Revision

COMP9311 24T3; Week 10

By Zhengyi Yang, UNSW

Details of Final Exam:

1. **Time: 1:50 PM to 4:00 PM, Sydney time, 28th Nov 2024** (2 hours + 10 minutes)
2. **Central Management**: Managed and invigilated by UNSW. **Please check your timetable for the location.**
3. **If you do not feel well on the exam day, please do not attend the exam.** By sitting for or submitting an exam or assessment on the scheduled date, you are declaring that you are fit to do so and cannot later apply for Special Consideration (i.e., no supplementary exam will be given).

My Experience Survey

The UNSW MyExperience survey for Term 3 2024 is still open, and with a response rate of less than 20%, your participation is highly encouraged: https://myexperience.unsw.edu.au/

Consultation:

We will run daily consultations (both in-person and online) before the final exam:

- **Dates:** Every day from 18th-22nd and 25th-27th November
- **Location:** Room K17 203
- **Times:** 9:00 AM - 11:00 AM and 1:00 PM - 3:00 PM

Question Types:

- **6 questions**, similar to those in the assignments and sample exam paper.
- A sample exam paper and additional practice questions have been released on WebCMS.

T/F Questions

- **Scoring:**
  - +1 mark for each correct answer
  - -1 mark for each incorrect answer (minimum score is 0)
  - 0 marks for unanswered questions

- **Example Question:**
  - SQL is a standard language for storing, manipulating, and retrieving data in databases.

Tips

- You can attempt the questions in any order, **arrange your time wisely**.
- Read all questions carefully
- Be fully prepared before the exam and relax yourself, no need to panic

Overview: Database Design 1

Data Models:

- ER, Relational Data Model and their mapping

Relational Algebra:

- Be able to use relational algebra to answer question.

Database Languages:

- SQL (not in exam)
- PL/pgSQL (not in exam)

Relational Database Design:

- Functional Dependency
- Normal Forms
- Design Algorithms for 3NF and BCNF

Overview: Database Design 2

Data Storage:

- ➢ Record Format
- ➢ Buffer Management

Query Optimisation (not in exam):

- ➢ Index
- ➢ Query Plan/Join Order Selection

Transaction Management:

- ➢ Concurrency Control
- ➢ Recovery

NoSQL :

- ➢ NoSQL Concept
- ➢ Different Data Model: Key-Value, Document, Column-family, Graph

Guest Lecture (not in exam)

Data Models

ER, Relational Data Model and their mapping

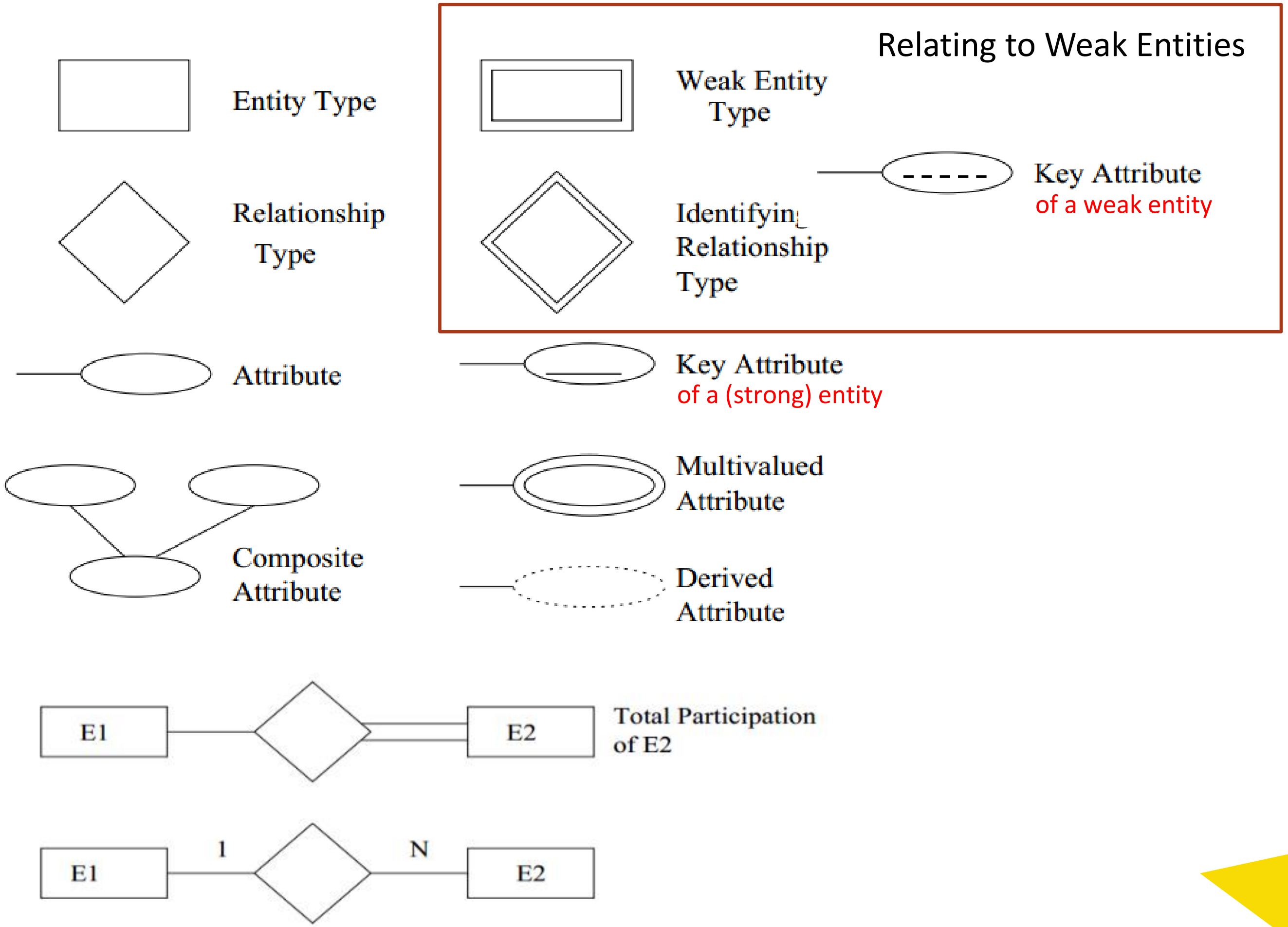
Application $\longrightarrow$ E-R $\longrightarrow$ Relational DB

Entity-Relationship Model

1. Entity Type: Group of object with the same properties
2. Entity: member of an entity type - analogous to an object.
3. Attribute: a property of object
4. Relationship: among objects

Notations

The notation used for ERDs is summarised in Elmasre/Navathe Figure 3.15.



Relational Data Model

- In the relational model, everything is described using relations.
- A relation can be thought of as a named table.
- Each column of the table corresponds to a named attribute.
- The set of allowed values for an attribute is called its domain.
- Each row of the table is called a tuple of the relation.
- N.B. There is no ordering of column or rows.

Keys

- *Keys* are used to identify tuples in a relation.
- A *superkey* is a set of attributes that uniquely determines a tuple.
- Note that this is a property of the relation that does not depend on the current relation instance.
- A *candidate key* is a superkey, none of whose *proper* subsets is a superkey.
- Keys are determined by the applications.

Integrity Constraints

There are several kinds of integrity constraints that are an integral part of the relational model:

1. *Key constraint:* candidate key values must be unique for every relation instance.
2. *Entity integrity:* an attribute that is part of a primary key cannot be NULL.
3. *Referential integrity:* The third kind has to do with “foreign keys”.

Foreign Keys

Foreign keys are used to refer to a tuple in another relation.

A set, FK, of attributes from a relation schema R1 may be a foreign key if

- the attributes have the same domains as the attributes in the primary key of another relation schema R2, and
- a value of FK in a tuple t1 of R1 either occurs as a value of PK for some tuple t2 in R2 or is null.

Referential integrity: The value of FK must occur in the other relation or be entirely NULL.

ER to Relational Model Mapping

One technique for database design is to first design a conceptual schema using a high-level data model, and then map it to a conceptual schema in the DBMS data model for the chosen DBMS.

Here we looked at a way to do this mapping from the ER to the relational data model. (see details in the lecture notes of Relational Data Model).

Composite and multivalued attributes are allowed in ER model, but not allowed in relational data model

Relational Algebra

Relational Algebra is a procedural data manipulation language (**DML**).

It specifies operations on relations to define new relations:

- **Unary Relational Operations**: Select, Project
- **Operations from Set Theory**: Union, Intersection, Difference, Cartesian Product
- **Binary Relational Operations**: Join, Divide.

Relational Algebra: be able to use relational algebra to answer question.

OPERATION	PURPOSE	NOTATION
SELECT	Selects all tuples that satisfy the selection condition from a relation R	$\sigma_{<selection\ condition>}(R)$
PROJECT	Produces a new relation with only some of the attributes of R and removes duplicate tuples.	$\pi_{<attribute\ list>}(R)$
THETA-JOIN	Produces all combinations of tuples from R and S that satisfy the join condition.	$R \bowtie_{<join\ condition>} S$
EQUI-JOIN	Produces all the combinations of tuples from R and S that satisfy a join condition with only equality comparisons.	$R \bowtie_{<join\ condition>} S$
NATURAL-JOIN	Same as EQUIJOIN except that the join attributes of S are not included in the resulting relation; if the join attributes have the same names, they do not have to be specified at all.	$R \bowtie_{<join\ condition>} S$
UNION	Produces a relation that includes all the tuples in R or S or both R and S; R and S must be union compatible.	$R \cup S$
INTERSECTION	Produces a relation that includes all the tuples in both R and S; R and S must be union compatible.	$R \cap S$
DIFFERENCE	Produces a relation that includes all the tuples in R that are not in S; R and S must be union compatible.	$R - S$
CARTESIAN PRODUCT	Produces a relation that has the attributes of R and S and includes as tuples all possible combinations of tuples from R and S.	$R \times S$
DIVISION	Produces a relation T(X) that includes all tuples t[X] in R(Z) that appear in R in combination with every tuple from S(Y), where Z = X ∪ Y.	$R(Z) \div S(Y)$

Relational Database Design

Relational Database Design:

- Functional Dependency
- Normal Forms
- Design Algorithms for 3NF and BCNF

Functional Dependencies

A function *f* from S_1 to S_2 has the property

$$if\ x, y \in S_1 \text{ and } x = y, \text{then } f(x) = f(y)\,.$$

A generalization of keys to avoid design flaws violating the above rule.

Let X and Y be sets of attributes in R.

X (*functionally*) determines *Y* , $X \rightarrow Y$, iff $t_1[X] = t_2[X]$ implies $t_1[Y] = t_2[Y]$.

i.e., f (t(X)) = t [Y]

We also say $X \rightarrow Y$ is a *functional* dependency, and that Y is *functionally* dependent on X.

X is called the *left side*, Y the *right side* of the dependency.

Armstrong's Axioms

Notation: If X and Y are sets of attributes, we write XY for their union.

e.g., X = {A, B}, Y = {B, C}, XY = {A, B, C}

1. F1 (Reflexivity) If $X \supseteq Y$ then $X \rightarrow Y$.
2. F2 (Augmentation) $\{X \rightarrow Y\} \models XZ \rightarrow YZ$.
3. F3 (Transitivity) $\{X \rightarrow Y, Y \rightarrow Z\} \models X \rightarrow Z$.
4. F4 (Additivity) $\{X \rightarrow Y, X \rightarrow Z\} \models X \rightarrow YZ$.
5. F5 (Projectivity) $\{X \rightarrow YZ\} \models X \rightarrow Y$.
6. F6 (Pseudo-transitivity) $\{X \rightarrow Y, YZ \rightarrow W\} \models XZ \rightarrow W$.

The Procedure for Computing F+

To compute the closure of a set of functional dependencies F:

$F^+ = F$
repeat
    for each functional dependency f in F^+
        apply reflexivity and augmentation rules on f
        add the resulting functional dependencies to F^+
    for each pair of functional dependencies f_1 and f_2 in F^+
        if f_1 and f_2 can be combined using transitivity
            then add the resulting functional dependency to F^+
until F^+ does not change any further

Algorithm to Compute X^+

An **algorithm** for you to follow step by step

```
X⁺ := X;
change := true;
while change do
        begin
        change := false;
        for each FD W → Z in F do
                begin
                if (W ⊆ X+) and (Z – X⁺ ≠ ∅) then do
                        begin
                        X⁺ := X⁺ ∪ Z;
                        change := true;
                        end
                end
        end
```

Compute a Candidate Key

Given a relational schema *R* and a set *F* of functional dependencies on R.

A key *X* of *R* must have the property that $X^+ = R$.

Algorithm

- Step 1: Assign *X* a superkey in F.
- Step 2: Iteratively remove attributes from *X* while retaining the property $X^+ = R$.
- The remaining *X* is a key.

Compute All the Candidate Keys

Given a relational schema R and a set F of functional dependencies on R, the algorithm to compute all the candidate keys is as follows:

T := ∅

Main:

  X := S where S is a super key which does not contain any candidate key in T

  remove := true

  While remove do

    For each attribute A ∈ X

    Compute $\{X\text{-}A\}^+$ with respect to F

    If $\{X\text{-}A\}^+$ contains all attributes of R then

      X := X – {A}

    Else

      remove := false

  T :=T ∪ X

Repeat *Main* until no available S can be found. Finally, T contains all the candidate keys.

Normal Forms

Normal Forms for relational databases:

- 1NF, 2NF, 3NF (Codd 1972)
- Boyce-Codd NF (1974)

First Normal Form (1NF)

This simply means that attribute values are *atomic and* is part of the definition of the relational model.

Atomic: multivalued attributes, composite attributes, and their combinations are disallowed.

There is currently a lot of interests in non-first normal form databases, particularly those where an attribute value can be a table (nested relations).

Second Normal Form (2NF)

A *prime* attribute is one that is part of a candidate key. Other attributes are *non-prime*.

Definition: In an FD $X \rightarrow Y$, Y is *fully functionally dependent* on X if there is no $Z \subset X$ such that $Z \rightarrow Y$. Otherwise, Y is *partially* dependent on X.

Definition (*Second Normal Form*): A relation scheme is in second normal form (2NF) if **all non-prime attributes are fully functionally dependent** on the relation keys.

A database scheme is in 2NF if all its relations are in 2NF.

Third Normal Form (3NF)

***Definition** (Third Normal Form):* A relation scheme is in *third normal form (3NF)* if for all non-trivial FD's of the form $X \rightarrow A$ that hold, **either X is a superkey or A is a prime attribute**.

Note: a FD $X \rightarrow Y$ is trivial iff Y is a subset of X.

Alternative definition: A relation scheme is in third normal form if every non-prime attribute is fully functionally dependent on the keys and not transitively dependent on any key.

A database scheme is in 3NF if all its relations are in 3NF.

Boyce-codd Normal Form (BCNF)

Definition (*Boyce-codd Normal Form*):

A relation scheme is in *Boyce-codd* Normal Form (BCNF) if whenever $X \rightarrow A$ holds (and $X \rightarrow A$ is non-trivial), **X is a superkey**.

A database scheme is in BCNF if all its relations are in BCNF.

On Relational Database Design

1. Anomalies can be removed from relation designs by decomposing them until they are in a normal form.
2. A **decomposition** of a relation scheme, R, is a set of relation schemes $\{R_1, \ldots, R_n\}$ such that $R_i \subseteq R$ for each i, and $\bigcup_{i=1}^{n} R_i = R$
3. In a decomposition $\{R_1, \ldots, R_n\}$, the intersect of each pair of R_i and R_j does not have to be empty.
   - Example: R = {A, B, C, D, E}, R_1 = {A, B}, R_2 ={A, C}, R_3 = {C, D, E}
4. A naive decomposition: each relation has only attribute.

Dependency Preserving

A decomposition $D=\{R_1, \ldots, R_n\}$ of R is **dependency-preserving** wrt a set F of FDs if:

$$(F_1 \cup \ldots \cup F_n)^+ = F^+,$$

where F_i means the **projection** of F onto R_i.

Lossless-join Decomposition

A good decomposition should have the following property.

A decomposition {R_1, . . . ,R_n} of R is a *lossless join* decomposition with respect to a set F of FD's if for every relation instance r that satisfies F:

$$r = \pi_{R_1}(r) \bowtie \cdots \bowtie \pi_{R_n}(r).$$

If $r \subset \pi_{R_1}(r) \bowtie \cdots \bowtie \pi_{R_n}(r)$, the decomposition is *lossy.*

Test Lossless Join property

This previous test works on **binary** decompositions, below is the general solution to testing lossless join property

Algorithm TEST_LJ:

1. Create a **matrix** *S*, each element $s_{i,j} \in S$ corresponds the relation R_i and the attribute A_j, such that: $s_{j,i}$ = a if $A_i \in R_j$, otherwise $s_{j,i}$ = b.
2. Repeat the following process until (1) S has no change OR (2) one row is made up entirely of “a” symbols.

   i. For each X→ Y , choose the rows where the elements corresponding to X take the value a.

   ii. In those chosen rows (must be at least two rows), the elements corresponding to Y also take the value a if one of the chosen rows take the value a on Y .

Verdict: Decomposition is *lossless* if one row is entirely made up by “a” values.

Lossless Decomposition into BCNF

Algorithm TO_BCNF

D := $\{R_1, R_2, \ldots R_n\}$

While $\exists$ a $R_i \in$ D and R_i is not in BCNF **Do**

- { find a $X \rightarrow Y$ in R_i that violates BCNF; replace R_i in D by $(R_i - Y)$ and $(X \cup Y)$; }

Computing a Minimum cover

A set F of FD's is **minimal** if

1. Every FD $X \to Y$ in F is simple: Y consists of a single attribute,
2. Every FD $X \to A$ in F is *left-reduced*: there is no proper subset

   $Y \subset X$ such that $X \to A$ can be replaced with $Y \to A$.

   that is, there is no $Y \subset X$ such that

   $$((F - \{X \to A\}) \cup \{Y \to A\})^+ = F^+$$

   i.e., Iff F |= Y -> A
3. No FD in F can be removed; that is, there is no FD $X \to A$ in F such that $(F - \{X \to A\})^+ = F^+$

   i.e., Iff X->A is inferred from F – { X->A}

Computing a Minimum cover

A *minimal cover* (or *canonical cover*) for F:

- a minimal set of FD's F_{min} such that $F^+ = F^+_{min}$.

Algorithm Min_Cover

Input: a set F of functional dependencies.

Output: a minimum cover of F.

Step 1: *Reduce right side*.

Apply Algorithm Reduce_right to F.

Step 2: *Reduce left side*.

Apply Algorithm Reduce_left to the output of Step 2.

Step 3: *Remove redundant* FDs.

Apply Algorithm Remove_redundancy to the output of Step 2.

The output is a minimum cover.

Next we detail the three algorithms (Reduce_right, Reduce_left, Reduce_redundancy) .

Computing a Minimum cover (cont)

1. **Algorithm Reduce_right**

   INPUT: F.

   OUTPUT: right side reduced F'.

   1. For each FD $X \rightarrow Y \in F$ where Y = $\{A_1, A_2, \ldots, A_k\}$, we use all X $\rightarrow \{A_i\}$ (for $1 \leq i \leq k$) to replace $X \rightarrow Y$.

2. **Algorithm Reduce_left**

   INPUT: right side reduced F.

   OUTPUT: right and left side reduced F'.

   1. For each $X \rightarrow \{A\} \in F$ where X = $\{A_i : 1 \leq i \leq k\}$, do the following. For i = 1 to k, replace X with $X - \{A_i\}$ if A $\in (X - \{A_i\})^+$.

3. **Algorithm Reduce_redundancy**

   INPUT: right and left side reduced F.

   OUTPUT: a minimum cover F' of F.

   1. For each FD $X \rightarrow \{A\} \in F$, remove it from F if: $A \in X^+$ with respect to $F - \{X \rightarrow \{A\}\}$.

Decomposition into 3NF

A **lossless and dependency-preserving decomposition** into 3NF is always possible.

Algorithm 3NF decomposition (Lossless and Dependency-preserving)

1. Find a minimal cover *G* for *F*.
2. For each left-hand-side *X* of a functional dependency that appears in *G*, create a relation schema in *D* with attributes {*X* ∪ {*A*1} ∪ {*A*2} ... ∪ {*Ak*} }, where *X* -> *A*1, *X* -> *A*2, ..., *X* -> *Ak* are the only dependencies in *G* with *X* as left-hand-side (*X* is the key to this relation).
3. If none of the relation schemas in *D* contains a key of *R*, then create one more relation schema in *D* that contains attributes that form a key of *R*.
4. Eliminate redundant relations from the resulting set of relations in the relational database schema. A relation *R* is considered redundant if *R* is a projection of another relation *S* in the schema; alternately, *R* is subsumed by *S*.

Overview: DBMS

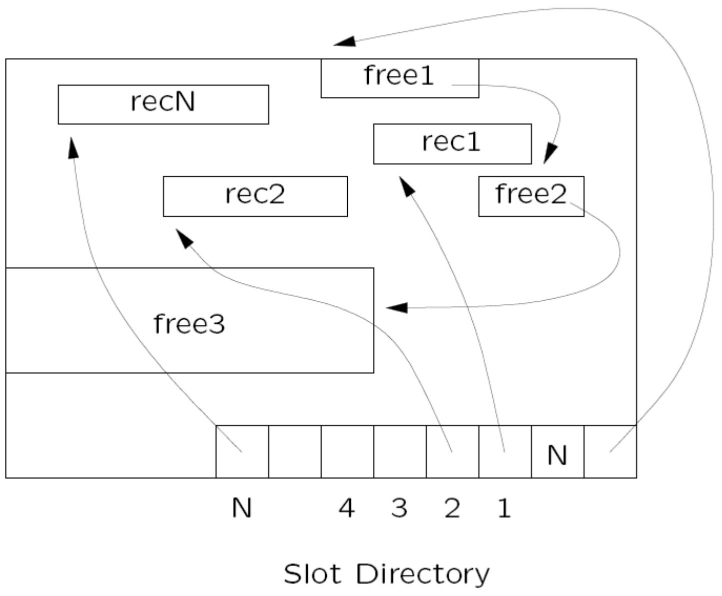
Disk, Buffer Replacement Policy

Transaction Management

- ACID properties
- Various schedules: Serializable, Conflict-Serializable, Schedule Graph, Wait for Graph, …
- Concurrency control (locking)

Record Format

- Format
  - Fixed-length
  - Variable Length
- Fragmented free space



Buffer Replacement Policies

1. Least Recently Used (LRU)
    - release the frame that has not been used for the longest period.
    - intuitively appealing idea but can perform badly
2. First in First Out (FIFO)
3. Most Recently Used (MRU):
    - release the frame used most recently

No one is guaranteed to be better than the others. Quite dependent on applications.

Desirable Properties of Transaction Processing

ACID Properties

Atomicity:

A transaction is either performed in its entirety or not performed at all.

Consistency:

A correct execution of the transaction must take the database from one consistent state to another.

Isolation:

A transaction should not make its updates visible to other transactions until it is committed.

Durability/ Permanency:

Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.

Check Conflict Serializability

Algorithm

Step 1: Construct a *schedule* (or ***precedence***) graph – a *directed graph*.

Step 2: Check if the graph is *cyclic*:

- Cyclic: non-serializable.
- Acyclic: serializable.

Construct a Precedence Graph

Schedule Graph GS = (V, A) for a schedule S

A vertex in *V* represents a transaction.

For two vertices T_i and T_j, an arc $T_i \rightarrow T_j$ is added to *A* if

- there are two *conflicting* operations $O_1 \in T_i$ and $O_2 \in T_j$,
- in *S*, O_1 is before O_2.

Recall: two operations O_1 and O_2 are *conflicting* if

- they are in different transactions but on the same data item,
- one of them must be a write.

Locking Rules

In this schema, every transaction *T* must obey the following rules.

1) If *T* has **only one** operation (read/write) manipulating an item *X*:

- obtain a read lock on *X* before reading it,
- obtain a write lock on *X* before writing it,
- unlock *X* when done with it.

2) If *T* has **several** operations manipulating *X*:

- obtain one proper lock only on *X*:
  a read lock if all operations on *X* are reads;
  a write lock if one of these operations on *X* is a write.
- unlock *X* after the last operation on *X* in *T* has been executed.

You should know **Two-Phase Locking**!

Thank you and all the best!