Database Systems

Luiz Jonatã Pires de Araújo
l.araujo@innopolis.university

5. Strategies for Query Processing

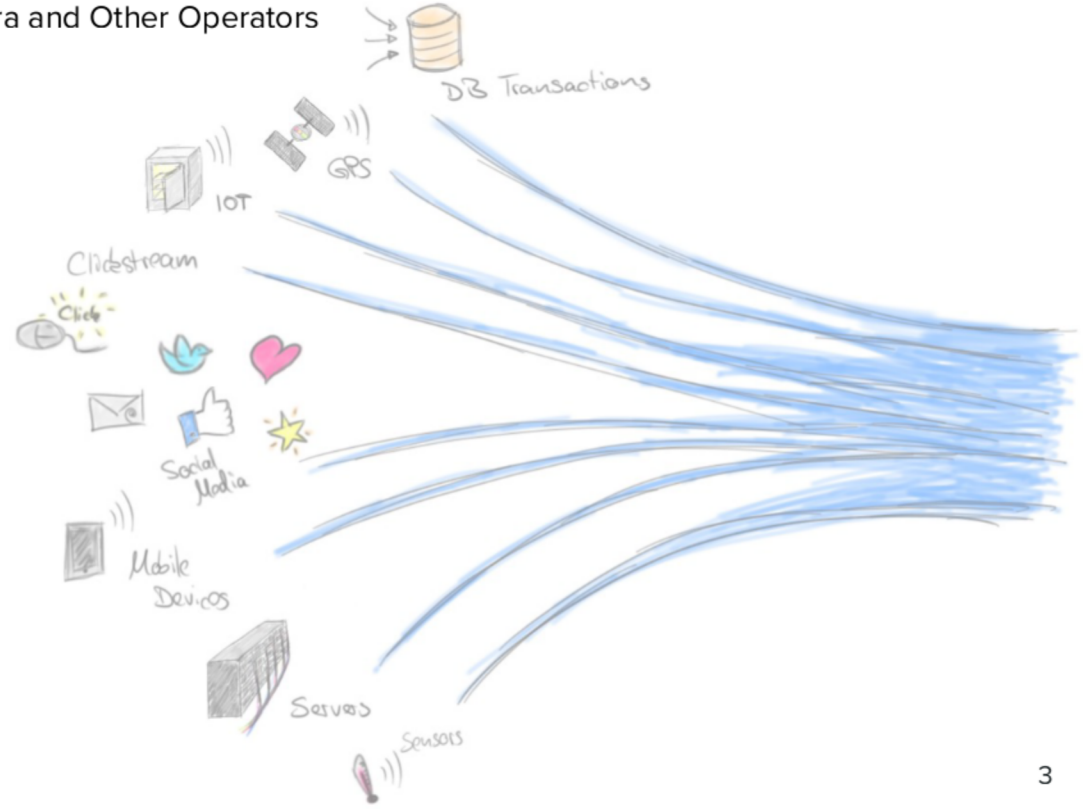
Giving credit where it's due:

- This material is based on the 7th edition of 'Fundamentals of Database Systems" by Elmasri and Navathe

  https://www.pearson.com/us/higher-education/program/Elmasri-Fundamentals-of-Database-Systems-7th-Edition/PGM189052.html
- Specialized Course "Query Optimization" - Saarland University, Germany
  http://resources.mpi-inf.mpg.de/departments/d5/teaching/ws06_07/queryoptimization/

Today

- Translating SQL Queries into Relational Algebra and Other Operators
- Algorithms for External Sorting
- Algorithms for SELECT Operation
- Implementing the JOIN Operation
- Set Operations
- Implementing Aggregate Operations
- Combining Operations Using Pipelining



Query Processing

Have you ever wondered how high-level queries are internally “compiled” and executed?

1. **Scanned**
    - It identifies the query tokens - such as SQL keywords, attribute names, and relation names - that appear in the text of the query
2. **Parsed**
    - It checks the query syntax to determine whether it is formulated according to the syntax rules (rules of grammar) of the query language
3. **Validated**
    - It checks whether all attribute and relation names are valid and semantically meaningful names in the schema of the particular database being queried
4. **An internal representation of the query is then created**, usually as a tree data structure called a query tree
5. The DBMS must then **devise an execution strategy** (query optimization) or query plan

Query Processing

This lecture focuses on how queries are processed and what algorithms are used to perform individual operations within the query.

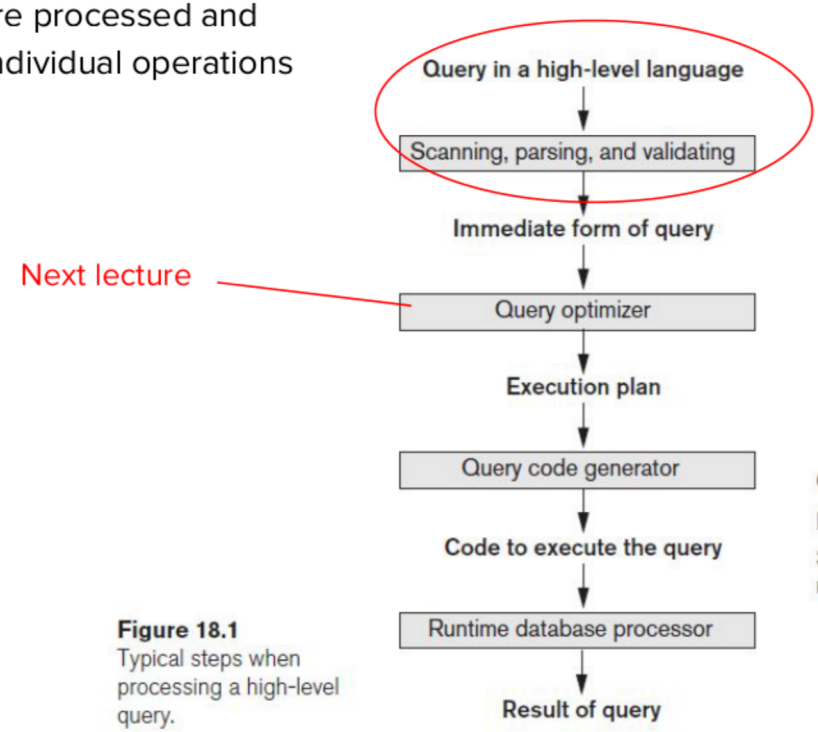
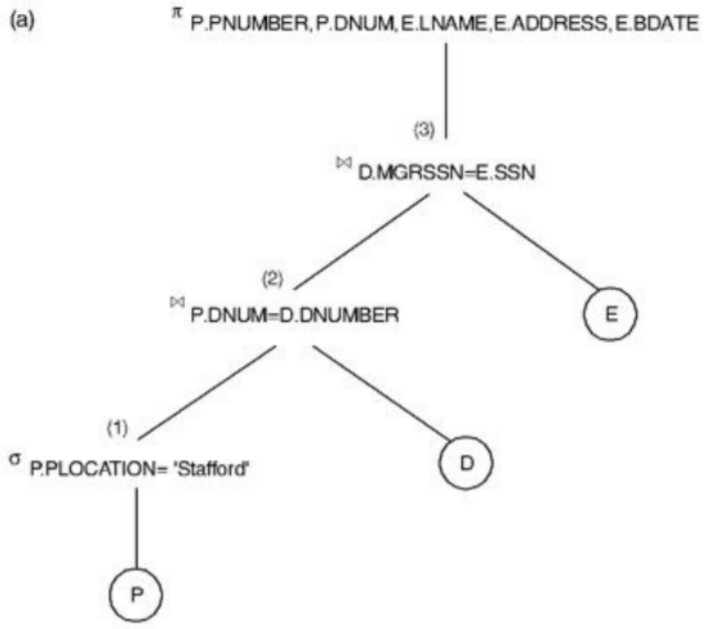


Figure 18.1
Typical steps when processing a high-level query.

Translating SQL Queries into Relational Algebra and Other Operators

SQL Queries → Relational Algebra

- In practice, SQL is the query language that is used in most commercial RDBMSs
- An SQL query is first translated into an equivalent extended relational algebra expression - represented as a query tree data structure - that is then optimized
- Typically, SQL queries are decomposed into query blocks, which form the basic units that can be translated into the algebraic operators and optimized



Query Block

- 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
- Because SQL includes aggregate operators - such as MAX, MIN, SUM, and COUNT - these operators must also be included in the extended algebra
- Example:

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

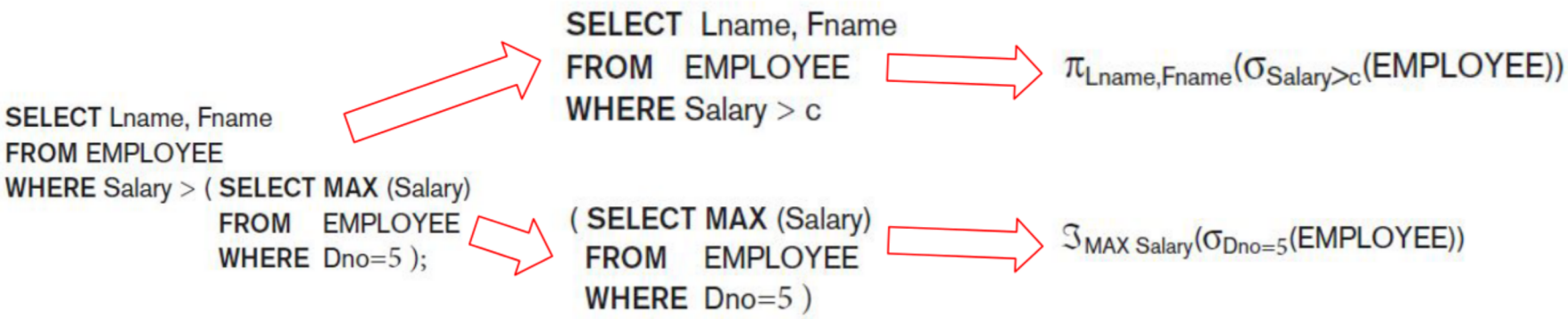
```
SELECT  Lname, Fname
FROM    EMPLOYEE
WHERE Salary > c
```

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

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

$$\Im_{MAX\ Salary}(\sigma_{Dno=5}(EMPLOYEE))$$

Query Block



- The query optimizer would then choose an execution plan for each query block
- It is more involved to optimize the more complex correlated nested subqueries, where a tuple variable from the outer query block appears in the WHERE-clause of the inner query block
- Many techniques are used in advanced DBMS to unnest and optimize correlated nested subqueries

Correlated or uncorrelated?

```
SELECT Name, Description
FROM Products
WHERE Quantity <  2 * ( SELECT AVG(Quantity)
                        FROM SalesOrderItems );
```

Uncorrelated

```
DELETE FROM table1 alias1
WHERE column1 operator in
               (SELECT expression FROM table2 alias2
                WHERE alias1.column = alias2.column);
```

Correlated

```
SELECT * FROM
  student
  WHERE dep_id = ( SELECT id FROM department
                   WHERE name = 'Computer' );
```

Uncorrelated

Correlated or uncorrelated?

```
SELECT last_name, salary, department_id
 FROM employees outer
 WHERE salary > (SELECT AVG(salary)
                 FROM employees
                 WHERE department_id = outer.department_id);
```

Correlated

```
SELECT Salesperson.Name FROM Salesperson
WHERE Salesperson.ID NOT IN(
      SELECT Orders.salesperson_id FROM Orders, Customer
      WHERE Orders.cust_id = Customer.ID and Customer.Name = 'Samsonic')
```

Uncorrelated

```
SELECT employee_id, last_name, job_id, department_id
FROM employees outer
WHERE EXISTS ( SELECT 'X' FROM employees WHERE manager_id = outer.employee_id);
```

Correlated

Algorithms for External Sorting

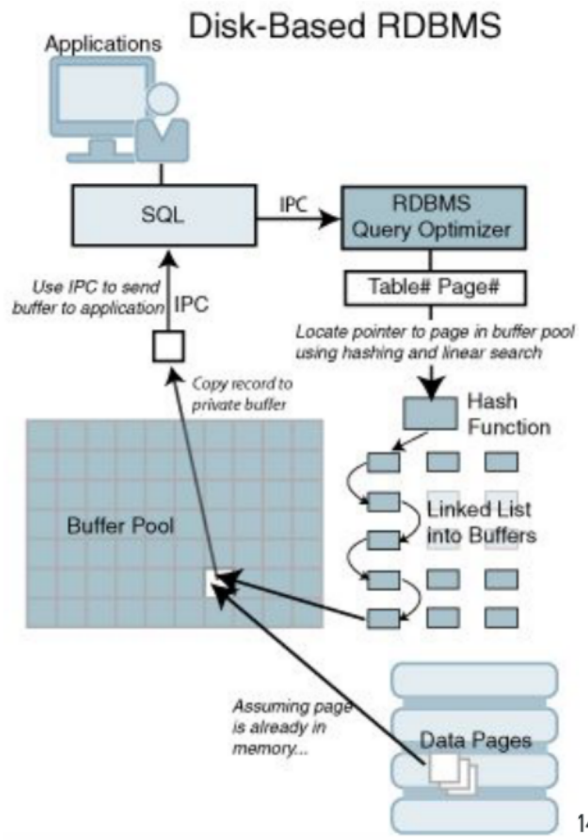
Algorithms for External Sorting

- Sorting is one of the primary algorithms used in query processing 
- For example, whenever an SQL query specifies an ORDER BY-clause, the query result must be sorted
- Also for duplicate elimination algorithms for the PROJECT operation (when an SQL query specifies the DISTINCT option in the SELECT clause)
- Sorting is also a key component in sort-merge algorithms used for JOIN and other operations (such as UNION and INTERSECTION) 

Note that sorting of a particular file may be avoided if an appropriate index - such as a primary or clustering index - exists on the desired file attribute to allow ordered access to the records of the file

Algorithms for External Sorting

- Sometimes is necessary to sort large files of records stored on disk that do not fit entirely in main memory
- The typical external sorting algorithm uses a sort-merge strategy, which starts by sorting small subfiles - called runs - of the main file and then merges the sorted runs, creating larger sorted subfiles that are merged in turn
- The sort-merge algorithm, like other database algorithms, requires buffer space in main memory, where the actual sorting and merging of the runs is performed



Analogy with MergeSort

Given:

- number of initial runs (n_R)
- Number of file (or disk) blocks in the file (b)
- Number of available main memory buffers (n_B)
- $n_R = \lceil b / n_B \rceil$

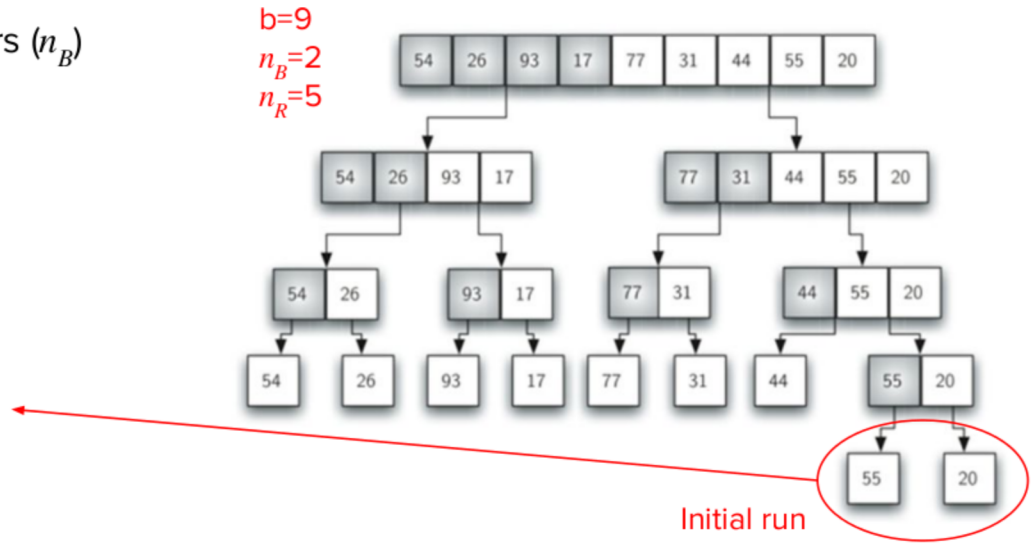
Runs are read into main memory,

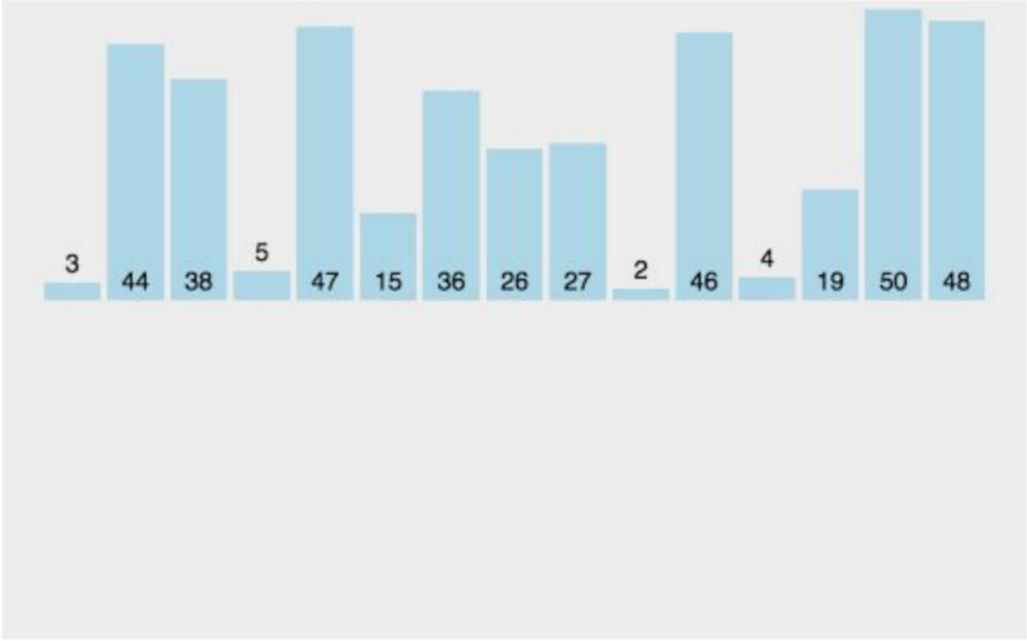
sorted using an internal sorting algorithm

Runs are written back to disk as temporary sorted

subfiles (or runs)

Worst-case: $O(n \lg n)$




3
44
38
5
47
15
36
26
27
2
46
4
19
50
48

Algorithms for SELECT Operation

SELECT operator

It is basically a search operation to locate the records in a disk file that satisfy a certain condition

Examples

- $\sigma_{Ssn = '123456789'}$ (EMPLOYEE)
- $\sigma_{Dnumber > 5}$ (DEPARTMENT)
- $\sigma_{Dno= 5}$ (EMPLOYEE)
- $\sigma_{Dno= 5 \text{ AND } Salary > 30000 \text{ AND } Sex = 'F'}$ (EMPLOYEE)
- $\sigma_{Essn = '123456789' \text{ AND } Pno =10}$(WORKS_ON)
- SELECT *
  FROM EMPLOYEE
  WHERE Dno IN (3,27, 49)
- SELECT First_name, Lname
  FROM Employee
  WHERE ((Salary*Commission_pct) + Salary) > 15000;

Search algorithms can either search for records from a file (file scans) or a index file (index scan)

Algorithms for SELECT Operation

Algorithms:

- S1- **Linear search** (brute force algorithm): Each disk block is read into a main memory buffer; search within the disk blocks (in memory)
- S2 - **Binary search**: The selection condition involves an equality comparison on a key attribute on which the file is ordered
- S3a - **Using a primary index**: The selection condition involves an equality comparison on a key attribute with a primary index
  - This condition retrieves a single record (at most)
  - Example: $\sigma_{Ssn = '123456789'}$ (EMPLOYEE)
- S3b - **Using a hash key**: ... equality comparison on a key attribute with a hash key
  - It retrieves a single record (at most)

Algorithms for SELECT Operation

Algorithms (continued):

- S4 - **Using a primary key to retrieve multiple records**: >, >=, <, or <= on a key field with a primary index
  - It uses the index to find the record satisfying the corresponding equality condition
  - then retrieve all subsequent (or precedent) records in the (ordered) file
- S5 - **Using a clustering index to retrieve multiple records**: ... equality comparison on a nonkey attribute with a clustering index
  - Use the index to retrieve all the matching records
  - $\sigma_{Dnumber > 5}$ (DEPARTMENT)

Algorithms for SELECT Operation

Algorithms (continued):

- S6 - **Using a secondary (B+-tree) index on an equality comparison**:
  - Indexing field is a key (unique values) ➔ It returns a single record
  - Indexing field is not a key ➔ It retrieves multiple records
  - Range queries: Leaf nodes contain the index field in order (remember properties of B+-trees)
- S7a - **Using a bitmap index**: ... involves a set of values for an attribute - SELECT * FROM EMPLOYEE WHERE Dno IN (3,27, 49) - the corresponding bitmaps for each value can be OR-ed to give the set of record ids that qualify. In this example, that amounts to OR-ing three bitmap vectors whose length is the same as the number of employees
- S7b - Using a functional index: It is suitable when the match is based on a function of one or more attributes on which a functional index (Oracle) exists

See: https://oracle-base.com/articles/8i/function-based-indexes

Algorithms for SELECT Operation

Algorithms (continued) - conjunctive selection:

- S8 - **Conjunctive selection (AND) using an individual index**: Use of one of the methods S2 to S6 to retrieve the records and then check whether each retrieved record satisfies the remaining simple conditions in the conjunctive select condition
- S9 - **Conjunctive selection (AND) using a composite index**: If a composite index (or hash structure) exists on the combined fields we can use the index directly

Algorithms for SELECT Operation

Algorithms (continued) - conjunctive selection:

- S10 - **Conjunctive selection (AND) by intersection of record pointers**:
  - Secondary indexes are available on more than one of the fields + indexes include record pointers
  - Each index can be used to retrieve the set of record pointers
  - The intersection of these sets of record pointers gives the record pointers that satisfy the conjunctive select 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

Disjunctive Selection

- Example: $\sigma_{Dno=5 \textbf{ OR } Salary > 30000 \textbf{ OR } Sex ='F'}$ (EMPLOYEE)
- Disjunctive Selection (OR) are much harder to process and optimize
- The records satisfying the disjunctive condition are the union of the records satisfying the individual conditions
- All the methods discussed in S1 through S7 are applicable

Which algorithm to choose for a certain SELECT statement?
This is the role of the query optimizer (next lecture).
It uses formulas that **estimate the costs** for each available access method

Estimating the Selectivity of a Condition

- Fraction that estimates the percentage of file records that will be retrieved
- Estimates of selectivities are possible from the information kept in the DBMS catalog and are used by the optimizer
- For each relation (table) r with schema R containing r_R tuples:
  - The number of rows: r_R
  - The "width" of the relation (i.e., the length of each tuple in the relation) this length of tuple is referred to as R
  - The number of blocks that relation occupies in storage: referred to as b_R
  - The blocking factor bfr, which is the number of tuples per block
- For each attribute A in relation R:
  - The number of distinct values (NDV) of A in R: NDV (A, R).
  - The max and min values of attribute A in R: max (A, R) and min (A, R)

Implementing the JOIN Operation

Implementing the JOIN Operation

- One of the most time-consuming operations in query processing
- Focus on two-way joins, which are joins on only two files

$$R \bowtie_{A=B} S$$

OP6: EMPLOYEE $\bowtie_{Dno=Dnumber}$ DEPARTMENT
OP7: DEPARTMENT $\bowtie_{Mgr\_ssn=Ssn}$ EMPLOYEE

- In the following slides, methods for implementing joins

Implementing the JOIN Operation

$$R \bowtie_{A=B} S$$

- J1 - **Nested-loop join (or nested-block join)**
  - This is the default (brute force)
  - For each record in the outer loop, retrieve every record from the inner loop and test whether the two records satisfy the join condition
- J2 - **Index-based nested-loop join**
  - Index (or hash key) exists for one of the two join attributes - say, attribute B of file S
  - Retrieve each record t in R (loop over file R)
  - Then use the access structure (such as an index or a hash key) to retrieve directly all matching records s from S that satisfy s[B] = t[A]
- J3 - **Sort-merge join**
  - Records of R and S are physically sorted (ordered) by value of the join attributes
  - Pairs of file blocks are copied into memory buffers in order and the records of each file are scanned only once each for matching with the other file

How buffer space and performance

EMPLOYEE $\bowtie_{Dno=Dnumber}$ DEPARTMENT

- It is advantageous to read as many blocks as possible at a time into memory from the file whose records are used for the outer loop ➔ reduce seek time
- The algorithm reads one block at a time for the inner-loop file and use its records to probe (that is, search) the outer-loop blocks that are currently in main memory for matching records
- An extra buffer in main memory is needed to contain the resulting records after they are joined, and the contents of this result buffer can be appended to the result file - the disk file that will contain the join result - whenever it is filled

Choice of outer-loop and performance

EMPLOYEE $\bowtie_{Dno=Dnumber}$ DEPARTMENT

Assume that the DEPARTMENT file consists of $r_D = 50$ records stored in $b_D = 10$ disk blocks
Assume EMPLOYEE file consists of $r_E = 6{,}000$ records stored in $b_E = 2{,}000$ disk blocks

It is advantageous to use the file with fewer blocks as the outer-loop file in the nested-loop join!

-2 because it's one block for the inner loop and one block for the saving the results

If EMPLOYEE is used for the outer loop, each block of EMPLOYEE is read once, and the entire DEPARTMENT file (each of its blocks) is read once for *each time* we read in $(n_B - 2)$ blocks of the EMPLOYEE file.

Total number of blocks accessed (read) for outer-loop file = b_E

Number of times $(n_B - 2)$ blocks of outer file are loaded into main memory = $\lceil b_E/(n_B - 2) \rceil$

Total number of blocks accessed (read) for inner-loop file = $b_D * \lceil b_E/(n_B - 2) \rceil$

Hence, we get the following total number of block read accesses:

$b_E + (\lceil b_E/(n_B - 2) \rceil * b_D) = 2000 + (\lceil (2000/5) \rceil * 10) = 6000$ block accesses

On the other hand, if we use the DEPARTMENT records in the outer loop, by symmetry we get the following total number of block accesses:

$b_D + (\lceil b_D/(n_B - 2) \rceil * b_E) = 10 + (\lceil (10/5) \rceil * 2000) = 4010$ block accesses

Set Operations

Set Operations

- CARTESIAN PRODUCT: computationally expensive
- UNION, INTERSECTION, and SET DIFFERENCE
  - Only to type-compatible relations
  - The customary way to implement these operations is to use variations of the sort-merge technique
    - The two relations are sorted on the same attributes, and, after sorting, a single scan through each relation is sufficient to produce the result
- Hashing

  One table is first scanned and then partitioned into an in-memory hash table with buckets, and the records in the other table are then scanned one at a time and used to probe the appropriate partition

Implementing Aggregate Operations

Implementing Aggregate Operations

- **MIN, MAX, COUNT, AVERAGE, SUM**: when applied to an entire table, can be computed by a table scan or by using an appropriate index
  - Example: MAX or MIN of an attribute for which a B+-tree exists → rightmost pointer in each index node
  - An index could be used for the AVERAGE and SUM aggregate functions, but only if it is a dense index (there is an index entry for every record in the main file)

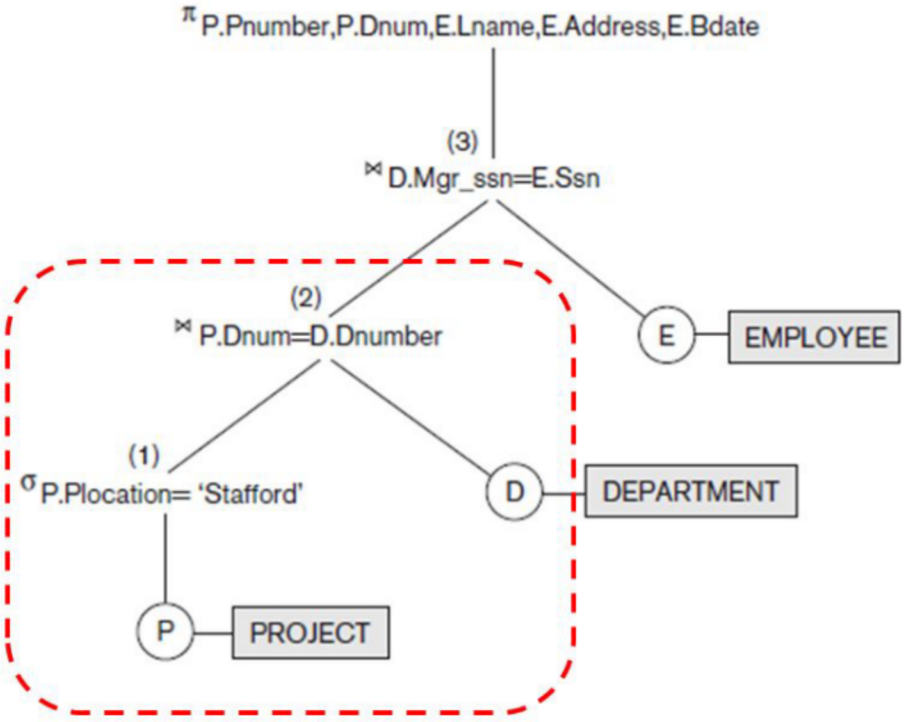
scanning the index file

- **GROUP BY**
  - Sorting or hashing on the grouping attributes to partition the file into the appropriate groups
  - A clustering index on the grouping attribute(s) the records are already partitioned (grouped) into the appropriate subsets only apply the computation to each group

Combining Operations Using Pipelining

Combining Operations Using Pipelining

- Executing a single operation at a time generates temporary files on disk to hold the results of these temporary operations (materialized evaluation)
  - Excessive overhead
- We want to reduce the generation and storing of large temporary files
- 1. Generate query execution code that corresponds to algorithms for combinations of operations in a query
- 2. Pipeline
  - As the result tuples from one operation are produced, they are provided as input for subsequent operations
  - Data is not copied nor materialized
  - Generating results as quickly as possible



It could be implemented by one algorithm with two input files and a single output file

Interesting reading: http://resources.mpi-inf.mpg.de/departments/d5/teaching/ws06_07/queryoptimization/Lecture8.pdf