UNIT - V

OBJECT RELATIONAL AND NO-SQL DATABASES

Syllabus

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.

Mapping an EER Schema to an 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 relation-ships with *n* > 2. However, the operations of classes are not specified in the EER dia-gram 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.^{38} *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 avail-able 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 refer-encing 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) 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 (see Section 9.2) 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 Types

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. There are also several alias types for oid: regproc, regprocedure, regoper, regoperator, regclass, and regtype. **Table 8-19** shows an overview.
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. (Beware of possible signed-versus-unsigned confusion if you do this.)
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');
```

While that doesn't look all that bad by itself, it's still oversimplified. A far more complicated sub-select would be needed to select the right OID if there are multiple tables named mytable in different schemas. The regclass input converter handles the table lookup according to the schema path setting, and so it does the "right thing" automatically. Similarly, casting a table's OID to regclass is handy for symbolic display of a numeric OID.

Table 8-19. 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

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.

Another identifier type used by the system is xid, or transaction (abbreviated xact) identifier. This is the data type of the system columns xmin and xmax. Transaction identifiers are 32-bit quantities.

A third identifier type used by the system is cid, or command identifier. This is the data type of the system columns cmin and cmax. Command identifiers are also 32-bit quantities.

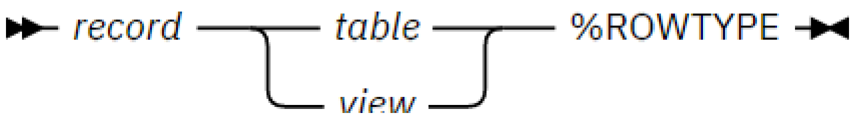
A final identifier type used by the system is tid, or tuple identifier (row identifier). This is the data type of the system column ctid. A tuple ID is a pair (block number, tuple index within block) that identifies the physical location of the row within its table.

%ROWTYPE attribute in record type declarations (PL/SQL)

The %ROWTYPE attribute, used to declare PL/SQL variables of type record with fields that correspond to the columns of a table or view, is supported by the Db2® data server. Each field in a PL/SQL record assumes the data type of the corresponding column in the table.

A *record* is a named, ordered collection of fields. A *field* is similar to a variable; it has an identifier and a data type, but it also belongs to a record, and must be referenced using dot notation, with the record name as a qualifier.

Syntax



Description

record

Specifies an identifier for the record.

table

Specifies an identifier for the table whose column definitions will be used to define the fields in the record.

view

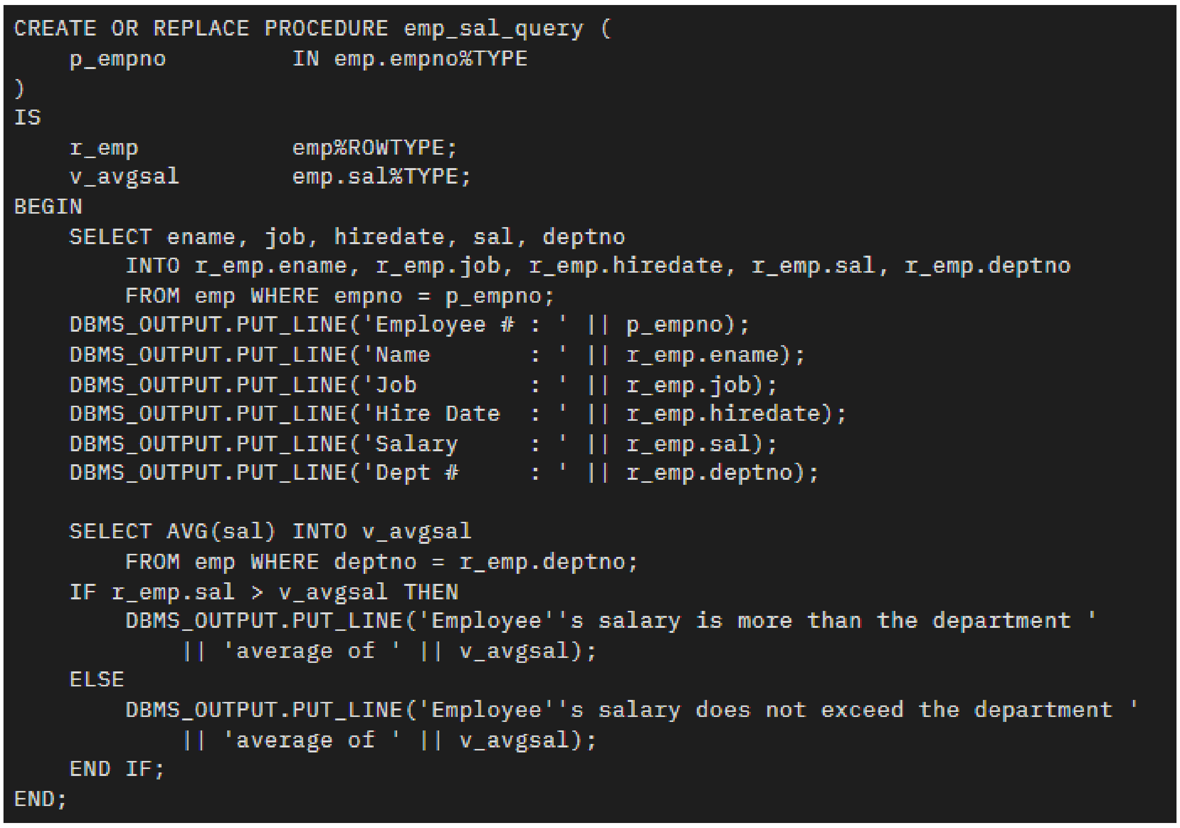
Specifies an identifier for the view whose column definitions will be used to define the fields in the record.

%ROWTYPE

Specifies that the record field data types are to be derived from the column data types that are associated with the identified table or view. Record fields do not inherit any other column attributes, such as, for example, the nullability attribute.

Example

The following example shows how to use the %ROWTYPE attribute to create a record (named r_emp) instead of declaring individual variables for the columns in the EMP table.



```
CREATE OR REPLACE PROCEDURE emp_sal_query (
    p_empno         IN emp.empno%TYPE
)
IS
    r_emp           emp%ROWTYPE;
    v_avgsal        emp.sal%TYPE;
BEGIN
    SELECT ename, job, hiredate, sal, deptno
        INTO r_emp.ename, r_emp.job, r_emp.hiredate, r_emp.sal, r_emp.deptno
        FROM emp WHERE empno = p_empno;
    DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || r_emp.ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || r_emp.job);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || r_emp.hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || r_emp.sal);
    DBMS_OUTPUT.PUT_LINE('Dept #     : ' || r_emp.deptno);

    SELECT AVG(sal) INTO v_avgsal
        FROM emp WHERE deptno = r_emp.deptno;
    IF r_emp.sal > v_avgsal THEN
        DBMS_OUTPUT.PUT_LINE('Employee''s salary is more than the department '
            || 'average of ' || v_avgsal);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee''s salary does not exceed the department '
            || 'average of ' || v_avgsal);
    END IF;
END;
```

Reference types

For every structured type you create, Db2® automatically creates a companion type. The companion type is called a reference type and the structured type to which it refers is called a referenced type. Typed tables can make special use of the reference type. You can also use reference types in SQL statements in the same way that you use other user-defined types. To use a reference type in an SQL statement, use REF(type-name), where type-name represents the referenced type.

Db2 uses the reference type as the type of the object identifier column in typed tables. The object identifier uniquely identifies a row object in the typed table hierarchy. Db2 also uses reference types to store references to rows in typed tables. You can use reference types to refer to each row object in the table.

References are strongly typed. Therefore, you must have a way to use the type in expressions. When you create the root type of a type hierarchy, you can specify the base type for a reference with the REF USING clause of the CREATE TYPE statement. The base type for a reference is called the representation type. If you do not specify the representation type with the REF USING clause, Db2 uses the default data type of VARCHAR(16) FOR BIT DATA. The representation type of the root type is inherited by all its subtypes. The REF USING clause is only valid when you define the root type of a hierarchy. In the examples used throughout this section, the representation type for the BusinessUnit_t type is INTEGER, while the representation type for Person_t is VARCHAR(13).

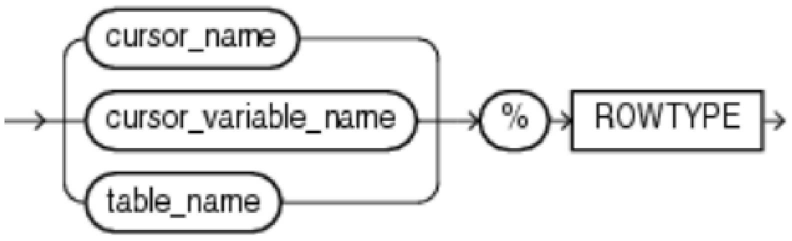
Reference types
For every structured type you create, Db2 automatically creates a companion type. The companion type is called a reference type and the structured type to which it refers is called a referenced type.
Relationships between objects in typed tables
You can define relationships between objects in one typed table and objects in another table. You can also define relationships between objects in the same typed table.
Defining semantic relationships with references
Using the WITH OPTIONS clause of CREATE TABLE, you can define that a relationship exists between a column in one table and the objects in the same or another table.
Referential integrity versus scoped references
Although scoped references do define relationships among objects in tables, they are different from referential integrity relationships.

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. Fields in a record and corresponding columns in a row have the same names and datatypes.

You can use the `%ROWTYPE` attribute in variable declarations as a datatype specifier. Variables declared using `%ROWTYPE` are treated like those declared using a datatype name. For more information, see "Using the %ROWTYPE Attribute".

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.

Examples

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;
/
```

(UDTs) User Defined Data Type:

One of the advantages of User Defined Data(UDT) is to Attach multiple data fields to a column. In Cassandra, UDTs play a vital role which allows group related fields (such that field 1, field 2, etc.) can be of data together and are named and type.
In Cassandra one of the advantage of UDTs which helps to add flexibility to your table and data model. we can construct UDT provided by Cassandra: UDT, which stands for User-Defined Type. As we can show in the example that User-defined types (UDTs) can attach multiple data fields in which each named and typed can be mapped to a single column.

To construct User Defined Type (UDT) any valid data type can be used for fields type in Cassandra, including collection (SET, LIST, MAP) or any other UDTs. Once a UDT in Cassandra is created then it can be used to define a column in the main table.

Syntax to define UDT:

```
CREATE TYPE UDT_name
(
  field_name 1 Data_Type 1,
  field_name 2 Data_Type 2,
  field_name 3 Data_Type 3,
  field_name 4 Data_Type 4,
  field_name 5 Data Type 5,
);
```

Simple steps to create UDTs:

Step-1: Create a KEYSPACE, If not existed. Syntax:

```
create_keyspace_statement ::=
      CREATE KEYSPACE [ IF NOT EXISTS ] keyspace_name
       WITH options
```

Step-2: To create keyspace used the following CQL query.

```
CREATE KEYSPACE Emp
    WITH replication = {'class': 'SimpleStrategy',
                        'replication_factor' : 1};
```

To check keyspace Schema used the following CQl query.

```
DESCRIBE KEYSPACE Emp;
```

Step-3: To Create Employee User Data Type for Current address used the following CQL query. For example, User Data Type for Current address

```
CREATE TYPE Emp.Current_add

(
Emp_id int,
h_no text,
city text,
state text,
pin_code int,
country text
);
```

Here Current_add is a Cassandra user-defined data type.

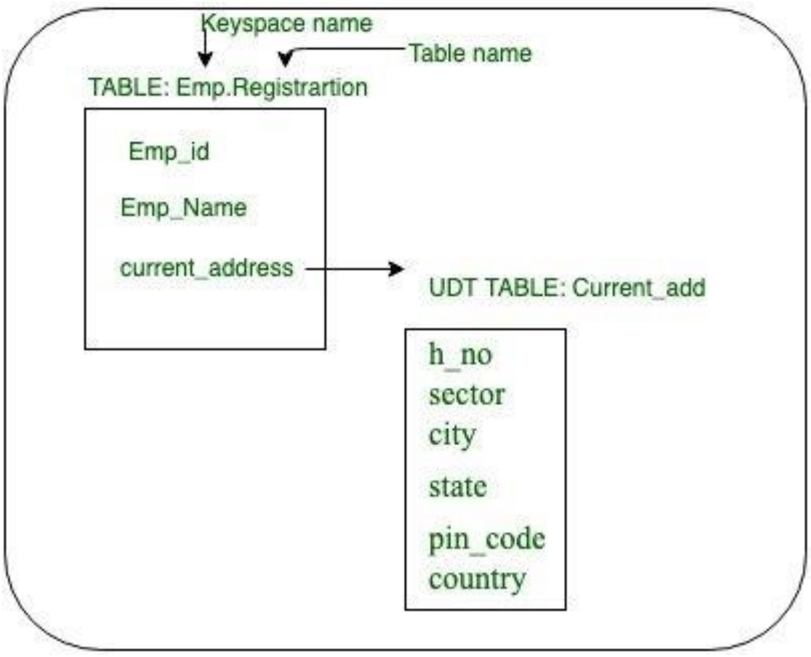


Figure – User-defined types (UDTs)

Step-4: Create table Registration that is having current_address UDT as one of the column, for example:

```
CREATE TABLE Registration

(
Emp_id int PRIMARY KEY,
Emp_Name text,
current_address FROZEN<Current_add>
);
```

Step-5: To insert data using UDT in Cassandra used the following CQL query.

Format-1: using JSON (JavaScript Object Notation) format.

In Cassandra we can also insert data in JSON (JavaScript Object Notation) format. For example: CQL query by using JSON format.

```
INSERT INTO Registration JSON'
```

```
{
"Emp_id"              : 2000,
"current_address"     :  { "h_no" : "A 210", "city" :
                          "delhi", "state" : "DL",
                          "pin_code" :"201307",
                          "country" :"india"},
"Emp_Name"            : "Ashish Rana"}' ;
```

Format-2: simple insertion

```
INSERT INTO Registration (Emp_id, Emp_Name, current_address )
            values (1000, 'Ashish', { h_no :'A 210', city :
'delhi', state : 'DL', pin_code
                                    :12345, country :'IN'});

INSERT INTO Registration(Emp_id, Emp_Name, current_address )
            values (1001, 'kartikey Rana', { h_no : 'B 210 ',  city
                                    : 'mumbai', state : 'MH',
pin_code
                                    :12345, country :'IN'});

INSERT INTO Registration(Emp_id, Emp_Name, current_address )
            values (1002, 'Dhruv Gupta', { h_no : 'C 210', city
                                    : 'delhi', state : 'DL',
pin_code
                                    :12345, country :'IN'});
```

Output:

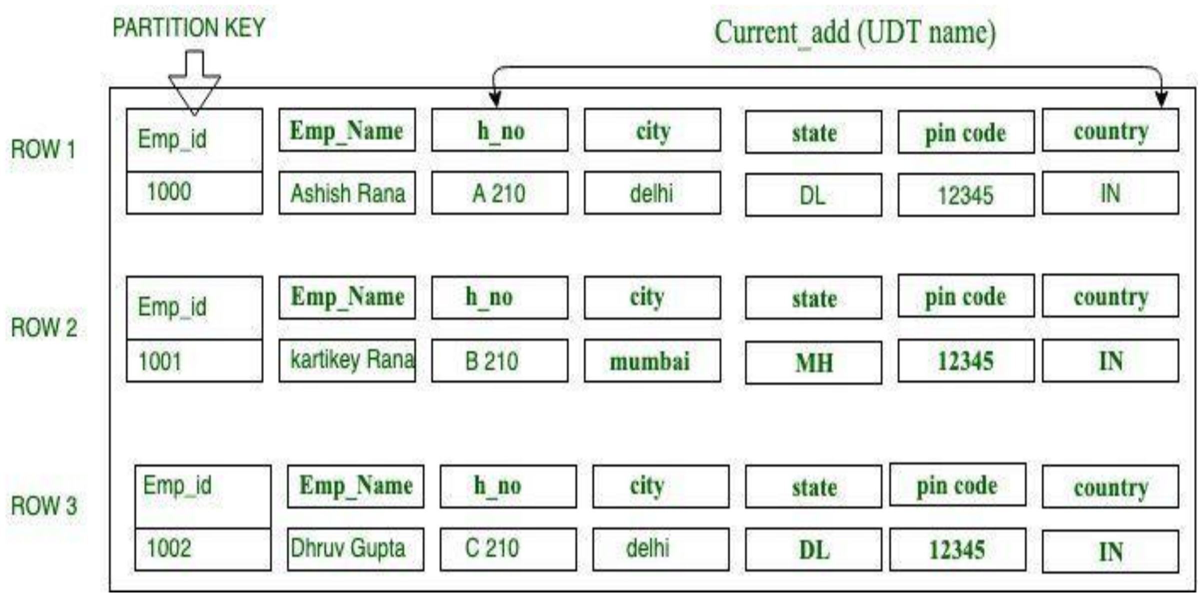


Figure – UDT Table-insertion

In this case, we will see how to insert command without inserting the value of one or more fields.
For example, we are not passing the value of the field here. How Cassandra

will handle this?
Well, it will insert this value as a normal value but it will take the field value is null. Every field value, except the primary key that we do not pass at the time of insertion, Cassandra will take it as null.

CQL query without insert one field or more field value of the UDT:

```
INSERT INTO Registration(Emp_id, Emp_Name, current_address )
                 values (1003, 'Amit Gupta', { h_no : 'D 210',
city
                               : 'Bangalore', state : 'MH',
pin_code :12345}
                               );

INSERT INTO Registration(Emp_id, Emp_Name, current_address )
                 values (1004, 'Shivang Rana', { h_no : 'E 210',
city
                               : 'Hyderabad', state : 'HYD'});
```

Output:

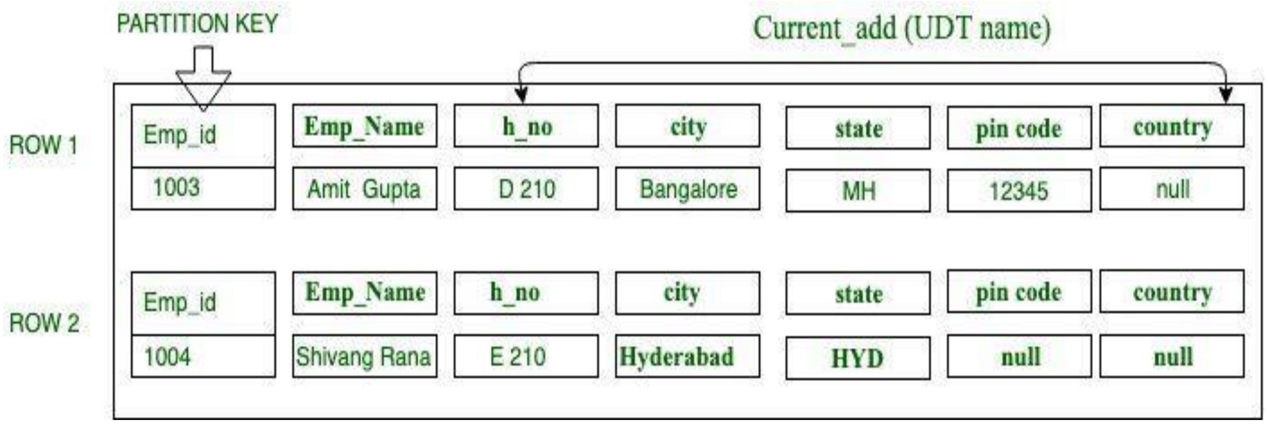


Figure – UDT table without insert one or more field value

Subtypes and Supertypes

SuperType and SubType in Data Modeling:

At times, few entities in a data model may share some common properties (attributes) within themselves apart from having one or more distinct attributes. Based on the attributes, these entities are categorized as Supertype and Subtype entities.

Supertype is an entity type that has got relationship (parent to child relationship) with one or more subtypes and it contains attributes that are common to its subtypes.

Subtypes are subgroups of the supertype entity and have unique attributes, but they will be different from each subtype.

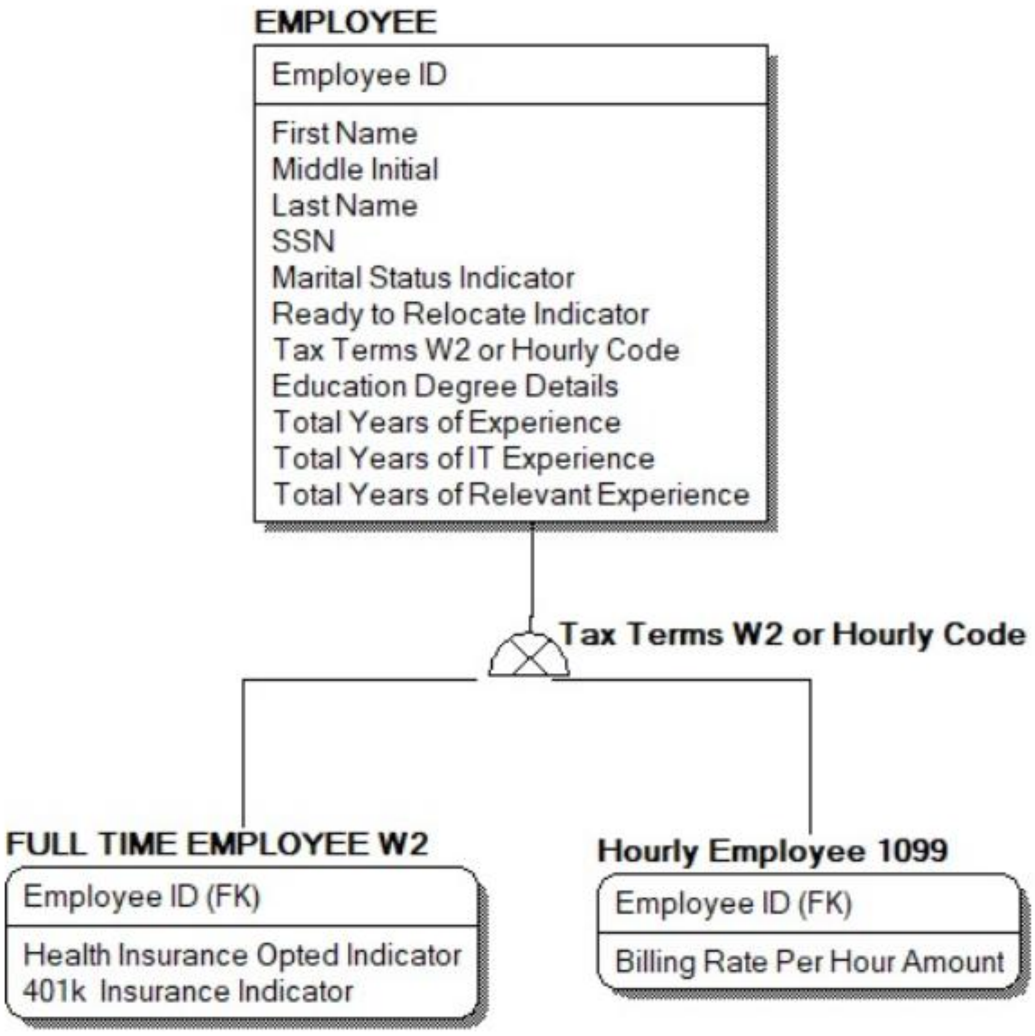
Supertypes and Subtypes are parent and child entities respectively and the primary keys of supertype and subtype are always identical.

E.g. People, Bank Account, Insurance, Asset, Liability, Credit Card.

When designing a data model for PEOPLE, you can have a supertype entity of PEOPLE and its subtype entities can be vendor, customer, and employee. People entity will have attributes like Name, Address, and Telephone number, which are common to its subtypes and you can design entities employee, vendor, and consumer with their own unique attributes. Based on this scenario, employee entity can be further classified under different subtype entities like HR employee, IT employee etc. Here employee will be the superset for the entities HR Employee and IT employee, but again it is a subtype for the PEOPLE entity.
A person can open a savings account or a certificate deposit (fixed deposit) in a bank. These accounts have attributes like account number, account opening date, account expiry date, principal amount, maturity amount, account balance, interest rate, checks issued, pre-cancellation fee etc. While designing a data model, you can create supertype parent entity as "Account" and subtype entities as Savings Account and Certificate Deposit. Account entity will store attributes like account number, interest rate that are common to savings account and certificate deposit entity. Savings account entity will have attributes like account balance and checks issued. While fixed deposit entity will have attributes like account opening, account expiry date, principal amount, maturity amount, pre-cancellation fee etc. When you design a logical data model in this manner, it provides more meaning to the business and the attributes are not cluttered in one table.

Supertype & Subtype Example:

One good example for explaining this SuperType & SubType is describing the Tax Terms related to Employees. Here Employee is the SuperType or the Parent Entity whereas the two Child Entities "**FULL TIME EMPLOYEE W2**" & "**HOURLY EMPLOYEE 1099**" are the SubTypes.


EMPLOYEE
Employee ID
First Name
Middle Initial
Last Name
SSN
Marital Status Indicator
Ready to Relocate Indicator
Tax Terms W2 or Hourly Code
Education Degree Details
Total Years of Experience
Total Years of IT Experience
Total Years of Relevant Experience
Tax Terms W2 or Hourly Code
FULL TIME EMPLOYEE W2
Employee ID (FK)
Health Insurance Opted Indicator
401k Insurance Indicator
Hourly Employee 1099
Employee ID (FK)
Billing Rate Per Hour Amount

User-defined routines

You can create routines to encapsulate logic of your own or use the database provide routines that capture the functionality of most commonly used arithmetic, string, and casting functions. The user-defined routines refer to any procedures, functions, and methods that are created by the user.

You can create your own procedures, functions and methods in any of the supported implementation styles for the routine type. Generally the prefix 'user-defined' is not used when referring to procedures and methods. User-defined functions are also commonly called UDFs.

User-defined routine creation

User-defined procedures, functions and methods are created in the database by executing the appropriate CREATE statement for the routine type. These routine creation statements include:

- CREATE PROCEDURE
- CREATE FUNCTION
- CREATE METHOD

The clauses specific to each of the CREATE statements define characteristics of the routine, such as the routine name, the number and type of routine arguments, and details about the routine logic. The database manager use the information provided by the clauses to identify and run the routine when it is invoked. Upon successful execution of the CREATE statement for a routine, the routine is created in the database. The characteristics of the routine are stored in the database catalog views that users can query. Executing the CREATE statement to create a routine is also referred to as defining a routine or registering a routine.

User-defined routine definitions are stored in the SYSTOOLS system catalog table schema.

User-defined routine logic implementation

There are three implementation styles that can be used to specify the logic of a routine:

- Sourced: user-defined routines can be *sourced* from the logic of existing built-in routines.
- SQL: user-defined routines can be implemented using only SQL statements.
- External: user-defined routines can be implemented using one of a set of supported programming languages.

  When routines are created in a non-SQL programming language, the library or class built from the code is associated with the routine definition by the value specified in the EXTERNAL NAME clause. When the routine is invoked the library or class associated with the routine is run.

User-defined routines can include a variety of SQL statements, but not all SQL statements.

User-defined routines are strongly typed, but type handling and error-handling mechanisms must be developed or enhanced by routine developers.

After a database upgrade, it may be necessary to verify or update routine implementations.

In general, user-defined routines perform well, but not as well as built-in routines.

User-defined routines can invoke built-in routines and other user-defined routines implemented in any of the supported formats. This flexibility allows users to essentially have the freedom to build a complete library of routine modules that can be re-used.

In general, user-defined routines provide a means for extending the SQL language and for modularizing logic that will be re-used by multiple queries or database applications where built-in routines do not exist.

PL/SQL - Collections

we will discuss the Collections in PL/SQL. A collection is an ordered group of elements having the same data type. Each element is identified by a unique subscript that represents its position in the collection.

PL/SQL provides three collection types −

- Index-by tables or Associative array
- Nested table
- Variable-size array or Varray

Oracle documentation provides the following characteristics for each type of collections −

Collection Type	Number of Elements	Subscript Type	Dense or Sparse	Where Created	Can Be Object Type Attribute
Associative array (or index-by table)	Unbounded	String or integer	Either	Only in PL/SQL block	No
Nested table	Unbounded	Integer	Starts dense, can become sparse	Either in PL/SQL block or at schema level	Yes
Variablesize array (Varray)	Bounded	Integer	Always dense	Either in PL/SQL block or at schema level	Yes

We have already discussed varray in the chapter **'PL/SQL arrays'**. In this chapter, we will discuss the PL/SQL tables.

Both types of PL/SQL tables, i.e., the index-by tables and the nested tables have the same structure and their rows are accessed using the subscript notation. However, these two types of tables differ in one aspect; the nested tables can be stored in a database column and the index-by tables cannot.

Index-By Table

An **index-by** table (also called an **associative array**) is a set of **key-value** pairs. Each key is unique and is used to locate the corresponding value. The key can be either an integer or a string.

An index-by table is created using the following syntax. Here, we are creating an **index-by** table named **table_name**, the keys of which will be of the subscript_type and associated values will be of the *element_type*

```
TYPE type_name IS TABLE OF element_type [NOT NULL] INDEX BY subscript_type;

table_name type_name;
```

Example

Following example shows how to create a table to store integer values along with names and later it prints the same list of names.

```
DECLARE
   TYPE salary IS TABLE OF NUMBER INDEX BY VARCHAR2(20);
   salary_list salary;
   name   VARCHAR2(20);
BEGIN
   -- adding elements to the table
   salary_list('Rajnish') := 62000;
   salary_list('Minakshi') := 75000;
   salary_list('Martin') := 100000;
   salary_list('James') := 78000;

   -- printing the table
   name := salary_list.FIRST;
   WHILE name IS NOT null LOOP
      dbms_output.put_line
      ('Salary of ' || name || ' is ' || TO_CHAR(salary_list(name)));
      name := salary_list.NEXT(name);
   END LOOP;
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Salary of James is 78000
Salary of Martin is 100000
Salary of Minakshi is 75000
Salary of Rajnish is 62000

PL/SQL procedure successfully completed.
```

Example

Elements of an index-by table could also be a **%ROWTYPE** of any database table or **%TYPE** of any database table field. The following example illustrates the concept. We will use the **CUSTOMERS** table stored in our database as −

```
Select * from customers;

+----+----------+-----+-----------+----------+
| ID | NAME | AGE | ADDRESS | SALARY |
+----+----------+-----+-----------+----------+
| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |
| 2 | Khilan | 25 | Delhi | 1500.00 |
| 3 | kaushik | 23 | Kota | 2000.00 |
| 4 | Chaitali | 25 | Mumbai | 6500.00 |
| 5 | Hardik | 27 | Bhopal | 8500.00 |
| 6 | Komal | 22 | MP | 4500.00 |
+----+----------+-----+-----------+----------+
```

```
DECLARE
   CURSOR c_customers is
      select name from customers;

   TYPE c_list IS TABLE of customers.Name%type INDEX BY binary_integer;
   name_list c_list;
   counter integer :=0;
```

```
BEGIN
  FOR n IN c_customers LOOP
    counter := counter +1;
    name_list(counter) := n.name;
    dbms_output.put_line('Customer('||counter||'):'||name_lis t(counter));
  END LOOP;
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Customer(1): Ramesh
Customer(2): Khilan
Customer(3): kaushik
Customer(4): Chaitali
Customer(5): Hardik
Customer(6): Komal

PL/SQL procedure successfully completed
```

Nested Tables

A **nested table** is like a one-dimensional array with an arbitrary number of elements. However, a nested table differs from an array in the following aspects −

- An array has a declared number of elements, but a nested table does not. The size of a nested table can increase dynamically.
- An array is always dense, i.e., it always has consecutive subscripts. A nested array is dense initially, but it can become sparse when elements are deleted from it.

A nested table is created using the following syntax −

```
TYPE type_name IS TABLE OF element_type [NOT NULL];

table_name type_name;
```

This declaration is similar to the declaration of an **index-by** table, but there is no **INDEX BY** clause.

A nested table can be stored in a database column. It can further be used for simplifying SQL operations where you join a single-column table with a larger table. An associative array cannot be stored in the database.

Example

The following examples illustrate the use of nested table −

```
DECLARE
  TYPE names_table IS TABLE OF VARCHAR2(10);
  TYPE grades IS TABLE OF INTEGER;
  names names_table;
  marks grades;
  total integer;
BEGIN
```

```
   names := names_table('Kavita', 'Pritam', 'Ayan', 'Rishav', 'Aziz');
   marks:= grades(98, 97, 78, 87, 92);
   total := names.count;
   dbms_output.put_line('Total '|| total || ' Students');
   FOR i IN 1 .. total LOOP
      dbms_output.put_line('Student:'||names(i)||', Marks:' || marks(i));
   end loop;
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Total 5 Students
Student:Kavita, Marks:98
Student:Pritam, Marks:97
Student:Ayan, Marks:78
Student:Rishav, Marks:87
Student:Aziz, Marks:92

PL/SQL procedure successfully completed.
```

Example

Elements of a **nested table** can also be a **%ROWTYPE** of any database table or %TYPE of any database table field. The following example illustrates the concept. We will use the CUSTOMERS table stored in our database as −

```
Select * from customers;
+----+----------+-----+-----------+----------+
| ID | NAME | AGE | ADDRESS | SALARY |
+----+----------+-----+-----------+----------+
| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |
| 2 | Khilan | 25 | Delhi | 1500.00 |
| 3 | kaushik | 23 | Kota | 2000.00 |
| 4 | Chaitali | 25 | Mumbai | 6500.00 |
| 5 | Hardik | 27 | Bhopal | 8500.00 |
| 6 | Komal | 22 | MP | 4500.00 |
+----+----------+-----+-----------+----------+
```

```
DECLARE
   CURSOR c_customers is
      SELECT  name FROM customers;
   TYPE c_list IS TABLE of customerS.No.ame%type;
   name_list c_list := c_list();
   counter integer :=0;
BEGIN
   FOR n IN c_customers LOOP
      counter := counter +1;
      name_list.extend;
      name_list(counter)  := n.name;
      dbms_output.put_line('Customer('||counter||'):'||name_list(counter));
   END LOOP;
END;  /
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Customer(1): Ramesh
Customer(2): Khilan
Customer(3): kaushik
Customer(4): Chaitali
Customer(5): Hardik
Customer(6): Komal

PL/SQL procedure successfully completed.
```

Collection Methods

PL/SQL provides the built-in collection methods that make collections easier to use. The following table lists the methods and their purpose −

S.No	Method Name & Purpose
1	**EXISTS(n)** Returns TRUE if the nth element in a collection exists; otherwise returns FALSE.
2	**COUNT** Returns the number of elements that a collection currently contains.
3	**LIMIT** Checks the maximum size of a collection.
4	**FIRST** Returns the first (smallest) index numbers in a collection that uses the integer subscripts.
5	**LAST** Returns the last (largest) index numbers in a collection that uses the integer subscripts.
6	**PRIOR(n)** Returns the index number that precedes index n in a collection.
7	**NEXT(n)** Returns the index number that succeeds index n.
8	**EXTEND** Appends one null element to a collection.
9	**EXTEND(n)** Appends n null elements to a collection.
10	**EXTEND(n,i)** Appends **n** copies of the ith element to a collection.
11	**TRIM** Removes one element from the end of a collection.

12	**TRIM(n)** Removes **n** elements from the end of a collection.
13	**DELETE** Removes all elements from a collection, setting COUNT to 0.
14	**DELETE(n)** Removes the **n**th element from an associative array with a numeric key or a nested table. If the associative array has a string key, the element corresponding to the key value is deleted. If **n** is null, **DELETE(n)** does nothing.
15	**DELETE(m,n)** Removes all elements in the range **m..n** from an associative array or nested table. If **m** is larger than **n** or if **m** or **n** is null, **DELETE(m,n)** does nothing.

Collection Exceptions

The following table provides the collection exceptions and when they are raised −

Collection Exception	Raised in Situations
COLLECTION_IS_NULL	You try to operate on an atomically null collection.
NO_DATA_FOUND	A subscript designates an element that was deleted, or a nonexistent element of an associative array.
SUBSCRIPT_BEYOND_COUNT	A subscript exceeds the number of elements in a collection.
SUBSCRIPT_OUTSIDE_LIMIT	A subscript is outside the allowed range.
VALUE_ERROR	A subscript is null or not convertible to the key type. This exception might occur if the key is defined as a **PLS_INTEGER** range, and the subscript is outside this range.

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. The components create and interact with their databases using 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. You can also issue OQL statements using the OQL Service Provider, for example, to create or modify databases, insert data into databases and retrieve data.

- **Conventions and sample databases**
  To illustrate the OQL keywords in use, a sample database has been used, the `staff` database, which contains three tables: `managers`, `employees`, and `contractors`.
- **Features of OQL**
  The following topics describe the features of Object Query Language (OQL).
- **Database and table creation**
  You can create databases and tables with the create command.
- **Inserting data into a table**
  Use the **`insert`** keyword to insert data into a table.
- **Selecting data from a table**
  You can query the data in a table using the **`select`** keyword. Use these examples to help you use the **`select`** keyword.
- **Counting rows in a table**
  You can count the number of rows in a table using the **`select`** keyword.
- **Conditional tests in OQL**
  Use comparison operators in OQL, for example in a **`select where`** statement, to perform conditional tests.
- **Use of select to perform subqueries**
  Subqueries are queries that are embedded within queries using double brackets [[ ]]. Any valid query can be embedded within the double brackets.
- **Selection of data into another table**
  The **`select into`** statement retrieves data from one table and inserts it into another. The select into command does not delete the existing record.
- **Updates to records in tables**
  Use the **`update`** keyword to update an existing record in a table.
- **Database and table listings**
  Use the **`show`** keyword to list the databases, columns, or tables or the current service.
- **Deletion of a record from a database table**
  You can delete a record from a table using the **`delete`** command.
- **Deletion of a database or table**
  You can delete a database or table using the **`drop`** command.
- **The eval statement**
  The eval statement is used to evaluate the value of a variable or a column within a record, and if necessary, convert it into another data type.

OQL - Object Query Language

The goal of this file is to help you get started with OQL. The examples presented in this file refer to classes defined in the file "O2 Tutorial".

Please note that the syntax in some of the examples might need minor adjustment before they will work with the current version of O2. If you find any errors, or places which are unclear, or if you have any suggestions or comments, please let us know (email to Michalis - mpetropo@cs.ucsd.edu). Your help is greatly appreciated.

Introduction

OQL is the way to access data in an O2 database. OQL is a powerful and easy-to-use SQL-like query language with special features dealing with complex objects, values and methods.

Using OQL

- Setup your environment
- SELECT, FROM, WHERE
- Dot notation and path expressions
- Subqueries in FROM clause
- Subqueries in WHERE clause
- Set operations and Aggregation
- GROUP BY
- Embedded OQL
- More documentation

Setup your environment

We've been able to create classes and write some programs. So far, O2 appears to be like an object-oriented programming language like C++ instead of a database system. Probably the main difference is that O2 supports queries. The queries that you'll be creating will look very similar to that of SQL.

In order to perform queries, you'll need to enter query mode. From within the O2 client, do the command:

- **query;**

This will put you in a sub-shell for queries. From here, you can enter your queries followed by Ctrl-D. To exit query mode, just hit Ctrl-D without entering a query.

SELECT, FROM, WHERE

```
SELECT <list of values>
FROM <list of collections and variable assignments>
WHERE <condition>
```

The SELECT clause extracts those elements of a collection meeting a specific condition. By using the keyword DISTINCT duplicated elements in the resulting collection get eliminated. Collections in FROM can be either extents (persistent names - sets) or expressions that evaluate to a collection (a set). Strings are enclosed in double-quotes in OQL. We can rename a field by if we prefix the path with the desired name and a colon.

Example Query 1
Give the names of people who are older than 26 years old:

```
SELECT SName: p.name
FROM p in People
WHERE p.age > 26
(hit Ctrl-D)
```

Dot Notation & Path Expressions

We use the dot notation and path expressions to access components of complex values.

Let variables t and ta range over objects in extents (persistent names) of Tutors and TAs (i.e., range over objects in sets Tutors and TAs).

ta.salary -> real
t.students -> set of tuples of type tuple(name: string, fee: real) representing students
t.salary -> real

Cascade of dots can be used if all names represent objects and not a collection.

Example of Illegal Use of Dot

t.students.name, where ta is a TA object.

This is illegal, because ta.students is a set of objects, not a single object.

Example Query 2
Find the names of the students of all tutors:

```
SELECT s.name
FROM Tutors t, t.students s
```

Here we notice that the variable t that binds to the first collection of FROM is used to help us define the second collection s. Because students is a collection, we use it in the FROM list, like t.students above, if we want to access attributes of students.

Subqueries in FROM Clause

Example Query 3
Give the names of the Tutors which have a salary greater than $300 and have a student paying more than $30:

```
SELECT t.name
FROM ( SELECT t FROM Tutors t WHERE t.salary > 300 ) r, r.students s
WHERE s.fee > 30
```

Subqueries in WHERE Clause

Example Query 4
Give the names of people who aren't TAs:

```
SELECT p.name
FROM p in People
WHERE not ( p.name in SELECT t.name FROM t in TAs )
```

Set Operations and Aggregation

The standard O2C operators for sets are + (union), * (intersection), and - (difference). In OQL, the operators are written as UNION, INTERSECT and EXCEPT , respectively.

Example Query 5
Give the names of TAs with the highest salary:

```
SELECT t.name
FROM t in TAs
WHERE t.salary = max ( select ta.salary from ta in TAs )
```

GROUP BY

The GROUP BY operator creates a set of tuples with two fields. The first has the type of the specified GROUP BY attribute. The second field is the set of tuples that match that attribute. By default, the second field is called PARTITION.

Example Query 6
Give the names of the students and the average fee they pay their Tutors:

```
SELECT sname, avgFee: AVG(SELECT p.s.fee FROM partition p)
FROM t in Tutors, t.students s
GROUP BY sname: s.name
```

1. Initial collection

We begin from collection Tutors, but technically it is a bag of tuples of the form:

tuple(t: t1, s: tuple(name: string, fee: real))

where t1 is a Tutor object and s denotes a student tuple. In general, there are fields for all of the variable bindings in the FROM clause.

2. Intermediate collection

The GROUP BY attribute s.name maps the tuples of the initial collection to the value of the name of the student. The intermediate collection is a set of tuples of type:

tuple(sname: string, partition: set(tuple(t: Tutor, s: tuple(name: string, fee: real))))

For example:

tuple(sname = "Mike", partition = set(tuple(t1, tuple("Mike", 10)), tuple(t2, tuple("Mike", 20))))

where t1,t2,... are all the tutors of student "Mike".

3. Output collection

Consists of student-average fee pairs, one for each tuple in the intermediate collection. The type of tuples in the output is:

tuple(sname: string, avgFee: real)

Note that in the subquery of the SELECT clause:

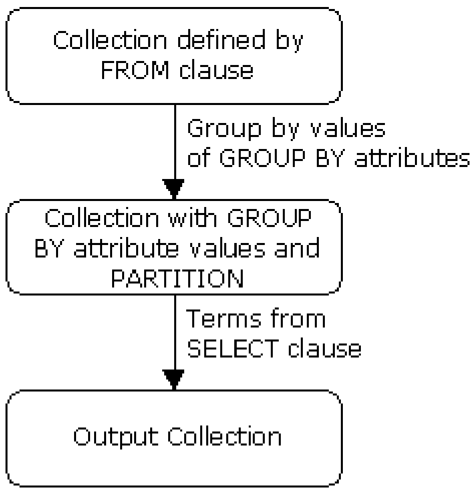
SELECT sname, avgFee: AVG(SELECT p.s.fee FROM partition p)

We let p range over all tuples in partition. Each of these tuples contains a Tutor object and a student tuple. Thus, p.s.fee extracts the fee from one of the student tuples.

A typical output tuple looks like this:

tuple(sname = "Mike", avgFee = 15)

The whole procedure of GROUP BY operator's evaluation is summarized in the following figure:


Collection defined by FROM clause
Group by values of GROUP BY attributes
Collection with GROUP BY attribute values and PARTITION
Terms from SELECT clause
Output Collection

Embedded OQL

Instead of using query mode, you can incorporate these queries in your O2 programs using the "o2query" command:

```
run body {
  o2 real total_salaries;
  o2query( total_salaries, "sum ( SELECT ta->get_salary \
  FROM ta in TAs )" );
  printf("TAs combined salary:  %.2f\n", total_salaries);
};
```

The first argument for o2query is the variable in which you want to store the query results. The second argument is a string that contains the query to be performed. If your query string takes up several lines, be sure to backslash (\) the carriage returns.

CAP Theorem and NoSQL Databases

What is the 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: Consistency, Availability and Partition 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.

The CAP theorem categorizes systems into three categories:

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.

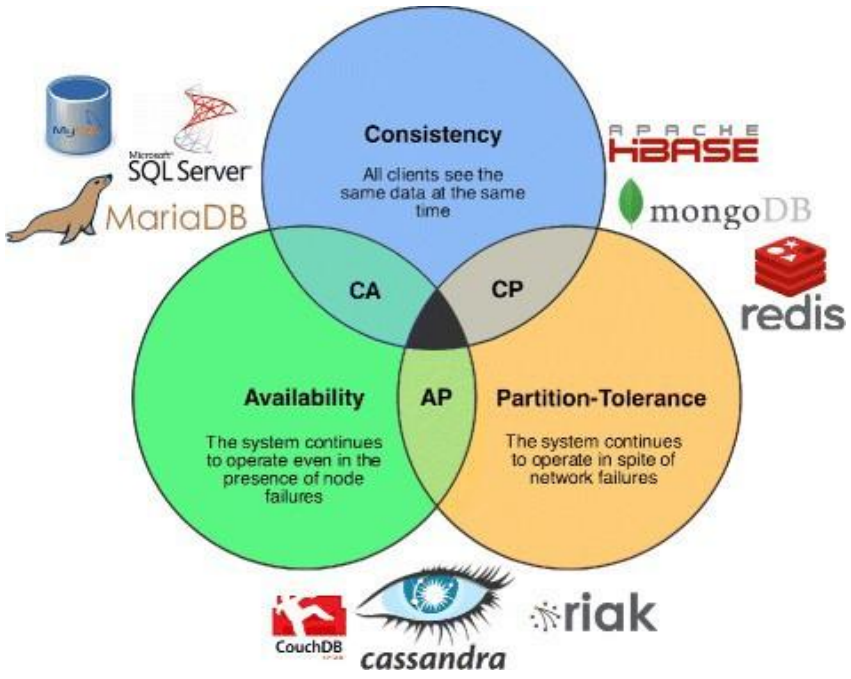
Partition *refers to a communication break between nodes within a distributed system. Meaning, if a node cannot receive any messages from another node in the system, there is a partition between the two nodes. Partition could have been because of network failure, server crash, or any other reason.*

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.


Consistency
All clients see the same data at the same time
Availability
The system continues to operate even in the presence of node failures
Partition-Tolerance
The system continues to operate in spite of network failures
CA
CP
AP
SQL Server
MariaDB
APACHE HBASE
mongoDB
redis
CouchDB
cassandra
riak

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. https://cloudxlab.com/assessment/displayslide/345/nosql-cap-theorem#:~:text=CAP%20theorem%20or%20Eric%20Brewers,data%20at%20the%20same%20time

Data Modeling in MongoDB

In MongoDB, data has a flexible schema. It is totally different from SQL database where you had to determine and declare a table's schema before inserting data. MongoDB collections do not enforce document structure.

The main challenge in data modeling is balancing the need of the application, the performance characteristics of the database engine, and the data retrieval patterns.

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.

Data Model Design

MongoDB provides two types of data models: — Embedded data model and Normalized data model. Based on the requirement, you can use either of the models while preparing your document.

Embedded Data Model

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

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 below −

```
{
		_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"
		}
}
```

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

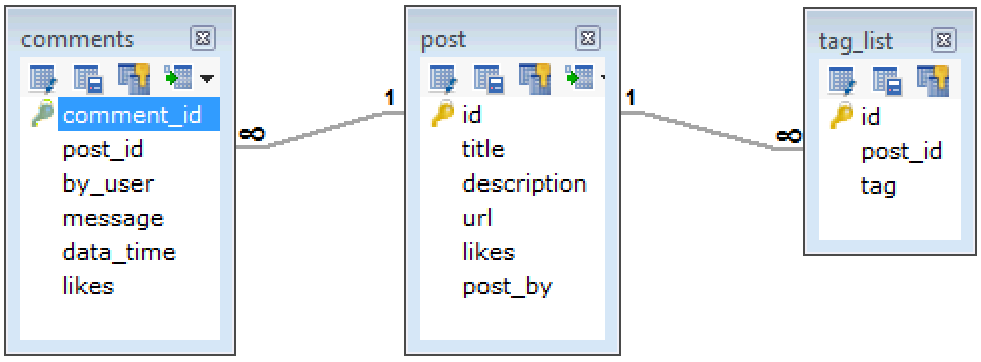
- Design your schema according to user requirements.
- Combine objects into one document if you will use them together. Otherwise separate them (but make sure there should not be need of joins).
- Duplicate the data (but limited) because disk space is cheap as compare to compute time.
- Do joins while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



While 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.

What is CRUD in MongoDB?

CRUD operations describe the conventions of a user-interface that let users view, search, and modify parts of the database.

MongoDB documents are modified by connecting to a server, querying the proper documents, and then changing the setting properties before sending the data back to the database to be updated. CRUD is data-oriented, and it’s standardized according to HTTP action verbs.

When it comes to the individual CRUD operations:

- The Create operation is used to insert new documents in the MongoDB database.
- The Read operation is used to query a document in the database.
- The Update operation is used to modify existing documents in the database.
- The Delete operation is used to remove documents in the database.

How to Perform CRUD Operations

Now that we’ve defined MongoDB CRUD operations, we can take a look at how to carry out the individual operations and manipulate documents in a MongoDB database. Let’s go into the processes of creating, reading, updating, and deleting documents, looking at each operation in turn.

Create Operations

For MongoDB CRUD, if the specified collection doesn’t exist, the create operation will create the collection when it’s executed. Create operations in MongoDB target a single collection, not multiple collections. Insert operations in MongoDB are atomic on a single document level.

MongoDB provides two different create operations that you can use to insert documents into a collection:

- db.collection.insertOne()
- db.collection.insertMany()

insertOne()
As the namesake, insertOne() allows you to insert one document into the collection. For this example, we’re going to work with a collection called RecordsDB. We can insert a single entry into our collection by calling the insertOne() method on RecordsDB. We then provide the information we want to insert in the form of key-value pairs, establishing the schema.

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

If the create operation is successful, a new document is created. The function will return an object where “acknowledged” is “true” and “insertID” is the newly created “ObjectId.”

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

insertMany()
It’s possible to insert multiple items at one time by calling the *insertMany()* method on the desired collection. In this case, we pass multiple items into our chosen collection (*RecordsDB*) and separate them by commas. Within the parentheses, we use brackets to indicate that we are passing in a list of multiple entries. This is commonly referred to as a nested method.

```
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}])
```

```
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")
        ]
}
```

Read Operations

The read operations allow you to supply special query filters and criteria that let you specify which documents you want. The MongoDB documentation contains more information on the available query filters. Query modifiers may also be used to change how many results are returned.

MongoDB has two methods of reading documents from a collection:

- db.collection.find()
- db.collection.findOne()

find()

In order to get all the documents from a collection, we can simply use the *find()* method on our chosen collection. Executing just the *find()* method with no arguments will return all records currently in the collection.

```
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 }
```

Here we can see that every record has an assigned “ObjectId” mapped to the “_id” key.

If you want to get more specific with a read operation and find a desired subsection of the records, you can use the previously mentioned filtering criteria to choose what results should be returned. One of the most common ways of filtering the results is to search by value.

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

findOne()

In order to get one document that satisfies the search criteria, we can simply use the *findOne()* method on our chosen collection. If multiple documents satisfy the query, this method returns the first document according to the natural order which reflects the order of documents on the disk. If no documents satisfy the search criteria, the function returns null. The function takes the following form of syntax.

```
db.{collection}.findOne({query}, {projection})
```

Let's take the following collection—say, *RecordsDB*, as an example.

```
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "8 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 }
```

And, we run the following line of code:

```
db.RecordsDB.find({"age":"8 years"})
```

We would get the following result:

```
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "8 years",
"species" : "Cat", "ownerAddress" : "521 E. Cortland", "chipped" : true }
```

Notice that even though two documents meet the search criteria, only the first document that matches the search condition is returned.

Update Operations

Like create operations, update operations operate on a single collection, and they are atomic at a single document level. An update operation takes filters and criteria to select the documents you want to update.

You should be careful when updating documents, as updates are permanent and can't be rolled back. This applies to delete operations as well.

For MongoDB CRUD, there are three different methods of updating documents:

- db.collection.updateOne()
- db.collection.updateMany()
- db.collection.replaceOne()

updateOne()

We can update a currently existing record and change a single document with an update operation. To do this, we use the *updateOne()* method on a chosen collection, which here is "RecordsDB." To update a document, we provide the method with two arguments: an update filter and an update action.

The update filter defines which items we want to update, and the update action defines how to update those items. We first pass in the update filter. Then, we use the "$set" key and provide the fields we want to update as a value. This method will update the first record that matches the provided filter.

```
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 }
```

updateMany()

updateMany() allows us to update multiple items by passing in a list of items, just as we did when inserting multiple items. This update operation uses the same syntax for updating a single document.

```
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 }
```

replaceOne()

The *replaceOne()* method is used to replace a single document in the specified collection. *replaceOne()* replaces the entire document, meaning fields in the old document not contained in the new will be lost.

```
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" }
```

Delete Operations

Delete operations operate on a single collection, like update and create operations. Delete operations are also atomic for a single document. You can provide delete operations with filters and criteria in order to specify which documents you would like to delete from a collection. The filter options rely on the same syntax that read operations utilize.

MongoDB has two different methods of deleting records from a collection:

- db.collection.deleteOne()
- db.collection.deleteMany()

deleteOne()

deleteOne() is used to remove a document from a specified collection on the MongoDB server. A filter criteria is used to specify the item to delete. It deletes the first record that matches the provided filter.

```
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 }
```

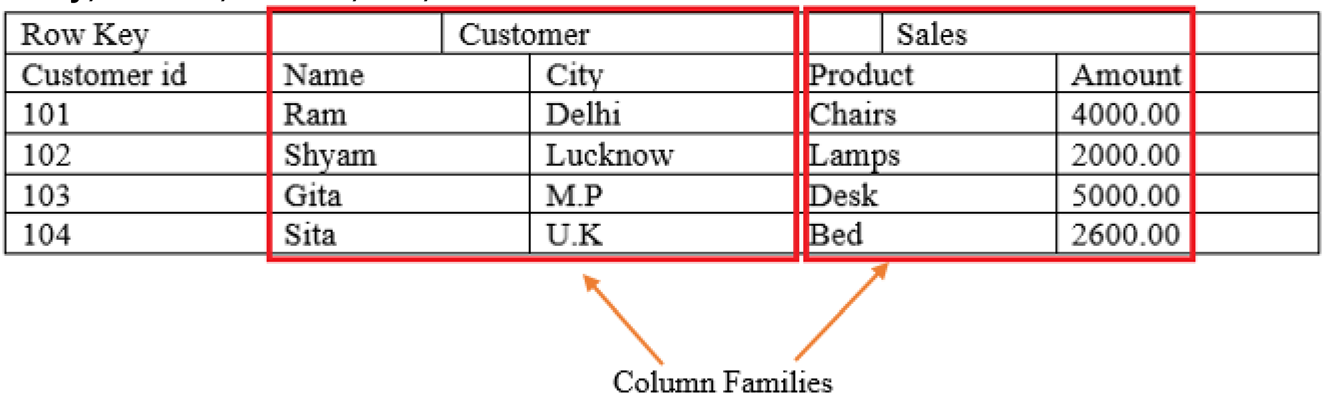
deleteMany()

deleteMany() is a method used to delete multiple documents from a desired collection with a single delete operation. A list is passed into the method and the individual items are defined with filter criteria as in *deleteOne()*.

```
db.RecordsDB.deleteMany({species:"Dog"})
{ "acknowledged" : true, "deletedCount" : 2 }
> db.RecordsDB.find()
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years",
"species"
```

HBase Data Model

The HBase Data Model is designed to handle semi-structured data that may differ in field size, which is a form of data and columns. The data model's layout partitions the data into simpler components and spread them across the cluster. HBase's Data Model consists of various logical components, such as a **table, line, column, family, column, column, cell,** and **edition.**



Row Key	Customer		Sales		
Customer id	Name	City	Product	Amount	
101	Ram	Delhi	Chairs	4000.00	
102	Shyam	Lucknow	Lamps	2000.00	
103	Gita	M.P	Desk	5000.00	
104	Sita	U.K	Bed	2600.00	

Column Families

Table:
An HBase table is made up of several columns. The tables in HBase defines upfront during the time of the schema specification.

Row:
An HBase row consists of a row key and one or more associated value columns. Row keys are the bytes that are not interpreted. Rows are ordered lexicographically, with the first row appearing in a table in the lowest order. The layout of the row key is very critical for this purpose.

Column:
A column in HBase consists of a family of columns and a qualifier of columns, which is identified by a character: (colon).

Column Family:
Apache HBase columns are separated into the families of columns. The column families physically position a group of columns and their values to increase its performance. Every row in a table has a similar family of columns, but there may not be anything in a given family of columns.
The same prefix is granted to all column members of a column family. For example, Column **courses: history** and **courses: math**, are both members of the column family of courses. The character of the colon (:) distinguishes the family of columns from the qualifier of the family of columns. The prefix of the column family must be made up of printable characters.
During schema definition time, column families must be declared upfront while columns are not specified during schema time. They can be conjured on the fly when the table is up and running. Physically, all members of the column family are stored on the file system together.

Column Qualifier
The column qualifier is added to a column family. A column standard could be **content** (html and pdf), which provides the content of a column unit. Although column families are set up at table formation, column qualifiers are mutable and can vary significantly from row to row.

Cell:

A Cell store data and is quite a unique combination of row key, Column Family, and the Column. The data stored in a cell call its value and data types, which is every time treated as a byte[].

Timestamp:

In addition to each value, the timestamp is written and is the identifier for a given version of a number. The timestamp reflects the time when the data is written on the Region Server. But when we put data into the cell, we can assign a different timestamp value.

HBase CRUD Operations

General Commands

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

status: This command will display the cluster information and health of the cluster.

```
hbase(main):>status
hbase(main):>status "detailed"
```

version: This will provide information about the version of HBase.

```
hbase(main):> version
```

whoami : This will list the current user.

```
hbase(main):> whoami
```

table_help : This will give the reference shell command for HBase.

```
hbase(main):009:> table_help
```

Create

Let’s create an HBase table and insert data into the table. Now that we know, while creating a table user needs to create required Column Families.

Here we have created two-column families for table ‘employee’. First Column Family is ‘Personal Info’ and Second Column Family is ‘Professional Info’.

```
create 'employee', 'Personal info', 'Professional Info'
0 row(s) in 1.4750 seconds

=> Hbase::Table - employee
```

Upon successful creation of the table, the shell will return 0 rows.

Create a table with Namespace:

A namespace is nothing but a logical grouping of tables.’company_empinfo’ is the namespace id in the below command.

```
create 'company_empinfo:employee', 'Personal info', 'Professional Info'
```

Create a table with version:

By default, versioning is not enabled in HBase. So users need to specify while creating.

Given below is the syntax for creating an HBase table with versioning enabled.

```
create 'tableName',{NAME=>"CF1",VERSIONS=>5},{NAME=."CF2",VERSIONS=>5}
create 'bankdetails',{NAME=>"address",VERSIONS=>5}
```

Put:

Put command is used to insert records into HBase.

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

Here in the above example all the rows having Row Key as 1 is considered to be one row in HBase.To add multiple rows

```
put 'employee', 2, 'Personal info:empId', 20
```

```
2       put 'employee', 2, 'Personal info:Name', 'Bob'
3       put 'employee', 2, 'Professional Info:Dept', 'Sales'
```

As discussed earlier, the user can add any number of columns as part of the row.

Read

'get' and 'scan' command is used to read data from HBase. Lets first discuss 'get' operation.
get: 'get' operation returns a single row from the HBase table. Given below is the syntax for the 'get' method.

```
1       get 'table Name', 'Row Key'
1       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
```

To retrieve a specific column of row:
Follow the command to read a specific column of a row.

```
1       get 'table Name', 'Row Key',{COLUMN => 'column family:column'}
2       get 'table Name', 'Row Key' {COLUMN => ['c1', 'c2', 'c3']
1       get 'employee', 1 ,{COLUMN => 'Personal info:empId'}
```

```
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
```

Note: Notice that there is a timestamp attached to each cell. These timestamps will update for the cell whenever the cell value is updated. All the old values will be there but timestamp having the latest value will be displayed as output.
Get all version of a column
Below given command is used to find different versions. Here 'VERSIONS => 3' defines number of version to be retrieved.

```
1       get 'Table Name', 'Row Key', {COLUMN => 'Column Family', VERSIONS => 3}
```

scan:
'scan' command is used to retrieve multiple rows.
Select all:
The below command is an example of a basic search on the entire table.

```
1       scan 'Table Name'
1       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
```

Note: All the Rows are arranged by Row Keys along with columns in each row.

Column Selection:
The below command is used to Scan any particular column.

```
1       hbase(main):001:>scan 'employee' ,{COLUMNS => 'Personal info:Name' }
```

```
ROW                              COLUMN+CELL
 1                  column=Personal info:Name, timestamp=1504600767520, value=Alex
 2                  column=Personal info:Name, timestamp=1504600767588, value=Bob
2 row(s) in 0.3660 seconds
```

Limit Query:
The below command is used to Scan any particular column.

```
1       hbase(main):002:>scan 'employee' ,{COLUMNS => 'Personal info:Name',LIMIT =>1 }
```

```
ROW                              COLUMN+CELL
 1                   column=Personal info:Name, timestamp=1504600767520, value=Alex
1 row(s) in 0.0270 seconds
```

<u>Update</u>
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.

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

The old value will not be deleted from the HBase table. Only the updated record with the latest timestamp will be shown as query output.
To check the old value of any row use below command.

```
1       get 'Table Name', 'Row Key', {COLUMN => 'Column Family', VERSIONS => 3}
```

<u>Delete</u>
'**delete**' command is used to delete individual cells of a record.
The below command is the syntax of delete command in the HBase Shell.

```
1       delete 'Table Name' ,'Row Key','Column Family:Column'
1       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.

```
1       disable 'employee'
```

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

```
1       drop 'employee'
```

You can verify the table in using 'exist' command and enable table which is already disabled, just use 'enable' command.