ADVANCED DBMS

Unit - 3

Complex Data Types

Object-Based Databases

- Introduction to Object-Relational Database
- Complex Data Types
- Structured Data Types and Inheritance in SQL
- Table Inheritance
- Array
- Query Planning
- Evaluation and Optimization Techniques

Object-Relational Data Models

- Extend the relational data model by including object orientation and constructs to deal with added data types.
- Allow attributes of tuples to have complex types, including non-atomic values such as nested relations.
- Preserve relational foundations, in particular the declarative access to data, while extending modeling power.
- Upward compatibility with existing relational languages.

Complex Types and SQL

- Extensions introduced in SQL:1999 to support complex types:
  - Collection and large object types
    - Nested relations are an example of collection types
  - Structured types
    - Nested record structures like composite attributes
  - Inheritance
  - Object orientation
    - Including object identifiers and references
- Not fully implemented in any database system currently
  - But some features are present in each of the major commercial database systems
    - Read the manual of your database system to see what it supports

Structured Types and Inheritance in SQL

- **Structured types** (a.k.a. **user-defined types**) can be declared and used in SQL

  create type *Name* **as**
  (first*name* **varchar**(20),
  lastname **varchar**(20))
  final

  create type *Address* **as**
  (*street* **varchar**(20),
  city **varchar**(20),
  zipcode **varchar**(20))
  not final

  – Note: **final** and **not final** indicate whether subtypes can be created

- Structured types can be used to create tables with composite attributes

  create table *person* (
  name Name,
  address Address,
  dateOfBirth **date**)

- Dot notation used to reference components: *name.firstname*

Structured Types (cont.)

- **User-defined row types**

```
create type PersonType as (
    name Name,
    address Address,
    dateOfBirth date)
    not final
```

- Can then create a table whose rows are a user-defined type

```
create table customer of PersonType
```

- Alternative using **unnamed row types**.

```
create table person_r(
                name    row(firstname  varchar(20),
                            lastname  varchar(20)),
                addressrow(street      varchar(20),
                           city        varchar(20),
                           zipcode     varchar(20)),
                dateOfBirth date)
```

Methods

- Can add a method declaration with a structured type.

  method *ageOnDate* (*onDate* **date**)

  returns interval year
- Method body is given separately.

  create instance method *ageOnDate* (*onDate* **date**)

  returns interval year

  for *CustomerType*

  begin

  return *onDate* **- self.***dateOfBirth*;

  end
- We can now find the age of each customer:

  select *name.lastname, ageOnDate* (**current_date**)

  from *customer*

Constructor Functions

- **Constructor functions** are used to create values of structured types
- E.g.

```
create function Name(firstname varchar(20), lastname
varchar(20))
returns Name
begin
    set self.firstname = firstname;
    set self.lastname = lastname;
end
```

- To create a value of type *Name,* we use

```
new Name('John', 'Smith')
```

- Normally used in insert statements

```
insert into Person values
     (new Name('John', 'Smith),
      new Address('20 Main St', 'New York', '11001'),
      date '1960-8-22');
```

Type Inheritance

- Suppose that we have the following type definition for people:

```
create type Person
    (name varchar(20),
    address varchar(20))
```

- Using inheritance to define the student and teacher types

```
create type Student
  under Person
  (degree       varchar(20),
   department   varchar(20))
create type Teacher
  under Person
  (salary       integer,
   department   varchar(20))
```

- Subtypes can redefine methods by using **overriding method** in place of **method** in the method declaration

Multiple Type Inheritance

- If our type system supports multiple inheritance, we can define a type for teaching assistant as follows:

  create type *Teaching Assistant*
  under *Student, Teacher*

- To avoid a conflict between the two occurrences of *department* we can rename them

  create type *Teaching Assistant*
  under
  Student **with** (*department* **as** *student_dept*),
  Teacher **with** (*department* **as** *teacher_dept*)

- Each value must have a **most-specific type**

Table Inheritance

- Tables created from subtypes can further be specified as **subtables**
- E.g. **create table** *people* **of** *Person;*
  create table *students* **of** *Student* **under** *people;*
  create table *teachers* **of** *Teacher* **under** *people;*
- Tuples added to a subtable are automatically visible to queries on the supertable
  - E.g. query on *people* also sees *students* and *teachers*.
  - Similarly updates/deletes on *people* also result in updates/deletes on subtables
  - To override this behaviour, use "**only** *people"* in query
- Conceptually, multiple inheritance is possible with tables
  - e.g. *teaching_assistants* under *students* and *teachers*
  - *But is not supported in SQL currently*
    - So we cannot create a person (tuple in *people*) who is both a student and a teacher

Array and Multiset Types in SQL

- Example of array and multiset declaration:

```
create type Publisher as
    (name            varchar(20),
     branch          varchar(20));
create type Book as
    (title           varchar(20),
     author_array    varchar(20) array [10],
     pub_date        date,
     publisher       Publisher,
     keyword-set     varchar(20) multiset);
create table books of Book;
```

Creation of Collection Values

- Array construction
  array [‘Silberschatz’,`Korth’,`Sudarshan’]

- Multisets
  multiset [‘computer’, ‘database’, ‘SQL’]

- To create a tuple of the type defined by the books relation:
  (‘Compilers’, **array**[`Smith’,`Jones’],
  new *Publisher* (`McGraw-Hill’,`New York’),
  multiset [`parsing’,`analysis’])

- To insert the preceding tuple into the relation books
  insert into *books*
  values
  (‘Compilers’, **array**[`Smith’,`Jones’],
  new *Publisher* (`McGraw-Hill’,`New York’),
  multiset [`parsing’,`analysis’]);

Query Processing and Optimization

- A SQL query must be first scanned, parsed and validated.
- The scanner identifies the query tokens (keywords, attribute name and relation name).
- The parser checks for the query syntax.
- The validation process includes that all attribute and relation names are valid and semantically meaningful names in the schema.

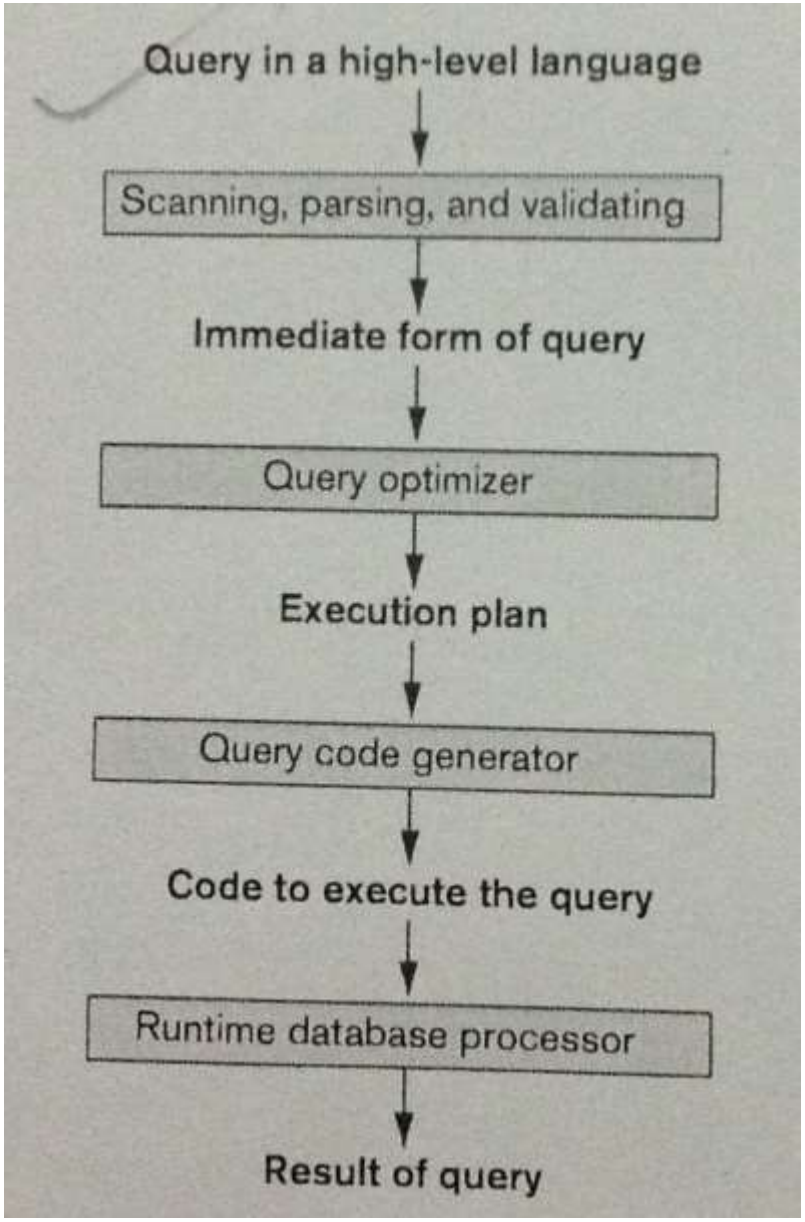
Query Processing and Optimization

- An internal representation of the query is then created as a tree data structure called a query tree.
- It is also possible to represent the query by using a graph data structure called a query graph.

Query Processing and Optimization

- The DBMS must then devise an execution strategy or query plan for retrieving the results.
- The process of choosing the suitable execution strategy is known as query optimization.
- The code generator generates the code to execute the plan.
- The runtime database processor executes the code.

Query Processing and Optimization



Evaluation of Relational algebra expressions

- SQL query is first translated into an equivalent extended relational algebra expression represented as a query tree data structure.
- SQL queries are decomposed into query blocks, which form the basic units that can be translated into the algebraic operators and optimized.
- The query optimizer choose the most suitable execution plan for each query block.

Evaluation of Relational algebra expressions

e.g.

a) select ename, job, salary from empl;

Πename, job, salary(empl)

b) select ename from empl where salary > 20000;

Πename (σ salary>2000(empl))

c) select * from empl where dept_name='Admin' and salary>12000;

σ dept_name="Admin" Λ salary>12000 (empl)

d) select ename,dname from empl,dept where emp.deptno=dept.deptno;

Πename,dname (σ emp.deptno=dept.deptno (emp X dept))