INFO2120 – INFO2820 – COMP5138 Database Systems

Week 7: Database Security and Integrity

(Kifer/Bernstein/Lewis – Chapter 3.2-3.3; Ramakrishnan/Gehrke – Chapter 5.7-5.9; Ullman/Widom – Chapter 7)

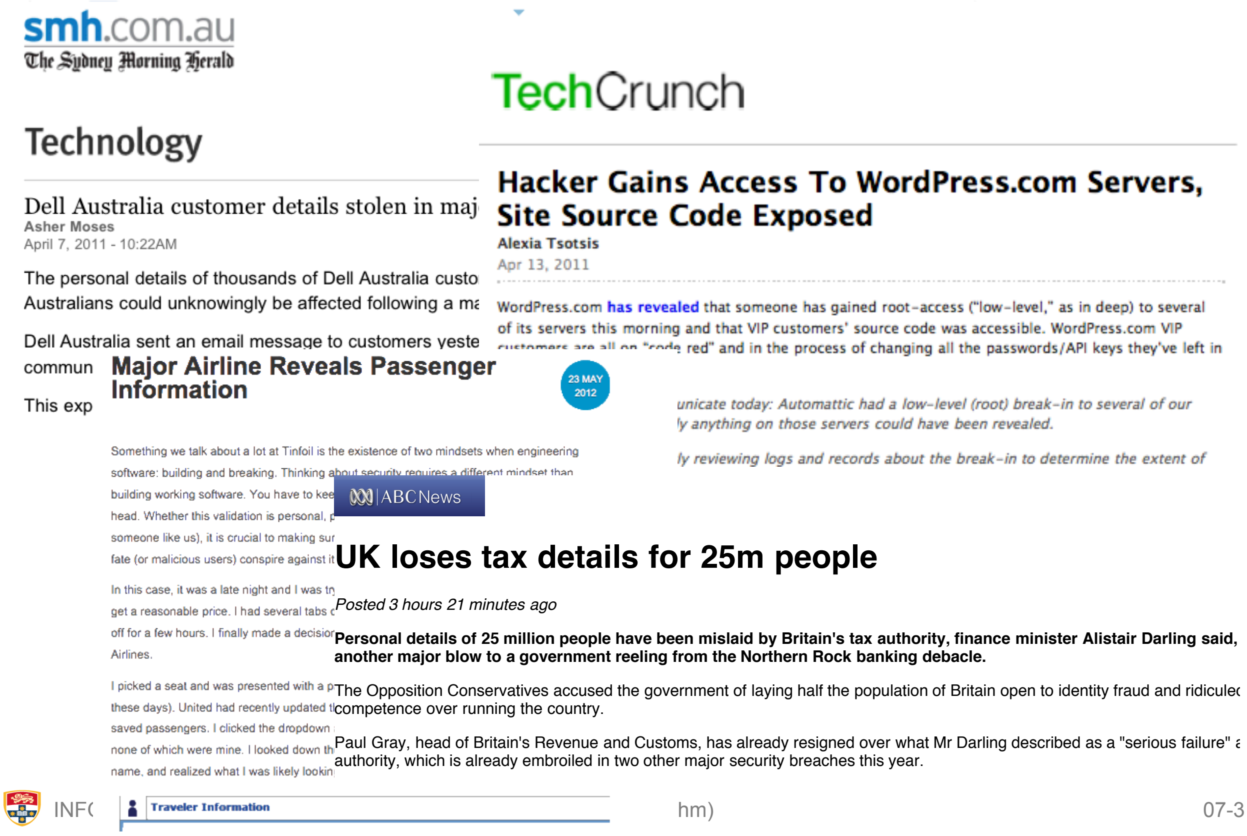
Dr. Uwe Röhm
School of Information Technologies

Outline

- **Database Security**
- **Static Integrity Constraints**
  - Domain Constraints
  - Key / Referential Constraints
  - Semantic Integrity Constraints
- **Dynamic Integrity Constraints**
  - **Triggers**

Based on slides from Kifer/Bernstein/Lewis (2006) “Database Systems” and from Ramakrishnan/Gehrke (2003) “Database Management Systems”, and also including material from Fekete and Röhm.

Small Snapshot of Recent IT News



"Houston – we have a problem…"

- **CWE's Top 25 Most Dangerous Software Errors**

This is a brief listing of the Top 25 items, using the general ranking.

NOTE: 16 other weaknesses were considered for inclusion in the Top 25, but their general scores were not high enough. They are listed in a separate "On the Cusp" page.

Rank	Score	ID	Name
[1]	93.8	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
[2]	83.3	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
[3]	79.0	CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
[4]	77.7	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
[5]	76.9	CWE-306	Missing Authentication for Critical Function
[6]	76.8	CWE-862	Missing Authorization
[7]	75.0	CWE-798	Use of Hard-coded Credentials
[8]	75.0	CWE-311	Missing Encryption of Sensitive Data
[9]	74.0	CWE-434	Unrestricted Upload of File with Dangerous Type
[10]	73.8	CWE-807	Reliance on Untrusted Inputs in a Security Decision
[11]	73.1	CWE-250	Execution with Unnecessary Privileges
[12]	70.1	CWE-352	Cross-Site Request Forgery (CSRF)
[13]	69.3	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
[14]	68.5	CWE-494	Download of Code Without Integrity Check
[15]	67.8	CWE-863	Incorrect Authorization
[16]	66.0	CWE-829	Inclusion of Functionality from Untrusted Control Sphere
[17]	65.5	CWE-732	Incorrect Permission Assignment for Critical Resource
[18]	64.6	CWE-676	Use of Potentially Dangerous Function
[19]	64.1	CWE-327	Use of a Broken or Risky Cryptographic Algorithm
[20]	62.4	CWE-131	Incorrect Calculation of Buffer Size
[21]	61.5	CWE-307	Improper Restriction of Excessive Authentication Attempts
[22]	61.1	CWE-601	URL Redirection to Untrusted Site ('Open Redirect')
[23]	61.0	CWE-134	Uncontrolled Format String
[24]	60.3	CWE-190	Integer Overflow or Wraparound
[25]	59.9	CWE-759	Use of a One-Way Hash without a Salt

[39] CWE-209: Information Exposure Through an Error Message

[Source: http://cwe.mitre.org/top25/]

Database Security

- Databases might contain sensitive information
- Need mechanisms to guarantee:
  - **Secrecy**: Users should not be able to see things they are not supposed to.
    - E.g., A student can't see other students' grades.
  - **Integrity**: Users should not be able to modify things they are not supposed to.
    - E.g., Only instructors can assign grades.
  - **Availability**: Users should be able to see and modify things they are allowed to.

- SQL:92 provides tools for specifying an authorization policy but does not support authentication (vendor specific)

Database Access Control

- Two main security mechanisms at the DBMS level:
- **Mandatory access control (*Authentification*)**
  - Every connection must login with login and password
  - **CREATE USER** or **CREATE LOGIN** commands etc.
- **Discretionary access control (*Authorization*)**
  - Based on the concept of access rights or **privileges** for objects (tables and views), and mechanisms for giving users privileges (and revoking privileges).
  - Creator of a table or a view automatically gets all privileges on it.
    - DMBS keeps track of who subsequently gains and loses privileges, and ensures that only requests from users who have the necessary privileges (at the time the request is issued) are allowed.

Access Control in SQL

```
GRANT privilege_list
   ON table             (any schema object)
   TO user_list
[WITH GRANT OPTION]
```

- privileges: **SELECT, INSERT, DELETE, UPDATE, REFERENCES**

- Examples:

  GRANT UPDATE(grade) ON Enrolled TO uroehm
  - Only the `grade` column can be updated by user 'uroehm'

  GRANT SELECT ON Enrolled TO jpoon
  - Individual columns cannot be specified for SELECT access (SQL standard) – all columns of **Enrolled** can be read (including any added later via ALTER TABLE command).
  - SELECT access control to individual columns can be simulated through views

Grant and Revoke Privileges

- If a user has a privilege with the **GRANT OPTION**, can pass privilege on to other users (with or without passing on the GRANT OPTION).
- Only owner can execute CREATE, ALTER, and DROP.
- Examples:

  GRANT *INSERT,SELECT* ON *Student* TO *Uwe*
  - Uwe can query students or insert tuples into it.

  GRANT *DELETE* ON *Students* TO *Jon* WITH GRANT OPTION
  - Jon can delete tuples, and also authorize others to do so.

  GRANT *UPDATE(title)* ON *UnitofStudy* TO *Dustin*
  - Dustin can update (only) the title field of *Courses* tuples.

  GRANT *SELECT* ON *FemaleStudents* TO *Guppy, Yuppy*

  This is a view on Students - what can the 'uppy's now see?

- **REVOKE:** When a privilege is revoked from X, it is also revoked from all users who got it solely from X.

Views and Security

- Views can be used to present necessary information (or a summary), while hiding details in underlying relation(s).
  - Given view
    `CREATE VIEW EnrolledStuds AS SELECT sid,uos_code FROM Enrolled`
    we can find students who have enrolled in courses, but not the *grades* they have achieved.
- Creator of view has a privilege on the view if (s)he has the privilege on all underlying tables.
  - Granting a privilege on a view does not imply granting any privileges on the underlying relations.
  - If creator of base tables revokes SELECT right, view is automatically dropped.
- Together with GRANT/REVOKE commands, views are a very powerful access control tool.

Example: GRANTs and VIEWs

- User A: CREATE TABLE Student (sid INT, …);
  GRANT SELECT ON Student TO B WITH GRANT OPTION;
  / note: without GRANT OPTION, B cannot pass SELECT privilege on its view on to C */*
- User B: CREATE VIEW MyStud AS
  SELECT sid FROM A.Student;
  GRANT SELECT ON MyStud TO C;
- User C: SELECT * FROM B.MyStud; -- works
  SELECT * FROM A.Student; -- does not work
- User A: REVOKE SELECT ON Student FROM B;
  -- what happens now?

Authorization Mode REFERENCES

- Foreign key constraint enforces relationships between tables; those could be exploited to
  - *control access*: can prevent deletion of rows

```
GREATE TABLE DontDismissMe (
        id INTEGER,
        FOREIGN KEY (id) REFERENCES Student
                         ON DELETE NO ACTION )
```

  - *reveal information*: successful insertion into **DontDissmissMe** means a row with a foreign key value exists in **Student**
  - Example:

```
INSERT INTO DontDismissMe VALUES (11111111);
```

- REFERENCES access mode allows to prevent this by only allowing authorized users to use foreign keys to a table

```
GRANT REFERENCES ON Student TO flexsis
```


Role-based Authorization

- In SQL-92, privileges are actually assigned to *authorisation ids*, which can denote a single user or a group of users.
- In SQL:1999 (and in many current systems), privileges are assigned to **roles**.
  - Roles can then be granted to users and to other roles.
  - Reflects how real organisations work.
  - Much more flexible and less error-prone, especially on large schemas

=> use role-based authorization whenever possible

- Example:

```
CREATE ROLE manager
GRANT select,insert ON students TO manager
GRANT manager TO shari
```

Limitations of SQL Authorization

- SQL does not support authorization at a tuple level
  - eg. we cannot restrict students to see only (the tuples storing) their own grades
  - can be simulated to a certain degree using Views, but VERY cumbersome
- With the growth in Web access to databases, database accesses come primarily from application servers.
  - End users don't have database user ids, they are all mapped to the same database user id
- All end-users of an application (such as a web application) may be mapped to a single database user
- The task of authorisation in above cases falls on the application program, with no support from SQL
  - Benefit: fine grained authorisations, such as to individual tuples, can be implemented by the application.
  - Drawback: Authorisation must be done in application code, and may be dispersed all over an application
  - Checking for absence of authorisation loopholes becomes very difficult since it requires reading large amounts of application code

Data Minimalism

- The best protection against unauthorized access to data in your database is to consider very carefully what you store in the first place!
- A database should only store information that is absolutely necessary for the operation of your application.
- Some data is even not allowed to be stored
  - For example: Sensitive authentication data such as the security code of a credit card
    - Cf. https://www.pcisecuritystandards.org/documents/pa-dss_v2.pdf
  - In Australia, the Tax File Number or the Medicare numbers is specifically protected from being used outside government
  - Any personal health information

Data Privacy

- Some information is specifically protected and requires specific standards and auditing procedures
  - especially for governmental organisations or large businesses
- In Australia, the Privacy Act 1988 (Cth) (the Privacy Act) governs the protection rules regarding personal information
  - Personal information: information where an individual is reasonably identifiable, i.e. information that identifies/could identify an individual
  - regulates e.g. what and how to collect, disclosure rules, requirement to ensure information quality, when to delete
- cf. National Privacy Principles (NPP)
  - http://www.privacy.gov.au/law/act/npp

Outline

- Database Security
- **Static Integrity Constraints**
  - Domain Constraints
  - Key / Referential Constraints
  - Semantic Integrity Constraints
- Dynamic Integrity Constraints
  - Triggers

Based on slides from Kifer/Bernstein/Lewis (2006) “Database Systems” and from Ramakrishnan/Gehrke (2003) “Database Management Systems”, and also including material from Fekete and Röhm.

Semantic 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
- Note: DBMS allow to capture more ICs than, e.g., ERM

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 enroll in actually offered unit of studies.
- Students must be assest 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.
- A lecturer can only be promoted, but never degraded.

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
  - With one degree of freedom:
    - After a SQL statement, or at the end of a transaction?
- Possible *reactions* if an IC is violated:
  - Undoing of a database operation
  - Abort of the 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.
- 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,
    semester    INTEGER     DEFAULT 1,
    birthday    DATE        NULL,
    country     VARCHAR(20)
);
```

Semantic:

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

Example:

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

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'))

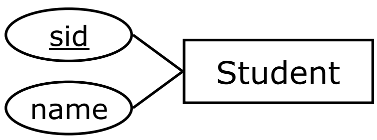
- User-defined types with SQL:1999:

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

- Will most probably replace the create domain mechanism
  - CREATE DOMAIN: Currently only Sybase and PostgreSQL
  - CREATE DISTINCT TYPE: so far, only supported by IBM DB2 (SQL Server has an *add_type()* procedure)

Primary Key Constraints

- Recall definition from week 2:
  - 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 keys: separate clause at end of **create table**

Foreign Keys & Referential Integrity

- **Foreign key :** Cf. Week 3, slides 03-26 & 03-27
  - 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 all tuples that refer to deleted tuple)
  - **SET NULL / SET DEFAULT** (sets foreign key value of referencing tuple)
  - *Cf. Example in Tutorial*

```
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 SET DEFAULT )
```


Semantic 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* **)**
- One can use subqueries to express constraint (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 our dbms.

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 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*.
- 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, 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 (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 $\forall X : P(X)$ 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 any DBMS at the moment…

Comparison of Constraints

- Principle differences among integrity constraints types

Type of constraint	Where declared	When activated	Guaranteed to hold?	Supported by DBMS
DEFAULT NOT NULL/NULL	CREATE TABLE on attribute	insert or updates	Yes	All
CREATE DOMAIN	Own schema object	n.a.	n.a.	Sybase; Postgres
Referential integrity	CREATE TABLE	Any table modification	Yes	All* (MySql since v4.x with InnoDB)
Attribute-based CHECK	CREATE TABLE on attribute	On insertion to relation or attribute update	Not if subquery	All except MySQL
Tuple-based CHECK	At end of CREATE TABLE	On insertion to relation or attribute update	Not if subquery	All except MySQL but subqueries only with Sybase
Assertion	Own schema object	On any change to any mentioned relation	Yes	none

CREATE DOMAIN PostgreSQL Syntax

Quote from the PostgreSQL 8.3 manual

```
Name
    CREATE DOMAIN -- define a new domain

Synopsis
    CREATE DOMAIN name [AS] data_type
              [ DEFAULT expression ]
              [ constraint [ ... ] ]

    where constraint is:
          [ CONSTRAINT constraint_name ]
          { NOT NULL | NULL | CHECK (expression) }

Description
    CREATE DOMAIN creates a new data domain. A domain is essentially a data type
    with optional constraints (restrictions on the allowed set of values). The
    user who defines a domain becomes its owner.
    If a schema name is given (for example, CREATE DOMAIN myschema.mydomain ...)
    then the domain is created in the specified schema. Otherwise it is created in
    the current schema. The domain name must be unique among the types and domains
    existing in its schema.
    Domains are useful for abstracting common fields between tables into a single
    location for maintenance. For example, an email address column may be used in
    several tables, all with the same properties. Define a domain and use that
    rather than setting up each table's constraints individually.
```

*At the moment, only PostgreSQL (and also Interbase/Firebird) support the CREATE DOMAIN statement. DB2 includes something similar - CREATE DISTINCTIVE TYPE - but doesn't allow a constraint to be included Sybase/SQLServer use a stored procedure - **sp_addtype**, which is similar to DB2's CREATE DISTINCTIVE TYPE. Oracle uses a variation on the CREATE TYPE syntax from SQL:1999 which is actual adding an object type. But just like Sybase, MS SQL Server and DB2 it does not accept a named constraint or CHECK clause*

Integrity Constraints in MySQL

Quotes from the MySQL 5.1 manual (Section 1.9):

Foreign Key / Referential Integrity Constraints
"In MySQL Server 3.23.44 and up, the `InnoDB` storage engine supports checking of foreign key constraints, including `CASCADE`, `ON DELETE`, and `ON UPDATE`. See Section 13.5.6.4, '`FOREIGN KEY` Constraints'. For storage engines other than `InnoDB`, MySQL Server parses the `FOREIGN KEY` syntax in `CREATE TABLE` statements, but does not use or store it."

Domain Constraints and NOT NULL
"If you are not using strict mode, then whenever you insert an 'incorrect' value into a column, such as a NULL into a NOT NULL column or a too-large numeric value into a numeric column, MySQL sets the column to the 'best possible value' instead of produ-cing an error: […] For strings, MySQL stores either the empty string or as much of the string as can be stored in the column. …" [and so on…]

Semantic Integrity Constraints (CHECK): Are parsed, but not supported as of MySQL 5.1
"The CONSTRAINT_TYPE column can contain one of these values: UNIQUE, PRIMARY KEY, FOREIGN KEY, CHECK. […] The CHECK value is not available until we support CHECK. "

Dynamic Integrity Constraints (Triggers)
"Stored procedures and functions are implemented beginning with MySQL 5.0."

Transactions:
"MySQL Server ([…] all versions 4.0 and above) supports transactions with the InnoDB transactional storage engine. InnoDB provides full ACID compliance. […]
The other non-transactional storage engines in MySQL Server (such as MyISAM) follow a different paradigm for data integrity called "atomic operations." In transactional terms, MyISAM tables effectively always operate in AUTOCOMMIT=1 mode.
Atomic operations often offer comparable integrity with higher performance. " [often!?!]

Views and Subqueries
Views are implemented since MySQL 5.0.1; Subqueries are available since MySQL 4.1

Today's Agenda

- Database Security
- Static Integrity Constraints
  - Domain Constraints
  - Key / Referential Constraints
  - Semantic Integrity Constraints
- **Dynamic Integrity Constraints**

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
- 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;
```


Design Space of Triggers

- **Activation** - Occurrence of the *event*
- **Consideration** - The point, after activation, when *condition* is evaluated
  - Immediate or deferred (when the transaction requests to commit)
  - *Condition* might refer to both the state before and the state after *event* occurs
- **Execution** – point at which *action* occurs
  - With deferred consideration, execution is also deferred
  - With immediate consideration, execution can occur immediately after consideration or it can be deferred
    - If execution is immediate, execution can occur before, after, or instead of triggering event.
    - Before triggers adapt naturally to maintaining integrity constraints: violation results in rejection of event.

Triggering Events and Actions in SQL

- 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
- Triggers can be activated before an event, which can serve as extra constraints.
  - E.g. convert blanks to null:

```
CREATE TRIGGER setnull-trigger BEFORE UPDATE ON s
 REFERENCING NEW ROW AS nrow
 FOR EACH ROW
     WHEN nrow.country = ' '
      SET nrow.country = null
```


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 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…

Triggers in SQL:1999

- **Events**: INSERT, DELETE, or UPDATE statements or changes to individual rows caused by these statements
  - Since SQL:2008: also INSTEAD OF triggers
- **Condition**: Anything that is allowed in a WHERE clause
- **Action**: An individual SQL statement or a program written in the language of Procedural Stored Modules (PSM) (which can contain embedded SQL statements)
- **Consideration**: *Immediate*
  - Condition can refer to both the state of the affected row or table before *and* after the event occurs
- **Execution**: *Immediate* – can be before or after the execution of the triggering event
  - Action of before trigger cannot modify the database
- **Granularity**: Both *row-level* and *statement-level*

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]
```

Before Trigger Example

(row granularity, PostgreSQL syntax)

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

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();
```

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

(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

Triggers - Oracle Syntax

```
CREATE OR REPLACE TRIGGER trigger-name

  BEFORE          INSERT
  AFTER           DELETE                 ON relation-name
  INSTEAD OF      UPDATE OF attr
-- some dbms do not support INSTEAD OF triggers (e.g. Oracle 9)

  REFERENCING     OLD    AS variable-name      -- optional
                  NEW

  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;
```


Before Trigger Example

(row granularity, Oracle syntax)

Check that enrollment ≤ limit

```
CREATE TRIGGER  Max_EnrollCheck
  BEFORE INSERT ON Transcript
  REFERENCING NEW AS N       --row to be added
  FOR EACH ROW
  WHEN ((SELECT COUNT (T.studId)
           FROM Transcript T
          WHERE T.uosCode = N.uosCode AND
                T.semester = N.semester)
          >=
        (SELECT U.maxEnroll
           FROM UnitOfStudy U
          WHERE U.uosCode = N.uosCode ))
  BEGIN
    RAISE_APPLICATION_ERROR(-20000, 'unit is full');
  END;
```

Oracle way to abort current tx with error msg

After Trigger Example
(statement granularity, Oracle syntax)

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_DATE, SELECT AVG(Salary)
                                FROM Employee );
    END;
```


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 overviews:
  - 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 now 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 consttraints
- Formulate complex semantic constraints using Assertions
- Know when to use Assertions, when triggers, and when 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

- 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.*
- 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

- Database Application Development
  - Embedded SQL in Client Code
  - Call-level Database APIs
  - Server-Side Application Development with Stored Procedures

- Readings:
  - Kifer/Bernstein/Lewis book, Chapter 8
  - <u>or</u> alternatively (if you prefer those books):
    - Ramakrishnan/Gehrke (Cow book), Chapter 6
    - Ullman/Widom, Chapter 9