CS2102 Database Systems

Intro

- Xiaokui Xiao (Cedric)
- xkxiao@nus.edu.sg
- COM1, #03-25

Announcements (March 12)

- Submission of revised answers for midterm test's questions 8, 9 & 10
  - Please refer to LumiNUS announcement on submission of revised answers to fix minor typo errors
- The deadline for this is **today (March 12, 11pm)**

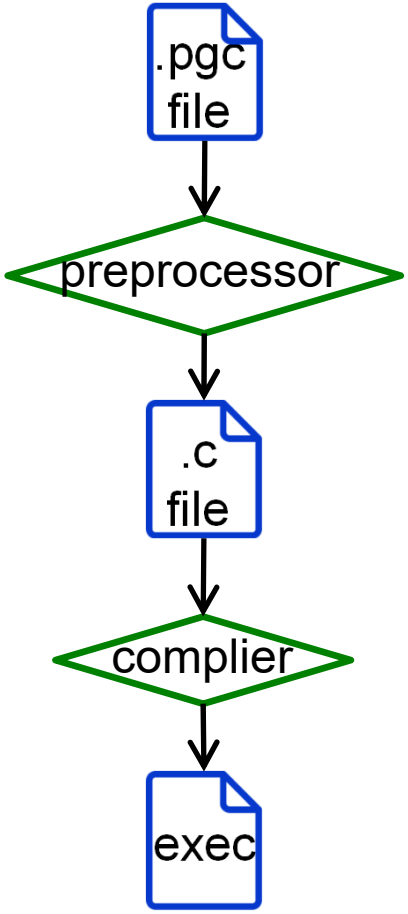
- Assignment 2 on SQL is due on March 19 (Friday, 6pm)
- Late submission penalty: One mark will be deducted for each late day up to two late days; submissions after the second late day will receive zero marks and will not be graded.

Programming with SQL

- The previous lectures have discussed how to use SQL to access a database
- But what if we are asked to write a C/Java/… program that accesses a database?
- We have several options:
  - Mixing C/Java/… and SQL
  - Write a C/Java/… program that calls some API to run SQL queries

Mixing a host language with SQL

- Basic idea:
  - Write a program that mixes a host language with SQL
  - Pass it to a preprocessor, which returns a program in the host language
  - Compile the program to executable code
- In PostgreSQL:



Example

```
int main() {
        EXEC SQL BEGIN DECLARE SECTION;
                char supplName[30], prodName[20];
                float price;
        EXEC SQL END DECLARE SECTION;

        EXEC SQL CONNECT TO @localhost USER john;

        // some code that assigns values to
        // supplName, prodName, and price;

        EXEC SQL INSERT INTO
                Sells(supplName, prodName, price)
                VALUES(:supplName, :prodName, :price);
        EXEC SQL DISCONNECT;
        return 0;
}
```

Example

```
int main() {
        EXEC SQL BEGIN DECLARE SECTION;
                char supplName[30], prodName[20];
                float price;
        EXEC SQL END DECLARE SECTION;

        EXEC SQL CONNECT TO @localhost USER john;

        // some code that assigns values to
        // supplName, prodName, and price;

        EXEC SQL INSERT INTO
                Sells(supplName, prodName, price)
                VALUES(:supplName, :prodName, :price);
        EXEC SQL DISCONNECT;
        return 0;
}
```

This part declares the shared variables that can be used by both the host language statements and SQL statements

Example

```
int main() {
        EXEC SQL BEGIN DECLARE SECTION;
                char supplName[30], prodName[20];
                float price;
        EXEC SQL END DECLARE SECTION;

        EXEC SQL CONNECT TO @localhost USER john;

        // some code that assigns values to
        // supplName, prodName, and price;

        EXEC SQL INSERT INTO
                Sells(supplName, prodName, price)
                VALUES(:supplName, :prodName, :price);
        EXEC SQL DISCONNECT;
        return 0;
}
```


This specifies the database to connect and the username to use

Example

```
int main() {
        EXEC SQL BEGIN DECLARE SECTION;
                char supplName[30], prodName[20];
                float price;
        EXEC SQL END DECLARE SECTION;

        EXEC SQL CONNECT TO @localhost USER john;

        // some code that assigns values to
        // supplName, prodName, and price;

        EXEC SQL INSERT INTO
                Sells(supplName, prodName, price)
                VALUES(:supplName, :prodName, :price);
        EXEC SQL DISCONNECT;
        return 0;
}
```

This part is in the host language

Example

```
int main() {
        EXEC SQL BEGIN DECLARE SECTION;
                char supplName[30], prodName[20];
                float price;
        EXEC SQL END DECLARE SECTION;

        EXEC SQL CONNECT TO @localhost USER john;

        // some code that assigns values to
        // supplName, prodName, and price;

        EXEC SQL INSERT INTO
                Sells(supplName, prodName, price)
                VALUES(:supplName, :prodName, :price);
        EXEC SQL DISCONNECT;
        return 0;
}
```

This part inserts a record into the Sells table.

Note that the each shared variable must be preceded by a colon

Example

```
int main() {
        EXEC SQL BEGIN DECLARE SECTION;
                char supplName[30], prodName[20];
                float price;
        EXEC SQL END DECLARE SECTION;

        EXEC SQL CONNECT TO @localhost USER john;

        // some code that assigns values to
        // supplName, prodName, and price;

        EXEC SQL INSERT INTO
                Sells(supplName, prodName, price)
                VALUES(:supplName, :prodName, :price);
        EXEC SQL DISCONNECT;
        return 0;
}
```

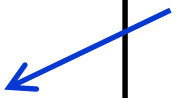
This closes the database connection.

Example

```
int main() {
        EXEC SQL BEGIN DECLARE SECTION;
                char supplName[30], prodName[20];
                float price;
        EXEC SQL END DECLARE SECTION;

        EXEC SQL CONNECT TO @localhost USER john;

        // some code that assigns values to
        // supplName, prodName, and price;

        EXEC SQL INSERT INTO
                Sells(supplName, prodName, price)
                VALUES(:supplName, :prodName, :price);
        EXEC SQL DISCONNECT;
        return 0;
}
```

Example

```
int main() {
        EXEC SQL BEGIN DECLARE SECTION;
                char supplName[30], prodName[20];
                float price;
        EXEC SQL END DECLARE SECTION;

        EXEC SQL CONNECT TO @localhost USER john;

        // some code that assigns values to
        // supplName, prodName, and price;

        EXEC SQL INSERT INTO
                Sells(supplName, prodName, price)
                VALUES(:supplName, :prodName, :price);
        EXEC SQL DISCONNECT;
        return 0;
}
```



In this example, the SQL statement is "fixed" beforehand.

This is referred to as static SQL.

The alternative is dynamic SQL.

Dynamic SQL

- Basic idea:
  - The SQL statement to be executed is not fixed in advance
  - Instead, the statement is stored in a string variable, and it is compiled and executed at runtime
  - i.e., syntax checking happens at runtime

Dynamic SQL Example

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

        EXEC SQL CONNECT TO @localhost USER john;

        // some code that writes a SQL statement
        // in stmt;

        EXEC SQL EXECUTE IMMEDIATE :stmt;

        EXEC SQL DISCONNECT;
        return 0;
}
```

This part executes the SQL statement stored in stmt.

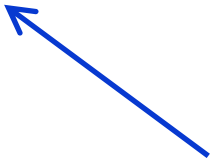
Example: "SELECT * FROM Sells;"

Dynamic SQL Example 2

```
int main() {
        EXEC SQL BEGIN DECLARE SECTION;
                const char *stmt = "INSERT INTO test VALUES(?, ?);";
        EXEC SQL END DECLARE SECTION;

        EXEC SQL CONNECT TO @localhost USER john;

        EXEC SQL PREPARE mystmt FROM :stmt;

        EXEC SQL EXECUTE mystmt USING 42, 'foobar';

        EXEC SQL DEALLOCATE PREPARE mystmt;

        EXEC SQL DISCONNECT;
        return 0;
}
```

Dynamic SQL Example 2

```
int main() {
        EXEC SQL BEGIN DECLARE SECTION;
                const char *stmt = "INSERT INTO test VALUES(?, ?);";
        EXEC SQL END DECLARE SECTION;

        EXEC SQL CONNECT TO @localhost USER john;

        EXEC SQL PREPARE mystmt FROM :stmt;

        EXEC SQL EXECUTE mystmt USING 42, 'foobar';

        EXEC SQL DEALLOCATE PREPARE mystmt;

        EXEC SQL DISCONNECT;
        return 0;
}
```



This part initializes a SQL statement with two parameters

Dynamic SQL Example 2

```
int main() {
        EXEC SQL BEGIN DECLARE SECTION;
                const char *stmt = "INSERT INTO test VALUES(?, ?);";
        EXEC SQL END DECLARE SECTION;

        EXEC SQL CONNECT TO @localhost USER john;

        EXEC SQL PREPARE mystmt FROM :stmt;

        EXEC SQL EXECUTE mystmt USING 42, 'foobar';

        EXEC SQL DEALLOCATE PREPARE mystmt;

        EXEC SQL DISCONNECT;
        return 0;
}
```

This part connects to the database

Dynamic SQL Example 2

```
int main() {
        EXEC SQL BEGIN DECLARE SECTION;
                const char *stmt = "INSERT INTO test VALUES(?, ?);";
        EXEC SQL END DECLARE SECTION;

        EXEC SQL CONNECT TO @localhost USER john;

        EXEC SQL PREPARE mystmt FROM :stmt;

        EXEC SQL EXECUTE mystmt USING 42, 'foobar';

        EXEC SQL DEALLOCATE PREPARE mystmt;

        EXEC SQL DISCONNECT;
        return 0;
}
```

This part parses the SQL statement, and gives it a name, mystmt

Dynamic SQL Example 2

```
int main() {
        EXEC SQL BEGIN DECLARE SECTION;
                const char *stmt = "INSERT INTO test VALUES(?, ?);";
        EXEC SQL END DECLARE SECTION;

        EXEC SQL CONNECT TO @localhost USER john;

        EXEC SQL PREPARE mystmt FROM :stmt;

        EXEC SQL EXECUTE mystmt USING 42, 'foobar';

        EXEC SQL DEALLOCATE PREPARE mystmt;

        EXEC SQL DISCONNECT;
        return 0;
}
```

This part executes the SQL statement, and using 42 and 'foobar' as parameters

Dynamic SQL Example 2

```
int main() {
        EXEC SQL BEGIN DECLARE SECTION;
                const char *stmt = "INSERT INTO test VALUES(?, ?);";
        EXEC SQL END DECLARE SECTION;

        EXEC SQL CONNECT TO @localhost USER john;

        EXEC SQL PREPARE mystmt FROM :stmt;

        EXEC SQL EXECUTE mystmt USING 42, 'foobar';

        EXEC SQL DEALLOCATE PREPARE mystmt;

        EXEC SQL DISCONNECT;
        return 0;
}
```

This part releases the resources allocated to mystmt

Every prepared statement should be deallocated when it is no longer needed

Dynamic SQL Example 2

```
int main() {
        EXEC SQL BEGIN DECLARE SECTION;
                const char *stmt = "INSERT INTO test VALUES(?, ?);";
        EXEC SQL END DECLARE SECTION;

        EXEC SQL CONNECT TO @localhost USER john;

        EXEC SQL PREPARE mystmt FROM :stmt;

        EXEC SQL EXECUTE mystmt USING 42, 'foobar';

        EXEC SQL DEALLOCATE PREPARE mystmt;

        EXEC SQL DISCONNECT;
        return 0;
}
```

Static and Dynamic SQL: Summary

- Static and dynamic SQL allow us to execute SQL statements in a host language
  - Using EXEC SQL …
- They are referred to as statement-level interfaces
- What if we don't want to mix the host language with SQL?
- We can use a call-level interface (CLI)
  - i.e., instead of using EXEC SQL, we call some API to run SQL queries
  - Everything is in the host language

Call-Level Interface (CLI)

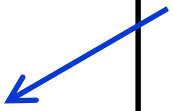
- Examples:
  - Java Database Connectivity (JDBC)
    - https://jdbc.postgresql.org/
  - Open Database Connectivity (ODBC)
    - https://odbc.postgresql.org/

ODBC Example (in C++)

```
void main() {
  char * sql;
  connection C("dbname = testdb user = postgres \
    password = cohondob hostaddr = 127.0.0.1 port = 5432");

  // Some code here

  // Create SQL statement
  sql = "CREATE TABLE COMPANY("  \
    "ID INT PRIMARY KEY     NOT NULL," \
    "NAME           TEXT    NOT NULL);";

  work W(C);      // Create a transactional object

  W.exec( sql );  // Execute SQL query
  W.commit();
  C.disconnect ();
}
```

ODBC Example (in C++)

```
void main() {
  char * sql;
  connection C("dbname = testdb user = postgres \
    password = cohondob hostaddr = 127.0.0.1 port = 5432");

  // Some code here

  // Create SQL statement
  sql = "CREATE TABLE COMPANY("  \
    "ID INT PRIMARY KEY     NOT NULL," \
    "NAME           TEXT    NOT NULL);";

  work W(C);      // Create a transactional object

  W.exec( sql );  // Execute SQL query
  W.commit();
  C.disconnect ();
}
```



This part creates a connection to the database

ODBC Example (in C++)

```
void main() {
   char * sql;
   connection C("dbname = testdb user = postgres \
      password = cohondob hostaddr = 127.0.0.1 port = 5432");

   // Some code here

   // Create SQL statement
   sql = "CREATE TABLE COMPANY("  \
      "ID INT PRIMARY KEY     NOT NULL," \
      "NAME           TEXT    NOT NULL);";

   work W(C);      // Create a transactional object

   W.exec( sql );   // Execute SQL query
   W.commit();
   C.disconnect ();
}
```

This part assigns the SQL statement as a string to the sql variable

ODBC Example (in C++)

```
void main() {
  char * sql;
  connection C("dbname = testdb user = postgres \
    password = cohondob hostaddr = 127.0.0.1 port = 5432");

  // Some code here

  // Create SQL statement
  sql = "CREATE TABLE COMPANY("  \
    "ID INT PRIMARY KEY     NOT NULL," \
    "NAME           TEXT    NOT NULL);";

  work W(C);      // Create a transactional object

  W.exec( sql );   // Execute SQL query
  W.commit();
  C.disconnect ();
}
```

This part creates an objective W for executing the SQL statement

ODBC Example (in C++)

```
void main() {
  char * sql;
  connection C("dbname = testdb user = postgres \
    password = cohondob hostaddr = 127.0.0.1 port = 5432");

  // Some code here

  // Create SQL statement
  sql = "CREATE TABLE COMPANY("  \
    "ID INT PRIMARY KEY     NOT NULL," \
    "NAME           TEXT    NOT NULL);";

  work W(C);      // Create a transactional object

  W.exec( sql );  // Execute SQL query
  W.commit();
  C.disconnect ();
}
```

This part executes the SQL statement recorded in sql

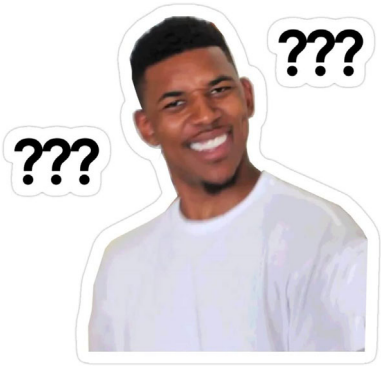
ODBC Example (in C++)

```
void main() {
   char * sql;
   connection C("dbname = testdb user = postgres \
      password = cohondob hostaddr = 127.0.0.1 port = 5432");

   // Some code here

   // Create SQL statement
   sql = "CREATE TABLE COMPANY("  \
      "ID INT PRIMARY KEY     NOT NULL," \
      "NAME           TEXT    NOT NULL);";

   work W(C);      // Create a transactional object

   W.exec( sql );   // Execute SQL query
   W.commit();
   C.disconnect ();
}
```

This part closes the database connection

Coming Next

- Statement-level interface
  - Code in a mix of a host language and SQL
- Call-level interface
  - Code in the host language only
- What if we want to code in SQL only?
  - "Why would you do that?"
  - To create functions/procedures in the database



Functions/procedures in SQL

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

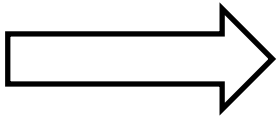
SQL-based Procedure Language

- ISO standard: SQL/PSM (Persistent Stored Modules)
- Different vendors' implementations:
  - Oracle: PL/SQL
  - PostgreSQL: PL/pgSQL
  - SQL Server: TransactSQL
  - …
- Our discussions will be based on PL/pgSQL

PL/pgSQL Example

Scores

Name	Mark
Alice	92
Bob	63
Cathy	58
David	47



Name	Grade
Alice	A
Bob	B
Cathy	C
David	D

- Suppose that we want to create a function that converts numeric marks to letter grades as follows:
  - [70, 100] -> A
  - [60, 70) -> B
  - [50, 60) -> C
  - [0, 50) -> D

PL/pgSQL Example

Scores

Name	Mark
Alice	92
Bob	63
Cathy	58
David	47

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

PL/pgSQL Example

Scores

Name	Mark
Alice	92
Bob	63
Cathy	58
David	47

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

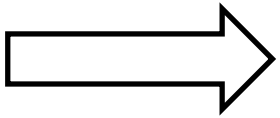
- Query: SELECT convert(90);
- Result: A

- Query: SELECT convert(40);
- Result: D

PL/pgSQL Example

Scores

Name	Mark
Alice	92
Bob	63
Cathy	58
David	47



Name	Grade
Alice	A
Bob	B
Cathy	C
David	D

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

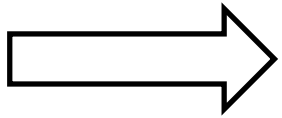
```
SELECT Name, convert( Mark )
FROM Scores;
```

This generates the results above

PL/pgSQL Example

Scores

Name	Mark
Alice	92
Bob	63
Cathy	58
David	47



Name
Bob

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

```
SELECT Name
FROM   Scores
WHERE convert( Mark ) = 'B';
```

This generates the results above

PL/pgSQL Example

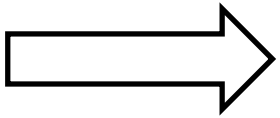
- The function here returns one character
- There are other options:
  - Return an existing tuple or tuple set
  - Return a new tuple or tuple set

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

Returning an existing tuple

Scores

Name	Mark
Alice	92
Bob	63
Cathy	58
David	47



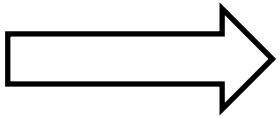
Name	Mark
Alice	92

- Create a function topStudent() that returns any one tuple in Scores with the highest mark
- That is, SELECT topStudent();
  should return Alice's record

Returning an existing tuple

Scores

Name	Mark
Alice	92
Bob	63
Cathy	58
David	47



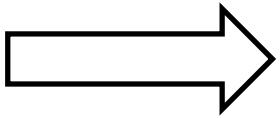
Name	Mark
Alice	92

```
CREATE OR REPLACE FUNCTION topStudent ()
RETURNS Scores AS $$
        SELECT      *
        FROM        Scores
        ORDER BY    Mark  DESC LIMIT 1;
$$ LANGUAGE sql;
```

Returning an existing set of tuples

Scores

Name	Mark
Alice	92
Bob	63
Cathy	58
David	92



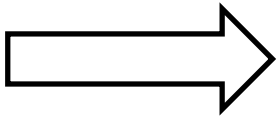
Name	Mark
Alice	92
David	92

- Create a function topStudents() that returns all tuples in Scores with the highest mark
- That is, SELECT * FROM topStudents(); should return Alice and David's records

Returning an existing set of tuples

Scores

Name	Mark
Alice	92
Bob	63
Cathy	58
David	92



Name	Mark
Alice	92
David	92

```
CREATE OR REPLACE FUNCTION topStudents ()
RETURNS SETOF Scores AS $$
        SELECT      *
        FROM        Scores
        WHERE       Mark =
                (SELECT MAX(Mark) FROM Scores);
$$ LANGUAGE sql;
```

Returning existing tuples

```
CREATE OR REPLACE FUNCTION topStudent ()
RETURNS Scores AS $$
        SELECT    *
        FROM      Scores
        ORDER BY Mark      DESC LIMIT 1;
$$ LANGUAGE sql;
```

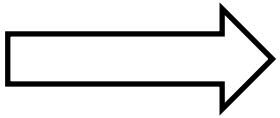
```
CREATE OR REPLACE FUNCTION topStudents ()
RETURNS SETOF Scores AS $$
        SELECT    *
        FROM      Scores
        WHERE     Mark =
                  (SELECT MAX(Mark) FROM Scores);
$$ LANGUAGE sql;
```

- The above two examples return tuples from the Scores table
- What if we want to return some other values?

Returning a new tuple

Scores

Name	Mark
Alice	92
Bob	63
Cathy	58
David	92



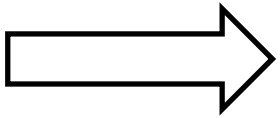
TopMark	Cnt
92	2

- Create a function topMarkCnt that returns the highest mark and its number of occurrences
- That is, SELECT topMarkCnt();
  should return the results above

Returning a new tuple

Scores

Name	Mark
Alice	92
Bob	63
Cathy	58
David	92



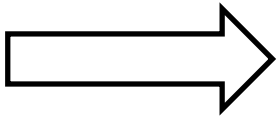
TopMark	Cnt
92	2

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

Returning a new set of tuples

Scores

Name	Mark
Alice	92
Bob	63
Cathy	58
David	92



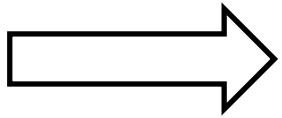
Mark	Cnt
92	2
63	1
58	1

- Create a function MarkCnt that returns each distinct Mark and its number of occurrences
- That is, SELECT MarkCnt();
  should return the results above

Returning a new set of tuples

Scores

Name	Mark
Alice	92
Bob	63
Cathy	58
David	92



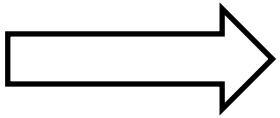
Mark	Cnt
92	2
63	1
58	1

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

Returning a new set of tuples 2

Scores

Name	Mark
Alice	92
Bob	63
Cathy	58
David	92



Mark	Cnt
92	2
63	1
58	1

```
CREATE OR REPLACE FUNCTION MarkCnt ()
RETURNS TABLE(Mark INT, Cnt INT) AS $$
        SELECT Mark, COUNT(*)
        FROM   Scores
        GROUP BY    Mark ;
$$ LANGUAGE sql;
```

SQL Functions vs. Procedures

- The above examples illustrate SQL functions
  - Each of which returns some values/tuples
- If no return value/tuple is needed, we may use SQL procedures instead

SQL Procedure Example

```
CREATE OR REPLACE PROCEDURE transfer (fromAcct TEXT,
                                      toAcct TEXT, toAcct TEXT, amount INT)
AS $$
        UPDATE Accounts
        SET balance = balance - amount
        WHERE name = fromAcct;


        UPDATE Accounts
        SET balance = balance + amount
        WHERE name = toAcct;
$$ LANGUAGE SQL
```

- To execute this procedure:
- CALL transfer('Alice', 'Bob', 100);

SQL with control structures

- The previous SQL functions/procedures are relatively simple
  - Each of them just sequentially executes some SQL statements
- But in general, SQL functions/procedures can be complex, having variables and control structures:
  - IF … THEN … ELSE … ENF IF
  - LOOP … END LOOP
  - EXIT … WHEN …
  - WHILE … LOOP … END LOOP
  - FOR … IN … LOOP … END LOOP
  - …

Example: Using variables

- The following function swaps two input integers

```
CREATE OR REPLACE FUNCTION swap (INOUT val1 INT, INOUT val2 INT)
RETURNS RECORD AS $$
DECLARE
        temp_val INTEGER;
BEGIN
        temp_val := val1;
        val1 := val2;
        val2 := temp_val;
END;
$$ LANGUAGE plpgsql;
```

- Query: SELECT swap(11, 22);
- Result: (22, 11)

Example: IF … THEN … END IF

- The following function sorts two integers

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

- SELECT swap(99, 11); → (11, 99)
- SELECT swap(11, 99); → (11, 99)

Example: LOOP … END LOOP

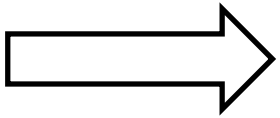
- The following function computes $1+2+\ldots+x$

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

Question

Scores

Name	Mark
Alice	92
Bob	63
Cathy	58
David	47



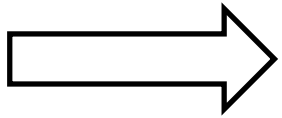
Name	Mark	Gap
Alice	92	NULL
Bob	63	29
Cathy	58	5
David	47	11

- Write a function for the following task:
  - Sort the students in Scores in descending order of their Marks
  - For each student, compute the different between his/her mark and the previous student

Question

Scores

Name	Mark
Alice	92
Bob	63
Cathy	58
David	47

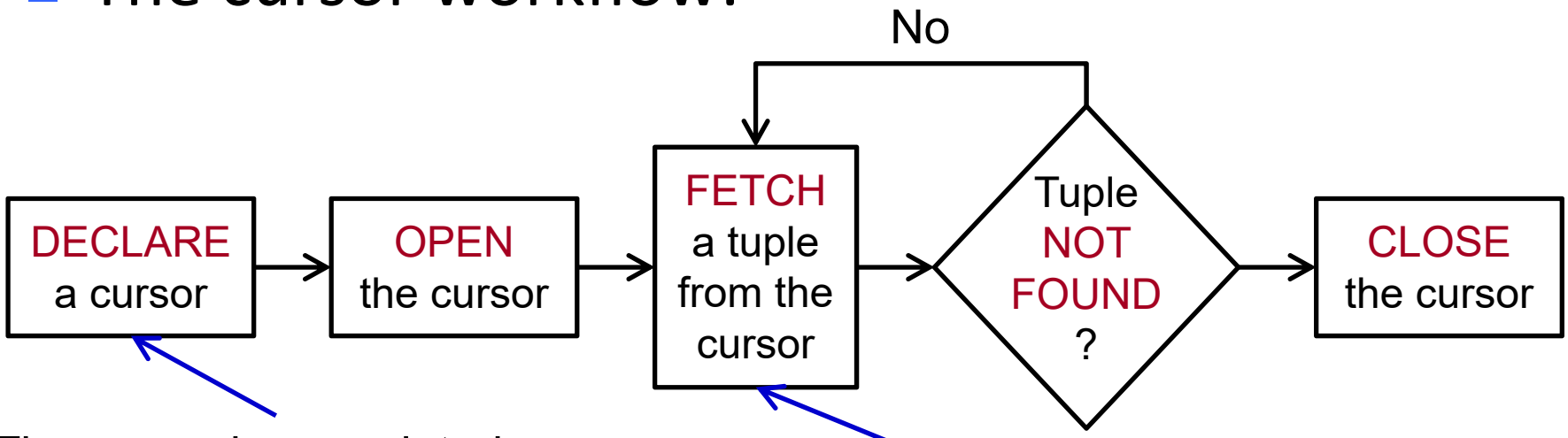


Name	Mark	Gap
Alice	92	NULL
Bob	63	29
Cathy	58	5
David	47	11

- Idea:
  - SELECT * FROM Scores ORDER BY Mark DESC;
  - Loop over the sorted sequence of students
    - For each student, computes the score difference with the previous student
- Problem: How we do loop over the sorted sequence?
- Answer: Use a cursor

Cursor

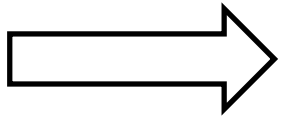
- A cursor enables us to access each individual row returned by a SELECT statement
- The cursor workflow:



Cursor Example

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:
  - Sort the students in Scores in descending order of their Marks
  - For each student, compute the different between his/her mark and the previous student

Cursor Example

Scores

Name	Mark
Alice	92
Bob	63
Cathy	58
David	47

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

Cursor Example

Scores

Name	Mark
Alice	92
Bob	63
Cathy	58
David	47

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

This part declares a cursor variable

Cursor Example

Scores

Name	Mark
Alice	92
Bob	63
Cathy	58
David	47

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

This executes the SQL statement, and let curs point to the beginning of the result

Cursor Example

Scores

Name	Mark
Alice	92
Bob	63
Cathy	58
David	47

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

Read the next tuple from curs, and put it into r.

Cursor Example

Scores

Name	Mark
Alice	92
Bob	63
Cathy	58
David	47

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

If the FETCH operation did not get any tuple, terminate the loop

Cursor Example

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

Scores

Name	Mark
Alice	92
Bob	63
Cathy	58
David	47

If the FETCH operation got a tuple, then we decide the values of name, mark, and gap.

Cursor Example

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

Scores

Name	Mark
Alice	92
Bob	63
Cathy	58
David	47

This inserts a tuple (name, mark gap) to the output of the function.

Cursor Example

Scores

Name	Mark
Alice	92
Bob	63
Cathy	58
David	47

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

This updates the prv_mark variable.

Cursor Example

Scores

Name	Mark
Alice	92
Bob	63
Cathy	58
David	47

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

This releases the resources allocated to curs.

Cursor Movement

- In the previous example, we have seen
  FETCH curs INTO r
- There are other variants of FETCH
- FETCH PRIOR FROM cur INTO r
  - Fetch the prior row
- FETCH FIRST FROM cur INTO r
- FETCH LAST FROM cur INTO r
- FETCH ABSOLUTE 3 FROM cur INTO r
  - Fetch the 3-rd tuple
- …

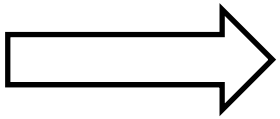
SQL/PSM Summary

- An extension of SQL with a procedural language
- Enables us to create complex functions/procedures in databases
  - as well as triggers (will be covered next week)
- For additional details about PL/pgSQL, please refer to
  - https://www.postgresql.org/docs/current/plpgsql.html

Exercise

Scores

Name	Mark
Alice	92
Bob	63
Cathy	58
David	47



Name	Mark
Bob	63
Cathy	58

- Write a function that retrieves the student(s) with the median mark(s):
- What we mean by "median"
  - Let n be the total number of students
  - If n is odd, then the median is the ((n+1)/2)-th
  - If n is even, then the medians are the (n/2)-th and the (n/2+1)-th
  - Ties are broken arbitrarily

```
CREATE OR REPLACE FUNCTION median_stu()
RETURNS TABLE ( name TEXT, mark INT ) AS $$
DECLARE
        curs CURSOR FOR (SELECT * FROM Scores ORDER BY Mark DESC);
        r RECORD;
        n INT;
BEGIN
        OPEN curs;
        SELECT COUNT(*) INTO n FROM Scores;
        IF n % 2 = 1 THEN
                FETCH ABSOLUTE (n+1)/2 FROM curs INTO r;
                name := r.name;
                mark := r.Mark;
                RETURN NEXT;
        ELSE
                FETCH ABSOLUTE n/2 FROM curs INTO r;
                name := r.name;
                mark := r.Mark;
                RETURN NEXT;
                FETCH ABSOLUTE n/2 + 1 FROM curs INTO r;
                name := r.name;
                mark := r.Mark;
                RETURN NEXT;
        END IF;
        CLOSE curs;
END;
$$ LANGUAGE plpgsql;
```

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

Coming Next

- SQL Injection Attacks

SQL Injection Attacks

- A type of attacks on dynamic SQL
- Example:
  - Suppose that we have a user-provided parameter *input_name*
  - We construct a string stmt by concatenating three sub-strings:
    - SELECT * FROM T WHERE Name = '
    - *input_name*
    - ';
  - And we have the following dynamic SQL:
    - EXEC SQL EXECUTE IMMEDIATE :stmt;
  - For instance, if *input_name* = "Alice", then the executed query is:
    - SELECT * FROM T WHERE Name = 'Alice';

SQL Injection Attacks

- stmt is the concatenation of three sub-strings:
  - SELECT * FROM T WHERE Name = '
  - *input_name*
  - ';
- EXEC SQL EXECUTE IMMEDIATE :stmt;

- A malicious user sets *input_name* to the following:
  a'; DROP TABLE T; SELECT 'a
- What would be the SQL statement in stmt?
- SELECT * FROM T WHERE Name = 'a'; DROP TABLE T; SELECT 'a';

- T will be deleted once stmt is executed

SQL Injection Attacks

- A malicious user sets *input_name* to the following:
  a'; DROP TABLE T; SELECT 'a
- What would be the SQL statement in stmt?
- SELECT * FROM T WHERE Name = 'a'; DROP TABLE T; SELECT 'a';

- Main issue:
  - *input_name* is supposed to be a name
  - But the malicious user hides a DROP TABLE query in it
  - The database executes it as it is
- Solution: Let the database know what it is supposed to execute

Solution: Use function/procedures

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

- Rationale:
  - The function is compiled and stored in the database
  - At runtime, anything in in_name would be treated as a string constant
  - Even if in_name is set to a'; DROP TABLE T; SELECT 'a

Functions/procedures in SQL

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

Solution: use prepared statements

```
EXEC SQL BEGIN DECLARE SECTION;
        const char *stmt = "SELECT * FROM T WHERE Name = ?;";
        char in_name[100];
EXEC SQL END DECLARE SECTION;

// get the user-provided name and put it into in_name;

EXEC SQL PREPARE mystmt FROM :stmt;

EXEC SQL EXECUTE mystmt USING :in_name;
```

- Rationale:
  - The SQL statement is compiled when it is prepared
  - Anything in in_name is treated a string constant