

# **CHAPTER 18**

## **Strategies for Query Processing**

# Introduction

- DBMS techniques to process a query
  - Scanner identifies query tokens
  - Parser checks the query syntax
  - Validation checks all attribute and relation names
  - Query tree (or query graph) created
  - Execution strategy or query plan devised
- Query optimization
  - Planning a good execution strategy

# Query Processing

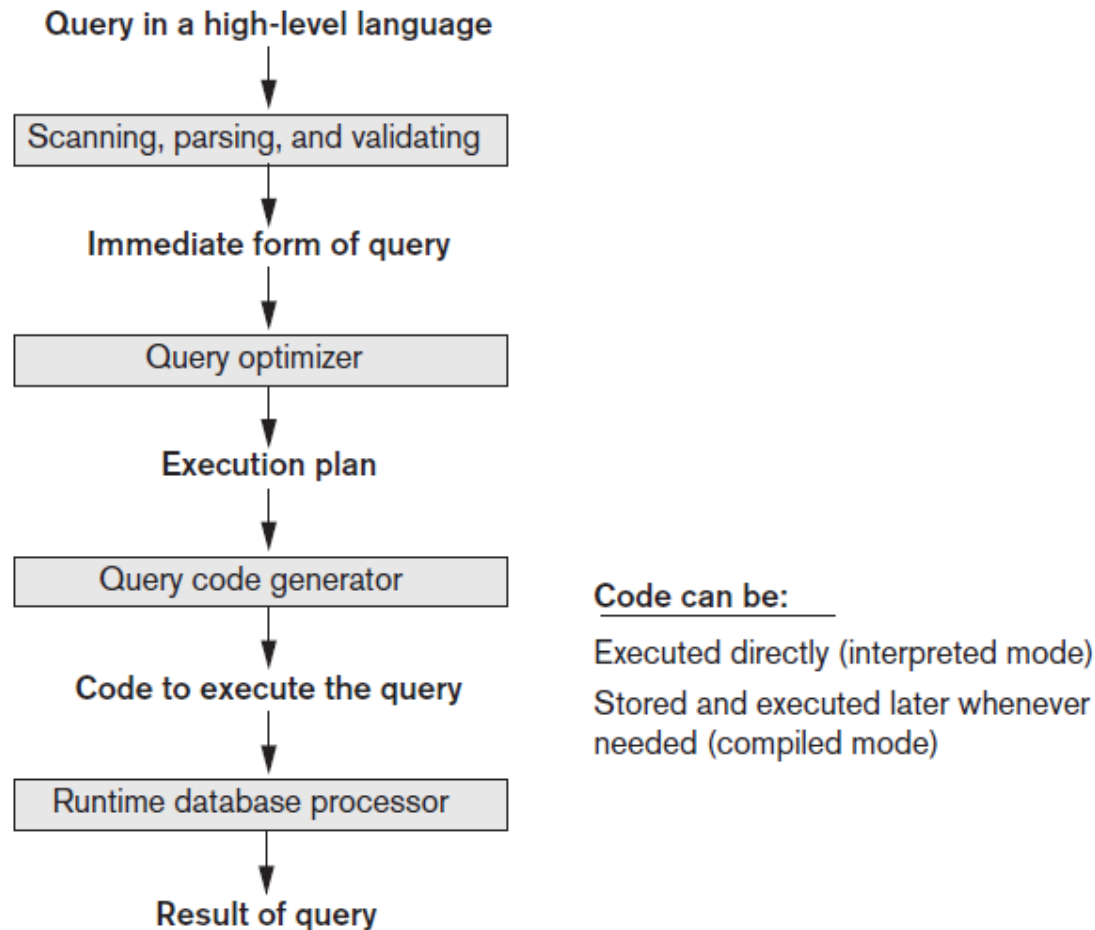


Figure 18.1 Typical steps when processing a high-level query

# 18.1 Translating SQL Queries into Relational Algebra and Other Operators

- SQL
  - Query language used in most RDBMSs
- Query decomposed into query blocks
  - Basic units that can be translated into the algebraic operators
  - Contains single SELECT-FROM-WHERE expression
    - May contain GROUP BY and HAVING clauses

# Translating SQL Queries (cont'd.)

- Example:

```
SELECT Lname, Fname
FROM EMPLOYEE
WHERE Salary > ( SELECT MAX (Salary)
                  FROM EMPLOYEE
                  WHERE Dno=5 );
```

- Inner block

```
( SELECT MAX (Salary)
  FROM EMPLOYEE
  WHERE Dno=5 )
```

- Outer block

```
SELECT Lname, Fname
FROM EMPLOYEE
WHERE Salary > c
```

# Translating SQL Queries (cont'd.)

- Example (cont'd.)

- Inner block translated into:

$\mathcal{S}_{\text{MAX Salary}}(\sigma_{\text{Dno}=5}(\text{EMPLOYEE}))$

- Outer block translated into:

$\pi_{\text{Lname, Fname}}(\sigma_{\text{Salary} > c}(\text{EMPLOYEE}))$

- Query optimizer chooses execution plan for each query block

# Additional Operators Semi-Join and Anti-Join

## ■ Semi-join

- Generally used for unnesting EXISTS, IN, and ANY subqueries
- Syntax:  $T1.X \text{ } S = T2.Y$ 
  - T1 is the left table and T2 is the right table of the semi-join
- A row of T1 is returned as soon as T1.X finds a match with any value of T2.Y without searching for further matches



# Additional Operators Semi-Join and Anti-Join (cont'd.)

## ■ Anti-join

- Used for unnesting NOT EXISTS, NOT IN, and ALL subqueries
- Syntax:  $T1.x \neq T2.y$ 
  - T1 is the left table and T2 is the right table of the anti-join
- A row of T1 is rejected as soon as T1.x finds a match with any value of T2.y
- A row of T1 is returned only if T1.x does not match with any value of T2.y

# 18.2 Algorithms for External Sorting

- Sorting is an often-used algorithm in query processing
- External sorting
  - Algorithms suitable for large files that do not fit entirely in main memory
  - Sort-merge strategy based on sorting smaller subfiles (runs) and merging the sorted runs
  - Requires buffer space in main memory
    - DBMS cache

```

set    $i \leftarrow 1$ ;
       $j \leftarrow b$ ;           {size of the file in blocks}
       $k \leftarrow n_B$ ;         {size of buffer in blocks}
       $m \leftarrow \lceil (j/k) \rceil$ ; {number of subfiles- each fits in buffer}

{Sorting Phase}
while ( $i \leq m$ )
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;
     $i \leftarrow i + 1$ ;
}

{Merging Phase: merge subfiles until only 1 remains}
set    $i \leftarrow 1$ ;
       $p \leftarrow \lceil \log_{k-1} m \rceil$  { $p$  is the number of passes for the merging phase}
       $j \leftarrow m$ ;
while ( $i \leq p$ )
do {
     $n \leftarrow 1$ ;
     $q \leftarrow \lceil j/(k-1) \rceil$ ; {number of subfiles to write in this pass}
    while ( $n \leq q$ )
    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;
         $n \leftarrow n + 1$ ;
    }
     $j \leftarrow q$ ;
     $i \leftarrow i + 1$ ;
}

```

Figure 18.2 Outline of the sort-merge algorithm for external sorting

# Algorithms for External Sorting (cont'd.)

- Degree of merging
  - Number of sorted subfiles that can be merged in each merge step
- Performance of the sort-merge algorithm
  - Number of disk block reads and writes before sorting is completed

# 18.3 Algorithms for SELECT Operation

- **SELECT operation**
  - Search operation to locate records in a disk file that satisfy a certain condition
  - File scan or index scan (if search involves an index)
- **Search methods for simple selection**
  - S1: Linear search (brute force algorithm)
  - S2: Binary search
  - S3a: Using a primary index
  - S3b: Using a hash key

# Algorithms for SELECT Operation (cont'd.)

- Search methods for simple selection (cont'd.)
  - S4: Using a primary index to retrieve multiple records
  - S5: Using a clustering index to retrieve multiple records
  - S6: Using a secondary (B+ -tree) index on an equality comparison
  - S7a: Using a bitmap index
  - S7b: Using a functional index

# Algorithms for SELECT Operation (cont'd.)

- Search methods for conjunctive (logical AND) selection
  - Using an individual index
  - Using a composite index
  - Intersection of record pointers
- Disjunctive (logical OR) selection
  - Harder to process and optimize

# Algorithms for SELECT Operation (cont'd.)

- Selectivity
  - Ratio of the number of records (tuples) that satisfy the condition to the total number of records (tuples) in the file
  - Number between zero (no records satisfy condition) and one (all records satisfy condition)
- Query optimizer receives input from system catalog to estimate selectivity



# 18.4 Implementing the JOIN Operation

- JOIN operation
  - One of the most time consuming in query processing
  - EQUIJOIN (NATURAL JOIN)
  - Two-way or multiway joins
- Methods for implementing joins
  - J1: Nested-loop join (nested-block join)
  - J2: Index-based nested-loop join
  - J3: Sort-merge join
  - J4: Partition-hash join

# Implementing the JOIN Operation (cont'd.)

Figure 18.3 Implementing JOIN, PROJECT, UNION, INTERSECTION, and SET DIFFERENCE by using sort-merge, where R has  $n$  tuples and S has  $m$  tuples. (a) Implementing the operation  $T \leftarrow R \bowtie_{A=B} S$ .

```
(a) sort the tuples in R on attribute A;
    sort the tuples in S on attribute B;
    set  $i \leftarrow 1, j \leftarrow 1$ ;
    while ( $i \leq n$ ) and ( $j \leq m$ )
    do { if  $R(i)[A] > S(j)[B]$ 
        then set  $j \leftarrow j + 1$ 
        elseif  $R(i)[A] < S(j)[B]$ 
        then set  $i \leftarrow i + 1$ 
        else { (*  $R(i)[A] = S(j)[B]$ , so we output a matched tuple *)
              output the combined tuple  $\langle R(i), S(j) \rangle$  to  $T$ ;
              (* output other tuples that match  $R(i)$ , if any *)
              set  $l \leftarrow j + 1$ ;
              while ( $l \leq m$ ) and ( $R(i)[A] = S(l)[B]$ )
              do { output the combined tuple  $\langle R(i), S(l) \rangle$  to  $T$ ;
                  set  $l \leftarrow l + 1$ 
              }
              (* output other tuples that match  $S(j)$ , if any *)
              set  $k \leftarrow i + 1$ ;
              while ( $k \leq n$ ) and ( $R(k)[A] = S(j)[B]$ )
              do { output the combined tuple  $\langle R(k), S(j) \rangle$  to  $T$ ;
                  set  $k \leftarrow k + 1$ 
              }
              set  $i \leftarrow k, j \leftarrow l$ 
            }
        }
    }
```

# Implementing the JOIN Operation (cont'd.)

```
(b) create a tuple  $t[\langle \text{attribute list} \rangle]$  in  $T'$  for each tuple  $t$  in  $R$ ;  
    (*  $T'$  contains the projection results before duplicate elimination *)  
    if  $\langle \text{attribute list} \rangle$  includes a key of  $R$   
        then  $T \leftarrow T'$   
    else { sort the tuples in  $T'$ ;  
        set  $i \leftarrow 1, j \leftarrow 2$ ;  
        while  $i \leq n$   
            do { output the tuple  $T'[i]$  to  $T$ ;  
                while  $T'[i] = T'[j]$  and  $j \leq n$  do  $j \leftarrow j + 1$ ;          (* eliminate duplicates *)  
                 $i \leftarrow j; j \leftarrow i + 1$   
            }  
        }  
    }  
    (*  $T$  contains the projection result after duplicate elimination*)
```

Figure 18.3 (cont'd.) Implementing JOIN, PROJECT, UNION, INTERSECTION, and SET DIFFERENCE by using sort-merge, where  $R$  has  $n$  tuples and  $S$  has  $m$  tuples.  
(b) Implementing the operation  $T \leftarrow \pi_{\langle \text{attribute list} \rangle}(R)$ .

# Implementing the JOIN Operation (cont'd.)

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

Figure 18.3 (cont'd.) Implementing JOIN, PROJECT, UNION, INTERSECTION, and SET DIFFERENCE by using sort-merge, where  $R$  has  $n$  tuples and  $S$  has  $m$  tuples.

(c) Implementing the operation  $T \leftarrow R \cup S$ .

# Implementing the JOIN Operation (cont'd.)

```
(d) sort the tuples in  $R$  and  $S$  using the same unique sort attributes;  
    set  $i \leftarrow 1, j \leftarrow 1$ ;  
    while  $(i \leq n)$  and  $(j \leq m)$   
    do { if  $R(i) > S(j)$   
        then set  $j \leftarrow j + 1$   
        elseif  $R(i) < S(j)$   
        then set  $i \leftarrow i + 1$   
        else { output  $R(j)$  to  $T$ ;          (*  $R(i) = S(j)$ , so we output the tuple *)  
             set  $i \leftarrow i + 1, j \leftarrow j + 1$   
        }  
    }
```

Figure 18.3 (cont'd.) Implementing JOIN, PROJECT, UNION, INTERSECTION, and SET DIFFERENCE by using sort-merge, where  $R$  has  $n$  tuples and  $S$  has  $m$  tuples.  
(d) Implementing the operation  $T \leftarrow R \cap S$ .

# Implementing the JOIN Operation (cont'd.)

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

Figure 18.3 (cont'd.) Implementing JOIN, PROJECT, UNION, INTERSECTION, and SET DIFFERENCE by using sort-merge, where  $R$  has  $n$  tuples and  $S$  has  $m$  tuples.  
(e) Implementing the operation  $T \leftarrow R - S$ .

# Implementing the JOIN Operation (cont'd.)

- Available buffer space has important effect on some JOIN algorithms
- Nested-loop approach
  - Read as many blocks as possible at a time into memory from the file whose records are used for the outer loop
  - Advantageous to use the file with fewer blocks as the outer-loop file

# Implementing the JOIN Operation (cont'd.)

## ■ Join selection factor

- Fraction of records in one file that will be joined with records in another file
- Depends on the particular equijoin condition with another file
- Affects join performance

## ■ Partition-hash join

- Each file is partitioned into  $M$  partitions using the same partitioning hash function on the join attributes
- Each pair of corresponding partitions is joined



# Implementing the JOIN Operation (cont'd.)

- Hybrid hash-join
  - Variation of partition hash-join
  - Joining phase for one of the partitions is included in the partition
  - Goal: join as many records during the partitioning phase to save cost of storing records on disk and then rereading during the joining phase

# 18.5 Algorithms for PROJECT and Set Operations

- PROJECT operation
  - After projecting  $R$  on only the columns in the list of attributes, any duplicates are removed by treating the result strictly as a set of tuples
- Default for SQL queries
  - No elimination of duplicates from the query result
    - Duplicates eliminated only if the keyword DISTINCT is included

# Algorithms for PROJECT and Set Operations (cont'd.)

- Set operations
  - UNION
  - INTERSECTION
  - SET DIFFERENCE
  - CARTESIAN PRODUCT
- Set operations sometimes expensive to implement
  - Sort-merge technique
  - Hashing

# Algorithms for PROJECT and Set Operations (cont'd.)

- Use of anti-join for SET DIFFERENCE
  - EXCEPT or MINUS in SQL
  - Example: Find which departments have no employees

Select Dnumber from DEPARTMENT MINUS Select Dno from EMPLOYEE;

becomes

```
SELECT DISTINCT DEPARTMENT.Dnumber
FROM   DEPARTMENT, EMPLOYEE
WHERE  DEPARTMENT.Dnumber < EMPLOYEE.Dno
```

# 18.6 Implementing Aggregate Operations and Different Types of JOINS

- Aggregate operators

- MIN, MAX, COUNT, AVERAGE, SUM
- Can be computed by a table scan or using an appropriate index

- Example:

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

- If an (ascending) B+ -tree index on Salary exists:
  - Optimizer can use the Salary index to search for the largest Salary value
  - Follow the rightmost pointer in each index node from the root to the rightmost leaf

# Implementing Aggregate Operations and Different Types of JOINS (cont'd.)

## ■ AVERAGE or SUM

- Index can be used if it is a dense index
- Computation applied to the values in the index
- Nondense index can be used if actual number of records associated with each index value is stored in each index entry

## ■ COUNT

- Number of values can be computed from the index

# Implementing Aggregate Operations and Different Types of JOINS (cont'd.)

- Standard JOIN (called INNER JOIN in SQL)
- Variations of joins

- Outer join

- Left, right, and full
    - Example:

```
SELECT E.Lname, E.Fname, D.Dname  
FROM   (EMPLOYEE E LEFT OUTER JOIN DEPARTMENT D ON E.Dno = D.Dnumber);
```

- Semi-Join
  - Anti-Join
  - Non-Equi-Join

# 18.7 Combining Operations Using Pipelining

- SQL query translated into relational algebra expression
  - Sequence of relational operations
- Materialized evaluation
  - Creating, storing, and passing temporary results
- General query goal: minimize the number of temporary files
- Pipelining or stream-based processing
  - Combines several operations into one
  - Avoids writing temporary files



# Combining Operations Using Pipelining (cont'd.)

- Pipelined evaluation benefits
  - Avoiding cost and time delay associated with writing intermediate results to disk
  - Being able to start generating results as quickly as possible
- Iterator
  - Operation implemented in such a way that it outputs one tuple at a time
  - Many iterators may be active at one time

# Combining Operations Using Pipelining (cont'd.)

- Iterator interface methods
  - Open()
  - Get\_Next()
  - Close()
- Some physical operators may not lend themselves to the iterator interface concept
  - Pipelining not supported
- Iterator concept can also be applied to access methods

# 18.8 Parallel Algorithms for Query Processing

- Parallel database architecture approaches
  - Shared-memory architecture
    - Multiple processors can access common main memory region
  - Shared-disk architecture
    - Every processor has its own memory
    - Machines have access to all disks
  - Shared-nothing architecture
    - Each processor has own memory and disk storage
    - Most commonly used in parallel database systems

# Parallel Algorithms for Query Processing (cont'd.)

- Linear speed-up
  - Linear reduction in time taken for operations
- Linear scale-up
  - Constant sustained performance by increasing the number of processors and disks

# Parallel Algorithms for Query Processing (cont'd.)

- Operator-level parallelism
  - Horizontal partitioning
    - Round-robin partitioning
    - Range partitioning
    - Hash partitioning
- Sorting
  - If data has been range-partitioned on an attribute:
    - Each partition can be sorted separately in parallel
    - Results concatenated
  - Reduces sorting time

# Parallel Algorithms for Query Processing (cont'd.)

- Selection
  - If condition is an equality condition on an attribute used for range partitioning:
    - Perform selection only on partition to which the value belongs
- Projection without duplicate elimination
  - Perform operation in parallel as data is read
- Duplicate elimination
  - Sort tuples and discard duplicates

# Parallel Algorithms for Query Processing (cont'd.)

- Parallel joins divide the join into  $n$  smaller joins
  - Perform smaller joins in parallel on  $n$  processors
  - Take a union of the result
- Parallel join techniques
  - Equality-based partitioned join
  - Inequality join with partitioning and replication
  - Parallel partitioned hash join

# Parallel Algorithms for Query Processing (cont'd.)

- Aggregation

- Achieved by partitioning on the grouping attribute and then computing the aggregate function locally at each processor

- Set operations

- If argument relations are partitioned using the same hash function, they can be done in parallel on each processor



# Parallel Algorithms for Query Processing (cont'd.)

- Intraquery parallelism
  - Approaches
    - Use parallel algorithm for each operation, with appropriate partitioning of the data input to that operation
    - Execute independent operations in parallel
- Interquery parallelism
  - Execution of multiple queries in parallel
  - Goal: scale up
  - Difficult to achieve on shared-disk or shared-nothing architectures

# 18.9 Summary

- SQL queries translated into relational algebra
- External sorting
- Selection algorithms
- Join operations
- Combining operations to create pipelined execution
- Parallel database system architectures