# Cloud Data Warehouse for PostgreSQL

# Data Ingestion

## Product Documentation

# Contents

# Data Ingestion
# Importing TencentDB Data Offline with DataX

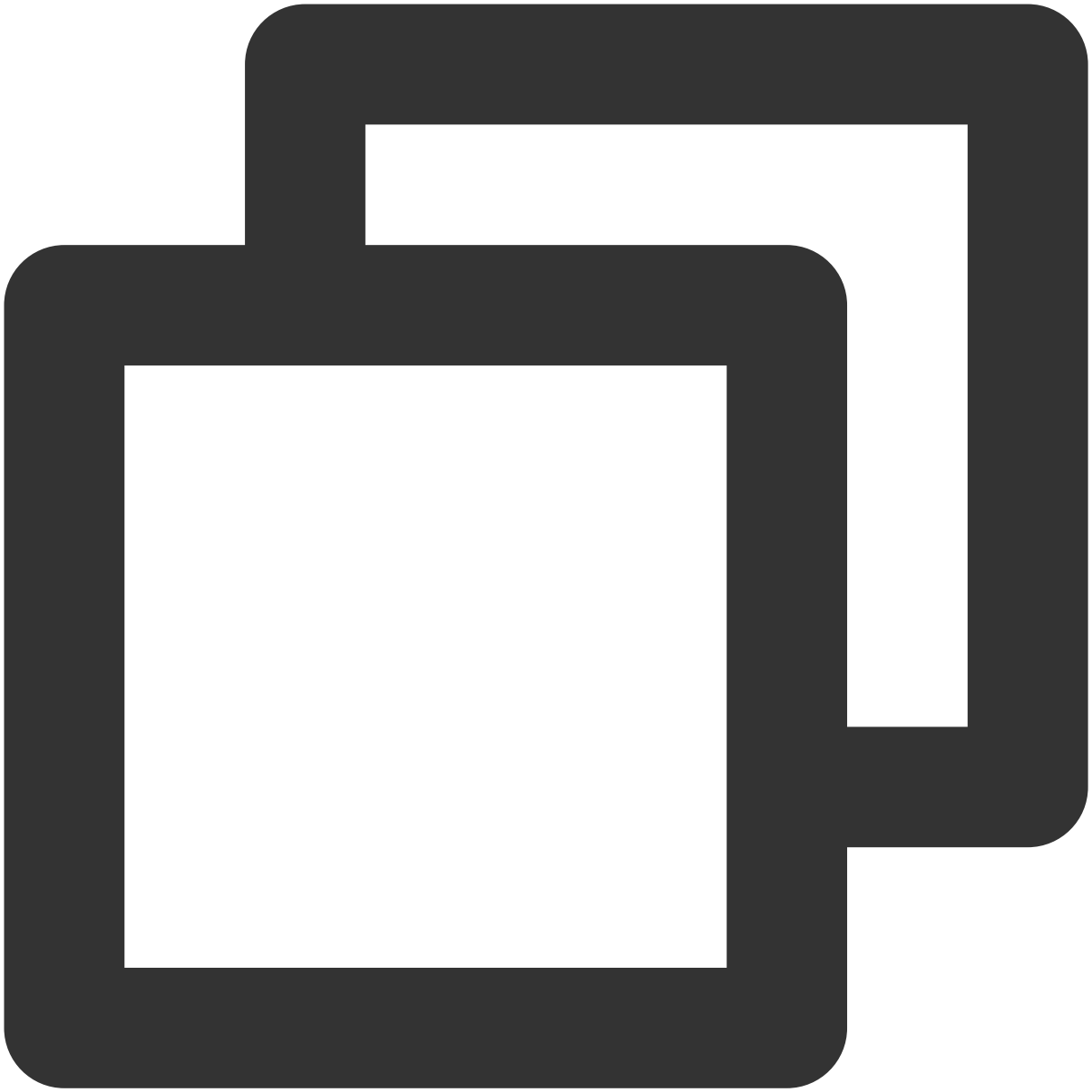
Last updated：2024-02-02 15:27:57

DataX is an open-source CLI that supports importing full or incremental data from TencentDB to CDWPG. The tool is developed in Java and uses JDBC to connect the source database to the target database. It can run on Windows and Linux. Install the Java environment before use.

**DataX installation:**

1. Download the source code here and compile it.
2. Directly use datax-v1.0.4-hashdata.tar.gz, an already compiled version.

The following section introduces DataX modified by HashData, which is more efficient to import data to CDWPG. Tests show that it can import more than 100,000 entries per second. The following is the configuration file to import data from MySQL to CDWPG:



```
{
    "job": {
        "setting": {
            "speed": {
                "channel": 3,
                "byte": 1048576,
                "record": 1000
            },
            "errorLimit": {
                "record": 2,
                "percentage": 0.02
```

```
            }
        },
        "content": [
            {
                "reader": {
                    "name": "mysqlreader",
                    "parameter": {
                        "username": "****",
                        "password": "****",
                        "column": [
                            "*"
                        ],
                        "splitPk": "id",
                        "connection": [
                            {
                                "table": [
                                    "test1"
                                ],
                                "jdbcUrl": [
                                    "jdbc:mysql://***:***/db1?serverTimezone=Asia/S
                                ]
                            }
                        ]
                    }
                },
                "writer": {
                    "name": "gpdbwriter",
                    "parameter": {
                        "username": "******",
                        "password": "******",
                        "column": [
                            "*"
                        ],
                        "preSql": [
                            "truncate table test1"
                        ],
                        "postSql": [
                            "select count(*) from test2"
                        ],
                        "segment_reject_limit": 0,
                        "copy_queue_size": 2000,
                        "num_copy_processor": 1,
                        "num_copy_writer": 1,
                        "connection": [
                            {
                                "jdbcUrl": "jdbc:postgresql://****:**/db1",
                                "table": [
```

```
                                    "test1"
                                ]
                            }
                        ]
                    }
                }
            }
        ]
    }
}
```

**Parameter description:**

1. The `writer` should be `gpdbwriter`. `postgresqlwriter` can also be used to write data to CDWPG, with a poor insertion efficiency though.
2. For specific meanings and parameter tuning, see DataX.
3. We recommend you add the `serverTimezone=Asia/Shanghai` parameter to the JDBC URL of `mysqlreader` to avoid data inconsistency caused by time zone issues.

# Syncing Incremental Data from MySQL with DataX

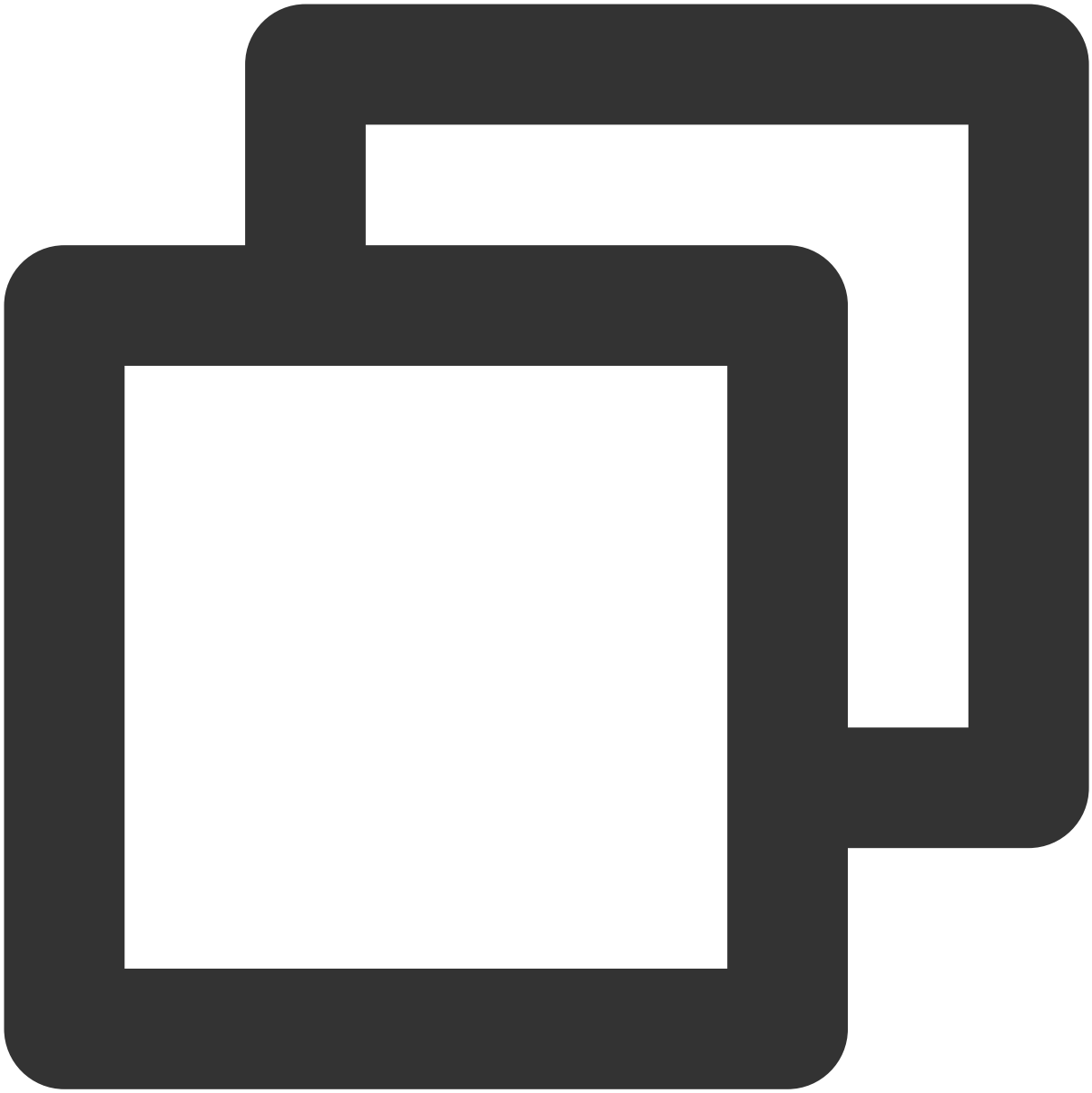
Last updated：2024-02-02 15:27:57

This document describes how to use DataX modified by HashData to incrementally sync data from MySQL to CDWPG.

Follow the steps below to use DataX to incrementally sync data from MySQL to CDWPG:

1. Read the `MaxTime` since the last successful sync from the local file (for the initial sync, you can specify an initial time value as required by your business).
2. Use `MaxTime` as the `LastTime` (lower limit of the incremental sync) and `CurTime` as the upper limit.
3. Modify the `datax.json` configuration to specify the time interval ( `WHERE` clause in SQL) of the synced table as `[LastTime, CurTime)` .
4. Execute DataX sync. After successful sync, write `CurTime` to the local file for the next sync.
5. Repeat steps 1–4 for multiple incremental syncs.

A sample `datax.json` configuration file is as shown below:



```
{
    "job": {
        "setting": {
            "speed": {
                "channel": 3,
                "byte": 1048576,
                "record": 1000
            },
            "errorLimit": {
                "record": 2,
                "percentage": 0.02
```

```
            }
        },
        "content": [
            {
                "reader": {
                    "name": "mysqlreader",
                    "parameter": {
                        "username": "******",
                        "password": "******",
                        "connection": [
                            {
                                "jdbcUrl": [
                                    "jdbc:mysql://***:***/test?serverTimezone=Asia/
                                ],
                                "querySql": [
                                    "select * from cdw_test_table where updateTime
                                ]
                            }
                        ]
                    }
                },
                "writer": {
                    "name": "gpdbwriter",
                    "parameter": {
                        "username": "******",
                        "password": "******",
                        "column": [
                                 "*"
                        ],
                        "segment_reject_limit": 0,
                        "copy_queue_size": 2000,
                        "num_copy_processor": 1,
                        "num_copy_writer": 1,
                        "connection": [
                            {
                                "jdbcUrl": "jdbc:postgresql://***:***/***",
                                "table": [
                                    "ods_cdw_test_table"
                                ]
                            }
                        ]
                    }
                }
            }
        ]
    }
}
```

# Importing and Exporting COS Data at High Speed with External Table

Last updated：2024-02-02 15:27:57

## Querying COS Data with COS_EXT

COS_EXT is an external data access extension for accessing COS files. By defining an external table through DDL, you can run DML in the external table as a normal data table to manipulate COS data. The following are supported currently:
Read COS data as an external table.
Export results to COS as an external table.
Perform simple analysis of COS data as an external table.

### Notes

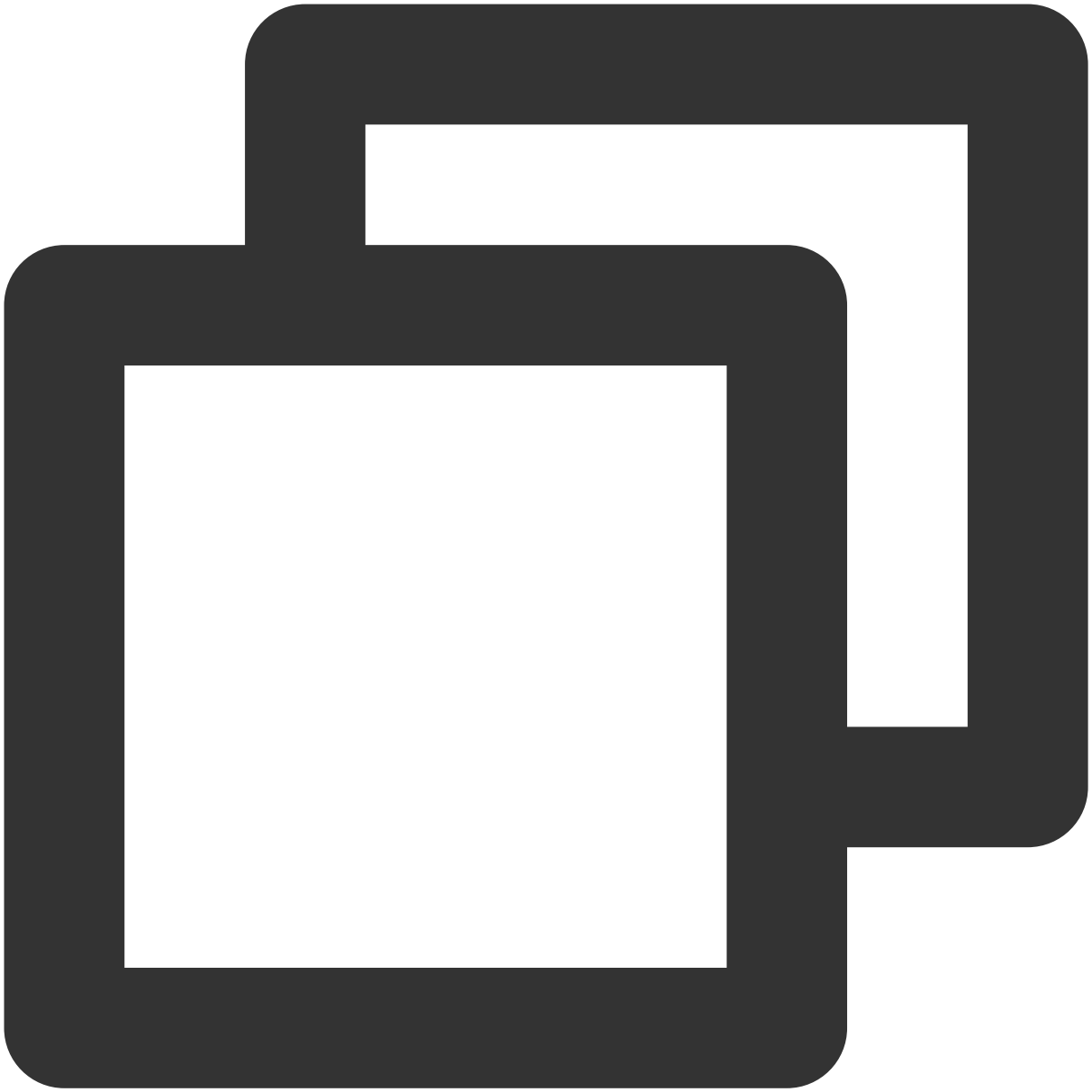
1. Only files in text formats such as CSV and GZIP compressed format files are supported.
2. Only COS data in the same region can be read. For example, a cluster in Guangzhou Zone 4 can only read COS data in the Guangzhou region.
3. Only your own COS data can be read by your cluster.
4. Write-only external tables can only be used for the `INSERT` statement but not `UPDATE`, `DELETE`, and `SELECT` statements.
5. Deleting an external table will not delete the corresponding data in COS.

### Directions

1. Define the COS_EXT extension.

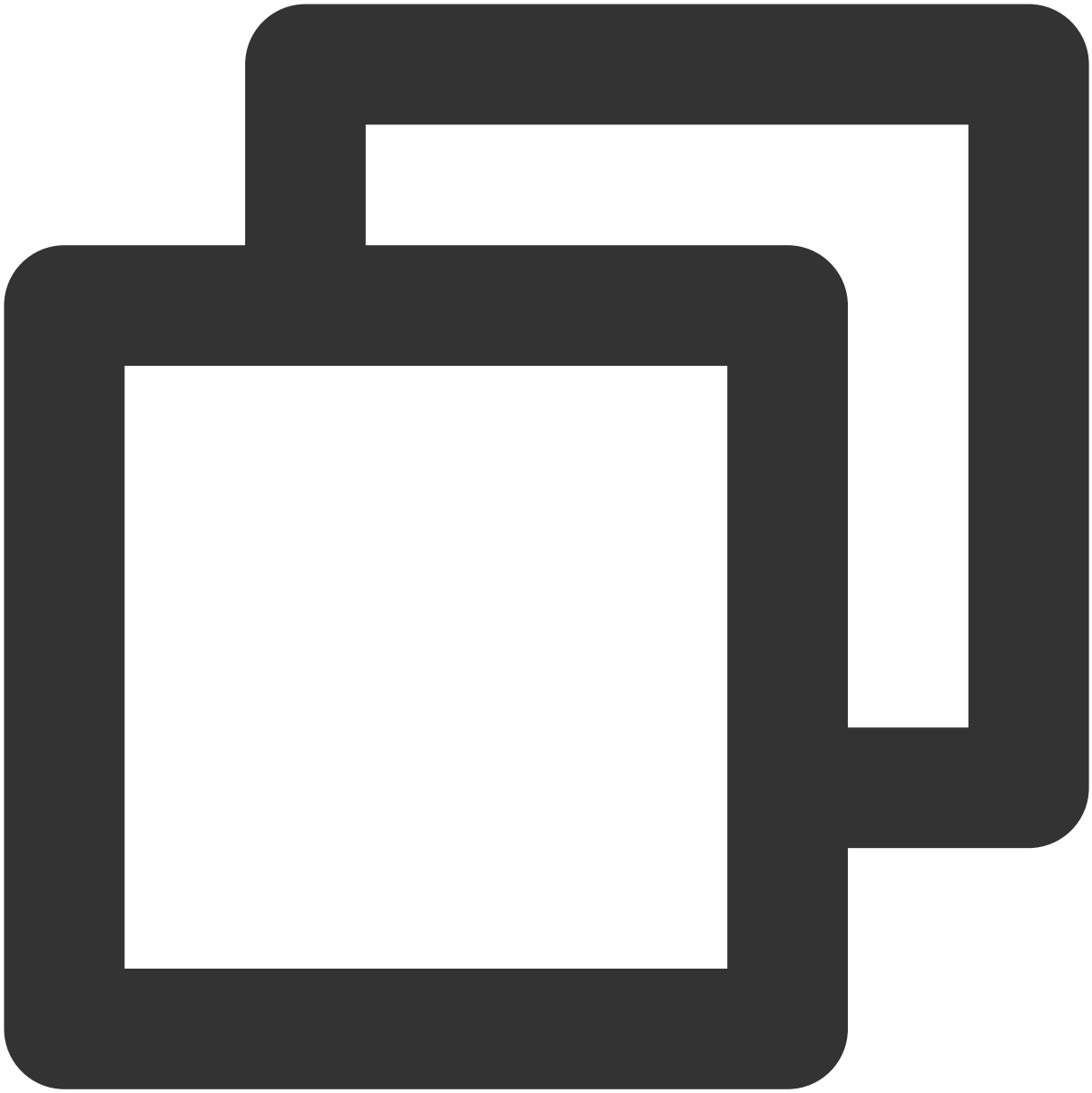
**Note:**

The scope of the COS external table extension is the database.
The creation command is as follows:



```
CREATE EXTENSION IF NOT EXISTS cos_ext SCHEMA public;
```

The deletion command is as follows:



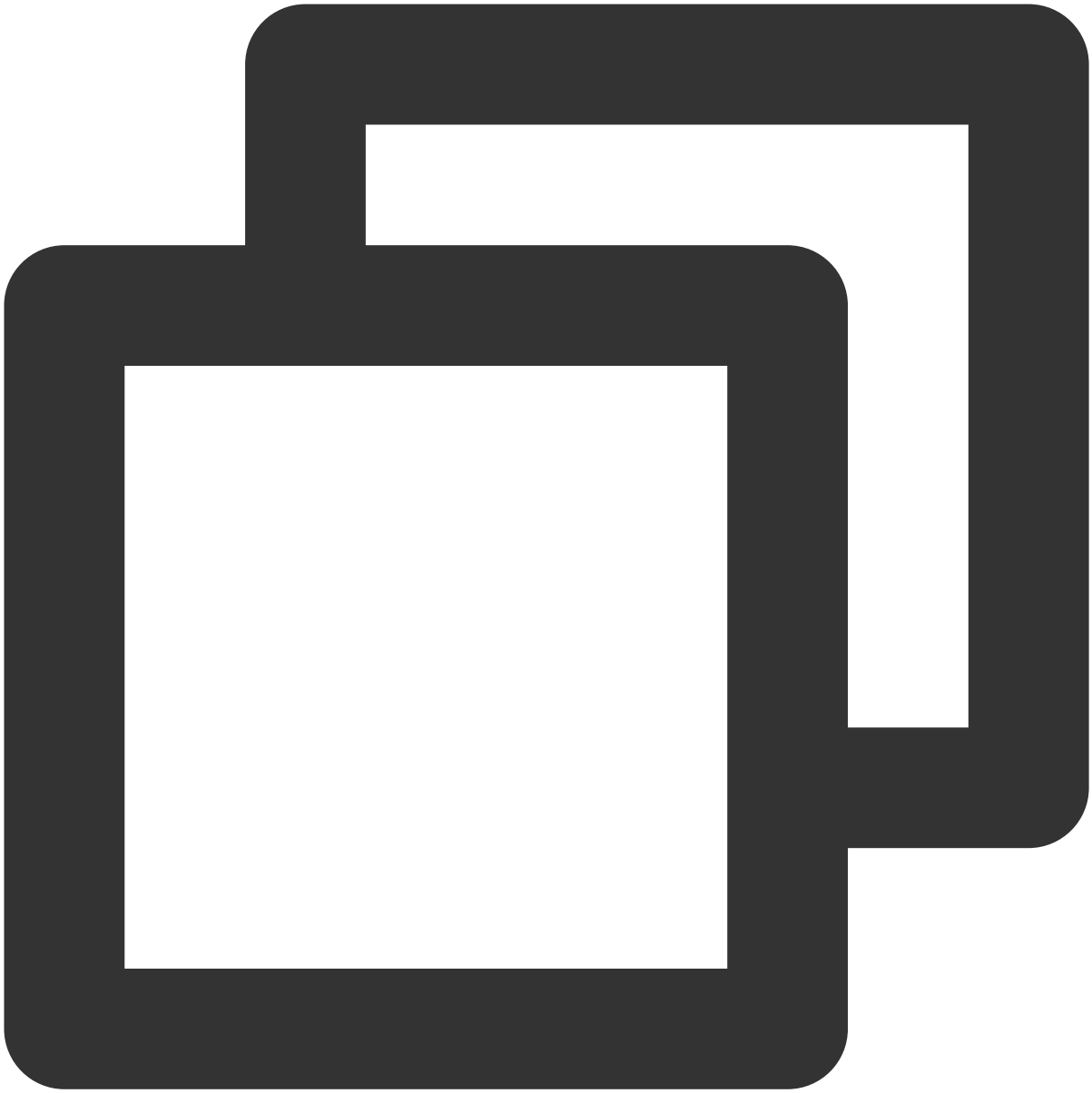
```
DROP EXTENSION IF EXISTS cos_ext;
```

2. Define the COS external table. For syntax, see Syntax description.

3. Manipulate data in the COS external table.

### Syntax description

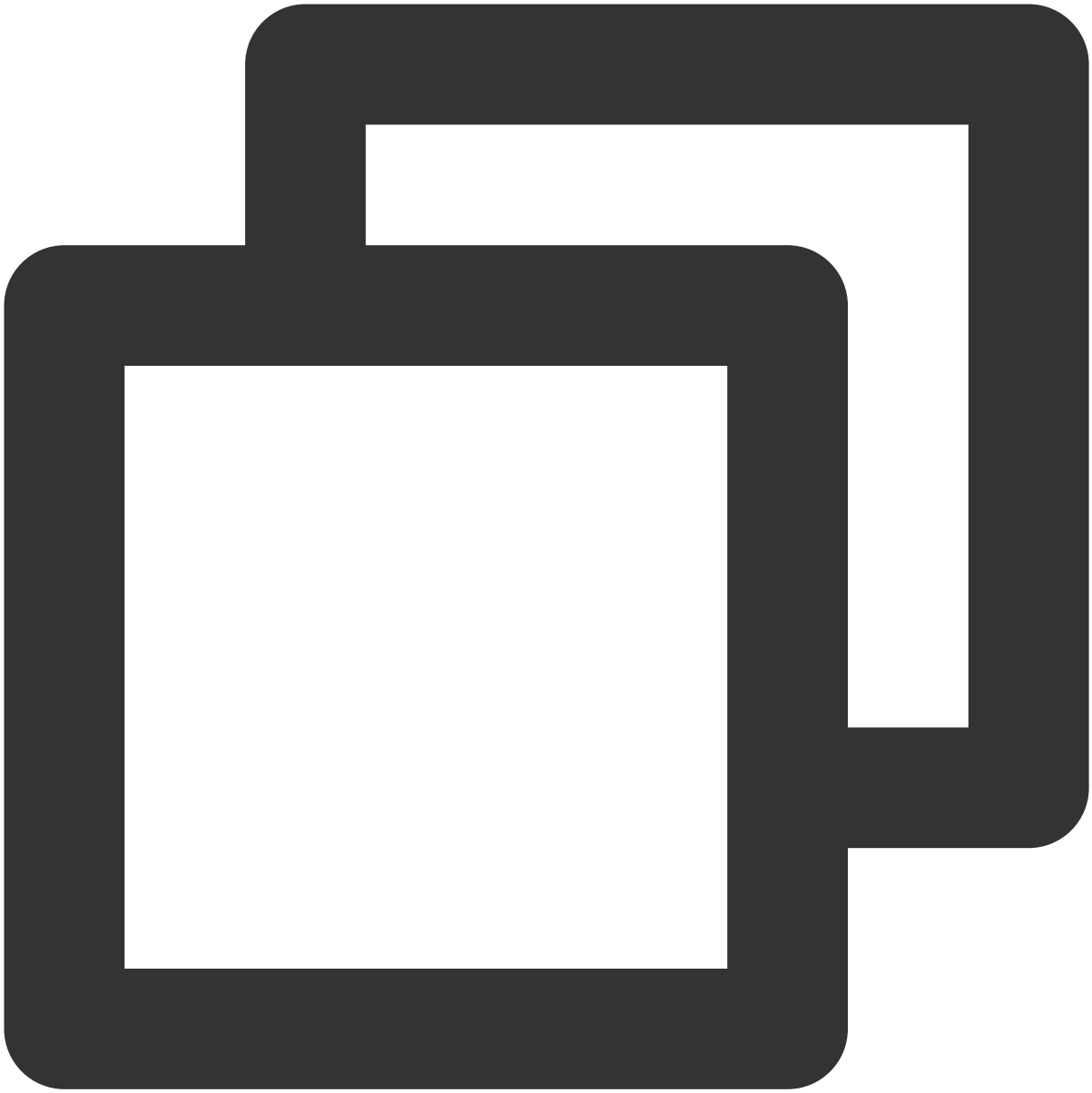
Definite the read-only input table



```
CREATE [READABLE] EXTERNAL TABLE tablename
( columnname datatype [, ...] | LIKE othertable )
LOCATION (cos_ext_params)
FORMAT 'TEXT'
      [( [HEADER]
             [DELIMITER [AS] 'delimiter' | 'OFF']
             [NULL [AS] 'null string']
             [ESCAPE [AS] 'escape' | 'OFF']
             [NEWLINE [ AS ] 'LF' | 'CR' | 'CRLF']
             [FILL MISSING FIELDS] )]
     | 'CSV'
```

```
        [( [HEADER]
               [QUOTE [AS] 'quote']
               [DELIMITER [AS] 'delimiter']
               [NULL [AS] 'null string']
               [FORCE NOT NULL column [, ...]]
               [ESCAPE [AS] 'escape']
               [NEWLINE [ AS ] 'LF' | 'CR' | 'CRLF']
               [FILL MISSING FIELDS] )]
  [ ENCODING 'encoding' ]
  [ [LOG ERRORS [INTO error_table]] SEGMENT REJECT LIMIT count
         [ROWS | PERCENT] ]
```

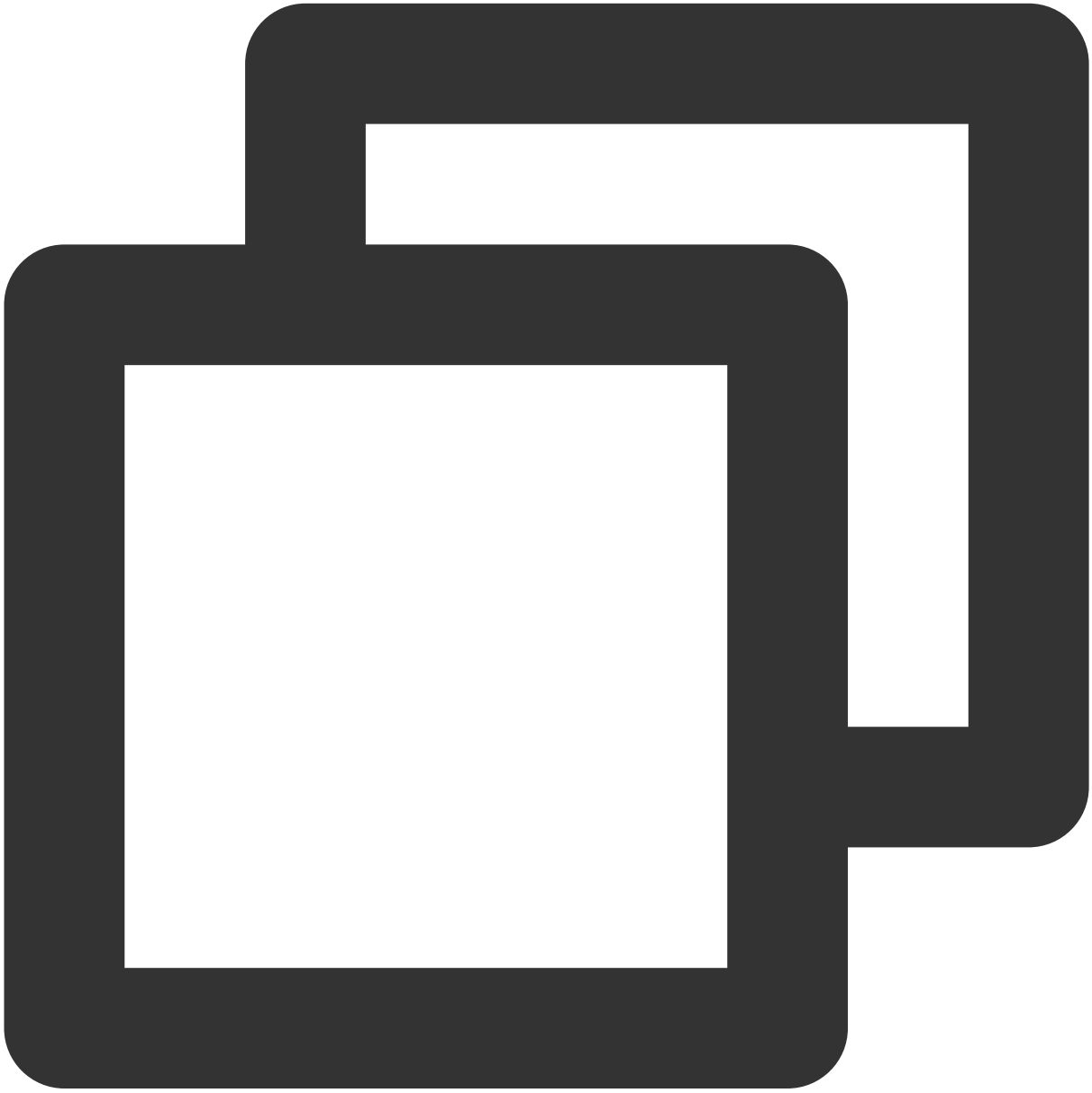
Define the write-only output table



```
CREATE WRITABLE EXTERNAL TABLE table_name
( column_name data_type [, ...] | LIKE other_table )
LOCATION (cos_ext_params)
FORMAT 'TEXT'
         [( [DELIMITER [AS] 'delimiter']
         [NULL [AS] 'null string']
         [ESCAPE [AS] 'escape' | 'OFF'] )]
     | 'CSV'
         [([QUOTE [AS] 'quote']
         [DELIMITER [AS] 'delimiter']
         [NULL [AS] 'null string']
```

```
          [FORCE QUOTE column [, ...] ]
          [ESCAPE [AS] 'escape'] )]
  [ ENCODING 'encoding' ]
  [ DISTRIBUTED BY (column, [ ... ] ) | DISTRIBUTED RANDOMLY ]
```

1. `cos_ext_params` description



```
cos://cos_endpoint/bucket/prefix secretId=id secretKey=key compressType=[none|gzip]
```

### Parameter description

| | | | |
|---|---|---|---|

| Parameter | Format | Required | Description |
| --- | --- | --- | --- |
| URL | COS V4: `cos://cos.{REGION}.myqcloud.com/{BUCKET}/{PREFIX}` COS V5: `cos:// {BUCKET}-{APPID}.cos.{REGION}.myqcloud.com/{PREFIX}` | Yes | See URL parameter description |
| secretId | None | Yes | Secret ID used for API access. See API Key Management |
| secretKey | None | Yes | Secret key used for API access. See API Key Management |
| HTTPS | true &lota; false | No | Whether to use HTTPS to access COS. Default value: true |
| compressType | gzip | No | Whether to compress COS files. Default value: empty (not to compress) |

**URL parameter description**

REGION: Region supported by COS, which needs to be the same region as the instance. For valid values, see Regions and Access Endpoints.

BUCKET: COS bucket name, which can be seen in COS Bucket List. **The name here does not contain the `APPID`** . If you see the bucket name `test-123123123` in the list, just enter "test".

PREFIX: COS object name prefix, which can be empty and include multiple "/".

In a read-only table scenario, `prefix` specifies the name prefix of the object to be read.

If `prefix` is empty, all files in the bucket will be read; if it ends with "/", all files in the folder and subfolders will be matched; otherwise, all files in the folder and subfolders matched by `prefix` will be read. For example, COS objects include `read-bucket/simple/a.csv` , `read-bucket/simple/b.csv` , `read-bucket/simple/dir/c.csv` , and `read-bucket/simple_prefix/d.csv` .

If `prefix` is specified as `simple` , all files will be read, including `simple_prefix` with the matching directory name prefix. The following is the list of objects:

read-bucket/simple/a.csv
read-bucket/simple/b.csv
read-bucket/simple/dir/c.csv
read-bucket/simple_prefix/d.csv

If `prefix` is specified as `simple/` , all files including `simple/` will be read, including:

read-bucket/simple/a.csv
read-bucket/simple/b.csv
read-bucket/simple/dir/c.csv

In a write-only table scenario, `prefix` specifies the output file prefix.

If no `prefix` is specified, files will be written to the bucket. If `prefix` ends with "/", files will be written to the directory specified by `prefix` ; otherwise, files will be prefixed with the given `prefix` . For example, if the files that need to be created include `a.csv` , `b.csv` , and `c.csv` , then:

If `prefix` is specified as `simple/` , the following objects will be generated:

read-bucket/simple/a.csv
read-bucket/simple/b.csv
read-bucket/simple/b.csv

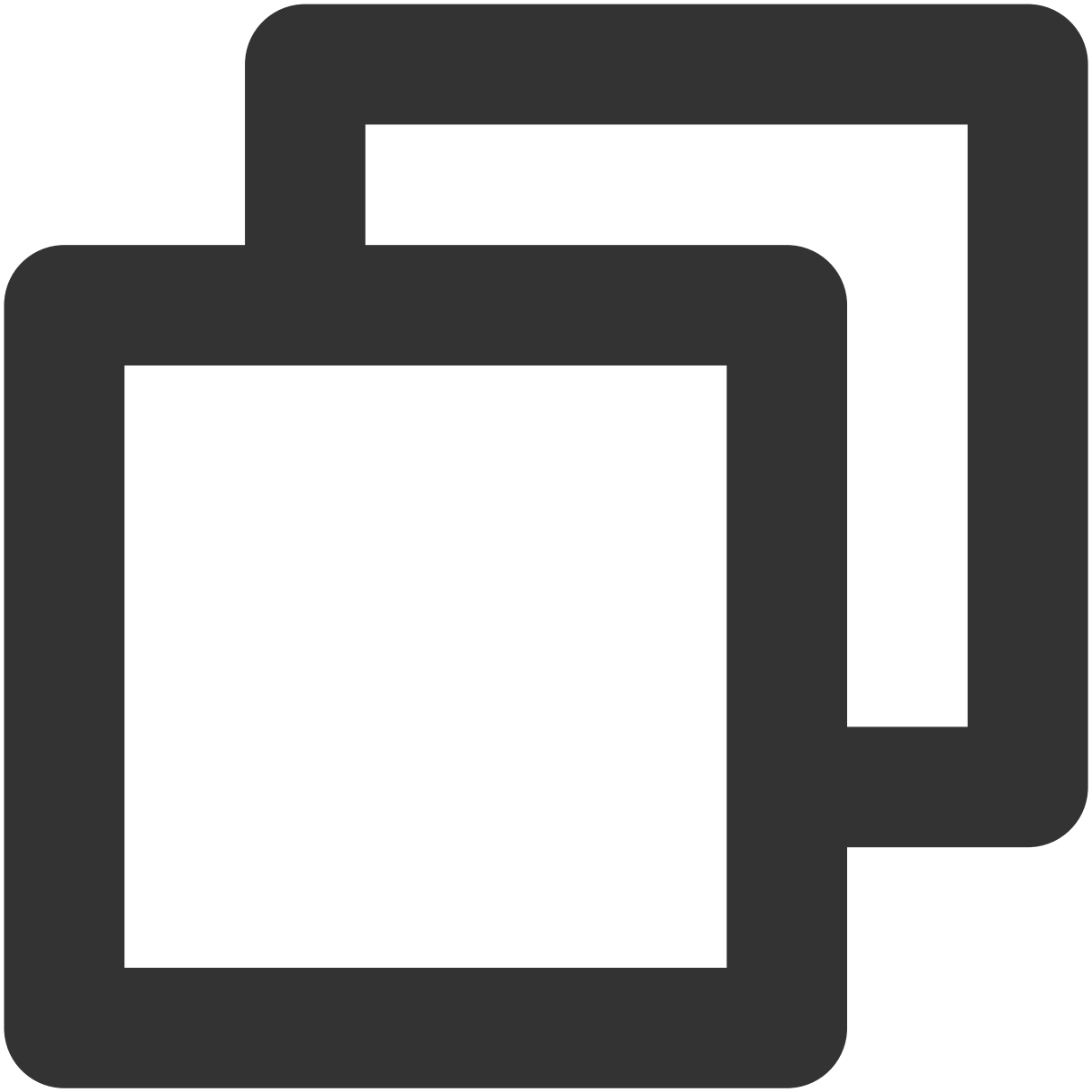
If `prefix` is specified as `simple\\_` , the following objects will be generated:

read-bucket/simple_a.csv
read-bucket/simple_b.csv
read-bucket/simple_b.csv

# Use Cases

### Importing COS data

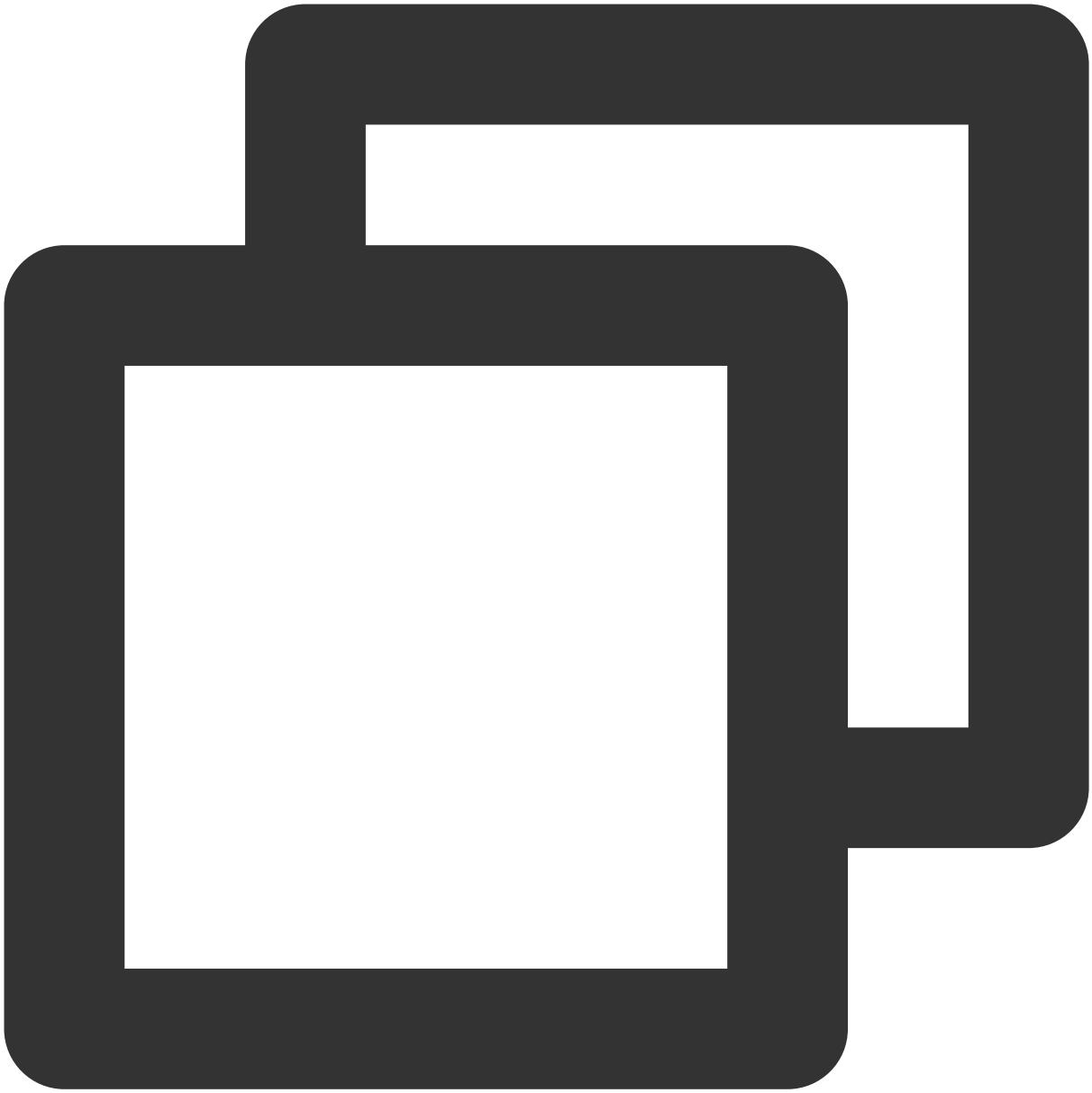
1. Define the COS extension.



```
CREATE EXTENSION IF NOT EXISTS cos_ext SCHEMA public;
```

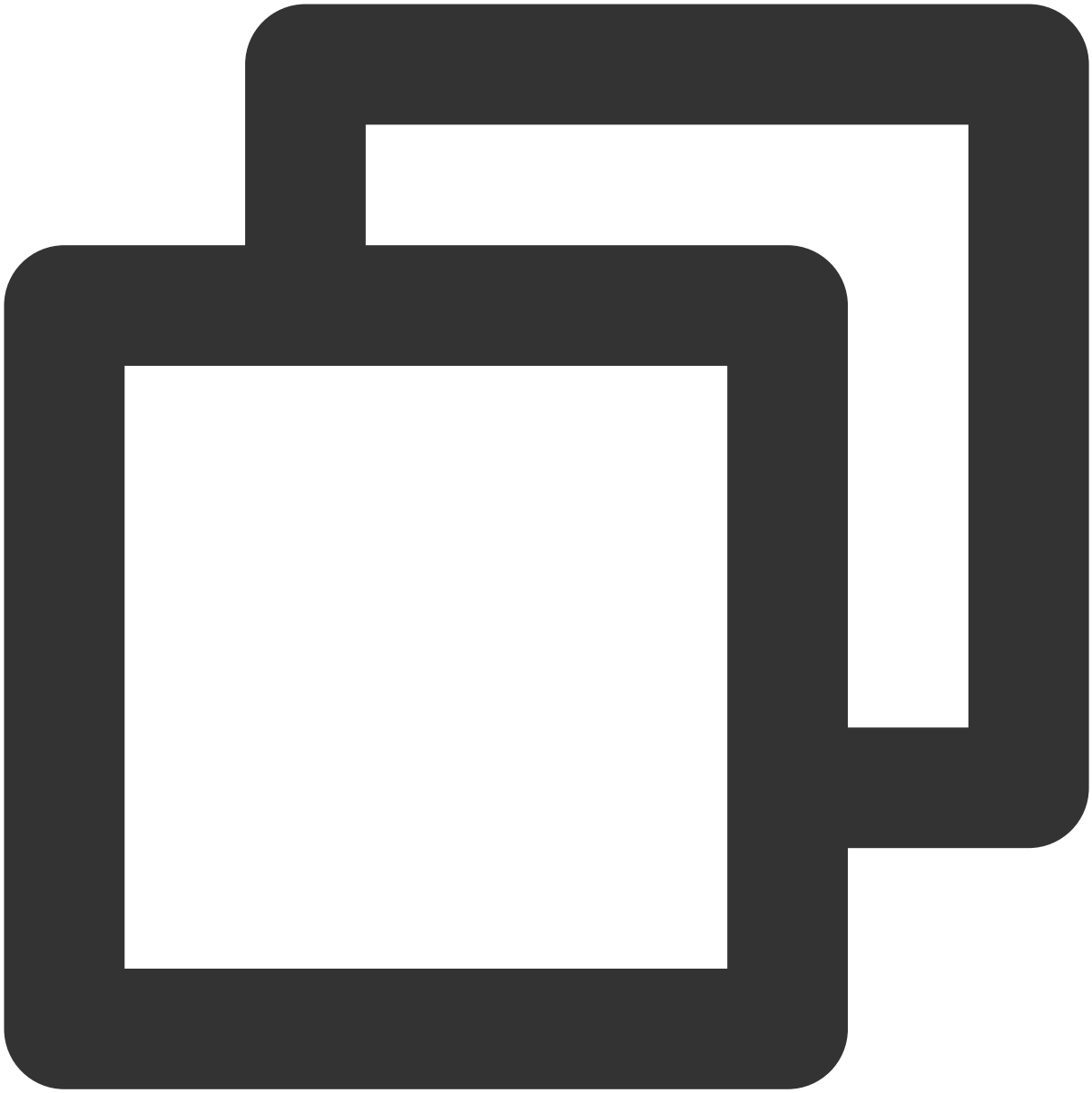
2. Define a COS read-only external table and a local table.

Local table:



```
CREATE TABLE cos_local_tbl (c1 int, c2 text, c3 int)
DISTRIBUTED BY (c1);
```

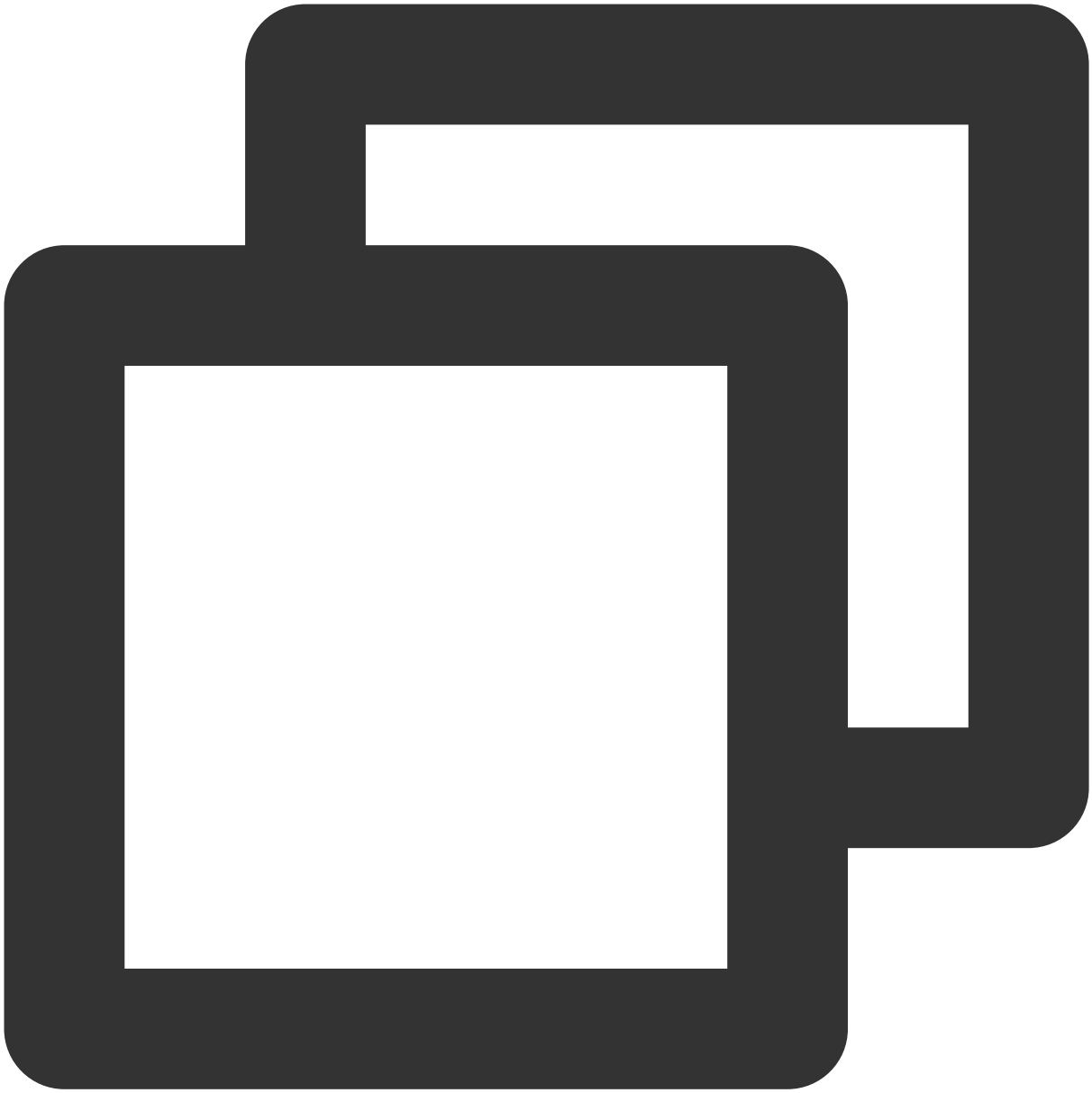
COS external table: Specifies to read all files in `simple-bucket` in Guangzhou.



```
CREATE READABLE EXTERNAL TABLE cos_tbl (c1 int, c2 text, c3 int)
LOCATION('cos://cos.ap-guangzhou.myqcloud.com/simple-bucket/from_cos/ secretKey=xxx
FORMAT 'csv';
```

3. Prepare local table data.

Upload the file to the `from_cos` directory in `simple-bucket` with the following content:

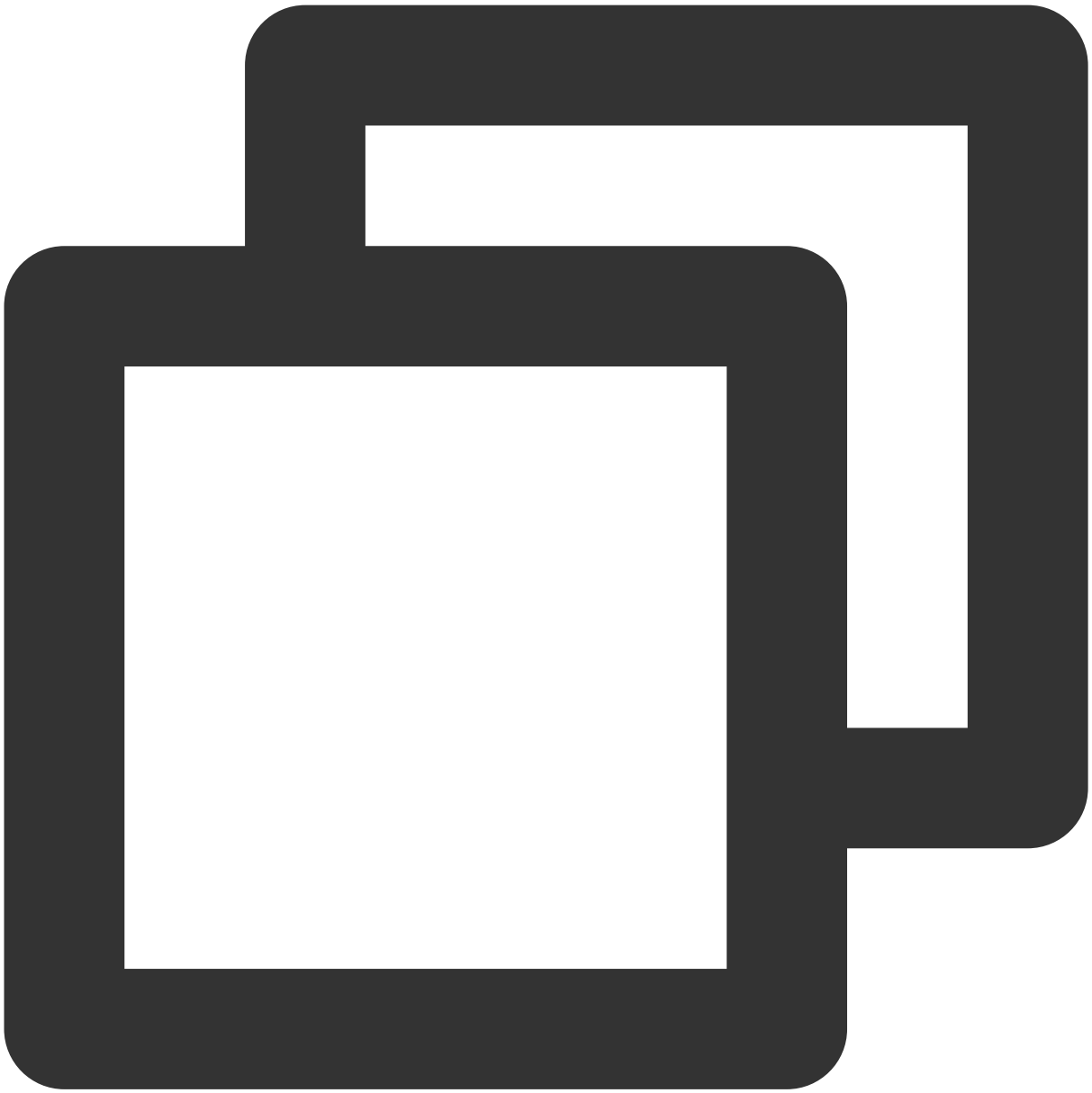


```
1,simple line 1,1
2,simple line 1,1
3,simple line 1,1
4,simple line 1,1
5,simple line 1,1
6,simple line 2,1
7,simple line 2,1
8,simple line 2,1
9,simple line 2,1
```

**Note:**

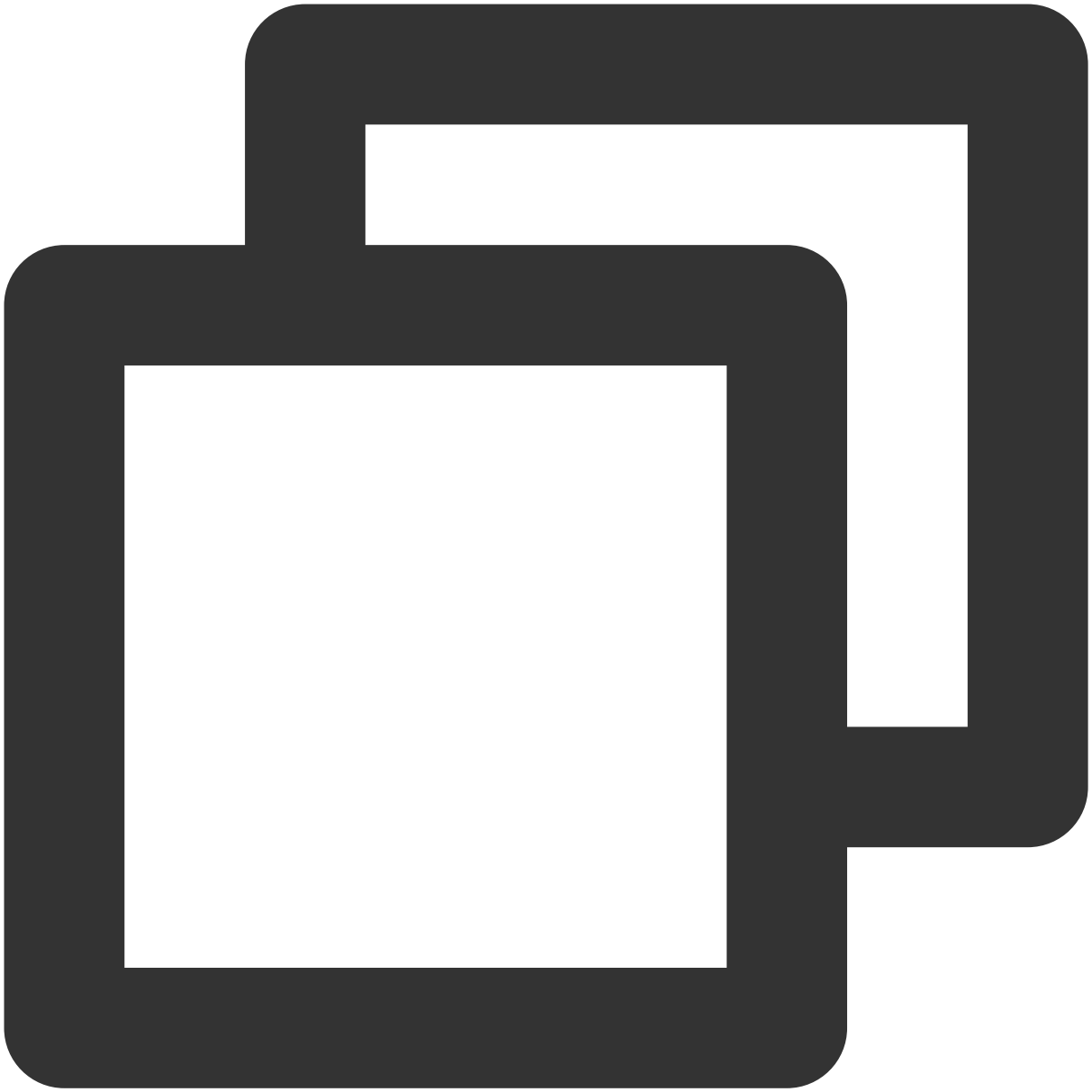
The imported data does not contain field rows of the table header.

4. Import COS data.



```
INSERT INTO cos_local_tbl SELECT * FROM cos_tbl;
```

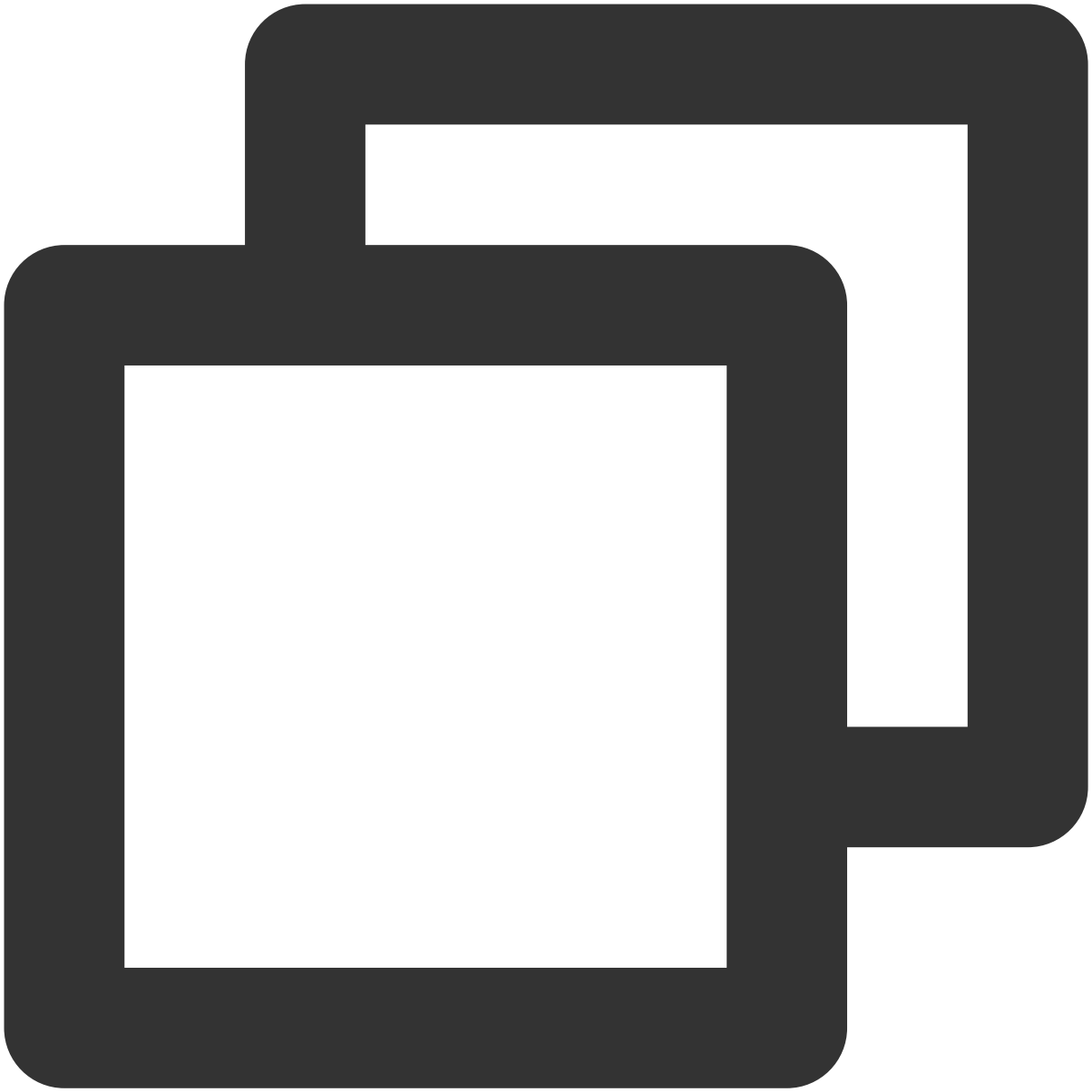
5. Check the result to see whether the data is consistent.



```
SELECT count(1) FROM cos_local_tbl;
SELECT count(1) FROM cos_tbl;
```

### Exporting data to COS

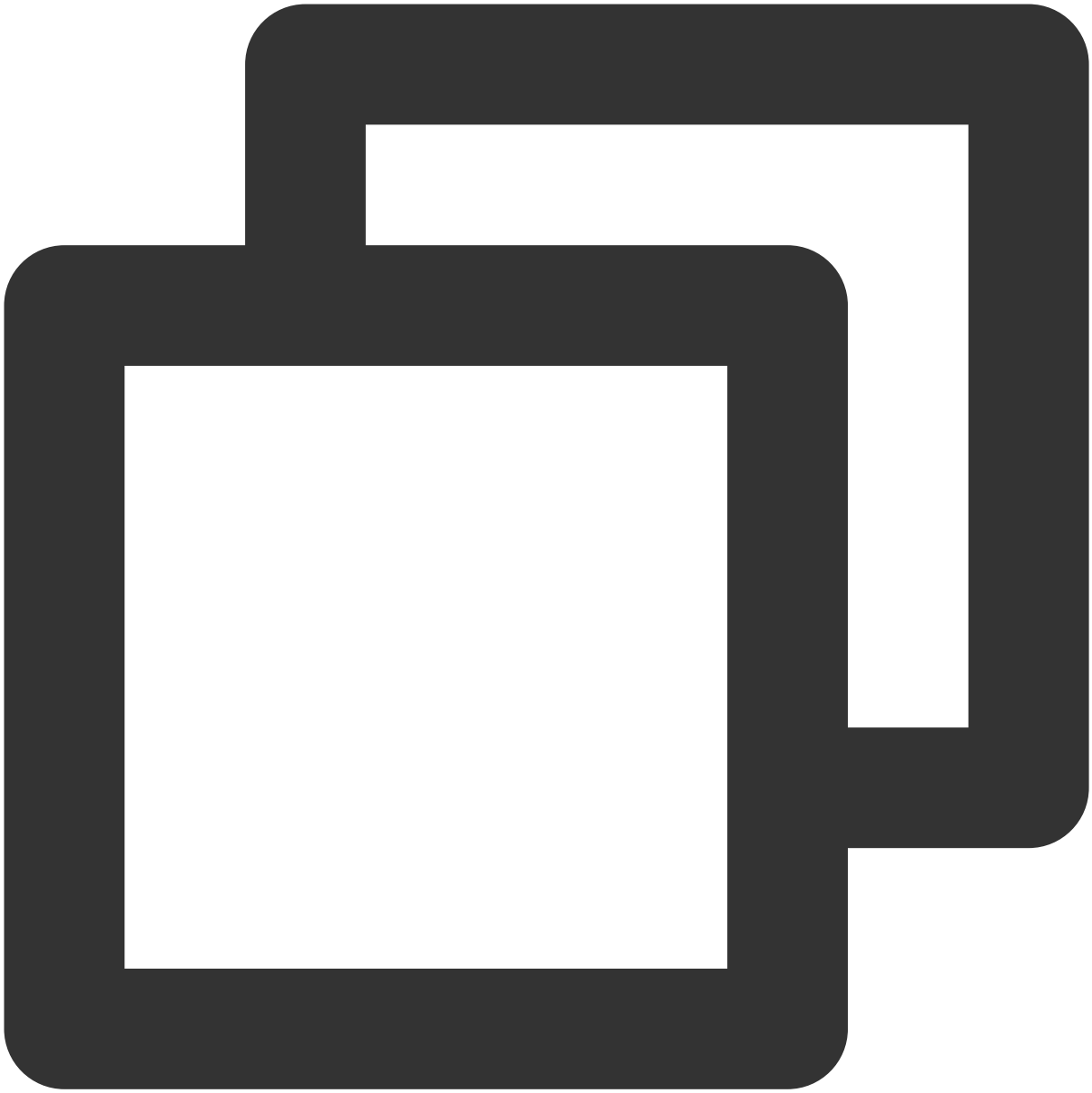
1. Define the COS extension.



```
CREATE EXTENSION IF NOT EXISTS cos_ext SCHEMA public;
```

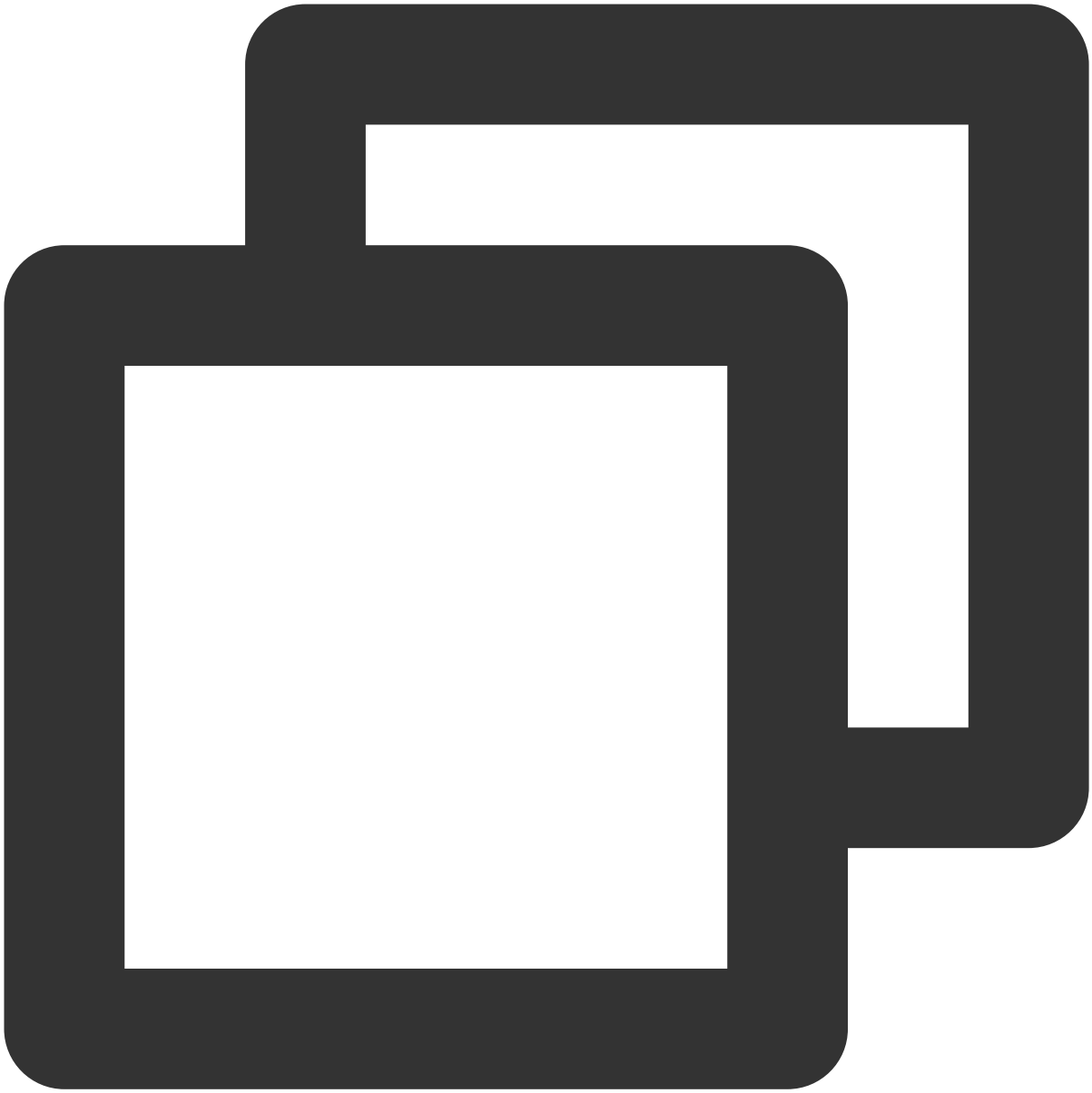
2. Define a COS write-only external table.

Local table:



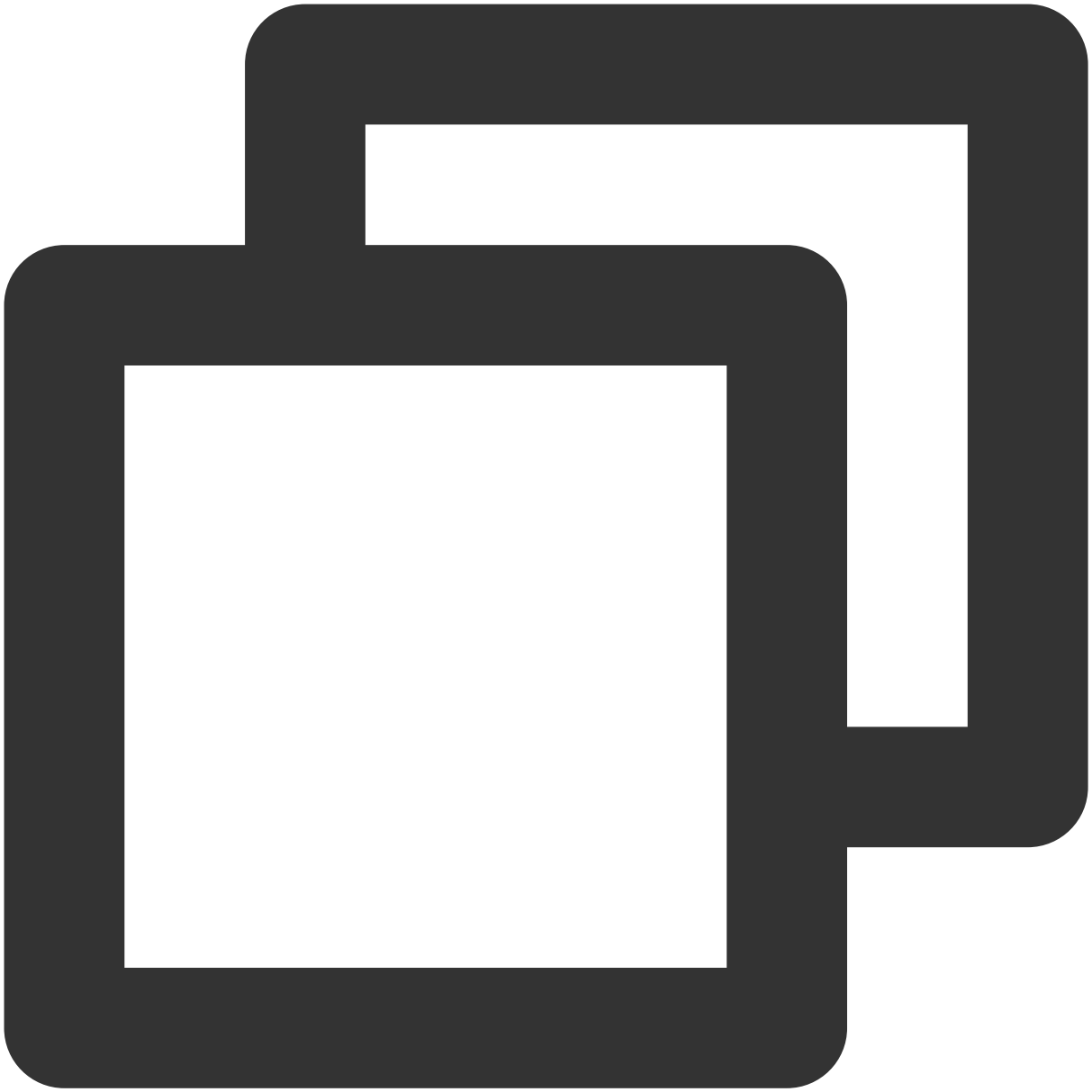
```
CREATE TABLE cos_local_tbl (c1 int, c2 text, c3 int)
DISTRIBUTED BY (c1);
```

COS external table: Specifies to write all files in `simple-bucket` in Guangzhou.



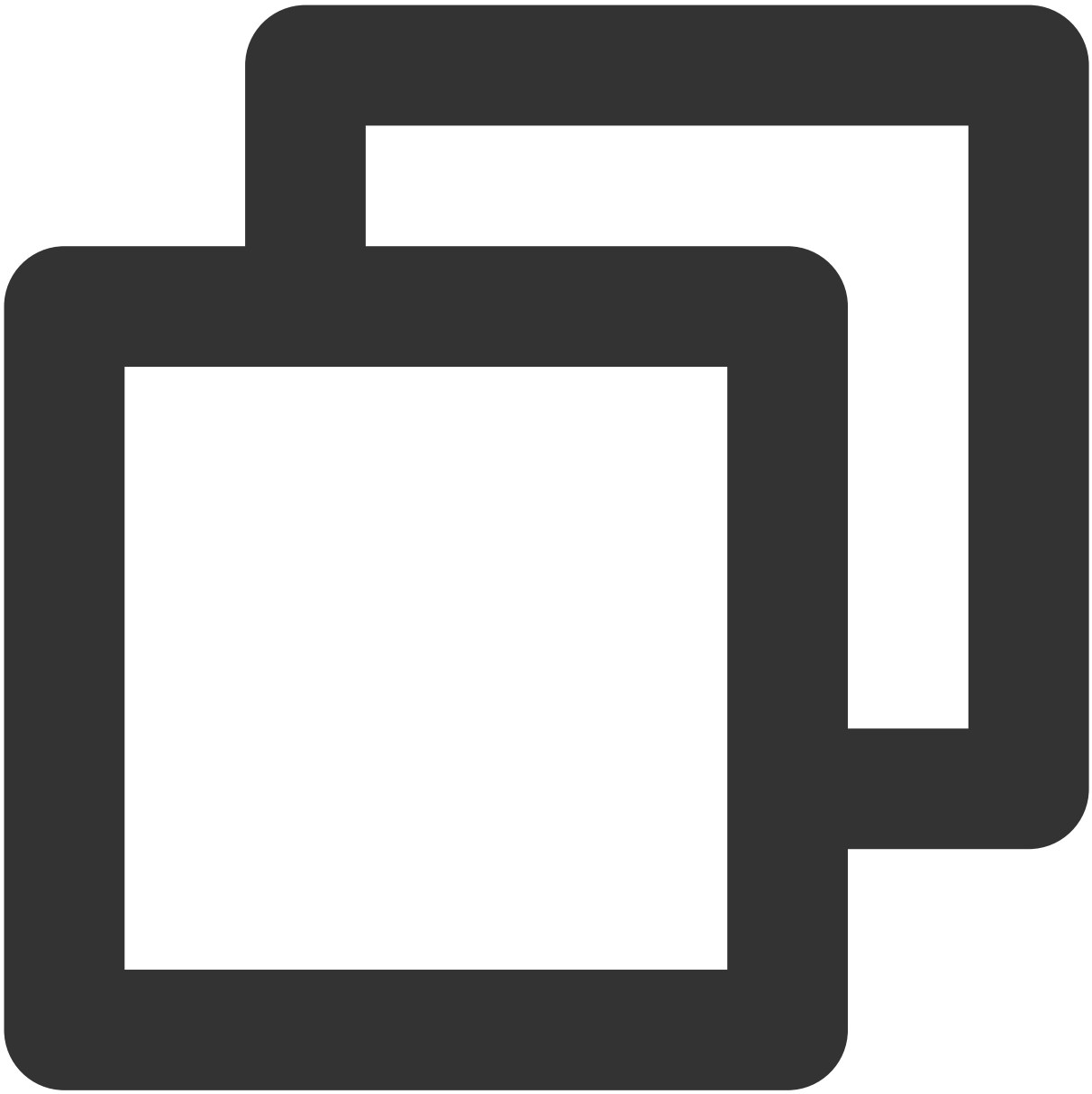
```
CREATE WRITABLE EXTERNAL TABLE cos_tbl_wr (c1 int, c2 text, c3 int)
LOCATION('cos://cos.ap-guangzhou.myqcloud.com/simple-bucket/to-cos/ secretKey=xxx s
FORMAT 'csv';
```

3. Construct the test data.



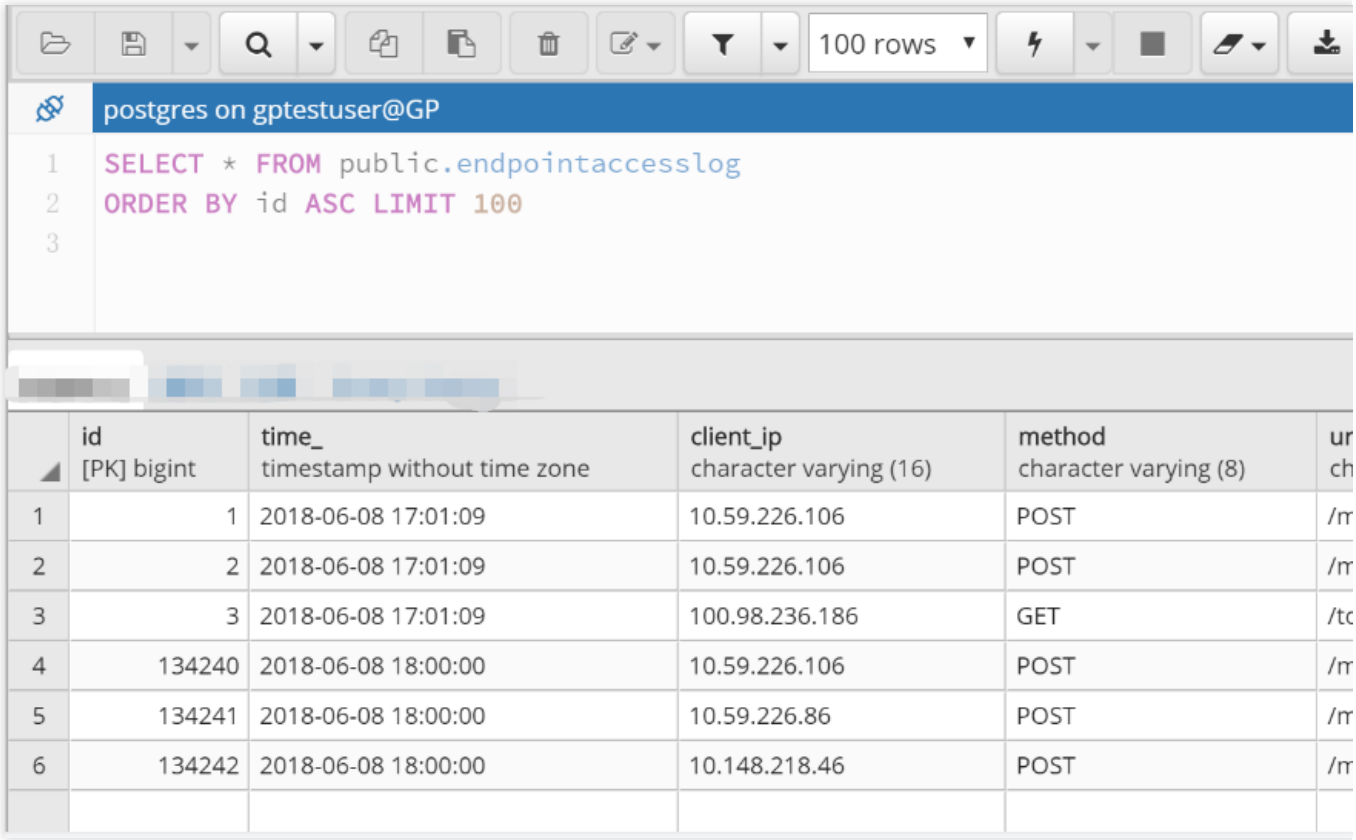
```
insert into cos_local_tbl values
(1, 'simple line 1' , 1),
(2, 'simple line 2', 2),
(3, 'simple line 3', 3) ,
(4, 'simple line 4', 4) ,
(5, 'simple line 5', 5) ,
(6, 'simple line 6', 6) ,
(7, 'simple line 7', 7) ,
(8, 'simple line 8', 8) ,
(9, 'simple line 9', 9);
```

4. Export the data to COS.



```
INSERT INTO cos_tbl_wr SELECT * FROM cos_local_tbl;
```

5. Check the result.



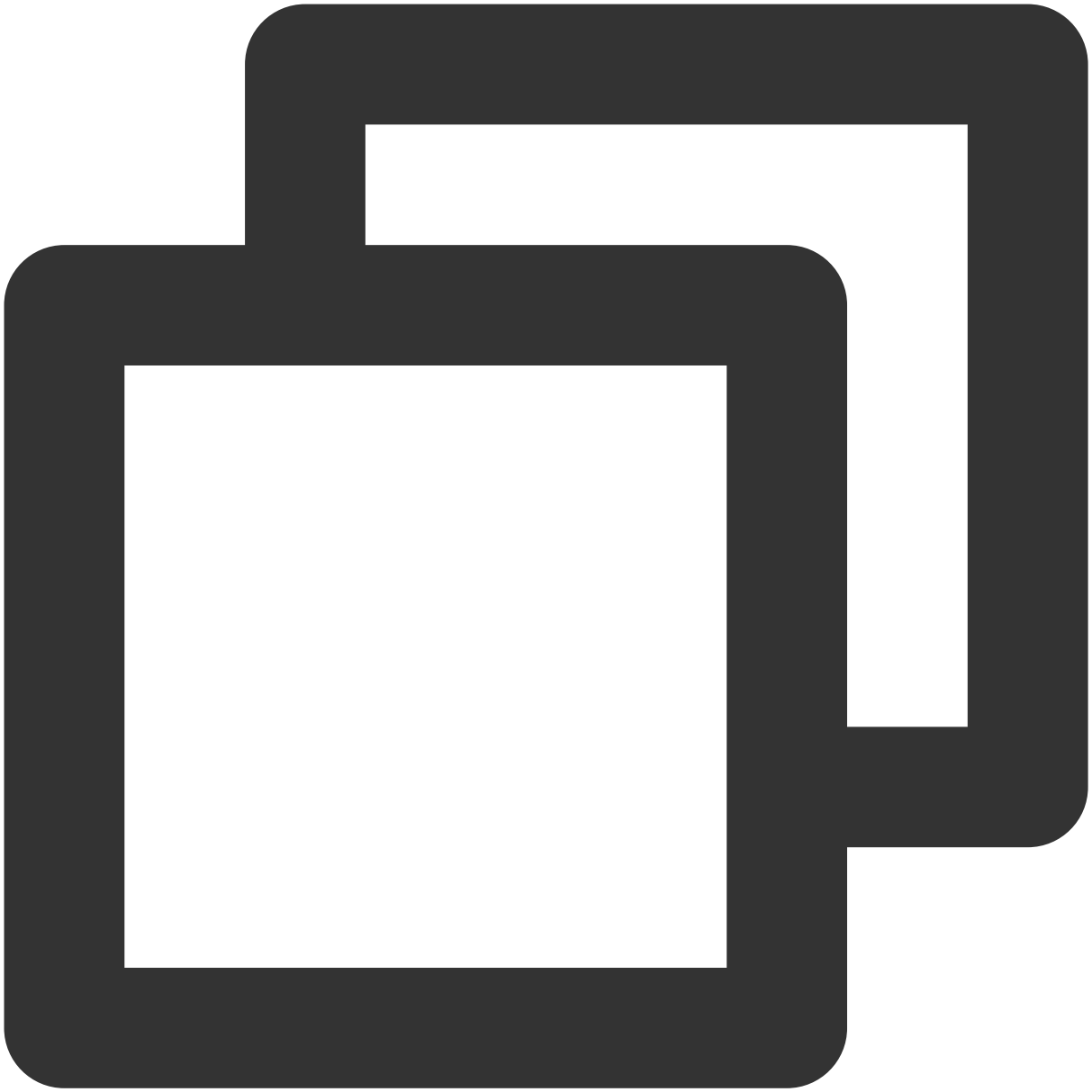

| | id [PK] bigint | time_ timestamp without time zone | client_ip character varying (16) | method character varying (8) | ur ch |
|---|---|---|---|---|---|
| 1 | 1 | 2018-06-08 17:01:09 | 10.59.226.106 | POST | /m |
| 2 | 2 | 2018-06-08 17:01:09 | 10.59.226.106 | POST | /m |
| 3 | 3 | 2018-06-08 17:01:09 | 100.98.236.186 | GET | /to |
| 4 | 134240 | 2018-06-08 18:00:00 | 10.59.226.106 | POST | /m |
| 5 | 134241 | 2018-06-08 18:00:00 | 10.59.226.86 | POST | /m |
| 6 | 134242 | 2018-06-08 18:00:00 | 10.148.218.46 | POST | /m |

## Simple analysis of COS data

**Note:**

When using COS external tables for query analysis without query optimization, we recommend you first import the data locally for complex queries.

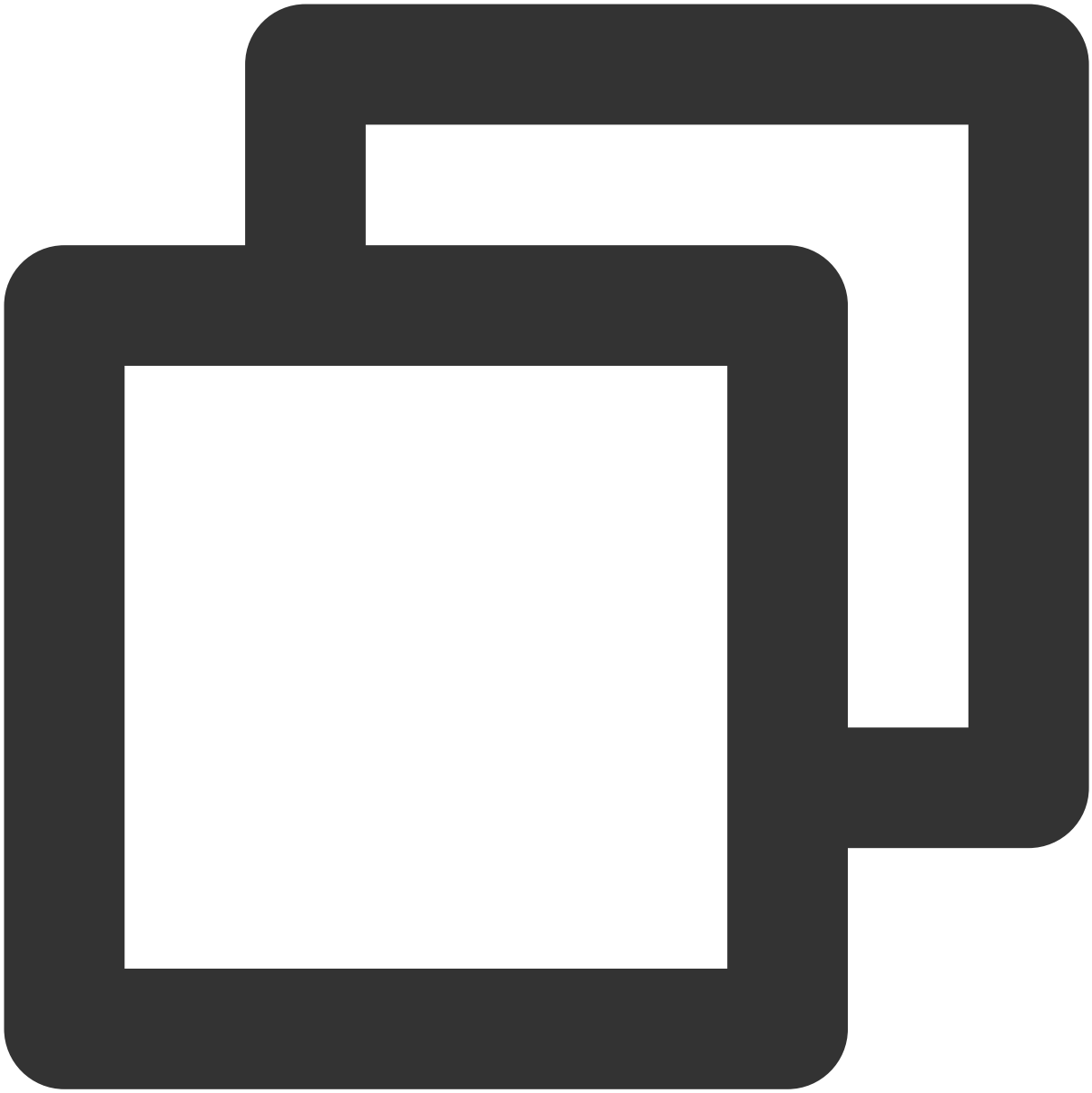
1. Define the COS extension.



```
CREATE EXTENSION IF NOT EXISTS cos_ext SCHEMA public;
```

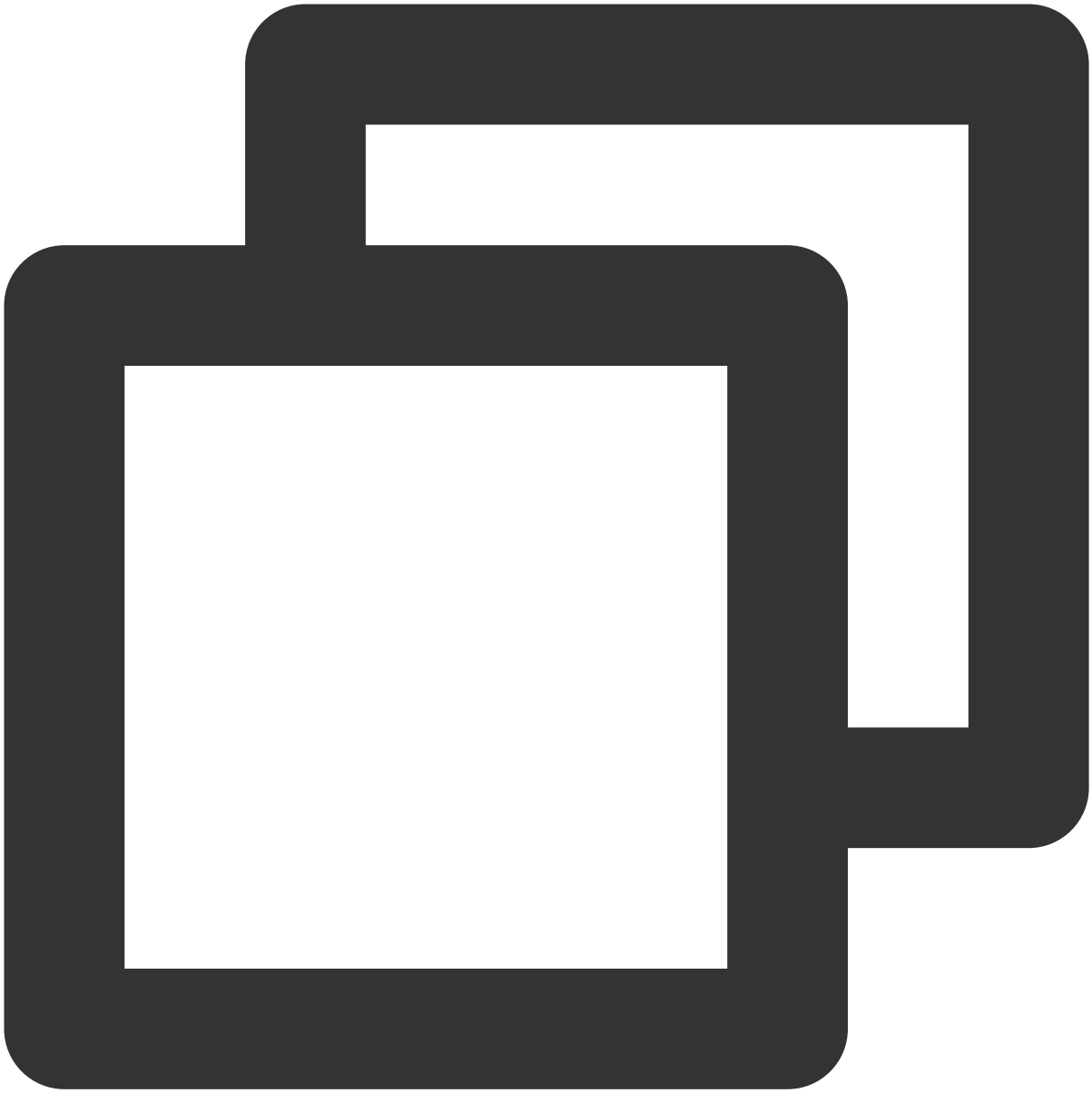
2. Prepare the data.

Upload the file to the `for-dml` directory in `simple-bucket` with the following content:



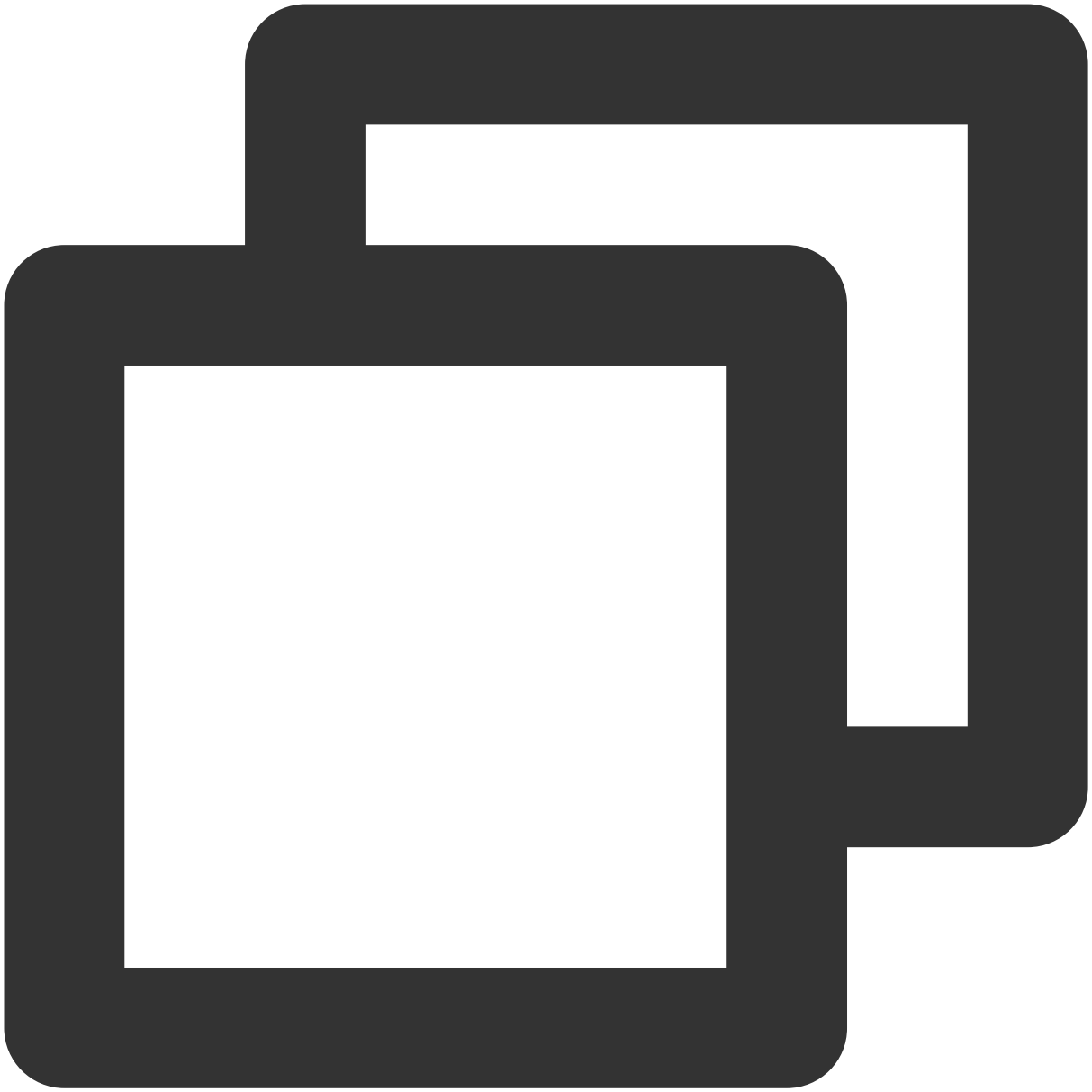
```
1,simple line 1,1
2,simple line 1,1
3,simple line 1,1
4,simple line 1,1
5,simple line 1,1
6,simple line 2,1
7,simple line 2,1
8,simple line 2,1
9,simple line 2,1
```

3. Define a COS read-only external table.



```
CREATE READABLE EXTERNAL TABLE cos_tbl_dml (c1 int, c2 text, c3 int)
LOCATION('cos://cos.ap-guangzhou.myqcloud.com/simple-bucket/for-dml/ secretKey=xxx
FORMAT 'csv';
```

4. Analyze the data in the COS external table.



```
SELECT c2, sum(c1) FROM cos_tbl GROUP BY c2;
```

# Syncing EMR Data with External Table

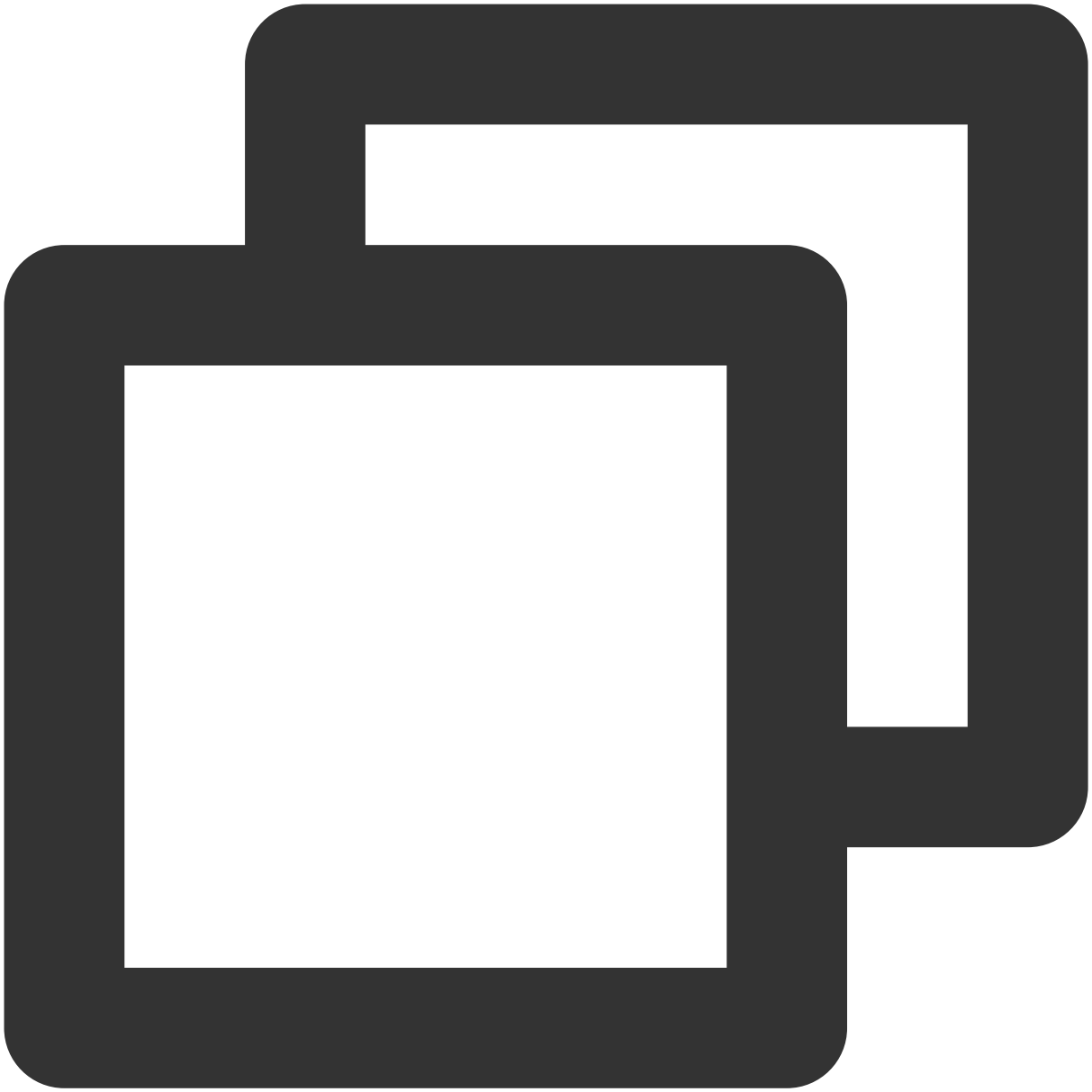
Last updated：2024-02-02 15:27:57

## Background

In data warehouse construction, Hive is usually used to process the raw data (at the petabyte level), perform time-consuming ETL jobs, and hand over the results (at the terabyte level) to a quasi-real-time computing engine such as CDWPG to connect BI tools and present reports in quasi real time.
This document describes how to import data from Hive on EMR to CDWPG via COS.

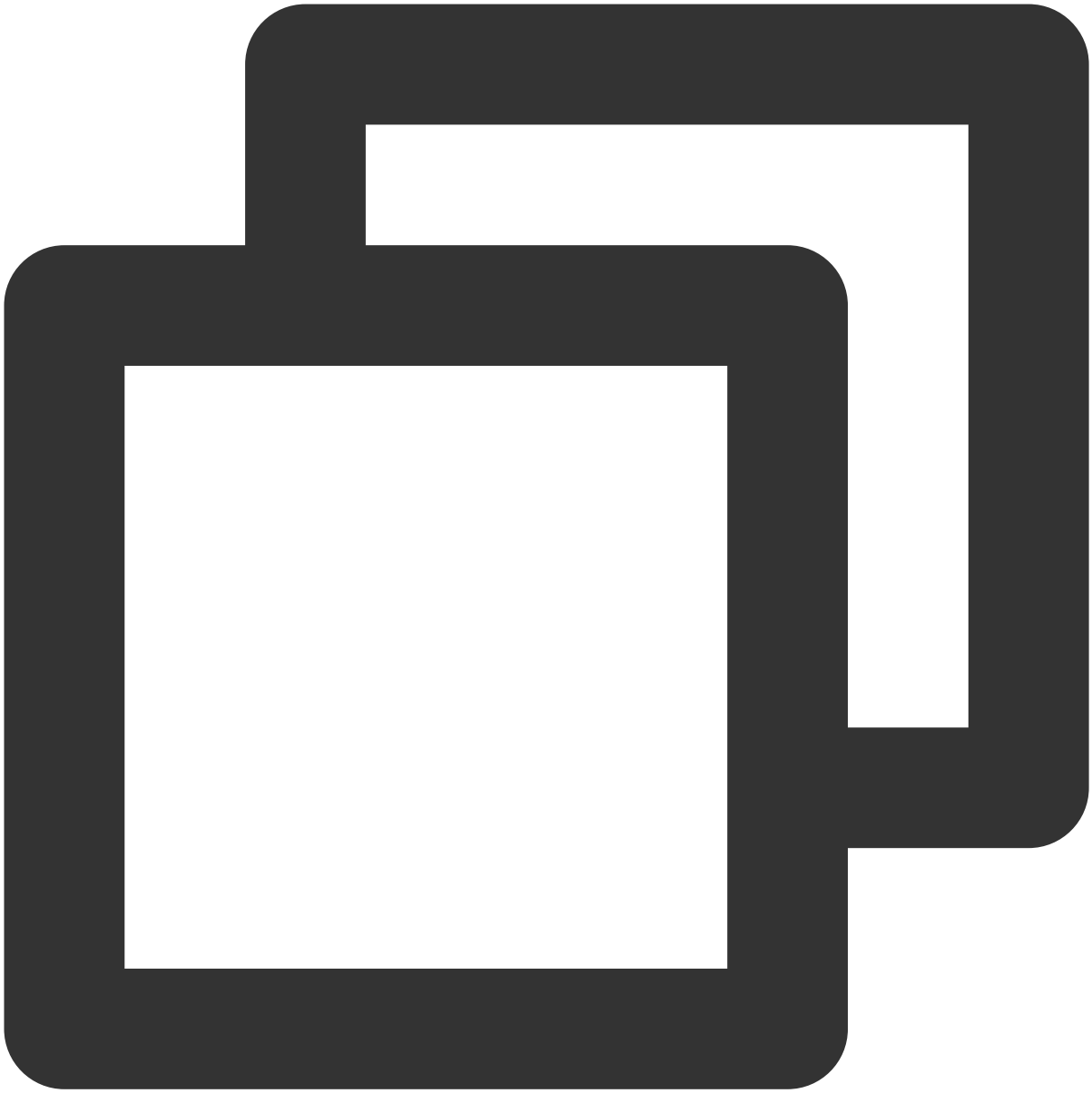
## Directions

**Note:**
CDWPG supports only CSV and GZIP but not ORC and Parquet formats.
The efficiency of importing COS data to CDWPG depends on the number of files, which is recommended to be N times the number of compute nodes in CDWPG.
1. Enable EMR's capability to read and write COS data.
First, you need to ensure that EMR is able to read and write COS data. You can click **Enable** COS when creating an EMR instance.
2. Create a Hive local table and write data into it.



```
create table hive_local_table(c1 int, c2 string, c3 int, c4 string);
insert into hive_local_table values(1001, 'c2', 99, 'c4'),(1002, 'c2', 100, 'c4'),(
```

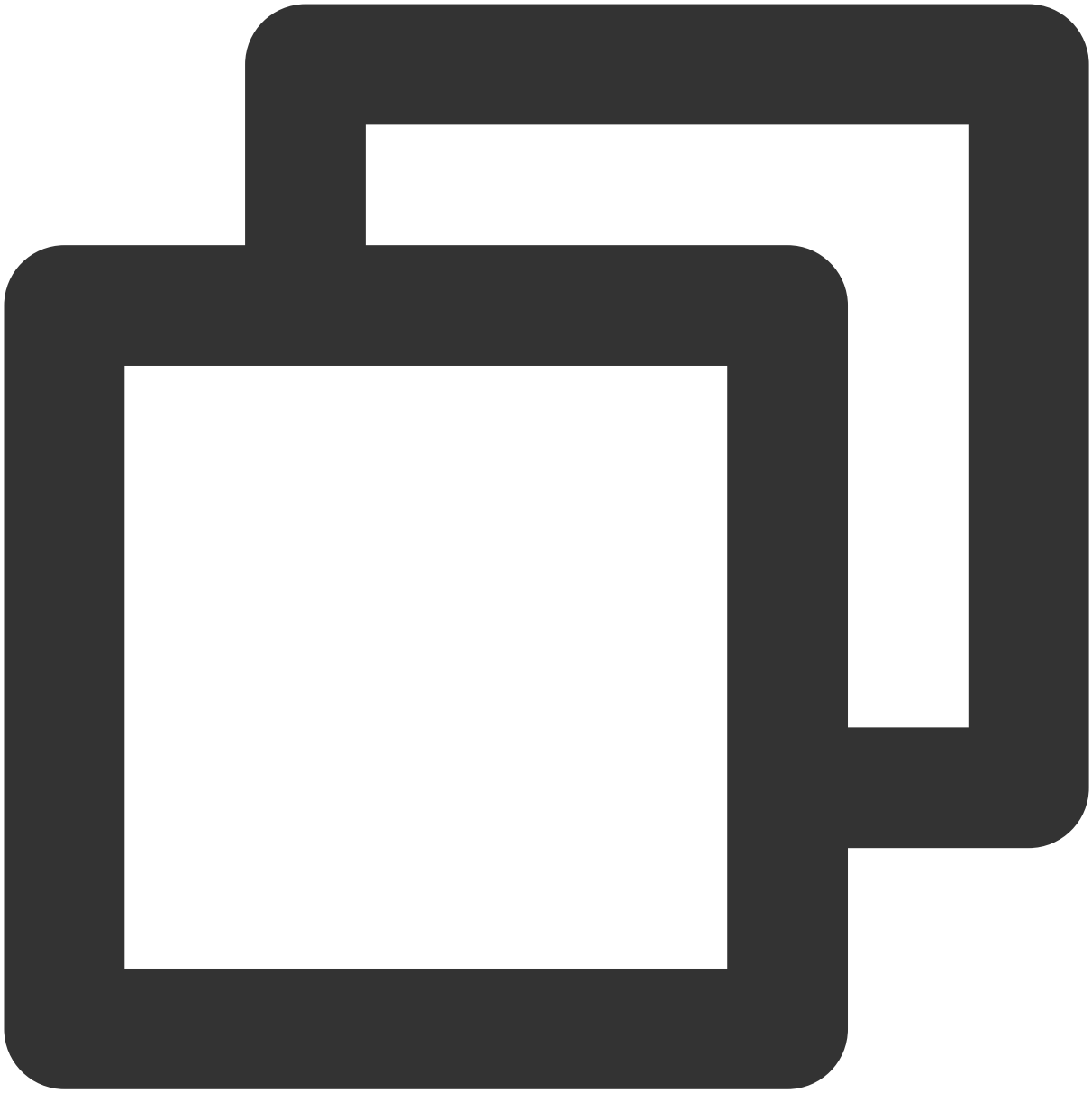
3. Create a Hive COS external table.



```
create table hive_cos_table(c1 int, c2 string, c3 int, c4 string)
row format delimited fields terminated by ','
LINES TERMINATED BY '\\n'
stored as textfile location 'cosn://{bucket_name}/{dir_name}';
```

For more information, see Creating Databases Based on COS.

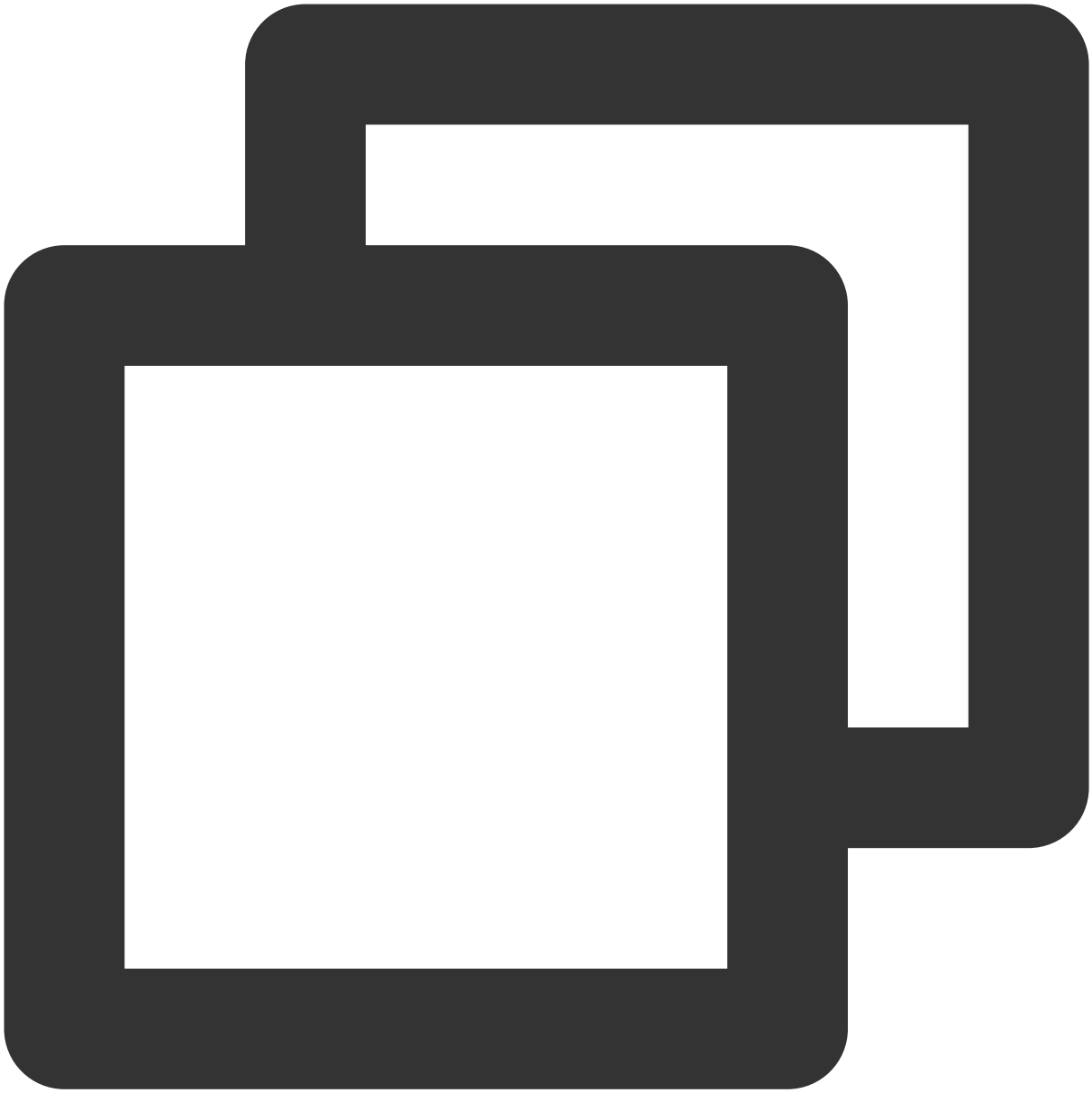
4. Import the local data into COS.



```
insert into hive_cos_table select * from hive_local_table;
```

After successful write, you can see the file in the corresponding COS directory.

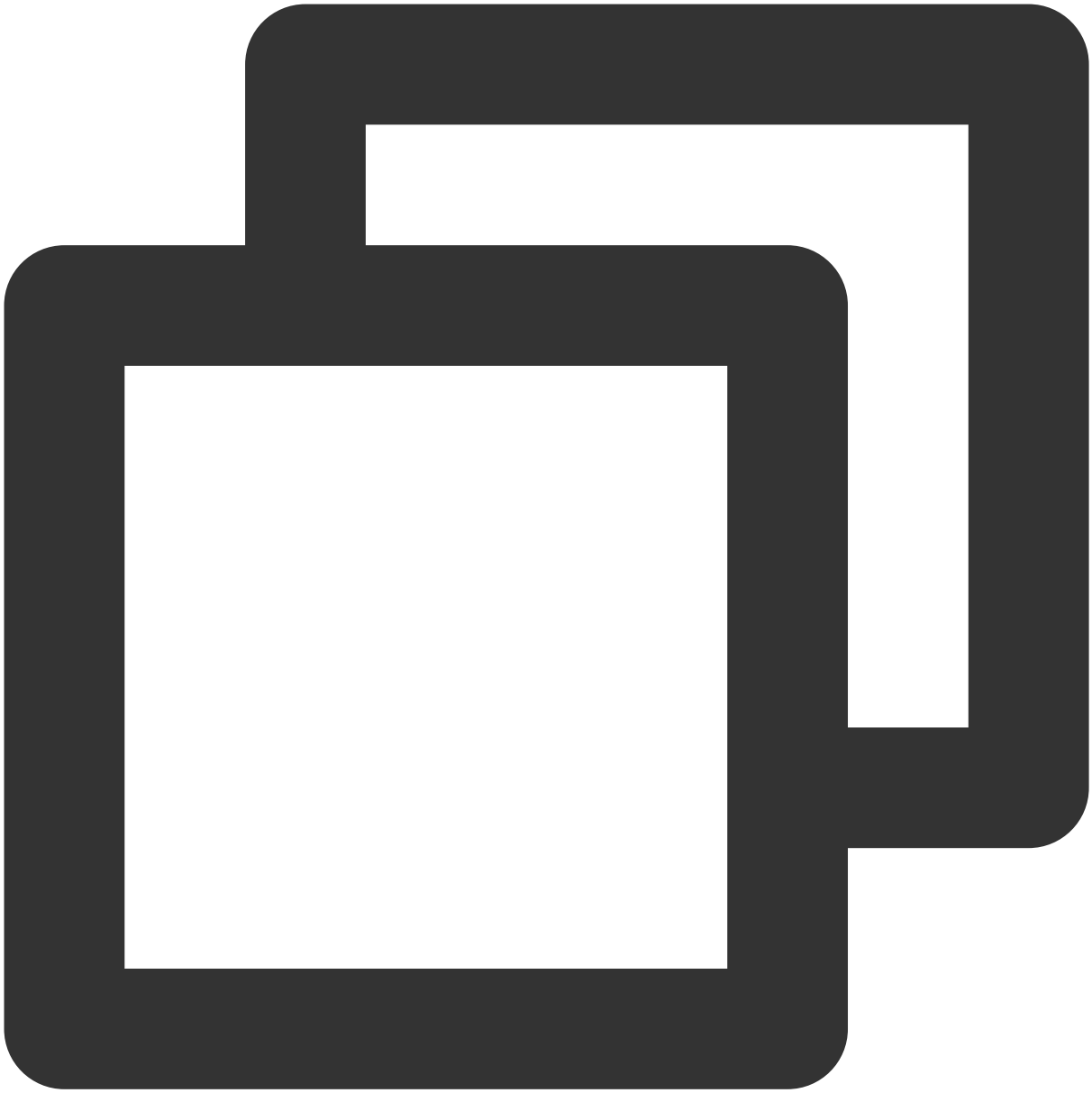
5. Create a COS external table in CDWPG.



```
CREATE READABLE EXTERNAL TABLE  snova_cos_table (c1 int, c2 varchar(32), c3 int, c4
LOCATION('cos:// {BUCKET}-{APPID}.cos.{REGION}.myqcloud.com/{PREFIX} secretKey=****
FORMAT 'csv';
```

For more information, see Importing and Exporting COS Data at High Speed with External Table.

6. Create a local table in CDWPG and import data into it.



```
create table snova_local_table(c1 int, c2 text, c3 int, c4 text);
insert into snova_local_table select * from snova_cos_table;
```

# Implementing CDWPG UPSERT with Rule

Last updated：2024-02-02 15:27:57

## Background

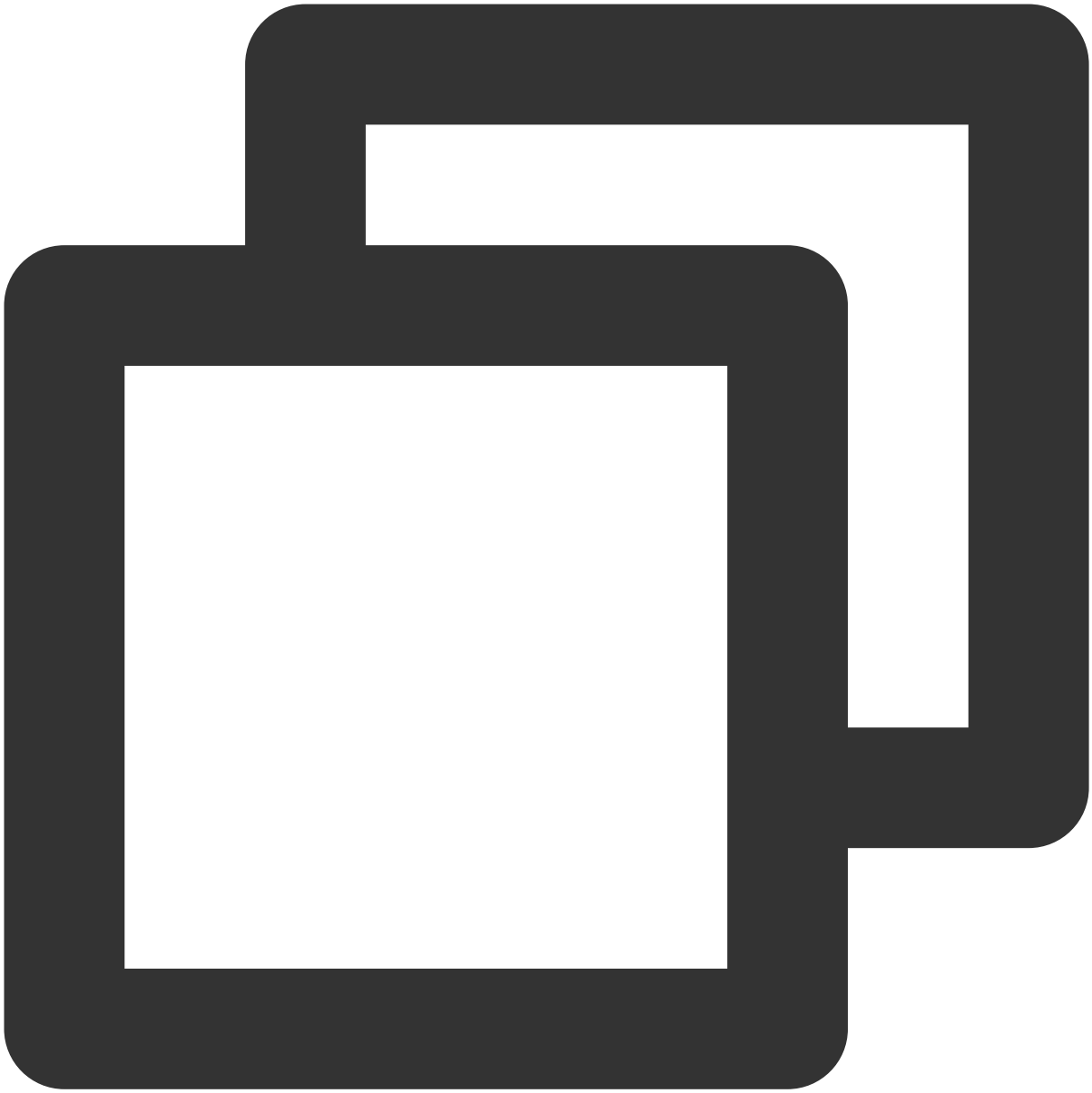
The underlying structure of CDWPG is built on Greenplum 6. The PostgreSQL kernel is v9.4, and its `INSERT .. ON CONFLICT` feature cannot be well supported for now. This document describes how to use a PostgreSQL rule to `UPSERT`, as it requires a different method.

## Rule Overview

The PostgreSQL rule system allows one to define an alternative action to be performed on insertions, updates, or deletions in database tables. Roughly speaking, a rule causes additional commands to be executed when a given command on a given table is executed. Alternatively, an `INSTEAD` rule can replace a given command by another, or cause a command not to be executed at all. Rules are used to implement table views as well. A rule is really a command transformation mechanism, or command macro. The transformation happens before the execution of the command starts.
For more information, see CREATE RULE.

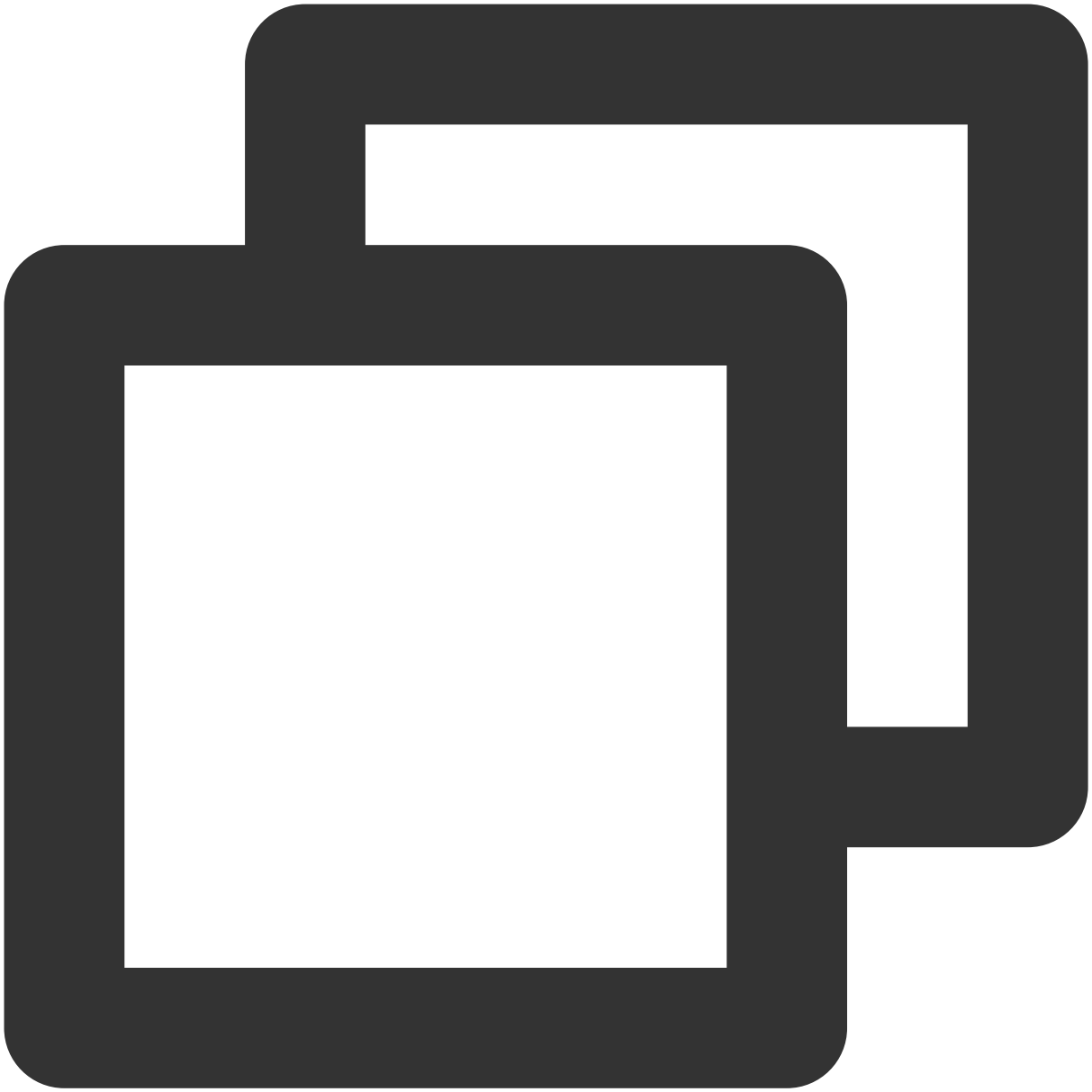
## `UPSERT` Rule

To implement an `UPSERT` operation, you need a rule that determines whether a corresponding record already exists during `INSERT`, allows for `UPDATE` if yes, and allows for `INSERT` if no.
The following database instance is used as an example:
Create a test database.



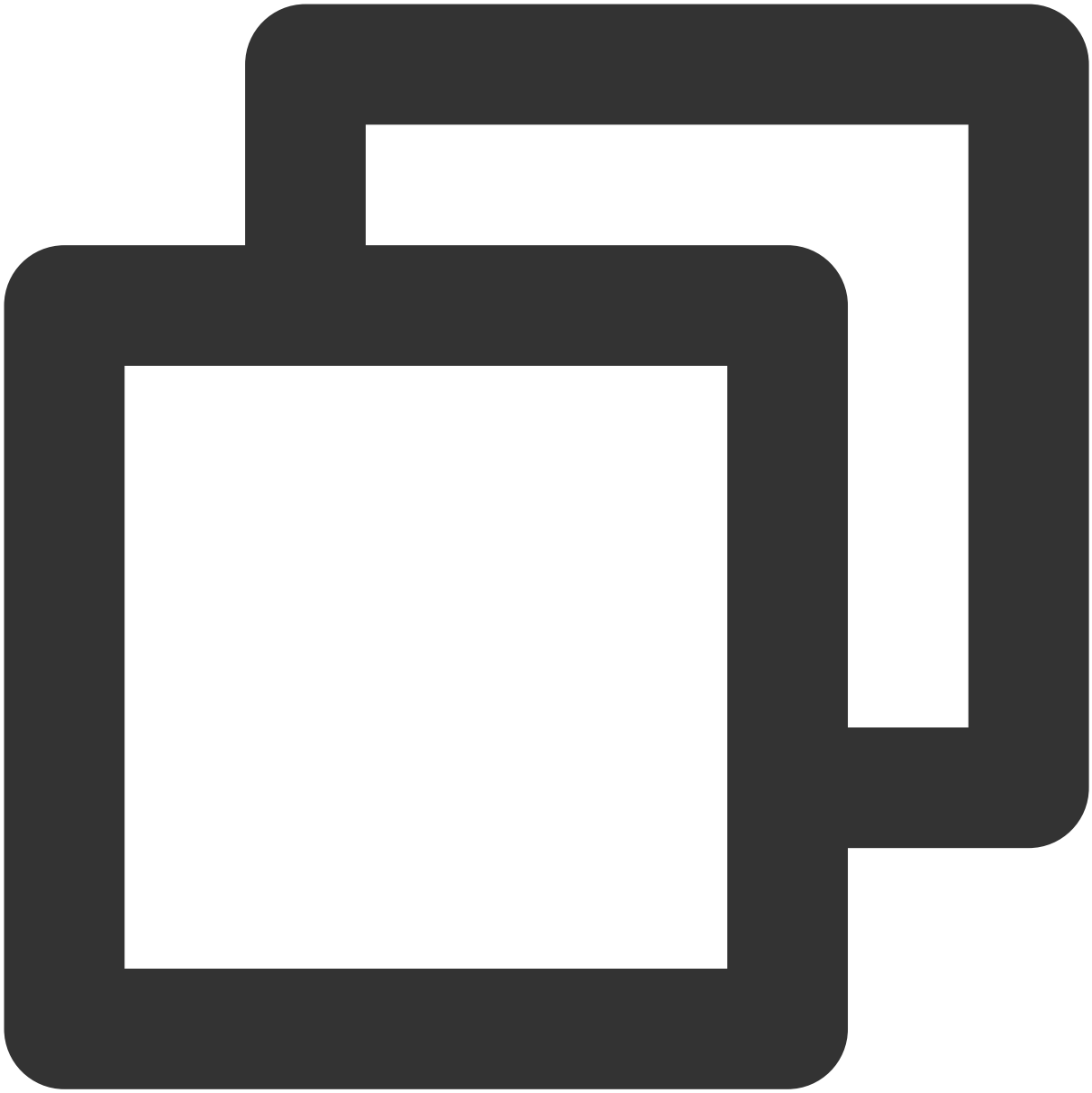
```
CREATE TABLE my_test(
   id integer,
   num1 integer,
   num2 decimal,
   str1 varchar(20),
   str2 text,
   PRIMARY KEY(id)
) distributed by (id);
```

Then, add a rule to the table.



```
create rule r1 as on insert to my_test where exists (select 1 from e t1 where t1.id
```

This rule is for the `INSERT` operation. If the `id` in the new `INSERT` statement already exists, the original data will be updated with the value inside the new `INSERT` , i.e., the `NEW.XXX` that can be seen after the operation. In this case, no errors will be reported due to the primary key constraint, and the `UPDATE` operation will be performed.



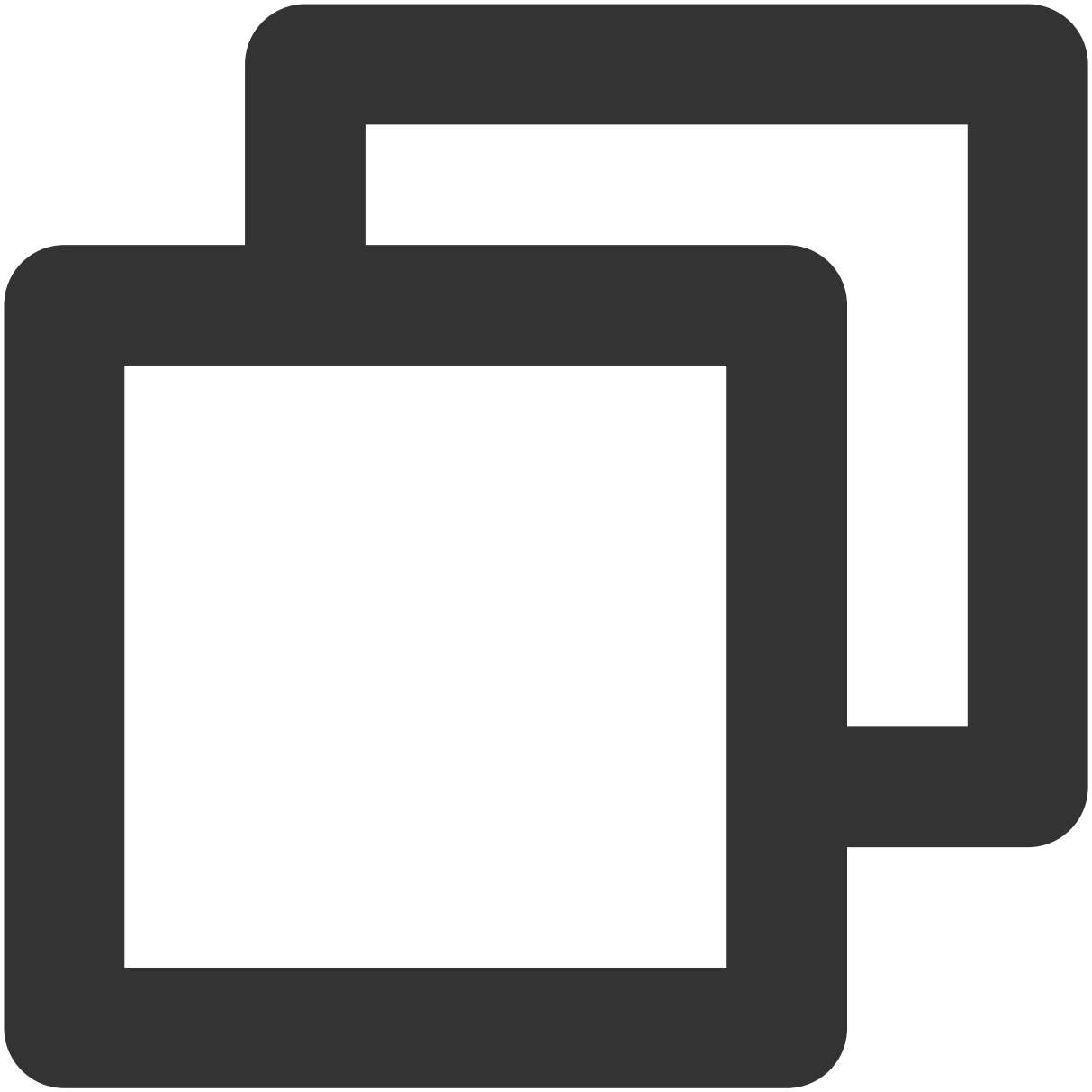
```
\\d my_test
                                  Table "public.my_test"
 Column |         Type          | Collation | Nullable |                Default
--------+-----------------------+-----------+----------+-----------------------------
 id     | integer               |           | not null | nextval('my_test_id_seq'::
 num1   | integer               |           |          |
 num2   | numeric               |           |          |
 str1   | character varying(20) |           |          |
 str2   | text                  |           |          |
Indexes:
    "my_test_pkey" PRIMARY KEY, btree (id)
```

```
Rules:
    r1 AS
    ON INSERT TO my_test
   WHERE (EXISTS ( SELECT 1
     FROM my_test my_test_1
     WHERE my_test_1.id = new.id
     LIMIT 1)) DO INSTEAD  UPDATE my_test SET num1 = new.num1, num2 = new.num2, str
```

## Notes

The rule is subject to the following limits:

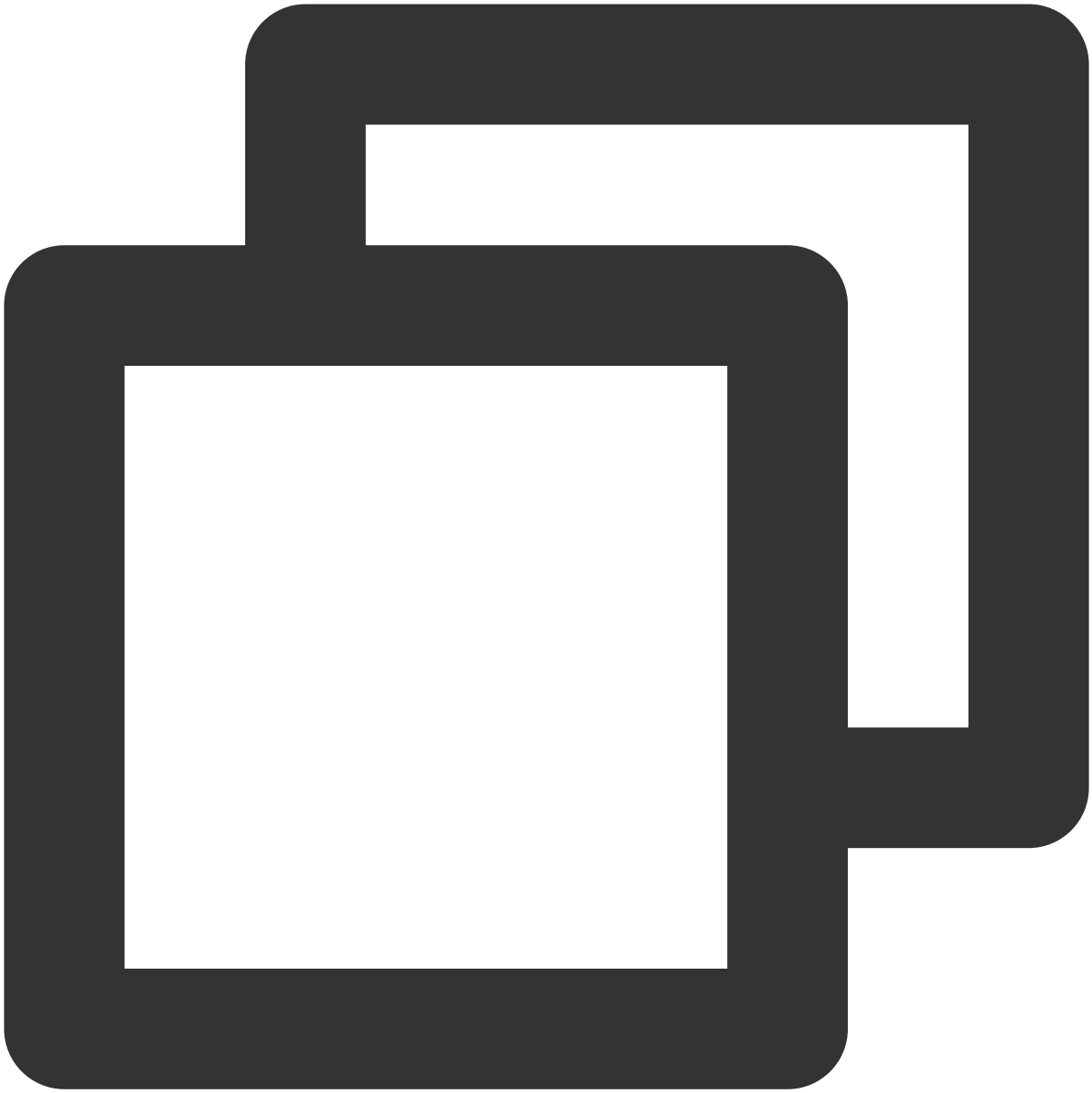
1. Bulk insert may not be proper. If the statement does not set a unique constraint or primary key constraint, duplicate data may be generated during bulk insert. Therefore, you should avoid using bulk insert. When using it, avoid determining whether the fields of `UPSERT` are not duplicated or add unique constraints to the fields that need to be determined. As shown in the following example, if there is no primary key constraint on the `id`, there may be duplicate data after execution.



```
insert into my_test (id,num1,num2,str1,str2)values(1,2,1.0,'111','555'),(1,3,2.0,'1
```

2. The rule does not support the `COPY` statement, which may also cause duplicate data just like bulk insert.

3. When you set the `UPDATE` rule, if the rule usage is configured, but the `INSERT` statement does not pass in the `num1` and `num2` fields, these two fields will be null after `UPDATE`, resulting in the loss of the original data.



```
update my_test set num1=NEW.num1,num2=NEW.num2,str1=NEW.str1,str2=NEW.str2
```