Comp 5311 Database Management Systems

2. Relational Model and Algebra

Basic Concepts of the Relational Model

- Entities and relationships of the E-R model are stored in tables also called relations (not to be confused with relationships in the E-R model)
- Well-defined semantics and languages for manipulating the tables
- Ease of implementation – write queries on tables without caring about the physical level and optimization issues
- Most popular DBMSs today are based on relational data model (or an extension of it, e.g., object-relational data model)

Terminology

- *Relation* ↔ table; denoted by $R(A_1, A_2, ..., A_n)$ where R is a *relation name* and $(A_1, A_2, ..., A_n)$ is the *relation schema* of R
- *Attribute (column)* ↔ denoted by A_i
- *Tuple (Record)* ↔ row
- *Attribute value* ↔ value stored in a table cell
- *Domain* ↔ legal type and range of values of an attribute denoted by $dom(A_i)$
  - Attribute: Age — Domain: [0-100]
  - Attribute: EmpName — Domain: 50 alphabetic chars
  - Attribute: Salary — Domain: non-negative integer

An Example Relation

STUDENT			
Name	Student-id	Age	CGA
Chan Kin Ho	99223367	23	11.19
Lam Wai Kin	96882145	17	10.89
Man Ko Yee	96452165	22	8.75
Lee Chin Cheung	96154292	16	10.98
Alvin Lam	96520934	15	9.65

Characteristics of Relations

- Tuples in a relation are *not* considered to be *ordered*, even though they appear to be in a tabular form. (Recall that a relation is a set of tuples.)
- All attribute values are considered *atomic*. Multivalued and composite attribute values are not allowed in tables, although they are permitted by the ER diagrams
- A special *null* value is used to represent values that are:
  - *Not applicable* (phone number for a client that has no phone)
  - *Missing values* (there is a phone number but we do not know it yet)
  - *Not known* (we do not know whether there is a phone number or not)

Keys

- Let $K \subseteq R$ *(I.e., K is a set of attributes* which is *a subset of* the schema of *R)*
- *K* is a *superkey* of *R* if *K* can identify a unique tuple in a given *relation* *r*(*R*)

Student(SID, HKID, Name, Address, …)
where SID and HKID are unique.
Possible superkeys: SID
HKID
{SID, Name}
{HKID, Name, Address}
plus many others

- K is a *candidate key* if K is *minimal*
  - In the above example there are *two* candidate keys: SID and HKID
- Every relation must have at least one candidate key.
- If there are multiple, one is chosen as the primary key.

Need for Multiple Tables

- Storing all information as a single relation such as *bank*(*account-number, balance, customer-name*, *customer-addr*, ..) results in
  - repetition of information (e.g. repeat the customer info for each of his/her accounts)
  - the need for null values (e.g. represent a customer without an account)
- That is why we need the ER diagrams (and some additional normalization techniques discussed later) to break up information into parts, with each relation storing one part.

  E.g.: *account* : stores information about accounts
  depositor : stores information about which customer owns which account
  customer : stores information about customers

Reduction of an E-R Schema to Relations

- A database which conforms to an E-R diagram can be represented by a collection of tables.
- Converting an E-R diagram to a table format is "automatic".
- For each entity set there is a unique table which is assigned the name of the corresponding entity set.
- Each table has a number of columns (generally corresponding to attributes), which have unique names.

Composite and Multivalued Attributes

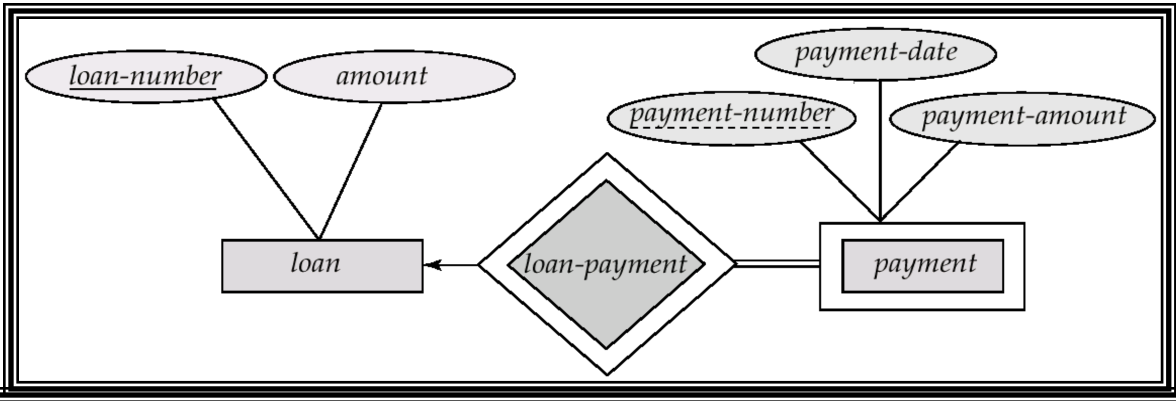
- Composite attributes are flattened out by creating a separate attribute for each component attribute
  - E.g. given entity set customer with composite attribute *name* with component attributes *first-name* and *last-name* the customer table has two attributes

    name.first-name and *name.last-name*
- A multivalued attribute M of an entity E is represented by a separate table EM
  - Table EM has attributes corresponding to the primary key of E and an attribute corresponding to multivalued attribute M
  - E.g. Multivalued attribute *phone-number* of *employee* is represented by a table

    employee-phone(*employee-id, phone-number*)
  - Each value of the multivalued attribute maps to a separate row of the table EM
    - E.g., an employee with primary key 19444 and phones 23580000, 95555555 maps to two rows: (19444, 23580000) and (19444, 95555555)

Representing Weak Entity Sets

A weak entity set becomes a table that includes a column for the primary key of the identifying strong entity set



loan-number	*payment-number*	*payment-date*	*payment-amount*
L-11	53	7 June 2001	125
L-14	69	28 May 2001	500
L-15	22	23 May 2001	300
L-16	58	18 June 2001	135
L-17	5	10 May 2001	50
L-17	6	7 June 2001	50
L-17	7	17 June 2001	100
L-23	11	17 May 2001	75
L-93	103	3 June 2001	900
L-93	104	13 June 2001	200

Representing Relationship Sets as Tables

- A many-to-many relationship set is represented as a table with columns for the primary keys of the two participating entity sets, and any descriptive attributes of the relationship set.
- E.g.: table for relationship set *borrower*

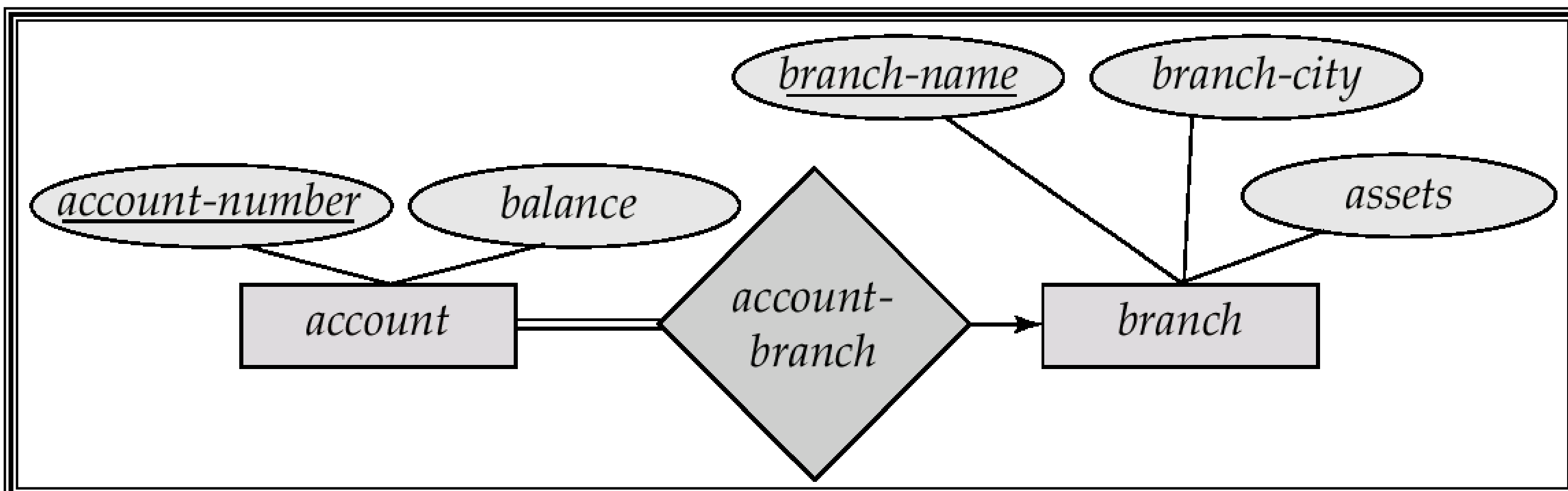
customer-id	*loan-number*
019-28-3746	L-11
019-28-3746	L-23
244-66-8800	L-93
321-12-3123	L-17
335-57-7991	L-16
555-55-5555	L-14
677-89-9011	L-15
963-96-3963	L-17

Redundancy of Tables

Many-to-one and one-to-many relationship sets that are total on the many-side can be represented by adding an extra attribute to the many side, containing the primary key of the one side

Instead of creating a table for relationship *account-branch*, add the key of branch (*branch-name*) to the entity set *account*

branch-name in account is a foreign key



Redundancy of Tables (Cont.)

- For one-to-one relationship sets, either side can be chosen to act as the "many" side
  - That is, extra attribute can be added to either of the tables corresponding to the two entity sets
- If participation is *partial* on the many side, replacing a table by an extra attribute in the relation corresponding to the "many" side could result in null values
- The table corresponding to a relationship set linking a weak entity set to its identifying strong entity set is redundant.
  - E.g. The *payment* table already contains the information that would appear in the *loan-payment* table (i.e., the columns loan-number and *payment-number*).

Representing Specialization as Tables

- Method 1:
  - Form a table for the higher level entity
  - Form a table for each lower level entity set, include primary key of higher level entity set and local attributes

table	table attributes
person	*id, name, street, city*
customer	*id, credit-rating*
employee	*id, salary*

  - Drawback: getting information about, e.g., *employee* requires accessing two tables

Specialization as Tables (Cont.)

- Method 2:
  - Form a table for each entity set with all local and inherited attributes

table	table attributes
person	*id, name, street, city*
customer	*id, name, street, city, credit-rating*
employee	*id, name, street, city, salary*

    If specialization is total, no need to create table for generalized entity (*person*)

  - Drawback: street and city may be stored redundantly for persons who are both customers and employees

Relational Query Languages

- *Query languages (QL)*: Allow retrieval of data from a database.

- Relational model supports simple, powerful QLs:
  - Strong formal foundation based on logic.
  - Allows for much optimization.
- Query Languages **!=** programming languages!
  - QLs not expected to be "Turing complete".
  - QLs not intended to be used for complex calculations.
  - QLs support easy and efficient access to large data sets.

Formal Relational Query Languages

- Two mathematical Query Languages form the basis for "real" languages (e.g. SQL), and for implementation:

  Relational Algebra: Procedural, very useful for representing execution plans.

  Relational Calculus: Lets users describe what they want, rather than how to compute it. (Non-Procedural, *declarative*.)

We focus on Algebra: *Understanding Algebra is key to understanding SQL and query processing!*

Relational Algebra

- Basic operations:
  - *Projection* (π) Deletes unwanted columns from relation.
  - *Selection* (σ) Selects a subset of rows from relation.
  - *Set-difference* (**-**) Finds tuples in table 1, but not in table 2.
  - *Union* ($\cup$) Finds tuples that belong to table 1 or table 2.
  - *Cross-product* (**x**) Allows us to combine two relations.
  - *Rename* (*p*) Allows us to rename a relation and/or its attributes.
- Additional operations:
  - *Intersection*, *join*, *division*: Not essential, but (very!) useful.
- Each operation returns a relation, and operations can be *composed*! Algebra is "closed".

Projection $\pi_L(R)$

- Deletes attributes that are not in *projection list L*.
- *Schema* of result contains exactly the fields in the projection list, with the same names that they had in the (only) input relation.
- Projection operator eliminates *duplicates*!

Plane

Maker	Model_No
Airbus	A310
Airbus	A320
Airbus	A330
Airbus	A340
MD	DC10
MD	DC9

π_{Maker}(Plane)

Maker
Airbus
MD

Selection $\sigma_c(R)$

- Selects rows (records/tuples) that satisfy a *selection condition c*.
- *Schema* of result identical to schema of (only) input relation.

- A condition c has the form**: Term** *Op* **Term**
  - where **Term** is an attribute name or **Term** is a constant
  - *Op* is one of <, >, =, ≠, etc.

- **(C1 ∧ C2)**, **(C1 ∨ C2)**, **(¬ C1)** are conditions where C1 and C2 are conditions.
- ∧ means AND
- ∨ means OR
- ¬ means NOT

Selection example

Plane

Maker	Model_No
Airbus	A310
Airbus	A320
Airbus	A330
Airbus	A340
MD	DC10
MD	DC9

$\sigma_{Maker=\text{“MD”}}(Plane)$

Maker	Model_No
MD	DC10
MD	DC9

- No duplicates in result! (Why?)
- The resulting relation can be the *input* for another relational algebra operation! (*Operator composition*)

Plane

Maker	Model_No
Airbus	A310
Airbus	A320
Airbus	A330
Airbus	A340
MD	DC10
MD	DC9

$\pi_{Model\_No}(\sigma_{Maker=\text{“MD”}}(Plane))$

Model_No
DC10
DC9

Set Operations

- Union, Intersection, Set-Difference
- These three operations take two input relations, which must be *union-compatible*:
  - Same number of fields.
  - Corresponding fields have the same type.
- Output is a single relation (that does not contain duplicates)

Set operations - Union

- **$Plane_1 \cup Plane_2$**

Maker	Model_No
Airbus	A310
Airbus	A320
Airbus	A330
Airbus	A340
MD	DC10
MD	DC9

$\cup$

Maker	Model_No
Boeing	B727
Boeing	B747
Boeing	B757
MD	DC10
MD	DC9

=

Maker	Model_No
Airbus	A310
Airbus	A320
Airbus	A330
Airbus	A340
Boeing	B727
Boeing	B747
Boeing	B757
MD	DC10
MD	DC9

Set operations – Set difference

- **$Plane_1$ — $Plane_2$**
  - Contains records that appear in **$Plane_1$** but not **$Plane_2$**

Maker	Model_No
Airbus	A310
Airbus	A320
Airbus	A330
Airbus	A340
MD	DC10
MD	DC9

—

Maker	Model_No
Boeing	B727
Boeing	B747
Boeing	B757
MD	DC10
MD	DC9

=

Maker	Model_No
Airbus	A310
Airbus	A320
Airbus	A330
Airbus	A340

Set operations - Intersection

- **$Plane_1 \cap Plane_2$**
  – Contains records that appear in both tables

Maker	Model_No
Airbus	A310
Airbus	A320
Airbus	A330
Airbus	A340
MD	DC10
MD	DC9

$\cap$

Maker	Model_No
Boeing	B727
Boeing	B747
Boeing	B757
MD	DC10
MD	DC9

=

Maker	Model_No
MD	DC9
MD	DC10

Intersection is not a primitive operation

- $R \cap S = ((R \cup S) - (R - S)) - (S - R)$

Compute all tuples belonging to R or S

Remove the ones that belong only to R

Remove the ones that belong only to S

Also: $R \cap S = R - (R - S)$

Cartesian Product

- Combines each row of one table with every row of another table
- Can_fly × Plane

Emp_No	Model_No
1001	B727
1001	B747
1001	DC10
1002	A320
1002	A340
1002	B757
1002	DC9
1003	A310
1003	DC9

×

Maker	Model_No
Airbus	A310
Airbus	A320
Airbus	A330
Airbus	A340
Boeing	B727
Boeing	B747
Boeing	B757
MD	DC10
MD	DC9

=

Emp_No	Model_No	Maker	Model_No
1001	B727	Airbus	A310
1001	B727	Airbus	A320
1001	B727	Airbus	A330
1001	B727	Airbus	A340
1001	B727	Boeing	B727
1001	B727	Boeing	B747
1001	B727	Boeing	B757
1001	B727	MD	DC10
1001	B727	MD	DC9
1001	B747	Airbus	A310
1001	B747	Airbus	A320
1001	B747	Airbus	A330
1001	B747	Airbus	A340
1001	B747	Boeing	B727
1001	B747	Boeing	B747
1001	B747	Boeing	B757
1001	B747	MD	DC10
1001	B747	MD	DC9
1001	B727	Airbus	A310
1001	B727	Airbus	A320
...	...	...	...

81 t-uples!!!

Join

- Generating all possible combinations of tuples is not usually meaningful.
- In the previous example, it makes more sense to combine each tuple of Can_Fly with the corresponding record of the Plane.
- Join is a cartesian product followed by a selection:

  $\mathbf{R_1} \bowtie_c \mathbf{R_2} = \sigma_c(R_1 \times R_2)$
- Sometimes we use the word JOIN instead of symbol $\bowtie$
- Types of joins:

  θ-join: arbitrary conditions in the selection

  Equijoin: all conditions are equalities

  Natural join: combines two relations on the equality of the attributes with the same names
- Both equijoin and natural join project only one of the redundant attributes

Natural Join Example

Can_fly $\bowtie_n$ Plane

Can_fly JOIN_n Plane

Can_fly $\text{JOIN}_{\text{Model\_No}}$ Plane

Can_fly $\text{JOIN}_{\text{Can\_fly.Model\_No=Plane.Model\_No}}$ Plane

Emp_No	Model_No
1001	B727
1001	B747
1001	DC10
1002	A320
1002	A340
1002	B757
1002	DC9
1003	A310
1003	DC9

$\bowtie_n$

Maker	Model_No
Airbus	A310
Airbus	A320
Airbus	A330
Airbus	A340
Boeing	B727
Boeing	B747
Boeing	B757
MD	DC10
MD	DC9

=

Emp_No	Model_No	Maker
1003	A310	Airbus
1002	A320	Airbus
1002	A340	Airbus
1001	B727	Boeing
1001	B747	Boeing
1002	B757	Boeing
1001	DC10	MD
1002	DC9	MD
1003	DC9	MD

θ-Join Example

- We have a Flight table that records the Number of the flight, Origin, Destination, Departure and Arrival Time.
- We join this table with itself (*self-join*) using the condition:
- Flight1.Dest = Flight2.Origin ∧ Flight1.Arr_Time < Flight2.Dept_Time
- What should we get?

Num	Origin	Dest	Dep_Time	Arr_Time
334	ORD	MIA	12:00	14:14
335	MIA	ORD	15:00	17:14
336	ORD	MIA	18:00	20:14
337	MIA	ORD	20:30	23:53
394	DFW	MIA	19:00	21:30
395	MIA	DFW	21:00	23:43

$\bowtie$ …

Num	Origin	Dest	Dep_Time	Arr_Time
334	ORD	MIA	12:00	14:14
335	MIA	ORD	15:00	17:14
336	ORD	MIA	18:00	20:14
337	MIA	ORD	20:30	23:53
394	DFW	MIA	19:00	21:30
395	MIA	DFW	21:00	23:43

θ-Join Example (cont)

Flight1.Dest = Flight2.Origin ∧ Flight1.Arr_Time < Flight2.Dept_Time

Flight1.Num	Flight1.Origin	Flight1.Dest	Flight1.Dep_Time	Flight1.Arr_Time	Flight2_1.Num	Flight2.Origin	Flight2.Dest	Flight2.Dep_Time	Flight2.Arr_Time
334	ORD	MIA	12:00	14:14	335	MIA	ORD	15:00	17:14
335	MIA	ORD	15:00	17:14	336	ORD	MIA	18:00	20:14
336	ORD	MIA	18:00	20:14	337	MIA	ORD	20:30	23:53
334	ORD	MIA	12:00	14:14	337	MIA	ORD	20:30	23:53
336	ORD	MIA	18:00	20:14	395	MIA	DFW	21:00	23:43
334	ORD	MIA	12:00	14:14	395	MIA	DFW	21:00	23:43

What happens if we add the condition Flight1.Origin ≠ Flight2.Dest

Renaming ρ

- If attributes or relations have the same name it may be convenient to rename one

$$\rho(R'(N_1 \text{ -> } N'_1, N_n \text{ -> } N'_n), R)$$

- The new relation R′ has the same instance as R, but its schema has attribute N'_i instead of attribute N_i
- **Example:** ρ(Staff(Name -> Family_Name, Salary -> Gross_salary), Employee)

- Necessary if we need to perform a cartesian product or join of a table with itself

Employee

Name	Salary	Emp_No
Clark	150000	1006
Gates	5000000	1005
Jones	50000	1001
Peters	45000	1002
Phillips	25000	1004
Rowe	35000	1003
Warnock	500000	1007

Staff

Family_Name	Gross_Salary	Emp_No
Clark	150000	1006
Gates	5000000	1005
Jones	50000	1001
Peters	45000	1002
Phillips	25000	1004
Rowe	35000	1003
Warnock	500000	1007

Division

Let A have two fields x and y

Let B have one field y

A/B contains all x tuples, such that for **every** y tuple in B there is a xy tuple in A

A

x	y
s1	p1
s1	p2
s1	p3
s1	p4
s2	p1
s2	p2
s3	p2
s4	p2
s4	p4

/

B

y
p2
p4

=

x
s1
s4

A/B

Example Division

Find the Employment numbers of the pilots who can fly **all** MD planes

Can_Fly / $\pi_{Model\_No}(\sigma_{Maker='MD'}Plane)$

Emp_No	Model_No
1001	B727
1001	B747
1001	DC10
1002	A320
1002	A340
1002	B757
1002	DC9
1003	A310
1003	DC9
1003	DC10

Maker	Model_No
Airbus	A310
Airbus	A320
Airbus	A330
Airbus	A340
Boeing	B727
Boeing	B747
Boeing	B757
MD	DC10
MD	DC9

Emp_No
1003

Additional Operators - Outer Join

- An extension of the join operation that avoids loss of information.
- Computes the join and then adds tuples from one relation that do not match tuples in the other relation to the result of the join.
- Uses **null** values in left- or right- outer join:
  - null signifies that the value is unknown or does not exist.
  - All comparisons involving null are false by definition.

Outer Join - Example

loan

branch-name	loan-number	amount
Downtown	L-170	3000
Perryridge	L-260	1700
Redwood	L-230	4000

borrower

cust-name	loan-number
Jones	L-170
Smith	L-230
Hayes	L-155

Loan ⋈ Borrower

branch-name	loan-number	amount	cust-name
Downtown	L-170	3000	Jones
Redwood	L-230	4000	Smith

Join returns only the matching (or "good") tuples
The fact that loan L-260 has no borrower is not explicit in the result
Hayes has borrowed an non-existent loan L-155 is also undetected

Left Outer Join -Example

Left outer join: Loan $⟕$ borrower

Keep the entire left relation (Loan) and fill in information from the right relation, use null if information is missing.

branch-name	loan-number	amount	cust-name
Downtown	L-170	3000	Jones
Perryridge	L-260	1700	null
Redwood	L-230	4000	Smith

Right and Full Outer Join - example

Loan ⟖ Borrower

branch-name	amount	cust-name	loan-number
Downtown	3000	Jones	L-170
Redwood	4000	Smith	L-230
null	null	Hayes	L-155

Loan ⟗ borrower

branch-name	amount	cust-name	loan-number
Downtown	3000	Jones	L-170
Redwood	4000	Smith	L-230
Perryridge	1700	null	L-260
null	null	Hayes	L-155