Data Independence, Integrity & Security

- Need for Data Independence was main motivation for DB technology
- However, tradeoff between maximising high degree of independence & maximising overall performance/efficiency (why?)
- To maximize data independence we want to bind references as late as possible - at query execution time.

Mixed Levels of Binding

- If bind at compilation time => changes to internal storage will require recompilation of query
- However, if bind at runtime => more computing time taken to resolve addresses etc.
- Typically DBs adopt a mixed approach to binding, with some taking place at compilation time, some when files are accessed the first time and the remainder at command execution time

Integrity & Security

- Integrity and Security are related but they are <u>not</u> the same thing!
- Integrity is concerned with ***accidental*** corruption of the DB where as Security is concerned with ***deliberate*** corruption of DB.
- Several types of integrity constraints have been presented: key, entity & referential constraints. These are part of the relational structure itself

=>having defined a primary key or foreign key of a relation, the system can enforce the relevant constraints

Semantic or Explicit Constraints

- However there are a large & complex set of constraints called ***semantic or explicit constraints*** which are dictated by the rules of the applications being modelled in the DB

  E.g. a student is only allowed to take 4 courses

- These constraints are called ***explicit*** to differentiate them from the constraints implied (***implicit***) in the relational model itself.
- The word ***semantic*** is also used to describe these constraints as they are concerned with the meaning of the data (as opposed to the structure or syntax)

Referential Integrity Constraints & Foreign Keys

Entity Constraint: no part of a primary key can be null.
But foreign keys can be null (depending on applic. req'ments)

STUDENT (<u>student_number</u>, student_name, student_address)
RESULT (<u>course_number, student_number</u>, grade)
COURSE (<u>course_number</u>, course_title, lecturer)

- course_number and student_number in RESULT are foreign keys of course_number and student_number in relations COURSE and STUDENT
- However neither course_number nor student_number are permitted to be NULL in RESULT since they are part of the primary key for RESULT

Referential Integrity Constraints (2)

PATIENT (patient_number, name, address, gp)
GPLIST (gpname, gpaddress, gptelno)

- gp in PATIENT is a foreign key of gpname in GPLIST, therefore the values entered for PATIENT.gp are constrained to those currently stored in GPLIST.gpname
- But its possible (in the real world) that a patient doesn't have a GP or as an accident & emergency patient, cannot provide gp information. Hence it would be logical and consistent in the mini world (of the DB) to allow PATIENT.gpname to contain NULL values.

Referential Integrity Constraints (3)

- Then, when defining that an attribute is a foreign key, we must also specify whether or not the foreign key is allowed to contain NULLS.
- NOTE: if a foreign key is allowed to contain NULLS, then in the case of a composite foreign key, either all the attributes (which make up the foreign key) are NULL or none of them!

Foreign Key Update or Delete

<u>What happens to a foreign key when the primary key it references is updated or deleted</u> ?

- The decision is dictated by the rules of the mini-world.
- In the RESULT relation, the deletion of a student from the STUDENT relation would presumably cascade to the RESULTS relation to delete all the student's RESULT tuples. (e.g. a student drops out of university)
- The deletion of a course from COURSE relation would presumably also cascade to the deletion of the RESULT tuples which referenced that course. (i.e. if the course no longer exists then students can't get a grade in it!).

Foreign Key Update or Delete (2)

- The alternatives to cascading of updates or deletes are:
  - To deny the update or delete as long as foreign key references exist
  - To set the corresponding foreign key to NULL
  - set the corresponding foreign key to some default value

Referential Integrity Rule

- The referential Integrity Rule can be defined in terms of foreign keys as:

  ‘A relation is not allowed to contain any unmatched foreign keys’

Explicit Constraints

- Most RDBMSs only provide limited form of explicit constraints
- SQL provides a ASSERTIONS for specifying constraints
- Not implemented yet on many RDBMSs

Table Constraints (in SQL-92)

- Constraints can be specified to restrict the attributes in one or more tables e.g. NOT NULL, UNIQUE
- Complex constraints can be specified using the CHECK clause
- CHECK clause(s) are specified within the CREATE TABLE statement

```
CREATE TABLE  movie_titles (
title   CHAR(30) NOT NULL,
movie_type  CHAR(10),
CONSTRAINT check_movie_type
CHECK (movie_type IN (‘Horror’, ‘Action’,‘Other’)));
```

ASSERTIONS - in SQL 92

- An assertion is a stand-alone constraint in a schema and is normally used to specify a restriction that affects more than one table
- Table constraints (using CHECK) can be used to specify multiple table constraints however it is better practice to use assertions which can be defined outside a single TABLE definition because:

(i) More natural to define ASSERTIONS separately

(ii)Table constraints are only evaluated if and only if the table to which it is attached has some data unlike assertions which are required to be true regardless of whether a table is empty or not

ASSERTIONS - Example

Example:
music_title (title, music_type, our_cost)
movies_titles (title, movie_type, our_cost)

```
CREATE ASSERTION max_inv
CHECK ( (SELECT SUM (our_cost) FROM movies_titles)
       + (SELECT SUM (our_cost) FROM music_title)
       < 500000);
```

The above assertion would ensure that the total value of stock of music articles and movie videos is below 500,000

ASSERTION Syntax

The general form of the ASSERTION command is:

```
CREATE ASSERTION <assertion-name>
CHECK ( <search-condition> )
```

- **`ASSERTION`** evaluated by integrity subsystem - if false then assert is violated and DB operation is not allowed
- **`ASSERTION`**s are placed on relations and are evaluated before an operation is allowed on those relations
- **`ASSERTION`**s define valid states of a DB
- **`ASSERTIONs`** are actually stored as rows in the ASSERTIONS table which is part of the system catalog

Evaluation of ASSERTIONS

- Assertions are ‘effectively checked’ at the end of each SQL statement (remember a transaction can be more than one SQL statement !)
- Assertion evaluation can be specified ‘deferred’ until the end of a transaction but is always evaluated prior to the completion of a transaction
- If an assertion fails, the DBMS returns an error messsage and the SQL statement is rejected

Active Databases

- Triggers or Event-Condition-Action (ECA) rules allow constraints to be checked on some specified events and actions (SQL operations) to be invoked
- Triggers are only tested when certain events occur e.g. insert, update etc
- Instead of immediately preventing the event that woke it up, a trigger test a specified condition. If the condition does not hold, then nothing else associated with the trigger happens in response to the event
- If the condition of the is satisfied, actions associated with the trigger is performed by the DBMS

Trigger Constraints

Triggers are not part of SQL-92 but are part of the evolving new SQL standard (called SQL3)

General form

```
CREATE TRIGGER trigger-name time event
 on tablenames [referencing] action
```

Trigger Contsraints (2)

- **`Trigger_Name`** - is name of trigger constraint
- **`time`**- indicates whether action is to be fired BEFORE or AFTER the specified event
- **`event`** - is either INSERT, DELETE or UPDATE
- **`table_name`** - identifies the table which the DBMS needs to watch for a triggering event
- **`referencing`** - this can only be specified if the event causing the triggering is an UPDATE. The clause correlates the values of columns in a row being updated before that update occurs and after that update occurs
- **`action`** - specifies the actions the DBMS is required to take whenever the trigger fires

Example Trigger Constraint

`MovieExec(name, address, cert#,netWorth)`

- We can specify that some SQL statements should be triggered when netWorth of an movie executive is updated

```
CREATE TRIGGER NetWorthTrigger
AFTER UPDATE OF netWorth ON MovieExec
REFERENCING
  OLD AS OldTuple,
  NEW AS NewTuple
WHEN (OldTuple.netWorth > NewTuple.netWorth)
     UPDATE MovieExec
     SET netWorth = OldTuple.netWorth
     WHETE cert# = NewTuple.cert#
FOR EACH ROW
```

Referencing Syntax

The format of referencing is:

```
REFERENCING OLD [AS] old-correlation-
  name
[NEW [AS] new-correlation-name]
```

[ ] indicate optional parts of statement

ACTION Syntax

The action syntax is:

```
[WHEN (search-condition) ]
(statement [ , statement ] … )
  granularity
```

search-condition - is a predicate statement which can include nested SELECT SQL statements

`statement` - is an SQL statement

`granularity` indicates whether the asction is to be executed **`FOR EACH ROW`** that the condition occurs or **`FOR EACH  STATEMENT`**

Static & Dynamic Constraints

- constraints can also be classified as static or dynamic (transition)
- *static constraints* specify *legal DB states*
- *dynamic constraints* describe *legitimate transitions* from one DB state to another

Security

- ensuring security for large DBs is an enormous task and imposes an overhead on all users of the system
- most advances in the development of secure systems come from the military world
- many different issues involved - legal, social, ethical etc.
- most countries have Data Protection Legislation which requires owners of personal info to take reasonable precautions to ensure that unauthorised people do not gain access to the data

Security (2)

- many security issues are outside control of DBMS; they are part of an organisation's *security policy*
- basic security mechanism in DBMS is to control access to data objects

e.g. User A is allowed to read objects X and Y, but is only allowed to update X

Granularity

- degree of granularity at which security control can be applied is important

  e.g. can range from individual attribute, to entire DB
- finer granularity allows more precise security control, nut involves much greater administrative overhead
- defaults are important in reducing administrative overhead

Security Policy

4 main types of security policy:

- need to know
- maximised sharing
- open systems
- closed systems

Need to Know

- is a user requires read (or write) access to a certain set of data objects then they are granted appropriate rights to those objects alone

  e.g. if granule is a relation and user needs access to one attribute then under need to know they have automatic access to entire relation
- most widely used with fine granularity in high security environments

Policy - Maximizing Sharing

- at opposite end of security spectrum aimed at environments which wish to encourage data sharing
- only those parts of the DB which must protected are protected

Open Policy & Closed Policy

Open Policy

- default is that users have full access to all data
- access controls must be explicitly specified for any data to be protected
- disadvantage is that accidental omission or deletion of security control makes confidential data publicly available

Closed Policy

- default is to deny access
- access privileges must be explicitly granted
- most widely used with need to know policy
- errors in rules will restrict rather than open up access

User profiles and authorisation matrices

- info on access privileges is called a *user profile* which are generally represented by authorisation matrix
- each entry A [*i*,*j*] specifies the set of operations which user *i* is allowed to perform on data object *j*.

	Data Objects			
USERS	**R1**	**R2**	**R3**	**R4**
A	**All**	**All**	**All**	**All**
B	**Select Update**	**Select**	**Select Delete**	**All**
C	**All**	**Select**	**Select Update**	**Select**
D	**Select**	**None**	**All**	**Select**
E	**All**	**None**	**Select Insert**	**Select Delete**

Types of DB Access Control

4 types of DB access control:

- content independent
- content dependent
- statistical control
- context dependent

Content Independent

- a user is allowed/not allowed access to a data object irrespective of the data content
- checking can be done at compile time as no DB access is required

Example

User A is allowed read access to the EMPLOYEE relation

Content dependent

- access to a data object depends on the content (value) of the object
- can only be checked at run-time

Example

Employee x is allowed to update the salary field of an employee provided the current salary is less than £15,000

Statistical control

Statistical control

- user is permitted to perform statistical operations such as SUM, AVERAGE etc. on data but not to access individual records

  e.g. census DBs, epidemiological DBs

Context Dependent

- Access depends on the context in which the request is being made

Example

User is only allowed to update the grade attribute of the RESULT relation is the user is the lecturer on the particular course

A user may update an employee's salary but only between 9 am and 5 pm and from a terminal located in the personnel department

Security facilities in SQL

- 2 basic security features:
  - views
  - authorisation rules

Views

- with the ANSI-SPARC architecture users can only access the data through views/external schemas
- hence access is automatically restricted to data within their view

Authorization Rules

- GRANT command is used to give users access rights; general format:

```
GRANT operation ON object TO user;
```

- operation can be SELECT, INSERT, DELETE, UPDATE, REFERENCES or ALL PRIVILEGES
- object can be TableName or ViewName
- In SQL-92 there are other objects in the DB for which access can be granted

Example of Authorization Rules (Grant Command)

Examples:

EMPLOYEE (name, ssn, bdate, address, sex, salary)

DEPARTMENT (dnumber, dnmase, mgrssn)

```
GRANT ALL PRIVILAGES On EMPLOYEE,
  DEPARTMENT TO USER1;
```

- USER1 has the right to use the GRANT option

```
GRANT INSERT, DELETE ON EMPLOYEE,
  DEPARTMENT TO USER2;
```

Example Authorization Rules (2)

```
GRANT SELECT ON EMPLOYEE, DEPARTMENT TO
USER3 WITH GRANT OPTION;
```

USER3 can propagate the privilege to other users

Suppose user3 then issues the command

```
GRANT SELECT ON EMPLOYEE TO USER4;
```

Privileges can also be revoked:

```
REVOKE SELECT ON EMPLOYEE FROM USER3
CASCADE;
```

DBMS will automatically revoke SELECT privilege on EMPLOYEE from USER4

If CASCADE not specified, USER4 would still have access to employee

Review

~~*Page under construction*~~

~~*Example*~~

- ~~The constraint that an employee's salary can only be increased, would be specified for the relation EMPLOYEE (emp#, name, address, dept, jobtitle, salary) *as:*~~

```
CREATE ASSERTION PAY_RISE_CONSTRAINT
NEW employee.salary >OLD employee.salary;
```

- ~~new keywords - UPDATE OF, OLD and NEW - have been added to ASSERT syntax~~
- ~~Update indicates that constraint is to be checked prior to update operation. NEW refers to salary value after update and old refers to salary value prior to update~~
- ~~Note: This is not part of the SQL-92 standard~~