**CSCI 620/Section 03/Mior, Introduction to Big Data, Spring 2215 Assignment 3 – Normalization**

**Submitted by – Prakhar Gupta(pg9349)**

**Ans1**

Refer to (joining.sql) the code create a table called movies.

**Ans2**

Refer to code q2naive.py(this code uses ‘uniquekey’ columns as well)

Here we are using the group by function of sql and executing the query using python. The following query is executed =

**“SELECT {i}, COUNT(DISTINCT {j}) FROM {table\_name} GROUP BY {i} HAVING COUNT(DISTINCT {j}) > 1”**

So basically what we are doing is that we are checking if i-🡪j functional dependency hold true ‘I’ can have more than one attributes. We are ignoring the trivial case where i🡪i because we know it holds true. I the above query executes and gives atleast a row as output we know the FD does not hold because for t1[i] = t2[i], t[j] is not equal to t2[j].

I ran the code with 20 combinations to estimate the time to check all dependencies. More combinations can be checked by changing the pairs variable to the desired value. The approach is naïve because we are checking across all the pairs of attributes some of which can be inferred.

It took 3.94 minutes to check 20 dependencies .To check for 211 pairs which would take around 8069 minutes.

**Ans 3**

Refer to code q3pruning.py (this code does not use ‘uniquekey’ column as we know that attribute is created by us to uniquely identify a tuple in the movies relation)

The code uses partitions, lattice, refinement and pruning to identify functional dependencies.

* Partitions are achieved using pandas group by function
* Lattice is created in a way of unidirectional graph adjacency dict where the key are single attribute connected with two attribute which can be pruned in case the single attribute satisfies a FD with some other attribute.
* Refinement- For refinement of partitions two techniques were used
  + Technique 1 – this uses the same algorithm which was discussed in the class of identifying subsets of two partitions A and B such that all equivalence classes of A are some subsets of B. Hence A refines B hence A🡪B holds. The function refinement () uses this technique. Code can be uncommented if the prune function to run the refinement using this technique. This technique does take considerable time to run. Approximate time is 4hours
  + Technique 2- While finding other ways to refine partitions I decided to exploit the properties of functional dependencies

Specially Theorem 2 on page4 of the following paper-

<https://iads.site/wp-content/uploads/papers/2003/859163.pdf>.

So to check for A-> B we can create partitions using our pandas groupby for A and A U B and check whether the number of equivalence classes is same for both. This lead to speeding up of refinement process. Currently the code uses this property to refine.

* Once the Functional dependency is identified the node and the nodes which are linked to it in the lattice are pruned as well.

For example-

If movieid----->type

('movieid', 'genreid')----->type

'movieid', 'genreid' together also determines type by inference.

The code run is about 16 minutes and identifies and write all dependencies to allfd.txt and non-inferable FD(Canonical Cover) to noninferable\_fd.txt.

**Ans4**

If the if we do not restrict that “the same actor only plays a single character in a given movie” as in question 1. The following functional dependencies would not hold true

('movieid', 'memberid')----->character

Because of same movie and member there can be an actor who was in a movie and played multiple characters.

Hence there would be no way to determining character from any the other attributes.

**Ans5**

We have not assumes the uniquekey column which we added to identify the rows in our relation for as it was a dummy column introduced by us.

**Canonical Cover-**

Canonical cover is the set of functional dependencies which are the base dependencies such that they cannot be inferred from other dependencies.

Our code q3pruning.py identifies and writes these dependencies in noninferable.txt file which are as follows-

movieid----->type  
movieid----->startyear  
movieid----->runtime  
movieid----->avgrating  
genre----->genreid  
genreid----->genre  
memberid----->birthyear  
('movieid', 'memberid')----->character

There are 8 such dependencies

**Candidate keys-**

To find candidate keys we need to identify core and exterior-

* Core- Core contains following attributes-
  + Those attributes which never appear in functional dependencies (None)
  + Those attributes that are only on the left hand side and never on right hand side(**‘movieid’** and **‘memberid’)**
* Exterior- Exterior consist of
  + Those attributes which appear on the right hand side of some FD-

**(‘type’,’startyear’,’runtime’,’avgrating’,’genreid’,’genre’,’birthyear’,**

**‘character’**)

To find the candidate keys we need to check for closure of core

Closure of core- **(‘type’,’startyear’,’runtime’,’avgrating’,’birthyear’,‘character’**)

This does not contain all attributes hence our core is not closed. There is one more candidate key.

We need to iteratively add one item from Exterior and check for closure of core-

2 pairs satisfy the closure-

(**‘movieid’** , **‘memberid’, ’genre’)**

(**‘movieid’** , **‘memberid’, ’genreid’)**

Hence the above 2 are the candidate keys.

**Final decomposition (3NF)**

We can decompose our relations using the set of attributes discovered earlier-

Following decomposition needs to happen .Primary key is bold and has PK following its name

1. Relation 1 - (**‘movieid’ (PK)** , **‘memberid’(PK), ’genreid’ (PK))**
2. Relation 2 – (**‘movieid’ (PK),** ‘type’, ’startyear’, ’runtime’, ’avgrating’)
3. Relation 3- (**’genreid’(PK)**, ’genre’)
4. Relation 4- (**‘memberid’ (PK),** birthyear)
5. Relation 5 – (**‘movieid’ (PK)** , **‘memberid’(PK),** ’character’)

References-

(Discovery of Functional and Approximate Functional Dependencies in Relational Databases RONALD S. KING† rking@mail.uttyl.edu Computer Science Department, The University of Texas at Tyler, Tyler, Texas 75799 JAMES J. LEGENDRE Marathon Oil, Houston, Texas)