COMP9120

Week 5: Database Integrity

Semester 2, 2016

(Ramakrishnan/Gehrke – Chapter 5.7-5.9;

Kifer/Bernstein/Lewis – Chapter 3.2-3.3;

Ullman/Widom – Chapter 7)

Based on material by Dr. Bryn Jeffries

THE UNIVERSITY OF SYDNEY

THE UNIVERSITY OF SYDNEY

Copyright warning

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**).

The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

Outline

- **Static Integrity Constraints**
  - Domain Constraints
  - Key / Referential Constraints
  - Semantic Integrity Constraints

- **Dynamic Integrity Constraints**
  - **Triggers**

Integrity Constraints

- Objective:
  - capture semantics of the miniworld 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 ensures semantic integrity constraints:
  - More effective integrity control
  - Stored data is more faithful to real-world meaning
  - 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’.
- Valid lecturer titles are ‘Lecturer’, ‘Senior Lecturer’ or ‘Professor’
- Students can only enrol in the units of study on offer.
- Students must be assessed by the lecturer who actually gave the course and the mark they achieve is between 0 and 100.
- The sum of all marks in a course cannot be higher than 100.

Integrity Constraints (IC) in a Database

- **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
- 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 certain parts of the database are modified
  - Can specify if ICs should be checked:
    - After a SQL statement, or at the end of a ‘transaction’?
      - Transaction: for now: a group of statements to be executed atomically
        (will later look at “ACID” properties)
- Possible ***reactions*** if an IC is violated:
  - Reject database operation
  - Abort of the ‘transaction’ – rollback operations part of current ‘transaction’
  - Execution of “maintenance” operations to make DB legal again

Types of Integrity Constraints

- **Static Integrity Constraints**
  describe conditions that every *legal instance* of a database must satisfy
  - Inserts / deletes / updates that violate ICs are disallowed
  - Three kinds:
    - *Domain Constraints*
    - *Key Constraints & Referential Integrity*
    - *Semantic Integrity Constraints; Assertions*

- **Dynamic Integrity Constraints**
  are predicates on database state changes
  - *Triggers*

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.
- Most simply, each attribute needs to have a *data type*
- SQL DDL allows domains of attributes to be further 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     NOT NULL,
    name        VARCHAR(20) NOT NULL,
    semester    INTEGER     DEFAULT 1,
    birthday    DATE        NULL,
    country     VARCHAR(20)
);
```

Semantic:

sid and **name** must not be NULL
all other attributes can be NULL (**semester, birthday** and **country**)
semester will be 1 if not specified by an insert
birthday and **country** will be NULL if not specified by an insert

Example:

```
INSERT INTO Student(sid,name) VALUES (123,'Pete');
```

Domain Check Constraints

- Limit the allowed values for an attribute by specifying extra conditions with an in-line check constraint

  att-name sql-data-type **CHECK (** *condition* **)**

- Examples:
  - Gender can be male or female

    ```
    gender VARCHAR(6) CHECK( gender IN ('male', 'female'))
    ```

  - Age must be positive

    ```
    age INTEGER CHECK( age >=0 )
    ```

- Check constraints can be used for more general constraints than domain constraints – return to this shortly

User-Defined Domains

- New domains can be created from existing data domains, with their own defaults and restrictions

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

create domain Grade **char default** 'P' **check**(value in ('F','P','C','D','H'))

- Currently only Sybase and PostgreSQL (NOT in Oracle)

- Can compare values of the created domain, to values of base type

- User-defined types with SQL:1999:

```
CREATE TYPE type-name AS sql-base-type
```

create type Dollars **as** numeric(12,2)
create type Pounds **as** numeric(12,2)

- so far, only supported by IBM DB2
- (SQL Server has an *add_type()* procedure; Oracle has a variant)

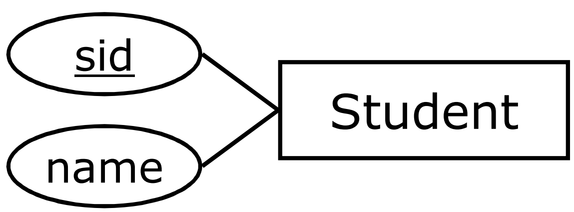
Key Constraints

› Recall definition from week 3:

- 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 key constraints using the **PRIMARY KEY** and **UNIQUE** clauses:




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

› A primary key is automatically unique and NOT NULL

- Can have multiple overlapping keys, but only one primary key

› Complex keys: separate clause at end of **create table**

Foreign Keys and Referential Integrity

See Week 3 slides

- **Foreign key :**
  - 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 α, there must be a tuple in the referred relation whose primary key value is also α
  - 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

Enforcing Referential Integrity in SQL

› SQL-92 and SQL-1999 support all 4 options on deletes and updates.

- Default is **NO ACTION** (delete/update is rejected)
- **CASCADE** (also delete/update all tuples that refer to deleted/updated tuple)
- **SET NULL / SET DEFAULT** (sets foreign key value of referencing tuple)

```
CREATE TABLE Enrolled
(
    -- the sid field default
    -- value is 1234567890
    sid CHAR(10) DEFAULT 1234567890,
    uos CHAR(8),
    grade CHAR(2),

    PRIMARY KEY (sid,uos),

    FOREIGN KEY (sid) REFERENCES Student
    -- the on delete cascade conveys
    -- that an enrolled row should be
    -- deleted when the student with sid
    -- that it refers to is deleted
    ON DELETE CASCADE

    -- the on update set default
    -- will attempt to update the
    -- value of sid to a default value
    -- that is specified as the default
    -- in this Enrolled schema definition
    ON UPDATE SET DEFAULT
);
```

Semantic Integrity Constraints

- Examples:
  - "Scores are from 0 to 100"
  - "Only lecturers of a course can award marks for that course."

- Use SQL CHECK constraints, in-line like before, or as separate named constraints:

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

- One can use subqueries to express constraints (SQL-92 standard)
  - Note: subqueries in CHECKs are NOT SUPPORTED by either PostgreSQL or Oracle (Sybase is one example that does this)

Semantic 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 Oracle 12c or PostgreSQL.

THE UNIVERSITY OF SYDNEY

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)
);
```

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 DEFERRABLE INITIALLY DEFERRED
);
```

- Allows us to insert a new course referencing a lecturer that is not present at the time, but who will be added later *in the same transaction*.
- Behaviour can be dynamically changed within a transaction with the SQL statement
  SET CONSTRAINT *UnitOfStudy_FK* **IMMEDIATE**;

Deferring Constraint Checking

- Any constraint - domain, key, foreign-key, semantic - may be declared:

  - **NOT DEFERRABLE**
    The default. It means that every time a database modification occurs to tuples a DBMS sees as being related, the constraint is checked immediately afterwards.

  - **DEFERRABLE**
    Gives the option to wait until a transaction is complete before checking the constraint.

    - **INITIALLY DEFERRED** wait until transaction end, but allow to dynamically change later
    - **INITIALLY IMMEDIATE** check immediate, but allow to dynamically change later

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 (Oracle syntax):

```
ALTER TABLE Enrolled MODIFY grade CHAR(2) NOT NULL;
```

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

  Then constraint doesn't get created!
  e.g. "SQL Error: ORA-02296: cannot enable (USER.) - null values found"

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, even 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)
- 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 that “some condition should hold”, is achieved in a round-about fashion using “there should be no case where a condition doesn’t hold”

- Example: All student assessments should have a mark greater then or equal to 0.

```
 CREATE ASSERTION mark-constraint CHECK
(
  not exists ( select sid
                 from Assessment
                where mark < 0
             )
);
```

- Note: Although generalizing nicely the semantic constraints, assertions are <u>not supported</u> by any mainstream DBMS at the moment…

THE UNIVERSITY OF SYDNEY

Comparison of Constraints

› Principle differences among integrity constraints types

Type of constraint	When activated	Guaranteed to hold?
DEFAULT NOT NULL/NULL	insert or updates	Yes
CREATE DOMAIN	n.a.	n.a.
Referential integrity	Any table modification	Yes
Attribute-based CHECK	On insertion to relation or attribute update	Not if subquery
Tuple-based CHECK	On insertion to relation or attribute update	Not if subquery
Assertion	On any change to any mentioned relation	Yes

THE UNIVERSITY OF SYDNEY

Today's Agenda

- **Static Integrity Constraints**
  - Domain Constraints
  - Key / Referential Constraints
  - Semantic Integrity Constraints

- **Dynamic Integrity Constraints**

Triggers

- A **trigger** is a section of code that is executed automatically if specified modifications occur to the DBMS, AND a certain condition holds true.

- 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.

THE UNIVERSITY OF SYDNEY

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
- Monitoring
  - E.g. to react on the insertion of some kind of sensor reading into db

Trigger Example (SQL:1999)

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

  END;
```

Triggering Events and Actions

- 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
```

- Values of attributes before and after an update can be referenced
  - **REFERENCING OLD ROW AS** *name* : for deletes and updates
  - **REFERENCING NEW ROW AS** *name* : for inserts and updates
  - In PostgreSQL: separate OLD and NEW variable automatically in trigger block
- E.g. convert blanks to null:

```
CREATE TRIGGER setnull_trigger  BEFORE UPDATE ON s
REFERENCING NEW  AS nrow  FOR EACH ROW
WHEN (nrow.country = ' ')
BEGIN
   :nrow.country := null;
END;
/
```

- Triggers can be activated before an event, which can serve as extra constraints.
  - Raise errors to reject operations: `RAISE_APPLICATION_ERROR(-20000, ’unit is full’);`

Trigger Granularity

- **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 a *BEFORE 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?
- How often is a ***row-level*** trigger executed?
- How often is a ***statement-level*** trigger executed?

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 statement
  - 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…

After Trigger Example (statement granularity)

Keep track of salary averages in the log

```
CREATE TRIGGER RecordNewAverage
   AFTER UPDATE OF Salary ON Employee
   FOR EACH STATEMENT
   BEGIN
      INSERT INTO Log
      VALUES (CURRENT_TIMESTAMP, SELECT AVG(Salary)
                                  FROM Employee );
   END;
```

Triggers

```
CREATE [OR REPLACE] TRIGGER trigger-name

 ⎡ BEFORE     ⎤ ⎛ INSERT            ⎞
 ⎢ AFTER      ⎥ ⎜ DELETE            ⎟  ON relation-name
 ⎣ INSTEAD OF ⎦ ⎝ UPDATE OF attr    ⎠

 REFERENCING    ⎡ OLD [TABLE] ⎤ AS variable-name     -- optional
                ⎣ NEW [TABLE] ⎦

 FOR EACH ROW                 -- optional; otherwise a statement trigger
 WHEN (condition)             -- optional; only for row-triggers
 DECLARE                      -- optional
    <local variable declarations>
 BEGIN
   <PL/SQL block>
 END;
```

Some Tips on Triggers

- Use BEFORE triggers
  - For checking integrity constraints
- Use AFTER triggers
  - For integrity maintenance and update propagation

- Good overviews:
  - Ramakrishnan – Brief overview §§5.8, 5.9
  - Kifer/Bernstein/Lewis: "Database Systems - An Application-oriented Approach", 2nd edition, Chapter 7.
  - Michael v.Mannino: "Database - Design, Application Development and Administration"
  - Oracle Application Developer's Guide, Chapter 15

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

You Should be Able To

- **Capture Integrity Constraints in an SQL Schema**
  - Including key constraints, referential integrity, domain constraints and semantic constraints
  - And simple triggers for dynamic constraints
- Formulate complex semantic constraints using Assertions
- Know when to use Assertions, triggers, and CHECK constraints
- Know the semantic of deferring integrity constraints
- **Be able to formulate simple triggers**
  - Know the difference between row-level & statement-level triggers

References

- 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*
- 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*
- Ullman/Widom (3rd edition)
  - Chapter 7
  - *Has a complete chapter dedicated to both integrity constraints&triggers. Good.*
- Michael v.Mannino: “Database - Design, Application Development and Administration”
  - *Include a good introduction to triggers.*
- Oracle Application Developer’s Guide, Chapter 15
  - *The technical details on the specific Oracle syntax and capabilities.*

Next Topic

- Advanced SQL
  - Complex Joins
  - Sub-queries
  - Grouping
- Relational Division
  - For-All queries in SQL

- Readings:
  - Ramakrishnan/Gehrke (Cow book), Chapter 5
  - Kifer/Bernstein/Lewis book, Chapter 5 & 3.2-3.3
  - Ullman/Widom, Chapter 6