Chapter 12: Query Processing

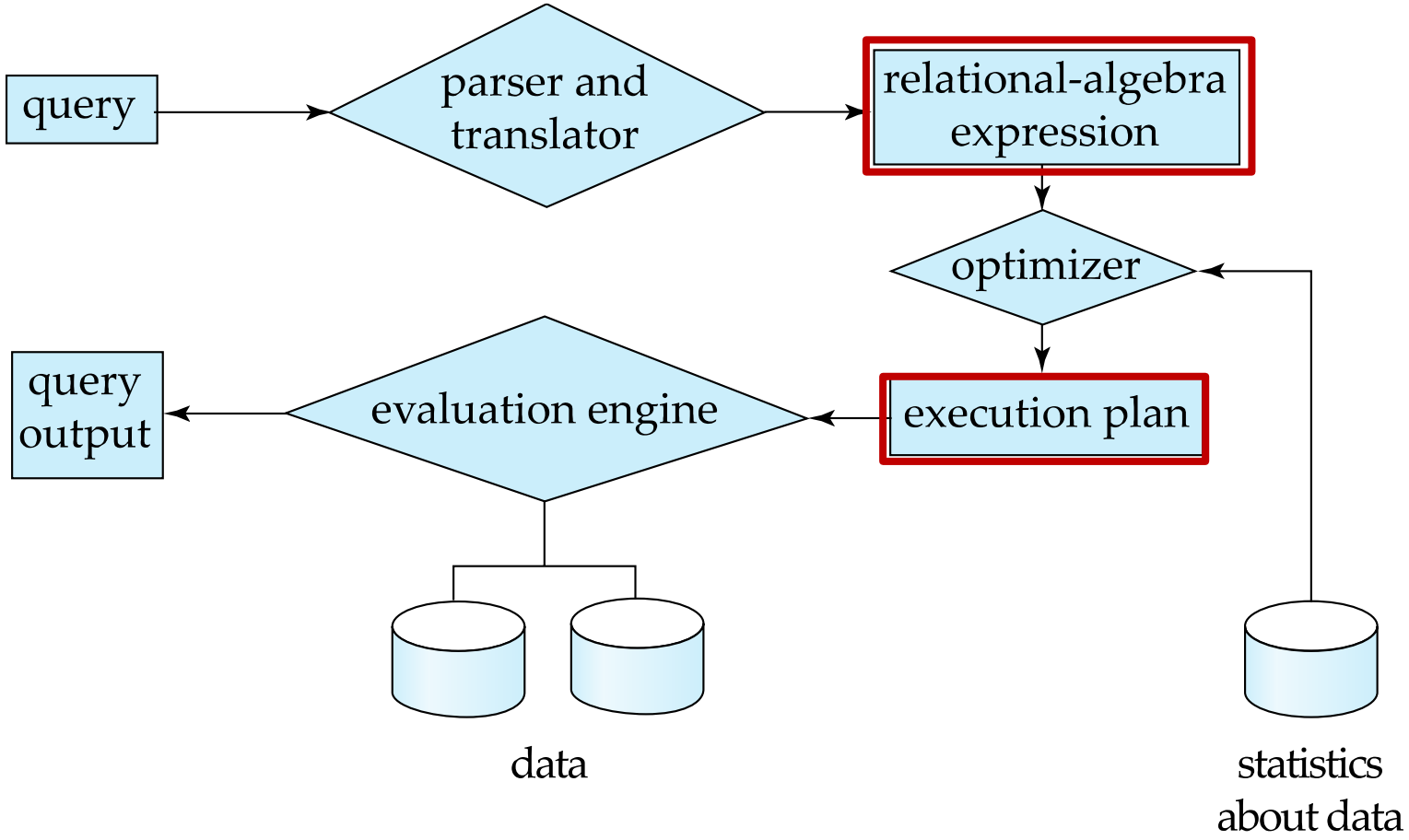
Database System Concepts, 6th Ed.

Chapter 12: Query Processing

- 12.1 Overview
- 12.2 Measures of Query Cost
- 12.3 Selection Operation
- 12.4 Sorting
- 12.5 Join Operation
- 12.6 Other Operations
- 12.7 Evaluation of Expressions

Basic Steps in Query Processing [1/2]

1. Parsing and translation
2. Optimization
3. Evaluation



Remember?

SQL and Multiset Relational Algebra

- **select** *A1, A2, .. An*
  from *r1, r2, …, rm*
  where P

  is equivalent to the following expression in multiset relational algebra

$$\Pi_{A1, .., An} (\sigma_{P} (r1 \times r2 \times .. \times rm))$$

- More generally, the non-aggregated attributes in the **select** clause may be a subset of the **group by** attributes, in which case the equivalence is as follows:

  select *A1,* **sum***(A3)*
  from *r1, r2, …, rm*
  where P
  group by *A1, A2*

  is equivalent to the following expression in multiset relational algebra

$$\Pi_{A1,sumA3} ({}_{A1,A2}\mathcal{G}_{\textbf{sum}(A3)\ \textbf{as}\ sumA3} (\sigma_{P} (r1 \times r2 \times .. \times rm)))$$

Basic Steps in Query Processing [2/2]

- Parsing and translation
  - translate the query into its internal form
  - This is then translated into relational algebra
  - Parser checks syntax, verifies relations
- Optimization
  - Enumerate all possible query-evaluation plans
  - Compute the cost for the plans
  - Pick up the plan having the minimum cost
- Evaluation
  - The query-execution engine takes a query-evaluation plan,
  - executes that plan,
  - and returns the answers to the query

Basic Steps in Query Optimization [1/2]

- A relational algebra expression may have many equivalent expressions
  - E.g., $\sigma_{salary<75000}(\Pi_{salary}(instructor))$ is equivalent to $\Pi_{salary}(\sigma_{salary<75000}(instructor))$
- Each relational algebra operation can be evaluated using one of several different algorithms
  - Correspondingly, a relational-algebra expression can be evaluated in many ways
- Annotated expression specifying detailed evaluation strategy is called an **evaluation-plan**
  - (Evaluation Plan 1) use an index on *salary* to find instructors with salary < 75000
  - (Evaluation Plan 2) perform complete relation scan and discard instructors with salary ≥ 75000

Basic Steps in Query Optimization [2/2]

- **Query Optimization**
  - Amongst all equivalent evaluation plans choose the one with lowest cost
  - Cost is estimated using statistical information from the database catalog
    - e.g. number of tuples in each relation, size of tuples, etc
- In this chapter we study
  - How to measure query costs
  - Algorithms for evaluating relational algebra operations
  - How to combine algorithms for individual operations in order to evaluate a complete expression
- In Chapter 13
  - We study how to optimize queries, that is, how to find an evaluation plan with lowest estimated cost

Chapter 12: Query Processing

- 12.1 Overview
- 12.2 Measures of Query Cost
- 12.3 Selection Operation
- 12.4 Sorting
- 12.5 Join Operation
- 12.6 Other Operations
- 12.7 Evaluation of Expressions

Measures of Query Cost [1/3]

- Cost is generally measured as total elapsed time for answering query
  - Many factors contribute to time cost
    - *disk accesses, CPU*, or even network *communication*
- Typically disk access is the predominant cost, and is also relatively easy to estimate
- Disk IO time is measured by taking into account
  - Number of seeks X average-seek-cost
    - seek ➔ block-access ➔ disk seek + rotational latency
  - Number of blocks read X average-block-read-cost
  - Number of blocks written X average-block-write-cost
    - Cost to write a block is greater than cost to read a block
      - data is read back after being written to ensure that the write was successful

Measures of Query Cost [2/3]

- For simplicity we just use the **number of block transfers** *from disk and the* **number of seeks** as the cost measures
  - t_T : average time to transfer one block
  - t_S : average time for one seek (disk seek time + rotational latency time)
  - Cost for b block transfers plus S seeks: $b * t_T + S * t_S$
- We ignore the difference in cost between sequential and random I/O for simplicity
- We ignore CPU costs for simplicity
  - Real systems do take CPU cost into account
- We do not include cost to writing output to disk in our cost formulae

Measures of Query Cost [3/3]

- Several algorithms can reduce disk IO by using extra buffer space
  - Amount of real memory available to buffer depends on other concurrent queries and OS processes, known only during execution
    - We often use worst case estimates, assuming only the minimum amount of memory needed for the operation is available
      - I.e., The buffer can hold only a few blocks, approximately only one block per relation

- Required data may be buffer resident already, avoiding disk I/O
  - But hard to take into account for cost estimation
  - So, we assume that data must be read from disk initially

Chapter 12: Query Processing

- 12.1 Overview
- 12.2 Measures of Query Cost
- 12.3 Selection Operation
- 12.4 Sorting
- 12.5 Join Operation
- 12.6 Other Operations
- 12.7 Evaluation of Expressions

Selection Operation

** **File scan** for search algorithms that retrieve records that fulfill a selection condition

- t_T : time to transfer one block
- t_S : time for one seek
- Cost for b block transfers plus S seeks: $b * t_T + S * t_S$

- Algorithm **A1** (**linear search, no index**)
  - If the query needs to check all records to see whether they satisfy the selection condition
    - Cost estimate = 1 initial seek (t_S) + b_r block transfers ($b_r * t_T$)
    - b_r denotes number of blocks containing records from relation *r*
  - If selection is on a key attribute (sorted), we can stop on finding record early
    - cost = 1 initial seek (t_S) + (b_r /2) block transfers ((b_r /2) * t_T) // average case
  - Linear search can be applied regardless of
    - selection condition or ordering of records in the file, or availability of indices

File Scan Cases

Query A1-1: find record with department = Finance
Query A1-2: find record with id = 60000

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

Note: binary search generally does not make sense since data is not stored consecutively

- except when there is an index available
- and even with an index, direct binary search on the relation requires more seeks than index search

Selections using Indices [1/2]

** **Index scan** – search algorithms that use an B+ tree index

- selection condition must be on search-key of index

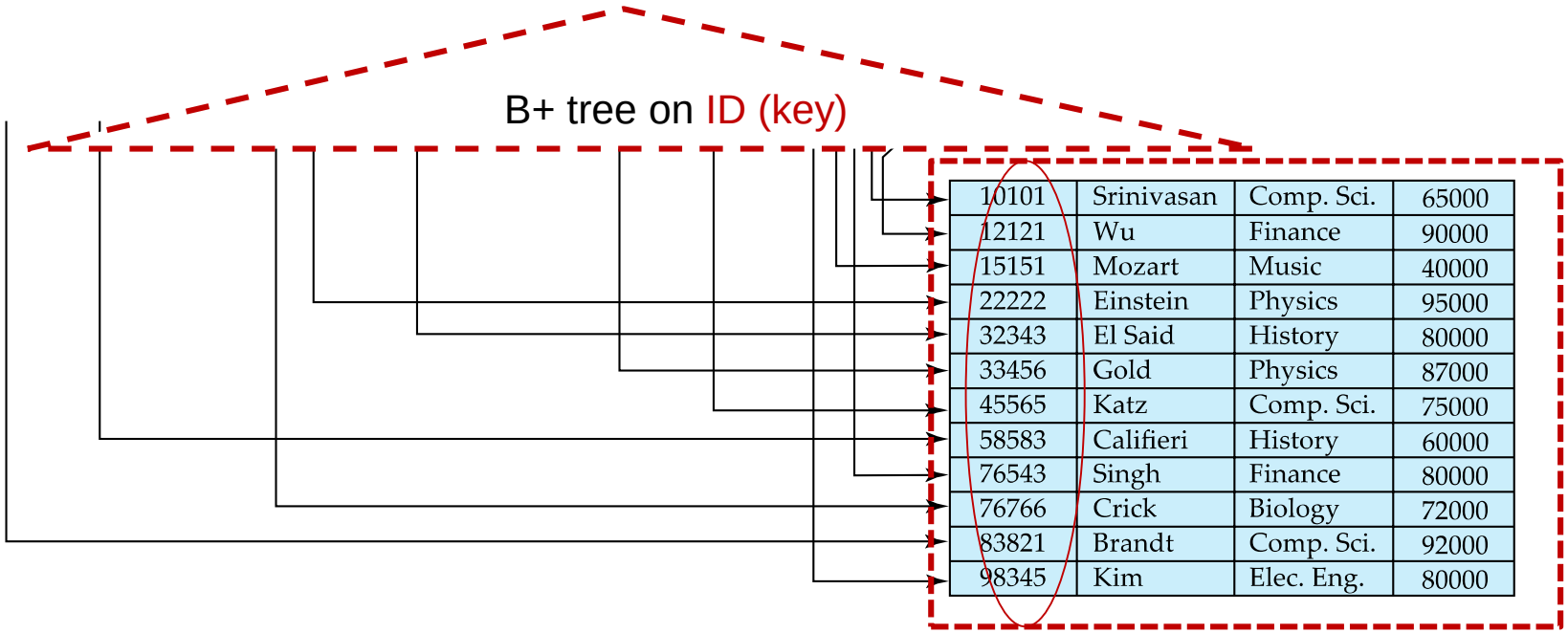
- t_T : time to transfer one block
- t_S : time for one seek
- Cost for b block transfers plus S seeks: $b * t_T + S * t_S$

- **A2** (**primary index on key (sorted attribute), equality on key**)
  - Retrieve a single record that satisfies the corresponding equality condition (h_i is the height of index and 1 block IO for fetching the data)
  - $Cost = (h_i + 1) * (t_T + t_S)$

- **A3** (**primary index on non key (unsorted attribute), equality on nonkey**)
  - Need to retrieve multiple records
  - Matching records will be on consecutive blocks
    - Let b = number of blocks containing matching records
  - $Cost = h_i * (t_T + t_S) + t_S + t_T * b$

Index Scan Cases

Query-A2: find record with Id = 76766



10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Query-A3: find record with Dept = EE

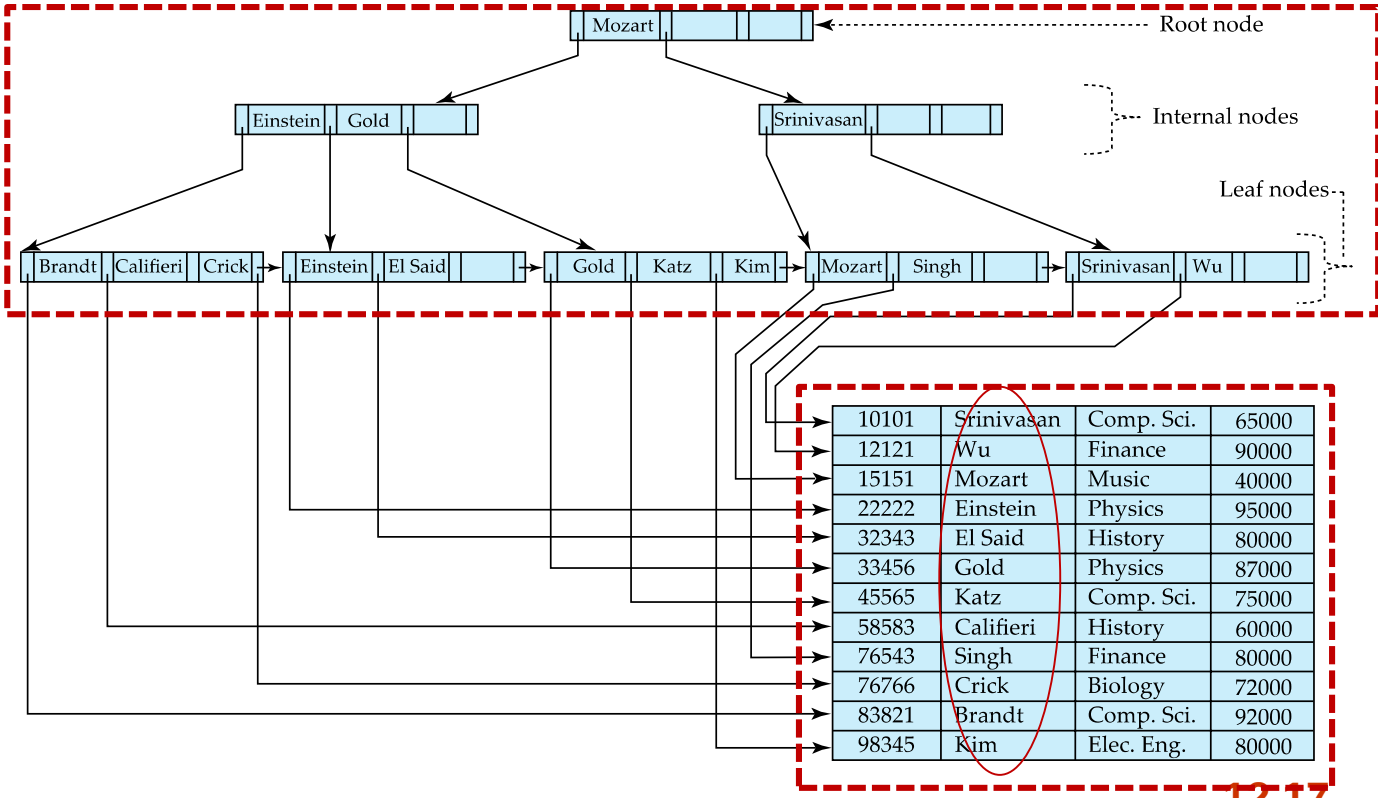
Multiple values

10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Selections using Indices [2/2]

- **A4** (**secondary index on nonkey, equality on the nonkey**)
  - Retrieve a single record if the search-key is a candidate key
    - *Cost* = $(h_i + 1) * (t_T + t_S)$
  - Retrieve multiple records if search-key is not a candidate key
    - each of *n* matching records may be on a different block (very expensive!)
    - Cost = $(h_i + n) * (t_T + t_S)$

Query-A4: Find record with Name = Brandt



Selections Involving Comparisons

** Can implement selections of the form $\sigma_{A \le V}(r)$ or $\sigma_{A \ge V}(r)$

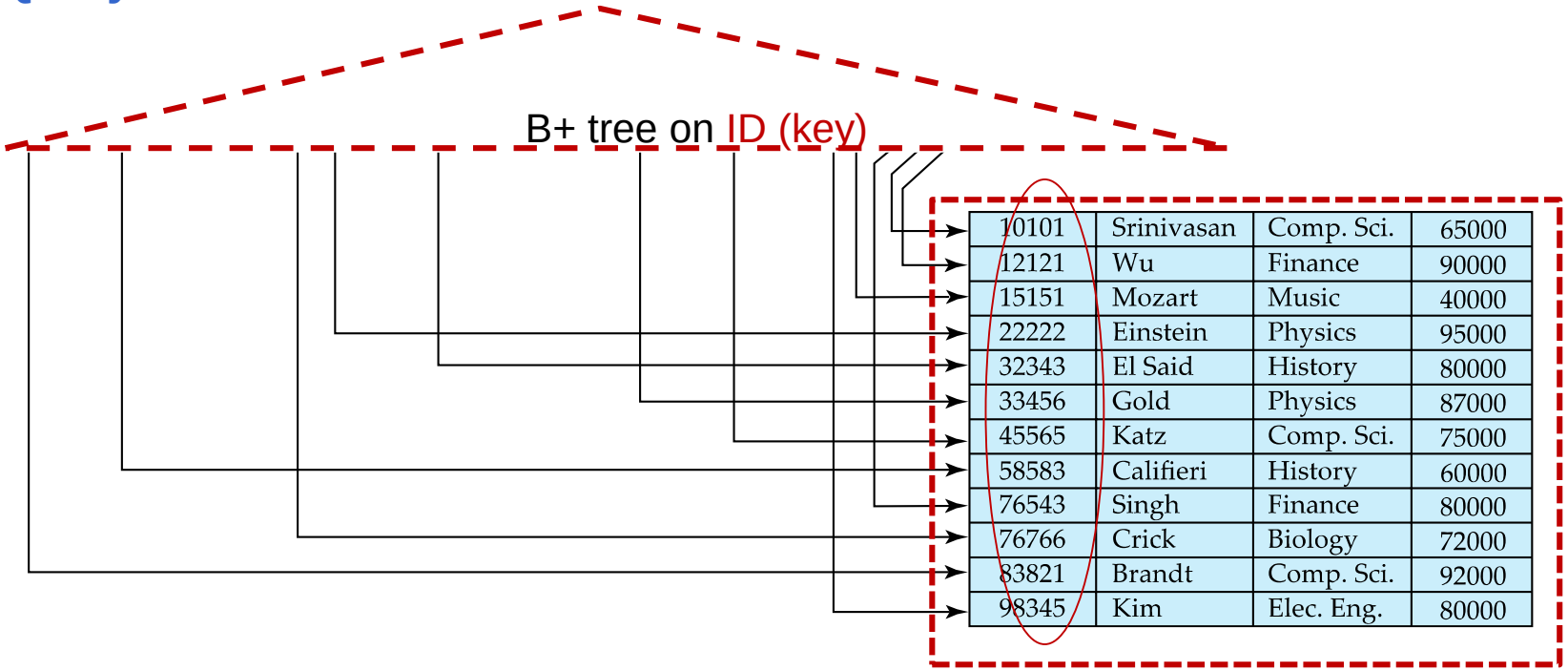
- by using a linear file scan or
- by using B+ tree indices in the following ways

- **A5 (primary index, comparison on the key).** (Relation is sorted on A) (similar to A3)
  - For $\sigma_{A \ge V}(r)$ use index to find first tuple $\ge v$ and scan relation sequentially from there
  - For $\sigma_{A \le V}(r)$ just scan relation sequentially till first tuple > *v;* do not use index
  - Let b = number of blocks containing matching records
  - *Cost* = $h_i * (t_T + t_S) + t_T * b$

- **A6 (secondary index, comparison on the nonkey).** (similar to A4)
  - For $\sigma_{A \ge V}(r)$ use index to find first index entry $\ge v$ and scan index sequentially from there, to find pointers to records.
  - For $\sigma_{A \le V}(r)$ just scan leaf pages of index finding pointers to records, till first entry > *v*
  - In either case, retrieve *n* records that are pointed to
    - Requires an I/O for each record
  - Cost = $(h_i + n) * (t_T + t_S)$
  - At sometimes, linear file scan may be cheaper if many records are to be fetched!

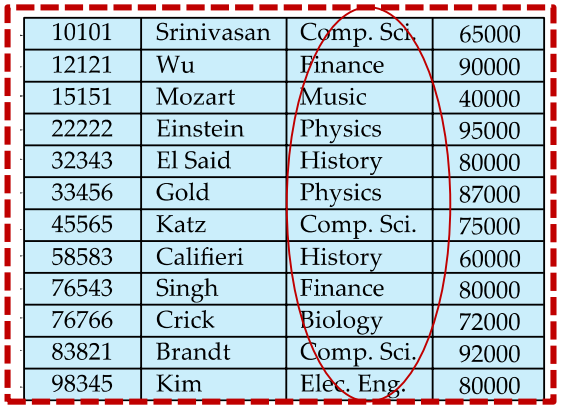
Cases for Index Scan involving Comparisons

Query-A5: find record with Id > 30000



Query-A6: find record with Dept > CS

Multiple values



Implementation of Complex Selections [1/2]

** **Conjunctive selection query:** $\sigma_{\theta 1 \wedge \theta 2 \wedge \ldots \theta n}(r)$

- **A7** (**conjunctive selection using one index**)
  - Suppose we have one B+ tree index for one attribute in one of conditions
  - Select a combination of θ_i and algorithms A2 -- A6 that results in the least cost for $\sigma_{\theta i}(r)$
  - Test other conditions on tuple after fetching it into memory buffer
  - The cost of A7 is the cost of the chosen algorithm

- **A8** (**conjunctive selection using composite index**)
  - Suppose we have appropriate composite (multiple-key) index for multiple attributes
  - A2, A3, A4 can be used
  - The cost of A8 is the cost of the chosen algorithm

- **A9** (**conjunctive selection by intersection of identifiers**)
  - Suppose we have several B+ tree indices with record pointers
  - Use corresponding index for each condition, and take intersection of sets of record pointers
  - Then fetch records from file using the record pointers
  - If some conditions do not have appropriate indices, apply test in memory
  - Cost: individual index scans + retrieving records in the intersection of pointers

Implementation for Complex Selections [2/2]

**** Disjunctive Selection Query:** $\sigma_{\theta 1 \vee \theta 2 \vee \ldots \theta n}(r)$

- **A10** (**disjunctive selection by union of identifiers**).
  - Suppose we have B+tree indices for all conditions
    - Use corresponding index for each condition, and take union of all the obtained sets of record pointers
    - Then fetch records from file
    - Cost: individual index scans + retrieving records in the union of pointers
  - Otherwise have to use linear scan
    - Cost: cost of A1

**** Negation Query:** $\sigma_{\neg\theta}(r)$

- Bring in all records to check the negation condition
  - Use linear scan on file with the cost of A1
- If very few records satisfy $\neg\theta$, and an index is applicable to θ
  - Find records satisfying θ using index and fetch from file
  - Example: Number of Instructors (Engineering:500, English:3, History:1)
    - Find instructors whose department is not Engineering
    - Find instructors whose department is English or History

Chapter 12: Query Processing

- 12.1 Overview
- 12.2 Measures of Query Cost
- 12.3 Selection Operation
- 12.4 Sorting
- 12.5 Join Operation
- 12.6 Other Operations
- 12.7 Evaluation of Expressions

Sorting

- Sorting is very important in Database
  - Building a B+ tree index is efficient on a sorted attribute
  - Join processing can be efficient on a sorted relation
  - Reading tuples in the sorted order is efficient in physically sorted relations
  - We may specify SQL query result to be sorted on a particular attribute

- For relations that fit in memory, techniques like **quicksort** can be used
  - Pick a middle element P in an array A
  - Push the elements having less value than P to the left array LA
  - Push the elements having bigger value than P to the right array RA
  - Apply the same idea recursively on LA and RA

- For relations that don't fit in memory, **external sort-merge** is a good choice

External Sort-Merge [1/2]

Let *M* denote memory size (in pages): M blocks in memory

- **Create N sorted runs**

  Let *i* be 0 initially.

  Repeatedly do the following till the end of the relation:

  (a) Read *M* blocks of relation into memory

  (b) Sort the in-memory M blocks (e.g. using quick-sort)

  (c) Write sorted data to make a run file R_i;

  increment *i*

  Let the final value of *i* be *N*

- *Merge the N runs (next slide)*

  If $M > N$ ➔ we can merge N runs in a single pass

  If $M \leq N$ ➔ we need more than a single pass

External Sort-Merge [2/2]

- **If M > N, merge the N sorted runs (N-way merge)** in a single pass

  Use *N* blocks of memory to buffer input runs, and 1 block to buffer output

  Read the first block of each run into its buffer page

  Repeat

  1. Select the first record (in sort order) among all buffer pages

  2. Write the record to the output buffer

     If the output buffer is full, **then** write it to disk

  3. Delete the record from its input buffer page

     If the buffer page becomes empty **then** read the next block

  until all input buffer pages are empty:

- If **M ≤ N**, several merge *passes* are required

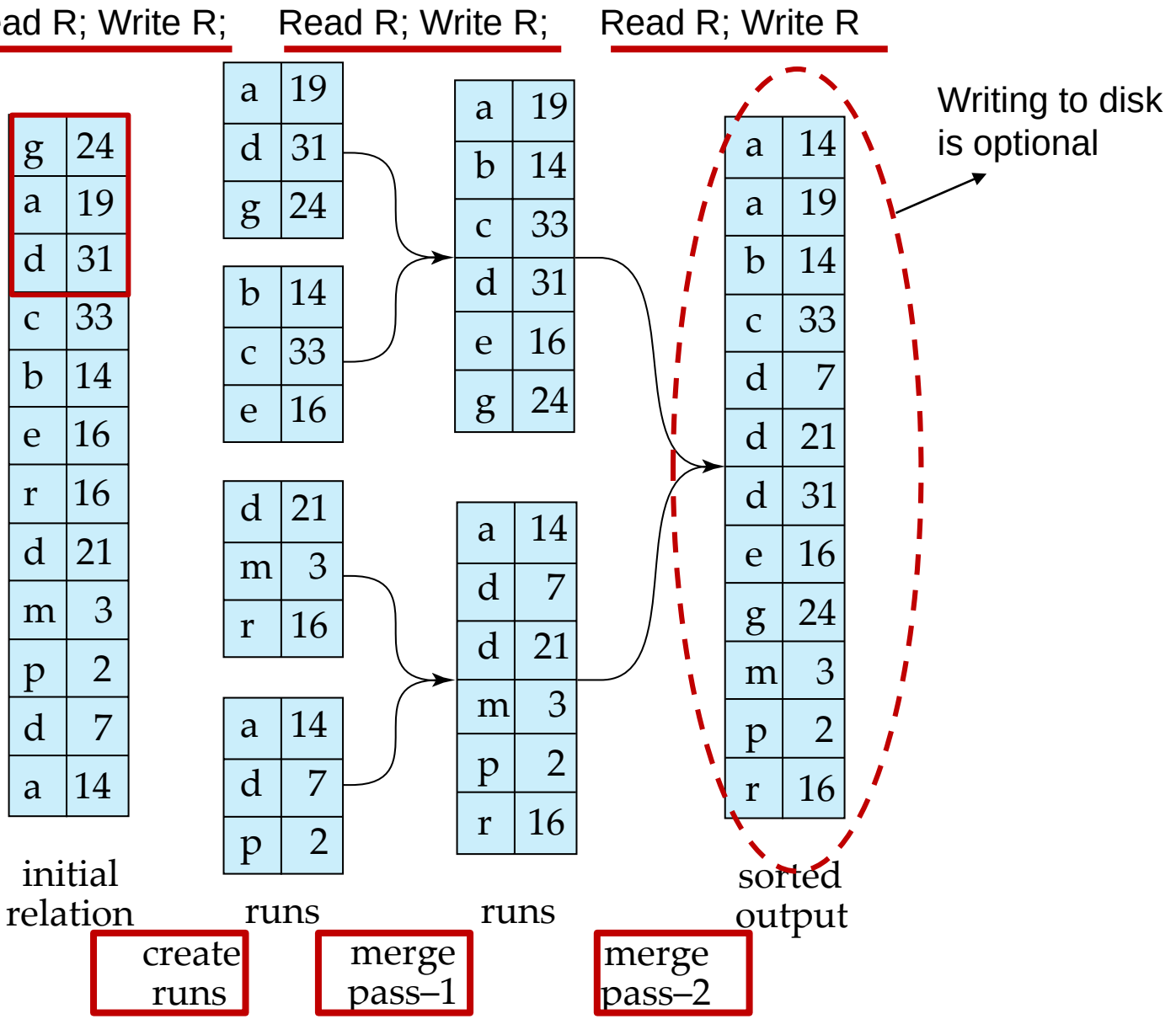
  - In each pass, contiguous groups of *M* - 1 runs are merged
  - A pass reduces the number of runs by a factor of *M* -1, and creates runs longer by the same factor
    - E.g. If M = 11, and there are 90 runs, one pass reduces the number of runs from 90 to 9, each run is 10 times the size of the initial runs
  - Repeated passes are performed till all runs have been merged into one

Example: External Sort-Merge

Suppose 1 record in 1 block

Suppose 3 block buffers in memory
M = 3

Initially 4 runs
N = 4



Run의 개수는 줄고, Run의 길이는 늘고!

Cost Analysis in External Sort-Merge [1/2]

- Total Number of Block IOs
  - b_r : the number of blocks in a relation r
  - M : the number of blocks in each run
  - $N = b_r / M$ *: the initial number of runs*
  - Total number of merge passes required: $\lceil \log_{M-1} N \rceil$
  - Block transfers for initial run creation as well as in each pass is $2b_r$
    - for final pass, we may not count write-cost
      - Sort된 relation을 저장할수 있지만, 어떤 query를 위해서 relation을 sort한 경우라면 final write to disk는 필요없을수도 있음
  - Thus total number of block transfers for external sorting:
    $b_r (2 \lceil \log_{M-1} N \rceil + 1)$

Cost Analysis in External Sort-Merge [2/2]

- Total Number of Seeks
  - b_r : the number of blocks in a relation r
  - M : the number of blocks in each run
  - $N = b_r / M$ *: the initial number of runs*
  - Total number of merge-passes required: $\lceil \log_{M-1} N \rceil$
  - During the run creation
    - one seek to read each run and one seek to write each run: $2\lceil N \rceil$
  - During the merge phase
    - Buffer size: b_b (read/write b_b blocks at a time)
      - Buffer가 크면 seek숫자를 줄인다
    - Need $\lceil b_r / b_b \rceil$ seeks for read runs of each merge-pass
      - except the final one which does not require a write
      - Thus, $\lceil b_r / b_b \rceil (2\lceil \log_{M-1} N \rceil - 1)$
  - Total number of seeks:
    $2\lceil N \rceil + \lceil b_r / b_b \rceil (2\lceil \log_{M-1} N \rceil - 1)$

Chapter 12: Query Processing

- 12.1 Overview
- 12.2 Measures of Query Cost
- 12.3 Selection Operation
- 12.4 Sorting
- 12.5 Join Operation
- 12.6 Other Operations
- 12.7 Evaluation of Expressions

Join Operation

- Several different algorithms to implement joins
  - Nested-Loop Join
  - Block Nested-Loop Join
  - Indexed Nested-Loop Join
  - Merge-Join and Hybrid Merge-Join
  - Hash-Join and Hybrid Hash-Join
- Choice based on cost estimate
- Examples use the following information (next slide)
  - Number of records of *student*: 5,000
  - Number of records of *takes*: 10,000
  - Number of blocks of *student*: 100
  - Number of blocks of *takes*: 400

Outer Relation

Inner Relation

The Student Relation r

ID	name	dept_name	tot_cred
00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
19991	Brandt	History	80
23121	Chavez	Finance	110
44553	Peltier	Physics	56
45678	Levy	Physics	46
54321	Williams	Comp. Sci.	54
55739	Sanchez	Music	38
70557	Snow	Physics	0
76543	Brown	Comp. Sci.	58
76653	Aoi	Elec. Eng.	60
98765	Bourikas	Elec. Eng.	98
98988	Tanaka	Biology	120

⋈

The Takes relation s

ID	course_id	sec_id	semester	year	grade
00128	CS-101	1	Fall	2009	A
00128	CS-347	1	Fall	2009	A-
12345	CS-101	1	Fall	2009	C
12345	CS-190	2	Spring	2009	A
12345	CS-315	1	Spring	2010	A
12345	CS-347	1	Fall	2009	A
19991	HIS-351	1	Spring	2010	B
23121	FIN-201	1	Spring	2010	C+
44553	PHY-101	1	Fall	2009	B-
45678	CS-101	1	Fall	2009	F
45678	CS-101	1	Spring	2010	B+
45678	CS-319	1	Spring	2010	B
54321	CS-101	1	Fall	2009	A-
54321	CS-190	2	Spring	2009	B+
55739	MU-199	1	Spring	2010	A-
76543	CS-101	1	Fall	2009	A
76543	CS-319	2	Spring	2010	A
76653	EE-181	1	Spring	2009	C
98765	CS-101	1	Fall	2009	C-
98765	CS-315	1	Spring	2010	B
98988	BIO-101	1	Summer	2009	A
98988	BIO-301	1	Summer	2010	*null*

n_r 5000 Records
b_r 100 Blocks

⋈

n_s 10000 Records
b_s 400 Blocks

1block에 50개 records

1block에 25개 records

Nested-Loop Join

- To compute the theta join $r \bowtie_\theta s$

  for each tuple t_r **in** r **do begin**
  for each tuple t_s **in** s **do begin**
  test pair (t_r, t_s) to see if they satisfy the join condition θ
  if they do, add $t_r \bullet t_s$ to the result.
  end
  end

- r is called the **outer relation** and s the **inner relation** of the join
- Requires no indices and can be used with any kind of join condition
- Expensive since it examines every pair of tuples in the two relations

Nested-loop Join Example [1/5]

보조자료

Borrower relation R
: N = 16, B = 4

$R \bowtie_\theta S$

Customer name	Loan number
Jones	L - 170
Bahn	L – 82
Kim	L – 42
Lee	L – 48
Jane	L – 112
Smith	L – 34
Hwang	L – 321
Choi	L – 109
Pedro	L – 90
Sammy	L – 112
Jun	L – 31
Jung	L – 62
Shin	L – 99
Koh	L – 70
Mark	L – 221
Harry	L - 116

Loan relation S:
N = 12, B = 3

Loan number	Branch name	amount
L- 170	Downtown	300
L – 42	Redwood	400
L – 48	Redwood	1500
L – 112	Perryridge	2300
L – 321	Redwood	3100
L – 90	Downtown	800
L – 112	Perryridge	2300
L – 31	Redwood	200
L - 70	Perryridge	600
L - 221	Downtown	1000
L – 155	Redwood	800
L - 320	Downtown	2500

Nested-loop Join Example [2/5]

보조자료

Borrower relation R
: N = 16, B = 4

$R \bowtie_\theta S$

Loan relation S:
N = 12, B = 3

Customer name	Loan number
Jones	L - 170
Bahn	L – 82
Kim	L – 42
Lee	L – 48
Jane	L – 112
Smith	L – 34
Hwang	L – 321
Choi	L – 109
Pedro	L – 90
Sammy	L – 112
Jun	L – 31
Jung	L – 62
Shin	L – 99
Koh	L – 70
Mark	L – 221
Harry	L - 116

Loan number	Branch name	amount
L- 170	Downtown	300
L – 42	Redwood	400
L – 48	Redwood	1500
L – 112	Perryridge	2300
L – 321	Redwood	3100
L – 90	Downtown	800
L – 112	Perryridge	2300
L – 31	Redwood	200
L - 70	Perryridge	600
L - 221	Downtown	1000
L – 155	Redwood	800
L - 320	Downtown	2500

보조자료

Nested-loop Join Example [3/5]

Borrower relation R
: N = 16, B = 4

$R \bowtie_\theta S$

Loan relation S:
N = 12, B = 3

Customer name	Loan number
Jones	L - 170
Bahn	L – 82
Kim	L – 42
Lee	L – 48
Jane	L – 112
Smith	L – 34
Hwang	L – 321
Choi	L – 109
Pedro	L – 90
Sammy	L – 112
Jun	L – 31
Jung	L – 62
Shin	L – 99
Koh	L – 70
Mark	L – 221
Harry	L - 116

Loan number	Branch name	amount
L- 170	Downtown	300
L – 42	Redwood	400
L – 48	Redwood	1500
L – 112	Perryridge	2300
L – 321	Redwood	3100
L – 90	Downtown	800
L – 112	Perryridge	2300
L – 31	Redwood	200
L - 70	Perryridge	600
L - 221	Downtown	1000
L – 155	Redwood	800
L - 320	Downtown	2500

보조자료

Nested-loop Join Example [4/5]

Borrower relation R
: N = 16, B = 4

$R \bowtie_\theta S$

Loan relation S:
N = 12, B = 3

Customer name	Loan number
Jones	L - 170
Bahn	L – 82
Kim	L – 42
Lee	L – 48
Jane	L – 112
Smith	L – 34
Hwang	L – 321
Choi	L – 109
Pedro	L – 90
Sammy	L – 112
Jun	L – 31
Jung	L – 62
Shin	L – 99
Koh	L – 70
Mark	L – 221
Harry	L - 116

Loan number	Branch name	amount
L- 170	Downtown	300
L – 42	Redwood	400
L – 48	Redwood	1500
L – 112	Perryridge	2300
L – 321	Redwood	3100
L – 90	Downtown	800
L – 112	Perryridge	2300
L – 31	Redwood	200
L - 70	Perryridge	600
L - 221	Downtown	1000
L – 155	Redwood	800
L - 320	Downtown	2500

Nested-loop Join Example [5/5]

보조자료

Borrower relation R
: N = 16, B = 4

$R \bowtie_\theta S$

Loan relation S:
N = 12, B = 3

Customer name	Loan number
Jones	L - 170
Bahn	L – 82
Kim	L – 42
Lee	L – 48
Jane	L – 112
Smith	L – 34
Hwang	L – 321
Choi	L – 109
Pedro	L – 90
Sammy	L – 112
Jun	L – 31
Jung	L – 62
Shin	L – 99
Koh	L – 70
Mark	L – 221
Harry	L - 116

Loan number	Branch name	amount
L- 170	Downtown	300
L – 42	Redwood	400
L – 48	Redwood	1500
L – 112	Perryridge	2300
L – 321	Redwood	3100
L – 90	Downtown	800
L – 112	Perryridge	2300
L – 31	Redwood	200
L - 70	Perryridge	600
L - 221	Downtown	1000
L – 155	Redwood	800
L - 320	Downtown	2500

- Borrower relation을 outer relation으로 하면
  - 16*3 + 4 = **52회의 disk access가 필요**
- Loan relation을 outer relation으로 했을 경우
  - 12*4 + 3 = **51회의 disk access가 필요**

The Student Relation r

ID	name	dept_name	tot_cred
00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
19991	Brandt	History	80
23121	Chavez	Finance	110
44553	Peltier	Physics	56
45678	Levy	Physics	46
54321	Williams	Comp. Sci.	54
55739	Sanchez	Music	38
70557	Snow	Physics	0
76543	Brown	Comp. Sci.	58
76653	Aoi	Elec. Eng.	60
98765	Bourikas	Elec. Eng.	98
98988	Tanaka	Biology	120

The Takes relation s

ID	course_id	sec_id	semester	year	grade
00128	CS-101	1	Fall	2009	A
00128	CS-347	1	Fall	2009	A-
12345	CS-101	1	Fall	2009	C
12345	CS-190	2	Spring	2009	A
12345	CS-315	1	Spring	2010	A
12345	CS-347	1	Fall	2009	A
19991	HIS-351	1	Spring	2010	B
23121	FIN-201	1	Spring	2010	C+
44553	PHY-101	1	Fall	2009	B-
45678	CS-101	1	Fall	2009	F
45678	CS-101	1	Spring	2010	B+
45678	CS-319	1	Spring	2010	B
54321	CS-101	1	Fall	2009	A-
54321	CS-190	2	Spring	2009	B+
55739	MU-199	1	Spring	2010	A-
76543	CS-101	1	Fall	2009	A
76543	CS-319	2	Spring	2010	A
76653	EE-181	1	Spring	2009	C
98765	CS-101	1	Fall	2009	C-
98765	CS-315	1	Spring	2010	B
98988	BIO-101	1	Summer	2009	A
98988	BIO-301	1	Summer	2010	*null*

n_r 5000 Records
b_r 100 Blocks

$\bowtie$

n_s 10000 Records
b_s 400 Blocks

Cost: Nested-Loop Join

n_r 5000 Records
b_r 100 Blocks

⋈

n_s 10000 Records
b_s 400 Blocks

- In the worst case, if there is enough memory only to hold one block of each relation, the estimated cost is
  $n_r * b_s + b_r$ block transfers, plus $n_r + b_r$ seeks
  where n_r is number of records in R and b_s and b_r are number of disk blocks in S and R
  (relation s 를 n_r 번 읽어들여야 하므로 $n_r * b_s$ block transfers 와 n_r seeks 가 필요)
- If the smaller relation fits entirely in memory, use that as the inner relation
  - Reduces cost to $b_r + b_s$ block transfers and 2 seeks
- Assuming worst case memory availability cost estimate is
  - with *student* as the outer relation:
    - 2,000,100 (= 5000 * 400 + 100) block transfers and 5,100 (= 5000 + 100) seeks
  - with *takes* as the outer relation
    - 1,000,400 (= 10000 * 100 + 400) block transfers and 10,400 (= 10000 + 400) seeks
- If smaller relation (*student*) fits entirely in memory, the cost estimate will be 500 block transfers
- Block nested-loops algorithm (next slide) is preferable.

Block Nested-Loop Join [1/2]

- Variant of nested-loop join in which every block of inner relation is paired with every block of outer relation

```
for each block B_r of r do begin
    for each block B_s of s do begin
        for each tuple t_r in B_r do begin
            for each tuple t_s in B_s do begin
                Check if (t_r,t_s) satisfy the join condition
                if they do, add t_r • t_s to the result.
            end
        end
    end
end
```

- **Won Kim's Join Method**
  - One chapter of '80 PhD Thesis at Univ of Illinois at Urbana-Champaign

Block Nested-Loop Join: Example [1/4]

보조자료

Borrower relation R
: N = 16, B = 4

$R \bowtie_\theta S$

Loan relation S:
N = 12, B = 3

Customer name	Loan number
Jones	L - 170
Bahn	L – 82
Kim	L – 42
Lee	L – 48
Jane	L – 112
Smith	L – 34
Hwang	L – 321
Choi	L – 109
Pedro	L – 90
Sammy	L – 112
Jun	L – 31
Jung	L – 62
Shin	L – 99
Koh	L – 70
Mark	L – 221
Harry	L - 116

Loan number	Branch name	amount
L- 170	Downtown	300
L – 42	Redwood	400
L – 48	Redwood	1500
L – 112	Perryridge	2300
L – 321	Redwood	3100
L – 90	Downtown	800
L – 112	Perryridge	2300
L – 31	Redwood	200
L - 70	Perryridge	600
L - 221	Downtown	1000
L – 155	Redwood	800
L - 320	Downtown	2500

Block Nested-Loop Join Example [2/4]

보조자료

Borrower relation

Customer name	Loan number
Jones	L - 170
Bahn	L – 82
Kim	L – 42
Lee	L – 48
Jane	L – 112
Smith	L – 34
Hwang	L – 321
Choi	L – 109
Pedro	L – 90
Sammy	L – 112
Jun	L – 31
Jung	L – 62
Shin	L – 99
Koh	L – 70
Mark	L – 221
Harry	L - 116

Loan relation

Loan number	Branch name	amount
L- 170	Downtown	300
L – 42	Redwood	400
L – 48	Redwood	1500
L – 112	Perryridge	2300
L – 321	Redwood	3100
L – 90	Downtown	800
L – 112	Perryridge	2300
L – 31	Redwood	200
L - 70	Perryridge	600
L - 221	Downtown	1000
L – 155	Redwood	800
L - 320	Downtown	2500

Block Nested-Loop Join Example [3/4]

보조자료

Borrower relation R
: N = 16, B = 4

$R \bowtie_\theta S$

Loan relation S:
N = 12, B = 3

Customer name	Loan number
Jones	L - 170
Bahn	L – 82
Kim	L – 42
Lee	L – 48
Jane	L – 112
Smith	L – 34
Hwang	L – 321
Choi	L – 109
Pedro	L – 90
Sammy	L – 112
Jun	L – 31
Jung	L – 62
Shin	L – 99
Koh	L – 70
Mark	L – 221
Harry	L - 116

Loan number	Branch name	amount
L- 170	Downtown	300
L – 42	Redwood	400
L – 48	Redwood	1500
L – 112	Perryridge	2300
L – 321	Redwood	3100
L – 90	Downtown	800
L – 112	Perryridge	2300
L – 31	Redwood	200
L - 70	Perryridge	600
L - 221	Downtown	1000
L – 155	Redwood	800
L - 320	Downtown	2500

Block Nested-Loop Join Example [4/4]

보조자료

Borrower relation R
: N = 16, B = 4

$R \bowtie_\theta S$

Loan relation S:
N = 12, B = 3

Customer name	Loan number
Jones	L - 170
Bahn	L – 82
Kim	L – 42
Lee	L – 48
Jane	L – 112
Smith	L – 34
Hwang	L – 321
Choi	L – 109
Pedro	L – 90
Sammy	L – 112
Jun	L – 31
Jung	L – 62
Shin	L – 99
Koh	L – 70
Mark	L – 221
Harry	L - 116

Loan number	Branch name	amount
L- 170	Downtown	300
L – 42	Redwood	400
L – 48	Redwood	1500
L – 112	Perryridge	2300
L – 321	Redwood	3100
L – 90	Downtown	800
L – 112	Perryridge	2300
L – 31	Redwood	200
L - 70	Perryridge	600
L - 221	Downtown	1000
L – 155	Redwood	800
L - 320	Downtown	2500

- borrower relation을 outer relation으로 하면
  - 4*3 + 4 = **16회의 disk access**가 필요
- Loan relation을 outer relation으로 했을 경우
  - 3*4 + 3 = **15회의 disk access**가 필요

Block Nested-Loop Join [2/2]

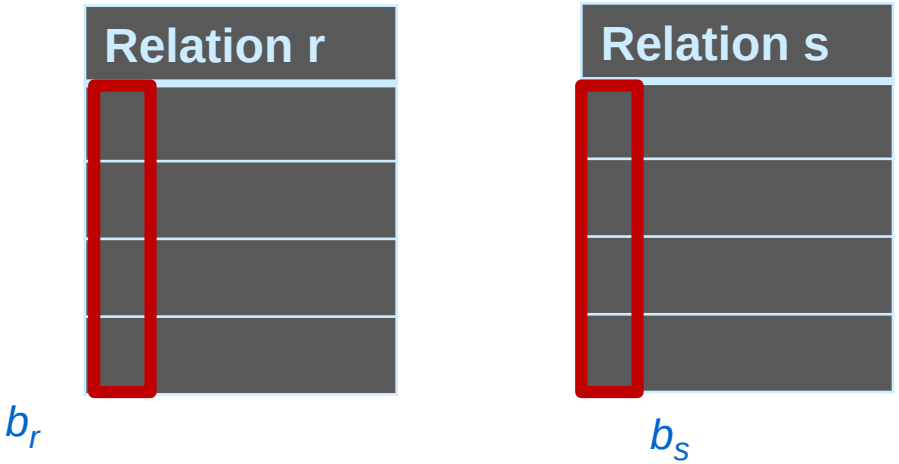
- Worst case estimate: $b_r * b_s + b_r$ block transfers + $2 * b_r$ seeks
  - Each block in the inner relation *s* is read once for each *block* in the outer relation

    * relation s 를 b_r 번 읽어들여야 하므로 $b_r * b_s$ block transfers 와 b_r seeks 가 필요

    * relation r 를 *1* 번 읽어들여야 하므로 b_r block transfers 와 b_r seeks 가 필요

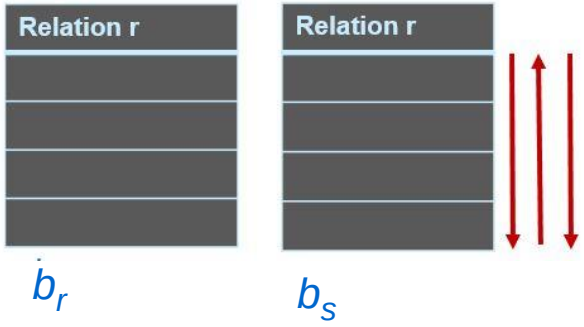
- 기왕이면 Smaller relation을 outer relation으로 하는것이 Block IO를 줄일수 있다
- If the inner relation fit in the memory,

  Best case estimate: $b_r + b_s$ block transfers + 2 seeks



Improvements to Nested-loop Join and Block Nested-loop Join

- If equi-join attribute forms a key of inner relation, stop inner loop on first match
- Scan inner loop forward and backward alternately, to make use of the blocks remaining in buffer (with OS LRU replacement)
  - Block1 → Block2 → Block3 → Block4 → Block3 → Block2 → Block1



- In block nested-loop join, (where *M* = memory size in blocks)
  - use *M* - 2 disk blocks as blocking unit for outer relations, use remaining 2 blocks to buffer inner relation and output
  - Improved Cost = $\lceil b_r / (M-2) \rceil * b_s + b_r$ block transfers $+ 2 \lceil b_r / (M-2) \rceil$ seeks
- In nested-loop join,
  - if an index is available on the inner relation's join attribute, index lookup may outperform inner relation scan (next slide)

Indexed Nested-Loop Join

- Index lookups can replace file scans if
  - join is an equi-join or natural join and
  - an index is available on the inner relation's join attribute
    - Can construct an index just to compute a join
- For each tuple t_r in the outer relation *r,* use the index (B+ tree) to look up tuples in *s* that satisfy the join condition with tuple t_r
- Worst case: buffer has space for only one page of *r*, and, for each tuple in *r*, we perform an index lookup on *s*
- Cost of the join: $b_r\,(t_T + t_S) + n_r * c$
  - Where *c* is the cost of traversing index and fetching all matching *s* tuples for one tuple or *r*
  - *c* can be estimated as cost of a single selection on *s* using the join condition
- If indices are available on join attributes of both *r* and *s,* use the relation with fewer tuples as the outer relation

Indexed Nested-Loop Join Example

보조자료

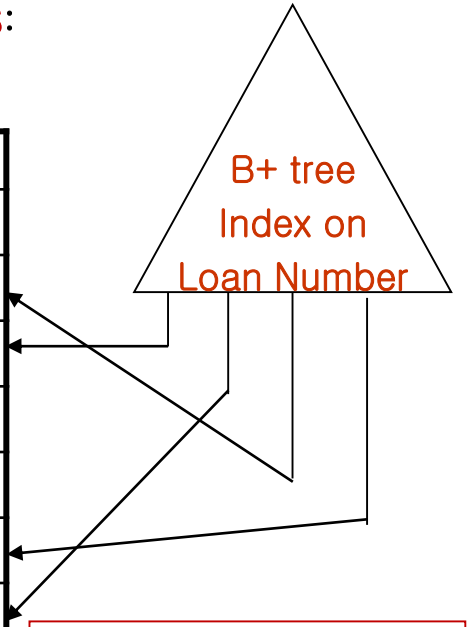
Borrower relation R
: N = 16, B = 4

$R \bowtie_\theta S$

Loan relation S:
N = 12, B = 3

Customer name	Loan number
Jones	L - 170
Bahn	L – 82
Kim	L – 42
Lee	L – 48
Jane	L – 112
Smith	L – 34
Hwang	L – 321
Choi	L – 109
Pedro	L – 90
Sammy	L – 112
Jun	L – 31
Jung	L – 62
Shin	L – 99
Koh	L – 70
Mark	L – 221
Harry	L - 116

Loan number	Branch name	amount
L- 170	Downtown	300
L – 42	Redwood	400
L – 48	Redwood	1500
L – 112	Perryridge	2300
L – 321	Redwood	3100
L – 90	Downtown	800
L – 112	Perryridge	2300
L – 31	Redwood	200
L - 70	Perryridge	600
L - 221	Downtown	1000
L – 155	Redwood	800
L - 320	Downtown	2500



Inner relation쪽 Join partner를 Index Scan으로 찾음.
Not file scan!

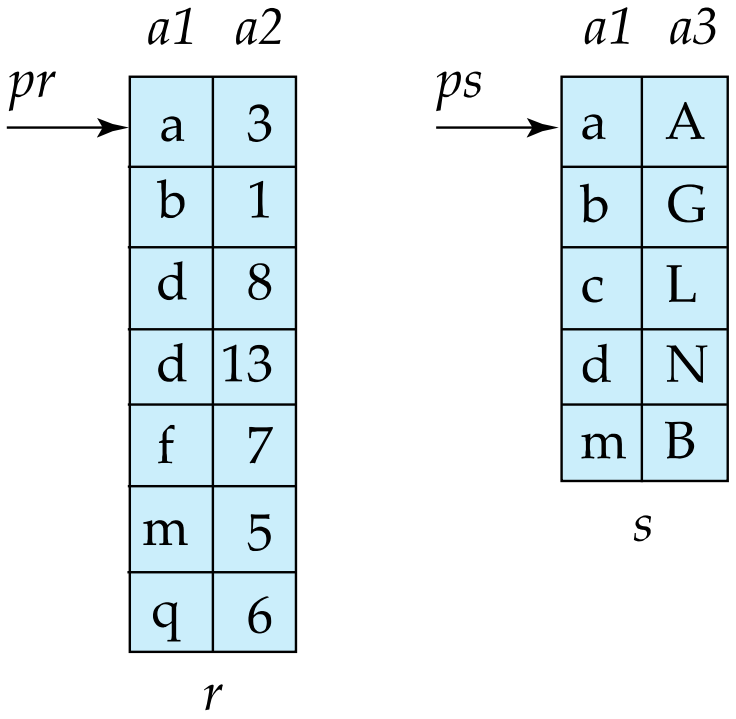
- Borrower relation의 16개의 tuple에 대해 각각 그 tuple과 join할 수 있는 loan relation의 tuple을 B+tree로 검색 (3회의 disk access)
  - Tree의 height가 2, 그리고, 실제의 data access를 위한 1회, 따라서, 3회의 disk access 필요
  - 따라서, 총 cost는 4 block access +16*3 block access = 43회의 disk access필요

Cost Example: Indexed Nested-Loop Join

- Compute *student* ⋈ *takes*, with *student* as the outer relation.
  - Let *takes* have a primary B^+-tree index on the attribute *ID,* which contains 20 entries in each index node.
  - Since *takes* has 10,000 tuples, the height of the tree is 4, and one more access is needed to find the actual data
  - *student* has 5000 tuples
- Cost of block nested-loop join
  - 40,100 (= 400*100 + 100) block transfers + 200 (= 2 * 100) seeks
    - assuming worst case memory
    - may be significantly less with more memory
- Cost of indexed nested-loop join
  - 25,100 (= 100 + 5000 * 5) block transfers and seeks
  - CPU cost likely to be less than that for block nested loops join

Merge-Join [1/2]

1. Sort both relations on their join attribute (if not already sorted on the join attributes)
2. Merge the sorted relations to join them
    1. Join step is similar to the merge stage of the sort-merge algorithm.
    2. Main difference is handling of duplicate values in join attribute — every pair with same value on join attribute must be matched
    3. Detailed algorithm in book



Join attribute 기준으로 sorted된
relation 2개를 Join하는것은 대박!

지금부터는 Natural Join과
EquiJoin에만 성립되는 Algorithm

Merge Join Example
on The *borrower and loan* Relation

보조자료

customer-name	*loan-number*
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17

loan-number	*branch-name*	*amount*
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500

After Sorting Two Relations

Custom-name	Loan - number	Loan-number	Branch-name	amount
Smith	L-11	L-11	Round Hill	900
Jackson	L-14	L-14	Downtown	1500
Hayes	L-15	L-15	Perryridge	1500
Adams	L-16	L-16	Perryridge	1300
Jones	L-17	L-17	Downtown	1000
Williams	L-17			
Smith	L-23	L-23	Redwood	2000
Curry	L-93	L-93	Mianus	500

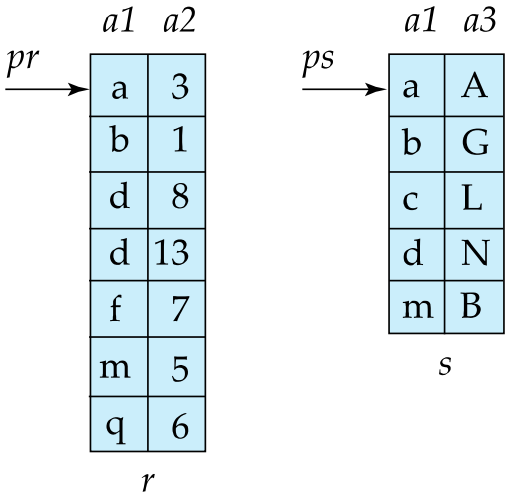
Merge-Join [2/2]

- Can be used only for equi-joins and natural joins
- Each block needs to be read only once (assuming all tuples for any given value of the join attributes fit in memory
- Thus the cost of merge join is: (b_b buffer blocks are assigned to each relation)

$b_r + b_s$ block transfers

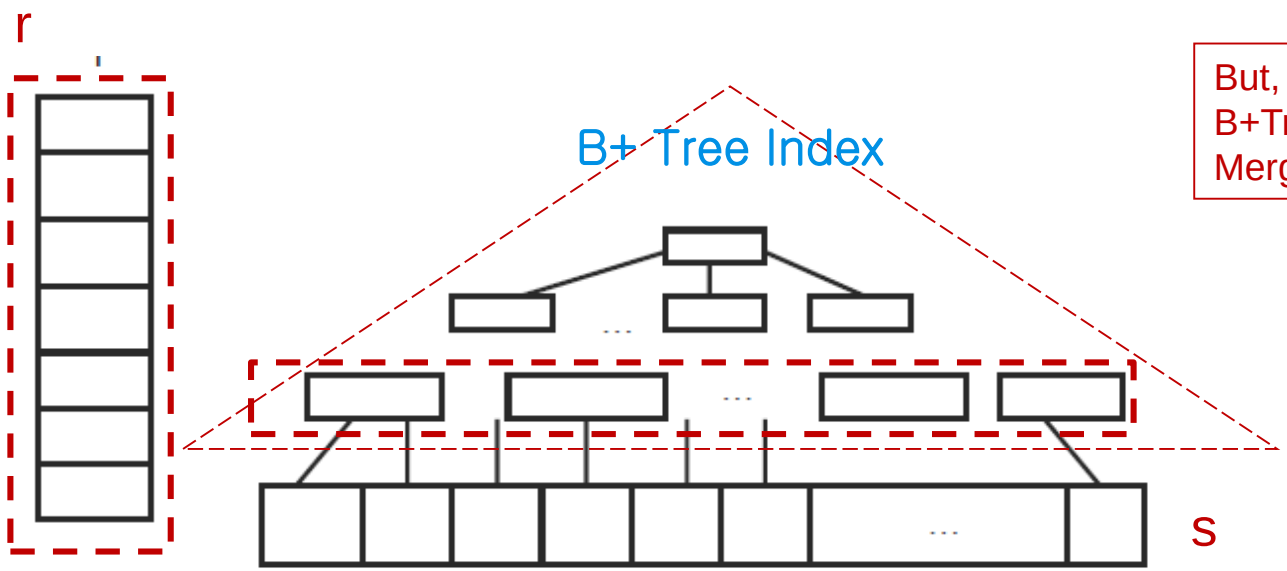
$+ \lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil$ seeks

+ the cost of sorting if relations are unsorted



Hybrid Merge-Join

- **Join의 2개 relation중에 1개만 sorted이고 1개는 unsorted & B+tree일때 Merge-Join를 확대**
- If one relation r is sorted, and the other relation s has a secondary B^+-tree index on the join attribute
  - Merge the sorted relation r with the leaf entries of the B^+-tree in s
    - Using the order property of the leaf entries of the B^+- tree
  - Sort the result on the addresses of the unsorted relation s's tuples
  - Scan the unsorted relation in physical address order and merge with previous result, to replace addresses by the actual tuples
    - Sequential scan more efficient than random lookup



But, 양쪽 relation에 secondary B+Tree Index를 이용한 Hybrid Merge-Join은 성능에 문제!

Hash-Join

- Applicable for equi-joins and natural joins (not for theta joins)
- A hash function *h* is used to partition tuples of both relations
- *The hash function h* maps *JoinAttrs* values to {0, 1, ..., *n*}, where *JoinAttrs* denotes the common attributes of *r* and *s* used in the natural join
  - $R_0, R_1, \ldots, R_n$ denote partitions of *r* tuples
    - Each tuple $t_r \in r$ is put in partition r_i where $i = h(t_r[JoinAttrs])$.
  - $S_0, S_1 \ldots, S_n$ denotes partitions of *s* tuples
    - Each tuple $t_s \in s$ is put in partition s_i, where $i = h(t_s[JoinAttrs])$
- 그러나 $R_0, R_1, \ldots, R_n$ & $S_0, S_1 \ldots, S_n$들의 내부에서는 sort가 전혀 안된 상태
- *Note:* In 559 page of the text, R_i is denoted as $H_{ri,}$ S_i is denoted as H_{si} and *n* is denoted as n_h

Hash Join

참고자료

- Case 1: Smaller relation *(S)* fits in memory
- Hash join:

  read S in memory and build a hash index on it

  for each tuple r in R

  use the hash index on S to find tuples such that S.a = r.a

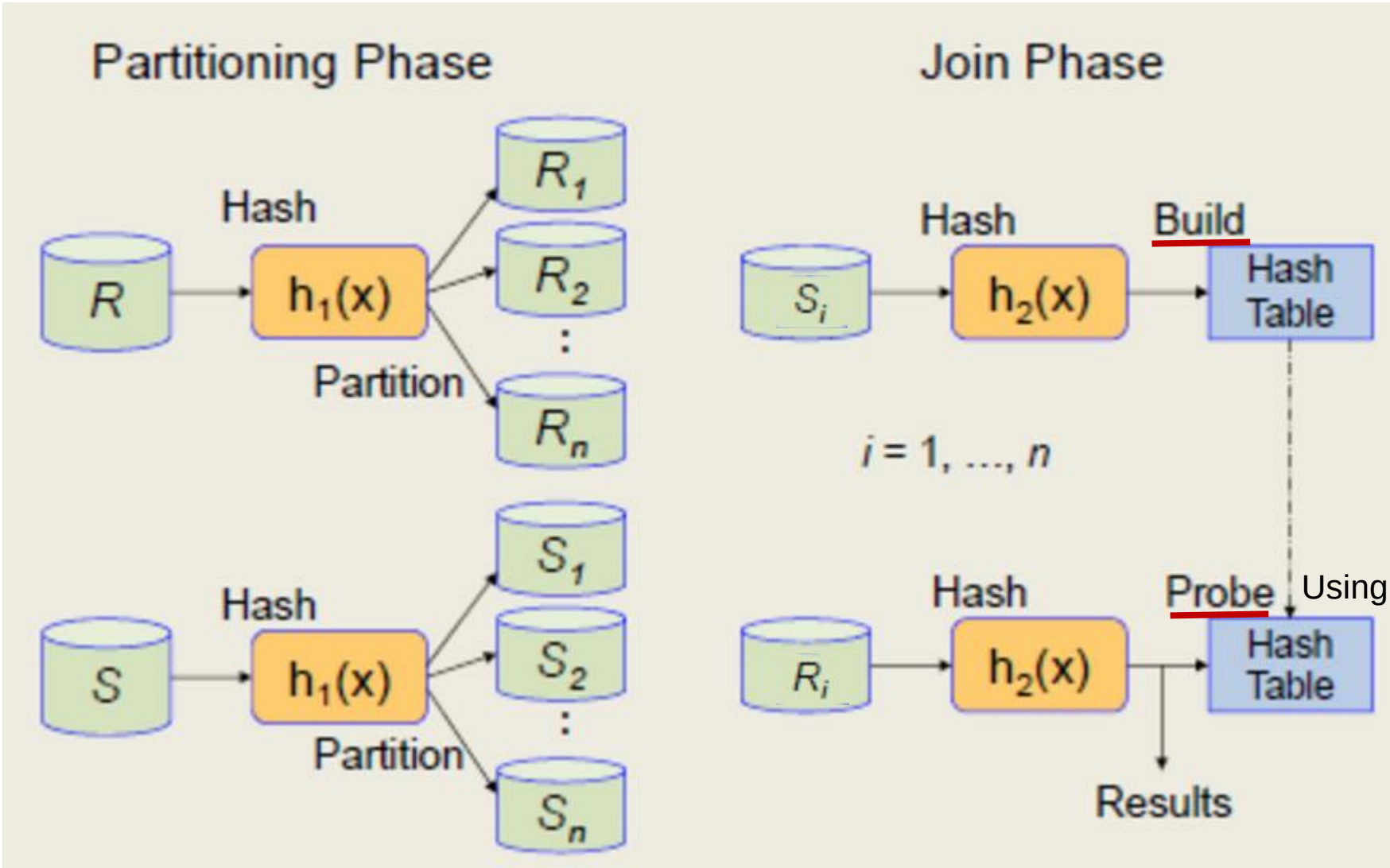
- Why good ?
  - CPU cost is much better (even though we don't care about it too much)

참고자료

Hash Join

- Case 2: Smaller relation *(S)* doesn't fit in memory
- Phase 1: (Partition Phase)
  - Read the relation *R* block by block and partition it using a hash function, *h1(a)*
    - Create one partition for each possible value of *h1(a)*
  - Write the partitions to disk (R gets partitioned into R1, R2, …, 까)
  - Similarly, read and partition S, and write partitions S1, S2, …, Sk to disk
  - Only requirement: Each S partition fits in memory
- Phase 2: (Build & Probe Phase)
  - Read S1 into memory, and build a hash index on it (S1 fits in memory)
    - Using a different hash function, $h_2(a)$
  - Read R1 block by block, and use the hash index to find matches
  - Repeat for S2, R2, and so on

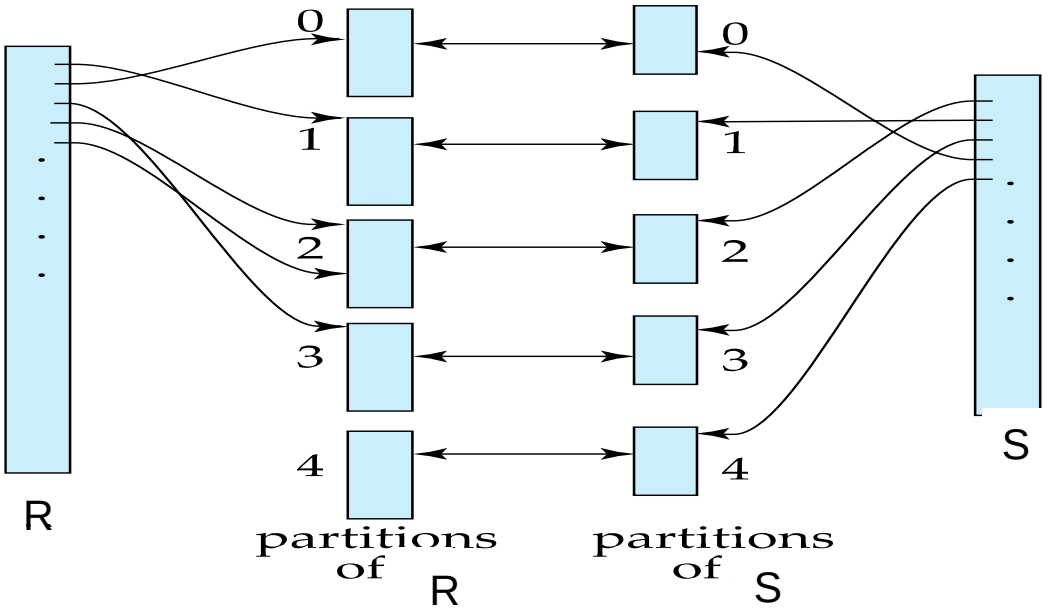
Diagram of Hash-Join



Hash-Join: Partition Phase

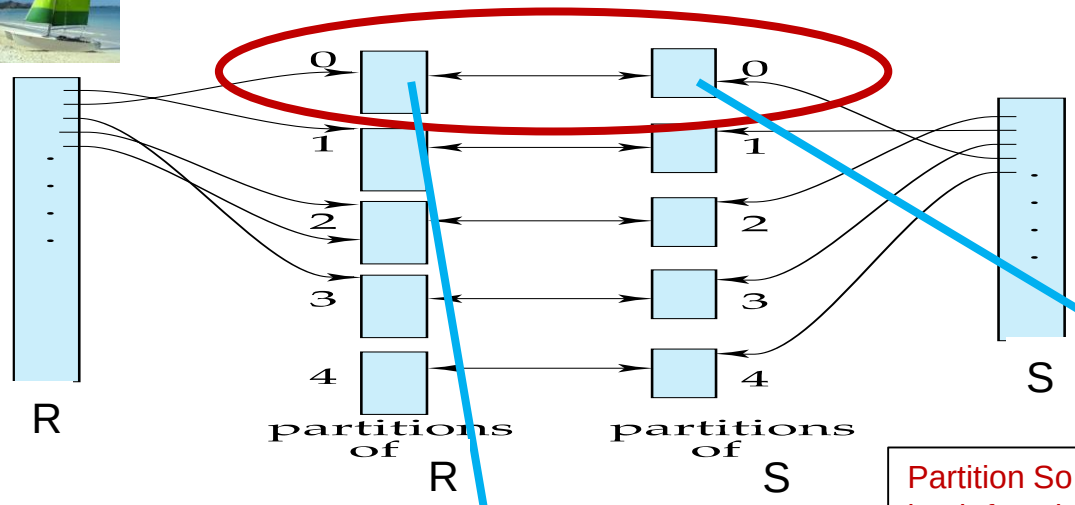
1st Hash Function을 적용하여
N개로 Partitioning하고 Disk에 저장함

S_i 는 main memory buffer에 fit하는것이 중요하고,
R_i 도 가급적 fit하면 좋고!,



- *T*uples in R_i need only to be compared with *t*uples in S_i
  - Need not be compared with *s* tuples in any other partition, since:
    - an *R* tuple and an *S* tuple that satisfy the join condition will have the same value for the join attributes
    - If that value is hashed to some value *i*, the *R* tuple has to be in R_i and the *S* tuple in S_i

Hash-Join: Build & Probe Phase



2nd Hash Function을 적용하여
S partition들에 대해서
In-memory Hash Index를 build하고
R partition들에서 probe을 시행

Partition So에 another hash function h2를 적용

Partition Ro에 있는 record 별로 probe 진행

Partition Ro
r_rec112
r_rec22
r_rec543
…….

Join-attr 값	h2 적용 값	Pointer to record
xyz	0	
www	1	
ppp	2	
zxc	3	
…..		

Partition So
s_rec23
s_rec287
s_rec14
…….

Hash Index에 overflow가 없는것이 바람직!

Build In-Memory Hash Index for So

- Probing: r_rec112 에 join-attribute의 값을 보고 2nd Hash function을 적용한값을 In-memory Hash Index에서 찾아본다

Hash-Join Algorithm [559 page]

1. (Partitioning Phase)
   - Partition the relation *s* using hashing function *h1*
     - When partitioning a relation, one block of memory is reserved as the output buffer for each partition.
   - Partition the relation *r* similarly
2. For each *i*: *(Build & Probe Phase)*
   - Load s_i into memory and build an in-memory hash index on it using the join attribute
     - This hash index uses a different hash function *h2* than the earlier one *h1*
   - Read the tuples in r_i from the disk one by one
     - For each tuple t_r
       - locate each matching tuple t_s in s_i using the in-memory hash index
   - Output the concatenation of their attributes

- Hash function이 적용되는 relation s ➔ build input (or build relation)

  Probing되는 relation r ➔ probe input (or probe relation)

Recursive Partitioning in Hash-Join

- The value *n* and the hash function *h* is chosen such that each S_i should fit in memory
  - Typically n is chosen as $\lceil b_s / M \rceil * f$
    - where M is the number of memory buffers, f is a "**fudge factor**" (~~ 1.2)
  - The probe relation partitions R_i need not fit in memory

- **Recursive partitioning** is required until each partition of the build relation S fits in memory
  - Instead of partitioning *n* ways, use *M* – 1 partitions for S using an initial hash function (M-1개의 memory block를 fully 잘 이용하기 위해서)
  - 첫단계에서 만들어진 partition들중에 Memory에 fit하지 않는 경우에는
    - Further partition using a different hash function
  - Rarely required
    - e.g., with block size of 4 KB, recursive partitioning not needed
      - for relations of < 1GB with memory size of 2MB
      - for relations of < 36 GB with memory size of 12 MB

S_i 를 main memory에 동시에 올려놓고 hash table을 만들어서 처리를 해야 함!

Handling of Hash-Table Overflows in the Build-Phase of Hash Join

- Partitioning is said to be **skewed** if some partitions have significantly more tuples than some others
- Hash-table is made in the build-phase of partition S_i
- **Hash-table overflow** occurs if S_i is the skewed partition
  - Reasons could be
    - Many tuples in S_i with same value for join attributes
    - Bad hash function
- **Overflow resolution** can be done in the build phase
  - Partition S_i is further partitioned using a different hash function (Rehashing)
  - Partition R_i must be similarly partitioned
- **Overflow avoidance** performs partitioning carefully to avoid overflows during the build phase
  - E.g. initially partition build relation into many small partitions, then combine them
- Both approaches fail with large numbers of duplicates in S_i
  - Fallback option: S_i 에 대해서 hash-table을 만들지 않고, use block nested loops join on overflowed partitions

S_i 들은 memory에 fit하도록 recursive partitioning으로 partition했지만, S_i 안에 있는 record들의 분포가 나쁘면 (혹은 second hash function과 안 맞으면) Build-Phase에 생기는 Hash-Table에 Overflow!

Cost of Hash-Join

- If recursive partitioning is not required: cost of hash join is

  $3^* (b_r + b_s) + 4 * n_h$ block transfers + $2 * (\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil)$ seeks

- If recursive partitioning required:
  - number of passes required for partitioning build relation s is $\lceil log_{M-1}(b_s) - 1 \rceil$
  - best to choose the smaller relation as the build relation
  - Total cost estimate is:

    $2 * (b_r + b_s) * \lceil log_{M-1}(b_s) - 1 \rceil + b_r + b_s$ block transfers +

    $2 * (\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil) * \lceil log_{M-1}(b_s) - 1 \rceil$ seeks

- If the entire build input can be kept in main memory, no partitioning is required
  - Cost estimate goes down to $b_r + b_s$

Example of Hash-Join Cost

instructor ⋈ *teaches*

- Assume that memory size is 20 blocks
- $b_{instructor}$= 100 and $b_{teaches}$ = 400.
- *instructor* is to be used as build input
  - Partition it into five partitions, each of size 20 blocks
  - This partitioning can be done in one pass
- Similarly, partition *teaches* into five partitions, each of size 80
  - This is also done in one pass

- Therefore total cost, ignoring cost of writing partially filled blocks:
  - 1500 (= 3*(100 + 400)) block transfers + 336 (= 2*(⌈100/3⌉ + ⌈400/3⌉)) seeks

Hybrid Hash Join [1/2]

- Motivation:
  - R = 10000 blocks, S = 101 blocks, M = 100 blocks
  - So *S* doesn't fit in memory
- Phase 1: (Partitioning in Regular Hash Join)
  - Use 2 partitions
    - Read 10000 blocks of R, write partitions R1 and R2 to disk
    - Read 101 blocks of S, write partitions S1 and S2 to disk
  - Only need 3 blocks for this (so remaining 97 blocks are being wasted)
- Phase 2: (Build & Probe in Regular Hash Join)
  - Read S1, build hash index, read R1 and probe
  - Read S2, build hash index, read R2 and probe
- Alternative:
  - Don't write partition *S1* to disk, just keep it memory – there is enough free memory for that

Hybrid Hash Join [2/2]

- Motivation:
  - R = 10000 blocks, S = 101 blocks, M = 100 blocks
  - So *S* doesn't fit in memory
- Alternative: Don't write partition *S1* to disk, just keep it memory
- Steps:
  - Use a hash function such that *S1 = 90 blocks, and S2 = 10 blocks*
  - Partitioning S: Keep S1 in memory, and build a hash table on it; Write only S2 to disk
  - Probe R using the hash table
- Saves huge amounts of I/O

- Useful when memory sizes are relatively large, and the build input is bigger than memory
- Main feature of hybrid hash join:
  Keep the first partition of the build relation in memory after partitioning
- 20% performance gain than the regular hash join

Complex Joins

- Join with a conjunctive condition: $r \bowtie_{\theta 1 \wedge \theta 2 \wedge \ldots \wedge \theta n} s$
  - Compute the result of one of these simpler joins $r \bowtie_{\theta i} s$
  - The final result comprises those tuples in the intermediate result that satisfy the remaining conditions

$$\theta_1 \wedge \ldots \wedge \theta_{i-1} \wedge \theta_{i+1} \wedge \ldots \wedge \theta_n$$

  - The cost is the cost of the chosen join

- Join with a disjunctive condition: $r \bowtie_{\theta 1 \vee \theta 2 \vee \ldots \vee \theta n} s$
  - Compute as the union of the records in individual joins $r \bowtie_{\theta i} s$:

$$(r \bowtie_{\theta 1} s) \cup (r \bowtie_{\theta 2} s) \cup \ldots \cup (r \bowtie_{\theta n} s)$$

  - The cost is the addition of the cost of all joins

- Nested-Loop Join and its variants are working for any type of join operations
- Other efficient join methods (such as Merge-Join, Hash-Join, etc) are working only for natural joins and equi-joins

Chapter 12: Query Processing

- 12.1 Overview
- 12.2 Measures of Query Cost
- 12.3 Selection Operation
- 12.4 Sorting
- 12.5 Join Operation
- 12.6 Other Operations
- 12.7 Evaluation of Expressions

Other Operations

- **Duplicate elimination** (in a big relation)
  - SQL query may have "unique" option
  - Duplicate elimination can be implemented via hashing or sorting
  - After sorting a file using external sort-merge, duplicates will come adjacent to each other, and all but one set of duplicates can be deleted
    - *Optimization:* duplicates can be deleted during run generation as well as at intermediate merge steps in external sort-merge
    - Cost: sorting cost
  - Hashing is similar – duplicates will come into the same bucket
    - Cost: hashing cost

- **Projection:**
  - perform projection on each tuple followed by duplicate elimination
  - Cost: Scan a relation

Other Operations : Set Operations [1/2]

- $r \cup s$ $r \cap s$ $r - s$

- Plain Method
  - Sort the relations R and S
  - Scan R and S

- Cost: $b_r + b_s$ block transfers if the relations are sorted

- If assuming a worst case of one block buffer for each relation
  Cost: $b_r + b_s$ block transfers + $b_r + b_s$ seeks

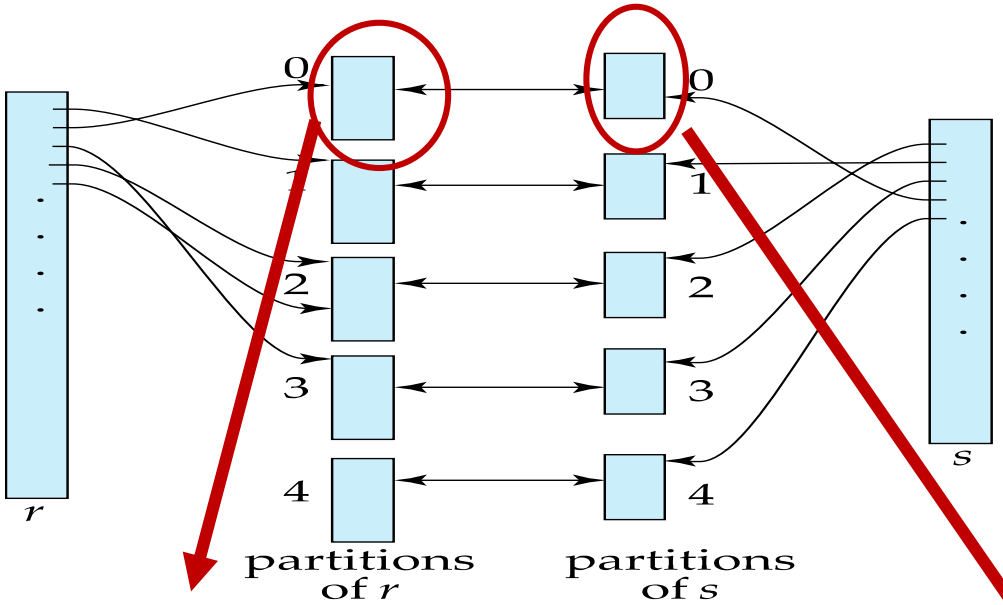
- If the relations are not sorted, sorting cost would be added

Other Operations : Set Operations [2/2]

- Using variant of merge-join after sorting, or variant of hash-join (Cost도 similar)
- E.g., Set operations using hashing:
  1. Partition both relations using the same hash function, thereby creating $r_0, r_1, .., r_n$ and $s_0, s_1, .., s_n$
  2. Process each partition *i* as follows.
     1. Using a different hashing function, build an in-memory hash index on r_i after it is brought into memory
     2. Process s_i as follows
        - $r \cup s$:
          1. Add tuples in s_i to the hash index only if they are not already in it
          2. Add the tuples in the hash index to the result
        - $r \cap s$:
          1. For each tuple in s_i, Probe the hash index and output tuples in s_i to the result only if they are already there in the hash index
        - $r - s$:
          1. For each tuple in s_i, Probe the hash index and if it is there in the hash index, delete it from the hash index
          2. Add remaining tuples in the hash index to the result.

Example: Set Operation *r* – *s* using Partition-Hashing

H1을 이용한 Partitioning은 앞에 있는 Hash-Join과 동일



Partition Ro에 another hash function H2를 적용

In-Memory Hash Index

임의의 attribute A	h2 적용 값	Pointer to record
xyz	0	
www	1	
ppp	2	
zxc	3	
…..		

Partition Ro
r_rec23
r_rec287
r_rec14
…….

So에 있는 tuple들의 A attribute값을 보고 Ro의 hash index에 있으면 Result_set에서 해당 record들 삭제

Partition Ro를 memory로 read하고 Result_Set으로 이름변경

Other Operations : Outer Join [1/2]

- Compute $r \mathbin{⟕} s$
  - First compute $r \bowtie s$ and save it as a temporary relation q1
  - Compute non participating tuples T are those in $r - \Pi_R(r \bowtie s)$ using projection, join, and set difference
  - Pad the tuples in T with nulls for attributes in s
  - Add the padded tuples to q1
- Right outer-join and full outer-join can be computed similarly
- Relation *instructor*

ID	*name*	*dept_name*
10101	Srinivasan	Comp. Sci.
12121	Wu	Finance
15151	Mozart	Music

- Relation *teaches*

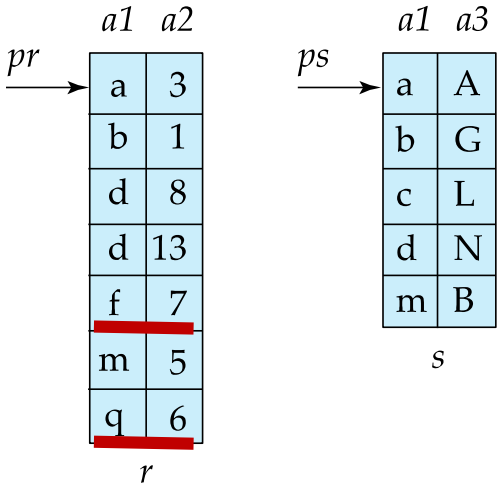
ID	*course_id*
10101	CS-101
12121	FIN-201
76766	BIO-101

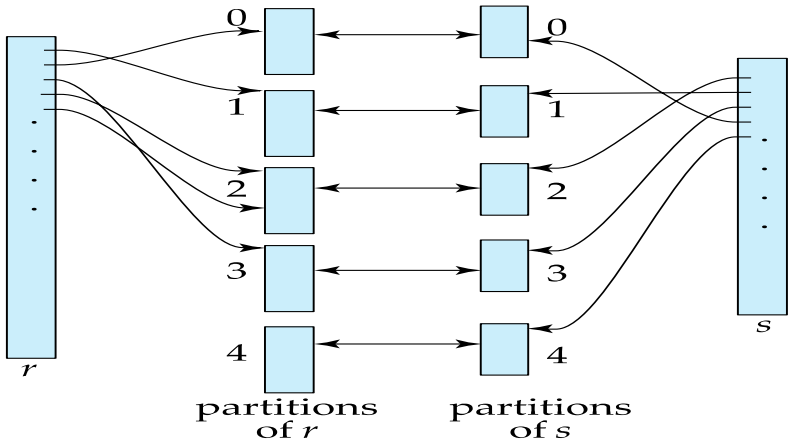
  - Left Outer Join: *instructor* $\mathbin{⟕}$ *teaches*

ID	*name*	*dept_name*	*course_id*
10101	Srinivasan	Comp. Sci.	CS-101
12121	Wu	Finance	FIN-201
15151	Mozart	Music	*null*

Other Operations : Outer Join [2/2]

- Outer Join by modifying the merge-join and hash-join algorithms (Cost도 similar)
- Modifying merge join to compute $r \mathbin{⟕} s$ *(left outer-join)*
  - In $r \mathbin{⟕} s$, non participating tuples are those in $r - \Pi_R(r \bowtie s)$
  - Modify merge-join to compute $r \mathbin{⟕} s$:
    - During merging, for every tuple t_r from r that do not match any tuple in s, output t_r padded with nulls to the result
- Modifying hash join to compute $r \mathbin{⟕} s$
  - If r is a probe relation, output non-matching r tuples padded with nulls to the result
  - If r is a build relation, when probing keep track of which r tuples matched s tuples, output non-matched r tuples padded with nulls at the end of s





Other Operations : Aggregation

- **Aggregation**: count, min, max, sum, avg (in a big relation)

- can be implemented in a manner similar to duplicate elimination
  - Sorting or hashing can be used to bring tuples in the same group together, and then the aggregate functions can be applied on each group
  - Cost도 duplicate elimination과 similar
  - *Optimization:* combine tuples in the same group during run-generation and intermediate merges, by computing partial aggregate values
    - For count, min, max, sum:
      - keep aggregate values on tuples found so far in the group
      - When combining partial aggregate for count, add up the aggregates
    - For avg:
      - keep sum and count values, and divide sum by count at the end

skip

Chapter 12: Query Processing

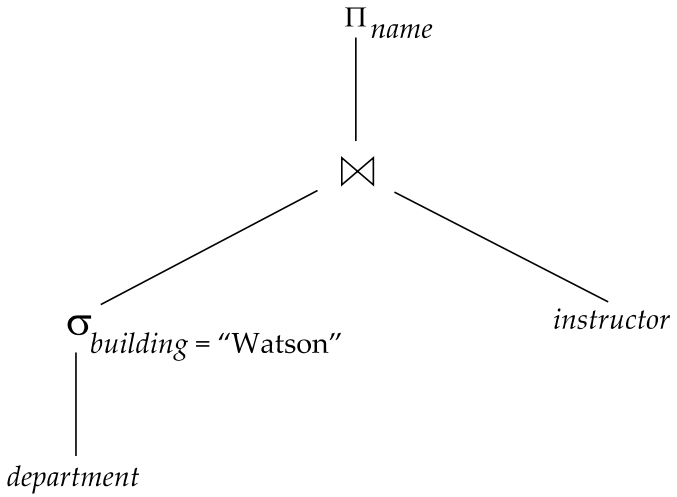
- 12.1 Overview
- 12.2 Measures of Query Cost
- 12.3 Selection Operation
- 12.4 Sorting
- 12.5 Join Operation
- 12.6 Other Operations
- 12.7 Evaluation of Expressions

Evaluation of Expressions

- So far, we have seen algorithms for individual operations
- Now, evaluate an expression containing multiple operations
- Alternatives for evaluating an entire expression tree
  - **Materialization**: (storing temporary relations)
    - generate results of an expression whose inputs are relations or are already computed,
    - **materialize** (store) it on disk.
    - repeat
  - **Pipelining**:
    - Pass on tuples to parent operations even as an operation is being executed
    - Toward Parallel Execution
    - 앞에서 배웠던 operation들을 pipeline version으로 바꿔야 함
      - Well-matched with pipelining vs Not-matched with pipelining

Materialization [1/2]

- **Materialized evaluation** is always applicable
  - Evaluate one operation at a time, starting at the lowest-level
  - Materialize intermediate results into temporary relations to evaluate next-level operations
- E.g., For the query $\prod_{name}(\sigma_{building="Watson"}(department) \bowtie instructor)$
  - Compute $\sigma_{building="Watson"}(department)$ and store the intermediate result
  - then compute the join operation with *instructor & the intermediate result,* and store *the intermediate result*
  - and finally compute the projection on *name with the intermediate result*



Materialization [2/2]

- The cost formulas for single operations ignore the cost of writing results to disk
- But materialization must consider writing the temporary relations to disk
- Cost of writing results to disk and reading them back can be quite high
  - Overall cost = costs of individual operations +
    cost of writing intermediate results to disk
- **Double buffering**: (use 2 output buffers for each operation)
  - One small step for parallelism (doing computation while block-IO)
  - When one is full, write it to disk while the other is getting filled
  - Allows overlap of disk writes with computation
  - Reduces execution time

Pipelining

- **We can improve query evaluation efficiency by reducing the number of temporary files that are produced**

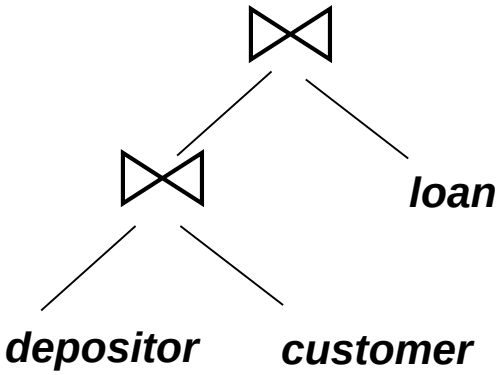
- **Pipelined evaluation :**
  - evaluate several operations simultaneously
  - passing the results of one operation on to the next
- E.g., in previous expression tree, don't store result of $\sigma_{building="\text{Watson}"}(department)$
  - instead, pass tuples directly to the join
  - Similarly, don't store result of join, pass tuples directly to projection

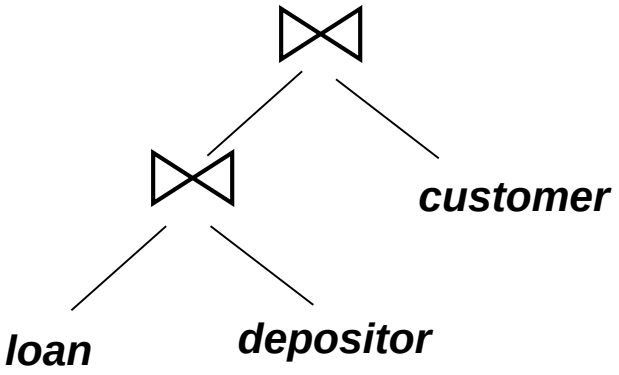
- Much cheaper than materialization: no need to store a temporary relation to disk
- Pipelining may not always be possible – e.g., sort
- For pipelining to be effective, need to invent the evaluation algorithms that generate output tuples even as input tuples are received to the operation
- Pipelines can be executed in two ways:
  - **Demand-driven pipeline**
  - **Producer-driven pipeline**

Processing Multiple Joins

- Join involving three relations: *loan* ⋈ *depositor* ⋈ *customer*
- **Strategy 1.**
  - Compute *depositor* ⋈ *customer* first, and then join the result with *loan*
- **Strategy 2.**
  - Computer *loan* ⋈ *depositor* first, and then join the result with *customer*
- **Strategy 3.**
  - Perform the pair of joins (*loan* ⋈ *depositor*, *depositor* ⋈ *customer*) at once
  - Build an index on *loan* for *loan-number,* and an index on *customer* for *customer-name*
    - For each tuple *t* in *depositor*, look up the corresponding tuples in *customer* and the corresponding tuples in *loan*
    - Each tuple of *depositor* is examined exactly once
- Strategy 1 & 2 may use materialization or pipelining
- Strategy 3 combines two operations into one special-purpose operation that is more efficient than implementing two joins of two relations

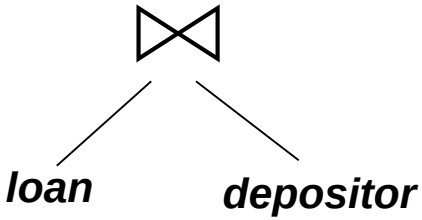
loan ⋈ depositor ⋈ customer

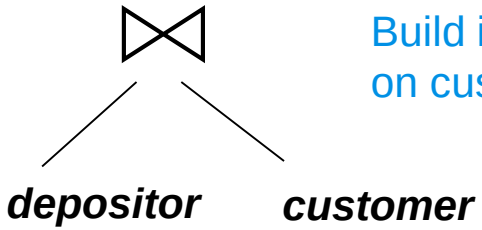




For each tuple in depositor,
Look up both loan and customer

Build index
on loan





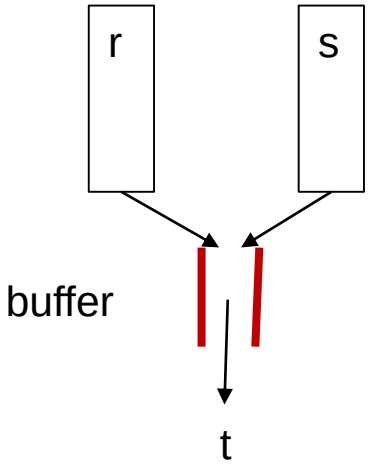
Build index
on customer

Demand-driven Pipelining (or Lazy Evaluation)

- System repeatedly requests next tuple from top level operation
  - Each operation requests next tuple from children operations as required, in order to output its next tuple
  - In between calls, operation has to maintain "**state**" so it knows what to return next
- Each operator implements **Iterator** interface
  - open(): *Initialize the state (sometimes called init())*
  - get_next(): *get the next tuple from the operator*
  - close(): *Finish and clean up*
- Sequential Scan: (state: pointer to beginning of file)
  - open(): open the file
  - get_next(): get the next tuple from file, and advance and store file pointer
  - close(): close the file
- Merge Join: (state: pointers to beginning of sorted relations)
  - open(): sort relations;
  - get_next(): continue with merge from earlier state till next output tuple is found. Save pointers as iterator state.
  - close()
- Alternative name: **pull** models of pipelining

Producer-driven Pipeline (or Eager Pipelining)

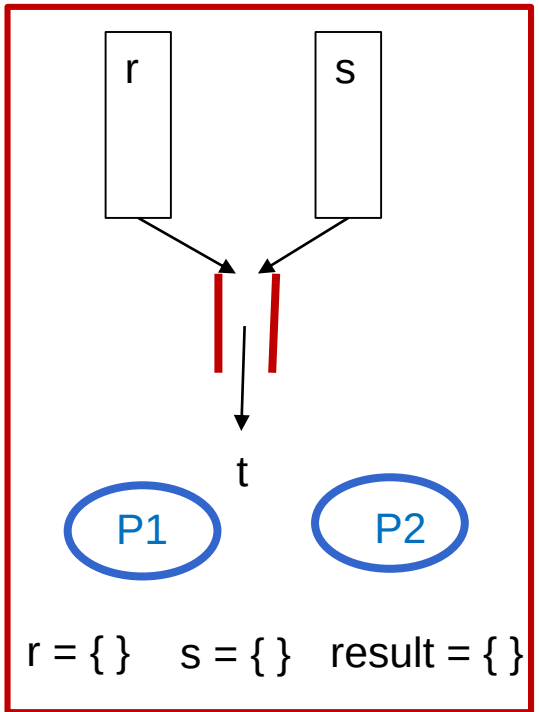
- Operators produce tuples eagerly and pass them up to their parents
  - Buffer maintained between operators, child puts tuples in buffer, parent removes tuples from buffer
  - If buffer is full, child waits till there is space in the buffer, and then generates more tuples
  - System schedules operations that have space in output buffer and can process more input tuples
- Alternative name: **push** models of pipelining



Inventing Evaluation Algorithms for Pipelining

- Some algorithms such as sorting are not able to output results even as they get input tuples (because intermediate results must be written to disk and then read back)
- We need to invent the pipelined algorithm variants to generate (at least some) results on the fly, as input tuples are read in
  - Invent Parallel Algorithms!

- Let's consider **Double-Pipelined Join Technique (Next Slide)**



Single queue를 도입하여 pipeline의 room을 만들었다!

P1 If t is from r, then r = r U {t}, result = result U ({t} ⋈ s}

P2 If t is from s, then s = s U {t}, result = result U ({t} ⋈ r}

```
done_r := false;
done_s := false;
r := ∅;
s := ∅;
result := ∅;
while not done_r or not done_s do
    begin
        if queue is empty, then wait until queue is not empty;
        t := top entry in queue;
        if t = End_r then done_r := true
          else if t = End_s then done_s := true
            else if t is from input r
              then
                  begin
                      r := r ∪ {t};
                      result := result ∪ ({t} ⋉ s);
                  end
              else /* t is from input s */
                  begin
                      s := s ∪ {t};
                      result := result ∪ (r ⋈ {t});
                  end
    end
```

Figure 12.12 Double-pipelined join algorithm.

Double-Pipelined Join for Large Relations

- The algorithm 12.12 in the previous slide assumes the relations R and S fit in memory

- Now, assume two relations R and S are bigger than memory
- 2 partitions (R0, S0) will keep tuples from R and S until main memory is full
- 2 queues (QueueR, QueueS) will keep tuples that arrive subsequently
- The join of QueueR with S0 and the join of QueueS with R0 is carried out in a pipelined way
- Finally, the join of R0 tuples with S0 tuples in memory must be carried out to complete join