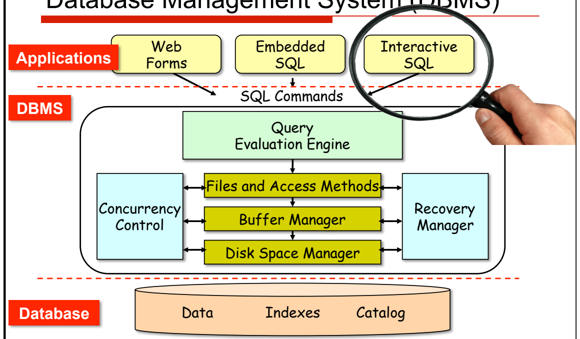
Database Management System (DBMS)



Relational Database Management Systems

- Data Abstraction
  - Overview
  - Integrity Constraints
  - SQL Queries
  - Views
  - Complex Integrity Constraints

Conceptual Evaluation Strategy

- Semantics of an SQL query defined in terms of the following conceptual evaluation strategy:

1. Compute the cross-product of ***from-list***
2. Discard resulting tuples that fail ***qualifications***
3. Delete attributes that are not in ***select-list***
4. If `DISTINCT` is specified, eliminate duplicate rows

Aggregate Queries

Aggregate Operators

1. `COUNT` ([`DISTINCT`] A):
   - The number of (unique) values in column A
2. `SUM` ([`DISTINCT`] A):
   - The sum of all (unique) values in column A
3. `AVG` ([`DISTINCT`] A):
   - The average of all (unique) values in column A
4. `MAX` (A):
   - The maximum value in column A
5. `MIN` (A):
   - The minimum value in column A

Sample Database Instances

BID	*BName*	*Color*
101	Star	blue
102	Star	red
103	Clipper	green
104	Marine	red

Instance *B* of *Boats*

SID	*SName*	*Rating*	*Age*
28	Yuppy	9	35
31	Lubber	8	55
44	Rusty	5	35
58	Yuppy	10	35

Instance *S* of *Sailors*

SID	*BID*	*Day*
22	101	10/10/96
58	103	11/12/96

Instance *R* of *Reserves*

Examples

- Find the average age of all sailors:

SELECT AVG (*S.age*)
FROM *Sailors S*

- Find the average age of all sailors with rating 10:

SELECT AVG (*S.age*)
FROM *Sailors S*
WHERE *S.rating = 10*

Examples

- Count the number of Sailors

SELECT COUNT (*)
FROM *Sailors S*

- Count the number of different sailor names

SELECT COUNT (**DISTINCT** *S.name*)
FROM *Sailors S*

Aggregation

- So far, we have applied aggregate operators to all "qualifying" tuples
- Sometimes, we want to apply aggregates to <u>each</u> of several **groups** of tuples

Aggregation

- Consider: *Find the age of the youngest sailor at each rating level*

SID	*SName*	*Rating*	*Age*
22	Dustin	7	45
29	Brutus	1	33
31	Lubber	8	55
32	Andy	8	25
58	Rusty	10	35
64	Jim	7	35
71	Zorba		
74	Jim	9	35
85	Art	3	25
95	Bob	3	65

Aggregation

- Suppose we know that rating values go from 1 to 10; we can write 10 queries that look like:

SELECT MIN (*S.age*)
FROM *Sailors S*
WHERE *S.rating = i* *i = 1, 2, …, 10*

- But, in general: **1)** we don't know how many rating levels exist; and **2)** what the rating values for these levels are!

Using the GROUP BY Clause

SELECT *S.rating*, **MIN** (*S.age*)
FROM *Sailors S*
GROUP BY *S.rating*

Is GROUP BY enough?

Consider: Find the number of sailors at each rating level (1,2, …) that has at least 2 sailors

```
SELECT S.rating, COUNT(*)
FROM Sailors S
GROUP BY S.rating
```

SID	SName	Rating	Age
22	Dustin	7	45
29	Brutus	1	33
31	Lubber	8	55
32	Andy	8	25
58	Rusty	10	35
64	Jim	7	35
71	Zorba		
74	Jim	9	35
85	Art	3	25
95	Bob	3	65

Is GROUP BY enough?

Consider: Find the number of sailors at each rating level (1,2, …) that has at least 2 sailors

```
SELECT S.rating, COUNT(*)
FROM Sailors S
GROUP BY S.rating
HAVING COUNT (*) > 1
```

SID	SName	Rating	Age
22	Dustin	7	45
29	Brutus	1	33
31	Lubber	8	55
32	Andy	8	25
58	Rusty	10	35
64	Jim	7	35
71	Zorba		
74	Jim	9	35
85	Art	3	25
95	Bob	3	65

Queries with GROUP BY and HAVING

```
SELECT [DISTINCT] select-list
FROM from-list
WHERE qualification
GROUP BY grouping-list
HAVING group-qualification
```

Conceptual Evaluation

1. Compute cross-product of tables in *from-list*
2. Apply the qualifications in the **WHERE** clause
3. Eliminate unwanted columns
   - Only columns in **SELECT**, **GROUP BY**, or **HAVING** are necessary
4. **Sort** table according to **GROUP BY** clause
5. **Apply group-qualification** in the **HAVING** clause
6. **Generate** one answer for each remaining group

Example: Initially…

```
SELECT S.rating, COUNT (*)
FROM Sailors S
GROUP BY S.rating
HAVING COUNT (*) > 1
```

SID	*SName*	*Rating*	*Age*
22	Dustin	7	45
29	Brutus	1	33
31	Lubber	8	55
32	Andy	8	25
58	Rusty	10	35
64	Jim	7	35
74	Jim	8	35
85	Art	3	25
95	Bob	3	65

Steps 1-4

Eliminate Columns

~~*SID*~~	~~*SName*~~	*Rating*	~~*Age*~~
~~22~~	~~Dustin~~	7	~~45~~
~~29~~	~~Brutus~~	1	~~33~~
~~31~~	~~Lubber~~	8	~~55~~
~~32~~	~~Andy~~	8	~~25~~
~~58~~	~~Rusty~~	10	~~35~~
~~64~~	~~Jim~~	7	~~35~~
~~74~~	~~Jim~~	8	~~35~~
~~85~~	~~Art~~	3	~~25~~
~~95~~	~~Bob~~	3	~~65~~

Sort

Rating
1
3
3
7
7
8
8
8
10

Steps 5-6

Rating
~~1~~
3
3
7
7
8
8
8
~~10~~

Apply Group Qualifications → Generate Answers

Rating	COUNT(*)
3	2
7	2
8	3

Another example

Consider: Find the average age of sailors at each rating level that has at least 2 sailors

```
SELECT S.rating, AVG(age)
FROM Sailors S
GROUP BY S.rating
HAVING COUNT (*) > 1
```

SID	SName	Rating	Age
22	Dustin	7	45
29	Brutus	1	33
31	Lubber	8	55
32	Andy	8	25
58	Rusty	10	35
64	Jim	7	35
71	Zorba		
74	Jim	9	35
85	Art	3	25
95	Bob	3	65

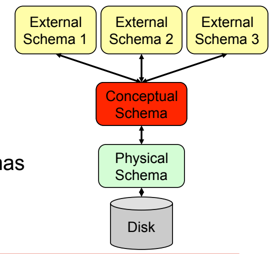
Relational Database Management Systems

- Data Abstraction
  - Overview
  - Integrity Constraints
  - SQL Queries
  - Views
  - Complex Integrity Constraints

Levels of Abstraction in a DBMS

- The data in a DBMS is described at three levels of abstraction:

  1. *Conceptual Schema*
  2. *Physical Schema*
  3. *External Schema*



  - Many external schemas
  - plus one conceptual
  - plus one physical

Conceptual Schema

- What we have seen so far is: **Conceptual Schema**
- Example: in a university database
  - Students (*sid*: `string`, *name*: `string`, *age*: `int`, *gpa*: `real`)
  - Courses (*cid*: `string`, *cname*: `string`)
  - Enrolled (*sid*: `string`, cid: `string`, *grade*: `string`)

SID	*Name*	*Age*	*GPA*
546007	Peter	18	3.8
546100	Bob	19	3.65
546500	Bill	20	3.7

CID	*CName*
CSC 343	DB
CSC 207	SW
CSC 369	OS

SID	*CID*	*Grade*
546007	CSC 343	A
546007	CSC 369	B+
546100	CSC 343	B

Conceptual Schema

- **Conceptual database design:** the process of designing a conceptual schema
- Sometimes called logical schema
- Describes data in terms of data model
- In a relational DBMS:
  - Describes all relations (tables) in database

Physical Schema

- Describes how relations are stored on disk
- It specifies:
  - File organizations
    - Example: unsorted files of records
  - Indexes
    - Example: index on *SID* & *CID*

- Designing a physical schema is based on how data is typically accessed
- **Physical database design:** the process of designing a physical database schema

External Schema

- Allows data access to be customized and authorized at the user-level
  - Defined in terms of data model
  - Consists of a collection of **views**
  - Guided by end-user requirements
  - Example:
    - CourseTotalEnroll (*course name*: `string`, *total enrollment*: `int`)

External Schema

Course Name	*Total Enrollment*

Course Name	*Total Enrollment*
DB	2
SW	0
OS	1

SID	*Name*	*Age*	*GPA*
546007	Peter	18	3.8
546100	Bob	19	3.65
546500	Bill	20	3.7

CID	*CName*
CSC 343	DB
CSC 207	SW
CSC 369	OS

SID	*CID*	*Grade*
546007	CSC 343	A
546007	CSC 369	B+
546100	CSC 343	B

Data Independence

- Applications insulated from how data is structured and stored
  - **Logical data independence:**
    - Protection from changes in logical data structure
    - Achieved by **external schema**
    - Examples: add columns, restructure schema
  - **Physical data independence:**
    - Protection from changes in physical data structure
    - Achieved by **conceptual scheme**
    - Examples: add/drop index, change file order

Views

Specification of Views

- SQL command: **`CREATE VIEW`**
  - A view name
  - A query to specify the table contents

Example:

SID	Name	Age	GPA	Major
546007	Peter	18	3.8	IT
546100	Bob	19	3.65	MM
546500	Bill	20	3.7	IT

Student table

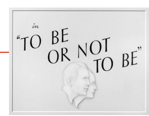
Creating a view:

```
CREATE VIEW IT_Students AS
SELECT name, age
FROM Student
WHERE Major = 'IT'
```

Using a view:

```
SELECT name
FROM IT_Students
WHERE age > 19
```


What is a view?



- It is a table:
  - as it can be queried just like a table!
- It is not a table:
  - as it does not physically exist!
- A view is a **"virtual table"** derived from **base tables**
- A view is a **"named query"**

Advantages of Views:

1. Logical independence
2. For **convenience** and clarity when writing queries
   - Views can be used just like tables
3. For **security**
   - Different data **access privileges** can be given to different users (i.e., authorization)

Views for Authorization

```
CREATE VIEW IT_Students AS
SELECT name, class
FROM Student
WHERE Major = 'IT'
```

```
GRANT SELECT ON IT_Students TO PUBLIC;
GRANT UPDATE ON IT_Students TO staff;
```

```
CREATE ROLE staff;
GRANT staff TO Sharaf;
GRANT staff TO Sutton;
```


Efficient View Implementation

- A DBMS implements views in two ways:
  - ✓ Query Modification
  - ✓ View Materialization
- With expected trade-offs…

Query Modification 1

View

```
CREATE VIEW IT_Students AS
SELECT name, age
FROM Student
WHERE Major = 'IT'
```

Original Query (user)

```
SELECT name
FROM IT_Students
where age > 19
```

Modified Query (DBMS)

```
SELECT name
FROM Student
WHERE Major = 'IT'
AND age>19
```

Query Modification 2

View

```
CREATE VIEW IT_Students AS
SELECT name, age
FROM Student
WHERE Major = 'IT'
   OR Major = 'MM'
```

Original Query

```
SELECT name
FROM IT_Students
where age > 19
```

Modified Query

```
SELECT name
FROM Student
WHERE (Major = 'IT'
   OR Major = 'MM')
   AND age>19
```


Query Modification 3

Specify attribute names

View

```
CREATE VIEW Majors (major, mtotal) AS
SELECT major, count(*)
FROM Student
GROUP BY Major
```

Original Query

```
SELECT *
FROM Majors
WHERE mtotal > 100
```

Modified Query

```
SELECT major, count(*)
FROM Student
GROUP BY Major
HAVING COUNT(*)>100
```


Query Modification

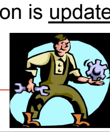
- Query modification:
  - presents the view query in terms of a query on the underlying base tables
- Disadvantage:
  - **re-compute** the view with every query
    - E.g., multiple queries SELECT name FROM IT_Students where age > 19, 20, 21, …
  - inefficient for views defined via complex queries (e.g., aggregate queries)

Virtual vs. Materialized Views

- Views:
  - Virtual tables
  - Evaluating a view (query) creates its data
- Materialized Views:
  - **Stored tables**
  - Physically store the view (query) **and its data**

Materialized Views

- Advantage:
  - Avoid re-computing the view with every query
  - Assumption: more queries can use the same view
- **But,** materialized **view maintenance** is needed
  - A materialized view should be updated when any **base table** used in the view definition is updated



Example:

SID	*Name*	*Age*	*GPA*	*Major*
546007	Peter	18	3.8	IT
546100	Bob	19	3.65	MM
546500	Bill	20	3.7	IT

Student table

Update on base table:

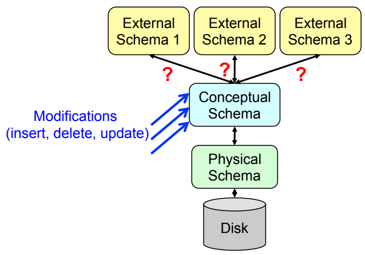
```
INSERT INTO Student
VALUES (456, …, MM)
```

```
CREATE VIEW Majors (major, mtotal) AS
SELECT major, count(*)
FROM Student
GROUP BY Major
```

Updated View:

major	*mtotal*
IT	2
MM	2

Updating Materialized Views



Updating Materialized Views

- Efficient strategies for automatically updating the materialized view when base tables are updated
  - Avoid re-computing the view from "scratch"
  - **Incremental update:**
    - determines what new tuples must be inserted, deleted, or modified in the view when an update is applied to the base tables

Trade-offs in view implementation

	(Virtual) Views	**Materialized Views**
Queries on Views		
Updates on Base Tables		

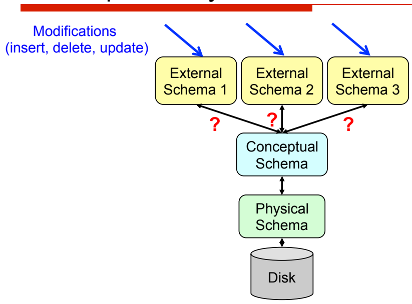
Trade-offs in view implementation

	(Virtual) Views	Materialized Views
Queries on Views	Re-compute view	Re-use view
Updates on Base Tables	Do nothing	View Maintenance

Updates on base tables vs. views

- All updates <u>on base tables</u> are reflected in corresponding views
  - In virtual views: query evaluation
  - In materialized views: view maintenance
- How about updates to views?!
  - A user expects that updates <u>on a view</u> will also be reflected in base table(s)
  - View updatability…

View Updatability



Changes ocming from the external views

Example 1

SID	Name	Age	GPA	Major
546007	Peter	18	6.0	IT
546100	Bob	19	6.6	MM
546500	Peter	20	3.0	IT

Student **table**

Updating a view:

```
UPDATE GoodStudents
SET gpa= 6.5
WHERE sid=546007
```

```
CREATE VIEW GoodStudents AS
SELECT sid, gpa
FROM Student
WHERE gpa> 3.5
```

Mohamed Sharaf, Univ. of Queensland INFS2200/7903 - Semester 1, 2014 49

Selecting student ID and GPA
of students with GPA 3.5

Update to 6.5 for student
with student ID 546007

Example 2

SID	Name	Age	GPA	Major
546007	Peter	18	6.0	IT
546100	Bob	19	6.6	MM
546500	Peter	20	3.0	IT

Student **table**

Updating a view:

```
INSERT INTO
    GoodStudents
    VALUES (Jane, 4.0)
```

What is Jane's SID?!

```
CREATE VIEW GoodStudents AS
SELECT name, gpa
FROM Student
WHERE gpa> 3.0
```

Mohamed Sharaf, Univ. of Queensland INFS2200/7903 - Semester 1, 2014 50

Cannot insert,
neither of them are primary keys

The DBMS doesn't have
enough information to find
the equivalent information
of the tuple

Example 3

SID	Name	Age	GPA	Major
546007	Peter	18	6.0	IT
546100	Bob	19	6.6	MM
546500	Peter	20	3.0	IT

Student **table**

Updating a view:

```
UPDATE Majors
SET agpa= 5.0
WHERE major=IT
```

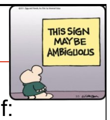
```
CREATE VIEW Majors (major, agpa) AS
SELECT major, avg(gpa)
FROM Student
GROUP BY Major
```

Infinite possibilities of values!

Mohamed Sharaf, Univ. of Queensland INFS2200/7903 - Semester 1, 2014 51

Infinite possibilities of
values. Incrementing any
of the GPAs (or both) for
a total of 1 will increase
the GPA to 5.

View Updateability



- In general, a view is called **updateable** if:

all updates on the view can be unambiguously translated back to tuples in the base tables

- A view update is **unambiguous** if:

Only one update on the base tables can accomplish the desired update effect on the view

- In general, a view is **not updateable** if:

an update on a view can be mapped to more than one possible update on the base tables

SQL Standard for View Updateability

1. A view with a single defining table is updatable if the view attributes contain the primary key
2. Views defined using aggregate functions are not updatable
3. Views defined on multiple tables using joins are generally not updatable

SQL will NOT allow a View to be updated on most systems. Some database systems will allow it, but generally not.

Aggregate functions or joins in views make them un updatable

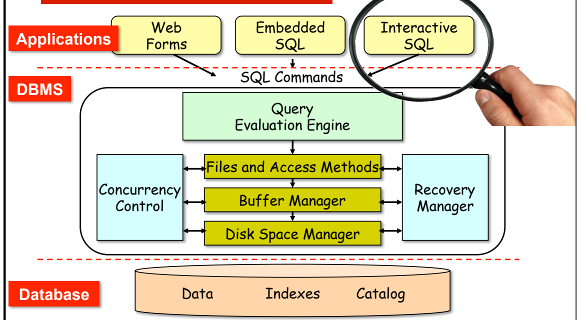
Relational Database Management Systems

- Data Abstraction
  - Overview
  - Integrity Constraints
  - SQL Queries
  - Views
  - Complex Integrity Constraints

If it is updatable, first the DBMS needs to know it is updatable

PRACTICAL INFO: How to create a virtual/materialized view? You will see the execution time.

Database Management System (DBMS)



Complex Integrity Constraints

W

- Three DDL constructs
  - Checks
  - Assertions
  - Triggers

Example Schema

SID	*Name*	*Age*	*GPA*	*Major*
546007	Peter	18	3.8	IT
546100	Bob	19	6.65	Cinema
546500	Peter	20	4.7	History

What kind of constraints might
I want to reinforce?
Primary key constraint
Domain constraint (sID should
be an integer)
Foreign key constraint

Example Schema

```
CREATE TABLE Student (
    Sid INTEGER,  Name CHAR(20),
    Age INTEGER,
    GPA REAL, CHECK (GPA >= 0.0 AND GPA <= 7.0)
    Major CHAR(10),
       CHECK Major in ('IT', 'Cinema', 'History');
    PRIMARY KEY (Sid));
```

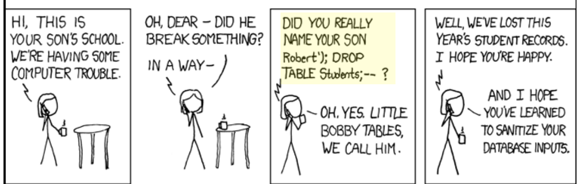

CHECK Constraint 1

```
CREATE TABLE Student (
    Sid INTEGER,  Name CHAR(20),
    Age INTEGER,
    GPA REAL,
       CHECK (GPA>=0.0 AND GPA <= 7.0);
    Major CHAR(10),

    PRIMARY KEY (Sid));
```


CHECK Constraint 2

```
CREATE TABLE Student (
    Sid INTEGER,  Name CHAR(20),
    Age INTEGER,
    GPA REAL,
    Major CHAR(10),
       CHECK (Major IN ('IT', 'Cinema', 'History'));

    PRIMARY KEY (Sid));
```

Be careful with your database inputs ☺



CHECK Constraint and DOMAIN

```
CREATE DOMAIN M_Code AS CHAR(10)
CHECK (M_Code IN ('IT', 'Cinema', 'History'));

CREATE TABLE Student (
    Sid INTEGER,  Name CHAR(20),
    Age INTEGER,
    GPA REAL,
    Major M_Code,
    PRIMARY KEY (Sid));
```


CHECK: attribute- vs. tuple-based

- CHECK <u>prohibits</u> an operation on a table that would violate a constraint
- CHECK clause <u>restricts</u> acceptable attribute values according to some definition
  - **attribute-based**
- CHECK is also used as a **tuple-based** constraint:
  - Apply to each tuple individually
  - Checked whenever a tuple is inserted or modified
  - See next example…

Example

```
CREATE DOMAIN M_Code AS CHAR(10)
CHECK (M_Code IN ('IT' ,'Cinema', 'History'));
CREATE TABLE Student (
    Sid INTEGER,  Name CHAR(20),
    Age INTEGER,
    GPA REAL,
    Major M_Code,
    Minor ..., what constraints are needed for Minor?
    PRIMARY KEY (Sid));
```

IC1: Minor IN …
IC2: Minor ≠ Major

Example: attribute-based

```
CREATE DOMAIN M_Code AS CHAR(10)
CHECK (M_Code IN ('IT', 'Cinema', 'History'));
CREATE TABLE Student (
    Sid INTEGER,  Name CHAR(20),
    Age INTEGER,
    GPA REAL,
    Major M_Code,
    Minor M_Code,
    PRIMARY KEY (Sid));
```

IC1: attribute-based

Major = IT. Minor = IT.
but Major =/= Minor due to check.

Example: attribute- and tuple-based

```
CREATE DOMAIN M_Code AS CHAR(10)
CHECK (M_Code IN ('IT', 'Cinema', 'History'));
CREATE TABLE Student (
    Sid INTEGER,  Name CHAR(20),
    Age INTEGER,
    GPA REAL,
    Major M_Code,
    Minor M_Code,
    CHECK (Major != Minor);
    PRIMARY KEY (Sid));
```

IC1: attribute-based

IC2: tuple-based

Check (Major != Minor) tells people that you can't have a major and a minor that are on the same subject.

Naming Constraints

- A constraint may be given a name using the keyword **CONSTRAINT**
  - E.g., **CONSTRAINT** Major_Minor
- Advantages of naming a constraint:
  - Facilitates editing
  - Identifies a particular constraint
    1. For reporting
    2. For constraint management

Naming Constraints

```
CREATE DOMAIN M_Code AS CHAR(10)
CHECK (M_Code IN ('IT', 'Cinema', 'History'));
CREATE TABLE Student (
    Sid INTEGER, Name CHAR(20),
    Age INTEGER,
    GPA REAL,
    Major M_Code,
    Minor M_Code,
    CONSTRAINT Major_Minor
        CHECK (Major != Minor););
```


Constraint Management

```
ALTER TABLE Student DROP CONSTRAINT Major_Minor;
```

```
ALTER TABLE Student ADD CONSTRAINT Major_Minor
    CHECK (Major != Minor);
```

- To modify a constraint:
  - drop it first then add a new one

Assertions

- Similar to `CHECK` but they are **global** constraints

  `CREATE ASSERTION` <assertion_name>

  `CHECK` <condition>;

  - Global: schema-based
  - <condition> must be TRUE for each database state
- Examples:
  - # of IT stduents cannot exceed 1800
  - # of students in a prac cannot exceed lab capacity
  - …

Assertions

```
CREATE ASSERTION <assertion_name>
CHECK NOT EXISTS (vquery);
```

1. Specify a query <*vquery*> such that:
   vquery: selects any tuple that **violates** *<condition>*
2. Include vquery inside a **`NOT EXISTS`** clause

Assertions

```
CREATE ASSERTION <assertion_name>
CHECK NOT EXISTS (vquery);
```

Result of *vquery*	`NOT EXISTS` (*vquery*)	`CHECK`
Empty (no tuples violate the condition)	TRUE	Satisfied
Not Empty (some tuples violate the condition)	FALSE	Violated

Example Schema

SID	Name	Age	GPA	Major_Code
546007	Peter	18	3.8	0
546100	Bob	19	3.65	50
546500	Peter	20	3.7	1

Major_Code	Major_name
0	IT
1	History
…	
50	Cinema

Example

- Number of students in any major cannot exceed 1800

```
CREATE ASSERTION Major_Limit
CHECK  NOT EXISTS (
              SELECT Major_Code, COUNT(*)
              FROM Student
              GROUP BY Major_Code
              HAVING COUNT(*) > 1800 );
```


Example

- The Number of students cannot exceed 1800 in each of the IT or Cinema majors

```
CREATE ASSERTION Major_Limit
CHECK  NOT EXISTS (
     SELECT Major_Code, COUNT(*)
     FROM Student AS S , Major AS M
     WHERE S.major_code = M.major_code
     AND (M.major_name = "IT" OR M.major_name =
   "Cinema")
     GROUP BY Major_Code
     HAVING COUNT(*) > 1800 );
```

Triggers

- A trigger consists of 3 parts:
  1. Event(s),
  2. Condition, and
  3. Action

  - E.g., notify the Dean whenever the number of students in any major exceeds 1800

Triggers vs. Assertions

- Assertion
  - Condition must be true for each database state
  - DBMS rejects operations that violate such condition
- Trigger
  - DBMS takes a certain **action** when condition is true
  - Action could be: stored procedure, SQL statements, Rollback, etc.

Example

- Notify the Dean when the # of students in any major exceeds 1800

CREATE TRIGGER Major_Limit

Event(s)
Condition
Action

Example

- Notify the Dean when the # of students in any major exceeds 1800

CREATE TRIGGER Major_Limit

Event(s)

```
WHEN( EXISTS (
        SELECT Major_Code, COUNT(*)
        FROM Student
        GROUP BY Major_Code
        HAVING COUNT(*) > 1800 ))
```

Action

Example

- Notify the Dean when the # of students in any major exceeds 1800

CREATE TRIGGER Major_Limit

Event(s)

```
WHEN( EXISTS (
        SELECT Major_Code, COUNT(*)
        FROM Student
        GROUP BY Major_Code
        HAVING COUNT(*) > 1800 ))
```

```
CALL email_dean(Major_code);
```


Example

- Notify the Dean when the # of students in any major exceeds 1800

CREATE TRIGGER Major_Limit

```
AFTER INSERT OR UPDATE OF Major_Code
ON Student
```

```
WHEN( EXISTS (
        SELECT Major_Code, COUNT(*)
        FROM Student
        GROUP BY Major_Code
        HAVING COUNT(*) > 1800 ))
```

```
CALL email_dean(Major_code);
```

Events

- Events = <time> <operations>
  - operations = `INSERT`, `DELETE`, `UPDATE`
  - time = `BEFORE` or `AFTER`
    - `BEFORE:` the trigger is executed before the operation
    - `AFTER:` the trigger is executed after the operation
- A Trigger might allow operations that **violate** a constraint! (*compare to assertions…*)
  - Automatic constraint **repair**

Database Management System (DBMS)