ISYS2120: Data & Information Management

Week 7B: Database Integrity

Alan Fekete

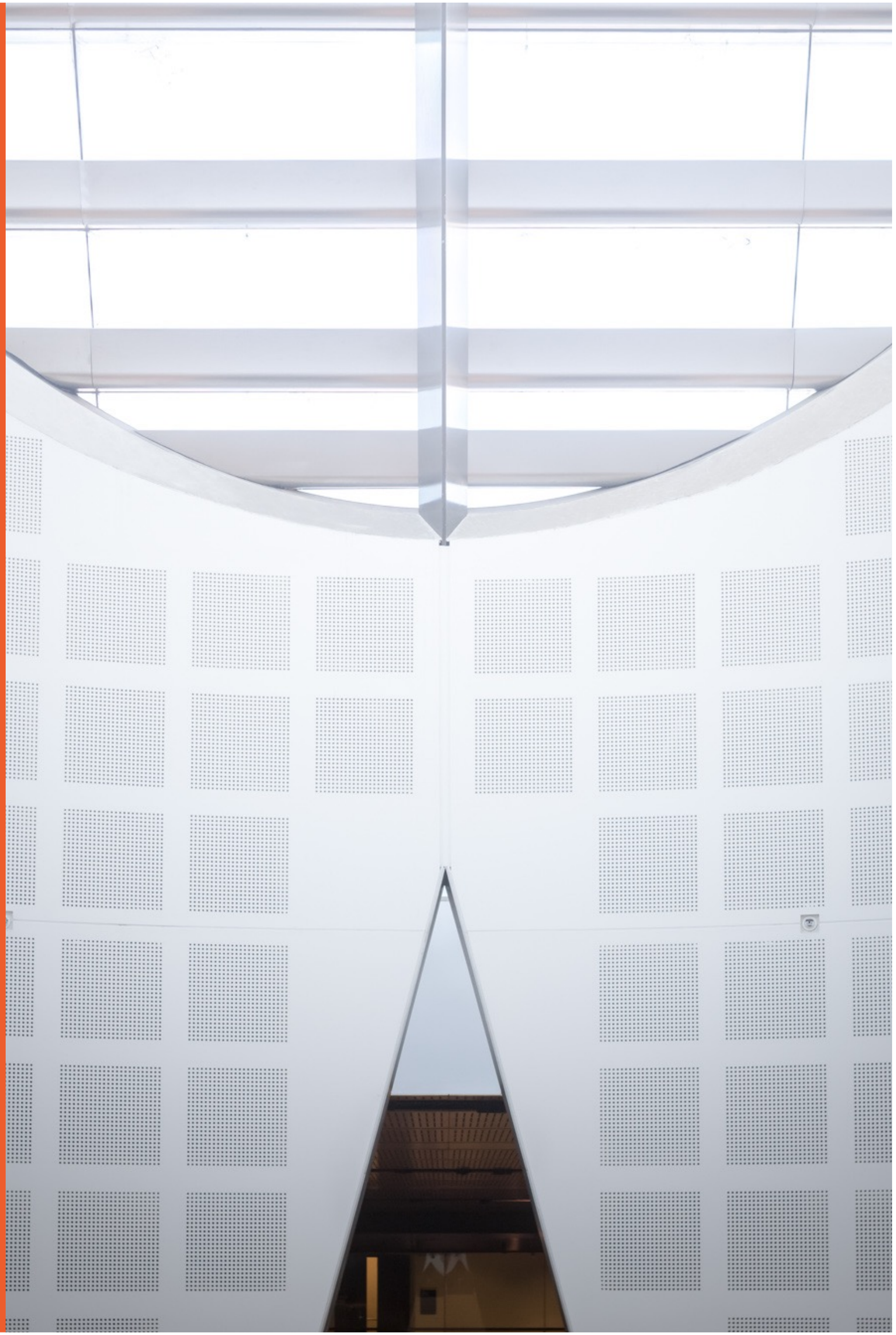
Cf. Kifer/Bernstein/Lewis – Chapter 3.2-3.3

Ramakrishnan/Gehrke – Chapter 5.7-5.9;

Silberschatz/Korth/Sudarshan – 4.2, 4.4, 5.3

Ullman/Widom – Chapter 7

THE UNIVERSITY OF SYDNEY



Semantic Integrity Constraints

- Recall: Our Objective
  - capture semantics of the domain in the database
  - ensuring that authorized changes to the database do not result in a loss of **data consistency**
  - guard against accidental damage to the database (avoid data entry errors)
- Advantages of a centralized, automatic mechanism to declare and enforce semantic integrity constraints:
  - Stored data is more faithful to real-world meaning
  - Declarations provide documentation of domain constraints
  - Easier application development, better maintainability

Examples of Integrity Constraints

- Each student ID must be unique.
- For every student, a name must be given.
- The only possible grades are either ‘F’, ‘P’, ‘C’, ‘D’, or ‘H’.
- Students can only enrol in a unit of studies that is offered in the semester.
- The sum of point values for the assessment tasks in a unit-offering must equal 100.
- Student year-level always increases or stays same; cannot go down

Integrity Constraint (IC)

- **Integrity Constraint (IC)**:
  condition that must be true for every instance of a database
  - A **legal** instance of a relation is one that satisfies all specified ICs
    - DBMS should never allow illegal instances….
- ICs are *specified* in the database schema
  - The database designer is responsible to ensure that the integrity constraints are not contradicting each other!
- ICs are *checked* when the database is modified
- Possible *reactions* if an IC is violated:
  - Undoing of the modification (return to previous state)
  - Execution of “maintenance” operations to make db legal again

Domain Constraints

- The most elementary form of an integrity constraint:
- Fields must be of right data domain
  - always enforced for values inserted in the database
  - Also: queries are tested to ensure that the comparisons make sense.
- SQL DDL allows domains of attributes to be restricted in the **create table** definition with the following clauses:
  - **DEFAULT** *default-value*
    default value for an attribute if its value is omitted in an insert stmnt.
  - **NOT NULL**
    attribute is not allowed to become NULL
  - ***NULL*** (note: not part of the SQL standard)
    the values for an attribute may be NULL (which is the default)

Example of Domain Constraints

```
CREATE TABLE Student
 (
    sid         INTEGER      PRIMARY KEY,
    name        VARCHAR(20) NOT NULL,
    gender      CHAR        NOT NULL,
    birthday    DATE         NULL,
    country     VARCHAR(20),
    level       INTEGER      DEFAULT 1
 );
```

Semantic:

sid is primary key of **Student**
name and **gender** must not be NULL
level will be 1 if not specified by an insert
all other attributes can be NULL (**birthday** and **country**)

Example:

```
INSERT INTO Student(sid,name,gender) VALUES (123,'James','M');
```

User-Defined Domains

- New domains can be created from existing data domains

  `CREATE DOMAIN` *`domain-name sql-data-type`*

- Example:

  create domain Dollars **numeric**(12,2)
  create domain Pounds **numeric**(12,2)

  cannot assign or compare a value of Dollars to a value of Pounds.

- Domains can be further restricted,e.g. with the **check** clause
  - E.g.: **create domain** Grade **char check**(value in ('F','P','C','D','H'))

- User-defined types with SQL:1999:

  `CREATE [DISTINCT] TYPE type-name AS sql-base-type`

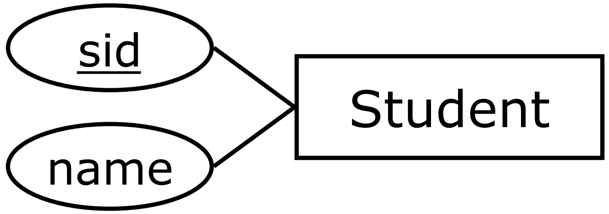
- PostgreSQL offers instead CREATE DOMAIN mechanism

Primary Key Constraints

- Recall
  - A set of fields is a key for a relation if :

    1. No two distinct tuples can have same values in all key attributes, and

    2. This is not true for any subset of the key.
- In SQL, we specify a primary key constraint using the **PRIMARY KEY** clause:




```
CREATE TABLE Student
(
    sid  INTEGER PRIMARY KEY,
    name VARCHAR(20)
);
```

- A primary key is automatically unique and NOT NULL
- Complex (multi-attribute) key: separate clause at end of **create table**

Foreign Keys & Referential Integrity

- **Recall**
  - **Foreign key :** a set of attributes in a relation that is used to `refer' to a tuple in a parent relation.
  - Must refer to a candidate key of the parent relation
  - Like a `logical pointer'

- **Referential Integrity**: for each tuple in the referring relation whose foreign key value is **X** either X is NULL, or **there must be a tuple in the referred relation whose primary key value is also X**
  - e.g. *sid* is a foreign key referring to Student:
    Enrolled(*sid*: integer, ucode: string, semester: string)
  - If all foreign key constraints are enforced, referential integrity is achieved, i.e., no dangling references

Foreign Keys in SQL

- Only students listed in the Students relation should be allowed to enroll for courses.

```
CREATE TABLE Enrolled
(   sid CHAR(10),  uos CHAR(8),  grade CHAR(2),
    PRIMARY KEY (sid,uos),
    FOREIGN KEY (sid) REFERENCES Student )
```

Student

sid	name	age	country
53666	Jones	19	AUS
53650	Smith	21	AUS
54541	Ha Tschi	20	CHN
54672	Loman	20	AUS

Enrolled

sid	uos	grade
53666	COMP5138	CR
53666	INFO4990	CR
53650	COMP5138	P
53666	SOFT4200	D
54221	INFO4990	F

??? Dangling reference

This state is NOT allowed by constraint

Enforcing Referential Integrity in SQL

- what happens on deletes and updates at the parent (referenced) table? SQL has four options in the referring table, for delete, and for update
  - Default is **NO ACTION** (delete/update is rejected if it will violate foreign key constraint)
  - **CASCADE** (also delete all tuples that refer to deleted tuple)
  - **SET NULL** (resets foreign key to NULL)
  - **SET DEFAULT** (sets foreign key value of referencing tuple to a default value)

```
CREATE TABLE Enrolled
(   sid CHAR(10),
    uos CHAR(8),
    grade CHAR(2),
    PRIMARY KEY (sid,uos),
    FOREIGN KEY (sid)
       REFERENCES Student
       ON DELETE CASCADE
       ON UPDATE NO ACTION )
```

refers to modifications at the parent table (Student)

Named Integrity Constraints

- Integrity constraints on more than one attribute?
- Also, a name for integrity constraint would be very useful for administration / maintenance…
- SQL:

  CONSTRAINT *name* **CHECK** **(** *semantic-condition* **)**

- SQL-92 standard allows use of subqueries to express constraint condition, but
  - subqueries in CHECKs are NOT SUPPORTED by either PostgreSQL or Oracle

Named Constraints Example

```
CREATE TABLE Assessment
(
   sid    INTEGER     REFERENCES Student,
   uos    VARCHAR(8)  REFERENCES UnitOfStudy,
   empid  INTEGER     REFERENCES Lecturer,
   mark   INTEGER,
   CONSTRAINT maxMarks CHECK (mark between 0 and 100),
   CONSTRAINT rightLecturer
        CHECK ( empid = (SELECT u.lecturer
                           FROM UnitOfStudy u
                          WHERE u.uos_code=uos) )
);
```

Note: The second constraint with a subquery is *not* supported by PostgreSQL

SQL: Naming Integrity Constraints

- The **CONSTRAINT** clause can be used to name all kinds of integrity constraints
- Example:

```
CREATE TABLE Enrolled
(
  sid       INTEGER,
  uos       VARCHAR(8),
  grade     CHAR(2),
  CONSTRAINT FK_sid_enrolled   FOREIGN KEY (sid)
                               REFERENCES Student
                               ON DELETE CASCADE,
  CONSTRAINT FK_cid_enrolled   FOREIGN KEY (uos)
                               REFERENCES UnitOfStudy
                               ON DELETE CASCADE,
  CONSTRAINT CK_grade_enrolled CHECK(grade in ('F',…)),
  CONSTRAINT PK_enrolled       PRIMARY KEY (sid,uos)
);
```

ALTER TABLE Statement

- Integrity constraints can be added, modified (only domain constraints), and removed from an existing schema using ALTER TABLE statements

  ALTER TABLE *table-name constraint-modification*

  where *constraint-modification* is one of:

  ADD CONSTRAINT *constraint-name new-constraint*
  DROP CONSTRAINT *constraint-name*
  RENAME CONSTRAINT *old-name* **TO** *new-name*
  ALTER COLUMN *attribute-name domain-constraint*
  (**Oracle Syntax** for last one: **MODIFY** *attribute-name domain-constraint*)

- Example (PostgreSQL syntax):

```
ALTER TABLE Enrolled ALTER COLUMN grade SET NOT NULL;
```

- What happens if the existing data in a table does not fulfil a newly added constraint?

  Then constraint gets not created!

  e.g. "ORA-02293: cannot validate (DAMAGECHECK) - check constraint violated"

Assertions

- The integrity constraints seen so far are associated with a single table
  - Plus: they are required to hold only if the associated table is nonempty!
- Need for a more general integrity constraints
  - E.g. integrity constraints over several tables
  - Always checked, independent if one table is empty

- **Assertion**: a predicate expressing a condition that we wish the database always to satisfy.
- SQL-92 syntax:
  create assertion <*assertion-name*> **check** (<*condition*>)

- Assertions are schema objects (like tables or views)
- When an assertion is made, the system tests it for validity, and tests it again on every update that may violate it
  - This testing may introduce a significant amount of overhead; hence assertions should be used with great care.

Assertion Example

- The number of boats plus the number of sailors should be less than 100.

```
CREATE TABLE Sailors (
        sid INTEGER
     sname CHAR (10)
    rating INTEGER
   PRIMARY KEY (sid)
    CHECK (rating >=1 AND rating <=10)
    CHECK ((SELECT count(s.sid) FROM Sailors s
           + (SELECT count(b.bid) FROM Boats b) < 100))
```

```
CREATE ASSERTION smallclub CHECK
(   (SELECT COUNT(s.sid) FROM Sailors s)
   + (SELECT COUNT(b.bid) FROM Boats b) < 100) )
```

Assertion Example II

- Asserting for all X : P(X) is achieved in a round-about fashion using "not exists X such that not P(X)"

- Example: For all students, the sum of all marks for a course must be less or equal than 100.

```
CREATE ASSERTION mark-constraint CHECK
(
  not exists ( select sid
                 from Assessment
             group by sid,uos_code
               having sum(mark) > 100 )
)
```

- Note: Although generalizing nicely the semantic constraints, assertions are not supported by most DBMS

Constraints checked externally

- Due to limitations of SQL constraint/assertion implementations, many applications do not declare in DBMS every known limitation on allowed domain state
- Instead, each relevant application code does some checks and takes appropriate response
  - This also allows more flexibility, for example, conditions that indicate a restriction based on the way the state has changed (ie, how the state after and state before, are related)
    - Eg salaries can only increase, but never decrease
  - This allows checks to be skipped, in cases where the condition cannot be violated
    - Eg removing a student, won't violate a restriction on maximum number of students in any class

Transaction

- A sequence of database actions, that collectively accomplish one real-world change
  - May involve modifying several tables (note that a single SQL statement can only UPDATE one table)
- The programmer can indicate that these actions are to be done as a transaction, which means system ensures all-or-nothing impact on database state
  - Details covered in week 11

Deferring Constraint Checking

- Any dbms-know constraint - domain, key, foreign-key, check/assertion - may be declared:
  - **NOT DEFERRABLE**
    The default. It means that every time a database modification occurs, the constraint is checked immediately afterwards.
  - **DEFERRABLE**
    Gives the option to wait until a transaction (with several operations) is complete before checking the constraint.

Example: Deferring Constraints

```
CREATE TABLE UnitOfStudy
(
   uos_code      VARCHAR(8),
   title         VARCHAR(220),
   lecturer      INTEGER,
   credit_points INTEGER,
   CONSTRAINT UnitOfStudy_PK PRIMARY KEY (uos_code),
   CONSTRAINT UnitOfStudy_FK FOREIGN KEY (lecturer)
     REFERENCES Lecturer DEFERABBLE INITIALLY DEFERRED
);
```

- Allows to insert a new course referencing a lecturer which is not present at that time, but who will be added later *in the same transaction*.

Trade-off

- Checking code in application is very flexible, but also carries risks
- The essential properties are not known to the dbms, so enforcement depends on programmers doing the right thing
  - Some-one might write new code (or modify existing code) and forget to do some checks
  - Some-one might produce buggy code
- Once the database state is not obeying the expected properties of the domain, any later activities might produce unexpected results and the data errors can spread

Triggers

- More flexible than constraints, but kept inside the dbms, are triggers
- A **trigger** is a statement that is executed automatically if specified modifications occur to the DBMS.
- A trigger specification consists of three parts:

  ON *event* **IF** *precondition* **THEN** *action*

  - *Event* (what activates the trigger?)
  - *Precondition* (guard / test whether the trigger shall be executed)
  - *Action* (what happens if the trigger is run)
- Triggers introduced to SQL standard in SQL:1999, but supported even earlier using non-standard syntax by most databases.

Why Triggers?

- Constraint maintenance
  - Triggers can be used to maintain foreign-key and semantic constraints; commonly used with ON DELETE and ON UPDATE
- Business rules
  - Some dynamic business rules can be encoded as triggers
  - Assertions can be implemented using <u>two</u> triggers
- Monitoring
  - E.g. to react on the insertion of some kind of sensor reading into db
- Maintenance of auxiliary cached data
  - Careful! Many systems now support *materialized views* which should be preferred against such maintenance triggers
- Simplified application design
  - E.g. exceptions modelled as update operations on a database (if applicable)

Trigger Example (SQL:1999)

```
CREATE TRIGGER gradeUpgrade
  AFTER INSERT OR UPDATE ON Assessment
  BEGIN
    UPDATE Enrolled E
       SET grade='P'
     WHERE grade IS NULL
       AND ( SELECT SUM(mark)
                FROM Assessment A
               WHERE A.sid=E.sid AND
                     A.uos=E.uosCode ) >= 50;
  END;
```

Triggers – PostgreSQL Syntax

```
CREATE TRIGGER trigger-name

 { BEFORE }  { INSERT }
 { AFTER  }  { DELETE }  ON relation-name
             { UPDATE }

  FOR EACH ROW                      -- optional; only for row-triggers
                                    -- optional; otherwise a statement trigger
  WHEN ( condition )                -- optional
  EXECUTE PROCEDURE stored-procedure-name(); --needs to be defined 1st
```

-- PL/pgSQL can be used to define trigger procedures
-- needs to be specified with no arguments
-- When a PL/pgSQL function is called as a trigger, several special variables
-- are created automatically in the top-level block:

NEW
OLD
TG_WHEN ('BEFORE' or 'AFTER')
TG_OP ('INSERT', 'DELETE, 'UPDATE', 'TRUNCATE')
…

[cf. http://www.postgresql.org/docs/8.4/static/plpgsql-trigger.html]

Trigger Events and Granularity

- Triggering event can be **insert**, **delete** or **update**
- Triggers on update can be restricted to specific attributes

```
CREATE TRIGGER overdraft-trigger AFTER UPDATE OF balance
                                                ON account
```

- **Granularity**
  - *Row-level granularity*: change of a single row is an event (a single UPDATE statement might result in multiple events)
  - *Statement-level granularity*: events are statements (a single UPDATE statement that changes multiple rows is a single event).
- Can be more efficient when dealing with SQL statements that update a large number of rows…

Statement vs. Row Level Trigger

- Example: Assume the following schema

```
Employee  ( name, salary )
```

with *1000 tuples* and an *ON UPDATE trigger* on salary…

- Now let's give employees a pay rise:

```
UPDATE Employee SET salary=salary*1.025;
```

- Update Costs:
  - How many rows are updated? **1000**
  - How often is a ***row-level*** trigger executed? **1000**
  - How often is a ***statement-level*** trigger executed? **1**

Trigger Granularity - Syntax

- Instead of executing a separate action for each affected row, a single action can be executed for all rows affected by a transaction
  - Use `FOR EACH STATEMENT` instead of `FOR EACH ROW` (actually the default)
  - Some systems (e.g. Oracle, but NOT PostgreSQL) allow to use
    `REFERENCING OLD TABLE`
    or **`REFERENCING NEW TABLE`**
    to refer to temporary tables (called *transition tables*) containing the affected rows
- Can be more efficient when dealing with SQL statements that update a large number of rows…

Before Trigger Example
(row granularity, PostgreSQL syntax)

```
CREATE FUNCTION AbortEnrolment() RETURNS trigger AS $$
  BEGIN
    RAISE EXCEPTION 'unit is full'; -- aborts
  END
$$ LANGUAGE pgplsql;
```

(1) In PostgreSQL, you first need to define a trigger function…

```
CREATE TRIGGER  Max_EnrollCheck
   BEFORE INSERT ON Transcript
   FOR EACH ROW
   WHEN ((SELECT COUNT (T.studId)
            FROM Transcript T
           WHERE T.uosCode = NEW.uosCode AND
                 T.semester = NEW.semester)
      >= (SELECT U.maxEnroll
            FROM UnitOfStudy U
           WHERE U.uosCode = NEW.uosCode ))
   EXECUTE PROCEDURE AbortEnrolment();
```

(2) … before you can declare the actual trigger, that uses it

Check that enrollment ≤ limit

After Trigger Example

(statement granularity, PostgreSQL syntax)

```
CREATE TABLE Log ( … );
CREATE FUNCTION SalaryLogger() RETURNS trigger AS $$
BEGIN
   INSERT INTO Log
        VALUES (CURRENT_DATE, SELECT AVG(Salary)
                                   FROM Employee );
   RETURN NEW;
END
$$ LANGUAGE plpgsql;

CREATE TRIGGER RecordNewAverage
    AFTER UPDATE OF Salary ON Employee
    FOR EACH STATEMENT
    EXECUTE SalaryLogger();
```

Keep track of salary averages in the log

Some Tips on Triggers

- Use BEFORE triggers
  - For checking integrity constraints
- Use AFTER triggers
  - For integrity maintenance and update propagation
- In Oracle, triggers cannot access "mutating" tables
  - e.g. AFTER trigger on the same table which just updates

- Good overview:
  - Kifer/Bernstein/Lewis: "Database Systems - An Application-oriented Approach", 2nd edition, Chapter 7.

When Not to Use Triggers

- Triggers were used earlier for tasks such as
  - maintaining summary data (e.g. total salary of each department)
  - Replicating databases by recording changes to special relations (called change or delta relations) and having a separate process that applies the changes over to a replica
- There are better ways of doing these now:
  - Databases today provide built-in materialized view facilities to maintain summary data
  - Databases provide built-in support for replication

References

- Kifer/Bernstein/Lewis (2nd edition)
  - Sections 3.2.2-3.3 and Chapter 7
  - *Integrity constraints are covered as part of the relational model, but a good dedicated chapter (Chap 7) on triggers*
- Ramakrishnan/Gehrke (3rd edition - the ‘Cow’ book)
  - Sections 3.2-3.3 and Sections 5.7-5.9
  - *Integrity constraints are covered in different parts of the SQL discussion; only brief on triggers*
- Ullman/Widom (3rd edition)
  - Chapter 7
  - *Has a complete chapter dedicated to both integrity constraints&triggers. Good.*