SQL Data Definition

Thomas Heinis

t.heinis@imperial.ac.uk
wp.doc.ic.ac.uk/theinis

Imperial College
London

Data Definition

- SQL's Data Definition Language (DDL) is concerned with schema creation and modification; as well as the specification of constraints and performance options such as materialised views and indexing.
- We'll mainly look at base relations (stored tables) and briefly at derived relations (computed views).
- Constraints in SQL include: domain (type) constraints, primary key constraints, foreign key constraints, unique constraints, not null constraints, check constraints, and assertions.

Creating a Relation

The **create table** statement is used to create a new named relation and declare its schema. The relation is persistent and stored on disk in specially organised files.

Example: movie(<u>title</u>, <u>year</u>, length, genre)

```
create table movie (
        title     varchar(120),
        year      int default 2011,
        length    int default 0,
        genre     char(20),
        primary key (title, year)
)
```

The attributes belonging to the primary key are usually underlined in textbooks.

The order that attributes are defined is sometimes used by other SQL statements. For example **select ***, and inserting values into a relation when no attribute list is given.

We can remove relations with **drop table**, e.g. **drop table** movie

Modifying the Schema

The **alter table** statement is used to add or remove attributes for a relation.

Example: movie(<u>title</u>, <u>year</u>, length, genre)

alter table movie **add** studio **char**(16) **default** ‘’;
alter table movie **drop** length;

After these two modifications the schema becomes

movie(<u>title</u>, <u>year</u>, genre, studio)

and each tuple of movie will now have a studio attribute set to the empty string. We can set different studio values by following **alter table** with one or more **update** statements.

Primary Key

- Each relation should have a **primary** key *(a candidate key*) that determines the other attributes in the relation and is used to uniquely identify each tuple of the relation.
- The primary key is also used to enforce **foreign key constraints** on the relation (see later)
- Only one primary key is permitted for a relation. If there is a choice of candidate keys, then choose one where the key will never be updated (or very rarely updated), otherwise choose one with few attributes (a small memory footprint).
- Other candidate keys can be defined using **unique** constraints (see later).

Primary Key Constraints

There are two constraints enforced by SQL over primary keys:

1. **null**s are not permitted in a primary key.

2. Primary key values (as a whole) must be unique i.e. no two tuples can have the same primary key value (i.e. where all attributes are the same).

If any database operation attempts to violate one of these constraints then the operation will fail. For example, if we attempt to assign a **null** value to a primary key attribute or if we attempt to insert a tuple where a tuple with the same primary key value already exists. Example: The insertion

insert into movie(title, genre) **values** (‘Up’, ‘cartoon’)

will fail since **insert** will attempt to assign **null** to **year**, and **year** is an attribute of the primary key.

Unique Constraints

Additional candidate keys (or superkeys or any attributes for that matter) can be declared using **unique constraints, which ensure that no two tuples can have the same set of values for the attributes listed in unique, i.e. that every tuple must have a unique value for the attributes listed in unique (as a whole).**

Example: movie(<u>title</u>, <u>year</u>, length, genre, ISAN)

```
create table movie (
    title     varchar(120),
    year      int default 2011,
    length    int default 0,
    genre     char(20),
    ISAN      char(24),
    primary key (title, year),
    unique (ISAN)
)
```

Not Null Constraints

We can add a **not null** constraint to any attribute to have the database prohibit assignment of **null** to that attribute.

Example: movie(<u>title</u>, <u>year</u>, length, genre, ISAN)

```
create table movie (
    title    varchar(120),
    year     int default 2011,
    length   int not null default 0,
    genre    char(20) not null,
    ISAN     char(24) not null,
    primary key (title, year),
    unique (ISAN)
)
```

A primary-key constraint is just a combination of a unique constraint and a not-null constraint.

Exercise

Q. Declare the SQL schema for the following relation - don't worry about getting all the details right - make 'educated' guesses.

staff(id, opened, openedby, updated, updatedby, validfrom, validto,
login, email, lastname, firstname, telephone, room)

Solution

Q. Declare the SQL schema for the following relation - don't worry about getting all the details right - make 'educated' guesses.

staff(id, opened, openedby, updated, updatedby, validfrom, validto,
login, email, lastname, firstname, telephone, room)

```
create table staff (
        id          int,
        opened      timestamp       not null default current_timestamp,
        openedby    char(10)        not null,
        updated     timestamp       not null default current_timestamp,
        validfrom   date            not null default current_date,
        validto     date            not null default date '2020-12-31',
        login       char(12)        not null,
        email       varchar(80)     not null,
        lastname    char(30)        not null,
        firstname   char(30)        not null,
        telephone   char(20)        not null default '',
        room        char(10)        not null default '',
        primary key (id), unique(login), unique(email)
)
```

Check Constraints

Attribute types and **not null** constraints allow us to limit the values that we store. **check** constraints allow us to define predicates that must be satisfied when a tuple is *inserted* or *updated*.

Example: movie(<u>title</u>, <u>year</u>, length, genre, ISAN)

```
create table movie (
        title   varchar(120),
        year    int,
        length  int,
        genre   char(20),
        ISAN    char(24),
        primary key (title, year),
        check(year between 1900 and 2020),
        check(genre in ('sf', 'comedy', 'drama', 'western'))
)
```

Check Constraints

Although **check** constraints are typically simple checks on a single attribute they can be arbitrary expressions involving several attributes and/or a query.

Example: movie(<u>title</u>, <u>year</u>, length, genre, ISAN)

```
create table movie (
        title     varchar(120),
        year      int,
        length    int,
        genre     char(20),
        ISAN      char(24) not null,
        primary key (title, year),
        check(ISAN in (select no from ISANcatalog))
)
```

Exercise

Q. For the staff relation add a check constraint for validfrom and validto and one for room given the following additional relation:

building(id, ..., room, area, desks, occupancy)

```
create table staff (
    id          int,
    ...
    updated     timestamp       not null   default current_timestamp,
    validfrom   date            not null   default current_date,
    validto     date            not null   default date '2020-12-31',
    ...
    room        char(10)        not null   default '',

    primary key (id), unique(login), unique(email),

)
```

Solution

Q. Add a check constraint for validfrom and validto **and one for** room given the relation building(id, ..., room, area, desks, occupancy)

```
create table staff (
        id          int,
        ...
        updated     timestamp     not null default current_timestamp,
        validfrom   date          not null default current_date,
        validto     date          not null default date '2020-12-31',
        ...
        room        char(10)      not null default '',

        primary key (id), unique(login), unique(email),

        check(validto>=validfrom),
        check(room in (select room in building)),

        check(updated>=current_timestamp)
)
```

Assertions

It's also possible to declare **assertion**s, which are **check** constraints over data in several relations.

Example:

movieboss(<u>name</u>, address, networth)
studio(<u>name</u>, address, boss)

```
create assertion nopoorbosses check (
        not exists (
                select s.name
                from studio s join movieboss m on (s.boss=m.name)
                where m.networth < 100000000
        )
)
```

Although assertions are a very powerful feature of SQL they are hard to implement efficiently. **Triggers** are an alternative, more powerful and more operational approach for letting the programmer deal with constraint checking when data is modified.

Naming Constraints

It's good practice to name constraints. This allows us to drop them using **alter table** but also clarifies error messages when a constraint violation is reported.

Example: movie(<u>title</u>, <u>year</u>, length, genre, ISAN)

```
create table movie (
    title     varchar(120),
    year      int,
    length    int,
    genre     char(20),
    primary key (title, year),
    constraint uniqueISAN unique (ISAN),
    constraint yearCheck check(year between 1900 and 2020),
    constraint genreCheck check(genre in ('sf','comedy'))
)
```

Foreign Key Constraints

Foreign key constraints specify that the value of one or more attributes in a relation must match (reference) values of a primary key or **unique** constraint (candidate key) in another (referenced) relation. This is an example of **referential integrity**.

Example: movie(<u>title</u>, <u>year</u>, length, genre, ISAN)
casting(title, year, name)

```
create table casting (
        title     varchar(120),
        year      int,
        name      varchar(60),
        foreign key (title, year) references movie (title,year)
)
```

Exercise

Q. Write foreign key constraints for some of the following relations:

staff(id, login, email, lastname, firstname, telephone, room, deptrole, department)
student(id, login, email, lastname, status, entryyear, externaldept)
course(id, code, title, syllabus, term, classes, popestimate)
class(id, ***degreeid***, yr, degree, degreeyr, major, majoryr, letter, letteryr)
degree(id, title, code, major, grp, letter, years)
xcourseclass(***courseid***, ***classid***, required, examcode)
xcoursestaff(***courseid***, ***staffid***, staffhours, role, term)
xstudentclass(***studentid***, ***classid***)
xstudentstaff(***studentid***, ***staffid***, role, grp, projecttitle)

Solution

Q. Write foreign key constraints for some of the following relations:

staff(id, login, email, lastname, firstname, telephone, room, deptrole, department)
student(id, login, email, lastname, status, entryyear, externaldept)
course(id, code, title, syllabus, term, classes, popestimate)
class(id, ***degreeid***, yr, degree, degreeyr, major, majoryr, letter, letteryr)
foreign key (degreeid) **references** degree (id),
degree(id, title, code, major, grp, letter, years)

xcourseclass(***courseid***, ***classid***, required, examcode)
foreign key (courseid) **references** course (id),
foreign key (classid) **references** class (id),
xcoursestaff(***courseid***, ***staffid***, staffhours, role, term)
foreign key (courseid) **references** course (id),
foreign key (staffid) **references** staff (id),
xstudentclass(***studentid***, ***classid***)
foreign key (studentid) **references** student (id),
foreign key (classid) **references** class (id),
xstudentstaff(***studentid***, ***staffid***, role, grp, projecttitle)
foreign key (studentid) **references** student (id),
foreign key (staffid) **references** staff (id),

Maintaining Referential Integrity - Cascade

The default policy when referential integrity is violated in SQL is to reject the modification. However, there are two other policies that can be defined for deletes and updates. **Cascade Policy** - with this policy any update to the referenced attribute(s) is cascaded back to the foreign key. Similarly deleting a referenced tuple will result in the referencing tuple being deleted as well (which might cascade again!)

Example: movie(<u>title</u>, <u>year</u>, length, genre, ISAN)
casting(title, year, name)

```
create table casting (
        title       varchar(120),
        year        int,
        name        varchar(60),
        foreign key (title, year) references movie (title,year)
        on update cascade   on delete cascade
)
```

Maintaining Referential Integrity - Set Null/Default

Rather than cascading updates or deletes to the referencing relation, we can, instead, set the value of the foreign key to **null** using a **set null** policy or to the default value using a **set default** policy. This will lead to unmatched tuples however!

Example: movie(<u>title</u>, <u>year</u>, length, genre, ISAN)
casting(title, year, name)

```
create table casting (
        title     varchar(120) default ‘’,
        year      int default 2011,
        name      varchar(60),
         foreign key (title, year) references movie (title,year)
        on delete set null  on update set default
)
```

Exercise

Q. Rewrite *some* of the foreign key constraints maintaining referential integrity.

staff(id, login, email, lastname, firstname, telephone, room, deptrole, department)
student(id, login, email, lastname, status, entryyear, externaldept)
course(id, code, title, syllabus, term, classes, popestimate)
class(id, ***degreeid***, yr, degree, degreeyr, major, majoryr, letter, letteryr)
degree(id, title, code, major, grp, letter, years)
xcourseclass(***courseid***, ***classid***, required, examcode)
xcoursestaff(***courseid***, ***staffid***, staffhours, role, term)
xstudentclass(***studentid***, ***classid***)
xstudentstaff(***studentid***, ***staffid***, role, grp, projecttitle)

Solution

Q. Rewrite *some* of the foreign key constraints maintaining referential integrity.

staff(id, login, email, lastname, firstname, telephone, room, deptrole, department)
student(id, login, email, lastname, status, entryyear, externaldept)
course(id, code, title, syllabus, term, classes, popestimate)
class(id, ***degreeid***, yr, degree, degreeyr, major, majoryr, letter, letteryr)
foreign key (degreeid) **references** degree (id) **on delete cascade on update cascade**,
degree(id, title, code, major, grp, letter, years)

xcourseclass(***courseid***, ***classid***, required, examcode)
foreign key (courseid) **references** course (id) **on delete cascade on update cascade**,
foreign key (classid) **references** class (id) **on delete cascade on update cascade**,
xcoursestaff(***courseid***, ***staffid***, staffhours, role, term)
foreign key (courseid) **references** course (id) **on delete cascade on update cascade**,
foreign key (staffid) **references** staff (id) **on delete cascade on update cascade**,
xstudentclass(***studentid***, ***classid***)
foreign key (studentid) **references** student (id) **on delete cascade on update cascade**,
foreign key (classid) **references** class (id) **on delete cascade on update cascade**,
xstudentstaff(***studentid***, ***staffid***, role, grp, projecttitle)
foreign key (studentid) **references** student (id) **on delete cascade on update cascade**,
foreign key (staffid) **references** staff (id) **on delete cascade on update cascade**,

Views

Views are relations that are defined using a query (a select). Views are not physically stored on disk unless they are materialised (see later).

Example: movie(<u>title</u>, <u>year</u>, length, genre)
casting(title, year, name)

Changes in the underlying relations are reflected in the view

create view comedies **as**
select title, year **from** movie **where** genre='comedy';

create view actorgenre(actorname, moviegenre) **as**
select distinct name**,** genre **from** movie **join** casting
using (title,year);

Views in a query are like subqueries.

Views can be queried just like stored relations (tables):

select * **from** comedies **where** year=2010;
select * **from** actorgenre **where** moviegenre = 'comedy';

Exercise

Q. Write a view on staff for all staff who are currently here (i.e. currently valid).

staff(id, opened, openedby, updated, updatedby, validfrom, validto,
login, email, lastname, firstname, telephone, room)

Q. Write a view on staff who were here in the academic year 2008 to 2009 (October to September). Tricky.

Solution

Q. Write a view on staff for all staff who are currently here (i.e. currently valid).

staff(id, opened, openedby, updated, updatedby, validfrom, validto,
login, email, lastname, firstname, telephone, room)

create view staffcurr **as**
select * **from** staff **where**
validfrom <= current_date **and** validto >= current_date

Q. Write a view on staff who were here in the academic year 2008 to 2009 (October to September).

create view staff0809 **as**
select * **from** staff **where**
(validto >= '2008-10-01' **and** validfrom <= '2009-09-30')

Uses for Views

Views allow us to:

- Declare commonly used subqueries (relational expressions). Views can also be defined in terms of other views (nested views)
- Declare a relation over several relations using joins, products etc.
- Declare a relation over calculated (expressions) and aggregated data (sums, averages, mins, counts) etc
- Partition data using a selection e.g. into years.
- Restrict access to a relation by providing access to a view not the whole relation.

Materialised Views

Views are normally recomputed each time they are needed. If a view is used sufficiently often then *it might* be more efficient to materialise (store) the view at the cost of extra storage and extra time to keep the view up-to-date when the underlying relations are changed by insertions, updates, and deletions.

Materialised view maintenance can be expensive however, for example, requiring lots of changes to the underlying relations versus few queries on the views. The decision is a tradeoff between extra-storage and view maintenance costs and the faster speed of querying a materialised view.

Rather than keeping a materialised view eagerly up-to-date, some RDBMSs allow a materialised view to be brought up-to-date only when the view is accessed (lazy maintenance). Others allow the view to become “stale” and only update it periodically e.g. when database activity is low (overnight) or explicitly by a refresh command. Others create/remove a materialised view transparently as a query optimisation.

Materialised views are a non-standard extension.

Updateable Views

Views are normally read-only, used to retrieve data. One could consider **updateable views** - views that allow inserts, deletes and updates directly on the view.

Updatable views usually don’t make sense e.g. deleting a tuple in actorgenre for example.

Even in simple cases, for example, inserting a tuple into comedies, there is the issue of missing attributes - attributes in the underlying relations that are not in the relation. In this case we could set them to null or their default value.

SQL has a complex set of rules for defining an updateable view including:

1. Only one relation in the from clause
2. Only attributes in the projection list, no expressions, aggregates, distinct etc.
3. Any attribute not in the projection list can be set to null
4. No group by and having clauses.
5. No subqueries.

Indexes

When relations have many tuples it can be very slow to scan the relation tuple-by-tuple in order to satisfy a query.

Example: **select** * **from** movie **where** year=2010 **and** genre='comedy';

If there were 10,000 movies then reading and testing all 10,000 tuples might be a little slow. Imagine if there were 100 million tuples in the relation.

Indexes are *copies* of an attribute's data that are automatically maintained by the RDBMS but can be searched very very quickly. For the example, we could create an index on both year and genre together, or separate indexes on one or both which might be more flexible:

create index yearindex **on** movies(year);
create index genreindex **on** movies(genre);

Like materialised views, there is a tradeoff between the space needed for indexes and the cost of maintaining them vs the greater speed of access to the indexed data.