CS2102 Database Systems

Lecture 7 – Programming with SQL

Instructor: Jiang Kan

Slides adapted from Prof Adi Yoga Sidi Prabawa

Roadmap

Refinement

3NF BCNF

Normal Forms

Triggers

Functional Dependencies

Stored Procedures

Query

Relational Algebra

SQL Query

Design

SQL DDL

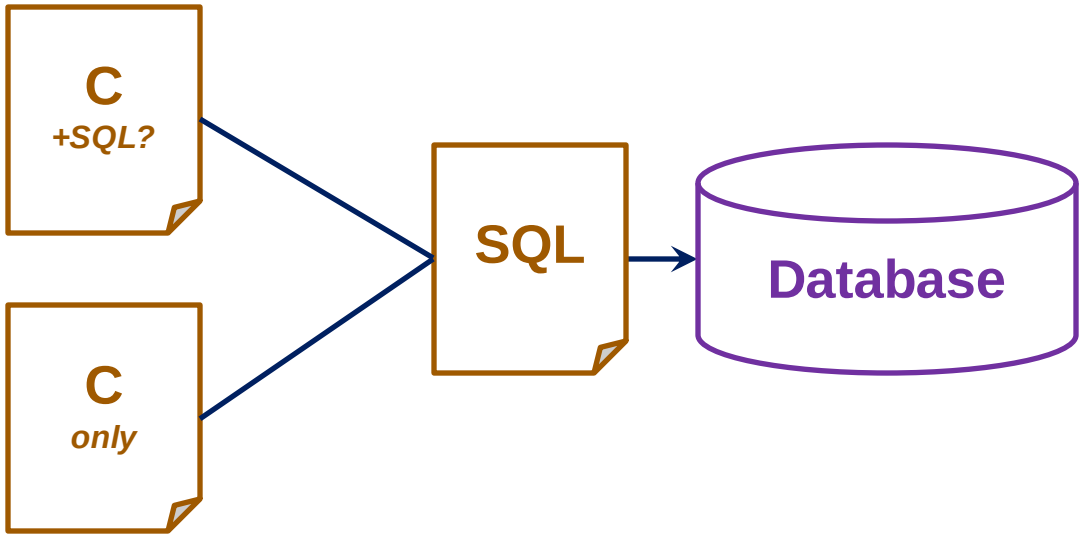
ER Diagram

Constraints

Slides adapted from Prof Adi Yoga Sidi Prabawa

Previous Lecture

We have discussed how to write **SQL** manually.



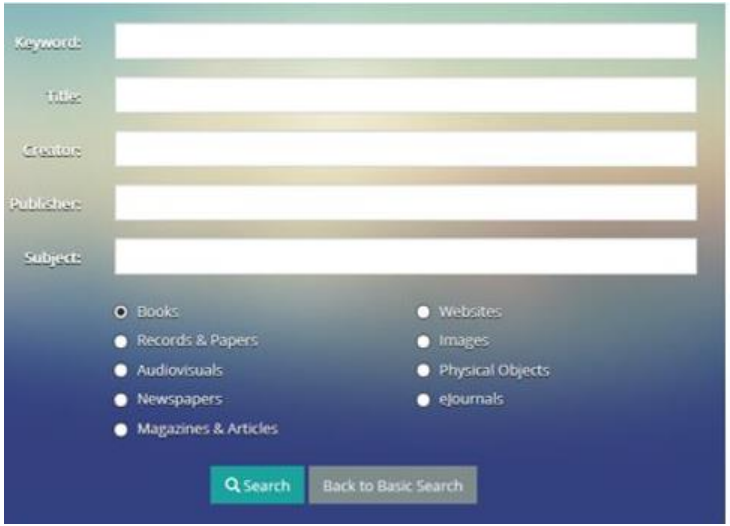
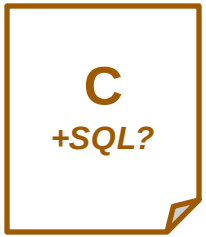


Image from National Library Board, Singapore

Often, we write Application which generates SQL!

Slides adapted from Prof Adi Yoga Sidi Prabawa

Statement-Level Interface

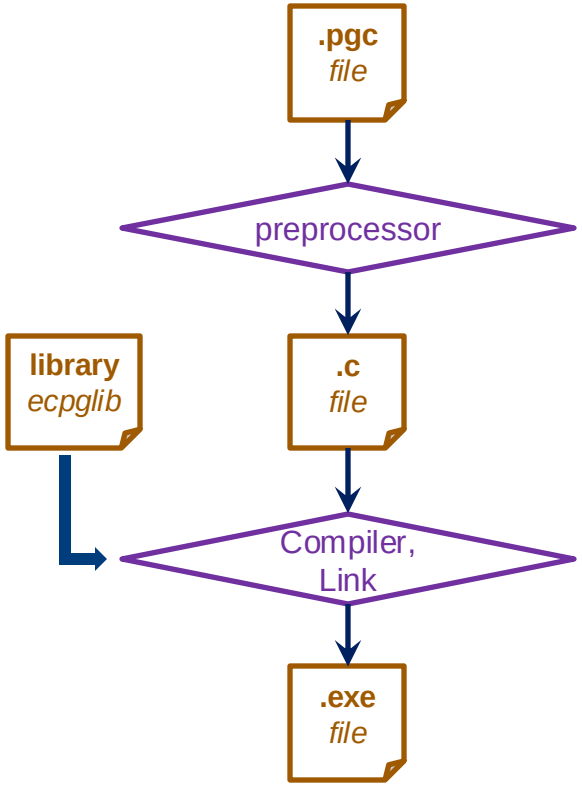


Slides adapted from Prof Adi Yoga Sidi Prabawa

Statement-Level Interface

- **Basic Idea**
  - Mix host language with SQL

    1. **Write** a program that mixes host language with SQL
    2. **Preprocess** the program using a preprocessor
    3. **Compile** the program into an executable code



Slides adapted from Prof Adi Yoga Sidi Prabawa

Statement-Level Interface

- **Declaration**
  - Declare variables
- **Connection**
  - Connect to database
- **Execution**
  - Prepare queries
  - Execute queries
  - Operate on result
- **Deallocation**
  - Release resources

Table "Sells"

supplName	prodName	price
...	...	...

```
void main() {
  EXEC SQL BEGIN DECLARE SECTION;
    char  supplier[30], product[30];
    float price;
  EXEC SQL END DECLARE SECTION;

  EXEC SQL CONNECT @localhost USER john;

  // some code that assigns values to
  // supplier, product and price

  EXEC SQL INSERT INTO
    Sells (supplName, prodName, price)
    VALUES (:supplier, :product, :price);

  EXEC SQL DISCONNECT;
}
```

Slides adapted from Prof Adi Yoga Sidi Prabawa

Statement-Level Interface

Table "Sells"

suppIName	prodName	price
...	...	...

Main keyword indicating statements to be preprocessed

Directives

```
void main() {
  EXEC SQL BEGIN DECLARE SECTION;
    char  supplier[30], product[30];
    float price;
  EXEC SQL END DECLARE SECTION;
  EXEC SQL CONNECT @localhost USER john;
  // some code that assigns values to
  // supplier, product and price

  EXEC SQL INSERT INTO
    Sells (supplName, prodName, price)
    VALUES (:supplier, :product, :price);
  EXEC SQL DISCONNECT;
}
```

Slides adapted from Prof Adi Yoga Sidi Prabawa

Statement-Level Interface

Table "Sells"

supplName	prodName	price
...	...	...

Connect to database with specified username

Disconnect releases resources

```
void main() {
  EXEC SQL BEGIN DECLARE SECTION;
    char  supplier[30], product[30];
    float price;
  EXEC SQL END DECLARE SECTION;
  EXEC SQL CONNECT @localhost USER john;
  // some code that assigns values to
  // supplier, product and price

  EXEC SQL INSERT INTO
    Sells (supplName, prodName, price)
    VALUES (:supplier, :product, :price);
  EXEC SQL DISCONNECT;
}
```

Slides adapted from Prof Adi Yoga Sidi Prabawa

Statement-Level Interface

Table "Sells"

supplName	prodName	price
...	...	...

Declare shared variable

Can be used by host language

Can be used by SQL

Automatically converts data between C and SQL

```
void main() {
EXEC SQL BEGIN DECLARE SECTION;
  char  supplier[30], product[30];
  float price;
EXEC SQL END DECLARE SECTION;
EXEC SQL CONNECT @localhost USER john;
// some code that assigns values to
// supplier, product and price

EXEC SQL INSERT INTO
  Sells (supplName, prodName, price)
  VALUES (:supplier, :product, :price);
EXEC SQL DISCONNECT;
}
```

Slides adapted from Prof Adi Yoga Sidi Prabawa

Statement-Level Interface

- **Execution**
  - Execute queries
  - Fetch results

Table "Sells"

suppIName	prodName	price
...	...	...

```
void main() {
  // declare variable, connect to database

  // Declare cursor
  EXEC SQL DECLARE cursor CURSOR FOR
  SELECT prodName FROM Sells WHERE supplName = :supplier;

  // Open cursor
  EXEC SQL OPEN cursor;
  EXEC SQL WHENEVER NOT FOUND DO BREAK;

  for(;;){
    EXEC SQL FETCH NEXT FROM cursor INTO :v_product;
    printf(">>> product: %s\n", v_product);
  }

  // close cursor, disconnect
}
```

Application Memory:

cursor →

v_product

prodName
...
...
...

Slides adapted from Prof Adi Yoga Sidi Prabawa

Statement-Level Interface

Table "Sells"

supplName	prodName	price
...	...	...

```
void main() {
  EXEC SQL BEGIN DECLARE SECTION;
    char  supplier[30], product[30];
    float price;
  EXEC SQL END DECLARE SECTION;
  EXEC SQL CONNECT @localhost USER john;
  // some code that assigns values to
  // supplier, product and price

  EXEC SQL INSERT INTO
    Sells (supplName, prodName, price)
    VALUES (:supplier, :product, :price);
  EXEC SQL DISCONNECT;
}
```

Get User Input

SQL query is **fixed**, only the argument changes

Static SQL

Slides adapted from Prof Adi Yoga Sidi Prabawa

Statement-Level Interface

- **Static SQL**
  - SQL query is fixed
- **Dynamic SQL**
  - SQL query is generated at run-time

```
void main() {
EXEC SQL BEGIN DECLARE SECTION;
  char  query[500];
EXEC SQL END DECLARE SECTION;

EXEC SQL CONNECT @localhost USER john;

// some code that generates SQL statement
// in the variable query

// For example:
//   query = "SELECT * FROM Sells";

EXEC SQL EXECUTE IMMEDIATE :query;

EXEC SQL DISCONNECT;
}
```

Slides adapted from Prof Adi Yoga Sidi Prabawa

Statement-Level Interface

- **Prepared SQL**
  - Prepare once
  - Execute multiple times
  - Use *placeholder (?)*

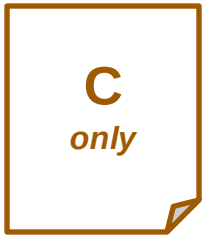
```
void main() {
EXEC SQL BEGIN DECLARE SECTION;
  const char *query = "INSERT INTO test
                       VALUES(?, ?);";
EXEC SQL END DECLARE SECTION;
EXEC SQL CONNECT @localhost USER john;
EXEC SQL PREPARE stmt FROM :query;

EXEC SQL EXECUTE stmt USING 42, 'foobar';
EXEC SQL DEALLOCATE PREPARE stmt;

EXEC SQL DISCONNECT;
}
```

Parsed, compiled by **database** only **ONCE**

Every prepared statement should be deallocated when no longer needed

Slides adapted from Prof Adi Yoga Sidi Prabawa

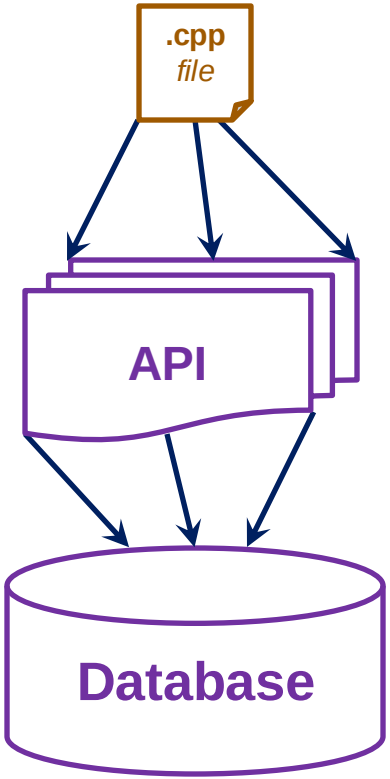
Call-Level Interface



Slides adapted from Prof Adi Yoga Sidi Prabawa

Call-Level Interface

- **Basic Idea**
  - Write everything in host language
    - **Call** functions from library through API
    - Library will handle the details
  - Examples:
    - Java Database Connectivity (JDBC)
      - https://jdbc.postgresql.org/
    - Open Database Connectivity (ODBS)
      - https://odbc.postgresql.org/



Slides adapted from Prof Adi Yoga Sidi Prabawa

Call-Level Interface

- **Declaration**
  - Declare variables
- **Connection**
  - Connect to database
- **Execution**
  - Execute queries
  - Operate on result
- **Deallocation**
  - Release resources

```
void main() {
    char *query;

    connection C("dbname = testdb user = postgres \
      password = **** hostaddr = 127.0.0.1 \
      port = 5432");

    query = "CREATE TABLE Company (" \
            "Id   INT  PRIMARY KEY," \
            "Name TEXT NOT NULL);";

    work W(C)
    W.exec(query);
    W.commit();

    C.disconnect();
}
```

Slides adapted from Prof Adi Yoga Sidi Prabawa

Call-Level Interface

- **Declaration**
  - Declare variables
- **Connection**
  - Connect to database
- **Execution**
  - Execute queries
  - Operate on result
- **Deallocation**
  - Release resources

```
import psycopg  # Host language library (Python)

# Connect to database
conn = psycopy.connect("host=...")

# Create cursor
cursor = conn.cursor()

# Open cursor by executing query
cursor.execute("SELECT supplName from Sells")

while True:
  row = cursor.fetchone()
  if row is None:
    break
  print(f">>> Supplier: {row[0]}")

# Cleanup
cursor.close()
conn.commit()
conn.close()
```

Slides adapted from Prof Christian von der Weth

Call-Level Interface

- **Declaration**
  - Declare variables
- **Connection**
  - Connect to database
- **Execution**
  - SQL statement
  - Parameter values
  - Sent to DB server separately
- **Deallocation**
  - Release resources

```
import psycopg  # Host language library

product = "some product"

# Connect to database
conn = psycopy.connect("host=...")

# Create cursor
cursor = conn.cursor()

# Execute Parameterized Statement
cursor.execute("SELECT supplName from Sells where prodName = %s", (product,))
# Following is NOT a parameterized statement
#cursor.execute("SELECT supplName from Sells where prodName = %s"%(product,))

while True:
  row = cursor.fetchone()
  if row is None:
    break
  print(f">>> Supplier: {row[0]}")

# Cleanup ...
```

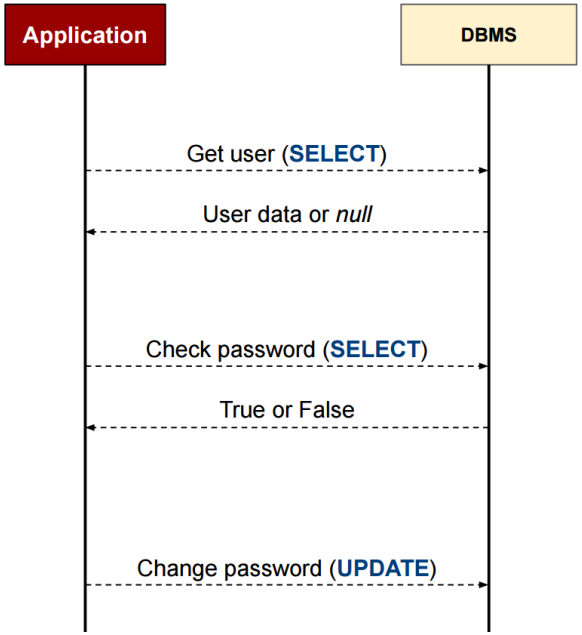
Slides adapted from Prof Christian von der Weth

Summary

- Statement-Level Interface *(SLI)*
  - Code is written in a mix of host language and SQL
    - Static SQL has fixed queries
    - Dynamic SQL generates queries at run-time
  - Preprocess directives before compiled into executable code
- Call-Level Interface *(CLI)*
  - Code is written only in host language
  - Use of API call to run SQL queries

Motivating example: Update User Password

- One task, to update user password
  - Check the user does exist
  - Check the new passwd is different from old
  - If all is OK, update the password table
- Three separate queries, poor performance
- What if requirement is changed:
  - New password different from last 2 old
  - Update application code
  - Re-deploy to all users
  - Maintenance is difficult



Slides adapted from Prof Christian von der Weth

SQL Functions and Procedures

SQL
only

Slides adapted from Prof Adi Yoga Sidi Prabawa

SQL Functions and Procedures

- SQL-Based Procedure Language
  - ISO standard: SQL/PSM *(persistent stored modules)*
  - Unfortunately, different vendors have different implementations:
    - Oracle `PL/SQL`
    - PostgreSQL `PL/pgSQL`
    - SQL Server `TransactSQL`
- Advantages
  - Code reuse
  - Ease of maintenance
  - Performance
  - Security

SQL Functions

Table "Scores"

Name	Mark
Alice	92
Bob	63
Cathy	58
David	47

Name	Grade
Alice	A
Bob	B
Cathy	C
David	F

- Suppose we want to convert numeric marks to letter grades
  - [70, 100] → A
  - [60, 70) → B
  - [50, 60) → C
  - [0, 50) → F

```
SELECT Name, CASE
    WHEN Mark >= 70 THEN 'A'
    WHEN Mark >= 60 THEN 'B'
    WHEN Mark >= 50 THEN 'C'
    ELSE 'F'
  END
FROM Scores;
```

Slides adapted from Prof Adi Yoga Sidi Prabawa

SQL Functions

Table "Scores"

Name	Mark
Alice	92
Bob	63
Cathy	58
David	47

- We can abstract away the main computation into functions
  - `SELECT convert(90);`
    - A
  - `SELECT convert(40);`
    - F

```
CREATE OR REPLACE FUNCTION convert(Mark INT)
RETURNS CHAR(1) AS $$
  SELECT       CASE
      WHEN Mark >= 70 THEN 'A'
      WHEN Mark >= 60 THEN 'B'
      WHEN Mark >= 50 THEN 'C'
      ELSE 'F'
    END;
$$ LANGUAGE sql;
```

Slides adapted from Prof Adi Yoga Sidi Prabawa

SQL Functions

Table "Scores"

Name	Mark
Alice	92
Bob	63
Cathy	58
David	47

Name	Grade
Alice	A
Bob	B
Cathy	C
David	F

- The query can now be simplified
  -
```
SELECT
  Name,
  convert(Mark)
FROM Scores;
```

```
CREATE OR REPLACE FUNCTION convert(Mark INT)
RETURNS CHAR(1) AS $$
  SELECT       CASE
      WHEN Mark >= 70 THEN 'A'
      WHEN Mark >= 60 THEN 'B'
      WHEN Mark >= 50 THEN 'C'
      ELSE 'F'
    END;
$$ LANGUAGE sql;
```

Slides adapted from Prof Adi Yoga Sidi Prabawa

SQL Functions

Table "Scores"

Name	Mark
Alice	92
Bob	63
Cathy	58
David	47

Name
Bob

- This allows for code reuse
  -
```
SELECT Name
FROM Scores
WHERE
  convert(Mark)='B';
```

```
CREATE OR REPLACE FUNCTION convert(Mark INT)
RETURNS CHAR(1) AS $$
  SELECT       CASE
      WHEN Mark >= 70 THEN 'A'
      WHEN Mark >= 60 THEN 'B'
      WHEN Mark >= 50 THEN 'C'
      ELSE 'F'
    END;
$$ LANGUAGE sql;
```

Slides adapted from Prof Adi Yoga Sidi Prabawa

SQL Functions

Table "Scores"

Name	Mark
Alice	92
Bob	63
Cathy	58
David	47

Name
Bob
Cathy

- and easy maintenance
  - SELECT Name
    FROM Scores
    WHERE
      convert(Mark)='B';

```
CREATE OR REPLACE FUNCTION convert(Mark INT)
RETURNS CHAR(1) AS $$
  SELECT       CASE
      WHEN Mark >= 70 THEN 'A'
      WHEN Mark >= 55 THEN 'B'
      WHEN Mark >= 40 THEN 'C'
      ELSE 'F'
    END;
$$ LANGUAGE sql;
```

Slides adapted from Prof Adi Yoga Sidi Prabawa

SQL Functions

- General Syntax
  - Interesting return types
    - *Set of existing tuples* *(table name)*
    - *Set of new tuples* *(RECORD/TABLE)*
    - *No return value* *(procedure)*
    - *Triggers* *(next week)*
  - Interesting languages
    - *plpgsql*

```
CREATE OR REPLACE FUNCTION <name>
  (<param> <type>, <param> <type>, ...)
RETURNS <type> AS $$

  <code>

$$ LANGUAGE <language>;
```

Function name

Function parameters and type

Return type

Enclosed within dollar-quote

Main body of the function

The language used by the code

Slides adapted from Prof Adi Yoga Sidi Prabawa

SQL Functions

- **One Existing Tuple**

Table "Scores"

Name	Mark
Alice	92
Bob	63
Cathy	58
David	47

Name	Mark
Alice	92

- Suppose we want to find any **one student** with the highest mark
  - `SELECT * FROM topStudent();`

Return all columns of Scores

```
CREATE OR REPLACE FUNCTION topStudent()
RETURNS Scores AS $$

  SELECT    *
  FROM      Scores
  ORDER BY  Mark DESC LIMIT 1;


$$ LANGUAGE sql;
```

Slides adapted from Prof Adi Yoga Sidi Prabawa

SQL Functions

- **Set of Existing Tuple**

Table "Scores"

Name	Mark
Alice	92
Bob	63
Cathy	58
David	92

Name	Mark
Alice	92
David	92

- Suppose we want to find all students with the highest mark
  - `SELECT * FROM topStudents();`
  - What about tuples not from `Scores` table?

```
CREATE OR REPLACE FUNCTION topStudents()
RETURNS SETOF Scores AS $$

  SELECT    *
  FROM      Scores
  WHERE     Mark = (SELECT MAX(Mark)
                    FROM Scores);

$$ LANGUAGE sql;
```

Slides adapted from Prof Adi Yoga Sidi Prabawa

SQL Functions

- **One New Tuple**

Table "Scores"

Name	Mark
Alice	92
Bob	63
Cathy	58
David	92

Mark	Count
92	2

- Suppose we want to find the highest mark and its count
  - `SELECT * FROM topMarkCount();`

```
CREATE OR REPLACE FUNCTION topMarkCount
  (OUT Mark INT, OUT Count INT)
RETURNS RECORD AS $$
  SELECT    Mark, COUNT(*)
  FROM      Scores
  WHERE     Mark = (SELECT MAX(Mark)
                    FROM Scores);
  GROUP BY  Mark;
$$ LANGUAGE sql;
```

Important: If we use RECORD, we must have at least two OUT parameters!

Slides adapted from Prof Adi Yoga Sidi Prabawa

SQL Functions

- **Set of New Tuple**

Table "Scores"

Name	Mark
Alice	92
Bob	63
Cathy	58
David	92

Mark	Count
92	2
63	1
58	1

- Suppose we want to find all distinct marks and their count
  - `SELECT * FROM markCounts();`

```
CREATE OR REPLACE FUNCTION markCounts
  (OUT Mark INT, OUT Count INT)
RETURNS SETOF RECORD AS $$
  SELECT    Mark, COUNT(*)
  FROM      Scores
  GROUP BY  Mark;


$$ LANGUAGE sql;
```

Slides adapted from Prof Adi Yoga Sidi Prabawa

SQL Functions

- **Set of New Tuple**

Table "Scores"

Name	Mark
Alice	92
Bob	63
Cathy	58
David	92

Mark	Count
92	2
63	1
58	1

- Suppose we want to find all distinct marks and their count
  - `SELECT * FROM markCounts();`
  - *Alternative*

```
CREATE OR REPLACE FUNCTION markCounts()
RETURNS TABLE(Mark INT, Count INT) AS $$

  SELECT    Mark, COUNT(*)
  FROM      Scores
  GROUP BY  Mark;


$$ LANGUAGE sql;
```

Slides adapted from Prof Adi Yoga Sidi Prabawa

SQL Functions

- **No Return Value?**
  - VOID

Table "Acct"

Name	Mark
Alice	1000
Bob	600
Cathy	1500
David	200

Name	Mark
Alice	900
Bob	600
Cathy	1500
David	300

- Suppose we want to transfer money from A to B
  - SELECT transfer ('Alice','Bob',100);

```
CREATE OR REPLACE FUNCTION transfer
  (frAcc TEXT, toAcc TEXT, amount INT)
RETURNS VOID AS $$
  UPDATE Acct SET balance = balance - amount
  WHERE  name = frAcc;
  UPDATE Acct SET balance = balance + amount
  WHERE  name = toAcc;

$$ LANGUAGE sql;
```

Slides adapted from Prof Adi Yoga Sidi Prabawa

SQL Procedures

- **No Return Value?**
  - PROCEDURE

Table "Acct"

Name	Mark
Alice	1000
Bob	600
Cathy	1500
David	200

Name	Mark
Alice	900
Bob	600
Cathy	1500
David	300

- Suppose we want to transfer money from A to B
  - `CALL transfer ('Alice','Bob',100);`

```
CREATE OR REPLACE PROCEDURE transfer
  (frAcc TEXT, toAcc TEXT, amount INT)
AS $$
  UPDATE Acct SET balance = balance - amount
  WHERE  name = frAcc;
  UPDATE Acct SET balance = balance + amount
  WHERE  name = toAcc;

$$ LANGUAGE sql;
```

Slides adapted from Prof Adi Yoga Sidi Prabawa

Stored Function vs Stored Procedure

- Return
  - Function must be declared to return something (a special case is void)
  - Procedure has no return, but can use out parameter

```
CREATE PROCEDURE add_proc
  (IN a INT, IN b INT, OUT sum INT)
AS $$
BEGIN
  sum := a + b;
END
$$ LANGUAGE plpgsql;
```

- Transaction
  - Procedure can commit or roll back.
  - Function cannot!

```
DO $$
DECLARE
  sum INT;
BEGIN
  CALL add_proc(2, 3, sum);
  RAISE NOTICE 'Sum: %', sum;
END $$;
```

Slides adapted from Prof Christian von der Weth

Summary

- SQL Functions
  - Returns a value
    - SQL data types
    - Set of existing tuples
    - Set of new tuples
  - `CREATE OR REPLACE FUNCTION <name>(...)`
    - `SELECT <name>(...)`
- SQL Procedures
  - No return value, but can use out parameter
  - `CREATE OR REPLACE PROCEDURE <name>(...)`
    - `CALL <name>(...)`

Is that it?
We use functions/procedures to execute SQL queries?

Control Structures

```
IF ... THEN ...
ELSIF ... THEN ...
ELSE ... END IF
```

```
LOOP ... END LOOP
     EXIT ... WHEN ...
WHILE ... LOOP ... END LOOP
FOR ... IN ... LOOP ... END LOOP
```

Slides adapted from Prof Adi Yoga Sidi Prabawa

Control Structures

- **Variables**

```
CREATE OR REPLACE FUNCTION swap(INOUT val1 INT, INOUT val2 INT)
RETURNS RECORD AS $$
DECLARE
  temp INT;
BEGIN

    temp := val1;   val1 := val2;   val2 := temp;

END;
$$ LANGUAGE plpgsql;
```

- Suppose we want to swap to integers
  - `SELECT swap(99, 11);` → `(11, 99)`
  - `SELECT swap(11, 99);` → `(99, 11)`

Slides adapted from Prof Adi Yoga Sidi Prabawa

Control Structures

- **Selection**

```
CREATE OR REPLACE FUNCTION sort(INOUT val1 INT, INOUT val2 INT)
RETURNS RECORD AS $$
DECLARE
  temp INT;
BEGIN
  IF val1 > val2 THEN
    temp := val1;   val1 := val2;   val2 := temp;
  END IF;
END;
$$ LANGUAGE plpgsql;
```

- Suppose we want to sort two integers
  - `SELECT sort(99, 11);` → `(11, 99)`
  - `SELECT sort(11, 99);` → `(11, 99)`

Slides adapted from Prof Adi Yoga Sidi Prabawa

Control Structures

- **Repetition**

```
CREATE OR REPLACE FUNCTION sum_to_x(IN x INT)
RETURNS INT AS $$
DECLARE
  s INT;  temp INT;
BEGIN
  s := 0;   temp := 1;
  WHILE temp <= x LOOP
    s := s + temp;   temp := temp + 1;
  END LOOP;
END;
$$ LANGUAGE plpgsql;
```

- Suppose we want to compute 1 + 2 + 3 + ... + x

Control Structures

- **Repetition**

```
CREATE OR REPLACE FUNCTION sum_to_x(IN x INT)
RETURNS INT AS $$
DECLARE
  s INT; temp INT;
BEGIN
  s := 0;   temp := 1;
  LOOP
    EXIT WHEN temp > x;
    s := s + temp;   temp := temp + 1;
  END LOOP;
END;
$$ LANGUAGE plpgsql;
```

in Imperative Language ...

```
while (true) {
  if (temp > x) break;
}
```

- Suppose we want to compute 1 + 2 + 3 + ... + x

Slides adapted from Prof Adi Yoga Sidi Prabawa

Control Structures

- **Question**

Table "Scores"

Name	Mark
Alice	92
Bob	63
Cathy	58
David	47

Name	Mark	Gap
Alice	92	NULL
Bob	63	29
Cathy	58	5
David	47	11

- Write a function for the following task:
  - Construct the table consisting of all students in Scores and their marks as well as the mark difference between them and the next highest mark or equal mark in some order
  - `SELECT * FROM score_gap();`

Control Structures

- **Question**

Table "Scores"

Name	Mark
Alice	92
Bob	63
Cathy	58
David	47

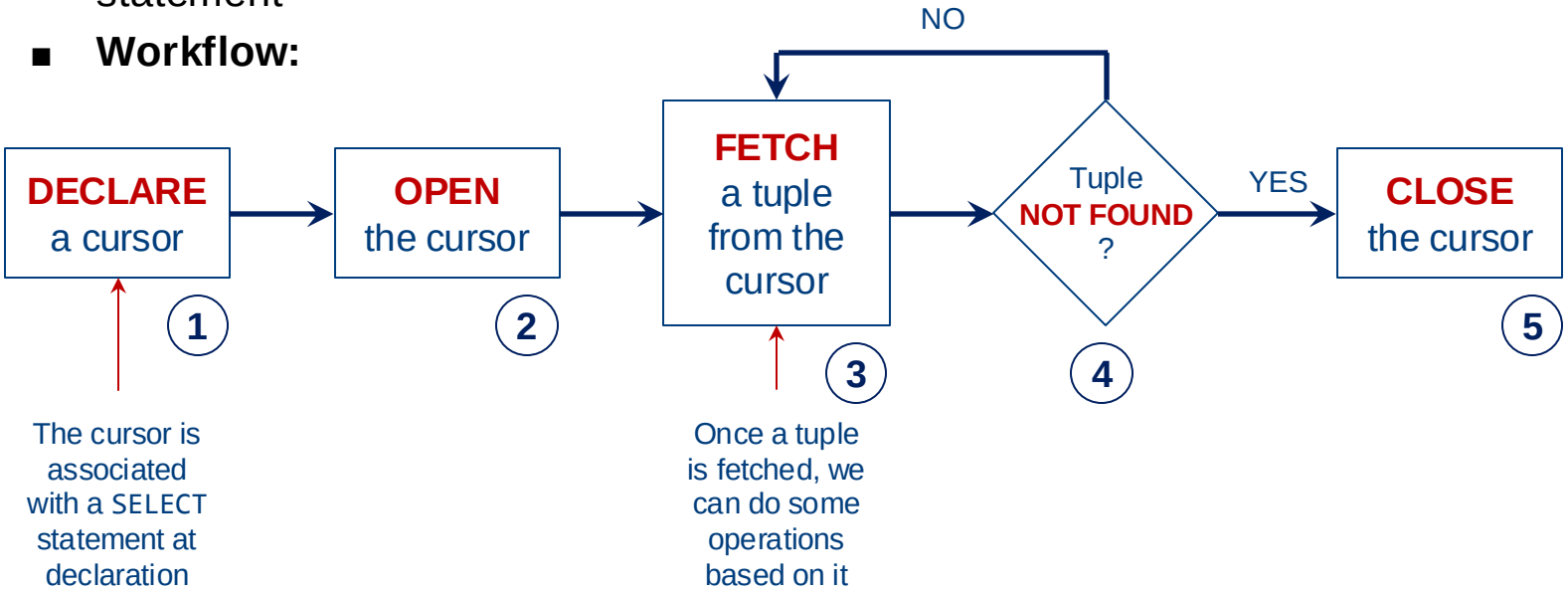
Name	Mark	Gap
Alice	92	NULL
Bob	63	29
Cathy	58	5
David	47	11

- Write a function for the following task:
  - **Idea?**
    - `SELECT * FROM Scores ORDER BY Mark DESC;`
    - Loop over the sorted sequence of students
      - How?
      - Use a cursor

Slides adapted from Prof Adi Yoga Sidi Prabawa

Control Structures

- **Cursor**
  - A cursor enables us to access each individual **row** returned by a SELECT statement
  - **Workflow:**



Slides adapted from Prof Adi Yoga Sidi Prabawa

Control Structures

- **Cursor**
  1. **DECLARE a cursor**
     - With an associated SELECT query
  2. **OPEN the cursor**
  3. **FETCH a tuple from the cursor**
  4. **Tuple NOT FOUND?**
     - Do operation on the value otherwise
     - Construct value, tuple by tuple
  5. **CLOSE the cursor**

```
CREATE OR REPLACE FUNCTION score_gap()
RETURNS TABLE(name_ TEXT, mark_ INT, gap INT) AS $$
DECLARE
  curs CURSOR FOR (SELECT * FROM Scores ORDER BY Mark DESC);
  r    RECORD;   prev INT;
BEGIN
  prev := -1;   OPEN curs;
  LOOP
    FETCH curs INTO r;
    EXIT WHEN NOT FOUND;
    name_ := r.Name;   mark_ := r.Mark;
    IF prev >= 0 THEN gap := prev – mark_;
    ELSE gap := NULL;
    END IF;
    RETURN NEXT;
    prev := r.Mark;
  END LOOP;
  CLOSE curs;
END;
$$ LANGUAGE plpgsql
```

Control Structures

- **Cursor**
  - Cursor movement
    - `FETCH curs INTO r;`
  - Other variants
    - `FETCH PRIOR FROM curs INTO r;`
      - Fetch from previous row
    - `FETCH FIRST FROM curs INTO r;`
    - `FETCH LAST FROM curs INTO r;`
    - `FETCH ABSOLUTE 3 FROM curs INTO r;`
      - Fetch the 3rd tuple
    - ...

curs →

Name	Mark
Alice	92
Bob	63
Cathy	58
⋮	
David	47
Eve	42
Felix	40
⋮	
Vanessa	10

Looping through Query Results using FOR

- **FOR**

```
FOR row IN query LOOP
  statements
END LOOP;
```

```
CREATE OR REPLACE FUNCTION score_gap()
RETURNS TABLE(name_ TEXT, mark_ INT, gap INT) AS $$
DECLARE
  r    RECORD;   prev INT;
BEGIN
  prev := -1;
  FOR r IN SELECT * FROM Scores ORDER BY Mark DESC
  LOOP
    name_ := r.Name; mark_ := r.Mark;
    IF prev >= 0 THEN gap := prev – mark_;
    ELSE gap := NULL;
    END IF;
    RETURN NEXT;
    prev := r.Mark;
  END LOOP;
END;
$$ LANGUAGE plpgsql
```

Slides adapted from Prof Christian von der Weth

Control Structures

- **Exercise**

Table "Scores"

Name	Mark
Alice	92
Bob	63
Cathy	58
David	47

Name	Mark
Bob	63
Cathy	58

- Write a function to retrieve the student(s) with **median** mark
  - What is median?
    - Let **n** be the total number of students
      - If **n** is odd, then the median is **$((n+1)/2)^{th}$** student
      - If **n** is even, then the median is the **$(n/2)^{th}$** and **$(n/2+1)^{th}$** students
      - Ties are broken *arbitrarily*

Slides adapted from Prof Adi Yoga Sidi Prabawa

Control Structures

- **Solution**
  1. **DECLARE a cursor**
     - What query?
  2. **OPEN the cursor**
  3. **FETCH a tuple from the cursor**
     - Can we fetch the median directly?
  4. **Tuple NOT FOUND?**
     - For simplicity, assume there is at least one
  5. **CLOSE the cursor**

```
CREATE OR REPLACE FUNCTION median_student()
RETURNS TABLE(Name_ TEXT, Mark_ INT) AS $$
```

```
DECLARE
  curs CURSOR FOR (SELECT * FROM Scores ORDER BY Mark DESC);
  r    RECORD;   num_student INT;
```

```
BEGIN
  OPEN curs;
```

```
  SELECT COUNT(*) INTO num_student FROM Scores;
  IF num_student%2 = 1 THEN
    FETCH ABSOLUTE (num_student+1)/2 FROM curs INTO r;
    Name_:= r.Name; Mark_ := r.Mark; RETURN NEXT;
  ELSE
    FETCH ABSOLUTE num_student/2 FROM curs INTO r;
    Name_:= r.Name; Mark_ := r.Mark; RETURN NEXT;
    FETCH ABSOLUTE num_student/2+1 FROM curs INTO r;
    Name_:= r.Name; Mark_ := r.Mark; RETURN NEXT;
  END IF;
```

```
  CLOSE curs;
END;
```

```
$$ LANGUAGE plpgsql
```

Control Structures

```
CREATE OR REPLACE FUNCTION median_student()
RETURNS setof Scores AS $$
DECLARE
  curs CURSOR FOR (SELECT * FROM Scores ORDER BY Mark DESC);
  r    RECORD;   num_student INT;
BEGIN
  OPEN curs;
  SELECT COUNT(*) INTO num_student FROM Scores;
  IF num_student%2 = 1 THEN
    FETCH ABSOLUTE (num_student+1)/2 FROM curs INTO r;
    RETURN NEXT r;
  ELSE
    FETCH ABSOLUTE num_student/2 FROM curs INTO r;
    RETURN NEXT r;
    FETCH ABSOLUTE num_student/2+1 FROM curs INTO r;
    RETURN NEXT r;
  END IF;
  CLOSE curs;
END;
$$ LANGUAGE plpgsql
```

Summary

- `plpgsql` Control Structures
  - Declare `DECLARE <var> <type> BEGIN`
  - Assignment `<var> := ...`
  - Selection `IF ... THEN ... ELSIF ... THEN ... ELSE ... END IF`
  - Repetition `LOOP ... END LOOP`
    `WHILE ... LOOP ... END LOOP`
    - Break `EXIT WHEN ...`
- Cursor
  - Declare → Open → Fetch → Check (*repeat*) → Close
    - `FETCH [PRIOR | FIRST | LAST | ABSOLUTE n] [FROM] <cursor> INTO <var>`

SQL Injection Attacks

Slides adapted from Prof Adi Yoga Sidi Prabawa

SQL Injection Attacks

- **What is It?**
  - A class of attacks on dynamic SQL
  - **Benign usage**
    - "Alice"
  - **Malicious Usage**
    - ";
      DROP TABLE T;
      SELECT '"

```
void main() {
    EXEC SQL BEGIN DECLARE SECTION;
      string query = "";
    EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO @localhost USER john;

    string name;
    cin >> name;  // user input
    query = query + "SELECT * FROM T WHERE Name = '"
                  + name
                  + "';";

    EXEC SQL EXECUTE IMMEDIATE :query;

    EXEC SQL DISCONNECT;
}
```

Generated Query

```
SELECT * FROM T
WHERE Name = '';
DROP TABLE T;
SELECT '';
```

Slides adapted from Prof Adi Yoga Sidi Prabawa

SQL Injection Attacks

- **How to Protect?**
  - Use prepares statements
  - **Why?**
    - SQL statement is compiled when it is prepared
    - At runtime, anything in `name` is treated as a string

```
void main() {
EXEC SQL BEGIN DECLARE SECTION;
  string query = "SELECT * FROM T WHERE Name = ?;";
  string name;
EXEC SQL END DECLARE SECTION;

EXEC SQL CONNECT TO @localhost USER john;

cin >> name;  // user input

EXEC SQL PREPARE stmt FROM :query;

EXEC SQL EXECUTE stmt USING :name;

EXEC SQL DISCONNECT;

}
```

Generated Query

```
SELECT * FROM T
WHERE Name = '\';
DROP TABLE T;
SELECT \'';
```

SQL Injection Attacks

- **How to Protect?**
  - Use stored function/procedure
  - **Why?**
    - At runtime, anything in `name` is treated as a string

```
CREATE OR REPLACE FUNCTION queryT (IN name TEXT)
RETURNS SETOF T AS $$
  SELECT * FROM T WHERE Name = name;
$$ LANGUAGE sql;
```

- Advantages
  - Code reuse
  - Ease of maintenance
  - Performance
  - Security

QUESTION?

Instructor: Jiang Kan

Slides adapted from Prof Adi Yoga Sidi Prabawa