DBMS: Database management system

Problem with files vs DBMS:

* Data isolation: multiple files and formats
* Data redundancy and inconsistency: duplication of information in different files.
* Difficulty in accessing data: need to write a new program to carry out each task.
* Integrity enforcement: Constraints become buried in program code rather than being stated explicitly. Hard to add new constraints without changing code base.
* Atomicity of updates: Failure may leave database in inconsistent state.
* Concurrent access by multiple users: uncontrolled access may lead to inconsistencies.
* Lack of Safety/security: hard to provide role-based authentications and authorization, hard to limit access to some data, crashes can leave teh system in an inconsistent state.

DBMS pros:

* Location/format independence
* Integration without visible redundancy
* Programs use uniform logical representation
* Different views by different programs are cleanly supported
* Concurrent updates are atomic, consistent, isolated and durable.
* Fault-resilience and seamless recovery.
* Role-based authentication and authorization.

DBMS levels:

* Physical level: how a record is stored as physical object.
* Logical level: how representations of real-world concepts are stored in database, and their relationship
* View level: how applications retrieve the data they need, hide data type details, hide information for security/privacy purposes.

Objection-relational:

* Extends the relational model by including object-based notions and constructed
* allow attributes of tuples to have complex type, including nested relations
* preserve relational foundations while extending modelling power
* provides upward/forward compatibility with existing relational database

XML extensible markup language:

* Ability to specify new tags, and create nested tag structures made XML a great way to exchange data not just documents.

Newest data models:

* Document based.
* Usually: key-value pairs, column-family mode, graph-based model.
* they are APIs into a distributable data stores.

Schema:

* Defines the logical structure of the database (Similar to types and variables) E.g. customers, accounts, their properties and relationship between them

Instances:

* is the actual content of the database at a particular point in time. (value of variable)

Physical vs logical schema:

* Physical:
  + Defines the design of database structure at physical level
  + Records, files, indices, compressions… etc
* Logical:
  + Defines the design of database structure at logical level
  + Tables, columns logical dependences, etc…

Logical data model come with three or more languages:

* DDL: data definition language - define data without constraining how it could be used
* QLS: query languages
* DMLs: data manipulation languages

Why there are three or more languages in logical data model:

To enforce different versions of the principle of separation of concerns: defining and using, retrieving and manipulation, describing what and telling how, programmatic paradigm and environments.

What does DLL serve?  
Define data without constraining how it could be used.

What QL + DML serve?

Constraining the expressive power that is available allow reduction in complexity to be efficiently used by system.

Declaratively specified query: states which data is needed without stating how to obtain it.

Procedurally specified query: describes how to obtain the data, in precise steps in precise order.

-----

**DDL:**

**create relation:**

CREATE TABLE <name> (

<elementName> <type>,

<elementName2> <type>,

PRIMARY KEY (elementName2)

);

Types:

INTEGER // INT

FLOAT // REAL

CHAR(n) // Fixed string of size n characters

VARCHAR(n) // VARCHAR2(n) for oracle, variable length of strings up to n characters

DATE // ‘yyyy-mm-dd’

TIME // ‘hh:mm:ss’

**Single quote to represent strings.**

**Primary key takes unique values (no two tuples can have same value for primary key)**

**No attribute of PRIMARY KEY is ever NULL**

**UNIQUE can may be NULL**

**Can only be one PRIMARY Key declaration for a relation but several unique attributes.**

**remove relation:**

DROP TABLE <name>;

**to add to an existing attributes:**

ALTER TABLE <name>

ADD

<attribute name>

<attribute type>

//can add default value

DEFAULT ‘default value’;

**If we add without setting a DEFAULT it will be set to NULL in all rows.**

**To remove an existing attribute**

ALTER TABLE <name>

DROP <attribute name>;

**Foreign keys**

CONSTRAINT <constraint name> FOREIGN KEY (elementName)

REFERENCES ParentTable(parentElement)

ON [UPDATE | DELETE]   
 [SET NULL | CASCADE]

If we don’t specify ON, it will reject the update. cascade is updating the values

Referenced attributes must have been declared as primary or unique keys in parents table.

----

**DML:**

**To insert:**

INSERT INTO

<relation name> (attribute1, attribute2)

VALUES (value1, value2);

can also do

INSERT INTO <relation name>

( <subquery, e.g. SELECT x FROM y WHERE z> );

= equals, <> doesn’t equal

**To delete:**

DELETE FROM <relation>

WHERE EXISTS <condition>; // without where, everything will be deleted

**To change attribute:**

UPDATE <relation>

SET <list of attributes assignments>

WHERE <condition on tuples>;

-------

**Relational algebra**

Arity: degree (columns)

Cardinality: rows

Core RA:

Union: A u B

Difference: A \ B

Product: concatenates every tuple of one operand with every tuple of the other.

Selection: pick only certain rows from the operand

Projection: pick only certain columns from the operand

Renaming: Renaming[b->x, e->y](R) will rename every b to x and every e to y

Derivable:

Intersection A n B = ((AuB)\(A\B)) \ (B\A)

Joins

Divsion

**Selection in RA:**

S := σc(R)

c is the condition, referring to attributes of R

S contains the results of the selection

**Projection in RA:**

S := πL(R)

L list of attributes from schema of R

S is schema constructed of the distinct tuples of projection

**Product in RA:**

R3 :=R1 ×R2

Pair each tuple of R1 with each tuple of R2

Concatenation is the tuple of R3

**Renaming in RA:**

R2 := ρR2(A1, ..., An)(R1) \\ or R2(A1, ..., An):= R1

Assigns to the resulting relation the name R2 with attributes named A1,..An

**Join in RA:**

R3 := R1 ⋈θ R2

it is equivalent to taking the product, then applying the selection

R:=R1 ×R2

R3 := σθ(R)

**Natural Join in RA:**

R3 := R1 ⋈ R2

applies a projection that eliminates duplicate columns

**Extended projection:**

Using the projection operator,

S:=πA+B⟶C,A,A⟶A2(R)

will result in having 3 columns: C which contains the sum of A +B, A, and A2 which is just copy of A.

Why use bags in SQL:

because they allow duplicates, while sets require eliminating duplicates which is computationally expensive.

**Extended RA:**

**δ: duplicate elimination:**

R2 := δ(R1), expensive to implement

**τ: sorting:**

S:=τ(X,A),(Y,D) (R)

Sort x in ascending order and sort y in descending order, starting with x where A for ascending, and D for descending order

**γ: aggregation/grouping:**

Applies on entire columns of a table and produce a single result.

E.G:

COUNT : COUNT(tableName.columnName)

SUM

AVG

MIN

MAX

Example:  
S:=γA,B,AVG(C)->X (R)

Just project A and B from R, get the average of C and rename the column to X and store it in S relation

**⟗, ⟖, ⟕: outer join, left outer join, right outer join:**

it is a join where if we join R with S and there are tuples in R that don’t exist in S then we padde the S column with NULL value, similarly for S. This is full outer join

left outer join only padd left tuples. right outer join only pad with right tuples.

Can set condition on the join

**⋉, ⋊: left semijoin, right semijoin**

left semijoin join on the common table but then only project the elements from first table.

right is the opposite

**⊳: antijoin**

It is the complement of a left semijoin. basically showing the tuples left semijoin didn’t show

**÷: division**

fkme

-----

**SQL**

**DISTINCT** removes duplicates from a SELECT result

**UNION** can combine the result of two select query must have   
(SELECT X  
<expression>)

UNION

(SELECT Y

<EXPRESSION>)

;

**INTERSECT** is the intersection of result of two select query, syntax same as Union

**EXCEPT (or MINUS in oracle)** finds the results that in the first SELECT statement but not in the SECOND.

Union, INTERSECT and EXCEPT eliminates duplicates. to not eliminate duplicates we use UNION ALL, INTERSECT ALL, EXCEPT ALL

**NATURAL JOIN:**matches tuples with the same values for all common attributes and retrains only one copy fo each common column

**Renaming:** using AS. <old-name> AS <new-name>

**LIKE:** use patterns that are specified using three special characters:  
 (%) which matches any substring   
 (\_) matches any character

(\) escape character

e.g ‘\_ \_ \_’ matches strings of exactly 3 characters. ‘ \_ \_ % ‘ matches string of at least 2 character

Finding tuples that of value null:

SELECT <name>

FROM <tableName>

WHERE x IS NULL;

---------------

* An identifying relationship is one that relates a weak entity type to its owner, and always has a total participation constraint
* If participation is total, then every entity in the total entity set must participate in some relationship instance
* ER diagram is read from left to right and from top to bottom (if possible).

----

**Embedding:** mixing sql statements in source code statement of a host language.

**impedance mismatch:** problem with embedding happen due to mix different programming paradigms, SQL is declarative language whereas common high level language are procedural.

**PL/SQL statements:**

**optional declaration part:** variable, constants, cursors, and exceptions are defined and initialised

**Mandatory executable part**: Variable are manipulated

**Optional exception part:** handle any exceptions raised during execution

[DECLARE -- optional

-- declarations

]

BEGIN -- mandatory

-- executable statements

[EXCEPTION -- optional

-- exception handlers

]

END;

Example:  
DECLARE

amount NUMBER := 0;

v\_error\_code NUMBER;

v\_error\_message VARCHAR2(255);

BEGIN

SELECT COUNT(\*) INTO amount

FROM Student

WHERE Student.class = 3;

DBMS\_OUTPUT.PUT\_LINE(amount);

EXCEPTION

WHEN OTHERS THEN ROLLBACK;

v\_error\_code := SQLCODE

v\_error\_message := SQLERRM

INSERT INTO t\_errors VALUES ( v\_error\_code, v\_error\_message);

END;

Variable declaration:

variable\_name

[CONSTANT]

datatype

[NOT NULL]

[:= | DEFAULT initial\_value];

Having %TYPE after the variable declares to have same type as an attribute in the database schema

E.G empId Employee.employeeId%TYPE NOT NULL;

%ROWTYPE declares to have the same type as the rows of a relation in the database schema

Assigning to a variable can be done in two ways:

assignment operator :=

Using keyword INTO to capture the result of SELECT or FETCH

**IF statement:**

IF condition1 THEN {statement list}

ELSIF condition2 THEN {statement list} -- optional

ELSE END IF;

{statement list} -- optional

**Case**:

CASE [ expression ]

{WHEN condition THEN result}

WHEN a < b THEN ‘HELLO’

WHEN owner=’USR’ THEN ‘Owner is user’

ELSE ‘Owner is not usr’

END CASE;

**Loop:**

[labelName:]

LOOP

{statement list}

END LOOP [labelName];

[labelName:]

LOOP

{statement list}

EXIT [labelName] [WHEN condition]

END LOOP [labelName];

WHILE condition

LOOP

{statement list}

END LOOP;

FOR variable IN [REVERSE] lowerBound .. upperBound

LOOP

{statement list}

END LOOP;

Exception raising and handling

DECLARE

vCount NUMBER;

vProj Project.id%TYPE := '31415';

-- No Type A project can have less than 100 employees allocated

e\_not\_enough\_employees EXCEPTION;

PRAGMA EXCEPTION\_INIT(e\_not\_enough\_employees, -1);

BEGIN

SELECT COUNT(\*) INTO vCount

FROM AllocatedTo E, Project P

WHERE P.id = vProj AND E.projId = P.id AND P.type = 'A';

IF vCount < 100

THEN RAISE e\_not\_enough\_employees;

END IF;

EXCEPTION

WHEN e\_not\_enough\_employees THEN

DBMS\_OUTPUT.PUT\_LINE('Type A project' || vProj || 'has less than 100 employees');

END;

Create a procedure:

A cursor allows the rows of a query result to be accessed one at a time

CREATE OR REPLACE PROCEDURE ProcedureName (parameter IN Attribute.parameter%TYPE)

AS

-- optional declarations

v\_variable Attribute.something%TYPE;

CURSOR attributeCursor (parameters) IS

SELECT x

FROM Y

WHERE z

ORDER BY s;

BEGIN

OPEN attributeCursor(parameter)

-- loop to fetch each row in result table

LOOP  
 FETCH attributeCursor INTO x\_attribute, y\_attribute, ... ;

EXIT WHEN attributeCursor%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE(x\_attribute);

END LOOp;

if attributeCursor%ISOPEN

THEN CLOS attributeCursor;

END IF;

EXECUTE ProcedureName(parameter);

**Function:**

Different between function and procedure: function must return a result, RETURN <parameter>;

parameter types:  
IN: specifies that the parameter is used as input only

OUT: which specifies that the parameter is used as an output only

IN OUT: which allows the parameter to be both input and output

E.G

CREATE OR REPLACE FUNCTION FindCourse

(cname IN VARCHAR2) RETURN NUMBER

AS

cnumber NUMBER;

CURSOR c IS

SELECT course\_number

FROM courses

WHERE course\_name = cname;

BEGIN

OPEN c;

FETCH c INTO cnumber;

IF c%NOTFOUND

THEN cnumber := 9999;

END IF;

CLOSE c;

RETURN cnumber;

END;

To invoke:  
SELECT course\_name, FindCourse(course\_name) AS course\_id

FROM courses

WHERE subject = 'Mathematics';

Packages:

Can create packages to abstract the design

e.g.

CREATE OR REPLACE PACKAGE StaffPackage

AS

PROCEDURE PropertiesForStaff (v\_staffNo PropertyForRent.staffNo%TYPE);

-- others would come in here

END PropertiesForStaffPackage

-- an example invocation

StaffPackage.PropertiesForStaff('SG14')

Create triggers:

CREATE TRIGGER triggerName

(BEFORE | AFTER | INSTEAD OF)

(INSERT | DELETE | UPDATE [OF triggerColumnList])

ON tableName

[REFERENCING (OLD | NEW) AS (oldName | newName)]

[FOR EACH (ROW | STATEMENT)]

[WHEN condition]

triggerAction

;

----

Transactions: series of actions carried out by the user or application which reads or updates contents of a database.

ACID properties:

**Atomicity**: transaction must be either complete all its work or else it must be as if no partial work was ever done.

**Consistency**: Transaction must transform database from one consistent state to another

**Isolation**: Partial effects of incomplete transactions should not be visible to other transactions.

**Durability**: Effects of a committed transaction are permanent and must not be lost because of later failure.

**Uncommitted dependency problem:** occurs when one transaction can see the intermediate results of another transaction before it has committed

**Inconsistent analysis problem:** occurs when a transaction T reads several values but another transaction T’ updates some of them during execution of T.

**Schedule:** is sequence of read and write by a set of concurrent transactions.

**Shared lock:** can read but not update item

**Exclusive lock:** can read and write to an item.

**Two-pbase locking:**

* growing phase: transaction must acquire all locks it needs but can’t release any locks.
* shrinking phase: transaction can release locks but can’t acquire any new locks

**cascading rollback:** when transactions are depending on previous transaction which rolledback so they have to rollback as well. to fix it leave release of all locks until end of transaction which is Rigorous 2PL.

**Deadlock prevention:**

* wait-die: only an older transaction can wait for a younger one, otherwise the transaction is aborted and restarted with same timestamp (it will eventually become the older)
* wound-wait: only younger transaction can wait for an older one. that if an older transaction requests a lock held by a younger one, the younger one is aborted.

**Granularity:** Is the size of data items chosen as unit of protection by a concurrency control protocol.

**Checkpoint:** point of synchronization between database and log file

if database is damaged:

we need to restore last backup copy of the database, reapply the updates of committed transaction using the log file.

if database is inconsisntant: we jsue need to undo the changes that caused inconsistency. sundo some transaction

Recovery:

shadow paging: maintain two tables: current page and a shadow page.

the shadow page never changed, but current page is, if failed use the shadow page in case of failure.