Introductory Databases
Revision Notes

James Brown

May 2, 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

3.1 INSERT

The `INSERT` command inserts data into a table. In general, we specify columns, rows and values to insert.

```
INSERT INTO Student VALUES
    (28, '28/01/09', 'reh', 'CS')
```

```
INSERT INTO
    Student(sid, dob, login, course)
    VALUES(28, '28/01/09', 'reh', 'CS')
```

3.2 UPDATE

We use the `UPDATE` command to modify values in rows. We specify the table and rows and attributes and their new values. All rows that meet the condition will be updated.

```
UPDATE Student
    SET course='CS'
    WHERE sid=23
```

3.3 DELETE

The `DELETE` command deletes rows from the table. We specify conditions for the rows, and all that meet the criteria are deleted.

```
DELETE FROM Students
WHERE sid=23
```

3.4 SELECT

The SELECT command is used to retrieve data from one table or combining data from several (more common). The result of a SELECT command is another table that meets the specification of the SELECT statement. SELECT allows the specification of columns in the result, conditions on rows (which can be arbitrarily compelex), ordering of rows, aggregate functions and more.

```
SELECT Student.sid, Student.login
FROM Student
WHERE Student.course='cs'
```

There are a few other things we can do to a SELECT statement. The DISTINCT keyword will remove duplicate rows. Also we can attach a label to a returned column using AS.

```
SELECT COUNT (DISTINCT course) AS NoC
FROM Student
```

By using ORDER BY we can specify the order of the rows to be returned. ORDER BY uses the natural order of the column, and we can specify whether this should be ascending or descending. We can specify multiple columns to order by which will sort by the first column, and then by the next if there are any matches. In the example below, it will order all courses. After doing this, it will order all the students on a particular course by their login number.

```
SELECT *
FROM Student
ORDER BY course, login
```

The selection criteria in the WHERE clause may contain arbitrarily complex expressions. We can use logical operators such as AND/OR, comparrison operators (less than, greater than), IN/NOT IN and more. LIKE provides pattern matching for strings. _ matches any character and % matches 0 or more characters. Most implementations of databases will also allow the use of regexps.

3.5 Joins

Selecting data from a single table is limiting. In almost all cases, we need to join data from several tables. An example can be seen below:

```
SELECT Student.sid, Marks.mid, Marks.mark
FROM Student, Marks
WHERE Student.sid = Marks.sid
```

There are several types of join - the most common of which is the inner join. Other joins all follow roughly the same pattern. The example join from above could also be written as such:

```
SELECT Student.sid, Marks.cid, Marks.mark
FROM Student
INNER JOIN Marks
ON Student.sid = Marks.sid
```

The fist example is more compact that the inner join and known as 'implicit notation'. House styles may preference one over the other, but conceptually they perform the exact same operation. They form the cross product of the set of tables, select columns, select rows that meet the WHERE condition and if DISTINCT remove duplicates. An important detail is that if there is ambiguity in a field name they must be fully qualified. It may be worth fully qualifying regardless.A

We may use a LEFT JOIN or RIGHT JOIN. These are like INNER JOINs, but include rows in the left/right table that are not matched. NULLs are added accordingly.

Another alternative is a FULL JOIN. This is just like an INNER JOIN but rows in the left and right tables that are not matched are also included. NULL values are added accordingly.

```
SELECT sid, marks
FROM Students
LEFT JOIN Marks
ON Student.sid=Marks.sid
```

```
SELECT sid, marks
FROM Students
RIGHT JOIN Marks
ON Student.sid=Marks.sid
```

Another possibilty is to use set operations, such as `UNION`, `INTERSECT`, `EXCEPT`. We can formulate a query as a set operation on two results, which may be easier and clearer than a complex expression. It may also be more efficient.

It is possbile to have subqueries whose reslts are then used in the outer query. This can make queries easier to formulate, clearer and more efficient.

```
SELECT S1.sid
FROM Student S1
WHERE S1.dob > ALL (SELECT S2.dob
                    FROM Student S2
                    WHERE S2.course='cs')
```

Nesting of queries can go arbitrarily deep. We can also reference the outer row within the nested query (in this case, reference S1 within the nested query). We don't need to use different names for Student in the two queries, but if the inner query references the outer, then we must.

We can group data returned from a query with the `GROUP BY` command. This allows for operations over groups of rows. We can use a `HAVING` command to specify selection criteria for groups. In order to group, the tables are joined, rows are filtered using the `WHERE` clause, the table is grouped using the attributes specified in the `GROUP BY` clause and then rows are removed where the `HAVING` clause does not return true.

```
SELECT customerName, SUM(orderQuantity)
FROM Sales
GROUP BY customerName
HAVING SUM(orderQuantity)>5
```

4 JDBC

For the exam we are not required explicitly code a solution, but may be asked questions about why we would take a certain approach with our code or to fix a problem in a piece of code given to us.

4.1 Prepared Statements

Prepared statements allow a SQL statement to be parameterised. This is more efficient if it's run many times, more secure and may also be more convenient in a generic command with parameters supplied by the user. One large benefit is that they protect from SQL injection. Anywhere the user has the opportunity to input data to a query, they may also place an arbitrary piece of SQL. In a normal statement, the database will read this SQL and execute it with no problem, as if it had come from the programmer. A prepared statement will sanitize inputs so that an SQL injection cannot happen. For this reason alone, they should always be used.

4.2 Transactions

There are some compund operations that must be performed in their entirety (as if they were atomic). For example, moving money from one account to another. If partially completed then the database will be inconsistent, and there may be real-world consequences. Partial completions could be caused by a number of things: application crashes, DB server crashes, other user interference etc.

The DBMS will provide support for these situations through **transactions**, which guarantee correct behavious and ACID properties. They implement mechanisms that ensure correct behaviour and robustness, possibly achieved through locks and logs. Usually we do not need to worry about the implementation of transactions in the DBMS.

In JDBC, by default each operation is an independant transaction. `Conn.setAutoCommit(false)` causes the program to explicitly handle transactions. To commit a transaction, we use `Conn.commit()` and the operations become permenant. Transactions can be aborted by `Conn.rollback()`, which undoes all operations since the last commit.

Transactions allow for concurrent access to the database across multiple applications, all whilst maintaining correct behaviour. It allows robustness against failure, whether it be hardware or software.

In order to achieve this, transactions typically maintain it's own copy of the database. If the transaction is successful, it will write the changes through to the permanent database. If unsuccessful, these changes are rolled back. Failure may come from several reasons: explicit or implicit failure, conflicts with other transactions or the detection of deadlock.

5 Database Design

A database is usually part of a much larger system, required to capture data requirements and capture constraints. It may also have non-functional requirements such as reliability, consistency, extensibility, security and performance. These factors all drive design, analysis, testing etc. of the DB component and integrated systems. We need to generate a description of the data, it's organisation and its mapping onto the database system. There are many different methodologies that can drive this process and notations that can be used. Almost universally, a data design is described used **Entity Relationship Diagrams (ER diagrams)**. They capture conceptual design and the data models which can be easily mapped to the database schema. We may partiton our ER diagrams for large, complex designs.

5.1 ER Diagrams

ER diagrams use different shaped components to represent different things. Ovals are attributes, rectangles are entity sets, diamonds are relation sets, lines show links. Labels on lines indicate role and cardinality. Double outlined boxes show weak entity sets.

6 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.