Triggers (VII)

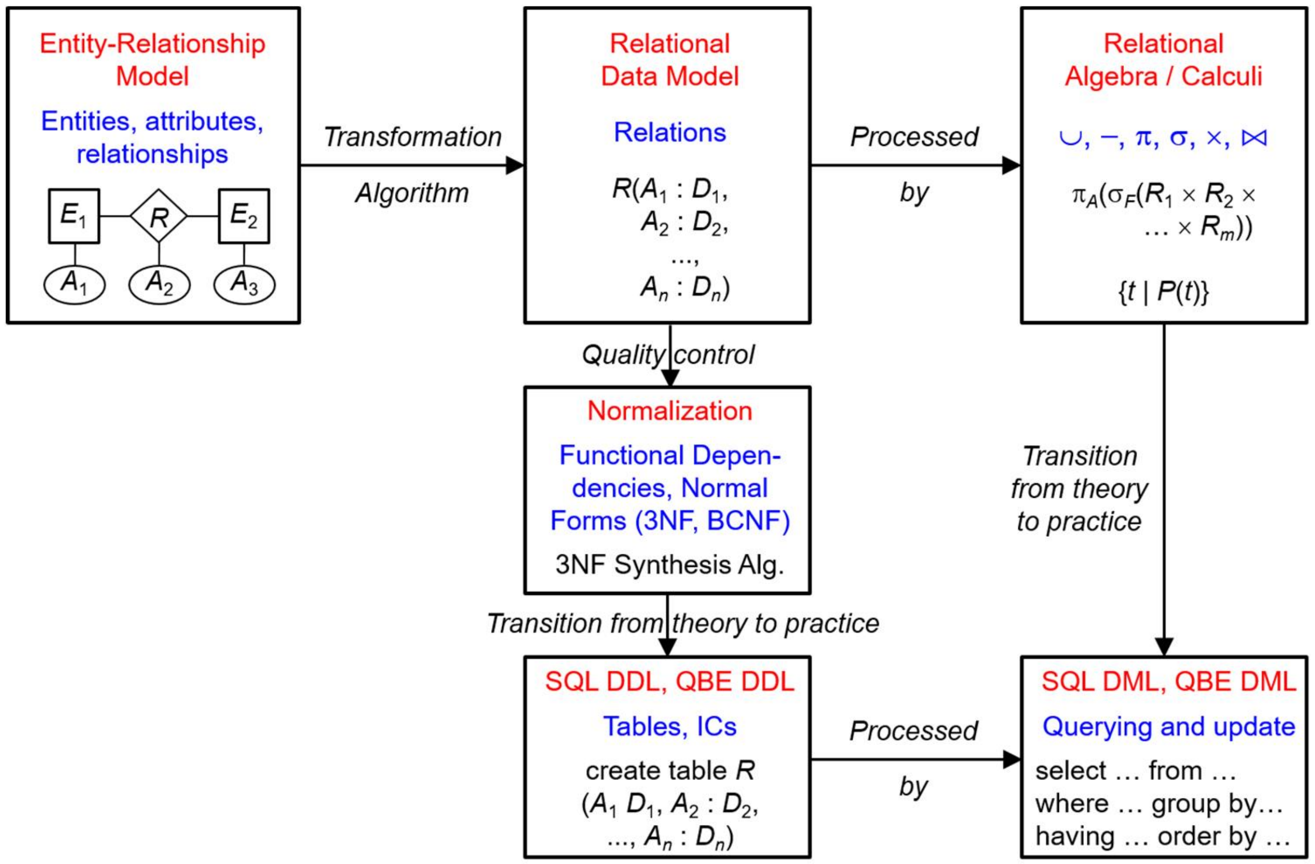
- Advantages of triggers
  - *Improved integrity*: Triggers can be very helpful for implementing integrity constraints that cannot be expressed by built-in SQL integrity constraints
  - *Simplifying modifications*: Changing a trigger requires changing in one place only; all the applications will automatically use the updated trigger
  - *Elimination of redundant code*: Instead of placing a copy of the trigger functionality in every client application that requires it, the trigger is stored only once in the database
  - *Improved processing functionality and power*: Triggers add processing power to the DBMS and to the whole database
- Disadvantages of triggers
  - *Performance overhead*: The management and execution of triggers causes a performance overhead that has to be balanced against the advantages of triggers
  - *Cascading effects*: The action of one trigger can cause another trigger to be fired, and so on, in a cascading manner

Triggers (VIII)

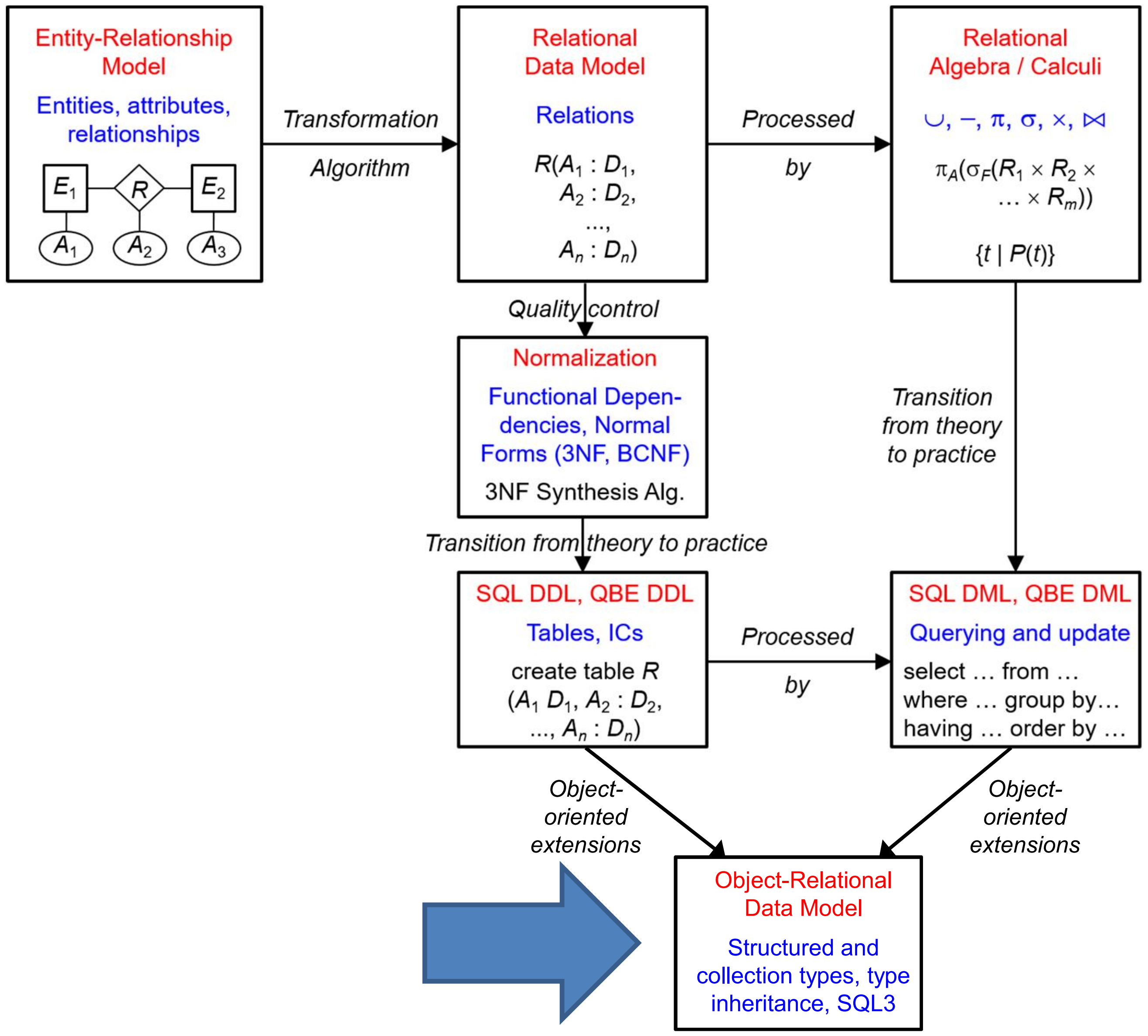
- Disadvantages of triggers (*continued*)
  - *Consistency problem*: Since users write and control triggers, they can design triggers that contradict each other
  - *Possibility of deadlock*: A trigger can activate another trigger which activates another trigger, and so on; this could lead to a cycle of trigger activations and thus a deadlock
  - *Lack of portability*: Although covered by the SQL standard, most DBMS vendors implement their own dialects for triggers, which affects portability
- If a consistency condition can be formulated by a built-in SQL integrity constraint, triggers should not be used

Object-Relational Databases

The Big Picture (I)


Entity-Relationship Model
Entities, attributes, relationships
E_1 R E_2
A_1 A_2 A_3
Transformation Algorithm
Relational Data Model
Relations
$R(A_1 : D_1, A_2 : D_2, \ldots, A_n : D_n)$
Processed by
Relational Algebra / Calculi
$\cup, -, \pi, \sigma, \times, \bowtie$
$\pi_A(\sigma_F(R_1 \times R_2 \times \ldots \times R_m))$
$\{t \mid P(t)\}$
Quality control
Normalization
Functional Dependencies, Normal Forms (3NF, BCNF)
3NF Synthesis Alg.
Transition from theory to practice
Transition from theory to practice
SQL DDL, QBE DDL
Tables, ICs
create table R $(A_1\ D_1, A_2 : D_2, \ldots, A_n : D_n)$
Processed by
SQL DML, QBE DML
Querying and update
select … from … where … group by… having … order by …

The Big Picture (II)


Entity-Relationship Model
Entities, attributes, relationships
E_1 R E_2
A_1 A_2 A_3
Transformation Algorithm
Relational Data Model
Relations
$R(A_1 : D_1, A_2 : D_2, \ldots, A_n : D_n)$
Processed by
Relational Algebra / Calculi
$\cup, -, \pi, \sigma, \times, \bowtie$
$\pi_A(\sigma_F(R_1 \times R_2 \times \ldots \times R_m))$
$\{t \mid P(t)\}$
Quality control
Normalization
Functional Dependencies, Normal Forms (3NF, BCNF)
3NF Synthesis Alg.
Transition from theory to practice
Transition from theory to practice
SQL DDL, QBE DDL
Tables, ICs
create table R $(A_1\ D_1, A_2 : D_2, \ldots, A_n : D_n)$
Processed by
SQL DML, QBE DML
Querying and update
select … from … where … group by… having … order by …
Object-oriented extensions
Object-oriented extensions
Object-Relational Data Model
Structured and collection types, type inheritance, SQL3

Introduction (I)

- Shortcomings of the relational data model from today's perspective
  - Limited type system
    - Atomic SQL data types, 1NF
    - Very well appropriate for administrative, business, and financial applications that process alphanumerical data
  - But advanced applications require
    - structured data types (e.g., an address consists of the zipcode, city, state, house number, street name)
    - type constructors (e.g., for records, nested records, arrays, lists, sets (as they are known in programming languages))

    that are not available in the relational data model
  - Advanced applications require advanced data structures and algorithms

    Examples: Computer-aided design (CAD); digital publishing; image representation for satellite images or weather forecasting; biological and genomic information; spatial, spatiotemporal, and geographical data in maps; audio data; video data

Introduction (II)

  - Differences between SQL types and programming language types: C++ and Java objects as well as structured values (structs, records) have to be flattened into a collection of values of SQL types in order to be stored in the database
  - Limited operations
    - The relational data model has only a fixed set of operations
    - SQL only allows to specify stored procedures and functions

- The object-relational data model extends the relational data model by providing a richer type system including structured data types, type constructors, and object orientation (object identity, inheritance, encapsulation of operations, polymorphism and operator overloading) incorporated in SQL:1999 (SQL3)
- Inheritance: Object types inherit both attributes and methods of previously defined object types
- Polymorphism and operator overloading: Method names can be overloaded to apply to different object types with different implementations
- Encapsulation of operations: Both the object structures and the applicable operations are included in the class/type definitions

Structured Types (I)

- In SQL structured types are called user-defined types (UDTs)
- Methods (procedures, functions) on UDTs are called user-defined functions (UDFs)
- Examples for creating composite attribute types

```
create type Name as
(firstname varchar(20),
 lastname varchar(30))
final;
```

```
create type Address as
(street varchar(20),
 city varchar(30),
 zipcode varchar(9))
not final;
```

- The keyword **final** (**not final**) means that subtypes may not (may) be created from the given type (discussed later)
- Example for using these types to create composite attributes in a table *person*

```
create table person
(pname Name,
 paddress Address,
 dateOfBirth date);
```

Structured Types (II)

- The components of a composite attribute can be accessed by using the “dot” notation, e.g., pname.lastname, paddress.city
- We can also create a table whose rows are of a user-defined type, e.g.,

  create type *PersonType* **as**
  (pname *Name*,
   paddress *Address*,
   dateOfBirth **date**)
  not final;
  create table *person* **of** *PersonType*;

- We can also create distinct types, that is, a new type is the renaming of an existing type, e.g.,

  create type *SizeType* **as integer**;
  create type *LengthType* **as integer**;

  Note that both types are incompatible; hence, their values cannot be compared, and assignments between variables of both types are not possible

Structured Types (III)

- An alternative way to define composite attributes in SQL is to use unnamed row types with an explicit *row* constructor, e.g.

  create table *person_r*
  (pname **row**(firstname **varchar**(20), lastname **varchar**(30)),
  paddress **row**(street **varchar**(20), city **varchar**(30), zipcode **varchar**(9)),
  dateOfBirth **date**);

- Access to composite attributes as well as component attributes of a composite attribute in SQL queries

  select pname, paddress **from** *person*;

  select pname.lastname, paddress.city **from** *person_r*;

- Insertion of tuples into the table *person_r*, e.g.,

  insert into *person_r* **values**(**row**(‘John’, ‘Smith’),
  row(‘’); University Ave’, ‘Gainesville’, ‘32611’),
  date ‘1978-04-25

Structured Types (IV)

- By default, every structured type has a constructor with no arguments, which sets the attributes to their default values
- Any other constructors have to be created explicitly by specifying constructor functions for the structured types
- A function (method) with the same name as the structured type is a constructor method for the structured type, e.g.,

```
create constructor method Name(fname varchar(20), lname varchar(30))
returns Name for Name
begin
    set self.firstname = fname; set self.lastname = lname;
    return self;
end;
```

- Insertion of tuples into the table *person* with the function **new**, e.g.,

```
insert into person values(new Name('John', 'Smith'),
                          new Address('University Ave', 'Gainesville', '32611'),
                          date '1978-04-25');
```

Collection Types (I)

- Collections
  - are type constructors that are used to define collections of other types
  - are used to store multiple values in a single attribute value of a tuple in a table
  - can result in nested tables where an attribute value of a tuple actually contains another table
- Four kinds of collection types
  - Array: One-dimensional array with a maximum number of elements
  - Multiset: Unordered collection of elements that allows duplicates
  - List: Ordered collection of elements that allows duplicates
  - Set: Unordered collection of elements that does not allow duplicates
- Each collection type has a parameter, called the element type, that may be a predefined type, a UDT, a row type, or another collection; it cannot be a reference type (see discussion later) or a UDT containing a reference type
- Each collection must be homogeneous, i.e., all elements must be of the same type (hierarchy)

Collection Types (II)

- Example

```
create type PublisherType as
(name varchar(20),
 branch varchar(20));

create type BookType as
(title varchar(30),
 authors varchar(20) array[10],
 pubDate date,
 publisher PublisherType,
 keywords varchar(20) multiset);

create table books of BookType;
```

- Comments
  - An array, instead of a multiset, is used to store the names of authors since the ordering of authors has some significance generally
  - The ordering of the keywords associated with a book is usually not relevant; each keyword should only appear once; hence, the set constructor is used

Collection Types (III)

- Array collection type
  - An array is an ordered collection of not necessarily distinct values, whose elements are referenced by their ordinal position (index) in the array
  - An array is declared by a data type and optionally a maximum cardinality, e.g.,
    authors **varchar**(20) **array**[10]
  - The array elements can be accessed by an index ranging from 1 to the maximum cardinality, e.g., authors[1], authors[**cardinality**(authors)]
  - The function **cardinality** returns the number of current elements in an array
  - An array type is specified by an array type constructor, which can be defined by enumerating the elements as a comma-separated list enclosed in square brackets or by using an SQL expression with a single attribute table as a result, e.g.,

    array['Silberschatz', 'Korth', 'Sudarshan']

    array(**select** name **from** professors)
  - The data type of the array is determined by the data types of the various array elements

Collection Types (IV)

- Array collection type (*continued*)
  - Insertion of a tuple into the relation *books*

```
insert into books values('Compilers',
                         array['Smith', 'Jones'],
                         new PublisherType('McGraw-Hill', 'New York'),
                         multiset['parsing', 'analysis']);
```

Assumption: Constructor function has been specified for *PublisherType*

Alternatively (by using empty type constructors):

```
insert into books values('Compilers', array[],
    new PublisherType('McGraw-Hill', 'New York'), multiset[]);

update books
    set authors = array['Smith', 'Jones'],
        keywords = multiset['parsing', 'analysis']
    where title = 'Compilers';

update books set authors[2] = 'Jones' where title = 'Compilers';
```

Collection Types (V)

- Array collection type (*continued*)
  - Unnesting of arrays (and multisets) into flat tables
    - An expression evaluating to a collection can appear in its *unnested form* anywhere in a SQL query where a relation name may appear
    - Example: Find the titles of all books that have the word "database" as one of their keywords

      select title **from** books **where** 'database' **in** (**unnest**(keywords));

      Usually, **in** expects a *select-from-where* subexpression
    - The transformation of a collection into a form with fewer (or no) collection-valued attributes is called unnesting (or flattening)
    - Example: Output pairs including the title and author name for each book and each author of the book

      select B.title, A.author

      from books **as** B, **unnest**(B.authors) **as** A(author);

      Since the attribute *authors* of *books* is a collection-valued attribute, **unnest**(B.authors) can be used in the **from** clause, where a relation is expected

Collection Types (VI)

- Array collection type (*continued*)
  - Unnesting of arrays (and multisets) into flat tables (*continued*)
    - Example: Convert the table *books* into a flat table named *flat_books*

```
select title, A.author, publisher.name as pub_name,
          publisher.branch as pub_branch, K.keyword
from books as B, unnest(B.authors) as A(author),
        unnest(B.keywords) as K(keyword);
```

The variable *B* ranges over books; the variable *A* ranges over the authors in the array *authors* for each book *B*; the variable *K* ranges over the keywords in the multiset *keywords* for each book *B*

- Example: The nested relation *books*

title	authors	publisher	keywords
Compilers	[Smith, Jones]	(McGraw-Hill, New York)	[parsing, analysis]
Networks	[Jones, Frick]	(Oxford, London)	[Internet, Web]

is unnested (flattened) into the 1NF relation *flat_books* . . .

Collection Types (VII)

- Array collection type (*continued*)
  - Unnesting of arrays (and multisets) into flat tables (*continued*)
    - Example (*continued*)

title	author	pub_name	pub_branch	keyword
Compilers	Smith	McGraw-Hill	New York	parsing
Compilers	Jones	McGraw-Hill	New York	parsing
Compilers	Smith	McGraw-Hill	New York	analysis
Compilers	Jones	McGraw-Hill	New York	analysis
Networks	Jones	Oxford	London	Internet
Networks	Frick	Oxford	London	Internet
Networks	Jones	Oxford	London	Web
Networks	Frick	Oxford	London	Web

Collection Types (VIII)

- Array collection type (*continued*)
  - Nesting of 1NF relations into arrays (and multisets)
    - Nesting can be performed by an extension of grouping in SQL: Usually, an aggregation function is applied to a collection of values in each group and calculates a single atomic value from them; instead, the **collect** function returns the collection of values as an array or a multiset
    - Example: The following query nests the flat table *flat_books* on the attribute *author*

```
select title, collect(author) as authors,
        PublisherType(pub_name, pub_branch) as publisher, keyword
from flat_books
group by title, pub_name, pub_branch, keyword;
```

title	authors	publisher	keyword
Compilers	[Smith, Jones]	(McGraw-Hill, New York)	parsing
Compilers	[Smith, Jones]	(McGraw-Hill, New York)	analysis
Networks	[Jones, Frick]	(Oxford, London)	Internet
Networks	[Jones, Frick]	(Oxford, London)	Web

Collection Types (IX)

- Multiset collection type
  - A multiset is an unordered collection of elements, all of the same time, with duplicates permitted
  - Since a multiset is unordered, there is no ordinal position to reference individual elements of a multiset
  - Operators are provided to convert a multiset to a table (**unnest**) and a table to a multiset (**multiset**)
  - There is no separate type for sets; instead, a set is a special kind of multiset with no duplicate elements
  - An multiset type is specified by a multiset type constructor, which can be defined by enumerating the elements as a comma-separated list enclosed in square brackets or by using an SQL expression with a single attribute table as a result, e.g.,

    multiset['parsing', 'analysis']

    multiset(**select** room **from** professors)

Collection Types (X)

- Multiset collection type (*continued*)
  - Operations on multisets
    - Function **set** removes duplicates from a multiset to produce a set
    - Function **cardinality** returns the number of current elements
    - Function **element** returns the element of a multiset if it only has one element, or null if the multiset is empty; otherwise, an exception is raised
    - Operation **multiset union** computes the union of two multisets; the keywords **all** or **distinct** can be specified to either retain duplicates or remove them
    - Operation **multiset intersect** computes the intersection of two multisets; the keyword **distinct** removes duplicates; the keyword **all** keeps the minimum number of instances of each value in either operand
    - Operation **multiset except** computes the difference of two multisets; the keyword **distinct** removes duplicates; if a value appears n times in the first operand and the same value appears m times in the second operand with $n > m$, the keyword **all** places this value $n – m$ times in the resulting multiset

Collection Types (XI)

- Multiset collection type (*continued*)
  - Example

    create table R1(A1 **integer**, A2 **double multiset**);

    create table R2(B1 **integer**, B2 **double multiset**);

    select A2 **multiset intersect** B2 **from** R1, R2 **where** A1 = B1;

    select A2 **multiset intersect all** B2 **from** R1, R2 **where** A1 = B1; -- default

    select A2 **multiset intersect distinct** B2 **from** R1, R2 **where** A1 = B1;
  - Unnesting and nesting of multisets
    - Use the function **unnest** for unnesting and the aggregation function **collect** with grouping for nesting (see discussion of the **array** type)
  - Special aggregate functions for multisets
    - Function **fusion** creates the multiset union of all multisets in a group
    - Function **intersection** creates the multiset intersection of all multisets in a group
  - Predicates on multisets
    - **distinct**, **member**, **submultiset**, **is a set** / **is not a set**

Object Identity and Reference Types (I)

- Object identity is the feature of an object that never changes and that distinguishes the object from all other objects
- An object's identity is independent of its name, structure, and location, and persists even after the object has been deleted so that it may never be confused with the identity of any other object
- Example
  - We had: **create table** *person* **of** *PersonType*;
  - To obtain object identifiers for each tuple of *person*, we write

    create table *person* **of** *PersonType*

    ref is person_id **system generated**;

    The object identifiers are system generated, i.e., generated automatically by the DBMS
- Other objects can use an object's identity as a unique way of referencing it
- Example
  - **create type** *DepartmentType*(name **varchar**(20),

    head **ref**(*PersonType*) **scope** *person*);

Object Identity and Reference Types (II)

- Example (*continued*)
  - **create table** *department* **of** *DepartmentType*;
  - References are restricted to tuples of the table *person*
  - The restriction of the scope of a reference to tuples of a table is mandatory
  - References behave like foreign keys
- Assigning a value to a reference attribute
  - Problem: The user does not know the object identifier explicitly
  - Solution: Create a tuple with a *null* reference first, then update the reference separately by an update command and an SQL query
- Example
  - **insert into** *department* **values** ('CS', **null**);

```
update department
        set head = (select p.person_id
                    from person as p
                    where pname = 'Jones');
where name = 'CS';
```

Object Identity and Reference Types (III)

- User generated references
  - Users are allowed to generate object identifiers with the **ref using** clause
  - Example

    create type *PersonType2* **as**
    (name **varchar**(20),
     address **varchar**(50))
    ref using varchar(10);

    create table *person2* **of** *PersonType2*
    ref is person_id2 **user generated**;
  - When inserting a tuple into the table *person2*, the user must provide a value for the object identifier that must be unique in *person2*

    insert into *person2*(person_id2, name, address)
    values (‘987123654’, ‘Jones’, ’23 Waterloo Rd, Gainesville, FL 32611’);
  - Inserting a tuple into a referencing table

    insert into *department* **values** (‘CS’, ‘987123654’);

Object Identity and Reference Types (IV)

- Derived references
  - Using an existing primary-key value as the object identifier with the **ref from** clause
  - Example

    create type *PersonType3* **as**
    (name **varchar**(20) **primary key**,
     address **varchar**(50))
    ref from(name);

    create table *person3* **of** *PersonType3*
    ref is person_id3 **derived**;
  - Inserting a tuple into a referencing table

    insert into *department* **values** (‘CS’, ‘Jones’);
- References are dereferenced by the **->** symbol or by the **deref** operator
  - Example of path expressions (Note: This is an implicit, hidden join operation)

    select head**->**address or **select deref**(head).address
    from *department* **where** head**->**name = ‘Jones’;

Operations on Object-Relational Data (I)

- A structured type can have methods defined on it; methods are part of the type definition of a structured type
- Example: Type definition including method declaration

```
create type PersonType as
    (pname Name, paddress Address, dateOfBirth date) instantiable not final
 instance method ageOnDate(onDate date) returns interval year;
```

The keyword **instantiable** indicates that instances can be created for this type; if **not instantiable** had been specified, we would not be able to create instances of this type but only from one of its subtypes

The method body is created separately

```
create instance method ageOnDate(onDate date) returns interval year
for PersonType
begin return onDate – self.dateOfBirth; end;
```

The **for** clause indicates which type this method is for; the keyword **instance** indicates that this method executes on an instance of the type *PersonType*; the variable **self** refers to the *PersonType* instance on which the method is invoked

Operations on Object-Relational Data (II)

- Methods can be invoked on instances of a type
- Example: Find the age of each person

  select pname.lastname, *ageOnDate*(**current_date**) **from** person;

- Constructor functions and the **new** expression to invoke either the system-supplied or a user-defined constructor function have been discussed before
- Each UDT has an implicitly defined observer method for each attribute of that UDT; the name of the observer method for an attribute *x* is *x*()
- Example

  select b.pubdate() **from** books **as** b **where** b.title() = 'Compilers';

  Tuple variable *b* is needed since we need a variable whose value is an object of type *BookType*, which is the UDT for relation *books*

  SQL allows the syntactical removal of the empty parenthesis but the tuple variable is still needed

  select b.pubdate **from** books **as** b **where** b.title = 'Compilers';

Type Inheritance and Function Overloading (I)

- Assume we have the type

```
create type PersonType4 as
(name varchar(20),
 address varchar(50))
not final;
```

We may want to store extra information about persons who are students and about persons who are teachers; we use inheritance expressed by the keyword **under** to define a student type and a teacher type

```
create type StudentType             create type TeacherType
    under PersonType4 as                under PersonType4 as
    (degree varchar(20),                (salary integer,
     department varchar(20))             department varchar(20));
    final;                              not final;
```

Both types *Studenttype* and *TeacherType* inherit the attributes of type *PersonType4* and have own attributes; *Studenttype* and *TeacherType* are said to be subtypes of *PersonType4*; *PersonType4*; is said to be a supertype of *Studenttype* and *TeacherType*

Type Inheritance and Function Overloading (II)

- The order of supertypes in the **under** clauses determines the inheritance hierarchy
- An instance of a subtype can be used in every context in which a supertype instance is used
- Methods of a structured type are inherited by its subtypes, just as attributes are
- A subtype can redefine the effect of a method by declaring the method again, using **overriding method** in place of **method** in the method declaration; the signature of the method must be the same
- Example

```
create type StudentType2 under PersonType as
    (degree varchar(20), department varchar(20)) instantiable not final
overriding instance method ageOnDate(onDate date) returns interval year;

create instance method ageOnDate(onDate date) returns interval year
for StudentType2
begin return onDate + 1 – self.dateOfBirth; end; -- different implementation
```

Table Inheritance (I)

- It is possible to define subtables of a table and to inherit the properties of the table to its subtables
- Example

  create table *person4* **of** *PersonType4*;

  create table *student* **of** *StudentType* **under** *person4*;

  create table *teacher* **of** *TeacherType* **under** *person4*;
- The types of the subtables (*StudentType* and *TeacherType*) are subtypes of the type of the parent table (*PersonType4*)
- Every attribute present in the parent table (supertable) (*person4*) is also present in its subtables (*student* and *teacher*)
- Every tuple present in the subtables (*student* or *teacher*) becomes implicitly present in the parent table (*person4*), i.e., if a query uses the parent table (*person4*), it will not only find the tuples directly inserted into that table but also tuples inserted into its subtables (*student* and *teacher*); however, only those attributes present in the parent table (*person4*) can be accessed by that query

Table Inheritance (II)

- If only tuples from the parent table (*person4*) but not from its subtables are to be retrieved, "**only** *person4*" instead of "*person4*" is used in a query

- The **only** keyword can also be used in delete and update statements

- Without the **only** keyword, a delete statement on a supertable also deletes tuples that were originally inserted in its subtables

- Example

  delete from *person4* **where** *P*;

  deletes all tuples from the tables *person4*, *student*, and *teacher* that satisfy *P*

  delete from only *person4* **where** *P*;

  does not affect the tuples inserted into the tables *student* and *teacher*

- The semantics maintained between supertable and subtable is that of *containment*: A tuple in a subtable is (at least conceptually) "contained" in its supertable

Table Inheritance (III)

- This has impact (at least conceptually) on data manipulation commands for insert, update, and delete to maintain consistency
  - When a tuple is inserted into a subtable, then the values of all inherited attributes of the table are inserted into the corresponding supertables, cascading upwards in the table hierarchy

    Example: If we insert a tuple into the table *student*, the values of its inherited attributes are inserted into the table *person4*
  - When a tuple is updated in a subtable, the modified values of inherited attributes will be propagated to the corresponding tuples in supertables
  - When a tuple is updated in a supertable, then the values of all inherited attributes in all corresponding tuples of its direct and indirect subtables are also updated accordingly; as the supertable may itself be a subtable, the previous condition will also have to be applied to ensure consistency
  - When a tuple is deleted in a subtable or a supertable, the corresponding tuples in the table hierarchy are deleted

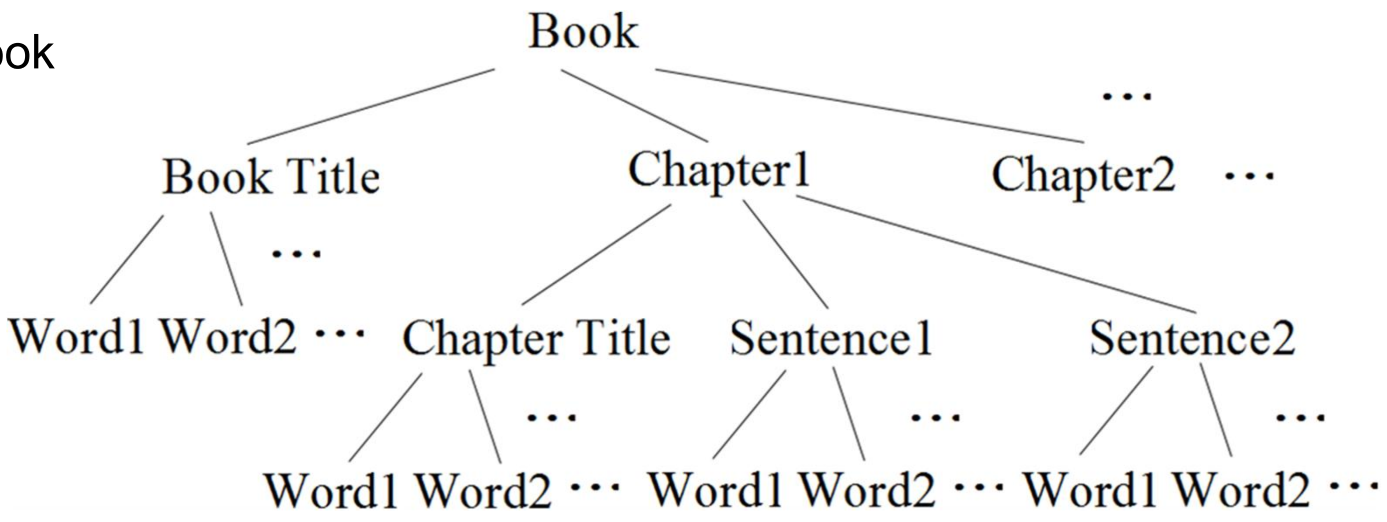
    Example: If we delete a tuple of the table *student*, the corresponding tuple of the table *person4* is deleted

Complex Objects (I)

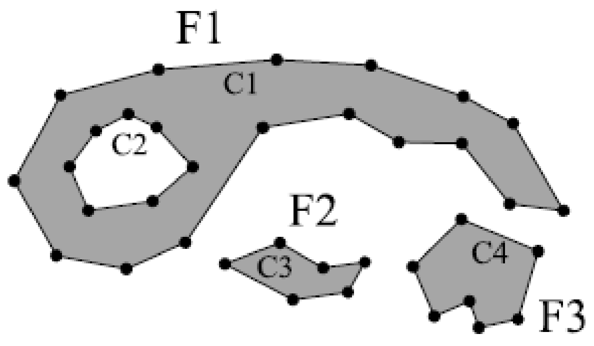
- Big Data
  - Loosely defined term that refers to extremely large, diverse, and distributed sets of alphanumerical and simply structured data that are difficult, or even impossible, to capture, store, manage, query, and analyze with conventional database management techniques and tools
  - For example, generated as raw data from sensing devices like instruments, sensors, satellites, mobile devices, cameras, microphones, radio-frequency identification readers, etc.
  - Representation and processing strategy: distributed, decentralized
- Big Objects
  - Individual large, highly structured, complex (application) objects of varying representation size that are handled as monolithic, self-contained entities (or objects in the object-oriented sense)
  - Complex objects are implemented as values of abstract data types (ADTs)
  - For example, images, videos, multimedia documents, hurricanes, genes, enzymes, proteins, spatial objects (point, line, region, map)
  - Representation and processing strategy: compact, centralized

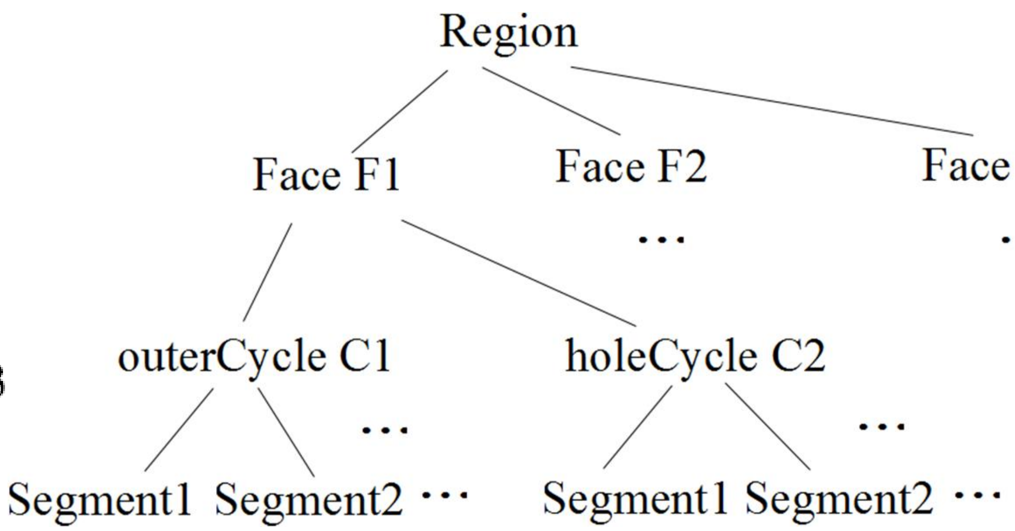
Complex Objects (II)

- Example: A book



- Example 2: A spatial region object



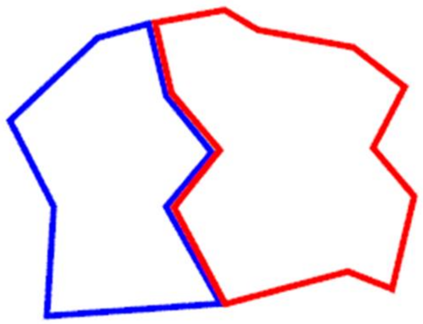


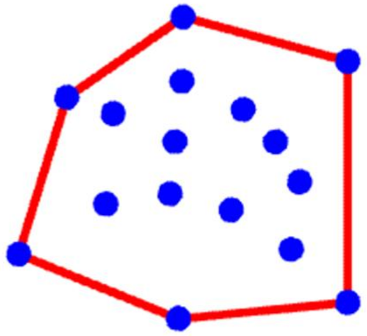
Complex Objects (III)

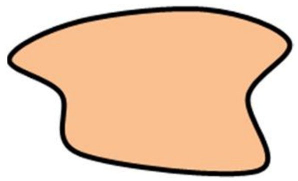
- The concepts we have learned so far make it very difficult, or even impossible, to adequately represent complex objects
- Two main reasons
  - SQL3 does not support *pure* ADTs in SQL: The internal structure is always visible and accessible (ADTs only allows access by means of operations)
  - All objects one can construct with SQL3 constructs have a relatively small and limited representation size (but videos, e.g., can have a size of several GB)
- Many ORDBMs provide the user with *vendor-specific* packages of ADTs for special purposes, e.g.,
  - Informix DataBlades for text, images, video, spatial, spatiotemporal, web, and time series data
  - Oracle Data Cartridges for multimedia, text, image, audio, video, spatial (Oracle Spatial), legal, and medical data
  - DB2 Extenders in DB2 for text, spatial, net search, and XML data
- No or very limited standardization for these extension packages

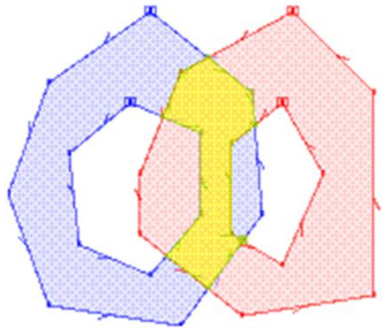
Complex Objects (IV)

- Example: Some geometric operations on the spatial data types *point*, *line*, and *region* that are implemented as ADTs whose internal structure is hidden
  - *distance*: $\alpha \times \beta \rightarrow$ *double* for $\alpha, \beta \in$ {*point*, *line*, *region*}
  - *meet, overlap*: $\alpha \times \beta \rightarrow$ *bool* for $\alpha, \beta \in$ {*line*, *region*}
  - *inside*: $\alpha \times$ *region* $\rightarrow$ *bool* for $\alpha \in$ {*point*, *line*, *region*}
  - *convexHull*: *point* $\rightarrow$ *region*
  - *intersection*: $\alpha \times \beta \rightarrow$ *line* for $\alpha, \beta \in$ {*line*, *region*}, $\alpha \neq \beta$
  - *mbb*: $\alpha \rightarrow$ *region* for $\alpha \in$ {*point*, *line*, *region*}
  - *intersection*: *region* $\times$ *region* $\rightarrow$ *region*


Meet predicate yields *true* for two *region* objects


Convex hull of a *point* object


Minimum bounding box of a *region* object


Intersection of two *region* objects

Complex Objects (V)

- Query example with spatial data types and spatial operations

  Consider the map of the 50 states of the USA. Each state has besides its thematic attributes like name and population also a geometry which is described by its region. A region can have holes (like enclaves) and consist of several components (like mainland and islands). Cities can, e.g., be represented as points indicating their location.

  We create two tables:

  create table *states*(sname **varchar**(30), spop **integer**, territory **region**);

  create table *cities* (cname **varchar**(20), cpop **integer**, loc **point**);

  The spatial data types *point* and *region* are in the same way attribute data types as the types *string* and *integer*

  Query: Determine the names of all cities and states where a city is located in a state; this can then be formulated as a so-called spatial join:

  select cname, sname
  from cities, states
  where loc inside territory

Complex Objects (VI)

- Query example with spatial data types and spatial operations (*continued*)

  The term *inside* is a so-called topological predicate testing whether a point is geometrically located inside a region

  Query: List the names and area of all states ordered by their geometric extent:

  select sname, area(territory) **as** sarea
  from states
  order by area(territory) **desc**;

  The operator *area*: *region* → *double* is a geometric function taking a region object as an argument and yielding a numerical value, namely the area of this object, as a result (area here in square miles)

sname	sarea	sname	sarea	sname	sarea	sname	sarea	sname	sarea
Alaska	665,384.04	Michigan	96,713.51	Missouri	69,706.99	Louisiana	52,378.13	West Virginia	24,230.04
Texas	268,596.46	Minnesota	86,935.83	Florida	65,757.70	Mississippi	48,431.78	Maryland	12,405.93
California	163,694.74	Utah	84,896.88	Wisconsin	65,496.38	Pennsylvania	46,054.35	Hawaii	10,931.72
Montana	147,039.71	Idaho	83,568.95	Georgia	59,425.15	Ohio	44,825.58	Massachusetts	10,554.39
New Mexico	121,590.30	Kansas	82,278.36	Illinois	57,913.55	Virginia	42,774.93	Vermont	9,616.36
Arizona	113,990.30	Nebraska	77,347.81	Iowa	56,272.81	Tennessee	42,144.25	New Hampshire	9,349.16
Nevada	110,571.82	South Dakota	77,115.68	New York	54,554.98	Kentucky	40,407.80	New Jersey	8,722.58
Colorado	104,093.67	Washington	71,297.95	North Carolina	53,819.16	Indiana	36,419.55	Connecticut	5,543.41
Oregon	98,378.54	North Dakota	70,698.32	Arkansas	53,178.55	Maine	35,379.74	Delaware	2,488.72
Wyoming	97,813.01	Oklahoma	69,898.87	Alabama	52,420.07	South Carolina	32,020.49	Rhode Island	1,544.89

Complex Objects (VII)

- Example: Some operations for processing images
  - Data type *image* offered with a large variety of standard storage formats, e.g., support of the formats tiff, gif, jpeg, photoCD, group 4, fax
  - Some operations

    rotate: *image* × *angle* → *image*

    crop: *image* × *polygon* → *image*

    flip: *image* × *image* → *image*

    minus: *image* × *image* → *image*

    intersection: *image* × *image* → *image*

    union: *image* × *image* → *image*

    superimposition: *image* × *image* → *image*