CS525: Advanced Database Organization

Notes 6: Query Processing
Part II: Parsing and pre-processing

Gerald Balekaki

Department of Computer Science

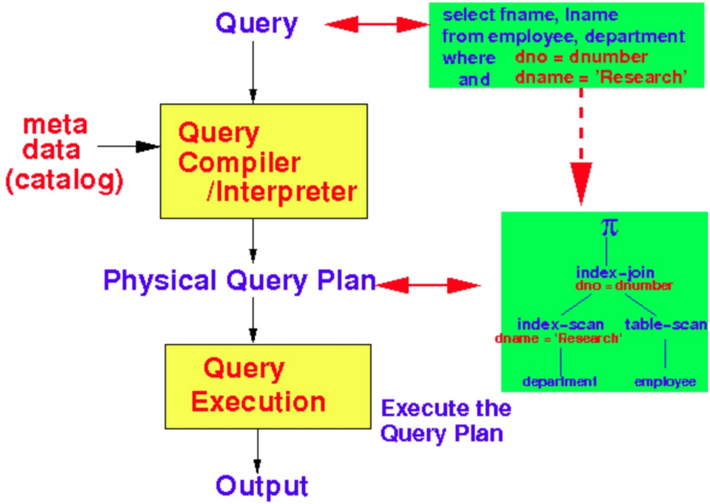
Illinois Institute of Technology

gbalekaki@iit.edu

July 17th 2023

Slides: adapted from a course taught by Shun Yan Cheung, Emory University

Steps needed to process a query (SQL command)



Query

```
select fname, lname
from employee, department
where   dno = dnumber
  and   dname = 'Research'
```

meta data (catalog)

Query Compiler /Interpreter

Physical Query Plan

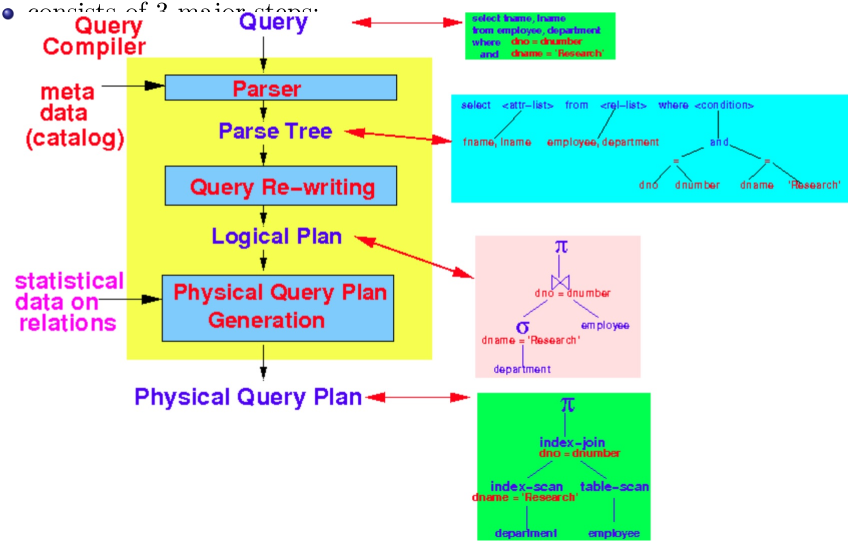
Query Execution

Execute the Query Plan

Output

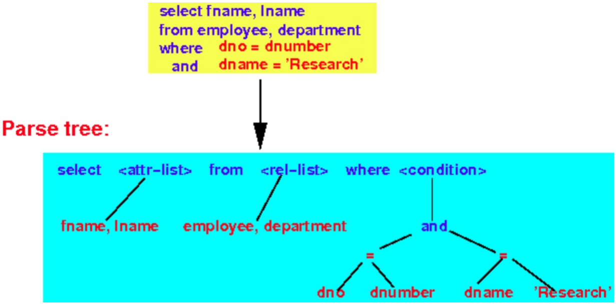
Query Compiler

- consists of 3 major steps:



Parser

- Parses the `SQL command` and constructs a query parse tree that represents the syntax elements in the `SQL command` (Queries need to be translated to an internal form)
  - Queries posed in a declarative DB language (“what should be returned”, not “where is it found”)
  - Queries can be evaluated in different ways

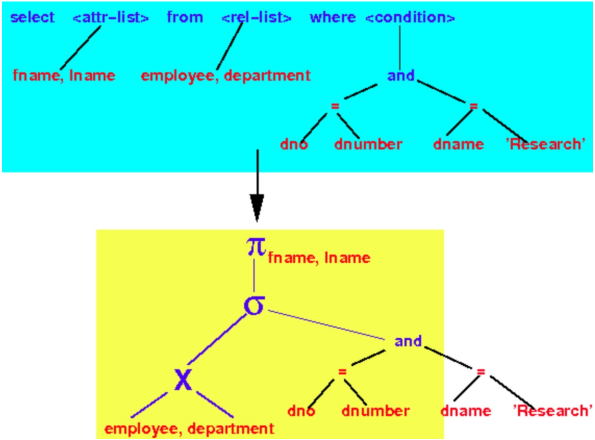


```
select fname, lname
from employee, department
where   dno = dnumber
  and   dname = 'Research'
```

Parse tree:

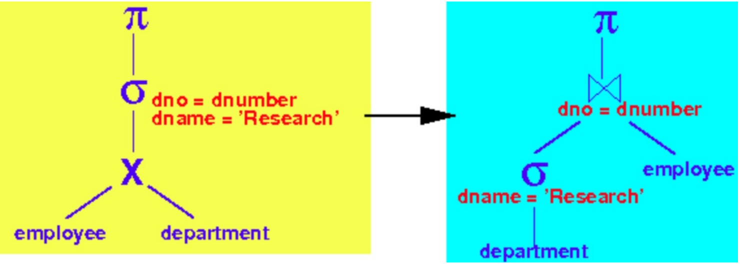
Query Re-writing

1. converts a query parse tree into an un-optimized logical query plan
   - A logical query plan consists of relational algebra operators



Query Re-writing

2. converts the un-optimized logical query plan into an optimized logical query plan



- The optimized logical query plan is a.k.a. the logical query plan
- *The cost measure used to decide on which query plan is better is the size (# tuples) of all the intermediate result relations generated by the logical query plan.*

Logical Query Plan/Physical Query Plan

- Logical Query Plan

  The optimal sequence of ***relational algebra operations*** to perform the query

- Physical Query Plan

  The optimal sequence of ***relational algebra algorithms*** to perform the query

  - Consists of
    - Physical Operators (algorithms used to execute some relational algebra operation, e.g., one-pass join, index-join, etc)
    - The order in which the physical operations are performed (a tree)
    - Way to obtain the input for each physical operator

      - Pipeline (using iterator)

      - Materialization

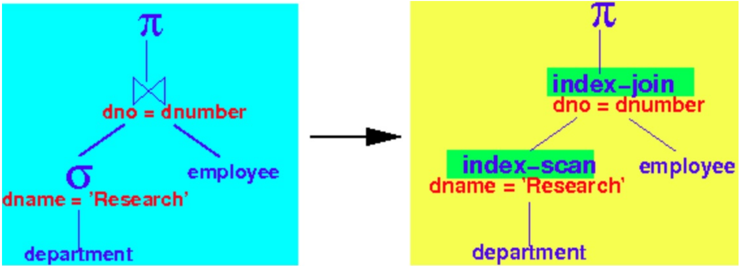
Physical Query Plan Generation

- Select the best algorithm to execute the logical query plan
  - Usually, there are multiple algorithms available to implement one `relational algebra operation`
  - We select the best algorithm depending on
    - Availability of indexes
    - How much main memory is available (# of available buffers) for the algorithm (Fast algorithms require more memory)

- *The cost measure used to decide on which physical query plan is better is the # disk I/O operations used by the physical operator (algorithm).*

Physical Query Plan Generation

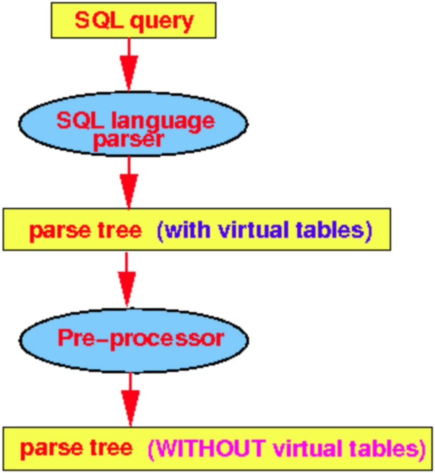
- Select the Physical query operator (algorithm) for each relational algebra operator in the optimal query plan.



SQL Query Parser

- The SQL query parser consists of 2 parts
  - The SQL language parser
    - Parses an `SQL command` into a `parse tree`
  - The SQL pre-processor
    - Checks for some semantic consistencies
    - Replaces `virtual tables (views)` by the corresponding `SQL query` used to obtain the `virtual tables (views)`

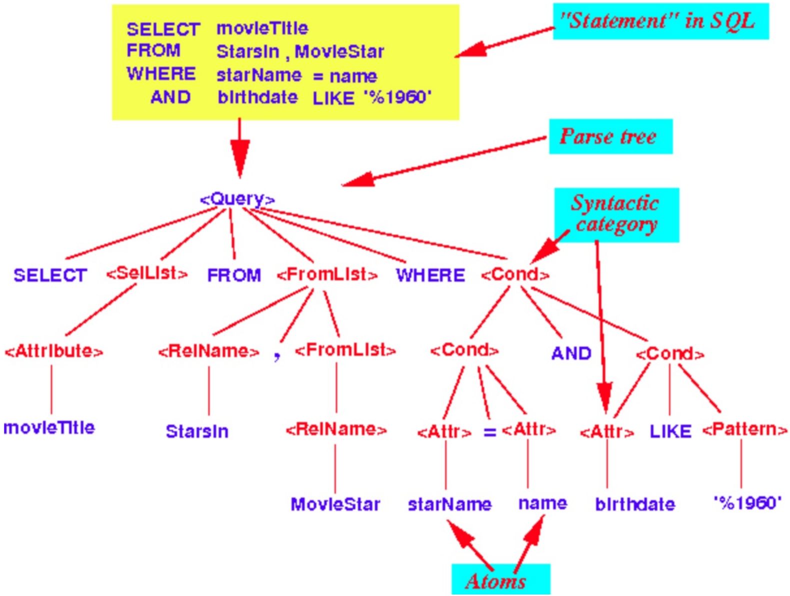
SQL query parser


SQL query
SQL language parser
parse tree (with virtual tables)
Pre–processor
parse tree (WITHOUT virtual tables)

Parser and parse tree

- *Parsing: Converting an (SQL) query into a query parse tree.*
- Parser
    - a computer program that translate statements ("sentences") in a programming language (e.g., SQL) into a parse tree
- Parse tree: a tree whose nodes corresponds to
    - `atoms` of the programming language or
    - `syntactic categories` of the programming language

Example



```
SELECT  movieTitle
FROM    StarsIn , MovieStar
WHERE   starName  = name
  AND   birthdate  LIKE '%1960'
```

Atoms and Syntactic Categories

- Atom
  - Lexical element in a (programming) language that cannot be expressed in more elementary lexical elements
  - *Atoms can not be divided any further*
- Atoms is a.k.a. terminals in a compiler course
- Examples
  - **keywords**: `SELECT, FROM, WHERE, etc`
  - **identifiers**: `employee, name, ...`
  - **Constants**: `3`, 3.14, `'April', ...`
  - **Operators**: `+,` >= `, LIKE, ...`
  - **Tokens**: `(`, `;` , , , ...

Syntactic category

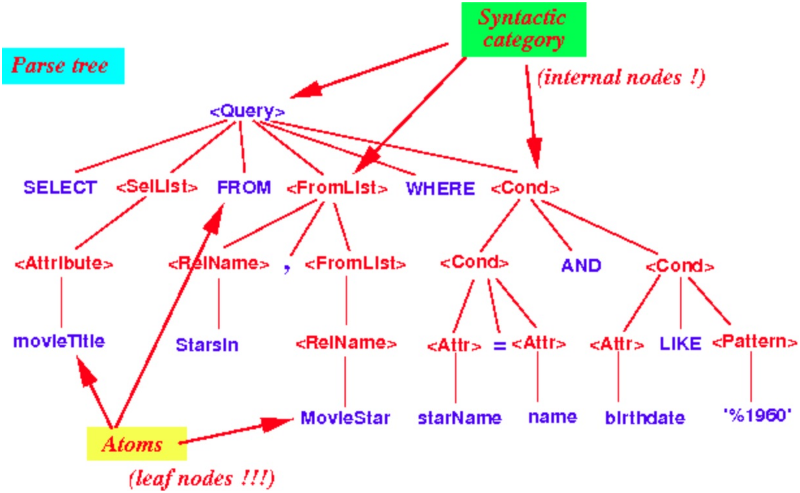
- Lexical construct in a (programming) language that is built up with other lexical elements following some syntactic rules[1].
  - Syntactic categories can be divided further
- A syntactic category is denoted as follows:
  - $<$ `Name-of-a-Syntactic-category` $>$
- Syntactic categories is a.k.a. non-terminals in a compiler course
- Examples of syntactic categories
  - $<$ `Query` $>$
  - $<$ `Arithmetic expression` $>$
  - $<$ `Condition`$>$ `(or Boolean expression)`

[1] *The rules of how to order words help the language parts make sense.*

Properties of a parse tree

- A node in the parse tree is either an atom or syntactic category
- If a node is an atom, then
  - that node does not have any children (i.e.: atoms are always leaf nodes)
- If a node is a syntactic category, then
  - the `subtree` of the node is the instantiation of one of the syntax rules of the grammar

Properties of a parse tree: Example



Grammar of programming languages

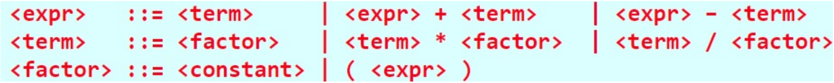
- A grammar is defined by a set of re-writing rules
- A re-writing rule has the following form:

  `<A> ::= Re-write_Rule`

- Meaning:

  <A> can be expressed (replaced by) the right-hand-side (re-write rule)

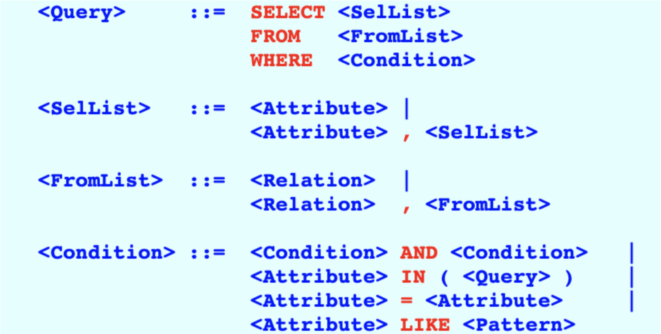
- Example: re-writing rules



```
<expr>   ::= <term>     | <expr> + <term>   | <expr> - <term>
<term>   ::= <factor>   | <term> * <factor> | <term> / <factor>
<factor> ::= <constant> | ( <expr> )
```

A simplified SQL grammar

- To illustrate the translation process from SQL query to logical query plan, we use a simplified SQL grammar



```
<Query>     ::=  SELECT <SelList>
                 FROM   <FromList>
                 WHERE  <Condition>

<SelList>   ::=  <Attribute> |
                 <Attribute> , <SelList>

<FromList>  ::=  <Relation>  |
                 <Relation>  , <FromList>

<Condition> ::=  <Condition> AND <Condition>   |
                 <Attribute> IN ( <Query> )    |
                 <Attribute> = <Attribute>     |
                 <Attribute> LIKE <Pattern>
```

- Note: This is the grammar used by the textbook. It is brief, but incomplete.

“Base” syntactic categories

- There are a number of special syntactic categories in any programming language.
- In `SQL`, these are
  - <`Relation`>
  - <`Attribute`>
  - <`Pattern`>
  - <`Identifier`>
  - <`Constant`>
- Properties
  - These **`syntactic categories`** are not defined using **`grammar rules`**
  - Instead, they are defined by rules about the **`atoms`**
  - Example
    - <**`Identifier`**> must start with letter or _ and followed by letters, digits or _
    - <**`Relation`**> must start with a letter or _ and followed by letters, digits or _
      And it must identify a **`relation`** in the database

Example of parse trees

- Relations used in the example
  - The movie **movieTitle** made in **movieYear** features movie star **starName**

    ```
    StarsIn(movieTitle, movieYear, startName)
    ```

  - The movie star **name** has the specified **address**, **gender** and **birthdate**

    ```
    MovieStar(name, address, gender, birthdate)
    ```

- SQL Query

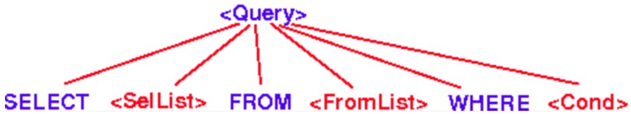
```sql
SELECT  movieTitle
FROM    StarsIn, MovieStar
WHERE   starName = name AND birthdate LIKE '%1960'
```

Example of parse trees

- The parse tree
    - We re-write a Query using this rule:

```
<Query>::=  SELECT <SelList>
            FROM   <FromList>
            WHERE  <Condition>
```

- The parse tree is now

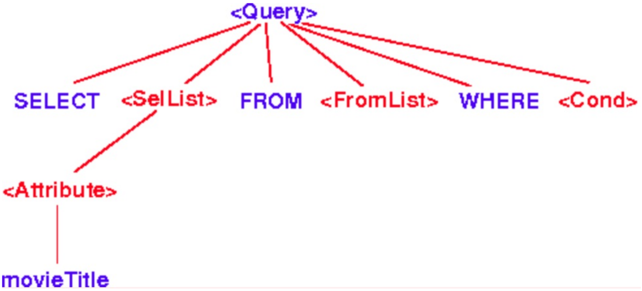


Example of parse trees

- Then we re-write SelList using

```
        <SelList> ::= <Attribute>
                  ::= movieTitle
```

- The parse tree is now

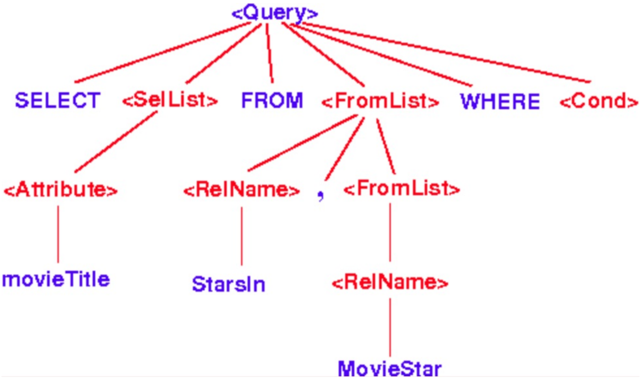


Example of parse trees

- Then we re-write FromList using

```
        <FromList>  ::=  <Relation> , <FromList>
                    ::=  <Relation> , <Relation>
                    ::=   StarsIn , <Relation>
                    ::=   StarsIn , MovieStar
```

- The parse tree is now



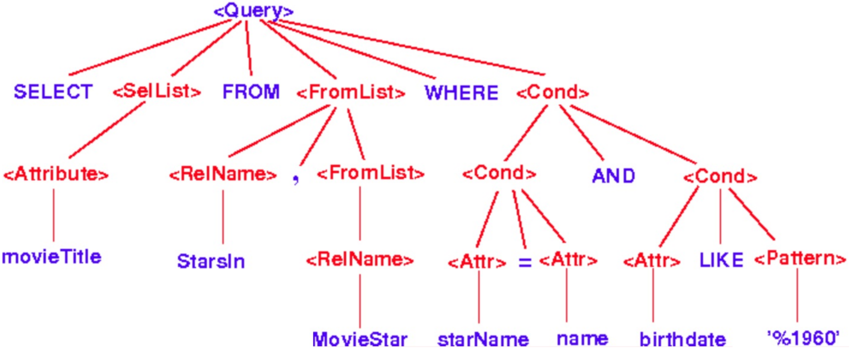
Example of parse trees

- Then we re-write Condition using

```
<Condition> ::= <Condition> AND <Condition>
            ::= <Attribute> = <Attribute>  AND  <Condition>
            ::= <Attribute> = <Attribute>  AND  <Attribute>
                                                LIKE <Pattern>
            ::= starName = name AND birthdate LIKE '%1960'
```

Example of parse trees

- The `parse tree` is



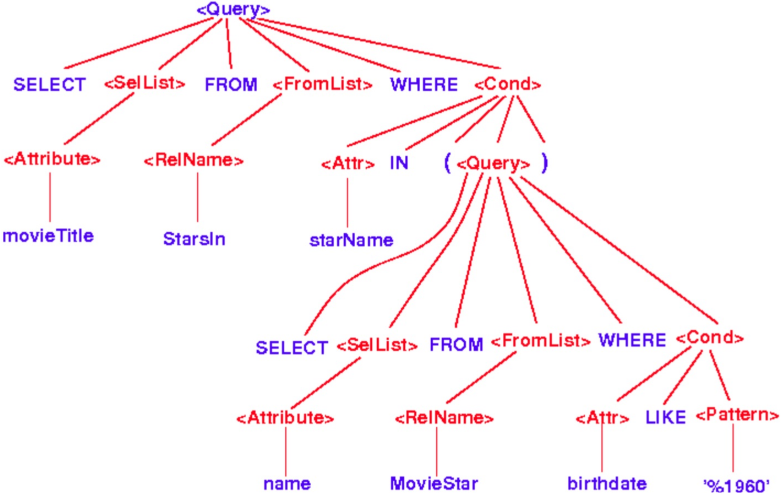
Example 2

- SQL query

```sql
SELECT movieTitle
FROM   StarsIn
WHERE  starName IN (SELECT name
                    FROM   MovieStar
                    WHERE  birthdate LIKE '%1960')
```

Example 2

- The `parse tree` is



Pre-processing an SQL query

- Example of a **query**:

```
SELECT   fname, dno
FROM     employee, department
WHERE    dnumber = dno
```

- Looks correct.
- Query can have semantical errors:
  - Does the database contain the relation **employee** (or **department**)?
  - Does the attribute **dno** (or **fname**) exist in the specified relations (**employee** and **department**)?
    - If it does, which relation does **dno** belong to?
  - And so on

Tasks in Pre-processing an SQL Query

- Check whether the `relations` used in the `FROM` clause exist in the database
- Check and resolve each `attributes` used in the `query`
  - Which `relation` does the `attribute` belong to? (Scope checks)
- Check the `data types` and correct usage of the `attributes`
  - Can some operations be applied to the `attribute`?
    e.g., `+` operation requires a `number` type
- Replace the `virtual relations (views)` by their corresponding `SQL query`

Semantic checks: Example

```
SELECT *
FROM R
WHERE R.a + 3 > 5
```

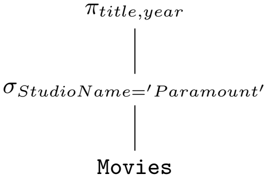
- `Relation R` exists?
- Expand `*`: which attributes in `R`?
- `R.a` is a column?
- Type of constants `3`, `5`?
- `Operator +` for types of `R.a` and `3` exists?
- `Operator` > for types of result of `+` and `5` exists?

Example: virtual relation pre-processing

- `Virtual table` definition

```
        CREATE VIEW  Paramount_Movies AS
             (SELECT  title, year
              FROM    Movies
              WHERE   StudioName = 'Paramount')
```

  - The `SELECT` query is `equivalent` to the following `logical query plan`



$\pi_{title,year}$

$\sigma_{StudioName='Paramount'}$

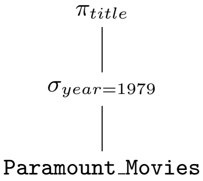
Movies

Example: virtual relation pre-processing

- Consider the following query on the virtual table `Paramount_Movies`:

```
SELECT    title
FROM      Paramount_Movies
WHERE     year = 1979
```

  - The **`Query Processor`** will first **`parse the query`** and create the following **`logical query plan`**



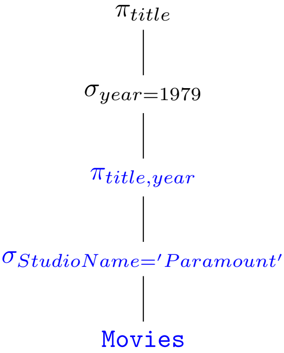
π_{title}

$\sigma_{year=1979}$

`Paramount_Movies`

Example: virtual relation pre-processing

- Then, the `virtual table` is replaced by the corresponding `logical query plan`



π_{title}

$\sigma_{year=1979}$

$\pi_{title,year}$

$\sigma_{StudioName='Paramount'}$

`Movies`

- Next: Convert Parse Tree into initial L.Q.P