CS422 Principles of Database Systems
SQL and Transactions

Chengyu Sun
California State University, Los Angeles

Structured Query Language (SQL)

- Data Definition Language (DDL)
  - `CREATE`, `DROP`, `ALTER`
- Data Manipulation Language (DML)
  - `SELECT`, `INSERT`, `DELETE`, `UPDATE`
- Data Control Language (DCL)
  - `GRANT`, `REVOKE`
  - `COMMIT`, `ROLLBACK`, `SAVEPOINT`

About SQL Dialects …

- Each DBMS has its own SQL dialect
- Basic syntax is mostly the same in all dialects
- Different in two major aspects
  - Advanced SQL features, e.g. various types of subqueries, recursive queries
  - Non-standardized features, e.g. most functions, procedural languages, FTS

… About SQL Dialects

- Generally speaking, anything can be done in any dialect, just in different ways
- Stick to standard when possible, use dialect when necessary

SQL Script

- A text file contains SQL *statements* and *comments*
  - Statements: `select`, `insert`, `create` …
  - Comments: lines started with **--**
- Usually uses the .sql suffix

Access PostgreSQL Server

- GUI client pgAdmin
- Command line client psql
- Web client phpPgAdmin

PostgreSQL Documentation

- http://www.postgresql.org/docs/

Examples: Create Tables

- Create the following tables:
  - 1. students(id, name, email)
  - 2. courses(id, name)
  - 3. sections(id, course_id, year)
  - 4. enrollment(id, section_id, student_id, grade)

Naming Conventions

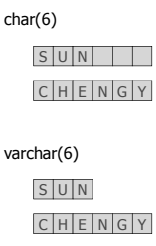
- Use plural form for table names
- Use singular form for column names
- Use underscore to concatenate multiple words, e.g. `course_id`
  - Do not use mixed cases in names (e.g. `CourseId`) because many DBMS treat names as case-insensitive

Data Type

- Determines the storage required for a field
- Common data types
  - String types
  - Numeric types
  - Date and time types
  - Other types

String Types

- `char(n)`
  - Fixed-length strings
  - Max length n
- `varchar(n)`
  - Variable-length strings
  - Max length n
- text
  - For articles, essays, …



Numeric Types

- Integer types
  - `integer`, `int`
  - Variations: `smallint`, `bigint`, `long`, …
  - Auto increment
    - `AUTO_INCREMENT`
    - `Serial`
- Boolean
  - `boolean`, `bool`
- Floating-point types
  - `real`
  - Variations: `float`, `double`, …
- Arbitrary precision number
  - `decimal(m,n)`
  - `numeric(m,n)`

Date and Time Types

- `date` – YYYY-MM-DD
- `time` – HH:MM:SS
- `datetime` – YYYY-MM-DD HH:MM:SS
- `timestamp` – YYYY-MM-DD HH:MM:SS

Data Integrity Constraints

- Not NULL
- Default
- Unique
- Primary key
  - Unique + Not NULL
  - Only one primary key per table
- Foreign key
- Check

Constraint Syntax

- Column constraint
- Table constraint
- Named constraint

Examples: Modify Tables

- Add grade point to grades
  - 5. Create `grades` table
  - 6. Drop the the `grade` column in the `enrollment` table
  - 7. Add a `grade_id` column to the `enrollment` table
  - 8. Add a foreign key constraint to the `grade_id` column

About ALTER TABLE

- Modify tables
  - Name
  - Schema
- Modify constraints
  - Add, remove
- Modify columns
  - Add, remove
  - Name
  - Type

Exactly what operations are supported depend on the DBMS.

Delete Table

```
drop table table_name;
```

Examples: Populate Tables

- Populate the tables we created so far
  - 9. Insert a record in each table
  - 10. Create all sections for 2009

SQL Literals

- Number: `10`, `30.2`
- String: `'CPU'`, `'John''s Kitchen'`
- Date: `'2007-06-01'`
- Time: `'12:00:00'`
- Boolean: `'t'`, `'f'`, `1`, `0`
- `NULL`

Sample Database: University

departments	id	name		
faculty	id	name	department_id	
students	id	name	graduation_date	major_id
grades	id	letter	value	
courses	id	title	department_id	
sections	id	course_id	instructor_id	year
enrollment	id	student_id	section_id	grade_id

Examples: Simple Selection

- 11. Find the sections taught by instructor #1 in 2004
- 12. List the names of the students whose names start with "A" in alphabetic order
- 13. List the id's of the courses that were offered before 2009

SQL Operators

- Arithmetic
  - +, -, *, /, %
- Comparison
  - <, >, <=, >=,=,<>
  - between
- Logical
  - and, or, not
- String
  - like
  - ||
- Other
  - is null
  - in
  - distinct
  - order by

LIKE

- Simple pattern matching
  - **%**: any zero or more characters
  - **_**: any single character

Example: Functions

- 14. Find the students who graduated in June
- 15. Find the students who graduated in the last six months

Functions in PostgreSQL

- http://www.postgresql.org/docs/current/interactive/functions.html

Common Functions in Databases

- Numerical functions
- String functions
- Date and time functions
- NULL related functions
- *Aggregation functions*

Most functions are DBMS specific.

Numerical functions

- Precision functions
- Power and square root
- Logarithmic functions
- Trigonometric functions
- Random number generator

String Functions

- String length
- Concatenation
- Locate/extract substring
- Trim white spaces
- Change cases
- Format numbers or dates

Date and Time Functions

- Extract date or time field
- Add or subtract a time interval
- Get current date or time
- Convert string to date or time

NULL Related Functions

- If NULL then *something*
- If *something* then NULL

Examples: Joins

- 16. Find the names of the departments that offer the course "Databases"
- 17. Find the names of the faculty who taught the course "Databases"
- 18. Find the courses that have never been offered

Join Syntax

- Equi-join syntax
- Inner join syntax

Cross Join

- A.K.A. Cartesian Product
- The results are *all possible combinations* of the rows from Table 1 with the rows from Table 2

table1

A	B
a_1	b_1
a_2	b_2

$\times$

table2

C	D
c_1	d_1
c_2	d_2
c_3	d_3

$=$

A	B	C	D
a_1	b_1	c_1	d_1
a_1	b_1	c_2	d_2
a_1	b_1	c_3	d_3
a_2	b_2	c_1	d_1
a_2	b_2	c_2	d_2
a_2	b_2	c_3	d_3

Problem of Cross Join

- Suppose Table 1 has 1000 rows, Table 2 has 1000 rows, how many rows does the result of Table 1 cross join Table 2 has??

Equi-Join

- Cross join with additional conditions

```
select … from T1, T2 where … … …
```

cross join

additional conditions

Inner Join

- a.k.a Join
- Combine two rows (one from each table) if they meet the join condition
- In other words, the results include the *matching rows* from the two tables

Inner Join Example

table1

A	B
1	10
2	12

table2

C	D
1	23
3	32
4	34

table1 *inner join* table2 on A=C

A	B	C	D
1	10	1	23

Outer Joins

- Include the results of an Inner Join and the unmatched rows from *one or both join tables*

Left Outer Join

- a.k.a. Left Join

table1

A	B
1	10
2	12

table2

C	D
1	23
3	32
4	34

table1 *left outer join* table2 on A=C

A	B	C	D
1	10	1	23
2	12	null	null

Right Outer Join

- a.k.a. Right Join

table1

A	B
1	10
2	12

table2

C	D
1	23
3	32
4	34

table1 *right outer join* table2 on A=C

A	B	C	D
1	10	1	23
null	null	3	32
null	null	4	34

Full Outer Join

- a.k.a. Full Join

table1

A	B
1	10
2	12

table2

C	D
1	23
3	32
4	34

table1 *full outer join* table2 on A=C

A	B	C	D
1	10	1	23
2	12	null	null
null	null	3	32
null	null	4	34

Examples: Subqueries

- 19. Find the student with the earliest graduation date
- 20. Find the departments that offered classes in 2007
- 21. Find the faculty who taught classes in 2007

Query Results

- Query results are either a table or a value*
  - E.g. `select * from products` or `select count(*) from products`
- *Query results can be used in places where a table/value can be used*

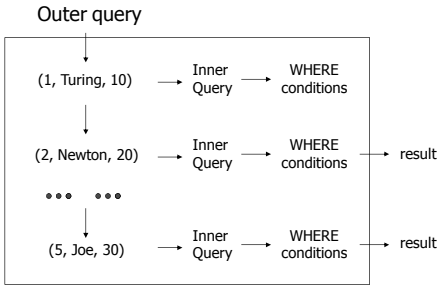
** A value can also be considered as a table with only one row and one column*

Correlated Subquery

- The inner query uses column(s) from the outer query
  - For example:

```
select * from faculty f where exists
        ( select * from sections
                where year = 2001
                and instructor_id = f.id );
```

How Correlated Subqueries Work



Examples: Set Operations

- 22. List all the names of the students and the faculty
- 23. List the names that appear in both the `students` table and the `faculty` table
- 24. List the names that appear in the `students` table but not in the `faculty` table

Set Operations

- Union
  - {1,2,3} ∪ {2,3,4} = {1,2,3,4}
- Intersect
  - {1,2,3} ∩ {2,3,4} = {2,3}
- Difference
  - {1,2,3} – {2,3,4} = {1}

Set Operations in Databases

- UNION
- INTERCEPT
- EXCEPT

About UNION

- Combine result tables of `SELECT` statements
- The result tables must have the same number of columns
- The corresponding columns must have the same (or at least "compatible") type
- Duplicates in union results
  - `UNION` – automatically remove duplicates
  - `UNION ALL` – keep duplicates

INTERSECT and EXCEPT

- Same syntax as UNION
- Some databases do not support INTERCEPT and EXCEPT, but the operations can be done in different ways
  - *How??*

Example: Aggregation Functions

- 25. Find the earliest graduation date
- 26. Find the number of courses offered by the Computer Science Department

Aggregation Functions

- Operate on multiple rows and return a single result
  - `sum`
  - `avg`
  - `count`
  - `max` and `min`

Be Careful with NULL

inventory

product_id	upc	quantity	price
1	1020301	20	100
2	1342193	null	200
3	null	100	null

max(price)?? min(price)?? avg(price)??

count(upc)?? count(*)??

sum(quantity) ??

Example: Aggregation Queries

- 27. List the number of students in each section
- 28. List the number courses offered by department
- 29. List the number of students graduated by year
- 30. Find the years in which there were more than 2 students graduated

Understand GROUP BY …

- Without aggregation/GROUP BY

select section_id, student_id from enrollment;

section_id	student_id
13	1
43	1
43	2
33	4
53	4
53	6

… Understand GROUP BY

- With aggregation/GROUP BY

select section_id, count(student_id) from enrollment
group by section_id;

Grouping attribute

Aggregation attribute

section_id	student_id	
13	1	count=1
43	1	count=2
43	2	
33	4	count=1
53	4	count=2
53	6	

How GROUP BY Works

1. Calculate the results *without* aggregation/GROUP BY
2. Divide the result rows into groups that *share the same value in the grouping attribute(s)*
3. Apply the aggregation function(s) to the aggregation attribute(s) *for each group*

The result attributes must be either a group attribute or a aggregation attribute.

HAVING vs. WHERE

1. Calculate the results *without* aggregation/GROUP BY ← *WHERE conditions*
2. Divide the result rows into groups that *share the same value in the grouping attribute(s)*
3. Apply the aggregation function(s) to the aggregation attribute(s) *for each group* ← *HAVING conditions*
4. *Final results*

Example: Top N Queries

- 31. Find the top 2 sections with the most students
- 32. Find the names of the top 3 faculty who taught the most number of sections

Top N Queries in PostgreSQL

- Using `ORDER BY`, `LIMIT` and `OFFSET`

```
select * from students
        order by graduation_date asc
        limit 3
        offset 2;
```

What if there is a tie??

Examples: Update and Delete

- 33. Change the name and department_id of faculty #5 to "John" and 10, respectively
- 34. Delete all the enrollment records of the Elocution class in 2001
- 35. Change all the B+ grades in the Calculus class in 2001 to A-

Update and Delete

```
delete from table [where condition(s)];

update table set field=value [, …]
        [where condition(s)];
```

Need for Transactions …

- Not all operations can be done with a single, atomic SQL statement, e.g. transferring money from one bank account to anther:

```
-- 1. Check the balance of account #1
select balance from accounts where id = 1;

-- 2. Withdraw $100 from account #1
update accounts set balance = balance – 100
        where id = 1;

-- 3. Deposit $100 to account #2
update accounts set balance = balance + 100
        where id = 2;
```

… Need for Transactions …

- Bad things could happen due to concurrent access and/or system failure

```
-- 1. Check the balance of account #1
select balance from accounts where id = 1;

-- 2. Withdraw $100 from account #1
update accounts set balance = balance – 100
        where id = 1;

-- 3. Deposit $100 to account #2
update accounts set balance = balance + 100
        where id = 2;
```

My wife withdraws all the money in account #1

… Need for Transactions …

- Bad things could happen due to concurrent access and/or system failure

```
-- 1. Check the balance of account #1
select balance from accounts where id = 1;

-- 2. Withdraw $100 from account #1
update accounts set balance = balance – 100
        where id = 1;

-- 3. Deposit $100 to account #2
update accounts set balance = balance + 100
        where id = 2;
```

My wife checks the balances of both accounts and notices that $100 is missing

… Need for Transactions

- Bad things could happen due to concurrent access and/or system failure

```
-- 1. Check the balance of account #1
select balance from accounts where id = 1;

-- 2. Withdraw $100 from account #1
update accounts set balance = balance – 100
        where id = 1;
```

System crash

```
-- 3. Deposit $100 to account #2
update accounts set balance = balance + 100
        where id = 2;
```

Transaction

- A transaction is a group of SQL statements treated by the DBMS as a *single unit of work*

Transaction Statements

- Start a transaction
  - `BEGIN, START TRANSACTION`
- End a transaction
  - `COMMIT`
  - `ROLLBACK`
- Nested transaction
  - `SAVEPOINT`
  - `ROLLBACK TO SAVEPOINT`

Example: Transactions

- Use a transaction to add two records to the `faculty` table
  - 36. Abort the transaction
  - 37. Commit the transaction

What happens if another transaction access the faculty table at the same time??

ACID Properties

- Database transactions are expected to have *ACID* properties
  - Atomic
  - Consistent
  - Isolated
  - Durable

Atomicity

- A transaction completes or fails as a whole, i.e. either all operations in the transaction are performed or none of them are.

Consistency

- Transaction should preserve database constraints.

Durability

- The changes made by *committed* transactions are guaranteed to be permanent, despite possible system failures.

Isolation

- Databases are often accessed by many users at the same time.
- Multiple transactions running concurrently should not interfere with each other, i.e. it should *appear* to the user that each transaction is executed in *isolation*.

SQL Isolation Levels

- Read uncommitted
- Read committed
- Repeatable read
- Serializable

Isolation Example

items

id	name	price
1	milk	2.99
2	beer	6.99

Transaction #1:

```
-- MIN
select name, price from items where price = (select min(price) from items);
-- MAX
select name, price from items where price = (select max(price) from items);
-- COUNT
select count(*) from items;
```

Read Uncommitted

- A transaction may read data written by another transaction that has not committed

Dirty Read

Transaction #2:

```
-- UPDATE
update items set price = 7.99 where name = 'beer';
-- ABORT
rollback;
```

Consider the interleaving of T1 and T2:

MIN, UPDATE, MAX, COUNT, ABORT

Read Committed

- A transaction reads only committed data.

Non-repeatable Read

Transaction #2:

```
-- UPDATE
update items set price = 7.99 where name = 'milk';
-- COMMIT
commit;
```

Consider the interleaving of T1 and T2:

MIN, UPDATE, COMMIT, MAX, COUNT

Repeatable Read

- A transaction reads only committed data, *and*, everything seen the first time will be seen the second time.

Phantom Read

Transaction #2:

```
-- INSERT
insert into items values (3, 'wine', 10.99);
-- COMMIT
commit;
```

Consider the interleaving of T1 and T2:

MIN, MAX, INSERT, COMMIT, COUNT

Serializable

- It appears to the user that the transactions are executed one at a time.

Isolation Levels in PostgreSQL

- Read committed (default)
- Serializable

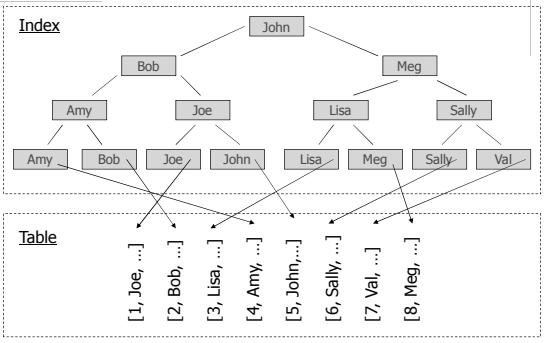
About Concurrent Transactions

- Concurrency is needed to maximize performance
- Concurrent transactions can lead to problems due to *aborted operations* and *interleaving operations*
- 4 isolation levels
- 3 problems

Example: Indexes and Views

- 38. Create an index on the name column of the students table
- 39. Create a view showing the id, course name, instructor's name, and the number of students in each section
- 40. Remove the view

Search with an Index



About Indexes

- Indexes make query execution more efficient
- Many DBMS automatically create indexes for primary key and unique columns
- There are many different types of indexes designed for different types of data and operations
  - E.g. B-tree, R-tree, Hash Index

About Views

- A view can be used as a table in SQL statements
- Most views cannot be updated
- The data in a view is dynamically computed, i.e. changes to *base tables* are automatically reflected in the view

Why Views

- Present the data in a user friendly way while keeping the base tables normalized
- Simplify SQL queries
- Security reasons
  - Views can be access controlled just like tables
  - Expose only part of the data to certain type of users

Summary

- Create and maintain database schema
- Query and update data
- Transactions and ACID
- Indexes and views