SQL:
Data Definition Language

CSC343 Introduction to Databases

UNIVERSITY OF TORONTO

Tables

- When creating a table, we must define the type of each attribute.

- This is analogous to declaring a variable's type in a program, e.g., "int num;" in Java or C.

- Pros and cons?

UNIVERSITY OF TORONTO

Built-in data types - 1

- `CHAR(n)`: fixed-length string of n characters. Padded with blanks if necessary.
- `VARCHAR(n)`: variable-length string of up to n characters.
- `TEXT`: variable-length, unlimited. Not in the SQL standard, but psql and others support it.
- `INT = INTEGER`
- `FLOAT = REAL`
- `BOOLEAN`
- `DATE`; `TIME`; `TIMESTAMP` (date plus time)

Built-in data types -2

- Strings: `‘Shakespeare’’s Sonnets’`
  → Must surround strings with single quotes.
- `INT`: `37`
- `FLOAT`: `1.49, 37.96e2`
- `BOOLEAN`: `TRUE, FALSE`
- `DATE`: `‘2011-09-22’`
- `TIME`: `’15:00:02’, ’15:00:02.5’`
- `TIMESTAMP`: `'Jan-12-2011 10:25'`

Built-in data types -3

- These are all defined in the SQL standard.

- There is much more, e.g.,
  - specifying the precision of numeric types
  - other formats for data values
  - more types

- For what psql supports, see the documentation.

User-defined types

- Defined in terms of built-in types.
- Defined using constraints and, optionally, a default value.

- Example:

```
create domain Grade as int
    default null
    check (value>=0 and value <=100);

create domain Campus as varchar(4)
    default ‘StG’
    check (value in (‘StG','UTM','UTSC'));
```

Semantics of type constraints

- Constraints on a type are checked every time a value is assigned to an attribute of that type.
- Constraints create a powerful type system.

Semantics of default values

- The default value for a type is used when no value has been specified.

- Useful! We can run a query and insert the resulting tuples into a relation even if the query does not give values for all the attributes in the relation if the types of the missing attributes have default values.

- Table attributes can also have default values.

- The difference:
  - attribute default: for that one attribute in that one table
  - type default: for every attribute defined to be of that type

Keys and Foreign Keys

Key constraints

- Declaring that a set of one or more attributes are the `PRIMARY KEY` for a relation means:
  - they form a key
  - their values will never be null (we don't need to separately declare that)

- Every table must have 0 or 1 primary key.
  - A table can have no primary key, but in practice, every table should have one.
  - You cannot declare more than one primary key.

Declaring primary keys

- For a single-attribute key, the constraint can be part of the attribute definition.

```
create table T1 (
    ID integer primary key,
    name varchar(25));
```

- It can also be placed at the end of the table definition. (This is the only way for multi-attribute keys.) The brackets are required.

```
create table Blah (
    ID integer,
    name varchar(25),
    primary key (ID));
```

Uniqueness constraints - 1

- Declaring that a set of one or more attributes is `UNIQUE` for a relation means:
  - they form a key (unique and no subset is)
  - their values *can* be null → if they must not be NULL, we need to separately declare that
- We can declare more than one set of attributes to be `UNIQUE`.

Uniqueness constraints - 2

- If only one attribute is involved, it can be part of the attribute definition.

```
create table T1 (
    ID integer unique,
    name varchar(25));
```

- It can also be placed at the end of the table definition. (This is the only way if multiple attributes are involved.) The brackets are required.

```
create table T1 (
    ID integer,
    name varchar(25),
    unique (ID));
```

How nulls affect the “unique” constraint

- **For uniqueness constraints, no two nulls are considered equal.**
- E.g., consider:

```
create table Testunique (
    first varchar(25),
    last varchar(25),
    unique(first, last))
```

- This would prevent two insertions of
  `('Yijun', ‘Xi')`
- But it would allow two insertions of
  `(null, 'Schoeler')`

UNIVERSITY OF TORONTO

Foreign key contraints

- E.g. in table Took:
  `foreign key (sID) references Student`

- Means that attribute sID in this table is a foreign key that references the primary key of table Student.
  - Every value for sID in this table must actually occur in the Student table.

- Requirements:
  - Must be declared either primary key or unique in the "home" table.

Declaring foreign keys

- Declare with the attribute (only possible if just a single attribute is involved) or as a separate table element.

- Can reference attribute(s) that are not the primary key as long as they are unique; just name them.

```
create table People (
   SIN integer primary key,
   name text,
   OHIP text unique);

create table Volunteers (
   email text primary key,
   OHIPnum text references People(OHIP));
```

Enforcing foreign-key constraints

- Suppose there is a foreign-key constraint from relation R to relation S.

- When must the DBMS ensure that:
  - the referenced attributes are `PRIMARY KEY` or `UNIQUE`?
  - the values actually exist?

- What could cause a violation?
- We get to define what the DBMS should do.
  - ➢ This is called specifying a "reaction policy."

Other Constraints and Assertions

“check” constraints

- E.g., a check clause on a user-defined domain:

```
create domain Grade as smallint
    default null
    check (value>=0 and value <=100);
```

- We can also define a check constraint
  - on an attribute
  - on the tuples of a relation
  - across relations

Attribute-based "check" constraints - I

- Defined with a single attribute. It constrains its value in every tuple.
- Can only refer to that attribute.
- Can include a subquery.

Example:

```
create table Student (
  sID integer,

  program varchar(5) check
    (program in (select post from P)),

  firstName varchar(15) not null, ...);
```

- The condition to be checked can be anything that could go in a WHERE clause.

Attribute-based "check" constraints -2

- Checked only when a tuple is inserted into that relation, or its value for that attribute is updated.
- If a change somewhere else violates the constraint, the DBMS will not notice. E.g.,
  - If a student's program changes to something not in table P, we get an error.
  - But if table P drops a program that some student has, there is no error.

"not null" constraints

- We can declare that an attribute of a table is NOT NULL.

```
create table Course(
  cNum integer,
  name varchar(40) not null,
  dept Department,
  wr boolean,
  primary key (cNum, dept));
```

- In practice, many attributes can/should be `not null`.

Tuple-based “check” constraints

- Defined as a separate element of the table schema, so can refer to *any* attributes of the table.

- The condition to be checked can be anything that could go in a `WHERE` clause, and can include a subquery.

- Example:

```
create table Student (
  sID integer,
  age integer, year integer,
  college varchar(4),

 check (year = age - 18),
 check college in
       (select name from Colleges));
```

When they are checked

- Only when a tuple is inserted into that relation, or it updated.

- If a change somewhere else violates the constraint, the DBMS will not notice.

How nulls affect "check" constraints

- A check constraint **only fails if it evaluates to false.**

- It is not picky like a WHERE condition.
- E.g.: `check (age > 0)`

age	Value of condition	CHECK outcome	WHERE outcome
19	TRUE	pass	pass
-5	FALSE	fail	fail
NULL	unknown	pass	fail

UNIVERSITY OF TORONTO

Example

- Suppose we created this table:

```
create table Frequencies(
    word varchar(10),
    num integer,
    check (num > 5));
```

- It would allow us to insert `('hello', null)` since `null` passes the constraint `check (num > 5)`

- If we need to prevent that, use a "not null" constraint.

```
create table Frequencies(
    word varchar(10),
    num integer not null,
    check (num > 5));
```

Naming your constraints

- If we name our constraint, we will get more helpful error messages.

- This can be done with any of the types of constraint we've seen.

- Add

  `constraint «name»`

  before the

  `check («condition»)`

Examples

```
create domain Grade as smallint
  default null
  constraint gradeInRange
      check (value>=0 and value <=100));

create domain Campus as varchar(4)
   not null
   constraint validCampus
     check (value in ('StG', 'UTM', 'UTSC'));

create table Offering(...
  constraint validCourseReference
   foreign key (cNum, dept) references Course);
```

- **The order of constraints doesn’t matter, and it doesn’t dictate the order in which they’re checked.**

Assertions

- **Check constraints apply to an attribute or table.** They can't express constraints across tables, e.g.,
  - Every loan has at least one customer, who has an account with at least $1,000.
  - For each branch, the sum of all loan amounts < the sum of all account balances.

- Assertions can express cross-table constraints:

```
create assertion (<name>) check
(<predicate>);
```

Assertions are powerful but costly

- SQL has a powerful syntax for expressing logical predicates, including quantification.

- Assertions are costly because
  - They have to be checked upon every database update (although a DBMS may be able to limit this).
  - Each check can be expensive.

- Testing and maintenance are also difficult.

- Assertions must be used with great care.

Triggers

- Assertions are powerful, but costly.

- Check constraints are less costly, but less powerful.

- Triggers are a compromise between these extremes:
  - They are cross-table constraints, as powerful as assertions.
  - **But you control the cost by having control over when they are applied.**

The basic idea

- You specify three things.
  - **Event:** Some type of database action, e.g., `after delete on Courses` or `before update of grade on Took`
  - **Condition:** A boolean-valued expression, e.g., `when grade > 95`
  - **Action:** Any SQL statements, e.g., `insert into Winners values (sID)`

Reaction Policies

Example

- Suppose R = Took and S = Student.

- What sorts of actions must be rejected?

Possible policies

- `cascade`: propagate the change to the referring table
- `set null`: set the referring attribute(s) to null

- There are other options we won't cover as many DBMSs don't support all of them.

- If we say nothing, the **default** is to forbid the change in the referred-to table.

Reaction policy example

- In the University schema, what should happen in these situations:
  - csc343 changes number to be 543
  - student 99132 is deleted
  - student 99132's grade in csc148 is raised to 85.
  - csc148 is deleted

UNIVERSITY OF TORONTO

Note the asymmetry

- Suppose table R refers to table S.
- We can define “fixes” that propogate changes backwards from S to R.
- (We define them in table R because it is the table that will be affected.)
- We **cannot** define fixes that propagate forward from R to S.

Syntax for specifying a reaction policy

- Add your reaction policy where you specify the foreign key constraint.

- Example:

```
create table Took (
   ...
   foreign key (sID) references Student
           on delete cascade
   ...
);
```

What you can react to

- A reaction policy can specify what to do either
  - `on delete`, i.e., when a deletion creates a dangling reference,
  - `on update`, i.e., when an update creates a dangling reference,
  - or both
  - Example:
    `on delete restrict on update cascade`

What your reaction can be

- A policy can specify one reaction (there are other reactions not covered here), e.g.:
  - `restrict`: Don't allow the deletion/update.
  - `cascade`: Make the same deletion/update in the referring tuple.
  - `set null`: Set the corresponding value in the referring tuple to null.

Semantics of Deletion

- What if deleting one tuple affects the outcome for a tuple encountered later?
- What if deleting one tuple violates a foreign key constraint, but deleting others does not?

- To prevent such interactions, deletion proceeds in two stages:
  - Mark all tuples for which the WHERE condition is satisfied.
  - Go back and delete the marked tuples.

Updating the schema itself

- Alter: alter a domain or table

```
alter table Course
    add column numSections integer;
alter table Course
    drop column breadth;
```

- Drop: remove a domain, table, or whole schema

```
drop table course;
```

- This different from

```
delete from course;
```

Which deletes the content of `course`

There's more to SQL DDL

- For example, we can also define:
  - indices: for making search
  - privileges: who can do what with what parts of the database