COMP1314 Coursework

Philbert Poku (pkp1u24@soton.ac.uk)

Brief You have just been promoted to the position of "Data Manager" at the prestigious University of Northeasthampton. Unfortunately, their records are a mess, and it's up to you to fix them.

1 Structured Data

The University of Northeasthampton currently stores all of their data in a pair of xml files. This would not be an ideal format at the best of times, but it is made worse by the way they have chosen to format their files.

1.1 Ex1

Bash Script

```bash
#!/bin/bash

xml_file="$1"
csv_file="$2"

# Extract the start and end tags
start_tag=$(sed -n '3p' "$xml_file")
tagname=$(echo "$start_tag" | sed -E 's/^<([^>]+)>$/\1/' | xargs)
# Extract and trim tag name
end_tag="</$tagname>"

# Extract headers and save to the CSV file
awk -v start_tag="$start_tag" -v end_tag="$end_tag" '
  BEGIN { block_started = 0 }

  $0 == start_tag { block_started = 1; next }
  $0 == end_tag { exit }

  block_started {
    if (match($0, /<([^\/>]+)(\/?)>/, matches)) {
      tag = matches[1]
      if (!(tag in seen)) {
        seen[tag] = 1
        headers = headers (headers ? "," : "") tag
      }
    }
  }

  END { print headers }
' "$xml_file" > "$csv_file"
```

This code section begins with taking an xml file and csv file as positional arguments. We then extract the start tag and then dynamically create the end tag to know when each data block begins and ends. Firstly, we must extract the headers and we can do this using a for loop in awk where it matches what is in between the tags to something it has already seen before. If it has not been seen it is added to the string headers. It then prints the headers and adds it to the csv file.

Bash Script

```
# Extract data from every line
grep -v '<?xml' "$xml_file" | \
awk '
  # Process <faculty> blocks
  /<faculty>/ && !/<\/faculty>/ {
    block_start=1
    block=""
    current_tag="faculty"
    next
  }
  /<\/faculty>/ && !/<faculty>/ && block_start && current_tag ==
  ↪  "faculty" {
    block_end=1
  }
  # Process <student> blocks
  /<student>/ && !/<\/student>/ {
    block_start=1
    block=""
    current_tag="student"
    gsub(/<student> */, "")
  }
  /<\/student>/ && block_start && current_tag == "student" {
    block_end=1
  }
  # Process completed blocks
  block_start && block_end {
    print block
    block=""
    block_start=block_end=0
    next
  }
  block_start && current_tag == "faculty" {
    line = gensub(/.*>([^<]*)<.*/, "\\1", "g")
    # Extract content between tags
    block = block (block ? "," : "") line
  }
```

Bash Script

```
  block_start && current_tag == "student" {
    # Handle <address> tag - start
    if ($0 ~ /<address>/) {
      line = gensub(/<address>/, "\"", "g")
      # Replace <address> with " for formatting purposes
      getline next_line  # Get the next line
      gsub(/^[[:space:]]+|[[:space:]]+$/, "", next_line)
      next_line = gensub(/<\/address>/, "\"", "g", next_line)
      # Replace </address> with " for formatting purposes
      line = line " " next_line
      # Concatenate the two lines
      gsub(/[[:space:]]+/, " ", line)
      block = block (block ? "," : "") line
      # Add the processed address to the block
    }
    # Treates null tags as empty fields
    else if ($0 ~ /\/>/) {
      block = block (block ? "," : "") ""
    }
    # Extract content between tags for every other line
    else {
      line = gensub(/.*>([^<]*)<.*/, "\\1", "g")
      gsub(/[[:space:]]+/, " ", line)
      block = block (block ? "," : "") line
      # Add the extracted data to the block
    }
  }
' | \
# Remove leading whitespace and comma before the first character
sed 's/^ *, *//' >> "$csv_file"  # Append the extracted data to the
↪  CSV file


```

The next part of the code is the extraction process in which we divide it into processing the faculty csv and the students csv. When the opening or closing tag of faculty is encountered this means the start or end of a block. However, if faculty is encountered on the same line we extract between the faculty tags. If a line starts with the opening address tag, it spans two lines. This means that after the opening tag is saved to a variable and by getting the data on the next line they can be concatenated together. During that, the tags are replaced with speech marks for formatting purposes due to the address containing commas. For null tags, we continue onto the next line and for every other tag we just extract from between the tags. At the end remove the whitespace and the first comma for formatting purposes

and append to the csv file.

1.2 Ex2

Bash Script

```bash
#!/bin/bash

csv_file=$1
txt_file=$2

# delete the header and sort the rest based on the first field
# write the sorted data to the text file
tail -n +2 "$csv_file" | awk -F ',' '{print $1}' | sort | uniq >
 ↪  "$txt_file"
```

This Bash script takes in a csv file and a text file as positional arguments and deletes the first line of the csv file. It does this by using the tail command to display the contents the of the file and only starts from the second line effectively deleting the header. It then uses the cut command to extract the first field of the csv file which is the student name and pipes this to the sort command to sort it alphabetically. It then outputs the result to a new text file.

2 Relational Model

Now that the data can be more easily examined, the next task is to inspect it. students.csv and faculty.csv can be considered as two relations.

2.1 Ex3

We can express each students and faculty relation in the form of R(A, B, C, D, . . .) by expressing the elements of each csv header as attributes. The faculty relation can be expressed as: faculty(faculty, building, room, capacity). The students relation can be expressed as:

`students(student_name,student_id,student_email,programme,year,address,`
`contact, module_id,module_name,module_leader,lecturer1,lecturer2,faculty,`
`building,room,exam_mark,coursework1,coursework2,coursework3)`

2.2 Ex4

For each relation, in order to list the minimal set of Functional Dependencies in the form $A \rightarrow B, C$, we have to find which attributes uniquely determine other attributes. For the faculty relation we have 4 attributes and we can identify that a faculty and a building determines a room and a room determines the capacity. Therefore the minimal set of Functional Dependencies for faculty are:

```
building → faculty
building, room → faculty
building, room → capacity
```

Similarly, for the student relation we have 19 attributes and we can identify that the student_id determines all the attributes related to information about the student and the module_id likewise for the module. However both ids can give you information about the various coursework and the exam mark. Therefore the minimal set of functional dependencies for students are: bullet point

```
student_id → student_name
student_id → student_email
student_id → year
student_id → address
student_id → contact
module_id → module_name
module_id → module_leader
module_id → lecturer1
module_id → lecturer2
module_id → faculty
module_id → building
module_id → room
student_id, module_id → exam_mark
student_id, module_id → coursework1
student_id, module_id → coursework2
student_id, module_id → coursework3
```

2.3 Ex5

A key is a set of attributes for which no two tuples in a relation instance have the same values for all attributes of the key. Therefore, to find a candidate key of our relation we must discover the minimal superkeys of our relation as all possible candidate keys are subsets of the superkey. To do this we can use a closure algorithm, which makes it easier since we already know all of the minimal functional dependencies. From the faculty relation we can start with all the possible subsets that contain more than 2 attributes from (faculty, building, room, capacity) starting with {building,room}. The closure of {building,room} is {faculty, building, room, capacity} because from building and room you can derive both the capacity and the faculty, so {building,room} is a superkey. From {building} you can uniquely determine the faculty but you cannot uniquely determine the room therefore it is not a superkey. Overall, there is 1 superkey for the faculty relation {room, building} meaning there is only one candidate key.

From the students relation we realise that there are a lot of possible subsets 2^{19} to be exact therefore having the minimal set of functional dependencies is useful here for the closure algorithm. From our functional dependencies we know {student_id} can derive the student_name, student_email, programme, year, address and contact however this does not include module information or grade information. From {module_id} we can derive the

module_name, module_leader, lecturer1, lecturer2, faculty, building and room which does not include student information or grade information. For the last functional dependency {student_id, module_id} we can derive all attributes related to student, module and grade information making it a candidate key. As it is a superkey, it must be the only candidate key.

2.4 Ex6

Now for each relation we need to identify a suitable primary key. A primary key is always one of the candidate keys therefore for the faculty relation we have a choice from only one candidate key which is {room, building}. {room, building} has all the attributes needed to uniquely identify a room and faculty. Thus {building, room} should be chosen as it has the least information required to identify a single row of attributes uniquely.

For the students relation we only have one choice and that is {student_id, module_id} as it has all the attributes needed to uniquely identify student information, module information and grade related information.

3 Normalisation

A clearer picture of the current state of the data has now been established, making it the right time to begin improving it.

3.1 Ex7

For the faculty relation the data is already in First Normal Form as the relation contains only atomic values which cannot be broken down any further single values. In addition there are no composite, multi-valued or nested values nor are there objections and arrays. There are also no repeating groups as every attribute is unique.

However, some attributes can be composed into new relations for the students relation. For example, lecturer1 and lecturer2 can be made into one attribute lecturers as it is a repeating group. We can then decompose this into a new relation cohort_lecturers(student_id, module_id, lecturers) using our primary key student_id, module_id}. The same can be done for coursework1, coursework2, and coursework3 where we decompose it into a new attribute coursework. Using our primary key, we can create another new relation courseworks(student_id, module_id,exam_mark, coursework) where we can add the exam_mark to the relation as it depends on the coursework. Therefore our original students relation is now

```
students(student\_name,student\_id,student\_email,programme,year,address,
contact,module\_id,module\_name,module\_leader,faculty,building,room)
```

3.2 Ex8

To resolve any conflicts or irregularities in the data decomposition was used for example in the students relation for there not to be a redundancy lecturers attribute it was moved into a

new relation cohort_lecturers(student_id, module_id, lecturers) and to make sure that it does not have any non-atomic values. The same was done for the courseworks relation where the redundancy was handled by creating a new relation.

3.3 Ex9

A partial key dependency is where every non-key attribute (that is not part of any candidate key) is dependent on all attributes of all candidate keys. What this means for the students relation is that all the attributes in the functional dependencies: student_id → student_name, student_email, programme, year, address, contact and module_id → module_name, module_leader, faculty, building, room are partial key dependencies as these attributes depend only on student_id and module_id and these create the only candidate key.

For the faculty relation, the functional dependency building → faculty is a partial key dependency as faculty depends on building which is a proper subset of the primary key {building, room} and faculty does not appear on any candidate key.

3.4 Ex10

To achieve 2nd Normal Form for the faculty relation we can introduce more relations through decomposition. For example, the functional dependency building → faculty can be decomposed into new relations by copying the determinant which is the part of the key they are dependent on. A new relation location(building, faculty) can be created leaving the faculty relation as faculty(building, room, capacity).

To achieve 2nd Normal Form for the students relation we can introduce more relations. For example, the functional dependencies for student_id and module_id can be decomposed into new relations by copying the determinant which is the part of the key they are dependent on. A new relation student_info can be created

```
student\_info(student\_name,student\_id,student\_email,programme,year,address,contact)
```

leaving the students relation as

```
students(student\_id, module\_id, module\_name, module\_leader, faculty, building, room)
```

The same can be done for module_id where a new relation module_info can be created

```
module\_info(module\_id,module\_name,module\_leader,faculty,building, room)
```

leaving the students relation as students(student_id, module_id) which is the primary key.

3.5 Ex11

To achieve 2nd Normal Form, the faculty relation can be decomposed based on its functional dependencies while preserving the appropriate primary key attributes building and room. Therefore this is why faculty was moved into the location relation. The students relation was also decomposed based on its functional dependencies while preserving the appropriate primary key attributes student_info and module_info. New relations were created where the attributes were partial key dependencies for example student_info and module_info.

3.6 Ex12

Transitive dependencies occur when a non-prime attribute (an attribute not part of any candidate key) depends on another non-prime attribute through an intermediary attribute. In the faculty relations there are none because all relations faculty(building, room,capacity) and location(building, faculty) have attributes that are candidate keys therefore, there are no non-key attributes that are transitively dependent on any candidate key. For the students relations students(student_id, module_id) student_info(student_name, student_id, student_email, programme, year, address, contact) module_info(module_id, module_name, module_leader, faculty, building, room,) cohort_lecturers(student_id, module_id, lecturers) courseworks(student_id, module_id, exam_mark, coursework) there are 2 transitive dependencies: exam_mark $\rightarrow$ coursework and building,room $\rightarrow$ faculty.

3.7 Ex13

As we have already established in the faculty relation that building and room determine faculty we can say the same thing here in the module_info relation. We can decompose it into new relations module_info(module_id, module_name, module_leader, building, room) and site(building, room, faculty). In the site relation building and room are the keys and all the fields have type TEXT. In the module_info module_id remains as the key and all the fields have type TEXT. Finally, as exam_mark determines coursework in the courseworks relation we can create a new relation grade(exam_mark, coursework) leaving the courseworks relation as courseworks(student_id, module_id, exam_mark). In the courseworks relation student_id and module_id are of type TEXT but exam_mark is of type INTEGER. In the grade relation all of the fields are of type INTEGER.

4 Modelling

The existing data must now be transformed into an SQLite database based on the specified relations.

4.1 Ex14

students

Attribute	Datatype
student_id	TEXT
module_id	TEXT

student_info

Attribute	Datatype
student_name	TEXT

student_id	TEXT
student_email	TEXT
programme	TEXT
year	INTEGER
address	TEXT
contact	TEXT

cohort_lecturers

Attribute	Datatype
student_id	TEXT
module_id	TEXT
lecturers	TEXT

module_info

Attribute	Datatype
module_id	TEXT
module_name	TEXT
module_leader	TEXT
building	TEXT
room	TEXT

site

Attribute	Datatype
building	TEXT
room	TEXT
faculty	TEXT

grade

Attribute	Datatype
exam_mark	INTEGER
coursework	TEXT

courseworks

Attribute	Datatype

student_id	TEXT
module_id	TEXT
exam_mark	INTEGER

faculty

Attribute	**Datatype**
building	TEXT
room	TEXT
capacity	INTEGER

location

Attribute	**Datatype**
building	TEXT
room	TEXT
faculty	TEXT

4.2 Ex15

Linux shell

```
sqlite3 university.db
--this creates a new database
--you must now create the tables: studentscsv and facultycsv
CREATE TABLE IF NOT EXISTS studentscsv (
    student_name TEXT,
    student_id INTEGER,
    student_email TEXT,
    programme TEXT,
    year INTEGER,
    address TEXT,
    contact TEXT,
    module_id TEXT,
    module_name TEXT,
    module_leader TEXT,
    lecturer1 TEXT,
    lecturer2 TEXT,
    faculty TEXT,
    building TEXT,
    room TEXT,
    exam_mark INTEGER,
    coursework1 INTEGER,
    coursework2 INTEGER,
    coursework3 INTEGER
);

CREATE TABLE IF NOT EXISTS facultycsv (
    faculty TEXT,
    building TEXT,
    room TEXT,
    capacity INTEGER
);
--The data in the csv files are then imported to populate the tables
.mode csv
.import /home/pkp1u24/comp1314-cs/students.csv studentscsv
.import /home/pkp1u24/comp1314-cs/faculty.csv facultycsv
--then we dump the database into as ex15.sql
.output /home/pkp1u24/comp1314-cs/ex15.sql
.dump
```

4.3 Ex16

Linux shell

```
sqlite3 university.db
--this opens the database
.output ex16.sql
--we then want to output our results to ex16.sql
--we have to create the tables to satisfy our normalised relations
--This is based on the existing data in university.db
CREATE TABLE IF NOT EXISTS students (
student_id INTEGER,
module_id TEXT,
PRIMARY KEY (student_id, module_id)
);

CREATE TABLE IF NOT EXISTS student_info (
student_id INTEGER PRIMARY KEY,
student_name TEXT,
student_email TEXT,
programme TEXT,
year INTEGER,
address TEXT,
contact TEXT
);

CREATE TABLE IF NOT EXISTS cohort_lecturers (
student_id INTEGER,
module_id TEXT,
lecturer TEXT,
PRIMARY KEY (student_id, module_id, lecturer)
);

CREATE TABLE IF NOT EXISTS module_info (
module_id TEXT PRIMARY KEY,
module_name TEXT,
module_leader TEXT,
building TEXT,
room TEXT
);

CREATE TABLE IF NOT EXISTS site (
building TEXT,
room TEXT,
faculty TEXT
);

CREATE TABLE IF NOT EXISTS grade (
exam_mark INTEGER,
coursework INTEGER
);
```

Linux shell

```
CREATE TABLE IF NOT EXISTS courseworks (
student_id TEXT,
module_id TEXT,
exam_mark INTEGER,
PRIMARY KEY (student_id, module_id)
);

CREATE TABLE IF NOT EXISTS faculty (
building TEXT,
room TEXT,
capacity INTEGER,
PRIMARY KEY (building, room)
);

CREATE TABLE IF NOT EXISTS location (
building TEXT,
room TEXT,
faculty TEXT,
PRIMARY KEY (building, room)
);


INSERT INTO students (student_id, module_id)
SELECT DISTINCT student_id, module_id
FROM studentscsv;

INSERT INTO student_info (student_id, student_name, student_email,
↪  programme, year, address, contact)
SELECT DISTINCT student_id, student_name, student_email, programme,
↪  year, address, contact
FROM studentscsv;

INSERT INTO cohort_lecturers (student_id, module_id, lecturer)
SELECT DISTINCT student_id, module_id, lecturer1
FROM studentscsv
UNION
SELECT DISTINCT student_id, module_id, lecturer2
FROM studentscsv;

INSERT INTO module_info (module_id, module_name, module_leader,
↪  building, room)
SELECT DISTINCT module_id, module_name, module_leader, building, room
FROM studentscsv;

INSERT INTO site (building, room, faculty)
SELECT DISTINCT building, room, faculty
FROM studentscsv;
```

Linux shell

```
INSERT INTO grade (exam_mark, coursework)
SELECT DISTINCT exam_mark, coursework1
FROM studentscsv
UNION ALL
SELECT DISTINCT exam_mark, coursework2
FROM studentscsv
UNION ALL
SELECT DISTINCT exam_mark, coursework3
FROM studentscsv;

INSERT INTO courseworks (student_id, module_id, exam_mark)
SELECT DISTINCT student_id, module_id, exam_mark
FROM studentscsv;

INSERT INTO faculty (building, room, capacity)
SELECT DISTINCT building, room, capacity
FROM facultycsv;

INSERT INTO location (building, room, faculty)
SELECT DISTINCT building, room, faculty
FROM facultycsv;
```

This creates the necessary tables using the Create Table statement and populates them by using insert statements. It also uses Union to combine elements from a table and union all to combine all the elements into one element.

5 Querying

Now that the data is finally in a sensible format, you can now use it to answer questions! For each exercise, write an SQL query against your newly created normalised tables in your database. You must not create further tables specifically for these queries. Each SQL statement should be written in the report, as well as saved as ex¡number¿.sql (e.g. ex16.sql). Write an SQLite statement to find:

5.1 Ex17

Linux shell

```
sqlite3 university.db
--this opens the database
.output /home/pkp1u24/comp1314-cs/ex17.sql
--redirects the output to ex17.sql
SELECT building, SUM(capacity) AS total_capacity
FROM faculty
GROUP BY building;
--This query then calculates the total capacity of each building in
↪  the university
```

5.2 Ex18

Linux shell

```
sqlite3 university.db
--this opens the database
.output /home/pkp1u24/comp1314-cs/ex18.sql
--redirects the output to ex18.sql
SELECT student.student_id, student.student_name, AVG(cw.exam_mark) AS
↪  average_exam_mark
FROM student_info student
JOIN courseworks cw ON student.student_id = cw.student_id
WHERE student.programme = 'Computer Science' AND student.year = 1
GROUP BY student.student_id, student.student_name
ORDER BY average_exam_mark DESC;
--This query then gets the average exam mark for each Year 1 Computer
↪  Science student by grouping them with their id and name and sorts
↪  it in descending order
```

5.3 Ex19

Linux shell

```
sqlite3 university.db
.output /home/pkp1u24/comp1314-cs/ex19.sql
-- Calculate average marks for each module and faculty
SELECT mi.module_id, mi.module_leader, s.faculty, AVG(g.exam_mark +
    g.coursework) AS average_mark
FROM module_info mi
JOIN site s ON mi.building = s.building AND mi.room = s.room
JOIN courseworks cw ON mi.module_id = cw.module_id
JOIN grade g ON cw.exam_mark = g.exam_mark
GROUP BY mi.module_id, mi.module_leader, s.faculty
HAVING AVG(g.exam_mark + g.coursework) = (
SELECT MAX(module_avg)
FROM (
SELECT AVG(g_inner.exam_mark + g_inner.coursework) AS module_avg,
    s_inner.faculty AS faculty
FROM module_info mi_inner
JOIN site s_inner ON mi_inner.building = s_inner.building AND
    mi_inner.room = s_inner.room
JOIN courseworks cw_inner ON mi_inner.module_id = cw_inner.module_id
JOIN grade g_inner ON cw_inner.exam_mark = g_inner.exam_mark
WHERE s_inner.faculty = s.faculty
GROUP BY mi_inner.module_id
)
)
ORDER BY average_mark DESC;

```

This code calculates the average marks for each module within each faculty. Finds the maximum average mark within each faculty. Joins the two queries to find the modules with the highest average marks for their respective faculty. Orders the results by the highest average mark.

```
The output from ex19.sql was :
ANIM3403|Michelle Mueller|Faculty of Art|50.0423796259451
ELEC4404|Margaret Sanders|Faculty of Technology|49.4737901381392
ENG1402|Bruce Schneider|Faculty of Languages|47.4435638152501
```

5.4 Ex20

Linux shell

```
sqlite3 university.db
--this opens the database
.output /home/pkp1u24/comp1314-cs/ex20.sql
SELECT mi.module_id, s.room, s.building, f.capacity,
COUNT(cw.student_id) AS student_count
FROM module_info mi
JOIN courseworks cw ON mi.module_id = cw.module_id
JOIN site s ON mi.building = s.building AND mi.room = s.room
JOIN faculty f ON s.building = f.building AND s.room = f.room
GROUP BY mi.module_id, s.room, s.building, f.capacity
HAVING COUNT(cw.student_id) > f.capacity
ORDER BY student_count DESC;
```

This code finds the tables Where student count exceeds room capacity by joining module_info, courseworks, site, and faculty. It uses COUNT(cw.student_id) to calculate the number of students for each module. It then compares the student count to room capacity and groups the data by module, room, building, and capacity. The HAVING statement ensures only rows where the student count exceeds the room's capacity are included. The ORDER BY statement sorts the results by student_count in descending order