KOREA TECH
KOREA UNIVERSITY OF TECHNOLOGY & EDUCATION

Database System

Query Processing and Optimization

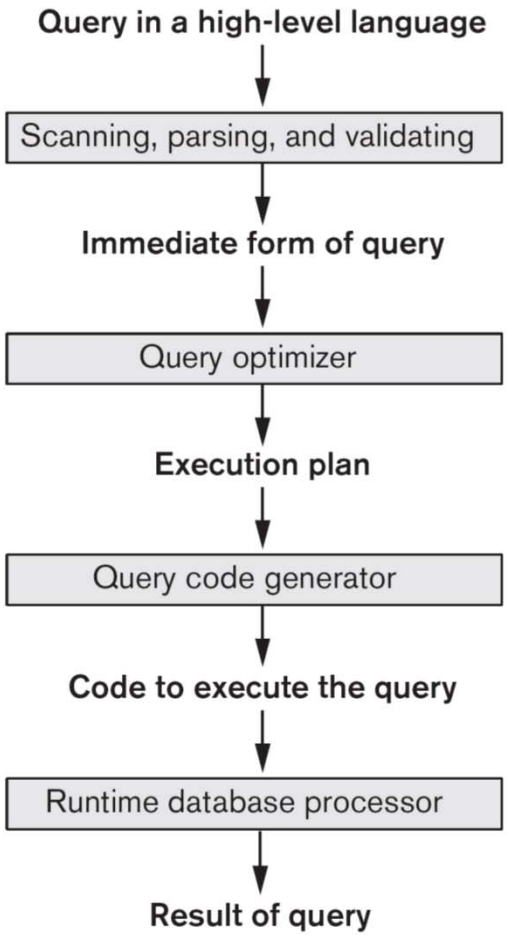
Muhammad Tariq Mahmood
tariq@koreatech.ac.kr
School of Computer Science and Engineering
Korea University of Technology and Education

Query Processing

- **Query optimization**:
  - The process of choosing a suitable execution strategy for processing a query.

- Two internal representations of a query:
  - Query Tree
  - Query Graph

Query Processing



Code can be:

Executed directly (interpreted mode)

Stored and executed later whenever needed (compiled mode)

Figure 19.1
Typical steps when processing a high-level query.

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 clause if these are part of the block.

- **Nested queries** within a query are identified as separate query blocks.

- Aggregate operators in SQL must be included in the extended algebra.

Translating SQL Queries into Relational Algebra

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

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

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

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

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

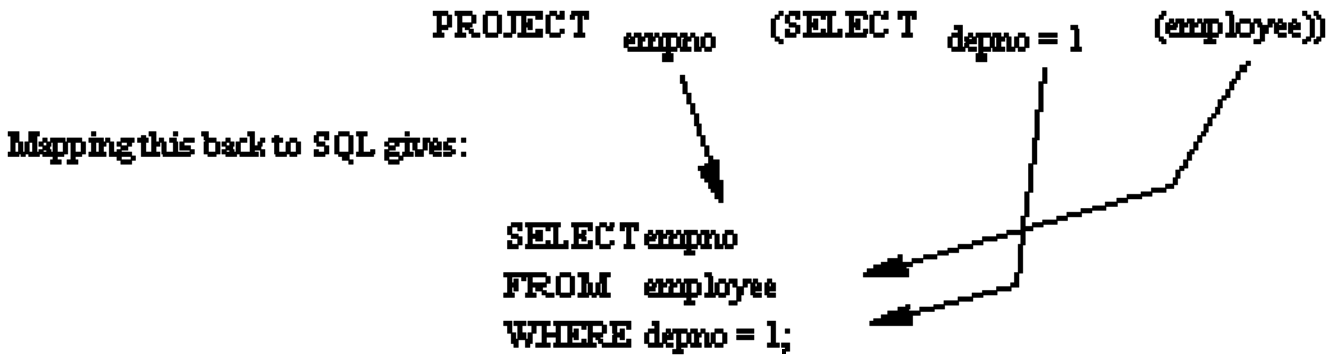
Translating SQL Queries into Relational Algebra

- Relational Algebra is :
  - the formal description of how a relational database operates
  - an interface to the data stored in the database itself
  - the mathematics which underpin SQL operations

- Terminology
  - Relation – a set of tuples.
  - Tuple – a collection of attributes which describe some real world entity.
  - Attribute – a real world role played by a named domain.
  - Domain – a set of atomic values.
  - Set – a mathematical definition for a collection of objects which contains no duplicates.

Translating SQL Queries into Relational Algebra

- There are two groups of operations:
  - Mathematical set theory based relations: UNION, INTERSECTION, DIFFERENCE, and CARTESIAN PRODUCT.
  - Special database operations: SELECT ($\text{SELECT}_{\text{dob '01/JAN/1950'}}(\text{employee})$, PROJECT, and JOIN.



Algorithms for External Sorting

- **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, such as most database files.

- **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: $d_M = \text{Min}\ (n_B - 1, n_R)$; $n_P = \lceil (\log_{dM}(n_R)) \rceil$
  - n_R: number of initial runs; b: number of file blocks;
  - n_B: available buffer space; d_M: degree of merging;
  - n_P: number of passes.

Algori

```
set     i ← 1;
        j ← b;          {size of the file in blocks}
        k ← n_B;        {size of buffer in blocks}
        m ← ⌈(j/k)⌉;

{Sorting 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;
}

{Merging Phase: merge subfiles until only 1 remains}
set     i ← 1;
        p ← ⌈log_{k−1} m⌉   {p is the number of passes for the merging phase}
        j ← m;
while (i ≤ p)
do {
        n ← 1;
        q ← ⌈j/(k−1)⌉;  {number of subfiles 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;
}
```

Figure 19.2
Outline of the sort-merge algorithm for external sorting.

Algorithms for SELECT and JOIN Operations

- Implementing the SELECT Operation

- Examples:
  - (OP1): $\sigma_{SSN='123456789'}(EMPLOYEE)$
  - (OP2): $\sigma_{DNUMBER>5}(DEPARTMENT)$
  - (OP3): $\sigma_{DNO=5}(EMPLOYEE)$
  - (OP4): $\sigma_{DNO=5\ AND\ SALARY>30000\ AND\ SEX=F}(EMPLOYEE)$
  - (OP5): $\sigma_{ESSN=123456789\ AND\ PNO=10}(WORKS\_ON)$

Algorithms for SELECT and JOIN Operations

- Implementing the SELECT Operation
- Search Methods for Simple Selection:
  - 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. (See OP1).
  - 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.

Algorithms for SELECT and JOIN Operations

- Implementing the SELECT Operation
- Search Methods for Simple Selection:
  - 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.
  - 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)

Algorithms for SELECT and JOIN Operations

- Implementing the SELECT Operation
- Search Methods for Simple Selection:
  - S7 **Conjunctive 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.
  - S8 **Conjunctive 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.

Algorithms for SELECT and JOIN Operations

- Implementing the SELECT Operation
- Search Methods for Complex Selection:
  - S9 **Conjunctive 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.

Algorithms for SELECT and JOIN Operations

- Implementing the SELECT Operation
  - 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. (See OP1, OP2 and OP3)
  - For **conjunctive selection conditions**, whenever *more than one* of the attributes involved in the conditions have an access path, query optimization should be done to choose the access path that *retrieves the fewest records* in the most efficient way.
  - **Disjunctive selection conditions**

Algorithms for SELECT and JOIN Operations

- Implementing the JOIN Operation:
  - Join (EQUIJOIN, NATURAL JOIN)
    - two-way join: a join on two files
    - e.g. $R \underset{A=B}{} S$
    - multi-way joins: joins involving more than two files.
    - e.g. $R \underset{A=B}{} S \underset{C=D}{} T$
- Examples
  - (OP6): EMPLOYEE $\underset{DNO=DNUMBER}{}$ DEPARTMENT
  - (OP7): DEPARTMENT $\underset{MGRSSN=SSN}{}$ EMPLOYEE

Algorithms for SELECT and JOIN Operations

- Methods for implementing joins:
  - 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].

Algorithms for SELECT and JOIN Operations

- Methods for implementing joins:
  - 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.

Algorithms for SELECT and JOIN Operations

- Methods for implementing joins:
  - 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.

Figure 19.3
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 $T \leftarrow R \bowtie_{A=B} S$. (b) Implementing the operation $T \leftarrow \pi_{<\text{attribute list}>}(R)$.

(a)

```
sort the tuples in R on attribute A;          (* assume R has n tuples (records) *)
sort the tuples in S on attribute B;          (* assume S has m tuples (records) *)
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  {  (* R(i) [A] = S(j) [B], so we output a matched tuple *)
               output the combined tuple <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 ← k, j ← l
      }
}
```

Algorithms for SELECT and JOIN Operations

(b) create a tuple t[<attribute list>] in T' for each tuple t in R;
(* T' contains the projection results *before* duplicate elimination *)
if <attribute list> 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 *)

Algorithms for SELECT and JOIN Operations

(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;

Algorithms for SELECT and JOIN Operations

(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 $i \leftarrow i + 1$, $j \leftarrow j + 1$
}
}

(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;

Algorithms for SELECT and JOIN Operations

- Implementing the JOIN Operation
- Factors affecting JOIN performance
  - Available buffer space
  - Join selection factor
  - Choice of inner VS outer relation

Algorithms for SELECT and JOIN Operations

- Implementing the JOIN Operation
- Other types of JOIN algorithms
- Partition hash join
  - Partitioning phase:
    - Each file (R and S) is first partitioned into M partitions using a partitioning hash function on the join attributes:
      - R1 , R2 , R3 , ... Rm and S1 , S2 , S3 , ... Sm
    - Minimum number of in-memory buffers needed for the partitioning phase: M+1.
    - A disk sub-file is created per partition to store the tuples for that partition.
  - Joining or probing phase:
    - Involves M iterations, one per partitioned file.
    - Iteration i involves joining partitions Ri and Si.

Algorithms for SELECT and JOIN Operations

- Implementing the JOIN Operation (contd.):
- Partitioned Hash Join Procedure:
  - Assume Ri is smaller than Si.
    1. Copy records from Ri into memory buffers.
    2. Read all blocks from Si, one at a time and each record from Si is used to *probe* for a matching record(s) from partition Si.
    3. Write matching record from Ri after joining to the record from Si into the result file.

Algorithms for SELECT and JOIN Operations

- Implementing the JOIN Operation (contd.):

- Cost analysis of partition hash join:
  1. Reading and writing each record from R and S during the partitioning phase:
     $(b_R + b_S), (b_R + b_S)$
  2. Reading each record during the joining phase:
     $(b_R + b_S)$
  3. Writing the result of join:
     b_{RES}
- Total Cost:
  - $3^* (b_R + b_S) + b_{RES}$

Algorithms for SELECT and JOIN Operations

- Implementing the JOIN Operation (contd.):
- **Hybrid hash join:**
  - Same as partitioned hash join except:
    - Joining phase of one of the partitions is included during the partitioning phase.
  - **Partitioning phase**:
    - Allocate buffers for smaller relation- one block for each of the M-1 partitions, remaining blocks to partition 1.
    - Repeat for the larger relation in the pass through S.)
  - **Joining phase**:
    - M-1 iterations are needed for the partitions R2 , R3 , R4 , ...Rm and S2 , S3 , S4 , ...Sm. R1 and S1 are joined during the partitioning of S1, and results of joining R1 and S1 are already written to the disk by the end of partitioning phase.

Algorithms for PROJECT and SET Operations

- Implementing the JOIN Operation (contd.):
- **Hybrid hash join:**
  - Same as partitioned hash join except:
    - Joining phase of one of the partitions is included during the partitioning phase.
  - **Partitioning phase**:
    - Allocate buffers for smaller relation- one block for each of the M-1 partitions, remaining blocks to partition 1.
    - Repeat for the larger relation in the pass through S.)
  - **Joining phase**:
    - M-1 iterations are needed for the partitions R2 , R3 , R4 , ...Rm and S2 , S3 , S4 , ...Sm. R1 and S1 are joined during the partitioning of S1, and results of joining R1 and S1 are already written to the disk by the end of partitioning phase.