CITY UNIVERSITY OF LONDON EST 1894

Academic excellence for business and the professions

CSR Building confidence in a computerised world
www.csr.city.ac.uk

Object-Relational Database Management Systems

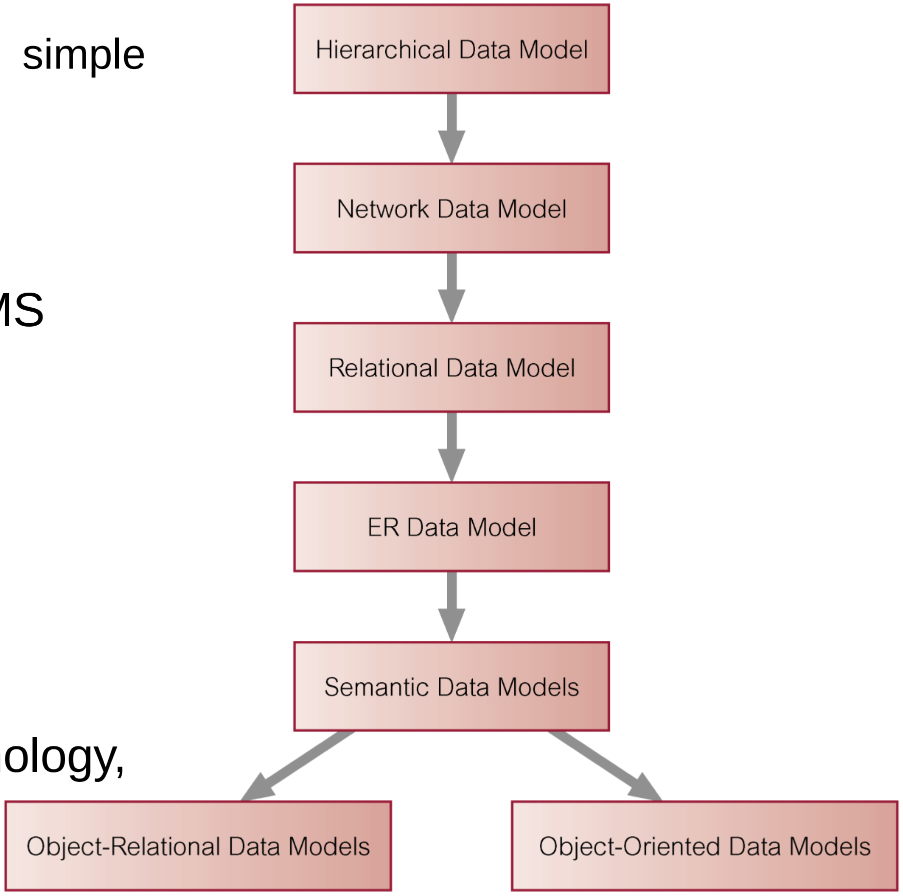
Overview

- Intro
  - Requirements for advanced database applications.
  - Why RDBMSs are not well suited to supporting such applications.
    - Problems associated with storing objects in a Relational DB.
- The main material
  - Object-oriented features in the SQL standard (SQL:2011)
  - Object-oriented extensions in Oracle RDBMS.

- The lecture slides are based on the course textbook:
  - Connolly. T. and Begg, C. "*Database Systems - A Practical Approach to Design, Implementation, and Management*.", 6th Ed., Pearson Education Ltd

Next Generation Database Systems

- First Generation DBMS: Hierarchical and Network
  - Required complex programs for even simple queries.
  - Minimal data independence.
  - No widely accepted theoretical foundation.
- Second Generation DBMS: Relational DBMS
  - Helped overcome these problems.
- Third Generation DBMS:
  - OODBMS (Object-oriented DBMS)
    - Revolutionary approach
  - ORDBMS
    - Evolutionary approach
- NB: NoSQL movement – a disruptive technology, but (very) unlikely to displace RDBMSs



The Revolution in Database Architecture, Jim Gray, MSR-TR-2004-31, March 2004. – covers a wider classification than one here
http://research.microsoft.com/apps/pubs/default.aspx?id=64551

Motivation

- Changes in the computer industry (started in 1990s)
- RDBMSs have proven to be inadequate for applications that have different needs from those of “traditional” business database application
  - Engineering databases
    - Computer-Aided Design/Manufacturing (CAD/CAM)
  - Computer-aided software engineering (CASE)
  - Multimedia databases
  - Office Information Systems (OIS)
  - Network Management Systems
  - Hypertext databases
  - Geographic information systems (GIS)
  - Interactive and Dynamic Web sites
  - XML Databases

Specific Types of Applications

- **Engineering databases**, CAx: computer-aided design (CAD), manufacturing (CAM), engineering (CAE); CIM (computer-integrated manufacturing). Tasks: A CAD database stores data required for an engineering design, including the components of the items being designed, the inter-relationship of components, and old versions of designs.

- **GIS**: store spatial and temporal information, such as that used in land management and underwater exploration. Much of data is derived from survey and satellite photographs, and tends to be very large. Searches may involve identifying features based, for example, on shape, color, or texture, using advanced pattern-recognition techniques.

- *Note: we present some of the application types here; some others are explained in the book – read about them.*

Issues with RDBMSs

- *Poor representation of ‘real world’ entities*: the process of normalization leads to the creation of relations that do not correspond to entities in the ‘real world.
  - Many relations lead to many join operations during query processing.
  - JOIN – (one of) the costliest operation(s)

- *Semantic Overloading*: lack of mechanism to differentiate the representation of data/entities and relationships between data – only *relation* concept exists.

- *Homogeneous data structure:* horizontal (same attributes in the tuples) and vertical (values of a column conform to the same domain) homogeneity

- *Limited operations*: only set- and tuple-oriented operations in SQL specification – too restrictive for modelling real-world behaviour.
  - E.g. one needs more for GIS: lines, points and polygons are used and one needs operations for distance, intersection and containment

Issues with RDBMSs (cont.)

- *Impedance Mismatch*
  - Most DMLs lack *computational completeness*.
  - To overcome this, SQL can be embedded in a high-level 3GL.
  - Mixing different programming paradigms.
  - At some point in ’90s/00s: Estimated that as much as 30% of programming effort and code space is expended on this type of conversion.

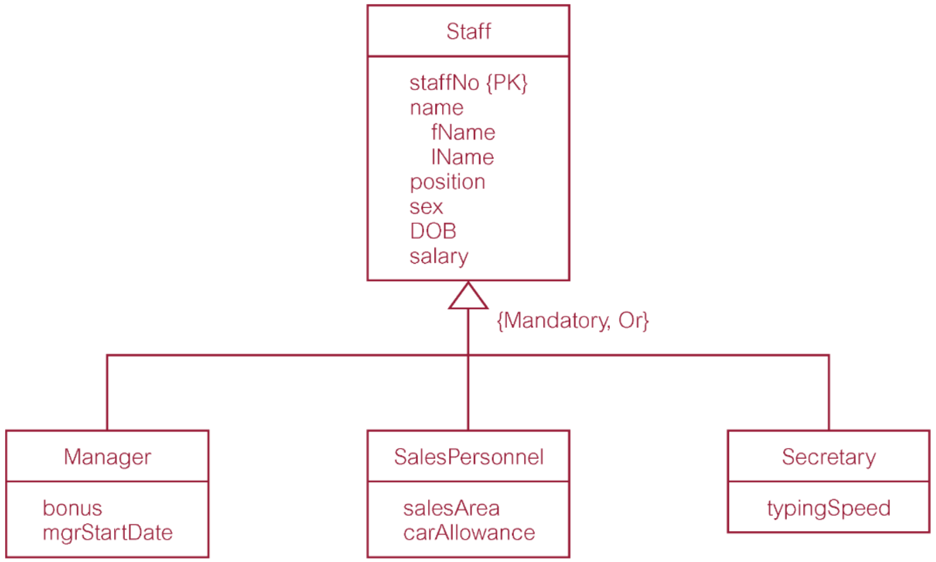
- *RDBMSs are poor at navigational access, i.e. movement between individual records.*

- *Long-duration transactions:*
  - Transactions are generally short-lived and concurrency control protocols not suited for long-lived transactions. Two-phase locking is not suitable for long transactions. [*we will mention this in the next week(s)*]

- *Schema changes are difficult.*

Storing Objects in a Relational Database

- One approach to achieve persistence with an OOPL is to use an RDBMS as the underlying storage engine.
- Requires mapping class instances (i.e. objects) to one or more tuples distributed over one or more relations.
- To handle class hierarchy, there are two basic tasks to perform:
  - (1) design relations to represent class hierarchy; [we look at this in the next slide(s)]
  - (2) design how objects will be accessed.



Mapping Classes to Relations

- Number of strategies for mapping classes to relations, although each results in a *loss* of semantic information.

(1) Map each class or subclass to a relation:

Staff (<u>staffNo</u>, fName, lName, position, sex, DOB, salary)

Manager (<u>staffNo</u>, bonus, mgrStartDate)

SalesPersonnel (<u>staffNo</u>, salesArea, carAllowance)

Secretary (<u>staffNo</u>, typingSpeed)

(2) Map each subclass to a relation

Manager (<u>staffNo</u>, fName, lName, position, sex, DOB, salary, bonus, mgrStartDate)

SalesPersonnel (<u>staffNo</u>, fName, lName, position, sex, DOB, salary, salesArea, carAllowance)

Secretary (<u>staffNo</u>, fName, lName, position, sex, DOB, salary, typingSpeed)

(3) Map the hierarchy to a single relation

Staff (staffNo, fName, lName, position, sex, DOB, salary, bonus, mgrStartDate, salesArea, carAllowance, typingSpeed, typeFlag)

- In none of these approaches the actual inheritance structure is fully preserved – even in the first alternative, it's unclear which are superclasses and which subclasses

ORDBMSs

- “*RDBMSs currently dominant database technology with estimated sales of US$24billion in 2011, expected to grow to US$37billion by 2016.*” [from the textbook]
  - The trend continues (see week 1 introductory slides)
- RDBMS vendors (were) conscious of “threat” and promise of ORDBMS/OODBMS.
- Agree that RDBMSs not currently suited to advanced database applications, and added functionality is required.
- RDBMS proponents reject claim that extended RDBMSs will not provide sufficient functionality or will be too slow to cope with new complexity.
- Can remedy shortcomings of relational model by extending model with OO features.

- Abstract, and User-Extensible, Data Types support in PostgreSQL from long ago – M. Stonebraker (a Turing Award winner):
  - http://cacm.acm.org/magazines/2016/2/197423-the-land-sharks-are-on-the-squawk-box/fulltext
  - https://www.postgresql.org/docs/current/extend-type-system.html:

ORDBMSs - Features

- OO features being added include:
  - user-extensible types,
  - encapsulation,
  - inheritance,
  - polymorphism,
  - dynamic binding of methods,
  - complex objects include values not compliant to the 1st Normal Form,
  - object identity.
- However, no single *object-relational* model. Thus, all models:
  - share basic relational tables and query language,
  - have some concept of ‘object’,
  - some can store methods (or procedures or triggers).

SQL:2011 - New OO Features

- Type constructors for row types and reference types.
- User-defined types (distinct types and structured types) that can participate in supertype/subtype relationships.
- User-defined procedures, functions, methods, and operators.
- Type constructors for collection types (arrays, multisets, etc.).
- *Support for Large Objects – Binary LOBs and Character LOBs.*
- *Recursion.*
  - *The concepts in the last 2 bullets not covered in the module*

– Note 1: ORDBMS, like pure RDBMSs, have their own dialects of SQL (different level of compliance to SQL standard)

– Note 2: we first cover the new features as defined in the SQL standard, and in the later part of the lecture cover Oracle’s implementation of some of these features (and continue this in the Tutorial).

Row Types

- Sequence of “attribute name - data type” pairs that provides data type to represent types of rows in tables.
- Allows complete rows to be:
  - stored in variables
  - passed as arguments to routines
  - returned as return values from function calls
- Also allows column of table to contain row values.

- Example – Use of Row Type

```
CREATE TABLE Branch (
    branchNo CHAR(4),
    address  ROW(
                 street   VARCHAR(25),
                 city     VARCHAR(15),
                 postcode ROW(cityIdentifier VARCHAR(4),subpart VARCHAR(4)))
);
INSERT INTO Branch VALUES
    (‘B005’, ROW (‘22 Deer Rd’, ‘London’, ROW(‘SW1’,‘4EH’)));
```

User-Defined Types (UDTs)

- SQL:2011 allows definition of UDTs.
- May be used in the same way as built-in types.
- Subdivided into two categories: *distinct* types and *structured* types.
- Distinct type allows differentiation between same underlying base types:

```
CREATE TYPE OwnerNoType AS VARCHAR(5) FINAL;
CREATE TYPE StaffNoType AS VARCHAR(5) FINAL;
```

- Error raised if an instance of one type treated as an instance of another type.
  - Not same as SQL *domains*, which constrain set of valid values that can be stored.
- Generally, UDT definition consists of one or more attribute definitions.
- Definition also consists of *routine* declarations (*operator* declarations are deferred for later versions of the standard).
- Can also define equality and ordering relationships
  - Instances of UDTs can be constrained to exhibit specified ordering properties

UDTs – Encapsulation and get/set functions

- Value of an attribute can be accessed using common dot notation:

```
   p.fName                   p.fName = ‘A. Smith’
```

- SQL encapsulates each attribute through an `observer (get)` and a `mutator (set)` function.

- These functions can be *redefined* by user in UDT definition.
- One input parameter can be designated as the result - RESULT keyword

```
FUNCTION fName(p PersonType) RETURNS VARCHAR(15)
     RETURN p.fName;

FUNCTION   fName(p   PersonType   RESULT,   newValue   VARCHAR(15))
     RETURNS PersonType
BEGIN
     p.fName = newValue;
     RETURN p;
END
```

UDTs – Constructors and NEW expression

- *(Public) constructor* function is automatically defined - creates new instances:

```
SET p = NEW PersonType();
```

- The constructor function has same name as type, takes zero arguments, and returns a new instance with attributes set to their default values.
- *User-defined constructor* methods can be provided to initialize new instances. Must have same name as UDT but different parameters to public constructor.
- Example constructor method

```
CREATE CONSTRUCTOR METHOD PersonType  (
      fN VARCHAR(15), lN VARCHAR(15), sx CHAR)
    RETURNS PersonType SELF AS RESULT
    BEGIN
      SET SELF.fName = fN;
      SET SELF.lName = lN;
      SET SELF.sex = sx;
      RETURN SELF;
    END;

SET p = NEW PersonType('John', 'White' 'M');
```

Example - Definition of new UDT

```
CREATE TYPE PersonType AS (
        dateOfBirth      DATE,
        fName            VARCHAR(15),
        lName            VARCHAR(15),
        sex              CHAR)
INSTANTIABLE /*instances can be created*/
NOT FINAL /*subtypes of this user-defined type can be created*/
REF IS SYSTEM GENERATED
INSTANCE METHOD age() RETURNS INTEGER,
INSTANCE METHOD age(DOB DATE) RETURNS PersonType;

CREATE INSTANCE METHOD age () RETURNS INTEGER
        FOR PersonType
    BEGIN
        RETURN   /* age calculated from SELF.dateOfBirth*/
    END;

CREATE INSTANCE METHOD age(DOB DATE) RETURNS PersonType
        FOR PersonType
    BEGIN
       SELF.dateOfBirth = /* code to set dateOfBirth from DOB*/;
       RETURN SELF;
    END;
```

Subtypes and Supertypes

- UDTs can participate in subtype/supertype hierarchy using **`UNDER`** clause.
- Multiple inheritance is **not** supported.
- Subtype inherits all the attributes and behavior of its supertypes.
- Can define additional attributes and methods and can override inherited methods.
- Concept of *substitutability* supported: whenever instance of supertype expected, an instance of a subtype can be used in its place.

Example - Creation of Subtype

```
CREATE TYPE StaffType UNDER PersonType AS (
   staffNo      VARCHAR(5),
   position     VARCHAR(10)   DEFAULT ‘Assistant’,
   salary       DECIMAL(7, 2),
   branchNo     CHAR(4))
INSTANTIABLE
NOT FINAL
INSTANCE METHOD isManager() RETURNS BOOLEAN;

CREATE INSTANCE METHOD isManager ()
   RETURNS BOOLEAN FOR StaffType
BEGIN
   IF SELF.position = ‘Manager’ THEN
       RETURN TRUE;
   ELSE
       RETURN FALSE;
   END IF
END;
```

Typo in the book

User-Defined Routines (UDRs)

- UDRs define approaches for manipulating data.
- Provide the required *behavior* for the UDTs
- UDRs may be defined as part of a UDT or *separately* as part of a schema.
- An SQL-invoked routine may be a *procedure*, *function*, or *method*.
- May be externally provided in standard programming language or defined completely in SQL.

- An SQL-invoked *procedure* is invoked from SQL CALL statement.
- May have zero or more parameters, each of which may be IN, OUT, or INOUT, and a body if defined fully within SQL.

- An SQL-invoked *function* returns a value.
- Any specified parameters must be input parameters, with one designated as result parameter (using RESULT keyword).
  – Mutator functions are always type-preserving

User-Defined Routines (UDRs)

- SQL-invoked *method* is similar to a function but:
  - method is associated with a single UDT;
  - signature of every method of a UDT must be specified in that UDT
- Three types of methods:
  - *constructor* methods, invoked using NEW;
  - *instance* methods, invoked using dot notation or using generalized invocation format; e.g. `p.fName` or `(p AS StaffType).fName()`;
  - *static* methods (analogous to class methods), invoked using `::` , e.g. `StaffType::totalStaff()`.
  - In the first two cases, the methods include additional implicit first parameter – `SELF`

- *External routine* defined by specifying an external clause that identifies 'compiled code' in operating system's file storage.
- ORDBMS will provide method to dynamically link this object file into the DBMS so that it can be invoked when required.
- This is outside of SQL standard and is left as implementation-defined.

Reference Types and Object Identity

- In SQL:2011, *reference types* can be used to define relationships between row types and uniquely identify a row within a table.
- Reference type value can be stored in one table and used as a direct reference to a specific row in some base table defined to be of this type (similar to *pointer type* in C/C++).
- Thus, references allow a row to be shared among multiple tables, and enable users to replace complex join definitions in queries with much *simpler path* expressions.
- References also give optimizer alternative way to *navigate* data instead of using value-based joins.
- REF IS SYSTEM GENERATED in CREATE TYPE indicates that actual values of associated REF type are provided by the system.

Table Creation based on UDT

```
CREATE TABLE Person (
  info    PersonType,
  …);
```

```
Person.info.fName
```

- or

```
CREATE TABLE Person OF PersonType (
  dateOfBirth
  …
  REF IS personID SYSTEM GENERATED);
```

```
Person.fName
```

Subtables and Supertables

- No mechanism to store all instances of given UDT, unless user explicitly creates a single table in which all instances are stored.
- Thus, in SQL:2011 may not be possible to apply an SQL query to all instances of a given UDT.
- Can use table inheritance, which allows table to be created that inherits all the attributes of a supertable using UNDER clause.
- Subtable/supertable *independent* from UDT inheritance facility!

- Example: Creation of Subtable

```
CREATE TABLE Staff OF StaffType UNDER Person;
```

  - Each row of supertable Person can correspond to at most one row in Staff.
  - Each row in Staff must have exactly one corresponding row in Person.
  - Containment used: row of subtable ‘contained’ in its supertable.

Example - Using Reference Type to Define a Relationship

```
CREATE TABLE PropertyForRent (
  propertyNo    PropertyNumber          NOT NULL,
  street        Street                  NOT NULL,
  …
  staffID REF(StaffType) SCOPE Staff
     REFERENCES   ARE   CHECKED   ON   DELETE
  CASCADE,
  PRIMARY KEY (propertyNo));
```

SYSTEM GENERATED
The StaffID actually points to SYSTEM GENERATED person ID (Staff table is created UNDER Person table, see previous)

- The SCOPE clause specifies the associated referenced table.
- Why is ON UPDATE not needed?

Example – Retrieve Specific Column/ Rows

- Find the names of all Managers.

```
SELECT s.lName
FROM Staff s
WHERE s.position = ‘Manager’;
```

- Implicitly defined `position` function is invoked

- SQL:2011 provides the same syntax as SQL2 for querying and modifying tables, but with various extensions to handle objects

Example - Invoke User-Defined Function

- Find the names and ages of all Managers.

```
SELECT s.lName, s.age
FROM Staff s
WHERE s.isManager;
```

- Uses user-defined function `isManager` as a predicate of the WHERE clause (returns TRUE if member of staff is a manager).
- Also uses inherited virtual observer function `age`.

Example - Use of `ONLY`

- Find names of all people over 65.

```
SELECT p.lName, p.fName
FROM Person p
WHERE p.age > 65;
```

- This will list out not only records explicitly inserted into Person table, but also records inserted directly/indirectly into subtables of Person.
- Can restrict access to specific instances of Person table, *excluding* any subtables, using `ONLY`.

```
SELECT p.lName, p.fName
FROM ONLY (Person) p
WHERE p.age > 65;
```

Example - Use of Dereference Operator

- Find full name of the member of staff who manages property PG4.

```
SELECT    p.staffID->fName AS fName,
          p.staffID->lName AS lName,
FROM      PropertyForRent p
WHERE     p.propertyNo = 'PG4';
```

- In SQL2, this query would have required a **JOIN** (between PropertyForRent and Staff tables) or nested subquery!

- To retrieve the member of staff for property PG4, rather than just the first and last name:

```
SELECT    DEREF(p.staffID) AS Staff
FROM      PropertyForRent p
WHERE     p.propertyNo = 'PG4';
```

Collection Types

- Collections are type constructors used to define collections of other types.
- Used to store multiple values in single column and can result in nested tables.
- SQL:2011 has parameterized ARRAY and MULTISET collection types.
- **Later** SQL may add parameterized LIST and SET collection types.
- The parameter may be predefined type, UDT, row type, or another collection (but **not** reference type or UDT containing reference type).

- **ARRAY: 1D array with maximum number of elements.**
- **MULTISET: unordered collection that allows duplicates.**
- LIST: ordered collection that allows duplicates.
- SET: unordered collection that does not allow duplicates.

Example - Use of ARRAY Collection

- Branch has up to three telephone numbers:

```
telNo     VARCHAR(13) ARRAY[3]
```

- Retrieve first and last phone number for B003.

```
SELECT telNo[1], telNo[CARDINALITY(telNo)]
FROM Branch
WHERE branchNo = ‘B003’;
```

- `CARDINALITY` – the f-on returns the number of current elements
- Two arrays are the same **iff**:
  - They have the same cardinality, and the values of the corresponding elements are the same.

MULTISET

- Unordered collection of elements, all of the same type, with duplicates permitted.
- Since multiset is unordered there is no ordinal position to reference individual elements. Unlike arrays, multiset is an unbounded collection.
- Operators are provided to convert multiset to table (UNNEST) and table to multiset (MULTISET).

Operations on MULTISET
(Look in the textbook)

- SET - removes duplicates from a multiset to produce a set.
- CARDINALITY - returns number of current elements.

- Other operations: ELEMENT, UNION, INTERSECT, EXCEPT, and Aggregate Functions and Predicates – see the textbook.

Example - Use of Collection MULTISET

- Extend Staff table to contain details of next-of-kin (we include the definition of `nok` column in Staff table):

```
nok        NameType MULTISET
```

Note: `NameType` contains `fName` and `lName` attributes

- Find first and last names of John White’s next of kin.

```
SELECT n.fName, n.lName
FROM Staff s, UNNEST(s.nok) AS n(fName, lName)
WHERE s.lName = ‘White’ AND s.fName = ‘John’;
```

- Note that in the FROM clause we may use multiset-valued field `s.nok` as a table reference

Oracle ORDBMs
Object-Oriented Extensions in Oracle

- Many of the object-oriented features that appear in SQL:2011 standard appear in Oracle in one form or another.
- Oracle supports two user-defined data types:
  - object types;
  - collection types.

Object Types in Oracle

- An object type is a schema object that has a name, a set of attributes based on the Oracle built-in data types or possibly other object types, and a set of methods.

```
CREATE TYPE AddressType AS OBJECT (
   street          VARCHAR2(25),
   city            VARCHAR2(15),
   postcode        VARCHAR2(8)
 )
```

Object Types in Oracle / UDTs

```
CREATE TYPE StaffType AS OBJECT (
      staffNo       VARCHAR2(5),
      fName         VARCHAR2(15),
      …
      MAP MEMBER FUNCTION age
             RETURN INTEGER,
      PRAGMA RESTRICT_REFERENCES( age, WNDS, WNPS, RNPS));
```

- `CREATE OR REPLACE TYPE BODY …` – the code for the methods that implement the type

- Pragma clause is a compiler directive that denies member functions read/write access to database tables and/or package variables.
  - `WNDS` – does not modify database tables. (“Write no database state”)
  - `WNPS` – does not modify package variables.
  - `RNDS` – does not query database tables.
  - `RNPS` – does not reference package variables.

Object Types in Oracle / UDTs

Error in the book

```
CREATE TYPE BranchType AS OBJECT (
      branchNo VARCHAR2(4),
      address  AddressType,
      MAP MEMBER FUNCTION getbranchNo RETURN VARCHAR2,
      PRAGMA   RESTRICT_REFERENCES(getbranchNo,   WNDS,   WNPS,   RNDS,
      RNPS));
```

- Can now create a Branch (Object) table:

```
    CREATE TABLE Branch OF BranchType
      (branchNo PRIMARY KEY);
```

Methods in Oracle

- Methods of an object type are classified as member, static, and comparison.
- *Member* method is a function/procedure that always has implicit SELF parameter as first parameter (whose type is containing object type).
  - Useful as *observer* and *mutator* functions.
  - `object.method().`
- *Static* method is a function/procedure that does not have an implicit SELF parameter.
  - Useful for specifying user-defined constructors or cast methods and may be invoked by qualifying method with the type name, as in `typename.method().`
  - Constructor methods
    - Every object has *a system-defined* constructor method
    - Same name as the object type, same name and datatypes of the parameters as the object type's attributes

Methods in Oracle (contd.)

- *Comparison* method used for comparing instances of objects.
- Oracle provides two ways to define an order relationship among objects of a given type:
  - a *map* method uses Oracle's ability to compare *built-in* types.
  - an *order* method uses its own internal logic to compare two objects of a given object type. It returns a value that encodes the order relationship. For example, may return -1 if first is smaller, 0 if they are equal, and 1 if first is larger.
- **Either** a map **or** an order method can exist, but not both.
- Methods can be implemented in PL/SQL, Java, and C.
- *Overloading* is supported provided the methods' formal parameters differ in number, order, or data type.

Object Identifiers

- Every row object in an object table has associated logical OID, which uniquely identifies the row.
- The OID column is hidden from users and there is no access to its internal structure.
- Oracle requires every row object to have a unique OID, which may be specified to come from the row object's PK or to be system-generated.

```
CREATE TABLE Branch OF BranchType
       (branchNo PRIMARY KEY)
OBJECT IDENTIFIER PRIMARY KEY;
```

- Objects that appear in object tables are called *row objects*; objects that occupy columns of relational tables or as attributes of other objects are called *column objects*.

REF Data Type

- Oracle provides a built-in data type called `REF` to encapsulate references to row objects of a specified object type.
- In effect, a `REF` is used to model an association between two row objects.
- A `REF` can be used to examine or update the object it refers to and to obtain a copy of the object it refers to.
- *Only* changes that can be made to a `REF` are: to replace its contents with a reference to a different object of same object type, or to assign it a null value.
- At implementation level, Oracle uses OIDs to construct REFs
- `SCOPE` – same as in SQL:2011 – a `REF` can be constrained to contain references to a specified object table
- `DEREF` – the operator to access the object the `REF` refers to

```
CREATE TYPE BranchType AS OBJECT (
   branchNo     VARCHAR2(4),
   address      AddressType,
   manager      REF StaffType,
   MAP MEMBER FUNCTION        getbranchNo RETURN VARCHAR2(4),
   PRAGMA RESTRICT_REFERENCES(getbranchNo, WNDS, WNPS, RNDS, RNPS));
```

Type Inheritance

- *Single* inheritance supported
- Subtype inherits all the attributes and the methods from the supertype
- Can add new attributes/methods, and override inherited methods
- UNDER clause used for defining supertype/subtype
  - Same as in SQL:2011

- (No direct Table inheritance, though)

Collection Types

- Oracle supports two collection types: <u>array types</u> and <u>(nested) table types</u>.

- An array is an ordered set of data elements, all of same data type.
- Each element has an *index*, a number corresponding to the element's position in the array.
- An array can have a *fixed* or *variable* size, although in latter case maximum size must be specified when array type is declared.

- `CREATE TYPE TelNoArrayType AS VARRAY(3) OF VARCHAR2(13)`

- The creation of array type does not allocate space, but defines a data type that can be used for: data type of a column in relational table, an object type attribute, a PL/SQL variable, parameter or function return type.
  - E.g. `phoneList TelNoArrayType ...` *[to modify the BranchType]*

Nested Tables

- An unordered set of data elements, all of same data type.
- It has a single column of a built-in type or an object type.
- If column is an object type, the nested table can also be viewed as a multi-column table, with a column for each attribute of the object type.

```
CREATE TYPE NextOfKinType AS OBJECT (
        fName    VARCHAR2(15),
        lName    VARCHAR2(15),
        telNo    VARCHAR2(13));
CREATE TYPE NextOfKinNestedType AS TABLE OF NextOfKinType;
```

- Can now modify `StaffType` to include this new type:

```
        nextOfKin         NextOfKinNestedType
```

- Can now create `Staff` table:

```
   CREATE TABLE Staff OF StaffType (
        PRIMARY KEY      staffNo)
        OBJECT IDENTIFIER PRIMARY KEY
        NESTED TABLE nextOfKin STORE AS NextOfKinStorageTable (
        (PRIMARY KEY(Nested_Table_Id, lName, telNo))
        ORGANIZATION INDEX COMPRESS)
        RETURN AS LOCATOR;
```

Nested Tables (contd.)

- Rows of a nested table stored in a separate storage table
  - This table cannot be *directly* queried by the user
  - But can be referenced in DDL operations for maintenance purposes
  - `STORE AS` indicates that index-organised storage approach is used (`ORGANIZATION INDEX`)
  - `COMPRESS` – store `Nested_Table_Id` part of the index key only once for each row of the parent row object.

- A hidden column in this table, `Nested_Table_Id`, matches the rows with their parent row.
  - All the elements of the nested table of a given row of Staff have the same value of `Nested_Table_Id`
- The clause `RETURN AS LOCATOR` indicates that entire nested table is to be returned in a locator /handle form.
  - The default is `VALUE`, which returns entire nested table – which might have performance implications

Manipulating Object Tables

```
INSERT INTO Staff VALUES ('SG37', 'Ann',
  'Beech', 'Assistant', 'F', '10-Nov-1960', 12000,
   NextOfKinNestedType());
```

- The expression `NextOfKinNestedType()` invokes the constructor method for this type to create empty `nextOfKin` attribute

```
INSERT INTO TABLE (SELECT s.nextOfKin
                   FROM Staff s
                   WHERE s.staffNo = 'SG5')
 VALUES ('John', 'Brand', '0141-848-2000');
```

- `TABLE` expression identifies the nested table as the target of insertion.
  - Namely, the nested table in the `nextOfKin` column of the row object in the Staff table that has value "SG5" for `staffNo`

Manipulating Object Tables

- Can now insert object into Branch table:

```
INSERT INTO Branch
   SELECT  'B003',
               AddressType('163 Main St','Glasgow', 'G11 9QX'),
     REF(s),
               TelNoArrayType('0141-339-2178', '0141-339-4439')
   FROM Staff s
   WHERE s.staffNo = 'SG5';
```

- Or alternatively:

```
INSERT INTO Branch VALUES(
   'B003',
   AddressType('163 Main St','Glasgow','G119QX'),
   (SELECT   REF(s)   FROM   Staff   s   WHERE   s.staffNo='SG5'),
   TelNoArrayType('0141-339-2178', '0141-339-4439')
);
```

Querying Object Tables

```
SELECT    b.branchNo
FROM      Branch b
ORDER BY VALUE(b);
```

- This query implicitly calls the *map* function `getBranchNo` to order data in ascending order of `branchNo`
- VALUE takes as its argument a table alias and returns object instances of the object table. The type of the object instances - the same type as the object table.

```
 SELECT   b.branchNo, b.address,
          DEREF(b.manager), b.phoneList
 FROM Branch b
 WHERE b.address.city = 'Glasgow'
 ORDER BY VALUE(b);
```

- This query returns all data for each corresponding branch
- `DEREF` operator used to access the manager object
- Returned values are from: branchNo column, all columns of an address, all columns of the manager object (of type `StaffType`), and all relevant Tel No-s

Querying Object Tables

```
SELECT b.branchNo, b.manager.staffNo, n.*
   FROM Branch b, TABLE (b.manager.nextOfKin) n
   WHERE b.branchNo = ‘B003’;
```

- The statement retrieves next of kin data for all staff at the specified branch

- Many applications cannot handle collection types and thus need a flattened view of the data
  - Flattening/unnesting done using the `TABLE` keyword

- See the following for further examples:

http://www.orafaq.com/wiki/NESTED_TABLE

Required Reading

- Connolly,T. and Begg, C. "*Database Systems - A Practical Approach to Design, Implementation, and Management*.", 6th Ed., Pearson Education Ltd
  - Chapter 9, but NOT
    - 9.3.2 "Accessing Objects in the Relational Database"
    - 9.5.10 "Typed Views"
    - 9.5.11 "Persistent Stored Modules"
    - 9.5.12 "Triggers"
    - 9.5.13 "Large Objects"
    - 9.5.14 "Recursion"
    - 9.6.3 "Object Views"
    - 9.6.4 "Privileges"

- This list is only indicative, and the text in these sections should be of course looked at as "additional", but required, reading to the lecture slides.
- This is certainly not an exhaustive list.
- The content of the chapters referred to might NOT be fully relevant to the module. You need to make the final decision about what is relevant and what is not yourselves, given the material that we covered in the lecture.