CSE 180, Database Systems I
Shel Finkelstein, UC Santa Cruz
Lecture 6

Chapter 4 : Intermediate SQL, Part 2

Database System Concepts, 7th Edition.

Modified by Shel Finkelstein for CSE 180

Important Notices

- **Reminder:** Midterm is on **Wednesday, February 12.**
  - No make-ups, no early Midterms, no late Midterms … and no devices.
  - You may bring a **single two-sided 8.5" x 11" sheet of paper** with as much info written (or printed) on it as you can fit and read unassisted.
    - No sharing of these sheets will be permitted.
  - "Practice Midterm" from Fall 2019 has been posted on Piazza under Resources→Exams.
    - We'll be covering more material before Midterm this quarter.
    - Solution has also been posted under Resources→Exams
      … but I strongly suggest that you take it yourself first, rather than just reading the solution.
  - Hope that all requests for DRC accommodation have been submitted.
    - DRC students will receive access to webcasts only if that specific accommodation has been requested by DRC.
  - Piazza announcement about Midterm describes required seating pattern for Midterm.

Important Notices

- Lab2 solution has been posted on Piazza as lab2.zip under Resources→Lab2.
- Lab3 has been posted on Piazza, with a lab3_create.sql file that creates tables for Lab3.
  - A lab3_data_loading.sql file that is **needed** to do Lab3 will be posted soon.
  - This Lecture, Lecture 5, includes material for Lab3.
  - A transaction (from Lecture 6) is also used in Lab3.
  - Lab3 is due on **Sunday, Feb 23, 11:59pm**. You have 3 weeks to complete Lab3 because the CSE 180 Midterm is on Wednesday, Feb 12.
- The third Gradiance Assignment, ”CSE 180 Winter 2020 #3", will be assigned after the Midterm.
- Access to all webcasts went away on Wednesday, January 29.
  - Attend Lectures, Lab Sections, Office Hours and LSS Tutoring.
- See Piazza notices about LSS Tutoring with Jeshwanth Bheemanpally.

Chapter 4: Intermediate SQL

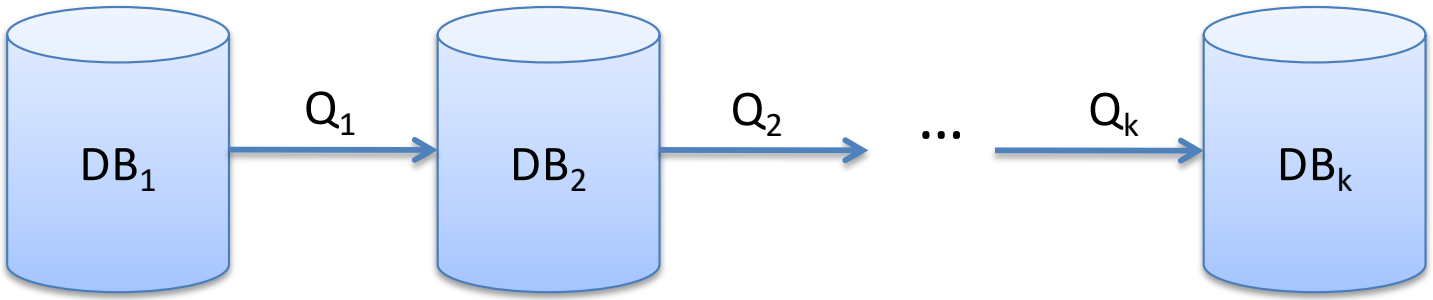
- Views
- Indexes
- Integrity Constraints
- Transactions
- Join Expressions
- SQL Data Types and Schemas
- Authorization

Transactions and ACID

- **A**tomicity
- **C**onsistency
- **I**solation
- **D**urability

One-Statement-At-a-Time Semantics

- So far, we have learnt how to query and modify the database.
- SQL statements posed to the database system were executed one at a time, retrieving data or changing the data in the database.



Transactions

- Applications such as web services, banking, airline reservations demand high throughput on operations performed on the database.
  - Manage hundreds of sales transactions every second.
  - Transactions often involve multiple SQL statements.
  - Database are transformed to new state based on (multiple statement) transactions, not just single SQL statements.

- It's possible for two operations to simultaneously affect the same bank account or flight, e.g. two spouses doing banking transactions, or an automatic deposit during a withdrawal, or two people reserving the same seat.
  - These "concurrent" operations must be handled carefully.

Transactions

- A **transaction** consists of a sequence of query and/or update statements and is a "unit" of work
- The SQL standard specifies that a transaction begins implicitly when an SQL statement is executed, although you can explicitly say **BEGIN TRANSACTION (**or **BEGIN WORK**).
- The transaction must end with one of the following statements:
  - **COMMIT TRANSACTION** (or **COMMIT WORK)**
    The updates performed by the transaction become permanent (Durable) in the database.
  - **ROLLBACK TRANSACTION** (or **ROLLBACK WORK**)
    All the updates performed by the SQL statements in the transaction are undone.
- If there is a failure of application or database before the transaction commit is processed by the database, then the transaction rolls back.

Transactions

- **Atomicity**
  - All statements in the transaction are either fully executed or rolled back as if none of them ever occurred. ("**All-or-nothing**")
- **Consistency**
  - Integrity constraints and other business rules for the database are always maintained.
- **Isolation**
  - Protection from actions taken by other transactions that are executing concurrently.
- **Durability**
  - After the database commits a transaction, the effects of that transaction don't go away, even if there are failures.
  - However, data can be modified by subsequent transactions.

What Could Go Wrong Without Transactions?

- Two different travelers see that a seat on a flight is available, and both reserve it.
- You put $2000 into your account and the transaction commits. But the database system fails, and when it re-starts, your $2000 isn't there.
- You're transferring money from your checking account to your savings account, and the database fails after the subtraction from the checking account, but before the addition to the savings account.
- Your bank account has $300. You're putting $50 into your bank account while your spouse is putting $100 into your account.
- A crook's transaction tentatively deposits $1000 into your account, but will rollback because the crook doesn't have the money. But another transaction sees the 1000 that was in the crook's account before the transaction rolls back, and gives him credit.
- Accounting totals for the bank don't balance correctly.

Read-Only Transactions

- If transactions were executed one-by-one, then they wouldn't interfere with each other.
- A DBMS can provide the illusion that the actions of Customer1 and Customer2 are executed *serially* (i.e., one at a time, with no overlap).
  - This ISOLATION LEVEL is called Serializability.
  - A DBMS that executed transactions one-by-one would have bad performance.
- **BEGIN TRANSACTION**
  [**READ WRITE** | **READ ONLY**]
  [**ISOLATION LEVEL** isolation_level];
  - Default is **READ WRITE**
  - Default for isolation_level in many DBMS is **READ COMMITTED**, not **SERIALIZABLE**
- You may also set this in using **SET TRANSACTION** instead of **BEGIN TRANSACTION**; this must be the first statement in the transaction.
- DBMS enforces **READ ONLY** if you specify it.
  - DBMS may take advantage of knowledge that a transaction is READ ONLY to improve its performance.
- We'll discuss isolation levels enforced by the DBMS next.

Dirty Reads (Read Uncommitted)

- A crook's transaction tentatively deposits $1000 into his account; this transaction rolls back. But an honest transaction sees the $1000 that was temporarily in the crook's account, before the crook's transaction rolls back, and the honest transaction gives the crook credit.
- *Dirty data* refers to data that is written by a transaction but has not yet been committed by the transaction.
  - A *dirty read* refers to the read of dirty data written by another transaction that has not committed.
- Bad things can happen when you build your finances on dirty reads, which are fantasies that haven't happened yet and may never happen.

Should Transactions Allow Dirty Reads?

- **Allow** Dirty Reads
  - More parallelism between transactions.
  - But may cause serious problems, as previous example shows.

- **Don't Allow** Dirty Reads
  - Less parallelism, more time is spent on waiting for other transactions to commit or rollback.
  - More overhead in the DBMS to prevent dirty reads.
  - Cleaner semantics.

Isolation levels

SET TRANSACTION READ WRITE
ISOLATION LEVEL READ UNCOMMITTED;

- First line: The transaction may write data (that's the default).
- Second line: The transaction can run with isolation level **READ UNCOMMITTED**, allowing Dirty Reads.

- Default isolation level depends on DBMS.
  - Most DBMS run using READ COMMITTED as their default isolation level.
  - Some run using isolation level SNAPSHOT ISOLATION.

Other Isolation Levels

- **SET TRANSACTION ISOLATION LEVEL READ COMMITTED;**
  - Only clean (committed) reads, no dirty reads.
  - But you might read data committed by *different* transactions.
    - You might not even get the same value even when you read same data a second time during a single transaction!
- **SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;**
  - Repeated queries of a tuple during a transaction will retrieve the same value, even if its value was changed by another transaction.
    - But *different* data reads might return values that were committed by *different* transactions at different times.
  - Also, a second scan of a range (e.g., salary>10000) may return "phantoms" not originally present in the scan..
    - Phantoms are tuples newly inserted while the transaction is running.
- **SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;**

Isolation Levels

Isolation Level	dirty reads	non-repeatable reads	phantoms
READ UNCOMMITTED	Y	Y	Y
READ COMMITTED	N	Y	Y
REPEATABLE READ	N	N	Y
SERIALIZABLE	N	N	N

Snapshot Isolation (SI)

- SI and Read Committed are most commonly used Isolation Levels.
  - Better performance (response time, throughput) than Serializability
- Transaction reads data as it existed when transaction began (repeatable).
  - As usual, transaction also sees its own updates
- Conflicts on Writes are avoided; equivalent of Serializable on Writes.
  - … but not on Read/Write interactions between transactions
- Example: Two transactions that are running under **Serializability** change both A and B. If both Commit, then one ran logically after the other.
- **SI Example**: A is supposed to be less than B. Initially A is 0 and B is 100.
  - T1 reads original A and B values, and changes A to 60.
  - T2 reads original A and B values, and changes B to 20.
  - Transactions T1 and T2 both maintained the consistency condition "A < B" … but what are the final values of A and B?
  - Could this happen with Serializability?

Join Operations

- **Join operations** take two relations and return as a result another relation.
- A join operation is a Cartesian product which requires that tuples in the two relations match (under some condition). It also specifies the attributes that are present in the result of the join
- The join operations are described as expressions in the **FROM** clause
- Three types of joins:
  - Natural join
  - Inner join
  - Outer join

Natural Join in SQL

- Natural join matches tuples with the same values for all common attributes, and retains only one copy of each common column.
- List the names of instructors along with the course ID of the courses that they taught
  - **SELECT** *name*, *course_id*
    FROM *students, takes*
    WHERE *student.ID* = *takes.ID*;
- Same query in SQL with "NATURAL JOIN" construct
  - **SELECT** *name*, *course_id*
    FROM *student* **NATURAL JOIN** *takes*;
- It's important to understand Natural Join, but it's not used very often in practice.
  - Results sometimes may surprise you, as we'll soon see.
  - And you can always write predicates explicitly in the **WHERE** clause.

Natural Join in SQL (Cont.)

- The **FROM** clause in can have multiple relations combined using natural join:

  SELECT $A_1, A_2, \ldots A_n$
  FROM r_1 **NATURAL JOIN** r_2 **NATURAL JOIN** ...
  NATURAL JOIN r_n
  WHERE P;

- And you may have other relations as well:

  SELECT $A_1, A_2, \ldots A_n$
  FROM r_1 **NATURAL JOIN** r_2 **NATURAL JOIN** ...
  NATURAL JOIN $r_n, s_1, s_2, \ldots$
  WHERE P;

Student Relation

ID	*name*	*dept_name*	*tot_cred*
00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
19991	Brandt	History	80
23121	Chavez	Finance	110
44553	Peltier	Physics	56
45678	Levy	Physics	46
54321	Williams	Comp. Sci.	54
55739	Sanchez	Music	38
70557	Snow	Physics	0
76543	Brown	Comp. Sci.	58
76653	Aoi	Elec. Eng.	60
98765	Bourikas	Elec. Eng.	98
98988	Tanaka	Biology	120

Takes Relation

ID	*course_id*	*sec_id*	*semester*	*year*	*grade*
00128	CS-101	1	Fall	2017	A
00128	CS-347	1	Fall	2017	A-
12345	CS-101	1	Fall	2017	C
12345	CS-190	2	Spring	2017	A
12345	CS-315	1	Spring	2018	A
12345	CS-347	1	Fall	2017	A
19991	HIS-351	1	Spring	2018	B
23121	FIN-201	1	Spring	2018	C+
44553	PHY-101	1	Fall	2017	B-
45678	CS-101	1	Fall	2017	F
45678	CS-101	1	Spring	2018	B+
45678	CS-319	1	Spring	2018	B
54321	CS-101	1	Fall	2017	A-
54321	CS-190	2	Spring	2017	B+
55739	MU-199	1	Spring	2018	A-
76543	CS-101	1	Fall	2017	A
76543	CS-319	2	Spring	2018	A
76653	EE-181	1	Spring	2017	C
98765	CS-101	1	Fall	2017	C-
98765	CS-315	1	Spring	2018	B
98988	BIO-101	1	Summer	2017	A
98988	BIO-301	1	Summer	2018	*null*

student **NATURAL JOIN** *takes*

ID	*name*	*dept_name*	*tot_cred*	*course_id*	*sec_id*	*semester*	*year*	*grade*
00128	Zhang	Comp. Sci.	102	CS-101	1	Fall	2017	A
00128	Zhang	Comp. Sci.	102	CS-347	1	Fall	2017	A-
12345	Shankar	Comp. Sci.	32	CS-101	1	Fall	2017	C
12345	Shankar	Comp. Sci.	32	CS-190	2	Spring	2017	A
12345	Shankar	Comp. Sci.	32	CS-315	1	Spring	2018	A
12345	Shankar	Comp. Sci.	32	CS-347	1	Fall	2017	A
19991	Brandt	History	80	HIS-351	1	Spring	2018	B
23121	Chavez	Finance	110	FIN-201	1	Spring	2018	C+
44553	Peltier	Physics	56	PHY-101	1	Fall	2017	B-
45678	Levy	Physics	46	CS-101	1	Fall	2017	F
45678	Levy	Physics	46	CS-101	1	Spring	2018	B+
45678	Levy	Physics	46	CS-319	1	Spring	2018	B
54321	Williams	Comp. Sci.	54	CS-101	1	Fall	2017	A-
54321	Williams	Comp. Sci.	54	CS-190	2	Spring	2017	B+
55739	Sanchez	Music	38	MU-199	1	Spring	2018	A-
76543	Brown	Comp. Sci.	58	CS-101	1	Fall	2017	A
76543	Brown	Comp. Sci.	58	CS-319	2	Spring	2018	A
76653	Aoi	Elec. Eng.	60	EE-181	1	Spring	2017	C
98765	Bourikas	Elec. Eng.	98	CS-101	1	Fall	2017	C-
98765	Bourikas	Elec. Eng.	98	CS-315	1	Spring	2018	B
98988	Tanaka	Biology	120	BIO-101	1	Summer	2017	A
98988	Tanaka	Biology	120	BIO-301	1	Summer	2018	*null*

Natural Join May Surprise You!

- Beware: Unrelated attributes that have the same name could be equated incorrectly.
  - Simple example: Several tables may have attributes named ID.
- More complicated example: List the names of students, together with the titles of courses that they have taken.
  - Incorrect version:

    SELECT *s.name*, c.*title*
    FROM *student s* **NATURAL JOIN** *takes t*
    NATURAL JOIN *course c*;

    - This query omits (*s.name, c.title*) pairs when the student takes a course in a department other than the student's own department …
      - because Natural Join matches *s.dept_name* and c.dept_name.
  - A correct version:

    SELECT *s.name*, c.*title*
    FROM *student s* **NATURAL JOIN** *takes t*, *course c*
    WHERE *t.course_id* = *c.course_id*;

Natural Join with USING Clause

- To avoid the danger of equating attributes erroneously, we can use the "**USING**" construct that allows us to specify exactly which columns should be equated.
- Query example

SELECT *name*, *title*
FROM (*student* **NATURAL JOIN** *takes*) **JOIN** *course*
USING (*course_id*)

Join Condition

- The **ON** condition allows a general predicate over the relations being joined, making queries more readable.
  - This predicate is written like a **WHERE** clause predicate, except for the use of the keyword **ON**.
- Query example

```
SELECT *
FROM  student JOIN takes
      ON student_ID  = takes_ID
```

  - The **ON** condition above specifies that a tuple from *student* matches a tuple from *takes* if their *ID* values are equal.
  - A **WHERE** clause can also appear in this query.
- This is equivalent to:

```
SELECT *
FROM  student , takes
WHERE  student_ID  = takes_ID
```

- "Ordinary" Join is sometimes called Inner Join, whether it's written with **ON** or **FROM** / **WHERE**.

Outer Join

- An extension of the join operation that avoids loss of information.
- Computes the join, and then adds to the result tuples from relation that has no matching tuples in the other relation.
  - "Dangling tuples"
- Uses **NULL** as value for attributes of the other relation.
- Three forms of Outer Join:
  - **LEFT OUTER JOIN**
  - **RIGHT OUTER JOIN**
  - **[FULL] OUTER JOIN**
    - **OUTER JOIN** means **FULL OUTER JOIN**

Outer Join Examples

- Relation *course*

course_id	*title*	*dept_name*	*credits*
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

- Relation *prereq*

course_id	*prereq_id*
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

- Observe that

  course information is missing for CS-437.

  prereq information is missing for CS-315.

Joined Relations – Examples

- *course* **INNER JOIN** *prereq*
  ON *course.course_id = prereq.course_id*

course_id	*title*	*dept_name*	*credits*	*prereq_id*	*course_id*
BIO-301	Genetics	Biology	4	BIO-101	BIO-301
CS-190	Game Design	Comp. Sci.	4	CS-101	CS-190

- Same as:

  course, prereq
  WHERE *course.course_id = prereq.course_id*

Left Outer Join

- *course* **LEFT OUTER JOIN** *prereq*
  ON *course.course_id = prereq.course_id*

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*

Right Outer Join

- *course* **RIGHT OUTER JOIN** *prereq*
  ON *course.course_id = prereq.course_id*

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-347	*null*	*null*	*null*	CS-101

Full Outer Join

- *course* **[FULL] OUTER JOIN** *prereq*
  ON *course.course_id = prereq.course_id*

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*
CS-347	*null*	*null*	*null*	CS-101

Join Types and Conditions

- **Join operations** take two relations as input, and return a relation as the result.
  - The join operations are described as expressions in the **FROM** clause
- **Join condition:** Defines which tuples in the two relations match, and what attributes are present in the result of the join.
  - **WHERE** <predicate>, **ON** <predicate>, **NATURAL, USING**
- **Join type:** Defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated.
  - **LEFT OUTER JOIN, RIGHT OUTER JOIN, [FULL] OUTER JOIN**
  - **JOIN, INNER JOIN**

Some Hard Questions about Joins
(Answered during Lecture)

- What is the result of A **JOIN** B when B is empty?
- If A has 40 tuples and B has 10 tuples, then how many tuples are there in A **JOIN** B?
- If A has 5 attributes, B has 4 attributes, and 2 of their attributes have the same name, then how many attributes are there in A **NATURAL JOIN** B?
- If there are no attributes with the same name in A and B, what is the result of A **OUTER JOIN** B?
- If A and B have exactly the same attributes, what is the result of A **OUTER JOIN** B?
- Is **NATURAL JOIN** a derived operation for Relational Algebra?
- Is **OUTER JOIN** a derived operation for Relational Algebra?

More Built-in Data Types in SQL
(In Section 4.5.2, but we've already discussed this.)

- **DATE:** Dates, containing a (4 digit) dept_name, month and day
  - Example: **DATE** '2005-7-27'
- **TIME:** Time of day, in hours, minutes and seconds.
  - Examples: **TIME** '09:00:30' **TIME** '09:00:30.75'
- **TIMESTAMP:** date plus time of day
  - Example: **TIMESTAMP** '2005-7-27 09:00:30.75'
- **INTERVAL:** period of time
  - Example: **INTERVAL** '1' **day**
  - Subtracting a DATE/TIME/TIMESTAMP value from another gives an INTERVAL value.
  - Interval values can be added to DATE/TIME/TIMESTAMP values, resulting in a DATE/TIME/TIMESTAMP value

Large-Object Types

- Large objects (photos, videos, CAD files, etc.) are stored as a *large object*:
  - **BLOB**: binary large object -- object is a large collection of uninterpreted binary data (whose interpretation is left to an application outside of the database system)
  - **CLOB**: character large object -- object is a large collection of character data
- When a query returns a large object, a pointer is returned rather than the large object itself.

User-Defined Types

- **CREATE TYPE** construct in SQL creates user-defined type

  CREATE TYPE *Dollars* **AS NUMERIC (12,2) FINAL;**

- Example:

  CREATE TABLE *department*
  (*dept_name* **VARCHAR** (20),
  building **VARCHAR** (15),
  budget Dollars);

Domains

- **CREATE DOMAIN** construct in SQL-92 creates user-defined domain types

  CREATE DOMAIN *person_name* **CHAR**(20) **NOT NULL;**

- Types and domains are similar. Domains can have constraints, such as **NOT NULL**, specified on them.
- Example:

  CREATE DOMAIN *degree_level* **VARCHAR**(10)
  CONSTRAINT *degree_level_test*
  CHECK (**VALUE in** ('Bachelors', 'Masters', 'Doctorate'));

Data Control Language (DCL)

- We may assign a user several forms of **authorizations** on parts of the database.
  - **Read** - allows reading, but not modification of data.
  - **Insert** - allows insertion of new data, but not modification of existing data.
  - **Update** - allows modification, but not deletion of data.
  - **Delete** - allows deletion of data.
- Each of these types of authorizations is called a **privilege**. We may authorize the user all, none, or a combination of these types of privileges on specified parts of a database, such as a relation or a view.

Authorization for Modifications

- Forms of authorization to modify the database schema
  - **Index** - allows creation and deletion of indices.
  - **Resources** - allows creation of new relations.
  - **Alteration** - allows addition or deletion of attributes in a relation.
  - **Drop** - allows deletion of relations.

Authorization Specification in SQL

- The **GRANT** statement is used to confer authorization

  GRANT <privilege list> **on** <relation or view > **to** <user list>

- <user list> is:
  - a user-id
  - **PUBLIC**, which allows all valid users the privilege GRANTed
  - A role (more on this later)
- Example:
  - **GRANT SELECT on** *department* **to** Amit, Satoshi
- Granting a privilege on a view does not imply GRANTing any privileges on the underlying relations.
- The person GRANT'ing of the privilege must already hold the privilege on the specified item (or be a database administrator, a DBA).

Privileges in SQL

- **SELECT**: allows read access to relation, or the ability to query using the view
  - Example: GRANT users U_1, U_2, and U_3 **SELECT** authorization on the *instructor* relation:

    GRANT SELECT on *instructor* **to** U_1, U_2, U_3
- **INSERT**: the ability to insert tuples
- **UPDATE**: the ability to update using the SQL update statement
- **DELETE**: the ability to delete tuples.
- **ALL PRIVILEGES**: used as a short form for all the allowable privileges

Revoking Authorization in SQL

- The **REVOKE** statement is used to REVOKE authorization.

  REVOKE <privilege list> **on** <relation or view> **FROM** <user list>

- Example:

  REVOKE SELECT on *student* **FROM** U_1, U_2, U_3

- <privilege-list> may be **all** to REVOKE all privileges the REVOKEe may hold.
- If <REVOKEe-list> includes **PUBLIC,** all users lose the privilege except those GRANTed it explicitly.
- If the same privilege was GRANTed twice to the same user by different GRANTees, the user may retain the privilege after the revocation.
- All privileges that depend on the privilege being REVOKEd are also REVOKEd.

Roles

- A **role** is a way to distinguish among various users as far as what these users can access/update in the database.
- To create a role we use:

  CREATE ROLE <name>
- Example:
  - **CREATE ROLE** instructor
- Once a role is created we can assign “users” to the role using:
  - **GRANT** <role> **TO** <users>

Roles Example

- **CREATE ROLE** instructor;
- **GRANT** *instructor* **TO** Amit**;**
- Privileges can be GRANTed to roles:
  - **GRANT SELECT ON** *takes* **TO** *instructor*;
- Roles can be GRANTed to users, as well as to other roles
  - **CREATE ROLE** *teaching_assistant*
  - **GRANT** *teaching_assistant* **TO** *instructor*;
    - *Instructor* inherits all privileges of *teaching_assistant*
- Chain of roles
  - **CREATE ROLE** *dean*;
  - **GRANT** *instructor* **TO** *dean*;
  - **GRANT** *dean* **TO** Satoshi;

Authorization on Views

- **CREATE VIEW** *geo_instructor* **as**
  (**SELECT** *
    FROM *instructor*
    FROM *dept_name* = 'Geology');
- **GRANT SELECT on** *geo_instructor* **TO** *geo_staff*
- Suppose that a *geo_staff* member issues
  - **SELECT** *
    FROM *geo_instructor*;
- What if:
  - *geo_staff* does not have permissions on *instructor?*
  - creator of view did not have some permissions on *instructor?*