

1 Introduction / DBMS Theory

Basic Concepts
Data: values representing real-world facts; their meaning depends on semantics.
Database: an organized collection of related data about a particular domain.
DBMS: programs that store, access and manage data; an interface between users/applications and the database.
Data model: concepts describing data, relationships, semantics and constraints.

Why Use a DBMS?

Drawbacks of file-based systems:

- (1) **Difficult access:** retrieving data may require custom programs or scanning entire files.
- (2) **Data isolation:** data scattered across files in different formats are difficult to combine.
- (3) **Integrity problems:** enforcing consistency constraints separately in every application is difficult.
- (4) **Redundancy / inconsistency:** duplicate data waste storage; updating only some copies causes disagreement.
- (5) **Concurrent access:** simultaneous updates can interfere, e.g. a lost update.
- (6) **Atomicity problems:** a failure may leave an operation partly completed, e.g. money deducted but not deposited. Atomicity requires all or nothing.

Abstraction Levels

- (1) **Physical:** how data are stored; files, indexes, pointers, compression.
- (2) **Logical:** what data are stored and how they are related, independent of storage details.
- (3) **View:** only the part of the database relevant to a user; different users may have different views.

Data Independence, Schema and Instance

Physical data independence: changes to physical storage need not change the logical schema or applications.
Schema: the database structure.
Instance: actual contents at a particular time.

Database Languages

DDL (Data Definition Language): defines or changes schema, e.g. CREATE, ALTER, DROP.
DML (Data Manipulation Language): retrieves or modifies data, e.g. SELECT, INSERT, UPDATE, DELETE.

System Architecture

Query processor

- (1) **DDL interpreter:** interprets schema definitions and updates the data dictionary.
- (2) **DML compiler:** translates queries into evaluation plans; optimization chooses a low-cost plan.
- (3) **Query evaluation engine:** executes the plan's low-level instructions.
- Storage manager:** interface between low-level stored data and applications/queries.

- (1) **File manager:** allocates disk space and organizes stored data structures.
- (2) **Transaction manager:** coordinates concurrent transactions and handles failures to preserve consistency.
- (3) **Authorization / integrity manager:** checks access rights and integrity constraints.
- (4) **Buffer manager:** fetches disk data into memory and decides what to cache.

Disk storage

- (1) **Data files:** actual database data.
- (2) **Data dictionary:** metadata, e.g. tables, attributes and constraints.
- (3) **Indexes:** structures for fast record access.

2 Entity Relationship Model

ER-Diagram Notation

Entity	rectangle
Attribute	ellipse
Primary Key	ellipse with underline
Composite Attribute	ellipse with sub-ellipses
Derived Attribute	dashed ellipse
Multivalued Attribute	double ellipse
Relationship	diamond
Role	text on line next to relationship
	useful for unary relationships

Mapping Cardinality

One	arrow
Many	no arrow

Participation

Total	double line
Partial	single line

Weak Entity

Weak Entity Set	double rectangle
Identifying Relationship	double diamond
Discriminator	dashed underline

Specialization	
Specialization	upside-down triangle with "ISA"
Disjoint	"disjoint" next to triangle
Overlapping	--
Total Specialization	double line from "parent"
Partial Specialization	single line

Keys

Super key:

a set of 1+ attributes that uniquely identifies a record

Candidate key:

a minimal super key

Primary key:

a candidate key chosen to identify records

Primary Key ⊆ Candidate Key ⊆ Super Key

Foreign key:

an attribute in a relation that is a primary key in another relation

Weak Entity Relationship

The weak entity set must have **total participation** (ensures identifying entity exists), **M-to-1/1-to-1** relationship with the identifying entity set (ensures unique identification).

PK(Weak Entity)=PK(Identifying)+Discriminator

ER-Diagrams to Schemas

Regular entity & attribute: entity → table; attribute → column

Primary key: underline attribute in schema

Composite attribute: dot-access path to leaf component

Derived attribute: not included in schema

Multivalued attribute: separate entity, with FK to owning entity

Weak entity set: FK to identifying entity set

Relationships:

M-to-M: relationship → table; FKs to participating entities

1-to-M/M-to-1: FK in M-side entity to 1-side entity

1-to-1: FK in either entity to the other entity

Specialization:

- (1) higher entity → table; lower entity → table; PK of higher entity → FK in lower entity. [+ve]: no redundancy; [-ve]: less efficient retrieving
- (2) lower entity table has all inherited attributes; PK of higher entity → FK in lower entity. [+ve]: more efficient retrieving; [-ve]: redundancy

3 SQL

DDL

--## create a table ##--

```
CREATE TABLE TableName (
    col1 datatype1 UNIQUE,
    col2 datatype2 NOT NULL,
    ...
    PRIMARY KEY (col1, ...), -- PK -> NOT NULL
    FOREIGN KEY (col2, ...)
        REFERENCES OtherTable(col1, ...)
);
```

--## common datatypes ##--

```
-- numeric: INT, SMALLINT, BIGINT, DECIMAL(p,s),
--          NUMERIC(p,s), REAL, DOUBLE PRECISION
-- character: CHAR(n), VARCHAR(n), TEXT
-- date/time: DATE, TIME, TIMESTAMP
```

--## delete a table ##--

DROP TABLE TableName; -- dependent FKs may block

--## modify a table ##--

```
ALTER TABLE TableName ADD newCol datatype;
ALTER TABLE TableName DROP colToDelete;
ALTER TABLE TableName ADD PRIMARY KEY (col1, ...);
ALTER TABLE TableName ADD FOREIGN KEY (col2, ...)
    REFERENCES OtherTable(col1, ...);
```

DML

--## insert a row ##--

```
INSERT INTO TableName VALUES
    (val1, val2, ...), -- one row
    (val1, val2, ...),
    ...;
```

--## load from a file ##--

LOAD DATA LOCAL INFILE 'file.txt'

INTO Table Name

FIELDS TERMINATED BY ','

LINEs TERMINATED BY '\n';

--## delete row(s) ##--

```
DELETE FROM TableName; -- delete all rows
DELETE FROM TableName -- delete rows conditionally
WHERE condition;
```

--## update row(s) ##--

```
UPDATE TableName SET col1 = val1, col2 = val2, ...
WHERE condition; -- update rows conditionally
UPDATE TableName SET col1 = col1 * 2; -- arithmetic
```

--## case statement ##--

```
UPDATE TableName SET col1 = CASE
    WHEN condition1 THEN val1
    WHEN condition2 THEN val2
    ELSE val3
END; -- CASE can be used elsewhere, e.g. SELECT
```

--## query rows ##--

```
SELECT col1, col2, ... FROM TableName;
SELECT * FROM TableName;
SELECT * FROM TableName WHERE condition;
```

--## string matching ##--

```
SELECT * FROM T WHERE col LIKE 'hello';
-- matches 'hello' exactly
SELECT * FROM T WHERE col LIKE 'hello%';
-- matches 'hello', 'hello world', ...
SELECT * FROM T WHERE col LIKE 'h_llo';
-- matches 'hello', 'hallo', 'hxllo', ...
```

Query

```
--## select / alias / group / sort ##--
SELECT DISTINCT dept, COUNT(*) AS n
FROM Employee E
WHERE salary > 0 -- filter rows
GROUP BY dept
HAVING COUNT(*) >= 2 -- filter groups
ORDER BY n DESC, dept ASC; -- default ASC
-- order: FROM -> WHERE -> GROUP BY -> HAVING
-- -> SELECT/DISTINCT -> ORDER BY
```

--## aggregate ##--

```
SELECT SUM(x), AVG(x), MIN(x), MAX(x),
    COUNT(*), COUNT(x), COUNT(DISTINCT x)
FROM T;
-- COUNT(*): rows; others: ignore NULL
-- empty input: COUNT -> 0; others -> NULL
SELECT SUM(CASE WHEN x > 0 THEN x ELSE 0 END)
FROM T; -- conditional sum
```

--## join ##--

```
SELECT A.*, B.* FROM A
INNER JOIN B ON A.id = B.id; -- matched pairs
-- LEFT/RIGHT JOIN: keep all left/right rows
-- unmatched side -> NULL; CROSS JOIN: m*n rows
SELECT A.id, COUNT(B.id) FROM A
LEFT JOIN B ON A.id = B.id
GROUP BY A.id; -- no matches -> 0, not COUNT(*)
```

--## set operations ##--

```
SELECT id FROM A UNION SELECT id FROM B;
-- UNION: either; INTERSECT: both; EXCEPT: A-B
-- same #columns + compatible types
-- duplicates removed; UNION ALL keeps them
```

Nested Query

--## membership ##--

```
SELECT * FROM Employee
WHERE dept IN (SELECT id FROM Dept
    WHERE budget > 100);
-- IN / NOT IN: member / not member
```

--## scalar: 1 column, <=1 row ##--

```
SELECT * FROM Employee
WHERE salary > (SELECT AVG(salary) FROM Employee);
-- 0 rows > NULL; >1 row -> error
```

--## correlated: uses outer alias ##--

```
SELECT D.id, (SELECT COUNT(*) FROM Employee E
    WHERE E.dept = D.id) AS n
FROM Dept D; -- no matches -> 0
```

--## exists / not exists ##--

```
SELECT D.* FROM Dept D
WHERE EXISTS (SELECT 1 FROM Employee E
    WHERE E.dept = D.id);
-- EXISTS: >=1 row; NOT EXISTS: 0 rows
-- EXISTS(SELECT COUNT(*) ...) -> true
-- (without GROUP BY/HAVING)
```

--## all / some (any) ##--

```
SELECT * FROM Employee
WHERE salary >= ALL (SELECT salary FROM Employee);
-- ALL: every; SOME/ANY: at least one
-- empty set: ALL -> true; ANY -> false
-- = ANY: IN; <> ALL: NOT IN
```

--## maximum: keep ties ##--

```
SELECT * FROM Employee
WHERE salary = (SELECT MAX(salary) FROM Employee);
-- MAX ignores NULL; ALL does not
-- choose population: global vs eligible only
```

--## derived table: max of group counts ##--

```
SELECT MAX(T.n) FROM
    (SELECT dept, COUNT(*) AS n FROM Employee
    GROUP BY dept) AS T;
```

NULL & UNKNOWN

```
SELECT * FROM T WHERE x IS NULL; -- not = NULL
SELECT * FROM T WHERE x IS NOT NULL;
-- arithmetic with NULL -> NULL
-- comparison with NULL -> UNKNOWN (U)
-- WHERE/HAVING: only TRUE survives
-- OR AND NOT
-- T U F T U F
-- T|T T T|T U F T|F
-- U|T U U U|U F U|U
-- F|T U F F|F F F|F
```

Useful Patterns

```
--## every child: x > 0 (NULL fails) ##--
SELECT P.* FROM Parent P
WHERE NOT EXISTS (SELECT 1 FROM Child C
    WHERE C.pid = P.id
    AND (C.x <= 0 OR C.x IS NULL));
-- empty set passes; add EXISTS if nonempty
```

```
--## every required item has a match ##--
SELECT P.* FROM Person P
WHERE NOT EXISTS (SELECT 1 FROM Required R
    WHERE NOT EXISTS (SELECT 1 FROM Match M
    WHERE M.pid = P.id AND M.rid = R.id));
```

```
--## exactly k distinct items ##--
SELECT pid FROM Match GROUP BY pid
HAVING COUNT(DISTINCT rid) = 2;
-- joins may duplicate rows: count IDs, not rows
```

View / Authorization / Assertion

--## view ##--

CREATE VIEW V AS SELECT id, name FROM Employee;

--## privileges / roles ##--

```
GRANT SELECT ON TableName TO username;
GRANT UPDATE(col) ON TableName TO roleName;
REVOKE SELECT ON TableName FROM username;
CREATE ROLE roleName;
GRANT roleName TO username;
```

--## assertion (standard SQL; not MySQL) ##--

```
CREATE ASSERTION AssertionName CHECK (
    (SELECT COUNT(*) FROM Dept) <=
    (SELECT COUNT(*) FROM Employee)
); -- DB-wide condition; check on updates
```