四川大學
SICHUAN UNIVERSITY

Database System Concepts

Intermediate SQL

伍元凯

College of Computer Science (Software), Sichuan University

wuyk0@scu.edu.cn

2023/03/15

Natural Join as a Cartesian Product

The natural join of $R(A, B, C, D)$ and $S(E, B, D)$ is defined as follows:

$$\begin{aligned} R \bowtie S = & \Pi_{R.A, R.B, R.C, R.D, S.E} \\ & \sigma_{R.B=S.B \wedge R.D=S.D}(R \times S) \end{aligned} \tag{1}$$

In other words, the **natural join** is a Cartesian product where only the tuples that match on the common attributes are retained.

employees

eid	ename	age	salary
1	John	30	50000
2	Sarah	25	40000
3	David	35	60000

departments

eid	dname	location
1	Sales	Shanghai
2	Engineering	Chengdu
3	Marketing	Chongqing

```sql
SELECT *
FROM employees
NATURAL JOIN departments;
```

eid	ename	age	salary	dname	location
1	John	30	50000	Sales	Shanghai
2	Sarah	25	40000	Engineering	Chengdu
3	David	35	60000	Marketing	Chongqing

Properties of Natural Join

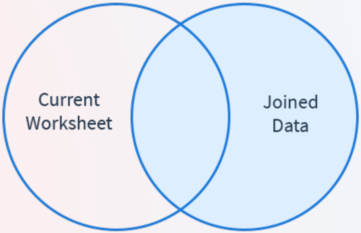
- **Commutativity(交换性):** $R \bowtie S = S \bowtie R$
- **Associativity(结合性):** $(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$
- **Idempotence(幂等性):** $R \bowtie R = R$
- **Compatibility with Projection(与投影的兼容性):** $\sigma_\theta(R \bowtie S) = \sigma_\theta(R) \bowtie \sigma_\theta(S)$

Theta join is a type of join operation in relational algebra that combines rows from two or more tables based on a condition specified using a comparison operator:

$$R \bowtie_C S = \sigma_C(R \times S) \tag{2}$$

Sometimes we want to include all the rows from one table in a join operation, even if **there is no matching row** in the other table. This is where outer join comes in.

- **Left Outer Join:**
  $R \bowtie S \cup (R - \Pi_R(R \bowtie S)) \times \{(NULL, \cdots, NULL)\}$
- **Right Outer Join:**
  $R \bowtie S \cup \{(NULL, \cdots, NULL)\} \times (S - \Pi_S(R \bowtie S))$
- **Full Outer Join:** Union of Left and Right Outer Join



CUST ID	LAST	FIRST
1	Smith	Jane
2	Johnson	Dave
3	Wilson	Mike
4	Jones	Sarah
5	Brown	Emma

CUST ID	ORDER ID	ORDER TOTAL
1	1001	$10.81
1	1002	$50.25
3	1003	$12.36
5	1004	$30.33
6	1005	$40.42

CUST ID	LAST	FIRST	CUST ID (1)	ORDER ID	ORDER TOTAL
1	Smith	Jane	1	1001	$10.81
1	Smith	Jane	1	1002	$50.25
3	Wilson	Mike	3	1003	$12.36
5	Brown	Emma	5	1004	$30.33
			6	1005	$40.42

目录

- $A(\texttt{ID}, \texttt{Name}, \texttt{Age})$
- $B(\texttt{ID}, \texttt{City}, \texttt{State})$

Query:

```sql
SELECT A.ID, A.Name, A.Age, B.City, B.State
FROM A, B
WHERE A.ID = B.ID AND B.State = 'CA';
```

Natural Join:

```sql
SELECT A.ID, A.Name, A.Age, B.City, B.State
FROM A NATURAL JOIN B
WHERE B.State = 'CA';
```

Thus, we see that these SQL queries and the natural join are all equivalent in this case.

General form in SQL:

Natural Join:

```
SELECT A1, A2, ... , An
FROM R1 NATURAL JOIN R2 NATURAL JOIN ... NATURAL JOIN Rm
WHERE P;
```

E an expression with natural joins:

```
SELECT E1, E2, ..., Em
```

- It **simplifies** queries.
- It **automatically** matches columns.
- It can be used to join **multiple** tables.
- It may not always produce the desired results if the column names or data types are not consistent across the tables being joined.
- It may require a lot of computation to determine the matching columns and perform the join.
- It may not be flexible enough to handle certain types of joins or conditions that require more complex logic.

Join … Using

```sql
SELECT A1, A2
FROM R1 JOIN R2 USING (A1, A2);
```

Only requiring R1.A1 = R2.A1 and R1.A2 = R2.A2. Even if R1 and R2 both have an attributes named A3, it is not required that R1.A3 = R2.A3.

JOIN-ON query

```
SELECT *
FROM table1
JOIN table2 ON table1.column_name = table2.column_name
WHERE table1.another_column_name > 10;
```

Equal query

```
SELECT *
FROM table1, table2
WHERE table1.column_name = table2.column_name AND table1.
    another_column_name > 10;
```

Order:

1. FROM clause
2. JOIN clause
3. WHERE clause

employees:

emp_id	emp_name	dept_id
1	John	1
2	Jane	1
3	Bob	2
4	Alice	null

departments:

dept_id	dept_name
1	Sales
2	Marketing

Intention: we want to get a list of **all** employees and their department names.

```
SELECT emp_name, dept_name
FROM employees
NATURAL JOIN departments;
```

emp_name	dept_name
John	Sales
Jane	Sales
Bob	Marketing

```sql
SELECT emp_name, dept_name
FROM employees
NATURAL LEFT OUTER JOIN departments;
```

emp_name	dept_name
John	Sales
Jane	Sales
Bob	Marketing
Alice	NULL

```sql
SELECT emp_name, dept_name
FROM departments
NATURAL RIGHT OUTER JOIN employees;
```

emp_name	dept_name
John	Sales
Jane	Sales
Bob	Marketing
Alice	NULL

The symmetry between LEFT OUTER JOIN and RIGHT OUTER JOIN.

employees:

emp_id	emp_name	dept_id
1	John	1
2	Jane	1
3	Bob	2
4	Alice	null

departments:

dept_id	dept_name
1	Sales
2	Marketing
3	IT

```sql
SELECT emp_name, dept_name
FROM employees
NATURAL FULL OUTER JOIN departments;
```

emp_name	dept_name
John	Sales
Jane	Sales
Bob	Marketing
Alice	NULL
NULL	IT

When we use an ON condition in an outer join, the join condition is **only applied to the matching rows** between the two tables. The rows that do not have a match will have **NULL** values in the columns of the non-matching table.

```
SELECT emp_name, dept_name
FROM employees
LEFT OUTER JOIN departments
ON employees.dept_id = departments.dept_id;
```

emp_name	dept_name
John	Sales
Jane	Sales
Bob	Marketing
Alice	NULL

```sql
SELECT emp_name, dept_name
FROM employees
LEFT OUTER JOIN departments
ON employees.dept_id = departments.dept_id
WHERE departments.dept_id IS NULL;
```

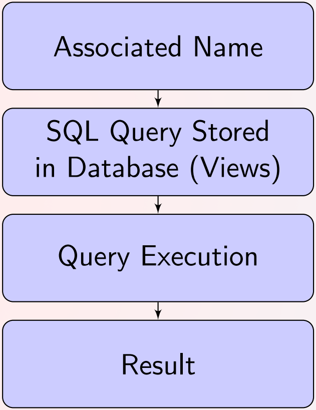
The WHERE condition then filters out the rows where there is a matching row in the "departments" table.

emp_name	dept_name
Alice	NULL

目录

- Inner Join: Returns only the rows that have matching values in both tables.
- Left Outer Join: Returns all the rows from the left table and the matching rows from the right table. If there is no match in the right table, the result will contain null values in the right table columns.
- Right Outer Join: Returns all the rows from the right table and the matching rows from the left table. If there is no match in the left table, the result will contain null values in the left table columns.
- Full Outer Join: Returns all the rows from both tables, including the non-matching rows. If a row in one table has no matching row in the other table, the result will contain null values in the columns of the non-matching table.

- ON: Specifies the join condition between two tables, including both the columns to join on and any additional conditions.
- USING: Specifies the join condition between two tables, but only for the columns with the same name in both tables.
- NATURAL: Joins two tables based on columns with the same name and data type.




```sql
SELECT * FROM my_view;
```

```sql
CREATE VIEW view_name AS
SELECT column1, column2, ...
FROM table1, table2, ...
WHERE condition;
```

column1	column2	...
...	...	...

View is not precomputed but instead is **computed by executing the query whenever the virtual result is used.**

Problem

Create a view that displays the names of all employees along with the name of their department.

Solution

To create this view in SQL, you can use the following syntax:

```sql
CREATE VIEW employee_details AS
SELECT e.name AS employee_name, d.name AS department_name
FROM employees e JOIN departments d
ON e.department_id = d.department_id;
```

You can then query this view just like you would query a table:

```sql
SELECT * FROM employee_details;
```

目录

```sql
CREATE VIEW sales_by_product AS
SELECT product_id, SUM(quantity) AS total_quantity
FROM sales
GROUP BY product_id;
CREATE VIEW sales_by_date AS
SELECT order_date, SUM(quantity) AS total_quantity
FROM sales
GROUP BY order_date;
```

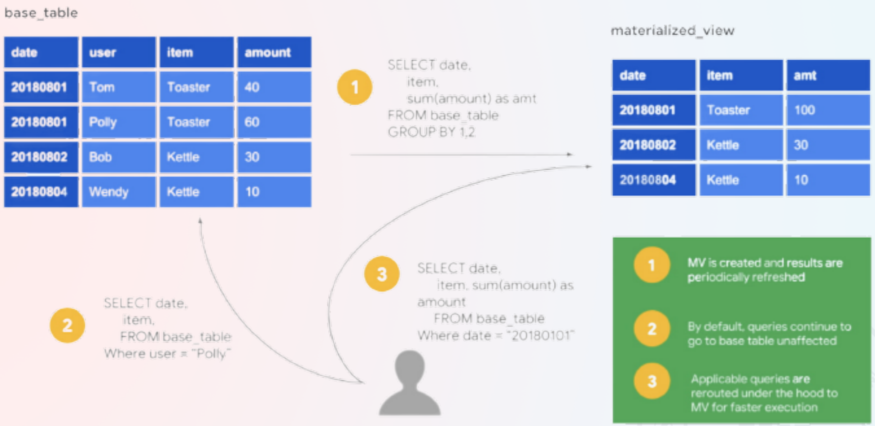
Queries with Views

```sql
SELECT p.product_id, d.order_date, p.total_quantity * d.
    total_quantity AS total_sales
FROM sales_by_product p
JOIN sales_by_date d
ON p.product_id = d.product_id;
```

Equavalent Queries

```sql
SELECT s.product_id, s.order_date, SUM(s.quantity) AS total_sales
FROM sales s
GROUP BY s.product_id, s.order_date;
```

目录



1

[1]https://cloud.google.com/blog/products/data-analytics/bigquery-materialized-views-now-ga

目录

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

Insert nonexist department and building

```sql
INSERT INTO ins_info VALUES ( "69" , "WHITE" , "Taylor" )
```

Modifications are generally **not permitted** on view relations, except in limited cases.

Some allowed conditions:

- The FROM has **only one relation**.
- The SELECT contains **only attributes** and does not have any expressions, aggregation, or distinct specification.
- It does not have a **NOT NULL constraint** and is not part of a **PRIMARY KEY**.
- The query does not have a **GROUP BY** or **HAVING** clause.

```sql
CREATE VIEW high_earners AS
SELECT *
FROM employees
WHERE salary >= 100000
WITH CHECK OPTION;
```

```sql
INSERT INTO high_earners (name, department_id, salary)
VALUES ('John Smith', 1, 90000);

ERROR: new row violates check option for view "high_earners"
```

The WITH CHECK OPTION clause ensures that any inserts or updates to the view must satisfy the conditions specified in the view's WHERE clause.

A TRANSACTION

```sql
BEGIN TRANSACTION;
SELECT balance FROM accounts WHERE account_number = '123456'; -- retrieve current balance
UPDATE accounts SET balance = balance - 500 WHERE account_number = '123456'; -- subtract $500 from balance
INSERT INTO transactions (account_number, amount, transaction_type) VALUES ('123456', 500, 'withdrawal'); -- log the transaction
COMMIT;
```

The BEGIN TRANSACTION statement marks the beginning of the transaction. Finally, the COMMIT statement marks the successful completion of the transaction.

Commit

- Saves changes made to the database since the last commit
- Makes changes permanent and visible to other users
- Releases any locks held on the affected data
- Ends the current transaction
- Syntax: `COMMIT;`

Detailed in Chapter 17

Rollback

- Undoes changes made to the database since the last commit
- Discards any changes made during the current transaction
- Reverts the database to its previous state
- Releases any locks held on the affected data
- Syntax: `ROLLBACK;`

Commit and Rollback

```
BEGIN TRANSACTION;
SELECT COUNT(*) FROM users WHERE email = 'example@example.com';
-- If the email already exists, rollback the transaction and
    display an error message
IF @@ROWCOUNT > 0
BEGIN
ROLLBACK;
PRINT 'Error: Email address already in use.';
END
-- If the email does not exist, insert the new user into the table
    and commit the transaction
ELSE
BEGIN
INSERT INTO users (name, email) VALUES ('John Doe', '
    example@example.com');
COMMIT;
PRINT 'Success: User created.';

END
```

目录

NOT NULL constraint

```sql
CREATE TABLE employees (
    id INTEGER PRIMARY KEY,
    name TEXT NOT NULL,
    age INTEGER,
    department TEXT
);
```

```sql
INSERT INTO employees (id, age, department) VALUES (1, 25, 'Sales')
    ;

ERROR:  null value in column "name" violates not-null constraint
```

UNIQUE constraint

```sql
CREATE TABLE users (
    id INTEGER PRIMARY KEY,
    username TEXT,
    email TEXT UNIQUE
);
```

```sql
INSERT INTO users (username, email) VALUES ('johndoe', 'johndoe@example.com');

ERROR:  duplicate key value violates unique constraint "users_email_key"
```

CHECK clause

```sql
CREATE TABLE orders (
    id INTEGER PRIMARY KEY,
    customer_name TEXT,
    order_date DATE,
    order_total NUMERIC CHECK (order_total >= 0)
);
```

```sql
INSERT INTO orders (customer_name, order_date, order_total) VALUES
    ('John Doe', '2023-03-13', -50.00);

ERROR:  new row for relation "orders" violates check constraint "
    orders_order_total_check"
```

```sql
CREATE TABLE customers (
  id INTEGER PRIMARY KEY,
  name TEXT NOT NULL
);

CREATE TABLE orders (
  id INTEGER PRIMARY KEY,
  customer_id INTEGER REFERENCES customers(id),
  order_date DATE NOT NULL,
  total_amount DECIMAL(10, 2) NOT NULL
);
```

INSERT an Nonexist ID

```sql
INSERT INTO orders (customer_id, order_date, total_amount)
VALUES (2, '2023-03-14', 100.00);

ERROR:  insert or update on table "orders" violates foreign key
    constraint "orders_customer_id_fkey"
DETAIL:  Key (customer_id)=(2) is not present in table "customers".
```

```sql
CREATE TABLE customers (
  id INTEGER PRIMARY KEY,
  name TEXT NOT NULL
);

CREATE TABLE orders (
  id INTEGER PRIMARY KEY,
  customer_id INTEGER REFERENCES customers(id) ON DELETE CASCADE,
  order_date DATE NOT NULL,
  total_amount DECIMAL(10, 2) NOT NULL,
  status TEXT DEFAULT 'pending'
);

CREATE TABLE payments (
  id INTEGER PRIMARY KEY,
  order_id INTEGER REFERENCES orders(id) ON DELETE SET NULL,
  payment_date DATE NOT NULL,


  amount DECIMAL(10, 2) NOT NULL
);
```

```sql
CREATE TABLE refunds (
  id INTEGER PRIMARY KEY,
  order_id INTEGER REFERENCES orders(id) ON DELETE SET DEFAULT,
  refund_date DATE NOT NULL,
  amount DECIMAL(10, 2) NOT NULL
);
```

```sql
INSERT INTO orders (customer_id, order_date, total_amount) VALUES
    (1, '2023-03-13', 50.00);
INSERT INTO orders (customer_id, order_date, total_amount) VALUES
    (1, '2023-03-14', 100.00);

INSERT INTO payments (order_id, customer_id,  payment_date, amount)
     VALUES (1, 1, '2023-03-15', 25.00);
INSERT INTO payments (order_id, customer_id,  payment_date, amount)
     VALUES (2, 1, '2023-03-16', 75.00);

INSERT INTO refunds (order_id, refund_date, amount) VALUES (1, '
    2023-03-17', 10.00);
```

```sql
DELETE FROM customers WHERE id = 1;
```

Orders Table

id	customer_id	order_date	total_amount	status

Payments Table

id	order_id	payment_date	amount
1	NULL	2023-03-15	25.00
2	NULL	2023-03-16	75.00

Refunds Table

id	order_id	refund_date	amount
1	NULL	2023-03-17	10.00

```sql

CREATE TABLE customers (
  id INT PRIMARY KEY,
  name VARCHAR(50),
  email VARCHAR(50) UNIQUE
);

CREATE TABLE orders (
  id INT PRIMARY KEY,
  customer_id INT,
  total_amount DECIMAL(10, 2),
  CONSTRAINT fk_orders_customers
    FOREIGN KEY (customer_id)
    REFERENCES customers(id)
    ON UPDATE CASCADE
    ON DELETE CASCADE
);
```

Example Tables

Customer

id	name
1	Alice
2	Bob

orders

id	customer_id	total_amount
1	1	100.00
2	1	50.00
3	2	75.00

```sql
UPDATE customers SET id = 4 WHERE id = 1;

DELETE FROM customers WHERE id = 2;
```

目录

```
CREATE TABLE example_table (
  id INTEGER PRIMARY KEY,
  name VARCHAR(50),
  email VARCHAR(100),
  CONSTRAINT email_unique UNIQUE (email)
);
```

In this example, the email$_{u}niqueconstraintiscreatedasauniqueconstraintontheemailcolumn.$

```
ALTER TABLE example_table DROP CONSTRAINT email_unique;
```

This will drop the email$_{u}niqueconstraintfromtheexample_{t}able.$

```sql
-- Create the orders table with a foreign key to the customers
    table
CREATE TABLE orders (
    id SERIAL PRIMARY KEY,
    customer_id INTEGER REFERENCES customers(id),
    total_amount NUMERIC(10, 2)
);
-- Create the customers table with a credit_limit column
CREATE TABLE customers (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100),
    credit_limit NUMERIC(10, 2)
);
-- Define the credit limit constraint as initially deferred
ALTER TABLE customers ADD CONSTRAINT credit_limit_constraint
    CHECK (SELECT SUM(total_amount) FROM orders WHERE customer_id =
        customers.id) <= credit_limit


    DEFERRABLE INITIALLY DEFERRED;
```

- Initially deferred constraint checking:
  - Defers checking of constraints until end of transaction
  - Allows transaction to proceed even if constraints are violated
  - Assumes constraints will be satisfied by end of transaction
  - May improve performance by reducing number of checks
  - Increases risk of violating constraints
- Immediate constraint checking (default):
  - Checks constraints immediately after each SQL statement
  - Prevents transactions with constraint violations
  - Provides immediate feedback on data inconsistencies or corruption
  - May be more resource-intensive
  - Can result in more frequent transaction rollbacks

目录

```sql
CREATE TABLE orders (
  id SERIAL PRIMARY KEY,
  customer_id INTEGER NOT NULL,
  total_amount DECIMAL(10,2) NOT NULL,
  status VARCHAR(20) NOT NULL CHECK (status IN ('pending', 'shipped', 'delivered')),
  CONSTRAINT check_total_amount CHECK (
    total_amount >= (
      SELECT COALESCE(SUM(total_amount), 0)
      FROM orders
      WHERE customer_id = NEW.customer_id
    )
  )
);
```

The check_total_amount constraint ensures that the total_amount for a new order is greater than or equal to the sum of all previous orders for the same customer.

```sql
CREATE ASSERTION max_total_amount
CHECK (
  SELECT MAX(total_amount)
  FROM orders
) <= 1000;
```

This assertion ensures that the maximum value of total_amount in the orders table is always less than or equal to 1000. If the condition is not satisfied, any attempts to modify the data in the orders table will fail.

目录

Date and Time Types in SQL

- **DATE**: Used to store dates in the format YYYY-MM-DD.
- **TIME**: Used to store times in the format HH:MM:SS.
- **DATETIME**: Used to store dates and times in the format YYYY-MM-DD HH:MM:SS.
- **TIMESTAMP**: Similar to DATETIME, but with the ability to store fractional seconds in the format YYYY-MM-DD HH:MM:SS[.fraction].
- `YEAR(datetime)`: Extracts the year from a DATETIME or TIMESTAMP value.
- `MONTH(datetime)`: Extracts the month (1-12) from a DATETIME or TIMESTAMP value.
- `DAY(datetime)`:...

- CURRENT_DATE: Returns the current date in the format YYYY-MM-DD.
- CURRENT_TIME: Returns the current time in the format HH:MM:SS.
- CURRENT_TIMESTAMP: ......
- DATEDIFF(interval, datetime1, datetime2): Returns the difference between two DATETIME or TIMESTAMP values in the specified interval (e.g., year, month, day, hour, minute, second).

```
UPDATE my_table
SET my_column = CURRENT_TIMESTAMP
WHERE my_condition;
```

目录

- **CAST** is used to convert data from one data type to another. For example, you might use CAST to convert a string into a numeric value so that you can perform mathematical calculations on it.
- **COALESCE** is used to return the first non-null value in a list of expressions. This can be useful when you want to use a default value if a particular column is null.
- **DECODE** is used to compare a value with a set of conditions and return a result based on the condition that is met. It is similar to the **CASE** statement, but with a different syntax.

```sql
SELECT AVG(CAST(total AS DECIMAL(10,2))) AS avg_total
FROM orders;
```

NULL middle_name to ''

```sql
SELECT CONCAT(first_name, ' ', COALESCE(middle_name, ''), ' ', last_name) AS full_name
FROM employees;
```

```sql
SELECT product_name, DECODE(category, 1, 'Electronics', 2, 'Clothing', 3, 'Home Goods', 'Unknown') AS category_name
FROM products;
```

目录

```sql
CREATE TABLE employees (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255),
    department VARCHAR(255) DEFAULT 'unknown',
    salary INT
);

INSERT INTO employees (name, salary)
VALUES ('John Smith', 50000);
```

id	name	department	salary
1	John Smith	unknown	50000

In SQL, **CLOB** (Character Large Object) and **BLOB** (Binary Large Object) are used to store large amounts of text or binary data (image, movie), respectively.

```
    id INT PRIMARY KEY,
    movie_review CLOB(100KB),
    movie BLOB(10GB)

INSERT INTO my_table (id, movie_review, movie)
VALUES (1, 'This is a movie.', x'0123456789abcdef');
```

User-Defined Types: These are custom data types that can be created using the CREATE TYPE statement.

```
CREATE TYPE person_type AS (first_name VARCHAR(50), last_name
    VARCHAR(50), age INT);
```

Domains: These are constraints that can be defined once and then applied to multiple columns in one or more tables.

```
CREATE DOMAIN email_address AS VARCHAR(255) CHECK (VALUE LIKE '%@
    %.%');
```

目录

```sql
CREATE TABLE my_table (
id INT AUTO_INCREMENT,
name VARCHAR(50),
PRIMARY KEY (id)
);

INSERT INTO my_table (name) VALUES ('New row');
```

Then the database will automatically generate a new id value that is **larger than any existing id values** in the table.

```sql
CREATE TABLE my_table (
  id INT GENERATED ALWAYS AS IDENTITY,
  name VARCHAR(50),
  PRIMARY KEY (id)
);
```

In SQL, a sequence is a database object that generates a series of unique integer values, which can be used as primary key values or other purposes. Sequences are supported by many SQL database systems, including PostgreSQL, Oracle, and IBM DB2.

```
CREATE SEQUENCE my_sequence
    START WITH 1
    INCREMENT BY 1
    NO MAXVALUE
    NO CYCLE;

CREATE TABLE my_table (
    id integer NOT NULL DEFAULT nextval('my_sequence'),
    name varchar(50),
    PRIMARY KEY (id)
);
```

Extensions

目录

```sql
CREATE TABLE new_table_name LIKE existing_table_name;
```

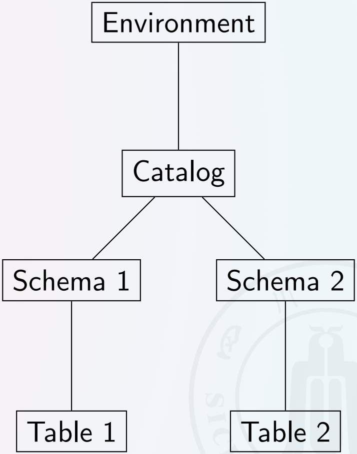
The new table will have the same column names, data types, and constraints as the existing table, but it will not **have any data**.

```sql
CREATE TABLE new_table_name AS
SELECT column1, column2, ...
FROM existing_table_name
WHERE ...
WITH DATA;
```

In this syntax, the "WITH DATA" clause specifies that the new table should be **created with the data from the SELECT statement**.

目录

- A **schema** is typically associated with a single user or application, and can be used to organize database objects based on the needs of that user or application.
- A **catalog** can contain multiple schemas, and can be used to organize database objects across multiple users or applications.
- **Environments** can be customized and configured to optimize database performance, security, and other settings based on the needs of the database or application.

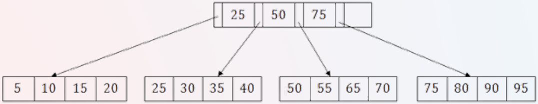


An index works like a **pointer** to the physical location of the data in the table.

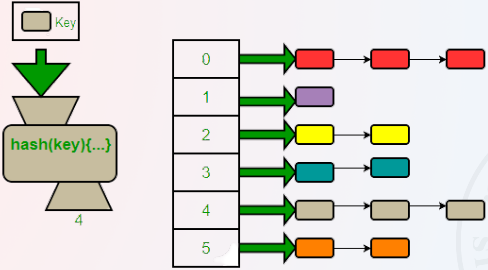
```sql
CREATE INDEX idx_customers_name ON customers (customer_name);
```

- **B-tree index**: This is the most common type of index in SQL, and it works by creating a balanced tree structure that allows for efficient searches, insertions, and deletions.
- **Hash index**: A hash index works by hashing the indexed column(s) to create a lookup table that maps each possible value to the corresponding rows in the table.
- **Bitmap index**: A bitmap index works by creating a bitmap for each possible value of the indexed column(s), where each bit represents a row in the table.
- ...

Detailed in Chapter 14



B+ Tree



Hash table

目录

海纳百川 有容乃大

```sql
GRANT permission [, permission, ...] ON object TO user [, user,
    ...] [WITH GRANT OPTION];
```

- permission is the type of permission to be granted (such as SELECT, INSERT, UPDATE, DELETE, etc.).
- object is the database object (such as a table, view, procedure, or function) on which the permission is being granted. **No tuple**
- user is the name of the user or role that is being granted the permission.
- WITH GRANT OPTION is an optional clause that allows the user being granted the permission to grant the same permission to other users.

```sql
REVOKE permission [, permission, ...] ON object FROM user [, user, ...];
```

Be careful when revoking permissions, as doing so can potentially disrupt any applications or processes that rely on those permissions.

```sql
CREATE ROLE president;

GRANT SELECT, INSERT, UPDATE, DELETE ON nuclear_weapon TO president;

GRANT president TO Trump;
```

Creates a new role named ”president”. Grants the all privileges on the nuclear weapon table to the president role. Grants the president role to the Trump, allowing them to inherit the permissions granted to use nuclear weapon.

目录

```sql
GRANT permission [, permission, ...] ON SCHEMA schema_name TO user [, user, ...] [WITH GRANT OPTION];

GRANT REFERENCES (deot.name) on department to Wang;
```

Authorization

目录

By Views

```sql
CREATE VIEW finance_employees AS
SELECT id, name, department
FROM employees
WHERE department = 'finance';

GRANT SELECT ON finance_employees TO finance_user;
```

In Oracle, you can use the DBMS_RLS package to define security policies. Here's an example:

```
CREATE OR REPLACE FUNCTION salary_policy (
  schema_name IN VARCHAR2,
  table_name IN VARCHAR2)
RETURN VARCHAR2
IS
BEGIN
  RETURN 'department = SYS_CONTEXT(''USERENV'', ''SESSION_DEPT'')';
END;

BEGIN
  DBMS_RLS.ADD_POLICY (
    object_schema   => 'my_schema',
    object_name     => 'employees',
    policy_name     => 'salary_policy',
    function_schema => 'my_schema',
    policy_function => 'salary_policy',
    statement_types => 'SELECT',

    update_check    => FALSE,
```

Attempt to improve and incorporate certain queries into the airlines dataset (or your own dataset) using the concepts covered in this week's course material. (尝试使用本周课程中所学的概念来改进并添加一些查询 (3 个) 到航空公司数据集（或您自己的数据集）中。)

Suppose we have three relation $R(A, B)$, $S(B, C)$, and $T(B, D)$, with all attributes declared as not null.

a. Given instances of relations R, S and T such that in the result of (R NATURAL LEFT OUTER JOIN S) NATURAL LEFT OUTER JOIN T, attribute C has a null value but attribute D has a non-null value.

b. Are there instances of R, S and T such that the result of R NATURAL LEFT OUTER JOIN (S NATURAL LEFT OUTER JOIN T) has a null value for C but a non-null value for D? Explain why of why not.

假设我们有三个关系 $R(A, B)$、$S(B, C)$ 和 $T(B, D)$，所有属性都被声明为非空。

a. 给出关系 R、S 和 T 的实例，使得在 (R NATURAL LEFT OUTER JOIN S) NATURAL LEFT OUTER JOIN T 的结果中，属性 C 具有空值，但属性 D 具有非空值。

b. 是否存在关系 R、S 和 T 的实例，使得 R NATURAL LEFT OUTER JOIN (S NATURAL LEFT OUTER JOIN T) 的结果对于 C 具有空值，但对于 D 具有非空值？解释为什么或为什么不。

Thanks
End of Chapter 4