Statistics

- The average points reached (out of 42) were 32 (77%), the maximum was 41.5 (99%), the minimum 14.5 (35%).
- 17 Students (out of 70, i.e. 24%) had more than 90% of the points, 31 students (44%) had more than 80% of the points, and 4 students (6%) had less than 50% of the points.

Exercise 1 (Translation: ER to Relational Model) 10 Points

The result of the translation of the cable channel database into the relational model is:

Channel(CNo, Name)
Package(PID, Price)
in(CNo → Channel, PID → Package)
Movie(Title, Year)
Broadcast(CNo → Channel, Day, From_Time, To_Time, Title → Movie)

You can use plural forms for the table names (e.g. Channels), and also a better name for the relationship table “in” might be in order (e.g. contains or Package_Contents). Another name for the foreign key “Title” in “Broadcast” is also possible (e.g. “of”). But apart from such minor variations, this is the only correct solution.

28 students (40%) got the full points, 18 students (26%) lost 0.5 or 1 point, 9 students (13%) lost 1.5 or 2 points, and 14 students (20%) lost more than two points. The maximum lost was 5 points. Probably, this exercise should not have been worth 10 points. Something around 6–8 points would have been better. But the second exercise should have been worth more than 5 points, and since nobody skipped one of these two exercises, nobody was treated unfairly.

14 students got the key for “Broadcast” wrong, this does not include making “Title” part of the key (that error was made by 11 students). In addition, 9 students dashed-underlined the part of the key which corresponds to the partial key defined in the “Broadcast” entity. But the relational model has no notion of partial keys. So the key in the
weak entity was a major source of errors. In total, 30 students (43%) made some error related to the key of “Broadcast”.

7 students didn’t use an extra table for the translation of the many-to-many relationship “in”, 2 students got the key wrong, 3 students forgot to translate the relationship, and 3 students translated it two times (once by the extra table, and once by adding a foreign key to the table “Channel”). So in total 15 students (21%) had problems with the many-to-many relationship.

5 students translated the one-to-many relationship “of” in the wrong direction (by adding the key of “Broadcast” to “Movie”), and 6 students forgot to translate it.

4 students declared attributes as optional, but in this example, null values are not allowed in any attribute.

Some students mentioned additional constraints, which did make sense, but were not required in order to ensure the equivalence of both schemas. Such constraints would have to be added to the ER-schema first.

Exercise 2 (SQL CREATE TABLE) 5 Points

The CREATE TABLE command for “COLS” could look as follows:

```
CREATE TABLE COLS(
    TABLE_NAME    VARCHAR(128) NOT NULL REFERENCES CAT,
    COLUMN_NAME   VARCHAR(128) NOT NULL,
    NULLABLE      CHAR(1) NOT NULL
    CONSTRAINT NULLABLE_YN
        CHECK(NULLABLE IN ('Y','N')),
    COLUMN_ID     NUMERIC(4) NOT NULL
    CHECK(COLUMN_ID > 0),
    PRIMARY KEY(TABLE_NAME, COLUMN_ID),
    UNIQUE(TABLE_NAME, COLUMN_NAME))
```

Possible variations include:

- The constraint enforcing that “NULLABLE” has only the two possible values “Y” and “N” can also be written as \( \text{CHECK(NULLABLE = 'Y' OR NULLABLE = 'N')} \).
- It is possible to put a comma in front of the check-constraints to make them table constraints instead of column constraints (which has no semantic difference).
- The foreign key constraint can be written as table constraint in the form:

\[
\text{FOREIGN KEY(TABLE_NAME) REFERENCES CAT.}
\]
Note that it is not allowed to use the "FOREIGN KEY" keywords if you formulate the foreign key as a column constraint. Also, the parentheses around TABLE_NAME are required, even though the foreign key consists only of single column in this case.

- Some students used "VARCHAR(1)" for "NULLABLE". Given the check-constraint, this makes no semantic difference. But in some systems (not in Oracle) more storage space might be required since the length must be stored in addition.

- In Oracle, you can write "NUMBER(4)" instead of "NUMERIC(4)". However, this is not portable to other systems. "DECIMAL(4)" instead of "NUMERIC(4)" is ok: It means basically the same. Some systems might allow larger numbers than 9999 for DECIMAL(4), but in this case we anyway would need to restrict the COLUMN_ID to 1000 (the maximal column number in Oracle).

15 students (21%) got the full points, 29 students (41%) lost 0.5 or 1 point, 19 students (27%) lost 1.5 or 2 points, and 6 students lost 2.5 or more points. The maximum lost was 4 points. This exercise should have been worth more than 5 points (e.g. 7 points). But since Exercise 1 was worth too many points, and nobody skipped one of the two exercises, it basically cancels out.

The most common error was to forget the primary key TABLE_NAME, COLUMN_ID. 28 students (40%) did this. Usually the alternative key TABLE_NAME, COLUMN_NAME was then declared as primary key. It might have been slightly unfair that the primary key was not explicitly mentioned in the exercise, but it was clearly shown in the table. Also, since an "alternative key" was mentioned in the exercise, it would have been natural to look for another key. After all, basically every table should have a primary key declared for it.

The alternative key was forgotten in 12 cases (17%), a wrong primary or alternative key (like only TABLE_NAME) was chosen in 10 cases (14%). It makes no sense to declare e.g. TABLE_NAME as primary key, and the combination of TABLE_NAME and COLUMN_NAME as secondary key. In this case, the secondary key is non-minimal and therefore superfluous (implied). Some authors would say that in this case “TABLE_NAME, COLUMN_NAME” is no key at all because it is not minimal.

5 students used the wrong syntax for the alternative key, e.g. there is no ALTERNATIVE KEY keyword. Also, it is wrong to write UNIQUE as a column constraint after TABLE_NAME and COLUMN_NAME. This would declare both columns as a key by themselves (which is too strong).

One or both of the check-constraints were forgotten in 6 cases, a wrong syntax for check constraints was used in 6 cases. E.g. all of the following are wrong:

- NULLABLE . . . CHECK(’Y’, ’N’)
- NULLABLE . . . CHECK(NULLABLE = (’Y’, ’N’))
- NULLABLE . . . CHECK IN (’Y’, ’N’)
- NULLABLE . . . = Y,N
Two students wrote `NUMERIC(1000)`, because the exercise said that column numbers until 1000 are needed. However, the parameter is the number of digits, not the maximal value. One student wrote `INT(4)`; but `INT` does not allow a parameter. Three students declared `COLUMN_ID` as `VARCHAR(4)`, but this allows illegal values (non-numbers) in this column, and even if you use only numbers, you get a wrong order (e.g. `'2' > '10'`).

`NOT NULL` constraints were forgotten only in two cases.
The results of this exercise were quite disappointing: Only 3 students (out of 70) got the full points, 5 students lost 0.5 or 1 point, 10 students lost 1.5 or 2 points, 5 students lost 2.5 or 3 points, 21 students lost between 3.5 and 6 points, 18 students lost between 6.5 and 9 points, 7 students lost more than 9 points. The maximum lost was 14.5 points. Many students said that they were not prepared to do the step to the meta-data, and that a standard bank, airline, or course database would have been simpler. However, we had a homework with queries to the data dictionary, and I did mention that such exercises might be possible. Also, only a few errors seemed to show that the student didn’t understand the exercise, and I corrected these errors quite generously. Other errors could have occurred independently from this application. However, it might be true that the data dictionary application was distracting or lead to some kind of “panic” which then caused errors you would not have done under normal circumstances.

Exercise 3 (SQL) 18 Points

a) Print the names of all indexes which include the two columns “FIRST_NAME” and “LAST_NAME” of the table “STUDENTS”.

- We need to find indexes which have at least two entries in USER_IND_COLUMNS, one for the column “FIRST_NAME” and one for the column “LAST_NAME”. So it is natural to use two tuple variables:

```sql
SELECT X.INDEX_NAME
FROM USER_IND_COLUMNS X, USER_IND_COLUMNS Y
WHERE X.INDEX_NAME = Y.INDEX_NAME
AND X.TABLE_NAME = 'STUDENTS'
AND X.COLUMN_NAME = 'FIRST_NAME'
AND Y.COLUMN_NAME = 'LAST_NAME'
```

It is possible to add the condition “Y.TABLE_NAME = 'STUDENTS'”. However, since an index can be defined only on one table, this is not required (the condition will automatically be satisfied).

- Some students, who did not like explicit tuple variables, used a subquery instead:

```sql
SELECT INDEX_NAME
FROM USER_IND_COLUMNS
WHERE TABLE_NAME = 'STUDENTS'
AND COLUMN_NAME = 'FIRST_NAME'
AND INDEX_NAME IN (SELECT INDEX_NAME
                     FROM USER_IND_COLUMNS
                     WHERE COLUMN_NAME = 'LAST_NAME')
```

- If you want a symmetric solution, the following query would be possible. How-
ever, it does an unnecessary join and runs a bit slower than the other two:

```sql
SELECT INDEX_NAME
FROM IND
WHERE TABLE_NAME = 'STUDENTS'
AND INDEX_NAME IN (SELECT INDEX_NAME
                    FROM USER_IND_COLUMNS
                    WHERE COLUMN_NAME = 'FIRST_NAME')
AND INDEX_NAME IN (SELECT INDEX_NAME
                    FROM USER_IND_COLUMNS
                    WHERE COLUMN_NAME = 'LAST_NAME')
```

- You can also use the intersection operator to get a symmetric solution without tuple variables. However, the intersection operator is not used very often, and the optimizer might be not very good in developing an efficient query evaluation plan for it. But in this case, there are only very few indexes on the given column, so the efficiency of the intersection might be not important.

```sql
SELECT INDEX_NAME
FROM USER_IND_COLUMNS
WHERE TABLE_NAME = 'STUDENTS'
AND COLUMN_NAME = 'FIRST_NAME'
INTERSECT
SELECT INDEX_NAME
FROM USER_IND_COLUMNS
WHERE TABLE_NAME = 'STUDENTS'
AND COLUMN_NAME = 'LAST_NAME'
```

Only 12 students (17%) got this fully correct. Now let us look at some errors.

- The most common error for this exercise was to forget “something”. 12 students (17%) forgot the condition `X.TABLE_NAME = 'STUDENTS'`. 11 students (16%) forgot the join condition on the index name. Another 6 students (9%) joined the tables on the table name instead of the index name. This is not sufficient since there might be two different indexes on the two columns, but we are looking only for a single index on the column combination:

```sql
SELECT X.INDEX_NAME     Wrong!
FROM USER_IND_COLUMNS X, USER_IND_COLUMNS Y
WHERE X.TABLE_NAME = Y.TABLE_NAME
AND X.TABLE_NAME = 'STUDENTS'
AND X.COLUMN_NAME = 'FIRST_NAME'
AND Y.COLUMN_NAME = 'LAST_NAME'
```
This shows again that a database design is difficult if there is more than one way to join two tables. This situation cannot always be avoided, but in this case, USER_IND_COLUMNS violates BCNF (see below). It should not include TABLE_NAME, and then there would be no problem.

- 10 students (14%) used an attribute without a tuple variable although there were two tuple variables with this attribute. So the attribute reference was ambiguous.
- 10 students (14%) had some kind of surely false condition in this exercise. A quite obvious example of this kind of error is this query:

```sql
SELECT INDEX_NAME  
FROM   USER_IND_COLUMNS 
AND    TABLE_NAME = 'STUDENTS' 
AND    COLUMN_NAME = 'FIRST_NAME' 
AND    COLUMN_NAME = 'LAST_NAME' 
```

It is impossible that the attribute COLUMN_NAME has two distinct values at the same time.

The following WHERE-condition is also always wrong (contradictory):

```sql
SELECT U.INDEX_NAME  
FROM   COLS C1, COLS C2, USER_IND_COLUMNS U 
WHERE  U.COLUMN_NAME = C1.COLUMN_NAME 
AND    U.COLUMN_NAME = C2.COLUMN_NAME 
AND    C1.COLUMN_NAME = 'FIRST_NAME' 
AND    C2.COLUMN_NAME = 'LAST_NAME' 
```

The first two conditions imply that C1.COLUMN_NAME = C2.COLUMN_NAME. But on the other hand, C1.COLUMN_NAME <> C2.COLUMN_NAME follows from the last two conditions. This is a contradiction.

Another WHERE-condition which can never be satisfied is

```sql
SELECT U.INDEX_NAME  
FROM   USER_IND_COLUMNS X, USER_IND_COLUMNS Y 
WHERE  X.INDEX_NAME = Y.INDEX_NAME 
AND    X.COLUMN_POSITION = Y.COLUMN_POSITION 
AND    X.TABLE_NAME = 'STUDENTS' 
AND    X.COLUMN_NAME = 'FIRST_NAME' 
AND    Y.COLUMN_NAME = 'LAST_NAME' 
```

Since INDEX_NAME and COLUMN_POSITION form a key of USER_IND_COLUMNS, the tuples X and Y must be the same. Thus, X.COLUMN_NAME = Y.COLUMN_NAME. This contradicts the last two conditions.
- The following query lists also indexes containing only one of the two columns:

```
SELECT INDEX_NAME  
FROM USER_IND_COLUMNS
AND TABLE_NAME = 'STUDENTS'
AND (COLUMN_NAME = 'FIRST_NAME'
OR COLUMN_NAME = 'LAST_NAME')
```

Wrong!

Since “and” is sometimes a bit ambiguous in natural language, I emphasized in the exercise that only indexes containing both columns should be printed.

- Two students added “AND X.COLUMN_NAME <> Y.COLUMN_NAME” to a query containing

```
WHERE X.COLUMN_NAME = 'FIRST_NAME'
AND Y.COLUMN_NAME = 'LAST_NAME'
```

However, this implies the additional condition. So it is an unnecessary complication to require that the column names in the two tuple variables are different.

- 10 students did not seem to understand the exercise. 3 students wrote

```
TABLE_NAME = STUDENTS
```

without the quotes around STUDENTS. This might also be an indication for not understanding that the table name is used here as data (as a string).

- 6 students (9%) used an unnecessary DISTINCT:

```
SELECT DISTINCT X.INDEX_NAME
FROM USER_IND_COLUMNS X, USER_IND_COLUMNS Y
WHERE X.INDEX_NAME = Y.INDEX_NAME
AND X.TABLE_NAME = 'STUDENTS'
AND X.COLUMN_NAME = 'FIRST_NAME'
AND Y.COLUMN_NAME = 'LAST_NAME'
```

Since INDEX_NAME and COLUMN_NAME is a key of USER_IND_COLUMNS, every index name will anyway be printed only once. This can be formally proven. Suppose that two assignments of tuples $X_1,Y_1$ and $X_2,Y_2$ to the tuple variables $X$ and $Y$ print the same index name. Then $X_1.INDEX_NAME = X_2.INDEX_NAME$. Because of the first WHERE-condition we also have $Y_1.INDEX_NAME = Y_2.INDEX_NAME$. The third WHERE-condition yields $X_1.COLUMN_NAME = X_2.COLUMN_NAME$. The fourth WHERE-condition gives us $Y_1.COLUMN_NAME = Y_2.COLUMN_NAME$. But now the key attributes of $X_1$ and $X_2$ and of $Y_1$ and $Y_2$ are equal, thus $X_1 = X_2$ and $Y_1 = Y_2$. 
b) Which indexes are only on one column?

- So we must ensure that there is only one row in USER_IND_COLUMNS for this index. One solution is to use NOT EXISTS:

```
SELECT INDEX_NAME, TABLE_NAME, COLUMN_NAME
FROM USER_IND_COLUMNS X
WHERE NOT EXISTS (SELECT *
    FROM USER_IND_COLUMNS Y
    WHERE Y.INDEX_NAME = X.INDEX_NAME
    AND Y.COLUMN_NAME <> X.COLUMN_NAME)
```

- Since the column position is a sequential number which always starts with 1, this is also possible:

```
SELECT INDEX_NAME, TABLE_NAME, COLUMN_NAME
FROM USER_IND_COLUMNS
WHERE X.INDEX_NAME NOT IN (SELECT INDEX_NAME
    FROM USER_IND_COLUMNS
    WHERE COLUMN_POSITION >= 2)
```

If there is a row for an index with column position 2, that index is not a single-column index.

- Another solution is to use HAVING. However, you must still use a subquery:

```
SELECT INDEX_NAME, TABLE_NAME, COLUMN_NAME
FROM USER_IND_COLUMNS
WHERE X.INDEX_NAME IN (SELECT INDEX_NAME
    FROM USER_IND_COLUMNS
    GROUP BY INDEX_NAME
    HAVING COUNT(*) = 1)
```

- This is a bit strange, but also works:

```
SELECT INDEX_NAME, TABLE_NAME, COLUMN_NAME
FROM USER_IND_COLUMNS X
WHERE 1 = (SELECT COUNT(*)
    FROM USER_IND_COLUMNS Y
    WHERE Y.INDEX_NAME = X.INDEX_NAME)
```

Only 16 students (23%) got this fully correct. Now some examples of incorrect solutions:
• A very common error was to do the `HAVING` in the same query as the selection of the column name. This error came in two variations. 9 students (13%) wrote basically this query:

```sql
SELECT INDEX_NAME, TABLE_NAME, COLUMN_NAME  
FROM   USER_IND_COLUMNS  
GROUP BY INDEX_NAME  
HAVING  COUNT(*) = 1
```

This does not work, because all attributes used in the select list outside aggregation functions must appear under `GROUP BY`. It would make sense, tough: `HAVING COUNT(*) = 1` selects groups consisting of a single row, so `TABLE_NAME` and `COLUMN_NAME` are actually uniquely defined. But the requirement is purely syntactic, the SQL parser does not bother to analyze the conditions for cases which guarantee groups of single rows. I tried such a situation in Oracle, DB2, and SQL Server, and all of them print an error message.

• Because of this problem, 18 students (26%) simply listed all the attributes under `GROUP BY`:

```sql
SELECT INDEX_NAME, TABLE_NAME, COLUMN_NAME  
FROM   USER_IND_COLUMNS  
GROUP BY INDEX_NAME, TABLE_NAME, COLUMN_NAME  
HAVING  COUNT(*) = 1
```

Listing the table name under `GROUP BY` is no problem (it is anyway functionally determined by the index name, so it does not change the groups). But putting the column name under `GROUP BY` does change the groups, and since `INDEX_NAME` and `COLUMN_NAME` together are a key of `USER_IND_COLUMNS`, all groups will consist only of a single row. So the `HAVING` condition will automatically be satisfied and the query becomes equivalent to

```sql
SELECT INDEX_NAME, TABLE_NAME, COLUMN_NAME  
FROM   USER_IND_COLUMNS
```

It simply lists all entries in `USER_IND_COLUMNS`.

• Two students grouped by the table name, which is also not correct.

• 7 students (10%) listed `COUNT(*)` in the `SELECT` clause, although they had the condition `HAVING COUNT(*) = 1`. The output will always be 1, so is not very interesting. Probably these students assumed that when you use an aggregation under `HAVING`, you also must list it under `SELECT`. But that is not true. One student even wrote
SELECT INDEX_NAME, TABLE_NAME, COLUMN_NAME  
FROM USER_IND_COLUMNS 
WHERE X.INDEX_NAME IN (SELECT INDEX_NAME, COUNT(*) 
FROM USER_IND_COLUMNS 
GROUP BY INDEX_NAME 
HAVING COUNT(*) = 1)

This is a syntax error because the left hand side and the right hand side of IN 
must have the same number of columns (typically one).

- 6 students only checked that the column position is 1:

SELECT INDEX_NAME, TABLE_NAME, COLUMN_NAME  
FROM USER_IND_COLUMNS 
WHERE COLUMN_POSITION = 1

This simply lists the first column of every index. However, there might be other 
columns in the index, which violate the query requirement.

The following query is also equivalent (and thus wrong), but uses a very 
strange (albeit legal) notation for the =:

SELECT INDEX_NAME, TABLE_NAME, COLUMN_NAME  
FROM USER_IND_COLUMNS 
WHERE COLUMN_POSITION IN (1)

Another equivalent query, but with a very strange condition is:

SELECT INDEX_NAME, TABLE_NAME, COLUMN_NAME  
FROM USER_IND_COLUMNS 
WHERE COLUMN_POSITION = 1 
AND COLUMN_POSITION NOT IN (SELECT COLUMN_POSITION 
FROM USER_IND_COLUMNS 
WHERE COLUMN_POSITION <= 2)

The subquery will return only numbers \(\geq 2\), so the NOT IN will always be true.

- I had never thought that anybody would use GROUP BY without an aggregation 
function. But it did happen several times. The following is an example:

SELECT INDEX_NAME, TABLE_NAME, COLUMN_NAME  
FROM USER_IND_COLUMNS 
WHERE COLUMN_POSITION = 1 
GROUP BY INDEX_NAME, TABLE_NAME, COLUMN_NAME
This is a legal SQL query. Using GROUP BY without an aggregation function is only a strange way of duplicate elimination. But in this case, there can be no duplicates, since a key of the table is part of the SELECT-list. So the GROUP BY clause does nothing here.

- Three students used an uncorrelated subquery with NOT EXISTS in this exercise, and three students wrote COUNT(*) = 1 directly under WHERE: Aggregation functions cannot be used under WHERE (except inside subqueries).
c) Print for every table, which has at least five columns, the number of columns. The output should contain the table name and this number.

- Most students used the following solution, which is similar to queries shown in the course handouts and in the solutions of previous exams:

```sql
SELECT TABLE_NAME, COUNT(*)
FROM COLS
GROUP BY TABLE_NAME
HAVING COUNT(*) >= 5
```

You can also write an attribute such as `COLUMN_NAME` inside the `COUNT`. Even `TABLE_NAME` would work, although that is quite strange.

- Since the `COLUMN_ID` sequentially numbers every column within a table, this also works:

```sql
SELECT TABLE_NAME, COLUMN_ID
FROM COLS X
WHERE COLUMN_ID >= 5
AND NOT EXISTS (SELECT *
FROM COLS Y
WHERE Y.TABLE_NAME = X.TABLE_NAME
AND Y.COLUMN_ID > X.COLUMN_ID)
```

However, this is more complicated and less natural than the first solution.

47 students (67%) got this fully correct. Now some incorrect solutions:

- Attributes declared in the `SELECT`-clause cannot be used under `WHERE`, `GROUP BY`, or `HAVING`. They can be used under `ORDER BY`. The reason is that the `SELECT`-clause is evaluated after `WHERE`, `GROUP BY`, and `HAVING`, but before `ORDER BY`. So the following is a syntax error:

```sql
SELECT TABLE_NAME, COUNT(*) NUM_COLS
FROM COLS
GROUP BY TABLE_NAME
HAVING NUM_COLS >= 5
```

- The following will give a type error. You can sum only numbers:

```sql
SELECT TABLE_NAME, SUM(COLUMN_NAME)
FROM COLS
GROUP BY TABLE_NAME
HAVING SUM(COLUMN_NAME) >= 5
```

Wrong!
• Three students used `DISTINCT`:

```sql
SELECT DISTINCT TABLE_NAME, COUNT(*)
FROM COLS
GROUP BY TABLE_NAME
HAVING COUNT(*) >= 5
```

However, if we group by `TABLE_NAME`, every table name will anyway be printed only once. So the `DISTINCT` does not change anything and is an unnecessary complication.

• Two students used `DISTINCT` inside the `COUNT`:

```sql
SELECT TABLE_NAME, COUNT(DISTINCT COLUMN_NAME)
FROM COLS
GROUP BY TABLE_NAME
HAVING COUNT(DISTINCT COLUMN_NAME) >= 5
```

Since we group by the table name, the column names will be distinct: `TABLE_NAME` and `COLUMN_NAME` are together a key of `COLS`. SO again this would work, but is an unnecessary complication.

• This solution is very complicated, but nevertheless false:

```sql
SELECT TABLE_NAME, COUNT(*)      Wrong!
FROM COLS X
GROUP BY TABLE_NAME
HAVING COUNT(*) >= ALL (SELECT COUNT(*)
                        FROM COLS Y
                        WHERE Y.TABLE_NAME = X.TABLE_NAME
                        GROUP BY Y.TABLE_NAME
                        HAVING COUNT(*) >= 5)
```

The `GROUP BY` in the subquery is actually not needed, since the table name is anyway fixed (the subquery is evaluated once for every table name in the outer query). If the table has at least 5 columns, the subquery returns the number of columns, and the `>= ALL` is true. If the table has less than 5 columns, the subquery returns the empty set. But in that case `>= ALL` is also true (`>= ANY` would be false). So the difficult `HAVING`-condition is always satisfied and the query is equivalent to

```sql
SELECT TABLE_NAME, COUNT(*)      Wrong!
FROM COLS X
GROUP BY TABLE_NAME
```

• The following solution is correct, but contains an unnecessary complication:
SELECT  TABLE_NAME, COUNT(*)  Bad Style!
FROM    COLS X
WHERE   COLUMN_ID > 0
GROUP BY TABLE_NAME
HAVING  COUNT(*) >= 5

The condition “COLUMN_ID > 0” was declared in Exercise 2 as a CHECK-constraint. That means that the DBMS enforces that this condition will always be true. So there is no need to check for it in a query condition. A similar case is to use IS NOT NULL for an attribute which is declared as NOT NULL. I have also seen the test IS NULL for an attribute which could never be null. And I have seen joins which could not change anything because of a foreign key constraint. The common theme of all this is that you should keep the integrity constraints in mind when writing queries.
d) Define a view which contains table name, column name, and ID of all columns which are contained in an index.

- This needs simply a join between **COLS** and **USER_IND_COLUMNS**:

```sql
CREATE VIEW INDEXED_COLUMNS AS
SELECT DISTINCT X.TABLE_NAME, X.COLUMN_NAME, X.COLUMN_ID
FROM COLS X, USER_IND_COLUMNS Y
WHERE X.TABLE_NAME = Y.TABLE_NAME
AND X.COLUMN_NAME = Y.COLUMN_NAME
```

Note that the **DISTINCT** is necessary here, since the same column can be contained in different (overlapping) indexes.

- The following variant does not need **DISTINCT**, but requires a subquery:

```sql
CREATE VIEW INDEXED_COLUMNS AS
SELECT TABLE_NAME, COLUMN_NAME, COLUMN_ID
FROM COLS X
WHERE EXISTS (SELECT *
FROM USER_IND_COLUMNS Y
WHERE Y.TABLE_NAME = X.TABLE_NAME
AND Y.COLUMN_NAME = X.COLUMN_NAME)
```

Note that **DISTINCT** is not necessary here, since every entry in **COLS** is considered only once.

- It is also possible to use **IN**, but table name and column name must be compared (two different tables can have columns with the same name):

```sql
CREATE VIEW INDEXED_COLUMNS AS
SELECT TABLE_NAME, COLUMN_NAME, COLUMN_ID
FROM COLS
WHERE (TABLE_NAME,COLUMN_NAME) IN (SELECT TABLE_NAME,COLUMN_NAME
FROM USER_IND_COLUMNS)
```

- In each of these solutions, you can also define column names for the view:

```sql
CREATE VIEW INDEXED_COLUMNS(TABLE_NAME, COLUMN_NAME, COLUMN_ID) AS
SELECT TABLE_NAME, COLUMN_NAME, COLUMN_ID FROM ...
```

20 students (29%) got this fully correct. Now some examples for incorrect solutions:

- The duplicate elimination was a major source of trouble in this exercise. 9 students (13%) forgot the **DISTINCT**. The exercise explicitly mentioned that there
can be overlapping indexes and that each column (for a given table) should be printed only once. Because of this unclear formulation, 8 students (11%) wrote

```
SELECT TABLE_NAME, DISTINCT COLUMN_NAME, COLUMN_ID
Wrong!
```

However, `DISTINCT` can only apply to entire output rows, and must be written immediately after `SELECT`. What should it mean to eliminate duplicates only for one output column? The following is also a syntax error:

```
SELECT DISTINCT (TABLE_NAME, COLUMN_NAME, COLUMN_ID)
Wrong!
```

Whereas parentheses can be freely used inside expressions and conditions, they cannot be used to combine several output columns. In the same way, this is wrong:

```
SELECT (DISTINCT TABLE_NAME, COLUMN_NAME, COLUMN_ID)
Wrong!
```

In total, 11 students (16%) had problems with the correct placement of `DISTINCT`.

- 8 students (11%) used `GROUP BY` without an aggregation. The following would actually work, but is a very strange kind of duplicate elimination:

```
CREATE VIEW INDEXED_COLUMNS AS
SELECT X.TABLE_NAME, X.COLUMN_NAME, X.COLUMN_ID
FROM COLS X, USER_IND_COLUMNS Y
WHERE X.TABLE_NAME = Y.TABLE_NAME
AND X.COLUMN_NAME = Y.TABLE_NAME
GROUP BY X.TABLE_NAME, X.COLUMN_NAME, X.COLUMN_ID
```

But most of the 8 students were less successful and lost points. `GROUP BY` is only intended to be used together with aggregations.

- Seven students joined `COLS` with `IND`, so they misunderstood the exercise. Three students misunderstood the meaning of the column ID. Two students used only the `COLS` table and had no test whether the column is indexed. So there was a relatively high rate of misunderstandings.

- Three students forgot the join on the column name.

- There were also a few errors with the syntax of subqueries. One student used a subquery under `WHERE` without `EXISTS` or `IN` etc. Another student wrote `NOT EXISTS IN` and a third student used `IN` without an expression on the left hand side.
e) Define a view which contains table name, column name, and ID of all columns which are not contained in any index.

- Now, we need entries in COLS without a matching entry in USER_IND_COLUMNS:

```sql
CREATE VIEW NONINDEXED_COLUMNS AS
SELECT TABLE_NAME, COLUMN_NAME, COLUMN_ID
FROM COLS X
WHERE NOT EXISTS (SELECT *
FROM USER_IND_COLUMNS Y
WHERE Y.TABLE_NAME = X.TABLE_NAME
AND Y.COLUMN_NAME = X.COLUMN_NAME)
```

- It is again possible to use IN:

```sql
CREATE VIEW NONINDEXED_COLUMNS AS
SELECT TABLE_NAME, COLUMN_NAME, COLUMN_ID
FROM COLS
WHERE (TABLE_NAME, COLUMN_NAME) NOT IN
(SELECT TABLE_NAME, COLUMN_NAME
FROM USER_IND_COLUMNS)
```

21 students (30%) got this completely correct. Now some examples of incorrect queries. For simplicity, the CREATE VIEW part is not shown.

- Because of its nonmonotonic behaviour, the query needs NOT EXISTS or NOT IN or something similar. The following solution is wrong:

```sql
SELECT X.TABLE_NAME, X.COLUMN_NAME, X.COLUMN_ID
FROM COLS X, USER_IND_COLUMNS Y
WHERE Y.TABLE_NAME <> X.TABLE_NAME
AND Y.COLUMN_NAME <> X.COLUMN_NAME
```

If USER_IND_COLUMNS contains at least two rows with different column names and different table names, every entry in COLS will be printed, and not only once, but very often (depending on the number of rows in USER_IND_COLUMNS with a different table and column name). Fortunately, only one student made this error. I warned often in the course that the non-existence of a row is something different than the existence of a row with a different value.

- If the join (anti-join) is done only on the table name, a column from a table will not be printed, even if only some other column of the same table is indexed. 11 students (16%) made this type of error (but sometimes IND instead
of \texttt{USER\_IND\_COLUMNS} was used), another 4 students joined only on the column name.

\begin{verbatim}
SELECT TABLE_NAME, COLUMN_NAME, COLUMN_ID  \hspace{2cm} \textbf{Wrong!}
FROM    COLS
WHERE   TABLE_NAME NOT IN (SELECT TABLE_NAME
                            FROM    USER\_IND\_COLUMNS)
\end{verbatim}

- It is also not correct to compare the table name and the column name in isolation:

\begin{verbatim}
SELECT TABLE_NAME, COLUMN_NAME, COLUMN_ID  \hspace{2cm} \textbf{Wrong!}
FROM    COLS
WHERE   TABLE_NAME NOT IN (SELECT TABLE_NAME
                            FROM    USER\_IND\_COLUMNS)
\end{verbatim}

\begin{verbatim}
AND     COLUMN_NAME NOT IN (SELECT COLUMN_NAME
                            FROM    USER\_IND\_COLUMNS)
\end{verbatim}

This is much too strong: It will print only columns in tables when no index exists on that table (on matter on which column) and also no column with the same name in another table is indexed. You must compare the combination of table name and column name.

- This is an example for a domain error. How should a column name and a table name ever match?

\begin{verbatim}
SELECT TABLE_NAME, COLUMN_NAME, COLUMN_ID  \hspace{2cm} \textbf{Wrong!}
FROM    COLS
WHERE   COLUMN_NAME NOT IN (SELECT TABLE_NAME
                            FROM    USER\_IND\_COLUMNS)
\end{verbatim}

So the output will simply be all entries in \texttt{COLS} (unless there happens to be a column with the same name as a table name). Since Oracle and most other systems do not yet support domain declarations, this kind of “type error” will not be caught by the system.

- In this example, the join condition for the column is expressed two times:

\begin{verbatim}
SELECT TABLE_NAME, COLUMN_NAME, COLUMN_ID  \hspace{2cm} \textbf{Bad Style!}
FROM    COLS X
WHERE   X.COLUMN_NAME NOT IN (SELECT Y.COLUMN_NAME
                                FROM    USER\_IND\_COLUMNS Y
                                WHERE Y.TABLE_NAME = X.TABLE_NAME
                                AND     Y.COLUMN_NAME = X.COLUMN_NAME)
\end{verbatim}

Since the subquery checks that “\texttt{Y.COLUMN_NAME = X.COLUMN_NAME}”, it can only return \texttt{X.COLUMN_NAME}. So if its result is not empty, the \texttt{NOT IN} will automatically
be violated. It is better to use `NOT EXISTS` here (so the unnecessary second comparison is avoided). Of course, it would also be possible to remove the condition “`Y.COLUMN_NAME = X.COLUMN_NAME`” from the subquery. But it is not nice style to do the comparison in part with `IN`, and in part inside the subquery. Although correlated subqueries with `IN` can be correct, they should be avoided.

- An uncorrelated subquery with `NOT EXISTS` is almost always an error. I did mention this in the course, but 9 students (13%) had an error of this kind.

```sql
SELECT TABLE_NAME, COLUMN_NAME, COLUMN_ID  
FROM COLS  
WHERE NOT EXISTS (SELECT *  
                  FROM USER_IND_COLUMNS)
```  

If there is a single index in the database (no matter for which column), this view will be empty.

- This is a more complicated example of an uncorrelated subquery:

```sql
SELECT TABLE_NAME, COLUMN_NAME, COLUMN_ID  
FROM COLS X  
WHERE NOT EXISTS (SELECT *  
                  FROM COLS X, USER_IND_COLUMNS Y  
                  WHERE Y.TABLE_NAME = X.TABLE_NAME  
                  AND Y.COLUMN_NAME = X.COLUMN_NAME)
```  

The problem is that `COLS X` is declared again in the subquery. In this way, the tuple variable from the outer query is shadowed, and `X` in the subquery refers to the tuple variable declared in the subquery. In this way, if there is a single column that is indexed in the database, no matter for what table, the output will be empty.

So it is important to realize that tuple variables from the outer query do not have to be redeclared in the inner query. Even if this should not lead to an error, this is at least bad style. Exceptions might only be implicit tuple variables, which have the same names as relations. But this situation must also be treated with great care.

- Some students also seemed to think that tuple variables used in the inner query must also be declared in the outer query. Again, this is wrong. An example of such an error is the following query. Its condition is always false. 6 students (9%) made an error of this type. 2 students had another kind of contradiction in this exercise.
SELECT X.TABLE_NAME, X.COLUMN_NAME, X.COLUMN_ID  
FROM  COLS X, USER_IND_COLUMNS Y 
WHERE  X.TABLE_NAME = Y.TABLE_NAME 
AND  X.COLUMN_NAME = Y.COLUMN_NAME 
AND  NOT EXISTS (SELECT *  
FROM  USER_IND_COLUMNS Y  
WHERE  X.TABLE_NAME = Y.TABLE_NAME  
AND  X.COLUMN_NAME = Y.COLUMN_NAME)

The WHERE clause can never be satisfied, because the join in the outer query requires that there is a matching entry in USER_IND_COLUMNS, while the NOT EXISTS subquery checks that there is no such entry.

- Now suppose we remove the join condition in the outer query:

SELECT X.TABLE_NAME, X.COLUMN_NAME, X.COLUMN_ID  
FROM  COLS X, USER_IND_COLUMNS Y 
WHERE  NOT EXISTS (SELECT *  
FROM  USER_IND_COLUMNS Y  
WHERE  X.TABLE_NAME = Y.TABLE_NAME  
AND  X.COLUMN_NAME = Y.COLUMN_NAME)

It would work, but you will get many duplicates (every result is printed as often as there are rows in USER_IND_COLUMNS). 4 students had such unnecessary tuple variables in this exercise.

    Putting a DISTINCT in the query fixes the problem with the duplicates, but creates a new performance problem. It is very important to understand the real source of a problem before trying to solve it.

- An even more direct contradiction (which was also actually written in the exam) is:

SELECT TABLE_NAME, COLUMN_NAME, COLUMN_ID  
FROM  COLS 
WHERE  COLUMN_NAME NOT IN (SELECT COLUMN_NAME  
FROM  COLS)
f) Use the views defined in d) and e) in a query which lists all columns (table name, column name, ID) together with an indication whether the column is indexed or not. I.e. you should print “Y” for the columns listed in the view d) and “N” for the columns listed in the view e). Please sort the output by table name and column ID.

- This needs a **UNION** for the two possible cases:

  ```sql
  SELECT TABLE_NAME, COLUMN_NAME, COLUMN_ID, 'Y' INDEXED
  FROM   INDEXED_COLS
  UNION
  SELECT TABLE_NAME, COLUMN_NAME, COLUMN_ID, 'N' INDEXED
  FROM   NONINDEXED_COLS
  ORDER BY TABLE_NAME, COLUMN_ID
  ```

- Since all columns of the views have to be printed, the following also works:

  ```sql
  SELECT C.*, 'Y' INDEXED
  FROM   INDEXED_COLS C
  UNION
  SELECT C.*, 'N' INDEXED
  FROM   NONINDEXED_COLS C
  ```

  Note that “*” without tuple variable can only be used if this is the only item of the SELECT-list.

- It was not required to name the column with the index indication, and one can also use the positional notation for the **ORDER BY**-clause. So the shortest solution is probably:

  ```sql
  SELECT C.*, 'Y'
  FROM   INDEXED_COLS C
  UNION
  SELECT C.*, 'N'
  FROM   NONINDEXED_COLS C
  ORDER BY 1, 3
  ```

18 students (26%) solved this exercise completely correct. Now some examples of incorrect solutions.

- The biggest problem was how to put the constant strings ‘Y’ and ‘N’ into the query result: 25 students (36%) did not use the correct syntax (plus 12 students skipped the exercise). I have seen many variations, but only the syntax shown above is correct (you can put double quotes around INDEXED if you prefer that).
For instance, writing `INDEXED = 'Y'` is neither correct in the `SELECT`-list (no conditions are allowed there), nor under `WHERE` (none of the tuple variables has a column called “INDEXED”).

- **ORDER BY** is allowed only at the very end of a query:

  ```sql
  SELECT C.*, 'Y' INDEXED           Wrong!
  FROM   INDEXED_COLS C
  ORDER BY TABLE_NAME, COLUMN_ID
  UNION
  SELECT C.*, 'N' INDEXED
  FROM   NONINDEXED_COLS C
  ```

  It is also wrong to repeat the **ORDER BY** at the end of both query parts.

- Six students forgot the **ORDER BY** clause, one used **GROUP BY** instead, and one got the **ORDER BY** specification wrong.

- A few students tried to use a join instead of a union.
Exercise 4 (FDs, BCNF) 6 Points

18 students (26%) got the full points, 17 students (24%) lost 1 point, 17 students (24%) lost 2 points, 9 students (13%) lost 3 points, 7 students (10%) lost 4 points, and one student lost 5 points.

a) Does the functional dependency “COLUMN_NAME → COLUMN_POSITION” hold here? It might hold in this example database state although we assume that it does not hold in general.

- No, the entries for STUD_ID violate this FD.
- No, the entries for EX_NO violate this FD.
- No, the entries for FIRST_NAME and LAST_NAME violate this FD.
- Yes, this FD is satisfied in the given database state.

19 students (27%) got this wrong (7 checked the second box, 6 the third, and 6 the fourth).

b) What does the functional dependency “INDEX_NAME → TABLE_NAME” mean?

- Every table has only one index.
- Every index is for a uniquely determined table.
- No index can have more than one column.

7 students (10%) got this wrong (4 checked the first box and 3 the second).

c) Somebody tells you that INDEX_NAME, COLUMN_POSITION → TABLE_NAME holds, too. What is your reaction?

- This is impossible. It is already violated in the example state.
- This is true, but not very interesting, since it is weaker than (implied by) the second given FD.
- This contradicts the given key. If it is really true, we must determine another key.

9 students (13%) got this wrong: 5 checked the first box and 4 the last.
d) Given the three functional dependencies, is “INDEX_NAME, COLUMN_NAME” an alternative key for the relation?

☑ Yes.
☐ No.

15 students (21%) got this wrong. It is an alternative key because of the FDs

- INDEX_NAME → TABLE_NAME
- INDEX_NAME, COLUMN_NAME → COLUMN_POSITION

The first FD is stronger than (implies)

- INDEX_NAME, COLUMN_NAME → TABLE_NAME

so that the key attributes together determine all other attributes of the table.

e) Is the relation in BCNF? You may ignore the third FD for this question, since otherwise the correct answer might depend on your answer for d).

☐ Yes.
☐ No. The FD “INDEX_NAME, COLUMN_POSITION → COLUMN_NAME” violates BCNF.
☑ No. The FD “INDEX_NAME → TABLE_NAME” violates BCNF.
☐ No. Both FDs violate the BCNF condition.

36 students (51%) got this wrong: 18 checked the first box, 10 the second, and 8 the last.

BCNF means that every (non-trivial) FD must have a key (or “superkey”) on its left hand side. INDEX_NAME alone is not a key, so BCNF is violated because INDEX_NAME determines TABLE_NAME. However, the combination of INDEX_NAME and COLUMN_POSITION is a key of the table, therefore the other FD does not violate BCNF. As mentioned in the exercise, it is no problem that USER_IND_COLUMNS violates BCNF, since it is actually a view and not a table.

f) What is one important purpose of normalization?

☐ We want to minimize the number of joins in queries.
☑ We want to enforce a specific class of constraints (FDs) via key constraints.
☐ We want to minimize the number of attributes in each relation.
21 students (30%) got this wrong: 6 checked the first box, and 15 the last.

The first answer is completely wrong: Normalization will actually lead to more joins. The last answer is also wrong, although normalization often reduces the number of columns (however, it increases the number of tables). Minimization of the number of columns is not the purpose of normalization, it is only sometimes a by-product. If we really wanted to minimize the number of columns, every table would consist only of the key plus possibly one more attribute.
Exercise 5 (Security)  
3 Points

37 students (53%) got the full points, 24 students (34%) lost one point, 7 students (10%) lost two points, and 1 student lost all points.

a) Ann creates a table SECRET1 and executes the following command:

```
GRANT SELECT, INSERT ON SECRET1 TO BILL.
```

Sometime later she enters this command:

```
GRANT DELETE ON SECRET1 TO BILL.
```

What rights on SECRET1 does Bill now have?

- [ ] He got the DELETE right, but lost the SELECT and INSERT rights.
- [ ] This command will result in an error message. You cannot grant DELETE without granting SELECT in the same command.
- [x] Bill now has SELECT, INSERT and DELETE rights on this table.

Only 4 students (96%) got this wrong: 2 checked the first box and 2 the second.

b) Ann creates a table SECRET2 and executes the following command:

```
GRANT SELECT ON SECRET2 TO BILL WITH GRANT OPTION.
```

Then Bill executes:

```
GRANT SELECT ON SECRET2 TO CHRIS.
```

Later Ann enters:

```
REVOKE SELECT ON SECRET2 FROM BILL.
```

Let us assume that Oracle is used in this example. What is the result?

- [ ] Bill loses the SELECT right, Chris keeps it.
- [x] Bill and Chris both lose the SELECT right.
- [ ] There is no way to revoke a right which was given with grant option.

Only 5 students (7%) got this wrong: 3 checked the first box and 2 the last.

c) Now Ann creates a table SECRET3 and executes the following command:

```
GRANT SELECT ON SECRET3 TO BILL.
```

Bill executes the command

```
CREATE VIEW MY_SECRET AS SELECT * FROM SECRET3
```

and then

```
GRANT SELECT ON MY_SECRET TO CHRIS.
```

Is it possible to share the secret in this way with Chris?
28 students (40%) got this wrong. If it were so easy to circumvent the missing grant option, it would make no sense to have it as an option. Oracle and DB2 require that you have the right you want to grant on the view with the grant option. SQL Server only allows you to grant rights on the view if you own the base tables used in the view definition. In DB2, you get the control right on the view only if you have the control right on the base tables.

Note however that Bill can use

```
CREATE TABLE MY_SECRET AS SELECT * FROM SECRET3
```

to create his own copy of the table. He then can give rights on this copy to anybody he wishes. Mandatory access control was invented to avoid such things, but it is not part of the SQL standard.