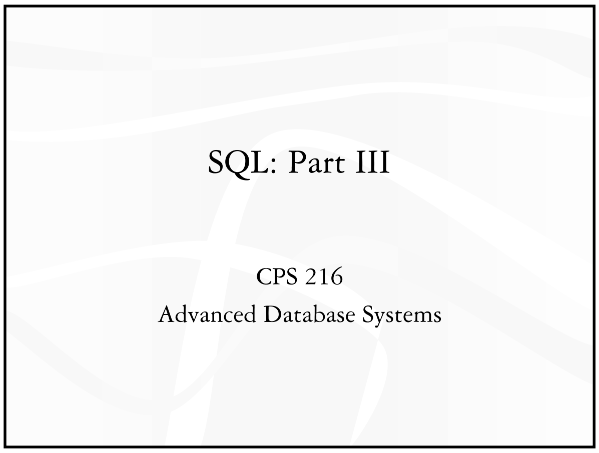
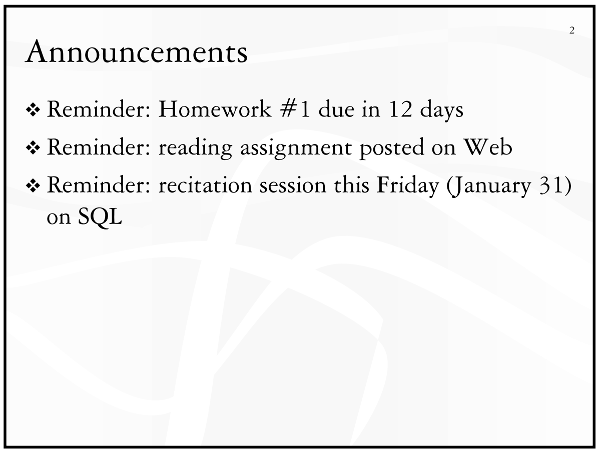


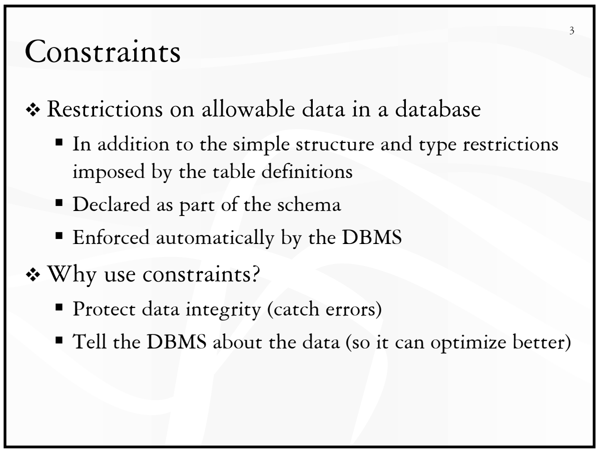
SQL: Part III

CPS 216
Advanced Database Systems



Announcements

- ❖ Reminder: Homework #1 due in 12 days
- ❖ Reminder: reading assignment posted on Web
- ❖ Reminder: recitation session this Friday (January 31) on SQL



Constraints

- ❖ Restrictions on allowable data in a database
  - In addition to the simple structure and type restrictions imposed by the table definitions
  - Declared as part of the schema
  - Enforced automatically by the DBMS
- ❖ Why use constraints?
  - Protect data integrity (catch errors)
  - Tell the DBMS about the data (so it can optimize better)

Types of SQL constraints

- NOT NULL
- Key
- Referential integrity (foreign key)
- General assertion
- Tuple- and attribute-based CHECK's

NOT NULL constraint examples

```
❖ CREATE TABLE Student
  (SID INTEGER NOT NULL,
   name VARCHAR(30) NOT NULL,
   email VARCHAR(30),
   age INTEGER,
   GPA FLOAT);
❖ CREATE TABLE Course
  (CID CHAR(10) NOT NULL,
   title VARCHAR(100) NOT NULL);
❖ CREATE TABLE Enroll
  (SID INTEGER NOT NULL,
   CID CHAR(10) NOT NULL);
```

Key declaration

- At most one PRIMARY KEY per table
  - Typically implies a primary index
  - Rows are stored inside the index, typically sorted by the primary key value
- Any number of UNIQUE keys per table
  - Typically implies a secondary index
  - Pointers to rows are stored inside the index

Key declaration examples

-
  ```
  CREATE TABLE Student
  (SID INTEGER NOT NULL PRIMARY KEY,
   name VARCHAR(30) NOT NULL,
   email VARCHAR(30) UNIQUE,
   age INTEGER,
   GPA FLOAT);
  ```

-
  ```
  CREATE TABLE Course
  (CID CHAR(10) NOT NULL PRIMARY KEY,
   title VARCHAR(100) NOT NULL);
  ```

-
  ```
  CREATE TABLE Enroll
  (SID INTEGER NOT NULL,
   CID CHAR(10) NOT NULL,
   PRIMARY KEY(SID, CID));
  ```

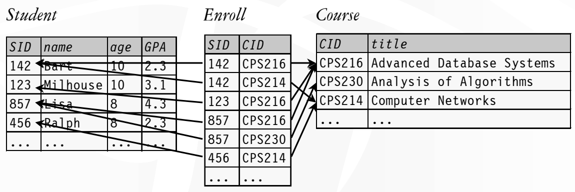
Works on Oracle but not DB2: DB2 requires `UNIQUE` key columns to be `NOT NULL`

This form is required for multi-attribute keys

Referential integrity example

- *Enroll.SID* references *Student.SID*
  - If an SID appears in *Enroll*, it must appear in *Student*
- *Enroll.CID* references *Course.CID*
  - If a CID appears in *Enroll*, it must appear in *Course*

☞ That is, no "dangling pointers"



Student

SID	*name*	*age*	*GPA*
142	Bart	10	2.3
123	Milhouse	10	3.1
857	Lisa	8	4.3
456	Ralph	8	2.3
...	...	...	...

Enroll

SID	*CID*
142	CPS216
142	CPS214
123	CPS216
857	CPS216
857	CPS230
456	CPS214
...	...

Course

CID	*title*
CPS216	Advanced Database Systems
CPS230	Analysis of Algorithms
CPS214	Computer Networks
...	...

Referential integrity in SQL

- Referenced column(s) must be `PRIMARY KEY`
- Referencing column(s) form a `FOREIGN KEY`
- Example

  -
    ```
    CREATE TABLE Enroll
     (SID INTEGER NOT NULL
       REFERENCES Student(SID),
      CID CHAR(10) NOT NULL,
      PRIMARY KEY(SID, CID),
      FOREIGN KEY CID REFERENCES Course(CID));
    ```

Enforcing referential integrity

Example: *Enroll.SID* references *Student.SID*

- Insert/update an *Enroll* row so it refers to a non-existent SID
  - Reject
-
  - Reject
  - Cascade: ripple changes to all referring rows
  - Set `NULL`: set all references to `NULL`
- Deferred constraint checking (e.g., only at the end of a transaction)
  - Good for
  - Required when

General assertion

- `CREATE ASSERTION` *assertion_name*
  `CHECK` *assertion_condition*`;`
- *assertion_condition* is checked for each modification that could potentially violate it
- Example: *Enroll.SID* references *Student.SID*

```
CREATE ASSERTION EnrollStudentRefIntegrity
CHECK (NOT EXISTS
       (SELECT * FROM Enroll
        WHERE SID NOT IN
        (SELECT SID FROM Student)));
```

☞ In SQL3, but not all (perhaps no) DBMS support it

Tuple- and attribute-based `CHECK`'s

- Associated with a single table
- Only checked when a tuple or an attribute is inserted or updated
- Example:

```
CREATE TABLE Enroll
(SID INTEGER NOT NULL
     CHECK (SID IN (SELECT SID FROM Student)),
 CID ...);
```

  - Is it a referential integrity constraint?

Summary of SQL features covered so far

- ❖ Query
  - `SELECT-FROM-WHERE` statements
  - Set and bag operations
  - Table expressions, subqueries
  - Ordering
  - Aggregation and grouping
- ❖ Modification
  - `INSERT/DELETE/UPDATE`
- ❖ Constraints

☞ Next: triggers, views, indexes

"Active" data

- ❖ Constraint enforcement: When a transaction violates a constraint, abort the transaction or try to "fix" the data
  - Example: enforcing referential integrity constraints
  - Generalize to arbitrary constraints?
- ❖ Data monitoring: When something happens to the data, automatically execute some action
  - Example: When price rises above $20 per share, sell
  - Example: When enrollment is at the limit and more students try to register, email the instructor

Triggers

- ❖ A trigger is an event-condition-action rule
  - When event occurs, test condition; if condition is satisfied, execute action
- ❖ Example:
  - Event: whenever there comes a new student…
  - Condition: with GPA higher than 3.0…
  - Action: then make him/her take CPS216!

Trigger example

```
CREATE TRIGGER CPS216AutoRecruit
  AFTER INSERT ON Student
  REFERENCING NEW ROW AS newStudent
  FOR EACH ROW
  WHEN (newStudent.GPA > 3.0)
  INSERT INTO Enroll
      VALUES(newStudent.SID, 'CPS216');
```

Trigger options

- Possible events include:
  - `INSERT ON` *table*
  - `DELETE ON` *table*
  - `UPDATE [OF` *column*`] ON` *table*
- Trigger can be activated:
  - `FOR EACH ROW` modified
  - `FOR EACH STATEMENT` that performs modification
- Action can be executed:
  - `AFTER` or `BEFORE` the triggering event

Transition variables

- `OLD ROW`: the modified row before the triggering event
- `NEW ROW`: the modified row after the triggering event
- `OLD TABLE`: a hypothetical read-only table containing all modified rows before the triggering event
- `NEW TABLE`: a hypothetical table containing all modified rows after the triggering event
- Not all of them make sense all the time, e.g.
  - `AFTER INSERT` statement-level triggers
    - Can use only `NEW TABLE`
  - `BEFORE DELETE` row-level triggers
  - etc.

Statement-level trigger example

```
CREATE TRIGGER CPS216AutoRecruit
  AFTER INSERT ON Student
  REFERENCING NEW TABLE AS newStudents
  FOR EACH STATEMENT
  INSERT INTO Enroll
  (SELECT SID, 'CPS216'
   FROM newStudents
   WHERE GPA > 3.0);
```

BEFORE trigger example

❖ Never give faculty more than 50% raise in one update

```
  CREATE TRIGGER NotTooGreedy
  BEFORE UPDATE OF salary ON Faculty
  REFERENCING OLD ROW AS o, NEW ROW AS n
  FOR EACH ROW
  WHEN (n.salary > 1.5 * o.salary)
  SET n.salary = 1.5 * o.salary;
```

☞ BEFORE triggers are often used to “condition” data

☞ Another option is to raise an error in the trigger body to abort the transaction that caused the trigger to fire

Statement- vs. row-level triggers

Why are both needed?

System issues

- ❖ Recursive firing of triggers
  - Action of one trigger causes another trigger to fire
  - Can get into an infinite loop
    - Some DBMS restrict trigger actions
    - Most DBMS set a maximum level of recursion (16 in DB2)
- ❖ Interaction with constraints (very tricky to get right!)
  - When do we check if a triggering event violates constraints?
    - After a `BEFORE` trigger (so the trigger can fix a potential violation)
    - Before an `AFTER` trigger
  - `AFTER` triggers also see the effects of, say, cascaded deletes caused by referential integrity constraint violations

  (Based on DB2; other DBMS may implement a different policy!)

Views

- ❖ A view is like a "virtual" table
  - Defined by a query, which describes how to compute the view contents on the fly
  - DBMS stores the view definition query instead of view contents
  - Can be used in queries just like a regular table

Creating and dropping views

- ❖ Example: CPS216 roster

  -
    ```
    CREATE VIEW CPS216Roster AS
    SELECT SID, name, age, GPA
    FROM Student
    WHERE SID IN (SELECT SID FROM Enroll
                  WHERE CID = 'CPS216');
    ```
    Called "base tables"
- ❖ To drop a view
  - `DROP VIEW` *view_name*`;`

Using views in queries

- Example: find the average GPA of CPS216 students
  - `SELECT AVG(GPA) FROM CPS216Roster;`
  - To process the query, replace the reference to the view by its definition

  -
    ```
    SELECT AVG(GPA)
    FROM (SELECT SID, name, age, GPA
          FROM Student
          WHERE SID IN (SELECT SID
                        FROM Enroll
                        WHERE CID = 'CPS216'));
    ```

Why use views?

- To hide data from users
- To hide complexity from users
- Logical data independence
  - If applications deal with views, we can change the underlying schema without affecting applications
  - Recall physical data independence: change the physical organization of data without affecting applications
- ☞ Real database applications use tons of views

Indexes

- An index is an auxiliary persistent data structure
  - Search tree (e.g., B^+-tree), lookup table (e.g., hash table), etc.
- ☞ More on indexes in following weeks!
- An index on $R.A$ can speed up accesses of the form
  - $R.A = value$
  - $R.A > value$ (sometimes; depending on the index type)
- An index on $\{ R.A_1, \ldots, R.A_n \}$ can speed up
  - $R.A_1 = value_1 \wedge \ldots \wedge R.A_n = value_n$
- ☞ Is an index on $\{ R.A, R.B \}$ equivalent to an index on $R.A$ plus another index on $R.B$?

Examples of using indexes

- `SELECT * FROM Student WHERE name = 'Bart'`
  - Without an index on Student.name: must scan the entire table if we store *Student* as a flat file of unordered rows
  - With index: go "directly" to rows with `name = 'Bart'`
- `SELECT * FROM Student, Enroll`
  `WHERE Student.SID = Enroll.SID;`
  - Without any index: for each *Student* row, scan the entire *Enroll* table for matching SID
    - Sorting could help
  - With an index on *Enroll.SID*: for each *Student* row, directly look up *Enroll* rows with matching SID

Creating and dropping indexes in SQL

- `CREATE INDEX` *index_name* `ON` *table_name*`(`*column_name*$_1$`, …,` *column_name*$_n$`);`
- `DROP INDEX` *index_name*`;`

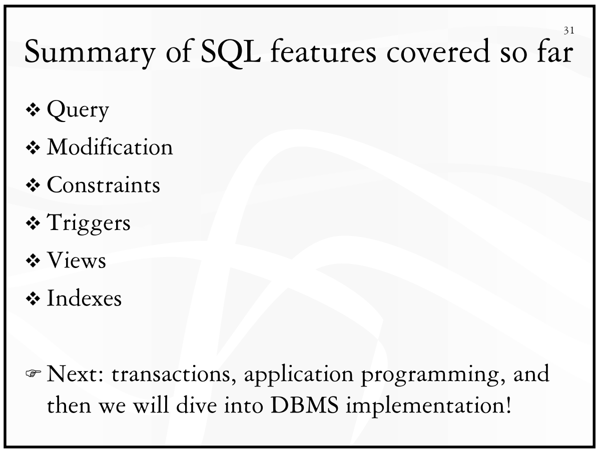
- Typically, the DBMS will automatically create indexes for `PRIMARY KEY` and `UNIQUE` constraint declarations

Choosing indexes to create

More indexes = better performance?

- Indexes take space
- Indexes need to be maintained when data is updated
- Indexes have one more level of indirection
  - Maybe not a problem for main memory, but can be really bad on disk

☞ Optimal index selection depends on both query and update workload and the size of tables
  - Automatic index selection is still an area of active research



Summary of SQL features covered so far

- ❖ Query
- ❖ Modification
- ❖ Constraints
- ❖ Triggers
- ❖ Views
- ❖ Indexes

☞ Next: transactions, application programming, and then we will dive into DBMS implementation!