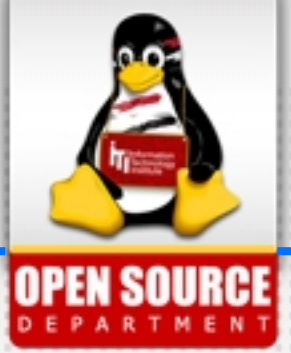
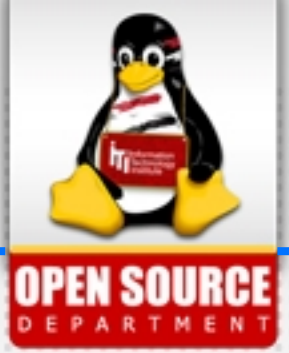
Day 4 Contents



- Inheritance
- PL/pgSQL Functions
- Triggers
- Backup and Restore

Arrays data type



- PostgreSQL allows columns of a table to be defined as variable-length multidimensional arrays

```
CREATE TABLE sal_emp (
    name            text,
    pay_by_quarter  integer[4],
    schedule        text[][]);
```

```
SELECT * FROM sal_emp;
 name  |      pay_by_quarter       |                    schedule
-------+---------------------------+-----------------------------------------
 Bill  | {10000,10000,10000,10000} | {{meeting,lunch},{training,presentation}}
 Carol | {20000,25000,25000,25000} | {{breakfast,consulting},{meeting,lunch}}
(2 rows)
```

Arrays data type

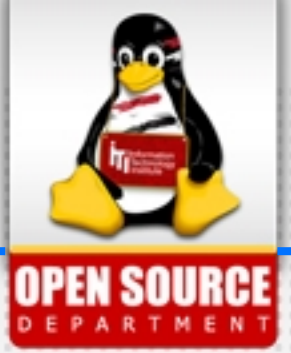
- To write an array value as a literal constant, enclose the element values within curly braces and separate them by commas. as

```
'{ val1 , val2 , ... }‘
```

Ex:

```
INSERT INTO sal_emp (pay_by_quarter)
VALUES ('{10000, 10000, 10000, 10000}’);
```

Arrays data type



You can access array elements as in programming, retrieves the third quarter pay of all employees

```
SELECT pay_by_quarter[3] FROM sal_emp;
```

Range types

- Range types are data types representing a range of values of some element type.
- PostgreSQL comes with the following built-in range types:

int4range — Range of integer

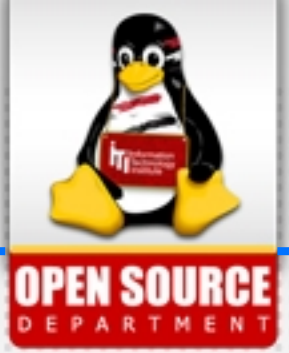
int8range — Range of bigint

numrange — Range of numeric

tsrange — Range of timestamp

daterange — Range of date

Range types



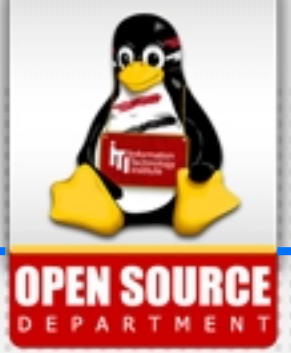
- ranges of timestamp might be used to represent the ranges of time that a call is started and ended.

```
CREATE TABLE calls (id int, during tsrange);
INSERT INTO calls VALUES
    (1108, '[2017-01-01 14:30, 2017-01-01 15:30]');
```

- In addition, you can define your own range type using CREATE TYPE.

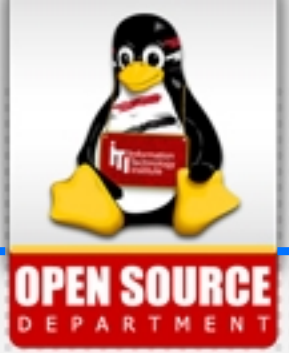
```
CREATE TYPE floatrange AS RANGE (
SUBTYPE = float);
```

Inheritance



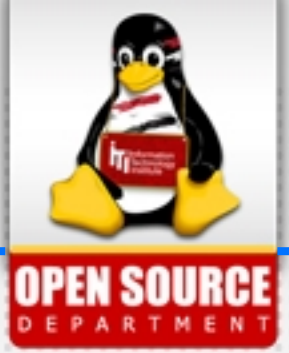
- Inheritance is one of the foundations of the object-oriented programming paradigm.
- Using inheritance, you can define a hierarchy of related tables
- Each layer in the inheritance hierarchy represents a specialization of the layer above it
- In PostgreSQL, a table can inherit from zero or more other tables..

Inheritance




```
CREATE TABLE video(
    video_id      int,
    Title         text,
    Duration      interval);

CREATE TABLE dvds(
    audio_tracks text[]
) INHERITS ( video );
```

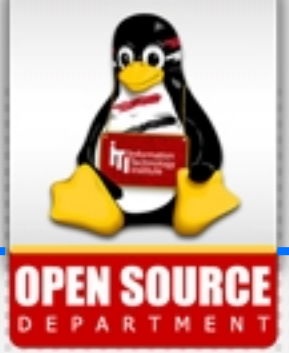
Inheritance




```
testdb4=# insert into dvds values(1,'starts wars','54 minutes','{arabic,english}');
INSERT 0 1
testdb4=# select * from dvds ;
 video_id |    title    | duration |   audio_tracks
----------+-------------+----------+------------------
        1 | starts wars | 00:54:00 | {arabic,english}
(1 row)
```

```
testdb4=# select * from video ;
 video_id |      title       | duration
----------+------------------+----------
        1 | starts wars      | 00:54:00
(1 row)
```

Inheritance



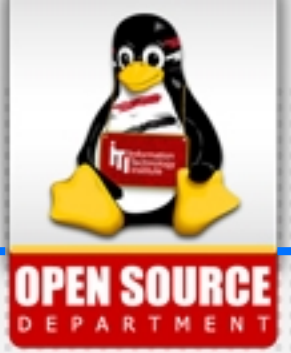
To select only Videos:

```
SELECT * FROM ONLY video;
```

To select Videos and dvds:

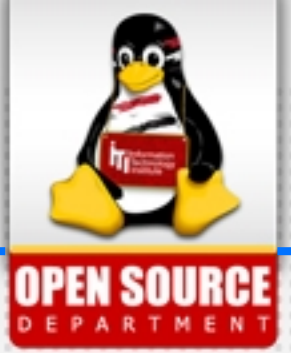
```
SELECT * FROM video;
```

PL/pgSQL



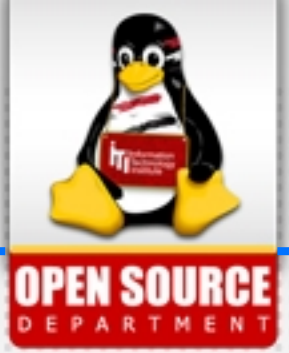
- PL/pgSQL is a procedural language for the PostgreSQL database system.
- The design goals of PL/pgSQL were to create a procedural language that
  - can be used to create functions and trigger procedures,
  - can perform complex computations,
  - is easy to use.

Function Declaration



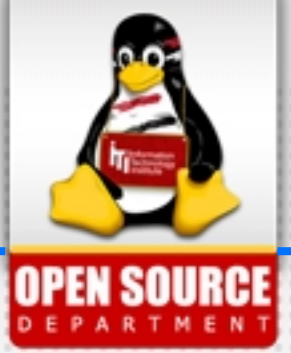

```
CREATE [OR REPLACE] FUNCTION function_name (arguments)
RETURNS return_datatype AS $$
DECLARE
    declaration;
BEGIN
    < function_body >
    RETURN { variable_name | value }
END;
$$LANGUAGE plpgsql;
```

Function Declaration



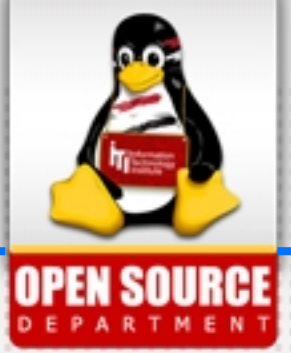

```
CREATE FUNCTION somefunc() RETURNS integer AS $$
DECLARE
    quantity integer = 30;
BEGIN
    quantity = quantity + 50;
    RETURN quantity;
END;
$$ LANGUAGE plpgsql;
```

Function execution




```
testdb4=# select somefunc();
 somefunc
----------
       80
(1 row)
```

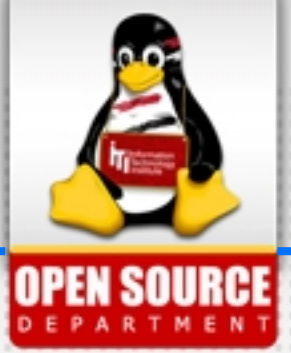
Parameters declaration



- Give the name to the parameter in CREATE FUNCTION statement

```
CREATE FUNCTION somefunc(param1 DATATYPE,param2 DATATYPE)...
```

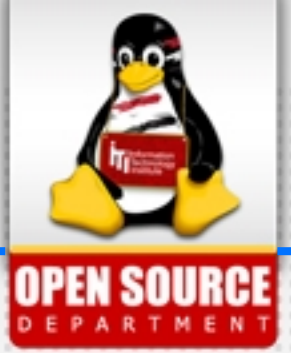
Parameters declaration



- Example

```
CREATE FUNCTION sales_tax(subtotal int) RETURNS float AS $$
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

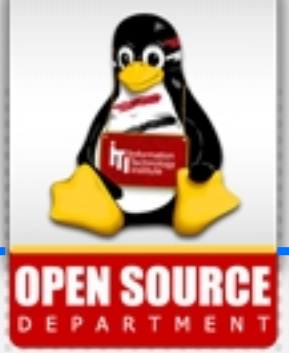
Variables declaration



- All variables used in a block must be declared in the declarations section.
- PL/pgSQL variables can have any SQL data type, such as integer, varchar, and char.

```
name [ CONSTANT ] type [ NOT NULL ][ := expression ];
```

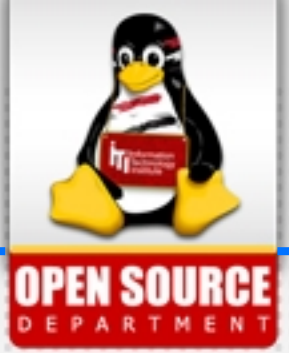
Variables declaration



- Examples

```
user_id integer;

Name varchar;

Name varchar = 'mohamed';

Name varchar := 'mohamed';

quantity numeric(5,2);

user_id CONSTANT integer := 10;

myfield tablename.columnname%TYPE;
```

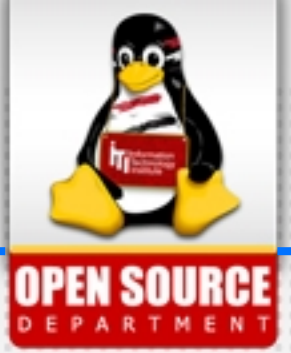
Control Structures



- IF and CASE statements let you execute alternative commands based on certain conditions:

```
IF ... THEN ... END IF;

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

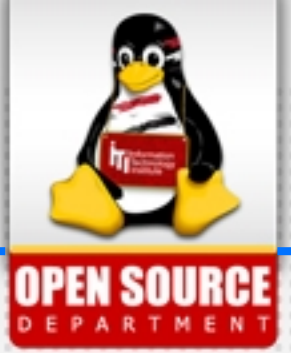
Control Structures



- IF Example:

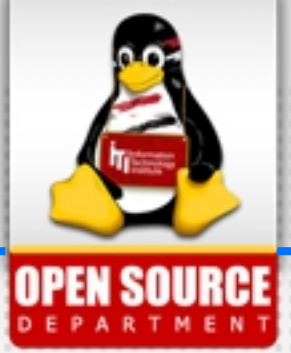
```
IF v_count > 0 THEN
    INSERT INTO users_count (count) VALUES (v_count);
    RETURN 't';
ELSE
    RETURN 'f';
END IF;
```

Control Structures




```
IF number = 0 THEN
    result := 'zero';
ELSIF number > 0 THEN
    result := 'positive';
ELSIF number < 0 THEN
    result := 'negative';
ELSE
    result := 'NULL';
END IF;
```

Control Structures

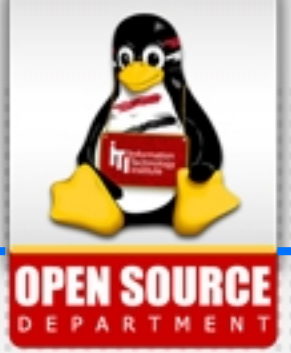



```
CASE ... WHEN ... THEN ... ELSE ... END CASE
```

- Case Example:

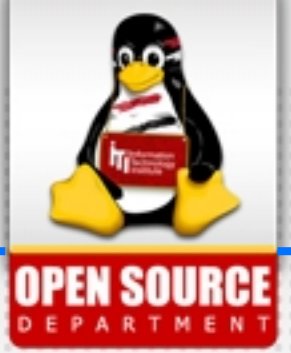
```
CASE x
    WHEN 1 THEN
        msg := 'one';
    ELSE
        msg := 'other value than one';
END CASE;
```

Control Structures




```
WHILE amount_owed > 0 AND gift_certificate_balance > 0
LOOP
    -- some computations here
   IF count > 0 THEN
        EXIT;  -- exit loop
   ELSE
      CONTINUE;
   END IF;
END LOOP;
```

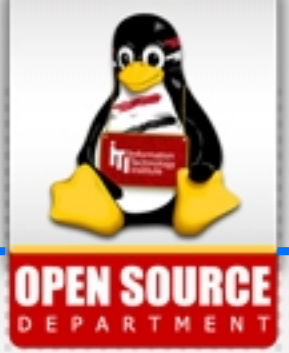
Control Structures



- For:

```
FOR i IN 1..10 LOOP
    -- i will take on the values 1,2,3,4,5,6,7,8,9,10
END LOOP;


FOR i IN REVERSE 10..1 LOOP
    -- i will take on the values 10,9,8,7,6,5,4,3,2,1
END LOOP;
```

Basic Statements



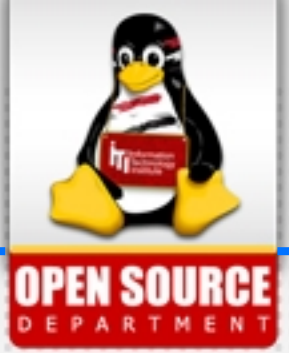
- The result of a SQL command yielding a **single row** (possibly of multiple columns) can be assigned to a record variable, row-type variable:

```
SELECT select_expressions INTO recordVarType FROM ...;
```

- Exmaple:

```
SELECT * INTO myrec FROM emp WHERE ...;
```

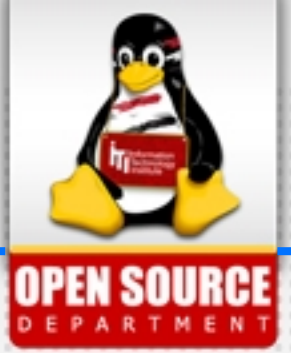
Variables declaration



- A variable of a **Row type** can hold a whole row of a SELECT.

```
CREATE FUNCTION merge_fields(t1_row table1) RETURNS text AS $$
DECLARE
    t2_row table2%ROWTYPE;
BEGIN
    SELECT * INTO t2_row FROM table2 WHERE ... ;
    RETURN concat(t1_row.col1, t2_row.col3, t1_row.col5);
END;$$ LANGUAGE plpgsql;
SELECT merge_fields(t.*) FROM table1 t WHERE ... ;
```

Variables declaration

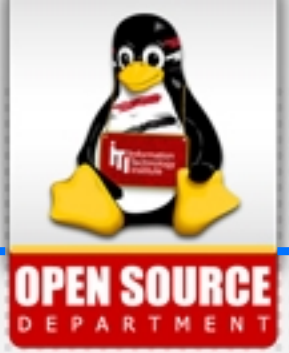


- **Record type** are similar to row-type variables, but they have no predefined structure.

  They take on the actual row structure of the row they are assigned during a SELECT.

```
CREATE FUNCTION merge_fields(t_row table1) RETURNS text AS $$
DECLARE
    t2_row record;
BEGIN
    SELECT col1,col4 INTO t2_row FROM table2 WHERE ... ;
    RETURN concat(t_row.col1, t2_row.col1, t_row.col4);
END;$$ LANGUAGE plpgsql;
```

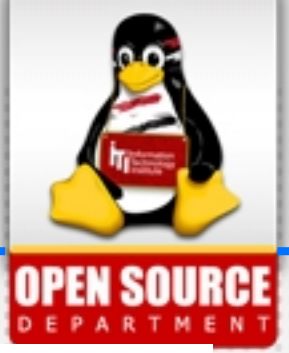
Return



- To return more than one value (as select)

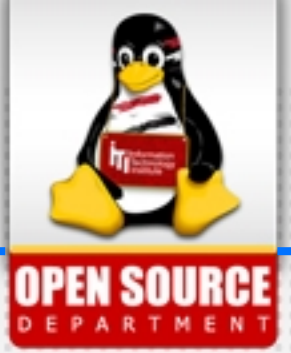
```
CREATE FUNCTION tryquery()
RETURNS TABLE(quantity int, total numeric) AS $$
BEGIN
    RETURN QUERY SELECT quantity, quantity * price FROM
sales WHERE ...;
END;
$$ LANGUAGE plpgsql;
```

Return



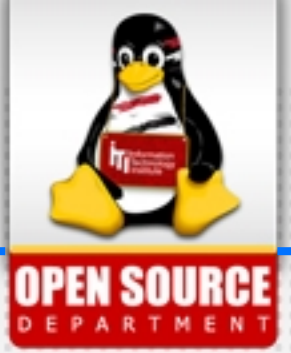

```
testdb4=# select * from  tryQuery();
 quantity | total
----------+--------
        1 |  34234
        4 | 423423
        5 |   3423
(3 rows)

testdb4=# select tryQuery();
  tryquery
-------------
 (1,34234)
 (4,423423)
 (5,3423)
(3 rows)
```

Triggers



- PostgreSQL **Triggers** are database callback functions, which are automatically performed/invoked when a specified database event occurs.
- Database events are (INSERT, UPDATE , DELETE or TRUNCATE), you can choose to run trigger (before or after) the statement.
- A trigger can be called once for every row that the operation modifies, or only executes once for any given operation..

Triggers

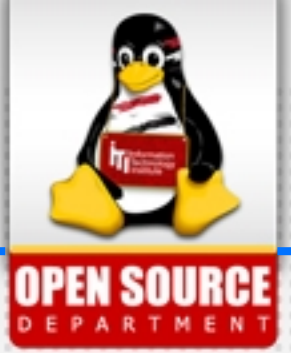


- The basic syntax of creating a trigger is as follows:

```
CREATE  TRIGGER trigger_name {BEFORE|AFTER} event_name
ON table_name FOR EACH { ROW | STATEMENT } EXECUTE
PROCEDURE functionName();
```

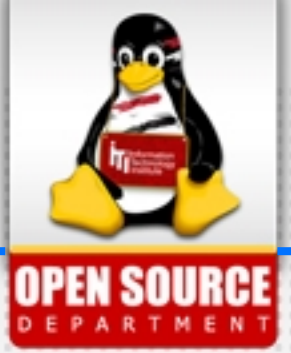
```
CREATE  TRIGGER trigger_name {BEFORE|AFTER} UPDATE OF
column_name ON table_name FOR EACH { ROW | STATEMENT } EXECUTE
PROCEDURE functionName();
```

Triggers



- The trigger function must be defined before the trigger itself can be created.
- The trigger function must be declared as a function taking no arguments and returning type trigger.

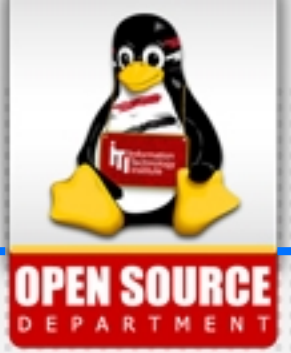
Triggers Special variables



- OLD: it is Data type RECORD; variable holding the old database row for **UPDATE/DELETE** operations in row-level triggers. This variable is NULL in statement-level triggers and for INSERT operations.

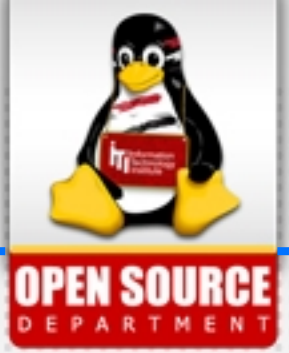
- NEW: it is Data type RECORD; variable holding the new database row for **INSERT/UPDATE** operations in row-level triggers. This variable is NULL in statement-level triggers and for DELETE operations.

Triggers



- A trigger function must return either NULL or a record value having exactly the structure of the table the trigger was fired for.
  - NULL: Row-level triggers fired BEFORE can return null to signal the trigger manager to skip the rest of the operation for this row(The INSERT/UPDATE/DELETE does not occur for this row)
  - NEW: proceed with rest of operation

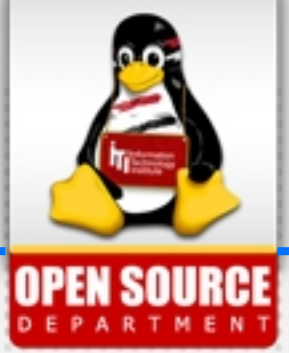
Triggers



- Example:

```
CREATE TABLE COMPANY(
   ID              INT PRIMARY KEY,
   NAME            TEXT,
   AGE             INT,
   ADDRESS         CHAR(50),
   SALARY          REAL
);
```

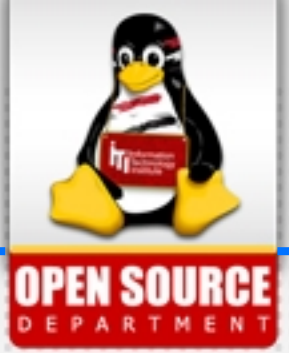
Triggers



- Example:

```
CREATE TABLE LOG(
    EMP_ID INT,
    ENTRY_DATE Date
);
CREATE TRIGGER example_trigger AFTER INSERT ON COMPANY
FOR EACH ROW EXECUTE PROCEDURE logfunc();
```

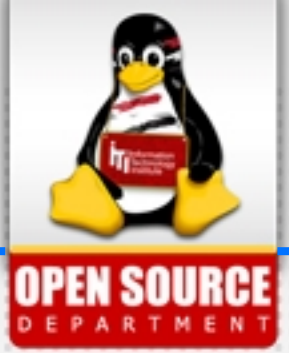
Triggers



- Example:

```
CREATE OR REPLACE FUNCTION logfunc() RETURNS TRIGGER AS $$
    BEGIN
        INSERT INTO Log(EMP_ID, ENTRY_DATE) VALUES
(new.ID, now());
        RETURN NEW;
    END;
$$ LANGUAGE plpgsql;
```

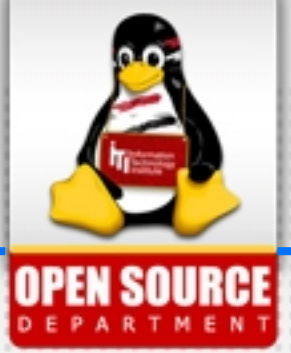
Triggers



- To Drop Trigger

DROP TRIGGER trigger_name;

Backup and Restore



- pg_dump: generate a text file with SQL commands that, when fed back to the server, will recreate the database in the same state as it was at the time of the dump.

- Backup:

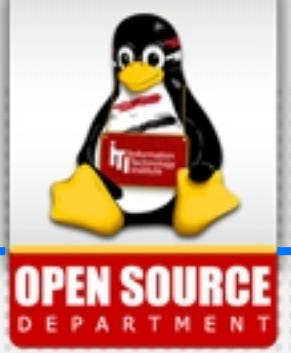
  # pg_dump dbname > outfile dump data and structure

- Restore:

  # psql dbname -f infile

  Beware: dbname must be created before importing.

Backup and Restore



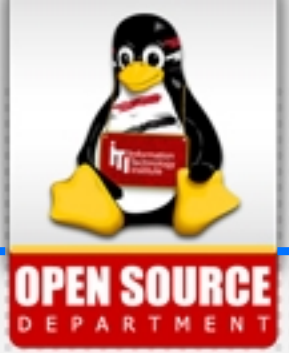
- COPY -- copy data between a file and a table

COPY { table_name | query } TO 'filename'

COPY table_name FROM 'filename'

* Tab separated

Backup and Restore

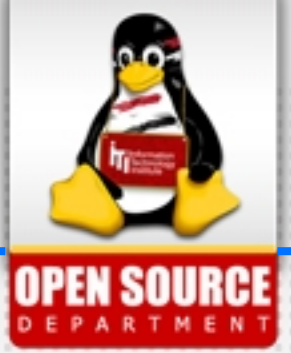


Examples:

- COPY country TO '/home/msabagh/countryTableBk.bk';

- COPY country FROM '/home/msabagh/countryTableBk.bk';

- COPY (SELECT * FROM country WHERE country_name LIKE 'A%') TO '/home/msabagh/a_list_countryTableBk.bk';



Thank You