ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

HS 2020

Prof. R. Wattenhofer
Simon Tanner

# Computational Thinking
# Solutions to Exercise 8 (Data and Storage)

## 1 Journal Article Database

**a)** Both {`ID,TR-ID`} and {`ID,title,TR-ID`} are superkeys, because they uniquely identify any row within the table. As there are two rows with the same `ID` and two rows with the same `TR-ID`, it is not sufficient to use only one of these two columns to identify a row. Hence, {`ID,TR-ID`} is a candidate key. Thus, {`ID,title,TR-ID`} is not a candidate key, because the `title` column can be omitted.

**b)** Query 3. results in:

```
ERROR: function sum(text) does not exist
```

PostgreSQL does not accept strings as input to the SUM function. STRING_AGG would work to concatenate strings.

Query 6. results in:

```
ERROR: aggregate functions are not allowed in WHERE
```

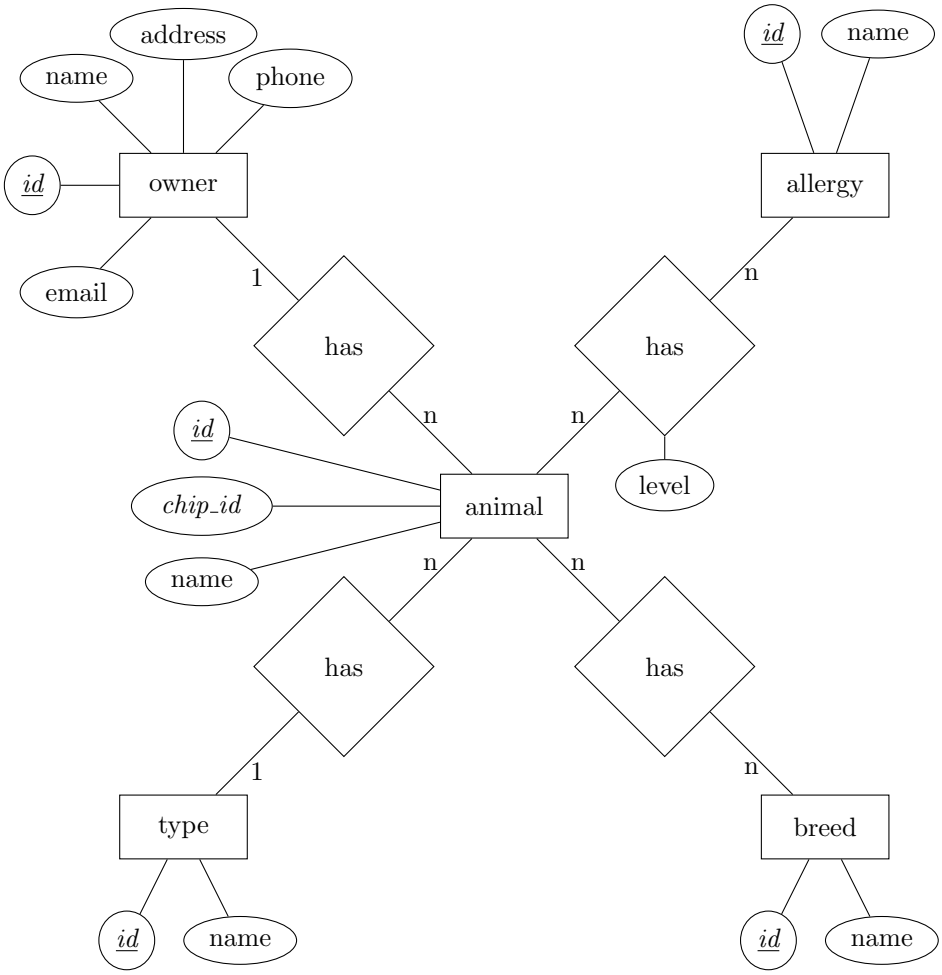
SQL's WHERE clause does not work with aggregate functions like SUM, AVG, MAX, COUNT and so on. Instead, the HAVING keyword was introduced to SQL in order to quantitatively compare aggregated values. A correct query would look like this:

```
SELECT year, COUNT(*) FROM Articles GROUP BY year HAVING COUNT(*) > 10;
```

**c)** 3 rows

## 2 Database Design

The following Entity Relationship Diagram describes the animals database. Owners and animals are in a 1-to-$n$ relation. Each owner may own multiple animals, but every animal can have exactly one registered owner in the database. Animals and animal types are in a $n$-to-1 relation. Any animal cannot be both a cat and a dog, but the animal type table may very well contain multiple cats or dogs. Animals and breeds are in a $n$-to-$n$ relation. Any animal can be a mixed breed and there may be multiple animals of the same breed in the database. Animals and allergies are in a $n$-to-$n$ relation. Any animal may have multiple allergies and any allergy may afflict more than one animal in the database. For every animal allergy, we reserve a level field that denotes how strongly allergic the animal is to the allergy in question. Notice that we underline primary key attributes and we use an italic font to label unique attributes. The id field is chosen as the primary key for the animal table because even though the chip_id is a unique value, we would like to allow for the possibility that the unique chip_id is changed, for example, when the animal chip breaks or if it's updated due to a change in chip standard.




# 3 Database Queries

**a)** `SELECT id, title FROM movie LIMIT 5;`

**b)** `SELECT * FROM movie ORDER BY title DESC LIMIT 2;`

**c)** `SELECT COUNT(*) FROM movie WHERE year > 2000;`

**d)** `SELECT title, tomatometer FROM movie WHERE title = 'The Matrix';`

**e)**

```
SELECT COUNT(*) FROM movie
WHERE tomatometer > (
  SELECT tomatometer FROM movie
  WHERE title = 'The Matrix');
```

**f)**

```
SELECT year, AVG(tomatometer) AS avg FROM movie
GROUP BY year
ORDER BY avg DESC LIMIT 5;
```

**g)**

```
SELECT title FROM movie
WHERE title LIKE 'X%'
ORDER BY title DESC;
```

**h)**

```
SELECT COUNT(*) FROM movie
WHERE title LIKE '%fight%';
```

# 4 Advanced Database Queries

**a)**

```
SELECT person.name, cast_info.role_id, person.gender
FROM cast_info
JOIN person ON person.id = cast_info.person_id
JOIN movie ON movie.id = cast_info.movie_id
JOIN role_type ON role_type.id = cast_info.role_id
WHERE role_type.role = 'actress' AND movie.title = 'The Matrix';
```

**b)**

```
SELECT COUNT(DISTINCT person.id)
FROM cast_info
JOIN role_type ON role_type.id = cast_info.role_id
JOIN person ON person.id = cast_info.person_id
WHERE role_type.role = 'director' AND person.gender = 'f';
```

**c)**

```
SELECT DISTINCT person.name FROM cast_info
    JOIN person ON person.id = cast_info.person_id
    JOIN movie ON movie.id = cast_info.movie_id
  WHERE (cast_info.role_id = 2 or cast_info.role_id = 1)
  AND EXISTS (
    SELECT DISTINCT ci.person_id FROM cast_info AS ci
      WHERE ci.role_id = 8
      AND cast_info.person_id = ci.person_id
      GROUP BY ci.person_id
      HAVING COUNT(ci.person_id) > 20
  );
```

Alternative solution:

```
SELECT DISTINCT person.name FROM person
  JOIN cast_info ON person.id = cast_info.person_id
  JOIN role_type ON cast_info.role_id=role_type.id
WHERE role_type.role IN ('actor','actress')
AND 20 < (
  SELECT COUNT(*) FROM cast_info AS ci
    JOIN role_type AS rt ON ci.role_id=rt.id
    WHERE ci.person_id = person.id
    AND rt.role='director'
);
```

**d)**

```
SELECT movie.title, COUNT(*) AS cnt
FROM movie_keyword
JOIN movie ON movie_keyword.movie_id = movie.id
GROUP BY movie.id
ORDER BY cnt DESC
LIMIT 1;
```

**e)**

```
SELECT AVG(cnt), MAX(cnt), MIN(cnt) FROM (
  SELECT movie.title, COUNT(*) AS cnt
  FROM movie_keyword
  JOIN movie ON movie_keyword.movie_id = movie.id
  GROUP BY movie.id
) AS countaverages;
```

**f)**

```
SELECT
  person.name,
  AVG(movie.tomatometer) AS average,
  COUNT(ci.person_id) AS cnt,
  MAX(movie.year) AS maxyear
FROM cast_info AS ci
JOIN movie ON movie.id = ci.movie_id
JOIN person ON person.id = ci.person_id
WHERE ci.role_id = 1
GROUP BY person.id
HAVING AVG(movie.tomatometer) > 85 AND COUNT(ci.person_id) > 30
  AND MAX(movie.year) > 2000
ORDER BY maxyear DESC, average DESC;
```

**g)**

```
SELECT person.name
FROM person
JOIN cast_info ON person.id = cast_info.person_id
JOIN movie ON cast_info.movie_id = movie.id
WHERE cast_info.role_id = 8 AND movie.tomatometer > 90
GROUP BY person.id
HAVING COUNT(*) > 10;
```