Entity-Relationship Model

Legend

- Rectangles: entity sets (table)
  - Double rectangle: weak entity set (table with column for primary key of identifying strong entity set)
- Ellipses: attributes of the entity (column in table)
  - Double ellipse: multi-valued attribute (separate table)
  - Dashed ellipse: derived attribute
  - Child ellipses: composite attributed (column in table)
- Diamonds: Relationship sets (between entities)
  - Relational Table:
    - many-to-many: seperate table with primary keys of participating entity sets
    - many-to-one: if 'many' is total, add the primary key of the 'one'
    - one-to-one: use many-to-one, where either side can be the 'many'
  - Double diamond: identifying relationship
- Triangle: Specialization relationship
  - Relational Table: either
    - Table containing primary key of the higher-level entity set and local attributes)
    - Table with all local and inherited attributes
      - PRO: Efficient data retrival
      - CON: Storage redundancy
- Directed Line (->): one
- Undirected Line (--): many
- Double line (=): total relationship
- Single line (-): partial relationship
- Underline: Primary key
  - Dotted underline: weak set identifier
- Disjoint label: disjoint specialization

Entity and Entity set

- Entity: an object
  - e.g Customer, Account
- Attribute: something an entity has

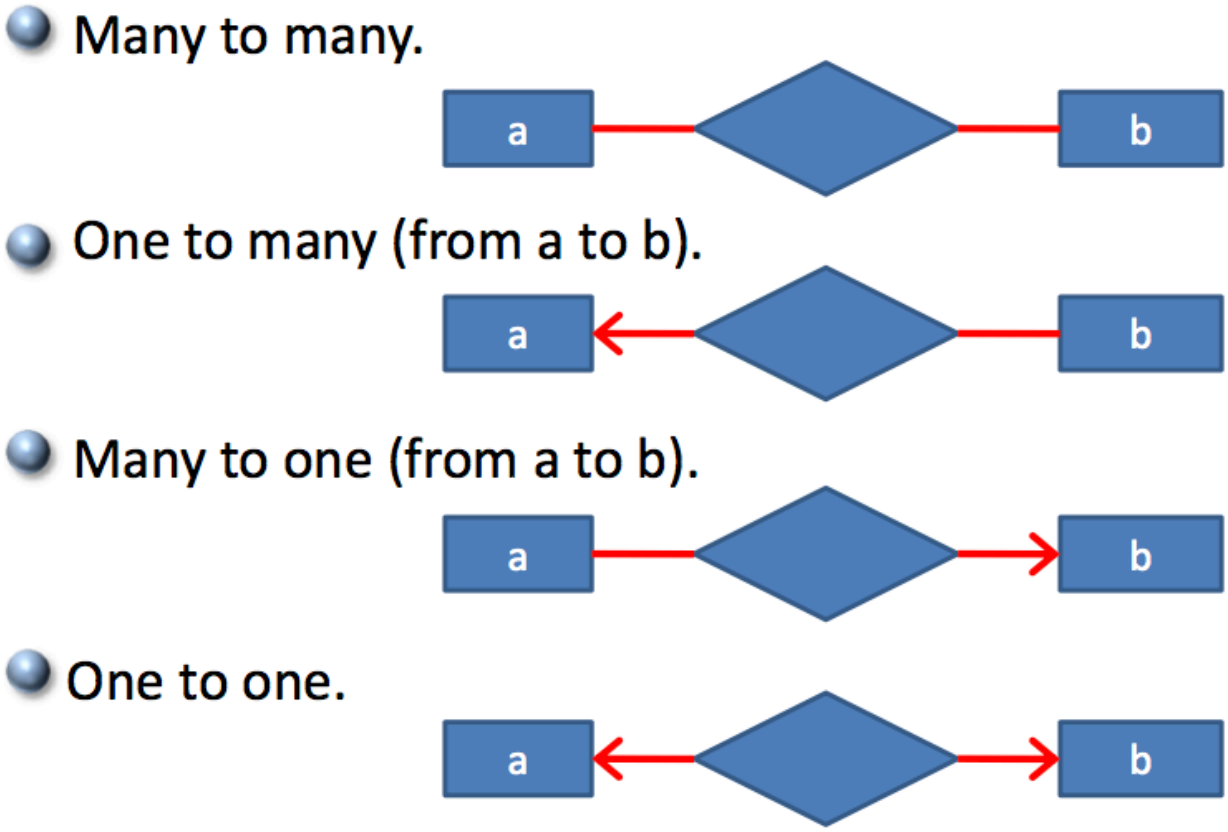
    - e.g Customers have names
- Entity set: set of entities of the same type that share the same properties
    - Represents a table with the same attributes
    - e.g All Customers

- Relationship: association among entities
- Relationship set: set of relationships of the same type
    - Diamond with relationship type

Constraints

Mapping cardinalities

- Directed Line (->): one
- Undirected Line (--): many



Participation constraints

- Total participaction (=): every entity participates in at least one relationship
- Partial participation (-): some entity may not participate in any relationship

Keys

- Super key: set of one or more attributes that uniquely determine each entity
  - e.g {user_id, name}
- Candidate key: minimal (no subset super key exists) super key
  - e.g {user_id}
- Primary key (underlined): the candidate key selected to be the primary key

Different Attribute Types

- Single vs Composite attributes (child ellipses)
  - Composite attributes are made of multiple attributes
  - e.g Address = {street, city, state}
- Single-valued vs Multi-valued attributes (double ellipses)
  - e.g Multiple phone numbers
  - Don't actually use this
- Derived attribute (dashed ellipses)
  - Values derived from other attributes
  - e.g age derived from date_of_birth

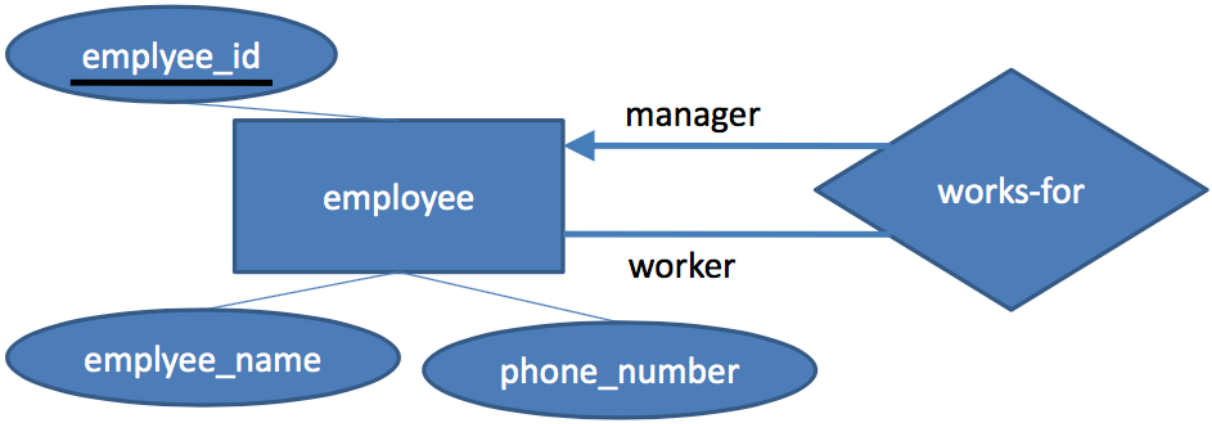
Weak Entity Set

- Represented by a double rectangle
- Entity set that does not have a primary key
- Must have a total, many-to-one identifying relationship set (double diamond) to the identifying entity set
- Discriminator (dotted underline): identifies entities within the weak set
- e.g Player's number is only unique within their team

Role

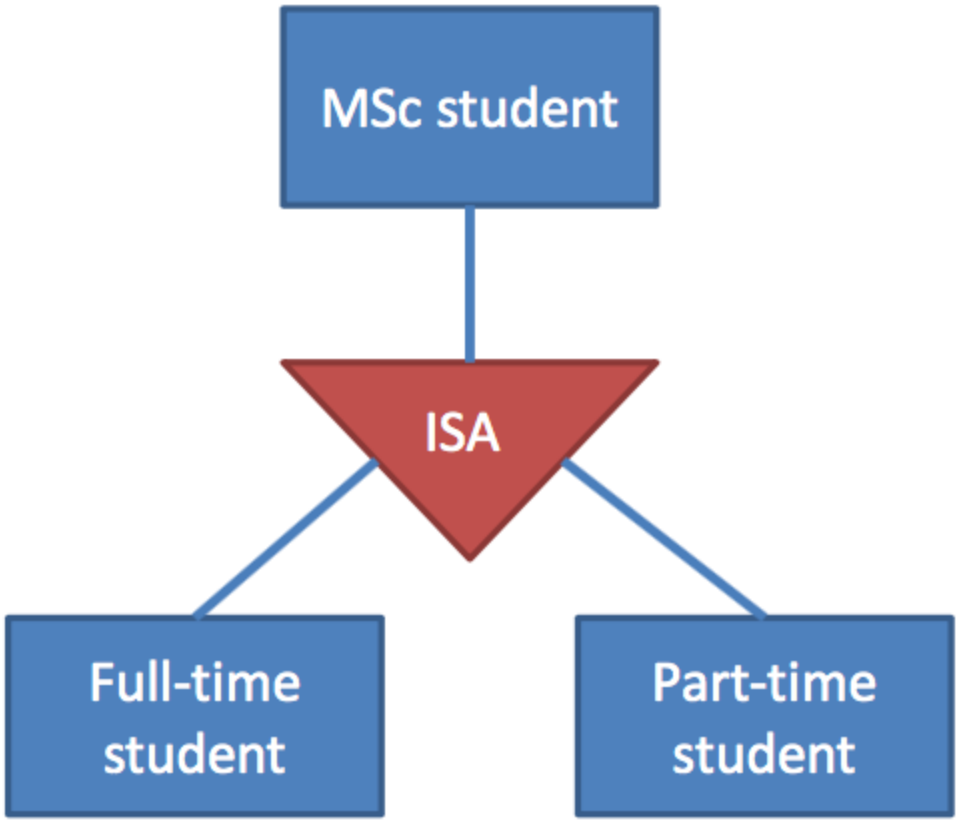
- Relationship between entities in the same entity set

- Labeled with the different roles of the relationship



Specialization

- Represented by a Triangle with relationship type (e.g ISA)
- Lower-level entity set that inherits all attributes and relationships of the higher level entity set
- Can also have its own attributes
- e.g Student can be full-time or part-time
- Disjoint vs Overlapping
  - Disjoint: entities can only belong to one specialization



Structured Query Language

- Language for defining, modifying and querying data
- Declarative (states what we want, not how)

Data Definition

Table Creation

```
CREATE TABLE table_name
(
    column1_name column_type CONDITIONS,
    ...
    CONSTRAINTS
);
```

- Conditions: `NOT NULL`
- Constraints:
  - `PRIMARY KEY(column_name)`
  - `FOREIGN KEY(column_name) REFERENCES table_name(key)`

Table Deletion

```
DROP TABLE table_name;
```

- Drop instruction may be rejected if table is referenced by other tables via constraints

Table Modification

```
ALTER TABLE table_name ADD column_name;
ALTER TABLE table_name DROP column_name;
ALTER TABLE table_name ADD CONSTRAINT;
```

Insert, Delete, Update

Insert

```
INSERT INTO table_name VALUES
    (column1_value, column2_value, column3_value) -- row 1
    (column1_value, column2_value, column3_value) -- row 2
    ;
LOAD DATA LOCAL INFILE 'file.txt'
    INTO TABLE table_name
    FIELDS TERMINATED BY ','
    LINES TERMINATED BY '\n';
```

Delete

```
DELETE FROM table_name; -- Delete all records
DELETE FROM table_name WHERE column_name = 'test_value';
```

Update

```
UPDATE table_name
    SET column1_name = 'value'
    WHERE column2_name = 'test_value';

UPDATE table_name
    SET column1_name = column1_name * 10;

UPDATE table_name
    SET column_name = CASE
        WHEN column_name <=500 THEN column_name * 1.05
        ELSE column_name * 1.06
        END;
```

Querying

Select

```sql
SELECT * FROM table_name; -- all columns
SELECT column_name FROM table_name;
SELECT name, amount * 2 FROM table; -- can do math

-- conditions
SELECT table1.column_name FROM table1, table2
    WHERE table1.property = table2.property;
SELECT cost FROM table1
    WHERE cost >= 100 AND seller = 'Kelly';

-- distinct
SELECT DISTINCT column_name FROM table_name;

-- renaming
SELECT column_name AS alias FROM table_name T;
```

String Operations

```sql
WHERE name LIKE 'J% D%';
```

- `%` matches any substring
- `_` matches any character

Ordering

```sql
ORDER BY column_name ASC;
ORDER BY column_name DESC;
```

Nesting

```sql
-- IN, SOME, ALL, EXISTS
SELECT a FROM t
WHERE id IN (
    SELECT user_id FROM users
);
```

Aggregation

```sql
-- AVG MIN MAX SUM COUNT
SELECT AVG(column_name) FROM table_name;
SELECT user_id, AVG(balance)
    FROM users
    GROUP BY user_id
    HAVING AVG(balance) >= 100;
```

Join

```sql
-- OUTER JOIN, INNER JOIN, CROSS JOIN
SELECT * FROM t1 JOIN t2
    ON t1.id = t2.id;
```

Null

- `value IS NULL`
- `(5 + null) == null)`
- `(null = null) == UNKNOWN`

Views

```sql
CREATE VIEW view_name AS (<query>)
```

- Mechanism to hide certain data from certain users

Authorization

```sql
GRANT privilege ON table_name TO user_or_role;
REVOKE privilege ON table_name FROM user_or_role;

CREATE ROLE role_name;
GRANT role_name TO user_name;
```

Assertions

```
CREATE ASSERTION assertion_name
CHECK (
    (SELECT COUNT(*) FROM t1) <= (SELECT COUNT(*) FROM t2)
);
```

- Checked everytime the table is updated

Relational Algebra

- Algebra with tables (relations) as operand, and new operators
- Models lower-level operations of a relational DBMS

Basic Operators

Symbol	Name	Description	Example
σ	Select	Filter on a condition	$\sigma_{\text{name="May"}}(\text{Author})$
π	Projection	Copy with fewer attributes	$\pi_{\text{id, name}}(\text{Users})$
∪	Union	Combine two sets with the same attributes	$\pi_{\text{name}}(\text{Students}) \cup \pi_{\text{name}}(\text{Teachers})$
-	Set difference	Difference of two sets with the same attributes	$\pi_{\text{id}}(\text{Students}) - \pi_{\text{id}}(\text{Enrolled})$
×	Cartesian product	Join of all possible combinations	Items × Orders
ρ	Rename	Rename a table	$\rho_{\text{c}}(\text{Course})$

Additional Operators

Symbol	Name	Description	Example
∩	Set intersection	Intersection of two sets with the same attributes	$(\sigma_{department\_id=1}(Works\_in))$ ∪ $(\sigma_{department\_id=3}(Works\_in))$
⋈	Natural Join	Join on matching attributes	Items ⋈ Orders
←	Assignment	Create a temporary relation	temp1 ← $\pi_a(R)$
⟕ ⟖	Left/Right outer join	Natural join including non-matching tuples	Users ⟕ Orders
÷	Division	Returns entries in R that cover all values in S	StoreFeatures ÷ $\pi_{feature\_id}(Features)$

Extended Operators

Symbol	Name	Description	Example

Aggregation

- avg, min, max, sum, count, count-distinct
- $_{G_1, G_2}g_{F_1(A_1),F_2(A_2)}(R)$
  - G_1, G_2 = attributes to group by
  - $F_1(A_1),F_2(A_2)$ = aggregate functions to apply

Algebraic Properties

Equivalence Rules

1. $\pi_1(\pi_2(\ldots(\pi_n(E)))) = \pi_1(E)$
2. $\sigma_{p \wedge q}(E)=\sigma_p(\sigma_q(E))$
3. $\sigma_p(\sigma_q(E)) = \sigma_q(\sigma_p(E))$
4. $(E_1 \bowtie E_2)\bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$
5. - 5a. $\sigma_p(E_1 \bowtie E_2)=\sigma_p(E_1)\bowtie E_2$
   - 5b. $\sigma_{p \wedge q}(E_1 \bowtie E_2)=\sigma_p(E_1)\bowtie\sigma_q(E_2)$
6. $\pi_{L_1 \cup L_2}(E_1 \bowtie E_2)=\pi_{L_1 \cup L_2}(\pi_{L_1 \cup L_3}(E_1)\bowtie\pi_{L_2 \cup L_3}(E_2)$
7. $E_1 \cup E_2 = E_2 \cup E_1$
   $E_1 \cap E_2 = E_2 \cap E_1$
8. $(E_1 \cup E_2)\cup E_3 = E_1 \cup (E_2 \cup E_3)$
   $(E_1 \cap E_2)\cap E_3 = E_1 \cap (E_2 \cap E_3)$
9. $\sigma_p(E_1 \cup E_2)=\sigma_p(E_1)\cup\sigma_p(E_2)$
   $\sigma_p(E_1 \cap E_2)=\sigma_p(E_1)\cap\sigma_p(E_2)$
   $\sigma_p(E_1 - E_2)=\sigma_p(E_1)-\sigma_p(E_2)$
10. $\pi_L(E_1 \cup E_2)=\pi_L(E_1)\cup\pi_L(E_2)$

Query Optimization

- Join smaller tables first
- Perform selection as early as possible
- Reduce attributes involved in joins

Database Design

Functional Dependency (FD)

- Certain set of attributes uniquely determine the values for another set of attributes
- e.g {employee_id} -> {name, phone}

Armstrong's Axioms

1. Reflexivity: if B ⊆ A, then A → B
2. Transitivity: if A → B and B -> C, then A → C
3. Augmentation: if A → B, then CA → CB
4. Union: if A → B and A → C, then A → BC
5. Decomposition: if A → BC, then A → B and A → C
6. Pseudo-transitivity: if A → B and BC → D, then AC → D

Attribute Set Closure

- The **closure** of α (denoted α+) is the set of attributes that can be functionally determined by α
    - e.g F = {A→B, B→C}
        - {A}+ = {A, B, C}
        - {C}+ = {C}

Candidate Key

- α is a candidate key if:
    - α is a **superkey**
        - α+ contains all attributes of R
    - α is minimum
        - No subset of α is a superkey

FD Closure

- Set of all functional dependencies that can be logically implied by the set
- To compute F+ in relation R
    1. For every subset α of R, compute α+
    2. For every subset of α+, generate a FD with α as the LHS
    3. F+ = all the FD's

Decomposition

Lossless-join

- Can a relation R be seperated into R_1 and R_2, such that no information is lost?
- $R = R_1 \cup R_2$
- A decomposition is a lossless-join IFF at least one of the following is true in F+:
    - Schema of R_1 ∩ Schema of R_2 → Schema of R_1
    - Schema of R_1 ∩ Schema of R_2 → Schema of R_2

Dependency Preserving

- Is the dependency X → Y preserved in R, if R is decomposed into R_1 and R_2?
- Test:
    - F_1 = projection of F+ on R_1
    - F_2 = projection of F+ on R_2
    - Decomposition is dependency preserving if $(F_1 \cup F_2)^+ = F^+$

Boyce-Codd Normal Form

- No redundencies in F
- For all FD's in F+ of the form α→β, at least one of the following holds:
  - α→β is trivial
  - α is a *superkey* for R
- Test:
  - Check non-trivial α→β dependencies in F for if α+ covers the whole relation
  - (Don't need to check all of F+)

BCNF Decomposition Algorithm

- Chapter 5B, Slide 50

Normalization

- Goals when decomposing a relation
  1. Lossless-join
  2. No redundancy
  3. Dependency preserving

Files and Storage

Storage Media

CPU Cache

- Extremely fast memory built into the CPU
- Unimportant to database storage XD

Main Memory

- Volatile, fast storage
- Too small to store the entire database

Magnetic Disk

- Large, non-volatile long-term storage
- Slow access time
- Read using read-write heads
- Has many platters, each with two surfaces of information with many tracks

- Tracks are divided into sectors
- Number of Cylinders = number of tracks per surface
- Blocks consist of multiple consecutive sectors
- Read time:
    1. Seek: Position read-write head to the right track
    2. Rotation: Spin disk to position data area
    3. Transfer data: Read and transfer the data

Magnetic Tape

- Used for backup
- Slow, non-volatile, cheap

Optical Storage

- Non-volatile, slow, small

Flash Memory (SSD)

- Read is fast, write is slower
- Limited write cycles

Storage Hierarchy

- Data on disks
- Memory for temporary storage and data manipulation
- Backups on tertiary storage

Reliability and Efficiency

- Hard disks may fail
- Disks are slow compared to CPU
- e.g RAID (Redundant Arrays of Independent Disks)
    - Level 0: Striping only, no mirroring
    - Level 1/10/1+0: Mirroring and Striping
    - Level 5: Strips all disks including parity disk

Solution: Mirroring

- Store a redundant copy on another disk
- Reads can be handled twice as quickly
- High reliability, but expensive

Solution: Data Striping

- Partition data into several disks that can be read in parallel
- Higher data-transfer rate, but no effect on reliability

Solution: Parity Check

- Use an extra disk to store parity bits
- If a disk fails, the parity bit can be used to reconstruct the data

File Organization

- Records are stored in files, managed by the database system
- Files are partitioned into blocks which contain records

Records

Fixed-Length Records

- e.g INT, CHAR
- Place data into blocks without crossing blocks
- Store pointers to deleted records in the header and free spaces

Variable-Length Records

- e.g VARCHAR(20), TEXT
- Stored using a slotted-page structure
    - Block header contains information about entires in the block

Organizing Blocks in Files

Heap File

- Records are unordered and can be placed anywhere
- Simple
- Data is scattered
- All data needs to be scanned to locate a record

Sequential File

- Store records sequentially based on search key
- Order maintained through overflow blocks

Hashing

- Block is chosen based on hash value

Multitable Clustering

- Put related relations in the same file
- Joins are faster

Buffer

- Aim to minimize block transfers between disk and memory
- Buffer cases:
  - Buffer Miss
    - Data is not in buffer; read it from disk
  - Buffer Hit
    - Read is from buffer
  - Write
    - Update buffer
    - Update disk when flushing buffer
  - Buffer Full
    - Replace data by LRU or LFU
- Data dictionary is always kept in buffer
  - Contains information about relations, statistics, file organization

Indexing

- Used to speed up access to data
- Indexed based on a search key
- Indexes are smaller than the original file
- Primary vs Secondary
  - Primary index = main search key
  - Secondary index = secondary search key
- Index Classes
  - Ordered
  - Hash
- Factors
  - Access types supported
    - Equality, range, multi-attribute
  - Access time

- Insertion / Deletion time
- Space overhead

B+ Tree

- All paths from root to leaf are the same length
- Efficient processing of equality and range queries
- Node contains `n-1` search-key values and `n` pointers
- Leaf node has `((n-1)/2, n-1]` values
- Non-leaf nodes have `(n/2, n]` pointers
- *Know how to search, insert, delete*

Hashing

- Efficient equality queries
- Not for ordering or ranges

Static Hashing

- Buckets for each hash value, with overflow buckets
- Problem: As database grows larger, performance degrades due to searching in overflow buckets
- Problem: Large space taken up in the beginning

Extendable Hashing

- Dynamic hashing
- Create more specific buckets as needed

Indexing in SQL

```
CREATE (UNIQUE) INDEX index_name ON relation_name
index_type -- USING {BTREE | HASH}

DROP INDEX index_name;
```