QUERY PROCESSING AND OPTIMIZATION

Query processing

SQL queries written in the declarative form:

```
SELECT * FROM STUDENT
WHERE Class_num = 2;
```

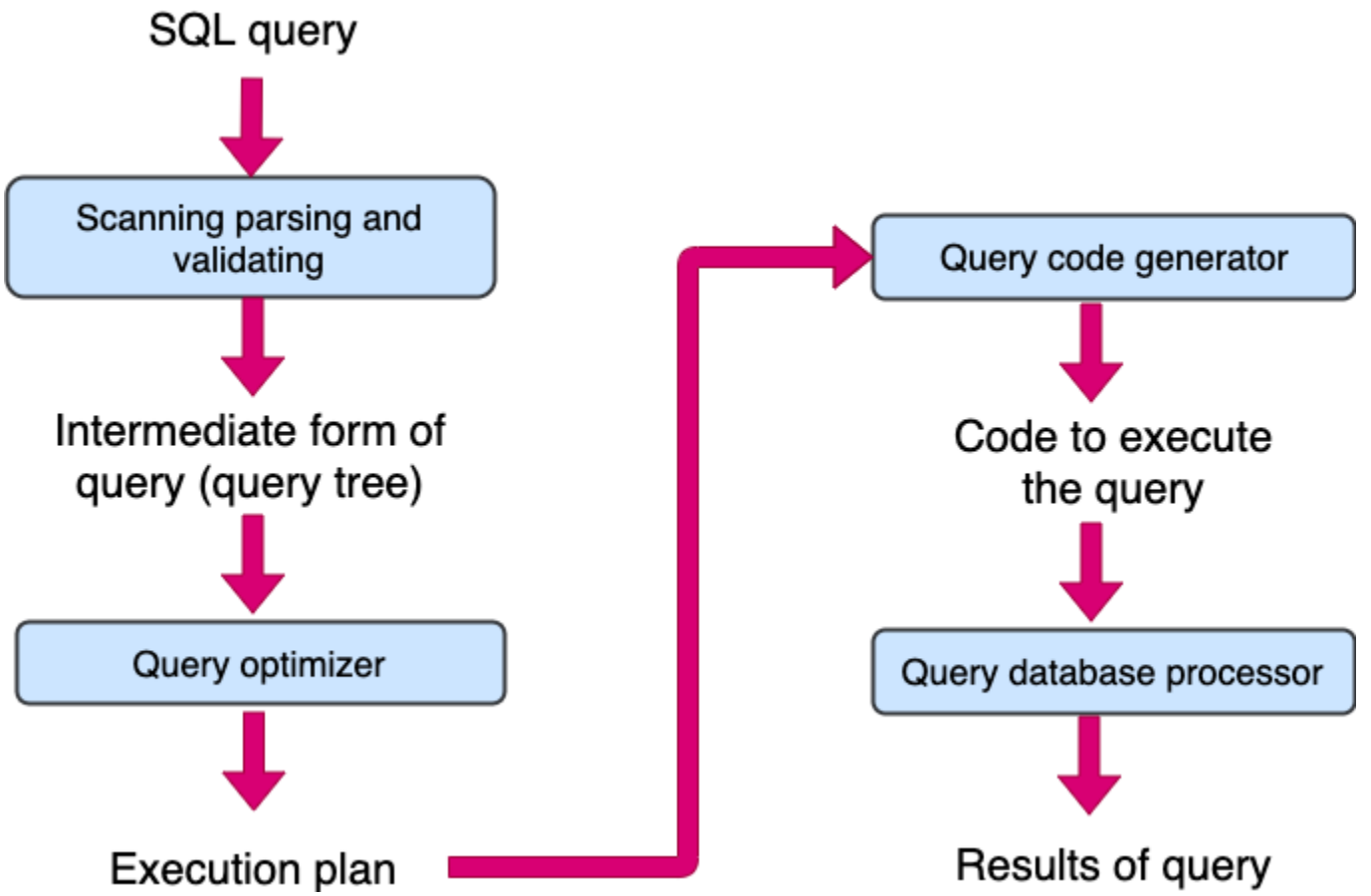
should be transformed into an imperative form similar to the following

```
students = read_students();
for student in students  :
    if  student.Class_num == 2:
        print(student)
```

and executed over a data file

The process of transforming SQL code into executable code goes through a number of steps.

Query processing - Architecture



Translating SQL queries into Relational algebra

Scanning, parsing and validation of the query creates an internal representation of the query called **query tree**

– query tree is created using relational algebra operators

SQL query is translated into an equivalent relational algebra expression

Consider the following SQL query:

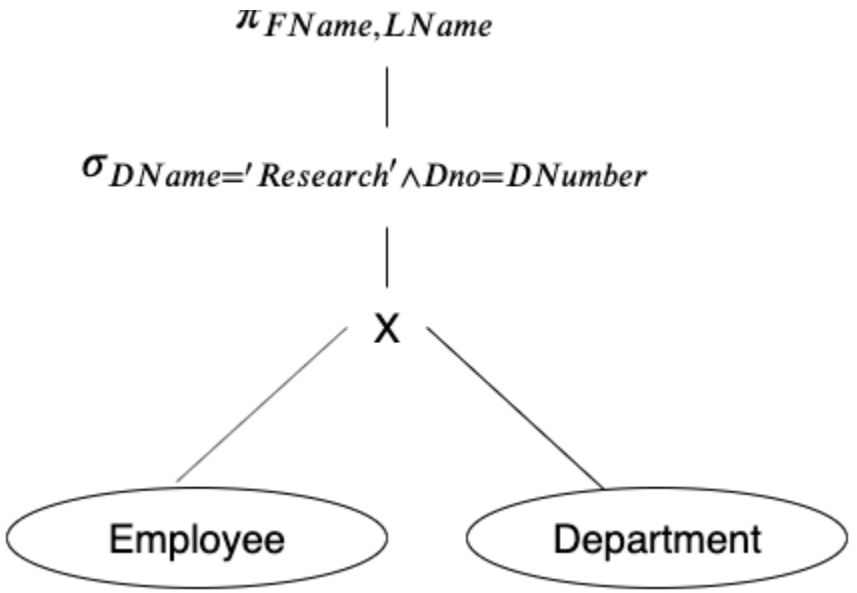
```
SELECT Fname, Lname
FROM Employee NATURAL JOIN Department
WHERE Department.DName='Research'
```

Relational algebra statement corresponding to this SQL query:

$$\pi_{Fname,Lname}(\sigma_{DName=\text{"}Research\text{"}}(Employee \bowtie_{DNo=DNumber} Department$$

Query tree

Query tree corresponding to the previous relational algebra expression



- *leaf nodes* of query trees are operands, i,e relations or tuple variables
- *internal nodes* of the query tree are relational algebra operations and expressions

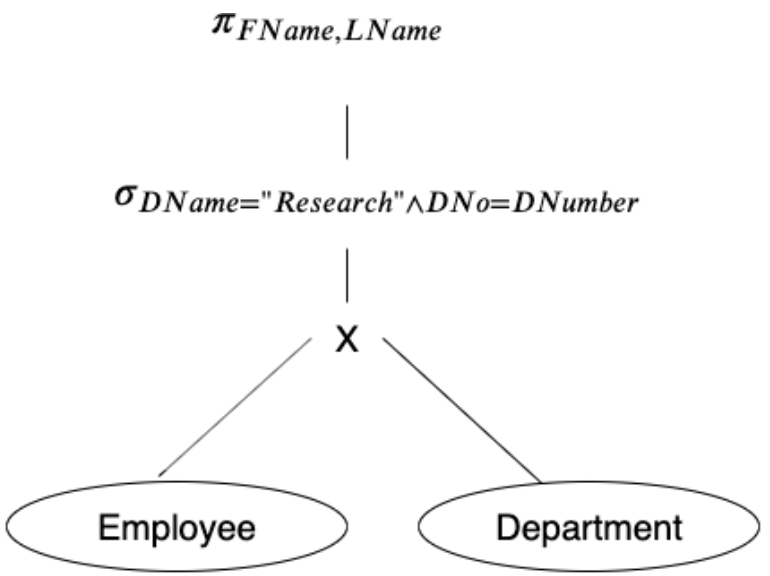
Execution of the query tree consists of executing internal node operations whenever the operands are available

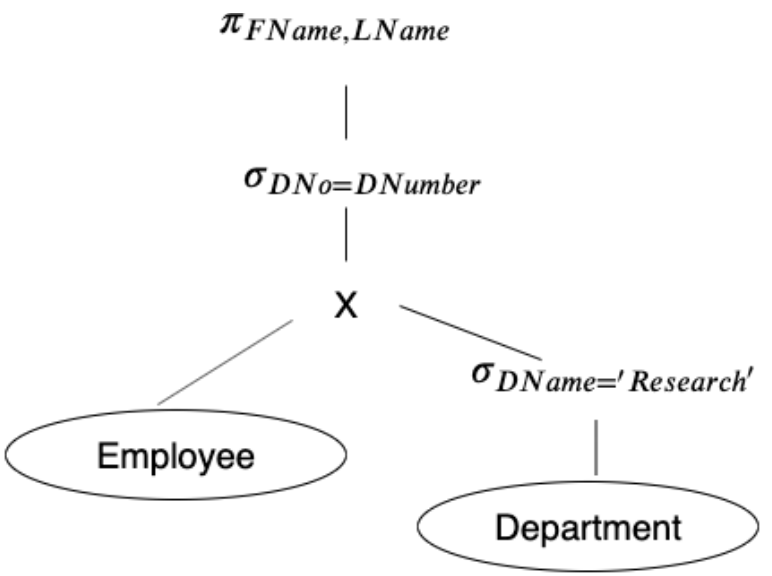
- execution starts at the leaf nodes and ends with the root node

Many query trees for the same query

A query typically has many possible execution strategies

– for instance, equivalent relational algebra expressing





Goal: Finding a query tree which would be executed with the lowest cost

Query optimization

Query tree is augmented with additional information of

- access methods available for each relation
- algorithms to be used in computing relational operators

Augmented query tree is an **execution plan**

Query optimization is a process of finding an **optimal execution plan** (optimal execution strategy)

Optimal execution plan is dependent on many properties:

- blocks size, number of tuples in a relation
- size of the buffer pool
- number of distinct values of attribute A in a relation R - $val_{A,R}$
- number of levels of index tree over attribute A : $lev_{I(R,A)}$
- …

Optimization - example

```
SELECT Fname, Lname FROM Employee NATURAL JOIN Department
WHERE Department.DName='Research'
```

Case 1 (trivial example): If relations Employee and Department are so small that each of them has only one record, the execution may include the following steps:

- transferring blocks containing relations all together into the buffer pool
- finding Cartesian product of two relations (single tuple)
- executing select condition over the tuple
- executing projection

Case 2: If relations Department and Employee have many thousands of records then the following steps can be considered

- executing selection operations at first to decrease the number of records
- executing projection lower in the tree to decrease the size of intermediate results
- if there is an index consider using it to find natural join of relations
- if there are no indexes use linear search (sequential scan) to find natural join

Goals of the optimization

The main goal of the optimization is to execute queries as fast as possible

- Decrease the number of block accesses during execution (to lower the seek and latency time)
- Decrease the number of tuples processed during execution
  - execute selection as soon as possible
  - execute operation without saving intermediate results if possible
  - make no redundant operations (operations which lead to empty results)
  - reuse of results of intermediate expressions if they are used many times

Algorithms for query processing in main memory are also optimized but we don't focus on them here because the block access operations are much more expensive

Search methods - File scans

File scan is a search algorithm for selecting records from a file

- retrieves records that satisfy a selection condition
- is called **index scan** if it includes the use of an index

We name here the following file scan algorithms which can be used for select operation:

- **Linear search** (brute force algorithm)
  - search every record in a file (it's called also full table scan or sequential scan)
- **Binary search** - involves equality on a key attribute on which the file is ordered
- **Scan using a hash key** - used for equality comparison on a key attribute
  - can be used only for equality comparison that retrieve at most one record
- **Scan using a primary index**
- **Scan using a clustering index**
- **Scan using a secondary (B^+-tree) index**
  - can be used for equality comparisons and for comparisons involving $>$, $\geq$, $<$, $\leq$ (*range queries*)

File scan - examples

Consider our table Student(Ssn, Fname, Lname, Year) from the previous examples and the following queries:

- retrieve all fields of all records from the Student table:

```
SELECT * FROM Student;
```

- retrieve all fields of all records from the Student table sorted by year in ascending order:

```
SELECT * FROM Student WHERE YEAR > 2 ORDER BY Year ASC;
```

First query does not require any ordering structure. The system applies linear search (full table scan).

- in this case, using any other access structure is more expensive than full table scan

Second query can get much benefit from using an index such as B^+-tree

Properties of file scans

Full table scan (linear search)

- represents brute force search that can always be executed
- reads a complete relation
- requires minimal number of block accesses to read entire relation
- results are listed in unsorted order as they are accessed in the data file

Index scans

- depend on having an appropriate access path
- can be used for both equality comparisons and to retrieve values in a certain range (e.g. salary between 30000 and 40000)
- might be inefficient and sometimes require even more block accesses than the number of blocks in the original table
  - in this case query planer should reject this method and execute full table scan

Halloween problem

Hallowing problem represents a situation in which an update operation updates a row in database and that changed row can be again visited by the same operation more times

As an example we can consider the following update query where exist a B-tree index on the field Year:

```
UPDATE Student SET Year=Year+1 WHERE Y<3;
```

Query execution can be as follows:

- index is used to access the first field of the relation
- record by record is changed and the index is updated
  - a record that had, for instance, value 1 has now value 2 and the corresponding entry in the updated index is still < 3
- during the further traversal of the index this record will be again accessed and updated
- can happen that all values that had $Year < 3$ end up having $Year = 3$

Halloween problem

Scenario described in the last example resulted in an unwanted state of the database

- modern databases are resistant to the Hallowing problem
- indexes are updated only after all records that has to be changed are changed

Hallowing problem gives an insight to

- aspects that should be considered during the use of index search
- the actual semantics of the update operation

Selection operation

Select operation is basically a search operation retrieving the records in a disk file that satisfy a certain condition

- there are several algorithms used for executing a Selection operation
- **brute force algorithm** is used if there are no access paths which can be used
  - each block is transferred into the buffer pool
  - each record in the buffer pool is checked whether satisfy selection condition
  - brute force can be always used (used when there no better options which can be applied)
- some search algorithms depend on the fact whether files have special access paths (indexes)

Selection operation

- **binary search algorithm** can only be used if the file is ordered according to the attribute in the search condition
  - in the following query, the binary search can only be applied if the file containing the student relation is ordered according to Ssn attribute

```
SELECT * FROM Student WHERE Ssn = 384888448;
```

- **scan using primary index** can be applied if there exist a **primary index** on the attribute in the search condition
  - in the previous example, if there is a primary key on Ssn attribute then that primary index can be mostly used to find the result faster than the binary search
  - primary index can also be used for range queries if the condition is defined on the ordering attribute

```
SELECT * FROM Student WHERE Ssn > 123456789;
```

  - first record has to be found with value $Ssn = 123456789$ and all records afterwards are read subsequently and given as the result

Selection operation

- **scan using clustering index** can be applied if there exist a *clustering index* defined on the attribute in the search condition
  - suppose that there is a clustering index defined on the attribute Year in the Student table

```
SELECT * FROM Student WHERE Year > 1;
```

  - clustering index can be used for these range queries
  - clustering index can also be used for equality conditions
- **scan using hash index** can only be applied for equality conditions when it's defined on a key attribute
  - can be applied for the following query

```
SELECT * FROM Student WHERE Ssn = 123456789;
```

  - but it cannot be used for range queries

Selection operation

- **scan using secondary indexes** (e.g. B-trees) can be applied if there exist a *secondary index* defined on the attribute in the search condition
  - suppose that there exist a secondary index (B^+-tree) defined on the attribute Year in the Student table
  - secondary index and B^+-tree can be used for both, for selection comparison with equality condition and for range queries

```
SELECT * FROM Student WHERE Year = 1;
```

```
SELECT * FROM Student WHERE Year > 1;
```

  - it can happen that brute force algorithm (sequential scan) performs better than scan with secondary index if big part of a table has to be retrieved

Select operation with conjunctive condition

Let's consider the following query with the conjunctive condition

```
SELECT * FROM Student WHERE Year = 2 AND Zip_code = '68163';
```

- if there exists an access path on any of attributes in simple condition than that access path can be used to perform index scan
- once that records are brought into the buffer pool, it is checked whether they satisfy the remaining conditions
- if there is a composite index on all attributes then it is used for both conditions
  - for example, a composite index defined on (Year, Zip_code) in our table Student

Selectivity of a condition

Selectivity of a condition, denoted sl, is defined as the ratio of the number of records (tuples) that satisfy the condition to the total number of records (tuples) in the file (relation)

$$sl = \frac{s}{r(R)}$$

– s - number of selected tuples

– $r(R)$ - number of all tuples in relation R

– selectivity represents percentage of the file that will be retrieved

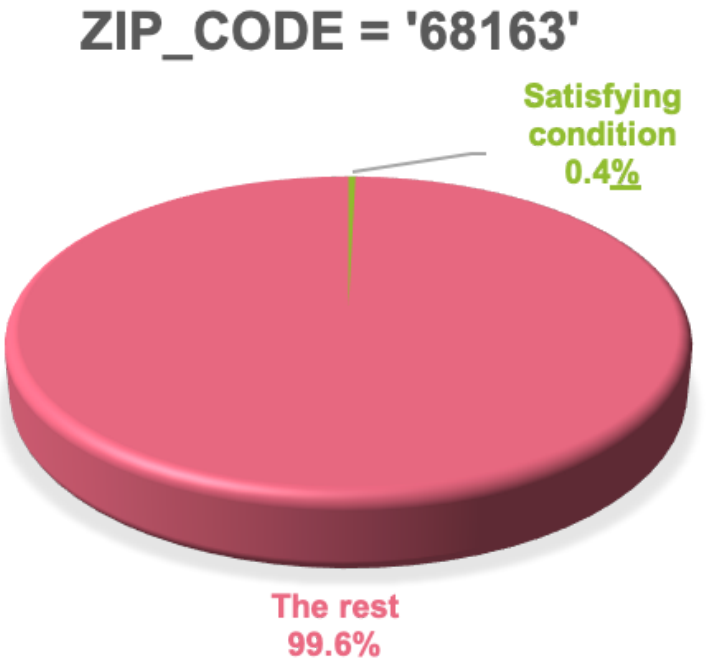
– takes values between 0 and 1

Selection condition in conjunctive queries

Assume that in our table Student

- 400 out of 1000 students satisfy the condition YEAR = 2 and
- 4 out of 1000 students have an address at ZIP_CODE = '68163'

then the following selectivity conditions hold:



Selectivity condition in conjunctive queries

The optimizer should first process conditions with the lower selectivity

– in our example condition Zip_code = ‘68163’ should be first processed

```
SELECT * FROM Student WHERE Year = 2 AND Zip_code = '68163';
```

– once that the Student records are brought in memory, they are checked whether the condition Year = 2 holds

Disjunctive selection conditions

Queries with disjunctive selection conditions are much harder to optimize and process than queries with conjunctive conditions.

- example:

```
SELECT * FROM Student WHERE Year = 2 OR Zip_code = '68163';
```

- result of the query is the union of records satisfying individual conditions
- if all conditions have access paths, they can be used to retrieve records for the individual conditions and then union operation is applied to eliminate duplicates
- if at least one of conditions does not have an access path brute force (linear search) has to be used

Examples of optimization

We can consider again our table *Student(Ssn, Fname, Lname, Year)* which has a B-tree index defined on the attribute Ssn and has no other access paths.

Consider optimization of the following two queries:

- retrieve all attributes of students which has Ssn > “123456789”:

```
SELECT * FROM Student WHERE Ssn > '123456789';
```

  - index will be used if the selectivity of this query is not that high (so that the brute force would be less effective than this index search)

- list all data of students which have the first name ‘John’

```
SELECT * FROM Student WHERE FName = "Smith";
```

  - brute force search is used in this case as the most effective solution (even though there is an index on Ssn it doesn’t help)

Join operations

Join operation is one the most time consuming operations in query processing

The most used type of Join is EQUIJOIN (NATURAL JOIN). We are discussing **two-way joins** - joins involving two files:

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

The most common techniques are:

- **nested loop join**
- **single loop join**
- **sort-merge join**
- **partition-hash join**

Optimization of joins can be very time consuming

Nested-loop join

Nested-loop algorithm is a the default (brute force) algorithm to implement join operation

– for each record r in R (outer loop), the algorithm retrieves every record s in S (inner loop) and checks whether records satisfy the join condition $r[A] = s[B]$

```
foreach r in R:
   foreach s in S:
       if r[A] == s[B]:
           results.add(r.concat(s))
```

– does not require any special access path on either file in the join

– time complexity of the operation is $O(|R| \cdot |S|)$

Single-loop join

Single-loop join uses access structures (e.g., indexes) to retrieve matching records

– suppose that there exist a B-tree or a hash index on attribute B of relation S in our example $R \bowtie_{A=B} S$ then the following algorithm can be applied

```
foreach r in R:
    s  = S.indexLookup(r[A])
    if not is_empty(s) :
        results.add(r.concat(s))
```

– algorithm loops over file R and uses index structure to find matching tuples from file S

– performance is better than of nested-loop join because only single loop is used

Cost estimation: $b_r(t_t + t_s) + n_r \cdot c$

– b_r - number of blocks in relation R

– $t_t + t_s$ - transfer time and seek time

– n_r - number of records in R

– c - cost to search for a tuple in S using the index

Sort-merge join

Sort-merge join can be used in the most efficient way if both files are sorted by value of join attributes A and B.

- both files are scanned concurrently and the records for which holds $r[A] = s[B]$ are matched
- only single pass through the both files is necessary

If the files are not sorted, they have to be sorted using external sorting algorithm and then the sort-merge join can be applied.

One variation of the algorithm is used when there exist secondary indexes on both join attributes, even though the records are not sorted.

- this method may be quiet inefficient

Partition-hash join

Partition hash join relies on partitioning of tables R and S into smaller files

1. in the **partitioning phase** a hashing function h is applied on join attributes of the smaller table (say that R has fewer rows)
   - partitioning is done using one pass through the smaller table and records are assigned to different hash buckets
   - hash bucket corresponds to a single value of $h(A)$
2. in the **probing phase**, the algorithm makes a single pass trough the bigger file (S) and hashes the records applying $h(B)$
   - current record from S is compared with all records from R which belong to the hash bucket corresponding to the value h(B)
     - if values on join attributes match, the rows are concatenated and included in the result
   - algorithm is specially effective when after hashing buckets of the smaller file (R) can fit into main memory

Partition-hash join algorithm

```
foreach r in R:
    i = h(r[A])
    partition[i].add(r)

foreach s in S:
    j = h(s[B])
    if not is_empty(partition[j]):
        foreach rp in partition[j]
            if (rp[A] == s[B])
                results.add(rp.concat(s))
```

Time complexity in the best case: $O(|R| + |S|)$

Review questions

- What is a query tree?
- Why can exist many query trees for a single SQL query?
- What is a range query? Can we use a hash index to execute range queries?
- Explain what is was the issue in the Halloween problem?
- Explain for which types of selection conditions is B-tree applicable.
- Explain the role of selectivity of a condition in queries.
- Why are queries with disjunctive conditions harder to optimize than queries with conjunctive conditions?
- What is the difference between single-loop join and nested-loop join algorithms?
- Explain in your words the process of the partition-hash join.