Introductory Databases
Revision Notes

James Brown

April 21, 2017

Contents

1 Introduction

These are notes I have written in preparation of the 2017 Introductory Databases exam. This year the module was run by Bob Hendley (R.J.Hendley@cs.bham.ac.uk).

2 Data Definition Language

2.1 CREATE

The `CREATE` command creates a table within our database. We must supply a name for the table, the fields the table should contain (the name and the type of the field) and any constraints on the values that we wish to have. Example:

```
CREATE TABLE Student (
    sid    INTEGER   NOT NULL UNIQUE,
    dob    CHAR(10),
    login  CHAR(20)  UNIQUE,
    course CHAR(10)
)
```

It is best practice to include constraints on the data, examples include `NOT NULL` and `UNIQUE`. Usually constraints will be defined when you define the table, but it is possible to add them after the creation.

2.2 DROP and ALTER

The `DROP` command deletes the table - for example: `DROP TABLE Student`. `ALTER` modifies the table definition and adds `NULL` as the value of any new columns. For example: `ALTER TABLE Student ADD COLUMN year_of_study INTEGER`.

The `ALTER` command is crucial as databases are persistent and we cannot rebuild them - so we have to evolve them on the fly as constraints and needs change.

2.3 Data Types

- `BOOLEAN`: either `TRUE`, `FALSE` or `NULL`.
- `CHAR(size)` or `CHARACTER(size)`
- **Strings** - e.g. `VARCHAR`
- `INTEGER` or `INT`
- `REAL`, `DOUBLE` or `FLOAT`
- `NUMERIC` or `DECIMAL`
- `DATE`
- `TIME`

Other data types are available, but these are the most widely used.

2.4 Constraints

Constraints place restrictions on the values that can be inserted into a database - they will be checked and enforced by the DBMS. They can be considered as metadata which are used to make explicit domain constraints and maintain the integrity of the database. Constraints aren't an excuse for laziness. It's important to distinguish between hard constraints, such as every student must have a unique sid, and desirable constraints, such as every student has a login. We manipulating data, if we violate a constraint then the operation will fail.

We've already seen examples of some constraints - the domain of the data (e.g. `INTEGER`) as well as things like `NOT NULL` and `UNIQUE`. We may also want to add constraints on the range of values, information about keys (which is especially important to ensure integrity across tables) and arbitrary checks.

2.5 Keys

One of the most common constraints is the **primary key** - it enforces `UNIQUE` and `NOT NULL`. It's important to note that a primary key is more than just a data constraint, it actually signals database structure as well. We can define a primary key in two different ways like so:

```
CREATE TABLE Student (
    sid    INTEGER   NOT NULL UNIQUE,
    dob    CHAR(10),
    login  CHAR(20)  UNIQUE,
    course CHAR(10)
    PRIMARY KEY (sid)
)
```

```
CREATE TABLE Student (
    sid    INTEGER   NOT NULL UNIQUE,
    dob    CHAR(10),
    login  CHAR(20)  UNIQUE,
    course CHAR(10)
    CONSTRAINT StudentsKey PRIMARY KEY (sid)
)
```

Foreign keys define links to another table and are usually used to specify the primary key in that other table. Any row that is inserted into the table with a foreign key must satisfy the constraint that the foreign key in this table must be matched by a key in the other table.

What happens if we modify a table so that a referenced key is removed? We have a few options. Typically we will either forbid the operation with `RESTRICT` or `NO ACTION` but we may also want to delete rows that reference the deleted key with `CASCADE`. `NO ACTION` is the default option. It is possible to also `SET DEFAULT` or `NULL` but these are rarely ever advised!

3 Manipulating Data

4 Database Design

4.1 ER Diagrams

5 Database Normalisation

We want to normalise our databases so that we can remove functional dependencies from a table. Should we not normalise our tables we may get **insertion anomalies**, **update anomalies**, **deletion anomalies**. There are various levels of normal forms, and typically these all work by identifying a removing functional dependencies that exist in the data. We can do this by creating new tables and unpacking the data.

There are both advantages and disadvantages to normalising our database. On the positive side, data is only represented once and consistency is improved. Maintenance of the database also becomes much easier. On the other hand, it slows everything down as we need more joins when querying data. The modelling of the data also becomes more complex.

In large corporate systems it's widely regarded that at least 3NF is required as there are many insertions, deletions and updates. The various anomalies that can occur in the end are much worse than the incurred performance hit, so it's best to just normalise the database. For smaller systems, the extra cost of design may not be worth the benefits. It's also important to consider how the system will be used - is it worth doing in systems that will never be updated? If efficiency and performance is critical it also may not be worth doing.