Data Management for Data Science

DBMS Query evaluation

Maurizio Lenzerini, Riccardo Rosati

Corso di laurea magistrale in Data Science
Sapienza Università di Roma
Academic Year 2015/2016

http://www.dis.uniroma1.it/~rosati/dmds/

SAPIENZA
UNIVERSITÀ DI ROMA

Evaluation of relational algebra operators

Our purpose is to analyze how the SQL engine computes the result of a query.

We will assume that the SQL engine translates the SQL expression into relational algebra. So, we will analyze the methods to evaluate single operators of relational algebra (and then will present query plans combining the evaluation of multiple operators).

We will focus on Selection, Projection and Join.

The evaluation of the selection operation has already been described in the slides on file organizations.

Projection

The main problem with projection is the (possibly requested) elimination of duplicates. Two algorithms can be used to this purpose:

1. Sort-based projection
2. Hash-based projection

We will analyze the first one, which is as follows:

1. Scan the relation R (the relation being projected), and project it writing the result in a temporary file (let T be its number of pages)
2. Order the temporary file, using as ordering key all the attributes (after projection)
3. Scan the ordered temporary file, deleting the contiguous multiple occurrences of the same record, and writing the result in a new file

Cost of (1): $O(M) + O(T)$ dove $T = O(M)$
Cost of (2): $O(T \log_H T)$ (H= number of free buffer slots for the mergesort)
Cost of 3): $O(T)$
Summing up: the total cost is $O(M \log_H M)$

Projection with “index-only scan”

If we have to compute the projection on attributes A1,…,An of relation R, and we have a B+-tree index whose search key includes all the above attributes, then we can execute the so-called “index-only scan”

It consists of scanning the leaves of the index (which is much faster than scanning the data file).

Furthermore, if A1,…,An are a prefix of the search key of the index, then during the scan we can efficiently:

- eliminate the attributes that are projected out
- eliminate the duplicate tuples

Join

We refer to the equi-join on the relations R and S. As a running example, we will assume that:

- R has M pages and p_R tuples per page
- S has N pages and p_S tuples per page

and we will consider the following values: M=1000, N=500, p_R=100, p_S=10, and 10 ms to acess a block in mass memory.

We will analyze the following algorithms:

- Nested loop
- Block nested loop
- Index nested loop
- Sort merge join

Nested loop

One of the two relations is the outer relation (say R), the other one is the inner relation (S).

scan the outer relation R, and:
  for each page P of R:
    scan S, looking for the record that join with the records in P

The cost is $M + (M \times N)$.

With the previous values, the cost of computing the join of R and S through the nested loop is $(1000 + (1000 \times 500)) \times 10\text{ms} = 5.010.000\text{ms}$, i.e., about 1,5 hours.

In general, the smallest relation between R and S should be chosen as the outer relation. If we do that in our example, the cost is 5.005.000ms.

Block nested loop

If the outer relation can fit into the buffer, and there are at least 2 extra free pages, we can load the whole outer relation in the buffer, and use one of the extra buffer pages to read the inner relation one page at the time, and use the remaining free page to store the result (one page at the time).

The cost is M + N (optimal).

In our example, the cost of the block nested loop is 1500 × 10ms, i.e., 15 seconds (1500*10ms=15 seconds).

Block nested loop

In many cases, the outer relation cannot fit into the buffer: we only have G free pages, and so can only devote G-2 buffer pages to load the outer relation, where G-2 is less than the number of pages of the outer relation.

The block nested loop algorithm changes as follows: load the first two G-2 pages of the outer relation in the buffer, scan the inner relation loading one page at the time in the buffere, and for each of such pages compute the join between the tuples in this page with the tuples of the outer relation that are currently in the buffer, writing the result in the remaining extra page.

If the outer relation is R, the cost è $M + (N \times M/(G-2))$. Of course, to make the algorithm effective, G must be sufficiently large.

In our example, assuming G=102, and choosing S as the outer relation, the cost is $(500 + (1000 \times 500/100)) \times 10ms = 55000ms$ (i.e., about a minute).

Index nested loop

The index nested loop algorithm can be executed only if there is an index on the inner relation whose search key is equal to (or contains) the attributes involved in the equi-join.

If the inner relation is S, the algorithm is:

```
scan R, and for each page P of R:
    for each tuple t in P:
        use the index on S to find the tuples of S that are in join with t
```

The cost depends on the type of index and on how many tuples of the inner relation are in join with a tuple of the outer relation. If, in particular, the equi-join attributes are a key for the innre relation S, then at most one tuple of the inner relation is in join with a tuple of the outer relation.

Index nested loop

Assuming that there is an hash index on S (inner relation) and that the equi-joins attributes are a key for S, and assuming 1,2 as the cost for the single access through hash index: since we have 100.000 tuples for R, the cost of the index nested loop algorithm is $1000 + 100.000 \times 1,2 = 121.000$ (about 20 minutes).

Sort merge join

The sort merge join algorithm is as follows:

1 – (if necessary) order (through the merge sort algorithm) R on the equi-join attribute(s)
2 - (if necessary) order (through the merge sort algorithm) S on the equi-join attribute(s)
3 - scan the pages of R (outer relation), and for each tuple of each page of R, look for the tuples that have a join with a tuple in the current page of S (inner relation), exploiting the ordering property.

Cost of steps 1 and 2:

The cost of the merge-sort algorithm in mass memory is $O(P \log_H P)$, where P is the number of pages in the file, and H is the number of free buffer slots that the algorithm can use.

Therefore:

- The cost of step 1 is $M \log_H M$
- The cost of step 2 is $N \log_H N$

Sort merge join

Cost of step 3:

In the worst case (all tuples of R are in join with all tuples of S) the cost of the algorithm is $M \times N$. In the favourable cases, the cost is much smaller. For instance, when the equi-join attributes are a key for the inner relation, then every page of S is analyzed only once, therefore the cost is M + N (optimal). In the average case, even if the same page of S is requested multiple times for the join operation, it is very likely that is still in the buffer, so it is still realistic to assume a cost of M + N.

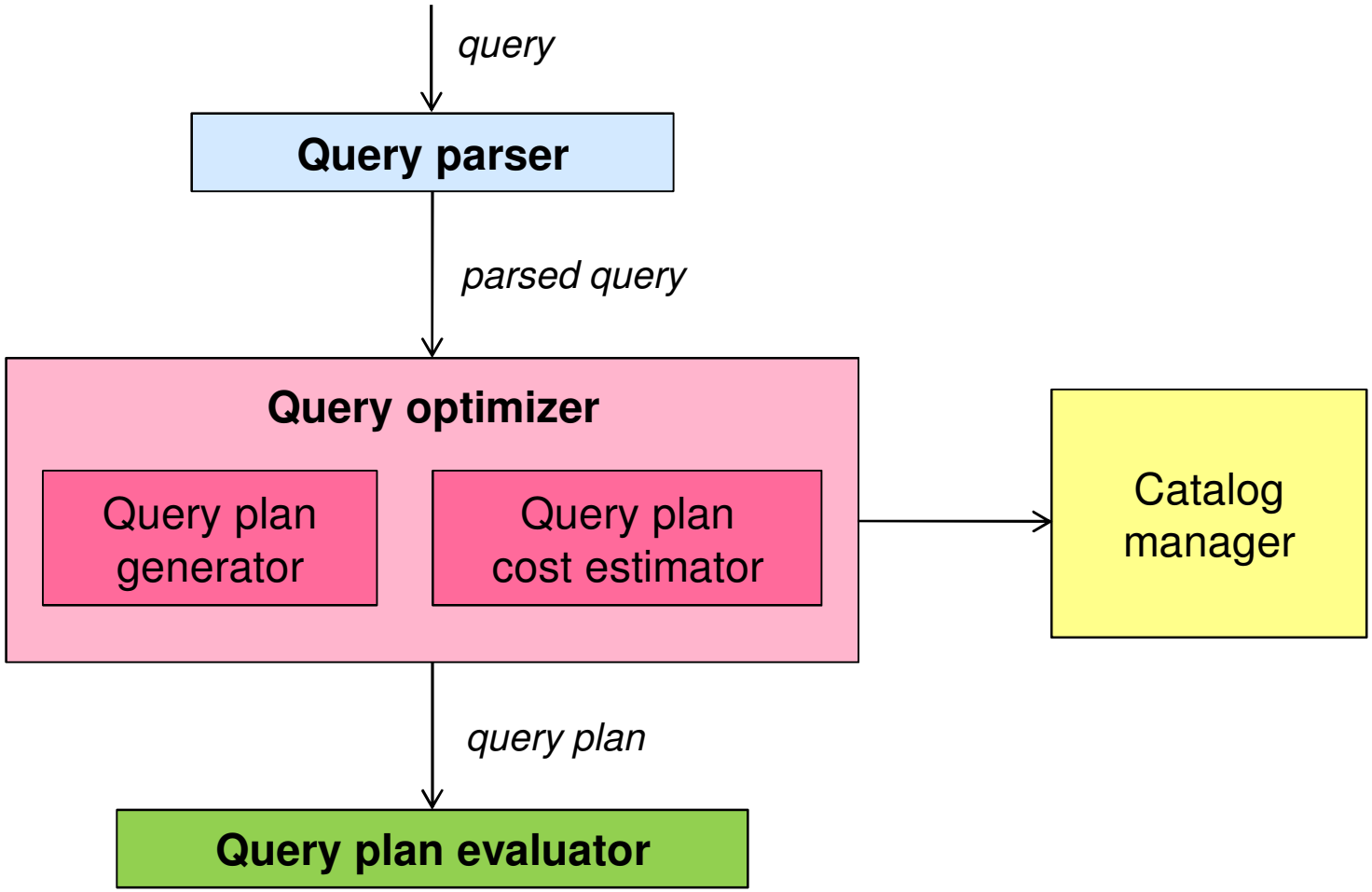
In our case, assuming that the buffer has 10 pages available for the merge sort:

- the cost of ordering R (step 1) is $1000 \log_{10} 1000 = 3000$
- the cost of ordering S (step 2) is $500 \log_{10} 500 = 1500$
- the costo of step 3 is 1000+500= 1500

Therefore, the total cost is 6000×10ms (i.e., one minute).

Query evaluation in a DBMS

Structure of the SQL engine:

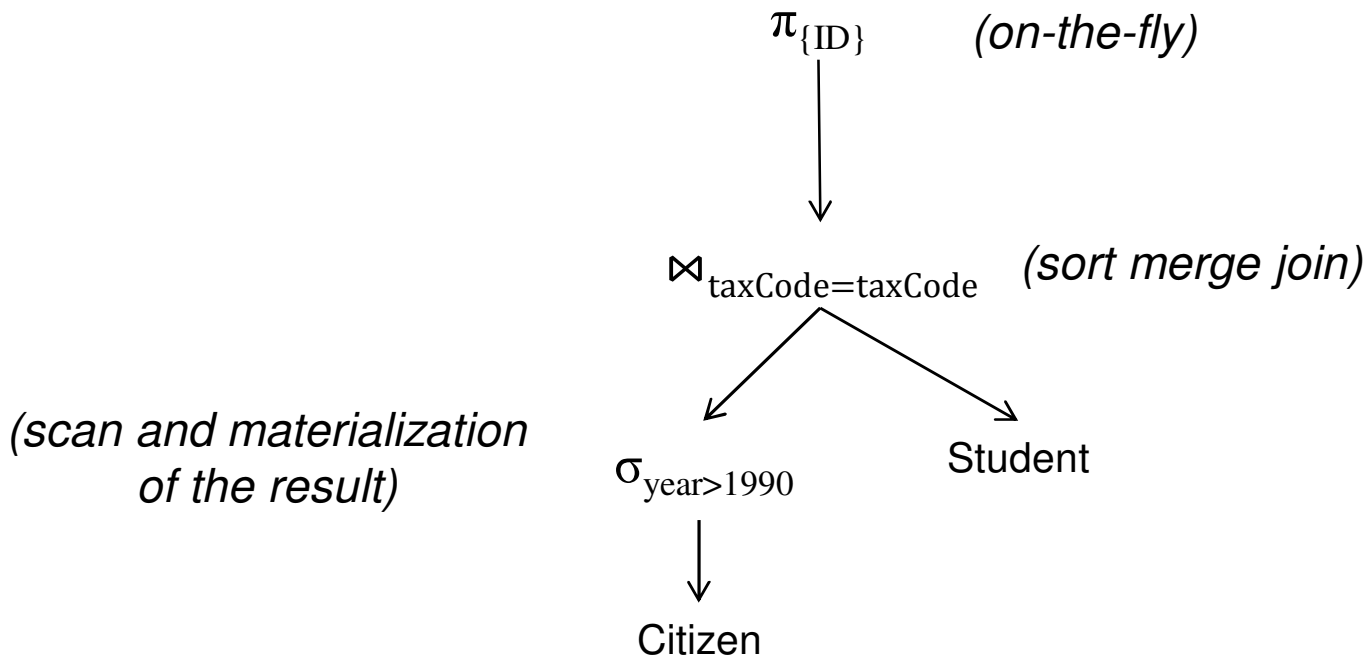


Query optimizer

- The **query optimizer** is the module that defines the query evaluation plan of the SQL query
- A **query evaluation plan** (or simply **query plan**) is a tree whose nodes are relational algebra operators (select, project, join, etc.) annotated by a description of the access methods and the algorithms for executing such an operation
- The query optimizer is constituted by:
  - **Query plan generator**: is the module that generate a set of possible query plans, each of which corresponds to a possible execution plan of the SQL query
  - **Query plan cost estimator**: for every query plan generated by the query plan generator, it computes an estimate of the execution cost of such a plan
- The query optimizer chooses, among all the generated plans, a plan with the minimum cost

Query plan

Example of a query plan:



Query plan

- Example: the following query Q:

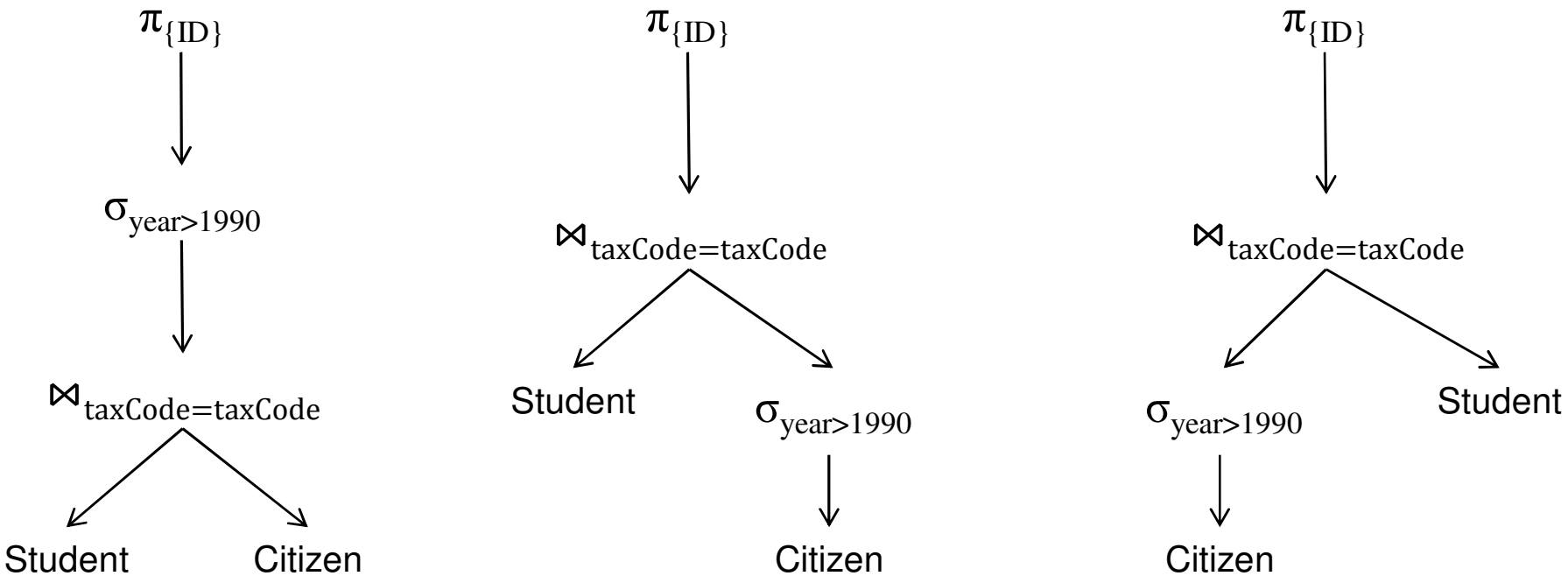
```
SELECT S.ID
FROM Student S, Citizen R
WHERE S.taxCode = R.taxCode AND R.year>1990
```

can be formulated in relational algebra through the expression

$$\pi_{\{ID\}}(\sigma_{year>1990}(Student \bowtie_{taxCode=taxCode} Citizen))$$

Query plan

- Every query plan for Q without a specification of the execution of the single operators corresponds to a relational algebra expression equivalent to Q
- For instance, from $\pi_{\{ID\}}(\sigma_{year>1990}(Student \bowtie_{taxCode=taxCode} Citizen))$, the following trees:



correspond to expressions that are equivalent to the initial one

Partitioning the SQL query

- The query parser partitions the query SQL into **blocks**
- every block is a query SELECT FROM WHERE (possibly with clauses HAVING and GROUP BY) with no nested queries
- A block essentially corresponds to a relational algebra expression that only uses the selection (σ), projection (π), join ($\bowtie$) and cartesian product ($\times$) operators
- Every block is independently processed by the subsequent query optimizer

Query plan generator

- The query plan generator takes as input a block, that is, a select-project-join relational algebra expression, and generates a set of query plans for such an expression
- To reach this goal:
  - It computes **equivalent transformations** of the expression
  - It uses the information about the existing **access methods** of the relations (namely, about the physical organization of the files storing the relations)
- The query plan generator does **not** generate all the possible plans, because the number of such plans is usually really huge (even for relatively simple query)
- The generation of the query plans is restricted by rules that allow for generating only predefined kinds of query plans
- Example of such a rule: every join and cartesian product must have a primitive relation as its inner relation (right child)

Query plan cost estimator

- Is the module that assign a **cost** to every query plan generated by the query plan generator
- The cost is assigned according to a cost model that considers a set of relevant parameters for the real execution cost of the query
- To compute the cost, the query plan cost estimator makes use of the meta-information given by the catalog manager, in particular:
  - Size of every relation (number of records, number of pages)
  - Type and size (number of bytes) of attributes
  - Size of indices (number of pages)
  - For every index, min and max value of the search key that is currently stored
  - etc.

Query plan evaluator

- The query optimizer selects, among all query plans generated by the query plan generator, one having the minimum cost according to the evaluation of the query plan cost estimator
- The query plan evaluator executes such a chosen plan, that is, it executes the tree corresponding to the query plan
- The plan is executed bottom-up and left-to-right (in the order corresponding to a depth-first post-order traversal of the tree): first, the innermost operations are executed (the ones closer to the leaves of the tree) from left to right, slowly going up until the root (which is the last operation executed)
- During the execution, the operations are executed according to the access methods and the algorithms defined in the annotations of the query plan

Relational algebra equivalences

- The query plan generator uses **equivalences between relational algebra expressions** to generate new trees starting from the one corresponding to the initial expression
- Such equivalences are very important in order to produce a query plan with minimum cost
- Such a step of the query plan generator is independent from the access methods of the relations and the available algorithms for executing the operations

Relational algebra equivalences

Examples:

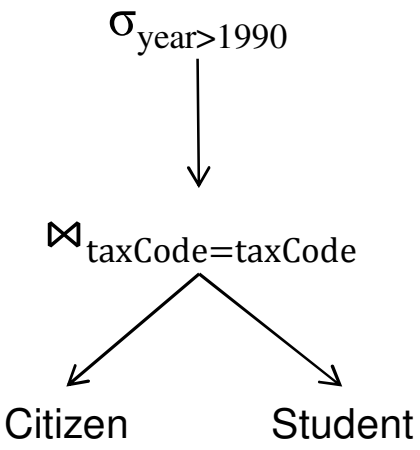
- $R \bowtie_C S \equiv S \bowtie_C R$
- $(R \bowtie_{C1} S) \bowtie_{C2} T \equiv R \bowtie_{C1} (S \bowtie_{C2} T)$
- $R \bowtie_C S \equiv \sigma_C (R \times S)$
- $\sigma_{C1} (\sigma_{C2} (R)) \equiv \sigma_{C2} (\sigma_{C1} (R))$
- $\pi_A(\sigma_C (R)) \equiv \sigma_C(\pi_A(R))$ if condition C only involves attributes of A
- $\pi_A(\sigma_C (R \times S)) \equiv \sigma_C (\pi_A(R) \times \pi_A(S))$ if condition C involves only attributes of A
- $\sigma_{C1 \wedge C2 \wedge C3} (R \bowtie S)) \equiv \sigma_{C3}(\sigma_{C1}(R) \bowtie \sigma_{C2}(S))$ where
  - $C1$ are the conditions involving only attributes of R
  - $C2$ are the conditions involving only attributes of S
  - $C3$ are the conditions involving both attributes of R and attributes of S

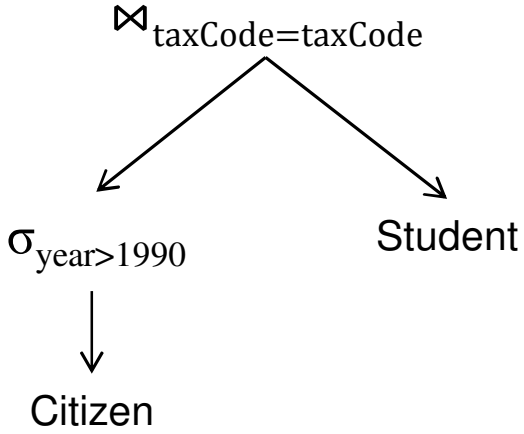
Equivalence-based optimizations

- Main heuristics:
  - **Push-down selections**: in the query plan, execute selections as soon as possible (that is, push selections as down as possible in the plan)
  - **Push-down projections**: in the query plan, execute projections as soon as possible (that is, push projections as down as possible in the plan)
- Both such heuristics are based on the fact that usually the cost of selection and projection operations is smaller than the cost of join operations
- For instance, if a selection and a join must be executed, it is very likely that, by executing the selection operation first, we significantly reduce the size of (at least) one of the two relations on which the join will be executed, which usually implies that the cost of the join operation will be significantly reduced

Push-down selection

- Example:





- Intuitively, the second query plan should be more efficient than the first one, because the join operation is performed after selecting, in the Citizen relation, the records whose year is greater than 1990. Hence, the intermediate relation on which the join is executed could have a size much smaller than the whole Citizen relation.

Limits of heuristics

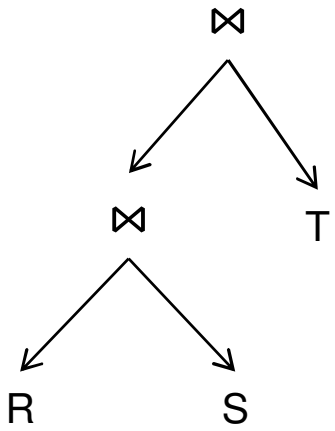
- Such heuristics do not always produce optimal solutions
- Indeed, we should evaluate the consequences of the priorities among operators with respect to the existing access methods of the relations and the available (in the DBMS) algorithms for executing the single operators
- For instance, if in the previous example there exists an index on the Citizen relation with search key taxCode, we could use such an index if we execute the first tree, while we could not use it if we execute the second tree: in many situations, this might imply that the execution of the first tree is more efficient than the execution of the second tree

Generation of intermediate results

- The results of intermediate operations (corresponding to the intermediate nodes in the query plan tree) can be handled in two ways:
  - **materialization of the result**: a temporary DB table is generated, which stores (in mass memory) the intermediate result
  - **no materialization of the result**: the result is not stored in mass memory: as long as it is produced, it is passed to the next operation one chunk at a time (like in an assembly line). In this way, a pipeline of operations is created
- Materialization implies a slower execution of the operation (because of the need of accessing the mass memory)
- On the other hand, not all the operation can be executed without materializing the result
- For instance, if the result must be ordered, then it is necessary to materialize it

Generation of intermediate results

- Example:



- If sort merge join is the only available algorithm for executing the join, then the intermediate result of the R ⋈ S operation must be materialized, because the sort merge join algorithm has to order the relations on which the join is executed (and so the execution of the outermost join requires that the result of R ⋈ S is materialized)