Exam 3

- ER-to-relational mapping problem points
  - derived attributes
    - these are computed from other things, not stored outright → not included directly in the relational schema (one might create a view including them)
  - weak entity types
    - the weak entity type becomes a relation with its own attributes plus any attributes of the identifying relationship(s) plus the PK of the entity type(s) on the other end of the identifying relationship(s)
      - the PK for the weak entity type's relation is its own partial key plus the PK of the entity type(s) on the other end of the identifying relationship(s)

```
ROOM(room_type,capacity,room_number,hotel)
  ROOM.hotel → HOTEL.id
```

Exam 3

- ER-to-relational mapping problem points
  - relationships
    - to capture cardinality and participation constraints, the FK approach is preferred for 1:N relationships – only use the separate relation approach for N:M
    - include NOT NULL constraints when there is total participation and that attribute is not already part of a PK
    - don't forget to include PKs and FKs

Exam 3

- ER-to-relational mapping problem points
  - specialization
    - there are several options for mapping specialization, but a reasonable one for a disjoint specialization where both supertype and subtypes have attributes is for every entity type to have its own relation
      - supertype → relation with its own attributes
      - subtype → relation with its own attributes plus the PK of the supertype as an FK

```
PROPERTY(id,address,rent_amount)

HOUSE(id,num_bedrooms,num_bath)
  HOUSE.id → PROPERTY.id

APARTMENT(id,num_bedrooms,unit_number,floor)
  APARTMENT.id → PROPERTY.id
```

Exam 3

- ER-to-relational mapping problem points
  - categories
    - both the category type itself and the types belonging to that category become relations
    - add a *surrogate key* to the category type if it doesn't already have a PK
    - add that surrogate key (or the category type's PK) to each of the types belonging to that category as an FK

```
PAYMENT(id,date,amount)
  tenant NOT NULL
  PAYMENT.tenant → TENANT.id

CREDIT_CARD(number,name,expiration,cvv,payment)
  CREDIT_CARD.payment → PAYMENT.id

PAYPAL(transaction_id,payment)
  PAYPAL.payment → PAYMENT.id
```

Exam 3

First Normal Form (1NF)
- all values are atomic – no composite attributes
- each tuple has a single value for each attribute (can be NULL) – no multivalued attributes
- every relation has a primary key

- don't forget the last part – having a PK
  - address all three points when explaining why a relational schema is in 1NF
    - (atomic and single-valued are givens – there's no notation indicating that because attributes in a relational schema are those things)

Exam 3

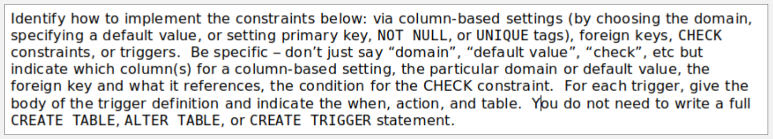
Second Normal Form (2NF)
- satisfies 1NF
- for all FDs X → A, either A has no non-key attributes or X is not a proper subset of a key

- 2NF eliminates dependencies on partial keys
  - a dependency on a non-key (such as `salesperson → commission`) is *not* a problem here
- if multiple attributes are underlined, all of them together constitute the PK

```
CAR_SALE(vin,make,model,year,color,date_sold,sale_price,
         salesperson,commission,buyer_name,buyer_address)
```

  - `vin → make, model, year, color` depends on a partial key – violates 2NF

Exam 3


Identify how to implement the constraints below: via column-based settings (by choosing the domain, specifying a default value, or setting primary key, `NOT NULL`, or `UNIQUE` tags), foreign keys, `CHECK` constraints, or triggers. Be specific – don't just say "domain", "default value", "check", etc but indicate which column(s) for a column-based setting, the particular domain or default value, the foreign key and what it references, the condition for the `CHECK` constraint. For each trigger, give the body of the trigger definition and indicate the when, action, and table. You do not need to write a full `CREATE TABLE`, `ALTER TABLE`, or `CREATE TRIGGER` statement.

- what to write –
  - for a domain constraint, give the actual data type
    - e.g. `TINYINT` or `DECIMAL(4,2)`
  - for a foreign key, give the actual foreign key
    - e.g. `DISTRIBUTION.habitat → HABITAT.id`
  - for a CHECK constraint, give the actual condition
    - e.g. `CHECK ( area > 0 )`
  - for a trigger, identify the when, action, table elements and give the body of the trigger – you don't need the full CREATE TRIGGER header
- including more isn't a problem, but not including enough is

Exam 3

- #4
  - (a) "is a number 1, 2, 3, …"
    - account for both that these are integers and that negative and 0 values are excluded
      - a domain constraint is necessary for the integer part and can accommodate "not negative" but isn't quite sufficient for also handling "not zero"
    - "there are at most a few dozen regions" factors into the most appropriate integer data type (`TINYINT`)
  - (b) the constraint here is that DISTRIBUTION.area is a number with two decimal places and is non-zero
    - "if there is no habitat of some type in a particular region, there should not be an entry in DISTRIBUTION" is not an additional constraint – it is the rationale for why area is constrained to be non-zero (if you can't put a 0 area value into DISTRIBUTION, you can't put an entry in DISTRIBUTION where the area should be 0)
    - `DECIMAL(m,n)` means that there are *m* digits total and *n* digits after the decimal place – so `DECIMAL(5,2)` means values up to 999.99 are possible

Exam 3

- #4
  - (c) “the total area for different habitat types within a region cannot exceed the area of the region itself”
    - this must be a trigger because the total area for the different habitat types within a region requires summing multiple rows in DISTRIBUTION and then that value must be compared to a value from REGION
    - be careful to account for all of the situations where this constraint needs to be checked
      - INSERT and UPDATE in DISTRIBUTION could affect the total area for habitat types for a region – increasing the total area could violate the constraint
        - » DELETE in DISTRIBUTION also does, but it can only decrease the total which won’t violate the constraint if it wasn’t already violated
      - UPDATE in REGION could affect the region’s area – decreasing the region’s area could violate the constraint
        - » INSERT in REGION isn’t a problem because of the FK DISTRIBUTION.region → REGION.number – there can’t be any entries in DISTRIBUTION for that region if the region is only just being added
        - » DELETE in REGION is handled by the FK ON DELETE settings – since DISTRIBUTION.region is part of the PK and thus NOT NULL, a DELETE in REGION would need to be blocked if there are entries in DISTRIBUTION for that region

Exam 3

- #4
  - (c) “the total area for different habitat types within a region cannot exceed the area of the region itself”
    - when summing rows in DISTRIBUTION to find the total area for the region, be careful to take into account the INSERT/UPDATE operation itself
      - for INSERT, there is an area value for the new row which is not yet in DISTRIBUTION – must add the new value into the sum from DISTRIBUTION
      - for UPDATE, there is already an area value for the region in DISTRIBUTION but the update may include a different value – must subtract the old value and add the new value from the sum for DISTRIBUTION
    - remember `NEW.col` and `OLD.col` to refer to columns in the row involved in the INSERT/UPDATE/DELETE operation

Exam 4

- procedures vs functions
  - functions cannot change data and return exactly one value
  - functions can signal errors (as can procedures)
  - prefer functions if functions are applicable – use a procedure if you have to, and a function otherwise
- returning values
  - functions return their one value using a RETURN statement
    - the caller can utilize the returned value in an expression
  - for procedures, use OUT parameters if there is a fixed number (> 1) of values returned
    - the caller can capture these values in variables so subsequent SQL statements can utilize them
  - unbound SELECTs (a SELECT without INTO) “return” a result set, but this result set is only available to an application, not subsequent SQL statements

Exam 4

- calling routines
  - function calls are expressions – use where a value is needed

```
SELECT *
FROM SAILOR S
WHERE numReservations(S.Sid) >= 2

SELECT S.Sid,S.Sname,numReservations(S.Sid)
FROM SAILOR S
```

  - procedures are called with a CALL statement
    - variables must be provided for OUT parameters

```
CALL addReservation(22,103,'2024-12-06');

CALL findDateRange(22,@startdate,@enddate);
SELECT @startdate, @enddate;
SELECT B.Bid,B.Bname,B.Color
FROM BOAT B NATURAL JOIN RESERVATION R
WHERE R.date >= @startdate AND R.date <= @enddate;
```

Exam 4

- transactions are needed to ensure atomicity and serializability when there are multiple steps in an operation and
  - (atomicity) it's possible for one or more steps to fail while others succeed
  - (serializability) changes to the database between those steps could result in inconsistent results

Exam 4

- #3 – the procedure needs to:
  - check if the existing flight number exists and the new flight number does not – query FLIGHT
  - insert a row into FLIGHT with the new flight number and arrival and departure times from the parameters and the origin, destination, and miles from the existing row for the existing flight
    - can either use SELECT … INTO … to retrieve the existing info into variables for a subsequent INSERT, or do everything in one step with INSERT … SELECT
  - insert a row into SEATS for each row of SEATS involving the existing flight number
    - while it is possible to do this in one step with INSERT … SELECT, the problem specified that you must use a CURSOR

Exam 4

- #5
  - a) "find the destination airports for which the cheapest route from ROC …"
    - this is a query about airports, so the FROM table should be AIRPORT – "destination" is referring to the usage of `cheapestCost(origin,dest)` which returns the cost of the cheapest route from origin to dest
      - though, because there won't be route at all to an airport that isn't the destination for some flight in FLIGHT, it is possible to use FLIGHT instead – SELECT DISTINCT would then be required
    - `cheapestCost(origin,dest)` returns the cost of the cheapest route – pick those airports for which the cost from ROC is more than $400
  - b)
    - also display the destination and length of flight – `longestFlightUnder` "returns" that info, but stored in variables, so SELECT is needed to display the values

Exam 4

- #6
  - (a) transactions not needed
    - this can be done with a single UPDATE statement (every statement is automatically its own transaction)
  - (b) needs transactions
    - this involves three steps – if reservations are made (for example) between steps, the numbers reported may not be consistent with a single state of the database
      - there may not be harm in inconsistent numbers, but that's an issue of whether serializability is important for this application, not whether transactions are needed to enforce serializability
  - (c) needs transactions
    - this involves multiple steps – checking for a route (where "cheapest *available* fare" implies also checking for availability of seats) and then making potentially multiple reservations for the flight(s) involved
      - need to ensure the last available seat isn't taken between finding the route and making the reservations and/or that if making one of the reservations fails (e.g. because the last available seat was taken), the customer isn't left with an incomplete set of reservations
      -