Chpater 4: Intermediate SQL

Contents

1 Join Expressions

Joins are used to combine information from two or more relations. They can be of various types.

1.1 The Natural Join

1) The **natural join** operation operates on two relations and produces a subset of the Cartesian product of those relations which contains tuples whose values of attributes that appear in the schema of both relationships is the *same*.
2) SQL syntax:

```
table1 natural join table2;
```

3) Note that while joining more than two relations, the order of joining matters. SQL provides a **join ... using** construct to specify which columns should be updated. Syntax:

```
table1 join table2 using (attr_1, attr_2,
    ..., attr_n);
```

1.2 Join Conditions

1) The **join ... using** clause requires values to match on specified attributes.
2) To join relations based on general conditions, the **join ... on** clause is used. Syntax:

```
table1 join table2 on pred;
```

3) An equivalent expression if the **on** clause is not used is to move the predicate to the **where** clause.
4) However, the reasons for using an **on** condition are:
   a) It behaves differently as compared to **where** clauses on outer joins.
   b) The join condition is specified separately. This makes the query readable by humans.

1.3 Outer Joins

1) Works similar to other joins, but preserves lost tuples by creating tuples in the result containing null values.
2) They are of three forms:
   a) **left outer join:** Preserves tuples only in the relation named before the operation.
   b) **right outer join:** Preserves tuples only in the relation named after the operation.
   c) **full outer join:** Preserves tuples in both relations.
3) Syntax similar to the inner join syntax.
4) To compute outer join:
   a) Compute inner join.
   b) For every tuple t in the relation (preceding/-following) the operation that does not match any tuple in the other relation, add a tuple r consisting of values of t with the remaining values padded by **null**.
   c) A full outer join is the union of left and right outer join.
   d) **on** clause can be used instead of **natural** as before.
   e) **on** and **where** behave differently for outer joins. The **on** clause is part of the outer join specification, while the **where** clause is applied after performing the join.

1.4 Join Types and Conditions

Normal joins are called **inner joins** in SQL. An inner join is the default behaviour when **join** is specified. Similarly, **natural join** is equivalent to **natural inner join**.

2 Views

1) Not always desirable for all users to see the entire set of relations in a database.
2) May require a personalized collection of relations to match a certain user's intention of what the database looks like.
3) A **view** is a *virtual relation* defined by a query and the relation contains the result of the query. It is not stored but computed whenever it is used from the existing relations.

2.1 View Definition

1) Use the **create view** command. Syntax:

```
create view tableName(viewSchema) as
    viewName;
```

2) Difference from **with** clause is that views remain available until explicity dropped, while the relations defined in the **with** clause are local to that query only.

2.2 Using Views in SQL Queries

Can be used wherever an actual relation appears in a query statement. One can define views using other existing views also.

2.3 Materialized Views

1) Views that are stored but updated when the underlying relations are updated are called **materialized views**.
2) The process of keeping a materialized view up to date is called **materialized view maintenance**. It can be immediate, lazy, or periodic.
3) Materialized views may be favourable if the view is frequently used, since it leads to faster response times. However, there is a tradeoff in storage and added overhead for updates.

2.4 Update of a View

1) Difficulties of updating a view:
   a) Modifications to the view must be translated to modifications of the database.
   b) Other attributes not included the view must be filled with nulls.
2) An SQL view is said to be **updatable** if:
   a) The **from** clause has only one relation.
   b) The **select** clause contiains only the attribute names of the relation, and not any aggregated, expressions, or **distinct** specification.
   c) Any attribute *not* listed in the **select** clause should be set to **null**.
   d) The query defining the view does *not* have a **group by** or **having** clause.
3) Views can be defined with the **with check option** clause, meaning any insertion or update not satisfying the **where** clause condition of the view is rejected by the database.
4) A preferred approach is to use trigger mechanism. The **instead of** feature replaces default update, insert and delete mechanisms with actions designed specifically for each case.

3 Transactions

1) A transaction consists of a sequence of query and/or update statements.
2) A transaction begins implicitly when an SQL statement is executed.
3) Must be ended by one of:
   a)
   b) **commit (work):** Commits the current transaction; makes the updates permanent in the database. A new transaction is started after commit.
   c) **rollback (work):** Rolls back the current transaction; undoes all updates performed by the transaction. Database restored to the state it was in prior to the execution of the first statement of the transaction. Cannot undo effects of already committed transactions.
4) Database provides abstraction of transaction as being **atomic**.
5) Many SQL implementations consider each update statement as a single transaction and commit after each action. This *auto-commit* feature can be disabled by typing **set autocommit off**.
6) A better alternative to this is to enclose statements that make up a transaction in **begin atomic ... end** as in SQL:1999. Some systems, however, use **commit/rollback (work)** instead of **end** as above.

4 Integrity Constraints

1) Ensure that changes made to the database by authorized users do not result in a loss of data consistency. Note that constraints on authorization are called *security constraints*.
2) They are specified in the DDL, either when creating a table, or when altering a table.

4.1 Not Null Constraint

1) Prohibits insertion of null value for the attribute. Example of **domain constraint**.
2) SQL prohibits insertion of null values in primary keys of a relation.

4.2 Unique Constraint

1) Syntax:

   unique (attr_1, attr_2, ..., attr_n);

2) Specifies that the attributes in the brackets form a *superkey*.
3) Note that attributes declared to be unique *can* be **null**.

4.3 The Check Clause

1) Syntax: **check**(P), where P is a predicate to be satisfied by every tuple in the relation.
2) A check is *satisfied* if it is NOT **false**, hence tuples returning **unknown** are not violations. A **not null** constraint can eliminate such nulls.
3) A check can appear in the schema as part of the declaration of an attribute, or at the end of the declaration of the schema.
4) According to the SQL standard, P can contain a predicate, though it is not widely implemented in popular database systems.

4.4 Referential Integrity

1) Foreign keys can be specified using the **foreign key** clause.
2) Syntax:

   foreign key forKey references tableName (schema);

   Here, `forKey` is compatible with the primary key of the referenced relation.
3) Can specify what happens if the referenced relation is changed, by specifying **on (delete — update) (cascade — set null — set** *v***)** in the declaration of the foreign key constraint. This means that the actions can be cascaded to the referencing relation or values in the referenced relation can be updated.
4) Default behaviour is to reject action on the referenced relation that causes the violation.

4.5 Assigning Names to Constraints

1) Syntax:

   constraint constrName **check** pred;

2) Helps in dropping constraints. Syntax:

   alter table tableName **drop constraint** constrName;

3) Not all database systems support system assigned names for constraints.

4.6 Integrity Constraint Violation During a Transaction

1) Integrity constraints can be violated in a transaction temporarily after one step, but restored in a later step of the transaction.
2) To allow for such flexibility, SQL provides a clause **initially deferred** that can be added to a constraint specification.
3) Alternatively, a constraint can be specified as **defferable**, meaning that it is checked immediately by default but can be deferred.
4) For deferrable constraints, we can specify a list of constraints to be treated as deferred as part of a transaction. Syntax:

   set constraints constrList **deferrable**;

5) We can workaround the constraints by assigning temporary **null** values, but that requires a lot of programming effort.

4.7 Complex Check Conditions and Assertions

1) The predicate in the **check** clause can be any predicate, and can also contain subqueries.
2) However, complex check conditions can be costly to test.
3) An **assertion** is a predicate expressing a condition that we wish the database always to satisfy.
4) Syntax:

   create assertion assertName **check** pred;

5 SQL Data Types and Schemas

5.1 Date and Time Types in SQL

1)
2) **date:** A calendar date containing a (four-digit) year, month and day.
3) **time:** The time of day in hours, minutes and seconds. An additional data type **time(*p*)** is used to specify the number of fractional digits (default = 0) for seconds. One can specify timezone information using **time with timezone**.
4) **timestamp:** A combination of **date** and **time**. A variant, **timestamp(*p*)** can be used to specify the number of fractional digits (default = 0) for seconds.
5) To extract individual fields from each data type, we use **extract** (*f* **from** *d*);
6) SQL defines several functions to get:
   a) Current date (**current_date**),
   b) Current time (**current_time**),
   c) Current timestamp with timezone (**current_timestamp**),
   d) Local timestamp without timezone (**local_timestamp**).
7) For comparisons of date and time types, an additional data type called **interval** is defined by SQL.
8) An **interval** is obtained on subtraction of two variables of type **date** or **time**. Conversely, adding or subtracting an interval from a **date** or **time** gives back a **date** or **time** respectively.

5.2 Type Conversion and Formatting Functions

1) For *explicit* type conversion, the syntax is:

   cast (val **as** newType);

2) The **coalesce** function can be used to change how null values are displayed. Syntax:

   coalesce (val_1, val_2, ..., val_n);

   It returns the first non-null value from the argument list.
3) Some database systems allow values of any data type to be shown instead of **null**, such as the **decode** function in Oracle.

5.3 Default Values

1) Syntax (in DDL):

   ...,
   attr **domain default** val,
   ...,

2) If an attribute has default value, it may not be declared when inserting a tuple into the relation.

5.4 Large Object Types

1) SQL has two data types to store large data items:
   a) **blob**(*s*): This is used to store Binary Large OBjects such as executables, movies, images, etc. upto size *s*.
   b) **clob**(*s*): This is used to store Character Large OBjects such as text files or documents upto size *s*.

2) Storing large objects in memory is inefficient and impractical. Hence, a *locator* is stored in the database and retrieved when required to access the data object.

5.5 User-Defined Types

1) There are two forms of user-defined types: **structured** and **distinct**.
2) Distinct data types are created using the following syntax:

```
create type typeName as actualType
    final;
```

Note that **final** is required by the SQL:1999 standard.

6 Index Definitions in SQL

1) An **index** is a data structure on an attribute that allows the database system to find tuples with a particular value of the attribute efficiently.
2) Redundant, form part of the physical and not logical schema.
3) Important for efficient processing of transactions, enforcement of constraints.
4) Not easy for the database system to determine what indices to maintain. Control over index creation and deletion left to programmer.
5) Syntax for index creation:

```
create index indexName on
tableName(attrList);
```

The listed attributes form the search key of the index.
6) To declare that the search key is a candidate key, add the attribute **unique** to the index definition.
7) To drop an index:

```
drop index indexName;
```

7 Authorization

1) Includes authorization to
   a) read data
   b) insert new data
   c) update data
   d) delete data

   Each authorization is called a **privilege**.
2) SQL implementation checks whether the user has access to perform the operation before performing the operation.
3) Ultimate authority resides with database administrators.

7.1 Granting and Revoking of Privileges

1) SQL standard includes **select, insert, update** and **delete** privileges.
2) To confer authorization:

```
grant privList
on tableName/viewName
to userList/roleList;
```

3) For the **update** privilege, one can optionally specify the authorized list of attributes in parentheses after **update**.
4) The usernaem **public** refers to current and future users of the database system.
5) To revoke authorizations:

```
revoke privList
on tableName/viewName
from userList/roleList;
```

7.2 Roles

1) Presents a better approach to managing authorizations for various users.
2) Can grant/revoke authorizations from roles instead of users.
3) To create roles

```
create role roleName;
```

4) To grant a role to a user

```
grant roleName to userName;
```

7.3 Authorization on Views

1) When a view is used, the query processor replaces the uses of the view with its definition, thereby producing a query on the underlying relation itself.
2) The database system checks authorization on the query of the user.
3) A user who creates a view receives no additional authorization beyond those they already have. **No additional privileges are granted.**

4) View creation request is denied if no privileges can be granted on it.
5) The **execute** privilege is granted on a function or a procedure. The function or procedure runs as if it was exected by the user who created it.
6) This behaviour is not always required. From SQL:2003 onwards, if the function definition has an extra clause **sql security invoker**, then it is executed under the privileges of the user who calls the function rather than the user who *defined* it.

7.4 Authorizations on Schema

1) *Only* the schema owner can modify the schema, such as create/delete relations, add/drop attributes or indices.
2) The **references** privilege permits a user to declare foreign keys when creating relations. The schema can be optionally specified in parentheses after the privilege.
3) Creating foreign keys with privileges can prevent activity in the referenced relation as the referencing relations may have to be updated.
4) Restricts creation of **check** constraints in referencing relations due to the foreign key privileges.

7.5 Transfer of Privileges

1) To authorize a user to further grant a privilege to another user/role the clause **with grant option** is appended to the **grant** command.
2) Leads to a notion of an **authorization graph**, which is a directed graph.
3) The nodes of this graph are the users. If there is an edge $U_i \rightarrow U_j$ in this graph, then user U_i grants the privilege to U_j.
4) A user has an authorization *iff* there is a path from the root of the authorization graph (representing the DBA) to the node representing the user.

7.6 Revoking of Privileges

1) Revocation of a privilege from a user/role may cause *cascading* revocation. Default behaviour in database systems.
2) To prevent this, append the revoke statement with the **revoke** clause. Error is raised and revocations are not performed if there are cascading revocations.
3) To revoke the grant option:

```
revoke grant option for privList on
tableName from userName;
```

4) Cascading revocation is not always appropriate. To prevent ill effects of the same, SQL permits granting of privileges by roles to other roles.
5) SQL has a notion of current role in a session. Usually set to null by default. Change using

```
set role roleName;
```

The statement executes if the role has been granted to the user.
6) To grant a privilege from the current role, append the **grant** statement with **granted by current_role** clause.

7.7 Row-Level Authorization

1) Fine-grained authorization. Prevents users from seeing irrelevant data.
2) Oracle's **Virtual Private Database** (VPD) supports this mechanism.
3) Database system adds predicates to **where** clauses to enforce the authorization.
4) A pitfall of this mechanism is that it can alter the meanings of issued queries.