1 Question 1

1.1 Premise

Consider the following schema and data:

```
course (c_id, dept_id, dept, evaluations, inst, office, sect, time_slot)
```

c_id	dept_id	dept	evaluations	inst	office	sect	time_slot
61	1	CS	HW, Midterm, Final	Eddie Kohler	345	A	8am-10am
61	1	CS	HW, Midterm, Final	Eddie Kohler	345	B	1pm-3pm
165	2	MATH	Project, Final	Stratos Idreos	346	A	10am-11am
165	2	MATH	Project, Final	Stratos Idreos	346	B	7pm-8pm
265	2	MATH	HW	Andy Pavlo	346	B	1pm-2pm
455	3	PHYS	Midterm, Final	Nesime Tatbul	347	B	6pm-7pm

Throughout this problem, we will take this schema through four different normal forms to gain a sense of what benefits each one brings. Note that it will be useful to you to create populated tables using the data above for each schema you create, but it is not required. Lastly, for each conversion, be sure to document and explain your steps; not too much, just enough for us to understand your thought process.

1.2 1.1 First Normal Form

A schema is in the first normal form when: - The attributes are atomic, and - The attribute names are unique.

We already have the latter, so we'll focus on the former. Once you've identified the non-atomic attributes, you can either split them into multiple attributes or into a separate relation entirely.

Convert this schema into the first normal form. You should have three relations by the end of your conversion.

1.3 1.2 Second Normal Form

A schema is in the second normal form when: - The schema is in first normal form (done!) - There are no partial dependencies.

In other words, say there's a candidate key A . Then, every other attribute in the relation should depend on all of A , not a subset of it. If there is an attribute such that only a part of A functionally determines it, there is a partial dependency. Once you've identified a partial dependency, you can normalize it by splitting the relation in two: one for attributes that are partially dependent, and one for those that aren't.

Convert this schema into the second normal form. You should have four relations by the end of your conversion.

1.4 1.3 Third Normal Form

A schema is in the third normal form when: - The schema is in second normal form (done!) - There are no transitive dependencies.

A transitive dependency is a dependency between two sets of attributes where both sets are still dependent on the primary key. Formally, if $A \rightarrow B$ and $B \rightarrow C$ and NOT $B \rightarrow A$, then $A \rightarrow C$ is a transitive dependency. Once you've identified a transitive dependency, you can separate the dependent attributes out into another relation.

Convert this schema into the third normal form. You should have five relations by the end of your conversion.

(Hint: in these instances of our relations, and conceivably for all potential instances, instructors only teach one course).

1.5 1.4 BCNF (Third-and-a-half Normal Form)

A schema is in BCNF when: - The schema is in third normal form (done!) - For all depencies, either: - $a \rightarrow b$ is a trivial dependency ($a \subseteq b$), or - a is a superkey.

Convert this schema into BCNF. You should have six relations by the end of your conversion.

1.6 Solution

Note: there was communication that `c_id, sect -> inst` *instead of* `c_id -> inst`*; if you wrote your solutions with the either in mind and were consistent, you should receive full credit.*

1.6.1 1.1 First Normal Form

First, let's lay out all of our dependencies. This will make it very clear why we are doing what we are doing.

```
c_id -> dept_id, evaluations, inst
c_id, sect -> time_slot
inst -> office, c_id
dept_id -> dept
```

We want to ensure atomicity. So, we split `time_slot` into two attributes `time_start` and `time_end`, and pull evaluations out into an `evaluation` table, as well as a `course_evaluation` table to join evaluations back to courses.

```
course (**c_id**, **sect**, dept_id, dept, inst, office, time_start, time_end)
evaluation (**e_id**, e_name)
course_evaluation (**c_id**, **e_id**)
```

Here is the resulting data for this decomposition:

c_id	sect	dept_id	dept	inst	office	time_start	time_end
61	A	1	CS	Eddie Kohler	345	8am	10am
61	B	1	CS	Eddie Kohler	345	1pm	3pm
165	A	2	MATH	Stratos Idreos	346	10am	11am
165	B	2	MATH	Stratos Idreos	346	7pm	8pm
265	B	2	MATH	Andy Pavlo	346	1pm	2pm
455	B	3	PHYS	Nesime Tatbul	347	6pm	7pm

e_id	e_name
1	HW
2	Project

e_id	e_name
3	Midterm
4	Final

c_id	e_id
61	1
61	3
61	4
165	2
165	4
265	1
455	3
455	4

1.6.2 1.2 Second Normal Form

Let's lay out our dependencies from last time, since they have changed slightly (ignoring the two new tables):

```
c_id -> dept_id, inst
c_id, sect -> time_start, time_end
inst -> office, c_id
dept_id -> dept
```

Our candidate key is `c_id, sect`. Now, notice that we have a partial dependency in the first two. Thus, we should split `c_id, sect, time_start, time_end` into its own table:

```
course (**c_id**, dept_id, dept, inst, office)
section (**c_id**, **sect**, time_start, time_end)
evaluation (**e_id**, e_name)
course_evaluation (**c_id**, **e_id**)
```

Here is the resulting data for this decomposition:

c_id	dept_id	dept	inst	office
61	1	CS	Eddie Kohler	345
165	2	MATH	Stratos Idreos	346
265	2	MATH	Andy Pavlo	346
455	3	PHYS	Nesime Tatbul	347

c_id	sect	time_start	time_end
61	A	8am	10am
61	B	1pm	3pm
165	A	10am	11am
165	B	7pm	8pm
265	B	1pm	2pm
455	B	6pm	7pm

e_id	e_name
1	HW
2	Project
3	Midterm
4	Final

c_id	e_id
61	1
61	3
61	4
165	2
165	4
265	1
455	3
455	4

1.6.3 1.3 Third Normal Form

Let's lay out our dependencies from last time, even though they are unchanged:

```
c_id -> dept_id, inst
c_id, sect -> time_start, time_end
inst -> office, c_id
dept_id -> dept
```

Notice that we have a transitive dependency in `c_id -> dept_id` and `dept_id -> dept`. So, we pull `dept_id, dept` into its own table.

```
course (**c_id**, dept_id, inst, office)
section (**c_id**, **sect**, time_start, time_end)
evaluation (**e_id**, e_name)
course_evaluation (**c_id**, **e_id**)
department(**dept_id**, dept)
```

Here is the resulting data for this decomposition:

c_id	dept_id	inst	office
61	1	Eddie Kohler	345
165	2	Stratos Idreos	346
265	2	Andy Pavlo	346
455	3	Nesime Tatbul	347

c_id	sect	time_start	time_end
61	A	8am	10am
61	B	1pm	3pm
165	A	10am	11am
165	B	7pm	8pm
265	B	1pm	2pm
455	B	6pm	7pm

e_id	e_name
1	HW
2	Project
3	Midterm
4	Final

c_id	e_id
61	1
61	3
61	4
165	2
165	4
265	1
455	3
455	4

dept_id	dept
1	CS
2	MATH
3	PHYS

1.6.4 1.4 BCNF (Third-and-a-half Normal Form)

Note: this particular question is broken, and relies on the bold assumption that `inst` *cannot be a superkey, even though the functional dependencies say it can be. As a result, this question will not be graded. Please take it as an illustration of BCNF.*

Let's lay out our dependencies from last time, even though they are unchanged:

```
c_id -> dept_id, inst
c_id, sect -> time_start, time_end
inst -> office, c_id
dept_id -> dept
```

Notice that we have `c_id -> dept_id, i_id, office` (the last one by Armstrong's), meaning the `c_id` is a superkey. However, since `inst` cannot be a superkey, it violates the requirements for BCNF. So, we split `inst, office` into its own table, and create an `i_id` to allow joining back to the original table.

```
course (**c_id**, dept_id, i_id)
section (**c_id**, **sect**, time_start, time_end)
evaluation (**e_id**, e_name)
course_evaluation (**c_id**, **e_id**)
department (**dept_id**, dept)
instructor (**i_id**, inst, office)
```

Here is the resulting data for this decomposition:

c_id	dept_id	i_id
61	1	1
165	2	2

c_id	dept_id	i_id
265	2	3
455	3	4

c_id	sect	time_start	time_end
61	A	8am	10am
61	B	1pm	3pm
165	A	10am	11am
165	B	7pm	8pm
265	B	1pm	2pm
455	B	6pm	7pm

e_id	e_name
1	HW
2	Project
3	Midterm
4	Final

c_id	e_id
61	1
61	3
61	4
165	2
165	4
265	1
455	3
455	4

dept_id	dept
1	CS
2	MATH
3	PHYS

i_id	inst	office
1	Eddie Kohler	345
2	Stratos Idreos	346
3	Andy Pavlo	346
4	Nesime Tatbul	347

2 Question 2

Consider the following schema S and constraints F :

```
users (user_id, name, dept_id, dept_name)
books (book_id, isbn, title, author, publisher)

user_id -> name
user_id -> dept_id
user_id -> dept_name
dept_id -> dept_name
book_id -> isbn
isbn -> title
isbn -> publisher
isbn -> author
```

Complete the following exercises:

1. What are the additional functional dependencies that can be generated by Armstrong's Transivity Rule?
2. Compute the canonical cover of this set of contraints, F_C . (Hint: use the algorithm described in class).
3. Normalize the given schema to BCNF. Was your normalization lossy? Was your normalization dependency preserving?

2.1 Solution

1. What are the additional functional dependencies that can be generated by Armstrong's Transivity Rule?

```
book_id -> title
book_id -> publisher
book_id -> author
```

2. Compute the canonical cover of this set of contraints, F_C . (Hint: use the algorithm described in class).

```
user_id -> name, dept_id
dept_id -> dept_name
book_id -> isbn
isbn -> title, publisher, author
```

3. Normalize the given schema to BCNF. Was your normalization lossy? Was your normalization dependency preserving?

```
users (user_id, name, dept_id)
department (dept_id, dept_name)
books (book_id, isbn)
book_data (isbn, title, author, publisher)
```

Lossy: No; BCNF is always lossless. Dependency preserving: Yes; all dependencies in the canonical cover of the original relation set are in the resulting relation set.

3 Question 3

Use Armstrong's axioms (reflexivity, augmentation, and transitivity ONLY) to prove the following rules:

1. If $a \rightarrow b$, then $a \rightarrow ab$ (extensivity).
2. If $a \rightarrow b$ and $a \rightarrow c$, then $a \rightarrow bc$ (union rule).
3. If $a \rightarrow b$ and $bc \rightarrow d$, then $ac \rightarrow d$ (pseudotransitivity).

3.1 Solution

1. If $a \rightarrow b$, then $a \rightarrow ab$ (extensivity).

aa -> ab (augmentation) a -> ab (rewrite the set)

2. If $a \rightarrow b$ and $a \rightarrow c$, then $a \rightarrow bc$ (union rule).

a -> b (given) a -> c (given) a -> ab (augmentation) ab -> bc (augmentation) a -> bc (transitivity)

3. If $a \rightarrow b$ and $bc \rightarrow d$, then $ac \rightarrow d$ (pseudotransitivity).

ac -> bc (augmentation) ac -> d (transitivity)

4 Question 4

You've learned how to create table-level checks during table creation using key words such as `NOT NULL`, `UNIQUE`, or `CHECK`. Now, we'll learn how to create arbitrary assertions over multiple tables that your database should check.

Note that global assertions can be expensive for your database system to maintain; they need to be checked every time a change is made. Moreover, not every SQL implementation has assertions, nor do they all do it the same way. Don't worry about performance or syntax-correctness too much for this homework, but keep it in mind for future database design decisions.

An assertion essentially takes a subquery and checks whether it evaluates to some result. The following makes sure that there are exactly 0 users:

```
CREATE ASSERTION assertion_name CHECK (
    0 = (SELECT COUNT(*) FROM users)
)
```

Consider the following schema:

```
users (user_id, name, dept_id, dept_name)
books (book_id, isbn, title, author, publisher)
reads (user_id, book_id)
```

Now, create assertions to check the following:

1. Make sure that there is never a book by "JK Rowling" that is also published by "Penguin" (Hint: use NOT IN).
2. Make sure that the number of users in a department never exceeds 100.
3. Make sure that nobody is reading more than 5 books.

4.1 Solution

1. Make sure that there is never a book by "JK Rowling" that is also published by "Penguin" (Hint: use NOT IN).

```
CREATE ASSERTION no_jkr_penguin CHECK (
    "JK␣Rowling" NOT IN (SELECT author FROM books WHERE publisher="Penguin")
)
```

2. Make sure that the number of users in a department never exceeds 100.

```
CREATE ASSERTION less_than_a_hundred CHECK (
    100 >= (SELECT COUNT(*) FROM users GROUP BY dept_id ORDER BY COUNT(*) DESC LIMIT 1)
)
```

3. Make sure that nobody is reading more than 5 books.

```
CREATE ASSERTION dont_read_too_much CHECK (
    5 >= (SELECT COUNT(*) FROM reads GROUP BY user_id ORDER BY COUNT(*) DESC LIMIT 1)
)
```

5 Question 5

As of now, we've mostly been focusing on OLTP workloads. OLTP (OnLine Transactional Processing) workloads are commonly used for many systems because they are very good at retrieving data. Unfortunately, OLTP databases are a poor choice for asking questions about the data, like calculating statistics over the data. In that case, we use OLAP (OnLine Analytical Processing). You don't need a perfect understanding between the two for this question. You only need to understand that we sometimes (not always!) structure data differently when the situation calls for it. We encourage you to read more about the differences here.

One schema design used in OLAP workloads is the Star Schema. In this style, we represent facts as a centralized table (for example, in this question: when a worker bee collects honey, this is recorded in the fact_work table). Dimensions about these facts are represented in related tables. Consider the following star schema for Barry's bees.

ER Diagram:

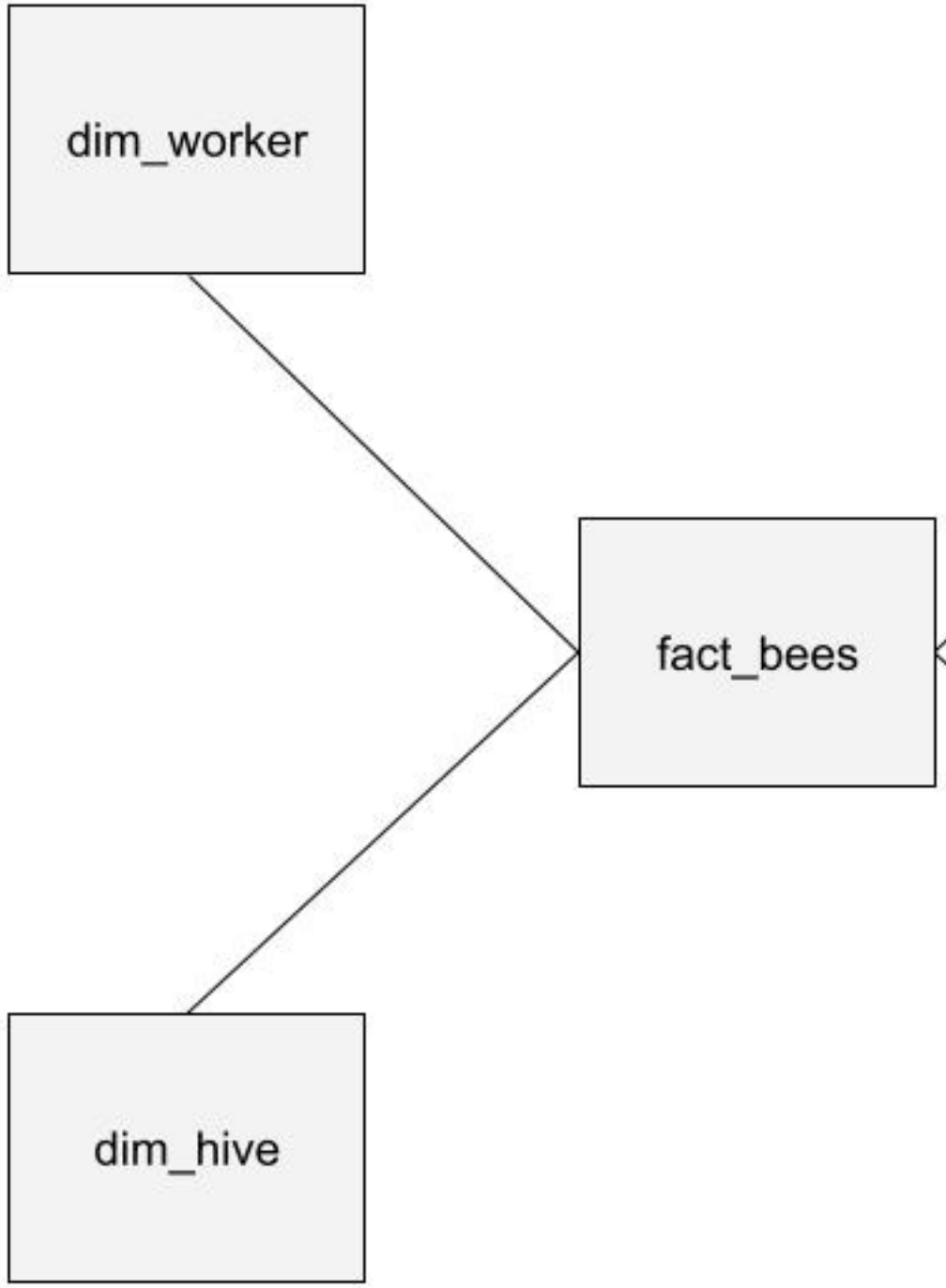
Star Schema:

```
fact_work(worker_id, queen_id, honey_id, hive_id, quantity_mililiters, hours_worked)
dim_worker(worker_id, worker_name)
dim_queen(queen_id, queen_name)
dim_hive(hive_id, location)
dim_honey(honey_id, type)
```

Normalized Schema:

```
workers(worker_id, worker_name, hive_id, queen_id)
queens(queen_id, queen_name)
hives(hive_id, location)
honey(honey_id, type)
work(work_id, worker_id, honey_id, quantity_mililiters, hours_worked)
```

1. Does the star schema meet 1NF? Why or why not?


dim_worker
fact_bees
dim_hive

2. Does the star schema meet 2NF? Why or why not?
3. Does the star schema meet 3NF? Why or why not?
4. Does the star schema meet BCNF? Why or why not?
5. Suppose we want to know how many workers live in each hive. Can we write an SQL query to find this for the star schema? How about the normalized schema?
6. Suppose we want to know how many quantity_milliliters of honey were produced for each queen, along with the queen bee's name. Write a SQL query to find this for both the star schema and the normalized schema.
7. In what ways is this star schema "better" than a normalized schema?

5.1 Solution

Our functional dependencies are:

```
worker_id -> queen_id, hive_id, worker_name
queen_id -> hive_id, queen_name
hive_id -> queen_id, location
honey_id -> type
```

Our star schema primary key is `(worker_id, honey_id, quantity_milliliters, hours_worked)`. We don't need `queen_id` or `hive_id` since we can functionally determine them.

1. Yes; attributes are atomic and attribute names are unique.
2. No; all of our dependencies are partial dependencies, since no dependency (except for the trivial one) stems from the entire column.
3. No; it isn't in 2NF and there are transitive dependencies: worker_id -> queen_id, queen_id -> hive_id, but queen_id -/-> worker_id.
4. No; the scheme is not in 3NF, and there are clearly non-trivial dependencies where the left side is not a superkey.
5. Star: We might not be able to answer this question with a star schema. For example, if none of the bees for a hive did any work, joining the hive dimension table with the worker dimension table produces no results for that hive.

Normalized: SELECT COUNT(worker_id) FROM workers GROUP BY hive_id;

6. Star: SELECT SUM(fact_work.quantitiy_milliliters), dim_queen.queen_name FROM fact_work JOIN dim_queen ON fact_work.queen_id = dim_queen.queen_id GROUP BY fact_work.queen_id;

Normalized: SELECT SUM(work.quantity_milliliters), queens.queen_name FROM work JOIN workers ON work.worker_id = workers.worker_id JOIN queens ON workers.queen_id = queens.queen_id GROUP BY workers.queen_id;

7. Simpler joins to retrieve data, since dimension tables are linked through a central fact tables. Less joins mean faster queries and faster analysis.

Feedback

As this is a new course, we appreciate any feedback you have for us! If you enjoyed this assignment, hated this assignment, or have other thoughts to share, please do so here!