Database security, integrity etc

ISYS2120 Data and Information Management

Prof Alan Fekete

University of Sydney

Acknowledge: slides from Uwe Roehm and Alan Fekete, and from the materials associated with reference books (c) McGraw-Hill, Pearson

Agenda

- *Overview*
- DBMS Access Control
- User identity and authentication
- Views
- Integrity and constraints
- Triggers, Stored Procedures

Introduction

- Many examples of data breach or other security failure
  - Eg ANU data breach (staff, student records)
  - Eg Medibank data breach (name, address, medicare number, medical insurance claims, etc)
  - See https://en.wikipedia.org/wiki/List_of_data_breaches
- Lots of damage to data holder, as well as to data subject
  - Financial loss
  - Reputation loss
  - Legal penalties

Database Security

This lecture, we focus on security in the dbms itself
In week 8, we will discuss security of the whole ecosystem

- Databases usually contain commercially valuable information
  - Possibly also personal information
- There are many adversaries who could benefit from accessing information, or altering it
- The organisation that owns the data should be willing to spend money to protect the data against inappropriate uses

Data Security Goals

- System and Organisation need to set things up so as to guarantee:
  - **Confidentiality Properties**: Users should not be able to see things which they are not supposed to see.
    - E.g., A student can’t see other students’ grades.
  - **Integrity Properties**: Users should not be able to modify things which they are not supposed to modify.
    - E.g., Only instructors can assign grades.
  - **Availability Properties**: Users should be able to see and modify things which they are allowed to see and modify.

Confusing terms: Integrity

- An integrity security property prevents the wrong people modifying the contents of the database
- Recall: integrity constraints which restrict what the contents might be (that is, the values in the database)
  - These can also indirectly assist to achieve integrity security property, by preventing some modifications

Policy and Mechanism

- A security policy indicates who *should be allowed* to do which actions, and who should not be allowed
  - This is set by senior management, based on legal obligations, business value, risk, etc
- A security mechanism is how the system controls who *is allowed* to do which actions
  - This is controlled by DBA, possibly with detailed adjustment by data owner
- We expect that the security mechanism will enforce the security policy in place
  - This correspondence needs to be checked!

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 except for specific purposes, or by specific organisations
  - 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 number 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 (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
- Australian Privacy Principles (APP)
  - https://www.oaic.gov.au/agencies-and-organisations/app-guidelines
  - http://www.privacy.gov.au/
- Other jurisdictions can also be relevant when storing personal information about their residents

Useful DBMS features

- In this lecture, we cover some capabilities of a DBMS, which can be useful in security mechanisms, but they can also be used for other purposes
  - Privileges
  - Views
  - Constraints
  - Triggers
  - Stored Procedures

You are expected to know whatever we teach about these aspects, not only the security uses

Agenda

- Overview
- *DBMS Access Control*
- User identity and authentication
- Views
- Integrity and constraints
- Triggers, Stored Procedures

Database Access Control

- **Access control (for authorization)**
  - A mechanism provided by dbms for controlling which users perform various operations on various parts of the database
  - It is called “discretionary” access control, because the decision about what is permitted or not, is set by appropriate people as the system operates, and can be adjusted as the situation changes
    - Contrast to mandatory access control, where the decisions are built in to the system, and unchangeable
- In SQL, based on the concept of access rights or **privileges** for tables, and commands for giving users privileges (and revoking privileges).
  - Creator of a table 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.

GRANT command

GRANT privilegelist ON *tablelist* TO *userlist* [WITH GRANT OPTION]

- The following are among the privileges **which** can be mentioned in privilegelist:
  - **SELECT**: Allows to read all columns of any of the tables in *tablelist* (including columns added later via ALTER TABLE command)
    - This can be to return the values because of a SELECT clause, or to use the values in processing a command (eg in determining whether the WHERE clause holds)
  - **INSERT:** Allows to insert extra tuples in any of the listed tables
  - **DELETE**: Allows to delete tuples from any of the listed tables.
  - **UPDATE**: allows to modify the values in any tuples of the tables
  - **REFERENCES**: Can define foreign keys (in other tables) that refer to this tables

GRANT command

GRANT privilegelist ON *tablelist* TO *userlist* [WITH GRANT OPTION]

- ❖ Provided this GRANT command is executed by someone who has the privilege to issue that command (see below), then the privileges mentioned will be granted to access any among a list of tables
  - ❖ Privileges can also be granted to access a view (discussed later)
  - ❖ Privileges can also be granted to roles (discussed later)
  - ❖ Privileges may be given “WITH GRANT OPTION” or just the privilege itself can be given
- If a user has a privilege with the GRANT OPTION, they can themselves pass privilege on to other users by executing a GRANT command
  - The owner has GRANT OPTION privileges
  - Anyone who gets privileges from a command “WITH GRANT OPTION” has the capacity to pass those privileges on to others (including right to give them WITH GRANT OPTION”)
  - GRANT OPTION is a very powerful right; it should be given rarely, and only to those who are highly trusted
- Only owner of a table (or superuser) can execute ALTER and DROP
  - The owner starts as the one who executed CREATE; but owner can pass ownership to another by executing ALTER TABLE name OWNER TO new_owner;

Access Control in SQL - Examples

- Examples:
- `GRANT SELECT ON Enrolled TO jason`
  - Jason is allowed to select (read) any tuples in the **`Enrolled`** table
- `GRANT UPDATE ON Enrolled TO roehm`
  - user 'roehm' can modify any tuples in the **`Enrolled`** table
  - In order for this to be useful, roehm should also have SELECT privilege on Enrolled (otherwise, they could not indicate which tuples to modify, or use existing values to calculate the new ones)
- `GRANT DELETE ON Student TO jon WITH GRANT OPTION`
  - Jon is allowed to delete tuples that are in Student table, and also Jon can authorize others to do so by executing the appropriate GRANT command himself.

Column-specific access control

- It is common for security policy to want different access rights on different columns in a single table
  - Eg User Fred can read Student sid, name, phone, but not see disability-status, nationality, or other columns
  - Eg User Jane can modify Student phone, but not sid or other columns
- Most dbms support variation of command: GRANT privilege(column) ON table
  - Eg GRANT SELECT(sid,name,phone) ON Student TO Fred
  - Eg GRANT UPDATE(phone) ON Student TO Jane
    - GRANT INSERT on specific columns, means that in the newly added row, any other columns will get the default value for that column

Revoke of Privileges

```
REVOKE privilege_list
    ON table
  FROM user_list
```

- When a privilege is revoked from X, it is also revoked from all users who got it *solely* from X.
- But if a user has this privilege via several routes, it is still there until all granters have revoked it

Authorization Mode REFERENCES

- Foreign key constraint could be exploited to
  - *prevent some kinds of modification*: eg can prevent deletion of rows in some other table

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

  - *reveal information*: successful insertion into **DontDismissMe** reveals that a row with particular value exists in **Student**
  - Example:

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

- Existence of REFERENCES privilege allows to limit these powers to appropriate users
  - Most users (other than the table owner) will not have REFERENCES privilege and so cannot use these tricks

```
GRANT REFERENCES ON Student TO StuServices
```

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 most current platforms), 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 student TO manager
GRANT manager TO shari
-- now shari can select and insert on student
GRANT manager TO keiko
-- now keiko can select and insert on student
REVOKE manager FROM shari
-- now shari does not have manager privileges
```

Working in PostgreSQL

- As well as granting access on a table etc, you need to also grant USAGE access to the schema that contains the table eg GRANT USAGE ON unidb to Joe;
- When opening the connection, the user needs to be sure to connect to the database in which the table exists

Agenda

- Overview
- DBMS Access Control
- *User identity and authentication*
- Views
- Integrity and constraints
- Triggers, Stored Procedures

User identity (authentication)

- How does the system know which user’s privilege to check, when a command is issued that requests access to data?
- Every request to dbms comes from some identified user
  - They connected to the dbms
  - Identity was determined then (eg they had to login with a password)
  - Every command they issue comes on a connection and carries the connection id

DBMS accounts

- In many systems, the dbms has its own set of user identities/passwords
  - These are different from those of the operating system etc
  - DBMS accounts are created, and initial passwords are set, by the DBA
  - DBMS passwords need to be carefully protected; as discovery of a password is a serious security threat
    - Consider where passwords are stored, how they are handed out to users, etc
- Some dbms may use Single Sign On, to avoid separate identity management

Powerful users

- A DBA has a lot of privileges, over objects of all the ordinary users
  - This is necessary for managing things, fixing errors etc
- So DBA identity must be carefully protected
  - Warning: never leave default password (as provided when system initialised) for dba account etc
    - see https://www.zdnet.com/article/pgminer-botnet-attacks-weakly-secured-postgresql-databases/

Agenda

- Overview
- DBMS Access Control
- User identity and authentication
- *Views*
- Integrity and constraints
- Triggers, Stored Procedures

Content-based access control

- The Grant statement works uniformly on a whole table or column
- Policy often calls for access to be limited to certain rows only, in a table
- The decision about whether a row should be accessed by someone, may depend on the contents of the some fields in the row
  - Eg John should be able to modify salaries, but only for tuples with Dept = 'Accounts'
  - Eg Each student should be able to see their own grades, but not grades of other students
- SQL DBMS allows one to achieve this by defining a VIEW with only the relevant data, and then GRANT access to that VIEW

Summary access

- Similarly, policy may call for certain users to be able to see aggregates and summaries, without seeing all the separate data items that combine for that summary
  - Eg Jane is allowed to know the level of average salary in each faculty, without knowing salaries of individual staff
- SQL DBMS allows one to achieve this by defining a VIEW with only the summaries, and then GRANT access to that VIEW

Relational Views

- A **view** is a *virtual* relation, but we store a *definition*, rather than a set of tuples.
  - When the view is used within an SQL statement, the result is as if the contents of the view were computed at that time, from the current values in each of the tables mentioned in the view definition
- This provides a mechanism to release access to certain data, without allowing access to all of a genuine table
  - Views can be used to present necessary information (or a summary), while hiding details in underlying relation(s).
  - Users can operate on the view as if it were a stored table
    - DBMS calculates the value whenever it is needed
- Syntax:

```
CREATE VIEW name AS <query expression>
```

  - where
    - <*query expression*> is any legal query expression (can even combine multiple relations)

View Examples

- A view on the students showing their age.

```
 CREATE VIEW ageStudents AS
        SELECT sid, name,
 (extract(year from current_date) -
        extract(year from birthdate)) AS age
          FROM Student
```

- A view on the female students enrolled in 2024sem2

```
CREATE  VIEW FemaleStudents (name, grade)
    AS SELECT S.name, E.grade
         FROM Student S, Enrolled E
        WHERE S.sid = E.sid  AND
           S.gender = ‘F’ AND
           E.semester = ‘2024sem2’
```

View Benefits: Security

- *Need-to-know*: Users not granted access to base tables. Instead they are granted access to the view of the database appropriate to their needs.
  - *External schema* is composed of views.
  - View allows owner to provide others with SELECT access to a subset of columns and a subset of rows, or to provide access to summaries without the full data

Views and Security

- Views can be used to present necessary information (or a summary), while hiding details in underlying relation(s).
- 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 changing any privileges on the underlying relations.
  - If creator of base tables revokes SELECT right from view creator, view is no longer useful
- Together with GRANT/REVOKE commands, views are a very powerful access control tool.

Example 1: 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 LimitedInfoStud AS
  SELECT sid, name FROM A.Student;
  GRANT SELECT ON MyStud TO C;
- C is allowed to observe sid and name, but not other information such as address of students
- User C: SELECT * FROM B.LimitedInfoStud; -- works
  SELECT * FROM A.Student; -- does not work

- Question: If User A does REVOKE SELECT ON Student FROM B;
  -- what happens now?

Example 2: 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 CSStud AS
      SELECT * FROM A.Student
      WHERE Degree = 'Bachelor of CS';
  GRANT SELECT ON CSStud TO C;
  ```
- C is allowed to observe information about students in one degree, but not about other students
- User C:
  ```
  SELECT * FROM B.CSStud; -- works
  SELECT * FROM A.Student; -- does not work
  ```

Updating Views

- Question: Since views look like tables to users, can they be updated?
- Answer: Sometimes, yes
  - under certain limitations, a modification command (UPDATE, INSERT, DELETE) on the view can be performed – the system will change the underlying base table(s) to produce the requested change to the view
- **Updatable Views:** SQL-92 allows updates only on simple views (defined by SELECT-FROM-WHERE on a single relation, without aggregates or distinct
    - SQL:1999 allows more, but many implementations don't support the extra cases

Subsequent slides consider this example view definition

```
CREATE VIEW   CsReg (StudId, CrsCode, Semester) AS
SELECT          T.StudId, T. CrsCode, T.Semester
FROM            Transcript T
WHERE    T.CrsCode LIKE 'CS%'  AND  T.Semester='S2024'
```

Updating Views - Issue

INSERT INTO CsReg (*StudId, CrsCode, Semester*)
VALUES (1111, ‘CSE305’, ‘S2024’)

- **Question**: What value should be placed in attributes of underlying table that have been projected out (e.g., *Grade*)?
- **Answer**: NULL (assuming null allowed in the missing attribute) or DEFAULT

Updating Views - Issue 2

INSERT INTO CsReg (*StudId, CrsCode, Semester*)
VALUES (1111, ‘ECO105’, ‘S2020’)

- **Problem**: New tuple will not be visible in view
- **Solution**: Create View WITH CHECK OPTION ensures the that new rows satisfy the view-defining condition; other changes are rejected

Updating Views - Issue 3

- Maybe there is not a unique way to change the base table(s) that results in the desired modification of the view
- Example (note: this is not within the SQL-92 category of updateable views)

```
CREATE VIEW   ProfDept (PrName, DeName)  AS
SELECT    P.Name, D.Name
FROM      Professor P, Department D
WHERE     P.DeptId = D.DeptId
```

Updating Views - Issue 3 (discussion)

- Eg If we try to perform

  DELETE FROM ProfDept WHERE PrName='Smith' AND DeName='CS'

  - An example showing how updates can be difficult or impossible to translate to actions on the base tables, and hence are disallowed for views that are not "updateable"
- Tuple <Smith, CS> can be deleted from ProfDept by any of the following ways to change the base tables:
  - Deleting row for Smith from Professor (but this is inappropriate if he is still at the University)
  - Deleting row for CS from Department (not what is intended)
  - Updating row for Smith in Professor by setting *DeptId* to null (seems like a good idea, but how would the computer know?)

Updating Views – More issues

Suppose we have:

```
  CREATE VIEW AvgSalary (DeptId, Avg_Sal ) AS
     SELECT  E.DeptId, AVG(E.Salary)
      FROM   Employee E
      GROUP BY E.DeptId
```

then how would we handle:

```
    UPDATE AvgSalary
      SET Avg_Sal = 1.1 * Avg_Sal
```

Which tuples should we change, by how much? There are many ways to achieve the required overall increase; that is why this view is also not considered to be updateable

Agenda

- Overview
- DBMS Access Control
- User identity and authentication
- Views
- *Integrity and constraints*
- Triggers, Stored Procedures

Semantic Integrity Constraints

- Recall: Declaring a constraint on valid states of the data, and enforcing it, is beneficial because
  - It captures semantics of the domain in the database
  - It ensures that authorized changes to the database do not result in a loss of **data consistency**
  - It helps 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

Review: 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)

User-Defined Domains

- PostgreSQL approach: 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)

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

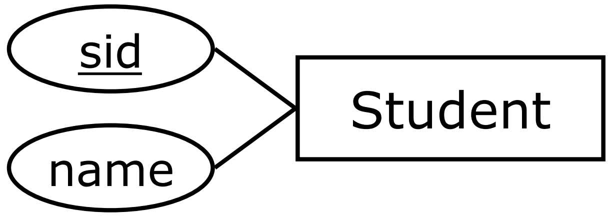
- Domains can be further restricted,e.g. with the **check** clause
  - E.g.: **create domain** Grade **char check**(value in (‘F’,’P’,’C’,’D’,’H’))

- User-defined types with SQL:1999:

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

Review: Primary Key Constraints

- A set of fields is a *candidate 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**

Review: Foreign Keys & Referential Integrity

- **Foreign key :** a set of attributes in a relation that is used to `refer' to a tuple in a parent relation.
  - In many platforms, it must refer to a candidate key of the target 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

Review: Naming Integrity Constraints

- The **CONSTRAINT** clause can be used to name <u>all</u> 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)
);
```

Review: 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 is not created!

  e.g. from Oracle "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.
- Note: Although generalizing nicely the semantic constraints, assertions are <u>not supported</u> by most DBMS

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
  - Some details covered in week 11

Deferring Constraint Checking

- Any dbms-known 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**
    Will 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*.

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

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

Agenda

- Overview
- DBMS Access Control
- User identity and authentication
- Views
- Integrity and constraints
- *Triggers, Stored Procedures*

Triggers

- A **trigger** is a statement that is executed automatically when specified actions occur to the DBMS
  - More flexible than constraints, but also declared and kept inside the dbms
- A trigger declaration conceptually consists of three parts:
  ON *event* **WHEN** *condition* **THEN** *action*
  - *event* (what activates the trigger?)
  - *condition* [optional] (guard / test whether the trigger shall be executed)
  - *action* (what happens if the trigger is executed)

- Triggers were introduced to SQL standard in SQL:1999, but supported even earlier by most platforms using non-standard
  - Some platforms (but not PostgreSQL) allow triggers ON SELECT as well as ON UPDATE/DELETE/INSERT
- In current PostgreSQL, the action must be to run a stored procedure
- See https://en.wikipedia.org/wiki/Database_trigger

Triggers for security

- Monitoring
  - E.g. to react to changes, by sending alerts or storing information elsewhere in the db, for later audit
- Constraint maintenance
  - Triggers can be used to maintain a variety of semantic constraints, more sophisticated than simple domain, Foreign Key, etc
  - This can help reduce damage from adversary attacks by preventing certain changes to the data

Other uses of triggers

- Business rules
  - Some dynamic business rules can be encoded as triggers
- Monitoring
  - Produce logs of activity, which can be used for later performance tuning etc
- Maintenance of auxiliary summary data
  - May allow reports with summaries to be produced faster
  - Careful! Many systems now support *materialized views* which should be preferred rather than maintenance triggers

Trigger Events

- Triggering event can be **insert**, **delete** or **update**

- Triggers on update can be restricted so only occur when specific columns are changed

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

Trigger Timing

- A trigger is declared as BEFORE or AFTER the event
  - BEFORE is often used to enforce constraints, eg raise application-specific errors, adjust data contents etc
  - AFTER is often used for monitoring by keeping audit information (even sending alerts), or for propagating consequential effects

Trigger Granularity

- **Granularity**
  - *Row-level granularity*: change of a single row is an event (a single UPDATE statement might result in multiple events, or none [if no rows are impacted])
  - *Statement-level granularity*: a statement execution is an event (a single UPDATE statement that changes multiple rows is one event).

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

                 INSERT
   BEFORE
                 DELETE                      ON relation-name
   AFTER
                 UPDATE [OF col-name]

 [FOR EACH ROW]   -- optional; otherwise a statement trigger

 [WHEN ( condition )] -- optional
 EXECUTE PROCEDURE stored-procedure-name();
             -- needs to be defined before trigger creation
```

Triggers – PostgreSQL Syntax

- Trigger procedures can be declared using a variety of languages supported by PostgreSQL
  - See later discussion of stored procedures
- 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. https://www.postgresql.org/docs/15/triggers.html]

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, statement-level is 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 (other than those declared as such)
- Use AFTER triggers
  - For audit-keeping, integrity maintenance and update propagation

When Not to Use Triggers

- Triggers were used in previous decades 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

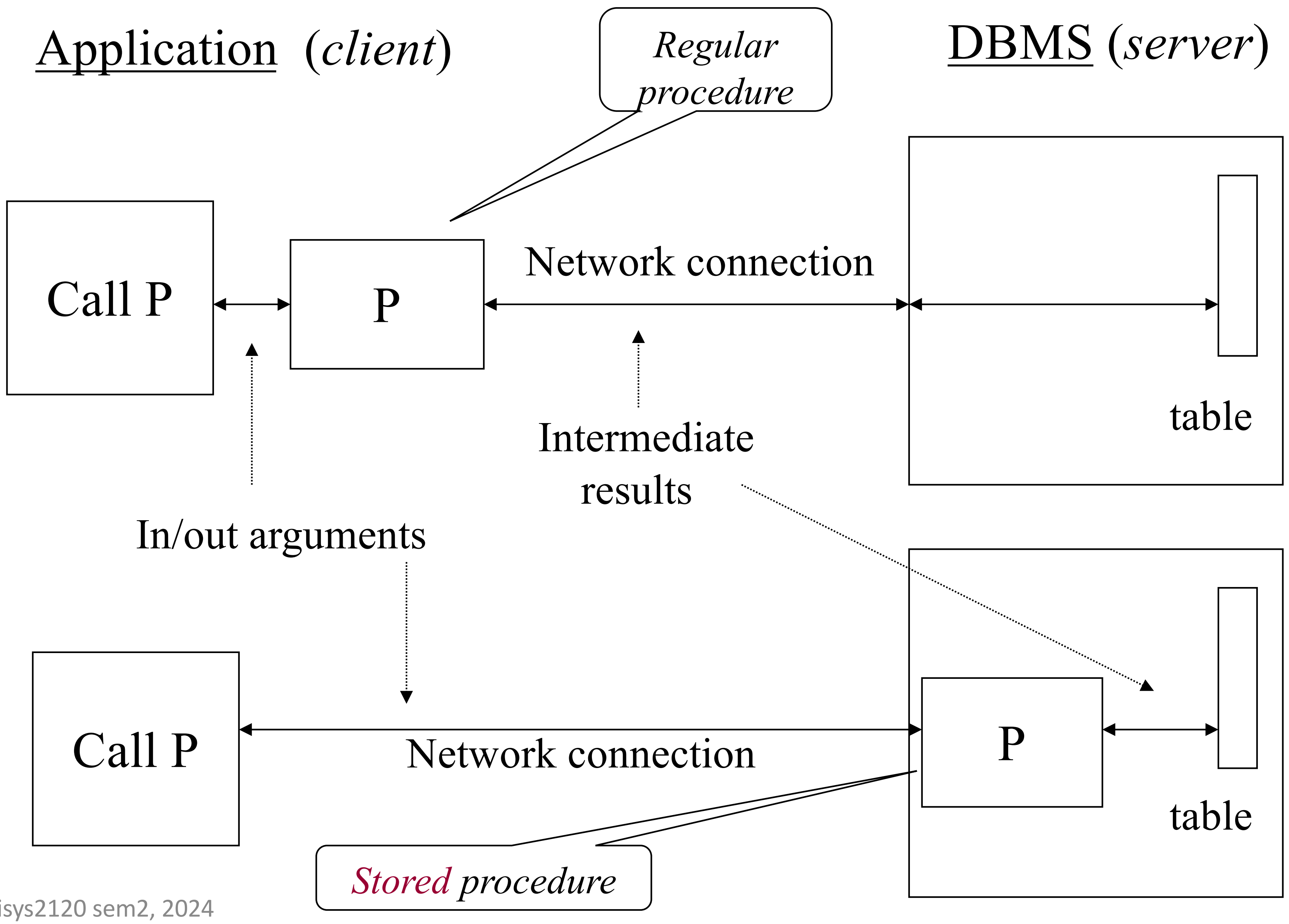
Stored Procedures

- Most DBMS can run application logic (a program written by users rather than by plaform engineers, that does domain-relevant calculations etc) *within the database platform*
  - Included as schema element (stored in DBMS)
  - Invoked by the application
- See https://en.wikipedia.org/wiki/Stored_procedure
- As we saw, the action of a trigger in PostgreSQL must be to run a stored procedure that was already declared

Stored Procedures

- Advantages (compared to code as part of application):
  - Central code-base for all applications
  - Improved maintainability
  - Additional abstraction layer (programmers do not need to know the schema)
  - Improved performance from less communication rounds to/from the dbms, and reduced amount data transfer
  - DBMS-centric security and consistent logging/auditing (important!)

Stored Procedures



SQL/PSM

- Stored Procedures have full access to SQL and also can use programming concepts (loops, conditional, etc)
- All major database systems provide extensions of SQL to a simple, general purpose language [but be careful: these systems have somewhat incompatible syntax and semantics, differing from one another and from the standard]
  - SQL:1999 Standard: SQL/PSM
  - PostgreSQL: PL/pgSQL , also supports PL/Python and others
  - Oracle: PL/SQL
  - Microsoft SQL Server: T-SQL, also supports a variety of languages executing with Common Language Runtime
- Extensions
  - Local variables, loops, if-then-else conditions
- Calling Stored Procedures: CALL statement
  - Example: `CALL ShowNumberOfEnrolments();`

Procedure Declarations

- Procedure Declarations (with SQL/PSM)

```
CREATE FUNCTION name ( parameter1,…, parameterN ) AS
  local variable declarations
   procedure code;
```

- Stored Procedures can have parameters
  - of a valid SQL type (parameter types must match)
  - three different modes
    - IN arguments to procedure
    - OUT return values
    - INOUT combination of IN and OUT

```
CREATE FUNCTION CountEnrolments( IN uos VARCHAR ) AS
         SELECT COUNT(*)
           FROM Enrolled
          WHERE uosCode = uos;

 CALL CountEnrolments (‘INFO2120’);
```

PostgreSQL: PL/pgSQL

- Extents SQL by programming language contructs
  - **CREATE FUNCTION** *name* ***RETURNS*** ... ***AS***...
  - Compound statements: **BEGIN** ... **END**;
  - SQL variables: **DECLARE section**
    variable-name sql-type;
  - Assignments: *variable* **:=** *expression*;
  - IF statement: **IF** *condition* **THEN** ... **ELSE** ... **END IF**;
  - Loop statements: **FOR** *var* **IN** *range* (**WHILE** *cond*)
    LOOP ... **END LOOP**;
  - Return values: **RETURN** *expression*;
  - Call statement: **CALL** procedure(parameters);
  - Transactions: **COMMIT**; **ROLLBACK**;

(cf. https://www.postgresql.org/docs/15/plpgsql.html)

PL/pgSQL Example

Tip: CREATE OR REPLACE to avoid 'name-already-used'

- PL/pgSQL procedure declaration

```
CREATE OR REPLACE FUNCTION
    name ( parameter1, …, parameterN ) RETURNS
   sqlType
AS $$
DECLARE
   variable    sqlType;
   …
BEGIN
      …
END;
$$ LANGUAGE plpgsql;
```

optional

Tip: final delimiter must match the one used after AS

- where parameterX is declared as (IN is default):
  [IN|OUT|IN OUT] ***name sqlType***

PostgreSQL PL/pgSQL Example

```
CREATE OR REPLACE FUNCTION GradeStudent
   (studId INTEGER, uos VARCHAR) RETURNS CHAR AS $$
   DECLARE
    grade  CHAR;
        marks  INTEGER;
   BEGIN
        SELECT SUM(mark) INTO marks
      FROM Assessment
     WHERE sid=$1 AND uosCode=$2;
    IF     ( marks>84 ) THEN grade := 'H';
    ELSIF  ( marks>74 ) THEN grade := 'D';
    ELSIF  ( marks>64 ) THEN grade := 'C';
    ELSIF  ( marks>50 ) THEN grade := 'P';
    ELSE                     grade := 'F';
    END IF;
    RAISE NOTICE 'Final grade is: %s', grade;
    RETURN grade;
   END;
$$ LANGUAGE plpgsql;
```

References

- Silberschatz/Korth/Sudarshan(7ed)
  - Chapters 4.2, 4.4, 4.7, 5.3

Also

- Kifer/Bernstein/Lewis(complete version, 2ed)
  - Chapter 3.3.3-3.3.9, 3.3.12, 7, 26
  - Most of ch 26 is about security of internet systems, which we cover in week 8
- Ramakrishnam/Gehrke(3ed)
  - Chapter 3.6, 5.7-5.9, 21
  - This also includes mandatory access control, which we do not require
- Garcia-Molina/Ullman/Widom(complete book, 2ed)
  - Chapter 7.1-7.5, 8.1-8.2, 10.1

Summary

- Security concepts
- DBMS Mechanisms: privileges, GRANT/REVOKE, DBMS accounts
- Views, useful for content-based access and summary access
- Declared constraints
- Active enforcement: triggers (use stored procedures)