Ho Chi Minh City University of Technology
Faculty of Computer Science and Engineering

Chapter 3: Algorithms for Query Processing and Optimization

Database Management Systems (CO3021)

Computer Science Program

Dr. Võ Thị Ngọc Châu

(chauvtn@hcmut.edu.vn)

Semester 1 – 2018-2019

Course outline

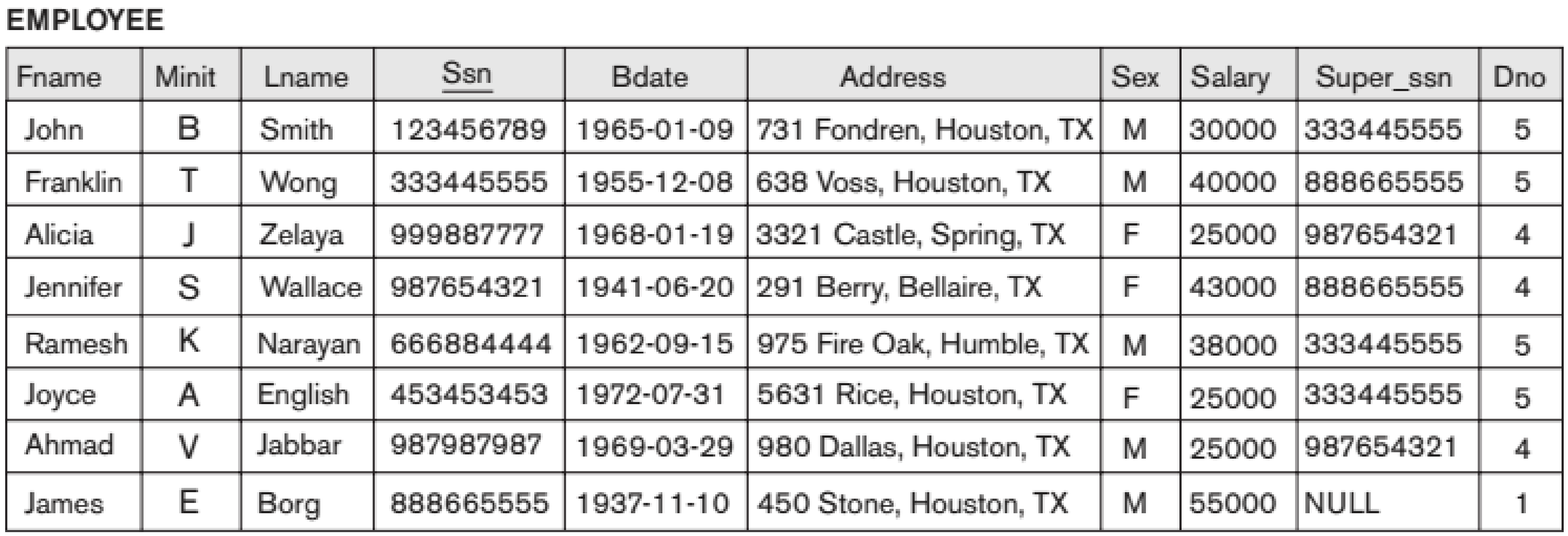
- Overall Introduction to Database Management Systems
- Chapter 1. Disk Storage and Basic File Structures
- Chapter 2. Indexing Structures for Files
- **Chapter 3. Algorithms for Query Processing and Optimization**
- Chapter 4. Introduction to Transaction Processing Concepts and Theory
- Chapter 5. Concurrency Control Techniques
- Chapter 6. Database Recovery Techniques

References

- [1] R. Elmasri, S. R. Navathe, Fundamentals of Database Systems- 6th Edition, Pearson- Addison Wesley, 2011.
  - ***R. Elmasri, S. R. Navathe, Fundamentals of Database Systems- 7th Edition, Pearson, 2016.***
- [2] H. G. Molina, J. D. Ullman, J. Widom, Database System Implementation, Prentice-Hall, 2000.
- [3] H. G. Molina, J. D. Ullman, J. Widom, Database Systems: The Complete Book, Prentice-Hall, 2002
- [4] A. Silberschatz, H. F. Korth, S. Sudarshan, Database System Concepts –3rd Edition, McGraw-Hill, 1999.
- [Internet] …

Content

3.1. Introduction to Query Processing

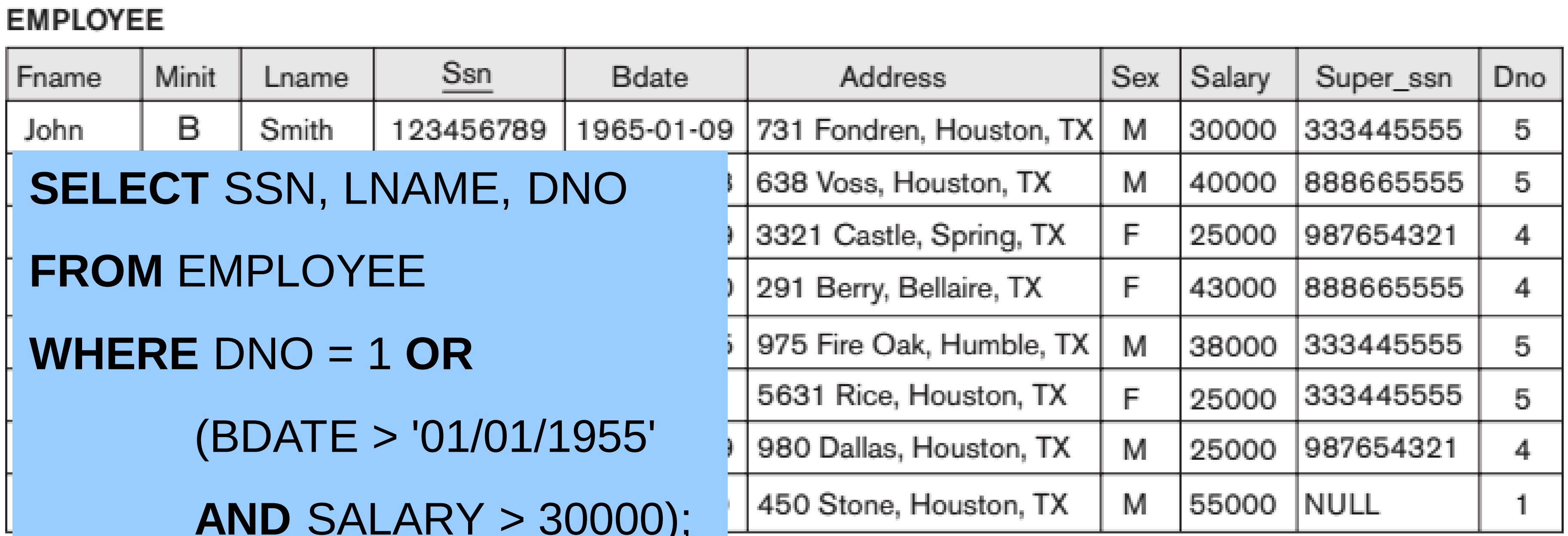


EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	NULL	1

```
CREATE TABLE EMPLOYEE (
    Fname VARCHAR(15) NOT NULL,
    Minit CHAR,
    Lname VARCHAR(15) NOT NULL,
    Ssn CHAR(9) NOT NULL,
    Bdate DATE,
    Address VARCHAR(30),
    Sex CHAR,
    Salary DECIMAL(10,2),
    Super_ssn CHAR(9),
    Dno INT NOT NULL DEFAULT 1,
    PRIMARY KEY (Ssn),
    CONSTRAINT EMPSUPERFK
    FOREIGN KEY (Super_ssn) REFERENCES
    EMPLOYEE(Ssn)
    ON DELETE SET NULL ON UPDATE CASCADE,
    CONSTRAINT EMPDEPTFK
    FOREIGN KEY(Dno) REFERENCES
    DEPARTMENT(Dnumber)
    ON DELETE SET DEFAULT ON UPDATE
    CASCADE);
```

3.1 Introduction to Query Processing



EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
					638 Voss, Houston, TX	M	40000	888665555	5
					3321 Castle, Spring, TX	F	25000	987654321	4
					291 Berry, Bellaire, TX	F	43000	888665555	4
					975 Fire Oak, Humble, TX	M	38000	333445555	5
					5631 Rice, Houston, TX	F	25000	333445555	5
					980 Dallas, Houston, TX	M	25000	987654321	4
					450 Stone, Houston, TX	M	55000	NULL	1

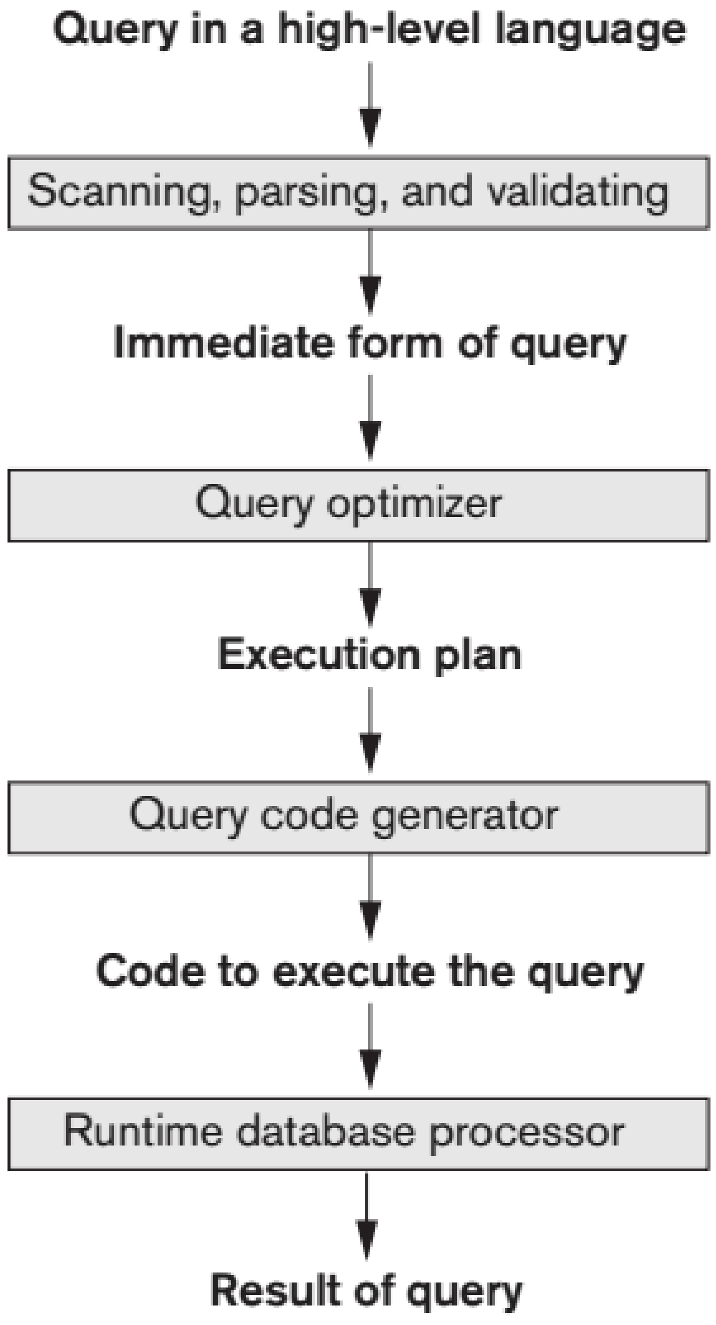
```
SELECT SSN, LNAME, DNO
FROM EMPLOYEE
WHERE DNO = 1 OR
          (BDATE > '01/01/1955'
          AND SALARY > 30000);
```

Retrieve SSN, last name, and department number of all the employees who work in department 1 or were born after 01/01/1955 with salary higher than 30000.

???

SSN	LNAME	DNO
333445555	Wong	5
666884444	Narayan	5
888665555	Borg	1

How would you do for such results?
How would you want to do that?


Query in a high-level language
Scanning, parsing, and validating
Immediate form of query
Query optimizer
Execution plan
Query code generator
Code to execute the query
Runtime database processor
Result of query

```
SELECT SSN, LNAME, DNO
FROM EMPLOYEE
WHERE DNO = 1 OR
        (BDATE > '01/01/1955'
        AND SALARY > 30000);
```

Code can be:

Executed directly (interpreted mode)

Stored and executed later whenever needed (compiled mode)

Typical steps when processing a high-level query

Figure 18.1, [1], pp. 656

3.1. Introduction to Query Processing

- A query is expressed in a high-level query language such as SQL.
  - scanned, parsed, validated
- The **scanner** identifies the query tokens (SQL keywords, attribute names, and relation names) that appear in the query text.
- The **parser** checks the query syntax to determine whether it is formulated according to the syntax grammar rules of the language.
- The **validator** checks if all attribute and relation names are valid and semantically meaningful names in the database schema.

3.1. Introduction to Query Processing

- The query is then represented in an intermediate form, i.e. internal representation.
  - **Query Tree**
  - **Query Graph**
- The DBMS must then devise an **execution strategy** or **query plan** for retrieving the results of the query from the database files.
  - A query has many possible execution plans, and the process of choosing a suitable one for processing a query is known as **query optimization**.

3.1. Introduction to Query Processing

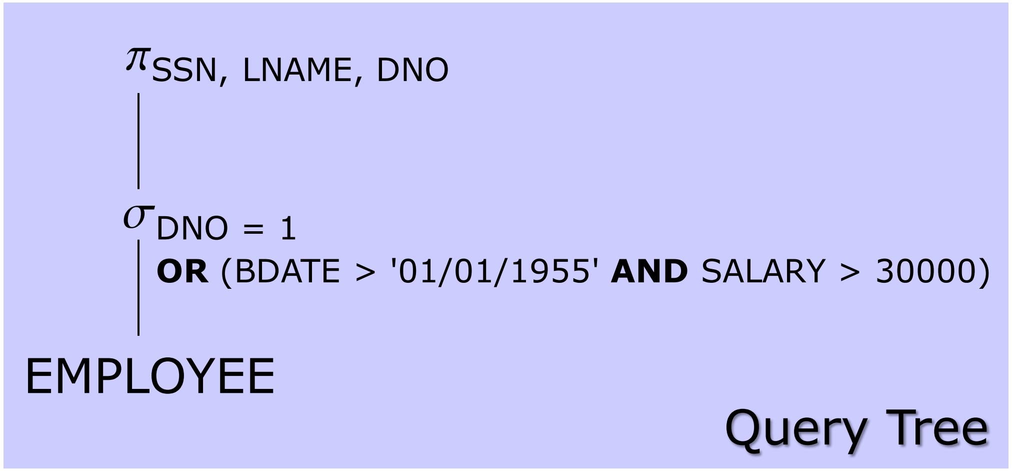
- Query Tree
  - A tree data structure corresponds to an ***extended relational algebra expression***.
  - It represents the input relations of the query as ***leaf nodes*** of the tree.
  - It represents the relational algebra operations as ***internal nodes***.
  - An execution of the query tree consists of executing an internal node operation whenever its operands are available and then replacing that internal node by the relation that results from executing the operation.
  - The order of execution of operations *starts at the leaf nodes* and *ends at the root node*.

3.1. Introduction to Query Processing

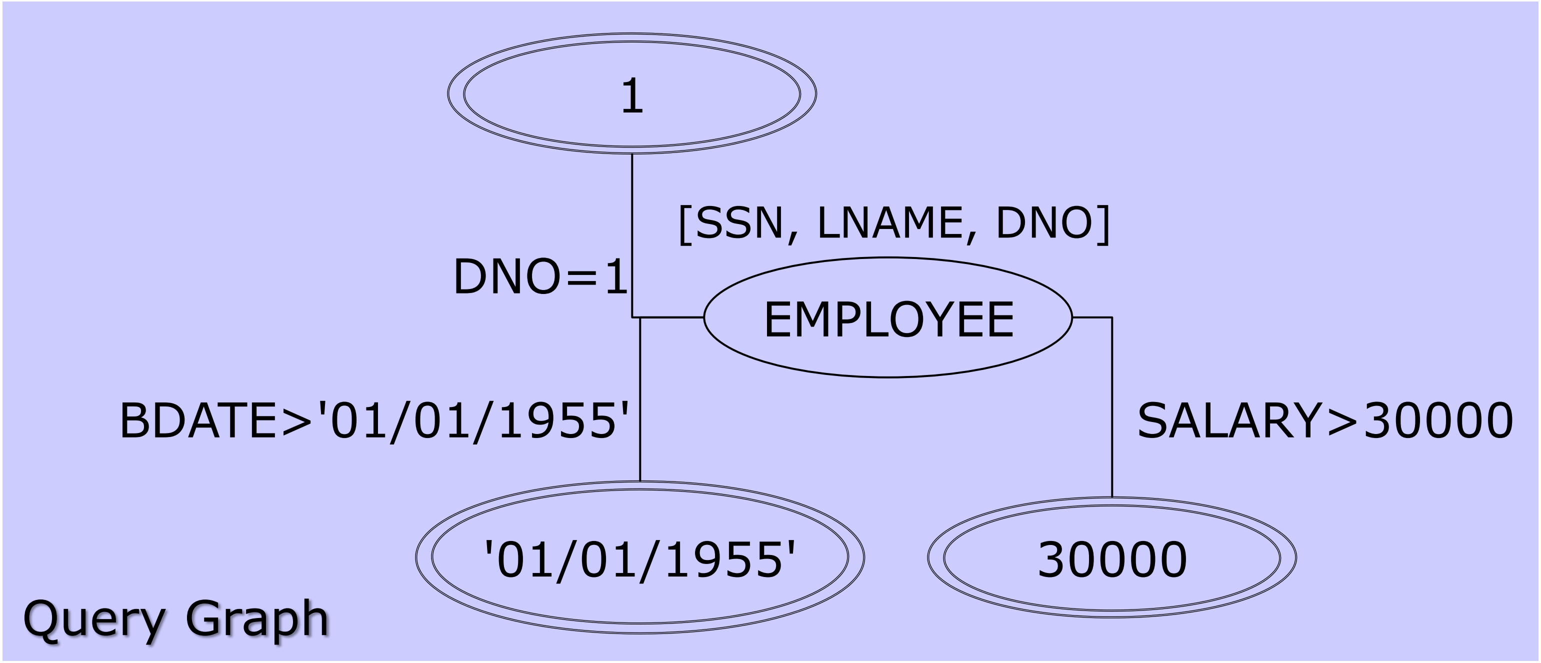
- Query Graph
  - Relations in the query are represented by ***relation nodes***, which are displayed as single circles.
  - Constant values, typically from the query selection conditions, are represented by ***constant nodes***, which are displayed as double circles or ovals.
  - Selection and join conditions are represented by the graph ***edges***.
  - The attributes to be retrieved from each relation are displayed in square brackets above each relation.

3.1. Introduction to Query Processing

```
SELECT SSN, LNAME, DNO
FROM EMPLOYEE
WHERE DNO = 1 OR
        (BDATE > '01/01/1955'
        AND SALARY > 30000);
```


$\pi_{\text{SSN, LNAME, DNO}}$
$\sigma_{\text{DNO = 1 OR (BDATE > '01/01/1955' AND SALARY > 30000)}}$
EMPLOYEE

Query Tree



Query Graph

3.1. Introduction to Query Processing

- The **query optimizer** module has the task of producing a good execution plan.
- the **code generator** generates the code to execute that plan.
- The **runtime database processor** has the task of running (executing) the query code, whether in compiled or interpreted mode, to produce the query result.
  - If a runtime error results, an error message is generated by the runtime database processor.

3.2. Translating SQL Queries into Relational Algebra

- An SQL query is first translated into an equivalent *extended* relational algebra expression—represented as a ***query tree*** data structure—that is then ***optimized***.

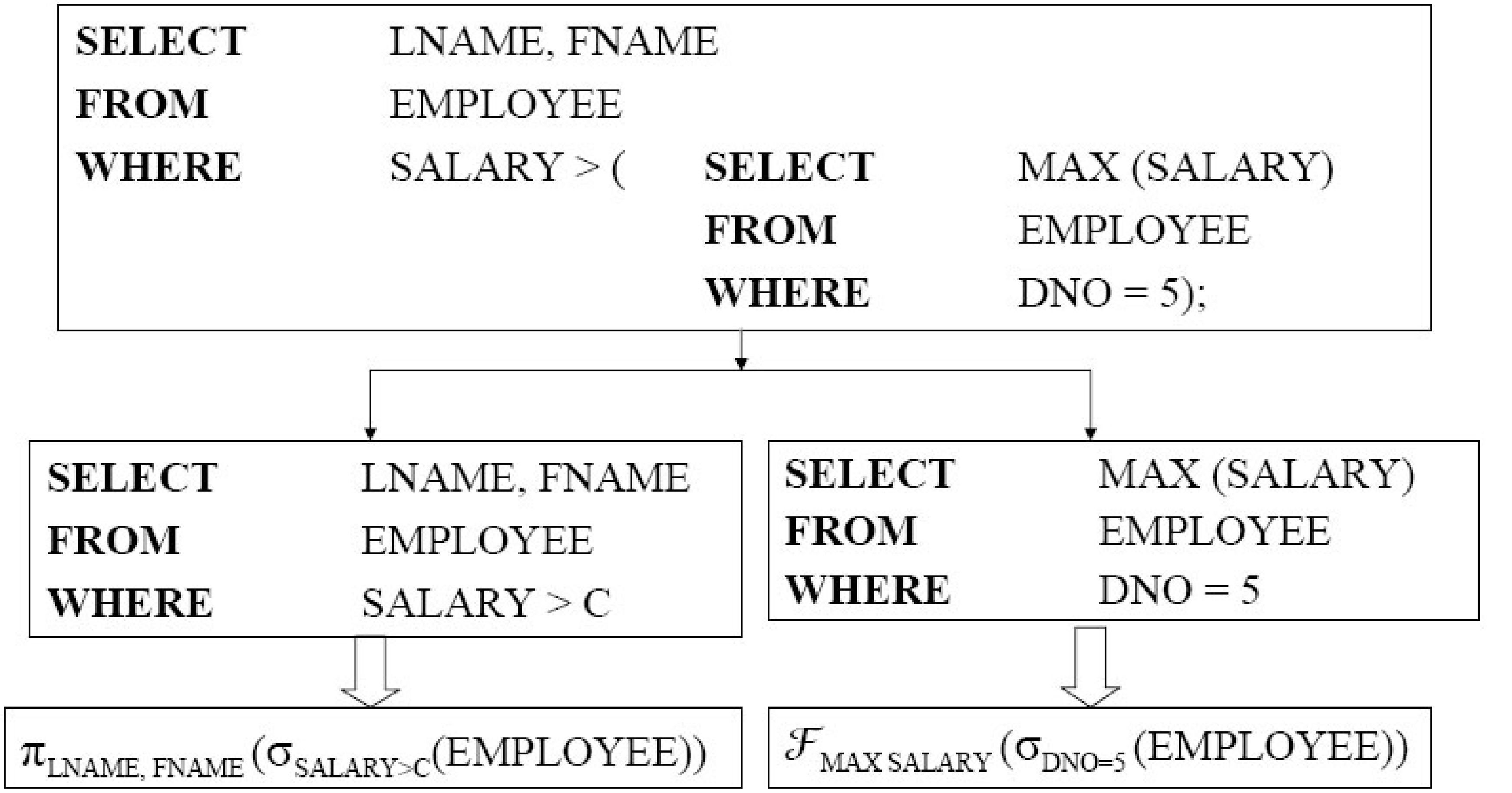
SQL clause	Relational operation	Meaning
FROM a single table	(none)	Input table
FROM table1, table2	table1 X table2	Cartesian product
FROM table1 JOIN table2 ON conditions	table1 $\bowtie_{\text{conditions}}$ table2	Theta join
WHERE conditions	$\sigma_{\text{conditions}}$	Selection
SELECT an attribute list	$\pi_{\text{an attribute list}}$	Projection
SELECT a function list … [GROUP BY a grouping attribute list]	<a grouping attribute list> $\mathfrak{I}$ <function list>	Aggregation

3.2. Translating SQL Queries into Relational Algebra

- **Query block:** the basic unit that can be translated into the algebraic operators and optimized.
- A query block contains a single SELECT-FROM-WHERE expression, as well as GROUP BY and HAVING clauses if these are part of the block.
- **Nested queries** within a query are identified as separate query blocks.
- Aggregate operators (MAX, MIN, COUNT, SUM) must be included in the extended algebra.

3.2. Translating SQL Queries into Relational Algebra

Retrieve the names of employees (from any department in the company) who earn a salary that is greater than the *highest salary in department 5*



```
SELECT    LNAME, FNAME
FROM      EMPLOYEE
WHERE     SALARY > (   SELECT    MAX (SALARY)
                       FROM      EMPLOYEE
                       WHERE     DNO = 5);
```

```
SELECT    LNAME, FNAME
FROM      EMPLOYEE
WHERE     SALARY > C
```

```
SELECT    MAX (SALARY)
FROM      EMPLOYEE
WHERE     DNO = 5
```

$\pi_{\text{LNAME, FNAME}}(\sigma_{\text{SALARY>C}}(\text{EMPLOYEE}))$

$\mathcal{F}_{\text{MAX SALARY}}(\sigma_{\text{DNO=5}}(\text{EMPLOYEE}))$

3.3. Algorithms for External Sorting

- Sorting is one of the primary algorithms used in query processing.
  - the ORDER BY clause
  - sort-merge algorithms for JOIN and set operations
  - duplicate elimination algorithms for the PROJECT operation
    - DISTINCT in the SELECT clause
- **External sorting** : refers to sorting algorithms that are suitable for large files of records stored on disk that do not fit entirely in main memory.

3.3. Algorithms for External Sorting

- **Sort-Merge strategy** : starts by sorting small subfiles (**runs**) of the main file and then merges the sorted runs, creating larger sorted subfiles that are merged in turn.
  - Sorting phase: $n_R = \lceil b/n_B \rceil$
  - Merging phase: $d_M = \min(n_B-1, n_R)$

    $n_P = \lceil \log_{dM}(n_R) \rceil$

  b: number of file blocks

  n_B: available buffer space

  n_R: number of initial runs

  d_M: degree of merging

  n_P: number of passes

3.3. Algorithms for External Sorting

```
set i ← 1; j ← b;  /* size of the file in blocks  */
    k ← $n_B$;          /* size of buffer in blocks */
    m ← $\lceil j/k \rceil$;       /* the number of runs */
/*Sort phase*/
while (i<= m) do
{
   read next k blocks of the file into the buffer or if
   there are less than k blocks remaining, then read
   in the remaining blocks;
   sort the records in the buffer and write as a
   temporary subfile;
   i ← i+1;
}
```

The number of block accesses for the sort phase = 2*b

```
/*Merge phase: merge subfiles until only one remains */
set i ← 1;
    p ← ⌈log_{k-1} m⌉;/*p: number of passes in the merging phase*/
    j ← m;          /* the number of runs */
while (i<= p) do
{
    n ← 1;
    q ← ⌈j/(k-1)⌉; /*the number of runs to write in this pass*/
    while ( n <= q) do
    {
        read next k-1 subfiles or remaining subfiles (from
      previous pass) one block at a time
        merge and write as new subfile one block at a time;
        n ← n+1;
     }
    j ← q;
    i ← i+1;
}
```

The number of block accesses for the merge phase = 2*(b*$\lceil \log_{dM} n_R \rceil$)

Total cost of external sorting = $\lceil 2 * b + 2 * (b * (\log_{dM} n_R)) \rceil$ block accesses

3.3. Algorithms for External Sorting

	a 19		
	d 31	a 19	
g 24	g 24	b 14	a 14
a 19		c 33	a 19
d 31	b 14	d 31	b 14
c 33	c 33	e 16	c 33
b 14	e 16	g 24	d 7
e 16			d 21
r 16	d 21	a 14	d 31
d 21	m 3	d 7	e 16
m 3	r 16	d 21	g 24
p 2		m 3	m 3
d 7	a 14	p 2	p 2
a 14	d 7	r 16	r 16
	p 2		
$\longrightarrow$	$\longrightarrow$	$\longrightarrow$	
Sort: runs	**Merge: pass-1**	**Merge: pass-2**	

Buffer-size = 3,

1 record/block

The first field is a sorting field.

3.4. Algorithms for SELECT and JOIN Operations

```
CREATE TABLE EMPLOYEE (
    Fname VARCHAR(15) NOT NULL,
    Lname VARCHAR(15) NOT NULL,
    Ssn CHAR(9) NOT NULL,
    Bdate DATE,
    Sex CHAR,
    Salary DECIMAL(10,2), …
    Dno INT NOT NULL DEFAULT 1,
    PRIMARY KEY (Ssn),
    CONSTRAINT EMPSUPERFK
    …
    CONSTRAINT EMPDEPTFK
    FOREIGN KEY(Dno) REFERENCES
    DEPARTMENT(Dnumber)
    ON DELETE SET DEFAULT ON UPDATE
    CASCADE);

CREATE TABLE DEPARTMENT (
        Dname VARCHAR(15) NOT NULL,
        Dnumber INT NOT NULL,
        Mgr_ssn CHAR(9) NOT NULL,
        Mgr_start_date DATE,
        PRIMARY KEY (Dnumber),
        UNIQUE (Dname),
        FOREIGN KEY (Mgr_ssn) REFERENCES EMPLOYEE(Ssn) );

CREATE TABLE WORKS_ON (
        Essn CHAR(9) NOT NULL,
        Pno INT NOT NULL,
        Hours DECIMAL(3,1) NOT NULL,
        PRIMARY KEY (Essn, Pno),
        FOREIGN KEY (Essn) REFERENCES EMPLOYEE(Ssn),
        FOREIGN KEY (Pno) REFERENCES PROJECT(Pnumber) );
```

3.4. Algorithms for SELECT and JOIN Operations

Given the tables, some examples for selection:

- OP1: $\sigma_{SSN='123456789'}$(EMPLOYEE)
- OP2: $\sigma_{DNUMBER>5}$(DEPARTMENT)
- OP3: $\sigma_{DNO=5}$(EMPLOYEE)
- OP4: $\sigma_{DNO=5 \text{ AND } SALARY>30000 \text{ AND } SEX='F'}$ (EMPLOYEE)
- OP4’: $\sigma_{Dno=5 \text{ OR } Salary > 30000 \text{ OR } Sex ='F'}$ (EMPLOYEE)
- OP5: $\sigma_{ESSN='123456789' \text{ AND } PNO=10}$(WORKS_ON)
- OP6: $\sigma_{DNO \text{ IN } (3, 27, 49)}$(EMPLOYEE)
- OP7: $\sigma_{((Salary*Commission\_pct) + Salary) > 5000}$(EMPLOYEE)

3.4. Algorithms for SELECT and JOIN Operations

Implementing the SELECT Operation: Search

- S1. **Linear search** (brute force): Retrieve *every record* in the file, and test whether its attribute values satisfy the selection condition.
- S2. **Binary search**: If the selection condition involves an equality comparison on a key attribute on which the file is ordered, binary search (which is more efficient than linear search) can be used.
- S3. **Using a primary index or hash key** to retrieve a single record: If the selection condition involves an equality comparison on a key attribute with a primary index (or a hash key), use the primary index (or the hash key) to retrieve the record.

3.4. Algorithms for SELECT and JOIN Operations

Implementing the SELECT Operation: Search

- S4.**Using a primary index** to retrieve multiple records: If the comparison condition is >, ≥ , <, or ≤ on a key field with a primary index, use the index to find the record satisfying the corresponding equality condition, then retrieve all subsequent records in the (ordered) file.
- S5.**Using a clustering index** to retrieve multiple records: If the selection condition involves an equality comparison on a non-key attribute with a clustering index, use the clustering index to retrieve all the records satisfying the selection condition.

3.4. Algorithms for SELECT and JOIN Operations

Implementing the SELECT Operation: Search

- S6. **Using a secondary (B+-tree) index** : On an equality comparison, this search method can be used to retrieve a single record if the indexing field has unique values (is a key) or to retrieve multiple records if the indexing field is not a key. In addition, it can be used to retrieve records on conditions involving >,>=, <, or <=. (FOR **RANGE QUERIES**)

3.4. Algorithms for SELECT and JOIN Operations

Implementing the SELECT Operation: Search

- S7.a. **Using a bitmap index**: If the selection condition involves a set of values for an attribute, the corresponding bitmaps for each value can be OR-ed to give the set of record identifiers that qualify.
- S7.b. **Using a functional index**: If there is a functional index defined, this index can be used to retrieve all the records that qualify.

CREATE INDEX income_ix

ON EMPLOYEE (Salary + (Salary*Commission_pct));

This index can be used for OP7.

3.4. Algorithms for SELECT and JOIN Operations

Implementing the SELECT Operation: Search

- S8. **Conjunctive (AND) selection** : If an attribute involved in any single *simple condition* in the conjunctive condition has an access path that permits the use of one of the methods S2 to S6, use that condition to retrieve the records and then check whether each retrieved record satisfies the remaining simple conditions in the conjunctive condition.
- S9. **Conjunctive (AND) selection using a composite index**: If two or more attributes are involved in equality conditions in the conjunctive condition and a composite index (or hash structure) exists on the combined field, we can use the index directly.

3.4. Algorithms for SELECT and JOIN Operations

Implementing the SELECT Operation: Search

- S10. **Conjunctive (AND) selection by intersection of record pointers** : This method is possible if secondary indexes are available on all (or some of) the fields involved in equality comparison conditions in the conjunctive condition and if the indexes include record pointers (rather than block pointers). Each index can be used to retrieve the *record pointers* that satisfy the individual condition. The *intersection* of these sets of record pointers gives the record pointers that satisfy the conjunctive condition, which are then used to retrieve those records directly. If only some of the conditions have secondary indexes, each retrieved record is further tested to determine whether it satisfies the remaining conditions.

3.4. Algorithms for SELECT and JOIN Operations

Implementing the SELECT Operation: Search

- **Disjunctive (OR) selection conditions**: With a disjunctive selection condition, the records satisfying the disjunctive condition are the *union* of the records satisfying the individual conditions. Hence, if any *one* of the conditions does not have an access path, we are compelled to use the brute force, linear search approach. Only if an access path exists on *every* simple condition in the disjunction can we optimize the selection by retrieving the records satisfying each condition—or their record identifiers—and then applying the *union* operation to eliminate duplicates.

3.4. Algorithms for SELECT and JOIN Operations

Implementing the SELECT Operation: Search

- Whenever a **single condition** specifies the selection, we can only check whether an access path exists on the attribute involved in that condition. If an access path exists, the method corresponding to that access path is used; otherwise, the "brute force" linear search approach of method S1 is used.
- The query optimizer must choose the appropriate one for executing each SELECT operation in a query.
  - This optimization uses formulas that estimate the costs for each available access method.
  - The optimizer chooses the access method with the lowest estimated cost.

Which search method should be used?

Given EMPLOYEE and DEPARTMENT tables:

- OP1: $\sigma_{SSN='123456789'}(EMPLOYEE)$
- OP2: $\sigma_{DNUMBER>5}(DEPARTMENT)$
- OP3: $\sigma_{DNO=5}(EMPLOYEE)$
- OP4: $\sigma_{DNO=5 \text{ AND } SALARY>30000 \text{ AND } SEX='F'}(EMPLOYEE)$
- OP4’: $\sigma_{Dno=5 \text{ OR } Salary > 30000 \text{ OR } Sex ='F'}(EMPLOYEE)$
- OP5: $\sigma_{ESSN='123456789' \text{ AND } PNO=10}(WORKS\_ON)$
- OP6: $\sigma_{DNO \text{ IN } (3, 27, 49)}(EMPLOYEE)$
- OP7: $\sigma_{((Salary*Commission\_pct) + Salary) > 5000}(EMPLOYEE)$

At least, linear search for all!!!

3.4. Algorithms for SELECT and JOIN Operations

Implementing the JOIN Operation:

- Join (EQUIJOIN, NATURAL JOIN)
  - two–way join: a join on two files

    e.g. $R \bowtie_{A=B} S$
  - multi-way join: a join involving more than two files

    e.g. $R \bowtie_{A=B} S \bowtie_{C=D} T$
- Examples

  OP8: $\text{EMPLOYEE} \bowtie_{\text{DNO=DNUMBER}} \text{DEPARTMENT}$

  OP9: $\text{DEPARTMENT} \bowtie_{\text{MGR\_SSN=SSN}} \text{EMPLOYEE}$

3.4. Algorithms for SELECT and JOIN Operations

Implementing the JOIN Operation:

- J1. **Nested-loop join** (brute force): For each record *t* in *R* (outer loop), retrieve every record *s* from *S* (inner loop) and test whether the two records satisfy the join condition *t*[*A*] = *s*[*B*].
- J2. **Single-loop join** (Using an access structure to retrieve the matching records): If an index (or hash key) exists for one of the two join attributes — say, *B* of *S* — retrieve each record *t* in *R*, one at a time, and then use the access structure to retrieve directly all matching records s from *S* that satisfy *s*[*B*] = *t*[*A*].

3.4. Algorithms for SELECT and JOIN Operations

Implementing the JOIN Operation:

- J1. **Nested loop join** (brute force): For each record *t* in *R* (outer loop), retrieve every record *s* from *S* (inner loop) and test whether the two records satisfy the join condition *t*[*A*] = *s*[*B*].

```
for each record t in each block of R
        for each record s in each block of S
                if (t[A] = s[B])
                        add (t, s) into the result
```

How many block accesses are needed with a (memory) buffer?

Which (large or small) table should be on the outer loop?

3.4. Algorithms for SELECT and JOIN Operations

Implementing the JOIN Operation:

- J1. **Nested loop join** (brute force):

```
for each record t in each block of R
        for each record s in each block of S
                if (t[A] = s[B])
                        add (t, s) into the result
```

OP8: EMPLOYEE $\bowtie_{DNO=DNUMBER}$ DEPARTMENT

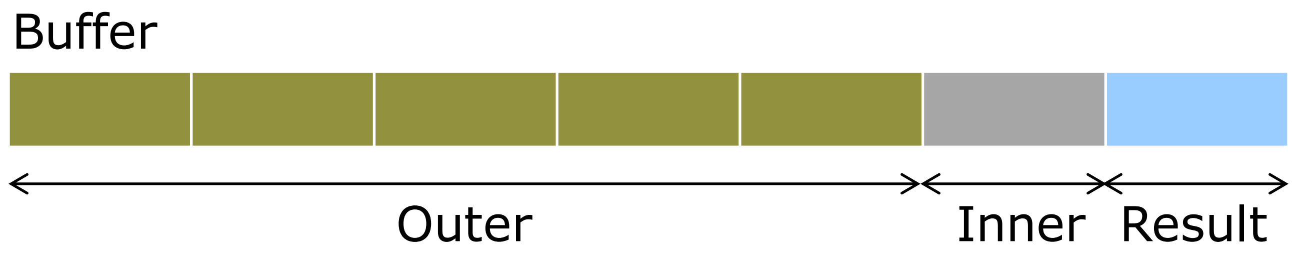
The number of blocks of EMPLOYEE b_E = 2000 blocks

The number of blocks of DEPARTMENT b_D = 10 blocks

Buffer size n_B = 7 blocks

3.4. Algorithms for SELECT and JOIN Operations

- J1. **Nested loop join** (brute force):



OP8: EMPLOYEE $\bowtie_{DNO=DNUMBER}$ DEPARTMENT

b_E = 2000 blocks, b_D = 10 blocks, n_B = 7 blocks

J1.1. EMPLOYEE on the outer loop

Cost = $b_E + b_D * \lceil b_E/(n_B-2) \rceil = 2000 + 10 * \lceil 2000/(7-2) \rceil$

Cost = 6000 block accesses

J1.2. DEPARTMENT on the outer loop

Cost = $b_D + b_E * \lceil b_D/(n_B-2) \rceil = 10 + 2000 * \lceil 10/(7-2) \rceil$

Cost = 4010 block accesses Smaller file on the outer loop!!!

3.4. Algorithms for SELECT and JOIN Operations

Implementing the JOIN Operation:

- J2. **Single-loop join** (Using an access structure to retrieve the matching records): If an index (or hash key) exists for one of the two join attributes — say, *B* of *S* — retrieve each record *t* in *R* (loop over *R*) and then use the access structure to retrieve directly all matching records s from *S* that satisfy *s*[*B*] = *t*[*A*].

```
for each record t in each block of R
        if (t[A] = s[B])
                retrieve s via the access structure (index or hash)
                add (t, s) into the result
```

3.4. Algorithms for SELECT and JOIN Operations

Implementing the JOIN Operation:

- J3. **Sort-merge join:** If the records of R and S are *physically sorted* (ordered) by value of the join attributes A and B, respectively, we can implement the join in the most efficient way possible. Both files are scanned in order of the join attributes, matching the records that have the same values for A and B. In this method, the records of each file are scanned only once each for matching with the other file—unless both A and B are non-key attributes, in which case the method needs to be modified slightly.

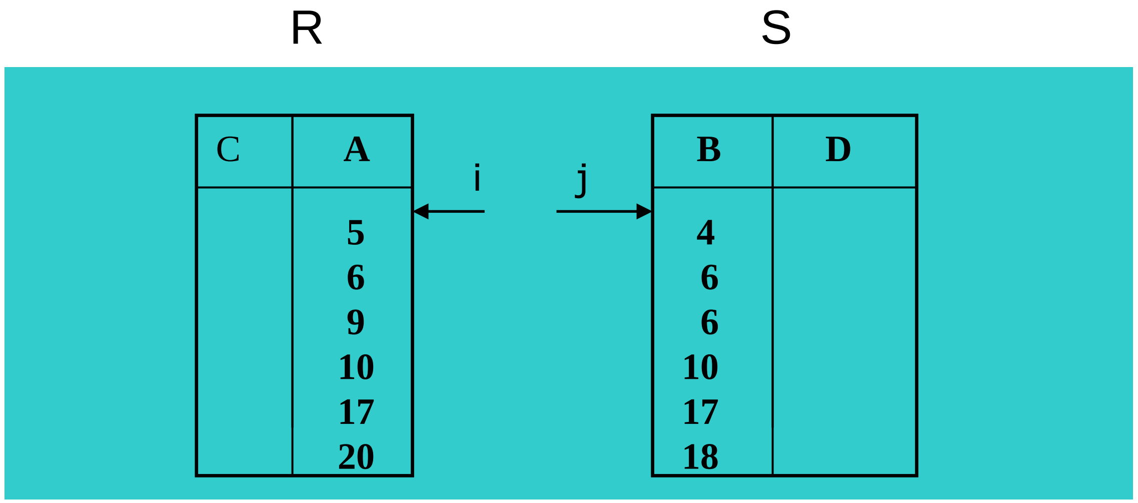
3.4. Algorithms for SELECT and JOIN Operations

J3. Implementing Sort-Merge Join: $T \leftarrow R \bowtie_{A=B} S$

```
sort the tuples in R on attribute A;  /* assume R has n tuples  */
sort the tuples in S on attribute B;  /* assume S has m tuples  */
set i ← 1, j ← 1;
while (i ≤ n) and (j ≤ m)
do {  if R(i)[A] > S(j)[B]
         then set j ← j + 1
         elseif R(i)[A] < S(j)[B]
         then set i ← i + 1
         else { /* output a matched tuple  */
            output the combined tupe <R(i), S(j)> to T;
            /* output other tuples that match R(i), if any */
            set l ← j + 1 ;
           while ( l ≤ m) and (R(i)[A] = S(l)[B])
           do { output the combined tuple <R(i), S(l)> to T;
                   set l ← l + 1
                }
            /* output other tuples that match S(j), if any */
           set k ← i+1
           while ( k ≤ n) and (R(k)[A] = S(j)[B])
           do { output the combined tuple <R(k), S(j)> to T;
                   set k ← k + 1
                }
           set i ← i+1, j ← j+1;
           }
      }
```

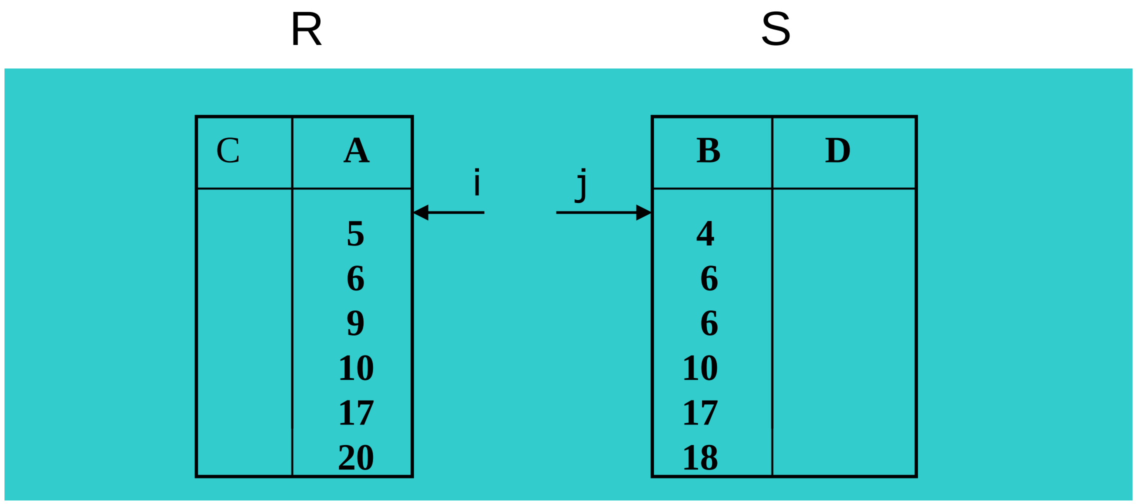
3.4. Algorithms for SELECT and JOIN Operations

J3. Implementing Sort-Merge Join: $T \leftarrow R \bowtie_{A=B} S$

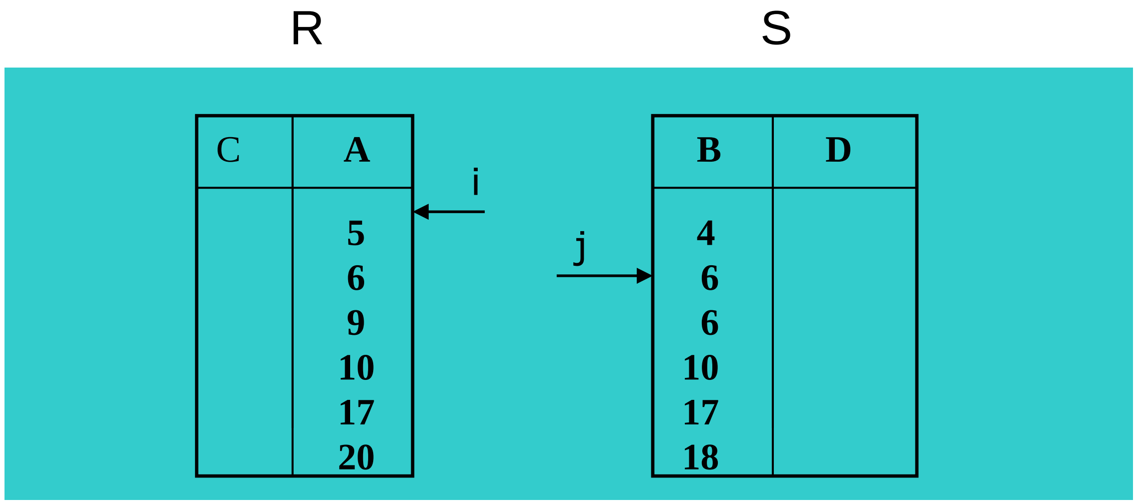


Assume that A is a key of R. Initially, two pointers are used to point to the two tuples of the two relations that have the smallest values of the two joining attributes.

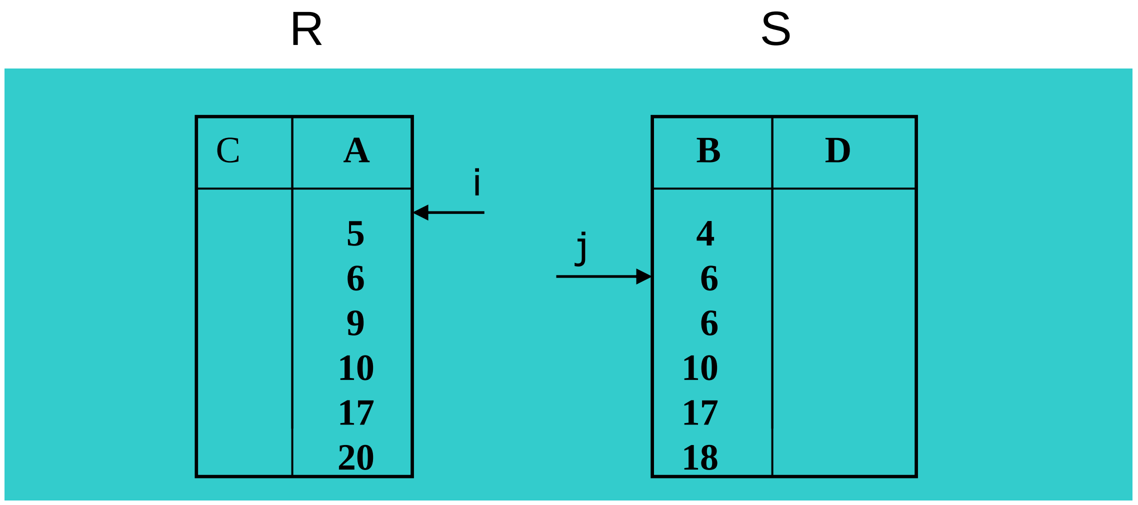
3.4. Algorithms for SELECT and JOIN Operations



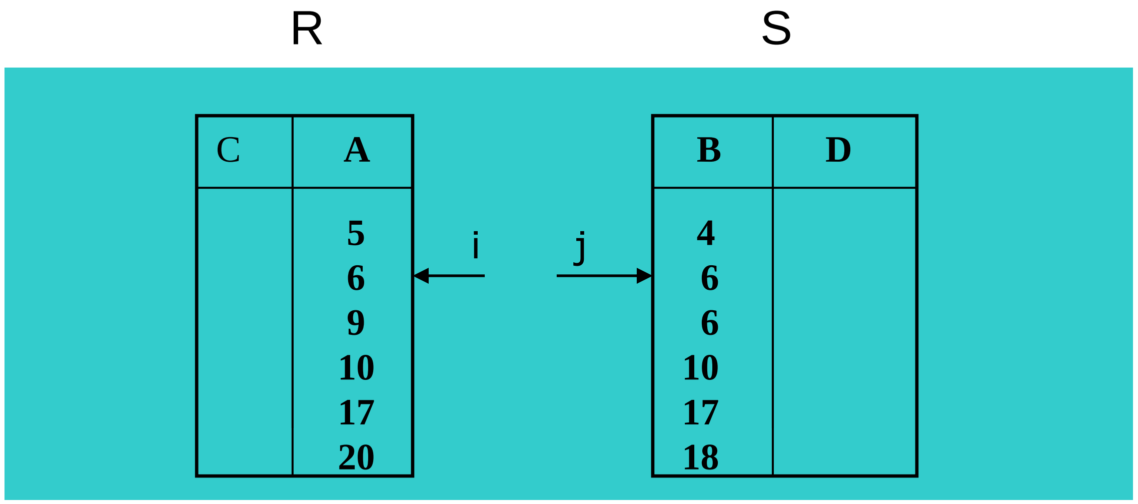
R(i)[A] > S(j)[B]



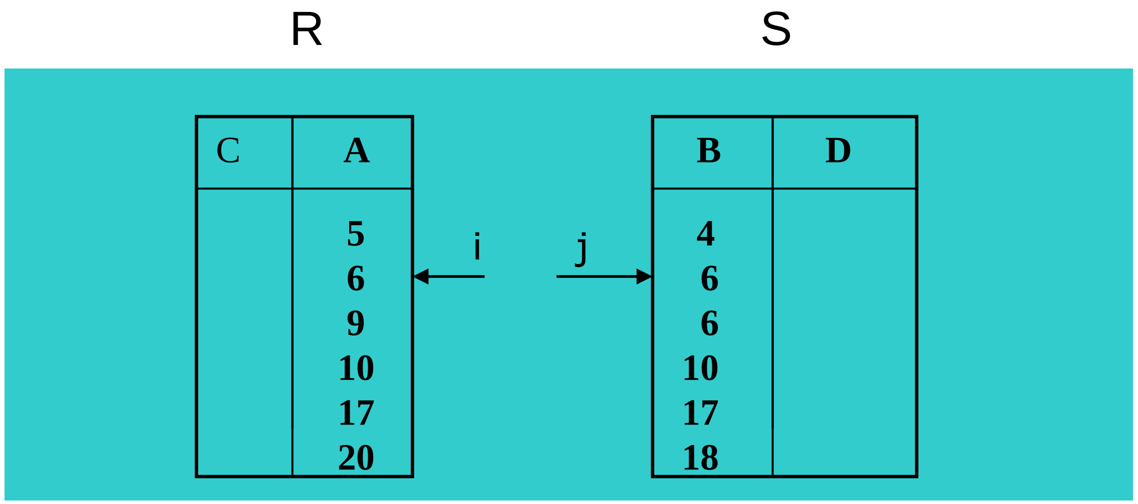
3.4. Algorithms for SELECT and JOIN Operations



R(i)[A] < S(j)[B]

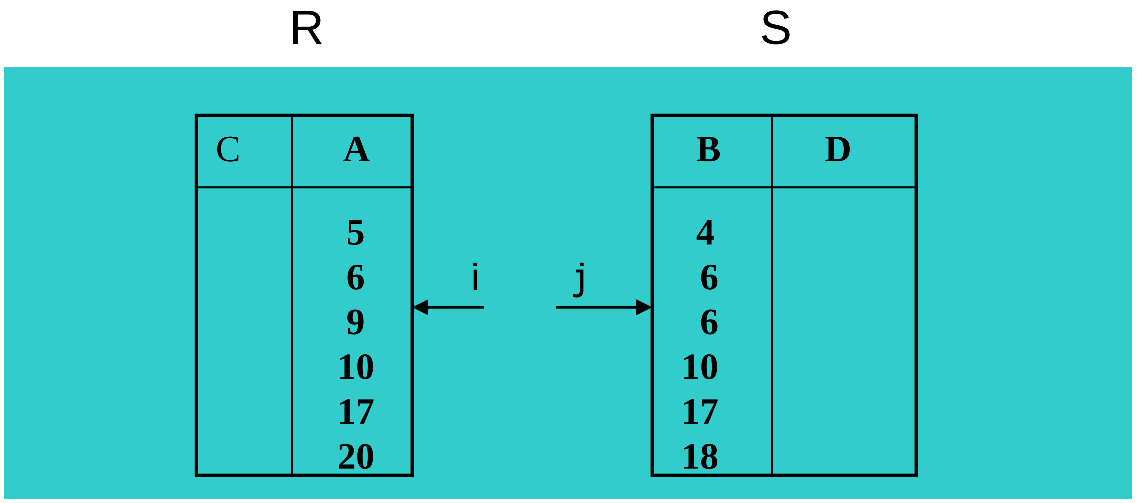


3.4. Algorithms for SELECT and JOIN Operations

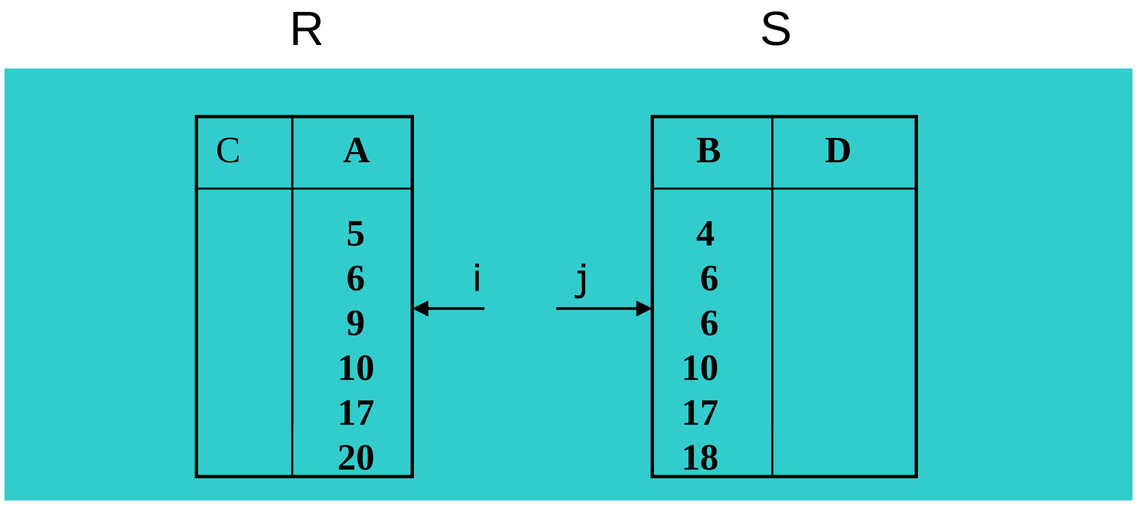


R(i)[A] = S(j)[B]

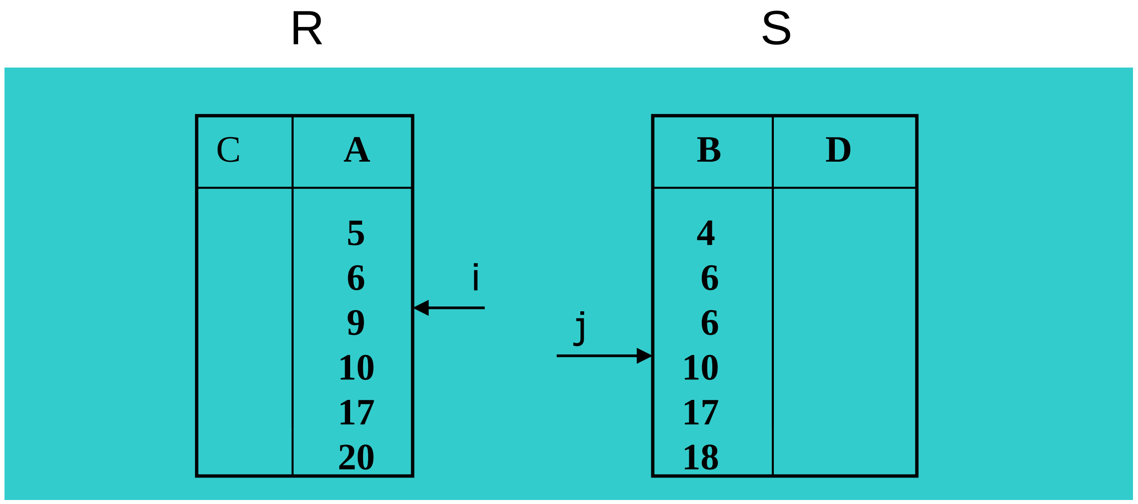
→ R(2), S(2)
→ R(2), S(3)



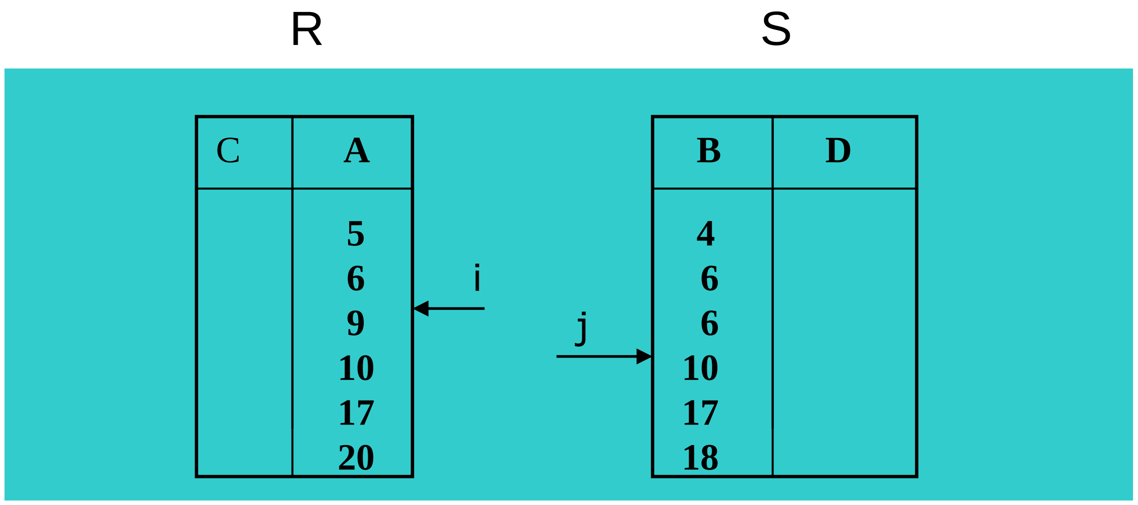
3.4. Algorithms for SELECT and JOIN Operations



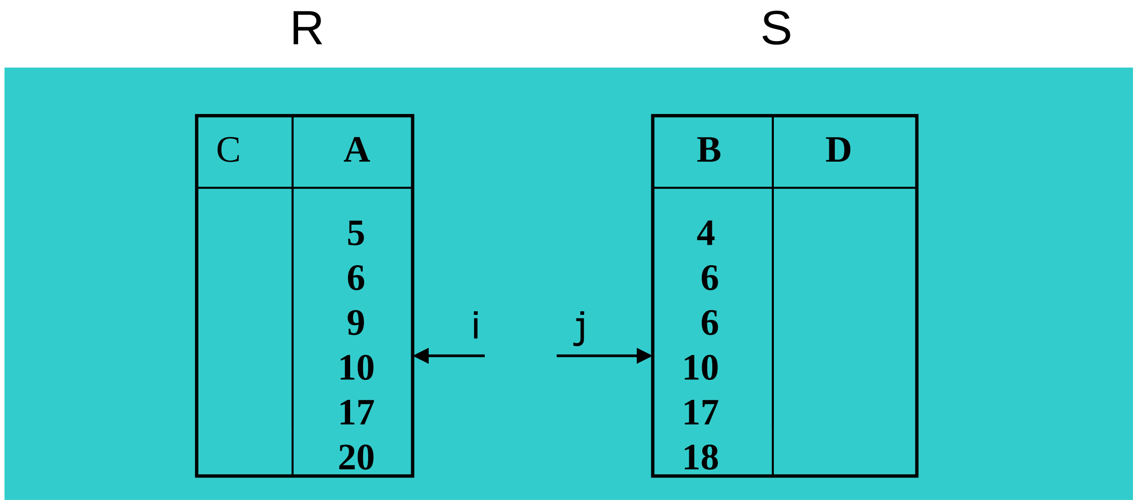
R(i)[A] > S(j)[B]



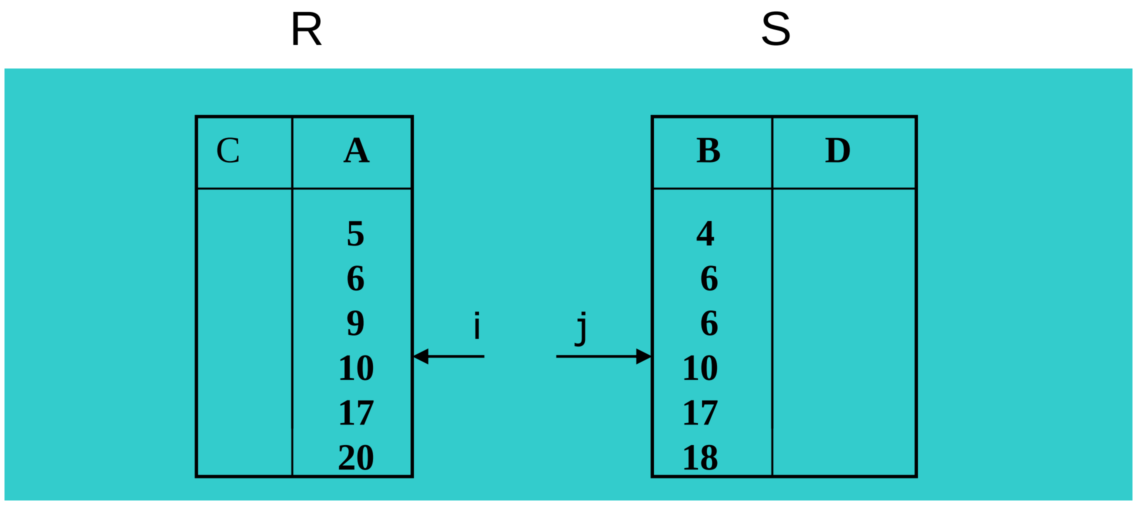
3.4. Algorithms for SELECT and JOIN Operations



R(i)[A] < S(j)[B]

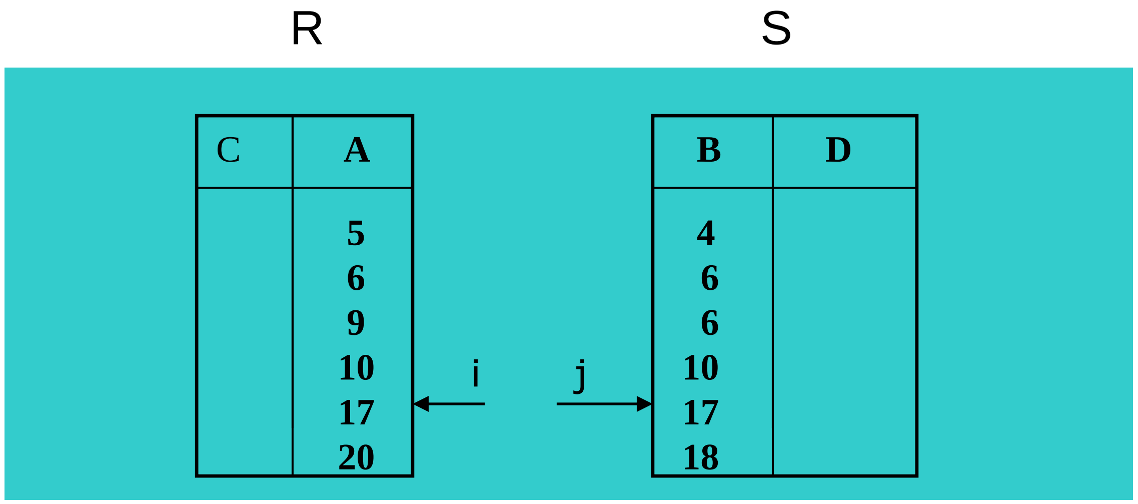


3.4. Algorithms for SELECT and JOIN Operations

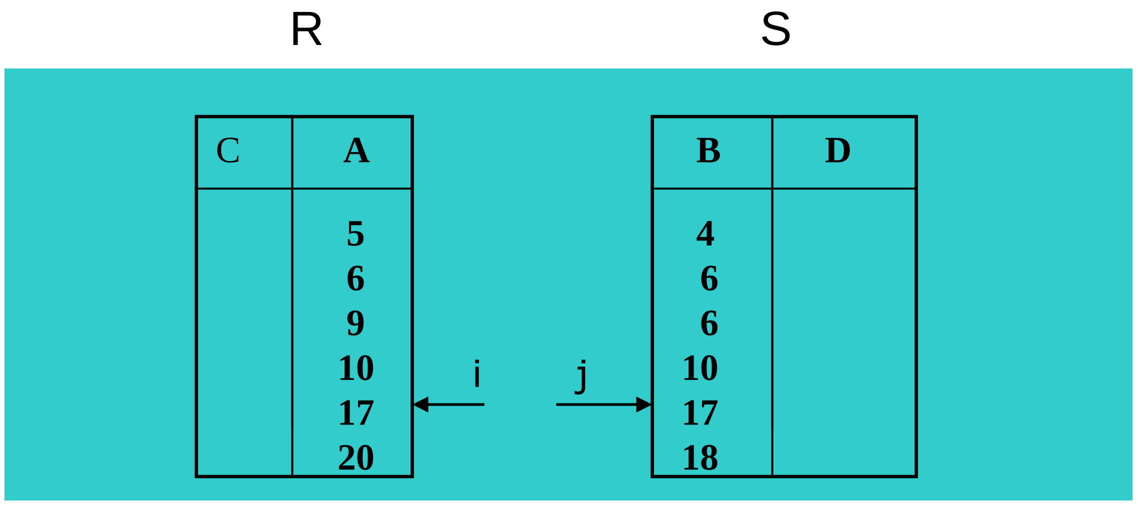


R(i)[A] = S(j)[B]

→ R(4), S(4)

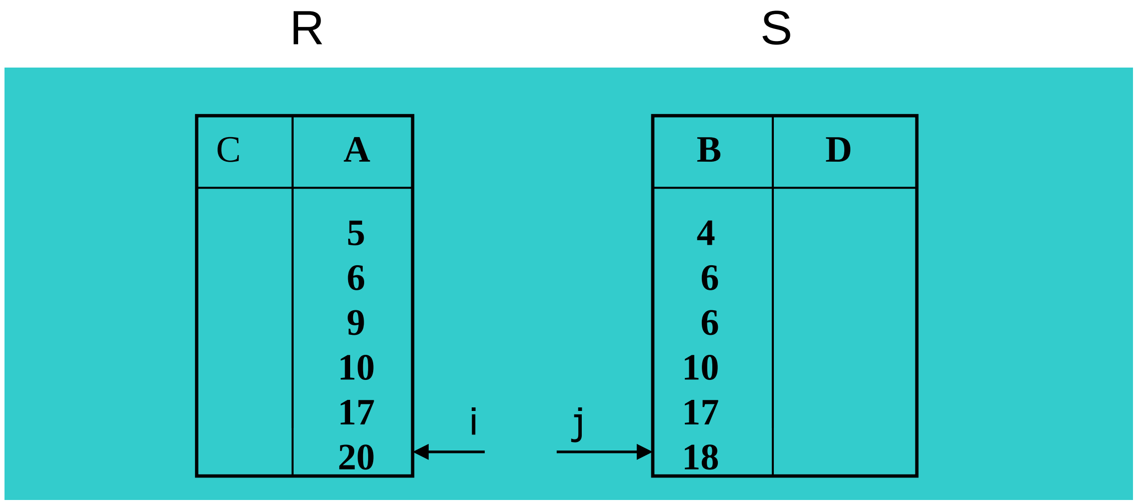


3.4. Algorithms for SELECT and JOIN Operations

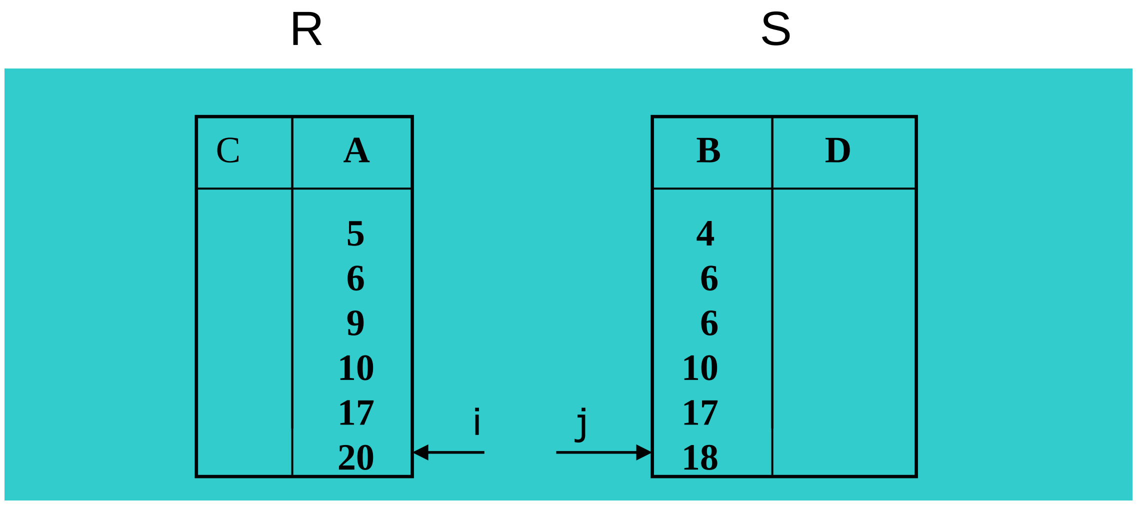


R(i)[A] = S(j)[B]

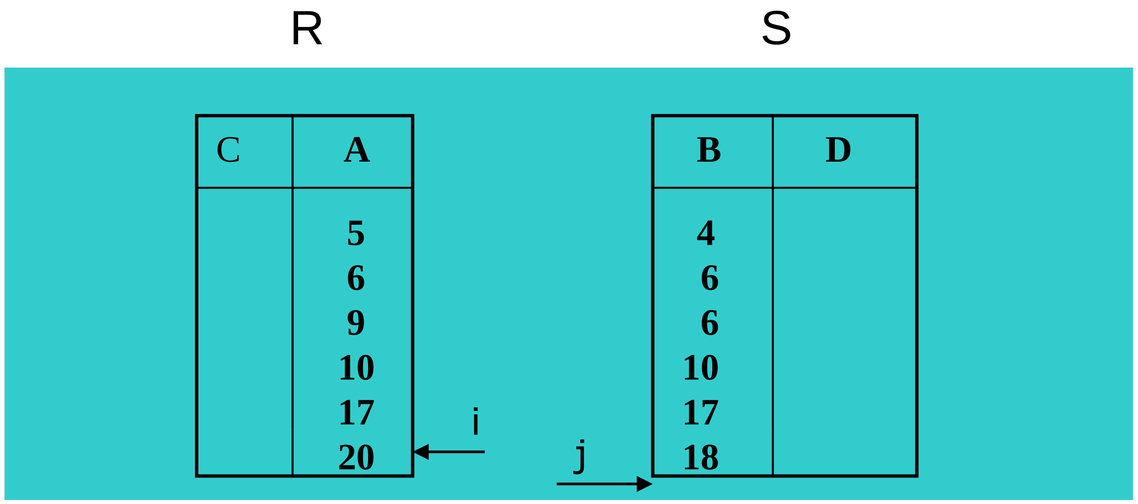
→ R(5), S(5)



3.4. Algorithms for SELECT and JOIN Operations



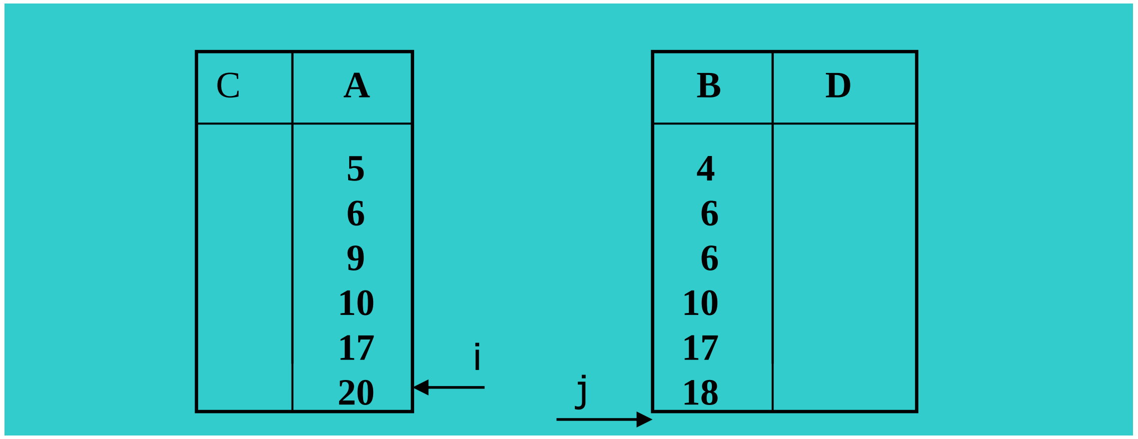
R(i)[A] > S(j)[B]



3.4. Algorithms for SELECT and JOIN Operations

J3. Implementing Sort-Merge Join: $T \leftarrow R \bowtie_{A=B} S$

R S



C	A
	5
	6
	9
	10
	17
	20

B	D
4	
6	
6	
10	
17	
18	

j > m → end.

Result:

R(2), S(2)

R(2), S(3)

R(4), S(4)

R(5), S(5)

C	A	B	D
	6	6	
	6	6	
	10	10	
	17	17	

3.4. Algorithms for SELECT and JOIN Operations

Implementing the JOIN Operation:

- J4. **Hash-join:** The records of files R and S are both hashed to the *same hash file,* using the *same hashing function* on the join attributes A of R and B of S as hash keys. A single pass through the *file with fewer records* (say, R) hashes its records to the hash file buckets. A single pass through the other file (S) then hashes each of its records to the appropriate bucket, where the record is combined with all matching records from R.

3.5. Algorithms for PROJECT and SET Operations

- **Algorithm for PROJECT operations**

$$\pi_{\textbf{<attribute list>}}(R)$$

- If <attribute list> has a key of relation R, extract all tuples from R with only the values for the attributes in <attribute list>.
- If <attribute list> does NOT include a key of relation R, duplicated tuples must be removed from the results.
- **Methods to remove duplicate tuples:**
    - **1. Sorting**
    - **2. Hashing**

3.5. Algorithms for PROJECT and SET Operations

Implementing $T \leftarrow \prod_{\text{<attribute list>}}(R)$

```
create a tuple t[<attribute list>] in T' for each tuple t in R;
/*T' contains the projection result before duplicate elimination*/
/*T contains the projection result after duplicate elimination*/
     if <attribute list> includes a key of R
     then T ← T'
     else  { sort the tuples in T';
                set i ← 1, j ← 2;
                while i ≤ n
                 do { output the tuple T'[i] to T;
                         while T'[i] = T'[j] and j ≤ n do j ← j+1;
                         set i ← j, j ← i+1;
                      }
             }
```

3.5. Algorithms for PROJECT and SET Operations

Algorithms for SET operations:

- **Set operations** : UNION, INTERSECTION, SET DIFFERENCE and CARTESIAN PRODUCT.
- **CARTESIAN PRODUCT** of relations R and S includes all possible combinations of records from R and S. The attributes of the result include all attributes of R and S.
- **Cost analysis** of CARTESIAN PRODUCT If R has *n* records and *j* attributes and S has *m* records and *k* attributes, the result relation will have *n*m* records and *j+k* attributes.
- CARTESIAN PRODUCT operation is very **expensive** and should be avoided if possible.

3.5. Algorithms for PROJECT and SET Operations

Algorithms for SET operations:

- **UNION:** R∪S
  - 1. Sort the two relations on the same attributes.
  - 2. Scan and merge both sorted files concurrently, whenever the same tuple exists in both relations, only one is kept in the merged results.
- **INTERSECTION:** R∩S
  - 1. Sort the two relations on the same attributes.
  - 2. Scan and merge both sorted files concurrently, keep in the merged results only those tuples that appear in both relations.
- **SET DIFFERENCE:** R-S
  - Keep in the merged results only those tuples that appear in relation R but not in relation S.

Union: $T \leftarrow R \cup S$

```
sort the tuples in R and S using the same unique sort attributes;
    set i ← 1, j ← 1;
    while (i ≤ n) and (j ≤ m) do
    {
        if R(i) > S(j)
        then
        {   output S(j) to T;
            set j ← j+1
        }
        elseif R(i) < S(j)
        then
        {   output R(i) to T;
            set i ← i+1
        }
        else set j← j+1
          /* R(i) = S(j), so we skip one of the duplicate tuples  */
    }
    if (i ≤ n) then add tuples R(i) to R(n) to T;
    if (j ≤ m) then add tuples S(j) to S(m) to T;
```

Intersection $T \leftarrow R \cap S$

```
sort the tuples in R and S using the same unique sort attributes;
    set i ← 1, j ← 1;
    while (i ≤ n) and (j ≤ m) do
    {
        if R(i) > S(j)
        then
            set j ← j+1
        elseif R(i) < S(j)
        then
            set i ← i+1
        else
        {
            output R(i) to T;
  /* R(i) = S(j), so we skip one of the duplicate tuples  */
            set  i ← i+1, j← j+1
        }
    }
```

Difference $T \leftarrow R - S$

```
sort the tuples in R and S using the same unique sort attributes;
    set i ← 1, j ← 1;
    while (i ≤ n) and (j ≤ m) do
    {
        if R(i) > S(j)
        then
            set j ← j+1
        elseif R(i) < S(j)
        then
        {
          output R(i) to T;
         /* R(i) has no matching S(j), so output R(i)  */
          set i ← i+1
        }
        else
            set  i ← i+1, j← j+1
    }
    if (i ≤ n) then add tuples R(i) to R(n) to T;
```

3.6. Implementing Aggregate Operations and Outer Joins

Implementing Aggregate Operations:

- **Aggregate operators** : MIN, MAX, SUM, COUNT and AVG
- **Options to implement aggregate operators:**
  - **Table Scan**
  - **Index**
- **Example:**

```
SELECT MAX(SALARY) FROM EMPLOYEE;
```

- If an (ascending) index on SALARY exists for the employee relation, then the optimizer could decide on traversing the index for the largest value, which would entail following the right most pointer in each index node from the root to a leaf.

3.6. Implementing Aggregate Operations and Outer Joins

Implementing Aggregate Operations:

- **SUM, COUNT and AVG**
  - 1. For a **dense index** (each record has one index entry): apply the associated computation to the values in the index.
  - 2. For a **non-dense index**: actual number of records associated with each index entry must be accounted for
- With **GROUP BY:** the aggregate operator must be applied separately to each group of tuples.
  - 1. Use sorting or hashing on the group attributes to partition the file into the appropriate groups;
  - 2. Compute the aggregate function for the tuples in each group.
- What if we have **Clustering index** on the grouping attributes?

3.6. Implementing Aggregate Operations and Outer Joins

Implementing Outer Join:

- **Outer Join Operators** : LEFT OUTER JOIN, RIGHT OUTER JOIN and FULL OUTER JOIN.
- The full outer join produces a result which is equivalent to the union of the results of the left and right outer joins.
- **Example:**

```
SELECT FNAME, DNAME
FROM EMPLOYEE LEFT OUTER JOIN DEPARTMENT
                              ON DNO = DNUMBER;
```

- **Note:** The result of this query is a table of employee names and their associated departments. It is similar to a regular join result, with the exception that if an employee does not have an associated department, the employee's name will still appear in the resulting table, although the department name would be indicated as null.

3.6. Implementing Aggregate Operations and Outer Joins

Implementing Outer Join:

- **Modifying Join Algorithms:**
  Nested Loop or Sort-Merge joins can be modified to implement outer join. e.g., for left outer join, use the left relation as outer relation and construct result from every tuple in the left relation. If there is a match, the concatenated tuple is saved in the result. However, if an outer tuple does not match, then the tuple is still included in the result but is padded with a null value(s).

3.6. Implementing Aggregate Operations and Outer Joins

Implementing Outer Join:

Implement the previous left outer join example:

1. Compute the JOIN of the EMPLOYEE and DEPARTMENT tables:

   $TEMP1 \leftarrow \pi_{FNAME,DNAME}(EMPLOYEE \bowtie_{DNO=DNUMBER} DEPARTMENT)$

2. Find the EMPLOYEEs that do not appear in the JOIN:

   $TEMP2 \leftarrow \pi_{FNAME}(EMPLOYEE) - \pi_{FNAME}(Temp1)$

3. Pad each tuple in TEMP2 with a null DNAME field:

   $TEMP2 \leftarrow TEMP2 \times \text{'null'}$

4. UNION the temporary tables to produce the LEFT OUTER JOIN result:

   $RESULT \leftarrow TEMP1 \cup TEMP2$

The cost of the outer join, as computed above, would include the cost of the associated steps (i.e., join, projections and union).

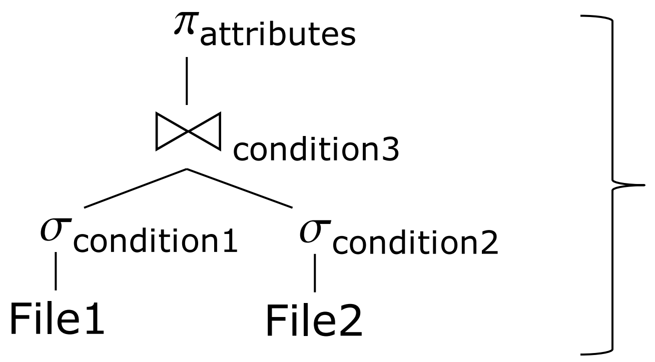
3.7. Combining Operations using Pipelining

- **Motivation**
  - A query is mapped into a sequence of operations.
    - Each execution of an operation produces a temporary result.
  - Generating and saving temporary files on disk is time consuming and expensive.
- **Alternative:**
  - Avoid constructing temporary results as much as possible
  - Pipeline the data through multiple operations - pass the result of a previous operator to the next without waiting to complete the previous operation

→ **Pipelining (stream-based processing)**

3.7. Combining Operations using Pipelining

- **Example:** A 2-way join is combined with the 2 selections on the inputs and one projection on the output.


$\pi_{\text{attributes}}$
$\bowtie_{\text{condition3}}$
$\sigma_{\text{condition1}}$
$\sigma_{\text{condition2}}$
File1
File2

One algorithm with pipelining
- Two input files
- One output file
- No four temporary files

- Dynamic code generation to allow for multiple operations to be pipelined
- Results of a select operation are fed in a "**pipeline**" to the join algorithm.

Introduction to Query Optimization

- **Query optimization**: the process of choosing a suitable execution strategy for processing a query.
- **Goal**: to arrive at the most efficient and cost-effective plan using the available information about the schema and the content of relations involved, and to do so in a reasonable amount of time
  - The chosen execution plan may not always be the most optimal plan possible!!!

→ Heuristic rule-based vs. cost-based optimization

3.8. Using Heuristics in Query Optimization

Process for heuristics optimization

1. The parser of a high-level query generates an *initial internal representation* (a query tree).

2. Apply heuristics rules to optimize the internal representation.

3. A query execution plan is generated to execute groups of operations based on the access paths available on the files.

- One of the **main heuristic rules** is to apply first the operations that reduce the size of intermediate results: SELECT and PROJECT before JOIN or other binary operations.

3.8. Using Heuristics in Query Optimization

- **Example**:

  For every project located in ‘Stafford’, retrieve the project number, the controlling department number and the department manager’s last name, address and birthdate.

- **SQL query**:

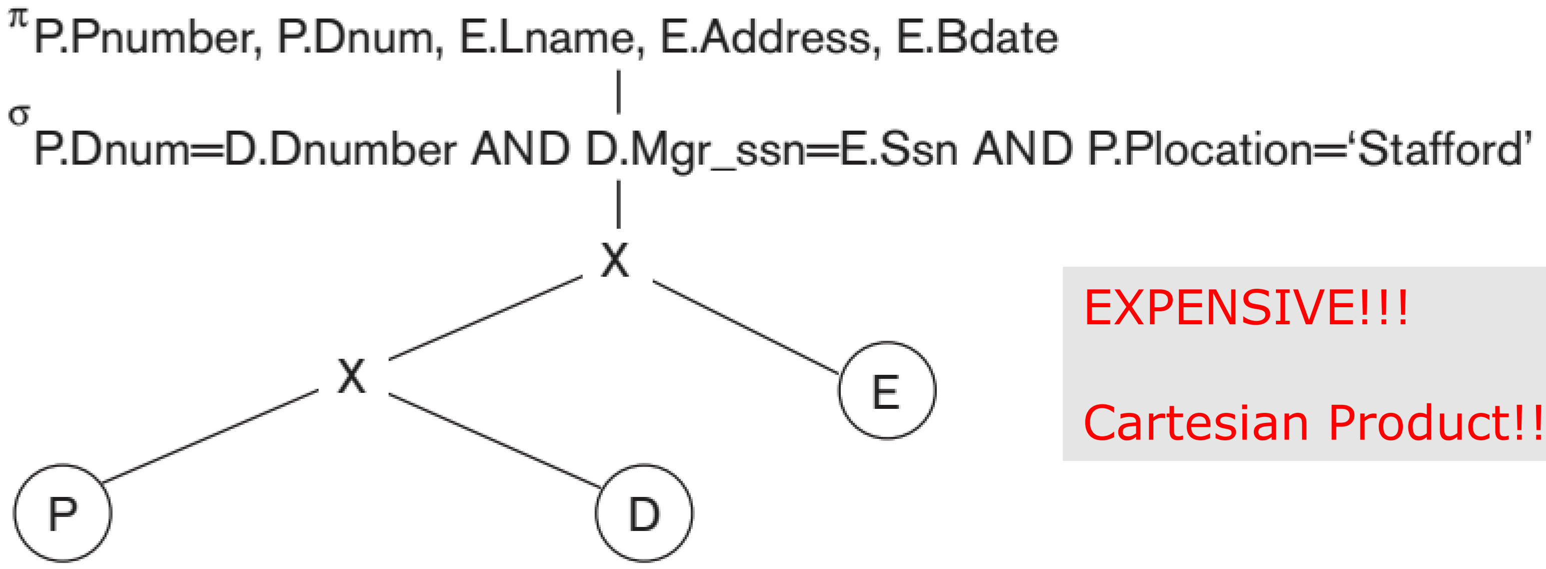
```
SELECT P.NUMBER, P.DNUM, E.LNAME, E.ADDRESS, E.BDATE

FROM PROJECT AS P,DEPARTMENT AS D, EMPLOYEE AS E

WHERE  P.DNUM=D.DNUMBER AND D.MGRSSN=E.SSN AND
P.PLOCATION='STAFFORD';
```

- **Relation algebra**:

$$\pi_{\text{PNUMBER, DNUM, LNAME, ADDRESS, BDATE}}(((\sigma_{\text{PLOCATION='STAFFORD'}}(\text{PROJECT})) \bowtie_{\text{DNUM=DNUMBER}} (\text{DEPARTMENT})) \bowtie_{\text{MGRSSN=SSN}} (\text{EMPLOYEE}))$$

3.8. Using Heuristics in Query Optimization

```
SELECT P.NUMBER, P.DNUM, E.LNAME, E.ADDRESS, E.BDATE
FROM PROJECT AS P, DEPARTMENT AS D, EMPLOYEE AS E
WHERE  P.DNUM=D.DNUMBER AND D.MGRSSN=E.SSN
         AND P.PLOCATION='STAFFORD';
```

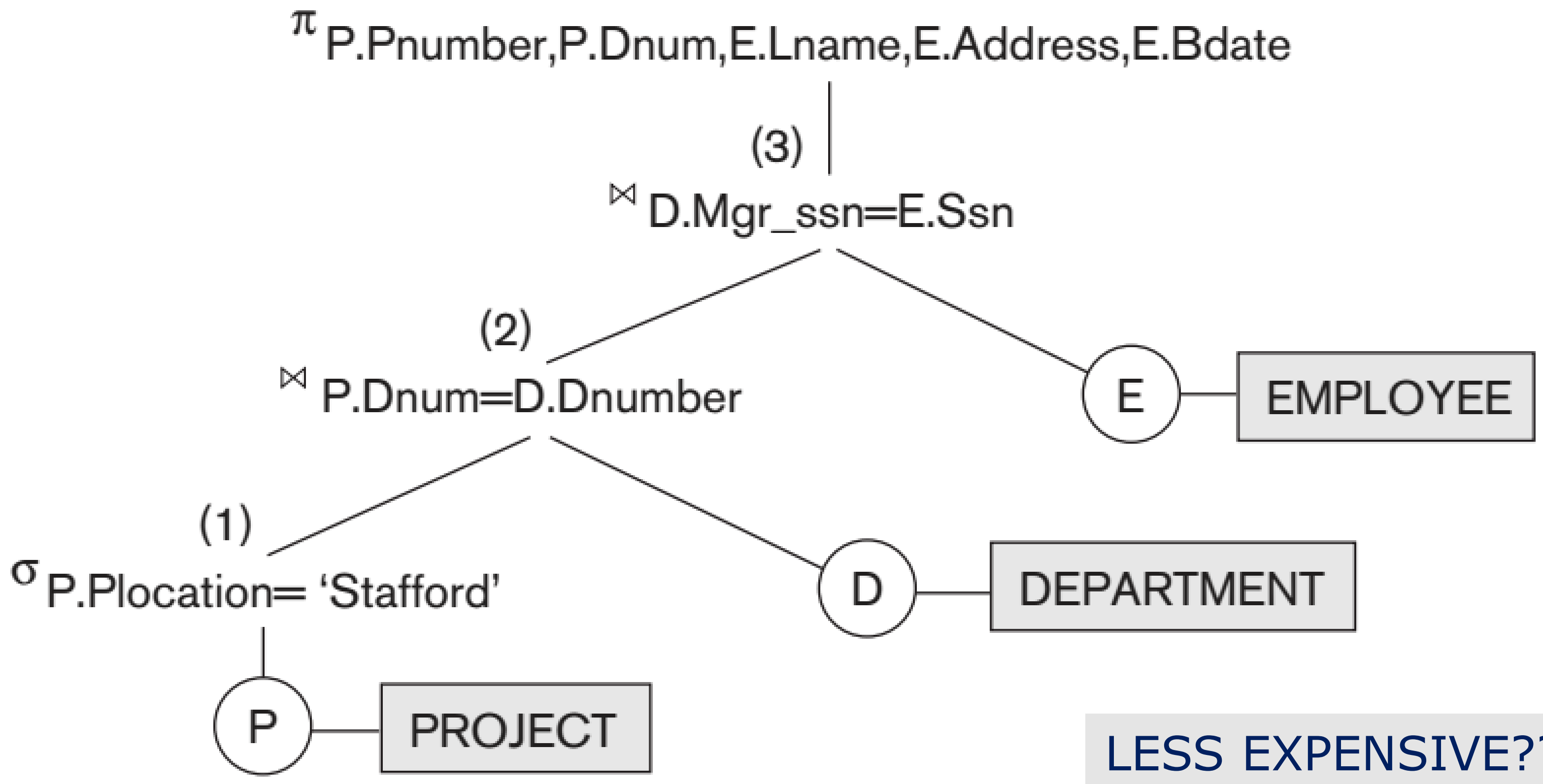

π P.Pnumber, P.Dnum, E.Lname, E.Address, E.Bdate
σ P.Dnum=D.Dnumber AND D.Mgr_ssn=E.Ssn AND P.Plocation='Stafford'
X
X
E
P
D

EXPENSIVE!!!

Cartesian Product!!!

An initial query tree
Figure 19.1.b, [1], pp. 693

3.8. Using Heuristics in Query Optimization



π P.Pnumber,P.Dnum,E.Lname,E.Address,E.Bdate
(3)
$\bowtie$ D.Mgr_ssn=E.Ssn
(2)
$\bowtie$ P.Dnum=D.Dnumber
E
EMPLOYEE
(1)
σ P.Plocation= 'Stafford'
D
DEPARTMENT
P
PROJECT

LESS EXPENSIVE???

NO Cartesian Product!!!

An initial query tree
Figure 19.1.a, [1], pp. 693

3.8. Using Heuristics in Query Optimization

Heuristic Optimization of Query Trees:

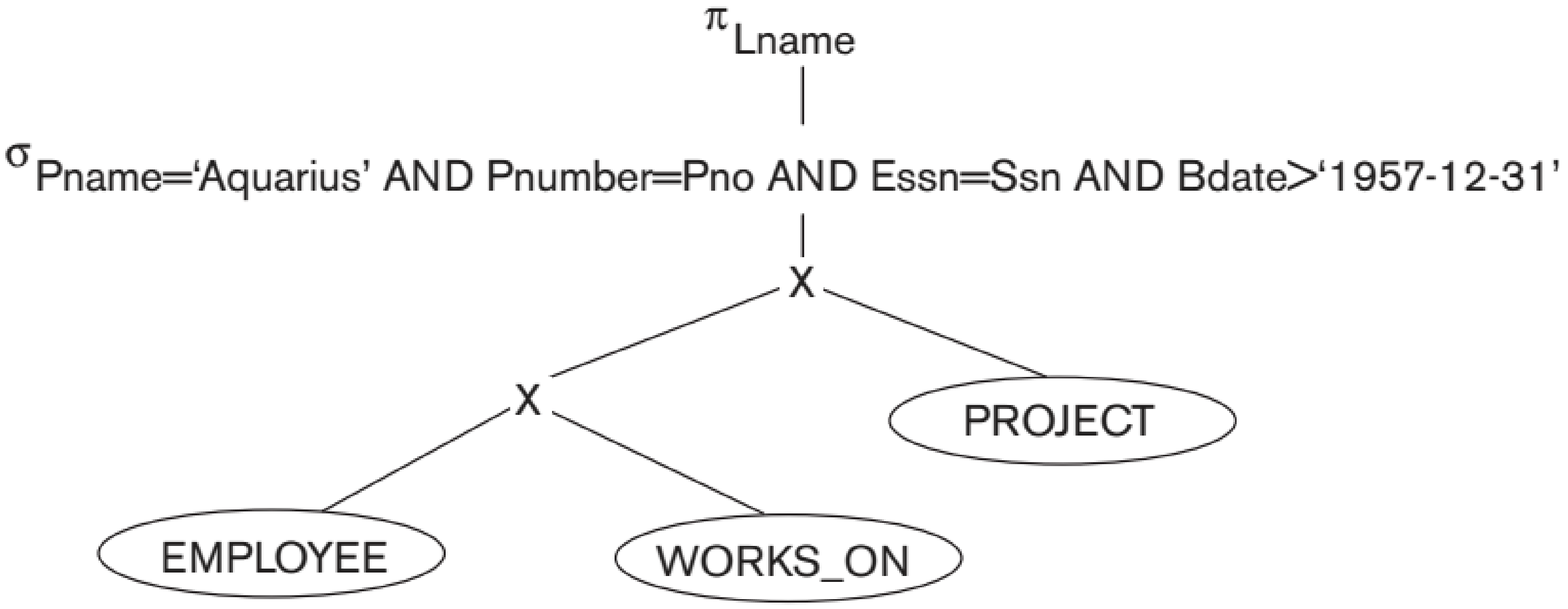
- The same query could correspond to many different relational algebra expressions — and hence many different query trees.
- The task of heuristic optimization of query trees is to find a **final query tree** that is efficient to execute.

→ Transform an initial query tree to a more efficient query tree by applying heuristic rules

3.8. Using Heuristics in Query Optimization

- **Example**: Find the last names of employees born after 1957 who work on a project named ‘Aquarius’

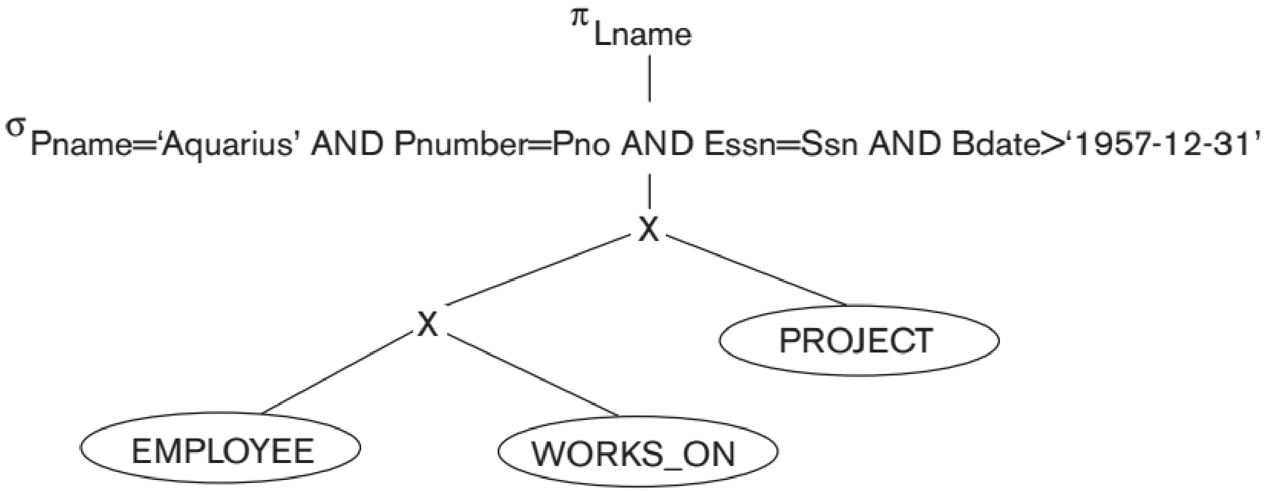
```
SELECT LNAME
FROM EMPLOYEE, WORKS_ON,PROJECT
WHERE  PNAME = 'AQUARIUS' AND PNMUBER=PNO
               AND ESSN=SSN AND BDATE > '1957-12-31';
```



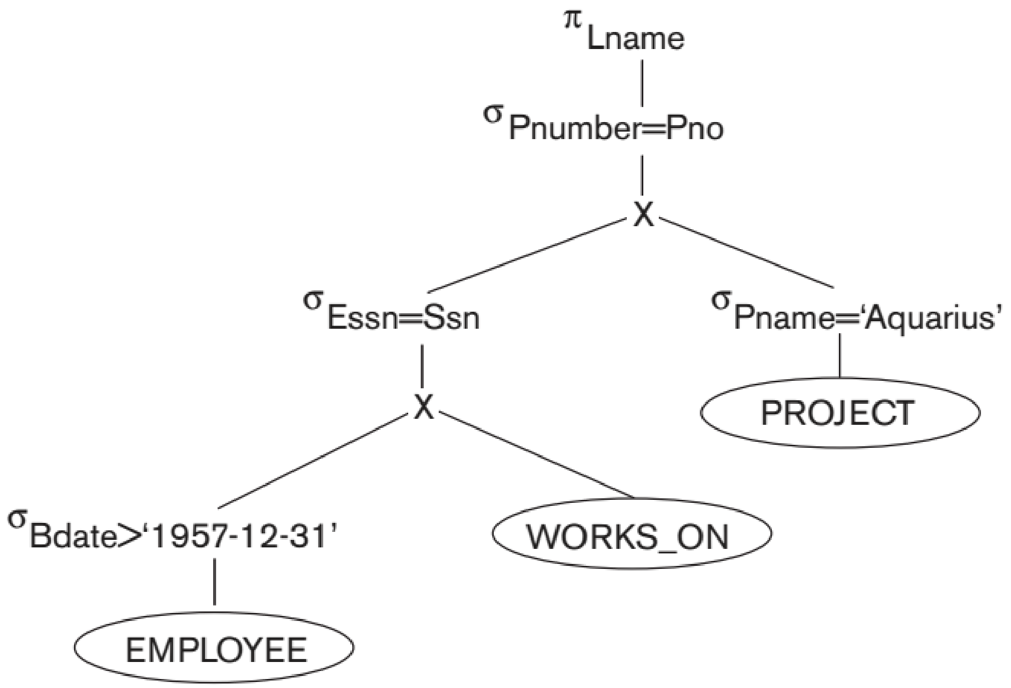
π_{Lname}
$\sigma_{\text{Pname='Aquarius' AND Pnumber=Pno AND Essn=Ssn AND Bdate>'1957-12-31'}}$
X
X
PROJECT
EMPLOYEE
WORKS_ON

```
SELECT LNAME
FROM EMPLOYEE, WORKS_ON,PROJECT
WHERE  PNAME = 'AQUARIUS' AND PNMUBER=PNO
                AND ESSN=SSN AND BDATE > '1957-12-31';
```

An initial query tree



π_{Lname}
$\sigma_{\text{Pname='Aquarius' AND Pnumber=Pno AND Essn=Ssn AND Bdate>'1957-12-31'}}$
X
X
PROJECT
EMPLOYEE
WORKS_ON



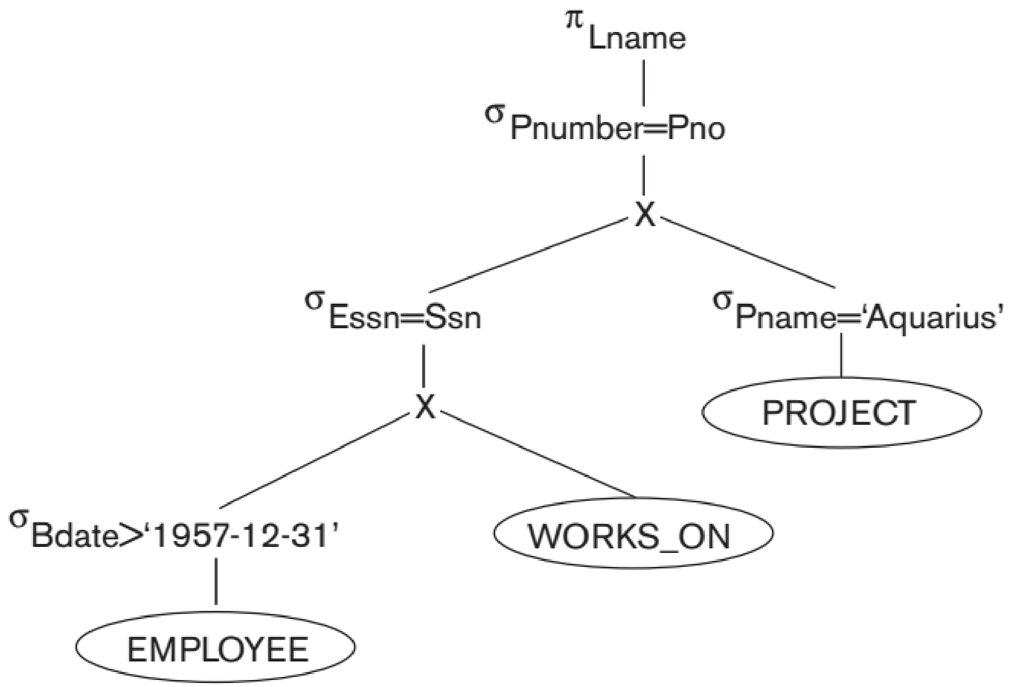
π_{Lname}
$\sigma_{\text{Pnumber=Pno}}$
X
$\sigma_{\text{Essn=Ssn}}$
$\sigma_{\text{Pname='Aquarius'}}$
X
PROJECT
$\sigma_{\text{Bdate>'1957-12-31'}}$
WORKS_ON
EMPLOYEE

Moving SELECT down the tree to their direct relations

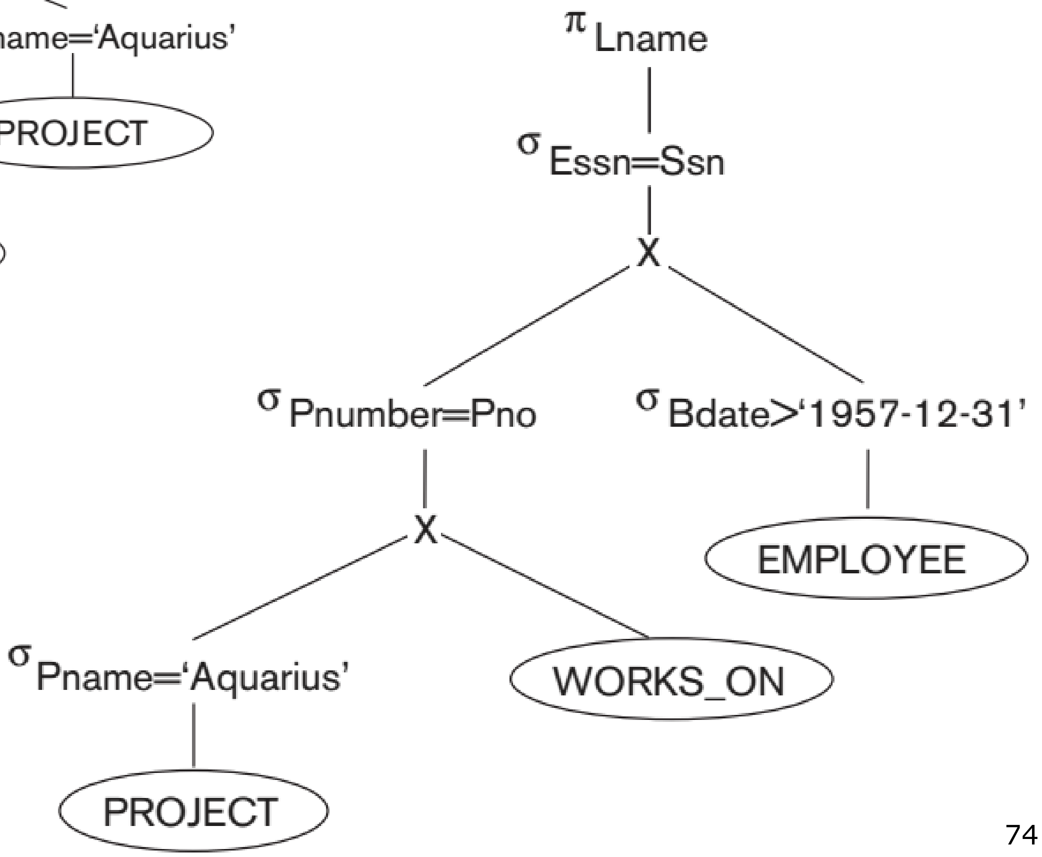
→ Reduce the size of each temporary result

```
SELECT LNAME
FROM EMPLOYEE, WORKS_ON,PROJECT
WHERE  PNAME = 'AQUARIUS' AND PNMUBER=PNO
            AND ESSN=SSN AND BDATE > '1957-12-31';
```

A previous query tree



π_{Lname}
$\sigma_{\text{Pnumber=Pno}}$
X
$\sigma_{\text{Essn=Ssn}}$
$\sigma_{\text{Pname='Aquarius'}}$
X
PROJECT
$\sigma_{\text{Bdate>'1957-12-31'}}$
WORKS_ON
EMPLOYEE



π_{Lname}
$\sigma_{\text{Essn=Ssn}}$
X
$\sigma_{\text{Pnumber=Pno}}$
$\sigma_{\text{Bdate>'1957-12-31'}}$
X
EMPLOYEE
$\sigma_{\text{Pname='Aquarius'}}$
WORKS_ON
PROJECT

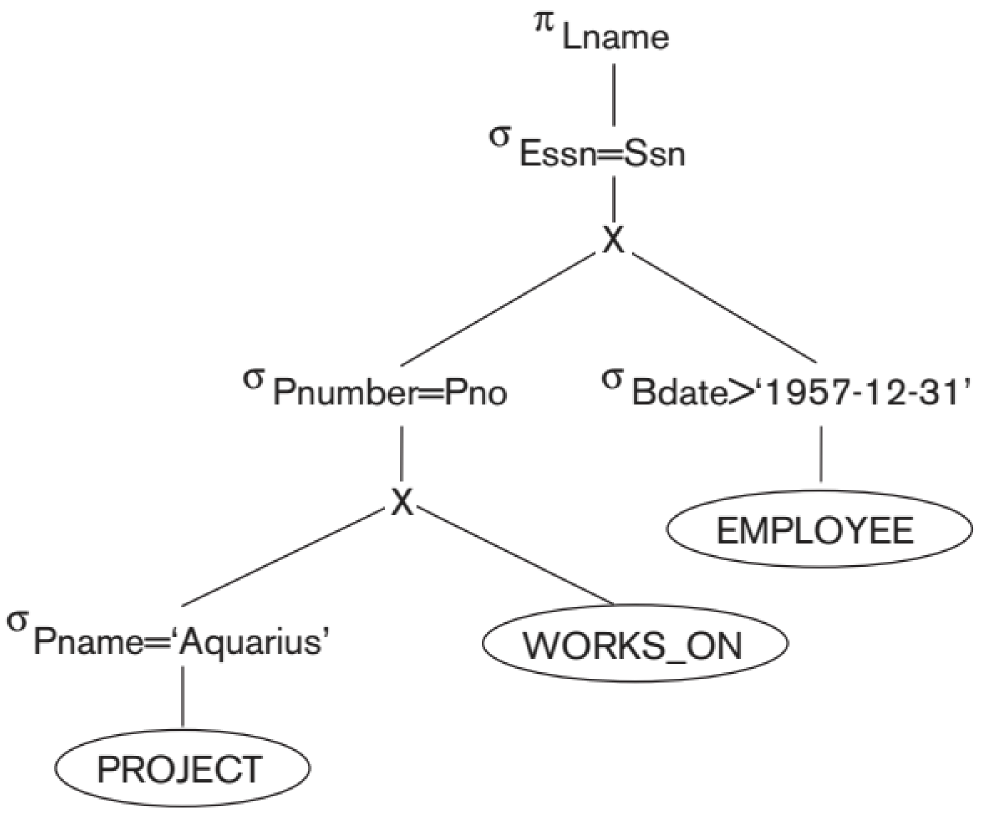
Apply the more restrictive SELECT operation first

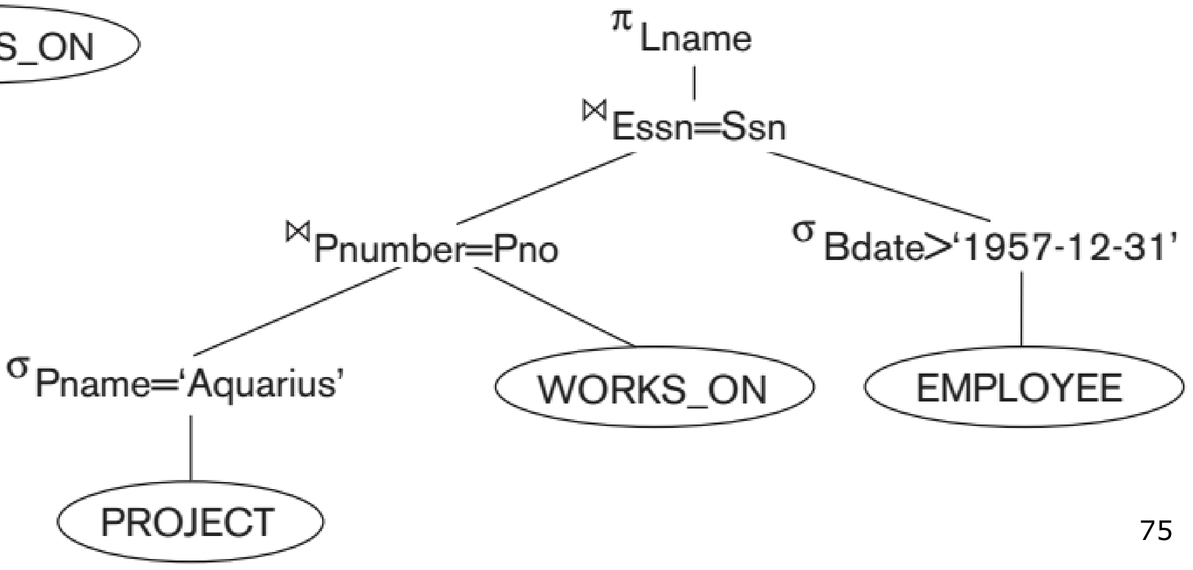
→ swap branches

→ reduce the size of each temporary result

```
SELECT LNAME
FROM EMPLOYEE, WORKS_ON,PROJECT
WHERE  PNAME = 'AQUARIUS' AND PNMUBER=PNO
              AND ESSN=SSN AND BDATE > '1957-12-31';
```

A previous query tree



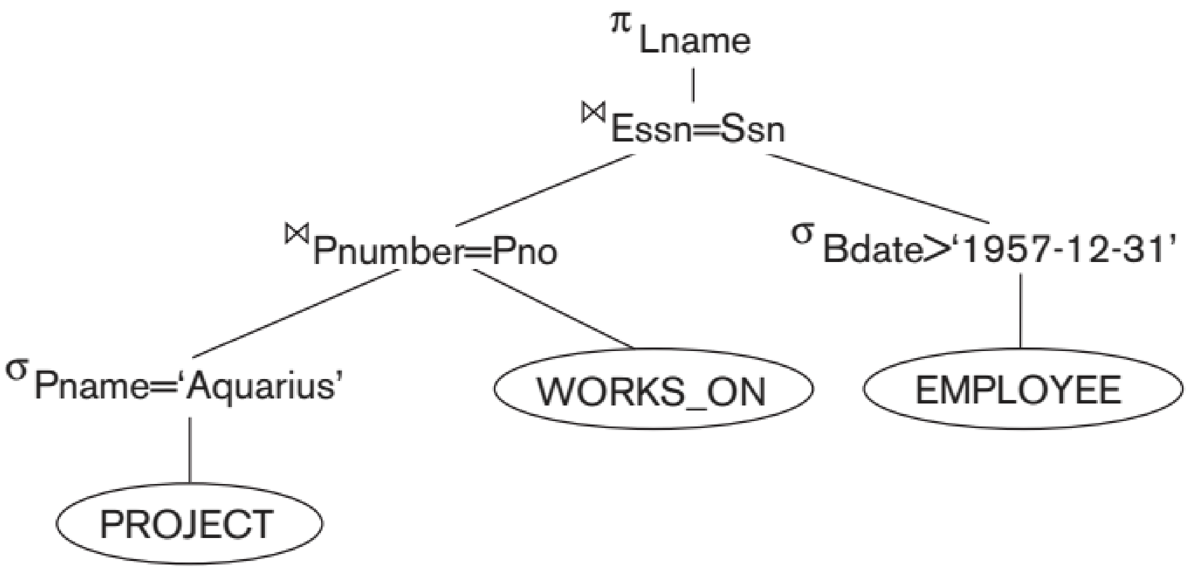


Cartesian Product

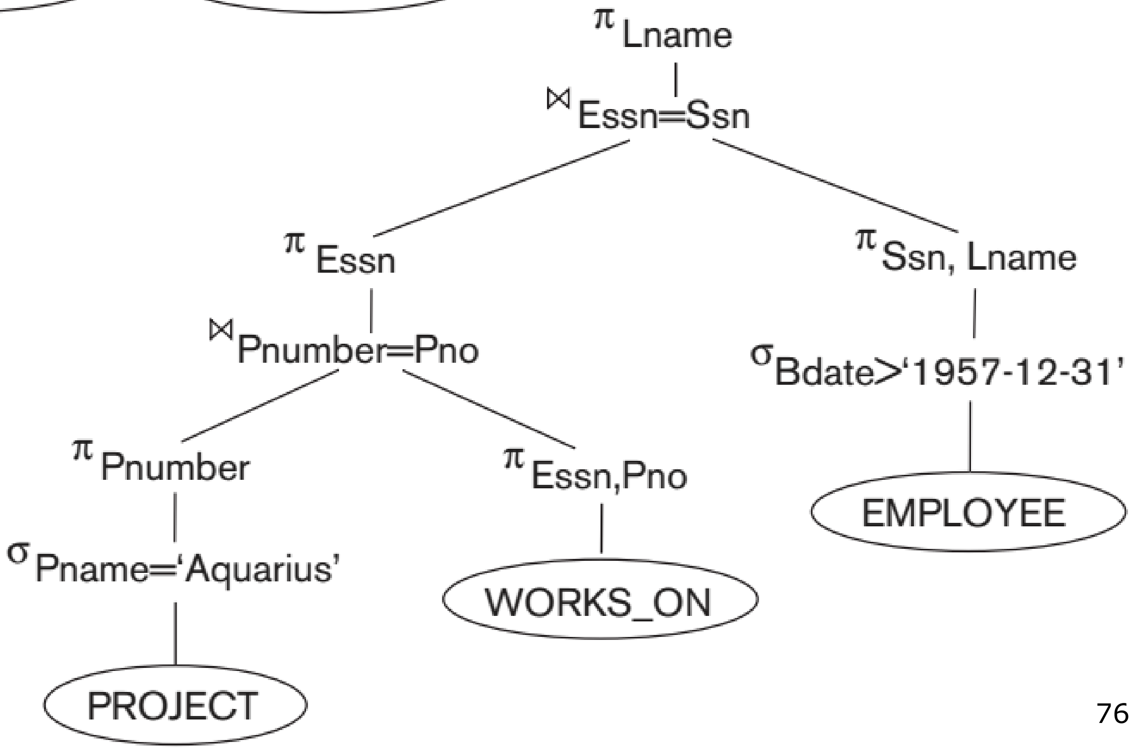
+ Select

= Join

```
SELECT LNAME
FROM EMPLOYEE, WORKS_ON,PROJECT
WHERE  PNAME = 'AQUARIUS' AND PNMUBER=PNO
           AND ESSN=SSN AND BDATE > '1957-12-31';
```



A previous query tree



Move PROJECT down the query tree

→ Reduce the size of each intermediate result

3.8. Using Heuristics in Query Optimization

- **General Transformation Rules for Relational Algebra Operations:**

  1. Cascade of σ: A conjunctive selection condition can be broken up into a cascade (sequence) of individual selection operations:

  $$\sigma_{c1 \text{ AND } c2 \text{ AND } \ldots \text{ AND } cn}(R) = \sigma_{c1}(\sigma_{c2}(\ldots(\sigma_{cn}(R))\ldots))$$

  2. Commutativity of σ: The σ operation is commutative:

  $$\sigma_{c1}(\sigma_{c2}(R)) = \sigma_{c2}(\sigma_{c1}(R))$$

  3. Cascade of π: In a cascade (sequence) of π operations, all but the last one can be ignored:

  $$\pi_{List1}(\pi_{List2}(\ldots(\pi_{Listn}(R))\ldots)) = \pi_{List1}(R)$$

  4. Commuting σ with π: If the selection condition *c* involves only the attributes A1, ..., An in the projection list, the two operations can be commuted:

  $$\pi_{A1, A2, ., An}(\sigma_c(R)) = \sigma_c(\pi_{A1, A2, ., An}(R))$$

3.8. Using Heuristics in Query Optimization

- **General Transformation Rules for Relational Algebra Operations:**

5. Commutativity of $\bowtie$ (and $\times$): The operation is commutative as the $\times$ operation:

$$R \bowtie S = S \bowtie R; \quad R \times S = S \times R$$

6. Commuting σ with $\bowtie$ (or $\times$): If all the attributes in the selection condition *c* involve only the attributes of one of the relations being joined - say, *R*- the two operations can be commuted as follows :

$$\sigma_c(R \bowtie S) = (\sigma_c(R)) \bowtie S$$

Alternatively, if the selection condition *c* can be written as (*c1* and *c2*), where condition *c1* involves only the attributes of *R* and condition *c2* involves only the attributes of *S*, the operations commute as follows:

$$\sigma_c(R \bowtie S) = (\sigma_{c1}(R)) \bowtie (\sigma_{c2}(S))$$

3.8. Using Heuristics in Query Optimization

- **General Transformation Rules for Relational Algebra Operations:**

  7. Commuting π with $\bowtie$ (or $\times$): Suppose that the projection list is L = {A1, ..., An, B1, ..., Bm}, where A1, ..., An are attributes of R and B1, ..., Bm are attributes of S. If the join condition *c* involves only attributes in *L*, the two operations can be commuted as follows:

$$\pi_L(R \bowtie_C S) = (\pi_{A1, \ldots, An}(R)) \bowtie_C (\pi_{B1, \ldots, Bm}(S))$$

  If the join condition *c* contains additional attributes not in *L*, these must be added to the projection list, and a final operation is needed.

3.8. Using Heuristics in Query Optimization

- **General Transformation Rules for Relational Algebra Operations:**

8. **Commutativity of set operations**: The set operations $\cup$ and $\cap$ are commutative but $-$ is not.

9. **Associativity of $\bowtie$, x, $\cup$, and** $\cap$: These four operations are individually associative; that is, if θ stands for any one of these four operations (throughout the expression), we have

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

10. **Commuting σ with set operations**: The σ operation commutes with $\cup$, $\cap$, and $-$. If θ stands for any one of these three operations, we have

$$\sigma_c (R \ \theta \ S) = (\sigma_c (R)) \ \theta \ (\sigma_c (S))$$

3.8. Using Heuristics in Query Optimization

- **General Transformation Rules for Relational Algebra Operations:**

11. **The π operation commutes with** $\cup$:

$$\pi_L(R \cup S) \quad = \quad (\pi_L(R)) \cup (\pi_L(S))$$

12. **Converting a (σ, $\times$) sequence into** $\bowtie$: If the condition *c* of a σ that follows a $\times$ corresponds to a join condition, convert the (σ, $\times$) sequence into a $\bowtie$ as follows:

$$(\sigma_C(R \times S)) \quad = \quad (R \bowtie_C S)$$

3.8. Using Heuristics in Query Optimization

- **General Transformation Rules for Relational Algebra Operations:**

13. Pushing σ in conjunction with set difference:

$$\sigma_c(R - S) = \sigma_c(R) - \sigma_c(S)$$

However, **σ** may be applied to only one relation**:**

$$\sigma_c(R - S) = \sigma_c(R) - S$$

14. Pushing σ to only one argument in ∩:

If in the condition σ_c all attributes are from relation R, then:

$$\sigma_c(R \cap S) = \sigma_c(R) \cap S$$

15. Some trivial transformations:

If S is empty, then $R \cup S = R$.
If the condition c in σ_c is true for the entire R, then $\sigma_c(R) = R$.

3.8. Using Heuristics in Query Optimization

A Heuristic Algebraic Optimization Algorithm

- 1. Using rule 1, break up any select operations with conjunctive conditions into a cascade of select operations.
- 2. Using rules 2, 4, 6, and 10, 13, 14 concerning the commutativity of select with other operations, move each select operation as far down the query tree as is permitted by the attributes involved in the select condition.
- 3. Using rules 5 and 9 concerning commutativity and associativity of binary operations, rearrange the leaf nodes of the tree so that the leaf node relations with the most restrictive select operations are executed first in the query tree representation.
- 4. Using Rule 12, combine a cartesian product operation with a subsequent select operation in the tree into a join operation.

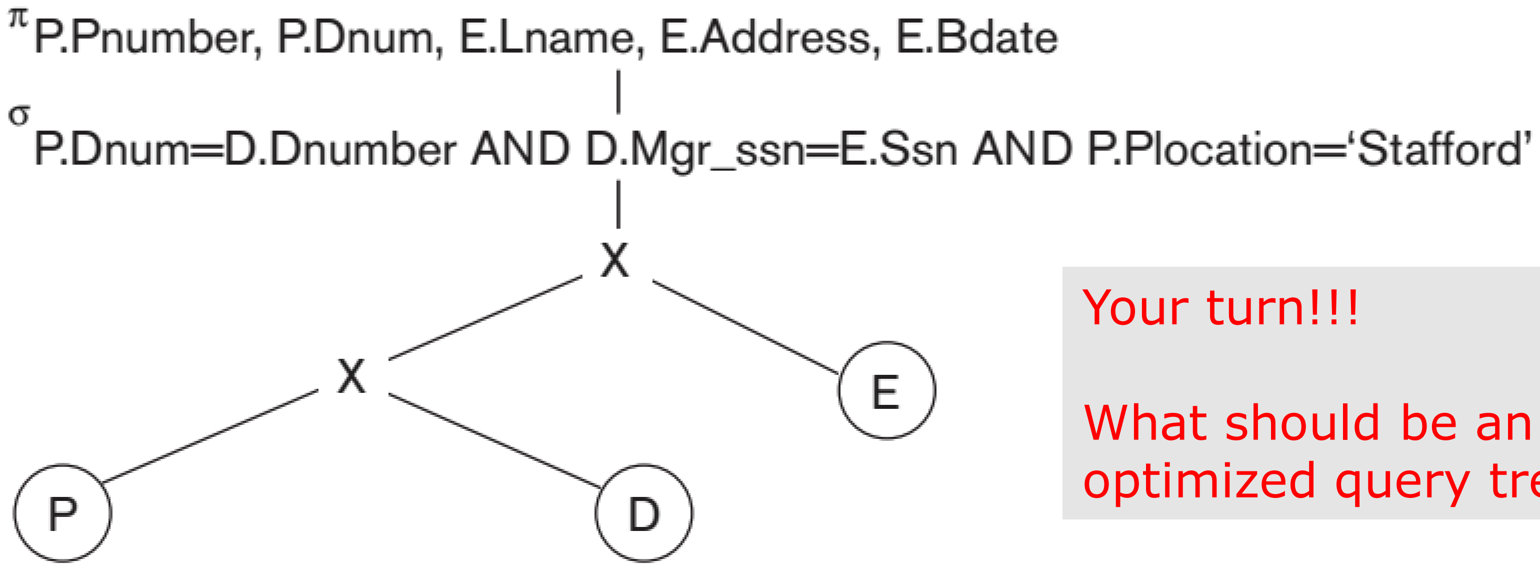
3.8. Using Heuristics in Query Optimization

A Heuristic Algebraic Optimization Algorithm:

- 5. Using rules 3, 4, 7, and 11 concerning the cascading of project and the commuting of project with other operations, break down and move lists of projection attributes down the tree as far as possible by creating new project operations as needed. Only those attributes needed in the query result and in subsequent operations in the query tree should be kept after each PROJECT operation.
- 6. Identify subtrees that represent groups of operations that can be executed by a single algorithm.

3.8. Using Heuristics in Query Optimization

```
SELECT P.NUMBER, P.DNUM, E.LNAME, E.ADDRESS, E.BDATE
FROM PROJECT AS P, DEPARTMENT AS D, EMPLOYEE AS E
WHERE  P.DNUM=D.DNUMBER AND D.MGRSSN=E.SSN
          AND P.PLOCATION='STAFFORD';
```


π P.Pnumber, P.Dnum, E.Lname, E.Address, E.Bdate
σ P.Dnum=D.Dnumber AND D.Mgr_ssn=E.Ssn AND P.Plocation='Stafford'
X
X
E
P
D

Your turn!!!

What should be an optimized query tree?

An initial query tree
Figure 19.1.b, [1], pp. 693

3.8. Using Heuristics in Query Optimization

Summary of Heuristics for Algebraic Optimization:

- The main heuristic is to apply first the operations that reduce the size of intermediate results.
  - performing as early as possible SELECT operations to reduce the number of tuples and PROJECT operations to reduce the number of attributes—by moving SELECT and PROJECT operations as far down the tree as possible
- The SELECT and JOIN operations that are most restrictive—that is, result in relations with the fewest tuples or with the smallest absolute size—should be executed before other similar operations.
- Reordering the leaf nodes of the tree among themselves while avoiding Cartesian products, and adjusting the rest of the tree appropriately

3.8. Using Heuristics in Query Optimization

How about nested subquery optimization?

- Case 1: a query block inside an outer query block
  - Evaluation of this query involves executing the nested query first, which yields a single value.
  - The outer block is simply executed with the selection condition using the resulting value of the inner block.
- Case 2: correlated nested queries
  - In a correlated subquery, the inner query contains a reference to the outer query via one or more variables. The subquery acts as a function that returns a set of values for each value of this variable or combination of variables.
  - Wherever possible, SQL optimizer tries to convert queries with nested subqueries into a join operation.
  - Other techniques used include creation of temporary result tables from subqueries and using them in joins.

3.8. Using Heuristics in Query Optimization

How about nested subquery optimization?

- Case 1: a query block inside an outer query block

```
SELECT E1.Fname, E1.Lname
FROM EMLOYEE E1
WHERE E1.Salary = ( SELECT MAX (Salary)
                    FROM EMPLOYEE E2)
```

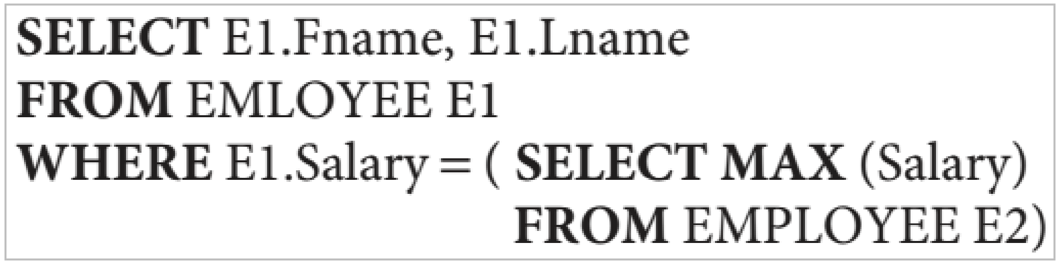
- Case 2: correlated nested queries

```
SELECT Fname, Lname, Salary
FROM EMPLOYEE E
WHERE EXISTS ( SELECT *
               FROM DEPARTMENT D
               WHERE D.Dnumber = E.Dno AND D.Zipcode=30332);
```

```
SELECT COUNT(*)
FROM DEPARTMENT D
WHERE D.Dnumber IN ( SELECT E.Dno
                     FROM EMPLOYEE E
                     WHERE E.Salary > 200000)
```

3.8. Using Heuristics in Query Optimization

How about nested subquery optimization?

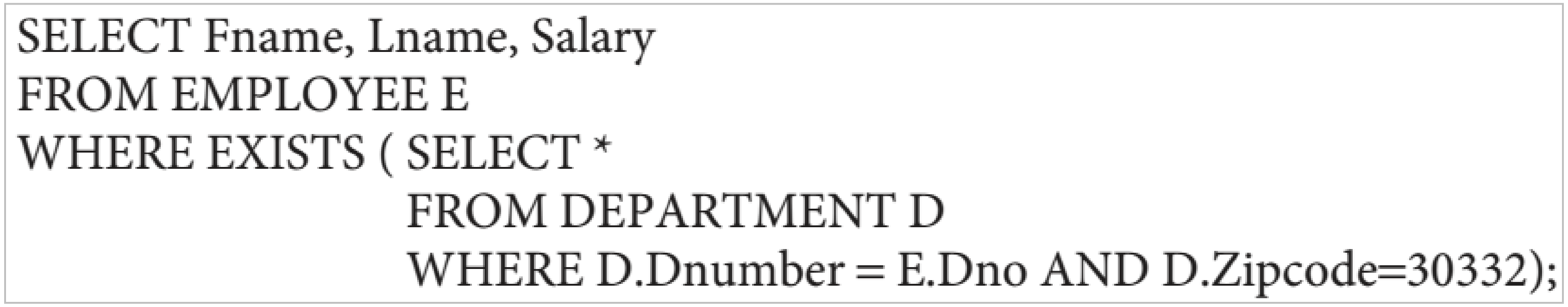


```
SELECT E1.Fname, E1.Lname
FROM EMLOYEE E1
WHERE E1.Salary = ( SELECT MAX (Salary)
                    FROM EMPLOYEE E2)
```

```
M ← (SELECT MAX(Salary) FROM EMPLOYEE E2);


SELECT E1.Fname, E1.Lname
FROM EMPLOYEE E1
WHERE E1.Salary = M;
```

3.8. Using Heuristics in Query Optimization

How about nested subquery optimization?



```
SELECT Fname, Lname, Salary
FROM EMPLOYEE E
WHERE EXISTS ( SELECT *
               FROM DEPARTMENT D
               WHERE D.Dnumber = E.Dno AND D.Zipcode=30332);
```

```
SELECT Fname, Lname, Salary
FROM EMPLOYEE E, DEPARTMENT D
WHERE E.Dno = D.Dnumber AND D.Zipcode = 30332;
```

3.8. Using Heuristics in Query Optimization

How about nested subquery optimization?

```
SELECT COUNT(*)
FROM DEPARTMENT D
WHERE D.Dnumber IN ( SELECT E.Dno
                     FROM EMPLOYEE E
                     WHERE E.Salary > 200000)
```

```
ET (Dno) ← (SELECT DISTINCT E.Dno
            FROM EMPLOYEE E
            WHERE E.Salary <200000);

SELECT COUNT(*)
FROM DEPARTMENT D, ET E
WHERE D.Dnumber = E.Dno;
```

```
SELECT COUNT(*)
FROM EMPLOYEE E, DEPARTMENT D
WHERE D.Dnumber S= E.Dno
            AND E.Salary > 200000;
```

NOTE: *S=* is a semi-join operation to take the first match if any.

3.8. Using Heuristics in Query Optimization

Query Execution Plans

- An execution plan for a relational algebra query consists of a combination of the relational algebra query tree and information about the access methods to be used for each relation as well as the methods to be used in computing the relational operators stored in the tree.
- **Materialized evaluation:** The result of an operation is stored as a temporary relation.
- **Pipelined evaluation:** as the result of an operator is produced, it is forwarded to the next operator in sequence.

3.9. Using Selectivity and Cost Estimates in Query Optimization

- **Cost-based query optimization:** Estimate and compare the costs of executing a query using different execution strategies and choose the strategy with the lowest cost estimate.
- **Issues**
  - **Cost function**
    - Estimation
  - **Number of execution strategies to be considered**

3.9. Using Selectivity and Cost Estimates in Query Optimization

- **Cost-based query optimization:**
  - For a given subexpression in the query, there may be multiple equivalence rules that apply.
  - It is necessary to resort to some quantitative measure for evaluation of alternatives. By using the space and time requirements and reducing them to some common metric called cost, it is possible to devise some methodology for optimization.
  - Appropriate search strategies can be designed by keeping the cheapest alternatives and pruning the costlier alternatives.
  - The scope of query optimization is generally a query block. Various table and index access paths, join permutations (orders), join methods, group-by methods, etc. provide the alternatives from which the query optimizer must chose.
  - In a global query optimization, the scope of optimization is multiple query blocks.

3.9. Using Selectivity and Cost Estimates in Query Optimization

Cost Components for Query Execution

- Access cost to secondary storage
- Storage cost
- Computation cost
- Memory usage cost
- Communication cost

→ Different database systems (large, small, distributed, …) may focus on different cost components.

→ In the following, cost functions consider a single factor – disk access.

3.9. Using Selectivity and Cost Estimates in Query Optimization

Catalog Information Used in Cost Functions

- Information about the size of a file
  - **number of records (tuples) (r)**
  - **record size (R)**
  - **number of blocks (b)**
  - **blocking factor (bfr)**
- Information about indexes and indexing attributes of a file
  - **Number of levels (x)** of each multilevel index
  - **Number of first-level index blocks (b_{I1})**
  - **Number of distinct values (d)** of an attribute
  - **Selectivity (sl)** of an attribute (the fraction of records satisfying an equality condition on the attribute)
  - **Selection cardinality (s)** of an attribute (s = sl * r)

3.9. Using Selectivity and Cost Estimates in Query Optimization

Cost Functions for SELECT:

- **S1. Linear search (brute force) approach**

  To retrieve all records satisfying the selection condition:

  $C_{S1a} = b$

  To retrieve a single record for an equality condition on a key:

  if the record is found, $C_{S1b} = \lceil b/2 \rceil$

  otherwise, $C_{S1a} = b$

- **S2. Binary search**

  $C_{S2} = \lceil \log_2 b \rceil + \lceil s/bfr \rceil - 1$

  For an equality condition on a unique (key) attribute (s=1),

  $C_{S2} = \lceil \log_2 b \rceil$

- **S3. Using a primary index (S3a) or hash key (S3b) to retrieve a single record**

  $C_{S3a} = x + 1$

  $C_{S3b} = 1$ for static or linear hashing

  $C_{S3b} = 2$ for extendible hashing

3.9. Using Selectivity and Cost Estimates in Query Optimization

Cost Functions for SELECT:

- **S4. Using an ordering index to retrieve multiple records**

  For the comparison condition (>, >=, <, or <=) on a key field with an ordering index

  $C_{S4} = x + \lceil b/2 \rceil$

- **S5. Using a clustering index to retrieve multiple records for an equality condition**

  $C_{S5} = x + \lceil s/bfr \rceil$

- **S6. Using a secondary (B^+-tree) index**

  For an equality comparison, $C_{S6a} = x + 1 + s$

  (The additional 1 is to account for the disk block that contains the record pointers after the index is searched.)

  For a comparison condition (>, <, >=, or <=),

  $C_{S6b} = x + \lceil b_{I1}/2 \rceil + \lceil r/2 \rceil$

  (Half the file records are assumed to satisfy the condition.)

3.9. Using Selectivity and Cost Estimates in Query Optimization

Cost Functions for SELECT:

- **S7. Conjunctive selection**

  Use either S1 or one of the methods S2 to S6 to solve.

  For the latter case, use one condition to retrieve the records and then check in the memory buffer whether each retrieved record satisfies the remaining conditions in the conjunction.

- **S8. Conjunctive selection using a composite index**

  Same as S3a, S5 or S6a, depending on the type of index.

- **S9. Selection using a bitmap index**

  A number of bit vectors may fit in one block. Then, if *s* records qualify, *s* blocks are accessed for the data records.

- **S10. Selection using a functional index**

  This works similar to S6 except that the index is based on a function of multiple attributes.

Example

- EMPLOYEE: r_E = 10,000 , b_E = 2000 , bfr_E = 5
- Access paths:
  - 1. A clustering index on SALARY, with levels x_{SALARY} = 3 and average selection cardinality S_{SALARY} = 20.
  - 2. A secondary index on the key attribute SSN, with x_{SSN} = 4 (S_{SSN} = 1).
  - 3. A secondary index on the nonkey attribute DNO, with x_{DNO}= 2 and first-level index blocks b_{I1DNO}= 4.

    There are d_{DNO} = 125 distinct values for DNO, so the selection cardinality of DNO is $S_{DNO} = \lceil r_E/d_{DNO} \rceil$ = 80.
  - 4. A secondary index on SEX, with x_{SEX} = 1.

    There are d_{SEX} = 2 values for the sex attribute, so the average selection cardinality is $S_{SEX} = \lceil r_E/d_{SEX} \rceil$ = 5000.

Example

- OP1: $\sigma_{SSN='123456789'}$ (EMPLOYEE)
  - $C_{S1b} = 1000$
  - $C_{S6a} = x_{SSN} + 1 = 4 + 1 = 5$
- OP2: $\sigma_{DNO>5}$ (EMPLOYEE)
  - $C_{S1a} = 2000$
  - $C_{S6b} = x_{DNO} + \lceil b_{I1DNO}/2 \rceil + \lceil r_E/2 \rceil = 2 + \lceil 4/2 \rceil + \lceil 10000/2 \rceil = 5004$

Example

- OP3: $\sigma_{DNO=5}$ (EMPLOYEE)
  - $C_{S1a} = 2000$
  - $C_{S6a} = x_{DNO} + s_{DNO} + 1 = 2 + 80 + 1 = 83$
- OP4: $\sigma_{DNO=5 \text{ AND } SALARY>30000 \text{ AND } SEX='F'}$ (EMPLOYEE)
  - $C_{S6a-DNO} = 83$
  - $C_{S4-SALARY} = x_{SALARY} + \lceil b/2 \rceil = 3 + \lceil 2000/2 \rceil = 1003$
  - $C_{S6a-SEX} = x_{SEX} + s_{SEX} + 1 = 1 + 5000 + 1 = 5002$

=> chose DNO=5 first and check the other conditions

Example

- OP4': $\sigma_{\text{Dno=5 OR Salary > 30000 OR Sex ='F'}}$ (EMPLOYEE)
  - $C_{S1a} = 2000$

  Using a bitmap index

  - $C_{S9} = s = s_{DNO} + s_{Salary} + s_{Sex} = 80 + 20 + 5000 = 5100$

  Using single access paths, a total cost with a disjunctive condition stems from the sum of all paths.

  - $C_{S6a\text{-}DNO} = x_{DNO} + s_{DNO} + 1 = 2 + 80 + 1 = 83$
  - $C_{S4\text{-}SALARY} = x_{SALARY} + \lceil b/2 \rceil = 3 + \lceil 2000/2 \rceil = 1003$
  - $C_{S6a\text{-}SEX} = x_{SEX} + s_{SEX} + 1 = 1 + 5000 + 1 = 5002$

  => Choose linear search with C_{S1a}

Example

- OP5: $\sigma_{ESSN='123456789' \text{ AND } PNO=10}(WORKS\_ON)$
  - $C_{S1a} = b_{WORKS\_ON}$

  Using a primary composite index on (ESSN, PNO)
  - $C_{S8} = C_{S3a} = x + 1$

  => choose the access path with a smaller cost: linear search or access via a primary composite index on (ESSN, PNO)

Example

- OP6: $\sigma_{\text{DNO IN (3, 27, 49)}}$(EMPLOYEE)
  - $C_{S1a} = 2000$

  Using a bitmap index
  - $C_{S9} = s = s_{DNO} + s_{DNO} + s_{DNO} = 80 + 80 + 80 = 240$

  Using a secondary index
  - $C_{S6a} = 3*(x_{DNO} + s_{DNO} + 1) = 3*(2 + 80 + 1) = 249$

  => Choose access via a bitmap index with C_{S9}
- OP7: $\sigma_{\text{((Salary*Commission\_pct) + Salary) > 5000}}$(EMPLOYEE)
  - $C_{S1a} = 2000$

  Using a functional index
  - $C_{S6b} = x + \lceil b_{I1}/2 \rceil + \lceil r/2 \rceil > 5000$

  => Choose linear search with CS1a

How about if > is changed to =?

3.9. Using Selectivity and Cost Estimates in Query Optimization

Cost Functions for JOIN:

- **Join selectivity (js)**

  $js = | (R \bowtie_C S) | / | R \times S | = | (R \bowtie_C S) | / (|R| * |S|)$

  If condition C does not exist, js = 1.

  If no tuples from the relations satisfy condition C, js = 0.

  Usually, **0 <= js <= 1**

  Size of the result file after join operation

  $| (R \bowtie_C S) | = js * |R| * |S|$

3.9. Using Selectivity and Cost Estimates in Query Optimization

Cost Functions for JOIN:

- **J1. Nested-loop join**

  $C_{J1} = b_R + b_R * b_S + \lceil (js * |R| * |S|)/bfr_{RS} \rceil$

  (Use *R* for outer loop)

- **J2. Single-loop join**: using an access structure to retrieve the matching record(s)

  If an index exists for the join attribute *B* of *S* with index levels

  x_B, we can retrieve each record *s* in *R* and then use the index to retrieve all the matching records *t* from *S* that satisfy t[B] = s[A].

  The cost depends on the type of index.

3.9. Using Selectivity and Cost Estimates in Query Optimization

Cost Functions for JOIN:

- **J2. Single-loop join**

  For a secondary index,

  $C_{J2a} = b_R + |R| * (x_B + s_B + 1) + \lceil (js * |R| * |S|)/bfr_{RS} \rceil$

  For a clustering index,

  $C_{J2b} = b_R + |R| * (x_B + \lceil s_B/bfr_B \rceil) + \lceil (js * |R| * |S|)/bfr_{RS} \rceil$

  For a primary index,

  $C_{J2c} = b_R + |R| * (x_B + 1) + \lceil (js * |R| * |S|)/bfr_{RS} \rceil$

  If a hash key exists for one of the two join attributes — B of S

  $C_{J2d} = b_R + |R| * h + \lceil (js * |R| * |S|)/bfr_{RS} \rceil$

  h: the average number of block accesses to retrieve a record, given its hash key value, h>=1

3.9. Using Selectivity and Cost Estimates in Query Optimization

Cost Functions for JOIN:

- **J3. Sort-merge join**

  $C_{J3a} = C_S + b_R + b_S + \lceil (js * |R| * |S|)/bfr_{RS} \rceil$

  (C_S: Cost for sorting files. If both files are sorted, $C_S = 0$.)

- **J4. Hash join**

  The records of files *R* and *S* are partitioned into smaller files. The partitioning of each file is done using the same hashing function *h* on the join attribute *A* of *R* (for partitioning file R) and *B* of *S* (for partitioning file S).

  For the larger files R and S that can not be hashed entirely in the memory: $C_{J4} = 3*(b_R + b_S) + \lceil (js * |R| * |S|)/bfr_{RS} \rceil$

  For a smaller file (either R or S) that can be hashed entirely in the memory: $C_{J4} = b_R + b_S + \lceil (js * |R| * |S|)/bfr_{RS} \rceil$

Example

- Suppose that we have the EMPLOYEE file described in the previous example
- The DEPARTMENT file:

  r_D = 125 and b_D = 13 ,

  $x_{DNUMBER}$ = 1,

  secondary index on MGRSSN of DEPARTMENT,

  s_{MGRSSN} = 1, x_{MGRSSN} = 2,

  js_{OP8} = (1/IDEPARTMENTI) = 1/125 , bfr_{ED} = 4

- OP8: EMPLOYEE $\bowtie_{DNO=DNUMBER}$ DEPARTMENT
- OP9: DEPARTMENT $\bowtie_{MGRSSN=SSN}$ EMPLOYEE

Example

- OP8: EMPLOYEE $\bowtie_{\text{DNO=DNUMBER}}$ DEPARTMENT
  - Method J1 (Nested loop) with Employee as outer:
    - $C_{J1} = b_E + b_E * b_D + \lceil (js_{OP6} * r_E * r_D)/bfr_{ED} \rceil$
      $= 2000 + 2000 * 13 + \lceil ((1/125) * 10{,}000 * 125)/4 \rceil = 30{,}500$
  - Method J1 (Nested loop) with Department as outer:
    - $C_{J1} = b_D + b_E * b_D + \lceil (js_{OP6} * r_E * r_D)/bfr_{ED} \rceil$
      $= 13 + 13 * 2000 + \lceil ((1/125) * 10{,}000 * 125)/4 \rceil = 28{,}513$
  - Method J2 (Single loop) with EMPLOYEE as outer loop:
    - $C_{J2c} = b_E + r_E * (x_{DNUMBER} + 1) + \lceil (js_{OP6} * r_E * r_D)/bfr_{ED} \rceil$
      $= 2000 + 10{,}000 * 2 + \lceil ((1/125) * 10{,}000 * 125)/4 \rceil = 24{,}500$
  - Method J2 (Single loop) with DEPARTMENT as outer loop:
    - $C_{J2a} = b_D + r_D * (x_{DNO} + s_{DNO} + 1) + \lceil (js_{OP6} * r_E * r_D)/bfr_{ED} \rceil$
      $= 13 + 125 * (2 + 80 + 1) + \lceil ((1/125) * 10{,}000 * 125)/4 \rceil = 12{,}888$
  - Method J4 (Hash join) with DEPARTMENT as a hashed file:
    - $C_{J4} = 3*(b_D + b_E) + \lceil (js_{OP6} * r_E * r_D)/bfr_{ED} \rceil$
      $= 3*(13+2000) + \lceil ((1/125) * 10{,}000 * 125)/4 \rceil = 8{,}539$

Chosen!!!

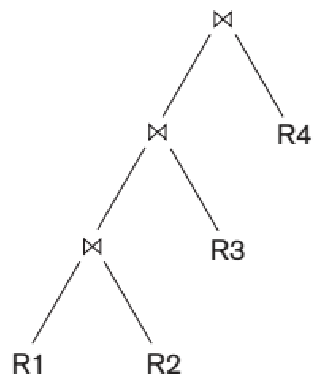
Assume that the DEPARTMENT file of $r_D = 125$ and $b_D = 13$, $x_{DNUMBER} = 1$, secondary index on MGRSSN of DEPARTMENT, $s_{MGRSSN} = 1$, $x_{MGRSSN} = 2$, $js_{OP9} = (1/|EMPLOYEE|) = 1/r_E = 1/10{,}000$, $bfr_{ED} = 4$
A secondary index on the key attribute SSN of EMPLOYEE, with $x_{SSN} = 4$ ($S_{SSN} = 1$).

- OP9: DEPARTMENT $\bowtie_{MGRSSN=SSN}$ EMPLOYEE
  - Method J1 (Nested loop) with Employee as outer:
    - $C_{J1} = b_E + b_E * b_D + \lceil (js_{OP7} * r_E * r_D)/bfr_{ED} \rceil$
      $= 2000 + 2000 * 13 + \lceil ((1/10{,}000) * 10{,}000 * 125)/4 \rceil$
      $= \lceil 28{,}031.25 \rceil = 28{,}032$
  - Method J1 (Nested loop) with Department as outer:
    - $C_{J1} = b_D + b_E * b_D + \lceil (js_{OP7} * r_E * r_D)/bfr_{ED} \rceil$
      $= 13 + 13 * 2000 + \lceil ((1/10{,}000) * 10{,}000 * 125)/4 \rceil$
      $= \lceil 26{,}044.25 \rceil = 26{,}045$
  - Method J2 (Single loop) with EMPLOYEE as outer loop:
    - $C_{J2c} = b_E + r_E * (x_{MGRSSN} + s_{MGRSSN} + 1) + \lceil (js_{OP7} * r_E * r_D)/bfr_{ED} \rceil$
      $= 2000 + 10{,}000 * (2 + 1 + 1) + \lceil ((1/10{,}000) * 10{,}000 * 125)/4 \rceil$
      $= \lceil 42{,}031.25 \rceil = 42{,}032$
  - Method J2 (Single loop) with DEPARTMENT as outer loop:
    - $C_{J2a} = b_D + r_D * (x_{SSN} + s_{SSN} + 1) + \lceil (js_{OP7} * r_E * r_D)/bfr_{ED} \rceil$
      $= 13 + 125 * (4 + 1 + 1) + \lceil ((1/10{,}000) * 10{,}000 * 125)/4 \rceil$
      $= \lceil 794.25 \rceil = 795$ **Chosen!!!**

3.9. Using Selectivity and Cost Estimates in Query Optimization

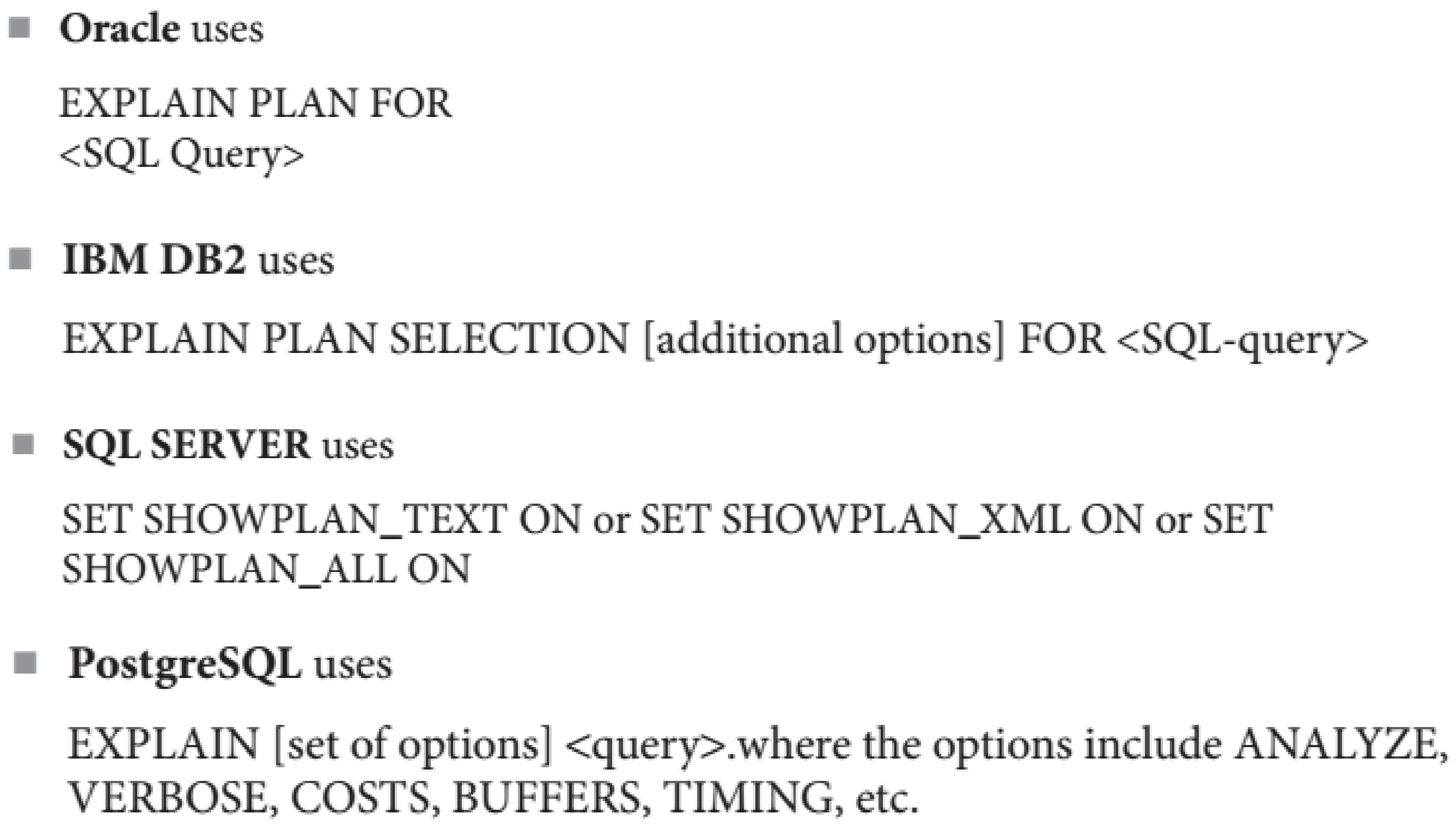
Multiple Relation Queries and Join Ordering

- A query joining n relations will have n-1 join operations, and hence can have a large number of different join orders when we apply the algebraic transformation rules.
- Current query optimizers typically limit the structure of a (join) query tree to that of left-deep (or right-deep) trees.
- **Left-deep tree** : a binary tree where the right child of each non-leaf node is always a base relation.
  - Amenable to pipelining
  - Utilize any access paths on the base relation (the right child) when executing the join



3.10. Overview of Query Optimization in DBMSs

- Displaying the System's Query Execution Plan



- **Oracle** uses

  EXPLAIN PLAN FOR
  <SQL Query>

- **IBM DB2** uses

  EXPLAIN PLAN SELECTION [additional options] FOR <SQL-query>

- **SQL SERVER** uses

  SET SHOWPLAN_TEXT ON or SET SHOWPLAN_XML ON or SET SHOWPLAN_ALL ON

- **PostgreSQL** uses

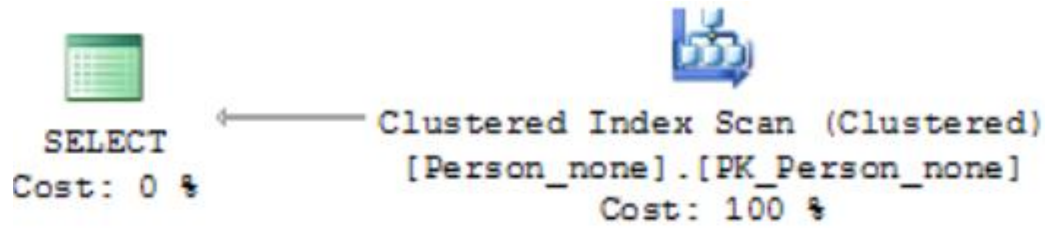
  EXPLAIN [set of options] <query>.where the options include ANALYZE, VERBOSE, COSTS, BUFFERS, TIMING, etc.

Execution Plan in Oracle

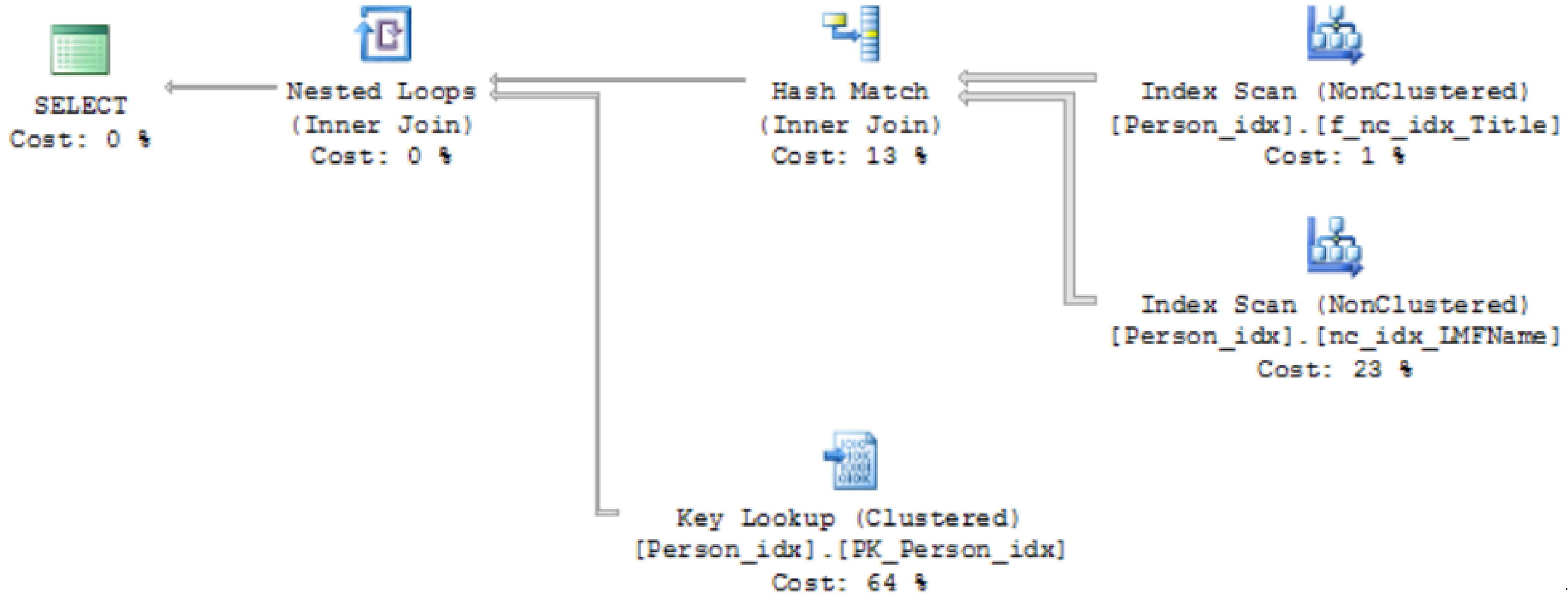
```
SELECT e.last_name, j.job_title, d.department_name
FROM   hr.employees e, hr.departments d, hr.jobs j
WHERE  e.department_id = d.department_id
AND    e.job_id = j.job_id
AND    e.last_name LIKE 'A%' ;

Execution Plan
----------------------------------------------------------
Plan hash value: 975837011


---------------------------------------------------------------------------------------------
| Id  | Operation                     | Name          | Rows  | Bytes | Cost  (%CPU)| Time     |
---------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |               |     3 |   189 |      7  (15)| 00:00:01 |
|*  1 |  HASH JOIN                    |               |     3 |   189 |      7  (15)| 00:00:01 |
|*  2 |   HASH JOIN                   |               |     3 |   141 |      5  (20)| 00:00:01 |
|   3 |    TABLE ACCESS BY INDEX ROWID| EMPLOYEES     |     3 |    60 |      2   (0)| 00:00:01 |
|*  4 |     INDEX RANGE SCAN          | EMP_NAME_IX   |     3 |       |      1   (0)| 00:00:01 |
|   5 |    TABLE ACCESS FULL          | JOBS          |    19 |   513 |      2   (0)| 00:00:01 |
|   6 |   TABLE ACCESS FULL           | DEPARTMENTS   |    27 |   432 |      2   (0)| 00:00:01 |
---------------------------------------------------------------------------------------------


Predicate Information (identified by operation id):
---------------------------------------------------


   1 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
   2 - access("E"."JOB_ID"="J"."JOB_ID")
   4 - access("E"."LAST_NAME" LIKE 'A%')
       filter("E"."LAST_NAME" LIKE 'A%')
```

Execution Plan in MS SQL Server

```sql
select *
from dbo.Person_none
where FirstName like 'C%' and Title is not NULL;
```


SELECT
Cost: 0 %
Clustered Index Scan (Clustered)
[Person_none].[PK_Person_none]
Cost: 100 %

```sql
select *
from dbo.Person_idx
where FirstName like 'C%' and Title is not NULL;
```

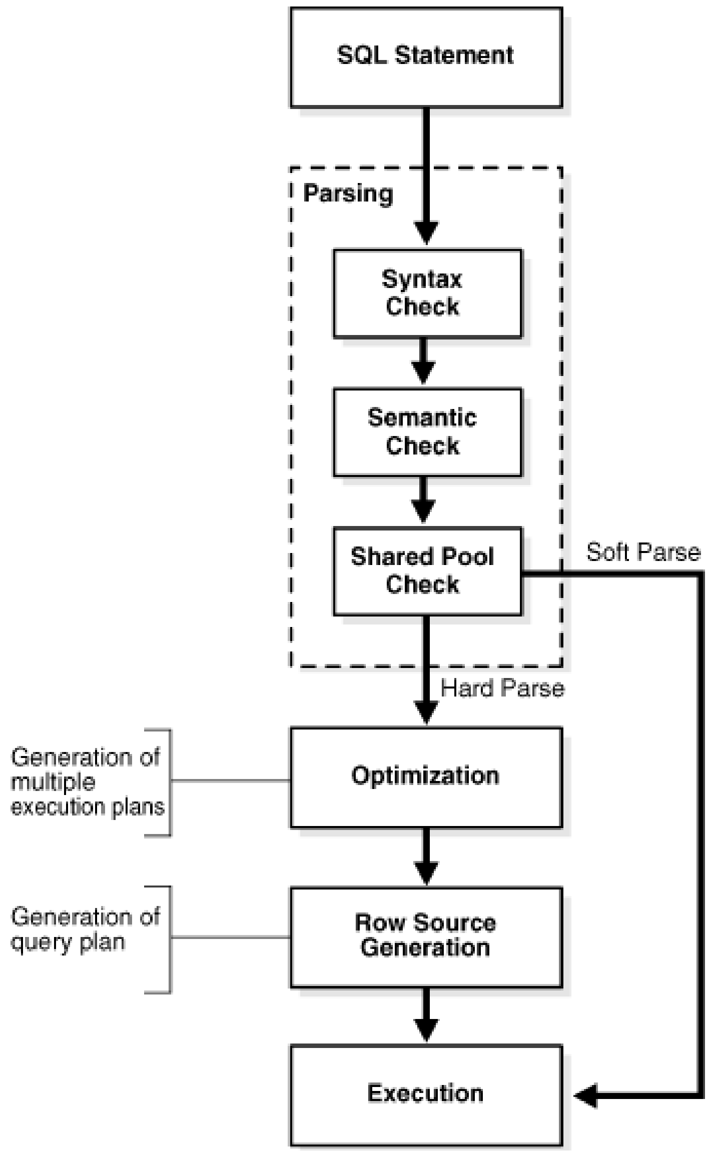

SELECT
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 0 %
Hash Match
(Inner Join)
Cost: 13 %
Index Scan (NonClustered)
[Person_idx].[f_nc_idx_Title]
Cost: 1 %
Index Scan (NonClustered)
[Person_idx].[nc_idx_LMFName]
Cost: 23 %
Key Lookup (Clustered)
[Person_idx].[PK_Person_idx]
Cost: 64 %

3.10. Overview of Query Optimization in Oracle

- Stages of query processing in Oracle
  - Parse tree ~ Query tree
  - Cost-based optimization
  - User's interaction
    - Hints

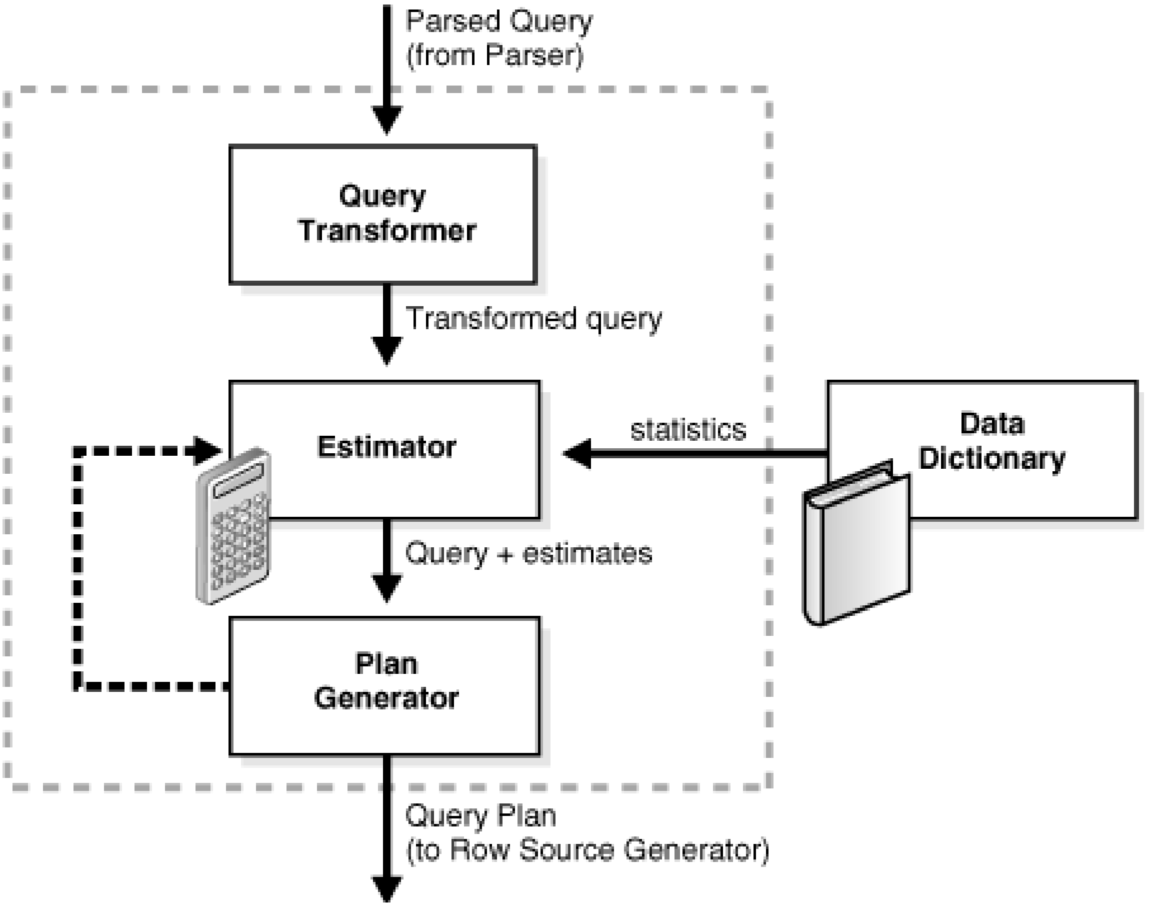
Oracle® Database Concepts 11*g* Release 2 (11.2)
Part Number E25789-01, 7 SQL

Oracle® Database Concepts, 18c
E84295-04, 7 SQL



3.10. Overview of Query Optimization in Oracle

- Query optimizer
  - determines which execution plan is most efficient by considering several sources of information, including query conditions, available access paths, statistics gathered for the system, and hints



Oracle® Database Concepts 11*g* Release 2 (11.2)
Part Number E25789-01, 7 SQL

3.10. Overview of Query Optimization in Oracle

- Optimizer hints
  - A hint is a comment in a SQL statement that acts as an instruction to the optimizer.
    - Create the capability for an application developer to specify hints (also called query *annotations* or *directives* in other systems) to the optimizer
  - Hints are embedded in the text of a statement.
  - Hints are commonly used to address the infrequent cases where the optimizer chooses a suboptimal plan.
  - Sometimes the application designer, who has more information about a particular application's data than is available to the optimizer, can choose a more effective way.

3.10. Overview of Query Optimization in Oracle

- Optimizer hints /*+ FIRST_ROWS(25) */
  - Suppose that an interactive application runs a query that returns 50 rows. User A wants the optimizer to generate a plan that gets the first 25 records as quickly as possible so that the user is not forced to wait.

```
SELECT /*+ FIRST_ROWS(25) */ employee_id, department_id
FROM   hr.employees
WHERE  department_id > 50;

------------------------------------------------------------------------
| Id | Operation                    | Name              | Rows | Bytes
------------------------------------------------------------------------
|  0 | SELECT STATEMENT             |                   | 26   | 182
|  1 |  TABLE ACCESS BY INDEX ROWID | EMPLOYEES         | 26   | 182
|* 2 |   INDEX RANGE SCAN           | EMP_DEPARTMENT_IX |      |
------------------------------------------------------------------------
```

No hint

```
SELECT employee_id, department_id
FROM   hr.employees
WHERE  department_id > 50;

------------------------------------------------------------------------
| Id | Operation             | Name                 | Rows | Bytes | Cos
------------------------------------------------------------------------
|  0 | SELECT STATEMENT      |                      | 50   | 350   |
|* 1 |  VIEW                 | index$_join$_001     | 50   | 350   |
|* 2 |   HASH JOIN           |                      |      |       |
|* 3 |    INDEX RANGE SCAN   | EMP_DEPARTMENT_IX    | 50   | 350   |
|  4 |    INDEX FAST FULL SCAN| EMP_EMP_ID_PK       | 50   | 350   |
```

User A want the optimizer to generate a plan that gets the first 25 records as quickly as possible.

hint

```
SELECT /*+ FIRST_ROWS(25) */ employee_id, department_id
FROM   hr.employees
WHERE  department_id > 50;

------------------------------------------------------------------------
| Id | Operation                    | Name              | Rows | Bytes
------------------------------------------------------------------------
|  0 | SELECT STATEMENT             |                   | 26   | 182
|  1 |  TABLE ACCESS BY INDEX ROWID | EMPLOYEES         | 26   | 182
|* 2 |   INDEX RANGE SCAN           | EMP_DEPARTMENT_IX |      |
------------------------------------------------------------------------
```

3.11. Semantic Query Optimization

- A different approach to query optimization, used in combination with the techniques discussed previously, uses constraints specified on the database schema - such as unique attributes and other more complex constraints—to modify one query into another query that is more efficient to execute.
- With the inclusion of active rules and additional metadata in database systems, semantic query optimization techniques are being gradually incorporated into DBMSs.

3.11. Semantic Query Optimization

```
SELECT   E.Lname, M.Lname
FROM     EMPLOYEE AS E, EMPLOYEE AS M
WHERE    E.Super_ssn=M.Ssn AND E.Salary > M.Salary
```

A constraint on the database schema that stated that no employee can earn more than his or her direct supervisor
=> The result of the query will be empty. => No processing

```
SELECT Lname, Salary
FROM EMPLOYEE, DEPARTMENT
WHERE EMPLOYEE.Dno = DEPARTMENT.Dnumber and
EMPLOYEE.Salary>100000
```

the attributes retrieved are only from one relation: EMPLOYEE;
the selection condition is also on that relation.
=> rewritten with the primary-key/foreign-key relationship semantics

```
SELECT Lname, Salary
FROM EMPLOYEE
WHERE EMPLOYEE.Dno IS NOT NULL and EMPLOYEE.Salary>100000
```

Summary

- Query processing includes several typical steps as follows:
  - Scanning, parsing, validating
  - Query optimizing
  - Query code generating
  - Runtime database processing
- Query optimization is a process of choosing a suitable execution plan for processing a query.
  - A *reasonably efficient or the best available plan* for executing the query
  - Heuristic optimizer vs Cost-based optimizer

Summary

- Heuristic optimizer
  - Heuristic rules for reordering the operations in a query tree of an execution plan
  - Rules for reducing the size of each intermediate result are applied first: SELECT, PROJECT
  - The most restrictive SELECT and JOIN operations are executed first: CONDITIONS for FEWER RECORDS.
  - Avoid the CARTESIAN PRODUCT operation

Summary

- Cost-based optimizer
  - Estimating cost for different execution plans and choosing the plan that minimizes estimated cost
  - Different database systems consider different cost components for a cost function.
  - The scope of query optimization is generally a query block. Various table and index access paths, join permutations (orders), join methods, group-by methods, and so on provide the alternatives from which the query optimizer must choose.
  - Catalog information used in cost functions
    - Selectivity, cardinality, …, other statistical information

Your turn for query optimization

Given the three following relations:

Supplier(Supp#, Name, City, Specialty)

Project(Proj#, Name, City, Budget)

Order(Supp#, Proj#, Part-name, Quantity, Cost)

and a SQL query:

SELECT Supplier.Name, Project.Name

FROM Supplier, Order, Project

WHERE Supplier.City = ‘New York City’ **AND** Project.Budget > 10000000

AND Supplier.Supp# = Order.Supp# **AND** Order.Proj# = Project.Proj#;

1. Write the relational algebraic expression that is equivalent to the above query and draw a query tree for the expression.

2. Apply the heuristic optimization transformation rules to find an efficient query execution plan for the above query. Assume that the number of the suppliers in New York is larger that the number of the projects with the budgets more than 10000000$.

Your turn for query optimization

3. Suppose that:

Supplier has r_S = 500 records, b_S = 100 blocks, bfr_S = 5 records/block, one primary index B+-tree on Supp# with $x_{Supp\#}$=2 and one secondary index B+-tree on City with x_{City} = 2, d_{City} = 50 distinct values for City.

Project has r_P = 1,000 records, b_P = 200 blocks, bfr_P = 5 records/block, one primary index B+-tree on Proj# with $x_{Proj\#}$ = 2 and another secondary index B+-tree on Budget with x_{Budget}=2 and first-level index blocks $b_{I1Budget}$ = 5.

Order has r_O = 20,000 records, b_O = 5,000 blocks, bfr_O = 4 records/block, a secondary index B+-tree on Supp# with $x_{Supp\#}$ = 3 and another secondary index B+-tree on Proj# with $x_{Proj\#}$ = 3 and first-level index blocks $b_{I1Proj\#}$ = 10.

Blocking factor for join results bfr_{PO} = 2, bfr_{SO} = 2.

What access paths should be for $\sigma_{Budget>10000000}$(Project),

$\sigma_{City=\text{'New York City'}}$(Supplier), and for Project $\bowtie_{Project.Proj\#=\ Order.Proj\#}$ Order?

Chapter 3: Algorithms for Query Processing and Optimization

Check for Understandings

- 3.1. List and describe typical steps when a query is processed.
- 3.2. Differentiate a query tree from a query graph.
- 3.3. Why does a SQL query need to be translated into relational algebra expressions?
- 3.4. Describe external sorting and calculate its cost. List some applications of sorting in query processing.

Check for Understandings

- 3.5. How are SELECT operations implemented? Give an example.
- 3.6. How are JOIN operations implemented? Give an example.
- 3.7. How are PROJECT operations implemented? Give an example.
- 3.8. How are aggregate operations implemented? Give an example.
- 3.9. How are SET operations implemented?
- 3.10. What is pipelining? Give an example.

Check for Understandings

- 3.11. Given queries as follows, for each query, write its corresponding SQL statement, draw its query tree, and then explain its processing to obtain the result.

  3.11.1. Retrieve the last name and salary of each employee who works in department 10 and has a salary higher than 30,000.

  3.11.2. Retrieve the last name and department number of each employee who works in the department where the minimum salary of the employees is higher than 30,000.

  3.11.3. Retrieve the department name and department number of each department where more than 10 employees work with a salary higher than 30,000.

Check for Understandings

- 3.12. What is an execution plan? Give an example of a query and its execution plan.
- 3.13. What is a heuristic optimizer? What are its heuristic rules?
- 3.14. What is a cost-based optimizer? How is it different from a heuristic optimizer?
- 3.15. Describe cost components for a cost function to estimate a query execution cost. What kind of databases uses each cost component?
- 3.16. Differentiate pipelining from materialization. Demonstrate their differences.

Check for Understandings

- 3.17. Draw query trees step by step to obtain a final optimized query tree using heuristic optimization for each query in 3.11.
- 3.18. Using the characteristics of the EMPLOYEE and DEPARTMENT data files as described in the previous slides, describe an optimized execution plan based on a decision of the cost-based optimizer for each query in 3.11.