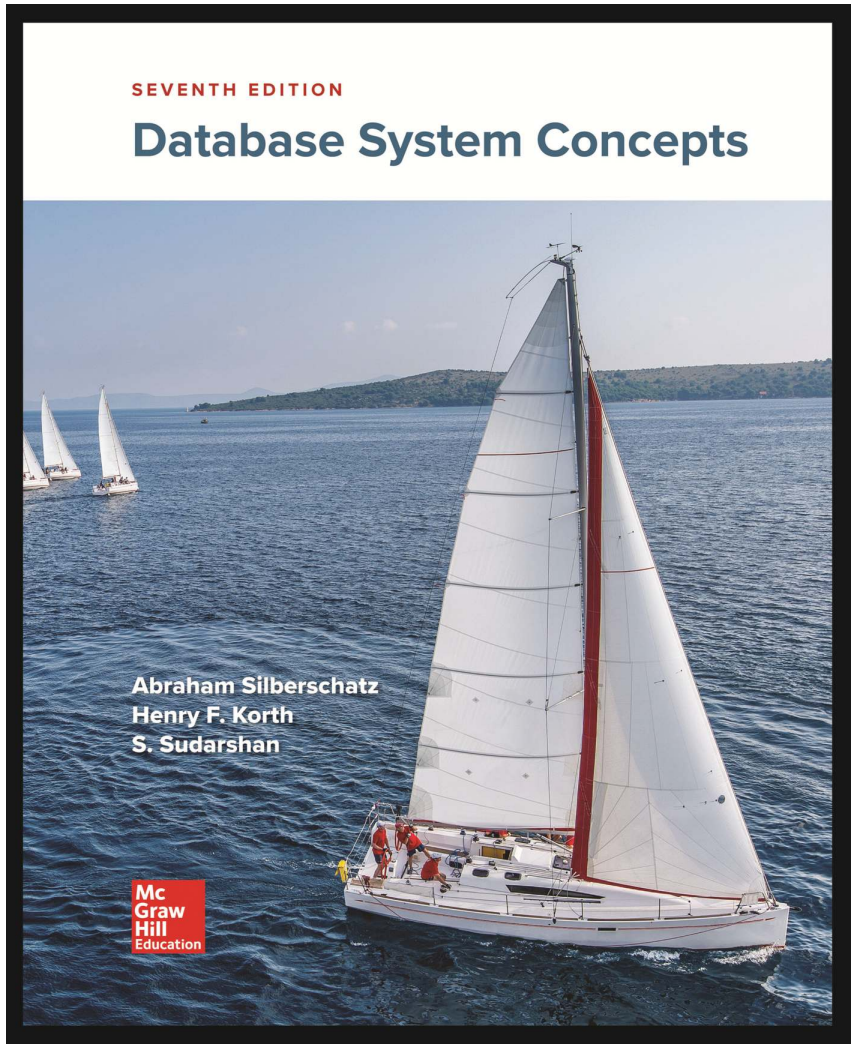
Database System Concepts, 7^{th} Edition
Chapter 4: Intermediate SQL

Silberschatz, Korth and Sudarshan

March 9, 2025

Database System Concepts



Content has been extracted from *Database System Concepts*, Seventh Edition, by Silberschatz, Korth and Sudarshan. Mc Graw Hill Education. 2019.
Visit `https://db-book.com/`.

Plan

- Join Expressions
- Views
- Transactions
- Integrity Constraints
- SQL Data Types and Schemas
- Index Definition in SQL
- Authorization

Joined Relations

- 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 typically used as subquery 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.
- For all students in the university who have taken some course, find their names and the course ID of all courses they took.

```
1        SELECT
2            name, course_id
3        FROM
4            students, takes
5        WHERE
6            student.ID = takes.ID;
```

Natural Join in SQL

- Same query in SQL with "natural join" construct.

```
1  SELECT
2      name, course_id
3  FROM
4      student NATURAL JOIN takes;
```

Natural Join in SQL (Cont.)

- The `FROM` clause 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 $\ldots$
    NATURAL JOIN r_n
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*

Dangerous in Natural Join

Beware of unrelated attributes with same name which get equated incorrectly

Example:

List the names of students along with the titles of courses that they have taken.

```
1    SELECT
2        name, title
3    FROM
4        student
5    natural join
6        takes
7    natural join
8        course;
```

Dangerous in Natural Join

Beware of unrelated attributes with same name which get equated incorrectly

Example:

List the names of students along with the titles of courses that they have taken.

```
1    SELECT
2        name, title
3    FROM
4        student
5    natural join
6        takes
7    natural join
8        course;
```

Incorrect! double check *dept_name* attribute

Dangerous in Natural Join

Example:

List the names of students along with the titles of courses that they have taken.

```
1    SELECT
2        name, title
3    FROM
4        student
5    NATURAL JOIN
6        takes
7    NATURAL JOIN
8        course;
```

- This query omits all (student name, course title) pairs where the student takes a course in a department other than the student's own department.

Dangerous in Natural Join

Example:

List the names of students along with the titles of courses that they have taken.

```
1    SELECT
2        name, title
3    FROM
4        student
5    NATURAL JOIN
6        takes, course
7    WHERE
8        takes.course_id = course.course_id;
```

- The correct version (above), correctly outputs such pairs.

Outer Join

- An extension of the join operation that avoids loss of information.
- Computes the `JOIN` and then adds tuples form one relation that does not match tuples in the other relation to the result of the join.
- Uses `null` values.
- Three forms of outer join:
  - left outer join
  - right outer join
  - 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-347
  - prereq information is missing for CS-315

Left Outer Join

- course `NATURAL LEFT OUTER JOIN` prereq

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`

- In relational algebra: course $\ltimes$ prereq

Right Outer Join

- course `NATURAL RIGHT OUTER JOIN` prereq

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

- In relational algebra: course ⟖ prereq

Full Outer Join

- course `NATURAL FULL OUTER JOIN` prereq

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

- In relational algebra: course $\ltimes$ prereq

Joined Types and Conditions

- **Join operations** – take two relations and return as a result another relation.
- These additional operations are typically used as subquery 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.
- **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.

Join types
inner join
left outer join
right outer join
full outer join

Join conditions
natural
on <predicate>
using $(A_1, A_2, \ldots, A_n)$

Joined Relations – Examples

- course `NATURAL RIGHT OUTER JOIN` prereq

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

- course `FULL OUTER JOIN` prereq `USING` (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

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

- What is the difference between the above, and a natural join?

- course `LEFT OUTER 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
CS-315	Robotics	Comp. Sci.	3	`null`	`null`

Joined Relations – Examples

- course `NATURAL RIGHT OUTER JOIN` prereq

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

- course `FULL OUTER JOIN` prereq `USING` (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

Plan

- Join Expressions
- Views
- Transactions
- Integrity Constraints
- SQL Data Types and Schemas
- Index Definition in SQL
- Authorization

Views

- In some cases, it is not desirable for all users to see the entire logical model (that is, all the actual relations stored in the database.)
- Consider a person who needs to know an instructors name and department, but not the salary. This person should see a relation described, in SQL, by

```
1    SELECT
2        ID, name, dept_name
3    FROM
4        instructor;
```

- A **view** provides a mechanism to hide certain data from the view of certain users.
- Any relation that is not of the conceptual model but is made visible to a user as a "virtual relation" is called a **view**.

View Definition

- A view is defined using the create view statement which has the form:

  CREATE VIEW v **AS** $< query\_expression >$;

  where $< query\_expression >$ is any legal `SQL` expression. The view name is represented by v.
- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.
- View definition is not the same as creating a new relation by evaluating the query expression
  - Rather, a view definition causes the saving of an expression; the expression is substituted into queries using the view.

View Definition and Use

- A view of instructors without their salary.

```
1    CREATE VIEW faculty AS
2    SELECT ID, name, dept_name
3    FROM instructor
```

View Definition and Use

- A view of instructors without their salary.

```
1  CREATE VIEW faculty AS
2  SELECT ID, name, dept_name
3  FROM instructor
```

- Find all instructors in the Biology department

```
1  SELECT name
2  FROM faculty
3  WHERE dept_name = 'Biology'
```

View Definition and Use

- A view of instructors without their salary.

```
CREATE VIEW faculty AS
SELECT ID, name, dept_name
FROM instructor
```

- Find all instructors in the Biology department

```
SELECT name
FROM faculty
WHERE dept_name = 'Biology'
```

- Create a view of department salary totals

```
CREATE VIEW departments_total_salary(dept_name, total_salary) AS
SELECT dept_name, SUM(salary)
FROM instructor
GROUP BY dept_name;
```

Views Defined Using Other Views

- One view may be used in the expression defining another view.
- A view relation v_1 is said to depend directly on a view relation v_2 if v_2 is used in the expression defining v_1.
- A view relation v_1 is said to depend on view relation v_2 if either v_1 depends directly to v_2 or there is a path of dependencies from v_1 to v_2.
- A view relation v is said to be recursive if it depends on itself.

Views Defined Using Other Views

```
1   CREATE VIEW
2       physics_fall_2017 AS
3   SELECT
4       course.course_id, sec_id, building, room_number
5   FROM
6       course, section
7   WHERE
8       course.course_id = section.course_id AND
9       course.dept_name = 'Physics' AND
10      section.semester = 'Fall' AND
11      section.year = '2017';
```

Views Defined Using Other Views

```
1    CREATE VIEW
2        physics_fall_2017_watson AS
3    SELECT
4        course_id, room_number
5    FROM
6        physics_fall_2017
7    WHERE
8        building= 'Watson';
```

View Expansion

- Expand the view:

```
1  CREATE VIEW physics_fall_2017_watson AS
2      SELECT course_id, room_number
3      FROM physics_fall_2017
4      WHERE building= 'Watson';
```

- To:

```
1  CREATE VIEW physics_fall_2017_watson AS
2      SELECT course_id, room_number
3      FROM ( SELECT course.course_id, sect_id, building, room_number
4               FROM course, section
5               WHERE course.course_id = section.course_id
6               AND course.dept_name = 'Physics'
7               AND section.semester = 'Fall'
8               AND section.year = '2017' )
9      WHERE building= 'Watson';
```

View Expansion (Cont.)

- A way to define the meaning of views defined in terms of other views.
- Let view v_1 be defined by an expression e_1 that may itself contain uses of view relations.
- View expansion of an expression repeats the following replacement step:

```
REPEAT
        Find any view relation v_i in e_i
        Replace the view relation v_i
            by the expression defining v_i
UNTIL no more view relations are present in e_i
```

- As long as the view definitions are not recursive, this loop will terminate.

Materialized Views

- Certain database systems allow view relations to be physically stored.
  - Physical copy created when the view is defined.
  - Such views are called **Materialized view**.
- If relations used in the query are updated, the materialized view result becomes out of date.
  - Need to **maintain** the view, by updating the view whenever the underlying relations are updated.

Update of a View

- Add a new tuple to faculty view which we defined earlier.

```
INSERT INTO faculty
VALUES ('30765', 'Green', 'Music');
```

- This insertion must be represented by the insertion into the instructor relation.
  - Must have a value for salary.
- Two approaches:
  - Reject the insert.
  - Insert the tuple:
    (‘30765’, ‘Green’, ‘Music’, null)
    into the *instructor* relation.

Some Updates Cannot be Translated Uniquely

```
CREATE VIEW instructor_info AS
    SELECT
        ID, name, building
    FROM
        instructor, department
    WHERE
        instructor.dept_name= department.dept_name;
```

```
INSERT INTO instructor_info
VALUES ('69987', 'White', 'Taylor');
```

- Issues:
  - Which department, if multiple departments in Taylor?
  - What if no department is in Taylor?

And Some Not at All

```
CREATE VIEW history_instructors AS
    SELECT *
    FROM instructor
    WHERE dept_name= 'History';
```

- What happens if we insert:
  (‘25566’, ‘Brown’, ‘Biology’, 100000)
  into *history_instructors*?

View Updates in SQL

- Most `SQL` implementations allow updates only on simple views:
  - The `FROM` clause has only one database relation.
  - The `SELECT` clause contains only attribute names of the relation, and does not have any expressions, aggregates, or distinct specification.
  - Any attribute not listed in the `SELECT` clause can be set to `null`.
  - The query does not have a `GROUP BY` or `HAVING` clause.

Plan

Join Expressions

Views

Transactions

Integrity Constraints

SQL Data Types and Schemas

Index Definition in SQL

Authorization

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.
- The transaction must end with one of the following statements:
  - Commit work. The updates performed by the transaction become permanent in the database.
  - Rollback work. All the updates performed by the `SQL` statements in the transaction are undone.
- Atomic transaction:
  - Either fully executed or rolled back as if it never occurred.
- Isolation from concurrent transactions.

Plan

Join Expressions

Views

Transactions

Integrity Constraints

SQL Data Types and Schemas

Index Definition in SQL

Authorization

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.

Constraints on a Single Relation

- not null.
- primary key.
- unique.
- check (P), where P is a predicate.

Not Null Constraints

- `not null`
  - Declare name and budget to be not null:

    ```
    name varchar(20) not null
    budget numeric(12,2) not null
    ```

Unique Constraints

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

The check clause

- The `check(P)` clause specifies a predicate `P` that must be satisfied by every tuple in a relation.
- 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`.

Referential Integrity (Cont.)

- Foreign keys can be specified as part of the `CREATE TABLE` statement:

  ```
  FOREIGN KEY (dept_name) REFERENCES department
  ```

- By default, a foreign key references the primary key attributes of the referenced table.
- `SQL` allows a list of attributes of the referenced relation to be specified explicitly:

  ```
  FOREIGN KEY (dept_name)
  REFERENCES department (dept_name)
  ```

Cascading Actions in Referential Integrity

- When a referential-integrity constraint is violated, the normal procedure is to reject the action that caused the violation.
- An alternative, in case of delete or update is to cascade:

```
1  CREATE TABLE course (
2      ...
3      dept_name varchar(20),
4      FOREIGN KEY (dept_name) REFERENCES department
5          ON DELETE CASCADE
6          ON UPDATE CASCADE,
7          ...
8  );
```

- Instead of `CASCADE` we can use :
  - `SET NULL`
  - `SET DEFAULT`

Integrity Constraint Violation During Transactions

- Consider:

```
1  CREATE TABLE person (
2      ID char(10),
3      name char(40),
4      mother char(10),
5      father char(10),
6      PRIMARY KEY ID,
7      FOREIGN KEY father REFERENCES person,
8      FOREIGN KEY mother REFERENCES person
9  );
```

- How to insert a tuple without causing constraint violation?

Integrity Constraint Violation During Transactions

- Consider:

```
1  CREATE TABLE person (
2      ID char(10),
3      name char(40),
4      mother char(10),
5      father char(10),
6      PRIMARY KEY ID,
7      FOREIGN KEY father REFERENCES person,
8      FOREIGN KEY mother REFERENCES person
9  );
```

- 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 Conditions

- The predicate in the `CHECK` clause can be an arbitrary predicate that can include a subquery:

```
CHECK (
    time_slot_id IN (
        SELECT time_slot_id FROM time_slot
    )
)
```

  The `CHECK` condition states that the `time_slot_id` in each tuple in the *section* relation is actually the identifier of a time slot in the *time_slot* relation.

- The condition has to be checked not only when a tuple is inserted or modified in *section*, but also when the relation *time_slot* changes.

Assertions

- An assertion is a predicate expressing a condition that we wish the database always to satisfy.
- The following constraints, can be expressed using assertions:
  - For each tuple in the *student* relation, the value of the attribute `tot_cred` must equal the sum of credits of courses that the student has completed successfully.
  - An instructor cannot teach in two different classrooms in a semester in the same time slot.
- An assertion in SQL takes the form:

```
CREATE ASSERTION <assertion-name> CHECK (<predicate>);
```

Plan

Built-in Data Types in SQL

- `date`: Dates, containing a (4 digit) year, month and date:
  - Example: `date '2005-7-27'`
- `time`: Time of day, in hours, minutes and seconds.
  - Example: `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.

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:

```
1  CREATE TABLE department (
2      dept_name varchar (20),
3      building varchar (15),
4      budget Dollars
5  );
```

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:

```
1  CREATE DOMAIN degree_level varchar(10)
2  CONSTRAINT degree_level_test
3      CHECK(
4          degree_value IN
5          ('Bachelors', 'Masters', 'Doctorate')
6      );
```

Plan

Index Creation

- Many queries reference only a small proportion of the records in a table.
- It is inefficient for the system to read every record to find a record with particular value
- An index on an attribute of a relation is a data structure that allows the database system to find those tuples in the relation that have a specified value for that attribute efficiently, without scanning through all the tuples of the relation.
- We create an index with the `CREATE INDEX` command:

  ```
  CREATE INDEX <name> ON <relation-name> (attribute);
  ```

Index Creation Example

```
1  CREATE TABLE student(
2      ID varchar (5),
3      name varchar (20) NOT NULL,
4      dept_name varchar (20),
5      tot_cred numeric (3,0) DEFAULT 0,
6      PRIMARY KEY (ID)
7  );
```

```
CREATE INDEX studentID_index ON student(ID);
```

- The query:

```
1  SELECT *
2  FROM student
3  WHERE ID = '12345';
```

can be executed by using the index to find the required record, without looking at all records of *student*.

Plan

- Join Expressions
- Views
- Transactions
- Integrity Constraints
- SQL Data Types and Schemas
- Index Definition in SQL
- Authorization

Authorization

- 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 (Cont.)

- 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 grantor of the privilege must already hold the privilege on the specified item (or be the database administrator).

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 <revokee list>
```

- Example:

  REVOKE SELECT ON `instructor` TO 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;
```

Roles Example

```
CREATE ROLE instructor;
GRANT instructor TO Amit;
```

- Privileges can be granted to roles:

```
GRANT SELECT ON takes TO instructor;
```

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`.

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

```
1  CREATE VIEW geo_instructor AS (
2      SELECT *
3      FROM instructor
4      WHERE dept_name = 'Geology'
5  );
6  GRANT SELECT ON geo_instructor TO geo_staff;
```

Authorization on Views

```
1  CREATE VIEW geo_instructor AS (
2      SELECT *
3      FROM instructor
4      WHERE dept_name = 'Geology'
5  );
6  GRANT SELECT ON geo_instructor TO geo_staff;
```

- Suppose that a `geo_staff` member issues:

```
SELECT *
FROM geo_instructor;
```

Authorization on Views

```
1  CREATE VIEW geo_instructor AS (
2      SELECT *
3      FROM instructor
4      WHERE dept_name = 'Geology'
5  );
6  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*?

Other Authorization Features

- `REFERENCES` privilege to create foreign key:

```
GRANT REFERENCE (dept_name) ON department TO Mariano;
```

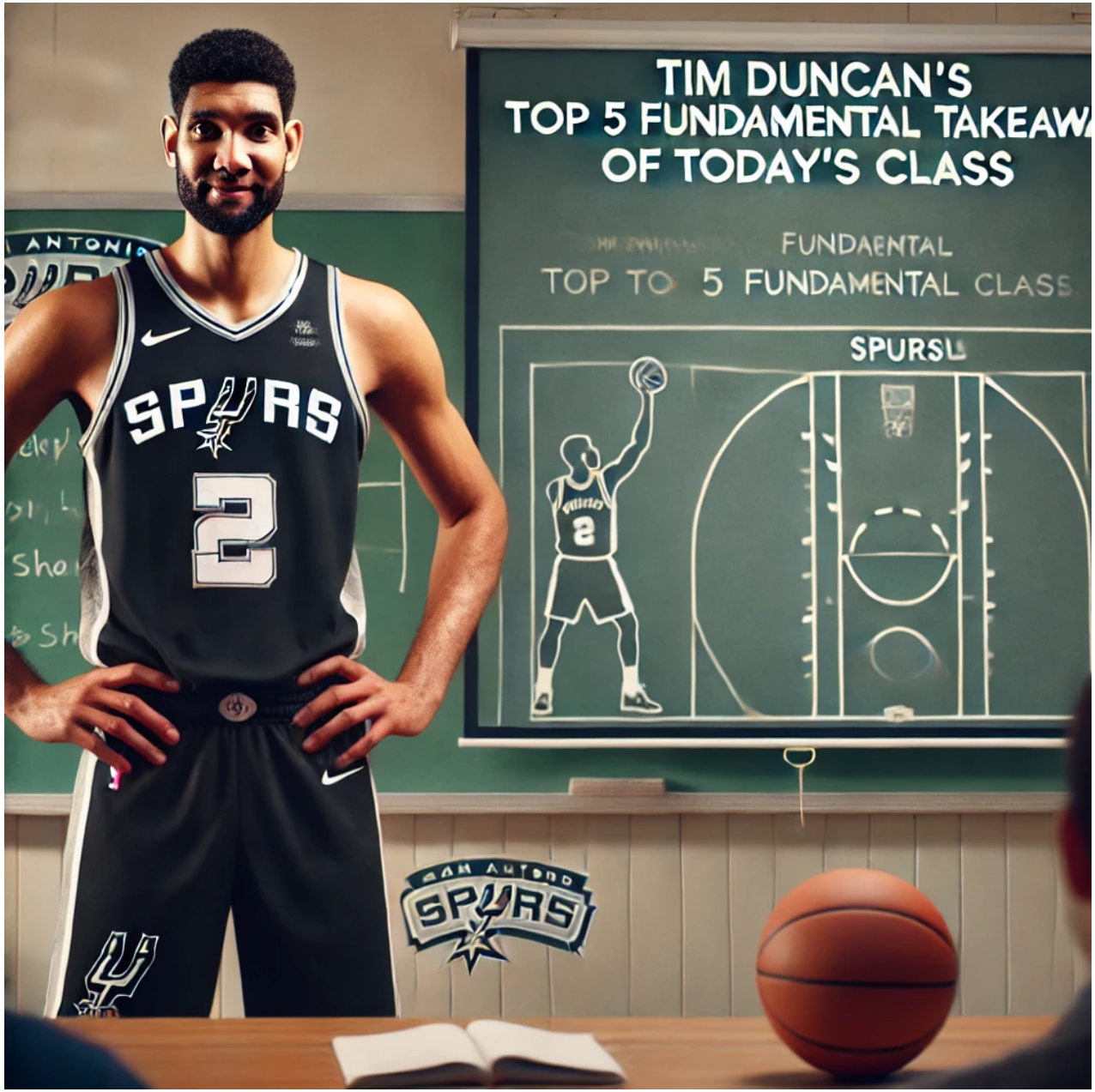
  - Why is this required?
- Transfer of privileges:

```
GRANT SELECT ON department TO Amit WITH GRANT OPTION;
REVOKE SELECT ON department FROM Amit, Satoshi CASCADE;
REVOKE SELECT ON department FROM Amit, Satoshi RESTRICT;
```

  - And more!

End of Chapter 4.

TDT5FTOTTC


TIM DUNCAN'S
TOP 5 FUNDAMENTAL TAKEAW
OF TODAY'S CLASS
FUNDAENTAL
TOP TO 5 FUNDAMENTAL CLASS
SPURS
SPURS
SPURS

Top 5 Fundamental Takeaways

5 **Join Operations in `SQL`** – `SQL` joins (natural, inner, and outer) combine tables based on matching attributes, with outer joins preserving unmatched rows using `null` values.

Top 5 Fundamental Takeaways

5 **Join Operations in `SQL`** – `SQL` joins (natural, inner, and outer) combine tables based on matching attributes, with outer joins preserving unmatched rows using `null` values.

4 **Views and Their Uses** – Views are virtual tables that simplify queries and restrict data access, but they are not physically stored unless materialized.

Top 5 Fundamental Takeaways

5 **Join Operations in `SQL`** – `SQL` joins (natural, inner, and outer) combine tables based on matching attributes, with outer joins preserving unmatched rows using `null` values.

4 **Views and Their Uses** – Views are virtual tables that simplify queries and restrict data access, but they are not physically stored unless materialized.

3 **Transactions and Atomicity** – Transactions ensure database consistency by executing multiple `SQL` operations as a unit, requiring either a commit to save changes or a rollback to undo them.

Top 5 Fundamental Takeaways

5 **Join Operations in `SQL`** – `SQL` joins (natural, inner, and outer) combine tables based on matching attributes, with outer joins preserving unmatched rows using `null` values.

4 **Views and Their Uses** – Views are virtual tables that simplify queries and restrict data access, but they are not physically stored unless materialized.

3 **Transactions and Atomicity** – Transactions ensure database consistency by executing multiple `SQL` operations as a unit, requiring either a commit to save changes or a rollback to undo them.

2 **Integrity Constraints and Referential Integrity** – Constraints like `NOT NULL`, `PRIMARY KEY`, and foreign keys enforce data integrity, ensuring valid relationships between tables.

Top 5 Fundamental Takeaways

5 **Join Operations in `SQL`** – `SQL` joins (natural, inner, and outer) combine tables based on matching attributes, with outer joins preserving unmatched rows using `null` values.

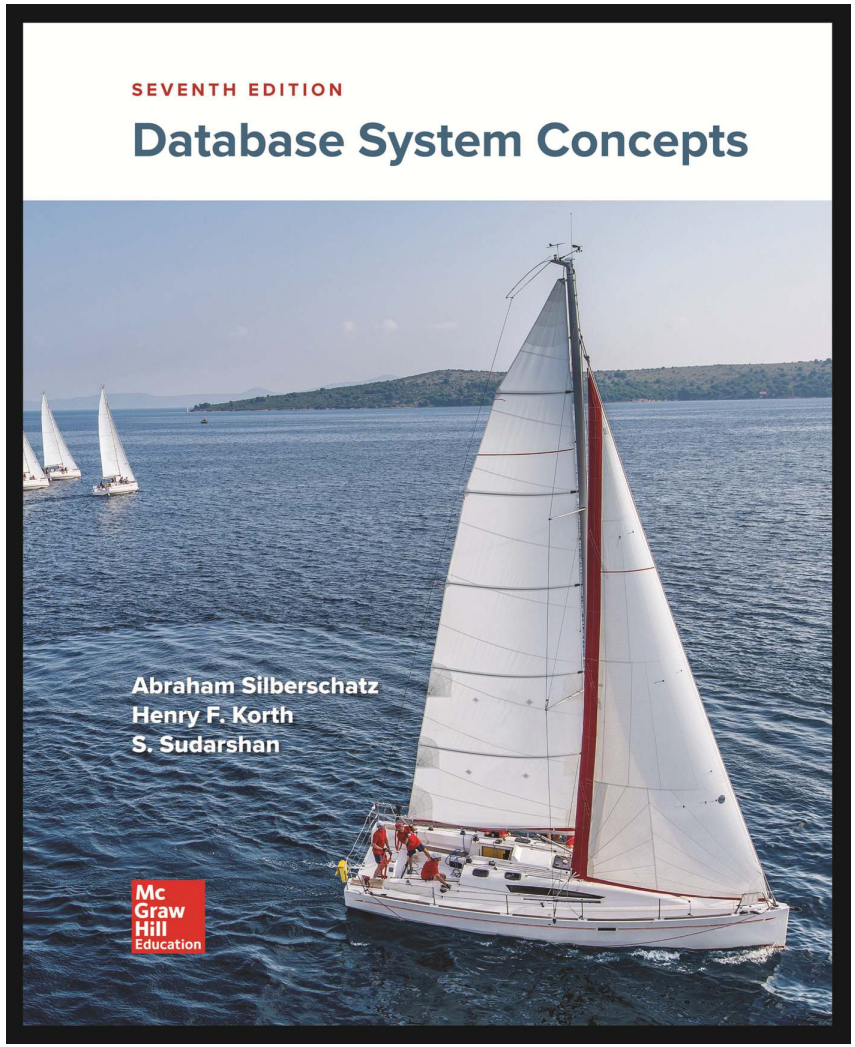
4 **Views and Their Uses** – Views are virtual tables that simplify queries and restrict data access, but they are not physically stored unless materialized.

3 **Transactions and Atomicity** – Transactions ensure database consistency by executing multiple `SQL` operations as a unit, requiring either a commit to save changes or a rollback to undo them.

2 **Integrity Constraints and Referential Integrity** – Constraints like `NOT NULL`, `PRIMARY KEY`, and foreign keys enforce data integrity, ensuring valid relationships between tables.

1 **Indexing for Performance Optimization** – Indexes speed up database queries by enabling efficient lookups, reducing the need for full table scans.

Database System Concepts



Content has been extracted from *Database System Concepts*, Seventh Edition, by Silberschatz, Korth and Sudarshan. Mc Graw Hill Education. 2019.
Visit `https://db-book.com/`.