**Chapter 4: Intermediate SQL**

Table of Contents

[Joined Relations 2](#_Toc50152366)

[Views 7](#_Toc50152367)

[Transactions 9](#_Toc50152368)

[Integrity Constraints 9](#_Toc50152369)

[Cascading Delete 10](#_Toc50152370)

[Date and Time in SQL 12](#_Toc50152371)

[Large Object Types 14](#_Toc50152372)

[Creating Table Structures 14](#_Toc50152373)

[Authorization 15](#_Toc50152374)

## Joined Relations

We can join relations in several ways. The two basic types of joins are Inner Joins and Outer Joins. Inner Joins are just natural joins. Outer Joins can be divided into Left Outer Joins, Right Outer Joins and Full Outer Joins.

We will be using these two tables as examples for this part:

table1

|  |  |
| --- | --- |
| ID | Name |
| 001 | "A" |
| 002 | "B" |

table2

|  |  |
| --- | --- |
| ID | CGPA |
| 001 | 3.00 |
| 003 | 4.00 |

An Inner Joins returns the tuples for which some attribute is equal on both tables. They allow us to split large tables into smaller parts, and combine them as needed. This helps remove redundancy and inconsistency. Typically, we use a linking attribute called a foreign key. Although the join itself works with the data and does not require the foreign key at all, it is still best to use a foreign key to ensure that we get the data from the right source and the data is automatically updated when any changes take place.

SELECT \* FROM table1 INNER JOIN table2 ON table1.id = table2.id;

SQL

This will give us a resulting table where tuples contain all of the attributes of table1 and table2, but only the tuples for which the attribute id is the same in both tables are present.

|  |  |  |
| --- | --- | --- |
| ID | Name | CGPA |
| 001 | "A" | 3.00 |

A Left Outer Join gives us all the tuples from the left-hand side table, i.e table1. If the attribute being compared is present in the right-hand side table, table2, then the data from table2 is also shown. If the attribute is not equal, NULL shown in the place of the attributes from table2.

SELECT \* FROM table1 LEFT OUTER JOIN table2 ON table1.id = table2.id;

SQL

|  |  |  |
| --- | --- | --- |
| ID | Name | CGPA |
| 001 | "A" | 3.00 |
| 002 | "B" | NULL |

A Right Outer Join does the opposite, taking every tuple from the right-hand side table and only the data that matches from the left-hand side table.

SELECT \* FROM table1 RIGHT OUTER JOIN table2 ON table1.id = table2.id;

SQL

|  |  |  |
| --- | --- | --- |
| ID | Name | CGPA |
| 001 | "A" | 3.00 |
| 003 | NULL | 4.00 |

A Full Outer Join takes every tuple from both tables. The tuples on the left table for which there is no match in the right table show NULL in place of the attributes from the right table and vice versa.

SELECT \* FROM table1 FULL OUTER JOIN table2 ON table1.id = table2.id;

SQL

|  |  |  |
| --- | --- | --- |
| ID | Name | CGPA |
| 001 | "A" | 3.00 |
| 002 | "B" | NULL |
| 003 | NULL | 4.00 |

A Full Outer Join should not be confused with a Cartesian Product. A Cartesian Product joins every tuple on the left-hand side table with every tuple on the right-hand side table. This gives us lots of invalid data. A Full Outer Join joins every tuple on the left-hand side table with the tuples that match the given condition on the right-hand side table, while showing NULL values in the places for which there are no matches. We do not get any invalid data with a Full Outer Join.

Now let us look at another example, with the old syntax for joined relations. Consider the following tables and the data inserted into them:

*/\* creating passports table \*/*

CREATE TABLE passports

(

pid NUMBER PRIMARY KEY,  
 name VARCHAR2(10),  
 issued\_on DATE,  
 expired\_on DATE  
);  
  
*/\* data for passports table \*/*INSERT INTO passports VALUES(301, 'a', sysdate-100, sysdate+360);  
INSERT INTO passports VALUES(302, 'b', sysdate-10, sysdate+265);  
INSERT INTO passports VALUES(303, 'c', sysdate-35, sysdate+163);  
  
COMMIT;

*/\* creating driving licenses table \*/*CREATE TABLE driving\_lic  
(  
 did NUMBER PRIMARY KEY,  
 name VARCHAR2(10),  
 issued\_on DATE,  
 expired\_on DATE,  
 vehicle VARCHAR2(10),  
 passport NUMBER,CONSTRAINT fk\_driv FOREIGN KEY (passport) REFERENCES passports  
);

*/\* data for driving licenses table \*/*INSERT INTO driving\_lic VALUES (801, 'a', SYSDATE-23, SYSDATE+230, 'Heavy', 301);  
INSERT INTO driving\_lic VALUES (802, 'c', SYSDATE-24, SYSDATE+30, 'Light', 303);  
*/\* following entries have no passports \*/*INSERT INTO driving\_lic VALUES (803, 'n1', SYSDATE-24, SYSDATE+30, 'Light', NULL);  
INSERT INTO driving\_lic VALUES (804, 'n2', SYSDATE-224, SYSDATE+432, 'Light', NULL);  
INSERT INTO driving\_lic VALUES (805, 'n3', SYSDATE-56, SYSDATE+238, 'Light', NULL);  
  
commit;

SQL

Note that a new data type, DATE, is being used. This is a special data type that allows us a lot of functionality related to date and time. For example, SYSDATE gives us the current system date.

Also note that the foreign key in the table, driving\_lic, has been given a different name than the attribute it is referencing. The name could have been the same as well.

Now let use look at the different types of joins. First, a typical inner join:

*/\* natural join -> old syntax \*/*select p.pid, p.name, p.issued\_on, d.did, d.issued\_on, d.vehicle  
from passports p, driving\_lic d  
where p.pid = d.passport;  
*/\* shows 2 records \*/*

SQL

Next, a left-outer join:

*/\* left outer -> old syntax \*/*select p.pid, p.name, p.issued\_on, d.did, d.issued\_on, d.vehicle  
from passports p, driving\_lic d  
where p.pid = d.passport(+);  
*/\* shows 3 records \*/*

SQL

A right-outer join:

*/\* right outer -> old syntax \*/*select p.pid, p.name, p.issued\_on, d.did, d.issued\_on, d.vehicle  
from passports p, driving\_lic d  
where p.pid(+) = d.passport;  
*/\* shows 5 records \*/*

SQL

Finally, a full outer join. The full outer join did not exist earlier, and thus does not have an old syntax.

*\* full outer -> new syntax (no old syntax; did not exist) \*/*select p.pid, p.name, p.issued\_on, d.did, d.issued\_on, d.vehicle  
from passports p full outer join driving\_lic d  
on p.pid = d.passport;  
*/\* shows 6 records \*/*

SQL

## Views

When we create a table and add data to it, the data needs to be stored somewhere in memory. We are not directly interacting with the memory, but rather interacting with the data through the table. The data that is stored is said to be in the physical view.

When we perform some query on the table, such as SELECT \* FROM table1, the results we obtain contains the data that is stored in the physical view. The data in the results itself is not being stored anywhere, and is only being derived. This is called the logical view. We can save the logical view that we obtain when we perform a query, somewhat like a variable:

CREATE OR REPLACE VIEW myview AS SELECT \* FROM table1 WHERE table1.id > 100;

SQL

This gives us several advantages. We can reuse the code multiple times simply by calling myview without having to write the entire code again. Any changes in the original data is also automatically reflected without any update being required, since the data is not actually saved in the view, only the query, which means the data is extracted again when the view is called. Lastly, different people have different levels of access to data and should therefore get different views. This is possible using this method.

A view mainly serves two purposes, to control access so that different users can be given different levels of access and control to data, and to help reuse code so that complicated code does not need to be repeatedly re-written.

When views were originally created, they were used as a method of viewing data only. It was considered laughable to try to update data through views, since that was not their purpose. However, in recent times, updating the database through a view is also allowed. Most SQL implementations only allow this to a limited extent, when a few conditions are satisfied. Note that these conditions apply when updating data using the view, not if we just want to create a view (without updating our data through it).

* Only DML statements can be executed, i.e. INSERT, DELETE and UPDATE.
* The view should only deal with a single underlying database relation
* In the DML statements, the SELECT clause cannot contain expressions, aggregates or distinct specifications, only attribute names.
* All attributes in the underlying table that have been set to NOT NULL *must* be included in the view.

We can store a view and when we use it, we would also want it to be kept up-to-date, i.e. reflecting any changes to the underlying table. A view that is stored and kept up-to-date is called a materialized view. The process of keeping the views up-to-date is handled through a repository on a server, using a process called data warehousing. Materialized views help operations stay efficient and smooth and give quick access time.

How and when we update views is also important. Consider a system where we have a huge number of databases spread throughout the world. The materialized views depend on all of this data, and we need to keep it up-to-date. One option would be to update the data in real-time. However, this would be a bad idea. All the applications we use work off of that database, so if it is constantly busy with sending updates to our views, it will slow down all the applications that our users need. Instead, batch processing is done at a suitable time, perhaps at midnight when servers are likely to be least busy with customers. The materialized view can then be used by business analysts to make business decisions.

## Transactions

A transaction is a unit of work that has an all or nothing property, i.e. it is done in its entirety from beginning to end or not done at all. For example, say we have five tasks occurring one after another. If all the tasks complete successfully, only then will we consider the transaction to be successful. If even one of the tasks fails, the transaction will be considered unsuccessful and whatever work the other tasks performed will be rolled back.

Transactions are handled using two commands in databases, COMMIT and ROLLBACK. The COMMIT command saves the changes made by a successful transaction. The ROLLBACK command removes the changes made by an unsuccessful transaction, up to the last COMMIT command.

## Integrity Constraints

Integrity constraints are a set of restrictions and rules that guard against accidental damage to the database by ensuring changes to the database do not result in a loss of data consistency. For example, the constraints we set for primary keys and NOT NULL values are integrity constraints that ensure data remains consistent.

Integrity constraints are introduced while designing the database schema and are specified within the DDL commands. It is possible to impose such restrictions from the UI design as well, but it is generally recommended to impose them on the back-end, i.e. the database itself, since that ensures better functionality. This is because UIs may be designed multiple times in different places. If the constraints are forgotten in even one of those places, our database will no longer be consistent.

There are four integrity constraints:

* NOT NULL
* PRIMARY KEY
* UNIQUE
* CHECK(p)

The only constraint that should be foreign to us at this point is CHECK. This constraint makes it very easy for us to check the data being given as input. For example, the current semester for a student record should always be either ‘Summer’ or ‘Winter’. The CHECK constraint can be used to make sure no other data is entered.

All four of the integrity constraints above work with individual relations. There are also integrity constrains that work with multiple relations. These are called referential integrity constraints. The only referential integrity constraint we have worked with is the FOREIGN KEY constraint.

There is another type of referential integrity constraint called cascading actions. Cascading actions are theoretically supported for deletion and updating, but in reality, most SQL databases only support cascading delete. We will be looking at that next.

### Cascading Delete

Say we have a departments relation, a students relation and a grades relation. Records from the grades relation are related to records from the students relation, and records form the students relation are related to records from the departments relation. Under cascading delete, if we try to delete a record from the departments relation, it will find all records in the students relation that are related to that record, and all records from the grades relation that are related to any relevant records from the students relation and delete all of those as well. The deletion occurs backwards, first deleting the required records from the grades relation, then from the students relation and finally the record we tried to delete from the departments relation. Cascading delete may not always be a good thing, but in many cases it is.

/\* creating departments relation \*/  
CREATE TABLE depts  
(  
 id NUMBER PRIMARY KEY,  
 name VARCHAR2(10)  
);  
  
INSERT INTO depts VALUES (1, 'MCE');  
INSERT INTO depts VALUES (2, 'EEE');  
INSERT INTO depts VALUES (3, 'TVE');  
INSERT INTO depts VALUES (4, 'CSE');  
INSERT INTO depts VALUES (5, 'CEE');  
  
COMMIT;

/\* creating students relation \*/  
CREATE TABLE students  
(  
 sid NUMBER PRIMARY KEY,  
 name VARCHAR2(20),  
 cgpa NUMBER(7, 6),  
 dept NUMBER,  
 CONSTRAINT fk\_stu FOREIGN KEY (dept) REFERENCES depts ON DELETE CASCADE  
 /\* notice the cascading delete \*/  
);  
  
INSERT INTO students VALUES (401, 'a1', 3.6, 4);  
INSERT INTO students VALUES (402, 'a2', 3.3, 4);  
INSERT INTO students VALUES (403, 'a3', 3.9, 4);  
  
INSERT INTO students VALUES (501, 'b1', 3.2, 5);  
INSERT INTO students VALUES (502, 'b2', 3.1, 5);  
INSERT INTO students VALUES (503, 'b3', 3.7, 5);  
  
CREATE TABLE grades  
(  
 cid VARCHAR2(10),  
 sid NUMBER,  
 grade VARCHAR2(4),  
 CONSTRAINT fk\_grades FOREIGN KEY (sid) REFERENCES students  
 ON DELETE CASCADE  
);  
  
INSERT INTO grades VALUES ('CSE 4101', 401, 'A+');  
INSERT INTO grades VALUES ('CSE 4101', 402, 'C');  
INSERT INTO grades VALUES ('CSE 4101', 501, 'A+');  
  
COMMIT;  
  
DELETE depts WHERE id=5;  
/\* all grades of CEE students delete;  
 all students of CEE deleted;  
 CEE deleted  
 \*/

SQL

## Date and Time in SQL

DATE and TIME are two data types that give us a larger range of functionality for dates and times respectively. Both are precise to a second. The default format for DATE is DD-MON-YY. The default format can be changed using the TO\_CHAR() function to a large number of different formats.

SELECT TO\_CHAR(SYSDATE, 'YYYY-MM-DD') FROM dual;

SQL

Notice the keyword dual. This is not really a table, but more like a placeholder to comply with SQL’s syntax. It can also be used to perform arithmetic.

SYSDATE gives us the current system date.

On the flip side, the TO\_DATE function can be used to convert a string to a DATE type in a specific format.

SELECT TO\_DATE('5 Jan 2020', 'DD MON YYYY') FROM dual;

SQL

This gives us the ability to use it with other date related functions.

Other date related functions include:

SELECT ADD\_MONTHS(SYSDATE, 13) FROM dual;  
*/\* prints the date 13 months from the system date \*/*

SELECT NEXT\_DAY(SYSDATE, 'FRIDAY') FROM dual;  
*/\* prints the date of the next Friday from the system date \*/*

SELECT MONTHS\_BETWEEN('18-FEB-92', '14-MAY-89') FROM dual;  
*/\* prints the number of months (even fractions) between two dates \*/*

SQL

## Large Object Types

Photos, videos and other large files are stored in databases as large objects. These are stored in two ways:

* Blob – Binary large object. The object is stored as a large collection of binary data that the database system cannot work with. It will require an external application to work with.
* Clob – Character large object. The object is stored as a large collection of character data.

When a query returns a large object, a pointer to the object is returned instead of the object itself. Special programming is needed to manipulate it. This is because command line SQL cannot show us the large objects.

## Creating Table Structures

Very often, we will find ourselves working with similar table structures. It would be useful to borrow the structure for such tables from existing ones. While copying, we can copy the data as well, which is what happens by default. To exclude the data, we will need to use a trick.

CREATE TABLE student2 AS (SELECT sid, name, cgpa FROM students WHERE 1=2);

SQL

Here, we copied the table students onto the new table students2. Due to the invalid condition we gave in the WHERE clause, none of the actual data is copied, only the structure.

## Authorization

Authorization is also called role-based access control. It is used to distribute large systems into a number of users with different access control. A role is like a package of access controls which is customized and distributed to different users. Then the owner can grant or revoke the created role to specific users. Roles can also be defined using other roles.

GRANT <privilege list> ON <relation or view name> TO <user or role list>;

SQL

The privilege list includes the commands SELECT, UPDATE, INSERT, DELETE and EXECUTE, the last of which we have not yet covered.

For example, consider a grading system. The owner is the result processing system (RPS), and roles could be of three types, one for viewing marks only, one for editing marks, and one for updating student information. Student accounts would be given the first role, teacher accounts the second role and administrative staff accounts the third role.

*/\* creating new user \*/*CREATE USER rps IDENTIFIED BY test123;  
GRANT DBA TO rps;  
  
*/\* switching to rps \*/*disc  
conn rps/test123  
  
*/\* lets pretend we made a bunch of tables and added data here \*/*

*/\* creating views \*/*CREATE OR REPLACE VIEW emp\_v  
AS  
SELECT name, designation, roomNo, counselHour  
FROM employees readonly;  
*/\* notice read only \*/  
  
/\* creating roles \*/*CREATE ROLE role\_readonly;  
CREATE ROLE role\_marks\_entry;  
  
*/\* granting SELECT permission on these tables and views \*/*GRANT SELECT ON course\_info TO role\_readonly;  
GRANT SELECT ON depts TO role\_readonly;  
GRANT SELECT ON emp\_v TO role\_readonly;  
GRANT SELECT ON grades TO role\_readonly;  
GRANT SELECT ON students TO role\_readonly;  
  
*/\* granting INSERT and UPDATE permissions on this table \*/*GRANT INSERT, UPDATE ON grades TO role\_marks\_entry;  
  
*/\* granting one role to another role \*/*GRANT role\_readonly TO role\_marks\_entry;  
  
*/\* creating users and giving them roles \*/*CREATE USER hod IDENTIFIED BY test123;  
*/\* this account has been created for the head of department \*/*GRANT role\_readonly, role\_marks\_entry to hod;  
  
*/\* connect to hod account \*/*disc  
conn hod/test123  
  
*/\* this account can read all those tables and views and edit marks but nothing else \*/*

SQL