



Chapter 12: Query Processing

Database System Concepts, 6th Ed.

©Silberschatz, Korth and Sudarshan

See www.db-book.com for conditions on re-use



Database System Concepts

- Chapter 1: Introduction
- **Part 1: Relational databases**
 - Chapter 2: Introduction to the Relational Model
 - Chapter 3: Introduction to SQL
 - Chapter 4: Intermediate SQL
 - Chapter 5: Advanced SQL
 - Chapter 6: Formal Relational Query Languages
- **Part 2: Database Design**
 - Chapter 7: Database Design: The E-R Approach
 - Chapter 8: Relational Database Design
 - Chapter 9: Application Design
- **Part 3: Data storage and querying**
 - Chapter 10: Storage and File Structure
 - Chapter 11: Indexing and Hashing
 - Chapter 12: Query Processing
 - Chapter 13: Query Optimization
- **Part 4: Transaction management**
 - Chapter 14: Transactions
 - Chapter 15: Concurrency control
 - Chapter 16: Recovery System
- **Part 5: System Architecture**
 - Chapter 17: Database System Architectures
 - Chapter 18: Parallel Databases
 - Chapter 19: Distributed Databases
- **Part 6: Data Warehousing, Mining, and IR**
 - Chapter 20: Data Mining
 - Chapter 21: Information Retrieval
- **Part 7: Specialty Databases**
 - Chapter 22: Object-Based Databases
 - Chapter 23: XML
- **Part 8: Advanced Topics**
 - Chapter 24: Advanced Application Development
 - Chapter 25: Advanced Data Types
 - Chapter 26: Advanced Transaction Processing
- **Part 9: Case studies**
 - Chapter 27: PostgreSQL
 - Chapter 28: Oracle
 - Chapter 29: IBM DB2 Universal Database
 - Chapter 30: Microsoft SQL Server
- **Online Appendices**
 - Appendix A: Detailed University Schema
 - Appendix B: Advanced Relational Database Model
 - Appendix C: Other Relational Query Languages
 - Appendix D: Network Model
 - Appendix E: Hierarchical Model



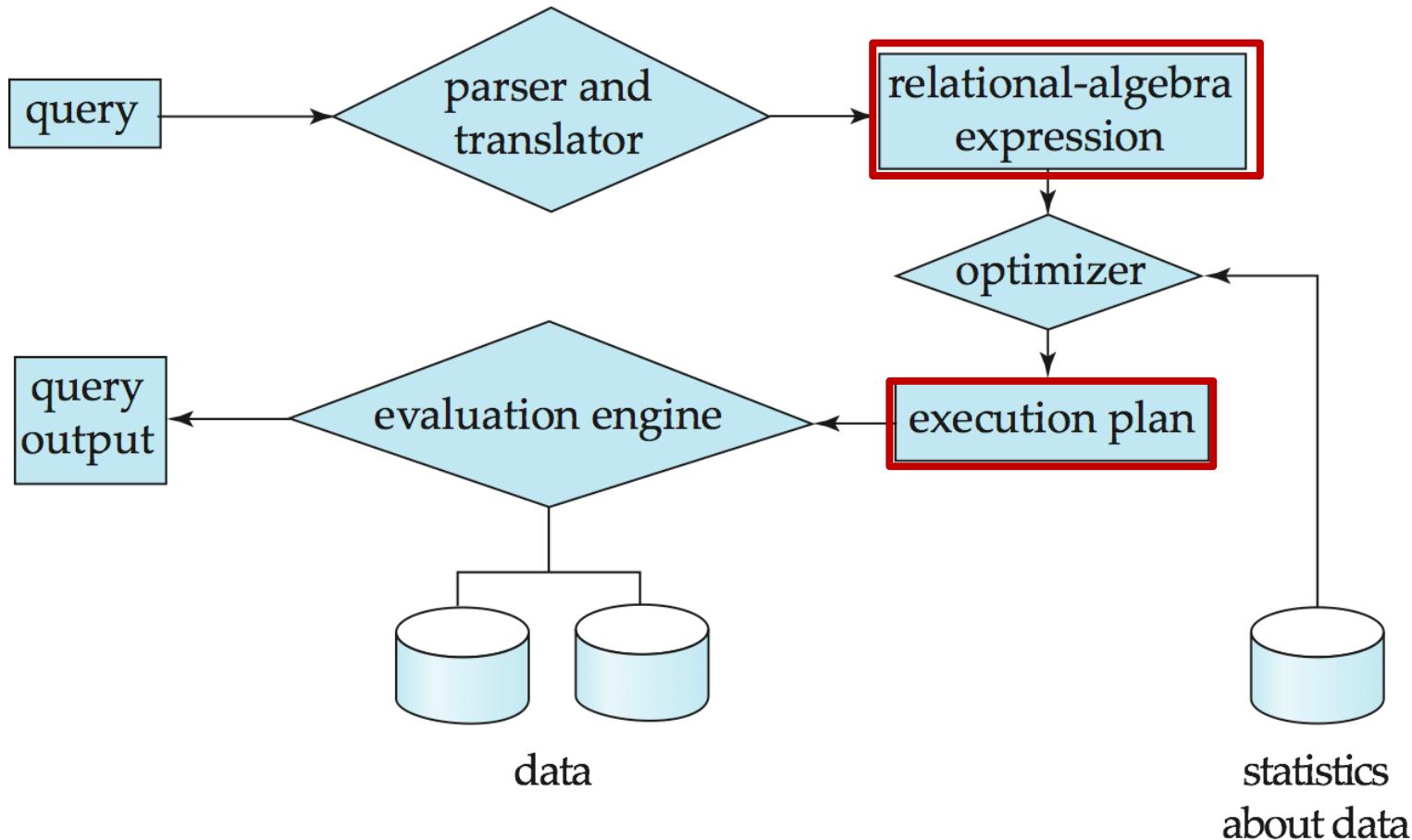
Chapter 12: Query Processing

- 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. Parsing and translation
2. Optimization
3. Evaluation





Basic Steps in Query Processing (Cont.)

■ 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 Processing : Optimization

- A relational algebra expression may have many equivalent expressions
 - E.g., $\sigma_{\text{salary} < 75000}(\Pi_{\text{salary}}(\text{instructor}))$ is equivalent to $\Pi_{\text{salary}}(\sigma_{\text{salary} < 75000}(\text{instructor}))$
- 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**.
 - (Plan 1) use an index on *salary* to find instructors with $\text{salary} < 75000$,
 - (Plan 2) perform complete relation scan and discard instructors with $\text{salary} \geq 75000$



Basic Steps: Optimization (Cont.)

■ 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 14

- 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

- 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 * average-seek-cost
 - Number of blocks read * average-block-read-cost
 - Number of blocks written * 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 (Cont.)

- For simplicity we just use the **number of block transfers** from disk and the **number of seeks** as the cost measures
 - t_T : time to transfer one block
 - t_S : time for one seek
 - Cost for b block transfers plus S seeks: $b * t_T + S * t_S$
- 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 (Cont.)

- 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

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

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

■ 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



Query1: find record with department = Finance

Query2: 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, binary search requires more seeks than index search



Selections Using Indices

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

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

- selection condition must be on search-key of index

■ A2 (primary index on key, 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, equality on nonkey)

- Need to retrieve multiple records
- Matching records will be on consecutive blocks
 - ▶ Let b = number of blocks containing matching records
- $Cost = h_i * (t_T + t_S) + t_S + t_T * b$



Query-A2: find record with Id = 76766

B+ tree on ID (key)

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

B+ tree on Dept (Nonkey)

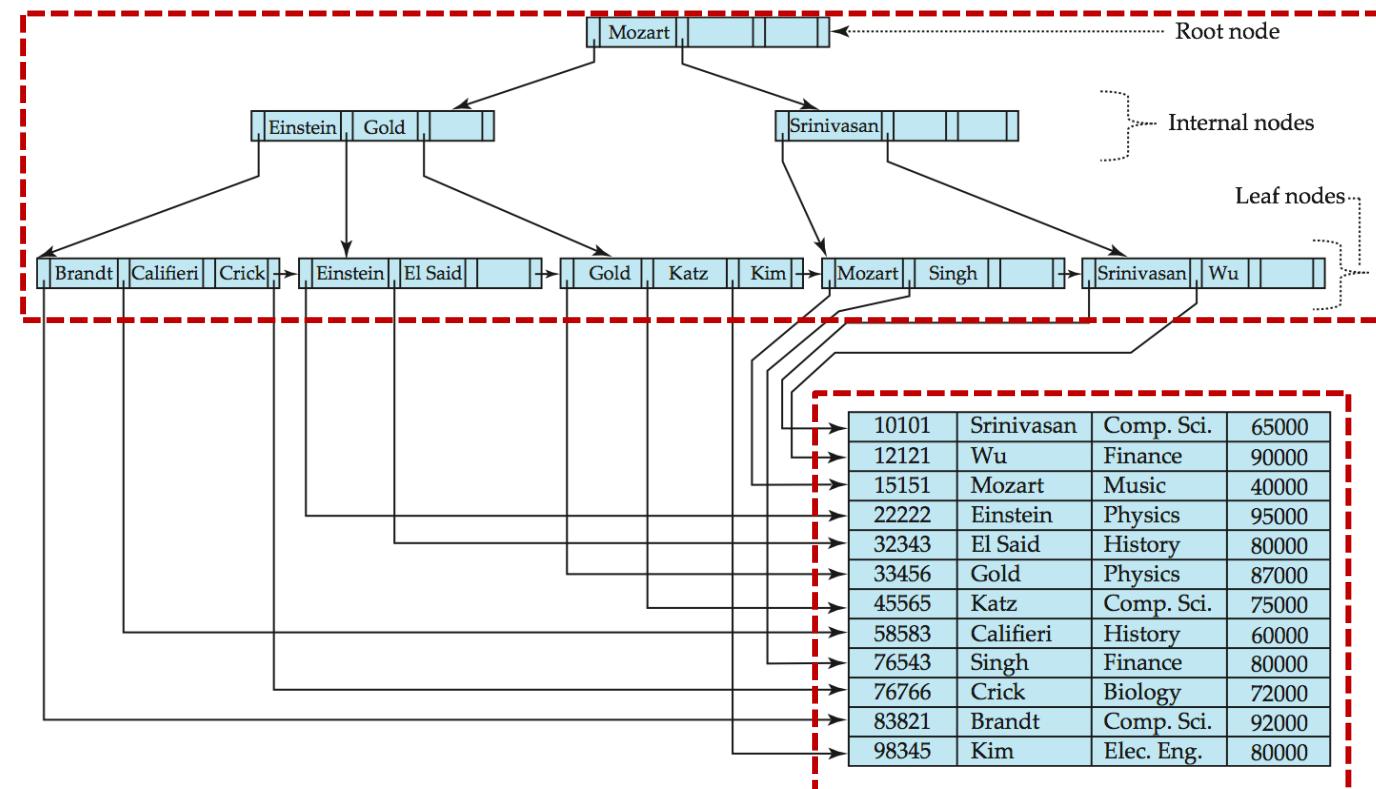
Dept	Prof	Room	Salary
CS	HJKim	406	10000
CS	MTLee	409	10500
EE	KKMin	604	11120
EE	JJCho	676	9000



Selections Using Indices (cont'd)

■ A4 (secondary index on nonkey, equality on the nonkey)

- Retrieve a single record if the search-key is a candidate key
 - $\text{Cost} = (h_i + 1) * (t_T + t_S)$
- Retrieve multiple records if search-key is not a candidate key
 - each of n matching records may be on a different block (very expensive!)
 - $\text{Cost} = (h_i + n) * (t_T + t_S)$





Selections Involving Comparisons

** Can implement selections of the form $\sigma_{A \leq v}(r)$ or $\sigma_{A \geq v}(r)$

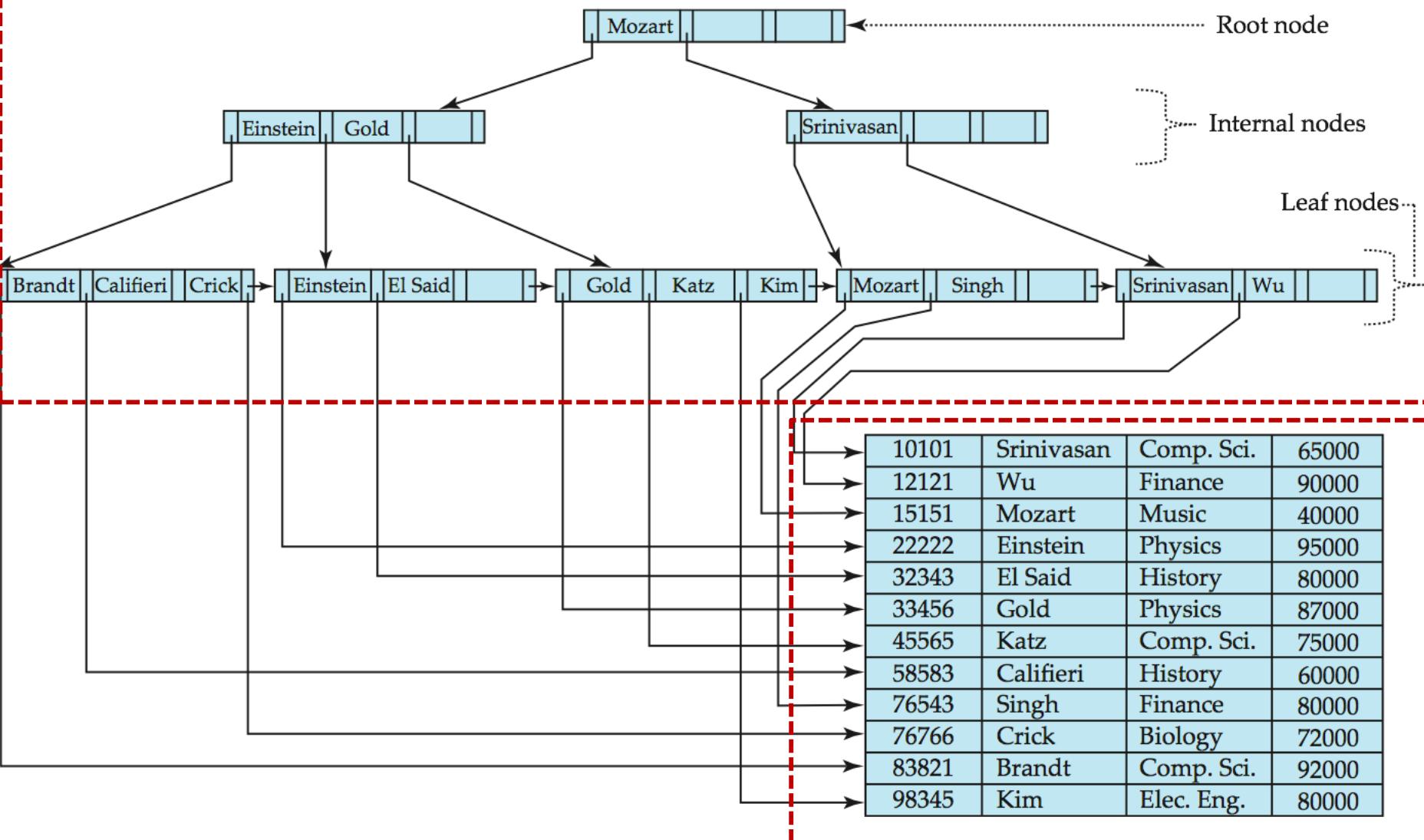
- by using a linear file scan or
- by using B+ tree indices in the following ways

■ **A5 (primary index, comparison on the key).** (Relation is sorted on A) (similar to A3)

- ▶ For $\sigma_{A \geq v}(r)$ use index to find first tuple $\geq v$ and scan relation sequentially from there
- ▶ For $\sigma_{A \leq v}(r)$ 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 $\sigma_{A \geq v}(r)$ use index to find first index entry $\geq v$ and scan index sequentially from there, to find pointers to records.
- ▶ For $\sigma_{A \leq v}(r)$ just scan leaf pages of index finding pointers to records, till first entry $> v$
- ▶ In either case, retrieve records that are pointed to
 - requires an I/O for each record
 - Linear file scan may be cheaper if many records are to be fetched!
- ▶ $Cost = (h_i + n) * (t_T + t_S)$





Implementation of Complex Selections

** Conjunctive selection query: $\sigma_{\theta_1} \wedge \theta_2 \wedge \dots \wedge \theta_n(r)$

■ 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



Algorithms for Complex Selections (contd)

** Disjunctive Selection Query: $\sigma_{\theta_1 \vee \theta_2 \vee \dots \vee \theta_n}(r)$.

■ 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: $\sigma_{-\theta}(r)$

- Bring in all records to check the negation condition
 - ▶ Use linear scan on file with the cost of A1
- If very few records satisfy $-\theta$, 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

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 run R_i ;

increment i

Let the final value of i be N

■ Merge $\text{the } N \text{ runs (next slide)}$

If $M > N$

then we can merge N runs in a single pass

else we need more than a single pass



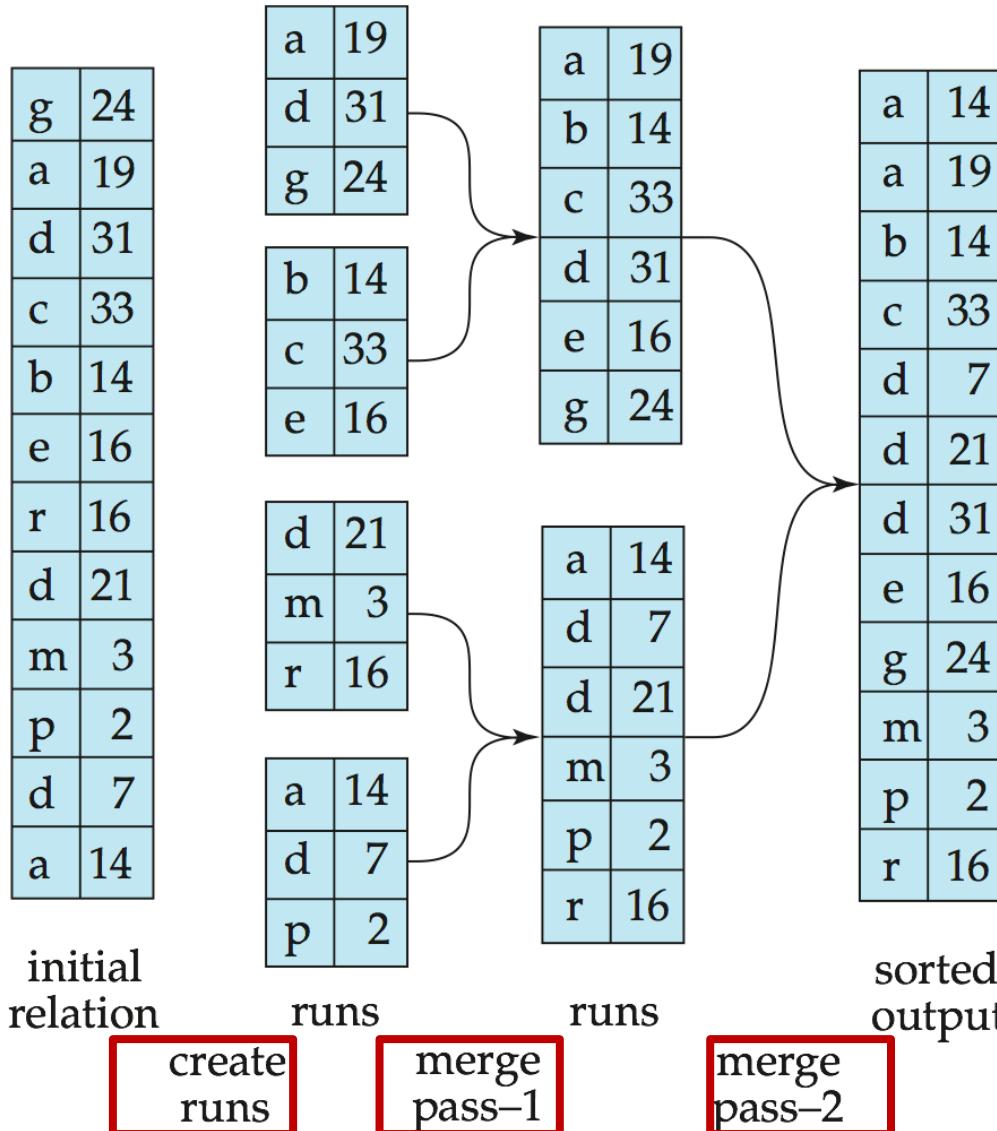
External Sort-Merge (Cont.)

- If $N < M$, merge the N sorted runs (N -way merge) in a single pass
 1. 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**
 2. **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 $N \geq M$, 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

Read R; Write R; Read R; Write R; Read R; Write R





Num of Block IOs in External Merge Sort

- Cost analysis (several merge-passes case):
 - b_r : the number of blocks in a relation r
 - M : the number of blocks in each run
 - b_r/M : the initial number of runs
 - Total number of merge passes required: $\lceil \log_{M-1}(b_r/M) \rceil$.
 - 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 write는 필요없을수도 있음
 - ▶ Thus total number of block transfers for external sorting:
 $b_r(2\lceil \log_{M-1}(b_r/M) \rceil + 1)$
 - Seeks: next slide



Num of Seek in External Merge Sort

Cost of seeks (several merge-passes case)

- b_r : the number of blocks in a relation r
- M : the number of blocks in each run
- b_r/M : the initial number of runs
- Total number of merge-passes required: $\lceil \log_{M-1}(b_r/M) \rceil$.

- During the run creation
 - one seek to read each run and one seek to write each run: $2\lceil b_r/M \rceil$
- During the merge phase
 - Buffer size: b_b (read/write b_b blocks at a time)
 - 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
 - Total number of seeks:
$$2\lceil b_r/M \rceil + \lceil b_r/b_b \rceil (2\lceil \log_{M-1}(b_r/M) \rceil - 1)$$



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
 - Hash-join
- Choice based on cost estimate
- Examples use the following information
 - 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



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  $\theta$   
    if they do, add  $t_r \cdot t_s$  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

Borrower relation: N = 16, B = 4

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 : 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

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



Nested-loop join

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



Nested-loop join

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



Nested-loop join

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

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



Nested-Loop Join (Cont.)

- In the worst case, if there is enough memory only to hold one block of each relation, the estimated cost is
$$n_r * b_s + b_r \text{ block transfers, plus } n_r + b_r \text{ seeks}$$
where n_r is number of records in R and b_s and b_r are number of disk blocks in S and R
- 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:
 - ▶ $5000 * 400 + 100 = 2,000,100$ block transfers,
 - ▶ $5000 + 100 = 5100$ seeks
 - with *takes* as the outer relation
 - ▶ $10000 * 100 + 400 = 1,000,400$ block transfers and 10,400 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

- 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  $t_r \cdot t_s$  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

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

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

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

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

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



Block Nested-Loop Join (Cont.)

- 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
- Best case estimate: $b_r + b_s$ block transfers + 2 seeks
- Improvements to nested loop and block nested loop algorithms:
 - In block nested-loop, use $M - 2$ disk blocks as blocking unit for **outer relations**, where M = memory size in blocks; use remaining two blocks to buffer **inner relation and output**
 - ▶ Cost = $\lceil b_r / (M-2) \rceil * b_s + b_r$ block transfers + $2 \lceil b_r / (M-2) \rceil$ seeks
 - If equi-join attribute forms a key or 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 LRU replacement)
 - Use **index** on inner relation if available (next slide)



Indexed Nested-Loop Join

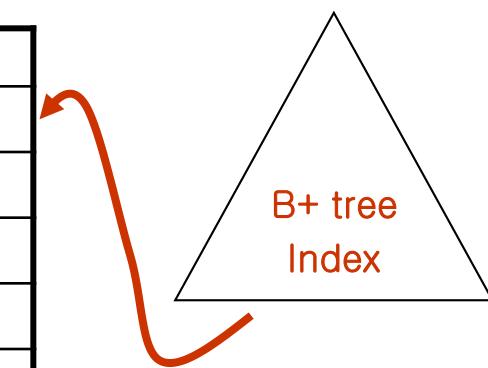
- Index lookups can replace file scans if
 - join is an equi-join or natural join and
 - an index is available on the inner relation's join attribute
 - ▶ Can construct an index just to compute a join.
- For each tuple t_r in the outer relation r , use the index (B+ tree) to look up tuples in s that 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 of 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

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 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 access 필요



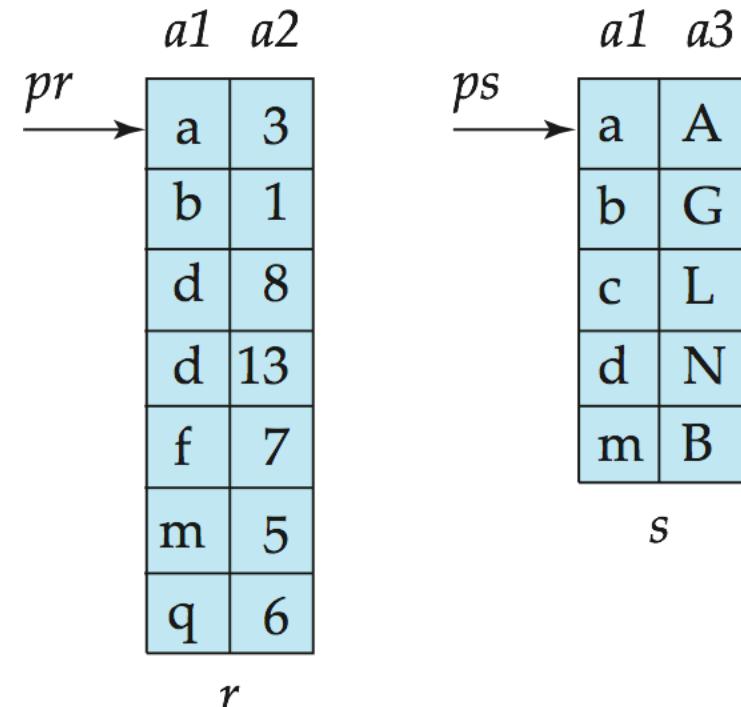
Example of Nested-Loop Join Costs

- Compute $\text{student} \bowtie \text{takes}$, 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 loops 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 loops 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. 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





Merge Join

보조자료

on The *borrower* and *loan* Relation

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

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 (Cont.)

- 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:
 $b_r + b_s$ 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:** If one relation is sorted, and the other has a secondary B⁺-tree index on the join attribute
 - Merge **the sorted relation** with **the leaf entries of the B⁺-tree**.
 - ▶ Using **the order property of the leaf entries of the B⁺- tree**
 - Sort the result on **the addresses of the unsorted relation'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

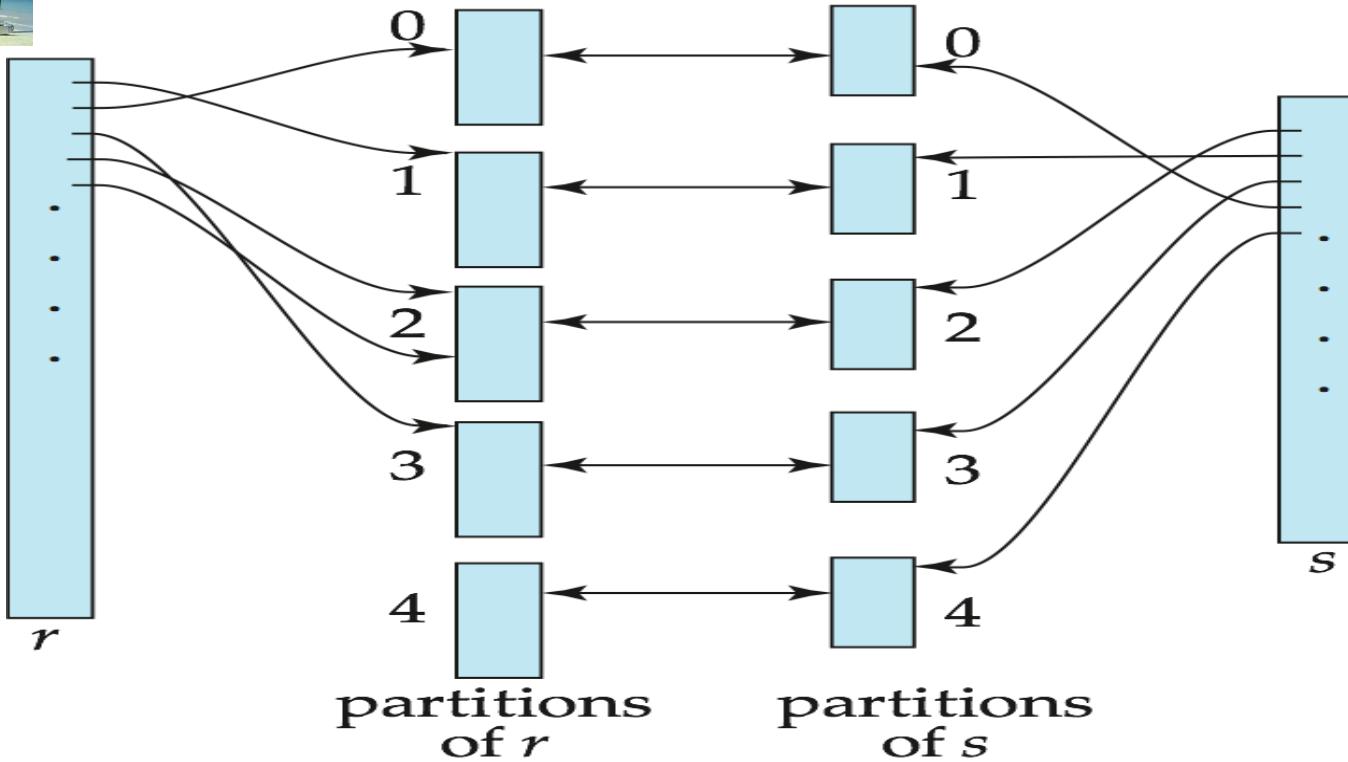


Hash-Join

- Applicable for equi-joins and natural joins.
- A hash function h is used to partition tuples of both relations
- *The hash function h maps JoinAttrs values to $\{0, 1, \dots, n\}$, where JoinAttrs denotes the common attributes of r and s used in the natural join.*
 - r_0, r_1, \dots, r_n denote partitions of r tuples
 - ▶ Each tuple $t_r \in r$ is put in partition r_i where $i = h(t_r[\text{JoinAttrs}])$.
 - s_0, s_1, \dots, s_n denote partitions of s tuples
 - ▶ Each tuple $t_s \in s$ is put in partition s_i , where $i = h(t_s[\text{JoinAttrs}])$
- Hash function이 적용되는 relation → build input (or build relation)
Probing되는 relation → probe input (or probe relation)
- Note: In book, r_i is denoted as H_{ri} , s_i is denoted as H_{si} and n is denoted as n_h



Hash-Join (Cont.)



- r tuples in r_i need only to be compared with s tuples 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

on The *borrower and loan* Relation

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

Custom-name	Loan - number	해시집합 (R)	해시집합 (S)	Loan-number	Branch-name	amount
Adams	L-16	3	0	L-11	Round Hill	900
Curry	L-93	6	1	L-14	Downtown	1500
Hayes	L-15	2	2	L-15	Perryridge	1500
Jackson	L-14	1	3	L-16	Perryridge	1300
Jones	L-17	4	4	L-17	Downtown	1000
Smith	L-11	0	5	L-23	Redwood	2000
Smith	L-23	5	6	L-93	Mianus	500
Williams	L-17	4				

<i>loan-number</i>	<i>branch-name</i>	<i>amount</i>
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

Custom-name	Loan - number	해시집합 (R)	해시집합 (S)	Loan-number	Branch-name	amount
Smith	L-11	0	0	L-11	Round Hill	900
Jackson	L-14	1	1	L-14	Downtown	1500
Hayes	L-15	2	2	L-15	Perryridge	1500
Adams	L-16	3	3	L-16	Perryridge	1300
Jones	L-17	4	4	L-17	Downtown	1000
Williams	L-17	4				
Smith	L-23	5	5	L-23	Redwood	2000
Curry	L-93	6	6	L-93	Mianus	500



Hash-Join Algorithm

The hash-join of r and s is computed as follows.

1. Partition **the relation s** using hashing function h . When partitioning a relation, one block of memory is reserved as the output buffer for each partition.
2. Partition **the relation r** similarly.
3. For each i :
 - (a) 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 than the earlier one h .
 - (b) 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.

Relation s is called the **build input** and r is called the **probe input**.



Hash-Join algorithm (Cont.)

- 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 f is a “**fudge factor**”, typically around 1.2
 - The probe relation partitions s_i need not fit in memory
- **Recursive partitioning** required if number of partitions n is greater than number of pages M of memory.
 - instead of partitioning n ways, use $M - 1$ partitions for s
 - Further partition the $M - 1$ partitions using a different hash function
 - Use same partitioning method on r
 - Rarely required: e.g., with block size of 4 KB, recursive partitioning not needed for relations of < 1GB with memory size of 2MB, or relations of < 36 GB with memory of 12 MB



Handling of Overflows

- Partitioning is said to be **skewed** if some partitions have significantly more tuples than some others
- **Hash-table overflow** occurs in partition s_i if s_i does not fit in memory. Reasons could be
 - Many tuples in s with same value for join attributes
 - Bad hash function
- **Overflow resolution** can be done in build phase
 - Partition s_i is further partitioned using different hash function.
 - Partition r_i must be similarly partitioned.
- **Overflow avoidance** performs partitioning carefully to avoid overflows during build phase
 - E.g. partition build relation into many partitions, then combine them
- Both approaches fail with large numbers of duplicates
 - Fallback option: use block nested loops join on overflowed partitions



Cost of Hash-Join

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

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

- 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:

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

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

- 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 Cost of Hash-Join

instructor \bowtie *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 + $336 (2 * (\lceil 100/3 \rceil + \lceil 400/3 \rceil))$ seeks



Hybrid Hash–Join

- 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.
- E.g. With memory size of 25 blocks, *instructor* can be partitioned into five partitions, each of size 20 blocks.
 - Division of memory:
 - ▶ The first partition occupies 20 blocks of memory
 - ▶ 1 block is used for input, and 1 block each for buffering the other 4 partitions.
- *teaches* is similarly partitioned into five partitions each of size 80
 - the first is used right away for probing, instead of being written out
- Cost of $3(80 + 320) + 20 + 80 = 1300$ block transfers for hybrid hash join, instead of 1500 with plain hash-join.
- Hybrid hash-join most useful if $M \gg \sqrt{b_s}$



Complex Joins

- Join with a conjunctive condition: $r \bowtie_{\theta_1 \wedge \theta_2 \wedge \dots \wedge \theta_n} s$
 - Compute the result of one of the simpler joins $r \bowtie_{\theta_i} s$
 - ▶ The final result comprises those tuples in the intermediate result that satisfy the remaining conditions

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

- Join with a disjunctive condition: $r \bowtie_{\theta_1 \vee \theta_2 \vee \dots \vee \theta_n} s$
 - Compute as the union of the records in individual joins $r \bowtie_{\theta_i} s$:
 $(r \bowtie_{\theta_1} s) \cup (r \bowtie_{\theta_2} s) \cup \dots \cup (r \bowtie_{\theta_n} s)$



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** can be implemented via hashing or sorting
 - On sorting 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
 - Hashing is similar – duplicates will come into the same bucket

- **Projection:**
 - perform projection on each tuple followed by **duplicate elimination**



Other Operations : Set Operations [1/2]

■ $r \cup s$ $r \cap s$ $r - s$

- Plain Method
 - Sort 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 sorted, sorting cost would be added

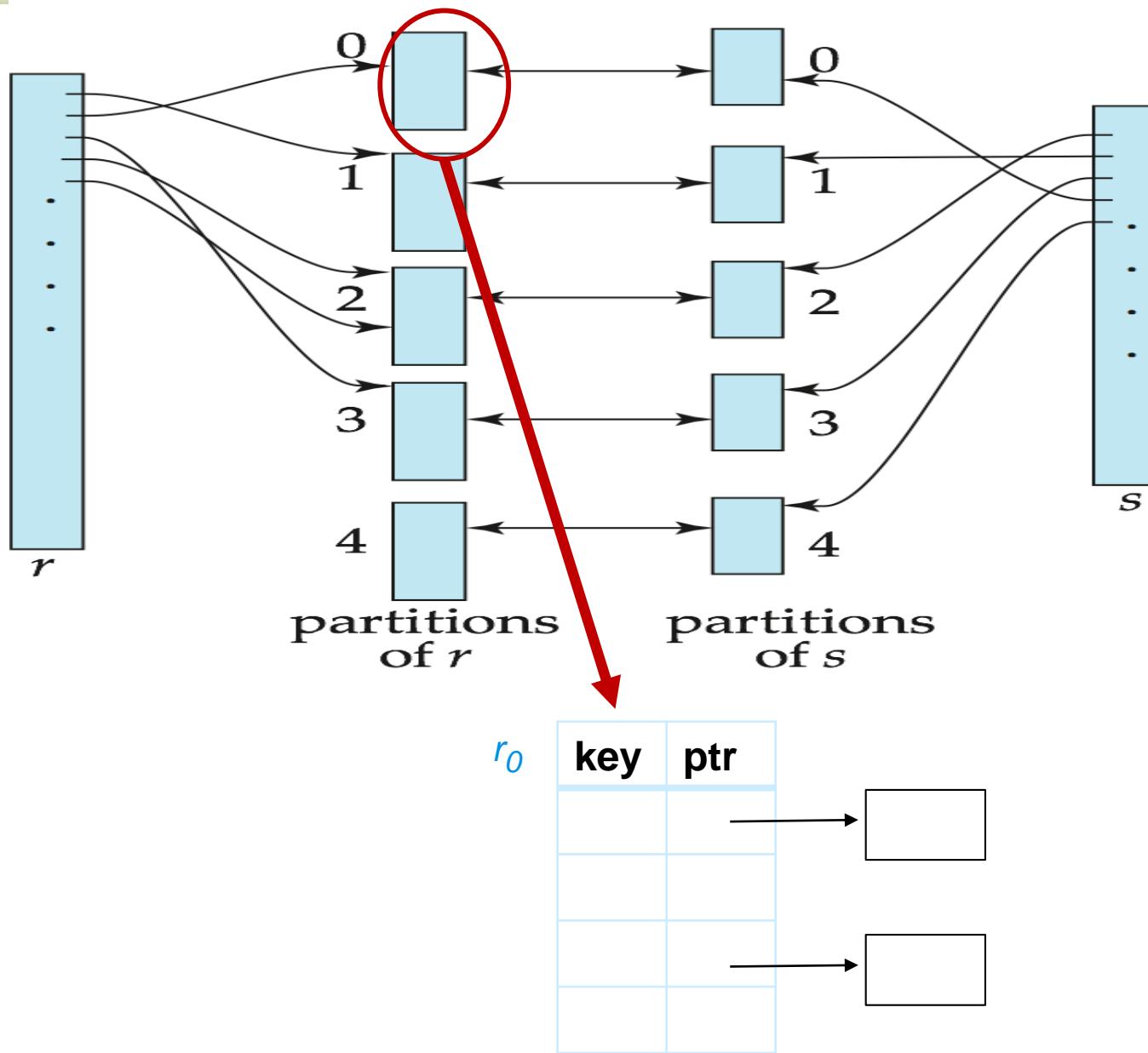


Other Operations : Set Operations [2/2]

- Using variant of merge-join after sorting, or variant of hash-join
- E.g., Set operations using hashing:
 1. Partition both relations using the same hash function, thereby creating r_0, r_1, \dots, r_n and s_0, s_1, \dots, 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 using Partition-Hashing





Other Operations : Outer Join [1/2]

■ Compute $r \bowtie s$

- 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 tuples in T with nulls for attributes in s
- Add the tuples to q1

■ Right outer-join and full outer-join can be computed similarly.

■ Relation *instructor*

<i>ID</i>	<i>name</i>	<i>dept_name</i>
10101	Srinivasan	Comp. Sci.
12121	Wu	Finance
15151	Mozart	Music

■ Relation *teaches*

<i>ID</i>	<i>course_id</i>
10101	CS-101
12121	FIN-201
76766	BIO-101

■ Left Outer Join: $instructor \bowtie teaches$

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



Other Operations : Outer Join [2/2]

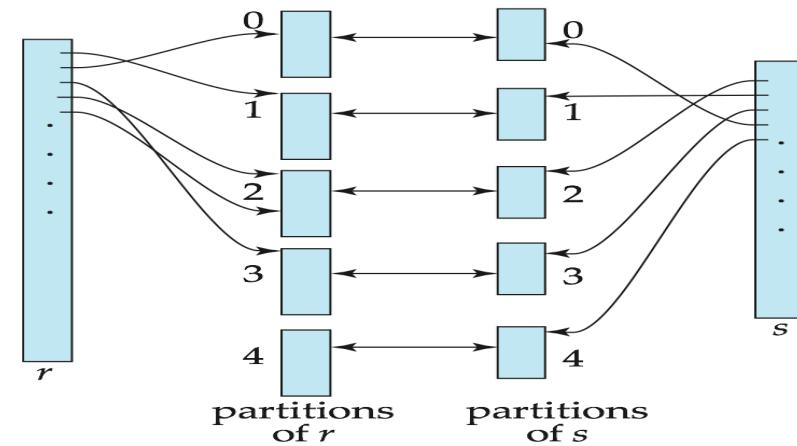
- Outer Join by modifying the merge-join and hash-join algorithms
- Modifying merge join to compute $r \bowtie s$ (*left outer-join*)
 - In $r \bowtie s$, non participating tuples are those in $r - \Pi_R(r \bowtie s)$
 - Modify merge-join to compute $r \bowtie s$:
 - During merging, for every tuple t_r from r that do not match any tuple in s , output t_r padded with nulls to the result
- Modifying hash join to compute $r \bowtie s$
 - 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

	a_1	a_2
pr	a	3
	b	1
	d	8
	d	13
	f	7
	m	5
	q	6

\xrightarrow{ps}

	a_1	a_3
	a	A
	b	G
	c	L
	d	N
	m	B

s





Other Operations : Aggregation

- **Aggregation** 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
 - **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



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
- We study above alternatives in more detail



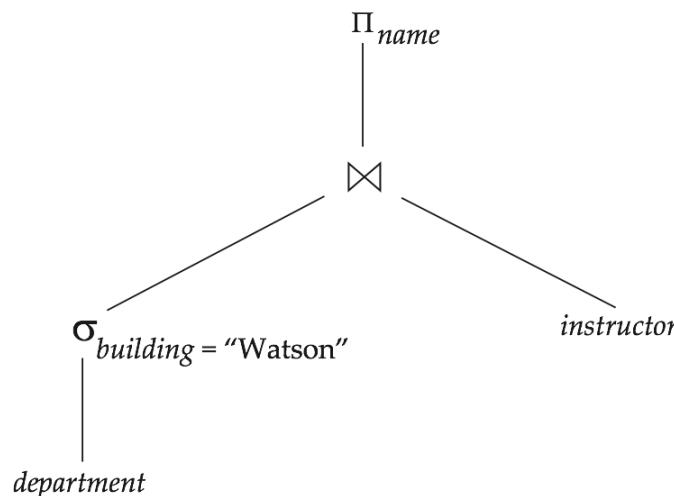
Materialization

■ Materialized evaluation

- 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 $\Pi_{name} (\sigma_{building = "Watson"}(department) \bowtie instructor)$

- compute and store $\sigma_{building = "Watson"}(department)$
- then compute the store its join with *instructor*,
- and finally compute the projection on *name*





Materialization (Cont.)

- The cost formulas for single operations ignore 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 = Sum of costs of individual operations + cost of writing intermediate results to disk
- **Double buffering:** (use two output buffers for each operation)
 - 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 $\sigma_{building = "Watson"}(department)$
 - 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, hash-join
- For pipelining to be effective, use evaluation algorithms that generate output tuples even as tuples are received for inputs to the operation
- Pipelines can be executed in two ways:
 - demand driven and producer driven



Processing Multiple Joins

- Join involving three relations: $loan \bowtie depositor \bowtie customer$
- **Strategy 1.**
 - Compute $depositor \bowtie customer$ first, and then join the result with $loan$
- **Strategy 2.**
 - Computer $loan \bowtie depositor$ first, and then join the result with $customer$
- **Strategy 3.**
 - Perform the pair of joins ($loan \bowtie depositor$, $depositor \bowtie customer$) 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 $deposit$ 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.



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 operation is implemented as an **iterator** implementing the following operations
 - **open()**
 - ▶ E.g. file scan: initialize file scan
 - state: pointer to beginning of file
 - ▶ E.g. merge join: sort relations;
 - state: pointers to beginning of sorted relations
 - **next()**
 - ▶ E.g. for file scan: Output next tuple, and advance and store file pointer
 - ▶ E.g. for merge join: 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**



Evaluation Algorithms for Pipelining

- Some algorithms are not able to output results even as they get input tuples
 - E.g. merge join, or hash join
 - intermediate results written to disk and then read back
- Algorithm variants to generate (at least some) results **on the fly**, as input tuples are read in
 - E.g. hybrid hash join generates output tuples even as probe relation tuples in the in-memory partition (partition 0) are read in
 - **Double-pipelined join technique:** Hybrid hash join, modified to buffer partition 0 tuples of both relations in-memory, reading them as they become available, and output results of any matches between partition 0 tuples
 - ▶ When a new r_0 tuple is found, match it with existing s_0 tuples, output matches, and save it in r_0
 - ▶ Symmetrically for s_0 tuples



End of Chapter

Database System Concepts, 6th Ed.

©Silberschatz, Korth and Sudarshan

See www.db-book.com for conditions on re-use