UNIT V OBJECT RELATIONAL AND NO-SQL DATABASES

Mapping EER to ODB schema – Object identifier – reference types – rowtypes – UDTs – Subtypes and supertypes – user-defined routines – Collection types – Object Query Language; No-SQL: CAP theorem – Document-based: MongoDB data model and CRUD operations; Column-based: Hbase data model and CRUD operations.

INTRODUCTION TO OBJECT RELATIONAL DATABASE(ORD):

- An **object-relational database (ORD)** is a database management system (DBMS) that's composed of both a **relational database** (RDBMS) and an **object-oriented database** (OODBMS).
- ORD supports the basic components of any object-oriented database model in its schemas and the query language used, such as objects, classes and inheritance.
- An object-relational database may also be known as an object relational database management systems (ORDBMS).
- ORD is said to be the middleman between relational and object-oriented databases because it contains aspects and characteristics from both models.
- In ORD, the basic approach is based on RDB, since the data is stored in a traditional database and manipulated and accessed using queries written in a query language like SQL.
- However, ORD also showcases an object-oriented characteristic in that the database is considered an object store, usually for software that is written in an object-oriented programming language. Here, APIs are used to store and access the data as objects.
- One of **ORD's aims is to bridge the gap between conceptual data modeling techniques for relational and object-oriented databases** like the entity-relationship diagram (ERD) and object-relational mapping (ORM). It also aims to connect the divide between relational databases and the object-oriented modeling techniques that are usually used in programming languages like Java, C# and C++.

RDBMS:

RDBMS stands for Relational Database Management System. It is a database management system based on the relational model i.e. the data and relationships are represented by a collection of inter-related tables. It is a DBMS that enables the user to create, update, administer and interact with a relational database. RDBMS is the basis for SQL, and for all

modern database systems like MS SQL Server, IBM DB2, Oracle, MySQL, and Microsoft Access.

OODBMS:

OODBMS stands for Object-Oriented Database Management System. It is a DBMS where data is represented in the form of objects, as used in object-oriented programming. OODB implements object-oriented concepts such as classes of objects, object identity, polymorphism, encapsulation, and inheritance. An object-oriented database stores complex data as compared to relational database. Some examples of OODBMS are Versant Object Database, Objectivity/DB, ObjectStore, Caché and ZODB.

Difference between RDBMS and OODBMS:

BASIS	RDBMS	OODBMS
Long Form	Stands for Relational Database Management System.	Stands for Object Oriented Database Management System.
Way of storing data	Stores data in Entities, defined as tables hold specific information.	Stores data as Objects.
Data Complexity	Handles comparatively simpler data.	Handles larger and complex data than RDBMS.
Grouping	Entity type refers to the collection of entity that share a common definition.	Class describes a group of objects that have common relationships, behaviors, and also have similar properties.
Data Handling	RDBMS stores only data.	Stores data as well as methods to use it.
Main Objective	Data Independence from application program.	Data Encapsulation.
Key	A Primary key distinctively identifies an object in a table..	An object identifier (OID) is an unambiguous, long-term name for any type of object or entity.

Advantages of Object Relational model

The advantages of the Object Relational model are –

Inheritance: The Object Relational data model allows its users to inherit objects, tables etc. so that they can extend their functionality. Inherited objects contains new attributes as well as the attributes that were inherited.

Complex Data Types : Complex data types can be formed using existing data types. This is useful in Object relational data model as complex data types allow better manipulation of the data.

Extensibility: The functionality of the system can be extended in Object relational data model. This can be achieved using complex data types as well as advanced concepts of object oriented model such as inheritance.

Disadvantages of Object Relational model

The object relational data model can get quite complicated and difficult to handle at times as it is a combination of the Object oriented data model and Relational data model and utilizes the functionalities of both of them.

Differences between Conceptual Design of ODB and RDB:

i) One of the main differences between ODB and RDB design is how relationships are handled.

- **In ODB, relationships are typically handled by having relationship properties or reference attributes that include OID(s) of the related objects.** These can be considered as ***OID references*** to the related objects. Both single references and collections of references are allowed. References for a binary relationship can be declared in a single direction, or in both directions, depending on the types of access expected. If declared in both directions, they may be specified as inverses of one another, thus enforcing the ODB equivalent of the relational referential integrity constraint.
- **In RDB, relationships among tuples (records) are specified by attributes with matching values.** These can be considered as ***value references*** and are specified via *foreign keys*, which are values of primary key attributes repeated in tuples of the referencing relation. These are limited to being single-valued in each record because multivalued attributes are not permitted in the basic relational model. Thus, M:N relationships must be represented not directly, but as a separate relation (table)

- Mapping binary relationships that contain attributes is not straightforward in ODBs, since the designer must choose in which direction the attributes should be included. If the attributes are included in both directions, then redundancy in storage will exist and may lead to inconsistent data. Hence, it is sometimes preferable to use the relational approach of creating a separate table by creating a separate class to represent the relationship. This approach can also be used for *n*-ary relationships, with degree $n > 2$.

ii) Second major area of difference between ODB and RDB design is how inheritance is handled.

- In ODB, these structures are built into the model, so the mapping is achieved by using the inheritance constructs, such as *derived* (:) and *extends*.
- In relational design, there are several options to choose from since no built-in construct exists for inheritance in the basic relational model.

iii) The third major difference is that in ODB design, it is necessary to specify the operations early on in the design since they are part of the class specifications. Although it is important to specify operations during the design phase for all types of data-bases, it may be delayed in RDB design as it is not strictly required until the implementation phase.

Mapping EER to ODB schema:

It is relatively straightforward to design the type declarations of object classes for an ODBMS from an EER schema that contains neither categories nor n-ary relationships with $n > 2$. However, the operations of classes are not specified in the EER diagram and must be added to the class declarations after the structural mapping is completed. The outline of the mapping from EER to ODL is as follows:

Step 1. Create an ODL class for each EER entity type or subclass. The type of the ODL class should include all the attributes of the EER class. Multivalued attributes are typically declared by using the set, bag, or list constructors. If the values of the multivalued attribute for an object should be ordered, the list constructor is chosen; if duplicates are allowed, the bag constructor should be chosen; otherwise, the set constructor is chosen. Composite attributes are mapped into a tuple constructor (by using a struct declaration in ODL).

Declare an extent for each class, and specify any key attributes as keys of the extent. (This is possible only if an extent facility and key constraint declarations are available in the ODBMS.)

Step 2.

- **Add relationship properties or reference attributes for each binary relation- ship into the ODL classes that participate in the relationship. These may be created in one or both directions.**
- If a binary relationship is represented by references in both directions, declare the references to be relationship properties that are inverses of one another, if such a facility exists. If a binary relationship is represented by a reference in only one direction, declare the reference to be an attribute in the referencing class whose type is the referenced class name.
- Depending on the cardinality ratio of the binary relationship, the relationship properties or reference attributes may be single-valued or collection types. They will be single-valued for binary relationships in the 1:1 or N:1 directions; they are collection types (set-valued or list-valued 41) for relationships in the 1:N or M:N direction. An alternative way to map binary M:N relationships is discussed in step 7.
- If relationship attributes exist, a tuple constructor (struct) can be used to create a structure of the form **<reference, relationship attributes>,** which may be included instead of the reference attribute. However, this does not allow the use of the inverse constraint. Additionally, if this choice is represented in both directions, the attribute values will be represented twice, creating redundancy.

Step 3. Include appropriate operations for each class. These are not available from the EER schema and must be added to the database design by referring to the original requirements. A constructor method should include program code that checks any constraints that must hold when a new object is created. A destructor method should check any constraints that may be violated when an object is deleted. Other methods should include any further constraint checks that are relevant.

Step 4. An ODL class that corresponds to a subclass in the EER schema inherits (via extends) the type and methods of its superclass in the ODL schema. Its specific (noninherited) attributes, relationship references, and operations are specified, as discussed in steps 1, 2, and 3.

Step 5. Weak entity types can be mapped in the same way as regular entity types. An alternative mapping is possible for weak entity types that do not participate in any relationships except their identifying relationship; these can be mapped as though they were composite multivalued attributes of the owner entity type, by using the set < struct < ... >> or list < struct < ... >> constructors. The attributes of the weak entity are included in the struct < ... > construct, which corresponds to a tuple constructor. Attributes are mapped as discussed in steps 1 and 2.

Step 6. Categories (union types) in an EER schema are difficult to map to ODL. It is possible to create a mapping similar to the EER-to-relational mapping by declaring a class to represent the category and defining 1:1 relationships between the category and each of its superclasses. Another option is to use a union type, if it is available.

Step 7. An n-ary relationship with degree n > 2 can be mapped into a separate class, with appropriate references to each participating class. These references are based on mapping a 1:N relationship from each class that represents a participating entity type to the class that represents the n-ary relationship. An M:N binary relationship, especially if it contains relationship attributes, may also use this mapping option, if desired.

OBJECT IDENTIFIER:

An object identifier (OID) is a string, of decimal numbers, that uniquely identifies an object. These objects are typically an object class or an attribute.

- Every row object in an object table has an associated logical object identifier (OID), which by default is a unique system-generated identifier assigned for each row object.
- The purpose of the OID is to uniquely identify each row object in an object table. It distinguishes the object from all other objects.
- To do this, Oracle implicitly creates and maintains an index on the OID column of the object table. The OID column is hidden from users and there is no access to its internal structure. Although OID values in themselves are not very meaningful, the OIDs can be used to fetch and navigate objects. (Note, objects that appear in object tables are called **row objects** and objects that occupy columns of relational tables or as attributes of other objects are called **column objects**.)

- Oracle requires every row object to have a unique OID. The unique OID value may be specified to come from the row object's primary key or to be system generated, using either the clause OBJECT IDENTIFIER IS PRIMARY KEY or OBJECT IDENTIFIER IS SYSTEM GENERATED (the default) in the CREATE TABLE statement.
- For example, we could restate the creation of the Branch table as:

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

- Ideally, 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.

- You can construct pointers (REFs) to the row objects in an object view. Because the view data is not stored persistently, you must specify a set of distinct values to be used as object identifiers. Object identifiers allow you to reference the objects in object views and pin them in the object cache.
- If you do not have an OID, you can specify the object class or attribute name appended with **-oid**. For example, if you create the attribute tempID, you can specify the OID as **tempID-oid**.
- Object identifiers (OIDs) are used internally by PostgreSQL as primary keys for various system tables. OIDs are not added to user-created tables, unless WITH OIDS is specified when the table is created, or the **default_with_oids** configuration variable is enabled. Type oid represents an object identifier.
- The oid type is currently implemented as an unsigned four-byte integer. Therefore, it is not large enough to provide database-wide uniqueness in large databases, or even in large individual tables. So, using a user-created table's OID column as a primary key is discouraged. OIDs are best used only for references to system tables.
- The oid type itself has few operations beyond comparison. It can be cast to integer, however, and then manipulated using the standard integer operators.

There are also several alias types for oid as shown below.

Object Identifier Types

Name	References	Description	Value Example
oid	**any**	**numeric object identifier**	564182
regproc	**pg_proc**	**function name**	sum
regprocedure	**pg_proc**	**function with argument types**	sum(int4)
regoper	**pg_operator**	**operator name**	+
regoperator	**pg_operator**	**operator with argument types**	*(integer,integer) **or** -(NONE,integer)
regclass	**pg_class**	**relation name**	pg_type
regtype	**pg_type**	**data type name**	integer

The OID alias types have no operations of their own except for specialized input and output routines. These routines are able to accept and display symbolic names for system objects, rather than the raw numeric value that type oid would use. The alias types allow simplified lookup of OID values for objects.

For example, to examine the pg_attribute rows related to a table mytable, one could write

```
SELECT * FROM pg_attribute WHERE attrelid = 'mytable'::regclass;
```

rather than

```
SELECT * FROM pg_attribute
  WHERE attrelid = (SELECT oid FROM pg_class WHERE relname = 'mytable');
```

- All of the OID alias types accept schema-qualified names, and will display schema-qualified names on output if the object would not be found in the current search path without being qualified.

- The regproc and regoper alias types will only accept input names that are unique (not overloaded), so they are of limited use; for most uses regprocedure or regoperator is more appropriate.
- For regoperator, unary operators are identified by writing NONE for the unused operand.
- An additional property of the OID alias types is that if a constant of one of these types appears in a stored expression (such as a column default expression or view), it creates a dependency on the referenced object.
- For example, if a column has a default expression nextval('my_seq'::regclass), PostgreSQL understands that the default expression depends on the sequence my_seq; the system will not let the sequence be dropped without first removing the default expression.

REFERENCE TYPES

- 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.
- Reference types have been able to be used to define relationships between row types and uniquely identify a row within a table.
- A reference type value can be stored in one (typed) table and used as a direct reference to a specific row in some base table that has been defined to be of this type (similar to the notion of a pointer type in C or C++). In this respect, a reference type provides a similar functionality as the object identifier (OID) of object-oriented DBMSs.
- 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.
- 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. The only changes that can be made to a REF are to replace its contents with a reference to a different object of the same object type or to assign it a null value.
- As it is possible for the object identified by a REF to become unavailable, for example through deletion of the object, Oracle SQL has a predicate IS DANGLING to test REFs for this condition. Oracle also provides a dereferencing operator, DEREF, to access the object referred to by a REF.

- For example, to model the manager of a branch we could change the definition of type BranchType 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));

In this case, we have modeled the manager through the reference type, REF StaffType.

- References also give the optimizer an alternative way to navigate data instead of using value-based joins.
- REF IS SYSTEM GENERATED in a CREATE TYPE statement indicates that the actual values of the associated REF type are provided by the system.
- Other options are available but we omit the details here; the default is REF IS SYSTEM GENERATED. As we see shortly, a base table can be created to be of some structured type.
- Other columns can be specified for the table but at least one column must be specified, namely a column of the associated REF type, using the clause REF IS SYSTEM GENERATED. This column is used to contain unique identifiers for the rows of the associated base table.
- The identifier for a given row is assigned when the row is inserted into the table and remains associated with that row until it is deleted.

Rules for REF Columns and Attributes

Rules for REF columns and attributes can be enforced by the use of constraints.

In Oracle Database, a REF column or attribute can be unconstrained or constrained using a SCOPE clause or a referential constraint clause. When a REF column is unconstrained, it may store object references to row objects contained in any object table of the corresponding object type.

Oracle Database does not ensure that the object references stored in such columns point to valid and existing row objects. Therefore, REF columns may contain object references that do not point to any existing row object. Such REF values are referred to as dangling references.

A SCOPE constraint can be applied to a specific object table. All the REF values stored in a column with a SCOPE constraint point at row objects of the table specified in the SCOPE clause. The REF values may, however, be dangling.

A REF column may be constrained with a REFERENTIAL constraint similar to the specification for foreign keys. The rules for referential constraints apply to such columns. That is, the object reference stored in these columns must point to a valid and existing row object in the specified object table.

PRIMARY KEY constraints cannot be specified for REF columns. However, you can specify NOT NULL constraints for such columns.

Example: Using a reference type to define a relationship

In this example, we model the relationship between PropertyForRent and Staff using a reference type.

CREATE TABLE PropertyForRent(

propertyNo PropertyNumber NOT NULL,

street Street NOT NULL,

city City NOT NULL,

postcode PostCode,

type PropertyType NOT NULL DEFAULT 'F',

rooms PropertyRooms NOT NULL DEFAULT 4,

rent PropertyRent NOT NULL DEFAULT 600,

staffID **REF(StaffType) SCOPE Staff**

REFERENCES ARE CHECKED ON DELETE CASCADE,

PRIMARY KEY (propertyNo));

- In the above example, we have used a reference type, **REF(StaffType),** to model the relationship.
- The **SCOPE** clause specifies the associated referenced table.

- **REFERENCES ARE CHECKED** indicates that referential integrity is to be maintained (alternative is REFERENCES ARE NOT CHECKED).
- **ON DELETE CASCADE** corresponds to the normal referential action. Note that an ON UPDATE clause is not required, as the column staffID in the Staff table cannot be updated.

ROWTYPES:

A row type is a sequence of field name/data type pairs that provides a data type to represent the types of rows in tables, so that complete rows can be stored in variables, passed as arguments to routines, and returned as return values from function calls. A row type can also be used to allow a column of a table to contain row values. In essence, the row is a table nested within a table.

EXAMPLE **Use of row type**

To illustrate the use of row types, we create a simplified Branch table consisting of the branch number and address, and insert a record into the new table:

```
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('23 Deer Rd', 'London', ROW('SW1', '4EH')));
UPDATE Branch
SET address = ROW ('23 Deer Rd', 'London', ROW ('SW1', '4EH'))
WHERE address = ROW ('23 Deer Rd', 'London', ROW ('SW1', '4EH'));
```

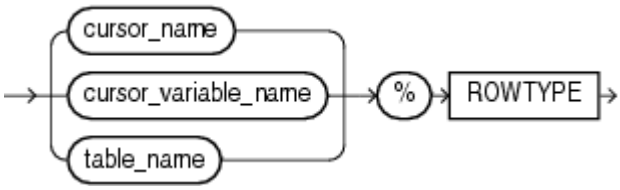
%ROWTYPE Attribute

- The %ROWTYPE attribute provides a record type that represents a row in a database table.
- The record can store an entire row of data selected from the table or fetched from a cursor or cursor variable.

- Variables declared using %ROWTYPE are treated like those declared using a datatype name. You can use the %ROWTYPE attribute in variable declarations as a datatype specifier.
- Fields in a record and corresponding columns in a row have the same names and datatypes. However, fields in a %ROWTYPE record do not inherit constraints, such as the NOT NULL column or check constraint, or default values.

Syntax

%rowtype attribute ::= {cursor_name | cursor_variable_name | table_name}%ROWTYPE



Keyword and Parameter Description

cursor_name

An explicit cursor previously declared within the current scope.

cursor_variable_name

A PL/SQL strongly typed cursor variable, previously declared within the current scope.

table_name

A database table or view that must be accessible when the declaration is elaborated.

Usage Notes

Declaring variables as the type *table_name*%ROWTYPE is a convenient way to transfer data between database tables and PL/SQL. You create a single variable rather than a separate variable for each column. You do not need to know the name of every column. You refer to the columns using their real names instead of made-up variable names. If columns are later added to or dropped from the table, your code can keep working without changes.

To reference a field in the record, use dot notation (record_name.field_name). You can read or write one field at a time this way.

There are two ways to assign values to all fields in a record at once:

- First, PL/SQL allows aggregate assignment between entire records if their declarations refer to the same table or cursor.
- You can assign a list of column values to a record by using the SELECT or FETCH statement. The column names must appear in the order in which they were declared. Select-items fetched from a cursor associated with %ROWTYPE must have simple names or, if they are expressions, must have aliases.

Example:

The following example uses %ROWTYPE to declare two records. The first record stores an entire row selected from a table. The second record stores a row fetched from the c1 cursor, which queries a subset of the columns from the table. The example retrieves a single row from the table and stores it in the record, then checks the values of some table columns.

```
DECLARE
  emp_rec   employees%ROWTYPE;
  my_empno  employees.employee_id%TYPE := 100;
  CURSOR c1 IS
    SELECT department_id, department_name, location_id FROM departments;
  dept_rec  c1%ROWTYPE;
BEGIN
  SELECT * INTO emp_rec FROM employees WHERE employee_id = my_empno;
  IF (emp_rec.department_id = 20) AND (emp_rec.salary > 2000) THEN
    NULL;
  END IF;
END;
/
```

UDT:

- User Defined Types (UDTs) is a data type that is derived from an existing data type. UDT is used to extend the built-in types already available and create own customized data types.
- They may be used in the same way as the predefined types (for example, CHAR, INT, FLOAT).
- UDTs are subdivided into two categories:
    - Distinct types.
    - Structured types.

Distinct types:

- The simpler type of UDT is the distinct type, which allows differentiation between the same underlying base types.

For example,

CREATE TYPE OwnerNumberType AS VARCHAR(5) FINAL;

CREATE TYPE StaffNumberType AS VARCHAR(5) FINAL;

Structured types:

- A structured type is a user-defined type that has a structure that is defined in the database. It contains a sequence of named attributes, each of which has a data type.
- A structured type also includes a set of method specifications.It can be used as the type of a table, view, or column.

For example,

CREATE TYPE Person_type AS (Name VARCHAR(20), Age INT, Address VARCHAR(40)) INSTANTIABLE FINAL;

In its more general case, a UDT definition consists of one or more attribute definitions, zero or more methods (routine declarations) and, in a subsequent order, operator declarations.

The value of an attribute can be accessed using the common dot notation (.).

For example,

p.fName

p.fName = ‘A. Smith’

Encapsulation and observer and mutator functions:

- SQL encapsulates each attribute of structured types by providing a pair of built-in routines that are invoked whenever a user attempts to reference the attribute, an observer (get) function and a mutator (set) function.
- The observer function returns the current value of the attribute;
- The mutator function sets the value of the attribute to a value specified as a parameter.

Observer functions:

For example, the observer function for the fName attribute of PersonType would be:

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

Mutator functions:

The corresponding mutator function to set the value to newValue would be:

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

Constructor functions and the NEW expression

- A constructor function is automatically defined to create new instances of the type.
- The constructor function has the same name and type as the UDT, takes zero arguments, and returns a new instance of the type with the attributes set to their default value.
- User-defined constructor methods can be provided by the user to initialize a newly created instance of a structured type.

For example,

CREATE CONSTRUCTOR METHOD PersonType (fN VARCHAR(15), lN VARCHAR(15), gn CHAR) **RETURNS** PersonType **SELF AS RESULT**

BEGIN

SET SELF.fName = fN;

SET SELF.lName = lN;

SET SELF.Gender = gn;

RETURN SELF;

END;

The NEW expression can be used to invoke the system-supplied constructor function;

For example:

SET p = NEW PersonType();

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

Other UDT methods

- Instances of UDTs can be constrained to exhibit specified ordering properties.
- The EQUALS ONLY BY and ORDER FULL BY clauses may be used to specify typespecific functions for comparing UDT instances.
- The ordering can be performed using methods that are qualified as:
- RELATIVE. The relative method is a function that returns a 0 for equals, a negative value for less than, and a positive value for greater than.
- MAP. The map method uses a function that takes a single argument of the UDT type and returns a predefined data type. Comparing two UDTs is achieved by comparing the two map values associated with them.
- STATE. The state method compares the attributes of the operands to determine an order.

Definition of a new UDT

```
CREATE TYPE PersonType AS (
            dateOfBirth     DATE,
            fName           VARCHAR(15),
            lName           VARCHAR(15),
            sex             CHAR)
INSTANTIABLE
NOT FINAL
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;
```

- 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, only from one of its subtypes.
- The keyword NOT FINAL indicates that we can create subtypes of this user-defined type.

Subtypes and Supertypes:

- SQL:2011 allows UDTs to participate in a subtype/supertype hierarchy using the UNDER clause.
- A type can have more than one subtype but currently only one supertype (that is, multiple inheritance is not supported).
- A subtype inherits all the attributes and behavior (methods) of its supertype and it can define additional attributes and methods like any other UDT and it can override inherited methods.

EXAMPLE **Creation of a subtype using the UNDER clause**

To create a subtype StaffType of the supertype PersonType, we write:

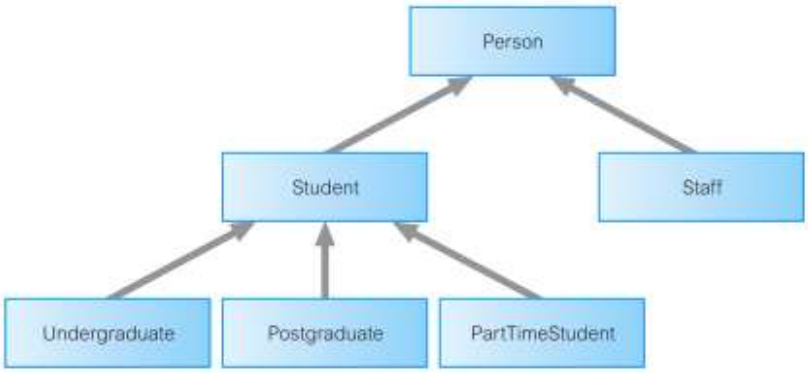
```
CREATE TYPE StaffType UNDER PersonType AS (
        staffNo      VARCHAR(5),
        position     VARCHAR(10)      DEFAULT 'Assistant',
        salary       DECIMAL(7, 2),
        branchNo     CHAR(4))
        INSTANTIABLE
        NOT FINAL
        INSTANCE METHOD isManager () RETURNS BOOLEAN;
        CREATE INSTANCE METHOD isManager() RETURNS BOOLEAN
        FOR StaffType
        BEGIN
                IF SELF.position = 'Manager' THEN
                   RETURN TRUE;
                ELSE
                   RETURN FALSE;
                END IF
        END)
```

StaffType as well as having the attributes defined within the CREATE TYPE, also includes the inherited attributes of PersonType, along with the associated observer and mutator functions and any specified methods. In particular, the clause REF IS SYSTEM GENERATED is also in effect inherited. In addition, we have defined an instance method isManager that checks whether the specified member of staff is a Manager. We show how this method can be used in Section 9.5.8.

- An instance of a subtype is considered an instance of all its supertypes.
- SQL:2011 supports the concept of substitutability: that is, whenever an instance of a supertype is expected an instance of the subtype can be used in its place.
- The type of a UDT can be tested using the TYPE predicate.

```
TYPE Udt1 IS OF (PersonType)          // Check Udt1 is the PersonType or any of
                                         its subtypes
TYPE Udt1 IS OF (ONLY PersonType)     // Check Udt1 is the PersonType
```

- In SQL:2011, as in most programming languages, every instance of a UDT must be associated with exactly one most specific type, which corresponds to the lowest subtype assigned to the instance.
- Thus, if the UDT has more than one direct supertype, then there must be a single type to which the instance belongs, and that single type must be a subtype of all the types to which the instance belongs.



(a)

- For example, a type hierarchy might consist of a maximal supertype Person, with Student and Staff as subtypes; Student itself might have three direct subtypes: Undergraduate, Postgraduate, and PartTimeStudent.
- If an instance has the type Person and Student, then the most specific type in this case is Student, a nonleaf type, since Student is a subtype of Person.

Privileges:

- To create a subtype, a user must have the UNDER privilege on the user-defined type specified as a supertype in the subtype definition.
- Prior to SQL:1999, the SELECT privilege applied only to columns of tables and views. From SQL:1999, the SELECT privilege also applies to structured types, but only when instances of those types are stored in typed tables and only when the dereference operator is used from a REF value to the referenced row and then invokes a method on that referenced row.
- When invoking a method on a structured value that is stored in a column of any ordinary SQL table, SELECT privilege is required on that column.
- If the method is a mutator function, UPDATE privilege is also required on the column. In addition, EXECUTE privilege is required on all methods that are invoked.

USER-DEFINED ROUTINES:

User-defined routines (UDRs) define methods for manipulating data and are an important adjunct to UDTs providing the required behavior for the UDTs.

- In SQL:2011, 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. It may be externally provided in a standard programming language such as C, C++, or Java.
- An **SQL-invoked procedure** is invoked from an SQL CALL statement. It may have zero or more parameters, each of which may be an input parameter (IN), an output parameter (OUT), or both an input and output parameter (INOUT), and it has a body if it is defined fully within SQL.
- An **SQL-invoked function** returns a value; any specified parameters must be input parameters. One input parameter can be designated as the result (using the RESULT keyword), in which case the parameter's data type must match the type of the RETURNS type. Such a function is called type preserving.
- An **SQL-invoked method** is similar to a function but has some important differences:
  • a method is associated with a single UDT;

  • the signature of every method associated with a UDT must be specified in that UDT and the definition of the method must specify that UDT.

There are three types of methods:

• **constructor methods,** which initialize a newly created instance of a UDT;

• **instance methods,** which operate on specific instances of a UDT;

• **static methods,** which are analogous to class methods in some object-oriented programming languages and operate at the UDT level rather than at the instance level

A method can be invoked in one of three ways:

• a **constructor method** is invoked using the NEW expression, as discussed previously;

• an **instance method** is invoked using the standard dot notation;

for example, p.fName, or using the generalized invocation format, for example, (p AS StaffType).fName();

•a **static method** is invoked using ::, for example, if totalStaff is a static method of StaffType, we could invoke it as StaffType::totalStaff().

An external routine is defined by specifying an external clause that identifies the corresponding "compiled code" in the operating system's file storage.

To create a thumbnail image for an object stored in the database, use the following CREATE FUNCTION statement with an EXTERNAL clause,

```
CREATE FUNCTION thumbnail(IN myImage ImageType) RETURNS BOOLEAN
EXTERNAL NAME '/usr/dreamhome/bin/images/thumbnail'
LANGUAGE C
PARAMETER STYLE GENERAL
DETERMINISTIC
NO SQL;
```

COLLECTION TYPES

- Collections are type constructors that are used to define collections of other types.
- Collections are used to store multiple values in a single column of a table and can result in nested tables where a column in one table actually contains another table.
- Each collection must be homogeneous: all elements must be of the same type, or at least from the same type hierarchy

TYPES:

- ARRAY—one-dimensional array with a maximum number of elements;
- MULTISET—unordered collection that does allow duplicates;
- LIST—ordered collection that allows duplicates;
- SET—unordered collection that does not allow duplicates.

ARRAY Collection Types

- An array is an ordered collection of not necessarily distinct values, whose elements are referenced by their ordinal position in the array. An array is declared by a data type and optionally a maximum cardinality;
- The elements of this array can be accessed by an index ranging from 1 to the maximum cardinality.
- To model the requirement that a branch has up to three telephone numbers, we could implement the column as an ARRAY collection type:

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

We could now retrieve the first and last telephone numbers at branch B003 using the following query:

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

MULTISET Collection Type

- A multiset is an unordered collection of elements, all of the same type, with duplicates permitted.
- Because a multiset is unordered there is no ordinal position to reference individual elements of a multiset.
- Unlike arrays, a multiset is an unbounded collection with no declared maximum cardinality

MULTISET & SET:

- There is currently no separate type proposed for sets. Instead, a set is simply a special kind of multiset: one that has no duplicate elements.
- A predicate is provided to check whether a multiset is a set.
- A multiset type constructor can be defined by enumerating their elements as a comma-separated list enclosed in square brackets.

Operations on MULTISET

- The SET function, to remove duplicates from a multiset to produce a set.
- The CARDINALITY function, to return the number of current elements.
- The ELEMENT function, to return the element of a multiset if the multiset only has one element (or null if the multiset has no elements). An exception is raised if the multiset has more than one element.

- MULTISET UNION, which computes the union of two multisets; the keywords ALL or DISTINCT can be specified to either retain duplicates or remove them.
- MULTISET INTERSECT, which computes the intersection of two multisets; the keyword DISTINCT can be specified to remove duplicates; the keyword ALL can be specified to place in the result as many instances of each value as the minimum number of instances of that value in either operand.

Aggregate functions for MULTISET

- COLLECT, which creates a multiset from the value of the argument in each row of a group;
- FUSION, which creates a multiset union of a multiset value in all rows of a group;
- INTERSECTION, which creates the multiset intersection of a multiset value in all rows of a group.

Example 1:

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,publish Publisher,Keyword_set VARCHAR(20) MULTISET_;

CREATE TABLE books OF Book;

INSERT INTO books VALUES('Compilers',ARRAY['Smith','Jones'],NEW Publisher('McGrah','New York'),MULTISET['Parsing','Analysis']);

Example 2:

propertyNo	viewDates
PA14	MULTISET['14-May-13', '24-May-13']
PG4	MULTISET['20-Apr-13', '14-May-13', '26-May-13']
PG36	MULTISET['28-Apr-13', '14-May-13']
PL94	Null

```
SELECT FUSION(viewDates) AS viewDateFusion,
       INTERSECTION(viewDates) AS viewDateIntersection
FROM PropertyViewDates;
```

Output:

viewDateFusion	viewDateIntersection
MULTISET['14-May-13', '14-May-13', '14-May-13', '24-May-13', '20-Apr-13', '26-May-13', '28-Apr-13']	MULTISET['14-May-13']

OBJECT QUERY LANGUAGE

Object Query Language (OQL) is a version of the Structured Query Language (SQL) that has been designed for use in Network Manager. It used to create new databases or insert data into existing databases (to configure the operation of Network Manager components) by amending the component schema files.

- The query language OQL was deliberately designed to have syntax similar to SQL to make it easy for users familiar with SQL to learn OQL.
- Use OQL to create new databases or insert data into existing databases (to configure the operation of Network Manager components) by amending the component schema files.

General rules of OQL:

✓ All complete statements must be terminated by a semi-colon.
✓ A list of entries in OQL is usually separated by commas but *not* terminated by a comma.
✓ Strings of text are enclosed by matching quotation marks.

Difference between OQL and SQL:

- OQL supports object referencing within tables. Objects can be nested within objects.
- Not all SQL keywords are supported within OQL. Keywords that are not relevant to Network Manager have been removed from the syntax.
- OQL can perform mathematical computations within OQL statements.

OQL can be used both for both associative and navigational access:

• An **associative query** returns a collection of objects. How these objects are located is the responsibility of the ODMS, rather than the application program.

• A **navigational query** accesses individual objects and object relationships are used to navigate from one object to another. It is the responsibility of the application program to specify the procedure for accessing the required objects.

An OQL query is a function that delivers an object whose type may be inferred from the operator contributing to the query expression.

Example:

Let us begin with a query that finds pairs of movies and theatres such that the movie is shown at the theatre and the theatre is showing more than one movie:

```
SELECT mname: M.movieName, tname: T.theatreName
                FROM  Movies M, M.shownAt T
                WHERE T.numshowing() > 1
```

The SELECT clause indicates how we can give names to fields in the result: The two result fields are called *mname* and *tname*. The part of this query that differs from SQL is from FROM clause. The variable M is bound in turn to each movie in the extent Movies. For a given movie M, we bind the variable T in turn to each theatre in the collection *M.shownAt.* Thus, the use of the path expression *M.shownAt* allows us to easily express a nested query. The following query illustrates the grouping construct in OQL:

```
SELECT T.ticketPrice,avgNum: AVG(SELECT P.T.numshowing() FROM
partition P) FROM  Theatres T   GROUP BY T.ticketPrice
```

For each ticket price, we create a group of theatres with that ticket price. This group of theatres is the partition for that ticket price, referred to using the OQL keyword partition. In the SELECT clause, for each ticket price, we compute the average number of movies shown at theatres in the partition for that ticketPrice. OQL supports an interesting variation of the grouping operation that is missing in SQL:

```
SELECT low, high ,avgNum: AVG(SELECT P.T.numshowing() FROM
partition P) FROM  Theatres T GROUP BY low:  T.ticketPrice < 5,  high:  T.ticketPrice
>=5
```

- The GROUP BY clause now creates just two partitions called *low* and *high.* Each theatre object T is placed in one of these partitions based on its ticket price.
- In the SELECT clause, *low* and *high* are Boolean variables, exactly one of which is true in any given output tuple; partition is instantiated to the corresponding partition of theatre objects.
- In our example, we get two result tuples. One of them has *low* equal to true and avgNum equal to the average number of movies shown at theatres with a low ticket price. The second tuple has *high* equal to true and avgNum equal to the average number of movies shown at theatres with a high ticket price.

The next query illustrates OQL support for queries that return collections other than set and multiset:

```
( SELECT T.theatreName
  FROM  Theatres T
  ORDER BY T.ticketPrice DESC) [0:4]
```

The ORDER BY clause makes the result a list of theatre names ordered by ticket price. The elements of a list can be referred to by position, starting with position 0. Therefore, the expression [0:4] extracts a list containing the names of the five theatres with the highest ticket prices.

OQL also supports DISTINCT, HAVING, explicit nesting of subqueries, view definitions, and other SQL features.

Types of EXPRESSIONS:

1. **Query definition expression** A query definition expression is of the form: DEFINE Q AS e. This defines a named query (that is, view) with name Q given a query expression e.
2. **Elementary expressions** An expression can be:
   - an atomic literal, for example, 10, 16.2, x, "abcde", true, nil, date 2012-12-01':
   - a named object, for example, the extent of the Branch class, branch Offices, is an expression that returns the set of all branch offices;
   - an iterator variable from the FROM clause of a SELECT-FROM-WHERE state ment, for example,

e AS x or ex or x IN e

where e is of type collection (T), then x is of type T(we discuss the OQL SELECT statement shortly);

- a query definition expression (Q previously)

3. **Construction expressions**

   If T is a type name with properties p... p. and e..., c, are expressions, then T(p1:e,...pn:en) is an expression of type T. For example, to create a Manager object, we could use the following expression:

Manager(staffNo: "SL21", fName: "John", lName: "White“ ,address: "19 Taylor St. London",position:"Manager",sex: "M",DOB: date 1945-10-01", salary: (30000)

- Similarly, we can construct expressions using struct, Set, List, Bag, and A example: struct (branch No: "B0037, stroot: "163 Main St“) is an expression, which dynamically creates an instance of this type.

4. **Atomic type expressions**

Expressions can be formed using the standard unary and binary operations on expressions. Further, if S is a string, expressions can beformed using:

- standard unary and binary operators, such as **not, abs**, +,-=,>, **andthen**, **and,orelse**, **or**.
- the string concatenation operation (||or +)
- a string offset Si (where i is an integer) meaning the i+1th character of the string:
- S[low up] meaning the substring of S from the low+1 th to ap+1 th character:
- “c in S" (where c is a character), returning a boolean true expression if the character c is in S
- "S like pattern," where pattern contains the characters "?" or”_” meaning any character, or the wildcard characters“*” or “%", meaning any substring includ ing the empty string. This returns a boolean true expression if S matches the pattern

5. **Object expressions**

Expressions can be formed using the equality and inequality operations ("=" and "!=“), returning a boolean value. If e is an expression of a type having an attribute or a relationship

p of type T. then we can extract the attribute or traverse the relationship using the expressions ep and e→p, which are of type T.

In a same way, methods can be invoked to return an expression. If the method has no parameters, the brackets in the method call can be omitted. For example, the method getAge() of the class Staff can be invoked as getAge (without the brackets).

6. Collections expressions

Expressions can be formed using universal quantifica tion (FOR ALL), existential quantification (EXISTS), membership testing (IN), select clause (SELECT FROM WHERE), order-by operator (ORDER BY), unary set operators (MIN. MAX, COUNT, SUM, AVG), and the group-by operator (GROUP BY). For example,

FOR ALL x IN managers: x.salary > 12000

returns true for all the objects in the extent managers with a salary greater than£12.000. The expression:

EXISTS x IN managers.manages: x.addres.city="London“;

returns true if there is at least one branch in London (managers manages returns a Branch object and we then check whether the city attribute of this object contains thevalue London)

7. Conversion expressions

- If e is an expression, then element(e) is an expression that checks e is a singleton,raising an exception if it is not.
- If e is a list expression, then listtoset(e) is an expression that converts the list into a set
- If e is a collection-valued expression, then flatten(e) is an expression that converts a collection of collections into a collection, that is, it flattens the structure.
- If e is an expression and c is a type name, then c(e) is an expression that asserts e is an object of type e, raising an exception if it is not.

8. Indexed collections expressions

If e1,e2 are lists or arrays and e3,e4 are integers.,then e1[e3],e1[e3:e4] first(e1), last(e1), and (e1, + e2) are expressions.

9. Binary set expressions

If e1,e2 are sets or bags, then the set operators union, except, and intersect of e1, and e2, are expressions.

Examples:

(1)Get the set of all staff who work in London (without identity).

```
DEFINE Londoners AS
 SELECT s
FROM s IN salesStaff
WHERE s.WorksAt.address.city =“London”;
SELECT s.name.lName FROM s IN Londoners;
```

(2) Get the structured set (without identity) containing the name, sex, and age of all sales staff who work in London.

```
SELECT struct (lName: s.name.lName, sex: s.sex, age: s.getAge)
FROM s IN salesStaff
WHERE s.WorksAt.address.city = “London”;
```

(3) Get the structured set (with identity) containing the name, sex, and age of all deputy managers over 60.

```
class Deputy {attribute string lName; attribute sexType sex; attribute integer age;} ;
typedef bag <Deputy> Deputies;
Deputies (SELECT Deputy (lName: s.name.lName, sex: s.sex, age: s.getAge)
  FROM s IN staffStaff
WHERE position = “Deputy” AND s.getAge > 60);
```

(4)Determine the number of sales staff at each branch.

```
SELECT struct(branchNumber, numberOfStaff: COUNT(partition))
FROM s IN salesStaff
GROUP BY branchNumber: s.WorksAt.branchNo;
```

NO-SQL:

- It is a non-relational database. They are databases that store data in a format other than relational tables.
- It come in a variety of types based on their data model.
- The main types are document, key-value, wide-column, and graph.
- They scale easily with large amounts of data and high user loads.

Features:

- Flexible schemas
- Horizontal scaling
- Fast queries due to the data model
- Ease of use for developers

Types of NoSQL databases:

4 Types:

i) Document databases store data in documents similar to JSON (JavaScript Object Notation) objects. Each document contains pairs of fields and values. The values can typically be a variety of types including things like strings, numbers, booleans, arrays, or objects.

ii) **Key-value databases** are a simpler type of database where each item contains keys and values.

iii)**Wide-column stores** store data in tables, rows, and dynamic columns.

iv)**Graph databases** store data in nodes and edges. Nodes typically store information about people, places, and things, while edges store information about the relationships between the nodes.

RDBMS vs NoSQL: Data Modeling Example

Let's consider an example of storing information about a user and their hobbies. We need to store a user's first name, last name, cell phone number, city, and hobbies.

In a relational database, we'd likely create two tables: one for Users and one for Hobbies.

Users

ID	first_name	last_name	cell	city
1	Leslie	Yepp	8125552344	Pawnee

Hobbies

ID	user_id	hobby
10	1	scrapbooking
11	1	eating waffles
12	1	working

In order to retrieve all of the information about a user and their hobbies, information from the Users table and Hobbies table will need to be joined together.

In document database like MongoDB, it store the same information about a user and their hobbies in a document as shown below

```
{
   "_id": 1,
   "first_name": "Leslie",
   "last_name": "Yepp",
   "cell": "8125552344",
   "city": "Pawnee",
   "hobbies": ["scrapbooking", "eating waffles", "working"]
}
```

In order to retrieve all of the information about a user and their hobbies, a single document can be retrieved from the database. No joins are required, resulting in faster queries.

CAP theorem:

- The CAP theorem is used to **makes system designers aware of the trade-offs while designing networked shared-data systems.**
- CAP theorem has influenced the design of many distributed data systems.
- It is very important to understand the CAP theorem as It makes the basics of choosing any NoSQL database based on the requirements.

CAP theorem states that in networked shared-data systems or distributed systems, we can only achieve at most two out of three guarantees for a database: **C**onsistency, **A**vailability and **P**artition Tolerance.

A **distributed system** is a network that stores data on more than one node (physical or virtual machines) at the same time.

Let's first understand C, A, and P in simple words:

Consistency: means that all clients see the same data at the same time, no matter which node they connect to in a distributed system. To achieve consistency, whenever data is written to one node, it must be instantly forwarded or replicated to all the other nodes in the system before the write is deemed successful.

Availability: means that every non-failing node returns a response for all read and write requests in a reasonable amount of time, even if one or more nodes are down. Another way to state this — all working nodes in the distributed system return a valid response for any request, without failing or exception.

Partition Tolerance: means that the system continues to operate despite arbitrary message loss or failure of part of the system. In other words, even if there is a network outage in the data center and some of the computers are unreachable, still the system continues to perform. Distributed systems guaranteeing partition tolerance can gracefully recover from partitions once the partition heals.

3 Categories of CAP theorem:

CP (Consistent and Partition Tolerant) database: A CP database delivers consistency and partition tolerance at the expense of availability. When a partition occurs between any two nodes, the system has to shut down the non-consistent node (i.e., make it unavailable) until the partition is resolved.

Partition *refers to a communication break between nodes within a distributed system. Meaning, if a node cannot receive any messages from another node in the system, there is a partition between the two nodes. Partition could have been because of network failure, server crash, or any other reason.*

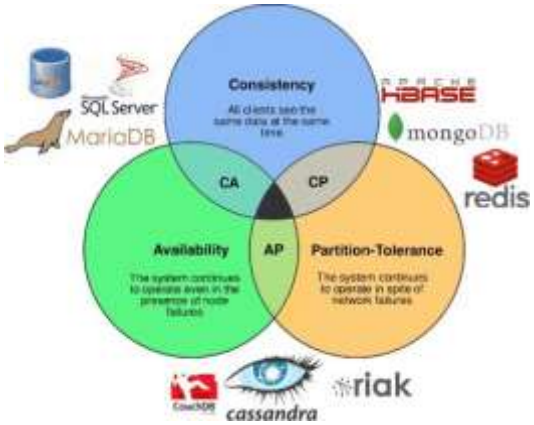
AP (Available and Partition Tolerant) database: An AP database delivers availability and partition tolerance at the expense of consistency. When a partition occurs, all nodes remain available but those at the wrong end of a partition might return an older version of data than

others. When the partition is resolved, the AP databases typically resync the nodes to repair all inconsistencies in the system.

CA (Consistent and Available) database: A CA delivers consistency and availability in the absence of any network partition. Often a single node's DB servers are categorized as CA systems. Single node DB servers do not need to deal with partition tolerance and are thus considered CA systems.

In any networked shared-data systems or distributed systems partition tolerance is a must. Network partitions and dropped messages are a fact of life and must be handled appropriately. Consequently, system designers must choose between consistency and availability.

The following diagram shows the classification of different databases based on the CAP theorem.



System designers must take into consideration the CAP theorem while designing or choosing distributed storages as one needs to be sacrificed from C and A for others.

DOCUMENT-BASED: MONGODB DATA MODEL:

- Data in MongoDB has a flexible schema.documents in the same collection.
- They do not need to have the same set of fields or structure Common fields in a collection's documents may hold different types of data.
- MongoDB provides two types of data models:

  — Embedded data model and

  — Normalized data model.

Embedded data model:

In this model, we can have (embed) all the related data in a single document, it is also known as de-normalized data model.

```
{
        _id: ,
        Emp_ID: "10025AE336"
        Personal_details:{
                First_Name: "Radhika",
                Last_Name: "Sharma",
                Date_Of_Birth: "1995-09-26"
        },
        Contact: {
                e-mail: "radhika_sharma.123@gmail.com",
                phone: "9848022338"
        },
        Address: {
                city: "Hyderabad",
                Area: "Madapur",
                State: "Telangana"
        }
}
```

For example, assume we are getting the details of employees in three different documents namely, Personal_details, Contact and, Address, you can embed all the three documents in a single one as shown −

Normalized data model:

In this model, you can refer the sub documents in the original document, using references. For example, you can re-write the above document in the normalized model as:

Employee:

```
{
        _id: <ObjectId101>,
        Emp_ID: "10025AE336"
}
```

Personal_details:

```
{
        _id: <ObjectId102>,
        empDocID: " ObjectId101",
        First_Name: "Radhika",
        Last_Name: "Sharma",
        Date_Of_Birth: "1995-09-26"
}
```

Contact:

```
{
        _id: <ObjectId103>,
        empDocID: " ObjectId101",
        e-mail: "radhika_sharma.123@gmail.com",
        phone: "9848022338"
}
```

Address:

```
{
        _id: <ObjectId104>,
        empDocID: " ObjectId101",
        city: "Hyderabad",
        Area: "Madapur",
        State: "Telangana"
}
```

Considerations while designing Schema in MongoDB:

- Design your schema according to user requirements.
- Combine objects into one document if you will use them together. Otherwise separate them (but make sure there should not be need of joins).
- Duplicate the data (but limited) because disk space is cheap as compare to compute time.
- Do joins while write, not on read.
- Optimize your schema for most frequent use cases.
- Do complex aggregation in the schema.

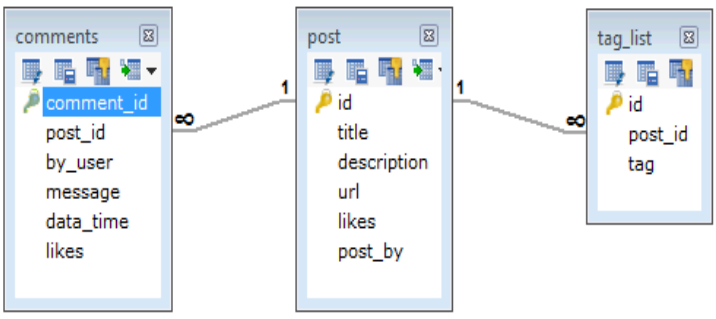
Example:

Suppose a client needs a **database design for his blog/website** and see the differences between RDBMS and MongoDB schema design.

Website has the following requirements.

- Every post has the unique title, description and url.
- Every post can have one or more tags.
- Every post has the name of its publisher and total number of likes.
- Every post has comments given by users along with their name, message, data-time and likes.
- On each post, there can be zero or more comments.

In RDBMS schema, design for above requirements will have minimum three tables.



In MongoDB schema, design will have one collection post and the following structure –

```
{
   _id: POST_ID
   title: TITLE_OF_POST,
   description: POST_DESCRIPTION,
   by: POST_BY,
   url: URL_OF_POST,
   tags: [TAG1, TAG2, TAG3],
   likes: TOTAL_LIKES,
   comments: [
      {
         user:'COMMENT_BY',
         message: TEXT,
         dateCreated: DATE_TIME,
         like: LIKES
      },
      {
         user:'COMMENT_BY',
         message: TEXT,
         dateCreated: DATE_TIME,
         like: LIKES
      }
   ]
}
```

So while showing the data, in RDBMS you need to join three tables and in MongoDB, data will be shown from one collection only.

MongoDB data model and CRUD operations:

- MongoDB provides a set of some basic but most essential operations that will help you to easily interact with the MongoDB server and these operations are known as **CRUD operations**.

i)Create Operations –

The create or insert operations are used to insert or add new documents in the collection

Method	Description
db.collection.insertOne()	It is used to insert a single document in the collection.
db.collection.insertMany()	It is used to insert multiple documents in the collection.

Example Create Operations –

insertOne()

```
> db.RecordsDB.insertOne({
... name: "Marsh",
... age: "6 years",
... species: "Dog",
... ownerAddress: "380 W. Fir Ave",
... chipped: true
... })
{
        "acknowledged" : true,
        "insertedId" : ObjectId("5fd989674e6b9ceb8665c57d")
}
```

insertMany()

```
db.RecordsDB.insertMany([{ name: "Marsh", age: "6 years", species: "Dog",
ownerAddress: "380 W. Fir Ave", chipped: true}, {name: "Kitana", age: "4 years",
species: "Cat", ownerAddress: "521 E. Cortland", chipped: true}])
{
        "acknowledged" : true,
        "insertedIds" : [
                ObjectId("5fd98ea9ce6e8850d88270b4"),
                ObjectId("5fd98ea9ce6e8850d88270b5")
        ]
}
```

ii) **Read Operations –**

The Read operations are used to retrieve documents from the collection, or in other words, read operations are used to query a collection for a document.

Method	Description
db.collection.find()	It is used to retrieve all documents from the collection.

Read Operations – db.collection.find()

```
db.RecordsDB.find()
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years", "species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "6 years", "species" : "Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "3 years", "species" : "Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
{ "_id" : ObjectId("5fd994efce6e8850d88270ba"), "name" : "Kevin", "age" : "8 years", "species" : "Dog", "ownerAddress" : "900 W. Wood Way", "chipped" : true }
```

```
db.RecordsDB.find({"species":"Cat"})
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years", "species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
```

iii) **Update Operations –**

The update operations are used to update or modify the existing document in the collection.

Method	Description
db.collection.updateOne()	It is used to update a single document in the collection that satisfy the given criteria.
db.collection.updateMany()	It is used to update multiple documents in the collection that satisfy the given criteria.

db.collection.replaceOne()	It is used to replace single document in the collection that satisfy the given criteria.

Update Operations – db.collection.updateOne()

```
db.RecordsDB.updateOne({name: "Marsh"}, {$set:{ownerAddress: "451 W. Coffee St. A204"}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "6 years", "species" : "Dog", "ownerAddress" : "451 W. Coffee St. A204", "chipped" : true }
```

Update Operations – db.collection.updateMany()

```
db.RecordsDB.updateMany({species:"Dog"}, {$set: {age: "5"}})
{ "acknowledged" : true, "matchedCount" : 3, "modifiedCount" : 3 }
> db.RecordsDB.find()
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years", "species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "5", "species" : "Dog", "ownerAddress" : "451 W. Coffee St. A204", "chipped" : true }
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "5", "species" : "Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
{ "_id" : ObjectId("5fd994efce6e8850d88270ba"), "name" : "Kevin", "age" : "5", "species" : "Dog", "ownerAddress" : "900 W. Wood Way", "chipped" : true }
```

Update Operations – db.collection.replaceOne()

```
db.RecordsDB.replaceOne({name: "Kevin"}, {name: "Maki"})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
> db.RecordsDB.find()
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years", "species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "5", "species" : "Dog", "ownerAddress" : "451 W. Coffee St. A204", "chipped" : true }
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "5", "species" : "Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
{ "_id" : ObjectId("5fd994efce6e8850d88270ba"), "name" : "Maki" }
```

iv) **Delete Operations –**

The delete operation are used to delete or remove the documents from a collection.

Method	Description
db.collection.deleteOne()	It is used to delete a single document from the collection that satisfy the given criteria.
db.collection.deleteMany()	It is used to delete multiple documents from the collection that satisfy the given criteria.

Delete Operations – db.collection.deleteOne():

```
db.RecordsDB.deleteOne({name:"Maki"})
{ "acknowledged" : true, "deletedCount" : 1 }
> db.RecordsDB.find()
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years",
"species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "5", "species" :
"Dog", "ownerAddress" : "451 W. Coffee St. A204", "chipped" : true }
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "5", "species" :
"Dog", "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
```

Delete Operations – db.collection.deleteMany()

```
db.RecordsDB.deleteMany({species:"Dog"})
{ "acknowledged" : true, "deletedCount" : 2 }
>db.RecordsDB.find()

{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4
years", "species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
```

COLUMN-BASED: HBASE DATA MODEL:

Hbase – Introduction:

- HBase is a distributed column-oriented database built on top of the Hadoop file system. It is an open-source project and is horizontally scalable.
- Column-oriented databases store table records in a sequence of columns, i.e. the entries in a column are stored in contiguous locations on disks.

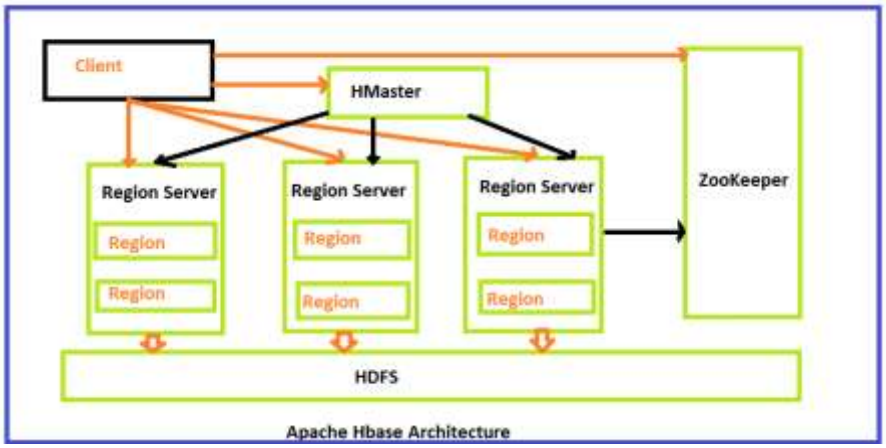
Example:

Customer ID	Name	Address	Product ID	Product Name
1	Paul Walker	US	231	Gallardo
2	Vin Diesel	Brazil	520	Mustang

Column-oriented databases store this data as: 1,2, Paul Walker, Vin Diesel, US, Brazil, 231, 520, Gallardo, Mustang

HBase Architecture:

- HBase has three major components i.e., HMaster Server, HBase Region Server, Regions and Zookeeper.



1. **HMaster**

   The implementation of Master Server in HBase is HMaster. It is a process in which regions are assigned to region server as well as DDL (create, delete table) operations. It monitor all Region Server instances present in the cluster.

2. **Region Server**

   HBase Tables are divided horizontally by row key range into Regions. **Regions** are the basic building elements of HBase cluster that consists of the distribution of tables and are comprised of Column families. Region Server runs on HDFS DataNode which is present in Hadoop cluster. Regions of Region Server are responsible for several things, like handling, managing, executing as well as reads and writes HBase operations on that set of regions. The default size of a region is 256 MB.

3. **Zookeeper**

   It is like a coordinator in HBase. It provides services like maintaining configuration information, naming, providing distributed synchronization, server failure notification etc. Clients communicate with region servers via zookeeper.

HBase Data Model

- The Data Model in HBase is designed to accommodate semi-structured data that could vary in field size, data type and columns.
- **HBase Data Model** is a set of components that consists of Tables, Rows, Column families, Cells, Columns, and Versions. HBase tables contain column families and rows with elements defined as Primary keys. A column in HBase data model table represents attributes to the objects.

HBase Data Model consists of following elements,

- Set of tables
- Each table with column families and rows
- Each table must have an element defined as Primary Key.
- Row key acts as a Primary key in HBase.
- Any access to HBase tables uses this Primary Key
- Each column present in HBase denotes attribute corresponding to object

HBase tables has following components, shown in the image below:

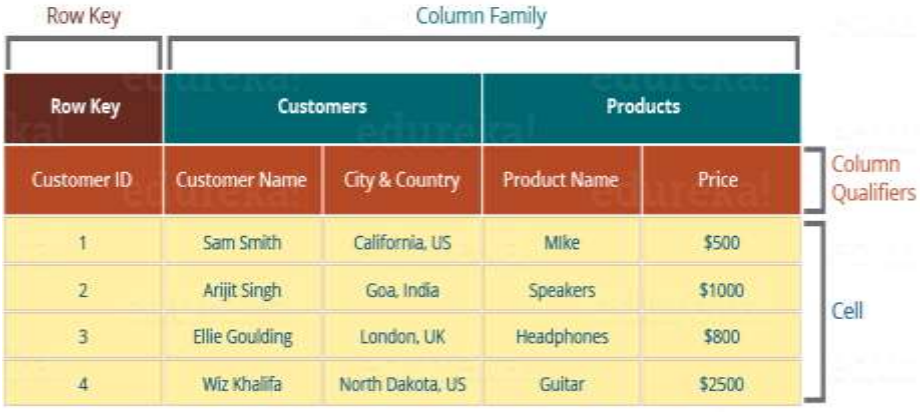


Figure: HBase Table

HBase tables has following components, shown in the image below:

Tables: Data is stored in a table format in HBase. But here tables are in column-oriented format.

Row Key: Row keys are used to search records which make searches fast. You would be curious to know how? I will explain it in the architecture part moving ahead in this blog.

Column Families: Various columns are combined in a column family. These column families are stored together which makes the searching process faster because data belonging to same column family can be accessed together in a single seek.

Column Qualifiers: Each column's name is known as its column qualifier.

Cell: Data is stored in cells. The data is dumped into cells which are specifically identified by rowkey and column qualifiers.

Timestamp: Timestamp is a combination of date and time. Whenever data is stored, it is stored with its timestamp. This makes easy to search for a particular version of data.

Hbase data model and CRUD operations:

- HBase provides shell commands to directly interact with the Database and below are a few most used shell commands.

i) CREATE:

Two-column families for table 'employee'

create 'employee', 'Personal info', 'Professional Info'

0 row(s) in 1.4750 seconds

=> Hbase::Table - employee

Put: Put command is used to insert records into HBase.

```
1  put 'employee', 1, 'Personal info:empId', 10
2  put 'employee', 1, 'Personal info:Name', 'Alex'
3  put 'employee', 1, 'Professional Info:Dept, 'IT'
```

ii) READ:

'get' and 'scan' command is used to read data from HBase.

get: 'get' operation returns a single row from the HBase table. Given below is the syntax for the 'get' method.

```
get 'table Name', 'Row Key'
```

```
hbase(main):022:get 'employee', 1
```

```
COLUMN                                  CELL
 Personal info:Name                     timestamp=1504600767520, value=Alex
 Personal info:empId                    timestamp=1504600767491, value=10
 Professional Info:Dept                 timestamp=1504600767540, value=IT
3 row(s) in 0.0250 seconds
```

scan:

'scan' command is used to retrieve multiple rows.

```
scan 'Table Name'
```

```
hbase(main):074:> scan 'employee'
```

```
ROW                                     COLUMN+CELL
 1                                      column=Personal info:Name,
timestamp=1504600767520, value=Alex
 1                                      column=Personal info:empId,
timestamp=1504606480934, value=15
 1                                      column=Professional Info:Dept,
timestamp=1504600767540, value=IT
 2                                      column=Personal info:Name,
timestamp=1504600767588, value=Bob
 2                                      column=Personal info:empId,
timestamp=1504600767568, value=20
 2                                      column=Professional Info:Dept,
timestamp=1504600768266, value=Sales
2 row(s) in 0.0500 seconds
```

(iii) **UPDATE:**

To update any record HBase uses 'put' command. To update any column value, users need to put new values and HBase will automatically update the new record with the latest timestamp.

put 'employee', 1, 'Personal info:empId', 30

(iv) **DELETE:**

'**delete**' command is used to delete individual cells of a record.

The below command is the syntax of delete command in the HBase Shell.

```
delete 'Table Name' ,'Row Key','Column Family:Column'
```

```
delete 'employee',1, 'Personal info:Name'
```

Drop Table:

To drop any table in HBase, first, it is required to disable the table. The query will return an error if the user is trying to delete the table without disabling the table. Disable removes the indexes from memory.

The below command is used to disable and drop the table.

```
disable 'employee'
```

Once the table is disabled, the user can drop using below syntax.

```
drop 'employee'
```