Stony Brook University

CSE 305 / CSE532

Lecture 19 (Chapter 10)

Query Processing: The Basics

Lecturer: Sael Lee

Slide adapted from the
author’s, Peter Bailis’s and Dr. Ilchul Yoon’s slides.

SUNY Korea

Query Processing Example

Select B,D

From R,S

Where R.A = “c” $\wedge$ S.E = 2 $\wedge$ R.C=S.C

Example cont.

R

A	B	C
a	1	10
b	1	20
c	2	10
d	2	35
e	3	45

S

C	D	E
10	x	2
20	y	2
30	z	2
40	x	1
50	y	3

Example cont.

R

A	B	C
a	1	10
b	1	20
c	2	10
d	2	35
e	3	45

S

C	D	E
10	x	2
20	y	2
30	z	2
40	x	1
50	y	3

Answer

B	D
2	x

How do we execute query?

One idea

- Do Cartesian product
- Select tuples
- Do projection

RXS

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
.					
.					

Peter Bailis’s slides

RXS

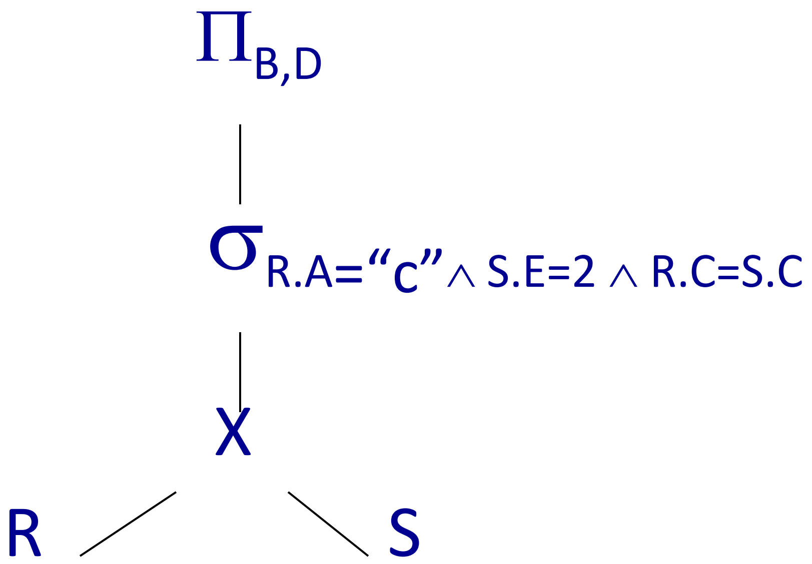
R.A	R.B	R.C	S.C	S.D	S.E
a	1	10	10	x	2
a	1	10	20	y	2
•					
•					
C	2	10	10	x	2
•					
•					

Bingo! →

Got one...

Relational Algebra - can be used to describe plans...

Plan I



$\Pi_{B,D}$

$\sigma_{R.A=\text{“c”} \wedge S.E=2 \wedge R.C=S.C}$

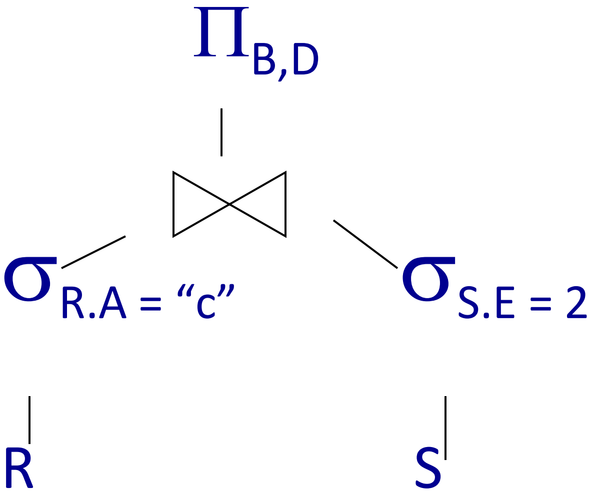
X

R S

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

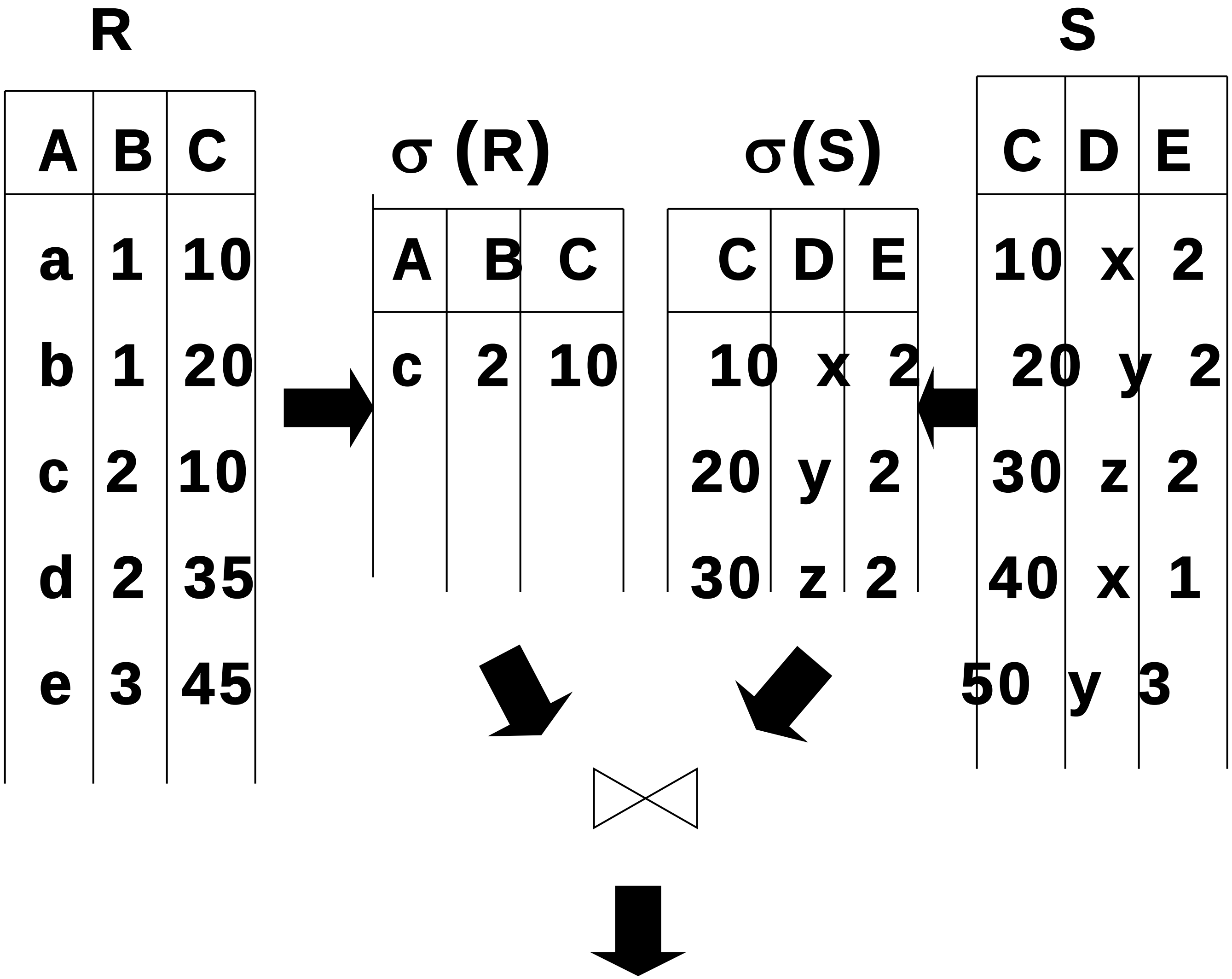
Another idea:

Plan II



$\bowtie$

natural join



R

A	B	C
a	1	10
b	1	20
c	2	10
d	2	35
e	3	45

σ (R)

A	B	C
c	2	10

σ(S)

C	D	E
10	x	2
20	y	2
30	z	2

S

C	D	E
10	x	2
20	y	2
30	z	2
40	x	1
50	y	3

$\bowtie$

Plan III: Utilizing Index

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

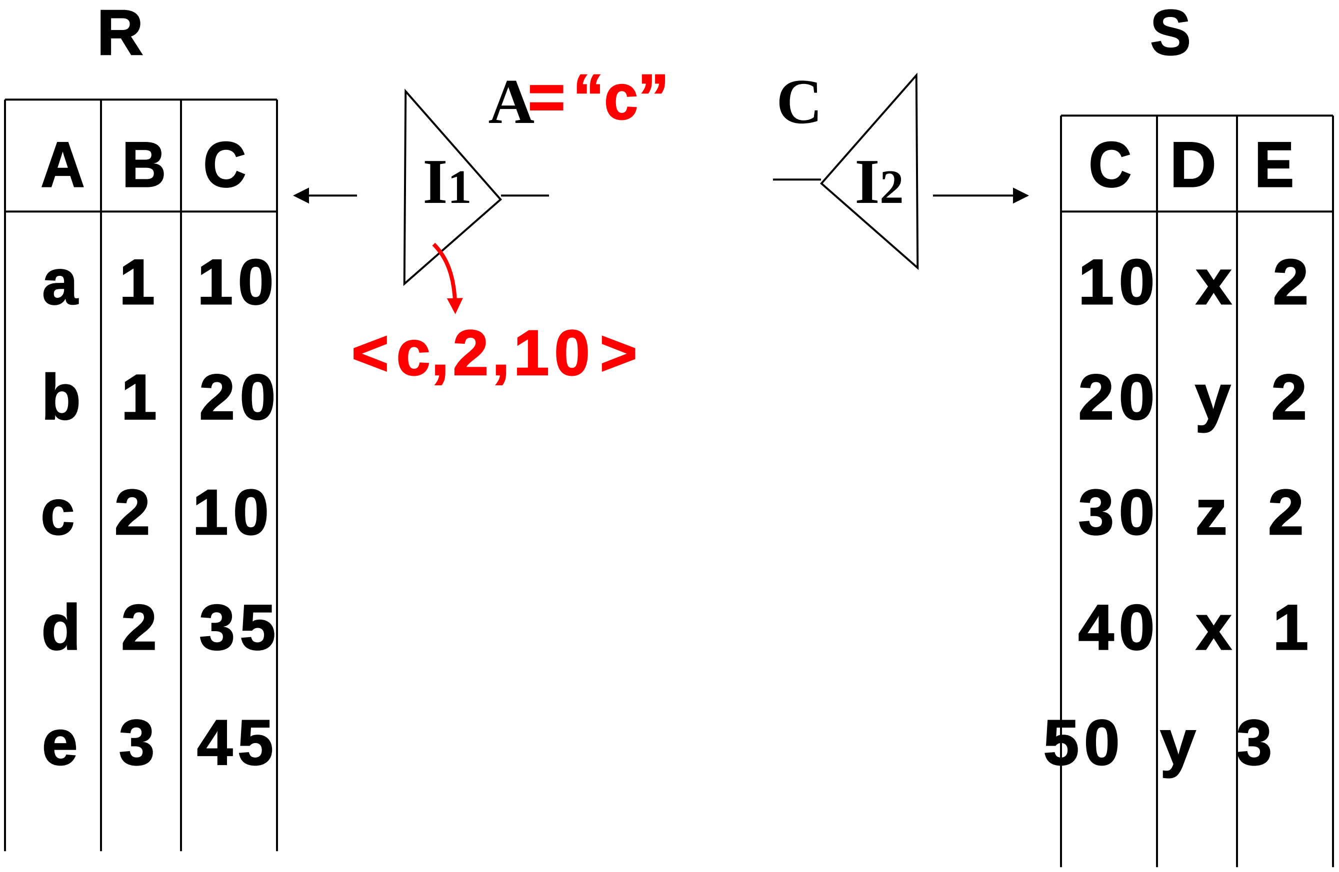
(4) Join matching R,S tuples, project B,D attributes and place in result

R

A	B	C
a	1	10
b	1	20
c	2	10
d	2	35
e	3	45

S

C	D	E
10	x	2
20	y	2
30	z	2
40	x	1
50	y	3



R

A	B	C
a	1	10
b	1	20
c	2	10
d	2	35
e	3	45

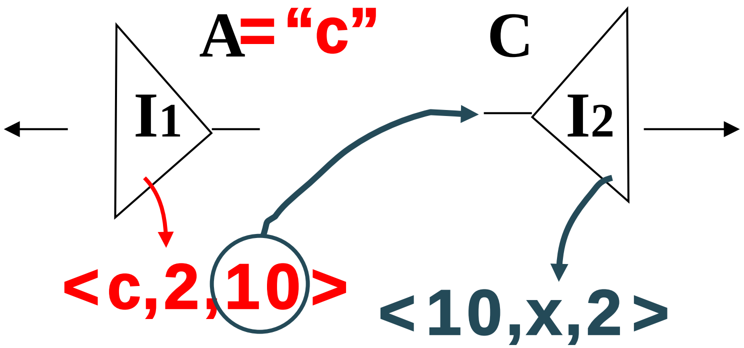
S

C	D	E
10	x	2
20	y	2
30	z	2
40	x	1
50	y	3

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

R

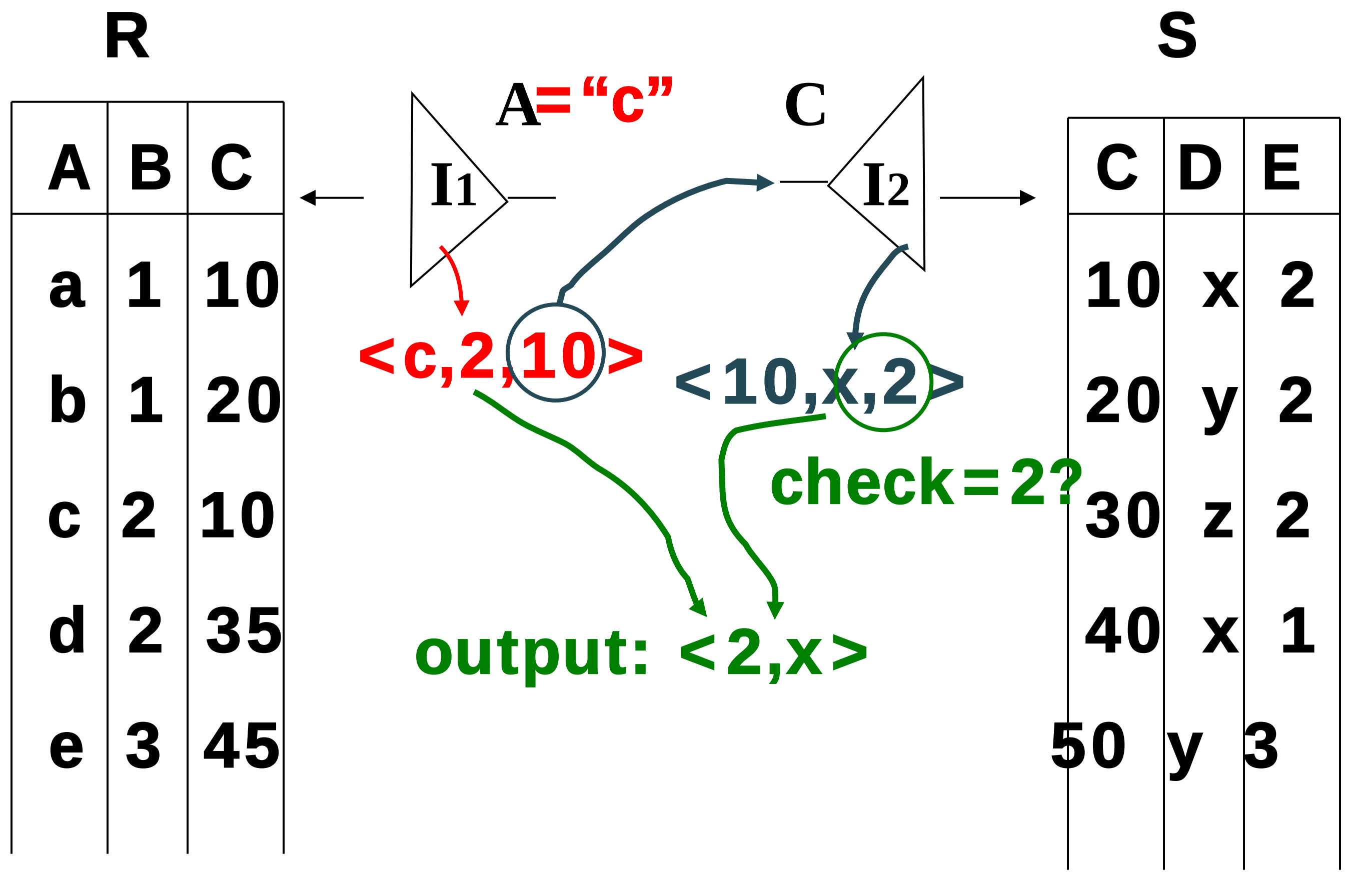
A	B	C
a	1	10
b	1	20
c	2	10
d	2	35
e	3	45



S

C	D	E
10	x	2
20	y	2
30	z	2
40	x	1
50	y	3

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



R

A	B	C
a	1	10
b	1	20
c	2	10
d	2	35
e	3	45

A="c"

I1

<c,2,10>

C

I2

<10,x,2>

check=2?

output: <2,x>

S

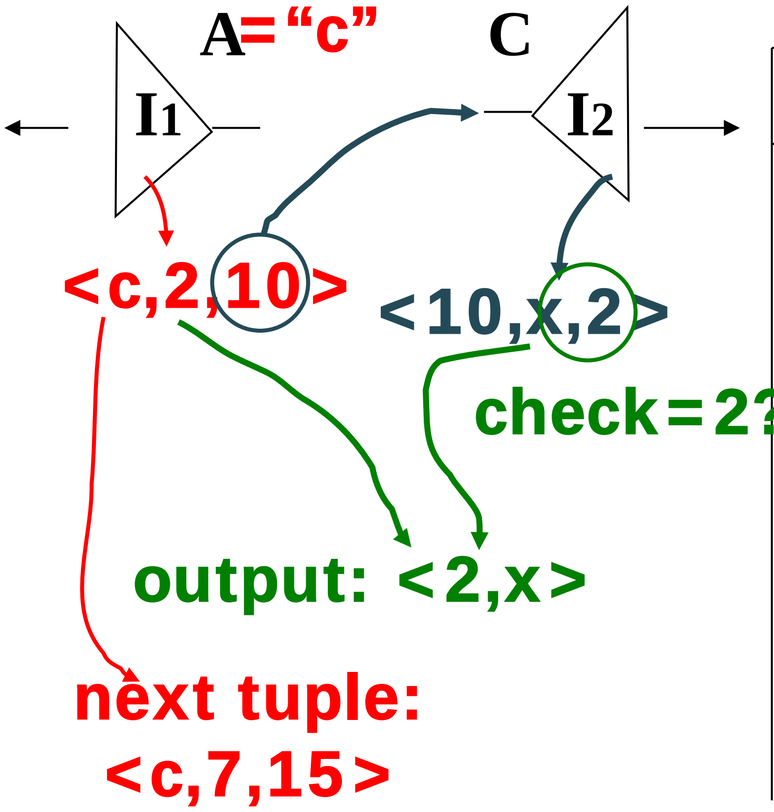
C	D	E
10	x	2
20	y	2
30	z	2
40	x	1
50	y	3

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

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

R

A	B	C
a	1	10
b	1	20
c	2	10
d	2	35
e	3	45



A= “c”

C

I1

I2

<c,2,10>

<10,x,2>

check=2?

output: <2,x>

next tuple:
<c,7,15>

S

C	D	E
10	x	2
20	y	2
30	z	2
40	x	1
50	y	3

External Sorting

- Sorting is used in implementing many relational operations

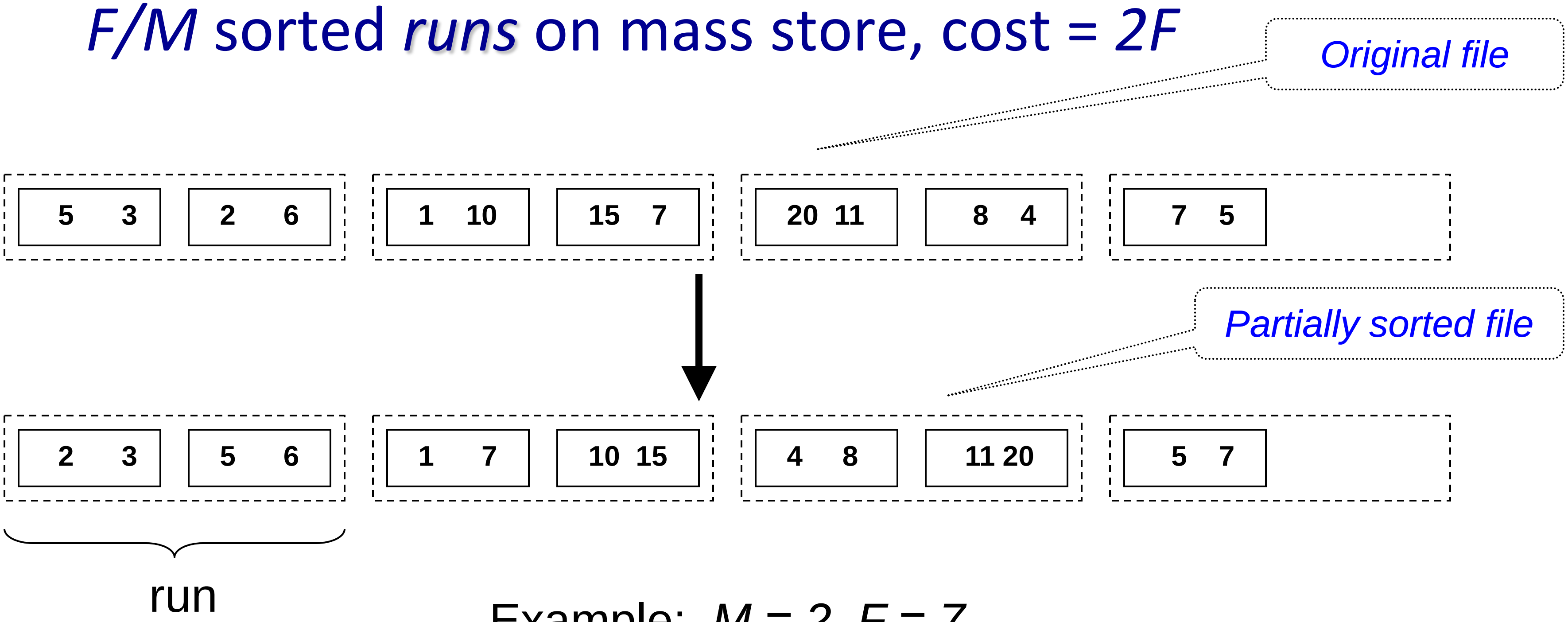
- Problem:
  - Relations are typically large, do not fit in main memory
  - So cannot use traditional in-memory sorting algorithms
- Approach used:
  - Combine in-memory sorting with clever techniques aimed at minimizing I/O
  - I/O costs dominate => cost of sorting algorithm is measured in the number of page transfers

External Sorting (cont’d)

- External sorting has two main components:
  - Computation involved in sorting records in buffers in main memory
  - I/O necessary to move records between mass store and main memory

Simple Sort Algorithm

- *M* = number of main memory page buffers
- *F* = number of pages in file to be sorted
- Typical algorithm has **two phases**:
  - **1 Partial sort phase**: sort *M* pages at a time; create *F/M* sorted *runs* on mass store, cost = *2F*



Original file

5 3 | 2 6 || 1 10 | 15 7 || 20 11 | 8 4 || 7 5

Partially sorted file

2 3 | 5 6 || 1 7 | 10 15 || 4 8 | 11 20 || 5 7

run

Example: *M = 2, F = 7*

Simple Sort Algorithm

- **2 Merge Phase**: merge all runs into a single run using M-1 buffers for input and 1 output buffer
  - Merge step: divide runs into groups of size *M-1* and merge each group into a run; cost = *2F*
    - *Each step reduces number of runs by a factor of M-1*

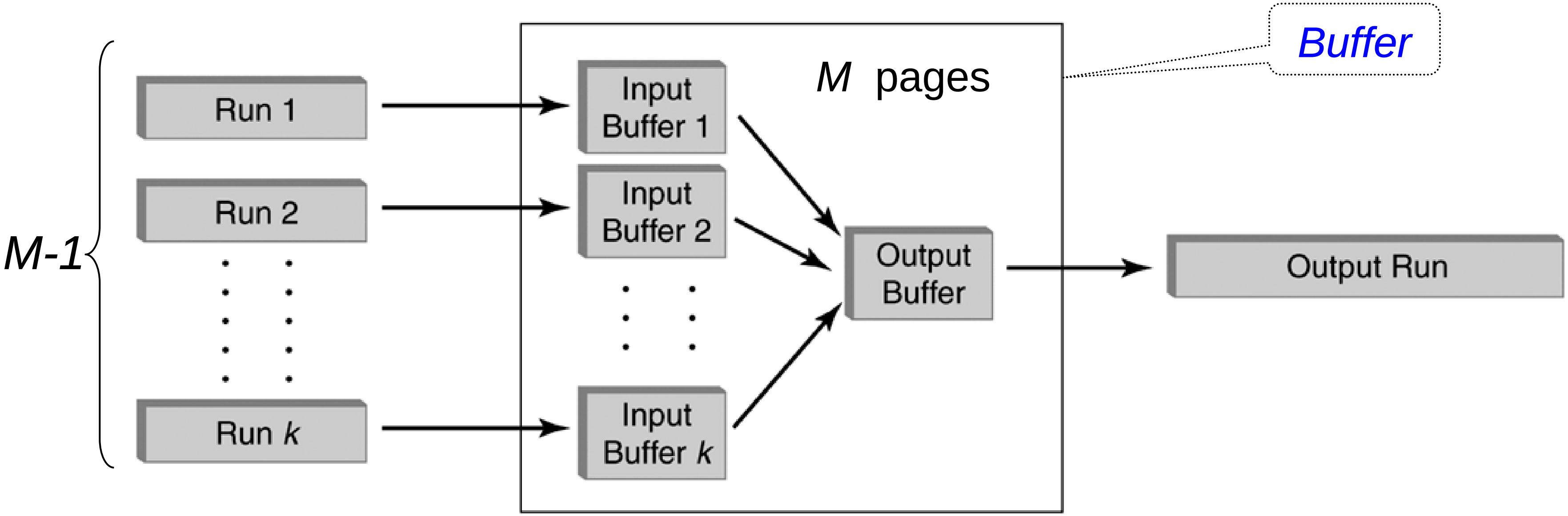
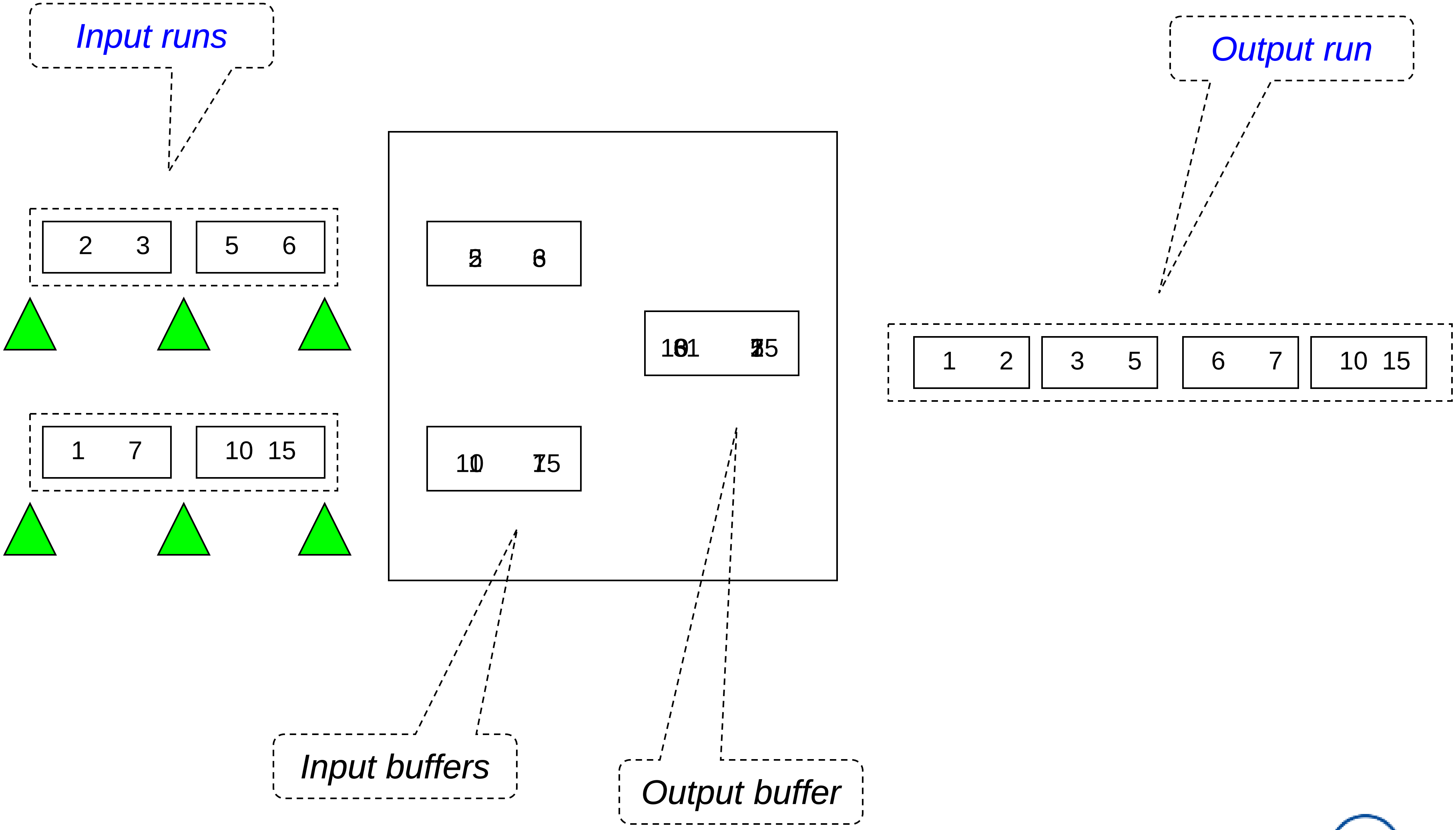


FIGURE 10.2 *k*-way merge.

Merge: An Example

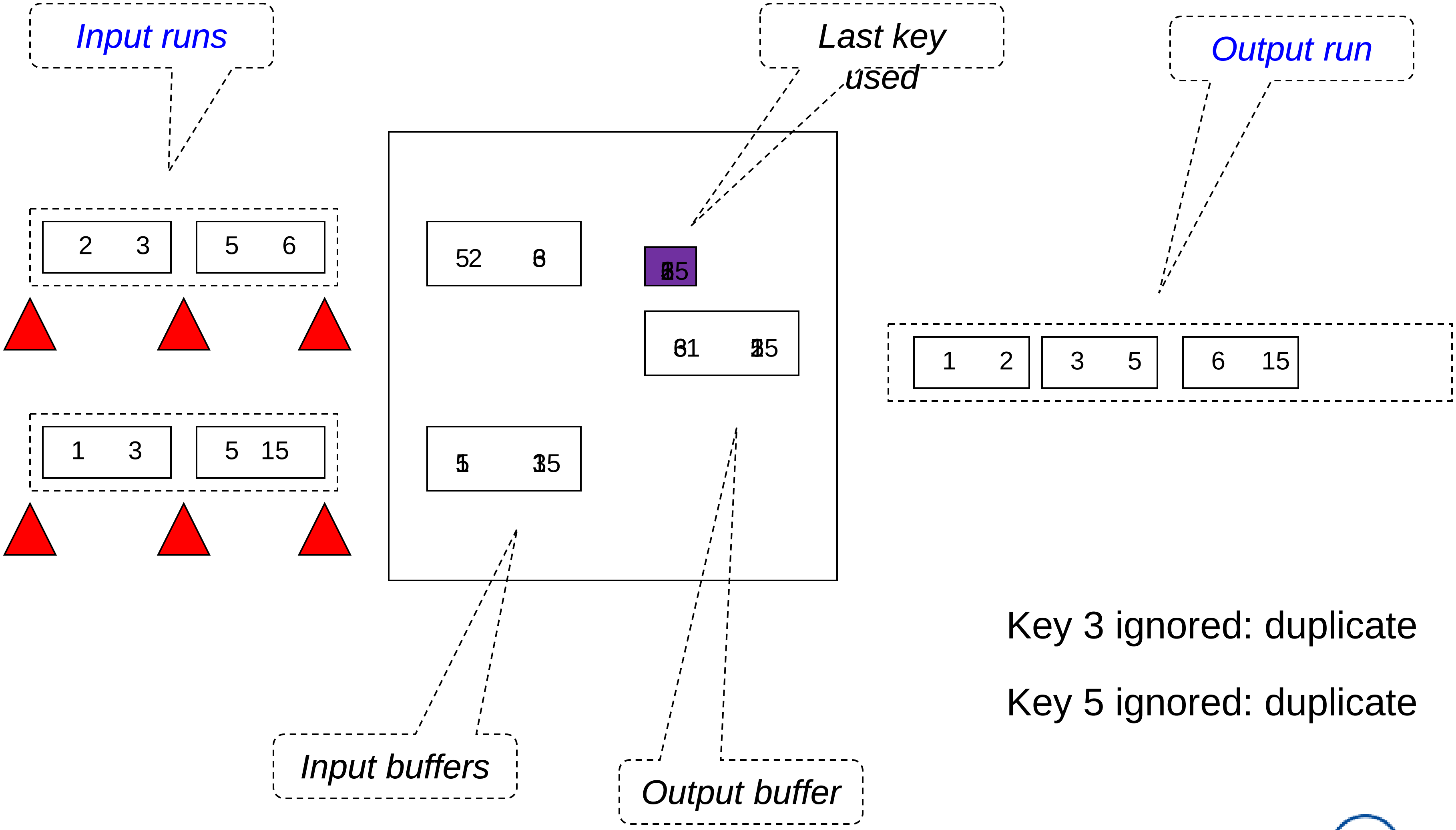


Duplicate Elimination

- A major step in computing *projection*, *union*, and *difference* relational operators

- Algorithm:
  - Sort
  - At the last stage of 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

Sort-Based Projection

- Algorithm:
  - Sort rows of relation at cost of $2F \text{ Log}_{M-1} F$
  - Eliminate unwanted columns in partial sort phase (no additional cost)
  - Eliminate duplicates on completion of last merge step (no additional cost)
- Cost: the cost of sorting

Hash-Based Projection

- *Phase* 1:
  - Input rows
  - Project out columns
  - Hash remaining columns using a hash function with range *1…M-1* creating *M-1* buckets on disk
  - **Cost** = *2F*
- *Phase* 2:
  - Sort each bucket to eliminate duplicates
  - **Cost** (assuming a bucket fits in *M-1* buffer pages) = *2F*
- **Total cost** = *4F*

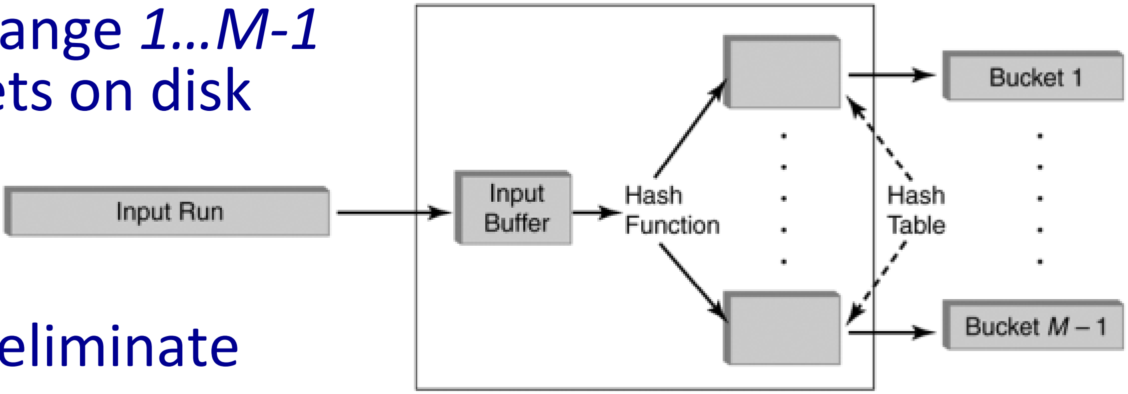


FIGURE 10.5 Hashing input relation into buckets.

Comparison

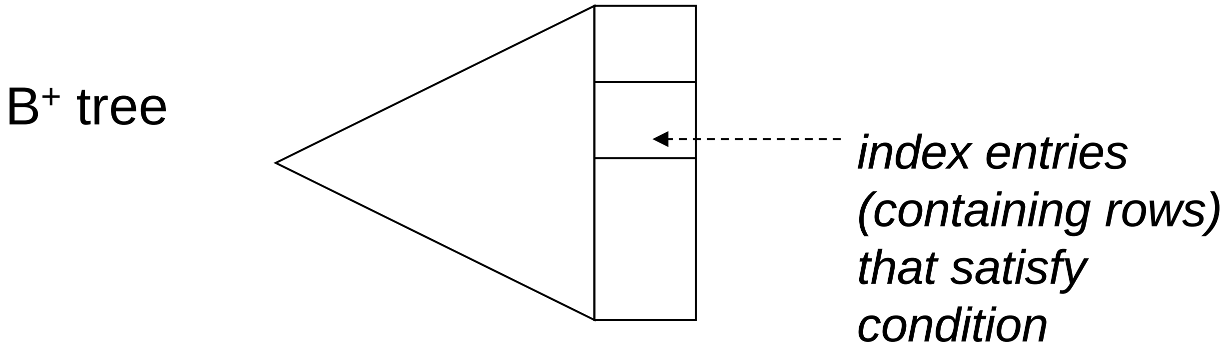
- Assume
  - M=10000-page buffer (40MB) ← use as hash table
  - We have F=10^8-page file to process (400GB = 40M*10000)
- Hash-based projection
  - $4*10^8$
- Sort-based projection
  - $2Flog_{(M-1)}F = 2 \times 10^8 \times log_{10^4-1}10^8 \geq 4 \times 10^8$
- However, it requires
  - Even distribution from hash function
  - In-memory sort of each bucket

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

- No index on *attr*:
    - If rows are not sorted on *attr:*
        - Scan all data pages to find rows satisfying selection condition
        - Cost = F
    - If rows are sorted on *attr* and op is =, >, < then:
        - Use *binary search* (at $\log_2 F$) to locate first data page containing row in which (*attr* = *value*)
        - Scan further to get all rows satisfying (*attr op value*)
        - Cost = $\log_2 F$ + (cost of scan)

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

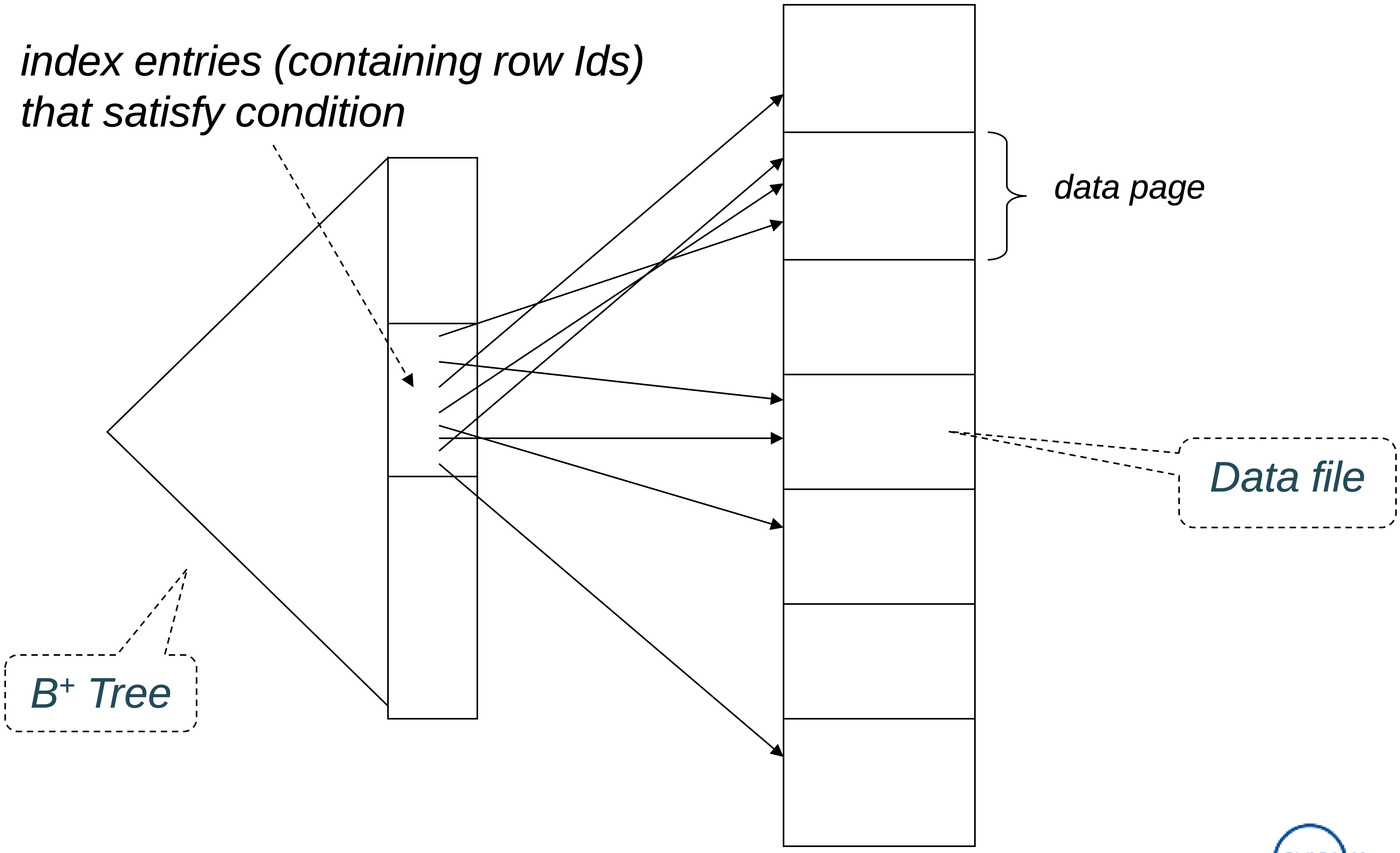
- Clustered B^+ tree index on *attr* (for “=” or range search):
  - Locate first index entry corresponding to a row in which (*attr* = *value*).
    - Cost = depth of tree
  - Rows satisfying condition packed in sequence in successive data pages; *scan those pages.*
    - Cost: number of pages occupied by qualifying rows



Computing Selection σ(attr op value)

- 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 pages
    - Scan entries and sort record Ids to identify table data pages with qualifying rows; Any page that has at least one such row must be fetched once.
    - **Cost** = number of rows that satisfy selection condition

Unclustered B^+ Tree Index



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

- Hash index on *attr* (for "=" search only):
  - Hash on *value*. Cost (of finding the right bucket) ≈ 1.2
    - 1.2 – typical average cost of hashing (> 1 due to possible overflow chains)

    - Finds first the (unique) bucket containing all index entries satisfying selection condition. Then,
    - Clustered index – all qualifying rows packed in the bucket (a few pages)

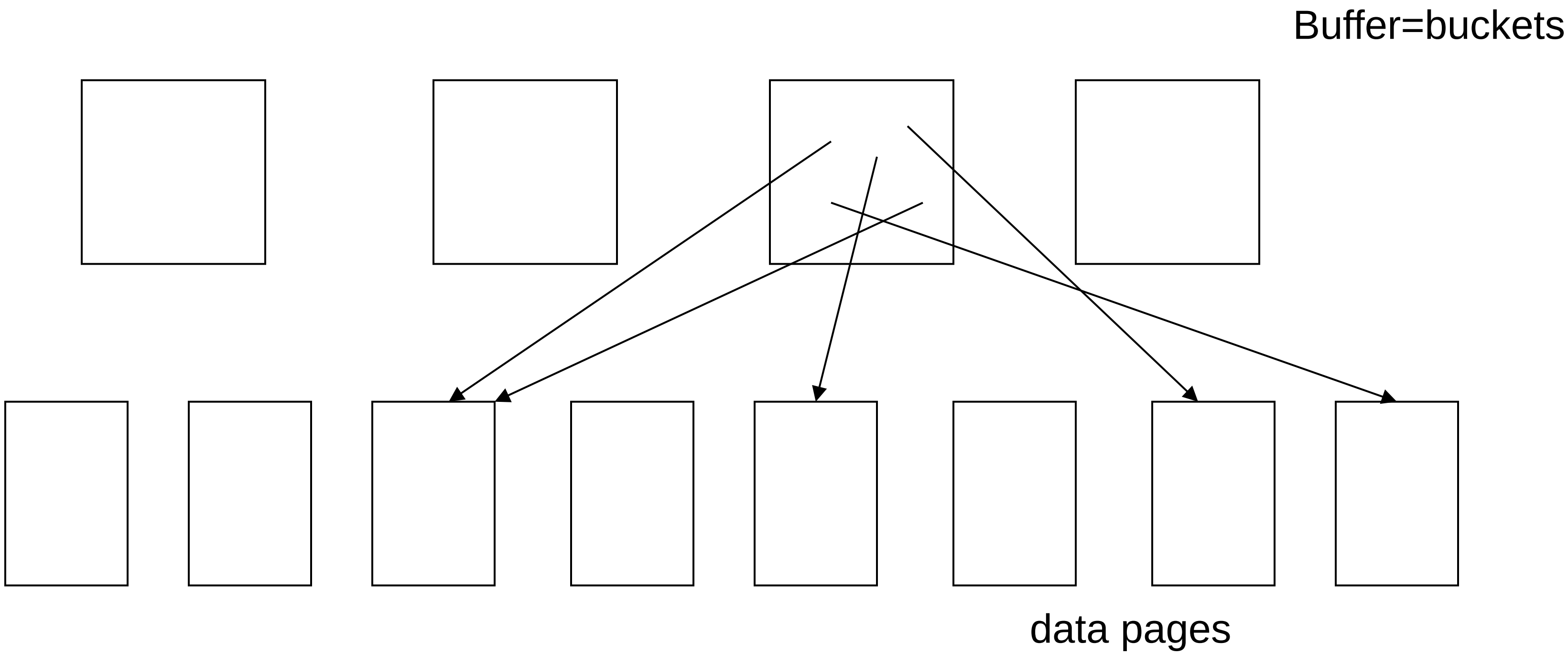
      Cost: number of pages occupies by the bucket
    - Unclustered index – sort row Ids in the index entries to identify data pages with qualifying rows

      Each page containing at least one such row must be fetched once

      Cost: min(*number of qualifying rows in bucket, number of pages in file*)

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

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



Access Path

- *Access path* is the notion that denotes *algorithm + data structure* used to locate rows satisfying some condition

- *Examples*:
  - *File scan*: can be used for any condition
  - *Hash*: equality search; *all* search key attributes of hash index are specified in condition
  - *B^+ tree*: equality *or* range search; a *prefix* of the search key attributes are specified in condition
    - B^+ tree supports a variety of access paths
  - *Binary search*: relation sorted on a sequence of attributes and some *prefix* of that sequence is specified in condition

Access Paths Supported by B^+ tree

- ***Example***: Given a B^+ tree whose search key is the sequence of attributes *a2, a1, a3, a4*
  - Access path for search $\sigma_{a1>5 \text{ AND } a2=3 \text{ AND } a3=\text{'x'}}(R)$:
    - find first entry having $a2=3$ AND $a1>5$ AND $a3=\text{'x'}$ and scan leaves from there until entry having $a2>3$ or $a3 \neq \text{'x'}$. Select satisfying entries
  - Access path for search $\sigma_{a2=3 \text{ AND } a3 > \text{'x'}}(R)$:
    - locate first entry having $a2=3$ and scan leaves until entry having $a2>3$. Select satisfying entries
  - Access path for search $\sigma_{a1>5 \text{ AND } a3 = \text{'x'}}(R)$:
    - Scan of *R*

Choosing an Access Path

- *Selectivity* of an access path = number of pages retrieved using that path
  - If several access paths support a query, DBMS chooses the one with *lowest* selectivity
  - Size of domain of attribute is an indicator of the selectivity of search conditions that involve that attribute

- Example: $\sigma_{\mathit{CrsCode}=\text{'CS305'}\ \text{AND}\ \mathit{Grade}=\text{'B'}}$ (Transcript)
  - Assume that we have *two* B^+ *trees*; one with search key *CrsCode*, and the other with *Grade*
  - a B^+ tree with search key *CrsCode* has lower selectivity than a B^+ tree with search key *Grade*

Selections with Complex Conditions

- Selection with conjunctive conditions
  - Use the most selective access path to retrieve the corresponding tuples
    - e.g., one condition is for an indexed attribute
  - Use several access paths that cover the expression
    - e.g., use the most selective first, and use the other ones.
- Selection with disjunctive conditions
  - If the condition contain disjunctions, convert to <u>disjunctive normal form</u>. (disjunction of conjunctive conditions)
  - Check available access paths for the individual disjuncts and choose the appropriate strategy
    - e.g., what if a disjunct need file scan?
    - e.g., what if each disjunct has better access path than file scan?

Computing Joins

- The cost of joining two relations makes the choice of a join algorithm crucial
- Simple *block-nested loops* join algorithm for computing $r \bowtie_{A=B} s$

```
foreach page $p_r$ in r do
    foreach page $p_s$ in s do
        output $p_r \bowtie_{A=B} p_s$
```

- If we do this in tuple level, Page(R) + Tuple(R) * Page(S)
- Consider that Page(R) = 1000, Page(S) = 100, tuple(R) = 10,000,
  - *If outer loop is for R*, 1000 + 10000*100 = 1,001,000 page transfer. --- too many…
  - *If outer loop is for S*,
  - 100 + 1000*1000 = 1,000,100 page transfer. --- fewer, too many…

Block-Nested Loops Join

- If β_r and β_s are the number of pages in **r** and **s**, the cost of algorithm is

Number of scans of relation **s**

$\beta_r + \beta_r * \beta_s$ + *cost of outputting final result*

- If **r** and **s** have 10^3 pages each,

  cost is $10^3 + 10^3 * 10^3$
- *Choose smaller relation for the outer loop*:
  - If $\beta_r < \beta_s$ then $\beta_r + \beta_r * \beta_s < \beta_s + \beta_r * \beta_s$

Block-Nested Loops Join

- Cost can be reduced to

$\beta_r + (\beta_r/(M-2)) * \beta_s$ + *cost of outputting final result*

Number of scans of relation ***s***

by using M buffer pages instead of 1.

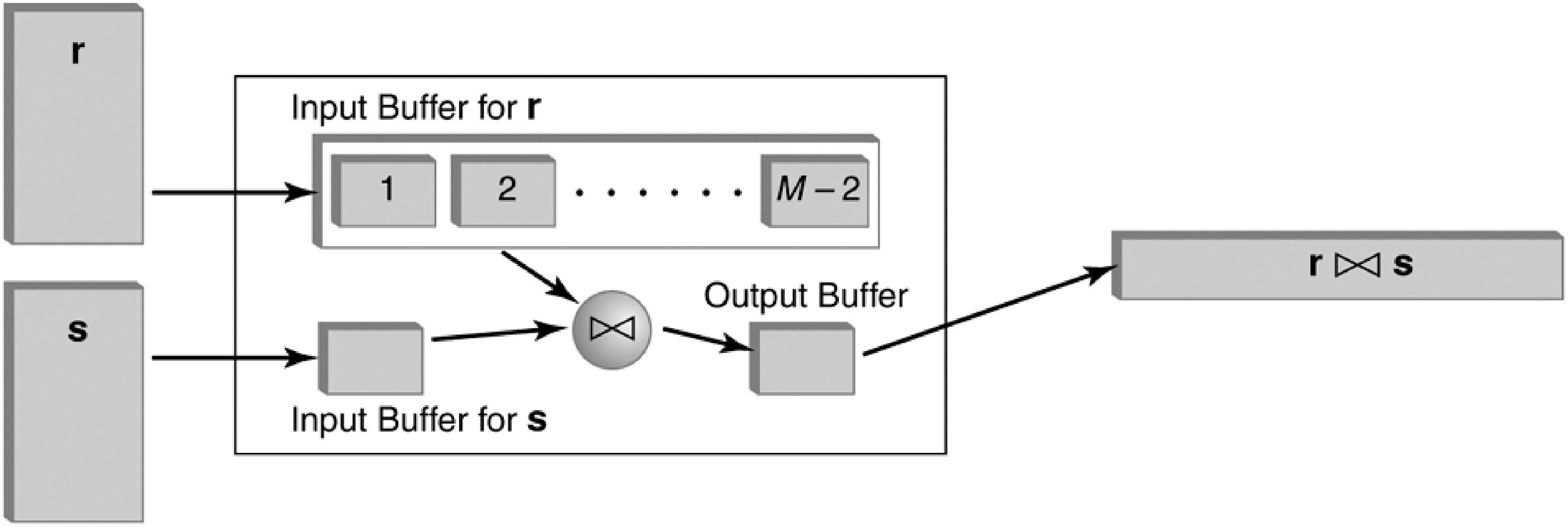
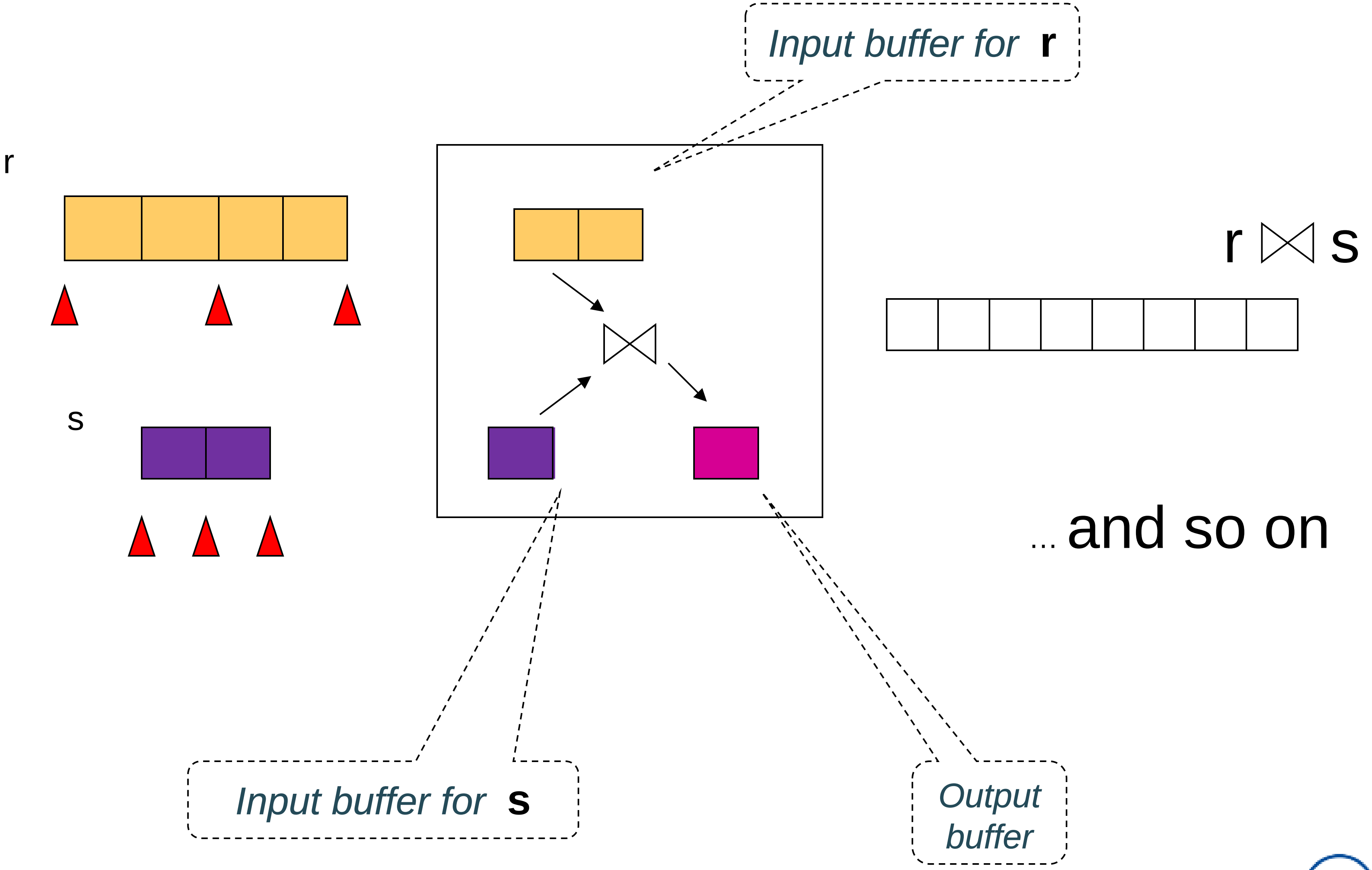


FIGURE 10.6 Block-nested loops join.

Block-Nested Loop Illustrated



… and so on

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 = $\beta_r + \tau_r * \omega$ + *cost of outputting final result*

Number of rows in **r**

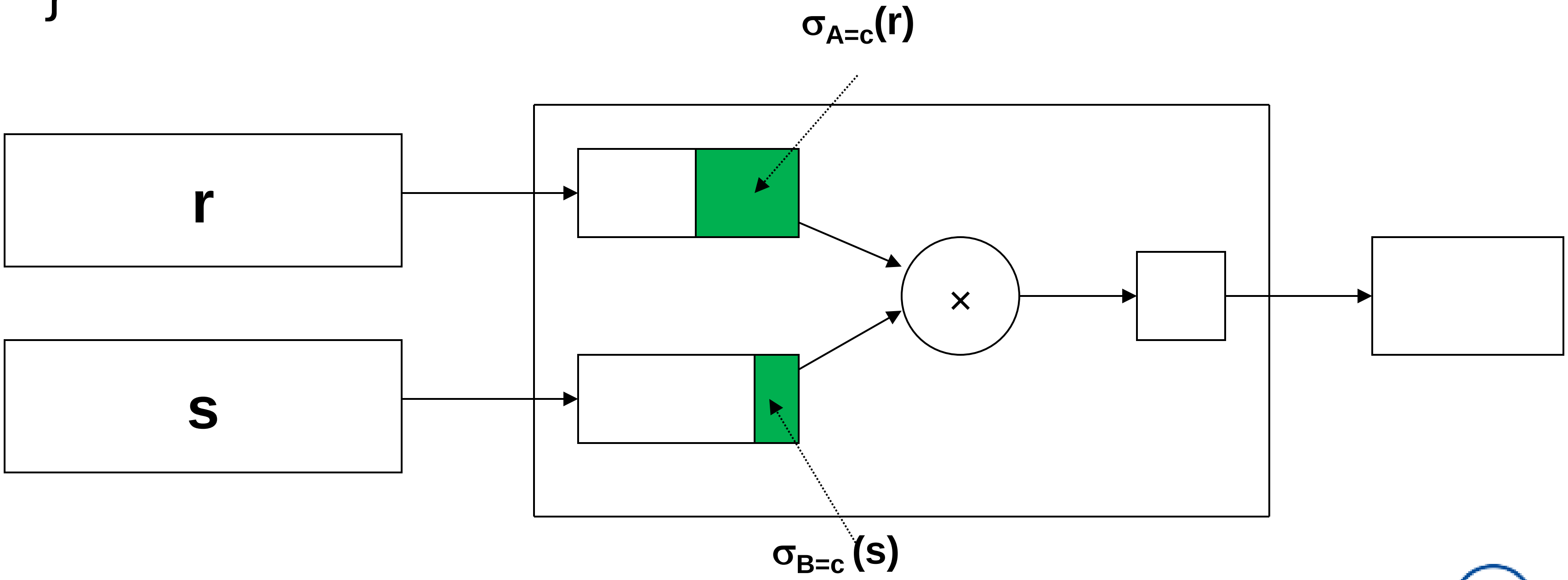
avg cost of retrieving *all* rows in **s** that match t_r

  - Effective if number of rows of **s** that match tuples in **r** is small (i.e., ω is small) and index is <u>clustered</u>

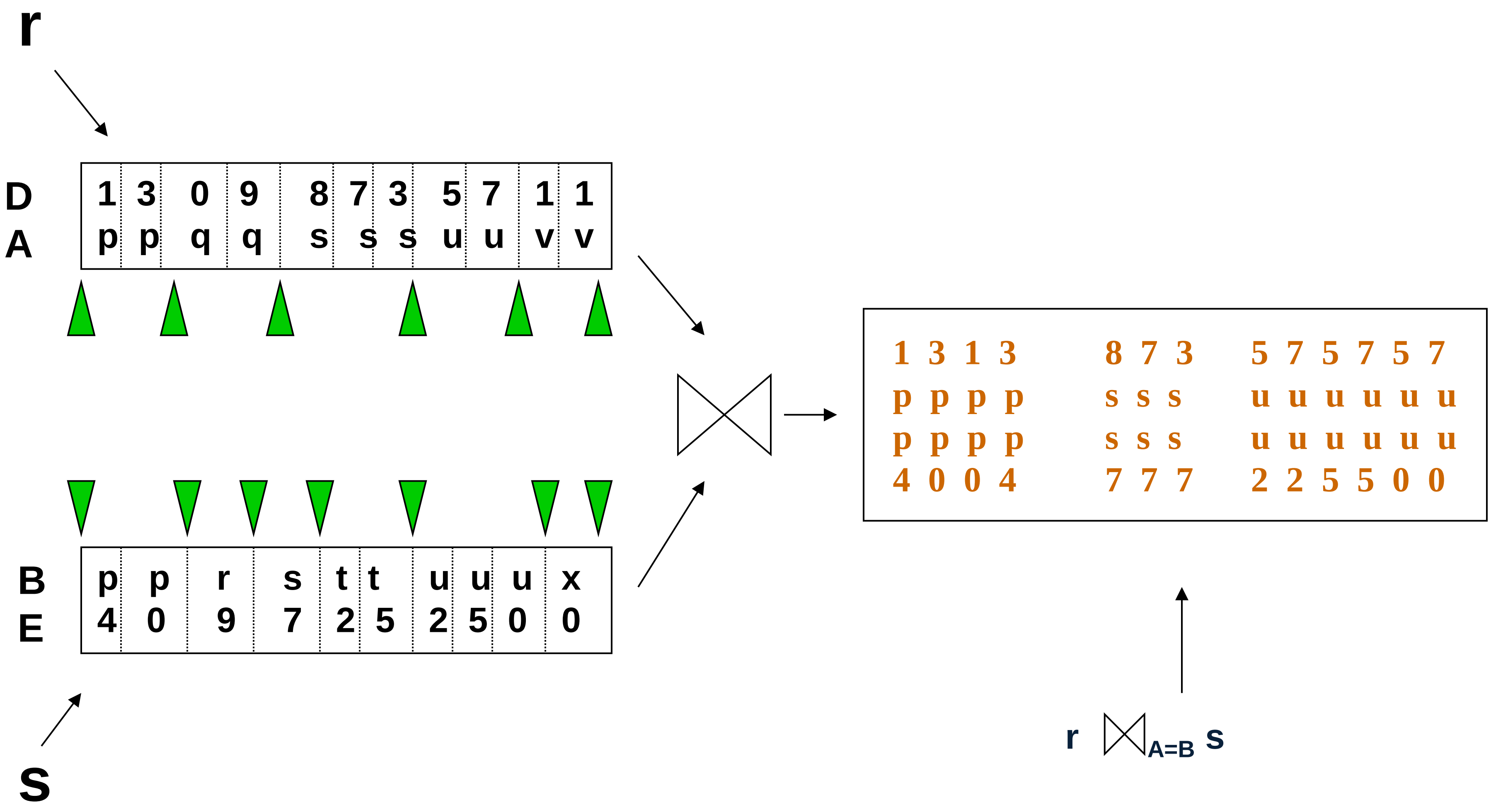
```
foreach tuple $t_r$ in r do {
    use index to find all tuples $t_s$ in s satisfying $t_r.A=t_s.B$;
    output ($t_r$, $t_s$)
}
```

Sort-Merge Join **r** $\bowtie_{A=B}$ **s**

```
sort r on A;
sort s on B;
while !eof(r) and !eof(s) do {
    Scan r and s concurrently until $t_r.A = t_s.B = c$;
    Output $\sigma_{A=c}(r) \times \sigma_{B=c}(s)$
}
```



Join During Merge Illustrated



D	1	3	0	9	8	7	3	5	7	1	1
A	p	p	q	q	s	s	s	u	u	v	v

B	p	p	r	s	t	t	u	u	u	x
E	4	0	9	7	2	5	2	5	0	0

1	3	1	3	8	7	3	5	7	5	7	5	7
p	p	p	p	s	s	s	u	u	u	u	u	u
p	p	p	p	s	s	s	u	u	u	u	u	u
4	0	0	4	7	7	7	2	2	5	5	0	0

$r \bowtie_{A=B} s$

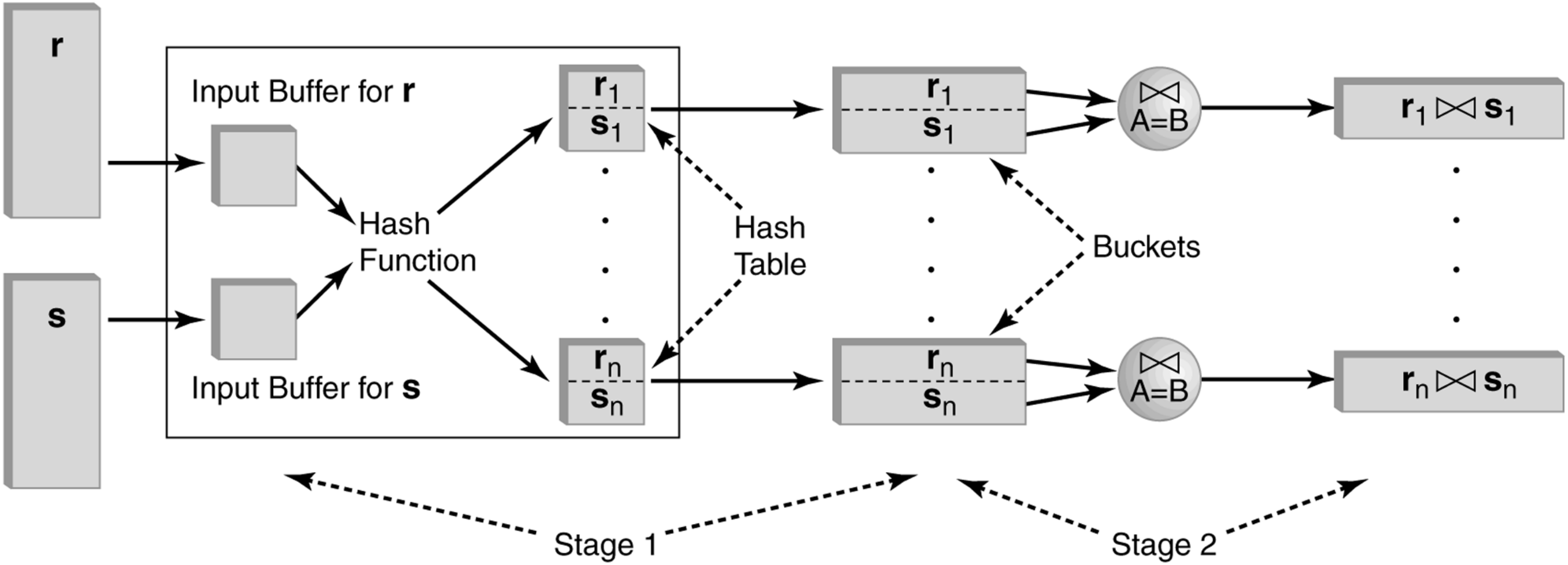
Cost of Sort-Merge Join

- Cost of *sorting* assuming *M* buffers:
  - $2\ \beta_r \log_{M-1} \beta_r + 2\ \beta_s \log_{M-1} \beta_s$
- Cost of *merging:*
  - Scanning $\sigma_{A=c}(\mathbf{r})$ and $\sigma_{B=c}(\mathbf{s})$ can be combined with the last step of sorting of **r** and **s** --- costs nothing
  - Cost of $\sigma_{A=c}(\mathbf{r})\times\sigma_{B=c}(\mathbf{s})$ depends on whether $\sigma_{A=c}(\mathbf{r})$ can fit in the buffer
    - If yes, this step costs 0
    - In not, each $\sigma_{A=c}(\mathbf{r})\times\sigma_{B=c}(\mathbf{s})$ is computed using *block-nested* join, so the cost is the cost of the join. (Think why indexed methods or sort-merge are inapplicable to Cartesian product.)
- Cost of outputting the *final result* depends on the size of the result

Hash-Join $\mathbf{r} \bowtie_{A=B} \mathbf{s}$

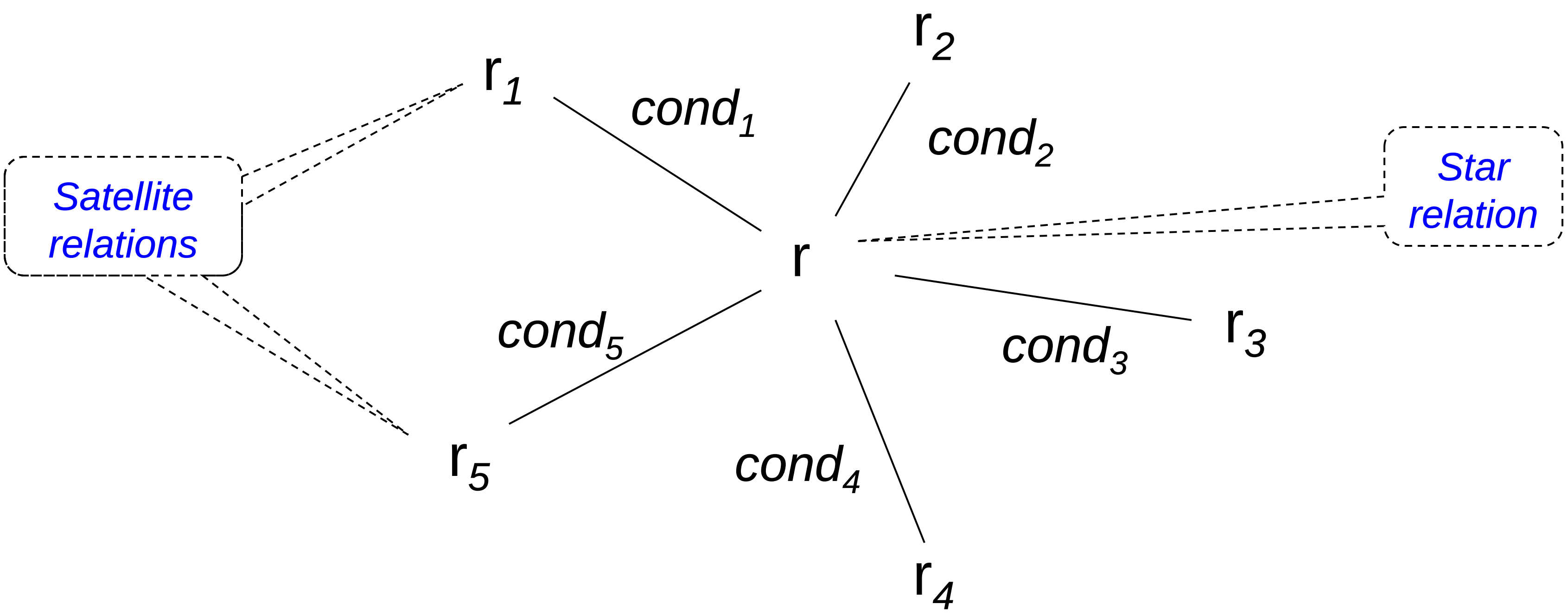
- *Step* 1: Hash **r** on A and **s** on B into the same set of buckets
- *Step* 2: Since matching tuples must be in same bucket, read each bucket in turn and output the result of the join
- *Cost*: $3(\beta_r + \beta_s) +$ *cost of output of final result*
    - assuming each bucket fits in memory

Hash Join

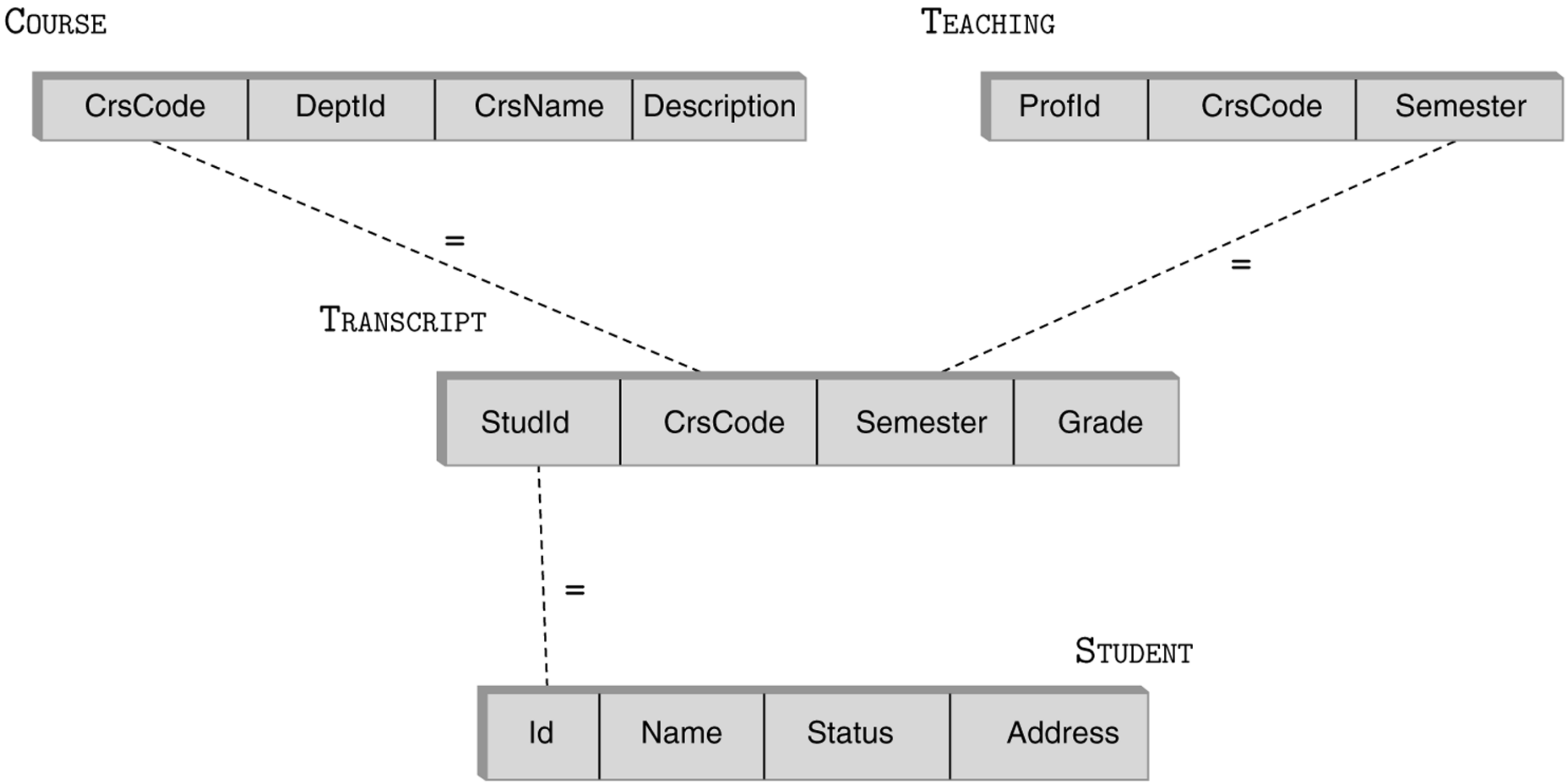

r
s
Input Buffer for r
Input Buffer for s
Hash Function
r_1
s_1
r_n
s_n
Hash Table
Buckets
$\bowtie$ A=B
$r_1 \bowtie s_1$
$r_n \bowtie s_n$
Stage 1
Stage 2

Star Joins

- $\mathbf{r} \bowtie_{cond_1} \mathbf{r}_1 \bowtie_{cond_2} \cdots \bowtie_{cond_n} \mathbf{r}_n$
  - Each $cond_i$ involves only the attributes of $\mathbf{r}_i$ and $\mathbf{r}$



Star Join


COURSE
CrsCode
DeptId
CrsName
Description
TEACHING
ProfId
CrsCode
Semester
=
=
TRANSCRIPT
StudId
CrsCode
Semester
Grade
=
STUDENT
Id
Name
Status
Address

Computing Star Joins

- Use *join index*
  - Scan **r** and the join index {<$r,r_1,\ldots,r_n$>} (which is a set of tuples of rids) in one scan
  - Retrieve matching tuples in $\mathbf{r}_1,\ldots,\mathbf{r}_n$
  - Output result

Computing Star Joins

- Use *bitmap indices*
  - Use one bitmapped join index, J_i, per each partial join

    $\mathbf{r} \bowtie_{cond_i} \mathbf{r}_i$
  - *Recall*: J_i is a set of <*v*, *bitmap*>, where *v* is an rid of a tuple in $\mathbf{r}_i$ and *bitmap* has 1 in *k*-th position iff *k*-th tuple of **r** joins with the tuple pointed to by *v*

1. Scan J_i and logically OR all bitmaps. We get all rids in **r** that join with $\mathbf{r}_i$
2. Now logically AND the resulting bitmaps for $J_1, \ldots, J_n$.
3. Result: a subset of **r**, which contains all tuples that can possibly be in the star join
   - *Rationale*: only a few such tuples survive, so can use indexed loops

Computing Aggregated Functions

- Require full scan
- In case that tuples are *grouped by attributes*,
  - Need to partition relation with the attribute values
    - Sorting
    - Hashing
    - Indexing

Choosing Indices

- DBMSs may allow user to specify
  - Type (hash, B^+ tree) and search key of index
  - Whether or not it should be clustered
- Using information about the frequency and type of queries and size of tables, designer can use cost estimates to choose appropriate indices
- Several commercial systems have tools that suggest indices
  - Simplifies job, but index suggestions must be *verified*

Choosing Indices – Example

- If a frequently executed query that involves selection or a join and has a large result set,
  - Use a clustered B^+ tree index
  - *e.g.*, Retrieve all rows of Transcript for *StudId*
- If a frequently executed query is an equality search and has a small result set,
  - An unclustered hash index is best, since only one clustered index on a table is possible, choosing unclustered allows a different index to be clustered
  - *e.g.*, Retrieve all rows of Transcript for (*StudId*, *CrsCode*)