Databases Exam

TDA357 (Chalmers), DIT620 (University of Gothenburg)

16 March 2018 at 8:30–12:30 in SB Multisal

Department of Computer Science and Engineering

Examiner: Aarne Ranta tel. 031 772 10 82.

Results: Will be published by 6 April.

Exam review: 19 April at 14-16 in room 6106, EDIT Building.

Grades: Chalmers: 24 for 3, 36 for 4, 48 for 5. GU: 24 for G, 42 for VG.

Help material: One "cheat sheet", which is an A4 sheet with hand-written notes. You may write on both sides of that sheet. If you bring a sheet, it must be handed in with your answers to the exam questions. One English language dictionary is also allowed.

Specific instructions: You can answer in English, Danish, Dutch, Finnish, French, German, Italian, Norwegian, Spanish, or Swedish (in this exam; next time it can be another set of languages ;-) Begin the answer to each question (numbers 1 to 6) on a new page. The a,b,c,... parts with the same number can be on the same page. If you need many pages for one question, number the pages as well, for instance, "Question 3 p. 2". Write the question number on every page.

Write clearly: unreadable = wrong! Fewer points are given for unnecessarily complicated solutions. Indicate clearly if you make any assumptions that are not given in the question. In particular: in SQL questions, use standard SQL or PostgreSQL. If you use any other variant (such as Oracle or MySQL), say this; but full points are not guaranteed since this may change the nature of the question.

.

1. Modelling (12p)

The domain to model is public transport networks. Such a network consists of lines, which run between stops (such as train stations and bus stops) on scheduled times, and are operated by different kinds of vehicles (such as buses and trams). A database on this domain is expected to serve for queries like "how can I get from A to B at 8:00", "how long does it take from A to B", "which lines serve the station A", etc.

We will approach the complex task in smaller steps. We start by building an ER (Entity-Relationship) model in two steps.

1a (5p)

Build an ER model for the following concepts:

- A **stop** has a **name**, which uniquely identifies it. A stop also has a **location**, which by nature is unique as well (since there cannot be two stops at the same location).
- A **line** has a **number**, which uniquely identifies it. A line also has two stops as its **start point** and **end point**. Moreover, a line has a **vehicle**, such as tram or bus or ferry.

Also write a database schema, in the form of SQL `CREATE TABLE` statements, corresponding to this description. The schema will be graded independently of your ER model. You can get started by deriving the schema from your ER model, but notice that a schema can often express constraints that the ER model cannot. Therefore. don't panic if you cannot express all the constraints in your ER model.

1b (7p)

This is a variation of 1a, where we take into account all the stops along a line, not only the start and end points.

- (like in 1a:) A **stop** has a **name**, which uniquely identifies it. A stop also has a **location**, which by nature is unique as well (since there cannot be two stops at the same location).
- A **line** has a **number** and a **vehicle**. The number identifies the line uniquely.
- Each line runs through a set of stops, **the stops on that line**. Each stop on a line has a **travel time**, which is the number of minutes it takes to travel to that stop from the start point of the line. A stop on a line is uniquely determined by the line number and the stop name. But we also require the travel times to be unique, i.e. that it takes at least one minute to travel from one stop to the next. (The start and the end points of a line are then indirectly defined by the minimum and maximum travel times.)

The task is again to build an ER model, as well as a schema in the form of SQL `CREATE TABLE` statements. You should build all of this separately from (1a). The schema should express all the relevant constraints that can be expressed in SQL. Some of them might not be expressible in the ER model.

2. Dependencies (8p)

The following table shows a few connections from the public transport system of Gothenburg.

line	**vehicle**	**model**	**start point**	**start city**	**end point**	**end city**	**start time**	**capacity**
2	tram	M28	AxelDs torg	Göteborg	Mölndal	Mölndal	16:26	116
2	tram	M31	AxelDs torg	Göteborg	Mölndal	Mölndal	17:16	202
4	tram	M31	Mölndal	Mölndal	Angered	Göteborg	16:26	202
16	bus	B9S	Fyrktorget	Göteborg	Eketrägatan	Göteborg	12:44	138
55	bus	7900	SvenHs plats	Göteborg	Lindholmen	Göteborg	10:05	105

2a (5p)

You can find many redundancies in this table, and your task is to identify their causes and eliminate them. More precisely:

- Show the functional dependencies that hold for this table (the basic ones; you can leave out derived ones).
  - Start by following the definition of a functional dependency (in the "standard cheat sheet") and applying it to the table.
  - List all the dependencies found in this way (but you can safely leave out derived dependencies).
  - This list will probably contain some bogus dependencies that cease to hold when you add more data, as well as some real dependencies.
  - Explain why you consider some dependencies bogus and leave them out, e.g. by giving examples of realistic data that would destroy those dependencies.

2b (3p)

Using the real dependencies, perform a BCNF decomposition.

- Show the steps that you perform, including the schema that results.
- Show the resulting tables with the original data in them, now represented in a less redundant form.

3. SQL queries (12p)

Assume tables with the following schemas for public transport:

```
  Stops (name, location)
  Lines (number, vehicle)
  StopsOnLines (line, stop, timeFromStart)
    line -> Lines(number)
    stop -> Stops(name)
```

(primary key information left out as irrelevant here).

The `StopsOnLines` table lists stops along each line, each with the total time it takes to reach them from the start of the line. For instance: ”line 13 takes 5 minutes to Chalmers”, ”line 13 takes 21 minutes to Central Station”. Your task is to write some SQL queries for such questions.

3a (3p)

Find all lines that stop at Chalmers, together with the vehicles they use. The result should look as follows:

```
  line | vehicle
 ------+-----------
  8    | tram
  16   | bus
```

3b (4p)

Find all connections from Chalmers to Brunnsparken that require exactly one change (on a stop common to two lines). The result should show the two lines involved, the stop where the change happens, and the total duration (ignoring the waiting time at the change place). The connections should not be optimal, just possible. The result should look as follows:

```
 line | line |        change        | duration
------+------+----------------------+----------
 8    | 2    | Marklandsgatan       |       28
 8    | 2    | Korsvägen            |       13
```

3c (5p)

Classify stops into cities on the basis of their location. We assume just two cities: Gothenburg and Mölndal. The locations are expressed by codes such as `G123e4` and `M36g89` where the first letter indicates the city. The query should return all stops names in alphabetical order together with the city they are in. The result should look as follows:

```
          stop          |    city
------------------------+------------
 Almedal                | Gothenburg
 Krokslätts torg        | Mölndal
 Kungsportsplatsen      | Gothenburg
```

4. Relational algebra and semantics (8p)

The following SQL query finds those stops names that exist in more than one different locations:

```
SELECT DISTINCT A.name
FROM Stops A, Stops B
WHERE A.name = B.name AND A.location <> B.location
```

Notice that the result will always be empty if the constraints in Question 1 are obeyed. However, in a larger setting we may have to relax these constraints and permit the same stop names in different cities. For example, the Gothenburg region has Almedal in both Gothenburg and Kungsbacka.

4a (2p)

Translate the above query to relational algebra.

4b (2p)

Trace the execution of this query by showing the number of rows and columns of each intermediate relation that is formed if the query is executed without any optimizations. You can assume that there are 10,000 rows in Stops and 50 names in the end result.

4c (4p)

Show an alternative query that generates the same result but without forming any intermediate relations with more than 10,000 rows. Show this query both in SQL (2p) and in relational algebra (2p).

5. Constraints and triggers (12p)

5a (2p)

In question 2, we showed a table with tram models M28 and M31, as well as bus models B9S and 7900. Write and SQL `CREATE TABLE` definition for the schema (with key information intentionally left out—it is up to you to add any)

```
Vehicles (vehicletype, model, capacity)
```

adding constraints that guarantee that models M28 and M31 can only be trams, and models B9S and 7900 can only be buses. The attribute `vehicletype` has values `'bus'`, `'tram'`, etc.

5b (3p)

In question 1b, we defined the stops along a line in terms of the absolute travel time from the start of the line. An alternative description is to give them ordering numbers (1,2,3,...), and define the time (the number of minutes) it takes to travel to each stop from the *previous* one. The schema of such a table is:

```
StopsAndIntervals (_line, _ordering,stop,timeFromPrevious)
```

A meaningful constraint for this schema is that the time from the previous stop is 0 if and only if the stop is the first stop (stop number 1) on the line. Express this constraint in SQL.

5c (3p)

Create a view listing the times when different lines stop at different stops:

```
StopTimes(line,stop,time)
```

based on the tables

- `StopsOnLines(line,stop,timeFromStart)` that shows the number of minutes needed to reach each stop on the line (from question 3)
- `Runs(line,startTime)` that lists all the runs of a line with the times they start, e.g. that line 4 starts at 17:10 (from its first stop)

You can use the `+` operator to add durations to times: e.g. `13:16 + 55` gives the result `14:11`. (The syntax in PostgreSQL is a bit more complicated, but in this question you are free to use this simplified syntax.)

5d (4p)

Write a trigger for scheduling stop times as defined in 5c. This trigger should interpret each addition to the view `StopTimes` as an addition to the table `Runs`, creating an entry that starts just the exact time to reach the wanted stop at the wanted time.

Note: such updates would enable the user to add transports freely, almost like calling a taxi. Certainly too generous for a public transport system, but simple enough to use as an example of a trigger!

6. Remaining topics: XML (8p)

In this question, we will investigate the use of XML in both of its double purposes: data and documents. We build a document structure that enables the display of complex routes in a transport system. An example is the following route found from Västtrafik's web service, when asking how to get from Almedal to Idala:

dep.	17:02	Almedal	Göteborg	tram 2
arr.	17:12	Mölndal Station	Mölndal	
dep.	17:23	Mölndal station	Mölndal	train 3069
arr.	17:41	Kungsbacka station	Kungsbacka	
dep.	17:47	Kungsbacka station	Kungsbacka	bus 744
arr.	18:14	Idala	Kungsbacka	

6a (5p)

Write a DTD that models routes like this, with the following properties

- a route is a sequence of alternating departures and arrivals
- the sequence can be of any length, but must start with a departure and end with an arrival
- each departure indicates a time, a station, a city, and a line
- each arrival indicates a time, a station, and a city (the line is not mentioned, since it is the same as in the preceding departure)

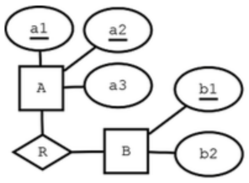
Every piece of information in the table (stop,city,line,time) must be represented as a distinct element or attribute value in the DTD.

6b (3p)

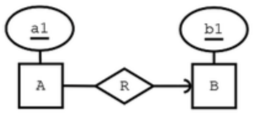
Write an XML element encoding the above route from Almedal to Idala. It must be valid according to your DTD.

Databases in a Nutshell (“Standard Cheatsheet”)

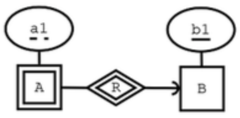
E-R diagrams and database schemas



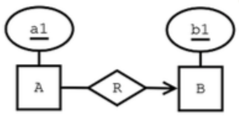
```
A(a1,a2,a3)
B(b1,b2)
R(a1,a2,b)
    (a1,a2)  -> A.(a1,a2)
    b        -> B.b1
```



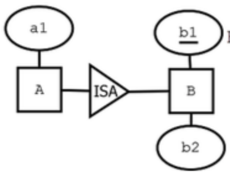
```
A(a1,b)
    b -> B.b1
B(b1)
```



```
A(a1,b)
    b -> B.b1
B(b1)
```



```
ER-Approach:
 A(a1)              Null Approach:
 B(b1)               A(a1,b (or Null))
 R(a,b)                 b -> B.b1
   a -> A.a1         B(b1)
   b -> B.b1
```



```
                     Null-Approach:
ER-Approach:          B(b1,b2,a1(or Null))
 B(b1,b2)
 A(b1,a1)            OO-Approach:
   b1 -> B.b1         A(b1,b2,a1)
                      B(b1,b2)
```

Functional dependencies

Definition (tuple, attribute, value). A **tuple** has the form

$$\{A_1 = v_1, \ldots, A_n = v_n\}$$

where $A_1, \ldots, A_n$ are **attributes** and $v_1, \ldots, v_n$ are their **values**.
Definition (signature, relation). The **signature** of a tuple, S, is the set of all its attributes, $\{A_1, \ldots, A_n\}$. A **relation** R of signature S is a set of tuples with signature S. But we will sometimes also say ”relation” when we mean the signature itself.
Definition (projection). If t is a tuple of a relation with signature S, the **projection** $t.A_i$ computes to the value v_i.
Definition (simultaneous projection). If X is a set of attributes $\{B_1, \ldots, B_m\} \subseteq S$ and t is a tuple of a relation with signature S, we can form a simultaneous projection,

$$t.X = \{B_1 = t.B_1, \ldots, B_m = t.B_m\}$$

Definition (functional dependency, FD). Assume X is a set of attributes and A an attribute, all belonging to a signature S. Then A is **functionally dependent** on X in the relation R, written $X \to A$, if

- for all tuples t,u in R, if $t.X = u.X$ then $t.A = u.A$.

If Y is a set of attributes, we write $X \to Y$ to mean that $X \to A$ for every A in Y.
Definition (multivalued dependency, MVD). Let X,Y,Z be disjoint subsets of a signature S such that $S = X \cup Y \cup Z$. Then Y has a **multivalued dependency** on X in R, written $X \twoheadrightarrow Y$, if

- for all tuples t,u in R, if $t.X = u.X$ then there is a tuple v in R such that
  - $v.X = t.X$
  - $v.Y = t.Y$

  – $v.Z = u.Z$

Definition. An attribute A **follows** from a set of attributes Y, if there is an FD $X \to A$ such that $X \subseteq Y$.
Definition (closure of a set of attributes under FDs). The **closure** of a set of attributes $X \subseteq S$ under a set FD of functional dependencies, denoted $X+$, is the set of those attributes that follow from X.
Definition (trivial functional dependencies). An FD $X \to A$ is **trivial**, if $A \in X$.
Definition (superkey, key). A set of attributes $X \subseteq S$ is a **superkey** of S, if $S \subseteq X+$.
A set of attributes $X \subseteq S$ is a **key** of S if

- X is a superkey of S
- no proper subset of X is a superkey of S

Definition (Boyce-Codd Normal Form, BCNF violation). A functional dependency $X \to A$ **violates BCNF** if

- X is not a superkey
- the dependency is not trivial

A relation **is in Boyce-Codd Normal Form** (BCNF) if it has no BCNF violations.
Definition (prime). An attribute A is prime if it belongs to some key.
Definition (Third Normal Form, 3NF violation). A functional dependency $X \to A$ **violates 3NF** if

- X is not a superkey
- the dependency is not trivial
- A is not prime

Definition (trivial multivalued dependency). A multivalued dependency $X \twoheadrightarrow A$ is trivial if $Y \subseteq X$ or $X \cup Y = S$.
Definition (Fourth Normal Form, 4NF violation). A multivalued dependency $X \twoheadrightarrow A$ **violates 4NF** if

- X is not a superkey
- the MVD is not trivial.

Algorithm (BCNF decomposition). Consider a relation R with signature S and a set F of functional dependencies. R can be brought to BCNF by the following steps:

1. If R has no BCNF violations, return R
2. If R has a violating functional dependency $X \to A$, decompose R to two relations
   - R_1 with signature $X \cup \{A\}$
   - R_2 with signature $S - \{A\}$
3. Apply the above steps to R_1 and R_2 with functional dependencies projected to the attributes contained in each of them.

Algorithm (4NF decomposition). Consider a relation R with signature S and a set M of multivalued dependencies. R can be brought to 4NF by the following steps:

1. If R has no 4NF violations, return R
2. If R has a violating multivalued dependency $X \twoheadrightarrow Y$, decompose R to two relations
   - R_1 with signature $X \cup \{Y\}$
   - R_2 with signature $S - Y$
3. Apply the above steps to *R1* and *R2*

Concept (minimal basis of a set of functional dependencies; not a rigorous definition). A **minimal basis** of a set F of functional dependencies is a set F- that implies all dependencies in F. It is obtained by first weakening the left hand sides and then dropping out dependencies that follow by transitivity. Weakening an LHS in $X \to A$ means finding a minimal subset of X such that A can still be derived from F-.
Algorithm (3NF decomposition). Consider a relation R with a set F of functional dependencies.

1. If R has no 3NF violations, return R.
2. If R has 3NF violations,
   - compute a minimal basis of F- of F
   - group F- by the left hand side, i.e. so that all depenencies $X \to A$ are grouped together
   - for each of the groups, return the schema $XA_1 \ldots A_n$ with the common LHS and all the RHSs
   - if one of the schemas contains a key of R, these groups are enough; otherwise, add a schema containing just some key

Relational algebra

relation ::=	
relname	**name of relation (can be used alone)**
| $\sigma_{\text{condition}}$ relation	**selection (sigma)** `WHERE`
| $\pi_{\text{projection+}}$ relation	**projection (pi)** `SELECT`
| $\rho_{\text{relname (attribute+)?}}$ relation	**renaming (rho)** `AS`
| $\gamma_{\text{attribute*,aggregationexp+}}$ relation	**grouping (gamma)** `GROUP BY, HAVING`
| $\tau_{\text{expression+}}$ relation	**sorting (tau)** `ORDER BY`
| δ relation	**removing duplicates (delta)** `DISTINCT`
| relation $\times$ relation	**cartesian product** `FROM, CROSS JOIN`
| relation $\cup$ relation	**union** `UNION`
| relation $\cap$ relation	**intersection** `INTERSECT`
| relation $-$ relation	**difference** `EXCEPT`
| relation $\bowtie$ relation	`NATURAL JOIN`
| relation $\bowtie_{\text{condition}}$ relation	**theta join** `JOIN ON`
| relation $\bowtie_{\text{attribute+}}$ relation	`INNER JOIN`
| relation $\bowtie^{o}_{\text{attribute+}}$ relation	`FULL OUTER JOIN`
| relation $\bowtie^{oL}_{\text{attribute+}}$ relation	`LEFT OUTER JOIN`
| relation $\bowtie^{oR}_{\text{attribute+}}$ relation	`RIGHT OUTER JOIN`
projection ::=	
expression	**expression, can be just an attribute**
| expression $\rightarrow$ attribute	**rename projected expression** `AS`
aggregationexp ::=	
aggregation(*|attribute)	**without renaming**
| aggregation(*|attribute) $\rightarrow$ attribute	**with renaming** `AS`
expression, condition, aggregation, attribute ::=	
as in SQL, but excluding subqueries	

SQL

```
statement ::=
      CREATE TABLE tablename (
    * attribute type inlineconstraint*
    * [CONSTRAINT name]? constraint deferrable?
    ) ;
  |
      DROP TABLE tablename ;
  |
      INSERT INTO tablename tableplaces? values ;
  |
      DELETE FROM tablename
    ? WHERE condition  ;
  |
      UPDATE tablename
      SET setting+
    ? WHERE condition ;
  |
      query ;
  |
      CREATE VIEW viewname
      AS ( query ) ;
  |
      ALTER TABLE tablename
   +  alteration ;
  |
      COPY tablename FROM filepath ;
        ## postgresql-specific, tab-separated

query ::=
      SELECT DISTINCT? columns
    ? FROM table+
    ? WHERE condition
    ? GROUP BY attribute+
    ? HAVING condition
    ? ORDER BY attributeorder+
  |
      query setoperation query
  |
      query ORDER BY attributeorder+
        ## no previous ORDER in query
  |
      WITH localdef+ query

table ::=
      tablename
  |   table AS? tablename  ## only one iteration allowed
  |   ( query ) AS? tablename
  |   table jointype JOIN table ON condition
  |   table jointype JOIN table USING (attribute+)
  |   table NATURAL jointype JOIN table

condition ::=
      expression comparison compared
  |   expression NOT? BETWEEN expression AND expression
  |   condition boolean condition
  |   expression NOT? LIKE 'pattern*'
  |   expression NOT? IN values
  |   NOT? EXISTS ( query )
  |   expression IS NOT? NULL
  |   NOT ( condition )
```

```
type ::=
     CHAR ( integer ) | VARCHAR ( integer ) | TEXT
   | INT | FLOAT

inlineconstraint ::=     ## not separated by commas!
     PRIMARY KEY
   | REFERENCES tablename ( attribute ) policy*
   | UNIQUE | NOT NULL
   | CHECK ( condition )
   | DEFAULT value

constraint ::=
     PRIMARY KEY ( attribute+ )
   | FOREIGN KEY ( attribute+ )
       REFERENCES tablename ( attribute+ ) policy*
   | UNIQUE ( attribute+ ) | NOT NULL ( attribute )
   | CHECK ( condition )

policy ::=
     ON DELETE|UPDATE CASCADE|SET NULL
deferrable ::=
     NOT? DEFERRABLE (INITIALLY DEFERRED|IMMEDIATE)?
tableplaces ::=
     ( attribute+ )

values ::=
     VALUES ( value+ )  ## VALUES only in INSERT
   | ( query )

setting ::=
     attribute = value

alteration ::=
     ADD COLUMN attribute type inlineconstraint*
   | DROP COLUMN attribute

localdef ::=
     WITH tablename AS ( query )

columns ::=
     *         ## literal asterisk
   | column+

column ::=
     expression
   | expression AS name

attributeorder ::=
     attribute (DESC|ASC)?

setoperation ::=
     UNION | INTERSECT | EXCEPT

jointype ::=
     LEFT|RIGHT|FULL OUTER?
   | INNER?

comparison ::=
     = | < | > | <> | <= | >=
expression ::=
      attribute
  |   tablename.attribute
  |   value
  |   expression operation expression
  |   aggregation ( DISTINCT? *|attribute)
  |   ( query )

value ::=
      integer | float | string ## string in single quotes
  |   value operation value
  |   NULL

boolean ::=
      AND | OR

## triggers

functiondefinition ::=
  CREATE FUNCTION functionname() RETURNS TRIGGER AS $$
  BEGIN
*   triggerstatement
  END
  $$ LANGUAGE 'plpgsql'
  ;

triggerdefinition ::=
  CREATE TRIGGER triggernane
    whentriggered
    FOR EACH ROW|STATEMENT
  ? WHEN ( condition )
    EXECUTE PROCEDURE functionname
    ;

whentriggered ::=
    BEFORE|AFTER events ON tablename
  | INSTEAD OF   events ON viewname

events ::= event | event OR events
event  ::= INSERT | UPDATE | DELETE

triggerstatement ::=
    IF ( condition ) THEN statement+ elsif* END IF ;
  | RAISE EXCEPTION 'message' ;
  | statement ;  ## INSERT, UPDATE or DELETE
  | RETURN NEW|OLD|NULL ;

elsif ::= ELSIF ( condition ) THEN statement+

compared ::=
     expression
   | ALL|ANY values

operation ::=
     "+" | "-" | "*" | "/" | "%"
   | "||"

pattern ::=
    % | _ | character  ## match any string/char
  | [ character* ]
  | [^ character* ]

aggregation ::=
    MAX | MIN | AVG | COUNT | SUM


## privileges

statement ::=
   GRANT  privilege+ ON object TO user+ grantoption?
 | REVOKE privilege+ ON object FROM user+ CASCADE?
 | REVOKE GRANT OPTION FOR privilege
     ON object FROM user+ CASCADE?
 | GRANT rolename TO username adminoption?

privilege ::=
   SELECT | INSERT | DELETE | UPDATE | REFERENCES
 | ALL PRIVILEGES ## | ...

object ::=
   tablename (attribute+)+ | viewname (attribute+)+
 | trigger ## | ...

user ::= username | rolename | PUBLIC

grantoption ::= WITH GRANT OPTION

adminoption ::= WITH ADMIN OPTION

## transactions

statement ::=
  START TRANSACTION mode* | BEGIN | COMMIT | ROLLBACK

mode ::=
   ISOLATION LEVEL level
 | READ WRITE | READ ONLY

level ::=
   SERIALIZABLE | REPEATABLE READ | READ COMMITTED
 | READ UNCOMMITTED

## indexes

statement ::=
   CREATE INDEX indexname ON tablename (attribute+)?
```

XML

```
document ::= header? dtd? element

header ::= "<?xml version=1.0 encoding=utf-8 standalone=no?>"
  ## standalone=no if with DTD

dtd ::= <! DOCTYPE ident [ definition* ]>

definition ::=
    <! ELEMENT ident rhs >
  | <! ATTLIST ident attribute* >

rhs ::=
   EMPTY | #PCDATA | ident
  | rhs"*" | rhs"+" | rhs"?"
  | rhs , rhs
  | rhs "|" rhs

attribute ::= ident type #REQUIRED|#IMPLIED

type ::= CDATA | ID | IDREF

element   ::= starttag element* endtag | emptytag
```

```
starttag  ::= < ident attr* >
endtag    ::= </ ident >
emptytag  ::= < ident attr* />

attr ::= ident = string ## string in double quotes

## XPath

path ::=
    axis item cond? path?
  | path "|" path

axis ::= / | //

item ::= "@"? (ident*) | ident :: ident

cond ::= [ exp op exp ] | [ integer ]

exp  ::=  "@"? ident | integer | string

op   ::= = | != | < | > | <= | >=
```

Grammar conventions

- CAPITAL words are SQL or XML keywords, to take literally
- small character words are names of syntactic categories, defined each in their own rules
- | separates alternatives
- `+` means one or more, separated by commas in SQL, by white space in XML
- `*` means zero or more, separated by commas in SQL, by white space in XML
- `?` means zero or one
- in the beginning of a line, `+ * ?` operate on the whole line; elsewhere, they operate on the word just before
- `##` start comments, which explain unexpected notation or behaviour
- text in double quotes means literal code, e.g. `"*"` means the operator `*`
- other symbols, e.g. parentheses, also mean literal code (quotes are used only in some cases, to separate code from grammar notation)
- parentheses can be added to disambiguate the scopes of operators, in both SQL and XML

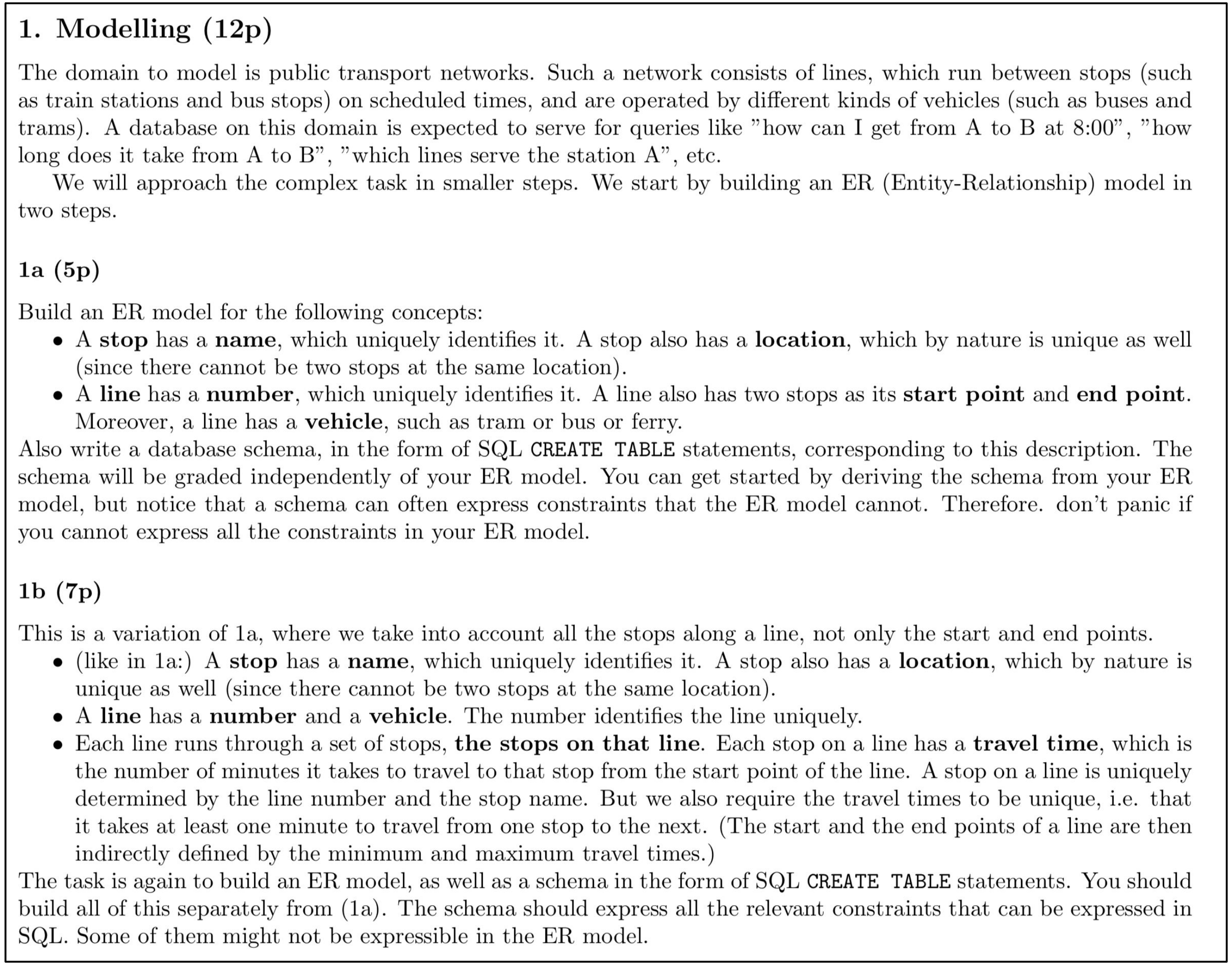
Databases Exam March 2018

TDA357 (Chalmers), DIT620 (University of Gothenburg)

Stepwise solutions

Aarne, Alejandro,...

1a (5p)



1. Modelling (12p)

The domain to model is public transport networks. Such a network consists of lines, which run between stops (such as train stations and bus stops) on scheduled times, and are operated by different kinds of vehicles (such as buses and trams). A database on this domain is expected to serve for queries like ”how can I get from A to B at 8:00”, ”how long does it take from A to B”, ”which lines serve the station A”, etc.

We will approach the complex task in smaller steps. We start by building an ER (Entity-Relationship) model in two steps.

1a (5p)

Build an ER model for the following concepts:

- A **stop** has a **name**, which uniquely identifies it. A stop also has a **location**, which by nature is unique as well (since there cannot be two stops at the same location).
- A **line** has a **number**, which uniquely identifies it. A line also has two stops as its **start point** and **end point**. Moreover, a line has a **vehicle**, such as tram or bus or ferry.

Also write a database schema, in the form of SQL `CREATE TABLE` statements, corresponding to this description. The schema will be graded independently of your ER model. You can get started by deriving the schema from your ER model, but notice that a schema can often express constraints that the ER model cannot. Therefore. don’t panic if you cannot express all the constraints in your ER model.

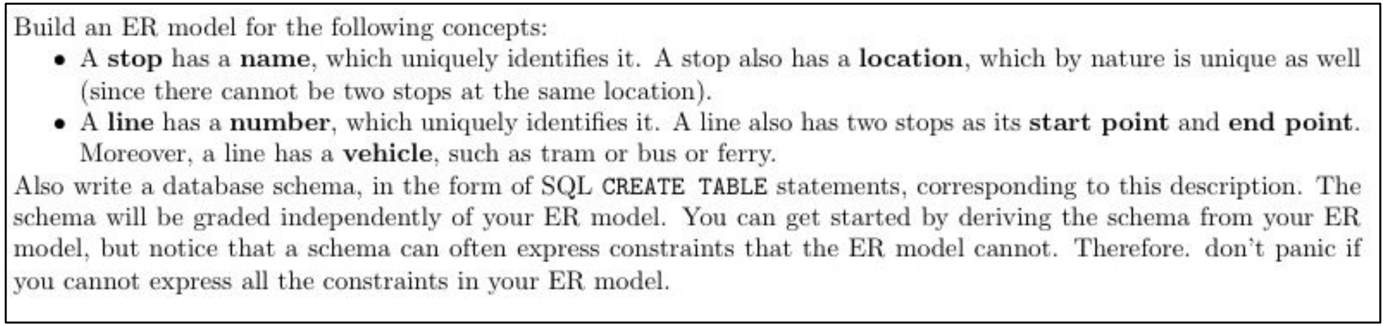
1b (7p)

This is a variation of 1a, where we take into account all the stops along a line, not only the start and end points.

- (like in 1a:) A **stop** has a **name**, which uniquely identifies it. A stop also has a **location**, which by nature is unique as well (since there cannot be two stops at the same location).
- A **line** has a **number** and a **vehicle**. The number identifies the line uniquely.
- Each line runs through a set of stops, **the stops on that line**. Each stop on a line has a **travel time**, which is the number of minutes it takes to travel to that stop from the start point of the line. A stop on a line is uniquely determined by the line number and the stop name. But we also require the travel times to be unique, i.e. that it takes at least one minute to travel from one stop to the next. (The start and the end points of a line are then indirectly defined by the minimum and maximum travel times.)

The task is again to build an ER model, as well as a schema in the form of SQL `CREATE TABLE` statements. You should build all of this separately from (1a). The schema should express all the relevant constraints that can be expressed in SQL. Some of them might not be expressible in the ER model.

1a (5p)

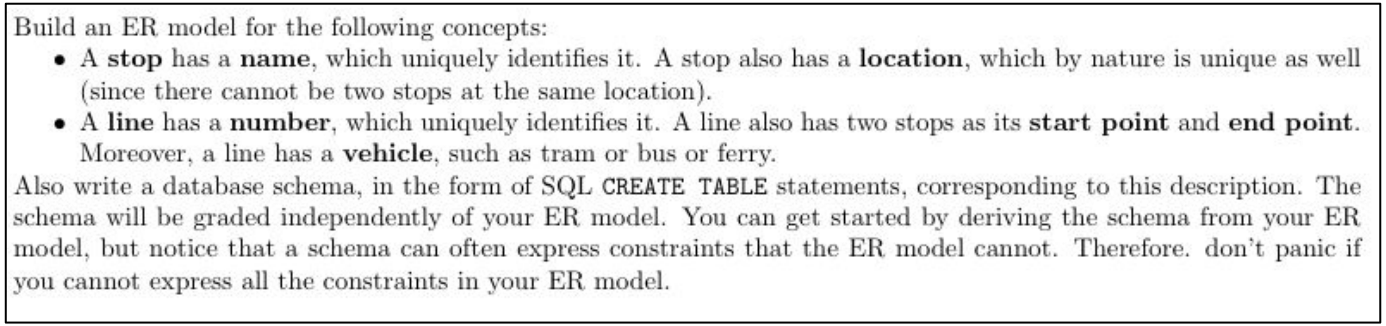


Build an ER model for the following concepts:

- A **stop** has a **name**, which uniquely identifies it. A stop also has a **location**, which by nature is unique as well (since there cannot be two stops at the same location).
- A **line** has a **number**, which uniquely identifies it. A line also has two stops as its **start point** and **end point**. Moreover, a line has a **vehicle**, such as tram or bus or ferry.

Also write a database schema, in the form of SQL `CREATE TABLE` statements, corresponding to this description. The schema will be graded independently of your ER model. You can get started by deriving the schema from your ER model, but notice that a schema can often express constraints that the ER model cannot. Therefore. don't panic if you cannot express all the constraints in your ER model.

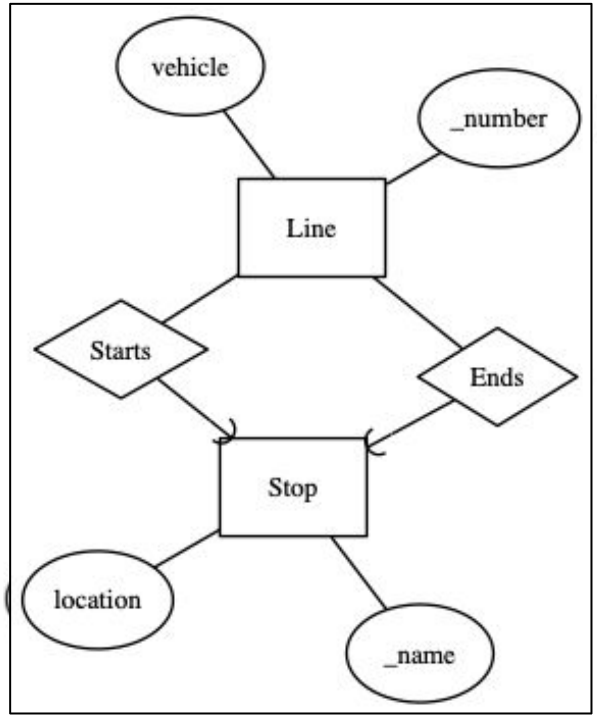
1a (5p)



Build an ER model for the following concepts:

- A **stop** has a **name**, which uniquely identifies it. A stop also has a **location**, which by nature is unique as well (since there cannot be two stops at the same location).
- A **line** has a **number**, which uniquely identifies it. A line also has two stops as its **start point** and **end point**. Moreover, a line has a **vehicle**, such as tram or bus or ferry.

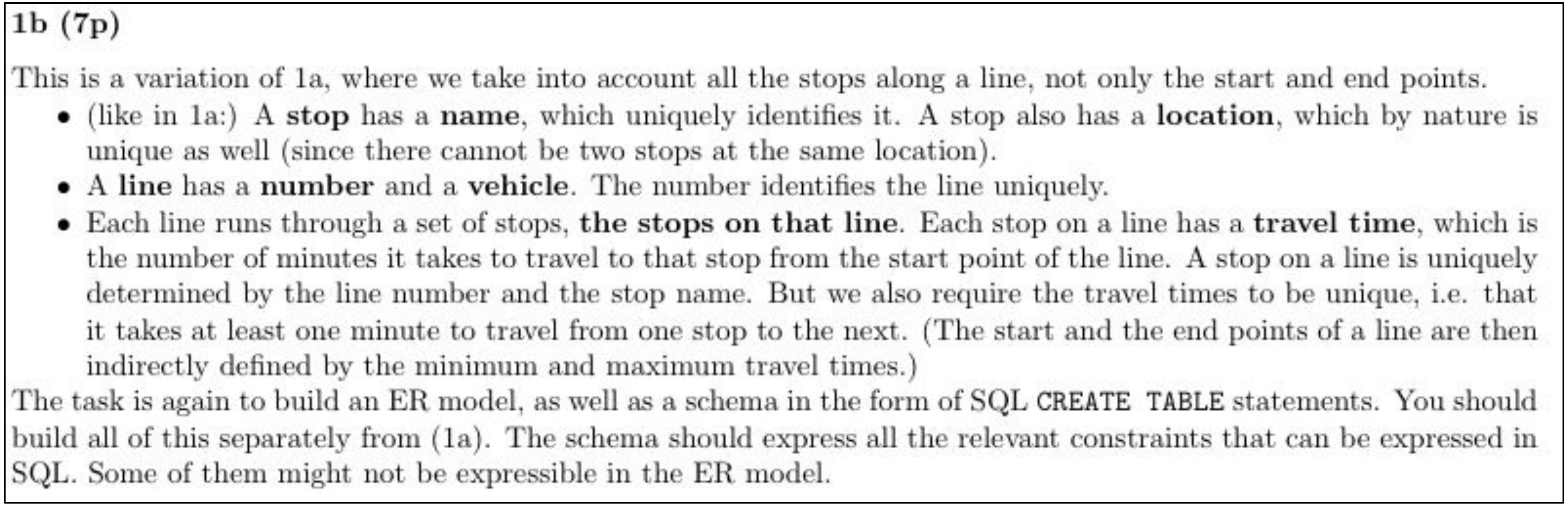
Also write a database schema, in the form of SQL `CREATE TABLE` statements, corresponding to this description. The schema will be graded independently of your ER model. You can get started by deriving the schema from your ER model, but notice that a schema can often express constraints that the ER model cannot. Therefore, don't panic if you cannot express all the constraints in your ER model.



```
CREATE TABLE Stop(
  name     TEXT PRIMARY KEY,
  location TEXT UNIQUE
);

CREATE TABLE Line(
  number    TEXT PRIMARY KEY,
  vehicle   TEXT,
  starpoint TEXT REFERENCES Stop(name),
  endpoint  TEXT REFERENCES Stop(name)
);
```

1b (7p)



1b (7p)

This is a variation of 1a, where we take into account all the stops along a line, not only the start and end points.

- (like in 1a:) A **stop** has a **name**, which uniquely identifies it. A stop also has a **location**, which by nature is unique as well (since there cannot be two stops at the same location).
- A **line** has a **number** and a **vehicle**. The number identifies the line uniquely.
- Each line runs through a set of stops, **the stops on that line**. Each stop on a line has a **travel time**, which is the number of minutes it takes to travel to that stop from the start point of the line. A stop on a line is uniquely determined by the line number and the stop name. But we also require the travel times to be unique, i.e. that it takes at least one minute to travel from one stop to the next. (The start and the end points of a line are then indirectly defined by the minimum and maximum travel times.)

The task is again to build an ER model, as well as a schema in the form of SQL `CREATE TABLE` statements. You should build all of this separately from (1a). The schema should express all the relevant constraints that can be expressed in SQL. Some of them might not be expressible in the ER model.

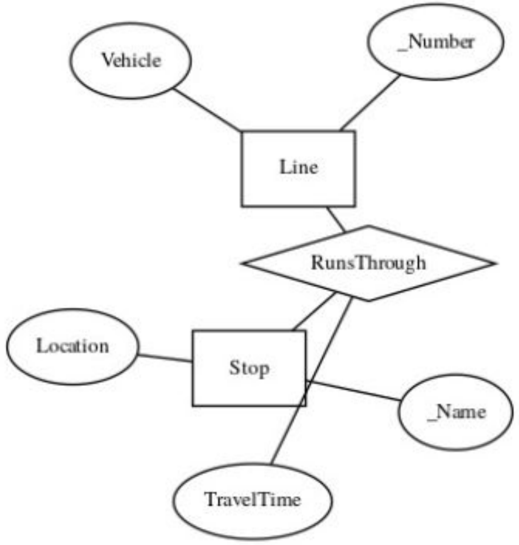
1b (7p)

1b (7p)

This is a variation of 1a, where we take into account all the stops along a line, not only the start and end points.

- (like in 1a:) A **stop** has a **name**, which uniquely identifies it. A stop also has a **location**, which by nature is unique as well (since there cannot be two stops at the same location).
- A **line** has a **number** and a **vehicle**. The number identifies the line uniquely.
- Each line runs through a set of stops, **the stops on that line**. Each stop on a line has a **travel time**, which is the number of minutes it takes to travel to that stop from the start point of the line. A stop on a line is uniquely determined by the line number and the stop name. But we also require the travel times to be unique, i.e. that it takes at least one minute to travel from one stop to the next. (The start and the end points of a line are then indirectly defined by the minimum and maximum travel times.)

The task is again to build an ER model, as well as a schema in the form of SQL `CREATE TABLE` statements. You should build all of this separately from (1a). The schema should express all the relevant constraints that can be expressed in SQL. Some of them might not be expressible in the ER model.

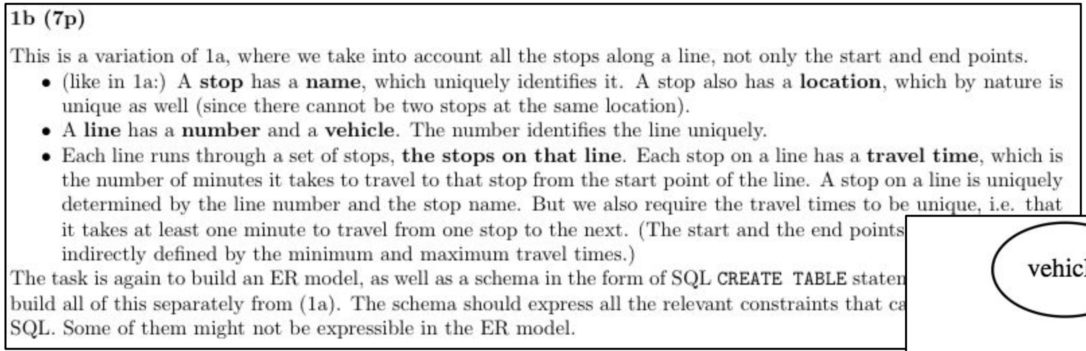


```
CREATE TABLE Stop(
  name     TEXT PRIMARY KEY,
  location TEXT UNIQUE
);

CREATE TABLE Line(
  number    TEXT PRIMARY KEY,
  vehicle   TEXT
);

CREATE RunsThrough(
  line      TEXT REFERENCES Line(number),
  stop      TEXT REFERENCES Stop(name),
  travetime INTEGER,
  PRIMARY KEY (line,stop),
  CONSTRAINT time_unique UNIQUE (line,stop,travetime)
);
```

1b (7p)

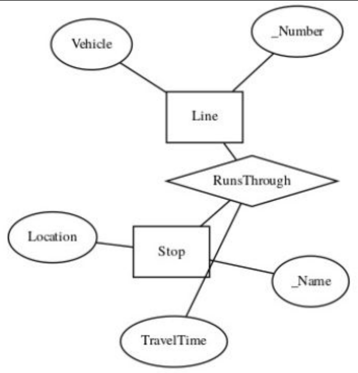


1b (7p)

This is a variation of 1a, where we take into account all the stops along a line, not only the start and end points.

- (like in 1a:) A **stop** has a **name**, which uniquely identifies it. A stop also has a **location**, which by nature is unique as well (since there cannot be two stops at the same location).
- A **line** has a **number** and a **vehicle**. The number identifies the line uniquely.
- Each line runs through a set of stops, **the stops on that line**. Each stop on a line has a **travel time**, which is the number of minutes it takes to travel to that stop from the start point of the line. A stop on a line is uniquely determined by the line number and the stop name. But we also require the travel times to be unique, i.e. that it takes at least one minute to travel from one stop to the next. (The start and the end points
indirectly defined by the minimum and maximum travel times.)

The task is again to build an ER model, as well as a schema in the form of SQL `CREATE TABLE` staten
build all of this separately from (1a). The schema should express all the relevant constraints that ca
SQL. Some of them might not be expressible in the ER model.

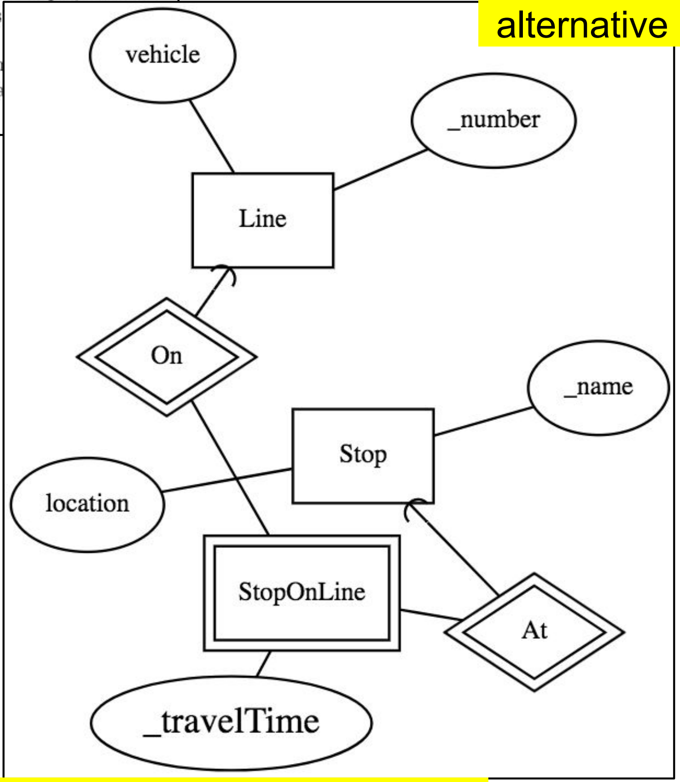


```
CREATE TABLE Stop(
  name     TEXT PRIMARY KEY,
  location TEXT UNIQUE
);

CREATE TABLE Line(
  number    TEXT PRIMARY KEY,
  vehicle   TEXT
);

CREATE RunsThrough(
  line      TEXT REFERENCES Line(number),
  stop      TEXT REFERENCES Stop(name),
  travetime INTEGER,
  PRIMARY KEY (line,stop),
  CONSTRAINT time_unique UNIQUE (line,stop,travetime)
);
```

alternative



Notice: none of the ER models in 1a,b is a precise expression of the UNIQUE constraints.

2. Dependencies (8p)

The following table shows a few connections from the public transport system of Gothenburg.

line	vehicle	model	start point	start city	end point	end city	start time	capacity
2	tram	M28	AxelDs torg	Göteborg	Mölndal	Mölndal	16:26	116
2	tram	M31	AxelDs torg	Göteborg	Mölndal	Mölndal	17:16	202
4	tram	M31	Mölndal	Mölndal	Angered	Göteborg	16:26	202
16	bus	B9S	Fyrktorget	Göteborg	Eketrägatan	Göteborg	12:44	138
55	bus	7900	SvenHs plats	Göteborg	Lindholmen	Göteborg	10:05	105

2a (5p)

You can find many redundancies in this table, and your task is to identify their causes and eliminate them. More precisely:

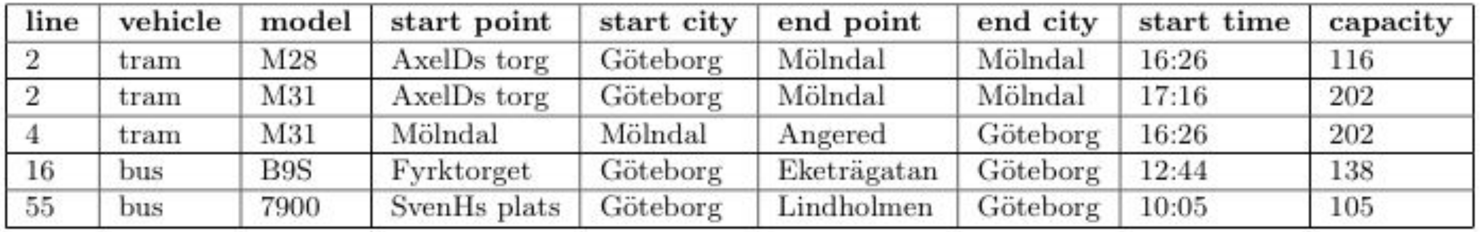
- Show the functional dependencies that hold for this table (the basic ones; you can leave out derived ones).
  - Start by following the definition of a functional dependency (in the "standard cheat sheet") and applying it to the table.
  - List all the dependencies found in this way (but you can safely leave out derived dependencies).
  - This list will probably contain some bogus dependencies that cease to hold when you add more data, as well as some real dependencies.
  - Explain why you consider some dependencies bogus and leave them out, e.g. by giving examples of realistic data that would destroy those dependencies.

2b (3p)

Using the real dependencies, perform a BCNF decomposition.

- Show the steps that you perform, including the schema that results.
- Show the resulting tables with the original data in them, now represented in a less redundant form.

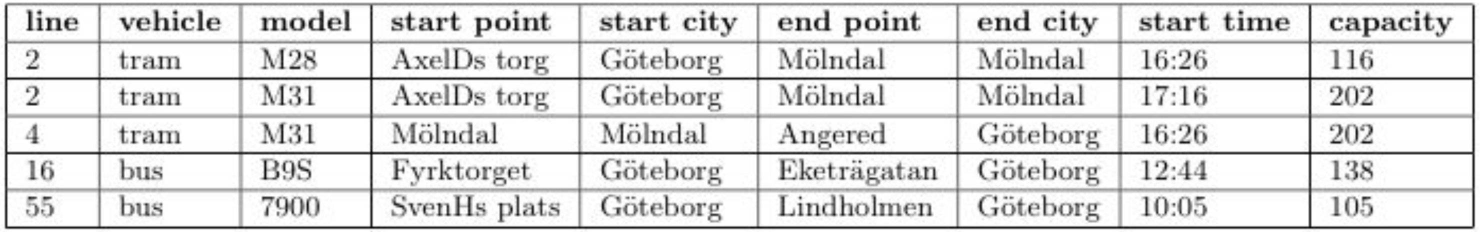
2a (5p)



line	vehicle	model	start point	start city	end point	end city	start time	capacity
2	tram	M28	AxelDs torg	Göteborg	Mölndal	Mölndal	16:26	116
2	tram	M31	AxelDs torg	Göteborg	Mölndal	Mölndal	17:16	202
4	tram	M31	Mölndal	Mölndal	Angered	Göteborg	16:26	202
16	bus	B9S	Fyrktorget	Göteborg	Eketrägatan	Göteborg	12:44	138
55	bus	7900	SvenHs plats	Göteborg	Lindholmen	Göteborg	10:05	105

startpoint -> startcity
endpoint -> endcity
model -> vehicle capacity
line -> vehicle startpoint endpoint
line starttime -> model
capacity -> vehicle model
starttime -> vehicle
starttime capacity -> line startpoint endpoint
endcity capacity -> line startpoint endpoint starttime
endcity starttime -> line model startpoint startcity endpoint capacity
endpoint -> line vehicle startpoint startcity
endpoint capacity -> starttime
endpoint starttime -> model capacity
startcity capacity -> line startpoint endpoint endcity starttime
startcity starttime -> line model startpoint endpoint endcity capacity vehicle
startpoint -> line vehicle endpoint
startpoint capacity -> starttime
startpoint starttime -> model capacity
model starttime -> line startpoint startcity endpoint
model endcity -> line startpoint endpoint starttime
model endpoint -> starttime
model startcity -> line startpoint endpoint starttime
model startpoint -> starttime
vehicle endcity -> startcity
vehicle startcity -> endcity
line capacity -> starttime
line model -> starttime

2a (5p)



line	vehicle	model	start point	start city	end point	end city	start time	capacity
2	tram	M28	AxelDs torg	Göteborg	Mölndal	Mölndal	16:26	116
2	tram	M31	AxelDs torg	Göteborg	Mölndal	Mölndal	17:16	202
4	tram	M31	Mölndal	Mölndal	Angered	Göteborg	16:26	202
16	bus	B9S	Fyrktorget	Göteborg	Eketrägatan	Göteborg	12:44	138
55	bus	7900	SvenHs plats	Göteborg	Lindholmen	Göteborg	10:05	105

Real FDs

startpoint -> startcity
endpoint -> endcity
model -> vehicle capacity
line -> vehicle startpoint endpoint
line starttime -> model

Key:
line starttime

Bogus FDs (no need to list all of these - just a few is enough)

capacity -> vehicle model (*explanation: many models could have the same capacity*)
starttime -> vehicle (*many vehicles could start at the same time*)
starttime capacity -> line startpoint endpoint (*many vehicles of same capacity could start at same time*)
endcity capacity -> line startpoint endpoint starttime (... easy to invent counterexamples to each ...)
endcity starttime -> line model startpoint startcity endpoint capacity
endpoint -> line vehicle startpoint startcity
endpoint capacity -> starttime
endpoint starttime -> model capacity
startcity capacity -> line startpoint endpoint endcity starttime
startcity starttime -> line model startpoint endpoint endcity capacity vehicle
startpoint -> line vehicle endpoint
startpoint capacity -> starttime
startpoint starttime -> model capacity
model starttime -> line startpoint startcity endpoint
model endcity -> line startpoint endpoint starttime
model endpoint -> starttime
model startcity -> line startpoint endpoint starttime
model startpoint -> starttime
vehicle endcity -> startcity
vehicle startcity -> endcity
line capacity -> starttime
line model -> starttime

2b (3p)

```
R: line vehicle startpoint startcity endpoint endcity starttime model capacity

FD:
startpoint -> startcity
endpoint -> endcity
model -> vehicle capacity
line -> vehicle startpoint endpoint
line starttime -> model
```

2b (3p)

```
R: line vehicle startpoint startcity endpoint endcity starttime model capacity

FD:
startpoint -> startcity
endpoint -> endcity
model -> vehicle capacity
line -> vehicle startpoint endpoint
line starttime -> model

violation: line -> vehicle startpoint startcity endpoint endcity
Decomposition:
  R1: line vehicle startpoint startcity endpoint endcity









  R2: model capacity starttime line
```

2b (3p)

```
R: line vehicle startpoint startcity endpoint endcity starttime model capacity

FD:
startpoint -> startcity
endpoint -> endcity
model -> vehicle capacity
line -> vehicle startpoint endpoint
line starttime -> model

violation: line -> vehicle startpoint startcity endpoint endcity
Decomposition:
  R1: line vehicle startpoint startcity endpoint endcity
  Violation: endpoint -> endcity
  Decomposition:
    R11: endpoint endcity
    R12: line vehicle startpoint startcity endpoint
    Violation: startpoint -> startcity
    Decomposition:
      R121: startpoint startcity
      R122: line vehicle startpoint endpoint
  R2: model capacity starttime line
  Violation: model -> capacity
  Decomposition:
    R21: model capacity
    R22: line starttime model
```

2b (3p)

```
R11: endpoint endcity
R121: startpoint startcity
R122: line vehicle startpoint endpoint
R21: model capacity
R22: line starttime model
```

endpoint	endcity
Mölndal	Mölndal
Angered	Göteborg
Eketrägatan	Göteborg
Lindholmen	Göteborg

startpoint	startcity
Mölndal	Mölndal
AxelDs torg	Göteborg
Fyrktorget	Göteborg
SvenHs plats	Göteborg

line	vehicle	startpoint	endpoint
2	tram	AxelDs torg	Mölndal
4	tram	Mölndal	Angered
16	bus	Fyrktorget	Eketrägata n
55	bus	SvenHs plats	Lindholmen

model	capacity
M28	116
M31	202
B9S	138
7900	105

line	starttime	model
2	16:26	M28
2	17:16	M31
4	16:26	M31
16	12:44	B9S
55	10:05	7900

3. SQL queries (12p)

Assume tables with the following schemas for public transport:

```
Stops (name, location)
Lines (number, vehicle)
StopsOnLines (line, stop, timeFromStart)
  line -> Lines(number)
  stop -> Stops(name)
```

(primary key information left out as irrelevant here).

The `StopsOnLines` table lists stops along each line, each with the total time it takes to reach them from the start of the line. For instance: ”line 13 takes 5 minutes to Chalmers”, ”line 13 takes 21 minutes to Central Station”. Your task is to write some SQL queries for such questions.

```
Stops (name, location)
Lines (number, vehicle)
StopsOnLines (line, stop, timeFromStart)
  line -> Lines(number)
  stop -> Stops(name)
```

3a (3p)

Find all lines that stop at Chalmers, together with the vehicles they use. The result should look as follows:

```
 line | vehicle
------+-----------
 8    | tram
 16   | bus
```

```
Stops (name, location)
Lines (number, vehicle)
StopsOnLines (line, stop, timeFromStart)
  line -> Lines(number)
  stop -> Stops(name)
```

3a (3p)

Find all lines that stop at Chalmers, together with the vehicles they use. The result should look as follows:

```
 line | vehicle
------+-----------
 8    | tram
 16   | bus
```

```
SELECT line, vehicle
FROM StopsOnLines, Lines
WHERE stop = 'Chalmers' AND line = Lines.number
```

3b (4p)

Find all connections from Chalmers to Brunnsparken that require exactly one change (on a stop common to two lines). The result should show the two lines involved, the stop where the change happens, and the total duration (ignoring the waiting time at the change place). The connections should not be optimal, just possible. The result should look as follows:

```
 line | line |         change          | duration
------+------+-------------------------+----------
 8    | 2    | Marklandsgatan          |       28
 8    | 2    | Korsvägen               |       13
```

3b (4p)

Find all connections from Chalmers to Brunnsparken that require exactly one change (on a stop common to two lines). The result should show the two lines involved, the stop where the change happens, and the total duration (ignoring the waiting time at the change place). The connections should not be optimal, just possible. The result should look as follows:

```
 line | line |        change        | duration
------+------+----------------------+----------
 8    | 2    | Marklandsgatan       |       28
 8    | 2    | Korsvägen            |       13
```

```
WITH Distances AS (
  SELECT A.stop AS startPoint, B.stop AS endPoint, A.line,
         (B.timeFromStart - A.timeFromStart)) AS minutes
  FROM StopsOnLines A, StopsOnLines B
  WHERE A.line = B.line AND B.timeFromStart > A.timeFromStart
  )
  SELECT A.line, B.line, A.endPoint AS change, A.minutes + B.minutes AS duration
  FROM Distances A, Distances B
  WHERE A.startPoint='Chalmers' AND A.endPoint=B.startPoint AND B.endPoint='Brunnsparken'
```

3c (5p)

Classify stops into cities on the basis of their location. We assume just two cities: Gothenburg and Mölndal. The locations are expressed by codes such as `G123e4` and `M36g89` where the first letter indicates the city. The query should return all stops names in alphabetical order together with the city they are in. The result should look as follows:

```
          stop           |    city
-------------------------+------------
 Almedal                 | Gothenburg
 Krokslätts torg         | Mölndal
 Kungsportsplatsen       | Gothenburg
```

3c (5p)

Classify stops into cities on the basis of their location. We assume just two cities: Gothenburg and Mölndal. The locations are expressed by codes such as `G123e4` and `M36g89` where the first letter indicates the city. The query should return all stops names in alphabetical order together with the city they are in. The result should look as follows:

```
         stop            |    city
-------------------------+------------
 Almedal                 | Gothenburg
 Krokslätts torg         | Mölndal
 Kungsportsplatsen       | Gothenburg
```

```
(SELECT name AS stop , 'Gothenburg' AS city
 FROM Stops
 WHERE location LIKE 'G%'
   UNION
 SELECT name AS stop, 'Mölndal' AS city
 FROM Stops
 WHERE location LIKE 'M%'
) ORDER BY name
```

4. Relational algebra and semantics (8p)

The following SQL query finds those stops names that exist in more than one different locations:

```
SELECT DISTINCT A.name
FROM Stops A, Stops B
WHERE A.name = B.name AND A.location <> B.location
```

Notice that the result will always be empty if the constraints in Question 1 are obeyed. However, in a larger setting we may have to relax these constraints and permit the same stop names in different cities. For example, the Gothenburg region has Almedal in both Gothenburg and Kungsbacka.

4a (2p)

Translate the above query to relational algebra.

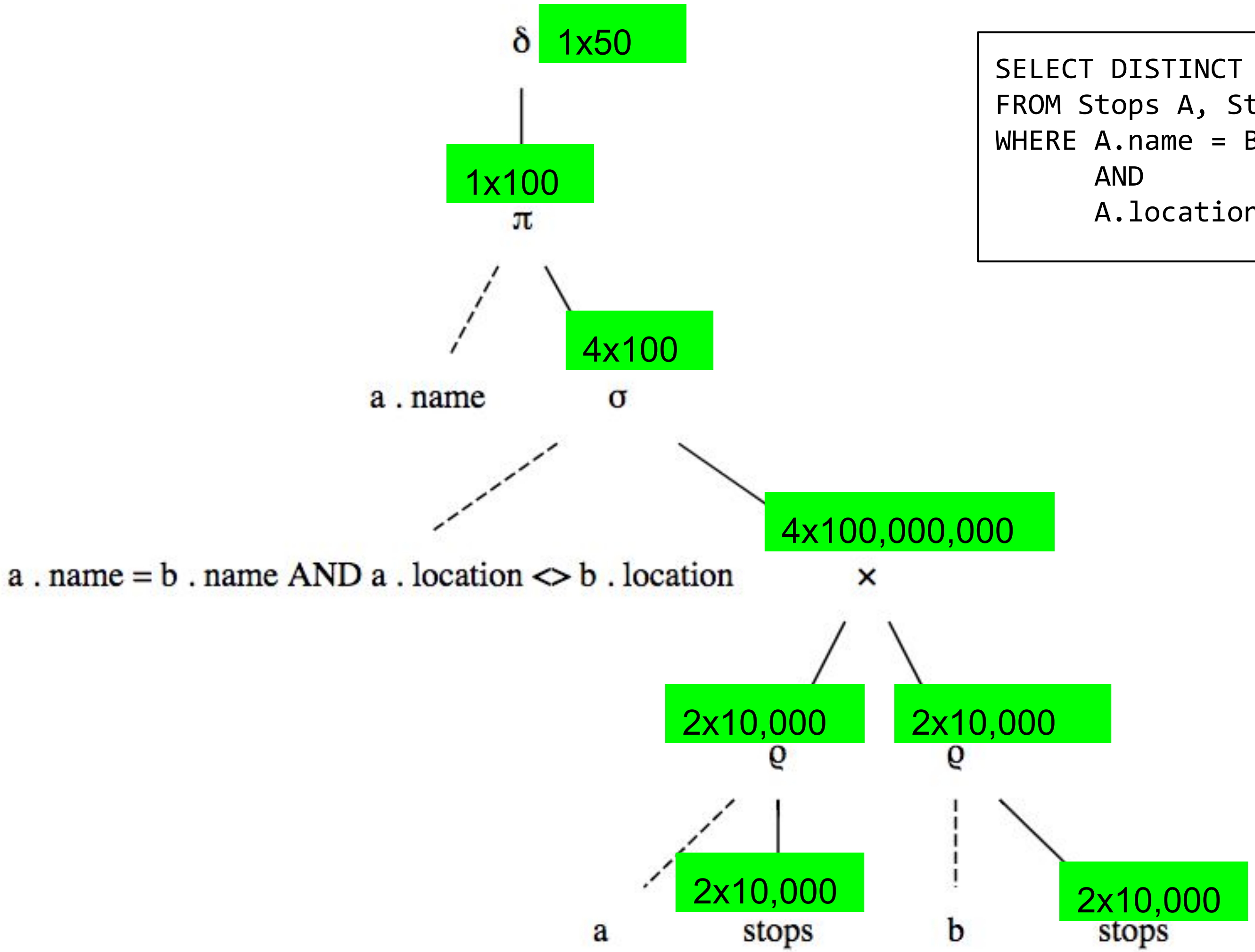
4b (2p)

Trace the execution of this query by showing the number of rows and columns of each intermediate relation that is formed if the query is executed without any optimizations. You can assume that there are 10,000 rows in Stops and 50 names in the end result.

4c (4p)

Show an alternative query that generates the same result but without forming any intermediate relations with more than 10,000 rows. Show this query both in SQL (2p) and in relational algebra (2p).

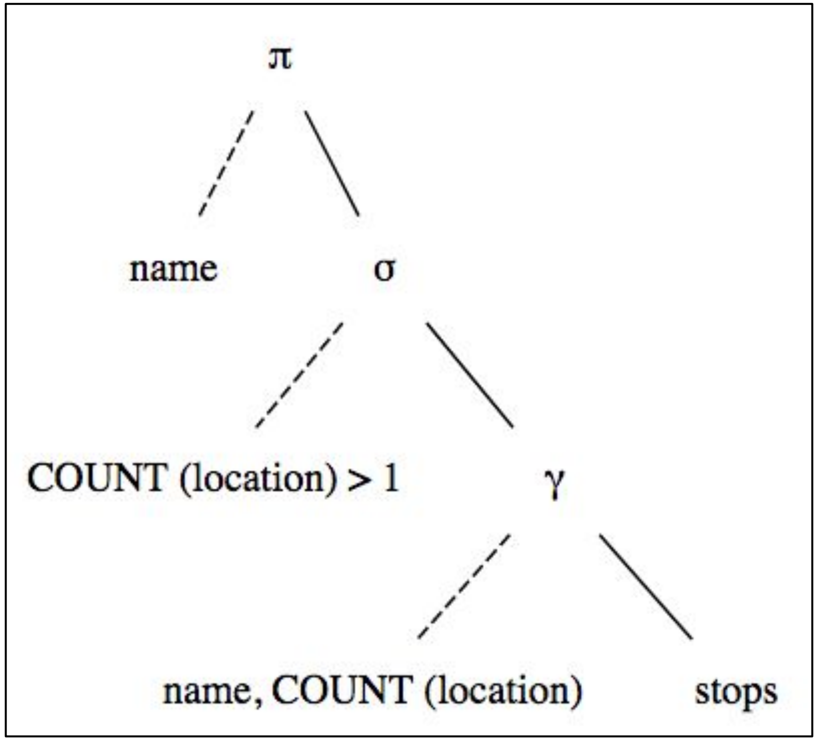
4a
4b



```
SELECT DISTINCT A.name
FROM Stops A, Stops B
WHERE A.name = B.name
      AND
      A.location <> B.location
```

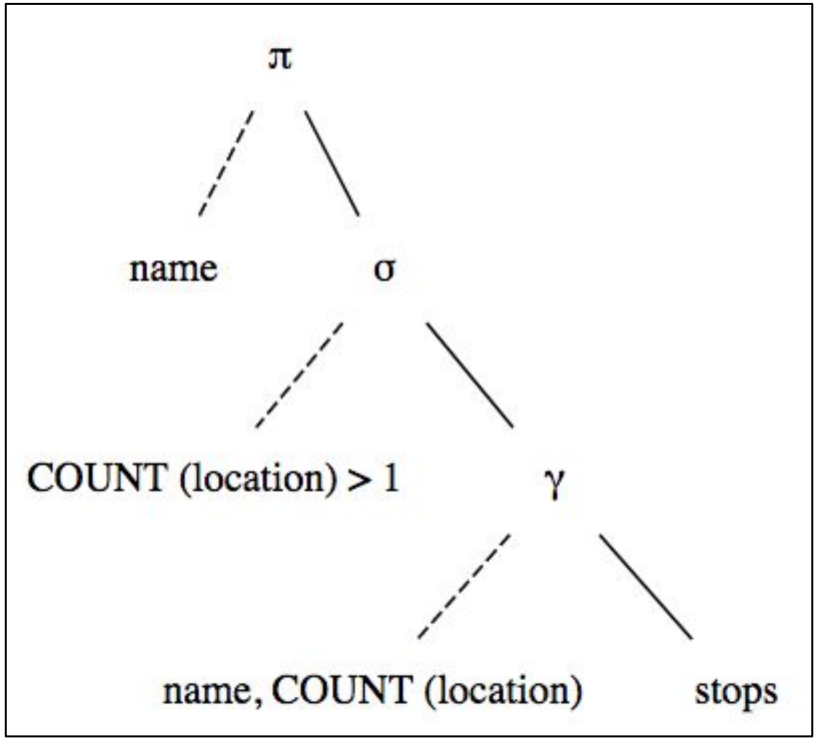
4c

```
SELECT name
FROM stops
GROUP BY name
HAVING COUNT (location)> 1
```



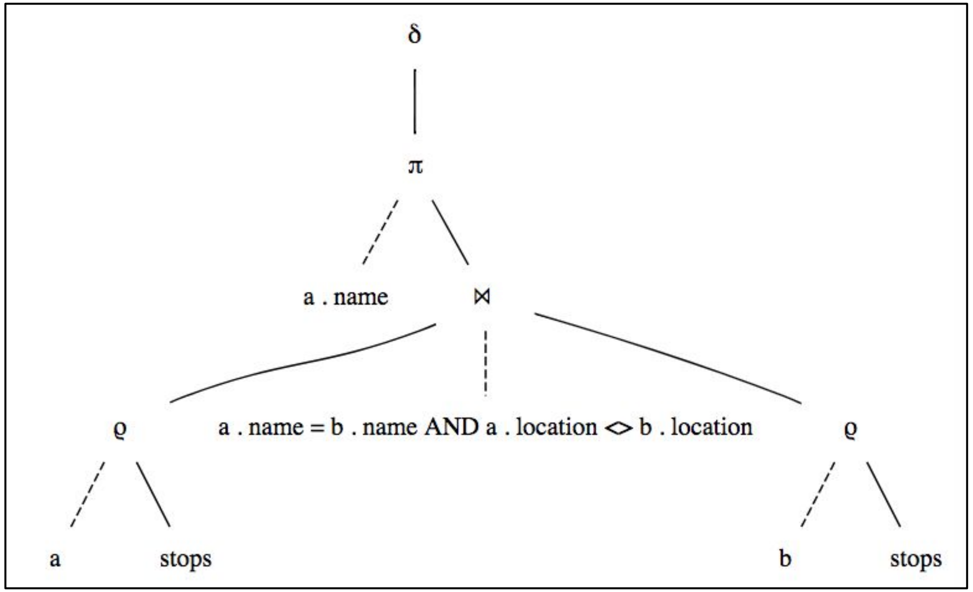
4c

```
SELECT name
FROM stops
GROUP BY name
HAVING COUNT (location)> 1
```



```
-- alternative; avoids the big table
but may need more comparisons

SELECT DISTINCT A.name
FROM Stops A INNER JOIN Stops B ON
  (A.name = B.name
     AND
   A.location <> B.location)
```



5. Constraints and triggers (12p)

5a (2p)

In question 2, we showed a table with tram models M28 and M31, as well as bus models B9S and 7900. Write and SQL `CREATE TABLE` definition for the schema (with key information intentionally left out—it is up to you to add any)

```
  Vehicles (vehicletype, model, capacity)
```

adding constraints that guarantee that models M28 and M31 can only be trams, and models B9S and 7900 can only be buses. The attribute `vehicletype` has values `'bus'`, `'tram'`, etc.

5. Constraints and triggers (12p)

5a (2p)

In question 2, we showed a table with tram models M28 and M31, as well as bus models B9S and 7900. Write and SQL `CREATE TABLE` definition for the schema (with key information intentionally left out—it is up to you to add any)

```
Vehicles (vehicletype, model, capacity)
```

adding constraints that guarantee that models M28 and M31 can only be trams, and models B9S and 7900 can only be buses. The attribute `vehicletype` has values `'bus'`, `'tram'`, etc.

```
CREATE TABLE Vehicles (
  vehicletype TEXT,
  model TEXT,
  capacity INT,
  CHECK (model NOT IN ('M28','M31') OR vehicletype = 'tram'),
  CHECK (model NOT IN ('B9S','7900') OR vehicletype = 'bus')
  ) ;
-- the constraints express “if A then B” by “not A or B”
```

5b (3p)

In question 1b, we defined the stops along a line in terms of the absolute travel time from the start of the line. An alternative description is to give them ordering numbers (1,2,3,...), and define the time (the number of minutes) it takes to travel to each stop from the *previous* one. The schema of such a table is:

```
StopsAndIntervals (_line, _ordering,stop,timeFromPrevious)
```

A meaningful constraint for this schema is that the time from the previous stop is 0 if and only if the stop is the first stop (stop number 1) on the line. Express this constraint in SQL.

5b (3p)

In question 1b, we defined the stops along a line in terms of the absolute travel time from the start of the line. An alternative description is to give them ordering numbers (1,2,3,...), and define the time (the number of minutes) it takes to travel to each stop from the *previous* one. The schema of such a table is:

```
StopsAndIntervals (_line, _ordering,stop,timeFromPrevious)
```

A meaningful constraint for this schema is that the time from the previous stop is 0 if and only if the stop is the first stop (stop number 1) on the line. Express this constraint in SQL.

```
CREATE TABLE StopsAndIntervals (
  line TEXT REFERENCES Lines(number),
  stop TEXT REFERENCES Stops(name),
  ordering INT,
  timeFromPrevious INT,
  PRIMARY KEY (line,ordering),
  CHECK (ordering <> 1 OR timeFromPrevious = 0),
  CHECK (timeFromPrevious <> 0 OR ordering = 1)
  ) ;
-- the same logic as in 5a, for “if A then B” and “if B then A”
```

5c (3p)

Create a view listing the times when different lines stop at different stops:

```
StopTimes(line,stop,time)
```

based on the tables

- `StopsOnLines(line,stop,timeFromStart)` that shows the number of minutes needed to reach each stop on the line (from question 3)
- `Runs(line,startTime)` that lists all the runs of a line with the times they start, e.g. that line 4 starts at 17:10 (from its first stop)

You can use the `+` operator to add durations to times: e.g. `13:16 + 55` gives the result `14:11`. (The syntax in PostgreSQL is a bit more complicated, but in this question you are free to use this simplified syntax.)

5c (3p)

Create a view listing the times when different lines stop at different stops:

```
StopTimes(line,stop,time)
```

based on the tables

- `StopsOnLines(line,stop,timeFromStart)` that shows the number of minutes needed to reach each stop on the line (from question 3)
- `Runs(line,startTime)` that lists all the runs of a line with the times they start, e.g. that line 4 starts at 17:10 (from its first stop)

You can use the `+` operator to add durations to times: e.g. `13:16 + 55` gives the result `14:11`. (The syntax in PostgreSQL is a bit more complicated, but in this question you are free to use this simplified syntax.)

```
CREATE VIEW StopTimes AS (
  SELECT Runs.line AS line, stop, startTime + timeFromStart AS time
  FROM StopsOnLines, Runs
  WHERE
    StopsOnLines.line = Runs.line
  ) ;
```

5d (4p)

Write a trigger for scheduling stop times as defined in 5c. This trigger should interpret each addition to the view `StopTimes` as an addition to the table `Runs`, creating an entry that starts just the exact time to reach the wanted stop at the wanted time.

Note: such updates would enable the user to add transports freely, almost like calling a taxi. Certainly too generous for a public transport system, but simple enough to use as an example of a trigger!

5d (4p)

Write a trigger for scheduling stop times as defined in 5c. This trigger should interpret each addition to the view `StopTimes` as an addition to the table `Runs`, creating an entry that starts just the exact time to reach the wanted stop at the wanted time.

Note: such updates would enable the user to add transports freely, almost like calling a taxi. Certainly too generous for a public transport system, but simple enough to use as an example of a trigger!

```
CREATE FUNCTION scheduleStopTime () RETURNS TRIGGER AS $$
            BEGIN
              INSERT INTO Runs VALUES (
                NEW.line,
                NEW.time - (SELECT timeFromStart FROM StopsOnLines S WHERE NEW.stop = S.stop)
                ) ;
              RETURN NEW ;
            END
            $$ LANGUAGE plpgsql ;
CREATE TRIGGER scheduleStopTimeTrigger
              INSTEAD OF INSERT ON StopTimes
              FOR EACH ROW
              EXECUTE PROCEDURE scheduleStopTime () ;
```

Question 6: Semi-structured data and other topics (10 p, 3+4+3)

Read all parts of the question before you start, the answers are not independent.

In this task you will store restaurant menus in a semi-structured data format.

Menus have dishes with prices, divided into categories that may further be divided into subcategories.

Here is an example menu with 3 categories (Starters, Salads and Burgers) and Burgers contains an additional category (Vegetarian burgers).

```
Starters
  Calamari          $8.50
Salads
  Caesar            $8.50
  Chicken           $9.25
Burgers
  Standard          $9
  Bacon             $10
  Vegetarian burgers
    Haloumi         $12
    Mushroom        $10
```

a) Encode the menu above in a JSON document (OR in XML if you prefer). Note that the order of items in the menu is important. Prices are all numbers (the $ does not need to be included).

Hint: Perhaps the root of the JSON document should not be an object?

This question is from the January 2019 exam, which is the first one using JSON instead of XML.

Question 6: Semi-structured data and other topics (10 p, 3+4+3)

Read all parts of the question before you start, the answers are not independent.

In this task you will store restaurant menus in a semi-structured data format.

Menus have dishes with prices, divided into categories that may further be divided into subcategories.

Here is an example menu with 3 categories (Starters, Salads and Burgers) and Burgers contains an additional category (Vegetarian burgers).

```
Starters
  Calamari            $8.50
Salads
  Caesar              $8.50
  Chicken             $9.25
Burgers
  Standard            $9
  Bacon               $10
  Vegetarian burgers
    Haloumi           $12
    Mushroom          $10
```

a) Encode the menu above in a JSON document (OR in XML if you prefer). Note that the order of items in the menu is important. Prices are all numbers (the $ does not need to be included).

Hint: Perhaps the root of the JSON document should not be an object?

```
[
  {"category":"Starters",
   "contents":[
     {"dish":"Calamari", "price":8.50}
   ]
  },
  {"category":"Salads",
   "contents":[
     {"dish":"Caesar",  "price":8.50},
     {"dish":"Chicken", "price":9.25}
   ]
  },
  {"category":"Burgers",
   "contents":[
     {"dish":"Standard", "price":9},
     {"dish":"Bacon",    "price":10},
     {"category":"Vegetarian Burgers",
       "contents":[
         {"dish":"Haloumi",  "price":12},
         {"dish":"Mushroom", "price":10}
       ]
     }
   ]
  }
]
```

b) Write a JSON Schema for menus (OR a XML DTD if you use XML). It is OK if your schema allows extra data (like descriptions of categories or ingredients of dishes), but it should not allow things like a category having a price or a dish containing other dishes. You should allow arbitrarily deep nesting of categories (i.e. categories in categories in categories...).

b) Write a JSON Schema for menus (OR a XML DTD if you use XML). It is OK if your schema allows extra data (like descriptions of categories or ingredients of dishes), but it should not allow things like a category having a price or a dish containing other dishes. You should allow arbitrarily deep nesting of categories (i.e. categories in categories in categories...).

```
{"type":"array",
 "items":{
   "type":"object",
   "oneOf":[
     { "properties":{
         "category":{"type":"string"},
         "contents":{"$ref":"#"},
         "dish":false,
         "price":false
         },
       "required":["category","contents"]
     },
     { "properties":{
         "category":false,
         "contents":false,
         "dish":{"type":"string"},
         "price":{"type":"number"}
         },
         "required":["dish","price"]
     }]
 }
}
```

c) Write a JSON Path expression (OR an XML Path expression if you use XML) for finding the prices of all burgers (in a menu valid in your schema, and structured similarly to the example above with burgers as a top level category). In this example the result would be 9, 10, 12, 10 (order is not important).

c) Write a JSON Path expression (OR an XML Path expression if you use XML) for finding the prices of all burgers (in a menu valid in your schema, and structured similarly to the example above with burgers as a top level category). In this example the result would be 9, 10, 12, 10 (order is not important).

```
$[?(@.category=="Burgers")]..price
```