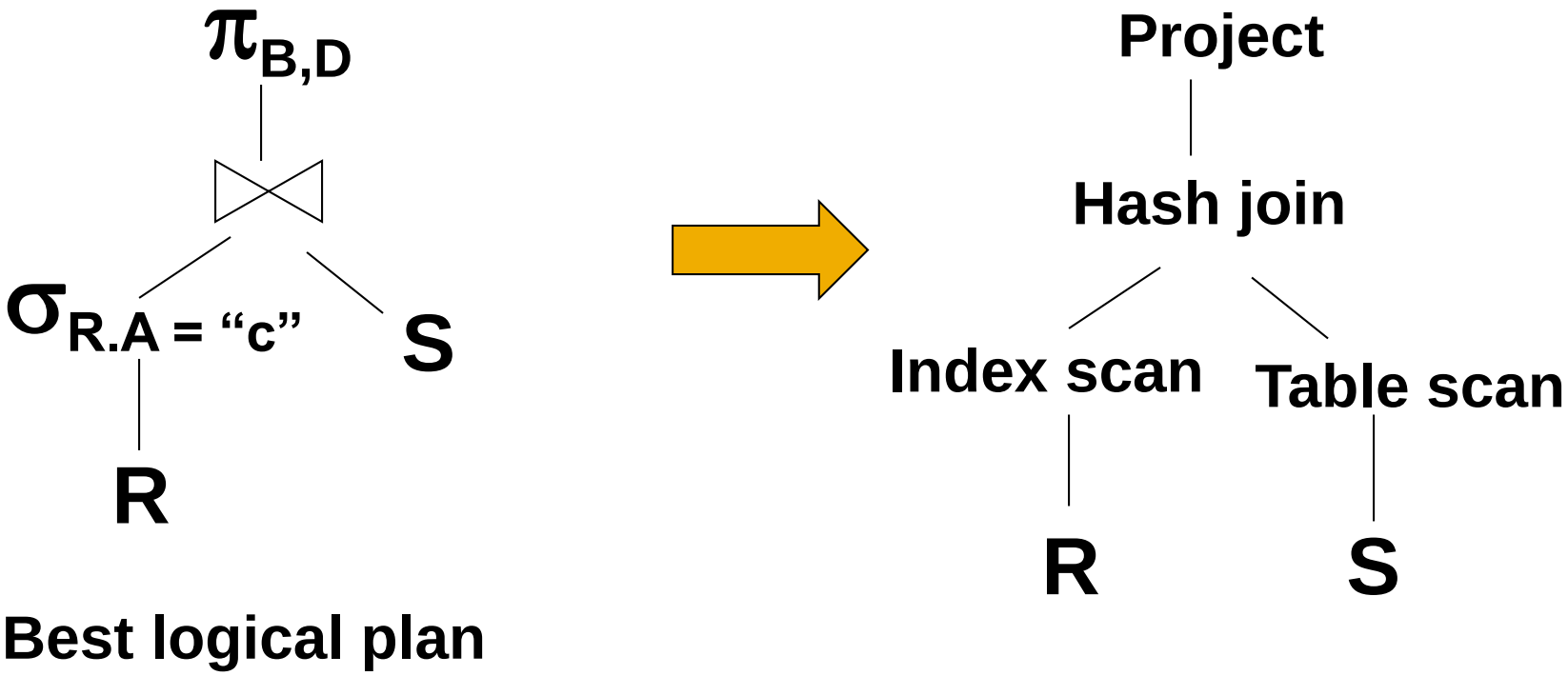


Approaches to Query Evaluation

- Many DB operations require reading tuples, comparing attributes of tuples with values, or comparing tuples from different tables
- Techniques generally used:
  - ***Iteration***: for/while loop to scan and compare tuples of a Relation
  - ***Index lookup***: if comparison of attribute that's indexed, look up matches in index & return those
  - ***Sort/merge***: iteration against presorted data
  - ***Hash***: build hash table of the tuple list, *probe* the hash table

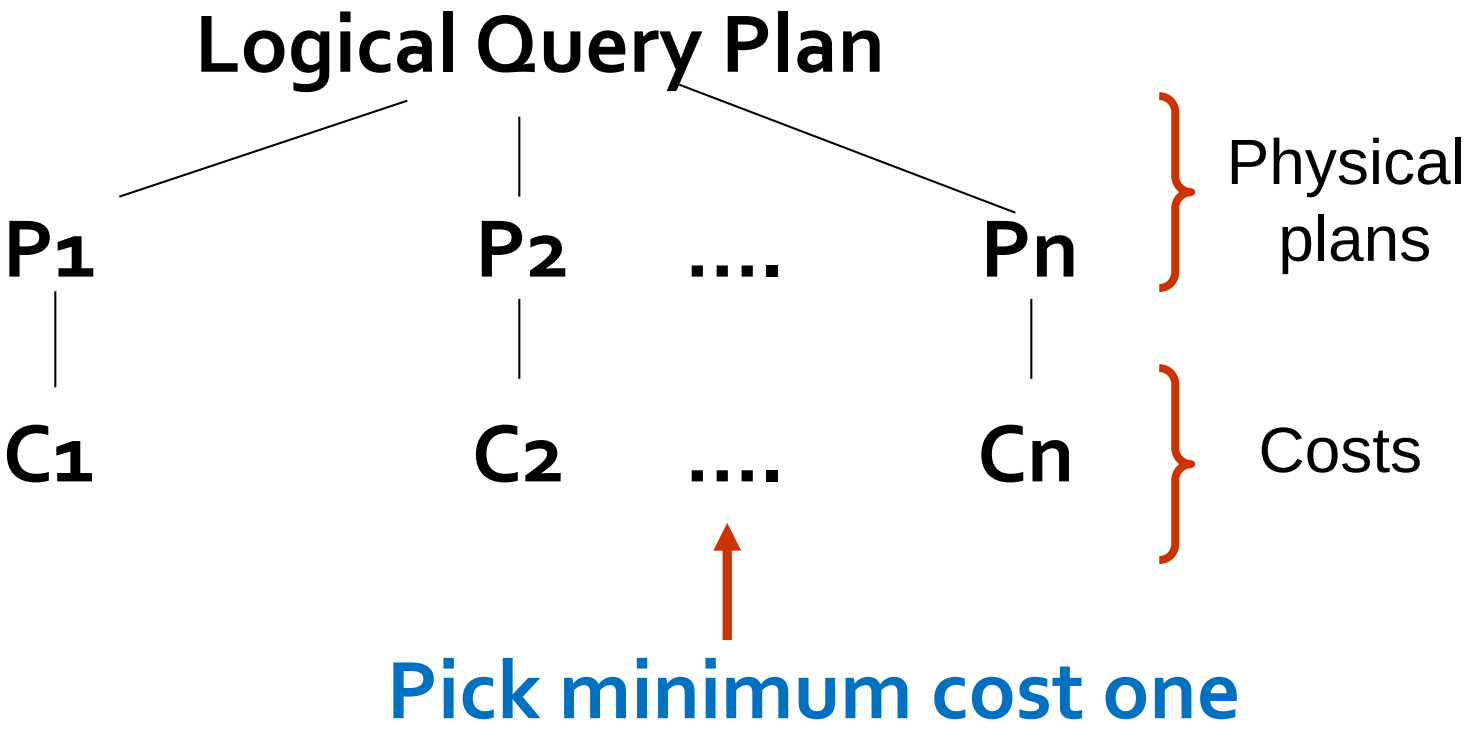
➢ *Must be able to support larger-than-memory data*

Logical Plans Vs. Physical Plans

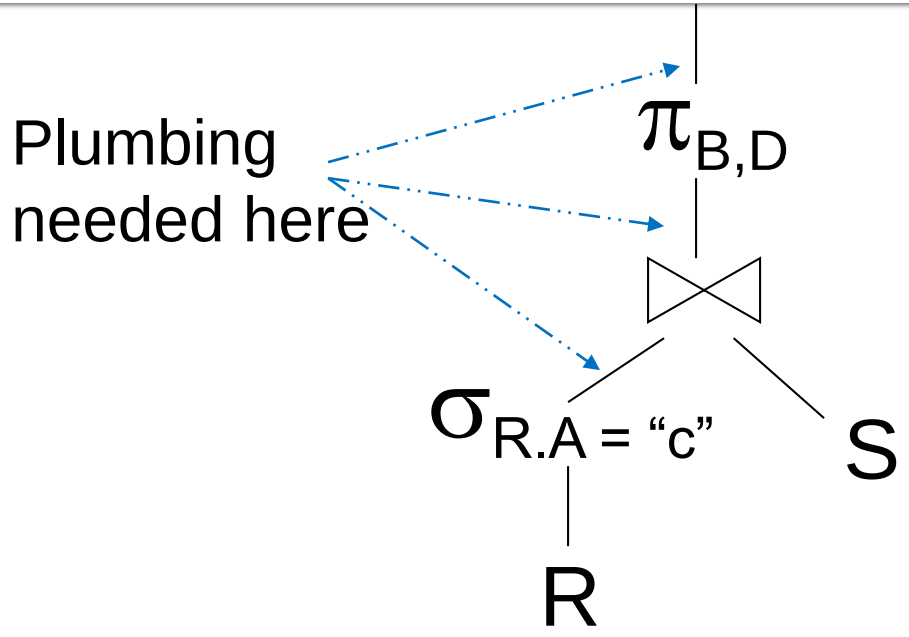


Physical Plan = Select algorithms for each of the operators in the query + Decide Pipelining vs. Materialization of intermediate results

Physical Plan Generation



Operator Plumbing: Materialization vs. pipelining



- **Materialization**:

output of one operator written to disk, next operator reads from the disk

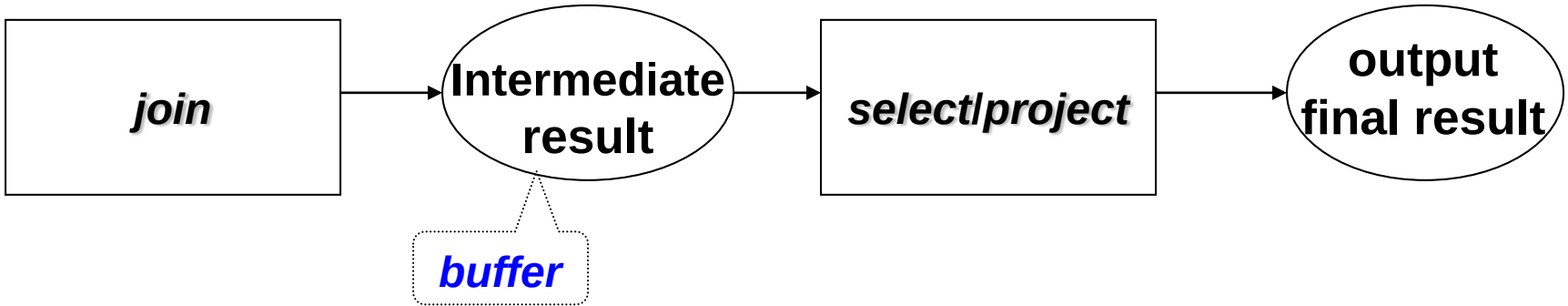
- **Pipelining**:

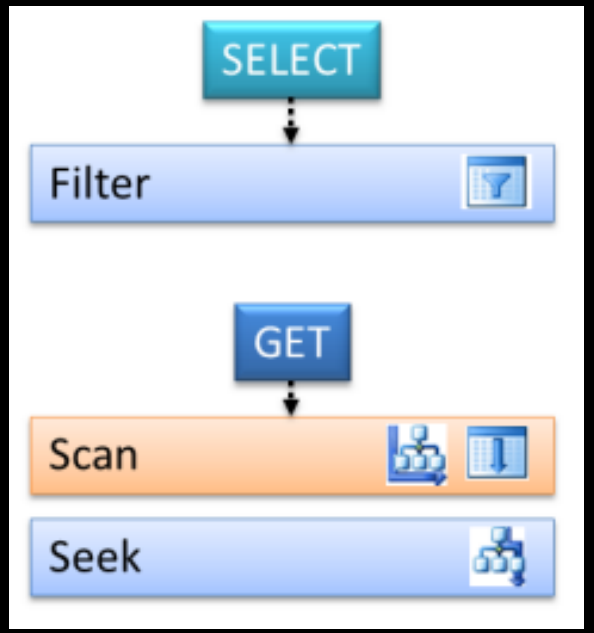
output of one operator directly fed to next operator

  - Saves on cost of creating temporary relations and reading results back in again

Pipelining Example

- *join* and *select/project* act as co-routines, operate as **producer/consumer sharing a buffer** in main memory.
  - When join fills buffer; ***select/project*** filters it and outputs result
  - Process is repeated until *select/project* has processed last output from join
- Performing select/project adds no additional cost





Implementing Selection

$\sigma_{(attr\ op\ value)}$

Computing Selection $\sigma_{(attr\ op\ value)}$

- The algorithm to use for implementing *Selection* depend on the indexes and data file organization
- *Key strategies*:
  - ***File scan***: can be used for any condition
  - ***B⁺ tree seek***: equality *or* range search
  - ***Hash index seek***: equality search

$\sigma_{(attr\ op\ value)}$ with No index on *attr*

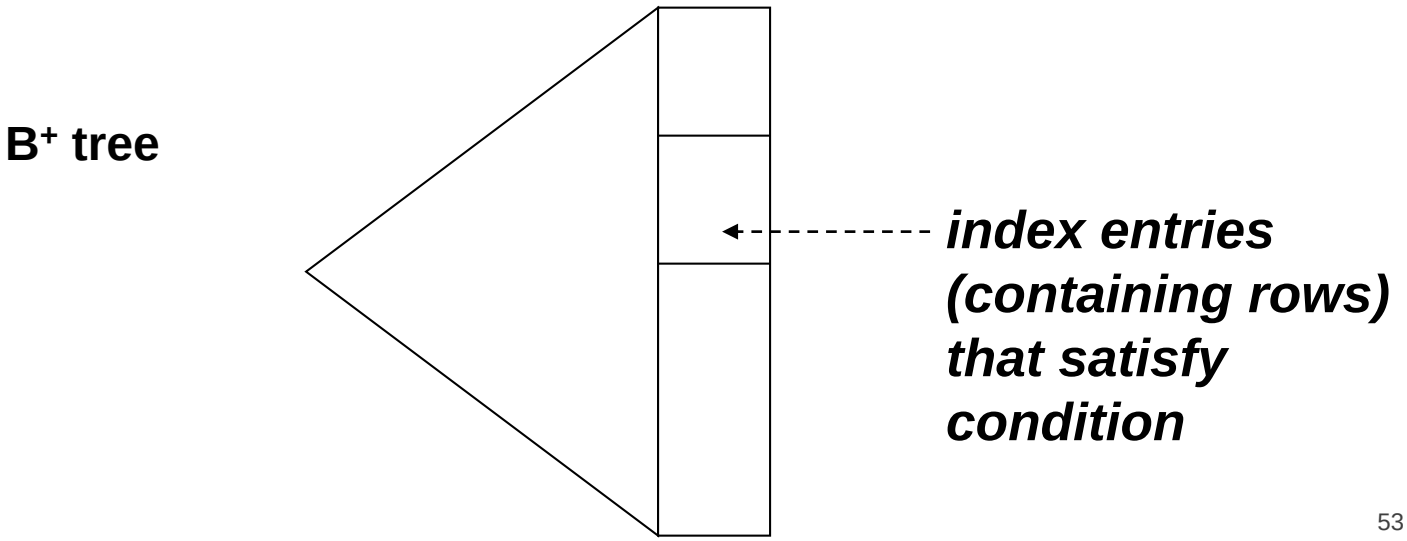
- If rows <u>are not</u> sorted on *attr:*
  - Scan all data blocks to find rows satisfying selection condition
  - **Cost = B(R)** **(# of blocks for relation R)**
  - **Avg Cost = B(R) / 2** - *if attr is a key (the scan is terminated when the required record is found)*

$\sigma_{(attr\ op\ value)}$ with Clustered B+ tree index on attr

- Clustered B^+ tree index on *attr* (for "=" or range search):
  - Locate first index entry corresponding to a row in which (*attr* = *value*).

    Cost = depth of tree
  - Rows satisfying condition packed in sequence in successive data blocks; scan those blocks.

    Cost = number of **blocks** occupied by qualifying rows



$\sigma_{(attr\ op\ value)}$ with Unclustered B+ tree index on attr

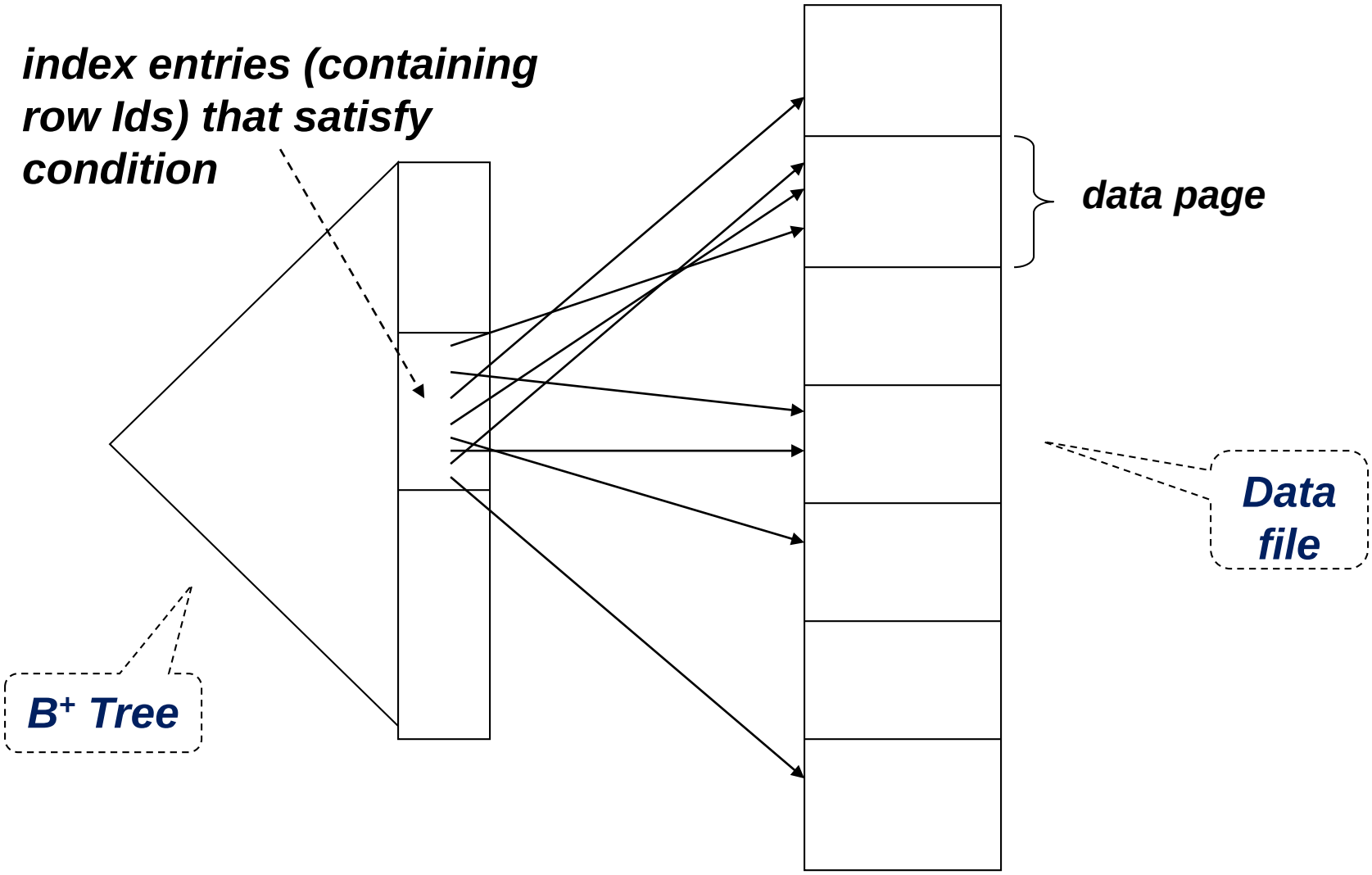
- **Unclustered** B^+ tree index on *attr* (for "=" or range search):
  - Locate first index entry corresponding to a row in which (*attr* = *value*).

    Cost = depth of tree
  - Index entries with pointers to rows satisfying condition are packed in sequence in successive index blocks
    - Scan entries to identify table data blocks with qualifying rows

      Any block that has at least one such row must be fetched once

    Cost = number of qualifying ***rows***

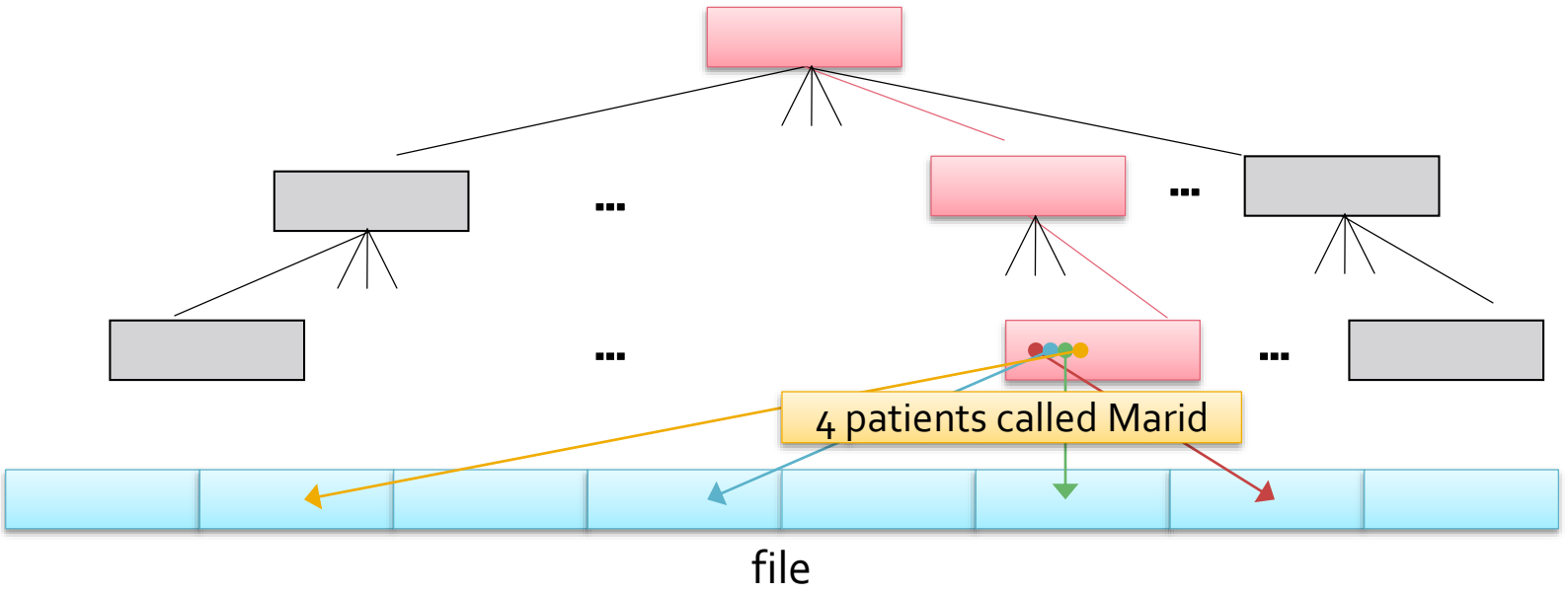
Unclustered B+ Tree Index



Unclustered B+ Tree Range Selection

```
SELECT *
FROM Patient
WHERE lName = 'Marid'
```

$\sigma_{\texttt{lName = "Marid"}}(\texttt{Patient})$



$\sigma_{(attr\ =\ value)}$ with Hash index on attr

- Hash index on *attr* (for "=" search only):

Cost ≈ 1.2

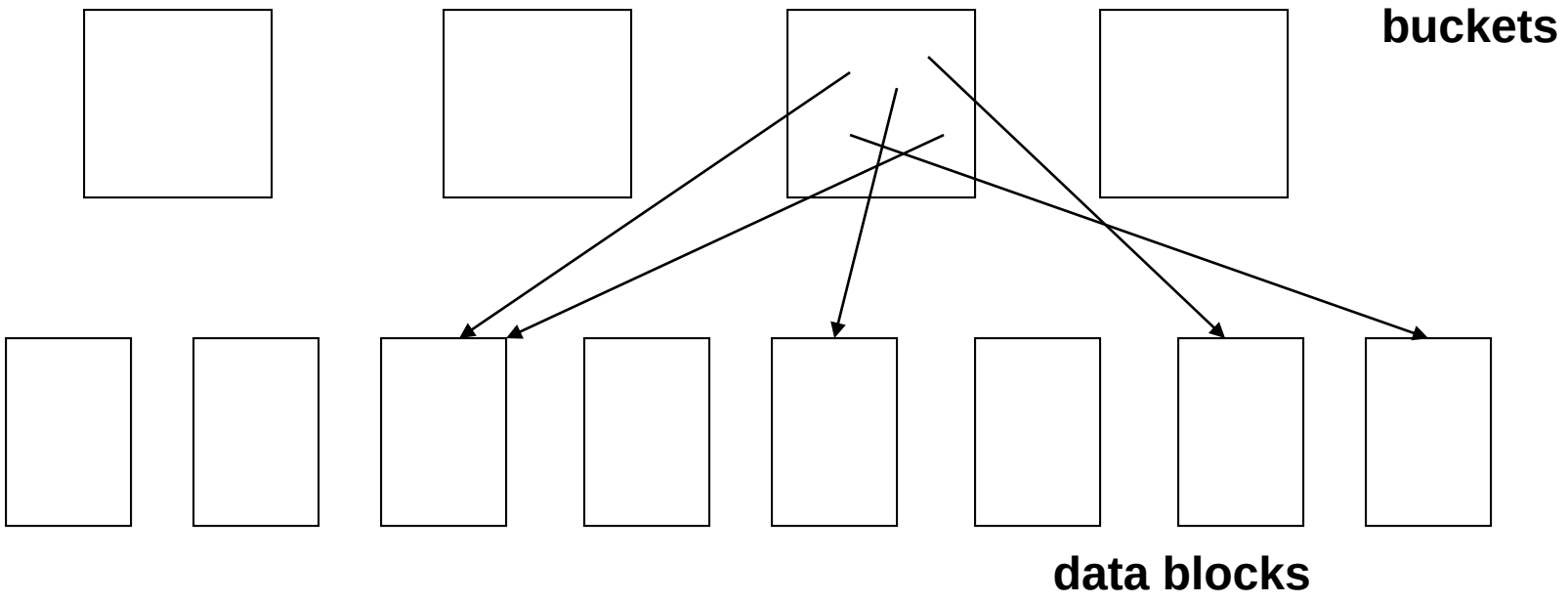
  - 1.2 – typical average cost of finding an entry in a hash index (> 1 due to possible overflow chains)
  - Finds the bucket containing all index entries satisfying selection condition
  - Clustered index – all qualifying rows packed in sequence in successive data blocks; scan those blocks

    Cost = number of **blocks** occupied by qualifying rows
  - Unclustered index – scan the index entries to identify data blocks with qualifying rows
    - Each page containing at least one such row must be fetched once

    Cost = *number of qualifying **rows** in bucket*

Unclustered Hash Index

- Unclustered hash index on *attr* (for equality search)



Sort-Merge Algorithm

External Sorting

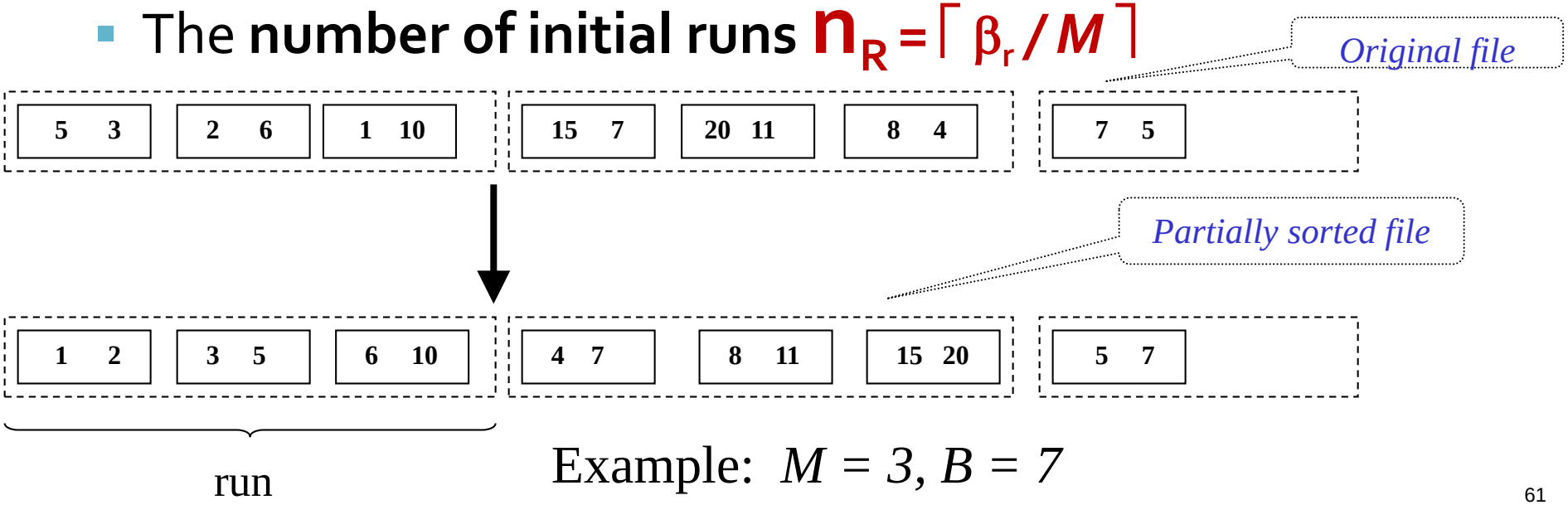
- Sorting is used in implementing many relational operations such *Select Distinct*
- Problem:
  - Relations are typically large, do not fit in main memory (e.g., sort 1Gb of data with 1Mb of RAM)
  - So cannot use traditional in-memory sorting algorithms

  => Combine in-memory sorting with clever techniques aimed at minimizing I/O
- External sorting has two main components:
  - I/O necessary to move records between disk and main memory
  - Computation involved in sorting records in buffers in main memory

Sort-Merge Algorithm
Sort phase

- M = # memory blocks available

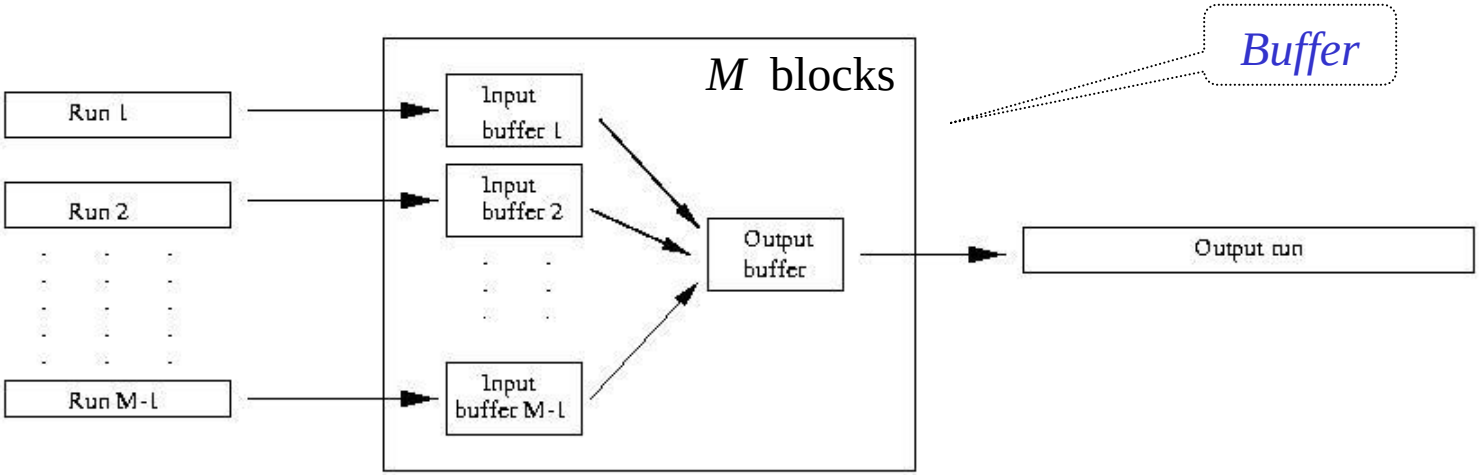
  β_r = # blocks in file to be sorted
- Sort-Merge algorithm has two phases: **Sort then Merge**
  - **Sort phase**: sort ***M*** blocks at a time; create ***B/M*** sorted ***runs*** on Disk as temporary sub-files. **Cost = $2\beta_r$**
  - The **number of initial runs** $n_R = \lceil \beta_r / M \rceil$



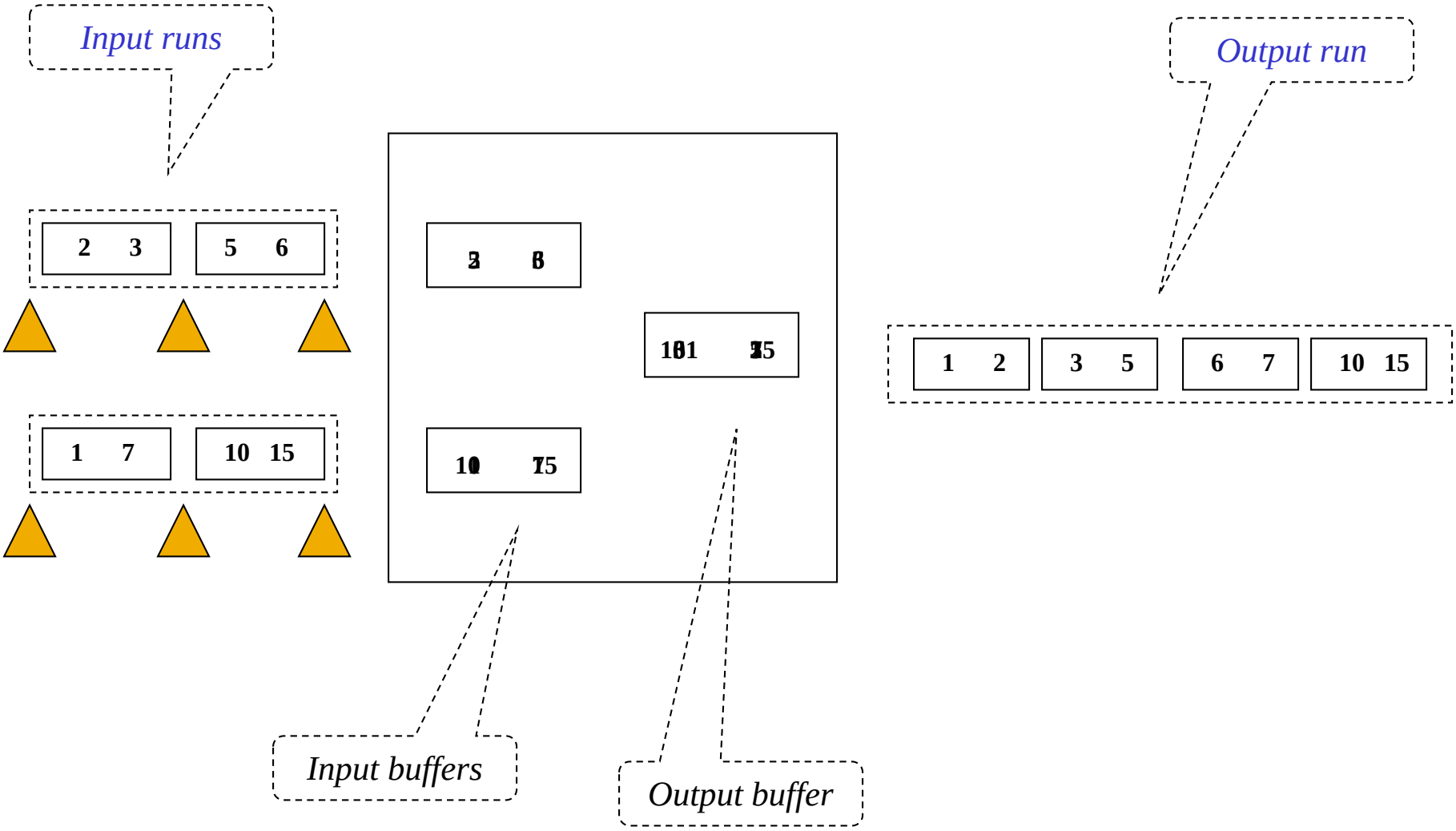
Example: $M = 3, B = 7$

Sort-Merge Algorithm
Merge Phase

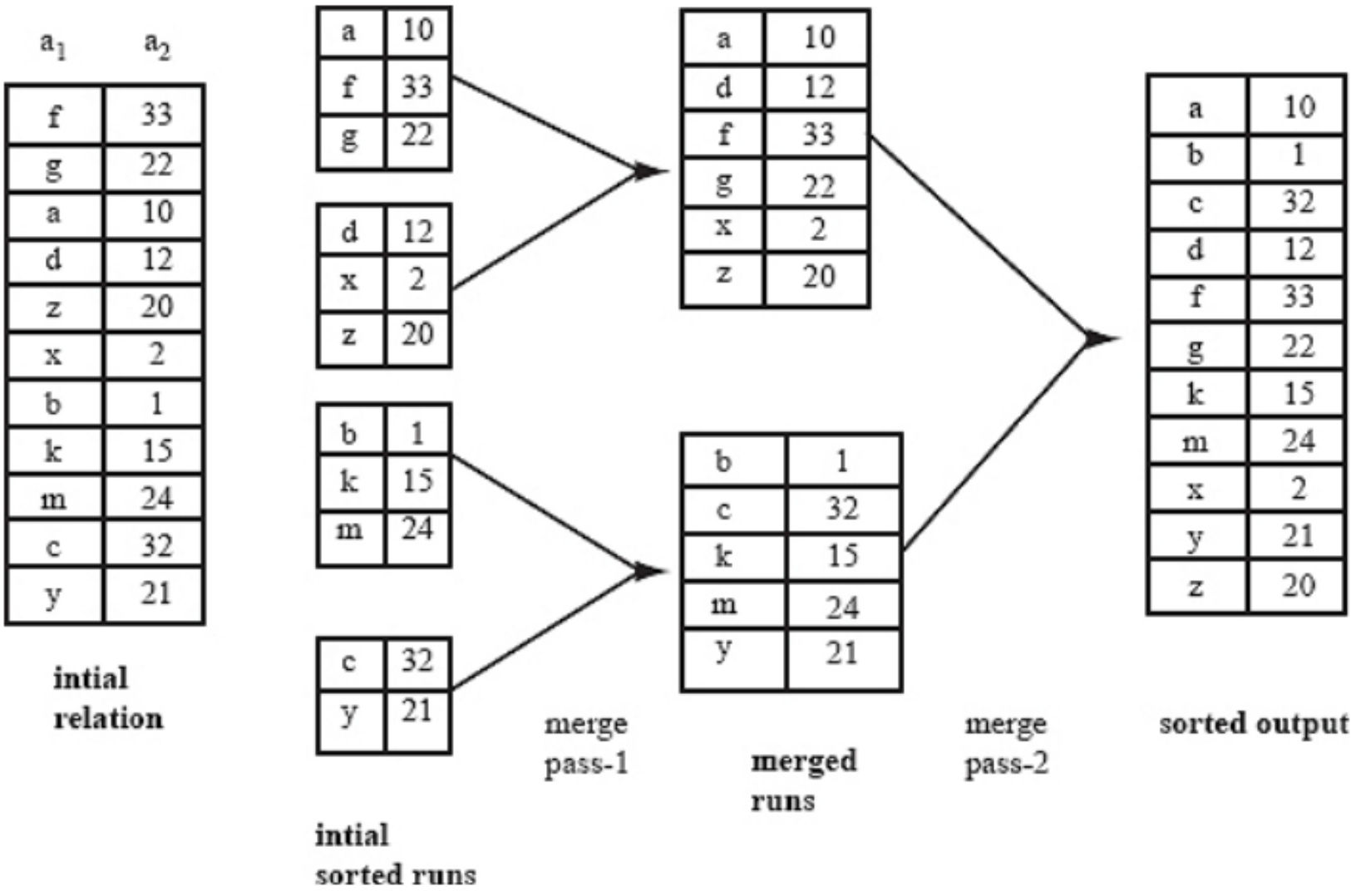
- **Merge Phase**: merge all runs into a single run using ***M-1*** buffers for input and ***1*** Output Buffer
  - Divide **runs** into groups of size M-1 and merge each group into a larger run of sorted records
  - **Number of Passes** for merge phase = $\lceil Log_{M-1} n_R \rceil$
  - **Cost** of Merge Phase = **2 β_r * Number of Passes**
  - **Total Cost = 2 β_r * (# of passes for Merge Phase + 1)**



Merge Example



Sort-Merge Example



Example Sort Merge Cost

- Buffer : with 5 buffer blocks
- File to sort : 108 blocks
  - The **number of initial runs?**
  - How many merge passes?

  - Pass 0 (Sort Phase):
    - Size of each run?
    - Number of runs?

  - Pass 1:
    - Size of each run?
    - Number of runs?

  - Pass 2: ???

Example Sort Merge Cost

- Buffer : with 5 buffer blocks
- File to sort : 108 blocks
- The **number of initial runs? 22**
- How many merge passes? $\lceil \text{Log}_{M-1} n_R \rceil = \lceil \text{Log}_4 22 \rceil = 3$

Sort
- Pass 0: $\lceil 108 / 5 \rceil$ = 22 sorted runs of 5 blocks each (last run is only 3 blocks)

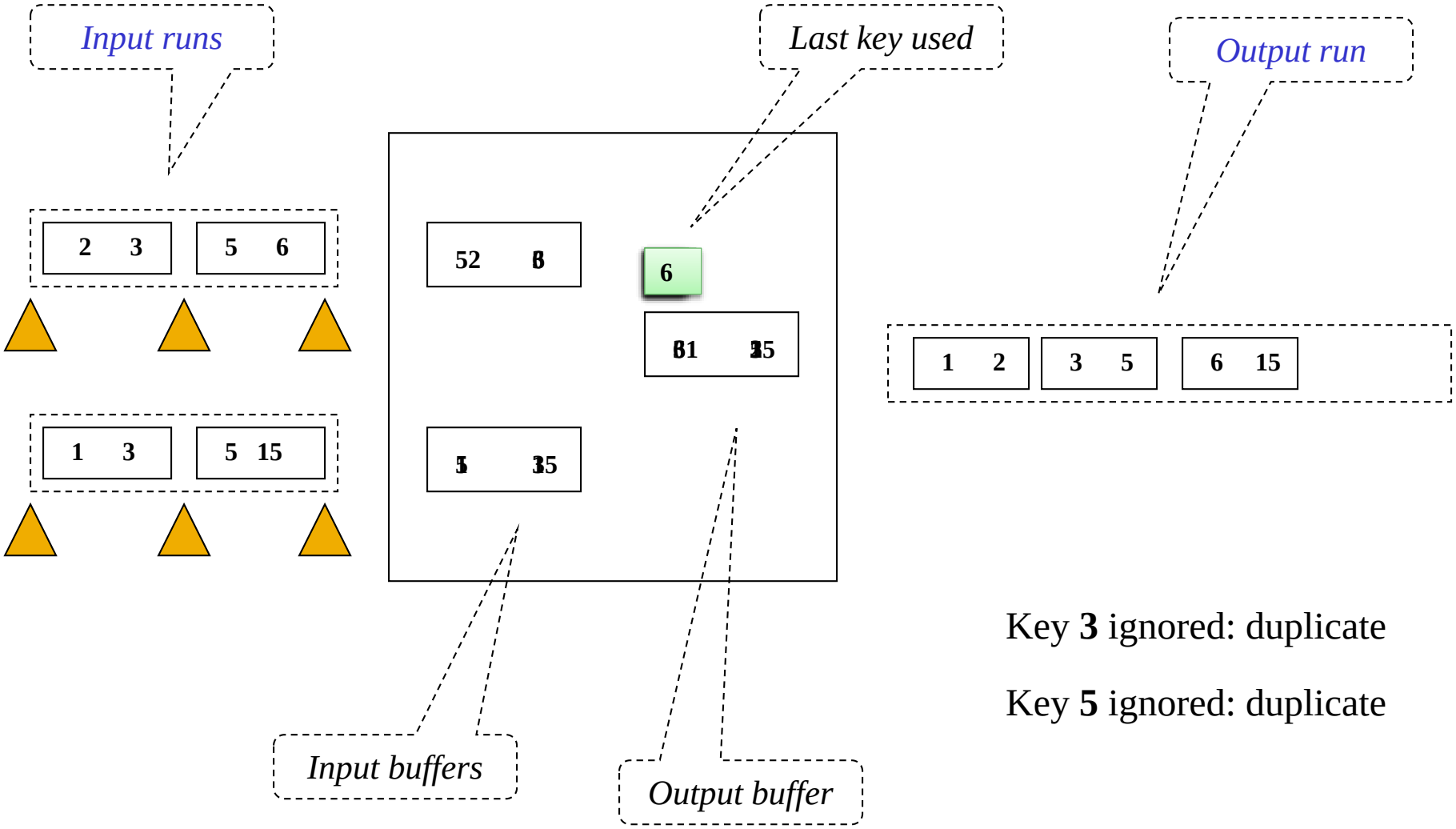
Merge
- Pass 1: $\lceil 22 / 4 \rceil$ = 6 sorted runs of 20 block each (last run is only 8 blocks)
- Pass 2: 2 sorted runs, 80 blocks and 28 blocks
- Pass 3: Sorted file of 108 blocks

- **Total I/O costs? = 2B * (4 passes)**

Duplicate Elimination

- A major step in computing ***Distinct*** *Projection*, ***Union***, and ***Difference*** relational operators
- Algorithm:
  - Sort
  - During the merge step eliminate duplicates on the fly
  - No additional cost (with respect to sorting) in terms of I/O

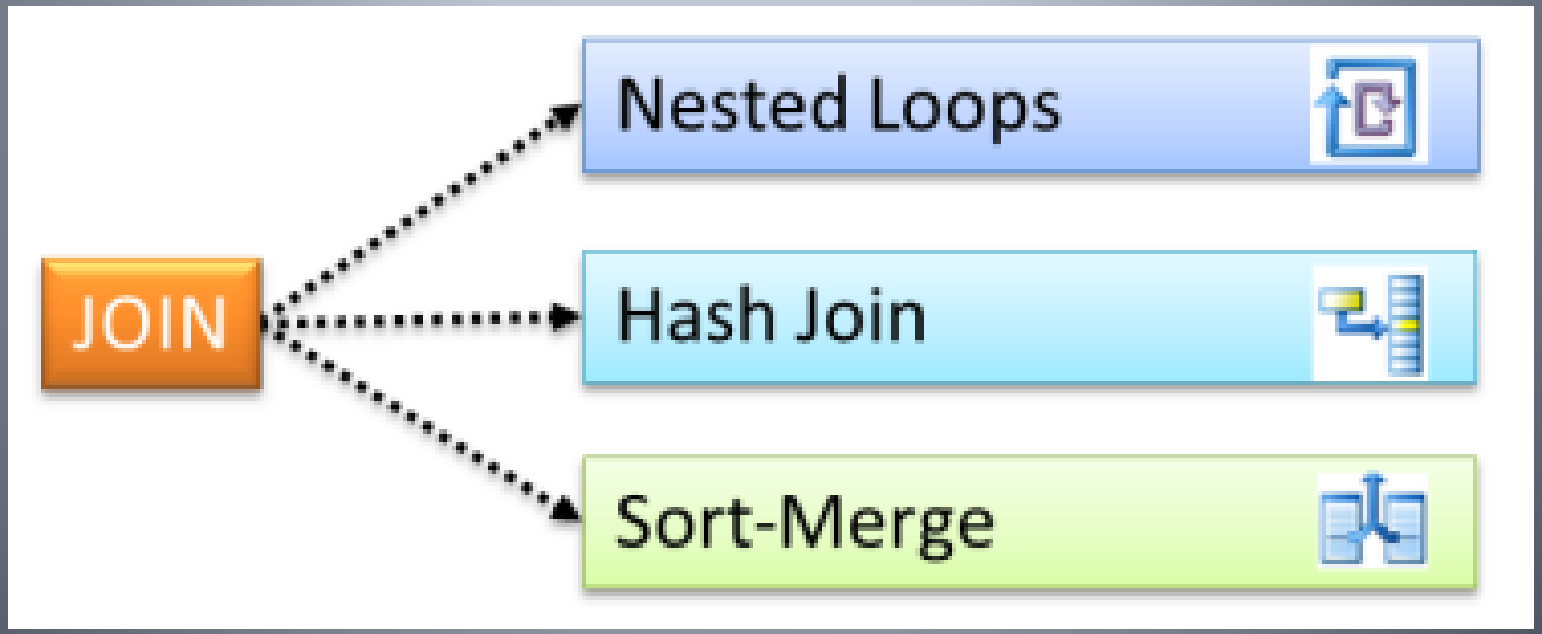
Duplicate elimination During Merge



Key **3** ignored: duplicate

Key **5** ignored: duplicate

Join Operator Implementation



Join Operation Implementation

- SELECT * FROM Reservations R, Sailors S where R.sid = S.id

- The main strategies for implementing the join are:
  - Block Nested Loop Join
  - Indexed Nested Loop Join
  - Sort-Merge Join
  - Hash-Join (building a Hash index on the fly)

Operator Cost Model

- We will compare the Join implementation strategies using a simple cost model
  - Count # of disk blocks read and written during operator execution
  - Cost of query plan = Sum of operator costs
  - Ignoring CPU costs
- **Parameters used in Cost Model**

β_r = # blocks storing R tuples

T_r = # tuples in R

M = # memory blocks available

Nested Loop Join (NLJ)

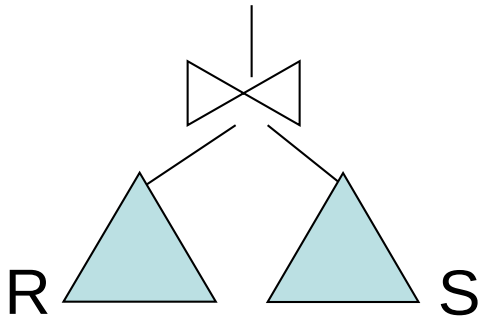
R

B	C
a	10
a	20
b	10
d	30

$\bowtie$

S

C	D
10	cat
40	dog
15	bat
20	rat



NLJ can be:

- Tuple-based
- Block-based

- **NLJ (conceptually)**

```
for each r ∈ R do
    for each s ∈ S do
        if (r.C = s.C) then output r,s pair
```

Nested Loop Join (NLJ) Example

- **select** * **from** Customer c, Rental r **where** c.accountId = r.accountId
- Simple nested loops join

```
while (not customer.eof()) {
  c = customer.read();
  rental.reset();
  while (not rental.eof()) {
   r = rental.read();
   if (c.accountId==r.accountId) {
         result.write(c, r);
  }
}
```

- Can reduce cost by comparing all records in block of customer against all rentals

Analysis of Tuple-based NLJ

- Cost with R as outer = $T_r + (T_r \times T_s)$
- Cost with S as outer = $T_s + (T_s \times T_r)$

Block-based NLJ

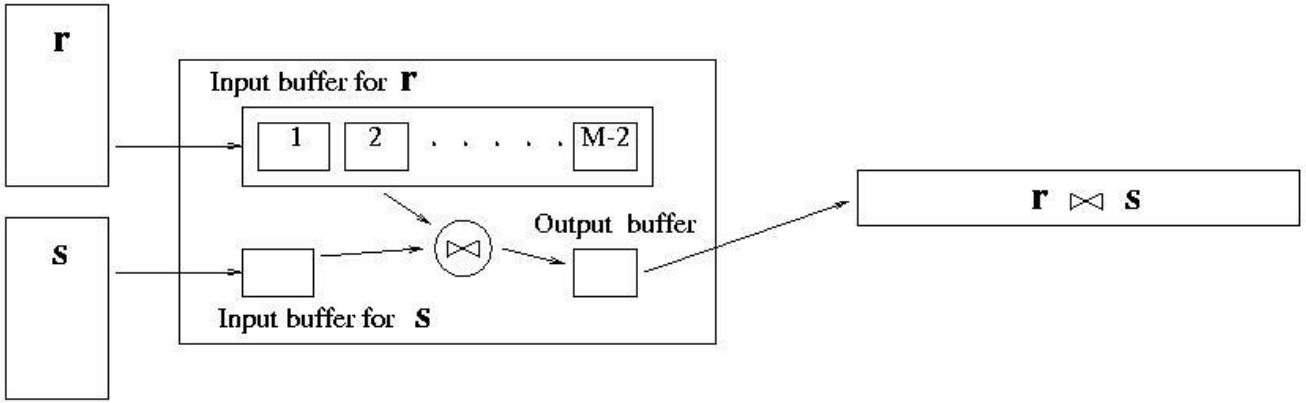
- Suppose R is outer

  Loop { Get the next (M-2) R blocks into memory

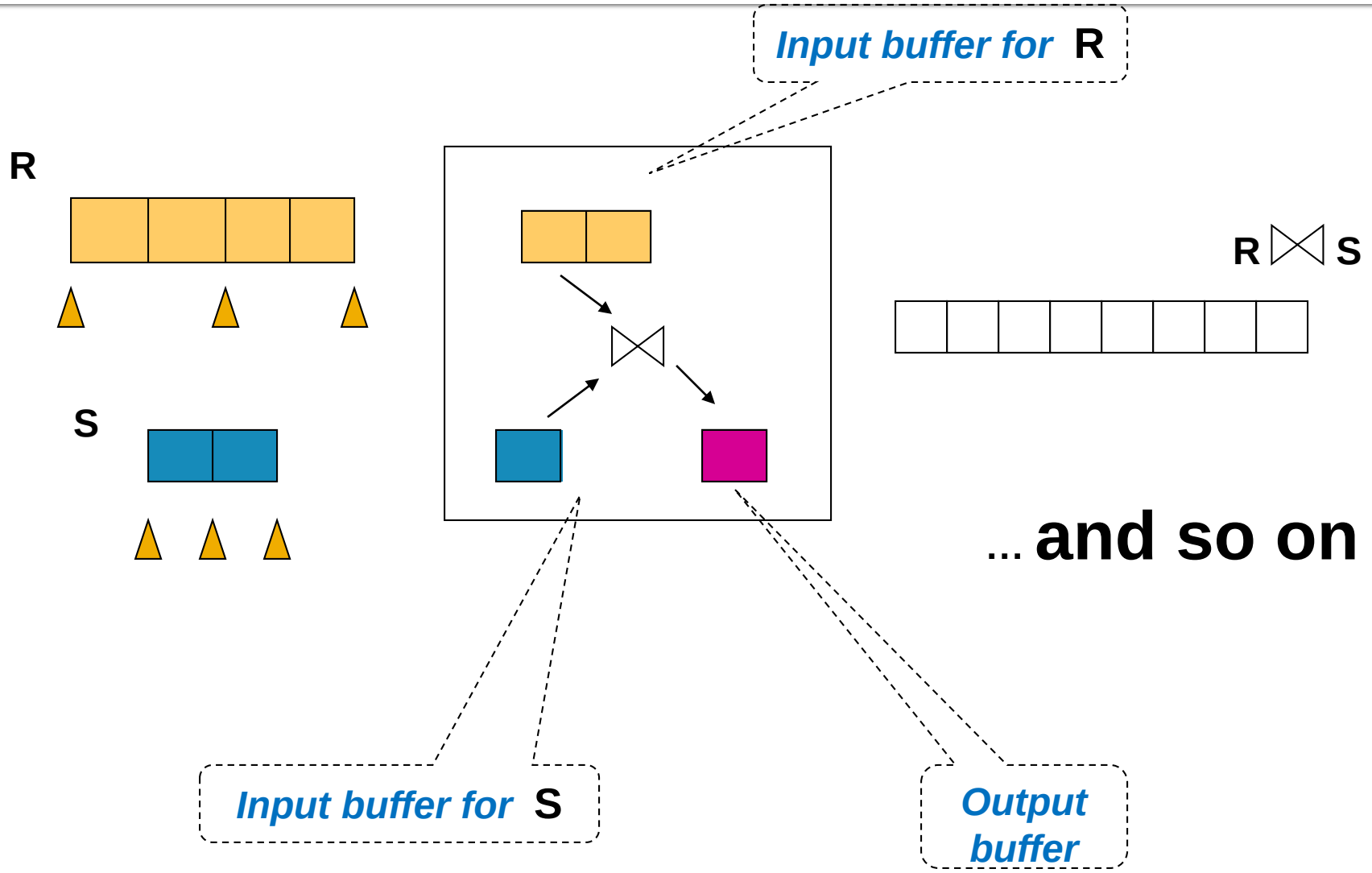
  Join these with each block of S }

- **Cost:** $\beta_r + (\beta_r/M\text{-}2) \times \beta_s$
- What if **S** is outer?
  - $\beta_s + (\beta_s/M\text{-}2) \times \beta_r$

Number of scans of relation S



Block-Nested Loop Illustrated



Index-Nested Loop Join $r \bowtie_{A=B} s$

- Use an index on **S** with search key B (instead of scanning **S**) to find rows of **S** that match t_r
  - Cost = $\beta_r + (T_r * \omega)$

Number of rows in r

avg cost of retrieving *all* rows in S that match t_r

  - Effective if number of rows of **S** that match tuples in **R** is small (i.e., ω is small) and index is clustered

foreach tuple t_r in R do {

***use index to find* all tuples t_s in s satisfying t_r.A=t_s.B;**

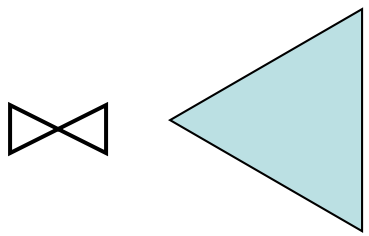
***output* (t_r, t_s)**

}

Indexed Nested Loop Join

R

B	C
a	10
a	20
b	10
d	30



C	D
10	cat
40	dog
15	bat
20	rat

S

- Indexed NLJ (conceptually)

for each r ∈ ***R*** **do**

scan index for matching tuples of ***S***

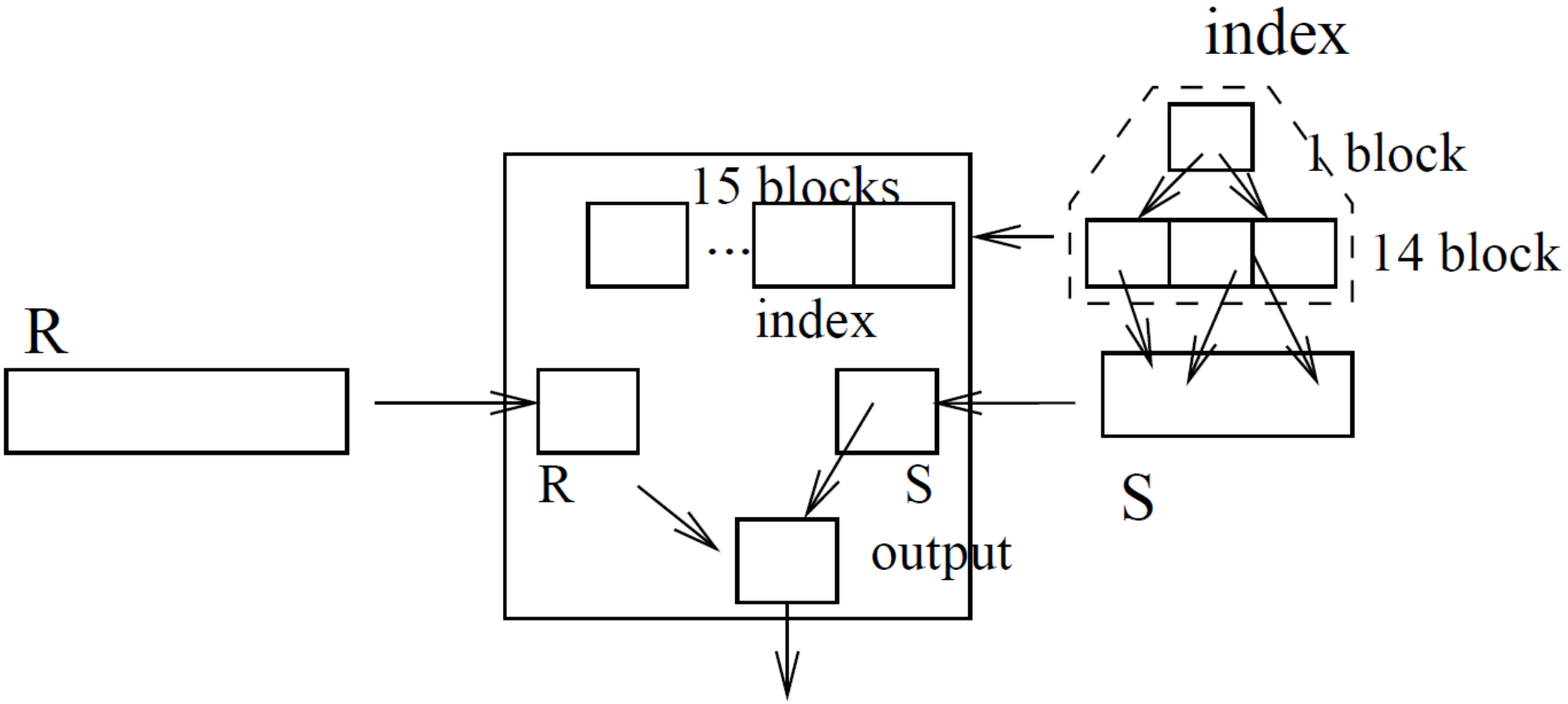
for each s ∈ matching tuples of ***S*** **do**

output r,s pair

Indexed Nested Loop Join Example

```
while (not customer.eof()) {
    Customer c= customer.read();
    Rental r[] =
    rental.readByAcctId(c.accountId);
    for (int i=0; i<r.length; i++) {
            result.write(c,r[i]);
    }
}
```

Indexed Nested Loop Join



Sort-Merge Join $R \bowtie S$

- Best when the *R* & *S* are both ordered on their join columns
- *R* & *S* are read in an alternating way:
  - You start reading the *R*, then you read *S* until you find a matching value or exceed the value
  - Append matching tuples to the result
  - If you pass the join value, you continue reading *R* again
- # of disk IOs during merge: $(B_R + B_S)$

Sort-Merge Example

Customers		
name	**street**	**cityID**
peter	minstreet	0
steve	macstreet	1
mike	longstreet	9
tim	unistreet	9
hans	msstreet	5
jens	shortstreet	1
frank	minstreet	0
olaf	macstreet	9
stefan	unistreet	0
alekh	unistreet	7
felix	macstreet	5
jorge	minstreet	9

Cities	
cityID	**city**
5	berlin
1	cuppertino
9	saarbruecken
3	paris
0	new york
7	london

Sort-Merge Example

R

Customers		
name	**street**	**cityID**
frank	minstreet	0
peter	minstreet	0
stefan	unistreet	0
jens	shortstreet	1
steve	macstreet	1
felix	macstreet	5
hans	msstreet	5
alekh	unistreet	7
jorge	minstreet	9
mike	longstreet	9
olaf	macstreet	9
tim	unistreet	9

PR

PS

S

Cities	
cityID	**city**
0	new york
1	cuppertino
3	paris
5	berlin
7	london
9	saarbruecken

Customers NATURAL JOIN Cities			
name	**street**	**cityID**	**city**

Sort-Merge Example

Customers		
name	**street**	**cityID**
frank	minstreet	0
peter	minstreet	0
stefan	unistreet	0
jens	shortstreet	1
steve	macstreet	1
felix	macstreet	5
hans	msstreet	5
alekh	unistreet	7
jorge	minstreet	9
mike	longstreet	9
olaf	macstreet	9
tim	unistreet	9

Cities	
cityID	**city**
0	new york
1	cuppertino
3	paris
5	berlin
7	london
9	saarbruecken

- **Values pointed to are equal => Found join result**
- **Move left pointer…**

Customers NATURAL JOIN Cities			
name	**street**	**cityID**	**city**
frank	minstreet	0	new york

Sort-Merge Example

Customers		
name	**street**	**cityID**
frank	minstreet	0
peter	minstreet	0
stefan	unistreet	0
jens	shortstreet	1
steve	macstreet	1
felix	macstreet	5
hans	msstreet	5
alekh	unistreet	7
jorge	minstreet	9
mike	longstreet	9
olaf	macstreet	9
tim	unistreet	9

Cities_Dictionary	
cityID	**city**
0	new york
1	cuppertino
3	paris
5	berlin
7	london
9	saarbruecken

- **Values pointed to are equal => Found join result**
- **Continue moving the left pointer…**

Customers NATURAL JOIN Cities_Dictionary			
name	**street**	**cityID**	**city**
frank	minstreet	0	new york
peter	minstreet	0	new york
stefan	unistreet	0	new york

Sort-Merge Example

Customers		
name	**street**	**cityID**
frank	minstreet	0
peter	minstreet	0
stefan	unistreet	0
jens	shortstreet	1
steve	macstreet	1
felix	macstreet	5
hans	msstreet	5
alekh	unistreet	7
jorge	minstreet	9
mike	longstreet	9
olaf	macstreet	9
tim	unistreet	9

Cities	
cityID	**city**
0	new york
1	cuppertino
3	paris
5	berlin
7	london
9	saarbruecken

- **Values pointed to are NOT equal => Move the pointer pointing to the smaller value**

Customers NATURAL JOIN Cities			
name	**street**	**cityID**	**city**
frank	minstreet	0	new york
peter	minstreet	0	new york
stefan	unistreet	0	new york

Sort-Merge Example

Customers		
name	**street**	**cityID**
frank	minstreet	0
peter	minstreet	0
stefan	unistreet	0
jens	shortstreet	1
steve	macstreet	1
felix	macstreet	5
hans	msstreet	5
alekh	unistreet	7
jorge	minstreet	9
mike	longstreet	9
olaf	macstreet	9
tim	unistreet	9

Cities	
cityID	**city**
0	new york
1	cuppertino
3	paris
5	berlin
7	london
9	saarbruecken

- **Values pointed to are equal => Found join result**
- **Continue moving the left pointer…**

Customers NATURAL JOIN Cities			
name	**street**	**cityID**	**city**
frank	minstreet	0	new york
peter	minstreet	0	new york
stefan	unistreet	0	new york
jens	shortstreet	1	cuppertino

Algorithm

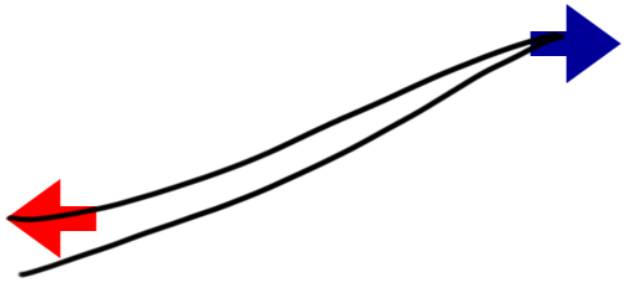
```
JP(r,s) := r.x == s.x                          //definition of the join predicate

SortMergeJoin( R, S, JP(r,s) ):
    sort(R on R.x);                            //sort R on join attribute x
    sort(S on S.x);                            //sort S on join attribute x
    Pointer PR = R[0];                         //initialize pointer to R
    Pointer PS = S[0];                         //initialize pointer to S
    do:                                        //start loop
        if PR.x == PS.x:                       //if values match
            output( (PR, PS) );                //output join result
        if PR.x <= PS.x:                       //find smaller join key (move left first!)
            PR++;                              //move pointer to R forward
        else:
            PS++;                              //move pointer to S forward
    while PR != R.end && PS != S.end           //both tables unprocessed
     ...                                       //output join results...
        ...                                    //...for non-empty table
```

Duplicates in both Columns =>

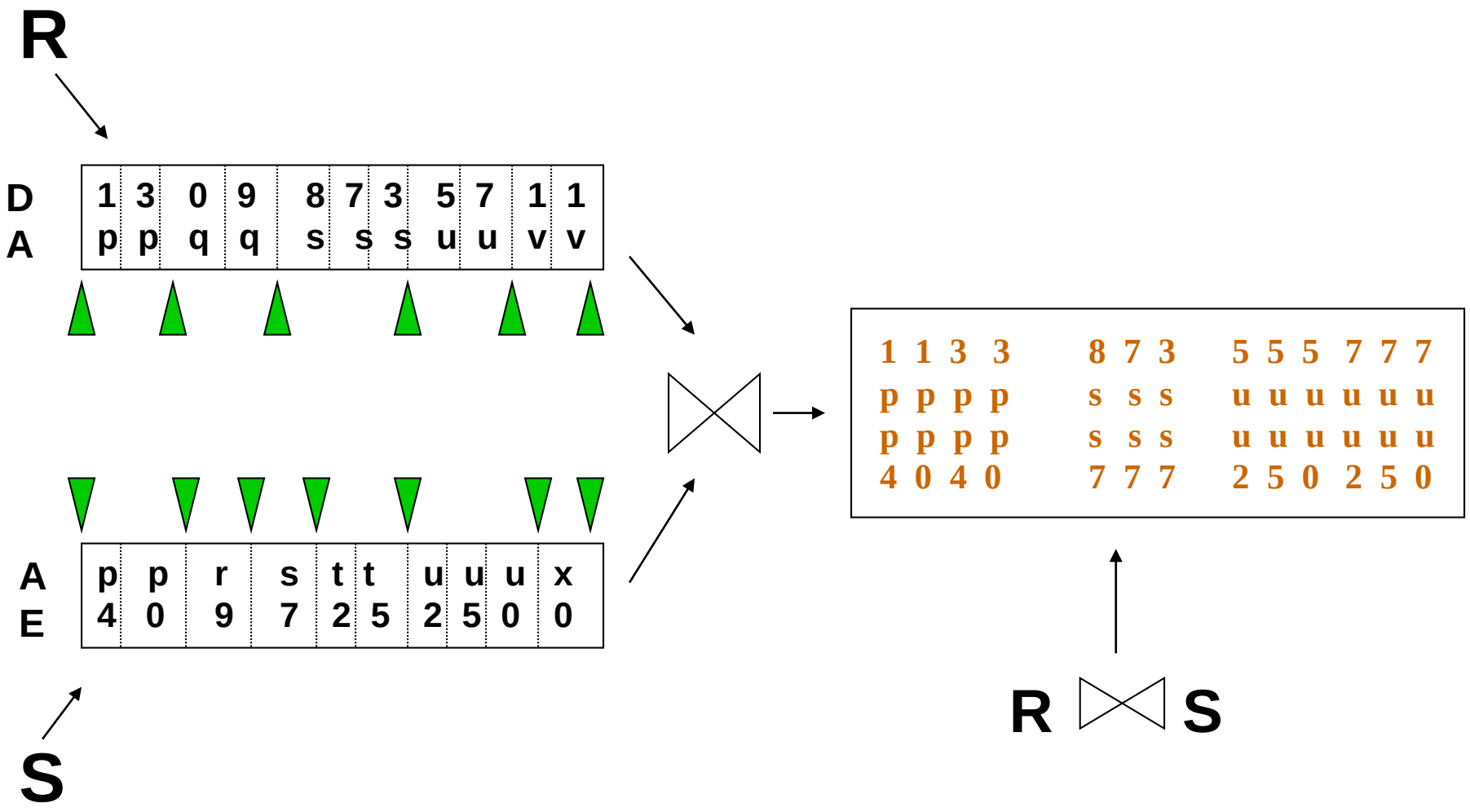
=> Use Co-Grouping then Cross-Join co-groups

Customers		
name	**street**	**cityID**
frank	minstreet	0
peter	minstreet	0
stefan	unistreet	0
jens	shortstreet	1
steve	macstreet	1
felix	macstreet	5
hans	msstreet	5
alekh	unistreet	7
jorge	minstreet	9
mike	longstreet	9
olaf	macstreet	9
tim	unistreet	9



Cities_Dictionary	
cityNo	city
1	new york
1	cuppertino
1	paris
5	berlin
5	london
9	saarbruecken

Co-Grouping then Cross-Join co-groups illustrated



Total Cost for Sort-Merge Join

$$2b_R(\lceil \log_{M-1}(b_R/M) \rceil + 1) + 2b_S(\lceil \log_{M-1}(b_S/M) \rceil + 1) + (b_R + b_S)$$

Sort Cost

Merge Cost

Hash-Join $R \bowtie_{A=B} S$

- Hash function h(v), range $1 \rightarrow k$

(1) Hashing stage (bucketizing): hash tuples into buckets

- Hash R tuples into buckets $R_1, R_2, \ldots R_k$
- Hash S tuples into buckets $S_1, S_2, \ldots S_k$

(1) Join stage: join tuples in matching buckets

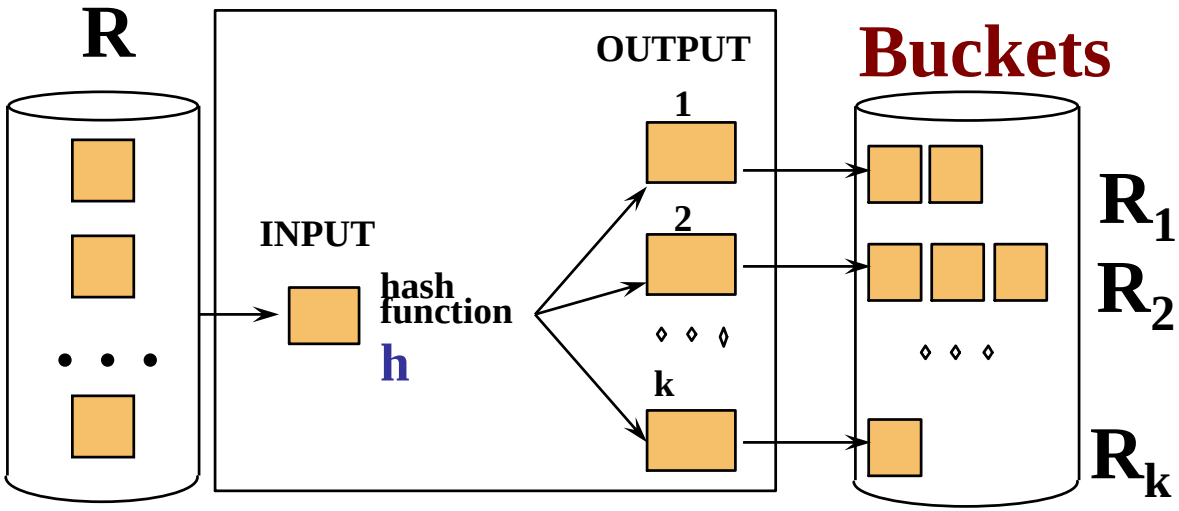
For i = 1 to k do

Match tuples in R_i, S_i buckets

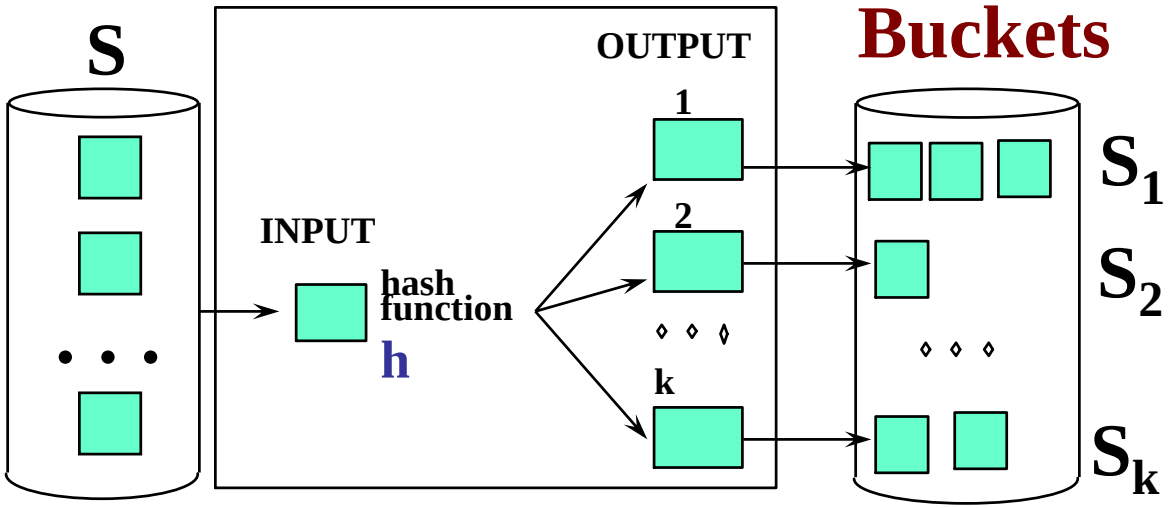
- Cost = $3\,(B_R + B_S)$

Hashing Stage

- Partition tuples in R and S using join attributes as key hash



- Tuples in partition R_i only match tuples in partition S_i.

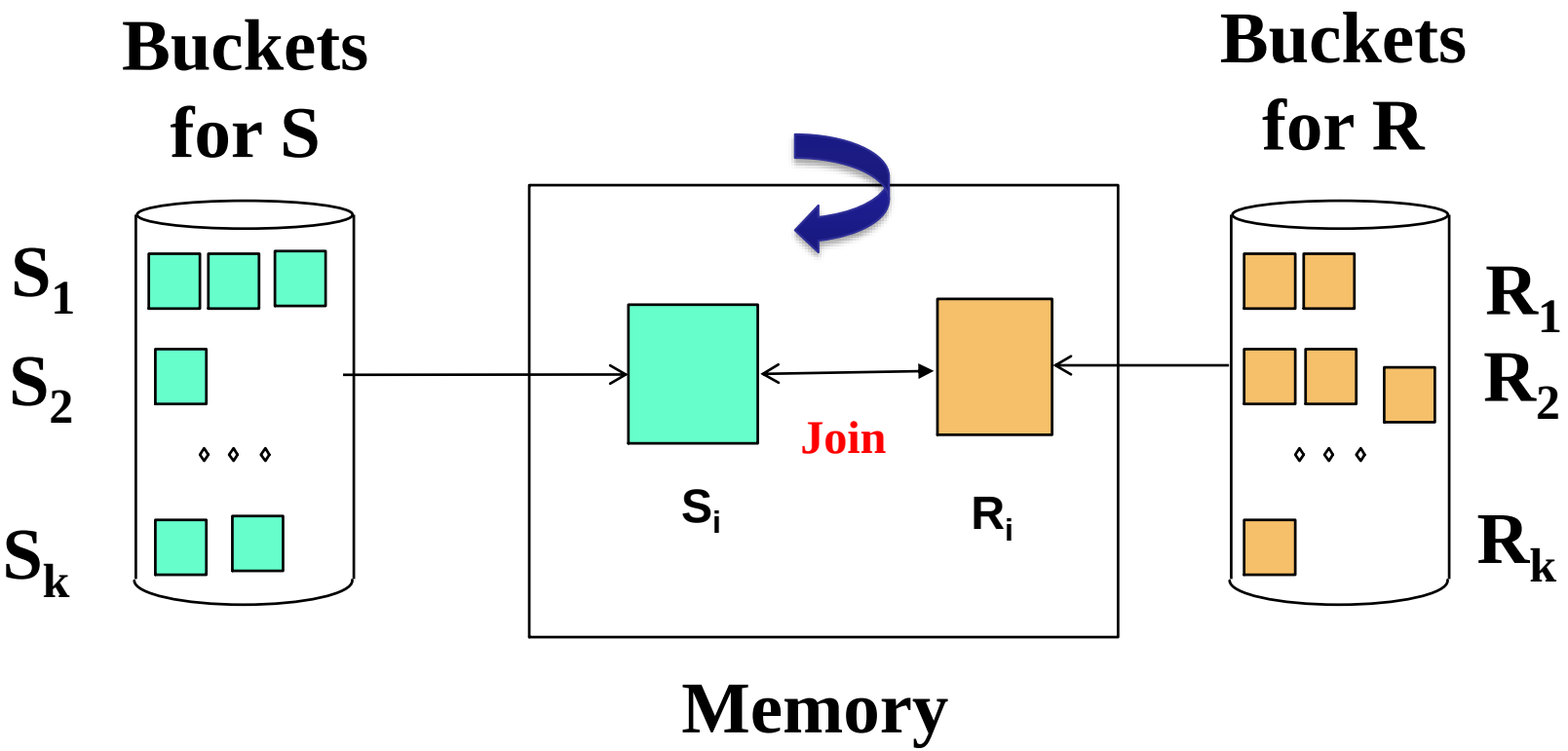


Join stage

Join tuples in matching buckets

For i = 1 to k do

Match tuples in R_i, S_i buckets



Comparing Join Algorithms

- **Nested Loop Join (brute force)**

Efficient if either R or S fits in main memory.

- **Index Join**

Suitable only if one of the table has an index on the join attributes.

- **Sort-Merge Join**

Good for non-equi-join (e.g., R.C > S.C). Good choice when both tables are very large

  - Less sensitive to data skew, very efficient given a sorted index, result is sorted

- **Hash Join**

Usually best for equi-join, where relations not sorted and no indexes exist. Requires less memory, highly parallelizable

Estimating Intermediate Result Sizes

What algorithm to be use for query processing depends very strongly on the sizes of the relations

The number of tuples after selection

$S = \sigma_{A=c}(R)$

- $0 <= T(S) <= T(R)$
- Expected value: $T(S) = T(R)/V(R,A)$

A good guess, though never true in practice. Does not require storing many statistics.

$S = \sigma_{A<c}(R)$

- $T(S)$ can be anything from 0 to $T(R)$
- Heuristic: $T(S) = T(R)/3$

Of course one can do better!

Histograms tell you how many tuples have R.A values within a certain range

- Maintained by the RDBMS
- Makes size estimation much more accurate (hence, cost estimations are more accurate)

Employee(ssn, name, salary, phone)

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

The number of tuples after a join

$R \bowtie_A S$

- When the set of A values are disjoint, then $T(R \bowtie_A S) = 0$
- When A is a key in S and a foreign key in R, then $T(R \bowtie_A S) = T(R)$
- When A is a key in both R and S, then

$T(R \bowtie_A S) = \min(T(R), T(S))$

Otherwise…

$T(R \bowtie_A S) = T(R)\,T(S) / \max(V(R,A),V(S,A))$

Join Size Estimation $R \bowtie_{A=B} S$

$$\frac{T(R) \times T(S)}{\max(V(R,A), V(S,B))}$$

- **Parameters:**

T(R) = # tuples in R

T(S) = # tuples in S

V(R, A) = # of distinct values of attribute A in R

V(R, B) = # of distinct values of attribute B in S

Example of estimating the number of tuples after a join

T(R) = 10,000 T(S) = 20,000

V(R,A) = 100 V(S,A) = 200

How large is $R \bowtie_A S$?

Answer: $T(R \bowtie_A S) = 10000 * 20000/200 = 1M$

The expected number of tuples after a join on multiple attributes is:

$$T(R \bowtie_{A,B} S) = \frac{T(R) * T(S)}{[\max(V(R,A),V(S,A)) * \max(V(R,B),V(S,B))]}$$

Summary

- Query optimization (QO) is an important task in a relational DBMS
- Understanding QO is necessary to write better queries that optimize well, and to debug plan-related performance problems
- QO has 2 parts
  - Enumeration of alternative plans
  - Estimation of cost of enumerated plans

  + Select the cheapest plan
- Key issues – query trees, operator implementation, use of indexes