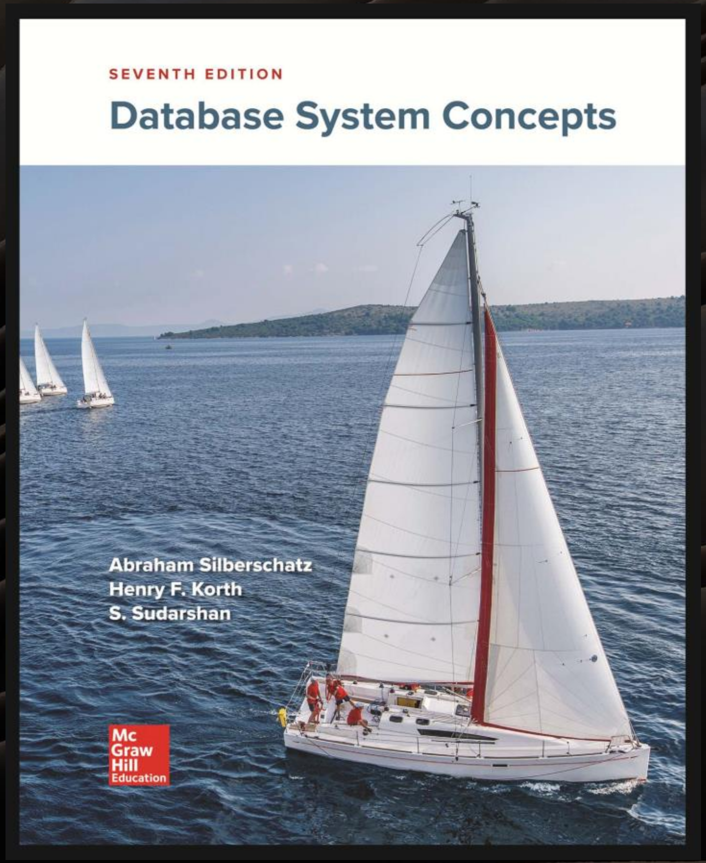
IF2240 Basis Data/II2250 Manajemen Basisdata
Integrity Constraints



References

Silberschatz, Korth, Sudarshan: "Database System Concepts", 7th Edition

- Section 4.4: Integrity Constraints
- Section 5.2: Functions and Procedures
- Section 5.3: Triggers

KNOWLEDGE & SOFTWARE ENGINEERING

OBJECTIVES

Understand the importance of Integrity Constraints

Design and implement constraints and assertions in a database

Design a trigger that has some control flow, to provide a given functionality

Design a function/procedure to allow "business logic"/rules to be implemented in SQL

Integrity Constraints

Integrity refers to the *consistency* of data in the database

- Integrity constraints cannot enforce *truth* or *correctness* of data

Integrity constraints guard against accidental damage to the database, by ensuring that authorized changes to the database do not result in a loss of data consistency.

A database might be subject to any number of integrity constraints, of arbitrary complexity. E.g.:

- A checking account must have a balance greater than $10,000.00
- A salary of a bank employee must be at least $4.00 an hour
- A customer must have a (non-null) phone number

Constraint Definition

Example:

```
CONSTRAINT SC3
  IS_EMPTY (S WHERE STATUS < 1 OR STATUS > 100);
```

When a new constraint is defined:

- The system must first make sure the database currently satisfies it
  - If it does not, the new constraint will be rejected
  - Otherwise, it is accepted and enforced from that point forward

Two approaches of constraint definitions:

- Declarative integrity support
  - As part of system
- Procedural integrity support
  - Must be declared in all application programs.

Constraint Examples

An instructor name cannot be *null*.

No two instructors can have the same instructor ID.

Every department name in the *course* relation must have a matching department name in the *department* relation.

The budget of a department must be greater than $0.00.

A Constraint Classification Scheme

1. **A type constraint**: specifies the legal values for a given type
2. **An attribute constraint**: specifies the legal values for a given attribute
3. **A relation (relvar) constraint**: specifies the legal values for a given relation
4. **A database constraint**: specifies the legal values for a given database

1. Type Constraints

A type constraint is an enumeration of the legal values of the type

Example:

```
TYPE WEIGHT POSSREP (RATIONAL)
       CONSTRAINT THE_WEIGHT (WEIGHT) > 0.0;
```

Type constraint can always be thought of as being checked during the execution of some selector invocation

- Type constraints are checked immediately

2. Attribute Constraints

An attribute constraint is basically just a declaration to the effect that a specified attribute of a specified relation is of a specified type

Example:

```
VAR S BASE RELATION (
  S#          S#,
  SNAME       SNAME, … );
```

In this relation, values of attributes S#, SNAME, … are constrained to be of types S#, SNAME, …, respectively

Attribute constraints are part of the definition of the attribute in question, and they can be identified by means of the corresponding attribute name

3. Relation Constraints

A relation constraint is a constraint on an individual relation

- It is expressed in terms of the relation in question only

Example:

```
CONSTRAINT SC5
  IS_EMPTY (S WHERE CITY = 'London' and status <> 20);
```

Relation constraints are always checked immediately

4. Database Constraints

A database constraint is a constraint that interrelates two or more distinct relations

Example:

```
CONSTRAINT DBC1
  IS_EMPTY ( (S JOIN SP)
                  WHERE STATUS < 20 AND QTY > QTY(500) );
```

In general, database constraint checking cannot be done immediately, but must be *deferred* to end of transaction

- If a database constraint is violated at the end of transaction, the transaction is rolled back

State vs. Transition Constraints

State constraints concern with **consistent states** of the database

Transition constraints concern with legal transitions from one correct state to another

- Example: changes or marital status

Valid transitions:	Invalid transitions:
never married to married	never married to widowed
married to widowed	never married to divorced
widowed to married	widowed to divorced

State vs. Transition Constraints (Cont.)

Example: no supplier's status must ever decrease

```
CONSTRAINT TRC1 IS_EMPTY
  ( ( ( S' (S#, STATUS) RENAME STATUS AS STATUS')
       JOIN S (S#, STATUS) )
     WHERE STATUS' > STATUS );
```

S' is referred to the corresponding relation as it was prior to the update

Transition constraints are only applied to relation and database constraints:

- A relation transition constraint can be checked immediately, while a database transition constraint would be deferred

Integrity in Relational Model

The integrity part of the relational model is the part that has evolved the most

- The original emphasis was on **primary** and **foreign keys** specifically
- Gradually, the importance of integrity constraints in general has begun to be better understood

Keys – Candidate Keys

Let *K* be a set of attributes of relation *R*. Then *K* is a candidate key for *R* if and only if it possesses both of the following properties:

- Uniqueness: no legal value of *R* ever contains two distinct tuples with the same value for *K*
- Irreducibility: no proper subset of *K* has the uniqueness property
  - Is needed to enforce the associated constraint properly

Candidate key provide the basic tuple-level addressing mechanism

Keys – Primary Keys

A superset of a candidate key is a *superkey*

A primary key is a candidate key that is selected to be the main key of the relation

- All the other candidate keys are called alternate keys

Keys – Foreign Keys

Let *R2* be a relation. Then a foreign key in *R2* is a set of attributes of *R2*, say *FK*, such that:

- There exists a relation *R1* (not necessarily a distinct relation) with a candidate key *CK*, and
- For all time, each value of *FK* in the current value of *R2* is identical to the value of *CK* in some tuple in the current value of *R1*.

In practice, the corresponding candidate key usually is the primary key of *R1*.

Foreign Keys

The candidate key corresponding to some given foreign key might contain a value that does not currently appears as a value of that foreign key

A foreign key can be simple or composite, depends to its matching candidate key

A foreign key value represents a reference to the tuple containing the matching candidate key value (the referenced tuple)

- Referential integrity problem: ensuring that the database does not include any invalid foreign key values
- Referential constraint
- Referencing relation vs. referenced relation

Foreign Keys (Cont.)

The referential constraints among the relations in a database can be represented using referential diagram.

A relation can be both referencing and referenced

- Referential path the chain of arrows from one relation to another relation

A relation is called self-referencing when its foreign key referring to its candidate key

- This may lead to referential cycle

Foreign-to-candidate-key matches is the “glue” that holds the database together

Referential integrity rule: the database must not contain any unmatched foreign key values

Referential Actions

Cascade: the delete/update cascades to delete/update the tuples related

Restrict: the delete/update is restricted to the ones that will not violate referential integrity constraints

No Action: the delete/update is performed exactly as requested

Remember: database updates are always atomic

Referential Constraints in a Transaction

Transactions may consist of several steps, and integrity constraints may be violated temporarily after one step,

a later step may remove the violation.

For instance, suppose we have a relation *person* with primary key *name*, and an attribute *spouse*, and suppose that *spouse* is a foreign key on *person*.

SQL Support for Integrity Constraints

Integrity Constraints

Integrity constraints guard against accidental damage to the database, by ensuring that authorized changes to the database do not result in a loss of data consistency.

- A checking account must have a balance greater than $10,000.00
- A salary of a bank employee must be at least $4.00 an hour
- A customer must have a (non-null) phone number

Integrity Constraints on a Single Relation

not null

primary key

unique

check (P), where P is a predicate

Not Null and Unique Constraints

not null

- Declare name and budget to be **not null**

  name **varchar**(20) **not null**

  budget **numeric**(12,2) **not null**

unique (A_1, A_2, …, A_m)

- The unique specification states that the attributes A_1, A_2, …, A_m form a candidate key.
- Candidate keys are permitted to be null (in contrast to primary keys).

The check clause

check (P)

- where P is a predicate

Example:
ensure that semester is one of fall, winter, spring or summer:

```
create table section (
    course_id varchar (8),
    sec_id varchar (8),
    semester varchar (6),
    year numeric (4,0),
    building varchar (15),
    room_number varchar (7),
    time slot id varchar (4),
    primary key (course_id, sec_id, semester, year),
    check (semester in ('Fall', 'Winter', 'Spring', 'Summer'))
);
```

Referential Integrity

Ensures that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.

- Example: If "Biology" is a department name appearing in one of the tuples in the instructor relation, then there exists a tuple in the department relation for "Biology".

Let A be a set of attributes. Let R and S be two relations that contain attributes A and where A is the primary key of S. A is said to be a foreign key of R if for any values of A appearing in R these values also appear in S.

Cascading Actions in Referential Integrity

```
create table course (
    course_id    char(5) primary key,
    title        varchar(20),
    dept_name   varchar(20) references department
)

create table course (
    …
    dept_name varchar(20),
    foreign key (dept_name) references department
            on delete cascade
            on update cascade,
    …
)

-- alternative actions to cascade:  set null, set default
```

Integrity Constraint Violation During Transactions

E.g.

```
create table person (
        ID  char(10),
        name char(40),
        mother char(10),
        father  char(10),
        primary key ID,
        foreign key father references person,
        foreign key mother references  person)
```

How to insert a tuple without causing constraint violation ?

- insert father and mother of a person before inserting person
- OR, set father and mother to null initially, update after inserting all persons (not possible if father and mother attributes declared to be not null)
- OR defer constraint checking

Complex Check Clauses

check (time_slot_id **in** (**select** time_slot_id **from** time_slot))

- why not use a foreign key here?

Every section has at least one instructor teaching the section.

- how to write this?

Unfortunately: subquery in check clause not supported by pretty much any database

- Alternative: **triggers** (later)

create assertion <assertion-name> **check** <predicate>;

- Also not supported by anyone

Trigger

Triggers

A **trigger** is a statement that is executed automatically by the system as a side effect of a modification to the database.

To design a trigger mechanism, we must:

- Specify the conditions under which the trigger is to be executed.
- Specify the actions to be taken when the trigger executes.

Triggers introduced to SQL standard in SQL:1999, but supported even earlier using non-standard syntax by most databases.

- Syntax illustrated here may not work exactly on your database system; check the system manuals

Triggering Events and Actions in SQL

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

Triggers on update can be restricted to specific attributes
- For example, **after update** of takes on grade

Values of attributes before and after an update can be referenced
- **referencing old row as** : for deletes and updates
- **referencing new row as** : for inserts and updates

Triggers can be activated before an event, which can serve as extra constraints. For example, convert blank grades to null.

```
create trigger setnull_trigger before update of takes
referencing new row as nrow
for each row
when (nrow.grade = ' ')
begin atomic
        set nrow.grade = null;
end;
```

Trigger to Maintain credits_earned value

```
create trigger credits_earned after update of takes on (grade)
referencing new row as nrow
referencing old row as orow
for each row
when nrow.grade <> 'F' and nrow.grade is not null
    and (orow.grade = 'F' or orow.grade is null)
begin atomic
    update student
    set tot_cred= tot_cred +
         (select credits
          from course
          where course.course_id= nrow.course_id)
    where student.id = nrow.id;
end;
```

Statement Level Triggers

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**
- 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

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

Encapsulation facilities can be used instead of triggers in many cases

- Define methods to update fields
- Carry out actions as part of the update methods instead of through a trigger

When Not To Use Triggers (Cont.)

Risk of unintended execution of triggers, for example, when

- Loading data from a backup copy
- Replicating updates at a remote site
- Trigger execution can be disabled before such actions.

Other risks with triggers:

- Error leading to failure of critical transactions that set off the trigger
- Cascading execution

Functions and Procedures

Use of Functions and Procedures

Procedures and functions allow "business logic" to be stored in the database and executed from SQL statements.

For example: university rules

- how many courses a student can take in a given semester,
- the minimum number of courses a full-time instructor must teach in a year,
- the maximum number of majors a student can be enrolled in,
- and so on.

Use of Functions and Procedures (cont.)

While such business logic can be encoded as programming-language procedures stored entirely outside the database, defining them as stored procedures in the database has several advantages.

For example:

- it allows multiple applications to access the procedures
- it allows a single point of change in case the business rules change, without changing other parts of the application. Application code can then call the stored procedures, instead of directly updating database relations

Functions and Procedures

SQL:1999 supports functions and procedures

- Functions/procedures can be written in SQL itself, or in an external programming language (e.g., C, Java).
- Functions written in an external languages are particularly useful with specialized data types such as images and geometric objects.
  - Example: functions to check if polygons overlap, or to compare images for similarity.
- Some database systems support **table-valued functions**, which can return a relation as a result.

SQL:1999 also supports a rich set of imperative constructs, including

- Loops, if-then-else, assignment

Many databases have proprietary procedural extensions to SQL that differ from SQL:1999.

SQL Functions

Define a function that, given the name of a department, returns the count of the number of instructors in that department.

```
create function dept_count (dept_name varchar(20))
returns integer
begin
        declare d_count  integer;
                select count (* ) into d_count
                from instructor
                where instructor.dept_name = dept_name
        return d_count;
end
```

The function dept_count can be used to find the department names and budget of all departments with more that 12 instructors.

```
select dept_name, budget
from department
where dept_count (dept_name ) > 12
```

SQL functions (Cont.)

Compound statement: **begin** … **end**

- May contain multiple SQL statements between **begin** and **end**.

returns -- indicates the variable-type that is returned (e.g., integer)

return -- specifies the values that are to be returned as result of invoking the function

SQL function are in fact parameterized views that generalize the regular notion of views by allowing parameters.

Table Functions

SQL:2003 added functions that return a relation as a result

Example: Return all instructors in a given department

```
create function instructor_of (dept_name char(20))
returns table  (
            ID varchar(5),
        name varchar(20),
            dept_name varchar(20),
        salary numeric(8,2))
return table
       (select ID, name, dept_name, salary
        from instructor
        where instructor.dept_name = instructor_of.dept_name)
```

Usage

```
select *
from table (instructor_of ('Music'))
```

SQL Procedures

The *dept_count* function could instead be written as procedure:

```
create procedure dept_count_proc (in dept_name varchar(20),
                                  out d_count integer)
begin

    select count(*) into d_count
     from instructor
     where instructor.dept_name = dept_count_proc.dept_name

end
```

Procedures can be invoked either from an SQL procedure or from embedded SQL, using the **call** statement.

```
declare d_count integer;
call dept_count_proc( 'Physics', d_count);
```

Procedures and functions can be invoked also from dynamic SQL

SQL:1999 allows more than one function/procedure of the same name (called name overloading), as long as the number of arguments differ, or at least the types of the arguments differ

Language Constructs for Procedures & Functions

SQL supports constructs that gives it almost all the power of a general-purpose programming language.

- Warning: most database systems implement their own variant of the standard syntax below.

Compound statement: **begin** … **end**

- May contain multiple SQL statements between begin and end.
- Local variables can be declared within a compound statements

While and repeat statements:

```
while boolean expression  do
        sequence of statements ;
end while
repeat
          sequence of statements ;
until boolean expression
end repeat
```

Language Constructs (Cont.)

For loop

- Permits iteration over all results of a query

Example: Find the budget of all departments

```
declare n  integer default 0;
for r  as
      select budget from department
do
      set n = n + r.budget
end for
```

Language Constructs (Cont.)

Conditional statements (if-then-else)
SQL:1999 also supports a case statement similar to C case statement

Example procedure: registers student after ensuring classroom capacity is not exceeded

- Returns 0 on success and -1 if capacity is exceeded

Signaling of exception conditions, and declaring handlers for exceptions

```
declare out_of_classroom_seats  condition
declare exit handler for out_of_classroom_seats
begin
        …
        ..  signal out_of_classroom_seats
end
```

- The handler here is **exit** -- causes enclosing **begin..end** to be exited
- Other actions possible on exception