Data Lake Compute

SQL Statement

Product Documentation

Contents

SQL Statement
SuperSQL Statement
Overview of SuperSQL Statement

Last updated：2024-08-07 17:09:16

DLC can run almost seamlessly on the DLC Serverless Spark and DLC Serverless Presto engines through a set of standard SQL statements. Statements and functions related to metadata and analysis are basically compatible with Hive and Spark statements, and custom functions are supported.
For supported system built-in functions, see Unified Functions. If you need to use Presto built-in functions, see Presto Built-in Functions.
If you need to query and analyze data in external Iceberg tables on DLC, some statements may differ from those for native tables. For details, see Iceberg External Tables vs. Native Tables Statement Differences.
The following table lists statements supported by DLC:

DDL Statements

Database-Related Statements

Purpose	Statement
Create a database.	CREATE DATABASE
Display all databases defined in this metadata.	SHOW DATABASES
View database attributes.	DESCRIBE DATABASE
Change database attributes.	ALTER DATABASE SET DBPROPERTIES
Change database storage location.	ALTER DATABASE SET LOCATION
Delete a database.	DROP DATABASE

Data Table-Related Statements

Purpose	Statement
Create a data table.	CREATE TABLE

Update table snapshot.	REPLACE TABLE AS SELECT
Query data table creation statements.	SHOW CREATE TABLE
Query table attributes.	SHOW TBLPROPERTIES
Query all tables in database.	SHOW TABLES
View column information and metadata information on the data table.	DESCRIBE TABLE
Query column information on the data table.	SHOW COLUMNS IN TABLE
Add a column to the data table.	ALTER TABLE ADD COLUMNS
Add a column to the data table.	ALTER TABLE ADD COLUMN AFTER/FIRST
Chang a field name.	ALTER TABLE ... RENAME COLUMN
Delete a field in the data table.	ALTER TABLE DROP COLUMN
Add a partition to the data table.	ALTER TABLE ADD PARTATION
List table partitions.	SHOW PARTITIONS
Delete a partition of the data table.	ALTER TABLE DROP PARTITION
Add a partition field to the Iceberg table.	ALTER TABLE ADD PARTITION FIELD
Delete a partition field of the Iceberg table.	ALTER TABLE DROP PARTITION FIELD
Change data table attributes.	ALTER TABLE SET TBLPROPERTIES
Change data table storage location.	ALTER TABLE SET LOCATION
Change the sorting method of data in the table.	ALTER TABLE ... WRITE ORDERED BY
Change data distribution policy of partitioned table.	ALTER TABLE ... WRITE DISTRIBUTED BY PARTITION
Add identifier fields attribute.	ALTER TABLE ... SET IDENTIFIER FIELDS
Delete identifier fields attribute.	ALTER TABLE ... DROP IDENTIFIER FIELDS
Update partition information.	MSCK REPAIR TABLE

Perform statistics on data table.	ANALYZE TABLES
Delete metadata table.	DROP TABLE
Display logical or physical plan for executing SQL statements.	EXPLAIN
Call table-related stored procedure.	CALL STATEMENT

Statements Related to View

Purpose	Statement
Create a view based on results of SELECT statement.	CREATE VIEW AS
Query view in the database.	SHOW VIEWS
Check view attributes.	DESCRIBE VIEW
Display view creation statements.	SHOW CREATE VIEW
Check view columns.	SHOW COLUMNS IN VIEW
Change view name.	ALTER VIEW RENAME TO
Change view attributes.	ALTER VIEW SET TBLPROPERTIES
Delete view.	DROP VIEW

Statements Related to Function

Purpose	
Create a function.	CREATE FUNCTION
View function creation statements	SHOW FUNCTION
Delete a function.	DROP FUNCTION

DML Statements

Purpose	
Insert a row of data.	INSERT STATEMENT

Replace a row of data.	INSERT OVERWRITE
Update data by row (for replacing INSERT OVERWRITE).	MERGE INTO
Query Iceberg table metadata.	TABLE METADATA
Insert query results to the data table.	INSERT INTO
Delete data of Iceberg table.	DELETE STATEMENT
Update the specified row.	UPDATE

DQL Statement

Purpose	
Query data.	SELECT STATEMENT

For related reserved words, see Reserved Words.

Unified Statement
Common Data Types

Last updated：2024-08-07 17:09:43

Data Type Category	Data Type	Description
Number	ByteType:BYTE, TINYINT	1-byte signed integer. Range of the number: -128 to 127.
	ShortType:SHORT, SMALLINT	2-byte signed integer. Range of the number: -32768 to 3276.
	IntegerType:INT, INTEGER	4-byte signed integer. Range of the number: -2147483648 to 214748364.
	LongType:LONG, BIGINT	8-byte signed integer. Range: -9223372036854775808 to 922337203685477580.
	FloatType:FLOAT, REAL	4-byte single-precision floating-point number
	DoubleType:DOUBLE	8-byte double-precision floating-point number
	DecimalType:DECIMAL, DEC, NUMERIC	Signed decimal number with random precision
String	StringType:STRING	String value. Example: 'abc'
Byte	BinaryType:BINARY	Byte sequence value.
Boolean	BooleanType:BOOLEAN	Boolean value. Example: true/false
Time	DateType:DATE	Value consisting of the year, month, and day fields, without the time zone. Example: '2023-10-01'
	TimestampType:TIMESTAMP	Value consisting of the year, month, day, hour, minute, and second fields. Example: '2023-10-01 23:59:59'
Complex type	ArrayType:ARRAY<element_type>	Sequence value consisting of elements of the specified type.

		Example: array<int>
	MapType:MAP<key_type, value_type>	Value consisting of a group of key-value pairs. key_type indicates the type of keys, and value_type indicates the type of values. Example: map<string, int>
	StructType:STRUCT<field1_name: field1_type, field2_name: field2_type, ...>	Struct value consisting of fields Example: struct<id:int, grade:string>

DDL Statement
CREATE DATABASE

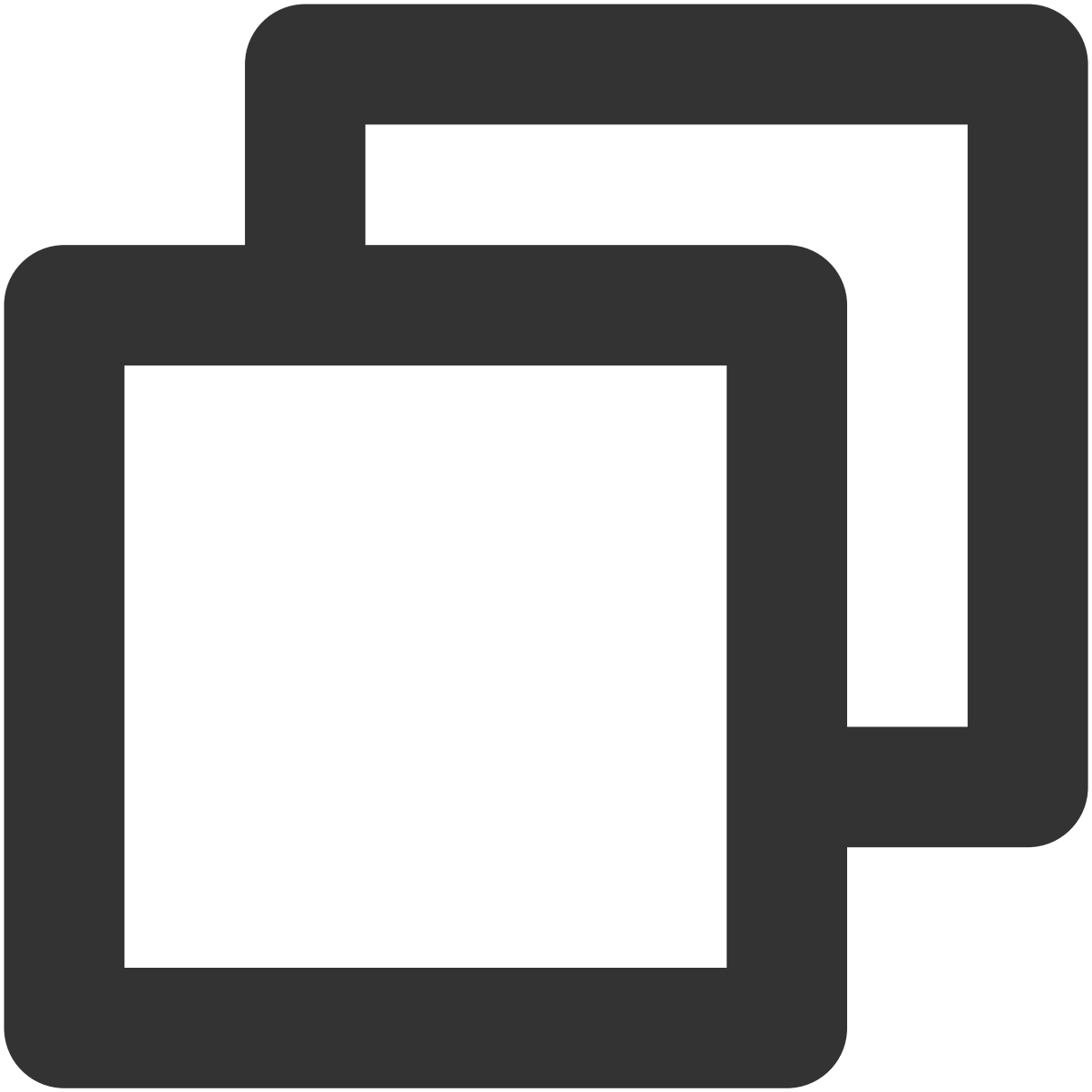
Last updated：2024-08-07 17:10:06

Description

Supported engines: Presto and SparkSQL

Purpose: Create a database, or specify or change the storage location of a table or partitioned table.

Statement



```
CREATE {DATABASE|SCHEMA} [IF NOT EXISTS] database_name
     [COMMENT 'database_comment']
     [WITH DBPROPERTIES ('property_name' = 'property_value') [, ...]]
```

Parameter

`DATABASE|SCHEMA` : Specify the database or schema.

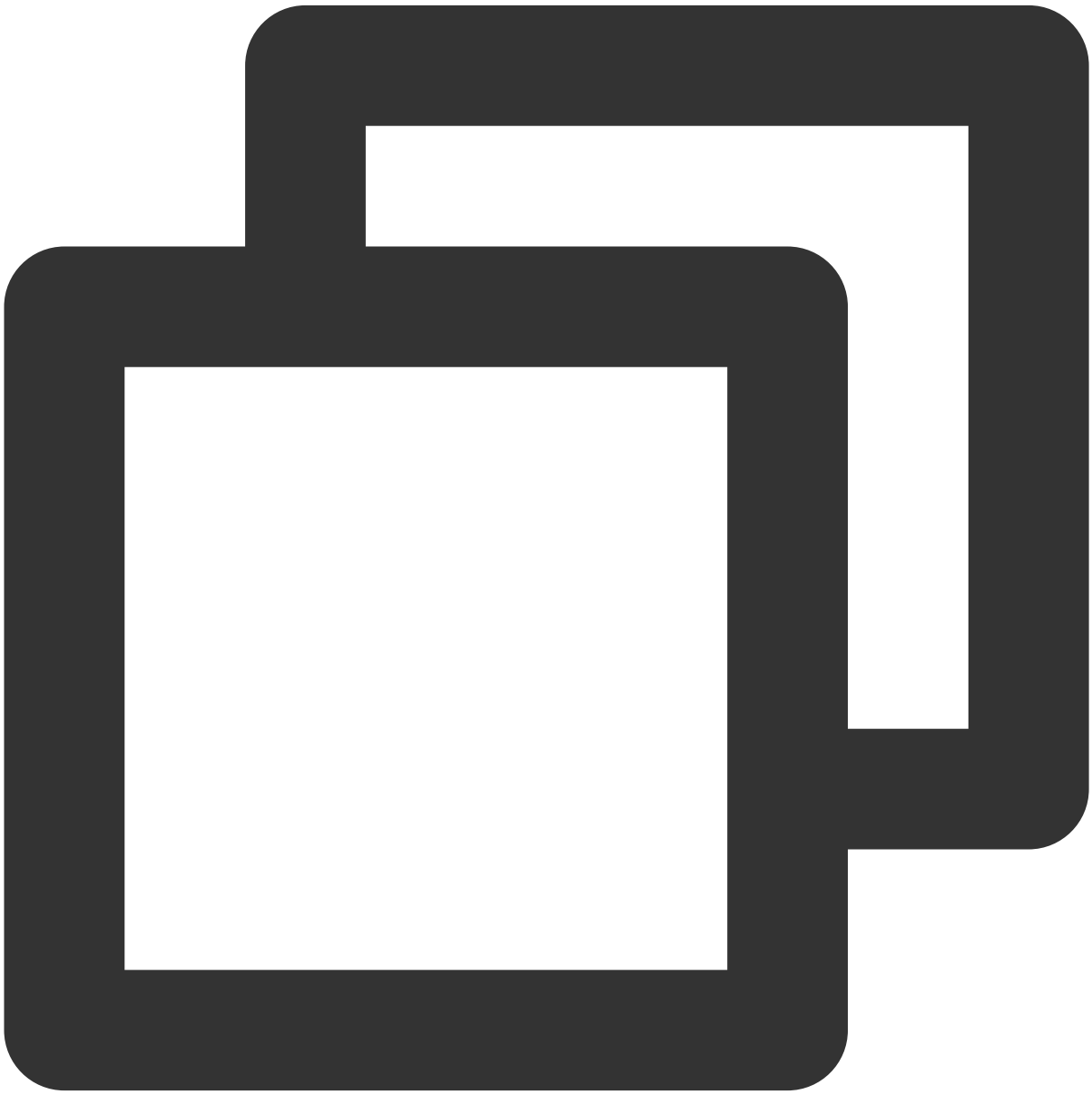
`database_name` : Database name.

`database_comment` : Database comment.

`[WITH DBPROPERTIES ('property_name' = 'property_value') [, ...]]` : Configure database parameters using `key-value` pairs.

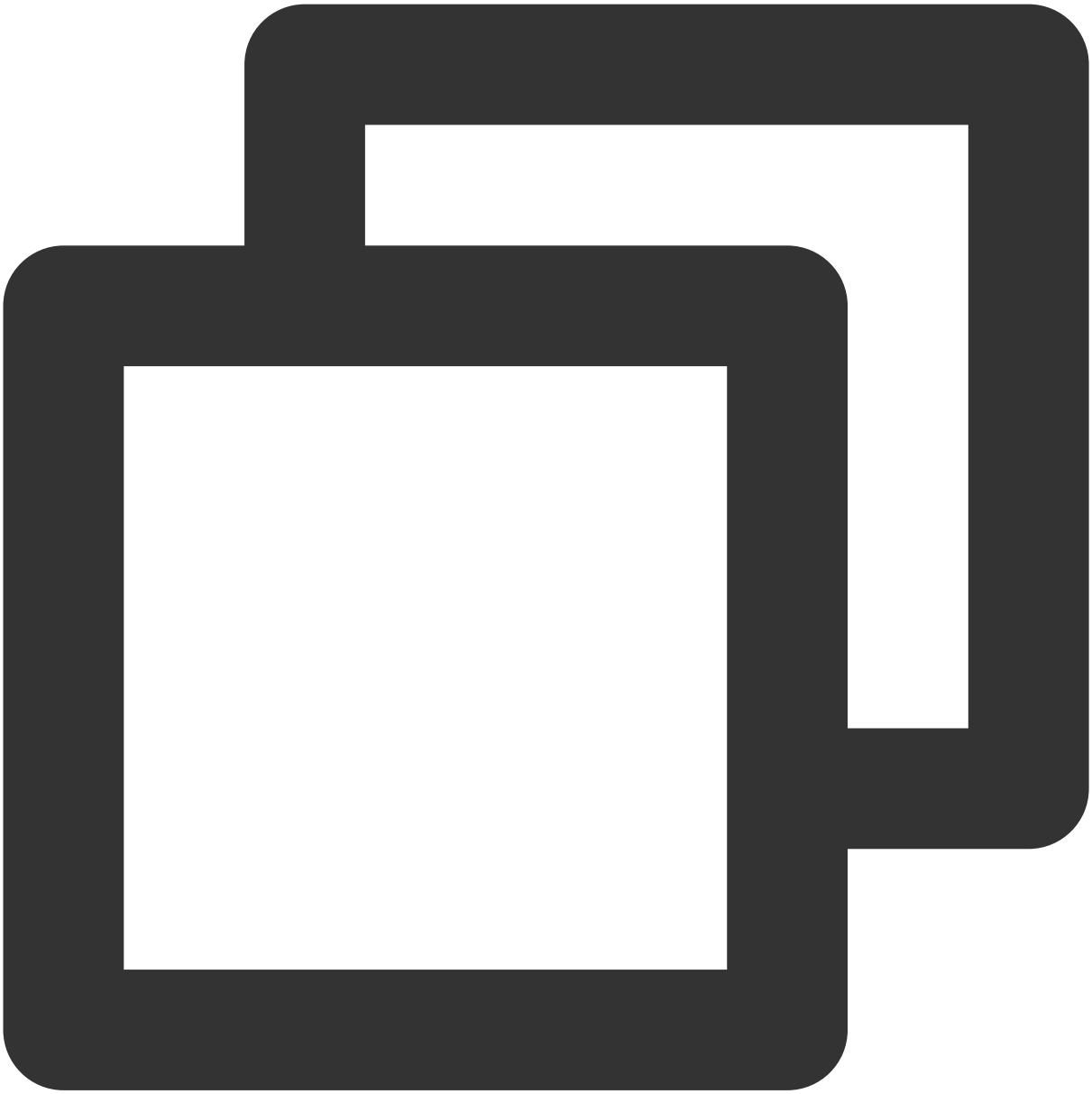
Example

Create database `db` only if database with same name doesn't exist.



```
CREATE DATABASE IF NOT EXISTS db;
```

Create database `db` only if database with same name doesn't exist with `Comment` and `Database Properties`.



```
CREATE DATABASE db
COMMENT 'db1_name'
WITH DBPROPERTIES('k1' = 'v1','k2' = 'v2');
```

SHOW DATABASES

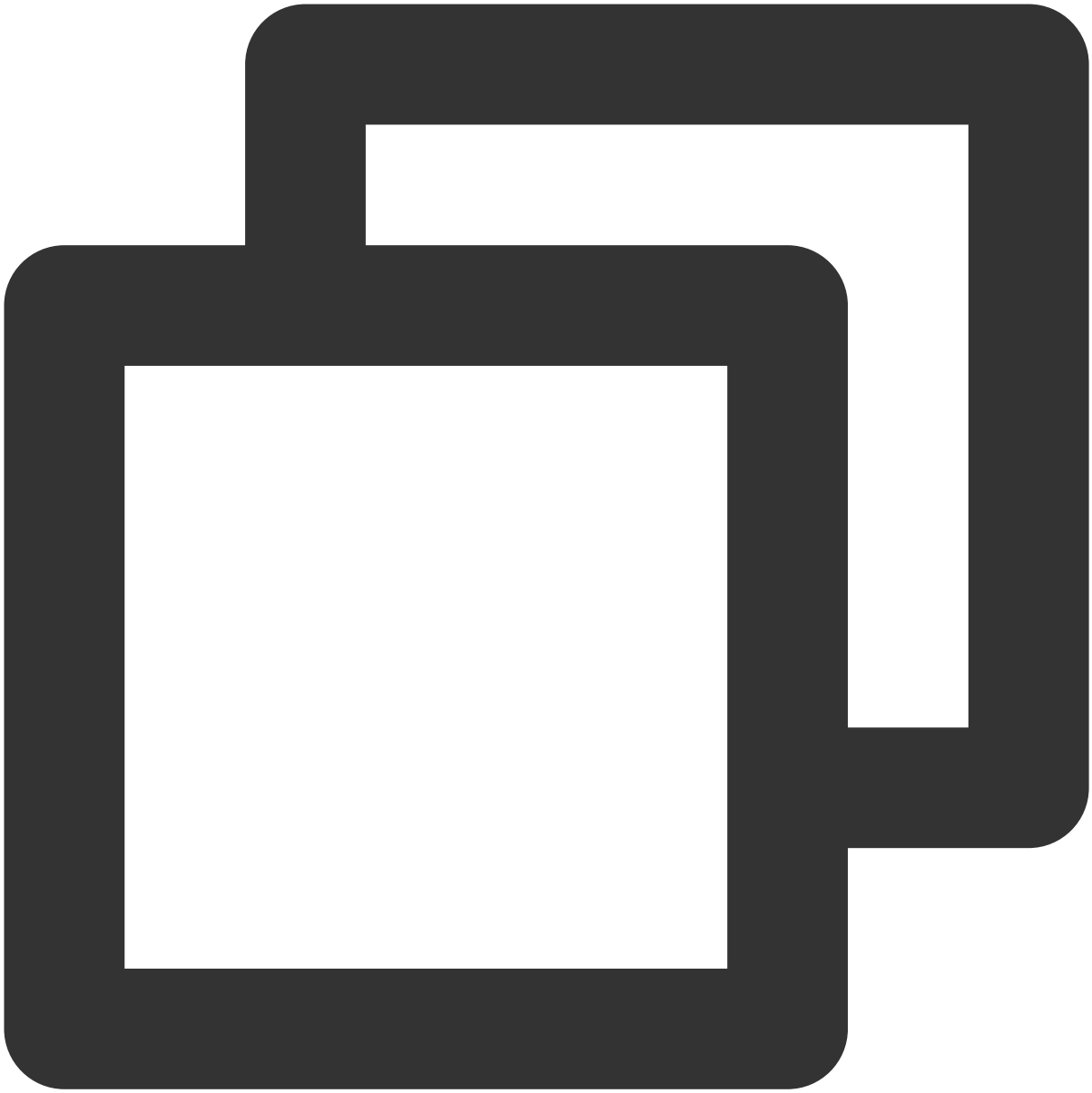
Last updated：2024-08-07 17:10:35

Description

Supported engines: Presto and SparkSQL

Purpose: List all databases defined in the metadata. DATABASES or SCHEMAS has the same effect.

Statement



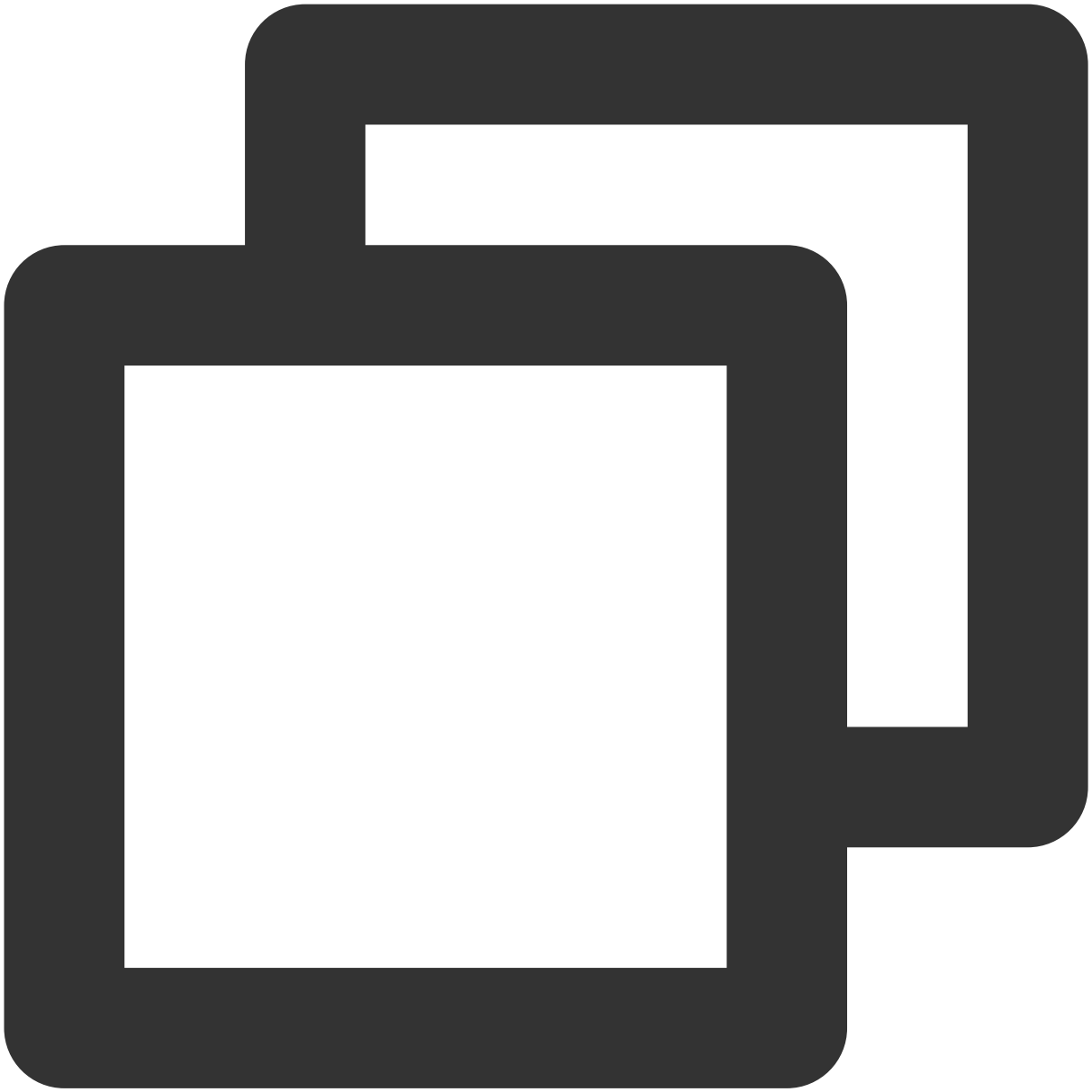
```
SHOW {DATABASES | SCHEMAS} [IN catalog_name] [LIKE 'regular_expression']
```

Parameter

`[IN catalog_name]` : Data source name.

`[LIKE 'regular_expression']` : Regular expression for filtering matching databases.

Example



```
SHOW DATABASES;
SHOW DATABASES LIKE '.*analytics';
SHOW DATABASES IN catalog1 LIKE '.*analytics';
```

DESCRIBE DATABASE

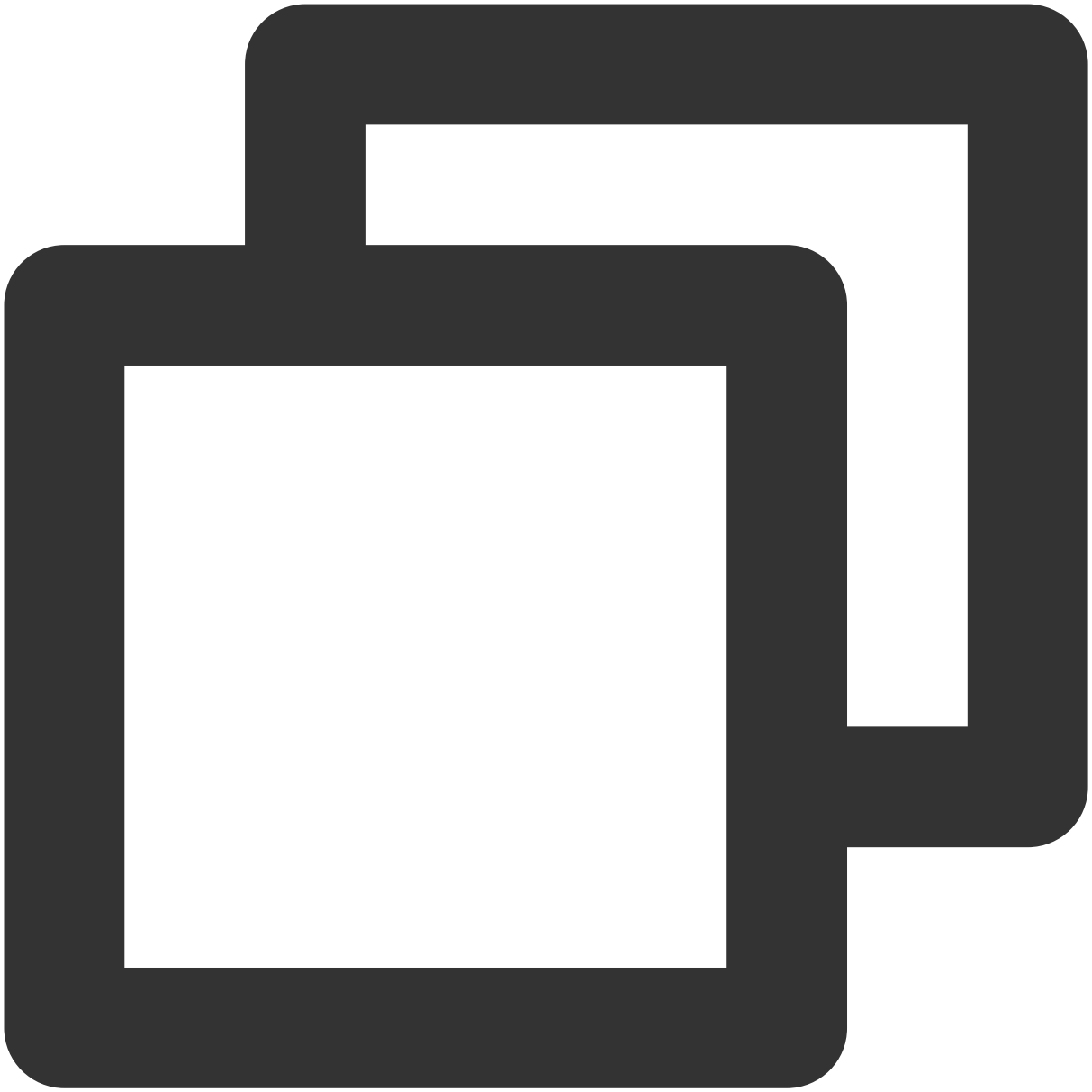
Last updated：2024-08-07 17:10:50

Description

Supported engines: Presto and SparkSQL

Purpose: View database attributes.

Standard Statement



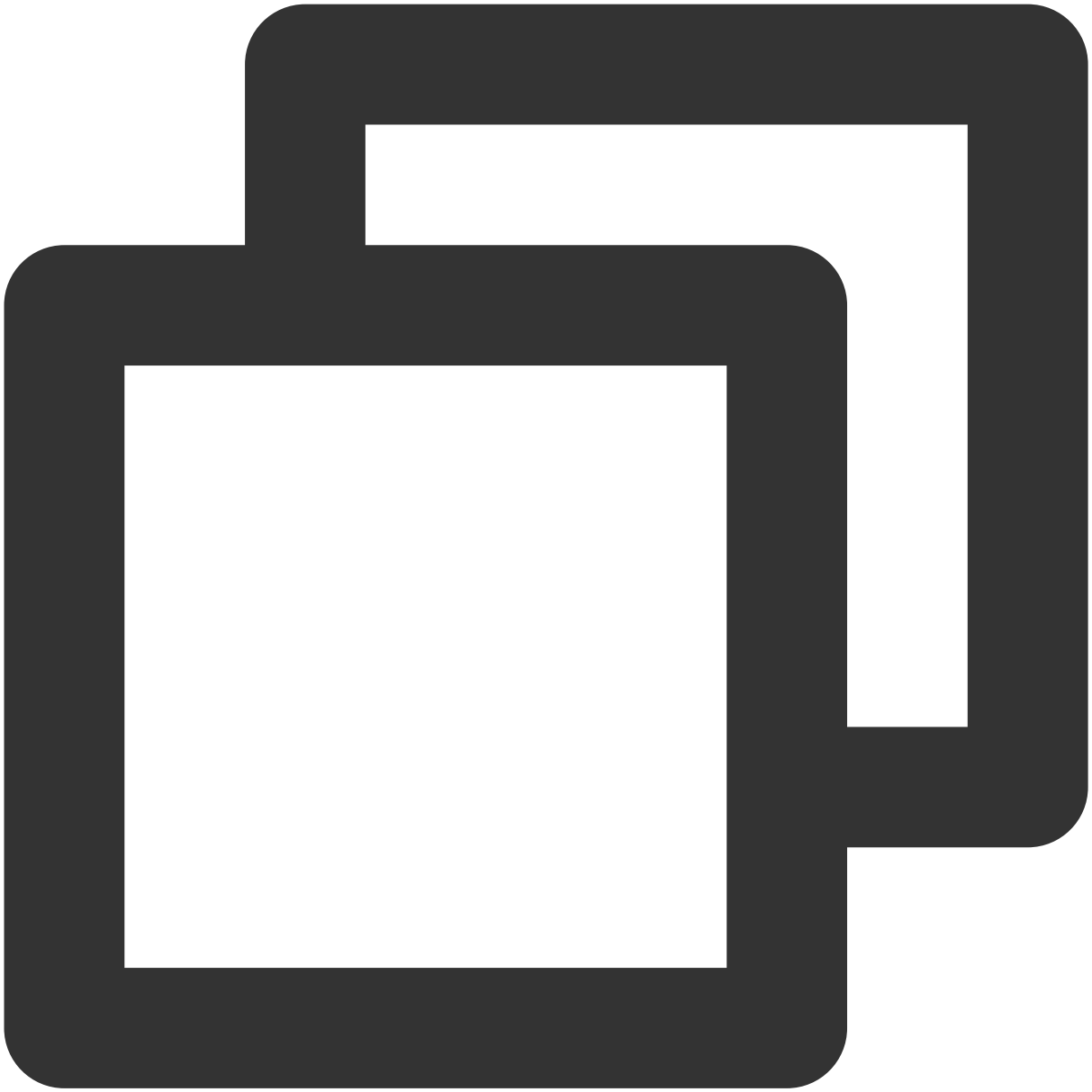
```
DESCRIBE SCHEMA|DATABASE [EXTENDED] DB_NAME
```

Parameter

`SCHEMA|DATABASE` : Specify the schema or database.

`EXTENDED` : Indicate whether the database is extended.

Example



```
DESCRIBE DATABASE db_name;
DESCRIBE DATABASE EXTENDED db_name;
```

ALTER DATABASE
ALTER DATABASE SET DBPROPERTIES

Last updated：2024-08-07 17:11:15

Description

Supported engine: Presto.

Purpose: Add one or more attributes to a database. If an attribute to be added already exists, the existing one will be overwritten.

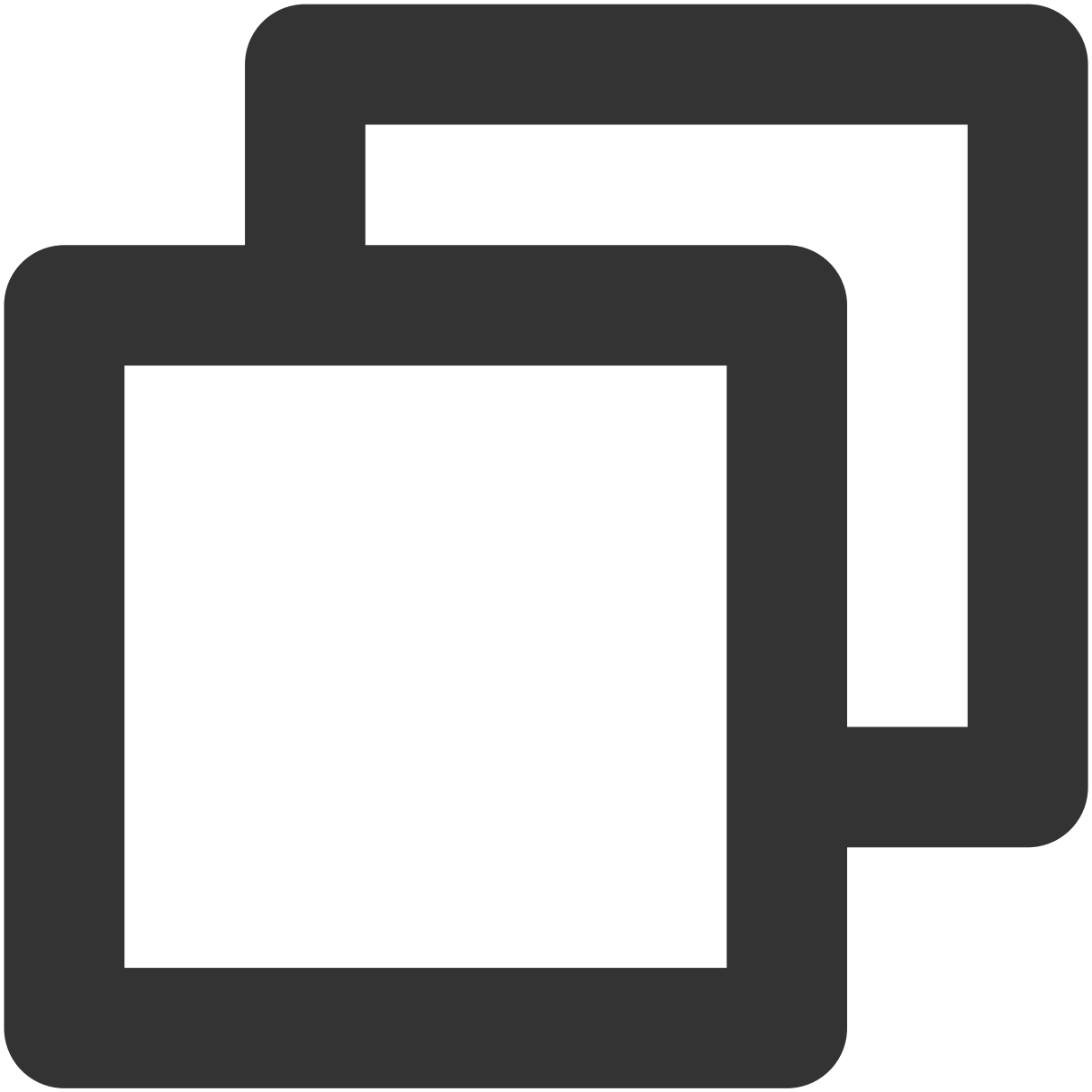
Standard Statement

```
ALTER (DATABASE|SCHEMA) database_name SET DBPROPERTIES (property_name=property_valu
```

Parameter

`database_name` : Database name.

Example



```
-- Alters the database to set properties `author`.
ALTER DATABASE product SET DBPROPERTIES ('author' = 'allen');
```

ALTER DATABASE SET LOCATION

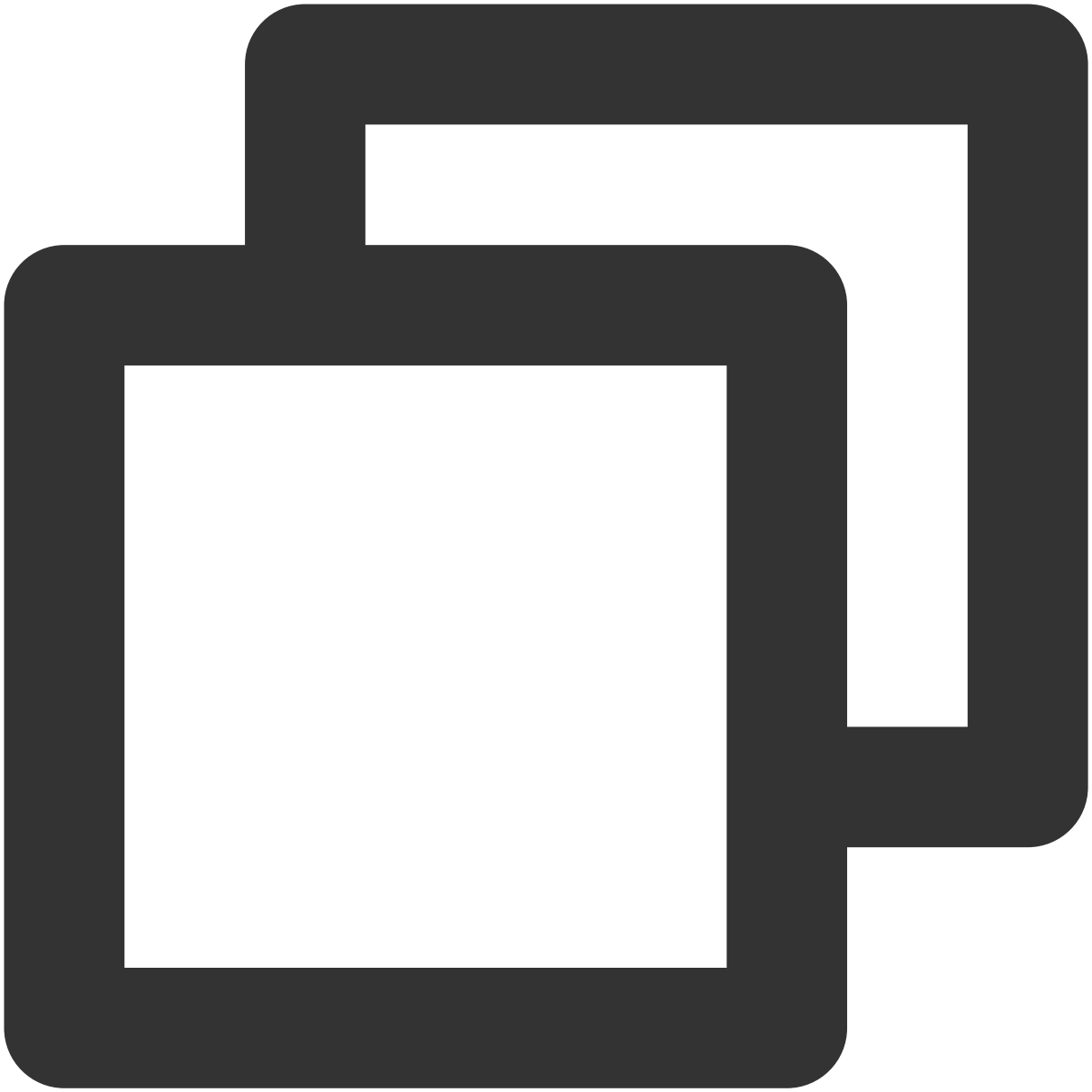
Last updated：2024-08-07 17:11:31

Description

Supported engine: Presto.

Purpose: Change database storage path.

Standard Statement

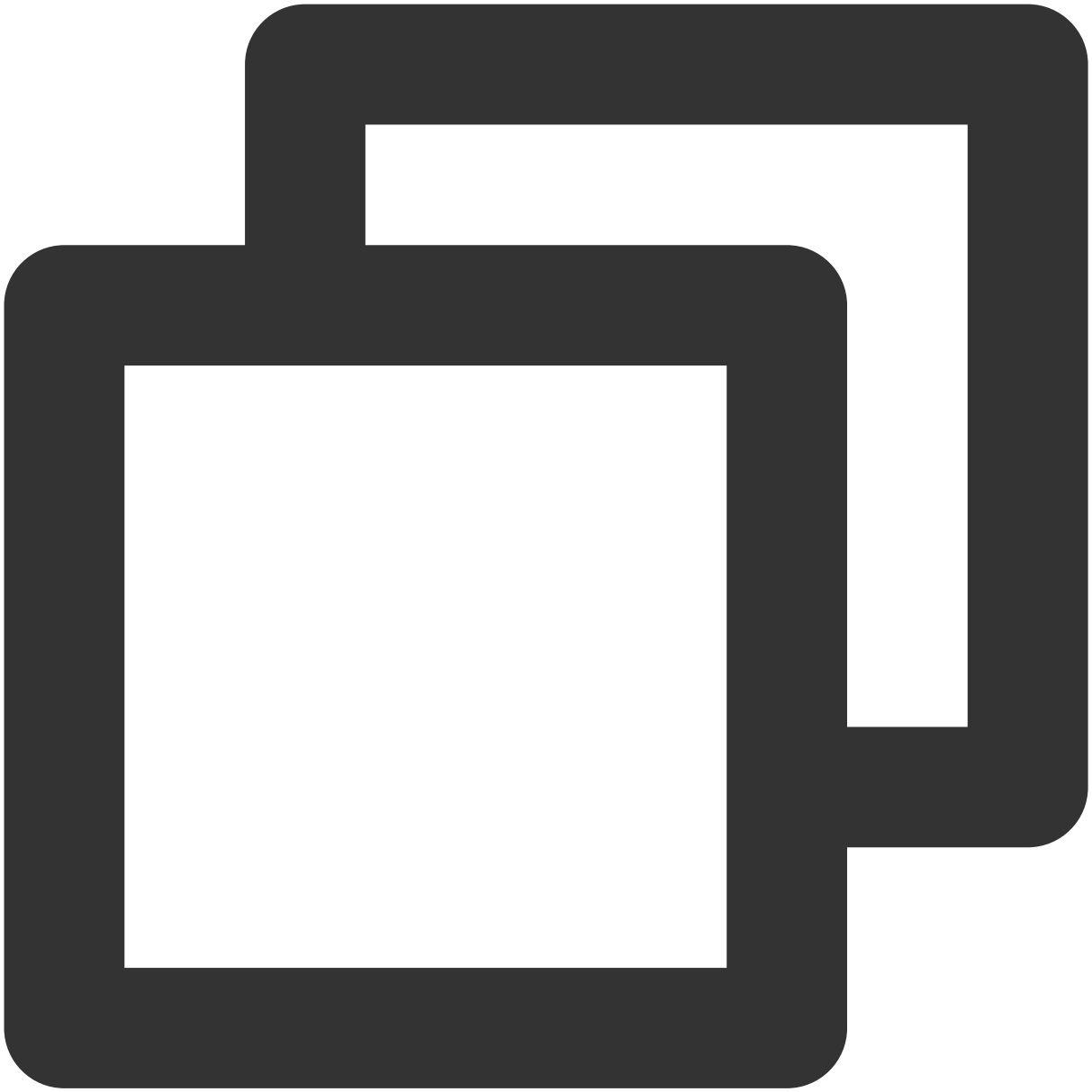


```
ALTER (DATABASE|SCHEMA) database_name SET LOCATION hdfs_path
```

Parameter

`[database_name]` : database name.

Example



```
ALTER DATABASE db01 SET LOCATION 'cosn:///new/path'
```

DROP DATABASE

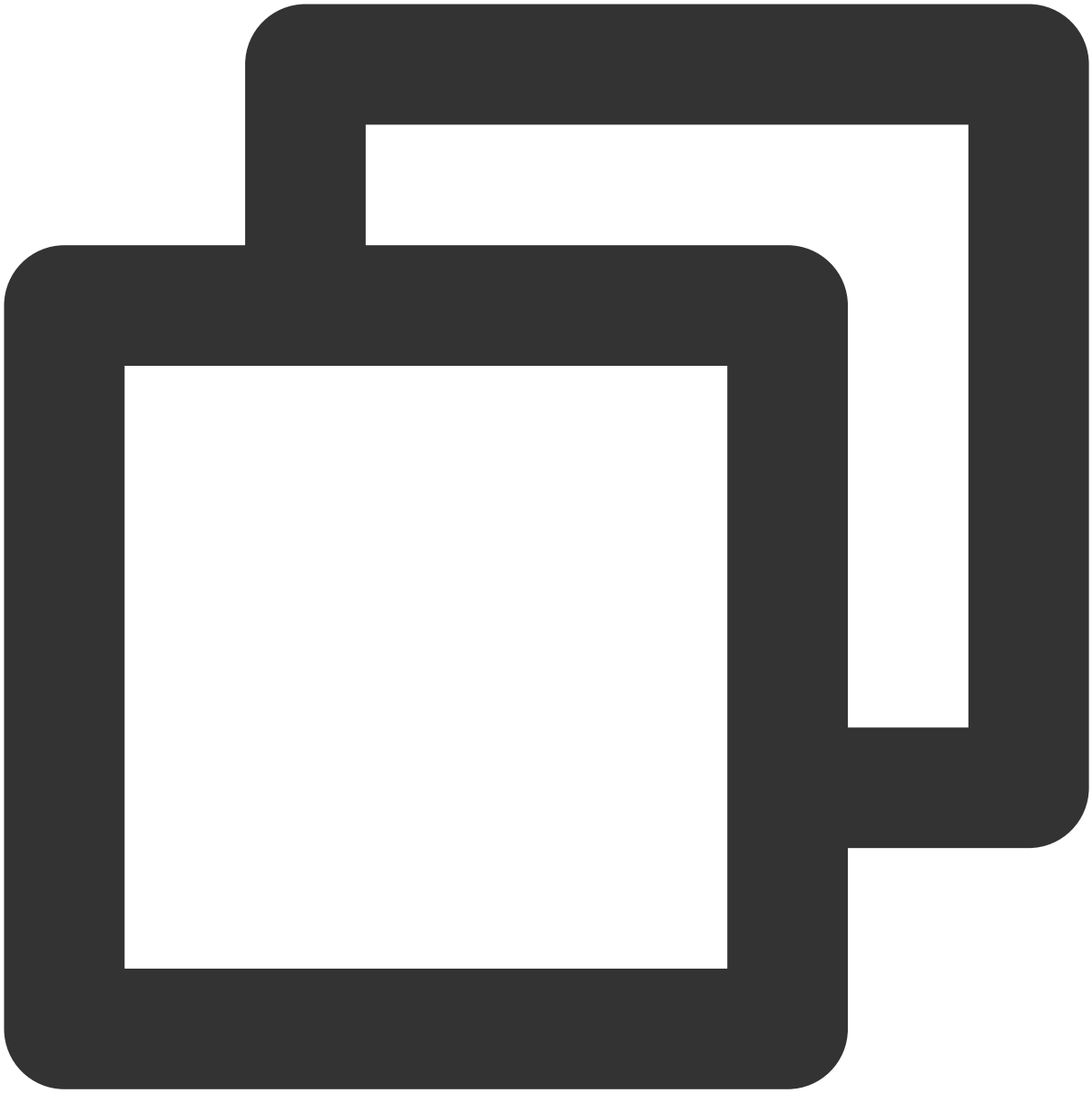
Last updated：2024-08-07 17:11:45

Description

Supported engines: Presto and SparkSQL

Purpose: Delete specified database.

Statement



```
DROP {DATABASE | SCHEMA} [IF EXISTS] database_name [RESTRICT | CASCADE]
```

Parameter

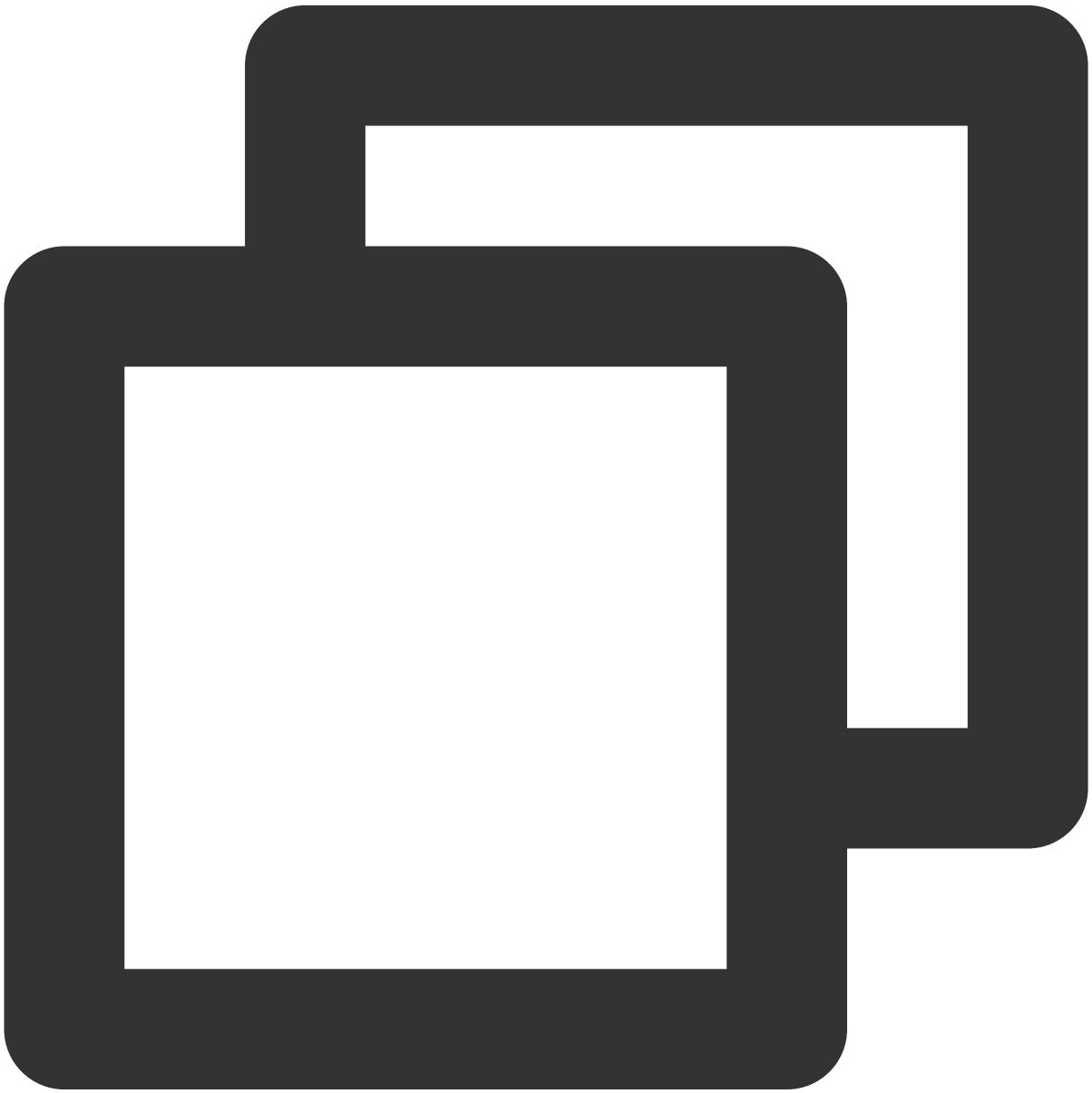
`DATABASE|SCHEMA` : Specify the database or schema.

`database_name` : Database name

`RESTRICT` : The database is not deleted if it contains tables. If not specified, the RESTRICT mode is used by default.

`CASCADE` : All databases and tables are deleted forcibly.

Example



```
-- Drop the database and it's tables
DROP DATABASE test CASCADE;
```

```
-- Drop the database using IF EXISTS
DROP DATABASE IF EXISTS test;
```

CREATE TABLE

Last updated：2024-08-07 17:12:15

Description

Supported engines: Presto and SparkSQL

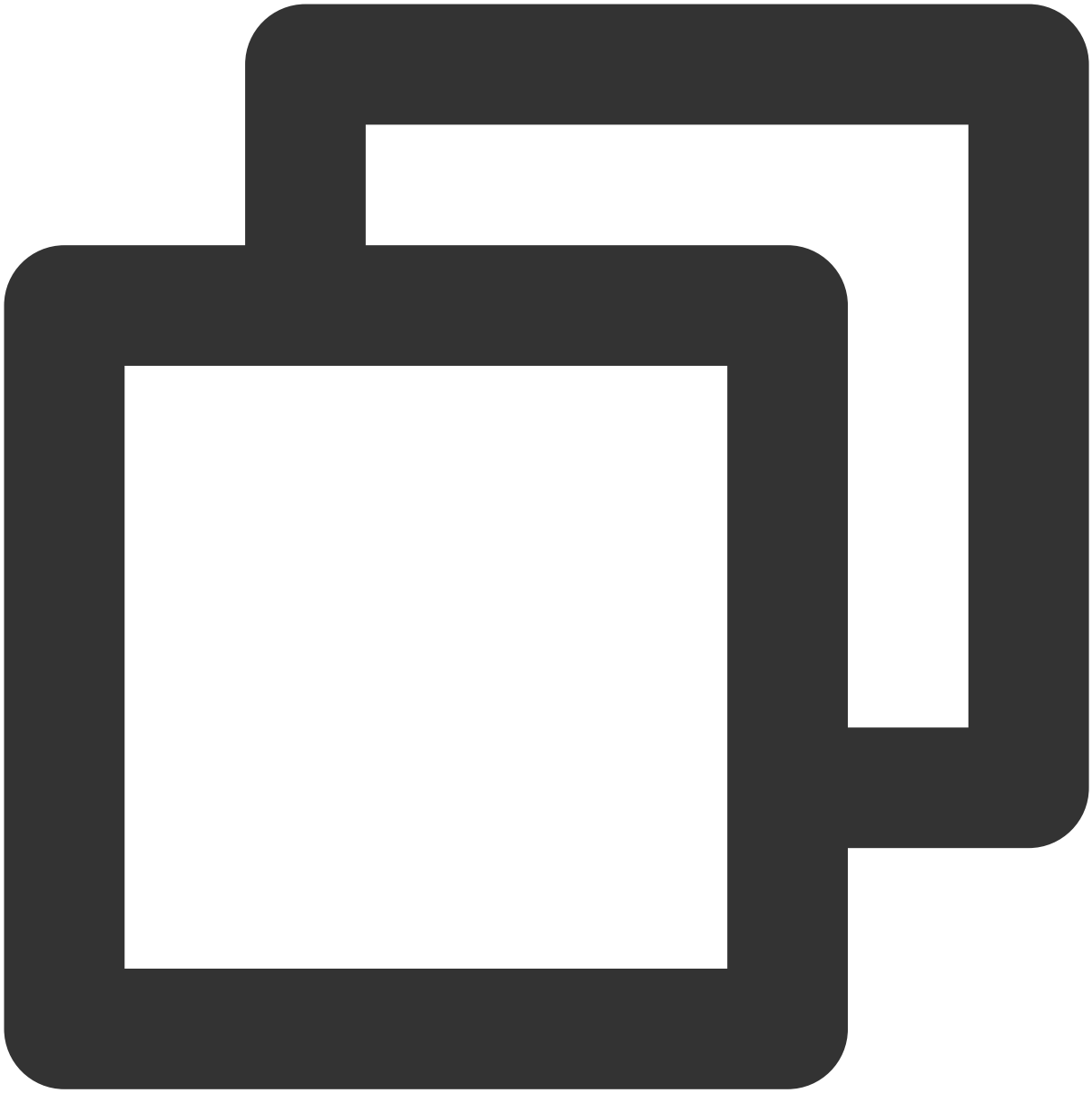
Applicable table: Native Iceberg tables and external tables

Purpose: Create a table with some attributes. CREATE TABLE AS can also be used.

Storage path for the created table: The path can be a COS directory but not a file.

Statement for External Table

Statement



```
CREATE TABLE [ IF NOT EXISTS ] table_identifier
    ( col_name[:] col_type [ COMMENT col_comment ], ... )
USING data_source
    [ COMMENT table_comment ]
    [ OPTIONS ( 'key1'='value1', 'key2'='value2' )]
    [ PARTITIONED BY ( col_name1, transform(col_name2), ... ) ]
    [ LOCATION path ]
    [ TBLPROPERTIES ( property_name=property_value, ... ) ]
```

Parameter

`USING data_source` : Data source used during table creation. Currently, the value of data_source can be CSV, ORC, PARQUET, and ICEBERG.

`table_identifier` : Table name in the three-part format. Example: catalog.database.table.

`COMMENT` : Table description

`OPTIONS` : Additional parameters of USING data_source, which are used to pass dynamic values during storage.

`PARTITIONED BY` : Create partitions based on specified columns.

`LOCATION path` : Storage path of the data table.

`TBLPROPERTIES` : A set of key-value pairs used to specify table parameters.

Detailed Description of USING and OPTIONS

USING CSV

USING ORC

USING PARQUET

Reference: Working with CSV.

Configurations supported by CSV data tables are as follows.

Key Supported by OPTIONS	Corresponding Default Value	Description
sep or delimiter	,	Delimiter between columns for storage in CSV files. The comma is used by default.
mode	PERMISSIVE	Processing mode when data conversion does not meet expectations. PERMISSIVE: A more relaxed mode, which is the default mode. If extra columns exist after the data conversion of a certain row, only the required columns are adopted automatically. DROPMALFORMED: Discard data that does not meet expectations. For example, if a row has an extra column, the row will be discarded. FAILFAST: Strictly follow the CSV format. The conversion fails if the result does not meet expectations, such as extra columns in a row.
encoding or charset	UTF-8	String encoding scheme Example: UTF-8, US-ASCII, ISO-8859-1, UTF-16BE, UTF-16LE, and UTF-16
quote	\\"	Indicate whether the quotation marks are single or double. Remember to use an escape character.
escape	\\\\	Escape character

charToEscapeQuoteEscaping	-	Escape characters required inside quotation marks
comment	\\u0000	Remarks.
header	false	Indicate whether a header exists.
inferSchema	false	Indicate whether to infer the data type of each column. If the inference is not performed, the data type of each column is string.
ignoreLeadingWhiteSpace	Read: false Write: true	Ignore the leading empty string.
ignoreTrailingWhiteSpace	Read: false Write: true	Ignore the trailing empty string.
columnNameOfCorruptRecord	_corrupt_record	Name of the column where data cannot be converted. If the value is inconsistent with that of spark.sql.columnNameOfCorruptRecord, the value specified in the CREATE TABLE statement prevails.
nullValue	-	Storage format for null values, which are empty strings by default. In this case, use the format defined by emptyValue.
nanValue	NaN	Storage format for non-numeric values
positiveInf	Inf	Storage format for positive infinity values
negativeInf	-Inf	Storage format for negative infinity values
compression or codec	-	Name of the compression algorithm. No compression is performed by default. Abbreviations such as bzip2, deflate, gzip, lz4, and snappy can be used.
timeZone	Default system time zone	Default time zone. If the value is inconsistent with that of spark.sql.session.timeZone (example: Asia/Shanghai), the value specified in the CREATE TABLE statement prevails.
locale	en-US	Language
dateFormat	yyyy-MM-dd	Default date format
timestampFormat	yyyy-MM-	Default time format, which is yyyy-MM-

	dd'T'HH:mm:ss.SSSXXX	dd'T'HH:mm:ss[.SSS][XXX] in non-LEGACY mode.
multiLine	false	Allow multiple rows.
maxColumns	20480	Maximum number of columns
maxCharsPerColumn	-1	Maximum number of characters per column. -1 indicates that the maximum number is not limited.
escapeQuotes	true	Escape quotation marks
quoteAll	quoteAll	Enclose all values in quotation marks.
samplingRatio	1.0	Sampling ratio
enforceSchema	true	Forcibly apply the specified schema for reading, and ignore the table headers.
emptyValue	Read: Write: \\"\\"	Read and write formats for empty values.
lineSep	-	Line break.
inputBufferSize	-	Buffer size for read. If the value is inconsistent with that of spark.sql.csv.parser.inputBufferSize, the value specified in the CREATE TABLE statement prevails.
unescapedQuoteHandling	STOP_AT_DELIMITER	Policy for handling unescaped quotation marks STOP_AT_DELIMITER: Stop reading when a delimiter is found. BACK_TO_DELIMITER: Back to the delimiter. STOP_AT_CLOSING_QUOTE: Stop reading when a closing quotation mark is found. SKIP_VALUE: Skip this column of data. RAISE_ERROR: Report an error.

Configurations supported by ORC data tables are as follows:

Key Supported by OPTIONS	Corresponding Default Value	Description
compression or orc.compress	snappy	Compression algorithm. Abbreviations such as snappy,

		zlib, lzo, lz3, and zstd are supported. If the value is inconsistent with that of spark.sql.orc.compression.codec, the value specified in the CREATE TABLE statement prevails.
mergeSchema	false	Merge schemas. If the value is inconsistent with that of spark.sql.orc.mergeSchema, the value specified in the CREATE TABLE statement prevails.

If you use HiveRead and HiveWriter (configuring spark.sql.hive.convertMetastoreOrc=false) for read and write, OPTIONS can also support native ORC configurations. For details, see LanguageManual ORC.

Most of the parameters related to PARQUET data tables can be configured using SparkConf (recommended). Configurations supported by OPTIONS are as follows:

Key Supported by OPTIONS	Corresponding Default Value	Description
compression or parquet.compression	snappy	Compression algorithm. Snappy is used by default. If the value is inconsistent with that of spark.sql.parquet.compression.codec, the value specified in the CREATE TABLE statement prevails.
mergeSchema	false	Indicate whether to merge schemas. If the value is inconsistent with that of spark.sql.parquet.mergeSchema, the value specified in the CREATE TABLE statement prevails.
datetimeRebaseMode	EXCEPTION	Policy of date conversion during PARQUET file write. Dates are converted based on the Gregorian calendar in LGACY mode but not converted in CORRECTED mode. In EXCEPTION mode, an error is reported when dates are in different formats. If the value is inconsistent with that of spark.sql.parquet.datetimeRebaseModeInRead, the value specified in the CREATE TABLE statement prevails.
int96RebaseMode	EXCEPTION	Policy of time conversion during PARQUET file

		read. Time values are converted based on the Gregorian calendar in LGACY mode but not converted in CORRECTED mode. In EXCEPTION mode, an error is reported when time values are in different formats. If the value is inconsistent with that of spark.sql.parquet.int96RebaseModeInRead, the value specified in the CREATE TABLE statement prevails.

If you use HiveRead and HiveWriter (setting spark.sql.hive.convertMetastoreParquet to false) for read and write, OPTIONS can also support native PARQUET configurations. Reference: Hadoop integration.

Example

```
CREATE TABLE dempts(
    id bigint COMMENT 'id number',
    num int,
    eno float,
    dno double,
    cno decimal(9,3),
    flag  boolean,
    data string,
    ts_year timestamp,
    date_month date,
    bno binary,
```

```
    point struct<x: double, y: double>,
    points array<struct<x: double, y: double>>,
    pointmaps map<struct<x: int>, struct<a: int>>
    )
USING iceberg
COMMENT 'table documentation'
PARTITIONED BY (bucket(16,id), years(ts_year), months(date_month), identity(bno),
LOCATION '/warehouse/db_001/dempts'
TBLPROPERTIES ('write.format.default'='orc');
```

Notes

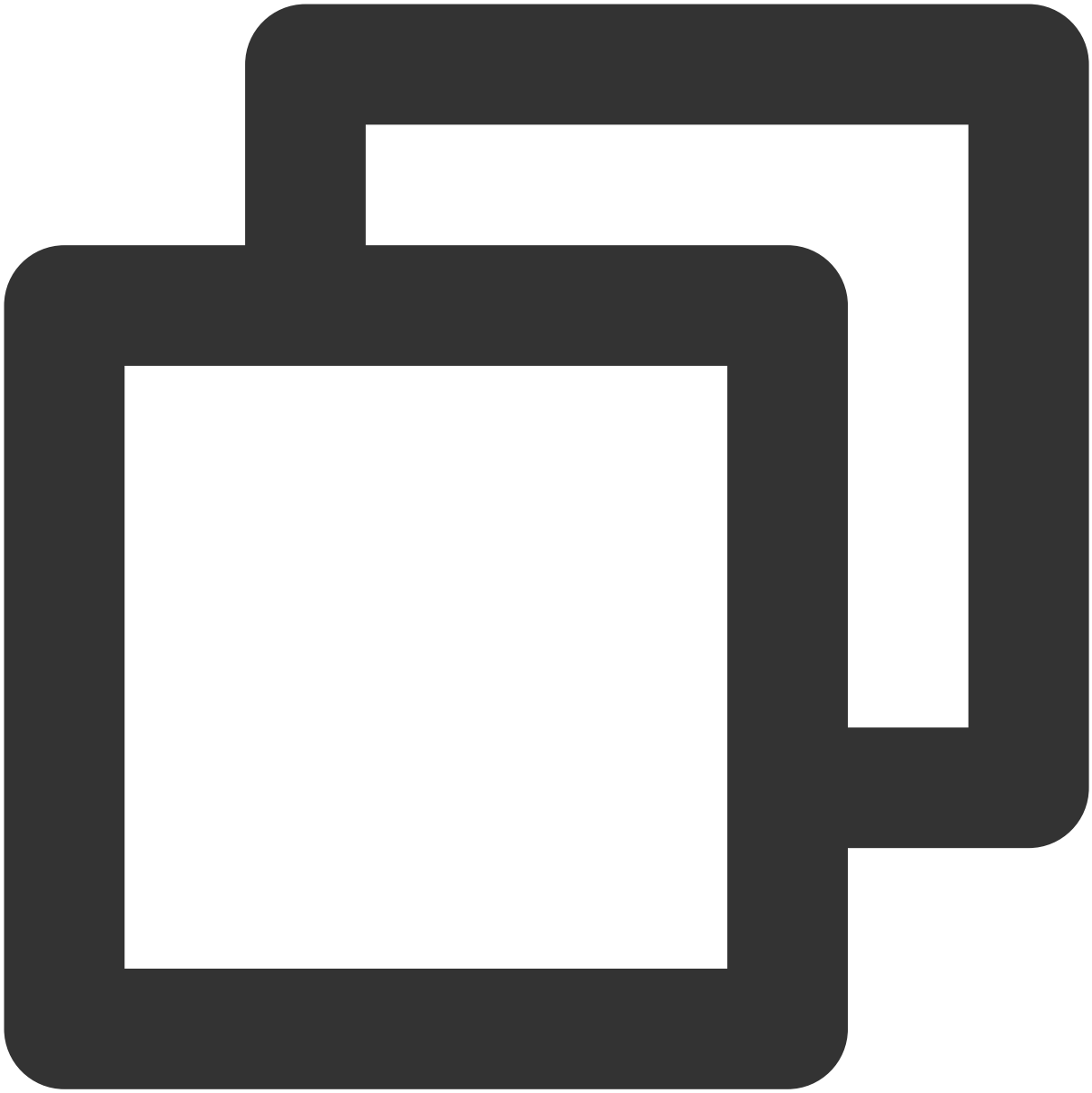
Spark's USING and Hive's STORED AS are different when they are used as keywords of CREATE_TABLE. The file format and read method after table creation may not meet expectations. Note:
USING DATA_SOURCE: Used in Spark statement. This keyword indicates the data source used as the input format for table creation and directly affects the file format and reading method of the table under the location. The value can be CSV, TXT, Iceberg, Parquet, Orc, etc.
STORED AS FILE_FORMAT: This keyword is used in Hive statements to specify the file format for table storage. The value can be TXT, Parquet, Orc, etc. This keyword is not recommended because the native reader/writer of Spark may not support the specified file format such as CSV.

Statement for Native Iceberg Table

Caution

This statement is supported for creating native tables only.

Statement



```
CREATE TABLE [ IF NOT EXISTS ] table_identifier
    ( col_name[:] col_type [ COMMENT col_comment ], ... )
[ COMMENT table_comment ]
[ PARTITIONED BY ( col_name1, transform(col_name2), ... ) ]
```

Parameter

`table_identifier` : Table name in the three-part format. Example: catalog.db.name.

Schemas and Data Types

```
col_type
  : primitive_type
  | nested_type

primitive_type
  : boolean
  | int/integer
  | long/bigint
  | float
  | double
  | decimal(p,s), p=maximum number of digits, s=maximum number of decimal places, s
```

```
    | date
    | timestamp, timestamp with timezone, time and without timezone are not supported
    | string, which corresponds to the UUID type of Iceberg tables.
    | binary, which corresponds to the FIXED type of Iceberg tables.

  nested_type
    : struct
    | list
    | map
```

Partition Transforms

```
transform
  : identity, support any type. DLC does not support this transformation.
  | bucket[N], partition by hashed value mod N buckets, supporting col_type: int,lo
  | truncate[L], partition by value truncated to L, supporting col_type: int,long,d
  | years, partition by year, supporting col_type: date,timestamp
  | months, partition by month, supporting col_type: date,timestamp
  | days/date, partition by date, supporting col_type: date,timestamp
  | hours/date_hour, partition by hour, supporting col_type: timestamp
```

Example

```
CREATE TABLE dempts(
    id bigint COMMENT 'id number',
    num int,
    eno float,
    dno double,
    cno decimal(9,3),
    flag  boolean,
    data string,
    ts_year timestamp,
    date_month date,
    bno binary,
```

```
    point struct<x: double, y: double>,
    points array<struct<x: double, y: double>>,
    pointmaps map<struct<x: int>, struct<a: int>>
    )
COMMENT 'table documentation'
PARTITIONED BY (bucket(16,id), years(ts_year), months(date_month), identity(bno),
```

REPLACE TABLE AS SELECT

Last updated：2024-08-07 17:12:31

Description

Supported engine: SparkSQL

Applicable table: External and native Iceberg tables

Purpose: Replace the table and update the snapshot while retaining the table history.

Statement

```
CREATE [OR REPLACE] TABLE table_identifier
USING iceberg
    [ COMMENT table_comment ]
    [ PARTITIONED BY ( col_name1, transform(col_name2), ... ) ]
    [ LOCATION path ]
    [ TBLPROPERTIES ( property_name=property_value, ... ) ]
AS select_statement
```

Example

```
CREATE OR REPLACE TABLE dempts_replace
USING iceberg
COMMENT 'table create as replace'
PARTITIONED BY (eno, dno)
TBLPROPERTIES ('write.format.default'='avro')
LOCATION '/warehouse/db_001/dempts_replace'
AS SELECT * from dempts;
```

SHOW TABLES

Last updated：2024-08-07 17:12:50

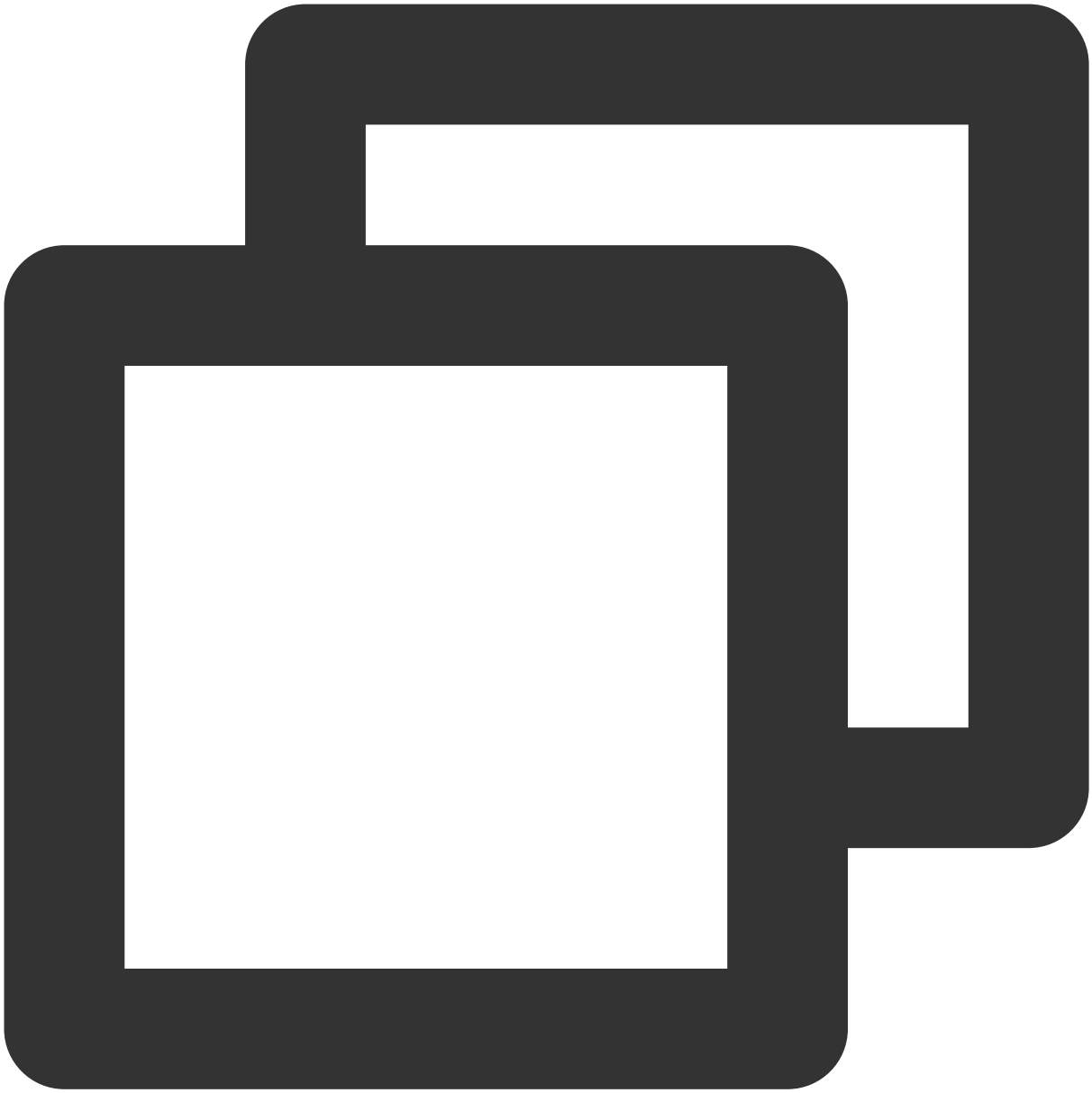
Description

Supported engines: Presto and SparkSQL

Applicable table: Native and external tables

Purpose: List all base tables and views in the database.

Statement



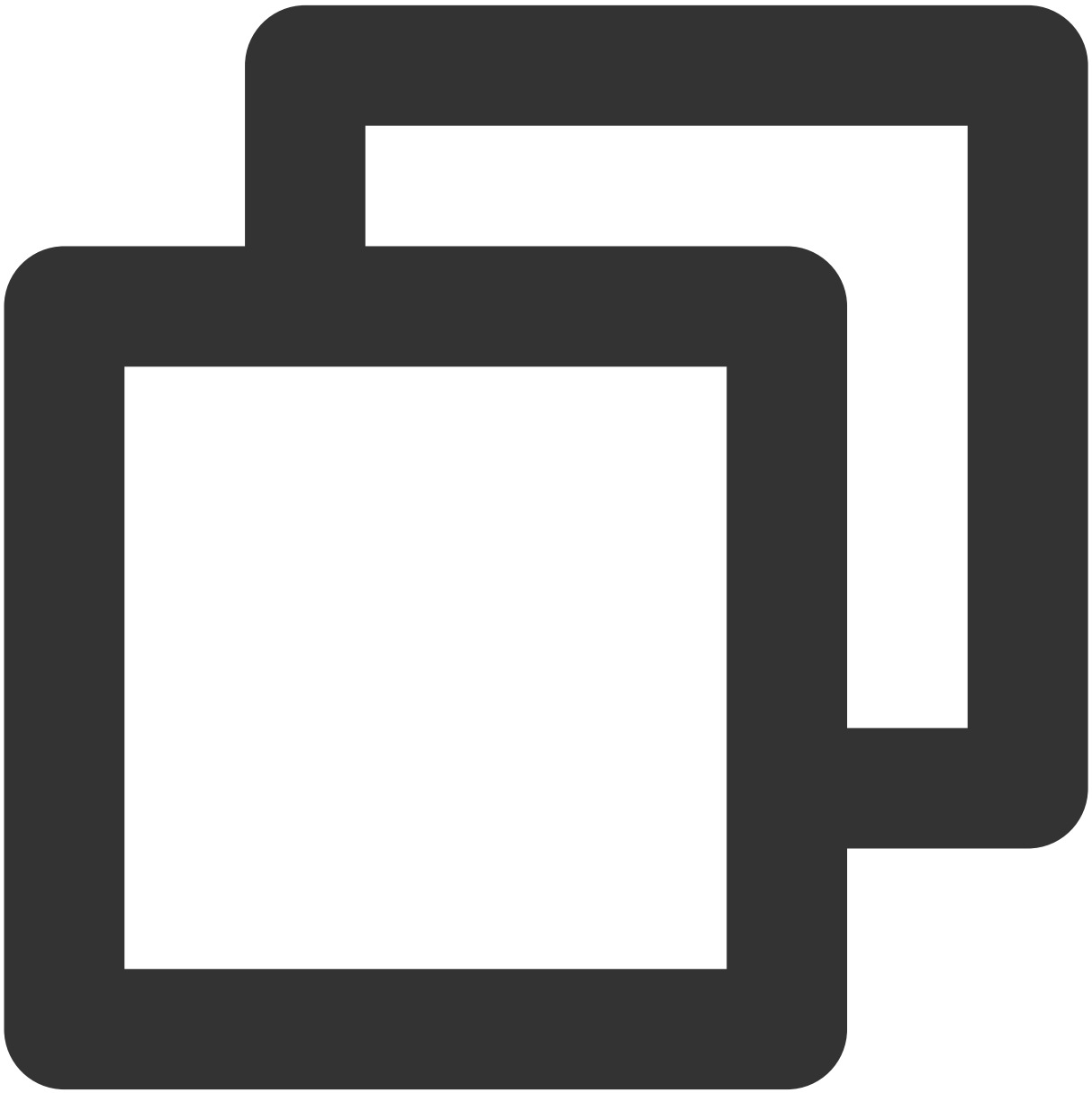
```
SHOW TABLES [IN database_name] ['regular_expression']
```

Parameter

`[IN database_name]` : Name of the database whose tables are to be listed. If this parameter is left blank, the database in the current context is selected.

`['regular_expression']` : Regular expression for filtering matching tables Only the wildcard character * representing any character or expressions representing spaces or special characters between normal characters can be used.

Example



```
SHOW TABLES IN sampledb;
```

```
SHOW TABLES IN sampledb '*flights*';
```

SHOW CREATE TABLE

Last updated：2024-08-07 17:13:07

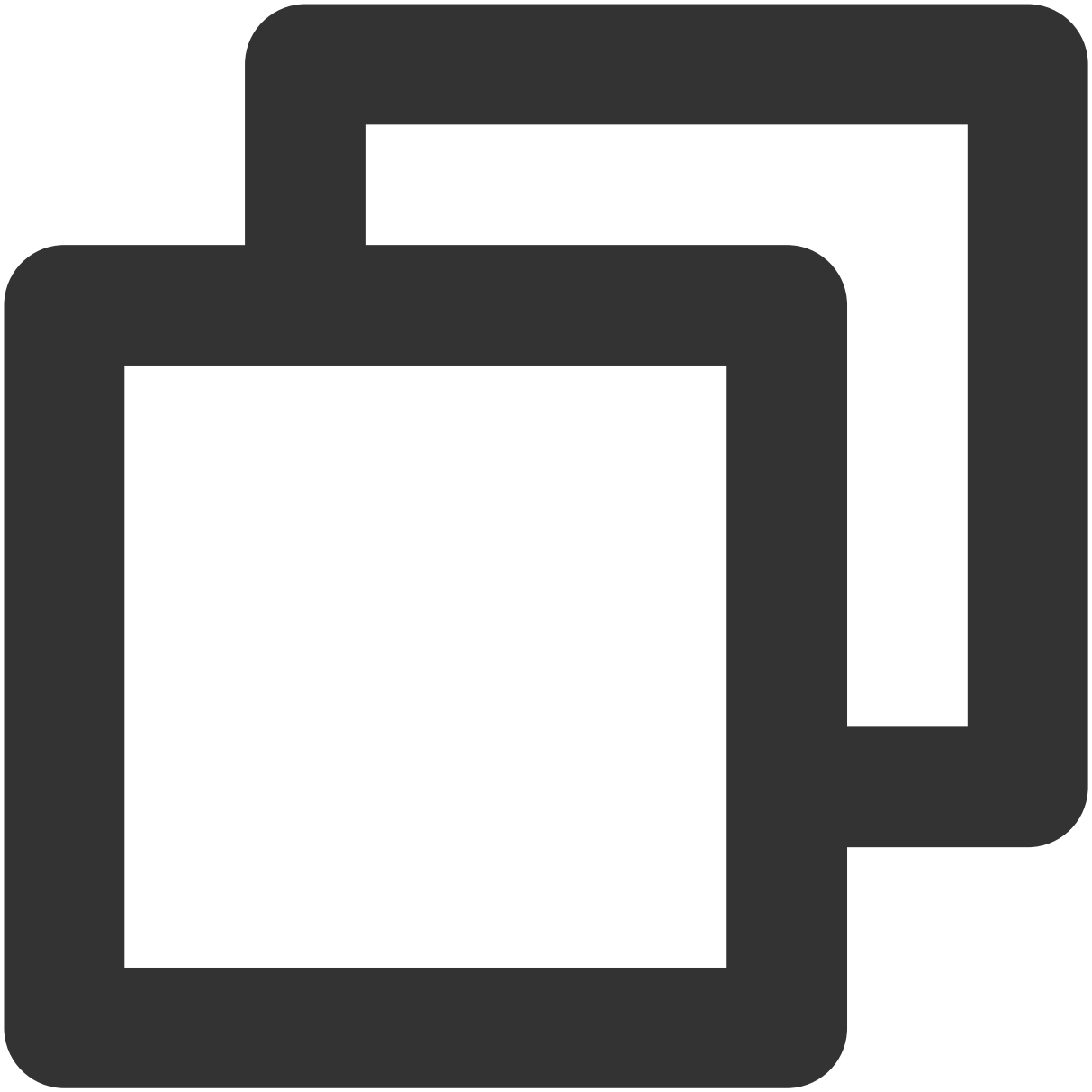
Description

Supported engines: Presto and SparkSQL

Applicable table: Native and external tables

Purpose: View table creation statements.

Statement



```
SHOW CREATE TABLE [catalog_name.][db_name.]table_name
```

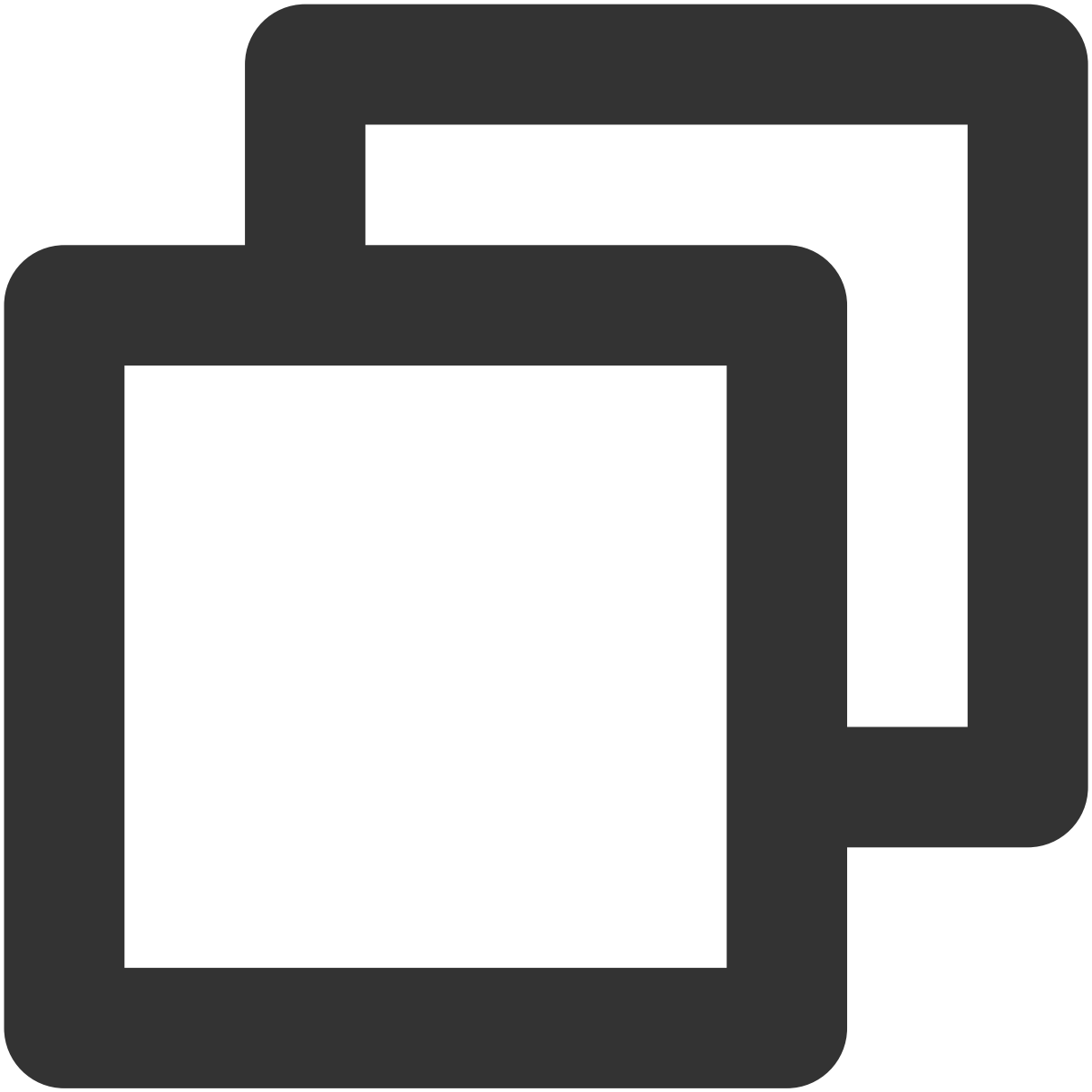
Parameter

`TABLE [db_name.]table_name`

`db_name` : Database name

`table_name` : Table name

Example



```
SHOW CREATE TABLE tbl;
```

SHOW TBLPROPERTIES

Last updated：2024-08-07 17:13:21

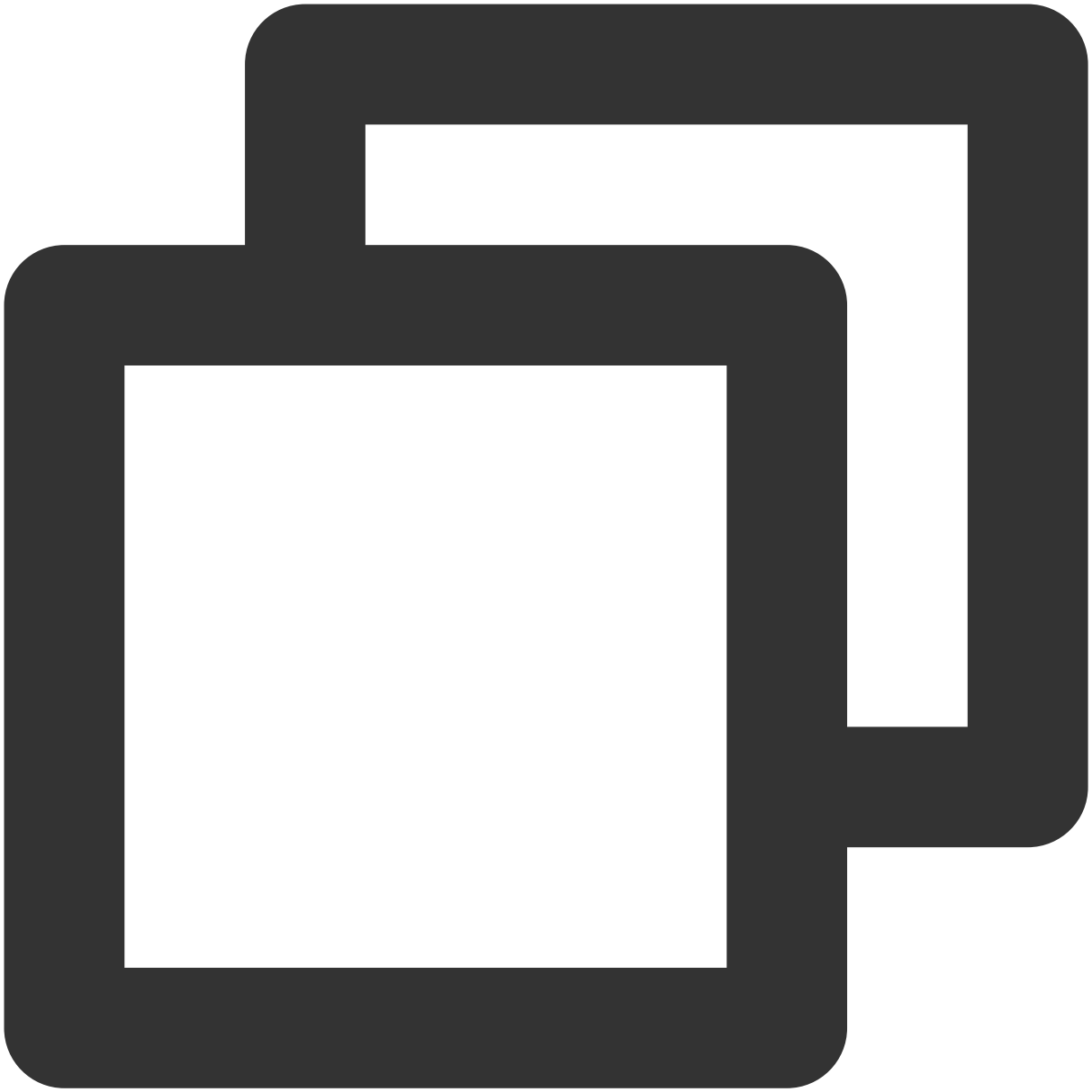
Description

Supported engines: Presto and SparkSQL

Applicable table: Native and external tables

Purpose: List table attributes.

Statement

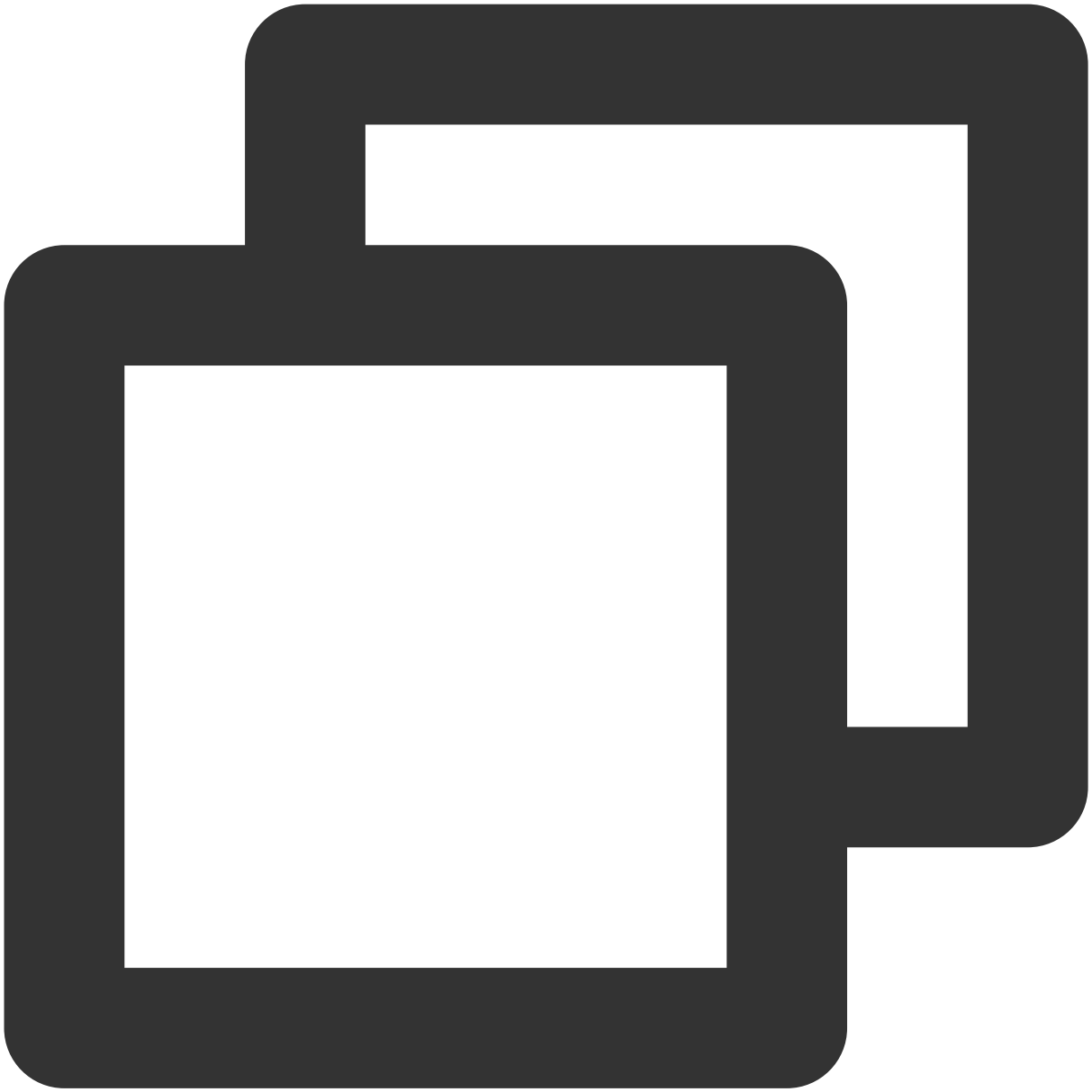


```
SHOW TBLPROPERTIES table_name [('property_name')]
```

Parameter

`[('property_name')]` : If the attribute exists, list only the attribute name and value.

Example



```
SHOW TBLPROPERTIES tb1;
```

```
SHOW TBLPROPERTIES orders('tb1');
```

DESCRIBE TABLE

Last updated：2024-08-07 17:13:35

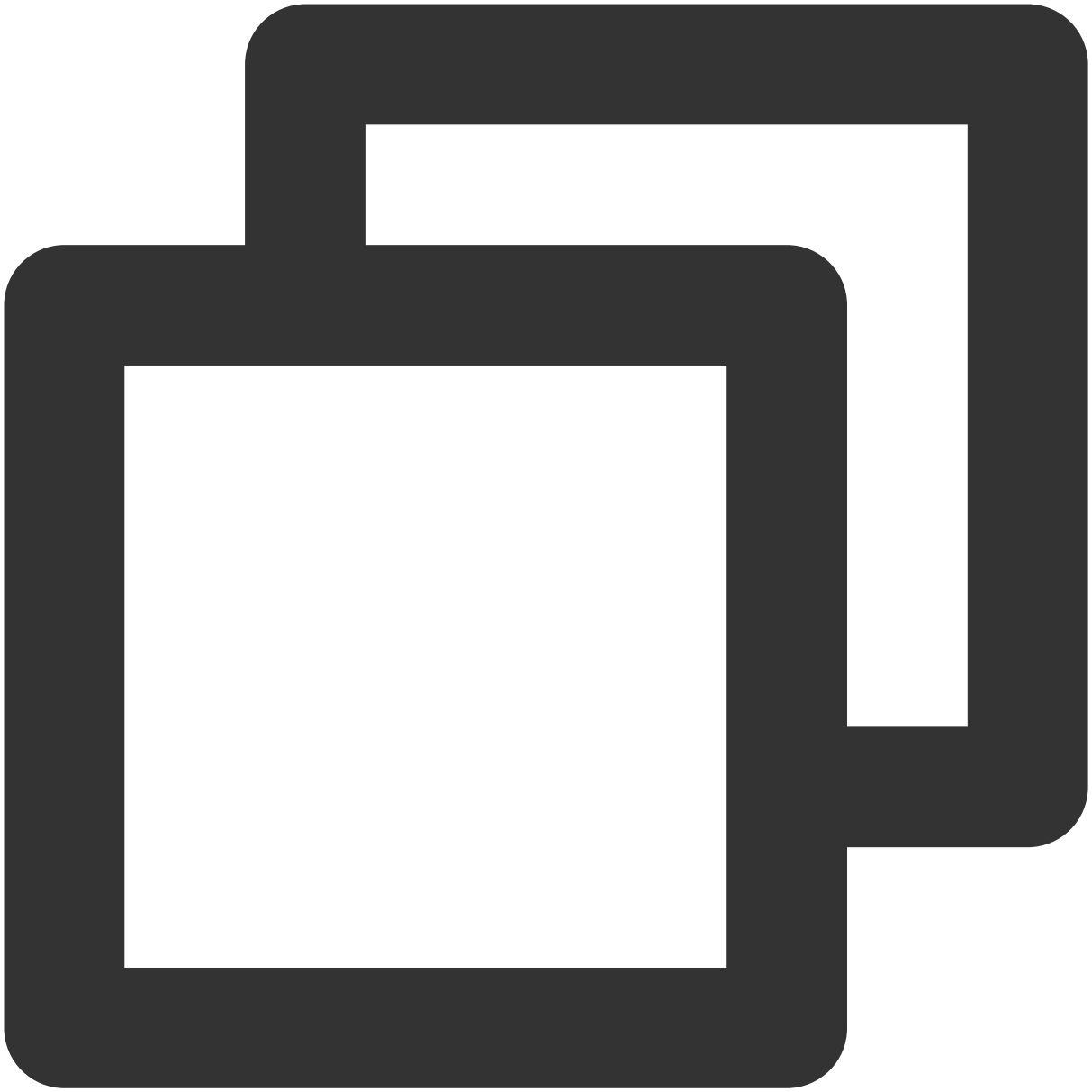
Description

Supported engines: Presto and SparkSQL

Applicable table: Native and external tables

Purpose: View column information and metadata information on the data table.

Standard Statement



```
DESCRIBE [EXTENDED | FORMATTED] [db_name.]table_name [PARTITION partition_spec];
```

Parameter

`[EXTENDED | FORMATTED]` : Specify the format of the table.

`table_name` : Table name

`[PARTITION partition_spec]` : List of partitions in the table

Example

```
DESCRIBE tbl;
DESCRIBE FORMATTED tbl PARTITION (date_id = '2019-01-07');
```

SHOW COLUMNS IN TABLE

Last updated：2024-08-07 17:13:48

Description

Supported engines: Presto and SparkSQL

Applicable table: Native and external tables

Purpose: View column information on data table.

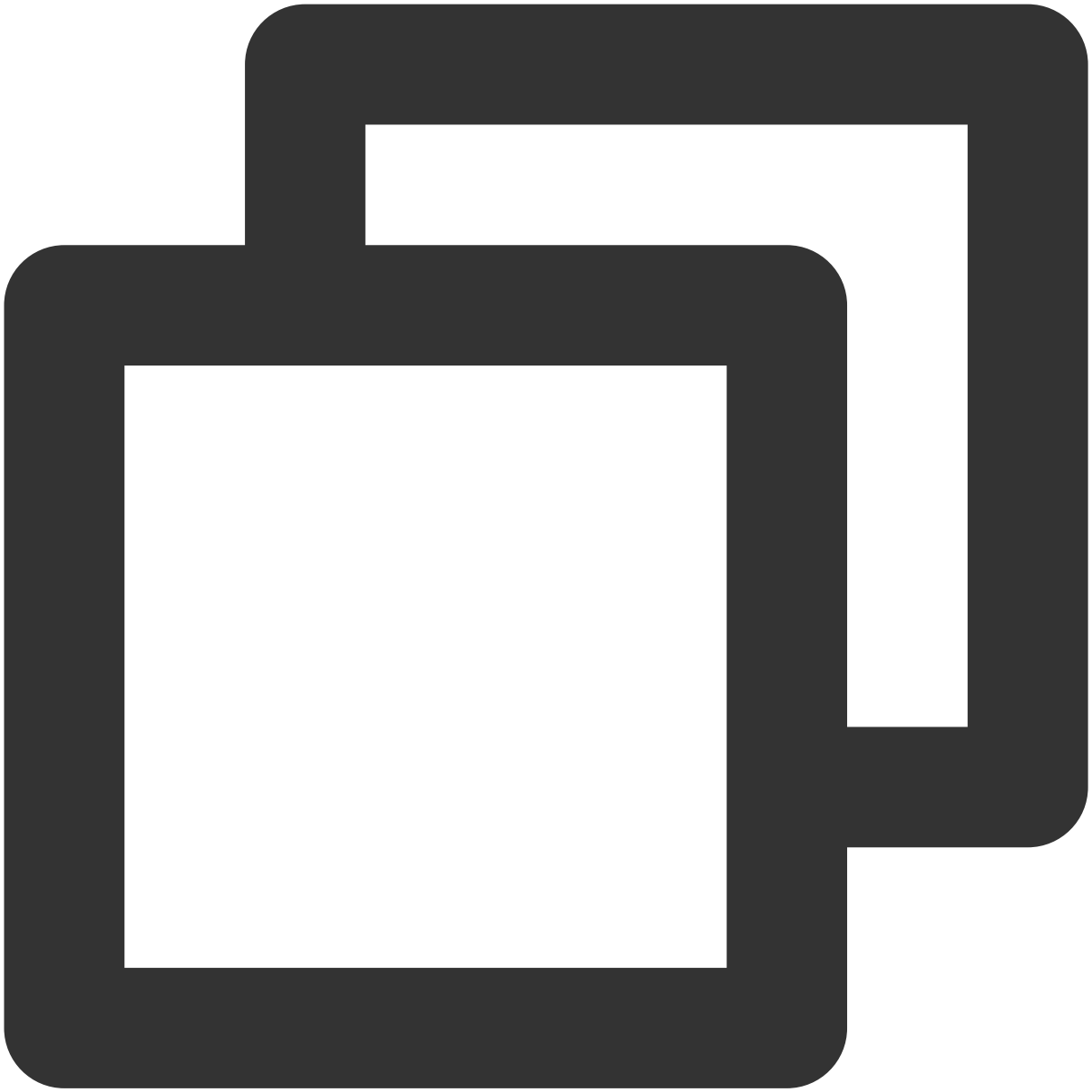
Statement

```
SHOW COLUMNS IN table_name
```

Parameter

`table_name` : Data table name

Example



```
SHOW COLUMNS IN clicks;
```

ALTER TABLE
ALTER TABLE ADD COLUMNS

Last updated：2024-08-07 17:14:57

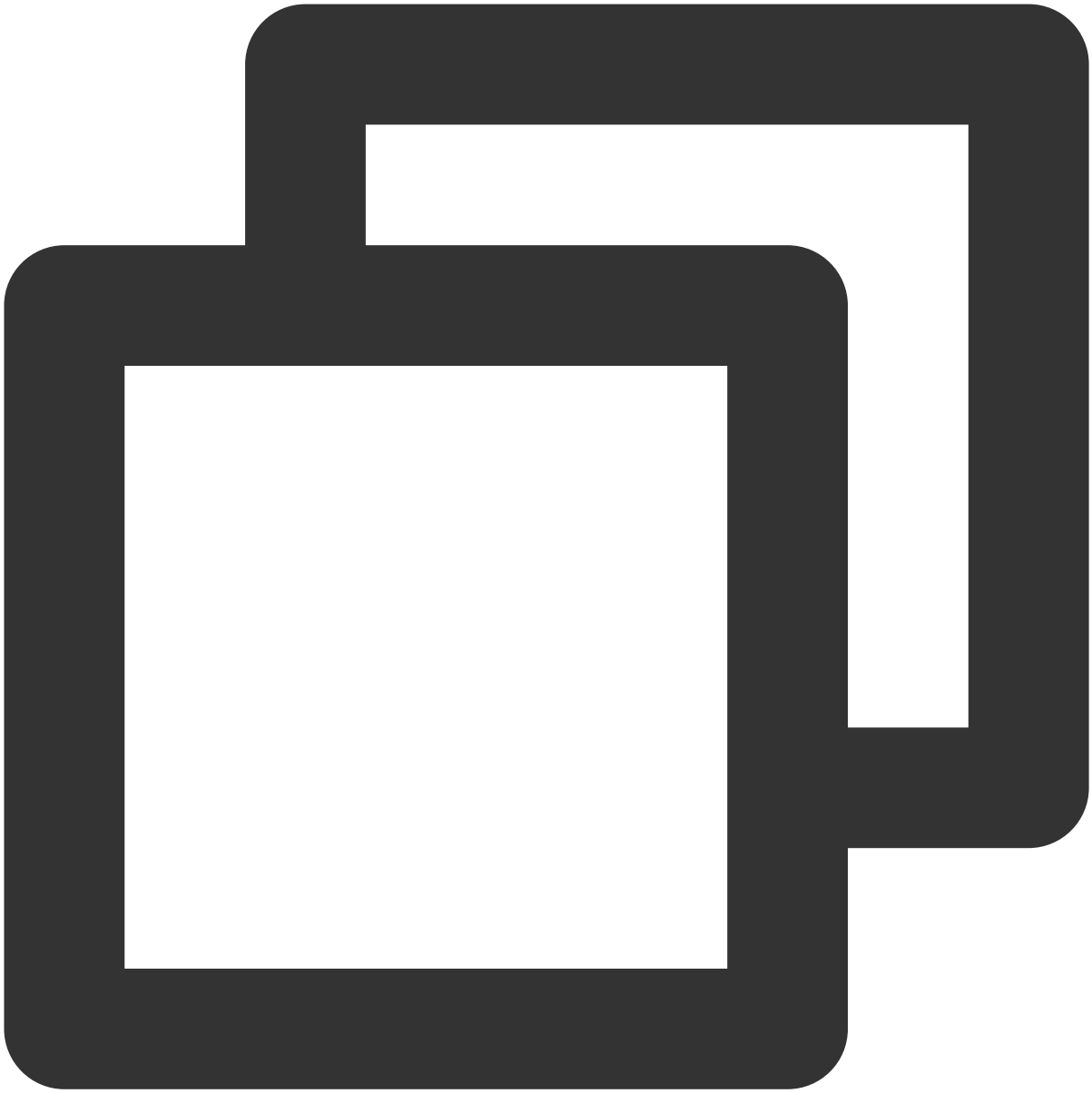
Description

Supported engines: Presto and SparkSQL

Applicable table: Native Iceberg tables and external tables

Purpose: Change data table attributes.

Standard Statement



```
ALTER TABLE table_name
 [PARTITION
   (partition_col1_name = partition_col1_value
   [,partition_col2_name = partition_col2_value][,...])]
  ADD COLUMNS (col_name data_type) [RESTRICT | CASCADE]
```

Parameter

`table_name` : Name of the target table

`partition_col1_name` : Partitioning column name

`partition_col1_value` : Partitioning column value

`col_name` : Name of the column to be added.

`data_type` : Type of the column to be added.

Example

```
ALTER TABLE events ADD COLUMNS (eventowner string);

ALTER TABLE events ADD COLUMNS (eventowner string) CASCADE;

//The ALTER TABLE PARTITION ADD COLUMNS statement supports only native DLC tables.
ALTER TABLE events PARTITION (year='2021') ADD COLUMNS (event string);
```

Caution

If you use the `ROW FORMAT SERDE 'org.apache.hive.hcatalog.data.JsonSerDe'` format during table creation, you cannot add columns after the table is created. If you use JsonSerDe to create a table, confirm the table structure if possible. If you must add columns, you can delete the existing table and create a new one.

ALTER TABLE ADD COLUMN AFTER/FIRST

Last updated：2024-08-07 17:15:16

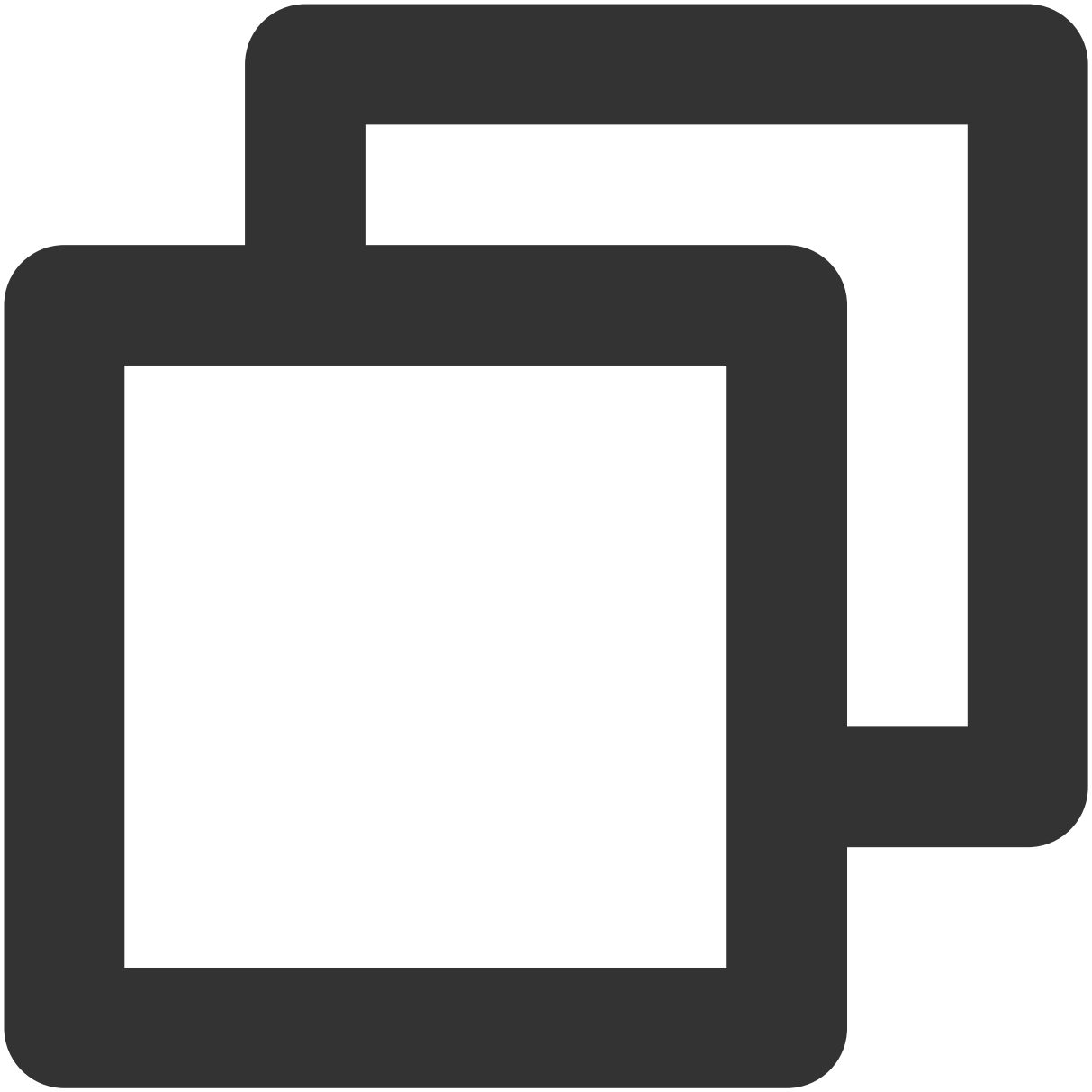
Description

Supported engines: Presto and SparkSQL

Applicable table: Native Iceberg tables and external tables

Purpose: Add column to data table.

Standard Statement



```
ALTER TABLE table_name ADD COLUMN column_name1 column_type [COMMENT col_comment] [F
```

Parameter

`table_name` : Name of the target table

`column_name1` : Column to be added

`column_type` : Type of the column to be added

`col_comment` : Comment for the column to be added

`column_name2` : Column followed by the added column

Example

```
ALTER TABLE `TBL` ADD COLUMN `COL2` STRING COMMENT 'test' AFTER `COL1`

ALTER TABLE `TBL` ADD COLUMN `COL2` STRING COMMENT 'test' FIRST
```

ALTER TABLE DROP COLUMN

Last updated：2024-08-07 17:15:45

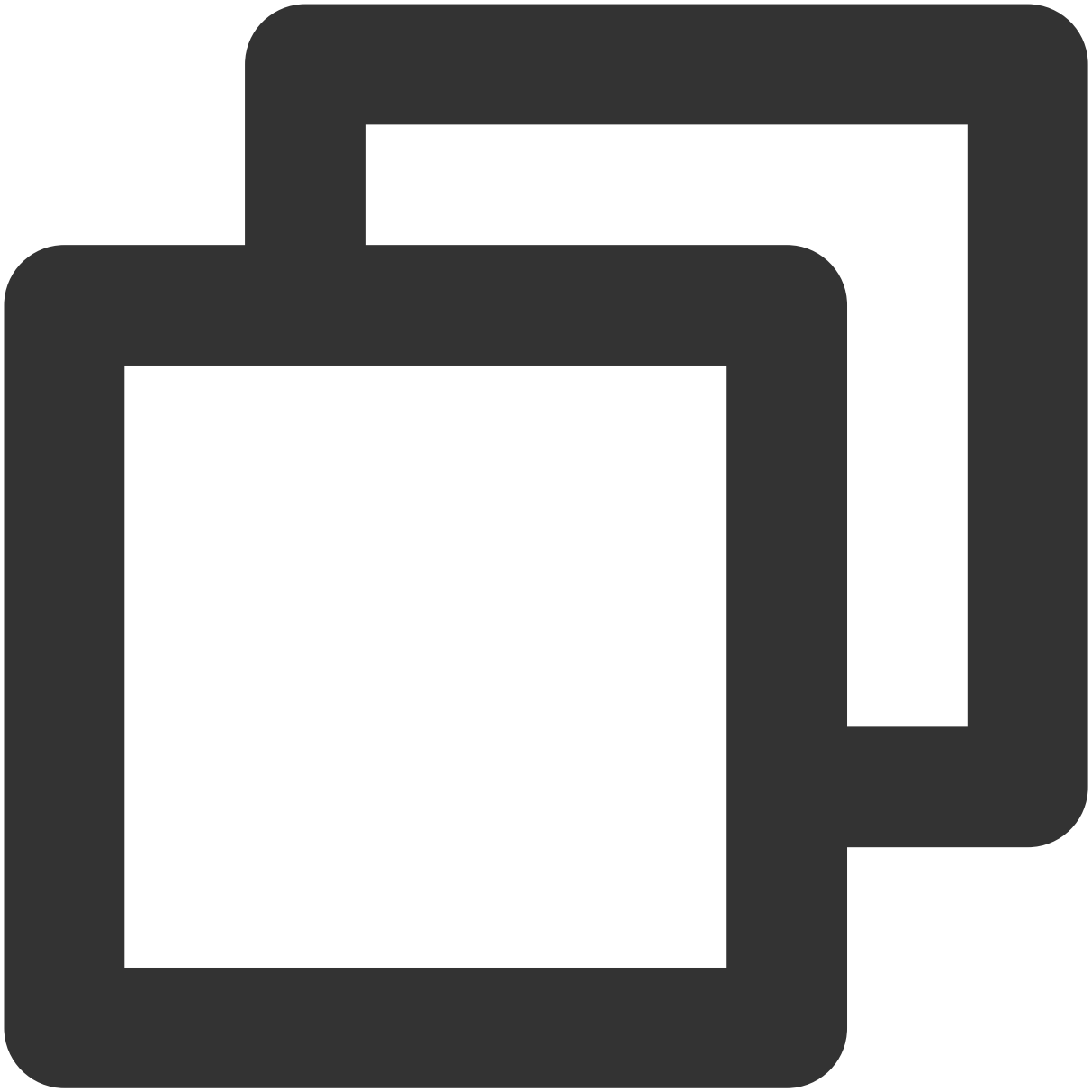
Description

Supported engines: Presto and SparkSQL

Applicable table: Native Iceberg tables and external tables

Purpose: Delete a field from a data table.

Standard Statement



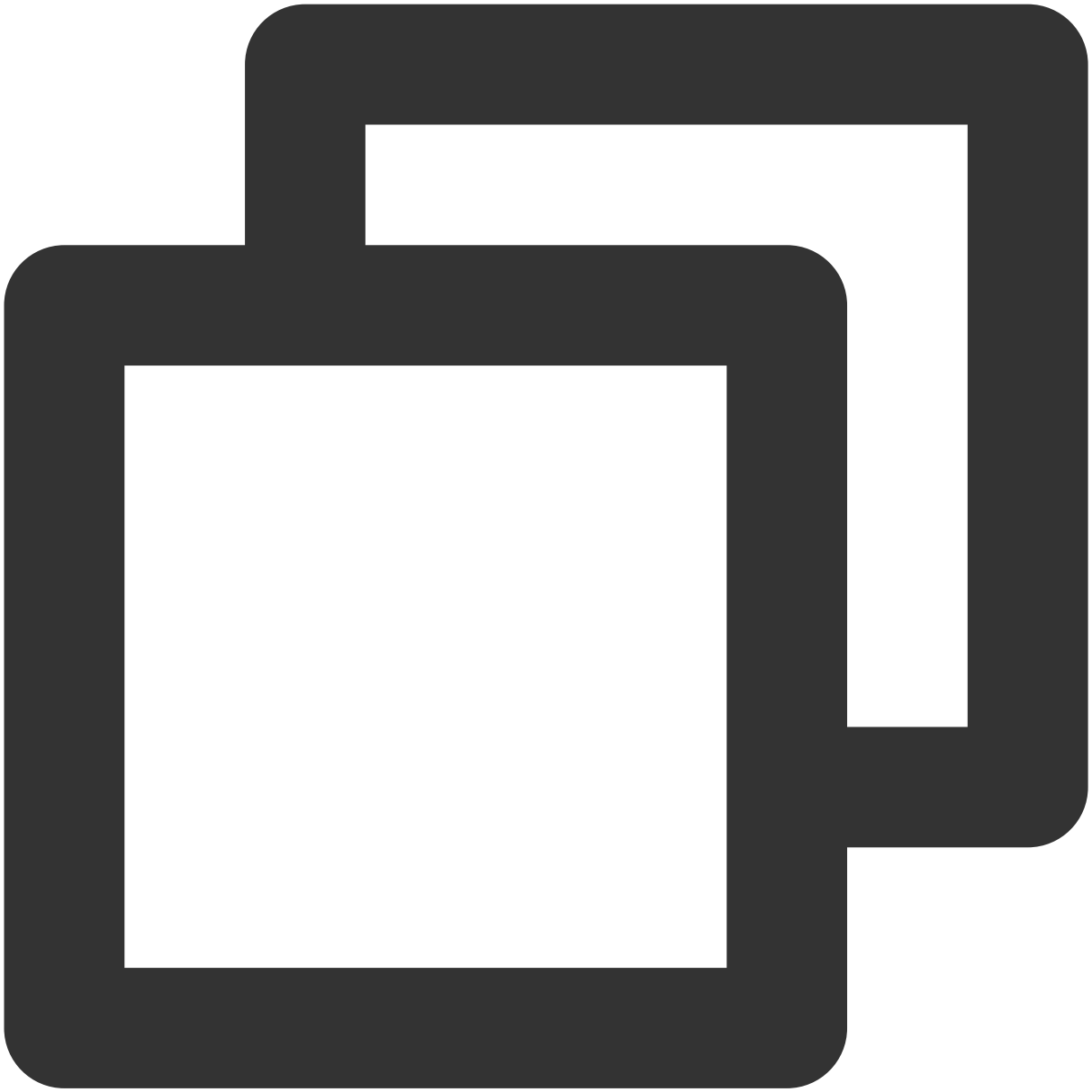
```
ALTER TABLE table_name DROP COLUMN column_name
```

Parameter

`table_name` : Name of the target table

`column_name` : Name of the column to be deleted

Example



```
ALTER TABLE `TBL` DROP COLUMN `COL2`
```

ALTER TABLE ADD PARTATION

Last updated：2024-08-07 17:15:56

Description

Supported engines: Presto and SparkSQL

Applicable table: External tables

Purpose: Create one or more partitioning columns for a data table.

Statement

```
ALTER TABLE table_name ADD [IF NOT EXISTS]
  PARTITION (partition_spec)
  [PARTITION (partition_spec) ...]

partition_spec:
  : partition_column = partition_col_value, partition_column = partition_col_value,
```

Parameter

`table_name` : Name of the target table

`partition_column` : Partitioning column name

`partition_col_value` : Partitioning column value

Example

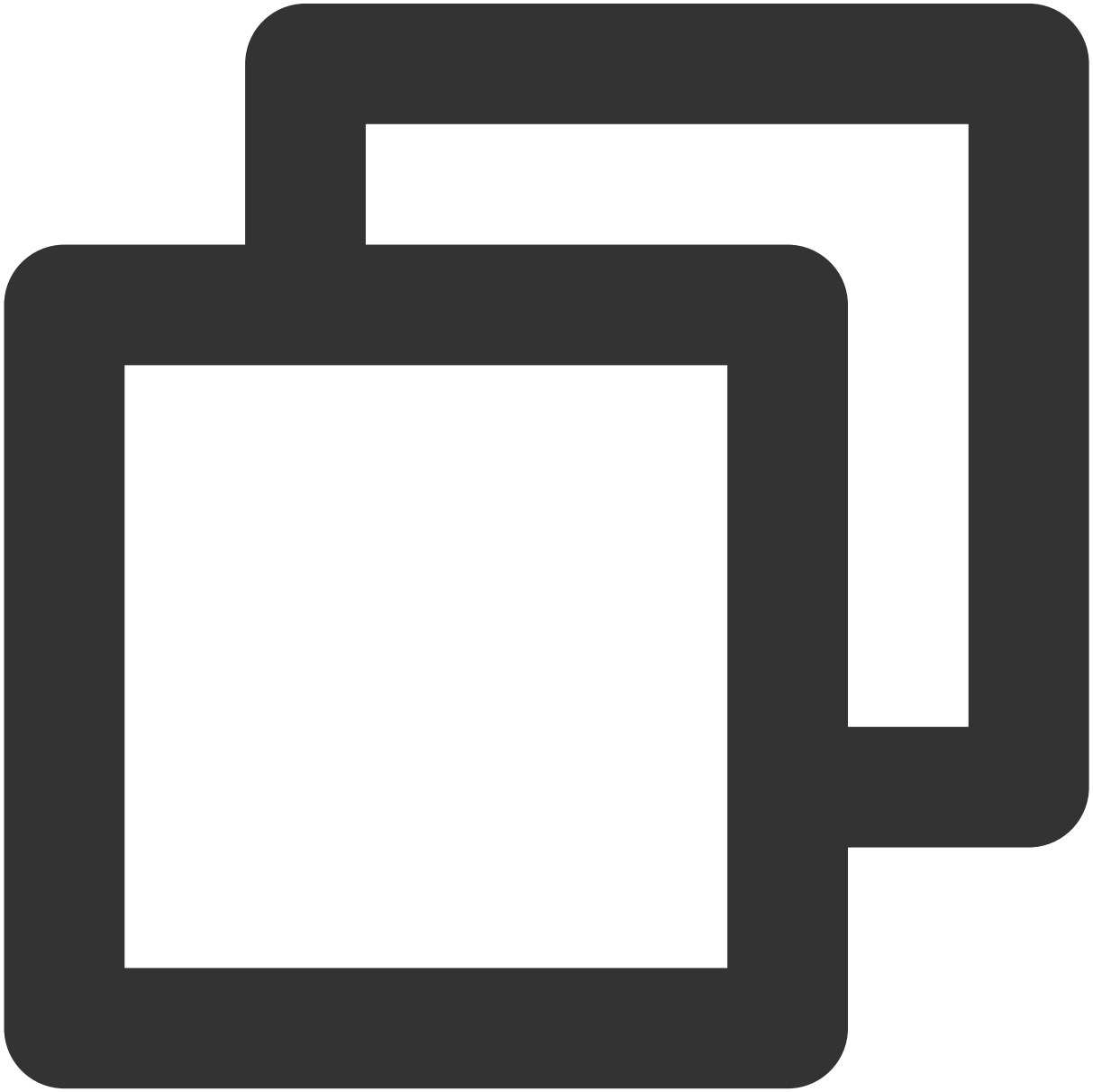
```
ALTER TABLE tbl ADD PARTITION (p1=1, p2='a');

ALTER TABLE tbl ADD IF NOT EXISTS PARTITION (P1 = 1) PARTITION (P2 = 2);
```

SHOW PARTITIONS

Last updated：2024-08-07 17:16:15

List all partitions in the table.

Statement



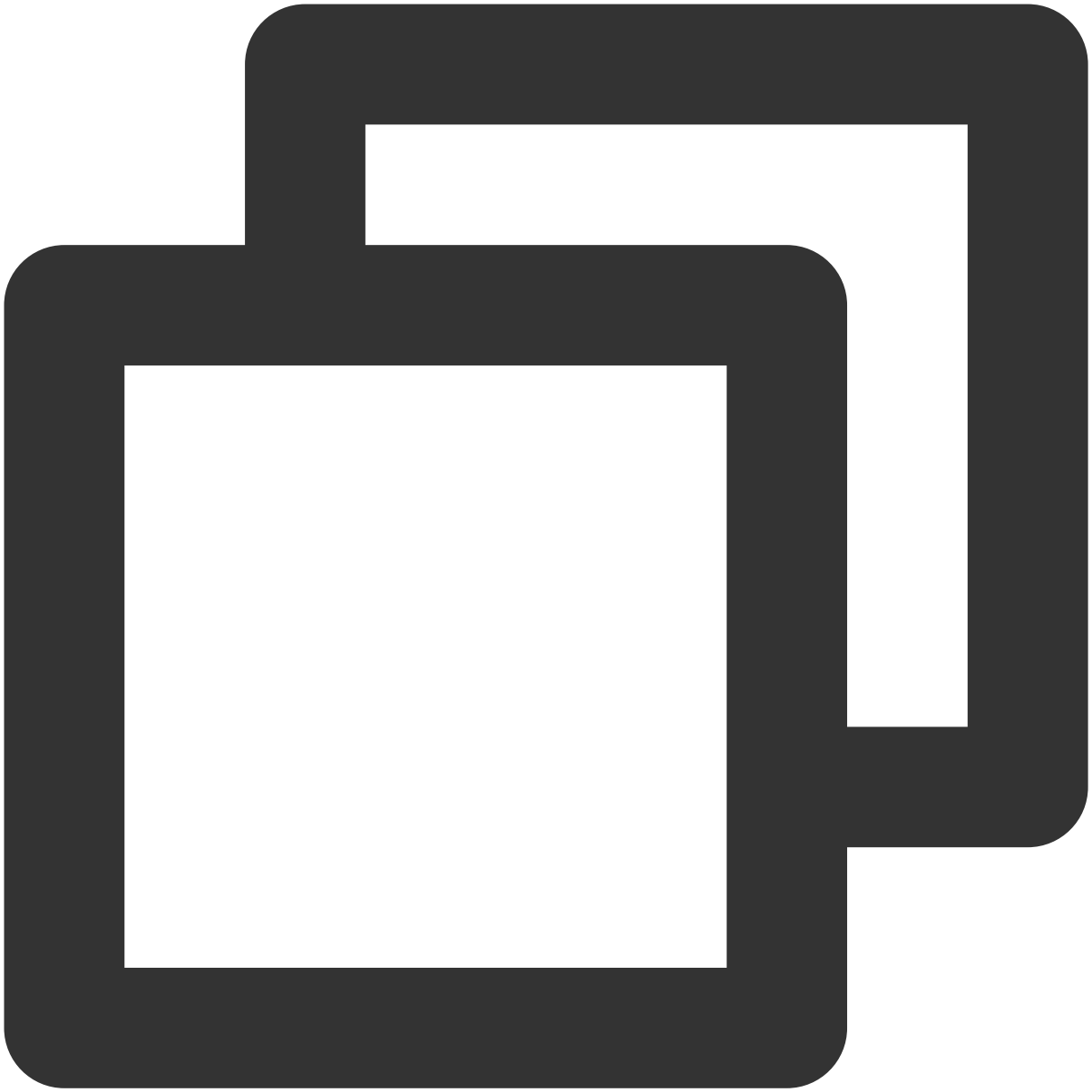
```
SHOW PARTITIONS [db_name.]table_name [PARTITION(partition_spec)];
```

Parameter

`TABLE [db_name.]table_name` : Table name.

`[PARTITION(partition_spec)]` : Partition columns. Multiple partition column conditions are allowed.

Example



```
SHOW PARTITIONS db01.table PARTITION(ds='2010-03-03', hr='12');
```

ALTER TABLE DROP PARTITION

Last updated：2024-08-07 17:16:26

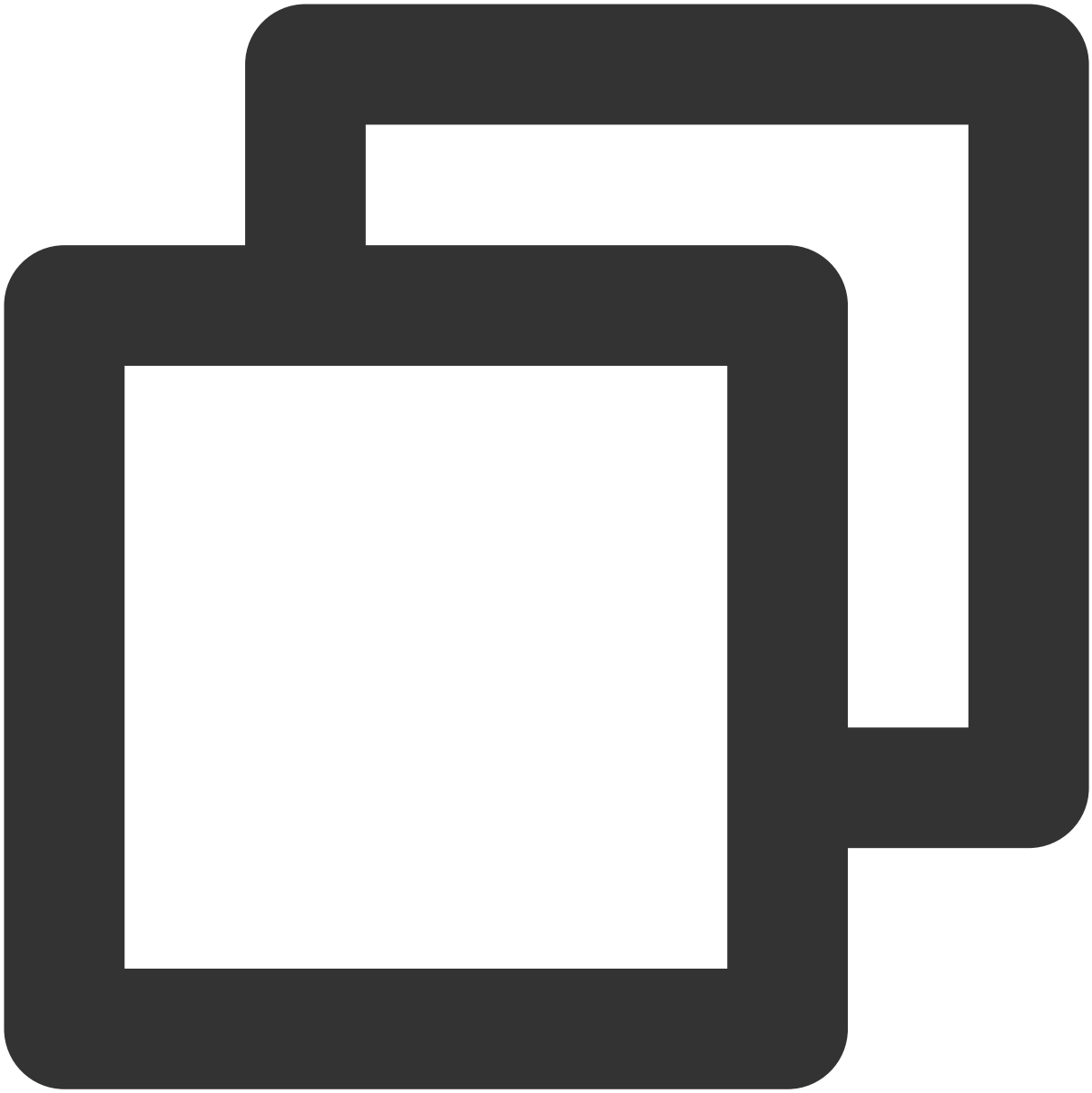
Description

Supported engines: Presto and SparkSQL

Applicable table: External tables

Purpose: Delete a partitioning column of a data table.

Standard Statement



```
ALTER TABLE table_name DROP [IF EXISTS] PARTITION (partition_spec) [,PARTITION (par

partition_spec:
  : partition_column = partition_col_value, partition_column = partition_col_value,
```

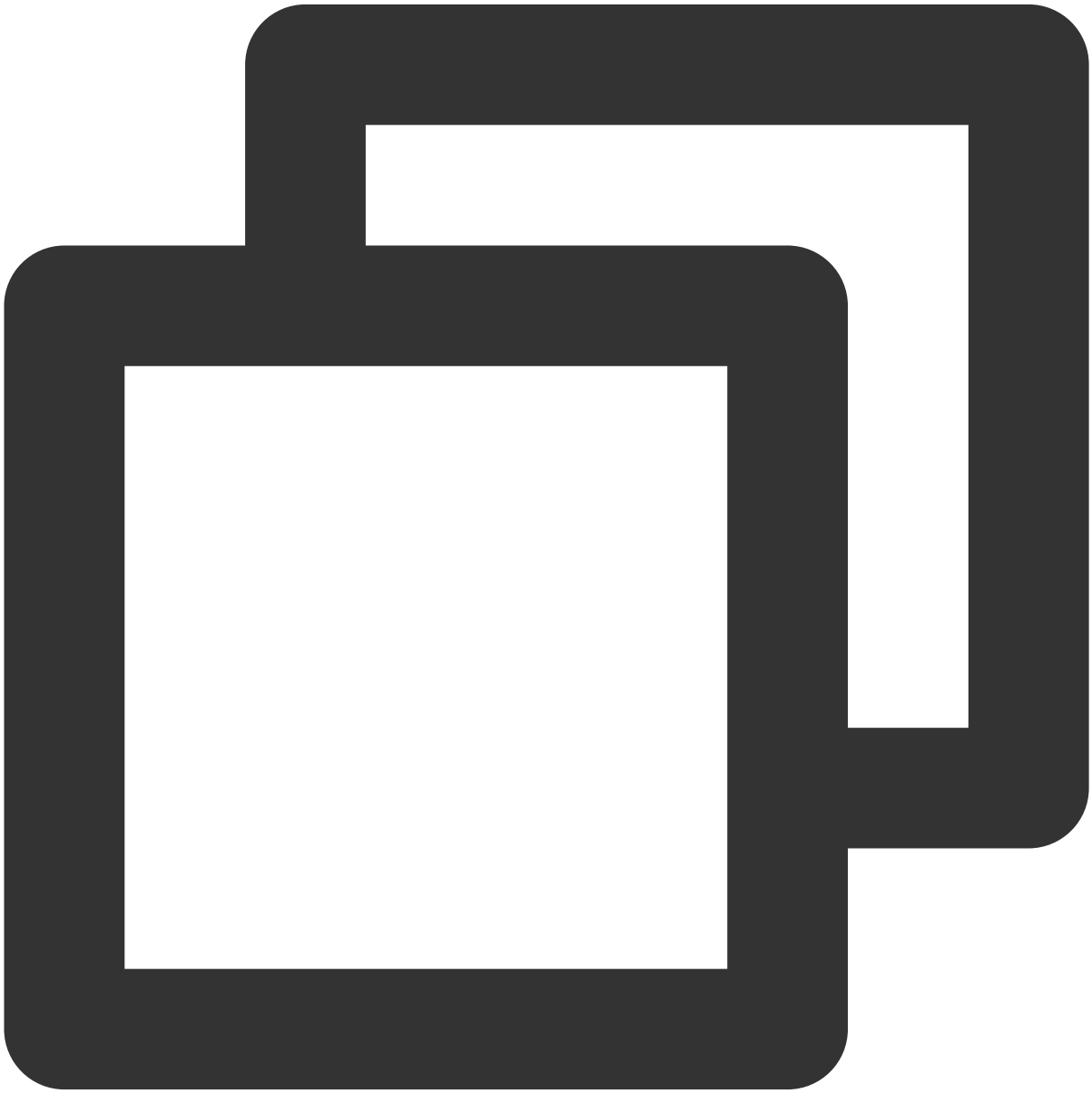
Parameter

`table_name` : Name of the target table

`partition_column` : Partitioning column name

`partition_col_value` : Partitioning column value

Example



```
ALTER TABLE page_view DROP PARTITION (dt='2008-08-08', country='us')
```

```
ALTER TABLE `page_view` DROP
PARTITION (`dt` = '2008-08-08', `country` = 'us'),
PARTITION (`dt` = '2008-08-08', `country` = 'us'),
PARTITION (`dt` = '2008-08-08', `country` = 'us')
```

ALTER TABLE ADD PARTITION FIELD

Last updated：2024-08-07 17:16:40

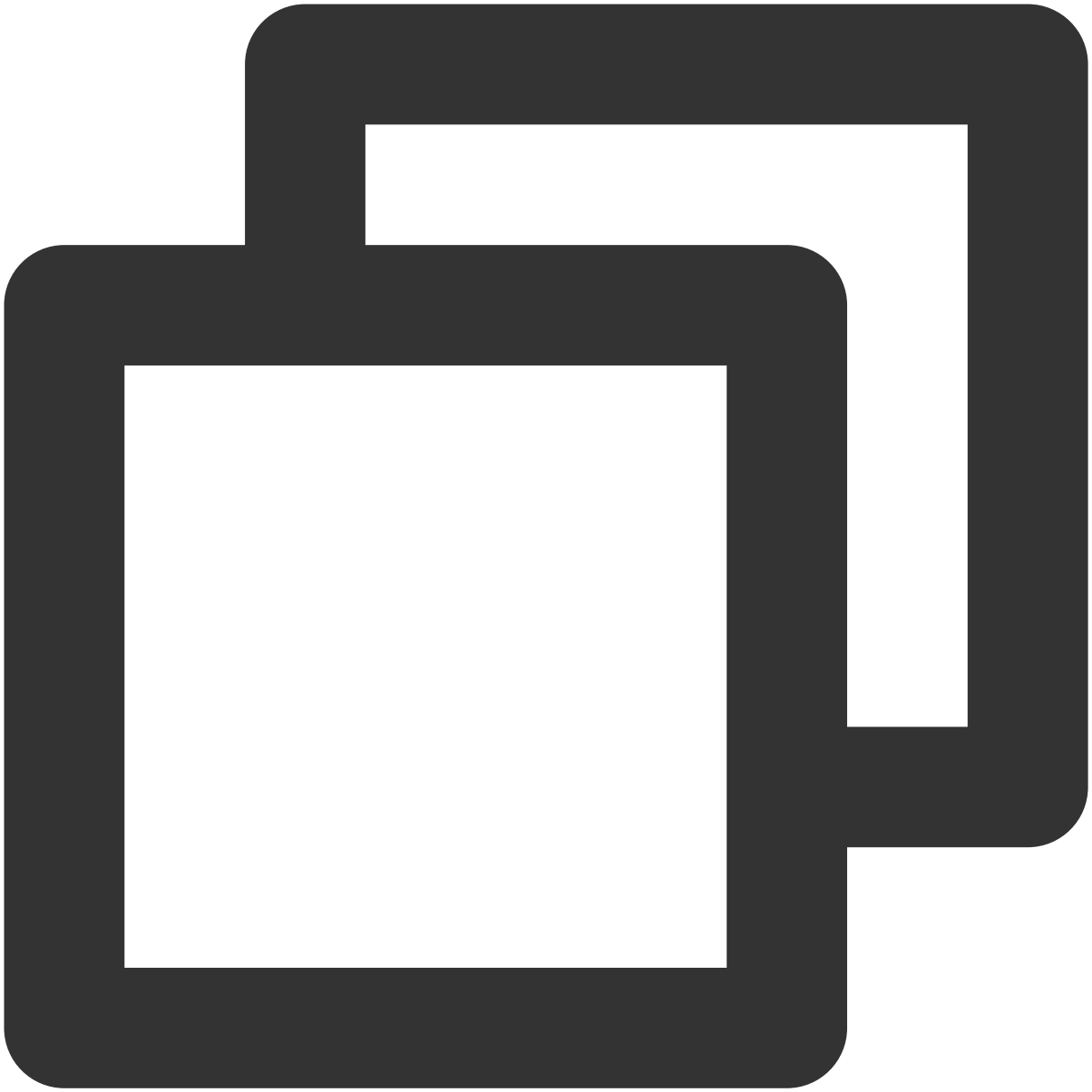
Description

Supported engines: Presto and SparkSQL

Applicable table: Native Iceberg tables and external tables

Purpose: Add a single partition field to a data table.

Standard Statement



```
ALTER TABLE table_name ADD PARTITION partition_column | hidden_partition_spec [AS a

hidden_partition_spec:
    Supported transformations are:
        years(ts): partition by year
        months(ts): partition by month
        days(ts) or date(ts): equivalent to dateint partitioning
        hours(ts) or date_hour(ts): equivalent to dateint and hour partitioning
        bucket(N, col): partition by hashed value mod N buckets
        truncate(L, col): partition by value truncated to L
            Strings are truncated to the given length
```

```
              Integers and longs truncate to bins: truncate(10, i) produces partition
```

Parameter Description

`table_name` : Name of the target table

`partition_column` : Partitioning column

`alias` : Alias of the partitioning column to be added.

Example

```
ALTER TABLE prod.db.sample ADD PARTITION FIELD bucket(16, id)
ALTER TABLE prod.db.sample ADD PARTITION FIELD truncate(data, 4)
ALTER TABLE prod.db.sample ADD PARTITION FIELD years(ts)
-- use optional AS keyword to specify a custom name for the partition field
ALTER TABLE prod.db.sample ADD PARTITION FIELD bucket(16, id) AS shard
```

ALTER TABLE DROP PARTITION FIELD

Last updated：2024-08-07 17:17:00

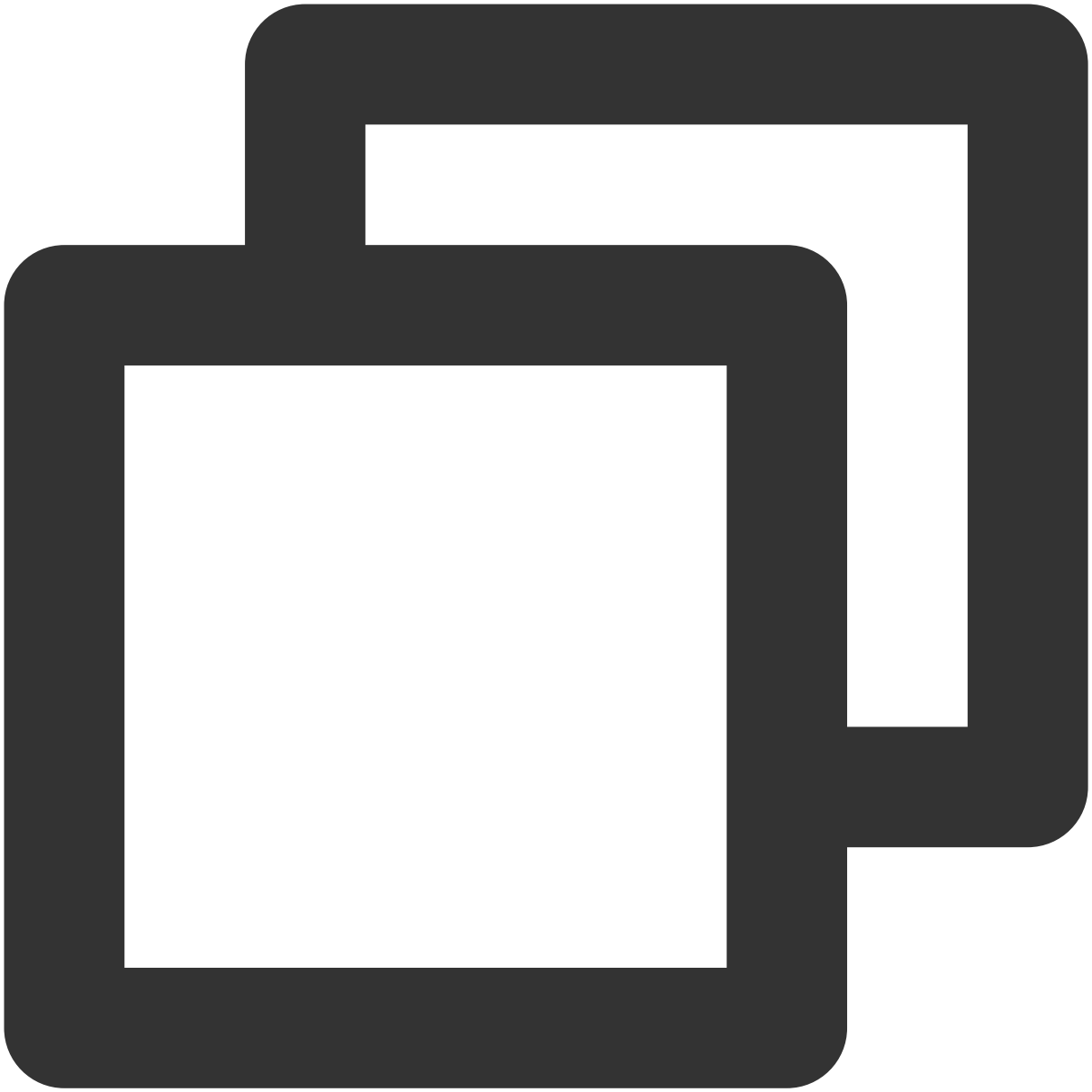
Description

Supported engines: Presto and SparkSQL

Applicable table: Native Iceberg tables and external tables

Purpose: Delete a partition field of a data table.

Standard Statement



```
ALTER TABLE table_name ADD PARTITION partition_column | hidden_partition_spec

hidden_partition_spec:
    Supported transformations are:
        years(ts): partition by year
        months(ts): partition by month
        days(ts) or date(ts): equivalent to dateint partitioning
        hours(ts) or date_hour(ts): equivalent to dateint and hour partitioning
        bucket(N, col): partition by hashed value mod N buckets
        truncate(L, col): partition by value truncated to L
            Strings are truncated to the given length
```

```
                Integers and longs truncate to bins: truncate(10, i) produces partition
```

Parameter Description

`table_name` : Name of the target table

`partition_column` : Partitioning column.

Example

```
ALTER TABLE prod.db.sample DROP PARTITION FIELD catalog
ALTER TABLE prod.db.sample DROP PARTITION FIELD bucket(16, id)
ALTER TABLE prod.db.sample DROP PARTITION FIELD truncate(data, 4)
ALTER TABLE prod.db.sample DROP PARTITION FIELD years(ts)
ALTER TABLE prod.db.sample DROP PARTITION FIELD shard
```

ALTER TABLE ... RENAME COLUMN

Last updated：2024-08-07 17:17:13

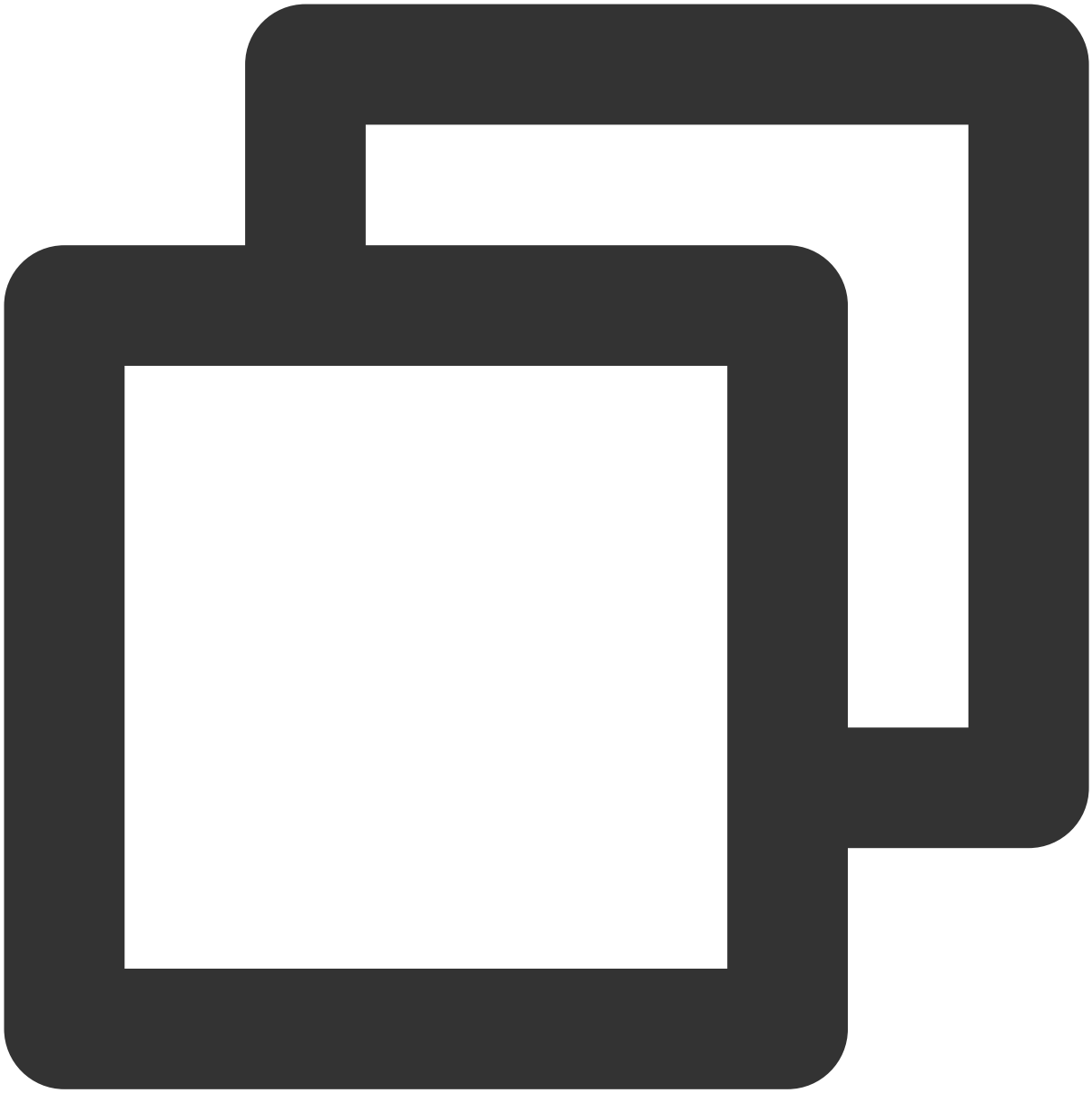
Description

Supported engine: SparkSQL

Applicable table: External and native Iceberg tables

Purpose: Change field name.

Statement



```
ALTER TABLE table_identifier
RENAME COLUMN old_column_name TO new_column_name
```

Parameter

`table_identifier` : Data table name

`old_column_name` : Field name to be changed

`new_column_name` : Field name after the change

Example

```
alter table iceberg_rename rename column id to id_2
```

ALTER TABLE SET TBLPROPERTIES

Last updated：2024-08-07 17:17:31

Description

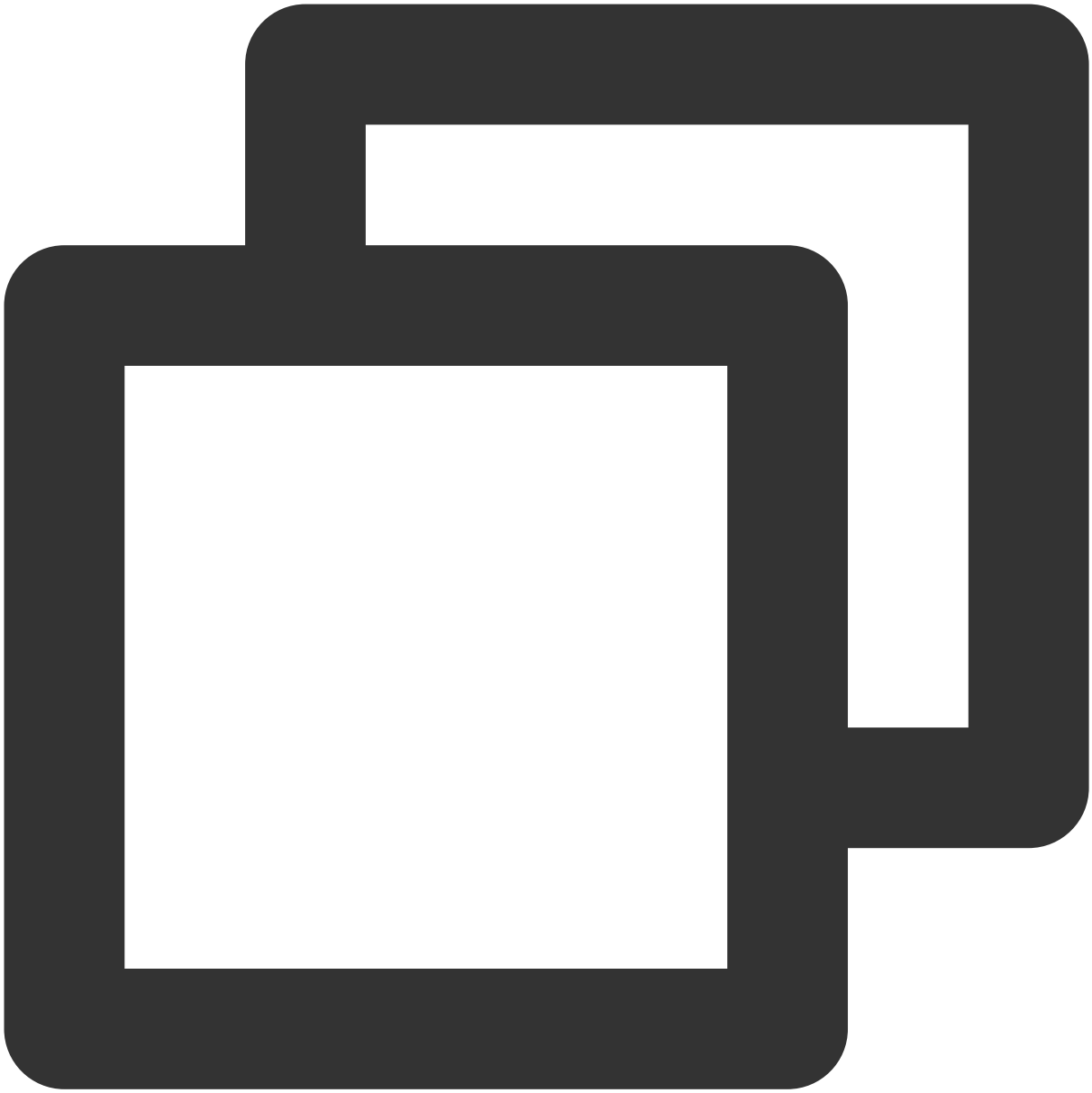
Supported engines: Presto and SparkSQL

Applicable table: Native Iceberg table and external table.

Purpose: Update/delete the data table attribute.

SET: Update Attribute

Statement



```
ALTER TABLE table_name
SET TBLPROPERTIES (property_name=property_value, ...)
```

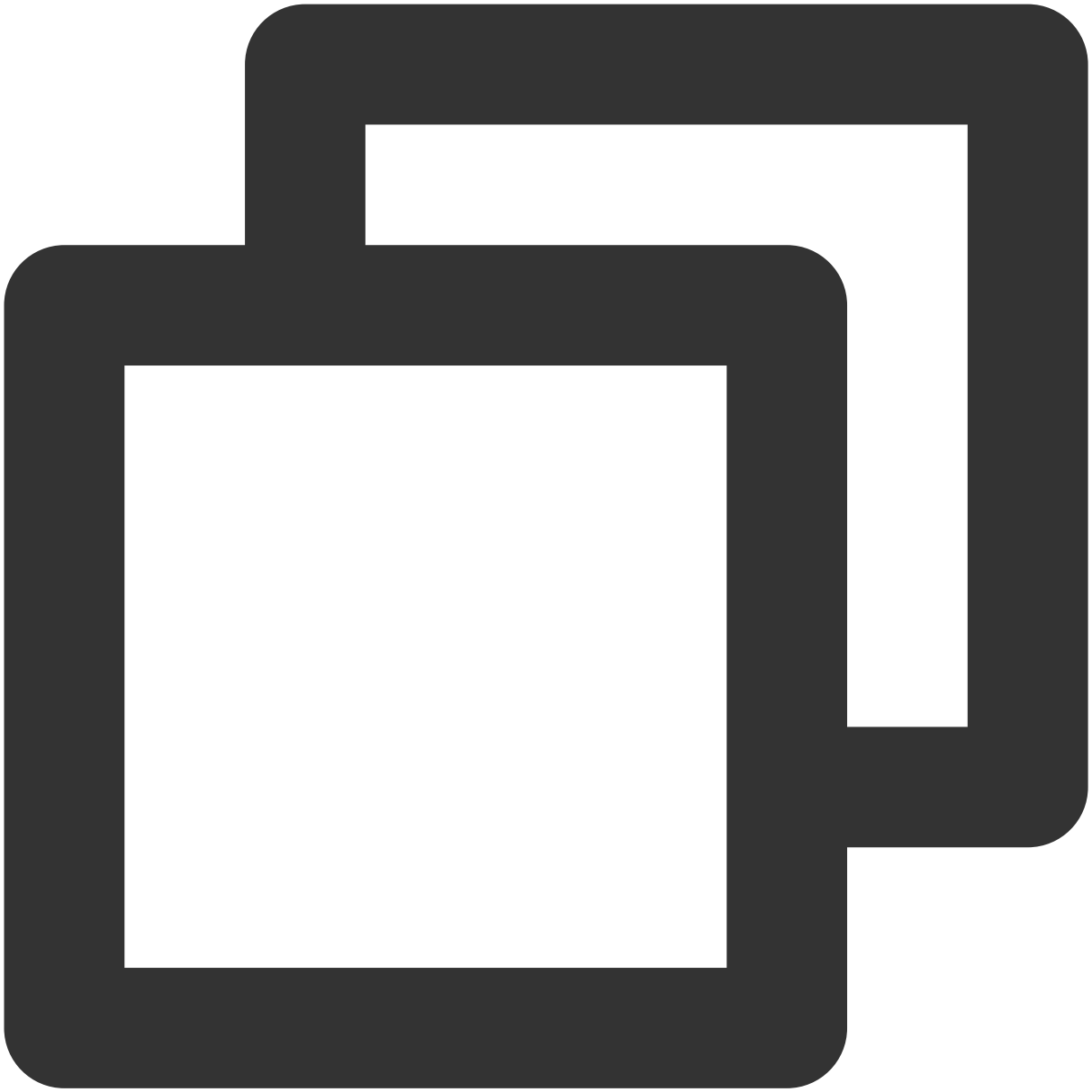
Parameter

`table_name` : Name of the target table.

`property_name` : Name of the attribute to be updated.

`property_value` : Value of the attribute to be updated.

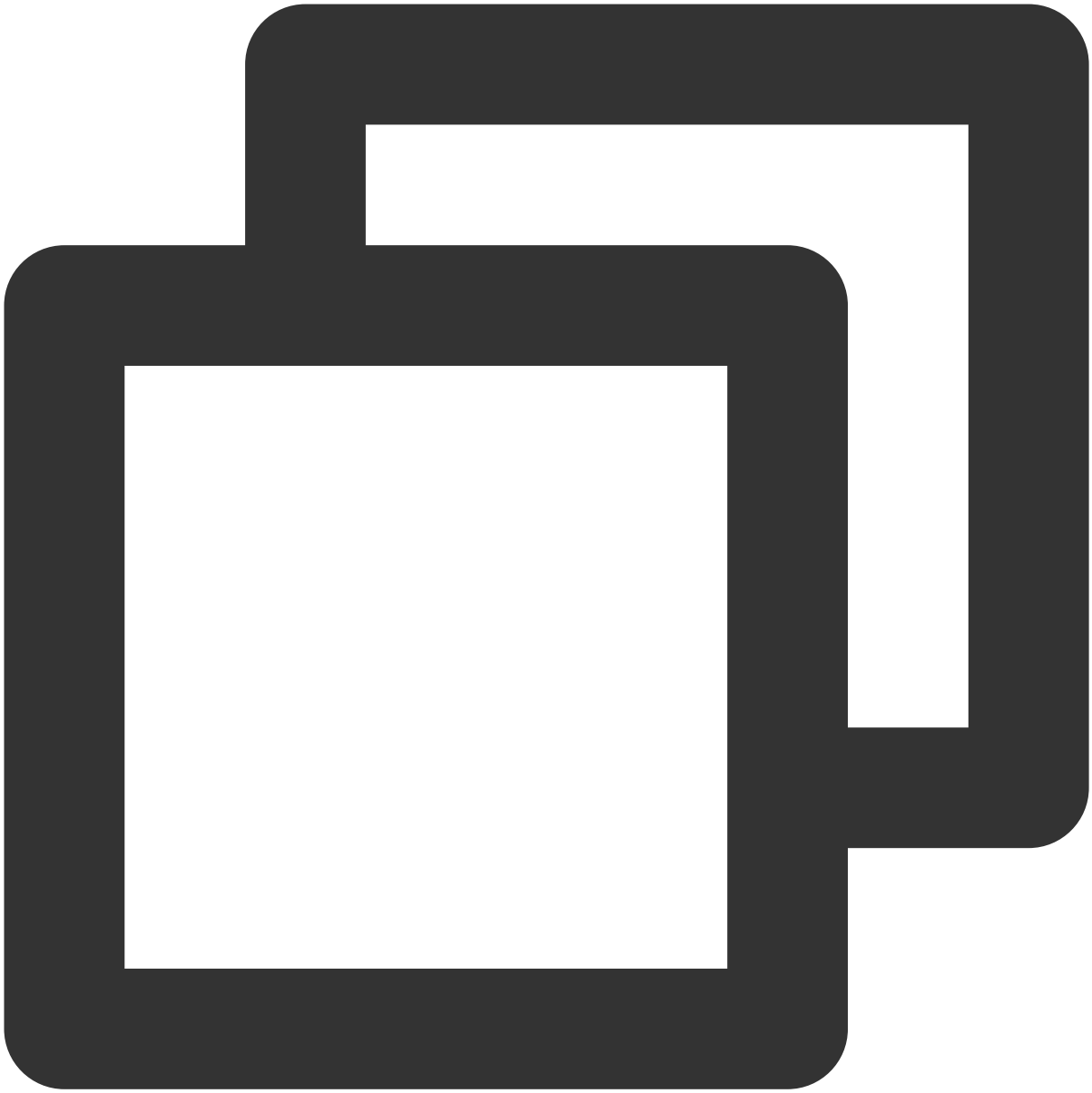
Example



```
ALTER TABLE orders SET TBLPROPERTIES ('notes'="Please don't drop this table.");
```

UNSET: Delete Attribute

Statement



```
ALTER TABLE table_name
UNSET TBLPROPERTIES (property_name, ...)
```

Parameter

`table_name` : Name of the target table

`property_name` : Name of the attribute to be deleted.

Example

```
ALTER TABLE dempts UNSET TBLPROPERTIES ('read.split.target-size')
```

ALTER TABLE SET LOCATION

Last updated：2024-08-07 17:17:49

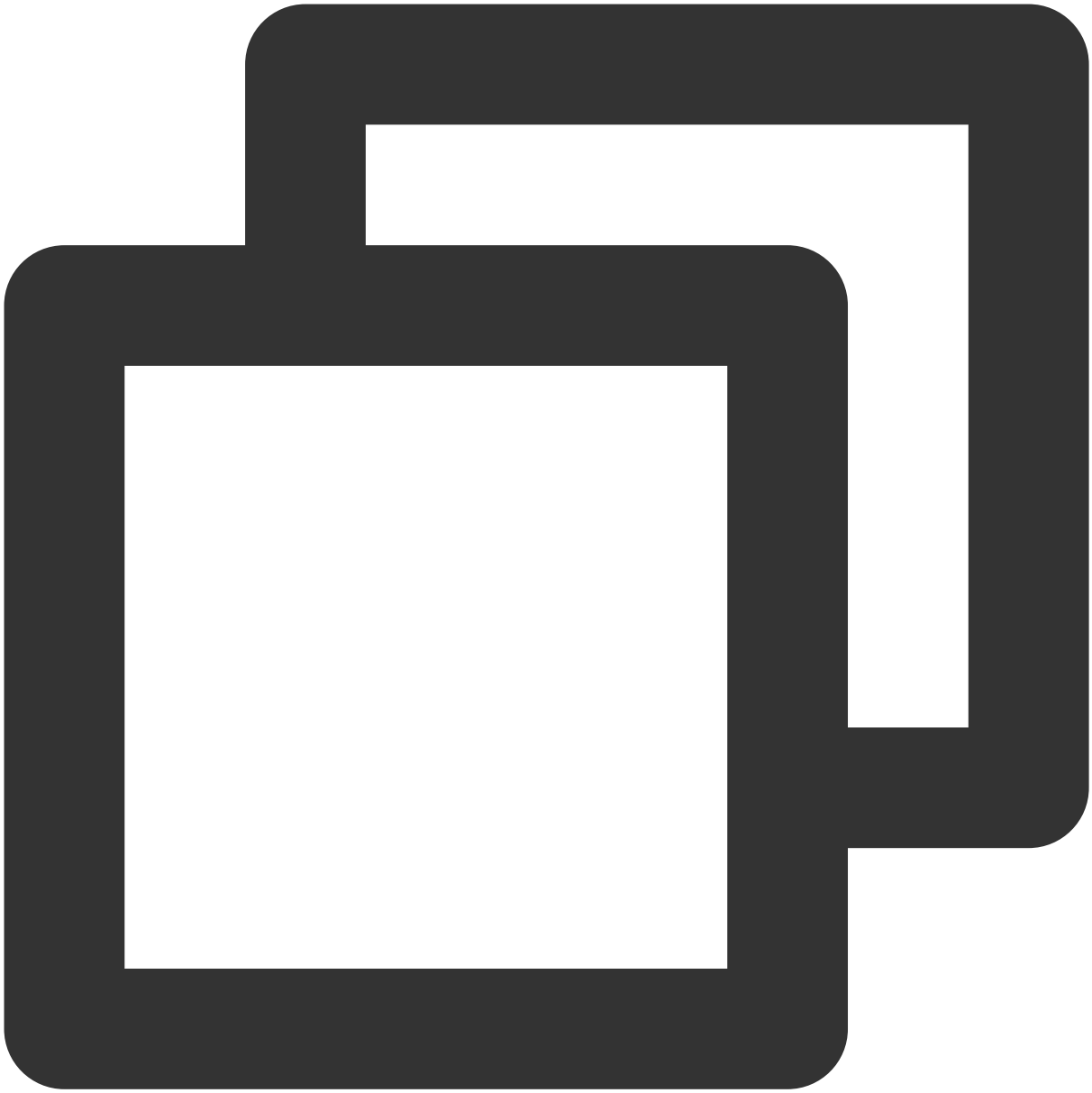
Description

Supported engines: Presto and SparkSQL

Applicable table: Native Iceberg table and external table.

Purpose: Change the storage path of data table.

Standard Statement



```
ALTER TABLE table_name [ PARTITION (partition_spec) ] SET LOCATION 'new location';
```

Parameter

`table_name` : Data table name

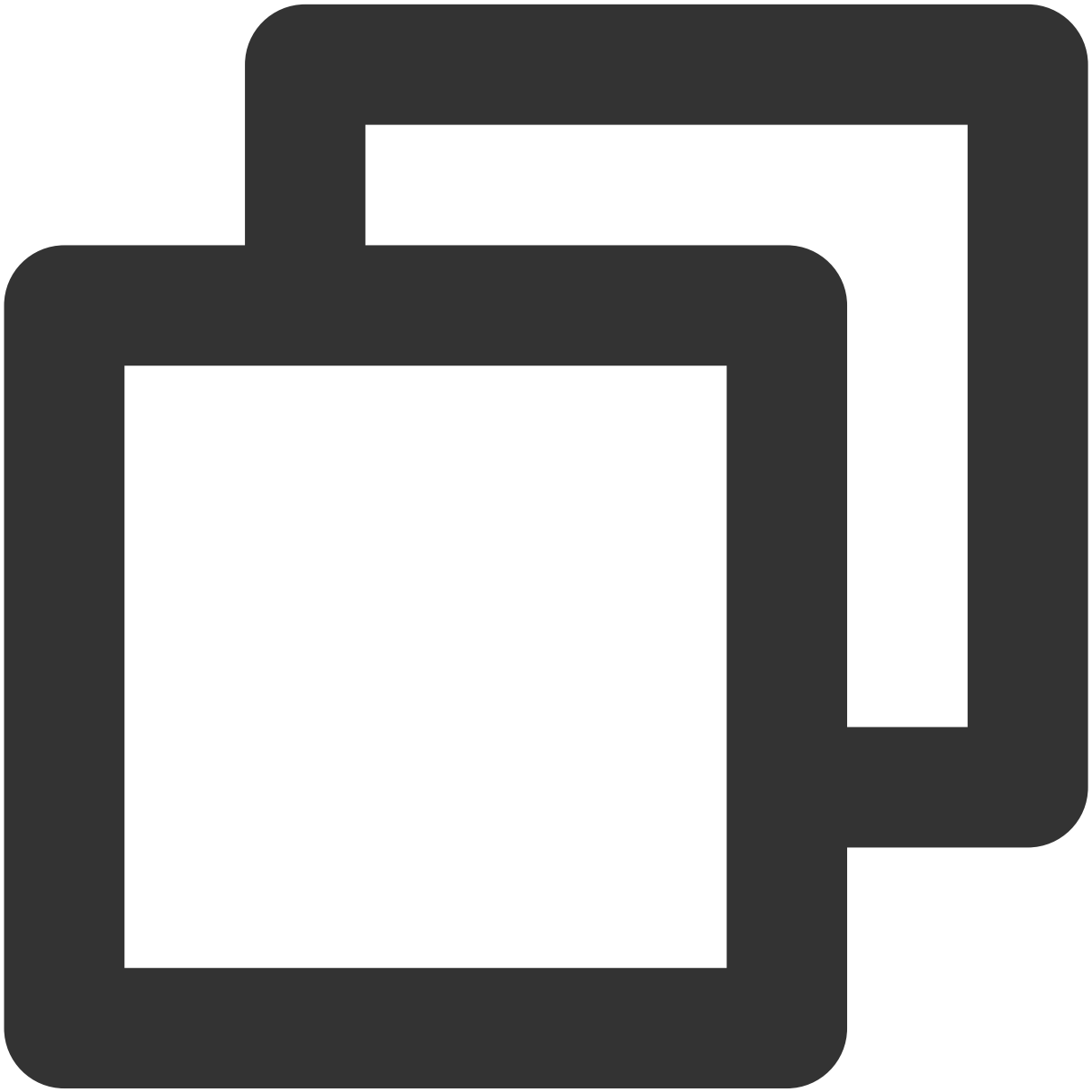
`PARTITION (partition_spec)` : Specify the partitioning column.

partition_col_name: Partitioning column name

partition_col_value: Partitioning column value

`'new location'` : New location of the table or partition on Tencent Cloud COS.

Example



```
ALTER TABLE tbl PARTITION (a='1', b='2') SET LOCATION '/path/to/part/ways';

ALTER TABLE tbl SET LOCATION '/path/to/part/ways';
```

ALTER TABLE ... WRITE ORDERED BY

Last updated：2024-08-07 17:18:07

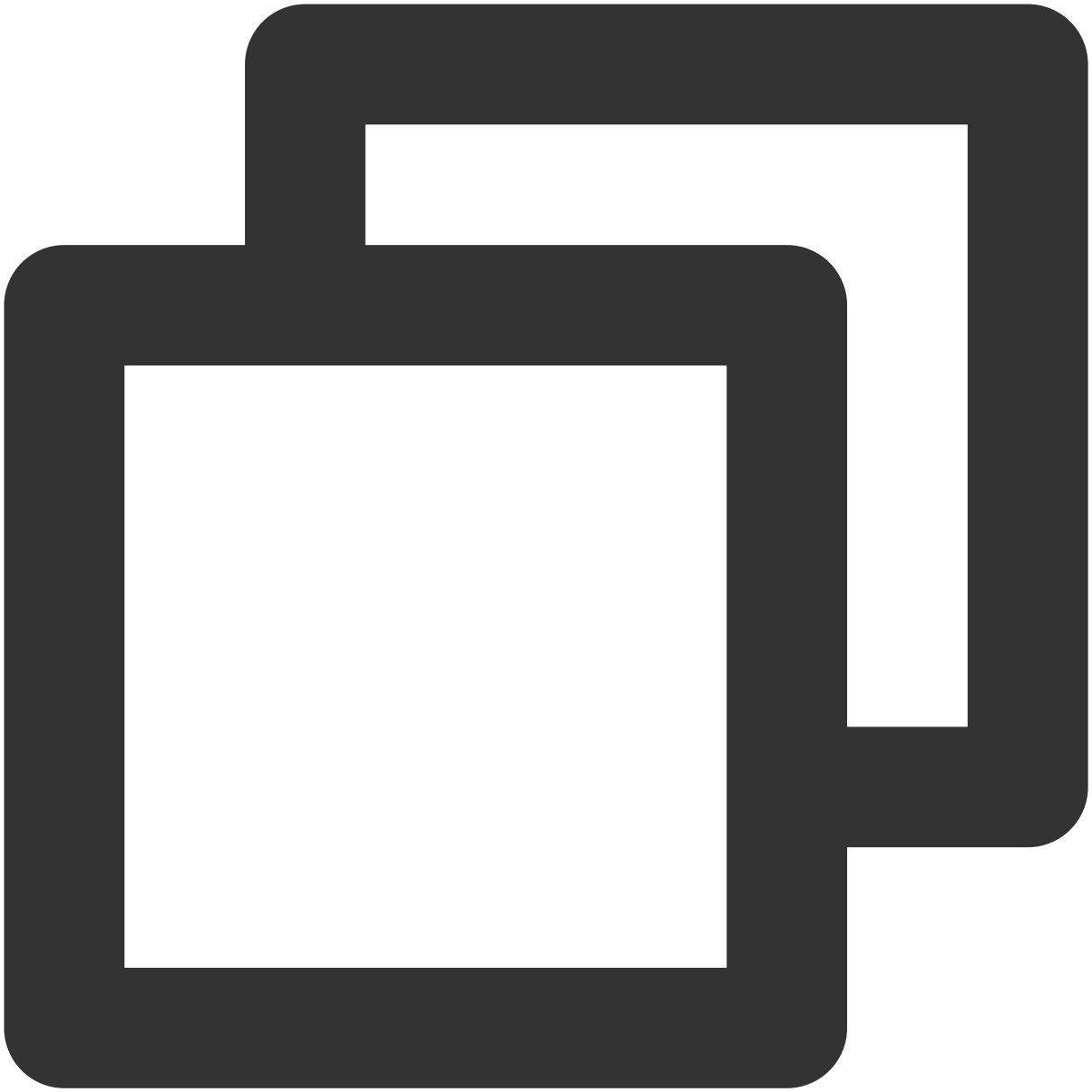
Description

Supported engine: SparkSQL

Applicable table: External and native Iceberg tables

Purpose: Set the sorting method data insertion into a table.

Statement

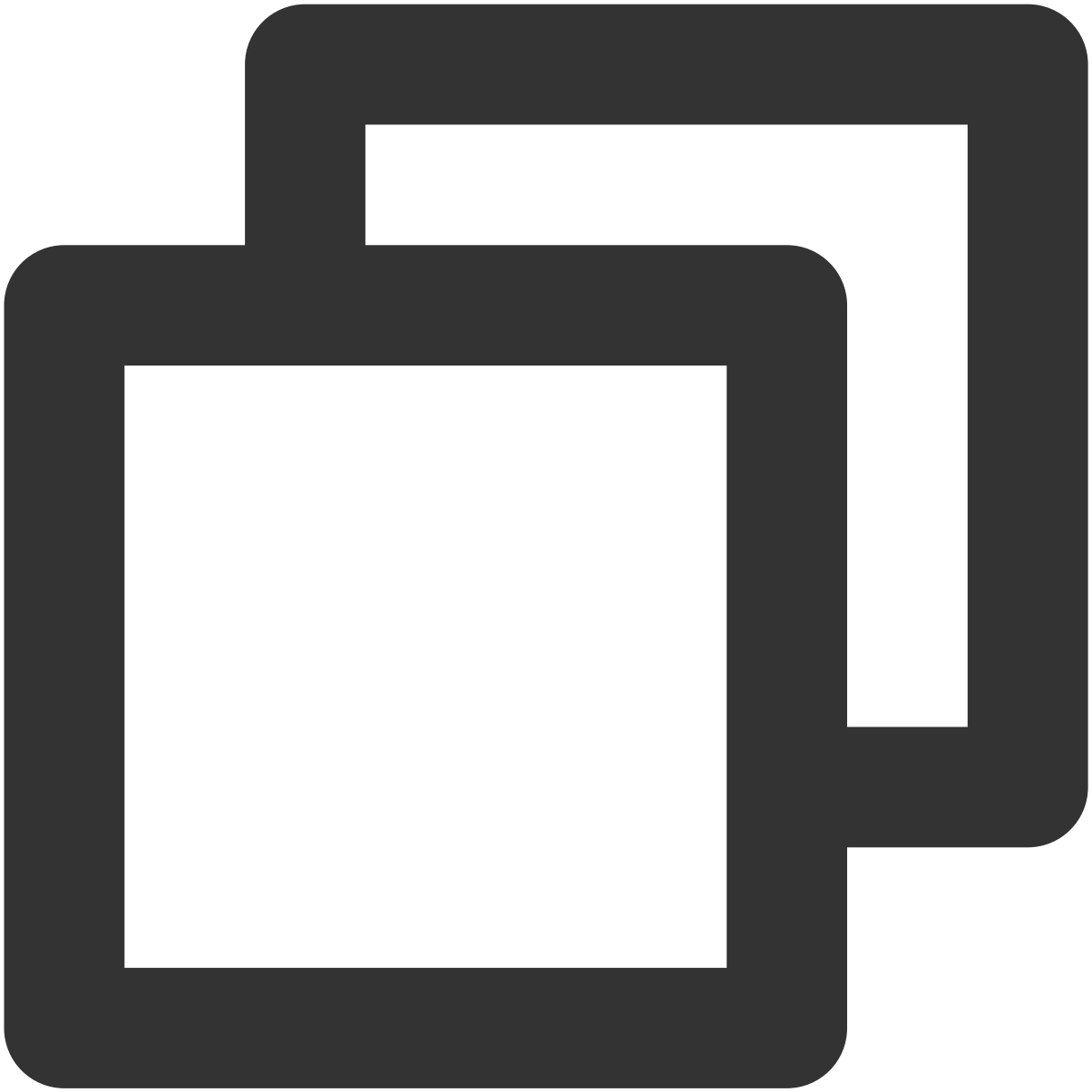


```
ALTER TABLE table_identifier
WRITE [LOCALLY] ORDERED BY
{col_name [ASC|DESC] [NULLS FIRST|LAST]}[, ...]
```

Parameter

`table_identifier` : Data table name

Example



```
ALTER TABLE dempts WRITE ORDERED BY category, id;
-- use optional ASC/DEC keyword to specify sort order of each field (default ASC)
ALTER TABLE dempts WRITE ORDERED BY category ASC, id DESC;
-- use optional NULLS FIRST/NULLS LAST keyword to specify null order of each field
ALTER TABLE dempts WRITE ORDERED BY category ASC NULLS LAST, id DESC NULLS FIRST;
-- To order within each task, not across tasks
ALTER TABLE dempts WRITE LOCALLY ORDERED BY category, id;
```

ALTER TABLE ... WRITE DISTRIBUTED BY PARTITION

Last updated：2024-08-07 17:18:21

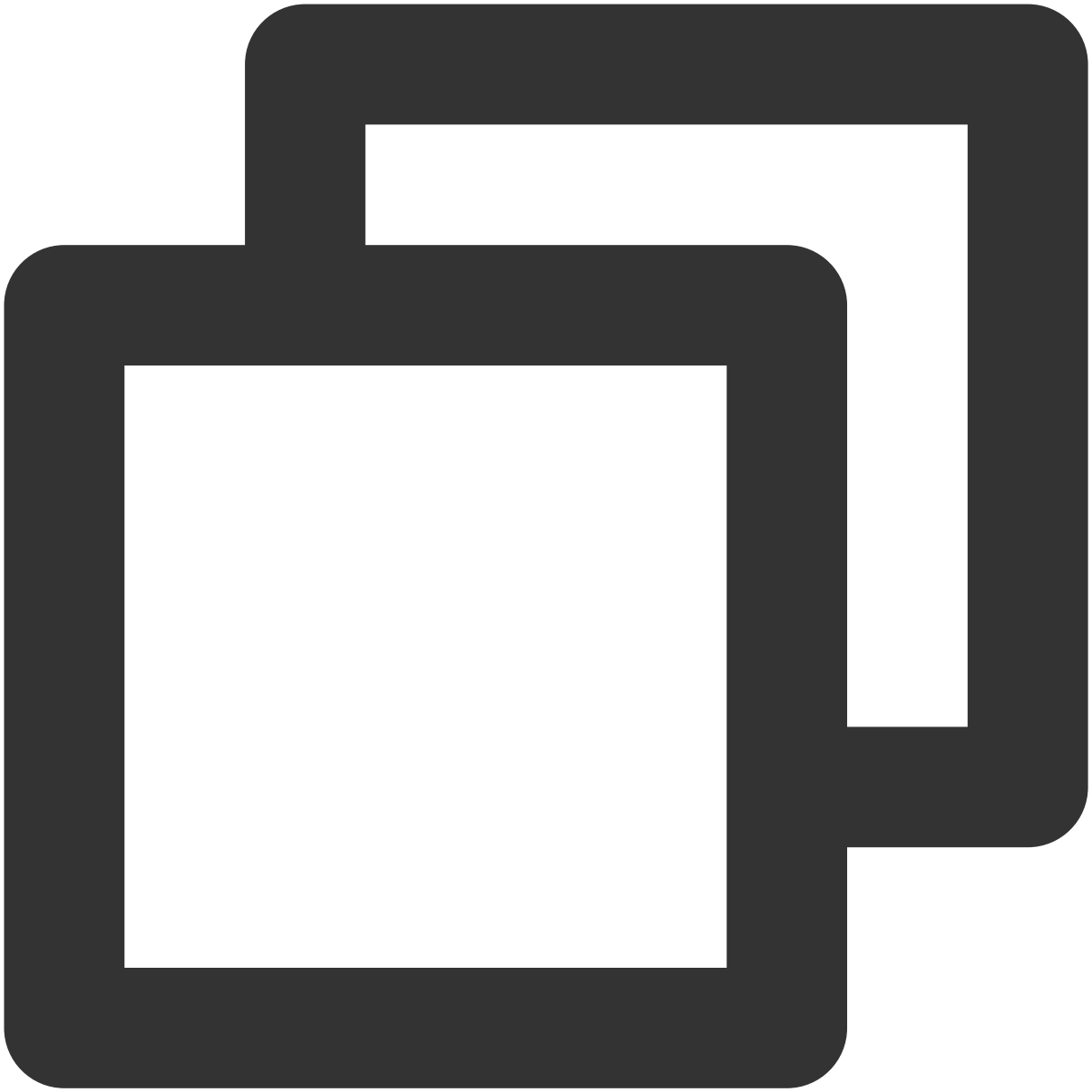
Description

Supported engine: SparkSQL

Applicable table: External and native Iceberg tables

Purpose: Change data distribution policy of partitioned table.

Statement



```
ALTER TABLE table_identifier
WRITE DISTRIBUTED BY PARTITION
[ LOCALLY ORDERED BY
{col_name [ASC|DESC] [NULLS FIRST|LAST]}[, ...]]
```

Example

```
ALTER TABLE dempts WRITE DISTRIBUTED BY PARTITION;
ALTER TABLE dempts WRITE DISTRIBUTED BY PARTITION LOCALLY ORDERED BY id;
```

ALTER TABLE ... SET IDENTIFIER FIELDS

Last updated：2024-08-07 17:19:05

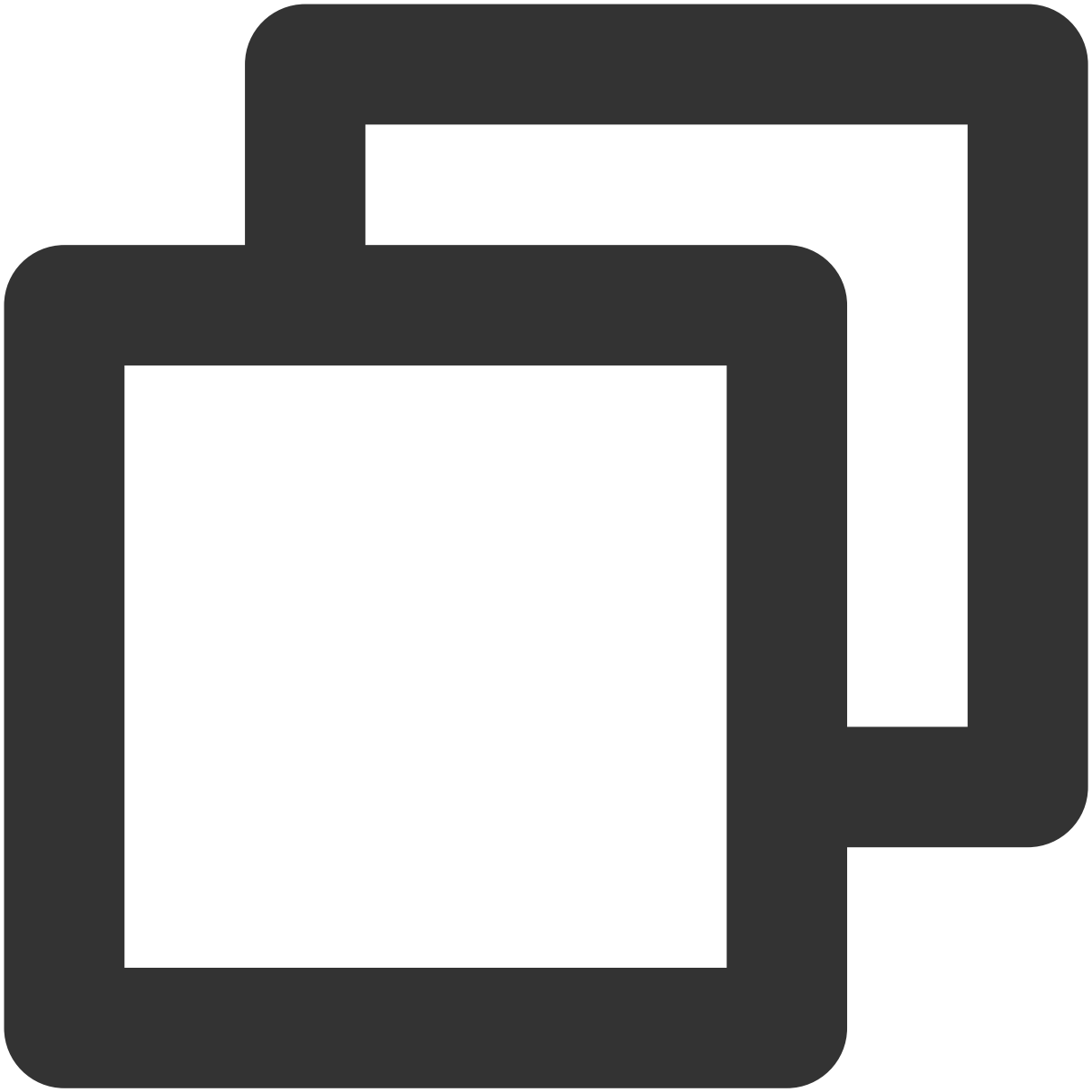
Description

Supported engine: SparkSQL

Applicable table: External and native Iceberg tables

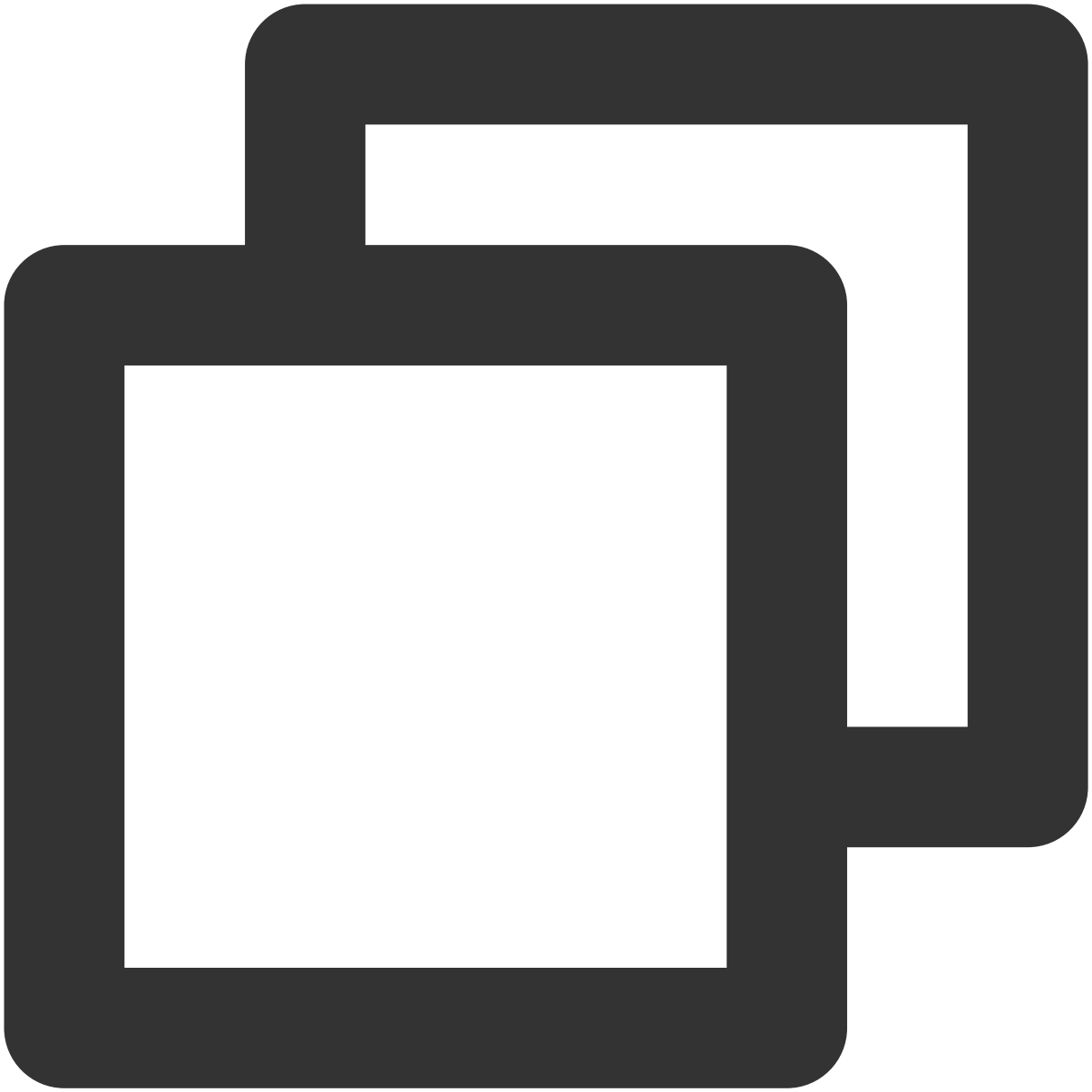
Purpose: Add identifier fields.

Statement



```
ALTER TABLE dempts SET IDENTIFIER FIELD empno, name
```

Example



```
ALTER TABLE tb1 SET IDENTIFIER FIELDS id, location.lon
```

ALTER TABLE ... DROP IDENTIFIER FIELDS

Last updated：2024-08-07 17:19:28

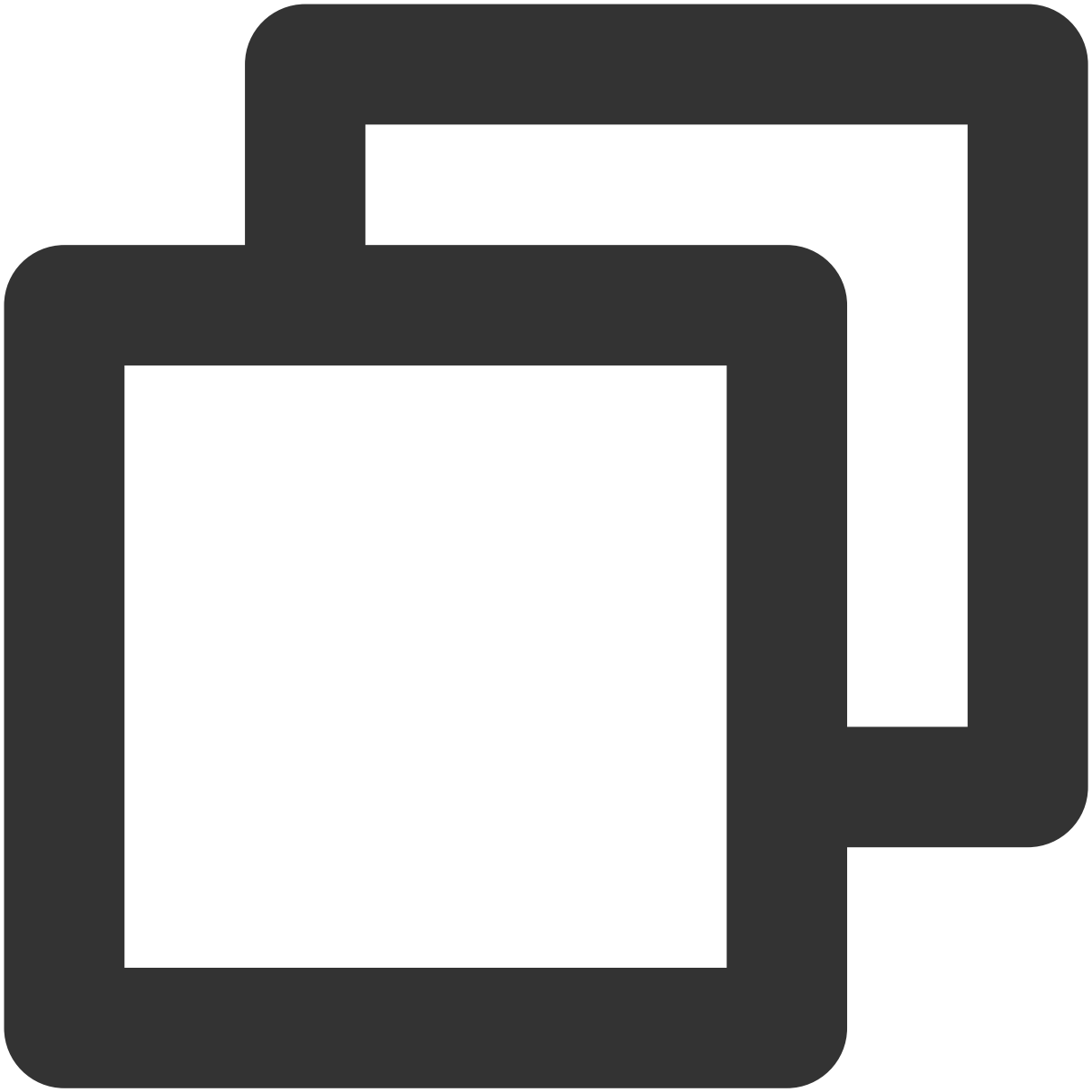
Description

Supported engine: SparkSQL

Applicable table: External and native Iceberg tables

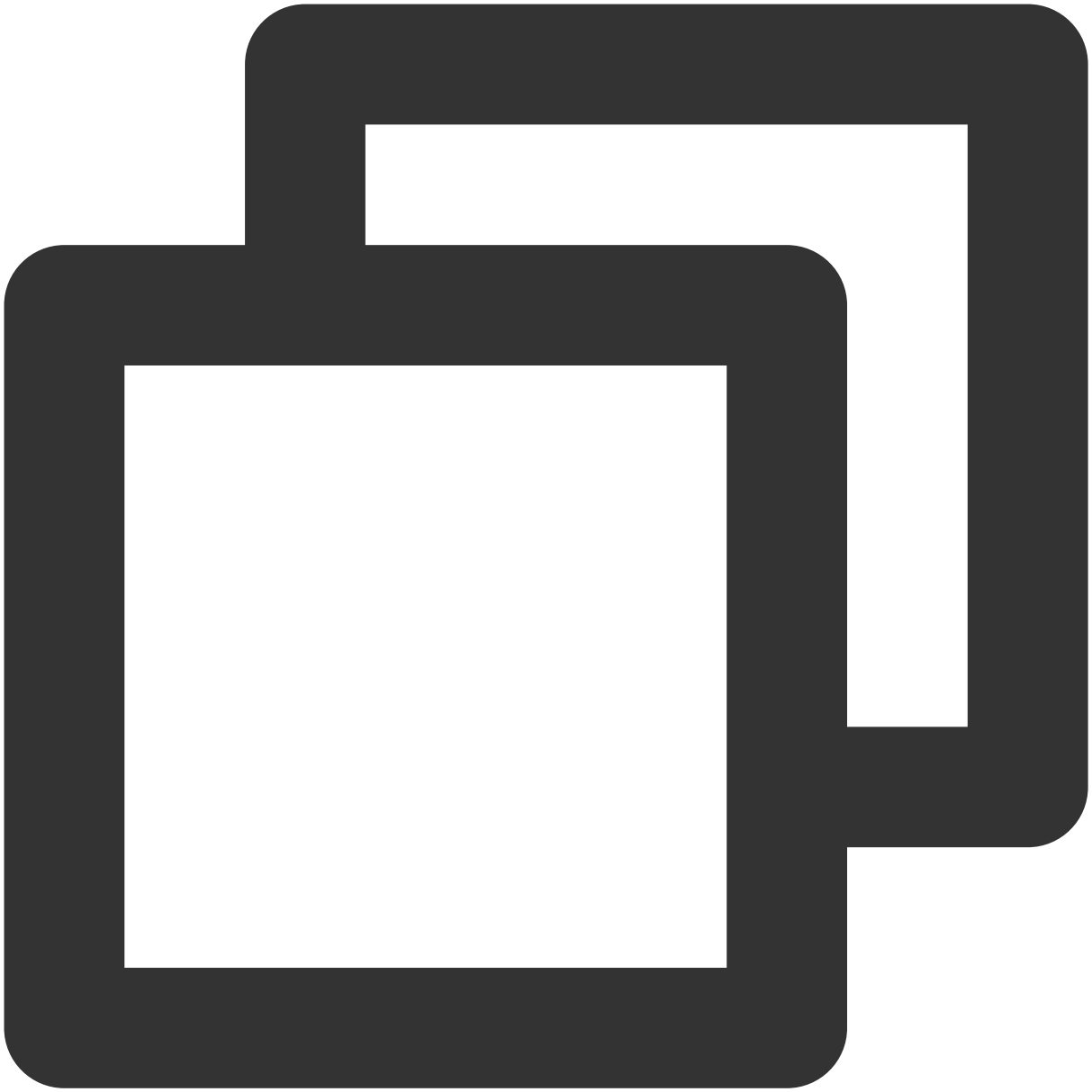
Purpose: Delete identifier fields.

Statement



```
ALTER TABLE dempts DROP IDENTIFIER FIELD empno, name
```

Example



```
ALTER TABLE tb1 DROP IDENTIFIER FIELDS id, location.lon
```

MSCK REPAIR TABLE

Last updated：2024-08-07 17:19:40

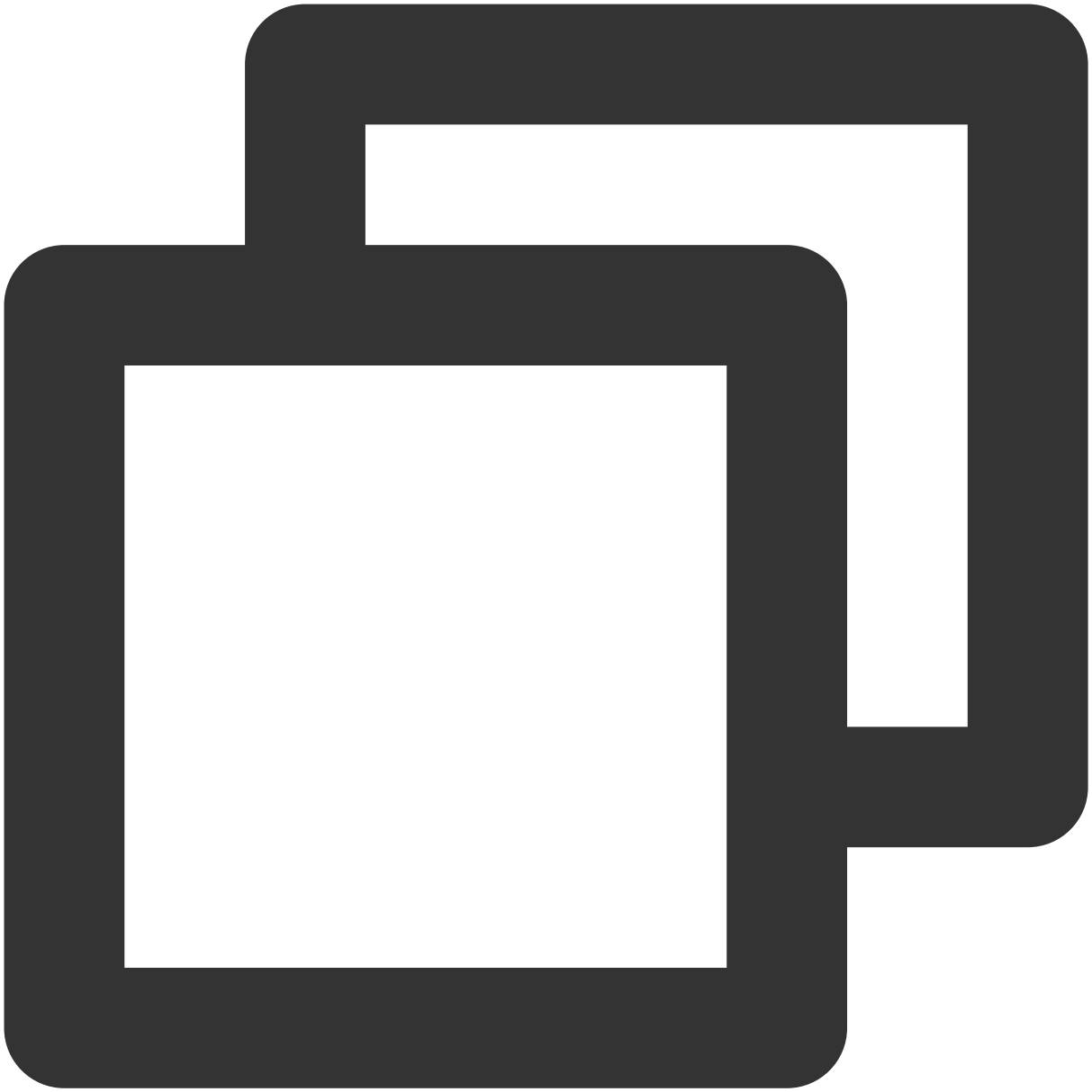
Description

Supported engines: Presto and SparkSQL

Applicable table: Native Iceberg tables and external tables

Purpose: Update partition information on the data table.

Standard Statement

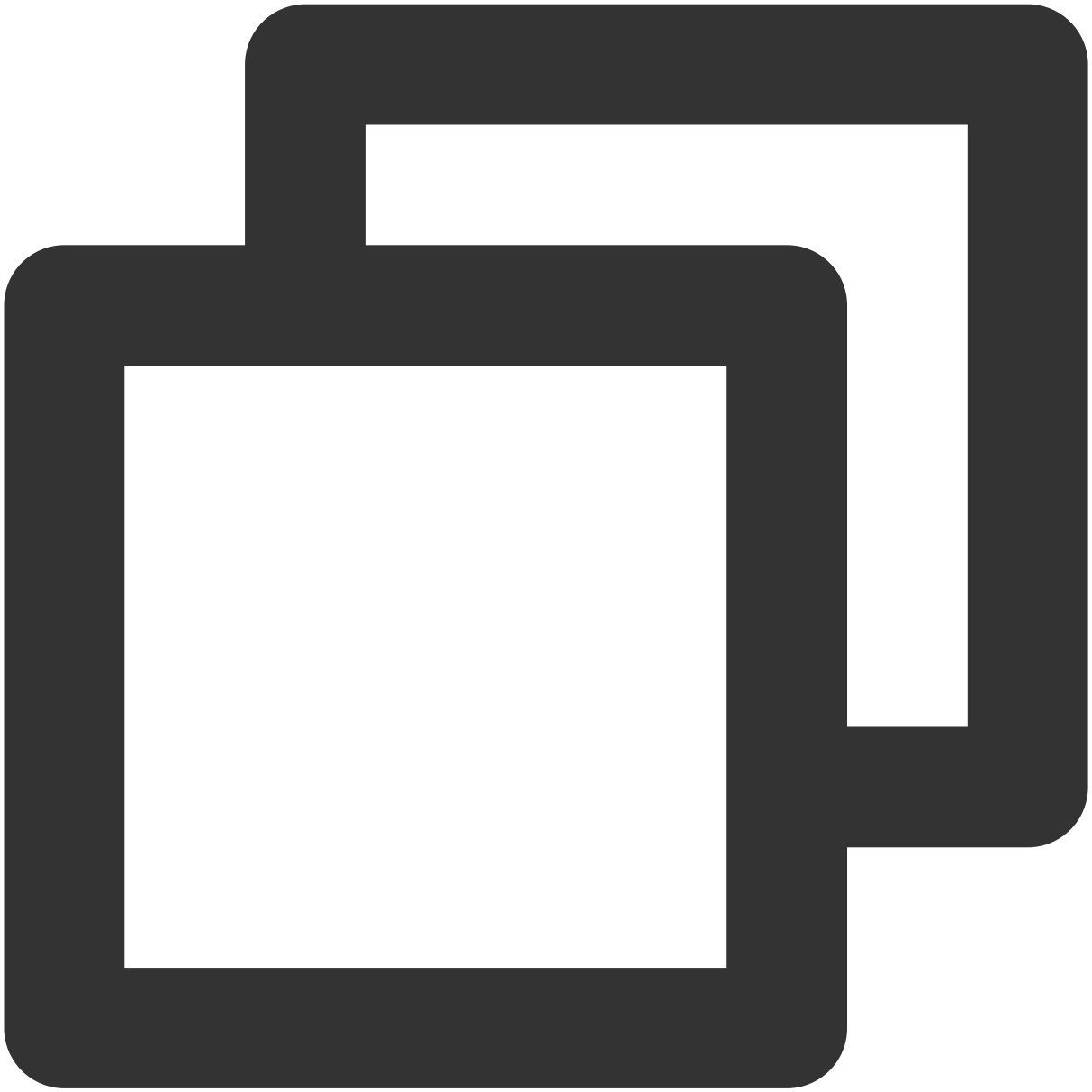


```
MSCK REPAIR TABLE table_identifier
```

Parameter

`table_identifier` : Table name.

Example



```
MSCK REPAIR TABLE t1
```

ANALYZE TABLES

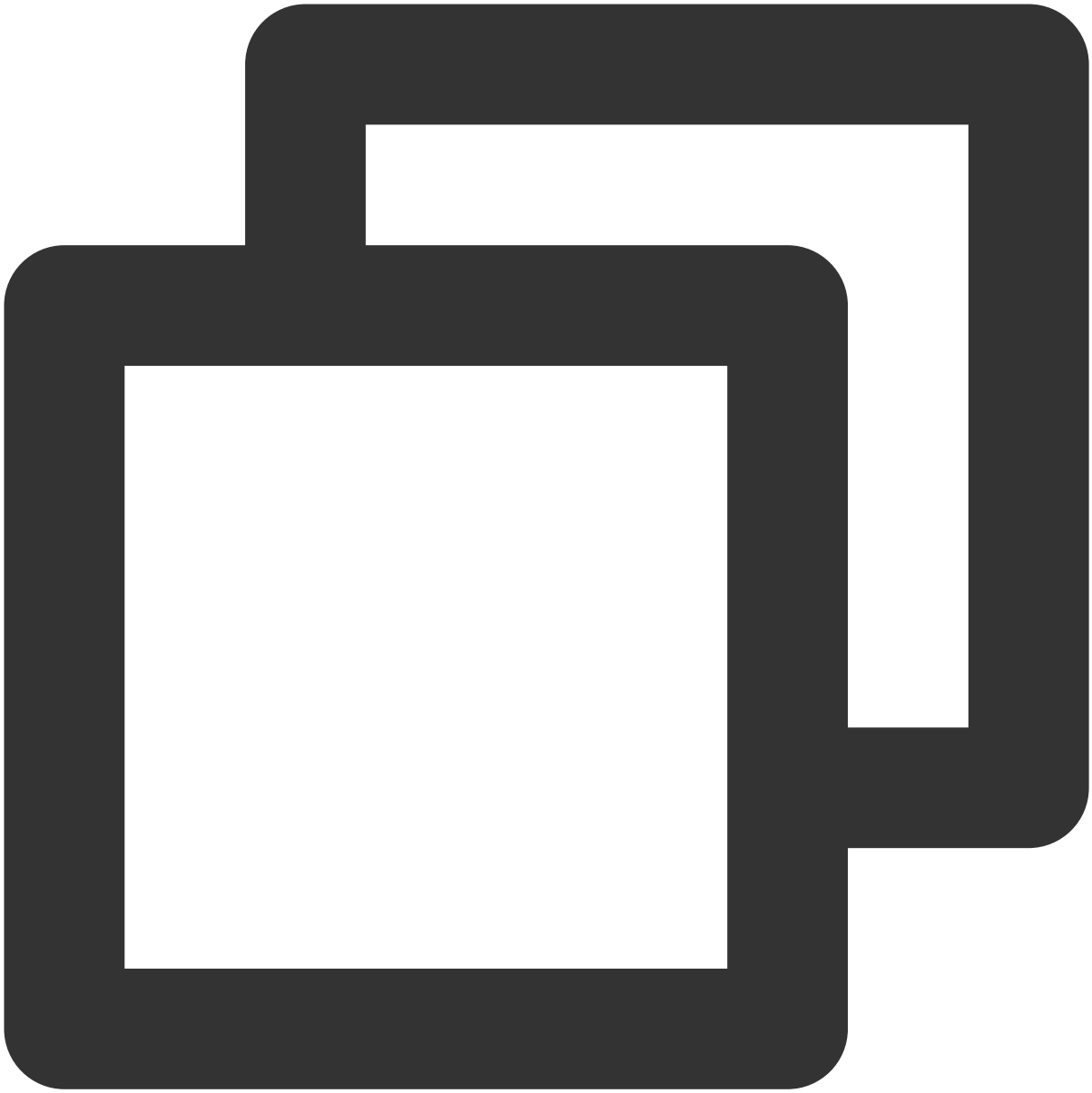
Last updated：2024-08-07 17:19:54

Description

Supported engines: Presto and SparkSQL

Purpose: Perform statistics on a database table.

Statement



```
ANALYZE TABLES [ { FROM | IN } database_name ] COMPUTE STATISTICS [ NOSCAN ]

ANALYZE TABLE table_identifier
[ PARTITION ( partition_col_name [ = partition_col_val ] [ , ... ] ) ]
COMPUTE STATISTICS [ NOSCAN | FOR COLUMNS col [ , ... ] | FOR ALL COLUMNS ]
```

Parameter

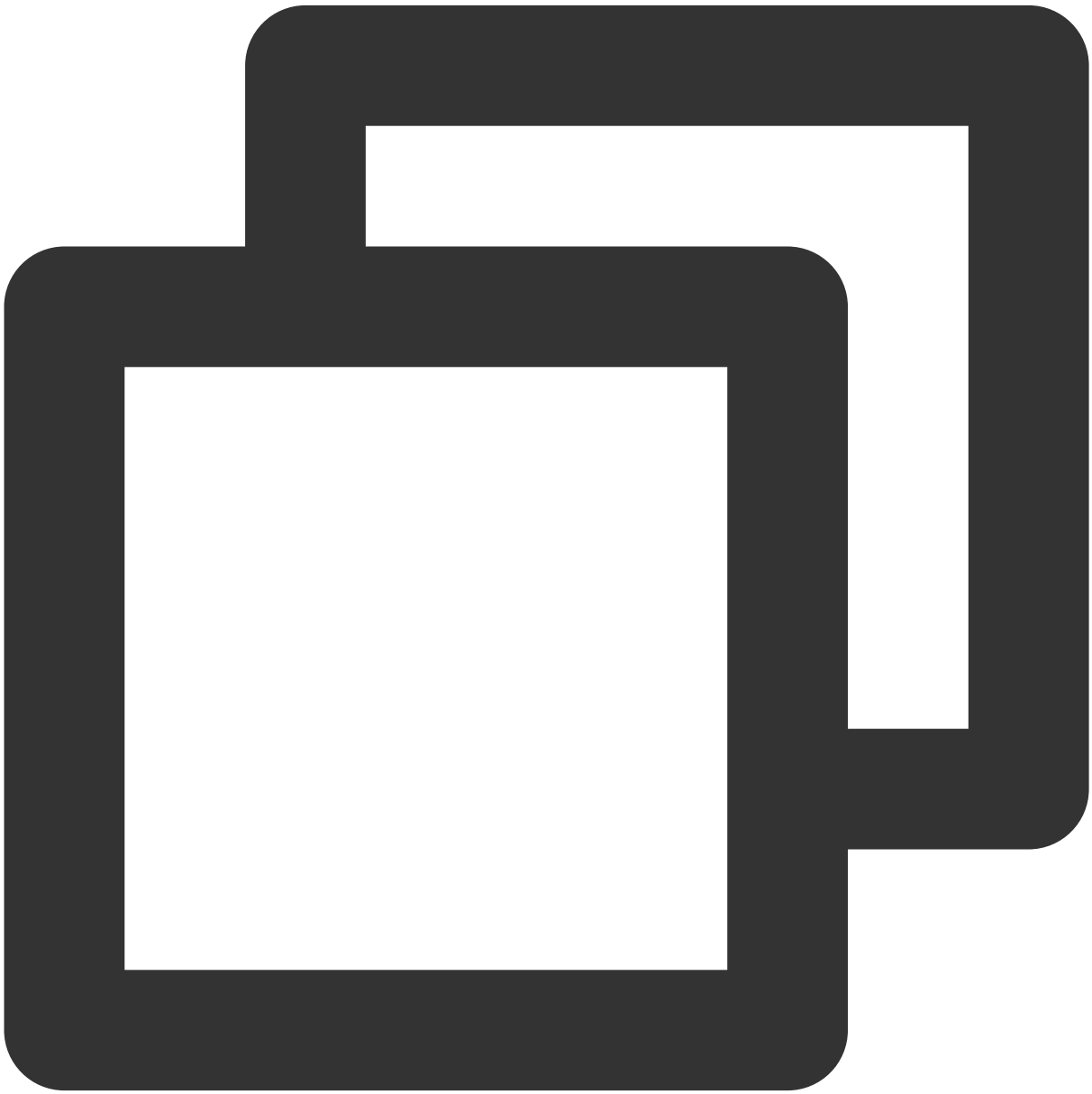
`database_name` : Name of the database on which table statistics are performed.

`table_identifier` : Name of the table on which statistics are performed.

`partition_col_name` : Name of the column in the partitioned table on which statistics are performed.

`partition_col_value` : Values in the column of the partitioned table on which statistics are performed.

Example



```
ANALYZE TABLE students COMPUTE STATISTICS
```

```
ANALYZE TABLE students COMPUTE STATISTICS FOR COLUMNS name
ANALYZE TABLE db.students COMPUTE STATISTICS FOR COLUMNS name
ANALYZE TABLE students COMPUTE STATISTICS NOSCAN
ANALYZE TABLE students COMPUTE STATISTICS FOR all COLUMNS
ANALYZE TABLE db.students COMPUTE STATISTICS FOR all COLUMNS
ANALYZE TABLE students PARTITION (student_id) COMPUTE STATISTICS
ANALYZE TABLE students PARTITION (student_id = 111111) COMPUTE STATISTICS
ANALYZE TABLE db.students PARTITION (student_id = 111111, name = 'test') COMPUTE ST

ANALYZE TABLES COMPUTE STATISTICS
ANALYZE TABLES COMPUTE STATISTICS NOSCAN
ANALYZE TABLES from school_db COMPUTE STATISTICS NOSCAN
ANALYZE TABLES IN school_db COMPUTE STATISTICS NOSCAN
```

DROP TABLE

Last updated：2024-08-07 17:21:06

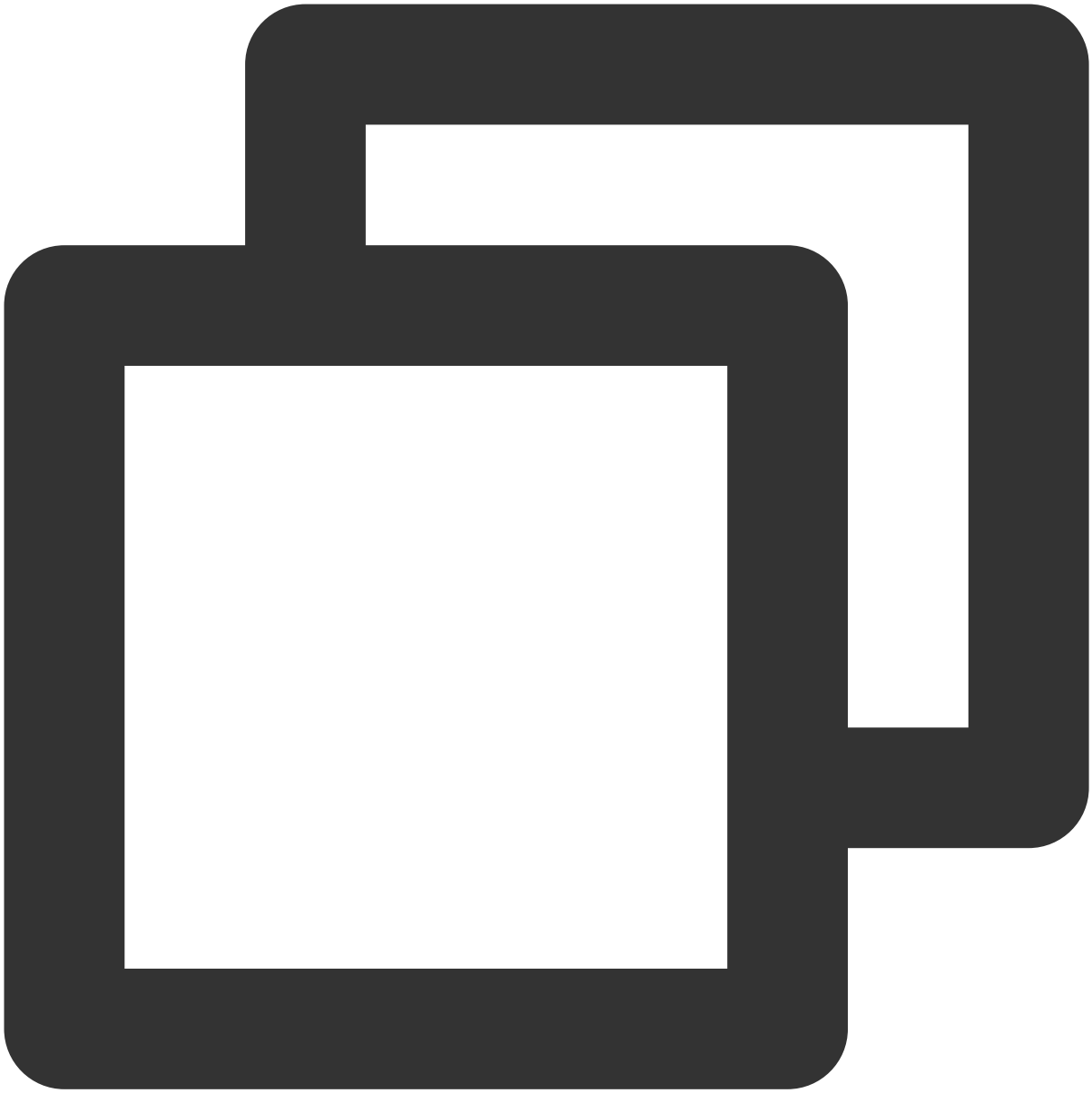
Description

Supported engines: Presto and SparkSQL

Applicable table: Native and external tables

Purpose: Remove metadata table.

Statement



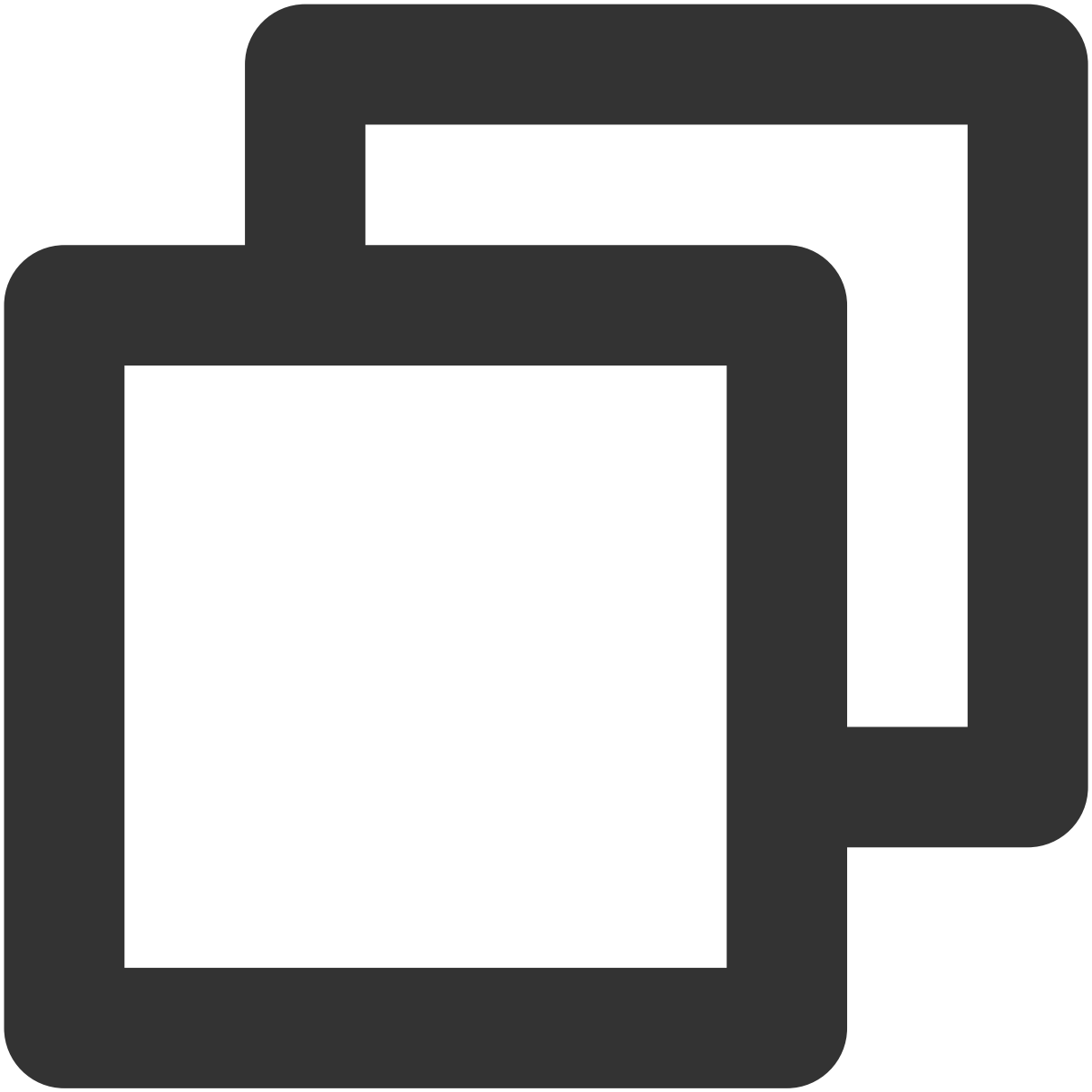
```
DROP TABLE [IF EXISTS] table_name;
```

Parameter

`IF EXISTS` : Optional. This parameter indicates whether the table exists.

`table_name` : Table name.

Example



```
DROP TABLE tbl
DROP TABLE IF EXISTS tbl
```

EXPLAIN

Last updated：2024-08-07 17:21:18

Description

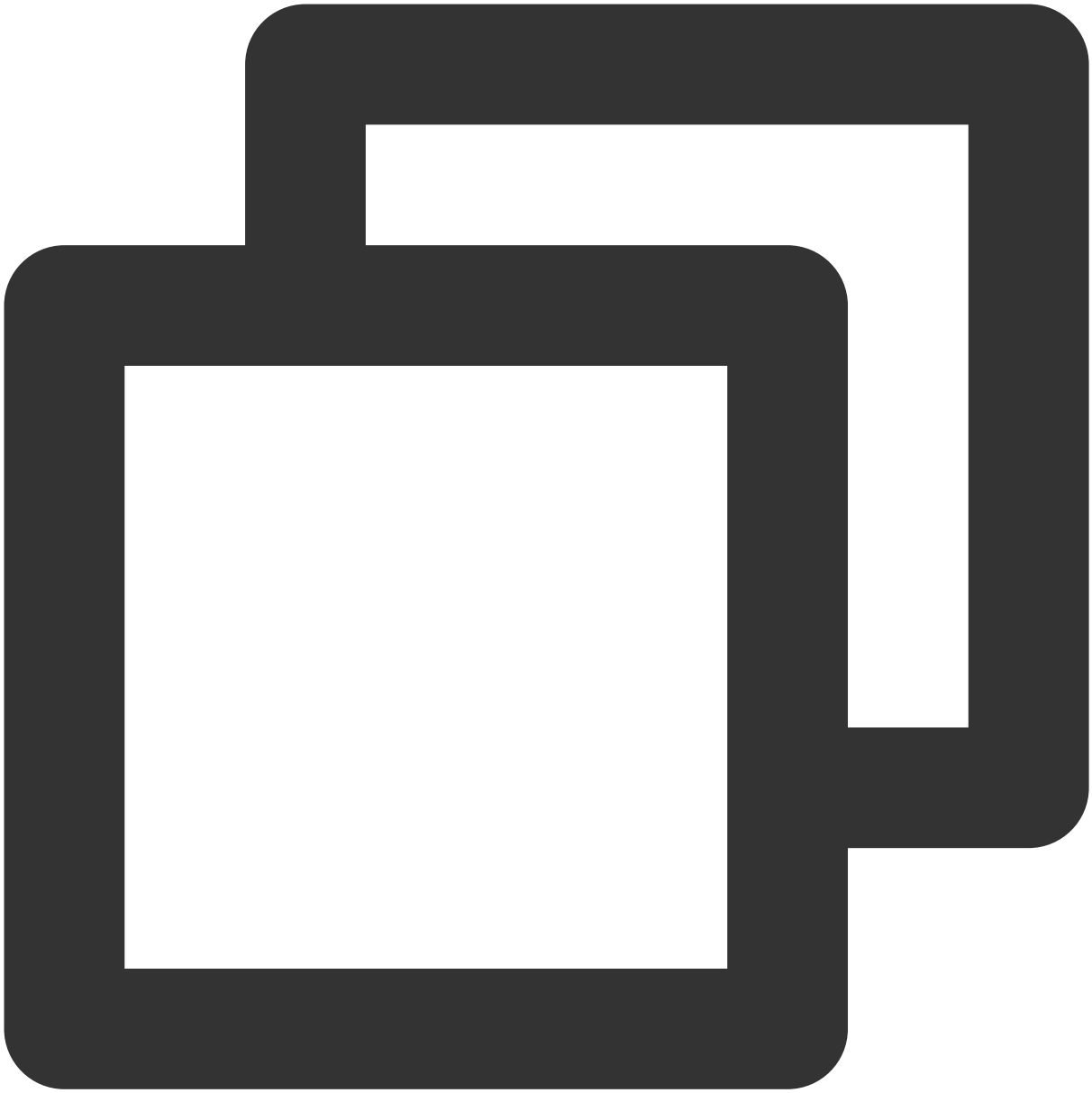
Supported engines: Presto and SparkSQL

Applicable table: Native and external tables

Purpose: Display logical or physical plan for executing SQL statements.

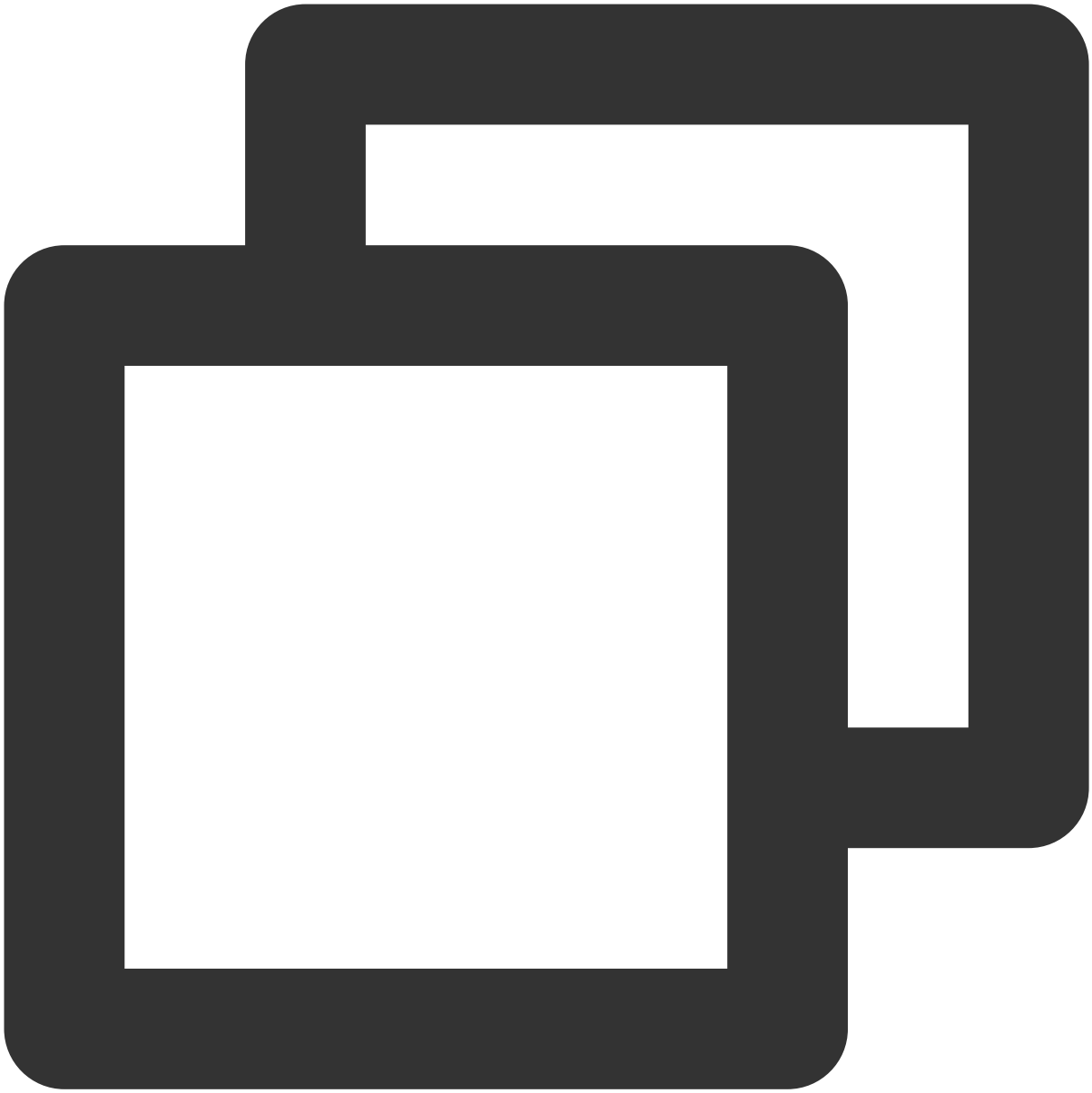
Statement

Presto



```
EXPLAIN [ ( option [, ...] ) ] statement
-- where option can be one of:
-- FORMAT { TEXT | GRAPHVIZ | JSON }
-- TYPE { LOGICAL | DISTRIBUTED | VALIDATE | IO }
```

SparkSQL



```
EXPLAIN [ EXTENDED | CODEGEN | COST | FORMATTED ] statement

EXPLAIN ANALYZE
EXPLAIN ANALYZE [VERBOSE] statement
```

Example

```
-- presto
EXPLAIN (TYPE VALIDATE) SELECT regionkey, count(*) FROM nation GROUP BY 1;
EXPLAIN (TYPE IO, FORMAT JSON) INSERT INTO test_nation SELECT * FROM nation WHERE r

-- EXPLAIN ANALYZE
EXPLAIN ANALYZE SELECT count(*), clerk FROM orders WHERE orderdate > date '1995-01-
EXPLAIN ANALYZE VERBOSE SELECT count(clerk) OVER() FROM orders WHERE orderdate > da
```

CALL STATEMENT

Last updated：2024-08-07 17:21:33

Stored procedures can be used only if you use [Iceberg SQL Extension] (https://iceberg.apache.org/spark-configuration/#sql-extensions!f708e62c6d20960cfe7921152e256ba5) in Spark.

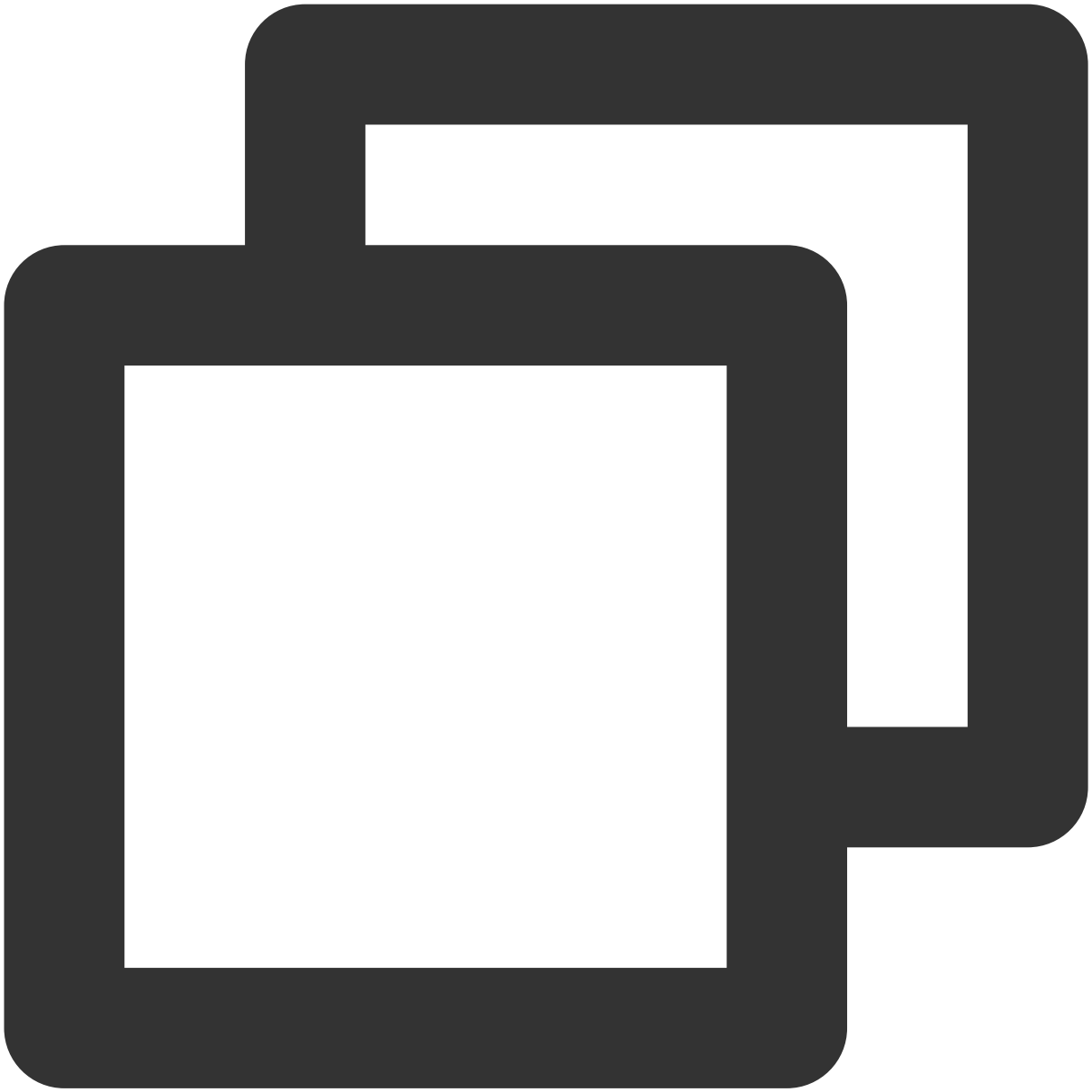
Statement

```
CALL expression
```

Parameter

`expression` : Function expression

Example



```
CALL catalog_name.`system`.procedure_name(arg_name_2 => arg_2, arg_name_1 => arg_1)

#When parameters are passed by position, only the end parameter can be omitted if t
CALL catalog_name.system.procedure_name(arg_1, arg_2, ... arg_n)

#Set the current snapshot to db.sample1:
CALL catalog_name.system.set_current_snapshot('db.sample', 1)
```

More Information

Spark Procedures.

CREATE VIEW AS

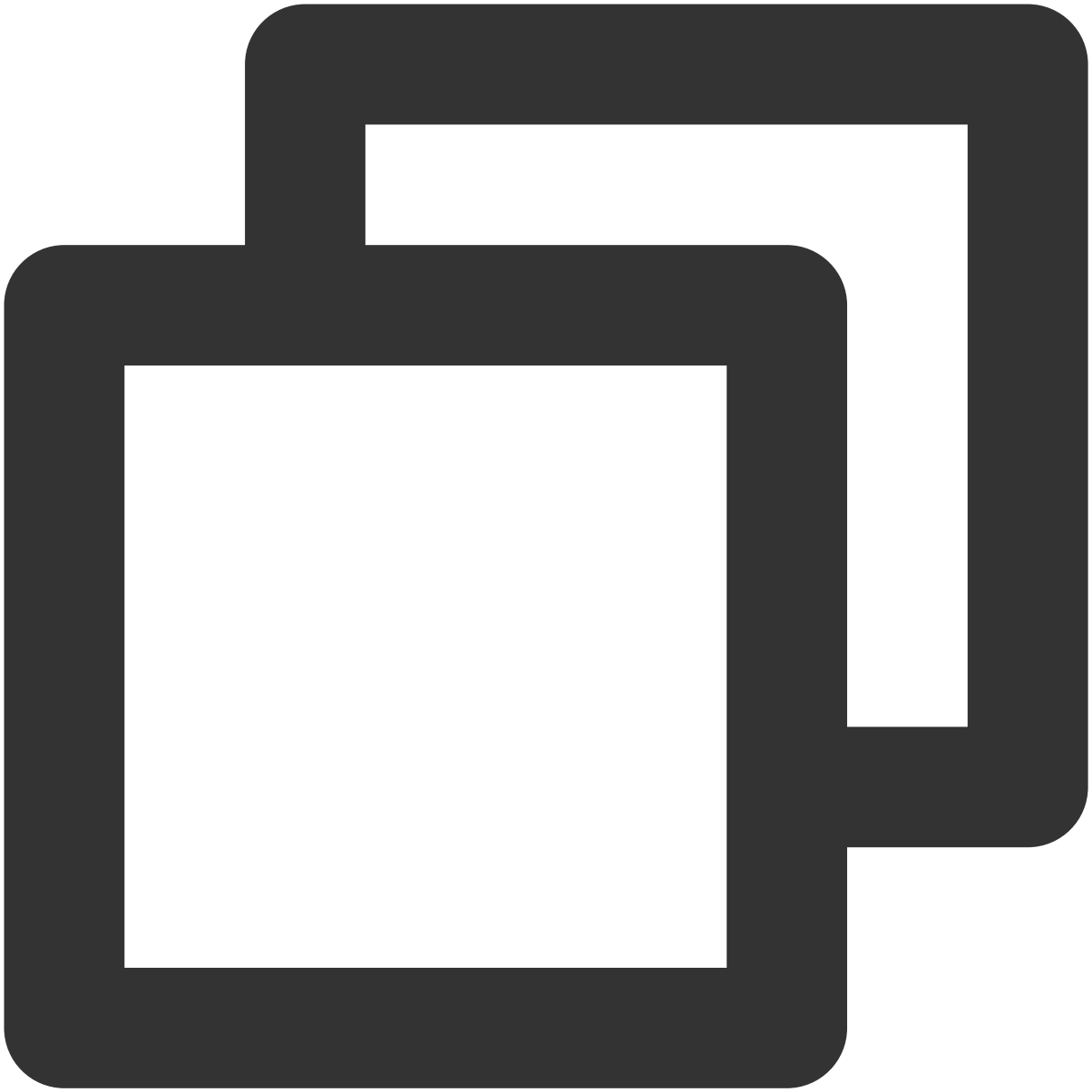
Last updated：2024-08-07 17:22:11

Description

Supported engines: Presto and SparkSQL

Purpose: Create view.

Standard Statement



```
CREATE [ OR REPLACE ] VIEW [IF NOT EXISTS] view_name
    [(column_name [COMMENT 'column_comment'][, ...])]
    [COMMENT 'view_comment']
  AS select_statement
```

Parameter

`[IF NOT EXISTS]` : Create a view if no such view exists.

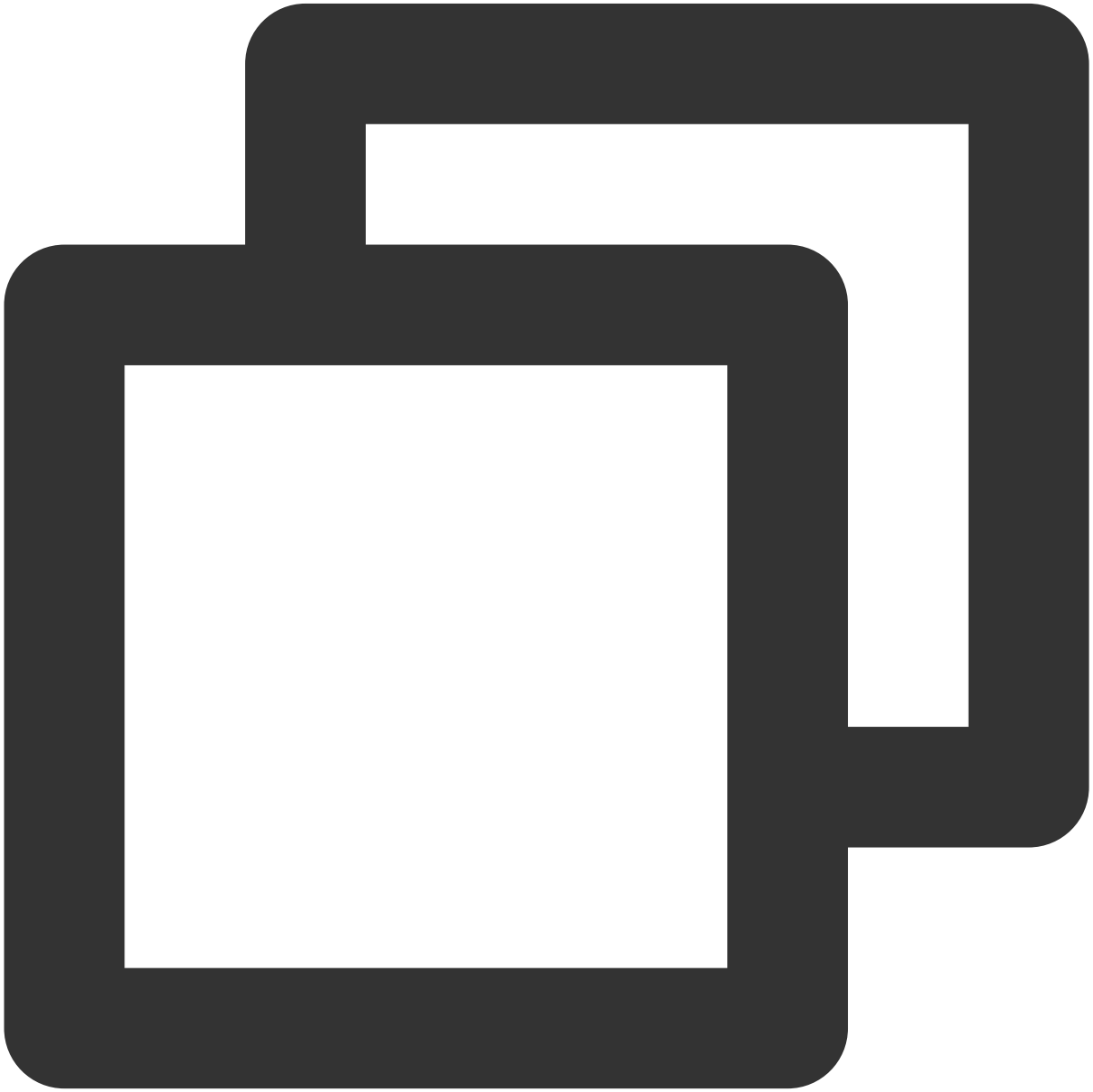
`view_name` : View name

`[(column_name [COMMENT 'column_comment'][, ...])]` : Column name, which can be followed by a column comment.

`[COMMENT 'view_comment']` : View comment

`select_statement` : Query statement.

Example



```
create or replace view db1.v1 as select x,y from tbl;

create view test_view (id comment 'test c1', name_length comment 'test name c2') as
```

SHOW VIEWS

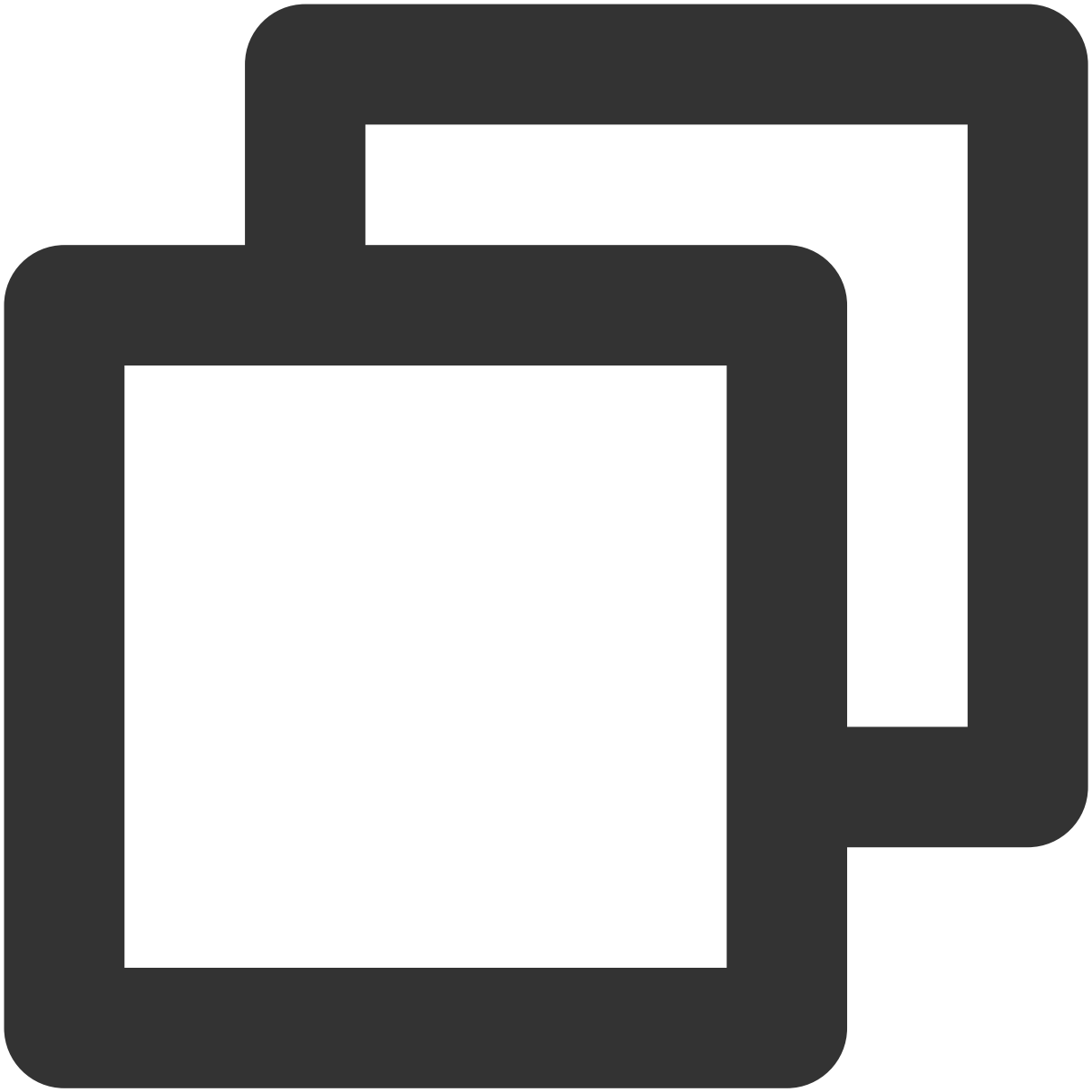
Last updated：2024-08-07 17:22:25

Description

Supported engines: Presto and SparkSQL

Purpose: List views in the specified database. If the database name is not specified, views in the current database are listed.

Statement



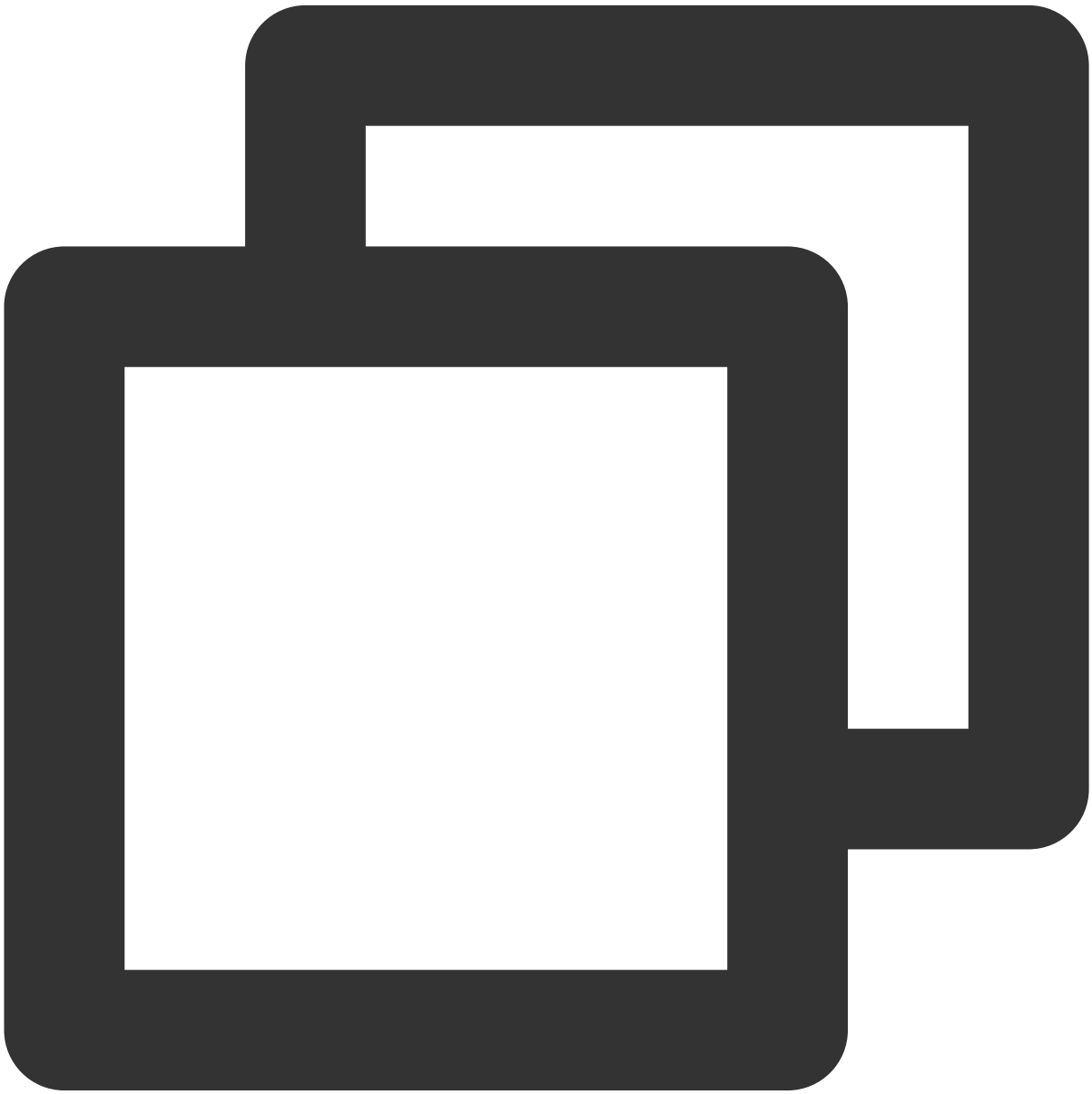
```
SHOW VIEWS [IN database_name] LIKE ['regular_expression']
```

Parameter

`[IN database_name]` : Name of the database whose views are to be listed. If this parameter is left blank, the database in the current context is selected.

`regular_expression` : Regular expression for filtering matching views Only the wildcard character * representing any character or expressions representing spaces or special characters between normal characters can be used.

Example



```
SHOW VIEWS;
```

```
SHOW VIEWS IN db01 LIKE 'view*';
```

DESCRIBE VIEW

Last updated：2024-08-07 17:22:38

Description

Supported engines: Presto and SparkSQL

Purpose: Check view attributes.

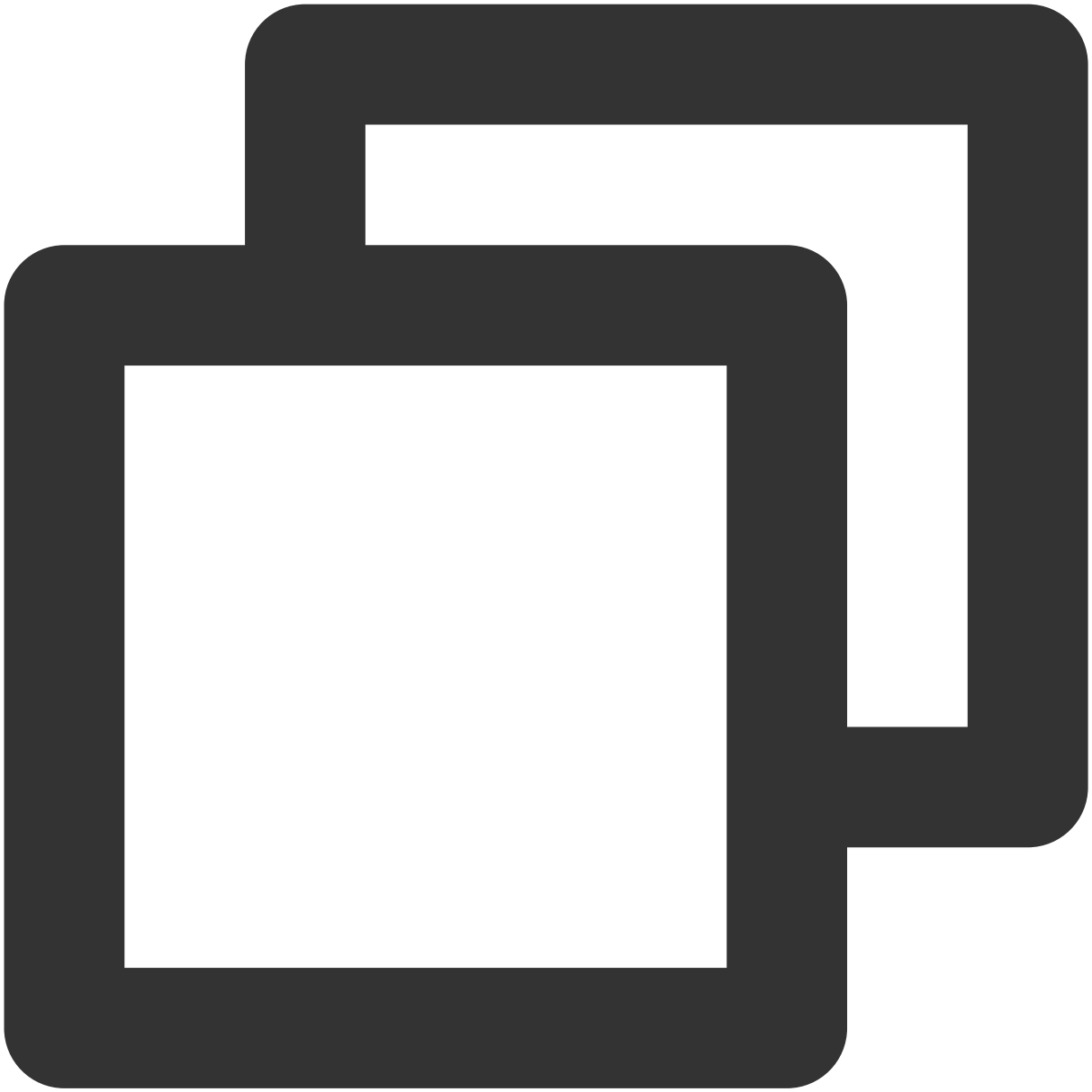
Statement

```
DESCRIBE [view_name]
```

Parameter

view_name: View name

Example



```
DESCRIBE view1;
```

SHOW CREATE VIEW

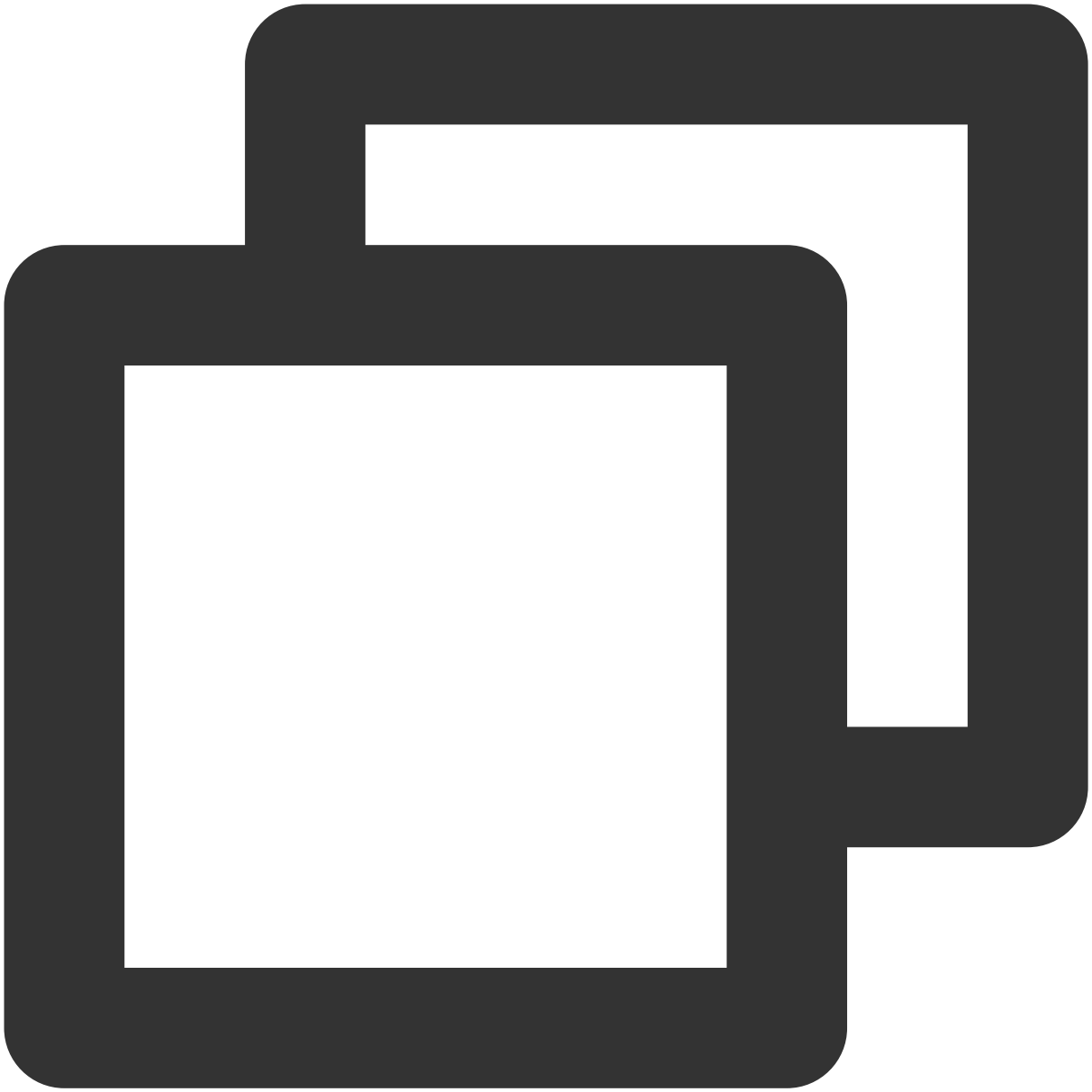
Last updated：2024-08-07 17:22:55

Description

Supported engines: Presto and SparkSQL

Purpose: Display view creation statements.

Statement

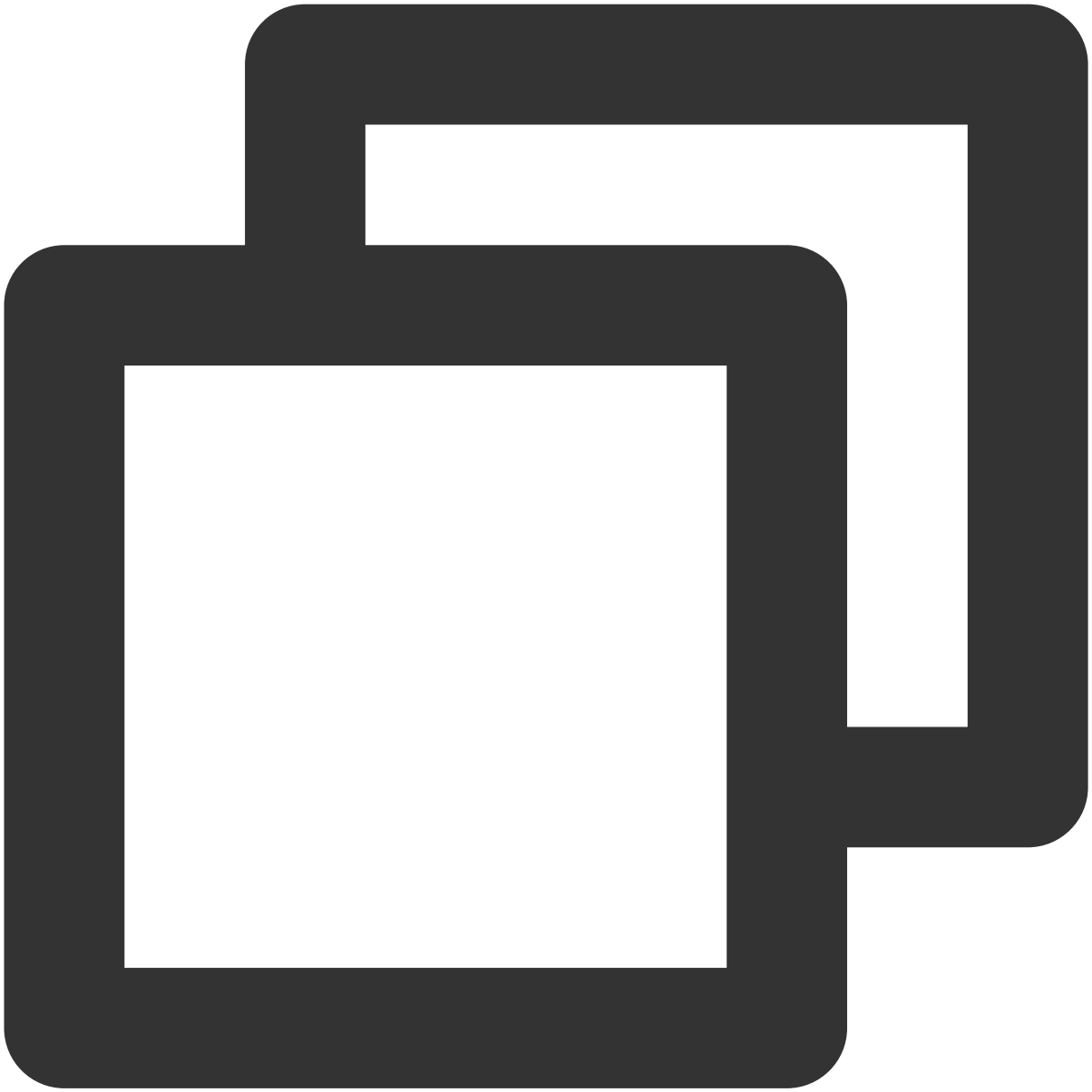


```
SHOW CREATE VIEW view_name
```

Parameter

`view_name` : View name.

Example



```
SHOW CREATE VIEW orders_by_date
```

SHOW COLUMNS IN VIEW

Last updated：2024-08-07 17:23:10

Description

Supported engines: Presto and SparkSQL

Purpose: Check view attributes.

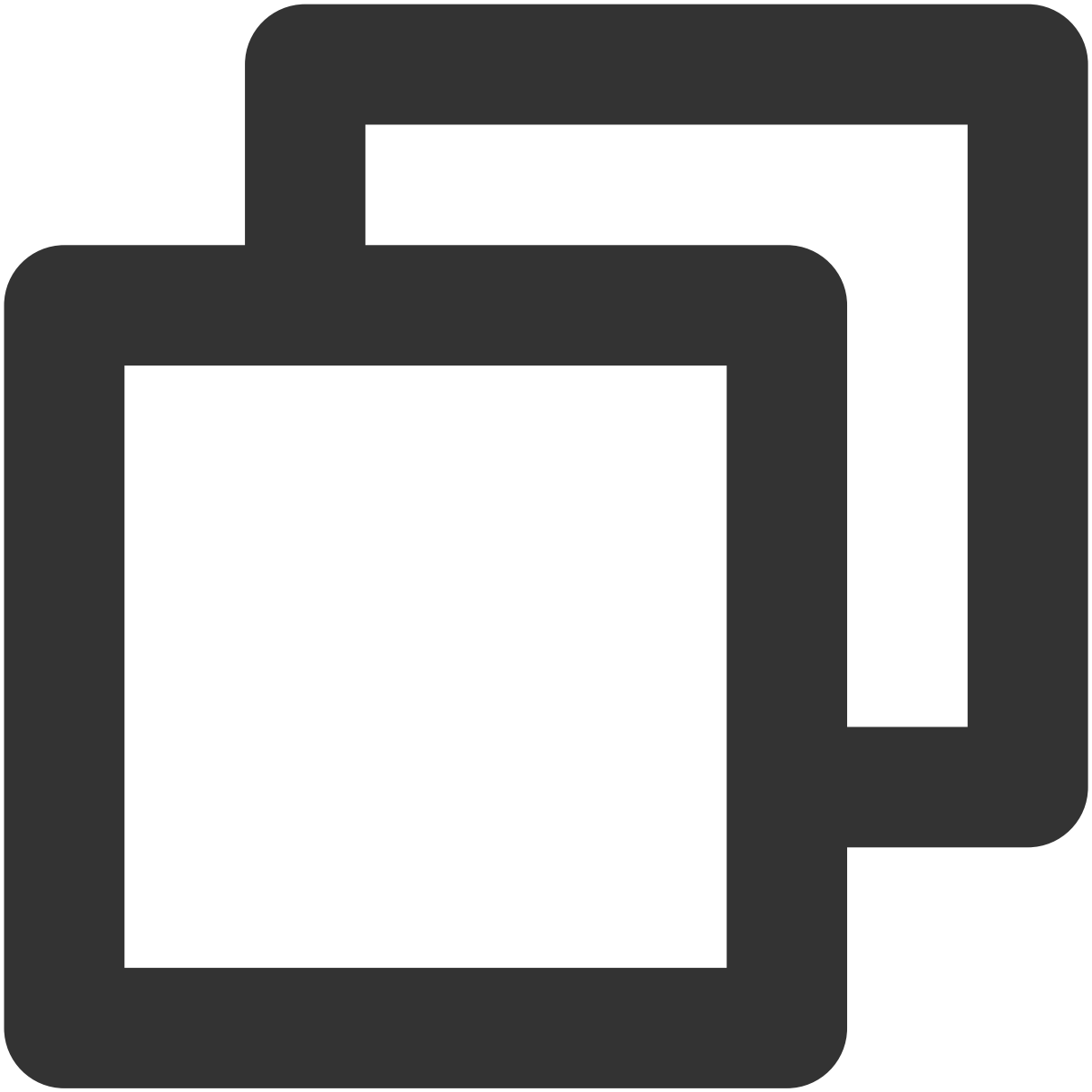
Standard Statement

```
SHOW COLUMNS IN view_name;
```

Parameter

`view_name` : View name.

Example



```
SHOW COLUMNS IN view_test
```

ALTER VIEW
ALTER VIEW RENAME TO

Last updated：2024-08-07 17:23:35

Description

Supported engines: Presto and SparkSQL

Purpose: Change view name.

Standard Statement

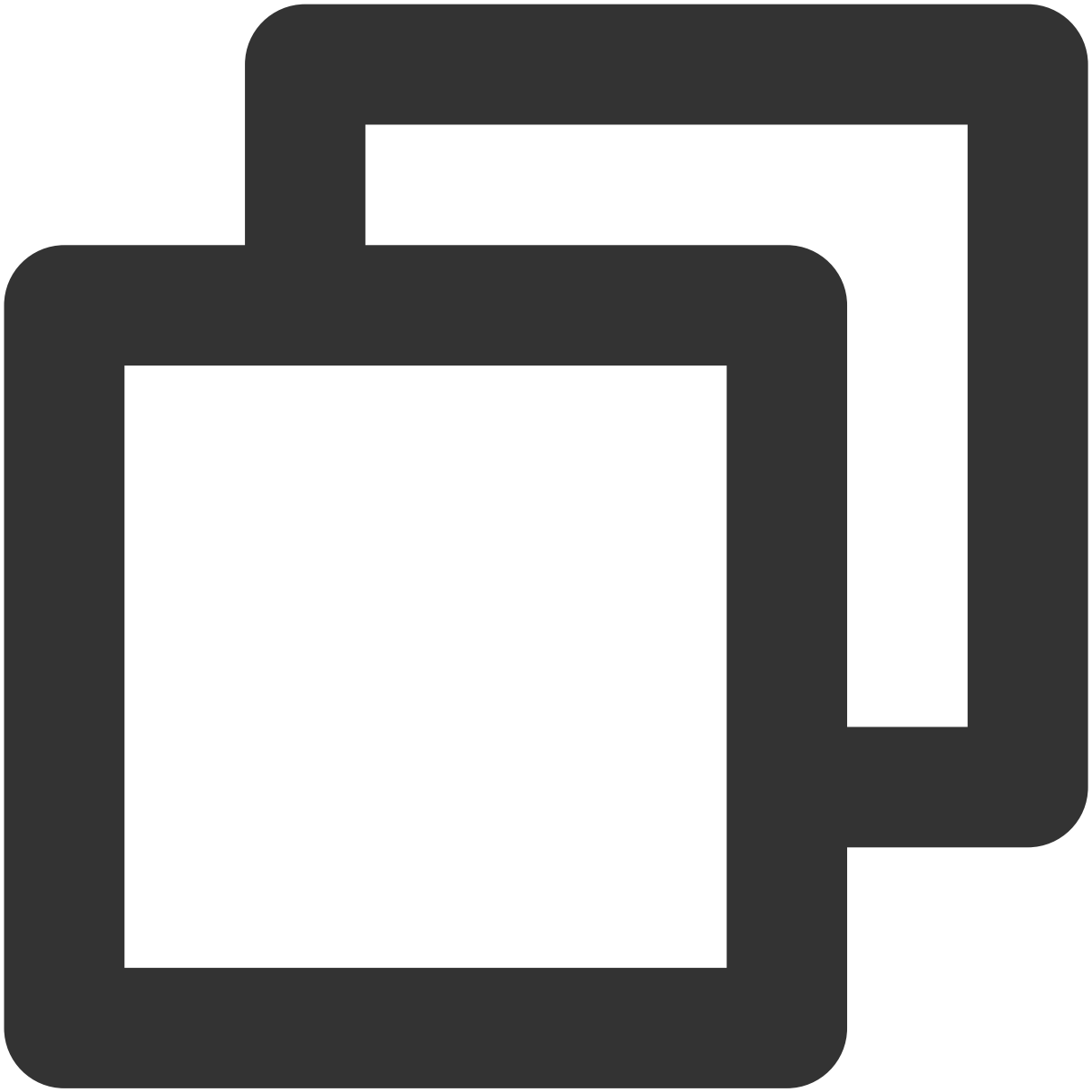
```
ALTER VIEW old_view_identifier RENAME TO new_view_identifier
```

Parameter

`old_view_identifier` : View name before change

`new_view_identifier` : View name after change

Example



```
alter view old_view rename to new_view
```

ALTER VIEW SET TBLPROPERTIES

Last updated：2024-08-07 17:23:59

Description

Supported engines: Presto and SparkSQL

Purpose: Change view attributes.

Standard Statement

```
ALTER VIEW view_identifier SET TBLPROPERTIES ( property_key = property_val [ , ...
```

Parameter

`view_identifier` : Name of the view whose attributes are to be changed.

`property_key` : Attribute name

`property_val` : Attribute value

Example

```
alter view view1 set tblproperties('comment' = 'view1')

alter view view1 set tblproperties('property1' = 'value1', 'property2' = 'value2')
```

DROP VIEW

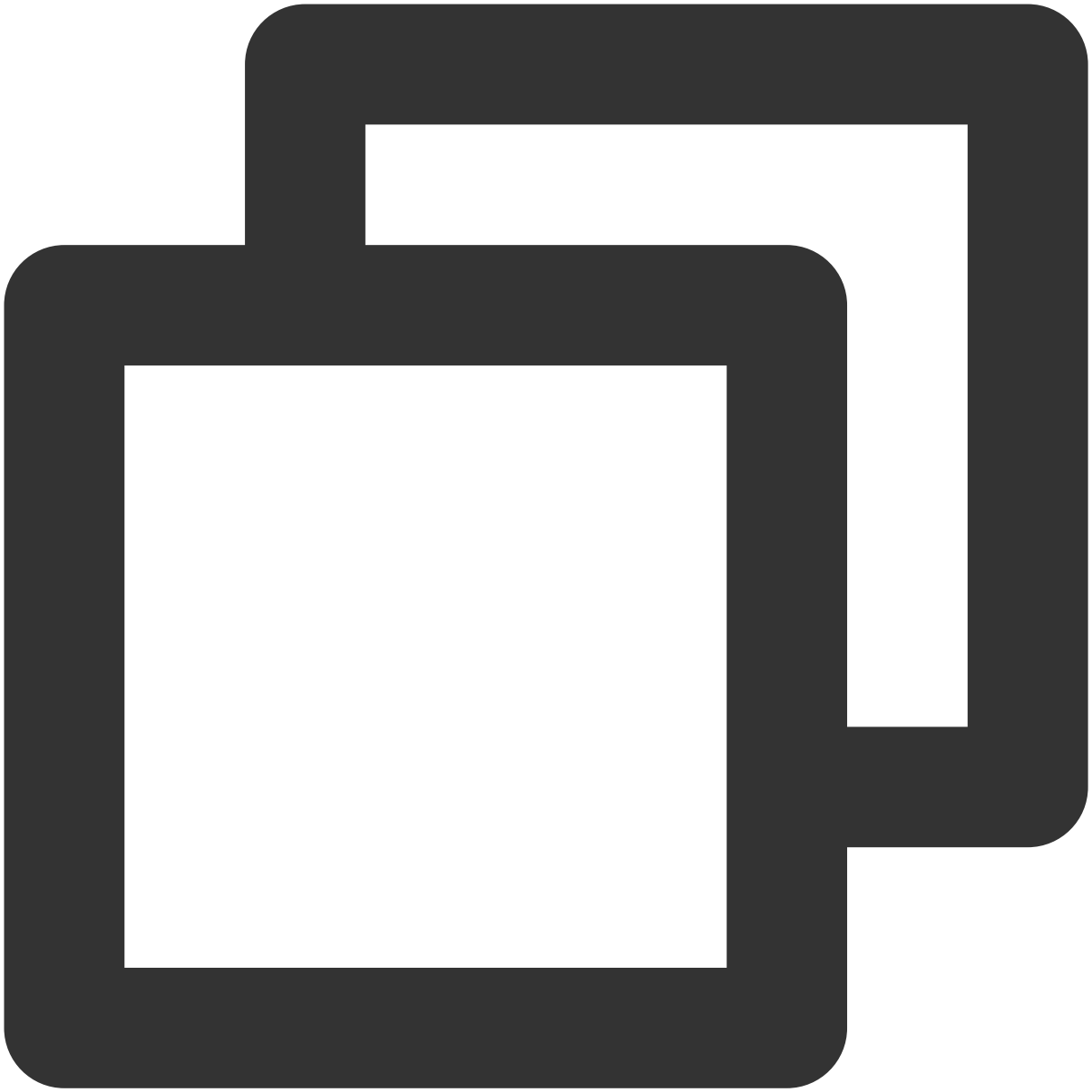
Last updated：2024-08-07 17:24:14

Description

Supported engines: Presto and SparkSQL

Purpose: Delete view.

Statement



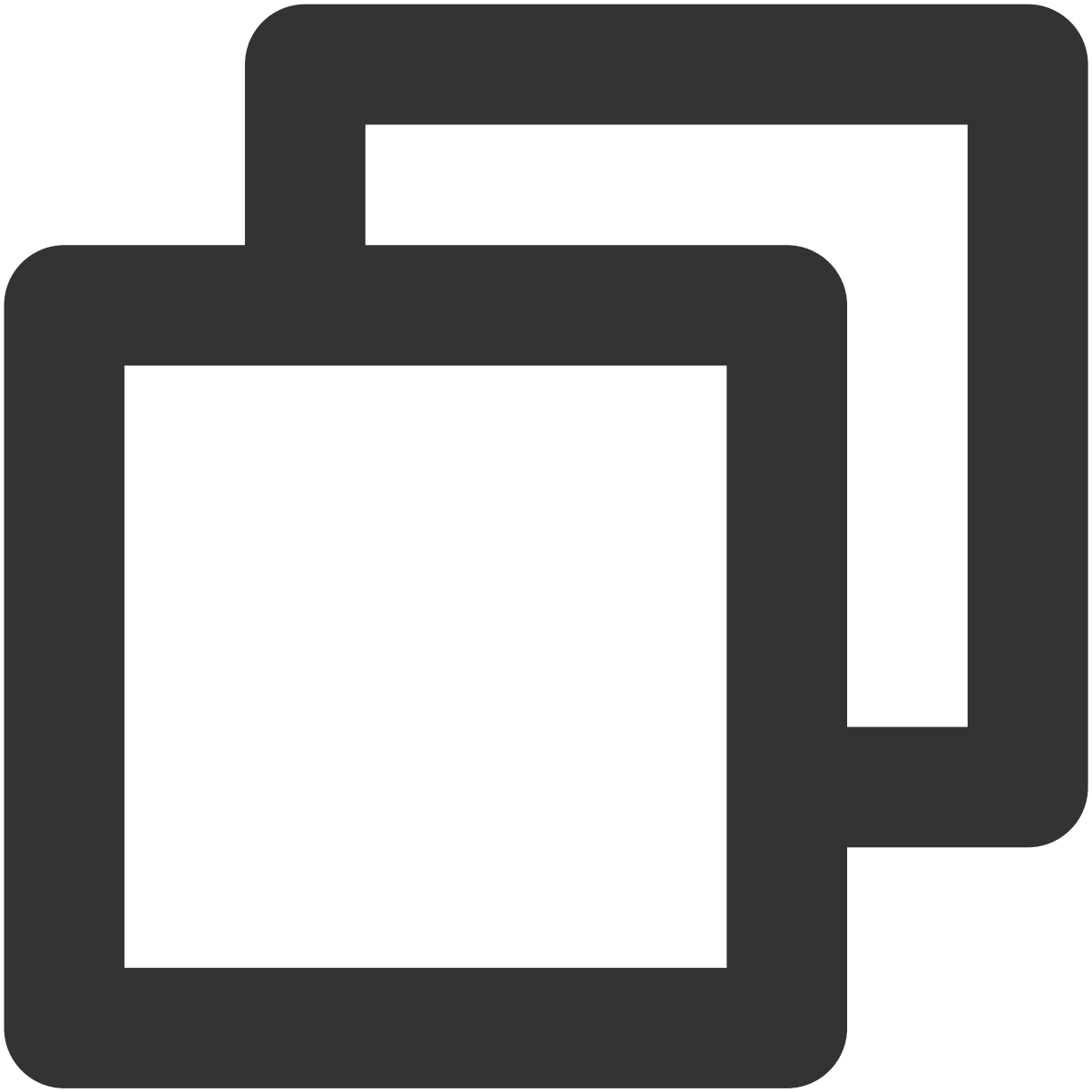
```
DROP VIEW [ IF EXISTS ] view_name;
```

Parameter

`IF EXISTS` : Optional. This parameter indicates whether the view exists.

`view_name` : View name

Example



```
DROP VIEW orders_by_date;
DROP VIEW IF EXISTS orders_by_date;
```

CREATE FUNCTION

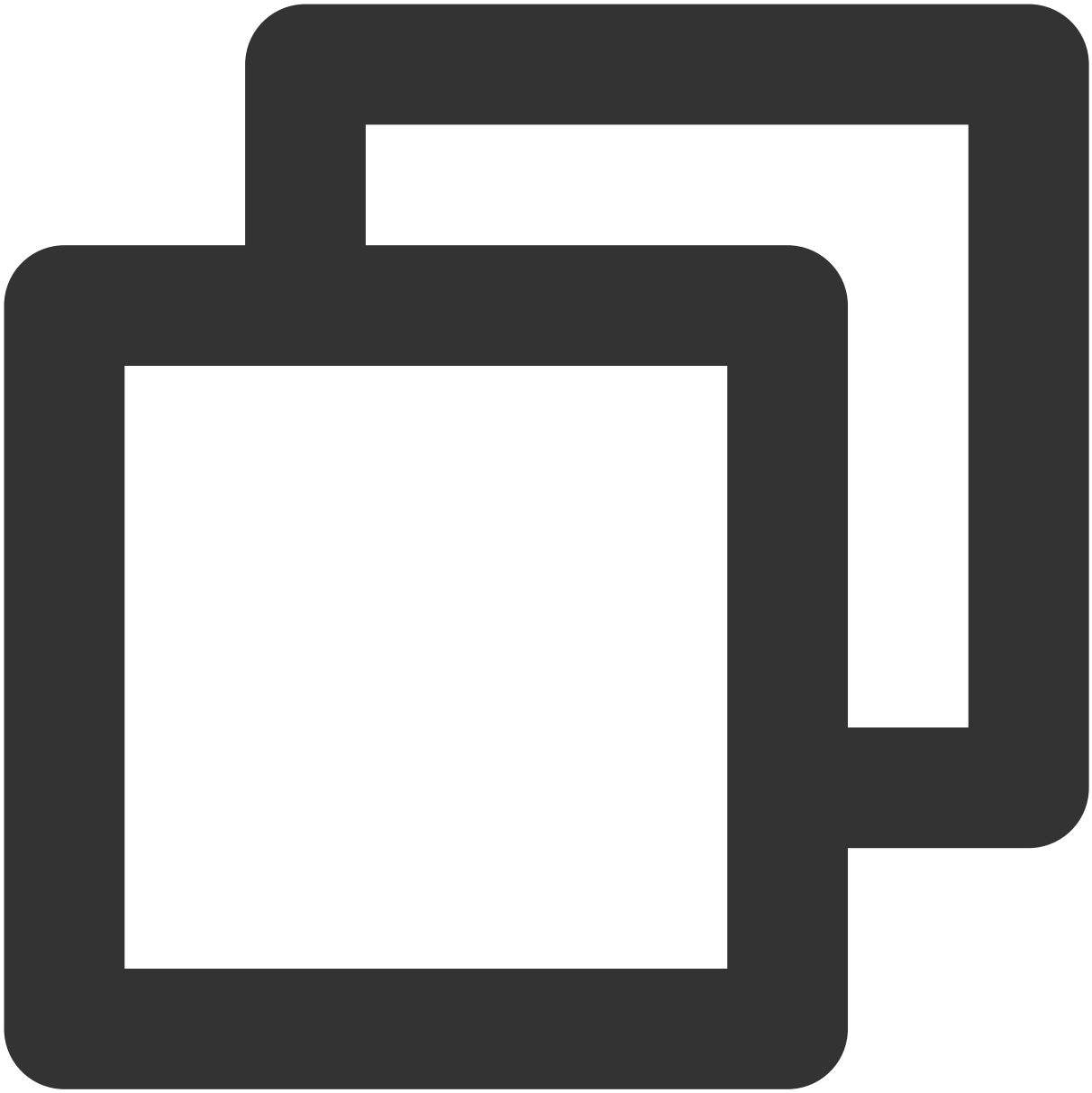
Last updated：2024-08-07 17:24:25

Description

Supported engines: Presto and SparkSQL

Purpose: Create a function that is implemented by the class name.

Statement



```
CREATE  FUNCTION [db_name.]function_name AS class_name
 [USING JAR|FILE|ARCHIVE 'file_uri' [, JAR|FILE|ARCHIVE 'file_uri'] ];

-- A function with the suffix name of _udtf will be recognized as a UDTF function.
CREATE FUNCTION [db_name].function_udtf AS class_name
 [USING JAR|FILE|ARCHIVE 'file_uri' [, JAR|FILE|ARCHIVE 'file_uri'] ];
```

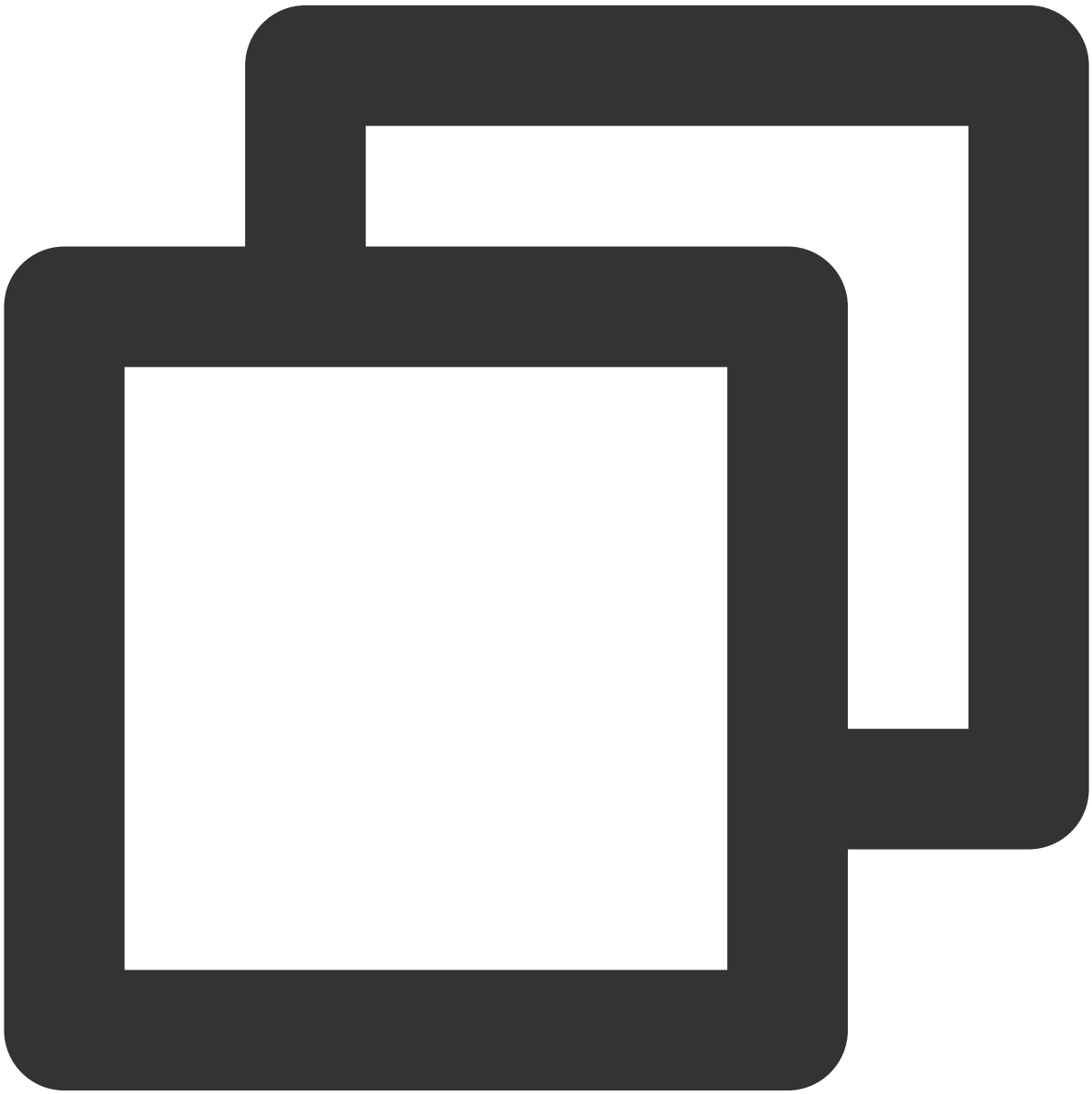
Parameter

`[db_name.]function_name` : Function name. The namespace is specified to be in the database represented by `db_name` during function creation.

`class_name` : Implementation class of the function.

`USING JAR|FILE|ARCHIVE 'file_uri'` : Path to resources used for function creation.

Example



```
CREATE FUNCTION `MYFUNC` AS 'myclass' USING JAR 'hdfs:///path/to/jar'
```

```
CREATE FUNCTION `MYFUNC` AS 'myclass' USING JAR 'hdfs:///path/to/jar', FILE 'file:/
```

SHOW FUNCTION

Last updated：2024-08-07 17:24:44

Description

Supported engines: Presto and SparkSQL

Purpose: Display function creation statements.

Statement

```
SHOW FUNCTIONS [ [ LIKE ] { function_name | regex_pattern }
```

Parameter

`function_name` : Function name

`regex_pattern` : Regular expression for obtaining matching functions

Example

```
SHOW FUNCTIONS

SHOW FUNCTIONS trim;

SHOW FUNCTIONS LIKE 't*'
SHOW FUNCTIONS LIKE 'yea*|windo*'
SHOW FUNCTIONS LIKE 't[a-z][a-z][a-z]'
```

DROP FUNCTION

Last updated：2024-08-07 17:25:03

Description

Supported engines: Presto and SparkSQL

Purpose: Delete custom functions.

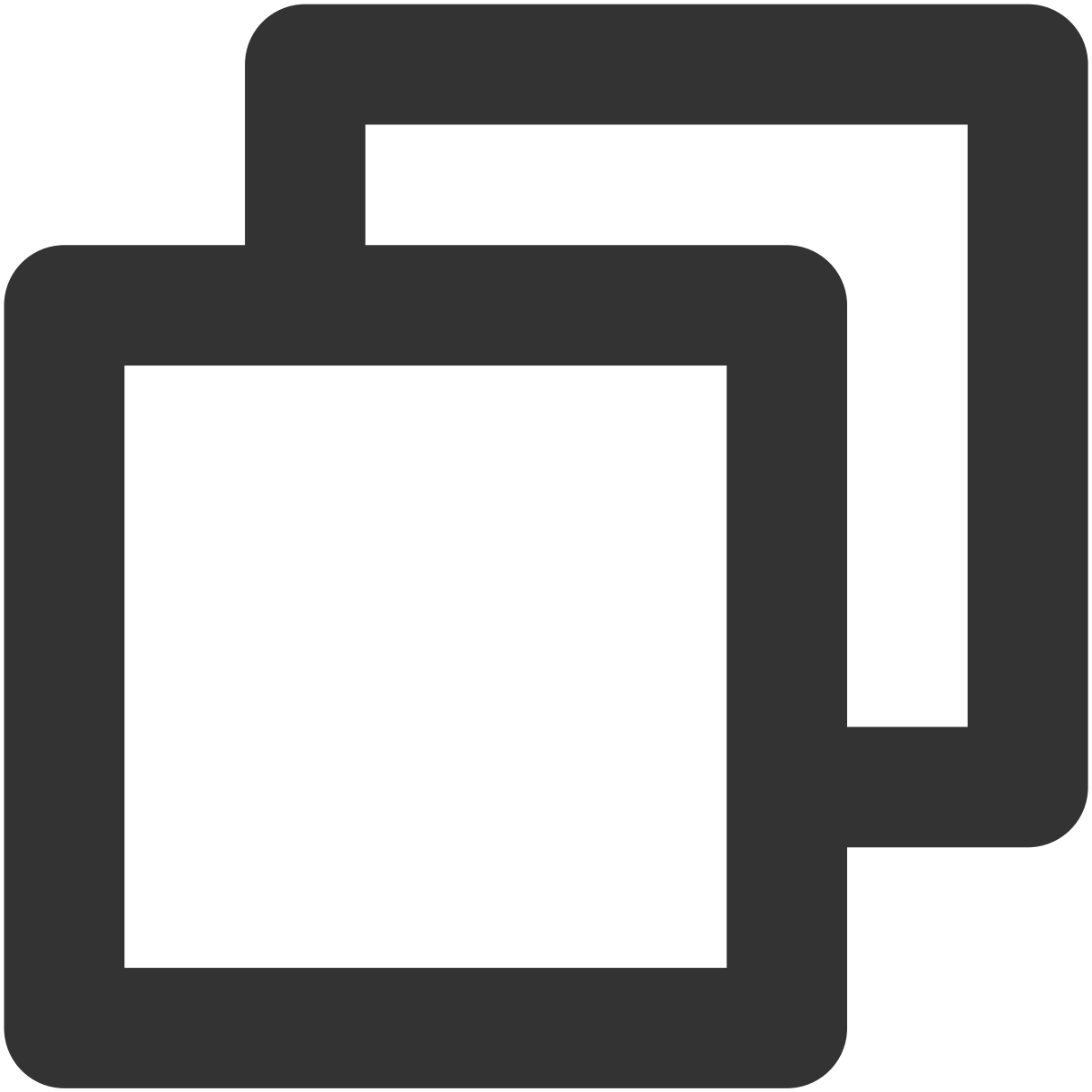
Statement

```
DROP FUNCTION [IF EXISTS] function_name
```

Parameter

`function_name` : Name of the function to be deleted.

Example



```
DROP FUNCTION IF EXISTS `FUNC`

DROP FUNCTION `FUNC
```

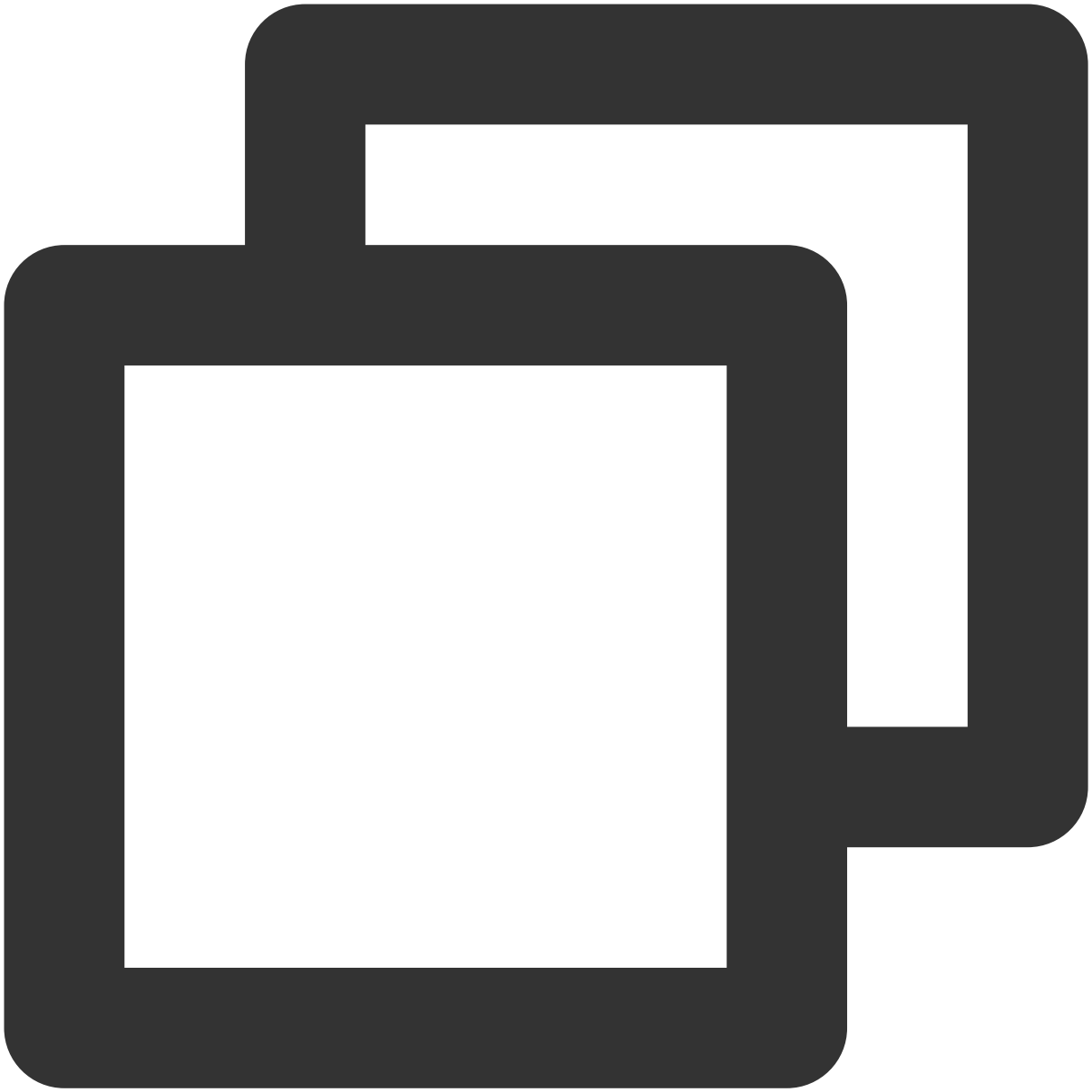
DML Statement
INSERT STATEMENT

Last updated：2024-08-07 17:25:24

Insert a new row into a table. If a column name list is specified, it must exactly match the column name list output by queries. Columns inserted into a table but not in the column name list will be filled with a null value. If no column name list is specified, the columns output by queries must exactly match those in the target table.

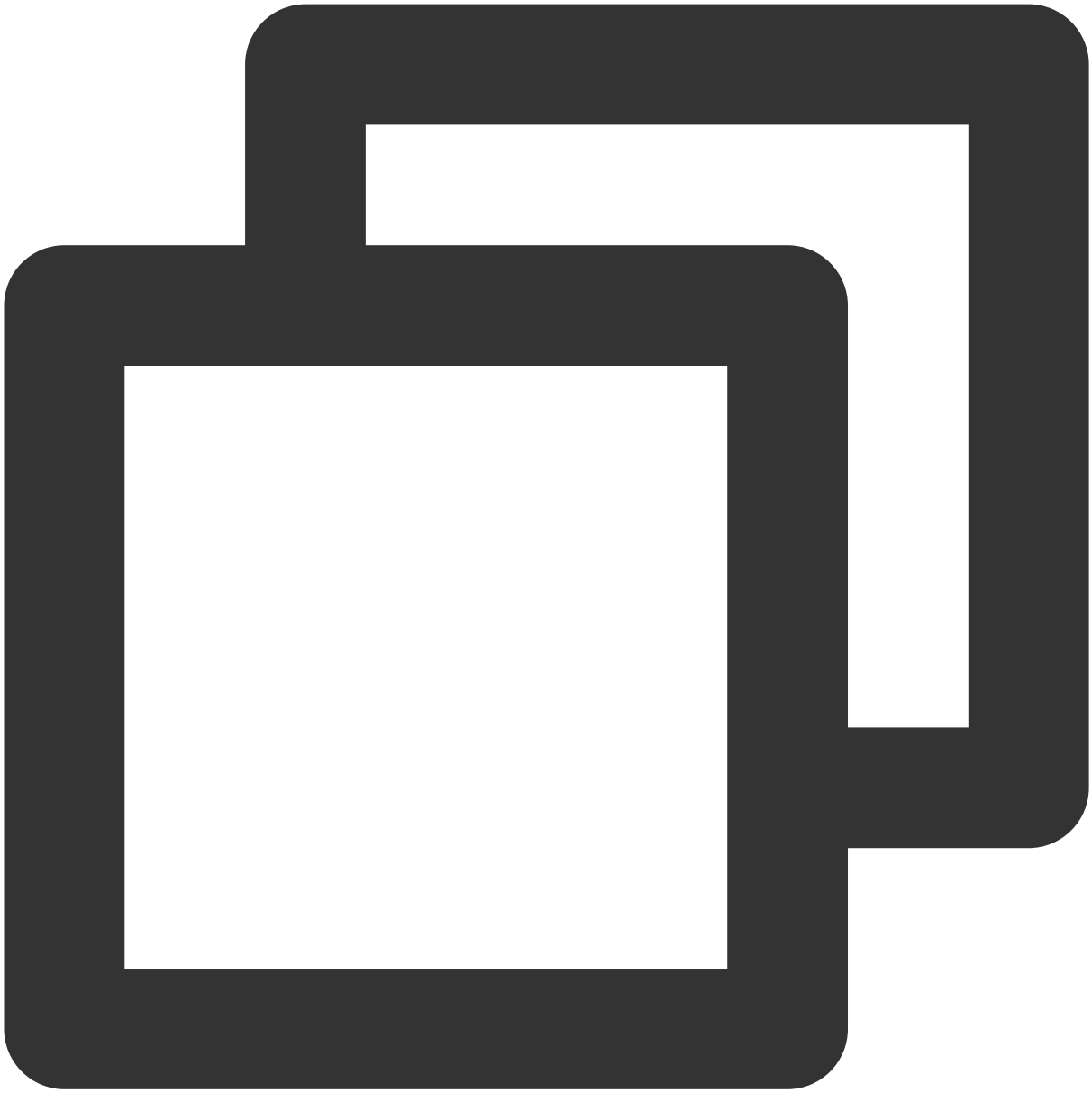
Statement

Presto:



```
INSERT INTO table_name [ ( column [, ... ] ) ] query
```

Spark:



```
INSERT INTO table_identifier [ partition_spec ] [ ( column_list ) ]
  { VALUES ( { value | NULL } [ , ... ] ) [ , ( ... ) ] | query }
```

Parameter

`[ partition_spec ]` : Partitioning column name and value Example: dt='2021-06-01'

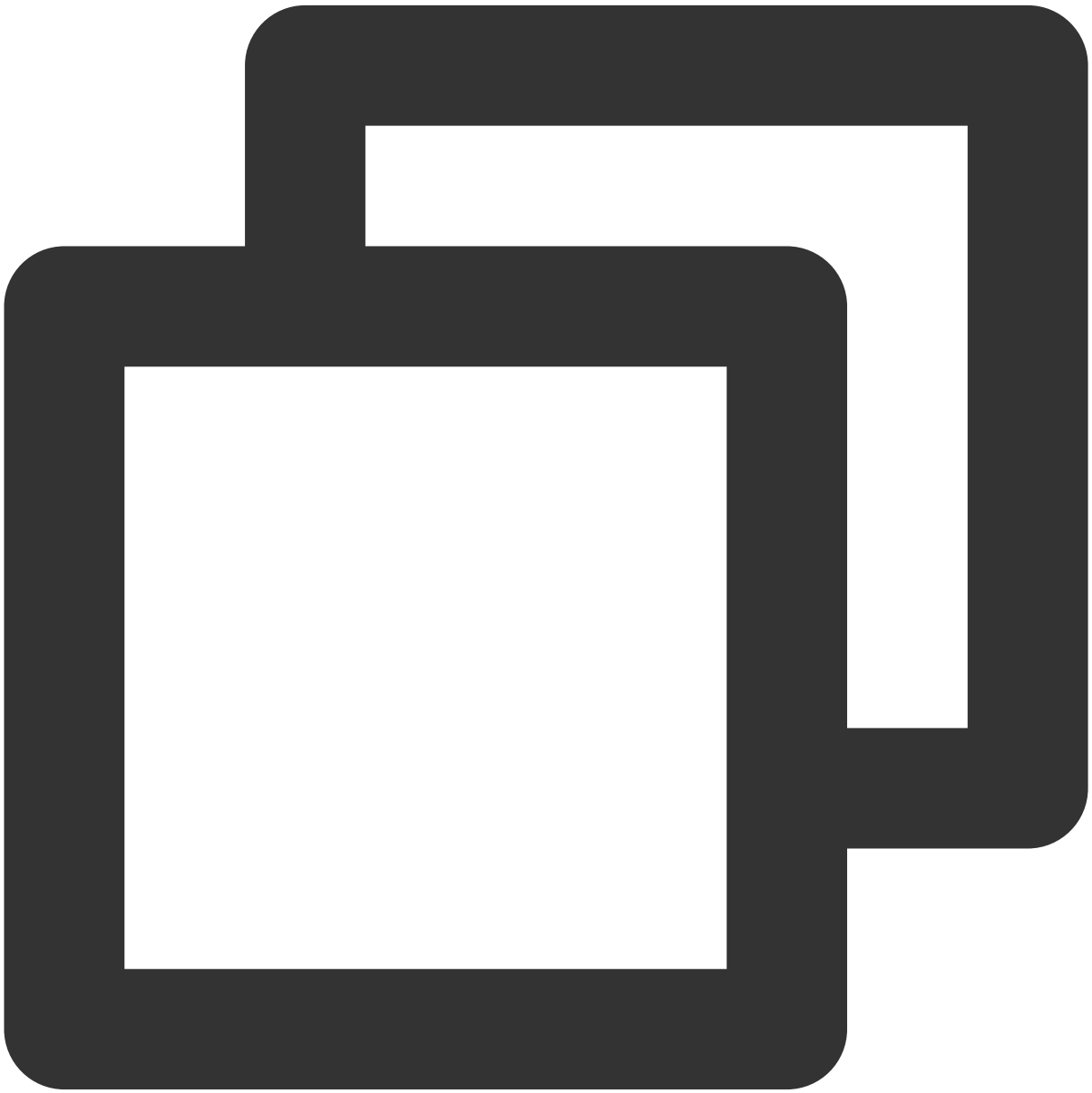
`[ ( column [, ... ] ) ]` ： All the columns

`[table_name] | table_identifier` : Table name

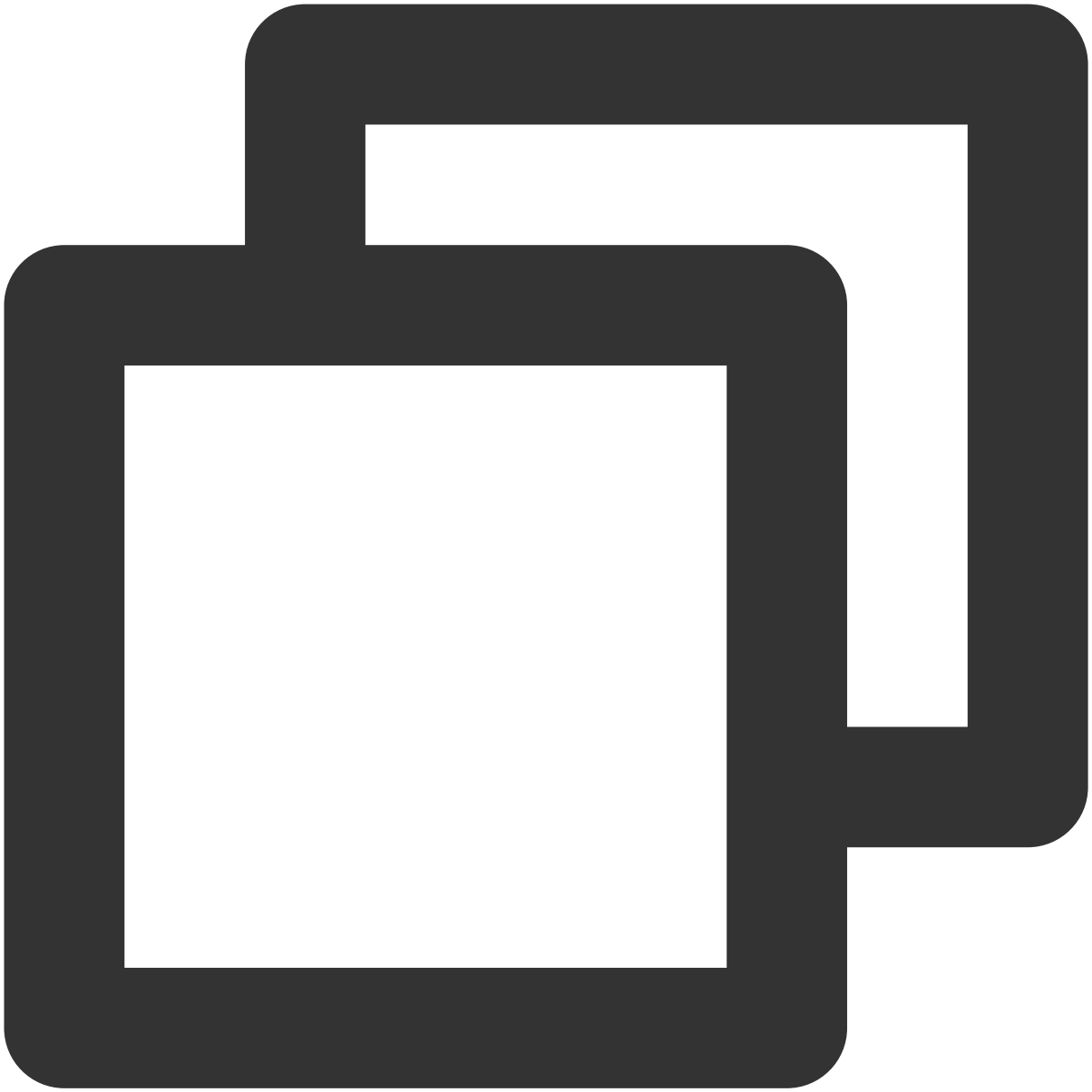
`[query]` : A general Select query statement

Example

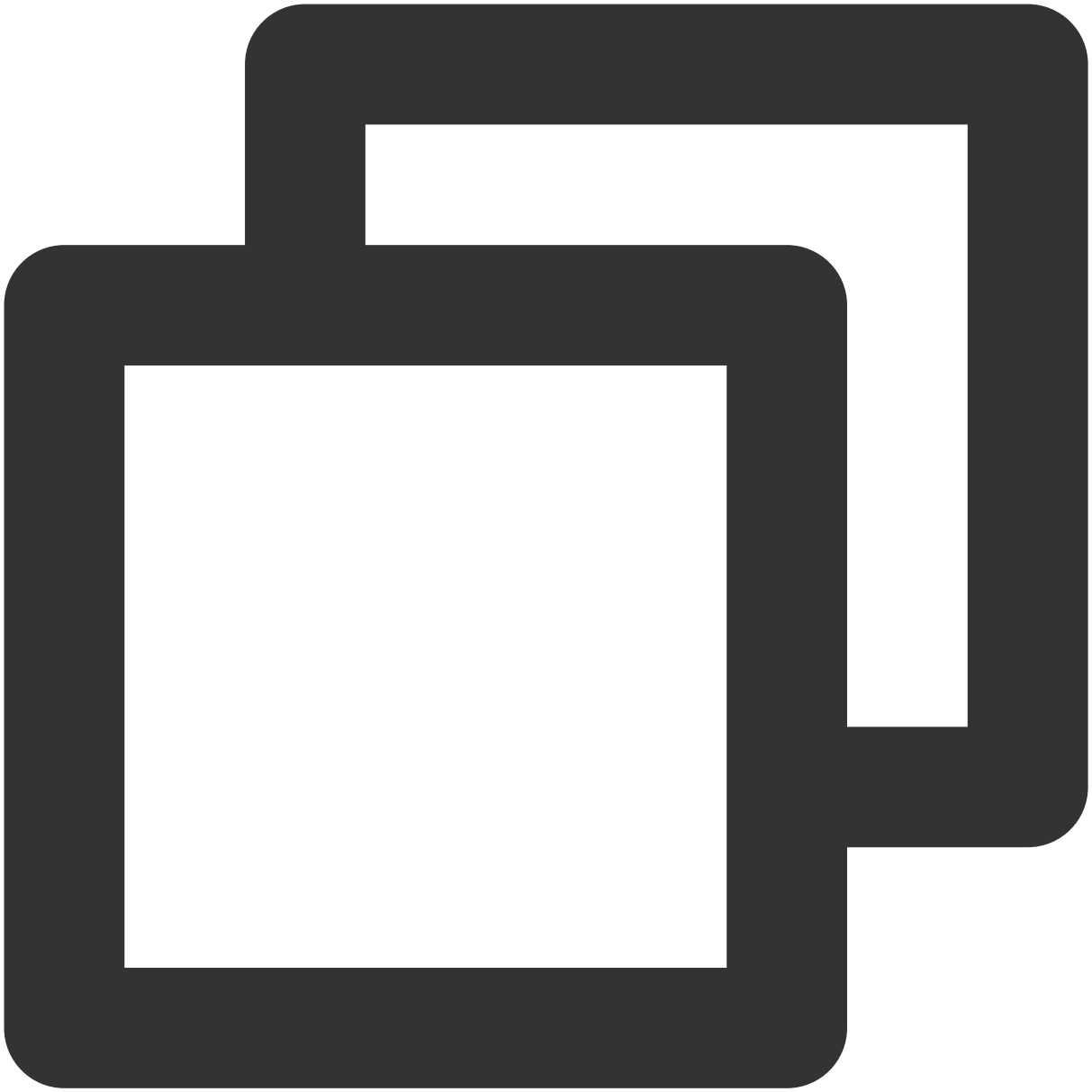
Example for Presto and Spark:



```
INSERT INTO orders SELECT * FROM new_orders;
```



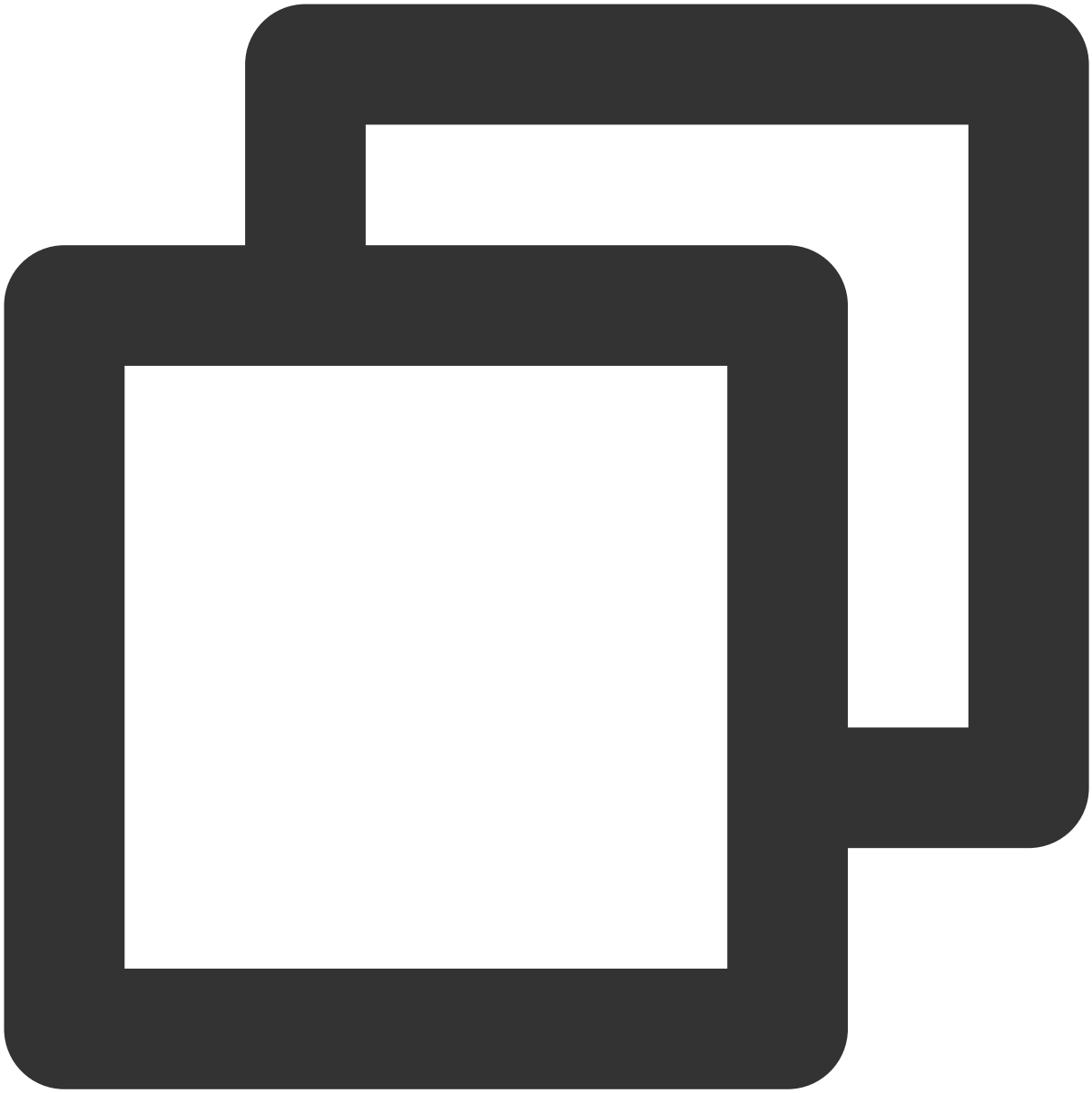
```
INSERT INTO cities VALUES (1, 'China');
```



```
INSERT INTO nation (nationkey, name, regionkey, comment)
VALUES (26, 'POLAND', 3, 'no comment');
```

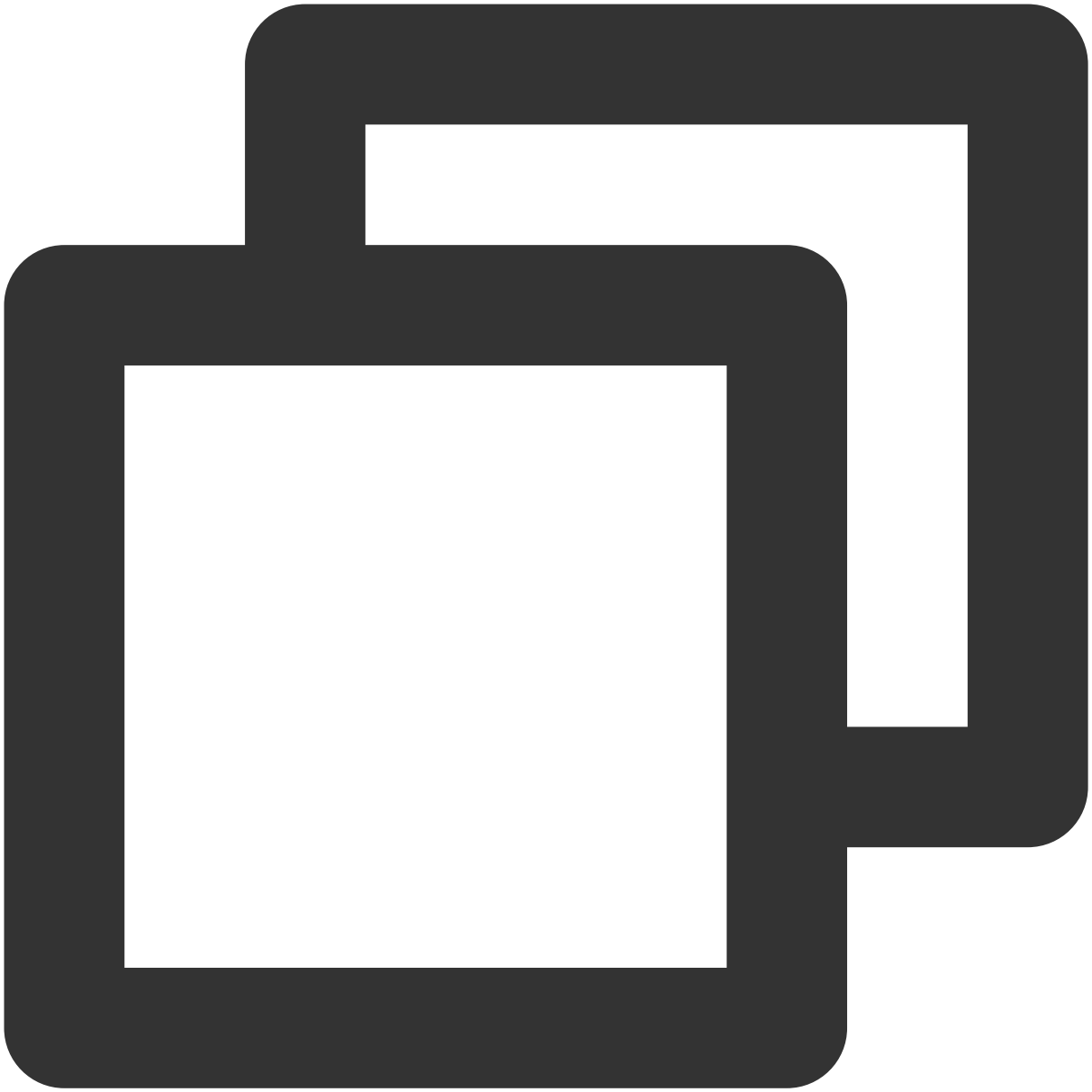
Example for Spark:

Use a Select query statement for inserting a column into a partition:



```
INSERT INTO students PARTITION (student_id = 444444) SELECT name, address FROM pers
```

Insert a column into a partition:



```
INSERT INTO students PARTITION (student_id = 11215017) (address, name) VALUES  ('Sh
```

Limitations

Presto does not support inserting columns into partitions. You can use Spark instead.

INSERT INTO

Last updated：2024-08-07 17:25:30

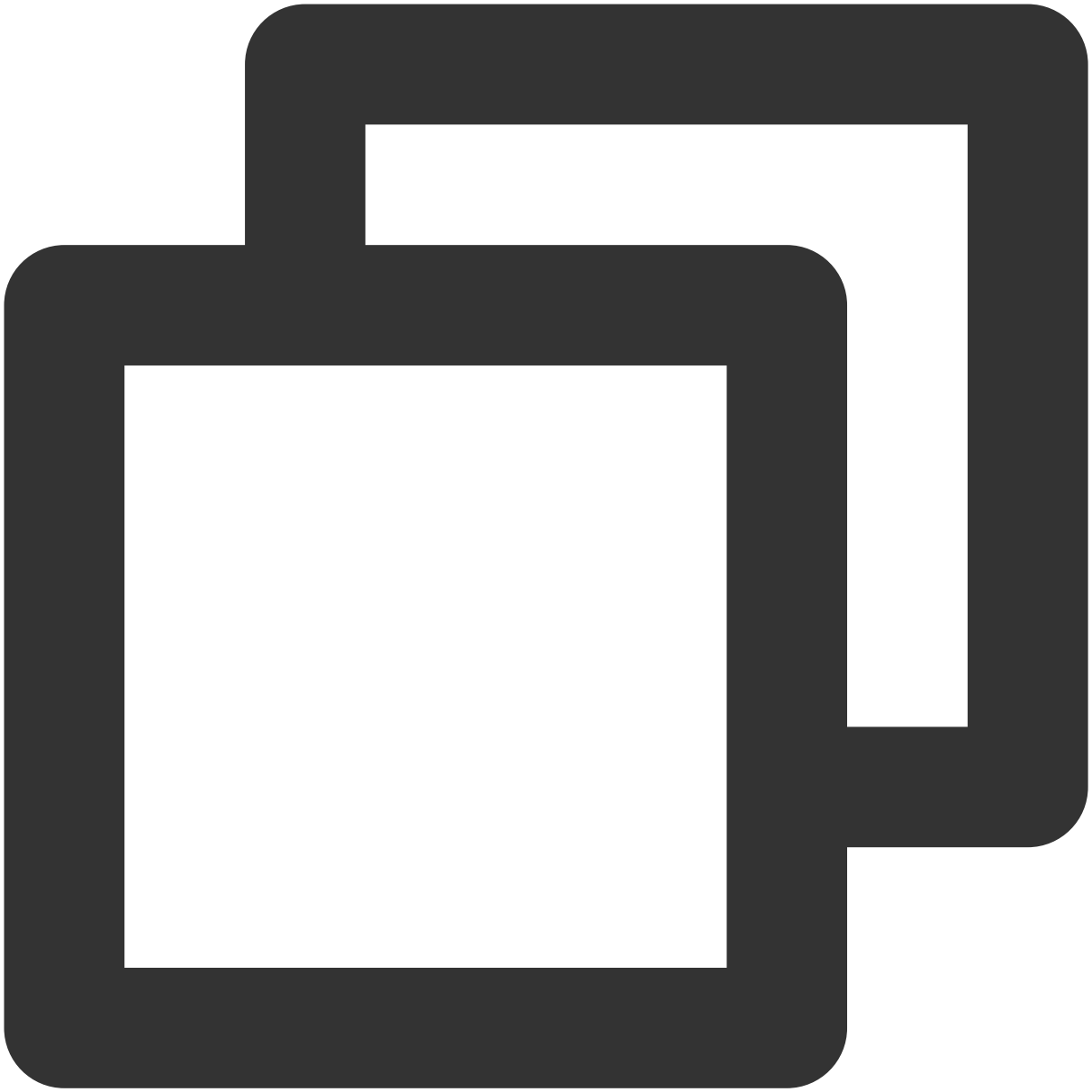
Description

Supported engines: Presto and SparkSQL

Applicable table: Native Iceberg tables and external tables

Purpose: Insert the results of a SELECT query on the source table into the target table as new rows.

Statement



```
[ WITH with_query [ , ... ] ]
INSERT {INTO [<TABLE>]| TABLE} table_identifier [ partition_spec ] [ ( column_list
    { VALUES ( { value | NULL } [ , ... ] ) [ , ( ... ) ] | query }
```

Parameter

`table_identifier` : Table name in the three-part format. Example: `catalog.database.table` .
`partition_spec` : Partitioning column name and value Example: dt='2021-06-01'

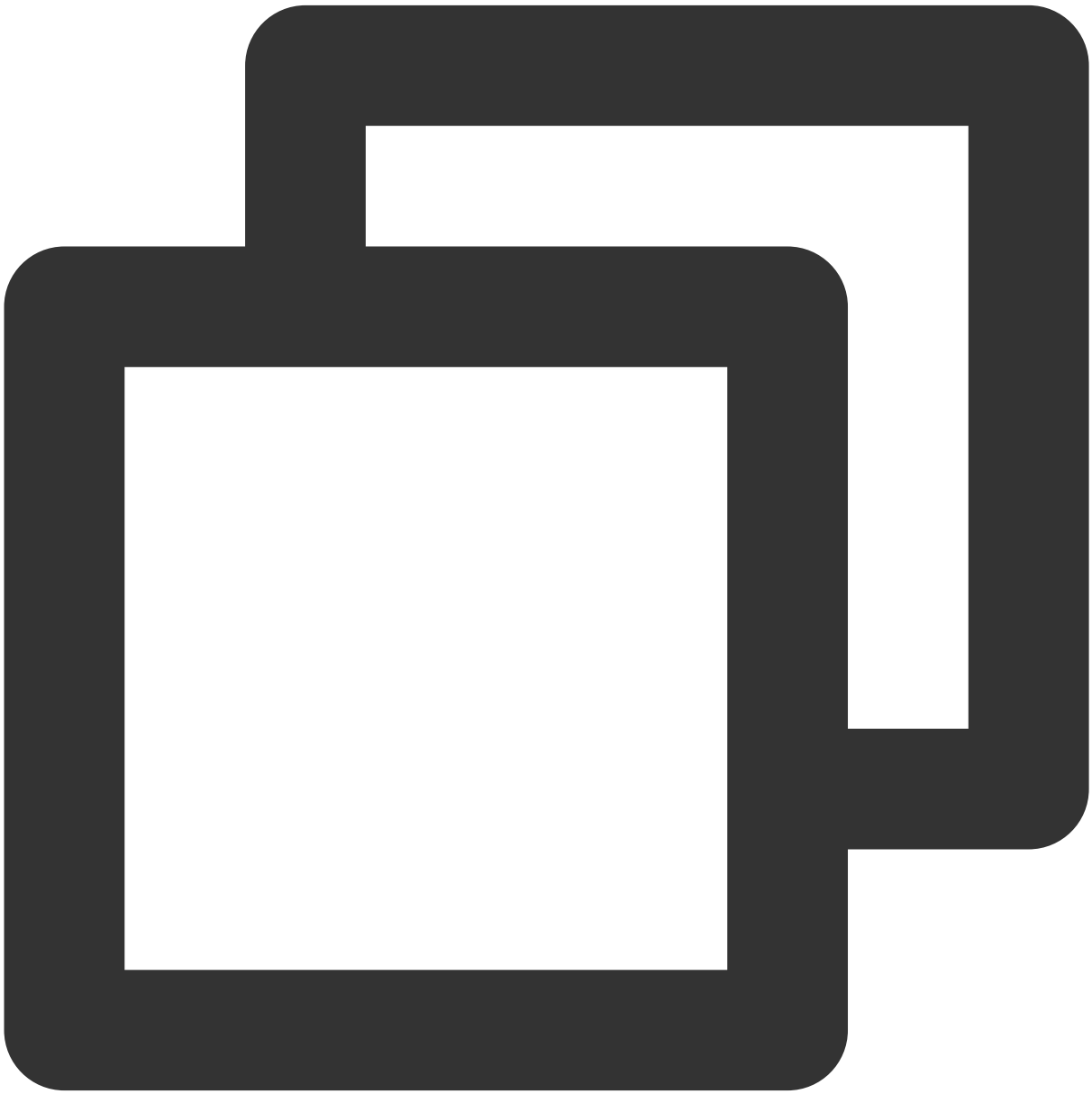
`column_list` : All the columns

`query` : A general Select query statement

1.1 `a SELECT statement`

1.2 `a TABLE statement`

Example



```
INSERT INTO orders SELECT * FROM new_orders;
```

```
INSERT INTO cities VALUES (1, 'China');
INSERT INTO nation (nationkey, name, regionkey, comment)
VALUES (26, 'POLAND', 3, 'no comment');

-- INSERT INTO partition
INSERT INTO students PARTITION (student_id = 444444) SELECT name, address FROM pers
INSERT INTO students PARTITION (student_id = 11215017) (address, name) VALUES  ('Sh

-- Insert Using a TABLE Statement
INSERT INTO students TABLE visiting_students;

-- with
WITH `tmp1` AS ((SELECT *
FROM `catalog1`.`db1`.`tbl1`)), `tmp2` AS ((SELECT *
FROM `tbl2`))
INSERT INTO `catalog1`.`db2`.`tbl1`
(SELECT `col1`, `col2`
FROM `tmp1` `a`
INNER JOIN `tmp2` `b` ON `a`.`col1` = `b`.`col2`)


INSERT INTO `catalog1`.`db2`.`tbl1`
WITH `tmp1` AS ((SELECT *
FROM `catalog1`.`db1`.`tbl1`)), `tmp2` AS ((SELECT *
FROM `tbl2`))
(SELECT `col1`, `col2`
FROM `tmp1` `a`
INNER JOIN `tmp2` `b` ON `a`.`col1` = `b`.`col2`)
```

INSERT OVERWRITE

Last updated：2024-08-07 17:25:36

Description

Supported engines: Presto and SparkSQL

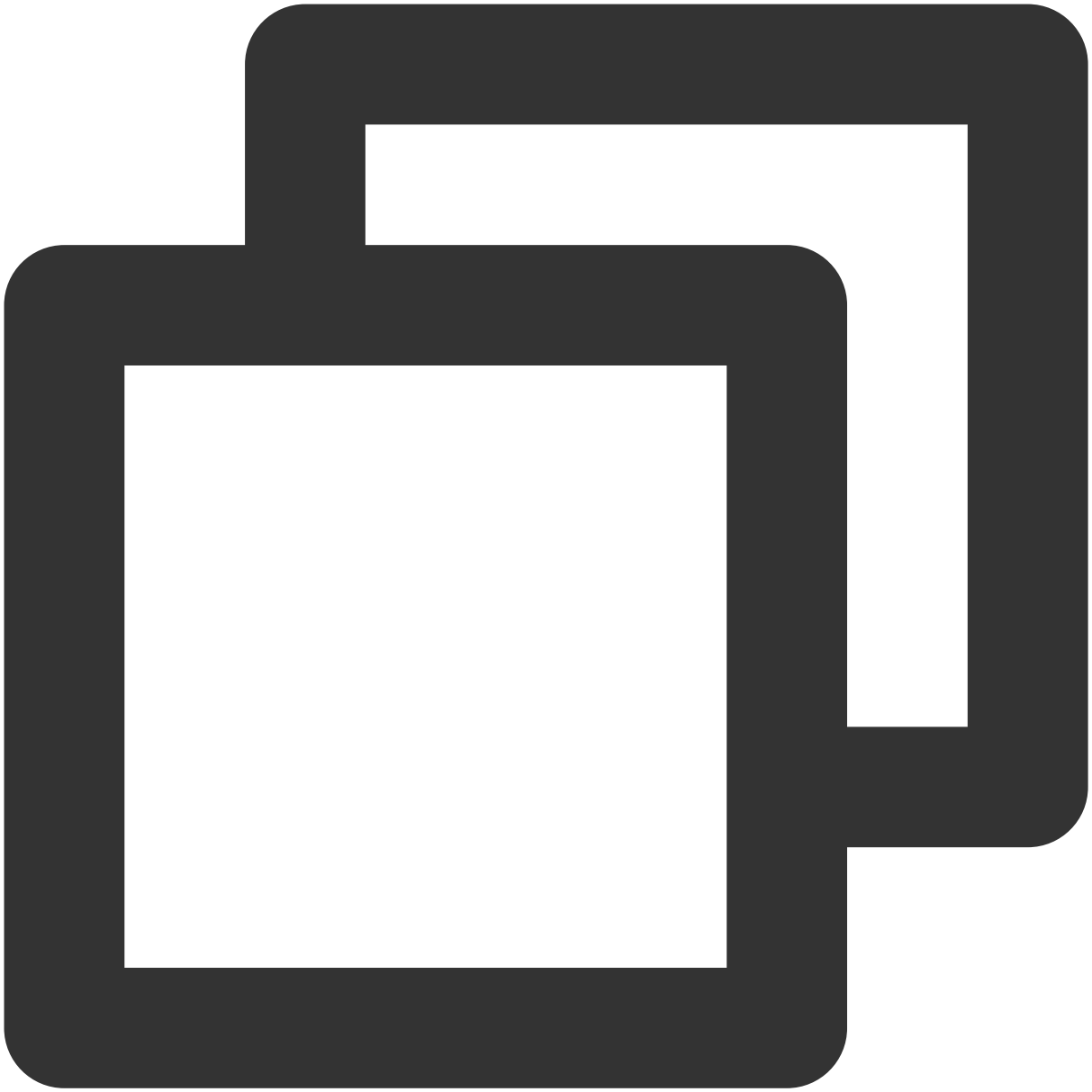
Applicable table: Native Iceberg tables and external tables

Purpose: Insert data into the row.

Note

Presto supports INSERT OVERWRITE only on partitioned tables of Hive data sources but not on non-partitioned tables and tables of Iceberg data sources.

Statement



```
INSERT OVERWRITE table_identifier [ partition_spec ] [ ( column_list ) ]
    { VALUES ( { value | NULL } [ , ... ] ) [ , ( ... ) ] | query
```

Parameter

`table_identifier` : Table name in the three-part format. Example: catalog.database.table.

`partition_spec` : Partitioning column name and value Example: dt='2021-06-01'

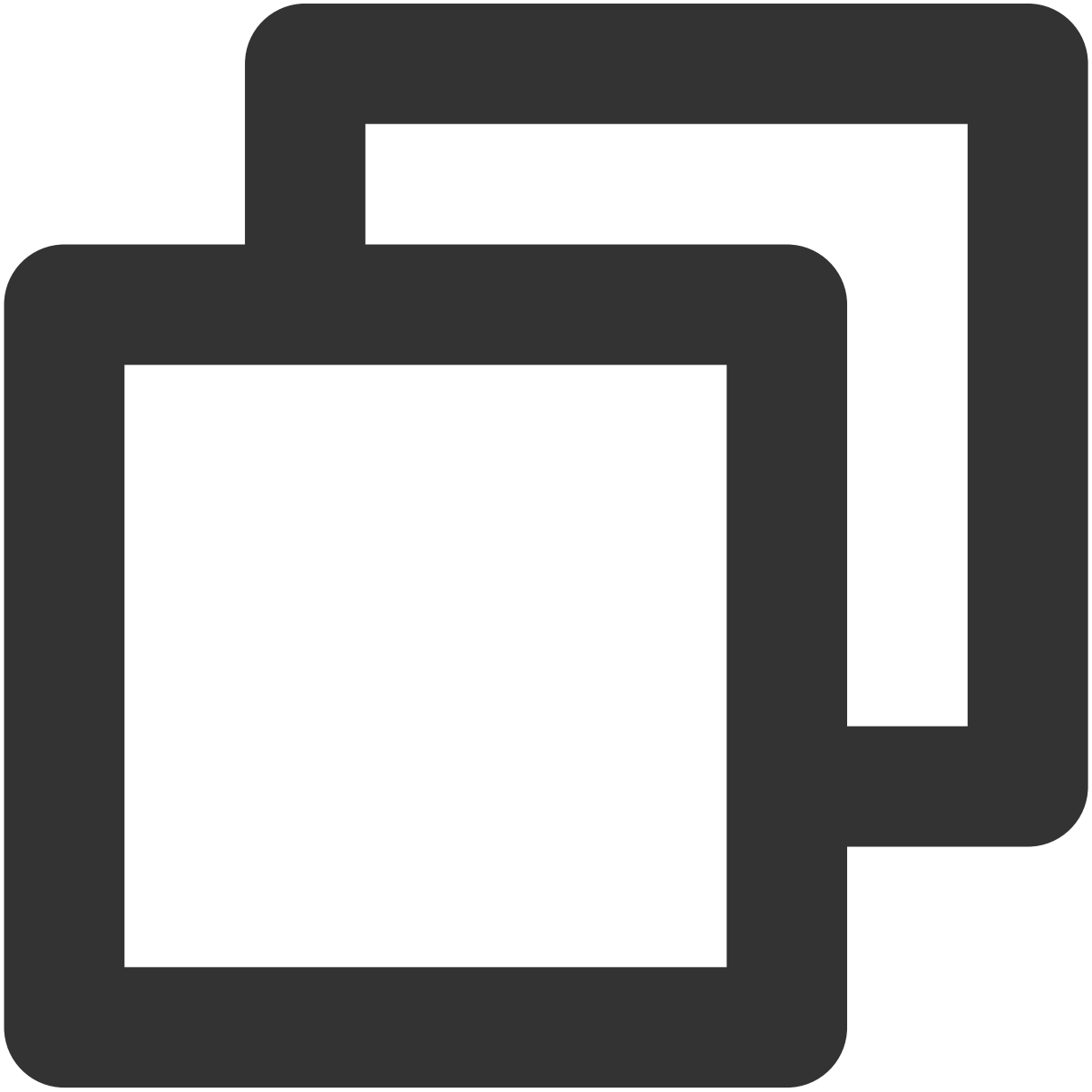
`column_list` : All the columns

`query` : A general Select query statement.

1.1 a SELECT statement

1.2 a TABLE statement

Example



```
-- Insert Using a VALUES Clause
```

```
INSERT OVERWRITE students VALUES
    ('Ashua Hill', '456 Erica Ct, Cupertino', 111111),
    ('Brian Reed', '723 Kern Ave, Palo Alto', 222222);

-- Insert Using a SELECT Statement
INSERT OVERWRITE students PARTITION (student_id = 222222)
    SELECT name, address FROM persons WHERE name = "Dora Williams"

-- Insert Using a TABLE Statement
INSERT OVERWRITE students TABLE visiting_students

-- Insert with a column list
INSERT OVERWRITE students (address, name, student_id) VALUES
    ('Hangzhou, China', 'Kent Yao', 11215016)

-- Insert with both a partition spec and a column list
INSERT OVERWRITE students PARTITION (student_id = 11215016) (address, name) VALUES
    ('Hangzhou, China', 'Kent Yao Jr.')
```

MERGE INTO

Last updated：2024-08-07 17:26:15

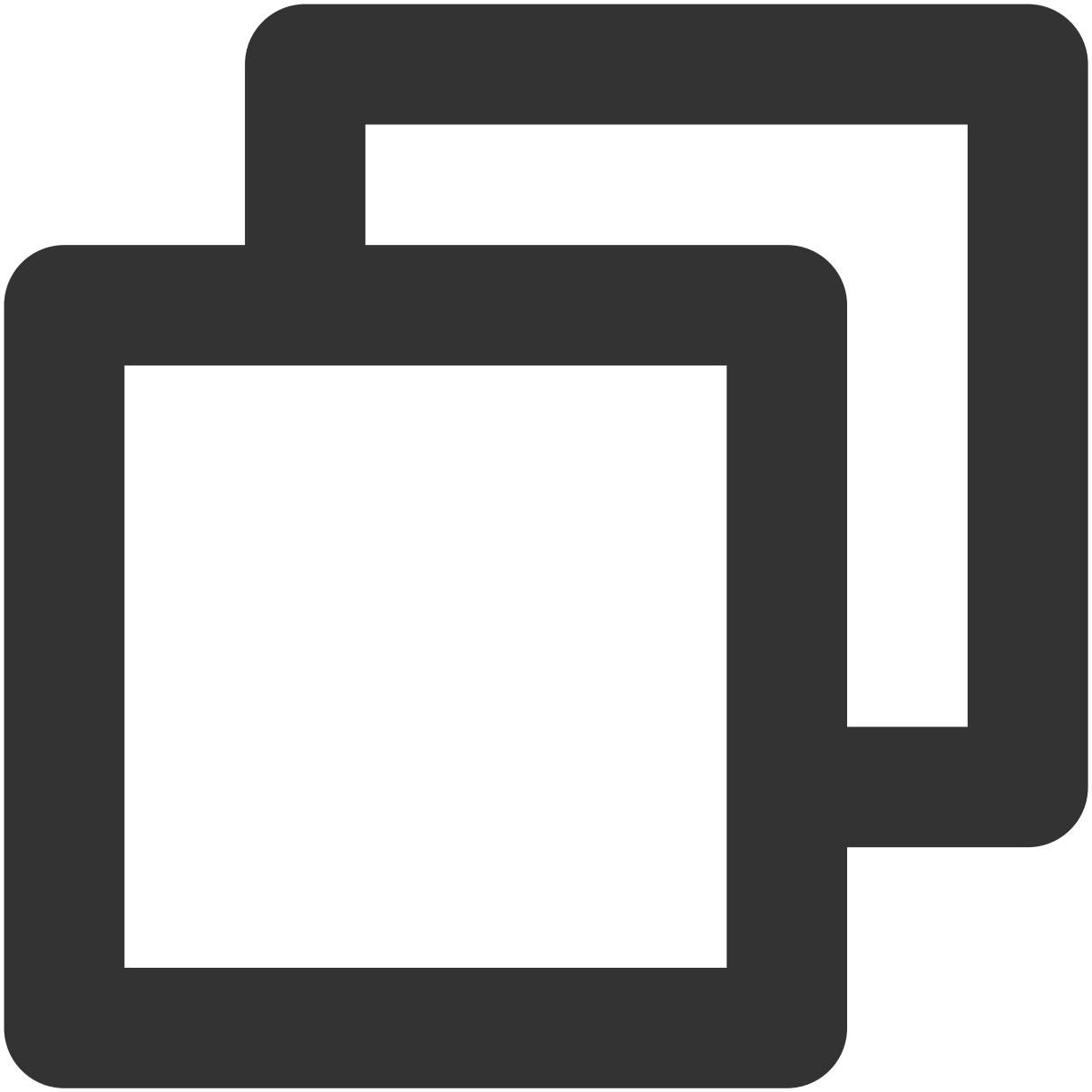
Description

Supported engine: SparkSQL

Applicable table: Native Iceberg tables and external tables

Purpose: Update row data. This statement can replace the INSERT OVERWRITE statement.

Statement



```
MERGE INTO tablePrimary1 [ [ AS ] alias ]
USING tablePrimary2
ON booleanExpression
[ WHEN MATCHED (AND matchedCond=booleanExpression)? THEN DELETE ]*
[ WHEN MATCHED (AND matchedCond=booleanExpression)? THEN UPDATE SET assign [, assig
[ WHEN NOT MATCHED (AND notMatchedCond=booleanExpression)? THEN INSERT VALUES '(' v
```

Parameter

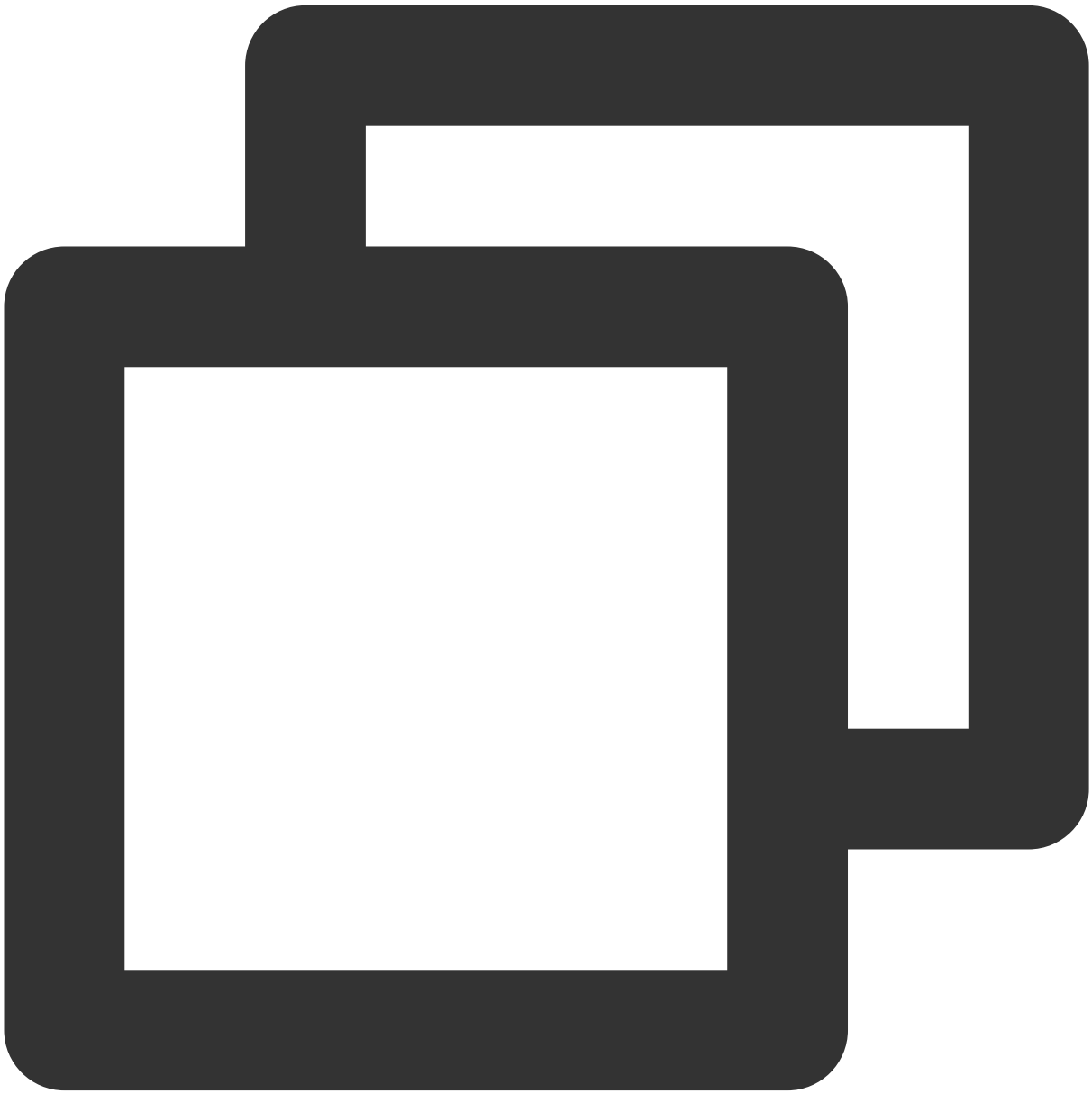
`tablePrimary1` : Table name in the three-part format. Example: catalog.database.table.

`alias` : Alias

`tablePrimary2` : Table name or subquery statement

`booleanExpression` : Boolean expression

Example



```
MERGE INTO catalog1.db2.tbl1 t
```

```
USING catalog1.db1.tbl1
ON t.col1 = tbl1.col1
WHEN MATCHED AND t.col1 = 14 THEN DELETE

MERGE INTO catalog1.db2.tbl1 t
USING (SELECT col1 FROM catalog1.db1.tbl1) s
ON t.col1 = s.col1
WHEN MATCHED AND t.col1 = 14 THEN UPDATE SET col1 = 2


MERGE INTO catalog1.db2.tbl1 t
USING (SELECT col1 FROM catalog1.db1.tbl1) s
ON t.col1 = s.col1
WHEN MATCHED AND t.col1 = 12 THEN UPDATE SET col1 = 0
WHEN MATCHED AND t.col1 = 13 THEN UPDATE SET col1 = 1
WHEN MATCHED AND t.col1 = 14 THEN UPDATE SET col1 = 2
WHEN MATCHED AND t.col1 = 15 or s.col1 = 16 THEN UPDATE SET col1 = t.col1 + 1
WHEN MATCHED AND t.col1 not in (12, 13, 14, 15) THEN UPDATE SET col1 = 4
WHEN NOT MATCHED AND t.col1 = 12 THEN INSERT (col1) VALUES (s.col1)
WHEN NOT MATCHED AND t.col1 = 13 THEN INSERT (col1) VALUES (s.col1 + 1)
WHEN NOT MATCHED AND t.col1 = 14 THEN INSERT (col1) VALUES (s.col1 + 2)


MERGE INTO catalog1.db2.tbl1 t
USING (SELECT col1, col2 FROM catalog1.db1.tbl1) s
ON t.col1 = s.col1
WHEN MATCHED AND t.col1 = fun1(s.col2) THEN DELETE
WHEN MATCHED AND t.col1 = db2.fun1(s.col2) THEN DELETE
WHEN MATCHED AND (t.col1 = length(s.col2) or t.col1 = catalog2.db2.fun3(s.col2)) TH
WHEN NOT MATCHED AND t.col1 = 12 THEN INSERT (col1) VALUES (db2.fun2(s.col2))
```

UPDATE

Last updated：2024-08-07 17:26:27

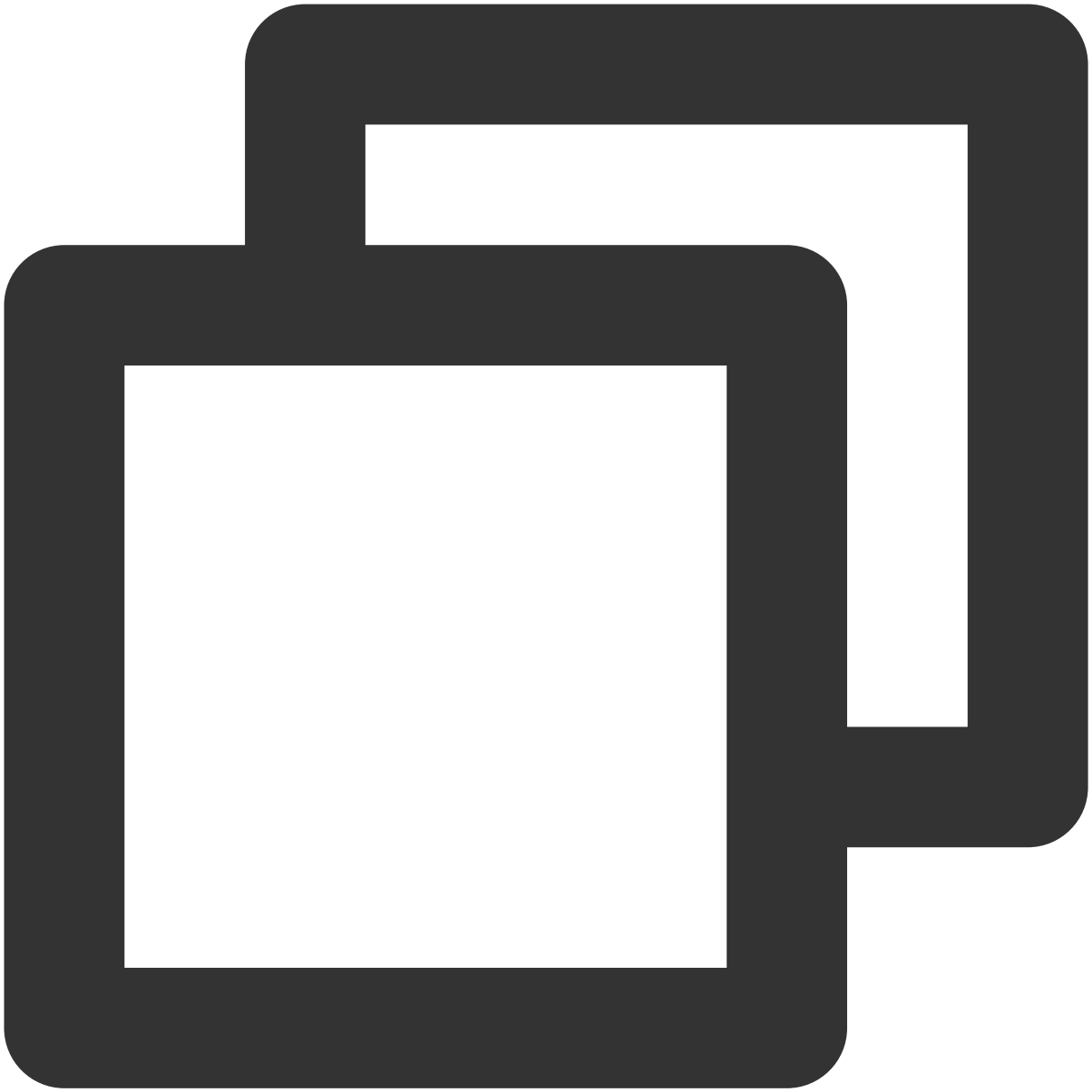
Description

Supported engine: SparkSQL

Applicable table: Native Iceberg tables

Purpose: Update specified row in data table.

Statement



```
UPDATE tablePrimary
SET assign [, assign ]*
[ WHERE booleanExpression ]
```

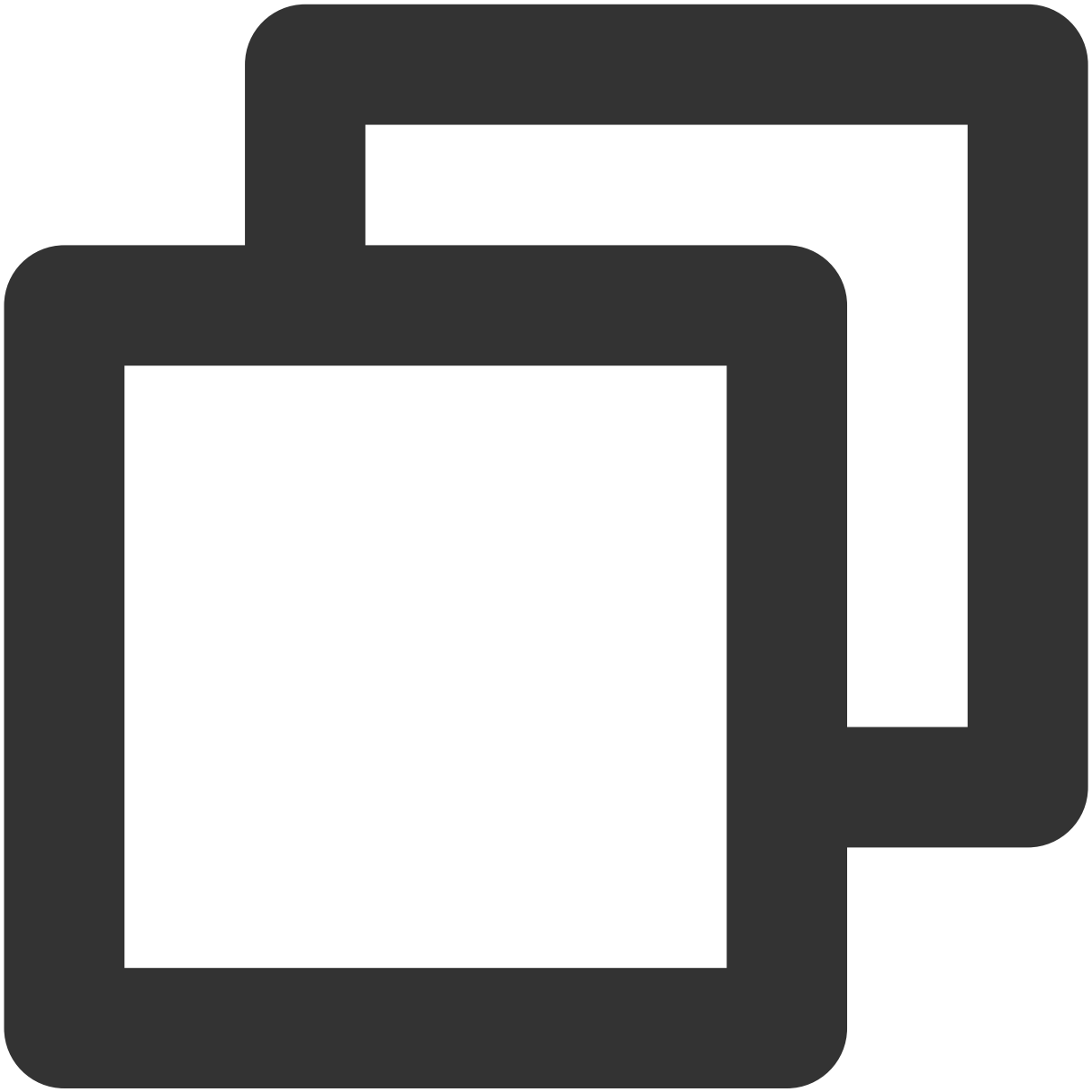
Parameter

`tablePrimary` : Table name in the three-part format. Example: catalog.database.table.

`assign` : Expression for row update. Example: col1 = 'new_data'.

`booleanExpression` : Boolean expression

Example



```
UPDATE prod.db.table
SET c1 = 'update_c1', c2 = 'update_c2'
WHERE ts >= '2020-05-01 00:00:00' and ts < '2020-06-01 00:00:00'

UPDATE prod.db.all_events
```

```
SET session_time = 0, ignored = true
WHERE session_time < (SELECT min(session_time) FROM prod.db.good_events)

UPDATE prod.db.orders AS t1
SET order_status = 'returned'
WHERE EXISTS (SELECT oid FROM prod.db.returned_orders WHERE t1.oid = oid)
```

DELETE STATEMENT

Last updated：2024-08-07 17:26:46

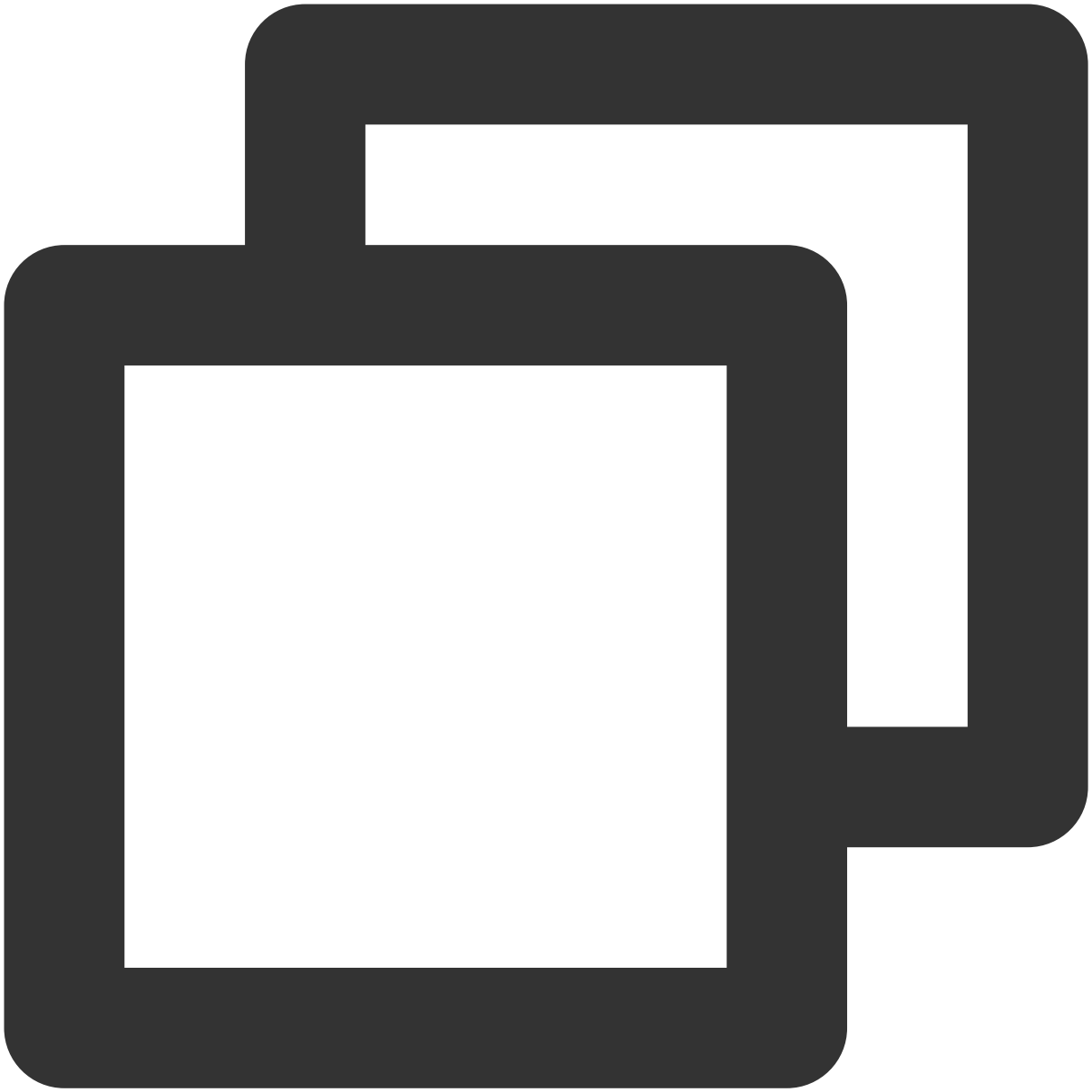
Description

Supported engine: SparkSQL

Applicable table: Native Iceberg tables

Purpose: Delete specified row in the data table.

Statement

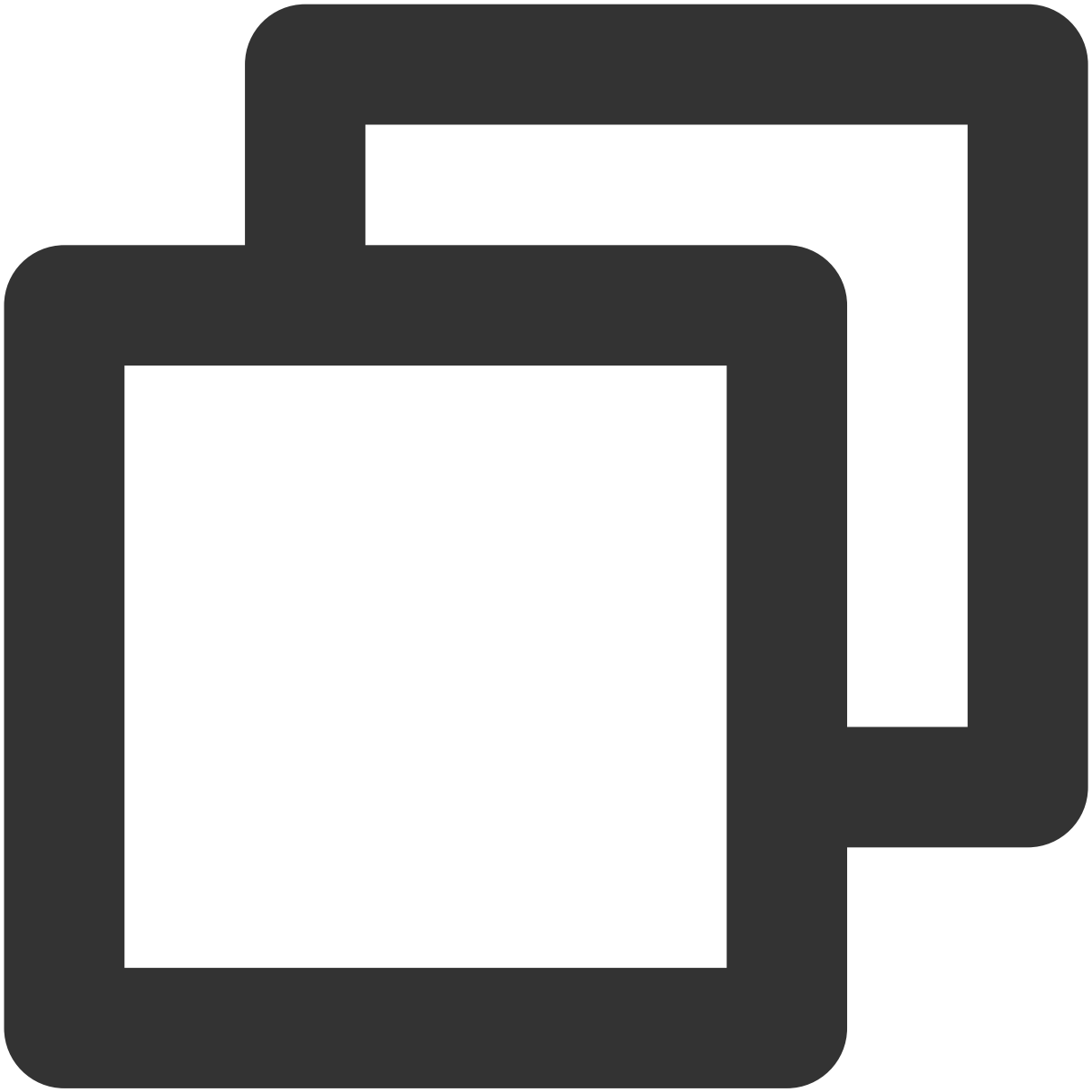


```
DELETE FROM table_name [ [ AS ] alias ]
[ WHERE booleanExpression ]
```

Parameter

`table_identifier` : Table name in the three-part format. Example: catalog.database.table.

Example



```
DELETE FROM lineitem WHERE shipmode = 'AIR';

DELETE FROM lineitem
WHERE orderkey IN (SELECT orderkey FROM orders WHERE priority = 'LOW');

DELETE FROM orders;
```

TABLE METADATA

Last updated：2024-08-07 17:26:59

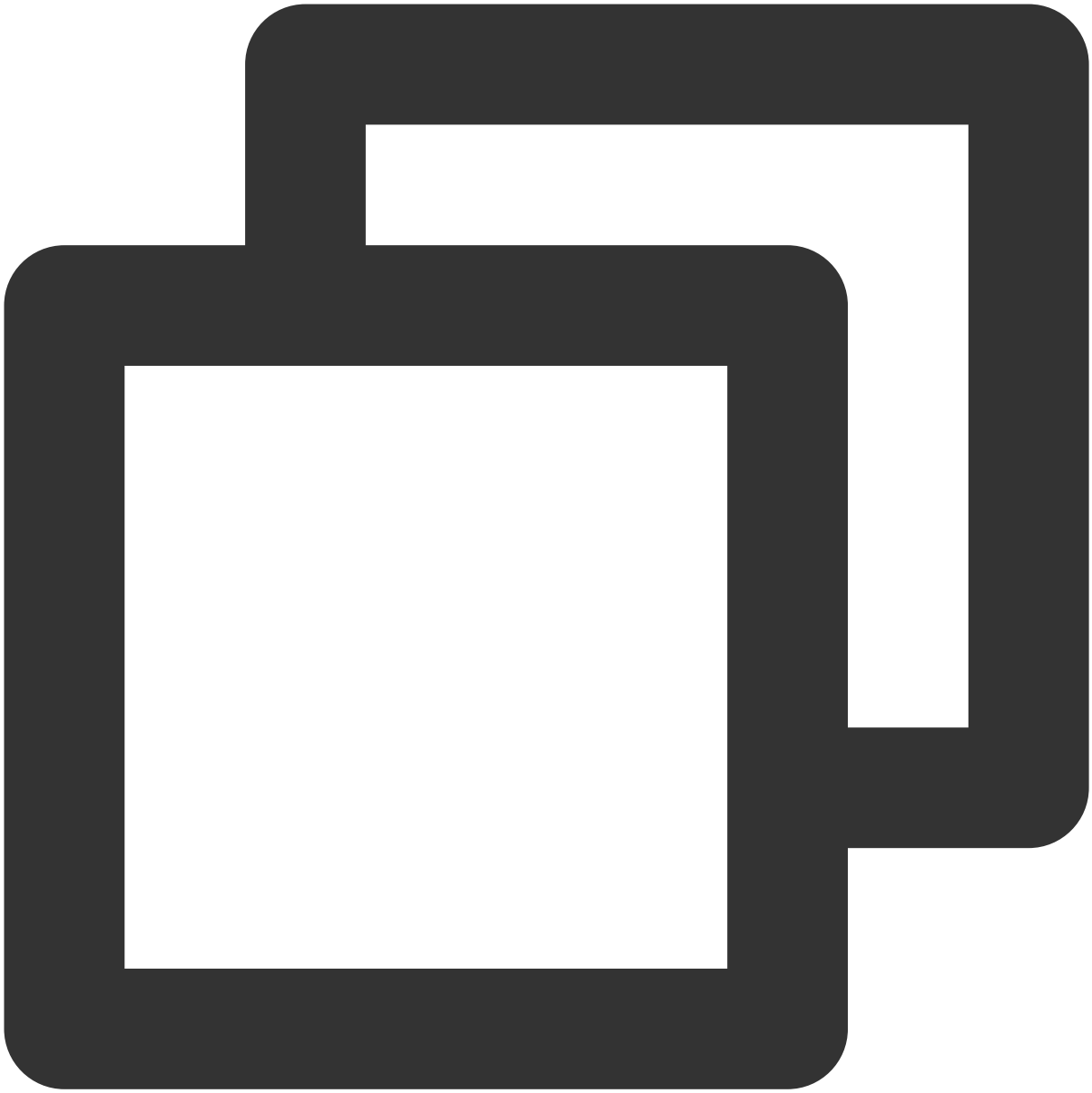
Description

Supported engine: SparkSQL

Applicable table: Native Iceberg tables.

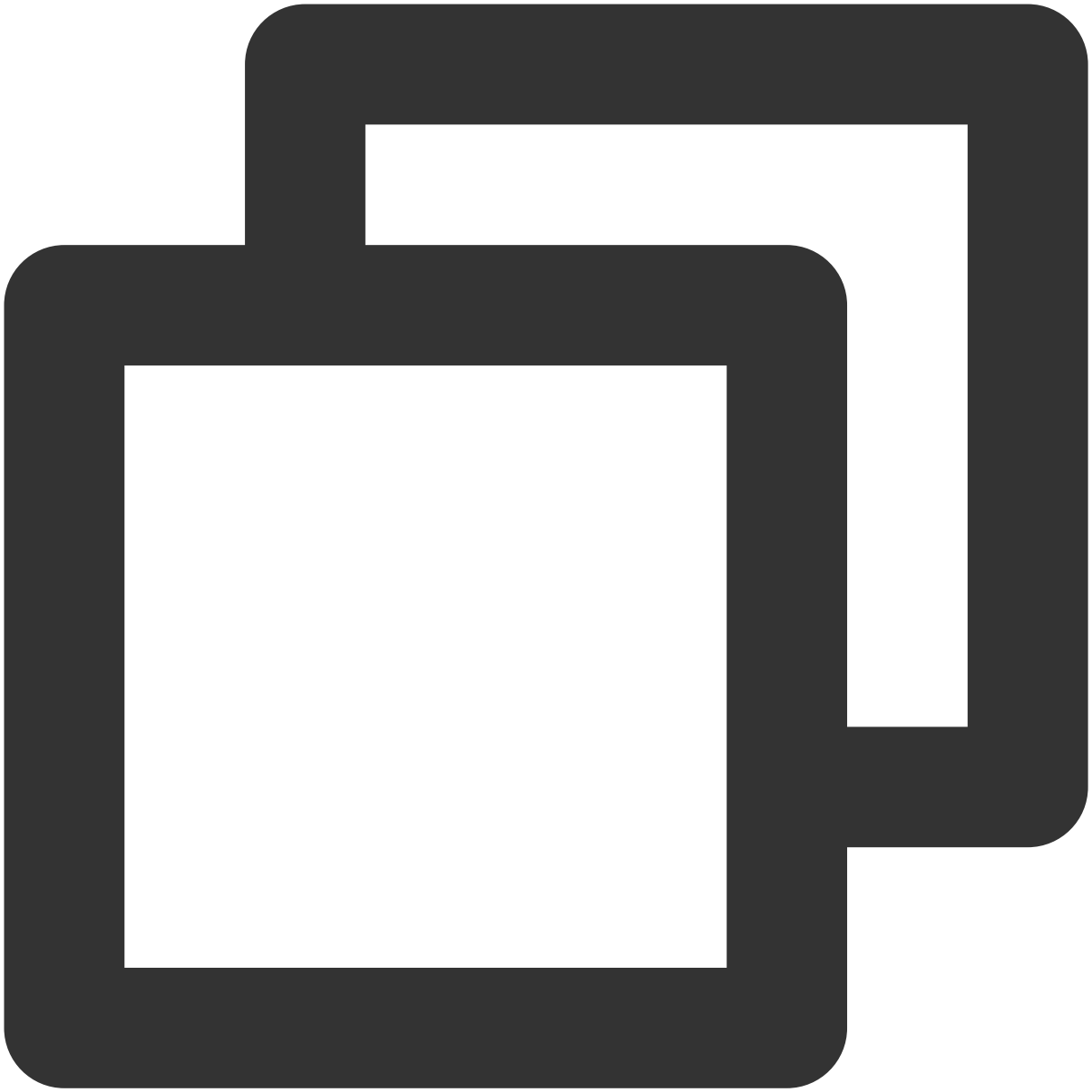
Purpose: Query Iceberg table metadata in four-part format, including history, snapshots, files, and manifests.

Statement



```
SELECT select_expr (, select_expr)*
FROM `Catalog`.`db`.`tableName${history|snapshots|files|manifests|partitions|all_da
[WHERE where_condition]
[LIMIT [offset,] rows]
```

Example



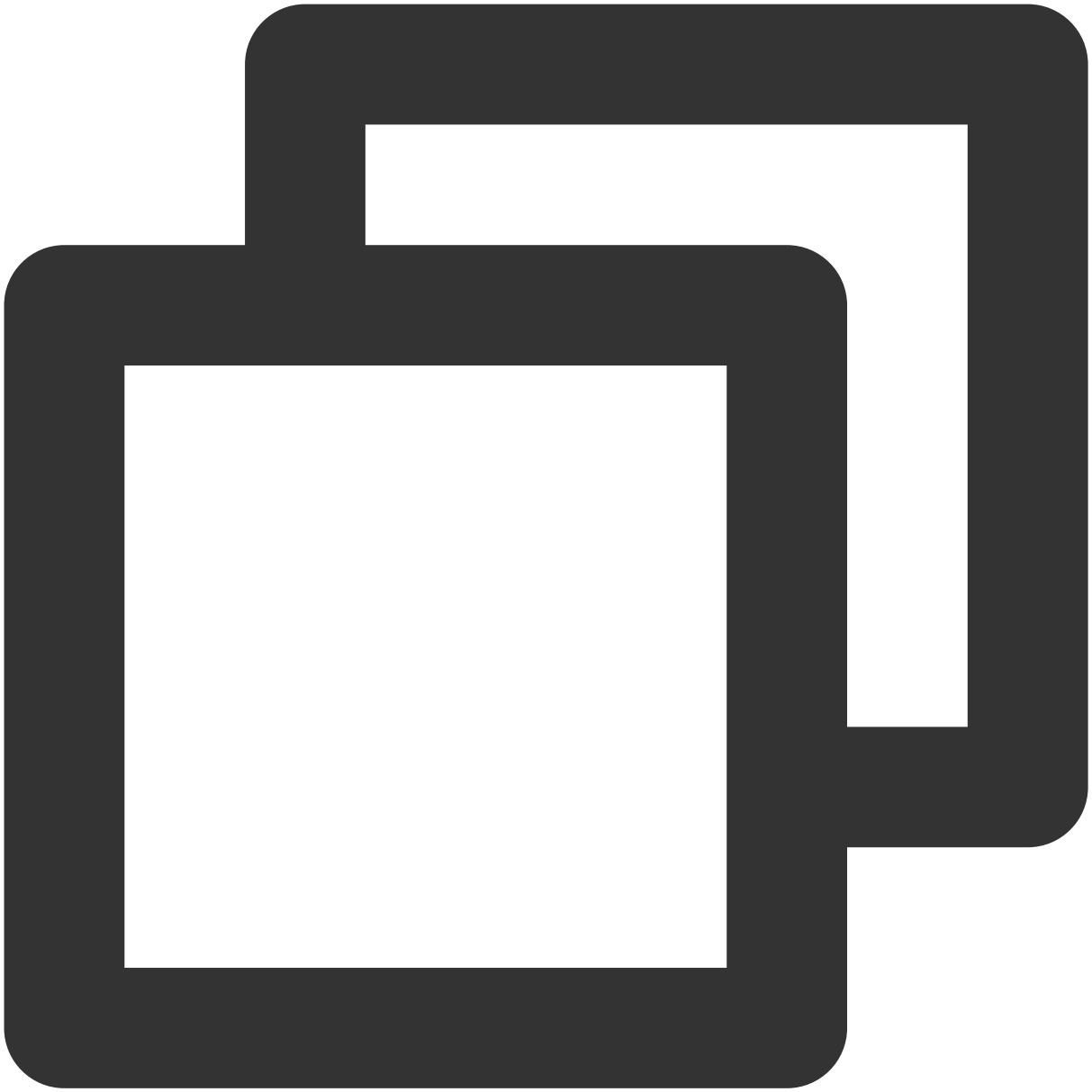
```
SELECT * FROM `DataLakeCatalog`.`validation`.`dempts$history` ORDER BY snapshot_id
SELECT * FROM `DataLakeCatalog`.`validation`.`dempts$snapshots` ORDER BY snapshot_i
SELECT * FROM `DataLakeCatalog`.`validation`.`dempts$files` ORDER BY file_size_in_b
SELECT * FROM `DataLakeCatalog`.`validation`.`dempts$manifests` ORDER BY length LIM
SELECT * FROM `DataLakeCatalog`.`validation`.`dempts$partitions` LIMIT 10;
SELECT * FROM `DataLakeCatalog`.`validation`.`dempts$all_data_files` LIMIT 10;
SELECT * FROM `DataLakeCatalog`.`validation`.`dempts$all_manifests` LIMIT 10;
```

DQL Statement
SELECT STATEMENT

Last updated：2024-08-07 17:27:53

`SELECT` statement: Retrieve rows from multiple tables.

Statement

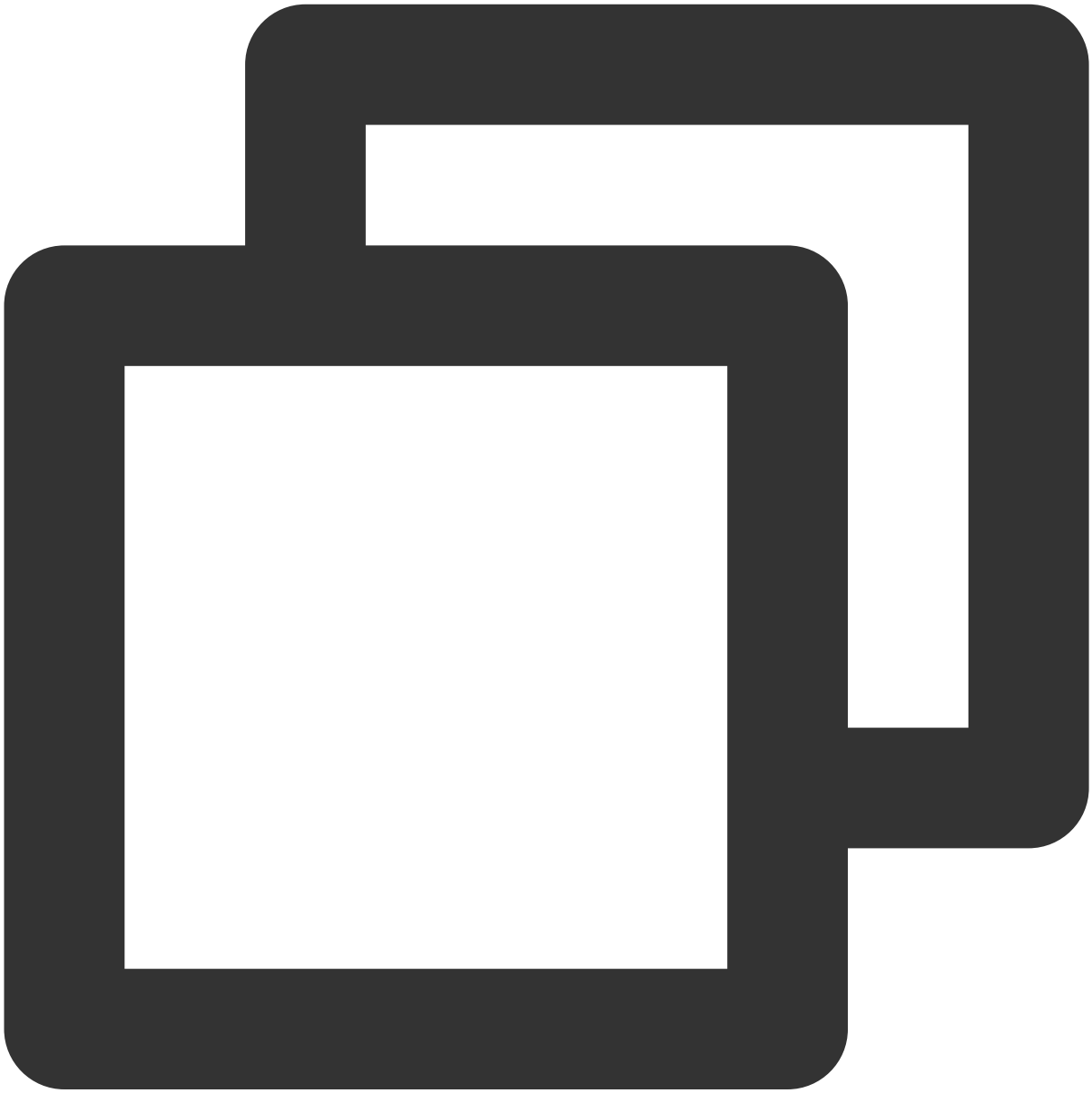


```
[ WITH with_query [, ...] ]
SELECT [ ALL | DISTINCT ] select_expression [, ...]
[ FROM from_item [, ...] ]
[ WHERE condition ]
[ GROUP BY [ ALL | DISTINCT ] grouping_element [, ...] ]
[ HAVING condition ]
[ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] select ]
[ ORDER BY expression [ ASC | DESC ] [ NULLS FIRST | NULLS LAST] [, ...] ]
[ LIMIT [ count | ALL ] ]
```

Parameter

[WITH with_query [, ...]]

You can use WITH to flatten nested queries or simplify subqueries. The `with_query` statement is as follows:



```
subquery_table_name [ ( column_name [, ...] ) ] AS (subquery)
```

subquery_table_name indicates the unique name of a temporary table used to define the results of WITH subqueries. Each subquery must contain a table name that can be referenced in the FROM clause.

column_name [, ...] indicates an optional list of output column names. The number of column names must be equal to or less than that of columns defined in the subquery.
subquery indicates a query statement.

[ALL | DISTINCT] select_expr

The ALL and DISTINCT options specify whether duplicate rows are returned. If none of these options are given, the default value ALL is used, which indicates that all matching rows are returned. DISTINCT specifies that duplicate rows are removed from the result set.

FROM from_item [, ...]

from_item can indicate a view, table, or subquery. If a multi-table join statement is used, supported JOIN statements are as follows:

`[ INNER ] JOIN`

`LEFT [ OUTER ] JOIN`

`RIGHT [ OUTER ] JOIN`

`FULL [ OUTER ] JOIN`

`CROSS JOIN`

`ON join_condition` : If `join_condition` is used, you can specify the column name for the join key of multiple tables. If `join_column` is used, the join column must exist in both tables.

[WHERE condition]

Results are filtered according to the condition you have specified, and the set of results meeting the condition is returned.

[GROUP BY [ALL | DISTINCT] grouping_expressions [, ...]]

The GROUP BY clause divides the output into groups based on the specified column names.

[HAVING condition]

This clause is used in conjunction with aggregate functions and the GROUP BY clause. It controls which groups are selected so as to eliminate groups that do not satisfy the given conditions. It filters groups after groups and aggregates are computed.

[{UNION | INTERSECT | EXCEPT} [ALL | DISTINCT] union_query]

`UNION` , `INTERSECT` , and `EXCEPT` combine the results of more than one statement. `UNION` combines all the rows in the result set from the first query with those in the result set from the second query. To eliminate duplication, UNION builds a hash table, which consumes memory resources. UNION ALL is recommended for better performance.

`INTERSECT` returns only the rows in the result sets of both the first and the second queries.

`EXCEPT` returns the rows in the result set of the first query, but not the second.

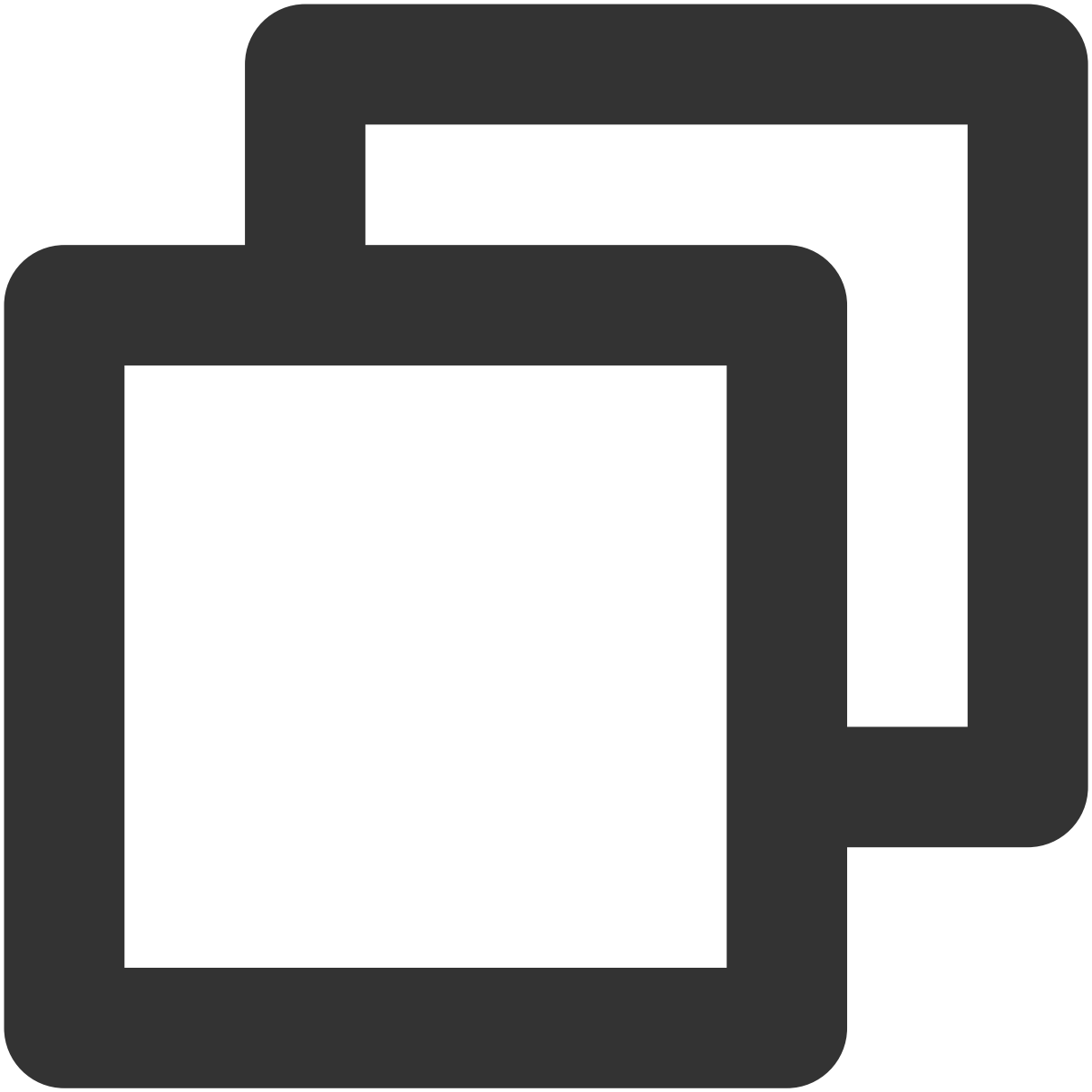
[ORDER BY expression [ASC | DESC] [NULLS FIRST | NULLS LAST] [, ...]]

This statement sorts a result set by one or more output expressions. If the clause contains multiple expressions, sorting is performed using the first `expression`. Then, the second `expression` is used to sort rows matching the first expression, and so on.

Example

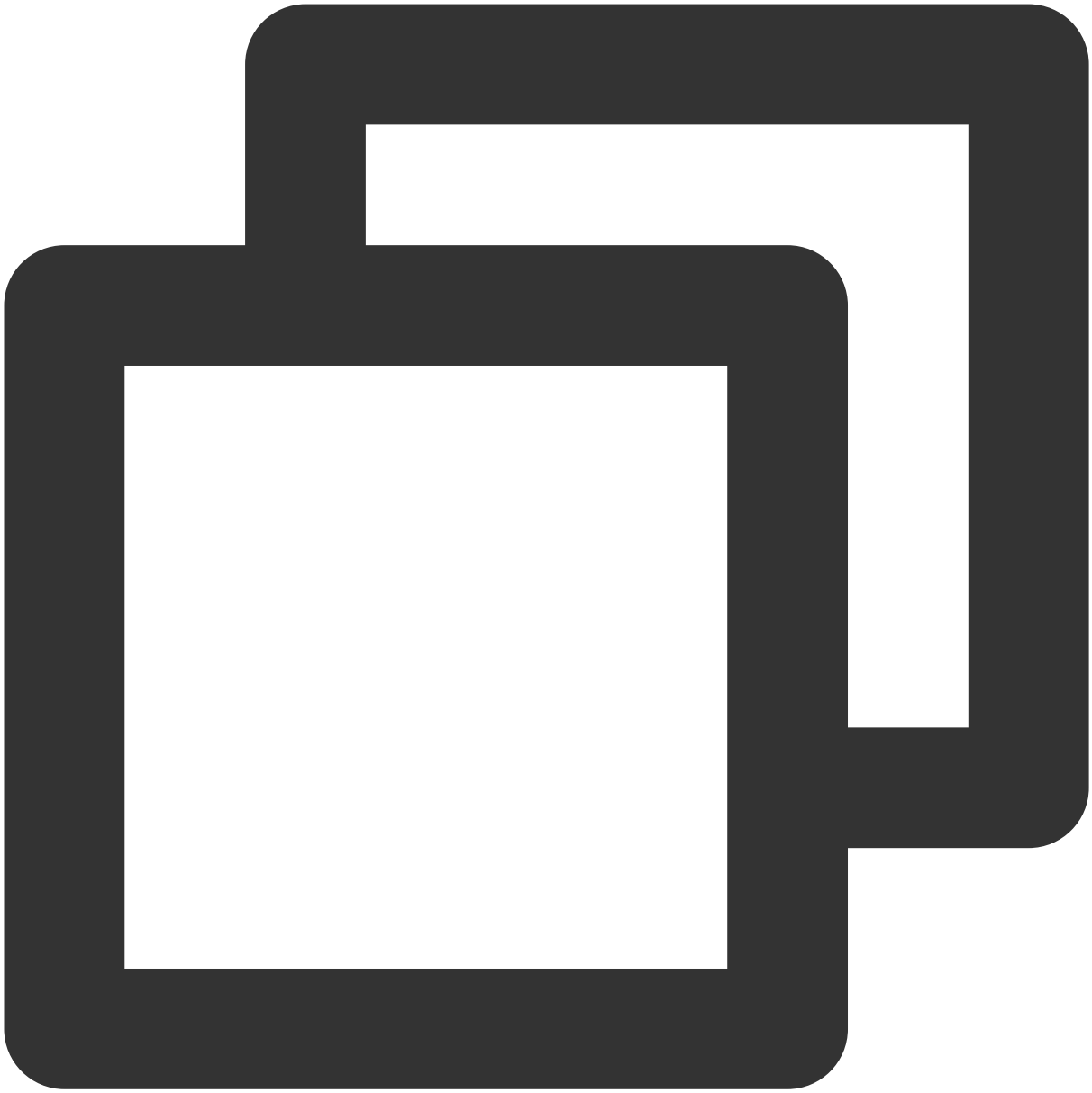
WITH Clause

The WITH clause defines named relations for use within a query. It allows flattening nested queries or simplifying subqueries. For example, the following queries are equivalent:



```
WITH x AS (SELECT a, MAX(b) AS b FROM t GROUP BY a)
SELECT a, b FROM x;
```

This also works with multiple subqueries:

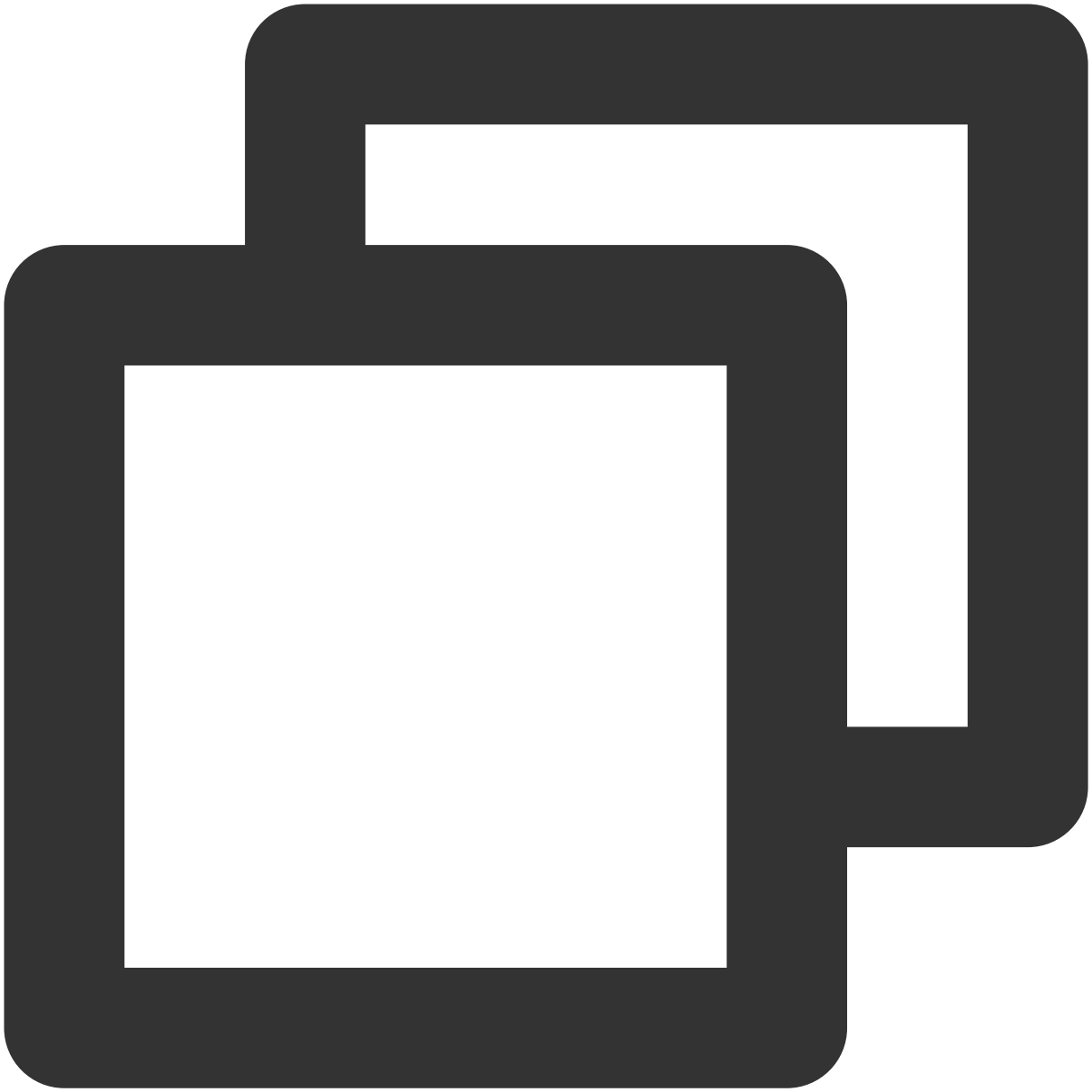


```
WITH
  t1 AS (SELECT a, MAX(b) AS b FROM x GROUP BY a),
  t2 AS (SELECT a, AVG(d) AS d FROM y GROUP BY a)
SELECT t1.*, t2.*
FROM t1
JOIN t2 ON t1.a = t2.a;
```

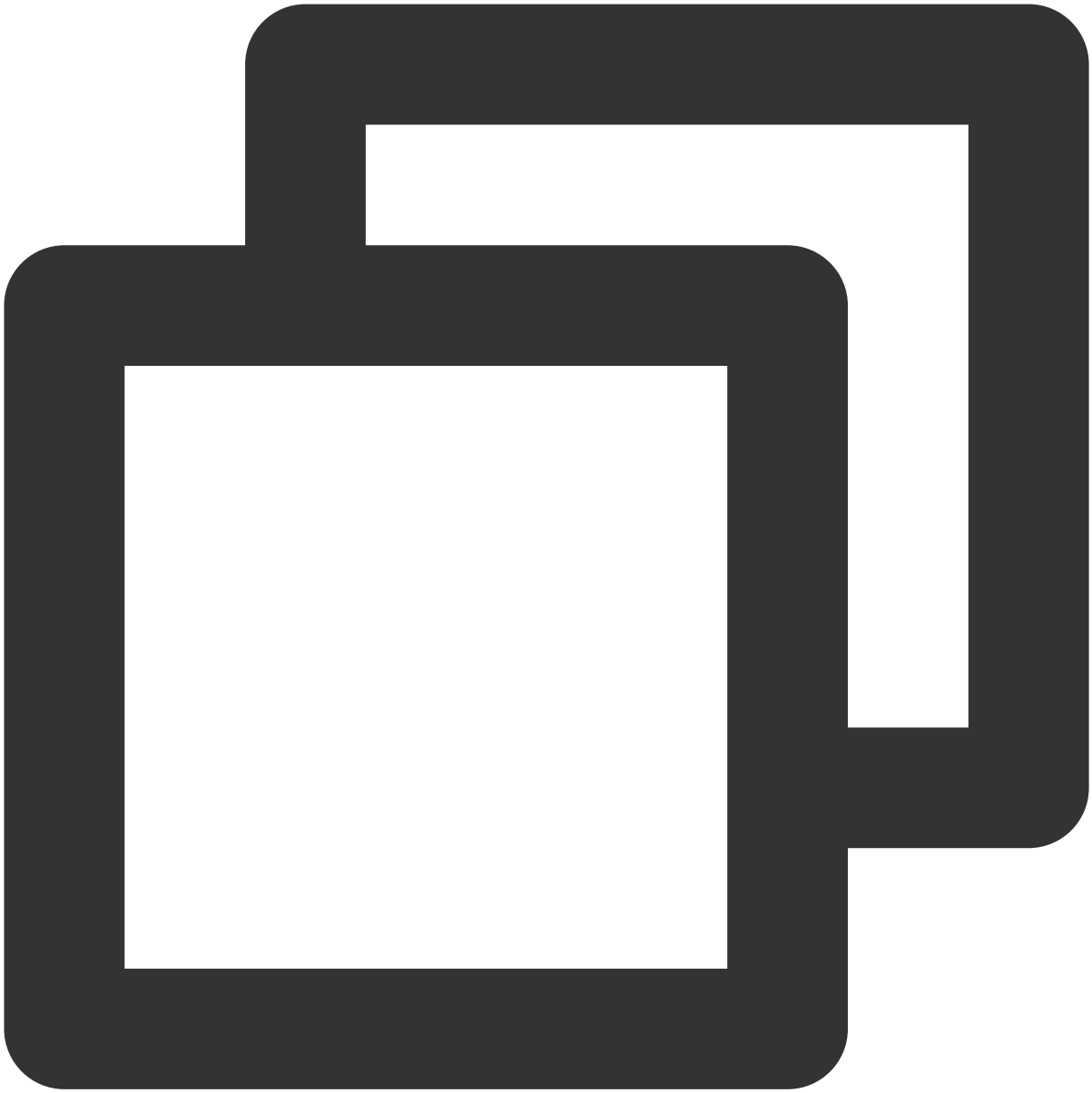
GROUP BY Clause

The GROUP BY clause divides the output of a SELECT statement into groups of rows containing matching values. A simple GROUP BY clause may contain any expression composed of input columns, or it may be an ordinal number selecting an output column by position.

```
SELECT count(*), nationkey FROM customer GROUP BY 2;
```



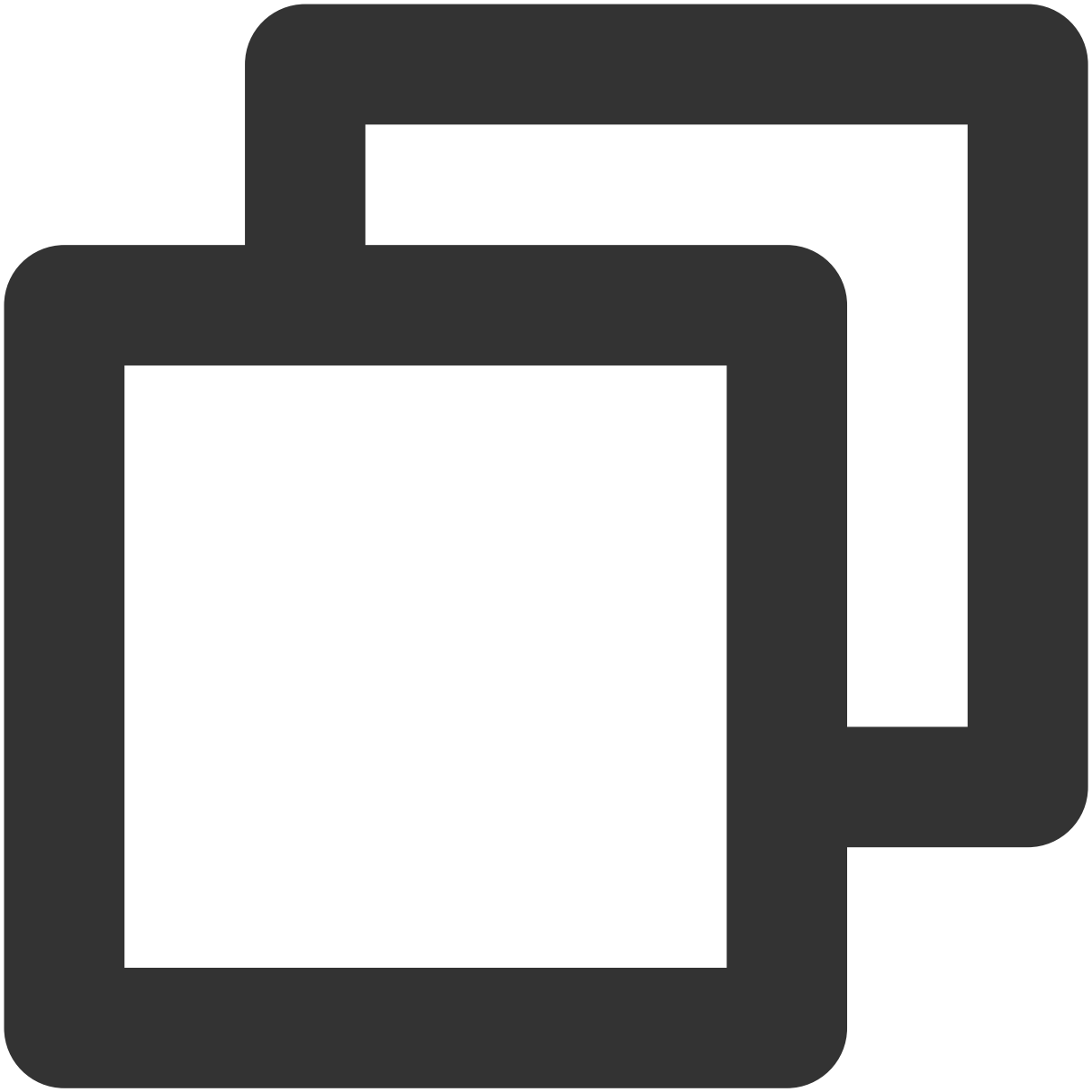
```
SELECT count(*), nationkey FROM customer GROUP BY nationkey;
```



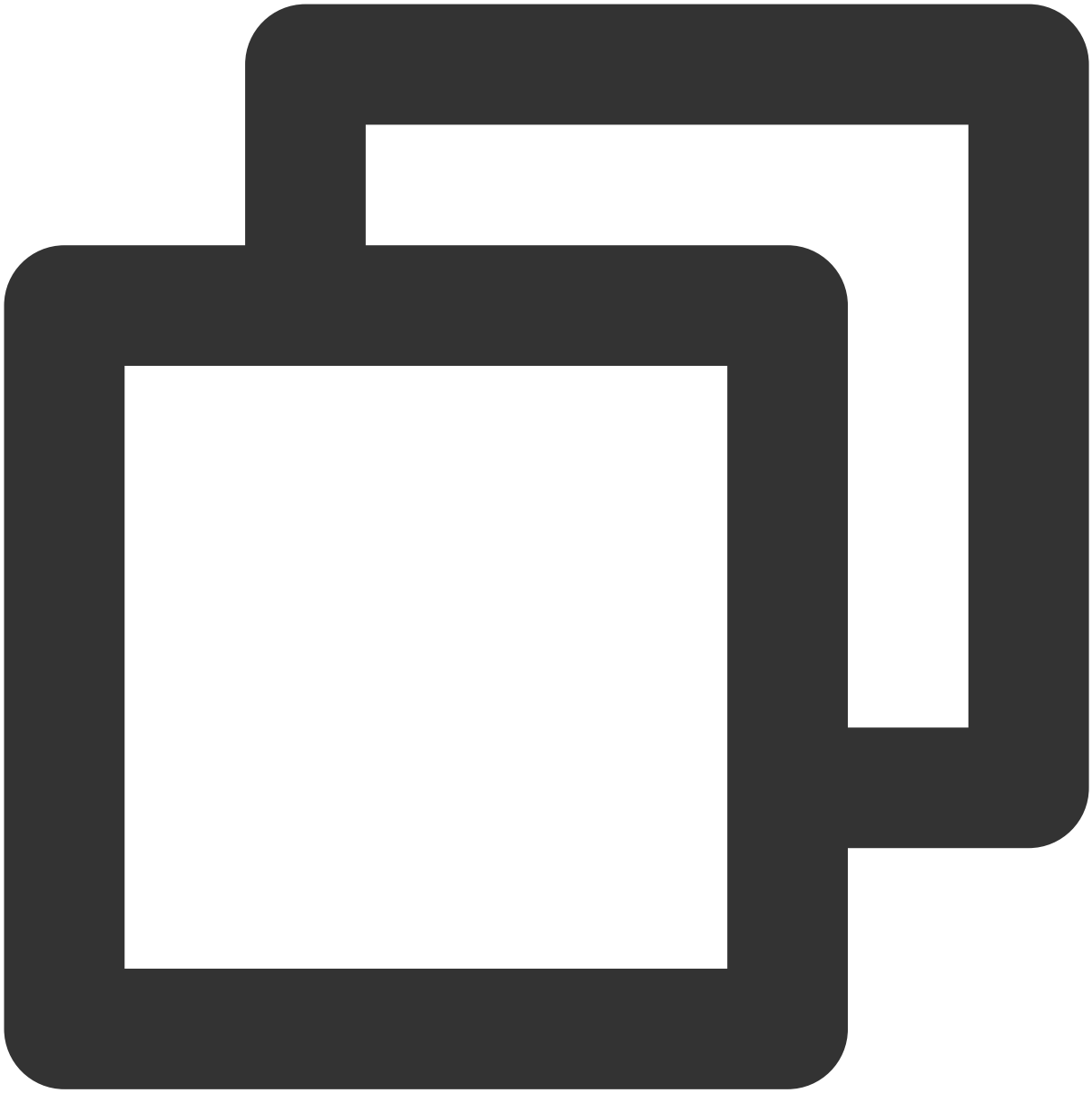
```
SELECT count(*) FROM customer GROUP BY mktsegment;
```

GROUPING SETS

Grouping sets allow users to specify multiple lists of columns to group on. The columns not part of a given sublist of grouping columns are set to NULL.



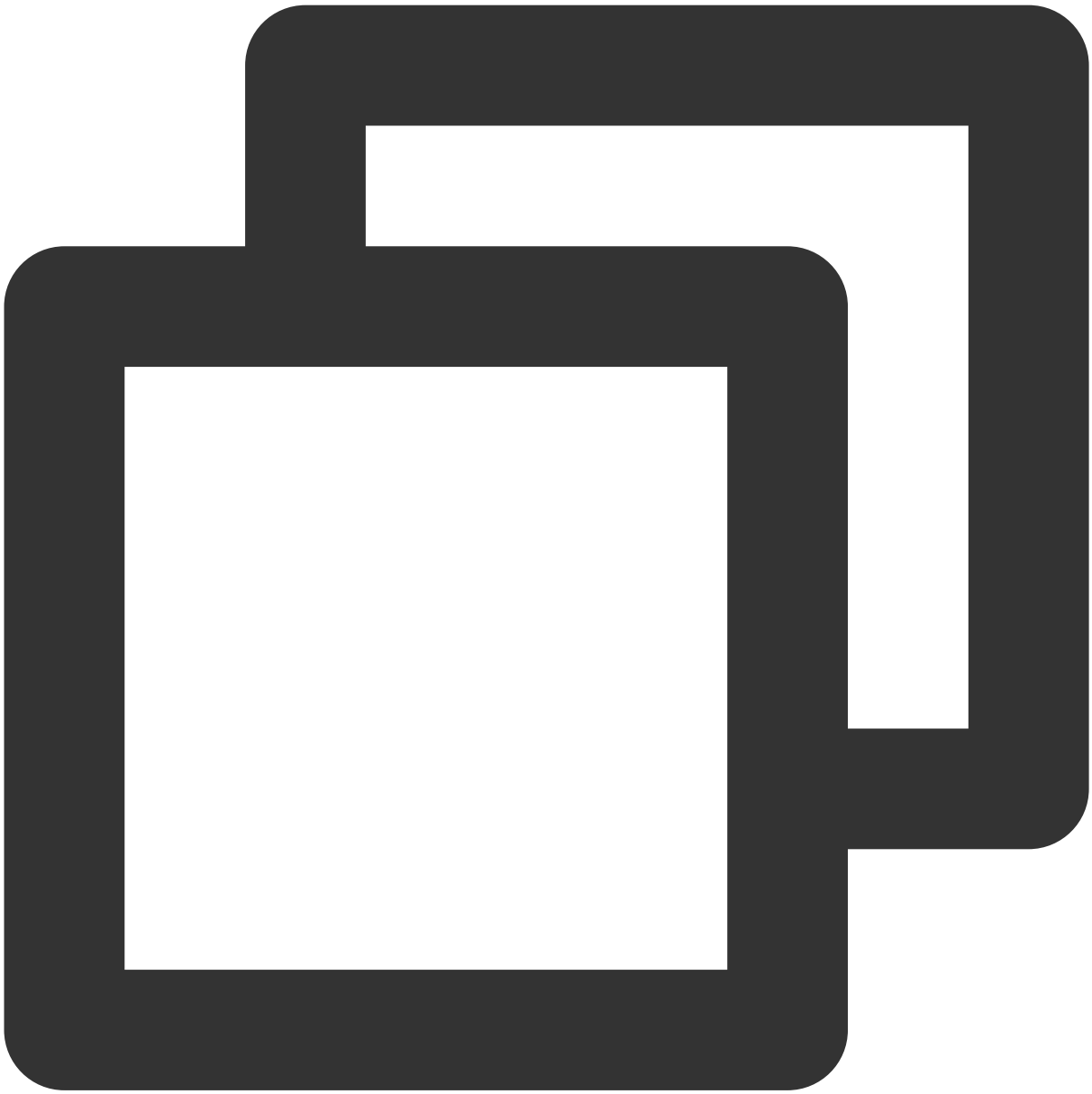
```
SELECT origin_state, origin_zip, destination_state, sum(package_weight)
FROM shipping
GROUP BY GROUPING SETS (
    (origin_state),
    (origin_state, origin_zip),
    (destination_state));
```



```
SELECT origin_state, NULL, NULL, sum(package_weight)
FROM shipping GROUP BY origin_state
UNION ALL
SELECT origin_state, origin_zip, NULL, sum(package_weight)
FROM shipping GROUP BY origin_state, origin_zip
UNION ALL
SELECT NULL, NULL, destination_state, sum(package_weight)
FROM shipping GROUP BY destination_state;
```

HAVING Clause

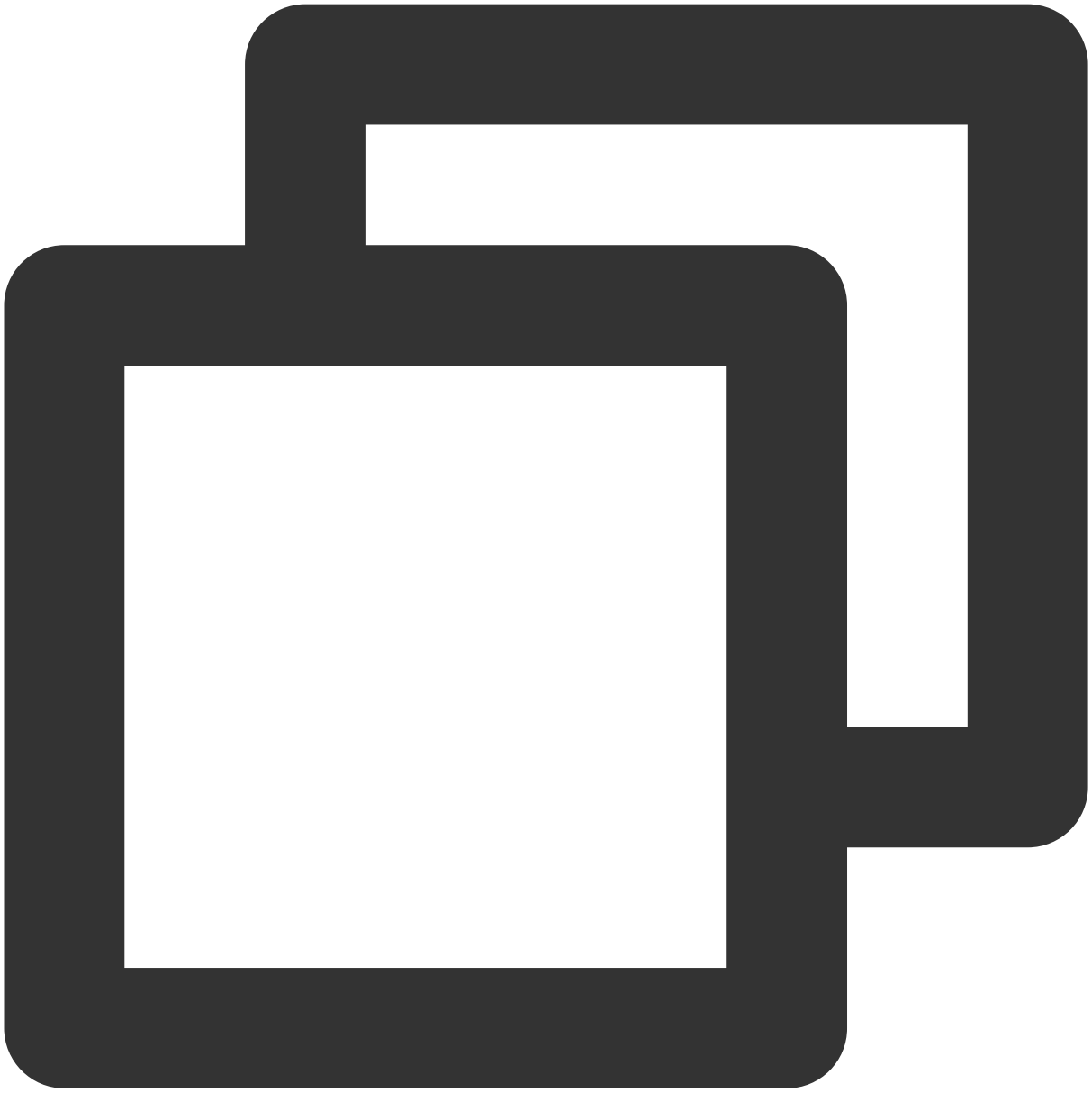
The HAVING clause is used in conjunction with aggregate functions and the GROUP BY clause to control which groups are selected. A HAVING clause eliminates groups that do not satisfy the given conditions.



```
SELECT count(*), mktsegment, nationkey,
       CAST(sum(acctbal) AS bigint) AS totalbal
FROM customer
GROUP BY mktsegment, nationkey
HAVING sum(acctbal) > 5700000
ORDER BY totalbal DESC;
```

IN

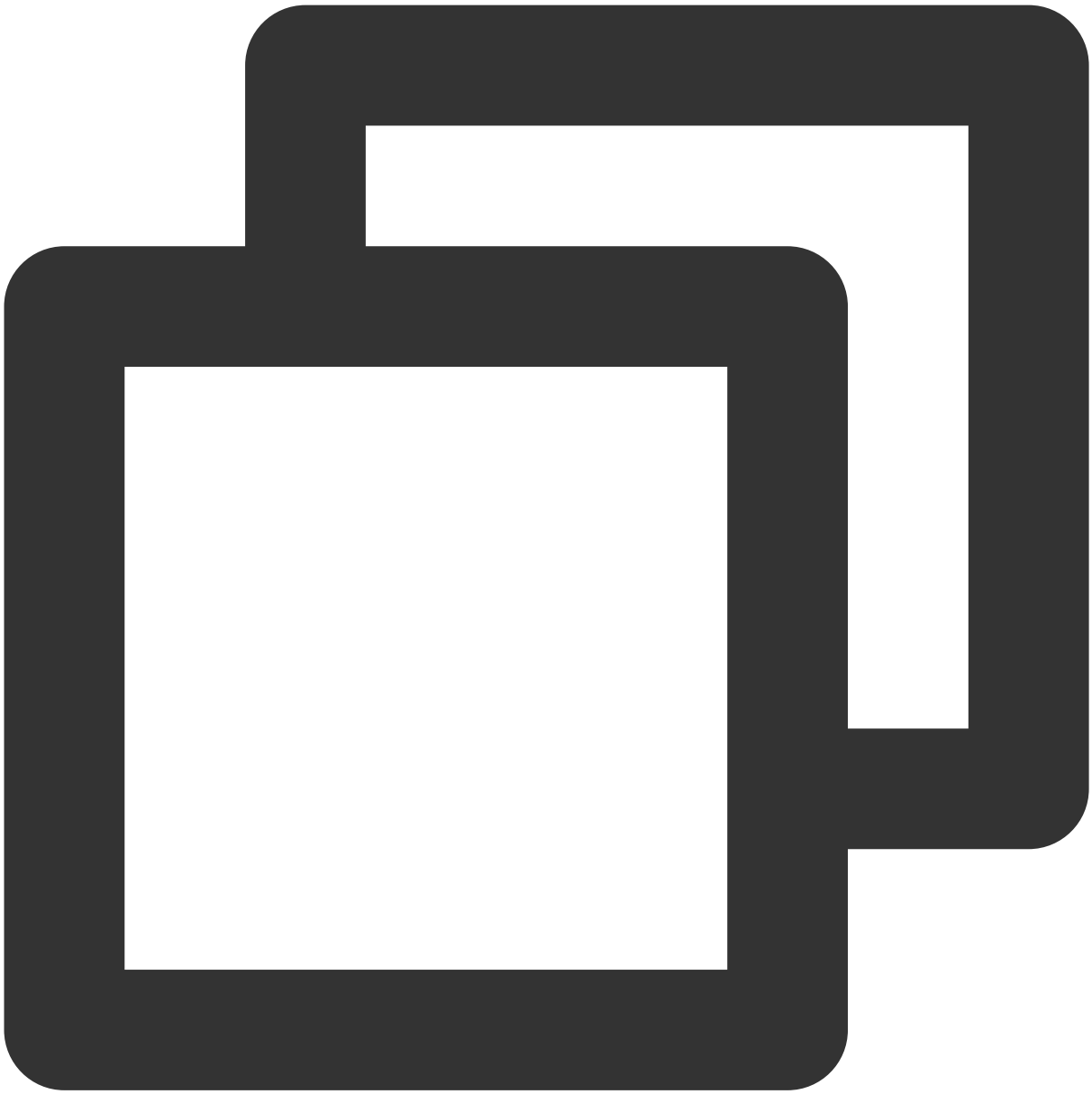
The IN operator allows users to specify multiple values in the WHERE clause.



```
SELECT name
FROM nation
WHERE regionkey IN (SELECT regionkey FROM region)
```

EXISTS

The EXISTS operator determines if a subquery returns any rows. If one or more rows are returned, this operator returns True. Otherwise, it returns False.



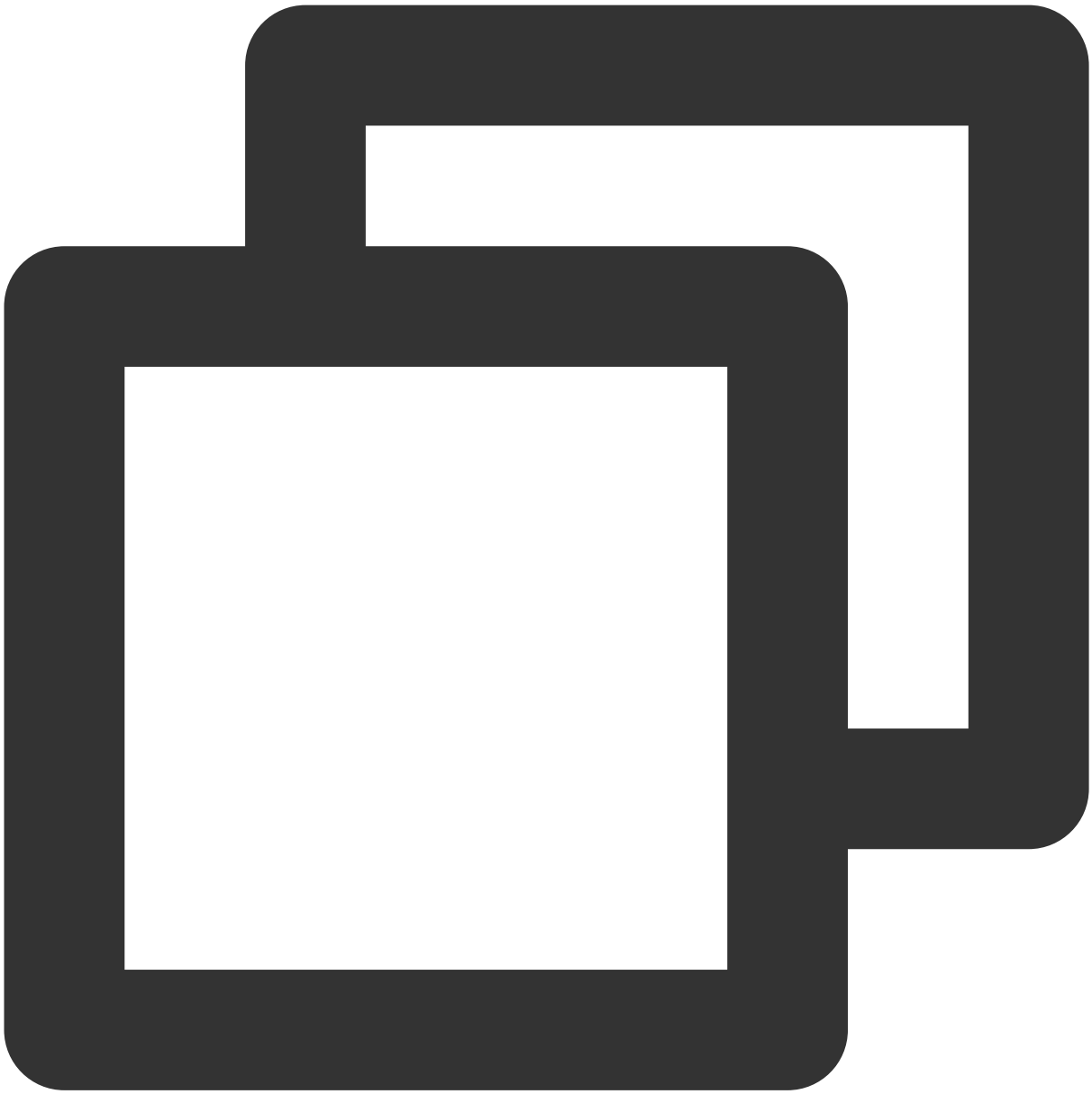
```
SELECT column_name(s)
FROM table_name
WHERE EXISTS
(SELECT column_name FROM table_name WHERE condition)
```

USING

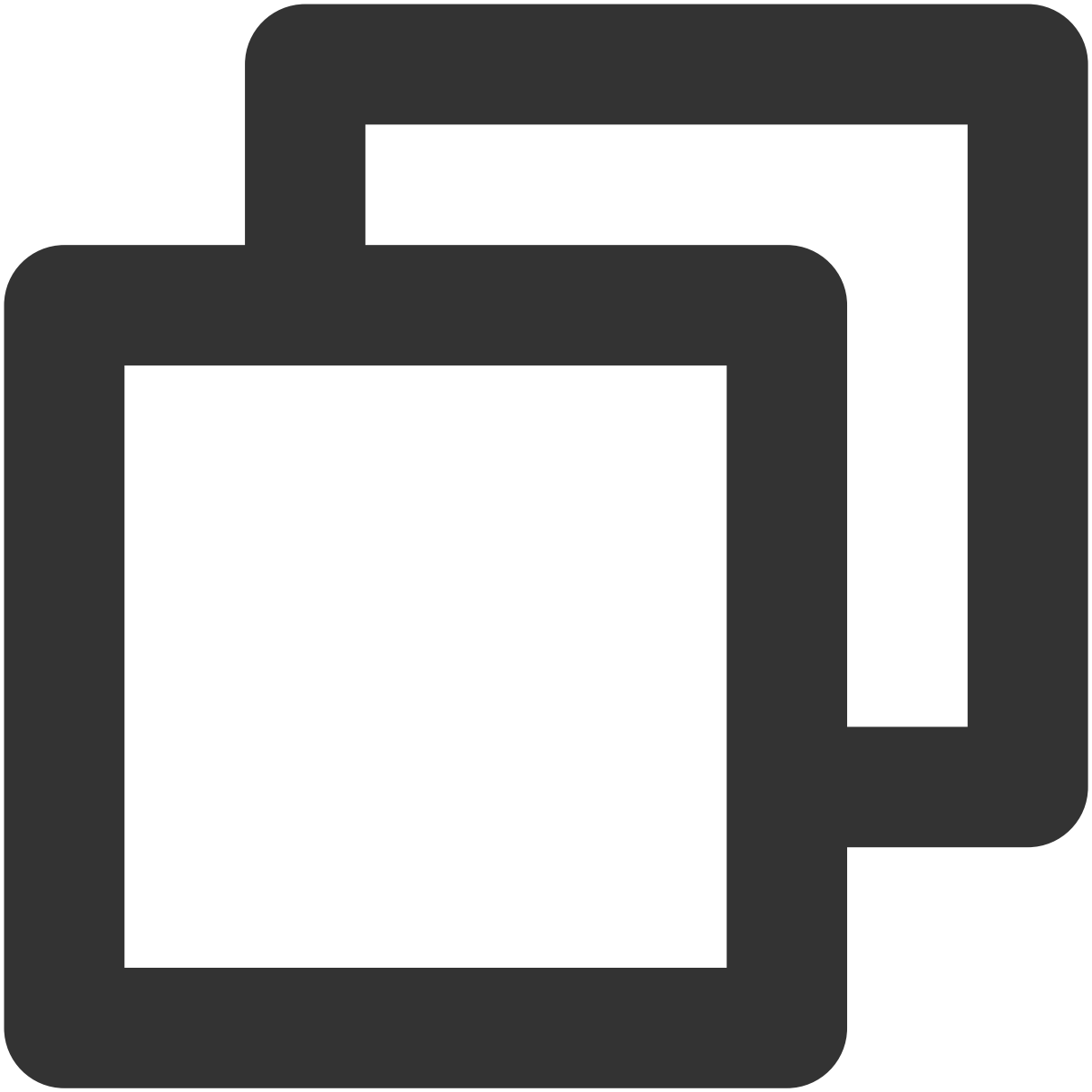
The USING keyword is used to simplify queries.

The query must be an equal join.

Columns in an equal join must have the same name and data type.



```
SELECT *
FROM table_1
JOIN table_2
USING (key_A, key_B)
```

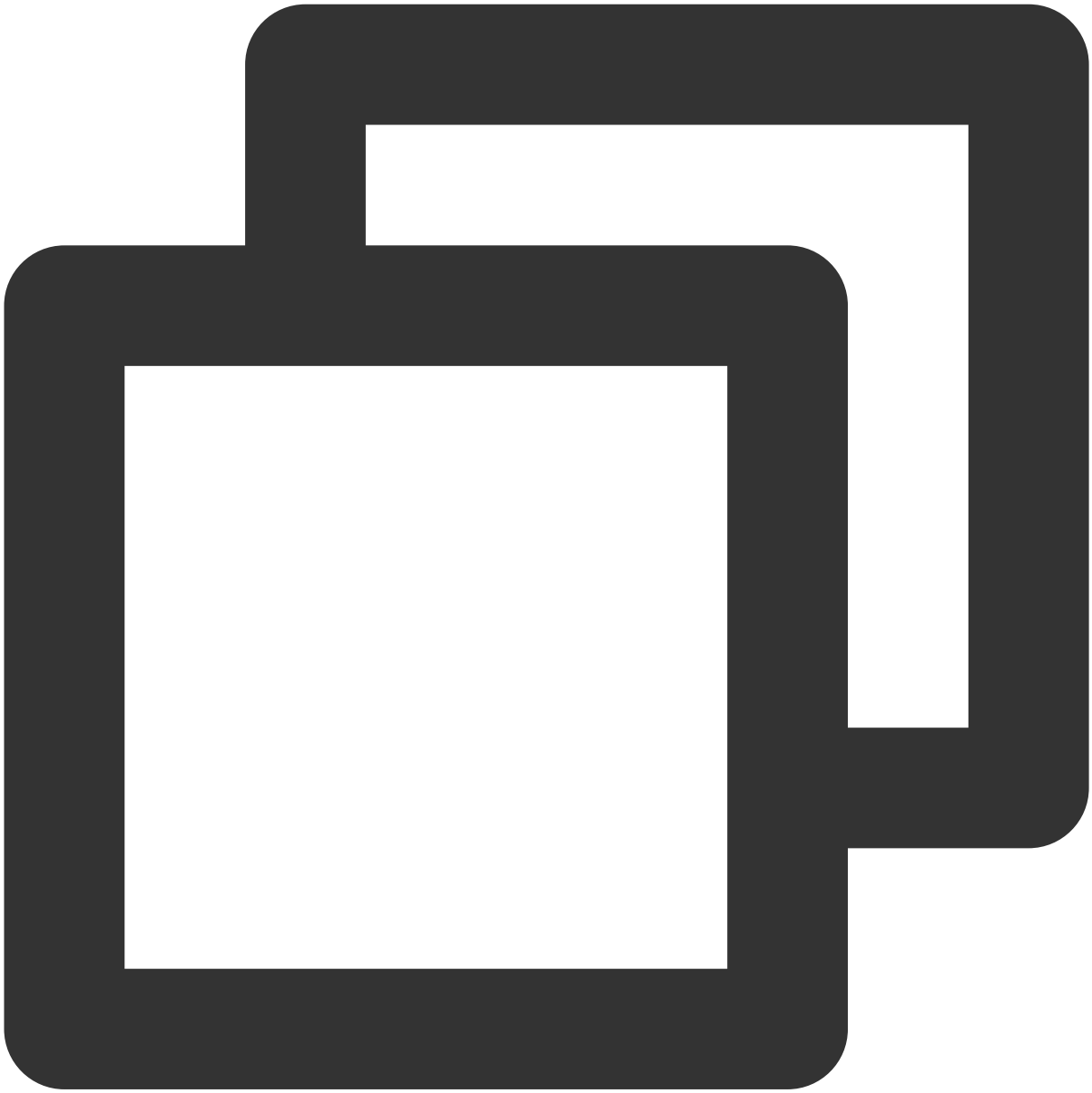


```
SELECT *
FROM (
    VALUES
        (1, 3, 10),
        (2, 4, 20)
) AS table_1 (key_A, key_B, y1)
LEFT JOIN (
    VALUES
        (1, 3, 100),
        (2, 4, 200)
) AS table_2 (key_A, key_B, y2)
```

```
USING (key_A, key_B);
```

CROSS JOIN

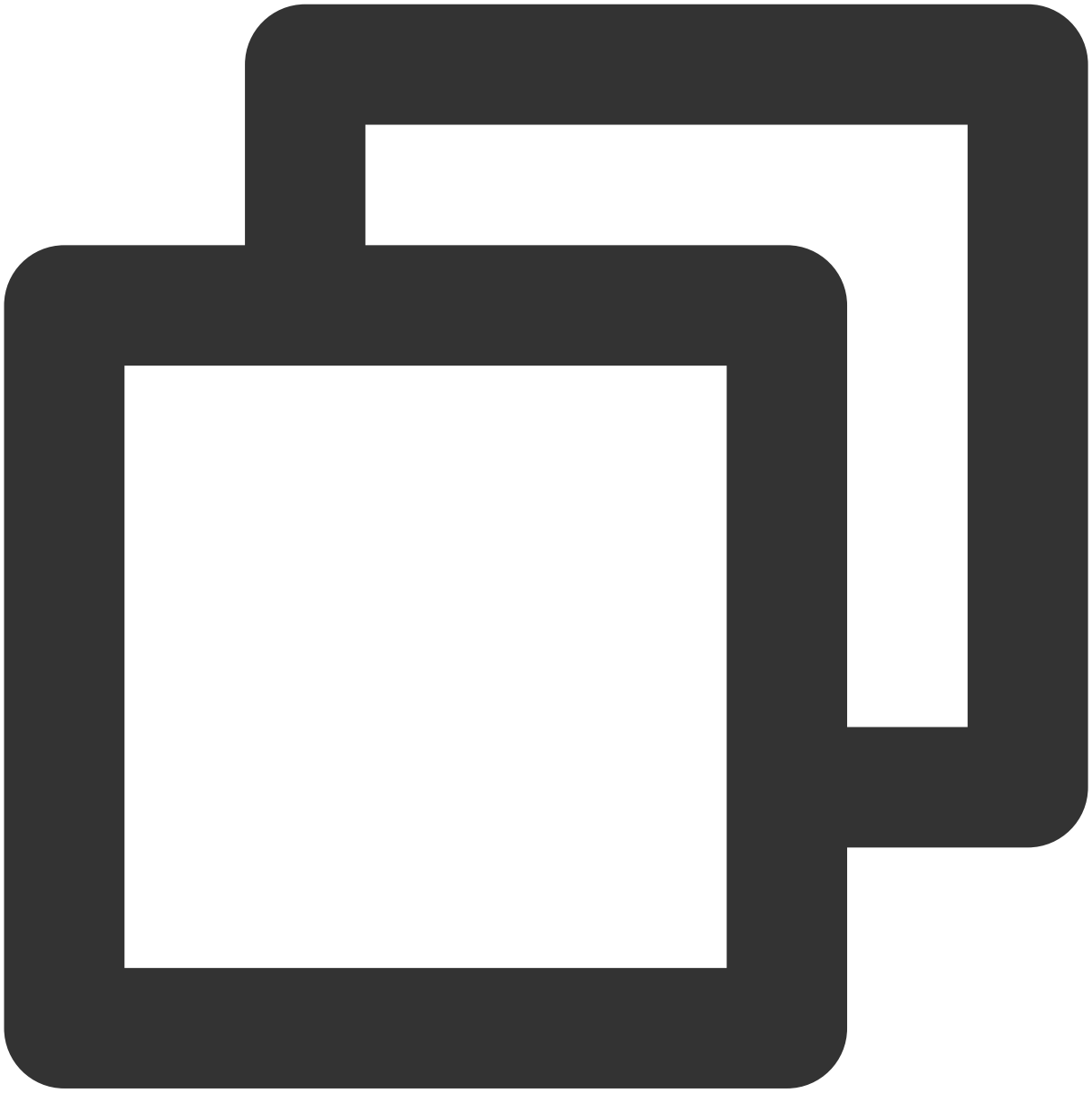
CROSS JOIN returns the Cartesian product (all combinations) of two relations.



```
SELECT *
FROM nation
CROSS JOIN region
```

LIMIT Clause

The LIMIT clause restricts the number of rows in the result set.

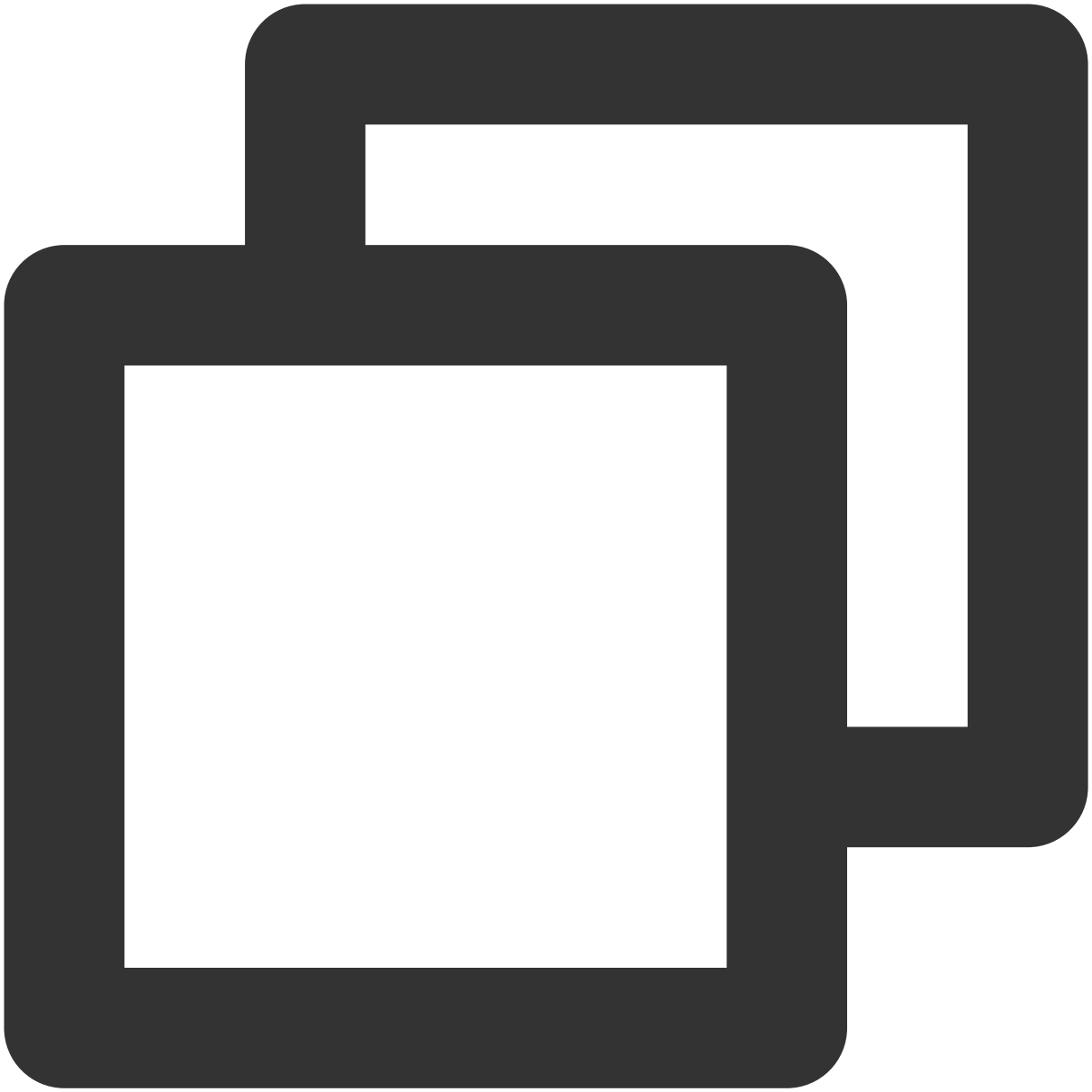


```
SELECT orderdate FROM orders LIMIT 5
```

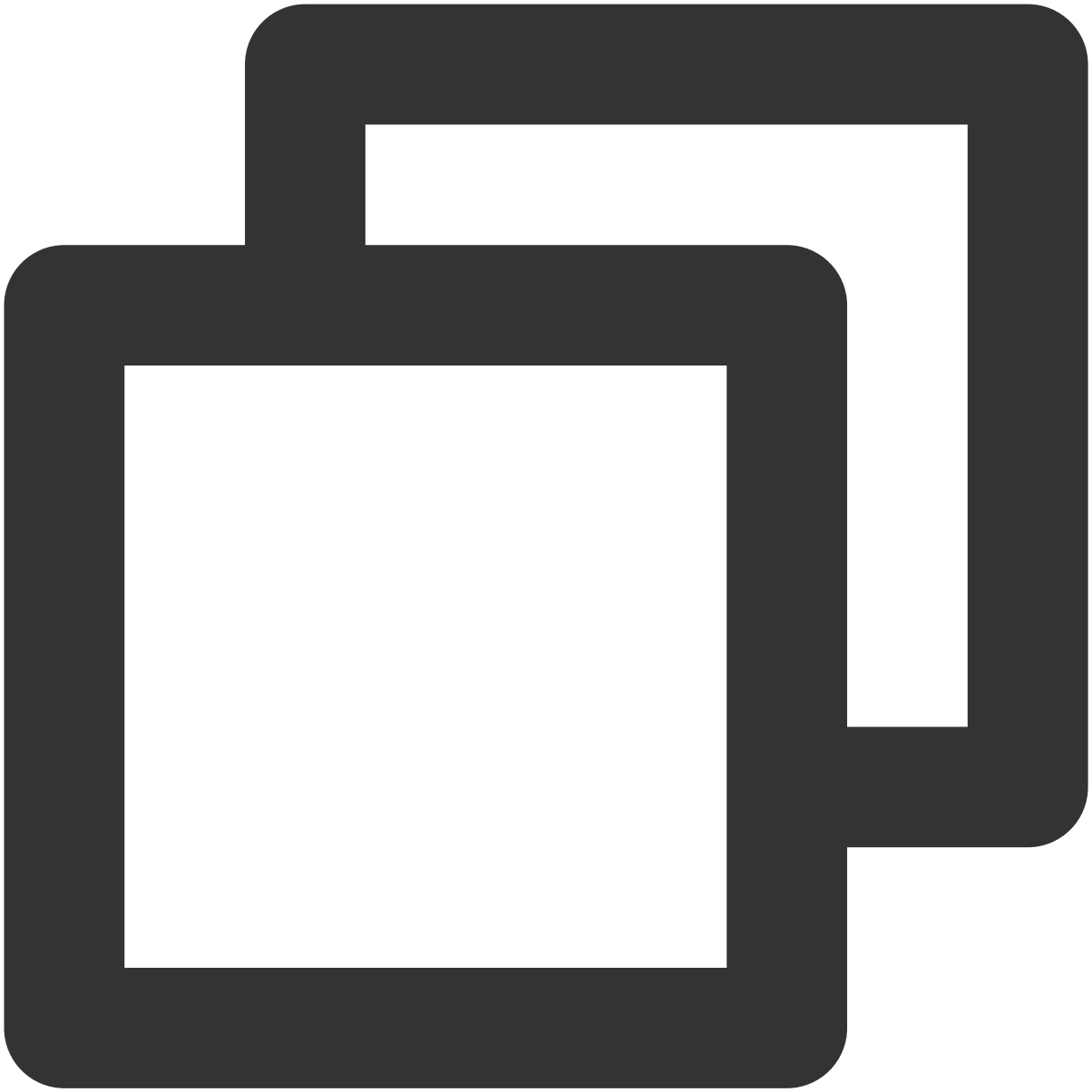
ORDER BY Clause

The ORDER BY clause sorts a result set by one or more output expressions

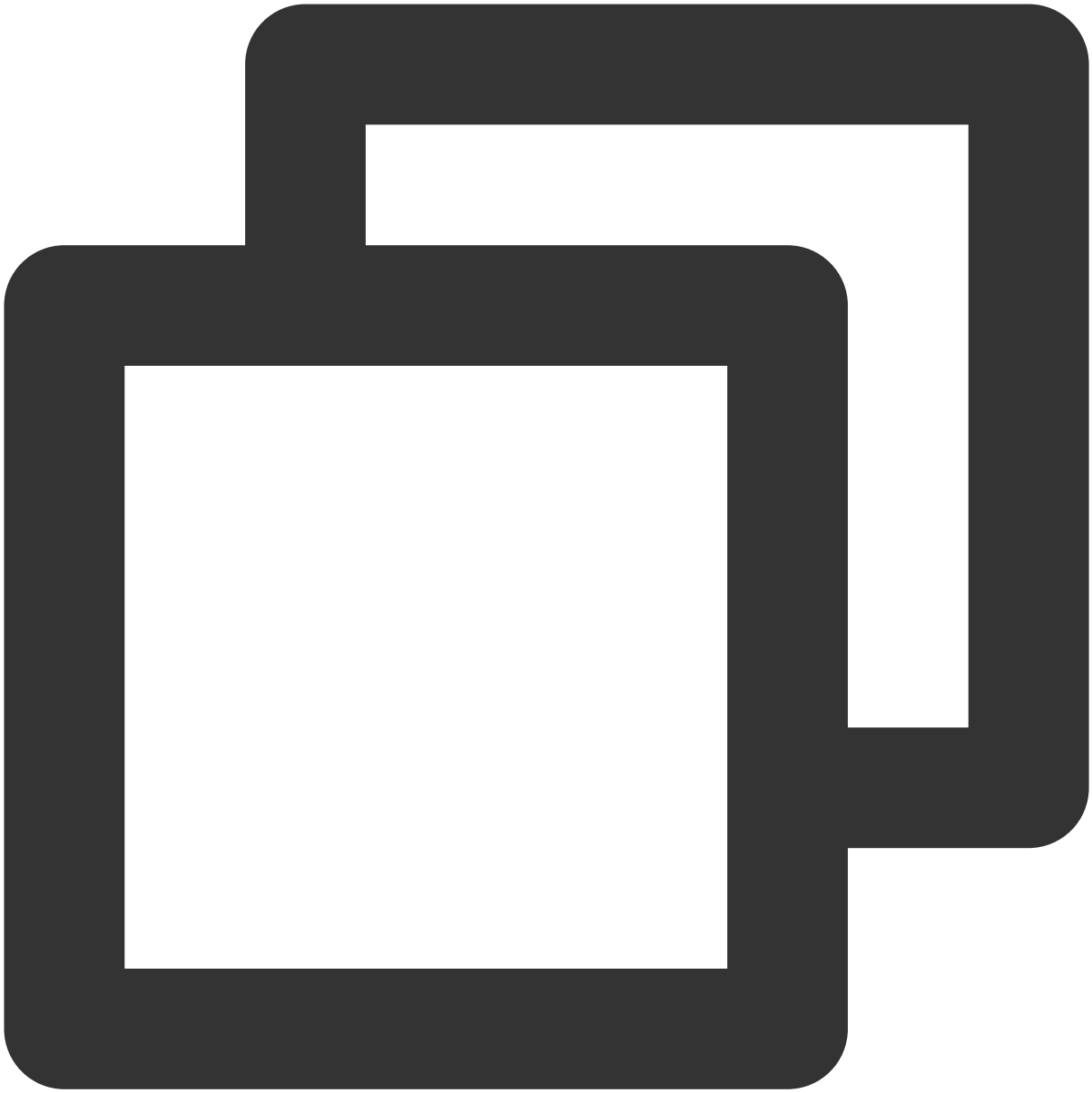
```
Statement: ORDER BY expression [ ASC | DESC ] [ NULLS { FIRST | LAST } ] [, ...]
```



```
SELECT name, age FROM person ORDER BY age
```



```
SELECT * FROM student
ORDER BY student_id
```

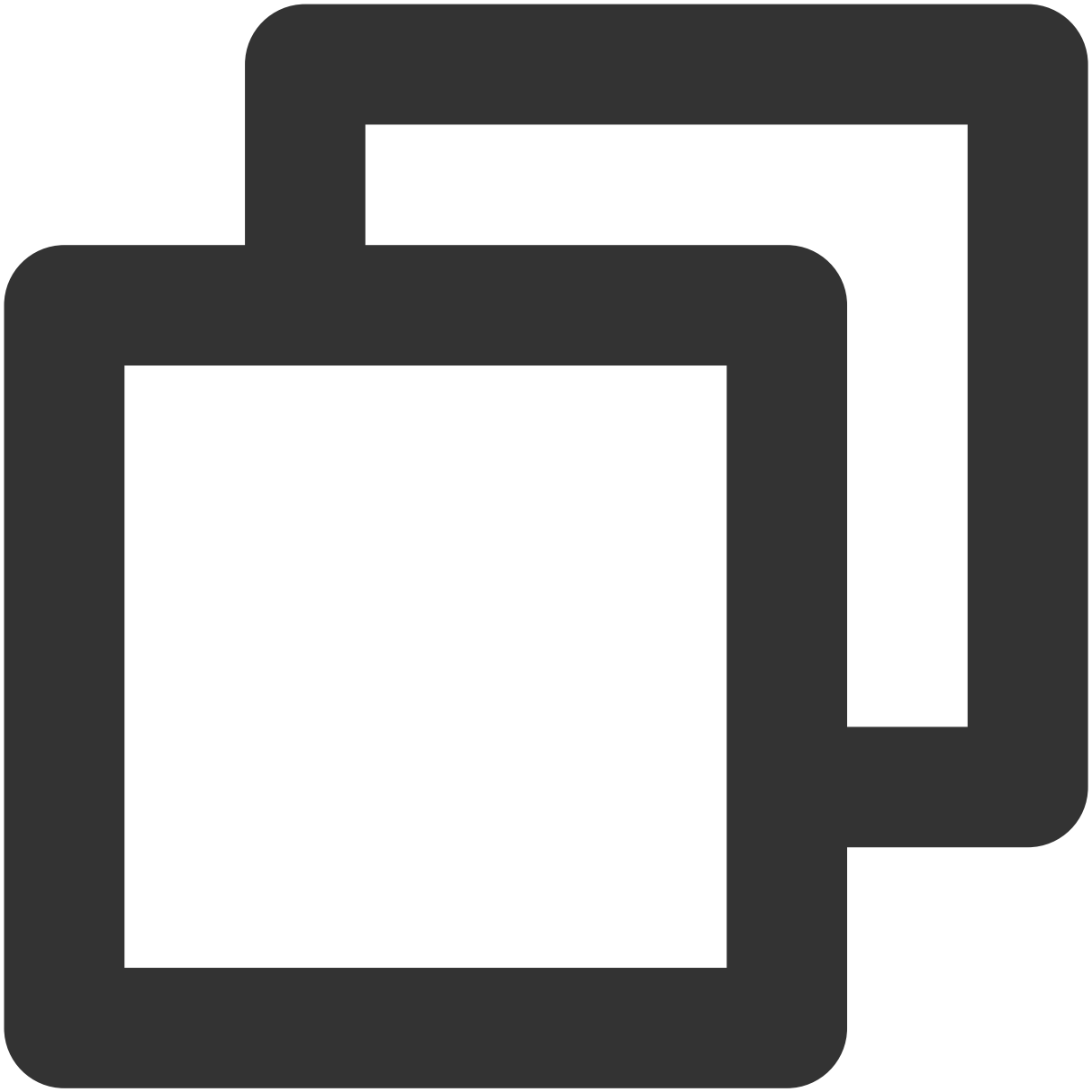


```
SELECT * FROM student
ORDER BY student_id,student_name
```

EXCEPT

The EXCEPT clause/operator combines two SELECT statements and returns the rows that are in the result set of the first SELECT statement, but not the second SELECT statement. This means that EXCEPT only returns rows and does not apply to the second SELECT statement.

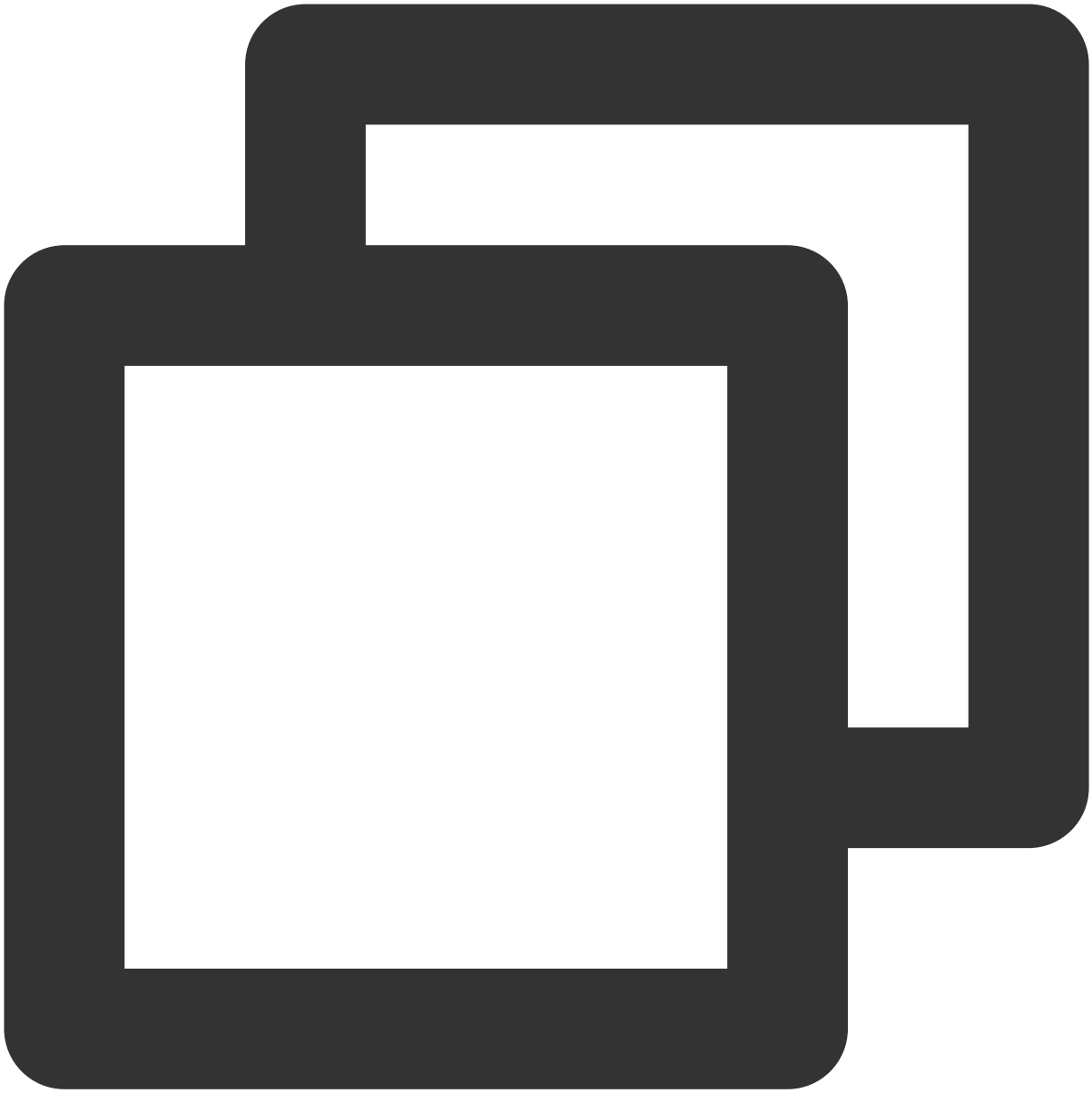
As with the UNION operator, the EXCEPT operator is better when the same rules are applied.



```
SELECT * FROM (VALUES 13, 42)
EXCEPT
SELECT 13
```

INTERSECT

INTERSET returns rows that exist in all result sets returned by two or more SELECT statements.

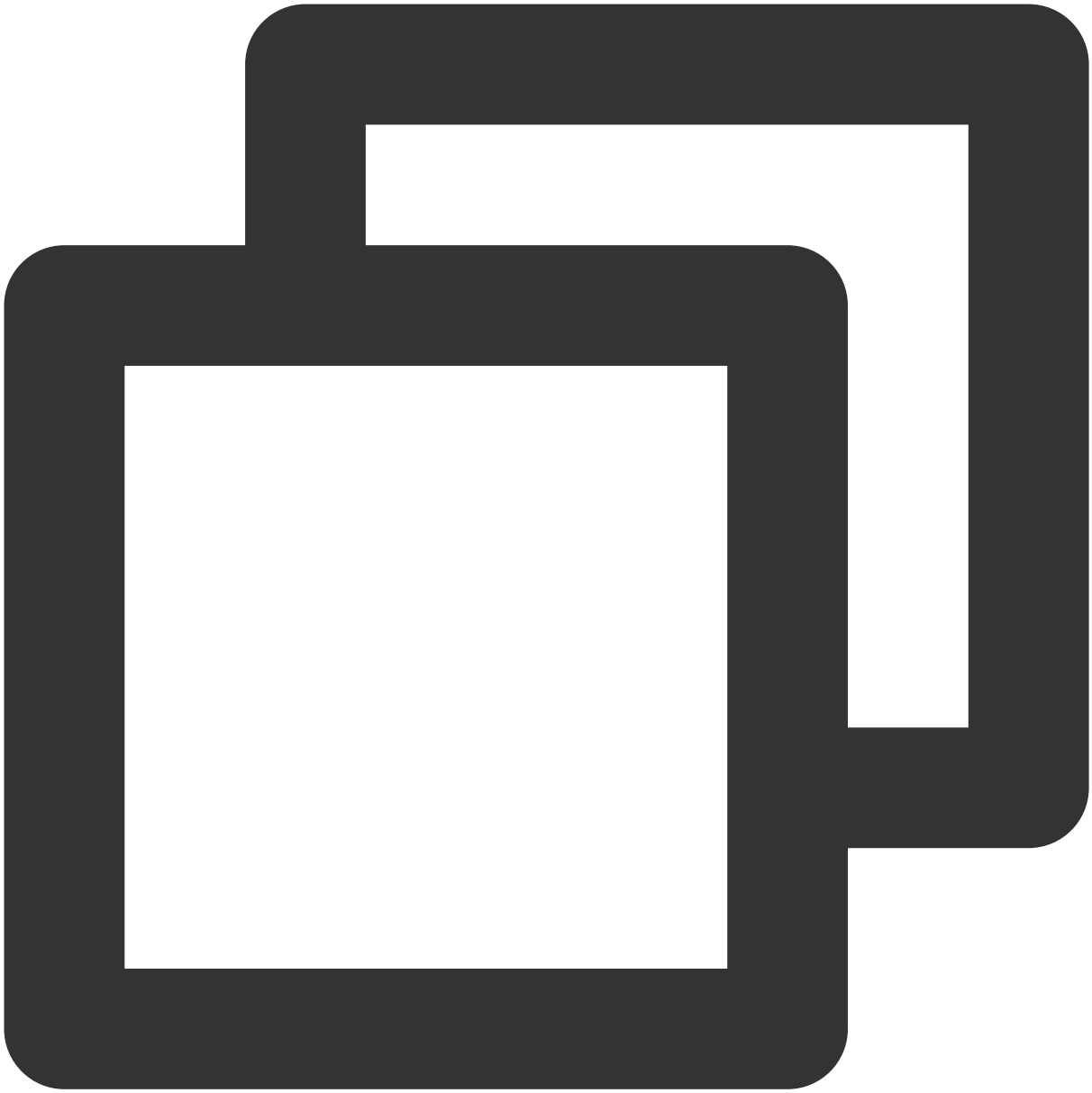


```
SELECT * FROM (VALUES 13, 42)
INTERSECT
SELECT 1
```

UNION

UNION combines the result sets of two or more SELECT statements into one result set. To retain duplicate rows in the result sets, use the `UNION ALL` operator.

```
SELECT 13
UNION
SELECT 42
```



```
SELECT id FROM a
UNION ALL
SELECT id FROM b;
```

TABLESAMPLE

`BERNOULLI` : Each row is selected to be in the table sample with a probability of the sample percentage. When a table is sampled using the Bernoulli method, all physical blocks of the table are scanned and certain rows are skipped (based on a comparison between the sample percentage and a random value calculated at runtime). The probability

of a row being included in the result is independent from any other row. This does not reduce the time required to read the sampled table from disk. It may have an impact on the total query time if the sampled output is processed further.

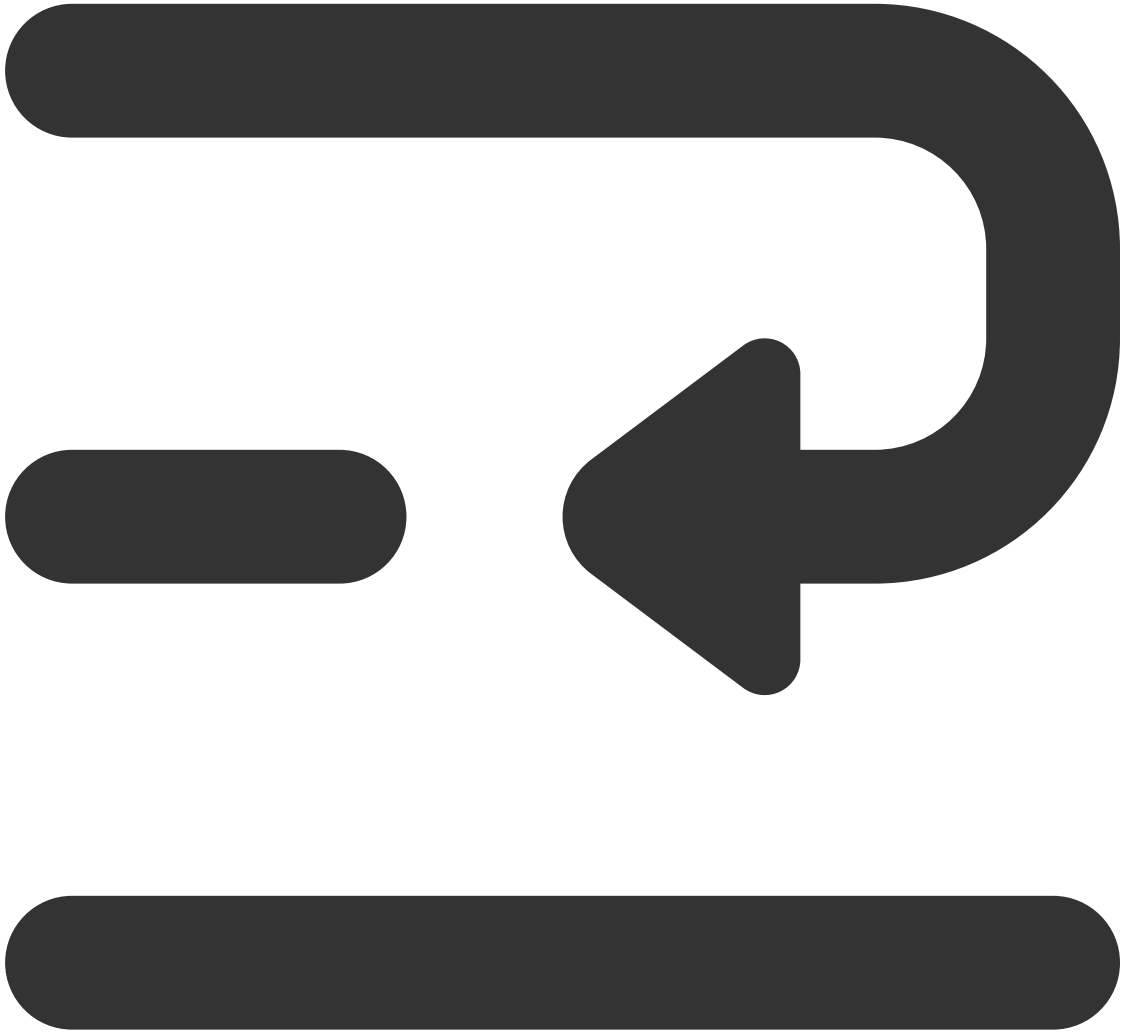
`SYSTEM` : This sampling method divides the table into logical segments of data and samples the table at this granularity. This sampling method either selects all rows from a specific data segment or skips it, based on a comparison between the sampling percentage and a randomly computed value at runtime. The row selected in system sampling depends on the connector used. For example, when Hive is used, it depends on how the data is laid out on HDFS. This method does not guarantee independent sampling probabilities.

```
SELECT *
FROM users TABLESAMPLE BERNOULLI (50);
```

```
SELECT *
FROM users TABLESAMPLE SYSTEM (75);
```

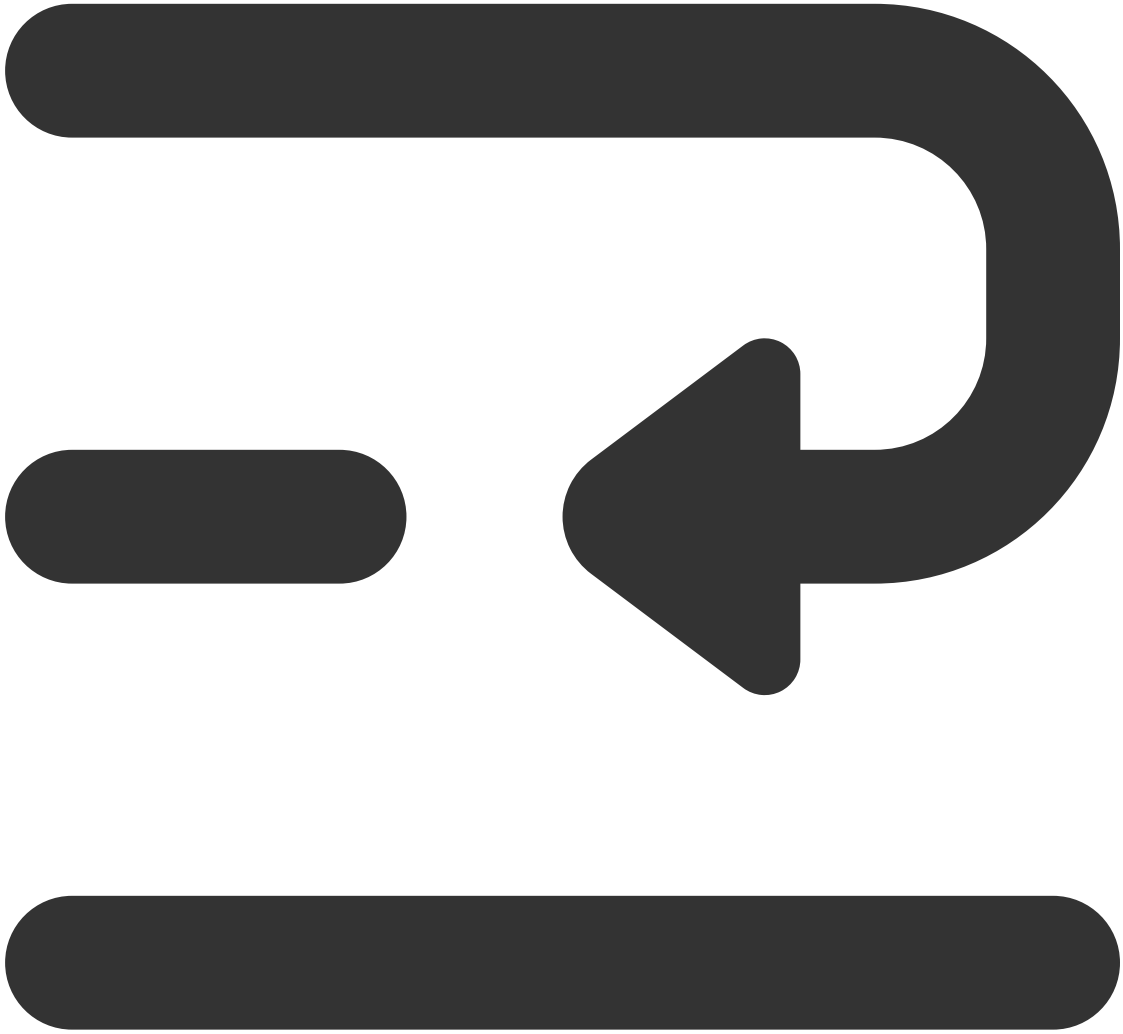
PIVOT Clause

Returns aggregate values based on specific columns.



```
SELECT * FROM person
    PIVOT (
      SUM(age) AS a, AVG(class) AS c
      FOR name IN ('John' AS john, 'Mike' AS mike)
     );
```

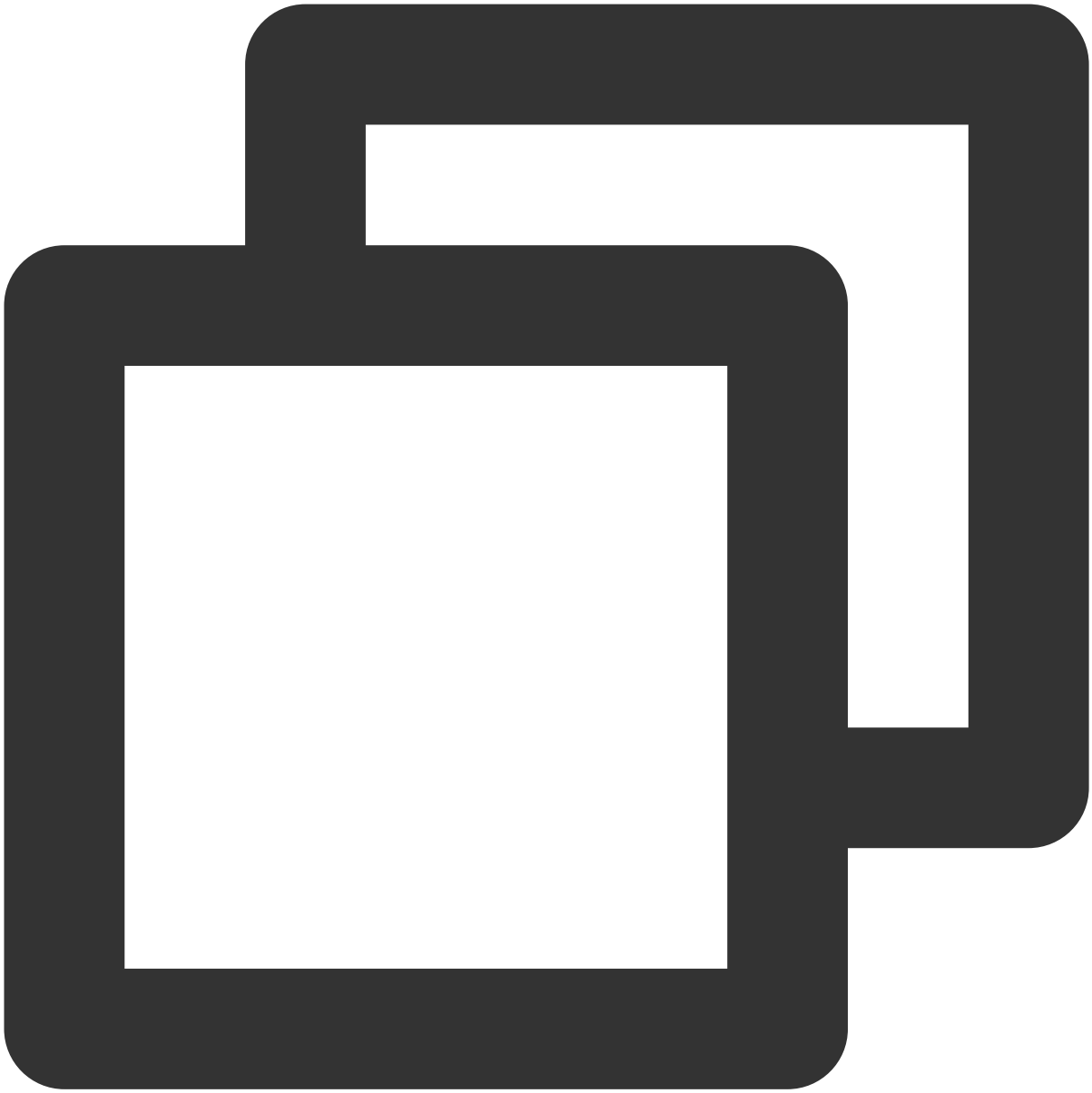
Lateral View Clause



```
LATERAL VIEW [ OUTER ] generator_function ( expression [ , ... ] ) [ table_alias ]
```

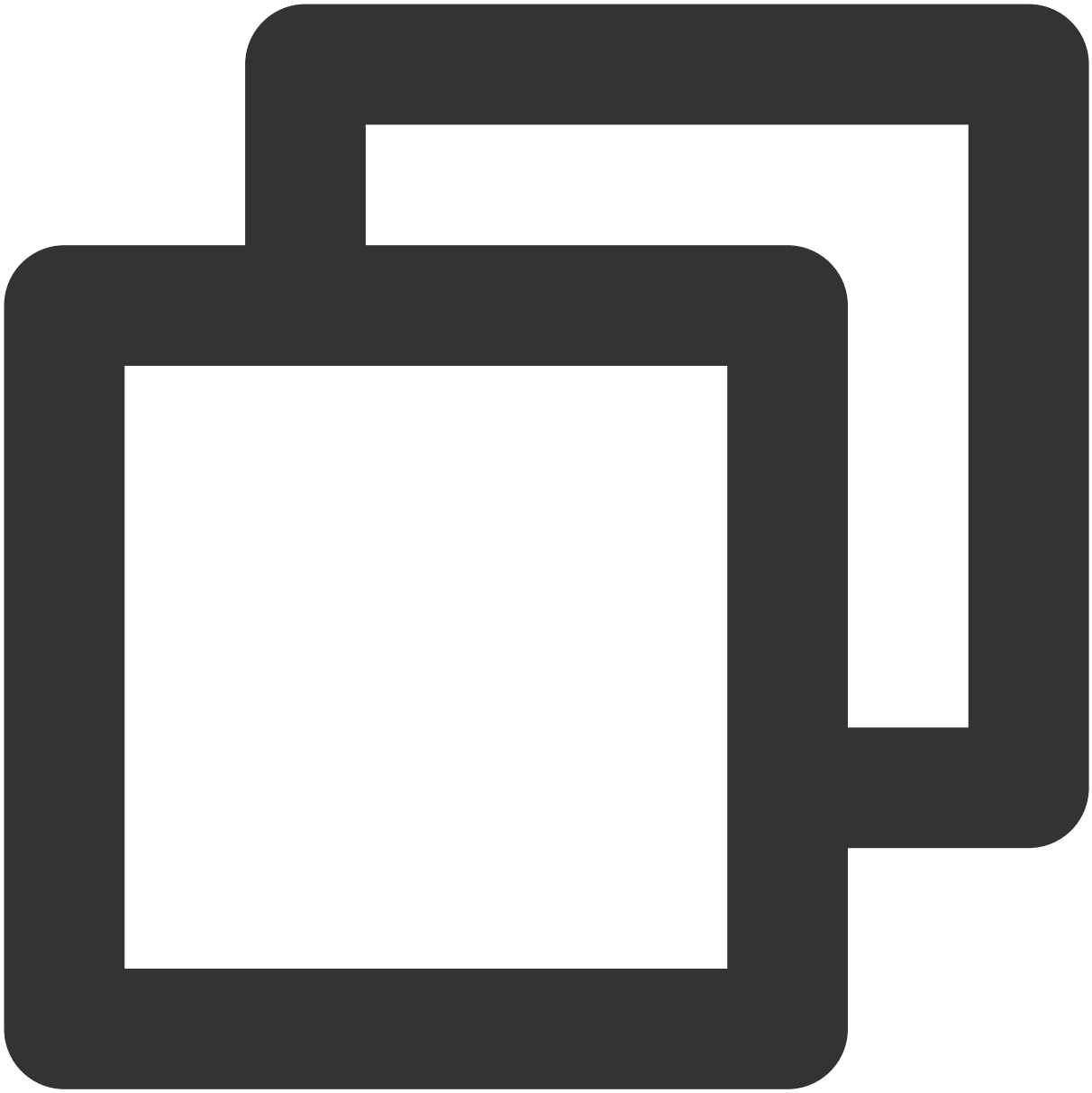
Escaping

To escape a single quote, precede it with another single quote, as shown in the following example:



```
Select 'dlc''test'
```

Specify escape characters or escape sequences when creating a table, for example, using the following method to create:



```
CREATE EXTERNAL TABLE IF NOT EXISTS `csv_test_2222` (
      `_c0`  STRING,
      `_c1`  INTEGER,
      `_c2`  INTEGER,
      `_c3`  INTEGER
    )
    ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde' WITH SERDEPROPERT
      'separatorChar' = '''',
      'quoteChar' = ''''
    )
    STORED AS `textfile`
```

```
    LOCATION 'cosn://dlc-nj-1258469122/csv/100M/
```

Iceberg Table Statement
DDL Statement

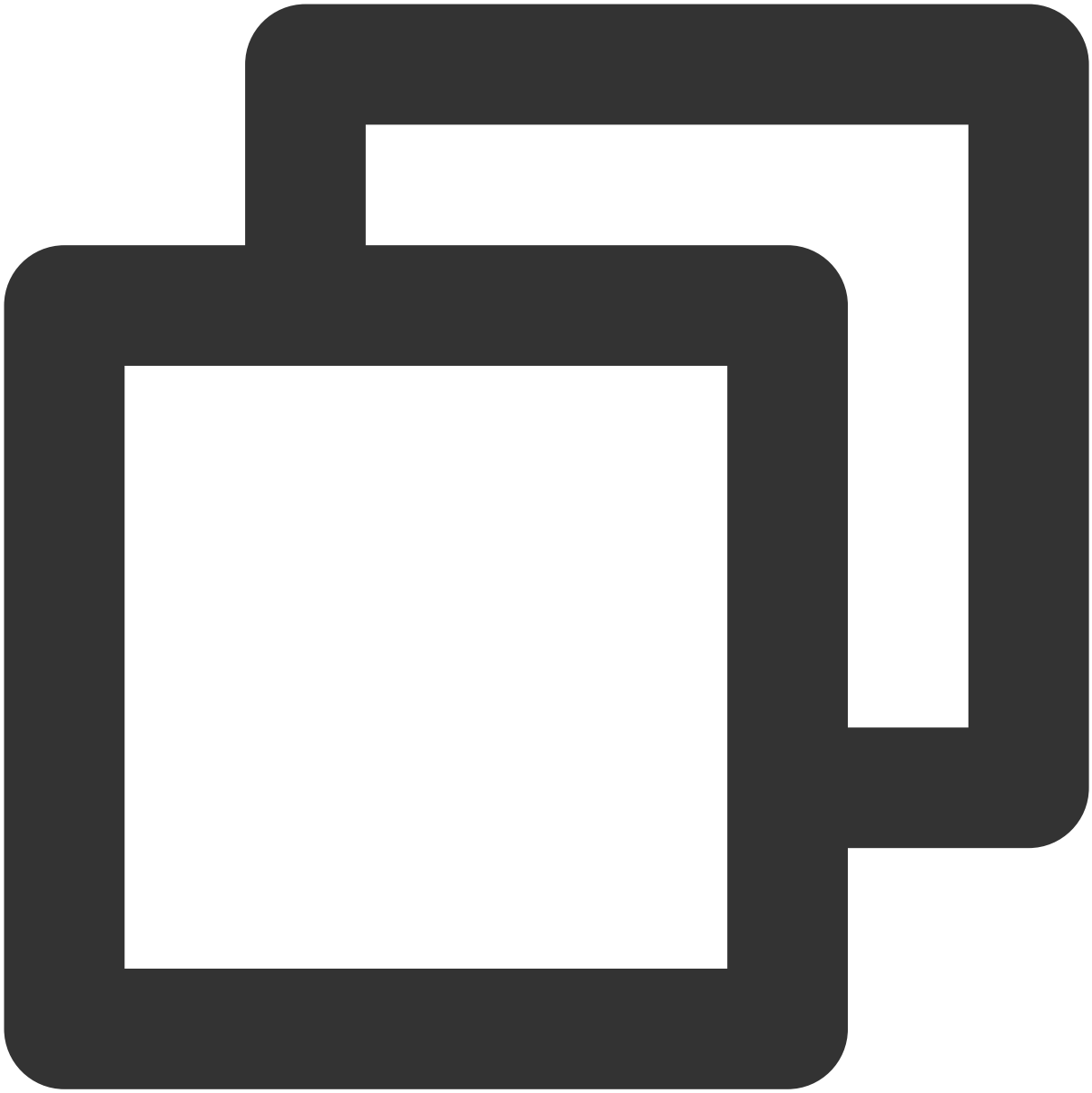
Last updated：2024-08-07 17:28:21

Description:

The following statement descriptions are for DLC native tables, which are Iceberg tables by default. If you are using Iceberg external tables, there will be detailed differences in DDL statement. For more information, see the documentation Differences between Iceberg External and Native Table Syntax.

CREATE TABLE

Statement



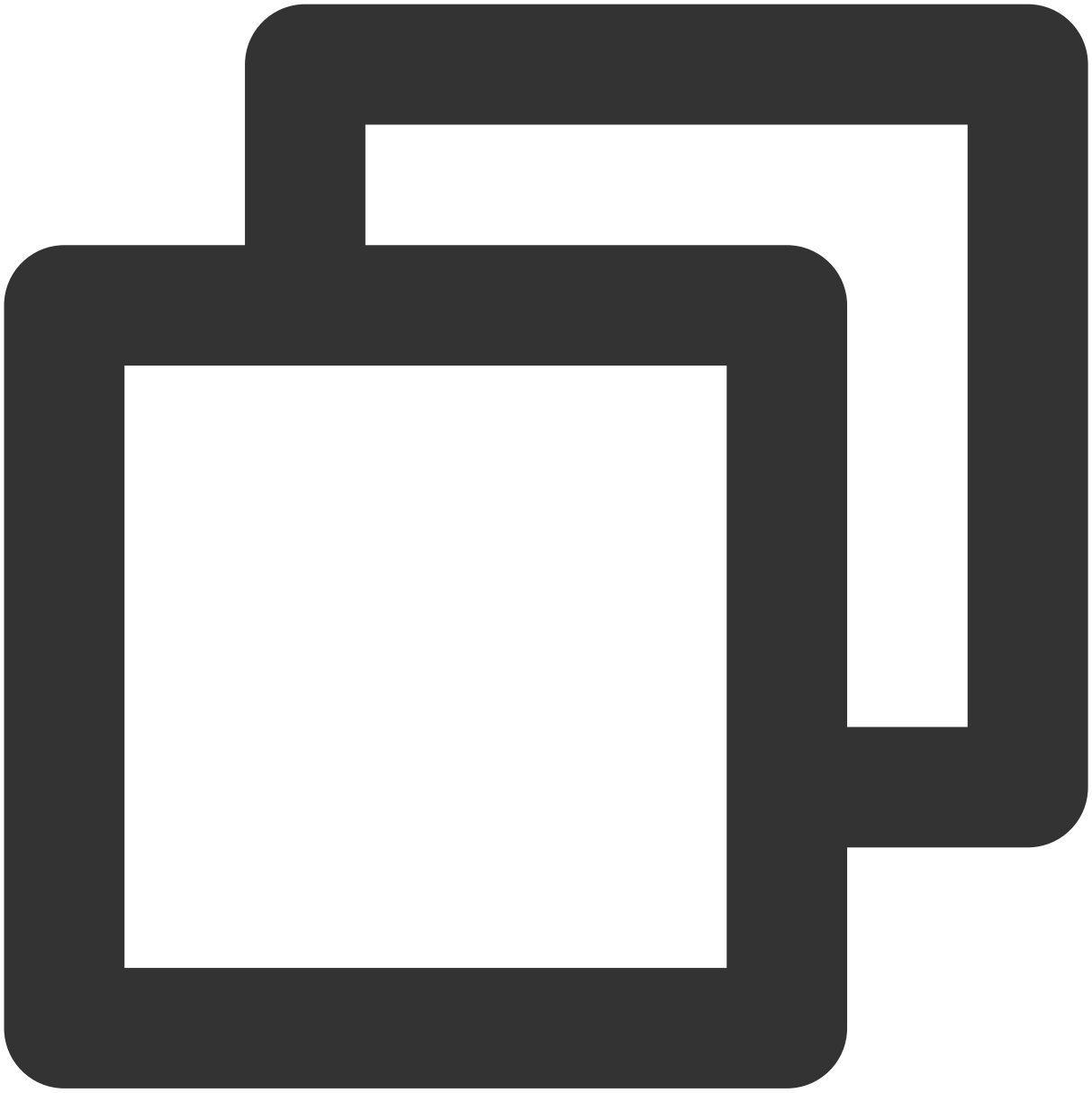
```
CREATE TABLE [ IF NOT EXISTS ] table_identifier
( col_name[:] col_type [ COMMENT col_comment ], ... )
[ COMMENT table_comment ]
[ PARTITIONED BY ( col_name1, transform(col_name2), ... ) ]
```

Parameter

table_identifier

`table_identifier` supports a three-part format: `catalog.db.name` .

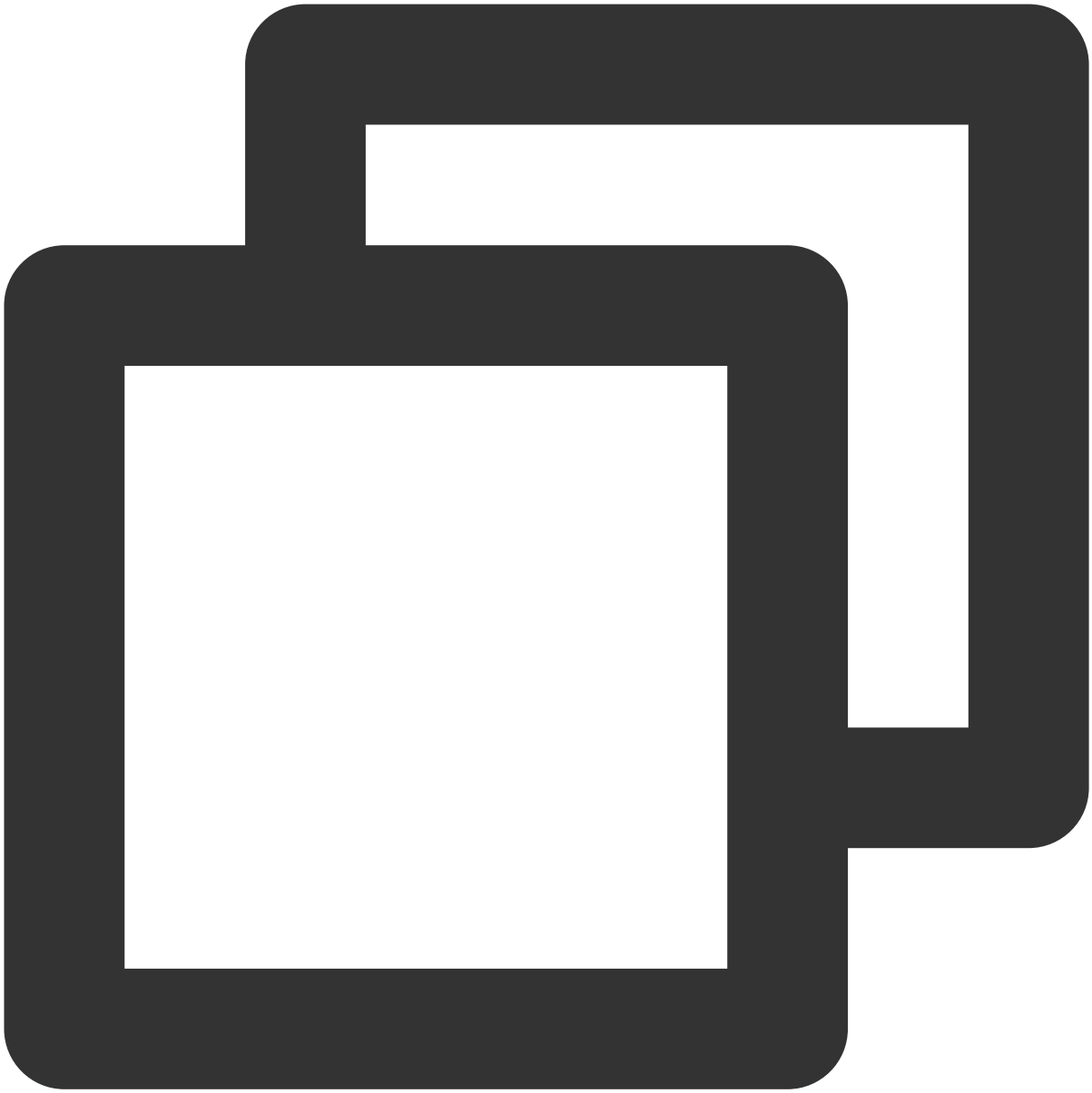
Schemas and Data Types



```
col_type
: primitive_type
  | nested_type

primitive_type
: boolean
| int/integer
| long/bigint
| float
```

```
| double
| decimal(p,s), p = maximum number of digits, s = maximum number of decimal places,
| date
| timestamp, timestamp with timezone. Does not support time and without timezone.
| string, can also correspond to Iceberg uuid type.
| binary, can also correspond to Iceberg fixed type.

nested_type
: struct
| list
| map
```

Partition Transforms



```
transform
: identity, supports any type, DLC does not support this transformation.
| bucket[N], hash mod N bucketing, supports col_type: int,long, decimal, date, time
| truncate[L], L-truncation bucketing, supports col_type: int,long,decimal,string
| years, year, supports col_type: date,timestamp
| months, month, supports col_type: date,timestamp
| days/date, date, supports col_type: date,timestamp
| hours/date_hour, hour, supports col_type: timestamp
```

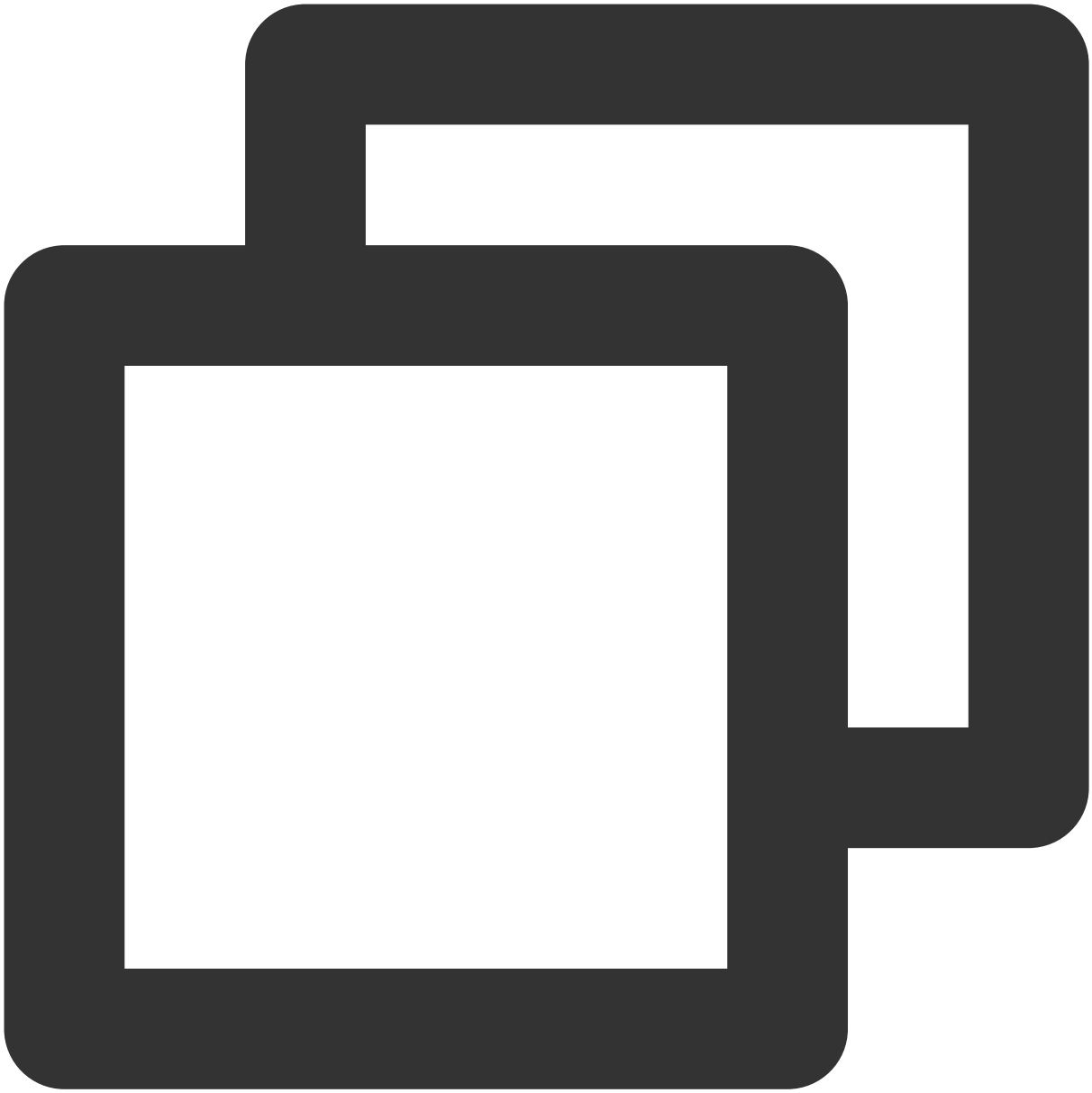
Example

```
CREATE TABLE dempts(
    id bigint COMMENT 'id number',
    num int,
    eno float,
    dno double,
    cno decimal(9,3),
    flag  boolean,
    data string,
    ts_year timestamp,
    date_month date,
    bno binary,
```

```
    point struct<x: double, y: double>,
    points array<struct<x: double, y: double>>,
    pointmaps map<struct<x: int>, struct<a: int>>
)
COMMENT 'table documentation'
PARTITIONED BY (bucket(16,id), years(ts_year), months(date_month), identity(bno), b
```

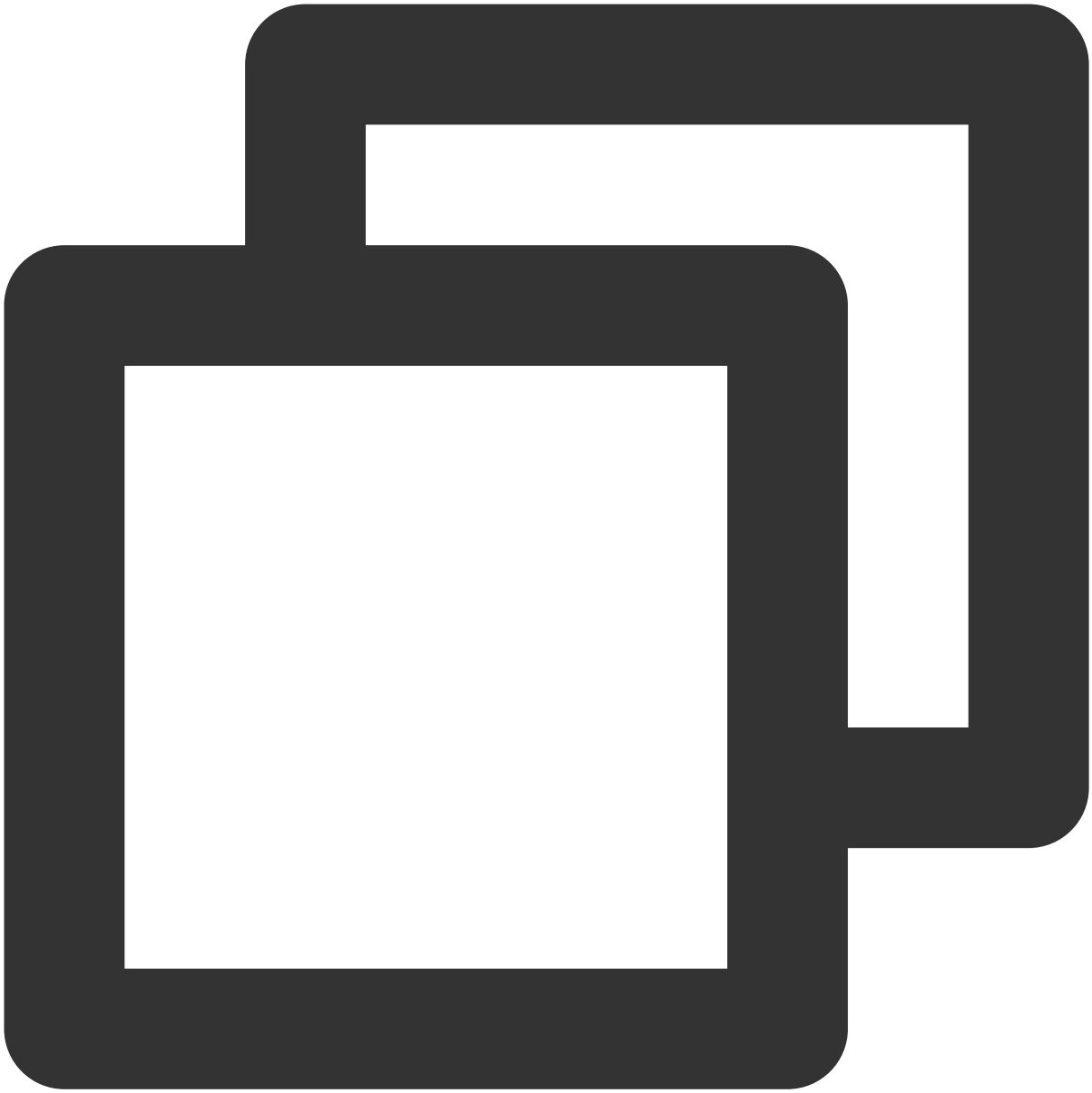
CREATE TABLE AS SELECT

Statement



```
CREATE TABLE [ IF NOT EXISTS ] table_identifier
    [ COMMENT table_comment ]
    [ PARTITIONED BY ( col_name1, transform(col_name2), ... ) ]
    [ TBLPROPERTIES ( property_name=property_value, ... ) ]
AS select_statement
```

Example

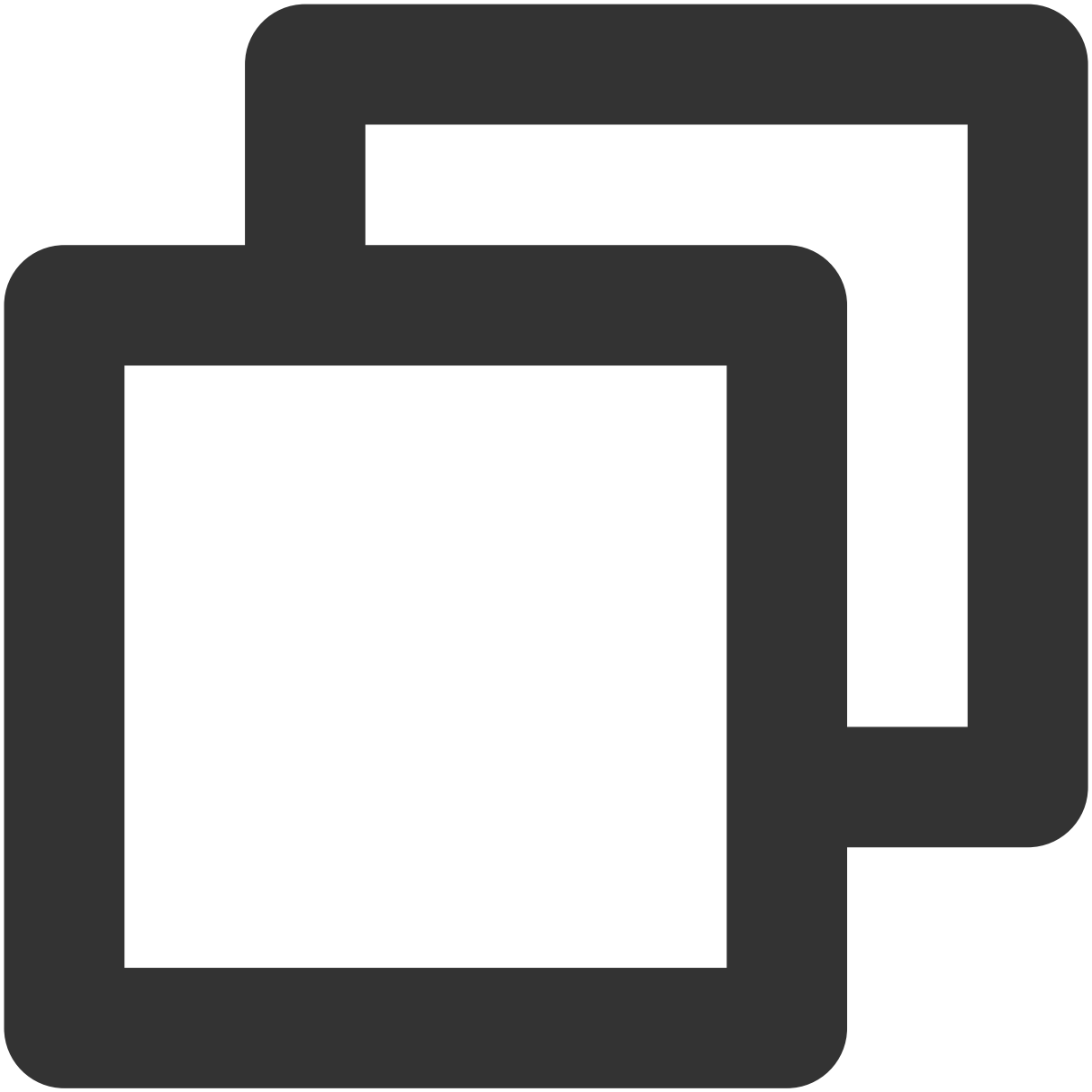


```
CREATE TABLE dempts_copy
COMMENT 'table create as select'
PARTITIONED BY (eno, dno)
AS SELECT * from dempts;
```

REPLACE TABLE AS SELECT

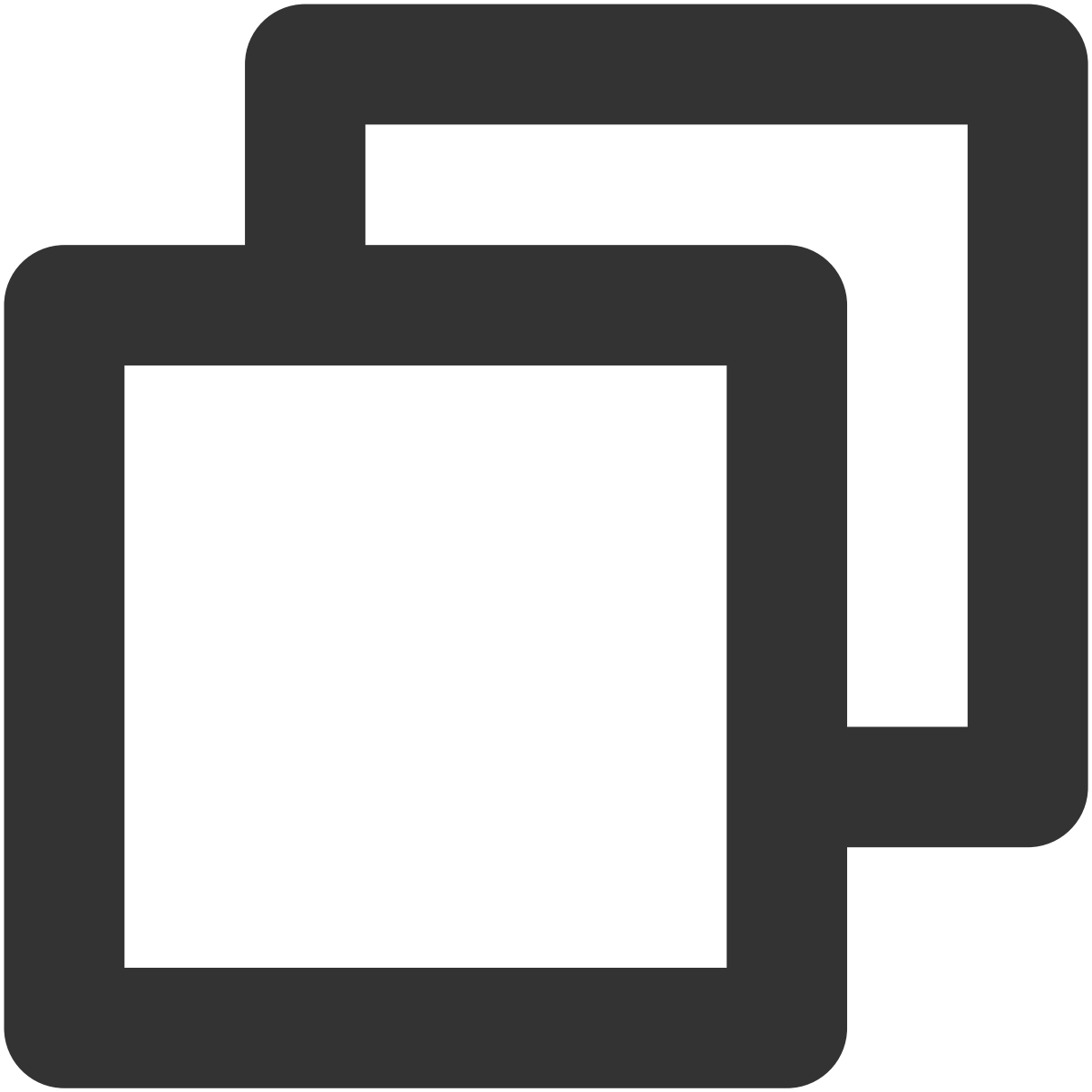
Generate a snapshot from the results of a SELECT query to update the table while retaining the table's history.

Statement



```
CREATE [OR REPLACE] TABLE table_identifier
    [ COMMENT table_comment ]
    [ PARTITIONED BY ( col_name1, transform(col_name2), ... ) ]
AS select_statement
```

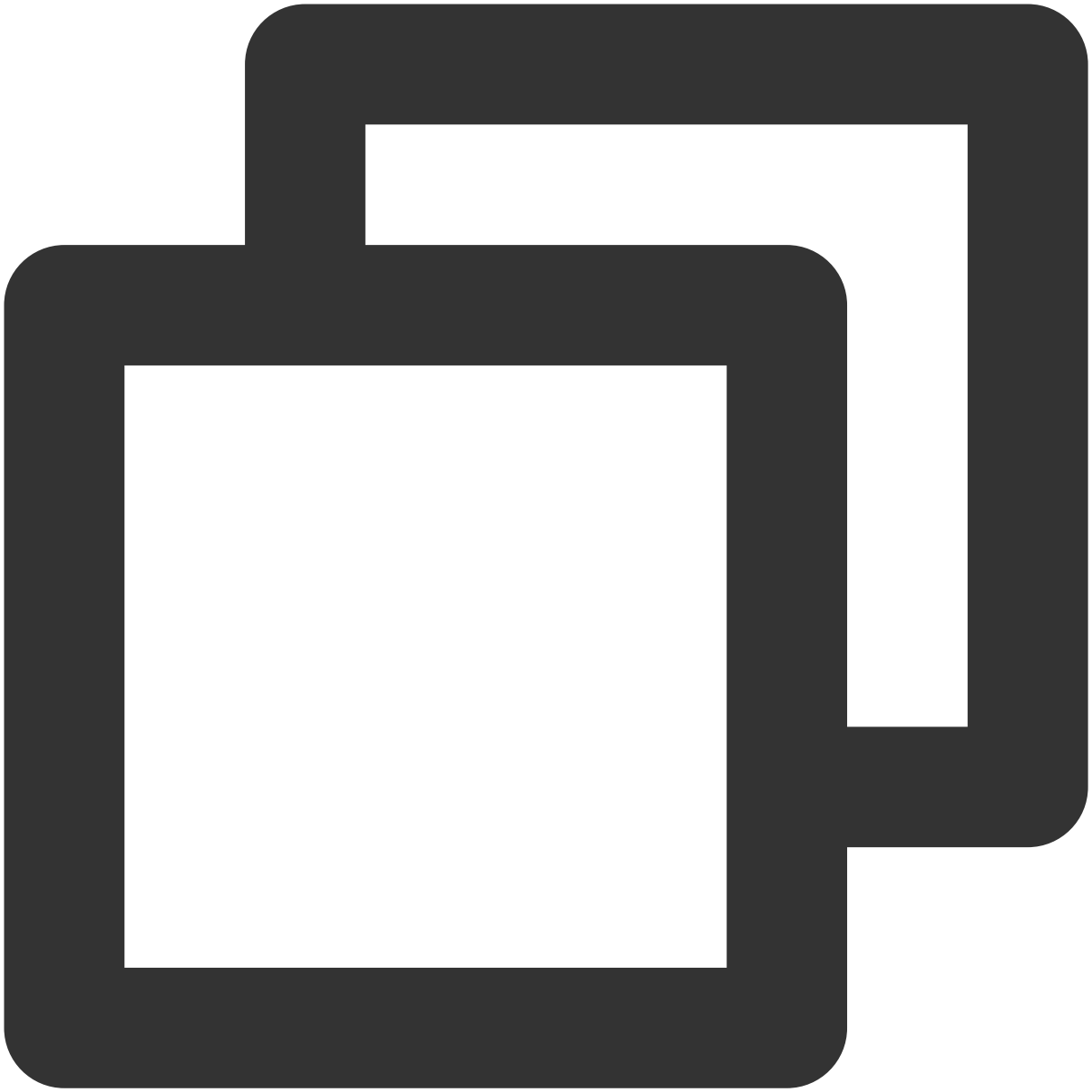
Example



```
CREATE OR REPLACE TABLE dempts_replace
    COMMENT 'table create as replace'
    PARTITIONED BY (eno, dno)
AS SELECT * from dempts;
```

DROP TABLE

Statement



```
DROP TABLE [ IF EXISTS ] table_identifier
```

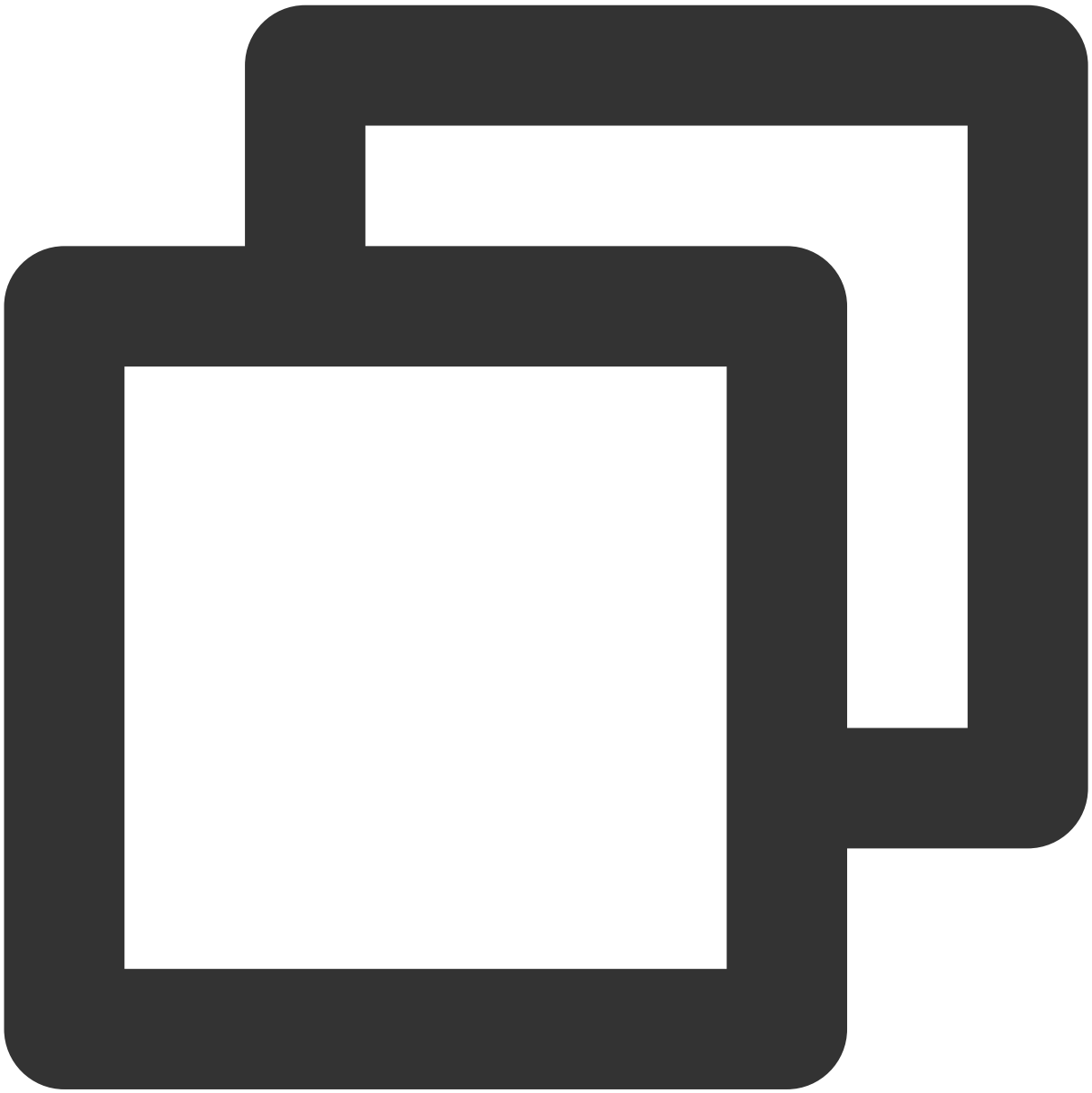
ALTER TABLE

Alter table statement.

ALTER TABLE ... RENAME TO

Change table name.

Statement



```
ALTER TABLE table_identifier RENAME TO new_table_identifier
```

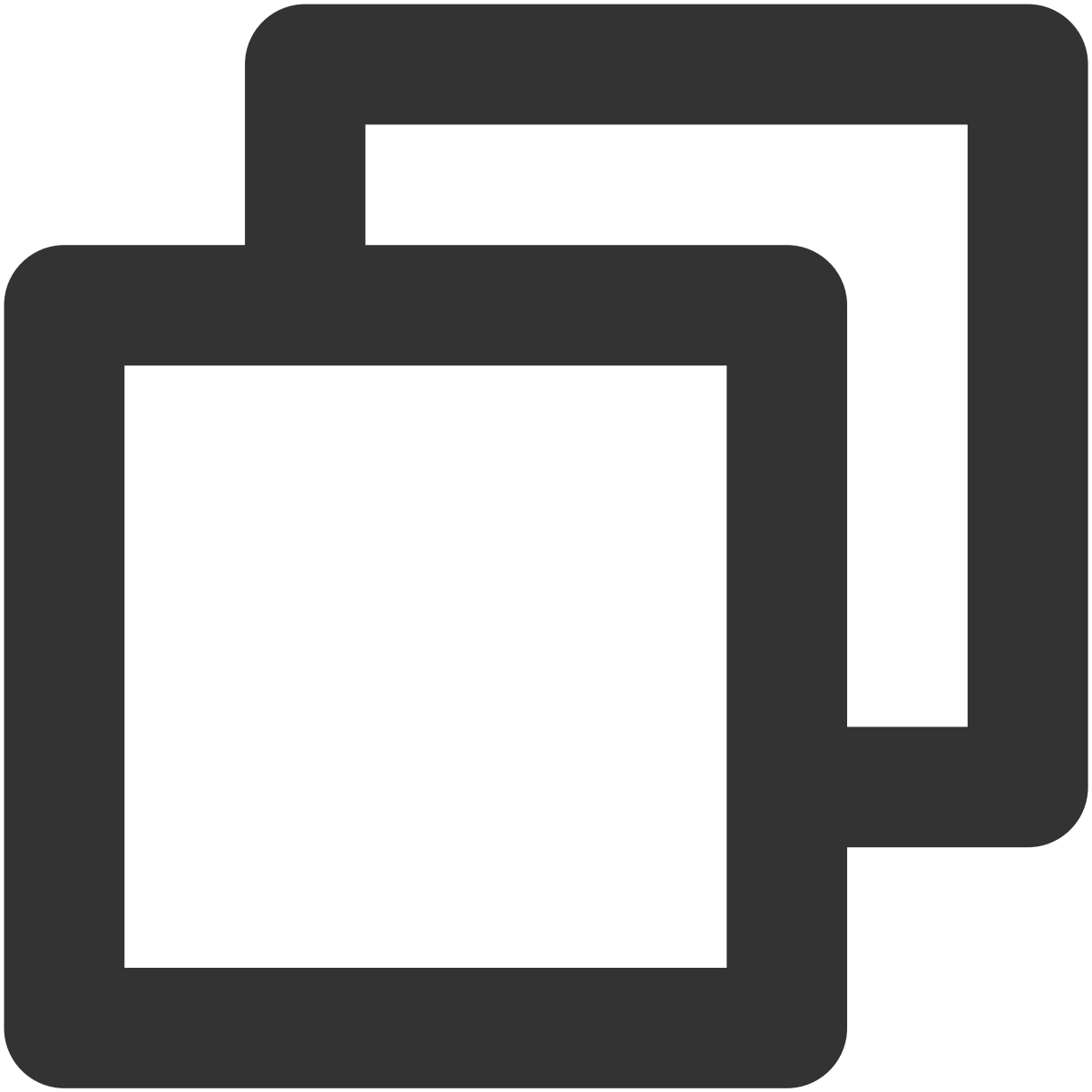
ALTER TABLE ... SET / UNSET TBLPROPERTIES

Update/delete data table attributes.

Statement

```
-- Update attribute configuration through SET.
ALTER TABLE table_identifier
SET TBLPROPERTIES (property_name=property_value, ...)

-- Delete attribute configuration through UNSET.
ALTER TABLE table_identifier
UNSET TBLPROPERTIES (property_name, ...)
```

Example

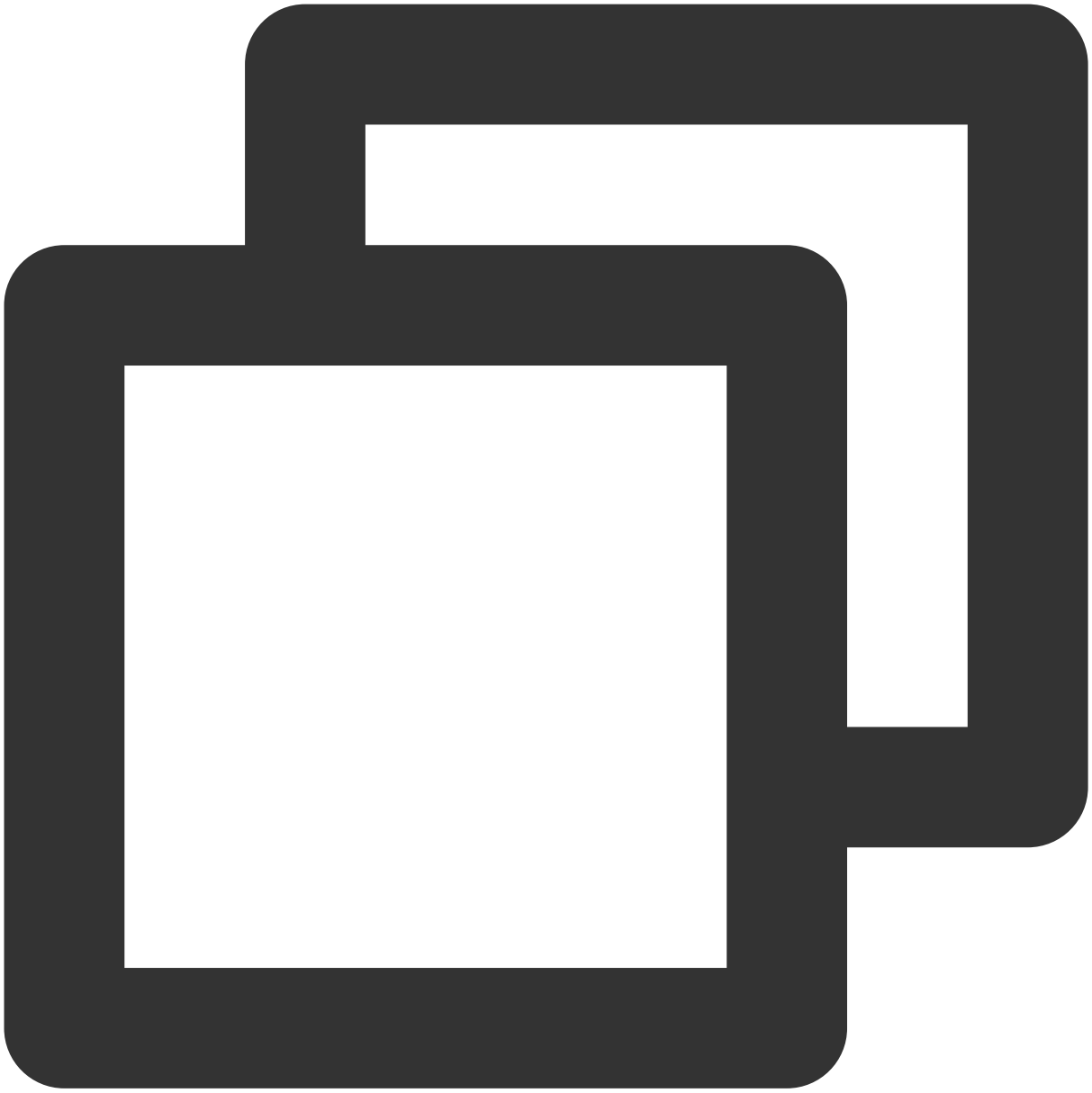


```
-- Update attribute configuration through SET.
ALTER TABLE dempts SET TBLPROPERTIES ('read.split.target-size'='268435456');

-- Delete attribute configuration through UNSET.
ALTER TABLE dempts UNSET TBLPROPERTIES ('read.split.target-size'='268435456');
```

ALTER TABLE ... WRITE ORDERED BY

Set the sort order of the table data during insertion.

Statement



```
ALTER TABLE table_identifier
WRITE [LOCALLY] ORDERED BY
{col_name [ASC|DESC] [NULLS FIRST|LAST]}[, ...]
```

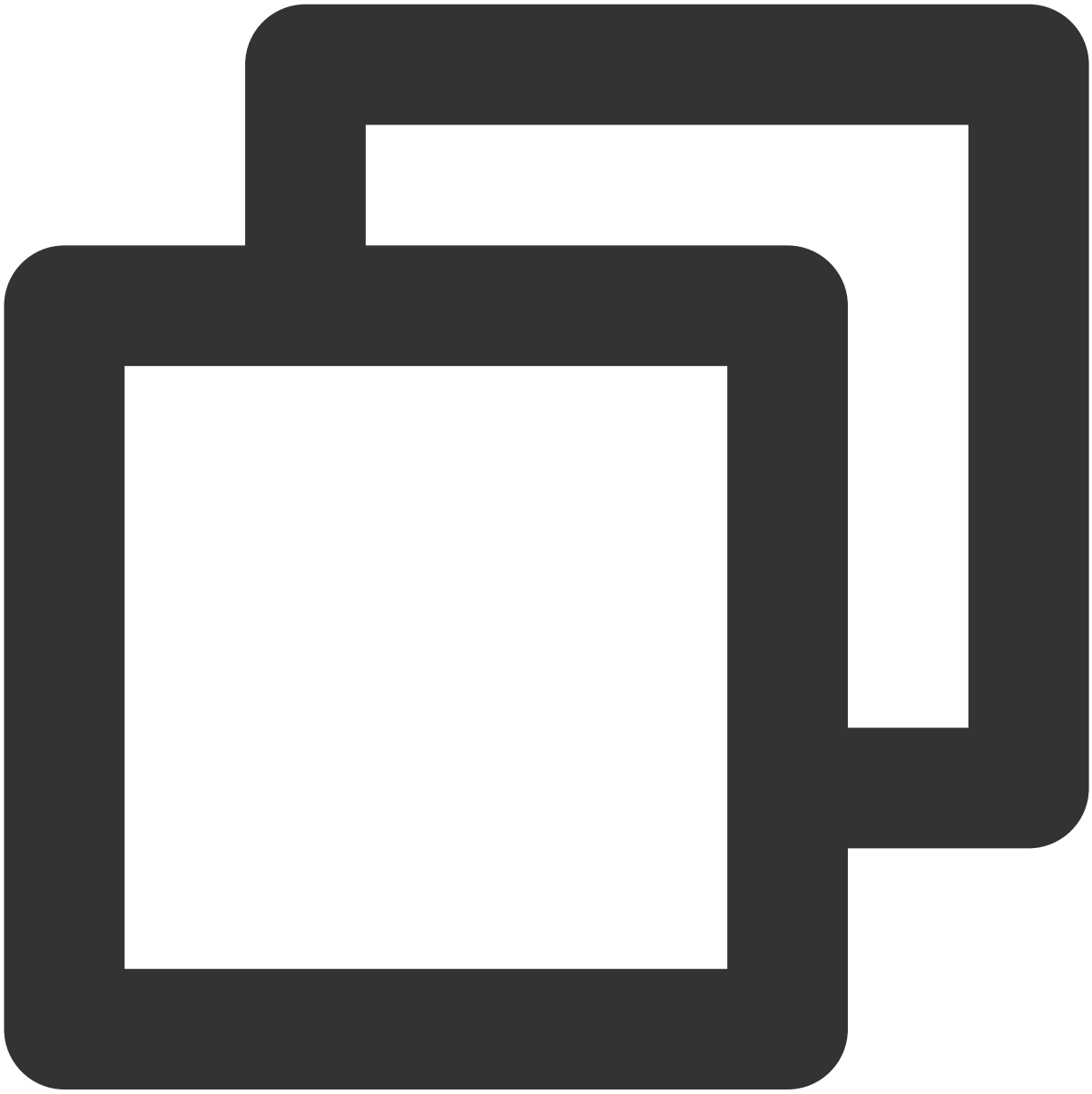
Example

```
ALTER TABLE dempts WRITE ORDERED BY category, id;

-- use optional ASC/DEC keyword to specify sort order of each field (default ASC)
ALTER TABLE dempts WRITE ORDERED BY category ASC, id DESC;

-- use optional NULLS FIRST/NULLS LAST keyword to specify null order of each field
ALTER TABLE dempts WRITE ORDERED BY category ASC NULLS LAST, id DESC NULLS FIRST;

-- To order within each task, not across tasks
ALTER TABLE dempts WRITE LOCALLY ORDERED BY category, id;
```

ALTER TABLE ... WRITE DISTRIBUTED BY PARTITION

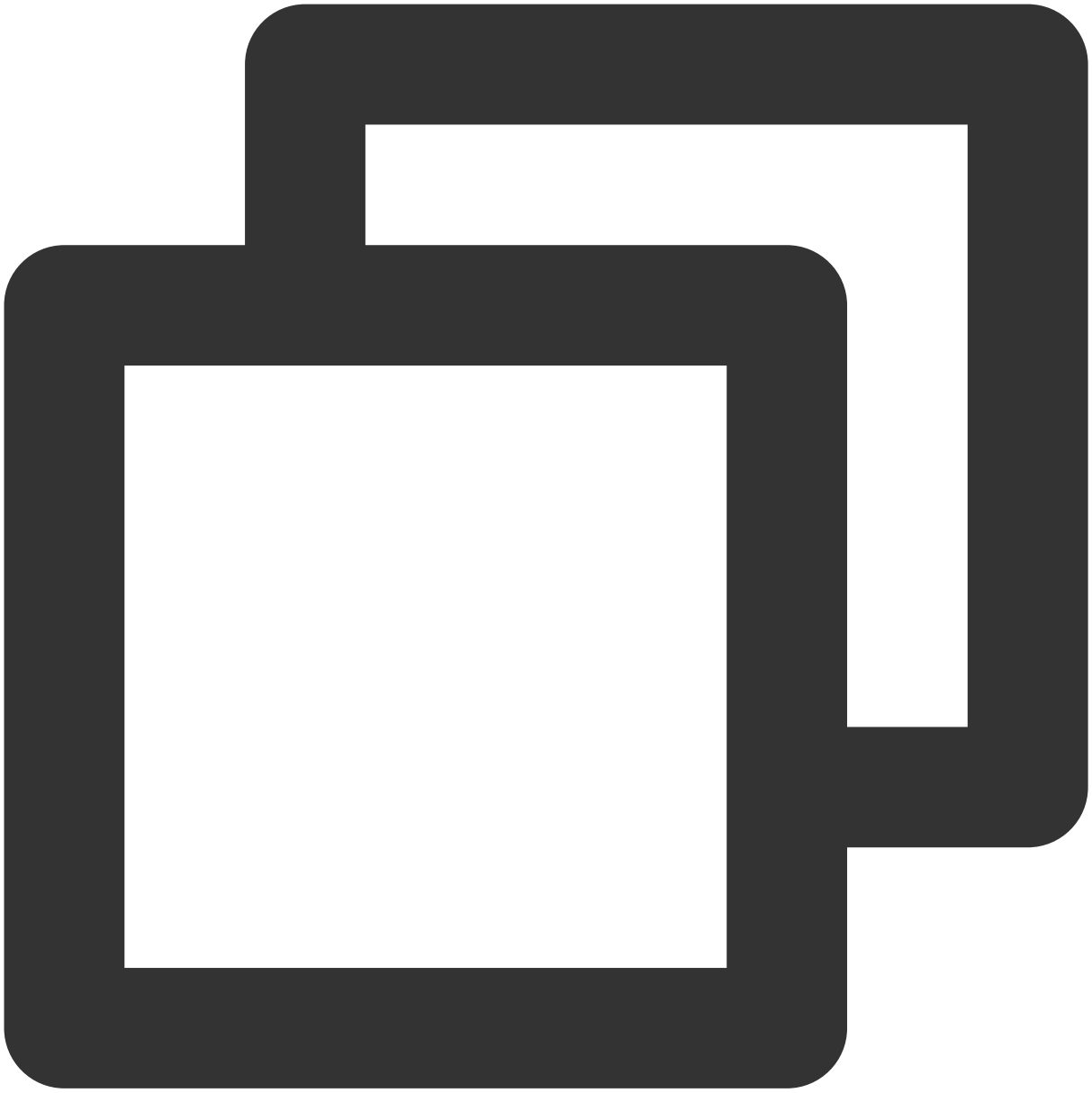
Modify the data distribution policy of partitioned tables.

Statement



```
ALTER TABLE table_identifier
WRITE DISTRIBUTED BY PARTITION
[ LOCALLY ORDERED BY
{col_name [ASC|DESC] [NULLS FIRST|LAST]}[, ...]]
```

Example



```
ALTER TABLE dempts WRITE DISTRIBUTED BY PARTITION;
ALTER TABLE dempts WRITE DISTRIBUTED BY PARTITION LOCALLY ORDERED BY id;
```

ALTER TABLE COLUMNS

Change field statement.

ALTER TABLE ... ADD COLUMNS

Add multiple fields.

Statement

```
-- Add multiple fields.
ALTER TABLE table_identifier
ADD COLUMNS (col_name col_type [COMMENT col_comment], ...)

-- Add a single field.
```

```
ALTER TABLE table_identifier
ADD COLUMN col_name col_type [COMMENT col_comment]
[FIRST | AFTER target_col_name]
```

Example

```
-- Add multiple fields.
ALTER TABLE dempts
ADD COLUMNS (
    new_column_1 string comment 'new_column_1 docs',
    new_column_2 int comment 'new_column_2 docs'
```

```
);

-- Add a single field.
ALTER TABLE dempts
ADD COLUMN new_column_3 string comment 'new_column docs';
```

ALTER TABLE ... RENAME COLUMN

Change field name.

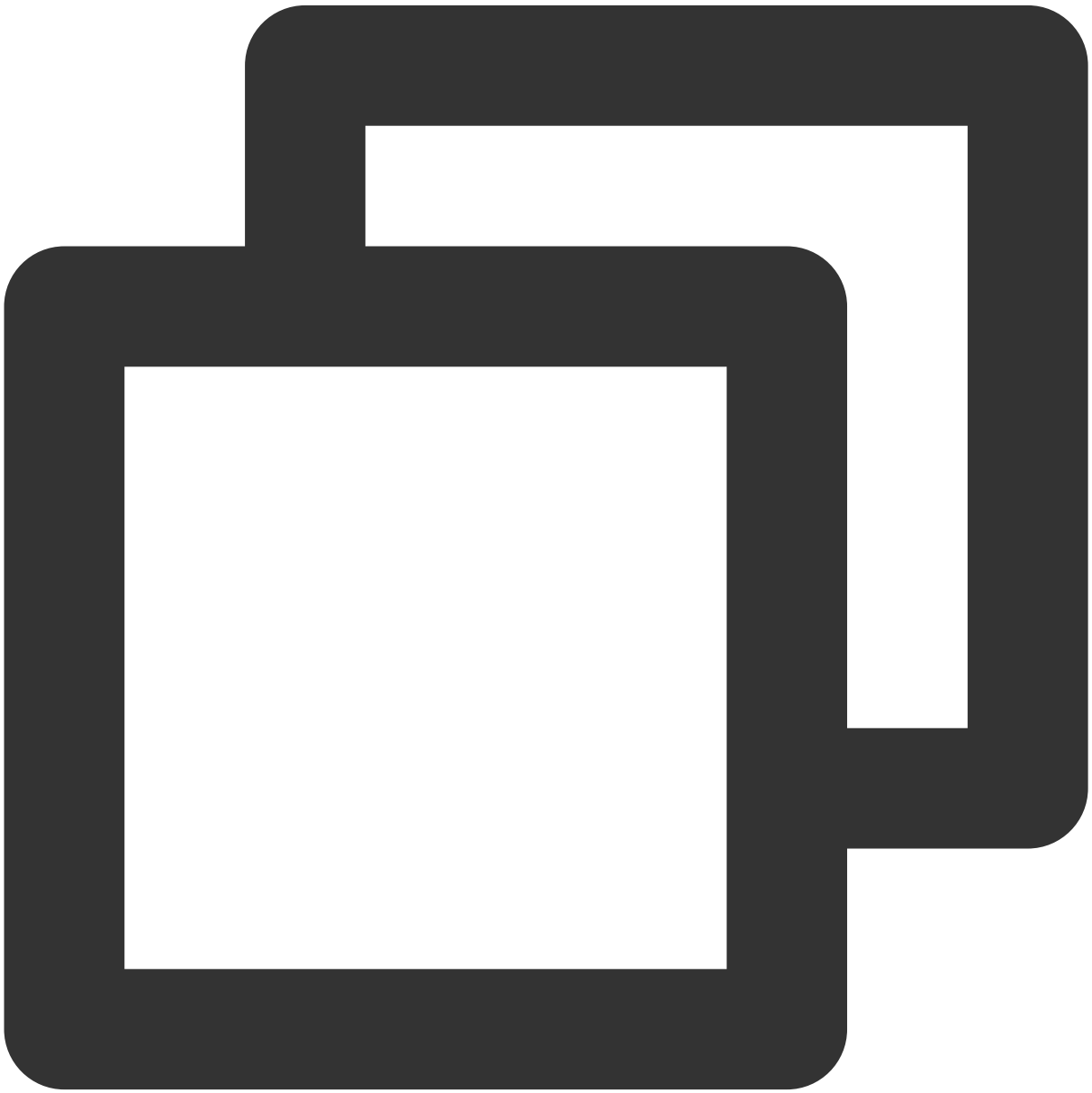
Statement

```
ALTER TABLE table_identifier
RENAME COLUMN old_column_name TO new_column_name
```

ALTER TABLE ... ALTER COLUMN

Change field type/remarks.

Statement



```
ALTER TABLE table_identifier
```

```
ALTER COLUMN col_name
{TYPE new_col_type | COMMENT col_comment}
```

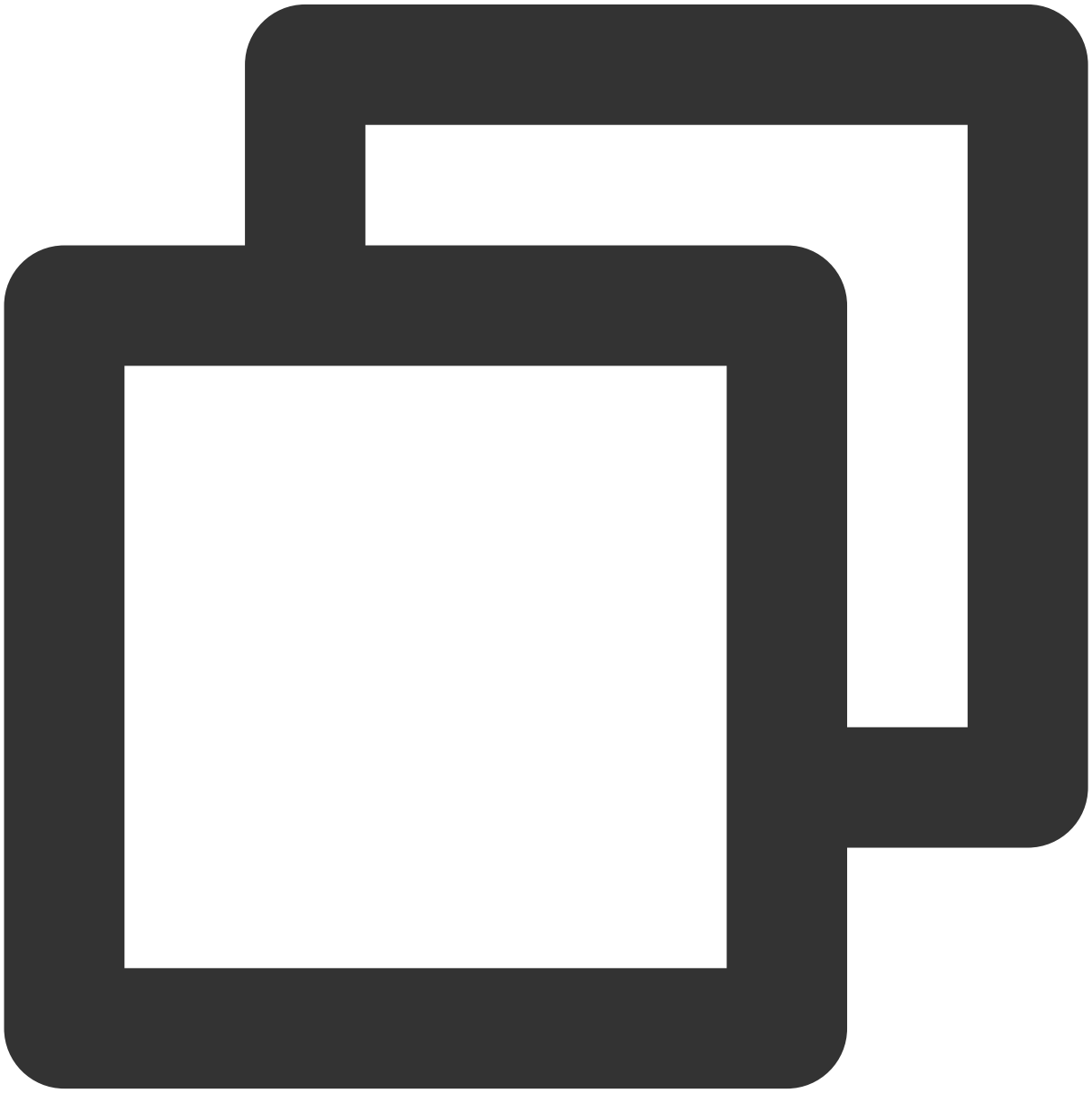
Currently, Iceberg TYPE changes only support type-safe field extensions:

int/integer -> long/bigint

float -> double

decimal(P,S) -> decimal(P2,S), where P2 > P, indicating increased precision.

Example



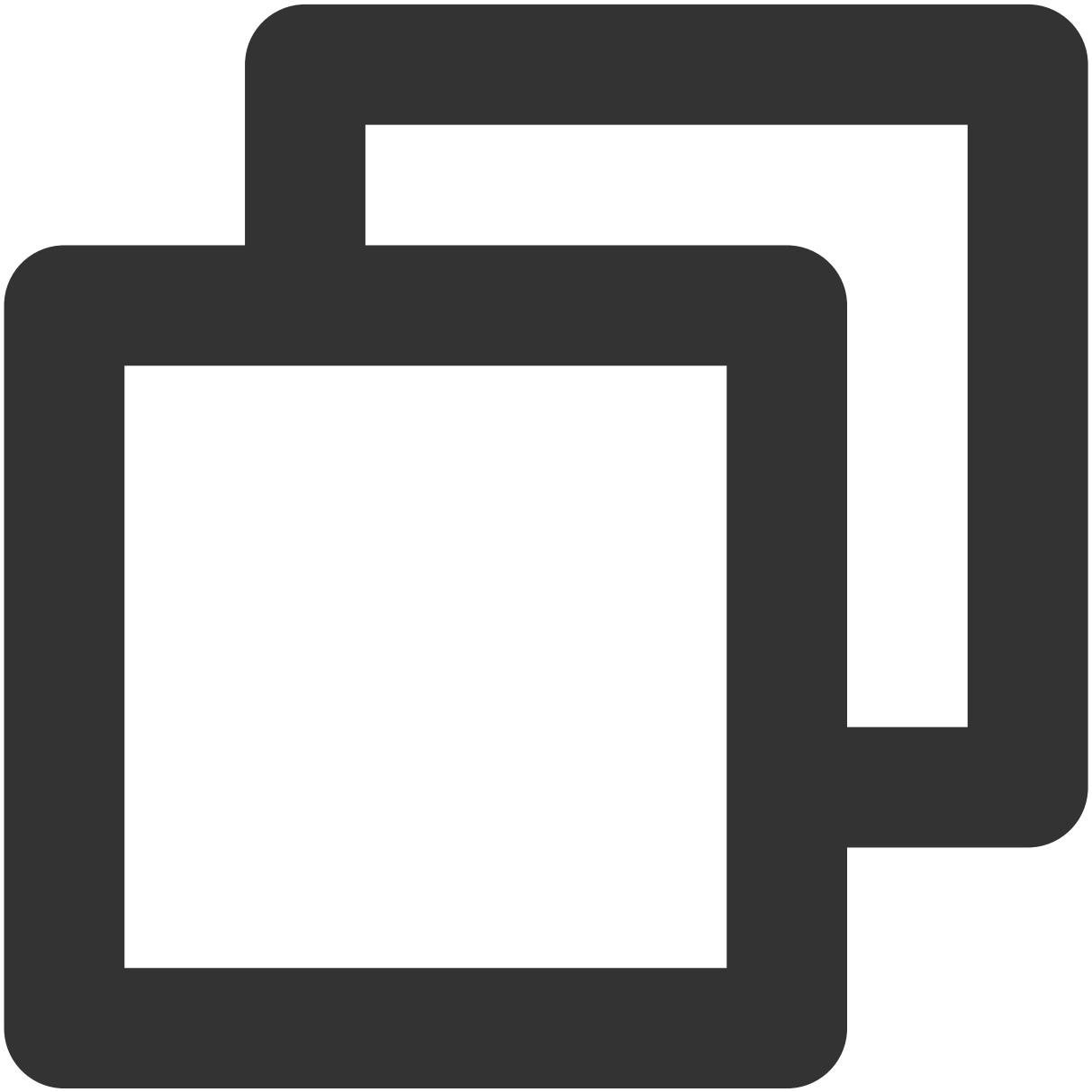
```
ALTER TABLE dempts ALTER COLUMN new_column_2 TYPE bigint;
```

```
ALTER TABLE dempts ALTER COLUMN new_column_2 comment 'alter docs';
```

ALTER TABLE ... DROP COLUMN

Delete a table field.

Statement



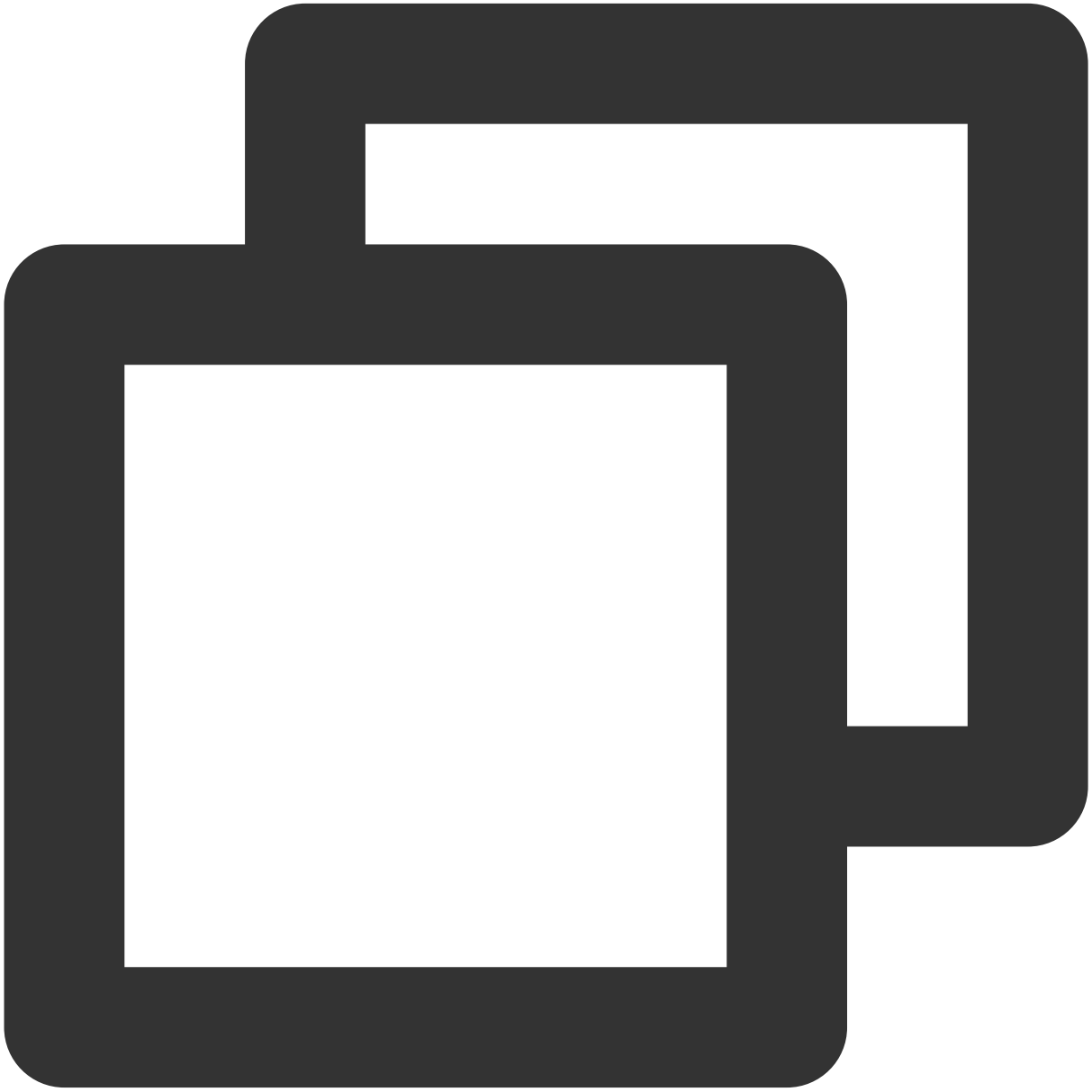
```
ALTER TABLE table_identifier  DROP COLUMN column_name
```

ALTER TABLE PARTITIONS

ALTER TABLE ... ADD PARTITION FIELD

Add a single partition field.

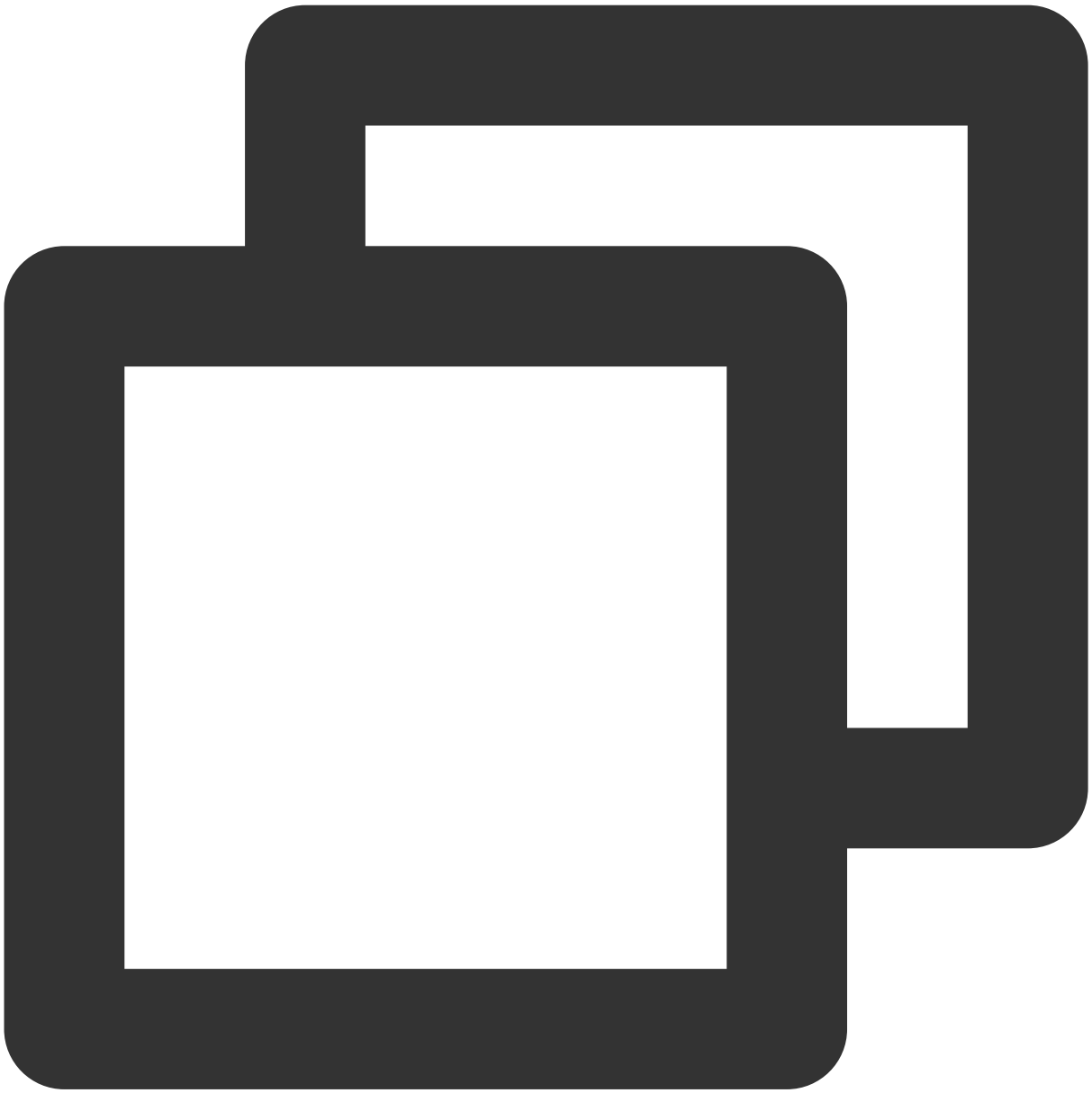
Statement



```
ALTER TABLE table_identifier
ADD PARTITION FIELD col_name|transform (col_name) [AS alias]
```

For transform, see the <1>CREATE TABLE< 1>.

Example

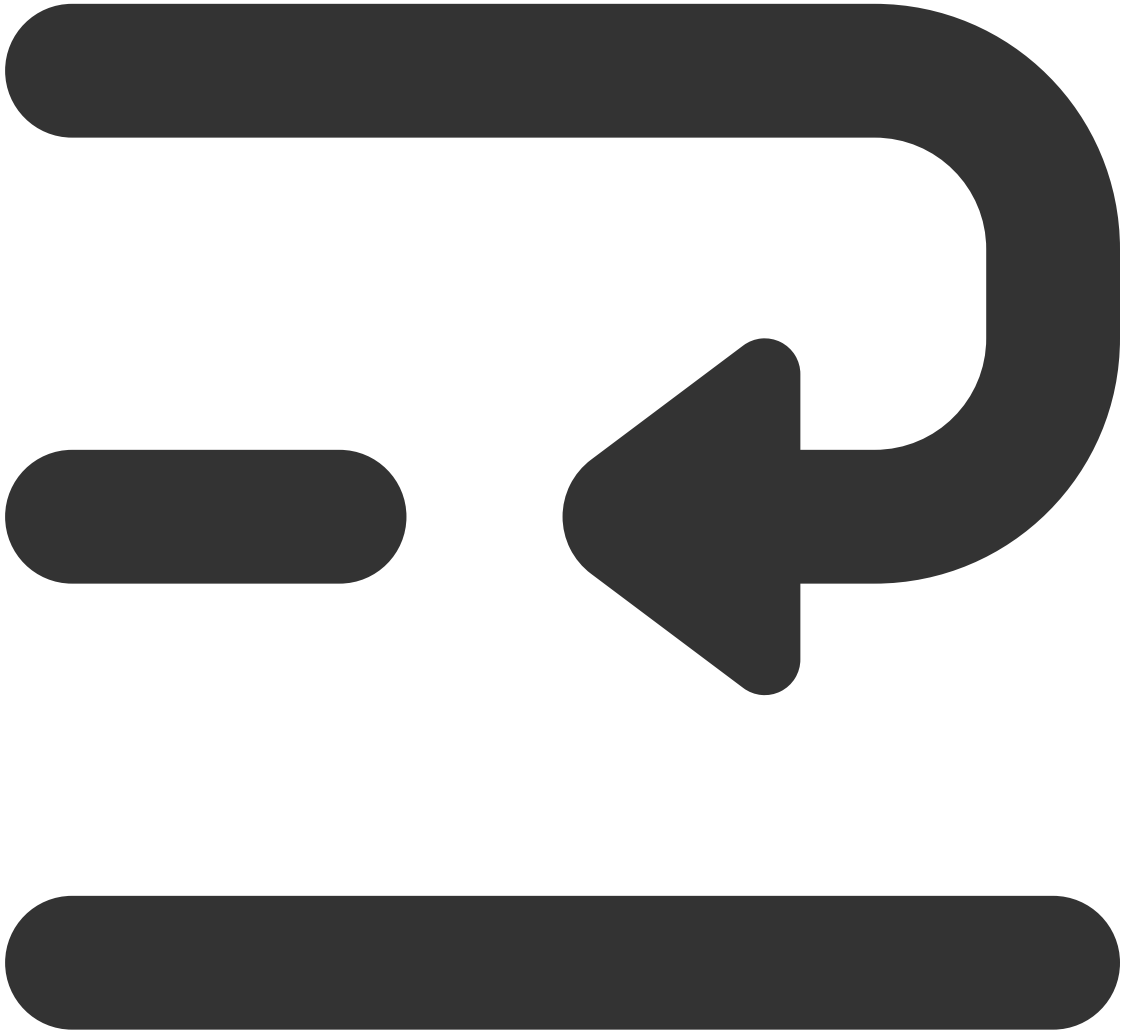


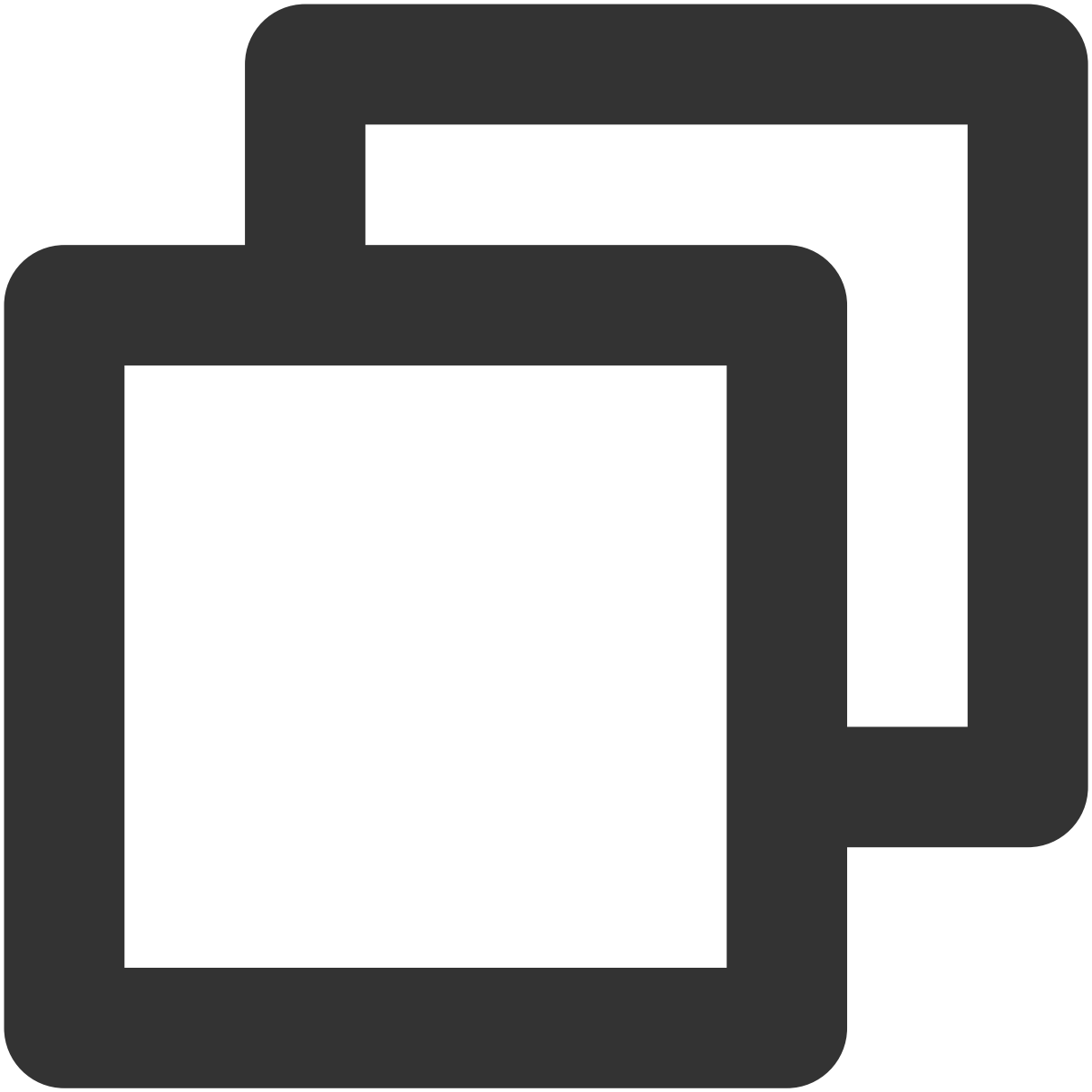
```
ALTER TABLE dempts ADD PARTITION FIELD new_column_1;
ALTER TABLE dempts ADD PARTITION FIELD bucket(3,new_column_2);
```

ALTER TABLE ... REPLACE PARTITION FIELD

Replace a single partition field.

Statement



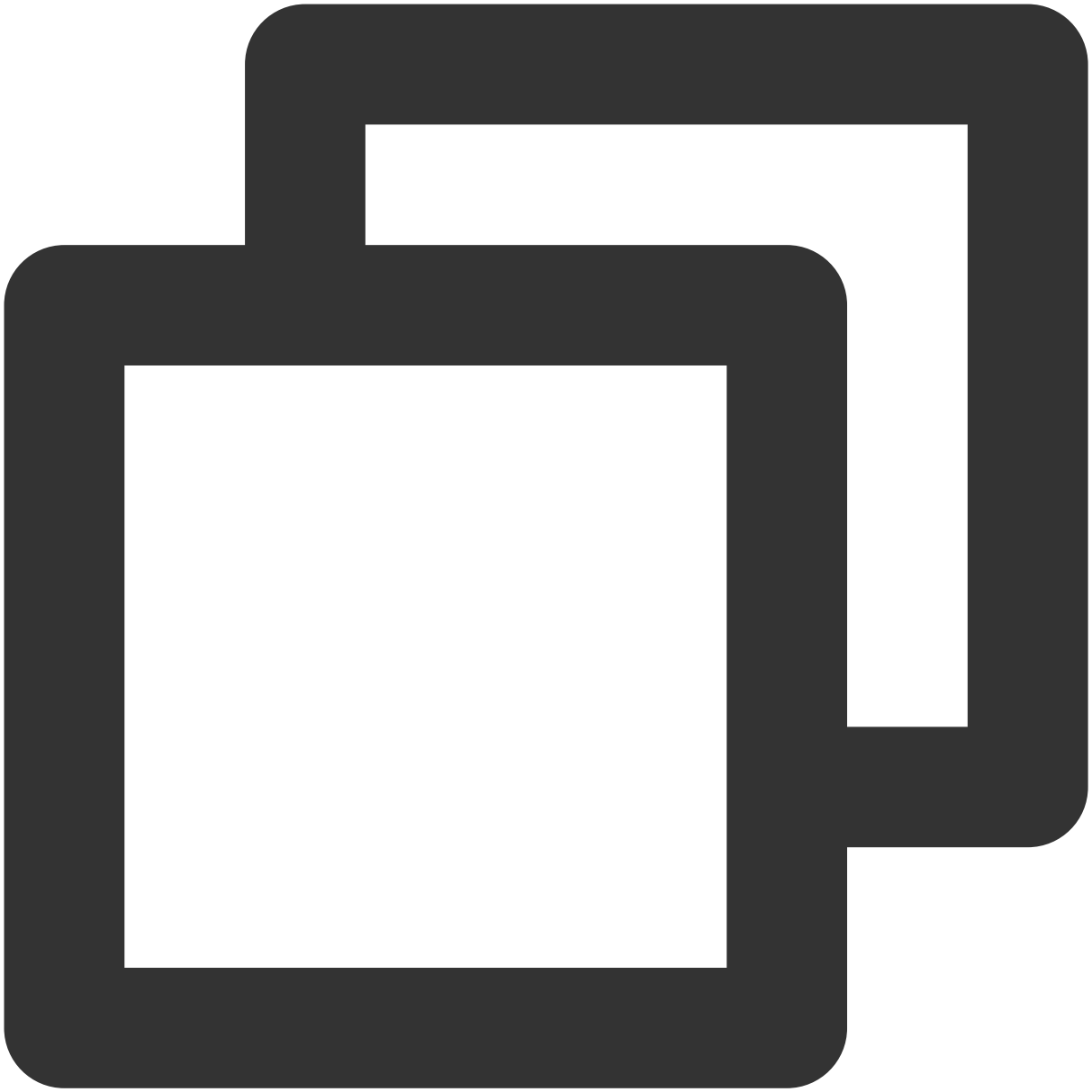


```
ALTER TABLE table_identifier REPLACE PARTITION FIELD col_name|transform (col_name)
```

ALTER TABLE ... DROP PARTITION FIELD

Delete a single partition field.

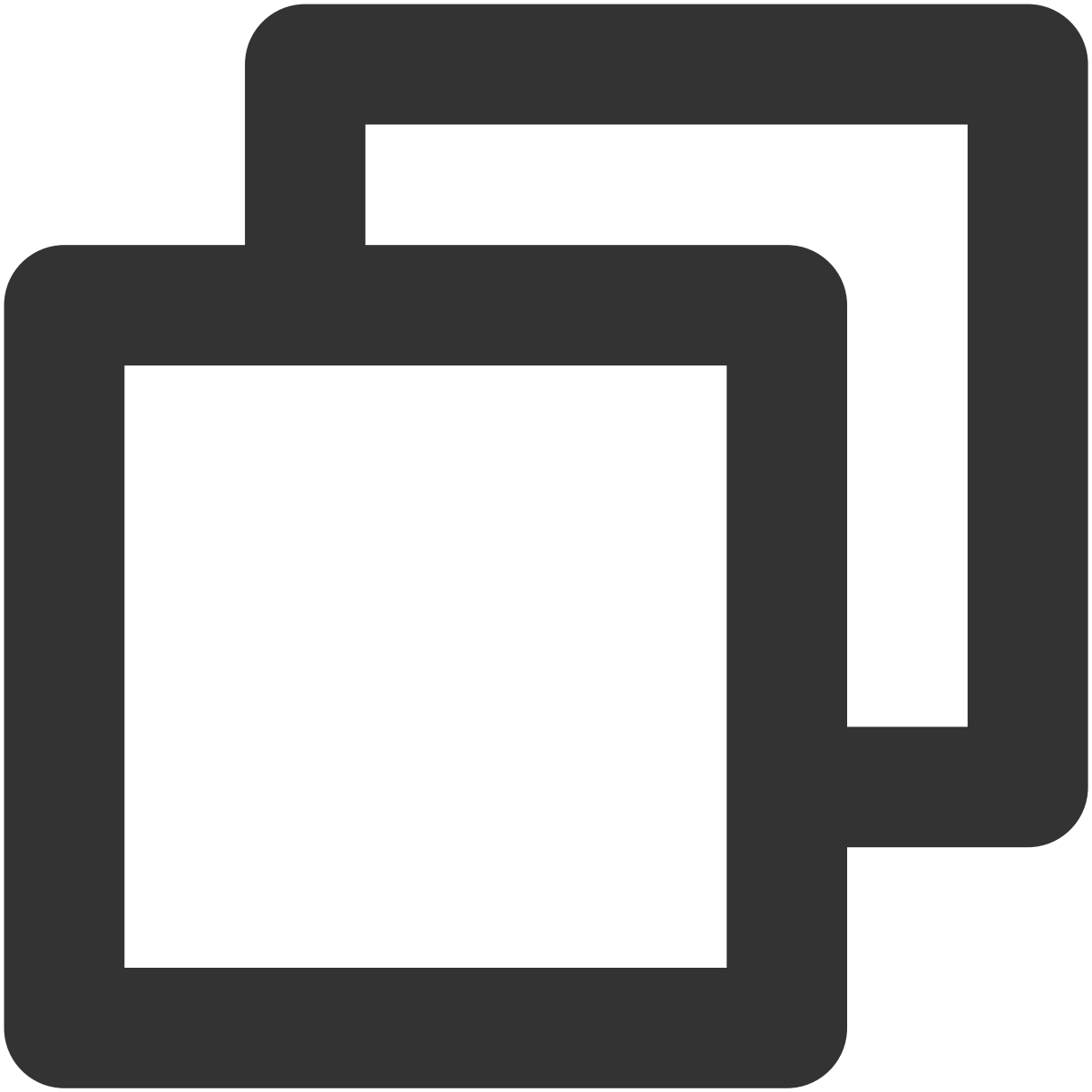
Statement



```
ALTER TABLE table_identifier
DROP PARTITION FIELD col_name|transform (col_name);
```

For transform, see the CREATE TABLE documentation.

Example

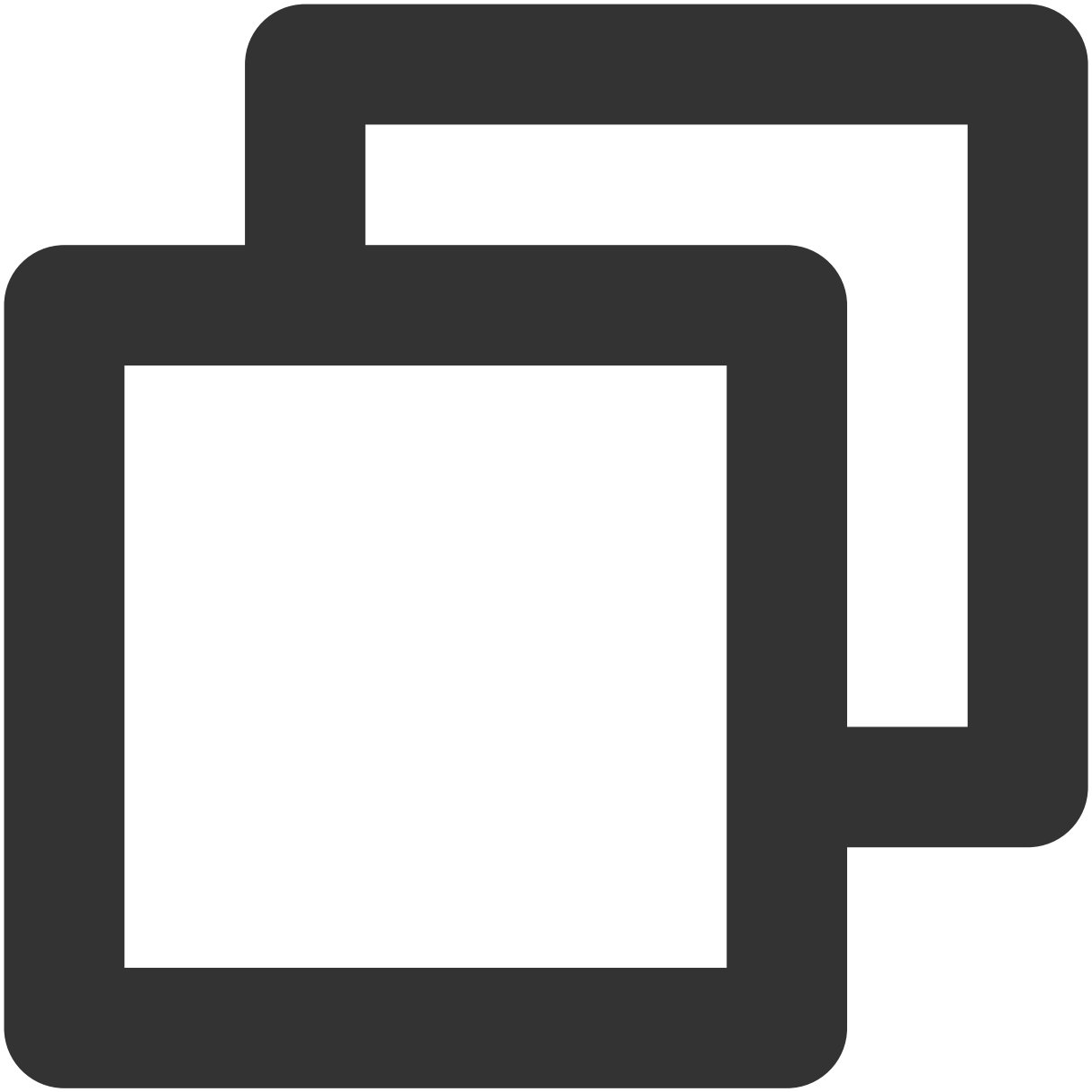


```
ALTER TABLE dempts DROP PARTITION FIELD new_column_1;
ALTER TABLE dempts DROP PARTITION FIELD bucket(3,new_column_2);
```

ALTER TABLE ... SET IDENTIFIER FIELDS

Add identifier fields attribute.

Statement



```
ALTER TABLE dempts SET IDENTIFIER FIELDS empno, name
```

ALTER TABLE .. DROP IDENTIFIER FIELDS

Add identifier fields attribute.

Statement

```
ALTER TABLE dempts DROP IDENTIFIER FIELDS empno, name
```

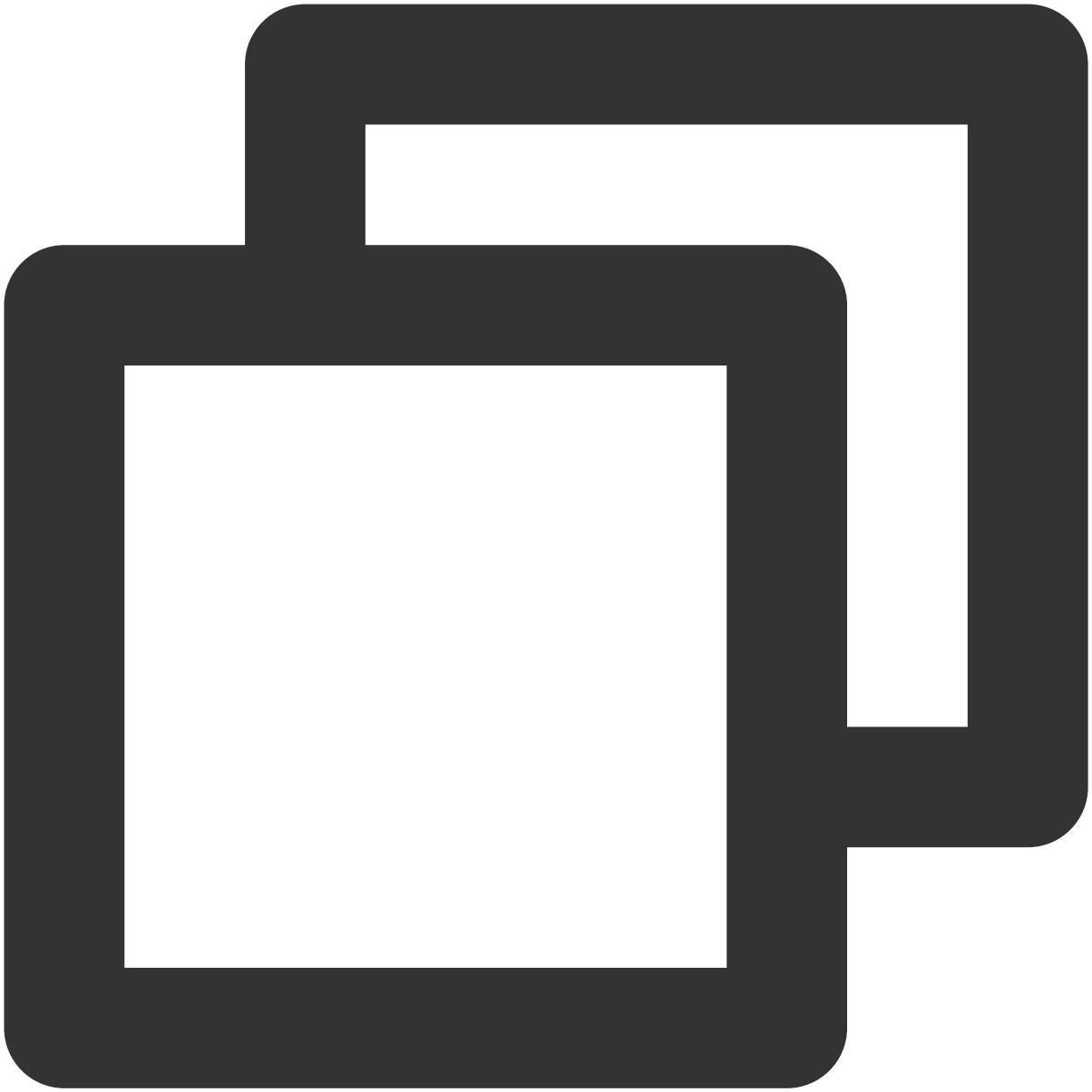
DML Statement

Last updated：2024-08-07 17:28:36

INSERT OVERWRITE | INTO

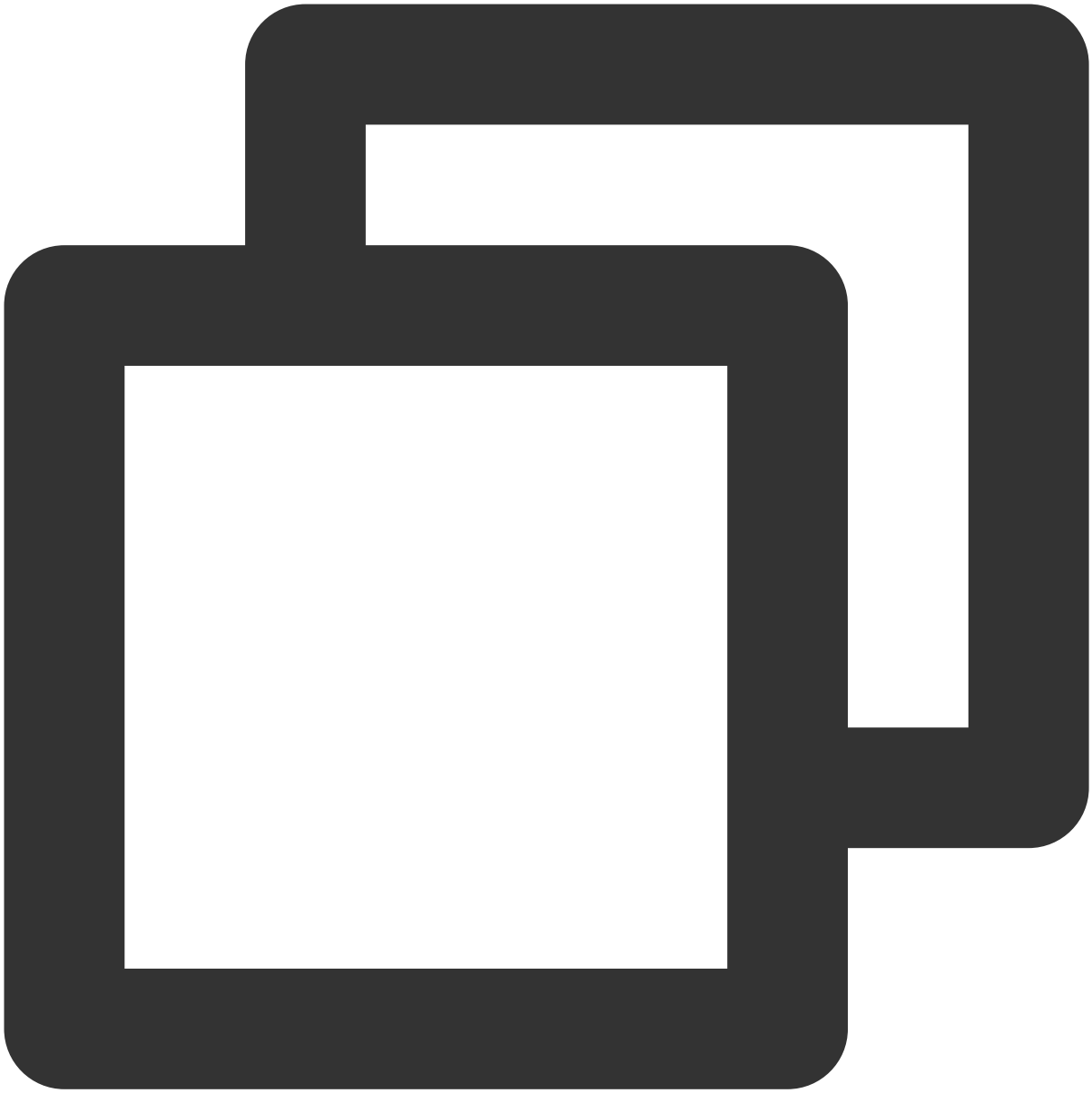
Row-level data insertion operation

Statement



```
INSERT { OVERWRITE | INTO } [ TABLE ] table_name
[ PARTITION clause ]
{ VALUES (column_values,...), (column_values,...)...
| SELECT select_expr}
```

Example

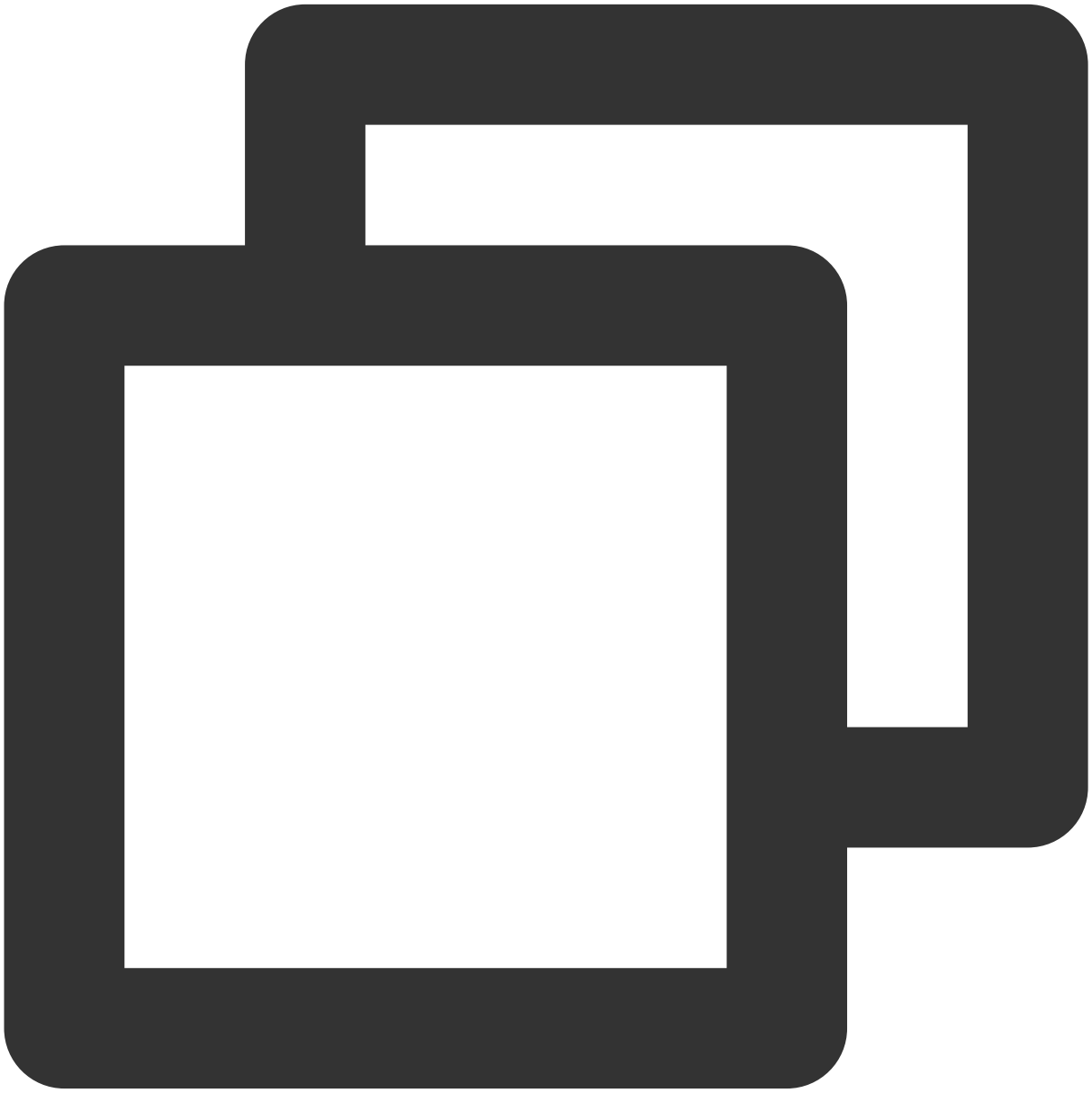


```
CREATE TABLE IF NOT EXISTS `table_01` (
    `id` INTEGER,
    `num` int,
    `name` STRING
) USING `iceberg`

INSERT INTO table_01 PARTITION(name='21') VALUES (1,2), (2,3);
INSERT INTO TABLE table_01 VALUES (3,2,'abc'), (4,3,'abd');
```

MERGE INTO

Update data by row (for replacing INSERT OVERWRITE).

Statement

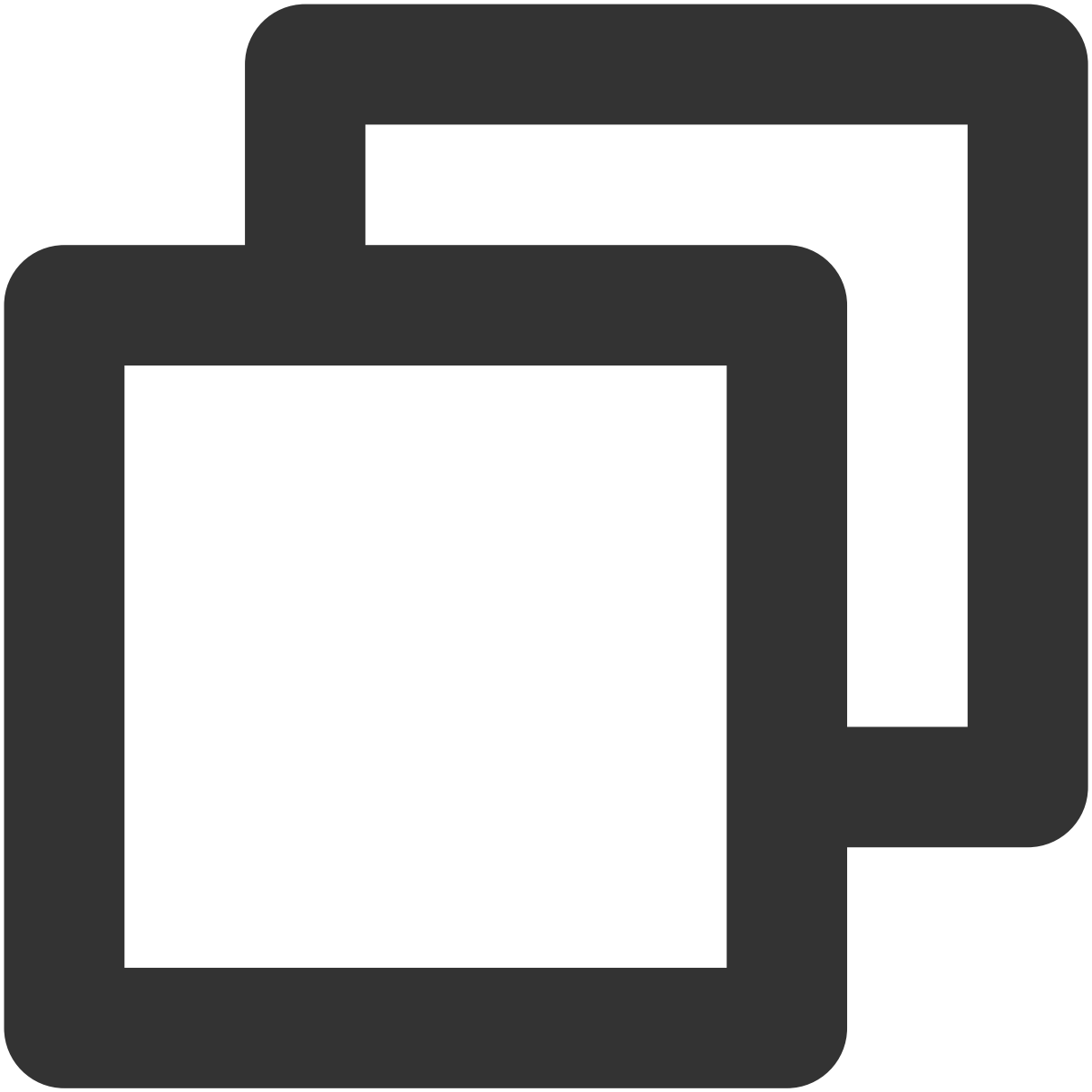


```
MERGE INTO target_table_name [target_alias]
USING source_table_reference [source_alias]
ON merge_condition
[ WHEN MATCHED [ AND condition ] THEN matched_action ] [...]
```

```
[ WHEN NOT MATCHED [ AND condition ] THEN not_matched_action ] [...]

matched_action
{ DELETE |
UPDATE SET * |
UPDATE SET { column1 = value1 } [, ...] }

not_matched_action
{ INSERT * |
INSERT (column1 [, ...] ) VALUES (value1 [, ...])
```

DELETE FROM

Statement

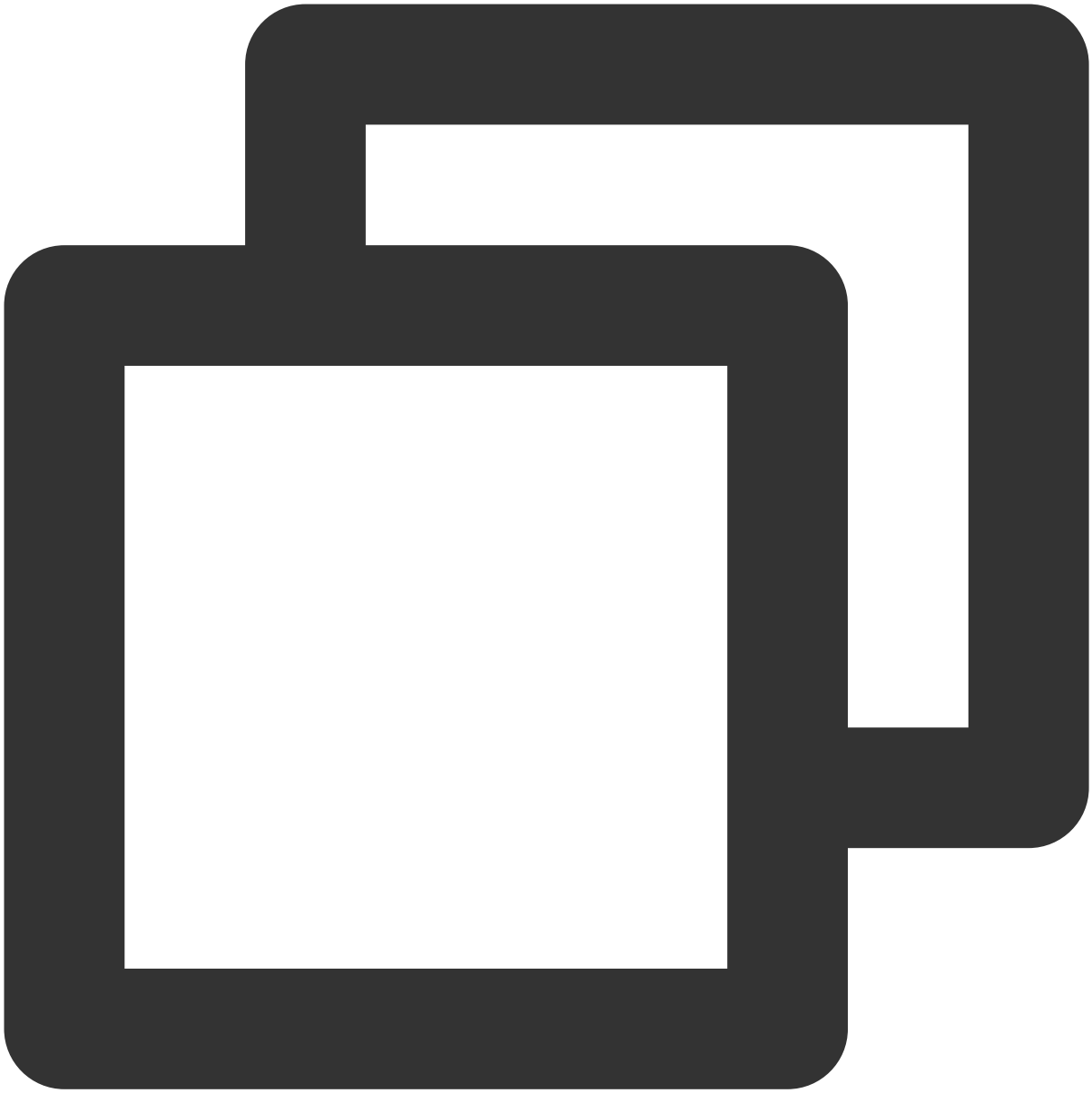


```
DELETE FROM table_name [table_alias] [WHERE predicate]
```

UPDATE

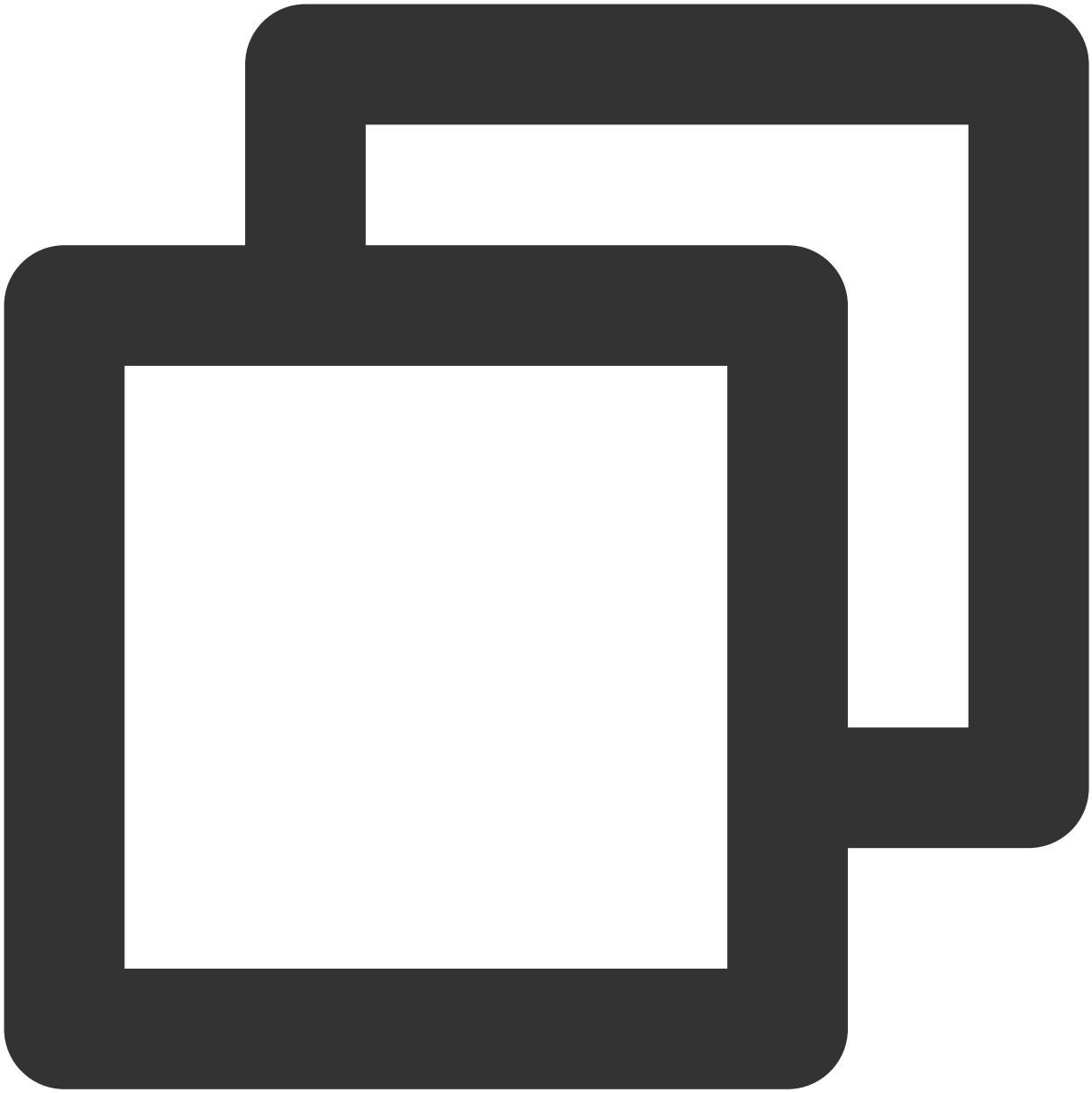
Support UPDATE operation starting from Spark 3.1.

Statement



```
UPDATE table_identifier [table_alias]
SET { { column_name | field_name } = expr } [, ...]
[WHERE clause]
```

Example



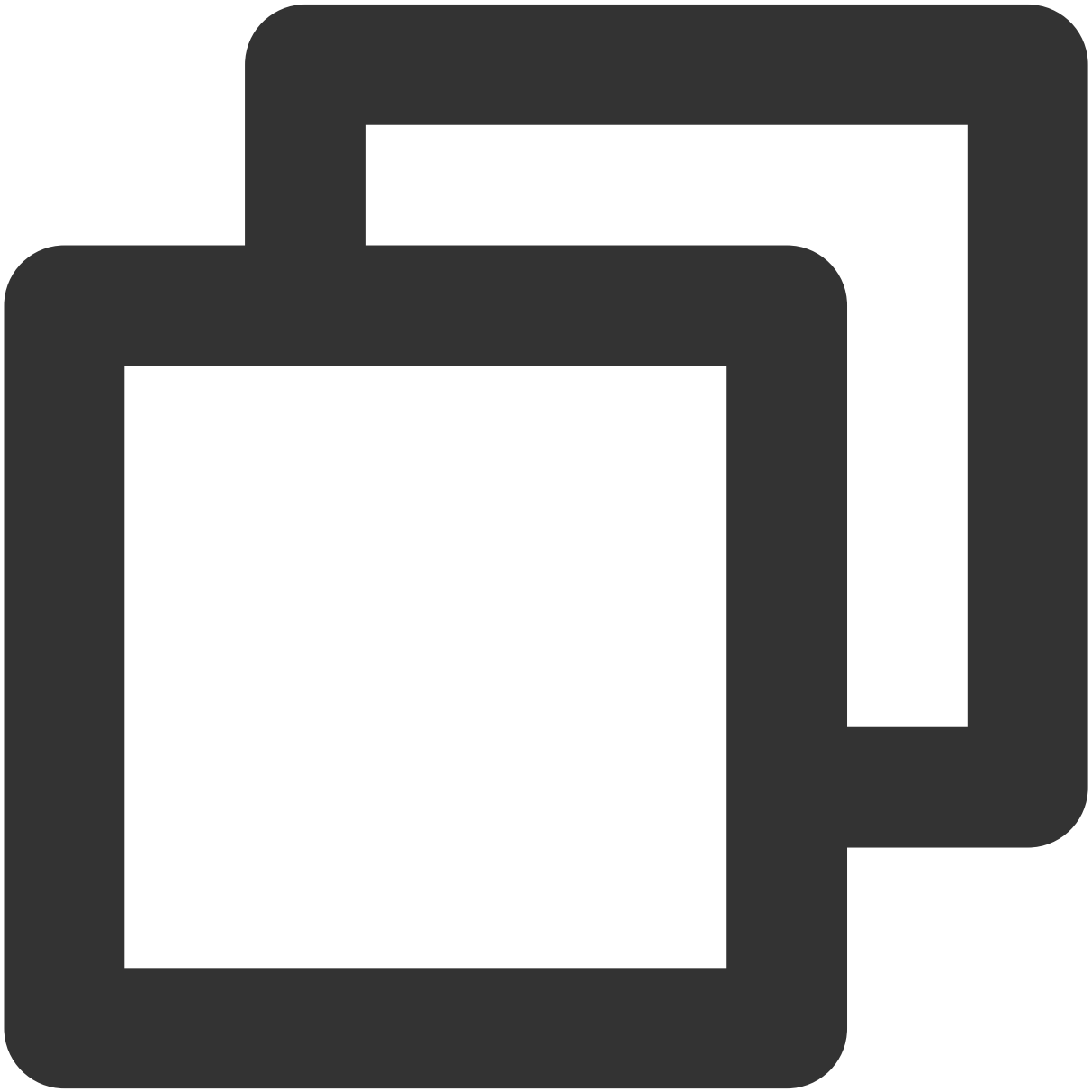
```
UPDATE dempts SET c1 = 'update_c1', c2 = 'update_c2'
WHERE ts >= '2020-05-01 00:00:00' and ts < '2020-06-01 00:00:00'

UPDATE dempts SET session_time = 0, ignored = true
WHERE session_time < (SELECT min(session_time) FROM prod.db.good_events)

UPDATE dempts AS t1 SET order_status = 'returned'
WHERE EXISTS (SELECT oid FROM prod.db.returned_orders WHERE t1.oid = oid)
```

DQL Statement

Last updated：2024-08-07 17:28:49

SELECT

Statement



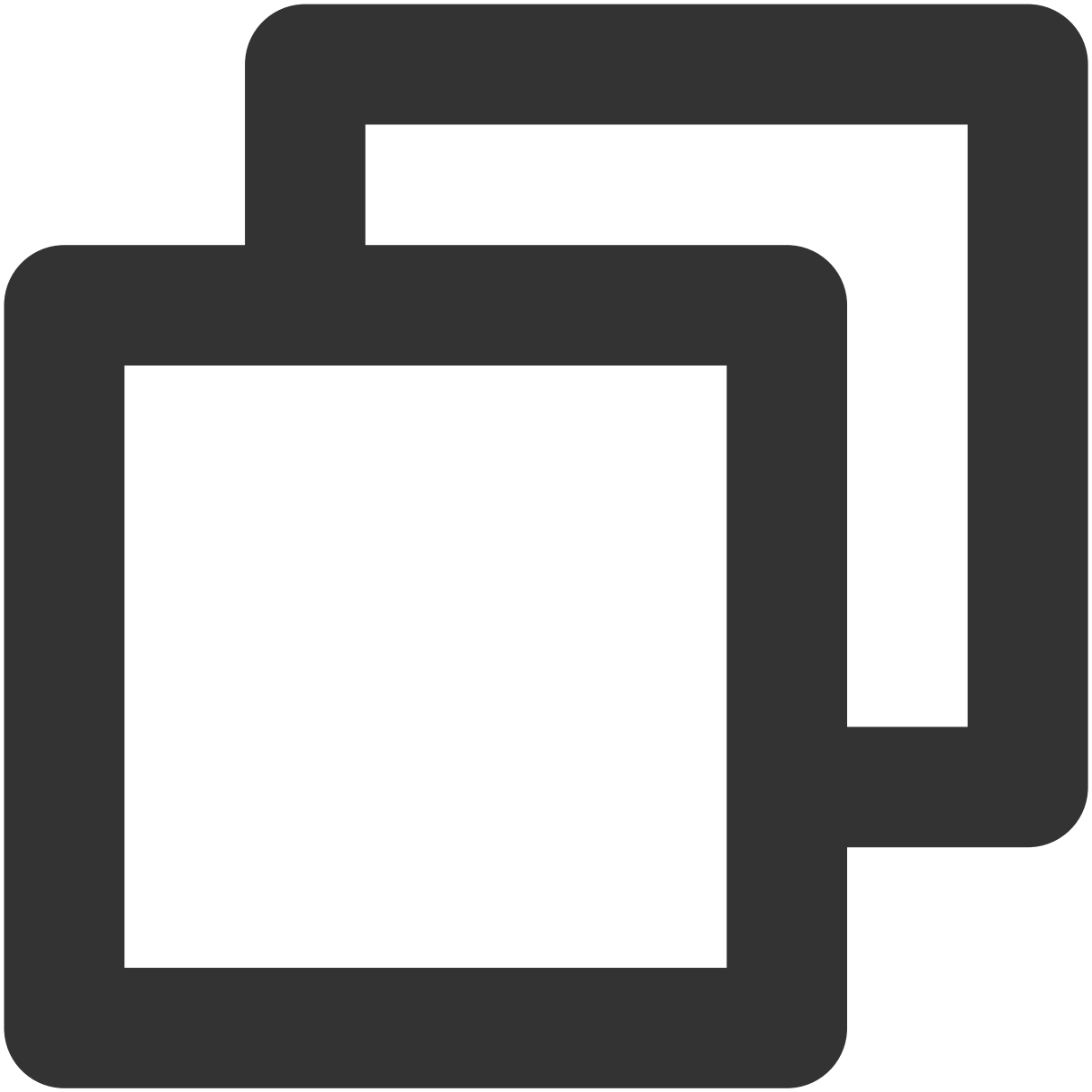
```
SELECT [ hints ] [ ALL | DISTINCT ] { named_expression | star_clause } [, ...]
```

```
FROM from_item [, ...]
    [ LATERAL VIEW clause ]
    [ PIVOT clause ]
    [ WHERE clause ]
    [ GROUP BY clause ]
    [ HAVING clause]
    [ QUALIFY clause ]

from_item
{ table_name [ TABLESAMPLE clause ] [ table_alias ] |
    JOIN clause |
    [ LATERAL ] table_valued_function [ table_alias ] |
    VALUES clause |
    [ LATERAL ] ( query ) [ TABLESAMPLE clause ] [ table_alias ] }

named_expression
    expression [ column_alias ]

star_clause
    [ { table_name | view_name } . ] *
```

TABLE METADATA

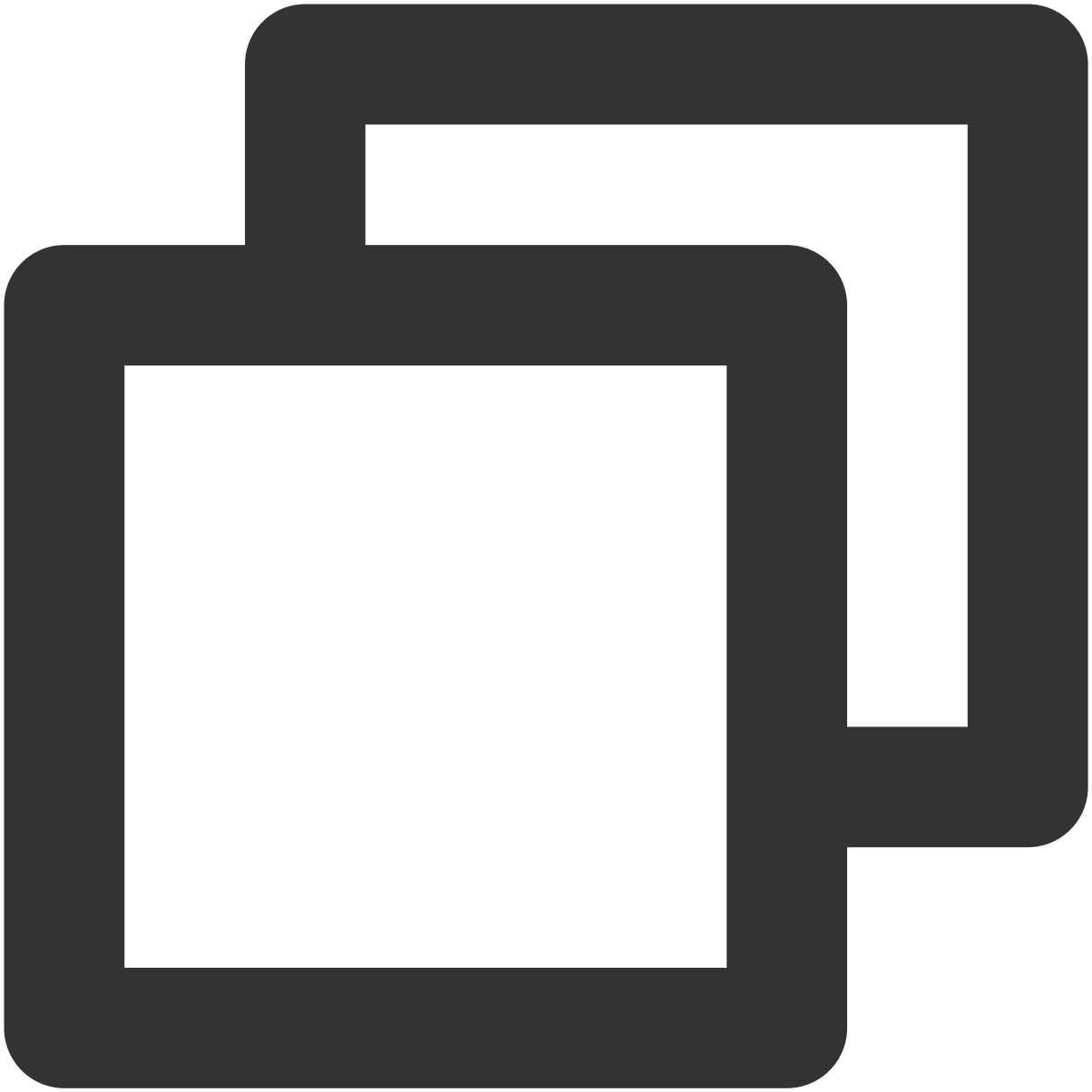
Support four-part Iceberg table metadata queries, including `history`, `snapshots`, `files`, and `manifests`.

Statement



```
SELECT select_expr (, select_expr)*
FROM `Catalog`.`db`.`tableName${history|snapshots|files|manifests|partitions|all_da
[WHERE where_condition]
[LIMIT [offset,] rows]
```

Example



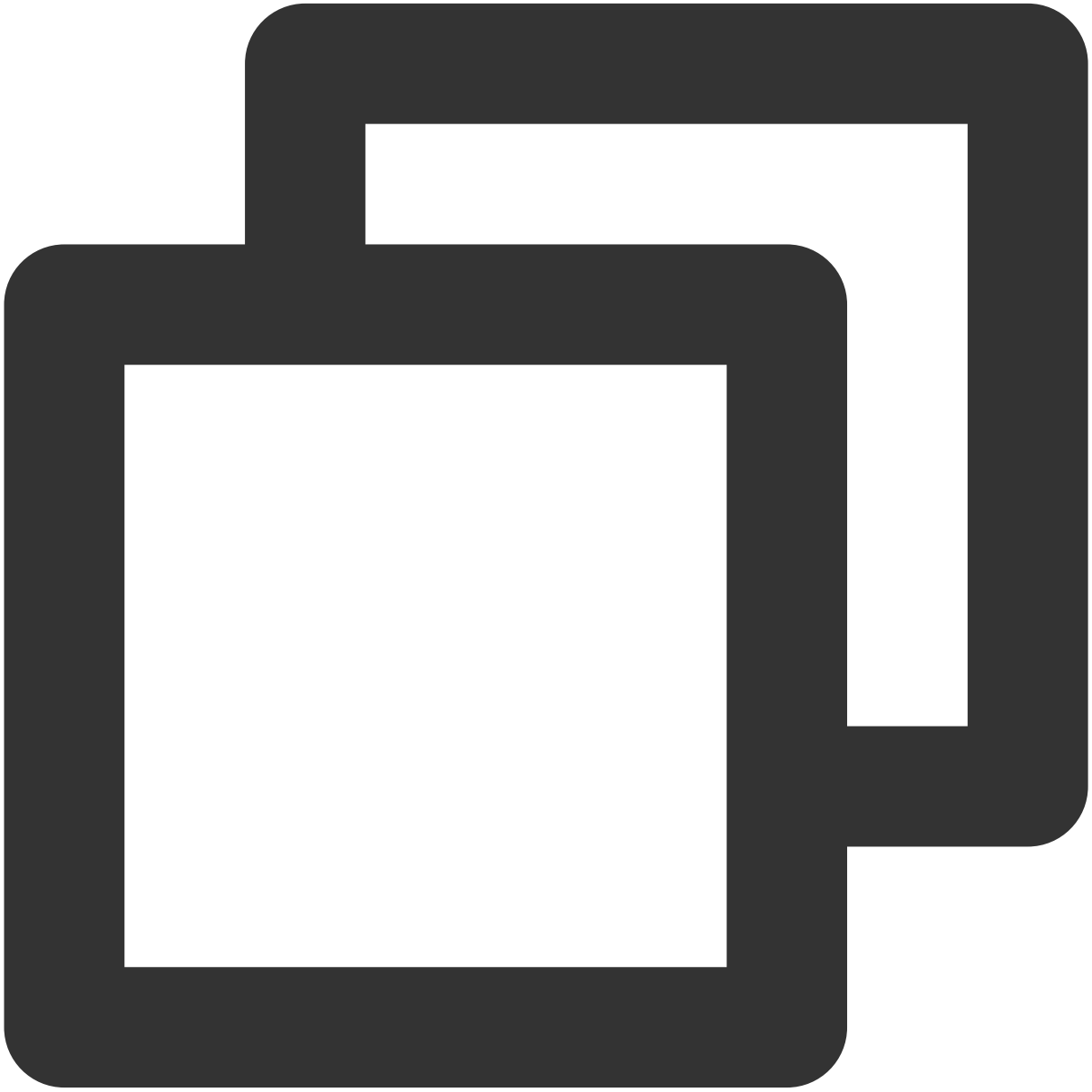
```
SELECT * FROM `DataLakeCatalog`.`validation`.`dempts$history` ORDER BY snapshot_id
SELECT * FROM `DataLakeCatalog`.`validation`.`dempts$snapshots` ORDER BY snapshot_i
SELECT * FROM `DataLakeCatalog`.`validation`.`dempts$files` ORDER BY file_size_in_b
SELECT * FROM `DataLakeCatalog`.`validation`.`dempts$manifests` ORDER BY length LIM
SELECT * FROM `DataLakeCatalog`.`validation`.`dempts$partitions` LIMIT 10;
SELECT * FROM `DataLakeCatalog`.`validation`.`dempts$all_data_files` LIMIT 10;
SELECT * FROM `DataLakeCatalog`.`validation`.`dempts$all_manifests` LIMIT 10;
```

TIME TRAVEL

FOR SYSTEM_TIME AS OF/ TIMESTAMP AS OF

Supported in Spark 3.3 and later, with support for both string and Unix timestamp formats.

Example



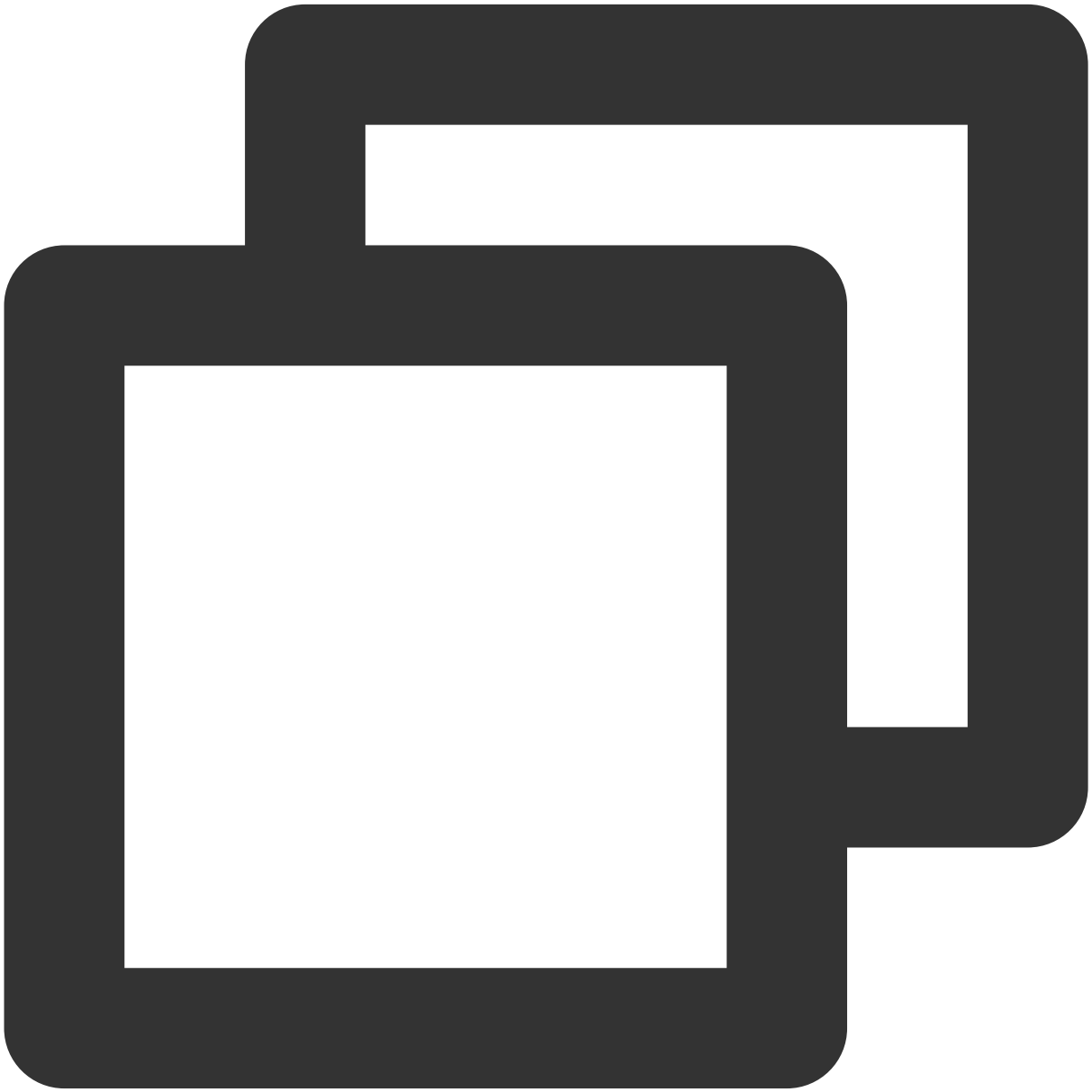
```
SELECT empno FROM sales.emp FOR SYSTEM_TIME AS OF '1986-10-26 01:21:00';
SELECT empno FROM sales.emp FOR SYSTEM_TIME AS OF 12324235546;
SELECT empno FROM sales.emp TIMESTAMP AS OF '1986-10-26 01:21:00';
```

```
SELECT empno FROM sales.emp TIMESTAMP AS OF 11111;
```

FOR SYSTEM_VERSION AS OF / VERSION AS OF (Supported in Spark 3.3)

Supports both string and snapshot ID.

Example

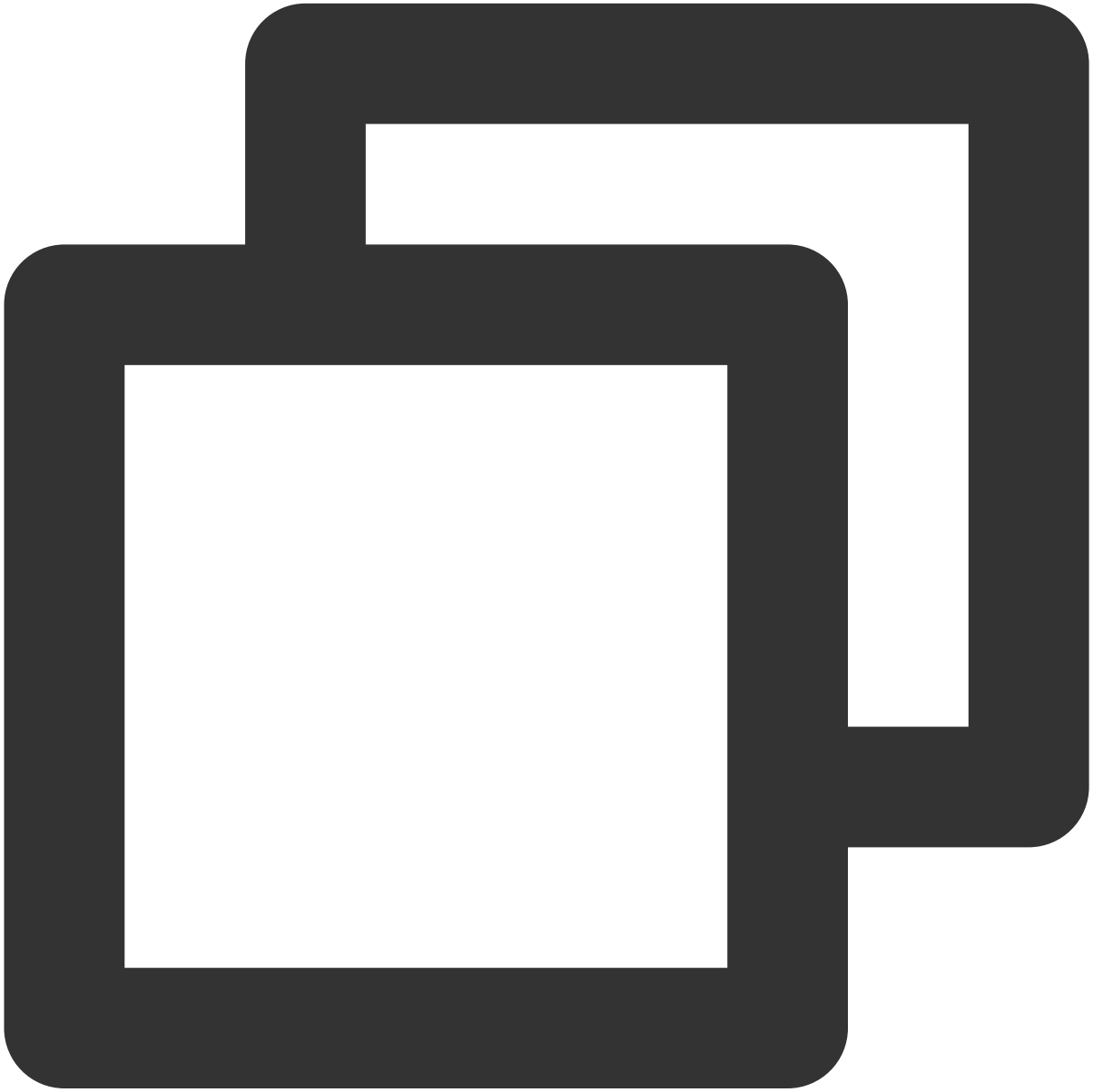


```
SELECT empno FROM sales.emp VERSION AS OF 'Snapshot123456789';
SELECT empno FROM sales.emp VERSION AS OF 11111;
SELECT empno FROM sales.emp FOR SYSTEM_VERSION AS OF 'Snapshot123456789';
```

```
SELECT empno FROM sales.emp FOR SYSTEM_VERSION AS OF 11111;
```

AT_TIMESTAMP_XXXX

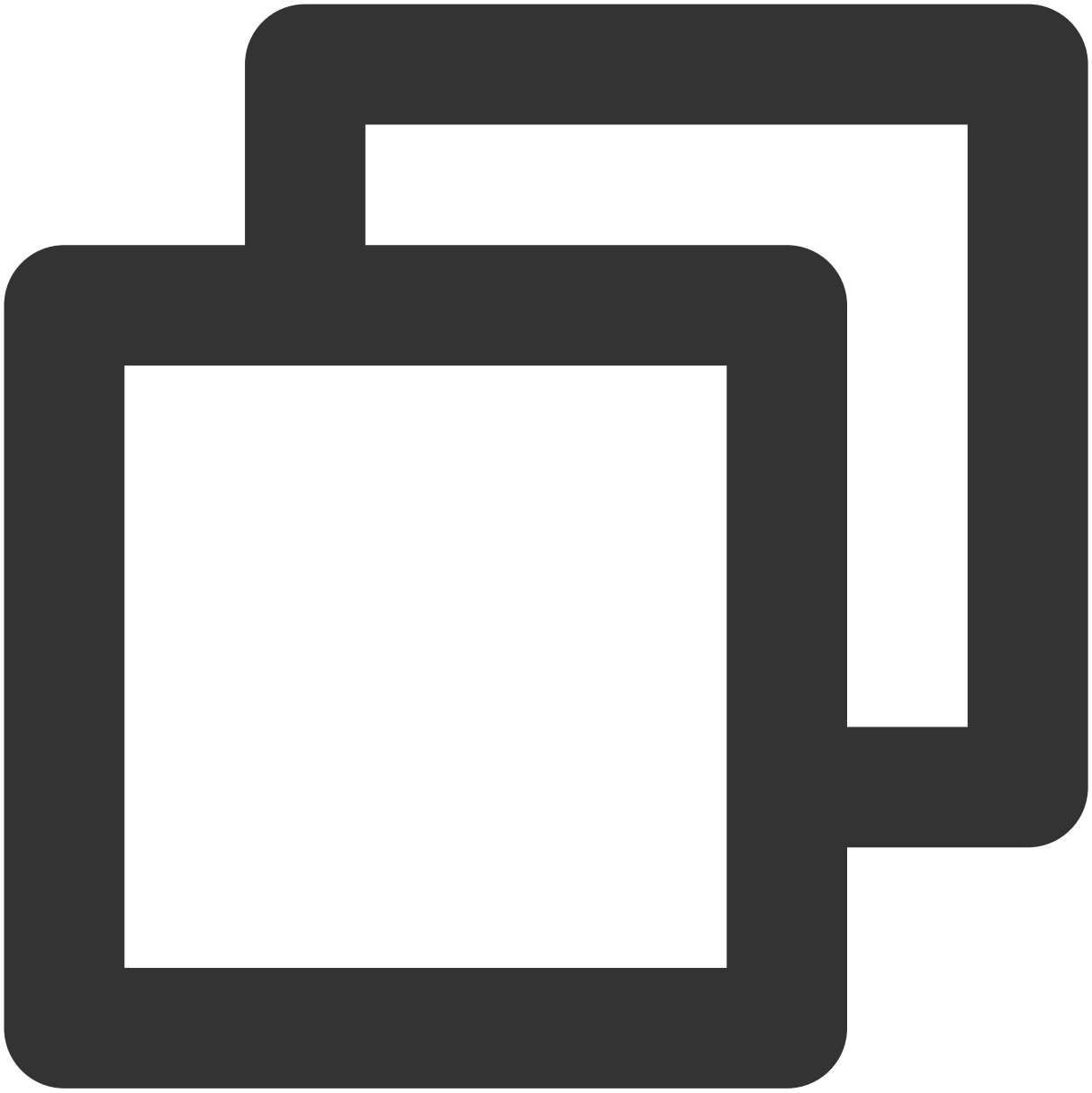
Example



```
SELECT * FROM `sales`.`emp$at_timestamp_1111`;
```

SNAPSHOT_ID_XXXX

Example



```
SELECT * FROM `sales`.`emp$snapshot_id_1111`
```

Procedure

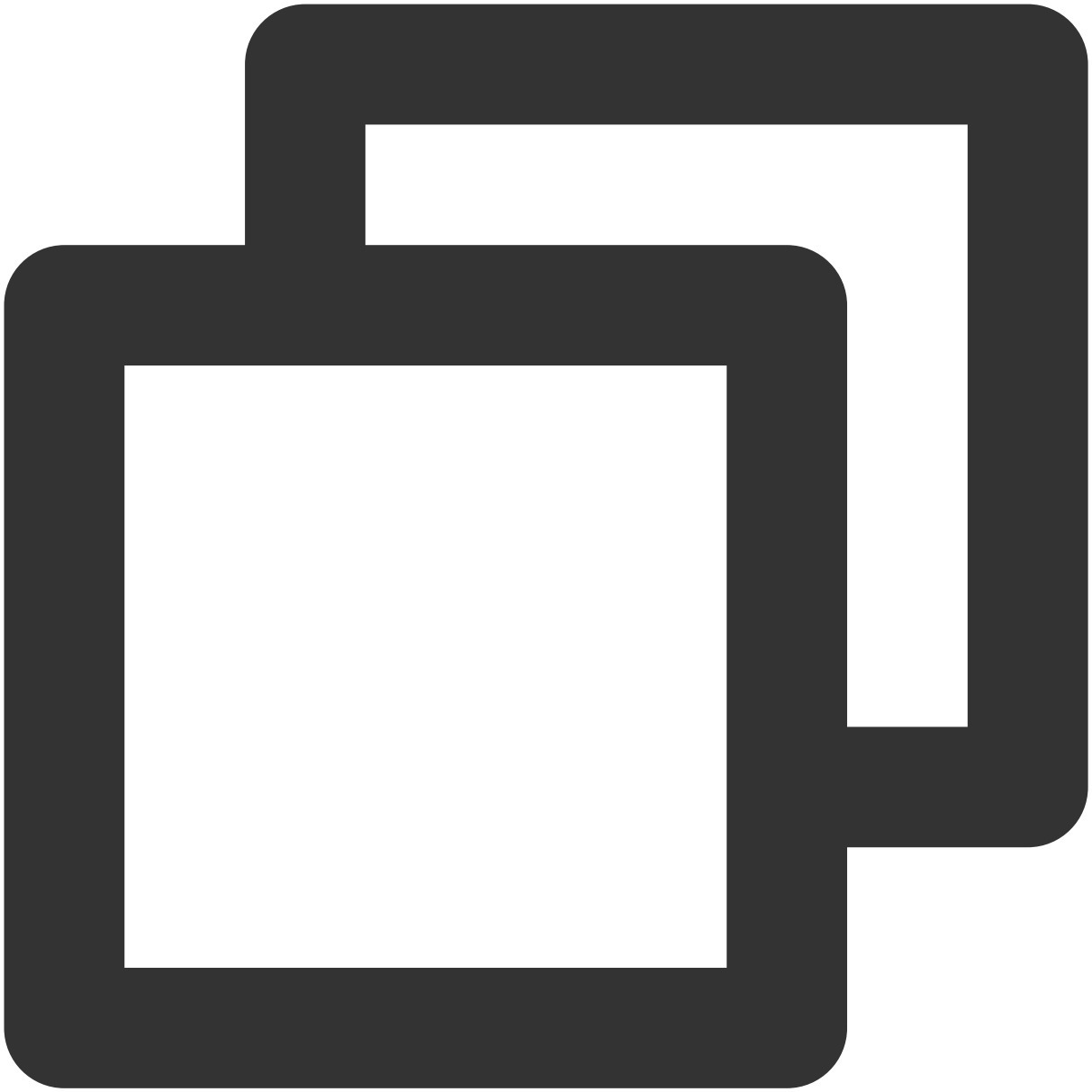
Last updated：2024-08-07 17:29:12

Description

Supported engine: SparkSQL

Applicable table: External and native Iceberg tables

Basic Statement



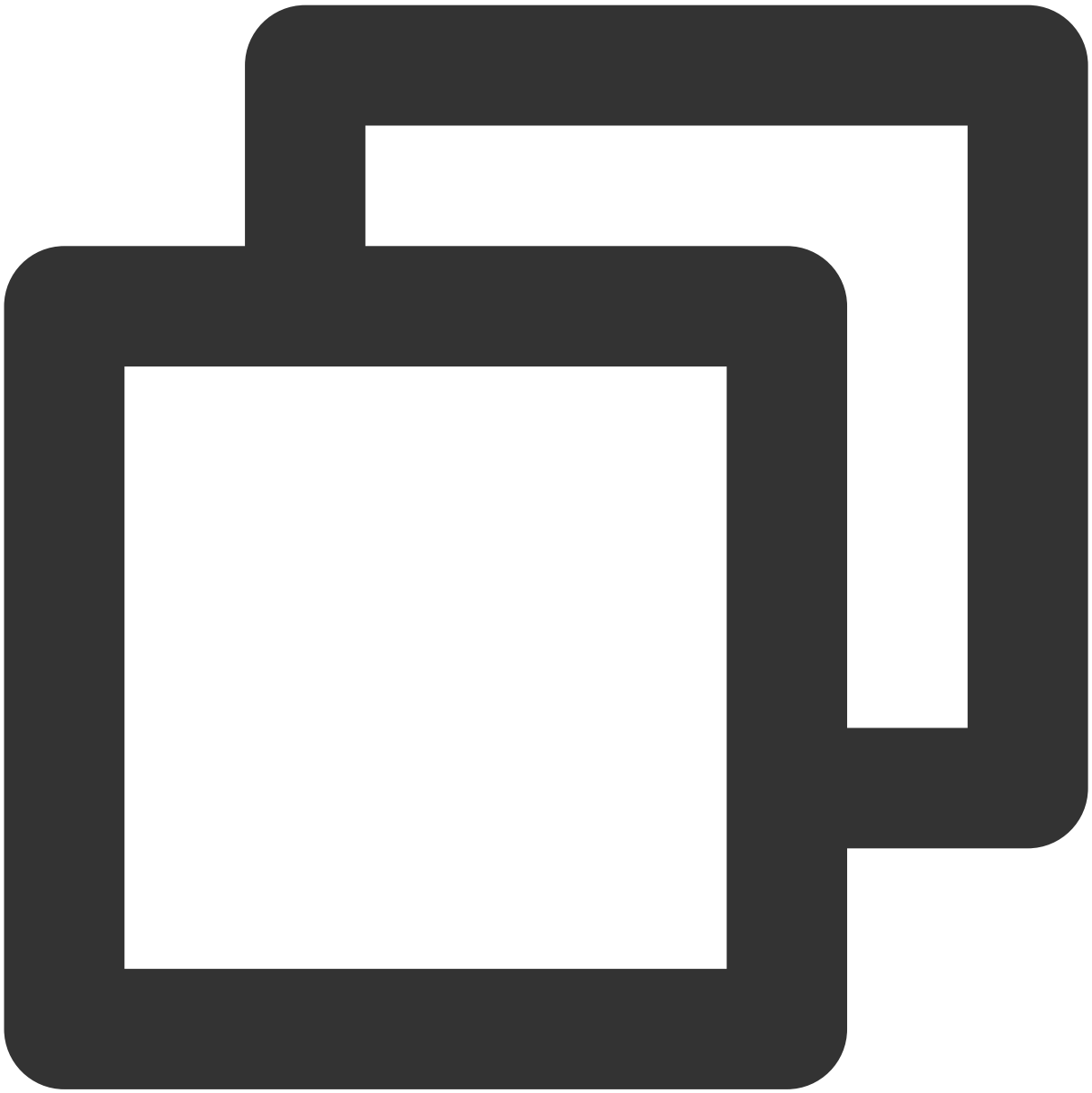
```
CALL catalog_name.system.procedure_name(arg_name_2 => arg_2, arg_name_1 => arg_1);

CALL catalog_name.system.procedure_name(arg_1, arg_2, ... arg_n);
```

Snapshot Management

rollback_to_snapshot

Roll back the snapshot to a specified version.

Input parameters: table name and version number.

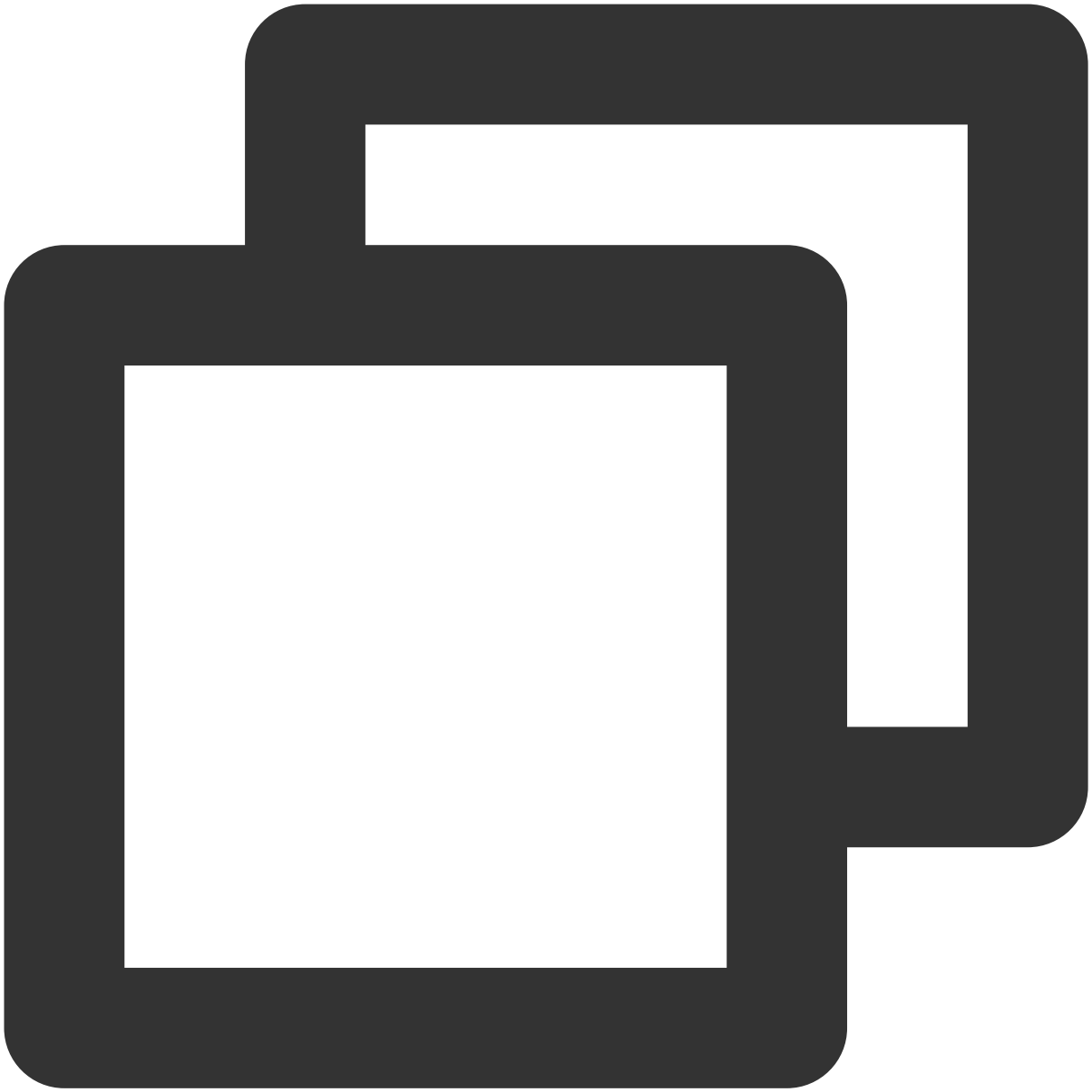


```
CALL `DataLakeCatalog`.`system`.rollback_to_snapshot('validation.dempts', 1);
```

rollback_to_timestamp

Roll back the snapshot to a specified timestamp.

Input parameters: table name and timestamp

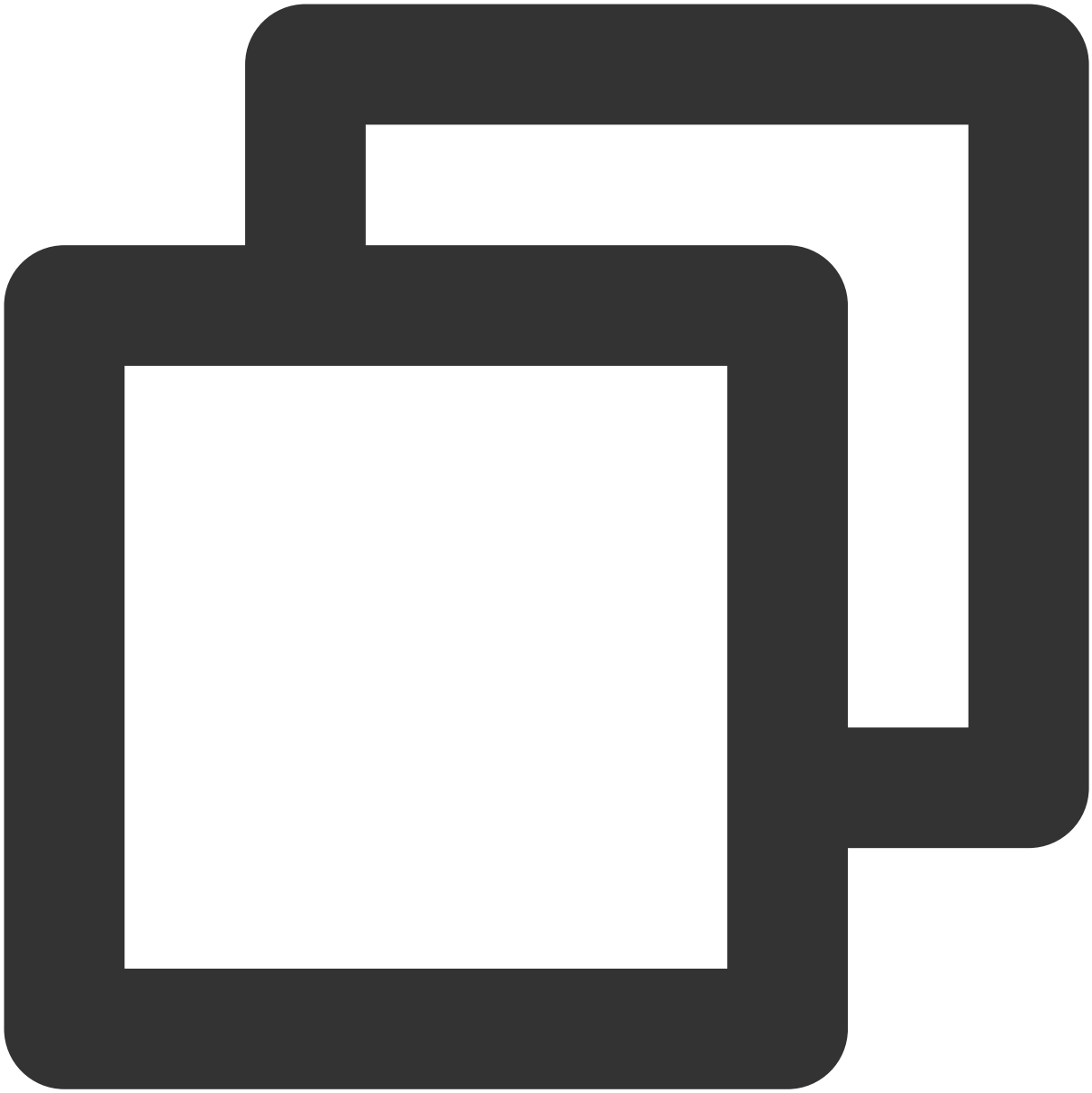


```
CALL `DataLakeCatalog`.`system`.rollback_to_timestamp('validation.dempts', TIMESTAM
```

set_current_snapshot

Set the current snapshot version.

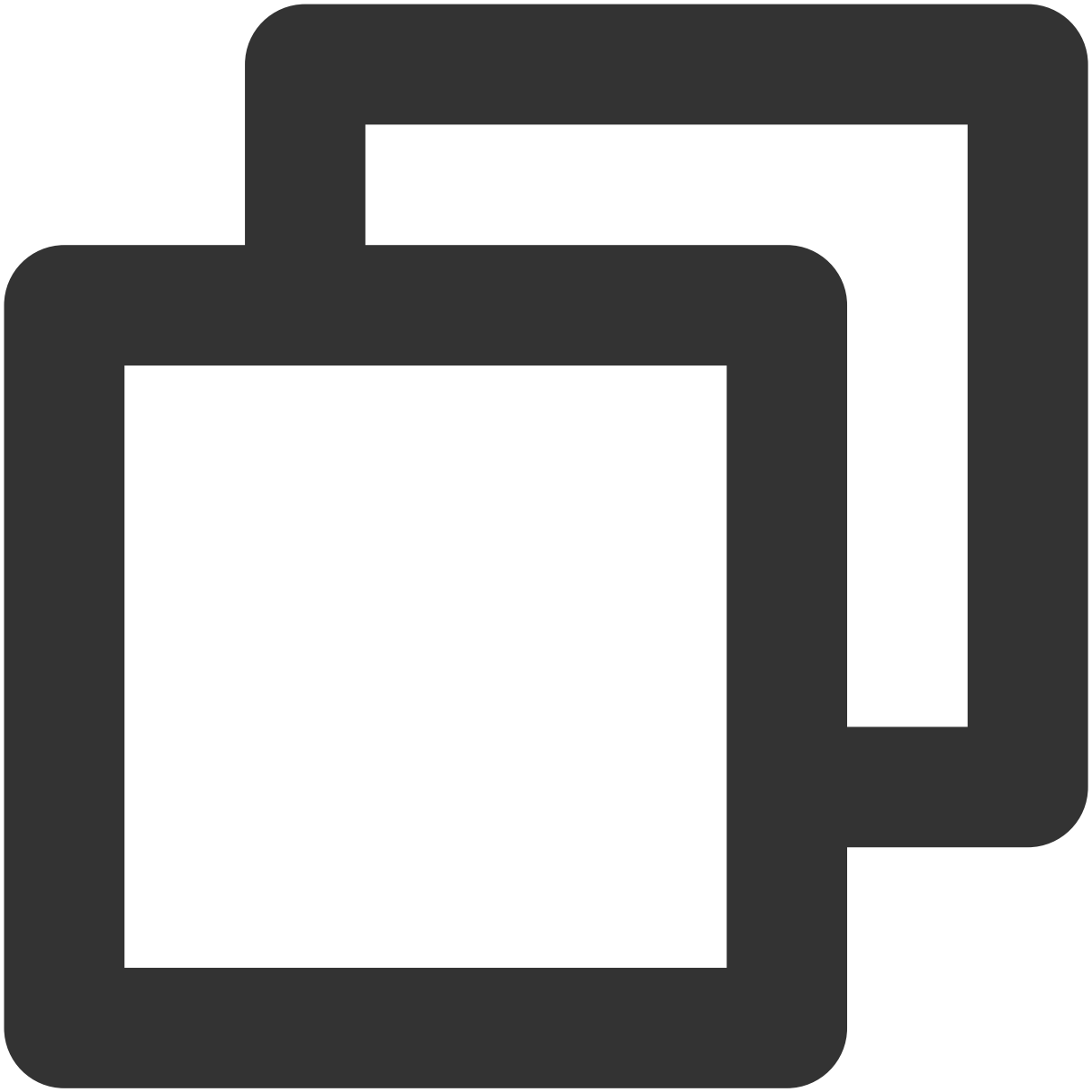
Input parameters: table name and version number.



```
CALL `DataLakeCatalog`.`system`.set_current_snapshot('validation.dempts', 1);
```

cherrypick_snapshot

Cherrypick from a specified snapshot version to the current snapshot.

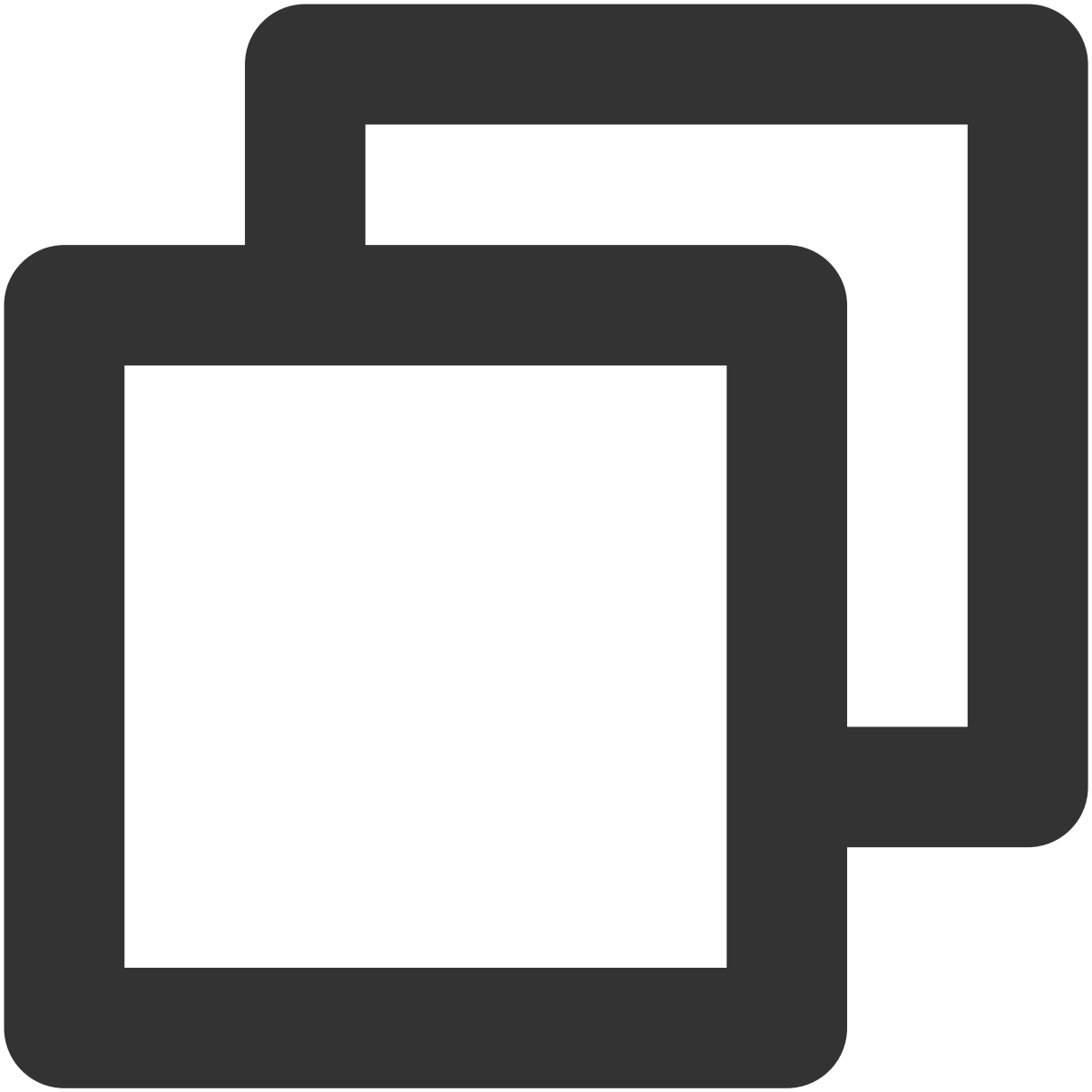


```
CALL `DataLakeCatalog`.`system`.cherrypick_snapshot('validation.dempts', 1);
CALL `DataLakeCatalog`.`system`.cherrypick_snapshot(snapshot_id => 1, table => 'my_
```

Metadata Management

expire_snapshots

Clean up expired snapshots to reduce the number of small files.



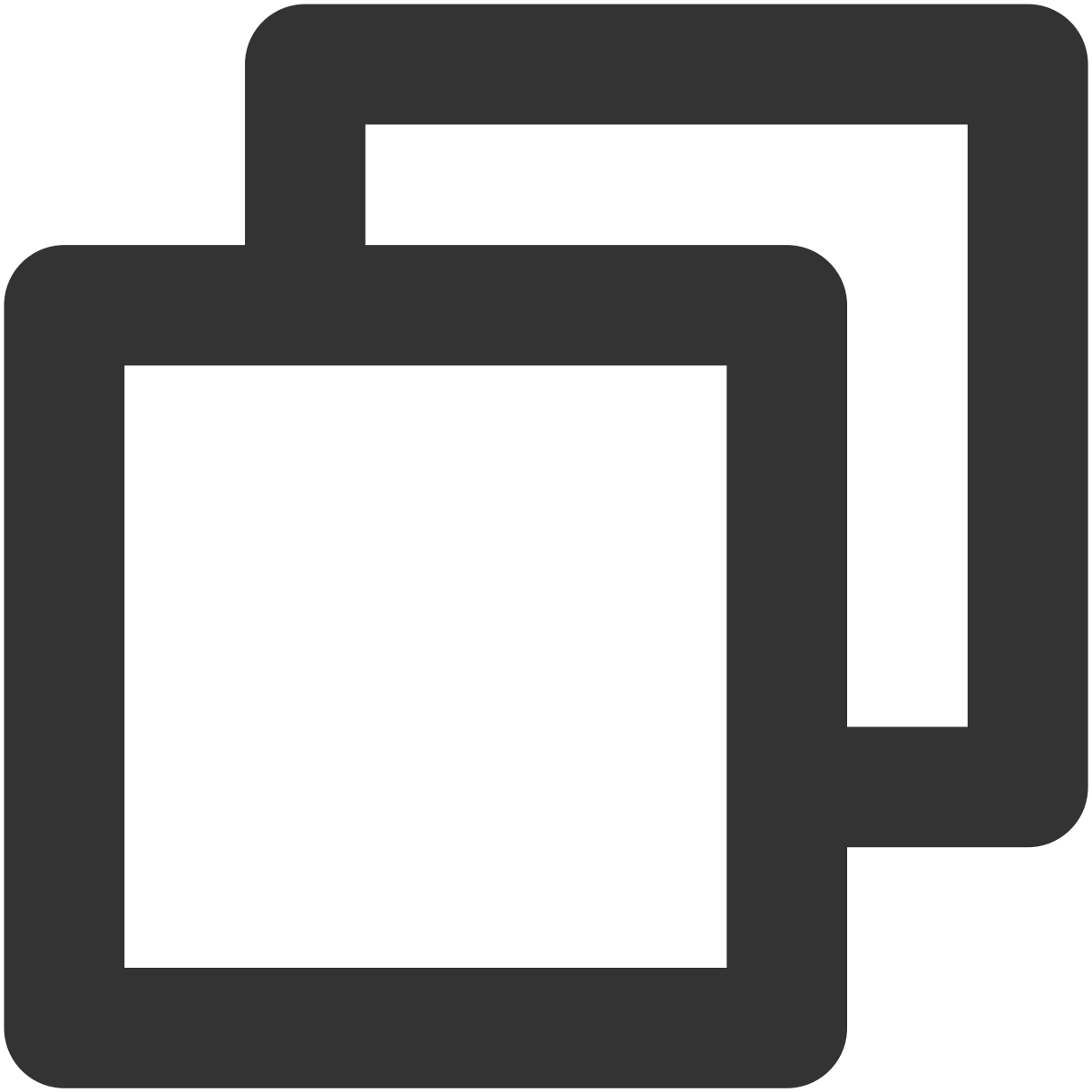
```
CALL `Catalog`.`system`.expire_snapshots(table_name, [older_than], [retain_last], [
```

Example:

```
CALL `DataLakeCatalog`.`system`.expire_snapshots('validation.dempts', TIMESTAMP '20
```

remove_orphan_files

Remove metadata files that are no longer referenced.



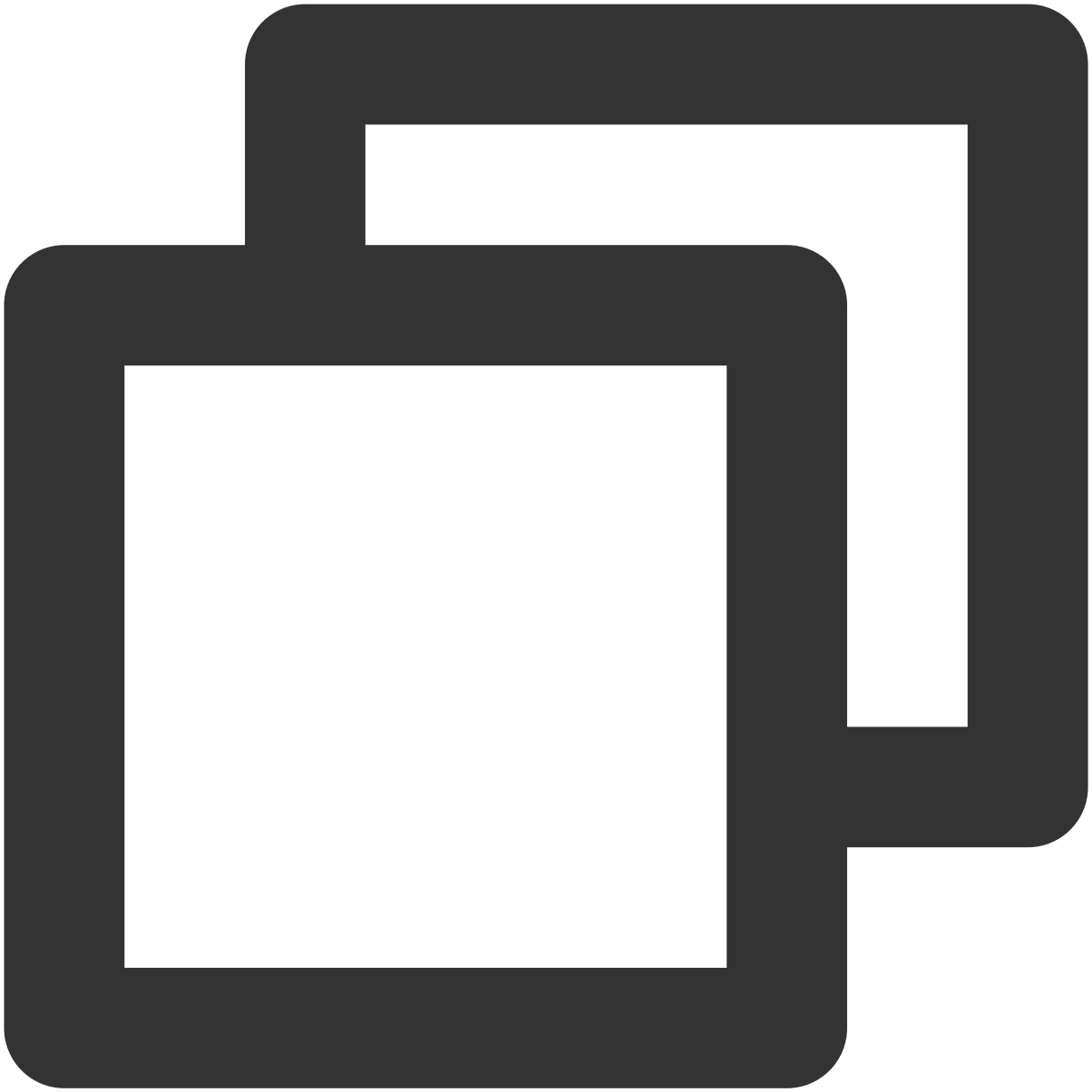
```
CALL `Catalog`.`system`.remove_orphan_files(table_name, [older_than], [location], [
```

Example:

```
CALL `DataLakeCatalog`.`system`.remove_orphan_files(`table`=>'validation.dempts', d
CALL `DataLakeCatalog`.`system`.remove_orphan_files(`table`=>'validation.dempts', `
CALL `DataLakeCatalog`.`system`.remove_orphan_files('validation.dempts', TIMESTAMP
```

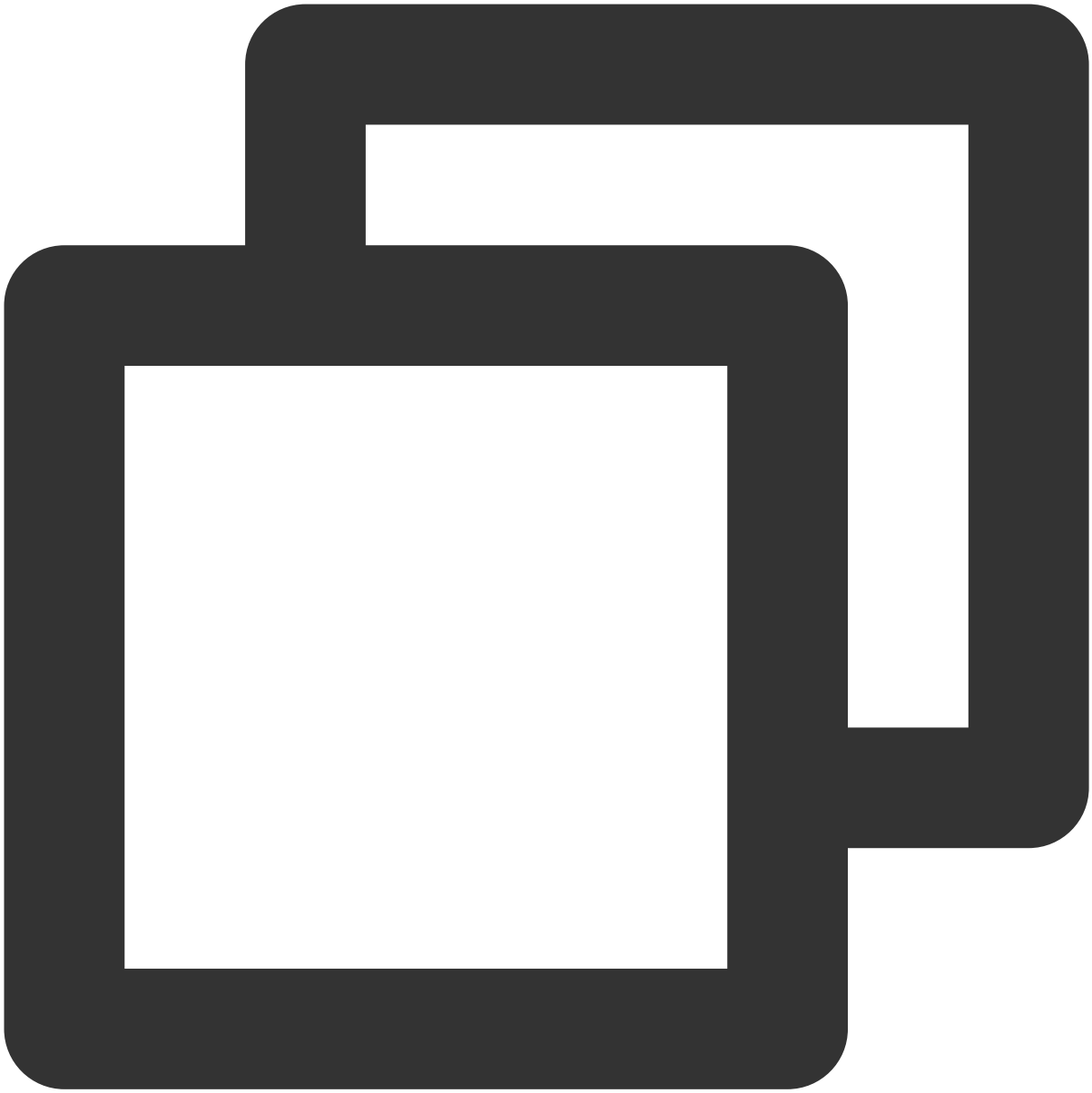
remove_orphan_files

Remove metadata files that are no longer referenced.



```
CALL `Catalog`.`system`.remove_orphan_files(table_name, [older_than], [location], [
```

Example:



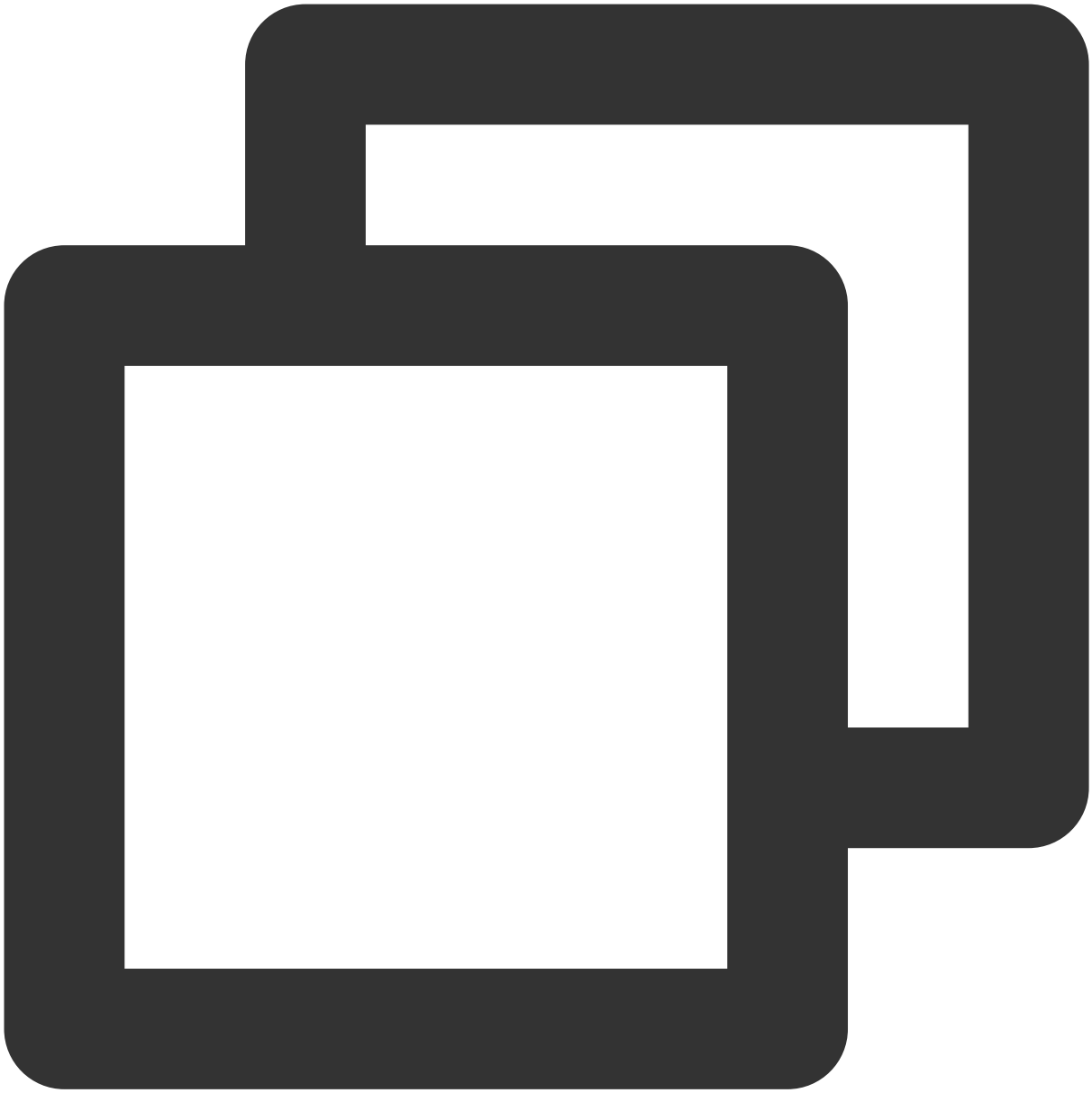
```
CALL `DataLakeCatalog`.`system`.remove_orphan_files(`table`=>'validation.dempts', d
CALL `DataLakeCatalog`.`system`.remove_orphan_files(`table`=>'validation.dempts', `
CALL `DataLakeCatalog`.`system`.remove_orphan_files('validation.dempts', TIMESTAMP
```

rewrite_data_files

Merge and rewrite data files, i.e. merging small data files.

```
CALL `Catalog`.`system`.rewrite_data_files(table_name, [strategy], [sort_order], [o
```

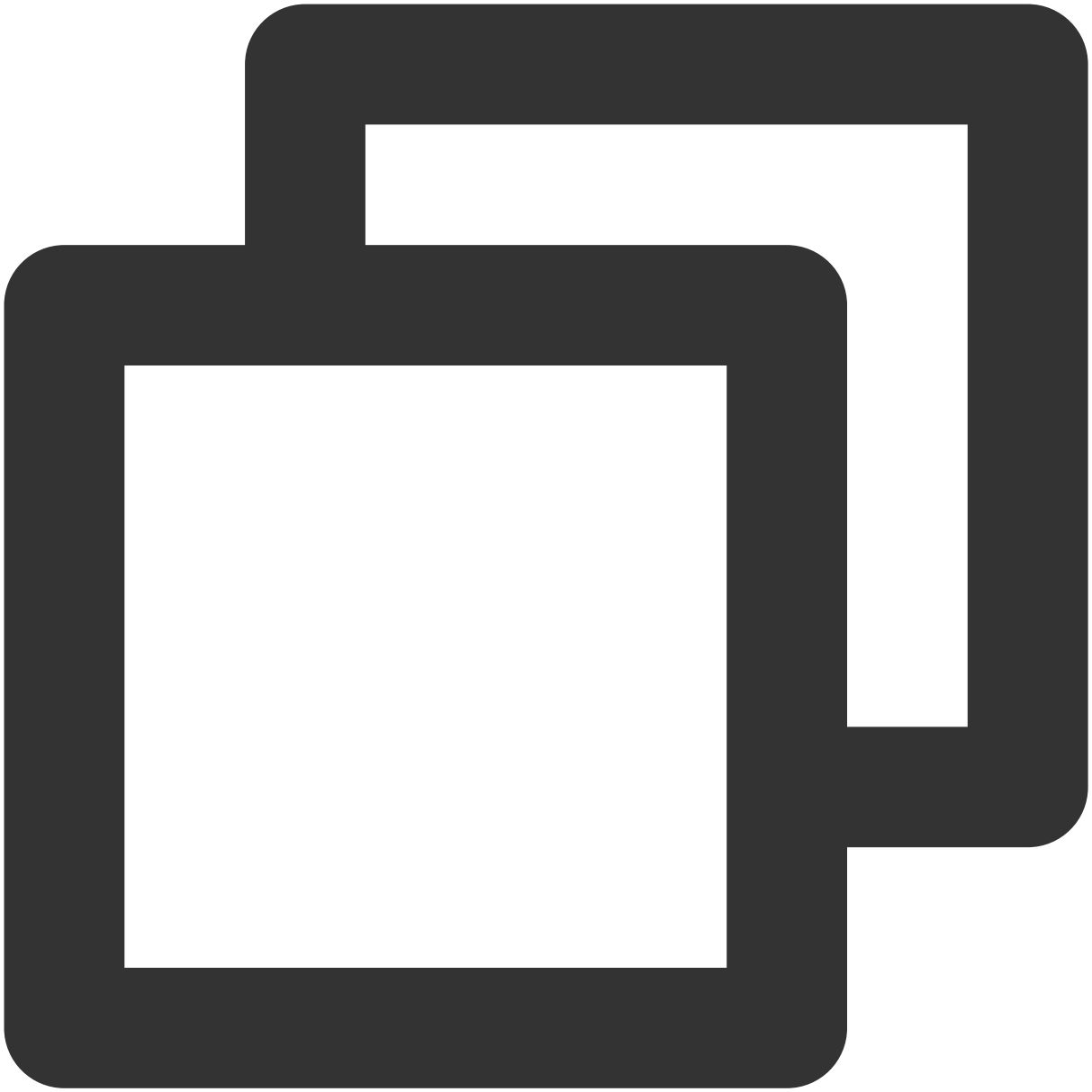
Example:



```
CALL `DataLakeCatalog`.`system`.rewrite_data_files('validation.dempts');
CALL `DataLakeCatalog`.`system`.rewrite_data_files(`table`=>'validation.dempts', `s
CALL `DataLakeCatalog`.`system`.rewrite_data_files(`table`=>'validation.dempts', `o
CALL `DataLakeCatalog`.`system`.rewrite_data_files(`table`=>'validation.dempts', `w
```

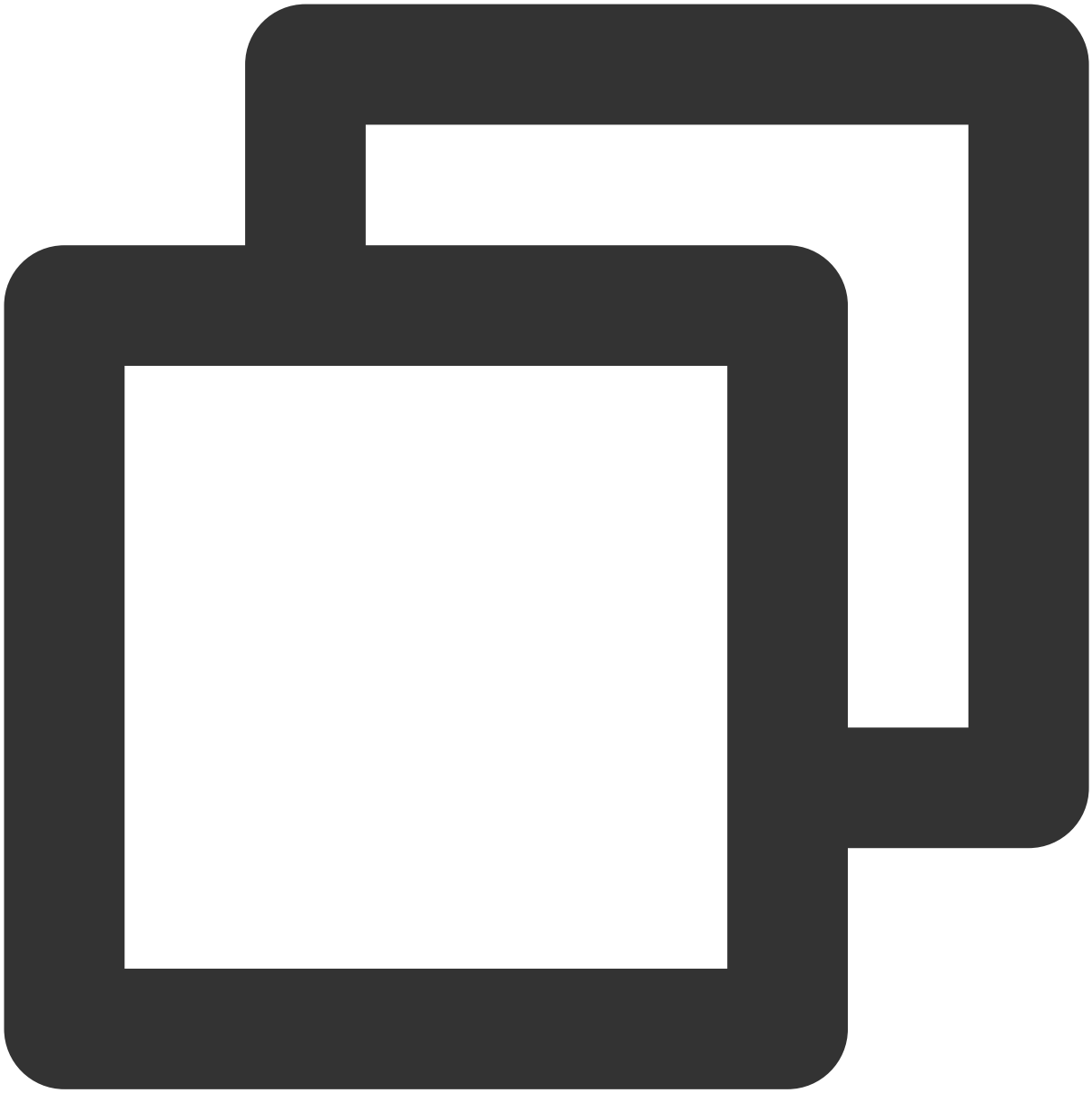
rewrite_manifests

Merge and rewrite manifests files.



```
CALL `Catalog`.`system`.rewrite_manifests(table_name, [using_caching]);
```

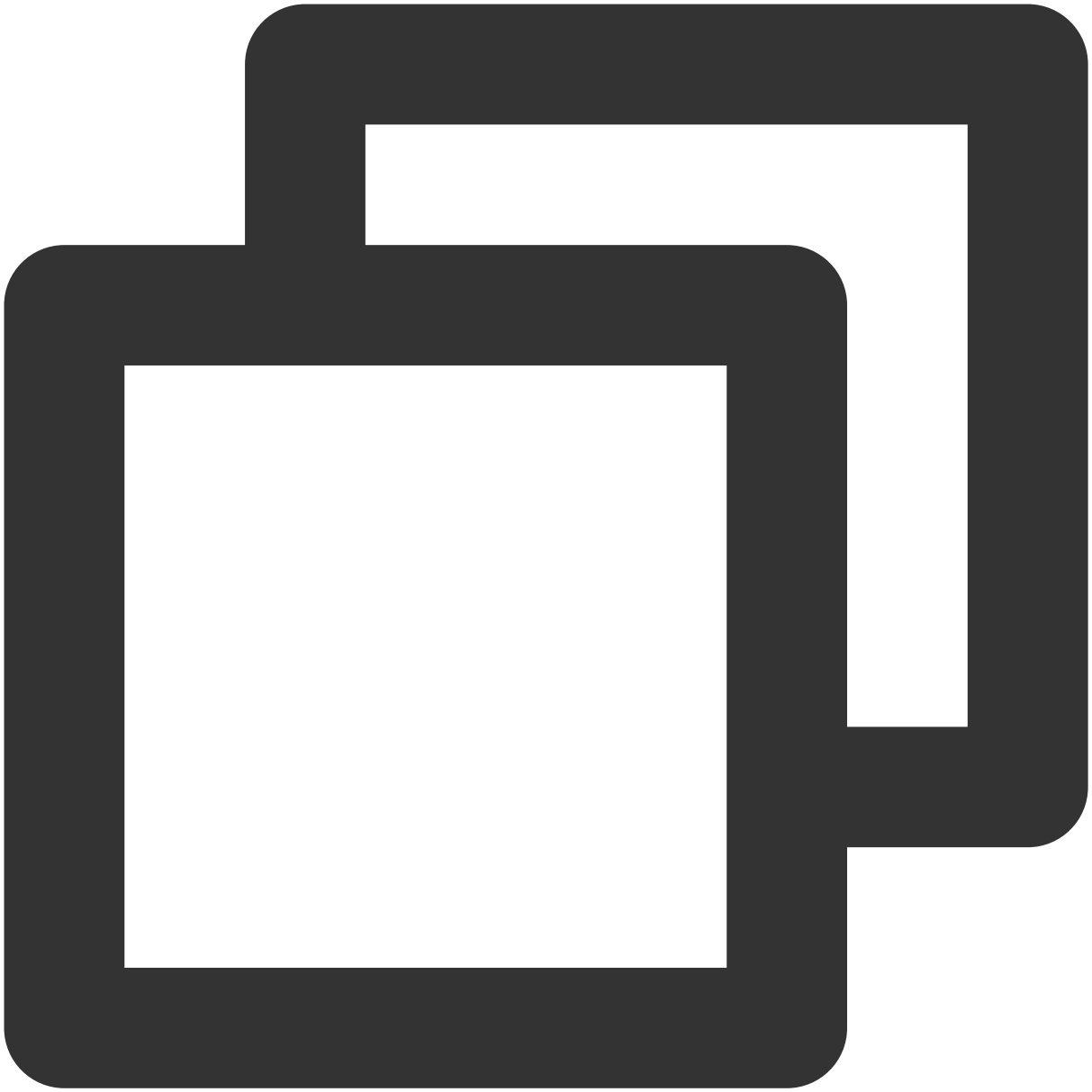
Example:



```
CALL `DataLakeCatalog`.`system`.rewrite_manifests('validation.dempts');
CALL `DataLakeCatalog`.`system`.rewrite_manifests('validation.dempts', FALSE);
```

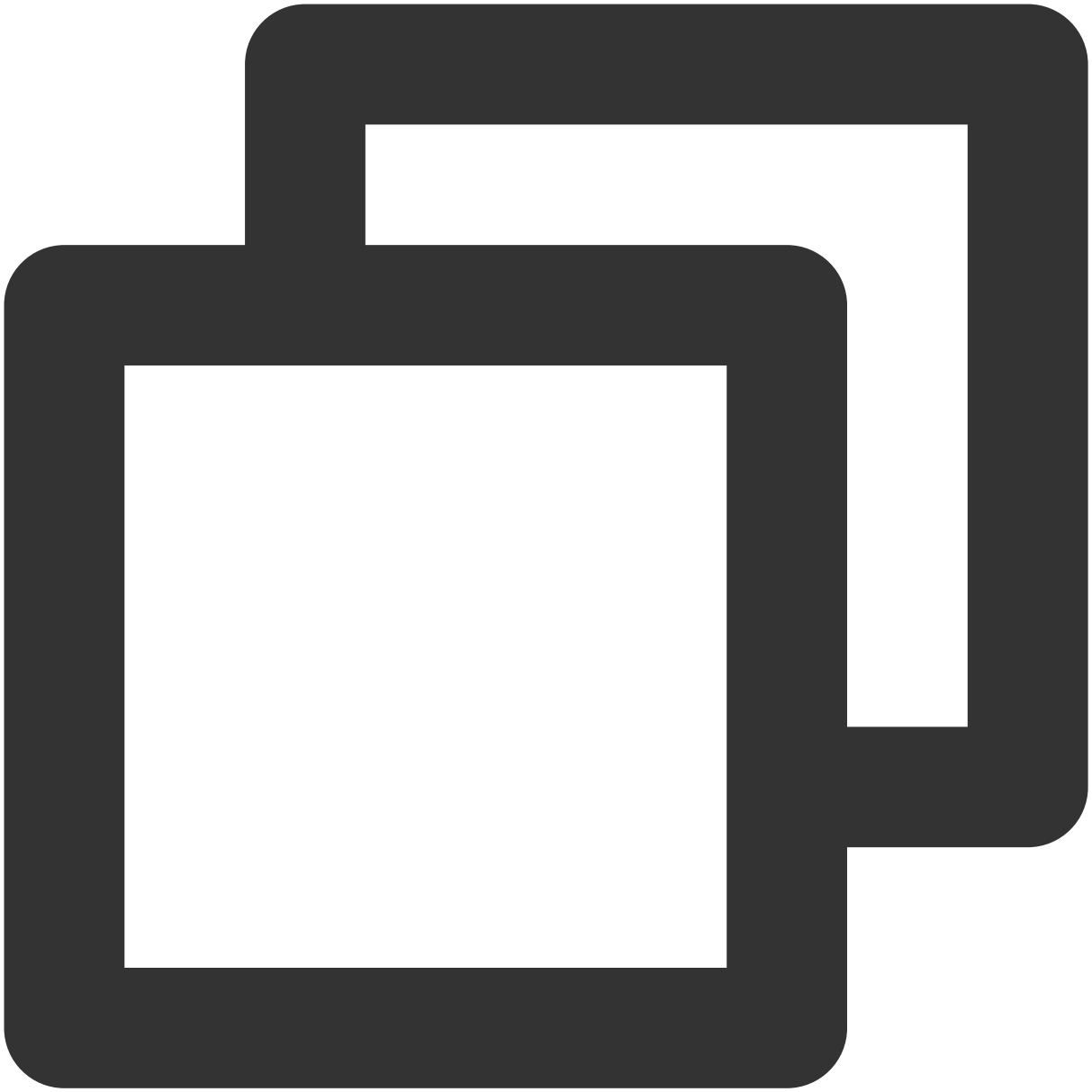
ancestors_of

Get the lineage information of a snapshot.



```
CALL `Catalog`.`system`.ancestors_of(table_name, [snapshot_id]);
```

Example:



```
CALL `DataLakeCatalog`.`system`.ancestors_of('validation.dempts');
CALL `DataLakeCatalog`.`system`.ancestors_of('validation.dempts', 1);
```

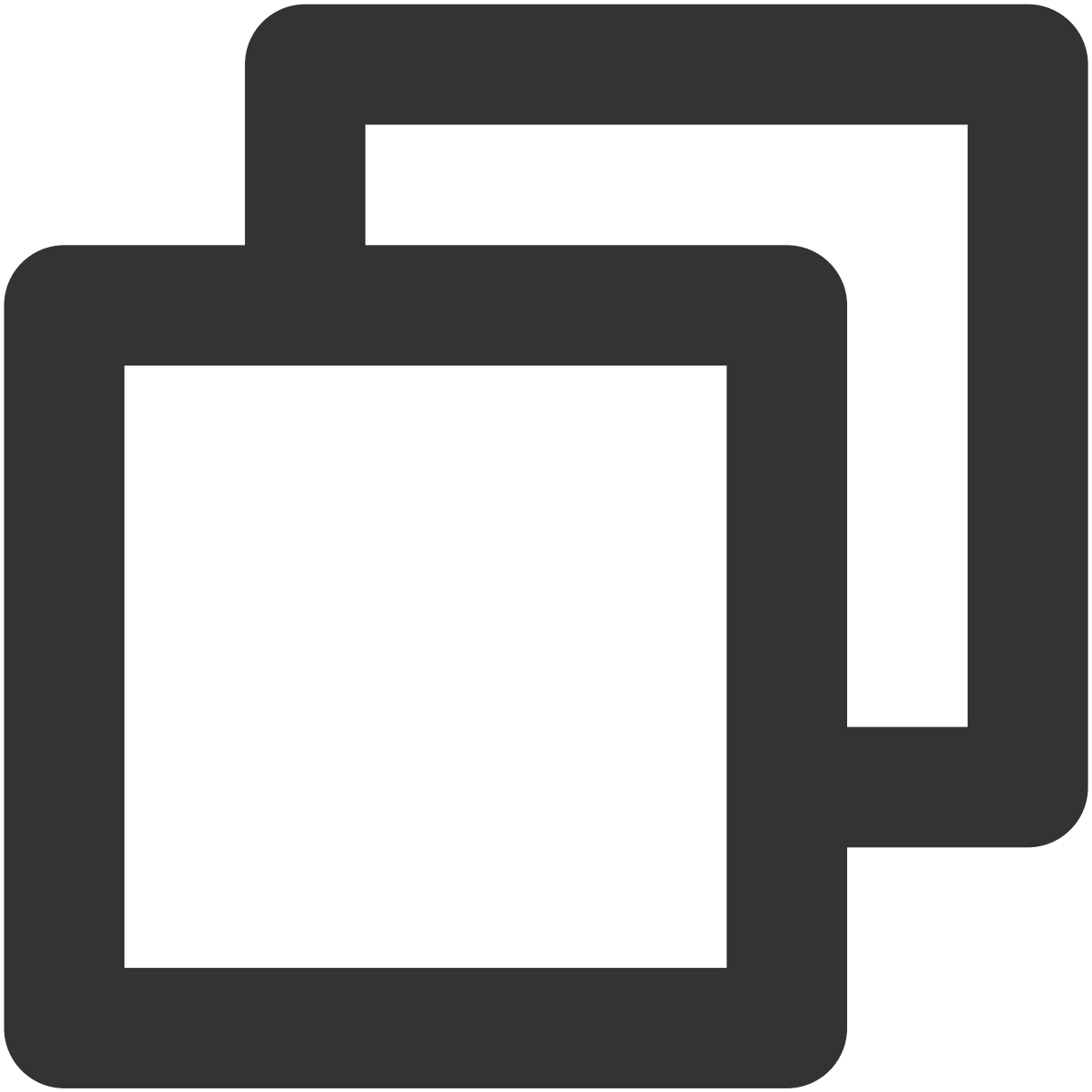
Data Table Migration Management

Note

The original table must be a Hive table or Spark table.

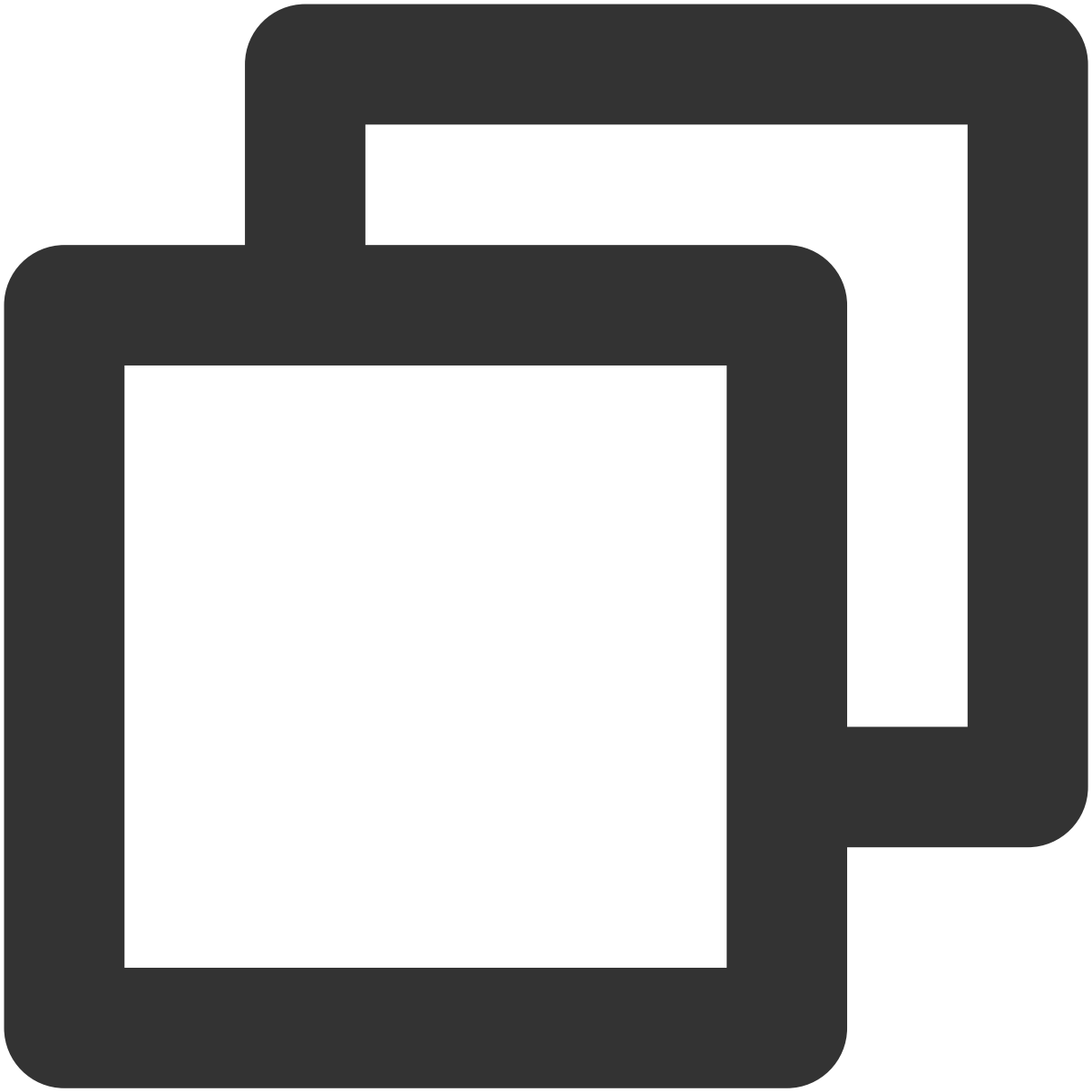
snapshot

Create lightweight temporary tables based on the original tables. The temporary tables directly reuse the snapshots of the original tables.



```
CALL `Catalog`.`system`.snapshot(source_table, table, [location], [properties]);
```

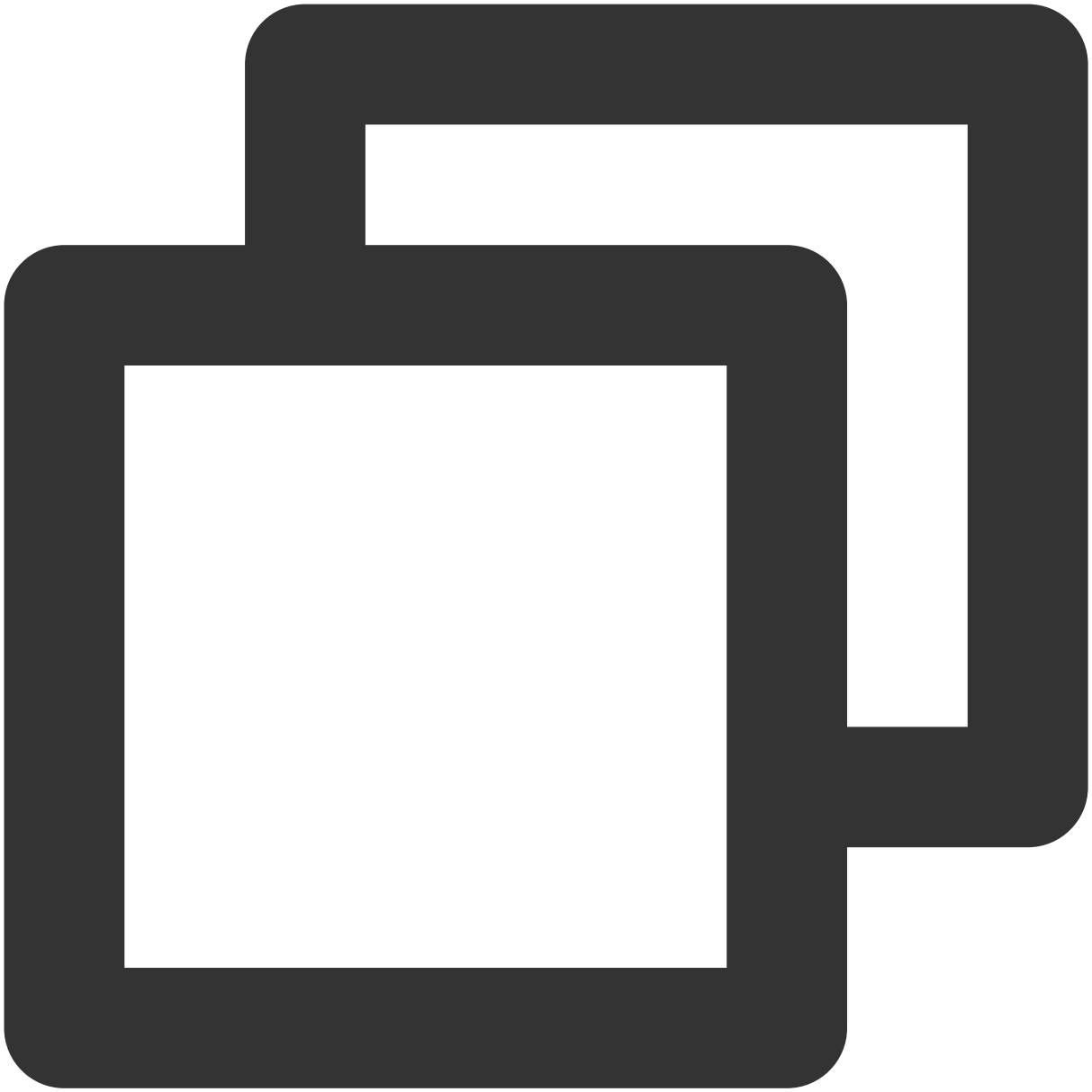
Example:



```
CALL `DataLakeCatalog`.`system`.snapshot('validation.table_01', 'validation.snap');
CALL `DataLakeCatalog`.`system`.snapshot('validation.table_01', 'validation.snap2',
```

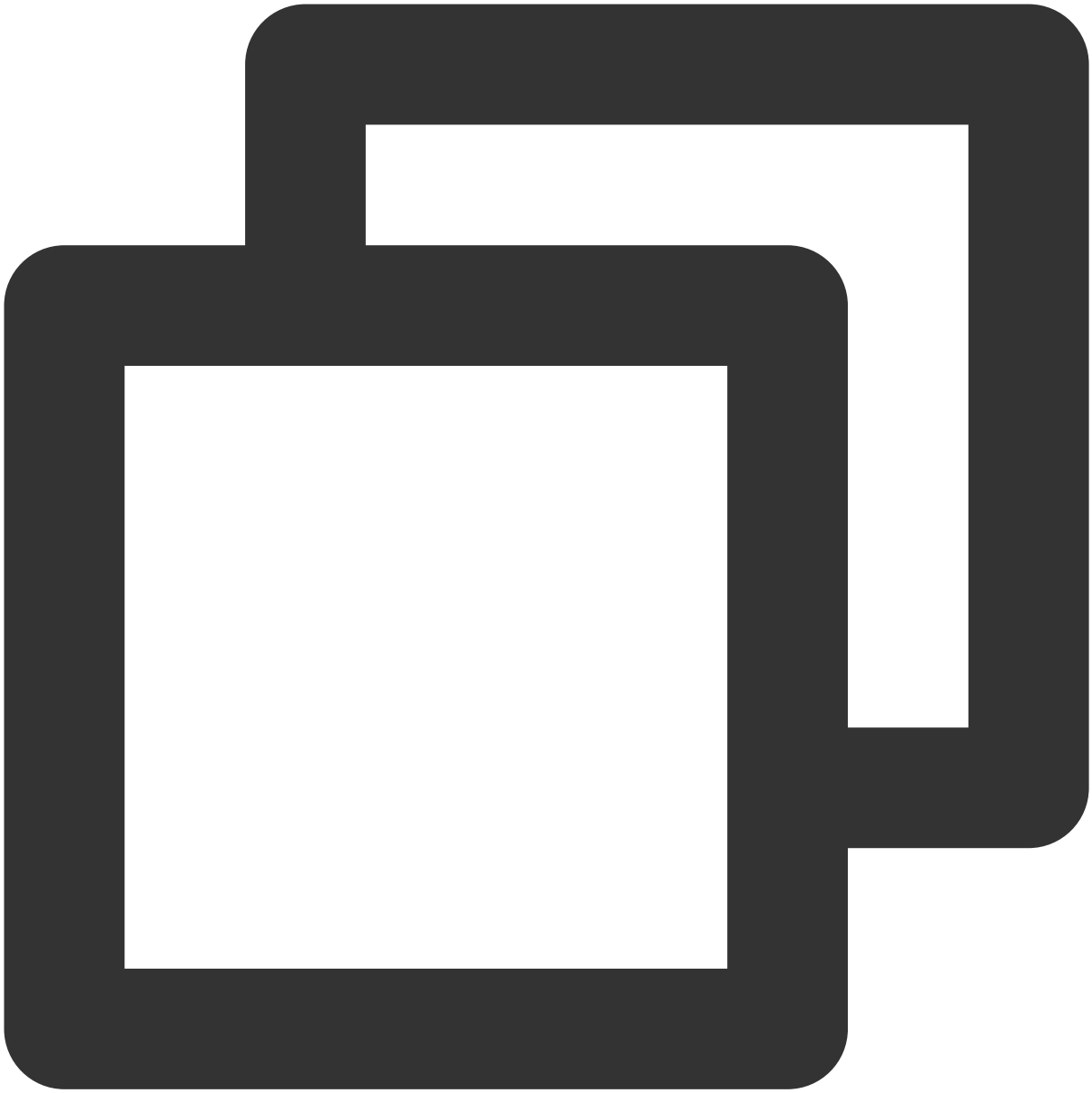
migrate

Update and replace table attributes.



```
CALL `Catalog`.`system`.migrate(table, [properties]);
```

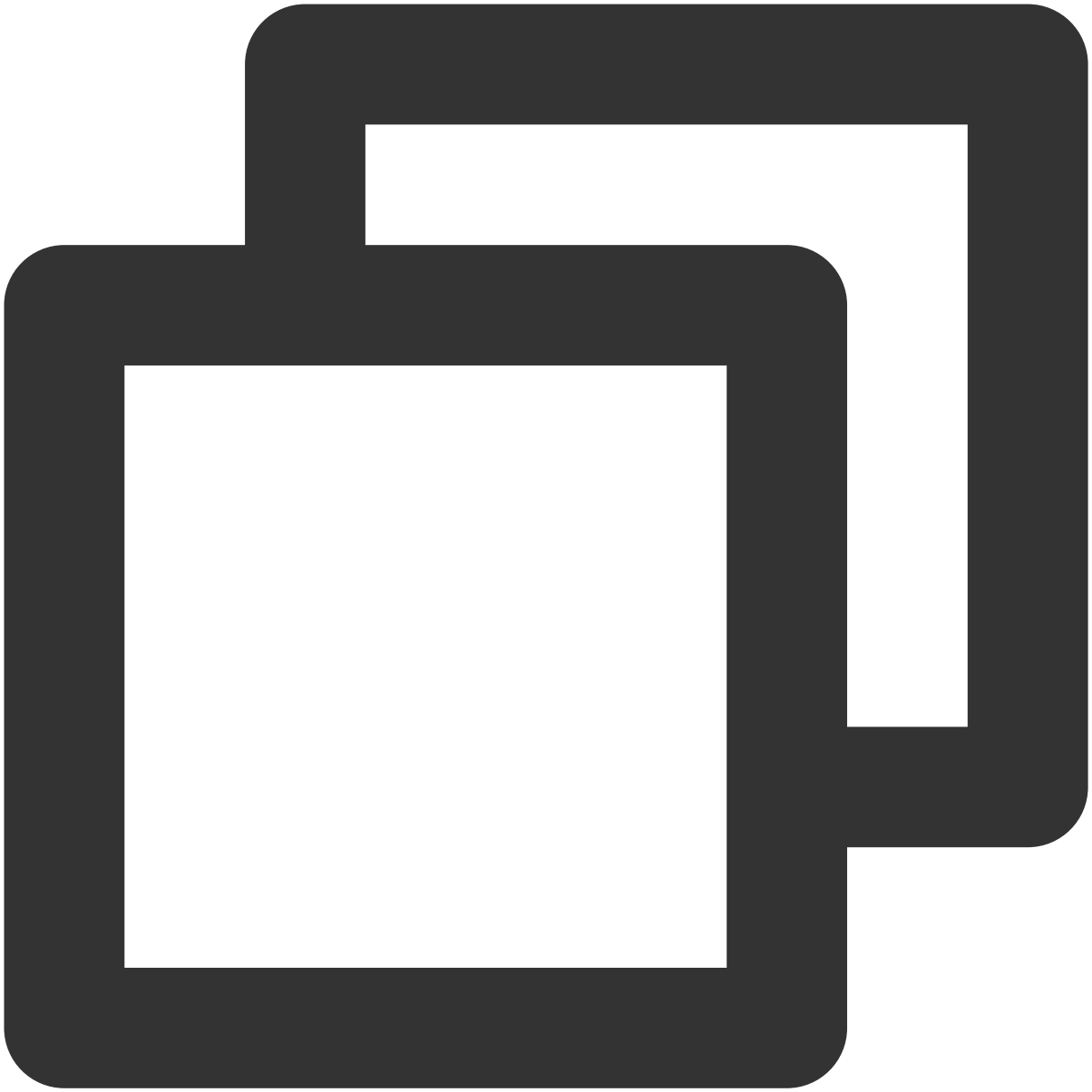
Example:



```
CALL `DataLakeCatalog`.`system`.migrate('validation.table_01');
CALL `DataLakeCatalog`.`system`.migrate('validation.table_01', map('data', 'name'))
```

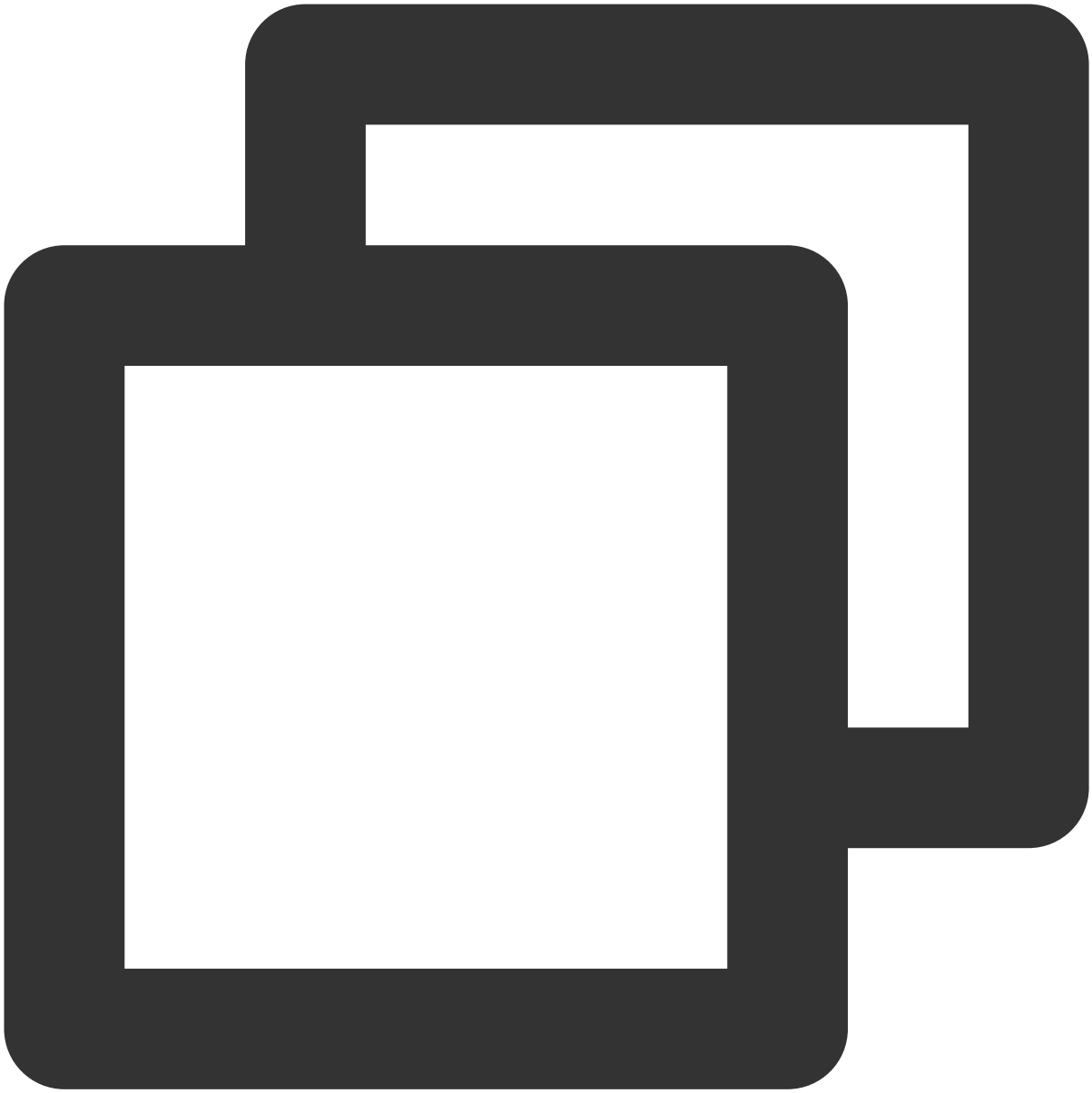
add_files

Load data files directly from Hive, and you can specify data files to a specific partition.



```
CALL `Catalog`.`system`.add_files(table, source_table, [partition_filter]);
```

Example:



```
CALL `DataLakeCatalog`.`system`.add_files(`table`=>'validation.table_02', `source_t
CALL `DataLakeCatalog`.`system`.add_files(`table`=>'validation.table_02', `source_t
```

Differences in Statement Between Iceberg External Tables and Native Tables

Last updated：2024-08-07 17:29:26

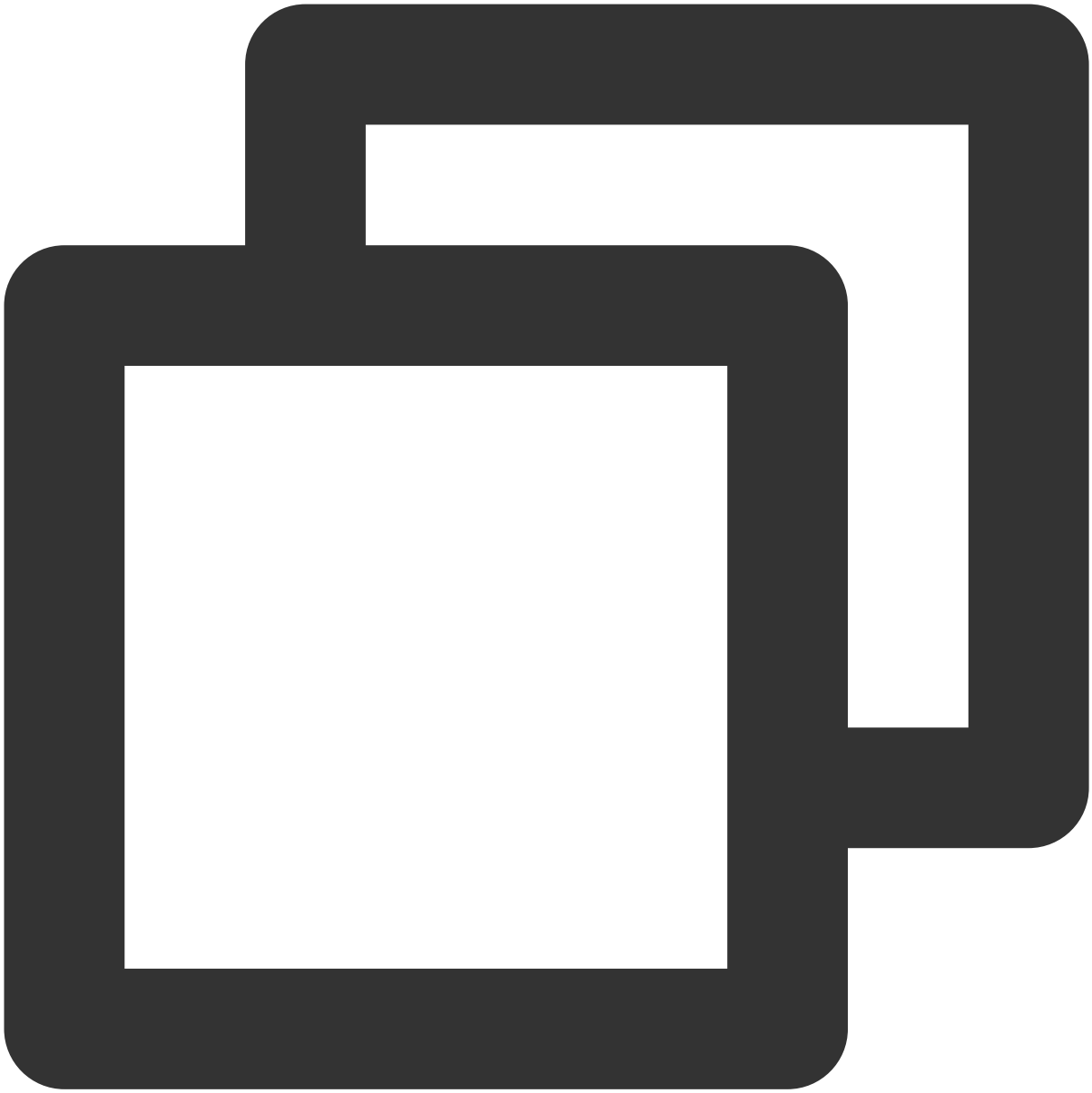
Data Lake Compute (DLC) uses Iceberg statement version 0.13.1. For detailed statement description, see the Iceberg Official Documentation.

When you use Iceberg external tables, the SQL statement differs from that of native Iceberg tables in the following ways.

CREATE TABLE

Native Tables

Statement



```
CREATE TABLE [ IF NOT EXISTS ] table_identifier
    ( col_name[:] col_type [ COMMENT col_comment ], ... )
[ COMMENT table_comment ]
[ PARTITIONED BY ( col_name1, transform(col_name2), ... ) ]
```

Example

```
CREATE TABLE dempts(
    id bigint COMMENT 'id number',
    num int,
    eno float,
    dno double,
    cno decimal(9,3),
    flag  boolean,
    data string,
    ts_year timestamp,
    date_month date,
    bno binary,
```

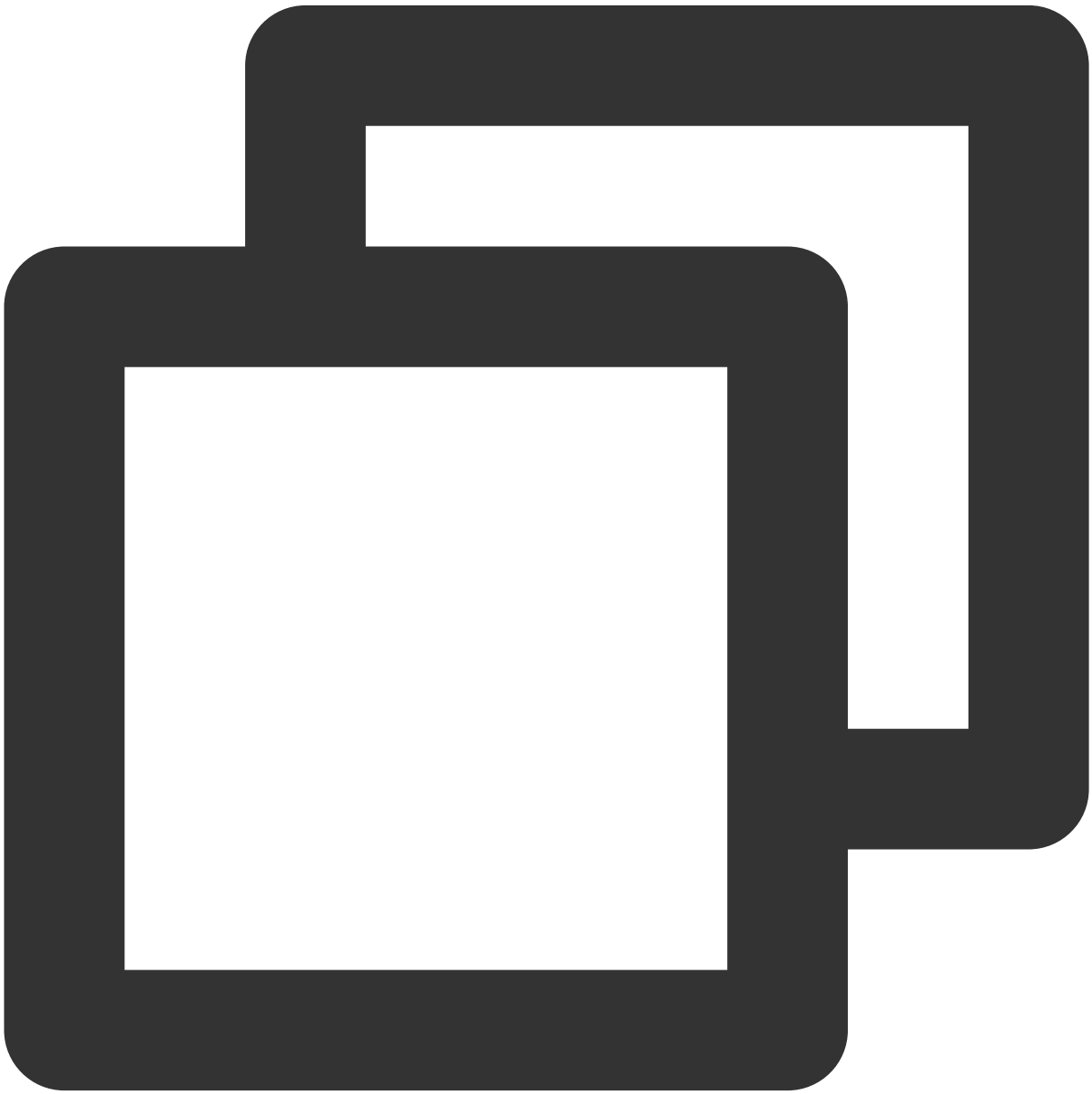
```
    point struct<x: double, y: double>,
    points array<struct<x: double, y: double>>,
    pointmaps map<struct<x: int>, struct<a: int>>
    )
COMMENT 'table documentation'
PARTITIONED BY (bucket(16,id), years(ts_year), months(date_month), identity(bno),
```

External Tables

Statement

```
CREATE TABLE [ IF NOT EXISTS ] table_identifier
    ( col_name[:] col_type [ COMMENT col_comment ], ... )
USING iceberg
    [ COMMENT table_comment ]
    [ PARTITIONED BY ( col_name1, transform(col_name2), ... ) ]
    [ LOCATION path ]
    [ TBLPROPERTIES ( property_name=property_value, ... ) ]
```

Example



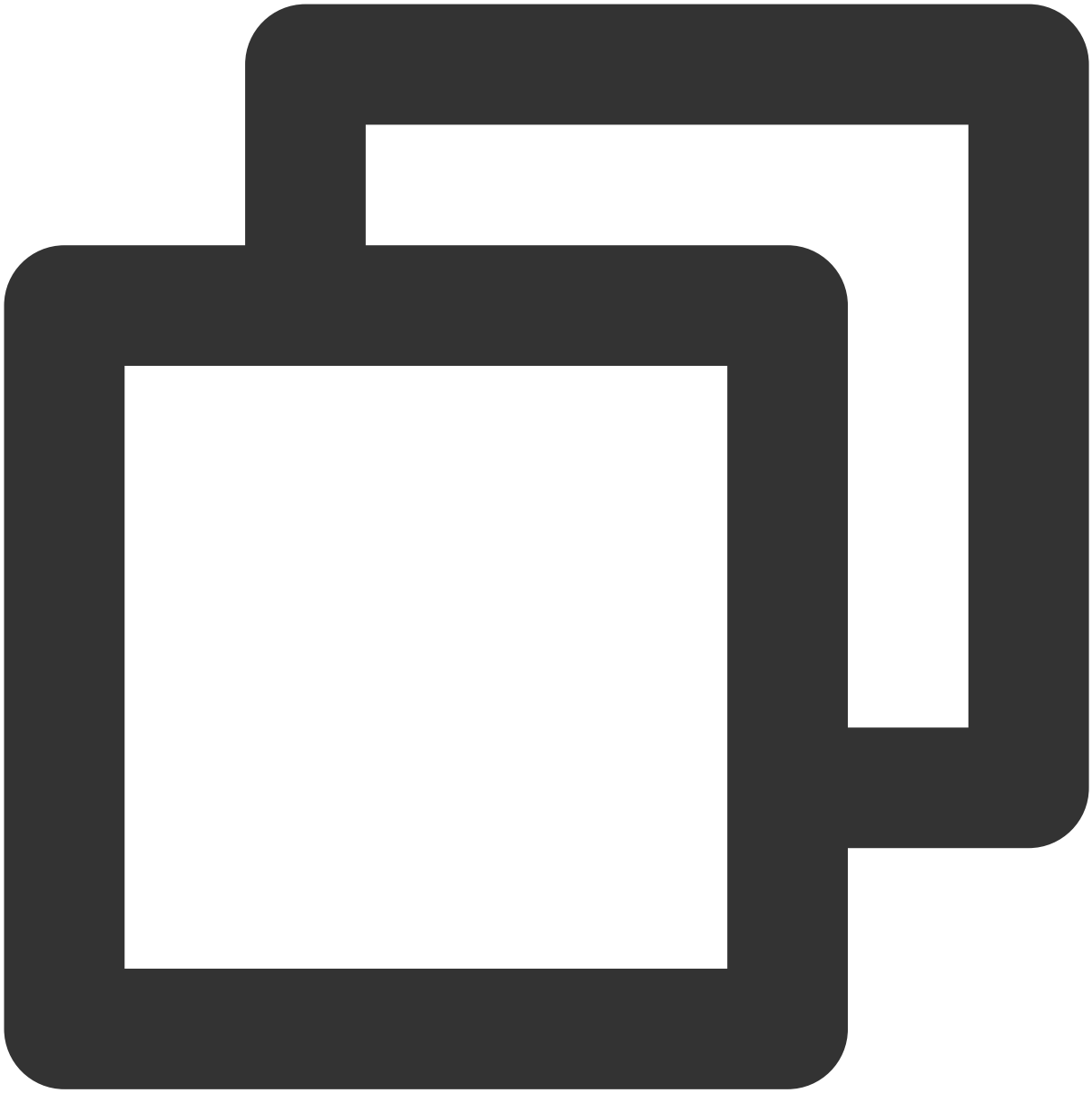
```
CREATE TABLE dempts(
```

```
    id bigint COMMENT 'id number',
    num int,
    eno float,
    dno double,
    cno decimal(9,3),
    flag  boolean,
    data string,
    ts_year timestamp,
    date_month date,
    bno binary,
    point struct<x: double, y: double>,
    points array<struct<x: double, y: double>>,
    pointmaps map<struct<x: int>, struct<a: int>>
    )
USING iceberg
COMMENT 'table documentation'
PARTITIONED BY (bucket(16,id), years(ts_year), months(date_month), identity(bno),
LOCATION 'cosn://rickytest-1305424723/channing-test/loc'
TBLPROPERTIES ('write.format.default'='orc');
```

CREATE TABLE AS SELECT

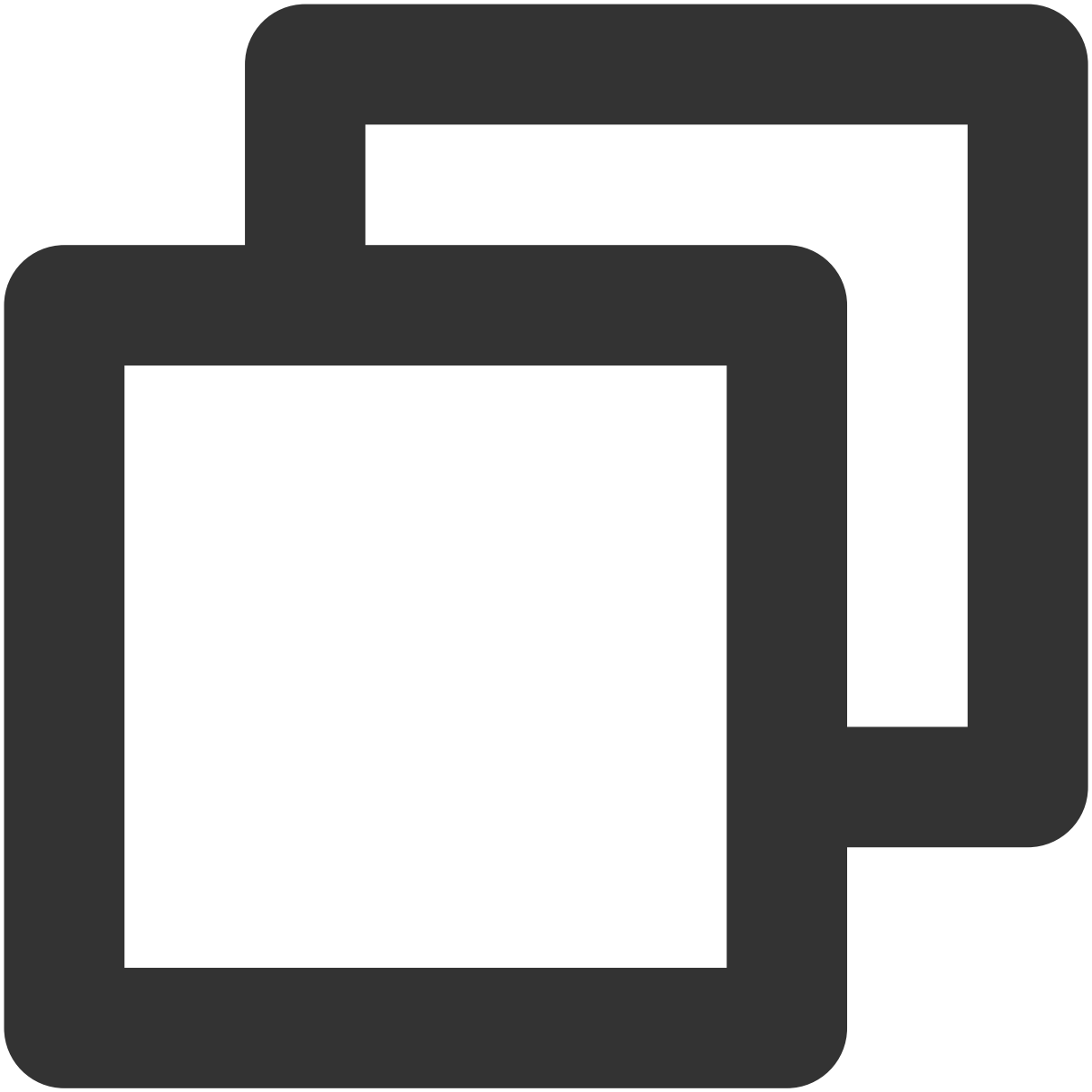
Native Tables

Statement



```
CREATE TABLE [ IF NOT EXISTS ] table_identifier
    [ COMMENT table_comment ]
    [ PARTITIONED BY ( col_name1, transform(col_name2), ... ) ]
AS select_statement
```

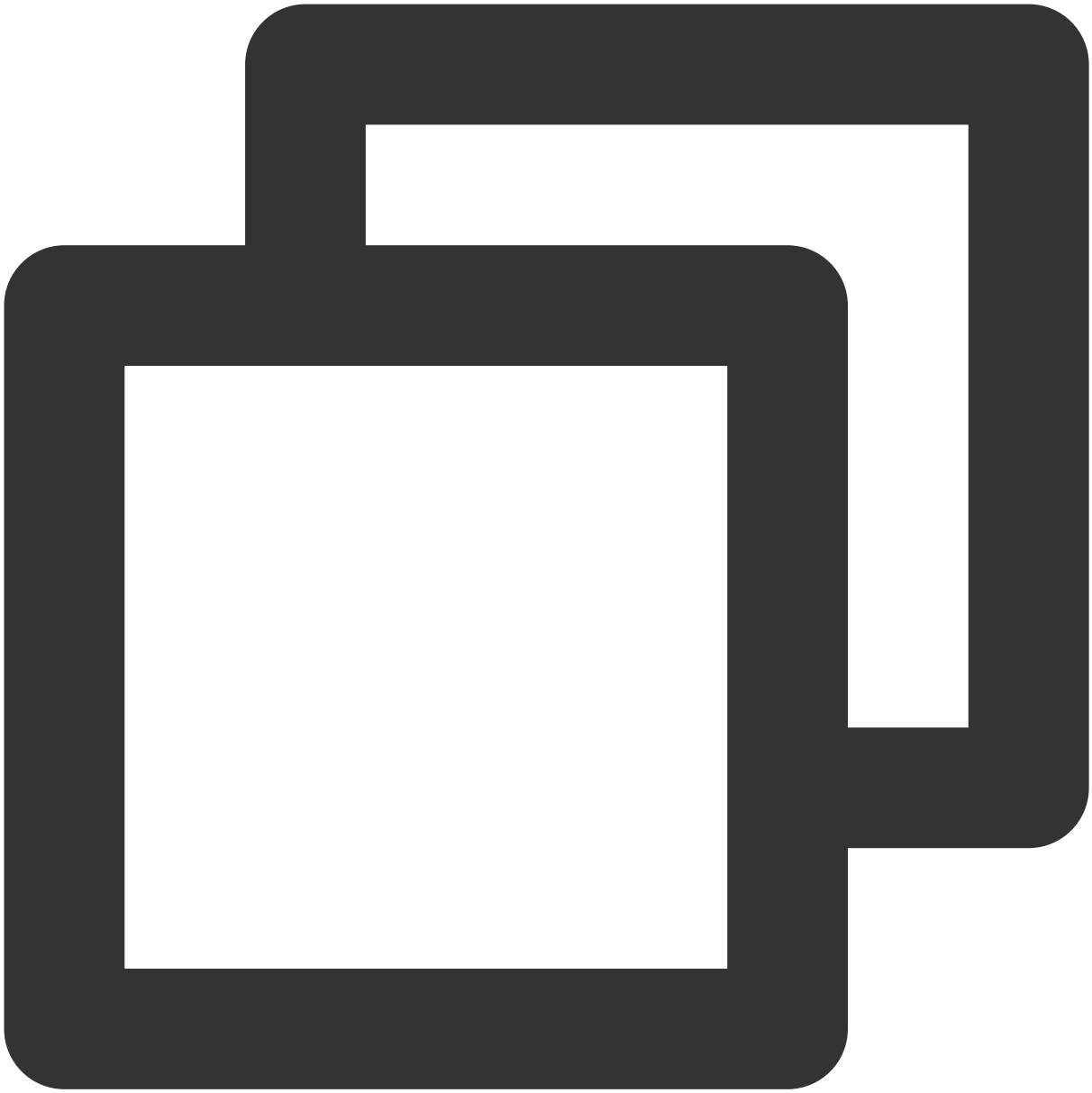
Example



```
CREATE TABLE IF NOT EXISTS dempts_copy
COMMENT 'table create as select'
PARTITIONED BY (eno, dno)
AS SELECT * from dempts;
```

External Tables

Statement



```
CREATE TABLE [ IF NOT EXISTS ] table_identifier
USING iceberg
    [ COMMENT table_comment ]
    [ PARTITIONED BY ( col_name1, transform(col_name2), ... ) ]
    [ LOCATION path ]
    [ TBLPROPERTIES ( property_name=property_value, ... ) ]
AS select_statement
```

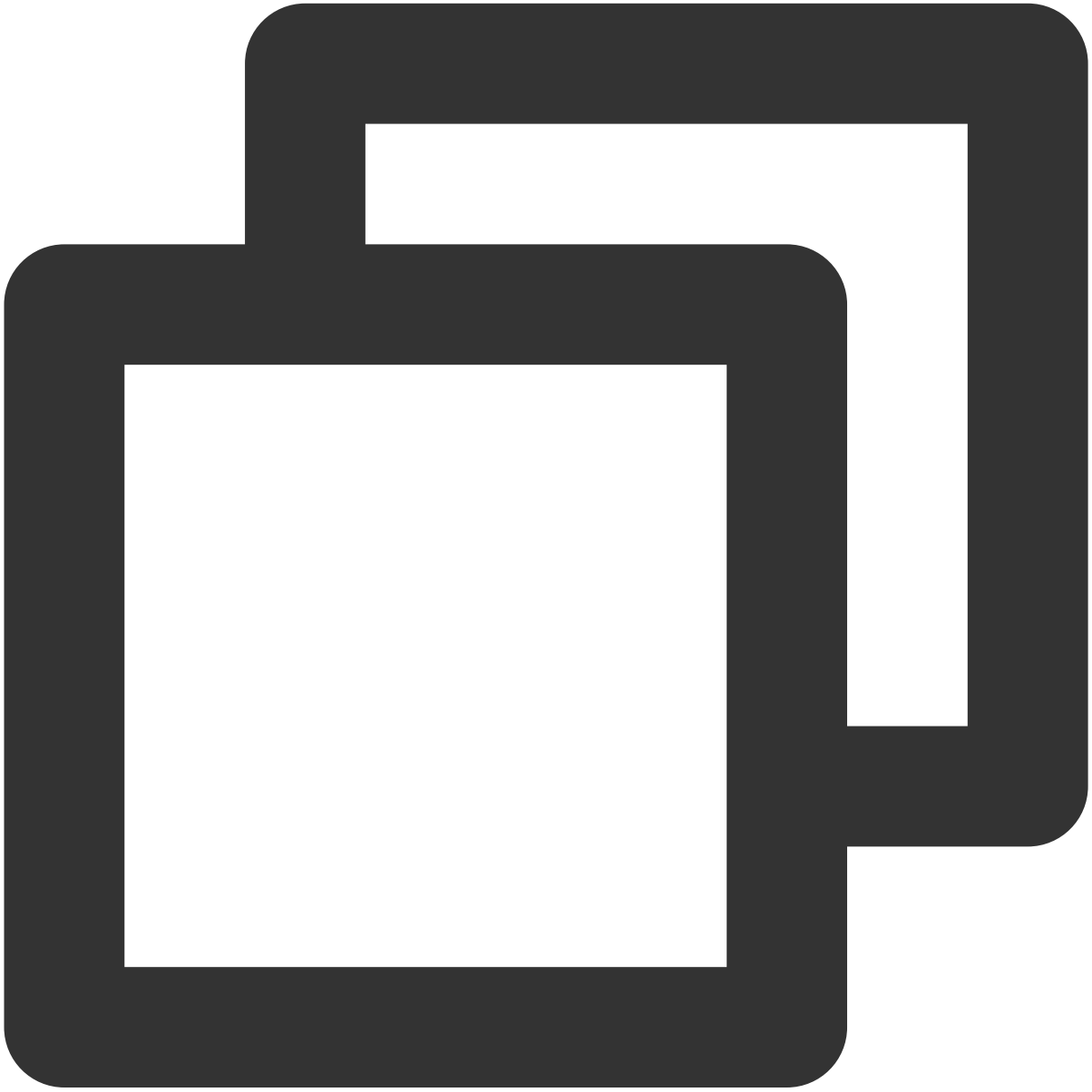
Example

```
CREATE TABLE dempts_copy
USING iceberg
COMMENT 'table create as select'
PARTITIONED BY (eno, dno)
LOCATION 'cosn://rickytest-1305424723/channing-test/loc'
TBLPROPERTIES ('write.format.default'='avro')
AS SELECT * from dempts;
```

REPLACE TABLE AS SELECT

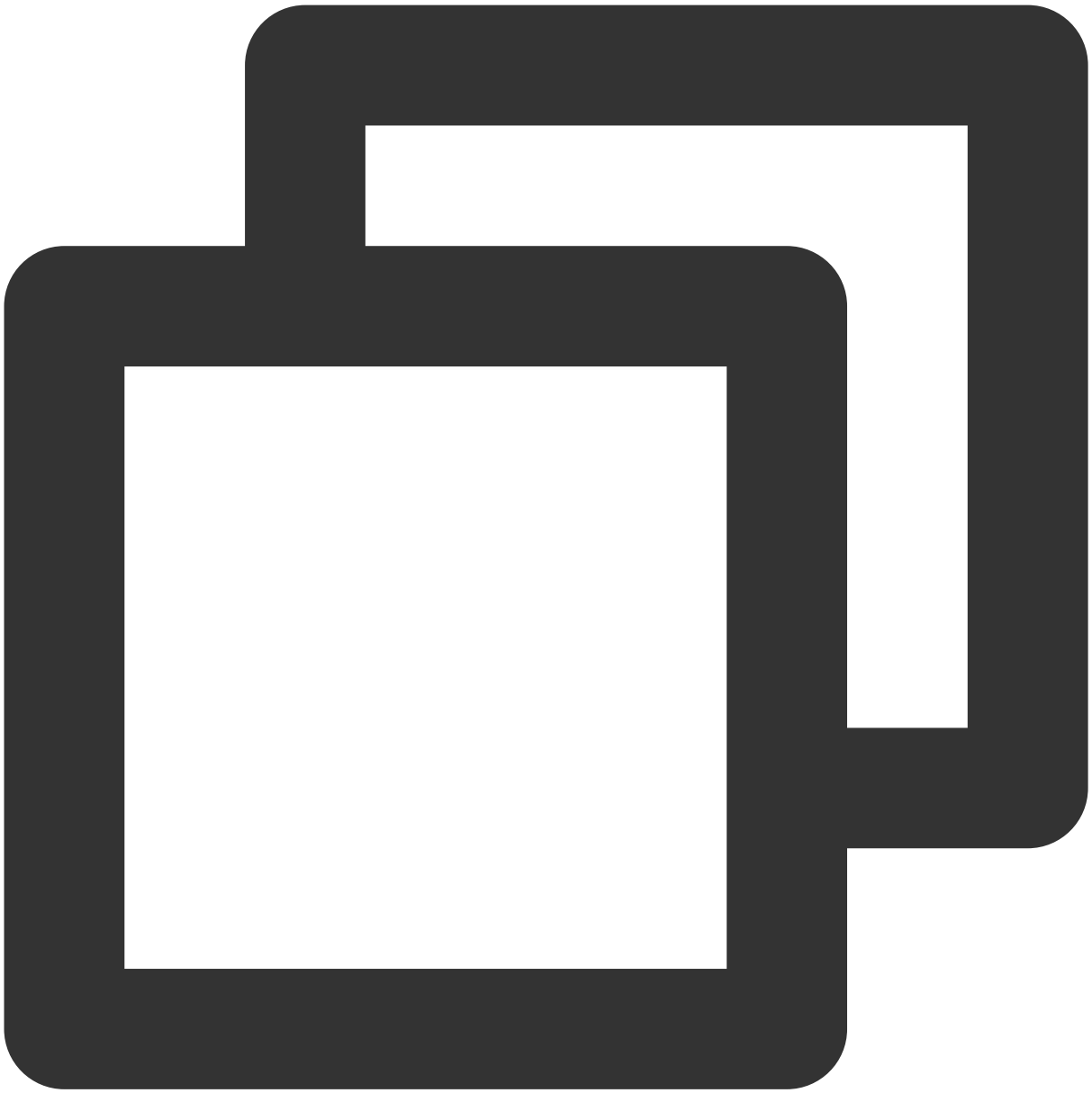
Native Tables

Statement



```
CREATE OR REPLACE TABLE table_identifier
    [ COMMENT table_comment ]
    [ PARTITIONED BY ( col_name1, transform(col_name2), ... ) ]
AS select_statement
```

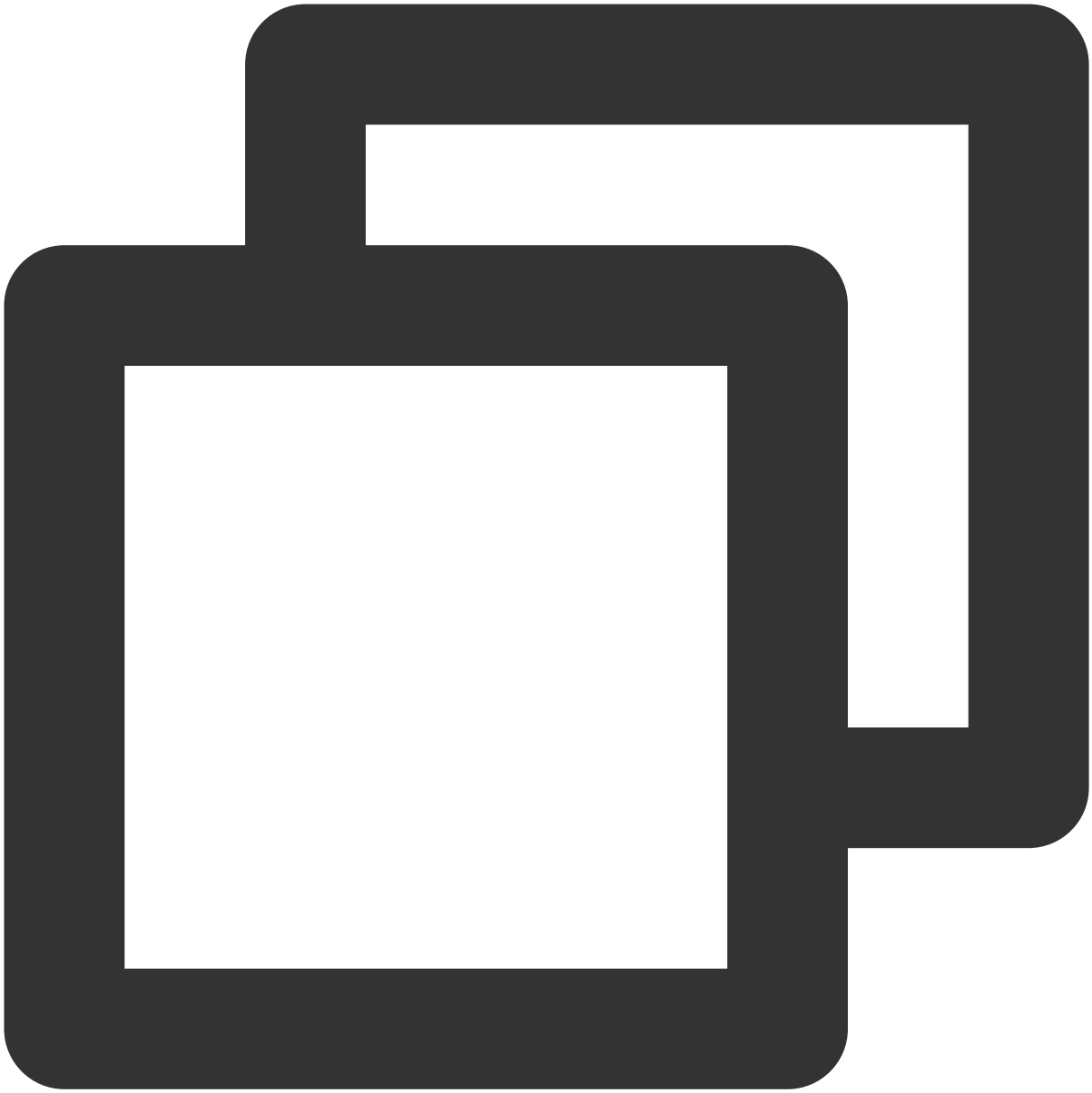
Example



```
CREATE OR REPLACE TABLE dempts_replace
COMMENT 'table create as replace'
PARTITIONED BY (eno, bucket(10, num))
AS SELECT * from dempts;
```

External Tables

Statement



```
CREATE [OR REPLACE] TABLE table_identifier
USING iceberg
    [ COMMENT table_comment ]
    [ PARTITIONED BY ( col_name1, transform(col_name2), ... ) ]
    [ LOCATION path ]
    [ TBLPROPERTIES ( property_name=property_value, ... ) ]
AS select_statement
```

Example

```
CREATE OR REPLACE TABLE dempts_replace
USING iceberg
COMMENT 'table create as replace'
PARTITIONED BY (eno, dno)
LOCATION 'cosn://rickytest-1305424723/channing-test/loc'
TBLPROPERTIES ('write.format.default'='avro')
AS SELECT * from dempts;
```

Procedure

Caution

The original table for migration must be a Hive table or a Spark table.

Native Tables

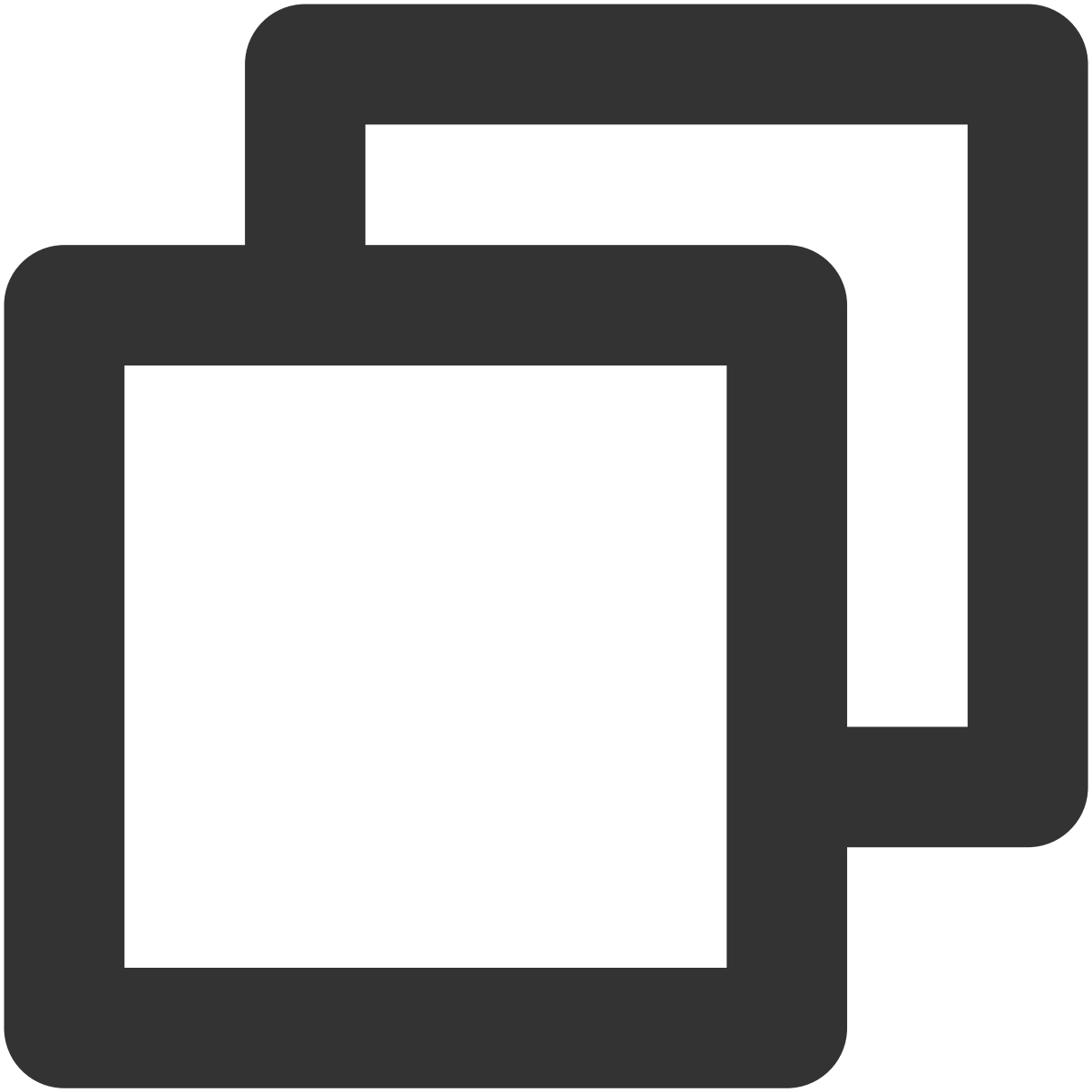
Not supported currently.

External Tables

snapshot

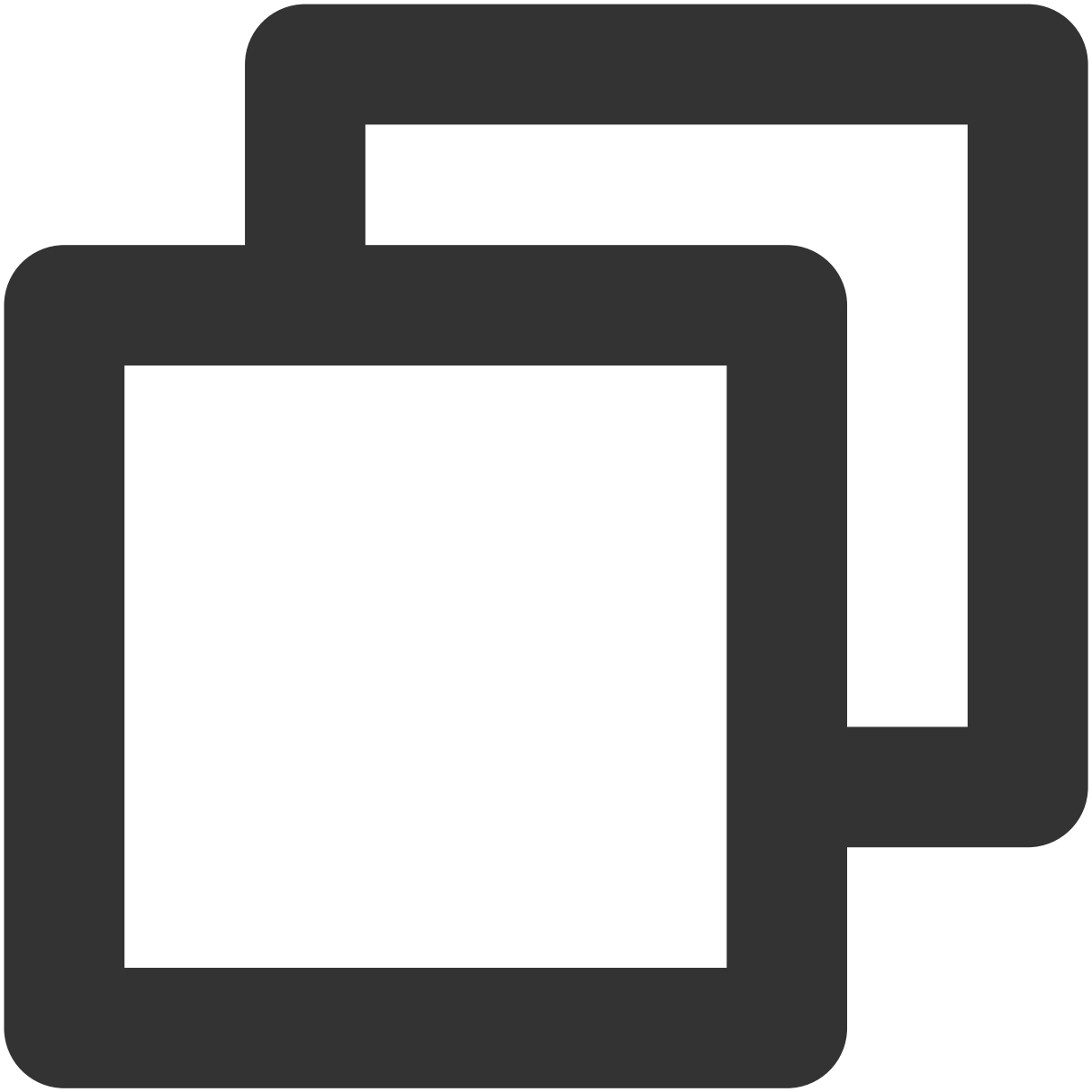
Create lightweight temporary tables based on the original tables. The temporary tables directly reuse the snapshots of the original tables.

Statement



```
CALL `Catalog`.`system`.snapshot(source_table, table, [location], [properties]);
```

Example

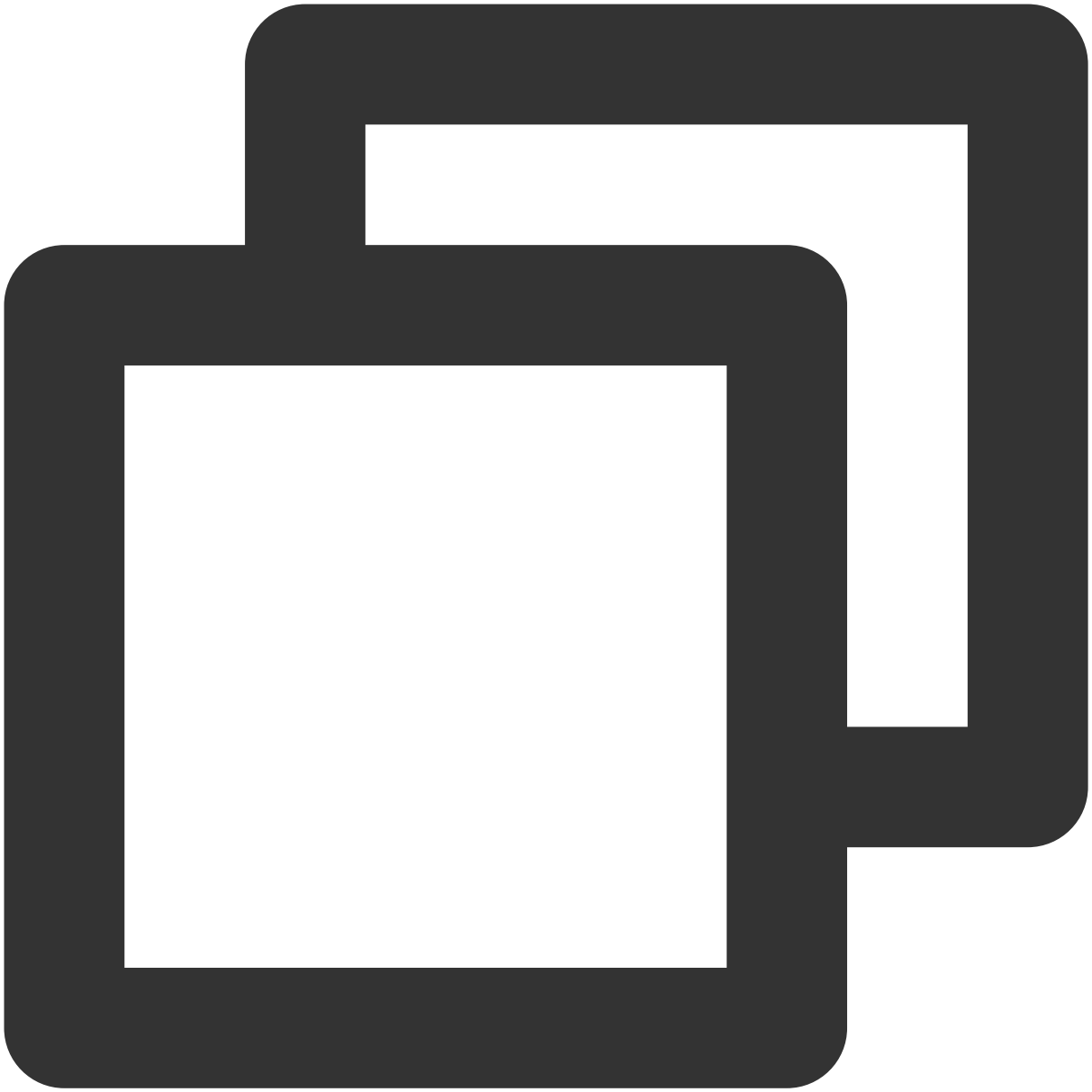


```
CALL `DataLakeCatalog`.`system`.snapshot('validation.table_01', 'validation.snap');
CALL `DataLakeCatalog`.`system`.snapshot('validation.table_01', 'validation.snap2',
```

call

Update and replace table attributes.

Statement



```
CALL `Catalog`.`system`.migrate(table, [properties]);
```

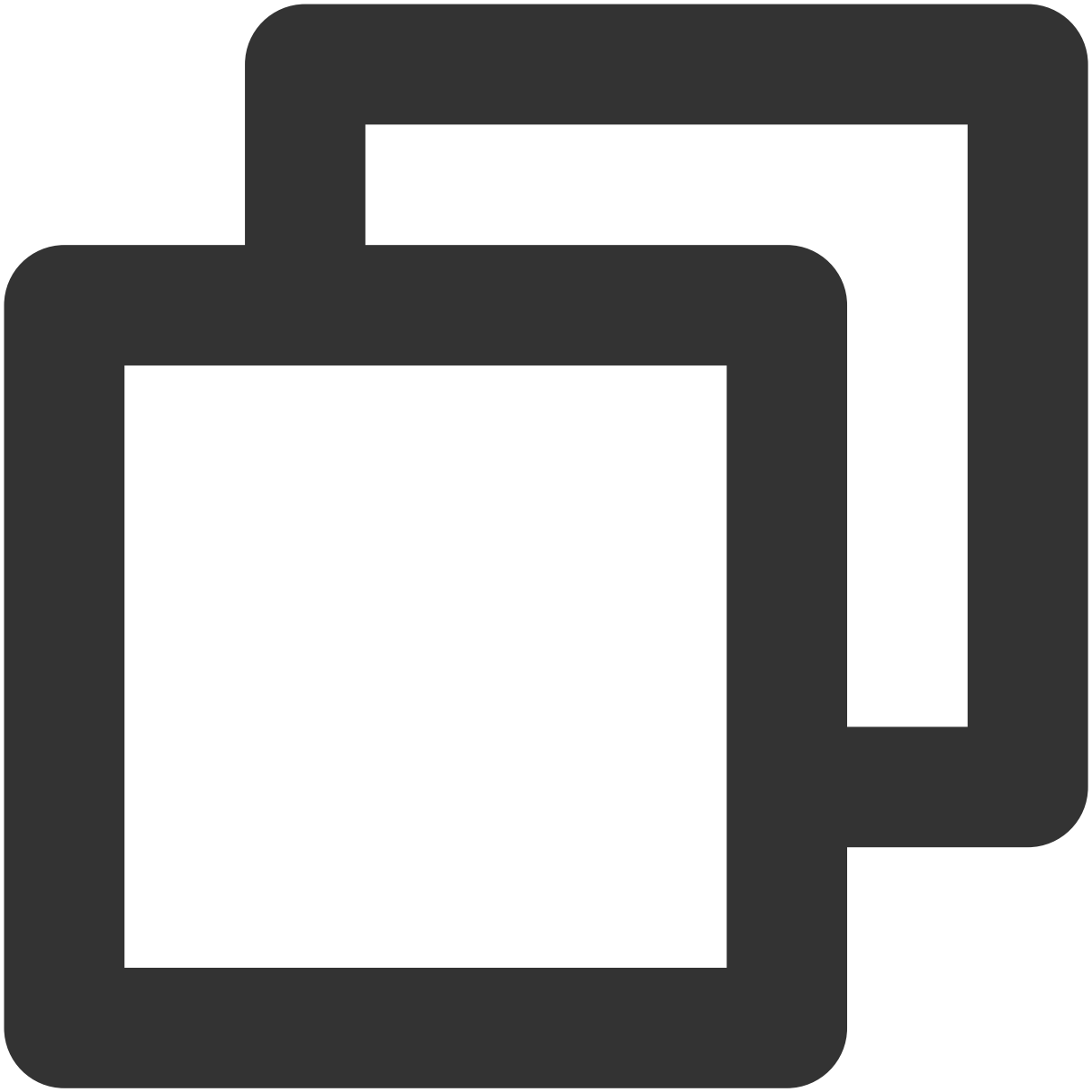
Example

```
CALL `DataLakeCatalog`.`system`.migrate('validation.table_01');
CALL `DataLakeCatalog`.`system`.migrate('validation.table_01', map('data', 'name'))
```

add_files

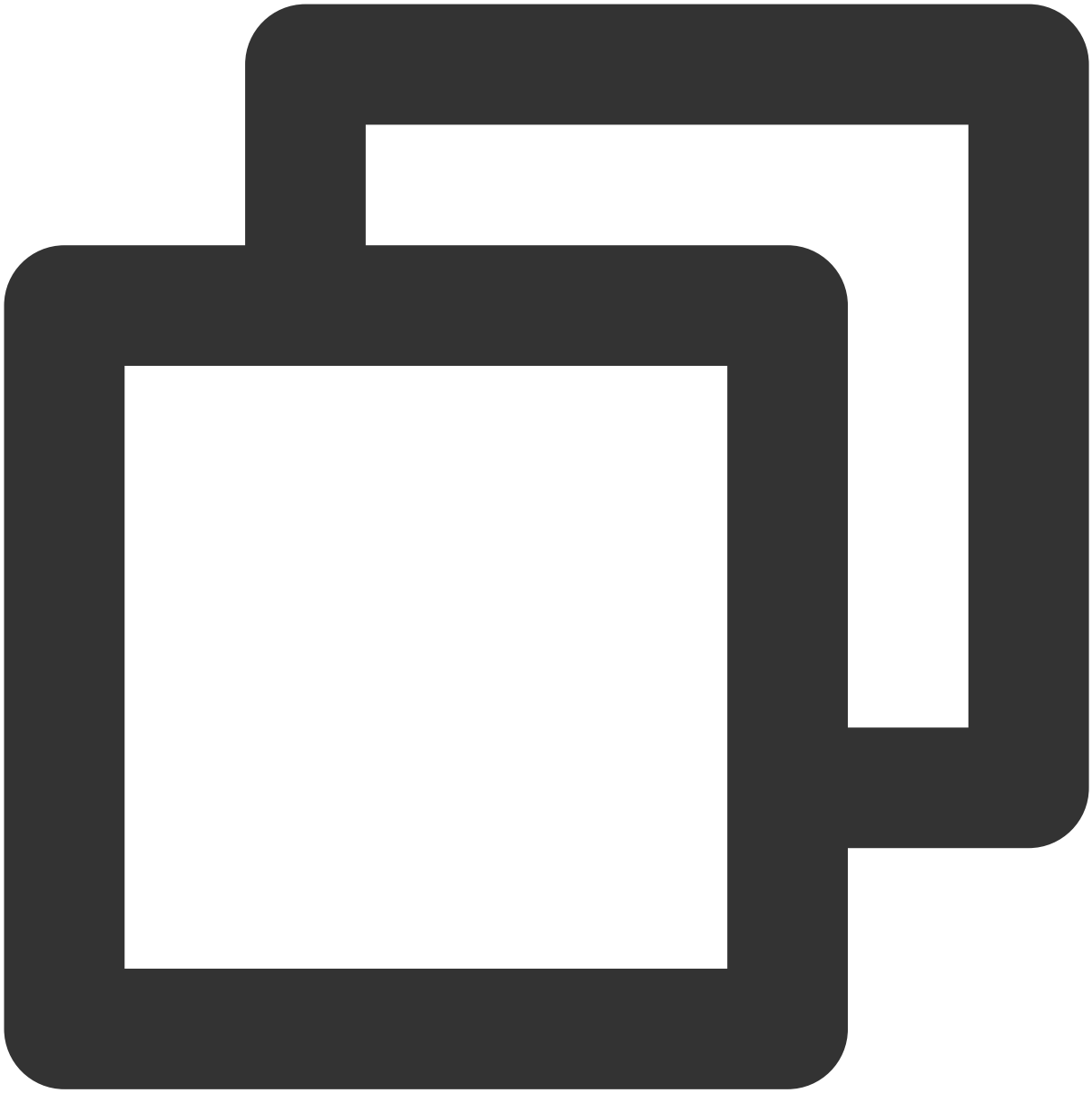
Load data files directly from Hive, and you can specify data files to a specific partition.

Statement



```
CALL `Catalog`.`system`.add_files(table, source_table, [partition_filter]);
```

Example



```
CALL `DataLakeCatalog`.`system`.add_files(`table`=>'validation.table_02', `source_t
CALL `DataLakeCatalog`.`system`.add_files(`table`=>'validation.table_02', `source_t
```

Materialized View Statement

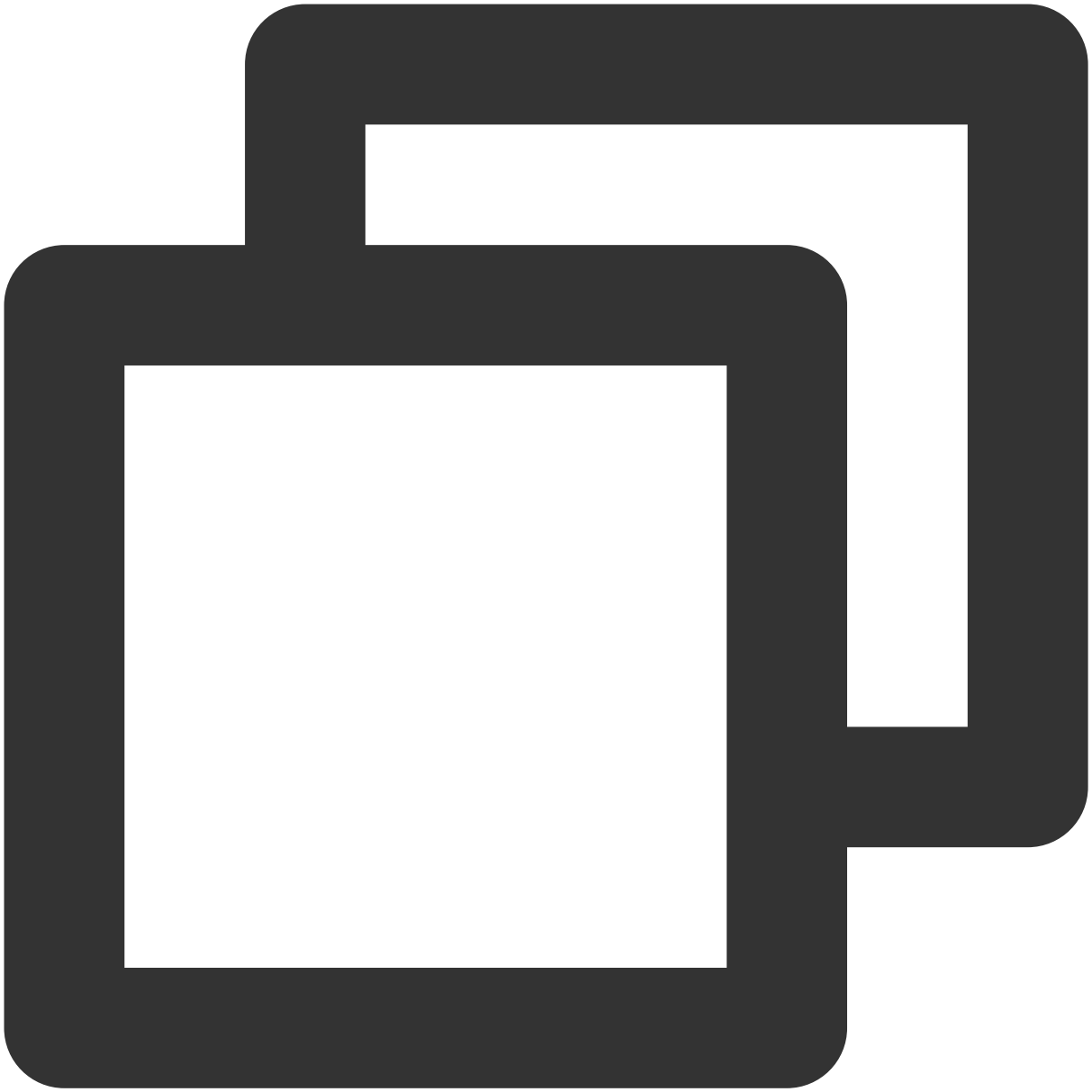
Last updated：2024-08-07 17:29:42

Description:

The following statement for materialized views is supported only by Presto and SparkSQL engines. For detailed usage instructions, see the Materialized View documentation.

CREATE MATERIALIZED

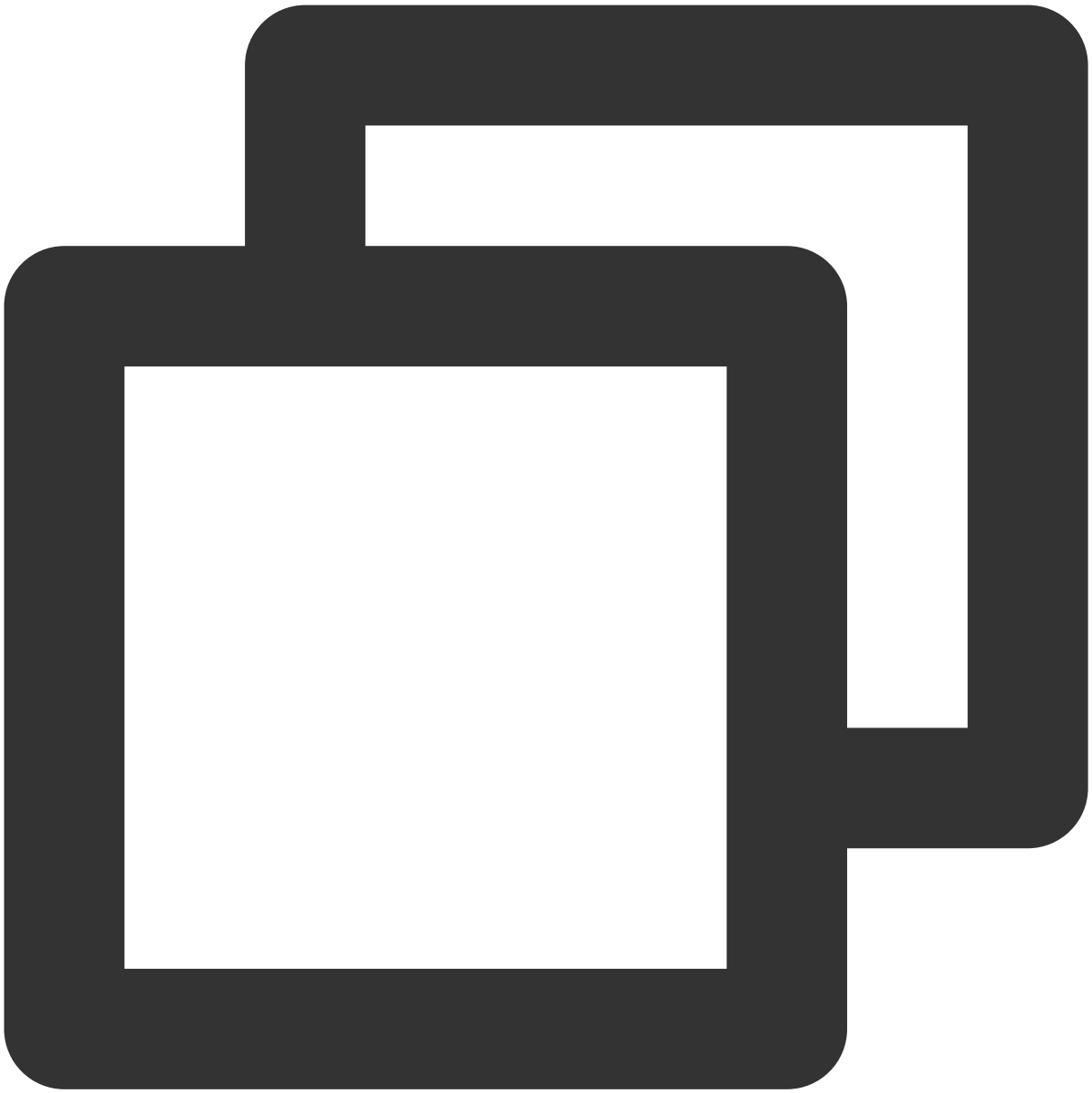
Create a materialized view:



```
CREATE MATERIALIZED VIEW [IF NOT EXISTS] mv_identifier
    [ENABLE | DISABLE REWRITE]
    [COMMENT mv_comment]
    [WITH META LINK]
    [TBLPROPERTIES (key1=value1, key2=value2,...)]
    [AS] select_query
```

DROP MATERIALIZED

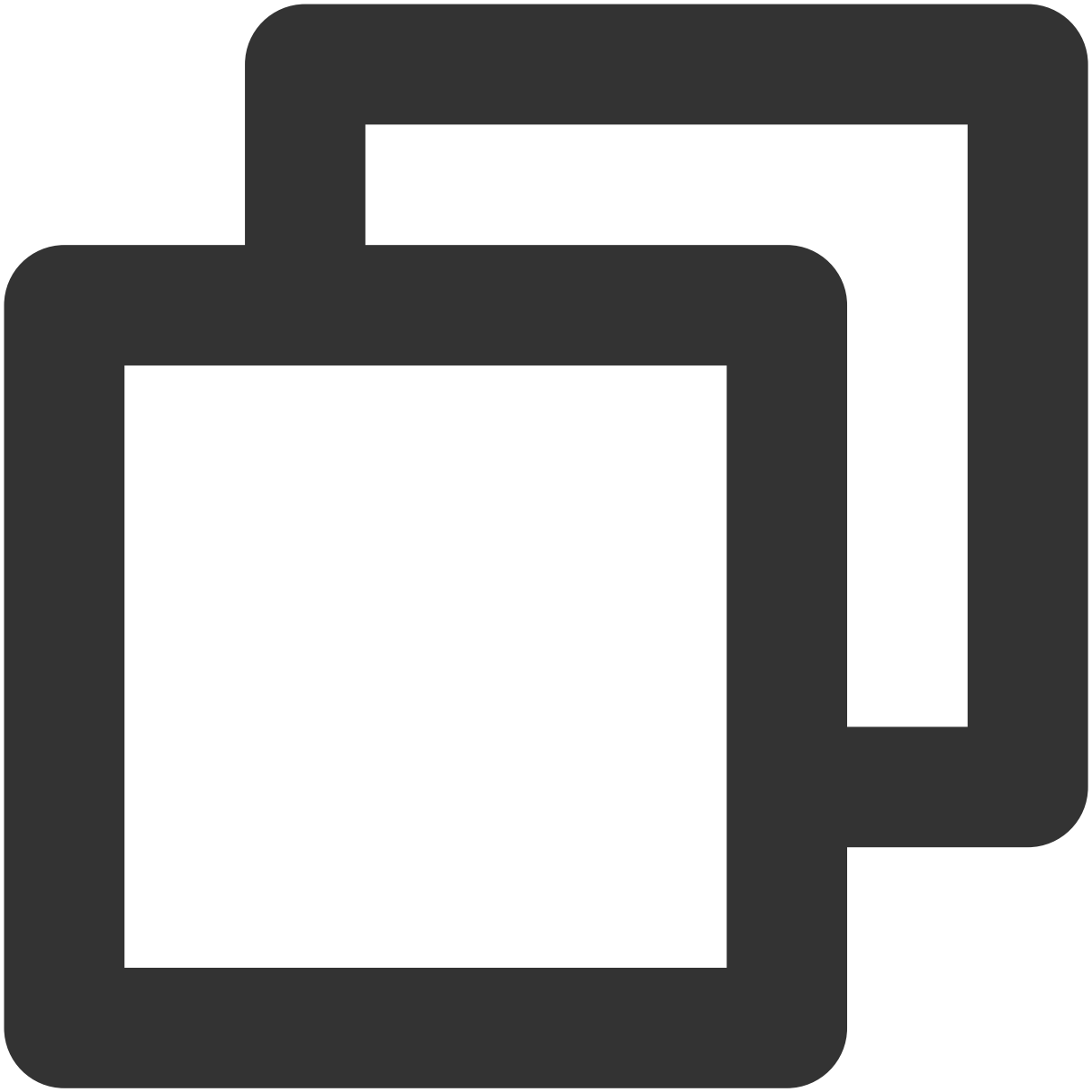
Delete a materialized view:



```
DROP MATERIALIZED VIEW [IF EXISTS] mv_identifier
```

DESCRIBE MATERIALIZED

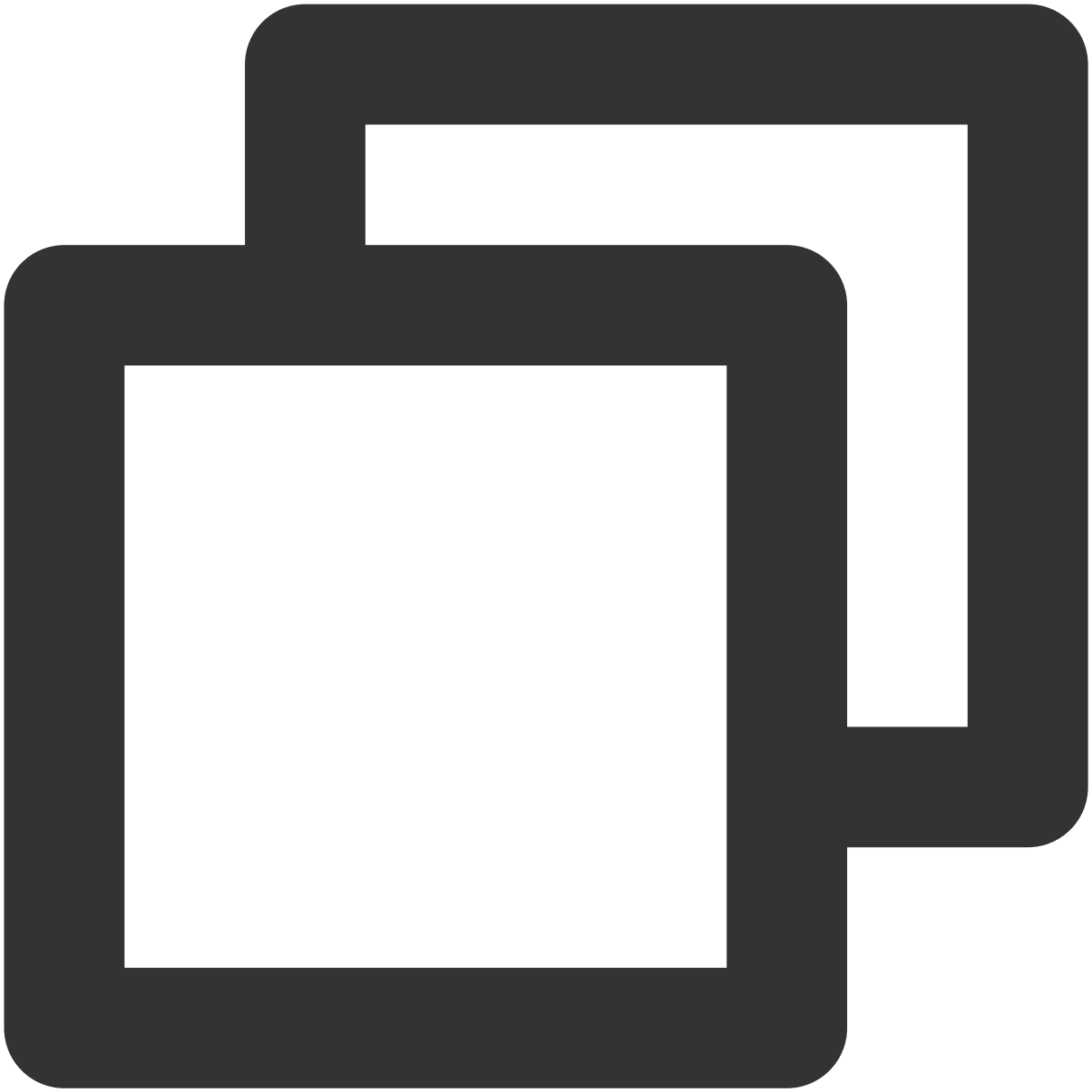
Describe a materialized view:



```
[DESCRIBE | DESC] MATERIALIZED VIEW mv_identifier
```

REFRESH MATERIALIZED

Refresh a materialized view:



```
REFRESH MATERIALIZED VIEW mv_identifier
```

SHOW MATERIALIZED VIEW JOBS

Display the execution task list of materialized views, providing insights into the refresh history and status of the materialized view.

```
SHOW MATERIALIZED VIEW JOBS IN mv_identifier
```

SQL Implicit Conversion

Last updated：2024-08-07 17:30:00

To facilitate SQL writing and quickly switch from Hive/Spark statement to the unified DLC SQL, we provide implicit SQL conversion capabilities. The conversion list is as follows:

I:implicit cast,E: explicit cast,--: not allowed

From-To	null	boolean	tinyint	smallint	int	bigint	decimal	float/real	double	inte
null	I	I	I	I	I	I	I	I	I	I
boolean	--	I	E	E	E	E	E	E	E	--
tinyint	--	E	I	I	I	I	I	I	I	E
smallint	--	E	I	I	I	I	I	I	I	E
int	--	E	I	I	I	I	I	I	I	E
bigint	--	E	I	I	I	I	I	I	I	E
decimal	--	E	I	I	I	I	I	I	I	E
float/real	--	E	I	I	I	I	I	I	I	--
double	--	E	I	I	I	I	I	I	I	--
interval	--	--	E	E	E	E	E	--	--	I
date	--	--	--	--	--	--	--	--	--	--
time	--	--	--	--	--	--	--	--	--	--
timestamp	--	--	E	E	E	E	E	E	E	--
[var]char	--	E	I	I	I	I	I	I	I	I
[var]binary	--	--	--	--	--	--	--	--	--	--

Functions
Unified Functions
Overview of Unified Functions

Last updated：2024-08-07 17:30:42

DLC supports the use of unified functions across different kernels, compatible with both Spark and Presto. Specific functions and supported kernels are listed in the table below.
DLC also supports Presto's built-in functions. For a list of supported functions and how to enable them, see Presto Built-in Functions.

Function	Spark	Presto
ABS	Input parameters: bigint|double|decimal	Input parameters: bigint|double|decimal
	Output parameter: same as input parameter	Output parameter: double
ACOS	Input parameters: bigint|double|decimal	Input parameters: bigint|double|decimal
	Output parameter: double	Output parameter: double
ACOSH	Input parameters: bigint|double|decimal	Input parameters: bigint|double|decimal
	Output parameter: double	Output parameter: double
ADD_MONTHS	Input parameters: date|timestamp|string, int	Input parameters: date|timestamp|string, int
	Output parameter: date	Output parameter: date
AES_DECRYPT	-	Input parameters: (binary, string|binary)
		Output parameter: binary
AES_ENCRYPT	-	Input parameters: (string|binary, string|binary)
		Output parameter: binary
AND	Input parameters: boolean AND boolean	boolean AND boolean

	Output parameter: boolean	Output parameter: boolean
ANY	Input parameter: boolean	Input parameter: boolean
	Output parameter: boolean	Output parameter: boolean
ANY_MATCH	Input parameters: (array<T>, lambda function)	-
	Output parameter: boolean	
APPROX_COUNT_DISTINCT	Input parameters: (bigint|double|decimal|date|timestamp,	-
	double|array<double>[, bigint])	
	Output parameter: same as the second input parameter	
APPROX_PERCENTILE	Input parameters: (bigint|double|decimal|, array<double>|double[, int])	Input parameters: (bigint|double|decimal, double[, int])
	Output parameter: double|array	Output parameter: double|array
ARG_MAX	Input parameters: (col1, col2 | expr(col2))	-
	Output parameters: the type of col2 or expr(col2)	
ARG_MIN	Input parameters: (col1, col2 | expr(col2))	
	Output parameters: the type of col2 or expr(col2)	
ARRAY	Input parameters: (T, ...)	Input parameters: (T, ...)
	Output parameter: array<T>	Output parameter: array<T>
ARRAYS_OVERLAP	Input parameters: (array<T>, array<T>)	Input parameters: (array<T>, array<T>)
	Output parameter: boolean	Output parameter: boolean
ARRAYS_ZIP	Input parameters: (array<T>, array< U> , ...)	-

	Output parameters: array<struct<T, U, ...>>	
ARRAY_CONTAINS	Input parameters: (array<T>, T)	Input parameters: (array<T>, T)
	Output parameter: boolean	Output parameter: boolean
ARRAY_DISTINCT	Input parameter: array<T>	Input parameter: array<T>
	Output parameter: array<T>	Output parameter: array<T>
ARRAY_EXCEPT	Input parameters: (array<T>, array<T>)	Input parameters: (array<T>, array<T>)
	Output parameter: array<T>	Output parameter: array<T>
ARRAY_INTERSECT	Input parameters: (array<T>, array<T>)	Input parameters: (array<T>, array<T>)
	Output parameter: array<T>	Output parameter: array<T>
ARRAY_JOIN	Input parameters: (array<T>, string[, string])	Input parameters: (array<T>, string[, string])
	Output parameter: string	Output parameter: string
ARRAY_MAX	Input parameter: array<bigint|double|decimal>	Input parameter: array<bigint|double|decimal>
	Output parameter: bigint|double|decimal	Output parameter: bigint|double|decimal
ARRAY_MIN	Input parameter: array<bigint|double|decimal>	Input parameter: array<bigint|double|decimal>
	Output parameter: bigint|double|decimal	Output parameter: bigint|double|decimal
ARRAY_POSITION	Input parameters: (array<T>, bigint)	Input parameters: (array<T>, bigint)
	Output parameter: bigint	Output parameter: bigint
ARRAY_REMOVE	Input parameters: (array<T>, T)	Input parameters: (array<T>, T)
	Output parameter: array<T>	Output parameter: array<T>
ARRAY_REPEAT	Input parameters: (array<T>, bigint)	Input parameters: (array<T>, bigint)

	Output parameter: array<T>	Output parameter: array<T>
ARRAY_SORT	Input parameters: (array<T>, function(T, T)->integer)	Input parameter: (array<T>)
	Output parameter: array<T>	Output parameter: array<T>
ARRAY_UNION	Input parameters: (array<T>, array<T>)	Input parameters: (array<T>, array<T>)
	Output parameter: array<T>	Output parameter: array<T>
ASCII	Input parameter: string	Input parameter: string
	Output parameter: int	Output parameter: int
ASIN	Input parameters: bigint|double|decimal	Input parameters: bigint|double|decimal
	Output parameter: double	Output parameter: double
ASINH	Input parameters: bigint|double|decimal	Input parameters: bigint|double|decimal
	Output parameter: double	Output parameter: double
ASSERT_TRUE	Input parameter: boolean	Input parameter: boolean
	Output parameter: null|throws an exception	Output parameter: null|throws an exception
ATAN	Input parameters: bigint|double|decimal	Input parameters: bigint|double|decimal
	Output parameter: double	Output parameter: double
ATAN2	Input parameters: (bigint|double|decimal, bigint|double|decimal)	Input parameters: (bigint|double|decimal, bigint|double|decimal)
	Output parameter: double	Output parameter: double
ATANH	Input parameters: bigint|double|decimal	Input parameters: bigint|double|decimal
	Output parameter: double	Output parameter: double
AVG	Input parameters: bigint|double|decimal	Input parameters:

		bigint|double|decimal
	Output parameter: double	Output parameter: double
BASE64	Input parameter: string|binary	Input parameter: binary
	Output parameter: string	Output parameter: string
BIGINT	Force the type to be converted to bigint.	Force the type to be converted to bigint.
BIN	Input parameter: bigint	Input parameter: bigint
	Output parameter: string	Output parameter: string
BINARY	Force the type to be converted to binary.	Force the type to be converted to binary.
BIT_AND	Input parameter: int	Input parameter: int
	Output parameter: int	Output parameter: int
BIT_COUNT	Input parameter: int|boolean	Input parameter: int|boolean
	Output parameter: int	Output parameter: int
BIT_GET	Input parameters: (int, int)	Input parameters: (int, int)
	Output parameter: int	Output parameter: int
BIT_LENGTH	Input parameter: string	Input parameter: string
	Output parameter: int	Output parameter: int
BIT_OR	Input parameter: int	Input parameter: int
	Output parameter: int	Output parameter: int
BIT_XOR	Input parameter: int	Input parameter: int
	Output parameter: int	Output parameter: int
BOOLEAN	Force the type to be converted to boolean.	Force the type to be converted to boolean.
BOOL_AND	Input parameter: boolean	Input parameter: boolean
	Output parameter: boolean	Output parameter: boolean
BOOL_OR	Input parameter: boolean	Input parameter: boolean

	Output parameter: boolean	Output parameter: boolean
BROUND	Input parameters: (bigint|double|decimal, int)	Input parameters: (bigint|double|decimal, int)
	Output parameter: int	Output parameter: int
BTRIM	Input parameters: (string[, string])	Input parameters: (string[, string])
	Output parameter: string	Output parameter: string
CARDINALITY	Input parameter: (array|map)	Input parameter: (array|map)
	Output parameter: int	Output parameter: int
CAST	Input parameter: (<expr> AS T)	Input parameter: (<expr> AS T)
	Output parameter: T	Output parameter: T
CBRT	Input parameters: bigint|double|decimal	Input parameters: bigint|double|decimal
	Output parameter: double	Output parameter: double
CEIL	Input parameters: bigint|double|decimal	Input parameters: bigint|double|decimal
	Output parameter: bigint	Output parameter: bigint
CEILING	Input parameters: bigint|double|decimal	Input parameters: bigint|double|decimal
	Output parameter: bigint	Output parameter: bigint
CHAR	Input parameter: int	Input parameter: int
	Output parameter: string	Output parameter: string
CHARACTER_LENGTH	Input parameter: string|binary	Input parameter: string
	Output parameter: int	Output parameter: int
CHAR_LENGTH	Input parameter: string|binary	Input parameter: string
	Output parameter: int	Output parameter: int
CHR	Input parameter: int|double	Input parameter: int|double

	Output parameter: string	Output parameter: string
CLUSTER_SAMPLE	Input parameters: (bigint [, bigint])	-
	Output parameter: boolean	
COALESCE	Input parameters: (T, T, ...)	Input parameters: (T, T, ...)
	Output parameter: T	Output parameter: T
COLLECT_LIST	Input parameter: T	Input parameter: T
	Output parameter: array<T>	Output parameter: array<T>
COLLECT_SET	Input parameter: T	Input parameter: T
	Output parameter: array<T>	Output parameter: array<T>
CONCAT	Input parameter: string|array	Input parameters: (string|binary, string|binary, ...)
	Output parameters: same as input parameters	Output parameter: string
CONCAT_WS	Input parameters: (string, [string | array<string>]+)	Input parameters: (string, [string | array<string>]+)
	Output parameter: string	Output parameter: string
CONTEXT_NGRAMS	-	Input parameters: (array<array<string>>, array<string>, int, int)
		Output parameter: array<struct<string, double>>
CONV	Input parameters: (bigint|double|decimal|string, int, int)	Input parameters: (bigint|double|decimal|string, int, int)
	Output parameter: string	Output parameter: string
CORR	Input parameter: bigint|double|decimal,	Input parameters: bigint|double|decimal
	Output parameter: double	Output parameter: double
	Output parameter: double	

COS	Input parameters: bigint|double|decimal	Input parameters: bigint|double|decimal
	Output parameter: double	Output parameter: double
COSH	Input parameters: bigint|double|decimal	Input parameters: bigint|double|decimal
	Output parameter: double	Output parameter: double
COT	Input parameters: bigint|double|decimal	Input parameters: bigint|double|decimal
	Output parameter: double	Output parameter: double
COUNT	Input parameter: T	Input parameter: T
	Output parameter: bigint	Output parameter: bigint
COUNT_IF	Input parameter: boolean	-
	Output parameter: int	

COUNT_MIN_SKETCH	Input parameters: (int|binary|string, double, double, int)	-
	Output parameter: binary	
COVAR_POP	Input parameters: bigint|double|decimal	Input parameter: bigint|double|decimal
	Output parameter: double	Output parameter: double
COVAR_SAMP	Input parameters: bigint|double|decimal	Input parameters: bigint|double|decimal
	Output parameter: double	Output parameter: double
CRC32	Input parameter: string|binary	Input parameter: string|binary
	Output parameter: bigint	Output parameter: bigint
CUME_DIST	Input parameter: None	Input parameter: None
	Output parameter: double	Output parameter: double

CURRENT_CATALOG	Input parameter: None	-
	Output parameter: string	
CURRENT_DATABASE	Input parameter: None	-
	Output parameter: string	
CURRENT_DATE	Input parameter: None	Input parameter: None
	Output parameter: date	Output parameter: date
CURRENT_TIMESTAMP	Input parameter: None	Input parameter: None
	Output parameter: timestamp	Output parameter: timestamp
CURRENT_TIMEZONE	Input parameter: None	Input parameter: None
	Output parameter: string	Output parameter: string
CURRENT_USER	Input parameter: None	Input parameter: None
	Output parameter: string	Output parameter: string
DATE	Input parameter: string	Input parameter: string
	Output parameter: date	Output parameter: date
DATEDIFF	Input parameters: (string|timestamp|date, string|timestamp|date)	Input parameters: (string|timestamp|date, string|timestamp|date)
	Output parameter: int	Output parameter: int
DATE_ADD	Input parameters: (string|date|timestamp, int)	Input parameters: (string|date|timestamp, int)
	Output parameter: date	Output parameter: date
DATEADD	Input parameters: (date/timestamp, int, string)	-
	Output parameter: timestamp	
DATE_FORMAT	Input parameters:	Input parameters: (string|date|timestamp,

	(string|date|timestamp, string)	string)
	Output parameter: string	Output parameter: string
DATE_FROM_UNIX_DATE	Input parameter: int	Input parameter: int
	Output parameter: date	Output parameter: date
DATE_PART	Input parameters: (string, string|date|timestamp)	Input parameters: (string, string|date|timestamp)
	Output parameter: bigint|double|decimal	Output parameter: bigint|double|decimal
DATE_SUB	Input parameter: (string|date|timestamp)	Input parameter: (string|date|timestamp)
	Output parameter: date	Output parameter: date
DATE_TRUNC	Input parameters: (string, string)	Input parameters: (string, string)
	Output parameter: timestamp	Output parameter: timestamp
DAY	Input parameter: string|timestamp|date	Input parameter: string|timestamp|date
	Output parameter: int	Output parameter: int
DAYOFMONTH	Input parameter: string|timestamp|date	Input parameter: string|timestamp|date
	Output parameter: int	Output parameter: int
DAYOFWEEK	Input parameter: string|timestamp|date	string|timestamp|date
	Output parameter: int	Output parameter: int
DAYOFYEAR	Input parameter: string|timestamp|date	string|timestamp|date
	Output parameter: int	Output parameter: int
DECIMAL	cast as decimal	cast as decimal
DECODE	Input parameters: (binary,	Input parameters: (binary, string)

	string) | (int, int, string[, int, string]...[, default])	
	Output parameter:	Output parameter: string
DEGREES	Input parameters: bigint|double|decimal	Input parameters: bigint|double|decimal
	Output parameter: double	Output parameter: double
DENSE_RANK	Input parameter: None	Input parameter: None
	Output parameter: int	Output parameter: int
DIV	Input parameters: bigint|double|decimal	-
	Output parameter: int	
DOUBLE	cast as double	cast as double
E	Input parameter: None	Input parameter: None
	Output parameter: double	Output parameter: double
ELEMENT_AT	Input parameters: (array|map, int)	Input parameters: (array, int)
	Output parameter: same as array element|map value	Output parameter: same as array element
ELT	Input parameters: (int, T, U,...)	Input parameters: (int, string, string, ...)
	Output parameter: Depends on the input.	Output parameter: string
ENCODE	Input parameters: (string|binary, string)	Input parameters: (string, string)
	Output parameter: binary	Output parameter: binary
EVERY	Input parameter: T	Input parameter: T
	Output parameter: boolean	Output parameter: boolean
EXP	Input parameters:	Input parameters: bigint|double|decimal

	bigint|double|decimal	
	Output parameter: double	Output parameter: double
EXPLODE	Input parameter: array<T>	-
	Output parameter: T	
EXPLODE_OUTER	Input parameter: array<T>	-
	Output parameter: T	
EXPM1	Input parameters: bigint|double|decimal	-
	Output parameter: double	
FACTORIAL	Input parameter: int	Input parameter: int
	Output parameter: bigint	Output parameter: bigint
FIELD	-	Input parameters: (T, T, T, ...)
		Output parameter: int
FIND_IN_SET	Input parameters: (string, string)	Input parameters: (string, string)
	Output parameter: int	Output parameter: int
FIRST_VALUE	Input parameters: (T[, boolean])	-
	Output parameter: T	
FLATTEN	Input parameter: array<array<T>>	Input parameter: array<array<T>>
	Output parameter: array<T>	Output parameter: array<T>
FLOAT	cast as float	cast as float
FLOOR	Input parameters: bigint|double|decimal	Input parameters: bigint|double|decimal
	Output parameter: bigint	Output parameter: bigint

FORMAT_NUMBER	Input parameters: (bigint|double|decimal, int|string)	Input parameters: (bigint|double|decimal, int)
	Output parameter: string	Output parameter: string
FORMAT_STRING	Input parameters: (string, T, ...)	Input parameters: (string, T, ...)
	Output parameter: string	Output parameter: string
FROM_CSV	Input parameters: (string, string[, map])	-
	Output parameter: struct	
FROM_JSON	Input parameters: (string, string[, map])	-
	Output parameter: struct	
FROM_UNIXTIME	Input parameters: (bigint[, string])	Input parameters: (bigint[, string])
	Output parameter: string	Output parameter: string
FROM_UTC_TIMESTAMP	Input parameters: (string, string)	Input parameters: (bigint|double|decimal|timestamp|date|string, string)
	Output parameter: timestamp	Output parameter: timestamp
GETBIT	Input parameters: (int, int)	Input parameters: (int, int)
	Output parameter: int	Output parameter: int
GET_IDCARD_AGE	Input parameter: string	-
	Output parameter: int	
GET_IDCARD_BIRTHDAY	Input parameter: string	-
	Output parameter: date	
GET_IDCARD_SEX	Input parameter: string	-
	Output parameter: string	

GREATEST	Input parameters: (bigint|double|decimal, bigint|double|decimal, ...)	Input parameters: (bigint|double|decimal, bigint|double|decimal, ...)
	Output parameter: same as input	Output parameter: same as input
GROUPING	Input parameter: T	T
	Output parameter: int	Output parameter: int
GROUPING_ID	Input parameters: [T[, T...]]	-
	Output parameter: int	
HASH	Input parameters: (T, T,...)	Input parameters: (T, T,...)
	Output parameter: int	Output parameter: int
HEX	Input parameter: int|string|binary	Input parameter: int|string|binary
	Output parameter: string	Output parameter: string
HISTOGRAM_NUMERIC	-	Input parameters: bigint|double|decimal
		Output parameter: array<struct<'x', 'y'>>
HOUR	Input parameter: string|timestamp	Input parameter: string|timestamp
	Output parameter: int	Output parameter: int
HYPOT	Input parameters: bigint|double|decimal	Input parameters: bigint|double|decimal
	Output parameter: double	Output parameter: double
IF	Input parameters: (boolean, T, T)	Input parameters: (boolean, T, T)
	Output parameter: T	Output parameter: T
IFNULL	Input parameters: (T, U)	-
	Output parameter: T|U	
IN	Input parameters: (T, T,...)	Input parameters: (T, T,...)

	Output parameter: boolean	Output parameter: boolean
INITCAP	Input parameter: string	Input parameter: string
	Output parameter: string	Output parameter: string
INLINE	Input parameter: array	-
	Output parameter: table	
INLINE_OUTER	Input parameter: array	-
	Output parameter: table	
INPUT_FILE_BLOCK_LENGTH	Input parameter: None	-
	Output parameter: int	
INPUT_FILE_BLOCK_START	Input parameter: None	-
	Output parameter: int	
INPUT_FILE_NAME	Input parameter: None	-
	Output parameter: string	
INSTR	Input parameters: (string, string)	Input parameters: (string, string)
	Output parameter: int	Output parameter: int
INT	Force the type to be converted to int.	Force the type to be converted to int.
ISNAN	Input parameter: T	Input parameter: T
	Output parameter: boolean	Output parameter: boolean
ISNOTNULL	Input parameter: T	Input parameter: T
	Output parameter: boolean	Output parameter: boolean
ISNULL	Input parameter: T	Input parameter: T
	Output parameter: boolean	Output parameter: boolean

JSON_ARRAY_LENGTH	Input parameter: string	Input parameter: string
	Output parameter: int	Output parameter: int
JSON_OBJECT_KEYS	Input parameter: string	Input parameter: string
	Output parameter: array	Output parameter: array
JSON_TUPLE	Input parameters: (string, ..., string)	-
	Output parameter: string...string	
KEYVALUE	Input parameters: (string [, string, string], string)	
	Output parameter: string	
KURTOSIS	Input parameters: bigint|double|decimal	-
	Output parameter: double	
LAG	Input parameters: (T[, int[, string]])	Input parameters: (T[, int[, string]])
	Output parameter: T	Output parameter: T
LAST_DAY	Input parameters: string|date|timestamp	Input parameter: string|date|timestamp
	Output parameter: string	Output parameter: string
LAST_VALUE	Input parameters: (T[, boolean])	-
	Output parameter: T	
LCASE	Input parameter: string	Input parameter: string
	Output parameter: string	Output parameter: string
LEAD	Input parameters: (T[, int[, string]])	Input parameters: (T[, int[, string]])
	Output parameter: T	Output parameter: T
LEAST	Input parameters:	(bigint|double|decimal,

	(bigint|double|decimal, bigint|double|decimal, ...)	bigint|double|decimal, ...)
	Output parameter: same as input	Output parameter: same as input
LEFT	Input parameters: (string, int)	Input parameters: (string, int)
	Output parameter: string	Output parameter: string
LENGTH	Input parameter: string	Input parameter: string
	Output parameter: int	Output parameter: int
LEVENSHTEIN	Input parameters: (string, string)	Input parameters: (string, string)
	Output parameter: int	Output parameter: int
LIKE	Input parameters: (string, string) | string LIKE string ESCAPE string	Input parameters: (string, string)
	Output parameter: boolean	Output parameter: boolean
LN	Input parameters: bigint|double|decimal	Input parameters: bigint|double|decimal
	Output parameter: double	Output parameter: double
LOCATE	Input parameters: (string, string[, int])	Input parameters: (string, string[, int])
	Output parameter: int	Output parameter: int
LOG	Input parameters: (bigint|double|decimal, bigint|double|decimal)	Input parameters: (bigint|double|decimal, bigint|double|decimal)
	Output parameter: double	Output parameter: double
LOG10	Input parameters: bigint|double|decimal	Input parameters: bigint|double|decimal
	Output parameter: double	Output parameter: double
LOG1P	Input parameters:	Input parameters: bigint|double|decimal

	bigint|double|decimal	
	Output parameter: double	Output parameter: double
LOG2	Input parameters: bigint|double|decimal	Input parameters: bigint|double|decimal
	Output parameter: double	Output parameter: double
LOWER	Input parameter: string	Input parameter: string
	Output parameter: string	Output parameter: string
LPAD	Input parameters: (string, int[, string])	Input parameters: (string, int, string)
	Output parameter: string	Output parameter: string
LTRIM	Input parameter: string	Input parameter: string
	Output parameter: string	Output parameter: string
MAKE_DATE	Input parameters: (int, int, int)	Input parameters: (int, int, int)
	Output parameter: date	Output parameter: date
MAKE_TIMESTAMP	Input parameters: (int, int, int, int, int, int[, string])	Input parameters: (int, int, int, int, int, int[, string])
	Output parameter: timestamp	Output parameter: timestamp

MAP	Input parameters: (K1, V1, K2, V2,...)	Input parameters: (array<K>
	Output parameter: map	Output parameter: map
MAP_CONCAT	Input parameters: (map, ...)	Input parameters: (map, ...)
	Output parameter: map	Output parameter: map
MAP_ENTRIES	Input parameter: map	Input parameter: map
	Output parameter: array	Output parameter: array
MAP_FROM_ARRAYS	Input parameters: (array, array)	Input parameters: (array, ar
	Output parameter: map	Output parameter: map

MAP_FROM_ENTRIES	Input parameter: array	Input parameter: array
	Output parameter: map	Output parameter: map
MAP_KEYS	Input parameter: map	Input parameter: map
	Output parameter: array	Output parameter: array
MAP_UINON_SUM	Input parameter: map	-
	Output parameter: map	
MAP_VALUES	Input parameter: map	Input parameter: map
	Output parameter: array	Output parameter: array
MAX	Input parameters: bigint|double|decimal	Input parameters: bigint|dou
	Output parameter: same as input parameter	Output parameter: double
MAX_BY	Input parameters: T, bigint|double|decimal	Input parameters: T, bigint|c
	Output parameter: T	Output parameter: T
MAX_PT	Input parameter: const string	-
	Output parameter: string	
MD5	Input parameter: string|binary	Input parameter: string|bina
	Output parameter: string	Output parameter: string
MEAN	Input parameters: bigint|double|decimal	Input parameters: bigint|dou
	Output parameter: double	Output parameter: double
MIN	Input parameters: bigint|double|decimal	Input parameters: bigint|dou
	Output parameter: same as input parameter	Output parameter: double
MINUTE	Input parameters: (stirng|timestamp)	Input parameters: (stirng|tim
	Output parameter: int	Output parameter: int

MIN_BY	Input parameters: (T, bigint|double|decimal)	Input parameters: (T, bigint|
	Output parameter: T	Output parameter: T
MOD	Input parameters: (bigint|double|decimal, bigint|double|decimal) | bigint|double|decimal MOD bigint|double|decimal	Input parameters: (bigint|do bigint|double|decimal)
	Output parameters: int|double	Output parameters: int|doub
MONOTONICALLY_INCREASING_ID	Input parameter: None	-
	Output parameter: bigint	
MONTH	Input parameters: date|timestamp|string	Input parameters: date|time
	Output parameter: int	Output parameter: int
MONTHS_BETWEEN	Input parameters: (date|timestamp|string, date|timestamp|string[, boolean])	Input parameters: (date|time date|timestamp|string)
	Output parameter: double	Output parameter: double
NAMED_STRUCT	Input parameters: (K1, V1, K2, V2, ...)	-
	Output parameter: struct	
NANVL	Input parameters: (bigint|double|decimal, bigint|double|decimal)	Input parameters: (bigint|do bigint|double|decimal)
	Output parameter: double	Output parameter: double
NEGATIVE	Input parameters: bigint|double|decimal	Input parameters: bigint|dou
	Output parameter: bigint|double|decimal	Output parameter: int | doub
NEXT_DAY	Input parameters: (string|date|timestamp, string)	Input parameters: (string|da string)
	Output parameter: date	Output parameter: string

NGRAMS	-	Input parameters: (array <ar int, int, int)
		Output parameter: array<str double>>
NOW	Input parameter: None	Input parameter: None
	Output parameter: timestamp	Output parameter: timestam
NTH_VALUE	Input parameters: (T[, int])	Input parameters: (T[, int])
	Output parameter: T	Output parameter: T
NTILE	Input parameter: int	Input parameter: int
	Output parameter: int	Output parameter: int
NULLIF	Input parameters: (T, T)	Input parameters: (T, T)
	Output parameter: T	Output parameter: T
NVL	Input parameters: (T, T)	Input parameters: (T, T)
	Output parameter: T	Output parameter: T
NVL2	Input parameters: (T, T, T)	Input parameters: (T, T, T)
	Output parameter: T	Output parameter: T
OCTET_LENGTH	Input parameter: string	Input parameter: string
	Output parameter: int	Output parameter: int
OR	Input parameter: expr1 OR expr2	Input parameter: expr1 OR e
	Output parameter: boolean	Output parameter: boolean
OVERLAY	Input parameters: (string PLACING string FROM int[FOR int])	-
	Output parameter:	
PARSE_URL	Input parameters: (string, string[, string])	Input parameters: (string, st
	Output parameter: string	Output parameter: string
PERCENTILE	Input parameters: (bigint|double|decimal,	Input parameters: (bigint|do bigint|double|decimal|array,

	bigint|double|decimal|array, int)	
	Output parameters: bigint|double|decimal|array	Output parameters: double|a
PERCENTILE_APPROX	Input parameters: (bigint|double|decimal, bigint|double|decimal | array[, int])	Input parameters: (bigint|do bigint|double|decimal[, int])
	Output parameter: double|array	Output parameter: double|ar
PERCENT_RANK	Input parameter: None	Input parameter: None
	Output parameter: double	Output parameter: double
PI	Input parameter: None	Input parameter: None
	Output parameter: double	Output parameter: double
PMOD	Input parameters: (bigint|double|decimal, bigint|double|decimal)	Input parameters: (bigint|do bigint|double|decimal)
	Output parameter: same as input parameter	Output parameter: same as
POSITION	Input parameters: (string, string[, int]) | (string IN string)	Input parameters: (string, st (string IN string)
	Output parameter: int	Output parameter: int
POSITIVE	Input parameters: bigint|double|decimal	Input parameters: bigint|dou
	Output parameter: same as input parameter	Output parameter: same as
POSEXPLODE	Input parameter: array	-
	Output parameter: table	
POSEXPLODE_OUTER	Input parameter: array	-
	Output parameter: table	
POW	Input parameters: (bigint|double|decimal, bigint|double|decimal)	Input parameters: (bigint|do bigint|double|decimal)

	Output parameter: double	Output parameter: double
POWER	Input parameters: (bigint|double|decimal, bigint|double|decimal)	Input parameters: (bigint|do bigint|double|decimal)
	Output parameter: double	Output parameter: double
PRINTF	Input parameters: (string, T, ...)	Input parameters: (string, T,
	Output parameter: string	Output parameter: string
QUARTER	Input parameters: string|date|timestamp	Input parameters: string|dat
	Output parameter: int	Output parameter: int
RADIANS	Input parameters: bigint|double|decimal	Input parameters: bigint|dou
	Output parameter: double	Output parameter: double
RAISE_ERROR	Input parameter: string	Input parameter: string
	Output parameter: throws an exception	Output parameter: throws a
RAND	Input parameter: [int]	Input parameter: [int]
	Output parameter: double	Output parameter: double
RANDN	Input parameter: [int]	Input parameter: [long]
	Output parameter: double	Output parameter: double
RANDOM	Input parameter: [int]	Input parameter: [int]
	Output parameter: double	Output parameter: double
RANK	Input parameter: None	Input parameter: None
	Output parameter: int	Output parameter: int
REGEXP	Input parameters: (string, string)	Input parameters: (string, st
	Output parameter: boolean	Output parameter: boolean
REGEXP_COUNT	Input parameters: (string, string)	-
	Output parameter: int	

REGEXP_EXTRACT	Input parameters: (string, string[, int])	Input parameters: (string, st
	Output parameter: int	Output parameter: int
REGEXP_EXTRACT_ALL	Input parameters: (string, string[, int])	Input parameters: (string, st
	Output parameter: array	Output parameter: array
REGEXP_LIKE	Input parameters: (string, string)	Input parameters: (string, st
	Output parameter: boolean	Output parameter: boolean
REGEXP_REPLACE	Input parameters: (string, string, string[, int])	Input parameters: (string, st
	Output parameter: string	Output parameter: string
REGEXP_SUBSTR	Input parameters: (string, string)	-
	Output parameter: string	
REPEAT	Input parameters: (string, int)	Input parameters: (string, int
	Output parameter: string	Output parameter: string
REPLACE	Input parameters: (string, string[, string])	Input parameters: (string, st
	Output parameter: string	Output parameter: string
REVERSE	Input parameters: string | array	Input parameter: string
	Output parameter: string	Output parameter: string
RIGHT	Input parameters: (string, int)	Input parameters: (string, int
	Output parameter: string	Output parameter: string
RINT	Input parameters: bigint|double|decimal	Input parameters: bigint|dou
	Output parameter: double	Output parameter: double
RLIKE	Input parameters: (string, string)	Input parameters: (string, st
	Output parameter: boolean	Output parameter: boolean
ROUND	Input parameters: (double, int)	Input parameters: (double, i
	Output parameter: double	Output parameter: double

ROW_NUMBER	Input parameter: None	Input parameter: None
	Output parameter: int	Output parameter: int
RPAD	Input parameters: (string, int[, string])	Input parameters: (string, int
	Output parameter: string	Output parameter: string
RTRIM	Input parameter: string	Input parameter: string
	Output parameter: string	Output parameter: string
SAMPLE	Input parameters: (long [, long [, cols...]])	-
	Output parameter: boolean	
SCHEMA_OF_CSV	Input parameters: (string[, map])	
	Output parameter: struct	
SCHEMA_OF_JSON	Input parameters: (string[, map])	-
	Output parameter:	
SECOND	Input parameters: string|timestamp	Input parameter: string|time
	Output parameter: int	Output parameter: int
SENTENCES	(string[, string, string])	Input parameters: (string[, s
	Output parameter: array<array<string>>	Output parameter: array<ar
SEQUENCE	Input parameters: (int|bigint|date|timestamp, int|bigint|date|timestamp, int)	Input parameters: (int|bigint| int|bigint|date|timestamp, int
	Output parameter: same as input parameter	Output parameter: same as
SHA	Input parameter: string|binary	Input parameter: string|bina
	Output parameter: string	Output parameter: string
SHA1	Input parameters: string|binary	Input parameter: string|bina
	Output parameter: string	Output parameter: string

SHA2	Input parameters: (string, int)	Input parameters: (string, int
	Output parameter: string	Output parameter: string
SHIFTLEFT	Input parameters: (int|bigint, int)	Input parameters: (int|biging
	Output parameter: int|bigint	Output parameter: int|bigint
SHIFTRIGHT	Input parameters: (int|bigint, int)	Input parameters: (int|bigint,
	Output parameter: int|bigint	Output parameter: int|bigint
SHIFTRIGHTUNSIGNED	Input parameters: (int|bigint, int)	Input parameters: (int|bigint,
	Output parameter: int|bigint	Output parameters: int|bigin
SHUFFLE	Input parameter: array	Input parameter: array
	Output parameter: array	Output parameter: array
SIGN	Input parameters: bigint|double|decimal	Input parameters: bigint|dou
	Output parameter: double	Output parameter: double
SIGNUM	Input parameters: bigint|double|decimal	Input parameters: bigint|dou
	Output parameter: double	Output parameter: double
SIN	Input parameters: bigint|double|decimal	Input parameters: bigint|dou
	Output parameter: double	Output parameter: double
SINH	Input parameters: bigint|double|decimal	Input parameters: bigint|dou
	Output parameter: double	Output parameter: double
SIZE	Input parameters: array | map	Input parameter: array | map
	Output parameter: int	Output parameter: int
SKEWNESS	Input parameters: bigint|double|decimal	-
	Output parameter: double	
SLICE	Input parameters: (array, int, int)	Input parameters: (array, int

	Output parameter: array	Output parameter: array
SMALLINT	cast as smallint	cast as smallint
SOME	Input parameter: boolean	Input parameter: boolean
	Output parameter: boolean	Output parameter: boolean
SORT_ARRAY	Input parameters: (array[, boolean])	Input parameter: array
	Output parameter: array	Output parameter: array
SOUNDEX	Input parameter: string	Input parameter: string
	Output parameter: string	Output parameter: string
SPACE	Input parameter: string	Input parameter: string
	Output parameter: string	Output parameter: string
SPARK_PARTITION_ID	Input parameter: None	-
	Output parameter: int	
SPLIT	Input parameters: (string, string, int)	Input parameters: (string, st
	Output parameter: string	Output parameter: string
SQRT	Input parameters: bigint|double|decimal	Input parameters: bigint|dou
	Output parameter: double	Output parameter: double
STACK	Input parameters: (T, U, ...)	-
	Output parameter: table	
STD	Input parameters: bigint|double|decimal	Input parameters: bigint|dou
	Output parameter: double	Output parameter: double
STDDEV	Input parameters: bigint|double|decimal	Input parameters: bigint|dou
	Output parameter: double	Output parameter: double
STDDEV_POP	Input parameters: bigint|double|decimal	Input parameters: bigint|dou

	Output parameter: double	Output parameter: double
STDDEV_SAMP	Input parameters: bigint|double|decimal	Input parameters: bigint|dou
	Output parameter: double	Output parameter: double
STRING	cast as string	cast as string
STRUCT	Input parameters: (T, U, ...)	Input parameters: (T, U, ...)
	Output parameter: struct	Output parameter: struct
STR_TO_MAP	Input parameters: (string[, string[, string]])	Input parameters: (string[, s
	Output parameter: map	Output parameter: map
SUBSTR	Input parameters: (string, int[, int]) | (string FROM int [FOR int])	Input parameters: (string|bir (string|binary FROM int [FO
	Output parameter: string	Output parameter: string
SUBSTRING	Input parameters: (string, int[, int]) | (string FROM int [FOR int])	Input parameters: (string|bir (string|binary FROM int [FO
	Output parameter: string	Output parameter: string
SUBSTRING_INDEX	Input parameters: (string, string, int)	Input parameters: (string, st
	Output parameter: string	Output parameter: string
SUM	Input parameters: bigint|double|decimal	Input parameters: bigint|dou
	Output parameter: same as input	Output parameter: double
TAN	Input parameters: bigint|double|decimal	Input parameters: bigint|dou
	Output parameter: double	Output parameter: double
TANH	Input parameters: bigint|double|decimal	Input parameters: bigint|dou
	Output parameter: double	Output parameter: double
TIMESTAMP	Input parameter: string	Input parameter: string

	Output parameter: timestamp	Output parameter: timestam
TIMESTAMP_ADD	Input parameters: (date/timestamp, int, string)	
	Output parameter: timestamp	
TIMESTAMP_MICROS	Input parameter: bigint	Input parameter: bigint
	Output parameter: timestamp	Output parameter: timestam
TIMESTAMP_MILLIS	Input parameter: bigint	Input parameter: bigint
	Output parameter: timestamp	Output parameter: timestam
TIMESTAMP_SECONDS	Input parameter: bigint	Input parameter: bigint
	Output parameter: timestamp	Output parameter: timestam
TIMEZONE_HOUR	Input parameter: string	-
	Output parameter: int	
TIMEZONE_MINUTE	Input parameter: string	-
	Output parameter: int	
TINYINT	cast as tinyint	cast as tinyint
TO_CHAR	Input parameters: (boolean|int|long|double|decimal [, string])	
	Output parameter: string	
TO_CSV	Input parameters: (struct[, map])	-
	Output parameter: string	
TO_DATE	Input parameters: (string[, string])	Input parameter: string
	Output parameter: date	Output parameter: date
TO_JSON	Input parameters: (string[, map])	-
	Output parameter: string	
TO_TIMESTAMP	Input parameters: (string[, string])	Input parameter: (string)
	Output parameter: timestamp	Output parameter: timestam

TO_UNIX_TIMESTAMP	Input parameters: (string[, string])	Input parameters: (string[, s
	Output parameter: timestamp	Output parameter: timestam
TO_UTC_TIMESTAMP	Input parameters: (string[, string])	Input parameters: (bigint|double|decimal|date| string)
	Output parameter: timestamp	Output parameter: timestam
TRANS_ARRAY	Input parameters: (int, string, cols...)	-
	Output parameter: same as cols type	
TRANS_COLS	Input parameters: (int, cols...)	
	Output parameters: (int, cols)	
TRANSLATE	Input parameters: (string, string, string)	-
	Output parameter: string	
TRIM	Input parameter: string	Input parameter: string
	Output parameter: string	Output parameter: string
TRUNC	Input parameters: (string|date|timestamp, string)	Input parameters: (string|da string)
	Output parameter: string	Output parameter: string
TRY_ADD	Input parameters: (bigint|double|decimal|date|timestamp|, bigint|double|decimal|date|timestamp)	Input parameters: (bigint|do bigint|double|decimal)
	Output parameter: same as input parameter	Output parameter: same as
TRY_DIVIDE	Input parameters: (bigint|double|decimal, bigint|double|decimal)	-
	Output parameter: double	
TYPEOF	Input parameter: T	Input parameter: T
	Output parameter: string	Output parameter: string
UCASE	Input parameter: string	Input parameter: string

	Output parameter: string	Output parameter: string
UNBASE64	Input parameter: string	Input parameter: string
	Output parameter: string	Output parameter: string
UNHEX	Input parameter: string	Input parameter: string
	Output parameter: binary	Output parameter: binary
UNIX_DATE	Input parameter: string|timestamp|date	Input parameters: string|time
	Output parameter: int	Output parameter: int
UNIX_MICROS	Input parameter: timestamp	Input parameter: timestamp
	Output parameter: bigint	Output parameter: bigint
UNIX_MILLIS	Input parameter: timestamp	Input parameter: timestamp
	Output parameter: bigint	Output parameter: bigint
UNIX_SECONDS	Input parameter: timestamp	Input parameter: timestamp
	Output parameter: bigint	Output parameter: bigint
UNIX_TIMESTAMP	Input parameters: [date|timestamp|string[, string]]	Input parameters: (date|time string])
	Output parameter: bigint	Output parameter: bigint
UPPER	Input parameter: string	Input parameter: string
	Output parameter: string	Output parameter: string
URL_DECODE	Input parameter: string	-
	Output parameter: string	
UUID	Input parameter: none	Input parameter: None
	Output parameter: string	Output parameter: string
VARIANCE	Input parameters: bigint|double|decimal	Input parameters: bigint|dou
	Output parameter: double	Output parameter: double
VAR_POP	Input parameters:	Input parameters: bigint|dou

	bigint|double|decimal	
	Output parameter: double	Output parameter: double
VAR_SAMP	Input parameters: bigint|double|decimal	Input parameters: bigint|dou
	Output parameter: double	Output parameter: double
VERSION	Input parameter: None	Input parameter: None
	Output parameter: string	Output parameter: string
WEEKDAY	Input parameters: date|timestamp|string	Input parameters: date|time
	Output parameter: int	Output parameter: int
WEEKOFYEAR	Input parameters: date|timestamp|string	Input parameters: date|time
	Output parameter: int	Output parameter: int
WIDTH_BUCKET	Input parameters: (bigint|double|decimal, bigint|double|decimal, bigint|double|decimal, int)	Input parameters: (bigint|do bigint|double|decimal, bigint int)
	Output parameter: int	Output parameter: int
WINDOW	Input parameters: (date|timestamp, string[, string[, string]])	-
	Output parameter: table	
XPATH	Input parameters: (string, string)	Input parameters: (string, st
	Output parameter: array	Output parameter: array
XPATH_BOOLEAN	Input parameters: (string, string)	Input parameters: (string, st
	Output parameter: boolean	Output parameter: boolean
XPATH_DOUBLE	Input parameters: (string, string)	Input parameters: (string, st
	Output parameter: double	Output parameter: double
XPATH_FLOAT	Input parameters: (string, string)	Input parameters: (string, st

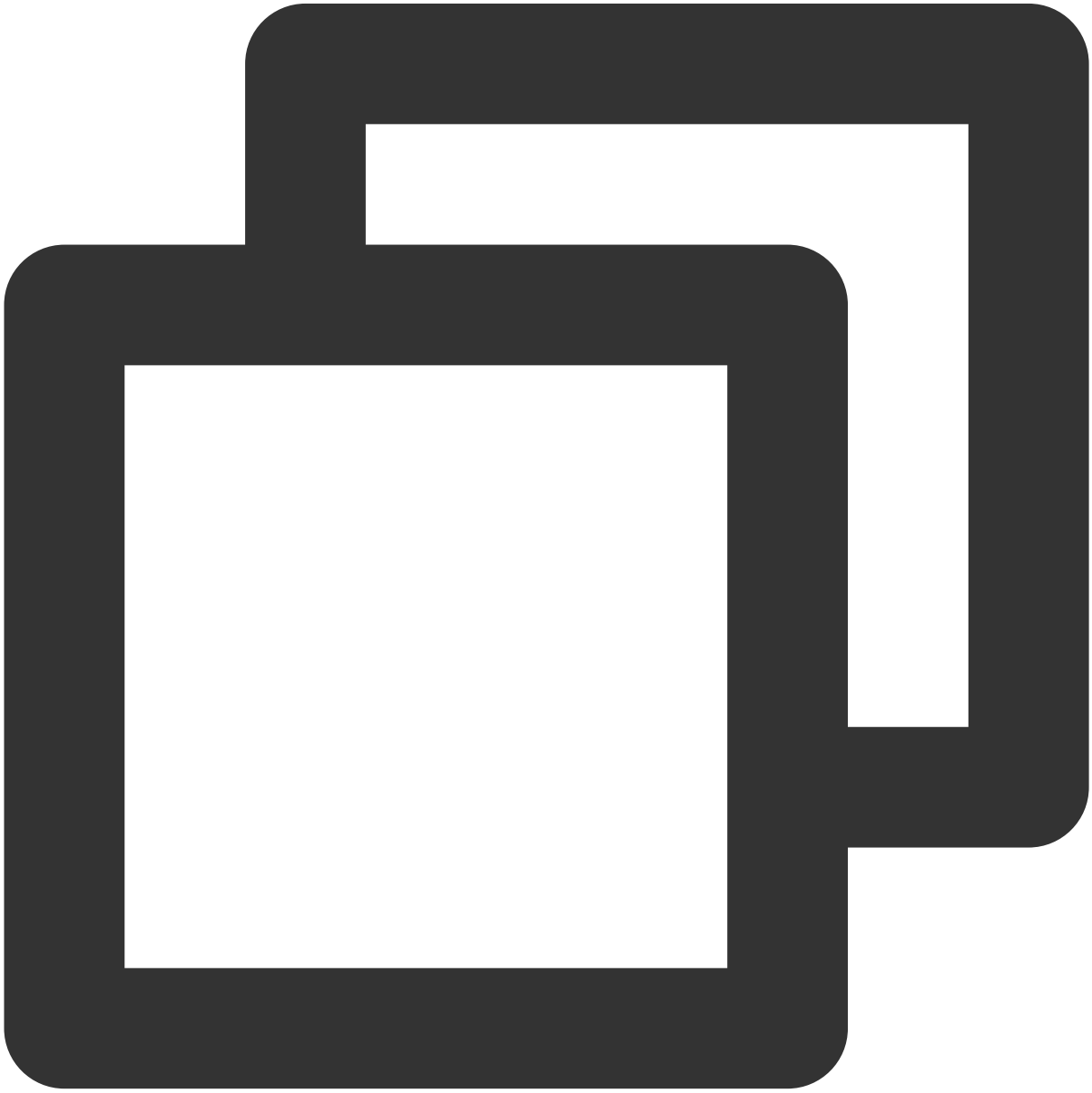
	Output parameter: float	Output parameter: float
XPATH_INT	Input parameters: (string, string)	Input parameters: (string, st
	Output parameter: int	Output parameter: int
XPATH_LONG	Input parameters: (string, string)	Input parameters: (string, st
	Output parameter: bigint	Output parameter: bigint
XPATH_NUMBER	Input parameters: (string, string)	Input parameters: (string, st
	Output parameter: double	Output parameter: double
XPATH_STRING	Input parameters: (string, string)	Input parameters: (string, st
	Output parameter: string	Output parameter: string
XXHASH64	Input parameters: (T, U, ...)	Input parameters: (T, U, ...)
	Output parameter: bigint	Output parameter: bigint
YEAR	Input parameters: date|timestamp|string	Input parameter: date|timest
	Output parameter: int	Output parameter: int

Binary Functions

Last updated：2024-08-07 17:30:56

CRC32

Function statement:



```
CRC32(<expr> string|binary)
```

Supported engines: `SPARKSQL` and `PRESTO`

Usage instructions: Compute the cyclic redundancy check value of the expression using the CRC32 algorithm.

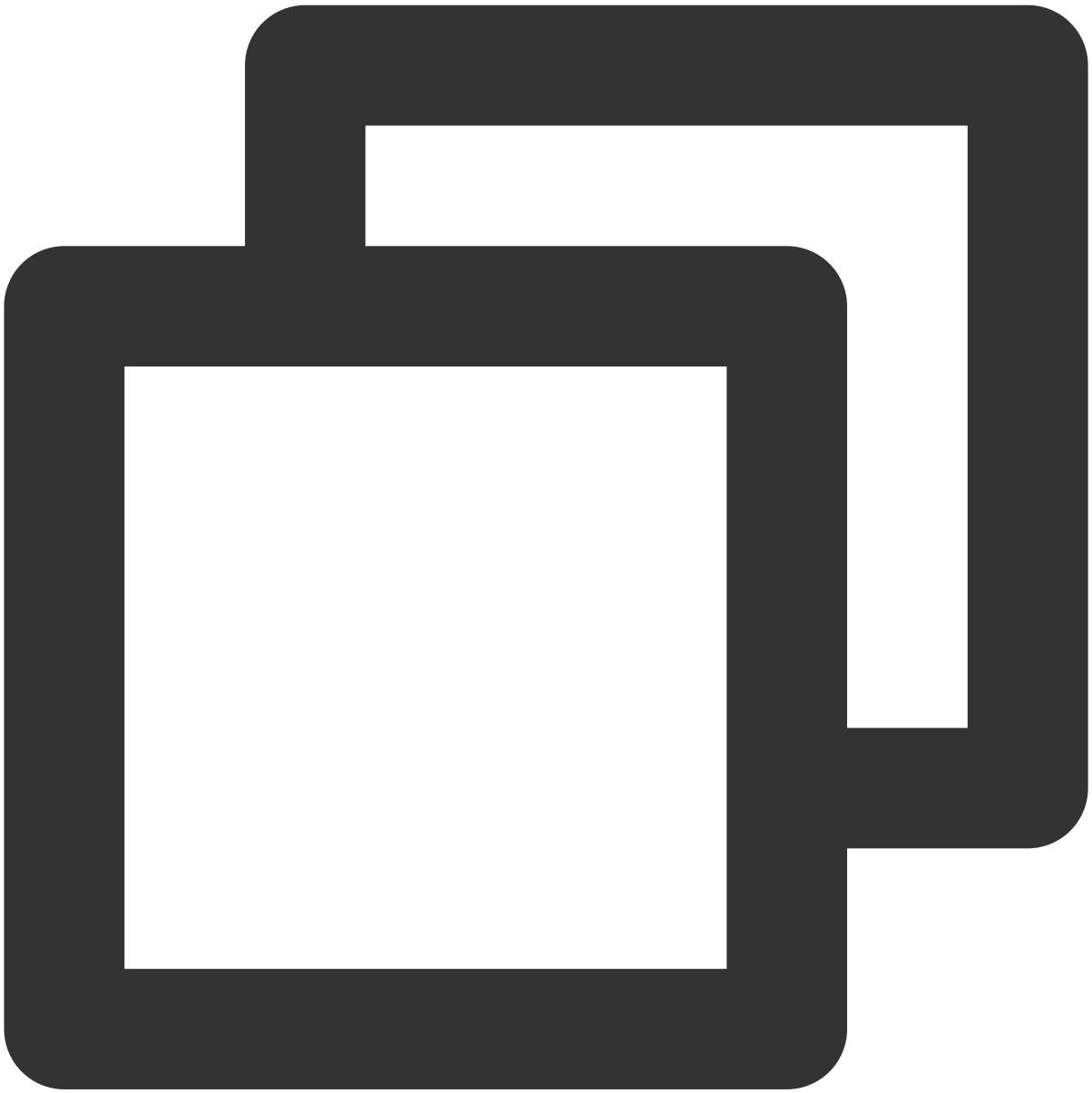
Return type: `bigint`

Example:

```
> select crc32('tencent');
820633257
```

MD5

Function statement:



```
MD5(<expr string|binary)
```

Supported engines: SPARKSQL and PRESTO

Usage instructions: Return the MD5 128-bit checksum as a hexadecimal string.

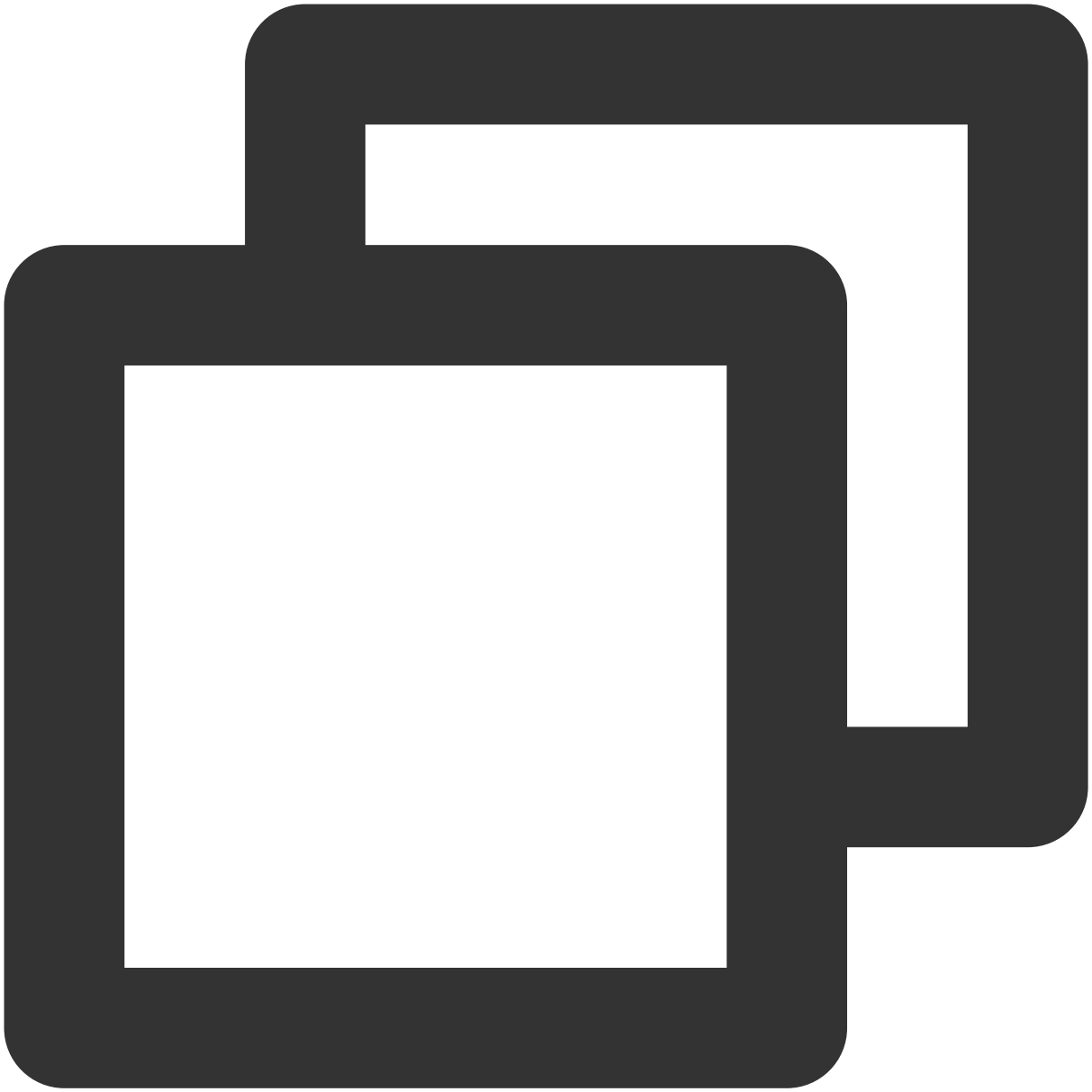
Return type: string

Example:

```
> select md5('tencent');
3da576879001c77b442b9f8ef95c09d6
```

HASH

Function statement:



```
HASH(&lt;expr1&gt; any[, &lt;expr2&gt; any, ...])
```

Supported engines: SPARKSQL and PRESTO

Usage instructions: Return the hash value of all parameters. The calculation methods in `SPARKSQL` and `PRESTO` are inconsistent, so the results may differ.

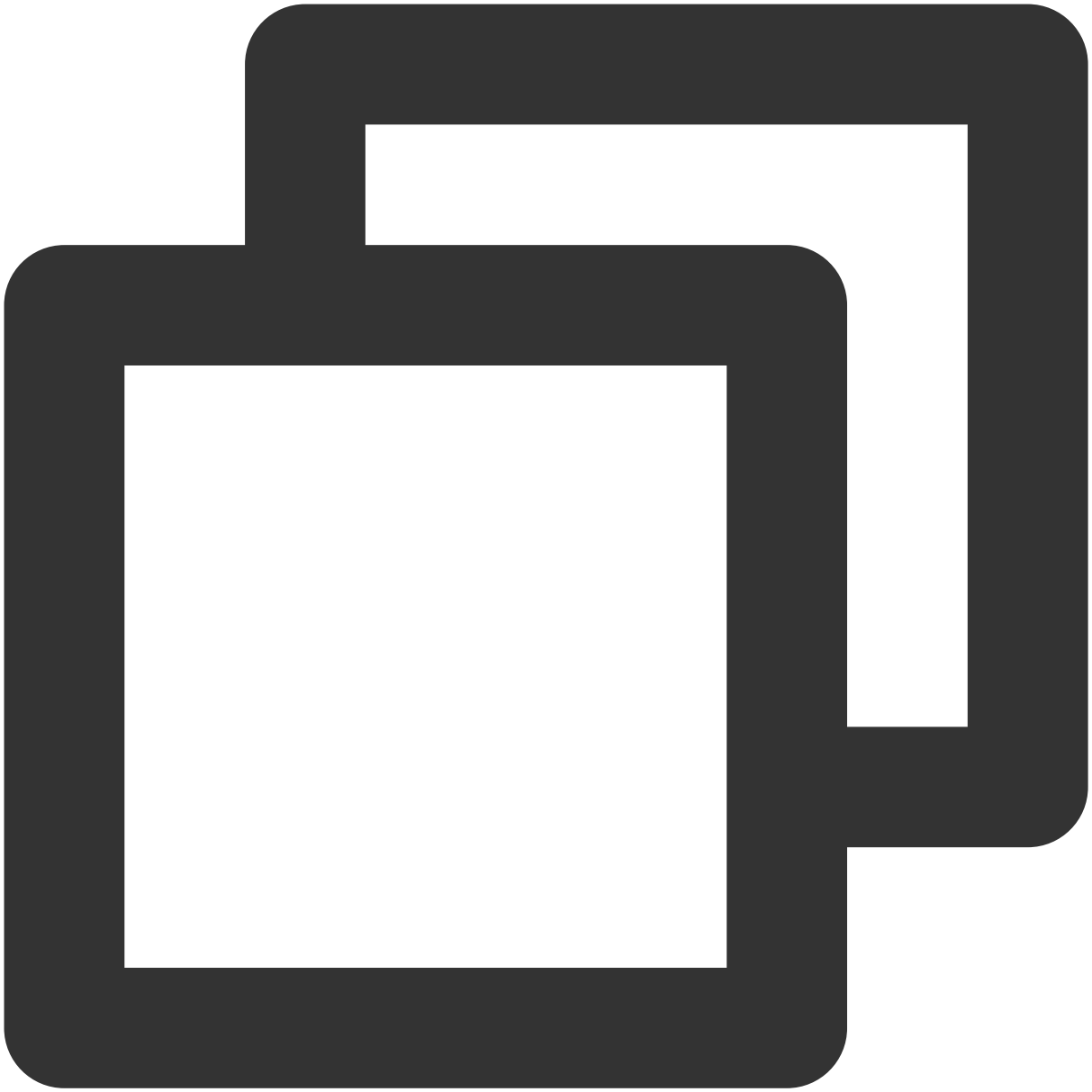
Return type: `integer`

Example:

```
> SELECT hash('tencent', array(123), 2);
-412995102
```

XXHASH64

Function statement:



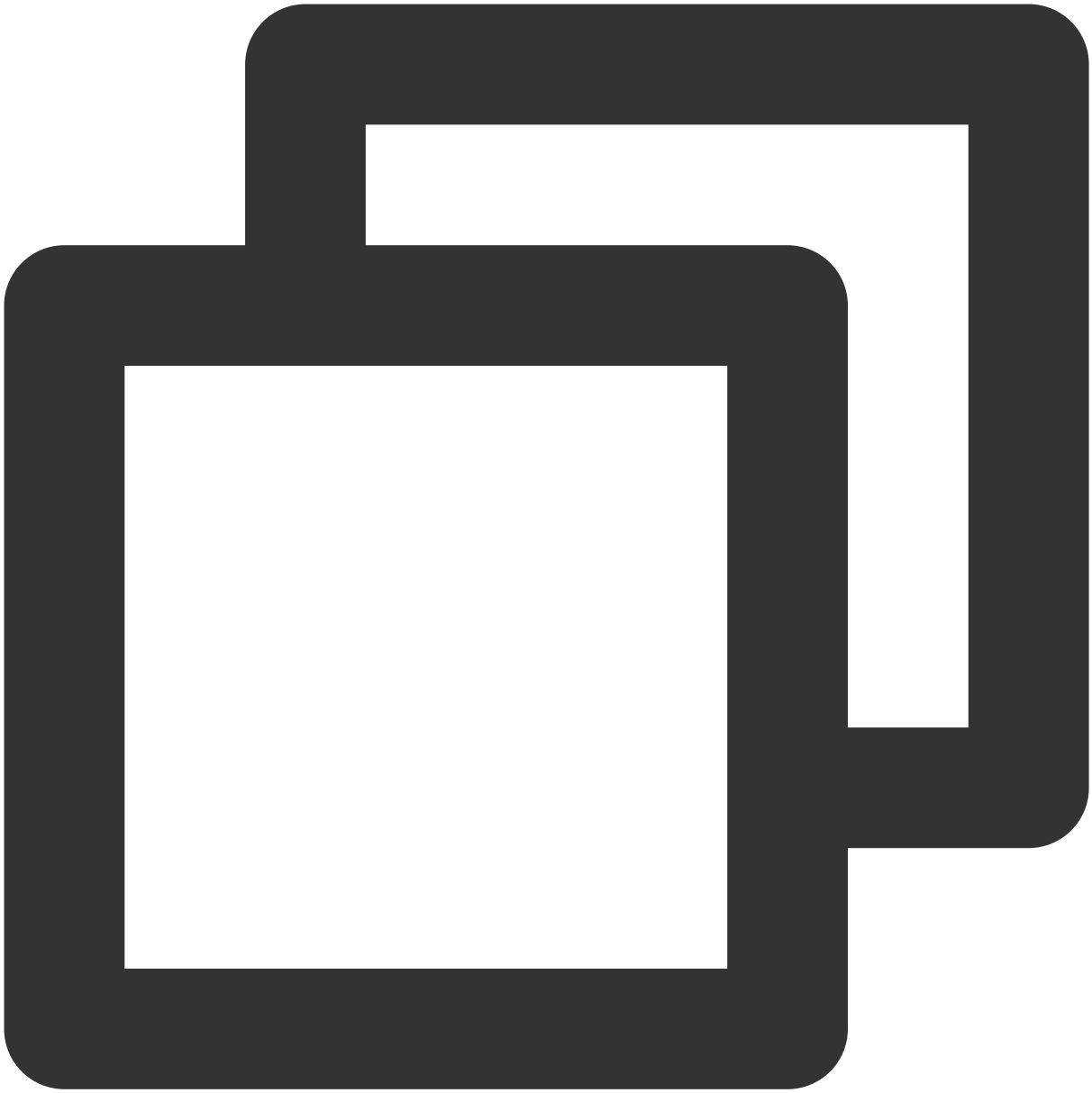
```
XXHASH64(&lt;expr1&gt; any[, &lt;expr2&gt; any, ...])
```

Supported engines: SPARKSQL and PRESTO

Usage instructions: Return the 64-bit hash value of the parameter. The calculation methods in `SPARKSQL` and `PRESTO` are inconsistent, so the results may differ.

Return type: bigint

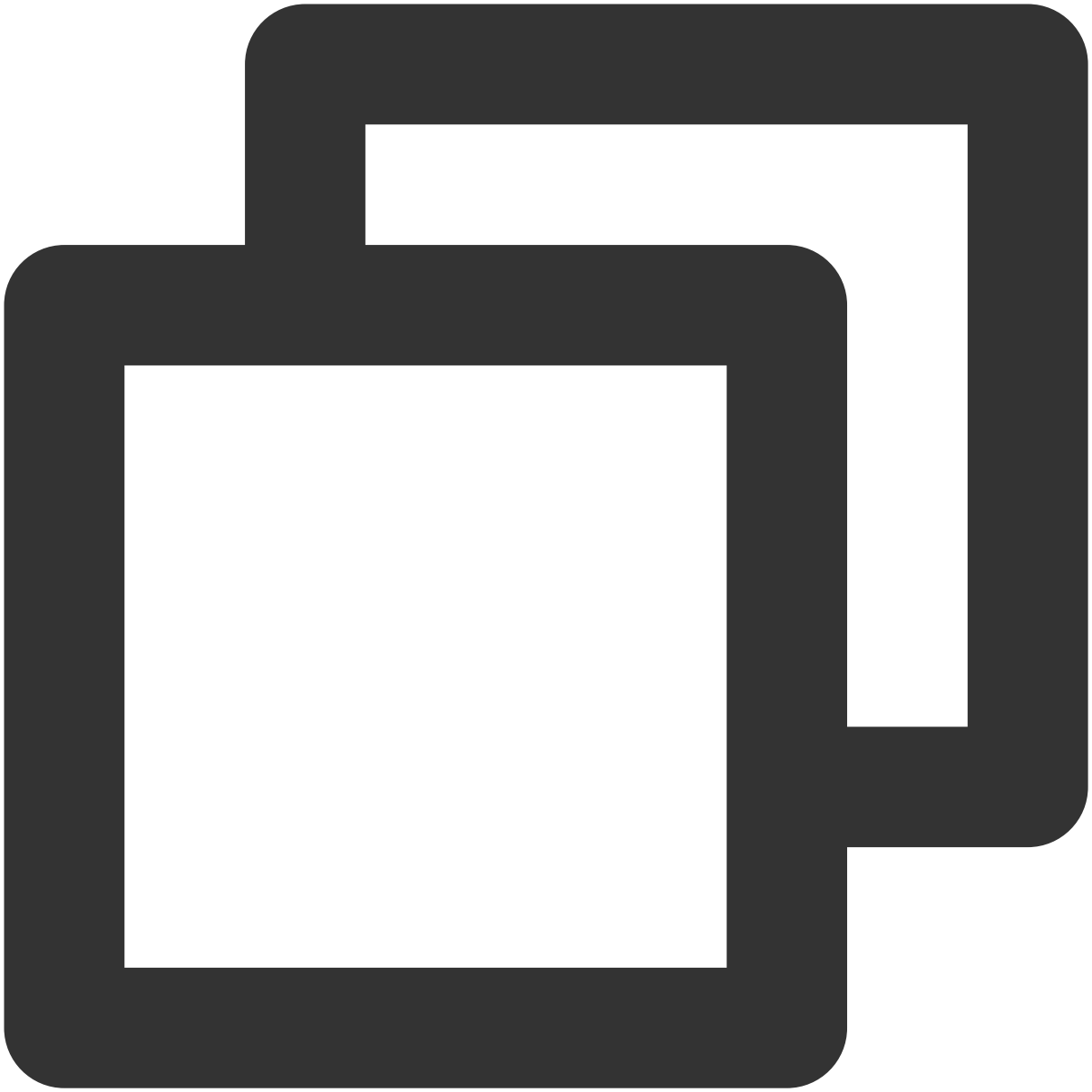
Example:



```
> SELECT xxhash64('tencent', array(123), 2);
-1900074178543885261
```

SHA

Function statement:



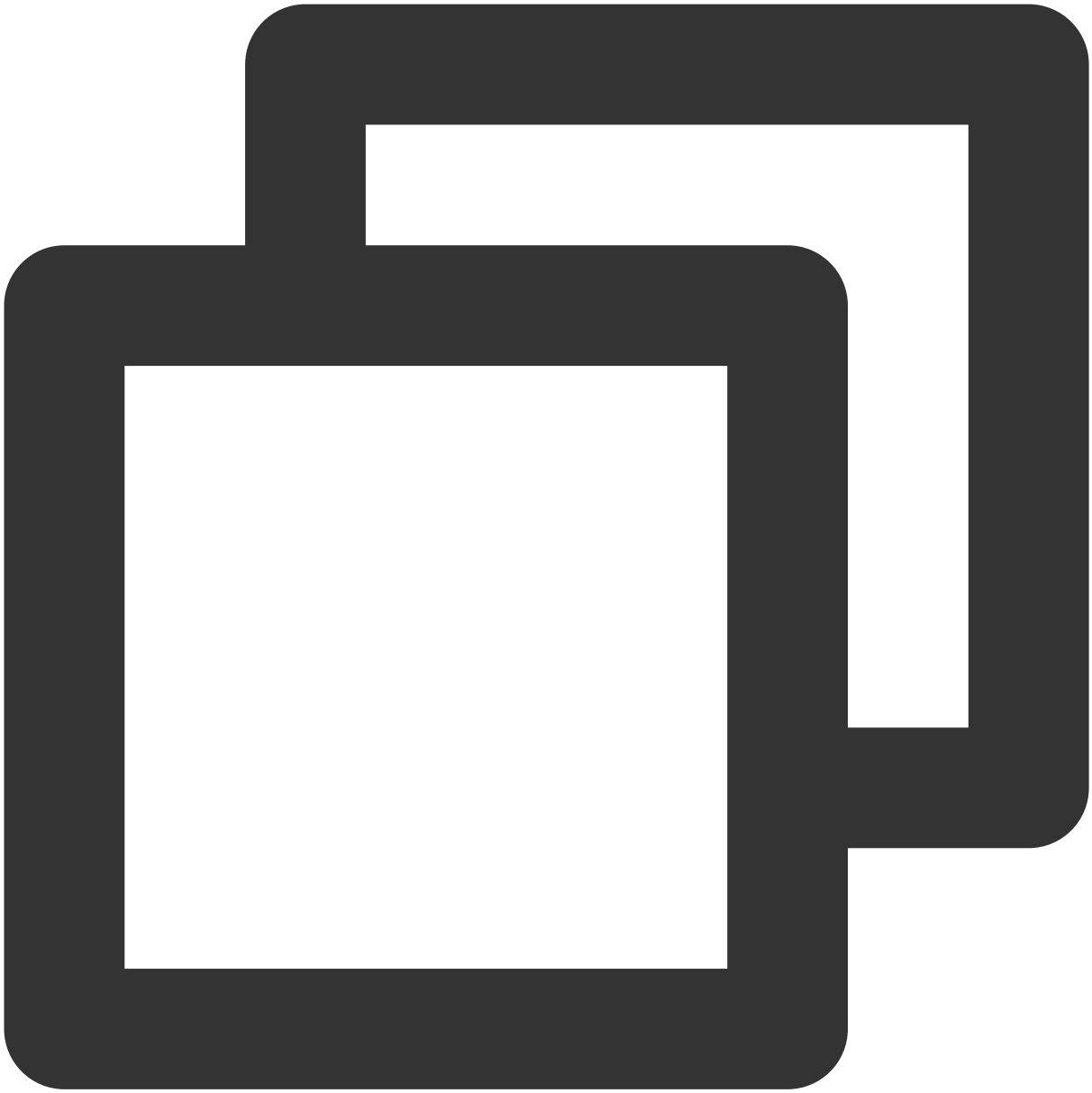
```
SHA(<expr> string|binary)
```

Supported engines: SPARKSQL and PRESTO

Usage instructions: Return the SHA-1 hash value of expr as a hexadecimal string.

Return type: `string`

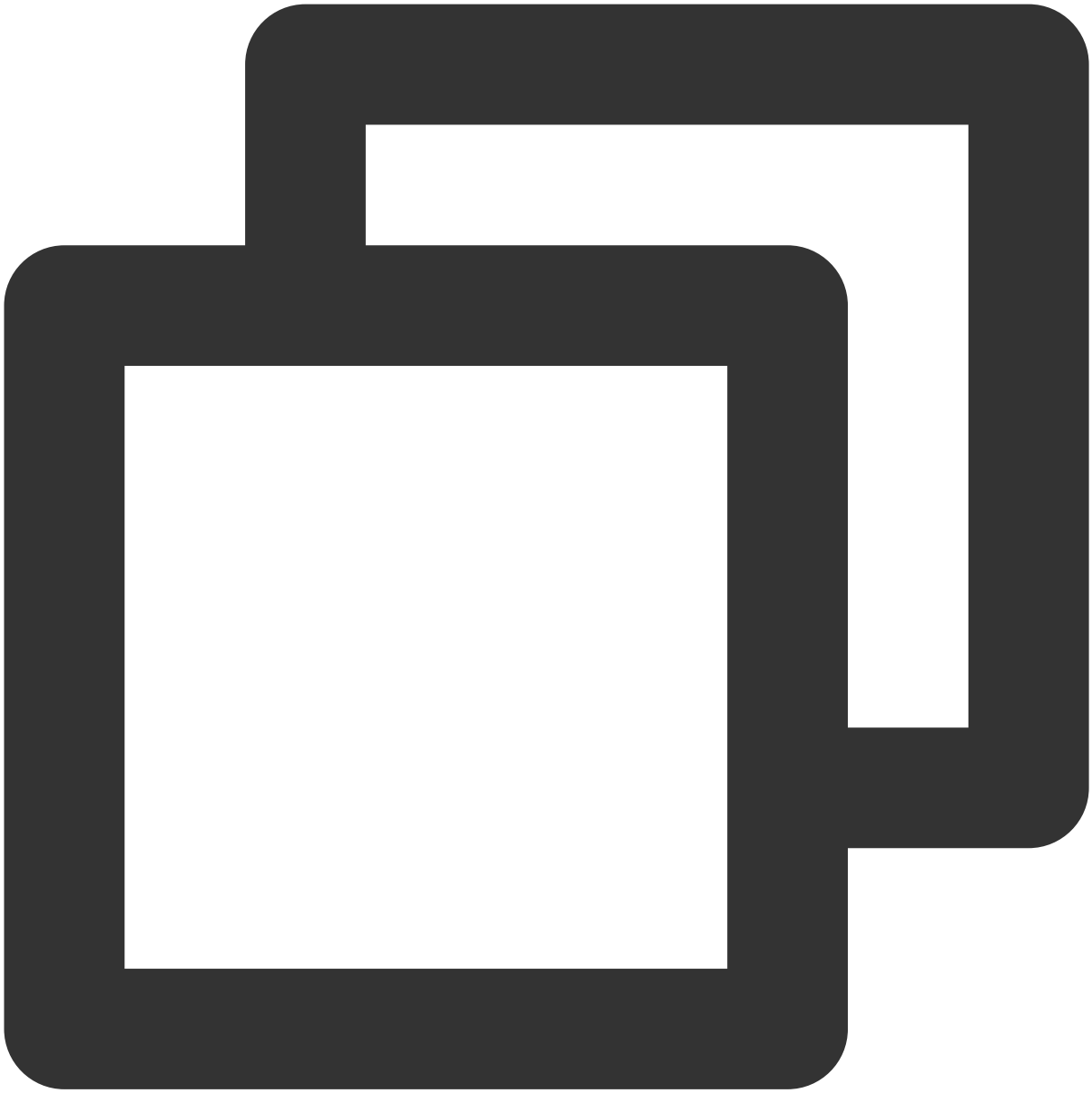
Example:



```
> select sha('tencent');
f94b2c96e2f127726ef4bcec6bc779f0f2e7888f
```

SHA1

Function statement:



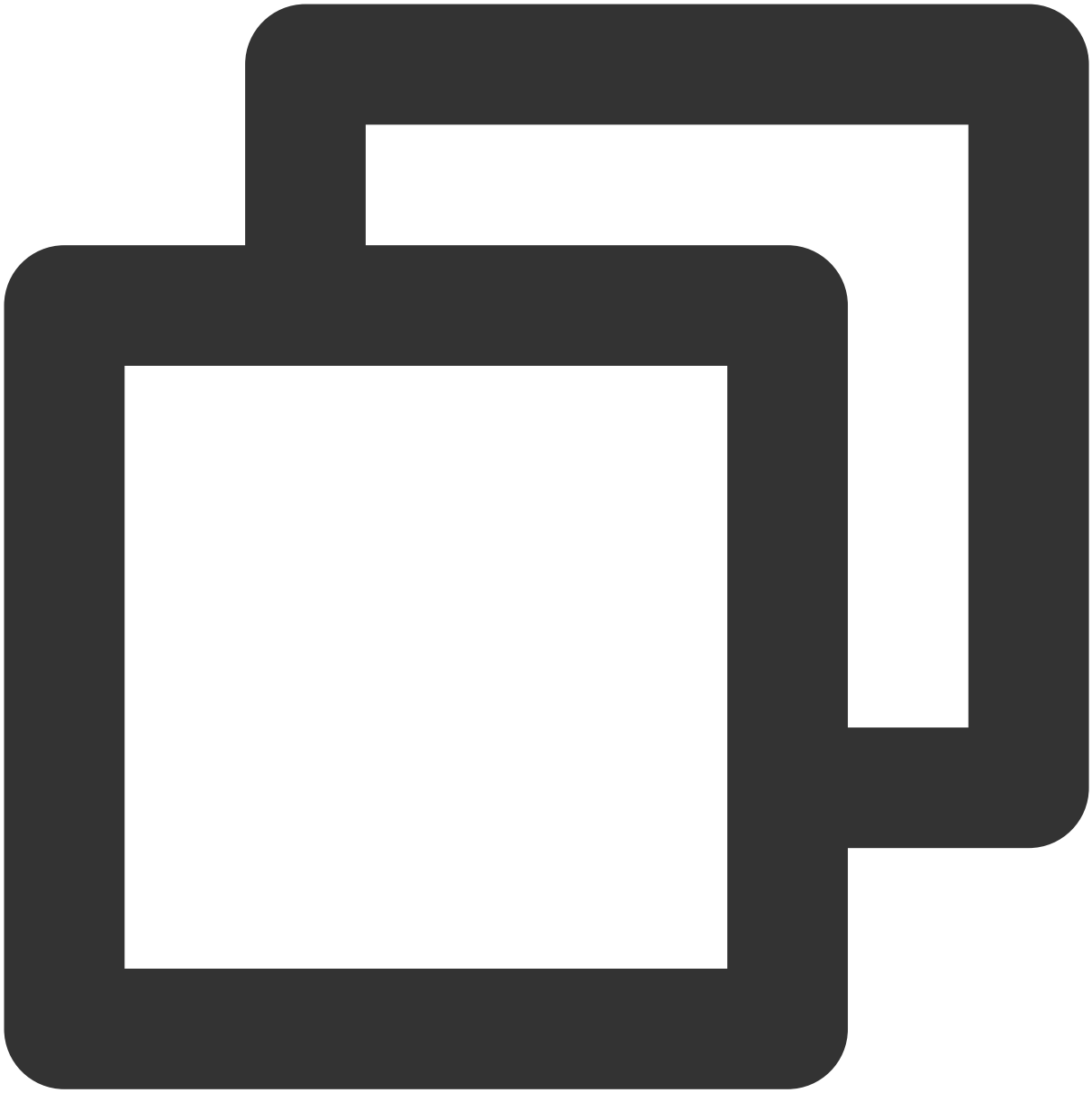
```
SHA1(<expr> string|binary)
```

Supported engines: SPARKSQL and PRESTO

Usage instructions: Return the SHA-1 hash value of `expr` as a hexadecimal string.

Return type: `string`

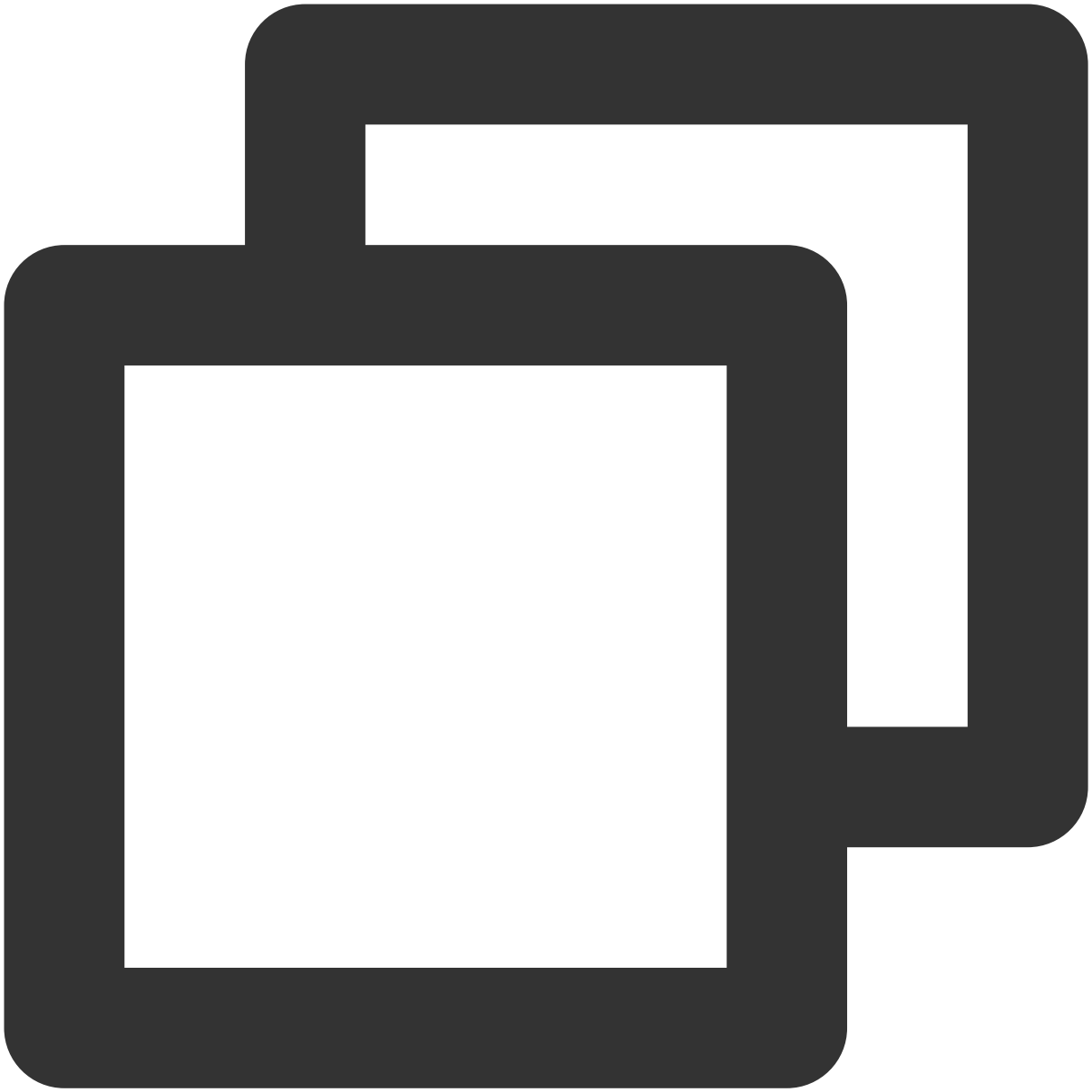
Example:



```
> select sha1('tencent');
f94b2c96e2f127726ef4bcec6bc779f0f2e7888f
```

SHA2

Function statement:



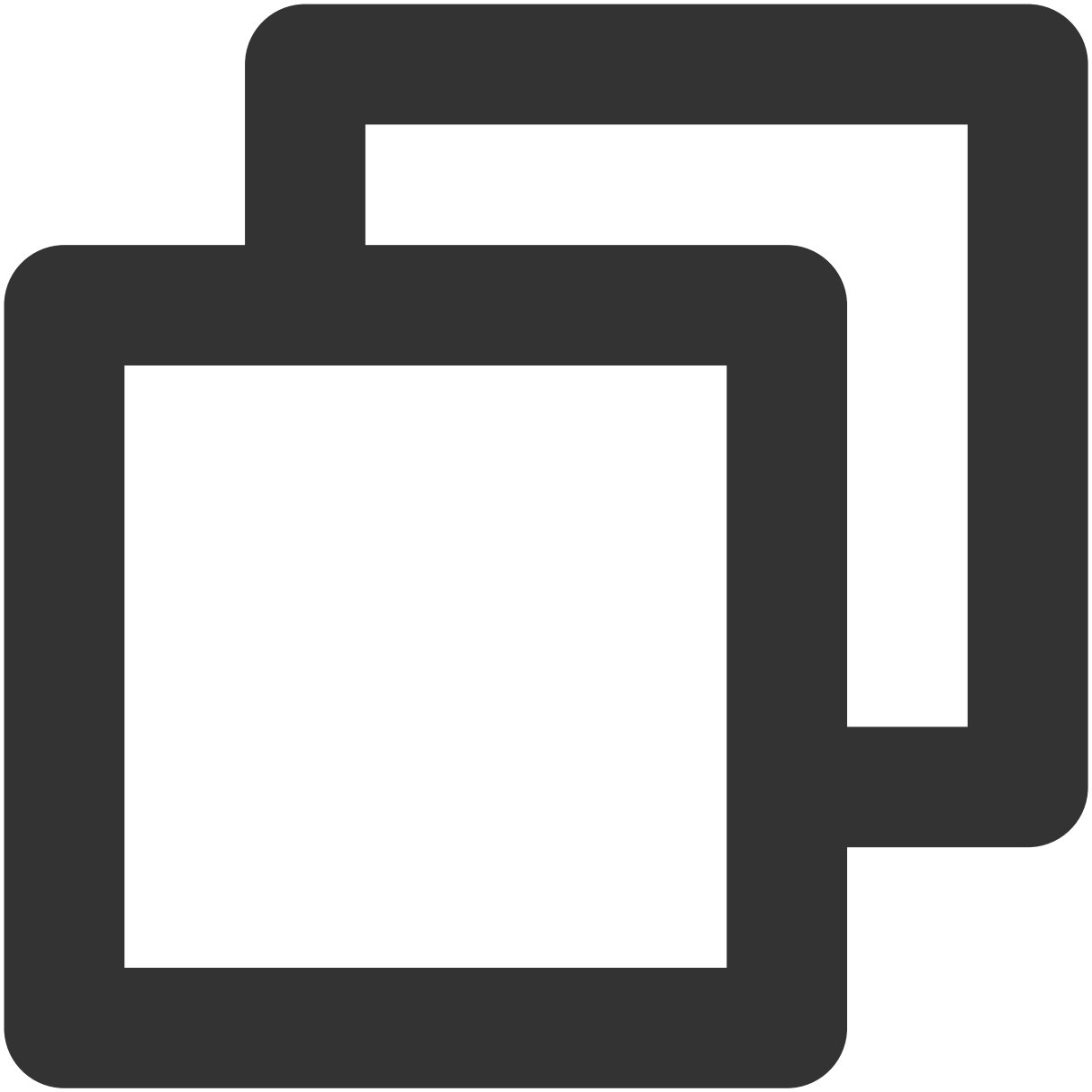
```
SHA2(expr string|binary, bitLength int)
```

Supported engines: SPARKSQL and PRESTO

Usage instructions: Return the checksum of `expr` for the SHA-2 family as a hexadecimal string. SHA-224, SHA-256, SHA-384, and SHA-512 are supported. A bit length of 0 equals to 256.

Return type: `string`

Example:



```
> select sha2('tencent', 256);
9c8ae69b84f21a2e46df9edf0063a697afec050188ff2884ddc8ab32b5e58c43
```

AES_ENCRYPT

Function statement:

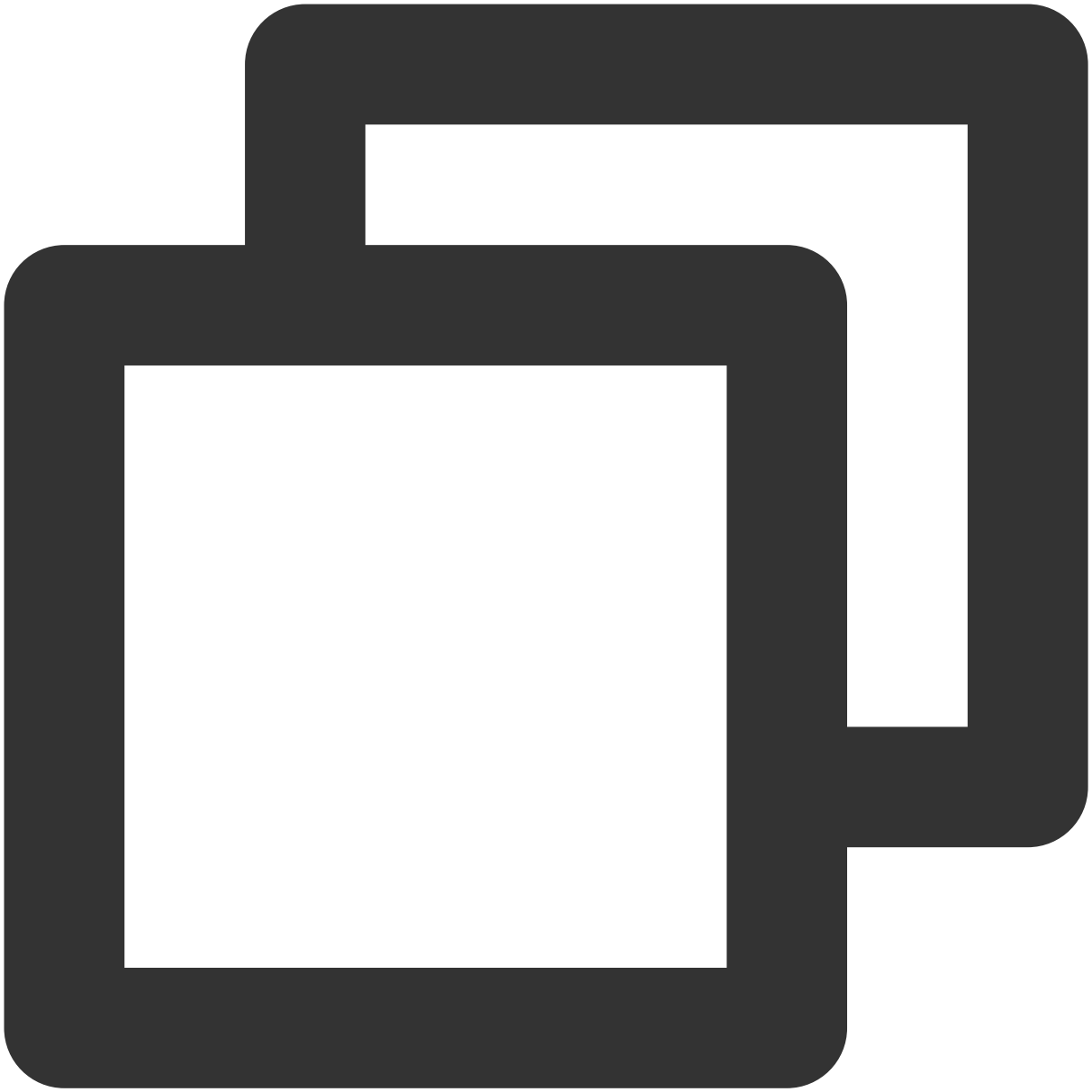
```
AES_ENCRYPT(<expr> string|binary, <key> string|binary)
```

Supported engines: PRESTO

Usage instructions: Encrypt `expr` using the AES algorithm.

Return type: `binary`

Example:



```
> SELECT hex(aes_encrypt('tencent', '0000111122223333'));
B99B99CE3359A736DBB9811ED8815C01
```

AES_DECRYPT

Function statement:

```
AES_DECRYPT(<expr> string|binary, <key> string|binary)
```

Supported engine: `PRESTO`

Usage instructions: Decrypt `expr` using the AES algorithm.

Return type: `binary`

Example:

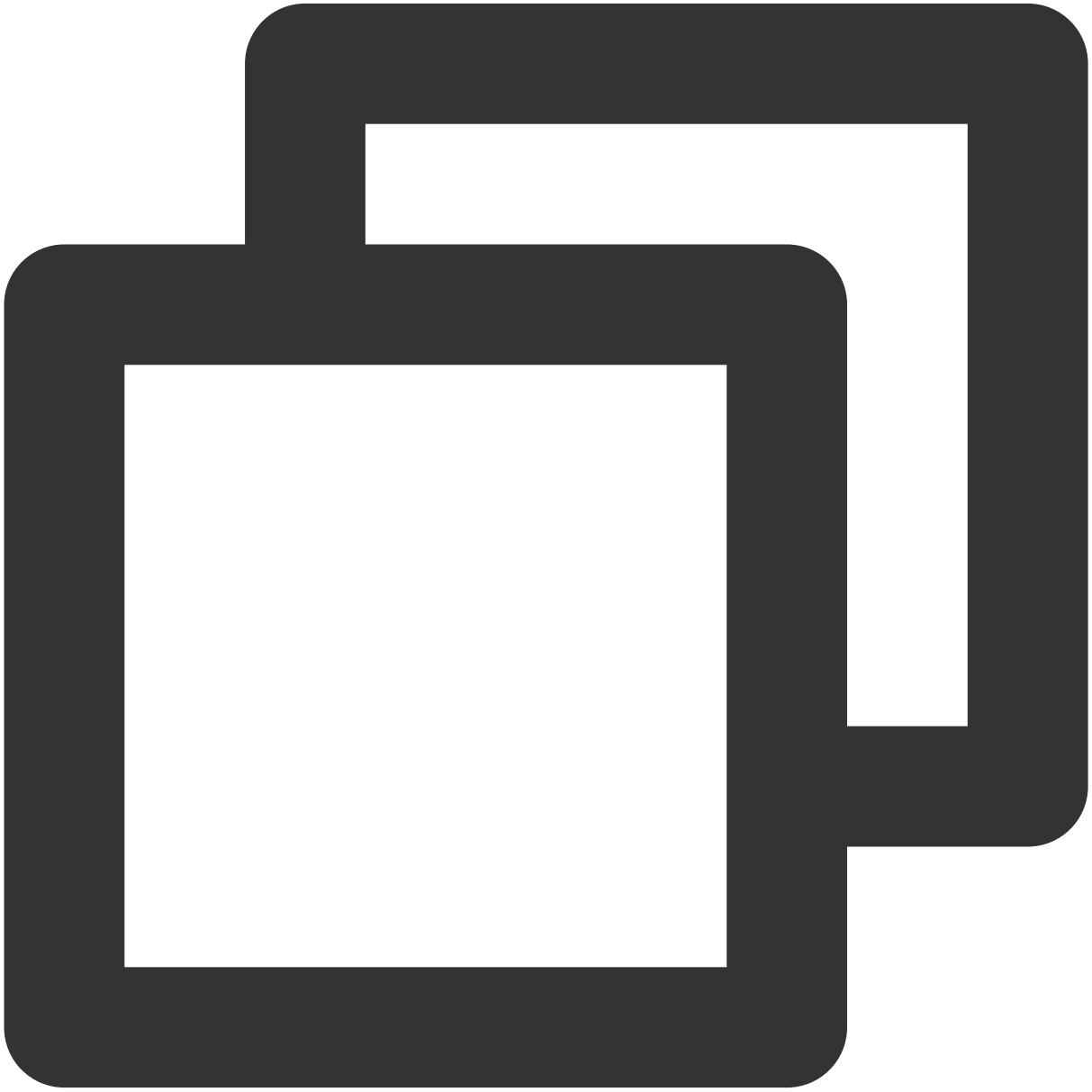
```
> SELECT aes_decrypt(unhex('B99B99CE3359A736DBB9811ED8815C01'), '0000111122223333')
tencent
```

Bitwise Functions

Last updated：2024-08-07 17:31:33

BIT_COUNT

Function statement:



```
BIT_COUNT(<expr> bigint|boolean)
```

Supported engines: SparkSQL and Presto

Usage instructions: Set `expr` to an unsigned 64-bit integer and return the number of bits that are 1. If the parameter is NULL, NULL is returned.

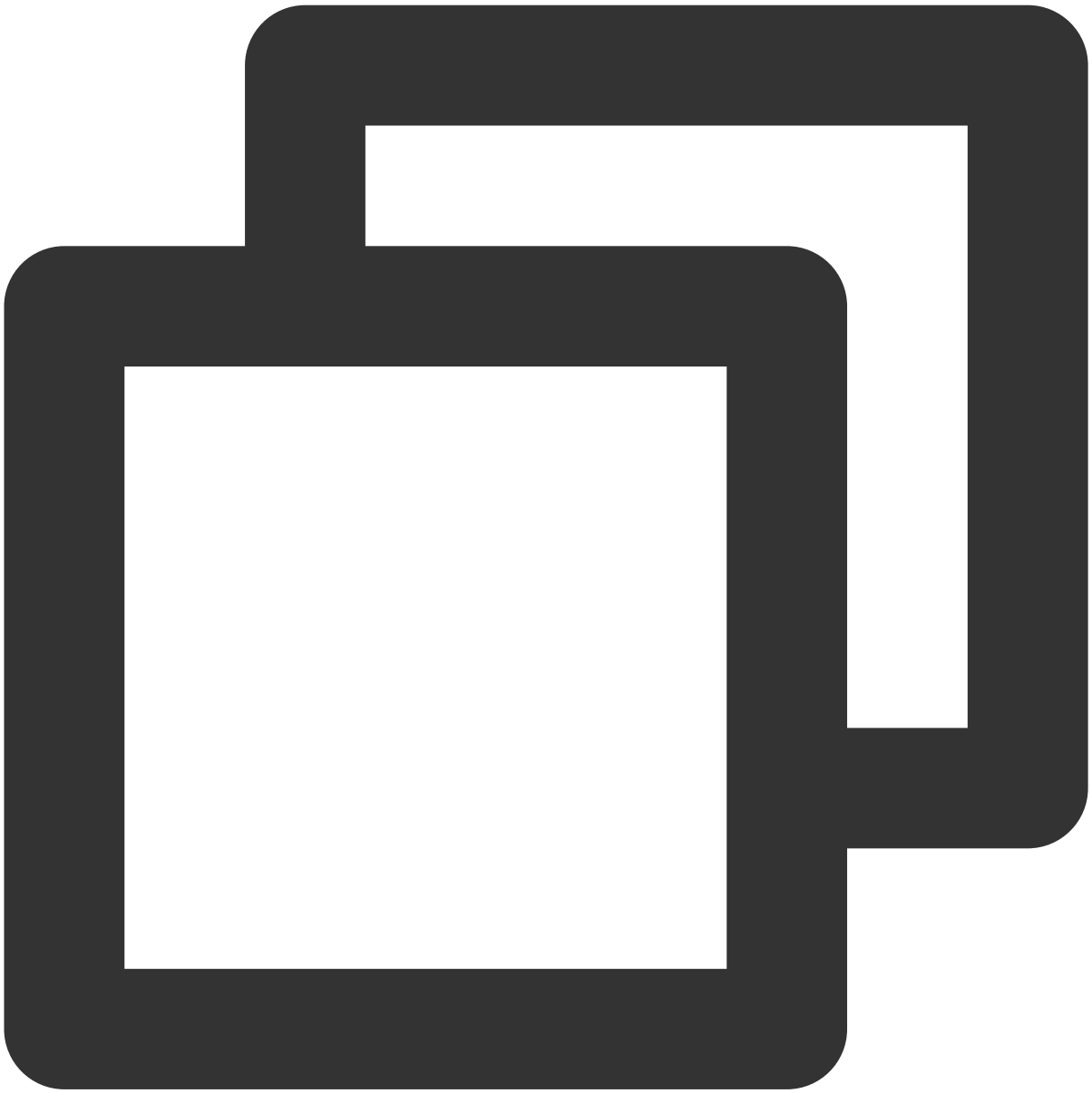
Return type: `integer`

Example:

```
> SELECT bit_count(5);
 2
```

BIT_GET

Function statement:



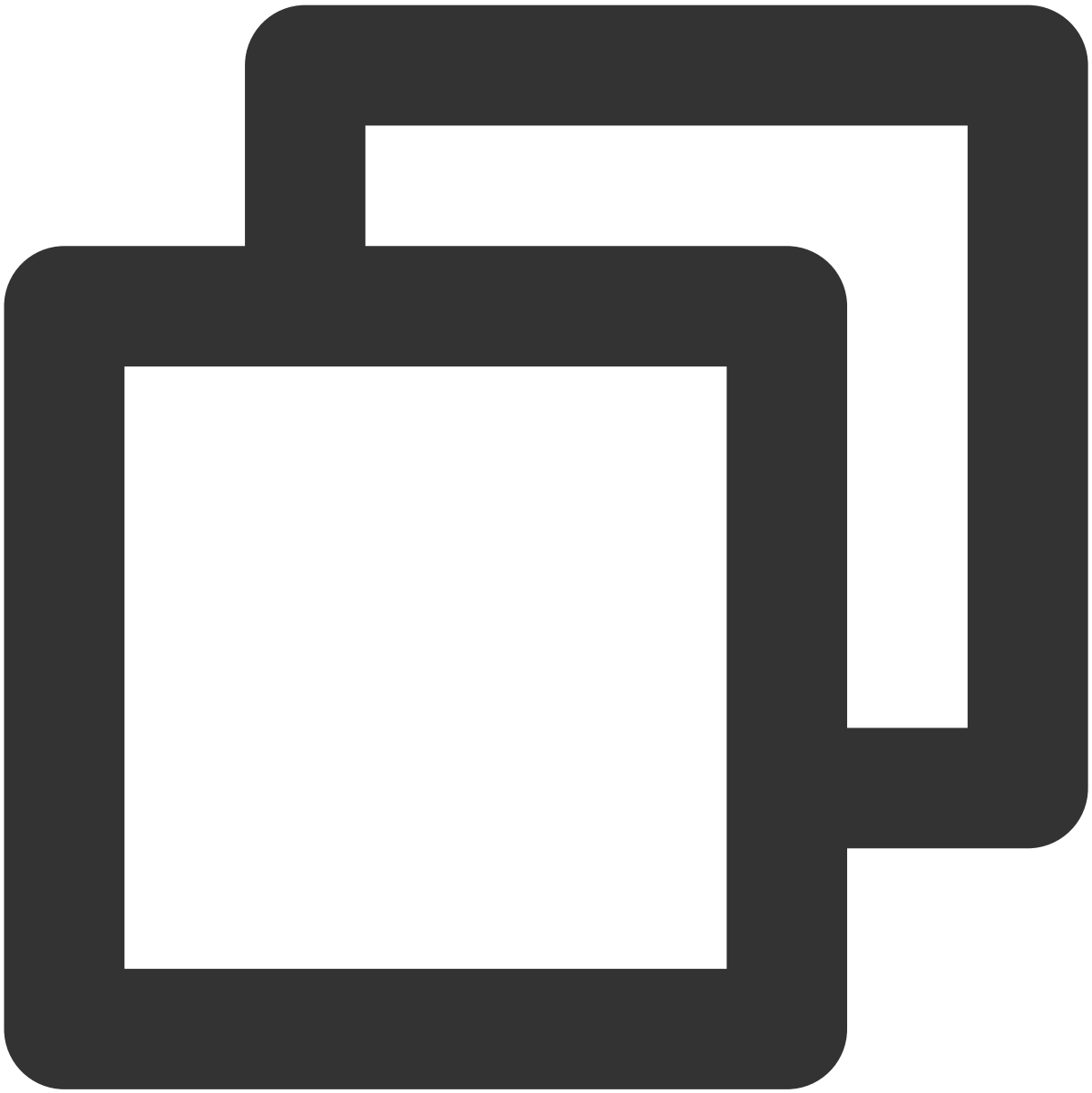
```
BIT_GET(<expr> bigint|boolean, <pos> integer)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the value of the bit at the specified position (0 or 1). Positions are numbered from right to left, starting at zero. The position parameter cannot be negative.

Return type: `integer`

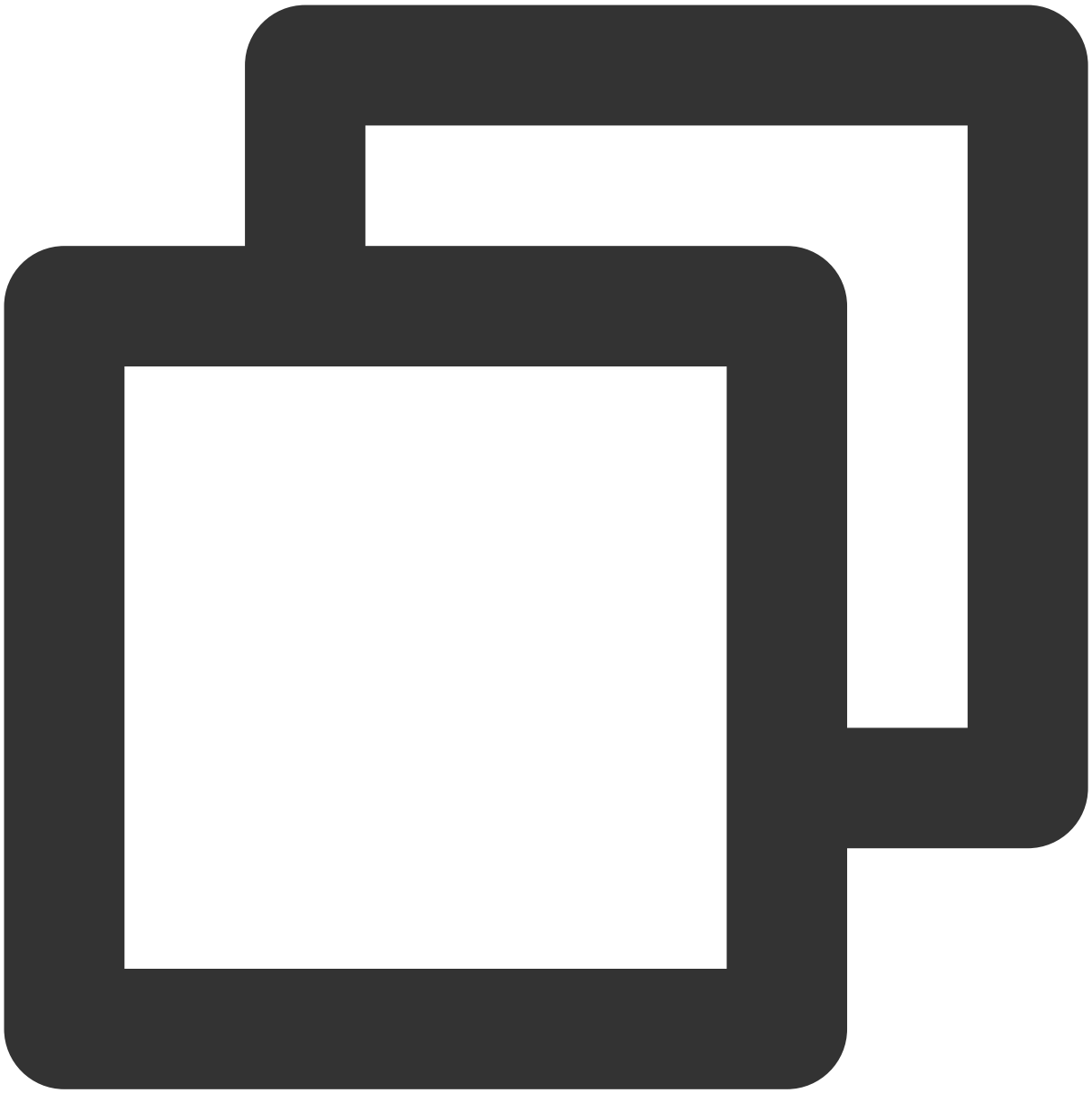
Example:



```
> SELECT bit_get(11, 0);
 1
> SELECT bit_get(11, 2);
 0
```

GETBIT

Function statement:



```
GETBIT(<expr> bigint|boolean, <pos> integer)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the value of the bit at the specified position (0 or 1). Positions are numbered from right to left, starting at zero. The position parameter cannot be negative.

Return type: `integer`

Example:

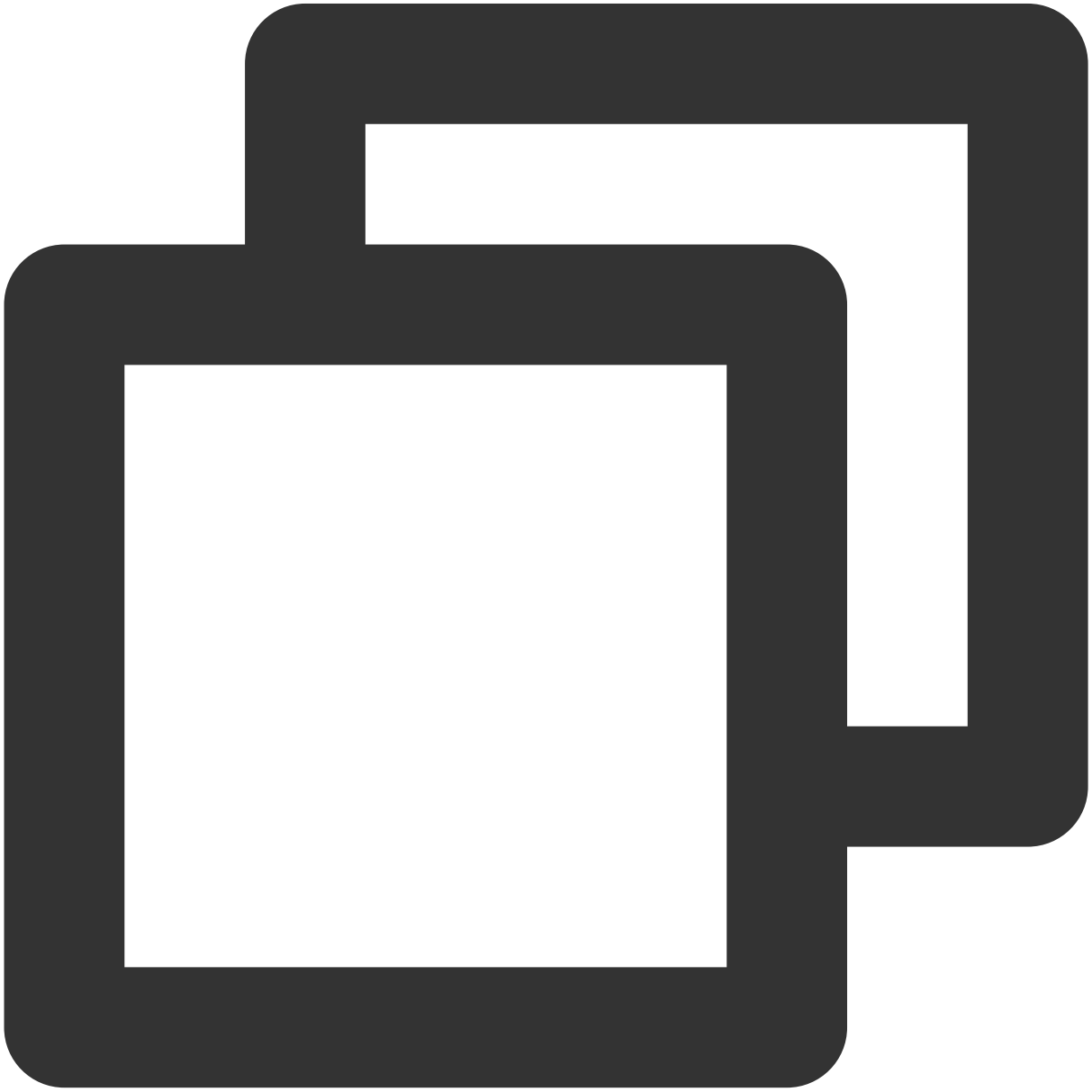
```
> SELECT getbit(11, 0);
 1
> SELECT getbit(11, 2);
 0
```

Collection Functions

Last updated：2024-08-07 17:31:46

ARRAY

Function statement:



```
ARRAY(<e1> T, ..., <en> T)
```

Supported engines: SparkSQL and Presto

Usage instructions: Generate an array from the given elements.

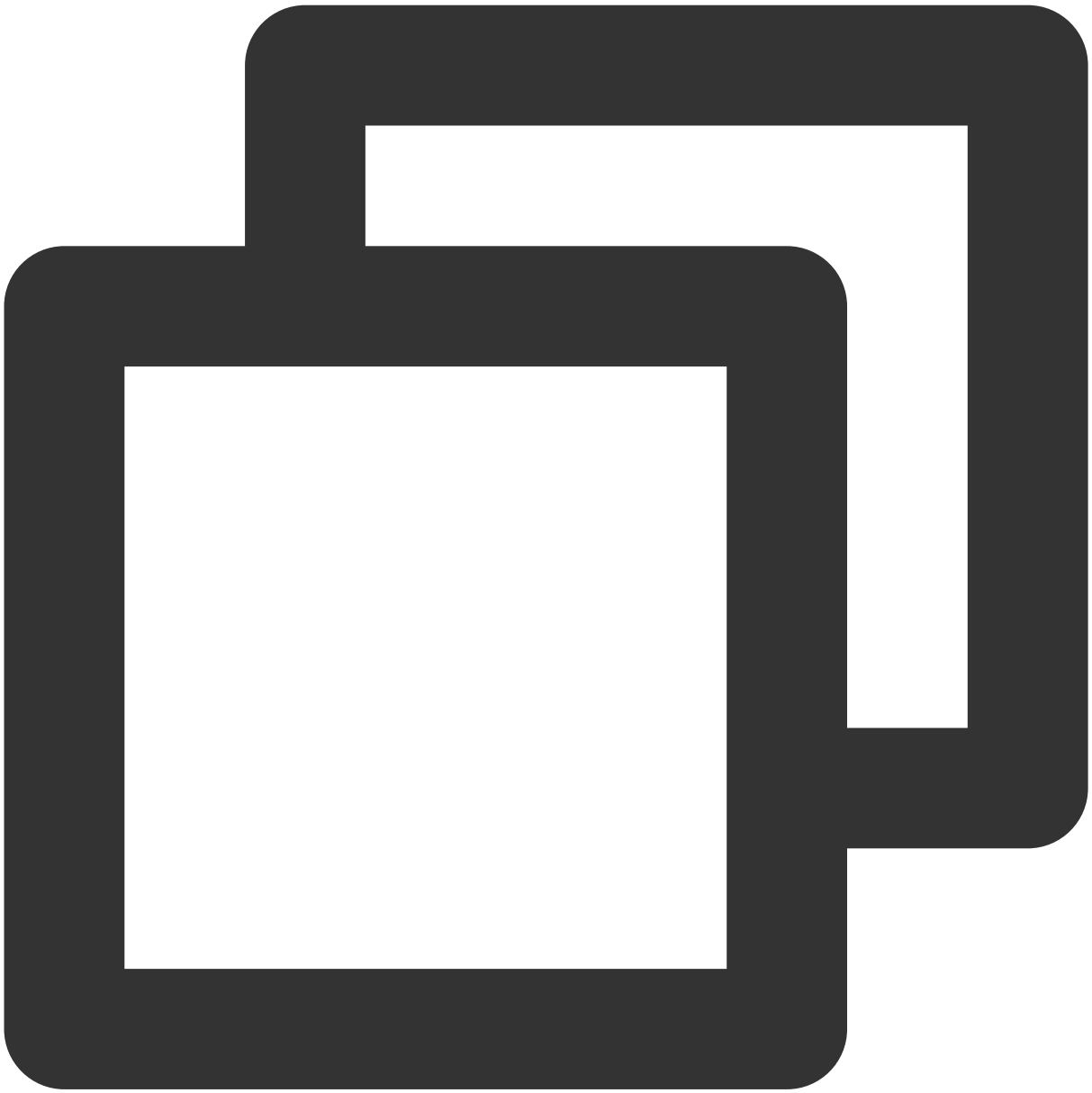
Return type: array<T>

Example:

```
> SELECT array(1, 2, 3);
 [1,2,3]
```

FILTER

Function statement:



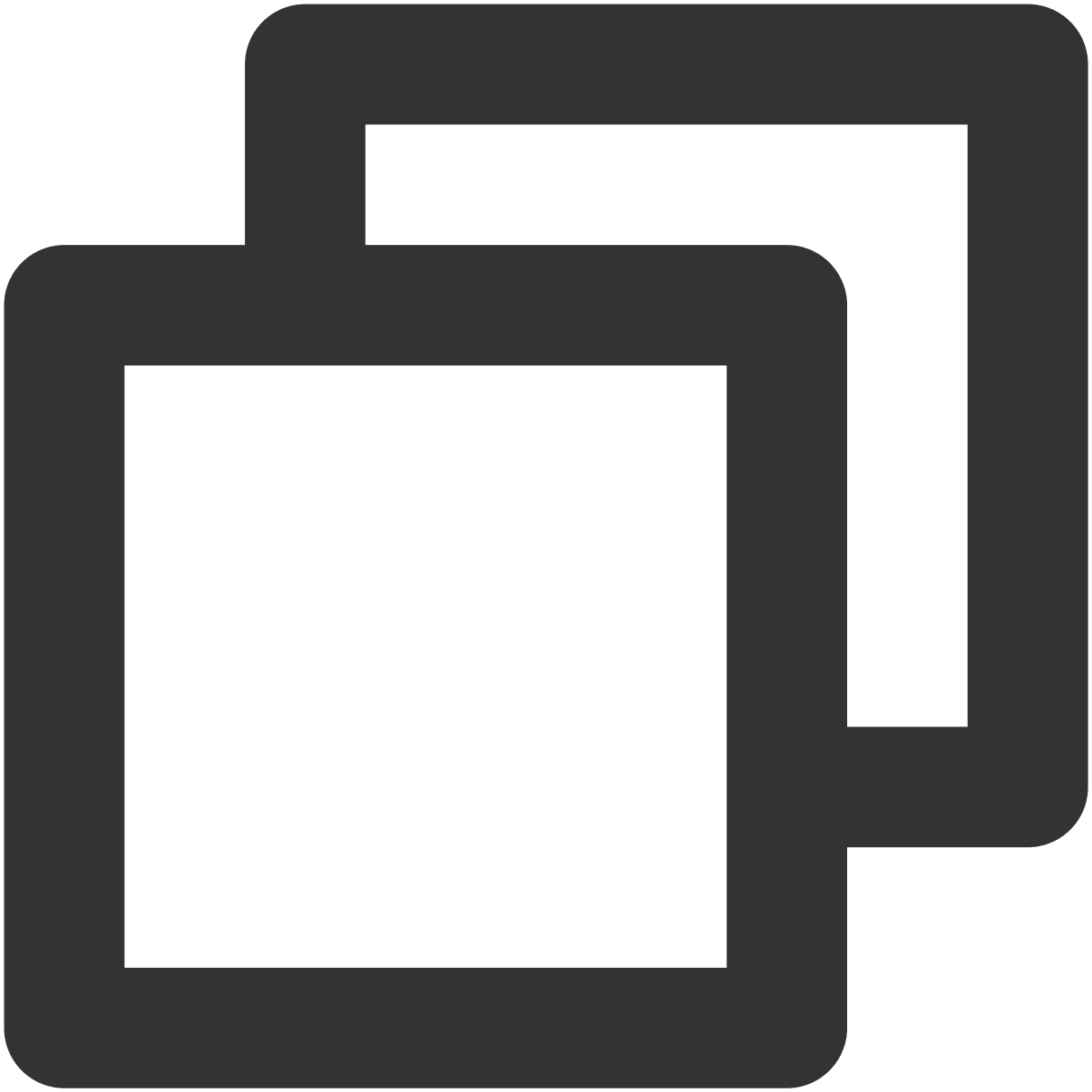
```
FILTER(<expr> array<T>, <predicate> function(T[, integer])->boolean)
```

Supported engine: SparkSQL

Usage instructions: Filter the input array using the given predicate.

Return type: array<T>

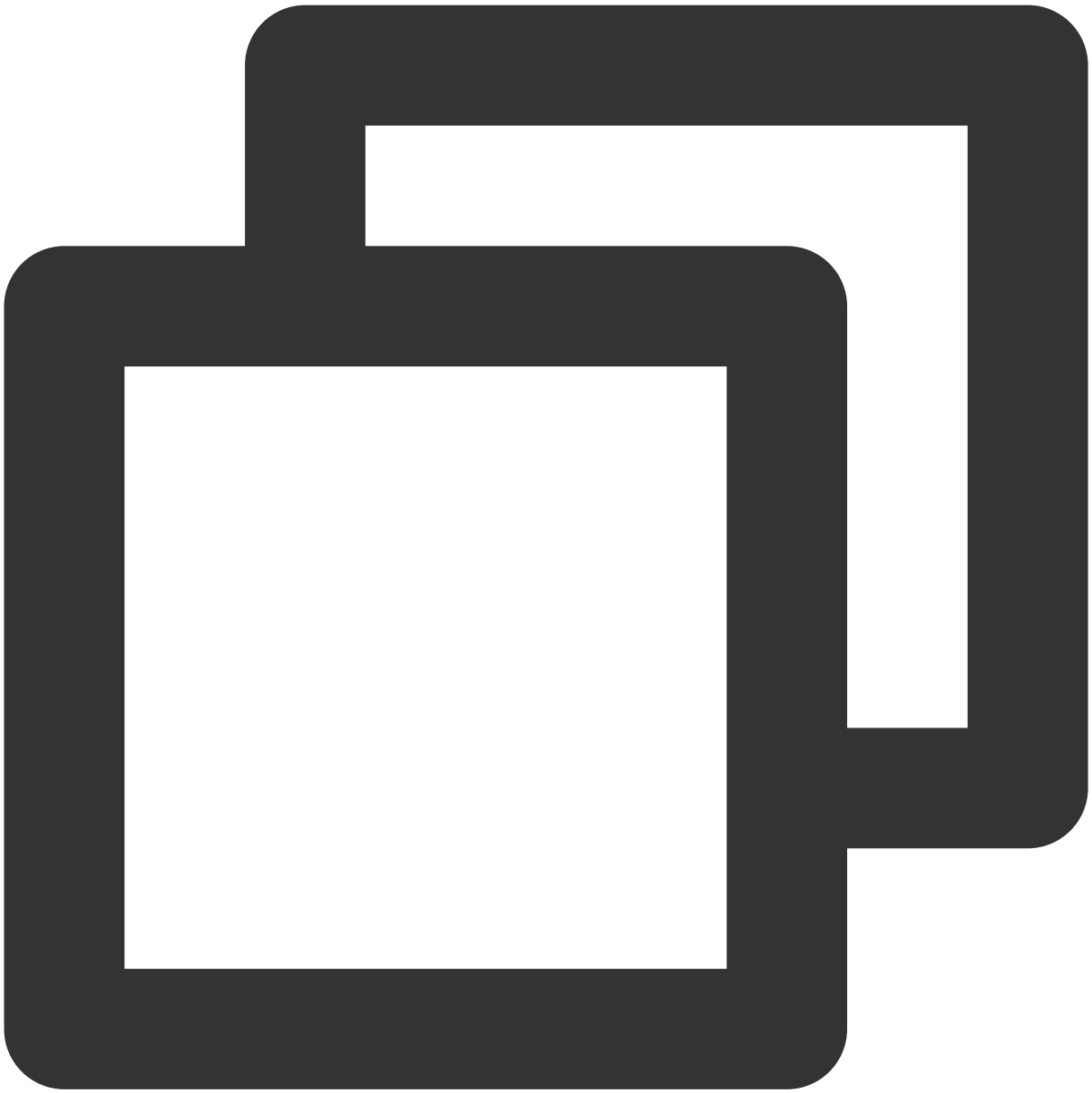
Example:



```
> SELECT `filter`(array(1, 2, 3), x -> x % 2 == 1);
 [1,3]
> SELECT `filter`(array(0, 2, 3), (x, i) -> x > i);
 [2,3]
> SELECT `filter`(array(0, null, 2, 3, null), x -> x IS NOT NULL);
 [0,2,3]
```

TRANSFORM

Function statement:



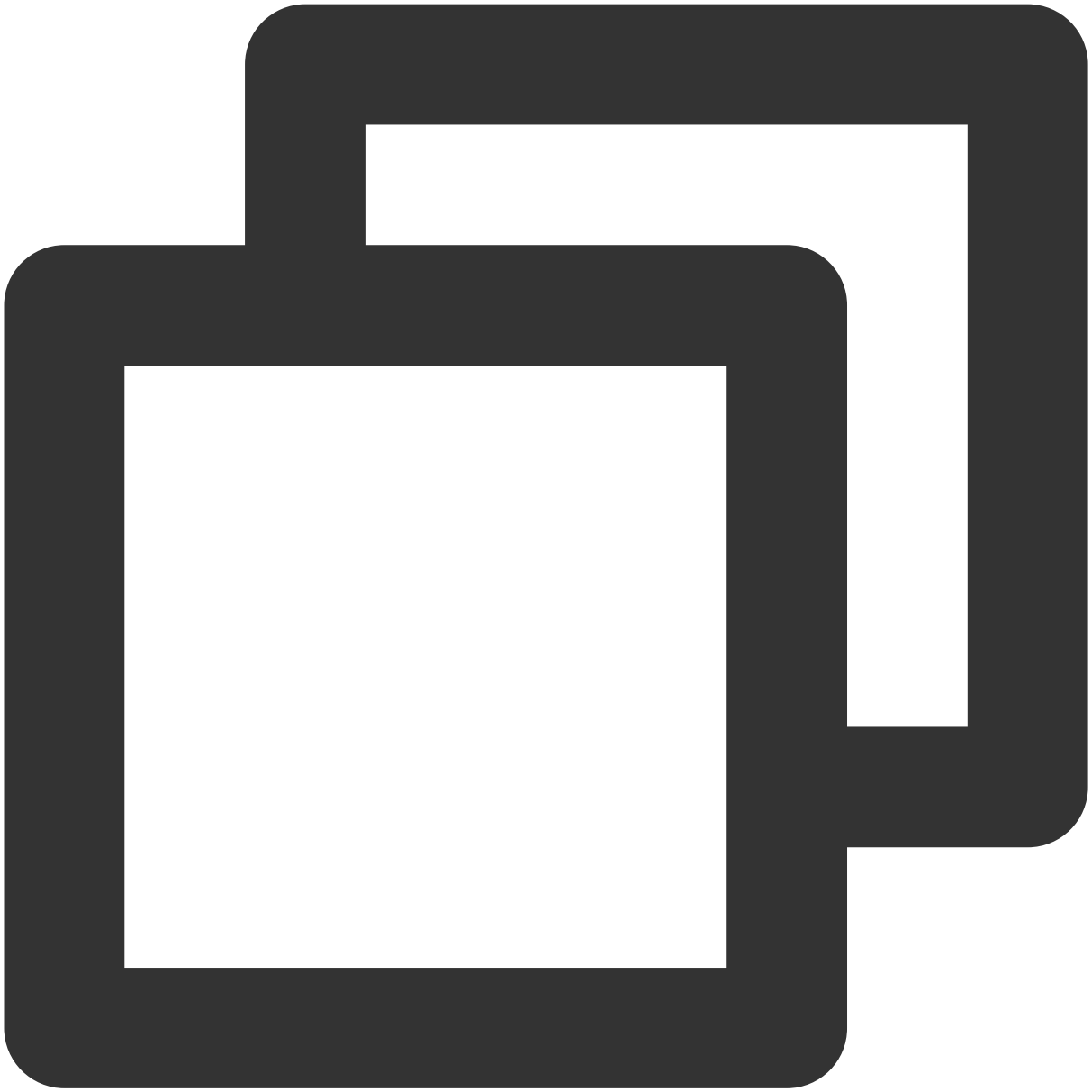
```
TRANSFORM(<expr> array<T>, <func> function(T[, integer])->U)
```

Supported engine: SparkSQL

Usage instructions: Use func to transform elements in an array.

Return type: array<U>

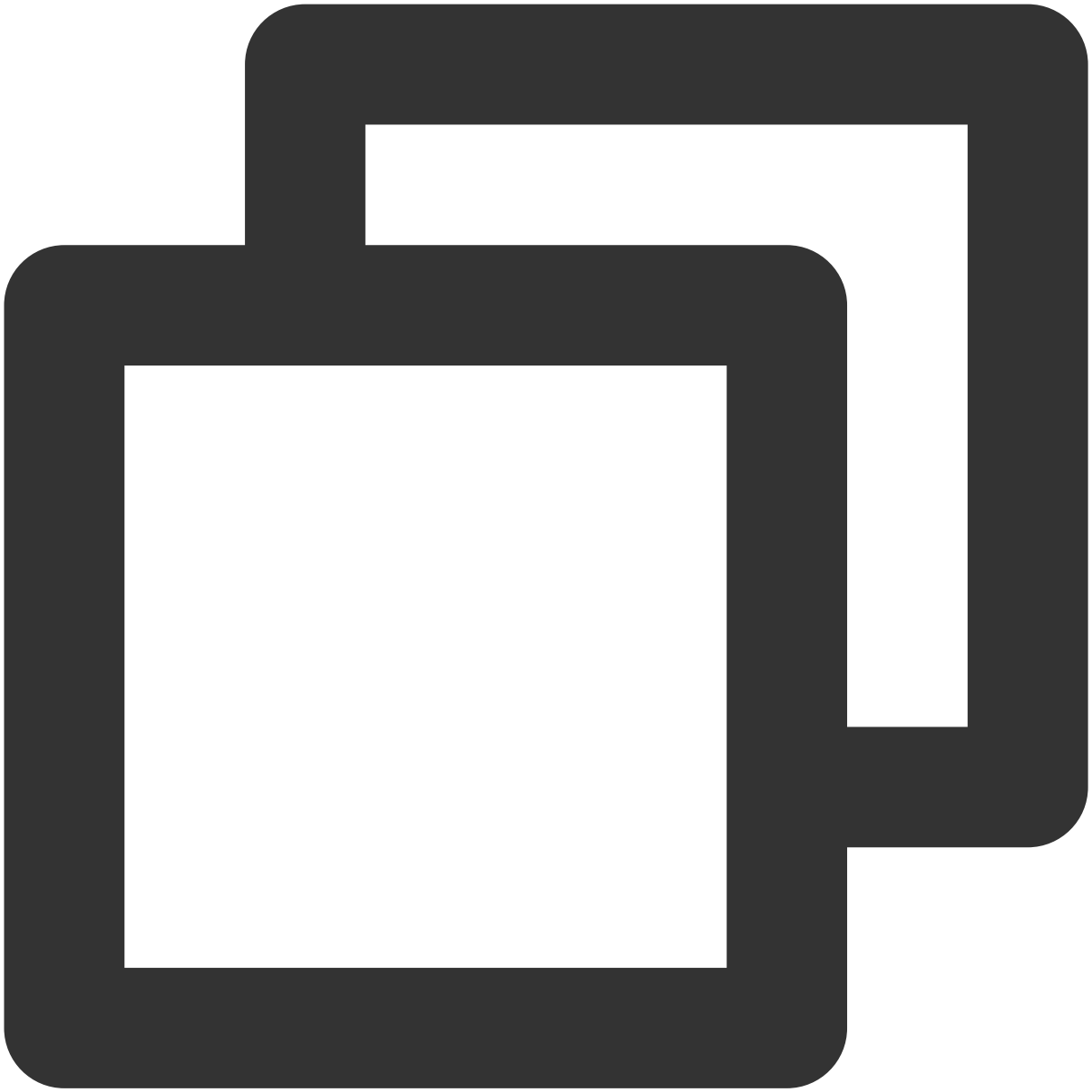
Example:



```
> SELECT transform(array(1, 2, 3), x -> x + 1);
 [2,3,4]
> SELECT transform(array(1, 2, 3), (x, i) -> x + i);
 [1,3,5]
```

ZIP_WITH

Function statement:



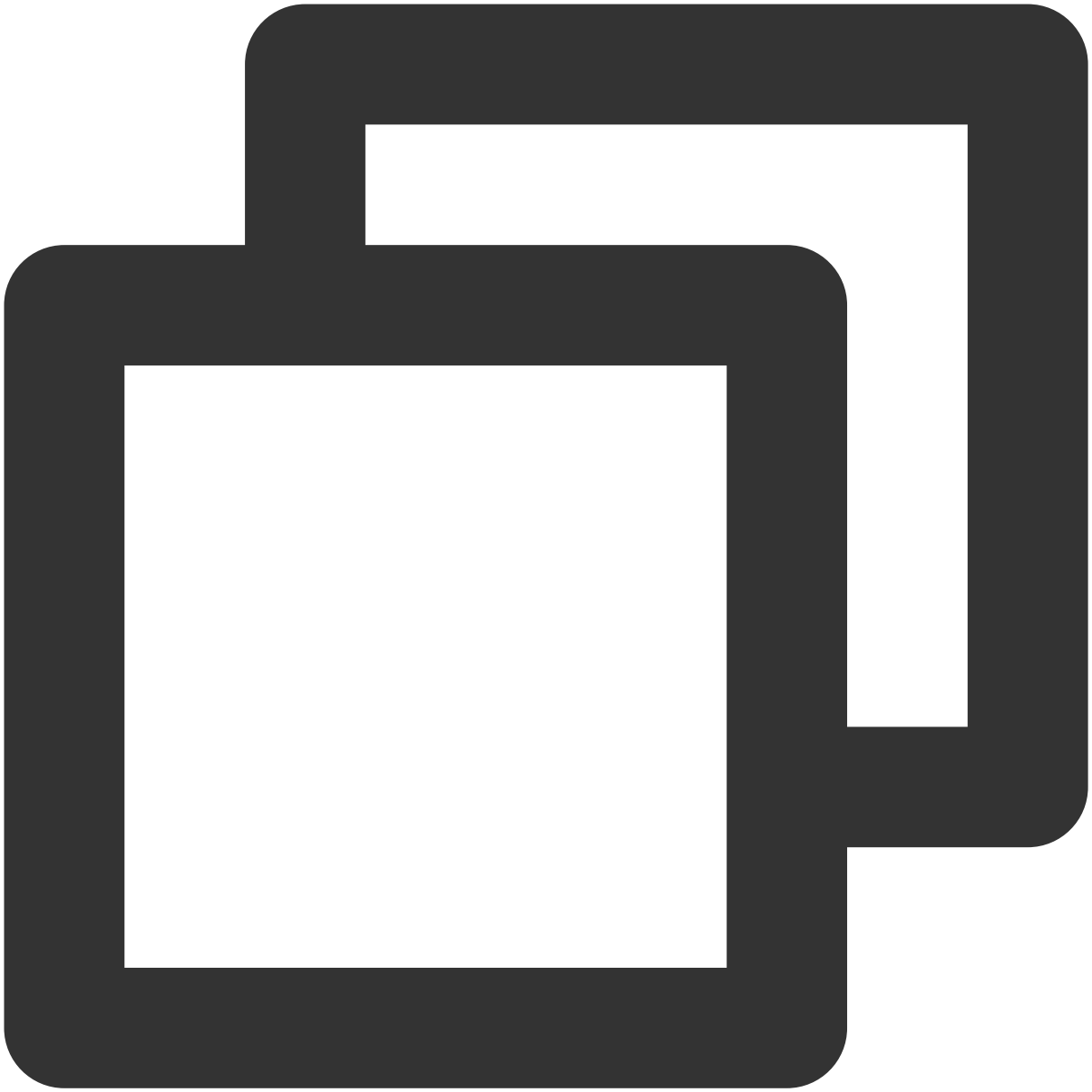
```
ZIP_WITH(<left> array<T>, <right> array<U>, <func> function(T, U)->R)
```

Supported engine: SparkSQL

Usage instructions: Use a function to merge two given arrays element-wise into a single array. If one array is shorter, nulls are appended at the end to match the length of the longer array before the function is applied.

Return type: array<R>

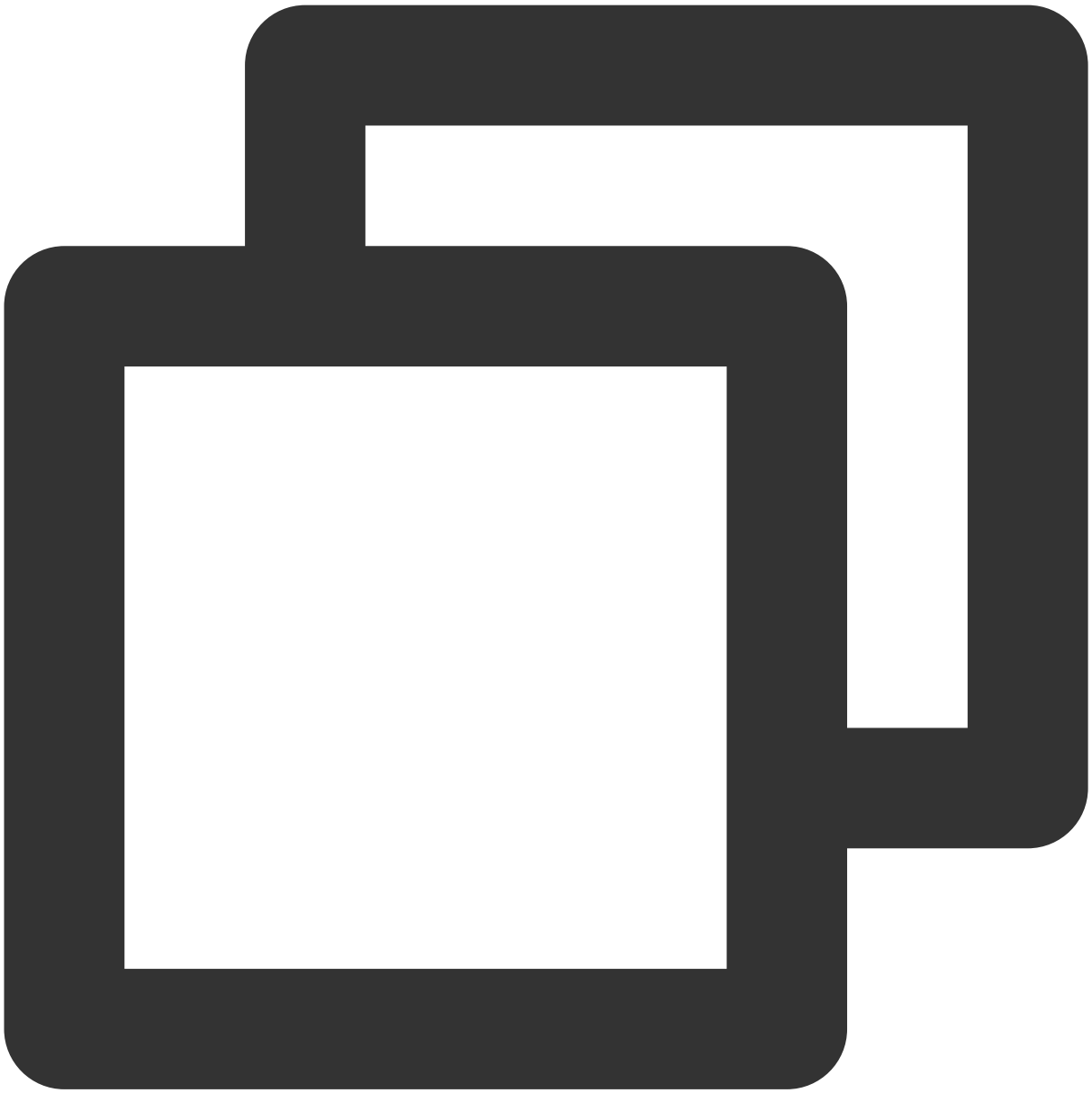
Example:



```
> SELECT zip_with(array(1, 2, 3), array('a', 'b', 'c'), (x, y) -> (y, x));
 [{"y":"a","x":1},{"y":"b","x":2},{"y":"c","x":3}]
> SELECT zip_with(array(1, 2), array(3, 4), (x, y) -> x + y);
 [4,6]
> SELECT zip_with(array('a', 'b', 'c'), array('d', 'e', 'f'), (x, y) -> concat(x, y
 ["ad","be","cf"]
```

FORALL

Function statement:



```
FORALL(<expr> array<T>, <pred> function(T)->boolean)
```

Supported engine: SparkSQL

Usage instructions: Test whether a predicate applies to all elements in the array.

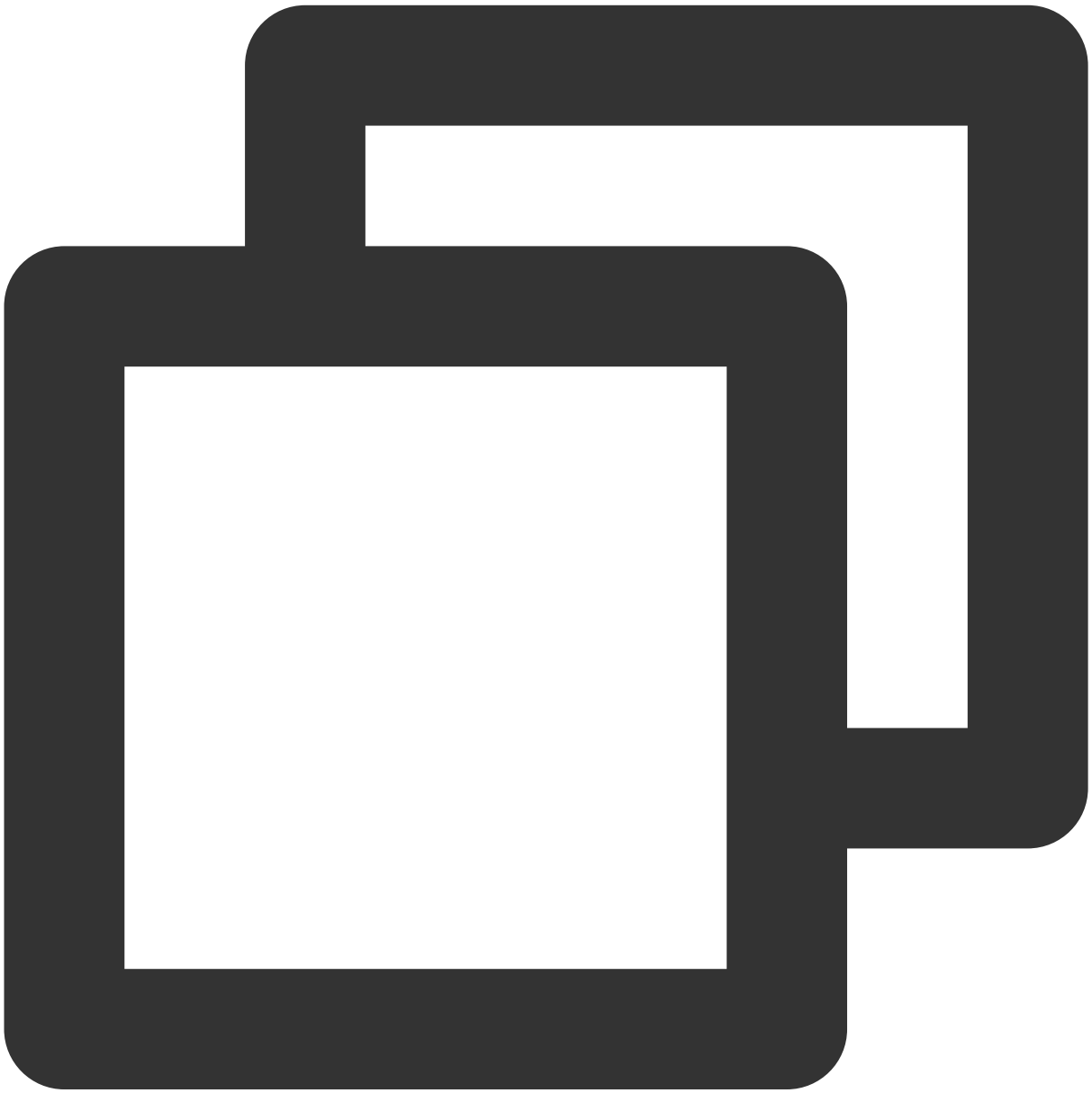
Return type: boolean

Example:

```
> SELECT forall(array(1, 2, 3), x -> x % 2 == 0);
 false
> SELECT forall(array(2, 4, 8), x -> x % 2 == 0);
 true
> SELECT forall(array(1, null, 3), x -> x % 2 == 0);
 false
> SELECT forall(array(2, null, 8), x -> x % 2 == 0);
 NULL
```

AGGREGATE

Function statement:



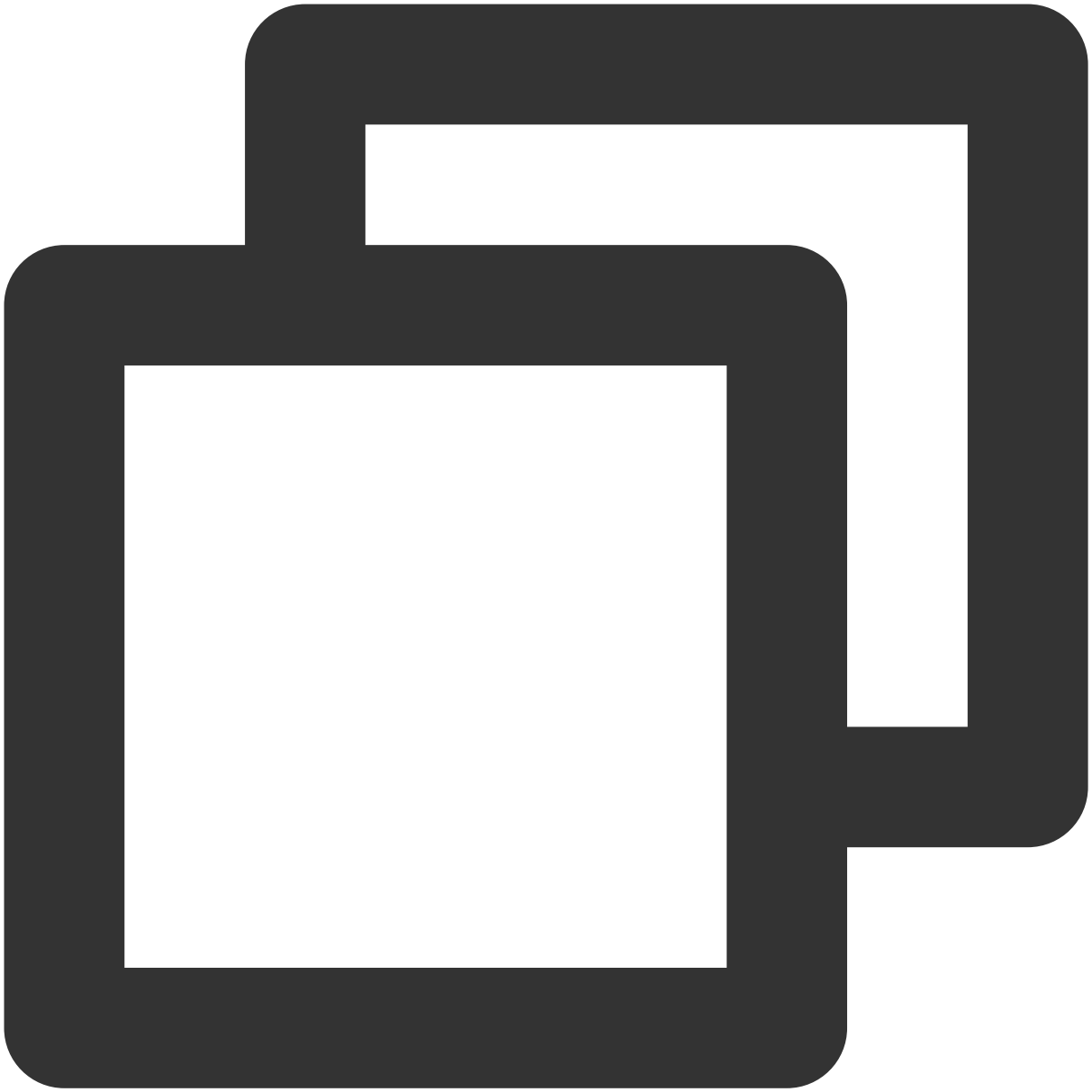
```
AGGREGATE(<expr> array<T>, <start> U, <merge> function(U, T)->U, <finish> function(
```

Supported engine: SparkSQL

Usage instructions: Aggregate the elements in the array expr.

Return type: R

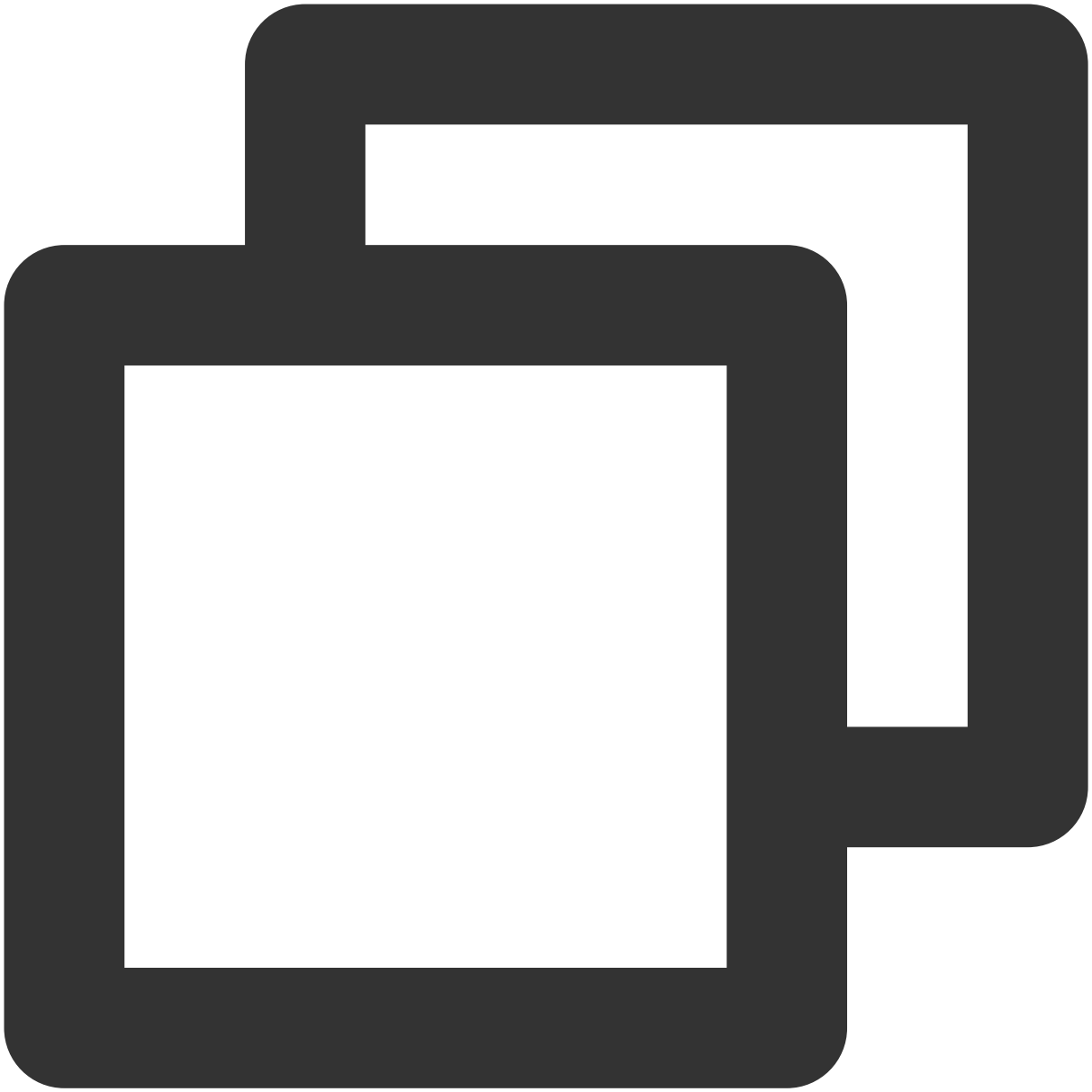
Example:



```
> SELECT aggregate(array(1, 2, 3), 0, (acc, x) -> acc + x);
 6
> SELECT aggregate(array(1, 2, 3), 0, (acc, x) -> acc + x, acc -> acc * 10);
 60
```

EXISTS

Function statement:



```
EXISTS(<expr> array<T>, <pred> function(T)->boolean)
```

Supported engine: SparkSQL

Usage instructions: Test whether a predicate applies to one or more elements in the array.

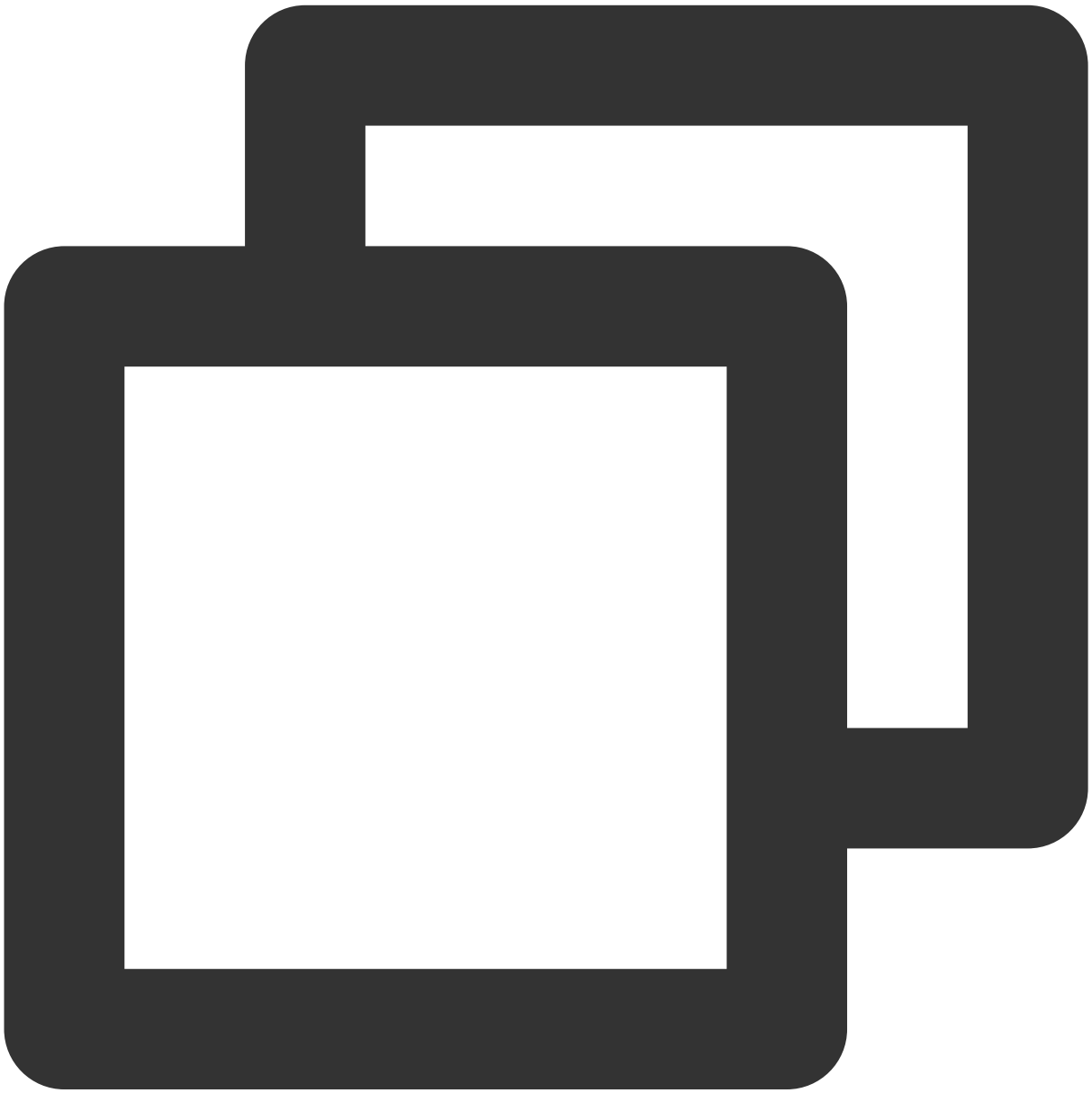
Return type: boolean

Example:

```
> SELECT exists(array(1, 2, 3), x -> x % 2 == 0);
 true> SELECT exists(array(1, 2, 3), x -> x % 2 == 10);
 false
> SELECT exists(array(1, null, 3), x -> x % 2 == 0);
 NULL
> SELECT exists(array(0, null, 2, 3, null), x -> x IS NULL);
 true
> SELECT exists(array(1, 2, 3), x -> x IS NULL);
 false
```

ARRAY_CONTAINS

Function statement:



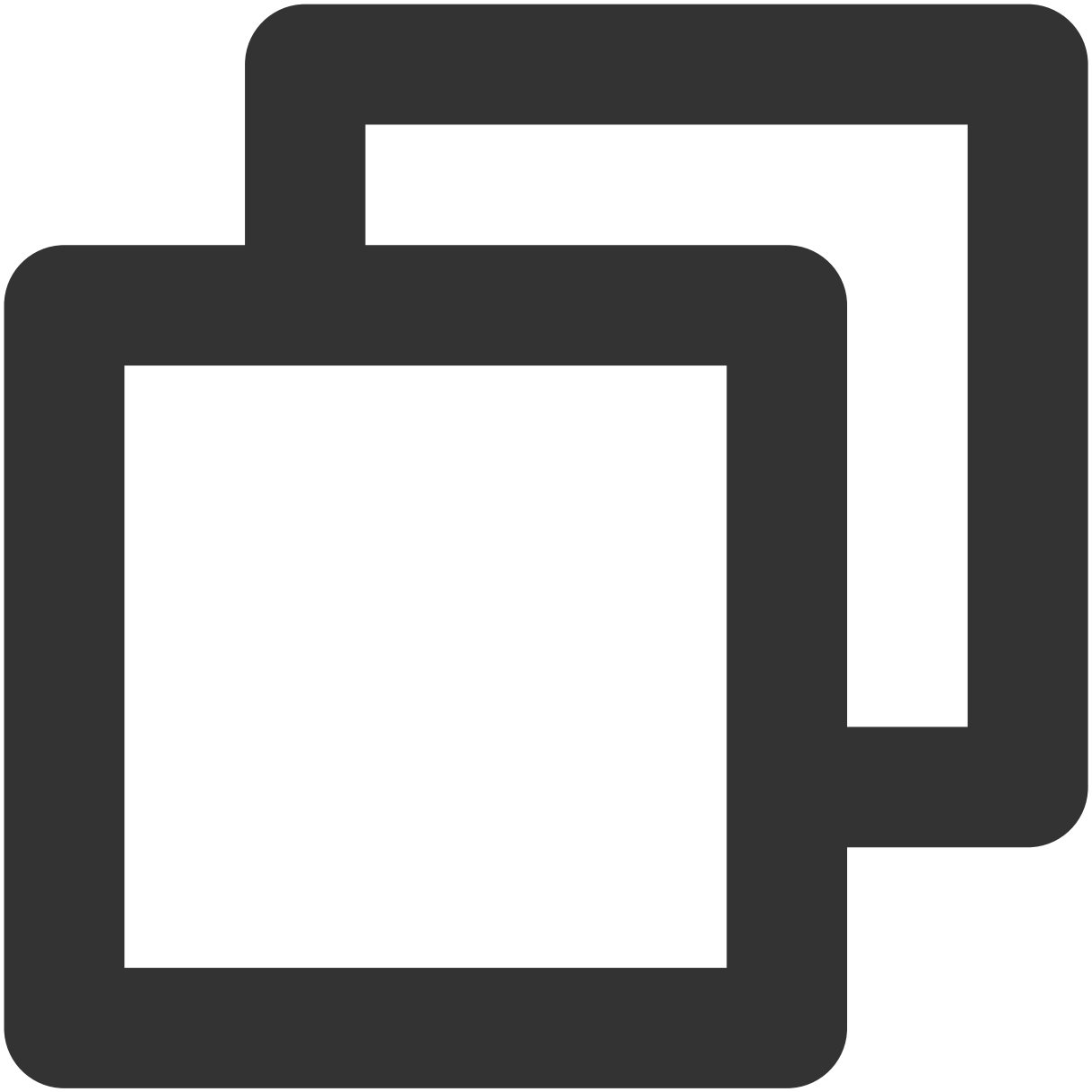
```
ARRAY_CONTAINS(<expr> array<T>, <value> T)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return true if the array contains value.

Return type: boolean

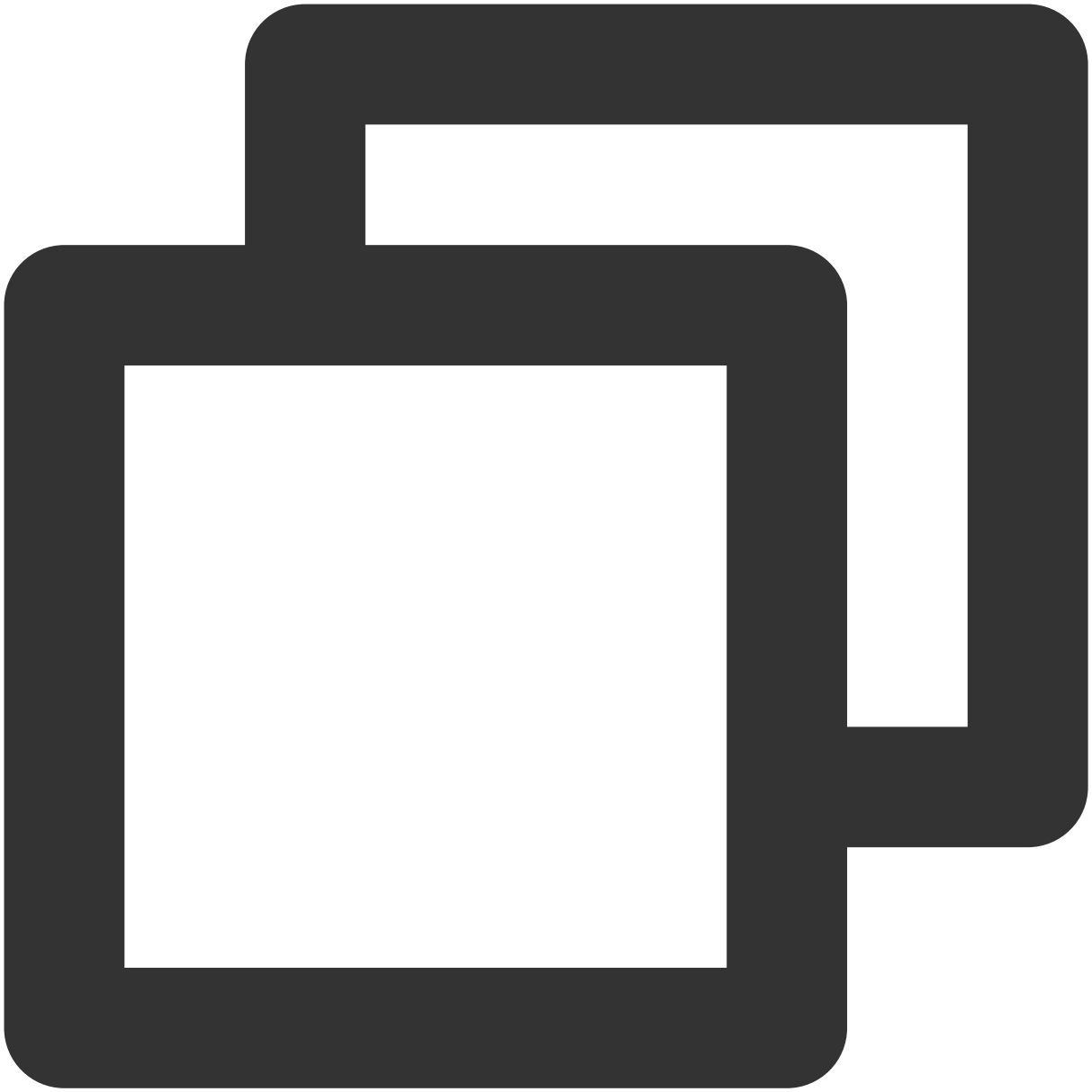
Example:



```
> SELECT array_contains(array(1, 2, 3), 2);
 true
```

ARRAYS_OVERLAP

Function statement:



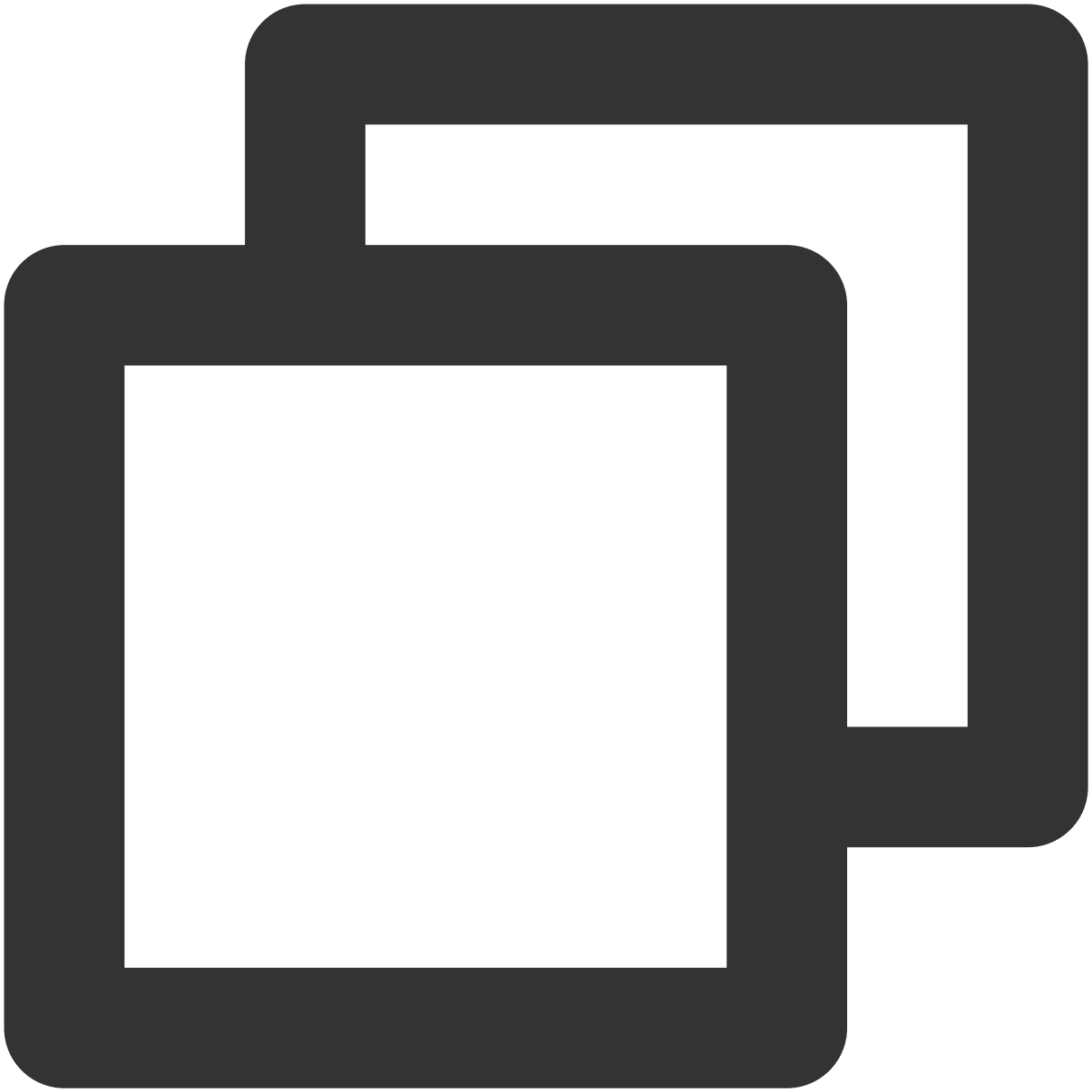
```
ARRAYS_OVERLAP(<a> array<T>, <b> array<U>)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return true if a contains at least one non-null element that also exists in b. If the arrays have no common elements, and they are non-empty, and either contains null elements, it returns null. Otherwise, it returns false.

Return type: boolean

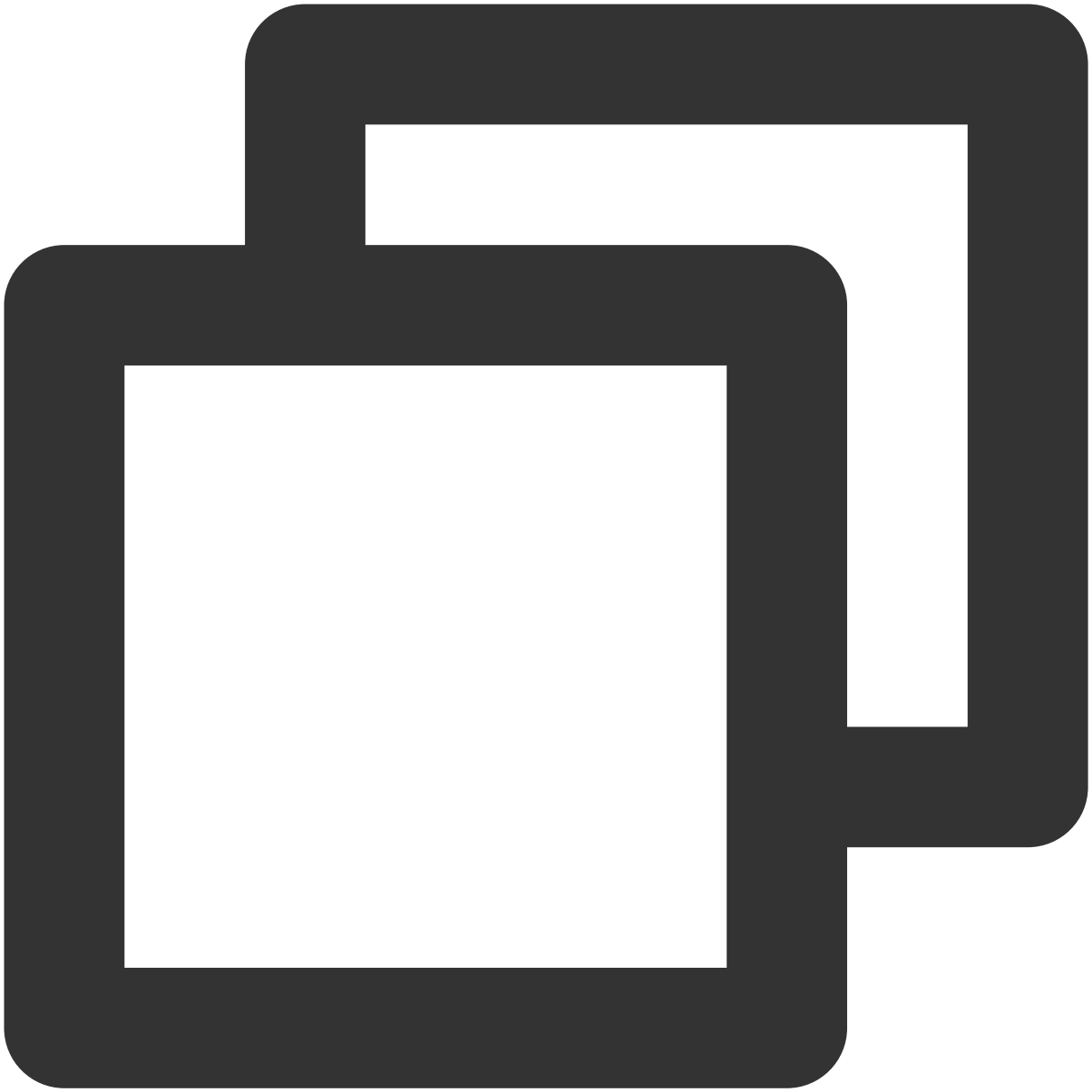
Example:



```
> SELECT arrays_overlap(array(1, 2, 3), array(3, 4, 5));
 true
```

ARRAY_INTERSECT

Function statement:



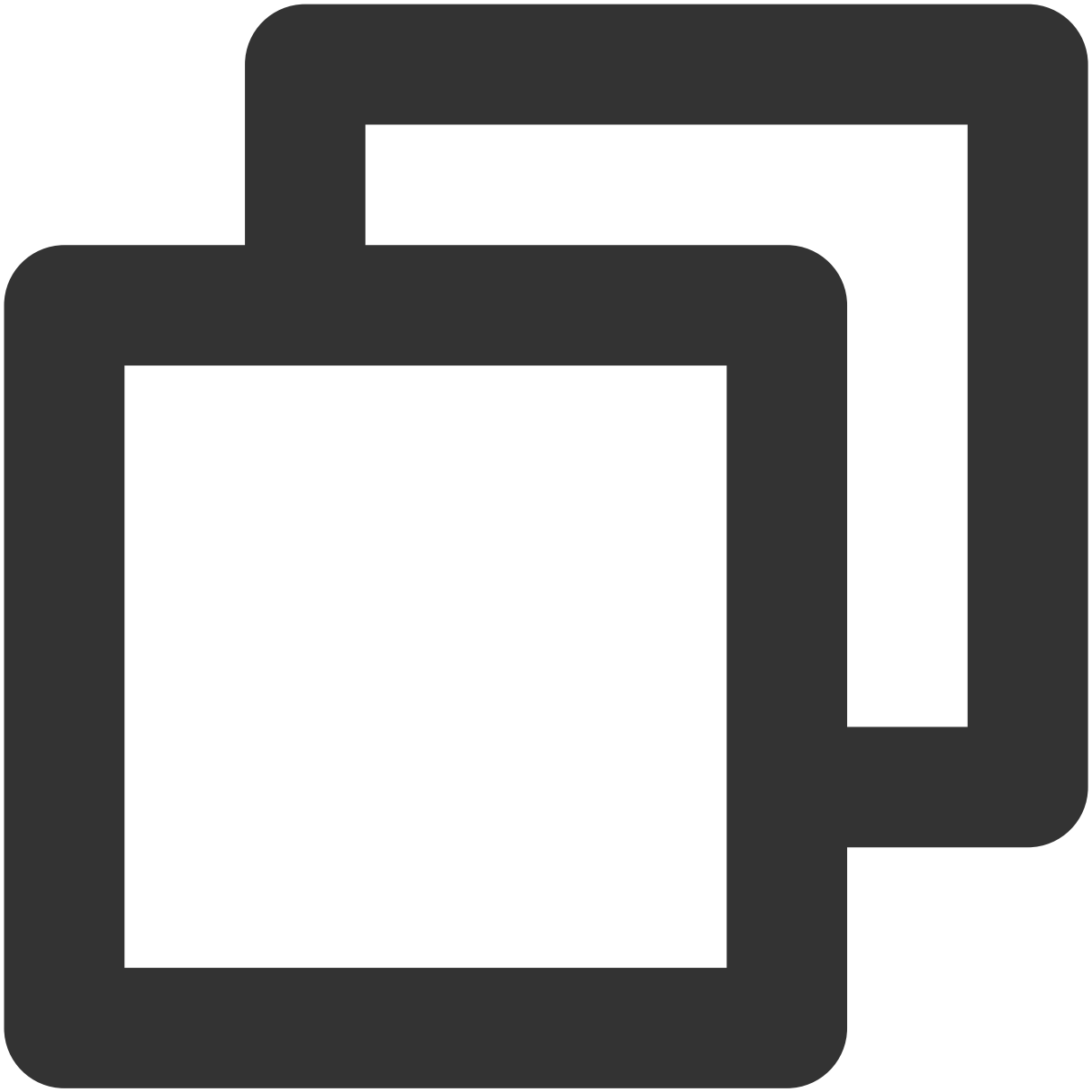
```
ARRAY_INTERSECT(<a> array<T>, <b> array<T>)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return an array of elements that are the intersection of a and b, without duplicates.

Return type: array<T>

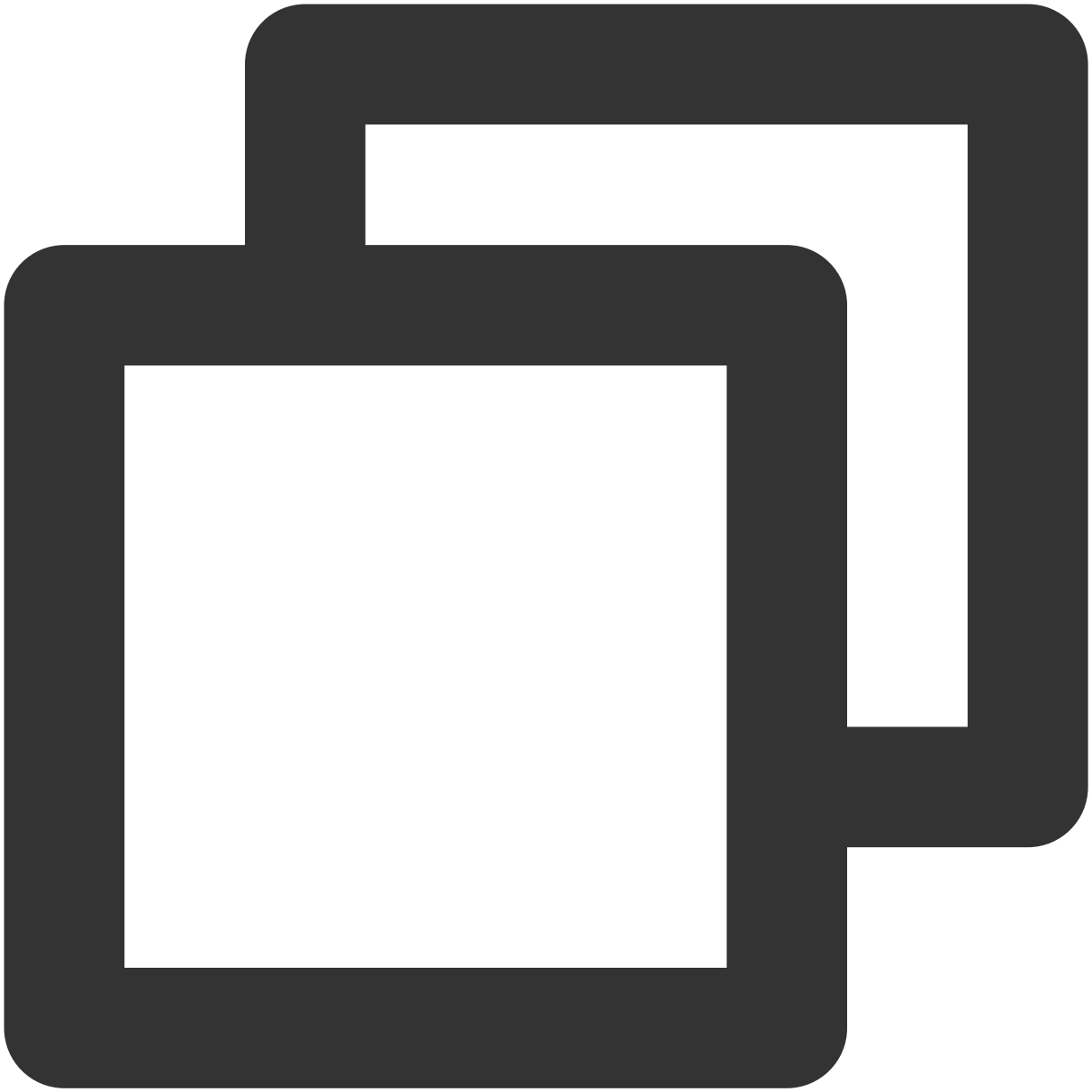
Example:



```
> SELECT array_intersect(array(1, 2, 3), array(1, 3, 5));
 [1,3]
```

ARRAY_JOIN

Function statement:



```
ARRAY_JOIN(<a> ARRAY<T>, <delimiter> string[, <nullReplacement> string])
```

Supported engines: SparkSQL and Presto

Usage instructions: Join the elements of the given array using a delimiter and an optional string to replace nulls. If nullReplacement is not set, all null values are filtered out.

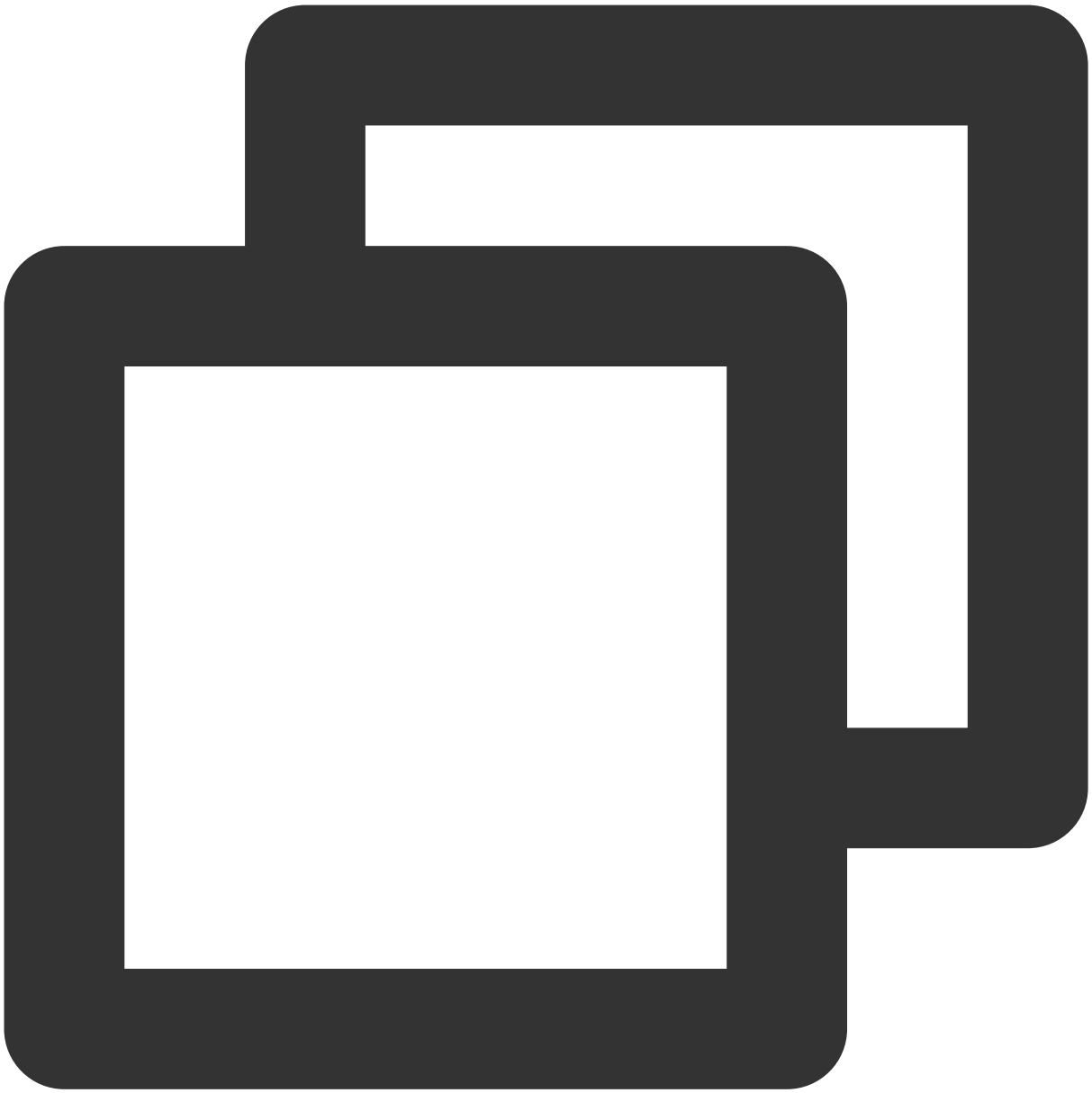
Return type: string

Example:

```
> SELECT array_join(array('hello', 'world'), ' ');
 hello world
> SELECT array_join(array('hello', null ,'world'), ' ');
 hello world
> SELECT array_join(array('hello', null ,'world'), ' ', ',');
 hello , world
```

ARRAY_POSITION

Function statement:



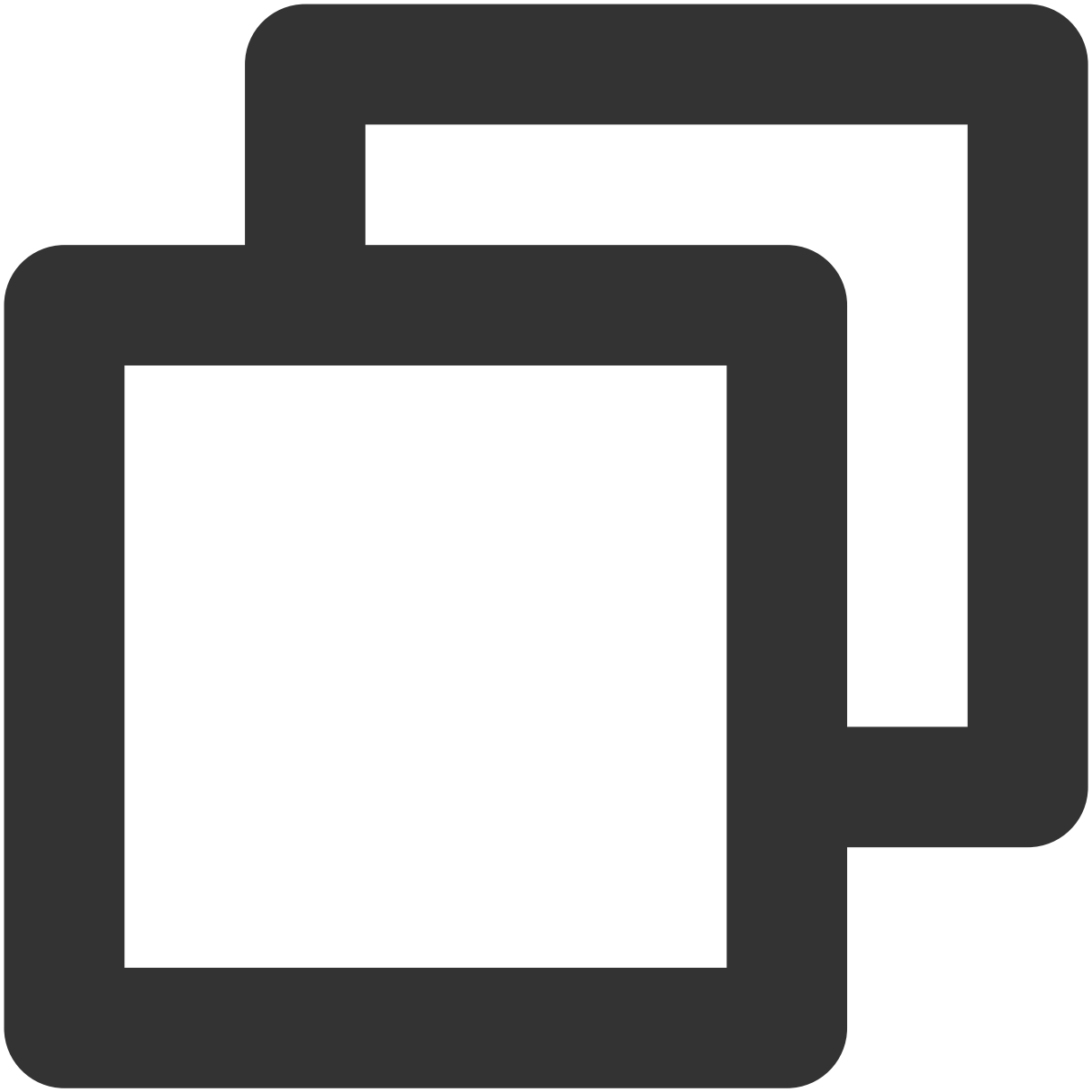
```
ARRAY_POSITION(<a> array<T>, <element> T)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the 1-based index of the first element of the array.

Return type: integer

Example:



```
> SELECT array_position(array(3, 2, 1), 1);
 3
```

ARRAY_SORT

Function statement:

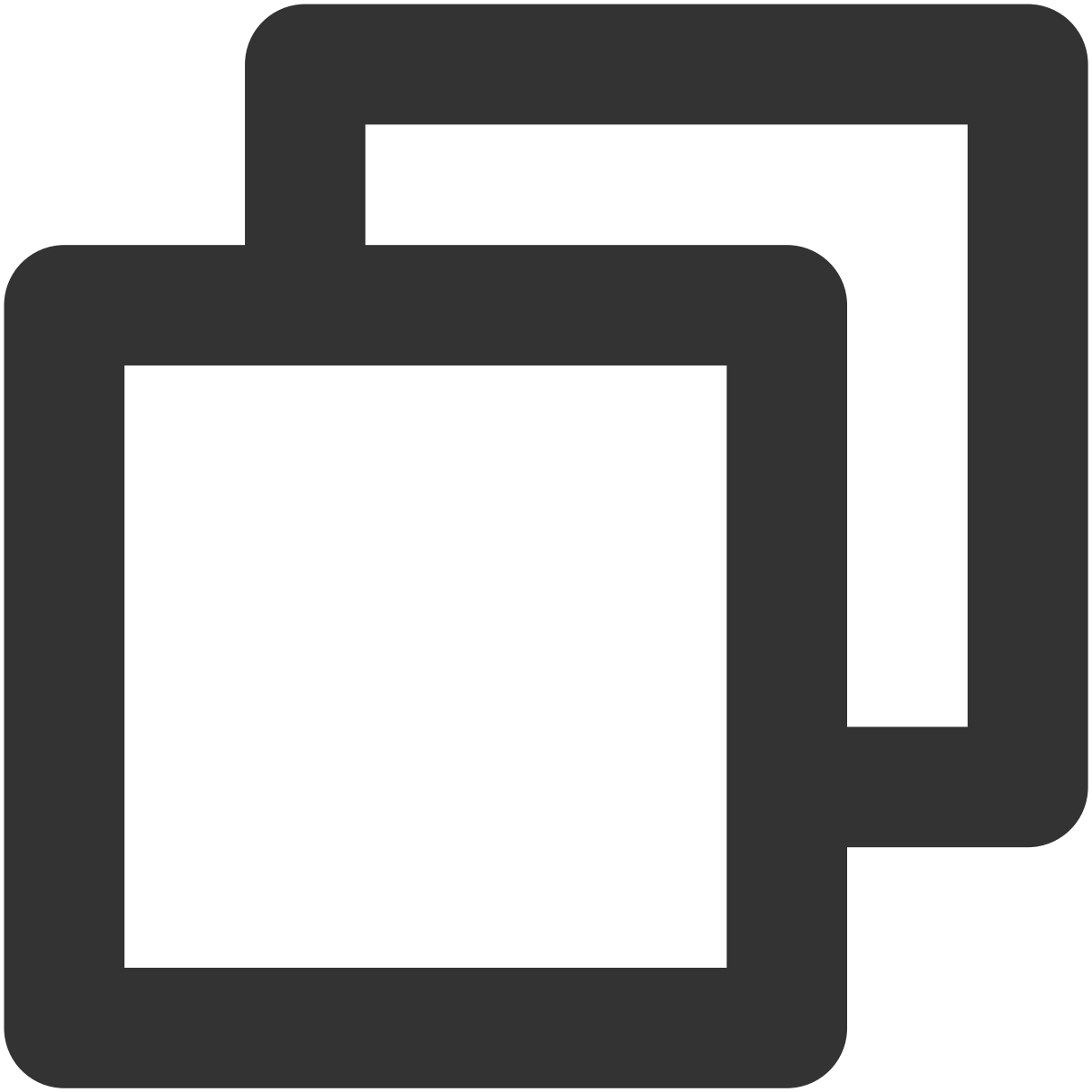
```
ARRAY_SORT(<a> array<T>[, <func> function(T, T)->integer])
```

Supported engines: SparkSQL and Presto

Usage instructions: Sort the input array. If func is omitted, the array is sorted in ascending order.

Return type: array<T>

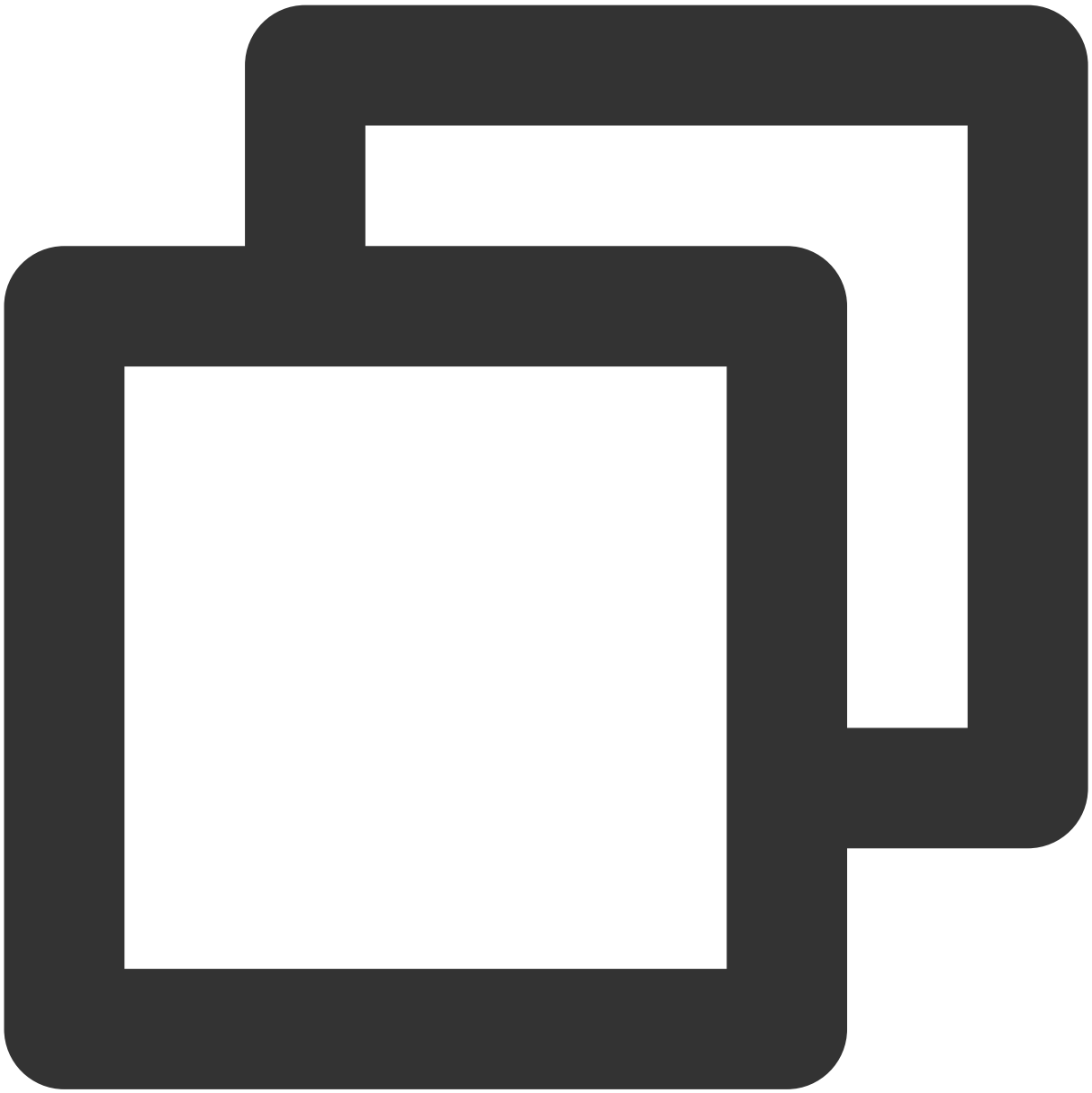
Example:



```
> SELECT array_sort(array(5, 6, 1), (left, right) -> case when left < right then -1
 [1,5,6]
> SELECT array_sort(array('bc', 'ab', 'dc'), (left, right) -> case when left is nul
 ["dc","bc","ab"]
> SELECT array_sort(array('b', 'd', null, 'c', 'a'));
 ["a","b","c","d",null]
```

ARRAY_EXCEPT

Function statement:



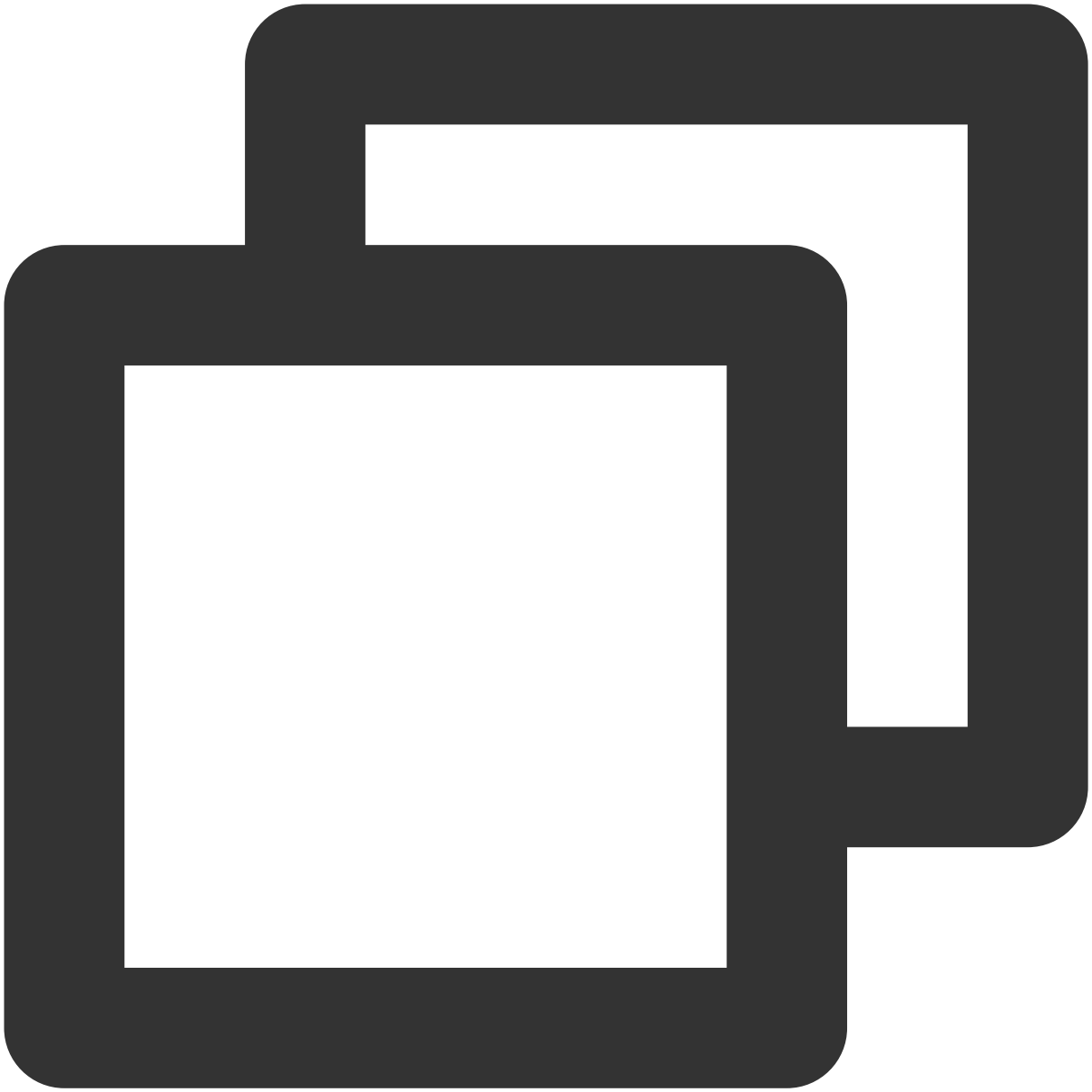
```
ARRAY_EXCEPT(<a> array<T>, <b> array<T>)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return an array of elements in a that are not in b, without duplicates.

Return type: array<T>

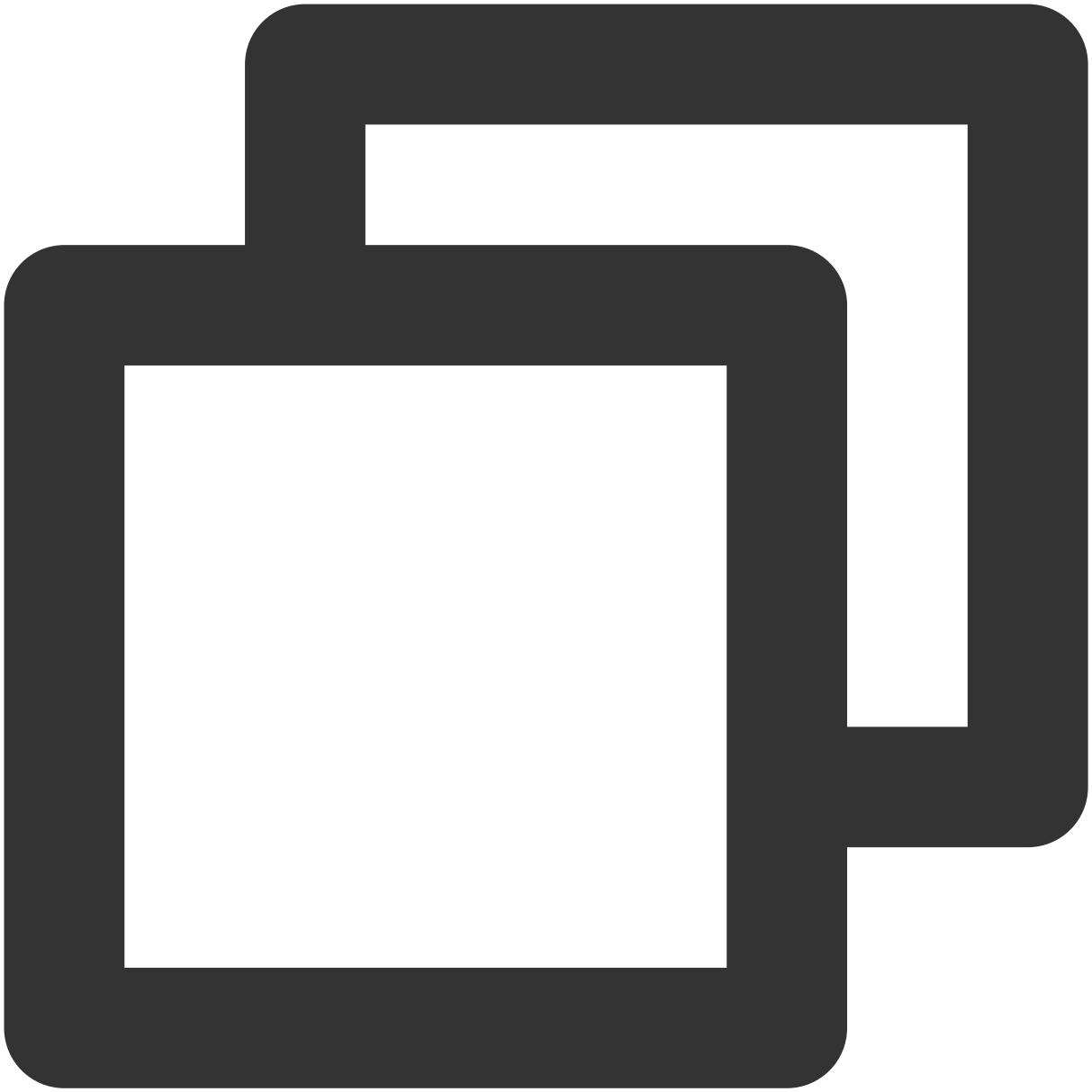
Example:



```
> SELECT array_except(array(1, 2, 3), array(1, 3, 5));
 [2]
```

ARRAY_UNION

Function statement:



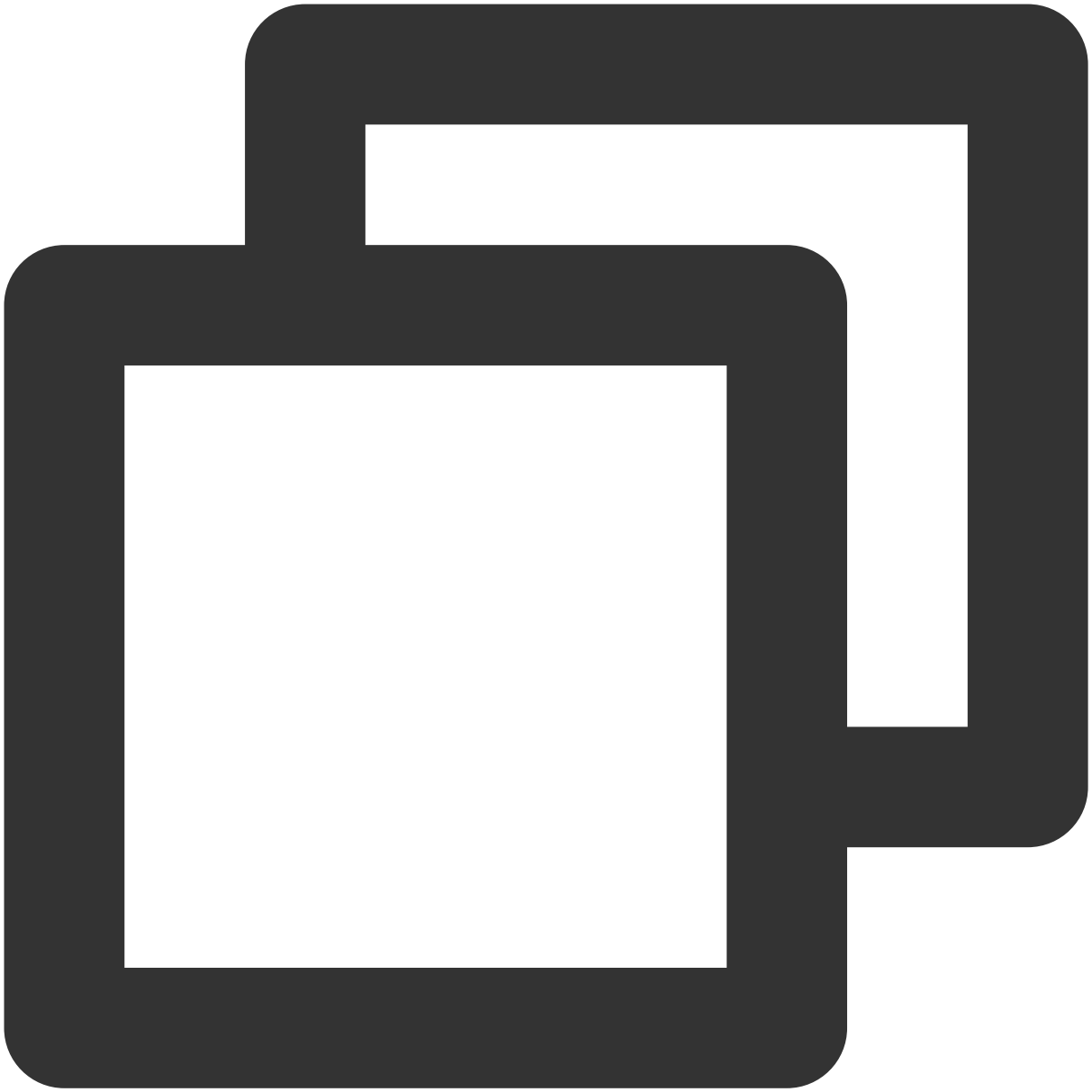
```
ARRAY_UNION(<a> array<T>, <b> array<T>)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return an array of elements that are the union of a and b, without duplicates.

Return type: array<T>

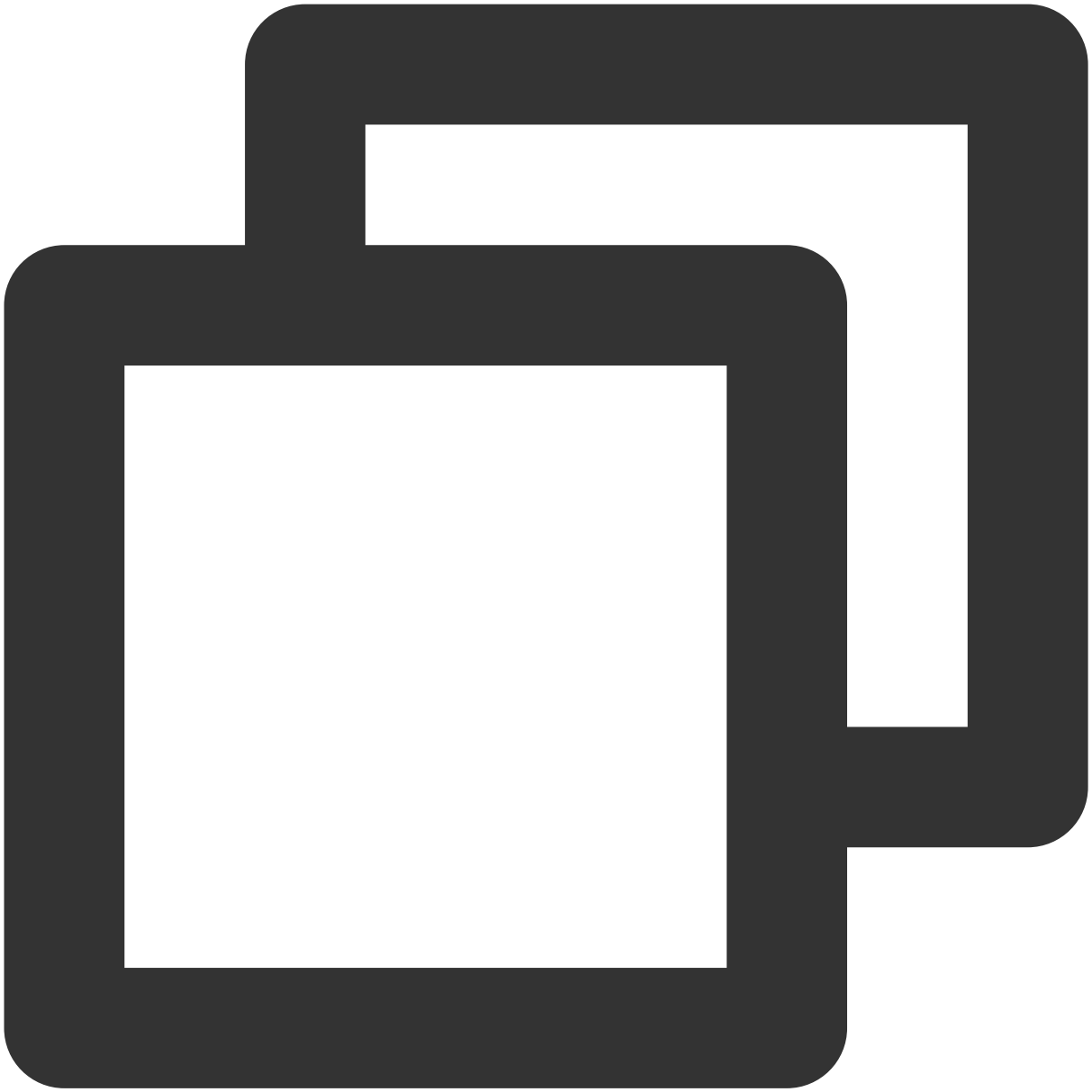
Example:



```
> SELECT array_union(array(1, 2, 3), array(1, 3, 5));
 [1,2,3,5]
```

NAMED_STRUCT

Function statement:



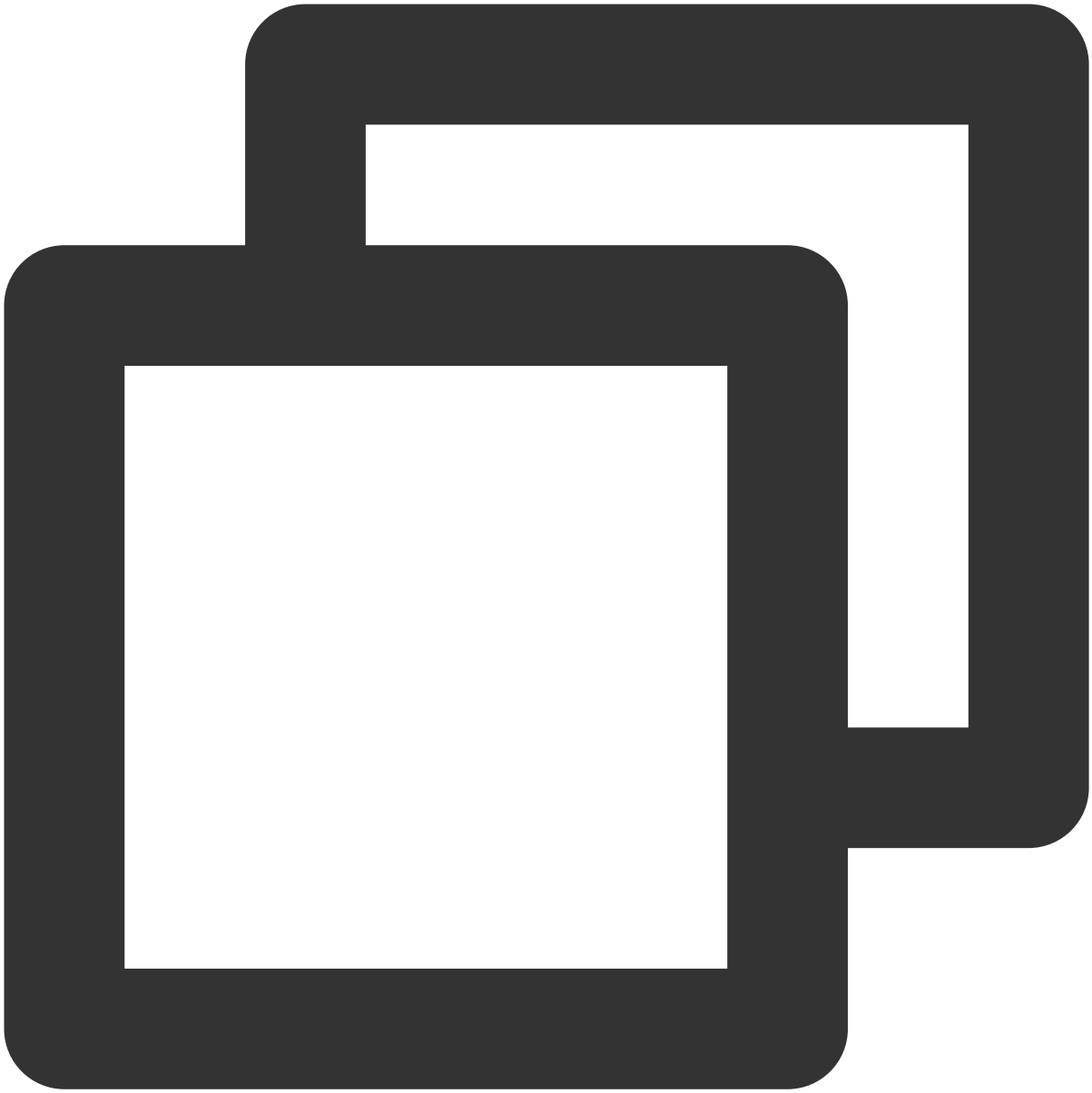
```
NAMED_STRUCT(name1 K, val1 V, ...)
```

Supported engine: SparkSQL

Usage instructions: Create a struct using the given field names and values.

Return type: struct

Example:



```
> SELECT named_struct("a", 1, "b", 2, "c", 3);
 {"a":1,"b":2,"c":3}
```

STRUCT

Function statement:

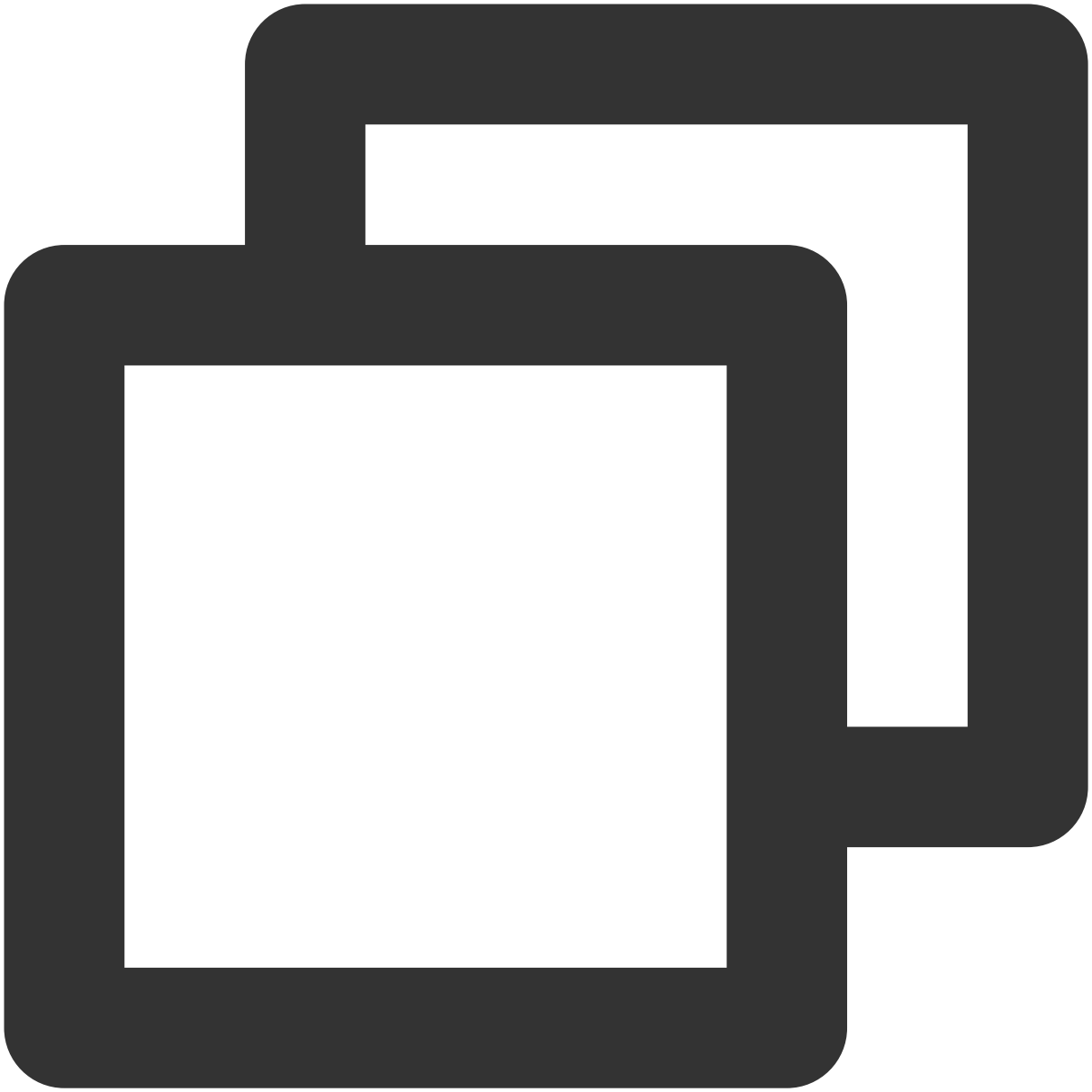
```
STRUCT(<col1> T1, <col2> T2, ...)
```

Supported engines: SparkSQL and Presto

Usage instructions: Create a struct using the given field names and values.

Return type: struct

Example:



```
> SELECT struct('a', 'b', 'c');
{"col1":"a","col2":"b","col3":"c"}

> SELECT struct('a', 'b', 'c', 1, 2);
{"col1":"a","col2":"b","col3":"c","col4":1,"col5":2}
```

SLICE

Function statement:

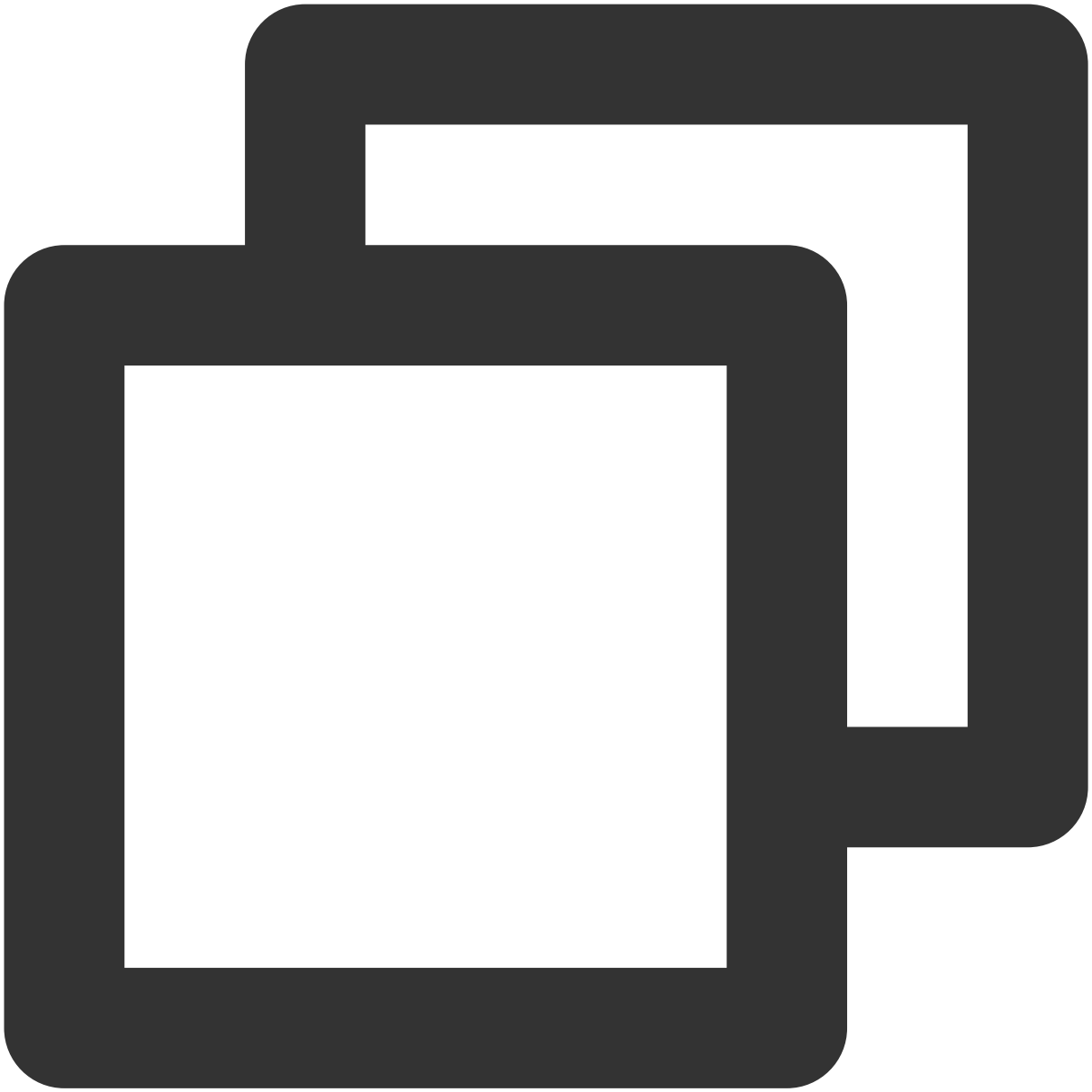
```
SLICE(<a> array<T>, <start> integer, <length> integer)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return a subset of array a starting from the index start (array hidden index starts from 1, if start is negative, it counts from the end) with the specified length length.

Return type: array<T>

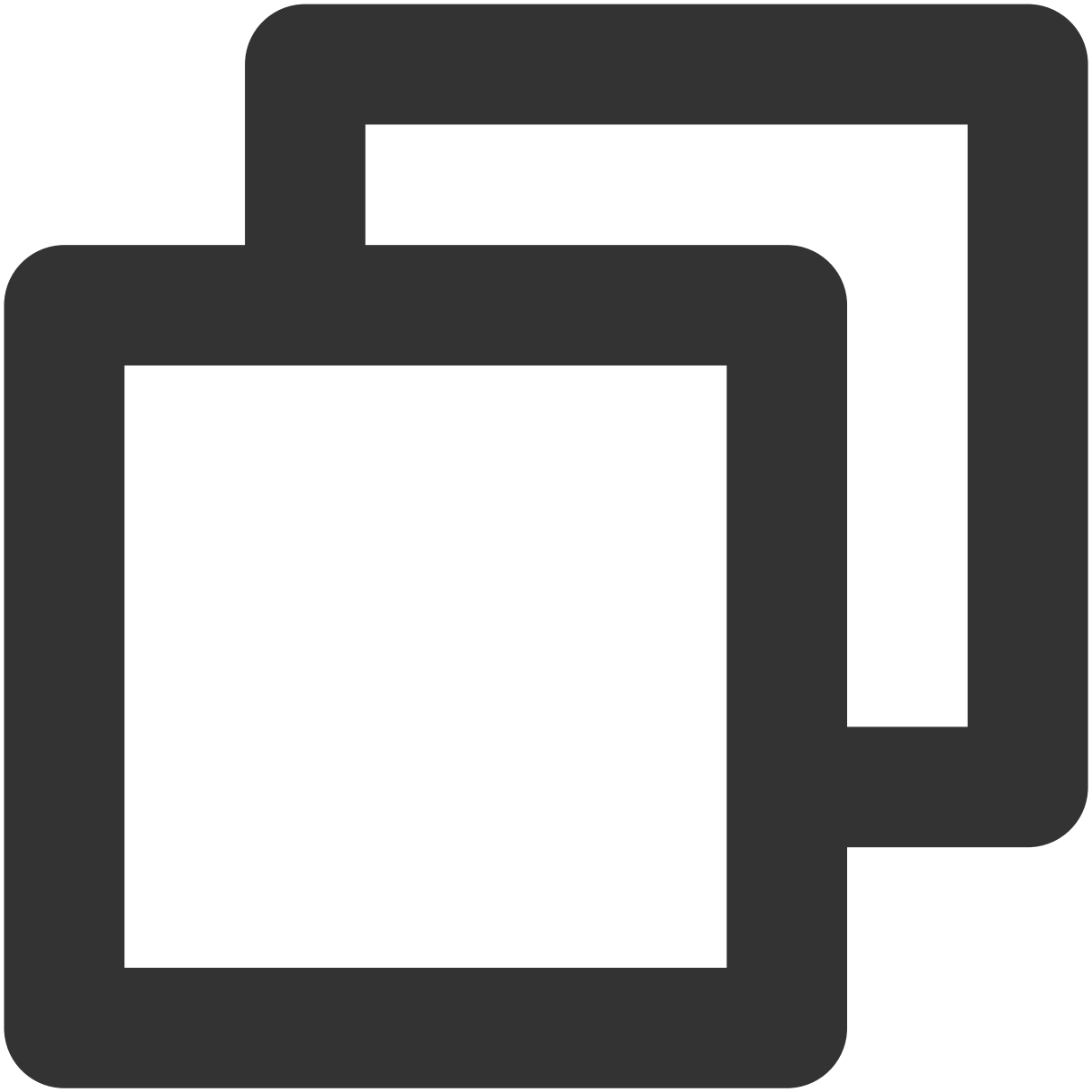
Example:



```
> SELECT slice(array(1, 2, 3, 4), 2, 2);
 [2,3]
> SELECT slice(array(1, 2, 3, 4), -2, 2);
 [3,4]
```

ARRAYS_ZIP

Function statement:



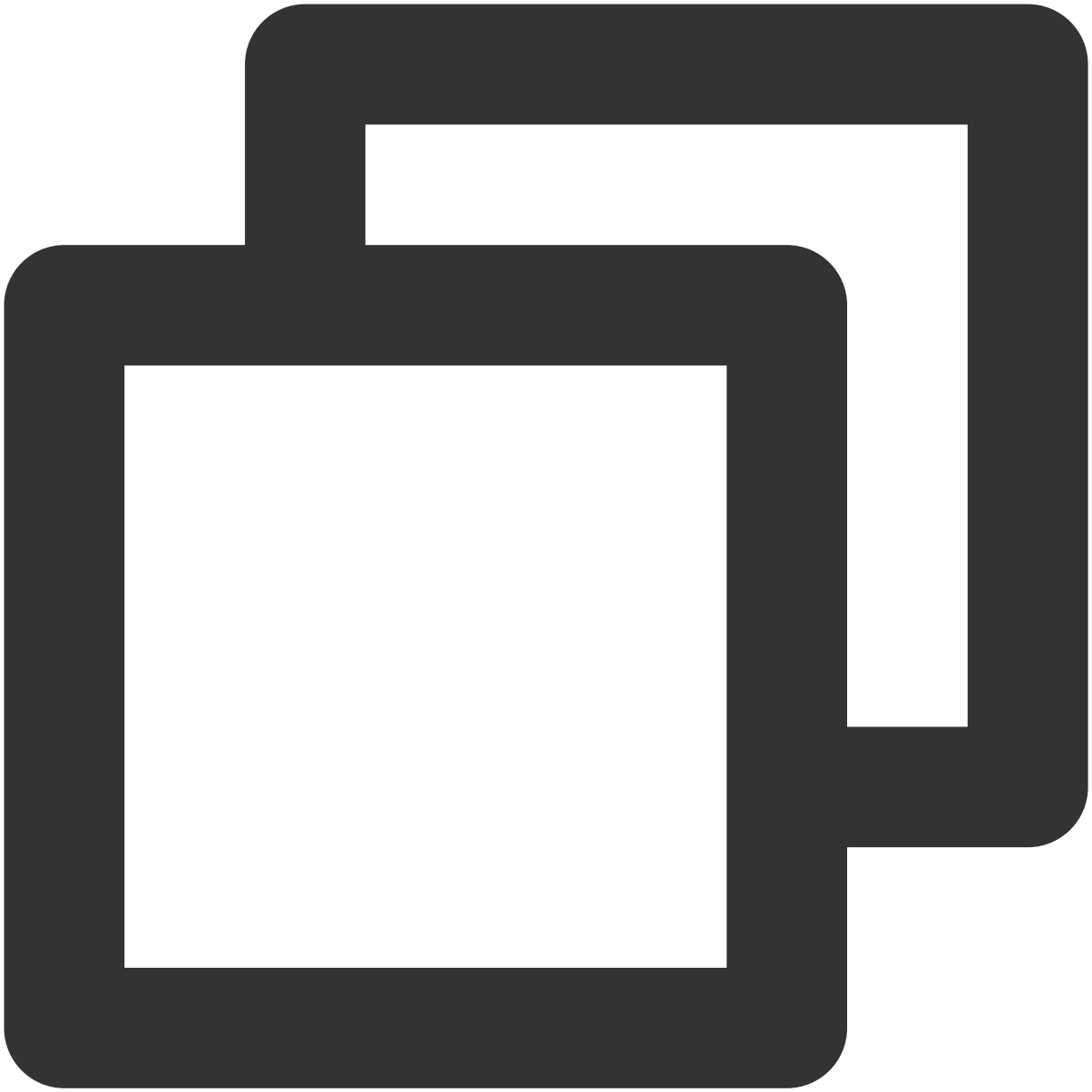
```
ARRAYS_ZIP(<a1> array<T>, ...)
```

Supported engine: SparkSQL

Usage instructions: Return a merged array where each element is of struct type, with the Nth struct containing the Nth value of each input array.

Return type: array<struct<string, T>>

Example:



```
> SELECT arrays_zip(array(1, 2, 3), array(2, 3, 4));
 [{"0":1,"1":2},{"0":2,"1":3},{"0":3,"1":4}]
> SELECT arrays_zip(array(1, 2), array(2, 3), array(3, 4));
 [{"0":1,"1":2,"2":3},{"0":2,"1":3,"2":4}]
```

SORT_ARRAY

Function statement:

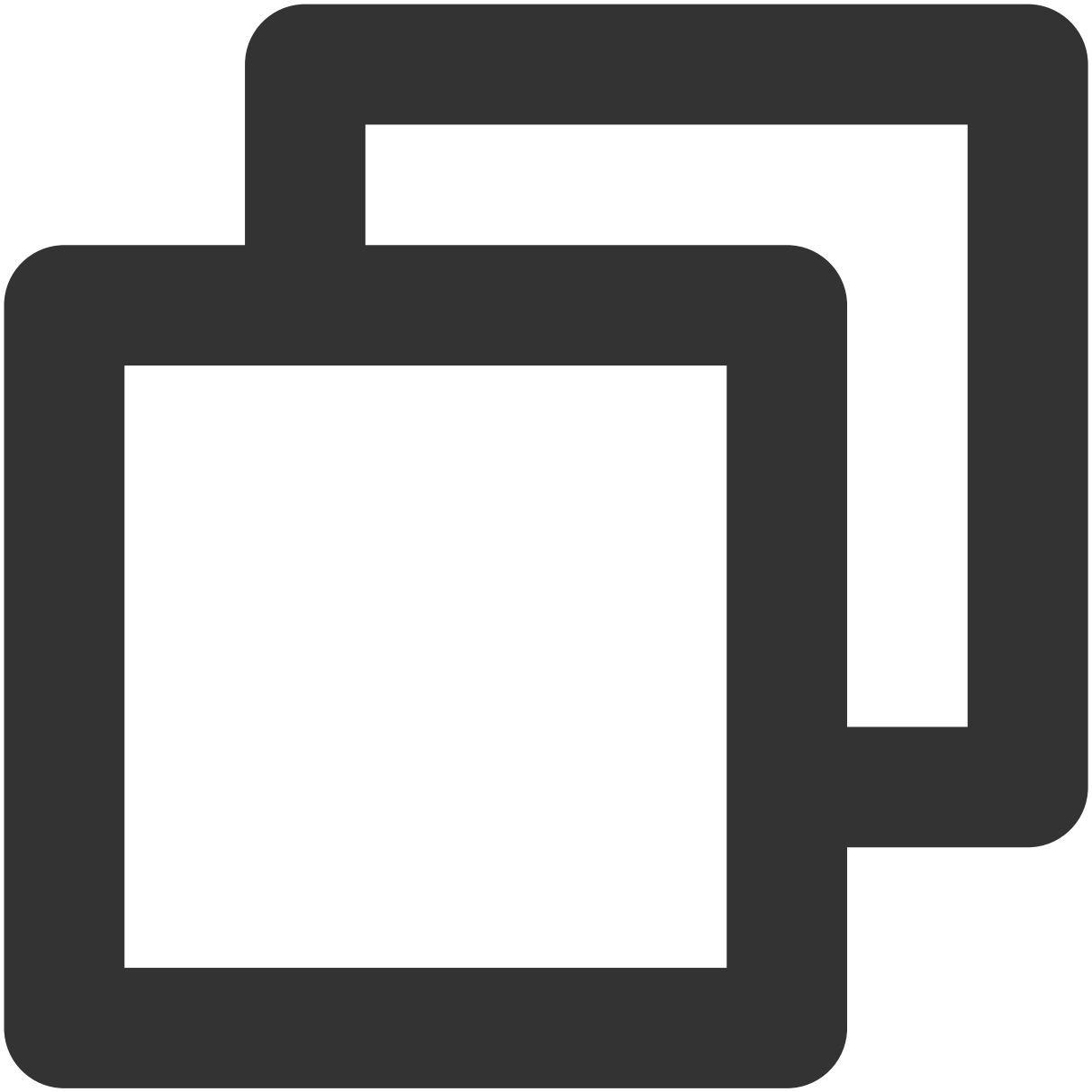
```
SORT_ARRAY(<a> array<T>[, ascendingOrder boolean])
```

Supported engines: SparkSQL and Presto

Usage instructions: Sort the input array in ascending or descending order. For double/float types, NaN is considered greater than any non-NaN element. Null elements are placed at the beginning of the returned array in ascending order or at the end in descending order.

Return type: array<T>

Example:



```
> SELECT sort_array(array('b', 'd', null, 'c', 'a'), true);
 [null,"a","b","c","d"]
```

SHUFFLE

Function statement:

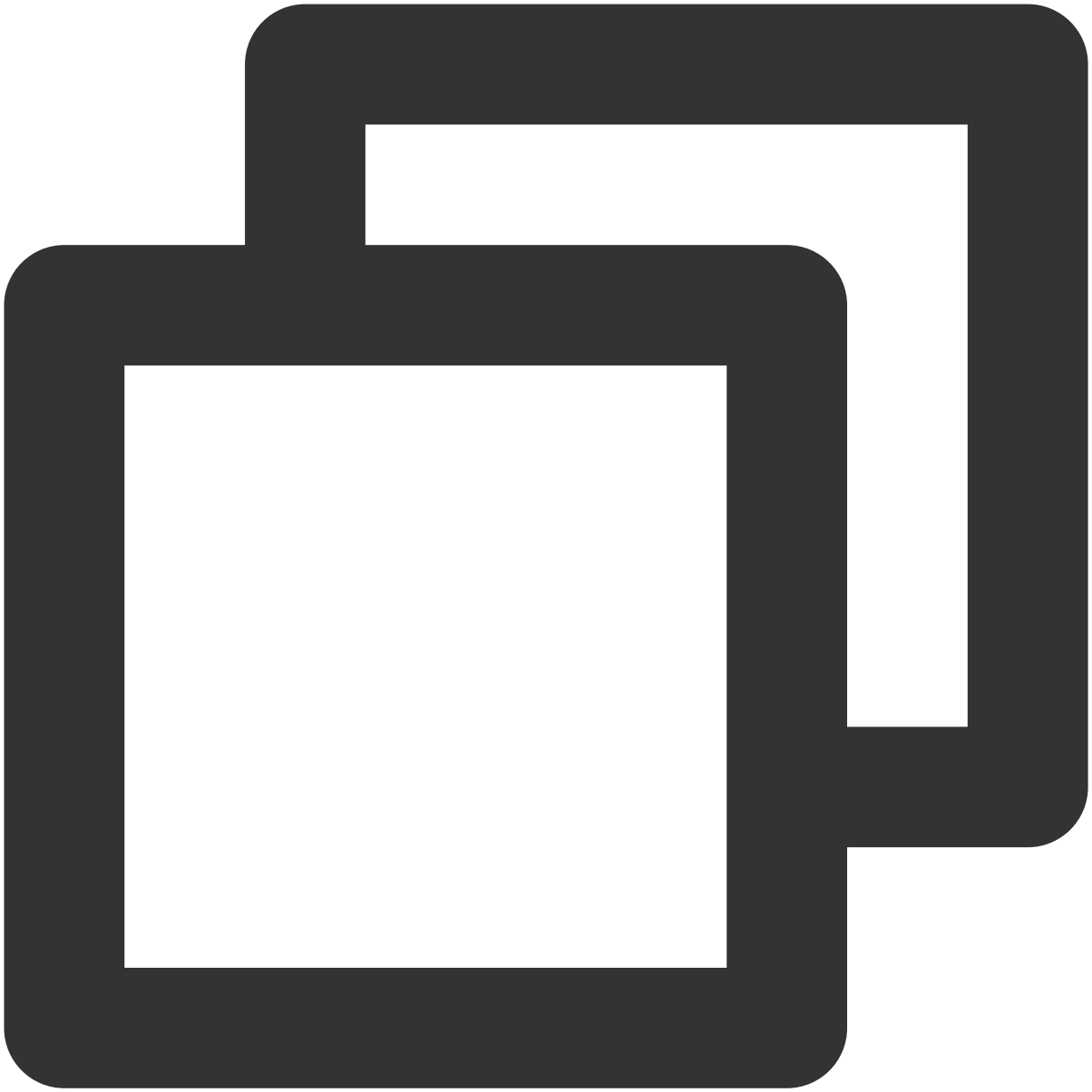
```
SHUFFLE(<a> array<T>)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return a random permutation of the given array.

Return type: array<T>

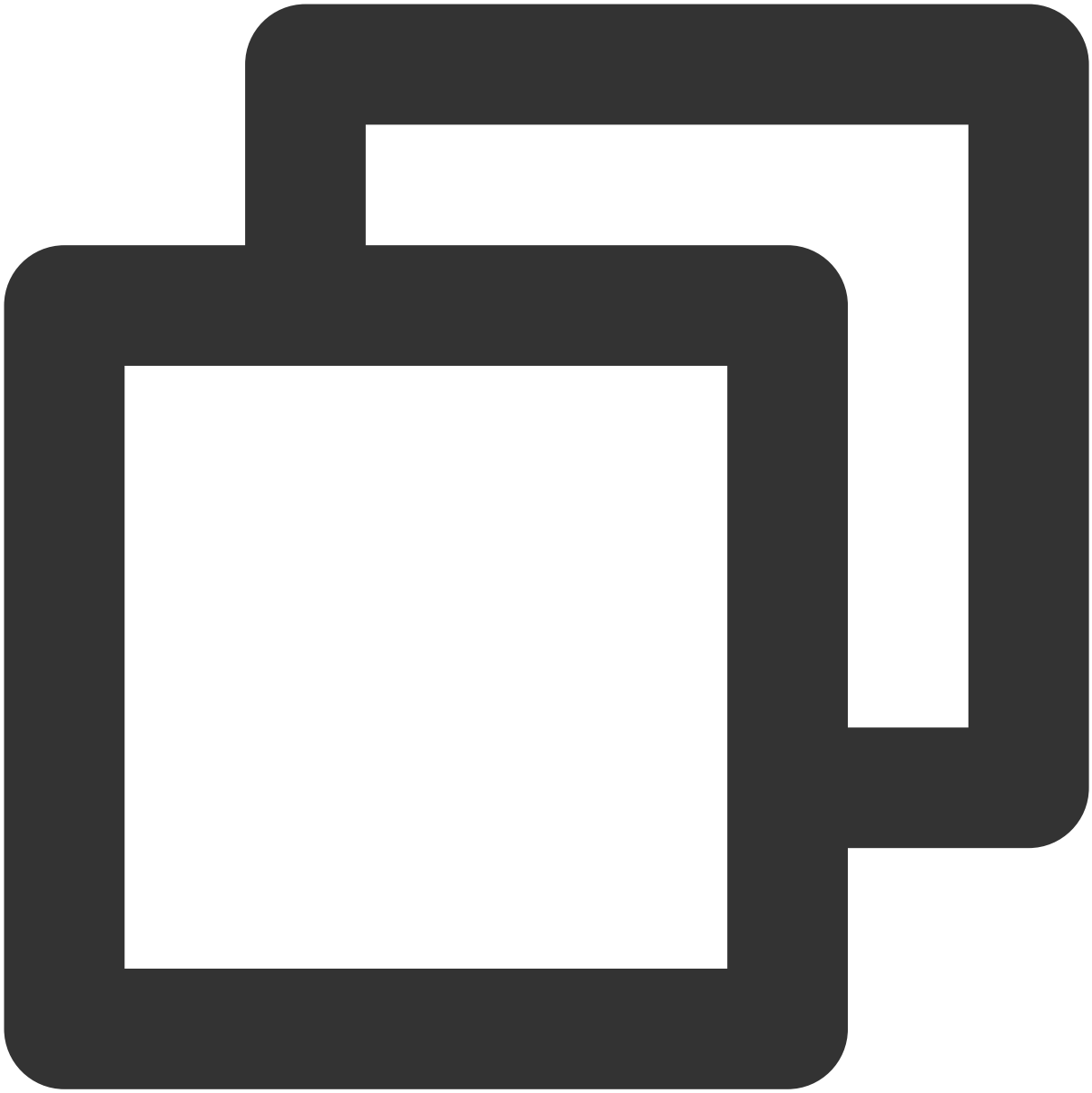
Example:



```
> SELECT shuffle(array(1, 20, 3, 5));
 [3,1,5,20]
> SELECT shuffle(array(1, 20, null, 3));
 [20,null,3,1]
```

ARRAY_MAX

Function statement:



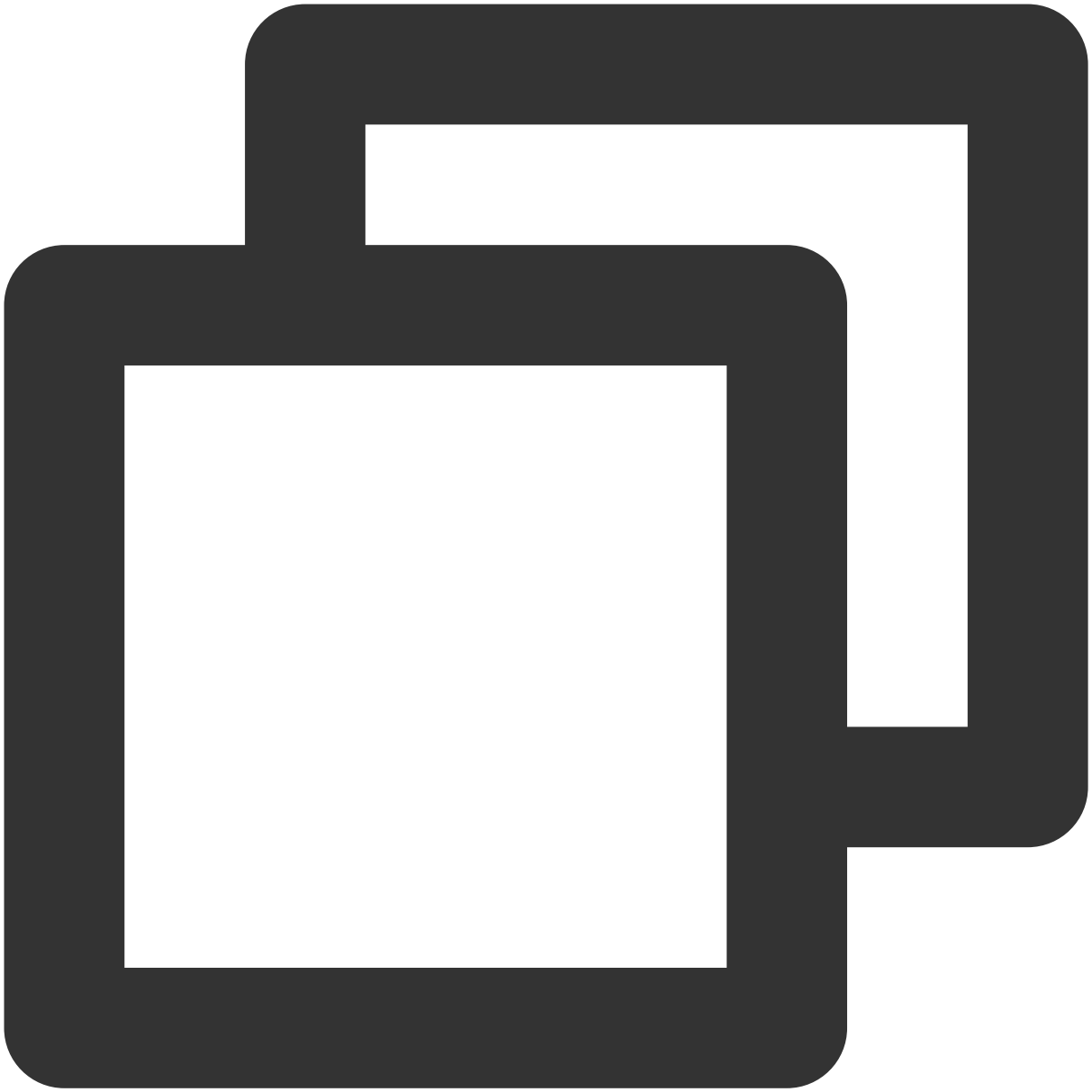
```
ARRAY_MAX(<a> array<T>)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the maximum value in the array. For double/float types, NaN is considered greater than any non-NaN element. Null elements are skipped.

Return type: array<T>

Example:



```
> SELECT array_max(array(1, 20, null, 3));
 20
```

ARRAY_MIN

Function statement:

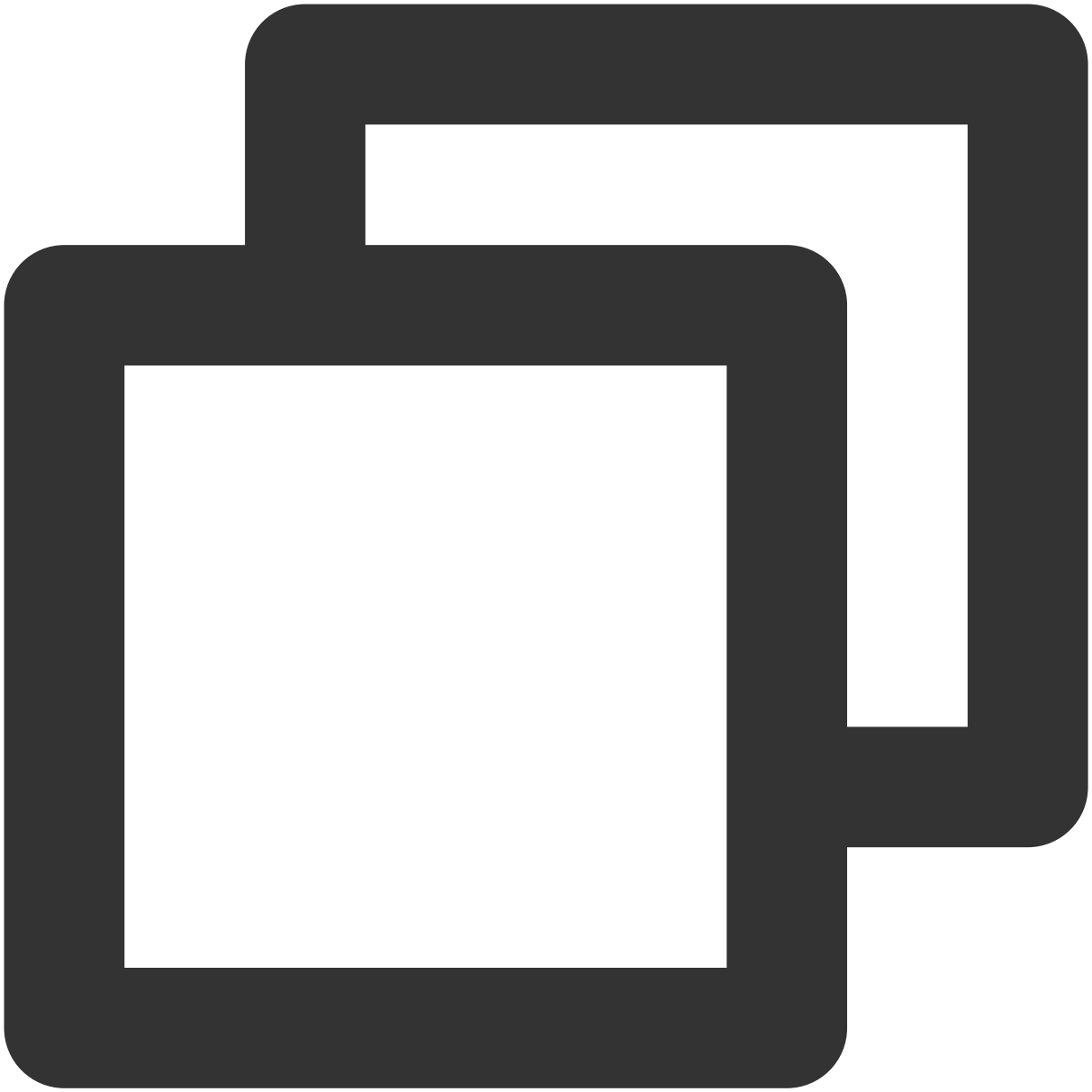
```
ARRAY_MIN(<a> array<T>)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the minimum value in the array. For double/float types, NaN is considered greater than any non-NaN element. Null elements are skipped.

Return type: array<T>

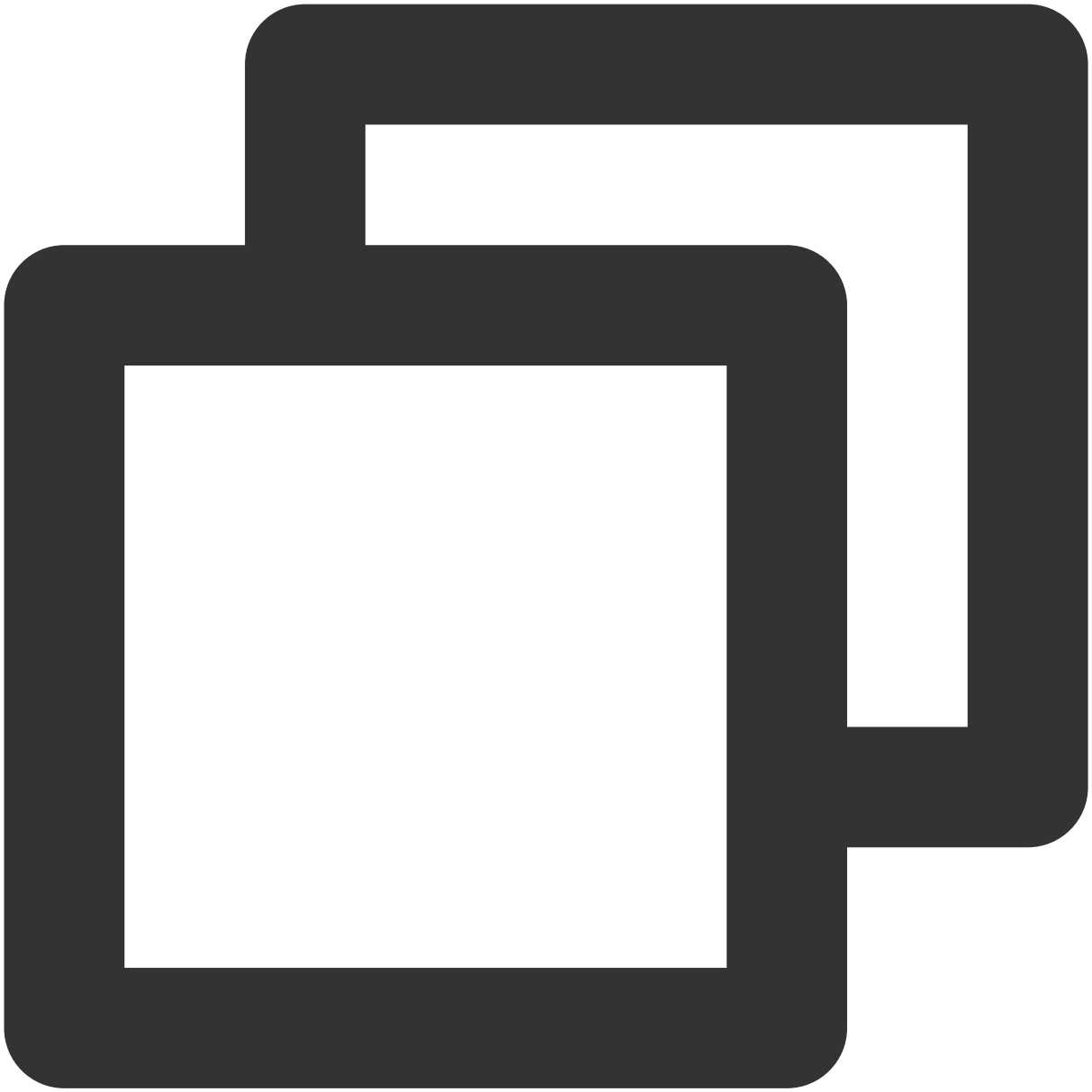
Example:



```
> SELECT array_min(array(1, 20, null, 3));
 1
```

FLATTEN

Function statement:



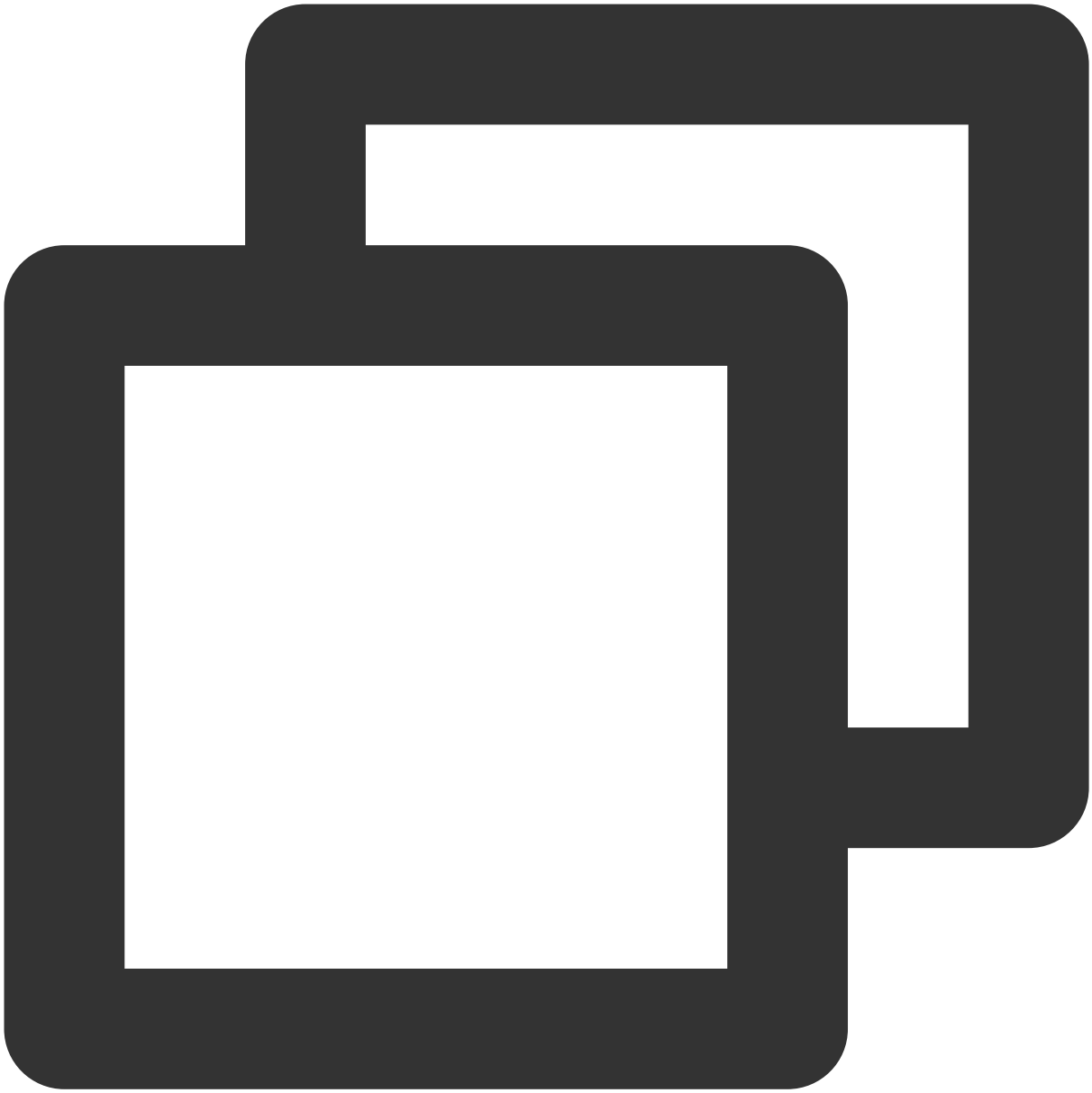
```
FLATTEN(<aa> array<array<T>>
```

Supported engines: SparkSQL and Presto

Usage instructions: Convert a two-dimensional array into a one-dimensional array.

Return type: array<T>

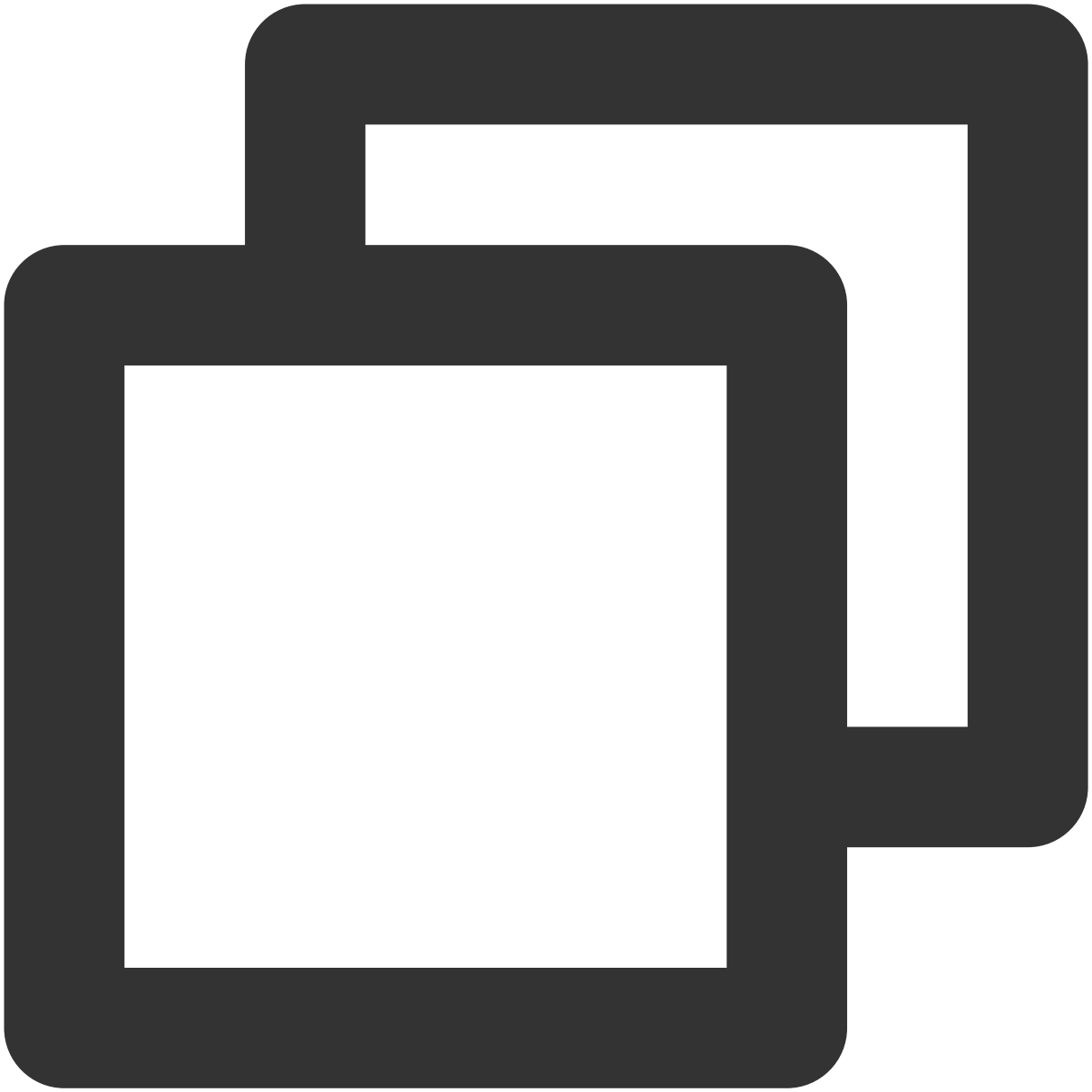
Example:



```
> SELECT flatten(array(array(1, 2), array(3, 4)));
 [1,2,3,4]
```

SEQUENCE

Function statement:



```
SEQUENCE(<start> integer|date|timestamp, end integer|date|timestamp[, step integer|
```

Supported engines: SparkSQL and Presto

Usage instructions: Generate an array from start to end (inclusive), incrementing by step. The type of the returned elements matches the type of start and end.

The start and stop expressions must parse to the same type. If the start and stop expressions are parsed to the date or timestamp type, then step must be parsed to an interval, year-month interval, or day-time interval type; otherwise, it must be parsed to the same type as start and end.

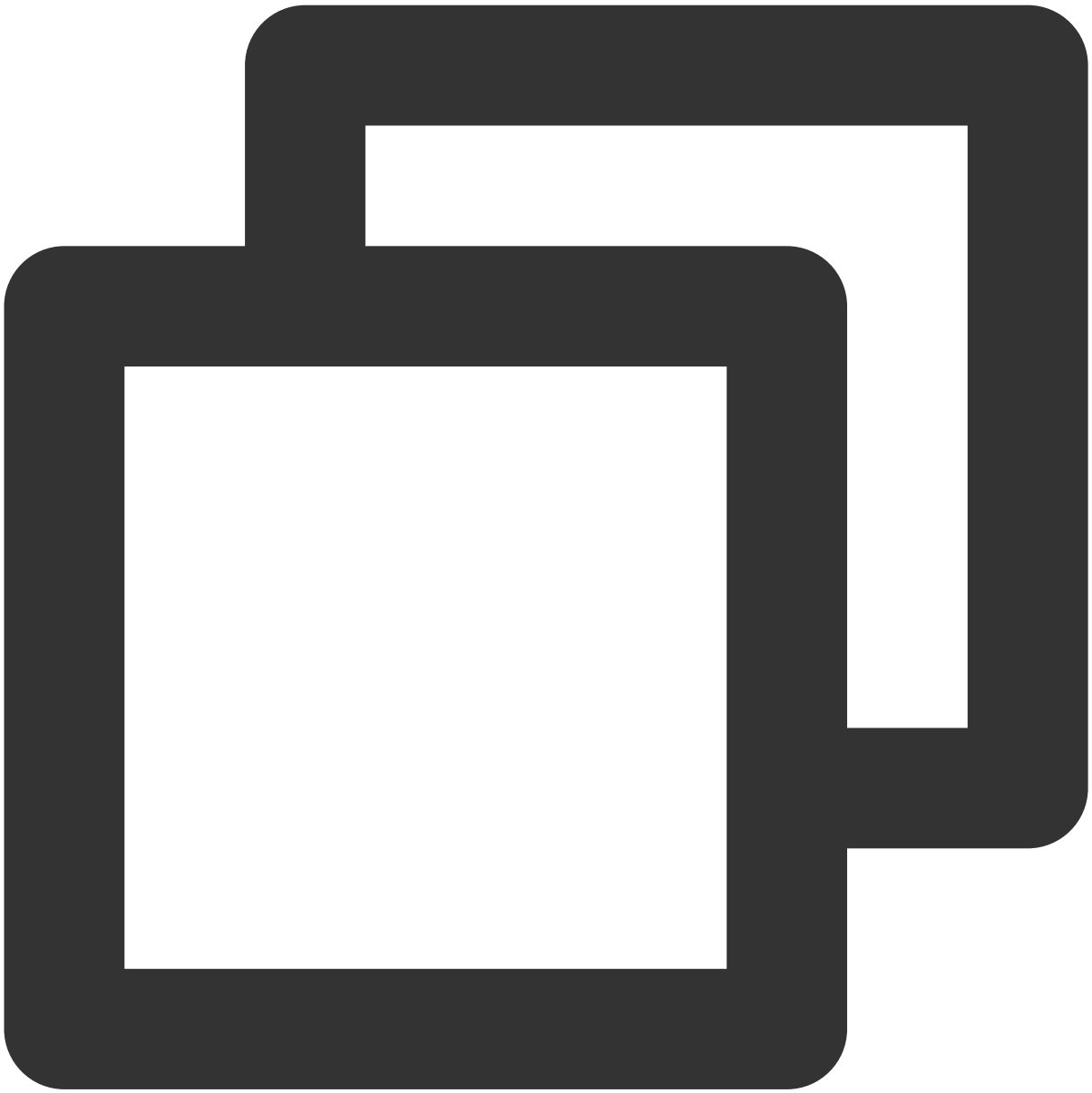
Return type: same as `start`

Example:

```
> SELECT sequence(1, 5);
 [1,2,3,4,5]
> SELECT sequence(5, 1);
 [5,4,3,2,1]
> SELECT sequence(to_date('2018-01-01'), to_date('2018-03-01'), interval 1 month);
 [2018-01-01,2018-02-01,2018-03-01]
> SELECT sequence(to_date('2018-01-01'), to_date('2018-03-01'), interval '0-1' year
 [2018-01-01,2018-02-01,2018-03-01]
```

ARRAY_REPEAT

Function statement:



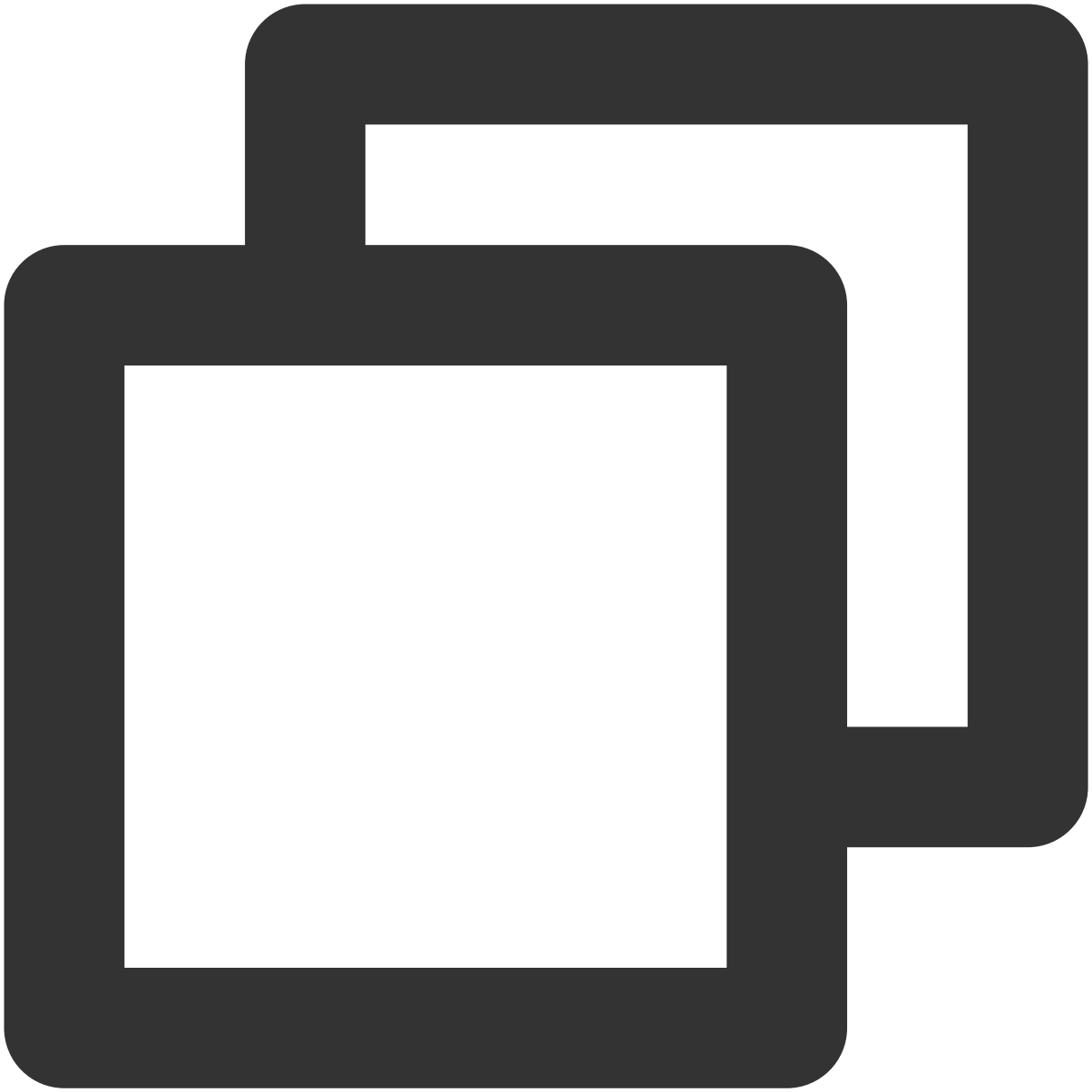
```
ARRAY_REPEAT(<element> T, <count> integer)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return an array containing count occurrences of element.

Return type: array<T>

Example:



```
> SELECT array_repeat('123', 2);
 ["123","123"]
```

ARRAY_REMOVE

Function statement:

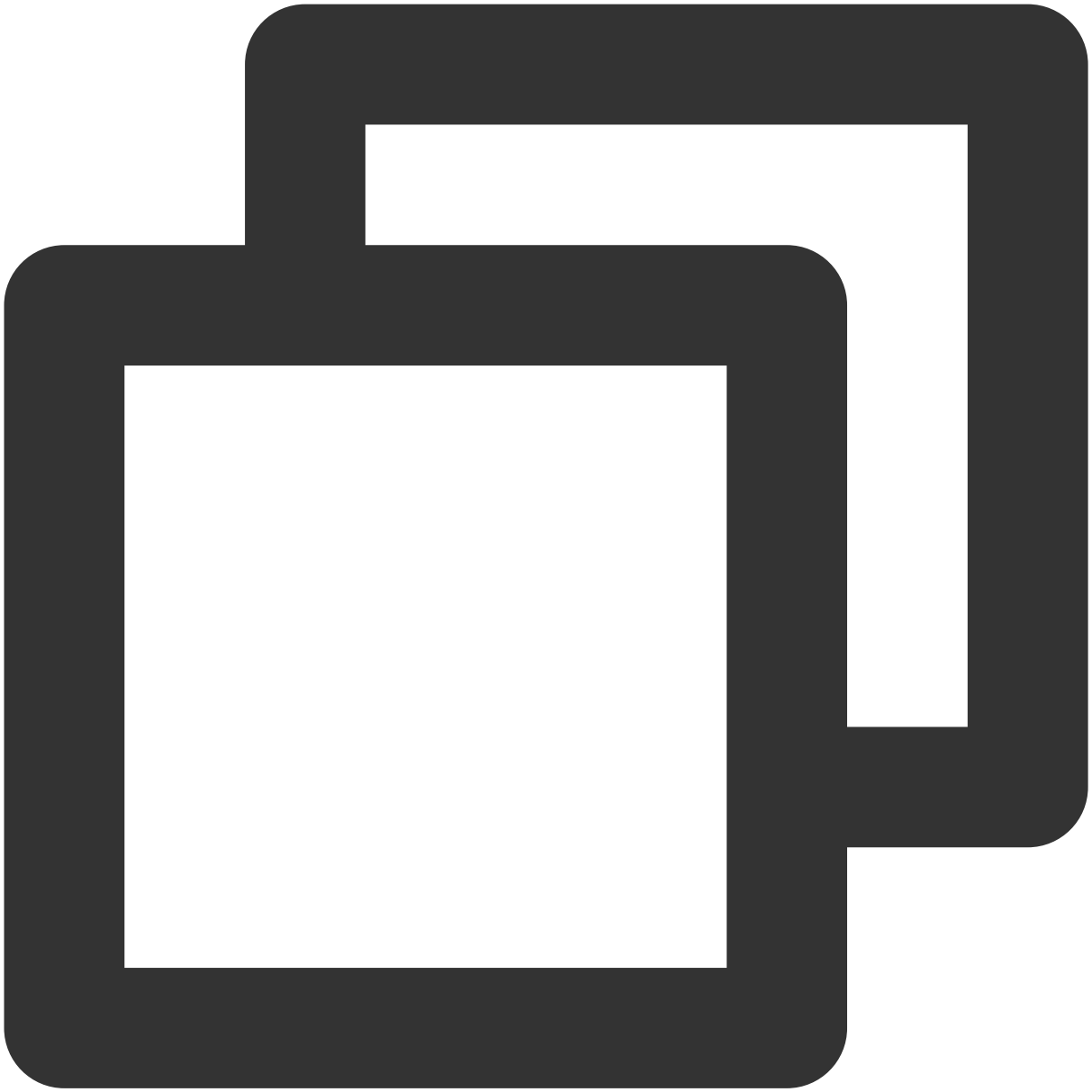
```
ARRAY_REMOVE(<a> array<T>, <element> T)
```

Supported engines: SparkSQL and Presto

Usage instructions: Delete all elements equal to the element from the array.

Return type: array<T>

Example:



```
> SELECT array_remove(array(1, 2, 3, null, 3), 3);
 [1,2,null]
```

ARRAY_DISTINCT

Function statement:

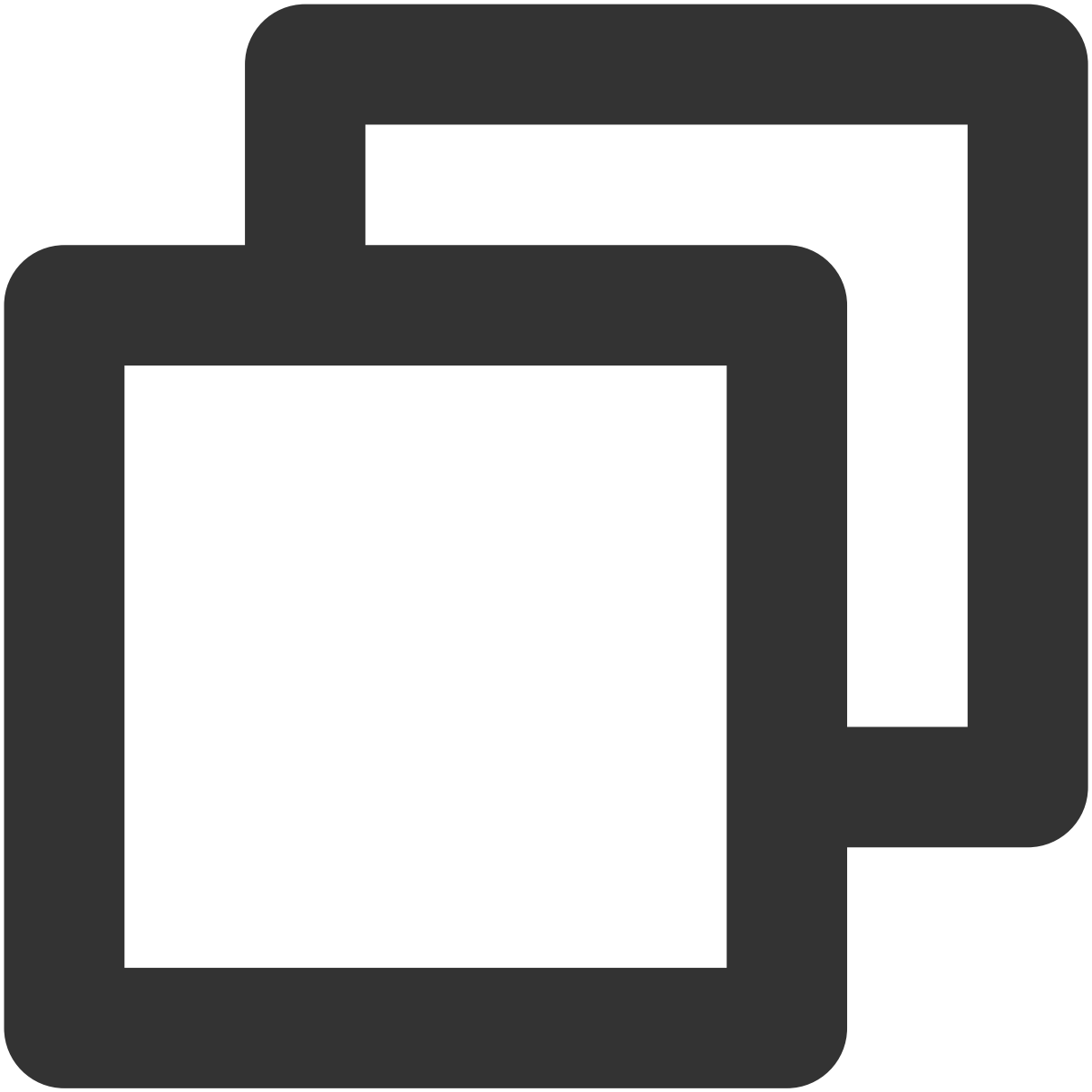
```
ARRAY_DISTINCT(<a> array<T>)
```

Supported engines: SparkSQL and Presto

Usage instructions: Delete duplicate values from the array.

Return type: array<T>

Example:



```
> SELECT array_distinct(array(1, 2, 3, null, 3));
 [1,2,3,null]
```

ELEMENT_AT

Function statement:

```
ELEMENT_AT(<a> array<T>, <index> integer)
ELEMENT_AT(<m> map<K, V>, <key> K)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the array element at the given (1-based) index or the value for the given key.

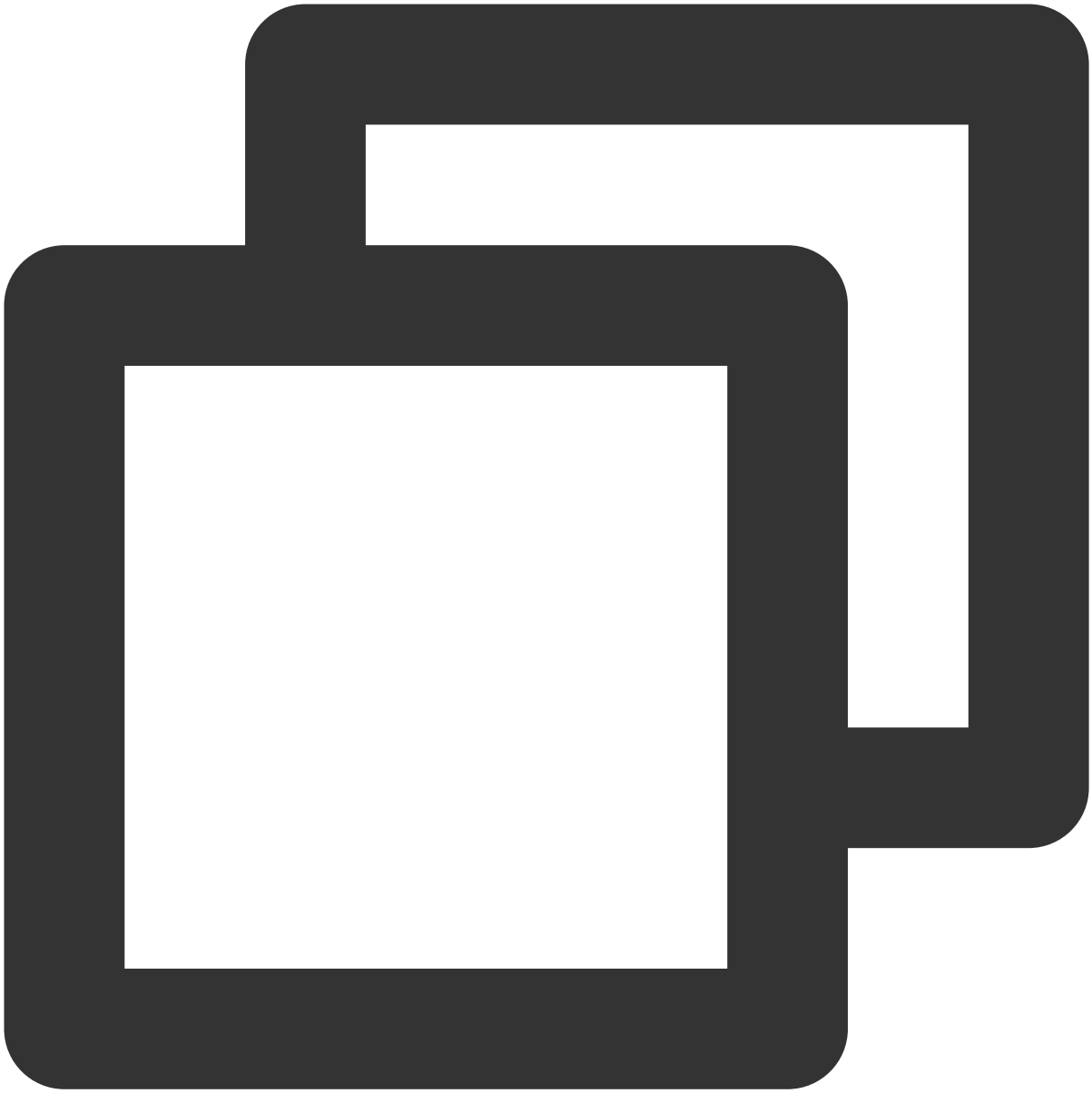
Return type: T and V

Example:

```
> SELECT element_at(array(1, 2, 3), 2);
 2
> SELECT element_at(map(1, 'a', 2, 'b'), 2);
 b
```

MAP

Function statement:



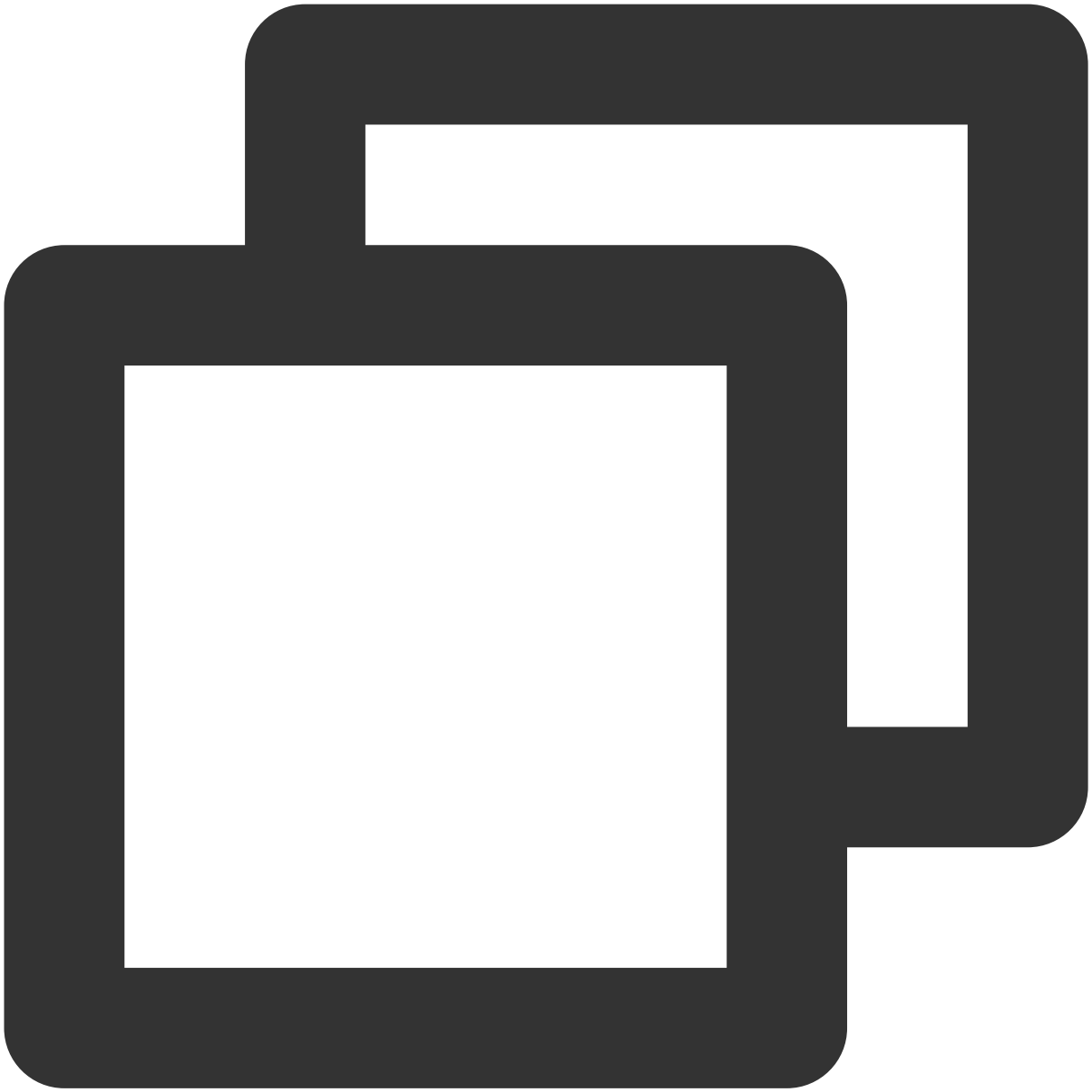
```
MAP(<k1> K, <v1> V, ...)
```

Supported engines: SparkSQL and Presto

Usage instructions: Generate a map from the given elements.

Return type: MAP<K, V>

Example:



```
> SELECT map(1.0, '2', 3.0, '4');
 {1.0:"2",3.0:"4"}
```

MAP_FROM_ARRAYS

Function statement:

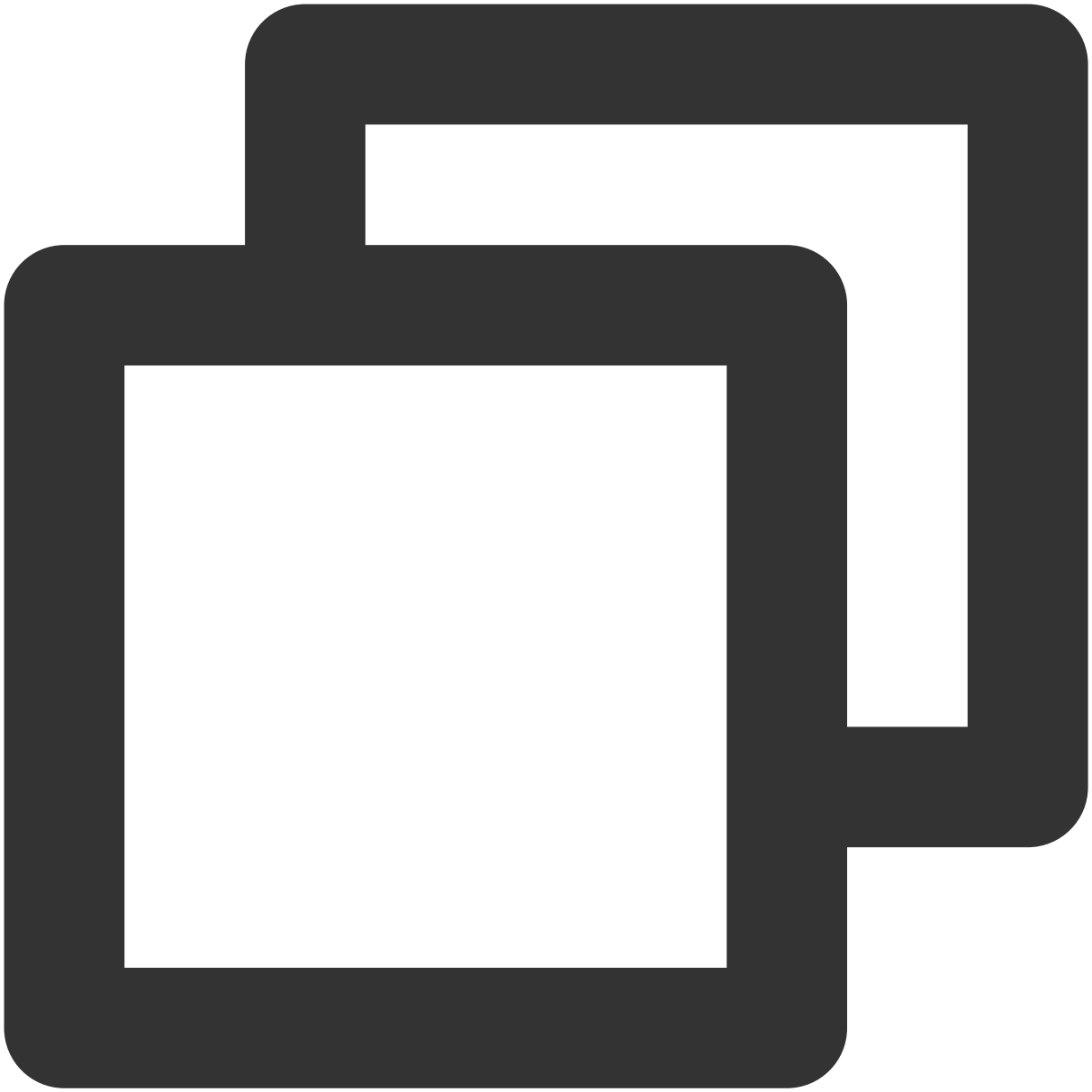
```
MAP_FROM_ARRAYS(<keys> array<K>, <values> array<V>)
```

Supported engines: SparkSQL and Presto

Usage instructions: Create a map using the given key/value arrays. None of the elements in keys should be null.

Return type: map<K, V>

Example:



```
> SELECT map_from_arrays(array(1.0, 3.0), array('2', '4'));
 {1.0:"2",3.0:"4"}
```

MAP_KEYS

Function statement:

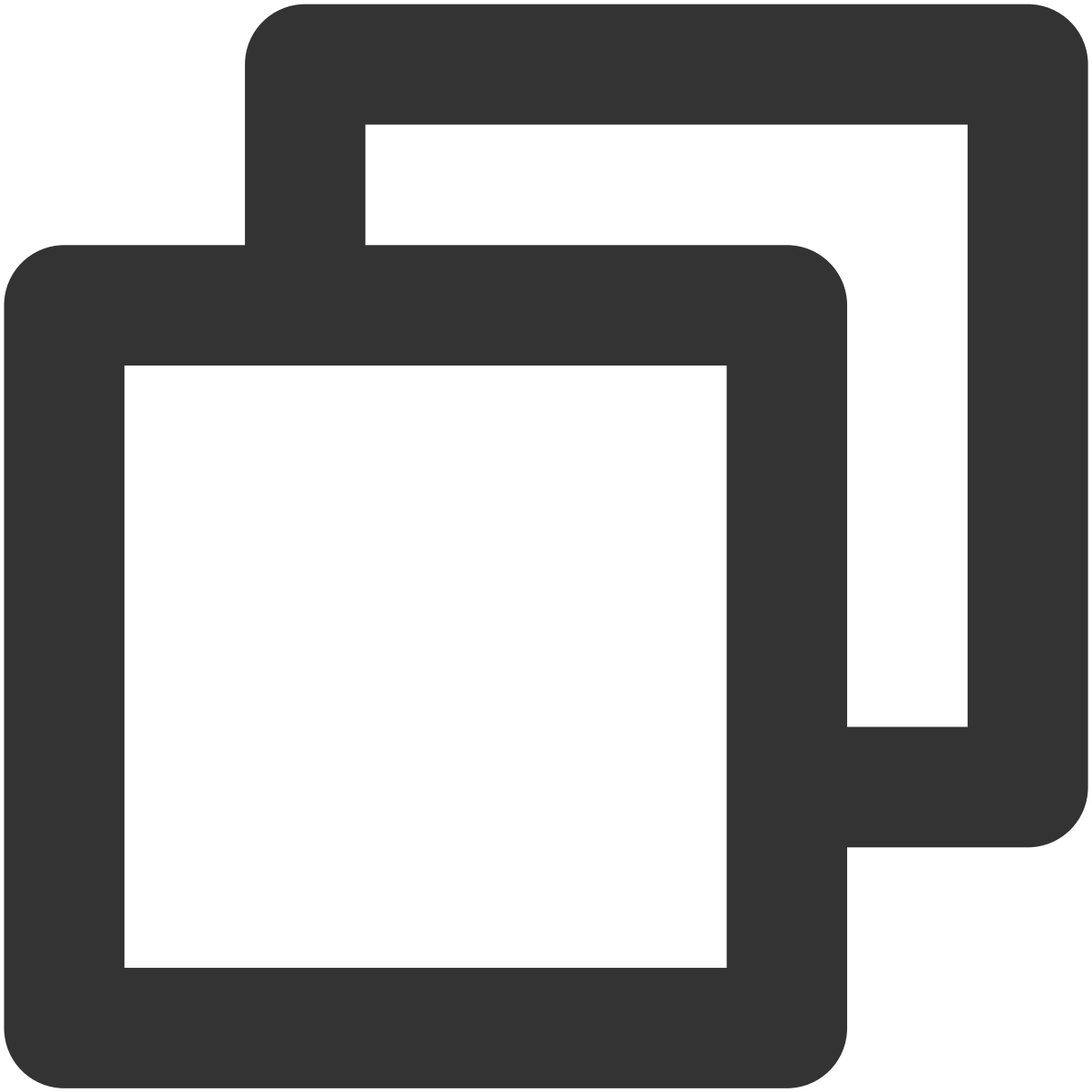
```
MAP_KEYS(<m> map<K, V>)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return an unordered array containing the keys of the map.

Return type: array<K>

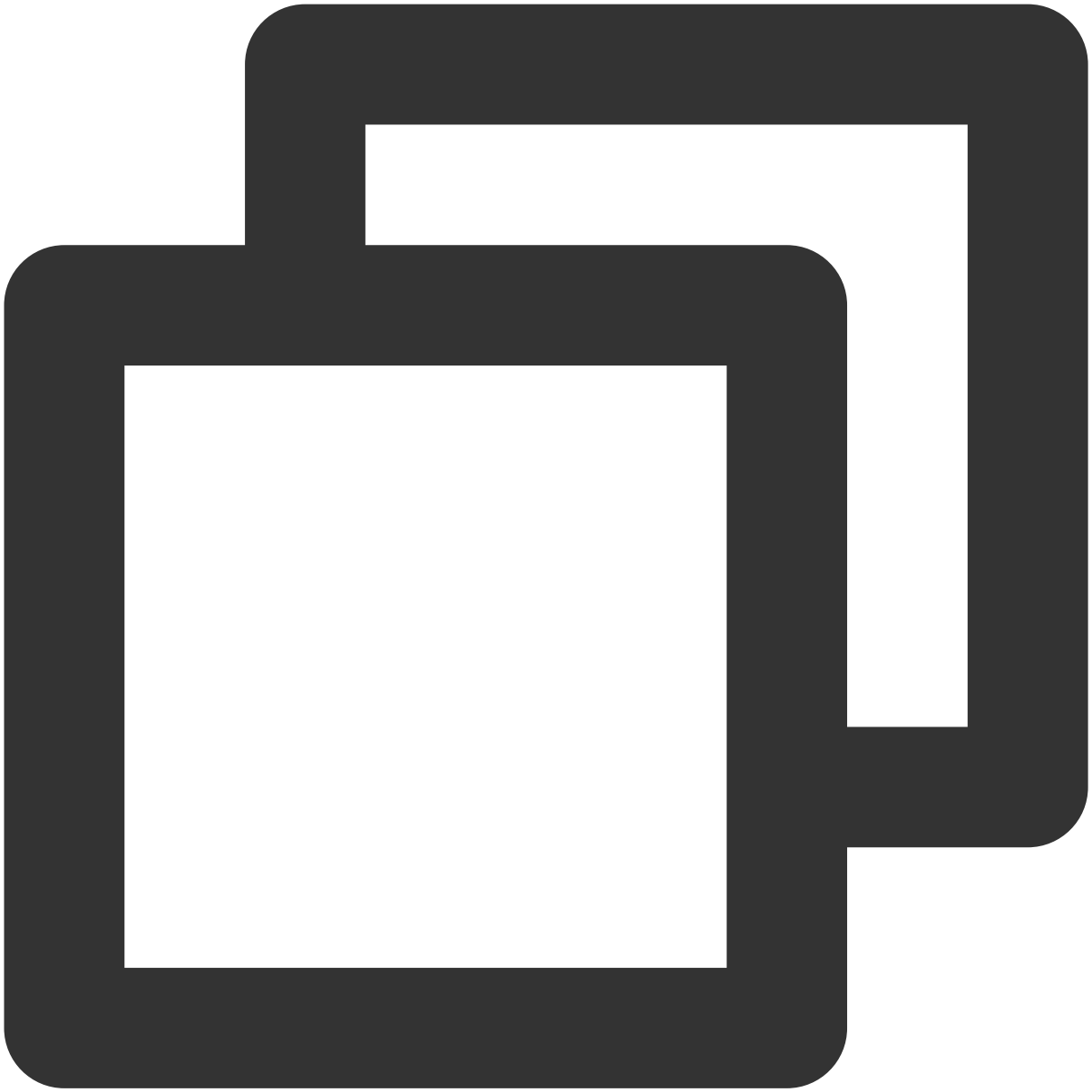
Example:



```
> SELECT map_keys(map(1, 'a', 2, 'b'));
 [1,2]
```

MAP_VALUES

Function statement:



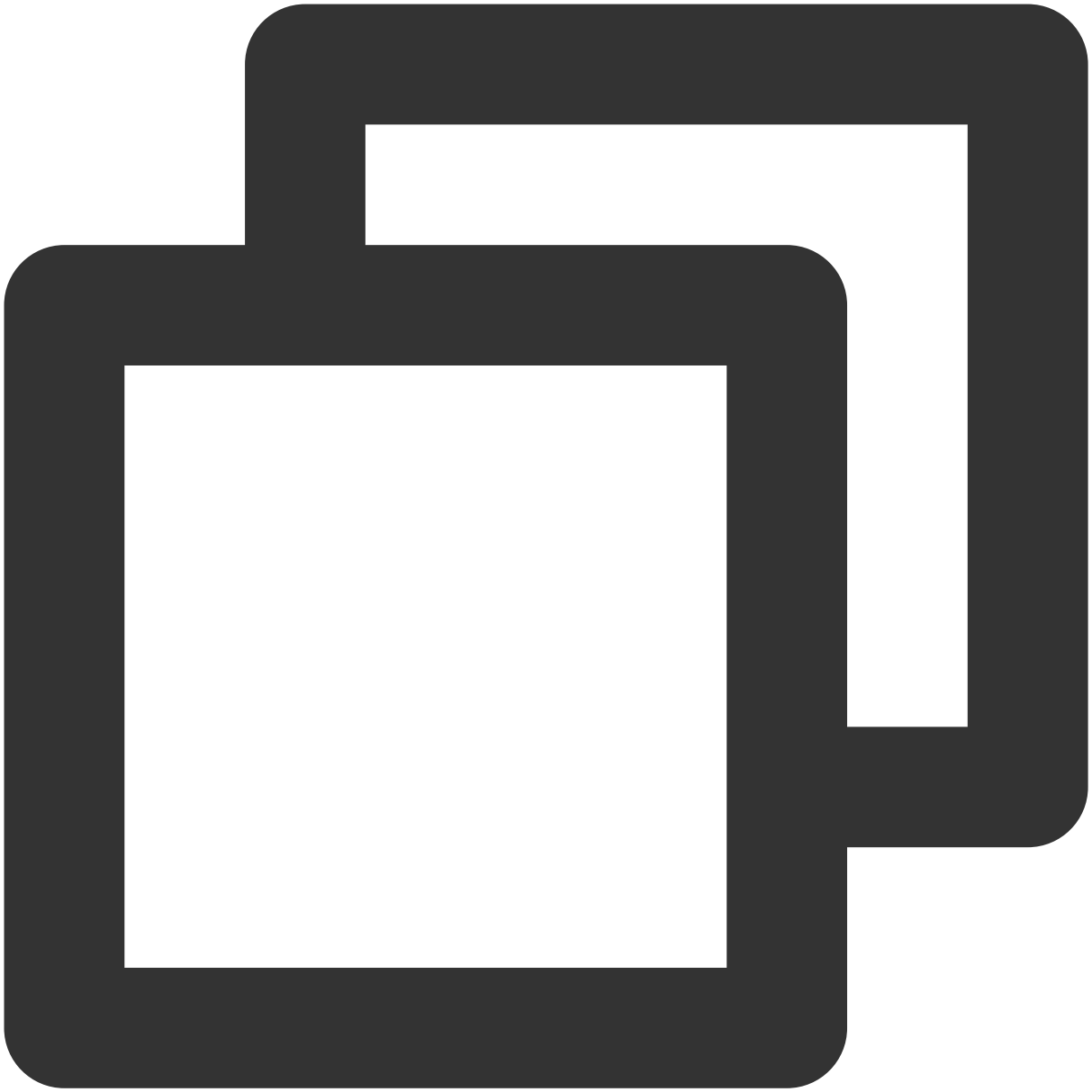
```
MAP_VALUES(<m> map<K, V>)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return an unordered array containing the values of the map.

Return type: array<V>

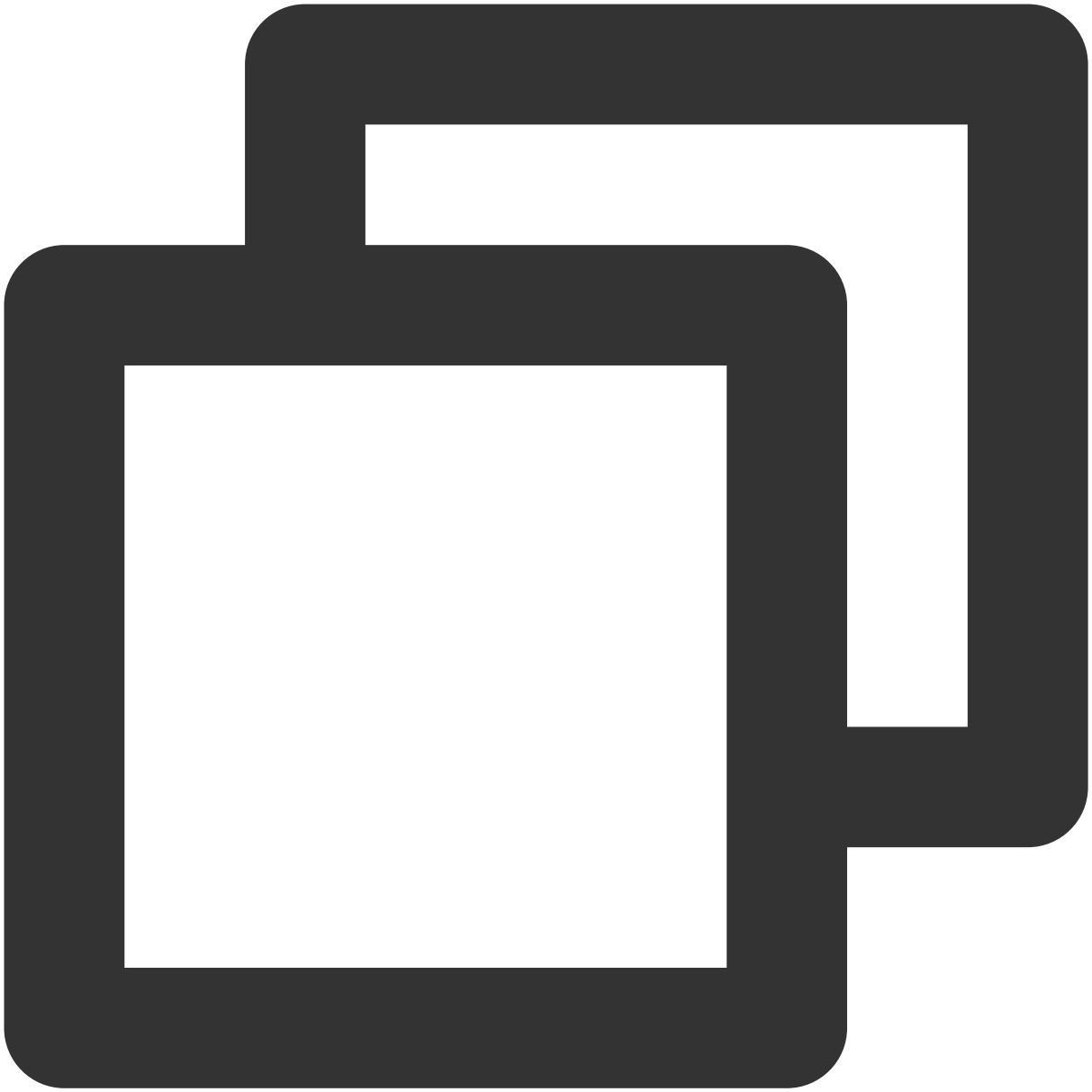
Example:



```
> SELECT map_values(map(1, 'a', 2, 'b'));
 ["a","b"]
```

MAP_ENTRIES

Function statement:



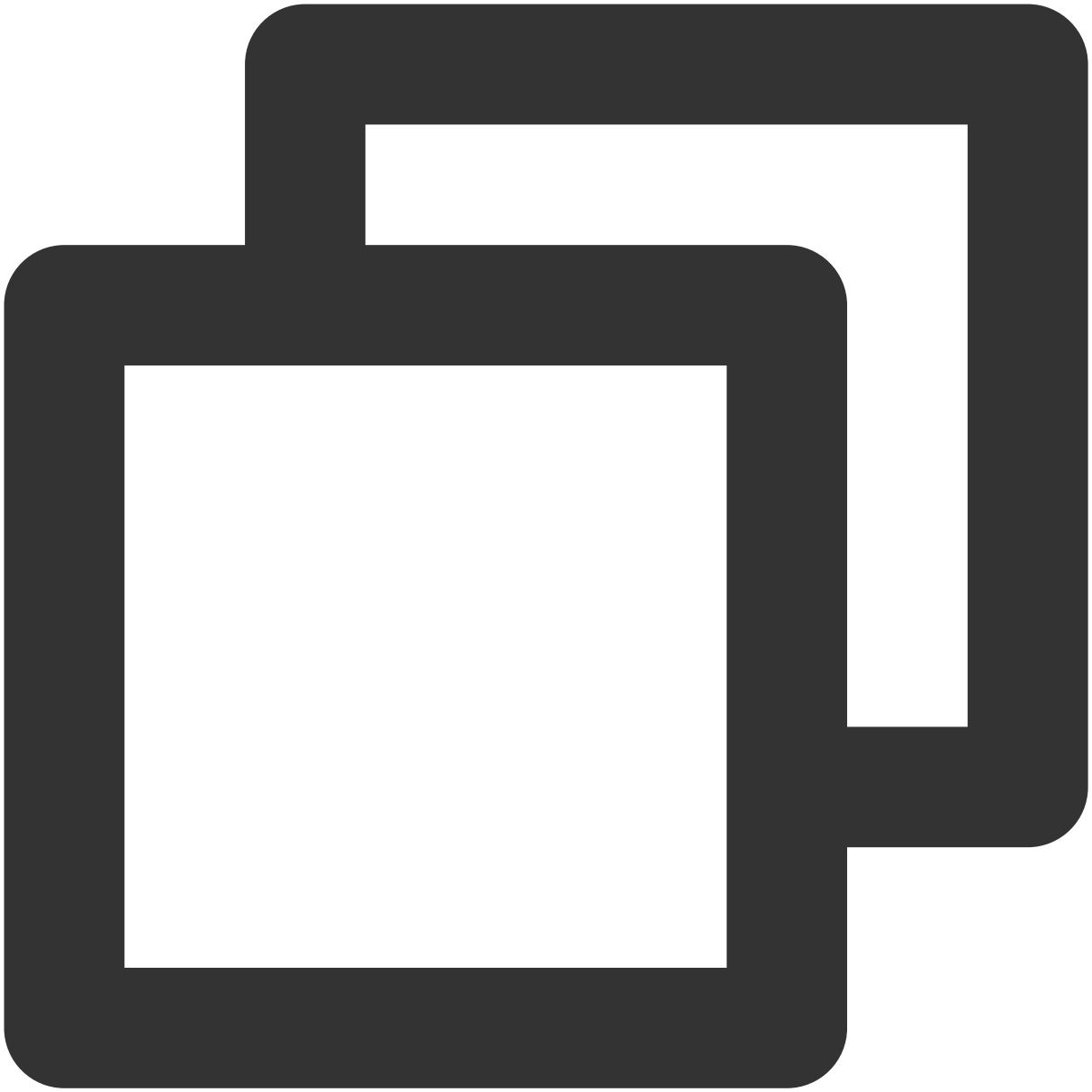
```
MAP_ENTRIES(<m> map<K, V>)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return an unordered array of all entries in the given map.

Return type: array<struct<K, V>>

Example:



```
> SELECT map_entries(map(1, 'a', 2, 'b'));
 [{"key":1,"value":"a"},{"key":2,"value":"b"}]
```

MAP_FROM_ENTRIES

Function statement:

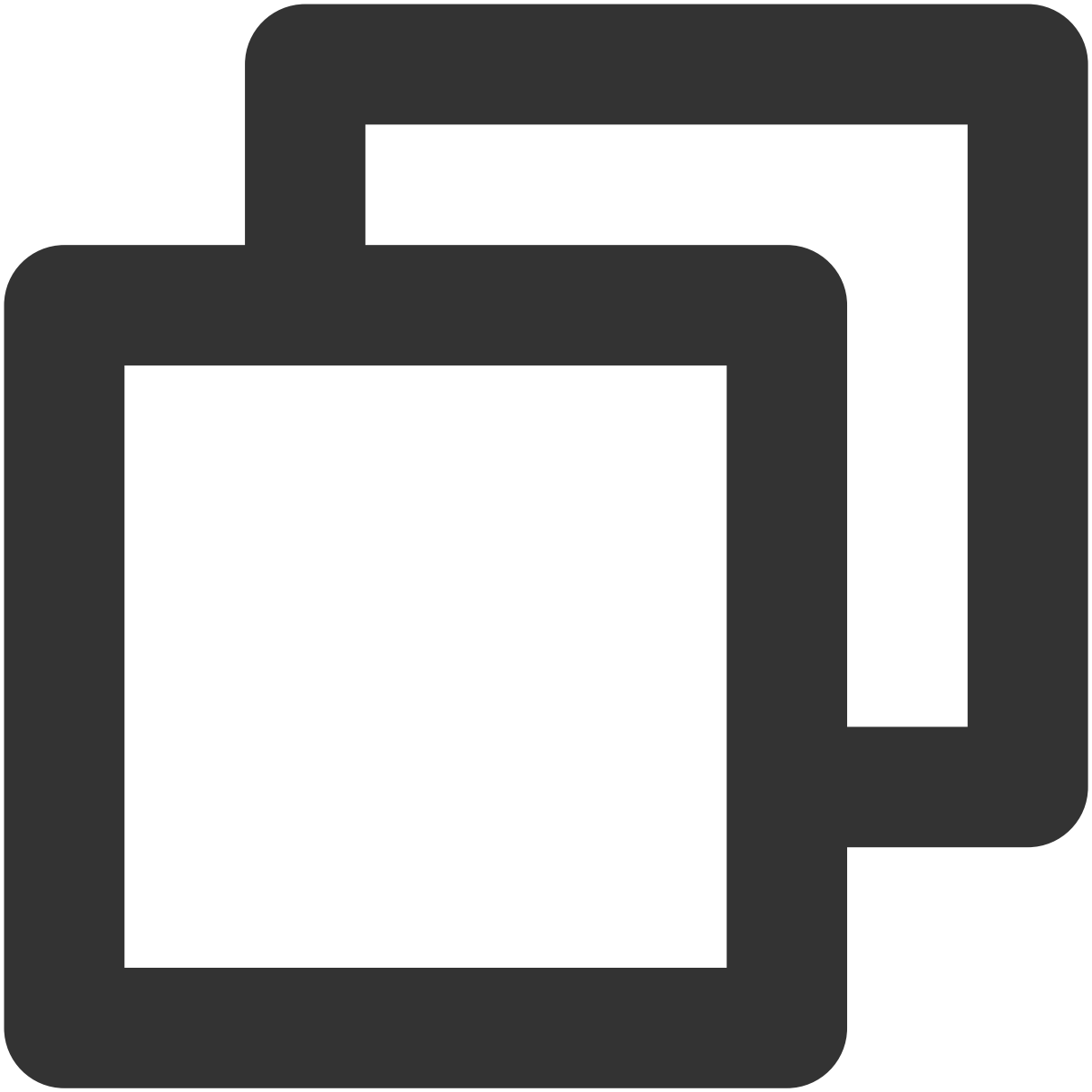
```
MAP_FROM_ENTRIES(<entries> array<struct<K, V>>)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return a map created from the given array of entries.

Return type: map<K, V>

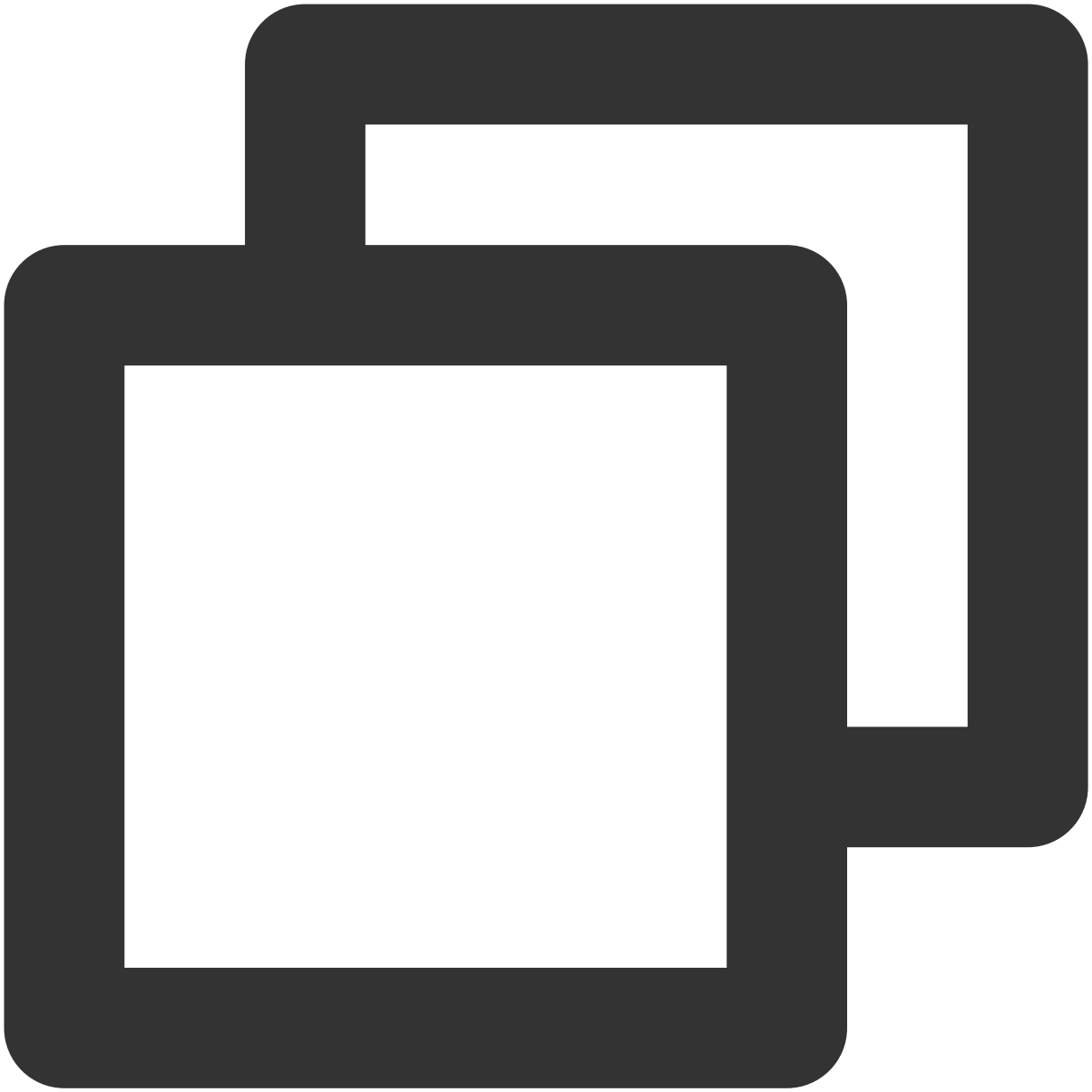
Example:



```
> SELECT map_from_entries(array(struct(1, 'a'), struct(2, 'b')));
 {1:"a",2:"b"}
```

MAP_CONCAT

Function statement:



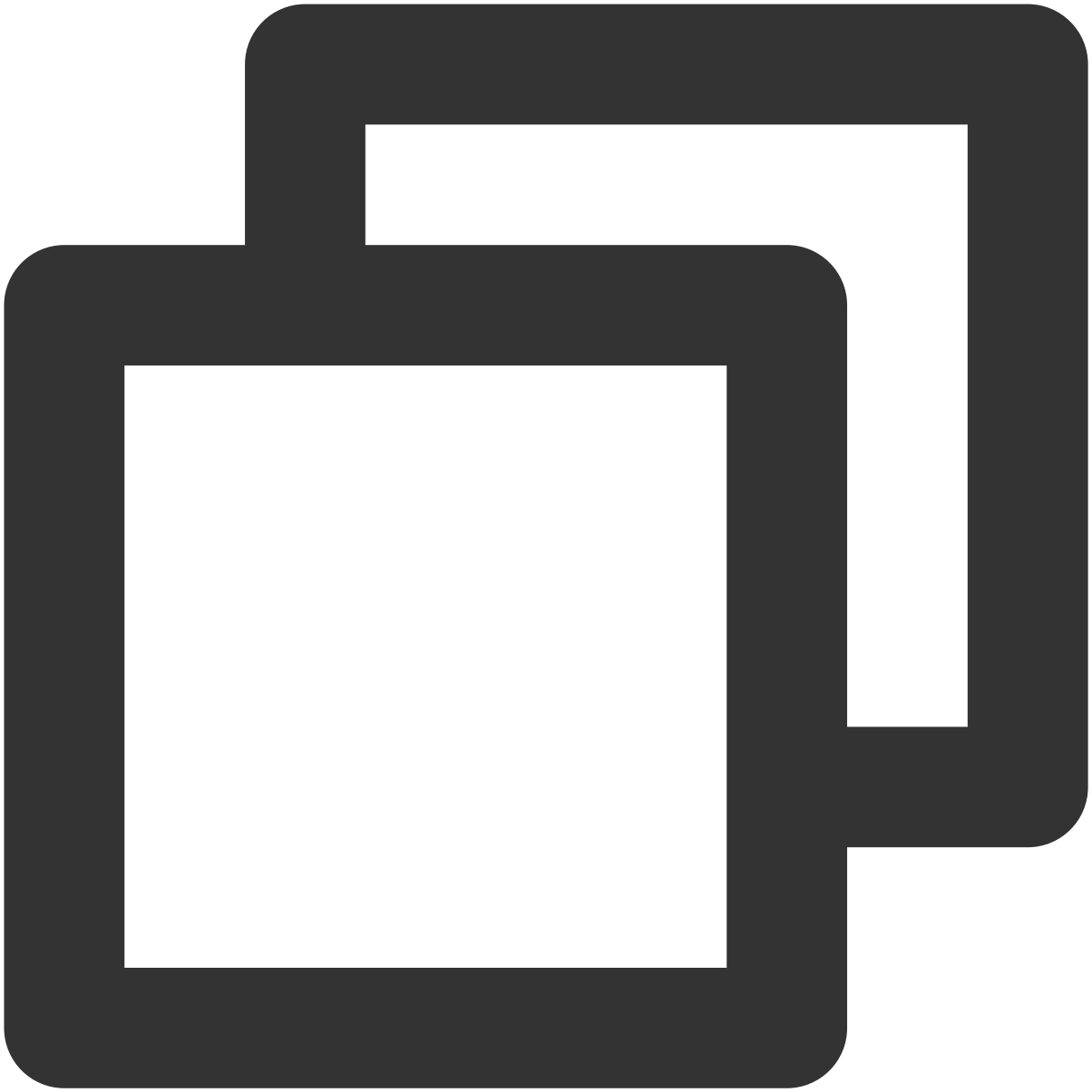
```
MAP_CONCAT(map1 map<K, V>, ...)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the union of all given maps.

Return type: map<K, V>

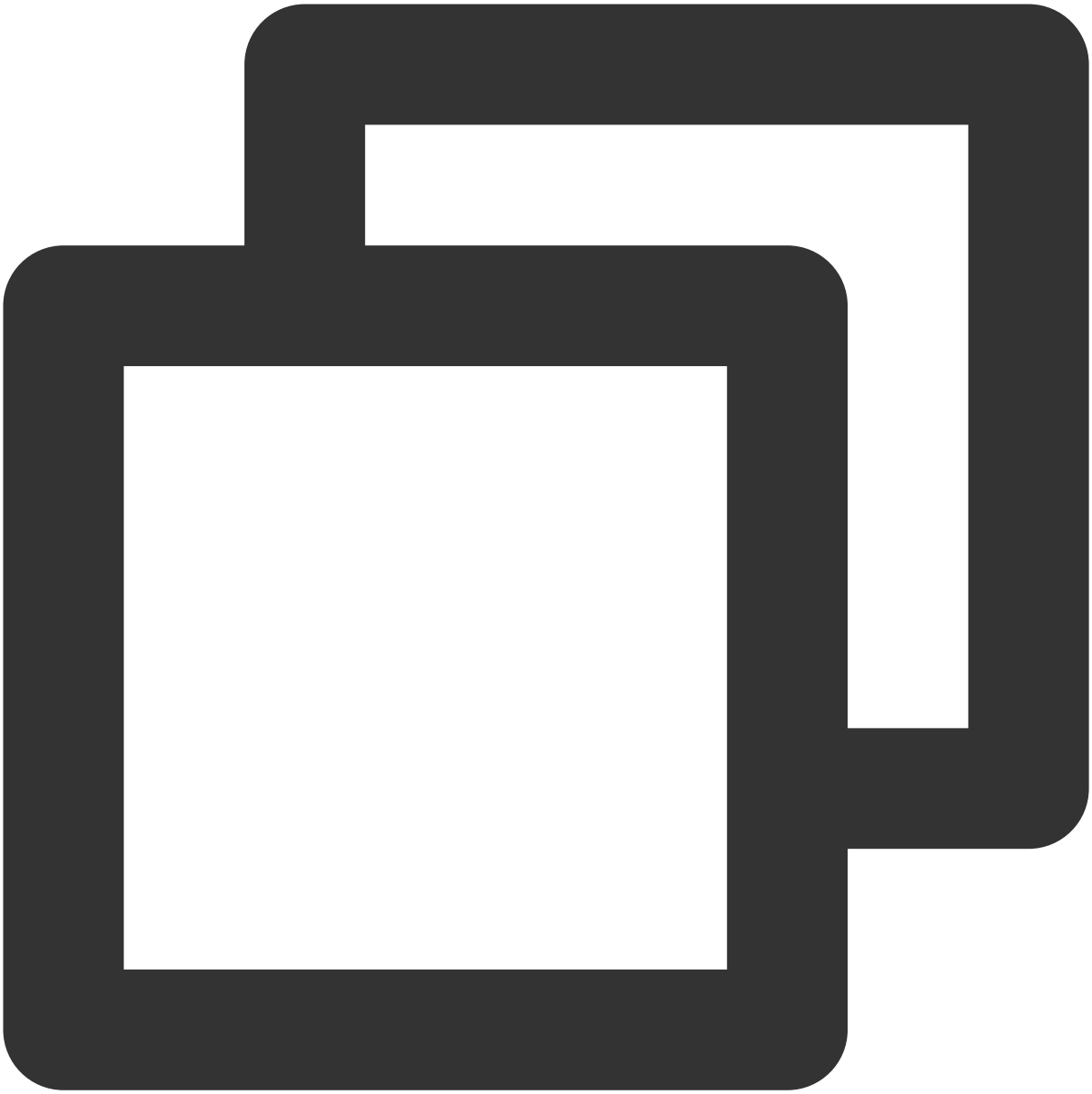
Example:



```
> SELECT map_concat(map(1, 'a', 2, 'b'), map(3, 'c'));
 {1:"a",2:"b",3:"c"}
```

MAP_FILTER

Function statement:



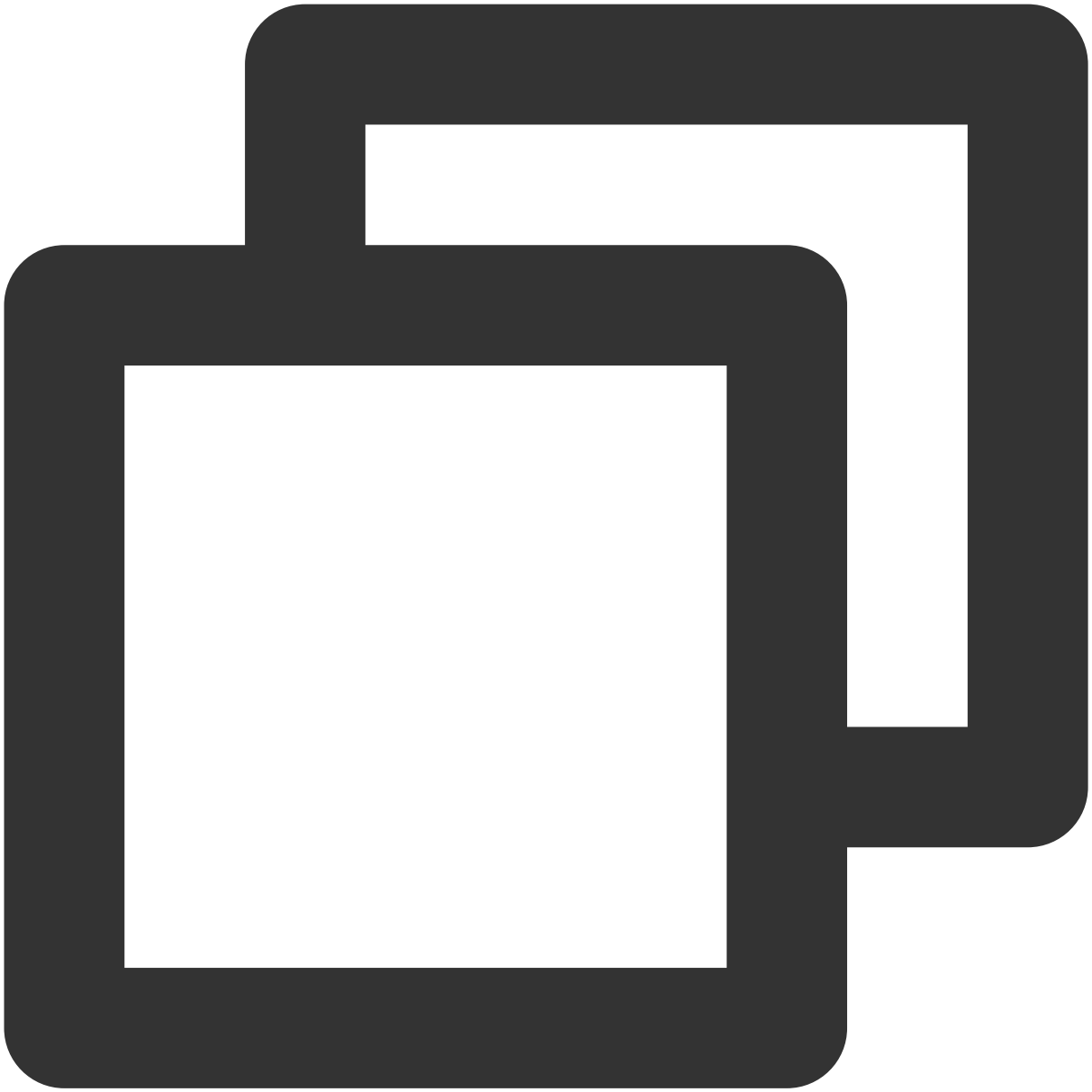
```
MAP_FILTER(<m> map<K, V>, <func> function(K, V)->boolean)
```

Supported engine: SparkSQL

Usage instructions: Filter the entries in m using func.

Return type: map<K, V>

Example:



```
> SELECT map_filter(map(1, 0, 2, 2, 3, -1), (k, v) -> k > v);
 {1:0,3:-1}
```

MAP_ZIP_WITH

Function statement:

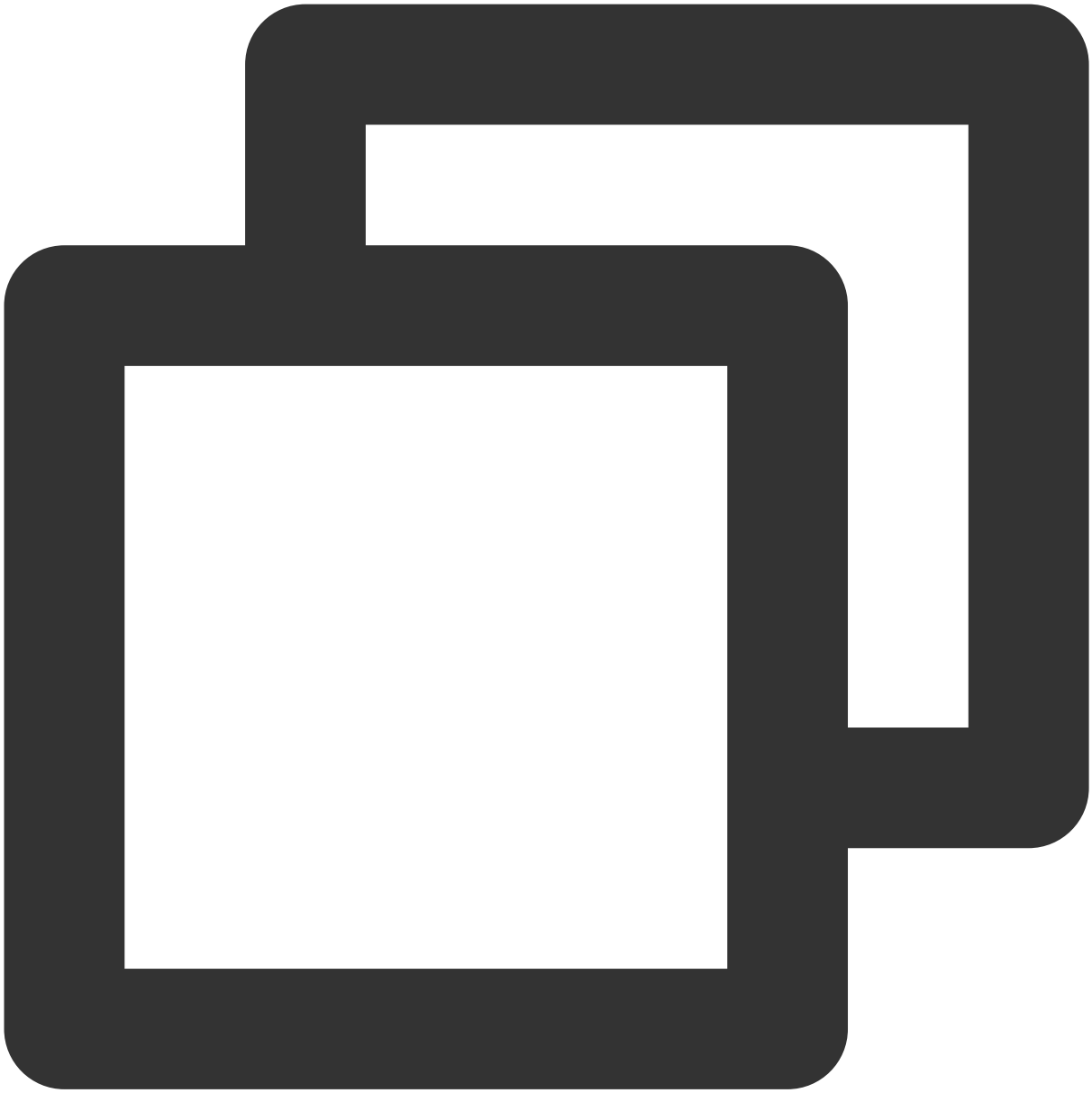
```
MAP_ZIP_WITH(<map1> map<K, V1>, <map2> map<K, V2>, <func> function(K, V1, V2)->R)
```

Supported engine: SparkSQL

Usage instructions: Merge two given maps into a single map using func. For keys that appear in only one map, their values will be set to NULL. If the input maps contain duplicate keys, only the first entry for each duplicate key is transmitted to func.

Return type: MAP<K, R>

Example:



```
> SELECT map_zip_with(map(1, 'a', 2, 'b'), map(1, 'x', 2, 'y'), (k, v1, v2) -> conc
 {1:"ax",2:"by"}
```

TRANSFORM_KEYS

Function statement:

```
TRANSFORM_KEYS(<m> map<K, V>, <func> function(K, V)->R)
```

Supported engine: SparkSQL

Usage instructions: Transform the keys in the map using func.

Return type: map<R, V>

Example:

```
> SELECT transform_keys(map_from_arrays(array(1, 2, 3), array(1, 2, 3)), (k, v) ->
 {2:1,3:2,4:3}
> SELECT transform_keys(map_from_arrays(array(1, 2, 3), array(1, 2, 3)), (k, v) ->
 {2:1,4:2,6:3}
```

TRANSFORM_VALUES

Function statement:

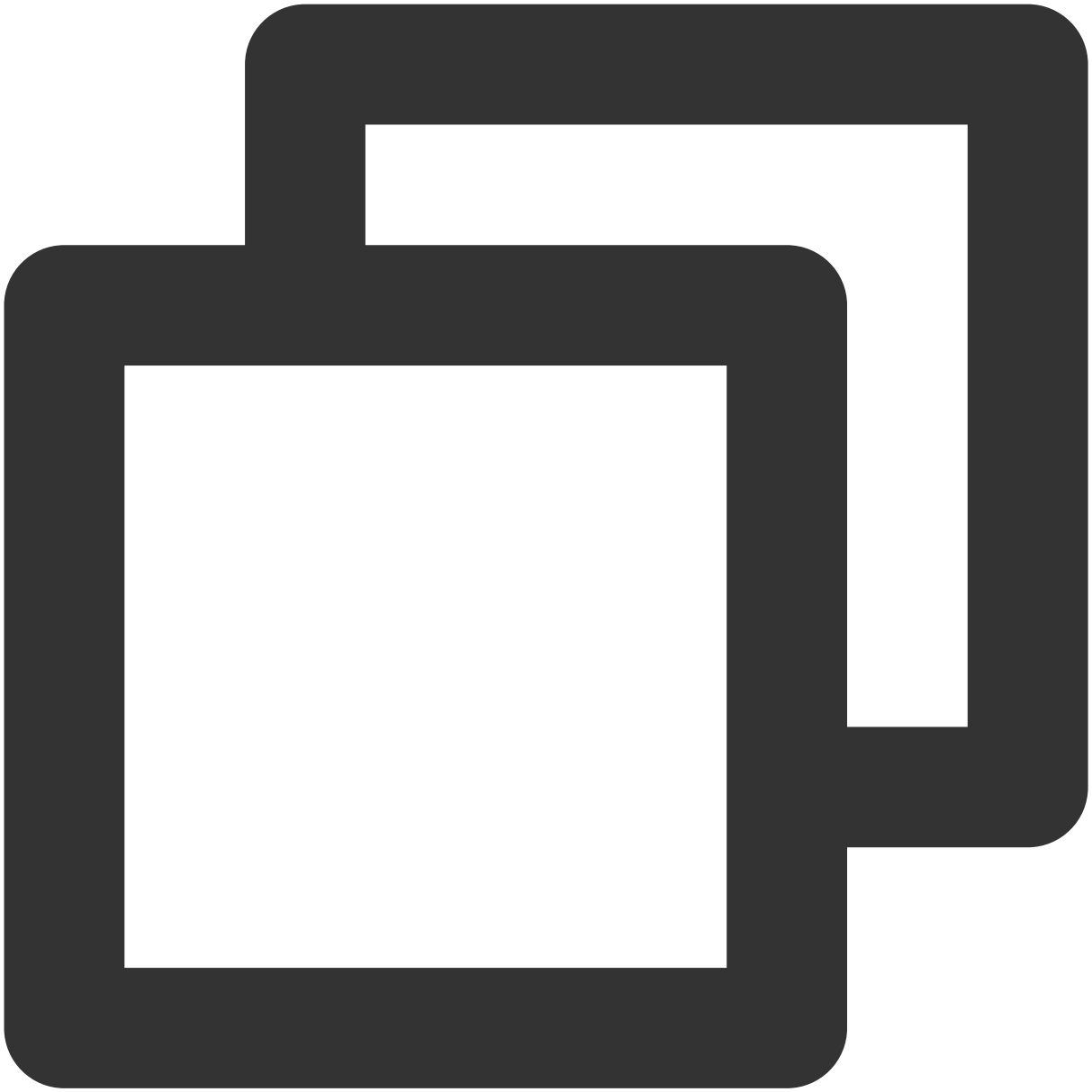
```
TRANSFORM_VALUES(<m> map<K, V>, <func> function(K, V)->R)
```

Supported engine: SparkSQL

Usage instructions: Transform the values in the map using func.

Return type: map<K, R>

Example:



```
> SELECT transform_values(map_from_arrays(array(1, 2, 3), array(1, 2, 3)), (k, v) -
 {1:2,2:3,3:4}
> SELECT transform_values(map_from_arrays(array(1, 2, 3), array(1, 2, 3)), (k, v) -
 {1:2,2:4,3:6}
```

SIZE

Function statement:

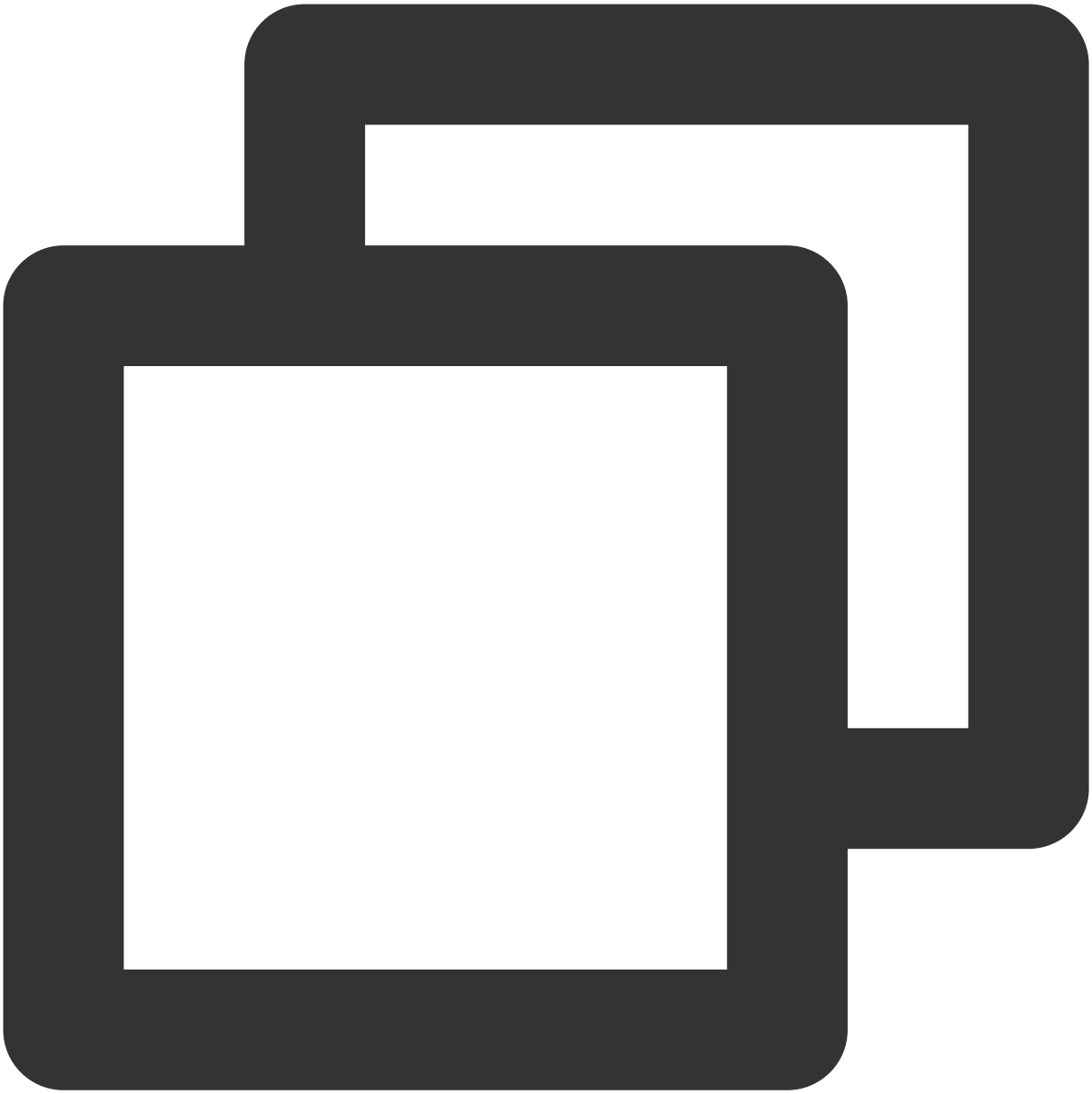
```
SIZE(<expr> array<T>|map<K, V>)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the size of an array or map.

Return type: integer

Example:



```
> SELECT size(array('b', 'd', 'c', 'a'));
 4
> SELECT size(map('a', 1, 'b', 2));
 2
> SELECT size(NULL);
 -1
```

CARDINALITY

Function statement:

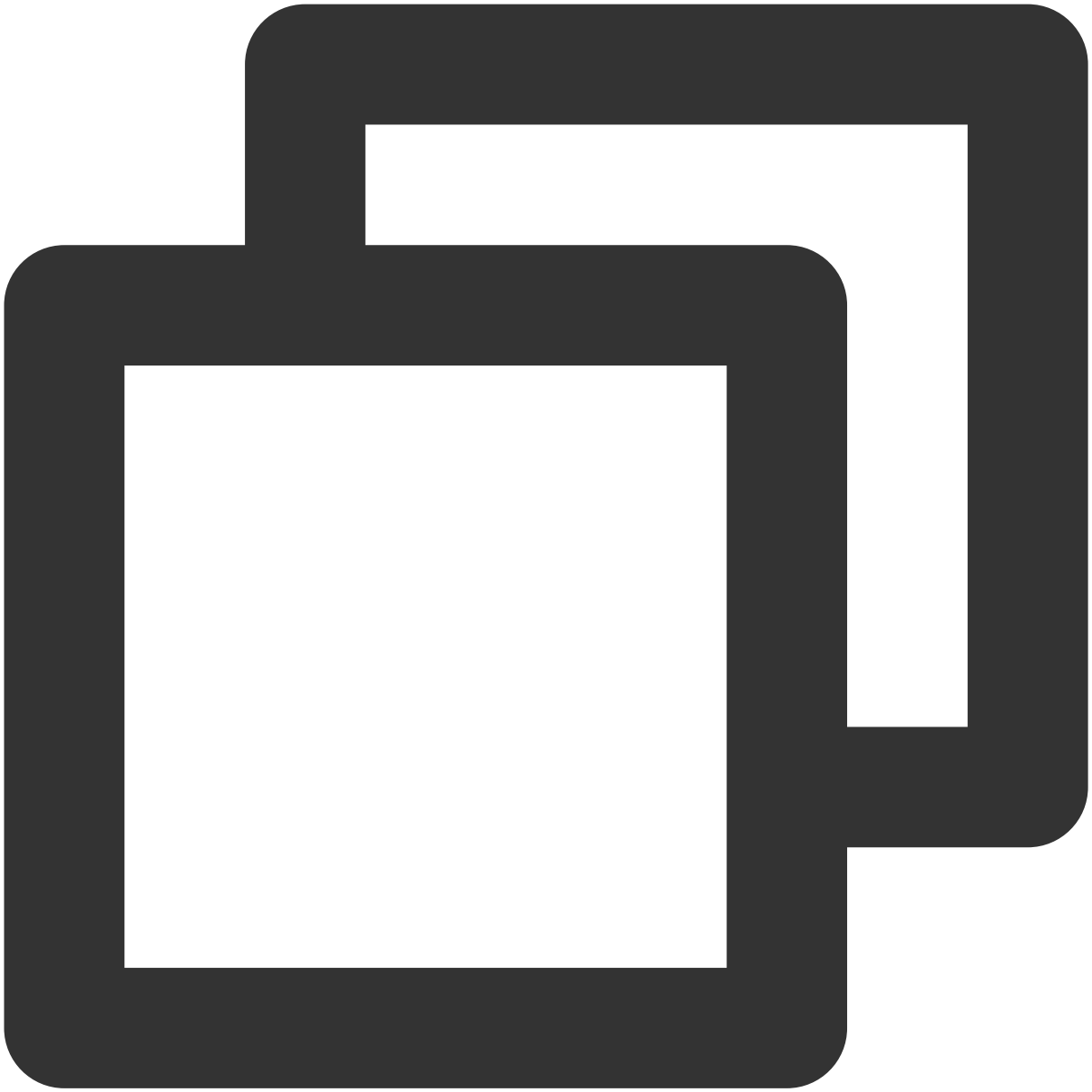
```
CARDINALITY(<expr> array<T>|map<K, V>)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the size of an array or map.

Return type: integer

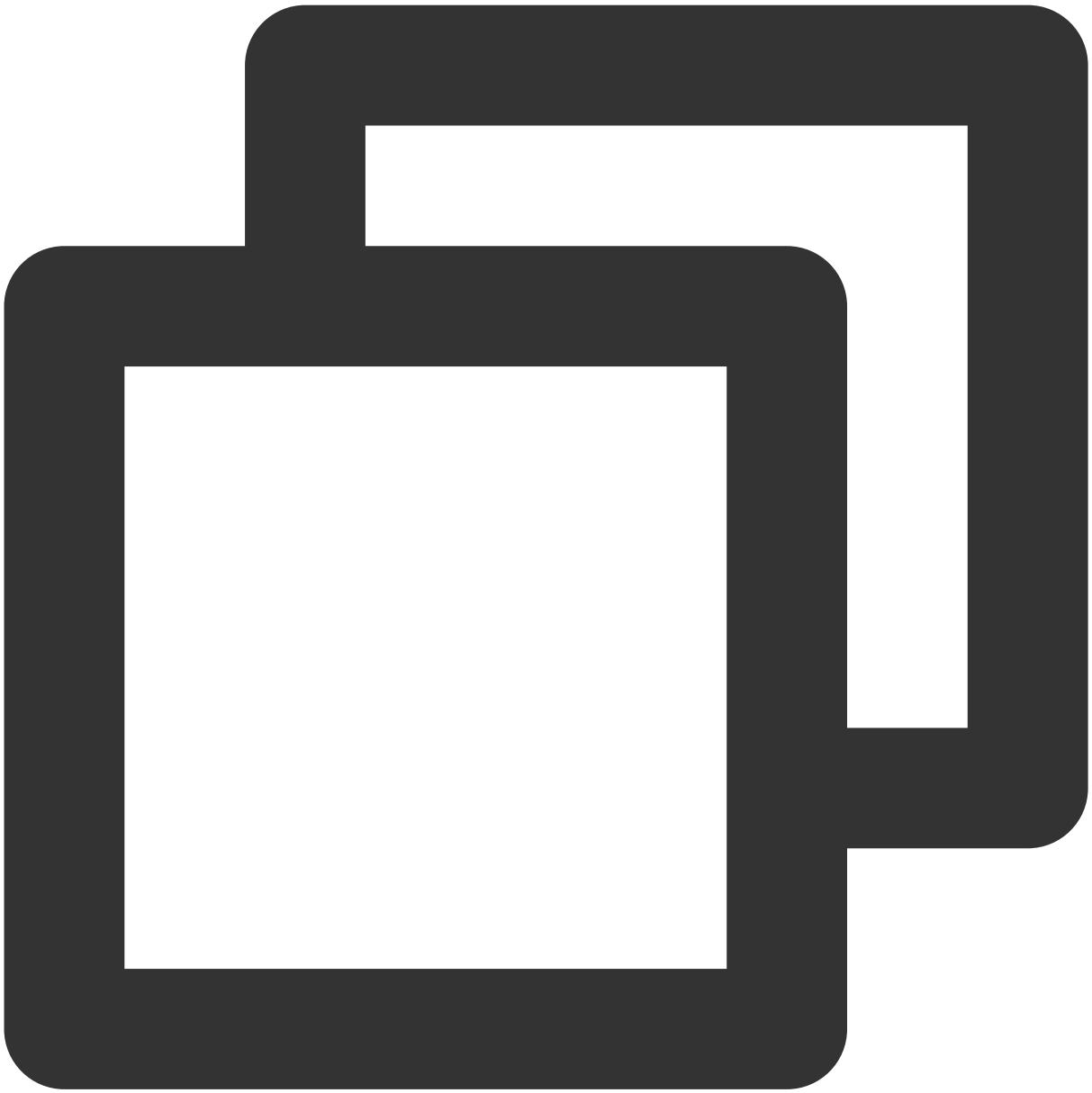
Example:



```
> SELECT cardinality(array('b', 'd', 'c', 'a'));
 4
> SELECT cardinality(map('a', 1, 'b', 2));
 2
> SELECT cardinality(NULL);
 -1
```

ANY_MATCH / FORALL

Function statement:



```
ANY_MATCH(<expr> array<T>, x -> lambda(x))
FORALL(<expr> array<T>, x -> lambda(x))
```

Supported engine: SparkSQL

Usage instructions: For each element in the array, execute the lambda expression in sequence. The function returns true if at least one element returns true; otherwise, it returns false.

If any element in the array is NULL, the function returns NULL.

Return type: boolean

Example:

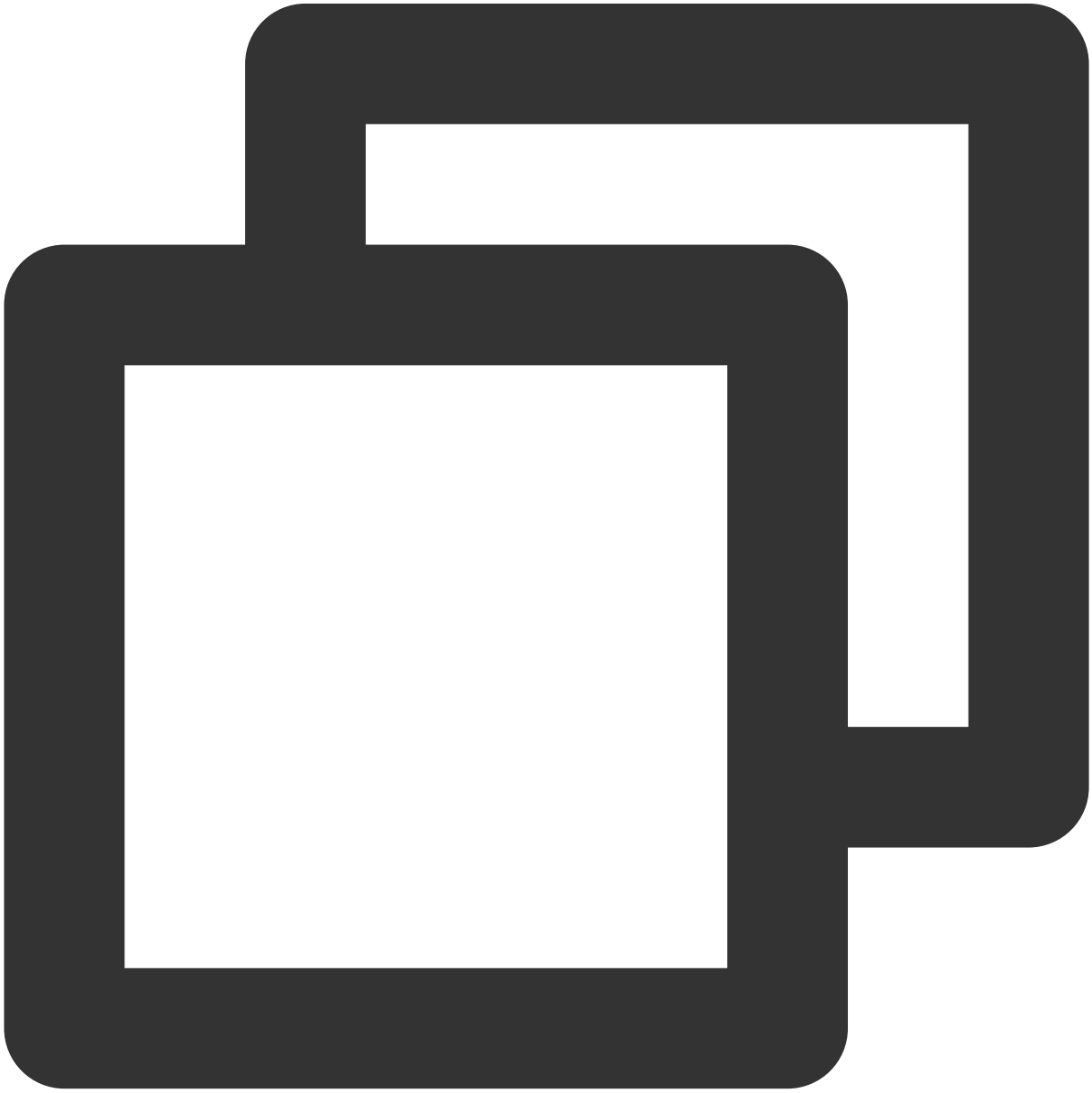
```
> SELECT any_match(array(1, 2, 3), x -> x % 2 == 0);
false
> SELECT any_match(array(2, 4, 8), x -> x % 2 == 0);
true
> SELECT any_match(array(1, null, 3), x -> x % 2 == 0);
NULL
```

Date and Time Functions

Last updated：2024-08-07 17:32:05

DATE

Function statement:



```
DATE(<expr> date|timestamp|string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Force the type to be converted to date.

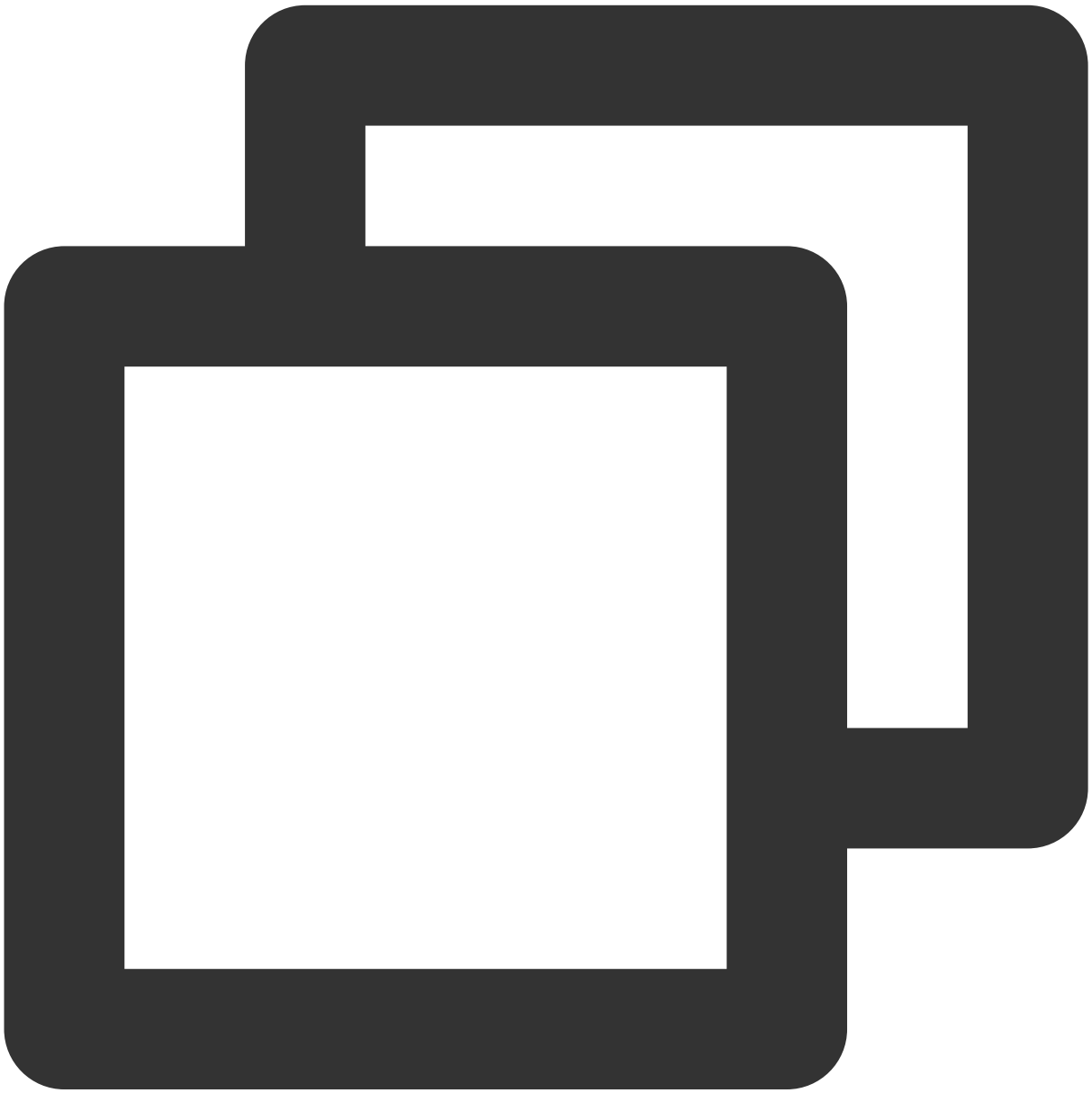
Return type: date

Example:

```
> select date('2022-02-02');
 2022-02-02
```

TIMESTAMP

Function statement:



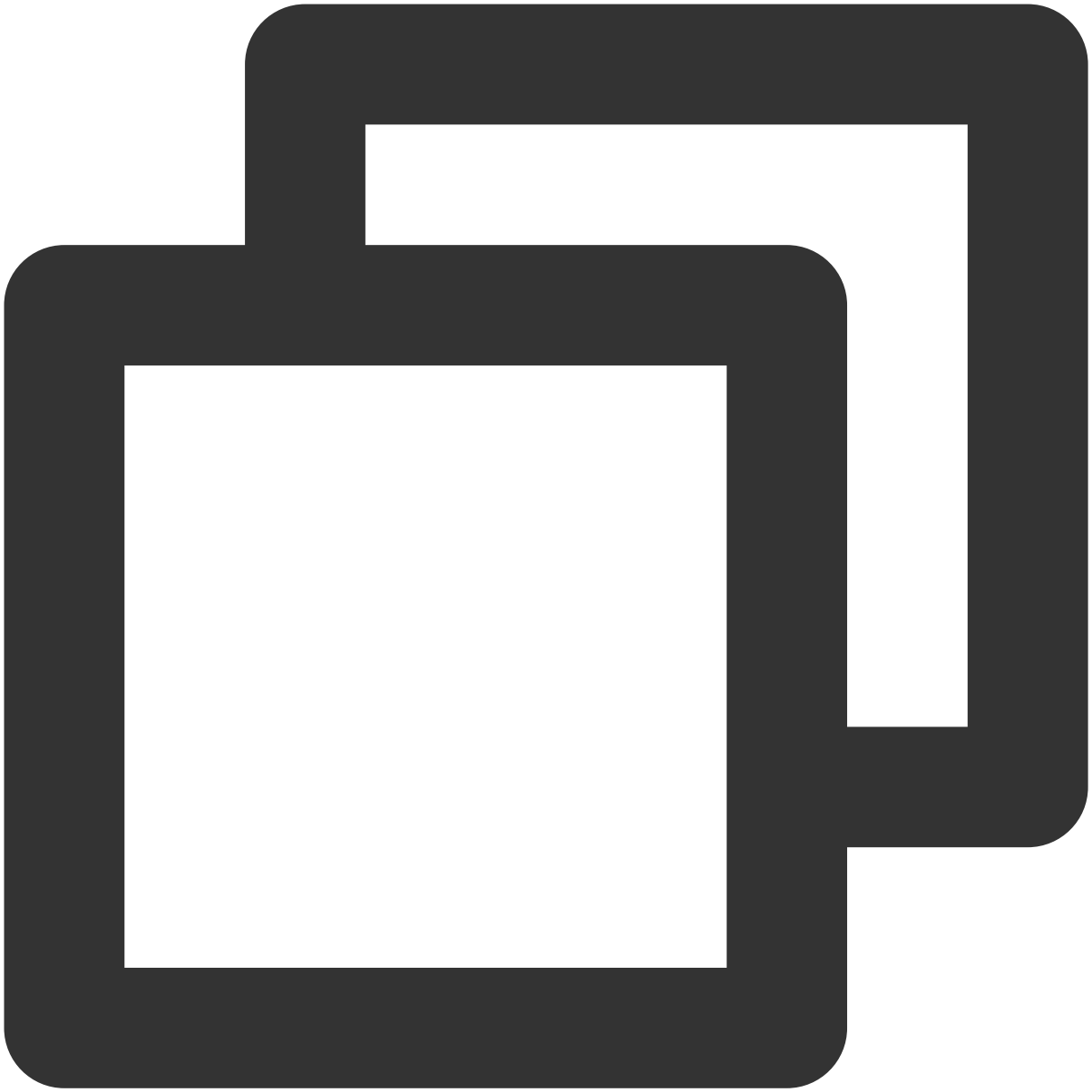
```
TIMESTAMP(<expr> date|timestamp|string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Force the type to be converted to timestamp.

Return type: timestamp

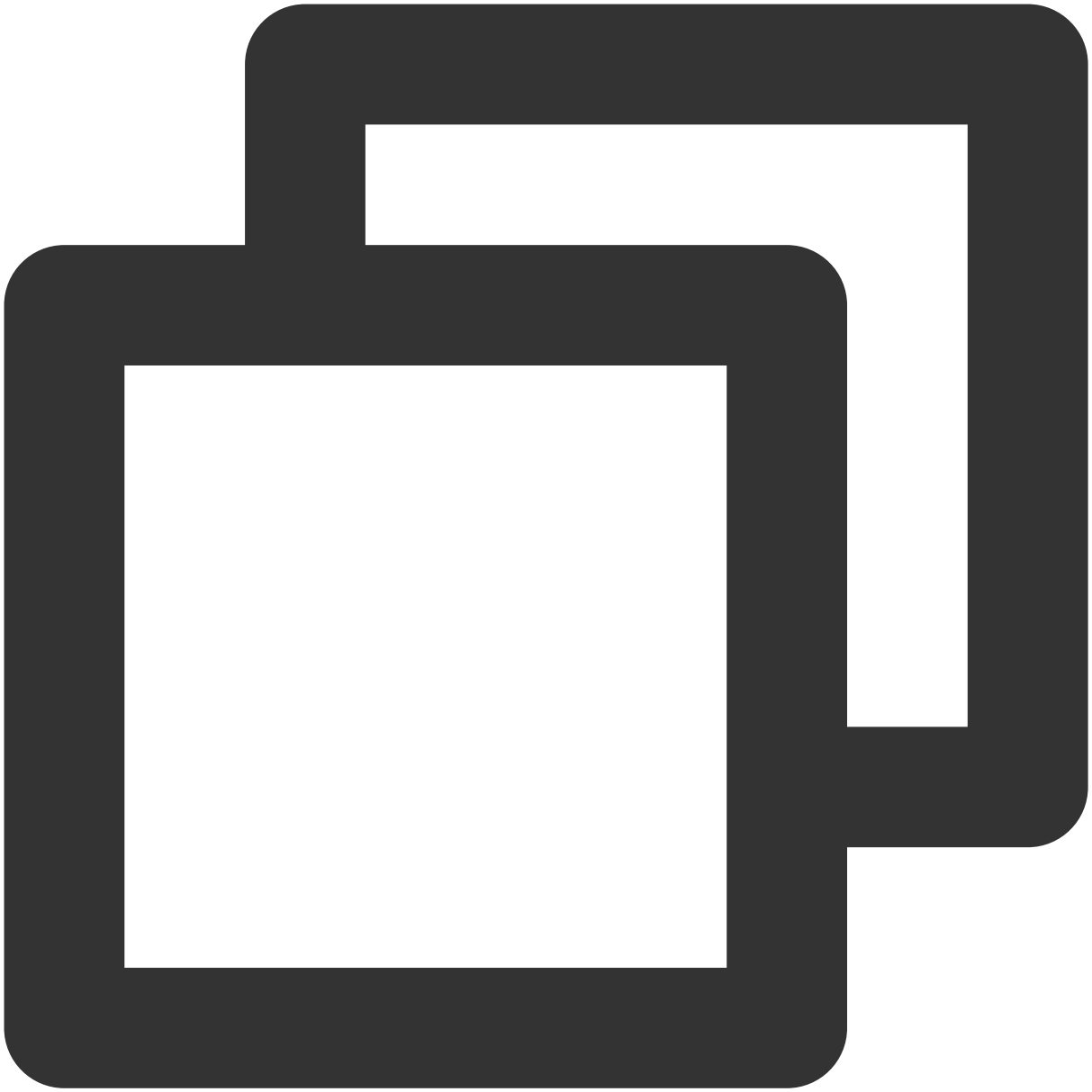
Example:



```
> select timestamp('2022-02-02 11:11:11');
 2022-02-02 11:11:11
```

ADD_MONTHS

Function statement:



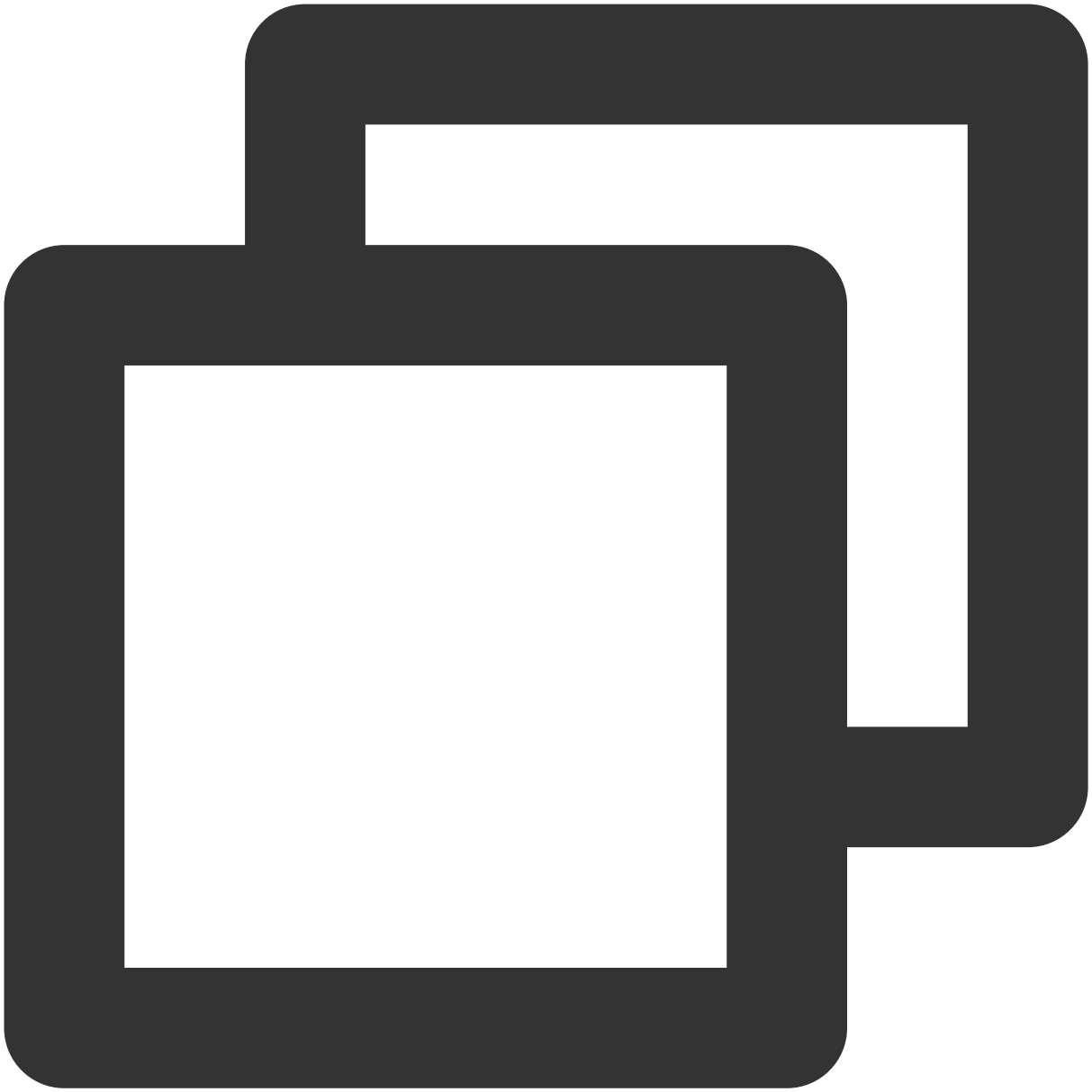
```
ADD_MONTHS(<start_date> date|timestamp|string, <num> integer)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the date that is num months after start_date.

Return type: date

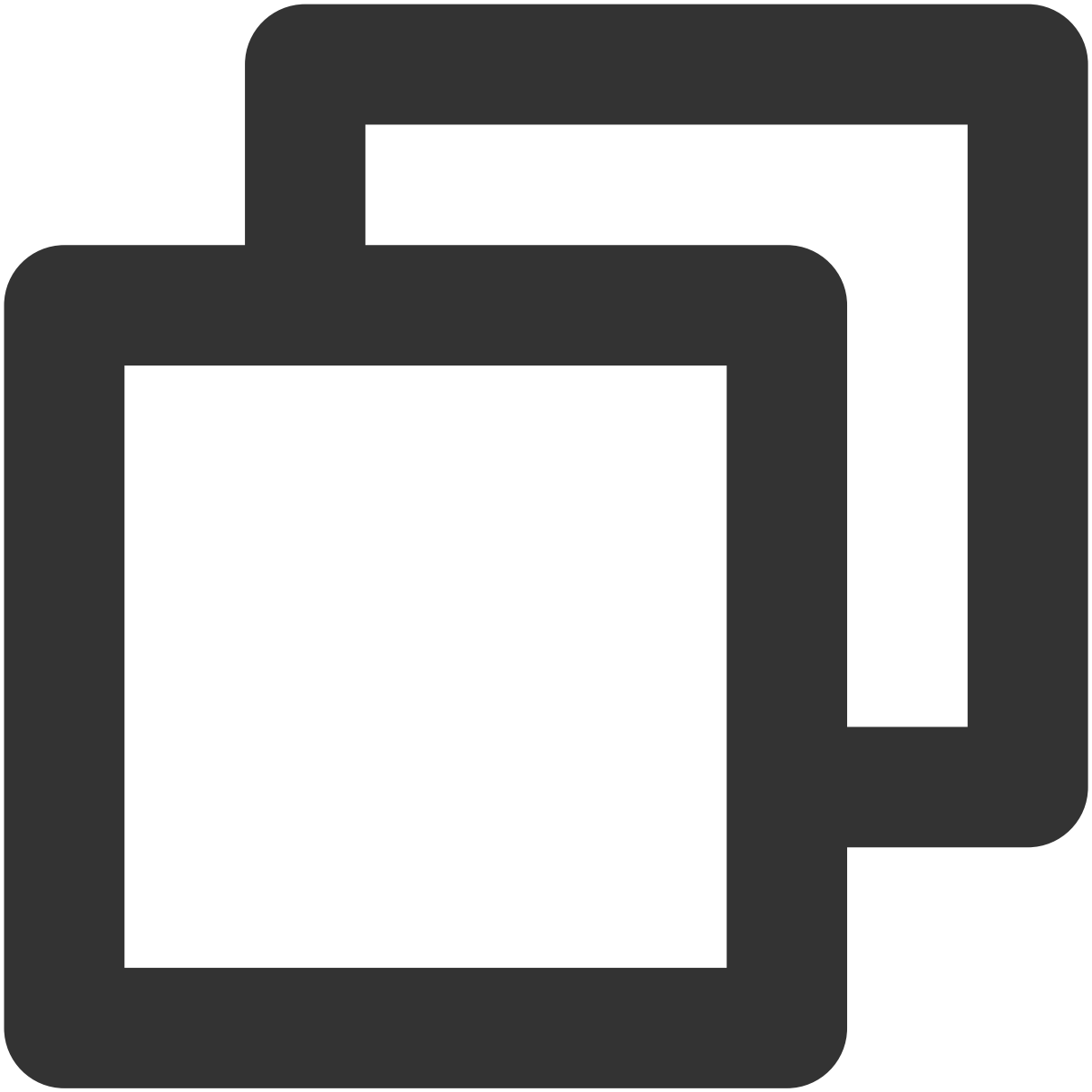
Example:



```
> SELECT add_months('2016-08-31', 1);
 2016-09-30
```

CURRENT_DATE

Function statement:



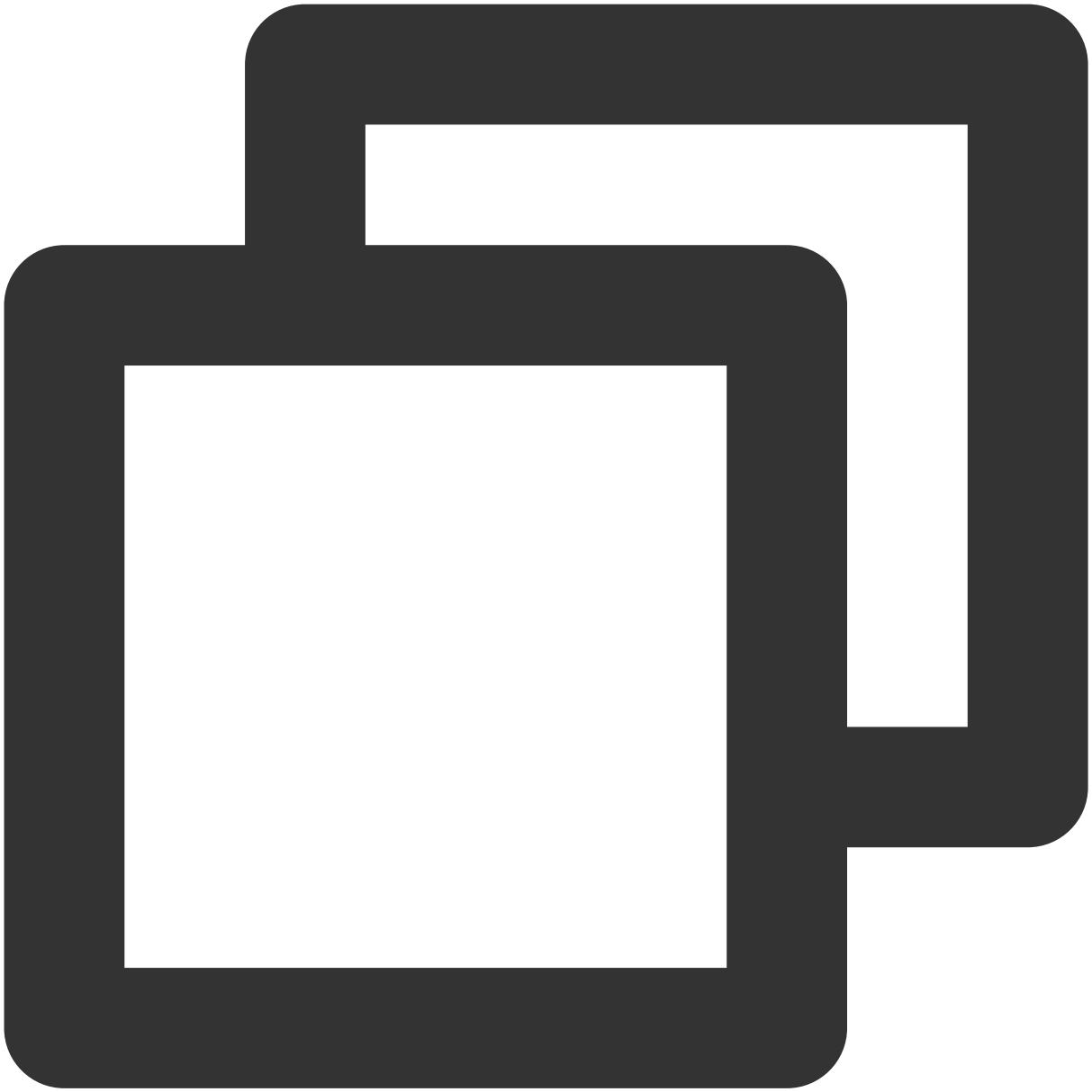
```
CURRENT_DATE
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the current date at the start of the query calculation.

Return type: date

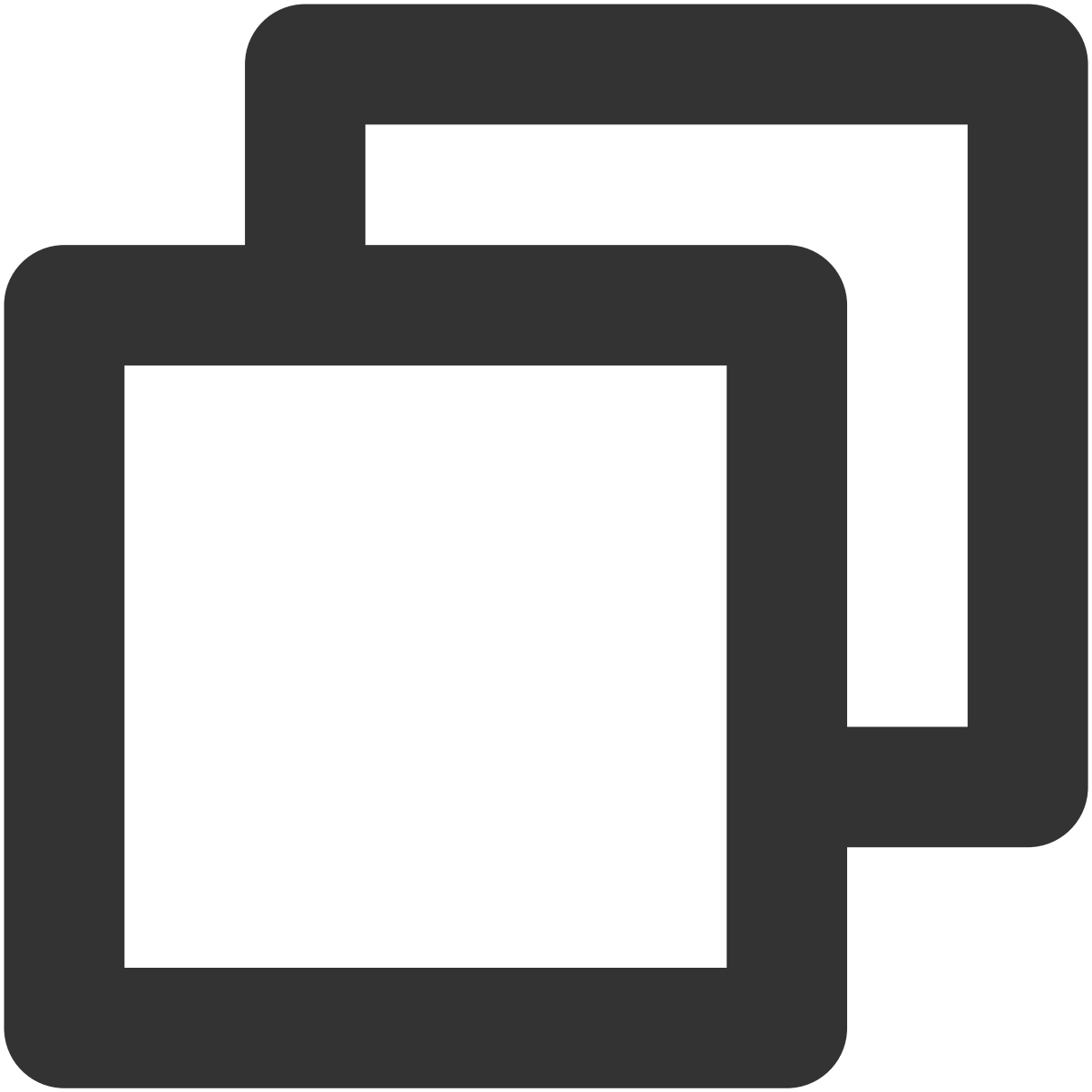
Example:



```
> SELECT CURRENT_DATE;
 2022-07-27
```

CURRENT_TIMESTAMP

Function statement:



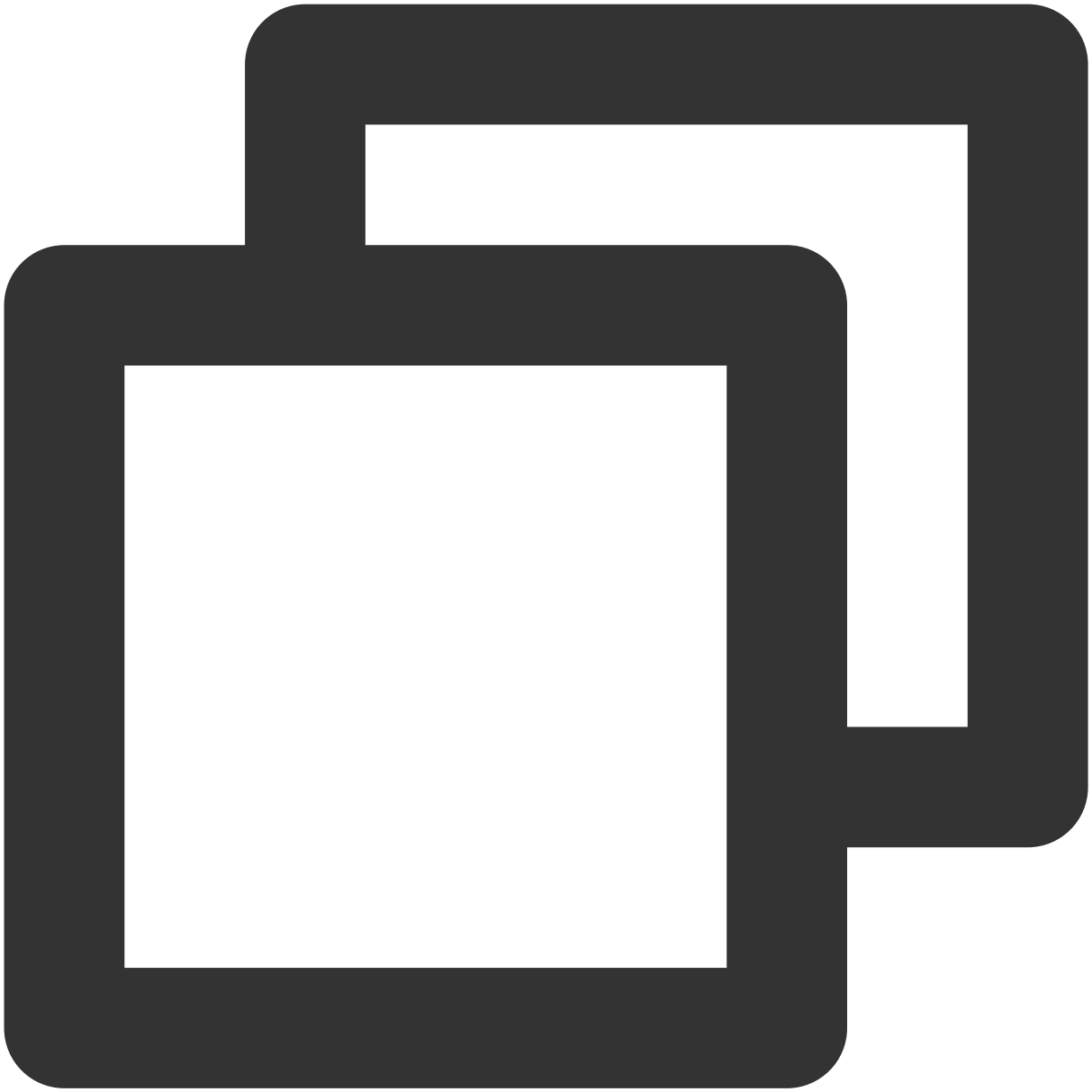
```
CURRENT_TIMESTAMP
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the current timestamp at the start of the query calculation.

Return type: timestamp

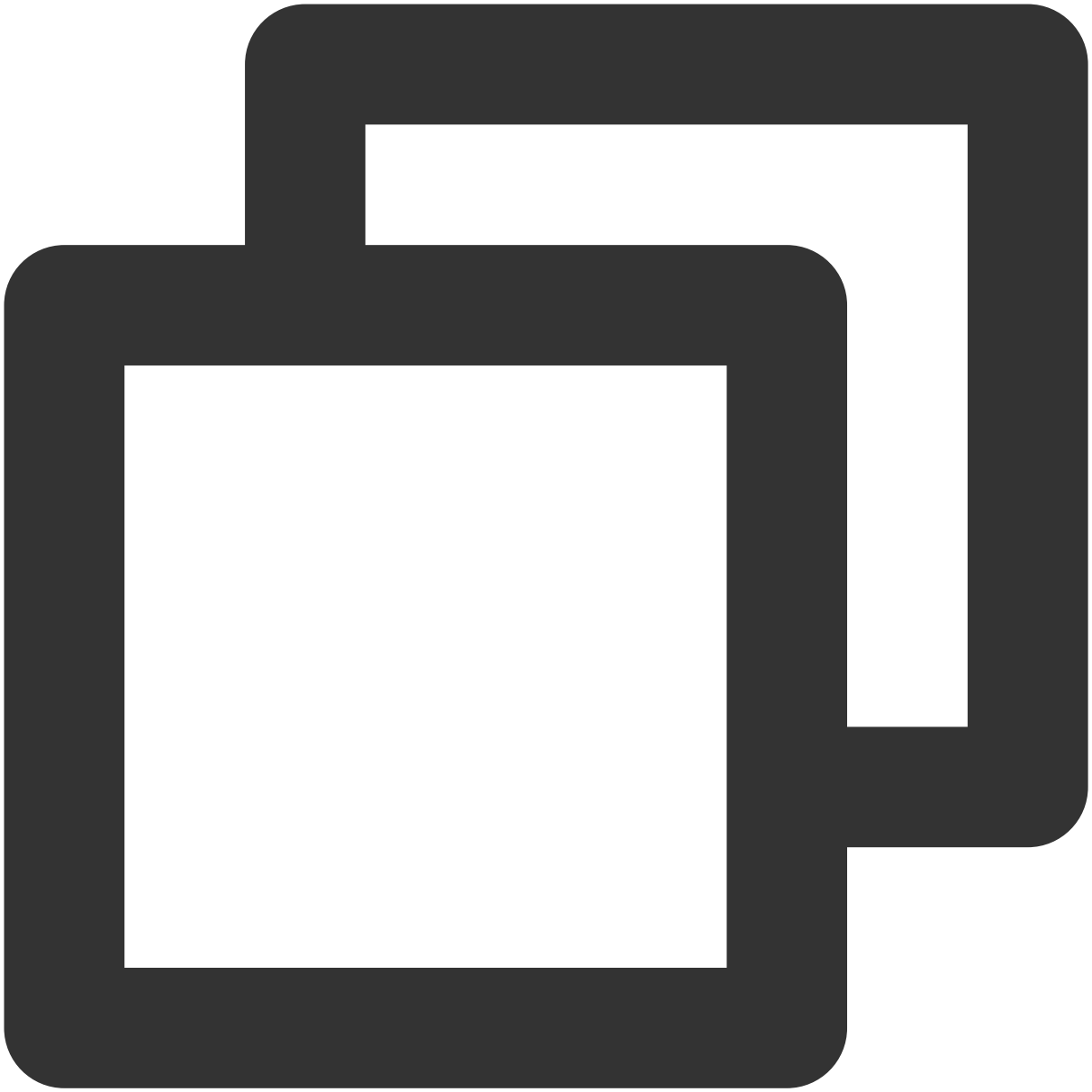
Example:



```
> SELECT CURRENT_TIMESTAMP;
 2022-07-27 18:06:00.632
```

CURRENT_TIMEZONE

Function statement:



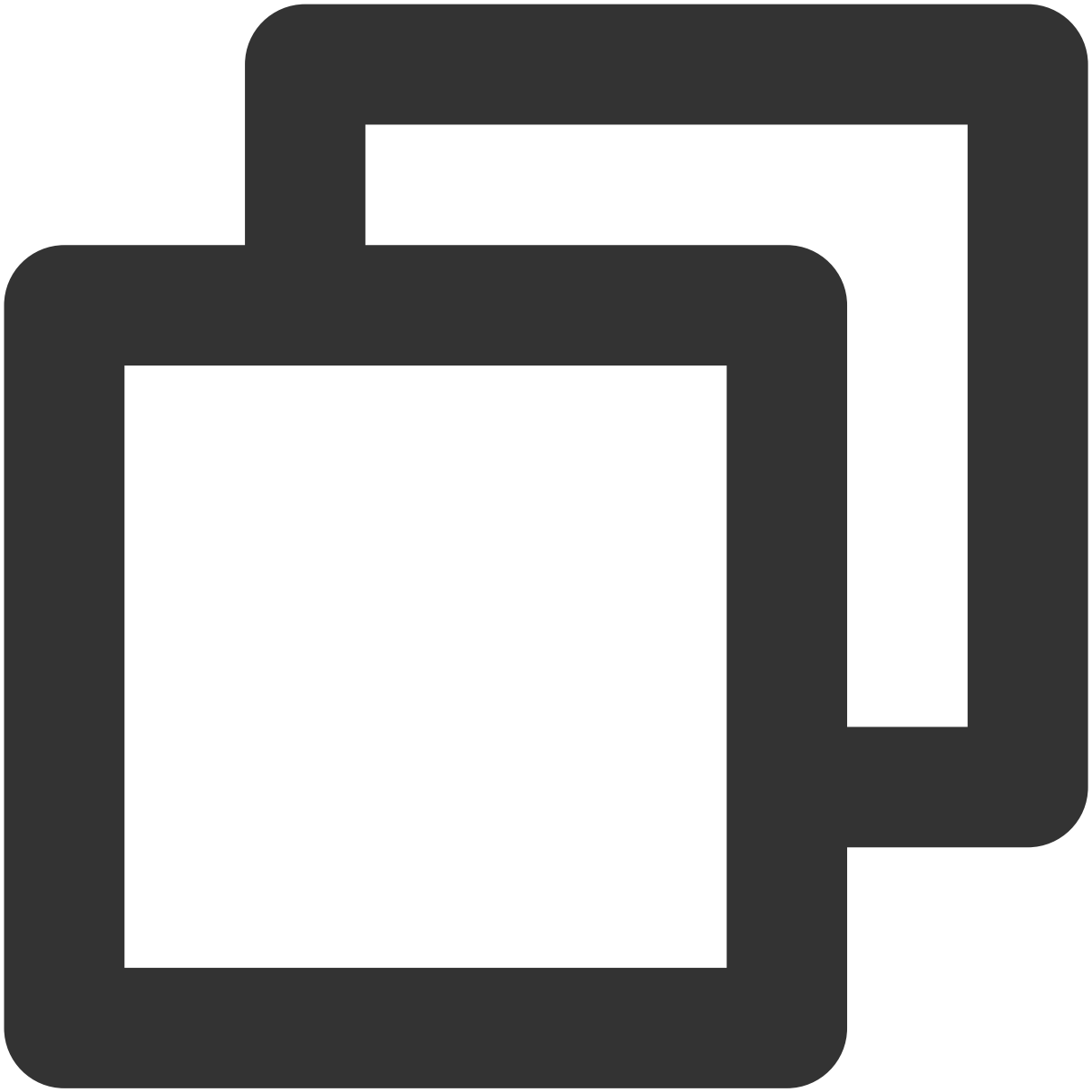
```
CURRENT_TIMEZONE(
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the local timezone of the current session.

Return type: string

Example:



```
> select CURRENT_TIMEZONE();
Asia/Shanghai
```

DATEDIFF

Function statement:

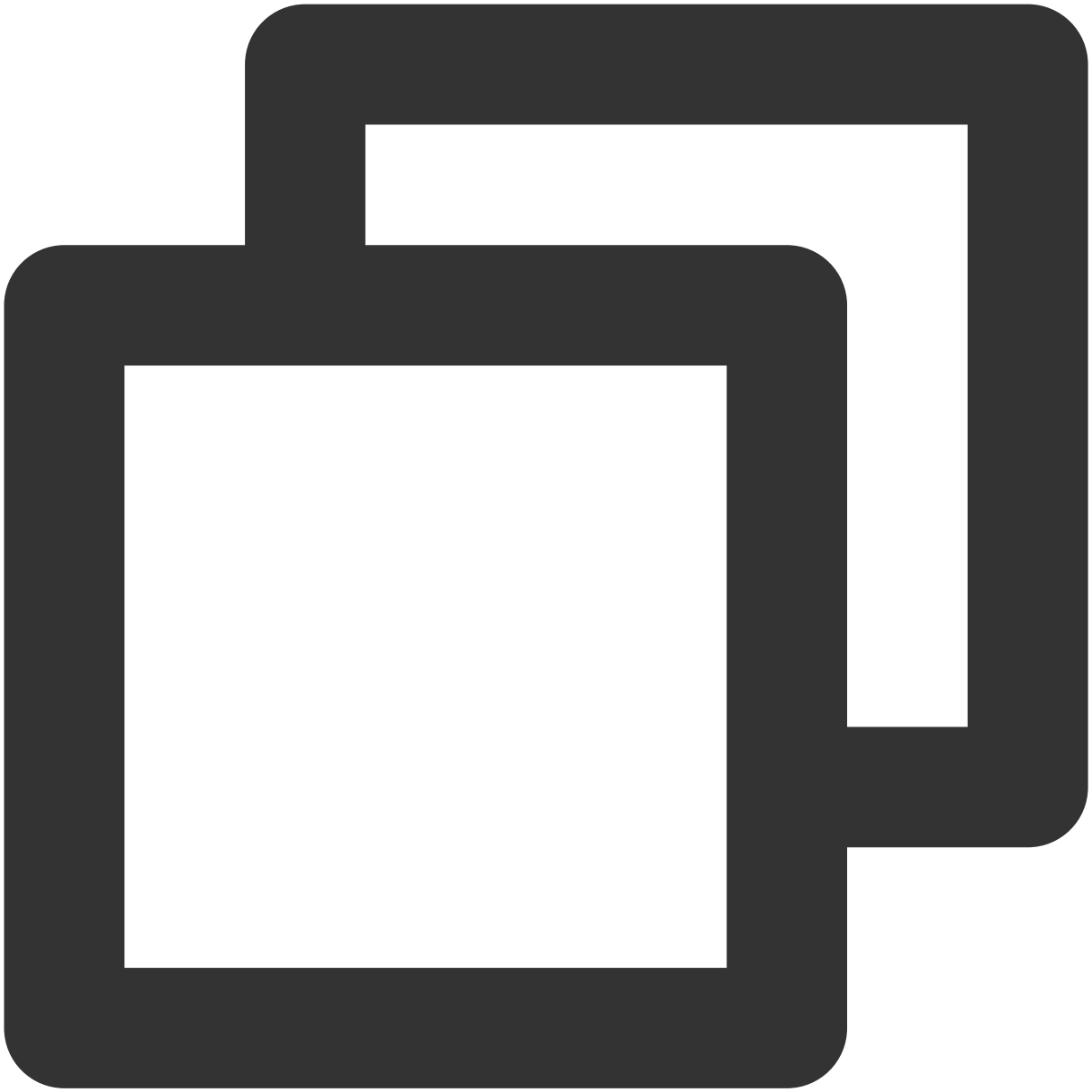
```
DATEDIFF(<end> date|timestamp|string, <start> date|timestamp|string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the number of days from start to end.

Return type: integer

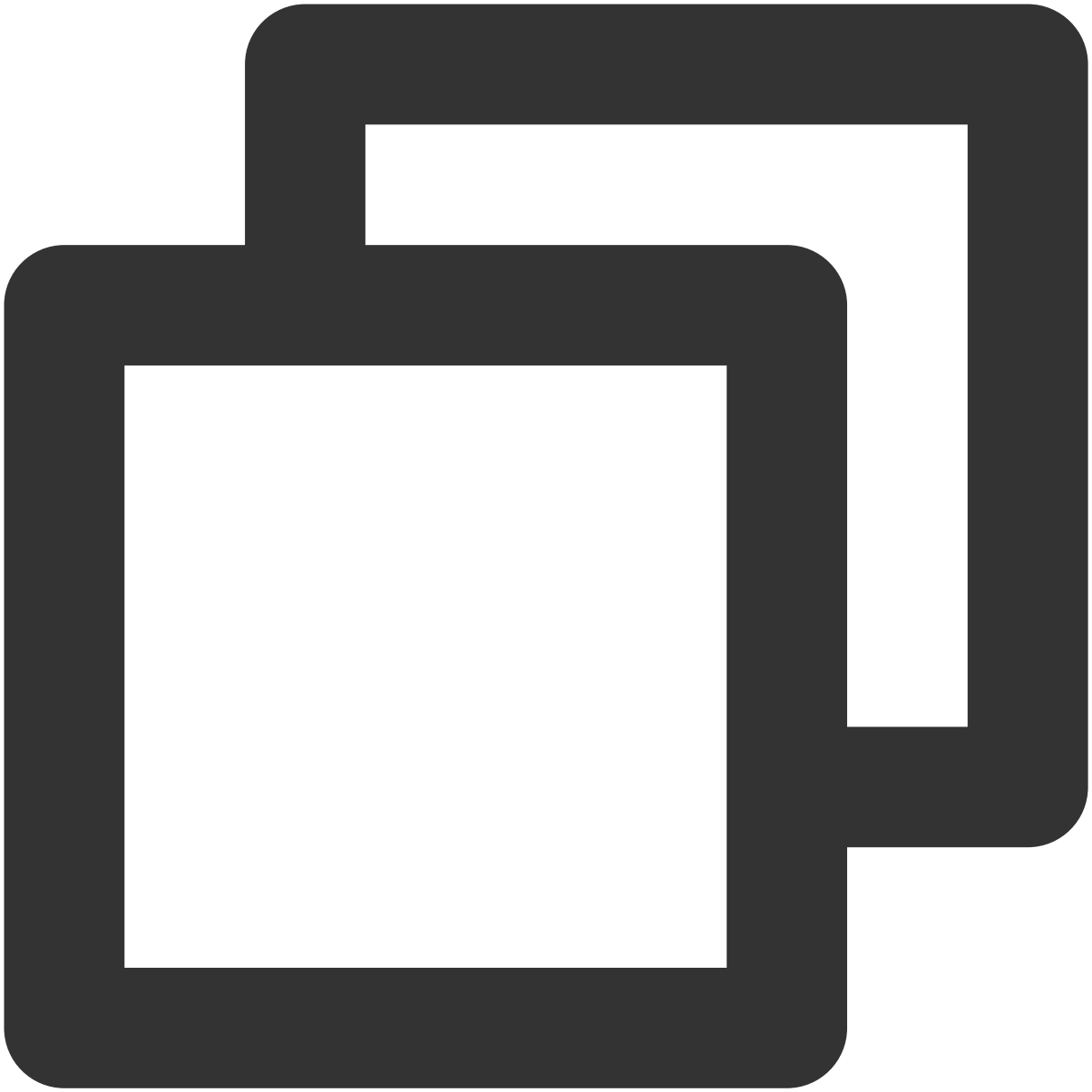
Example:



```
> SELECT datediff('2009-07-31', '2009-07-30');
 1
> SELECT datediff('2009-07-30', '2009-07-31');
 -1
```

DATE_ADD

Function statement:



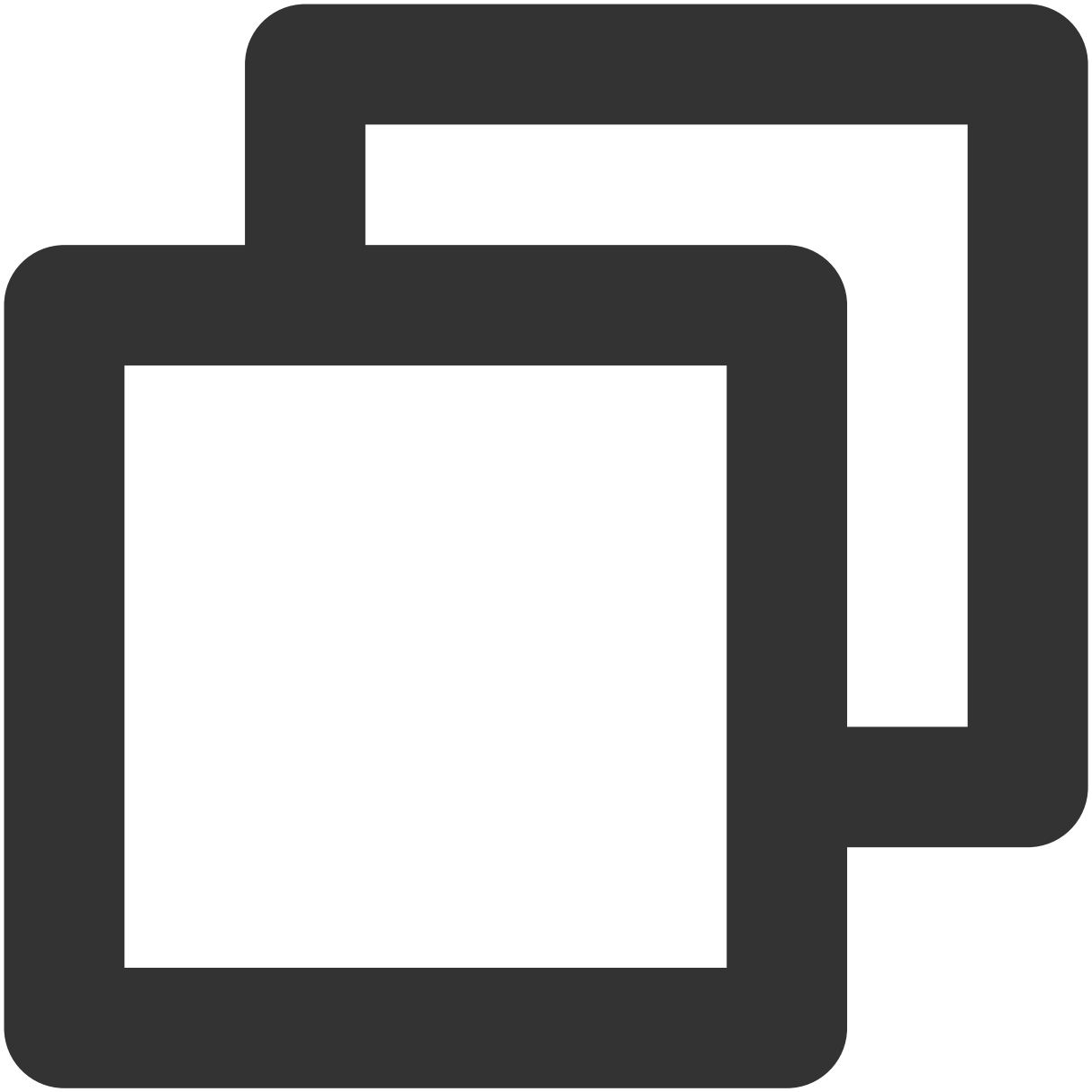
```
DATE_ADD(<start_dates> date|timestamp|string, <num> integer)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the date that is num days after start_date.

Return type: date

Example:



```
> SELECT date_add('2016-07-30', 1);
 2016-07-31
```

DATE_FORMAT

Function statement:

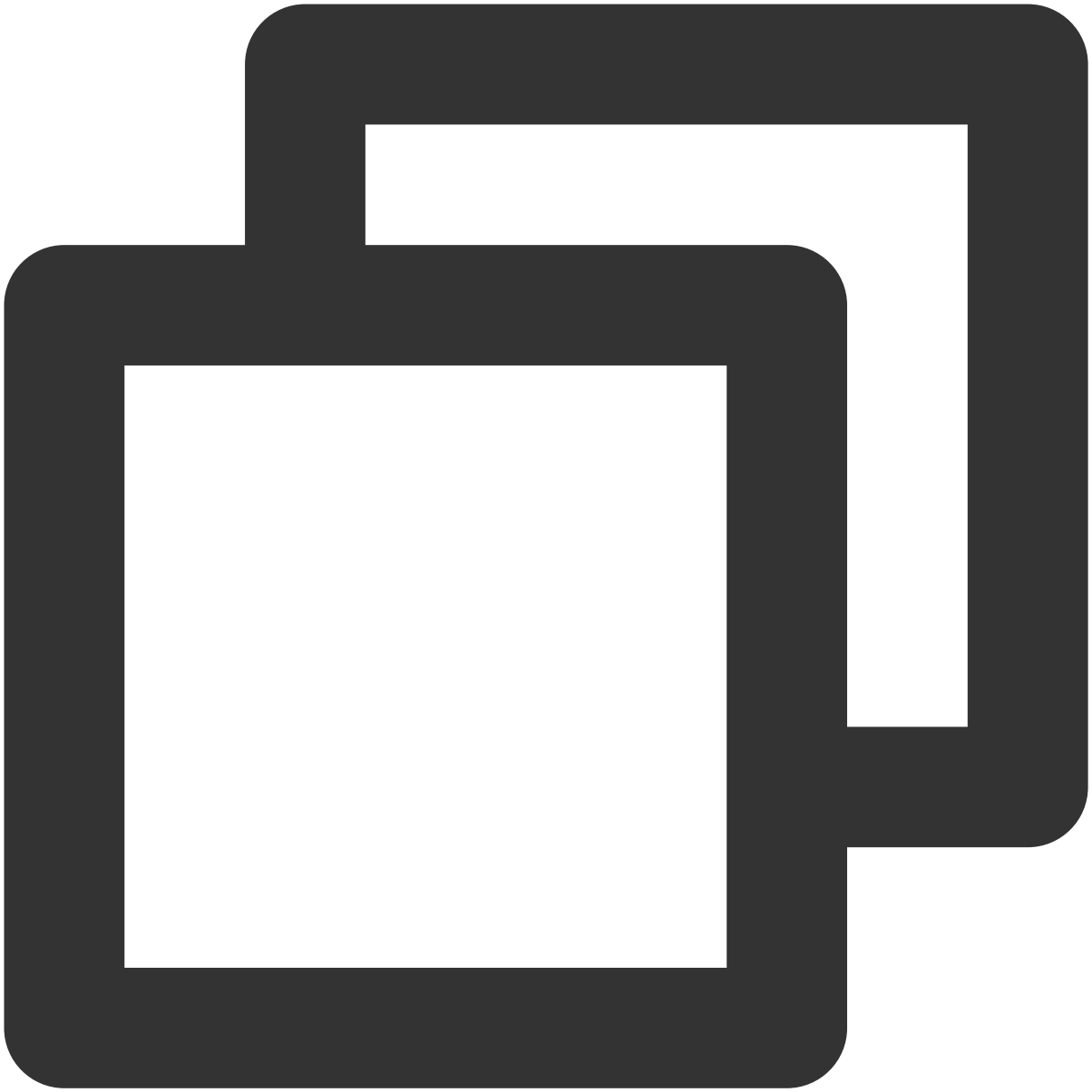
```
DATE_FORMAT(<ts> date|timestamp|string, <format> string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Convert a timestamp to a string value in the specified date format.

Return type: string

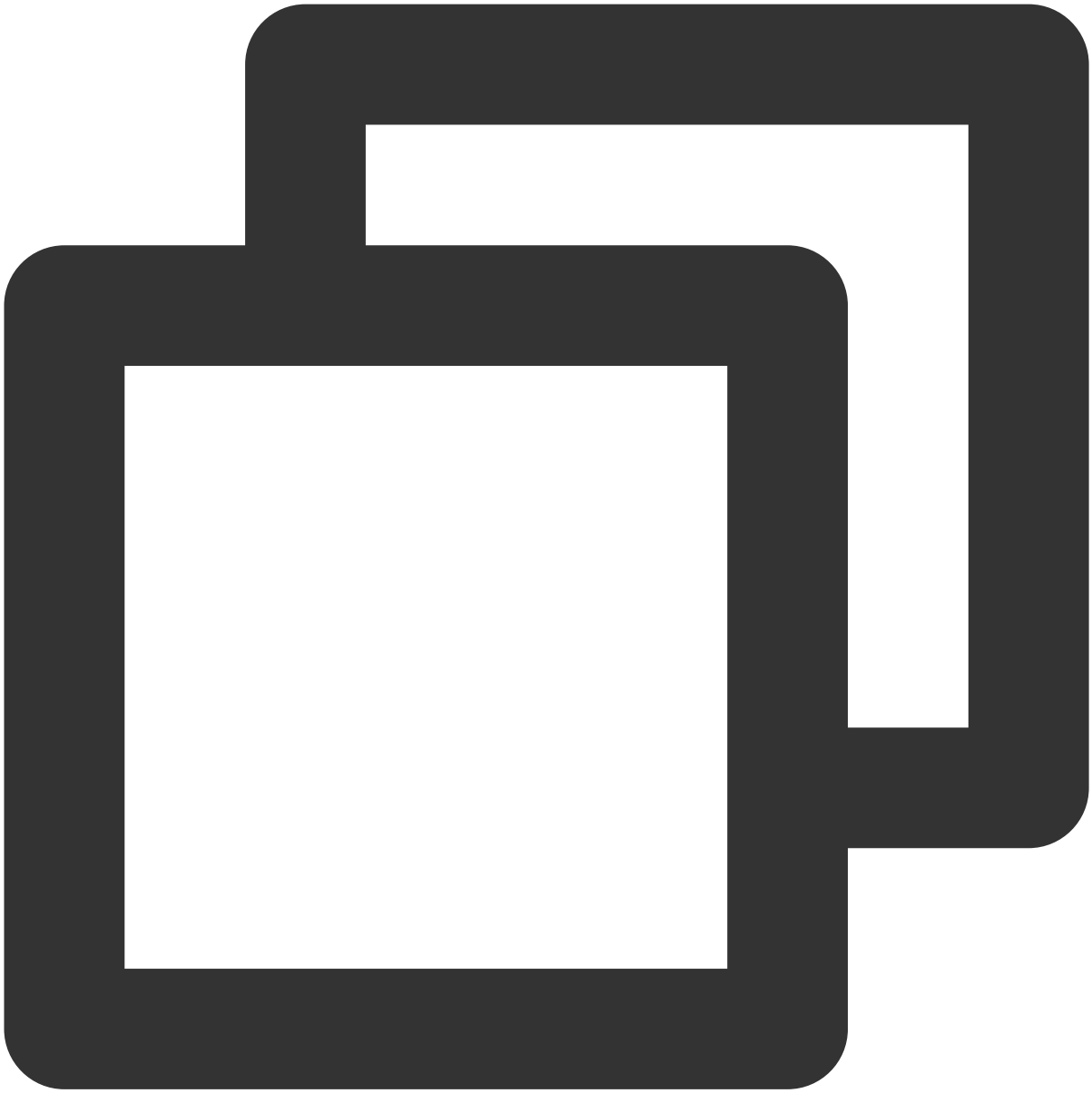
Example:



```
> SELECT date_format('2016-04-08', 'y');
 2016
```

DATE_SUB

Function statement:



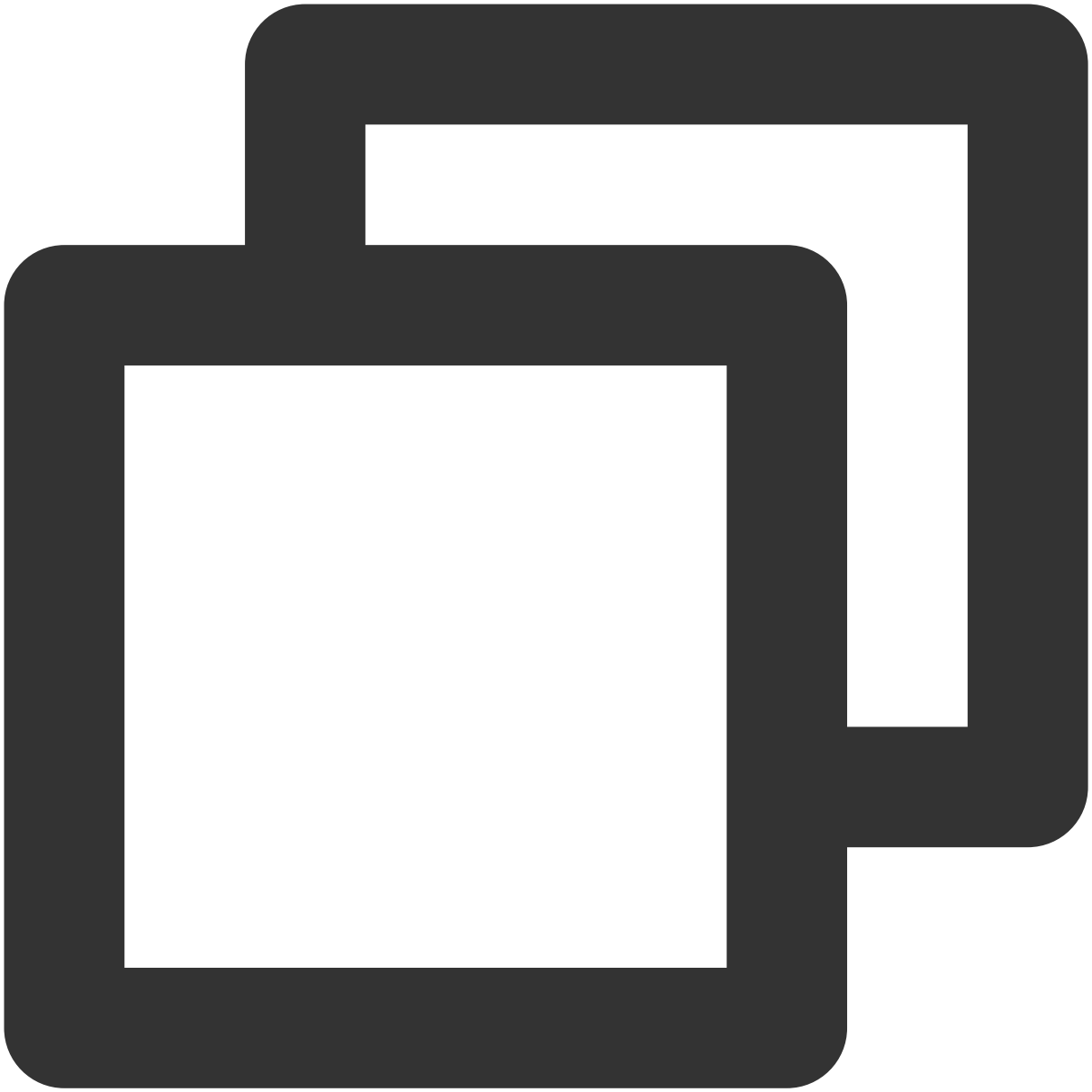
```
DATE_SUB(<start_date> date|timestamp|string, <num> integer)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the date that is num days before start_date.

Return type: date

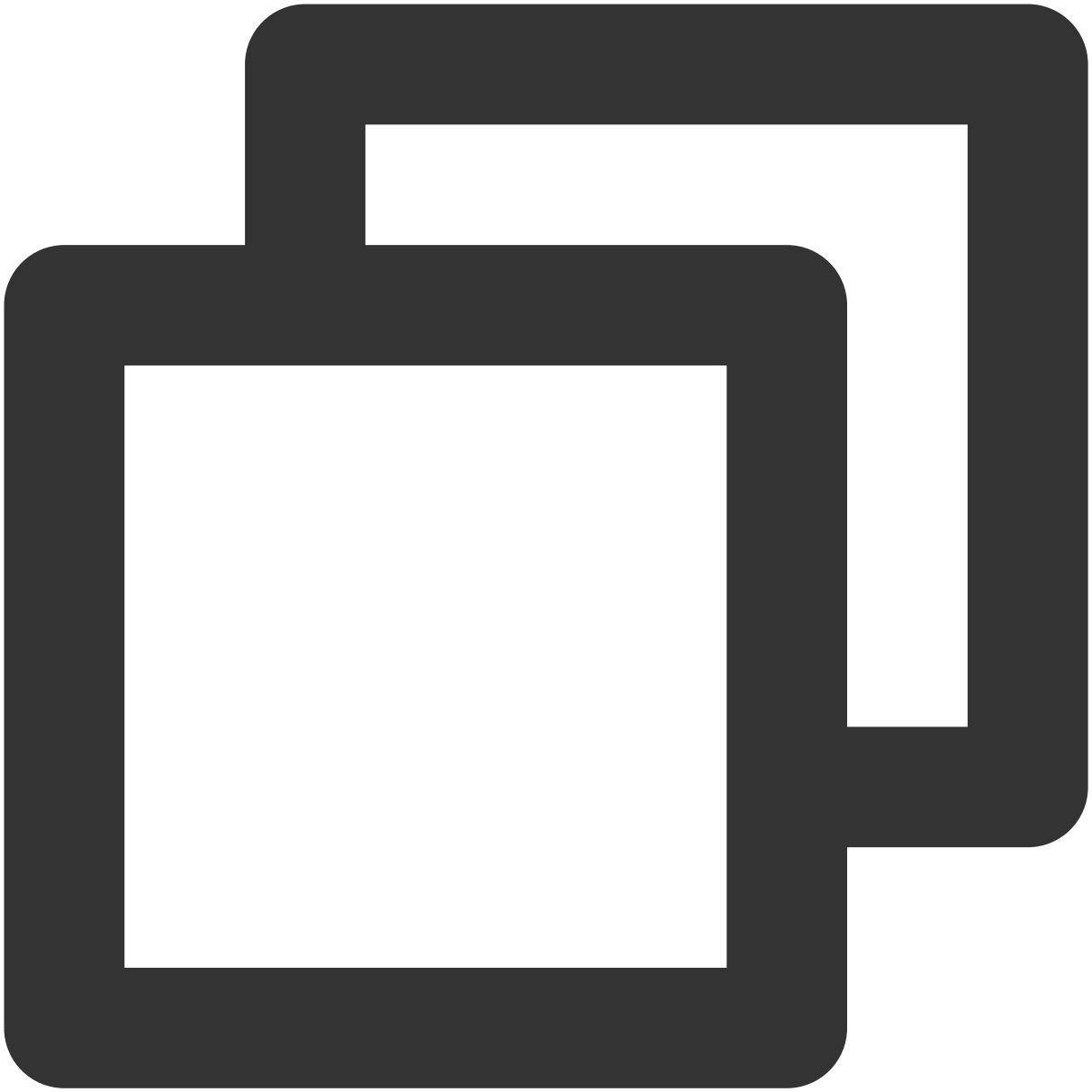
Example:



```
> SELECT date_sub('2016-07-30', 1);
 2016-07-29
```

DAY

Function statement:



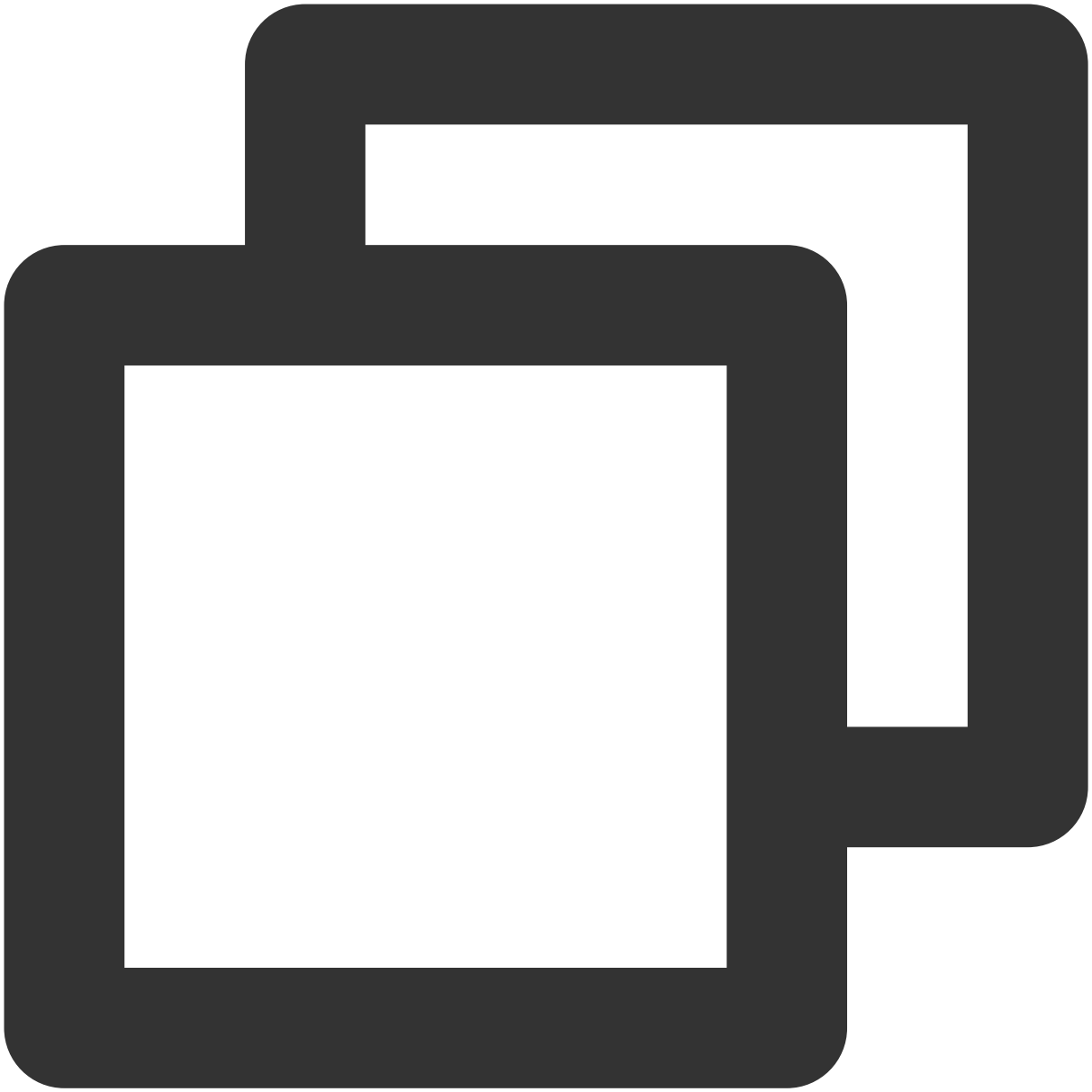
```
DAY(<d> date|timestamp|string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Calculate which day of the month the date/timestamp d is.

Return type: integer

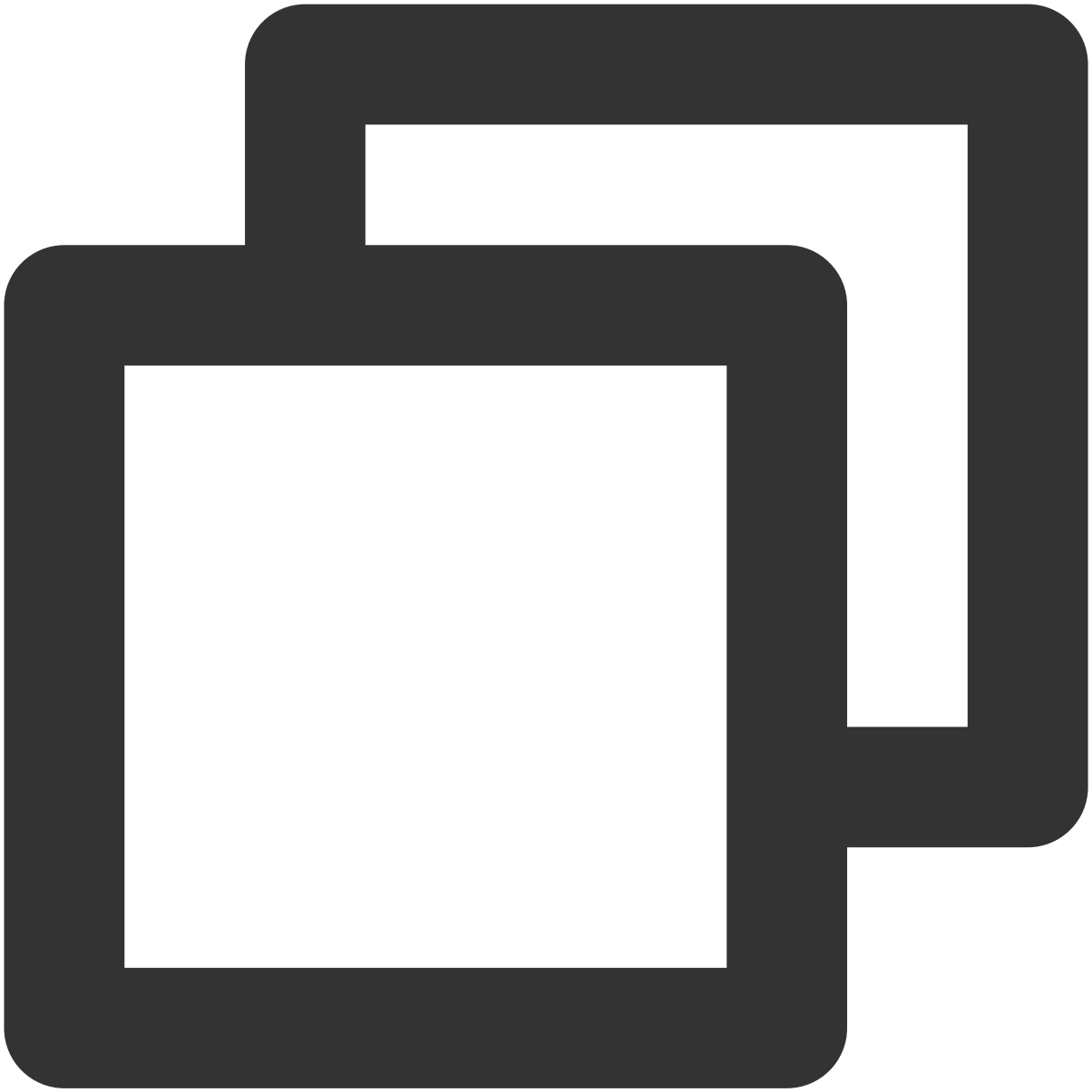
Example:



```
> SELECT day('2009-07-30');
 30
```

DAYOFYEAR

Function statement:



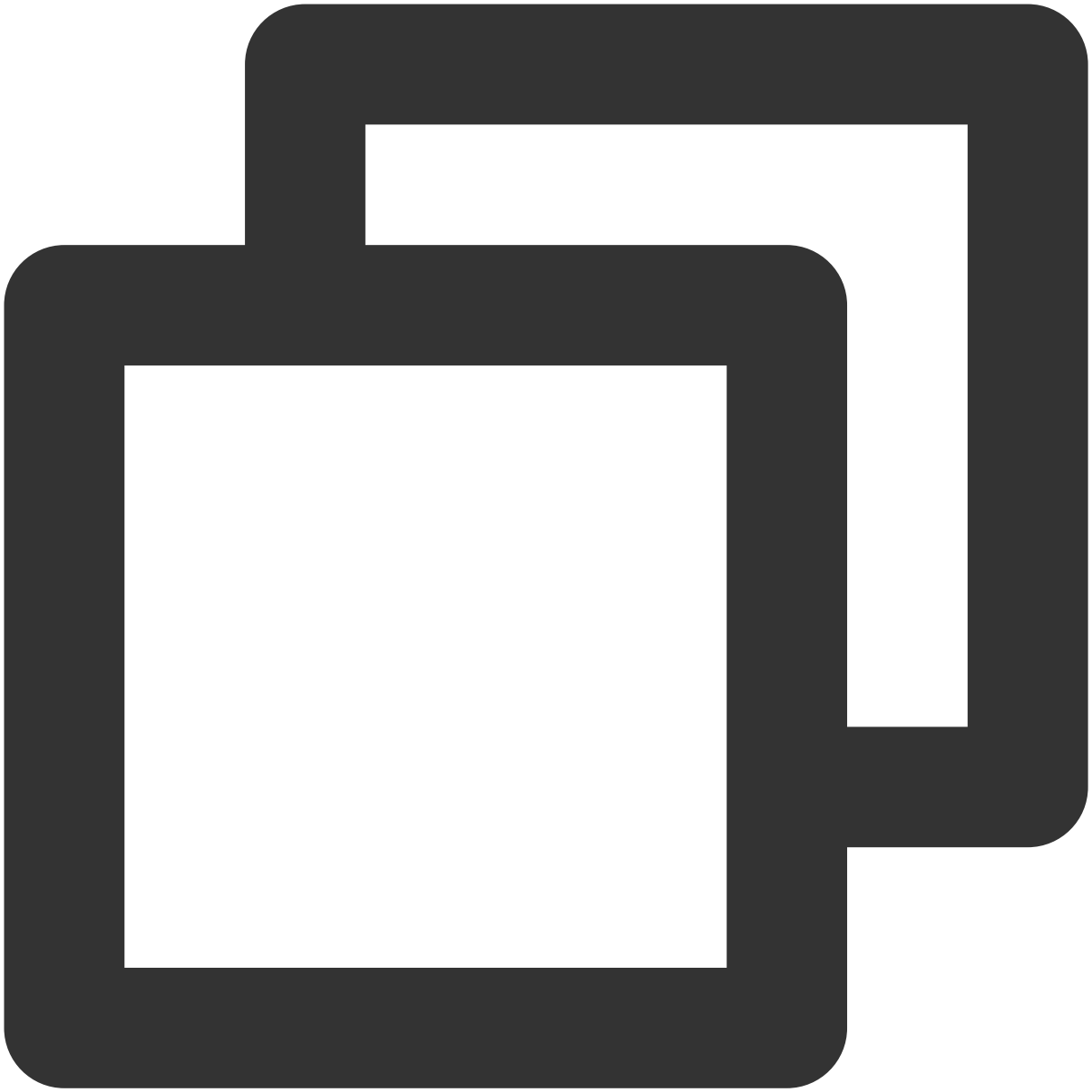
```
DAYOFYEAR(<d> date|timestamp|string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Calculate which day of the year the date/timestamp d is.

Return type: integer

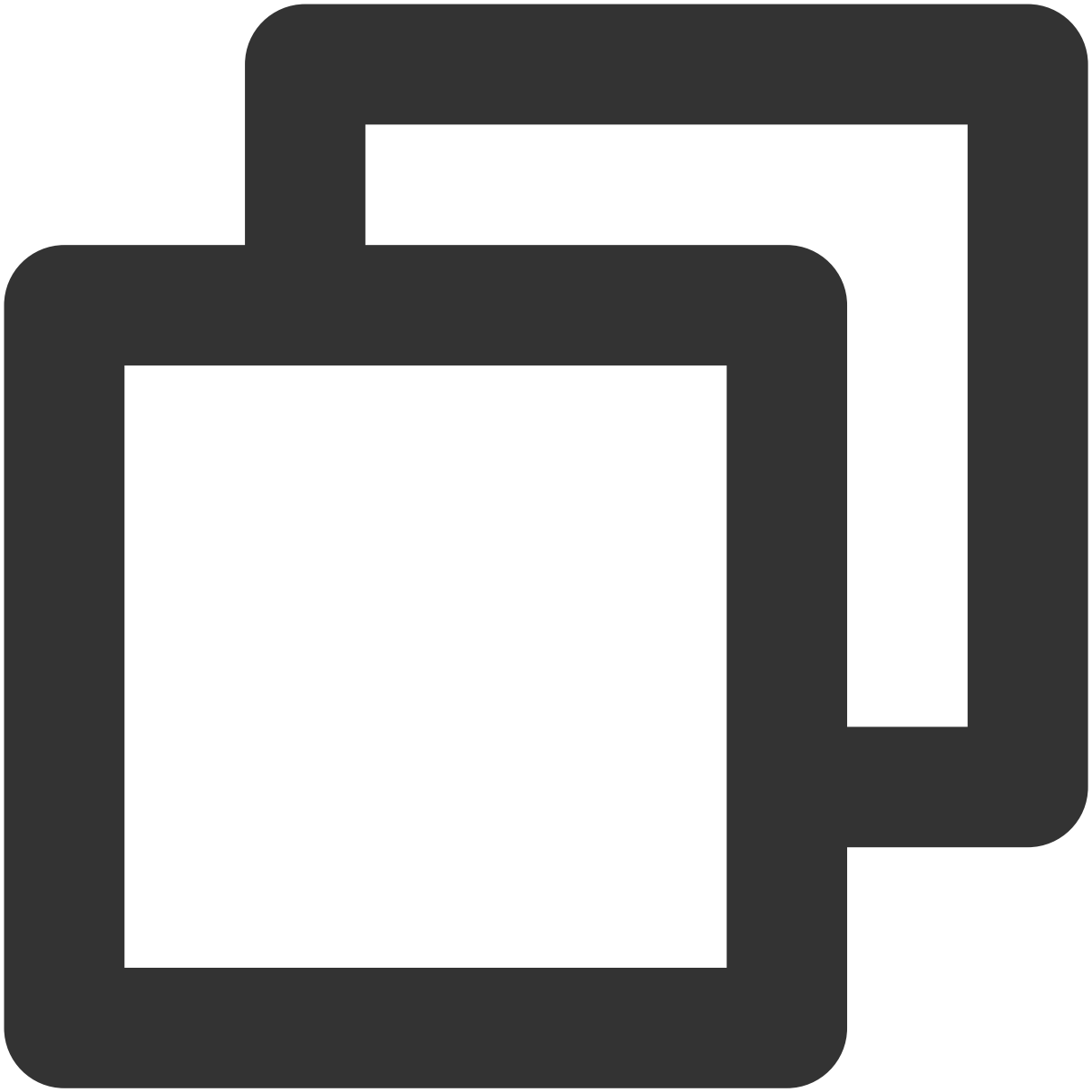
Example:



```
> SELECT dayofyear('2016-04-09');
 100
```

DAYOFMONTH

Function statement:



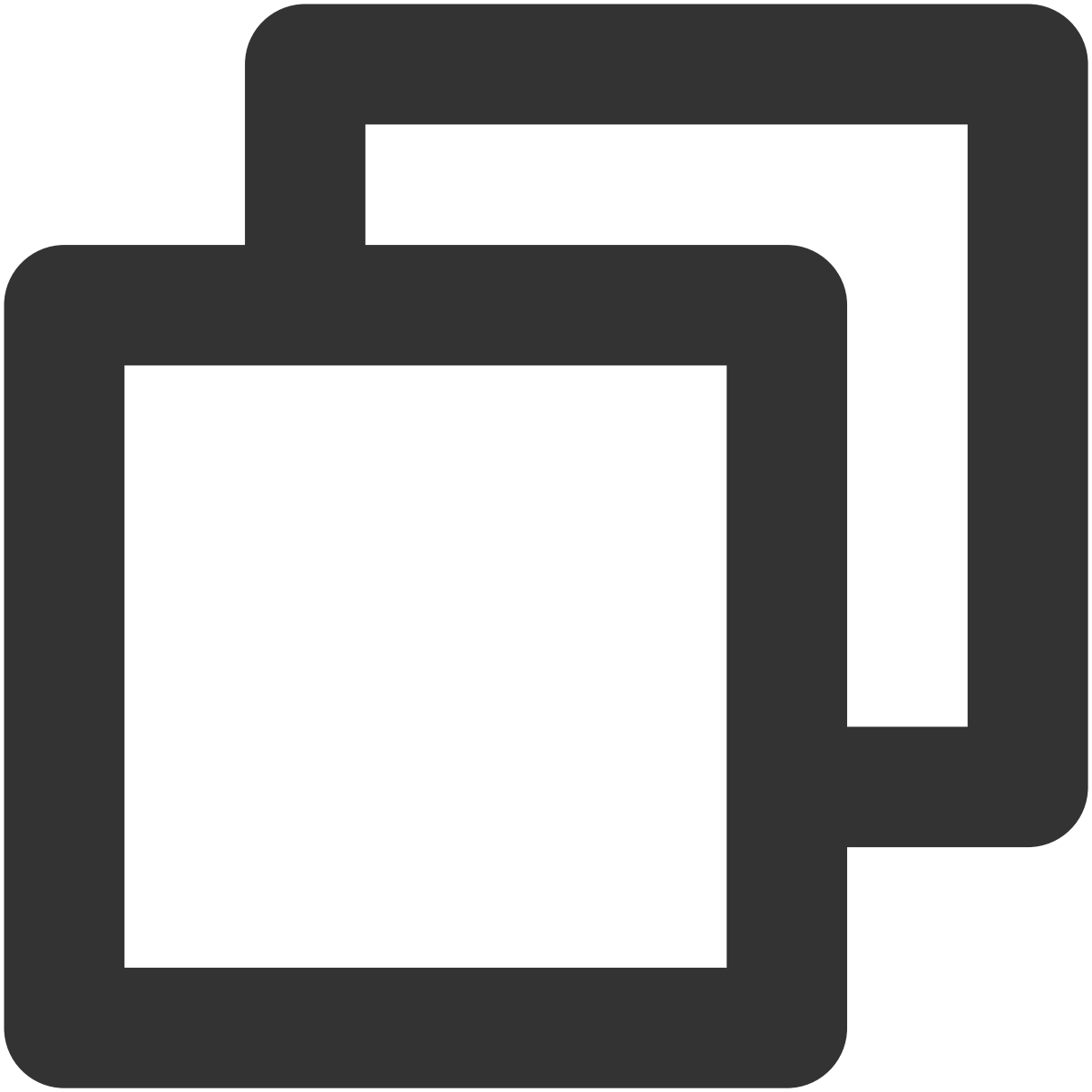
```
DAYOFMONTH
```

Supported engines: SparkSQL and Presto

Usage instructions: Calculate which day of the month the date/timestamp d is.

Return type: integer

Example:



```
> SELECT dayofmonth('2009-07-30');
 30
```

FROM_UNIXTIME

Function statement:

```
FROM_UNIXTIME(<unix_time> bigint[, <fmt> string])
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the date/time represented by unix_time in the format fmt. If fmt is omitted, the format 'yyyy-MM-dd HH:mm:ss' is used.

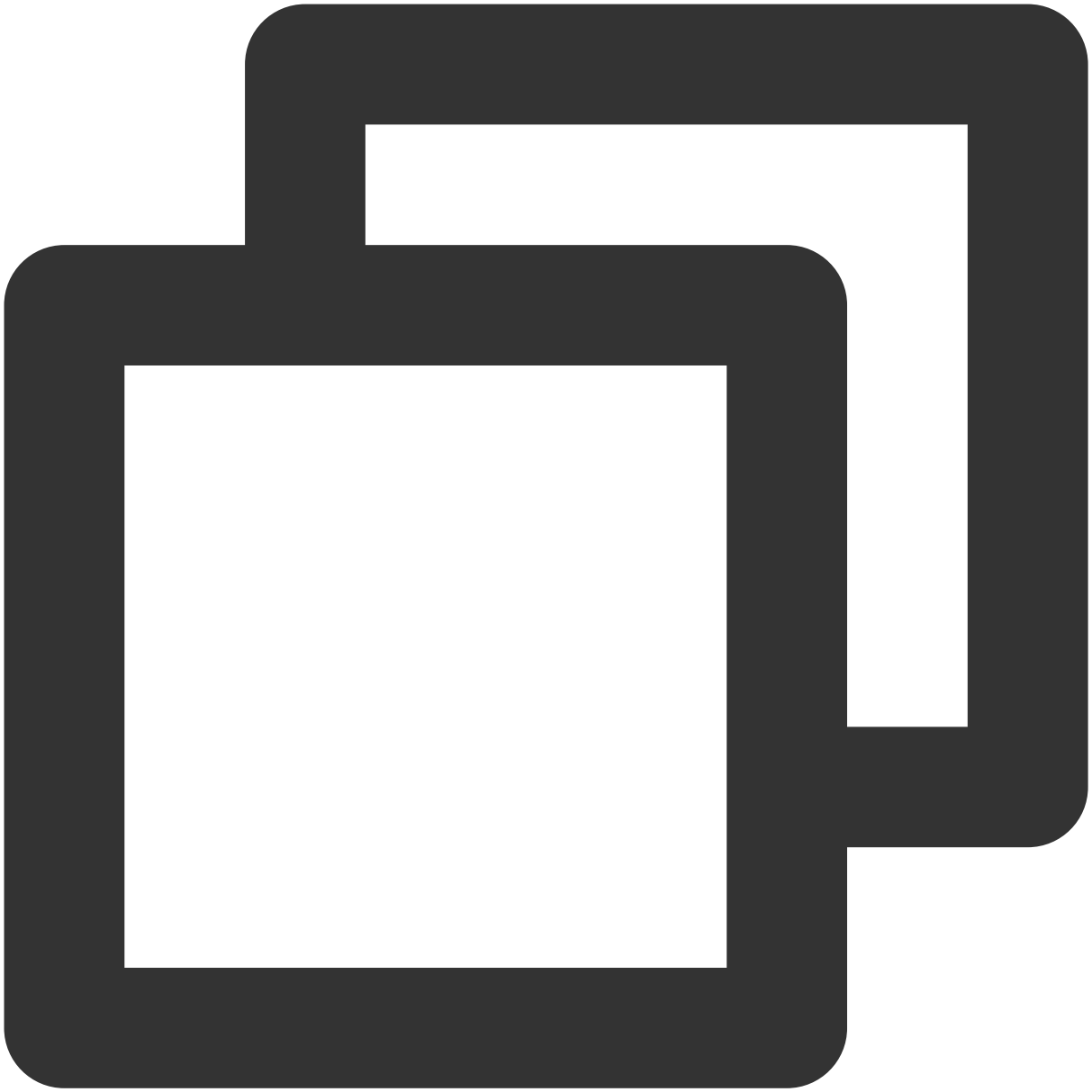
Return type: string

Example:

```
> SELECT from_unixtime(0, 'yyyy-MM-dd HH:mm:ss');
 1969-12-31 16:00:00
> SELECT from_unixtime(0);
 1969-12-31 16:00:00
```

FROM_UTC_TIMESTAMP

Function statement:



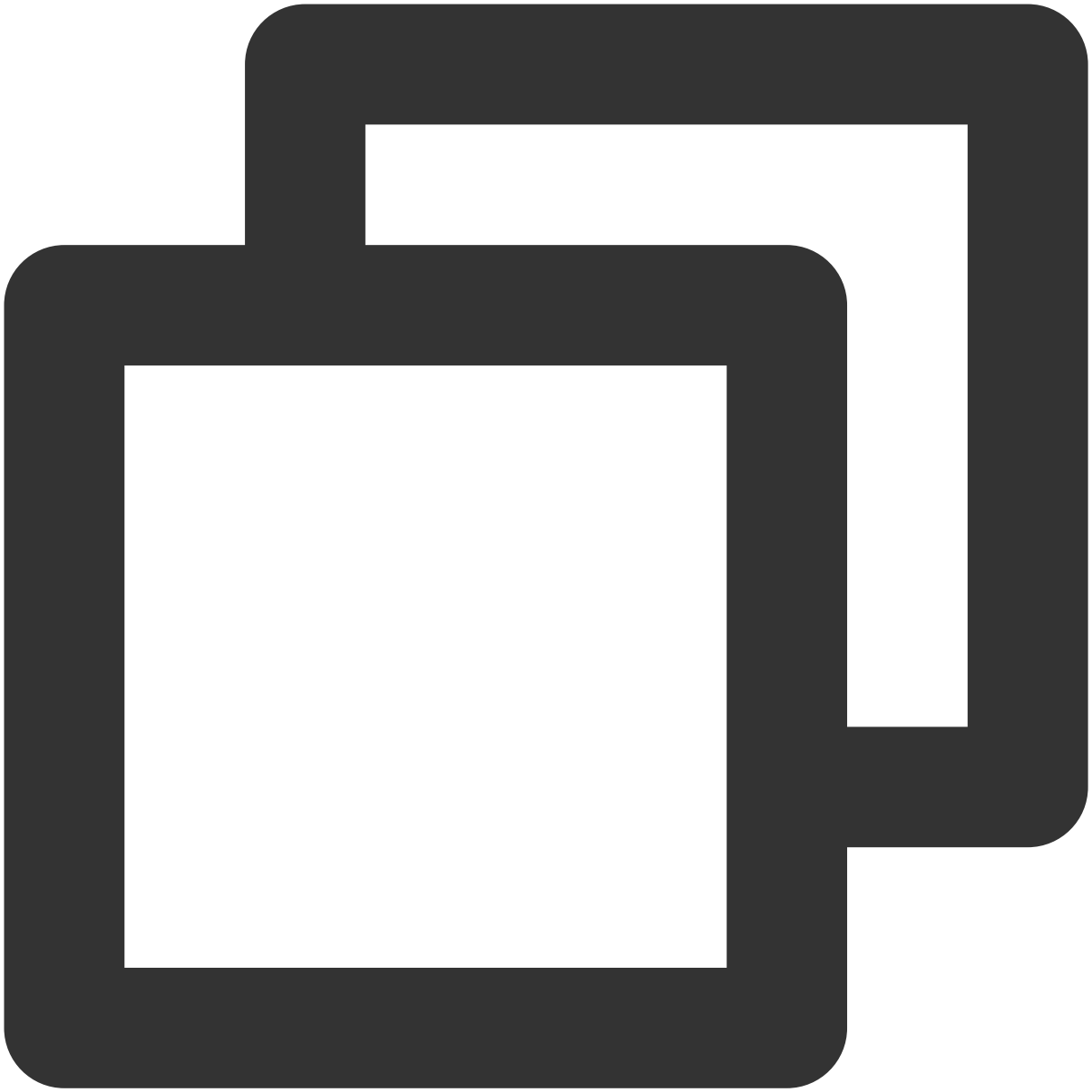
```
FROM_UTC_TIMESTAMP(<ts> timestamp, <timezone> string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Given a UTC timestamp, present the time as a timestamp in the specified timezone.

Return type: timestamp

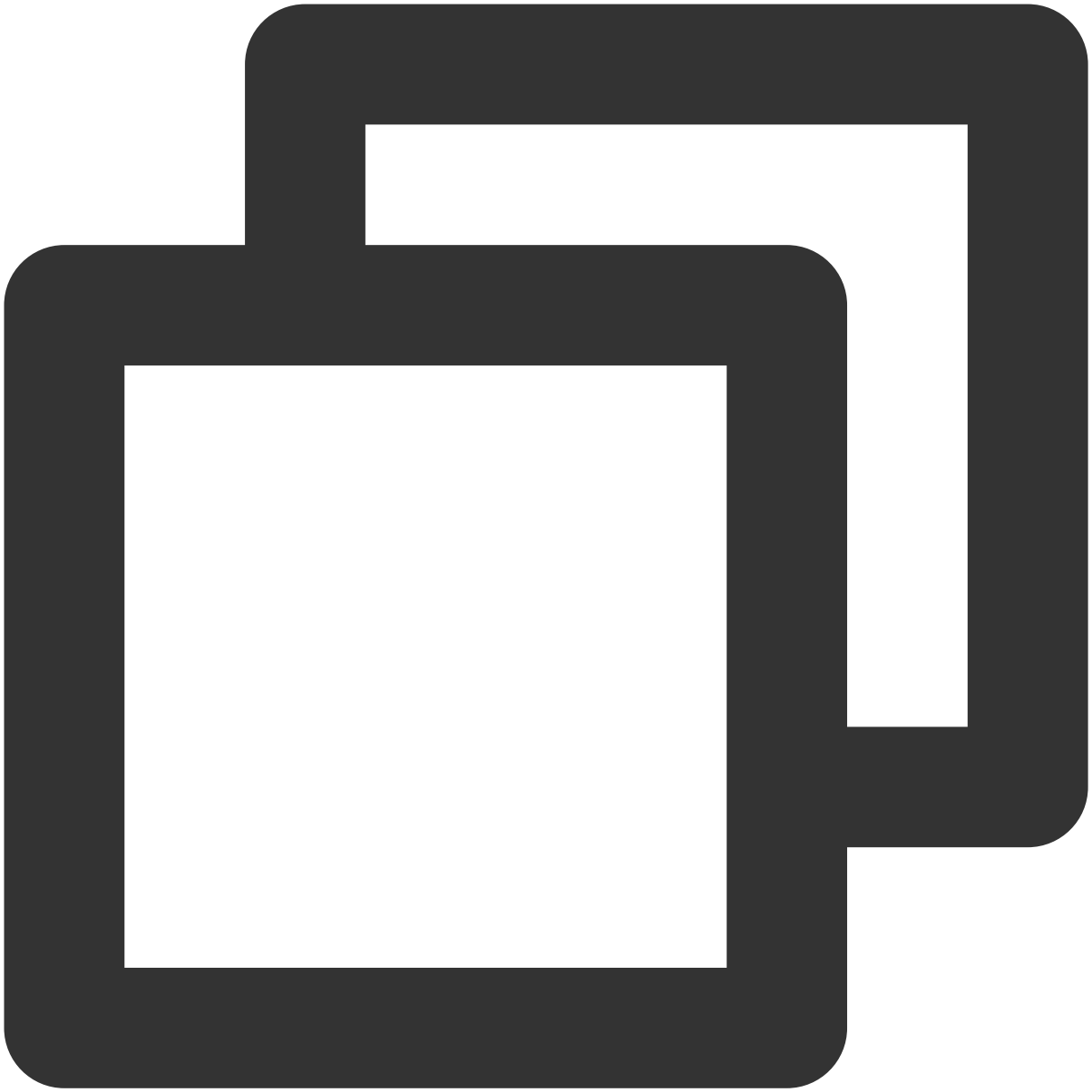
Example:



```
> SELECT from_utc_timestamp('2016-08-31', 'Asia/Seoul');
 2016-08-31 09:00:00
```

HOUR

Function statement:



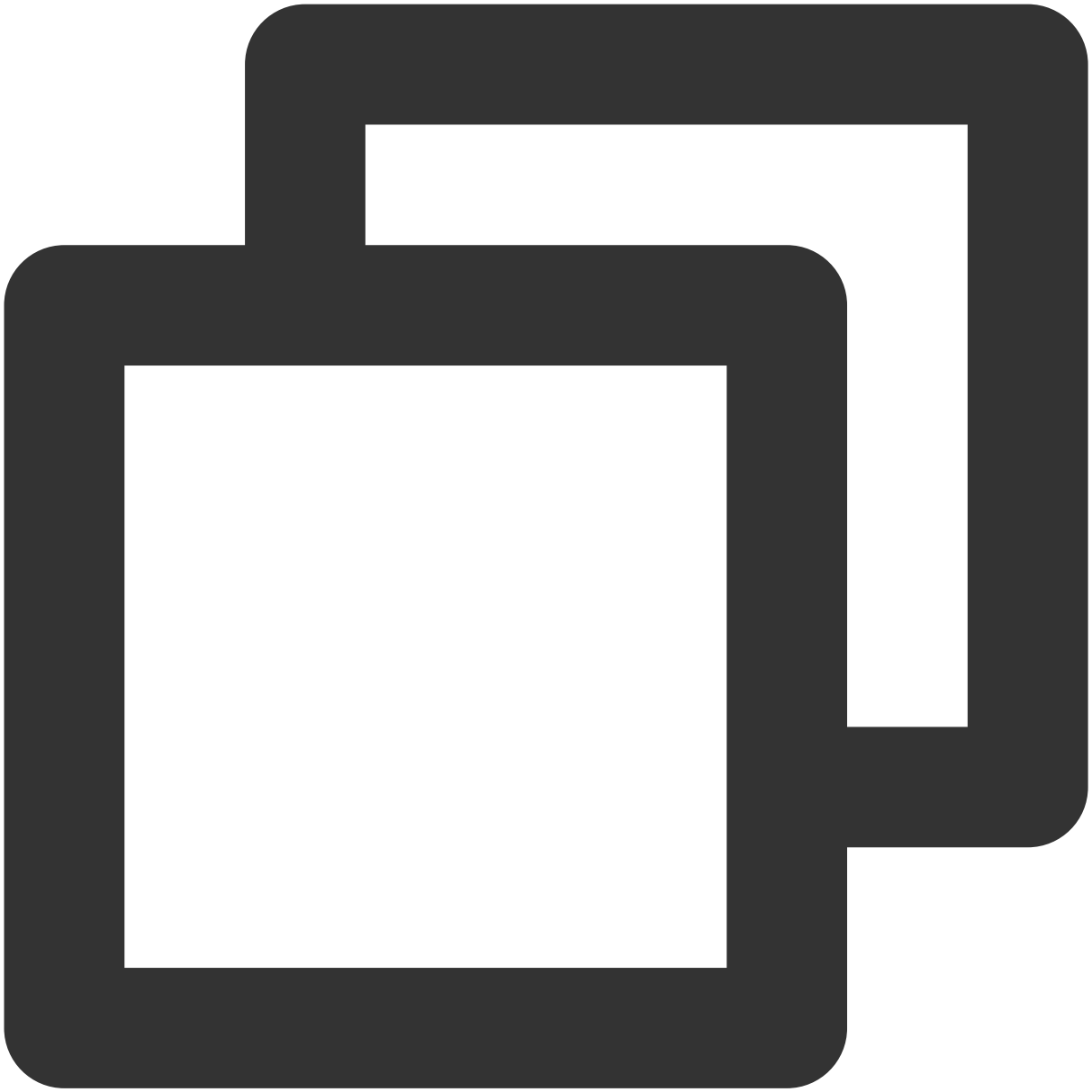
```
HOUR(<ts> string|timestamp)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the hour part of the specified timestamp ts.

Return type: integer

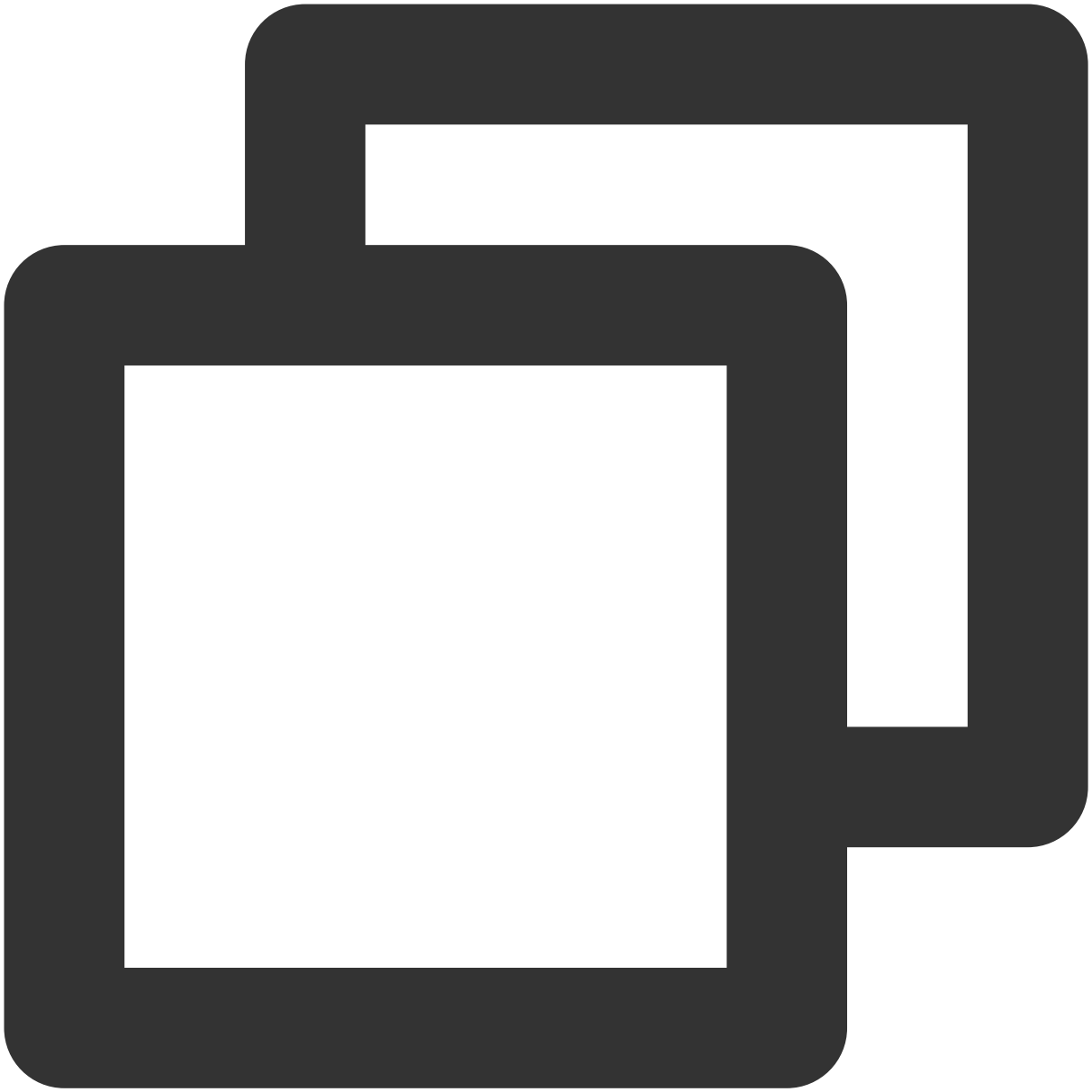
Example:



```
> SELECT hour('2009-07-30 12:58:59');
 12
```

LAST_DAY

Function statement:



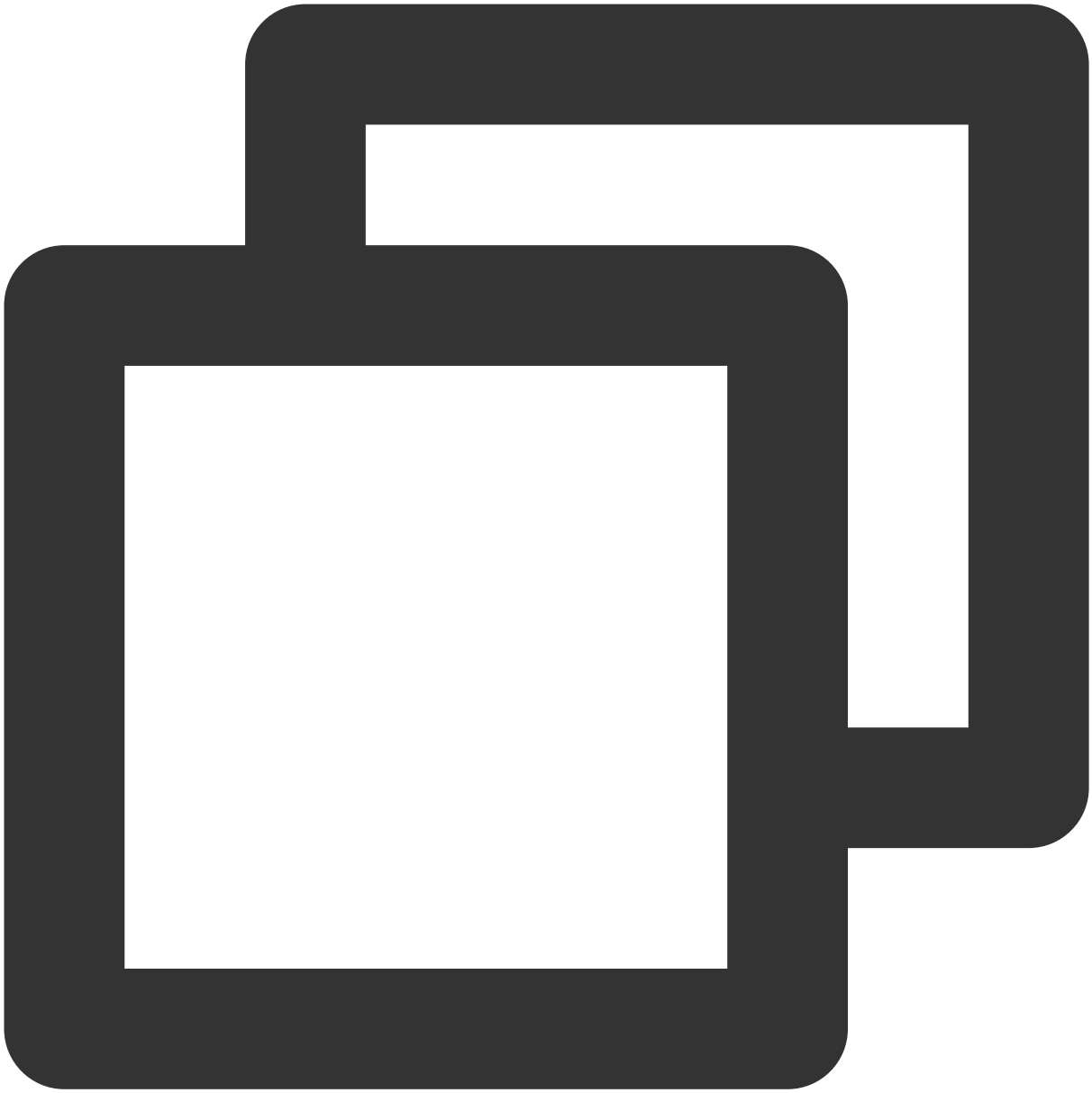
```
LAST_DAY(<d> date|string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the last day of the current month for the date d.

Return type: date

Example:



```
> SELECT last_day('2009-01-12');
 2009-01-31
```

MINUTE

Function statement:

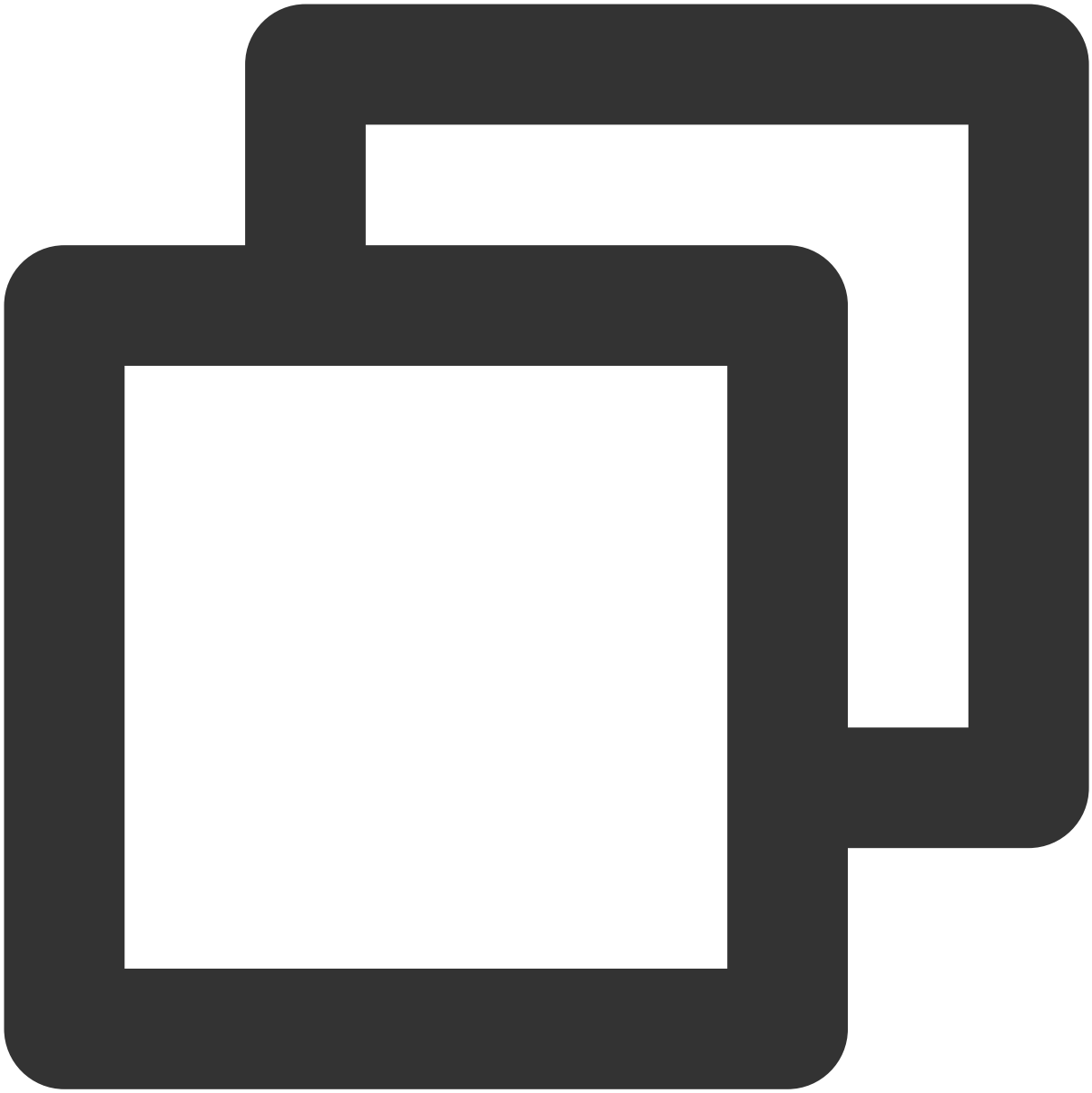
```
MINUTE(<ts> timestamp|string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the minute part of the timestamp ts.

Return type: integer

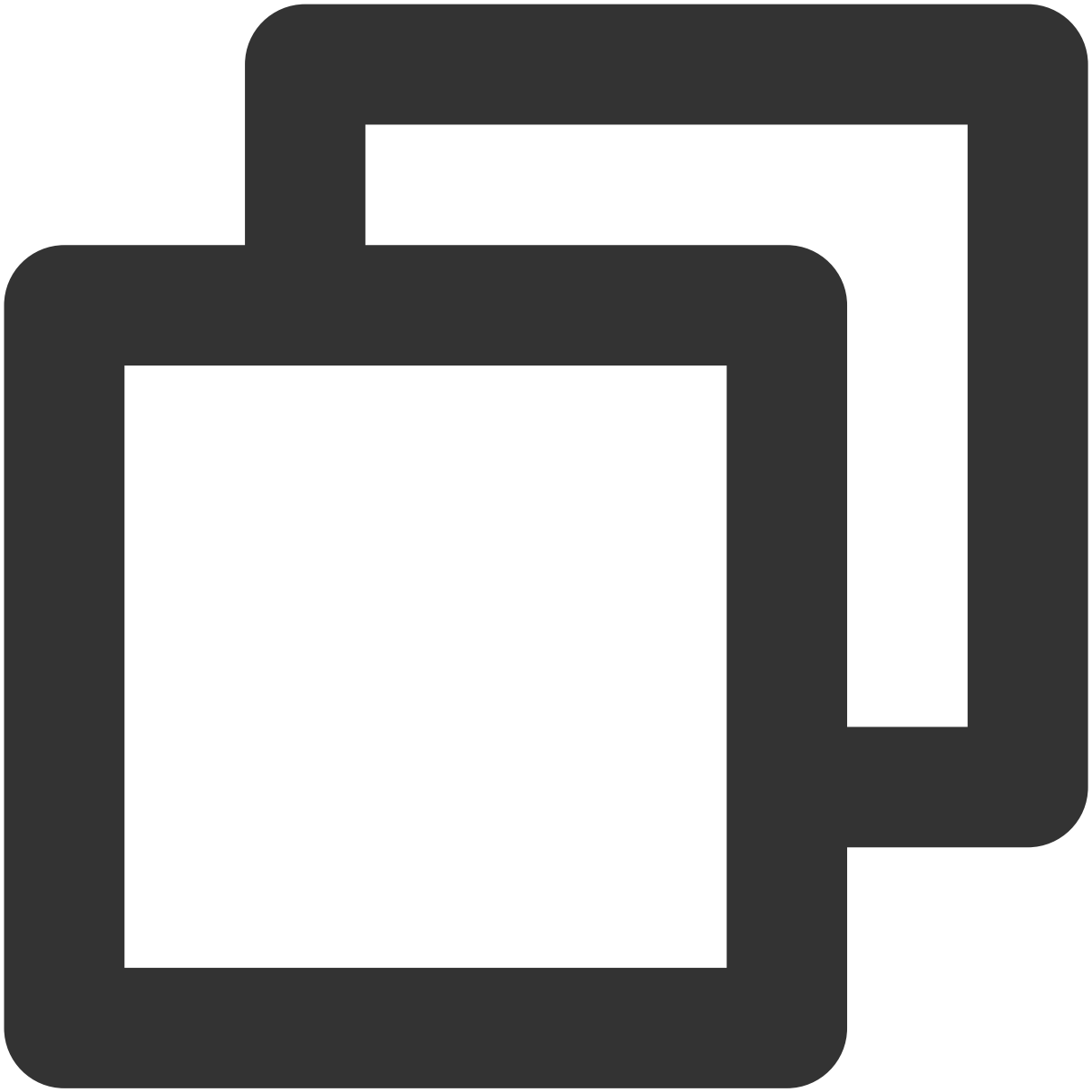
Example:



```
> SELECT minute('2009-07-30 12:58:59');
 58
```

MONTH

Function statement:



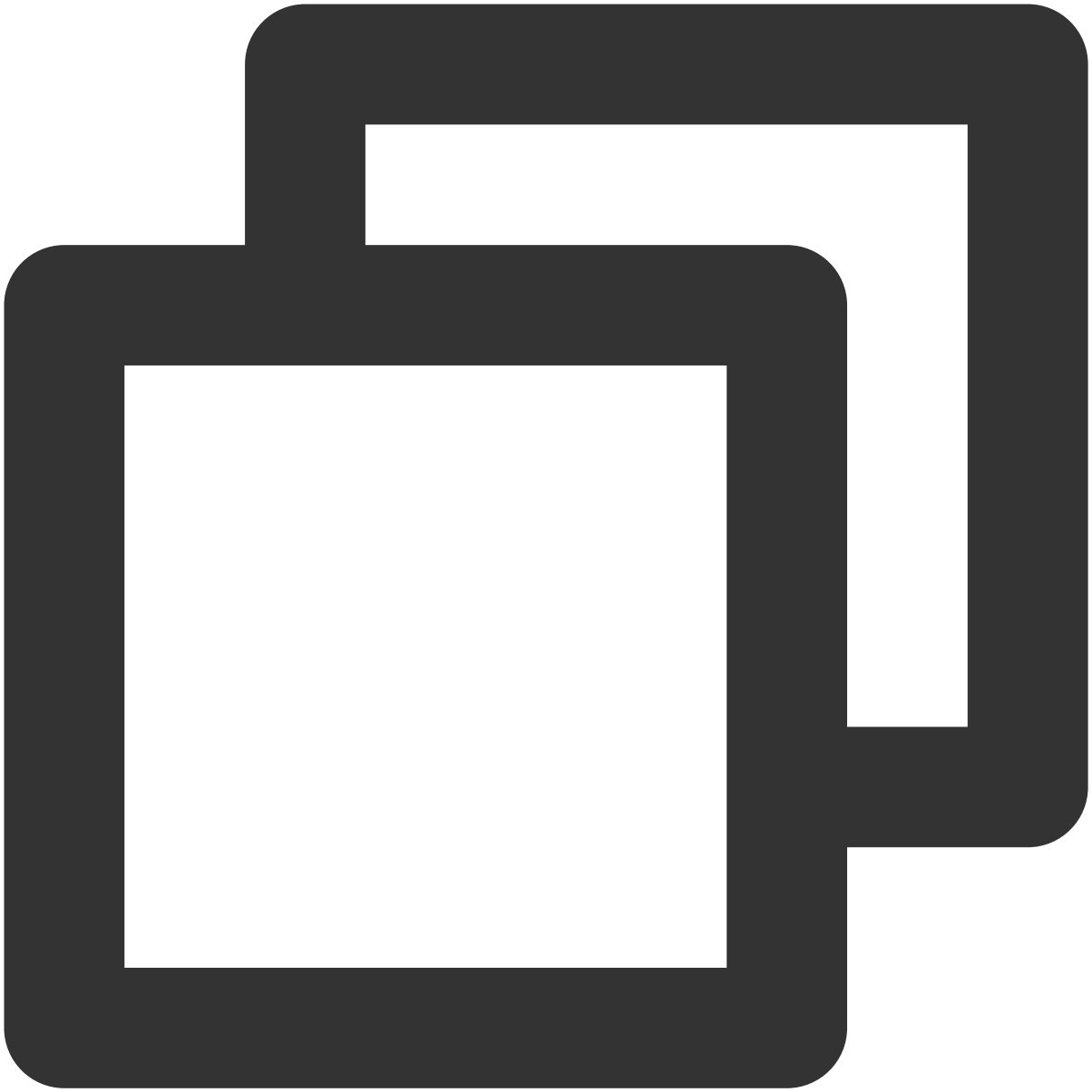
```
MONTH(<d> date|timestamp|string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the month of the date d.

Return type: integer

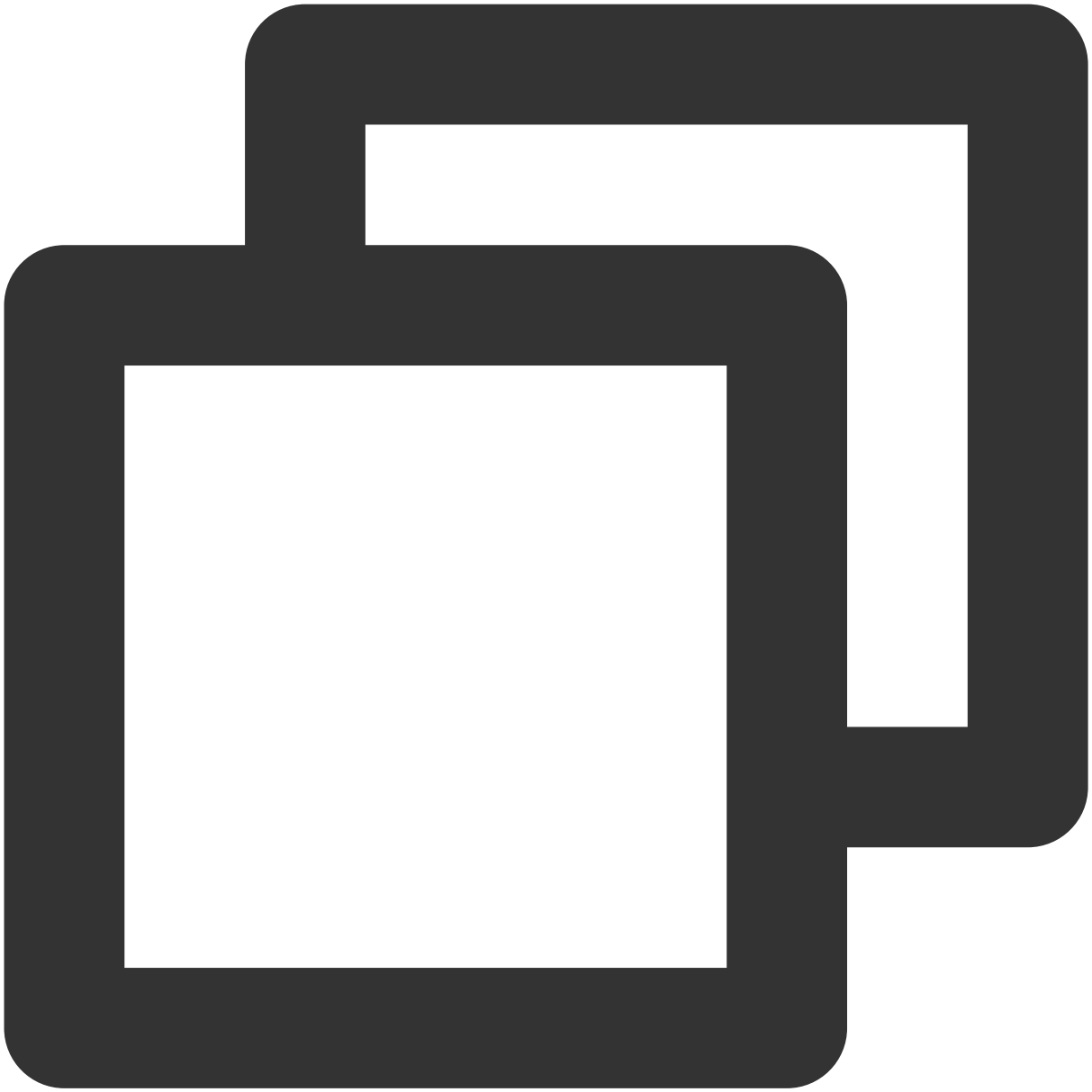
Example:



```
> SELECT month('2016-07-30');
 7
```

MONTHS_BETWEEN

Function statement:



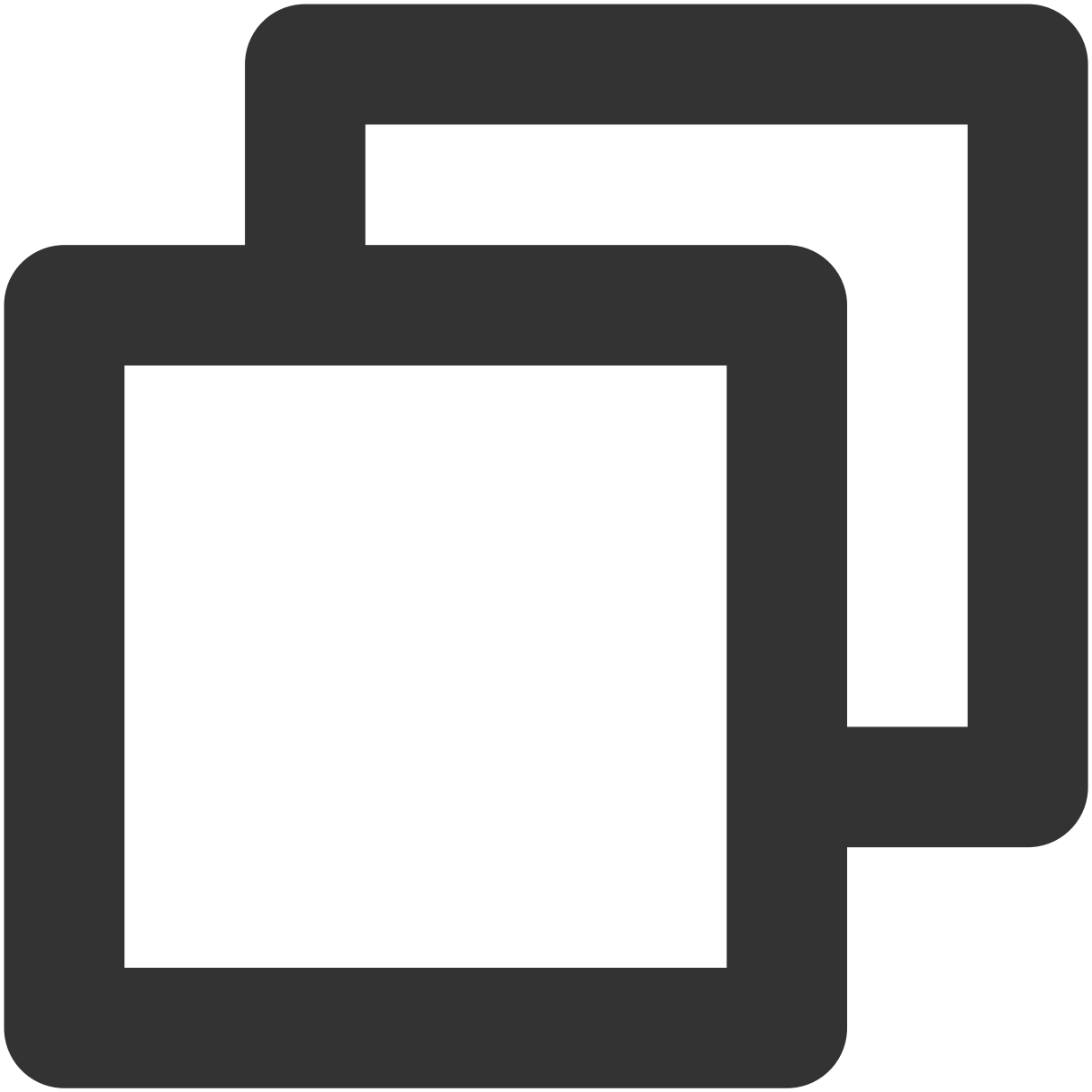
```
MONTHS_BETWEEN(<ts1> date|timestamp|string, <ts2> date|timestamp|string, <roundOff>
```

Supported engines: SparkSQL and Presto

Usage instructions: If ts1 is later than ts2, the result is positive. If ts1 and ts2 are on the same day of the month or both are the last day of the month, the time of the day is ignored. Otherwise, the difference is calculated assuming 31 days per month and rounded to 8 decimal places unless roundOff is false.

Return type: double

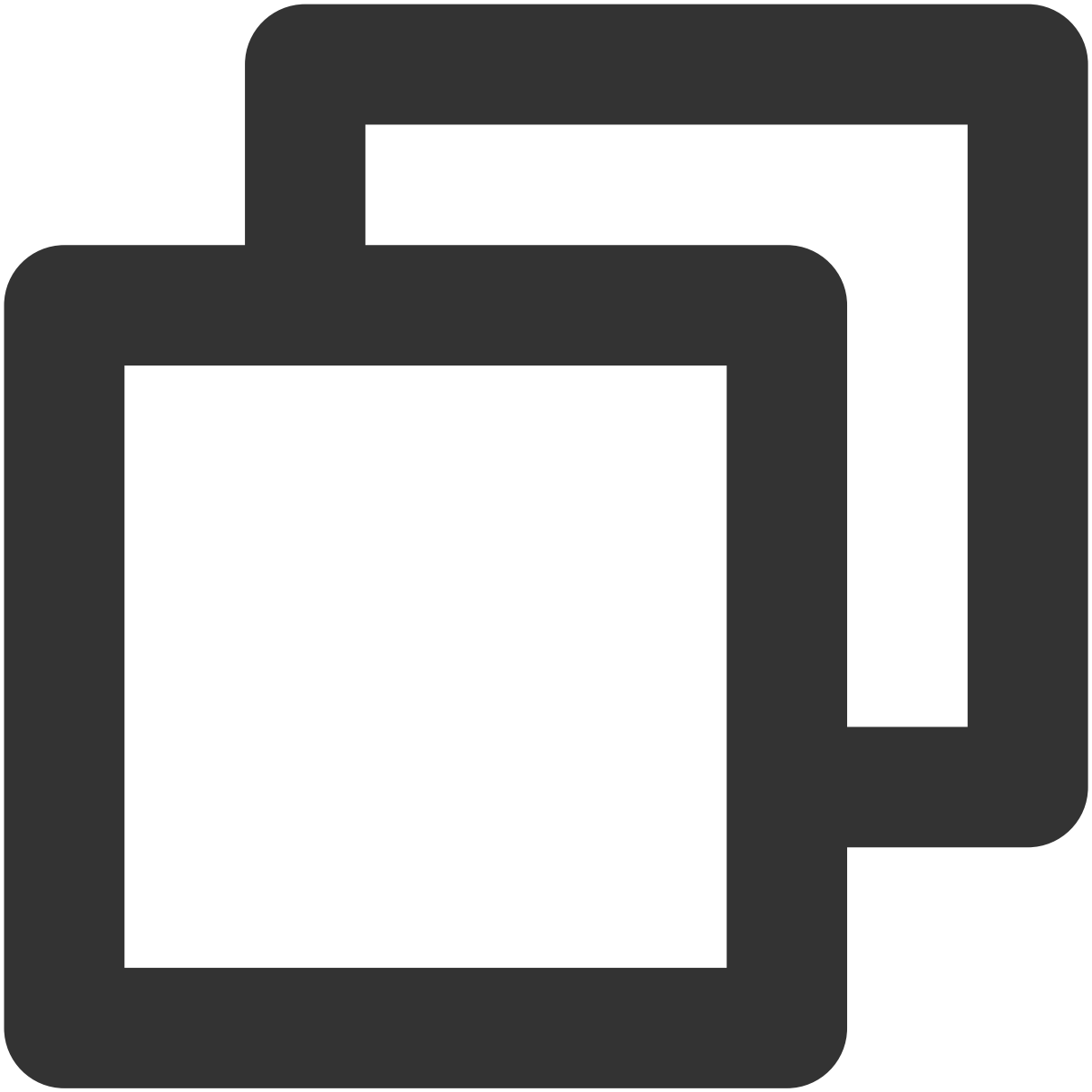
Example:



```
> SELECT months_between('1997-02-28 10:30:00', '1996-10-30');
 3.94959677
> SELECT months_between('1997-02-28 10:30:00', '1996-10-30', false);
 3.9495967741935485
```

NEXT_DAY

Function statement:



```
NEXT_DAY(<start_date> date|timestamp|string, <day_of_week> string
```

Supported engines: SparkSQL and Presto

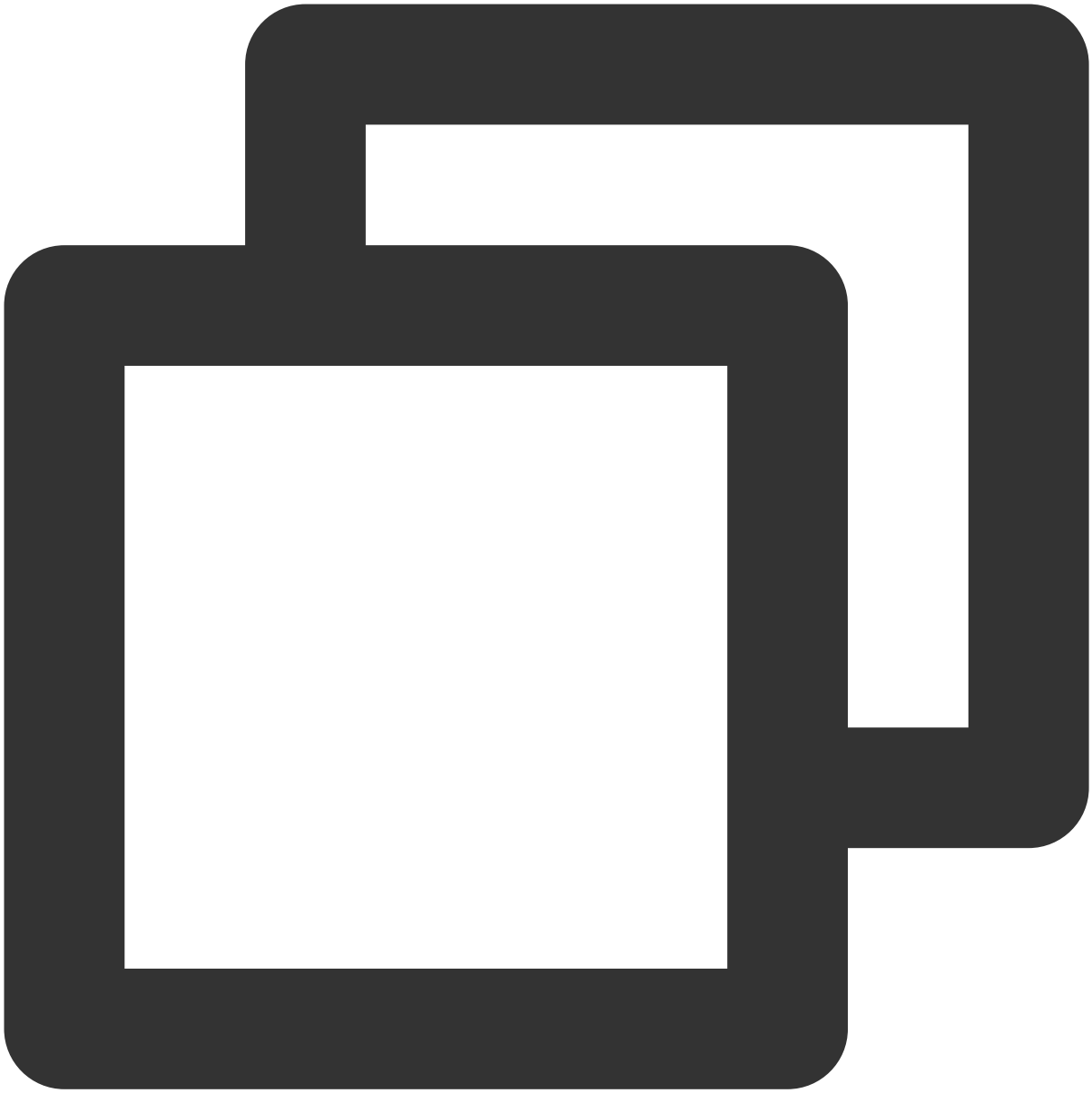
Usage instructions: Return the first specified week after the start_date.

Return type:

SparkSQL:date

Presto:string

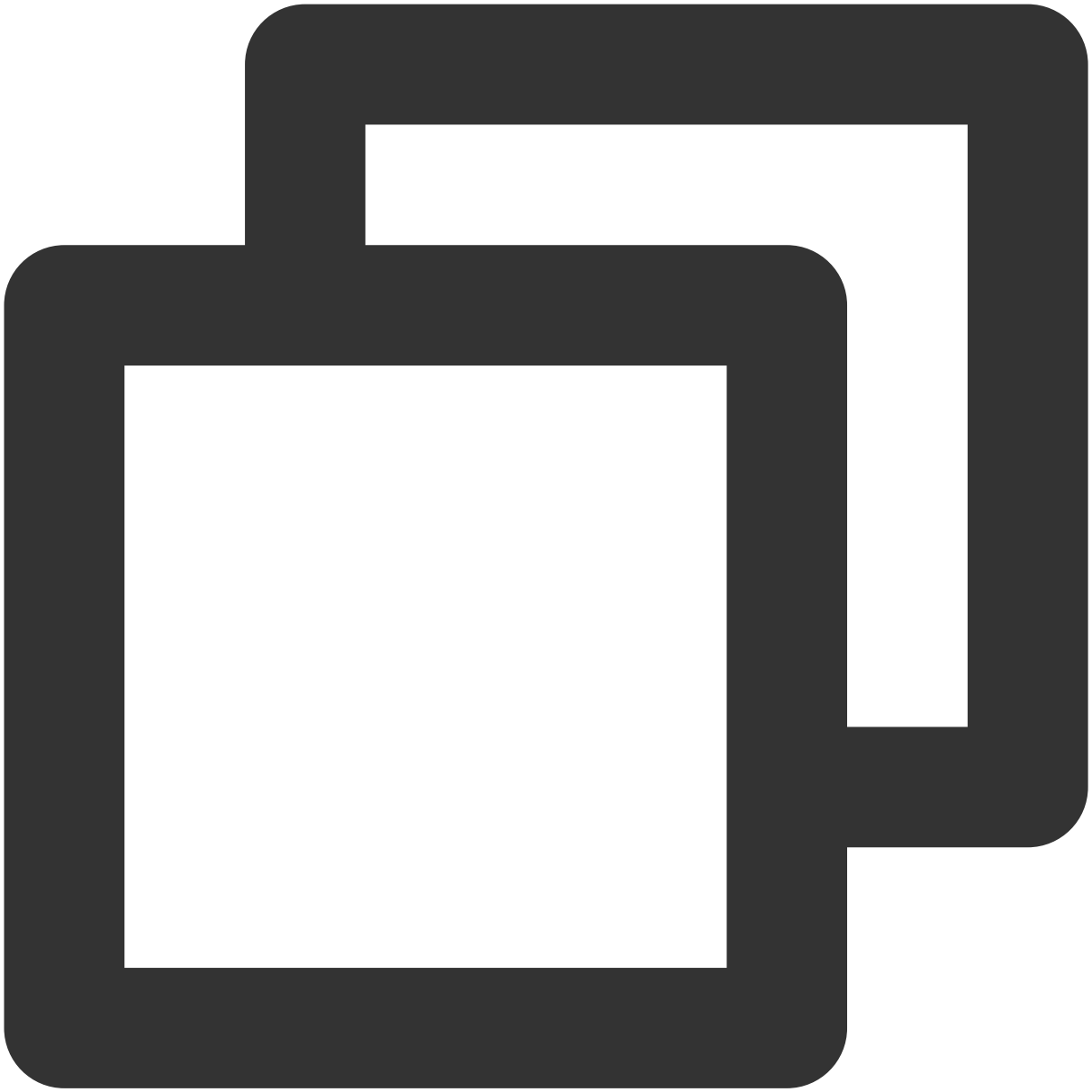
Example:



```
> SELECT next_day('2015-01-14', 'TU');
 2015-01-20
```

NOW

Function statement:



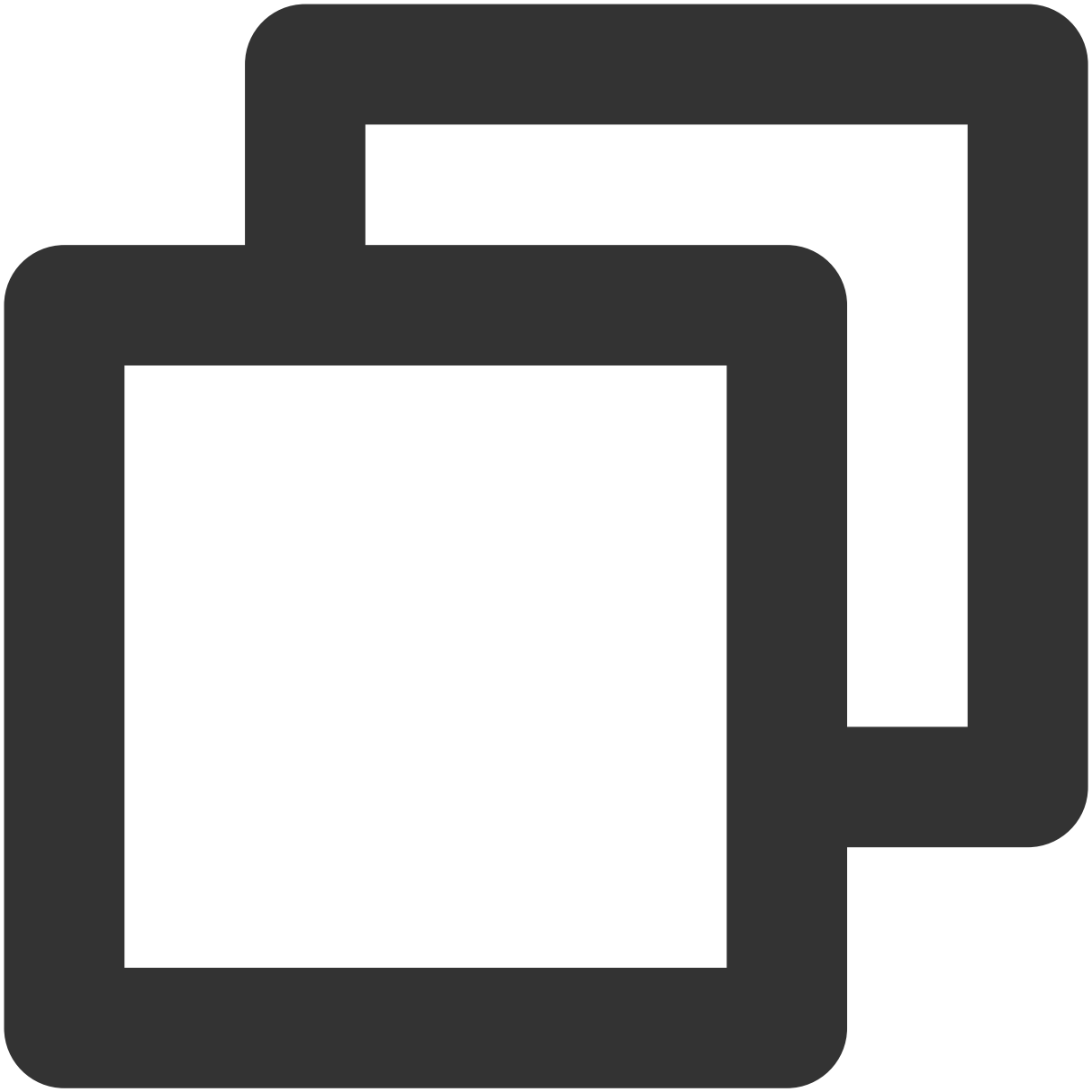
```
NOW()
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the current timestamp.

Return type: timestamp

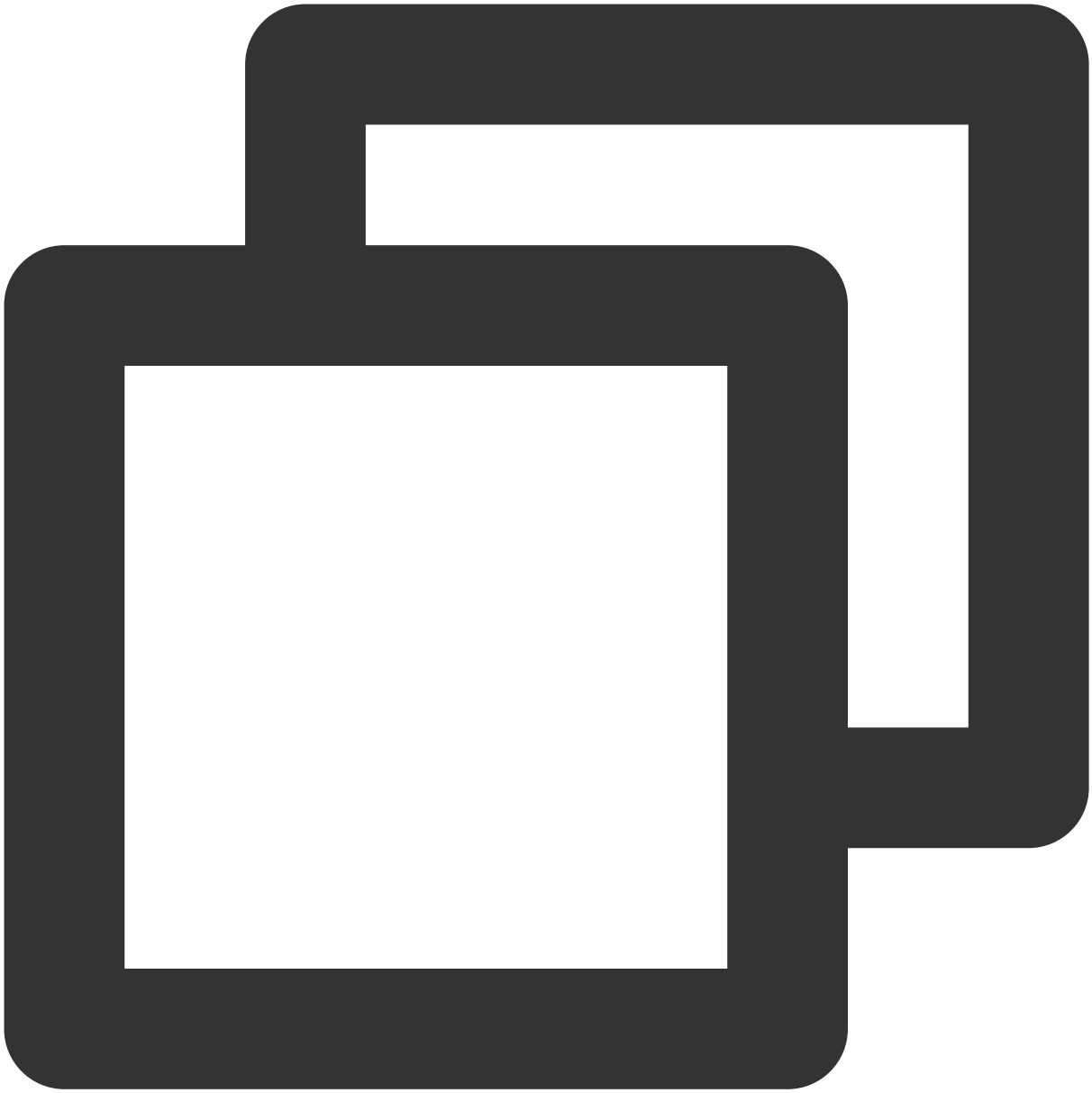
Example:



```
> SELECT now();
 2020-04-25 15:49:11.914
```

QUARTER

Function statement:



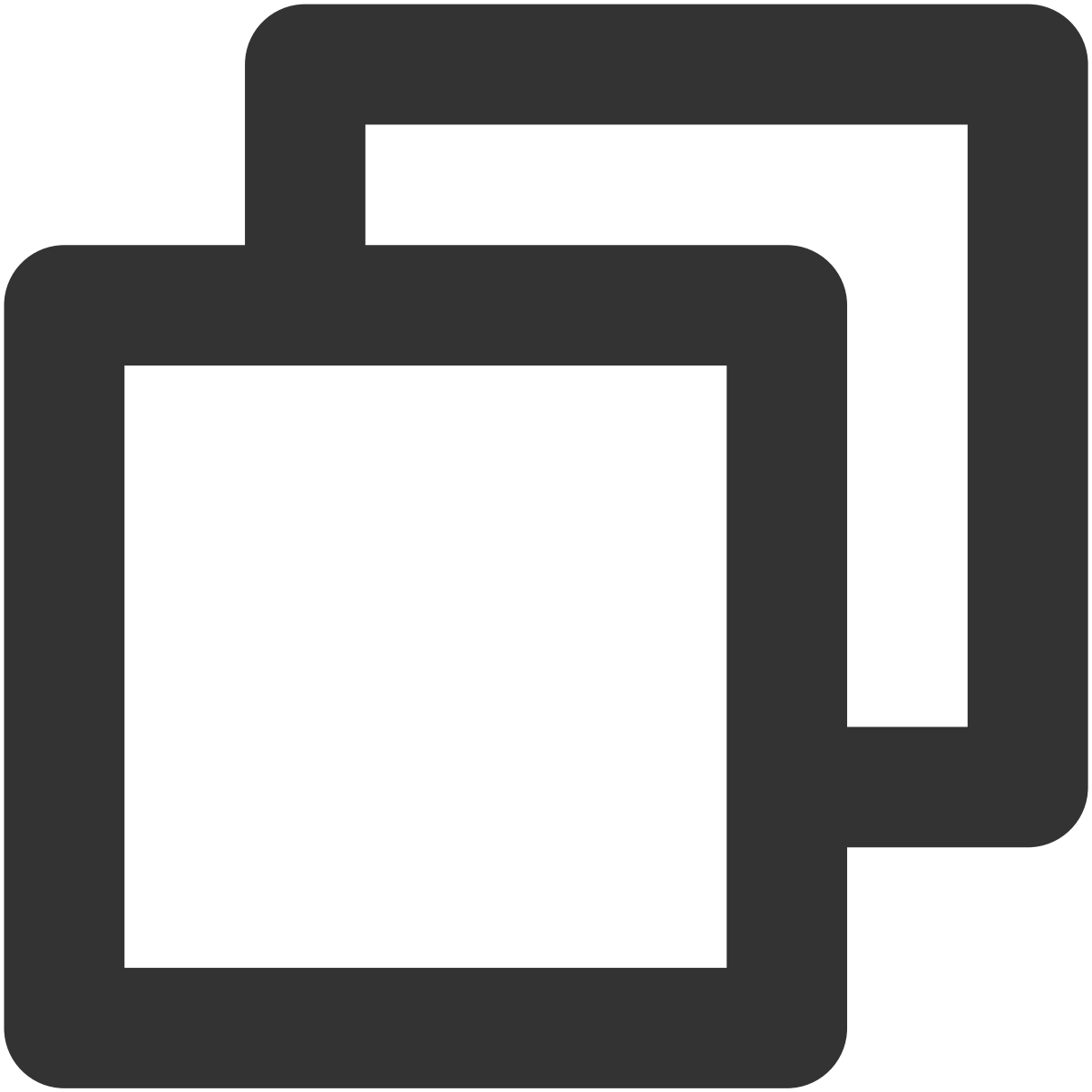
```
QUARTER(<d> date|timestamp|string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the quarter of the year for date d.

Return type: integer

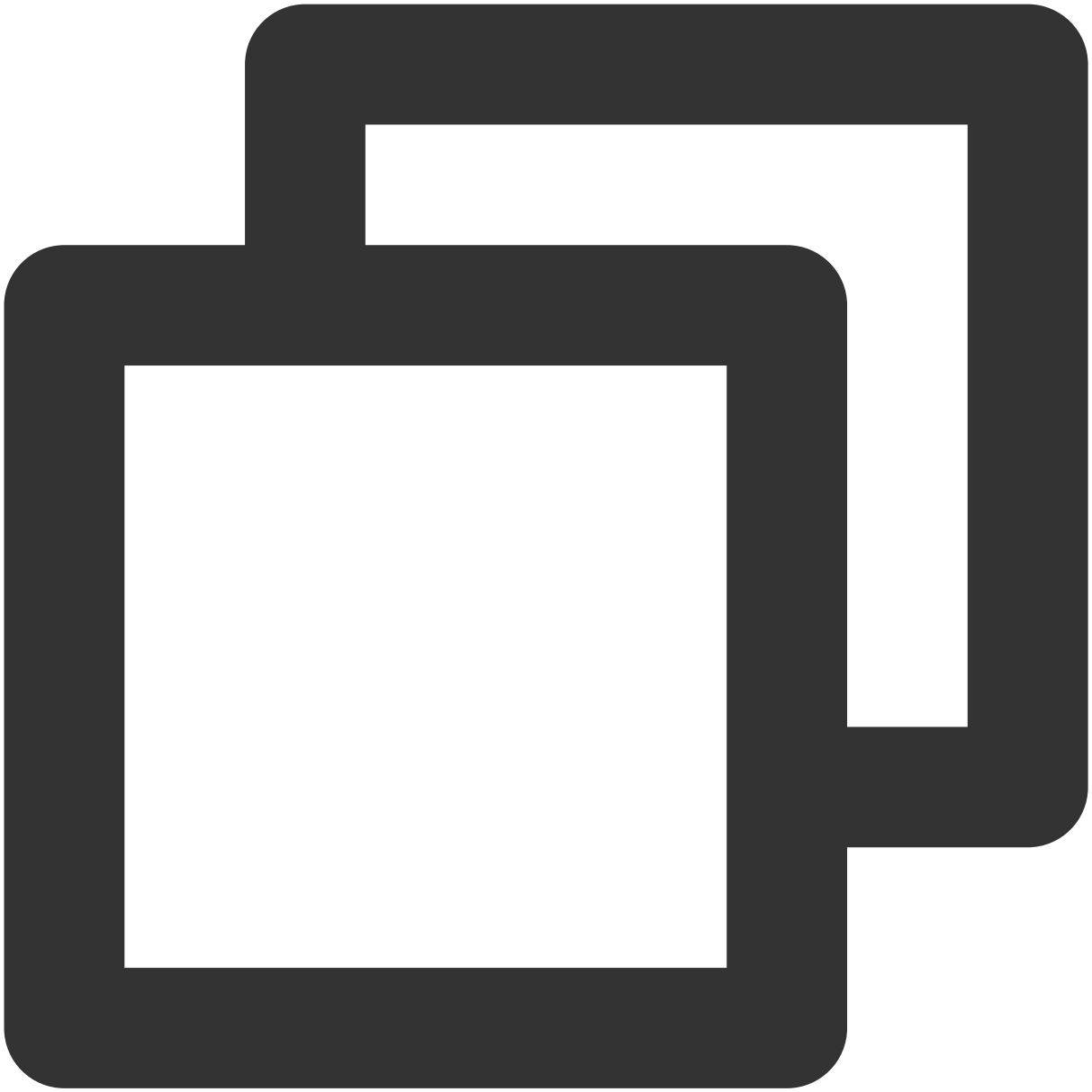
Example:



```
> SELECT quarter('2016-08-31');
 3
```

SECOND

Function statement:



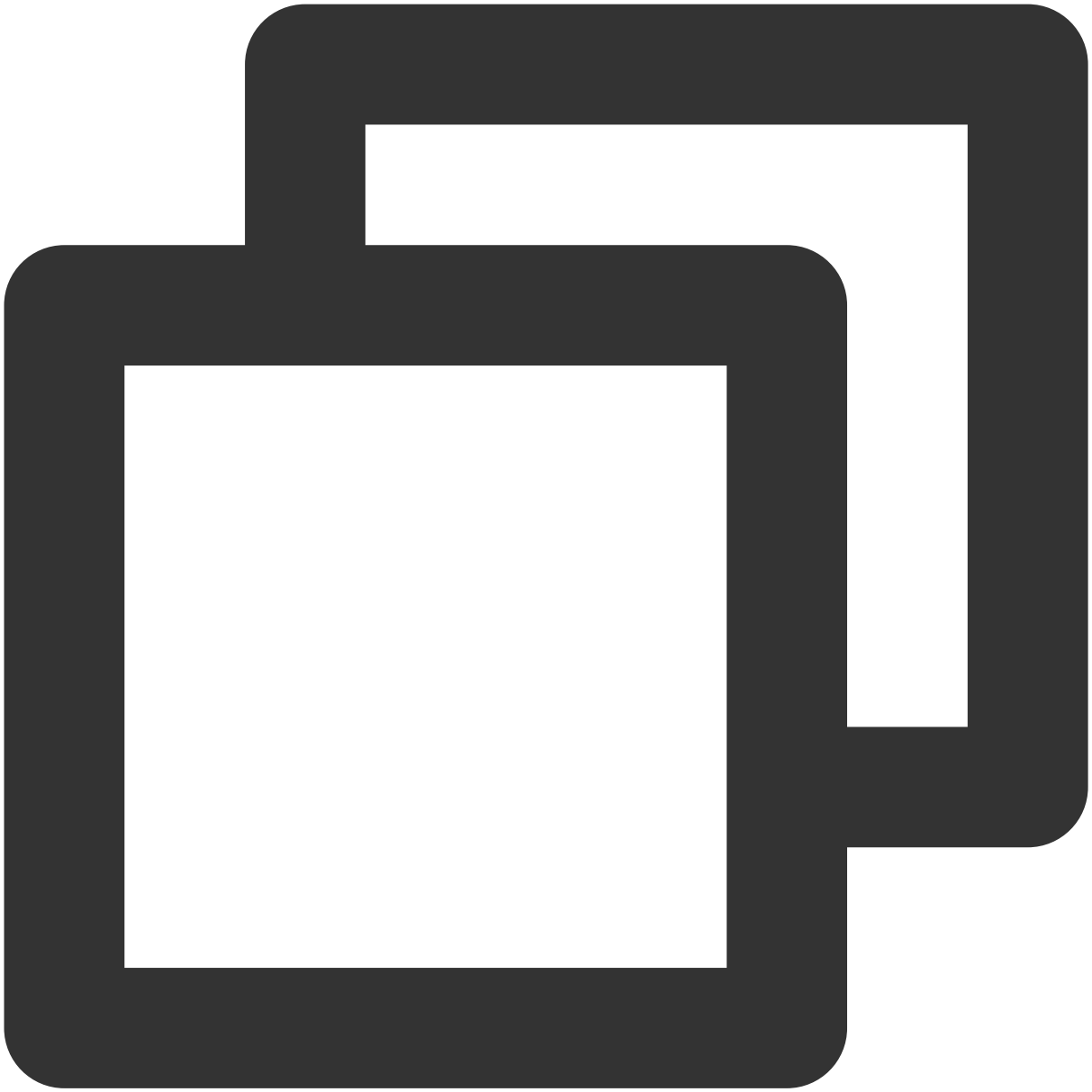
```
SECOND(<ts> timestamp|string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the seconds of the current timestamp.

Return type: integer

Example:



```
> SELECT second('2009-07-30 12:58:59');
 59
```

TO_TIMESTAMP

Function statement:

```
TO_TIMESTAMP(<ts_str> string[, <fmt> string]
```

Supported engines: SparkSQL and Presto

Usage instructions: Parse the ts_str expression in fmt format to a timestamp. If the input is invalid, NULL is returned. By default, if fmt is omitted, it follows the forced conversion rules for the timestamp. The result data type is consistent with the configuration value.

Return type: timestamp

Example:

```
> SELECT to_timestamp('2016-12-31 00:12:00');
 2016-12-31 00:12:00
> SELECT to_timestamp('2016-12-31', 'yyyy-MM-dd');
 2016-12-31 00:00:00
```

TO_DATE

Function statement:

```
TO_DATE(<date_str> string[, <fmt> string])
```

Supported engines: SparkSQL and Presto

Usage instructions: Parse the date_str expression in fmt format to a date. If the input is invalid, NULL is returned. By default, if fmt is omitted, it follows the forced conversion rules for the date. The result data type is consistent with the configuration value.

Return type: date

Example:

```
> SELECT to_date('2009-07-30 04:17:52');
 2009-07-30
> SELECT to_date('2016-12-31', 'yyyy-MM-dd');
 2016-12-31
```

TO_UNIX_TIMESTAMP

Function statement:

```
TO_UNIX_TIMESTAMP(<ts> date|timestamp|string[, <fmt> string])
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the unix timestamp of ts.

Return type: bigint

Example:

```
> SELECT to_unix_timestamp('2016-04-08', 'yyyy-MM-dd');
 1460098800
```

TO_UTC_TIMESTAMP

Function statement:

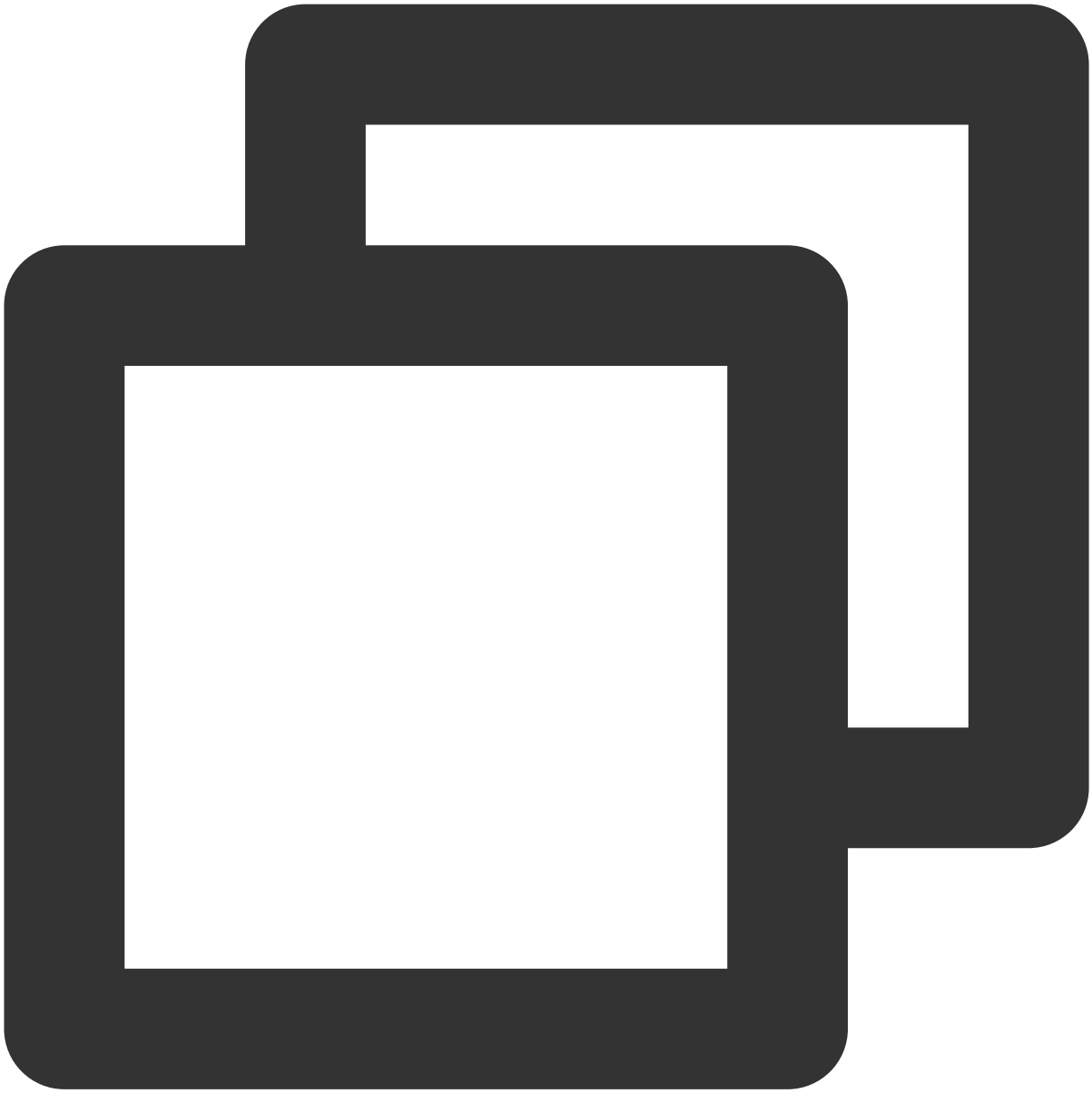
```
-- SparkSQL
TO_UTC_TIMESTAMP(<ts> date|timestamp|string, <timezone> string)

-- Presto
TO_UTC_TIMESTAMP(<ts> date|timestamp|string|interger|double|decimal, <timezone> str
```

Supported engines: SparkSQL and Presto

Usage instructions: Convert a timestamp in the given timezone to UTC.

Return type: timestamp

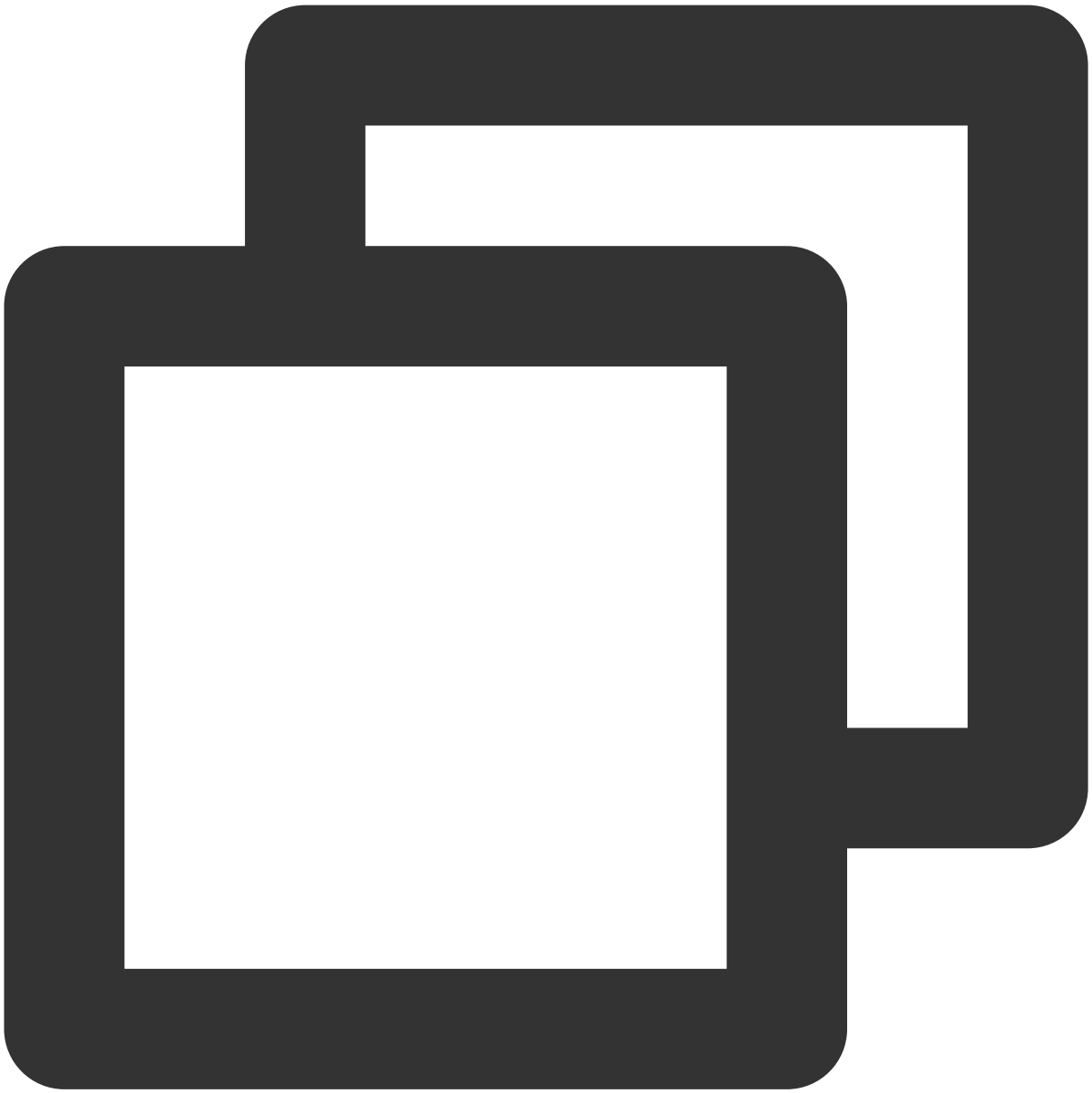
Example:



```
-- SparkSQL
> SELECT to_utc_timestamp('2016-08-31', 'Asia/Seoul');
 2016-08-30 15:00:00
-- Presto
> select to_utc_timestamp(10000, 'UTC');
 1970-01-01 08:00:10
```

TRUNC

Function statement:



```
TRUNC(<d> date|string, <fmt> string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the date value obtained by truncating the date d to the time unit specified by fmt.

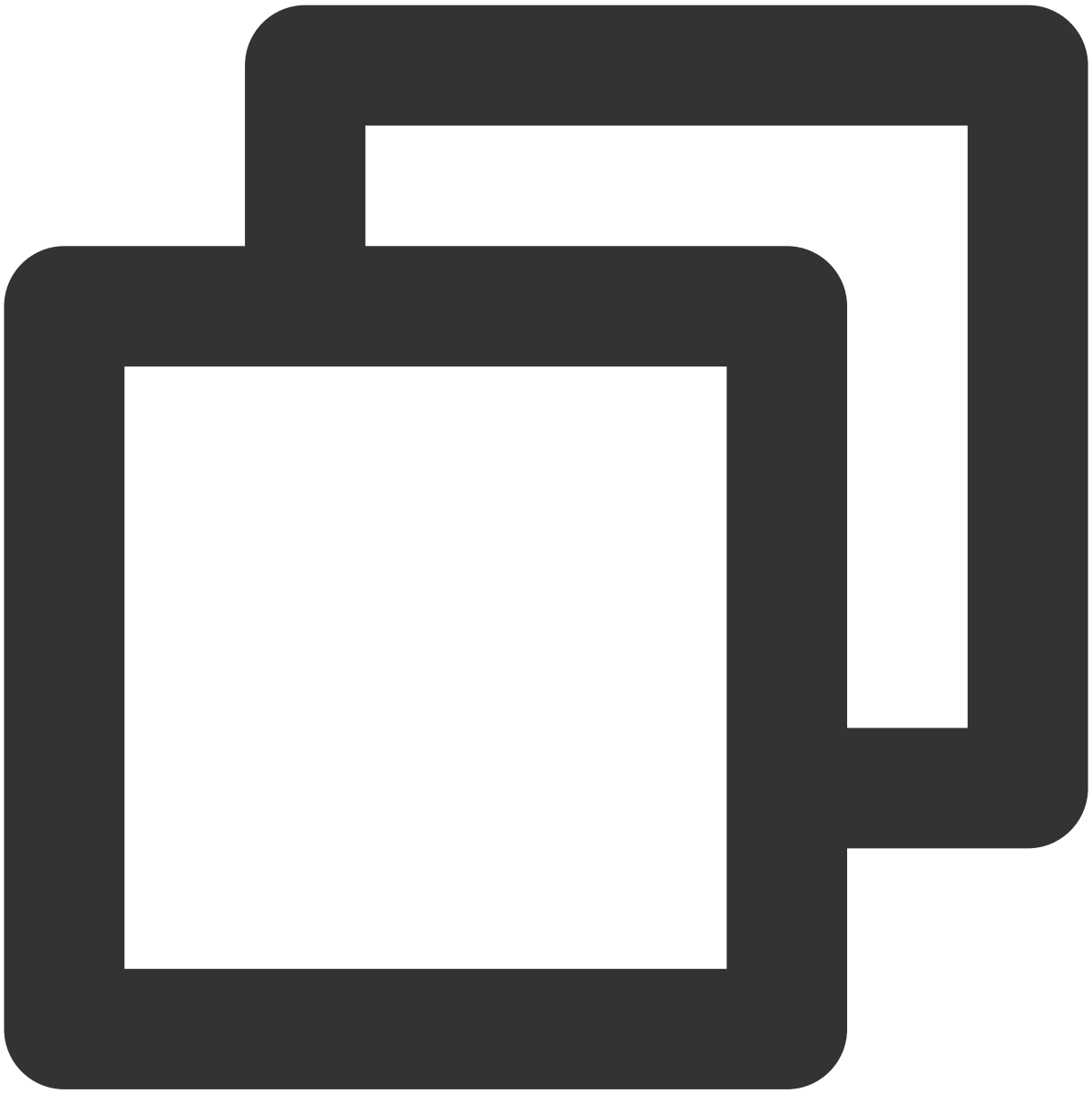
Return type: date

Example:

```
> SELECT trunc('2019-08-04', 'week');
 2019-07-29
> SELECT trunc('2019-08-04', 'quarter');
 2019-07-01
> SELECT trunc('2009-02-12', 'MM');
 2009-02-01
> SELECT trunc('2015-10-27', 'YEAR');
 2015-01-01
```

DATE_TRUNC

Function statement:



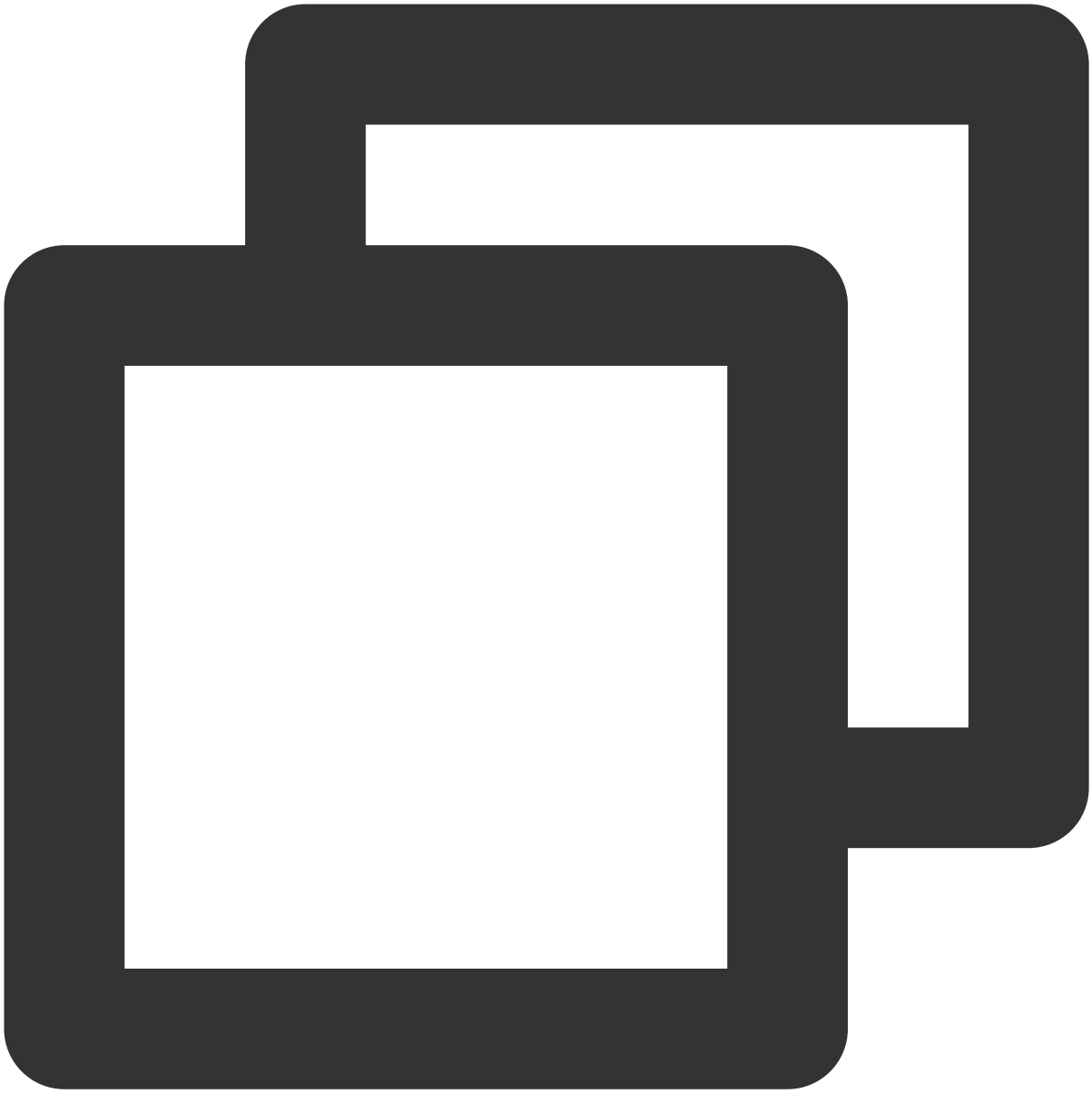
```
DATE_TRUNC(<fmt> string, <ts> date|timestamp|string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the timestamp obtained by truncating the timestamp ts according to fmt.

Return type: timestamp

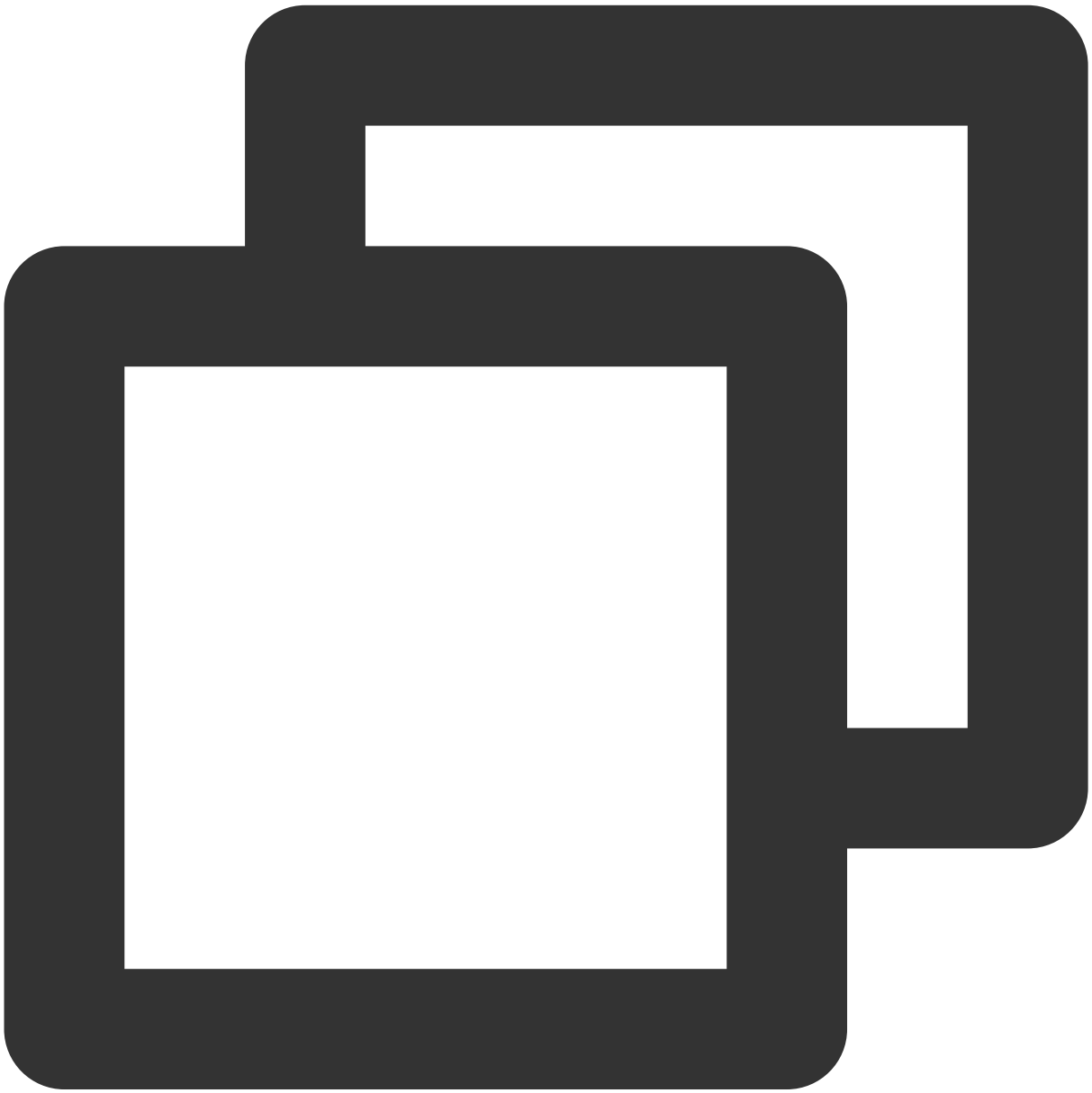
Example:



```
> SELECT date_trunc('YEAR', '2015-03-05T09:32:05.359');
 2015-01-01 00:00:00
> SELECT date_trunc('MM', '2015-03-05T09:32:05.359');
 2015-03-01 00:00:00
> SELECT date_trunc('DD', '2015-03-05T09:32:05.359');
 2015-03-05 00:00:00
> SELECT date_trunc('HOUR', '2015-03-05T09:32:05.359');
 2015-03-05 09:00:00
> SELECT date_trunc('MILLISECOND', '2015-03-05T09:32:05.123456');
 2015-03-05 09:32:05.123
```

UNIX_TIMESTAMP

Function statement:



```
UNIX_TIMESTAMP([<ts> date|timestamp|string[, fmt]])
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the UNIX timestamp of the current or specified time.

Return type: bigint

Example:

```
> SELECT unix_timestamp();
 1476884637
> SELECT unix_timestamp('2016-04-08', 'yyyy-MM-dd');
 1460041200
```

DAYOFWEEK

Function statement:

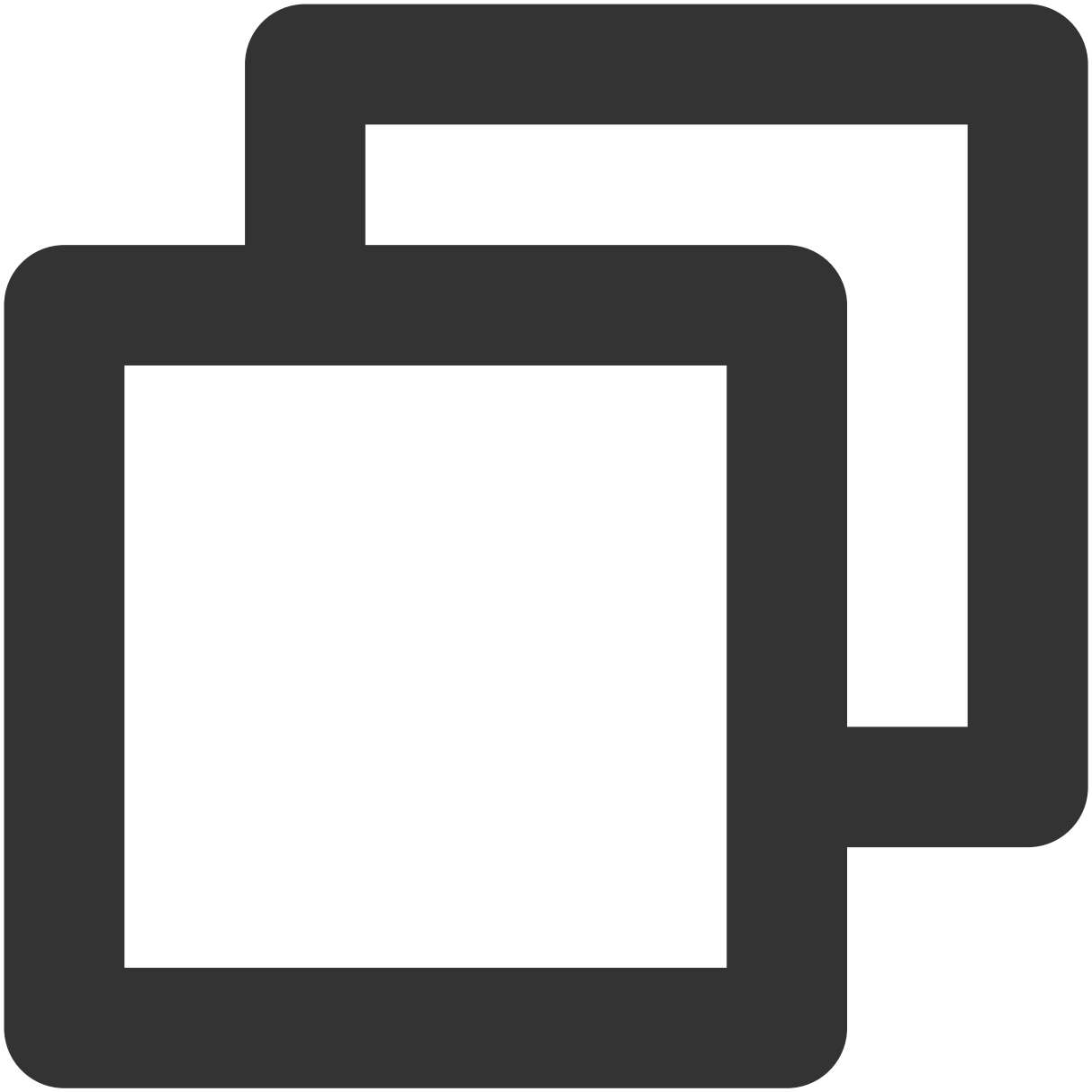
```
DAYOFWEEK(<d> date|timestamp|string)
```

Supported engines: parkSQL and Presto

Usage instructions: Return the day of the week for the date/timestamp d.

Return type: nteger

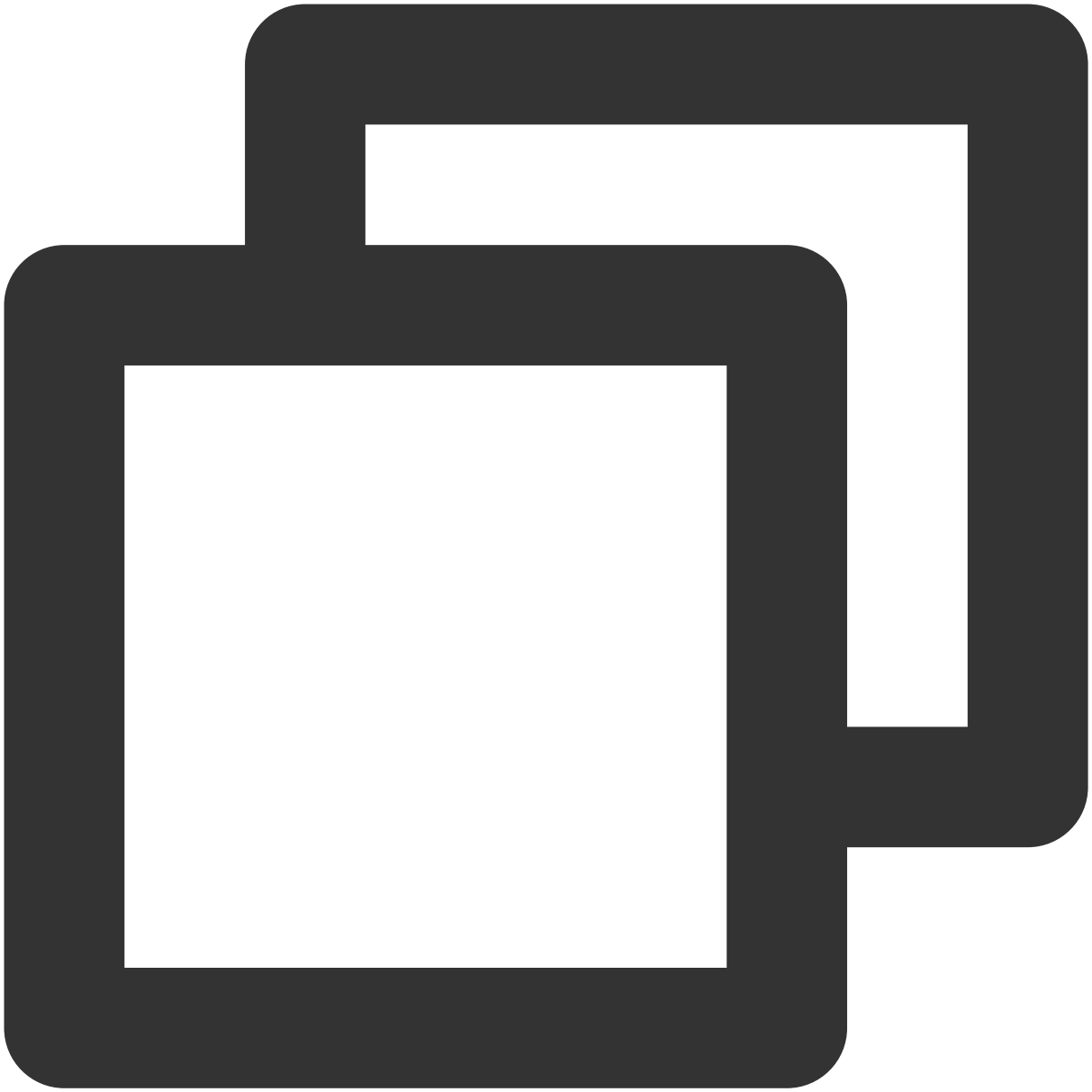
Example:



```
> SELECT dayofweek('2009-07-30');
 5
```

WEEKDAY

Function statement:



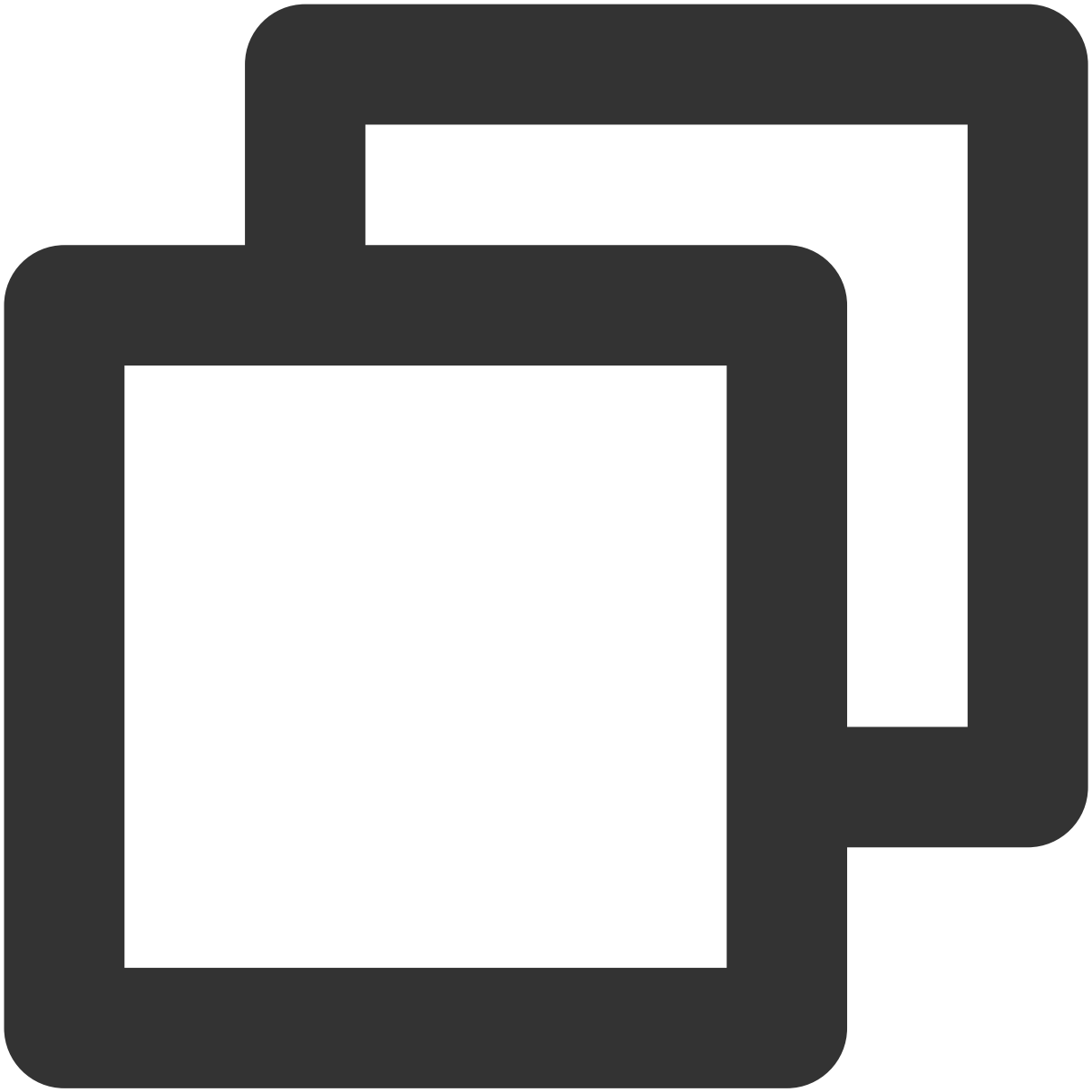
```
WEEKDAY(<d> date|timestamp|string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the day of the week for the date/timestamp d.

Return type: integer

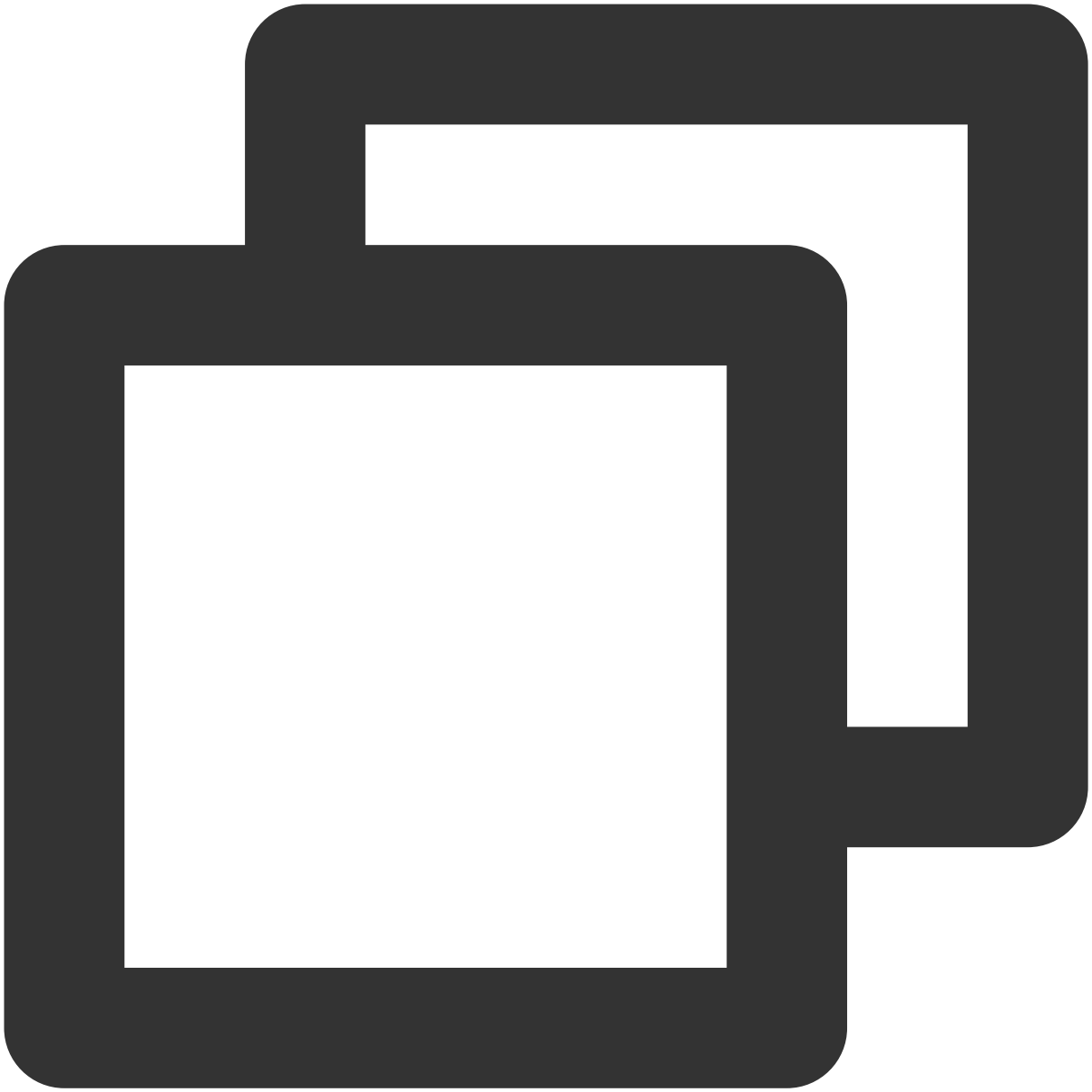
Example:



```
> SELECT weekday('2009-07-30');
 3
```

WEEKOFYEAR

Function statement:



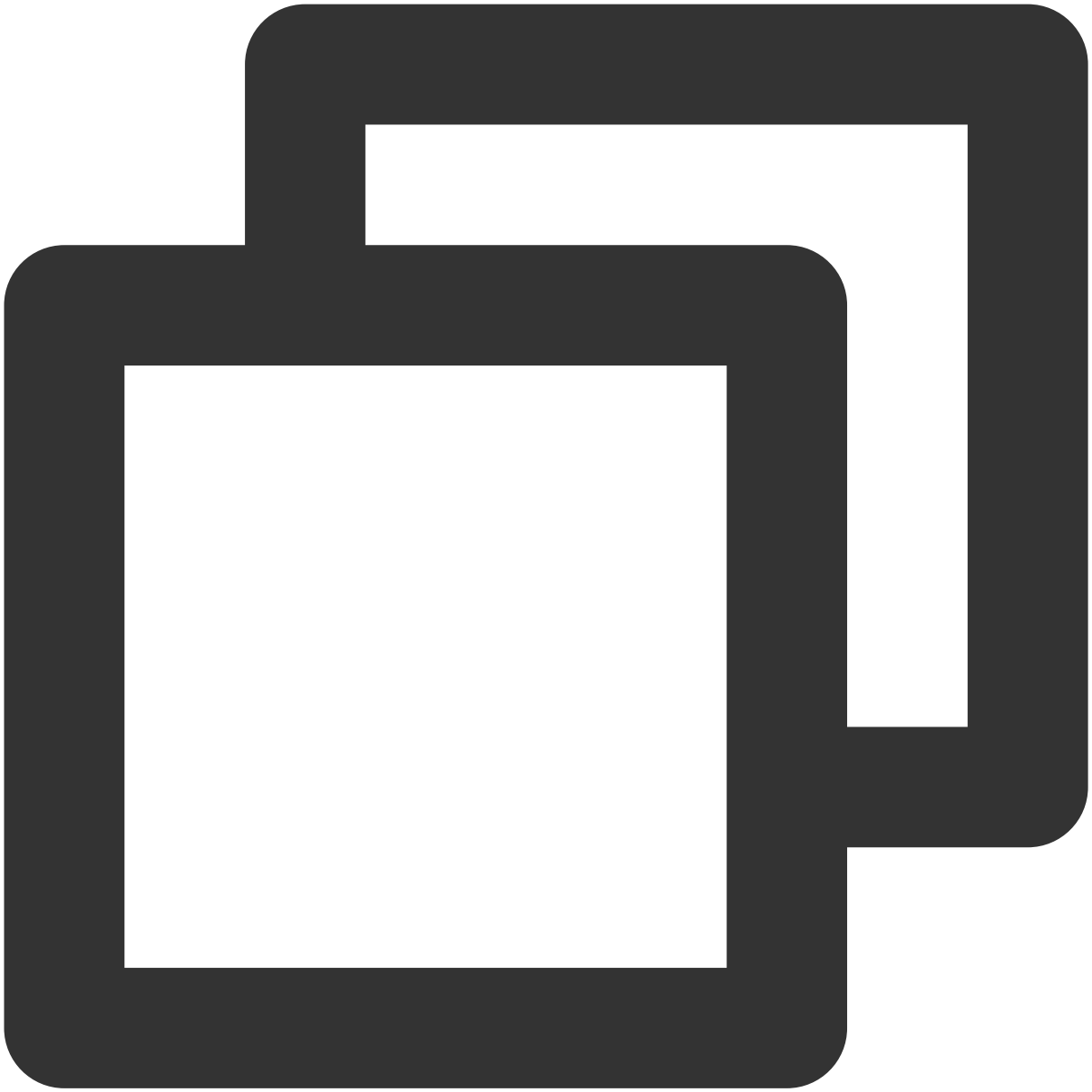
```
WEEKOFYEAR(<d> date|timestamp|string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the week of the year for the given date d.

Return type: integer

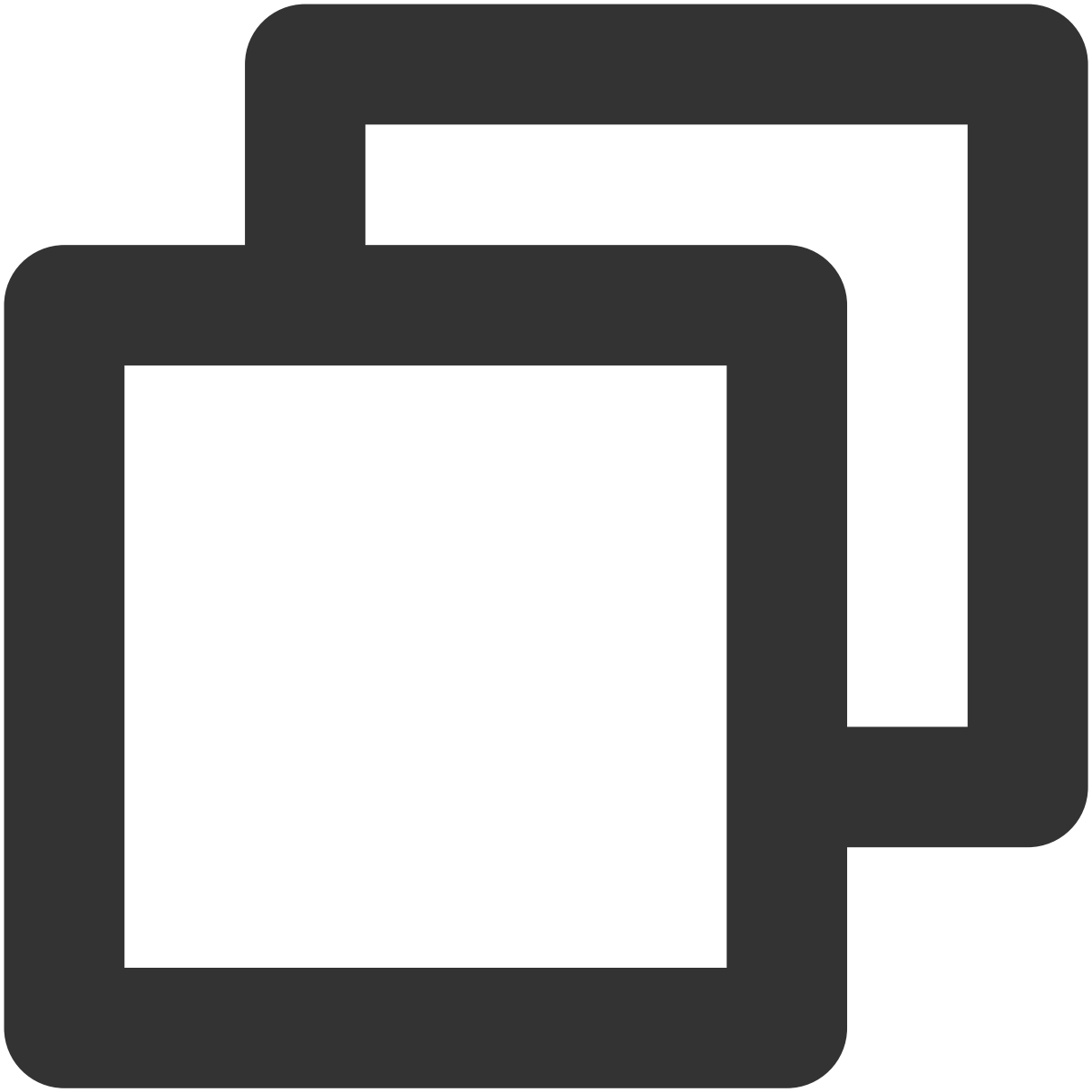
Example:



```
> SELECT weekofyear('2008-02-20');
 8
```

YEAR

Function statement:



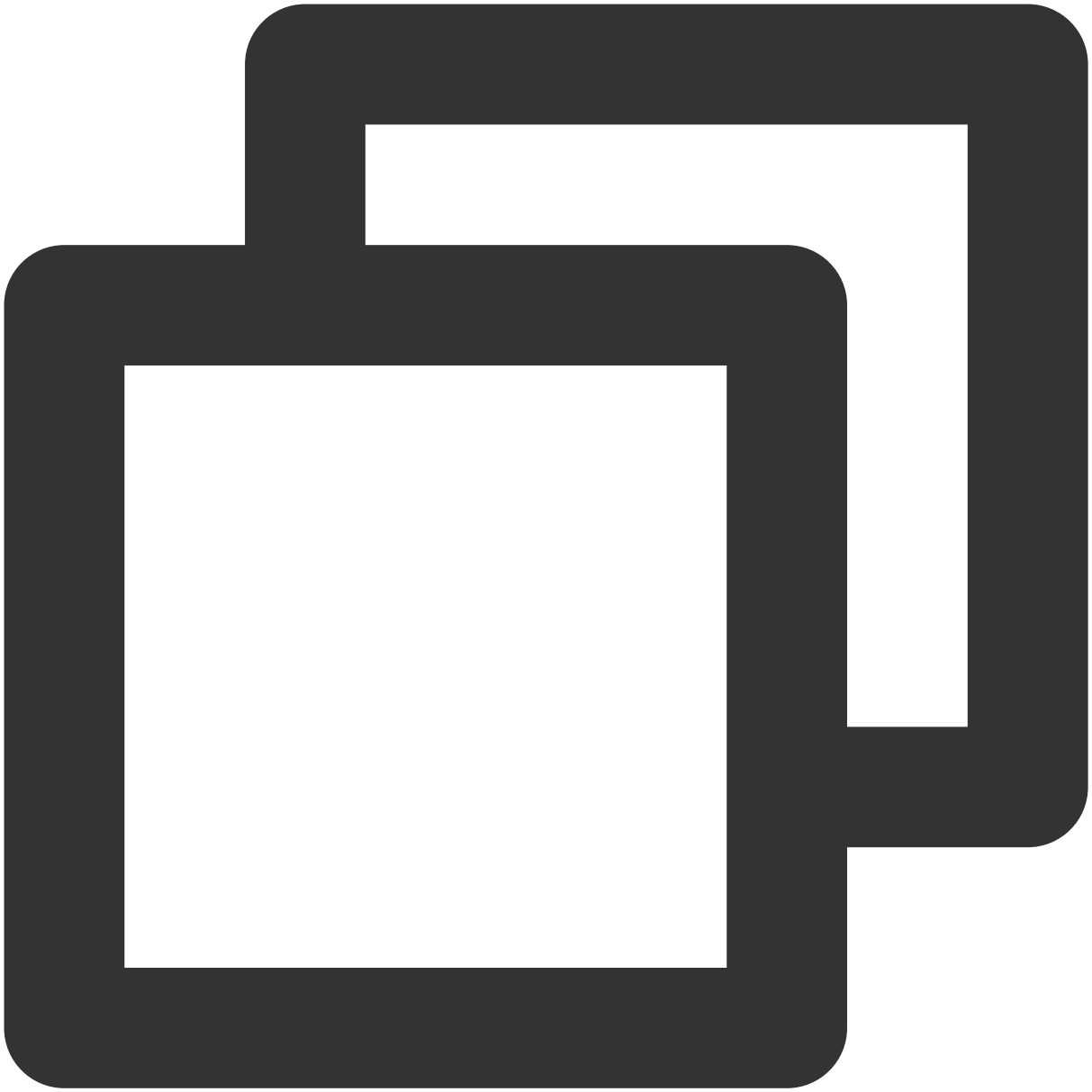
```
YEAR(<d> date|timestamp|string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the year of the date/timestamp d.

Return type: integer

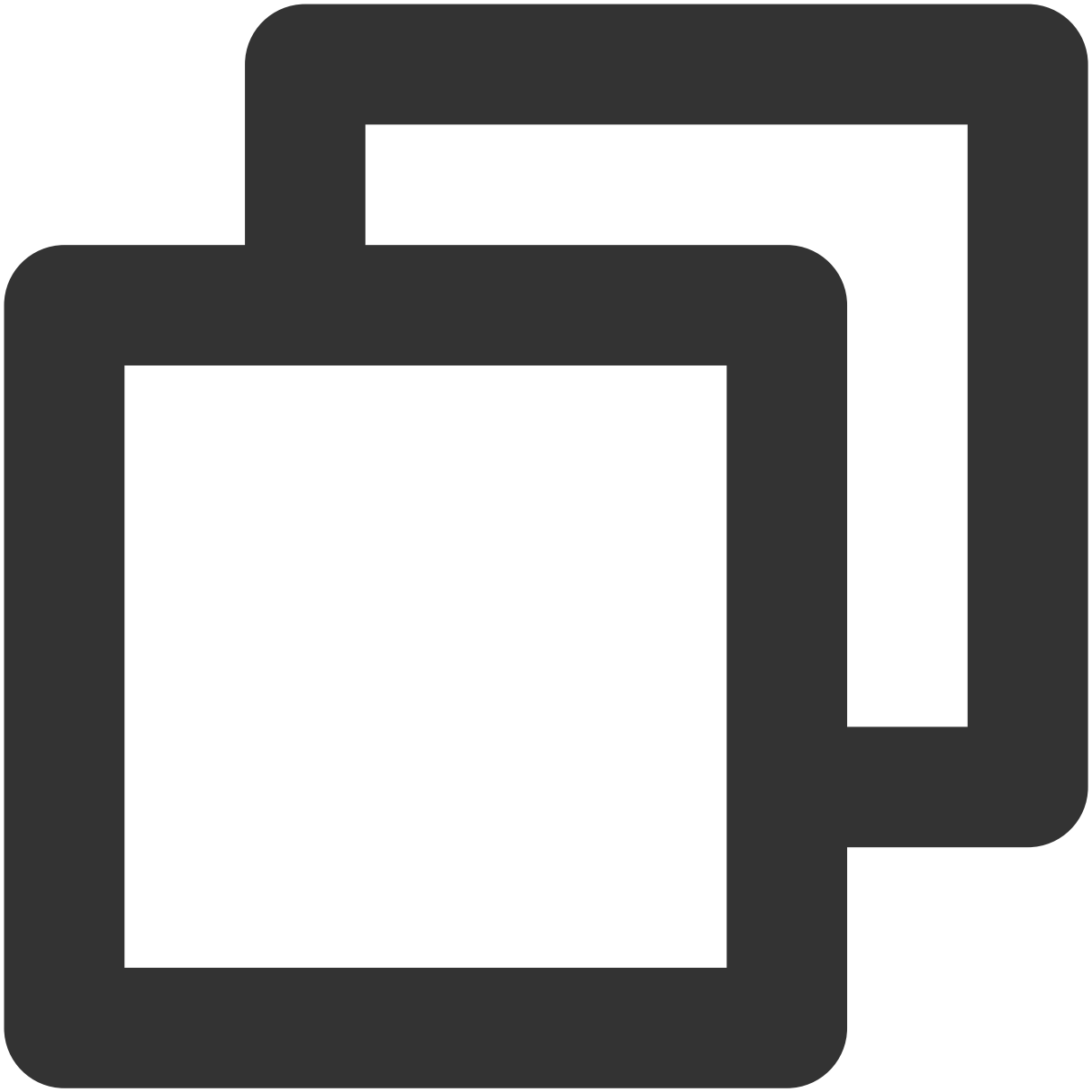
Example:



```
> SELECT year('2016-07-30');
 2016
```

MAKE_DATE

Function statement:



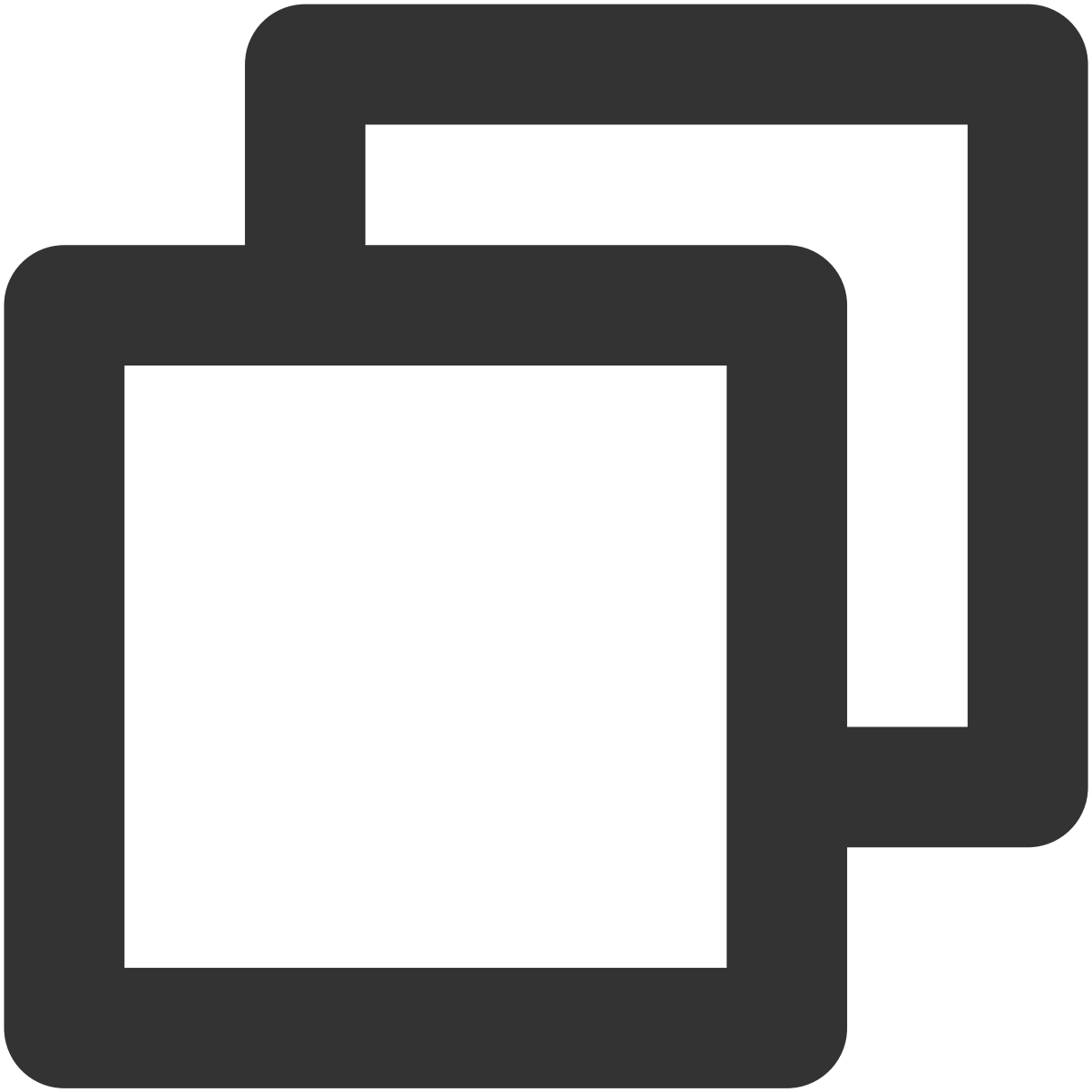
```
MAKE_DATE(<year> integer, <month> integer, <day> integer)
```

Supported engines: SparkSQL and Presto

Usage instructions: Create a date from the year, month, and day fields.

Return type: date

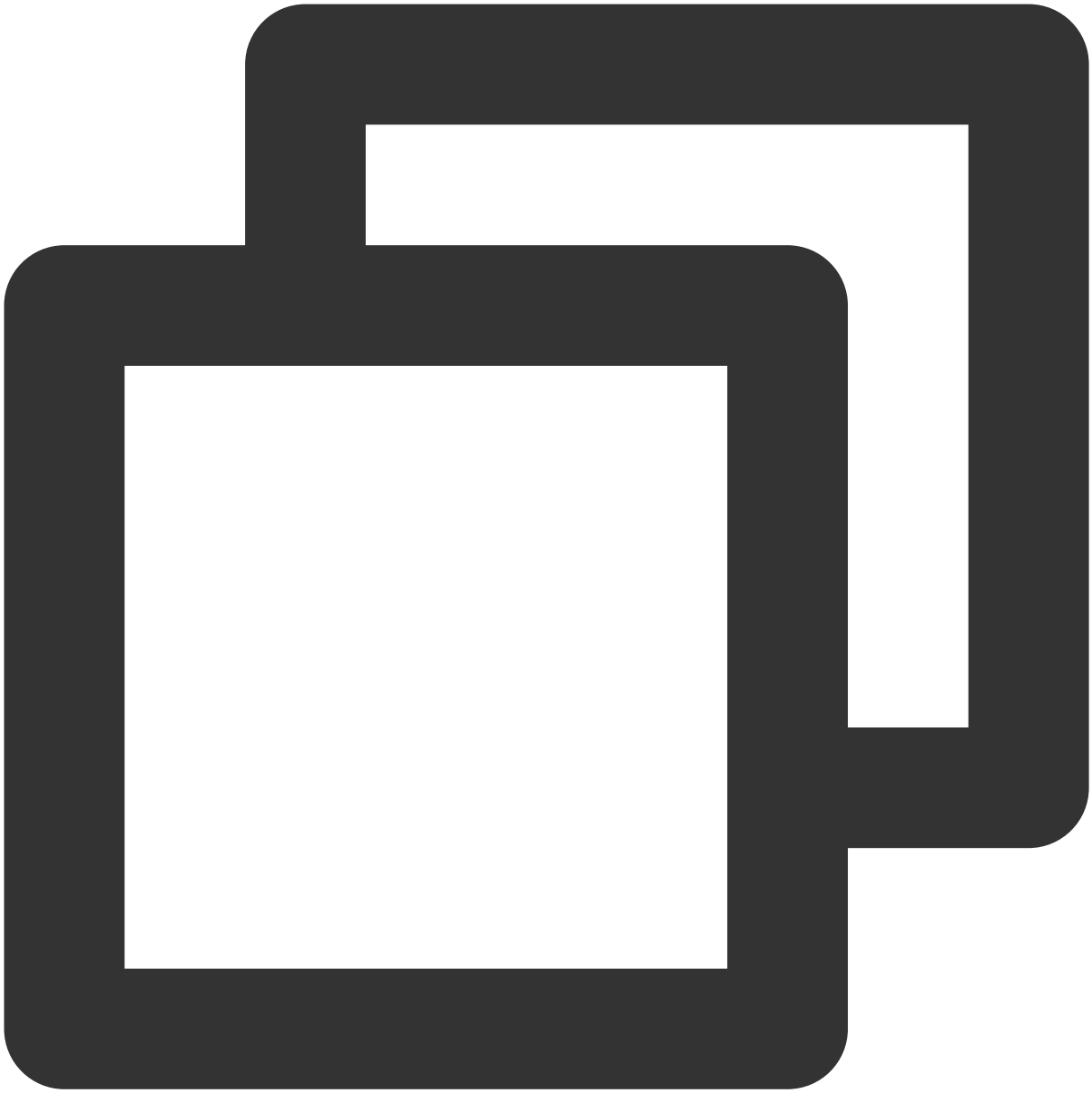
Example:



```
> SELECT make_date(2013, 7, 15);
 2013-07-15
> SELECT make_date(2019, 7, NULL);
 NULL
```

MAKE_TIMESTAMP

Function statement:



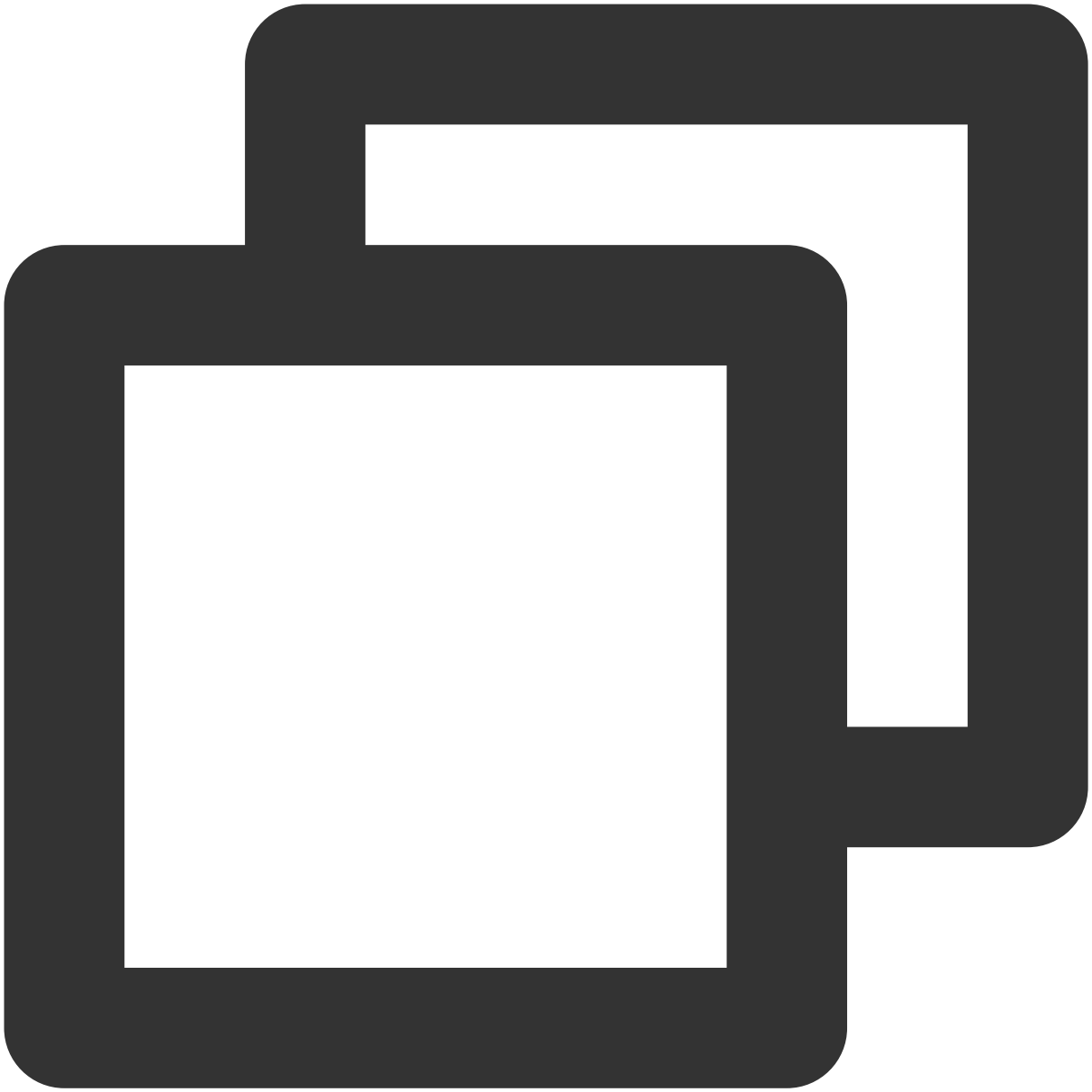
```
MAKE_TIMESTAMP(<year> integer, <month> integer, <day> integer, <hour> integer, <min
```

Supported engines: SparkSQL and Presto

Usage instructions: Create a timestamp according to the given fields.

Return type: timestamp

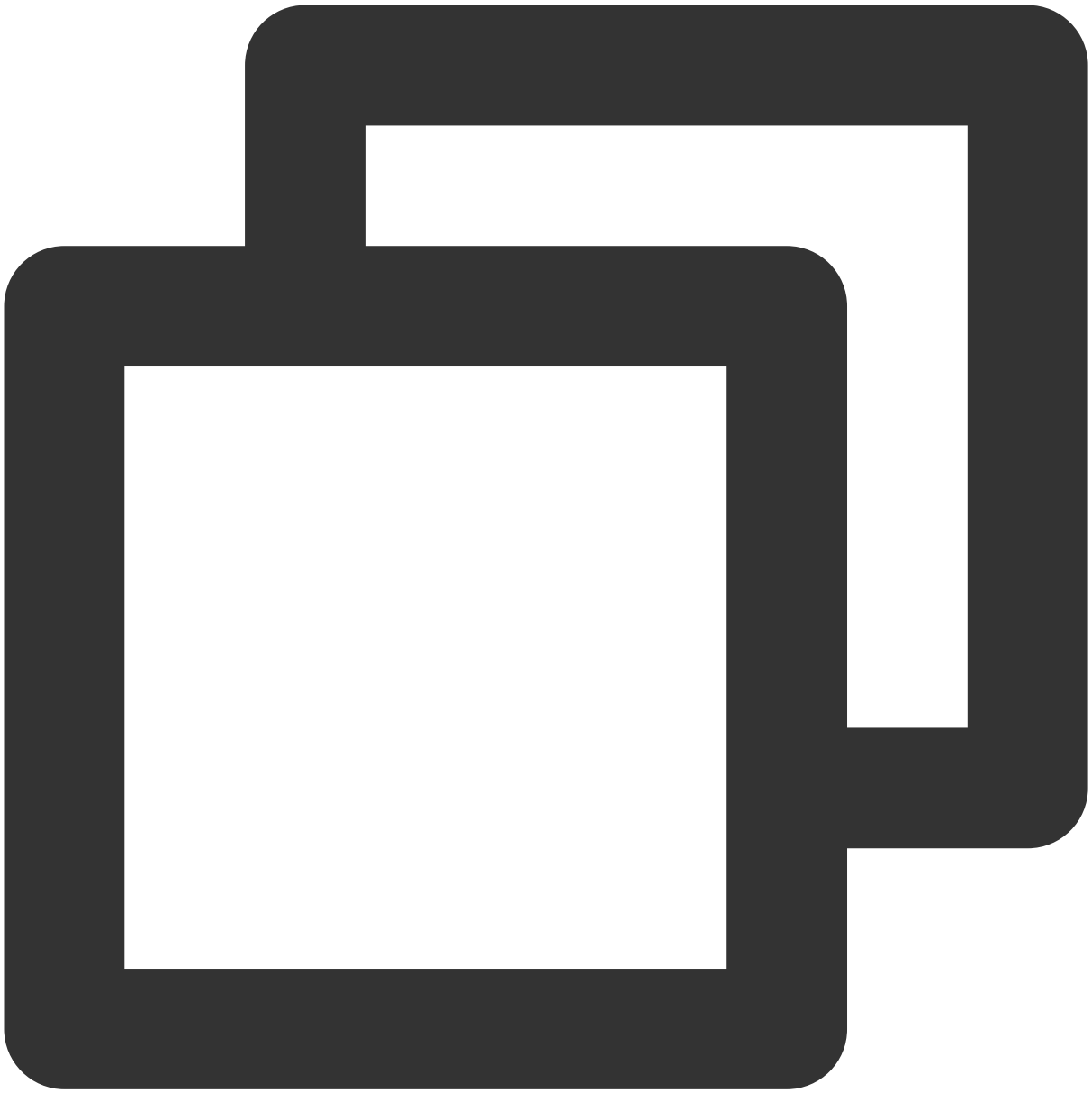
Example:



```
> SELECT make_timestamp(2014, 12, 28, 6, 30, 45.887);
 2014-12-28 06:30:45.887
> SELECT make_timestamp(2014, 12, 28, 6, 30, 45.887, 'CET');
 2014-12-27 21:30:45.887
> SELECT make_timestamp(2019, 6, 30, 23, 59, 60);
 2019-07-01 00:00:00
> SELECT make_timestamp(2019, 6, 30, 23, 59, 1);
 2019-06-30 23:59:01
> SELECT make_timestamp(null, 7, 22, 15, 30, 0);
 NULL
```

DATE_PART

Function statement:



```
DATE_PART(<field> string, <source> date|timestamp)
```

Supported engines: SparkSQL and Presto

Usage instructions: Extract a part of the date/timestamp.

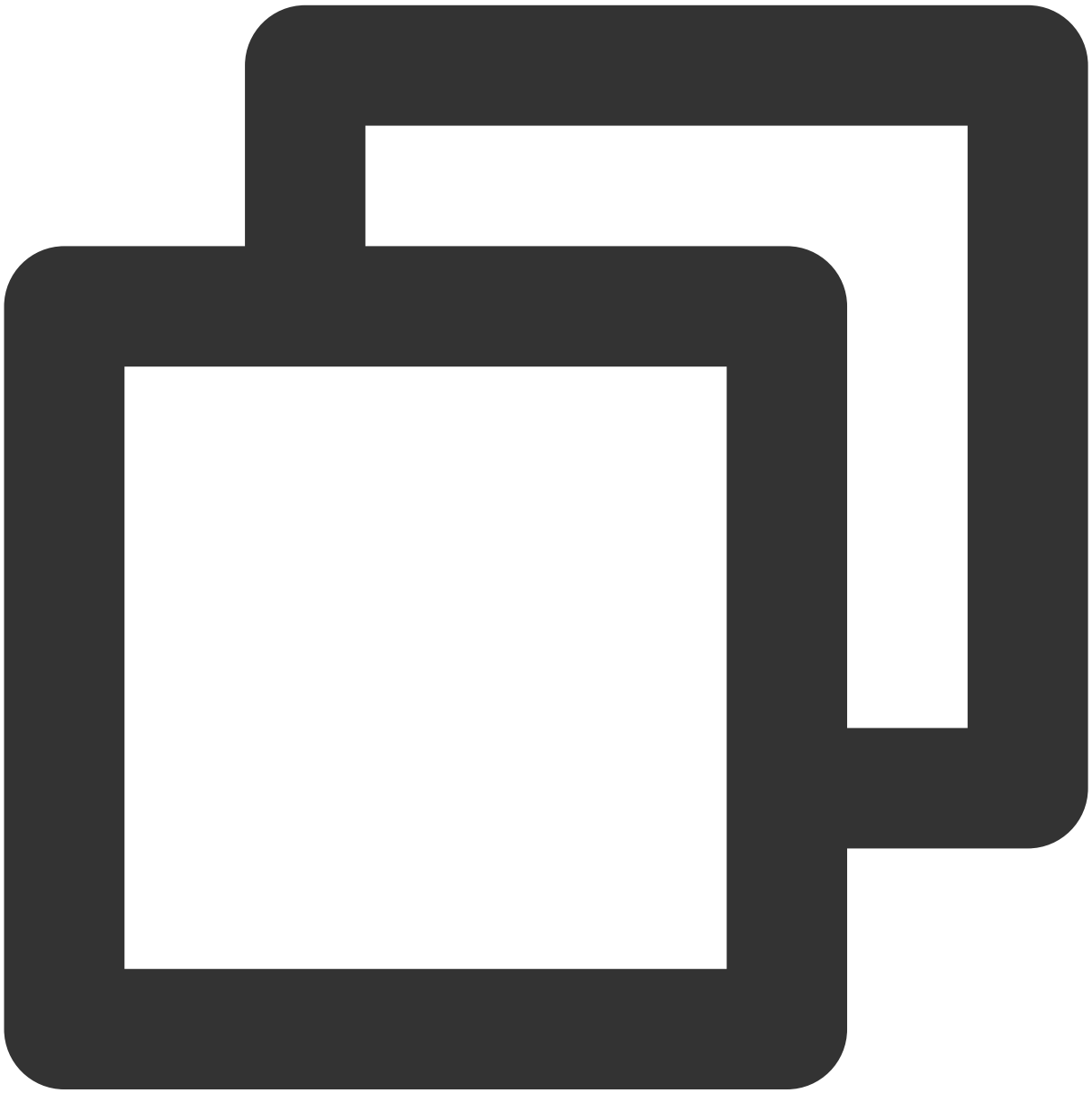
Return type: integer|double

Example:

```
> SELECT date_part('YEAR', TIMESTAMP '2019-08-12 01:00:00.123456');
 2019
> SELECT date_part('week', timestamp '2019-08-12 01:00:00.123456');
 33
> SELECT date_part('doy', DATE'2019-08-12');
 224
> SELECT date_part('SECONDS', timestamp'2019-10-01 00:00:01.000001');
 1.000001
```

DATE_FROM_UNIX_DATE

Function statement:



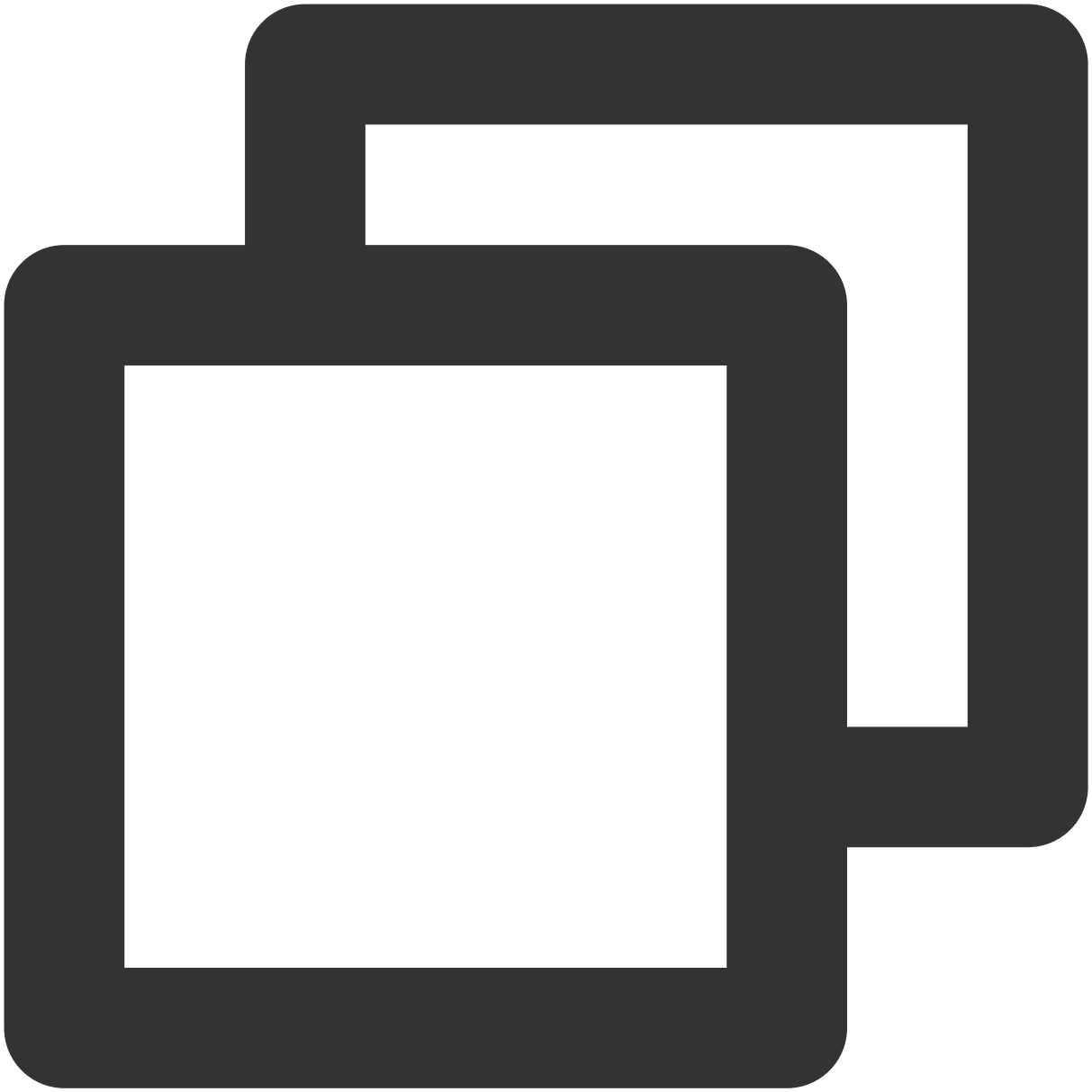
```
DATE_FROM_UNIX_DATE(<unix_timestamp> integer)
```

Supported engines: SparkSQL and Presto

Usage instructions: Create a date based on the number of days since 01-01-1970.

Return type: date

Example:



```
> SELECT date_from_unix_date(1);
 1970-01-02
```

UNIX_DATE

Function statement:

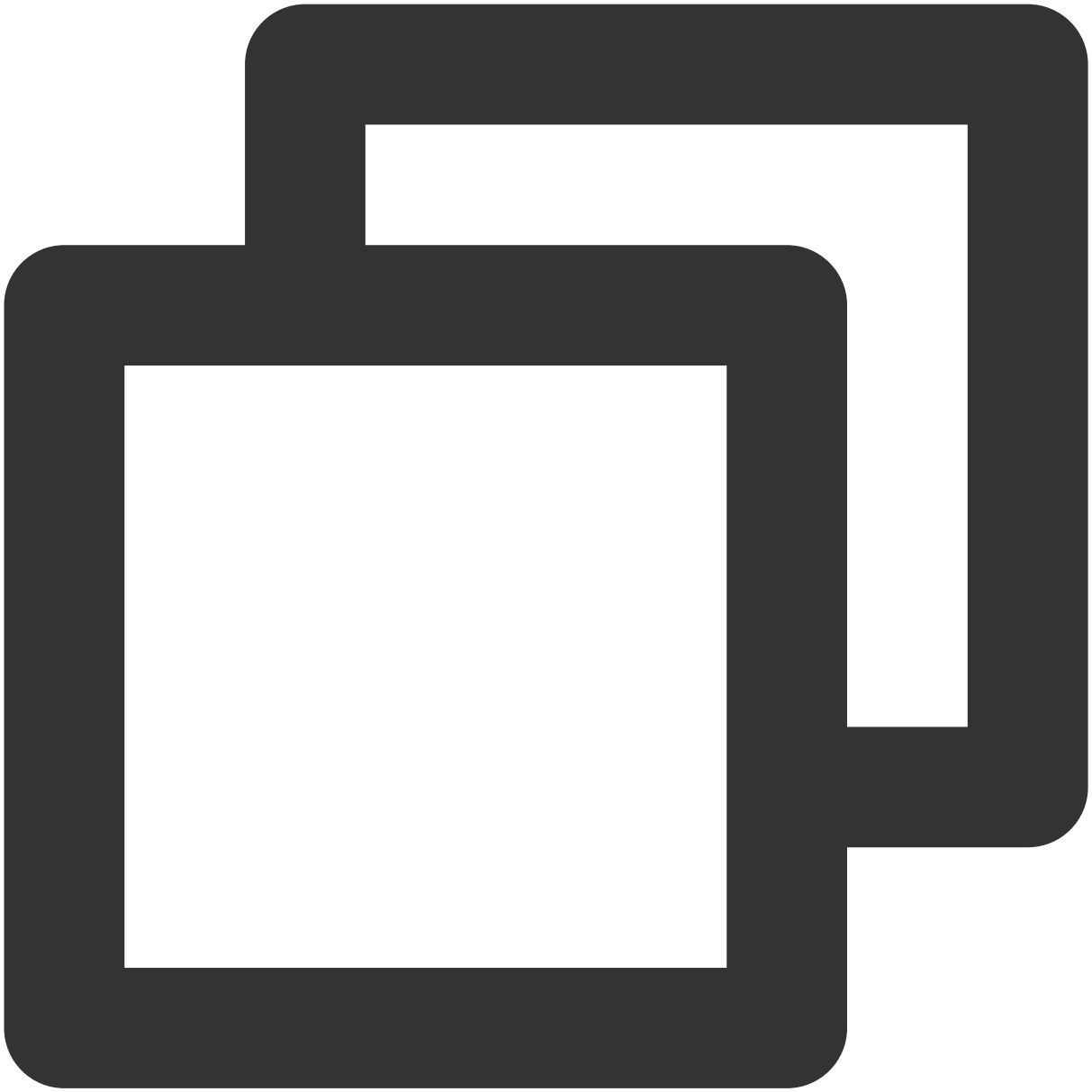
```
UNIX_DATE(<d> date)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the number of days since 01-01-1970.

Return type: integer

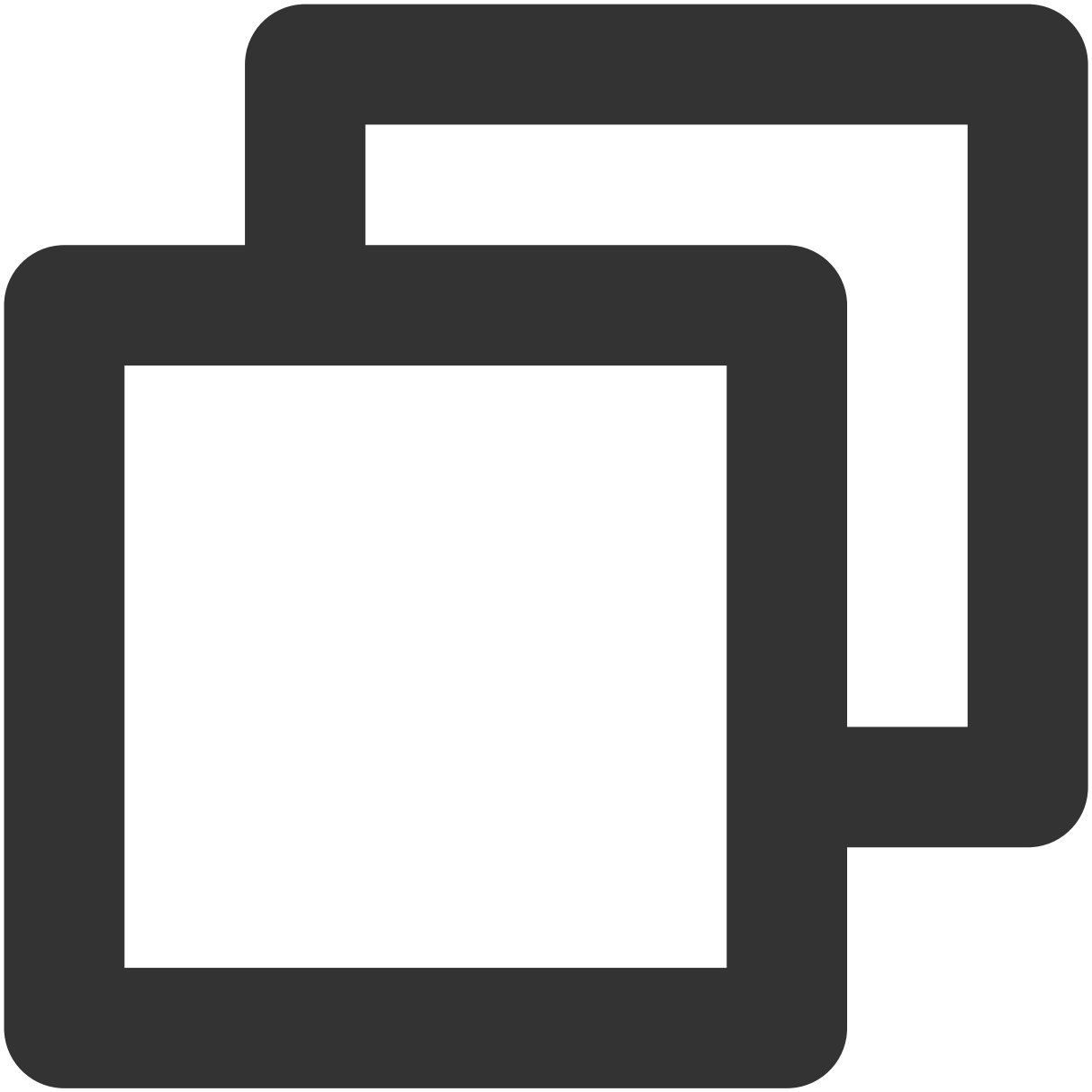
Example:



```
> SELECT unix_date(DATE("1970-01-02"));
 1
```

TIMESTAMP_SECONDS

Function statement:



```
TIMESTAMP_SECONDS(<sec> bigint|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Create a timestamp from the number of seconds since the UTC epoch.

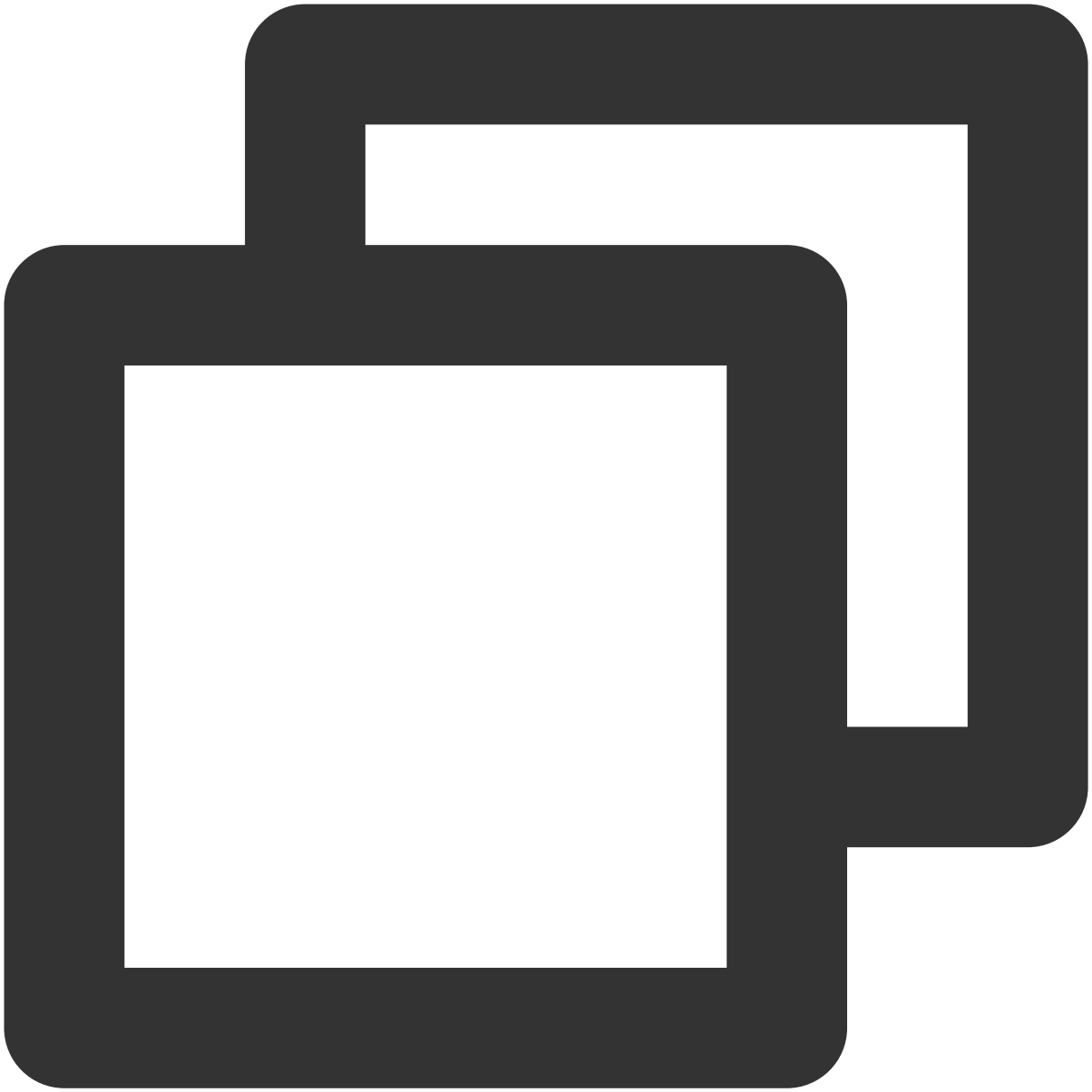
Return type: timestamp

Example:

```
> SELECT timestamp_seconds(1230219000);
 2008-12-25 07:30:00
> SELECT timestamp_seconds(1230219000.123);
 2008-12-25 07:30:00.123
```

TIMESTAMP_MILLIS

Function statement:



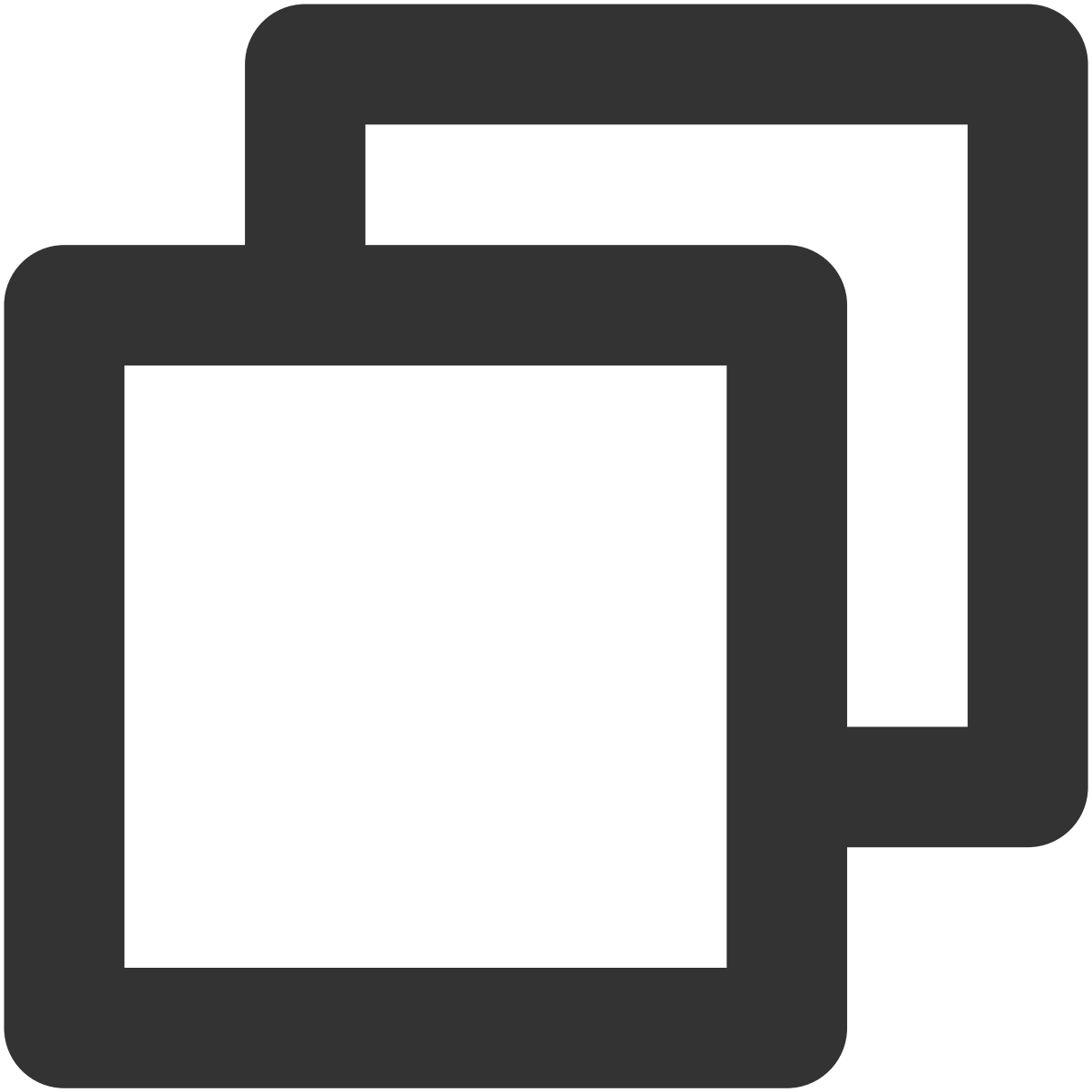
```
TIMESTAMP_MILLIS(<milli> bigint|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Create a timestamp from the number of milliseconds since the UTC epoch.

Return type: timestamp

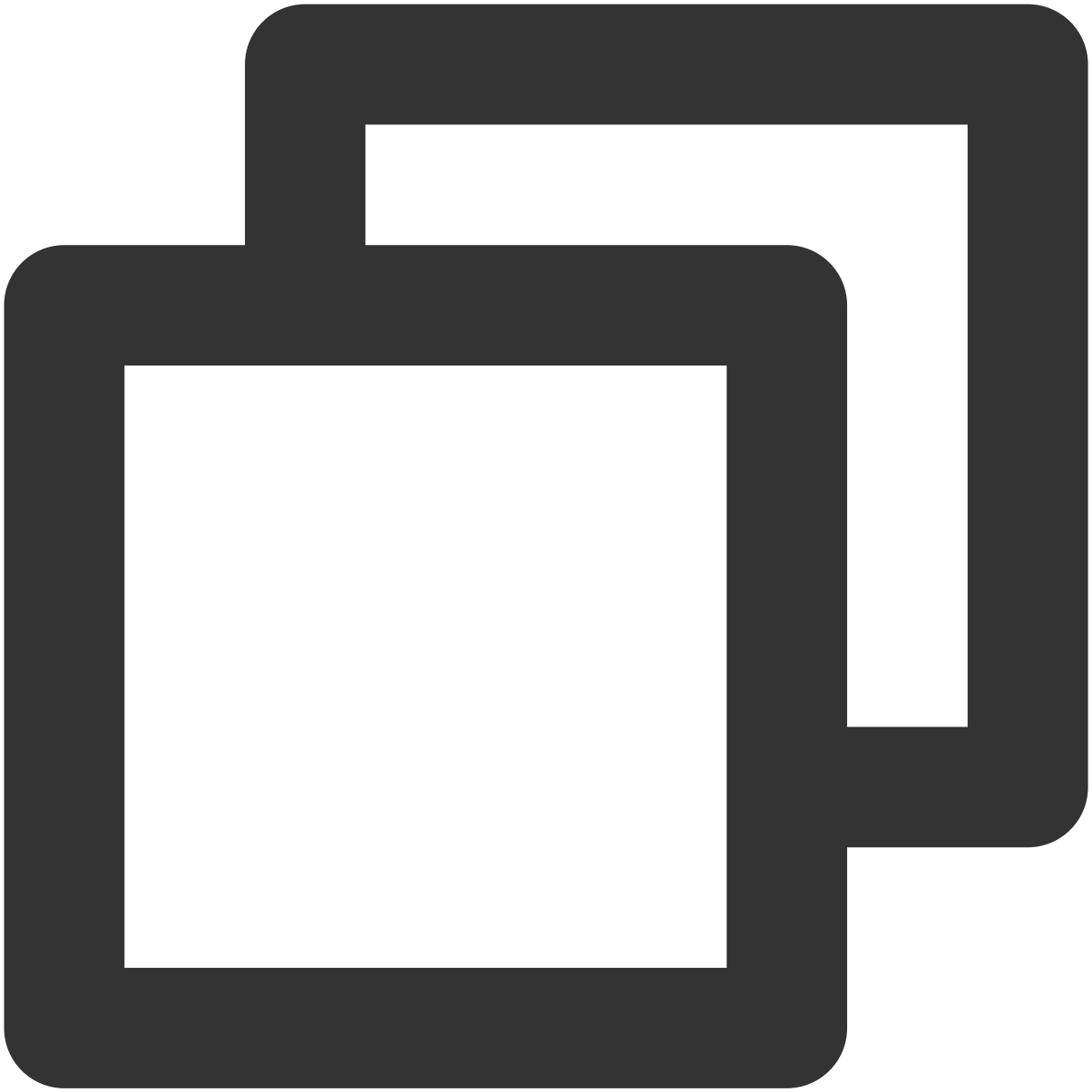
Example:



```
> SELECT timestamp_millis(1230219000123);
 2008-12-25 07:30:00.123
```

TIMESTAMP_MICROS

Function statement:



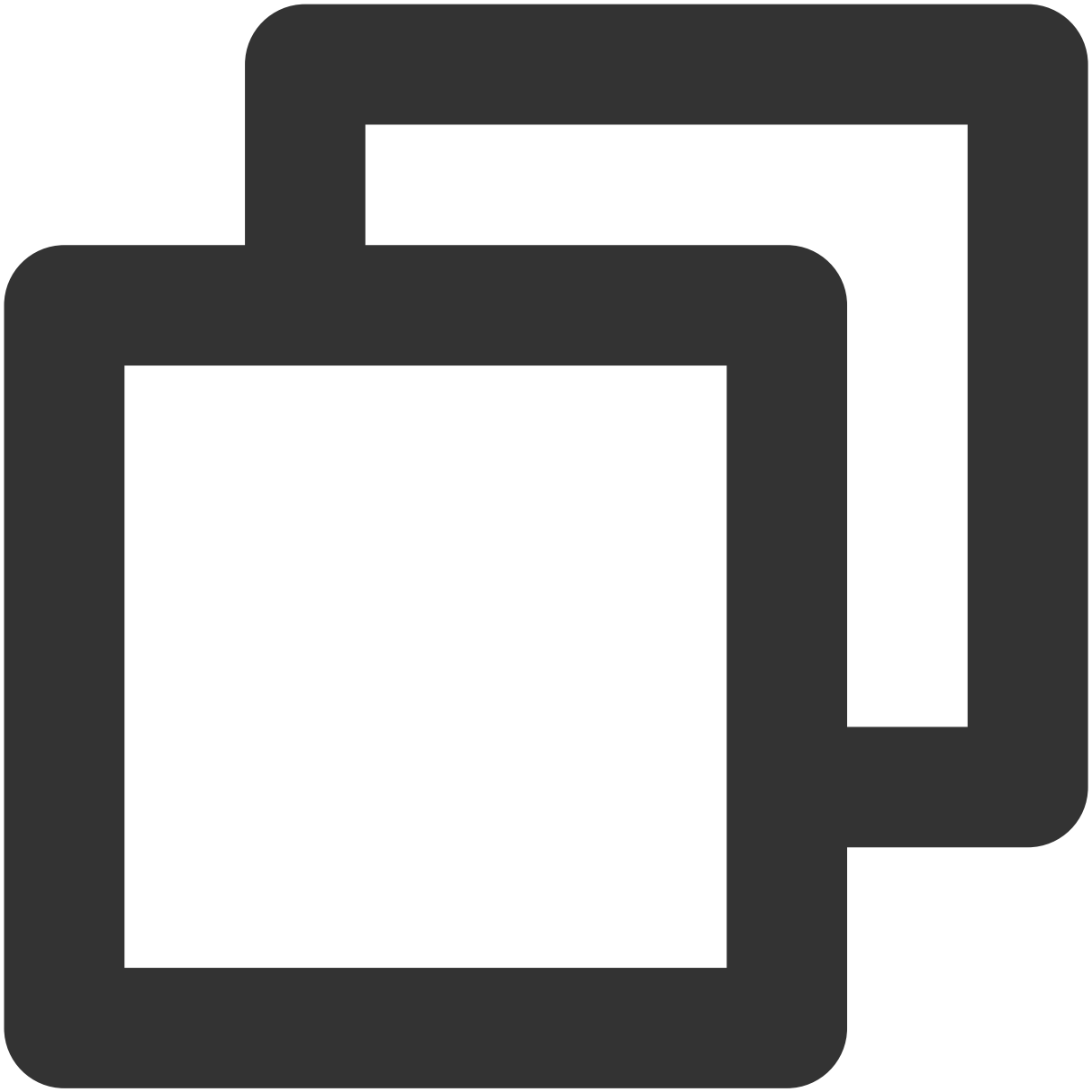
```
TIMESTAMP_MICROS(<micro> bigint)
```

Supported engines: SparkSQL and Presto

Usage instructions: Create a timestamp from the number of milliseconds since the UTC epoch.

Return type: timestamp

Example:



```
> SELECT timestamp_micros(1230219000123123);
 2008-12-25 07:30:00.123123
```

UNIX_SECONDS

Function statement:

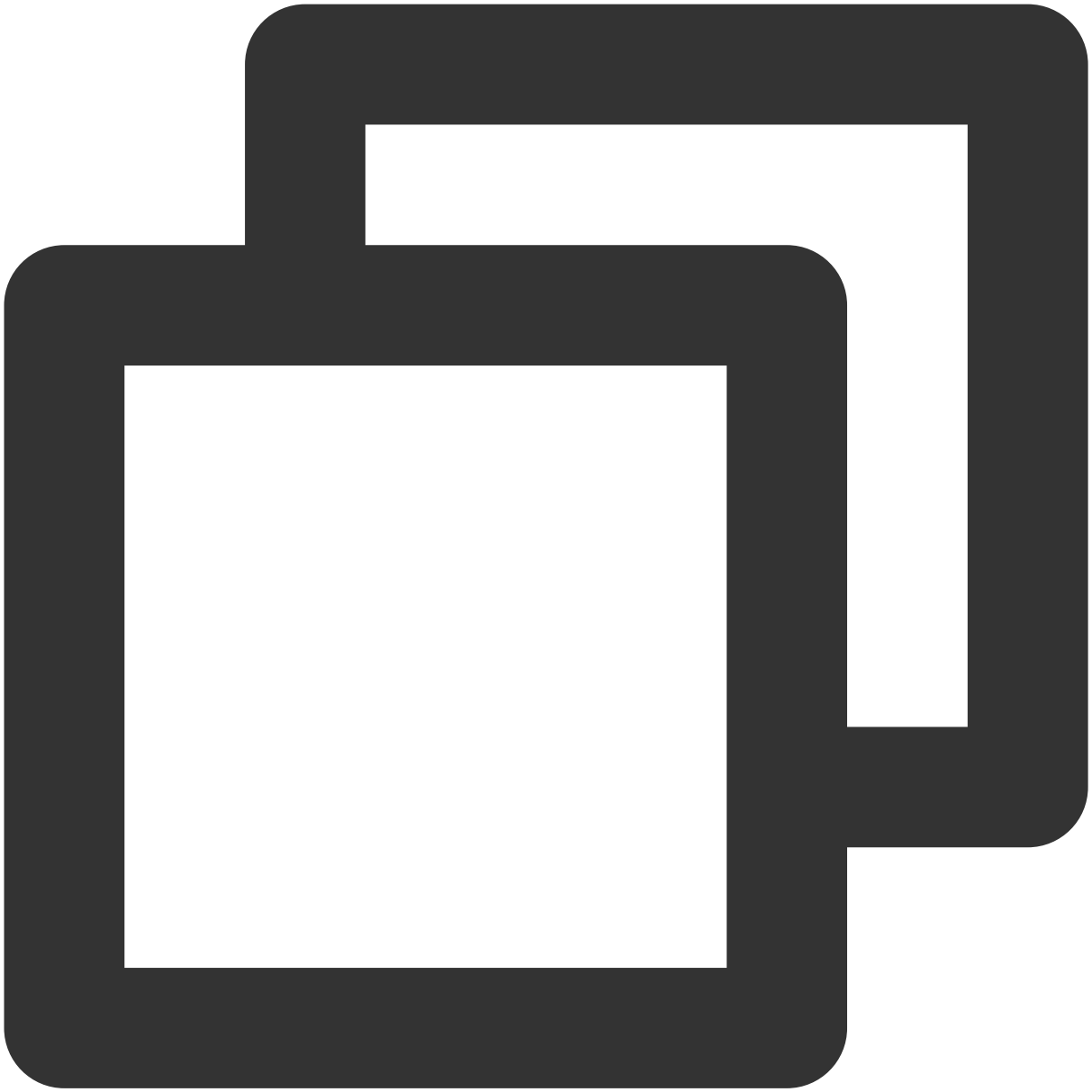
```
UNIX_SECONDS(<ts> timestamp)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the number of seconds since 01-01-1970 00:00:00 UTC.

Return type: bigint

Example:



```
> SELECT unix_seconds(TIMESTAMP('1970-01-01 00:00:01Z'));
 1
```

UNIX_MILLIS

Function statement:

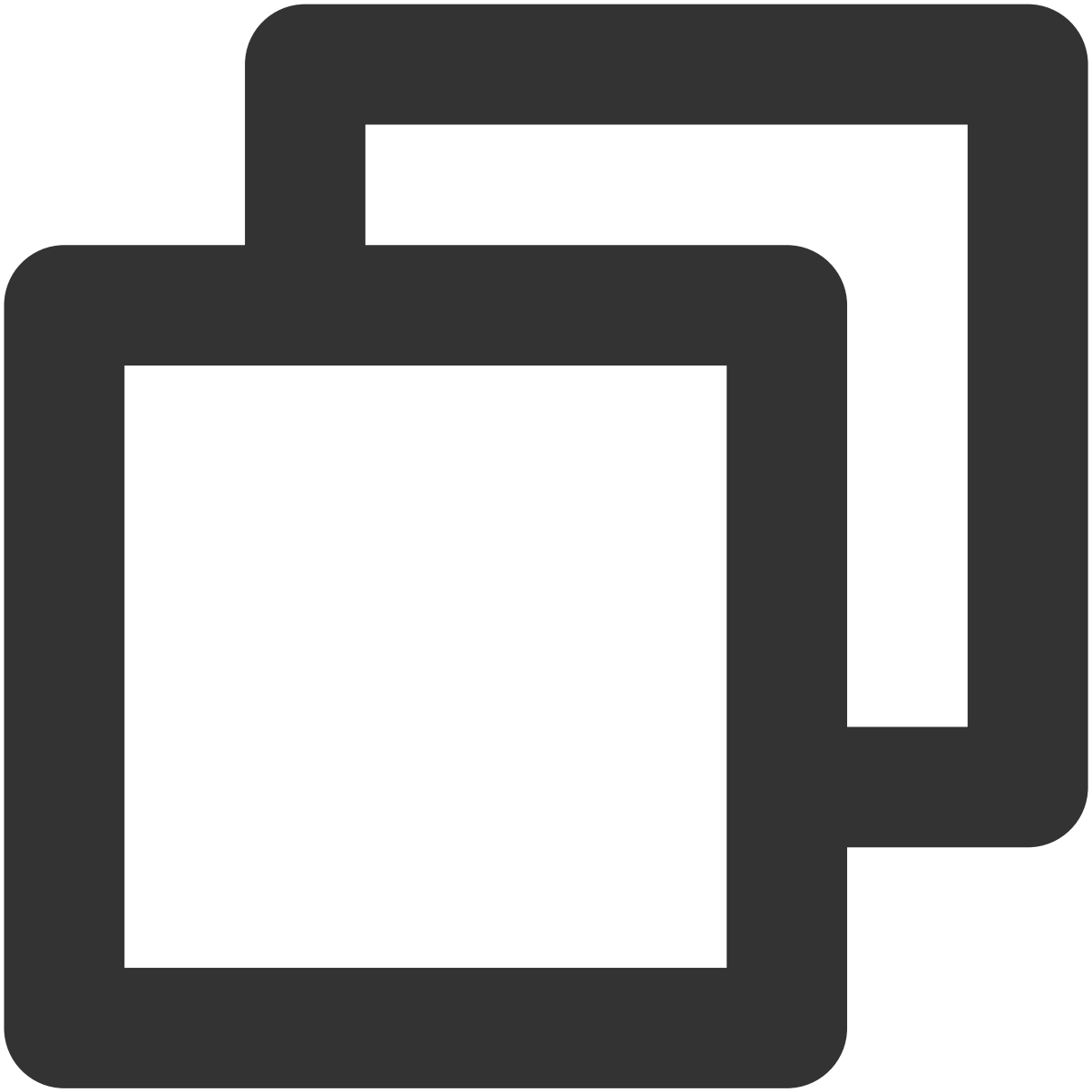
```
UNIX_MILLIS(<ts> timestamp)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the number of milliseconds since 01-01-1970 00:00:00 UTC.

Return type: bigint

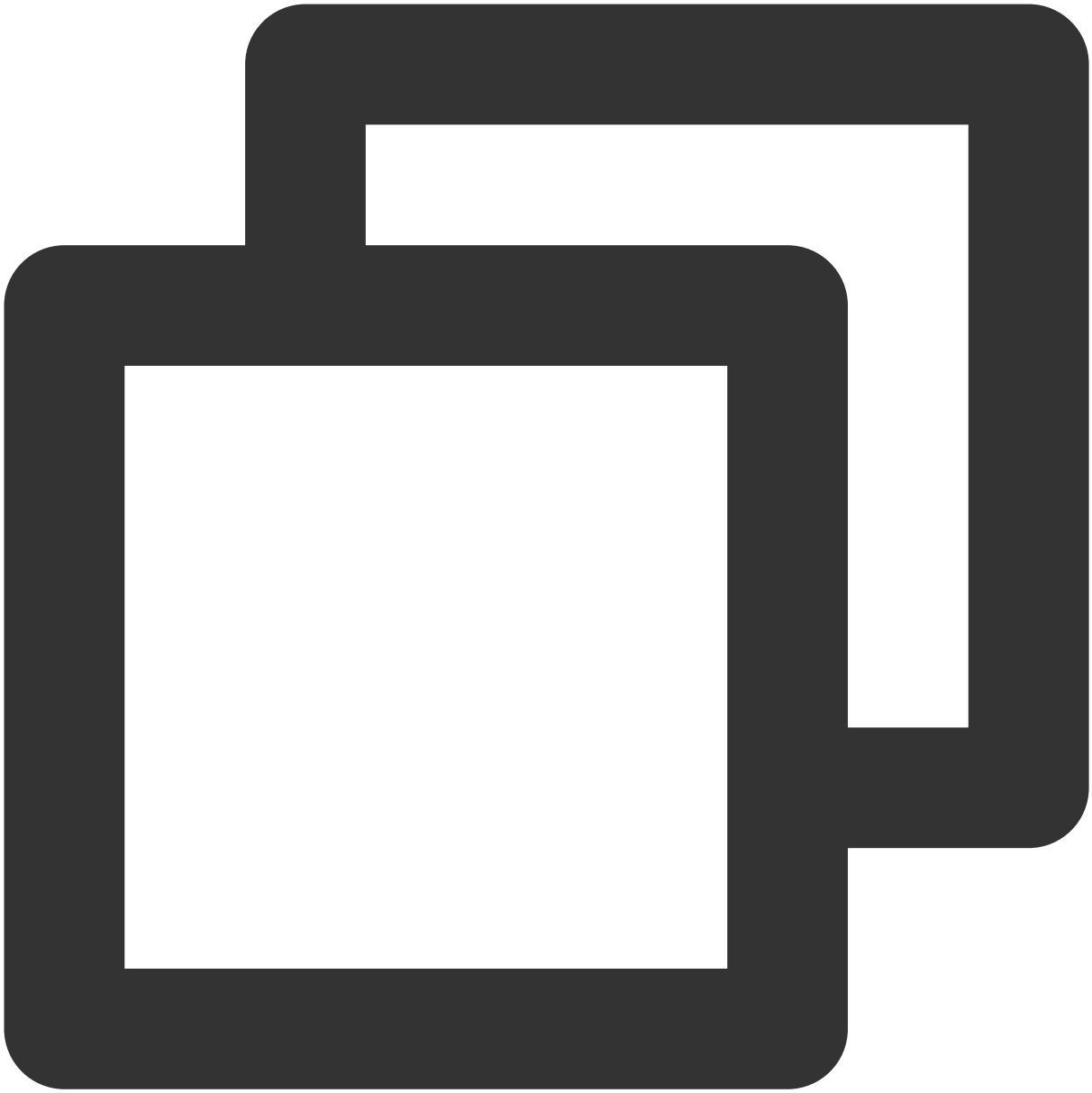
Example:



```
> SELECT unix_millis(TIMESTAMP('1970-01-01 00:00:01Z'));
 1000
```

UNIX_MICROS

Function statement:



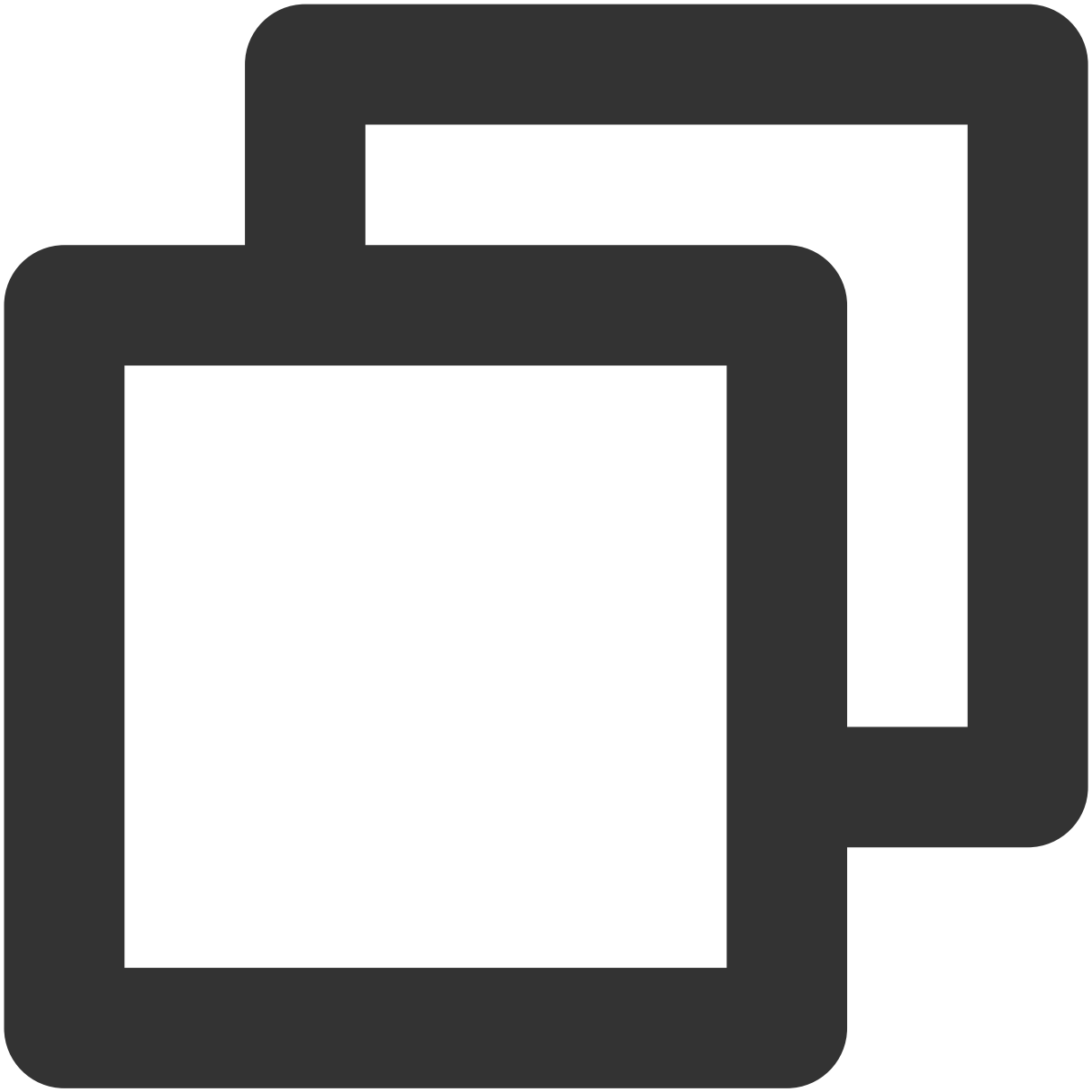
```
UNIX_MICROS(<ts> timestamp)
```

Supported engine: SparkSQL

Usage instructions: Return the number of microseconds since 01-01-1970 00:00:00 UTC.

Return type: bigint

Example:



```
> SELECT unix_micros(timestamp '1970 00:00:00.001')
 1000
```

DATEADD/TIMESTAMP_ADD

Function statement:

```
TIMESTAMP_ADD(<date|timestamp> dt, <int> delta, <string> pattern)
TIMESTAMP_ADD(<date|timestamp> dt, <int> delta, <string> pattern)
```

Supported engine: SparkSQL

Engine version requirement: An engine purchased on or after December 7, 2023, or upgrade to the latest engine.

Usage instructions: Calculate the final time based on the specified unit and offset on the given date or time.

dt: Input date or time format. If it is null, null is returned.

delta: Required, int type. If it is null, null is returned.

pattern: Required, string type, unit. Pattern supports:

Year (standard y, compatible with -year and yyyy)

Month (standard M, compatible with -month, -mon, and MM)

Day (standard d, compatible with -day and dd)

Hour (standard H, compatible with -hour, hh, HH, and h)

Minute (standard m, compatible with mi and mm)

Second (standard s, compatible with ss)

Microsecond (standard S, compatible with SSS)

Other types are not allowed, otherwise an error will occur.

Note:

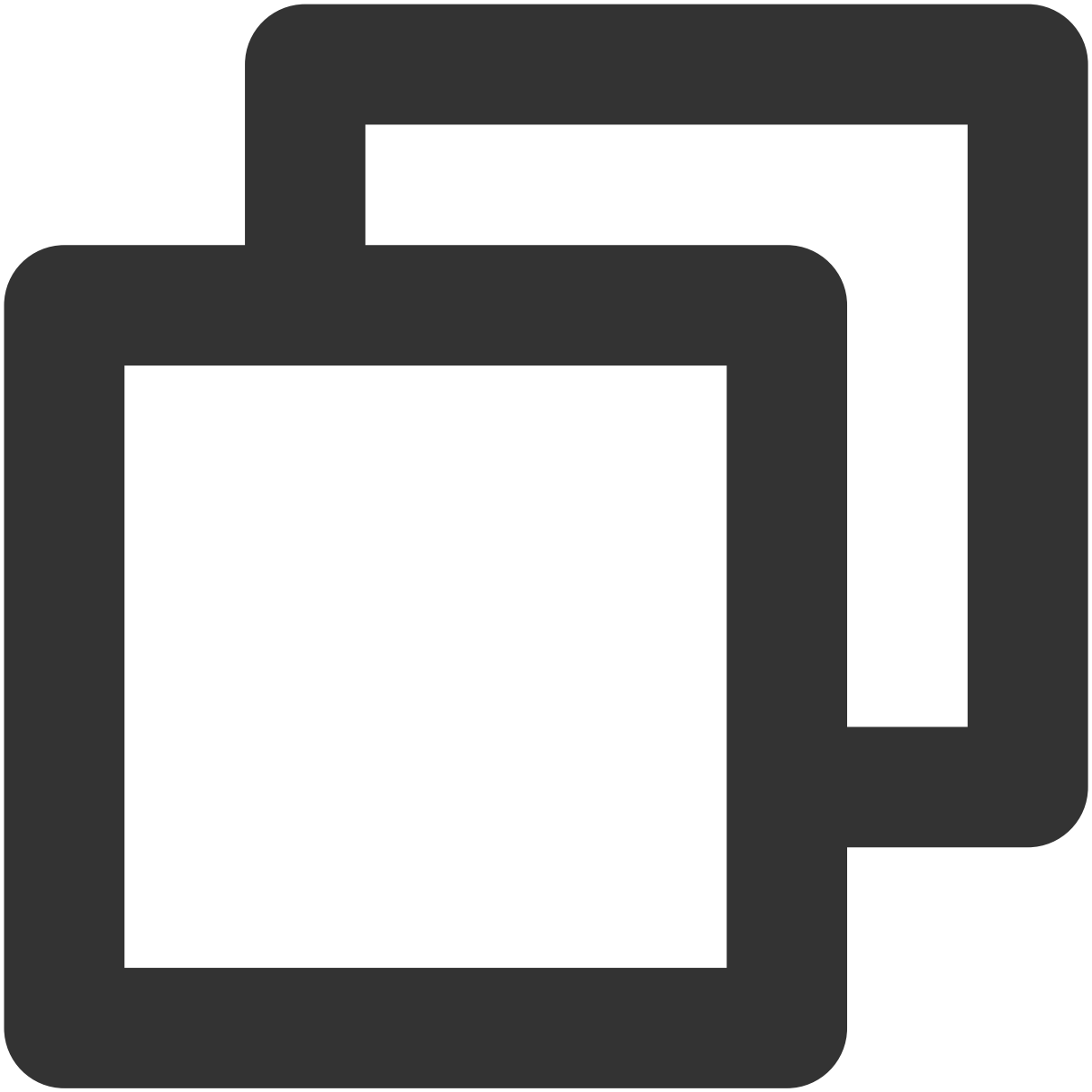
When the unit of delta is months, if adding the delta value to the month part does not cause a Day overflow, the Day value remains unchanged; otherwise, the Day value is set to the last day of the resulting month.

Note that the output time is in the default timezone.

Difference between the units for month M and minute m; supports both 12-hour format h and 24-hour format H, but it is recommended to use H.

Return type: timestamp

Example:



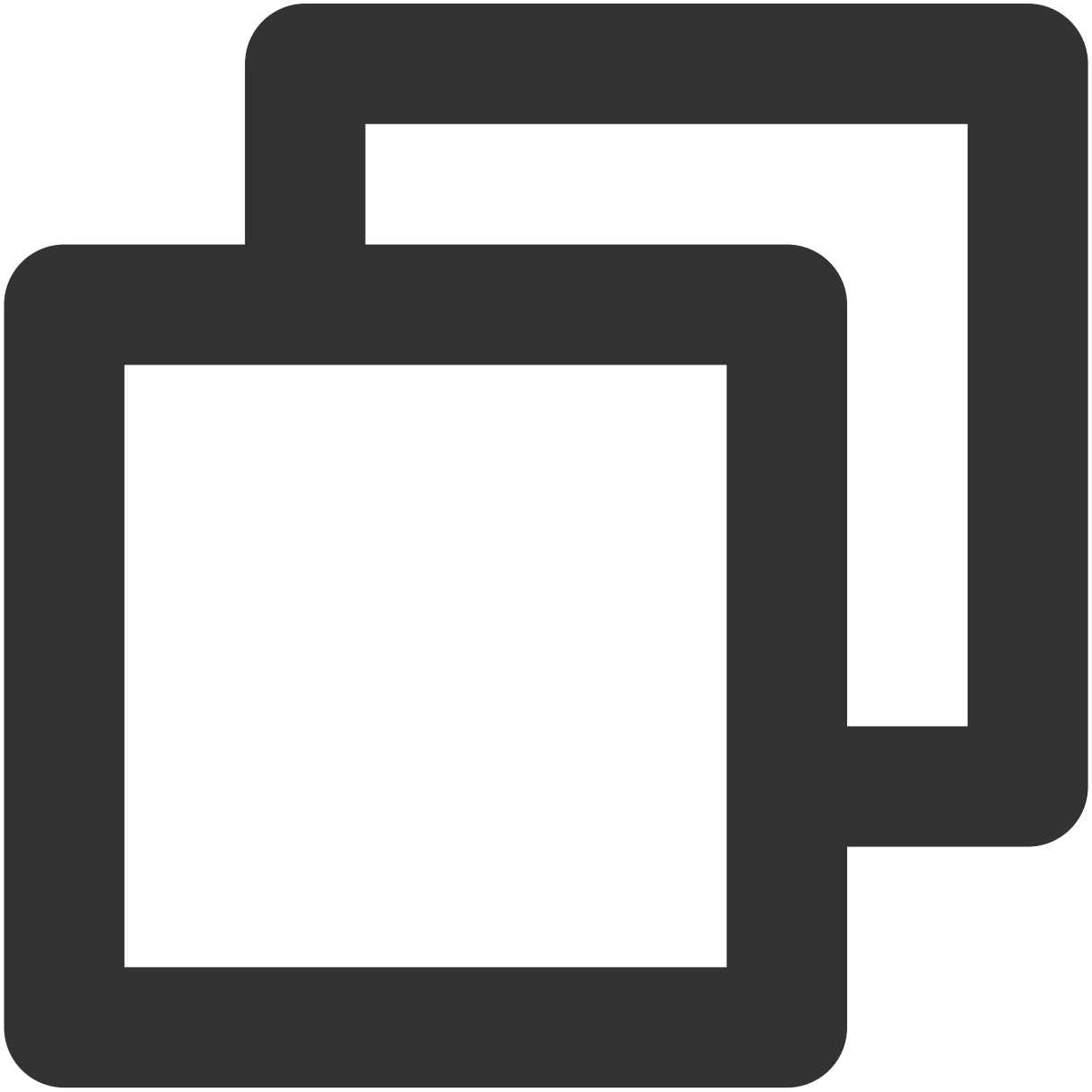
```
> SELECT timestamp_add(date '2016-12-31', -1, 'M');
2016-11-30T00:00:00.000+08:00
> SELECT timestamp_add(timestamp '2016-12-31 00:12:00', 1, 'm');
2016-12-31T00:13:00.000+08:00
> SELECT timestamp_add(timestamp '2016-12-31 00:00:00', -1, 'm');
2016-12-30T23:59:00.000+08:00
```

JSON Functions

Last updated：2024-08-07 17:32:18

GET_JSON_OBJECT

Function statement:



```
GET_JSON_OBJECT(<json> string, <path> string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Extract JSON object.

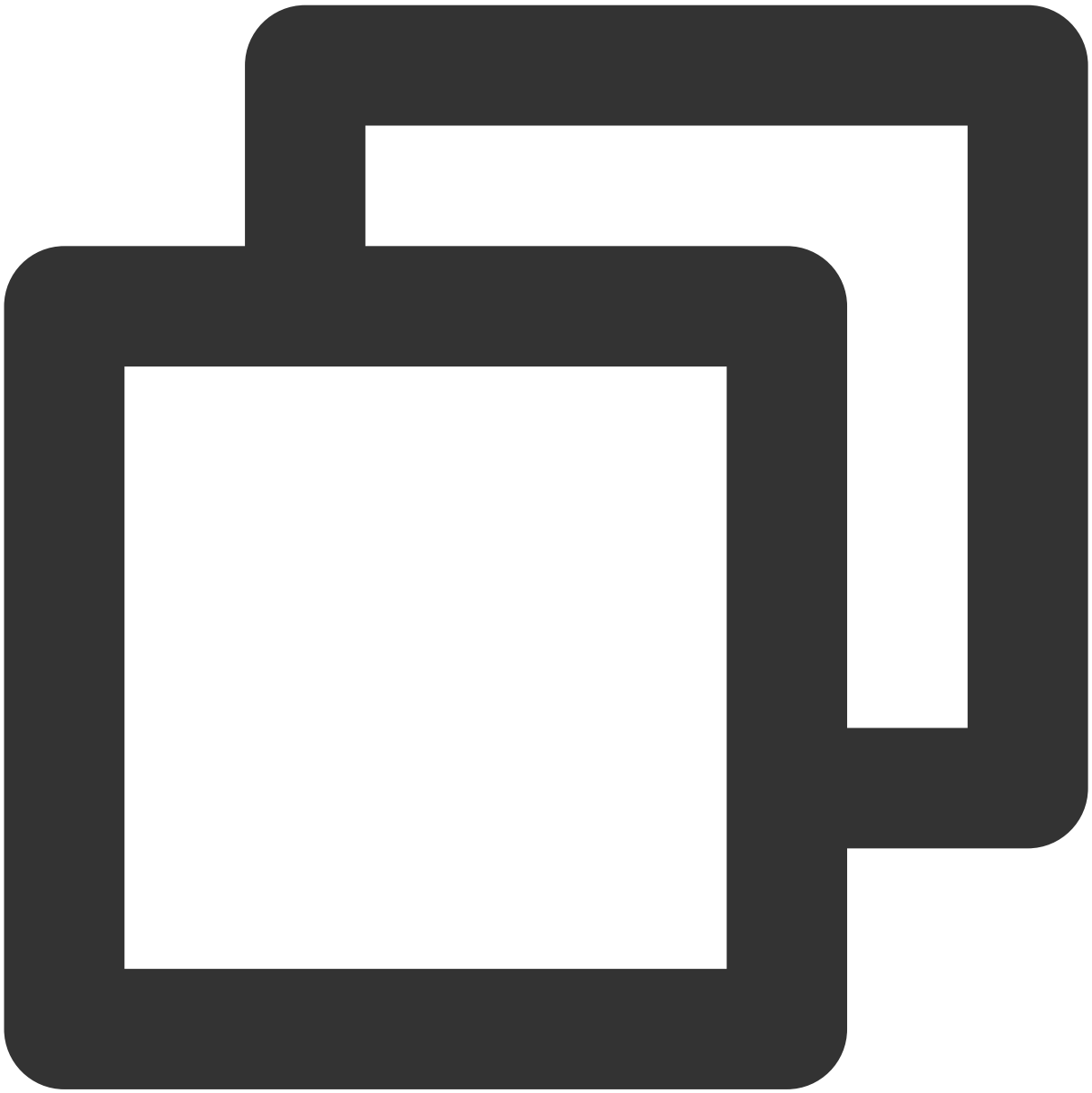
Return type: string

Example:

```
> SELECT get_json_object('{"a":"b"}', '$.a');
 b
```

JSON_TUPLE

Function statement:



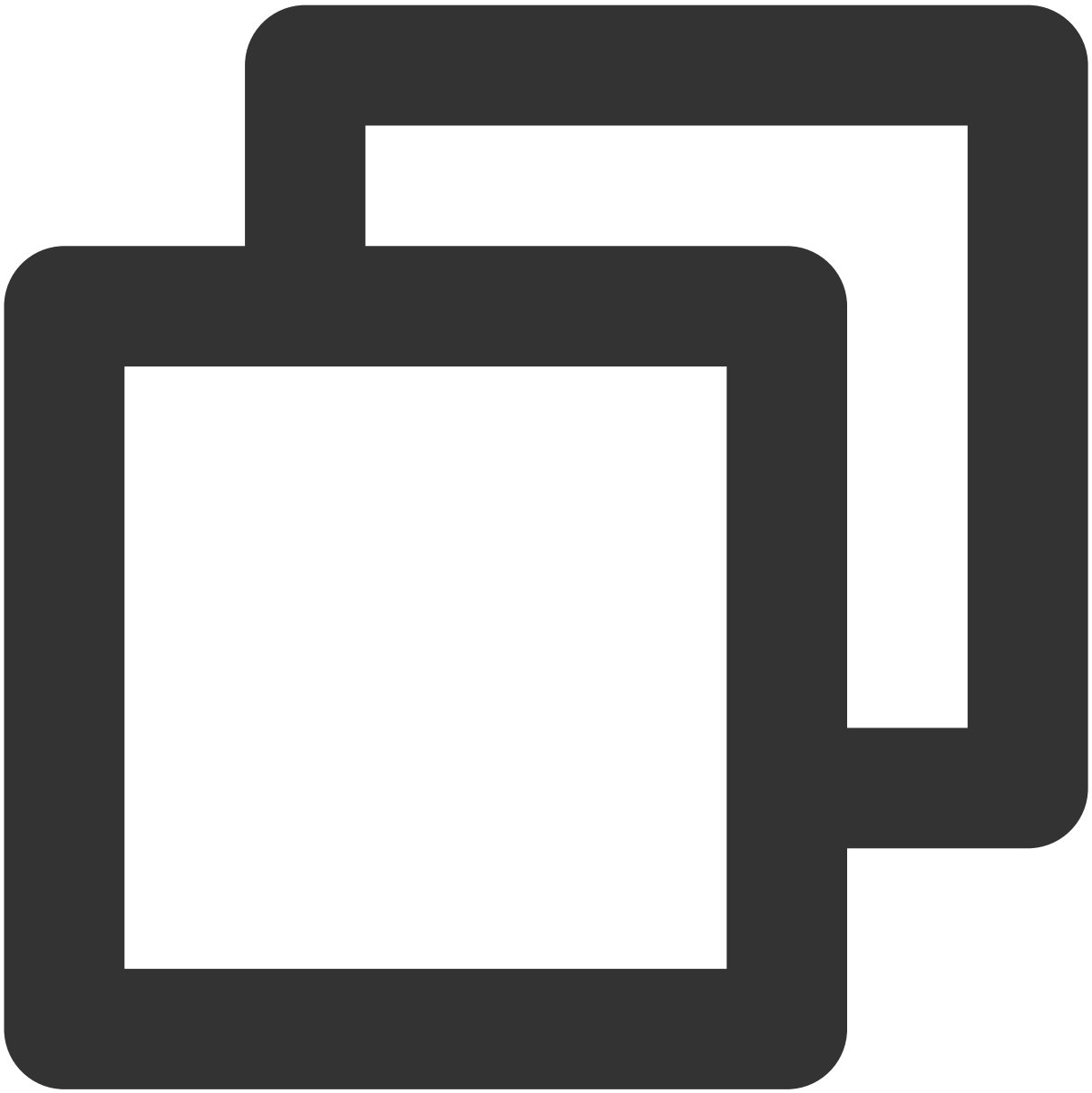
```
JSON_TUPLE(<json> string, <p1> string, ..., <pn> string)
```

Supported engine: SparkSQL

Usage instructions: Return a tuple similar to the function get_json_object, but it requires multiple input names. All input parameters and output column types are strings.

Return type: struct<string, ..., string>

Example:



```
> SELECT json_tuple('{"a":1, "b":2}', 'a', 'b');
 1  2
```

TO_JSON

Function statement:

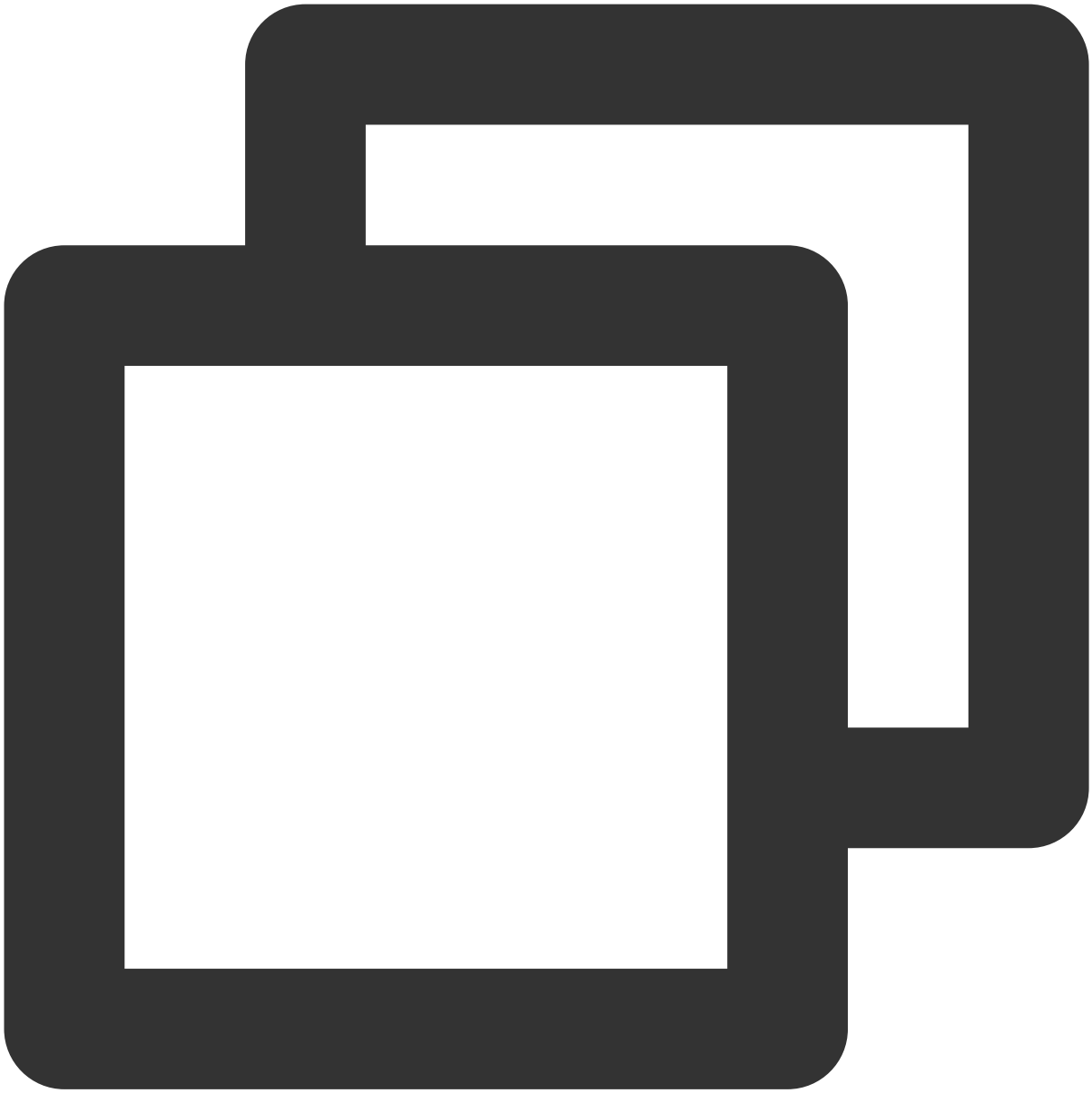
```
TO_JSON(<expr> struct|map|array[, <option> map])
```

Supported engines: SparkSQL and Presto

Usage instructions: Return a JSON string with the given structured value.

Return type: string

Example:

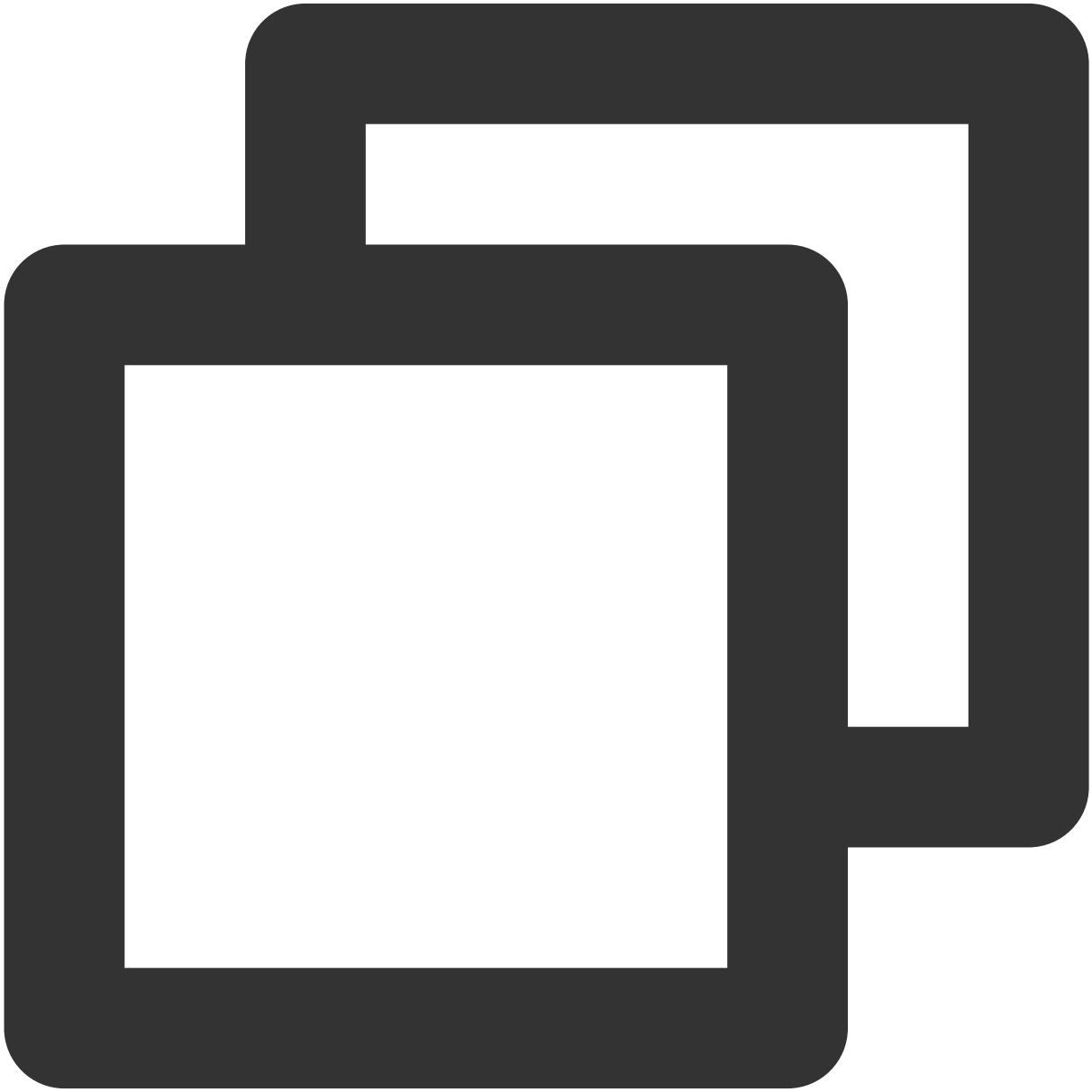


```
> SELECT to_json(named_struct('a', 1, 'b', 2));
 {"a":1,"b":2}
> SELECT to_json(named_struct('time', to_timestamp('2015-08-26', 'yyyy-MM-dd')), ma
 {"time":"26/08/2015"}
> SELECT to_json(array(named_struct('a', 1, 'b', 2)));
 [{"a":1,"b":2}]
> SELECT to_json(map('a', named_struct('b', 1)));
 {"a":{"b":1}}
> SELECT to_json(map(named_struct('a', 1),named_struct('b', 2)));
 {"[1]":{"b":2}}
> SELECT to_json(map('a', 1));
```

```
 {"a":1}
> SELECT to_json(array((map('a', 1))));
 [{"a":1}]
```

FROM_JSON

Function statement:



```
FROM_JSON(<json> string, <schema> string[, <options> map<string, string>])
```

Supported engines: SparkSQL and Presto

Usage instructions: Return a structured value with the given jsonStr and schema.

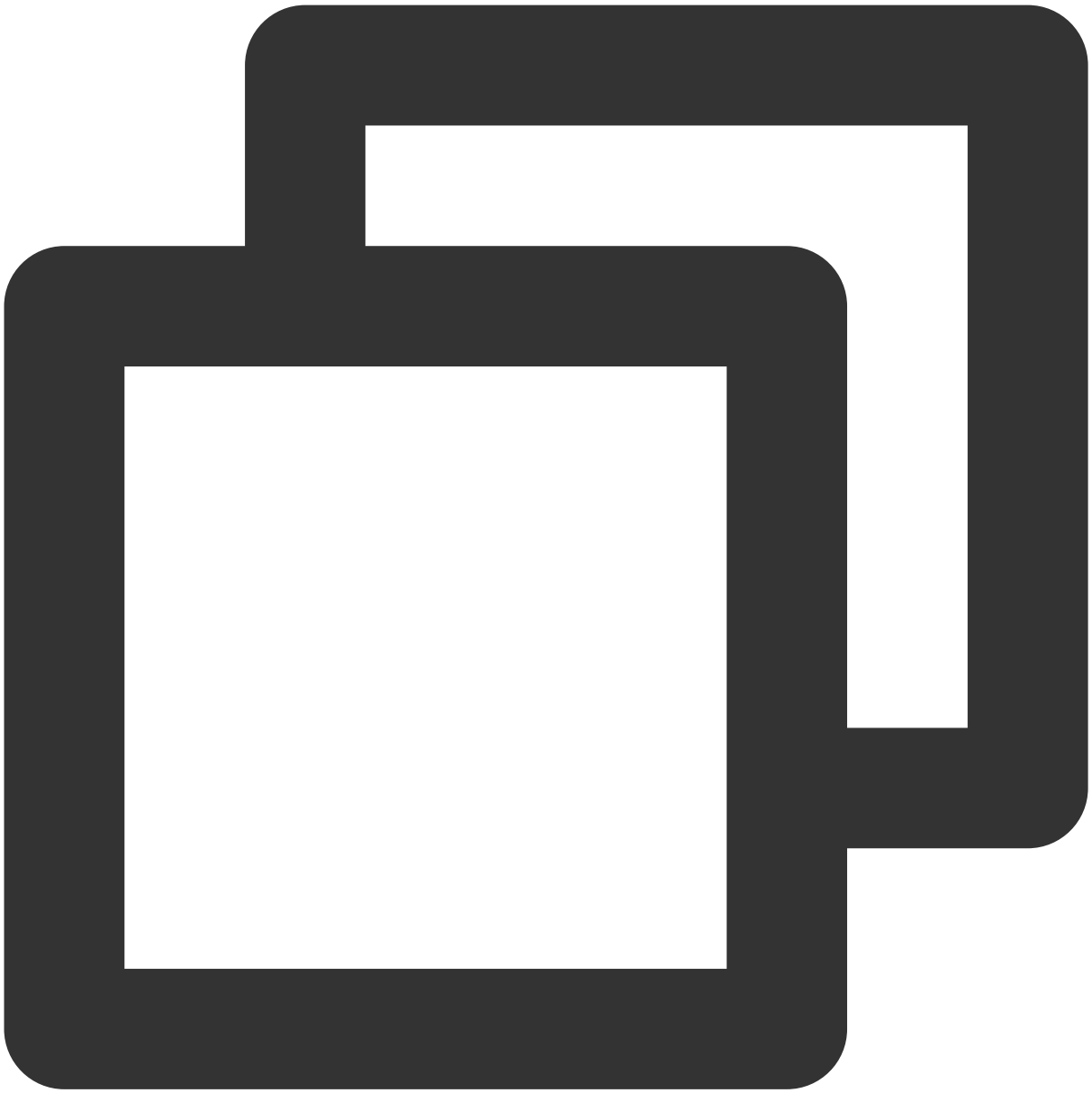
Return type: struct

Example:

```
> SELECT from_json('{"a":1, "b":0.8}', 'a INT, b DOUBLE');
 {"a":1,"b":0.8}
> SELECT from_json('{"time":"26/08/2015"}', 'time Timestamp', map('timestampFormat'
 {"time":2015-08-26 00:00:00}
```

SCHEMA_OF_JSON

Function statement:



```
SCHEMA_OF_JSON(<json> string[, <options> map<string, string>])
```

Supported engine: SparkSQL

Usage instructions: Return the DDL-formatted schema of the JSON string.

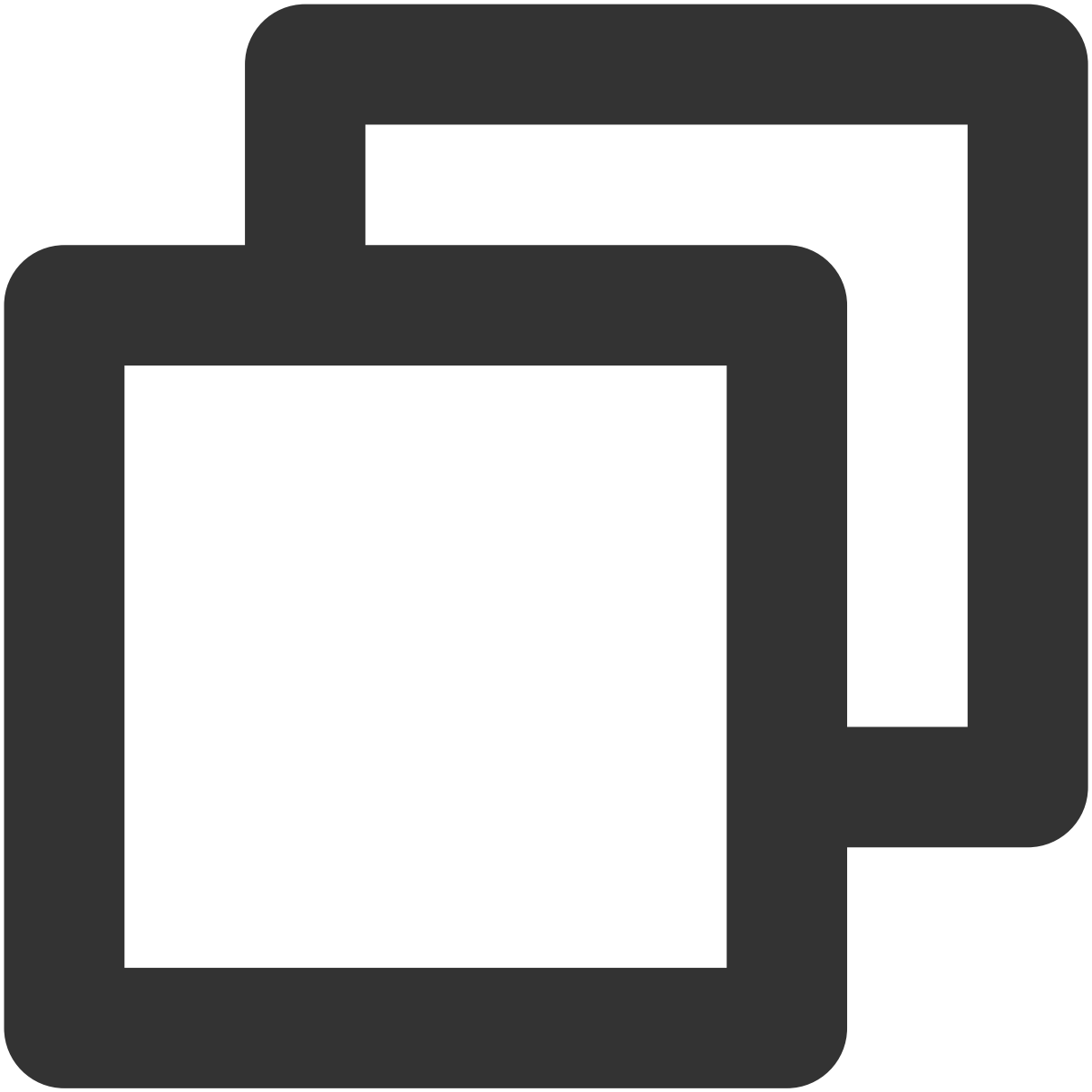
Return type: string

Example:

```
> SELECT schema_of_json('[{"col":0}]');
 ARRAY<STRUCT<`col`: BIGINT>>
> SELECT schema_of_json('[{"col":01}]', map('allowNumericLeadingZeros', 'true'));
 ARRAY<STRUCT<`col`: BIGINT>>
```

JSON_ARRAY_LENGTH

Function statement:



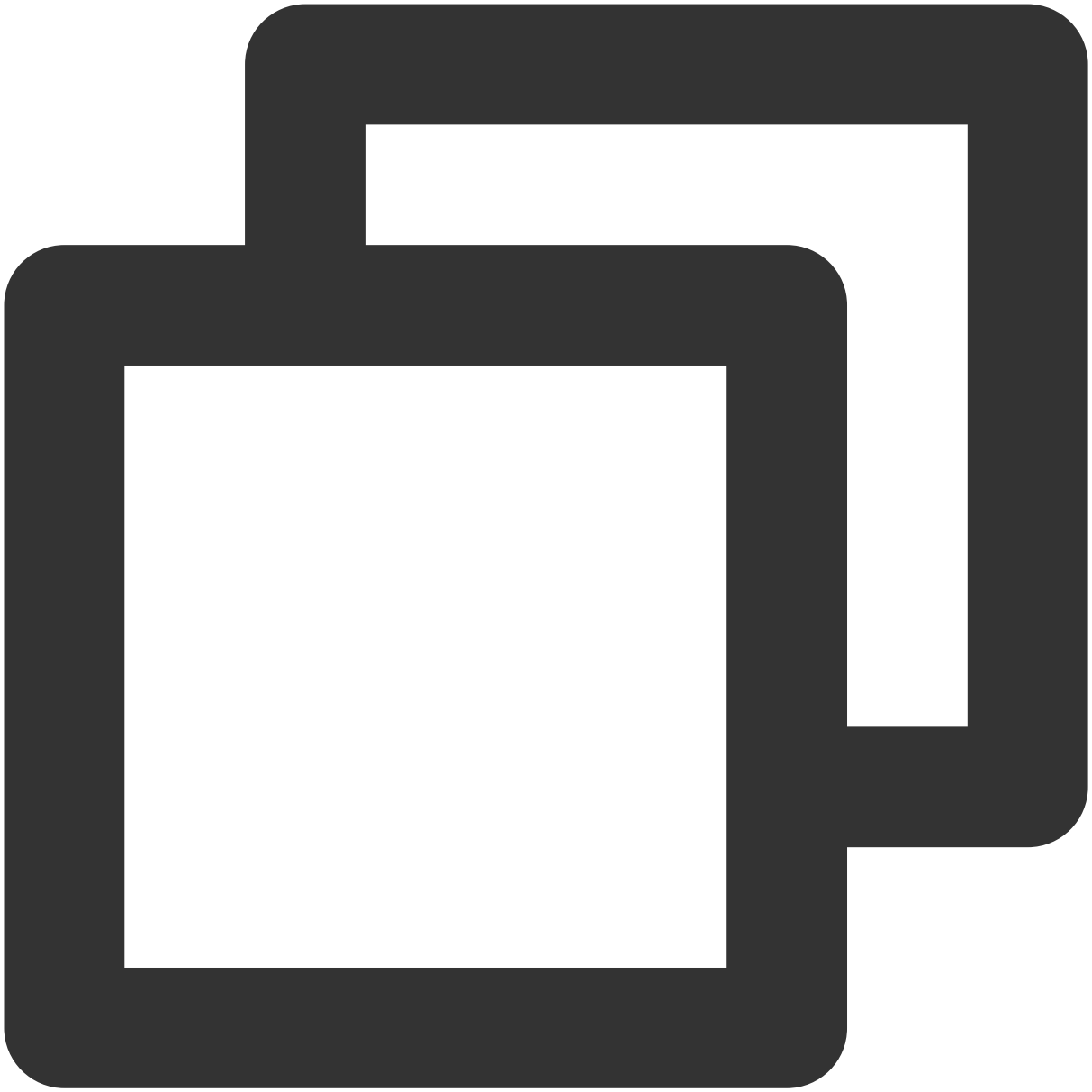
```
JSON_ARRAY_LENGTH(<jsonArray> string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the number of elements in the outermost JSON array.

Return type: integer

Example:



```
> SELECT json_array_length('[1,2,3,4]');
4
> SELECT json_array_length('[1,2,3,{"f1":1,"f2":[5,6]},4]');
5
> SELECT json_array_length('[1,2');
NULL
```

JSON_OBJECT_KEYS

Function statement:

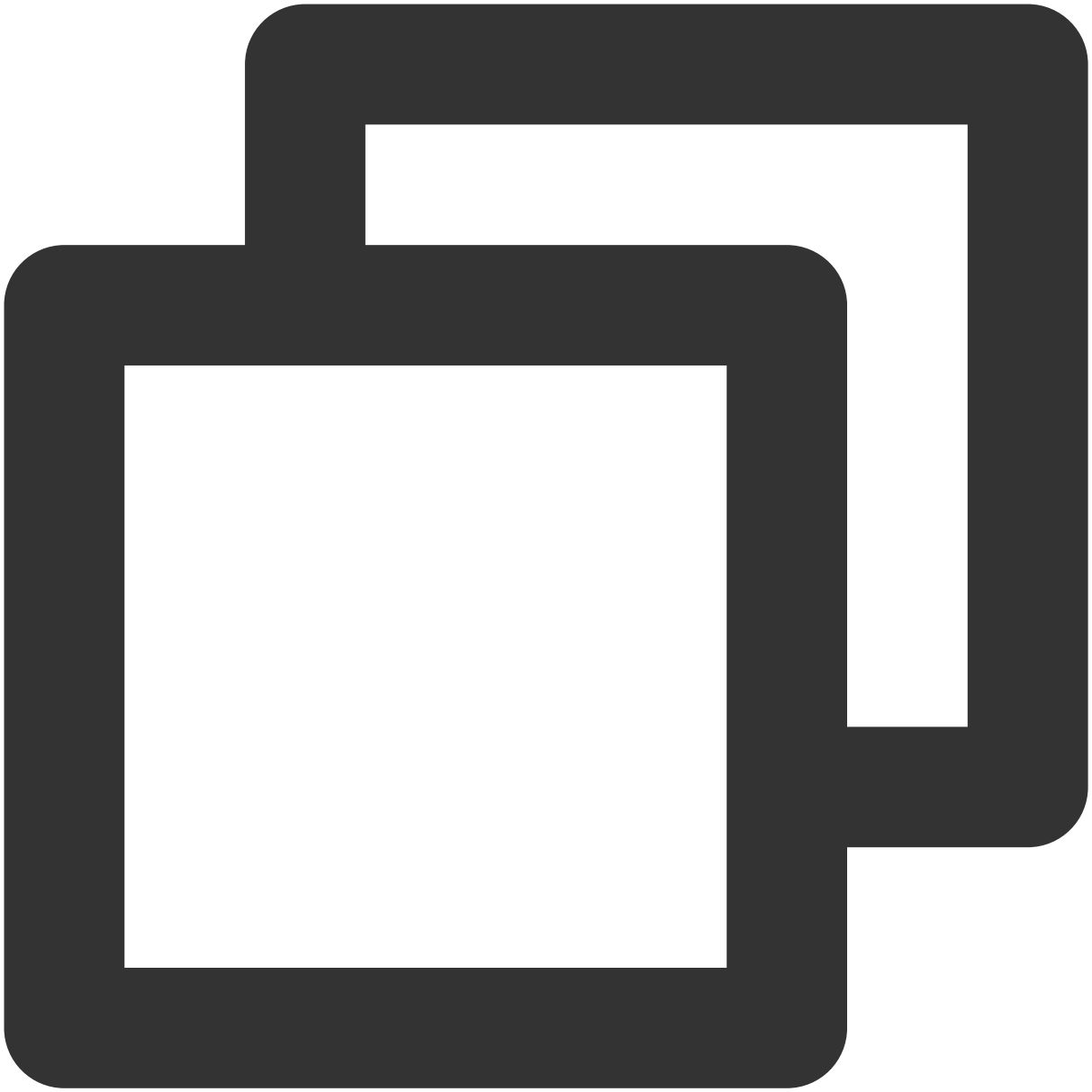
```
JSON_OBJECT_KEYS(<json> string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return all keys of the outermost JSON object as an array.

Return type: array<string>

Example:



```
> SELECT json_object_keys('{}');
[]
> SELECT json_object_keys('{"key": "value"}');
["key"]
> SELECT json_object_keys('{"f1":"abc","f2":{"f3":"a", "f4":"b"}}');
["f1","f2"]
```

Mathematical Functions

Last updated：2024-08-07 17:32:33

ABS

Function statement:

```
ABS(<expr> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the absolute value of expr.

Return type: Same as <expr>.

Example:

```
> SELECT abs(-1);
 1
```

ACOS

Function statement:

```
ACOS(<expr> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the arc cosine of expr.

Return type: double

Example:

```
> SELECT acos(1);
 0.0
> SELECT acos(2);
 NaN
```

ACOSH

Function statement:

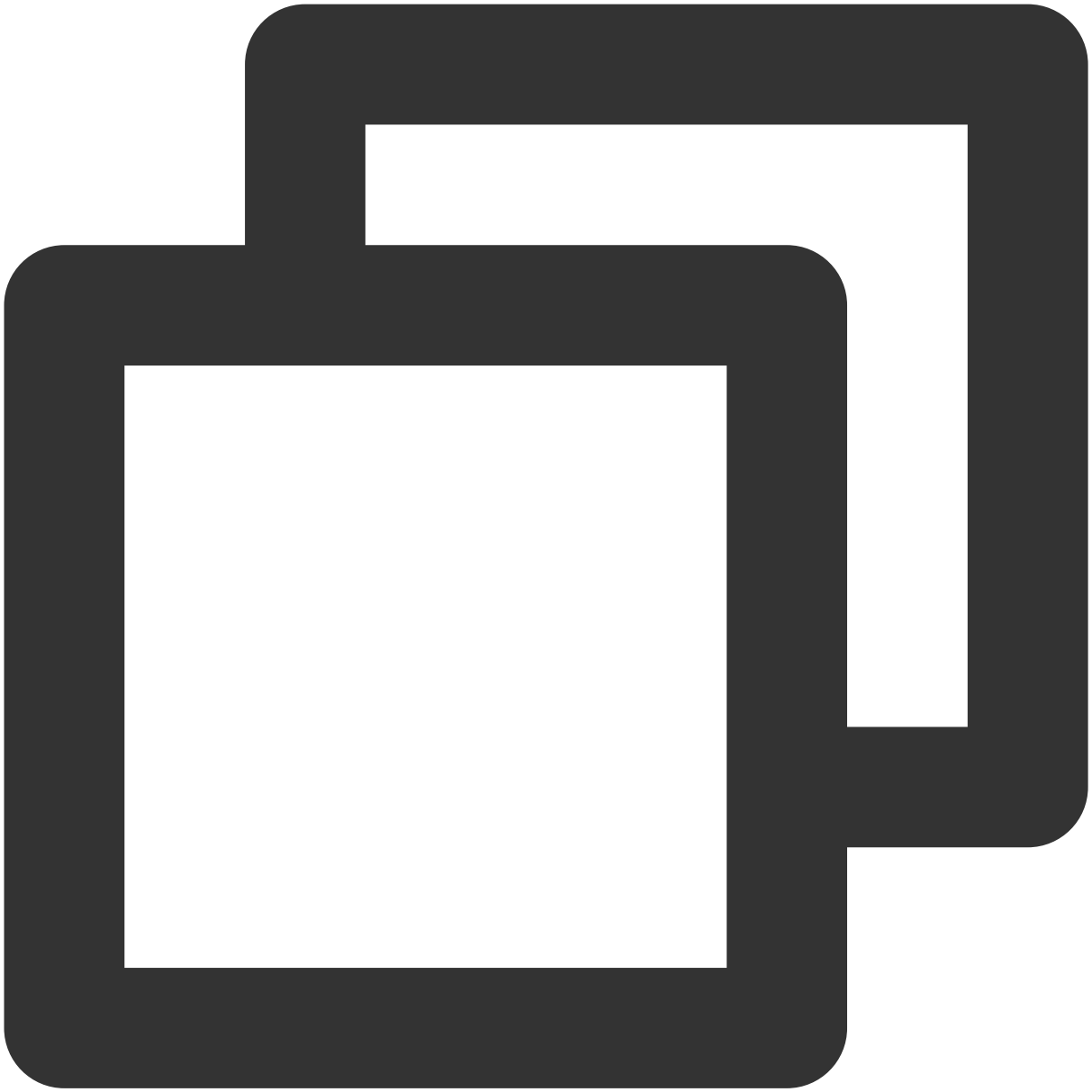
```
ACOSH(<expr> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the hyperbolic arc cosine of expr.

Return type: double

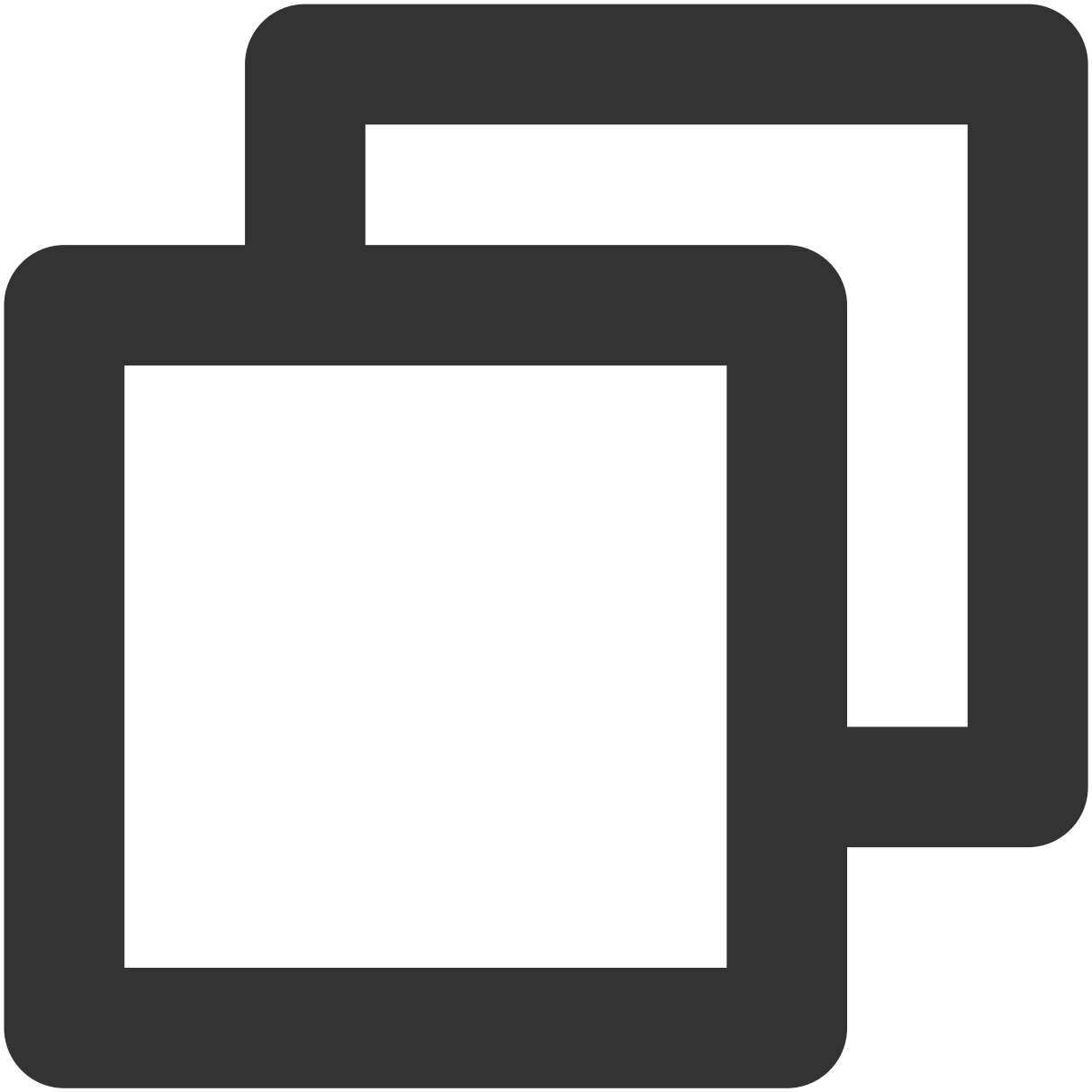
Example:



```
> SELECT acosh(1);
 0.0
> SELECT acosh(0);
 NaN
```

ASIN

Function statement:



```
ASIN(<expr> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the arc sine of expr.

Return type: double

Example:

```
> SELECT asin(0);
 0.0
> SELECT asin(2);
 NaN
```

ASINH

Function statement:

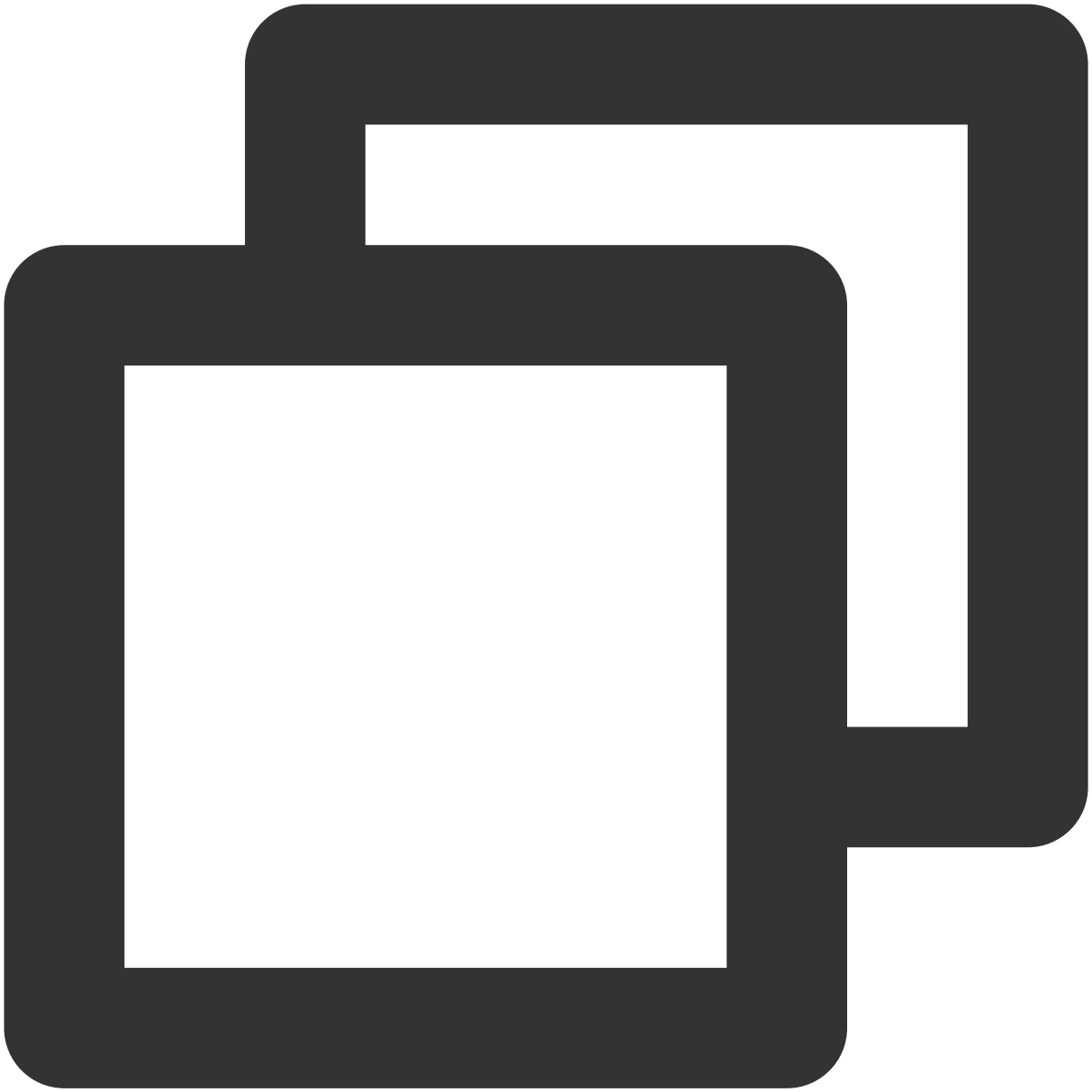
```
ASINH(<expr> integer|double|decimal)
```

Supported engine: SparkSQL

Usage instructions: Return the hyperbolic arc sine of expr.

Return type: double

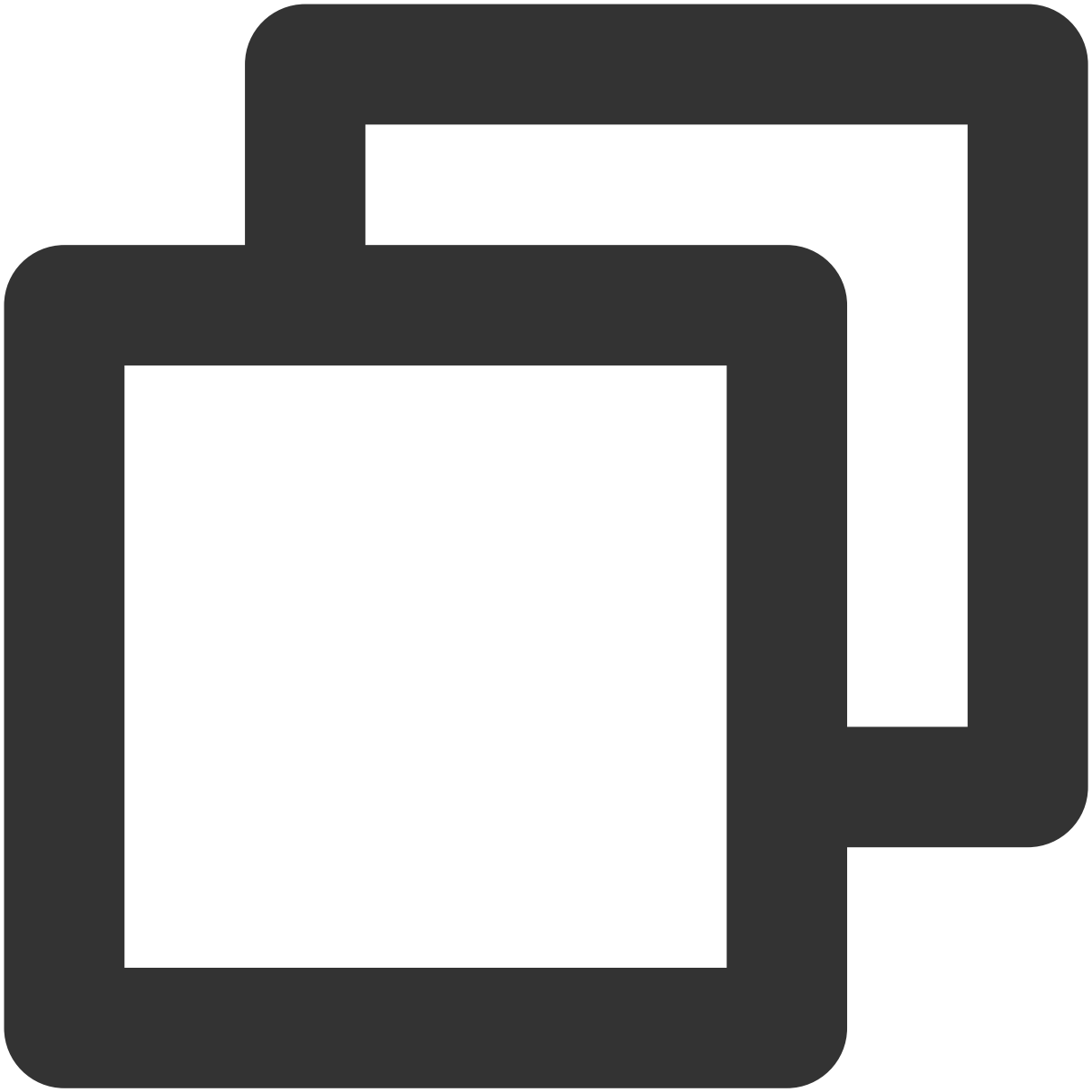
Example:



```
> SELECT asinh(0);
 0.0
```

ATAN

Function statement:



```
ATAN(<expr> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the arc tangent of expr.

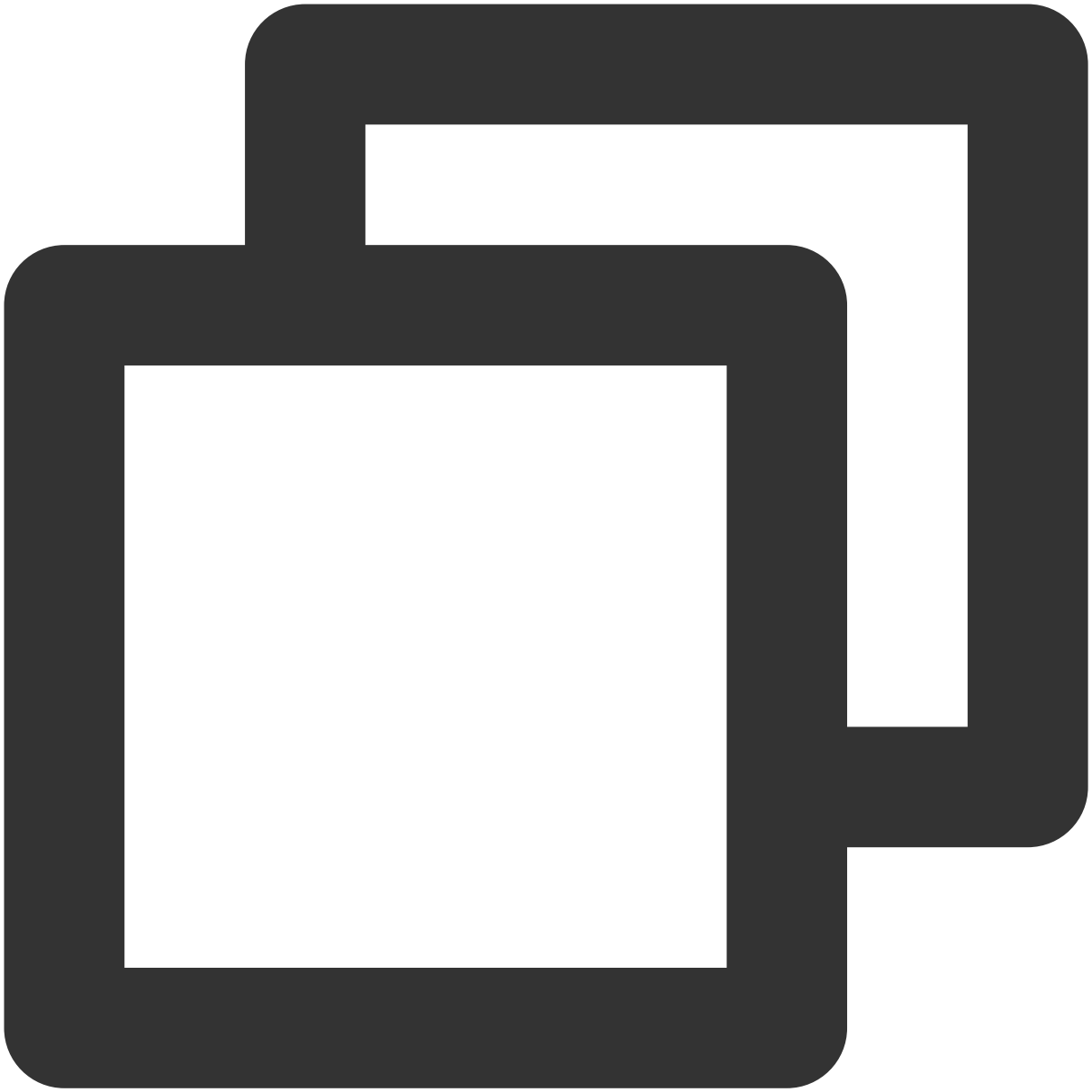
Return type: double

Example:

```
> SELECT atan(0);
 0.0
```

ATAN2

Function statement:



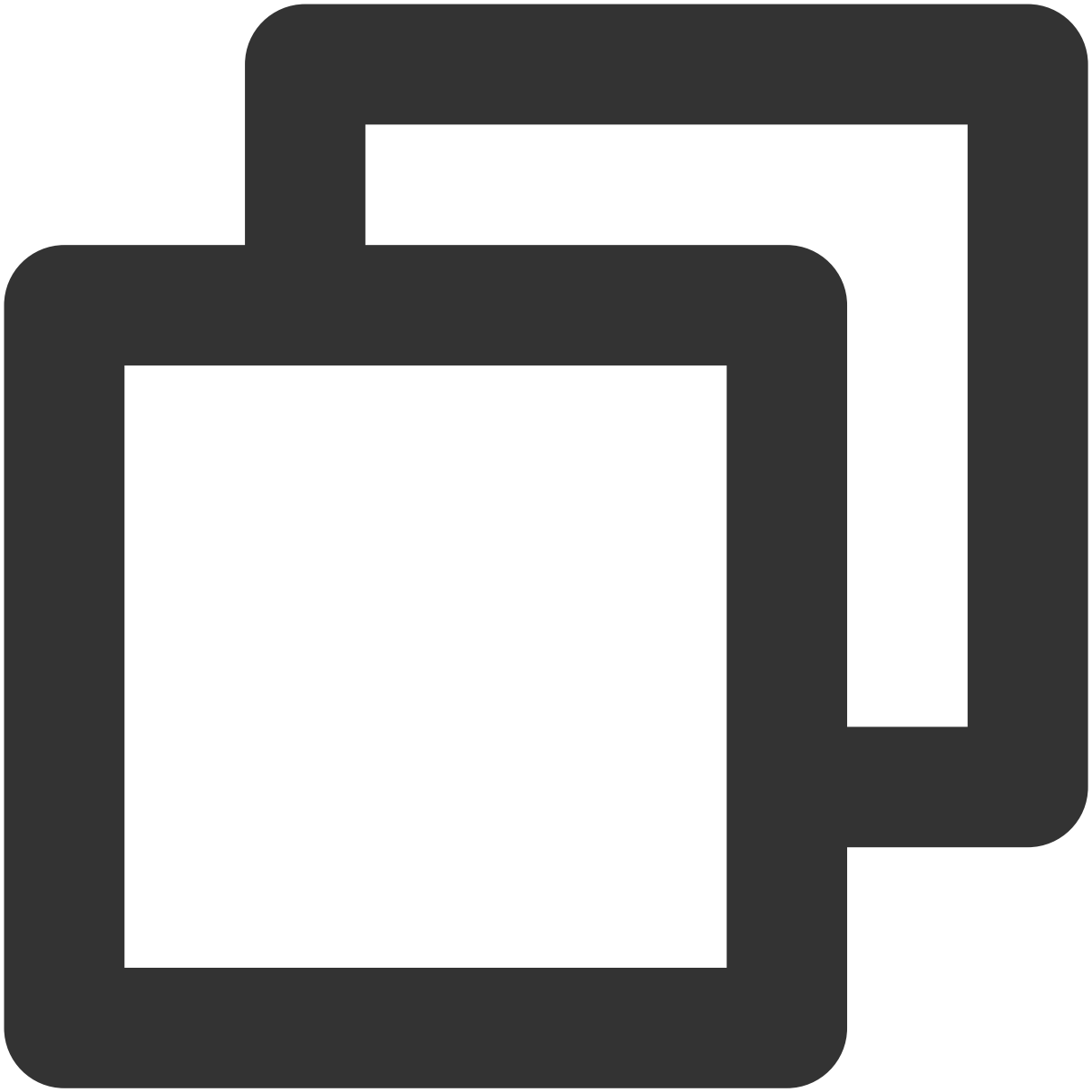
```
ATAN2(<x>, integer|double|decimal, <y> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the angle in radians between the positive x-axis and the point given by the coordinates.

Return type: double

Example:



```
> SELECT atan2(0, 0);
 0.0
```

ATANH

Function statement:

```
ATANH(<expr> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the hyperbolic arc tangent of expr.

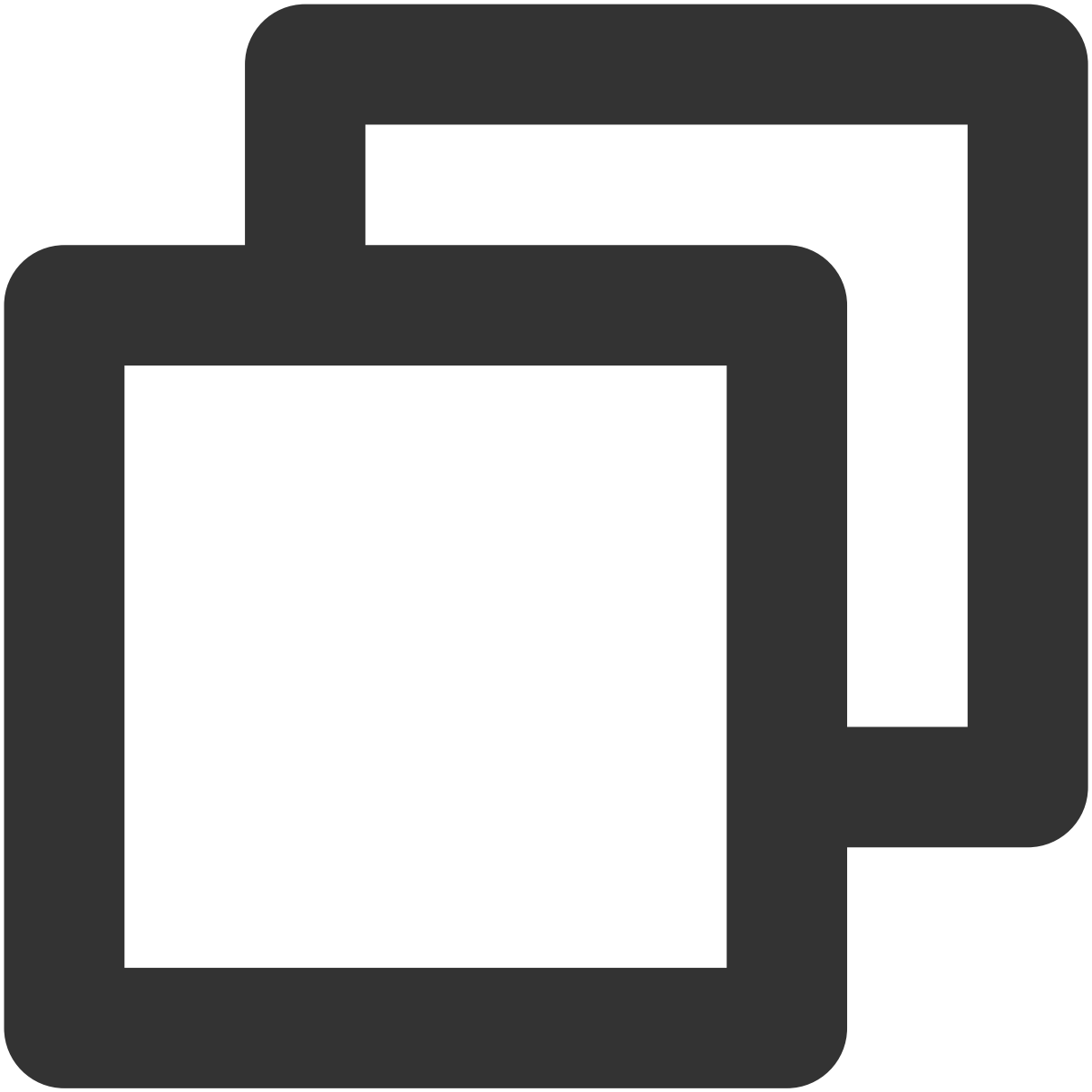
Return type: double

Example:

```
> SELECT atanh(0);
 0.0
> SELECT atanh(2);
 NaN
```

BIN

Function statement:



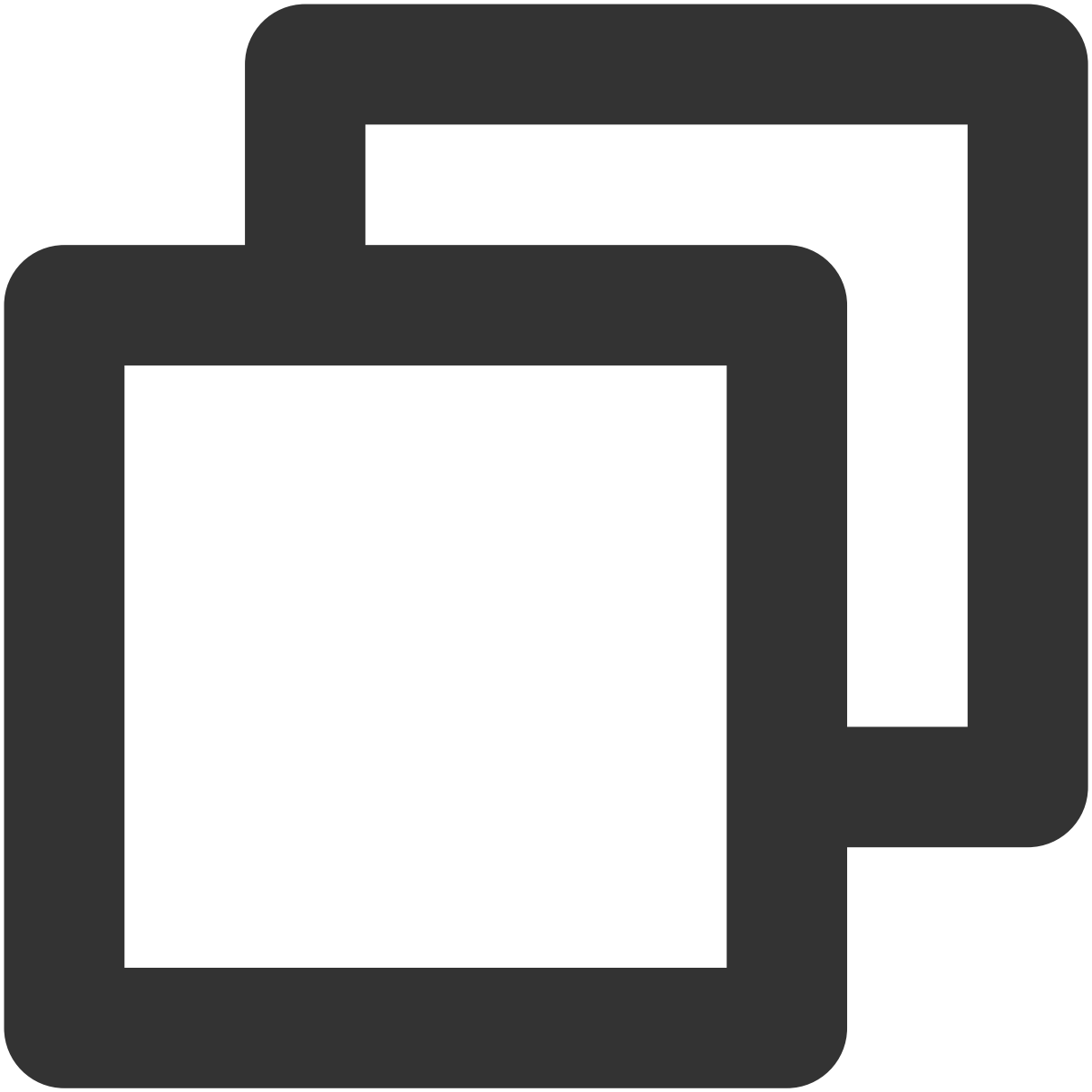
```
BIN(<expr> integer)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the string representation of the long value expr in binary format.

Return type: string

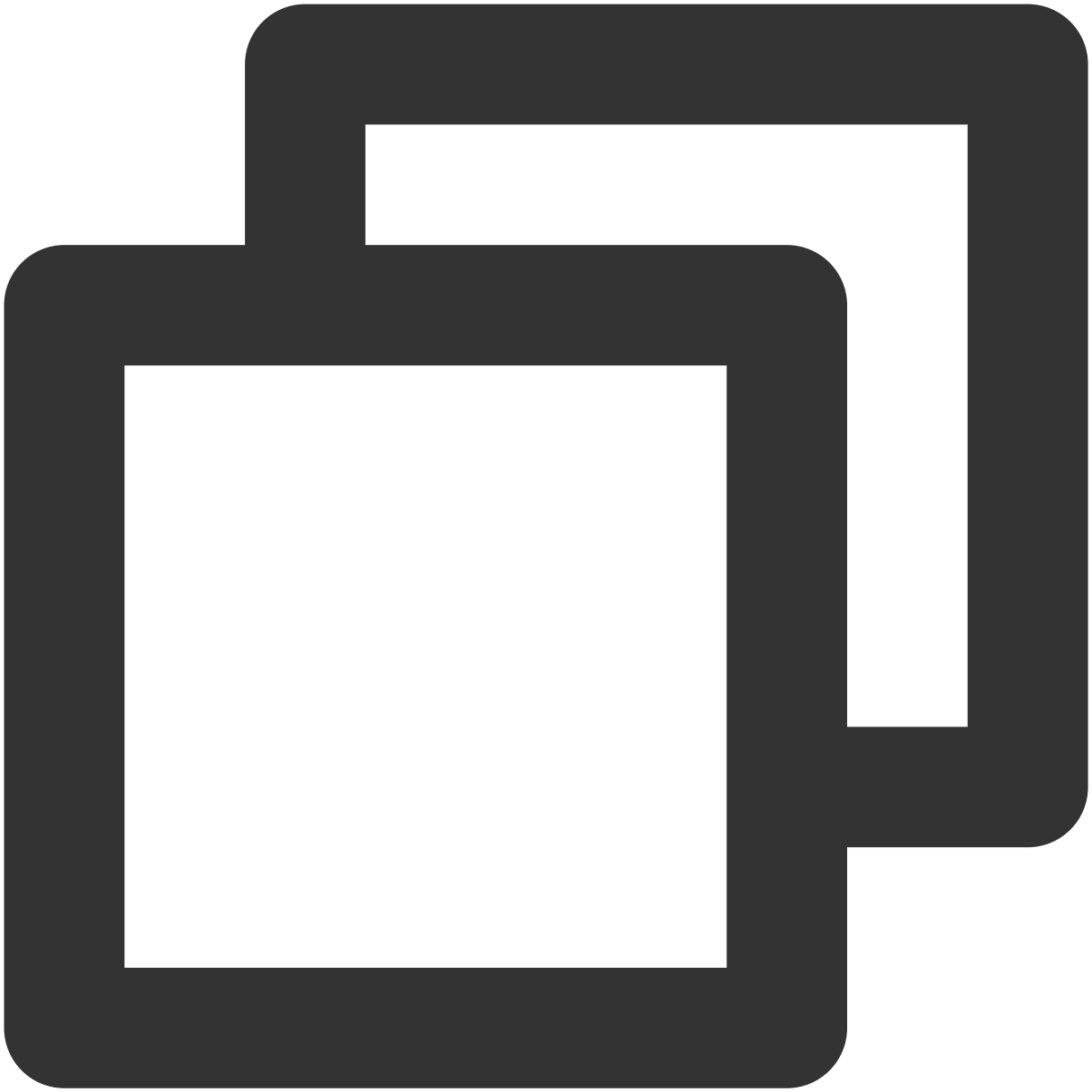
Example:



```
> SELECT bin(13);
 1101
> SELECT bin(-13);
 1111111111111111111111111111111111111111111111111111111111110011
```

BROUND

Function statement:



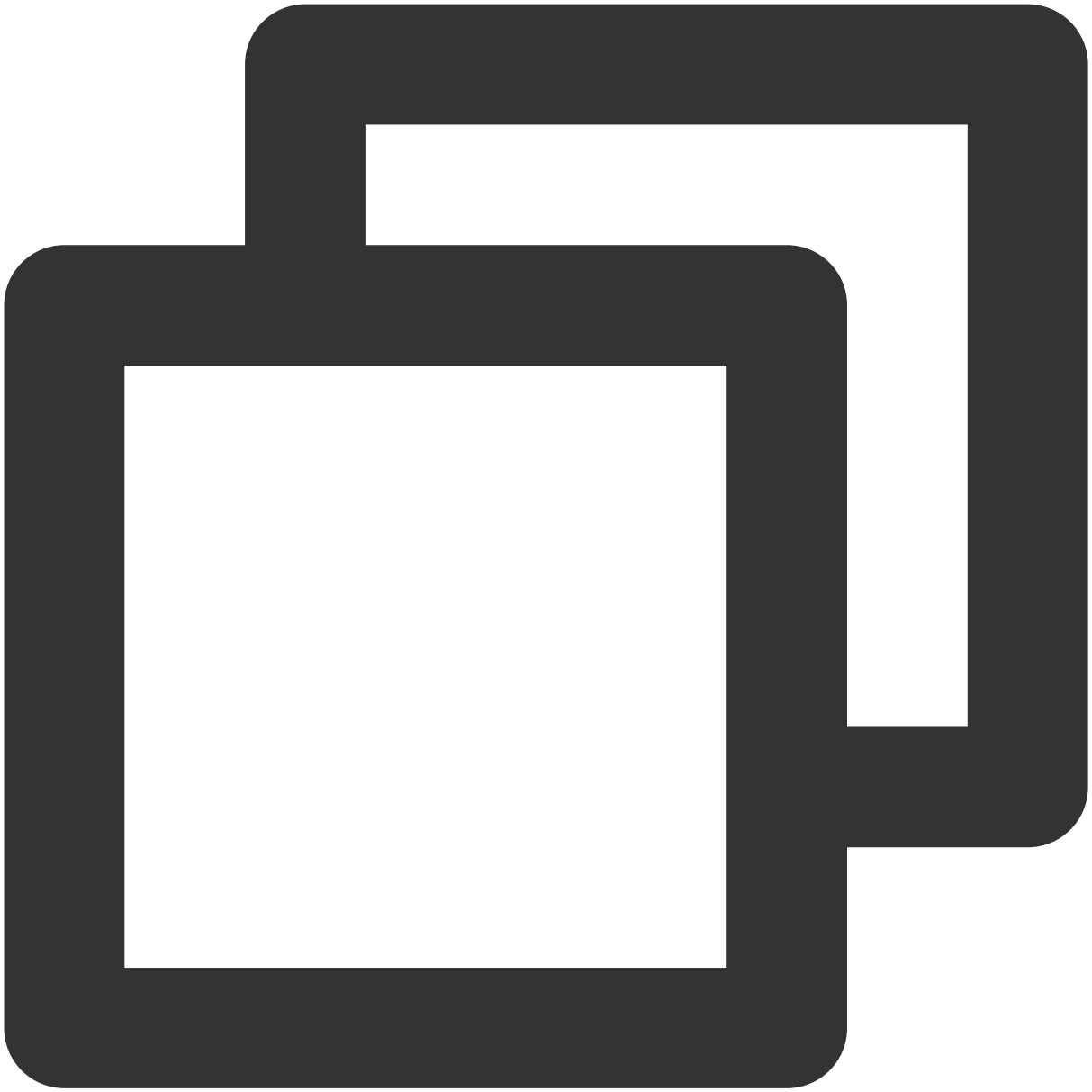
```
BROUND(<expr> integer|double|decimal, <d> integer)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return expr rounded to d decimal places using the HALF_EVEN rounding mode.

Return type: decimal

Example:



```
> SELECT bround(2.5, 0);
 2
```

CBRT

Function statement:

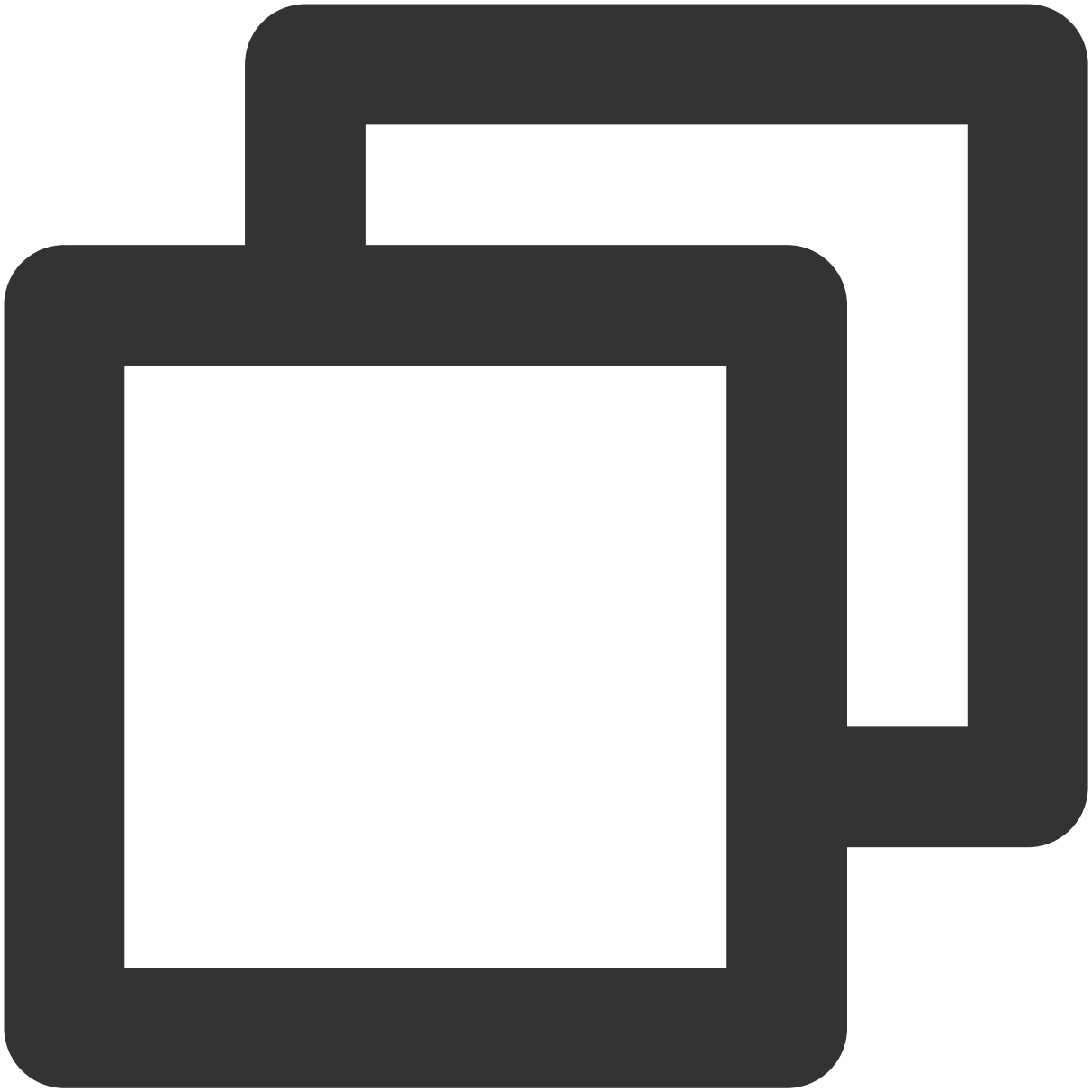
```
CBRT(<expr> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the cube root of expr.

Return type: double

Example:



```
> SELECT cbrt(27.0);
 3.0
```

CEIL

Function statement:

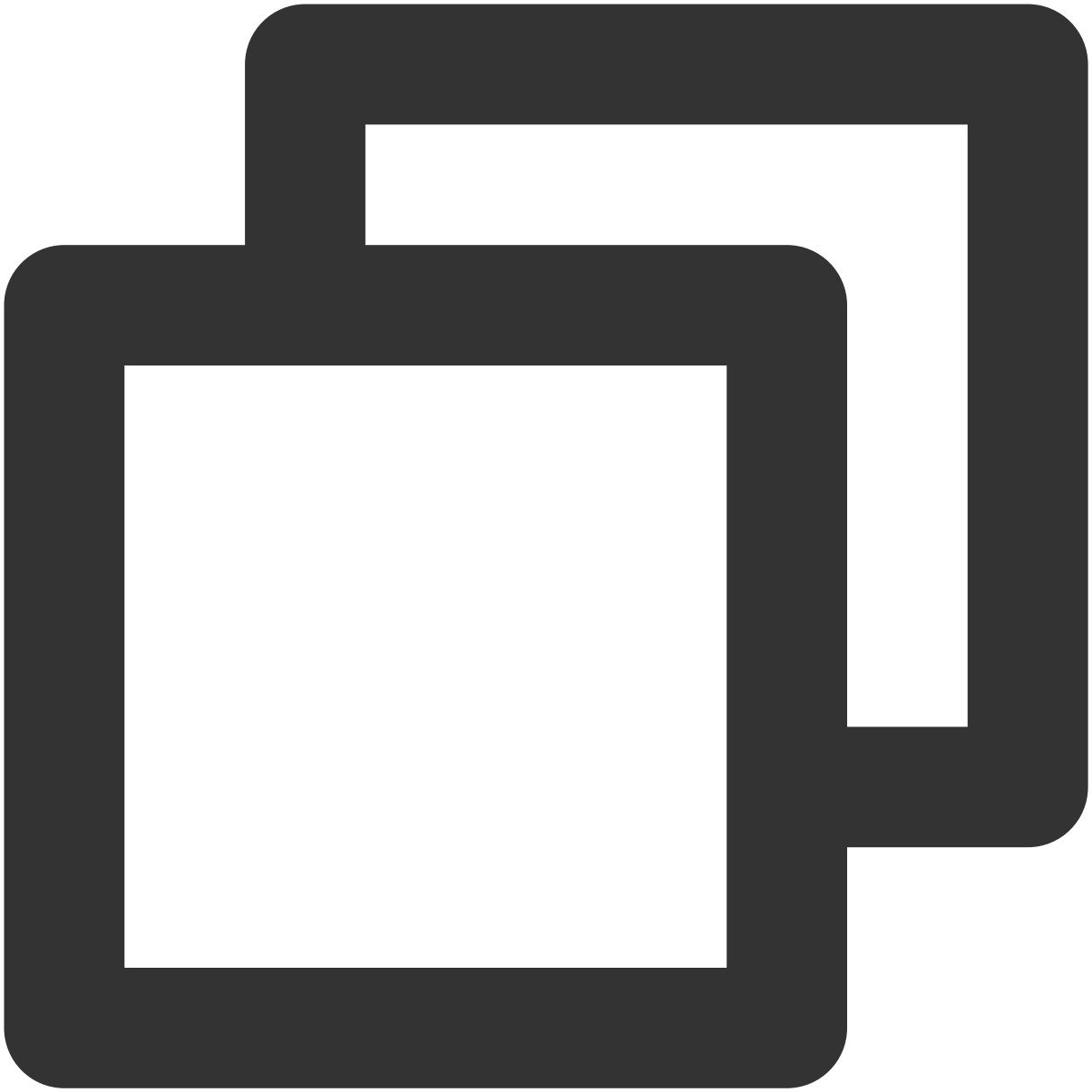
```
CEIL(<expr> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the smallest integer not less than expr.

Return type: integer

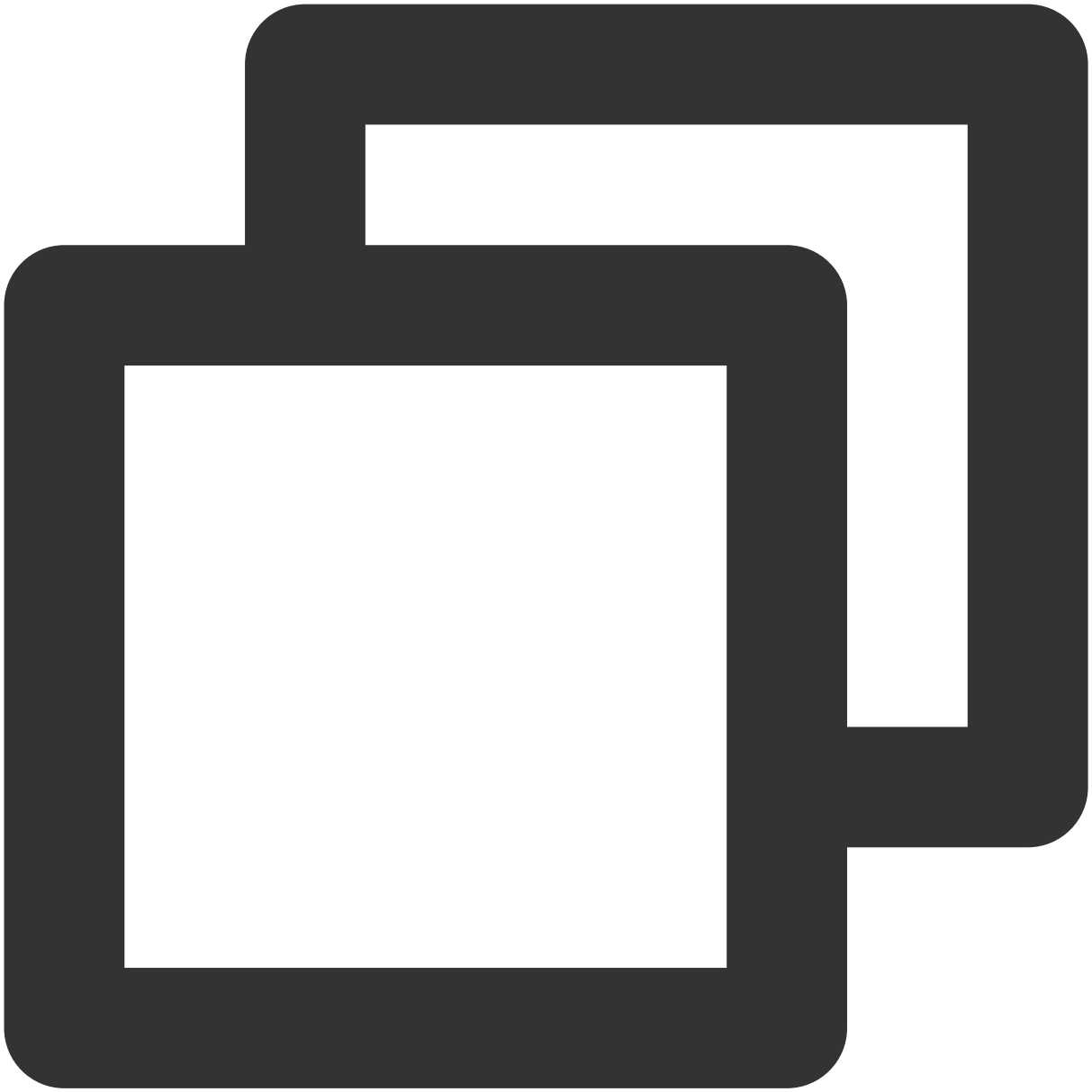
Example:



```
> SELECT ceil(-0.1);
 0
> SELECT ceil(5);
 5
```

COS

Function statement:



```
COS(<expr> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the cosine of expr.

Return type: double

Example:

```
> SELECT cos(0);
 1.0
```

COSH

Function statement:

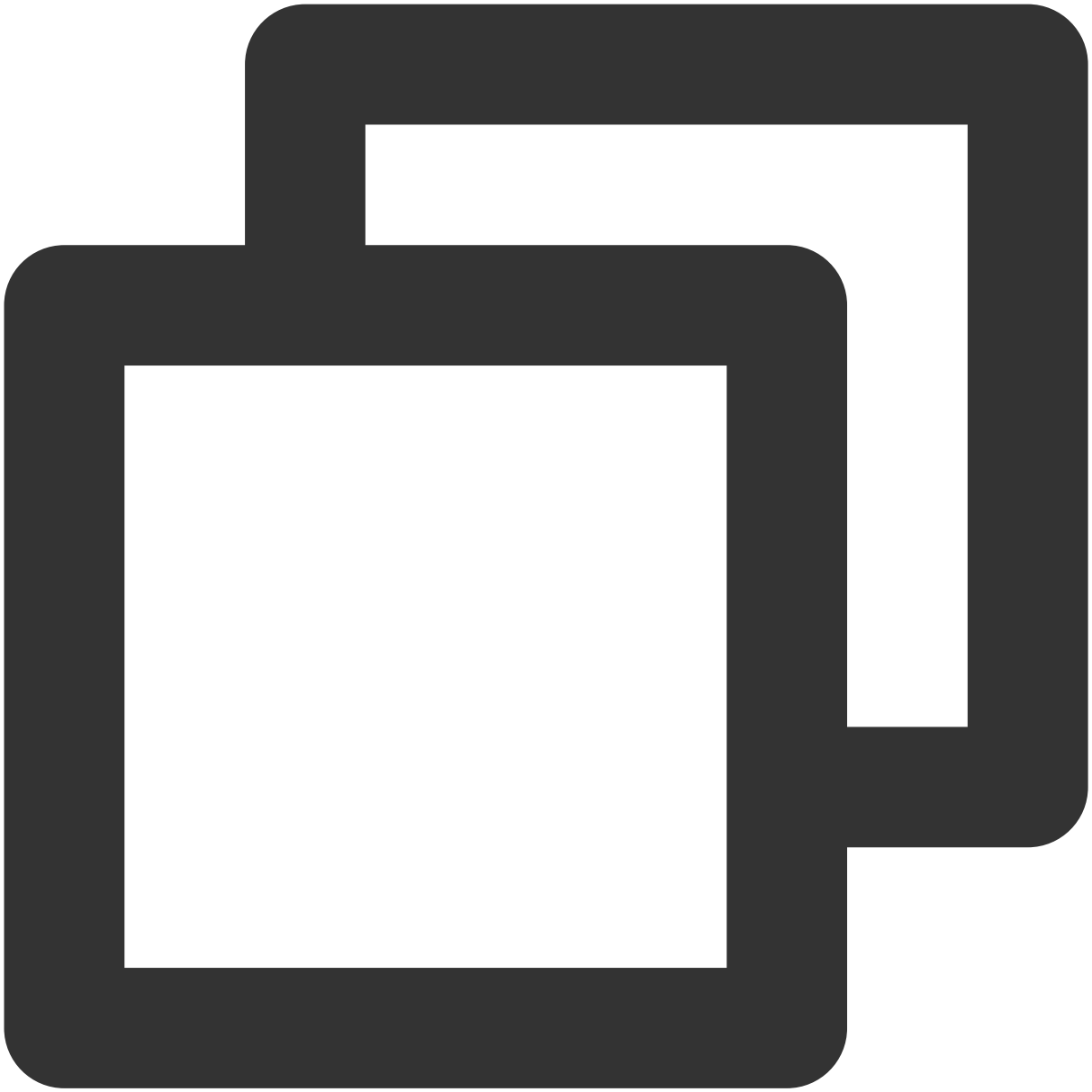
```
COSH(<expr> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the hyperbolic cosine of expr.

Return type: double

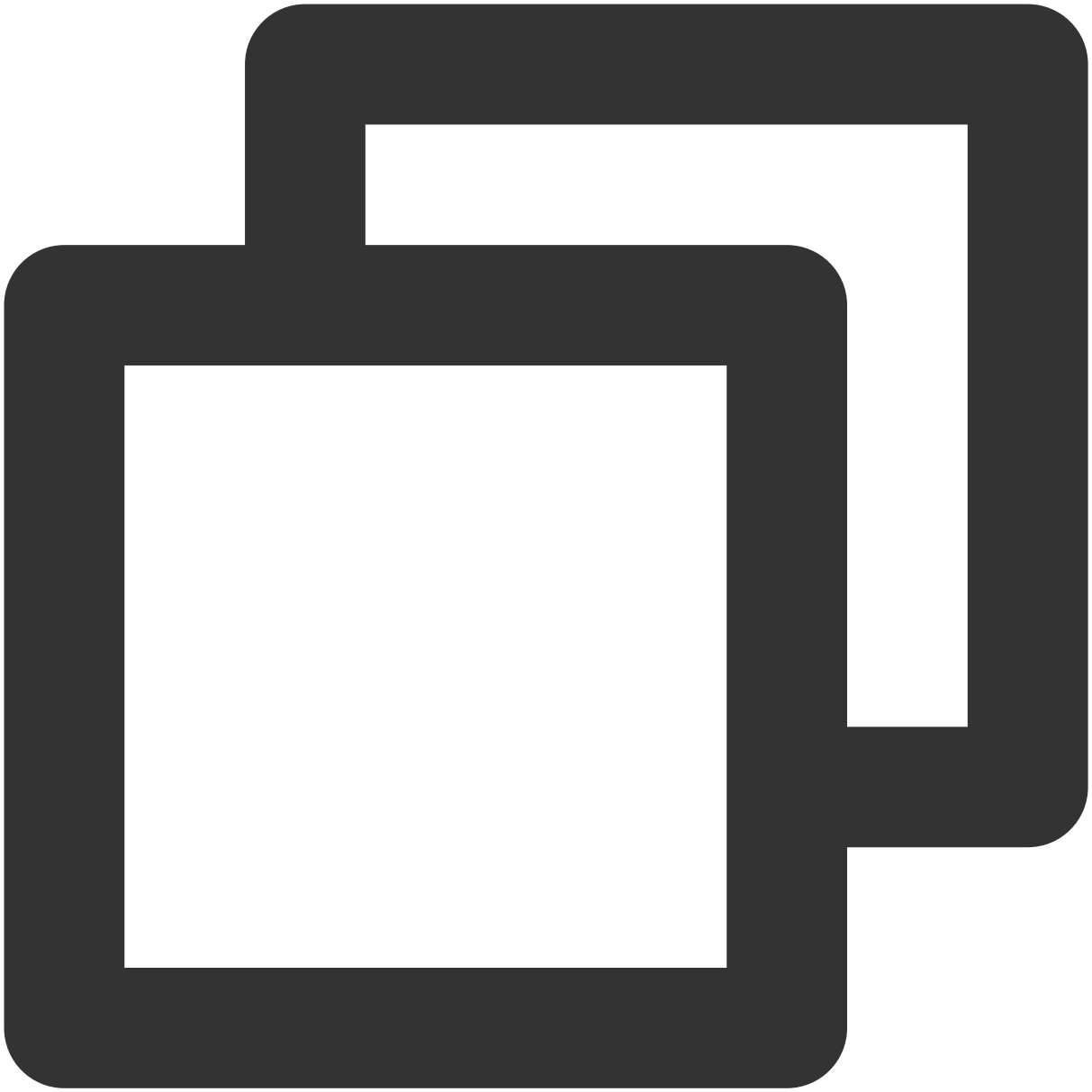
Example:



```
> SELECT cosh(0);
 1.0
```

CONV

Function statement:



```
CONV(<num> bigint|string, <from_base> integer, <to_base> integer)
```

Supported engines: SparkSQL and Presto

Usage instructions: Convert num from from_base to to_base.

Return type: string

Example:

```
> SELECT conv('100', 2, 10);
 4
> SELECT conv(-10, 16, -10);
 -16
```

DEGREES

Function statement:

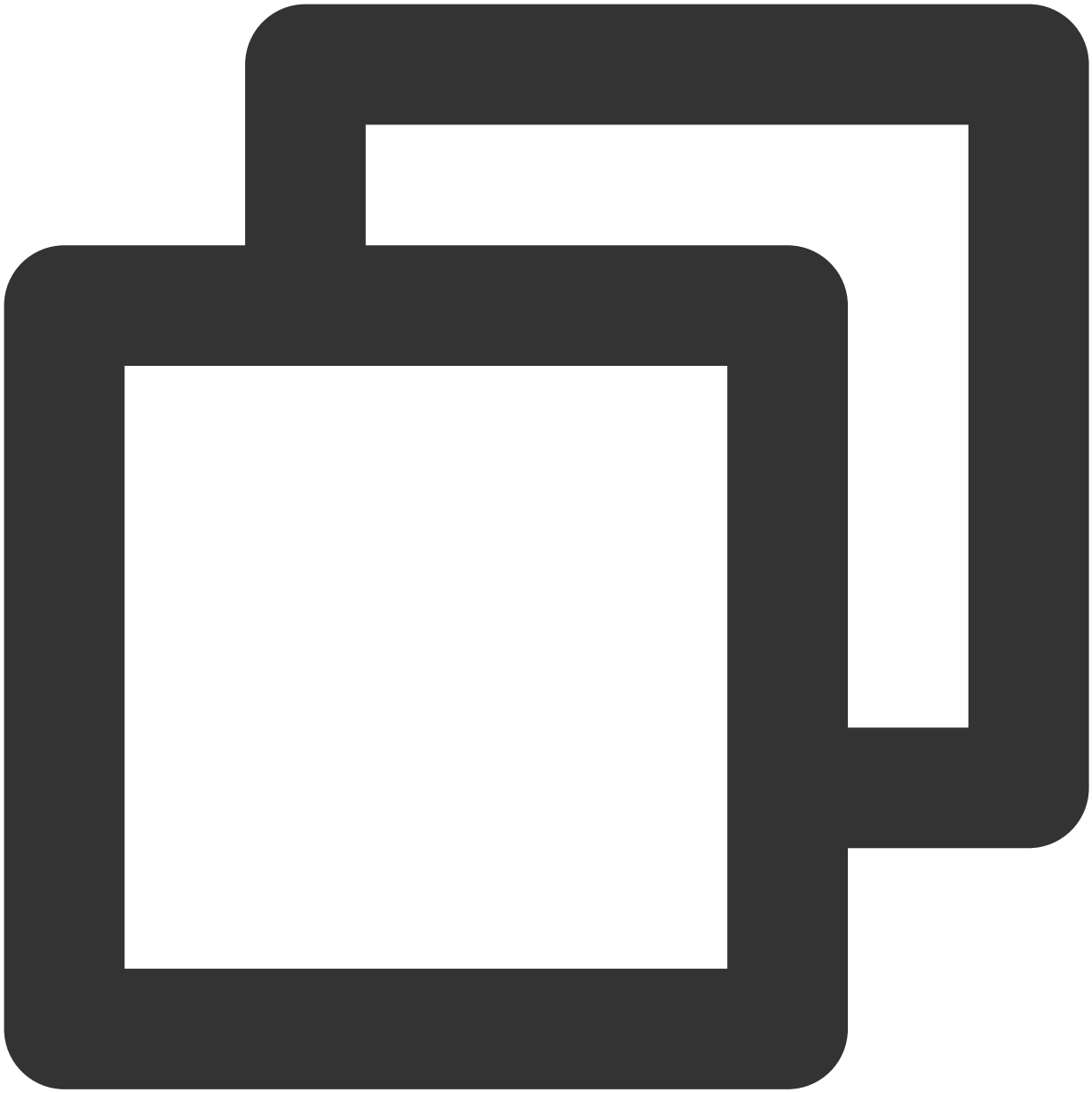
```
DEGREES(<expr> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Convert radians to degrees.

Return type: double

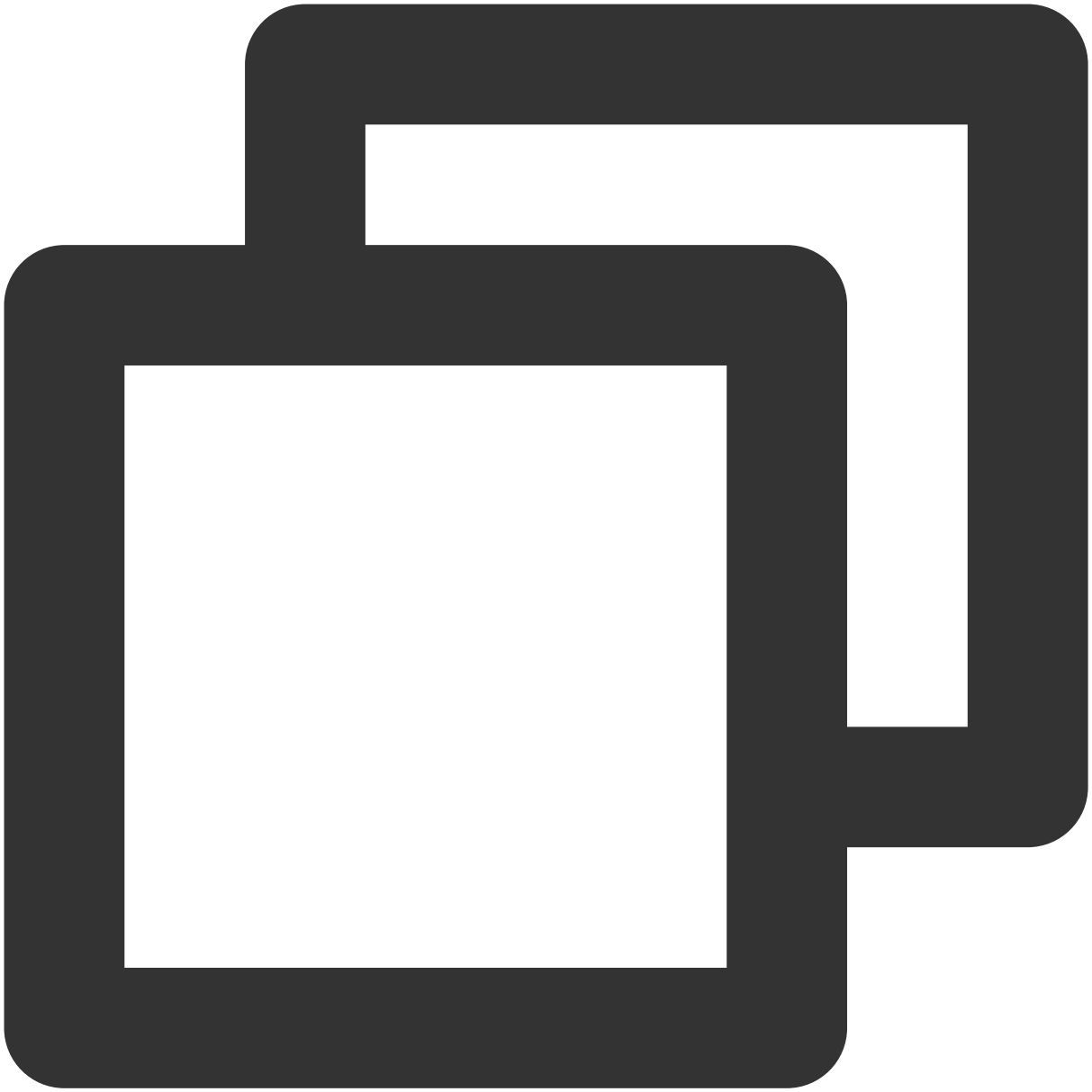
Example:



```
> SELECT degrees(3.141592653589793);
 180.0
```

E

Function statement:



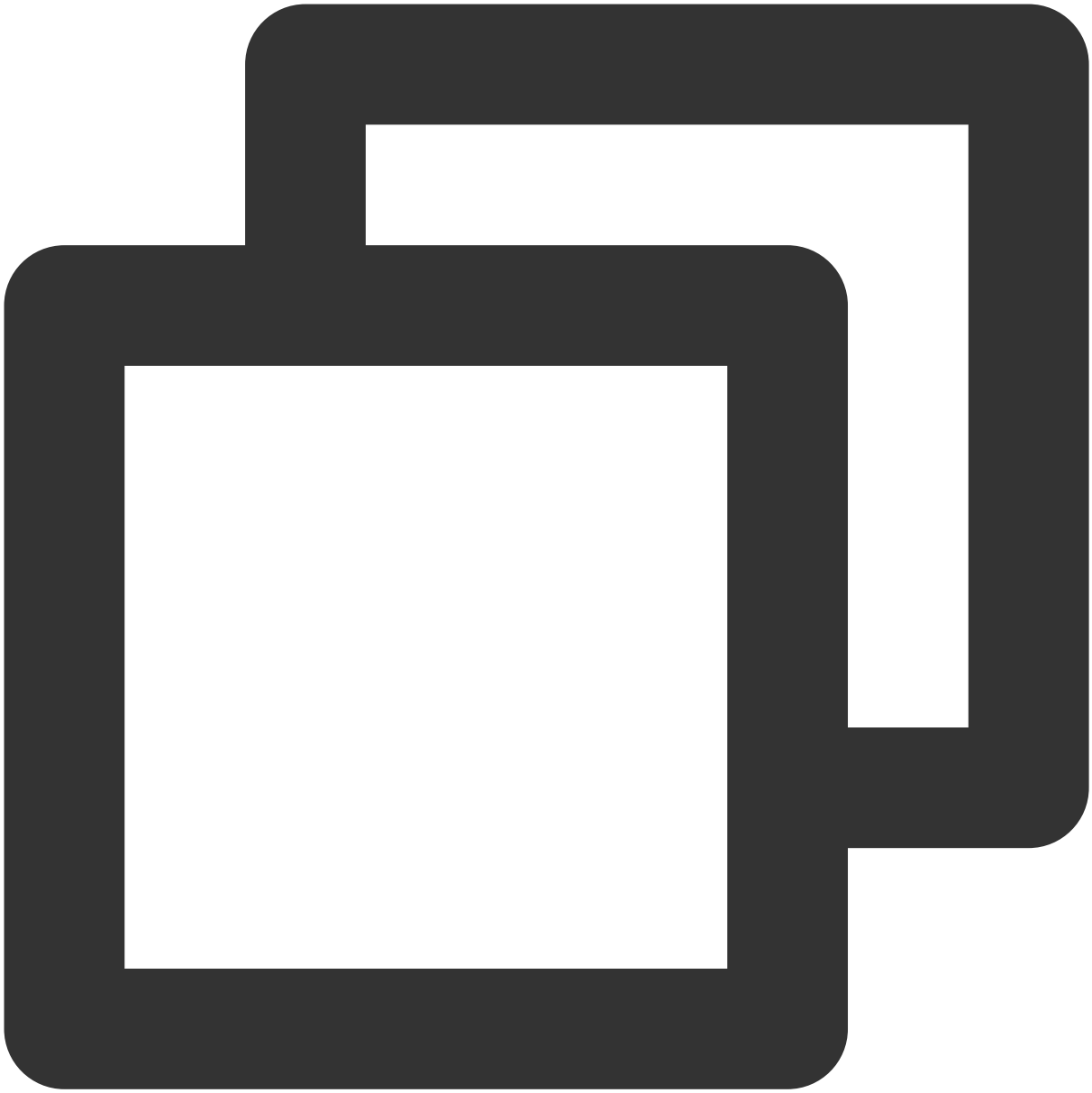
```
E()
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the Euler's constant.

Return type: double

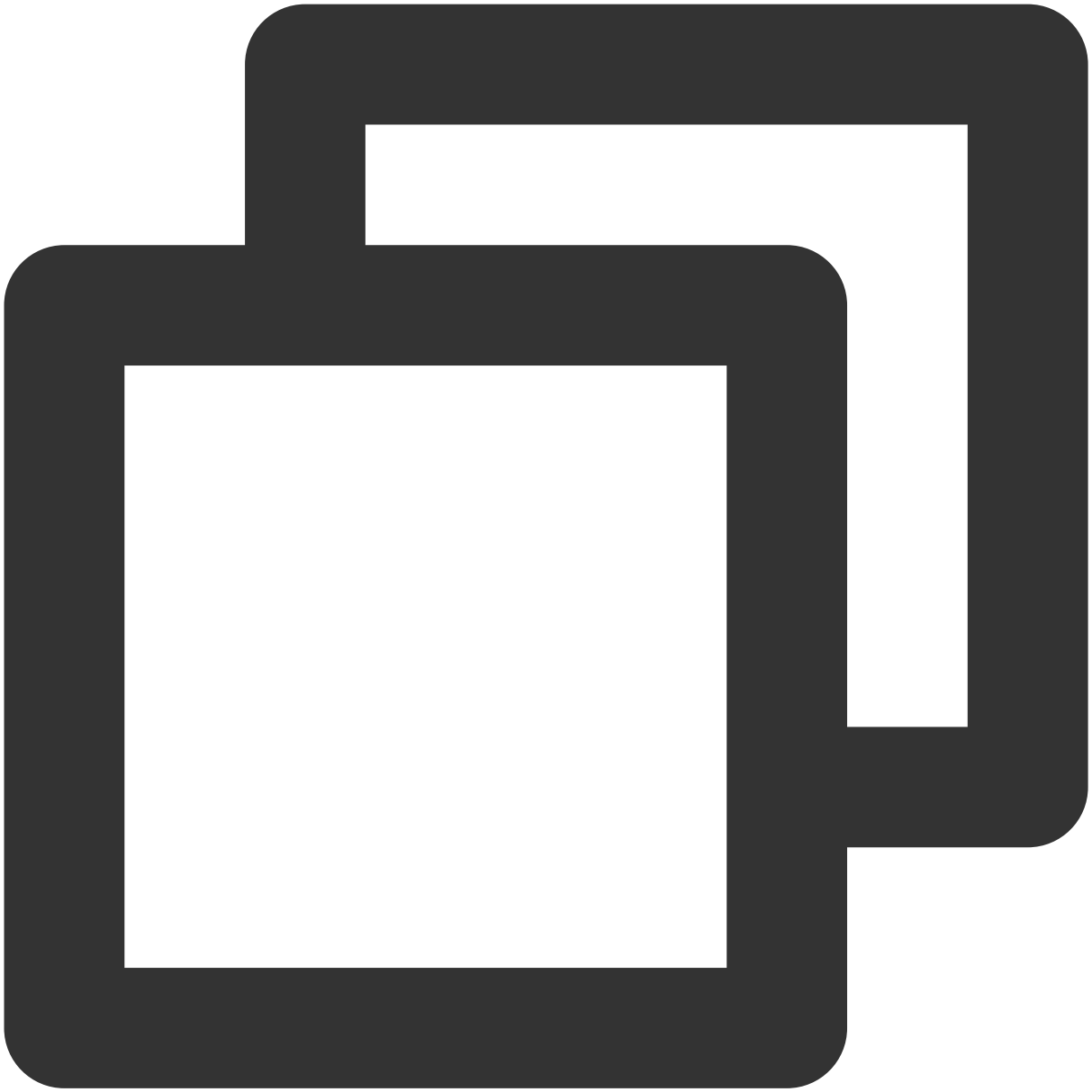
Example:



```
> SELECT e();
 2.718281828459045
```

EXP

Function statement:



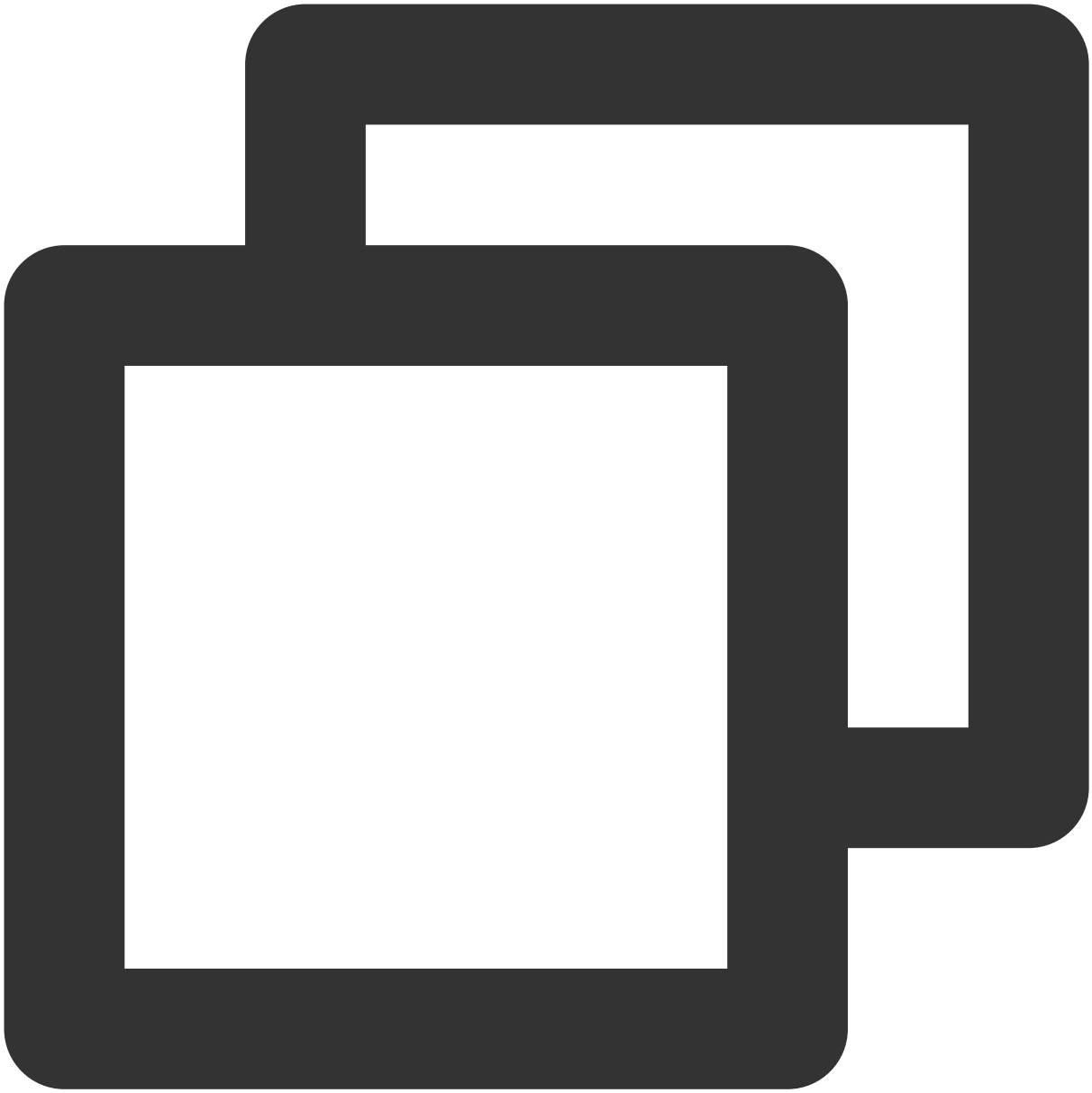
```
EXP(<expr> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return Euler's constant e raised to the power of expr.

Return type: double

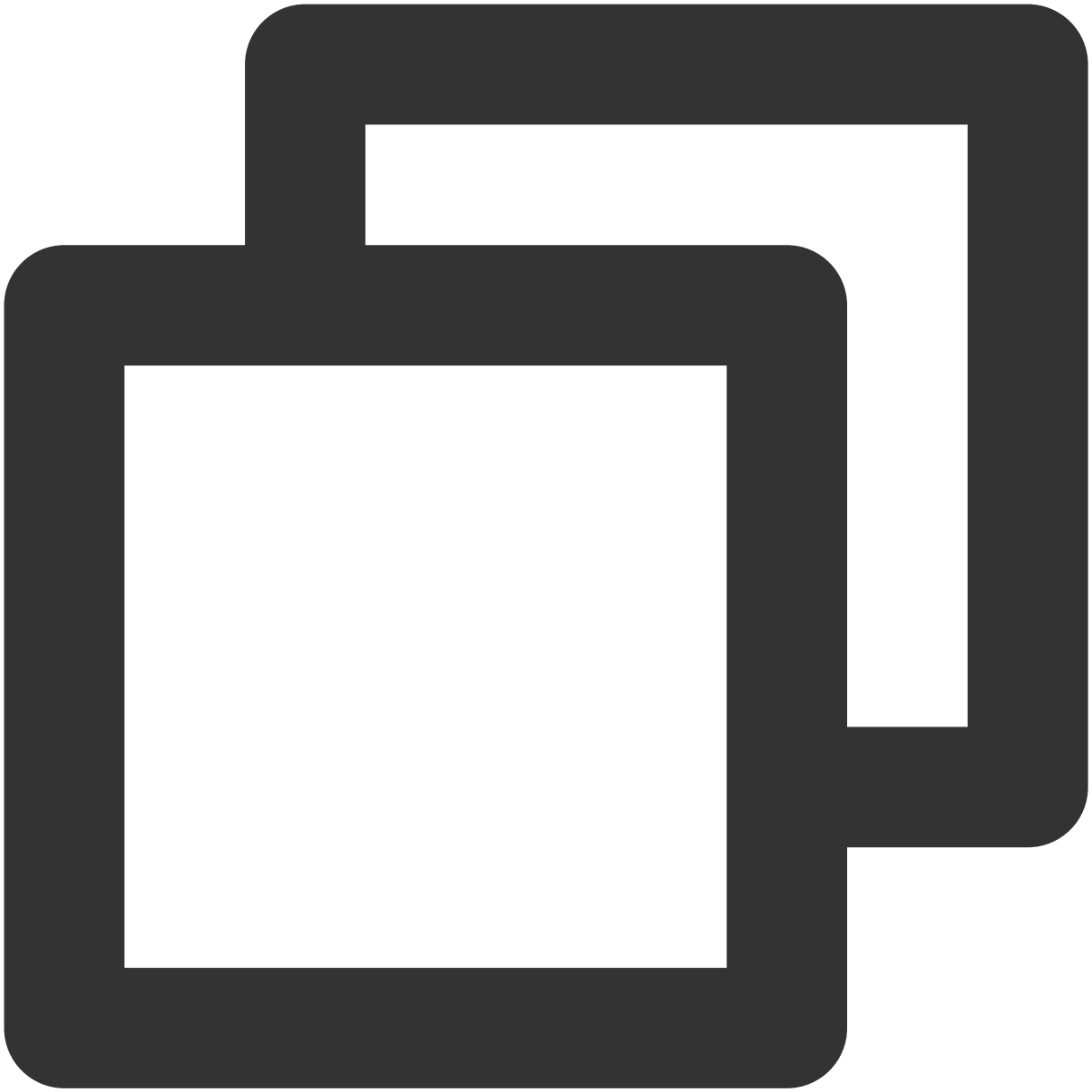
Example:



```
> SELECT exp(0);
 1.0
```

EXPM1

Function statement:



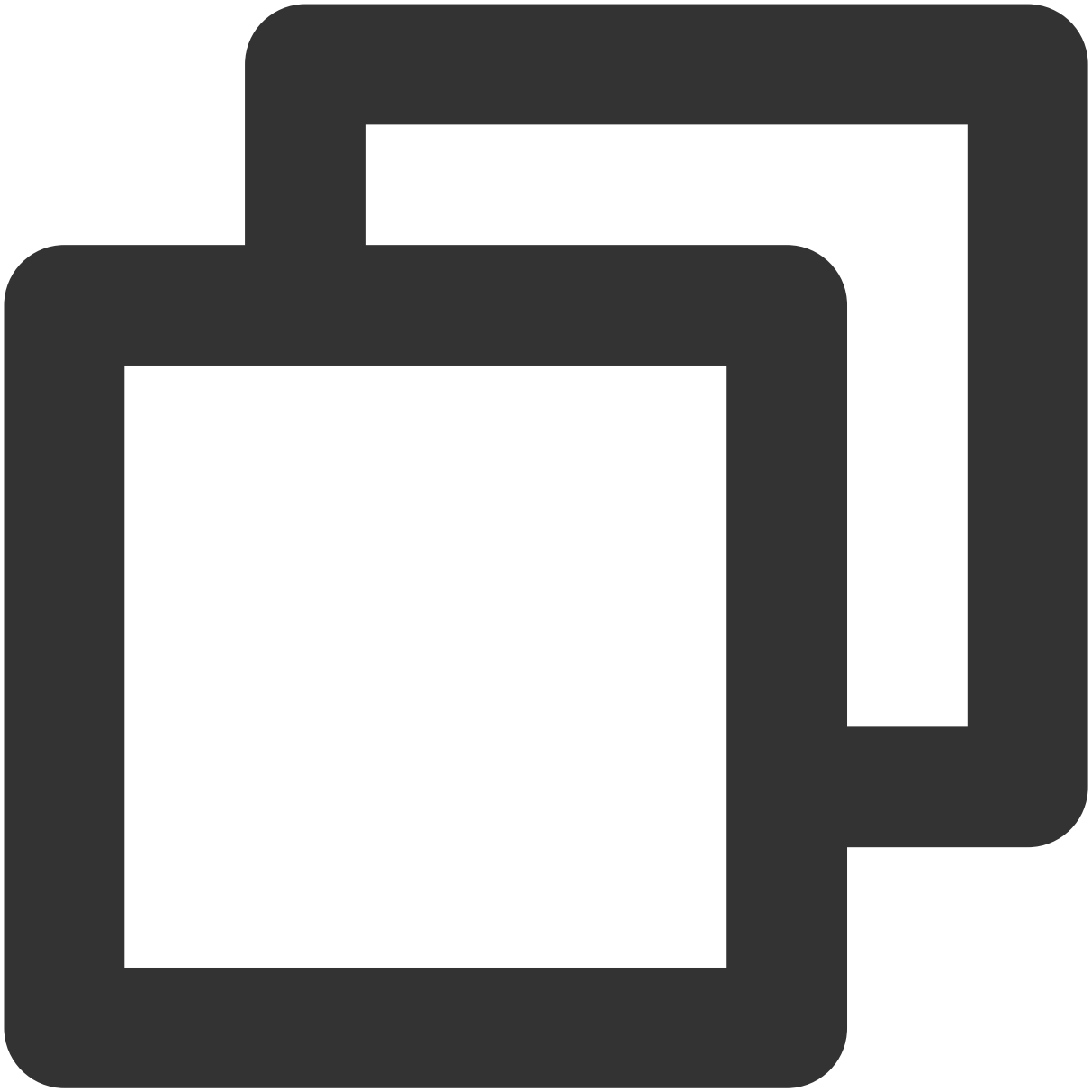
```
EXPM1(<expr> integer|double|decimal)
```

Supported engine: SparkSQL

Usage instructions: Return EXP(expr)-1.

Return type: double

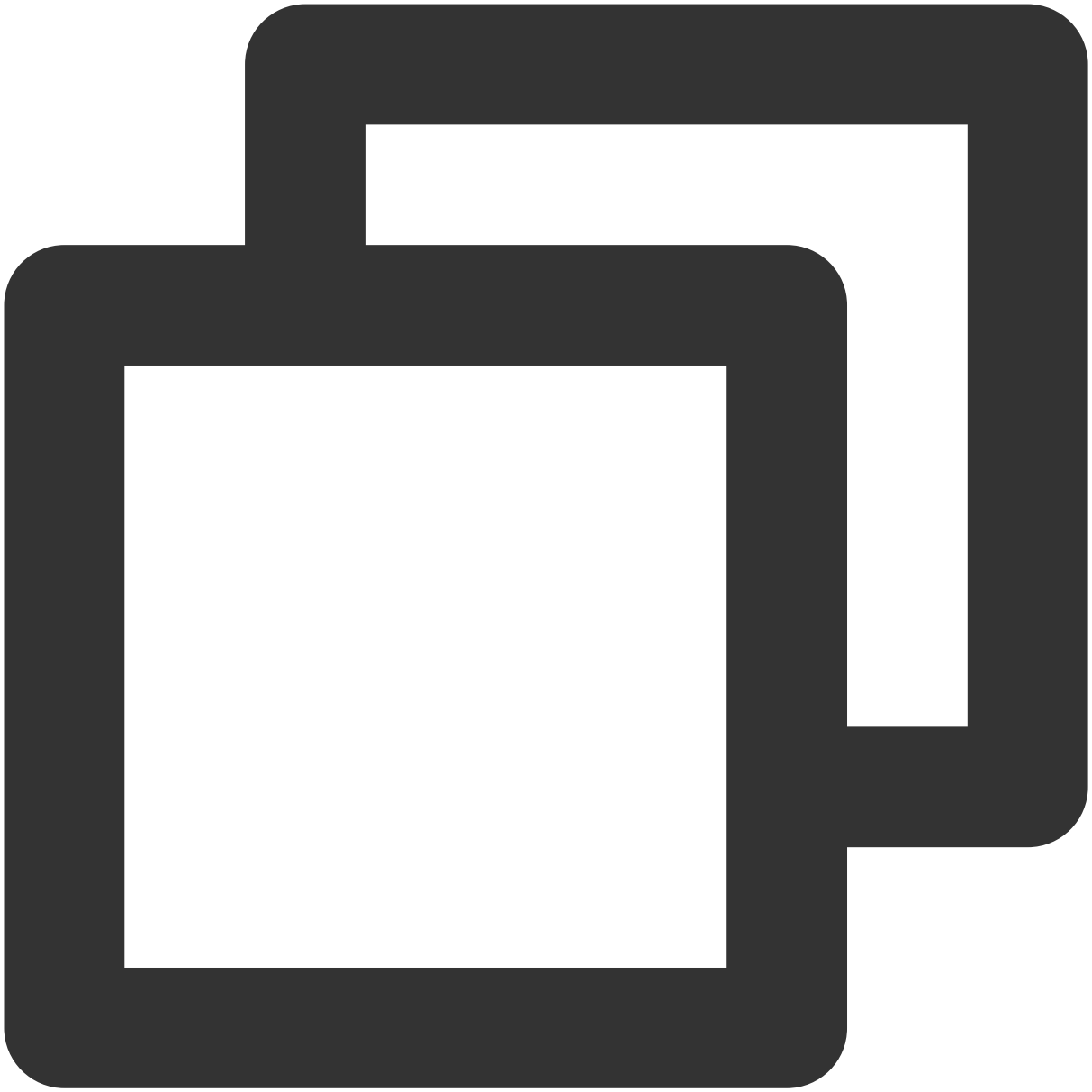
Example:



```
> SELECT expm1(0);
 0.0
```

FLOOR

Function statement:



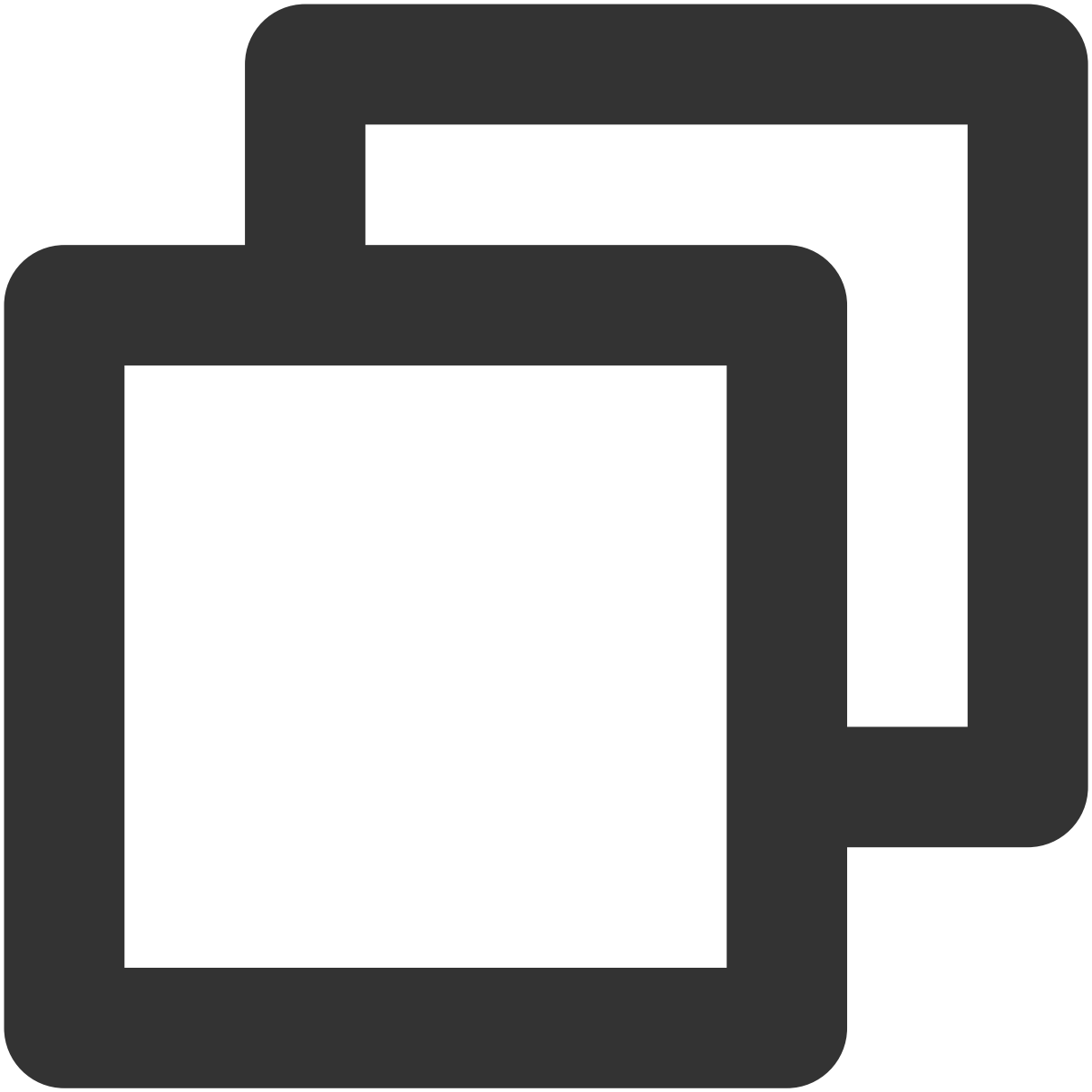
```
FLOOR(<expr> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the largest integer less than or equal to expr.

Return type: integer

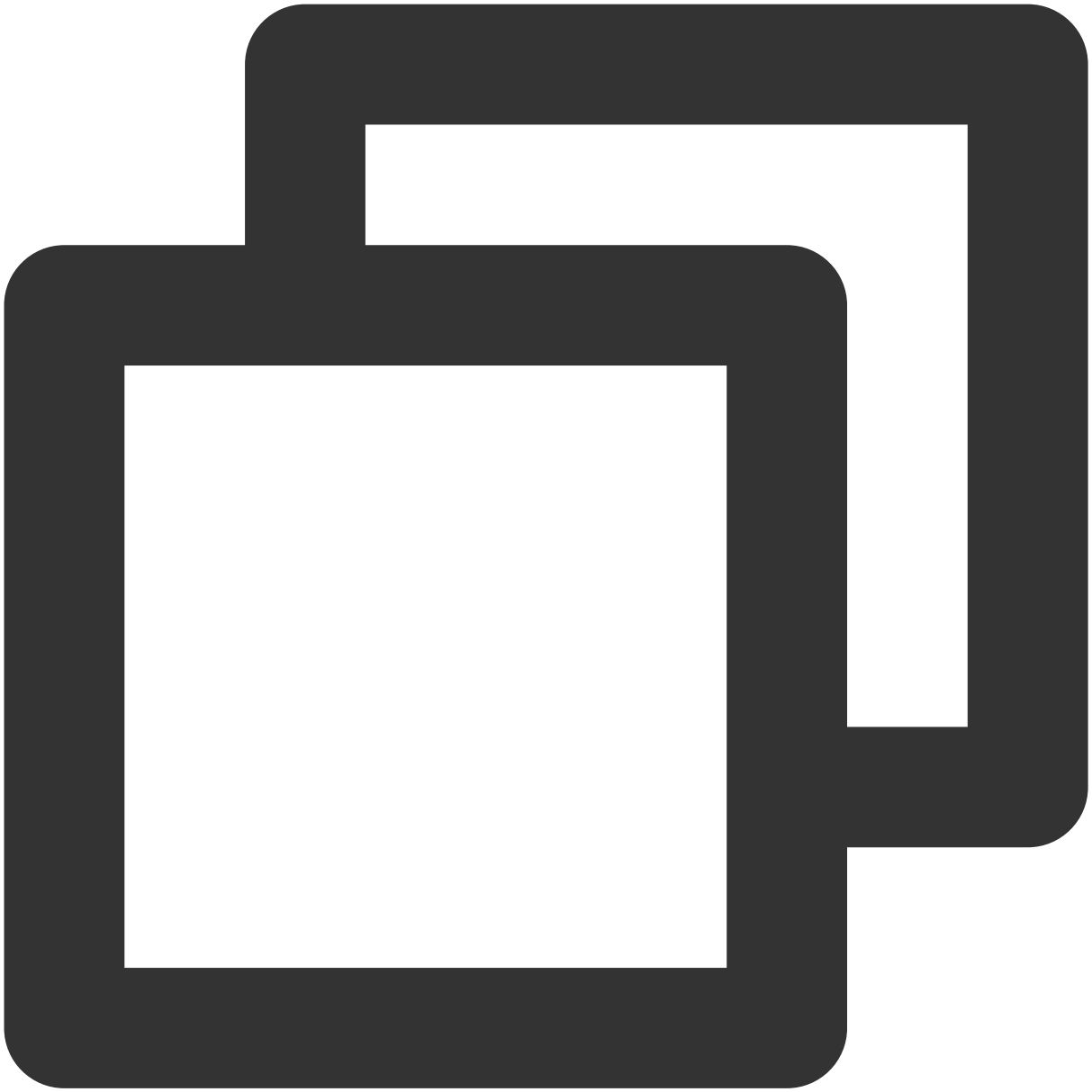
Example:



```
> SELECT floor(-0.1);
 -1
> SELECT floor(5);
 5
```

FACTORIAL

Function statement:



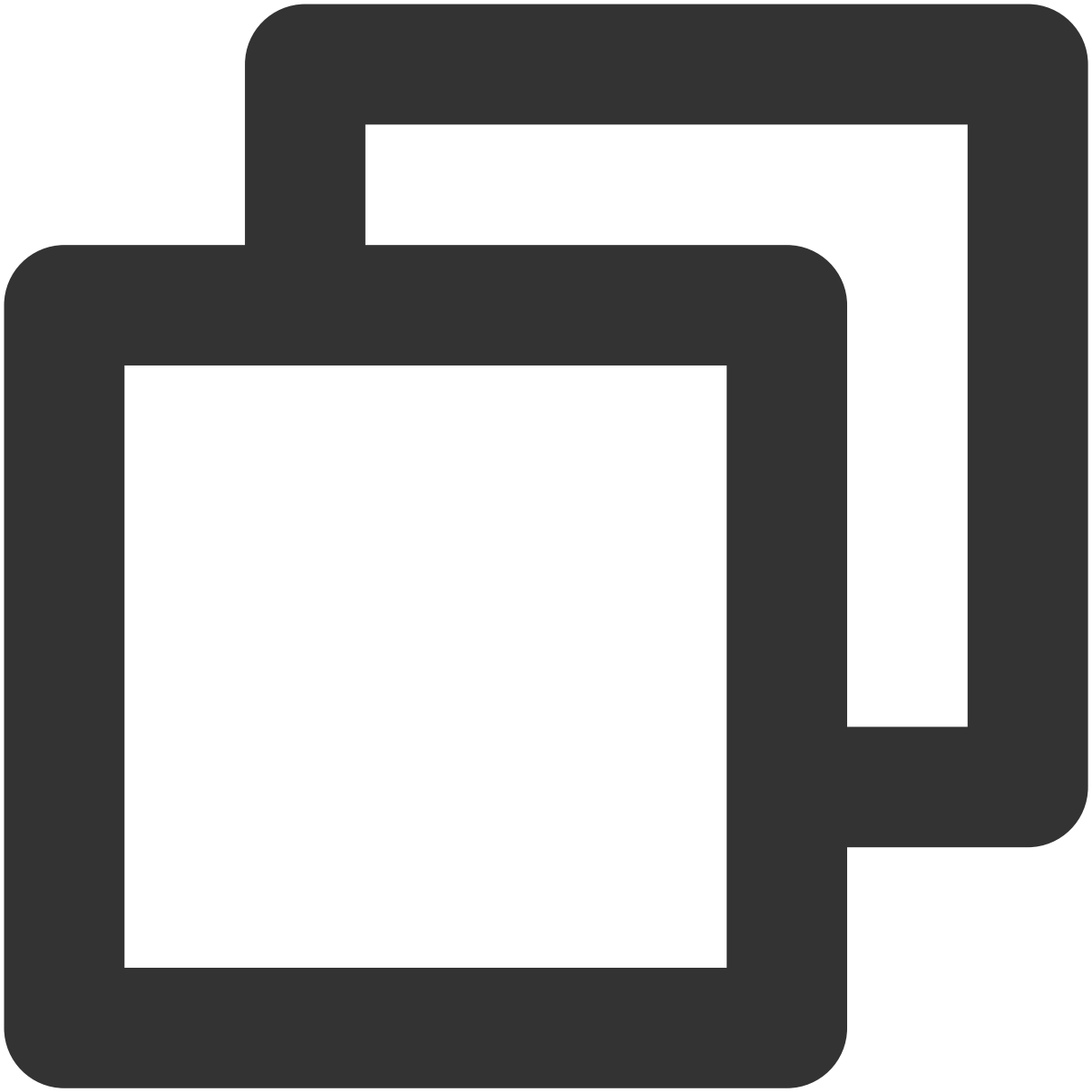
```
FACTORIAL(<expr> integer)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the factorial of expr. expr must be in the range [0..20]. Otherwise, NULL is returned.

Return type: bigint

Example:



```
> SELECT factorial(5);
 120
```

HEX

Function statement:

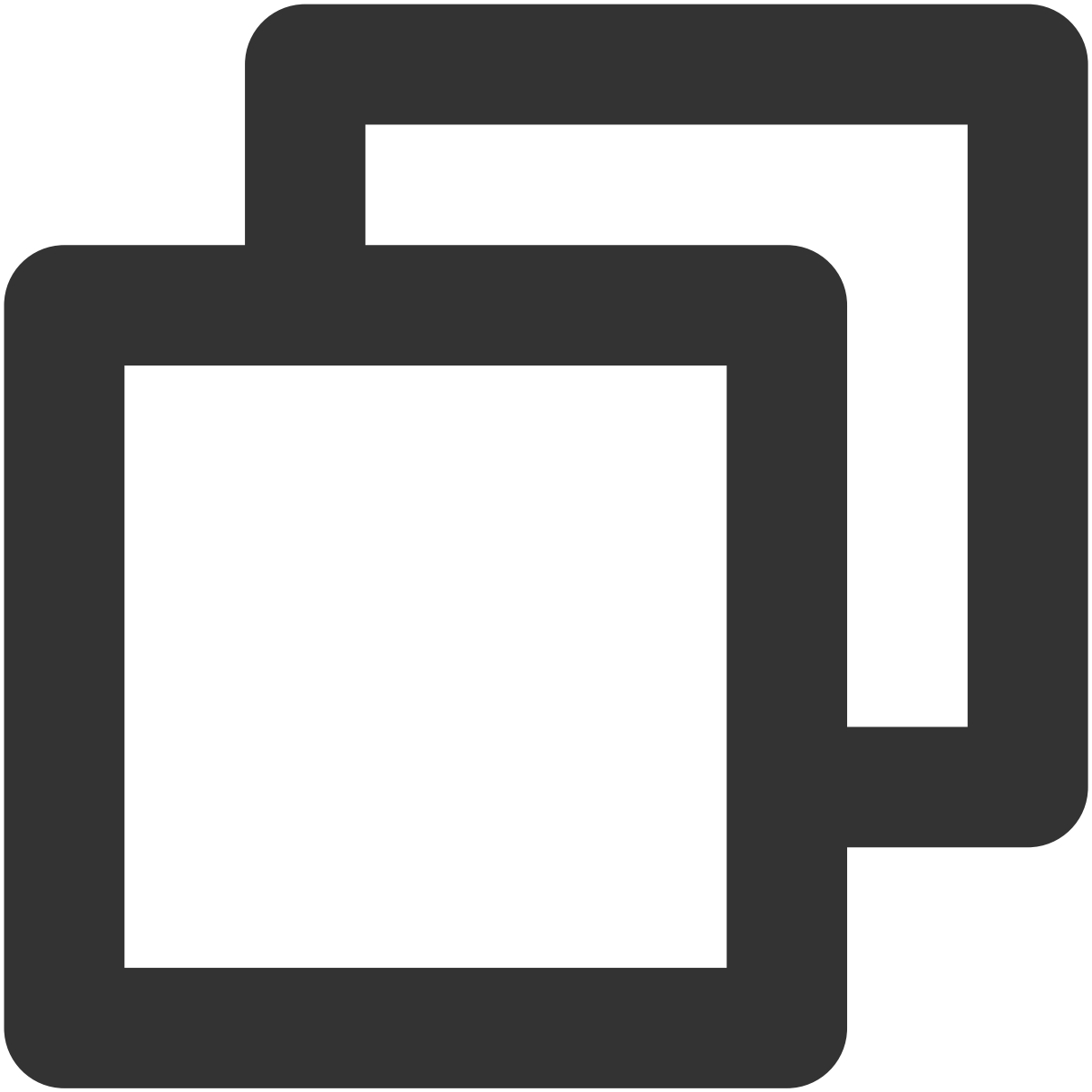
```
HEX(<expr> bigint|string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return expr as a hexadecimal string.

Return type: string

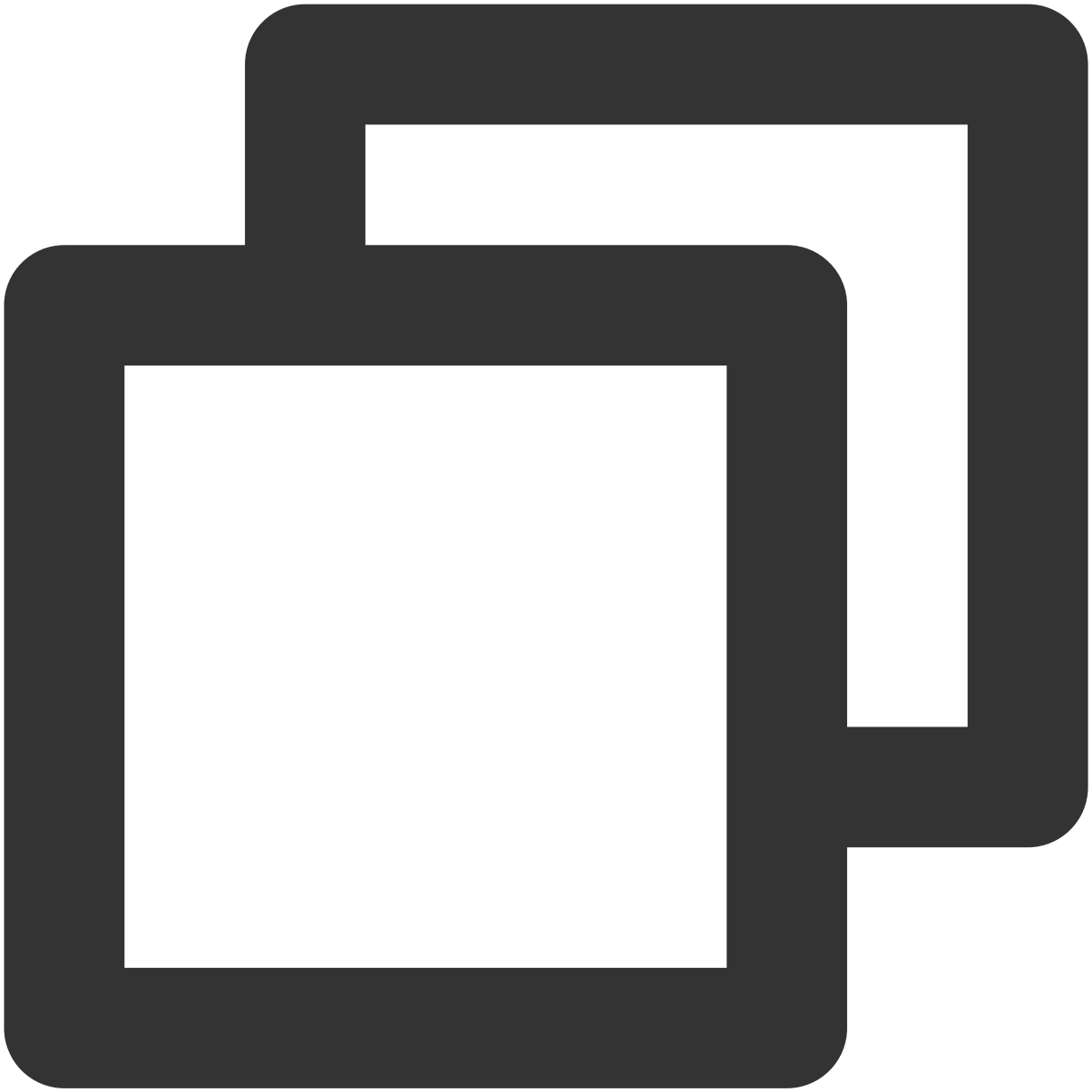
Example:



```
> SELECT hex(17);
 11
> SELECT hex('Spark SQL');
 537061726B2053514C
```

HYPOT

Function statement:



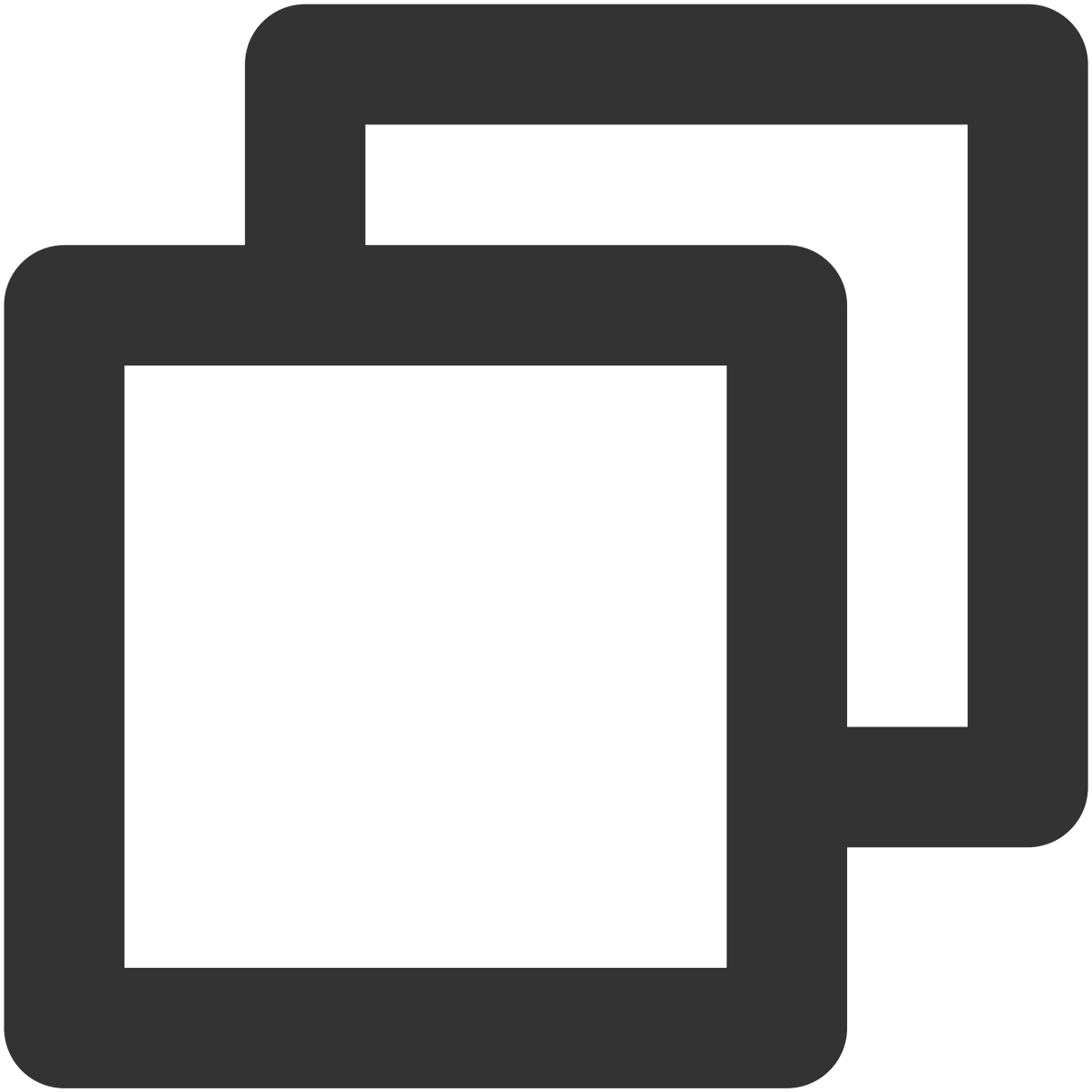
```
HYPOT(<expr1> integer|double|decimal, <expr2> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return sqrt(pow(expr1, 2) + pow(expr2, 2)).

Return type: double

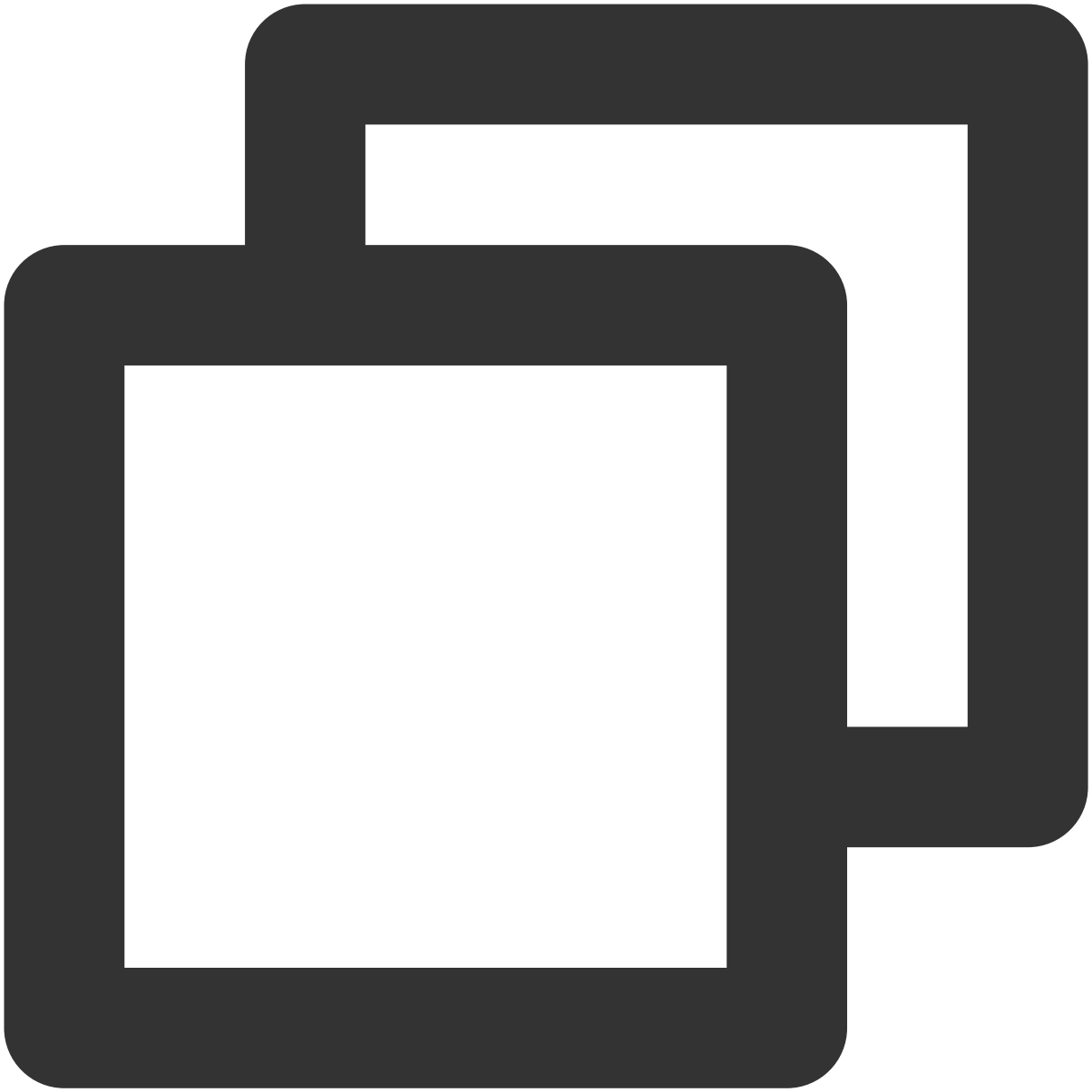
Example:



```
> SELECT hypot(3, 4);
 5.0
```

LOG

Function statement:



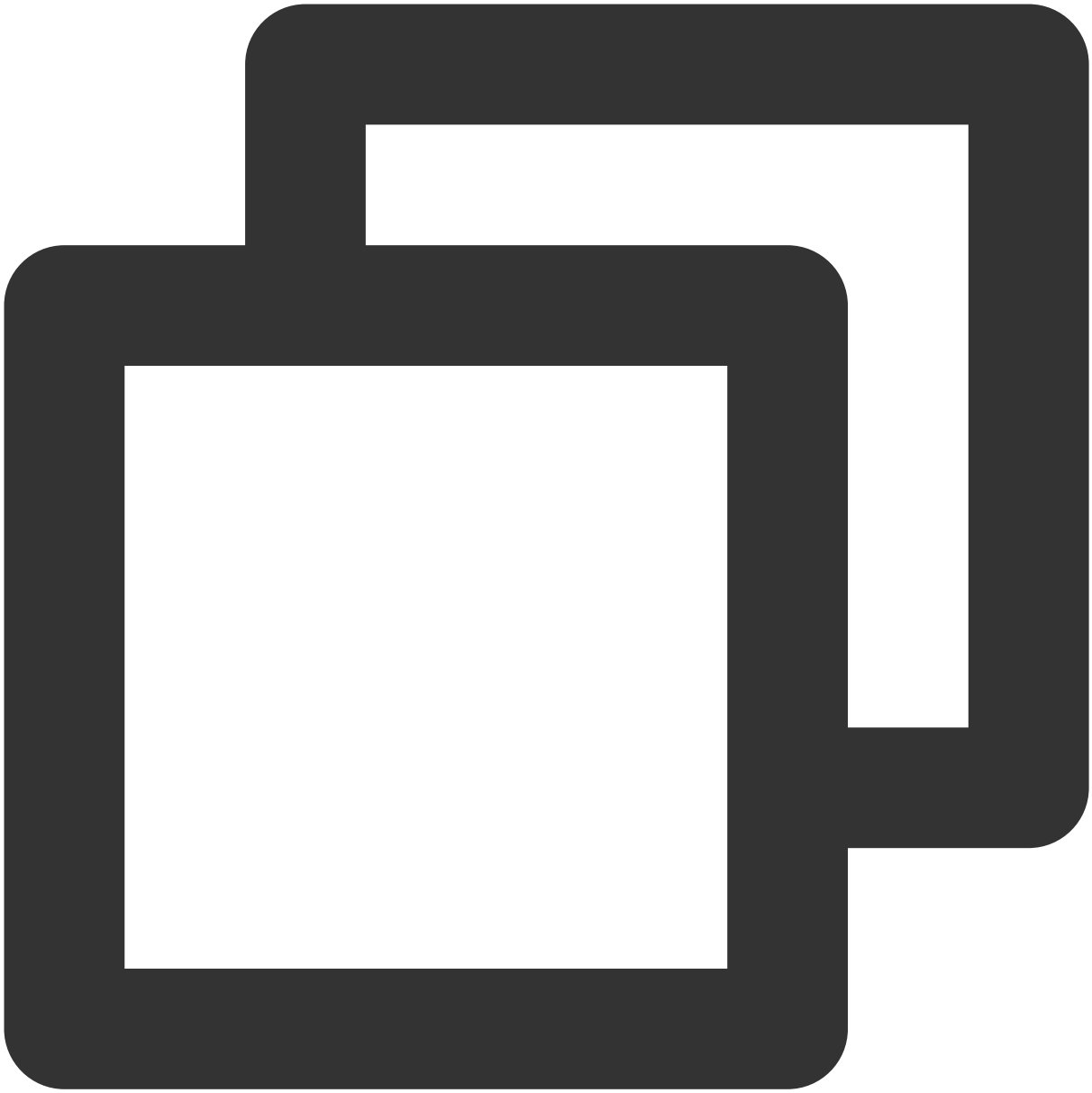
```
LOG(<base> integer|double|decimal, <expr> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the logarithm of expr with base as the base.

Return type: double

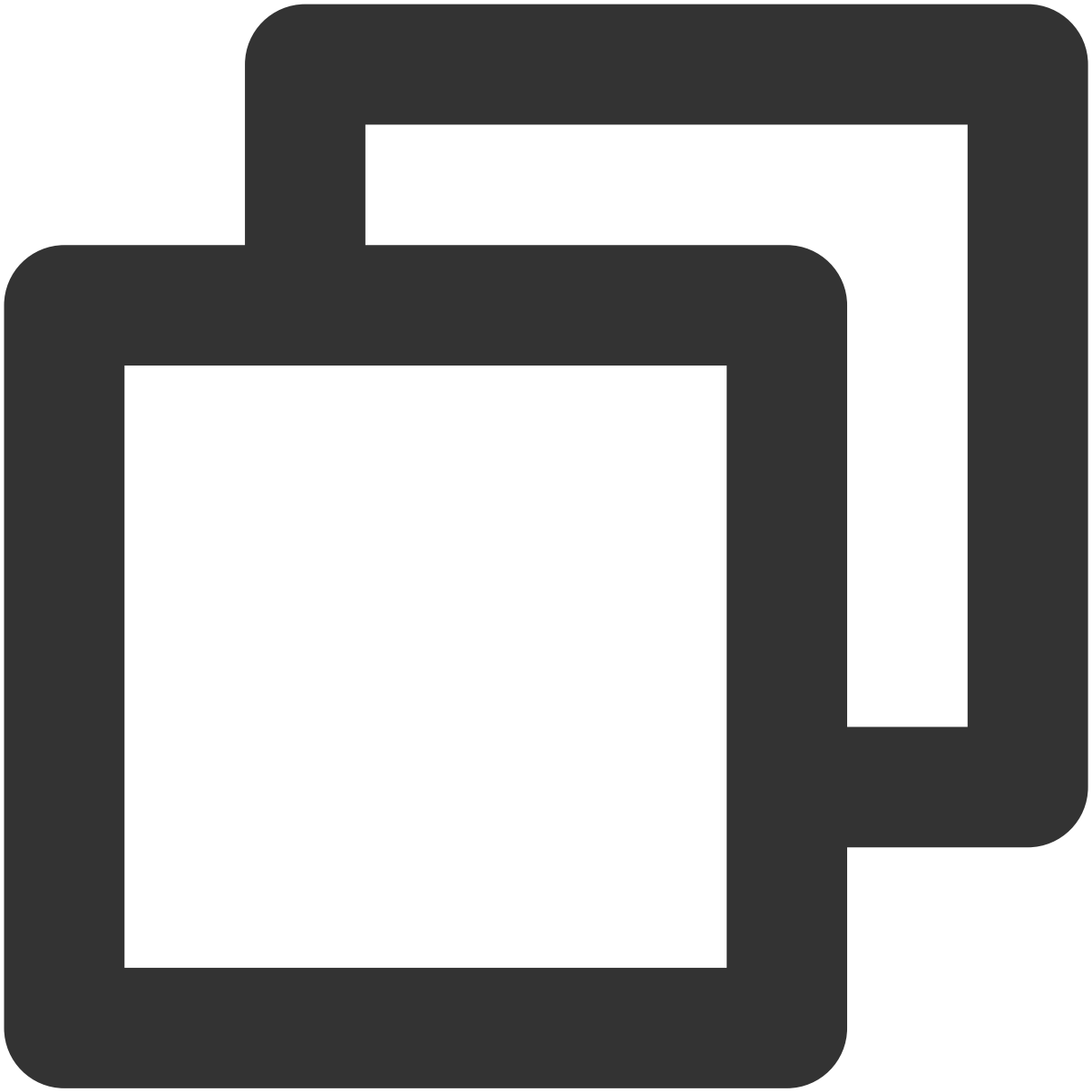
Example:



```
> SELECT log(10, 100);
 2.0
```

LOG10

Function statement:



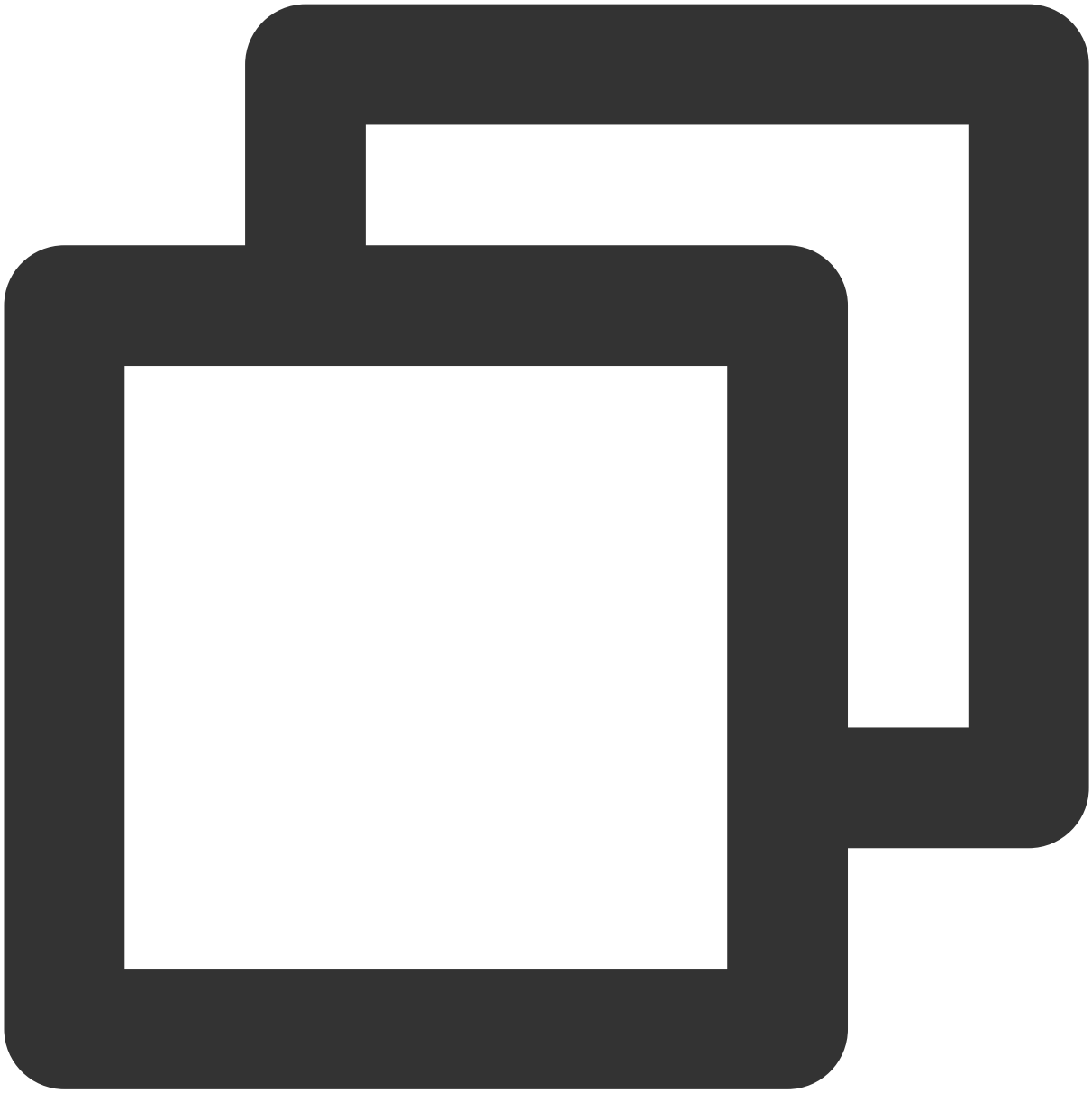
```
LOG10(<expr> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the base-10 logarithm of expr.

Return type: double

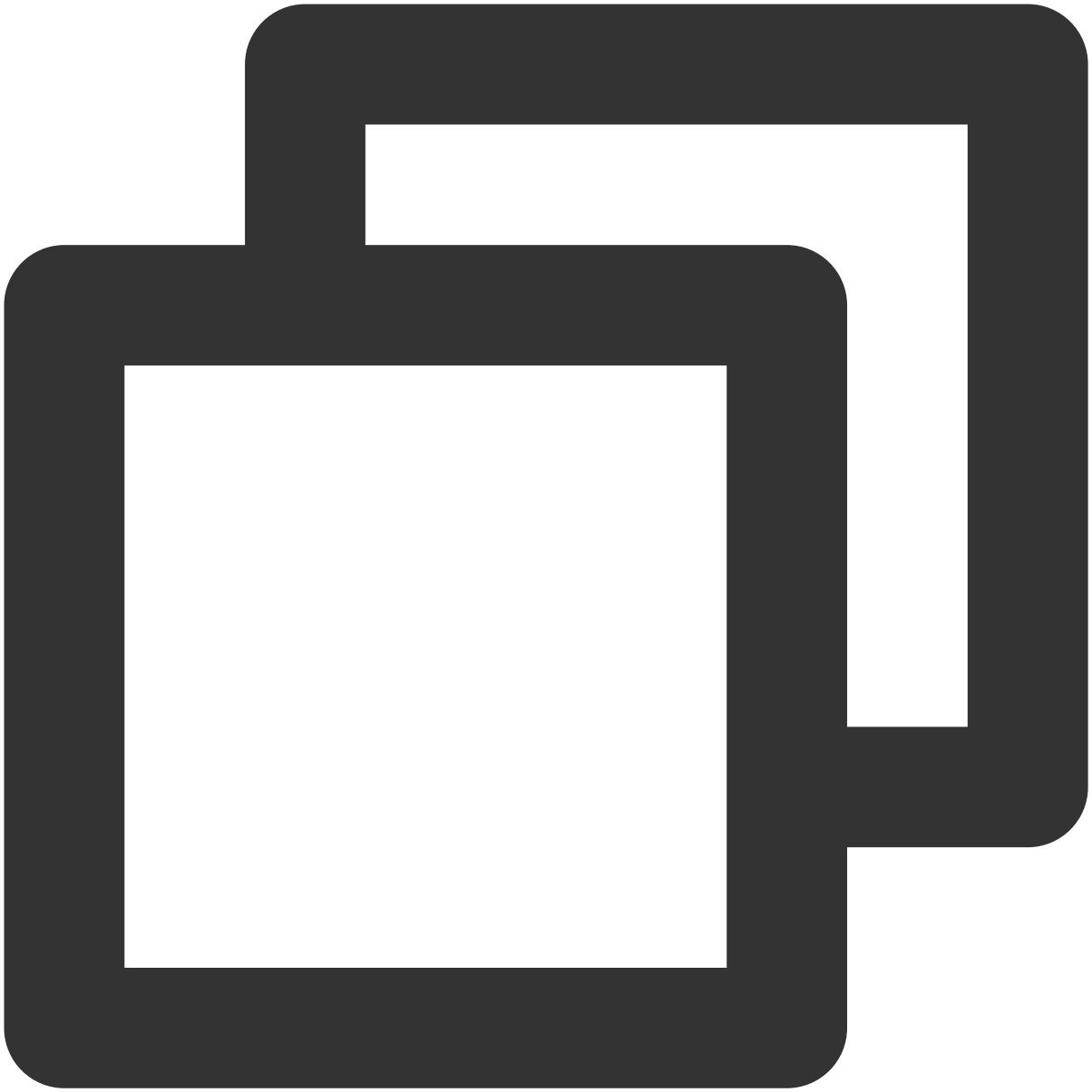
Example:



```
> SELECT log10(10);
 1.0
```

LOG1P

Function statement:



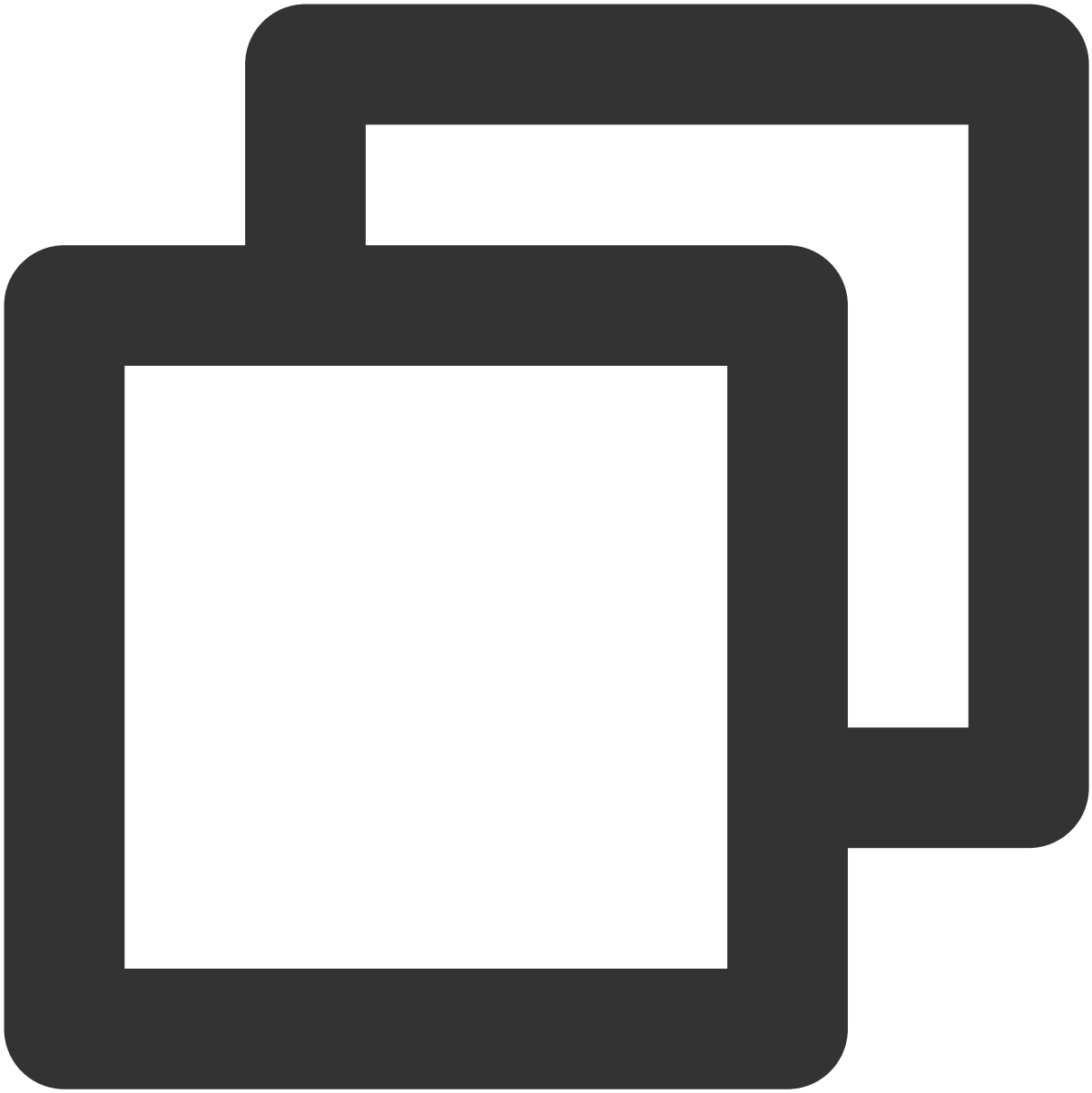
```
LOG1P(<expr> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return log(1 + expr).

Return type: double

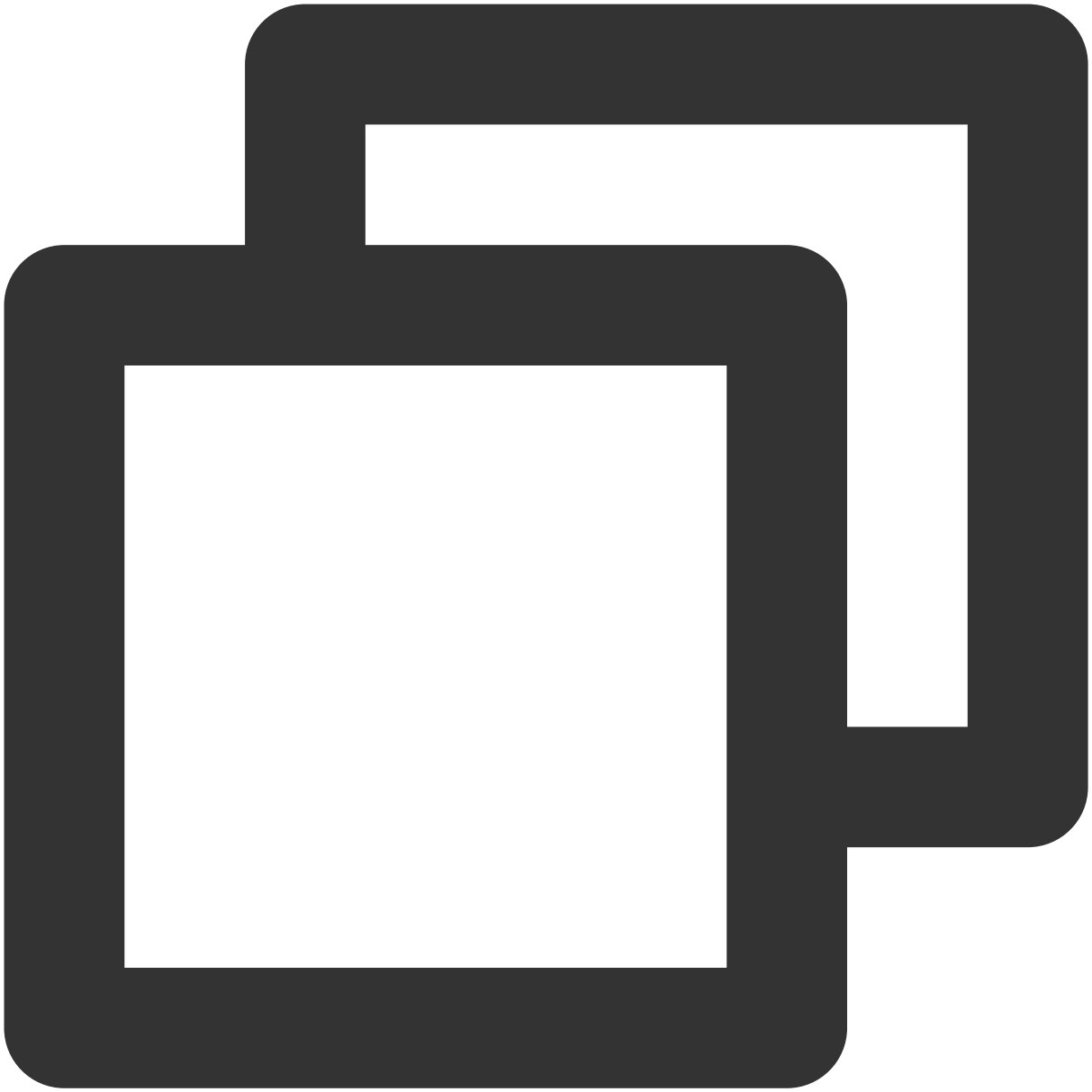
Example:



```
> SELECT log1p(0);
 0.0
```

LOG2

Function statement:



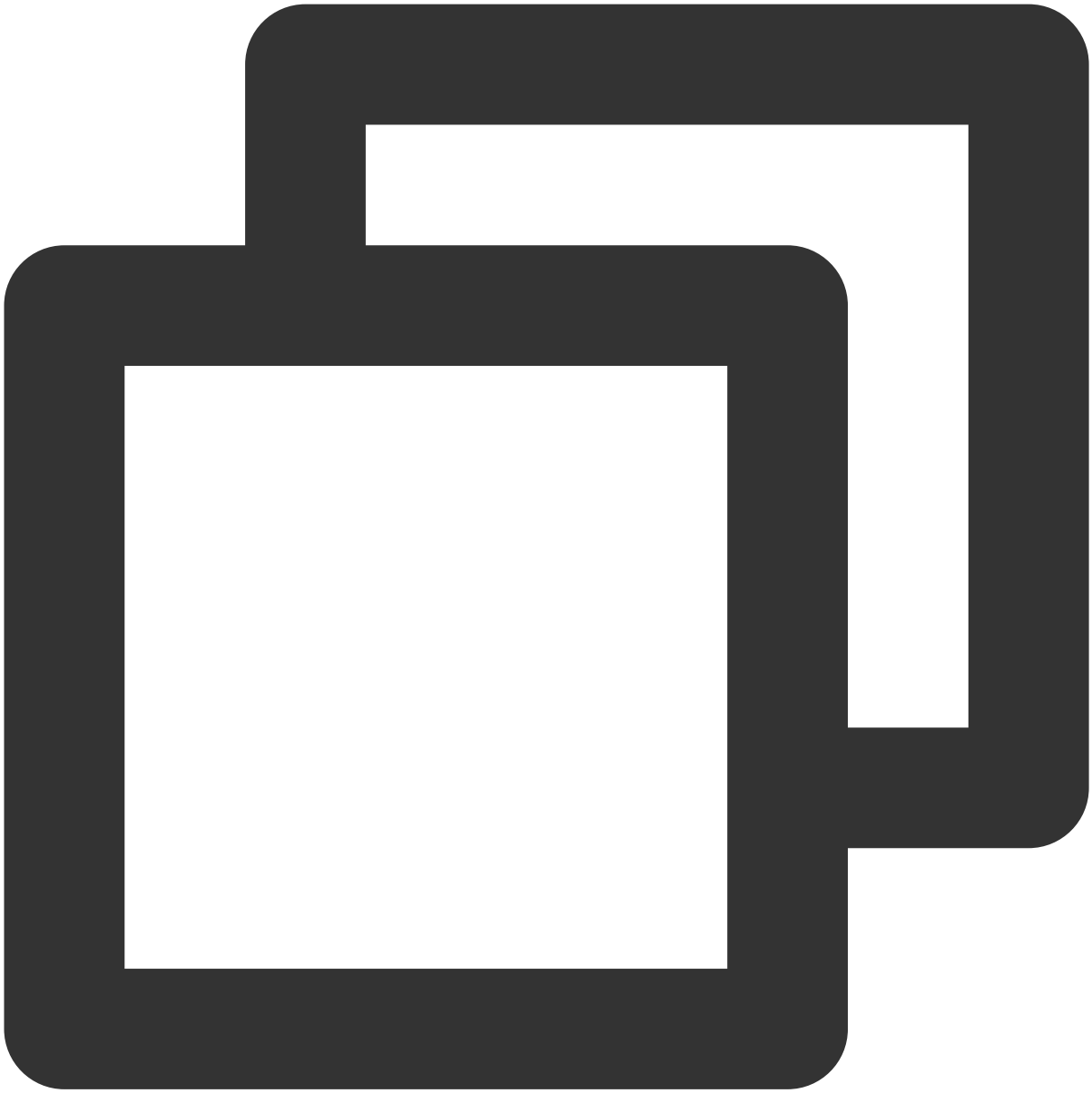
```
LOG2(<expr> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the base-2 logarithm of expr.

Return type: double

Example:



```
> SELECT log2(2);
 1.0
```

LN

Function statement:

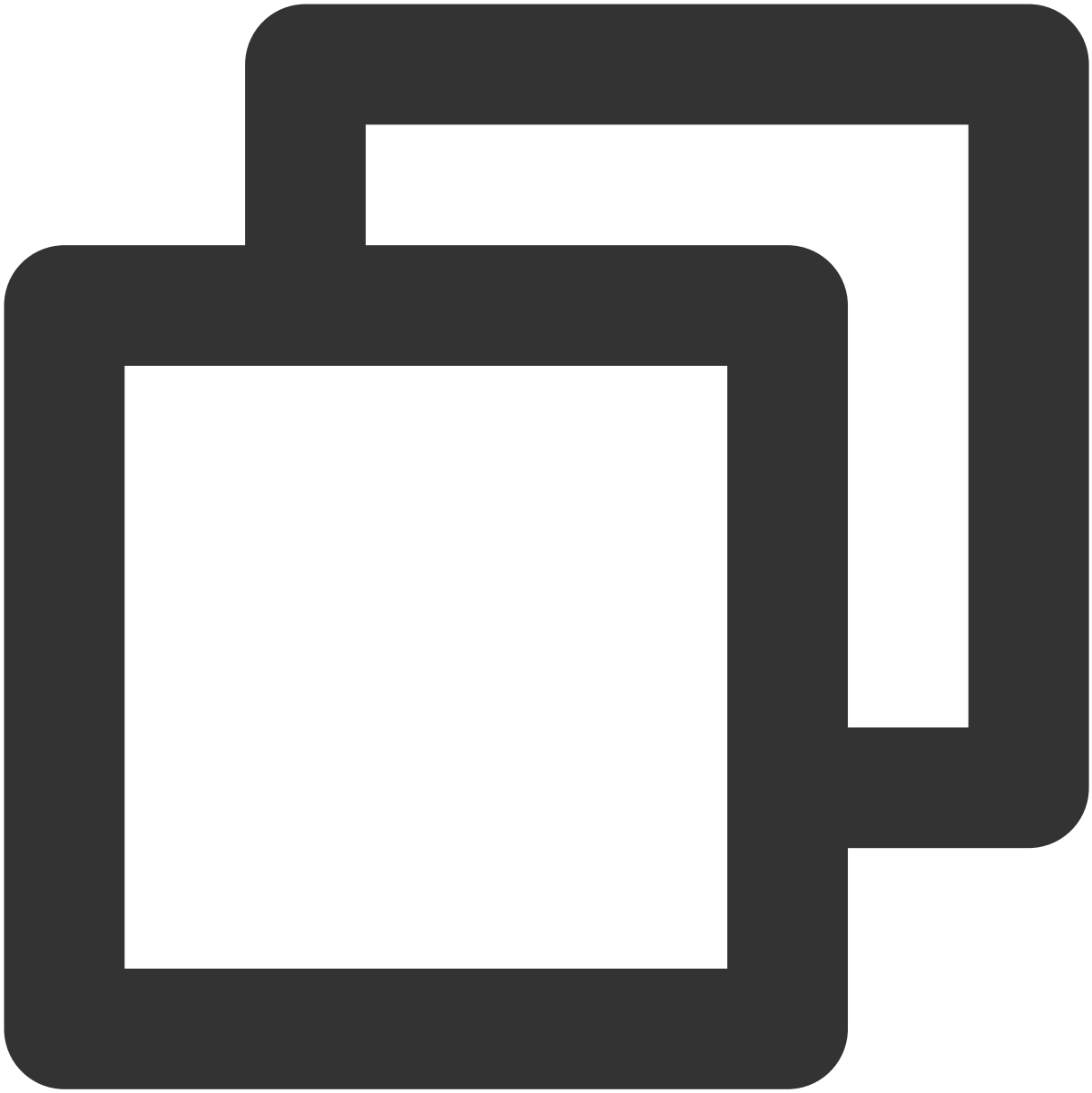
```
LN(<expr> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the natural logarithm of expr (base e).

Return type: double

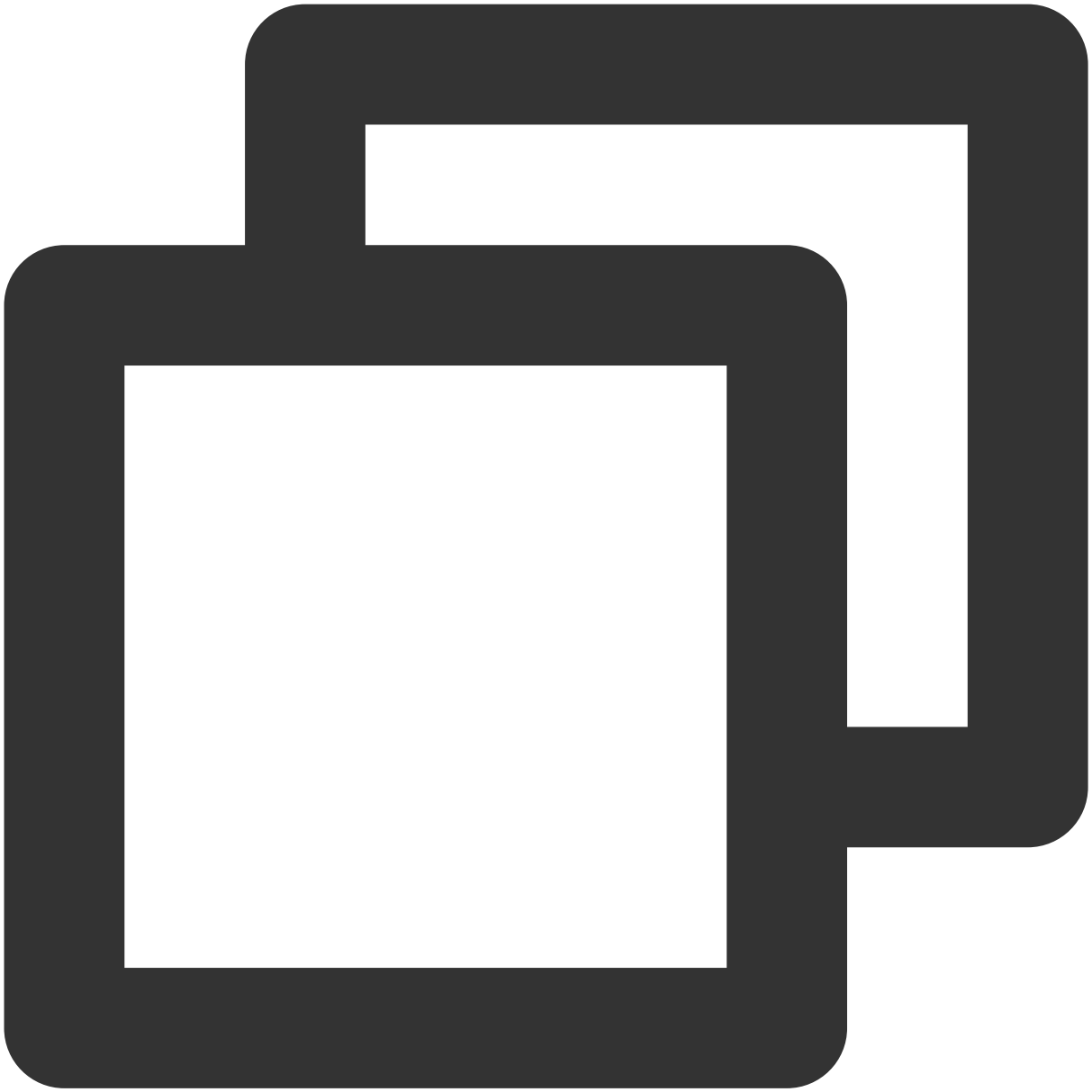
Example:



```
> SELECT ln(1);
 0.0
```

MOD

Function statement:



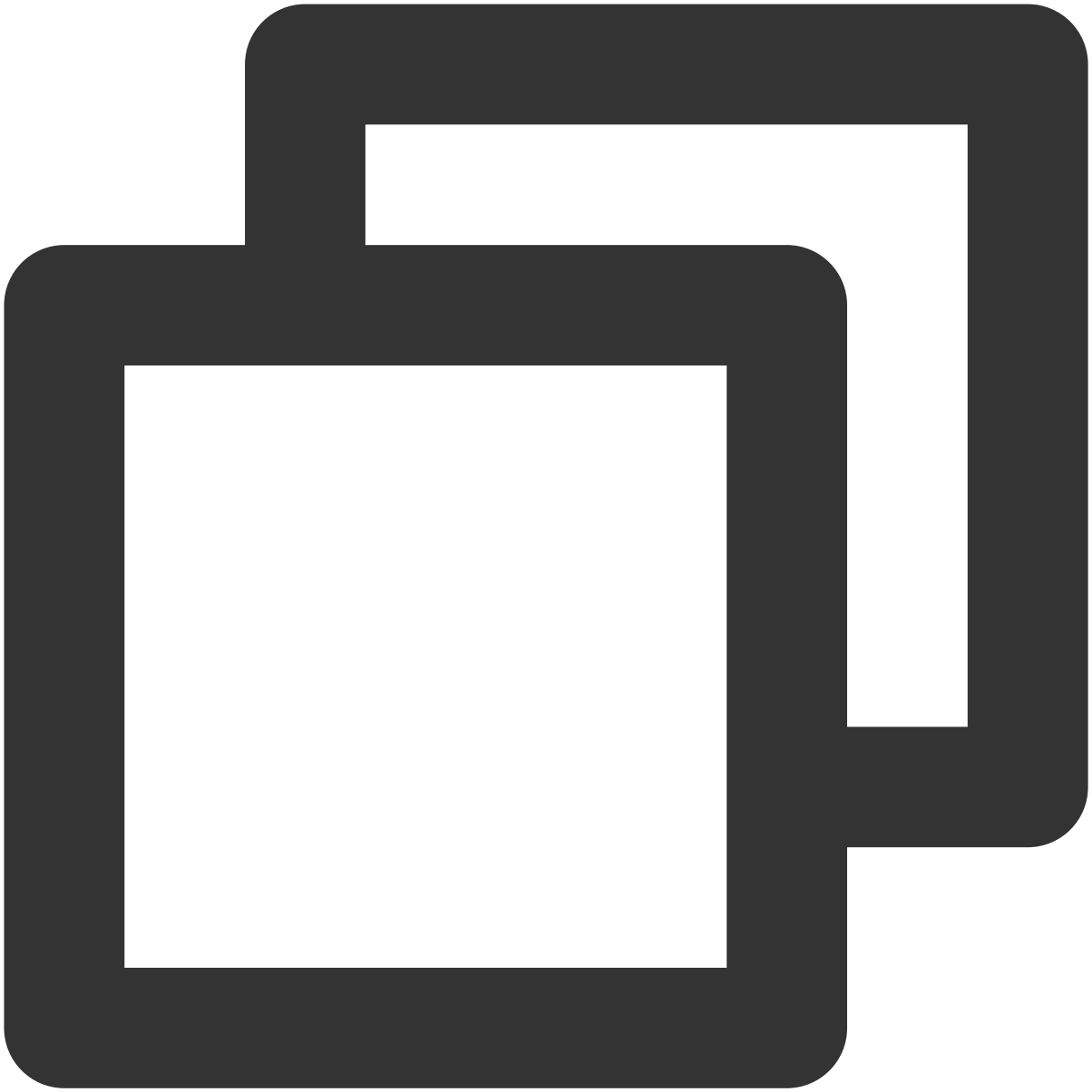
```
MOD(<expr1> integer|double|decimal, <expr2> integer|double|decimal)
<expr1> MOD <expr2>
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the remainder of expr1/expr2.

Return type: double|integer

Example:



```
> SELECT 2 % 1.8;
 0.2
> SELECT MOD(2, 1.8);
 0.2
```

NEGATIVE

Function statement:

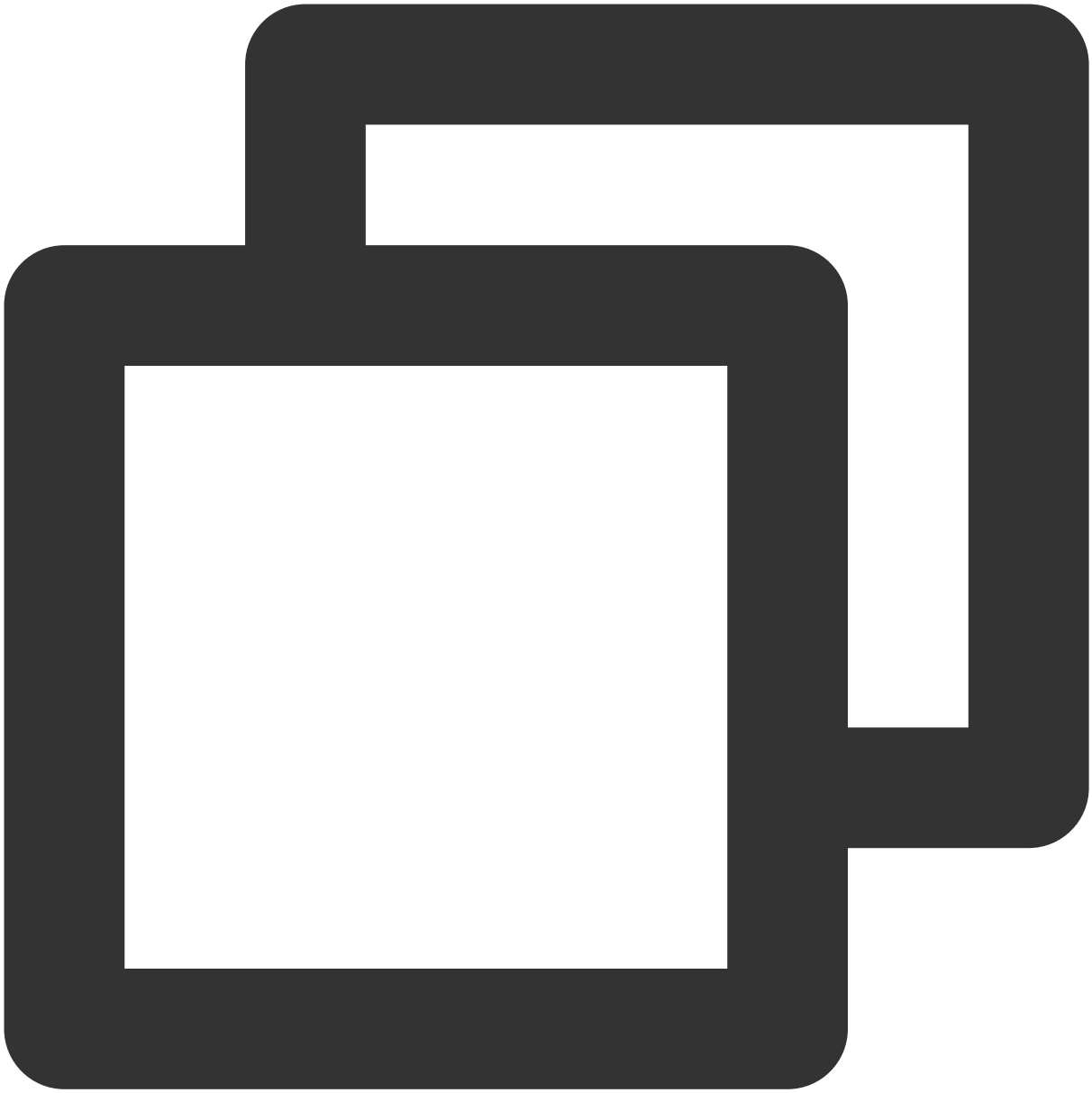
```
NEGATIVE(<expr> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the negative of expr.

Return type: Same as expr.

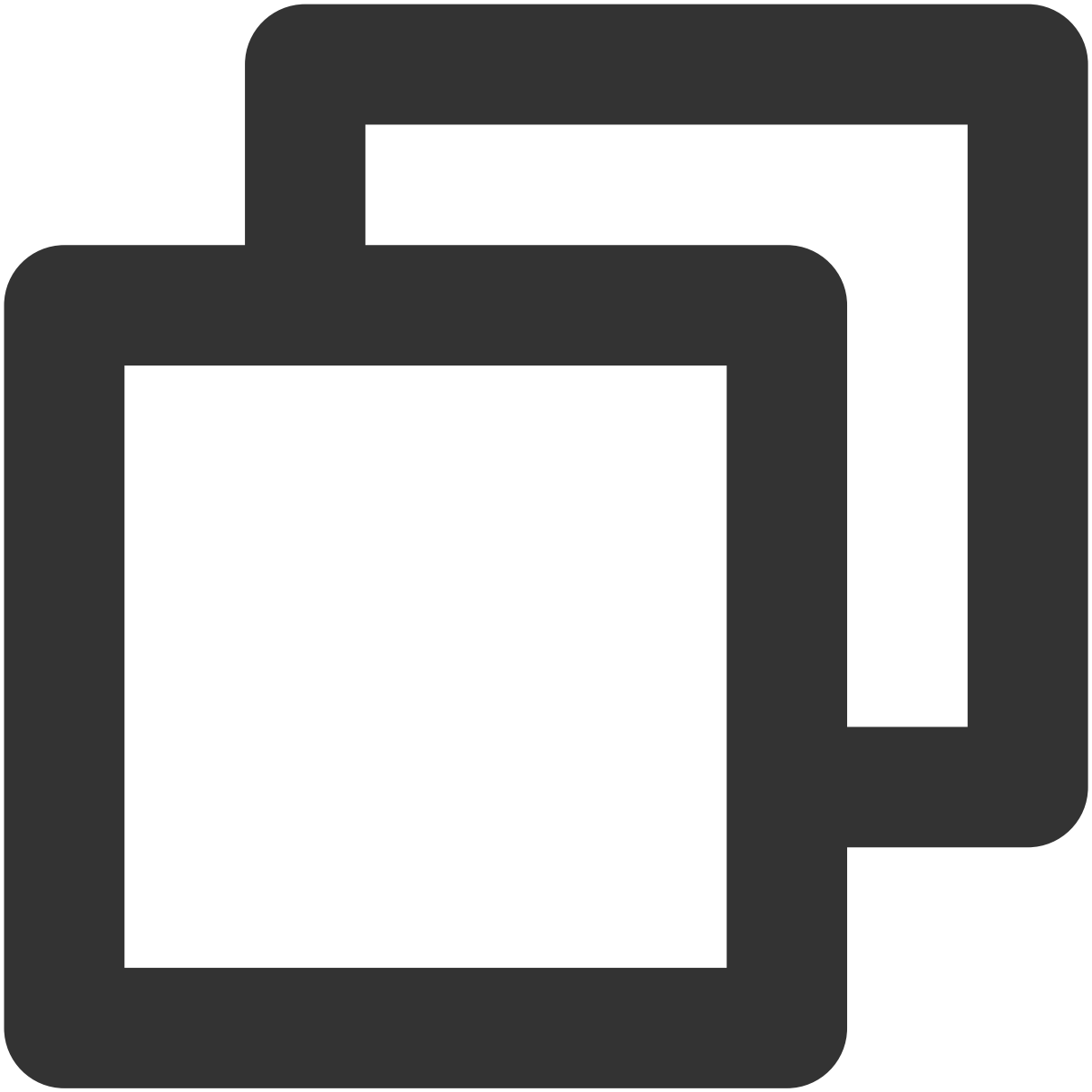
Example:



```
> SELECT negative(1);
 -1
```

PI

Function statement:



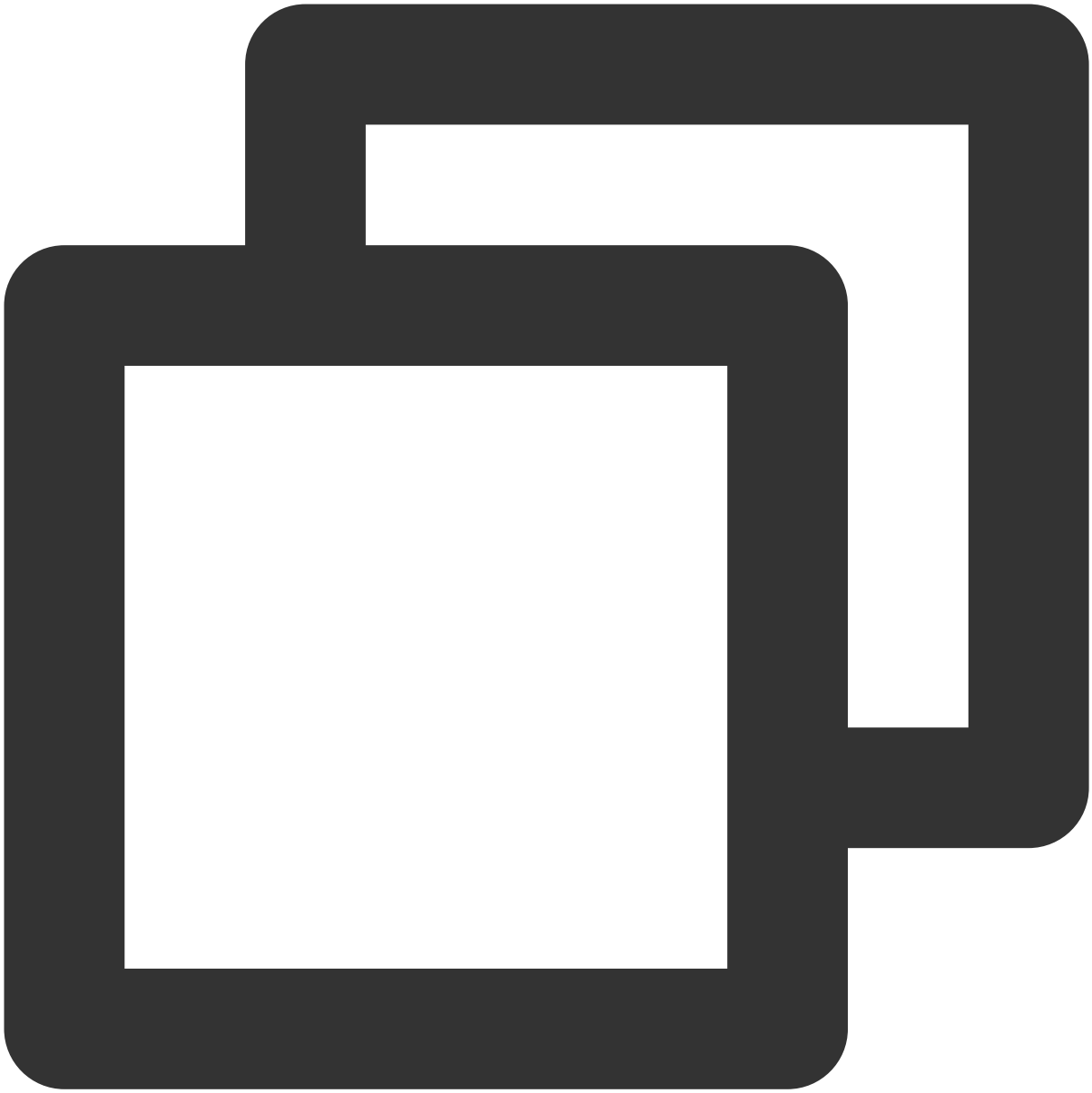
```
PI()
```

Supported engines: SparkSQL and Presto

Usage instructions: Return PI.

Return type: double

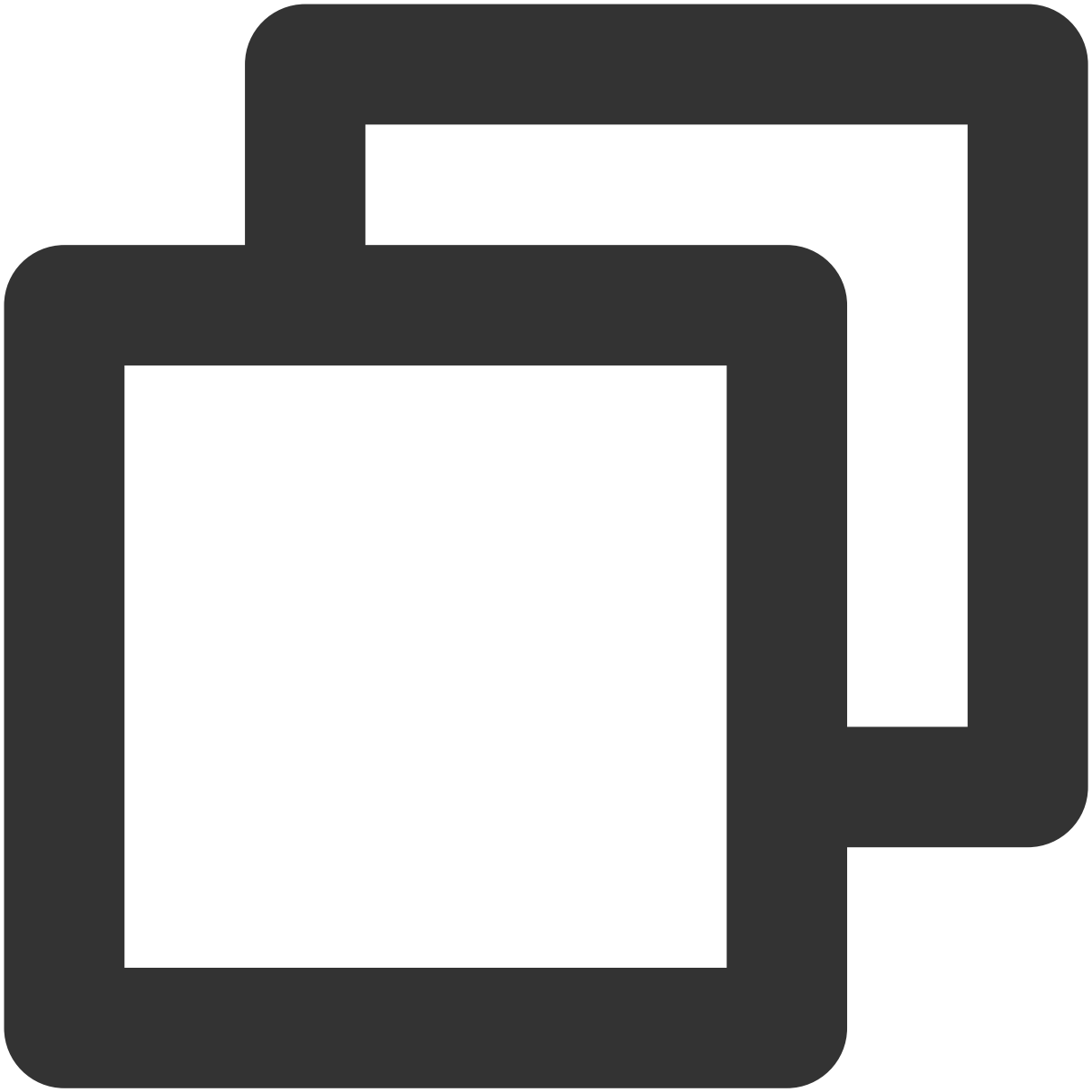
Example:



```
> SELECT pi();
 3.141592653589793
```

PMOD

Function statement:



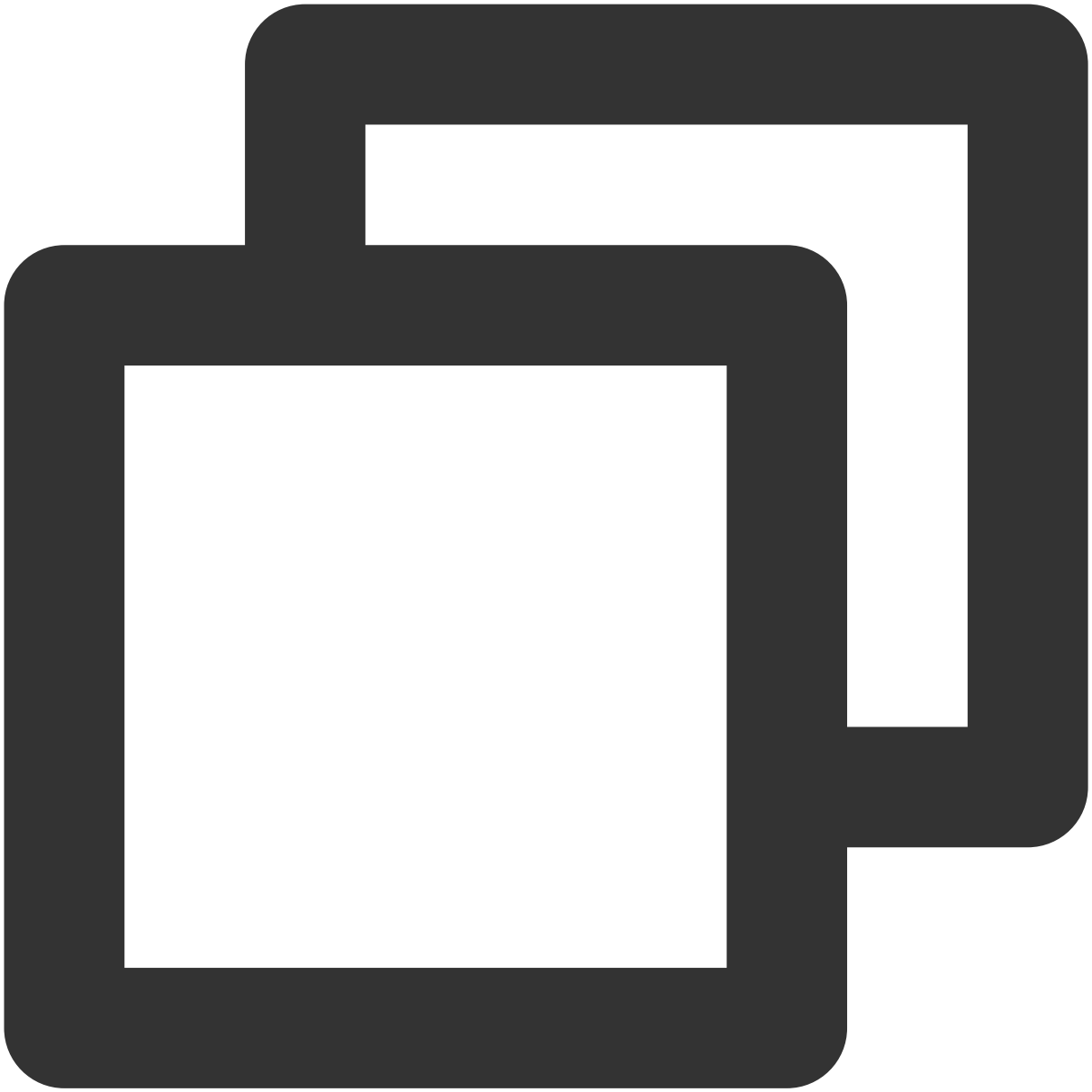
```
PMOD(<expr1> integer|double|decimal, <expr2> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the positive value of expr1 mod expr2.

Return type: double

Example:



```
> SELECT pmod(10, 3);
 1
> SELECT pmod(-10, 3);
 2
```

POSITIVE

Function statement:

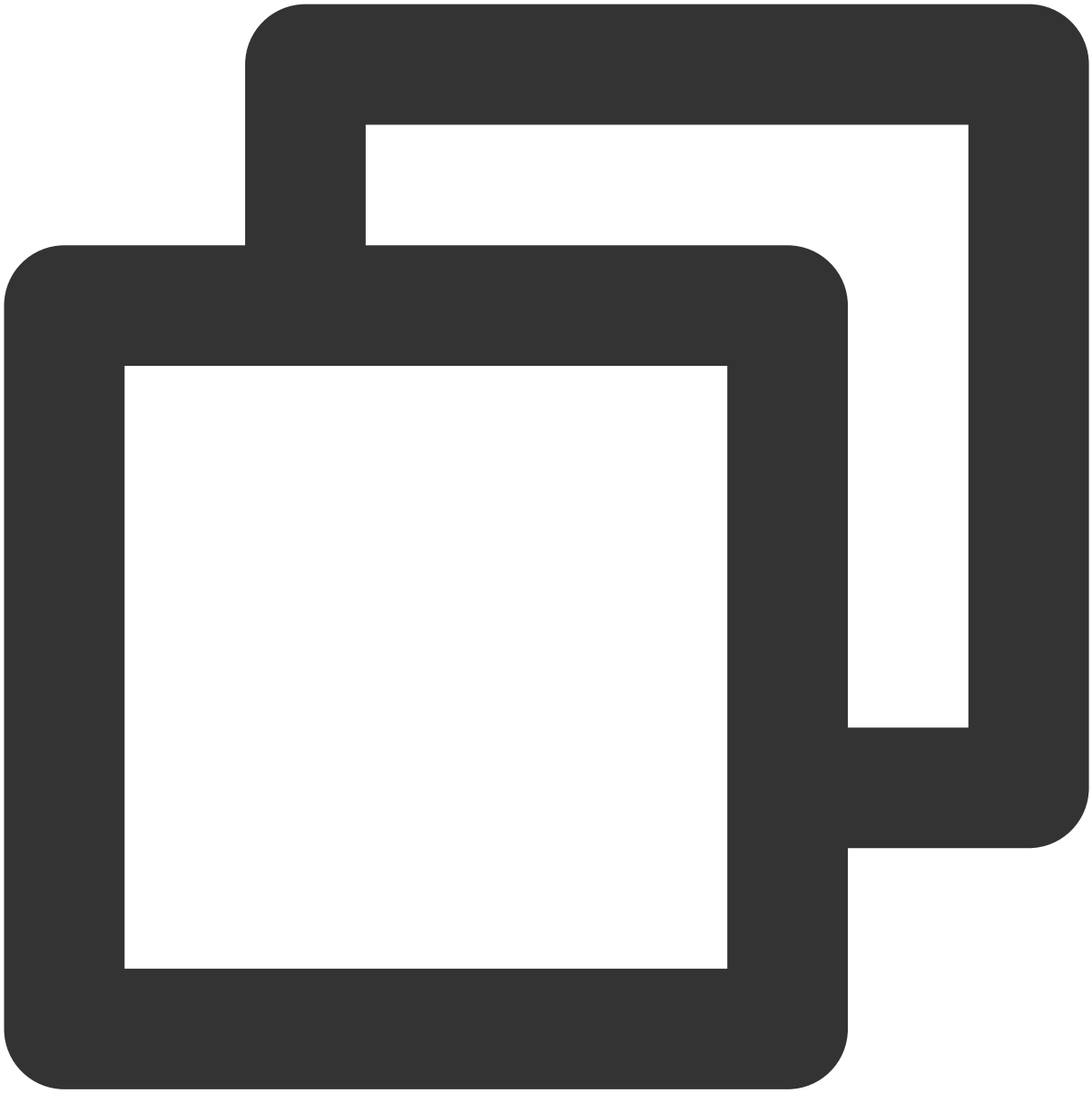
```
POSITIVE(<expr> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return expr.

Return type: Same as expr.

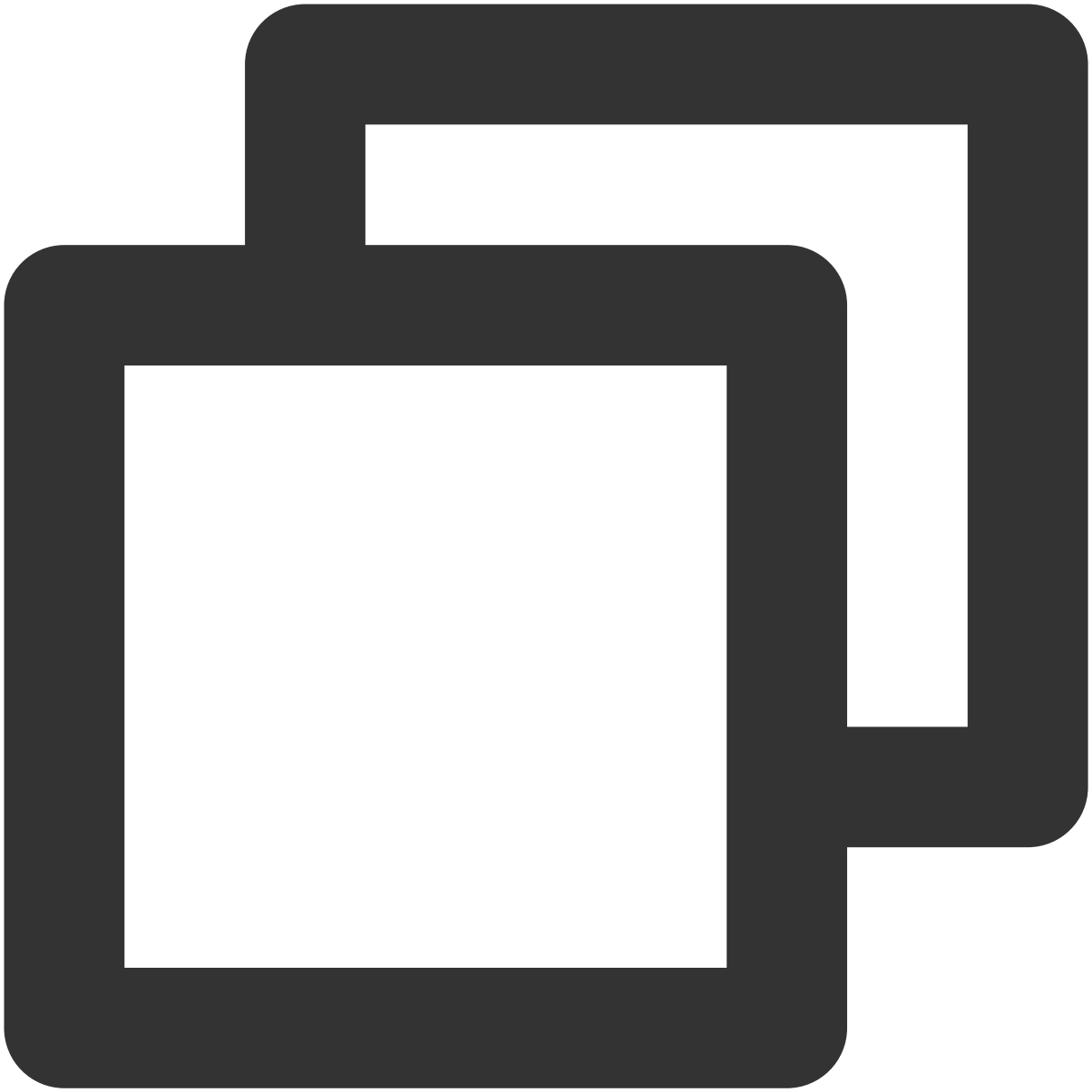
Example:



```
> SELECT positive(1);
 1
```

POWER

Function statement:



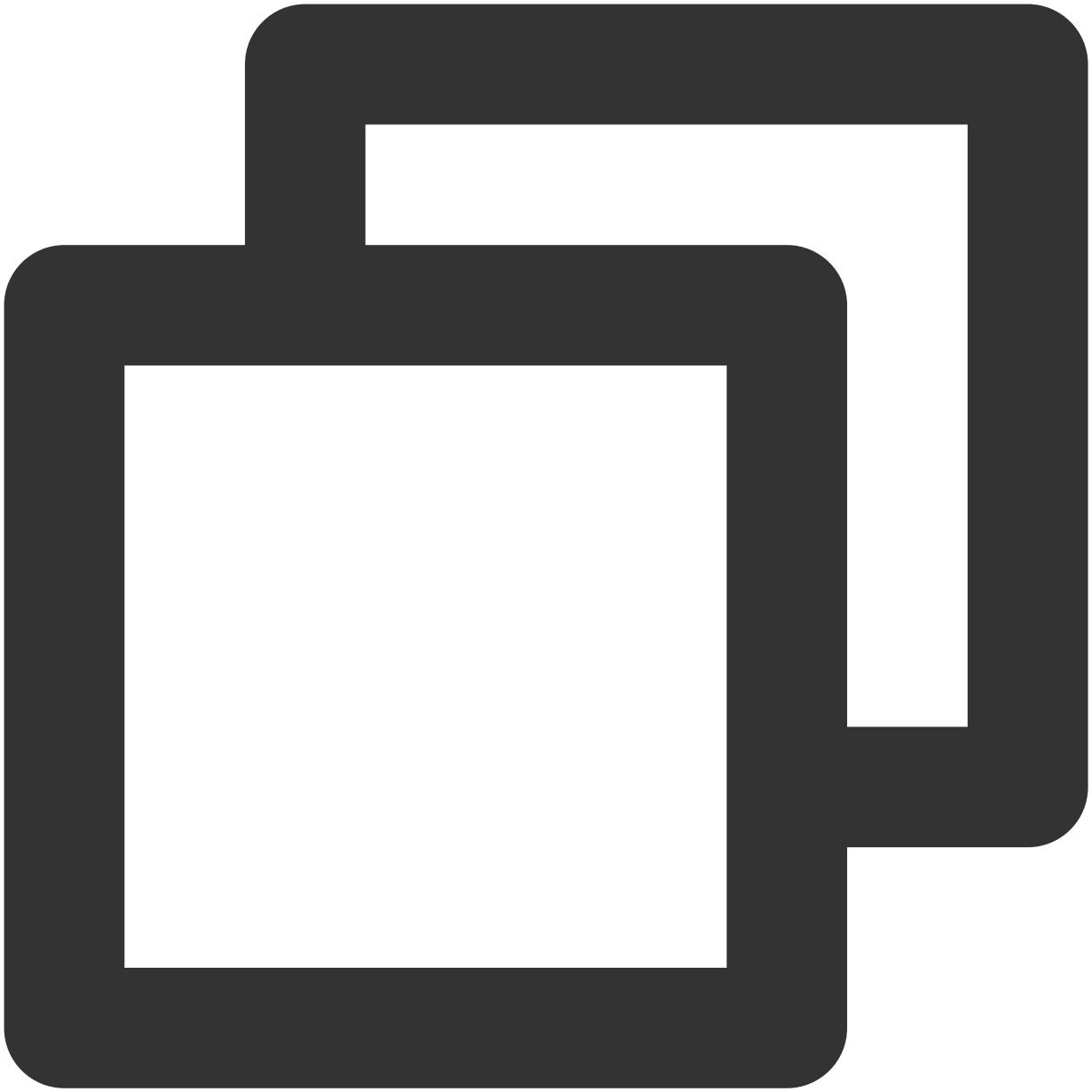
```
POWER(<base> integer|double|decimal, <number> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return base raised to the power of number.

Return type: double

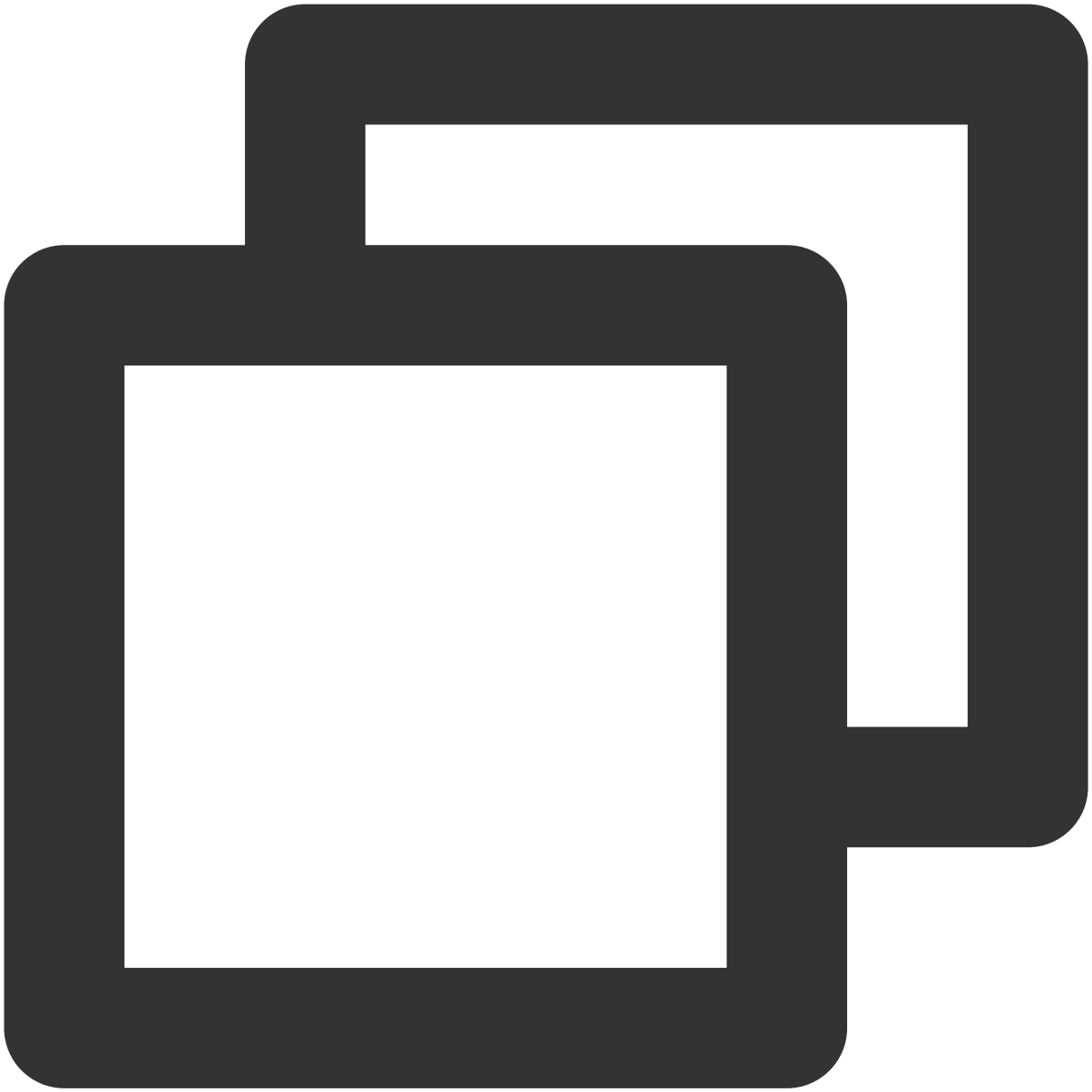
Example:



```
> SELECT power(2, 3);
 8.0
```

POW

Function statement:



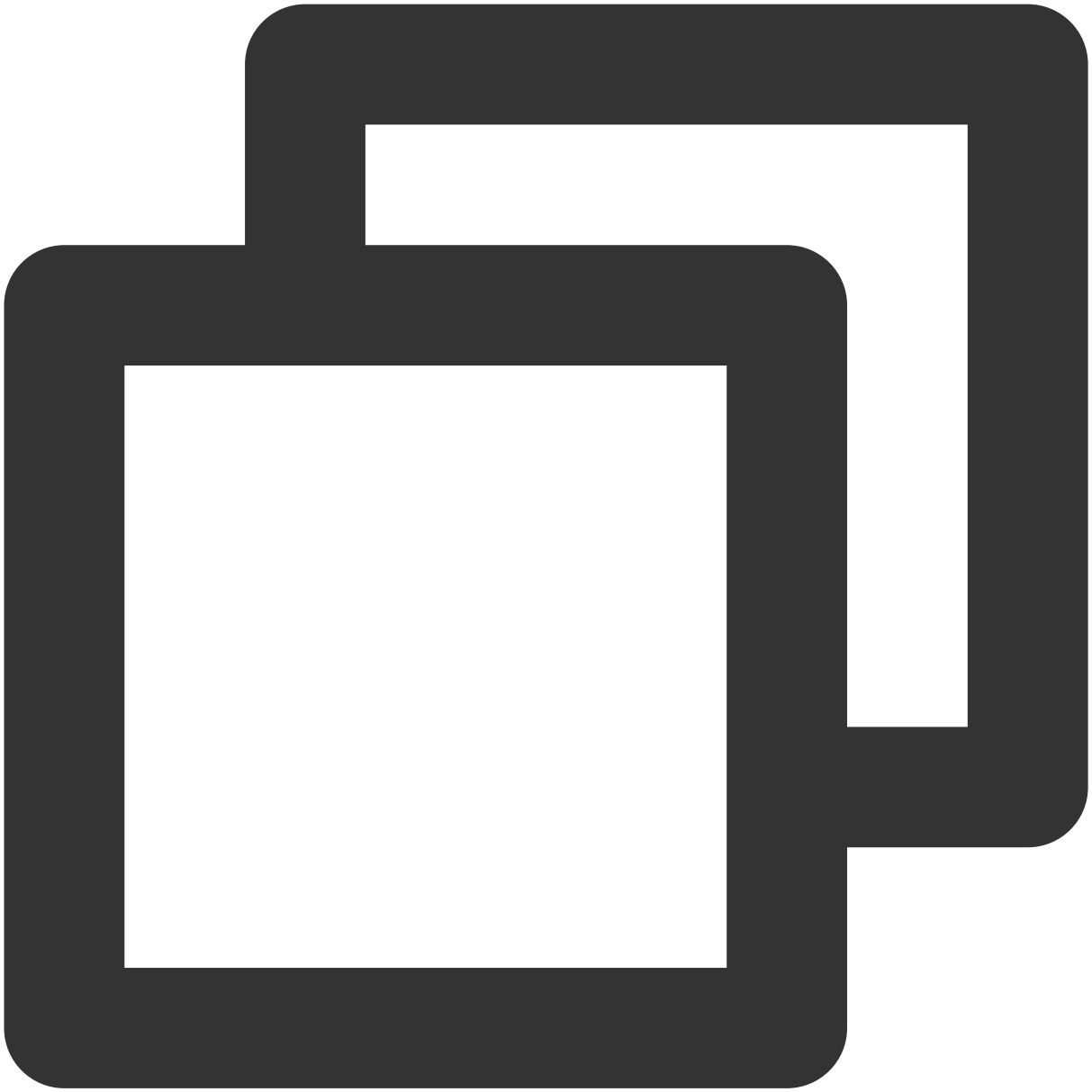
```
POW(<base> integer|double|decimal, <number> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return base raised to the power of number.

Return type: double

Example:



```
> SELECT pow(2, 3);
 8.0
```

RADIANS

Function statement:

```
RADIANS(<expr> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Convert degrees to radians.

Return type: double

Example:

```
> SELECT radians(180);
 3.141592653589793
```

RINT

Function statement:

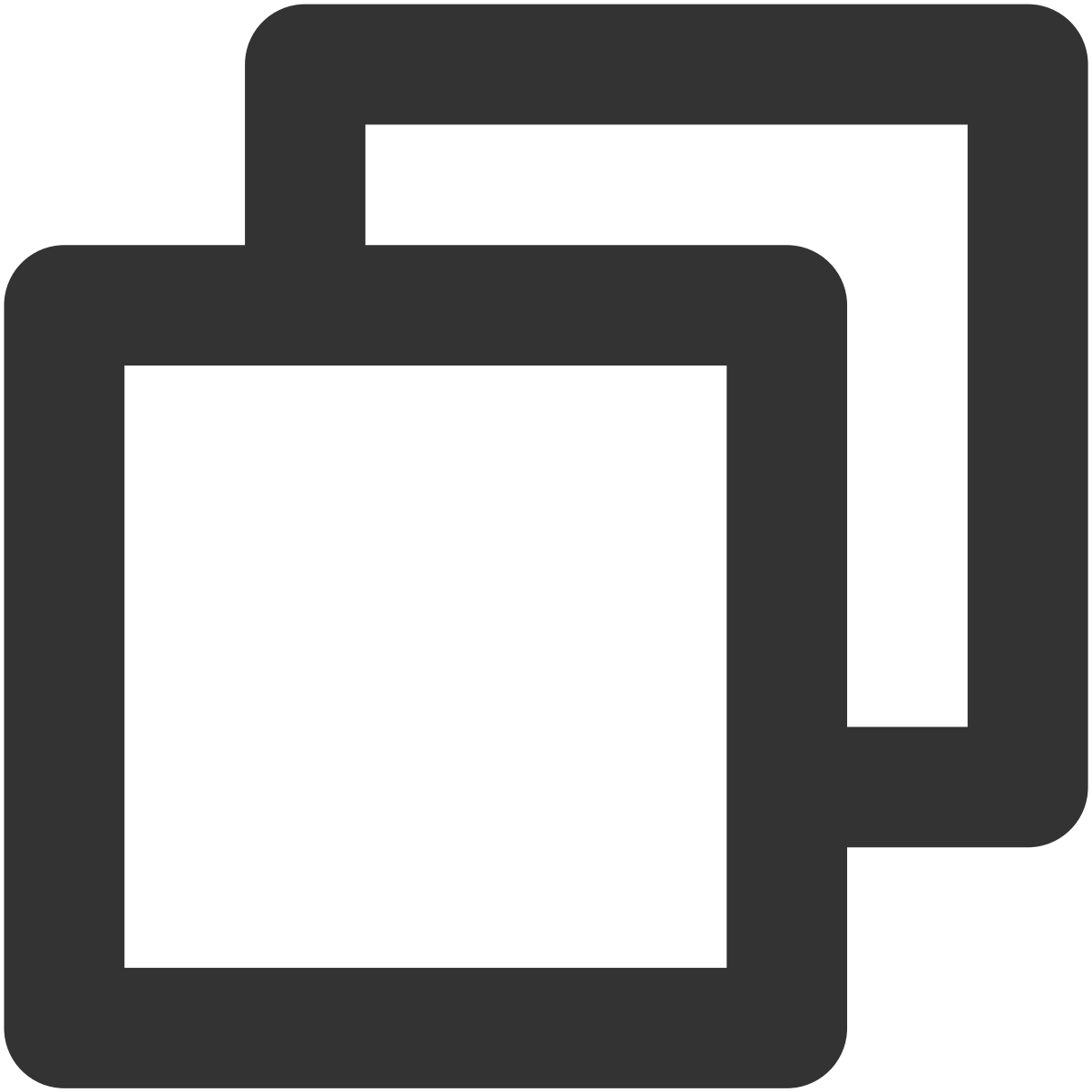
```
RINT(<expr> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the double value that is closest to the parameter and equal to a mathematical integer.

Return type: double

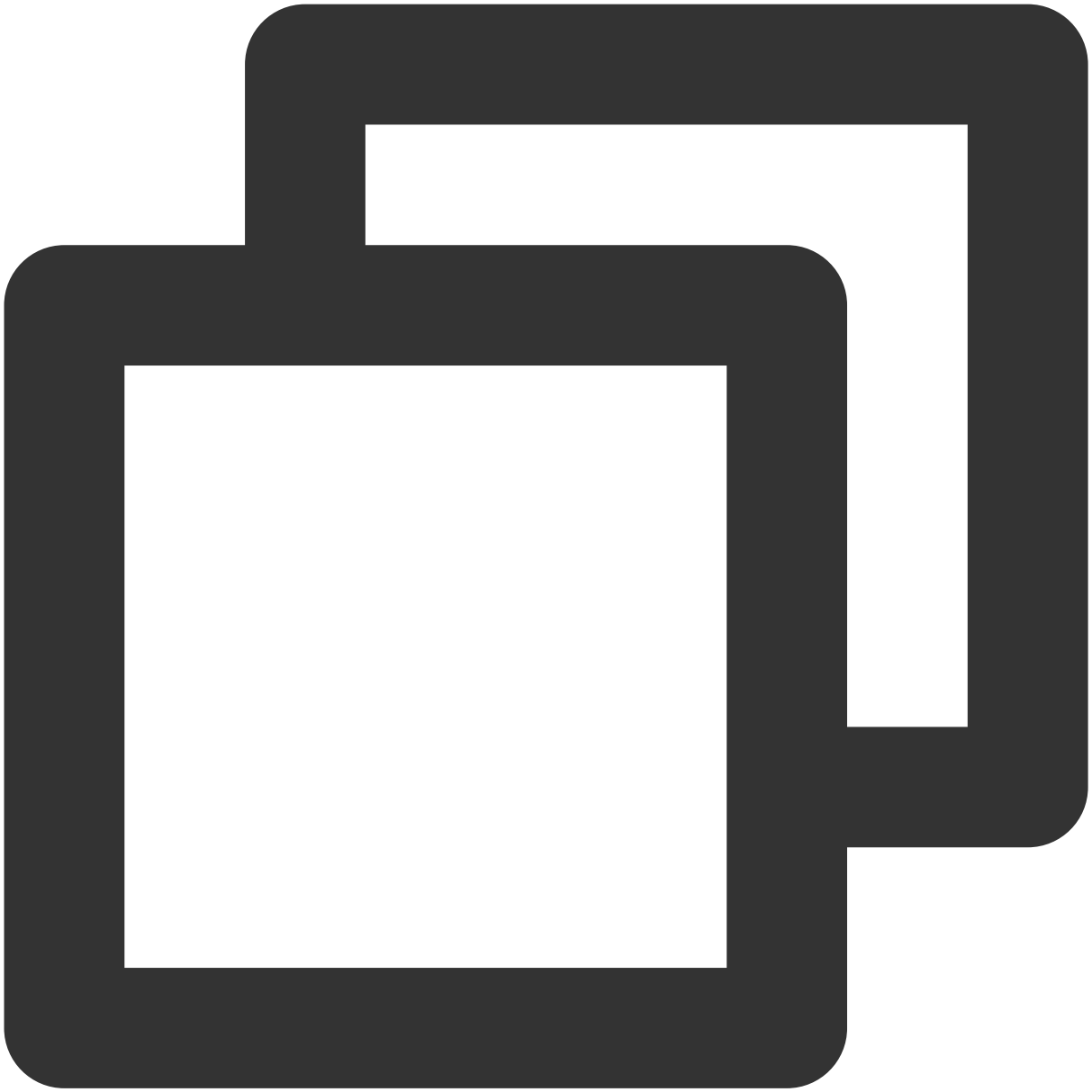
Example:



```
> SELECT rint(12.3456);
 12.0
```

ROUND

Function statement:



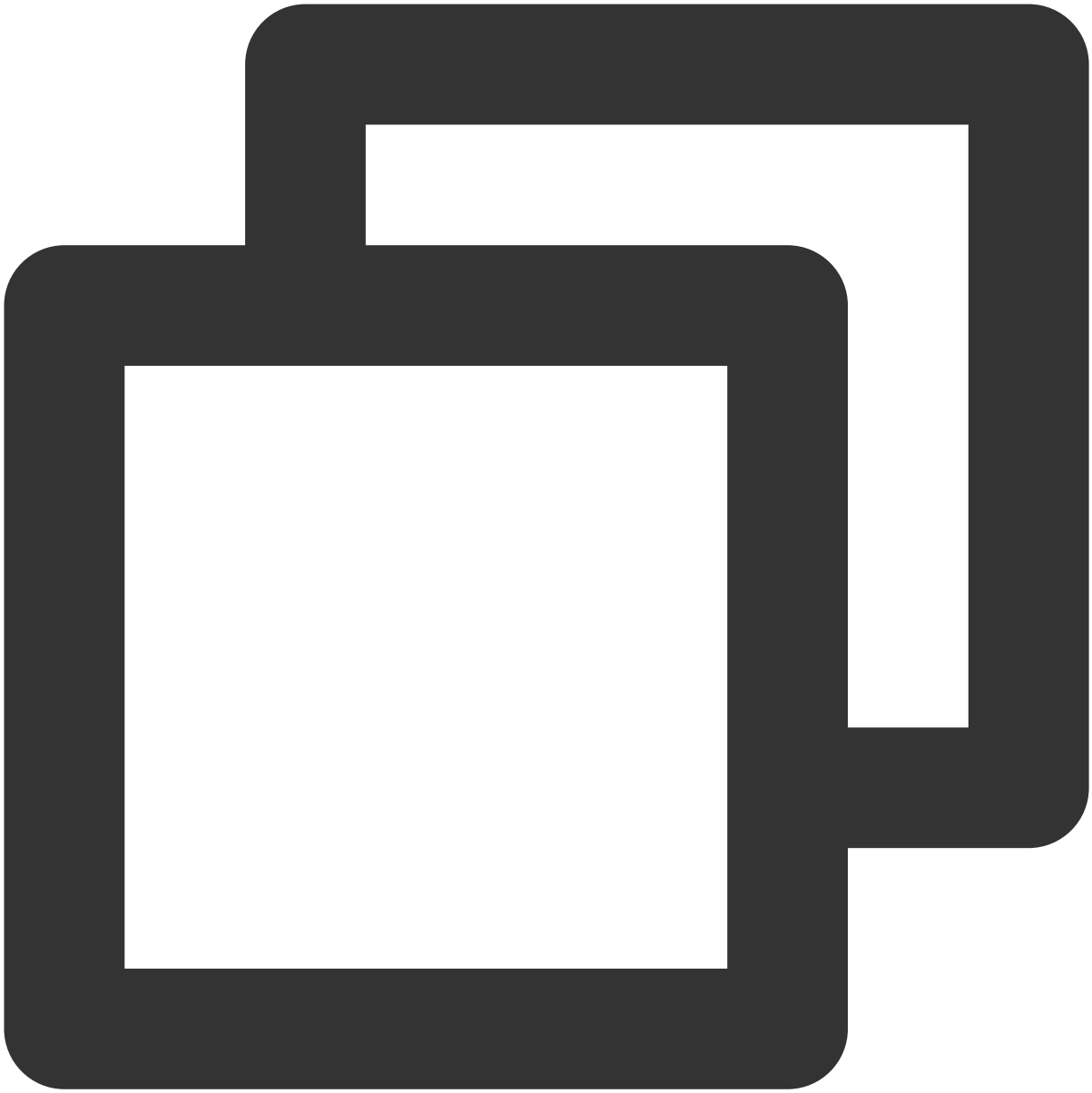
```
ROUND(<expr> integer|double|decimal, <d> integer)
```

Supported engines: SparkSQL and Presto

Usage instructions: Round expr to d decimal places using the half-up rounding mode.

Return type: double

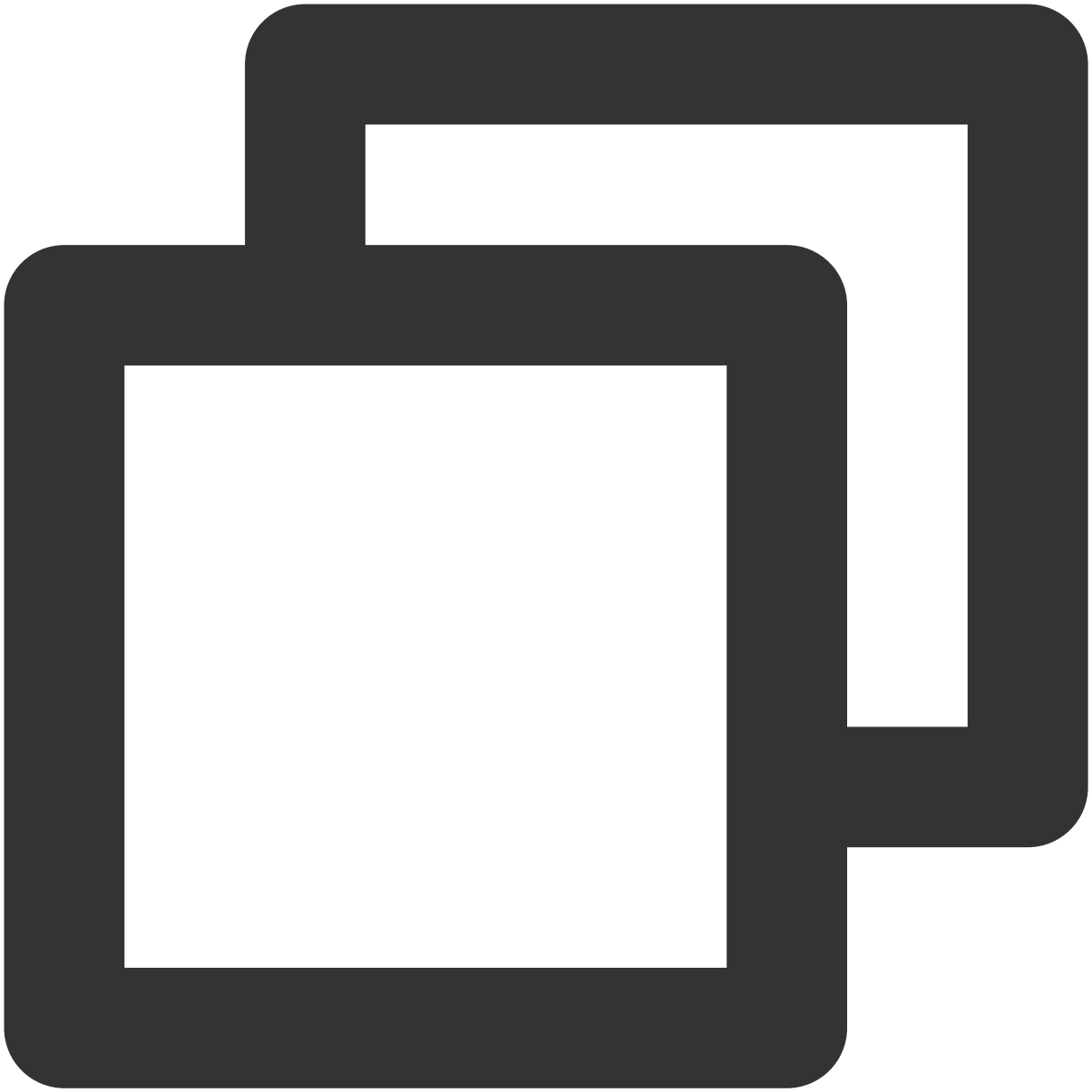
Example:



```
> SELECT round(2.5, 0);
 3
```

SHIFTLEFT

Function statement:



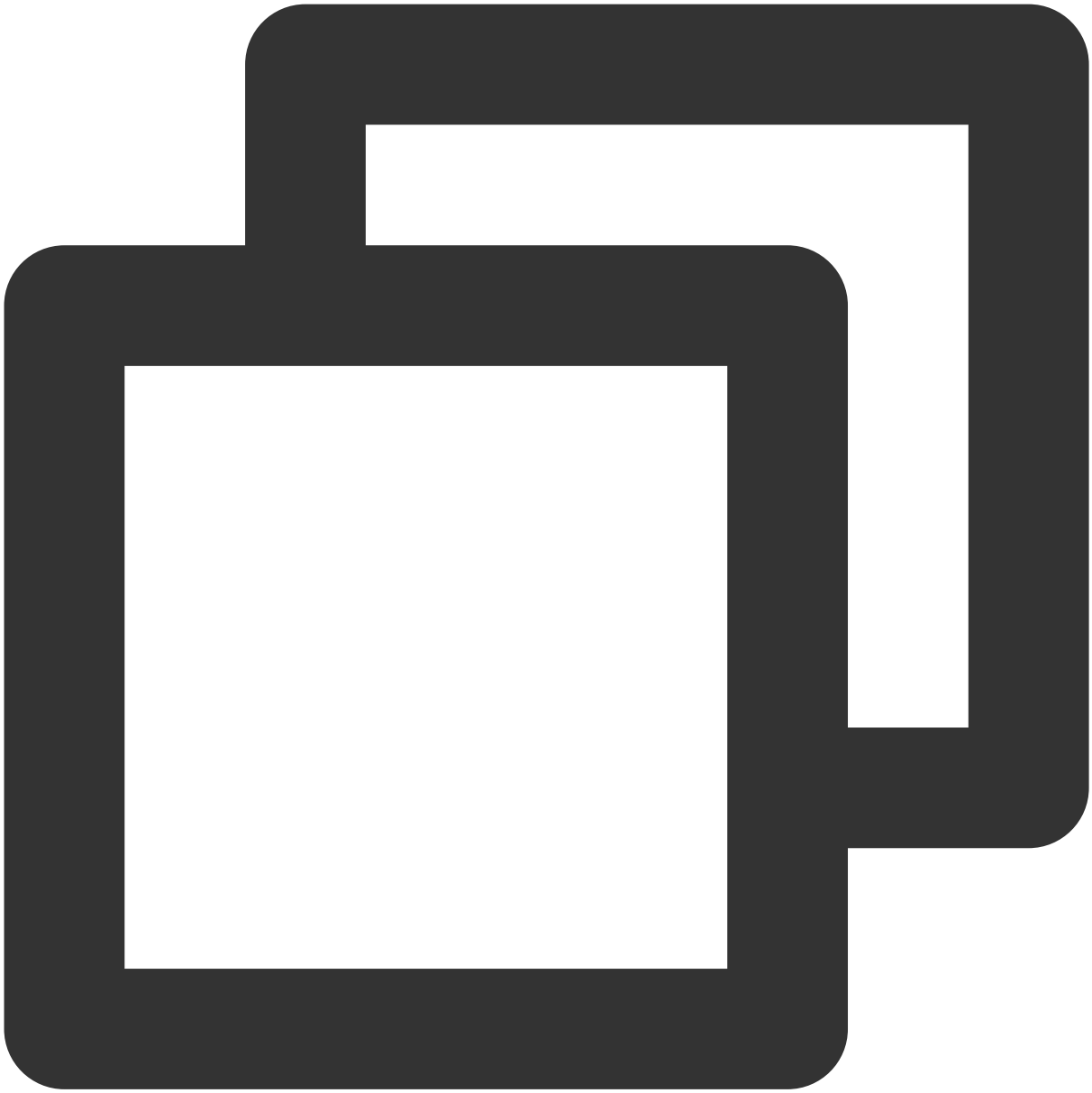
```
SHIFTLEFT(<base> integer|double|decimal, <expr> integer)
```

Supported engines: SparkSQL and Presto

Usage instructions: Perform a bitwise left shift.

Return type: int |bigint

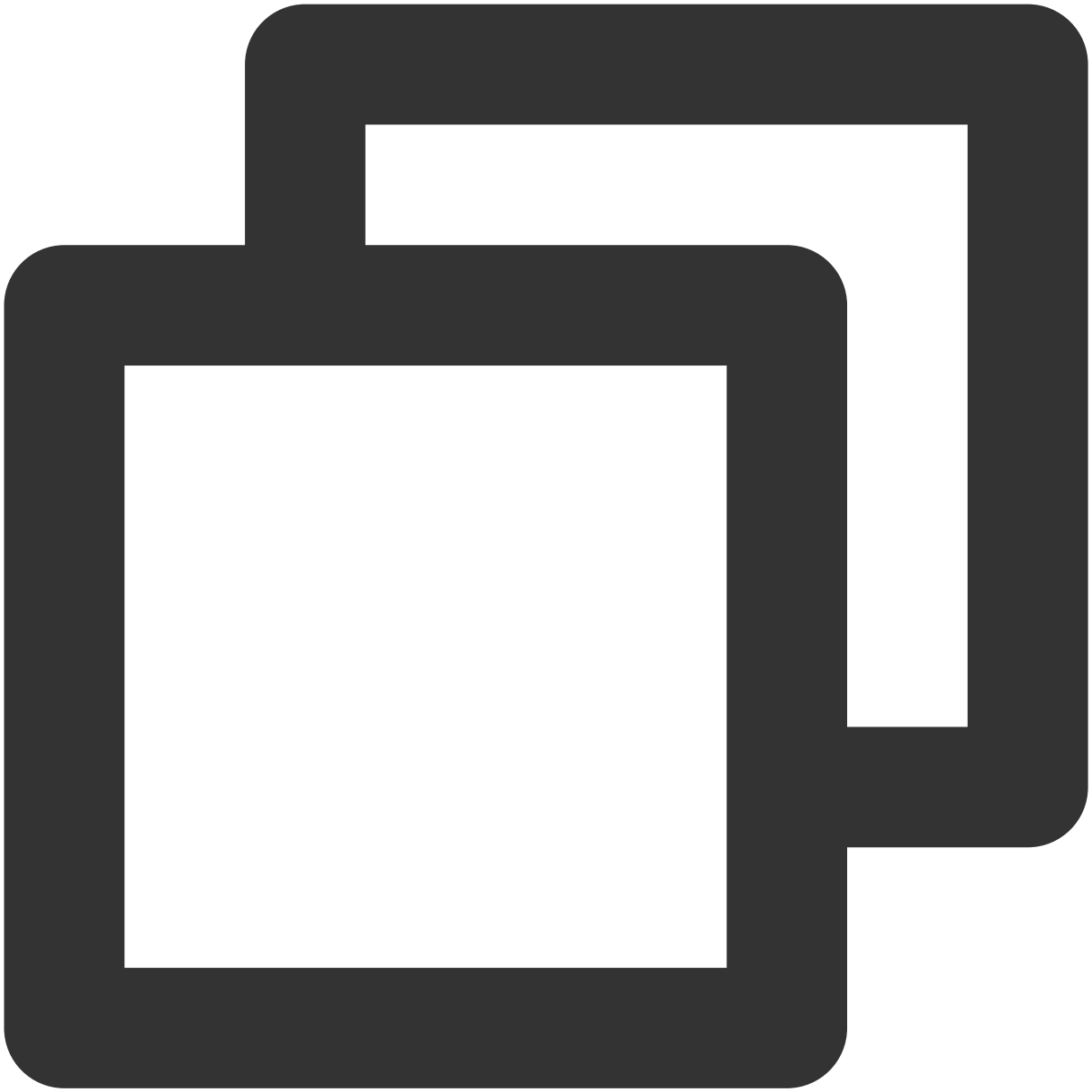
Example:



```
> SELECT shiftleft(2, 1);
 4
```

SHIFTRIGHT

Function statement:



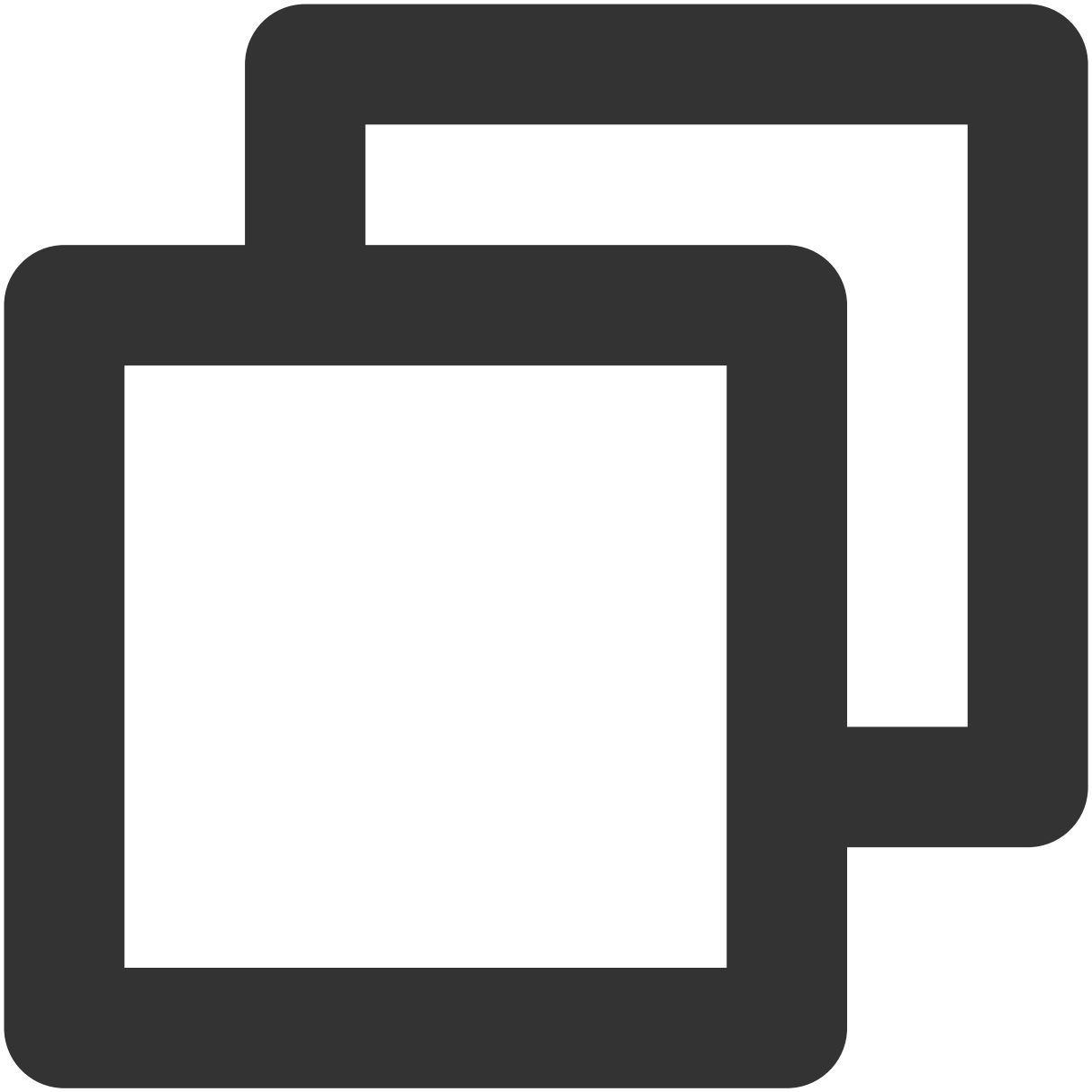
```
SHIFTRIGHT(<base> integer|double|decimal, <expr> integer)
```

Supported engines: SparkSQL and Presto

Usage instructions: Perform a bitwise right shift.

Return type: int |bigint

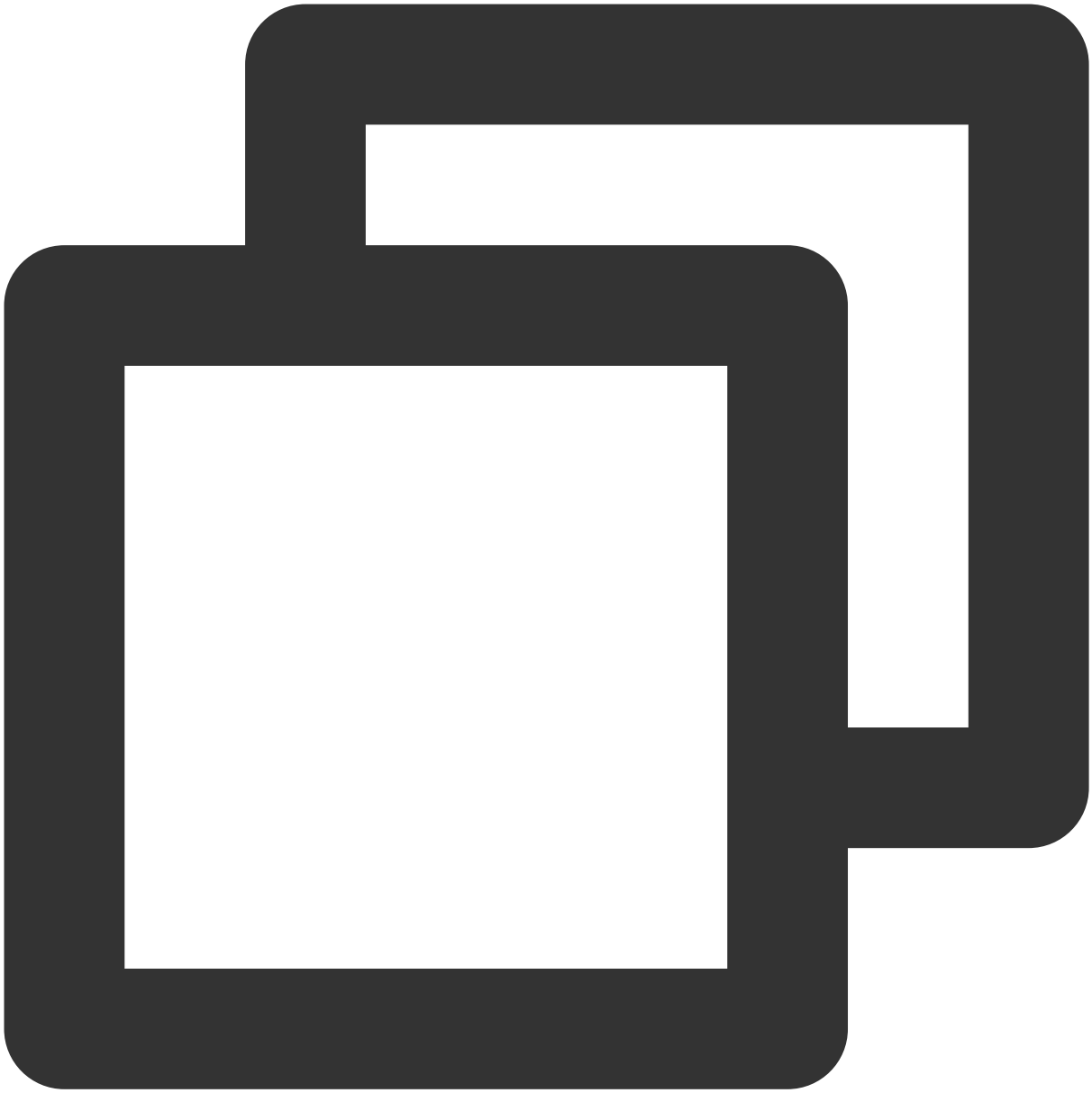
Example:



```
> SELECT shiftright(4, 1);
 2
```

SHIFTRIGHTUNSIGNED

Function statement:



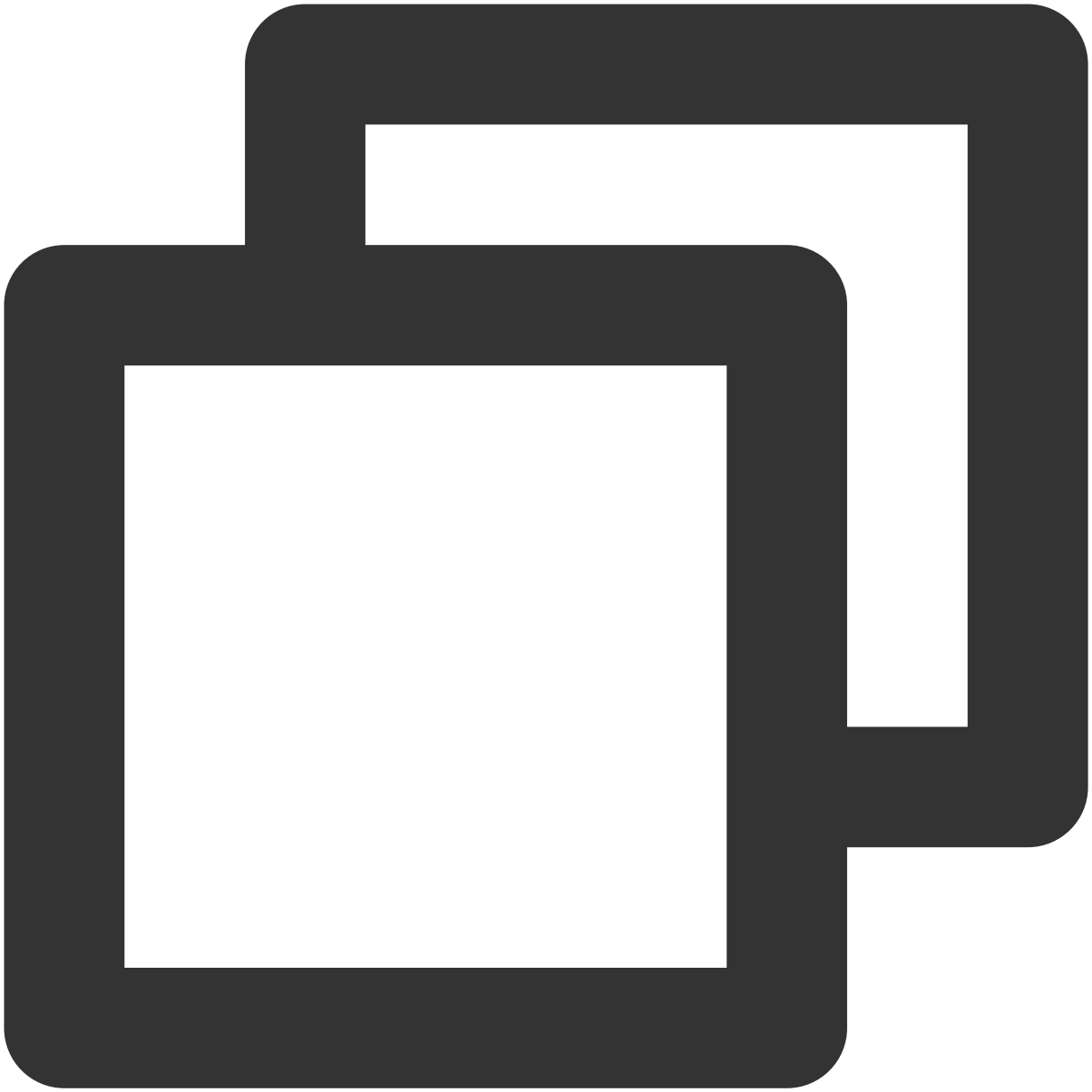
```
SHIFTRIGHTUNSIGNED(<base> integer|double|decimal, <expr> integer)
```

Supported engines: SparkSQL and Presto

Usage instructions: Perform a bitwise unsigned right shift.

Return type: int |bigint

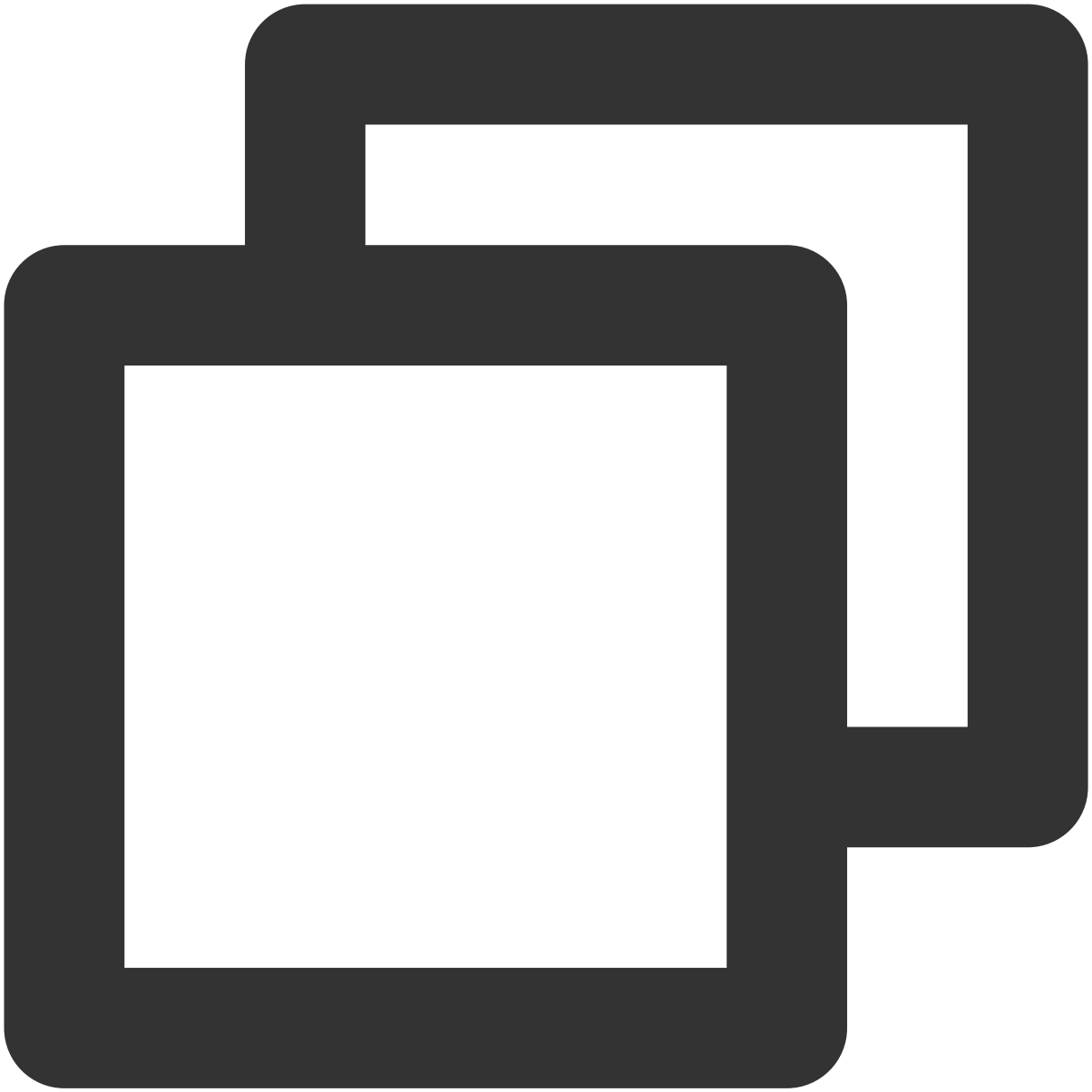
Example:



```
> SELECT shiftrightunsigned(4, 1);
 2
```

SIGN

Function statement:



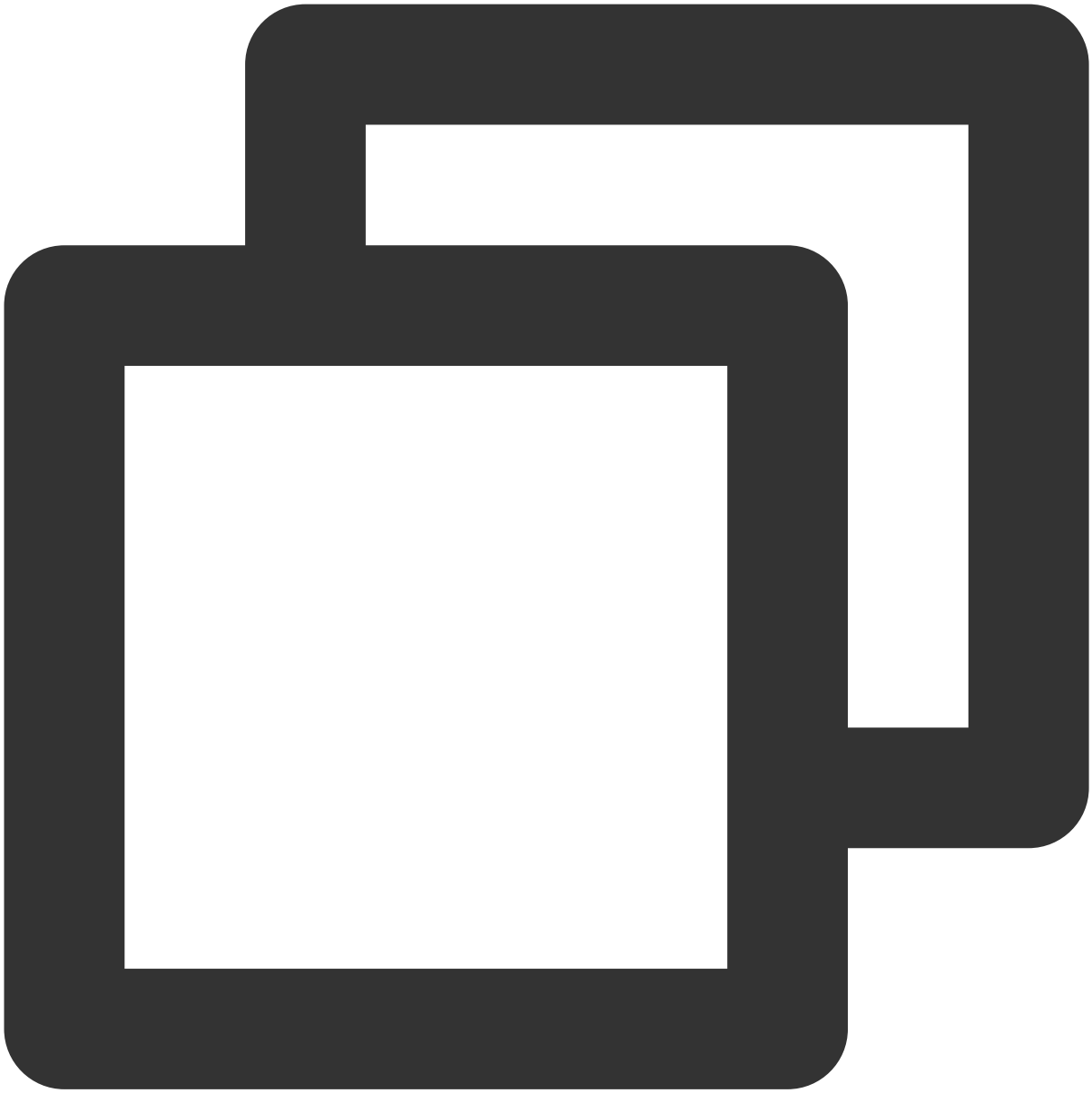
```
SIGN(<expr> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return -1.0, 0.0, or 1.0 respectively when expr is negative, 0, or positive.

Return type: double

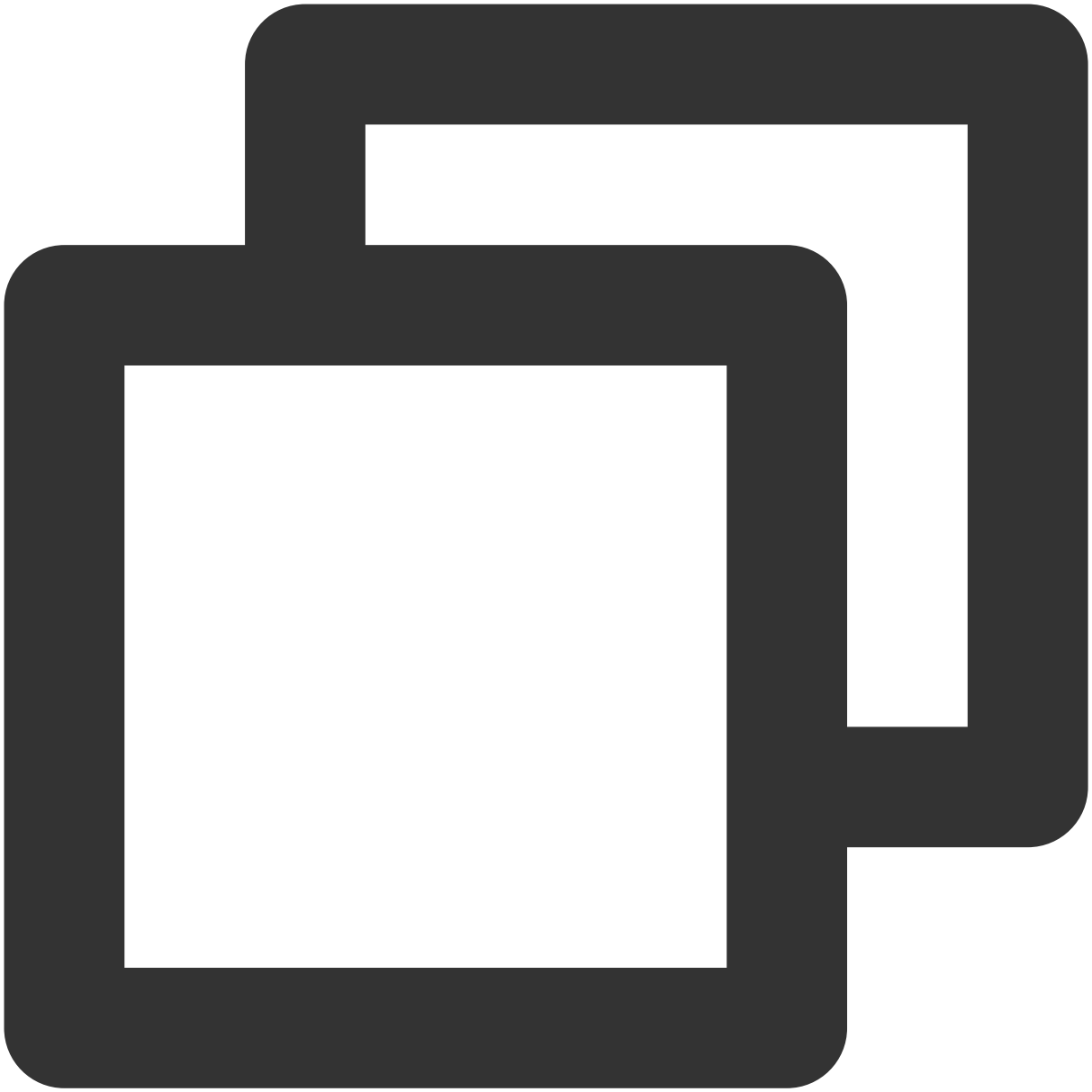
Example:



```
> SELECT sign(40);
 1.0
```

SIGNUM

Function statement:



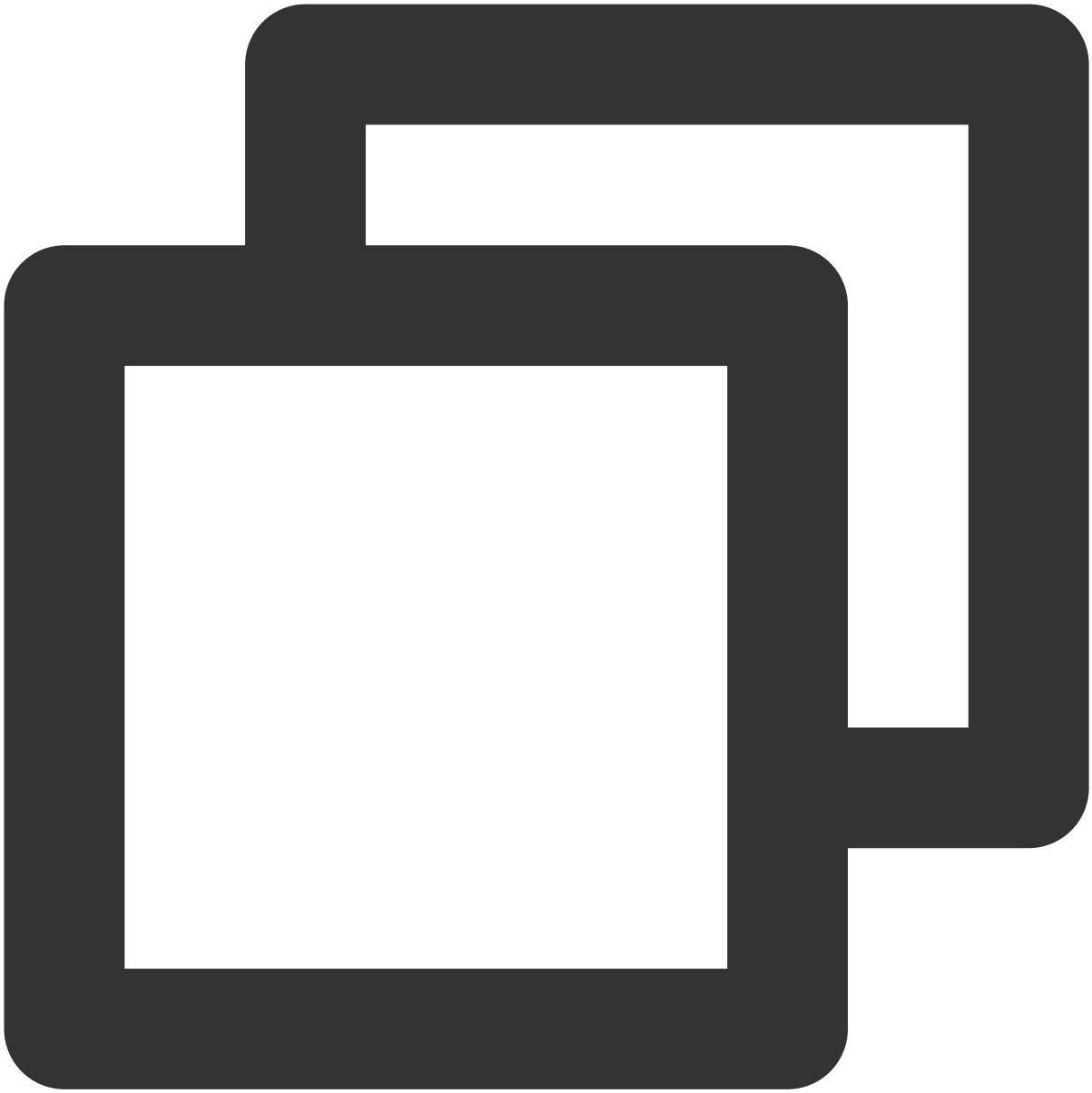
```
SIGNUM(<expr> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return -1.0, 0.0, or 1.0 respectively if expr is negative, 0, or positive.

Return type: double

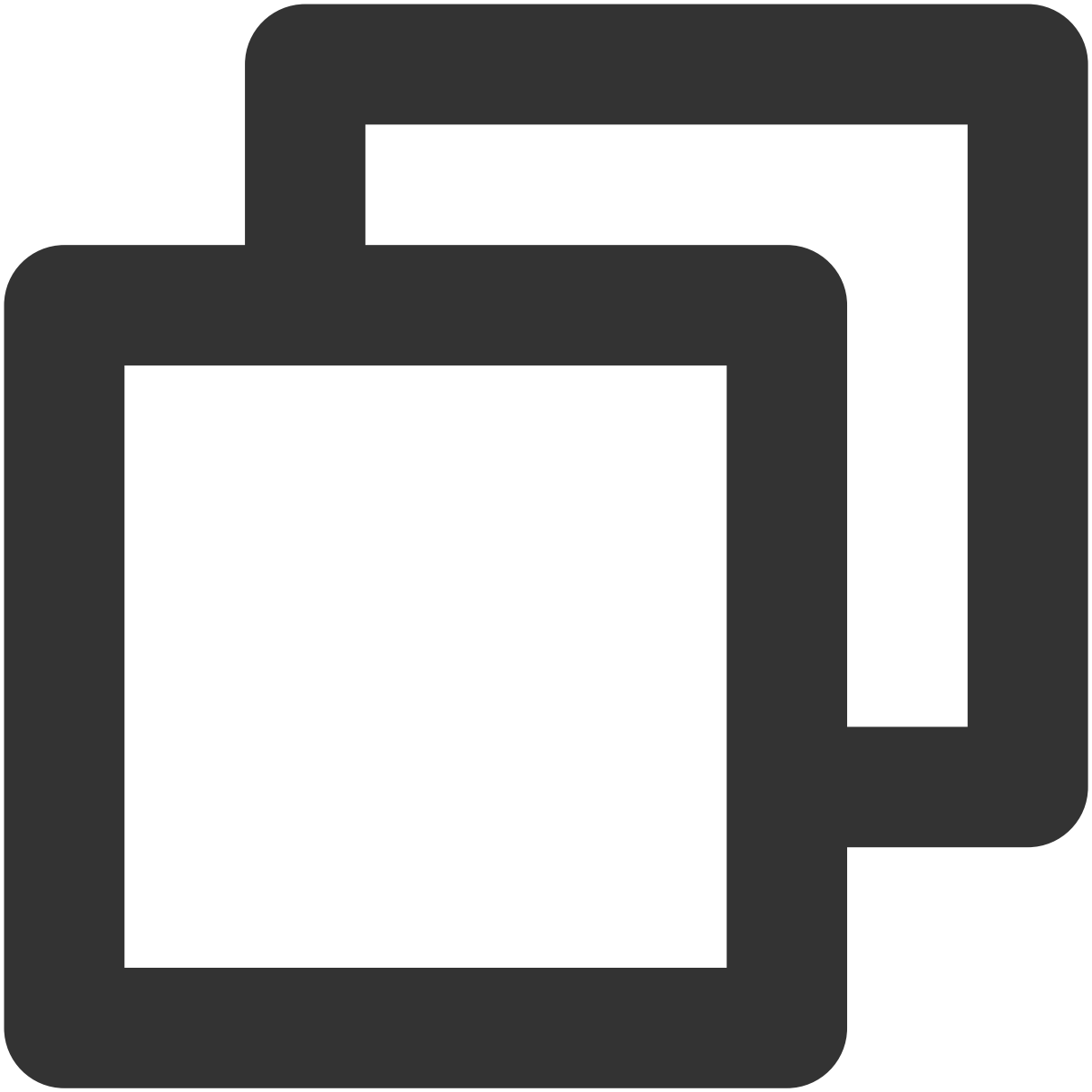
Example:



```
> SELECT signum(40);
 1.0
```

SIN

Function statement:



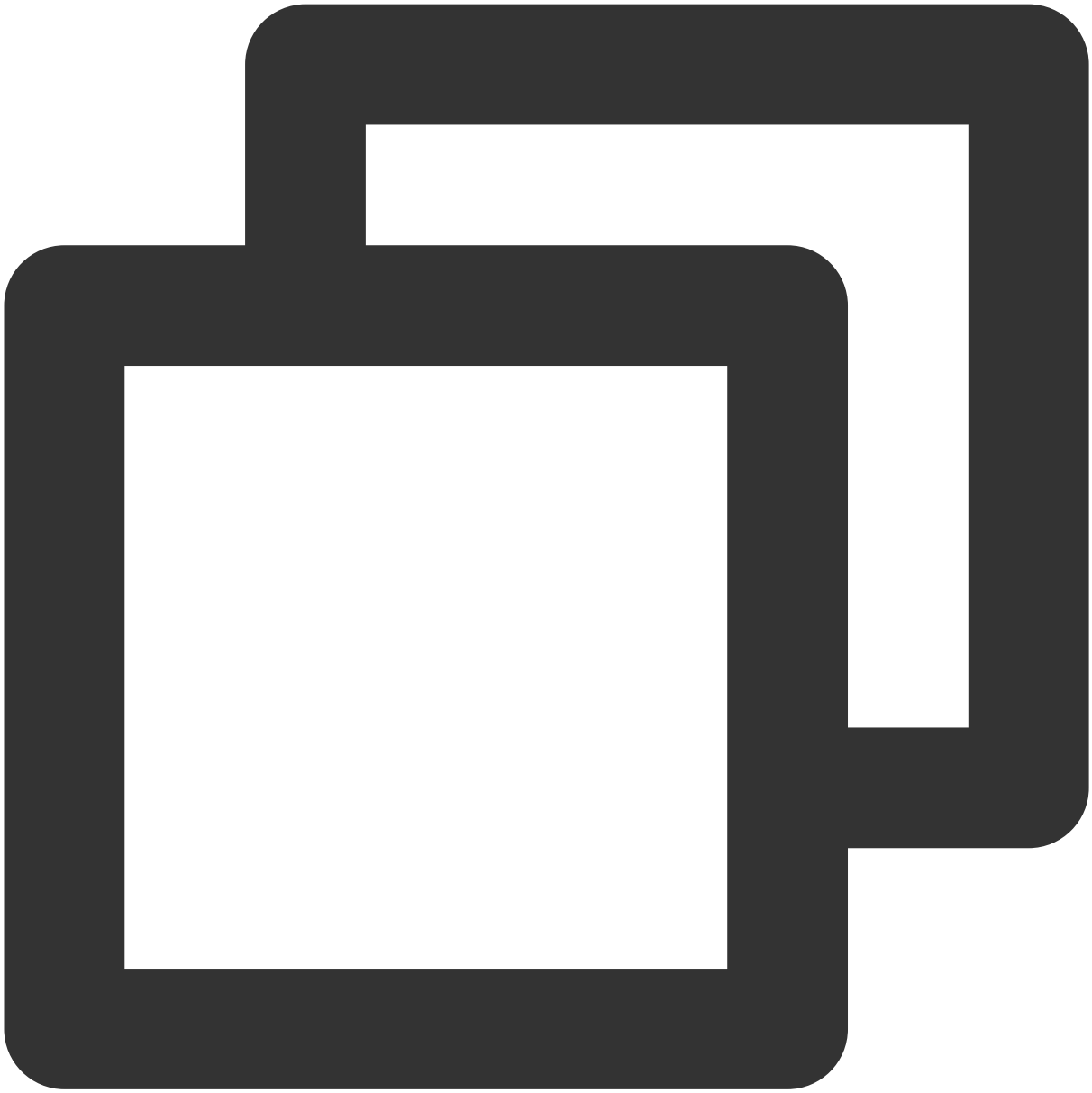
```
SIN(<expr> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the sine of expr.

Return type: double

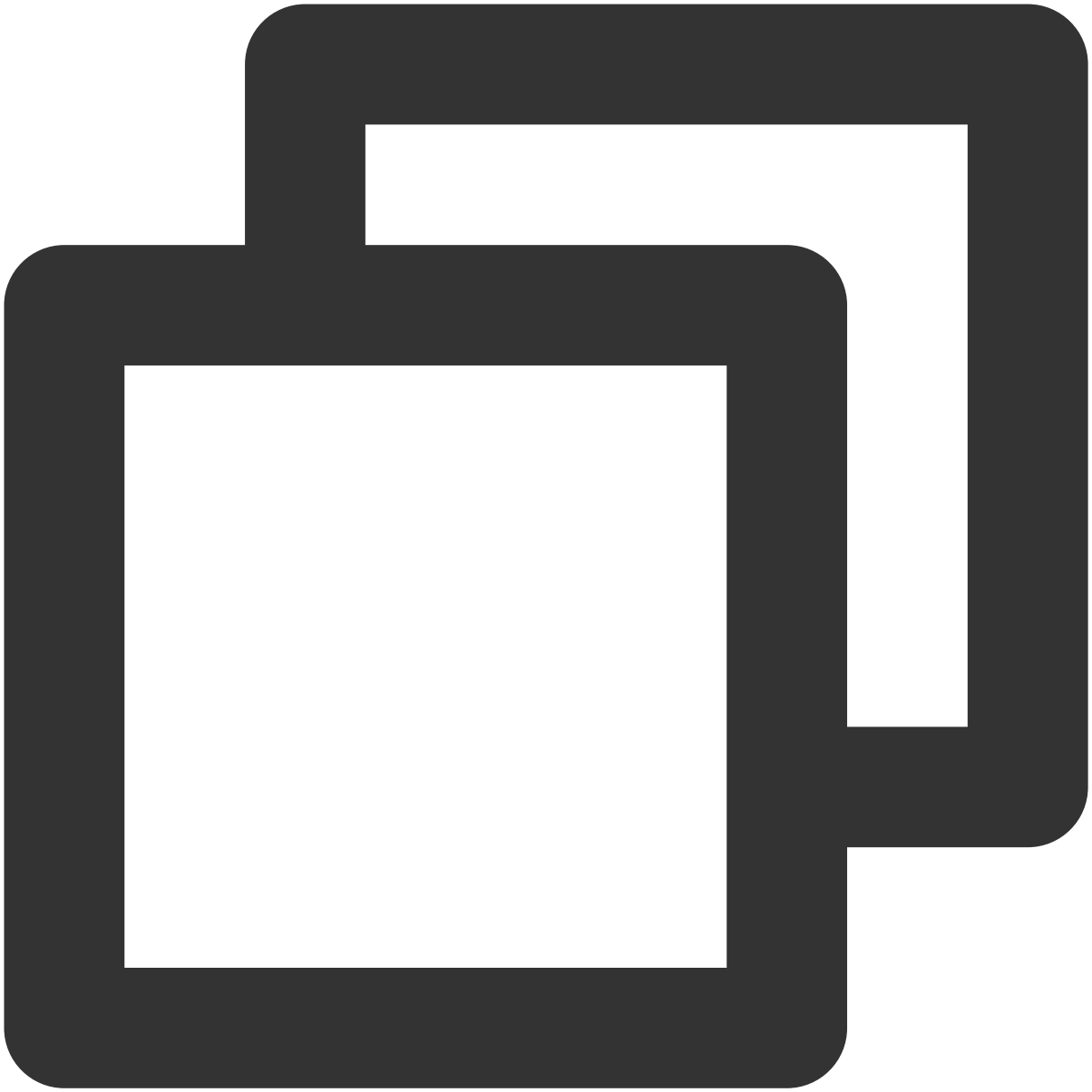
Example:



```
> SELECT sin(0);
 0.0
```

SINH

Function statement:



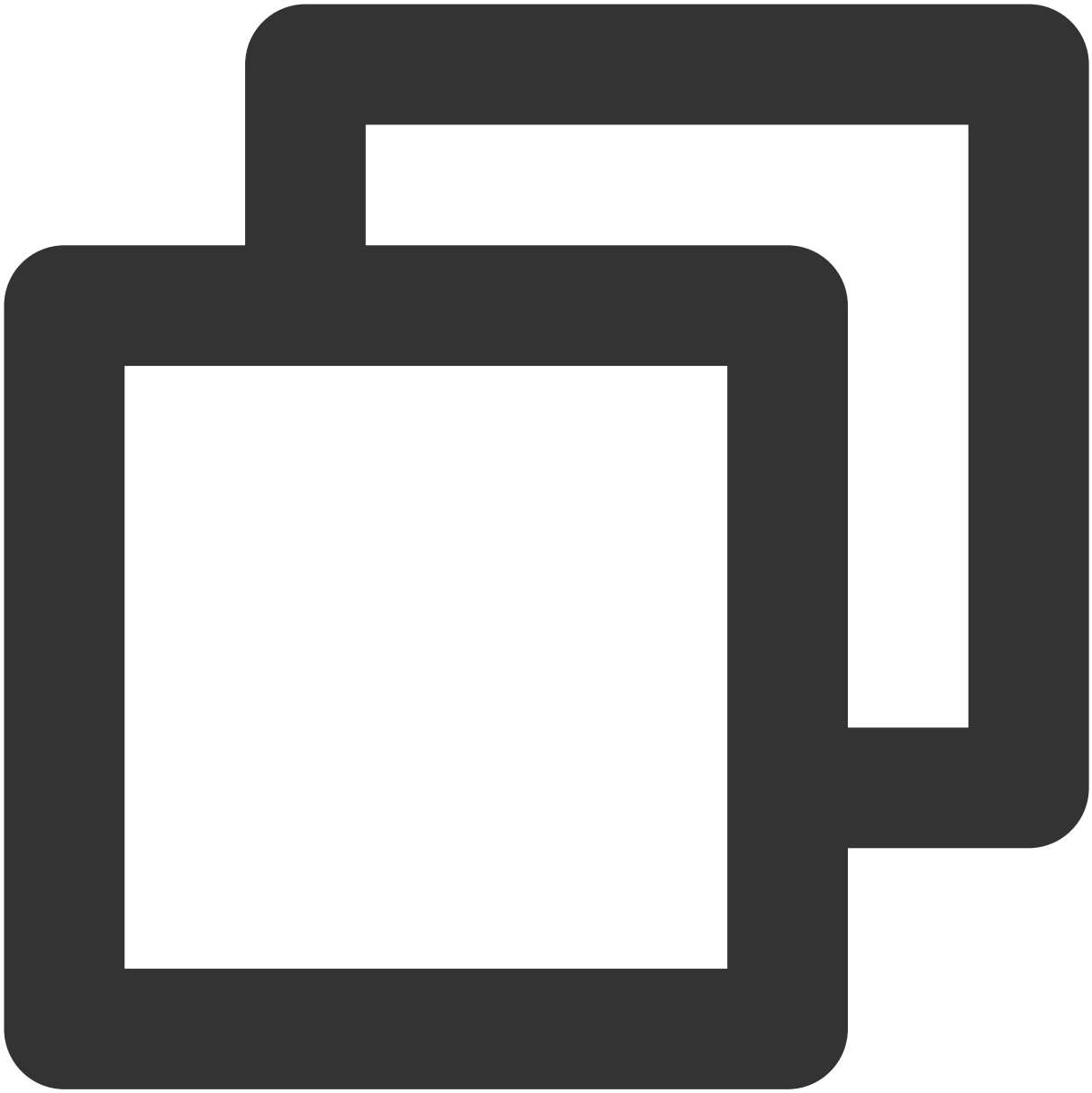
```
SINH(<expr> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the hyperbolic sine of expr.

Return type: double

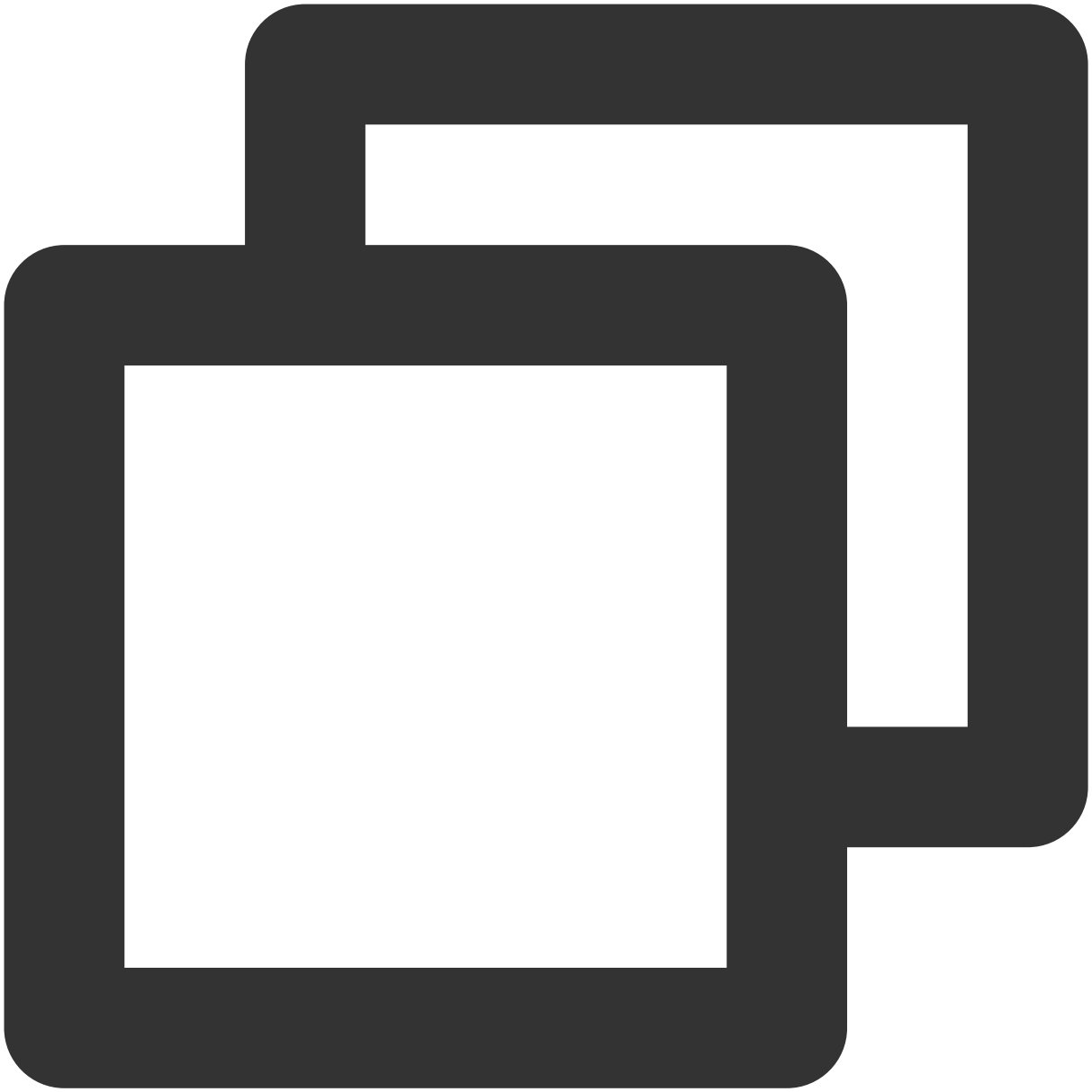
Example:



```
> SELECT sinh(0);
 0.0
```

SQRT

Function statement:



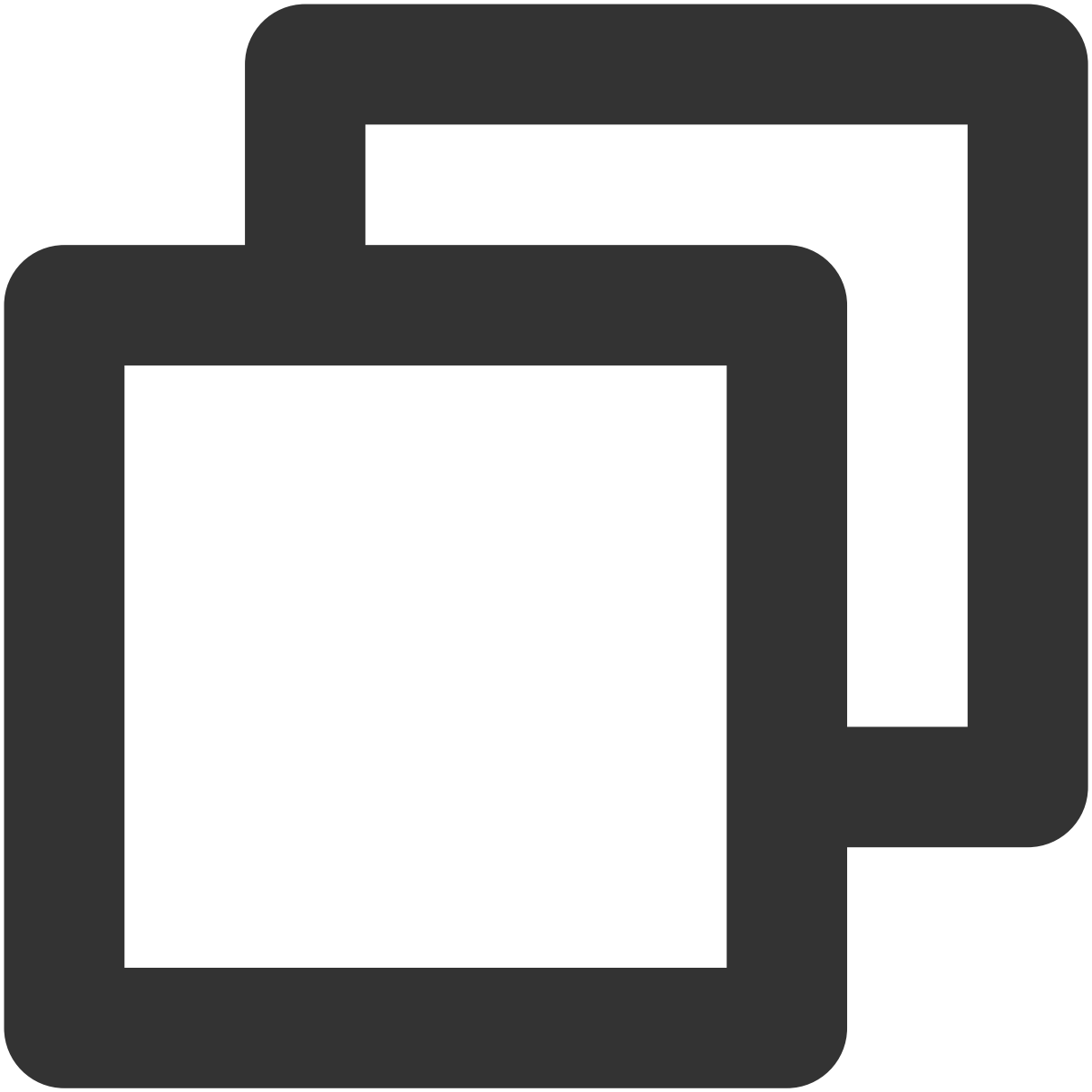
```
SQRT(<expr> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the square root of expr.

Return type: double

Example:



```
> SELECT sqrt(4);
 2.0
```

TAN

Function statement:

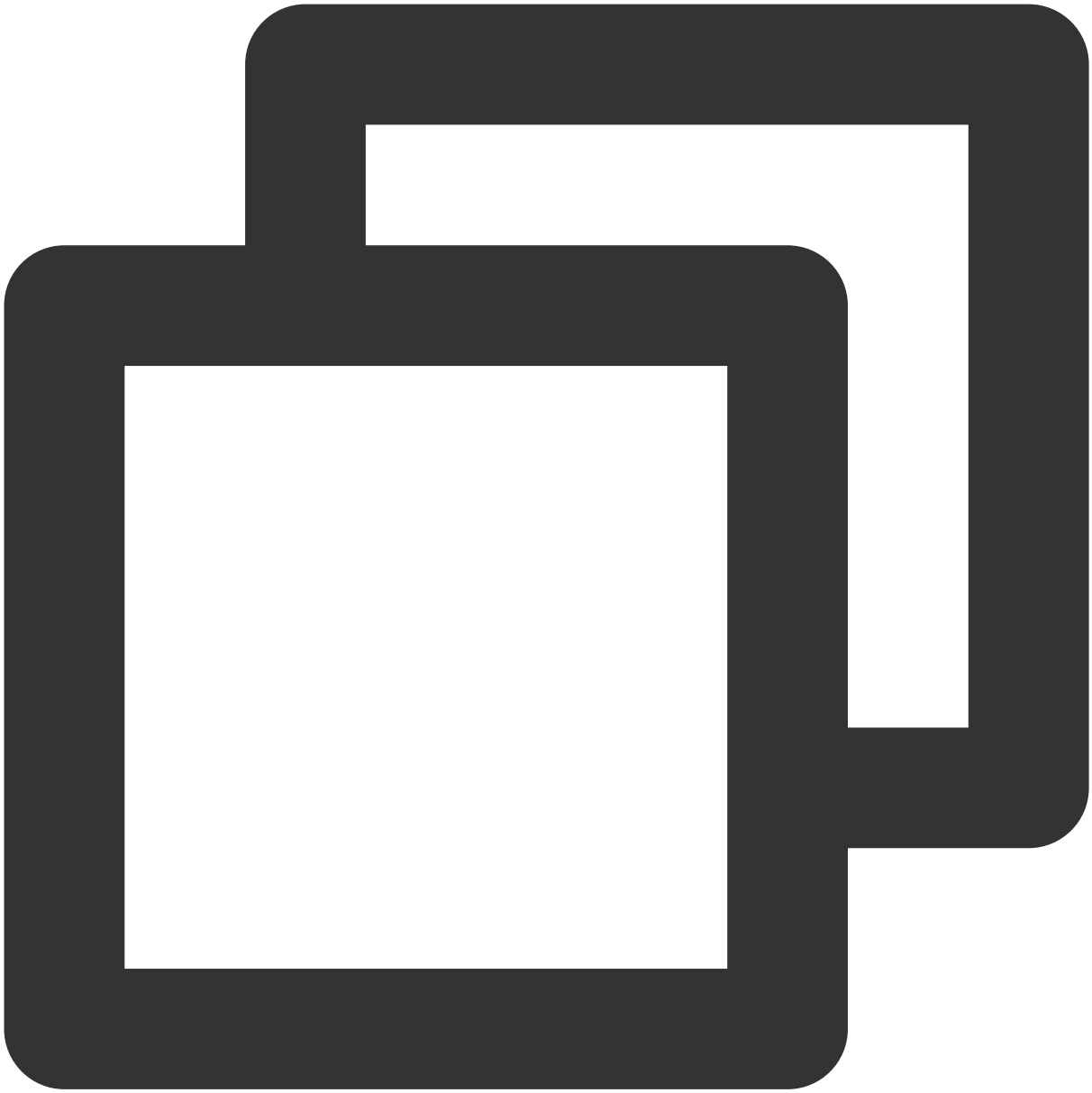
```
TAN(<expr> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the tangent of expr.

Return type: double

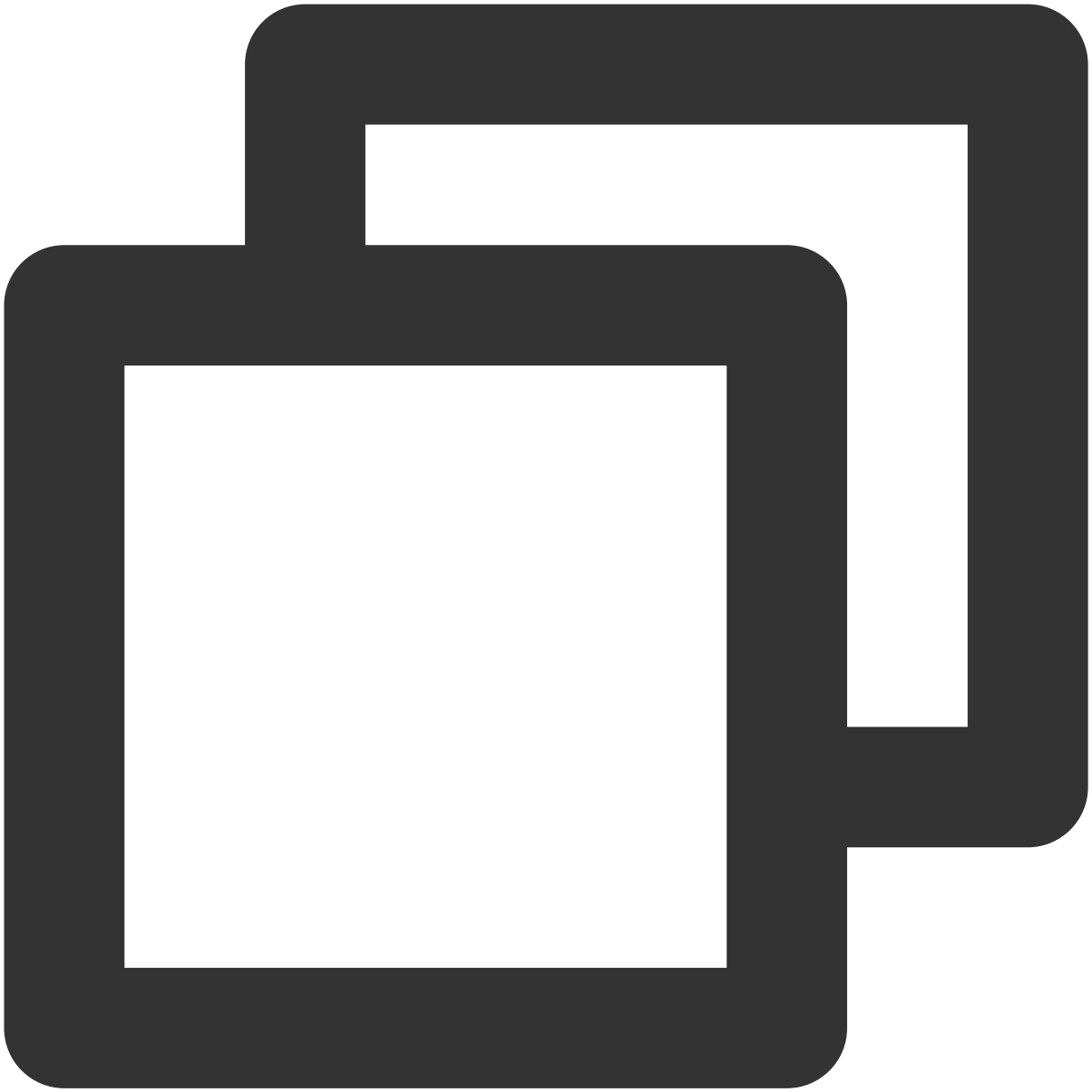
Example:



```
> SELECT tan(0);
 0.0
```

COT

Function statement:



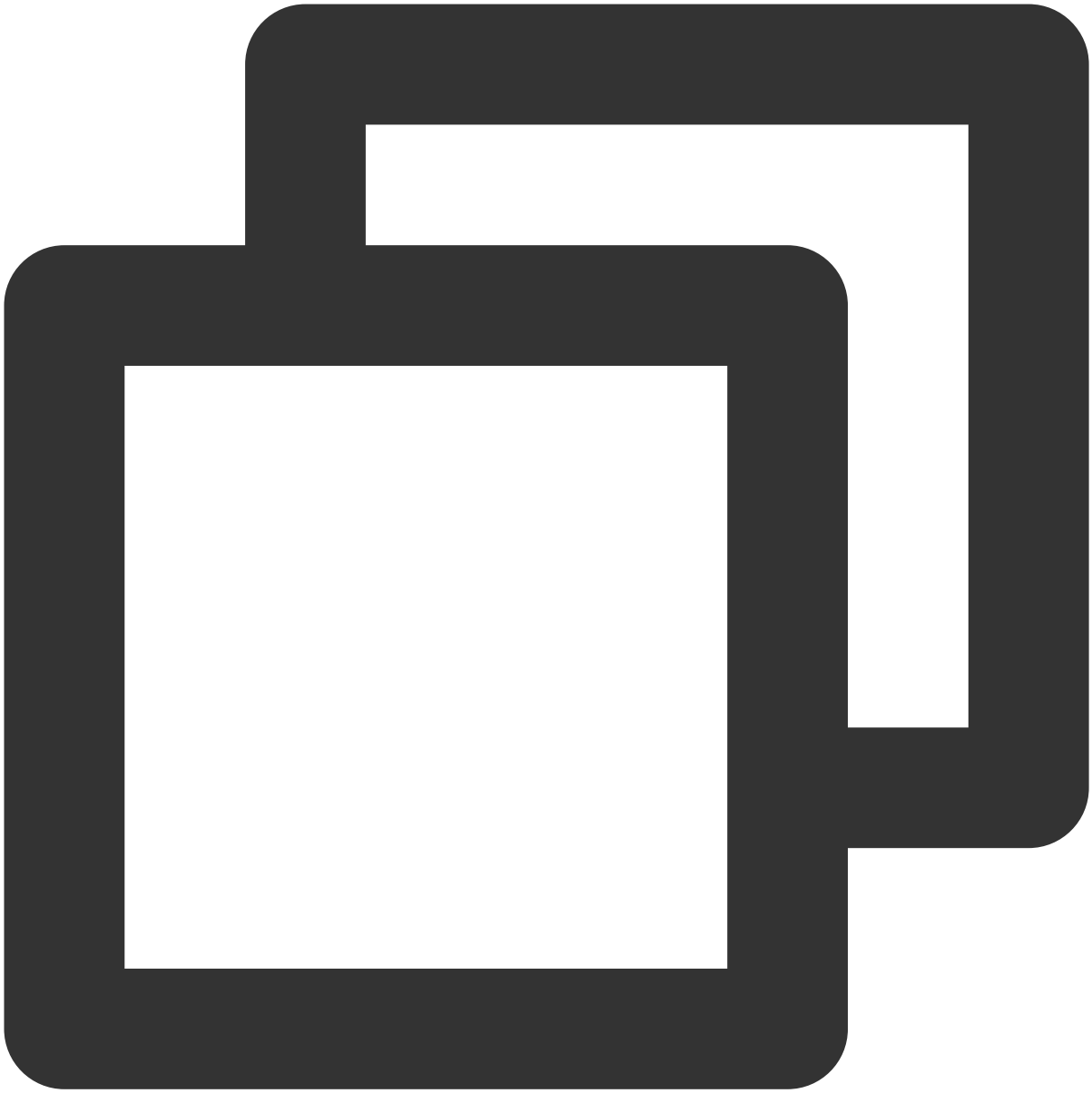
```
COT(<expr> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the cotangent of the expression.

Return type: double

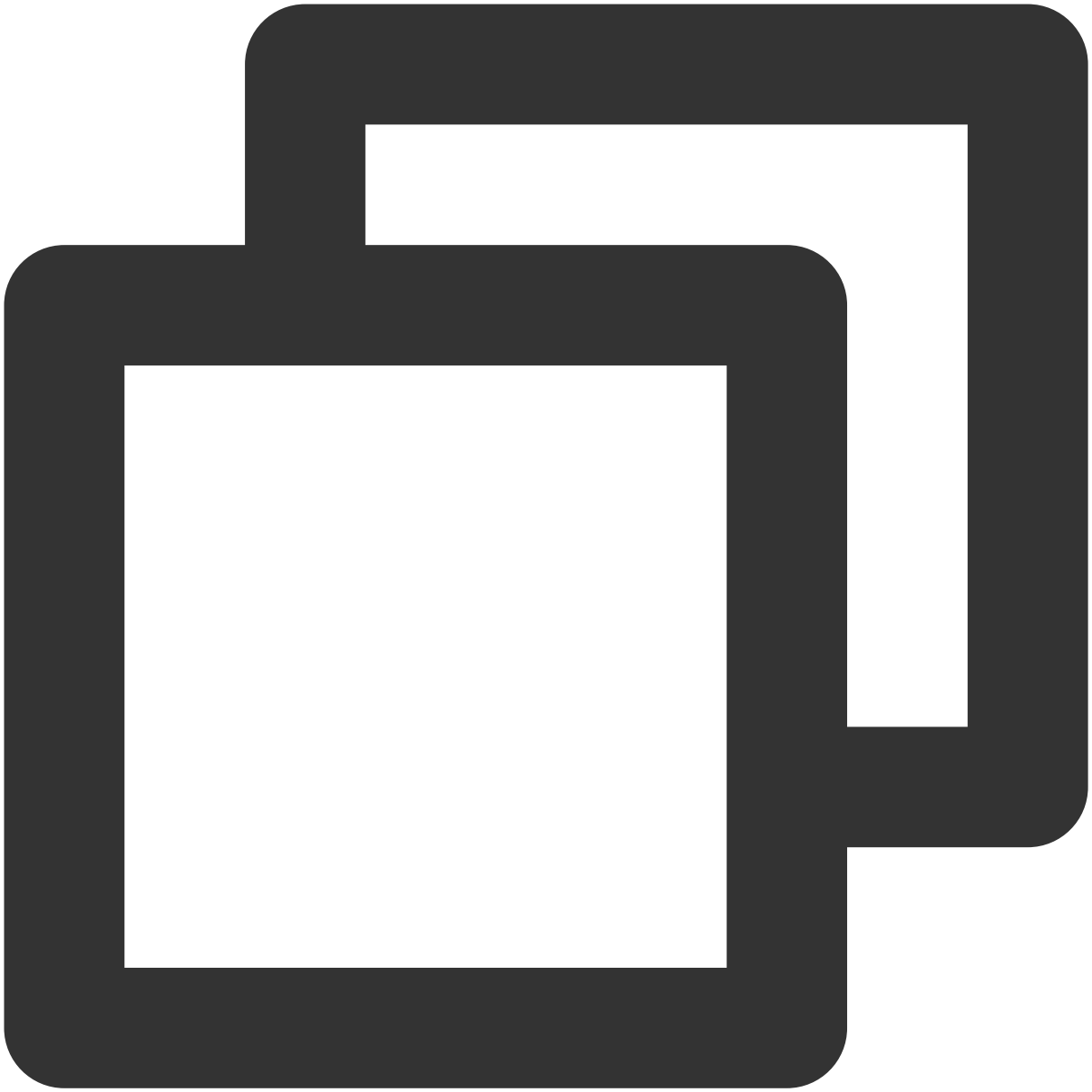
Example:



```
> SELECT cot(1);
 0.6420926159343306
```

TANH

Function statement:



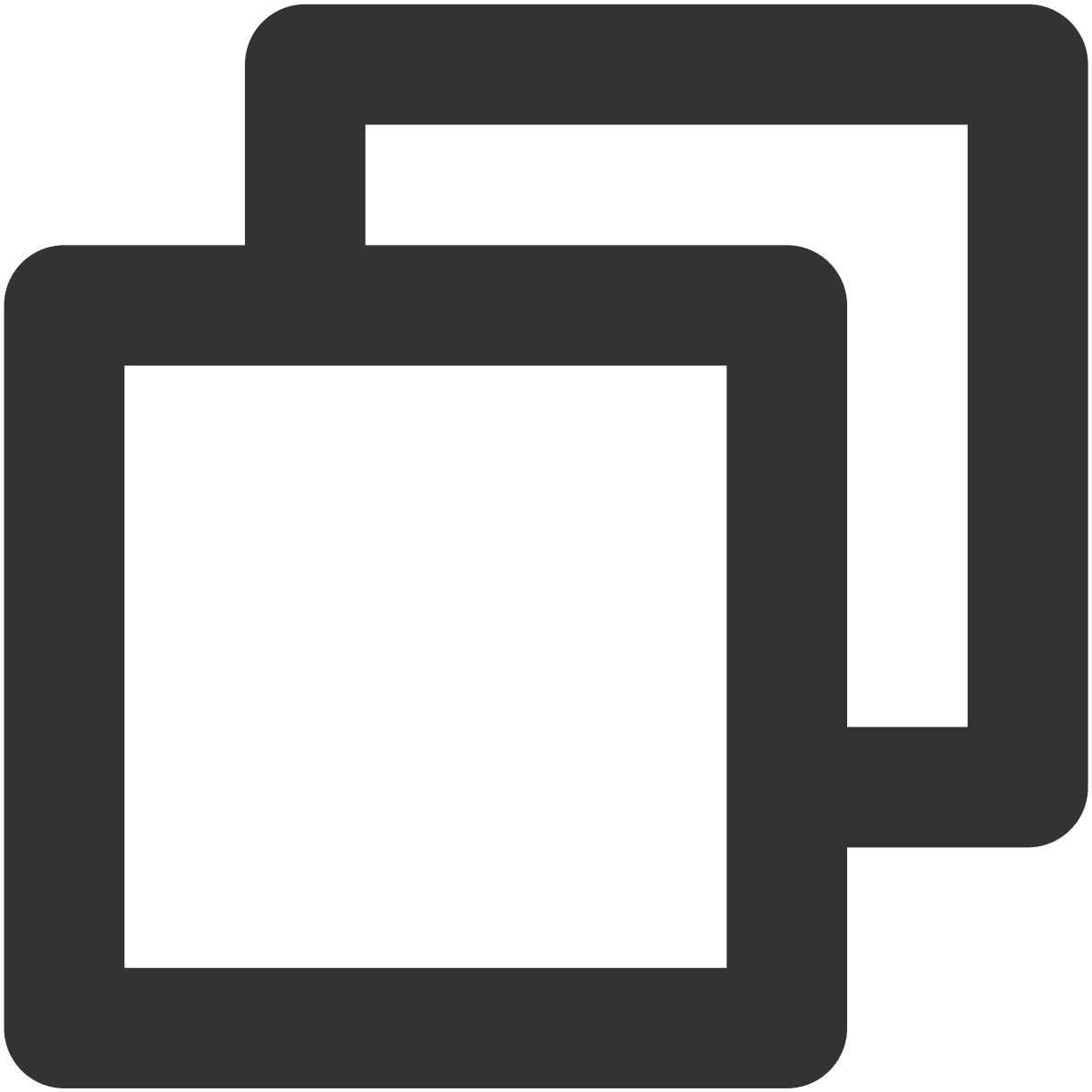
```
TANH(<expr> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the hyperbolic tangent of expr.

Return type: double

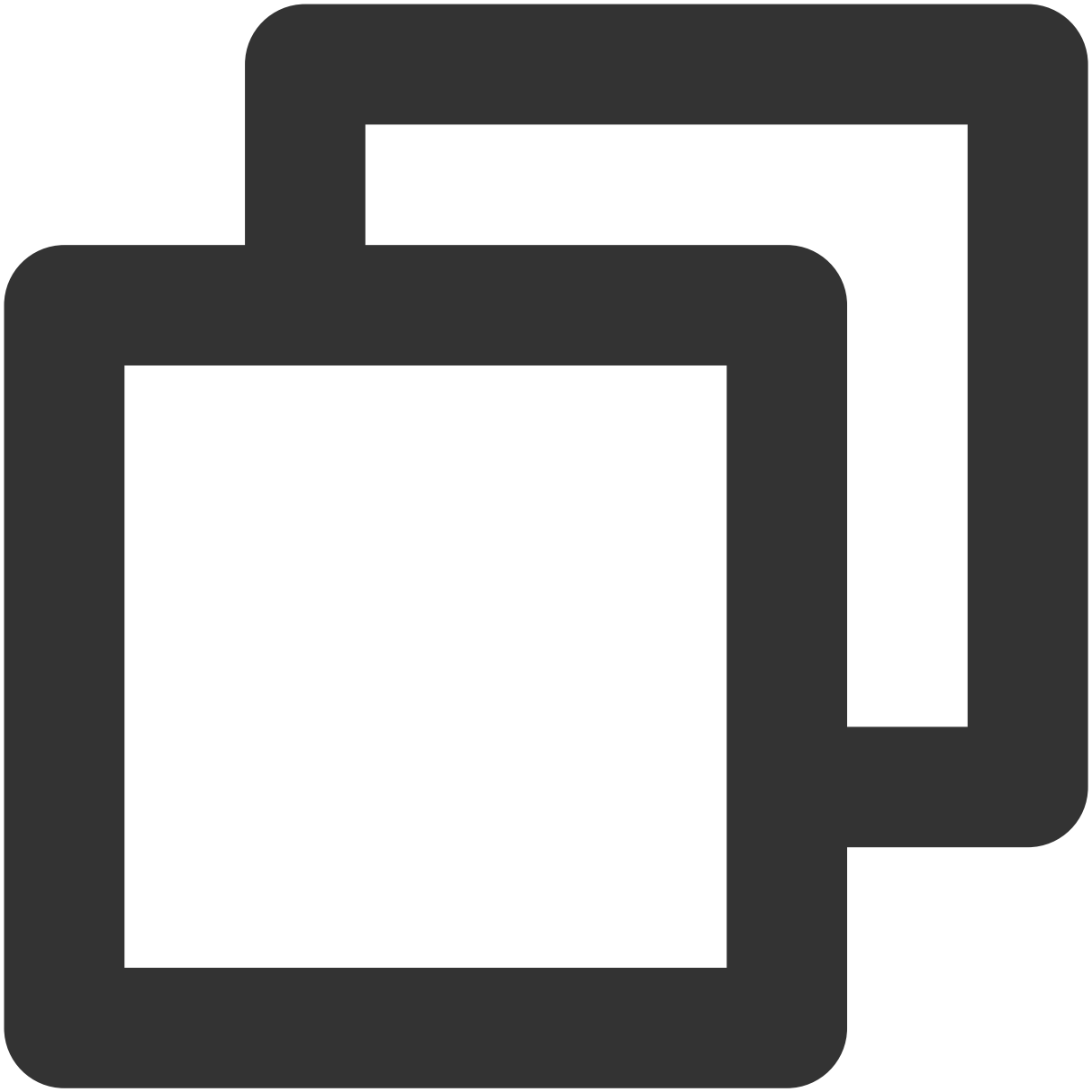
Example:



```
> SELECT tanh(0);
 0.0
```

WIDTH_BUCKET

Function statement:



```
WIDTH_BUCKET(<value> integer|double|decimal, <min> integer|double|decimal, <max> in
```

Supported engines: SparkSQL and Presto

Usage instructions: Divide the range from min to max into num_bucket equal groups, and return the group number into which value falls.

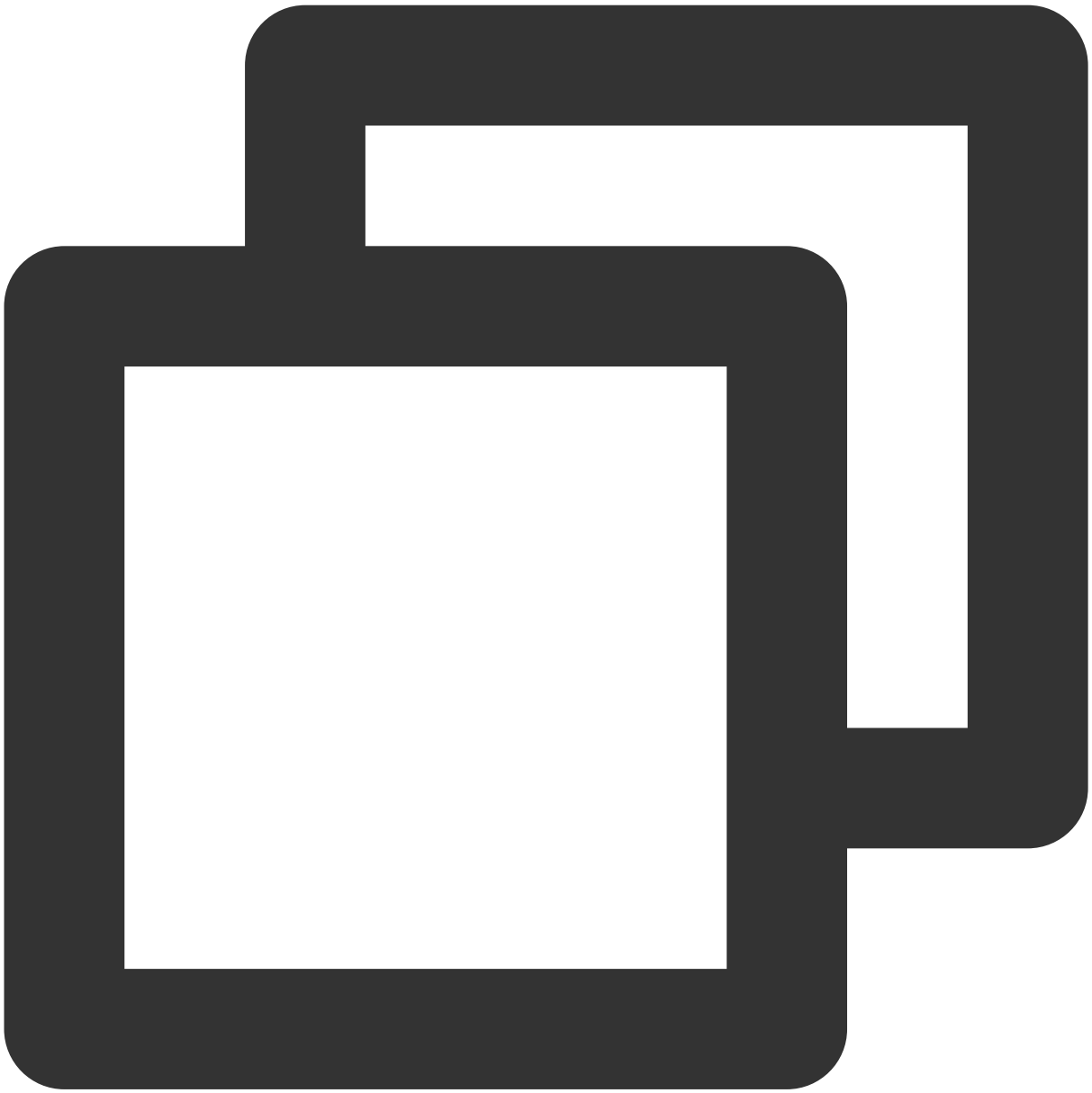
Return type: integer

Example:

```
> SELECT width_bucket(5.3, 0.2, 10.6, 5);
 3
> SELECT width_bucket(-2.1, 1.3, 3.4, 3);
 0
> SELECT width_bucket(8.1, 0.0, 5.7, 4);
 5
> SELECT width_bucket(-0.9, 5.2, 0.5, 2);
 3
```

TRY_ADD

Function statement:



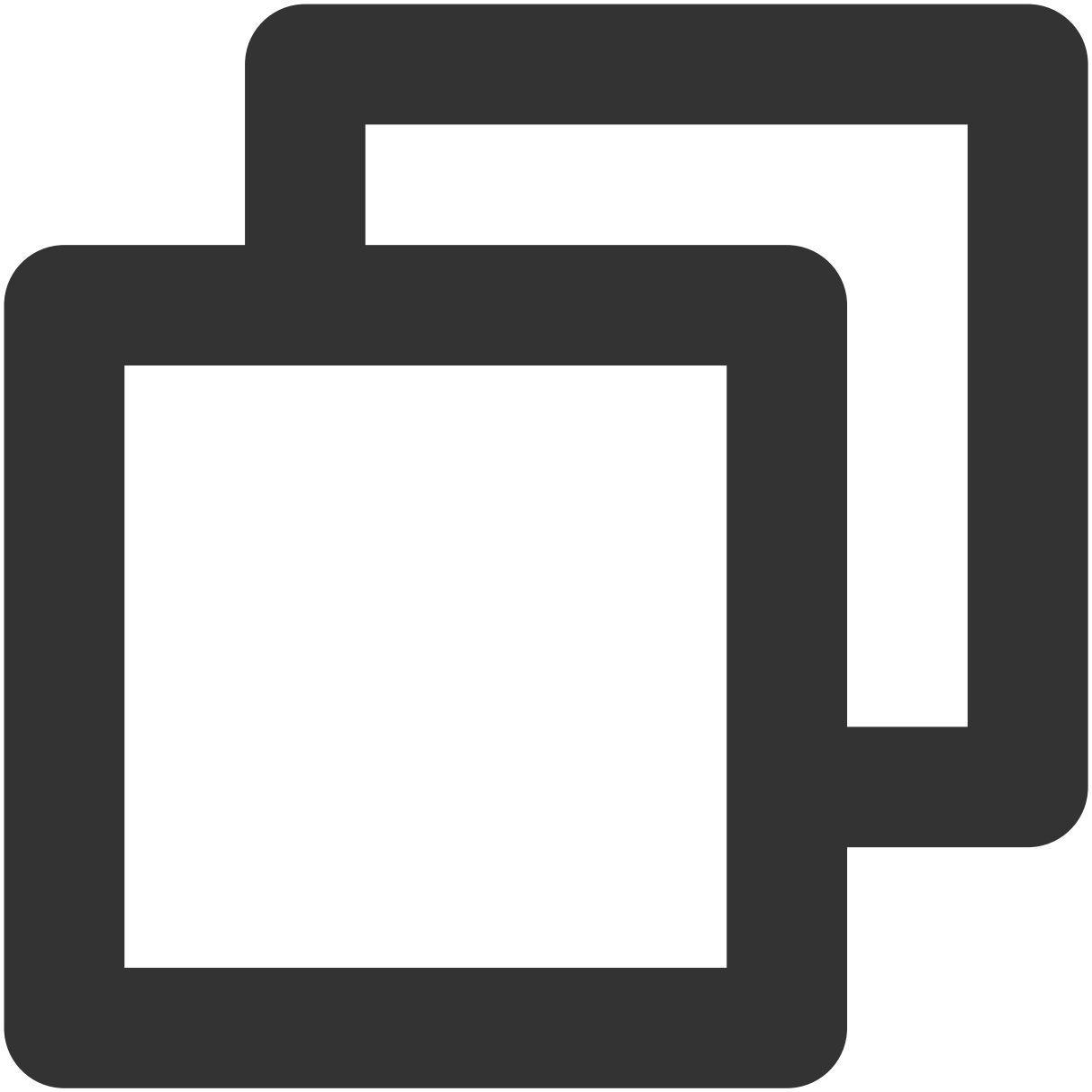
```
TRY_ADD(<expr1> integer|double|decimal|date|timestamp, <expr2> integer|double|decim
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the sum of expr1 and expr2. If there is an overflow, null is returned.

Return type: integer|double|decimal|date|timestamp

Example:



```
> SELECT try_add(1, 2);
 3
> SELECT try_add(2147483647, 1);
 NULL
> SELECT try_add(date'2021-01-01', 1);
 2021-01-02
```

TRY_DIVIDE

Function statement:

```
TRY_DIVIDE(<dividend> integer|double|decimal, <divisor> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return dividend divided by divisor.

Return type: double

Example:

```
> SELECT try_divide(3, 2);
 1.5
> SELECT try_divide(2L, 2L);
 1.0
> SELECT try_divide(1, 0);
 NULL
```

RAND

Function statement:

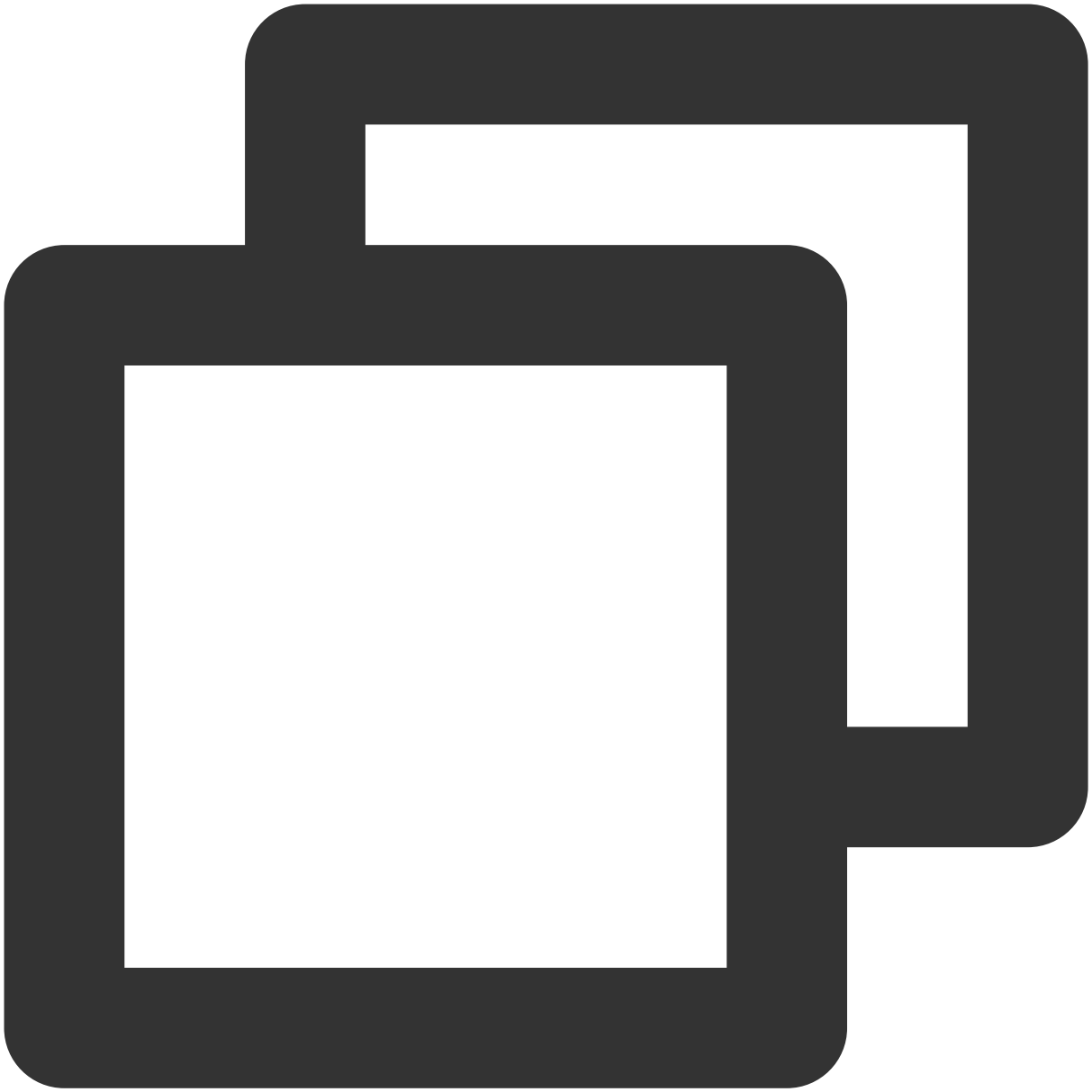
```
RAND([<seed> integer])
```

Supported engines: SparkSQL and Presto

Usage instructions: Return a random value that is independently and uniformly distributed within the range [0, 1].

Return type: double

Example:



```
> SELECT rand();
 0.9629742951434543
> SELECT rand(0);
 0.7604953758285915
> SELECT rand(null);
 0.7604953758285915
```

RANDOM

Function statement:

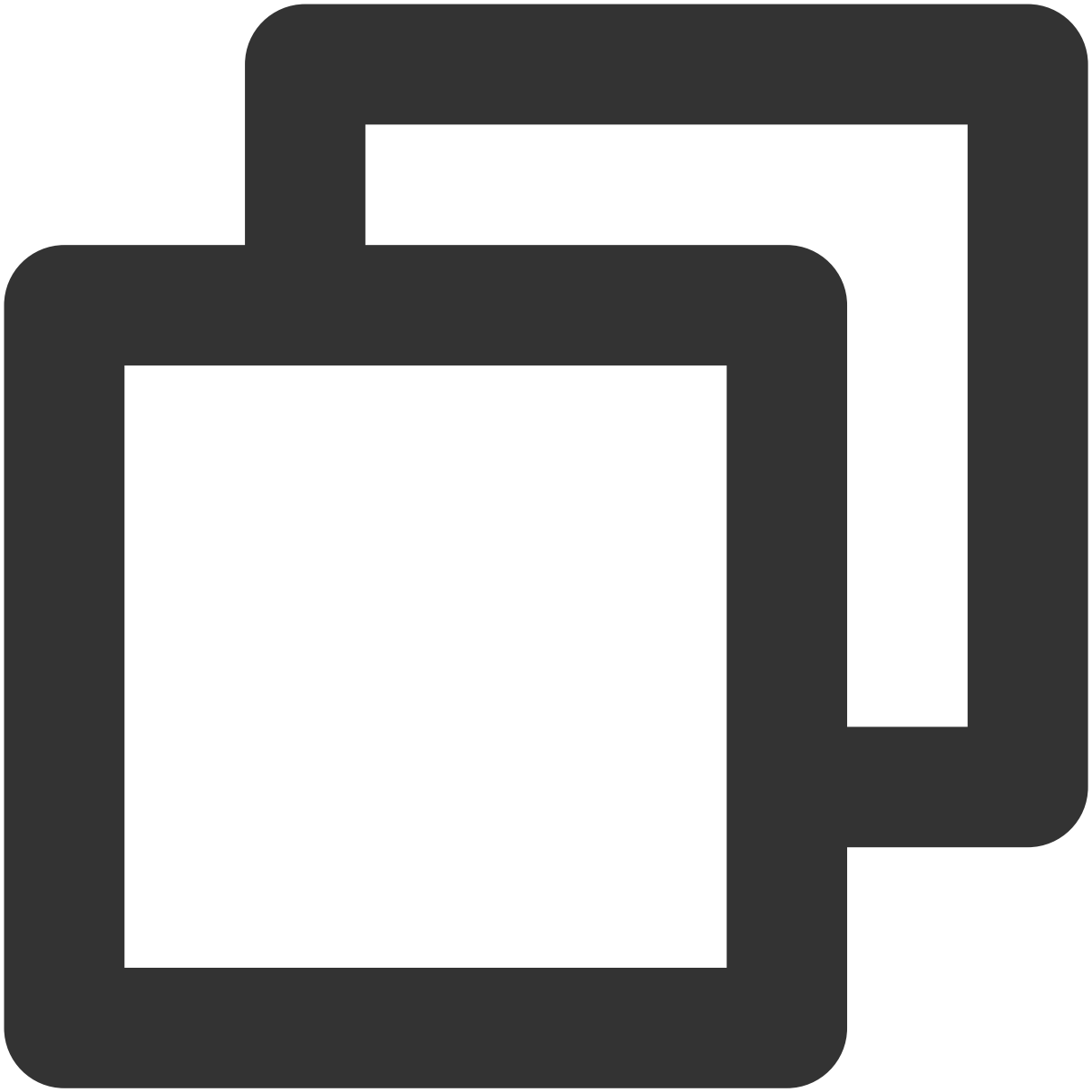
```
RANDOM([<seed> integer])
```

Supported engines: SparkSQL and Presto

Usage instructions: Return a random value that is independently and uniformly distributed within the range [0, 1].

Return type: double

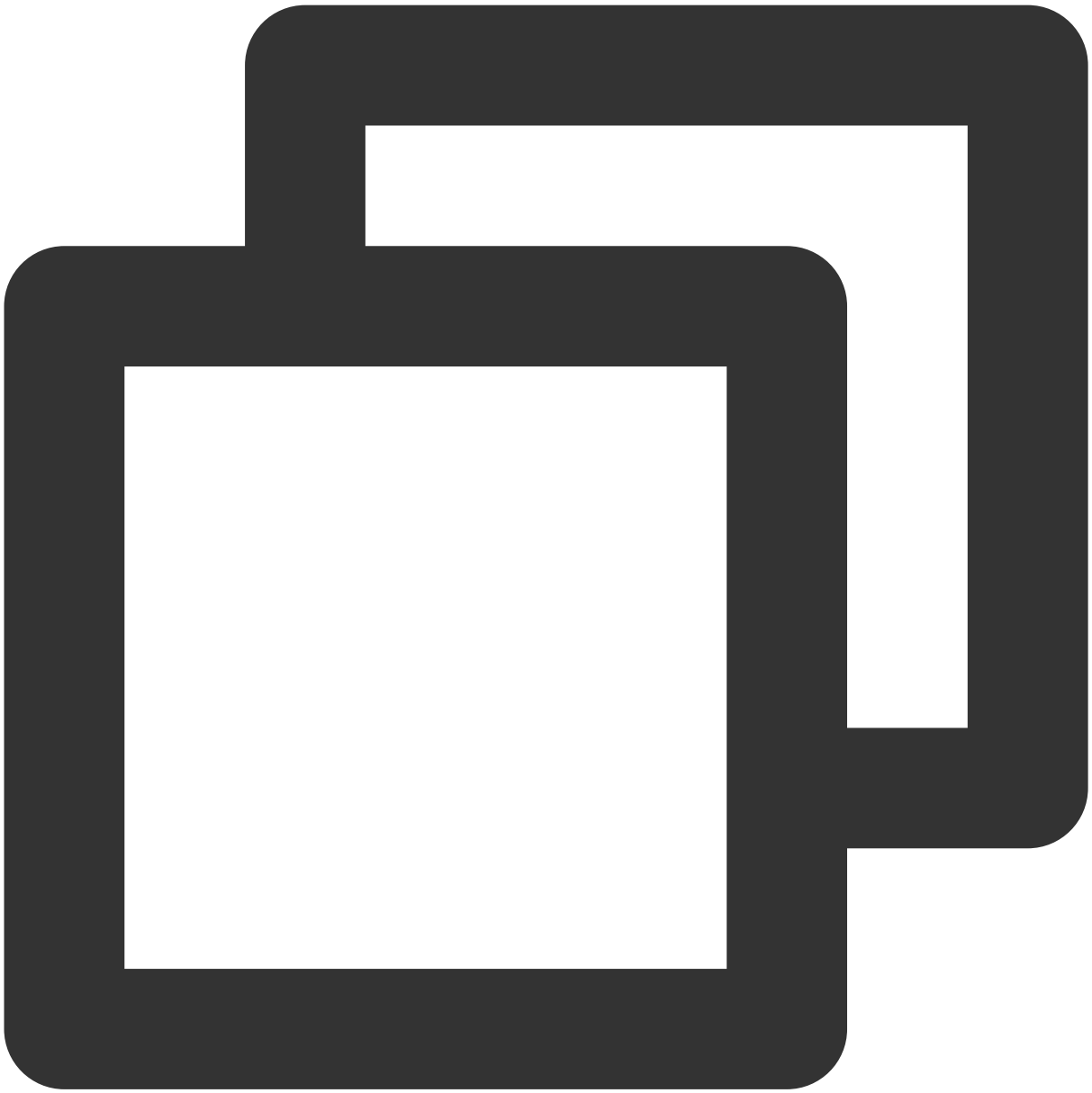
Example:



```
> SELECT rand();
 0.9629742951434543
> SELECT rand(0);
 0.7604953758285915
> SELECT rand(null);
 0.7604953758285915
```

RANDN

Function statement:



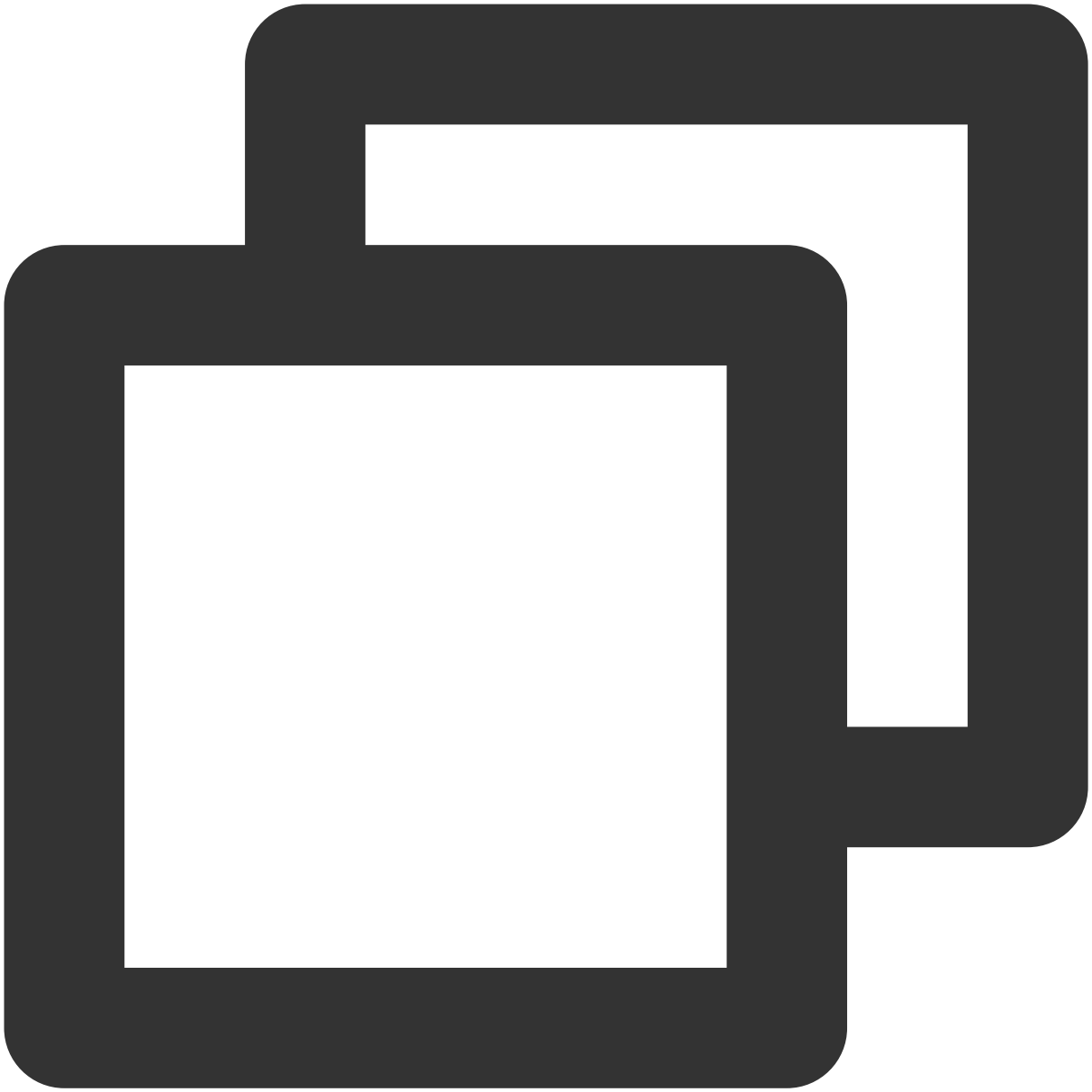
```
RANDN([<seed> integer])
```

Supported engines: SparkSQL and Presto

Usage instructions: Return a random value that is independently and identically distributed (i.i.d.) from a standard normal distribution. The implementation of this function differs between SPARKSQL and PRESTO. Using the same seed may yield different results.

Return type: double

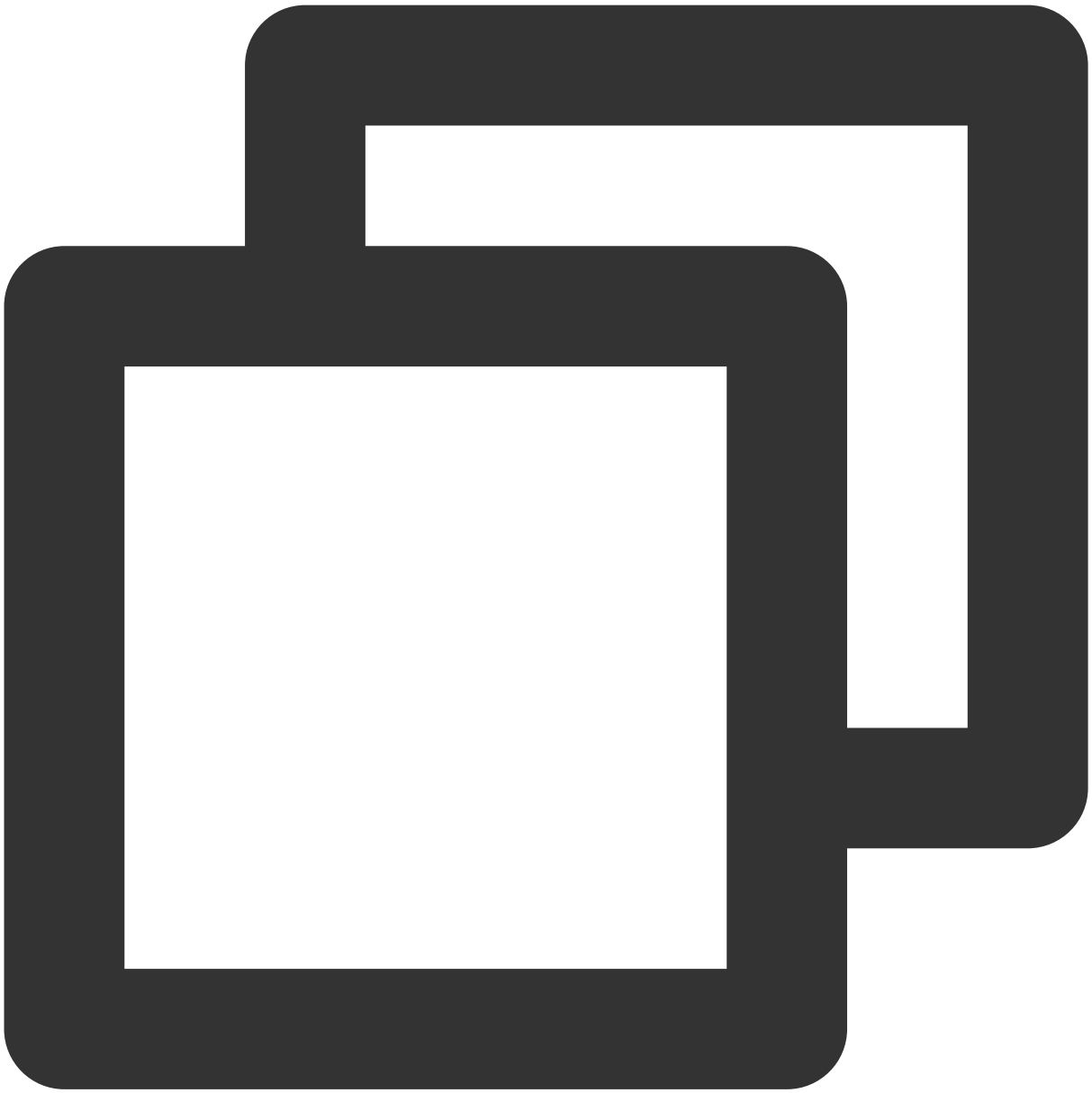
Example:



```
> SELECT randn();
 -0.3254147983080288
> SELECT randn(0);
 1.6034991609278433
> SELECT randn(null);
 1.6034991609278433
```

DIV

Function statement:



```
<expr1> DIV <expr2>
```

Supported engine: SparkSQL

Usage instructions: Divide expr1 by expr2.

Return type: integer

Example:

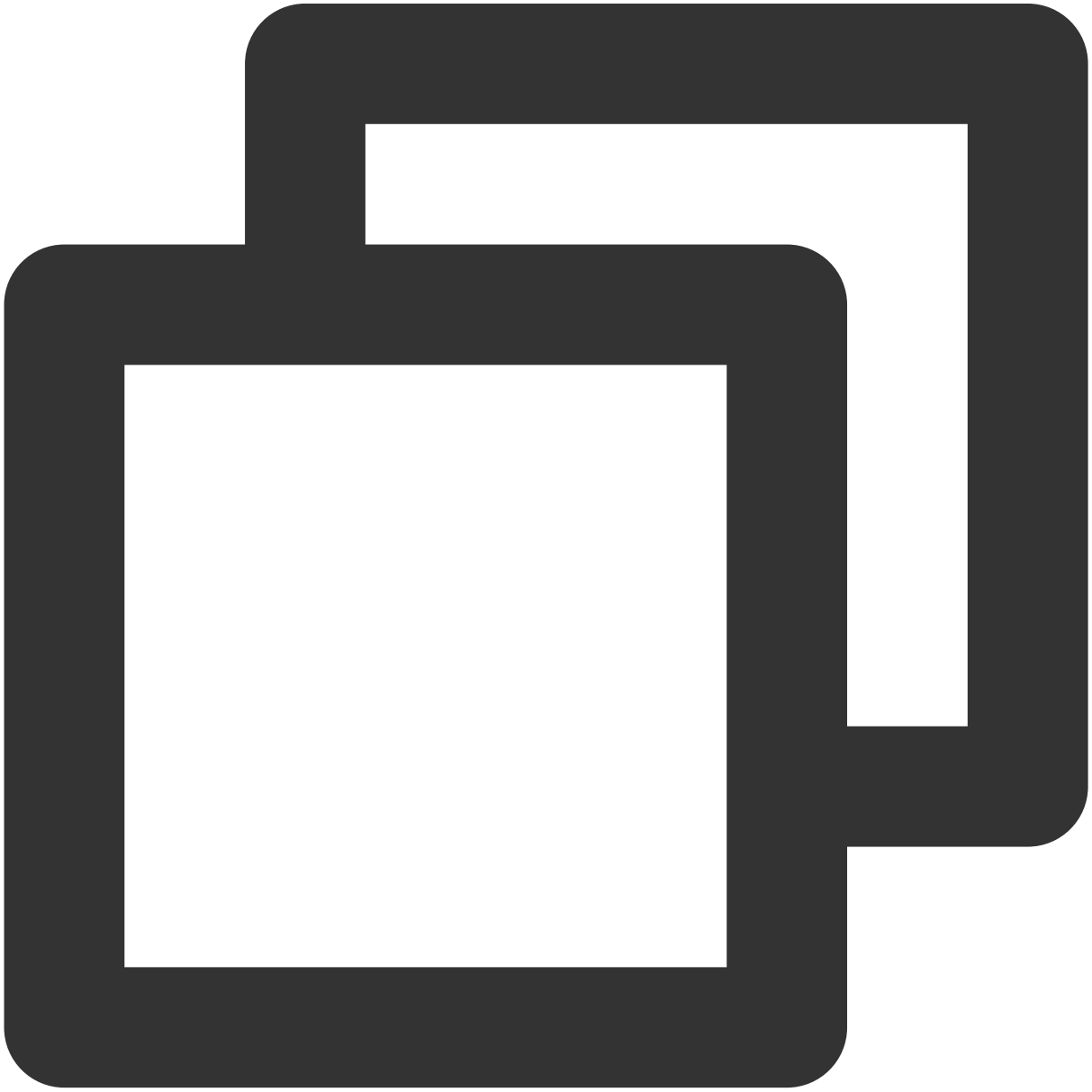
```
> SELECT 3 div 2;
 1
```

String Functions

Last updated：2024-08-07 17:32:55

ASCII

Function statement:



```
ASCII(<str> string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the numeric value of the first character of str.

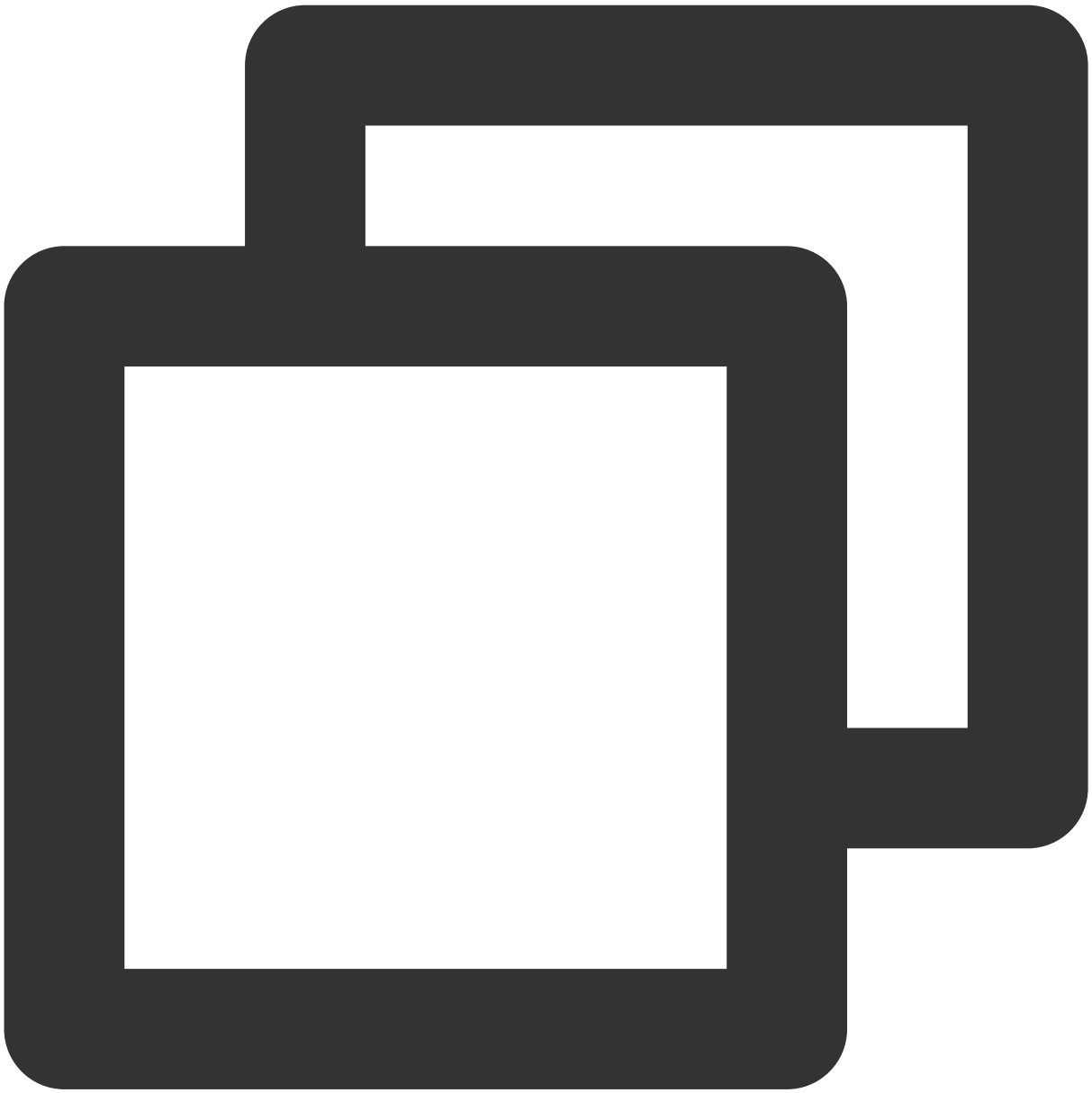
Return type: integer

Example:

```
> SELECT ascii('222');
 50
```

BASE64

Function statement:



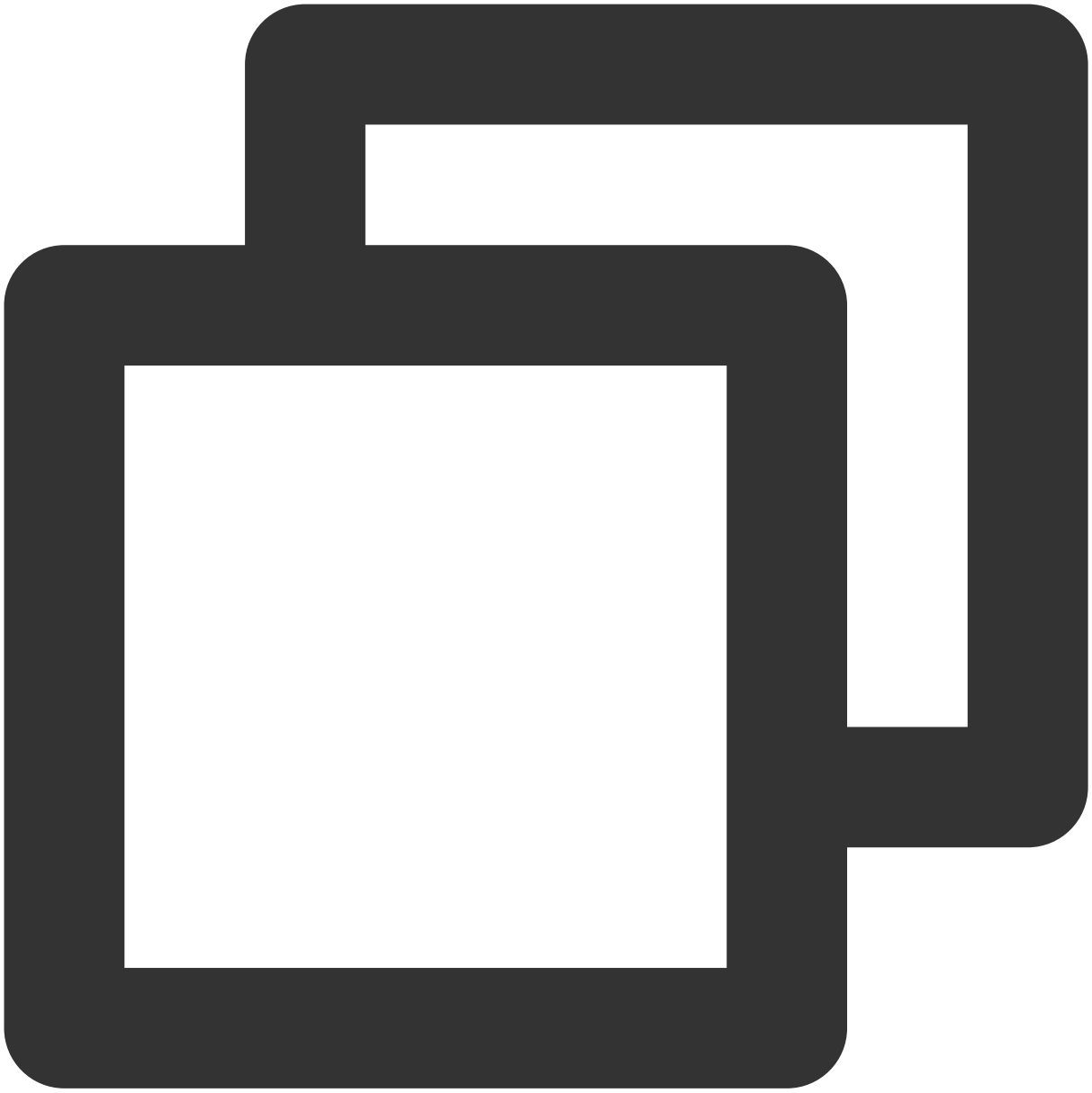
```
-- SparkSQL
BASE64(<str> string|binary)
-- Presto
BASE64(<str> binary)
```

Supported engines: SparkSQL and Presto

Usage instructions: Convert the parameter to a base64 string.

Return type: string

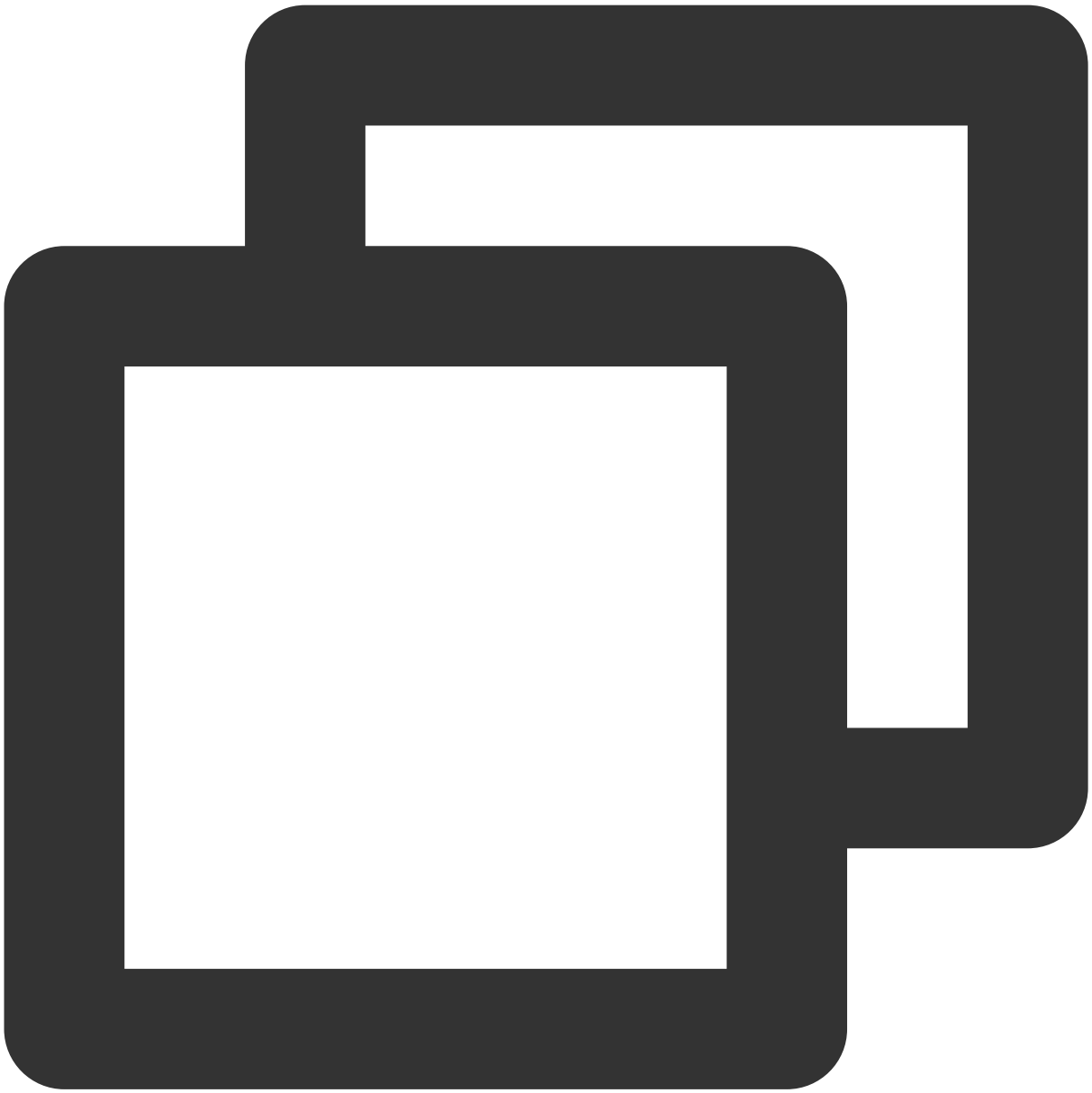
Example:



```
> SELECT base64('tencent');
 dGVuY2VudA==
```

BIT_LENGTH

Function statement:



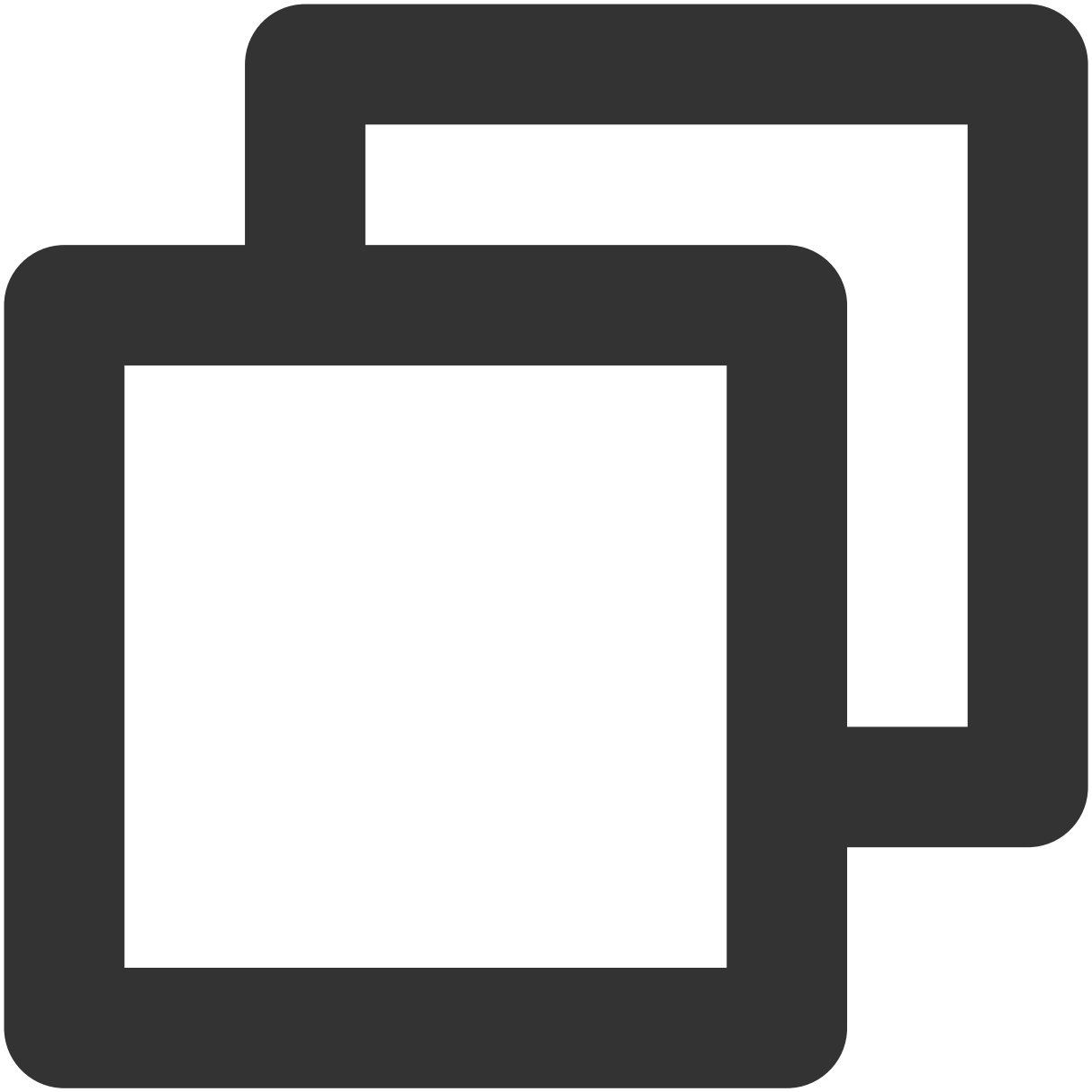
```
BIT_LENGTH(<expr> string|binary)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the bit length of string data or the number of bits in binary data.

Return type: integer

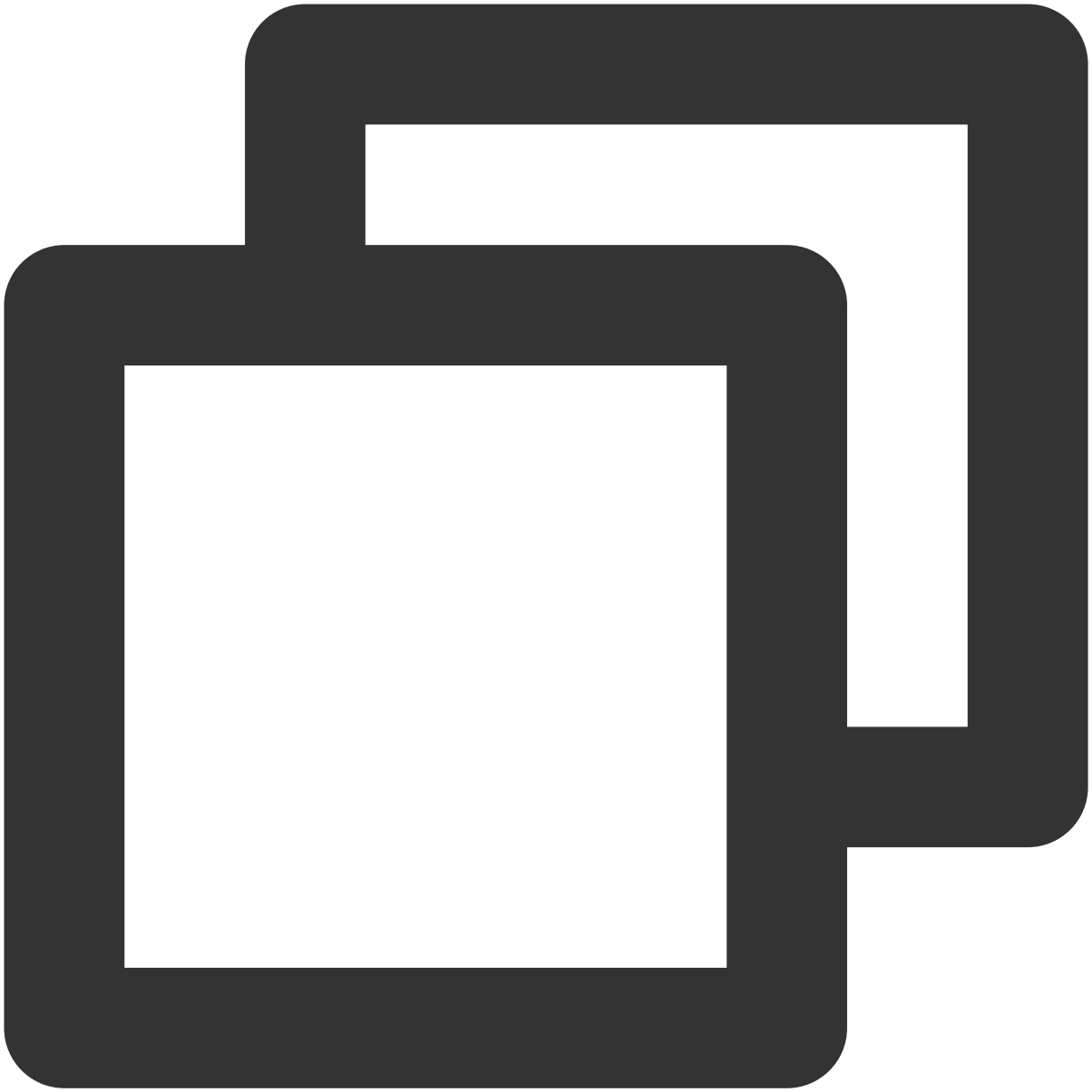
Example:



```
> select bit_length('tencent');
 56
> select bit_length(binary('tencent'));
 56
```

CHAR

Function statement:



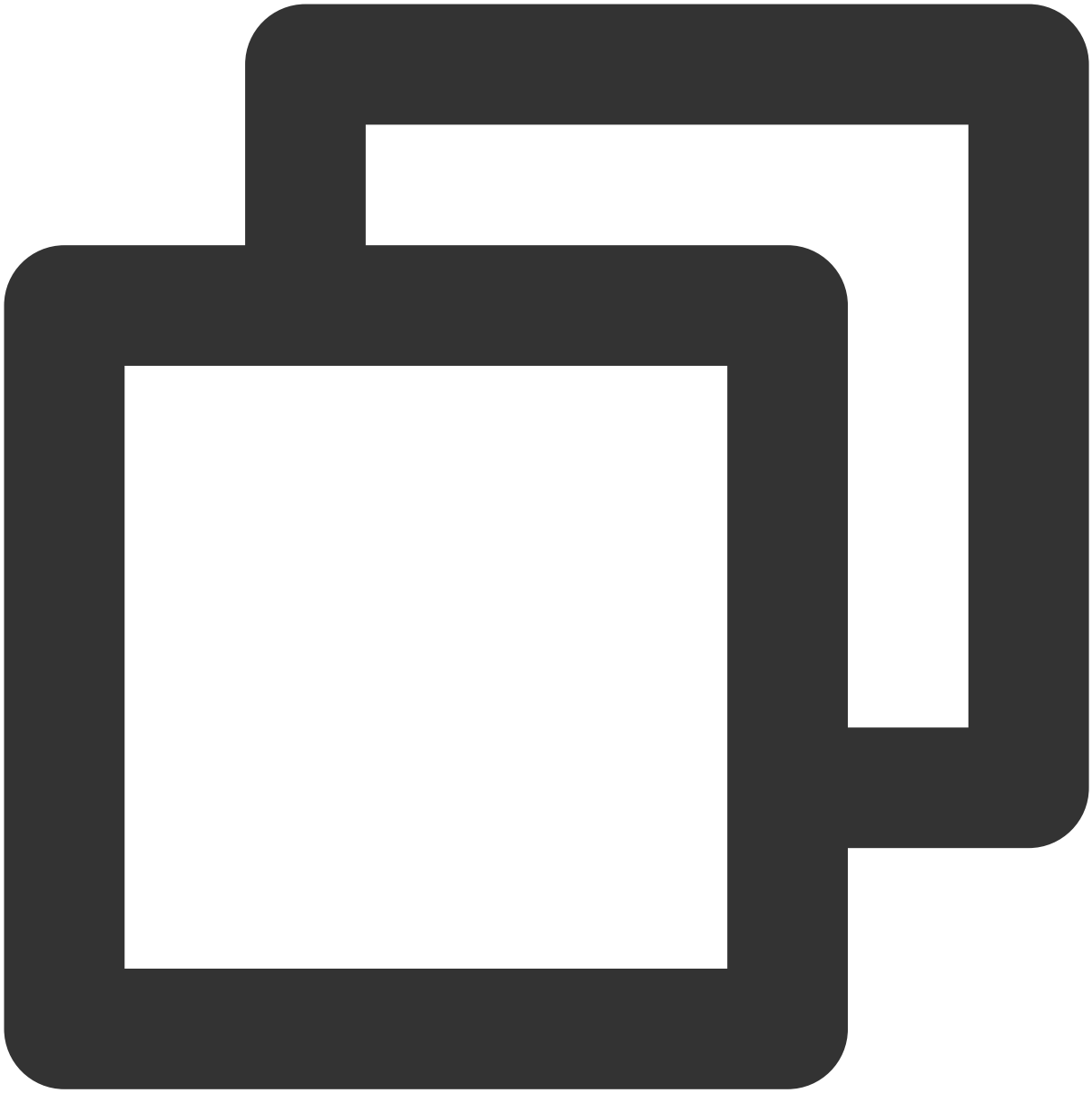
```
CHAR(<expr> integer)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the ASCII character of expr. If expr is greater than 256, then expr = expr %256.

Return type: string

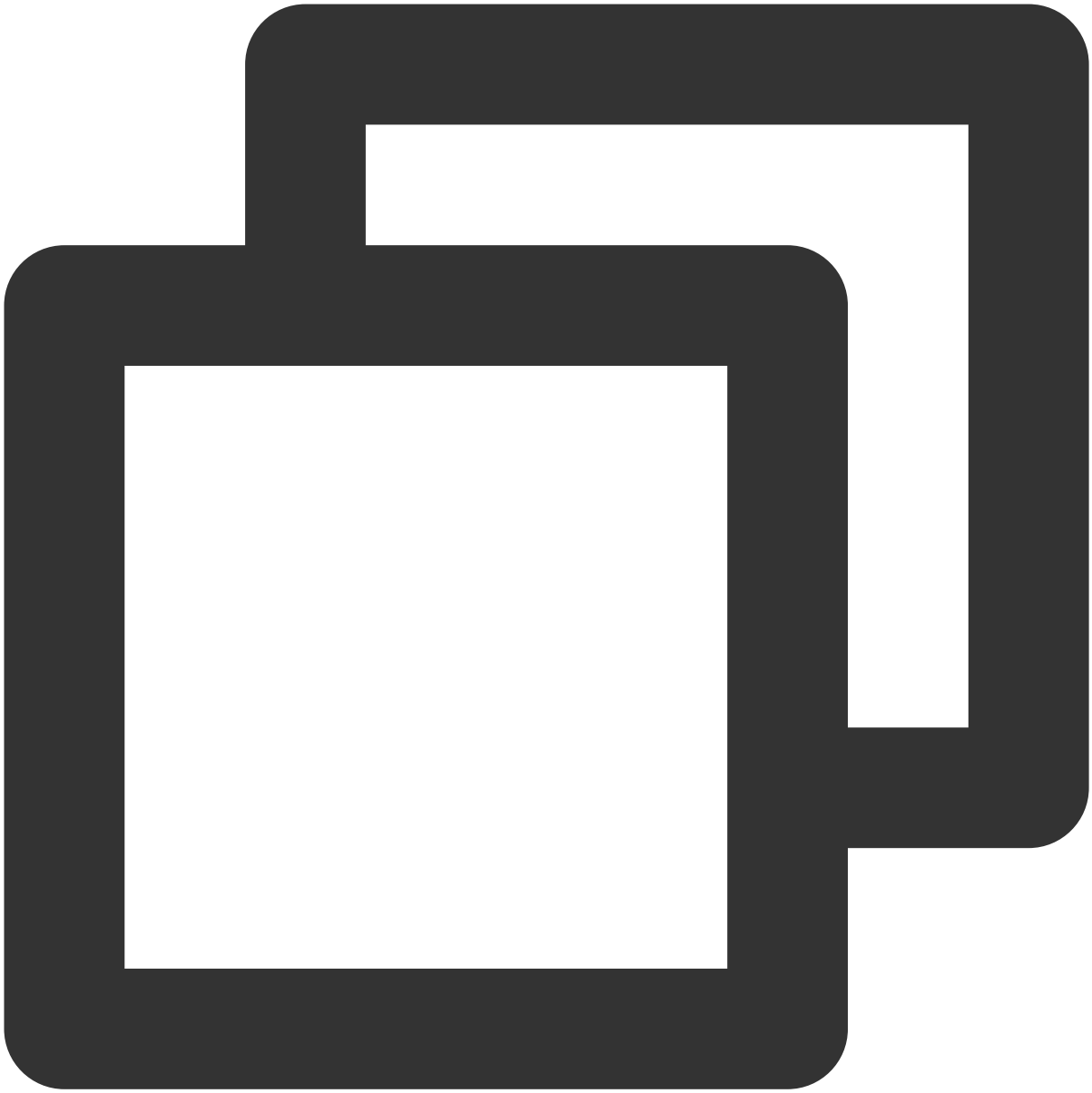
Example:



```
> SELECT char(65);
 A
```

CHR

Function statement:



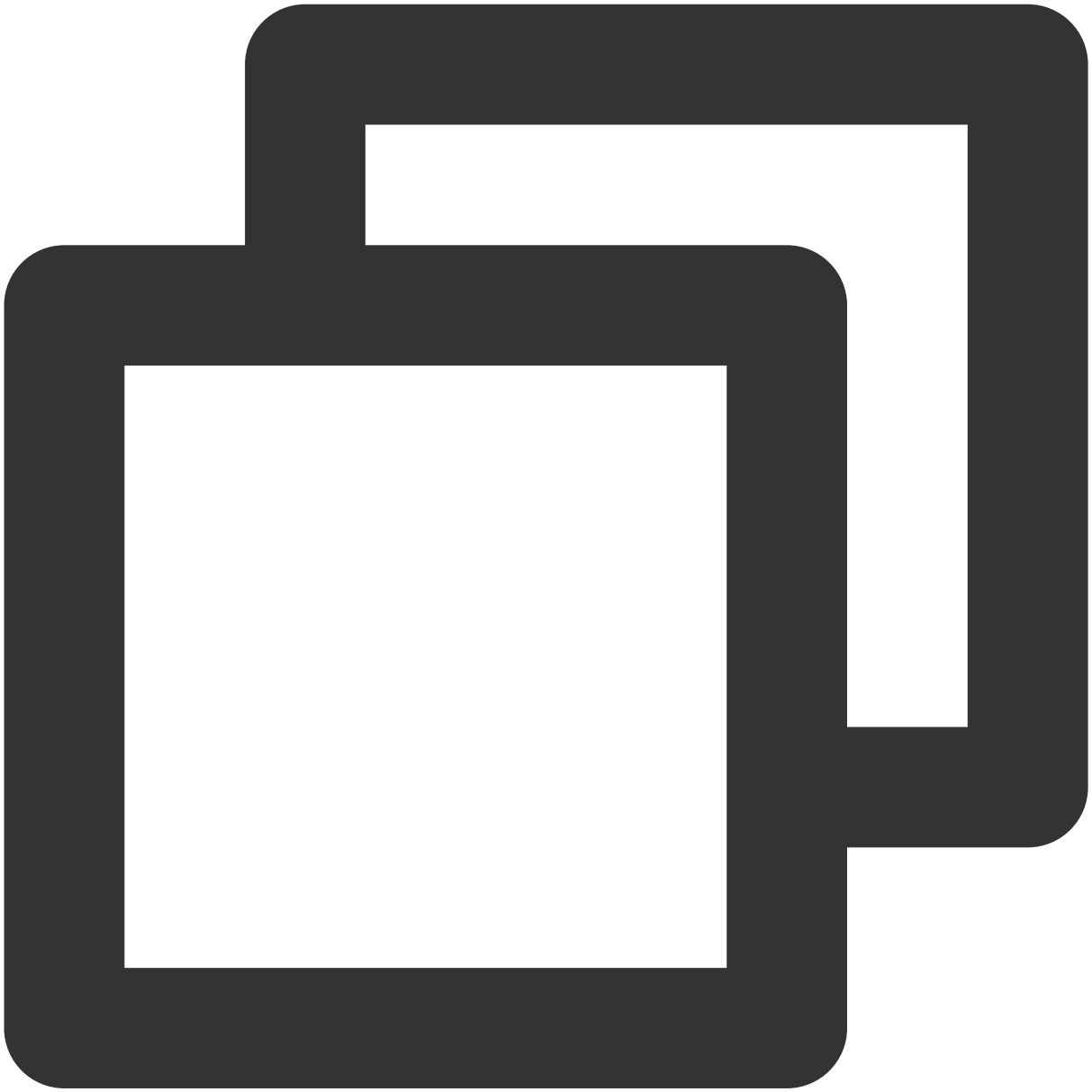
```
CHR(<expr> integer)
```

Supported engine: SparkSQL

Usage instructions: Return the ASCII character of expr. If expr is greater than 256, then expr = expr %256.

Return type: string

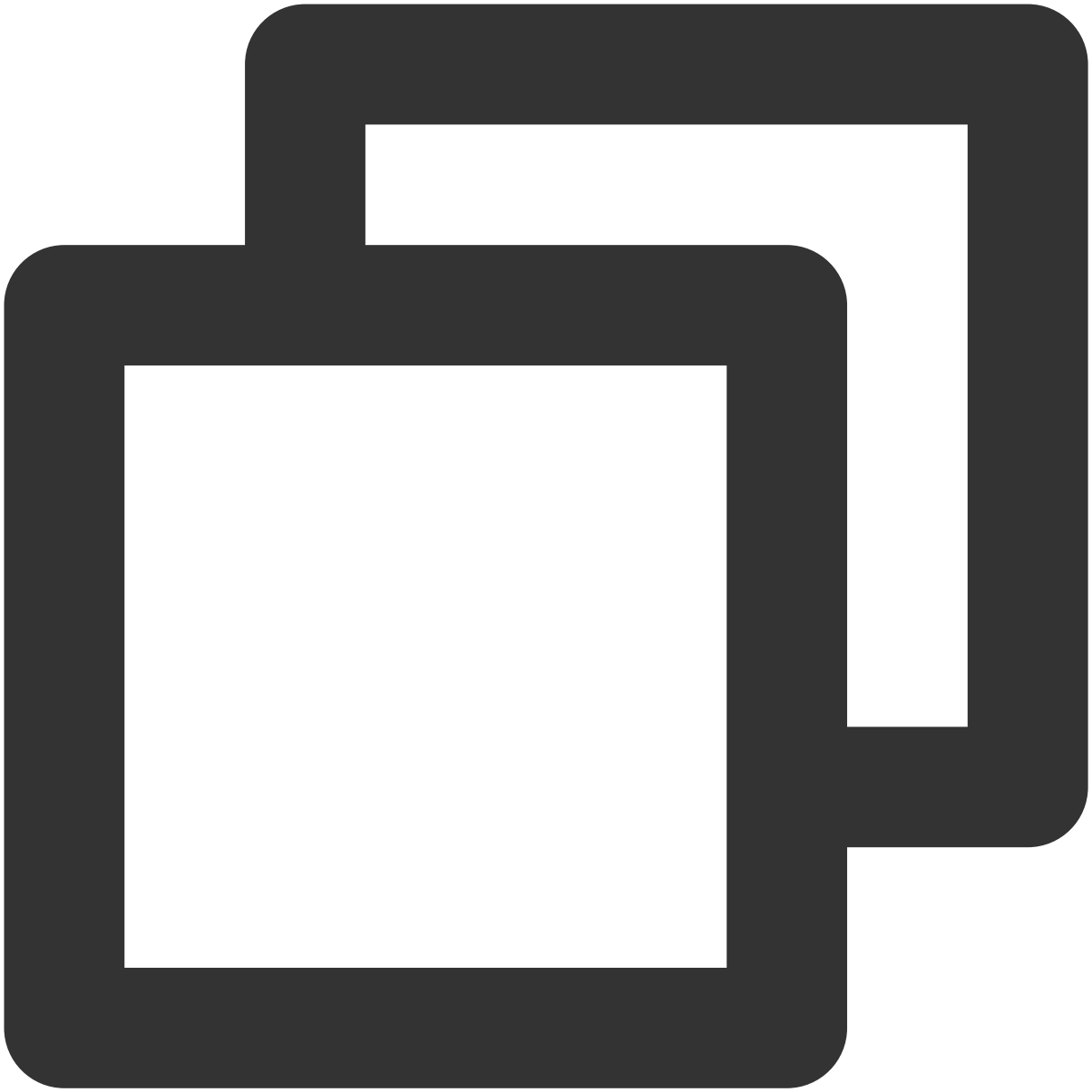
Example:



```
> SELECT chr(65);
 A
```

CHAR_LENGTH

Function statement:



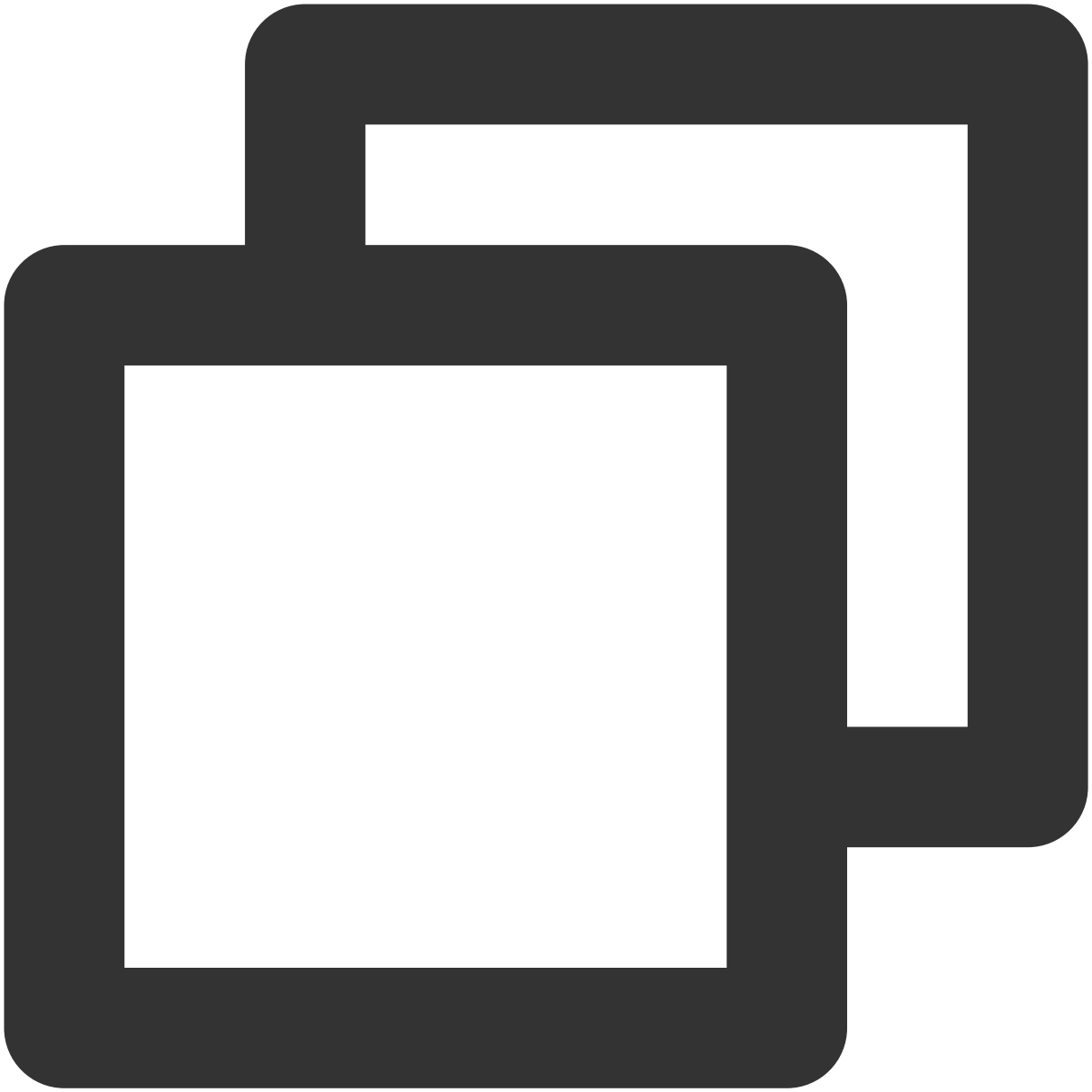
```
CHAR_LENGTH(<expr> string|binary)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the character length of the string data or the number of bytes in the binary data. The length of the string data includes trailing spaces. The length of the binary data includes binary zeros.

Return type: integer

Example:



```
> select char_length(binary('tencent'));
 7
> select char_length('tencent');
 7
```

CHARACTER_LENGTH

Function statement:

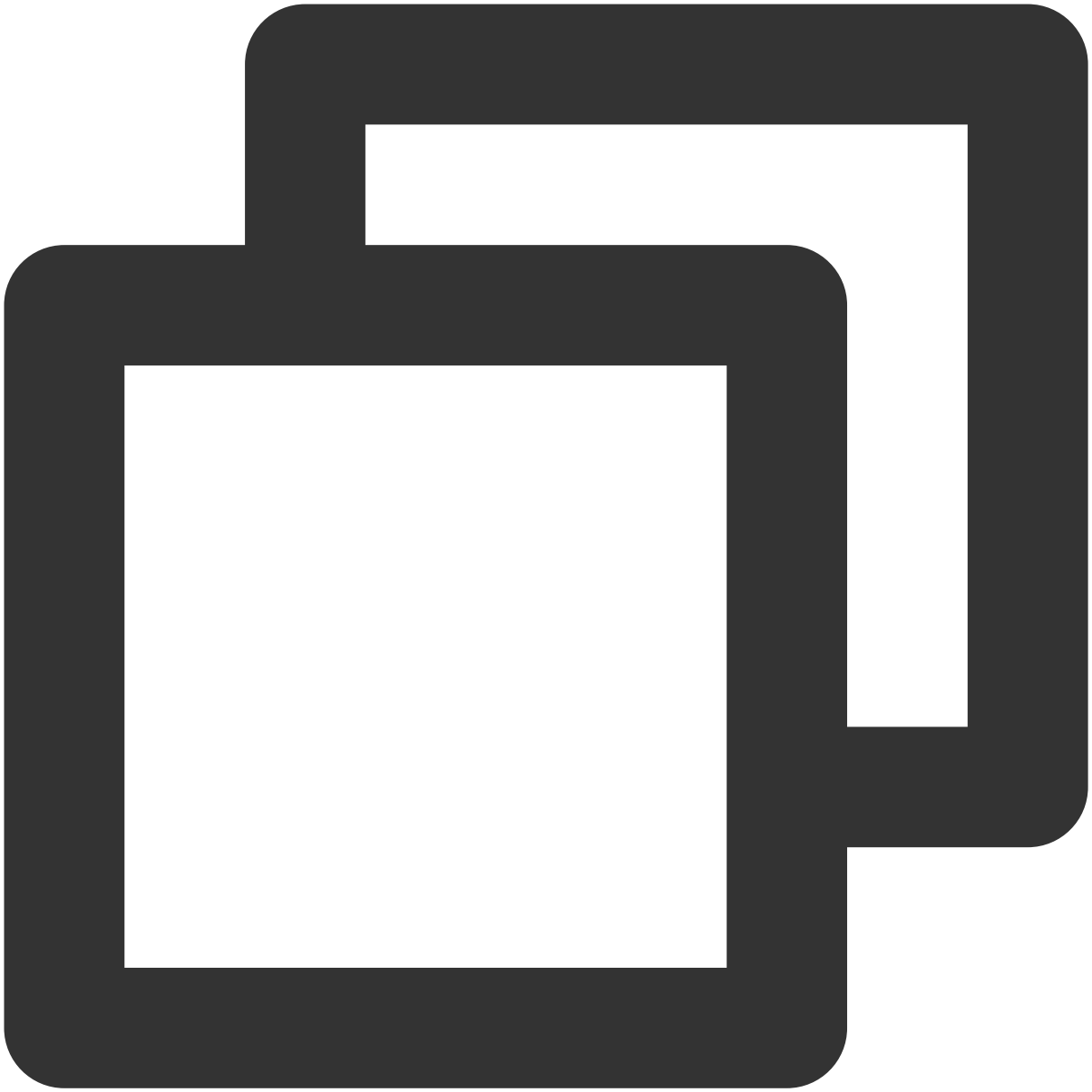
```
CHARACTER_LENGTH(<expr> string|binary)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the character length of the string data or the number of bytes in the binary data. The length of the string data includes trailing spaces. The length of the binary data includes binary zeros.

Return type: integer

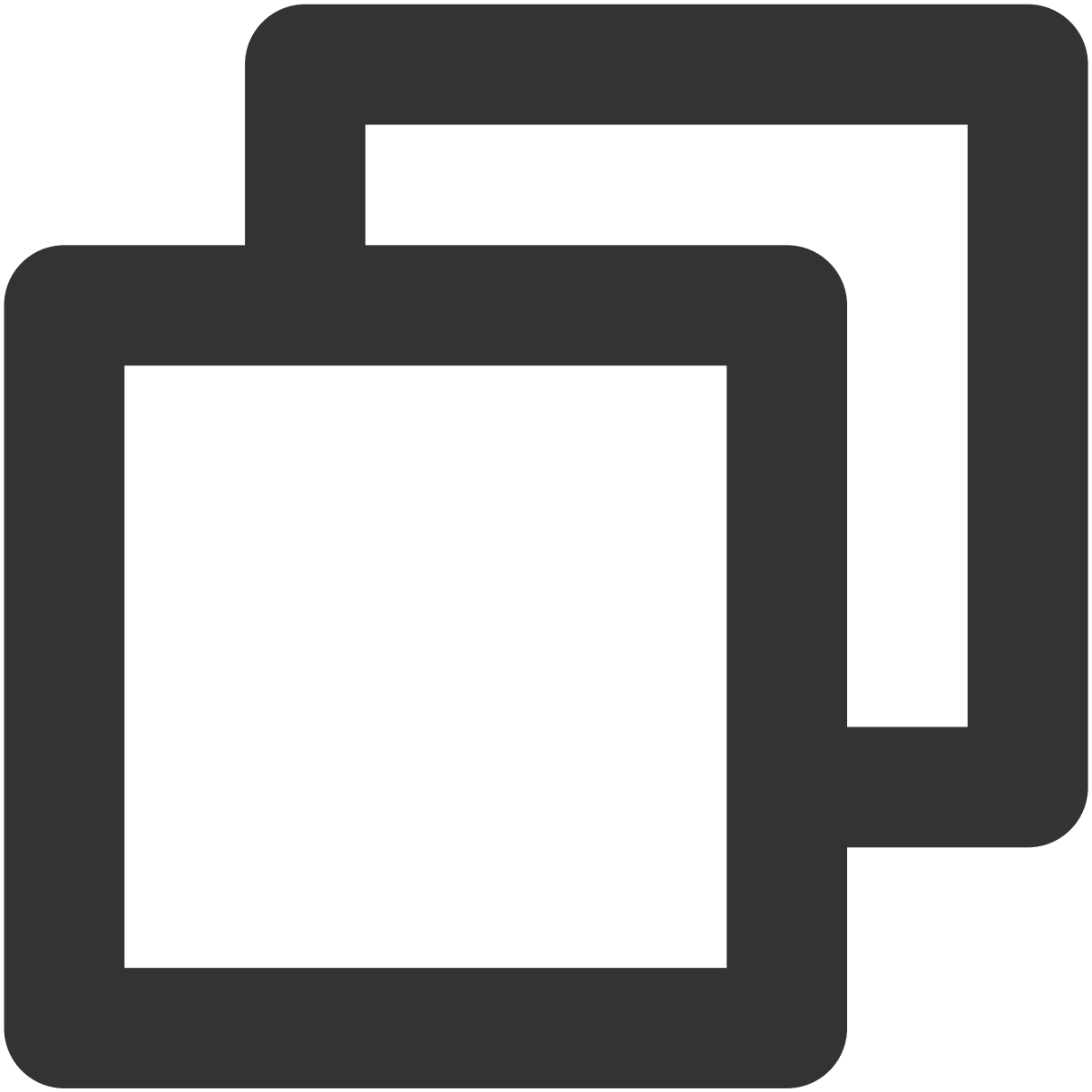
Example:



```
> select character_length(binary('tencent'));
 7
> select character_length('tencent');
 7
```

CONCAT_WS

Function statement:



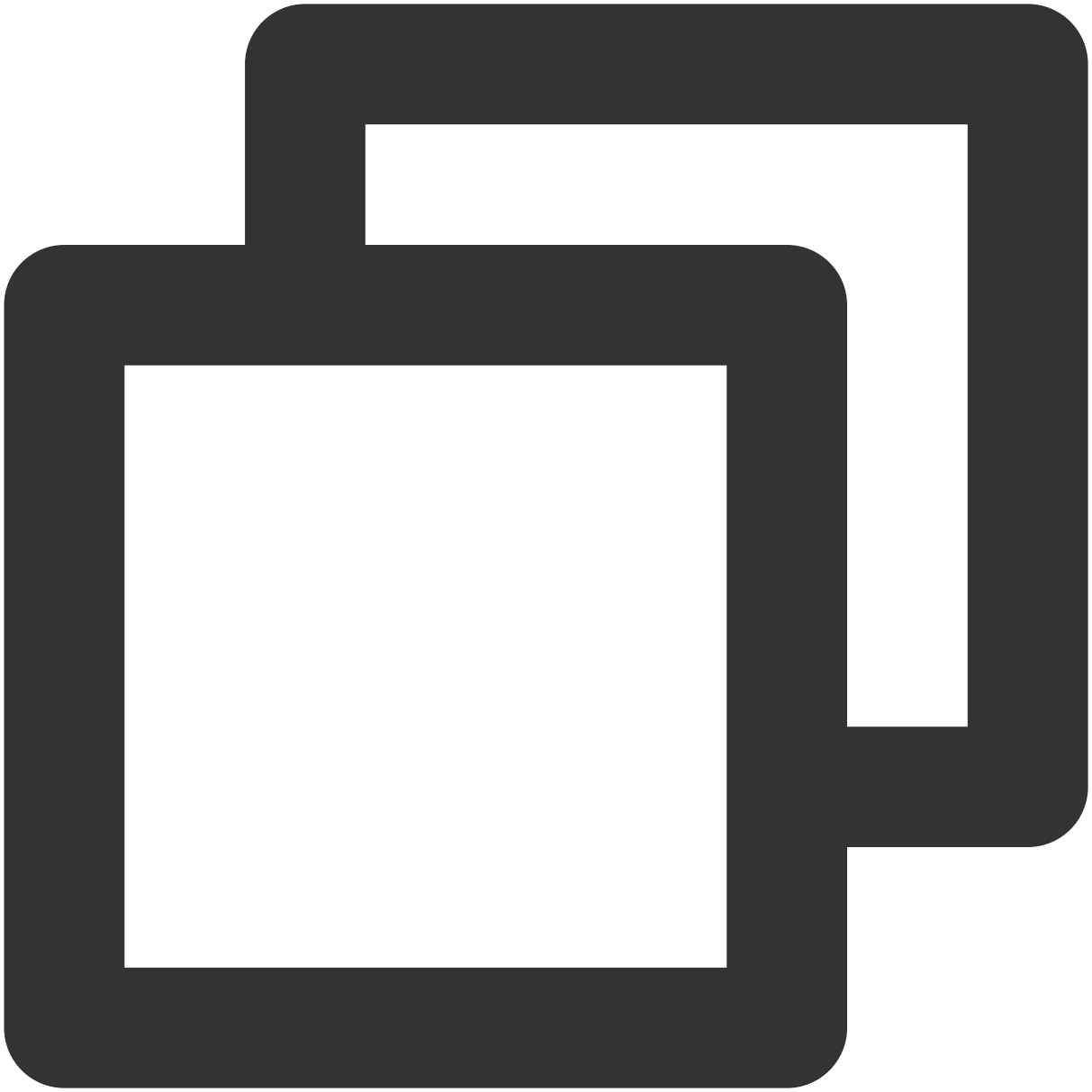
```
CONCAT_WS(<sep> string[, <s> string|array<string>}]+)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return a string separated by sep.

Return type: string

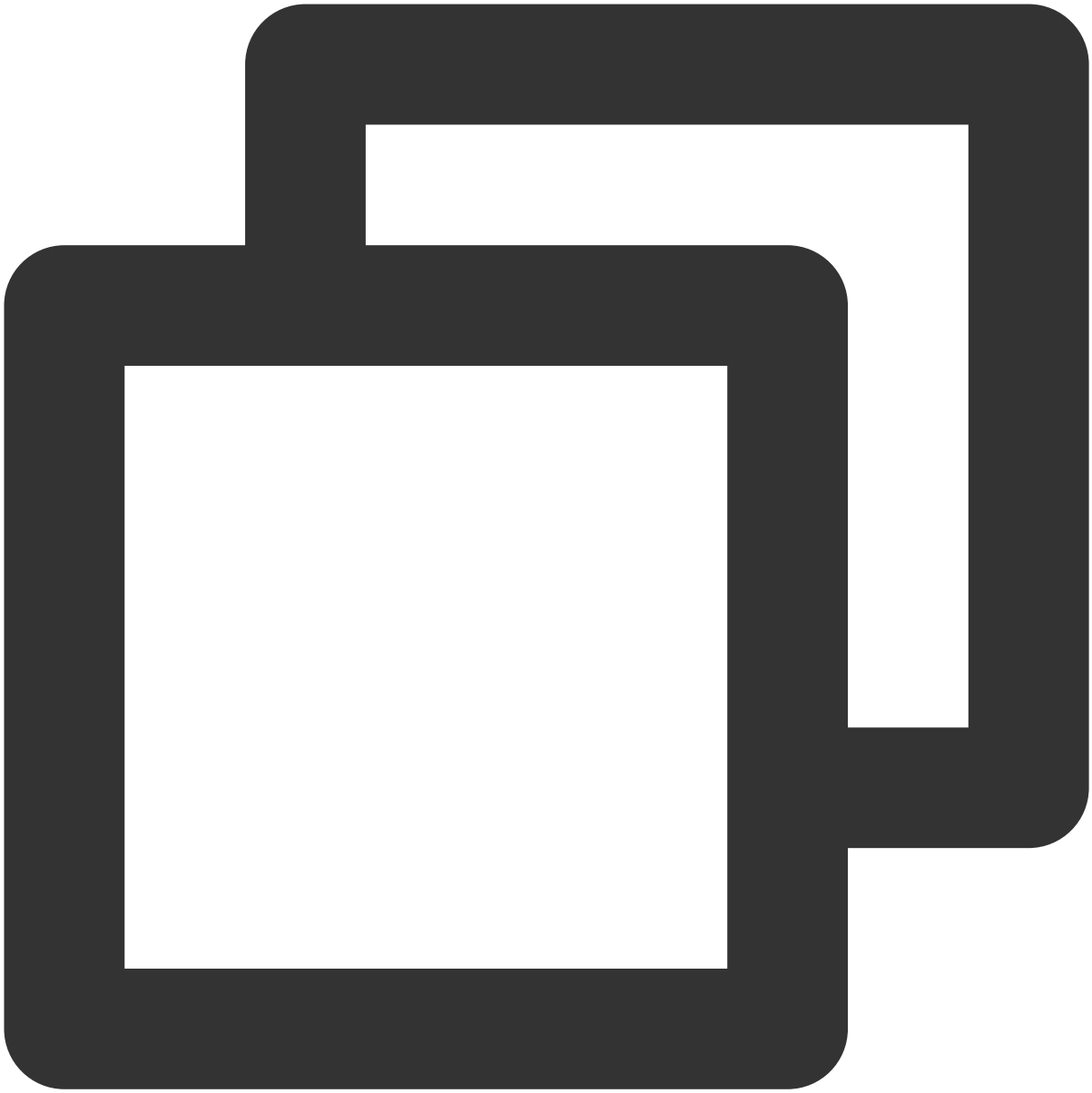
Example:



```
> SELECT concat_ws(' ', 'tencent', 'dlc');
tencent dlc
```

DECODE

Function statement:



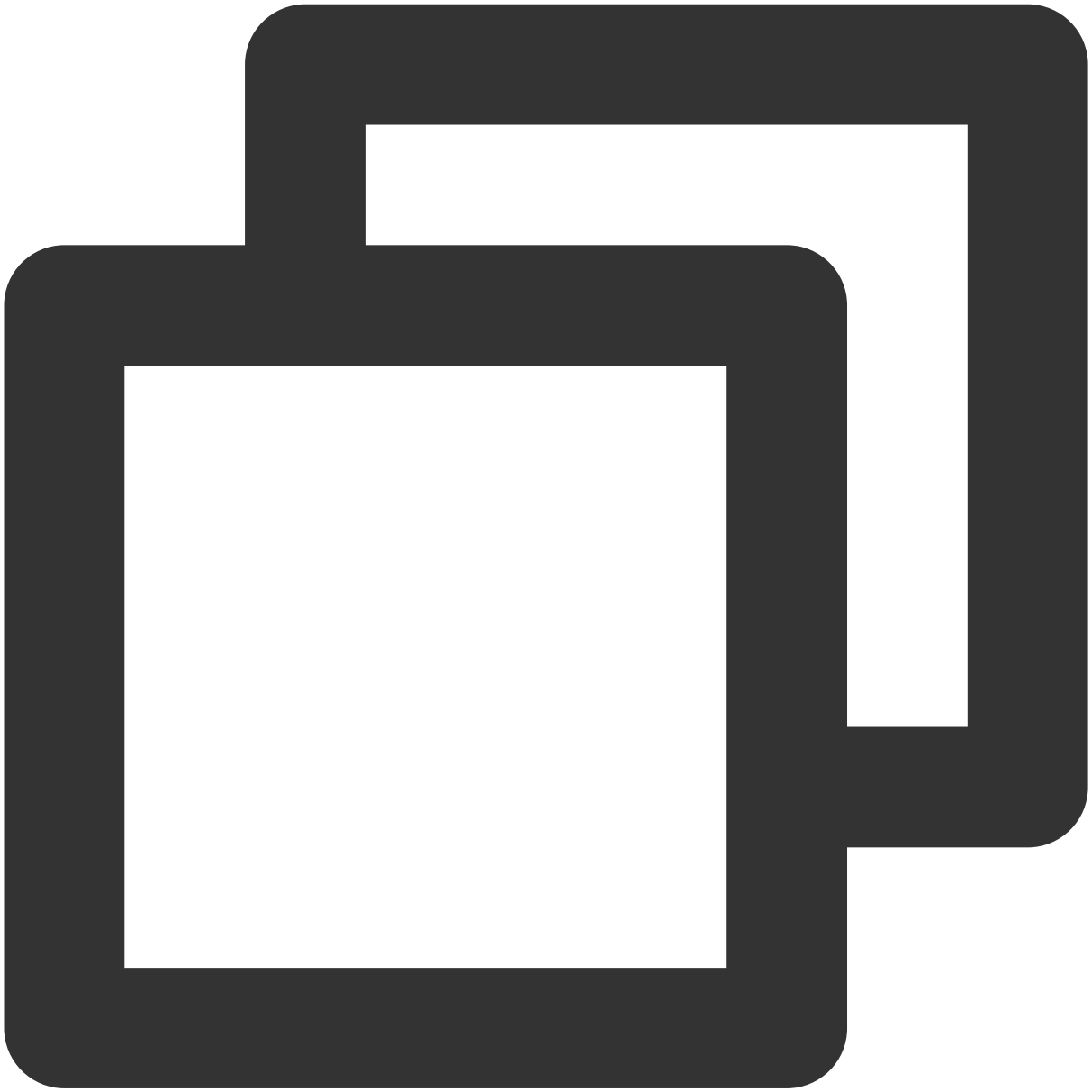
```
DECODE(<expr> binary|string, <charset>string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Decode the first parameter using the character set specified by the second parameter.

Return type: string

Example:



```
> SELECT decode(encode('abc', 'utf-8'), 'utf-8');
 abc
```

ELT

Function statement:

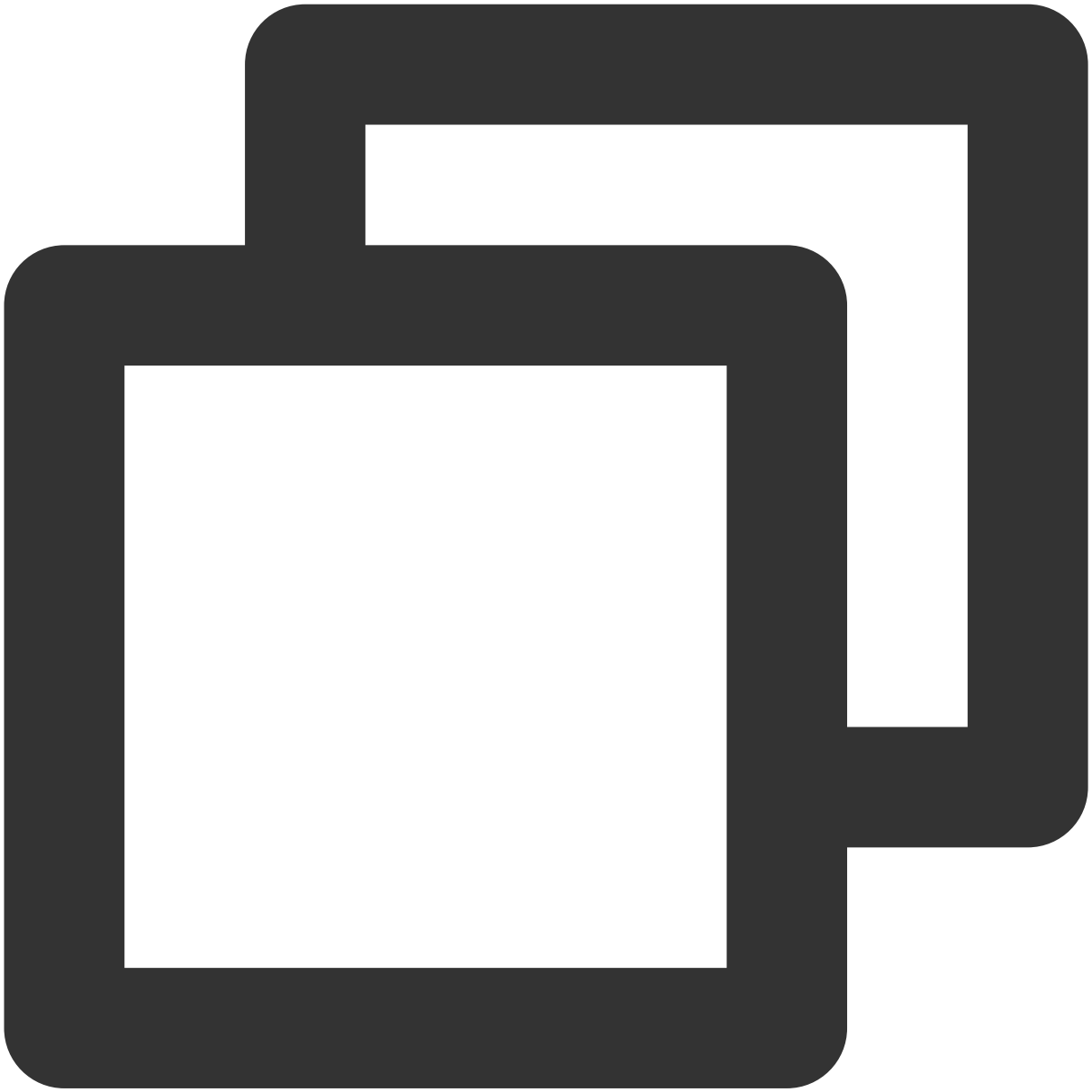
```
ELT(<n> integer, <s1> string, <s2> string, ...)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the nth element.

Return type: string

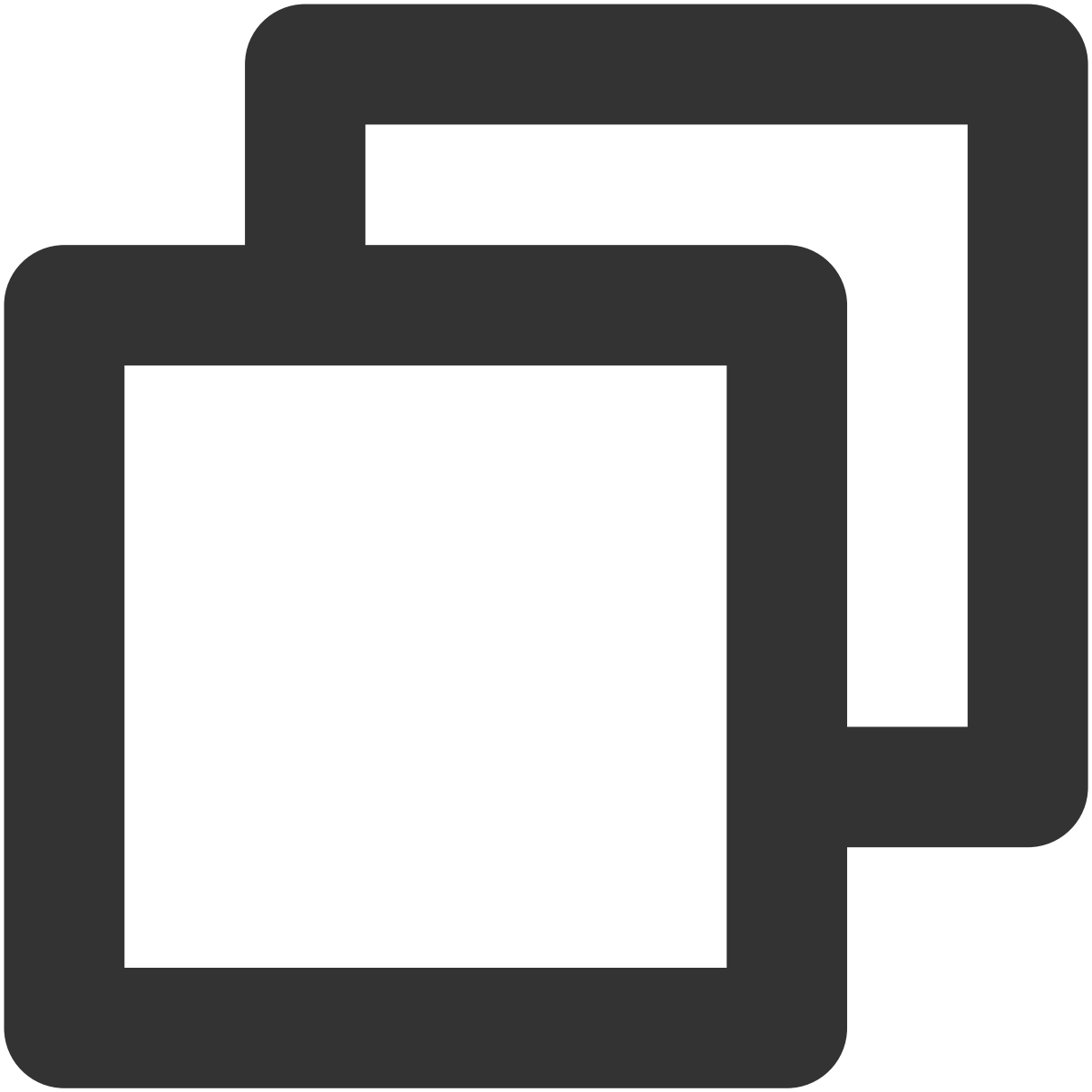
Example:



```
> SELECT elt(1, 'scala', 'java');
 scala
```

ENCODE

Function statement:



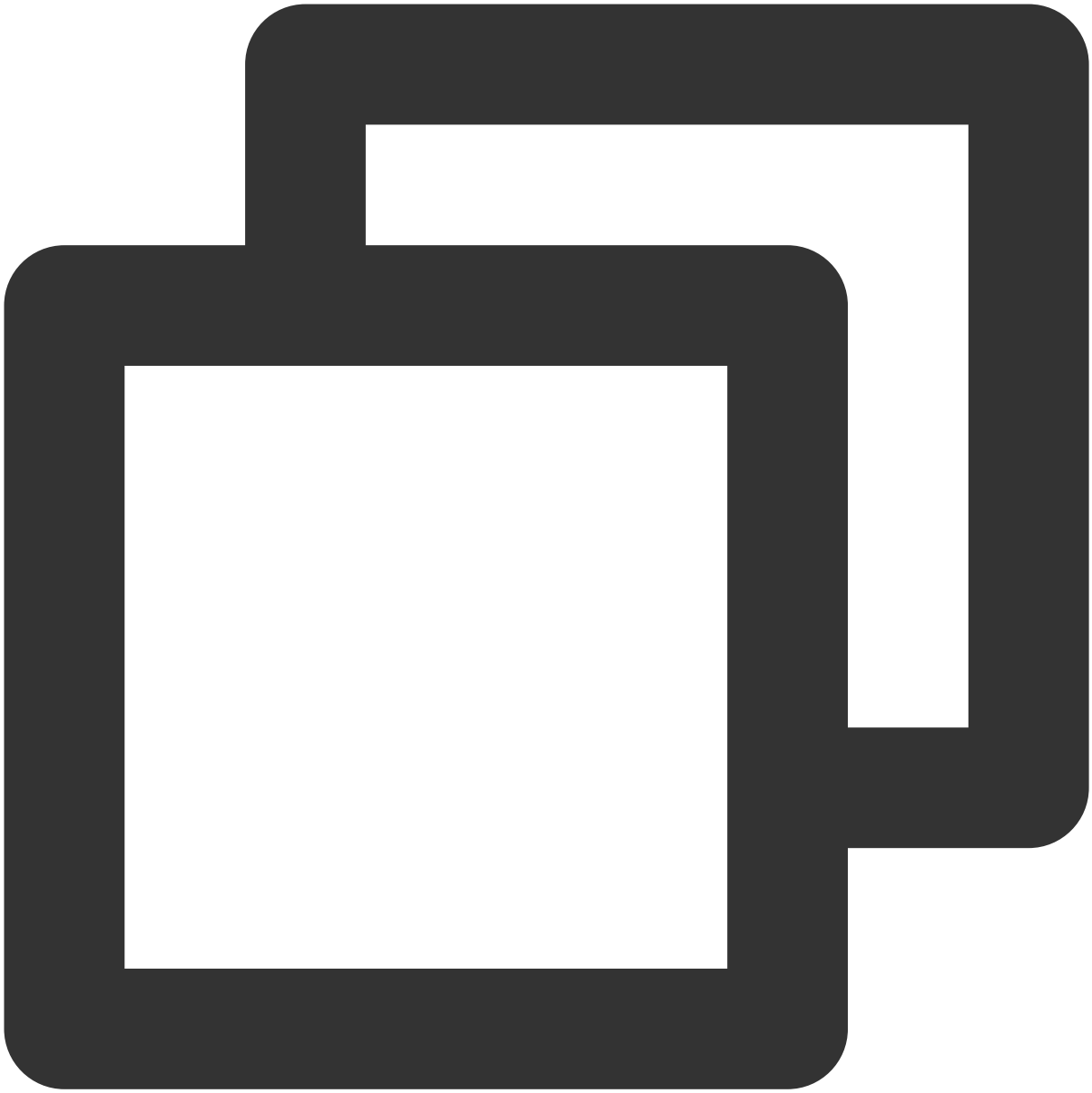
```
ENCODE(<expr> binary|string, <charset> string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Encode the first parameter using the character set specified by the second parameter.

Return type: string

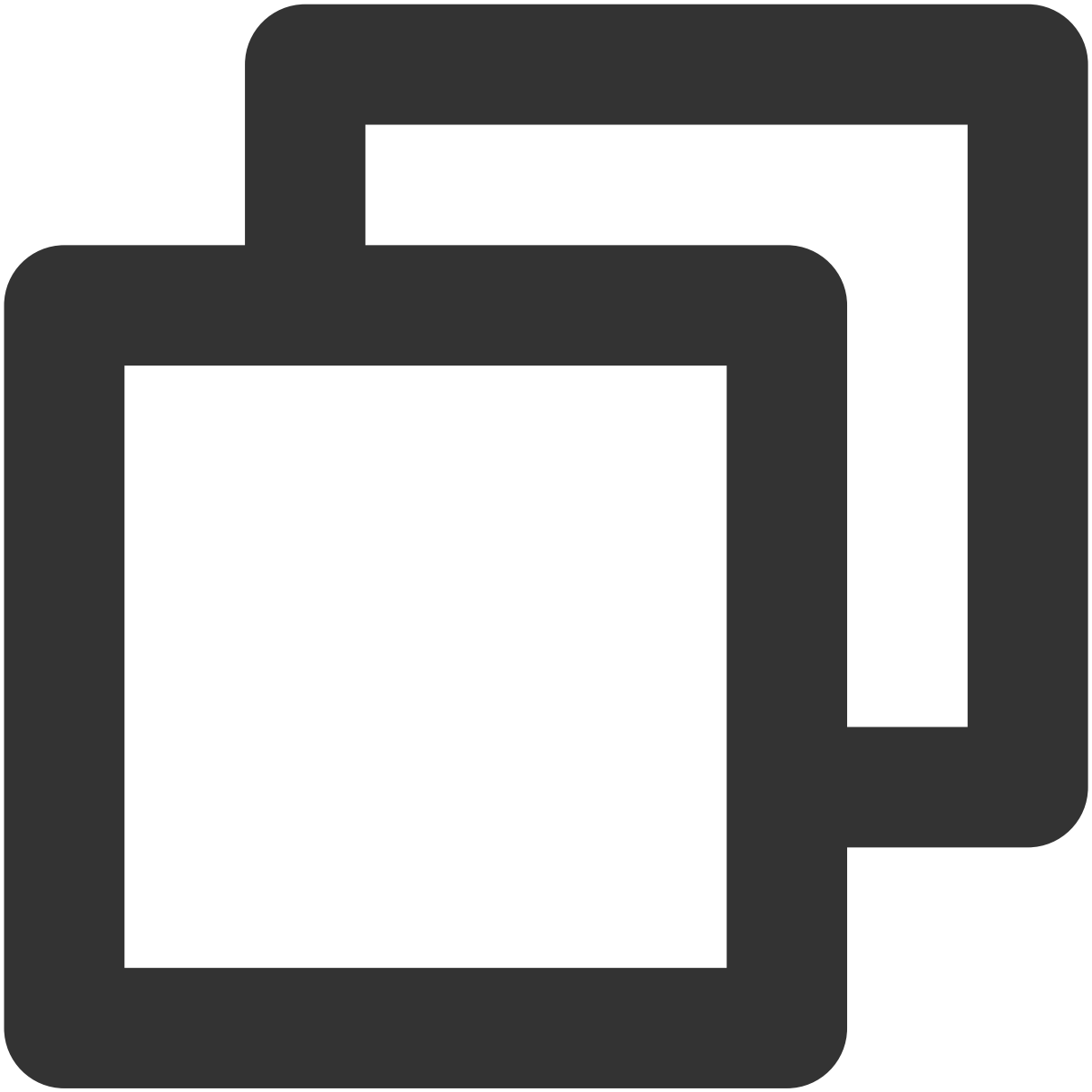
Example:



```
> SELECT encode('abc', 'utf-8');
 abc
> SELECT encode(x'616263', 'utf-8');
 abc
```

FIND_IN_SET

Function statement:



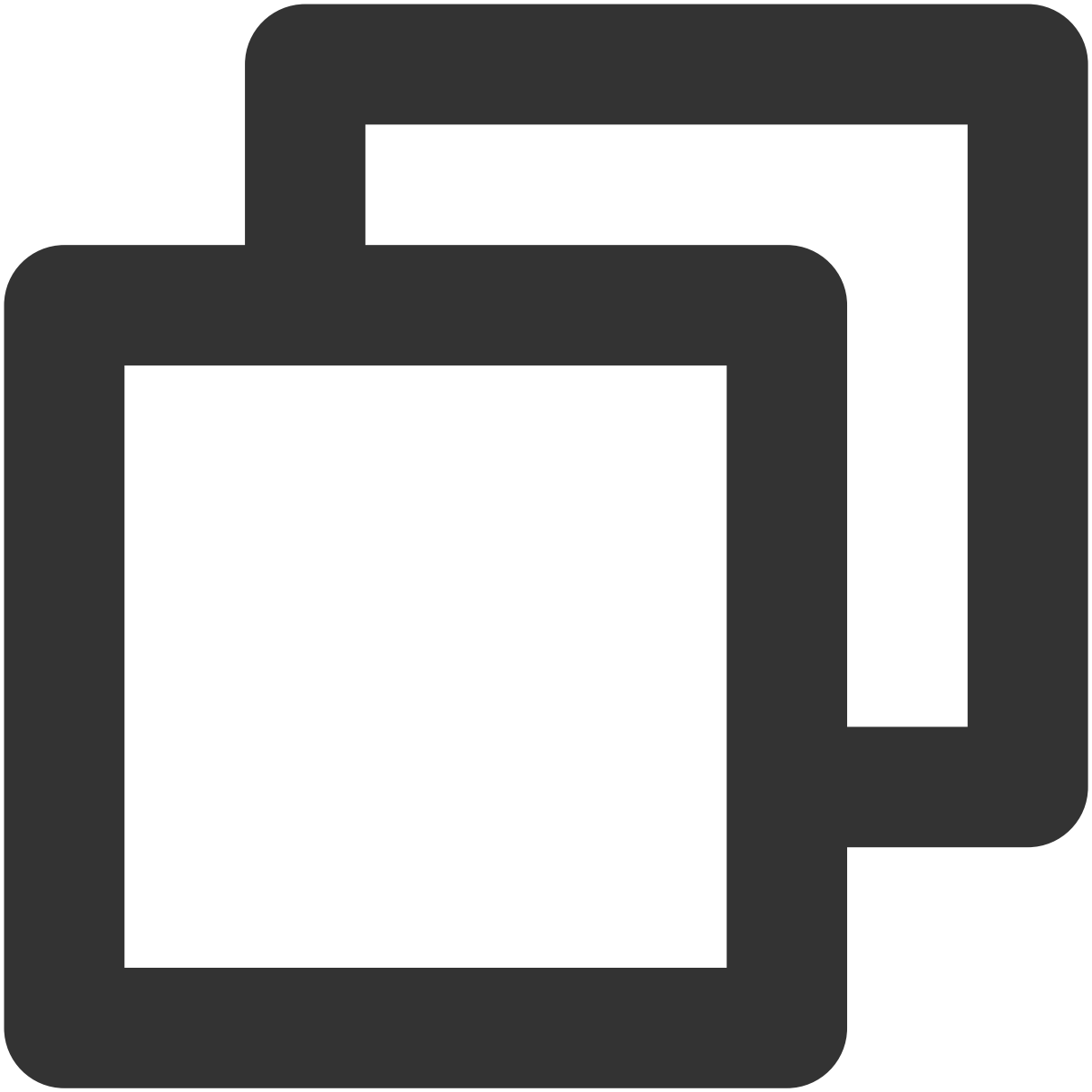
```
FIND_IN_SET(<str> string, <str_array> string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the index (starting from 1) of the given string str in the comma-separated list atr_array. If the string is not found or if str contains a comma, 0 is returned.

Return type: integer

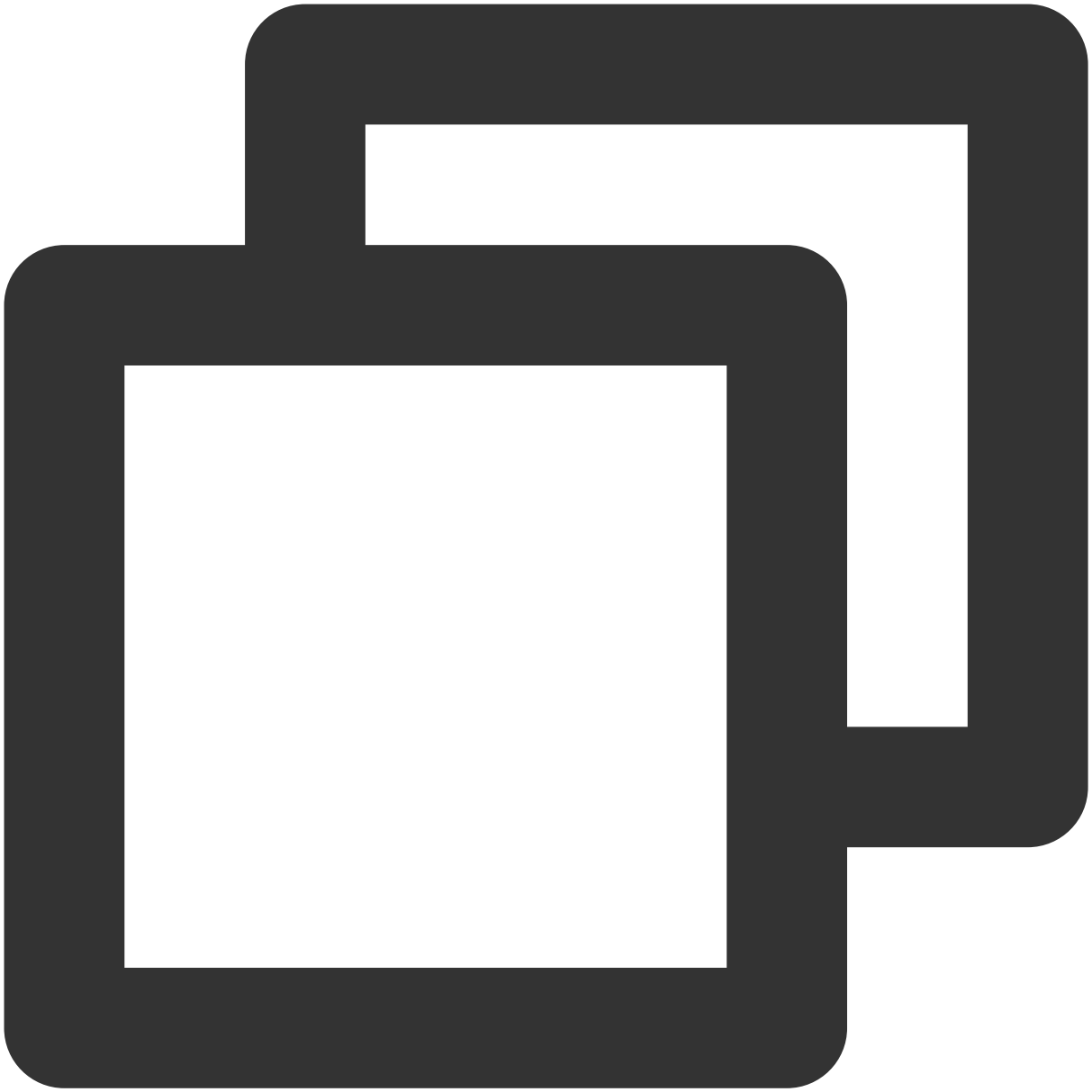
Example:



```
> SELECT find_in_set('ab','abc,b,ab,c,def');
 3
```

FORMAT_NUMBER

Function statement:



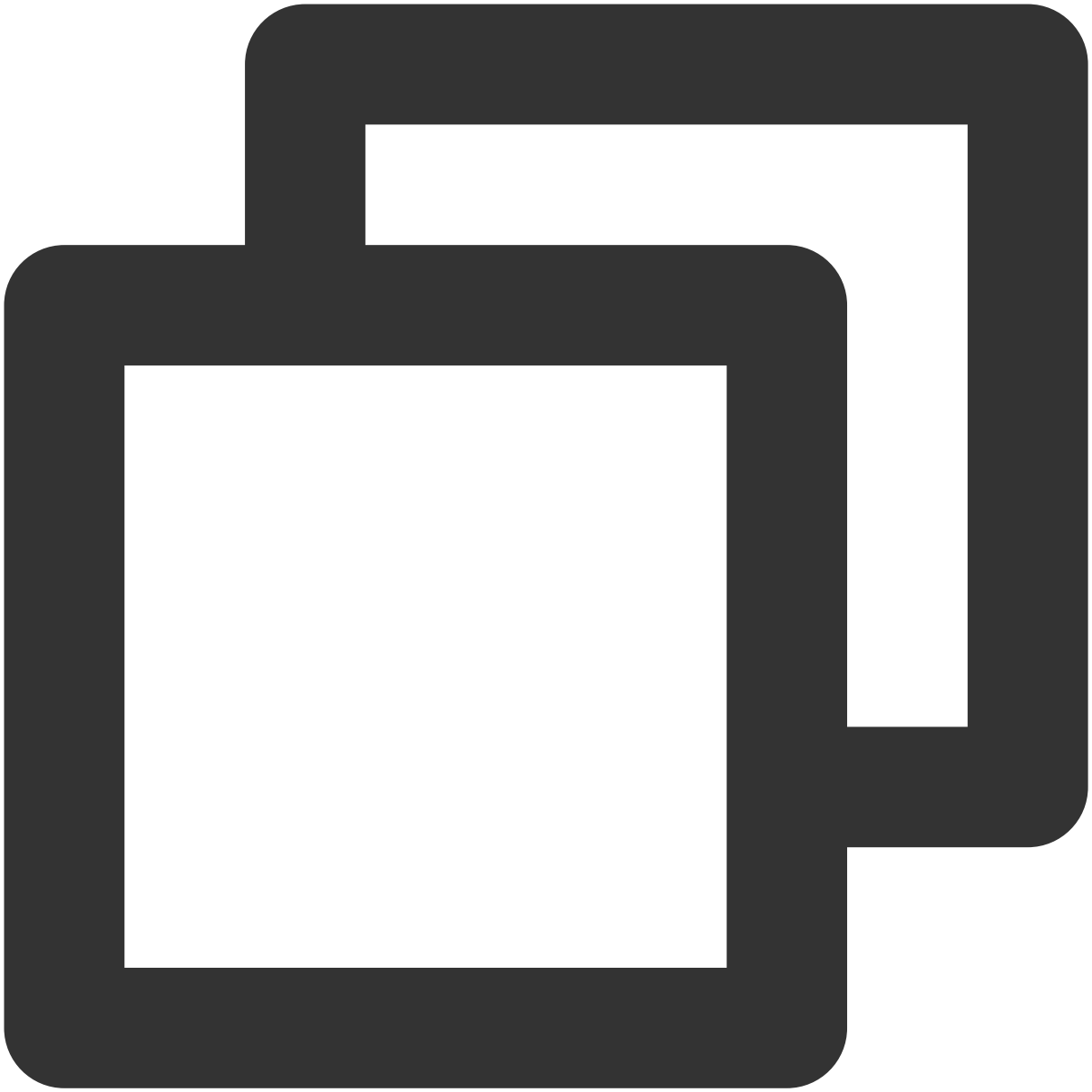
```
FORMAT_NUMBER(<expr1> integer|double|decimal, <expr2> string|integer)
```

Supported engines: SparkSQL and Presto

Usage instructions: Format expr1 as '#,####,####.##' and round to expr2 decimal places. If expr2 is 0, the result will have no decimal point or decimal part.

Return type: string

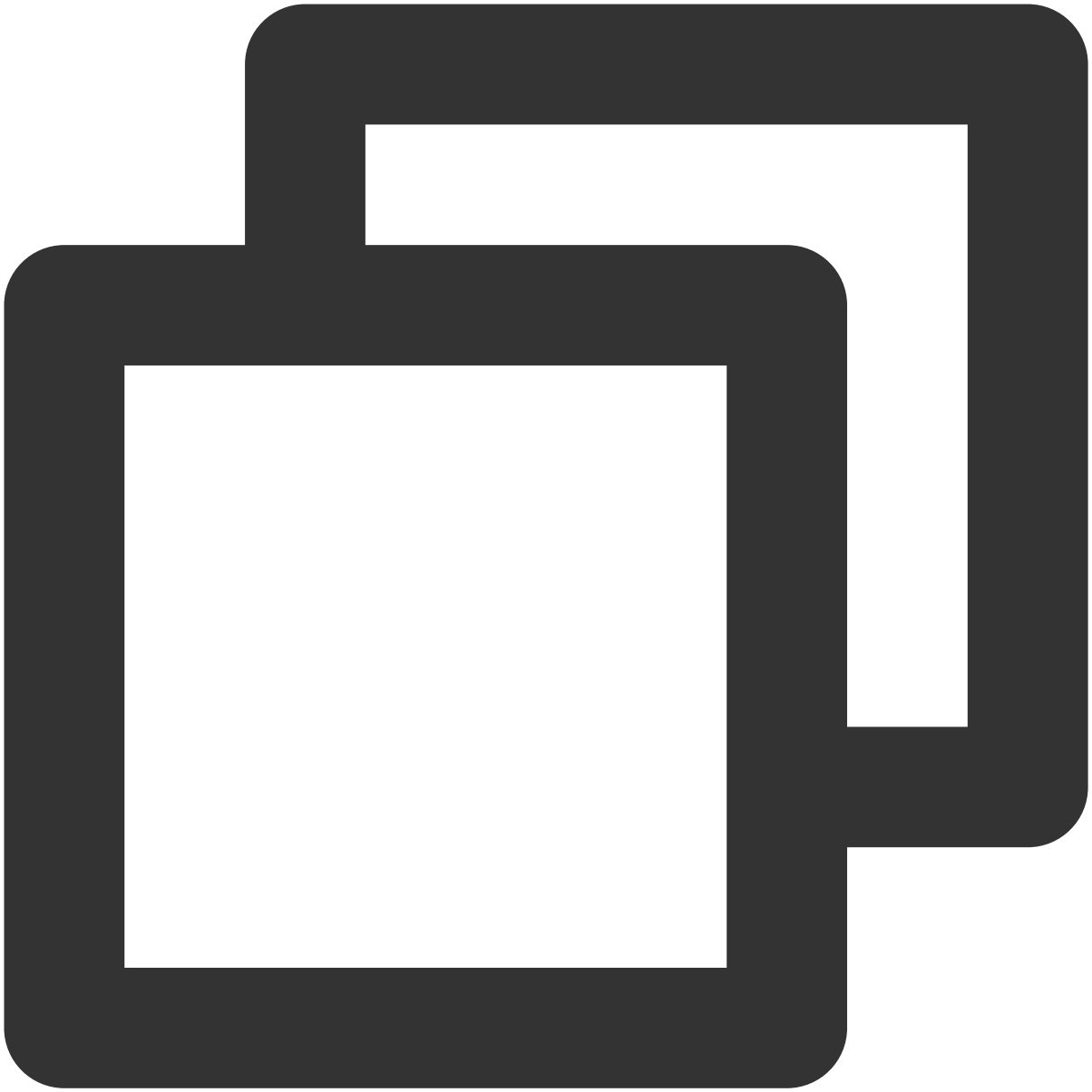
Example:



```
> SELECT format_number(12332.123456, 4);
 12,332.1235
> SELECT format_number(12332.123456, '##################.###');
 12332.123
```

FORMAT_STRING

Function statement:



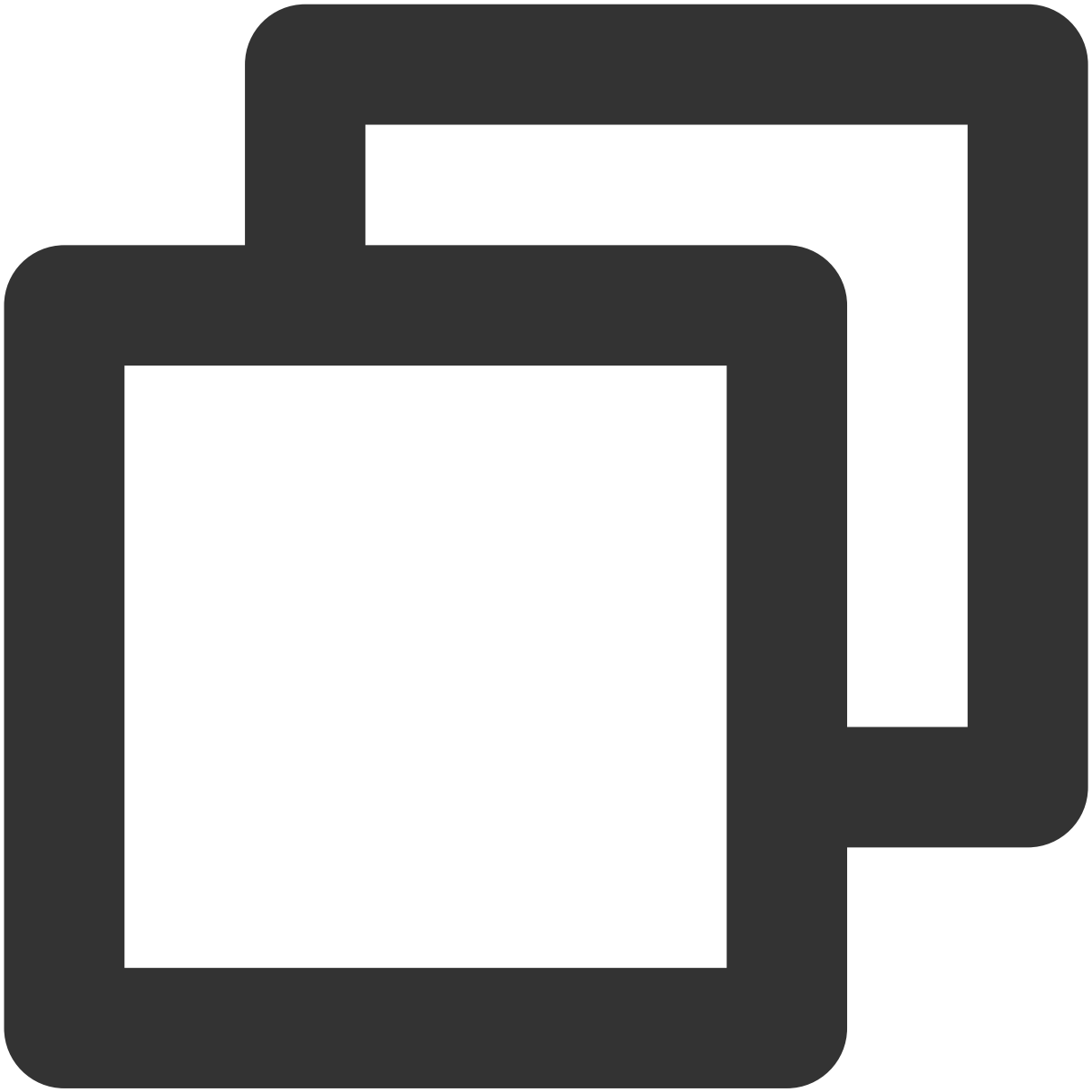
```
FORMAT_STRING(<str> string, obj <T>, ...)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the formatted string from a printf-style format string.

Return type: string

Example:



```
> SELECT format_string("Hello World %d %s", 100, "days");
 Hello World 100 days
```

INITCAP

Function statement:

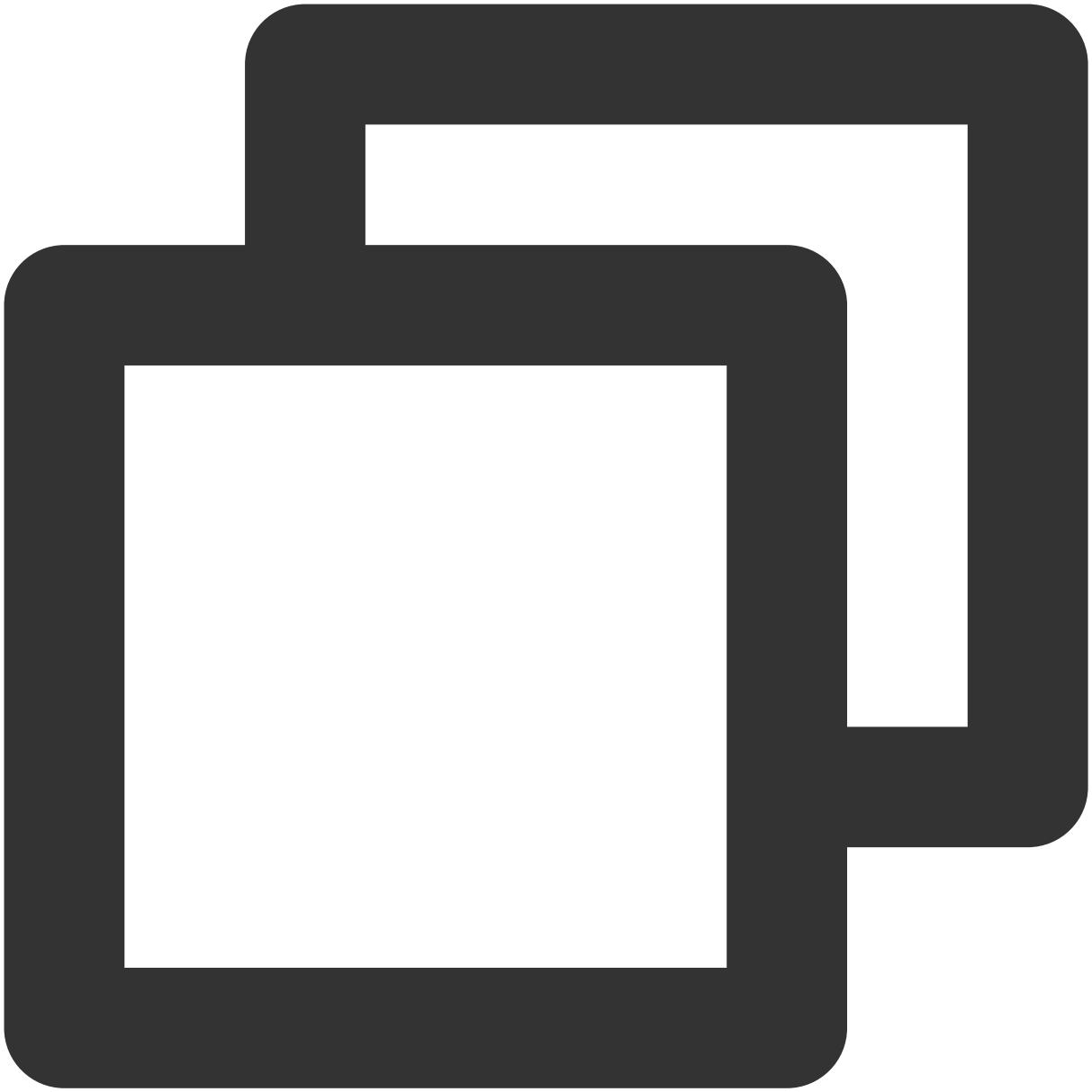
```
INITCAP(<str> string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Capitalize the first letter of each word, with all other letters in lowercase.

Return type: string

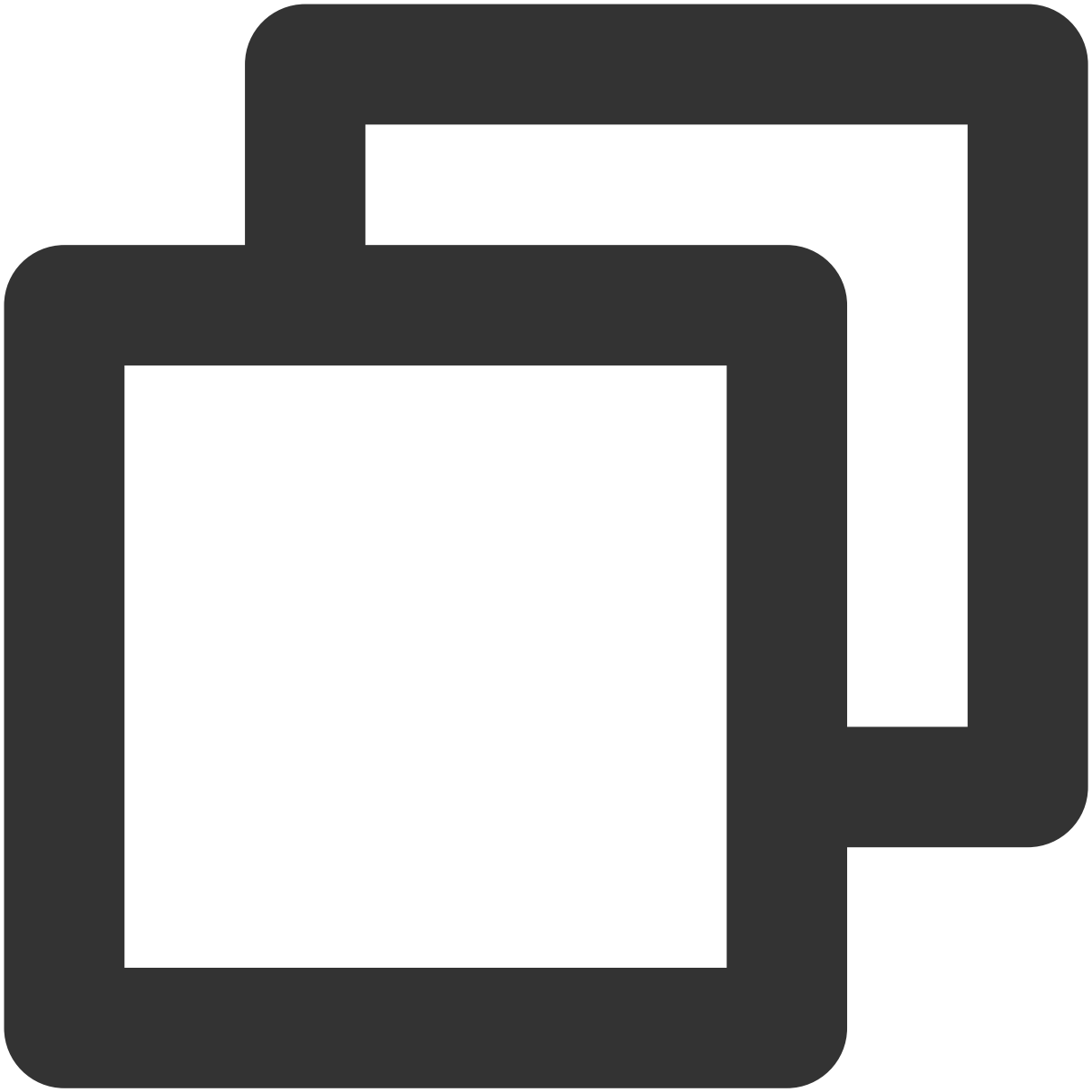
Example:



```
> SELECT initcap('sPark sql');
 Spark Sql
```

INSTR

Function statement:



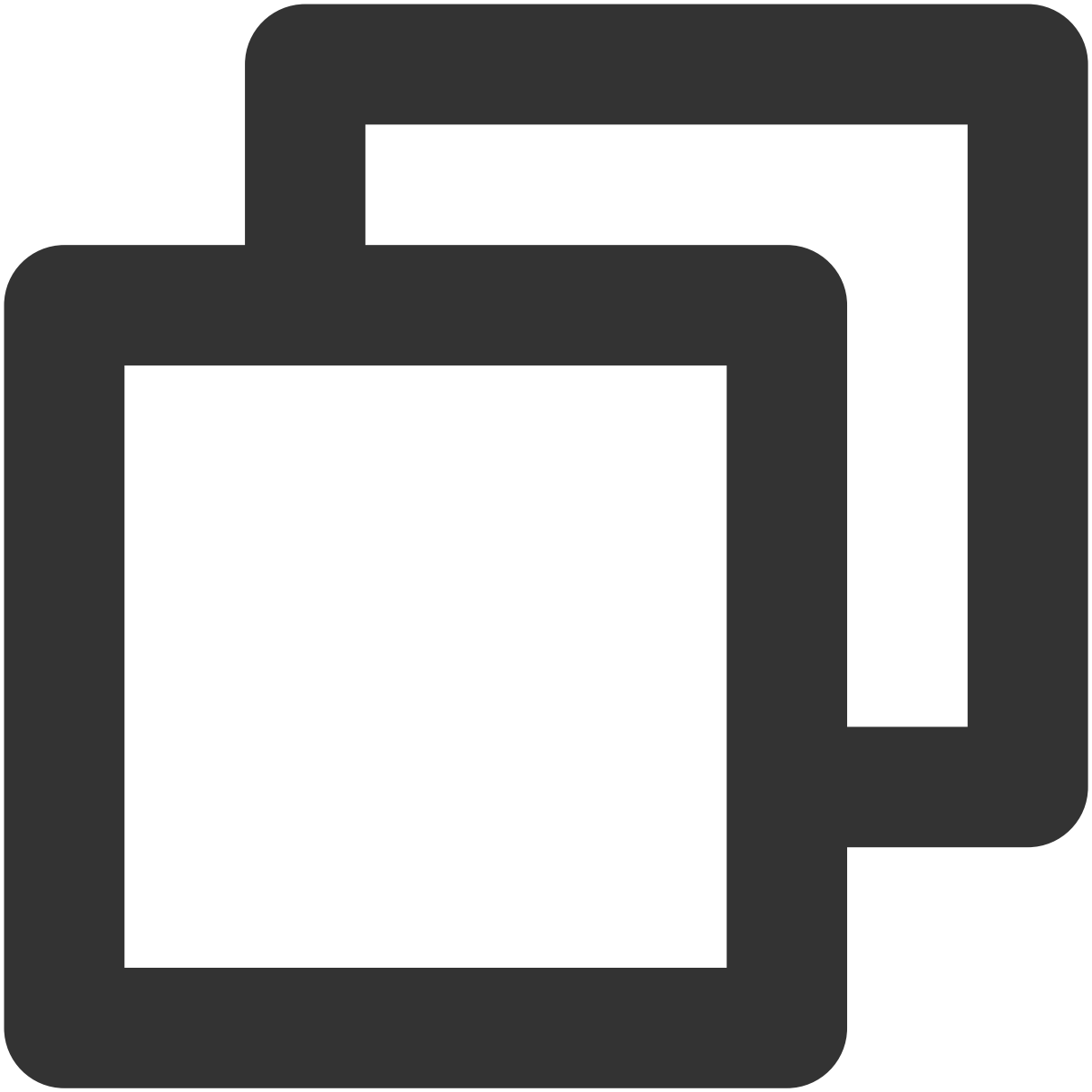
```
INSTR(<str> string, <substr> string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the index (starting from 1) of the first occurrence of substr in str.

Return type: integer

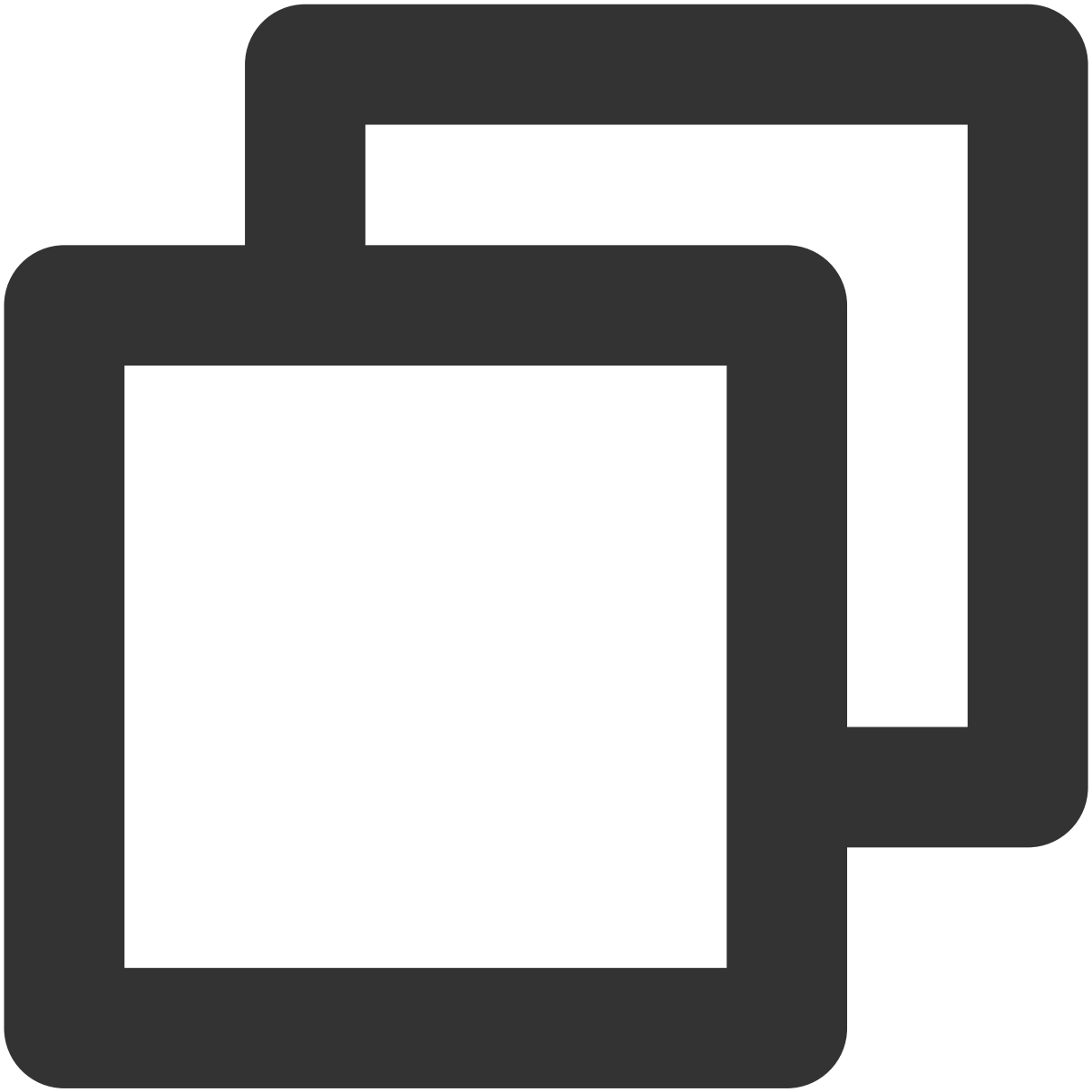
Example:



```
> SELECT instr('SparkSQL', 'SQL');
 6
```

LCASE

Function statement:



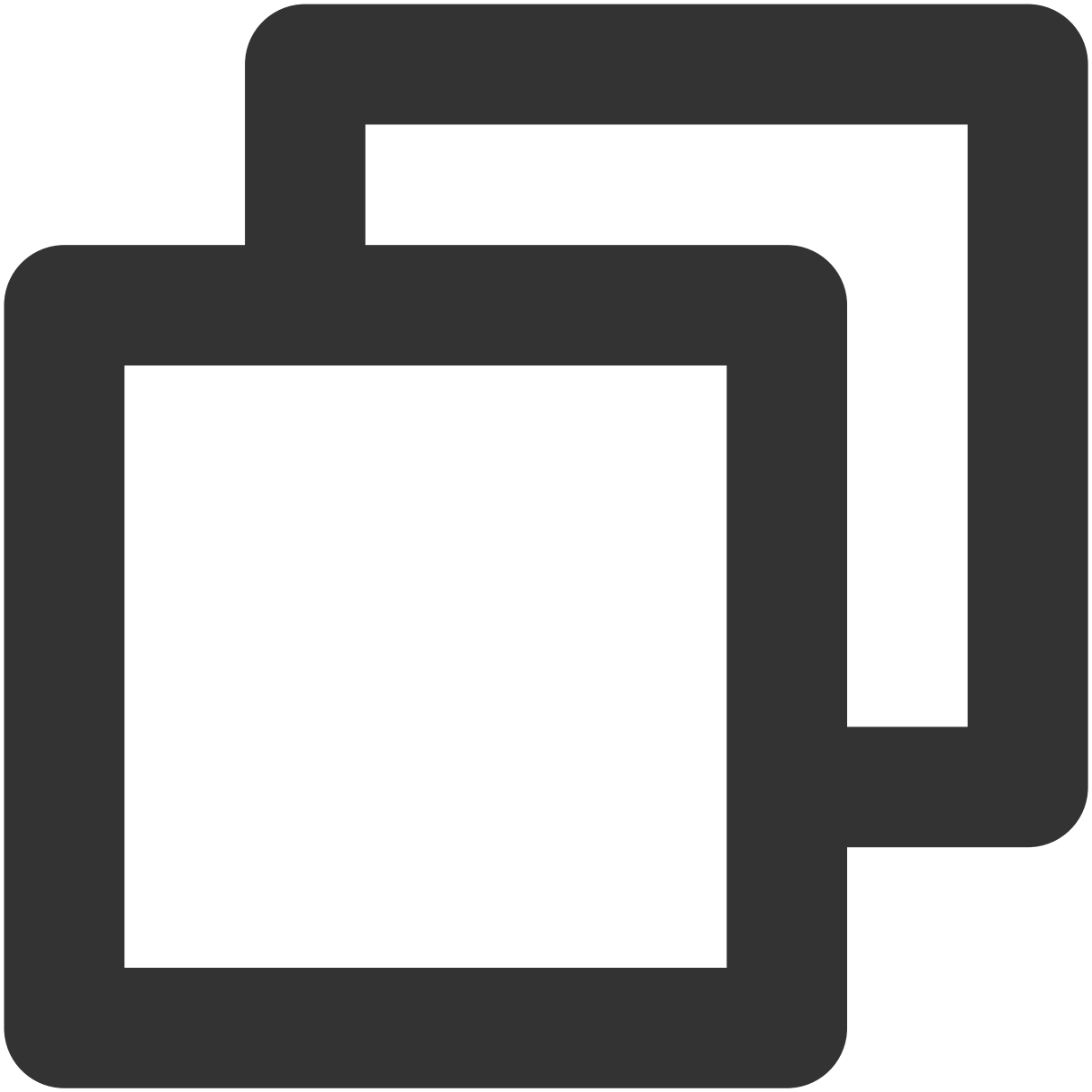
```
LCASE(<str> string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Convert str to lowercase.

Return type: string

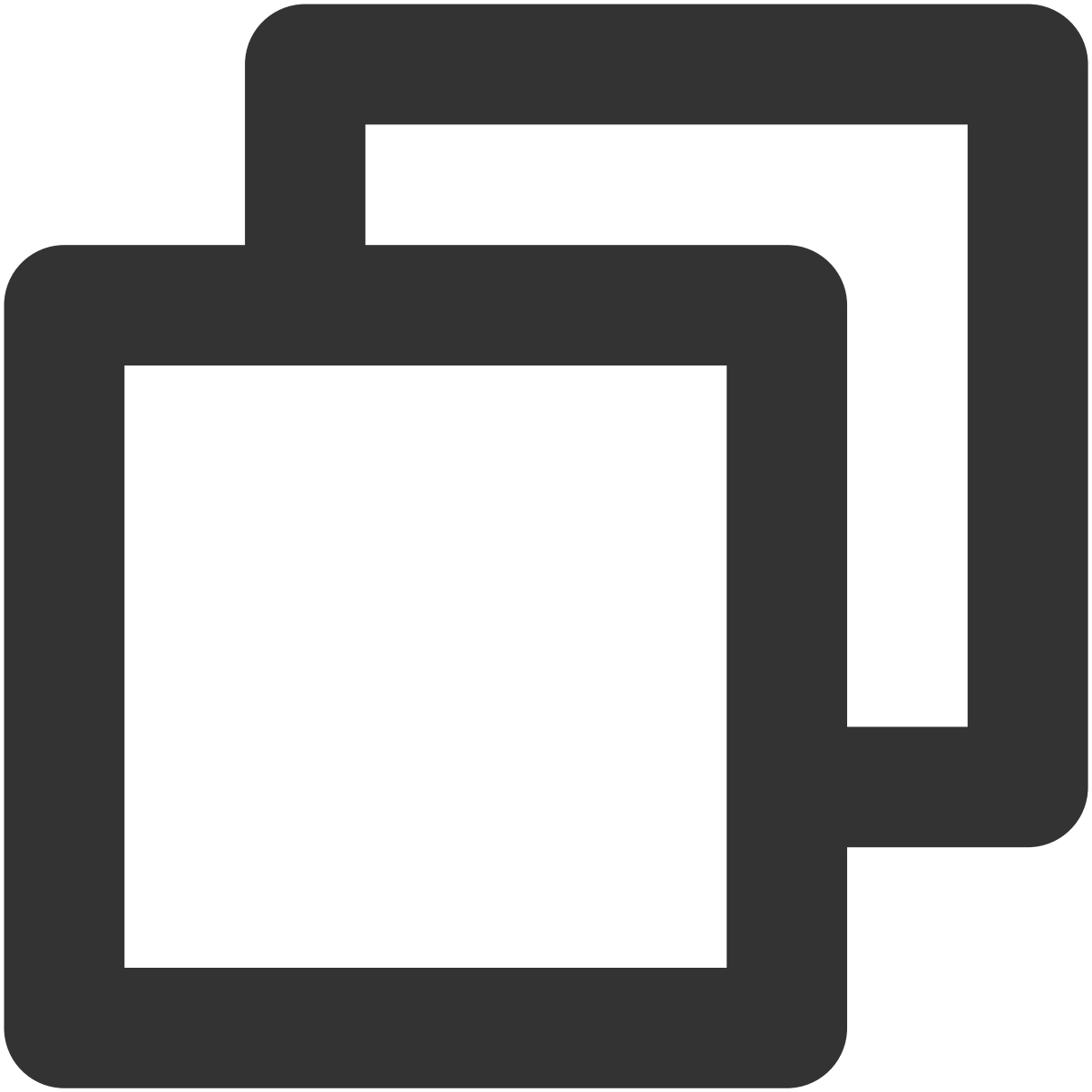
Example:



```
> SELECT lcase('SparkSQL');
 sparksql
```

LENGTH

Function statement:



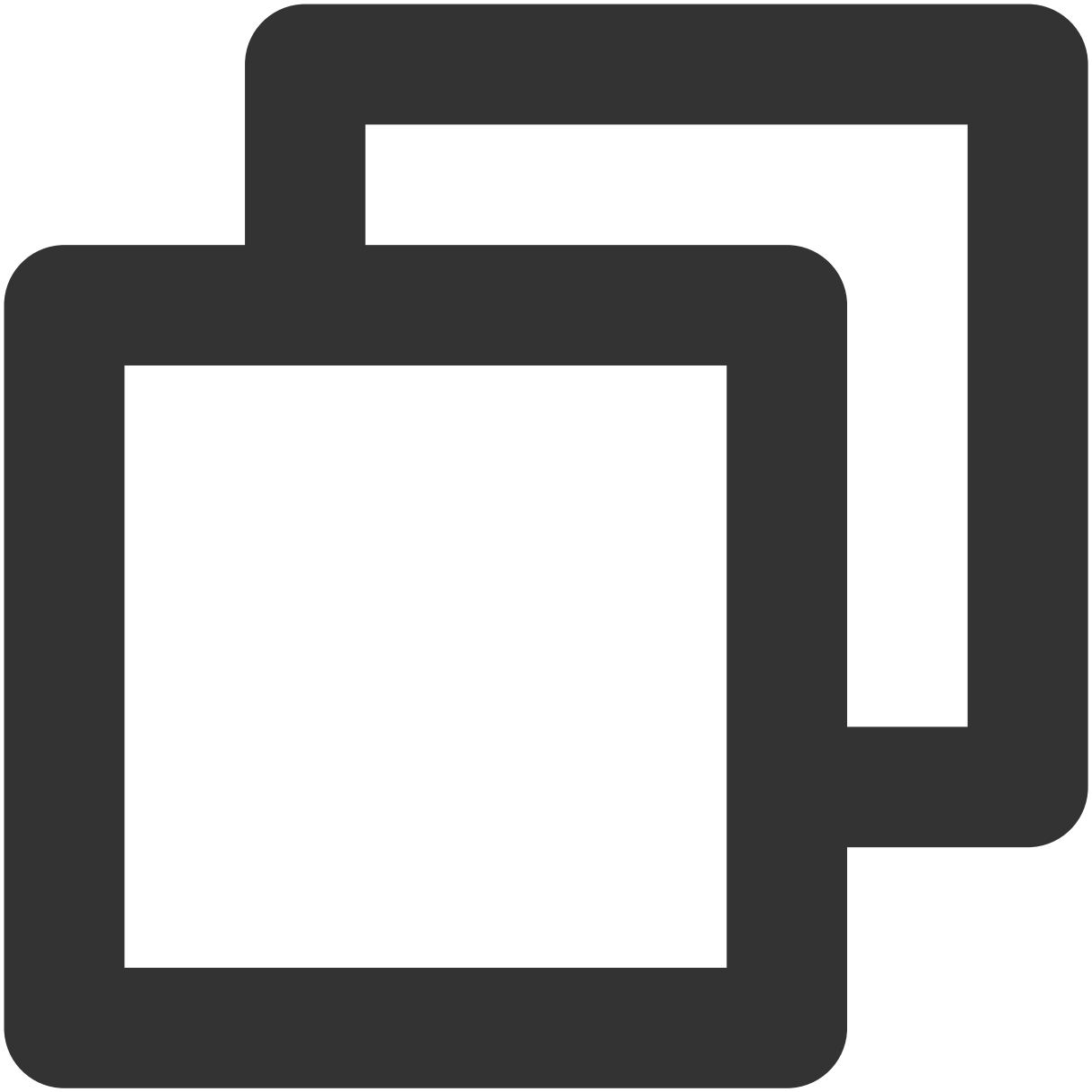
```
LENGTH(<expr> string|binary)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the character length of the string data or the number of bytes in the binary data. The length of the string data includes trailing spaces. The length of the binary data includes binary zeros.

Return type: integer

Example:



```
> SELECT length('Spark SQL ');
 10
```

LEVENSHTEIN

Function statement:

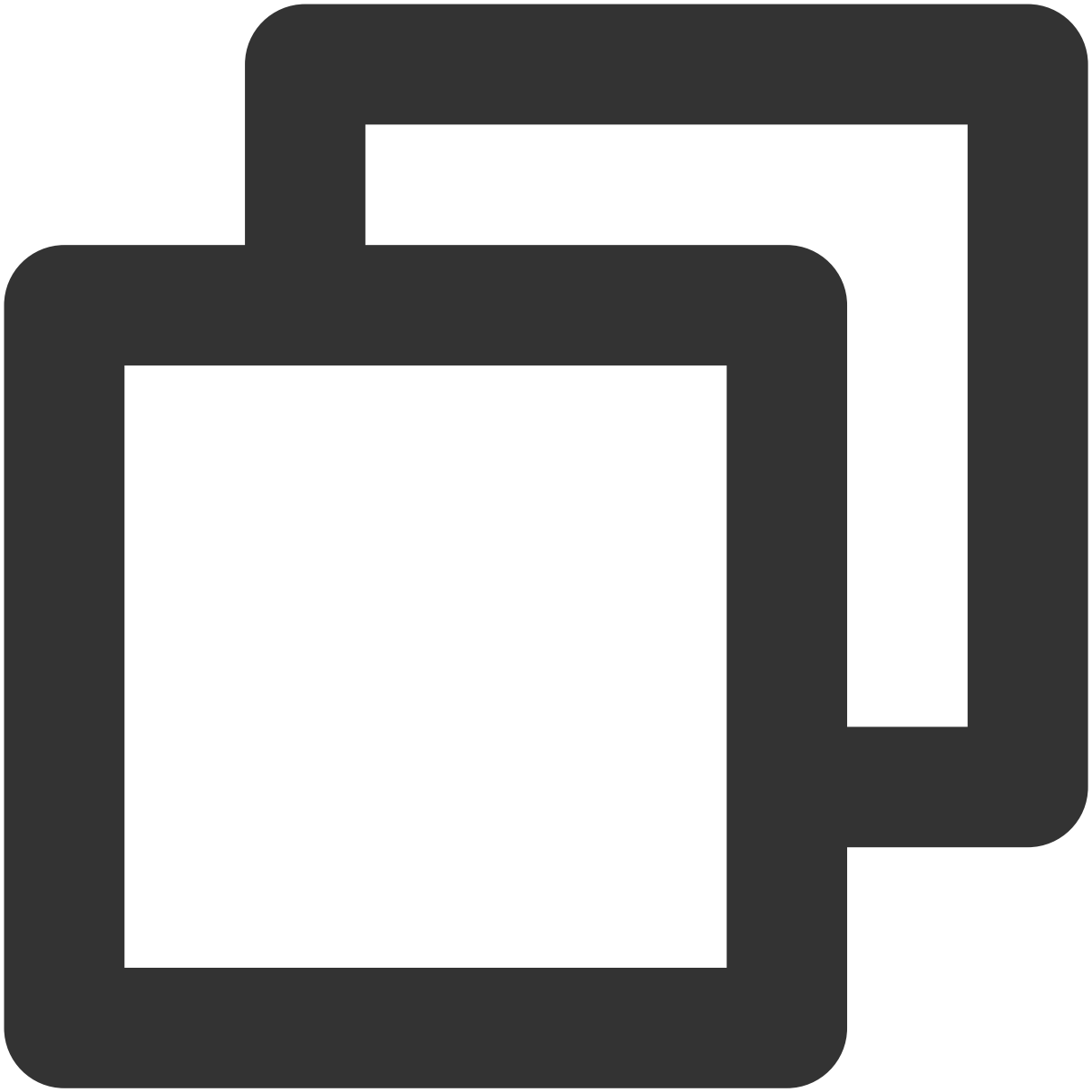
```
LEVENSHTEIN(<s1> string, <s2> string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the Levenshtein distance between the two given strings.

Return type: integer

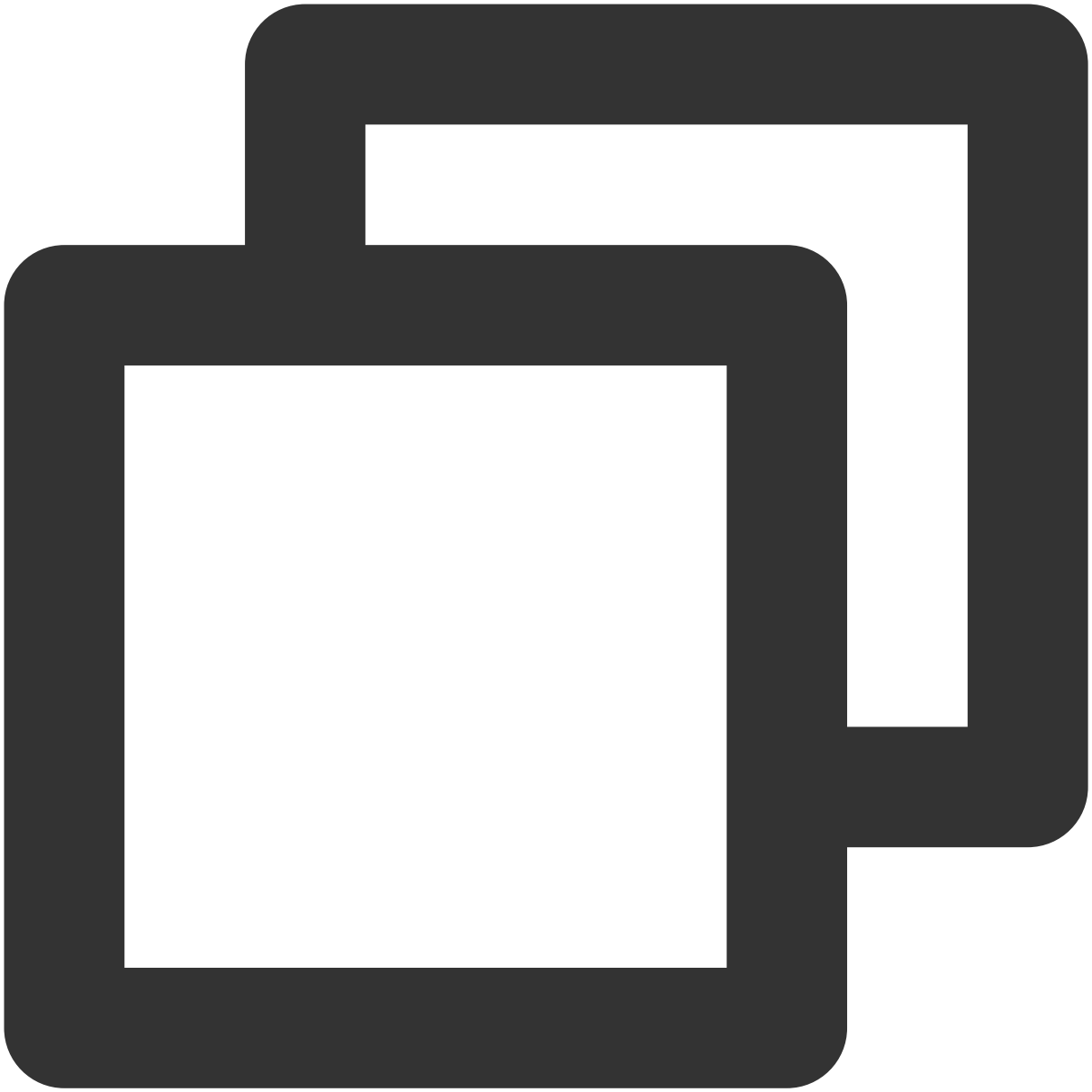
Example:



```
> SELECT levenshtein('kitten', 'sitting');
 3
```

LIKE

Function statement:



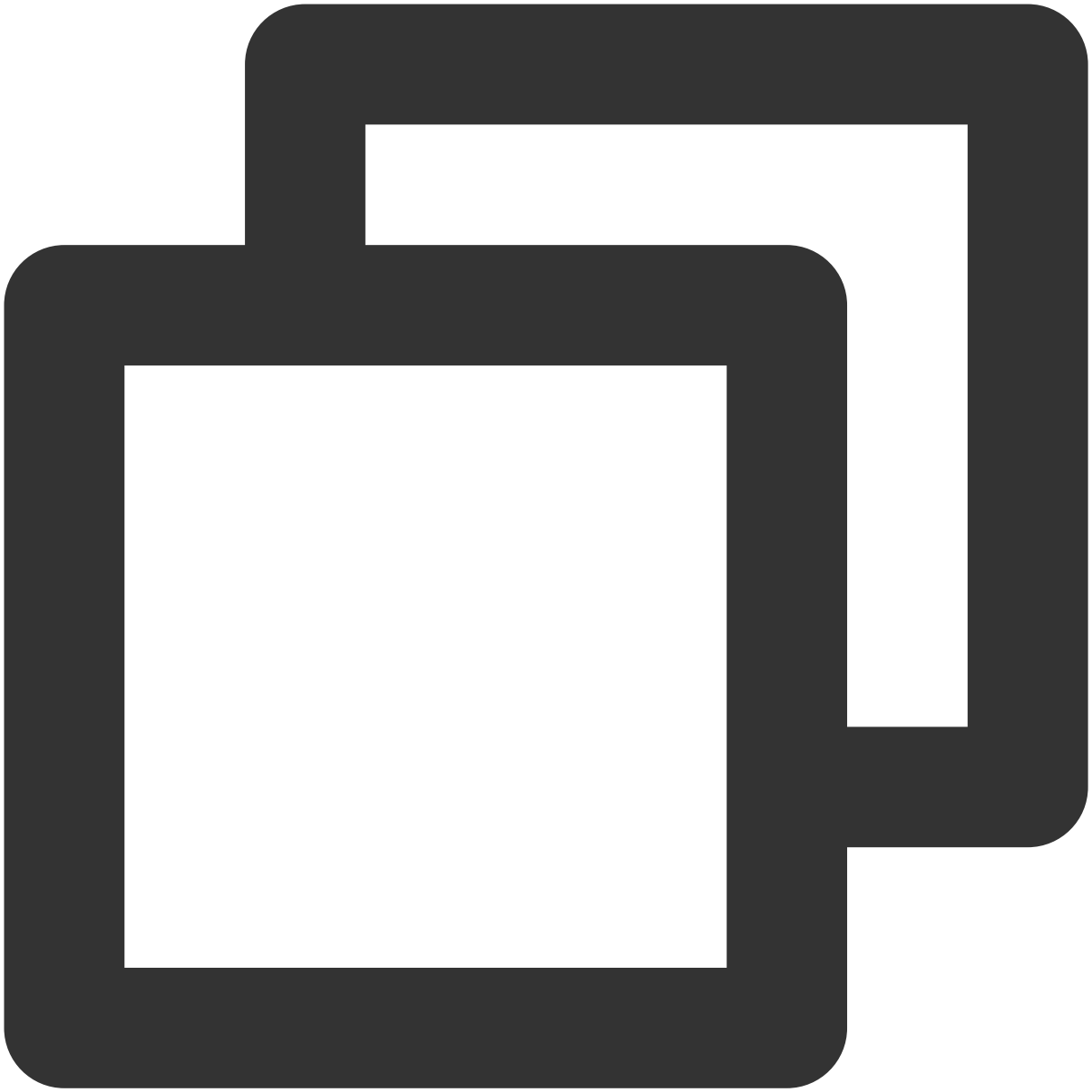
```
LIKE(<s1> str, <s2> pattern)
<str> like <pattern>[ ESCAPE <escape>]
```

Supported engines: SparkSQL and Presto

Usage instructions: Return true if str matches the pattern with the escape character; return null if any parameter is null; otherwise, return false.

Return type: boolean

Example:



```
> SELECT like('Spark', '_park');
true
> SET spark.sql.parser.escapedStringLiterals=true;
spark.sql.parser.escapedStringLiterals  true
> SELECT '%SystemDrive%\\Users\\John' like '\\%SystemDrive\\%\\\\Users%';
true
> SET spark.sql.parser.escapedStringLiterals=false;
spark.sql.parser.escapedStringLiterals  false
> SELECT '%SystemDrive%\\\\Users\\\\John' like '\\%SystemDrive\\%\\\\\\\\Users%';
false
> SELECT '%SystemDrive%/Users/John' like '/%SystemDrive/%//Users%' ESCAPE '/';
```

```
true
```

LOWER

Function statement:

```
LOWER(<str> string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return str with all characters converted to lowercase.

Return type: string

Example:

```
> SELECT lower('TENCENT');
 tencent
```

LOCATE

Function statement:

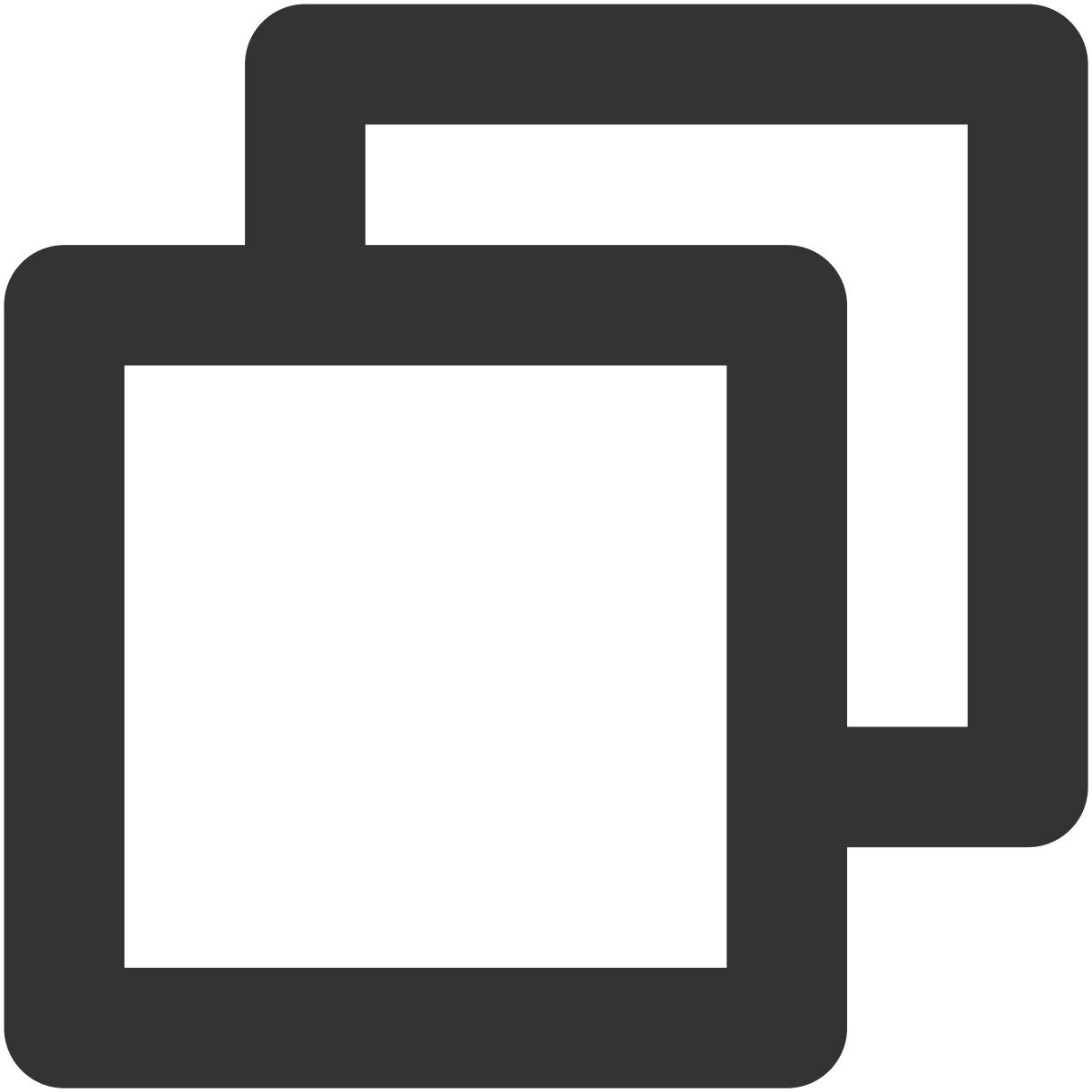
```
LOCATE(<substr> string, <str> string[, <pos> integer])
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the position of the first occurrence of substr in str after the pos position.

Return type: integer

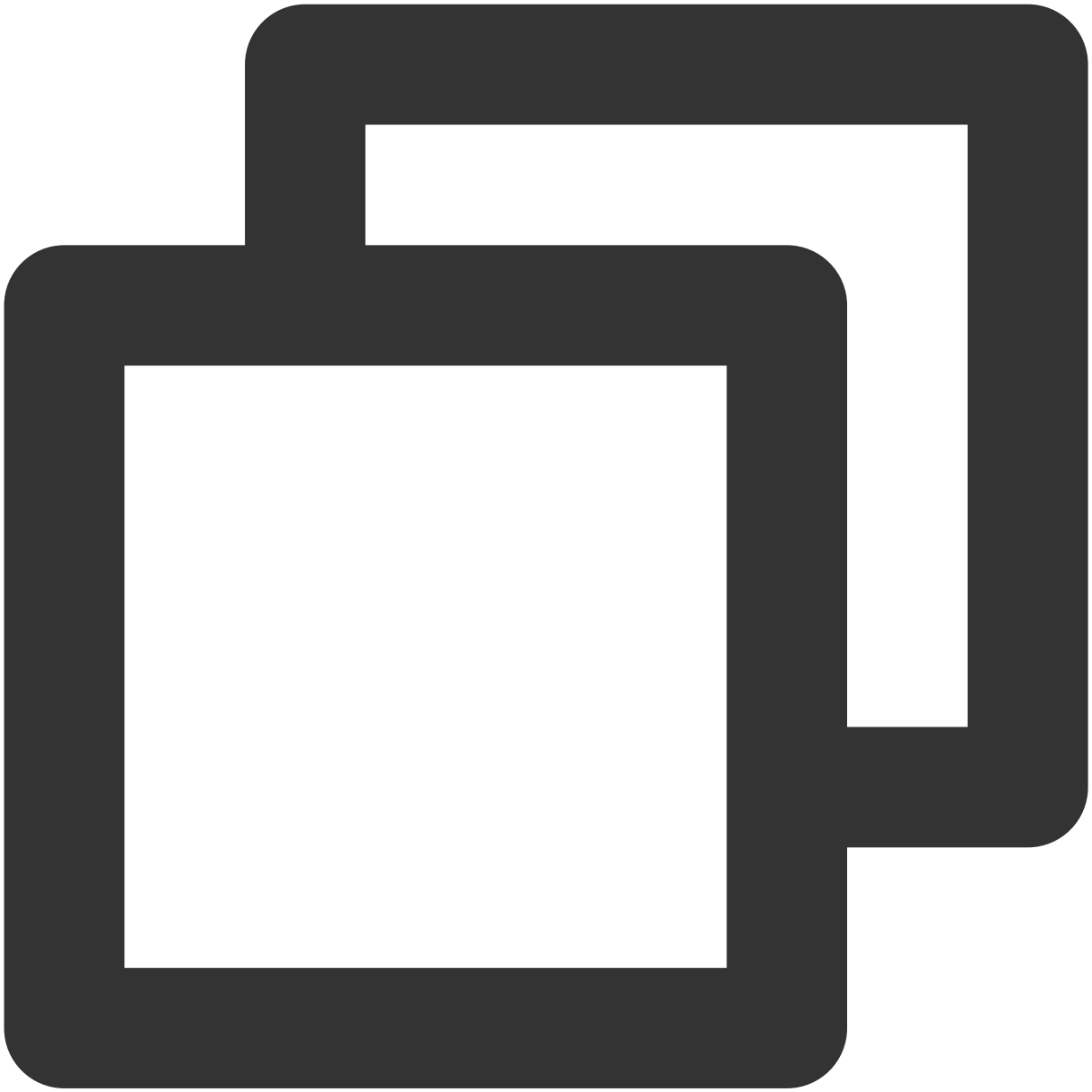
Example:



```
> SELECT locate('bar', 'foobarbar');
 4
> SELECT locate('bar', 'foobarbar', 5);
 7
```

OCTET_LENGTH

Function statement:



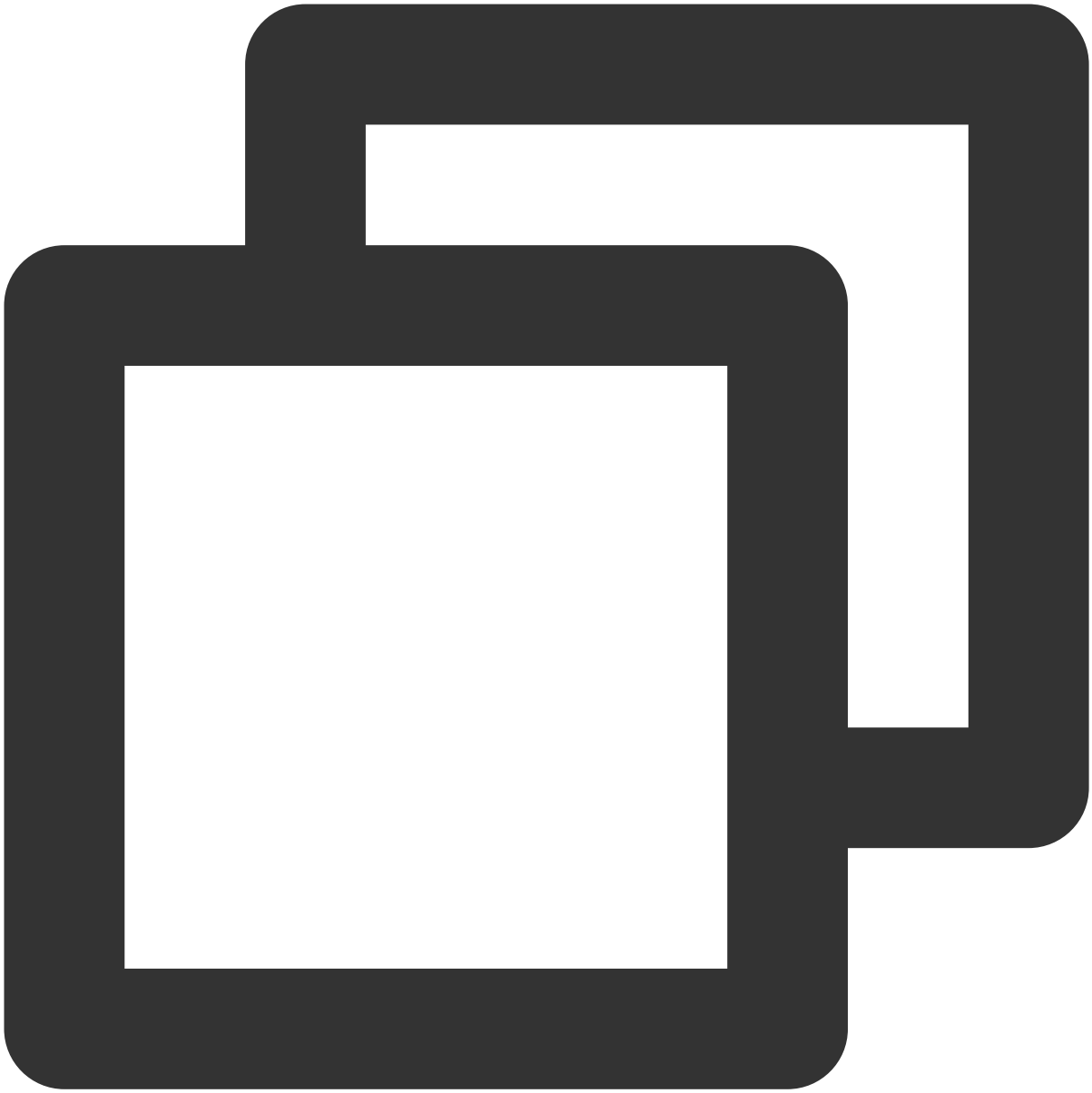
```
OCTET_LENGTH(<expr> string|binary)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the character length of the string data or the number of bytes in the binary data.

Return type: integer

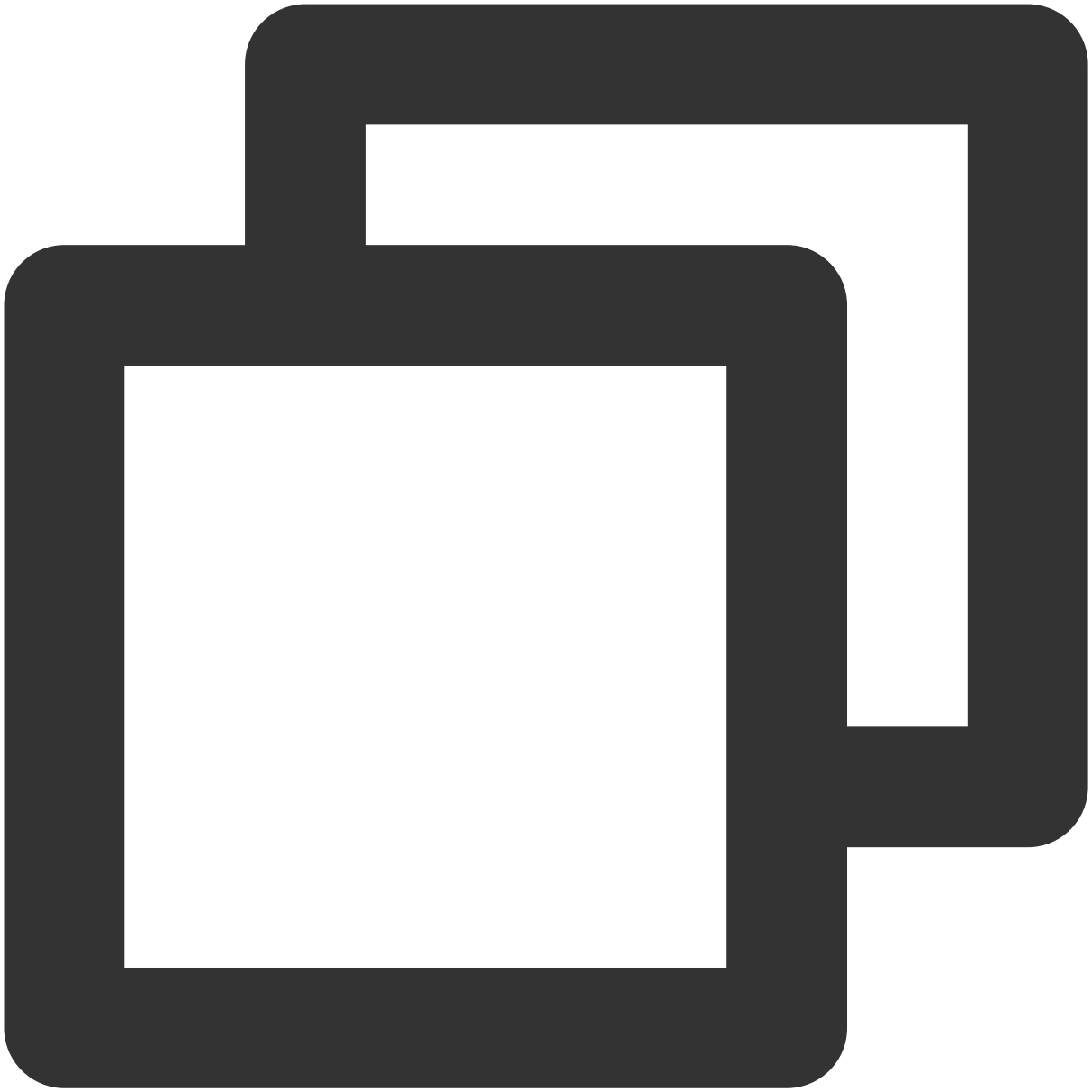
Example:



```
> SELECT octet_length('Spark SQL');
 9
```

LPAD

Function statement:



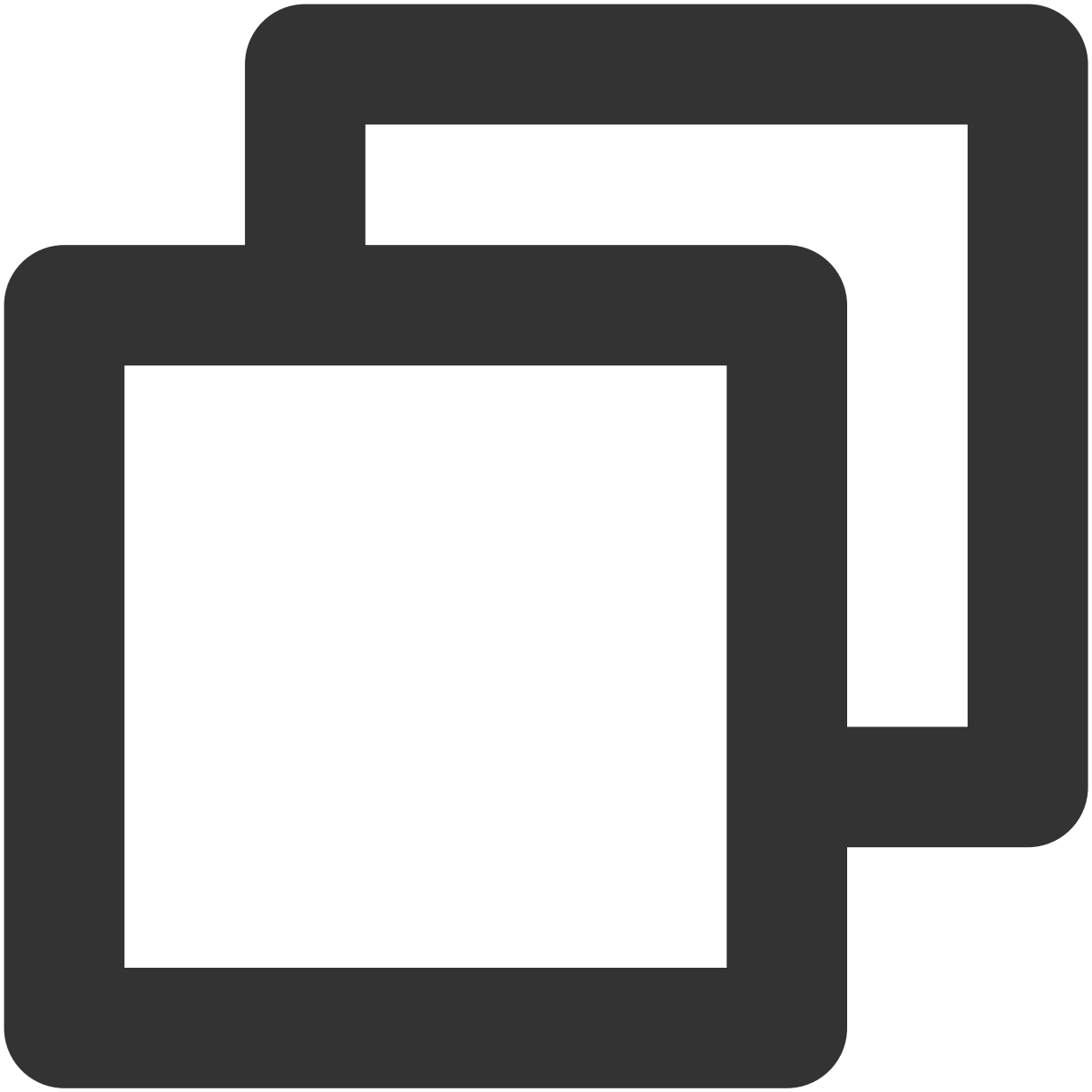
```
LPAD(<str> string, <len> integer[, <pad> string])
```

Supported engines: SparkSQL and Presto

Usage instructions: Return str, left-padded with pad to a length of len. If str is longer than len, the returned value is truncated to len characters. If pad is not specified, str will be padded with space characters.

Return type: string

Example:



```
> SELECT lpad('hi', 5, '??');
 ???hi
> SELECT lpad('hi', 1, '??');
 h
> SELECT lpad('hi', 5);
  hi
```

LTRIM

Function statement:

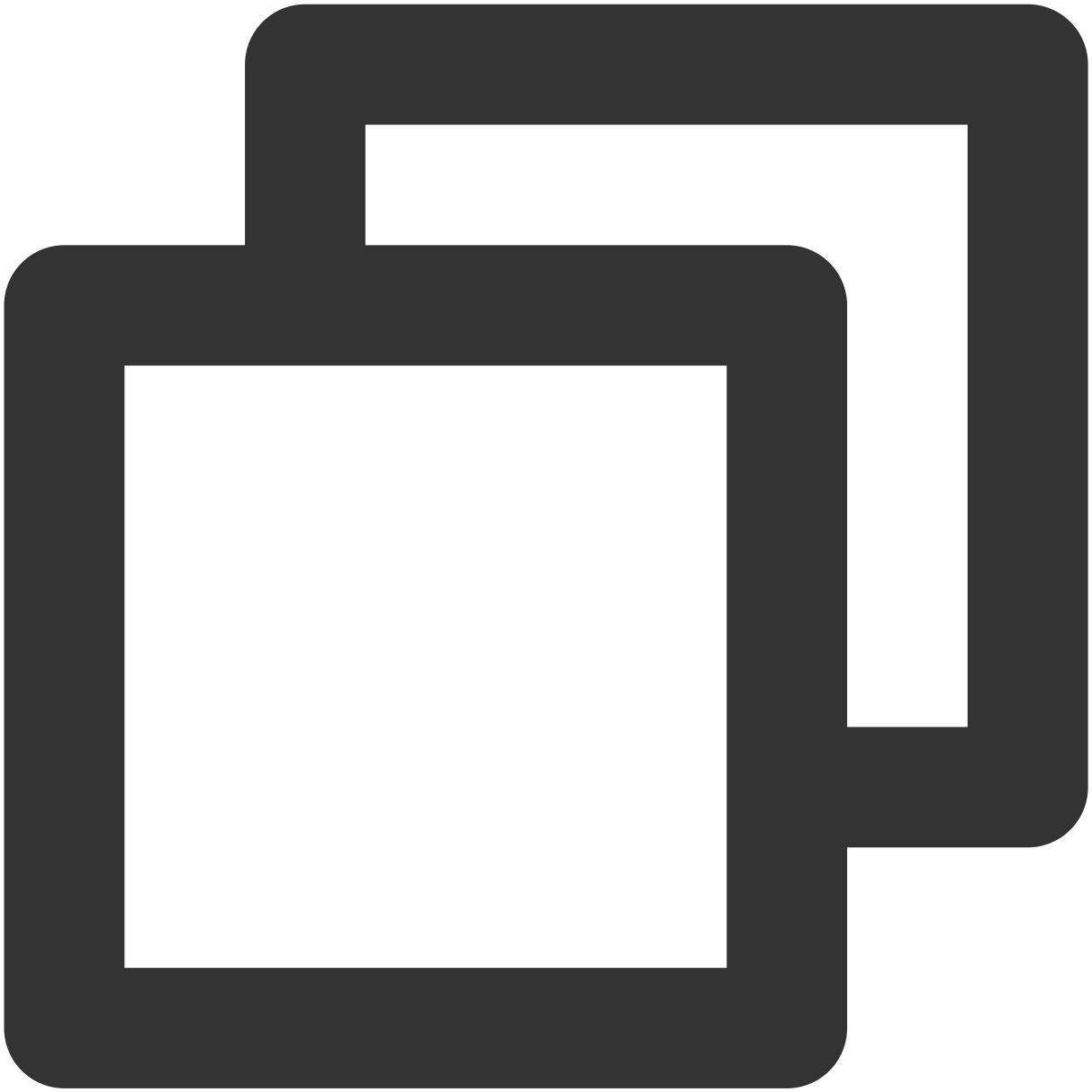
```
LTRIM(<str> string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Remove leading space characters from str.

Return type: string

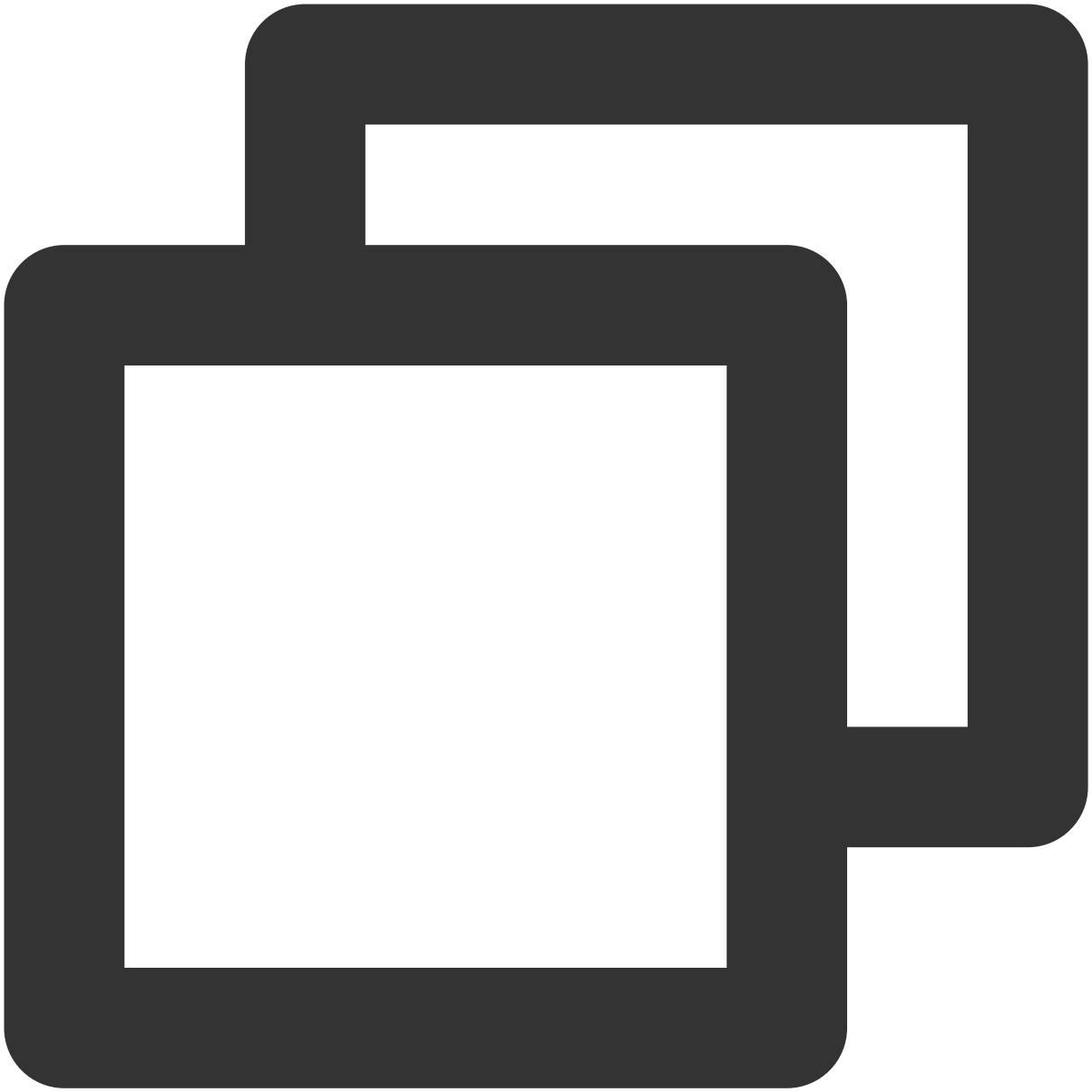
Example:



```
> SELECT ltrim('    SparkSQL   ');
 SparkSQL
```

PARSE_URL

Function statement:



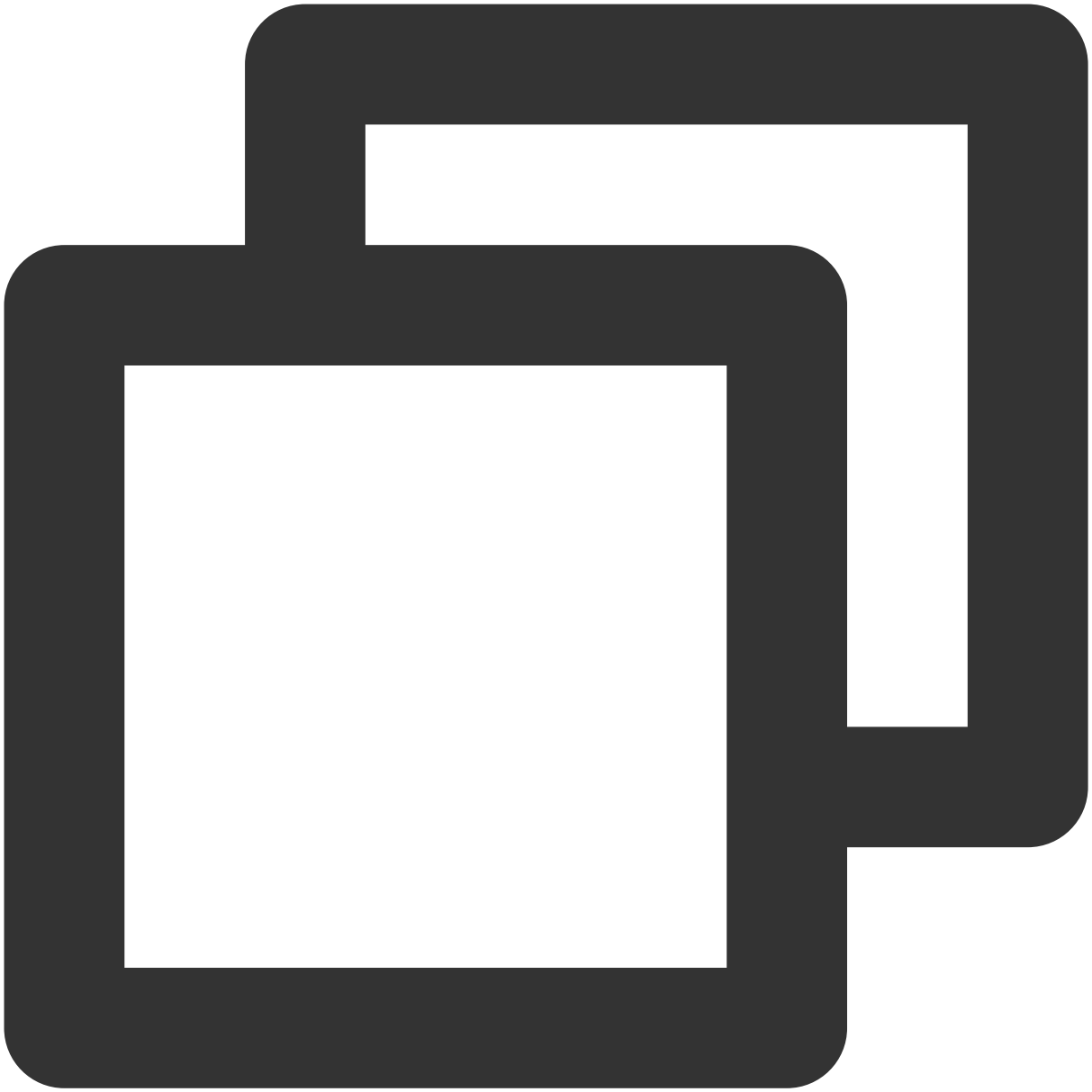
```
PARSE_URL(<url> string, <path> string[, <key> string])
```

Supported engines: SparkSQL and Presto

Usage instructions: Extract the path from the URL.

Return type: string

Example:



```
> SELECT parse_url('http://spark.apache.org/path?query=1', 'HOST');
 spark.apache.org
> SELECT parse_url('http://spark.apache.org/path?query=1', 'QUERY');
 query=1
> SELECT parse_url('http://spark.apache.org/path?query=1', 'QUERY', 'query');
 1
```

POSITION

Function statement:

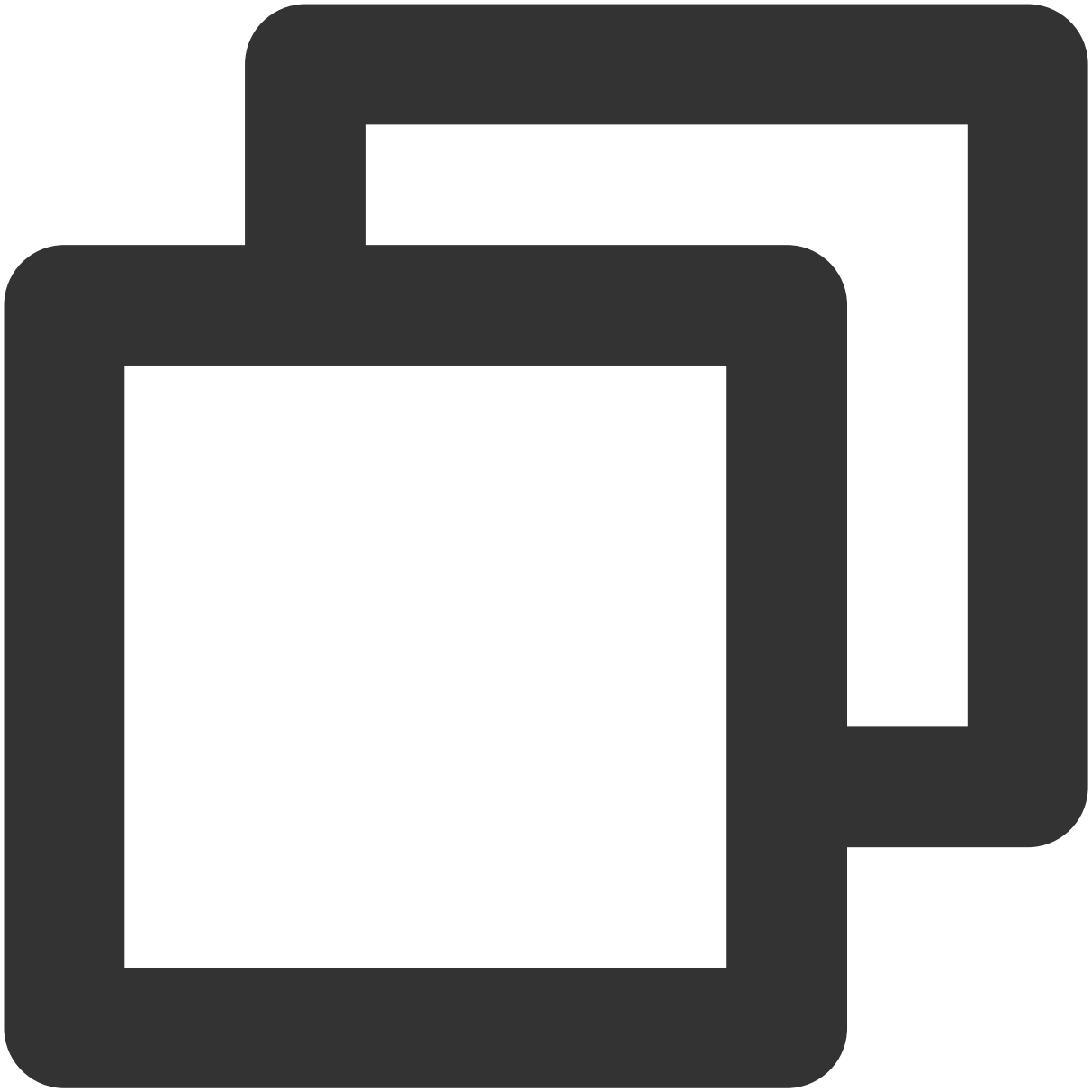
```
POSITION(<substr> string, <str> string[, <pos> integer])
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the position of the first occurrence of substr in str after the pos position.

Return type: integer

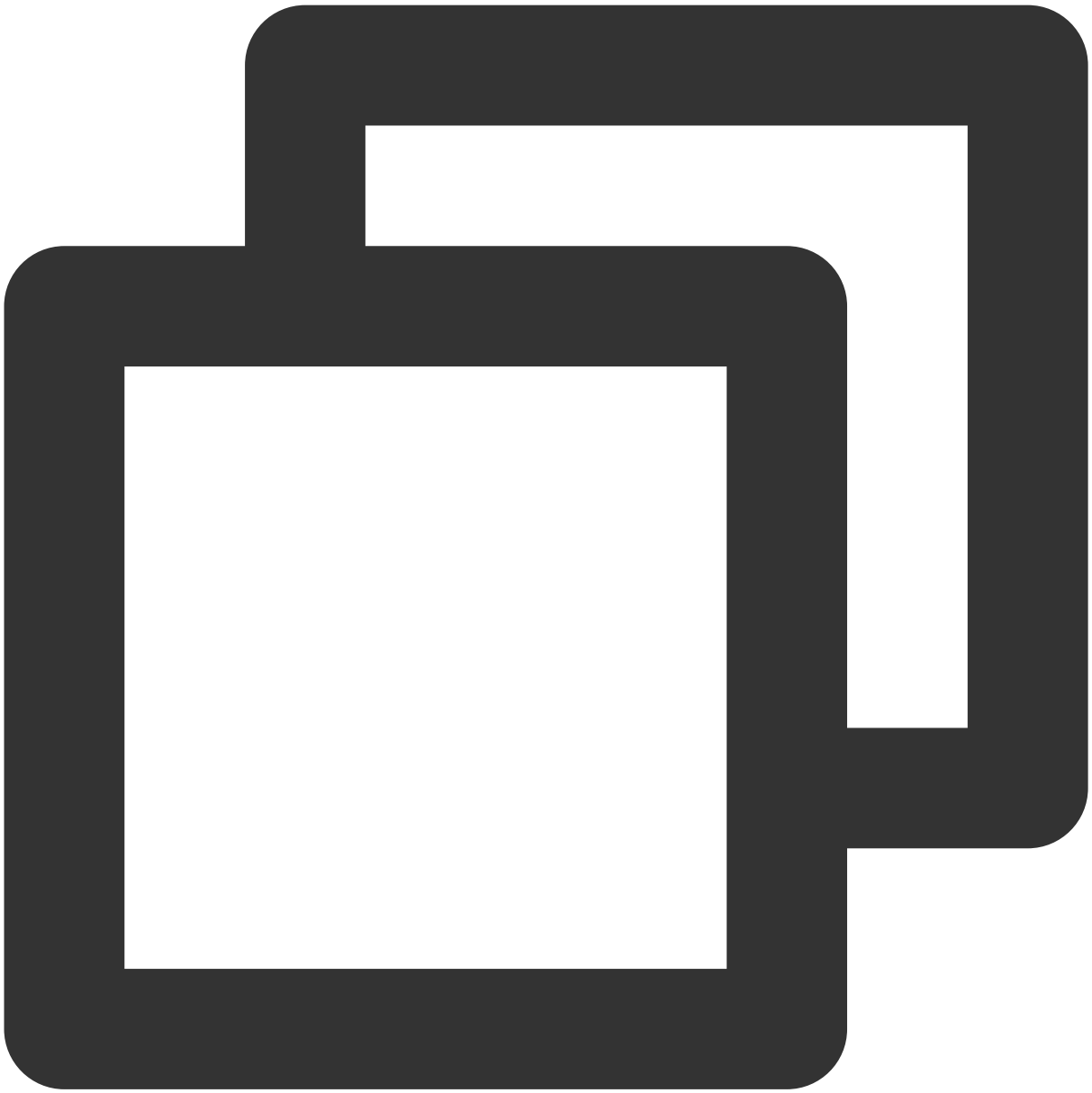
Example:



```
> SELECT position('bar', 'foobarbar');
 4
> SELECT position('bar', 'foobarbar', 5);
 7
> SELECT POSITION('bar' IN 'foobarbar');
 4
```

PRINTF

Function statement:



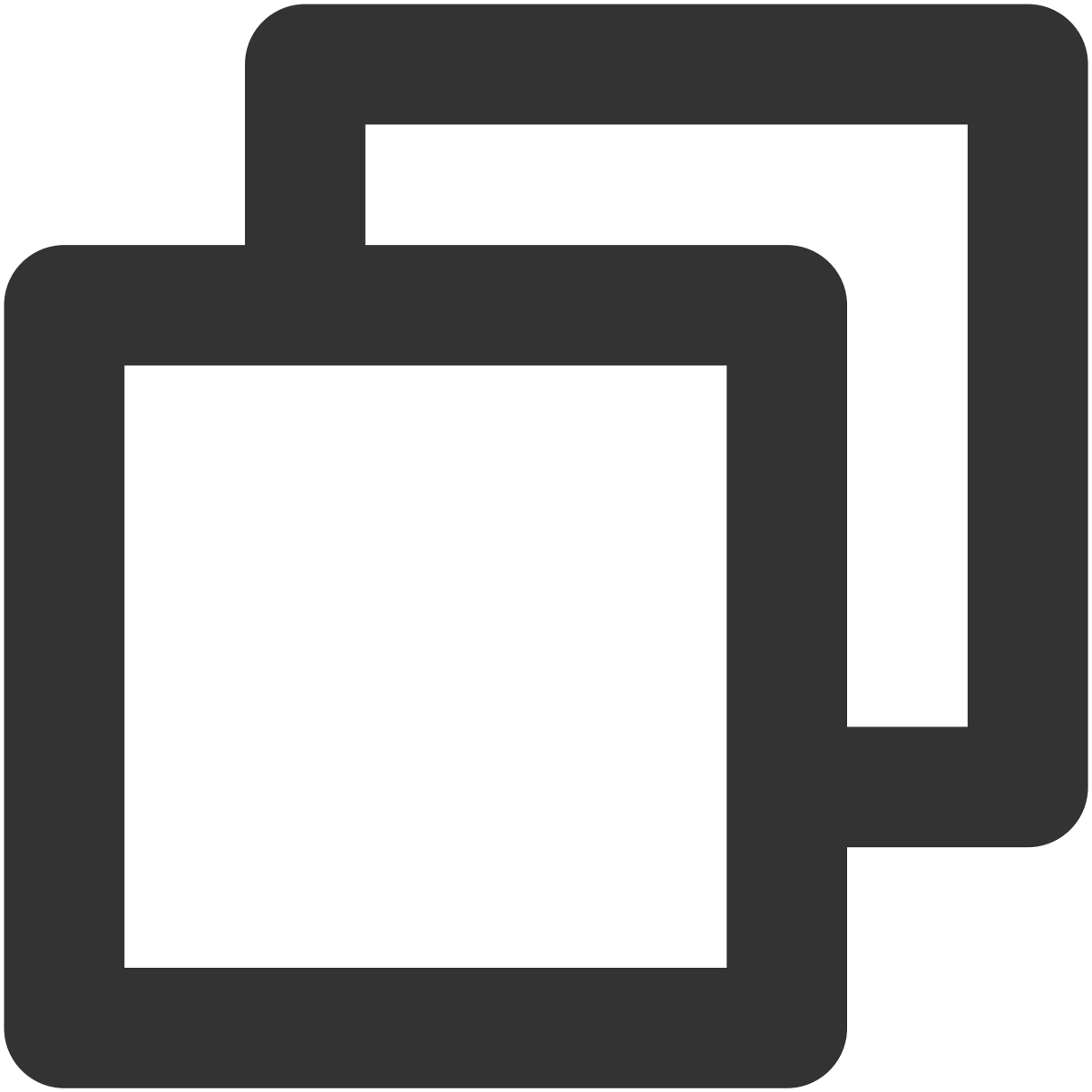
```
PRINTF(<str> string, obj <T>, ...)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the formatted string from a printf-style format string.

Return type: string

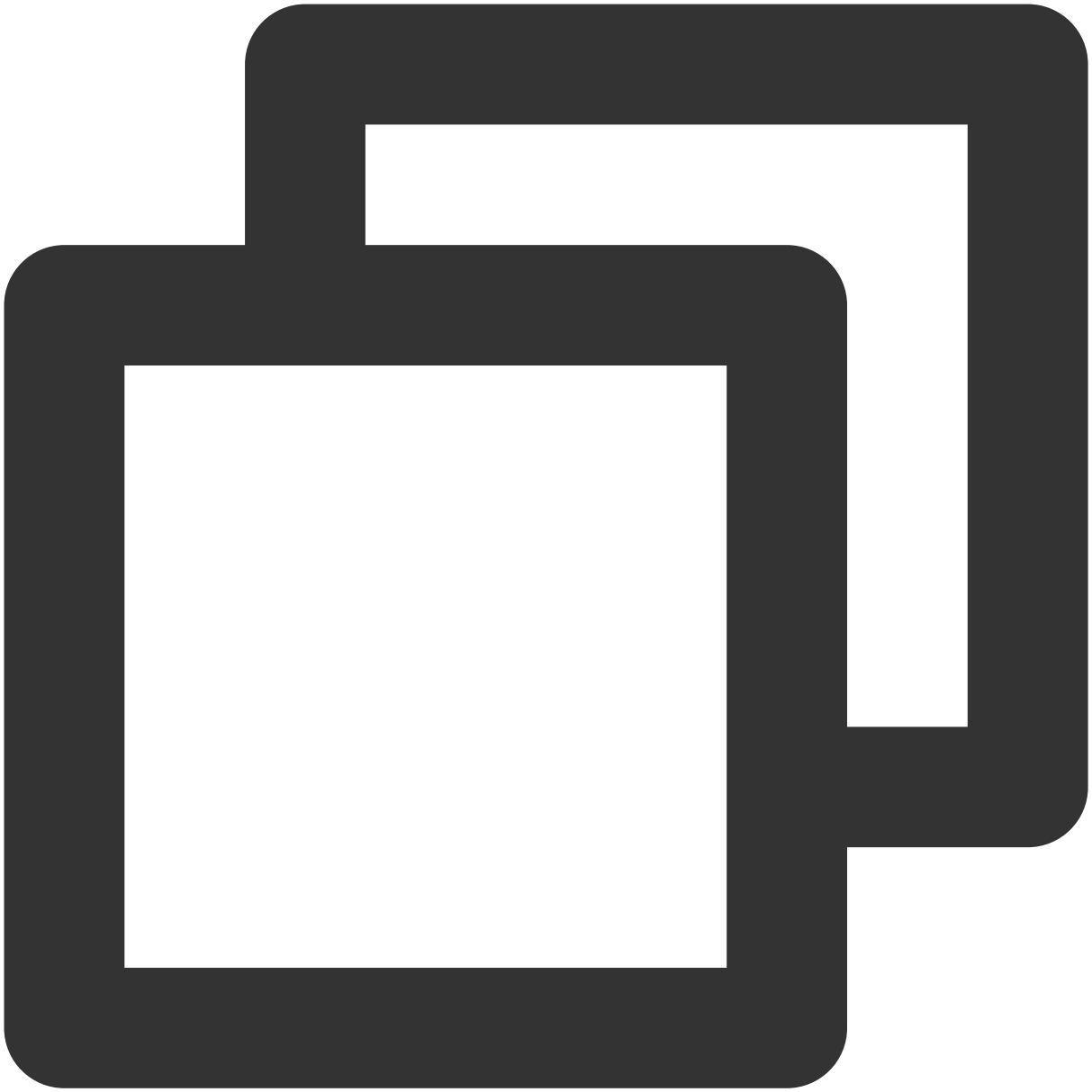
Example:



```
> SELECT printf("Hello World %d %s", 100, "days");
 Hello World 100 days
```

REPEAT

Function statement:



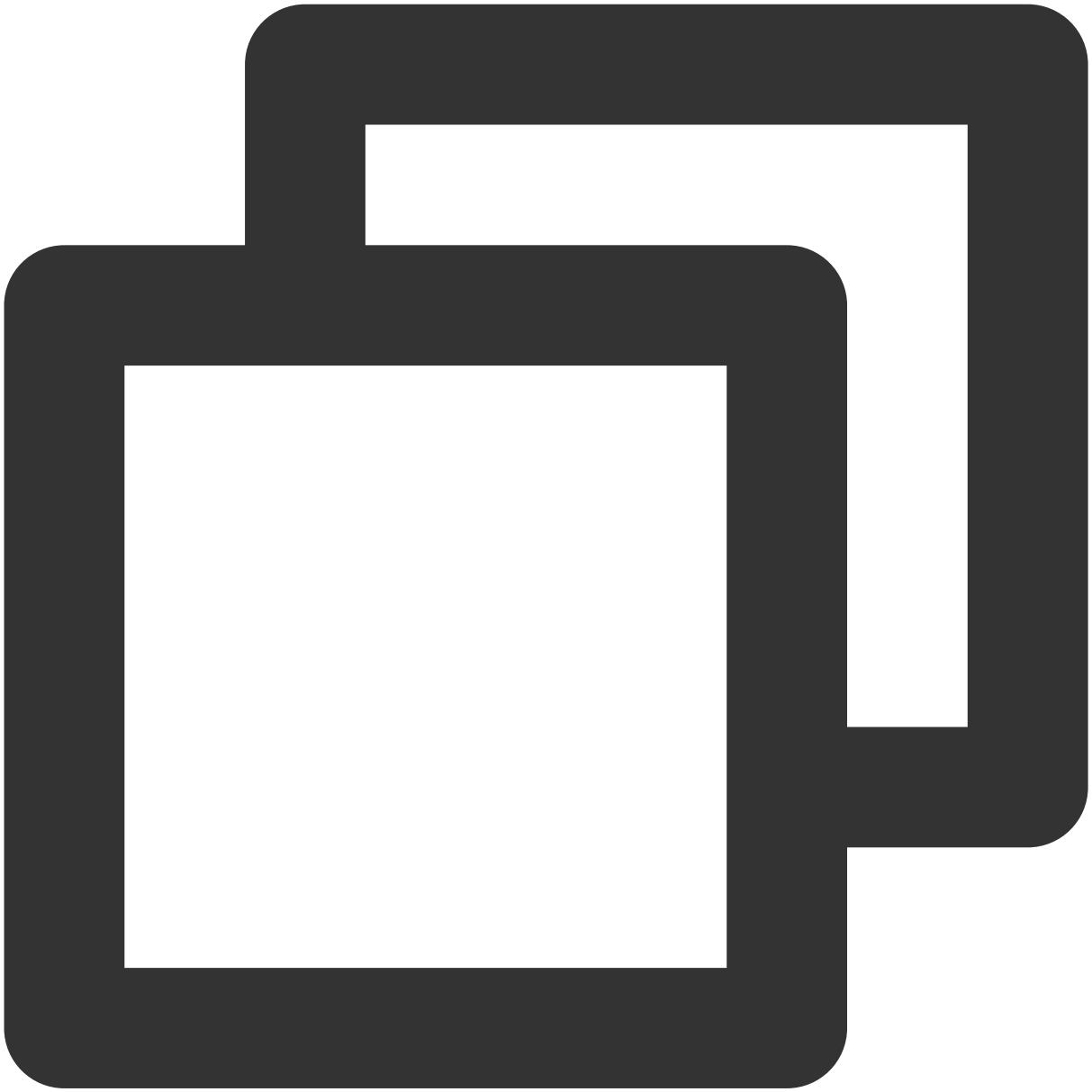
```
REPEAT(<str> string, <n> integer)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return a string that repeats the given string n times.

Return type: string

Example:



```
> SELECT repeat('123', 2);
 123123
```

REPLACE

Function statement:

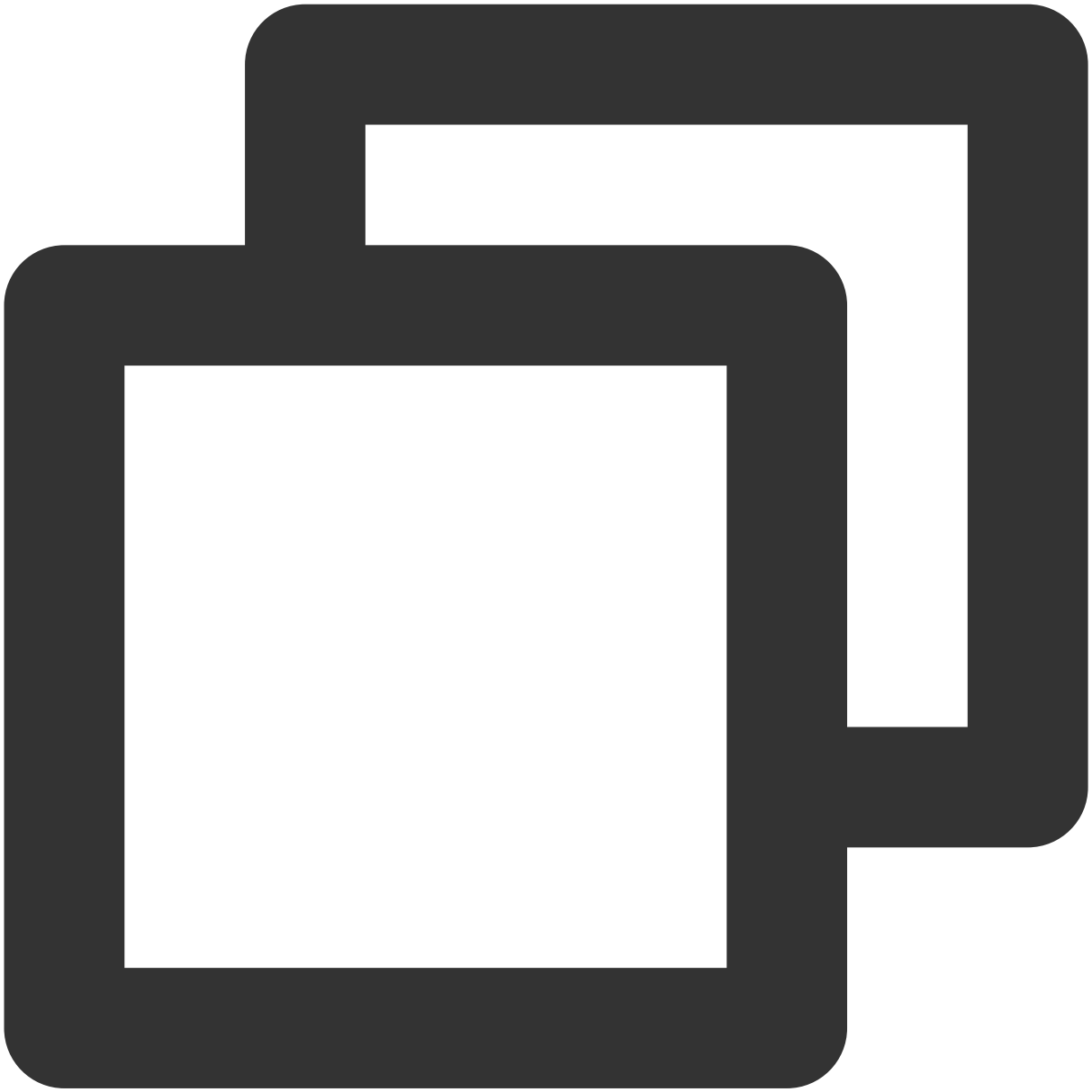
```
REPLACE(<str> string, <search> string[, <replace> string])
```

Supported engines: SparkSQL and Presto

Usage instructions: Replace all occurrences of search in str with replace.

Return type: string

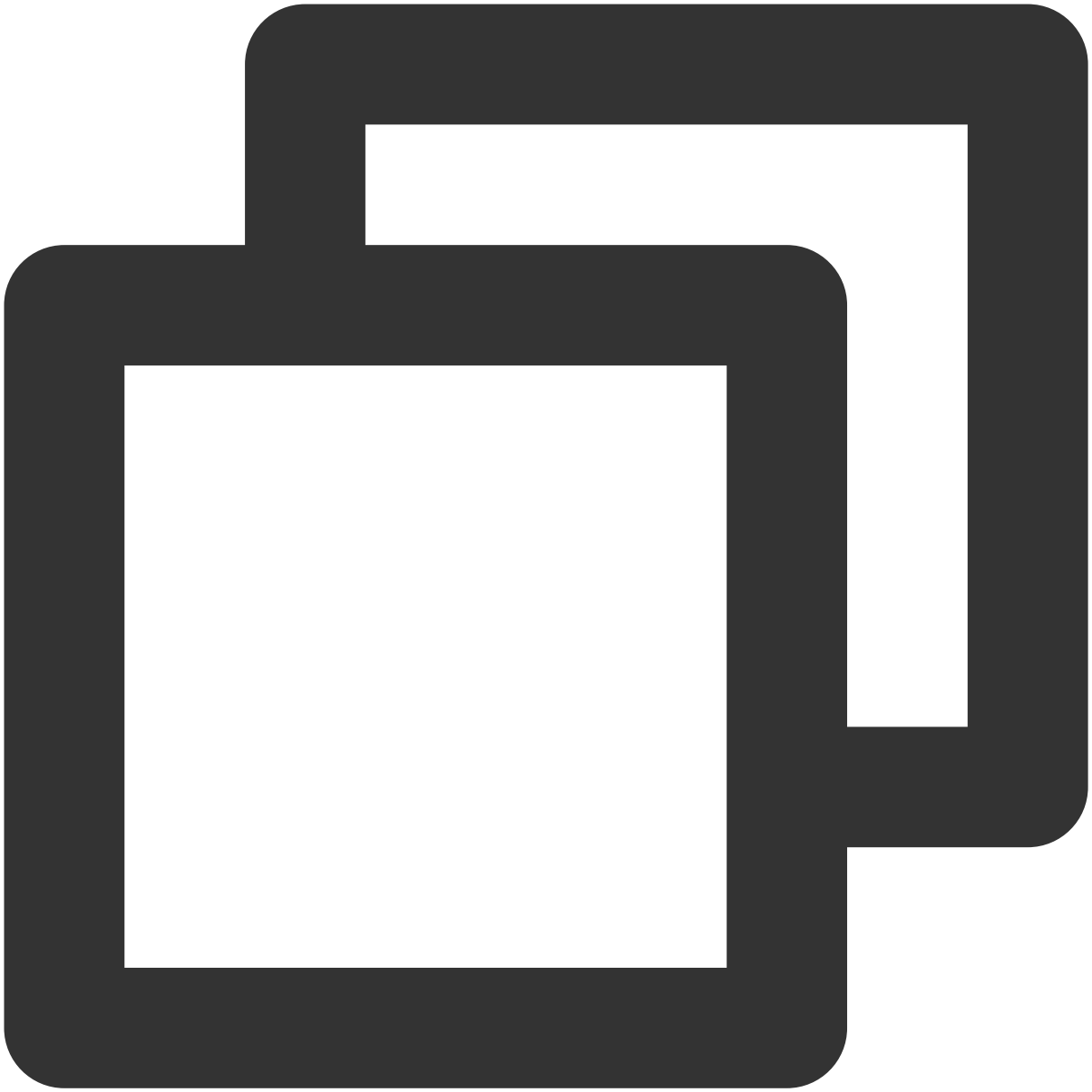
Example:



```
> SELECT replace('ABCabc', 'abc', 'DEF');
 ABCDEF
```

OVERLAY

Function statement:



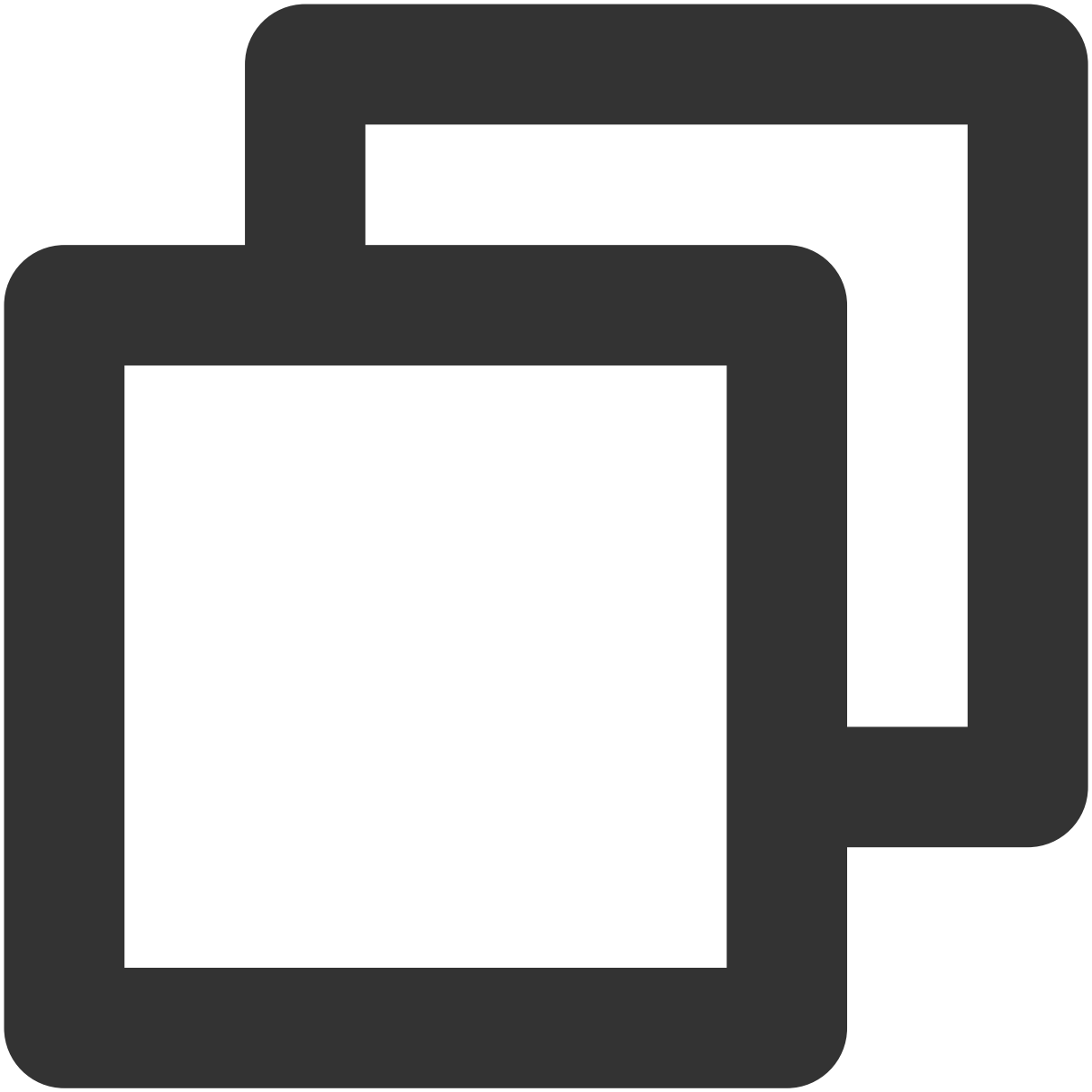
```
OVERLAY(<input> string, <replace> string, <pos> integer[, <len> integer])
```

Supported engine: SparkSQL

Usage instructions: Replace input with replace, starting from pos and spanning len.

Return type: string

Example:

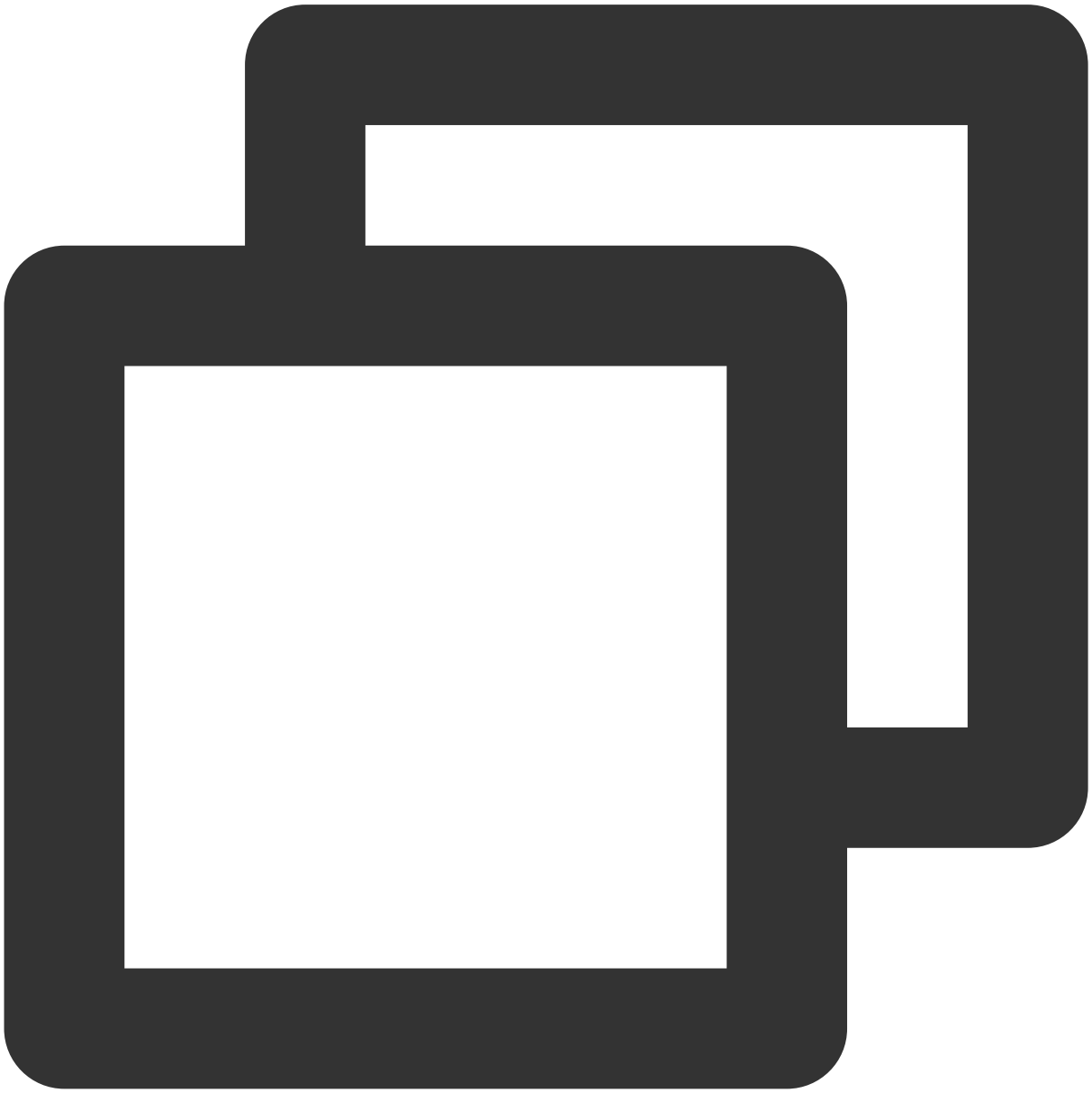


```
> SELECT overlay('Spark SQL' PLACING '_' FROM 6);
 Spark_SQL
> SELECT overlay('Spark SQL' PLACING 'CORE' FROM 7);
 Spark CORE
> SELECT overlay('Spark SQL' PLACING 'ANSI ' FROM 7 FOR 0);
 Spark ANSI SQL
> SELECT overlay('Spark SQL' PLACING 'tructured' FROM 2 FOR 4);
 Structured SQL
> SELECT overlay(encode('Spark SQL', 'utf-8') PLACING encode('_', 'utf-8') FROM 6);
 Spark_SQL
> SELECT overlay(encode('Spark SQL', 'utf-8') PLACING encode('CORE', 'utf-8') FROM
```

```
 Spark CORE
> SELECT overlay(encode('Spark SQL', 'utf-8') PLACING encode('ANSI ', 'utf-8') FROM
 Spark ANSI SQL
> SELECT overlay(encode('Spark SQL', 'utf-8') PLACING encode('tructured', 'utf-8')
 Structured SQL
```

RPAD

Function statement:



```
RPAD(<str> string, <len> integer[, <pad> string])
```

Supported engines: SparkSQL and Presto

Usage instructions: Return str, right-padded with pad to a length of len. If str is longer than len, the returned value is truncated to len characters. If pad is not specified, str will be padded with space characters.

Return type: string

Example:

```
> SELECT rpad('hi', 5, '??');
 hi???
```

```
> SELECT rpad('hi', 1, '??');
 h
> SELECT rpad('hi', 5);
 hi
```

RTRIM

Function statement:

```
RTRIM(<str> string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Remove trailing space characters from str.

Return type: string

Example:

```
> SELECT rtrim('    SparkSQL   ');
   SparkSQL
```

SENTENCES

Function statement:

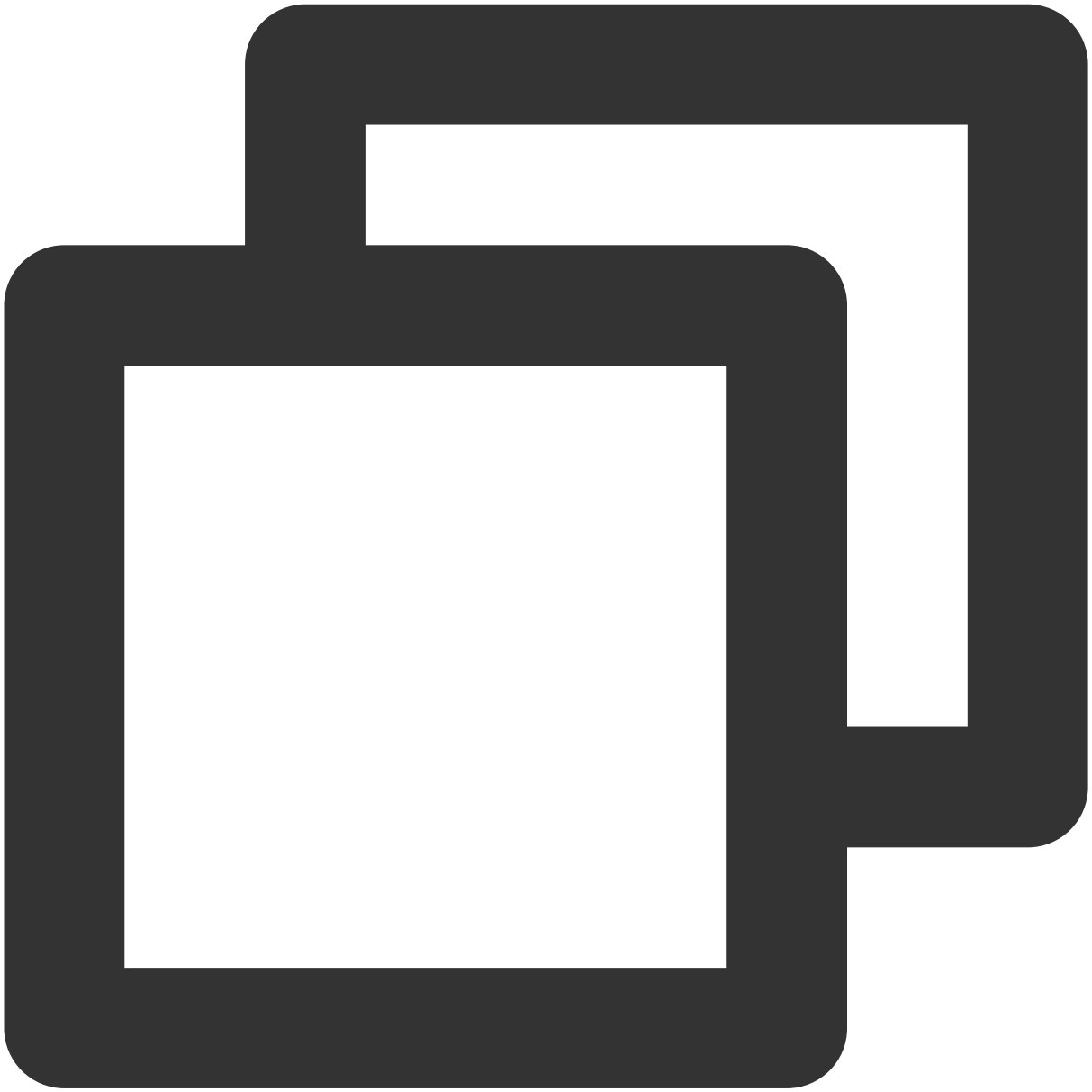
```
SENTENCES(<str> string[, <lang> string, <country> string])
```

Supported engines: SparkSQL and Presto

Usage instructions: Split str into an array of words.

Return type: array <string>.

Example:



```
> SELECT sentences('Hi there! Good morning.');
 [["Hi","there"],["Good","morning"]]
```

SOUNDEX

Function statement:

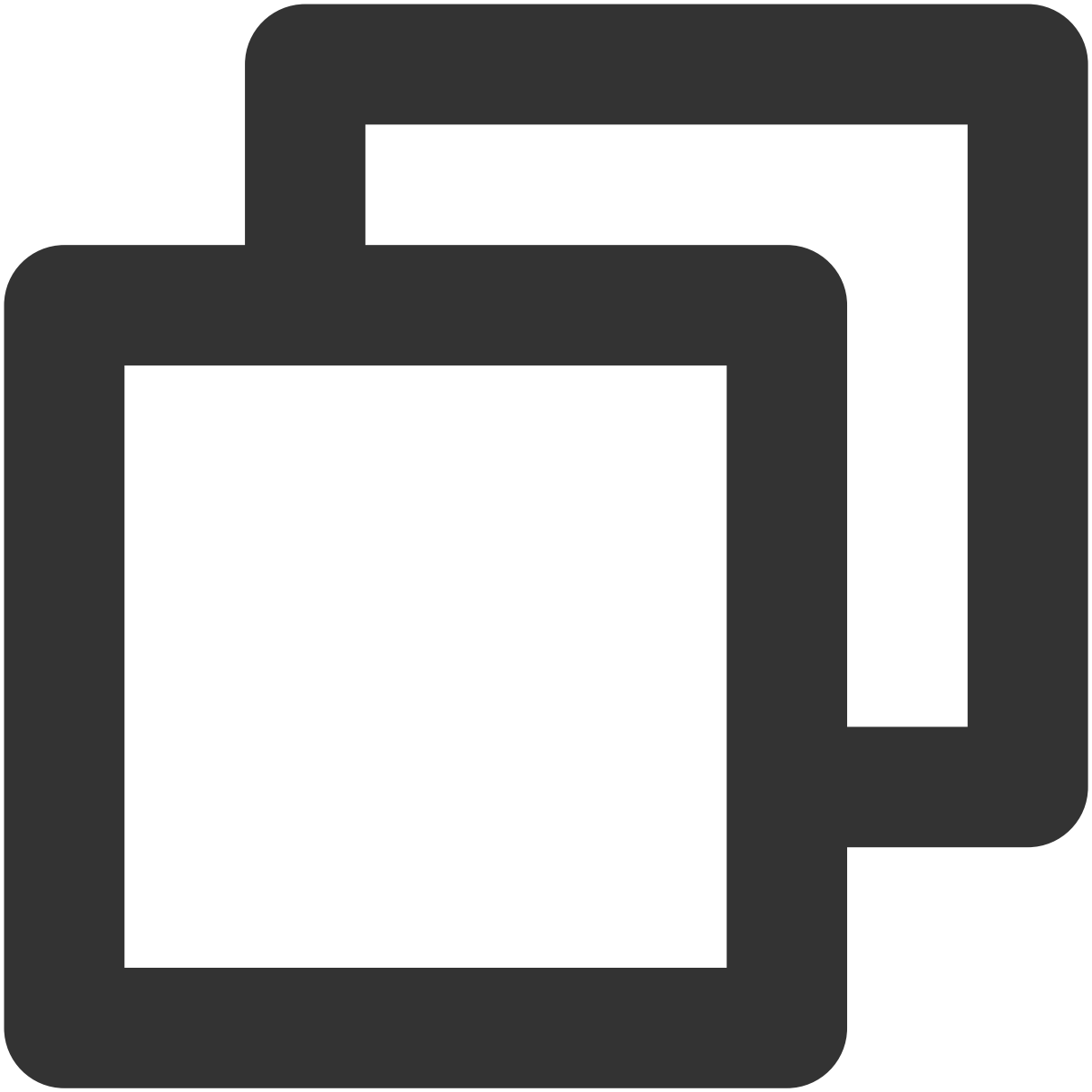
```
SOUNDEX(<str> string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the Soundex code of the string.

Return type: string

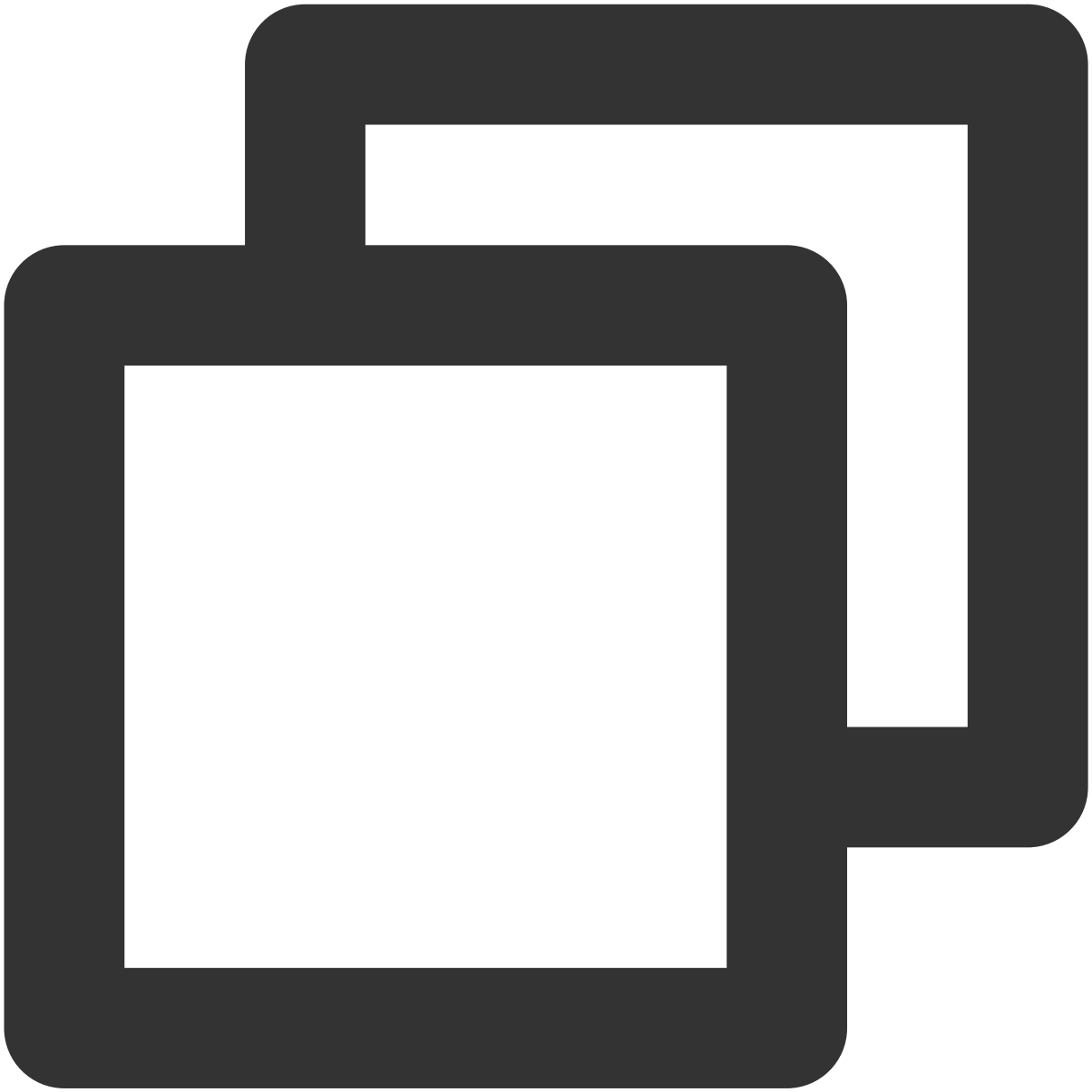
Example:



```
> SELECT soundex('Miller');
 M460
```

SPACE

Function statement:



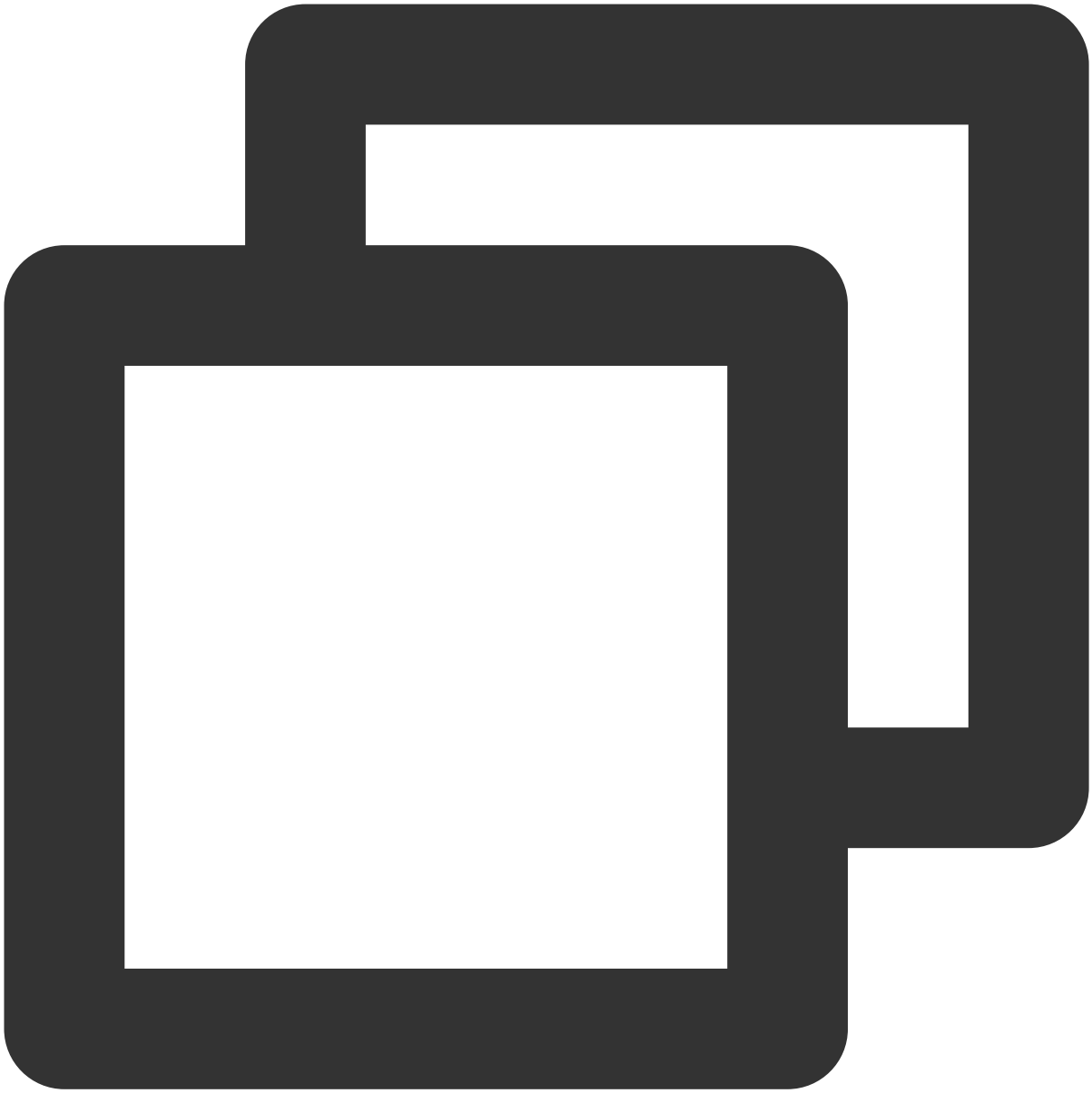
```
SPACE(<n> integer)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return a string consisting of n spaces.

Return type: string

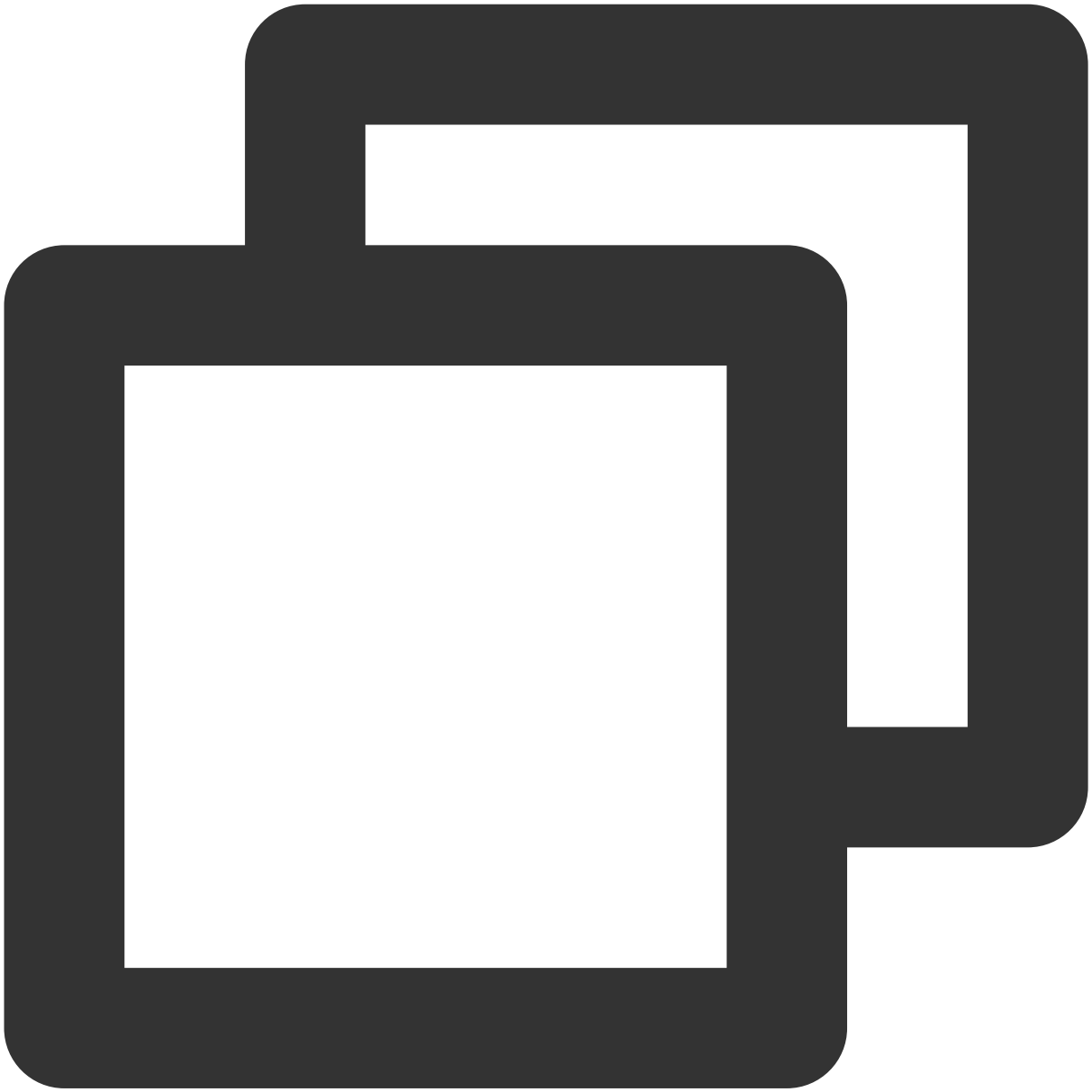
Example:



```
> SELECT concat(space(2), '1');
 1
```

SPLIT

Function statement:



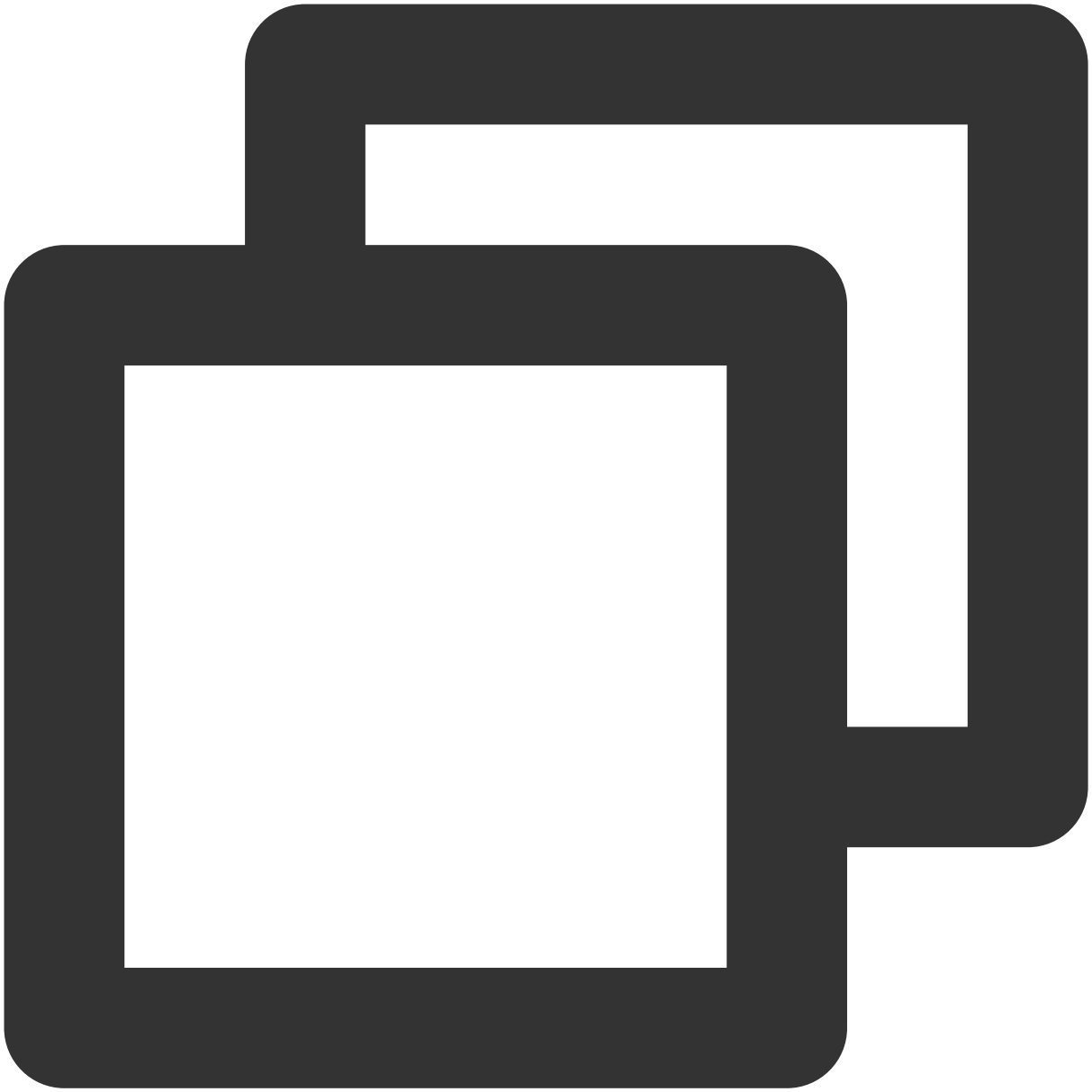
```
SPLIT(<str> string, <regex> string, <limit> integer)
```

Supported engines: SparkSQL and Presto

Usage instructions: Split str using the string that matches the regex as the delimiter, and return an array with a maximum length of limit.

Return type: array <string>.

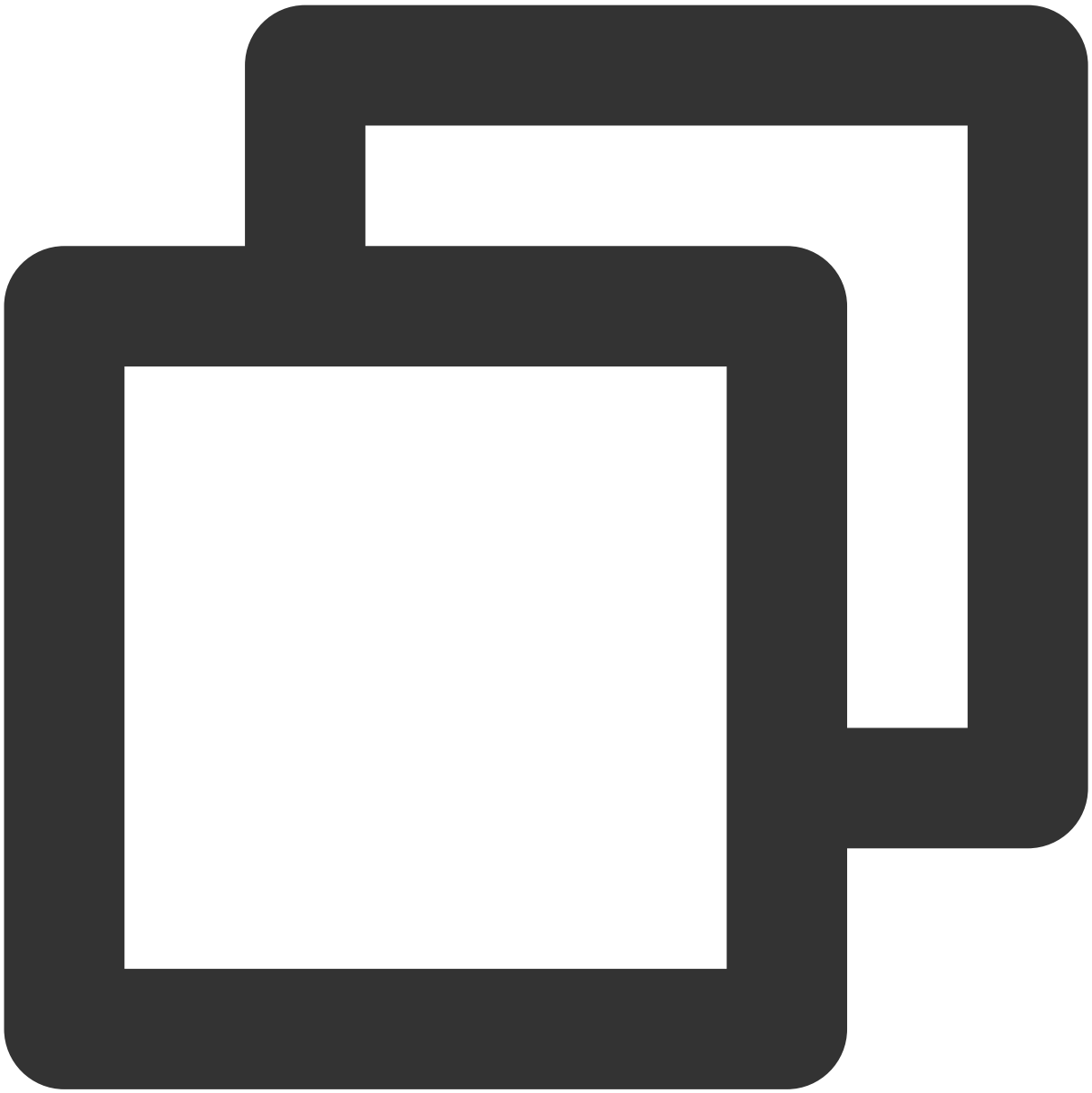
Example:



```
> SELECT split('oneAtwoBthreeC', '[ABC]');
 ["one","two","three",""]
> SELECT split('oneAtwoBthreeC', '[ABC]', -1);
 ["one","two","three",""]
> SELECT split('oneAtwoBthreeC', '[ABC]', 2);
 ["one","twoBthreeC"]
```

SUBSTRING

Function statement:



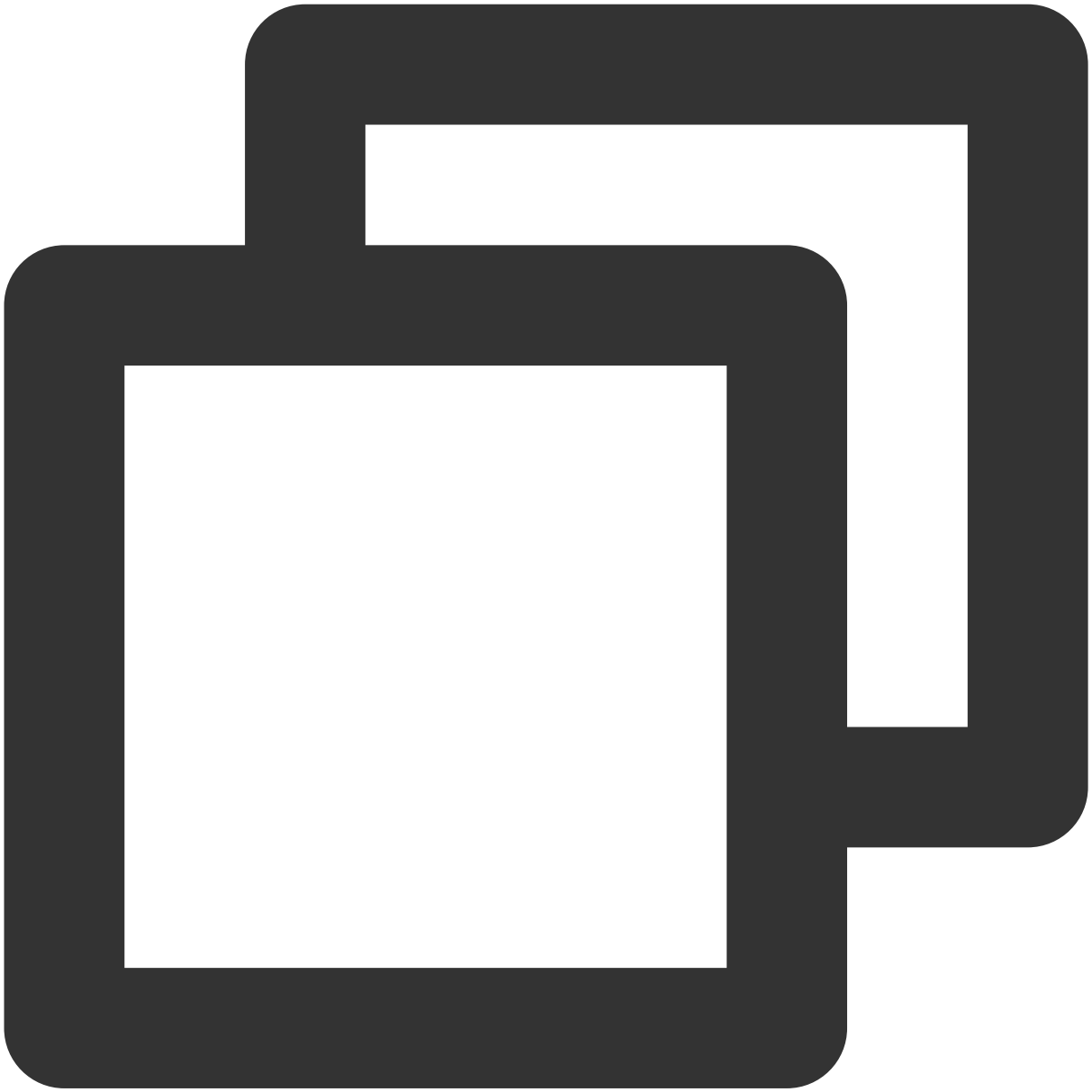
```
SUBSTRING(<str> string, <pos> integer[, <len> integer])
SUBSTRING(<str> FROM <pos>[ FOR <len>])
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the substring of str starting from pos with a length of len, or the byte array slice starting from pos with a length of len.

Return type: string

Example:

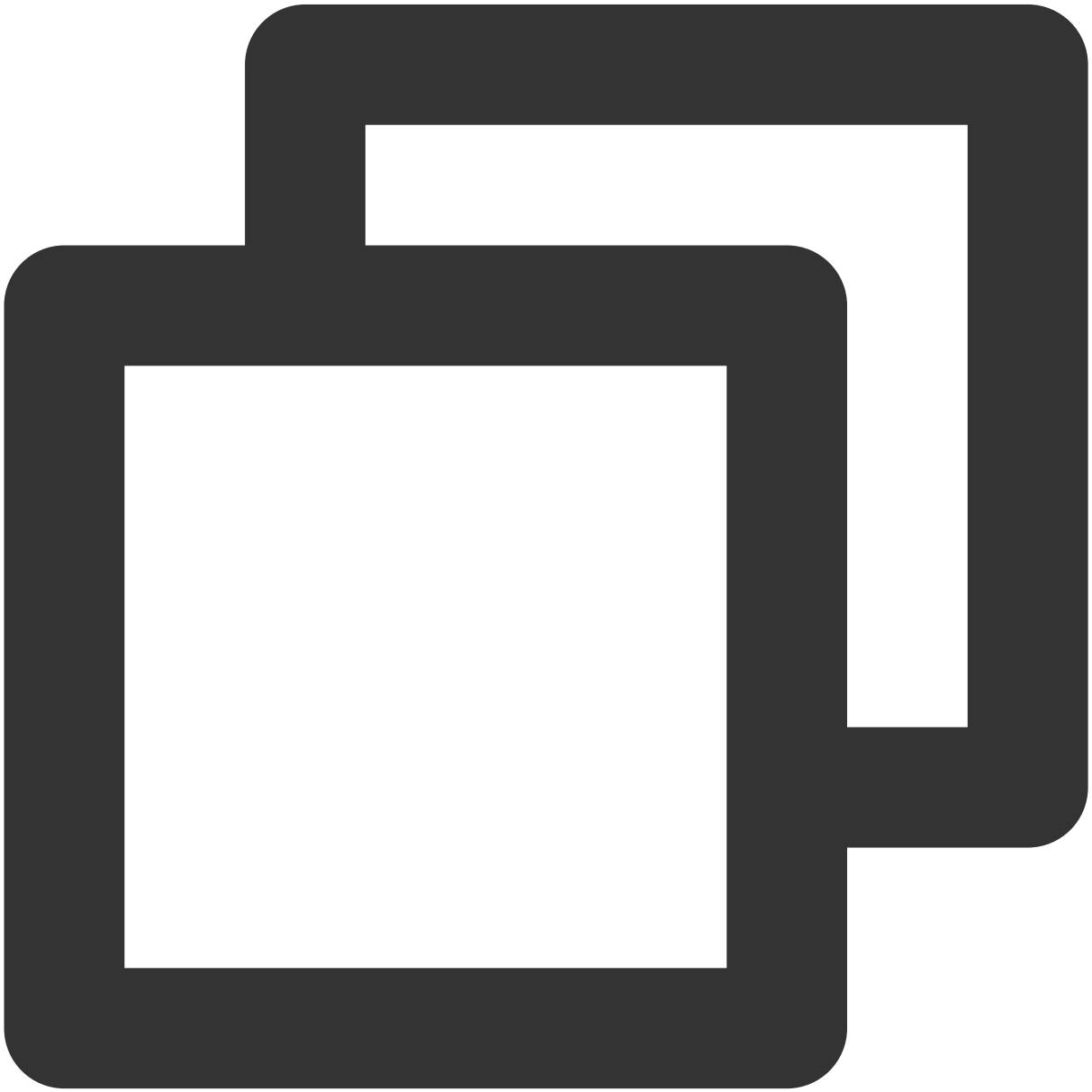


```
> SELECT substring('Spark SQL', 5);
 k SQL
> SELECT substring('Spark SQL', -3);
 SQL
> SELECT substring('Spark SQL', 5, 1);
 k
> SELECT substring('Spark SQL' FROM 5);
 k SQL
> SELECT substring('Spark SQL' FROM -3);
 SQL
> SELECT substring('Spark SQL' FROM 5 FOR 1);
```

```
k
```

SUBSTR

Function statement:



```
SUBSTR(<str> string, <pos> integer[, <len> integer])
SUBSTR(<str> FROM <pos>[ FOR <len>])
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the substring of str starting from pos with a length of len, or the byte array slice starting from pos with a length of len.

Return type: string

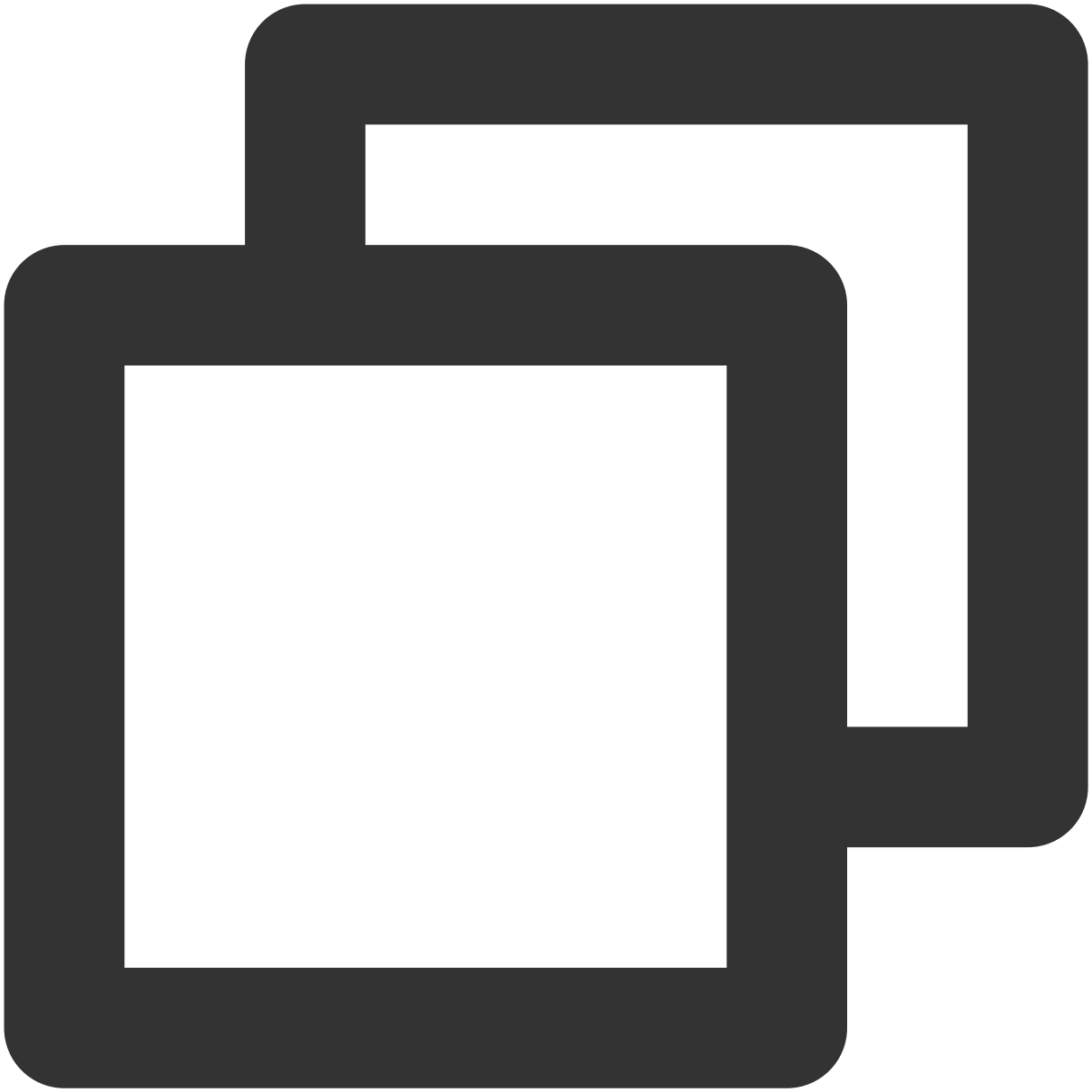
Example:

```
> SELECT substr('Spark SQL', 5);
 k SQL
> SELECT substr('Spark SQL', -3);
 SQL
```

```
> SELECT substr('Spark SQL', 5, 1);
 k
> SELECT substr('Spark SQL' FROM 5);
 k SQL
> SELECT substr('Spark SQL' FROM -3);
 SQL
> SELECT substr('Spark SQL' FROM 5 FOR 1);
 k
```

LEFT

Function statement:



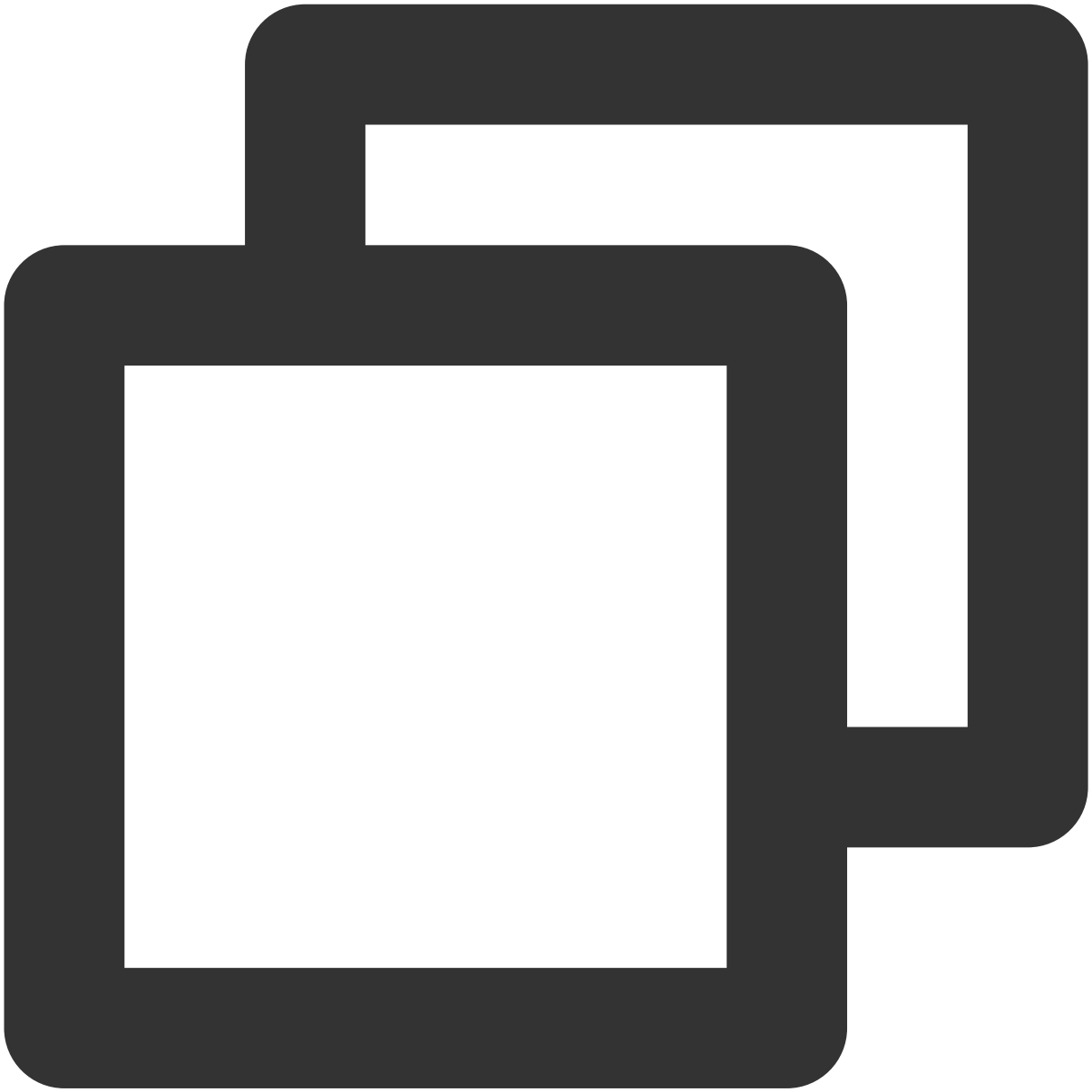
```
LEFT(<str> string, <len> integer)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the leftmost len characters of the string str. If len is less than or equal to 0, the result is a null character string.

Return type: string

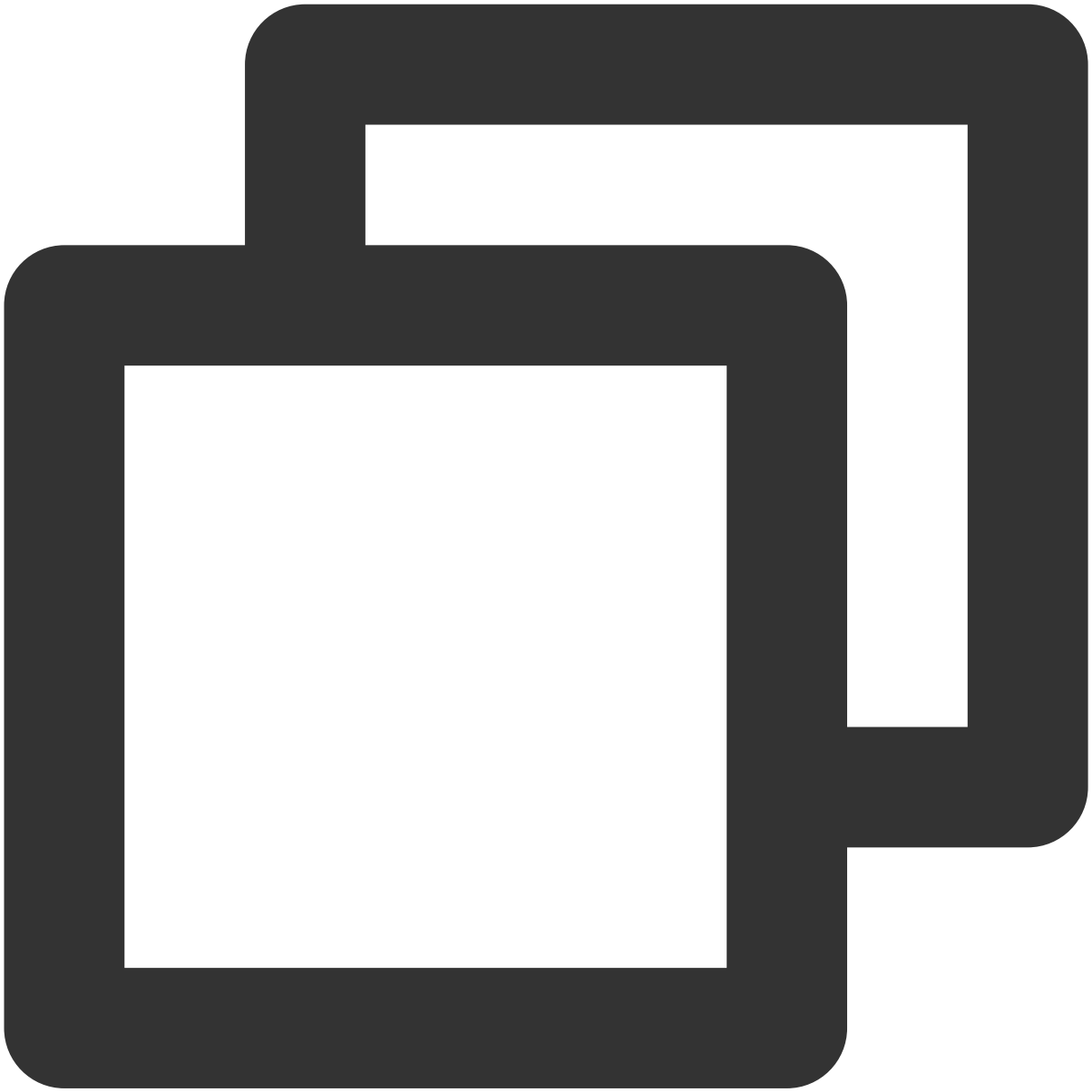
Example:



```
> SELECT left('tencent', 3);
 ten
```

RIGHT

Function statement:



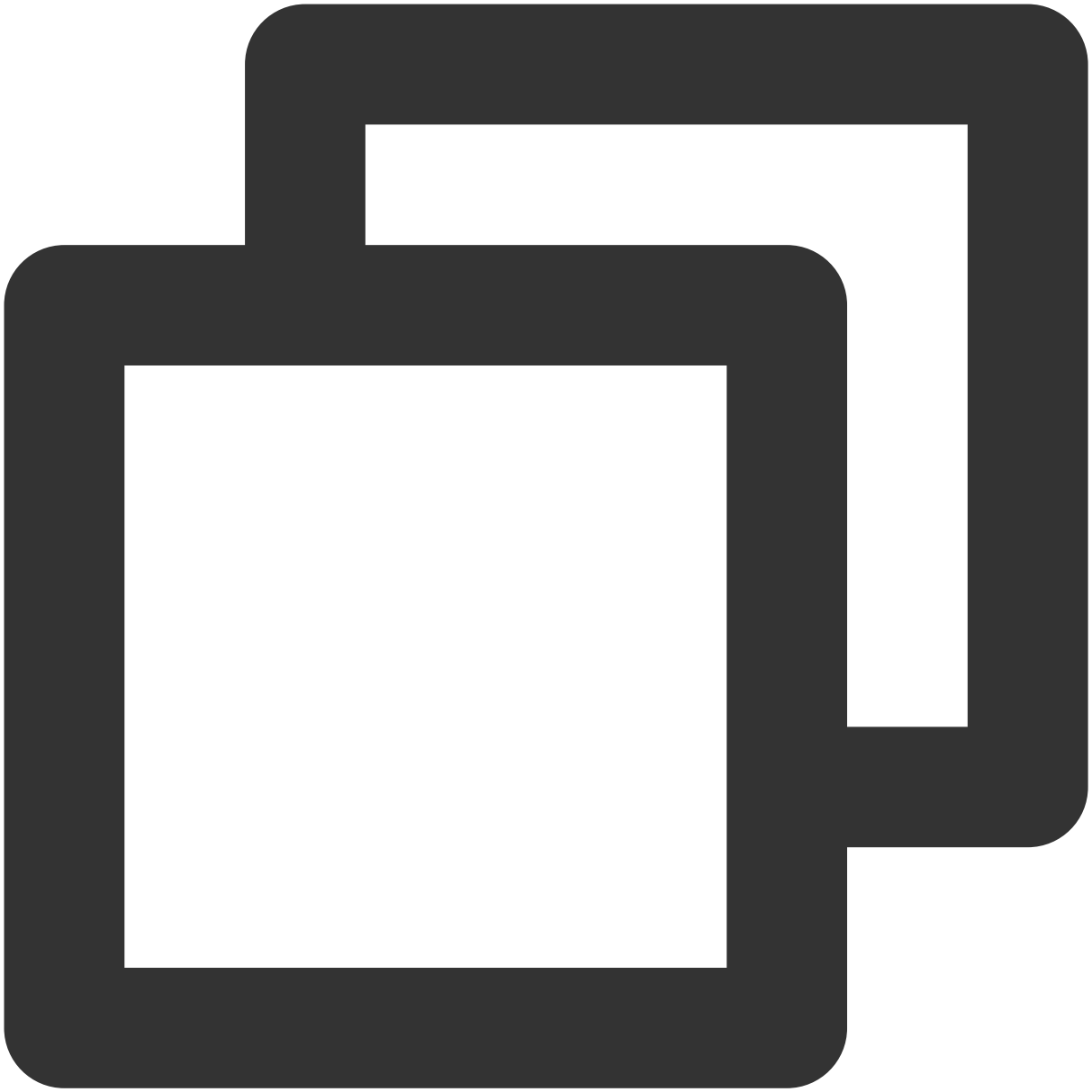
```
RIGHT(<str> string, <len> integer)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the leftmost len characters of the string str. If len is less than or equal to 0, the result is a null character string.

Return type: string

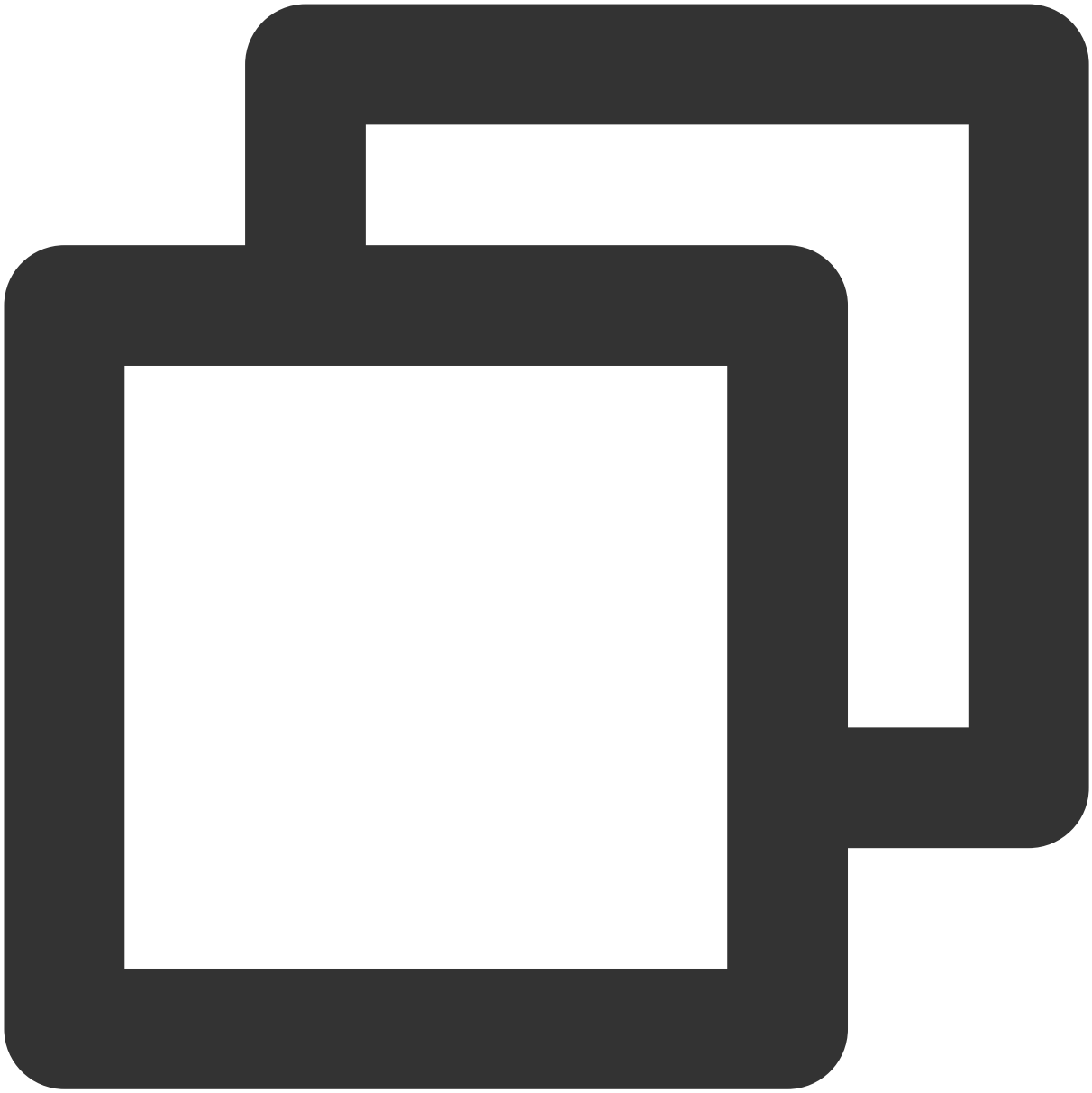
Example:



```
> SELECT left('tencent', 3);
 ten
```

SUBSTRING_INDEX

Function statement:



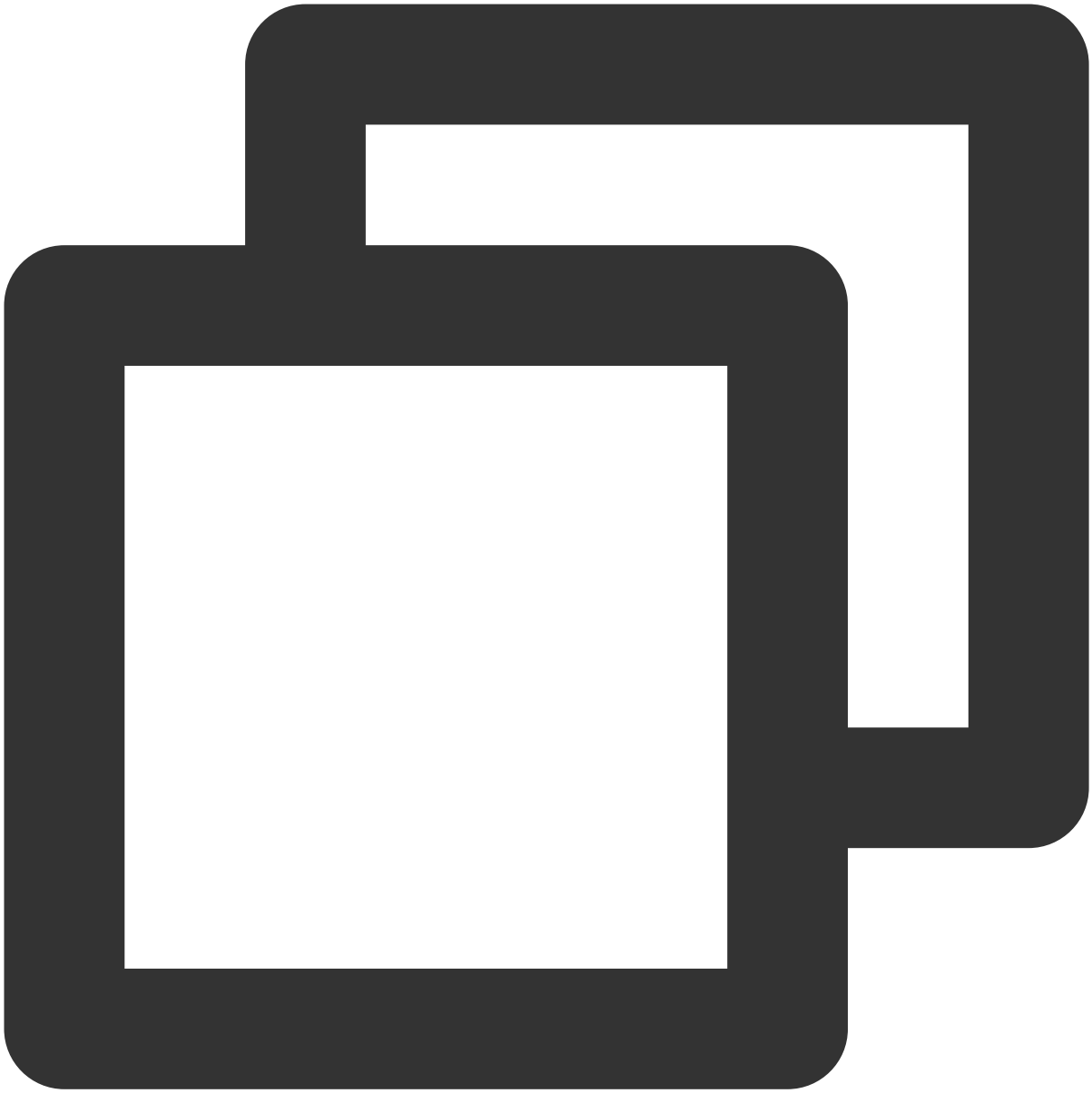
```
SUBSTRING_INDEX(<str> string, <delim> string, <count> integer)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the substring of str before the count occurrence of delim. If count is positive, return all content to the left of the final delimiter (counted from the left). If count is negative, return all content to the right of the final delimiter (counted from the right). The function is case-sensitive when matching delim.

Return type: string

Example:



```
> SELECT substring_index('cloud.tencent.com', '.', 2);
 cloud.tencent
```

TRANSLATE

Function statement:

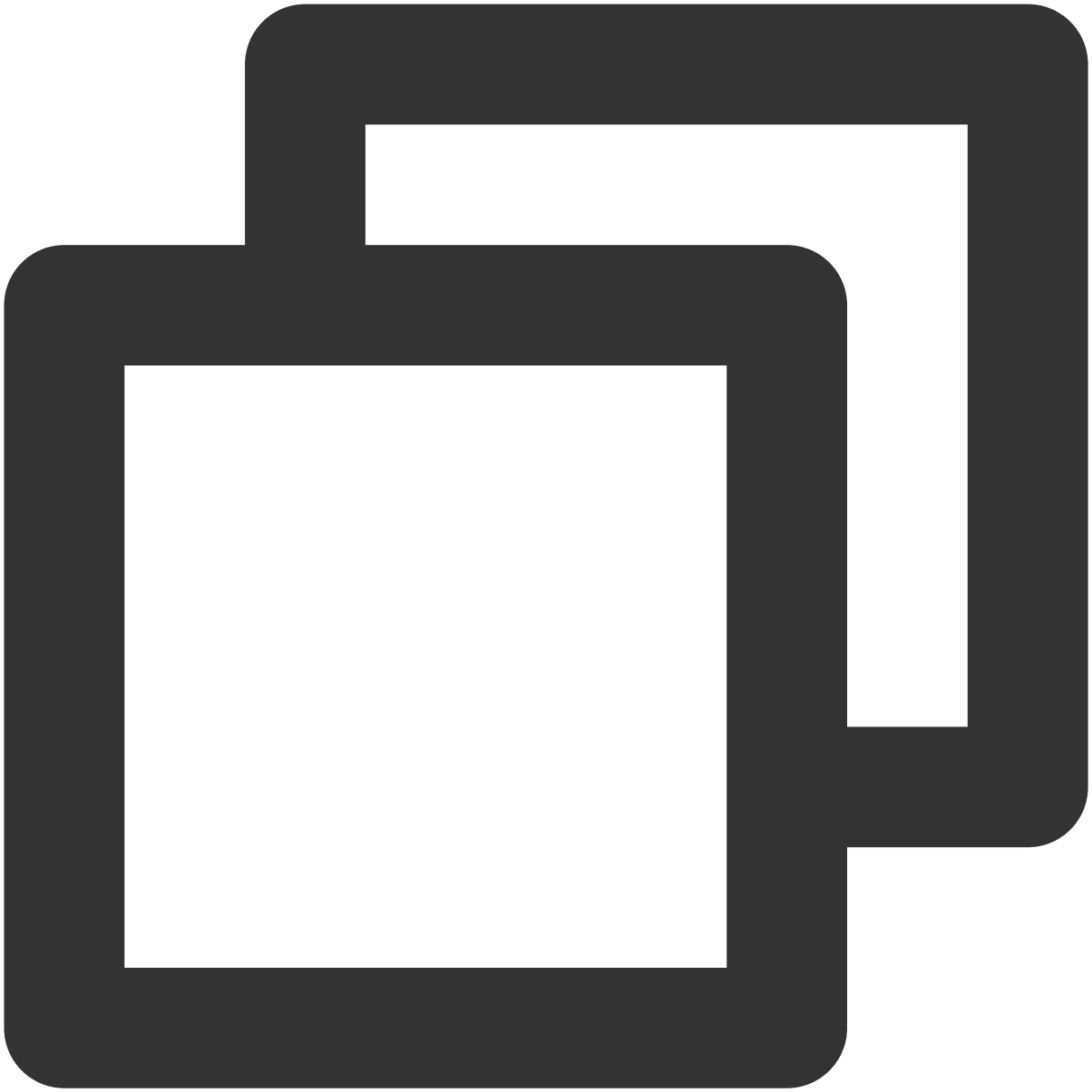
```
TRANSLATE(<input> string, <from> string, <to> string)
```

Supported engine: SparkSQL

Usage instructions: Transform the input string by replacing characters in the from string with the corresponding characters in the to string.

Return type: string

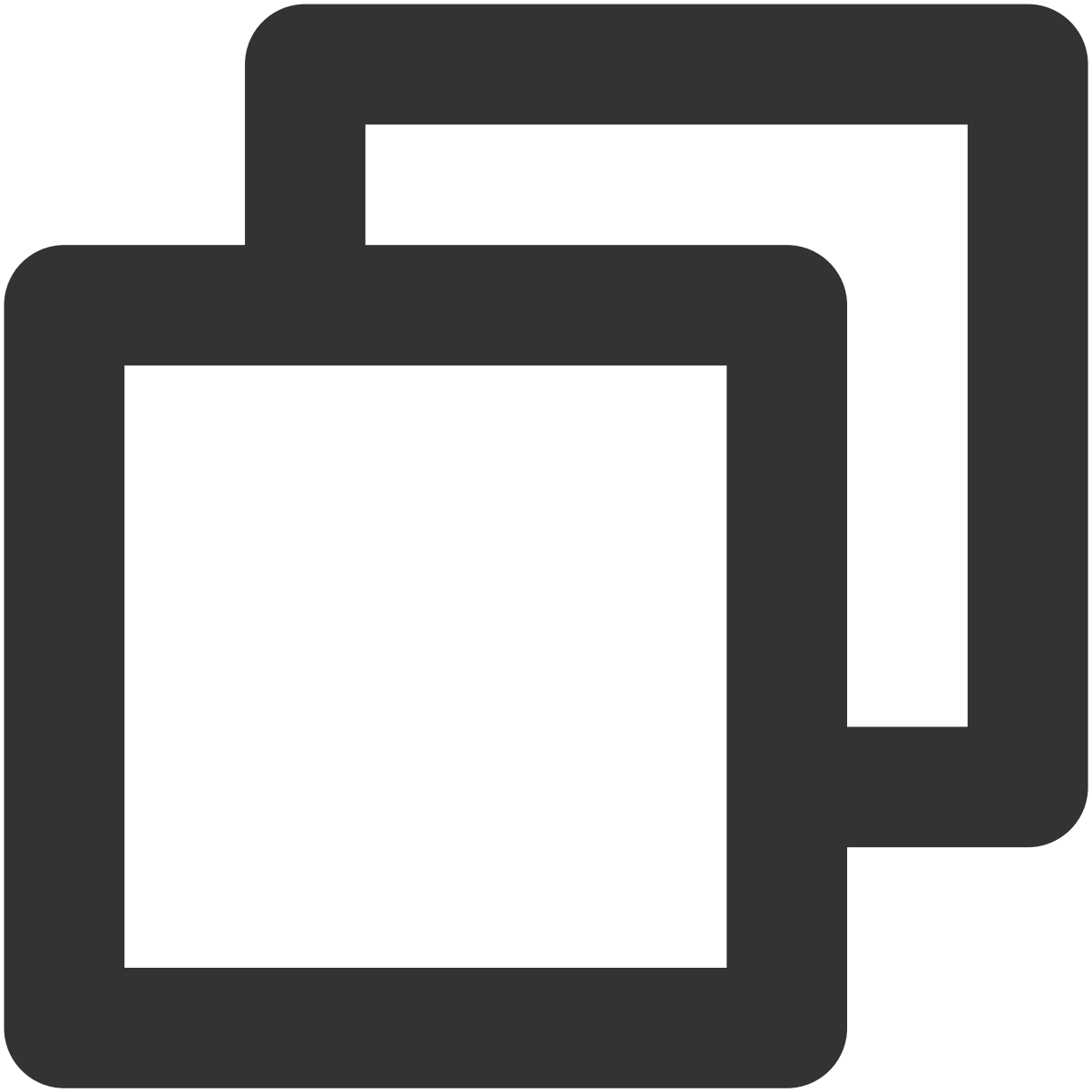
Example:



```
> SELECT translate('AaBbCc', 'abc', '123');
 A1B2C3
```

TRIM

Function statement:



```
TRIM(<str> string)
trim(BOTH FROM str)
trim(LEADING FROM str)
trim(TRAILING FROM str)
trim(trimStr FROM str)
trim(BOTH trimStr FROM str)
trim(LEADING trimStr FROM str)
trim(TRAILING trimStr FROM str)
```

Supported engines: SparkSQL and Presto

Usage instructions:

trim(str) - Remove leading and trailing space characters from str.

trim(BOTH FROM str): Remove leading and trailing space characters from str.

trim(LEADING FROM str): Remove leading space characters from str.

trim(TRAILING FROM str): Remove trailing space characters from str.

trim(trimStr FROM str): Remove leading and trailing trimStr characters from str.

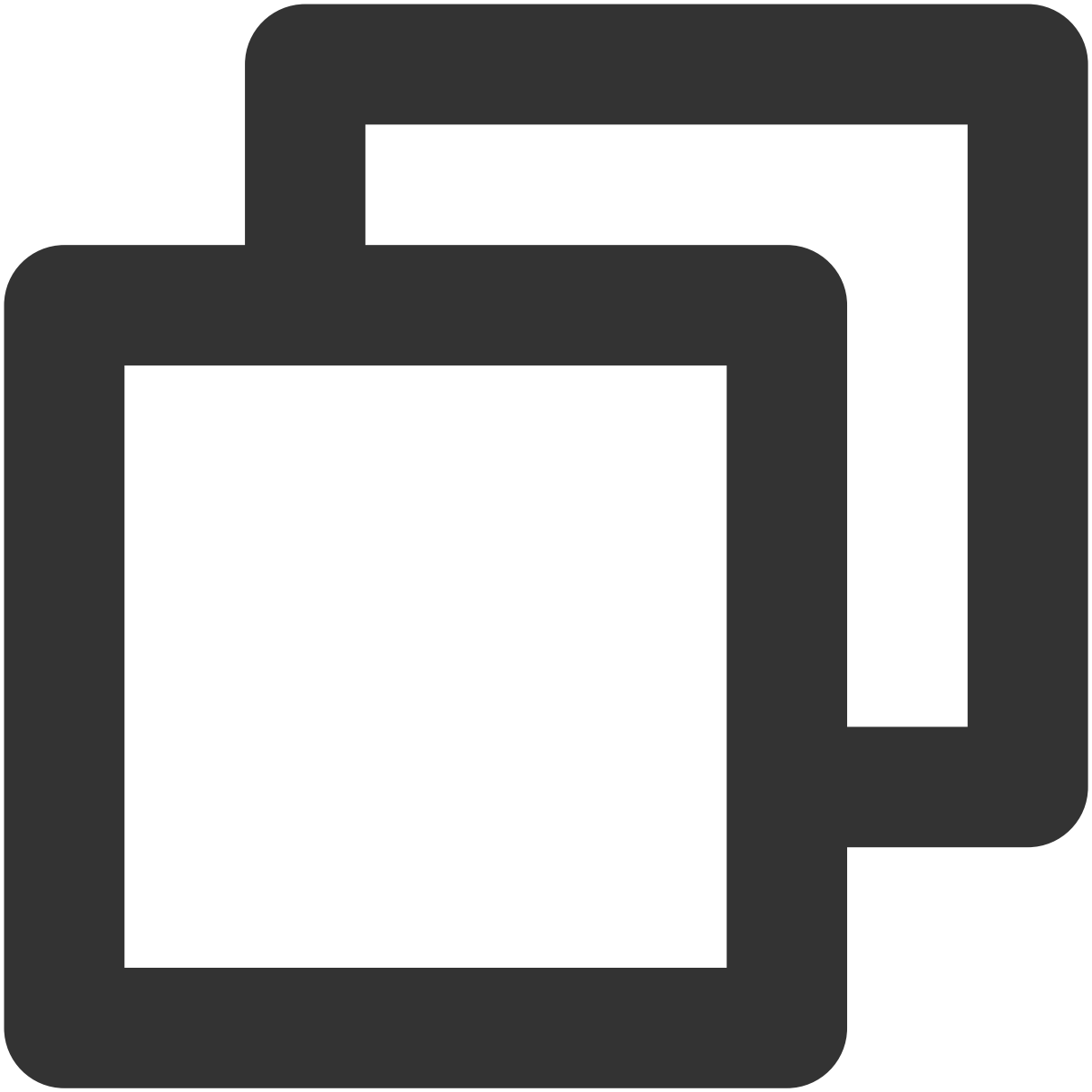
trim(BOTH trimStr FROM str): Remove leading and trailing trimStr characters from str.

trim(LEADING trimStr FROM str): Remove leading trimStr characters from str.

trim(TRAILING trimStr FROM str): Remove trailing trimStr characters from str.

Return type: string

Example:



```
> SELECT trim('    SparkSQL   ');
 SparkSQL
> SELECT trim(BOTH FROM '    SparkSQL   ');
 SparkSQL
> SELECT trim(LEADING FROM '    SparkSQL   ');
 SparkSQL
> SELECT trim(TRAILING FROM '    SparkSQL   ');
   SparkSQL
> SELECT trim('SL' FROM 'SSparkSQLS');
 parkSQ
> SELECT trim(BOTH 'SL' FROM 'SSparkSQLS');
```

```
 parkSQ
> SELECT trim(LEADING 'SL' FROM 'SSparkSQLS');
 parkSQLS
> SELECT trim(TRAILING 'SL' FROM 'SSparkSQLS');
 SSparkSQ
```

BTRIM

Function statement:

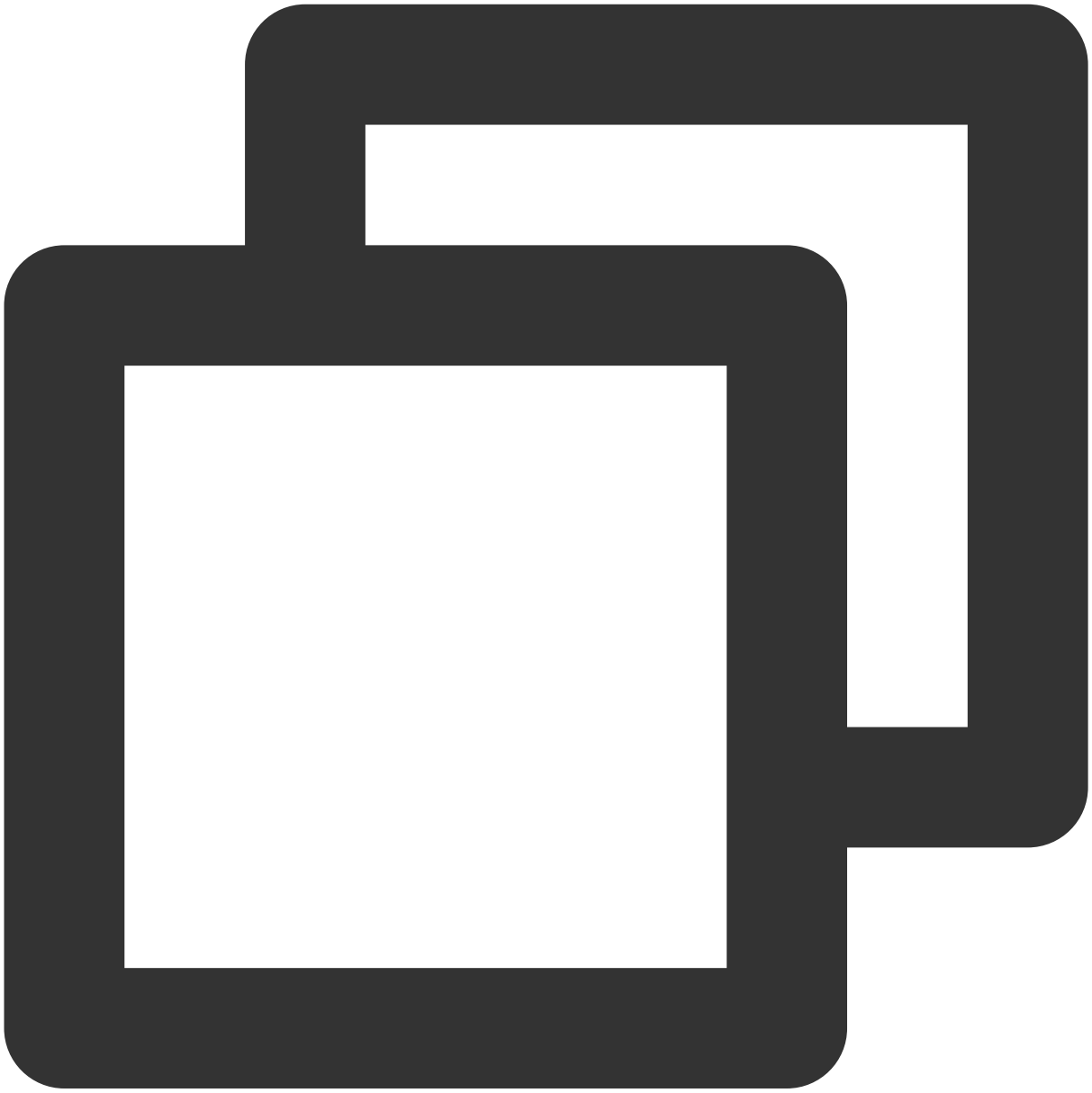
```
BTRIM(<str> string[, <trimStr> string])
```

Supported engines: SparkSQL and Presto

Usage instructions: Remove leading and trailing trimStr characters (default is space) from str.

Return type: string

Example:



```
> SELECT btrim('    SparkSQL   ');
 SparkSQL
> SELECT btrim(encode('    SparkSQL   ', 'utf-8'));
```

```
 SparkSQL
> SELECT btrim('SSparkSQLS', 'SL');
 parkSQ
> SELECT btrim(encode('SSparkSQLS', 'utf-8'), encode('SL', 'utf-8'));
 parkSQ
```

UCASE

Function statement:

```
UCASE(<str> string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return str with all characters converted to uppercase.

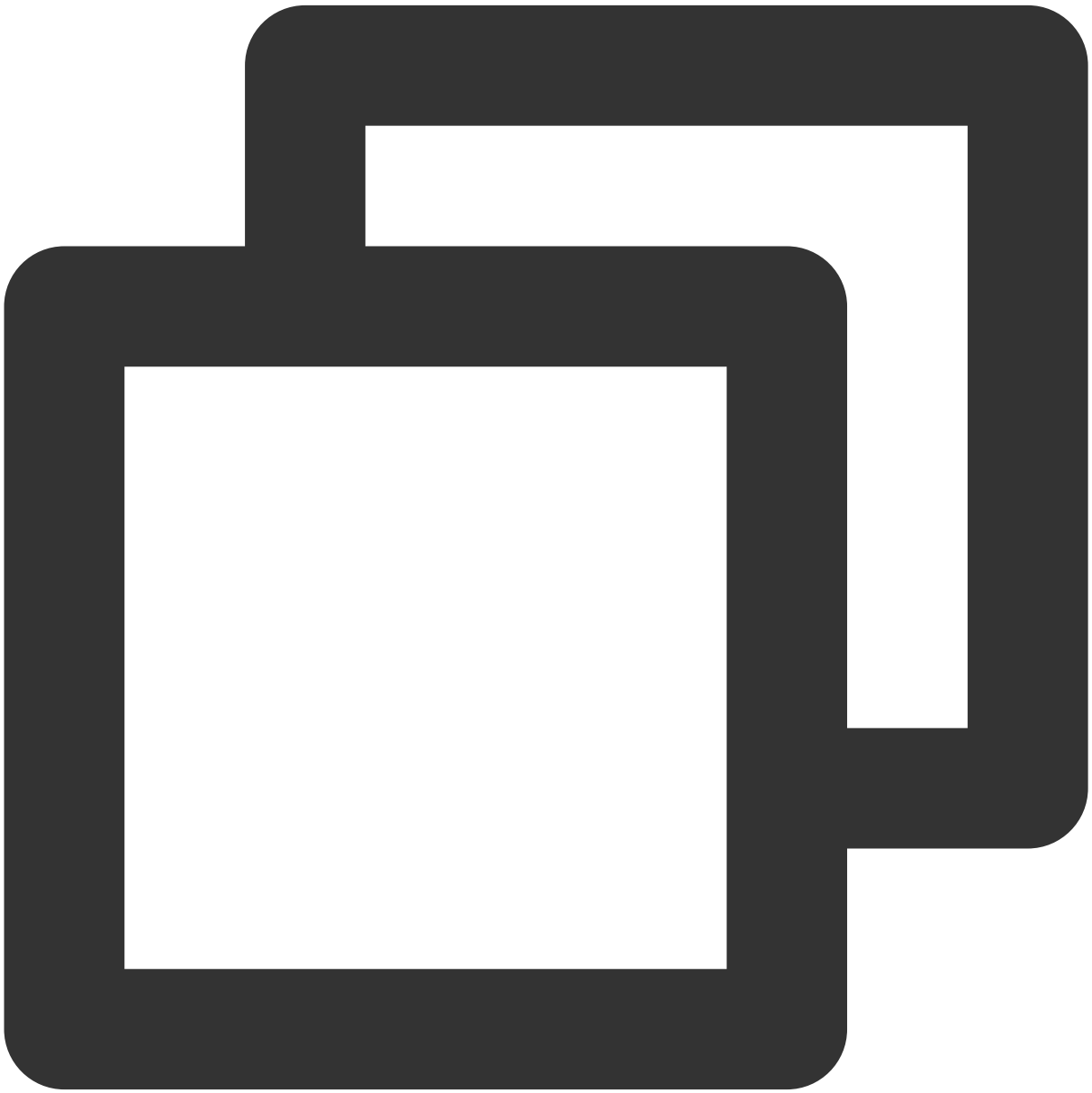
Return type: string

Example:

```
> SELECT ucase('SparkSQL');
 SPARKSQL
```

UNBASE64

Function statement:



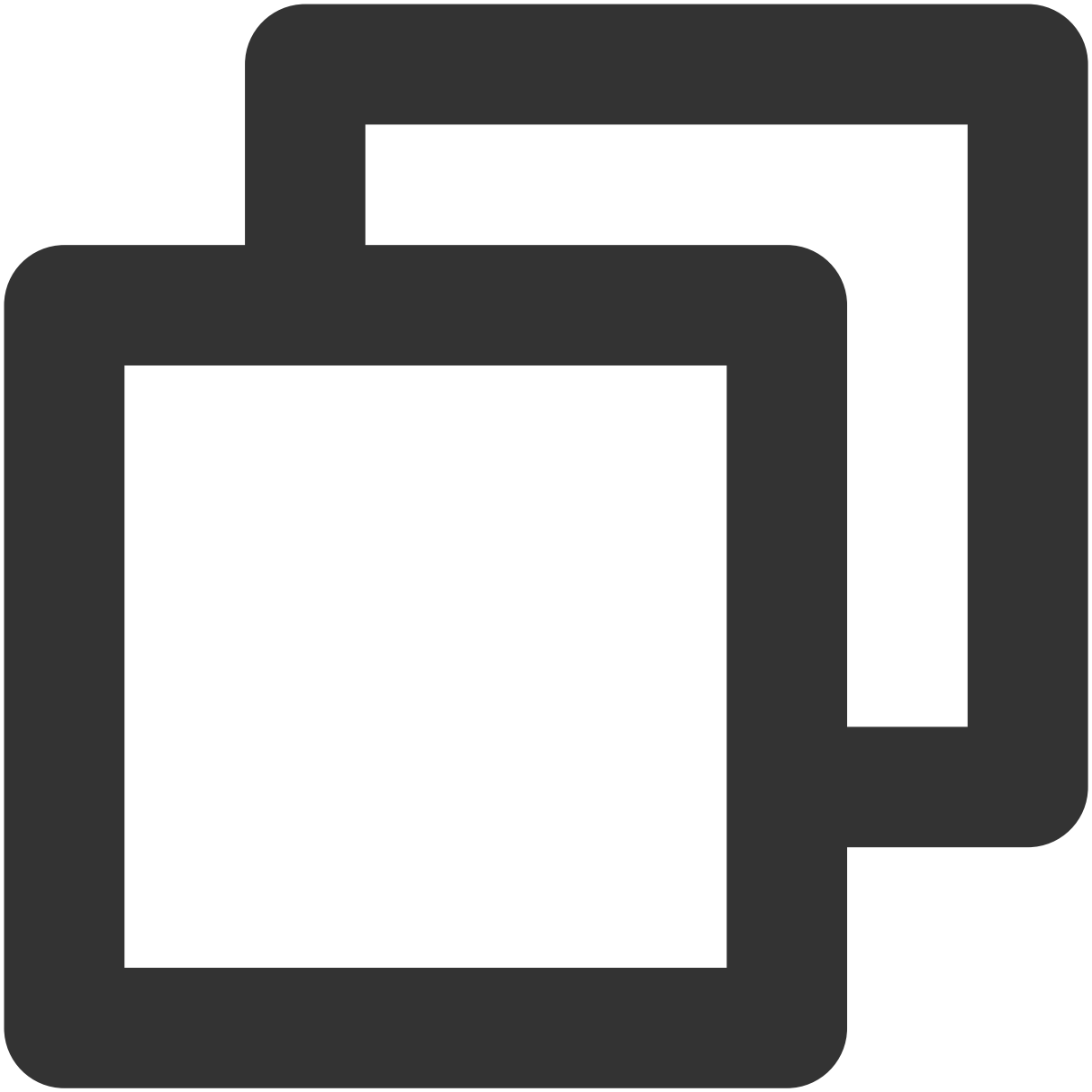
```
UNBASE64(<str> string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Convert str from a base64 string to binary.

Return type: binary

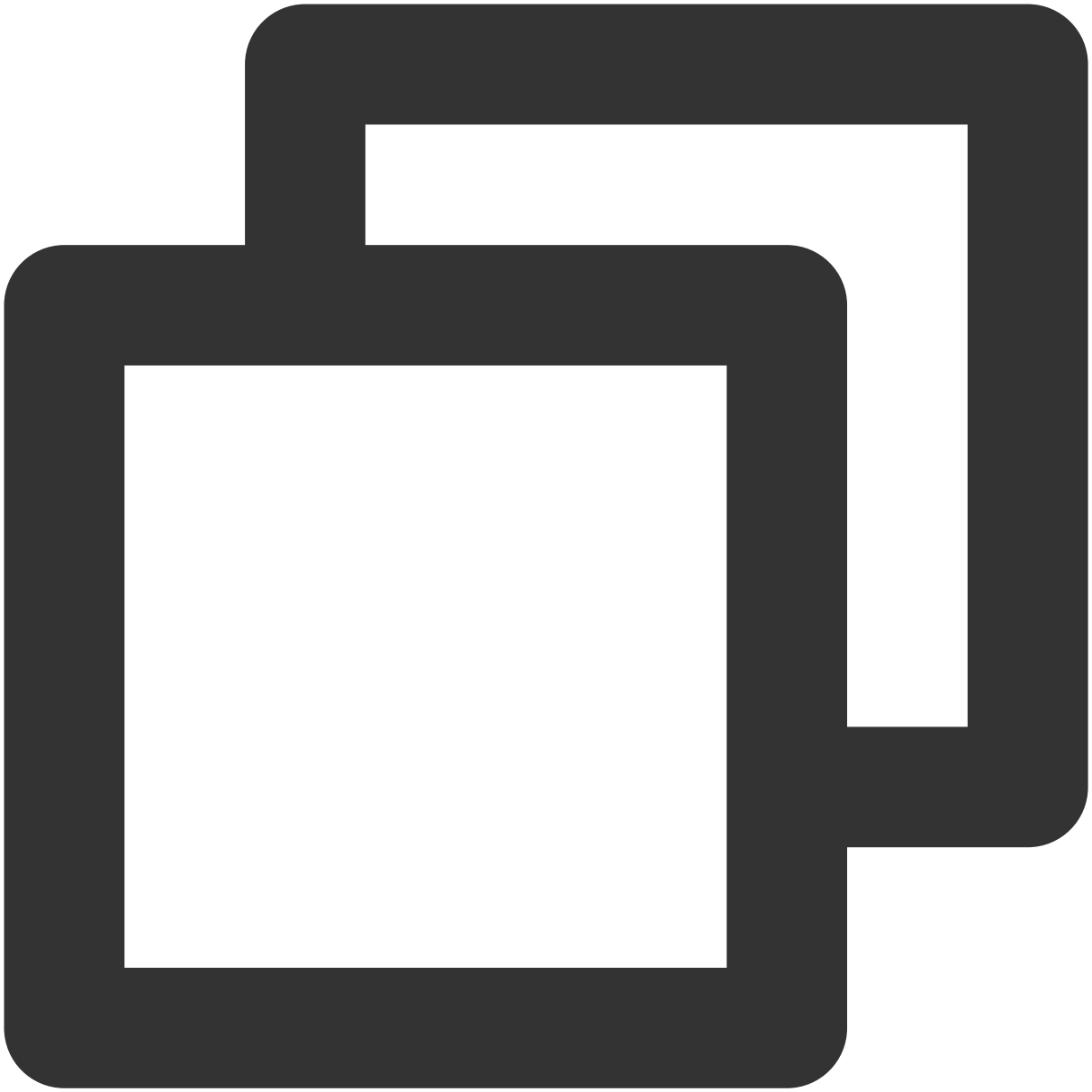
Example:



```
> SELECT unbase64('U3BhcmsgU1FM');
 Spark SQL
```

UNHEX

Function statement:



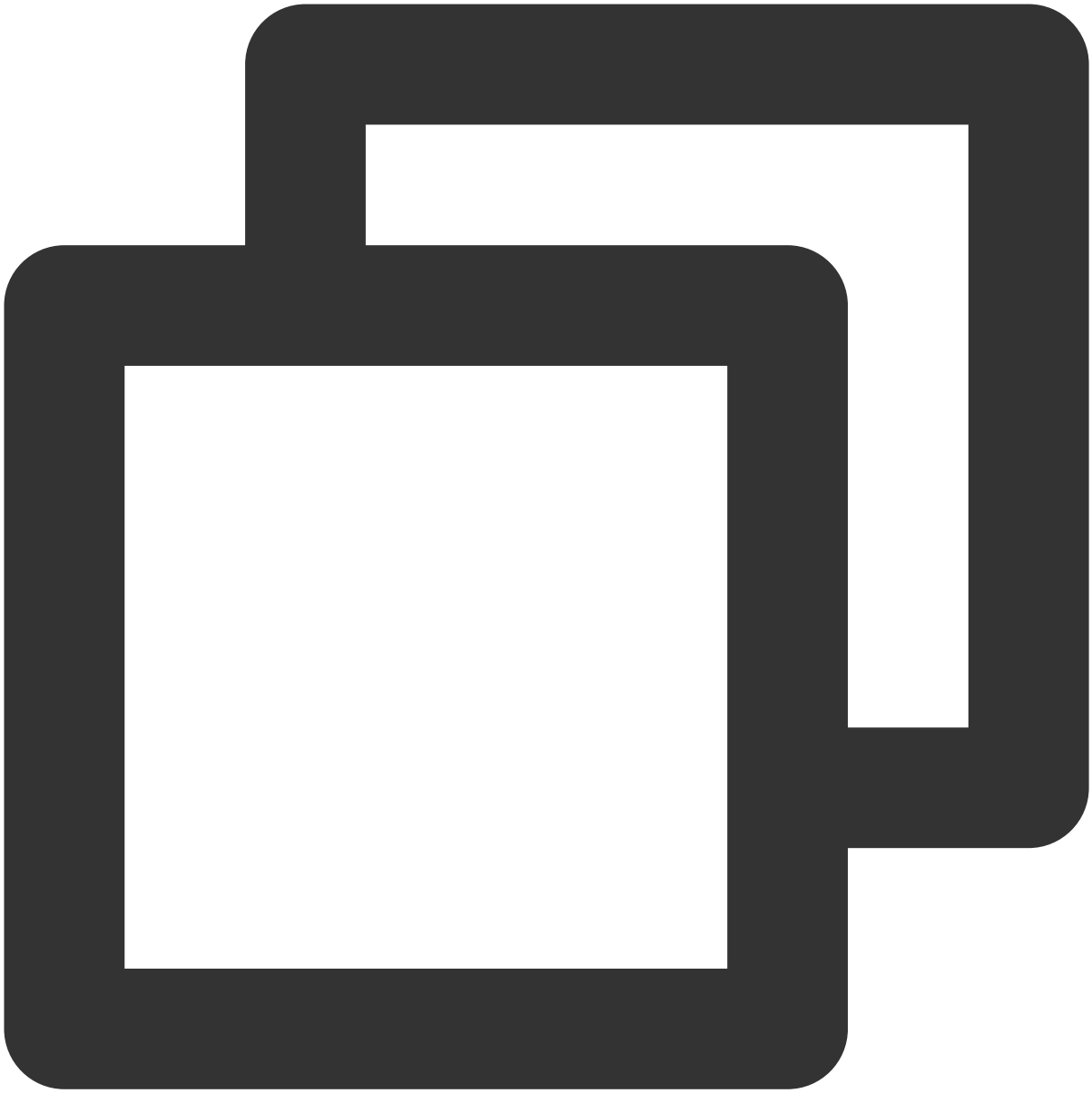
```
UNHEX(<str> string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Convert the hexadecimal str to binary.

Return type: binary

Example:



```
> select unhex('74656E63656E74');
 tencent
```

UPPER

Function statement:

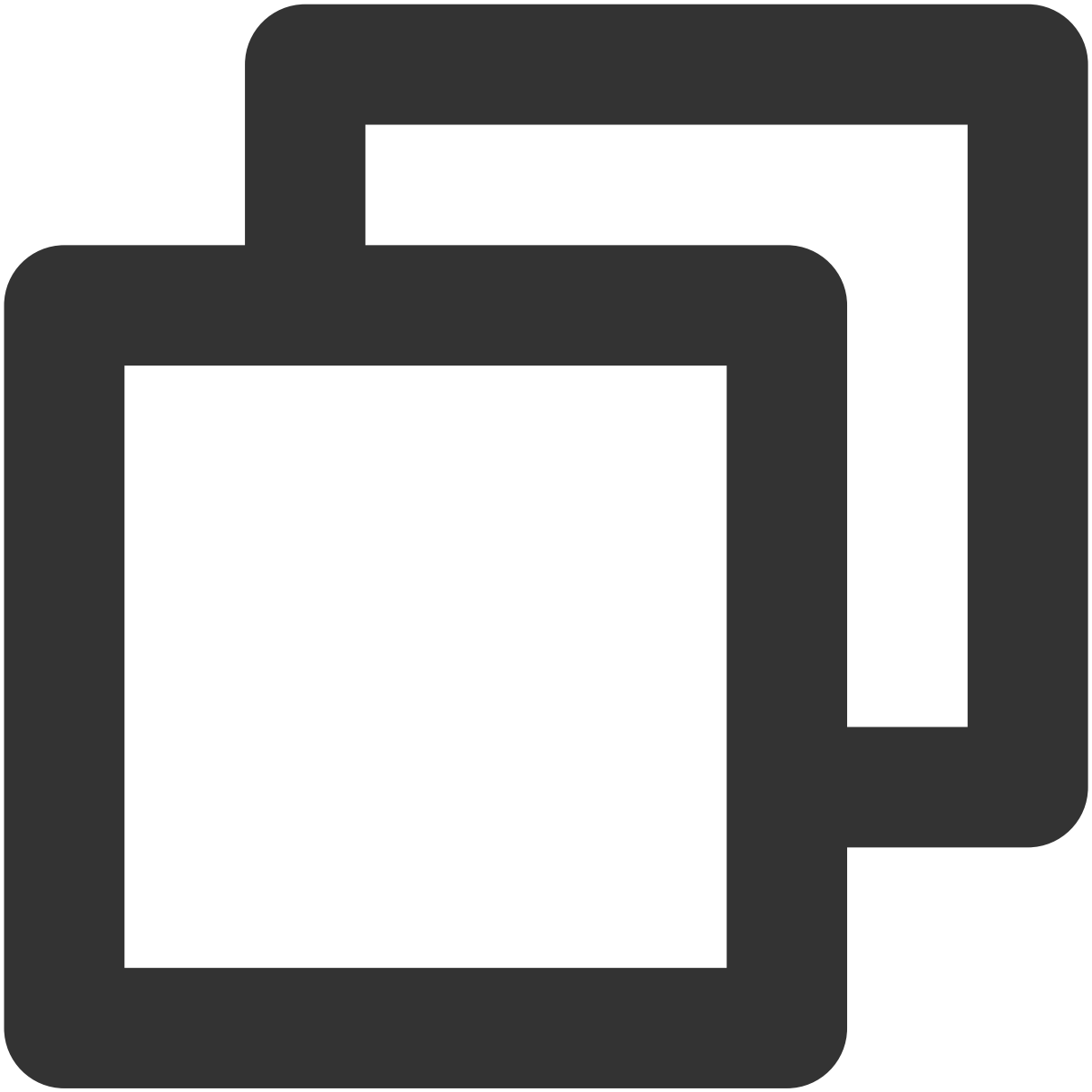
```
UPPER(<str> string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return str with all characters converted to uppercase.

Return type: string

Example:



```
> SELECT upper('tencent');
TENCENT
```

UUID

Function statement:

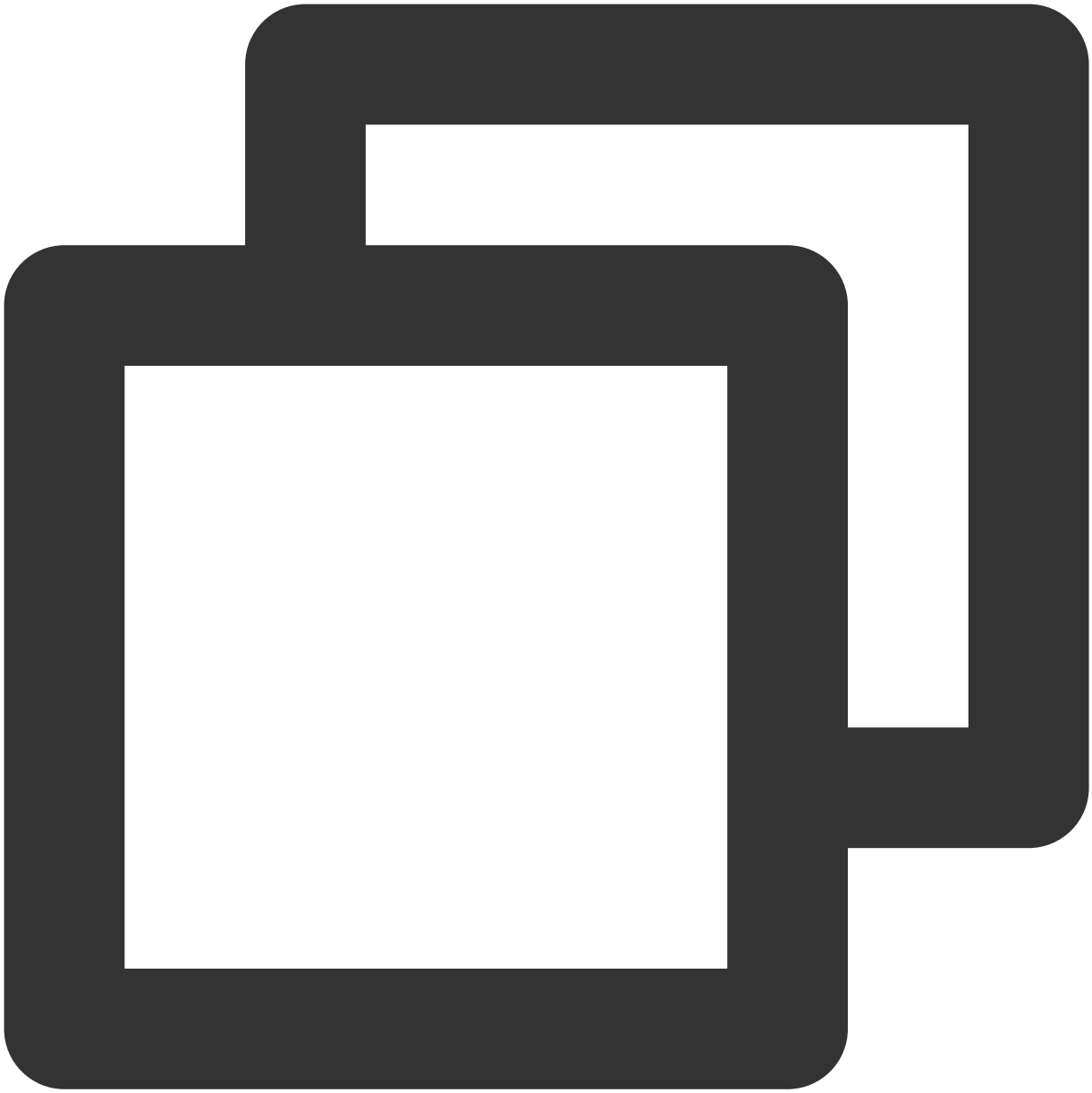
```
UUID()
```

Supported engines: SparkSQL and Presto

Usage instructions: Return a 36-character UUID.

Return type: string

Example:



```
> SELECT uuid();
 46707d92-02f4-4817-8116-a4c3b23e6266
```

XPATH

Function statement:

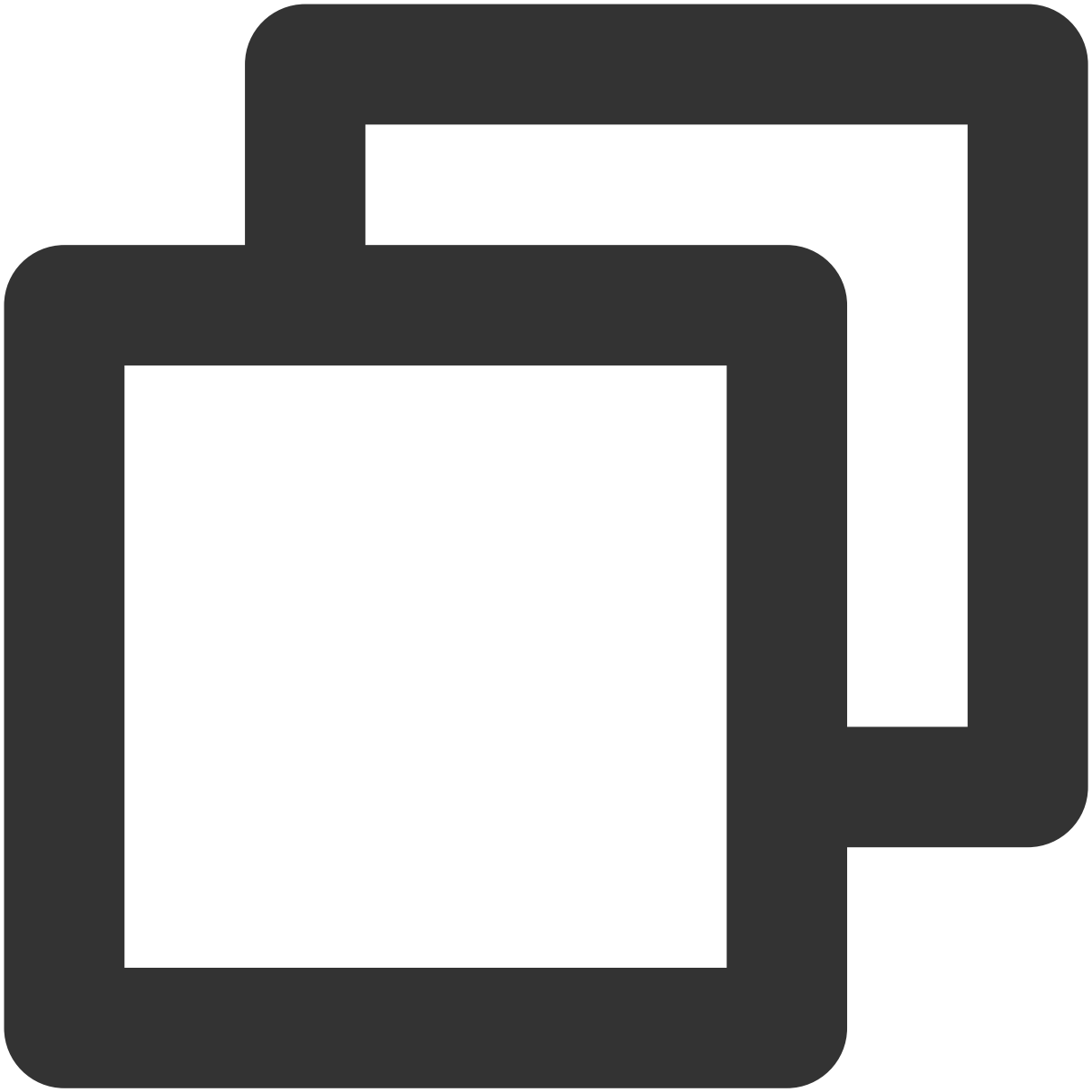
```
XPATH(<xml> string, <xpath> string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return an array of strings that matches the XPath expression in the XML node.

Return type: array <string>

Example:



```
> SELECT xpath('<a><b>b1</b><b>b2</b><b>b3</b><c>c1</c><c>c2</c></a>','a/b/text()')
 ["b1","b2","b3"]
```

XPATH_BOOLEAN

Function statement:

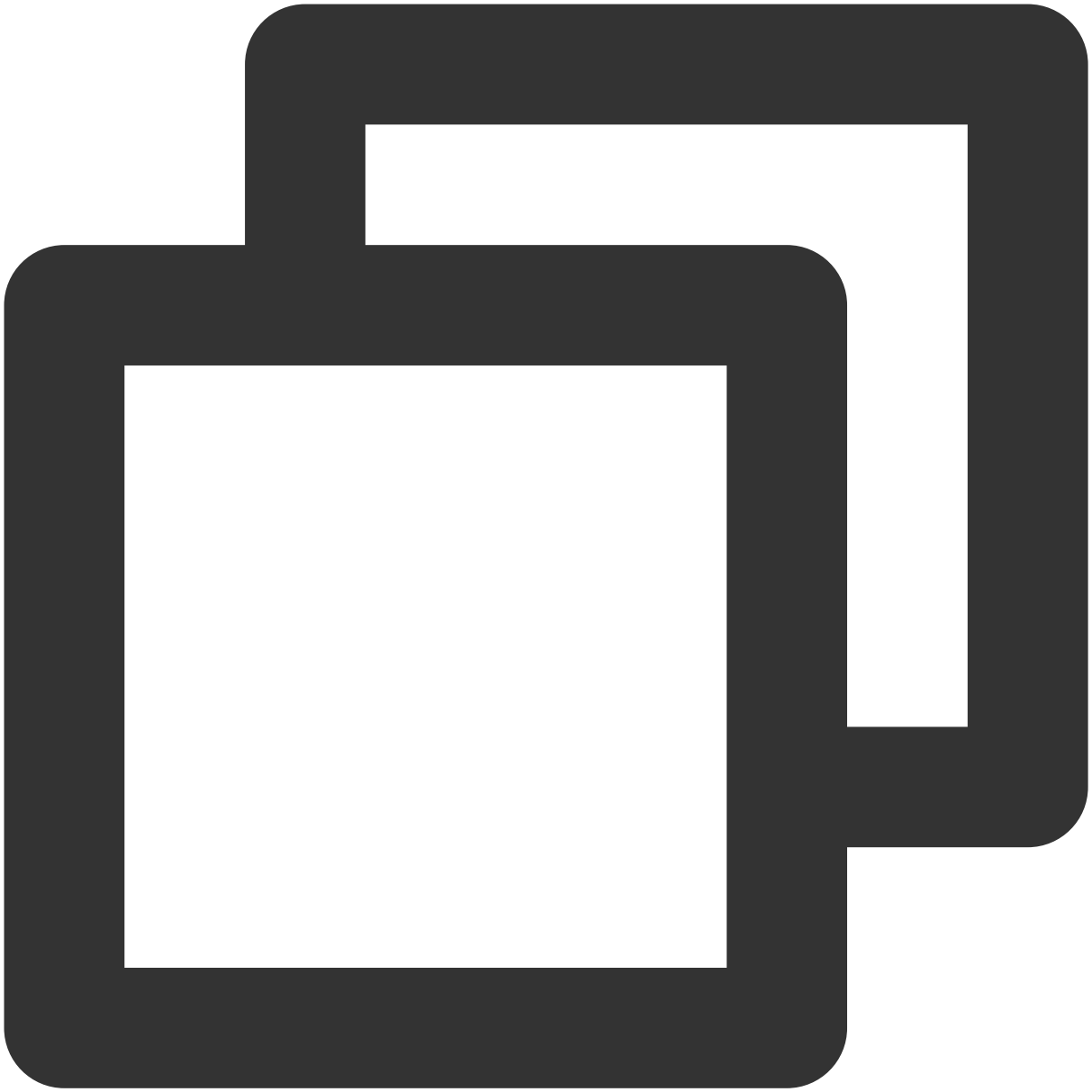
```
XPATH_BOOLEAN(<xml> string, <xpath> string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return true if the evaluation of the XPath expression is true or if a matching node is found.

Return type: boolean

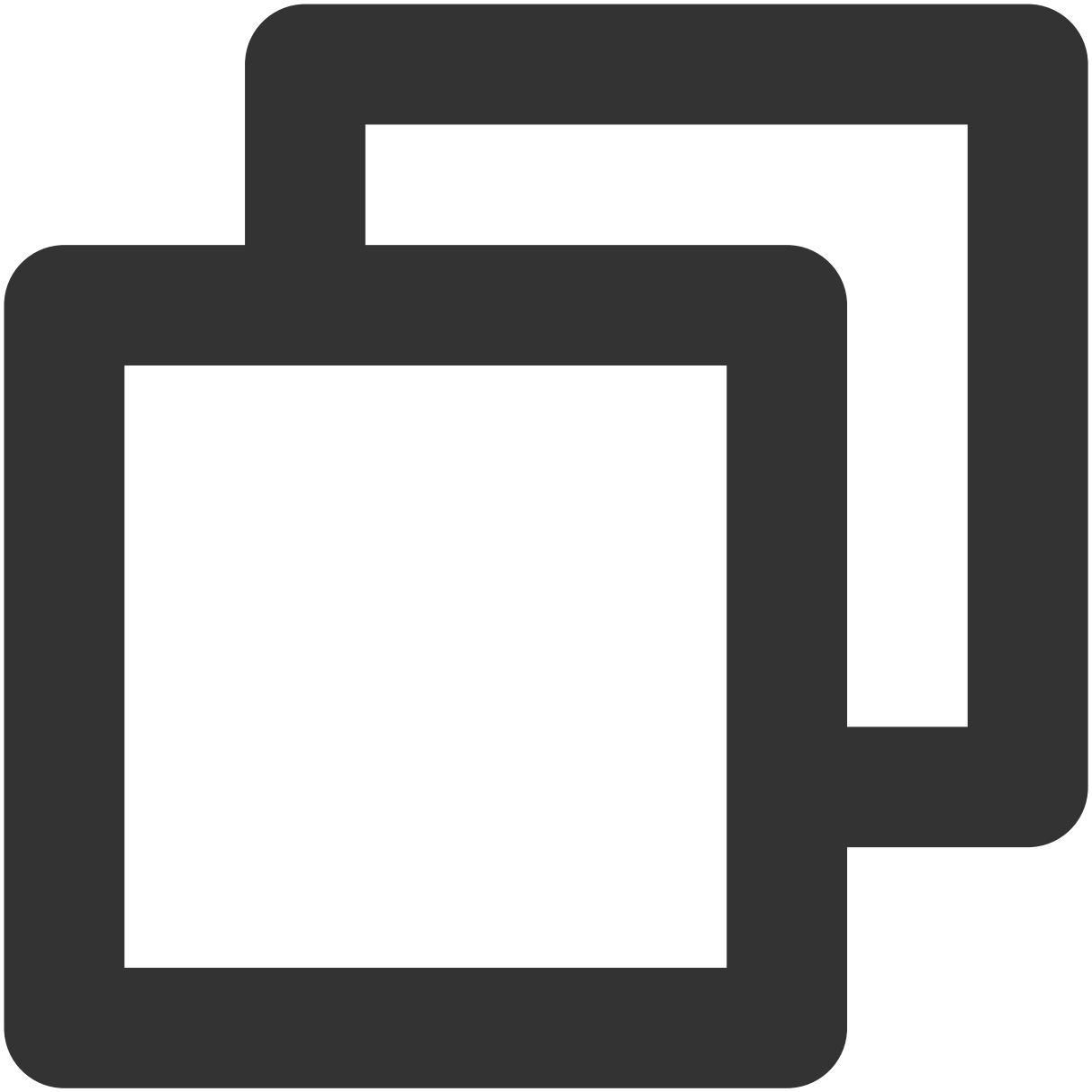
Example:



```
> SELECT xpath_boolean('<a><b>1</b></a>','a/b');
 true
```

XPATH_DOUBLE

Function statement:



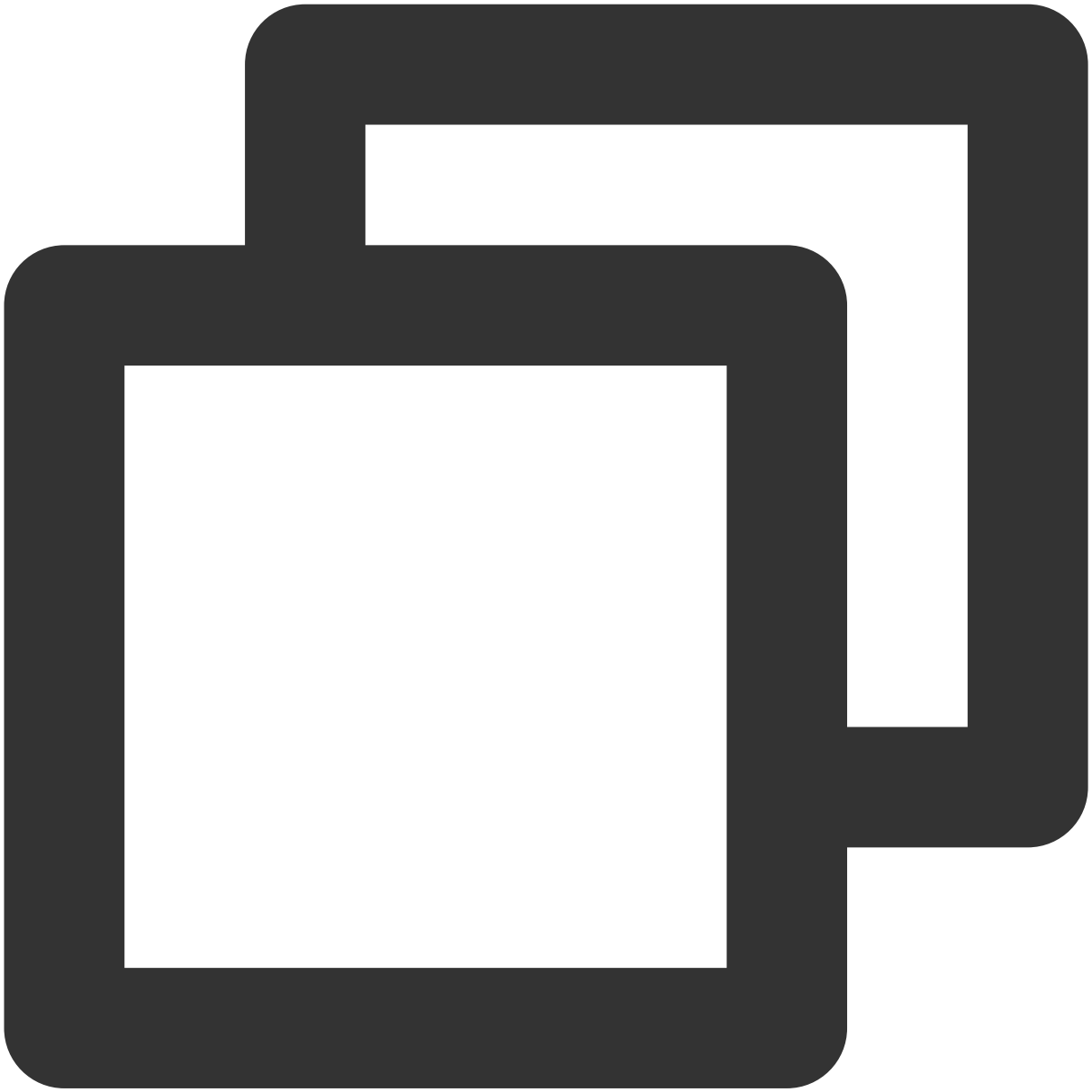
```
XPATH_DOUBLE(<xml> string, <xpath> string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return a double value; if no match is found, return zero; if a match is found but the value is not a number, return NaN.

Return type: double

Example:



```
> SELECT xpath_double('<a><b>1</b><b>2</b></a>', 'sum(a/b)');
 3.0
```

XPATH_NUMBER

Function statement:

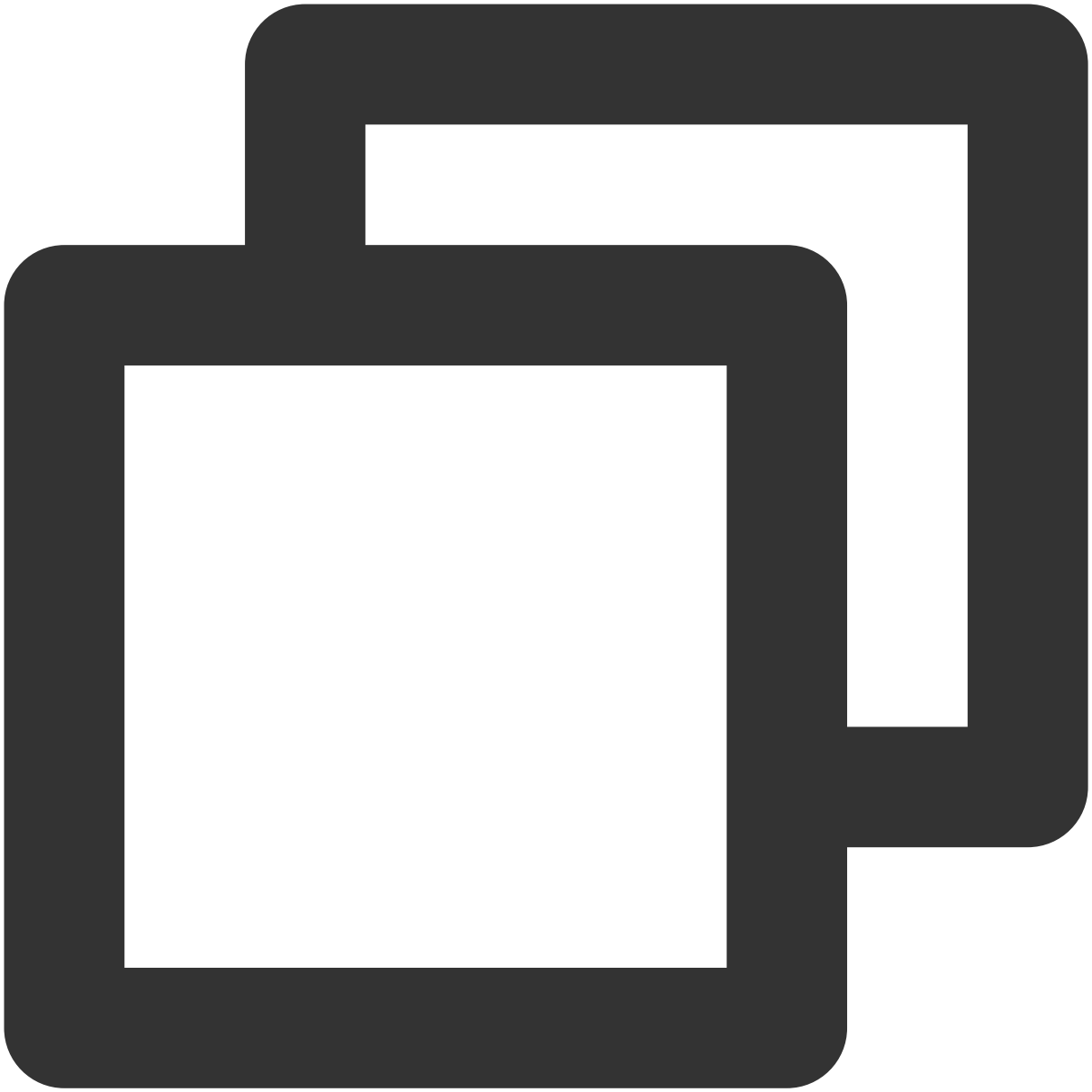
```
XPATH_NUMBER(<xml> string, <xpath> string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return a double value; if no match is found, return zero; if a match is found but the value is not a number, return NaN.

Return type: double

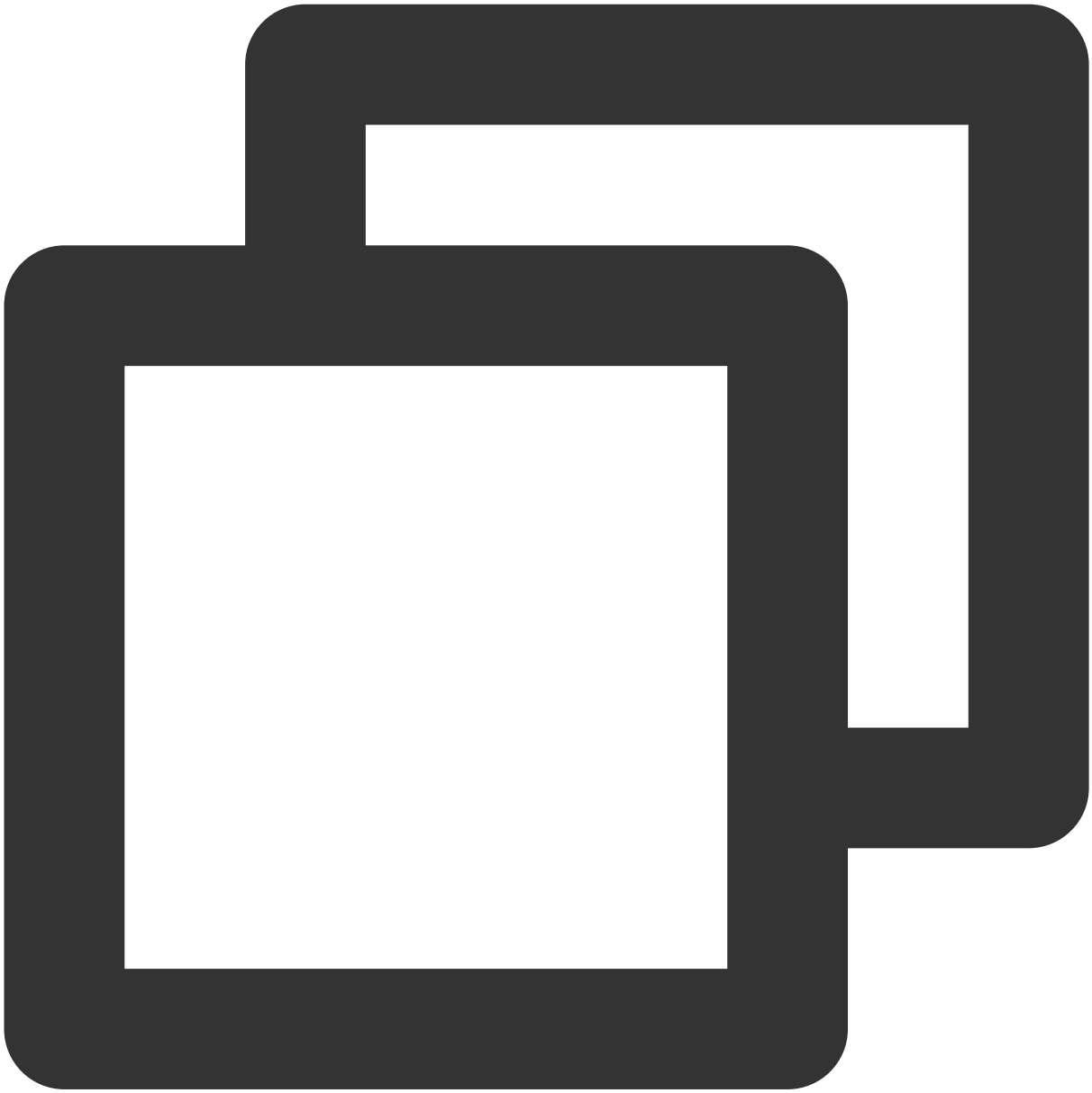
Example:



```
> SELECT xpath_number('<a><b>1</b><b>2</b></a>', 'sum(a/b)');
 3.0
```

XPATH_FLOAT

Function statement:



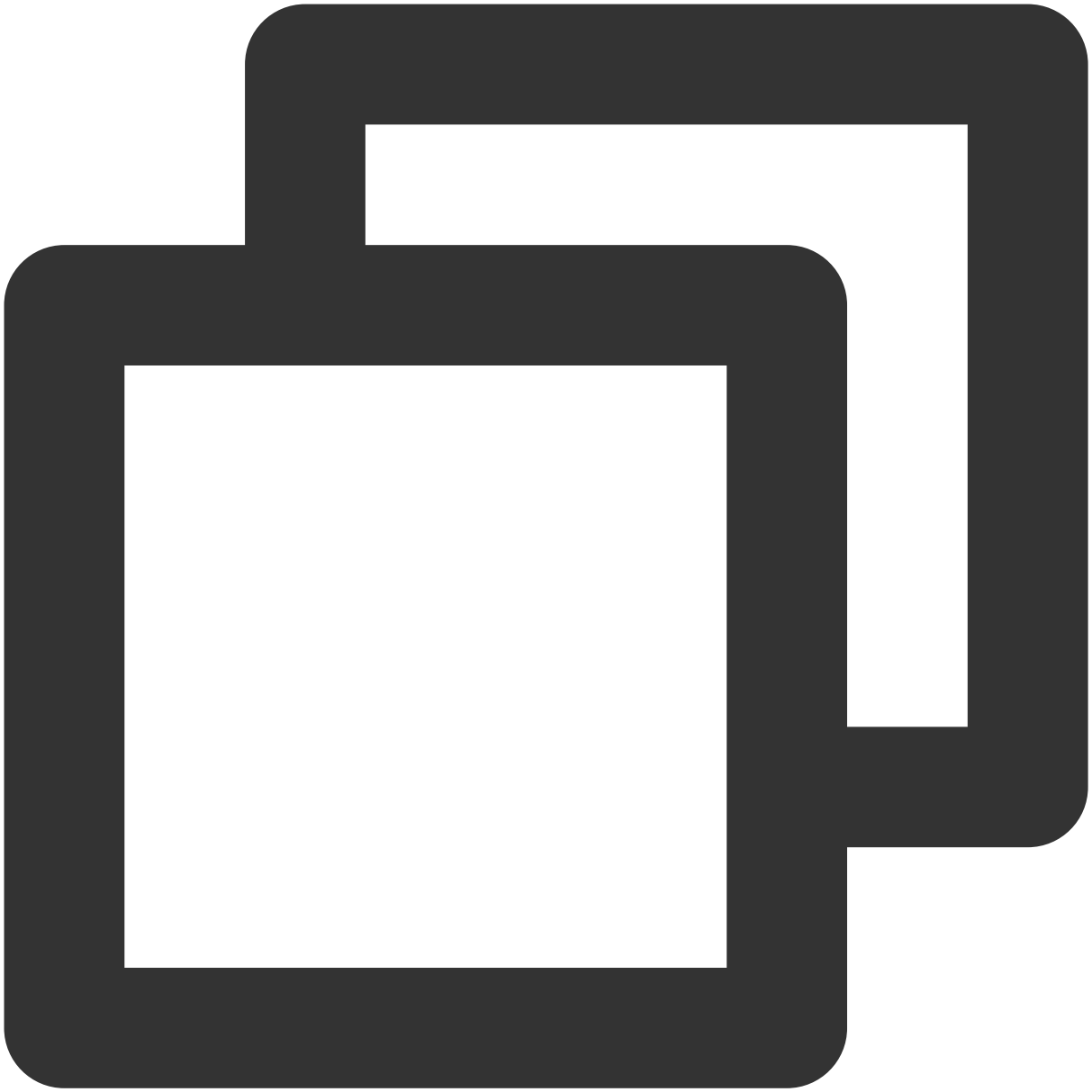
```
XPATH_FLOAT(<xml> string, <xpath> string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return a float value; if no match is found, return zero; if a match is found but the value is not a number, return NaN.

Return type: float

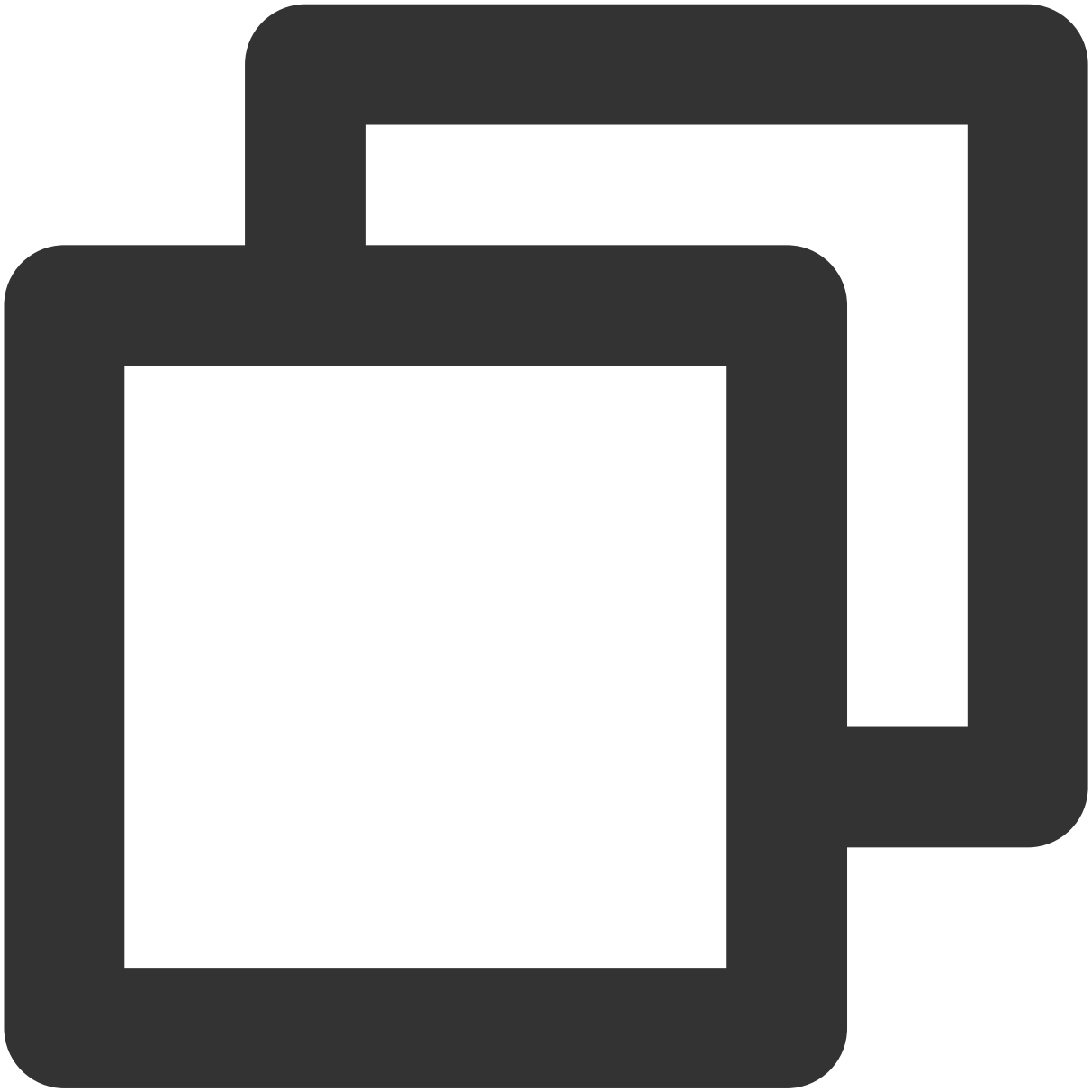
Example:



```
> SELECT xpath_float('<a><b>1</b><b>2</b></a>', 'sum(a/b)');
 3.0
```

XPATH_INT

Function statement:



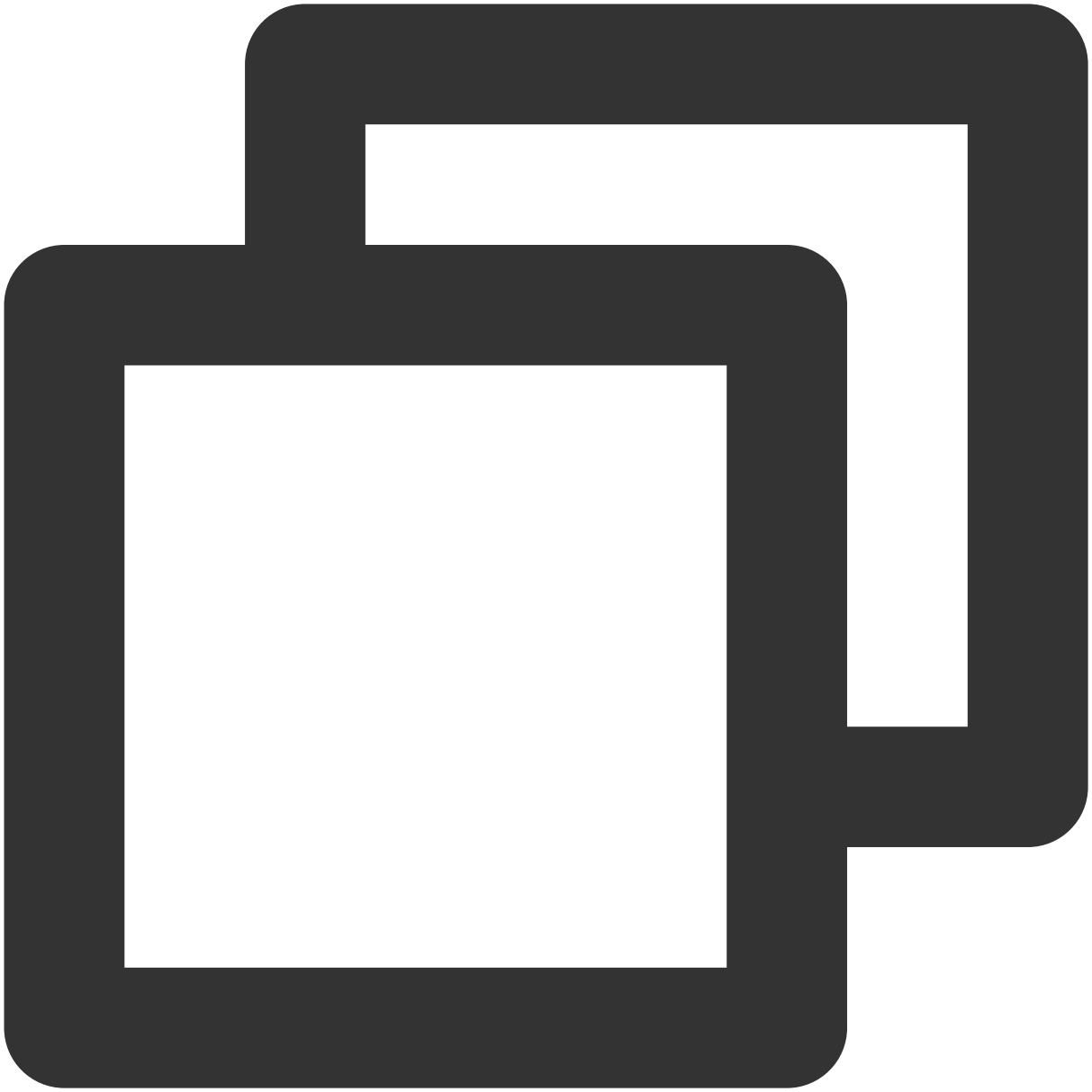
```
XPATH_INT(<xml> string, <xpath> string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return an int value; if no match is found, return zero; if a match is found but the value is not a number, return NaN.

Return type: integer

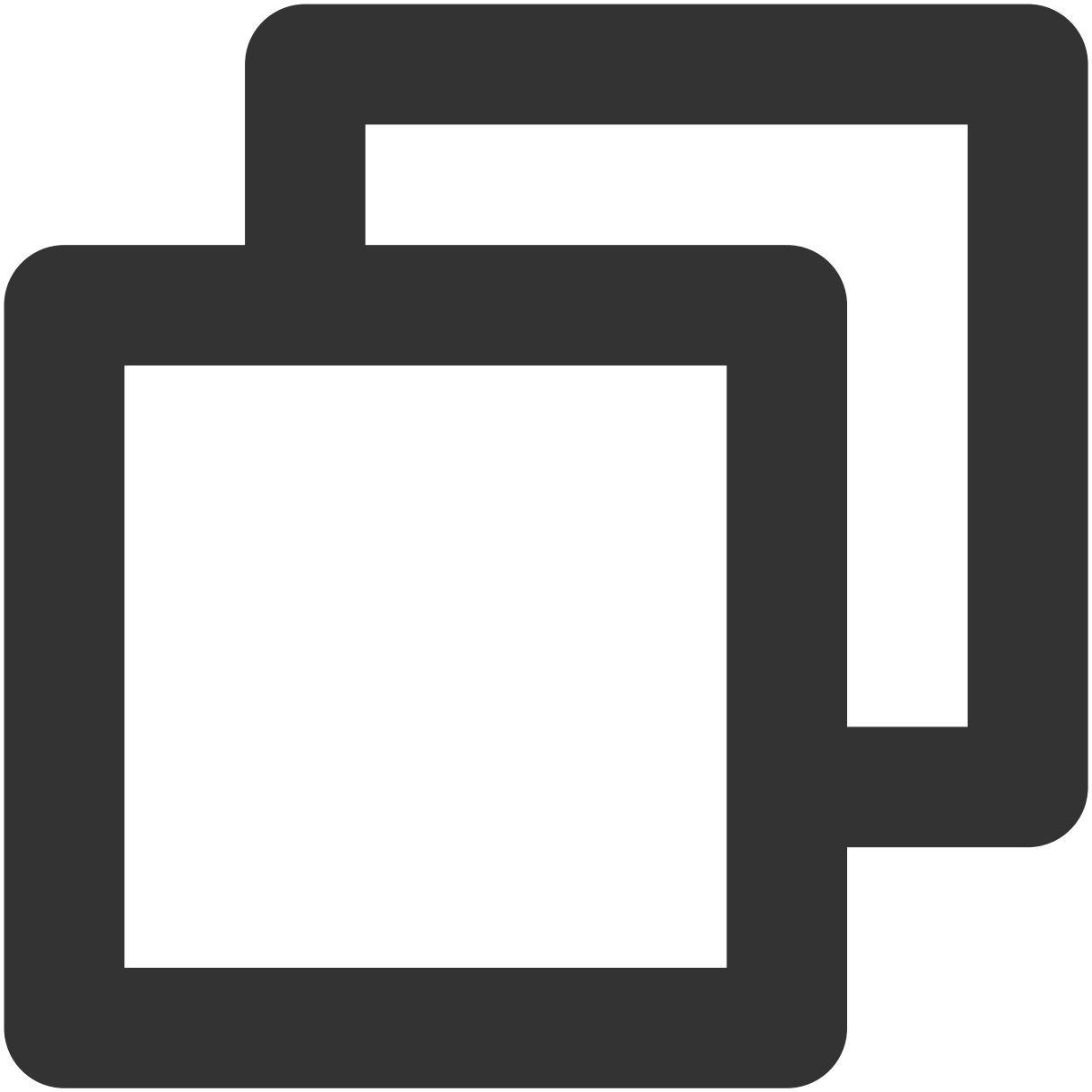
Example:



```
> SELECT xpath_int('<a><b>1</b><b>2</b></a>', 'sum(a/b)');
 3
```

XPATH_LONG

Function statement:



```
XPATH_LONG(<xml> string, <xpath> string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return a bigint value; if no match is found, return zero; if a match is found but the value is not a number, return NaN.

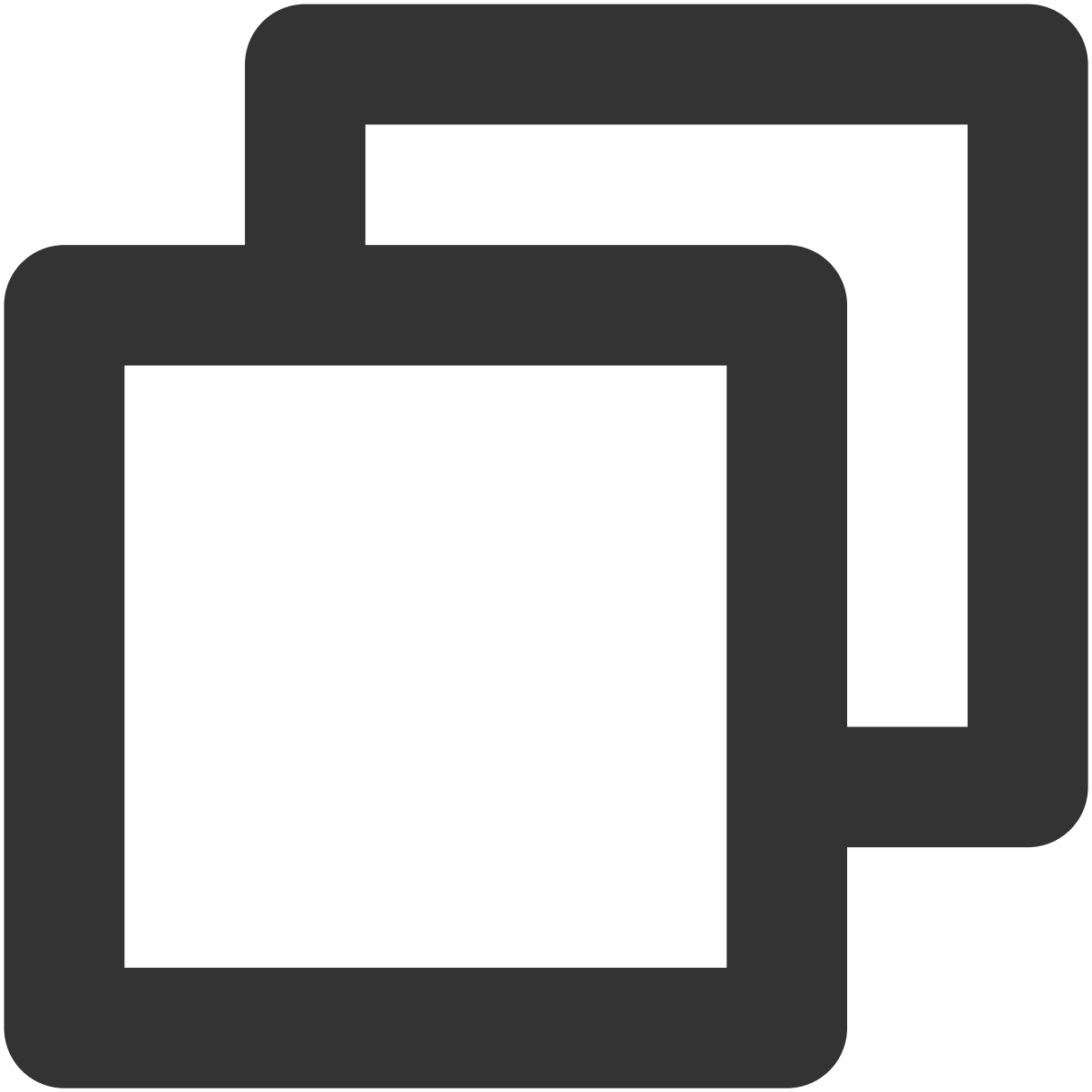
Return type: bigint

Example:

```
> SELECT xpath_long('<a><b>1</b><b>2</b></a>', 'sum(a/b)');
 3
```

XPATH_SHORT

Function statement:



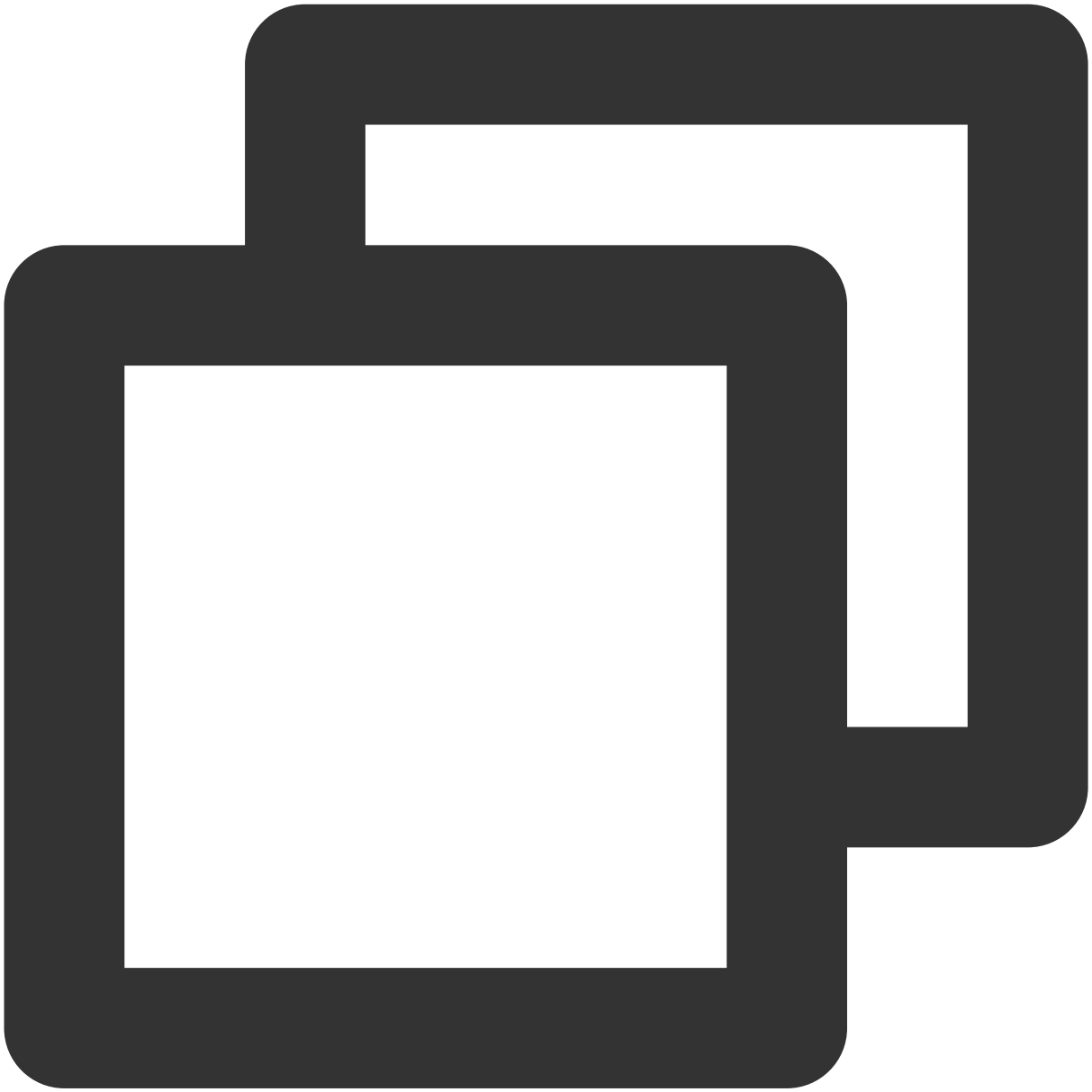
```
XPATH_SHORT(<xml> string, <xpath> string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the value of short type; if no match is found, return zero; if a match is found but the value is not a number, return NaN.

Return type: short

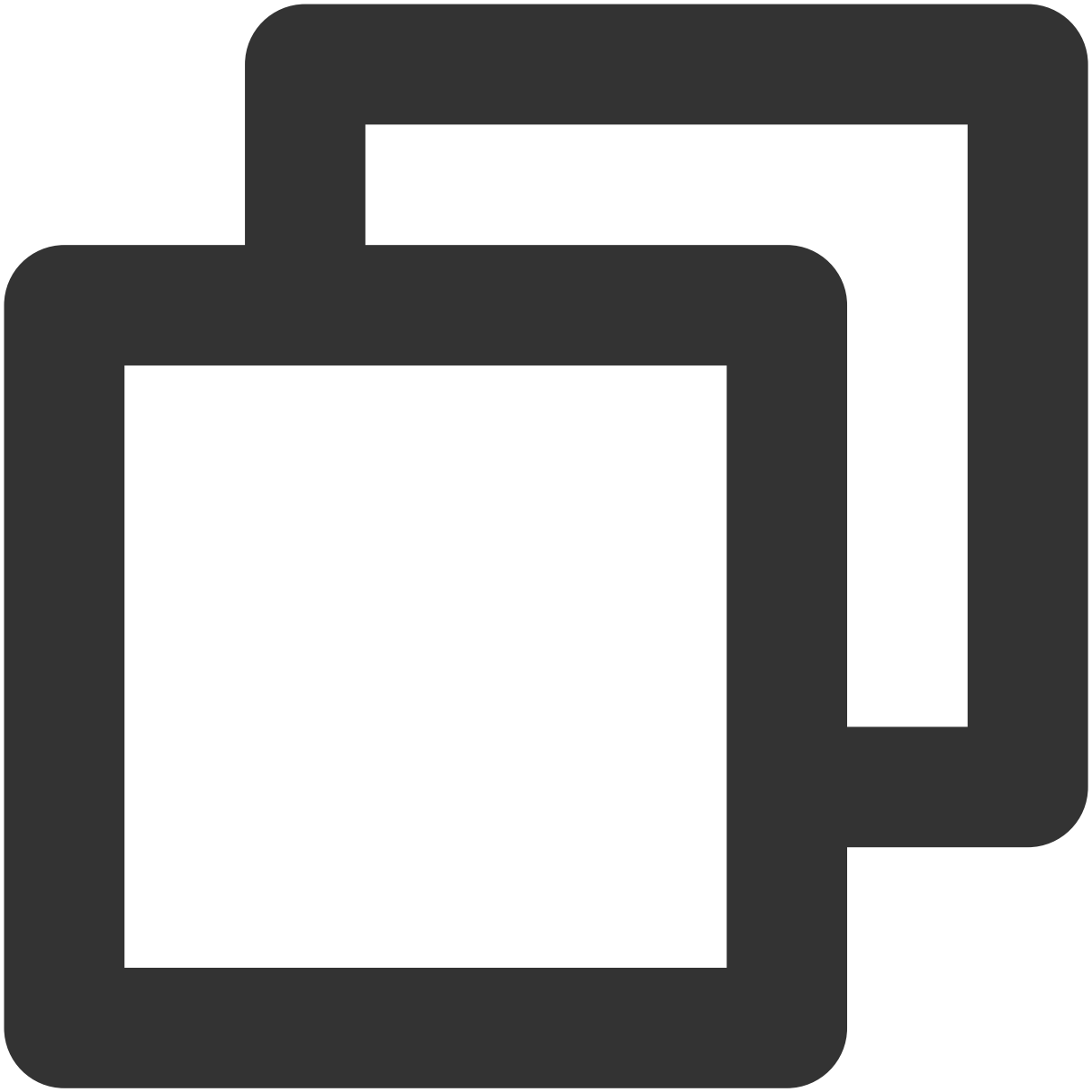
Example:



```
> SELECT xpath_short('<a><b>1</b><b>2</b></a>', 'sum(a/b)');
 3
```

XPATH_STRING

Function statement:



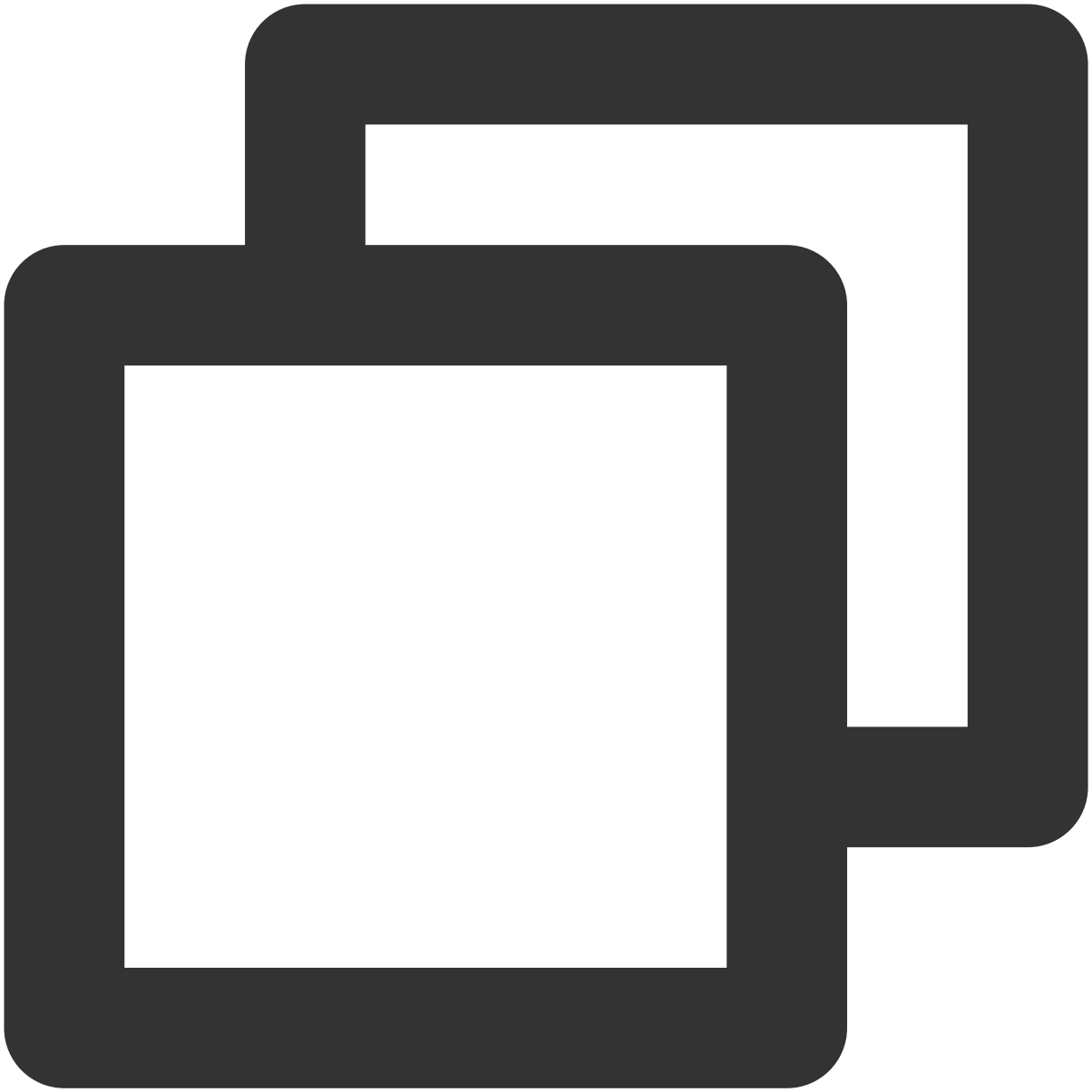
```
XPATH_STRING(<xml> string, <xpath> string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the text content of the first XML node that matches the XPath expression.

Return type: string

Example:



```
> SELECT xpath_string('<a><b>b</b><c>cc</c></a>','a/c');
 cc
```

REGEXP_EXTRACT

Function statement:

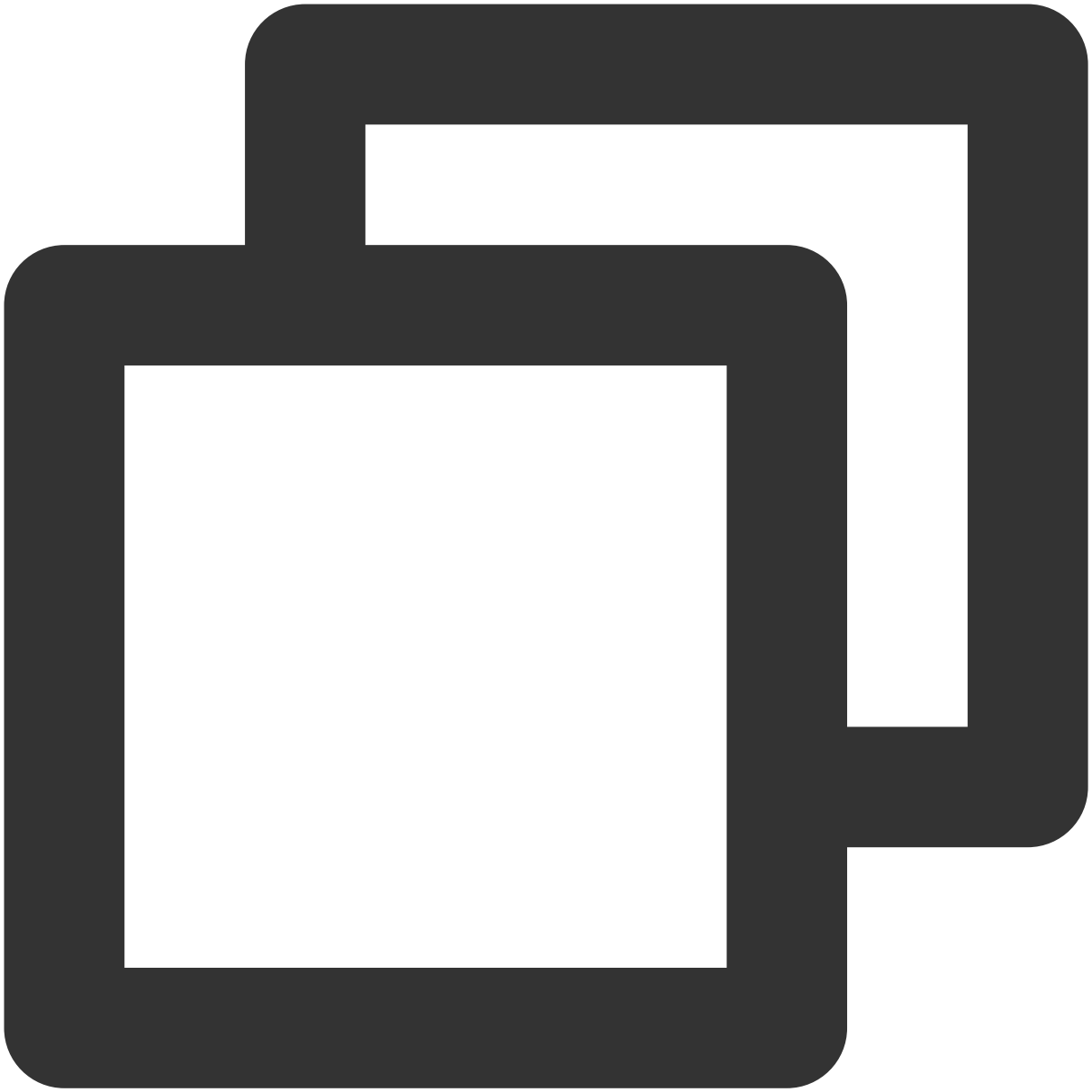
```
REGEXP_EXTRACT(<str> string, <regexp> string[, <idx> integer])
```

Supported engines: SparkSQL and Presto

Usage instructions: Extract the first string in str that matches the regexp expression and corresponds to the regex group show idx.

Return type: string

Example:



```
> SELECT regexp_extract('100-200', '(\\\\d+)-(\\\\d+)', 1);
 100
```

REGEXP_EXTRACT_ALL

Function statement:

```
REGEXP_EXTRACT_ALL(<str> string, <regexp> string[, <idx> integer])
```

Supported engines: SparkSQL and Presto

Usage instructions: Extract all strings in str that match the regexp expression and correspond to the regex hidden Index.

Return type: array <string>

Example:

```
> SELECT regexp_extract_all('100-200, 300-400', '(\\\\d+)-(\\\\d+)', 1);
 ["100","300"]
```

REGEXP_REPLACE

Function statement:

```
REGEXP_REPLACE(<str> string, <regexp> string, <rep> string[, <position> integer])
```

Supported engines: SparkSQL and Presto

Usage instructions: Replace all substrings in str that match the regexp with rep.

Return type: string

Example:

```
> SELECT regexp_replace('100-200', '(\\\\d+)', 'num');
 num-num
```

REGEXP_LIKE

Function statement:

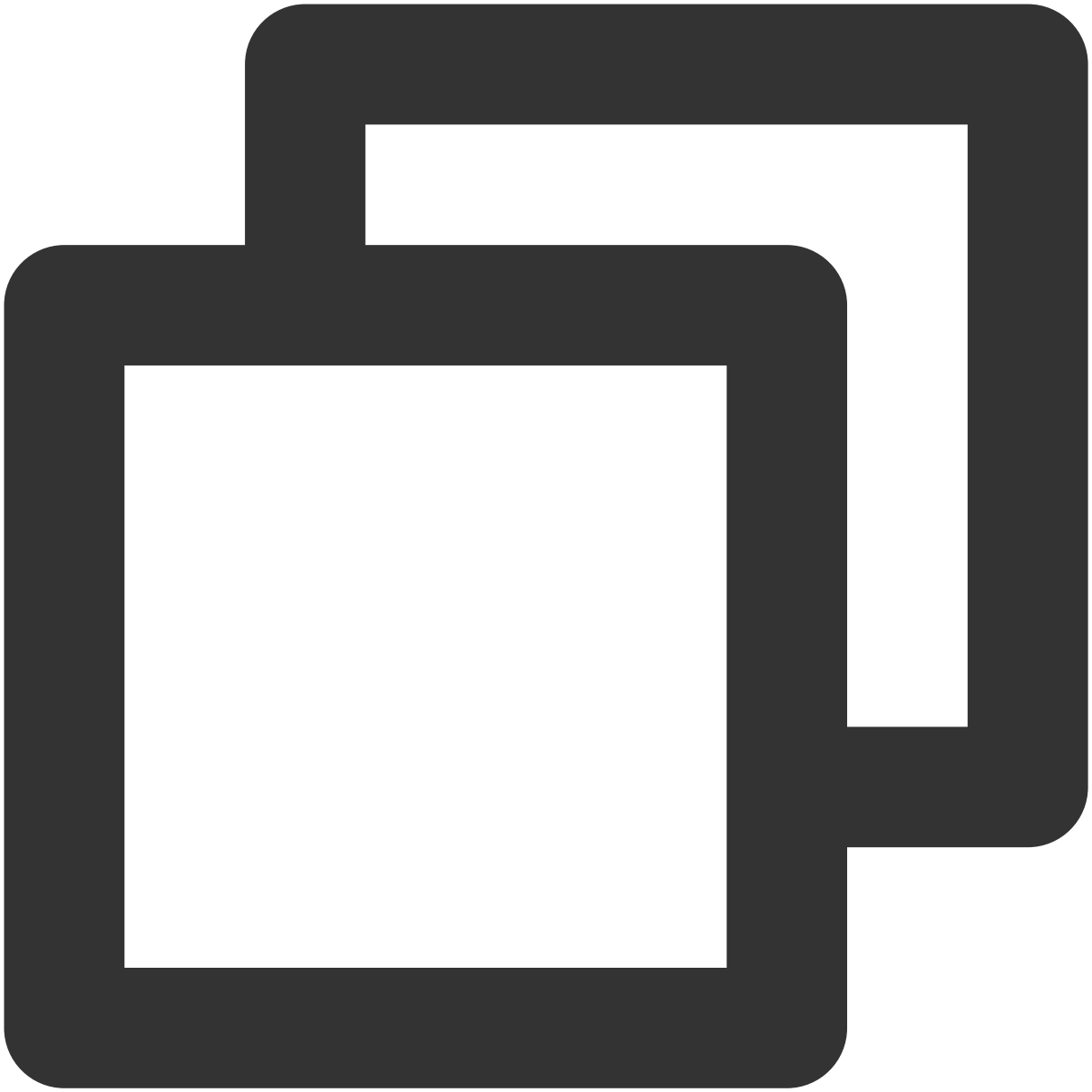
```
REGEXP_LIKE(<str> string, <regexp> string)
```

Supported engine: SparkSQL

Usage instructions: Return true if the string matches the regular expression; otherwise, return false.

Return type: boolean

Example:



```
> SELECT regexp_like('%SystemDrive%\\Users\\John', '%SystemDrive%\\Users.*');
 true
```

REGEXP

Function statement:

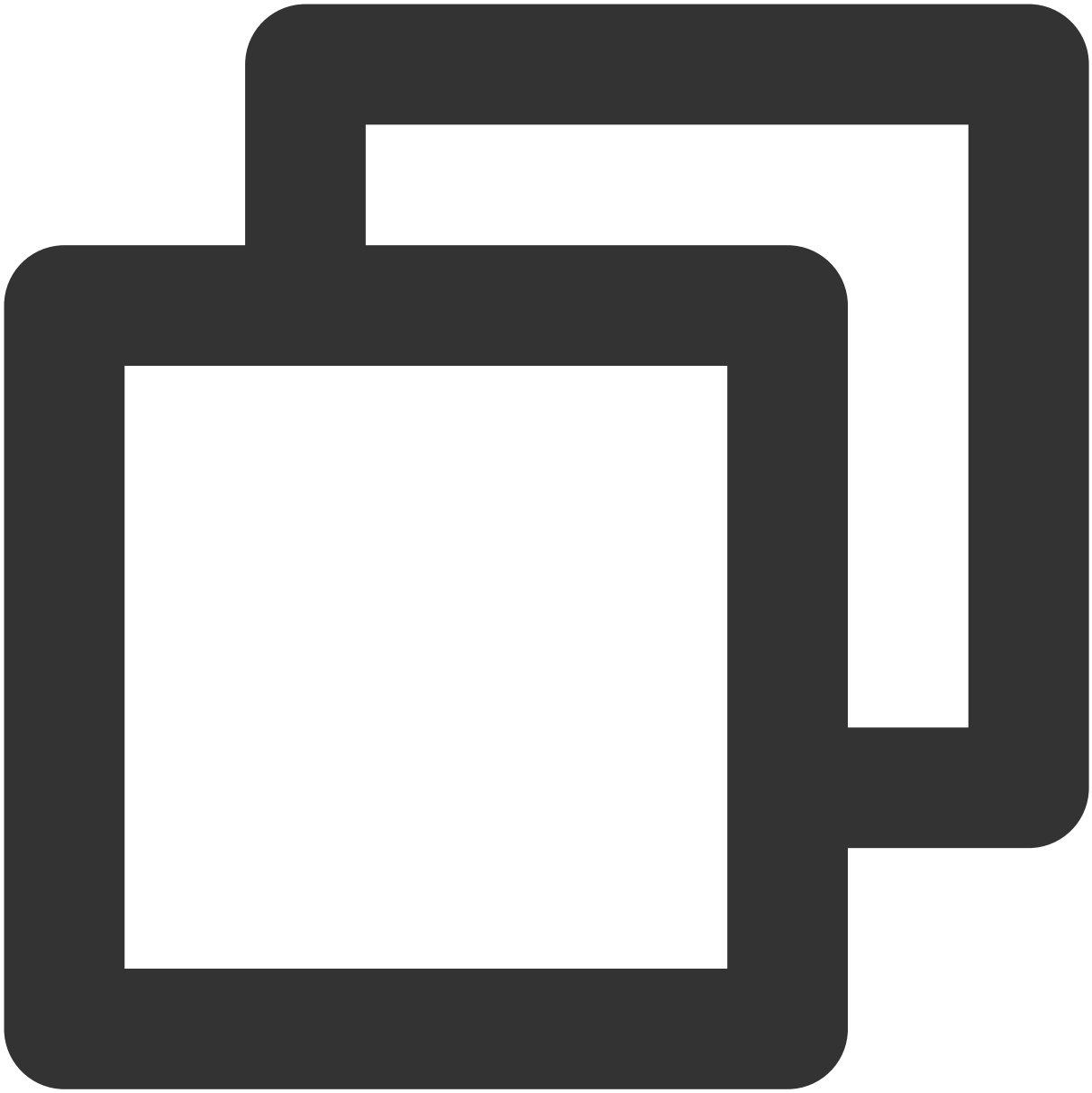
```
REGEXP(<str> string, <regexp> string)
```

Supported engine: SparkSQL

Usage instructions: Return true if the string matches the regular expression; otherwise, return false.

Return type: boolean

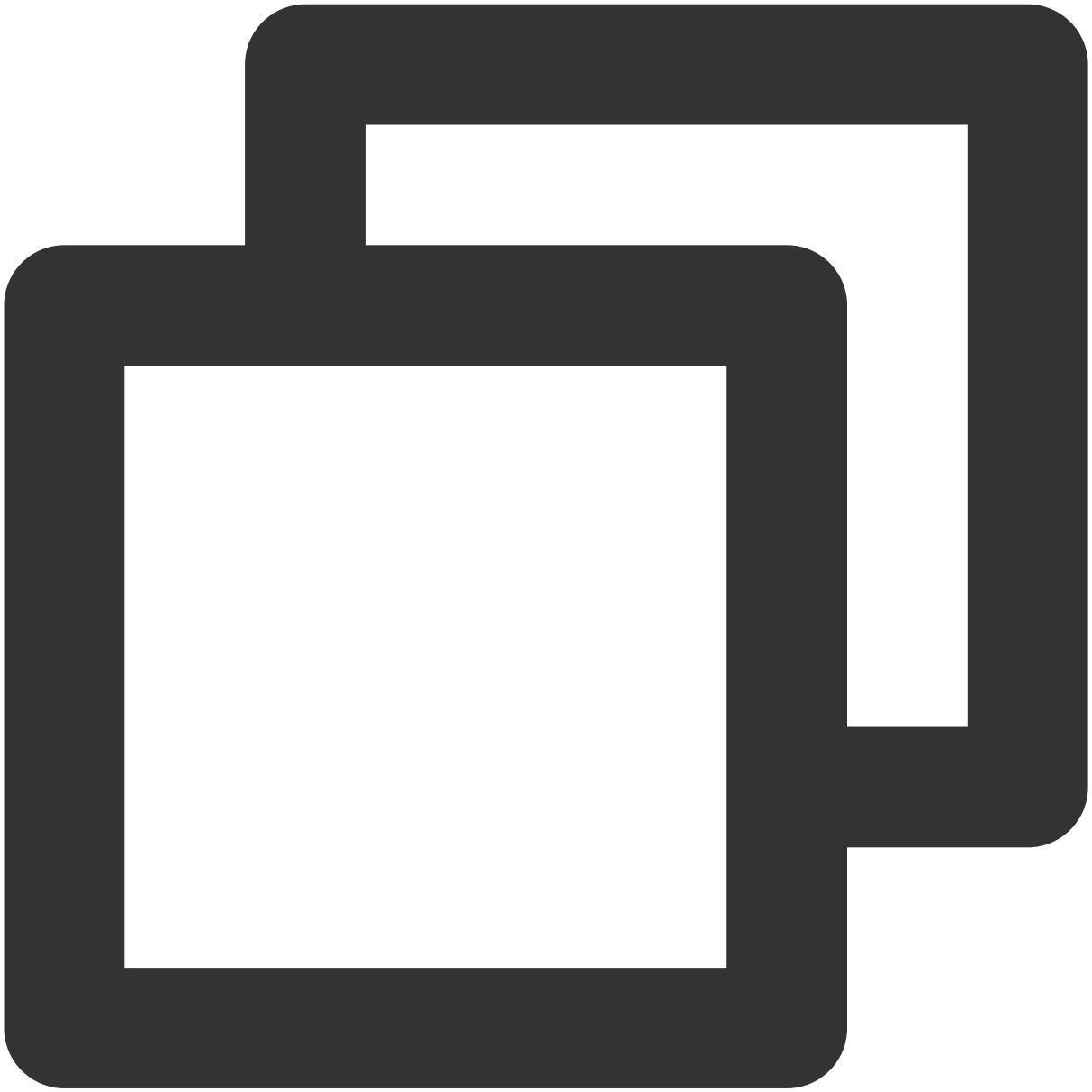
Example:



```
> SELECT regexp('%SystemDrive%\\Users\\John', '%SystemDrive%\\Users.*');
true
```

CONCAT

Function statement:



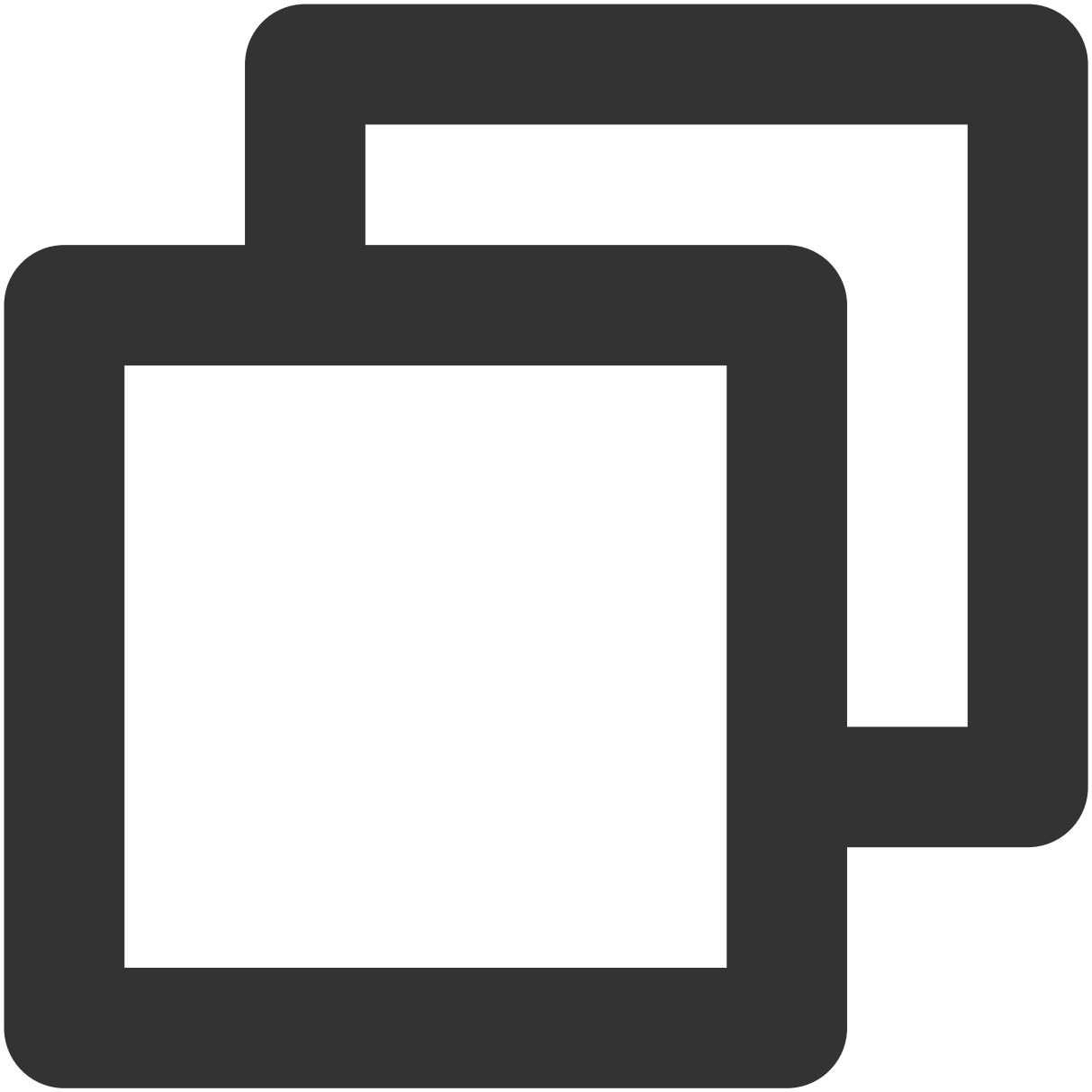
```
CONCAT(<s1> string, <s2> string, ...)
```

Supported engines: SparkSQL and Presto

Usage instructions: Concatenate s1, s2, ...

Return type: string

Example:



```
> SELECT concat('Spark', 'SQL');
 SparkSQL
```

STR_TO_MAP

Function statement:

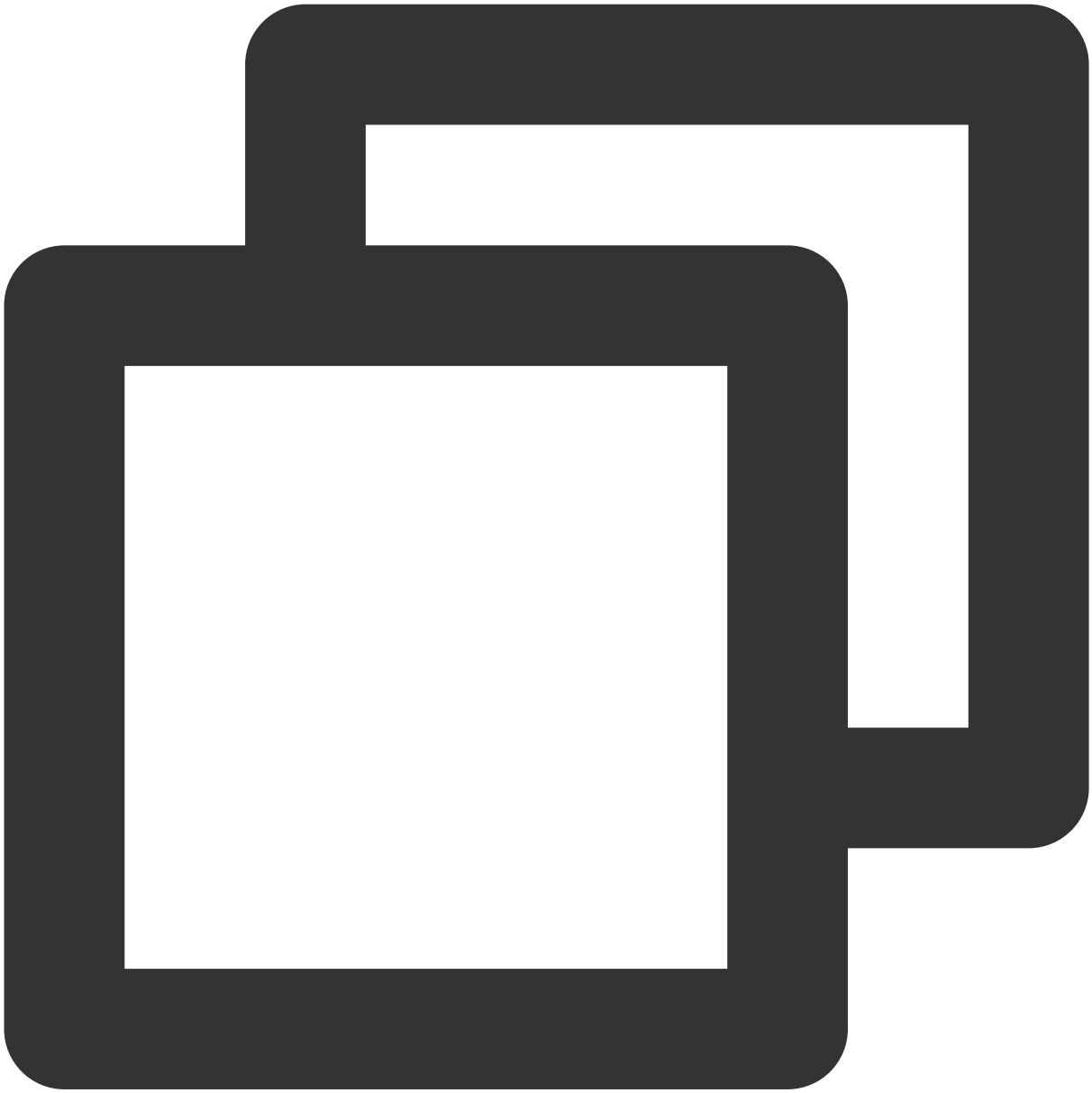
```
str_to_map(<text> string[, <pairDelim> string[, <keyValueDelim> string]])
```

Supported engines: SparkSQL and Presto

Usage instructions: Create a map by splitting text into key/value pairs using the delimiter. The default delimiter for pairDelim is a comma and for keyValueDelim is a colon. Both pairDelim and keyValueDelim are treated as regular expressions.

Return type: map <string, string>

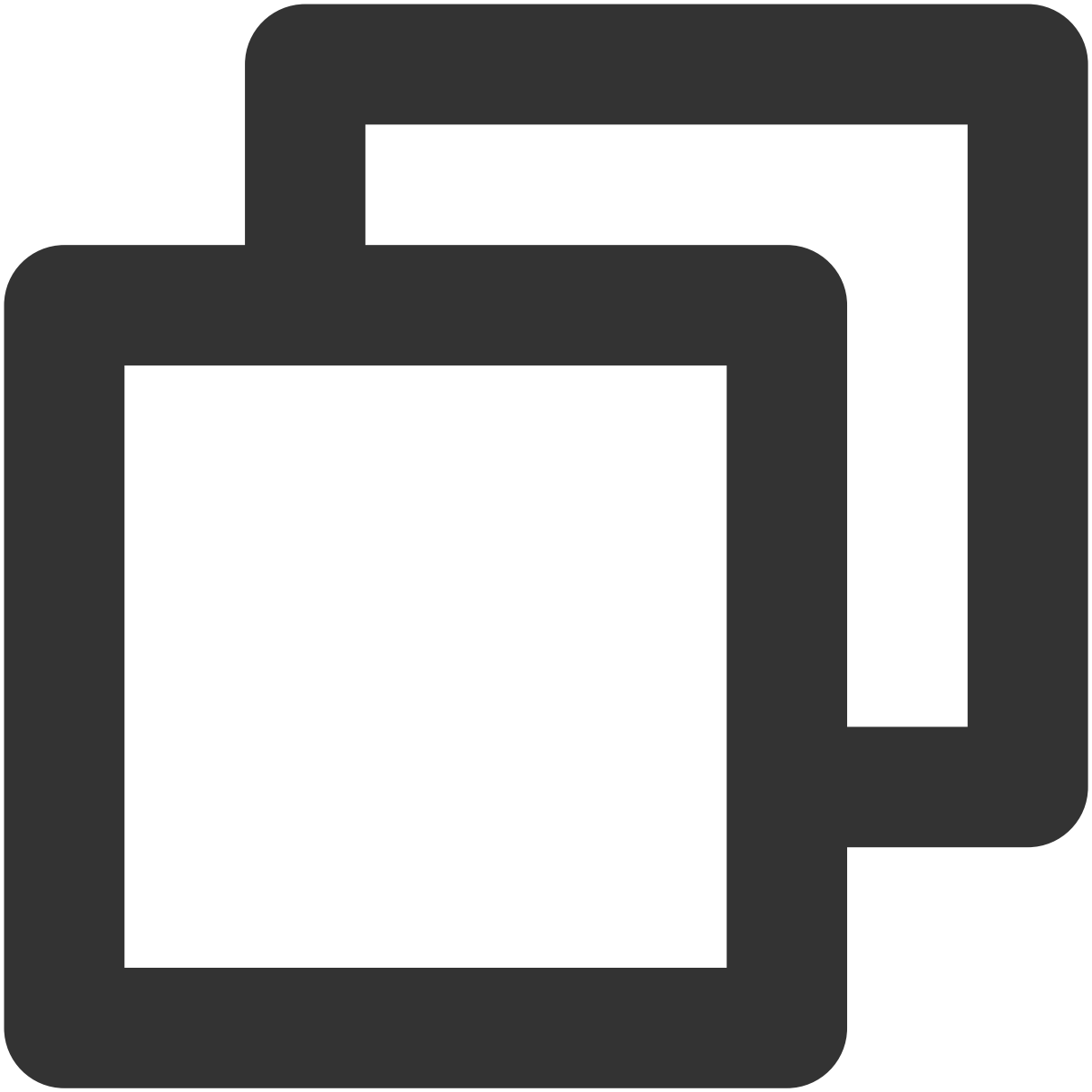
Example:



```
> SELECT str_to_map('a:1,b:2,c:3', ',', ':');
 {"a":"1","b":"2","c":"3"}
> SELECT str_to_map('a');
 {"a":null}
```

REVERSE

Function statement:



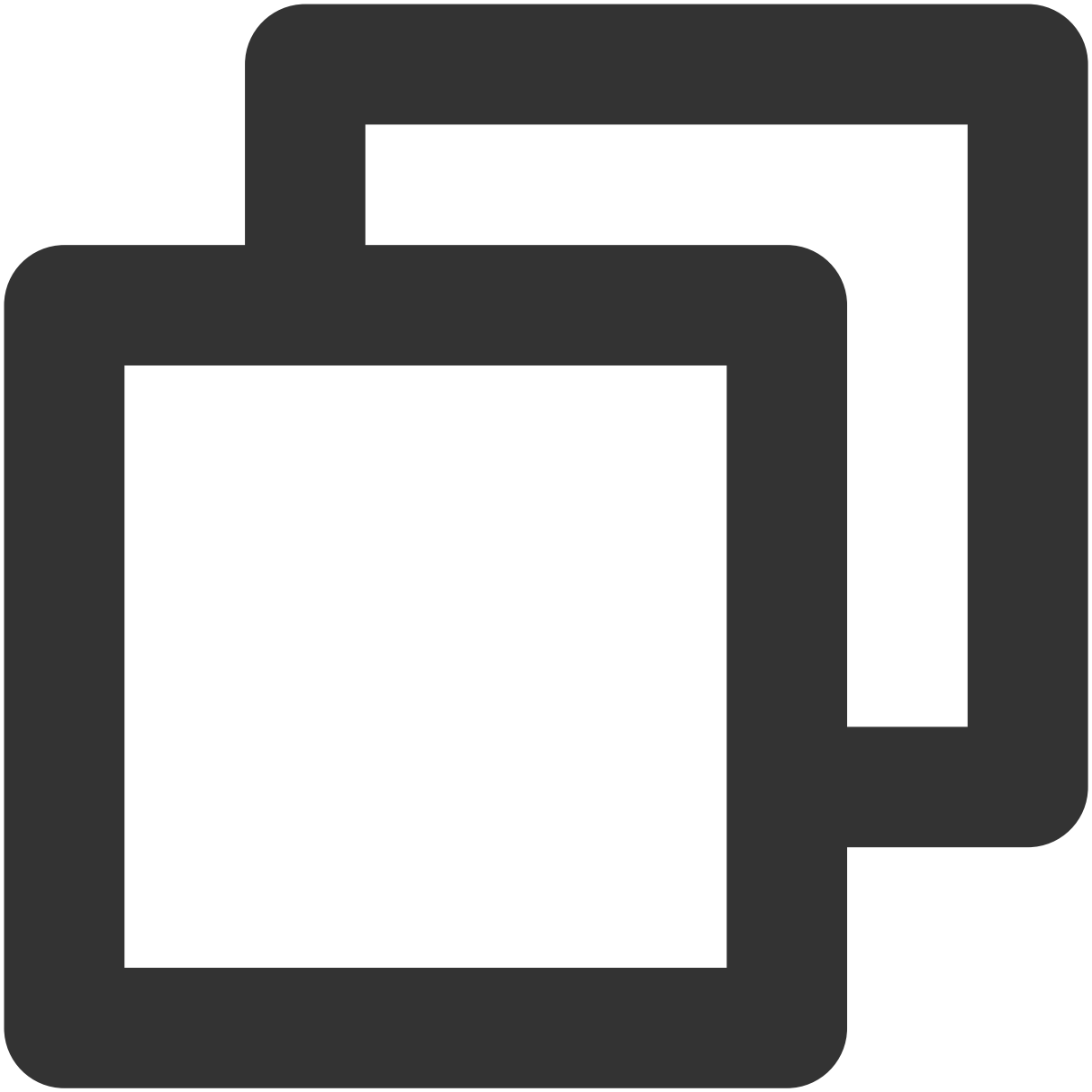
```
REVERSE(<str> string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return a reversed string.

Return type: string

Example:



```
> SELECT reverse('Spark SQL');
 LQS krapS
```

RLIKE

Function statement:

```
RLIKE(<str> string, <regexp> string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return true if str matches regexp; otherwise, return false.

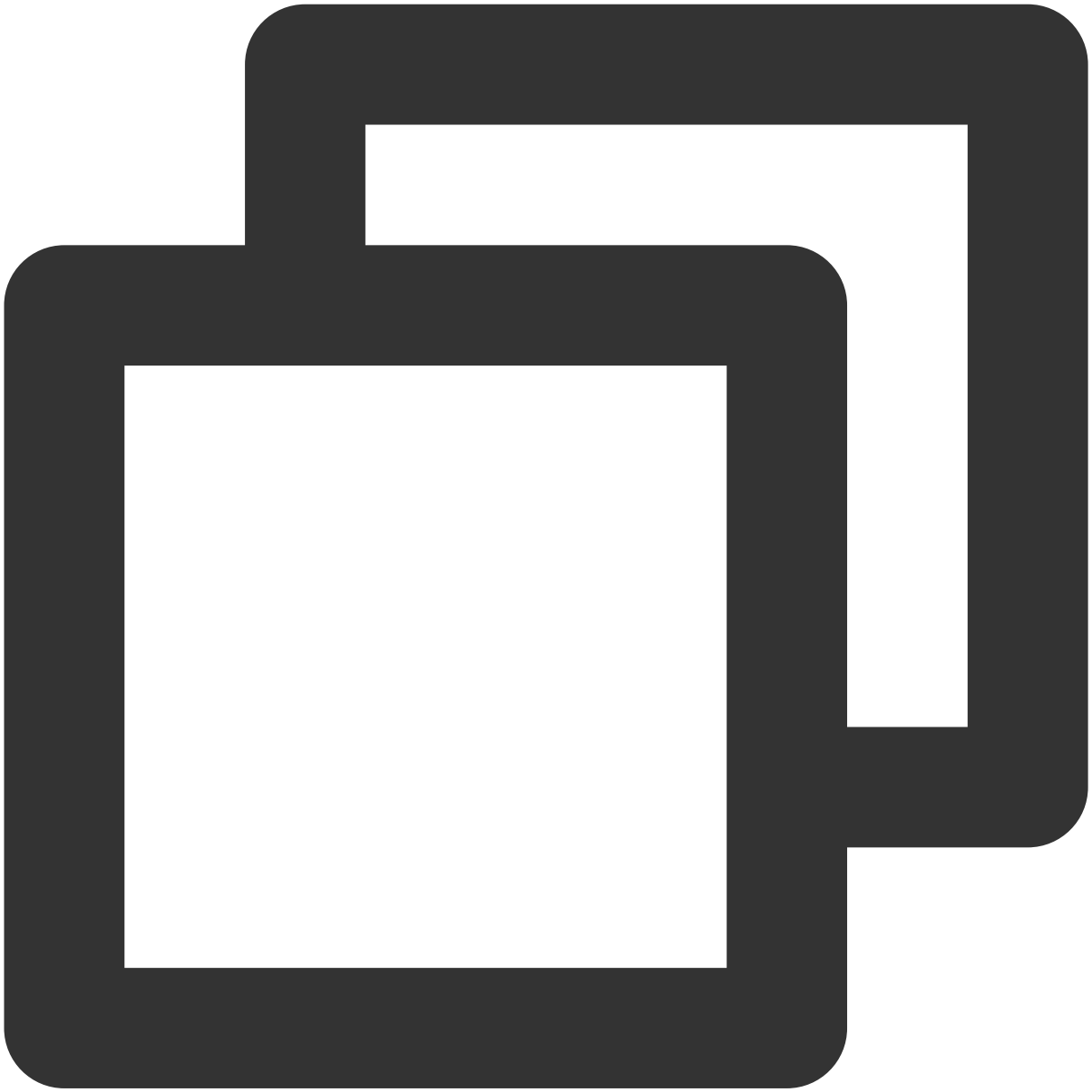
Return type: boolean

Example:

```
> SELECT rlike('%SystemDrive%\\Users\\John', '%SystemDrive%\\Users.*');
true
```

FROM_CSV

Function statement:



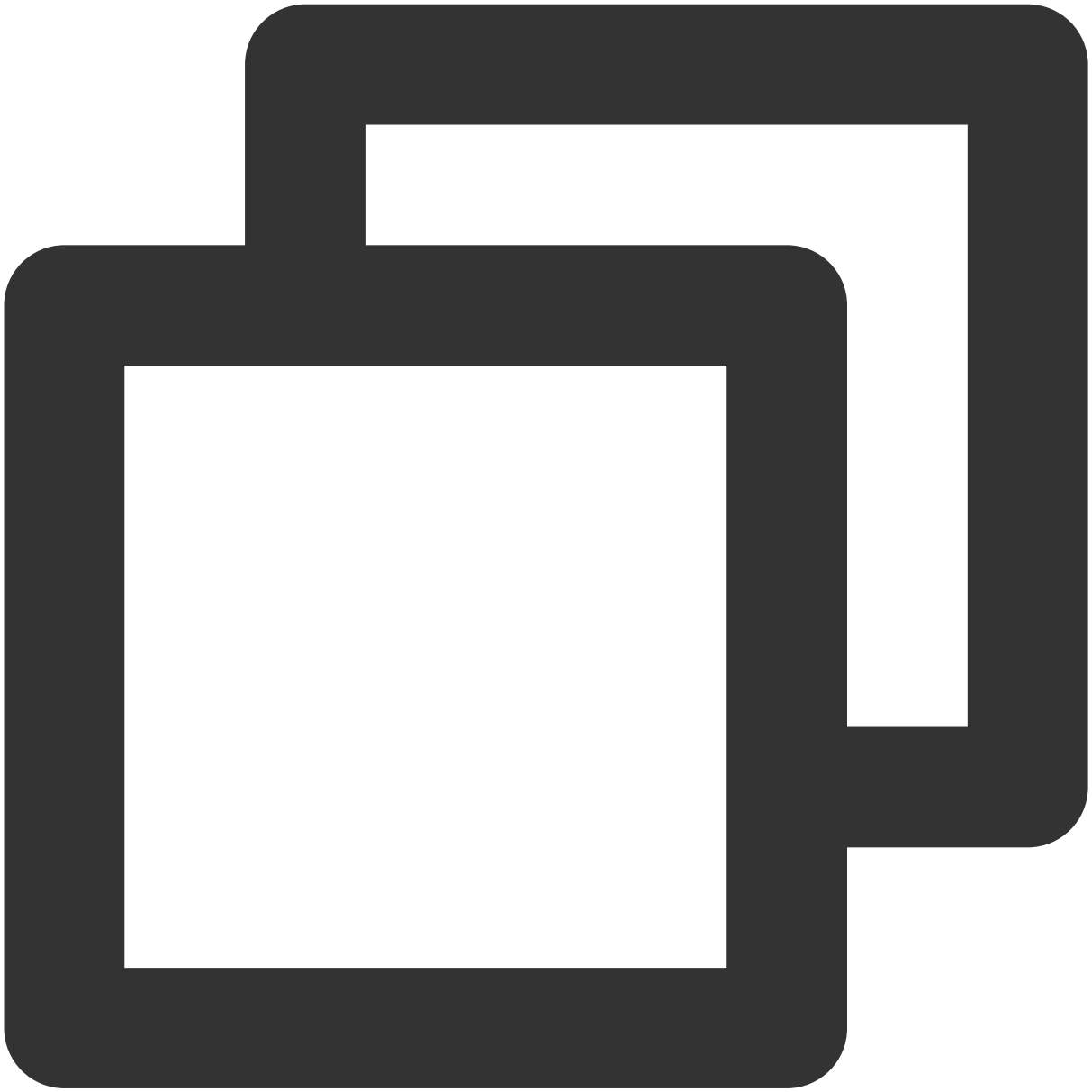
```
FROM_CSV(<cvsStr> string, <schema> string, <options> map<string, string>)
```

Supported engine: SparkSQL

Usage instructions: Return a structured value with the given csvStr and schema.

Return type: struct

Example:



```
> SELECT from_csv('1, 0.8', 'a INT, b DOUBLE');
 {"a":1,"b":0.8}
> SELECT from_csv('26/08/2015', 'time Timestamp', map('timestampFormat', 'dd/MM/yyy
 {"time":2015-08-26 00:00:00}
```

SCHEMA_OF_CSV

Function statement:

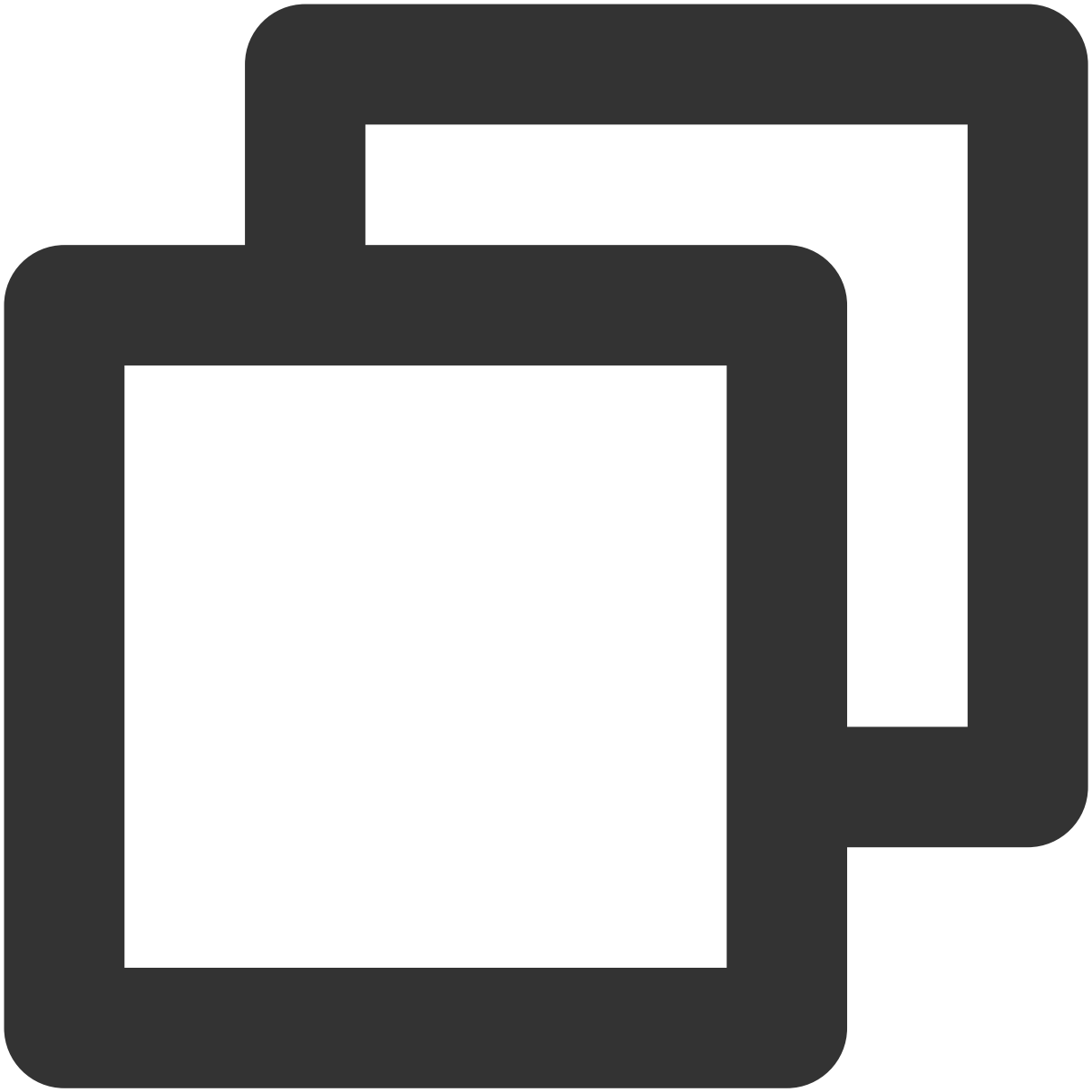
```
SCHEMA_OF_CSV(<csvStr> string[, options map<string, string>])
```

Supported engine: SparkSQL

Usage instructions: Return the schema of the csv string.

Return type: string

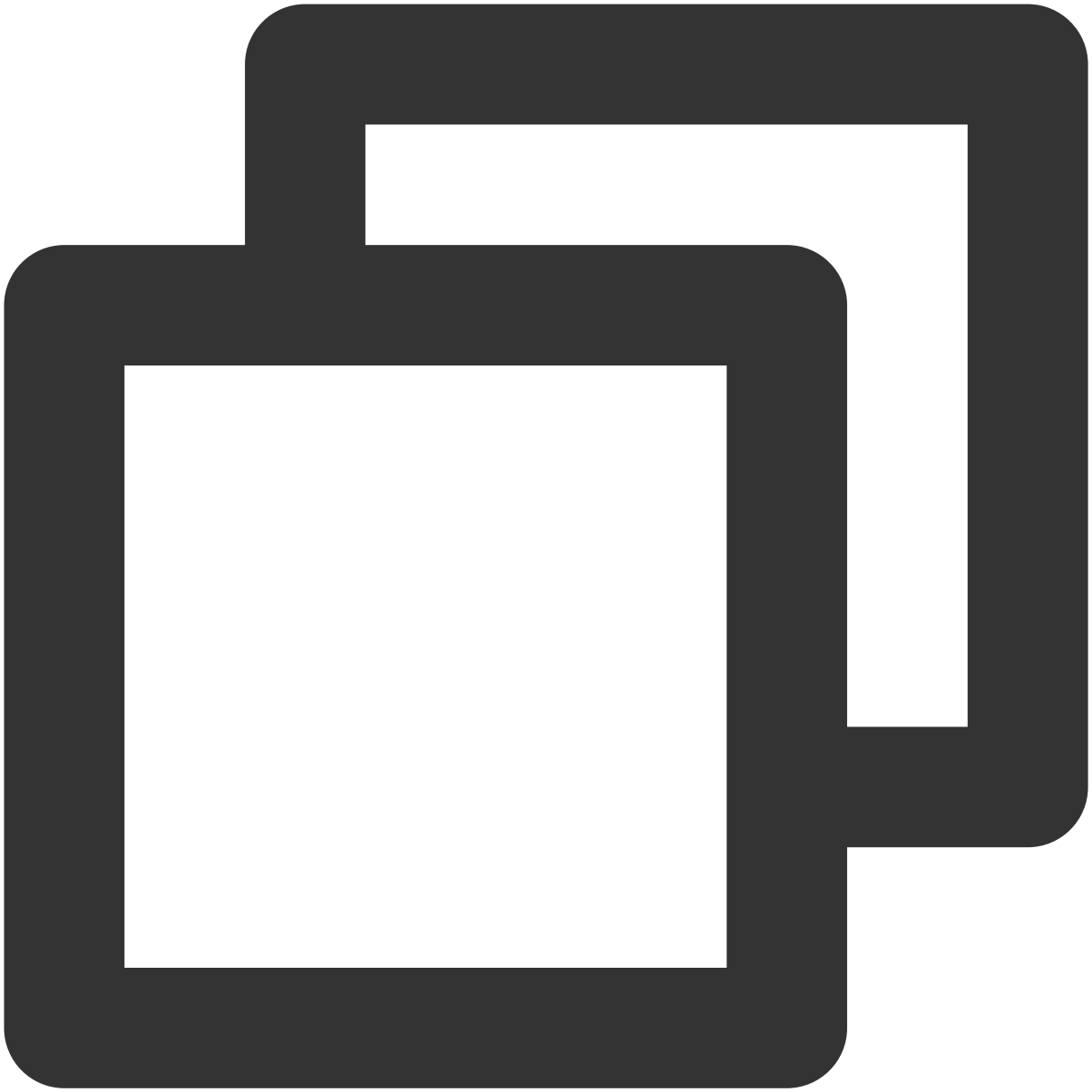
Example:



```
> SELECT schema_of_csv('1,abc');
 STRUCT<`_c0`: INT, `_c1`: STRING>
```

TO_CSV

Function statement:



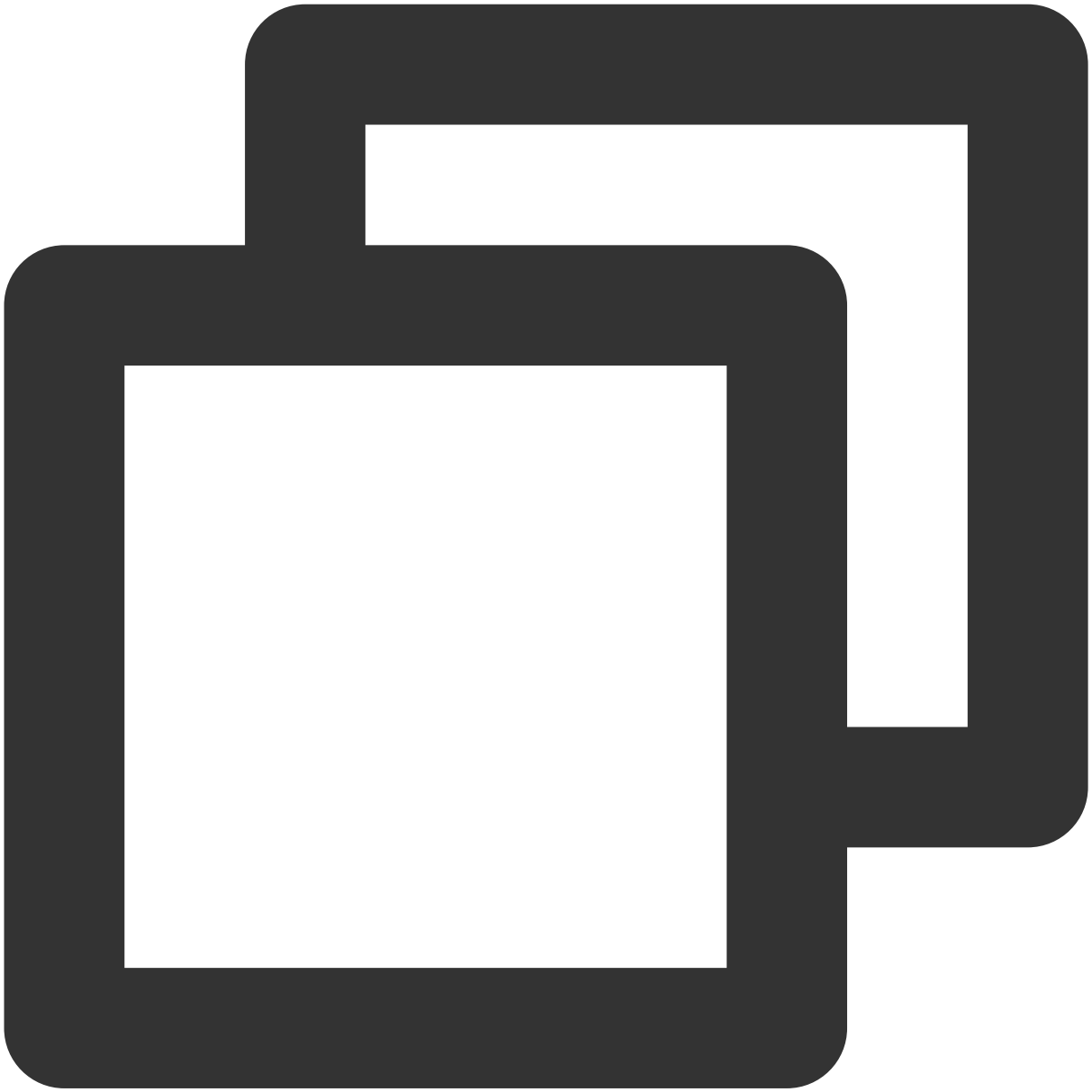
```
TO_CSV(<expr> struct[, <options> map<string, string>])
```

Supported engine: SparkSQL

Usage instructions: Return a CSV string with the given structured value.

Return type: string

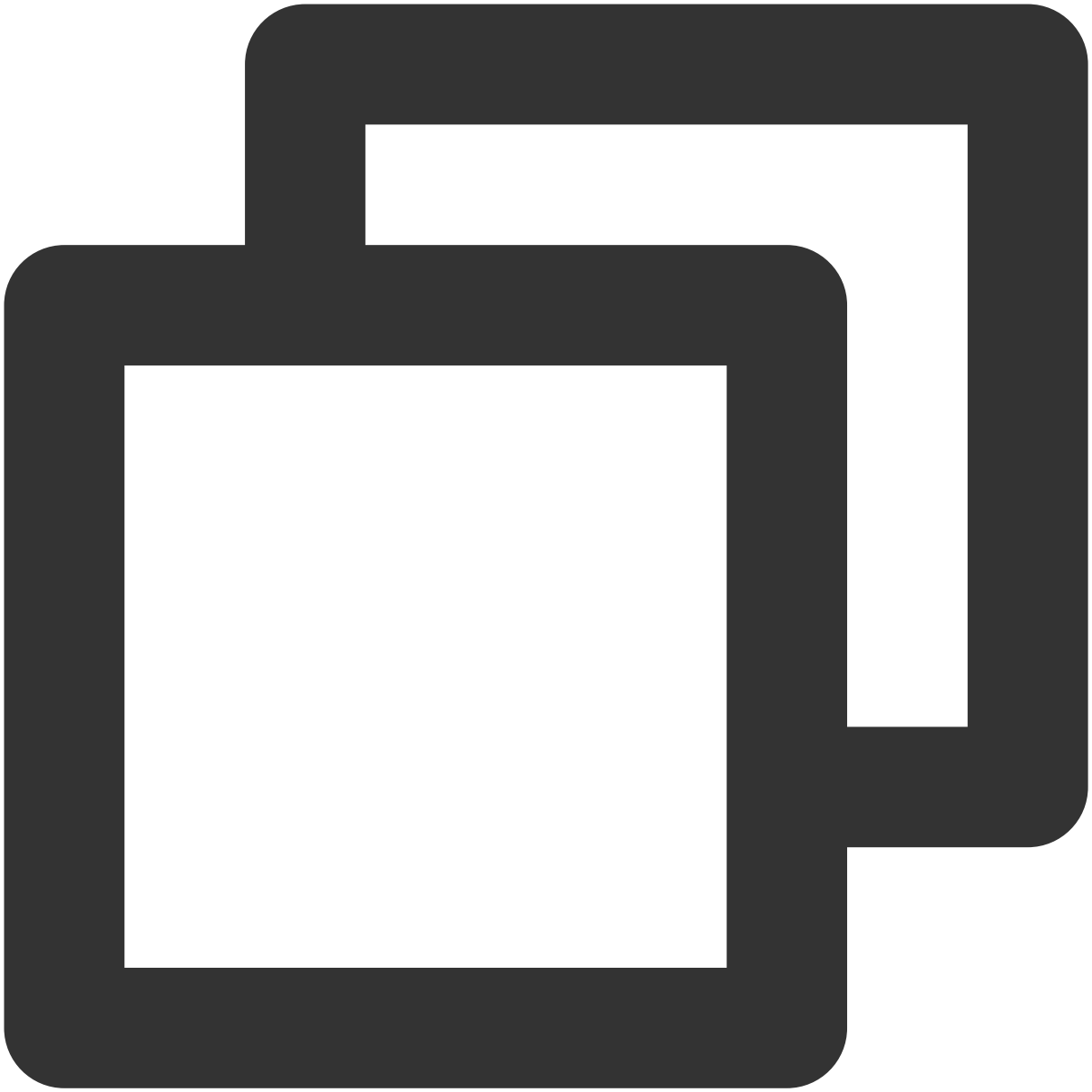
Example:



```
> SELECT to_csv(named_struct('a', 1, 'b', 2));
 1,2
> SELECT to_csv(named_struct('time', to_timestamp('2015-08-26', 'yyyy-MM-dd')), map
 26/08/2015
```

NGRAMS

Function statement:



```
NGRAMS(<a> array<array<string>>, <N> integer, <K> integer, <pf> integer)
```

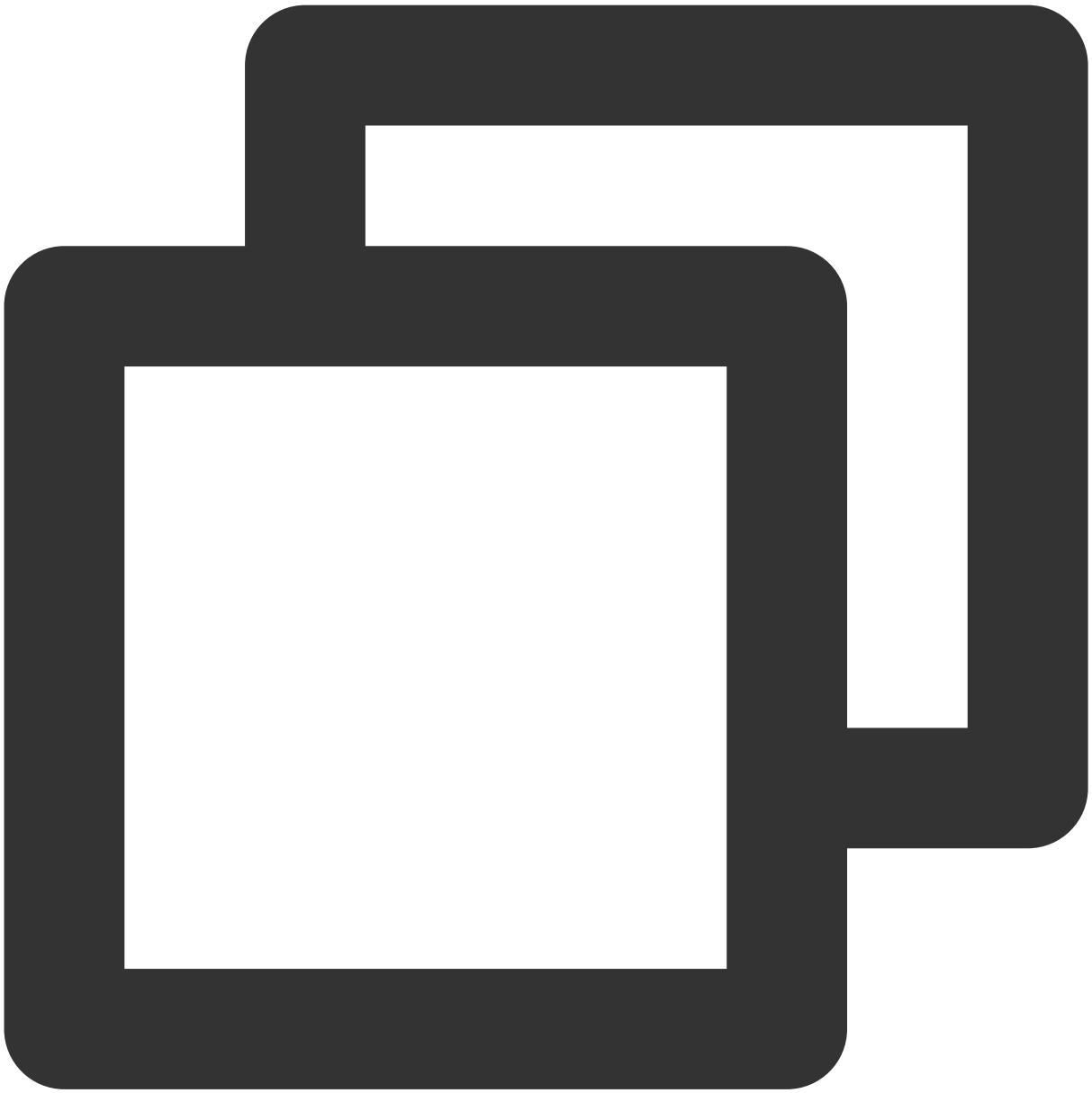
Supported engine: Presto

Usage instructions: Return the top k N-grams from a set of tokenized sentences.

Return type: array <struct <string,double>>

CONTEXT_NGRAMS

Function statement:



```
CONTEXT_NGRAMS((array &lt;array &lt;string>>, array &lt;string>, int, int))
```

Supported engine: Presto

Usage instructions: Given a contextual N-grams, return the top k contextual N-grams from a set of tokenized sentences.

Return type: array <struct <string, double>>

Collection Functions

Last updated：2024-08-07 17:33:09

APPROX_COUNT_DISTINCT

Function statement:

```
APPROX_COUNT_DISTINCT(<expr> any [, <relativeSD> integer|double|decimal])
```

Supported engine: SparkSQL

Usage instructions: Return the cardinality estimated by HyperLogLog++. relativeSD defines the allowable maximum relative standard deviation.

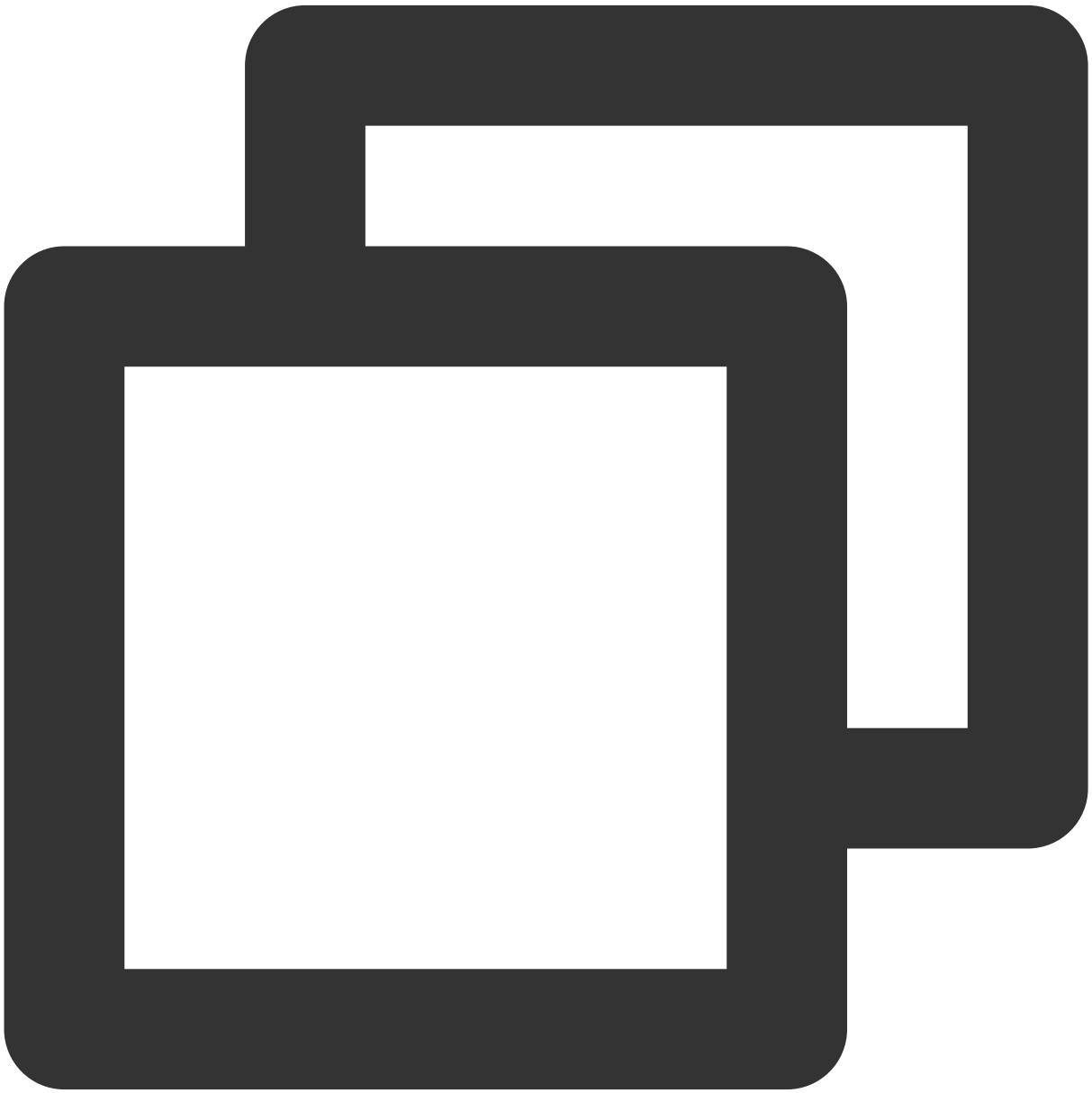
Return type: bigint

Example:

```
SELECT approx_count_distinct(col1) FROM (VALUES (1), (1), (2), (2), (3)) tab(col1);
 3
```

AVG

Function statement:



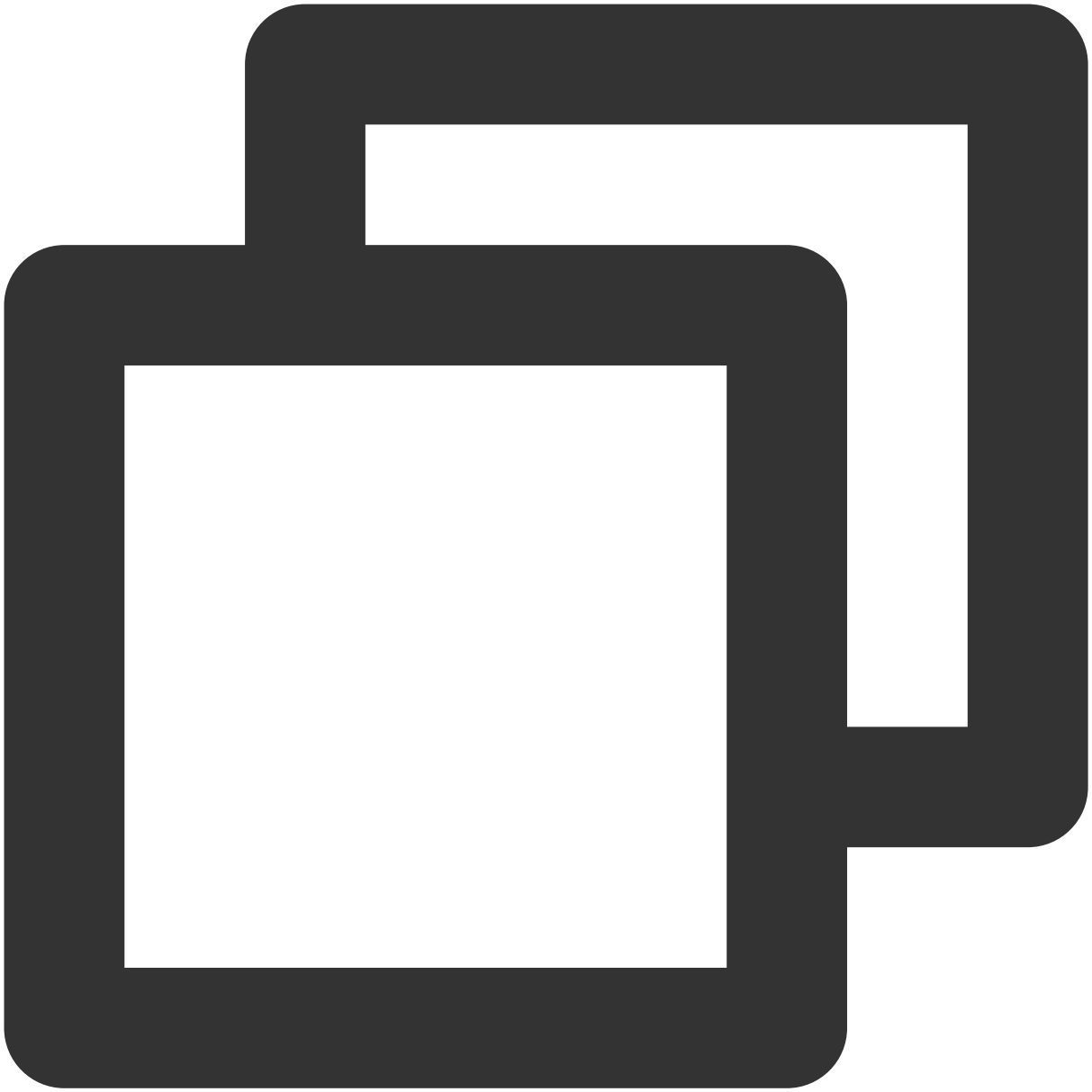
```
AVG(<expr> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the average value calculated based on the group's values.

Return type: double

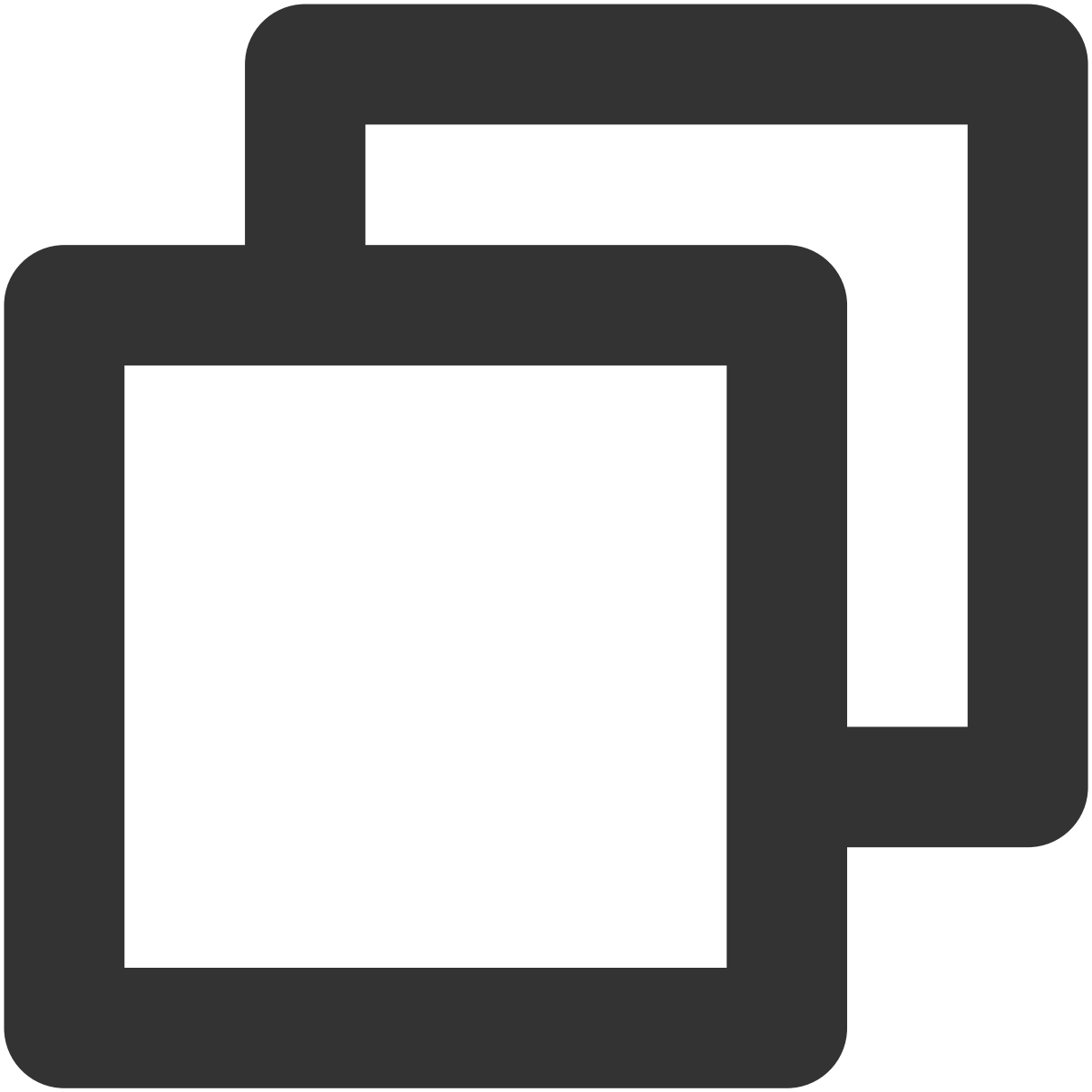
Example:



```
> SELECT avg(col) FROM (VALUES (1), (2), (3)) AS tab(col);
 2.0
> SELECT avg(col) FROM (VALUES (1), (2), (NULL)) AS tab(col);
 1.5
```

CORR

Function statement:



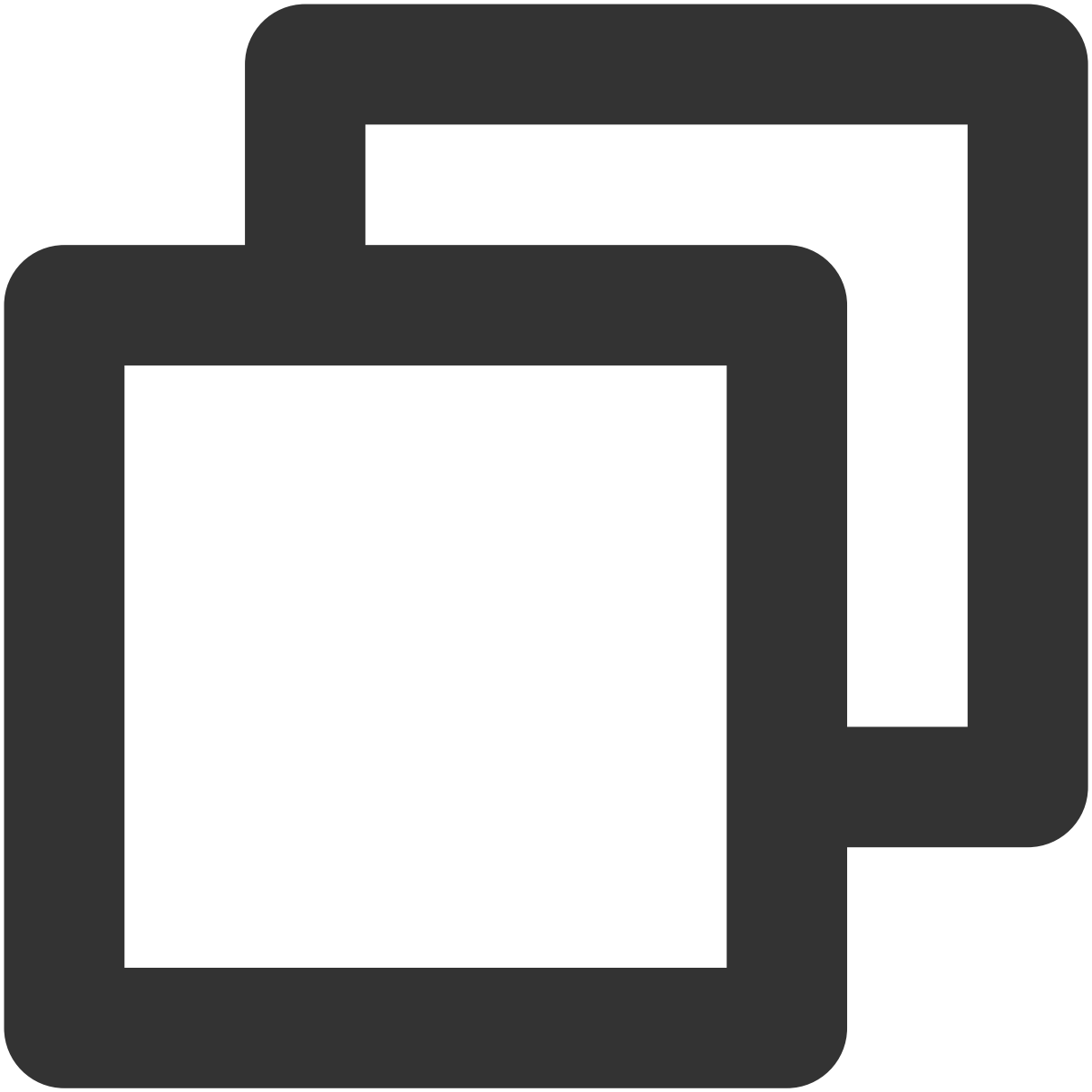
```
CORR(<expr> integer|double|decimal, <expr> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the Pearson correlation coefficient between a set of numeric pairs.

Return type: double

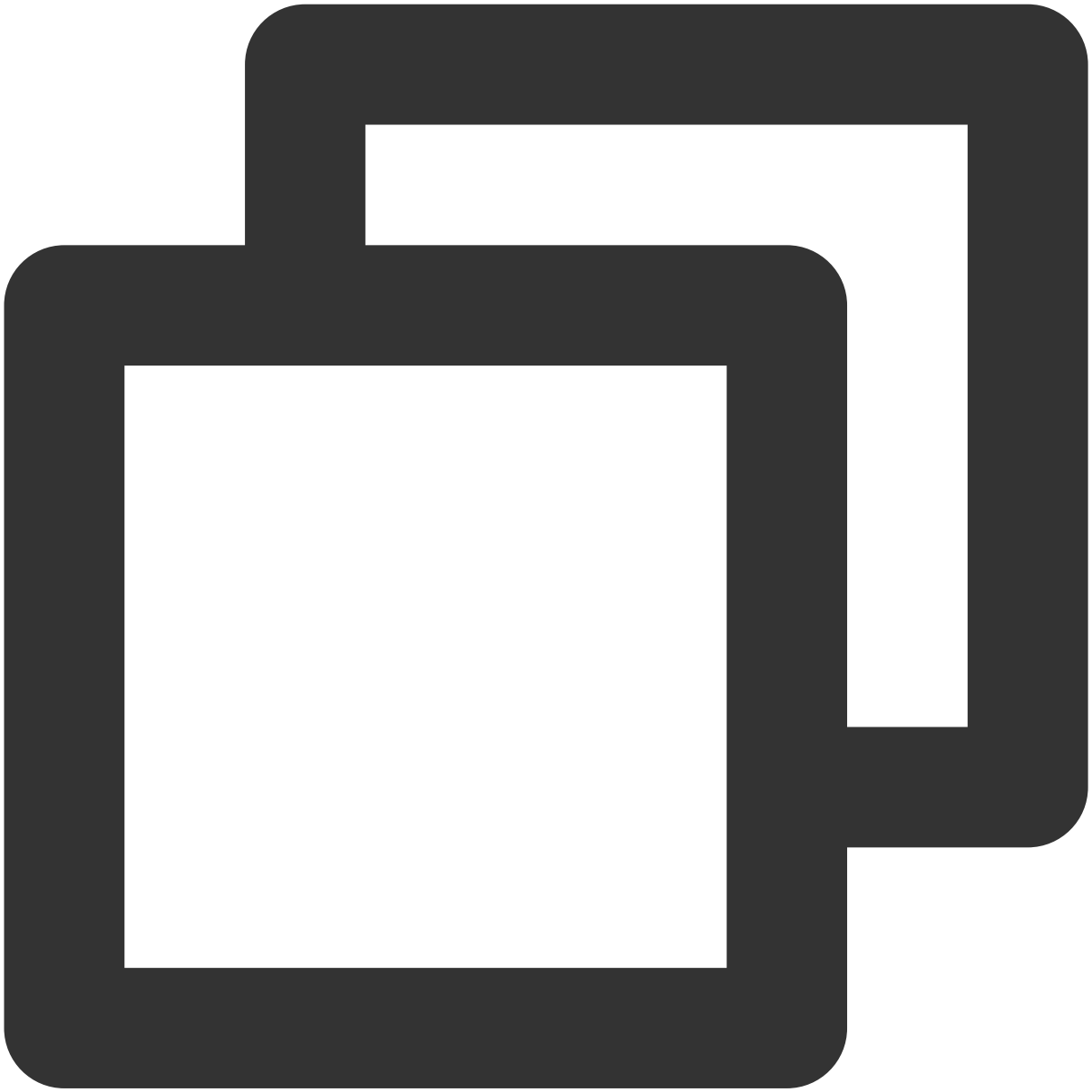
Example:



```
> SELECT corr(c1, c2) FROM (VALUES (3, 2), (3, 3), (6, 4)) as tab(c1, c2);
 0.8660254037844387
```

COUNT

Function statement:



```
COUNT(*)
COUNT([DISTINCT] <col1> ANY, <col2> ANY, ...)
```

Supported engines: SparkSQL and Presto

Usage instructions:

COUNT(*): Return the total number of retrieved rows, including rows with null values.

COUNT(<col1> ANY, <col2> ANY, ...): Return the number of rows where all provided expressions are non-null.

COUNT([DISTINCT] <col1> ANY, <col2> ANY, ...): Return the number of unique and non-null rows for the provided expressions.

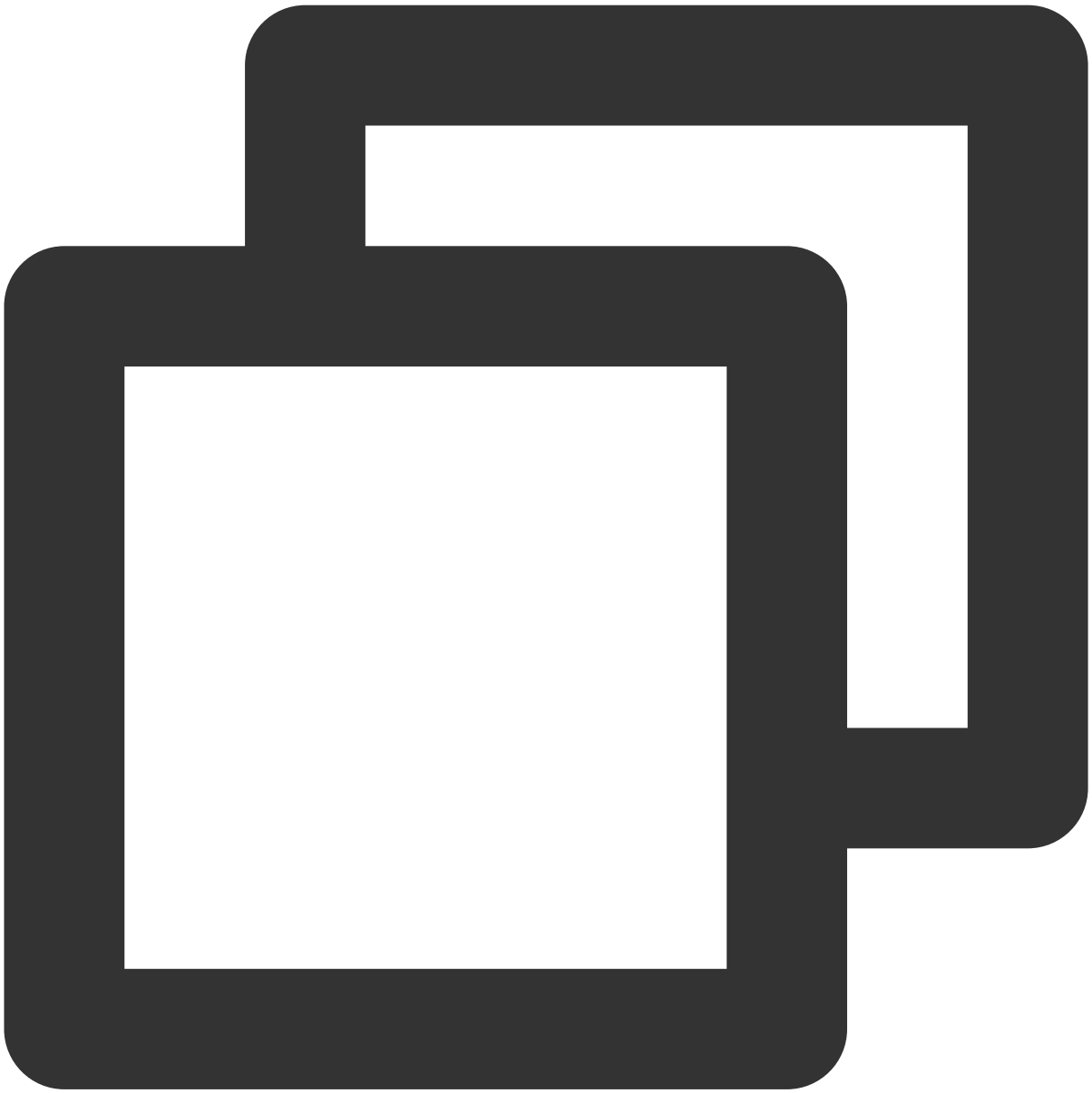
Return type: integer

Example:

```
> SELECT count(*) FROM (VALUES (NULL), (5), (5), (20)) AS tab(col);
 4
> SELECT count(col) FROM (VALUES (NULL), (5), (5), (20)) AS tab(col);
 3
> SELECT count(DISTINCT col) (FROM VALUES (NULL), (5), (5), (10)) AS tab(col);
 2
```

COUNT_IF

Function statement:



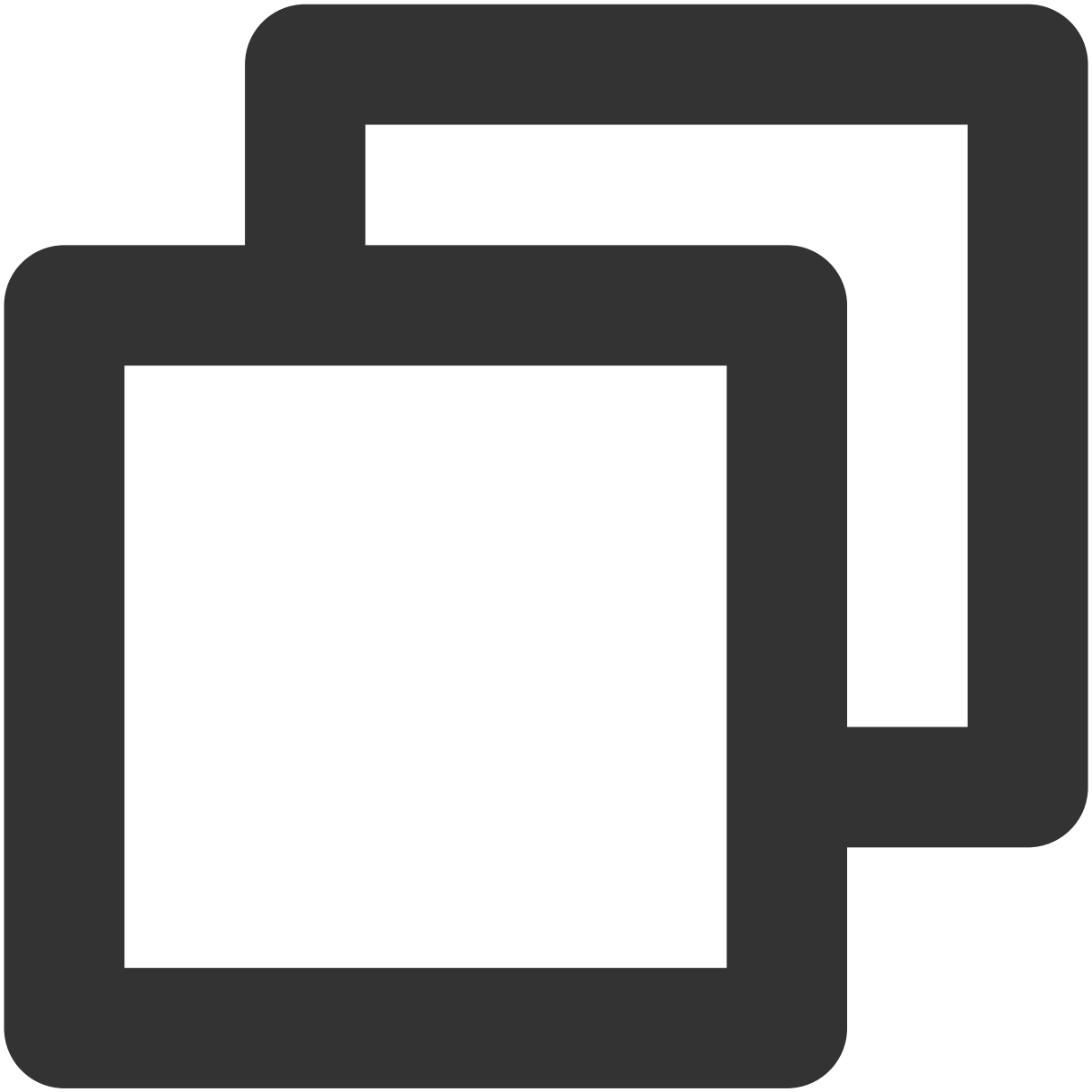
```
COUNT_IF(<expr> ANY)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the number of rows where the expression is TRUE.

Return type: int

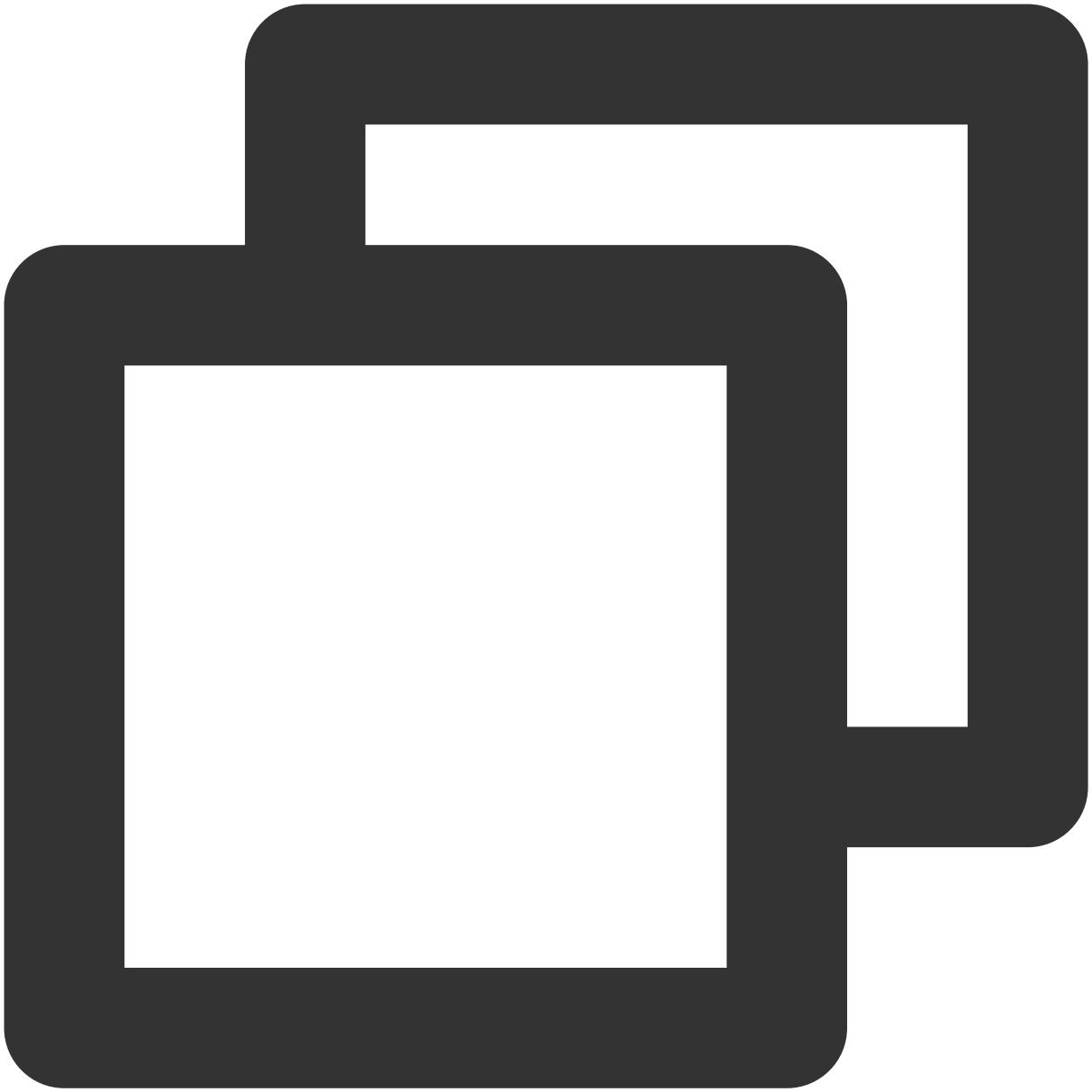
Example:



```
> SELECT count_if(col % 2 = 0) FROM (VALUES (NULL), (0), (1), (2), (3)) AS tab(col)
 2
> SELECT count_if(col IS NULL) FROM (VALUES (NULL), (0), (1), (2), (3)) AS tab(col)
 1
```

COVER_POP

Function statement:



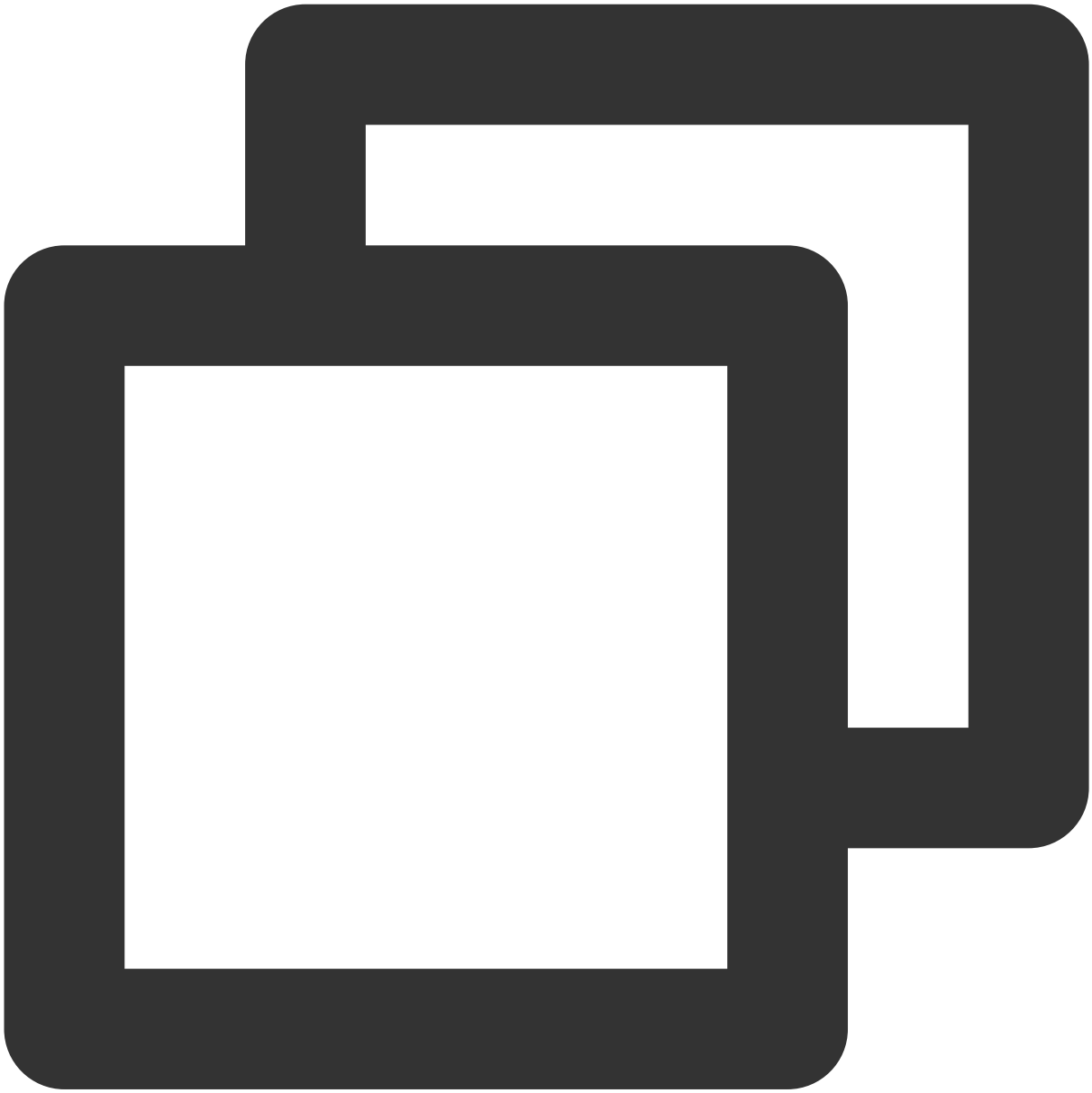
```
COVAR_POP(<expr1> integer|double|decimal, <expr2> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the population covariance of a set of numeric pairs.

Return type: double

Example:



```
> SELECT covar_pop(c1, c2) FROM (VALUES (1,1), (2,2), (3,3)) AS tab(c1, c2);
 0.6666666666666666
```

COVER_SAMP

Function statement:

```
COVAR_SAMP(<expr1> integer|double|decimal, <expr2> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the sample covariance of a set of numeric pairs.

Return type: double

Example:

```
> SELECT covar_samp(c1, c2) FROM (VALUES (1,1), (2,2), (3,3)) AS tab(c1, c2);
 1.0
```

FIRST_VALUE

Function statement:

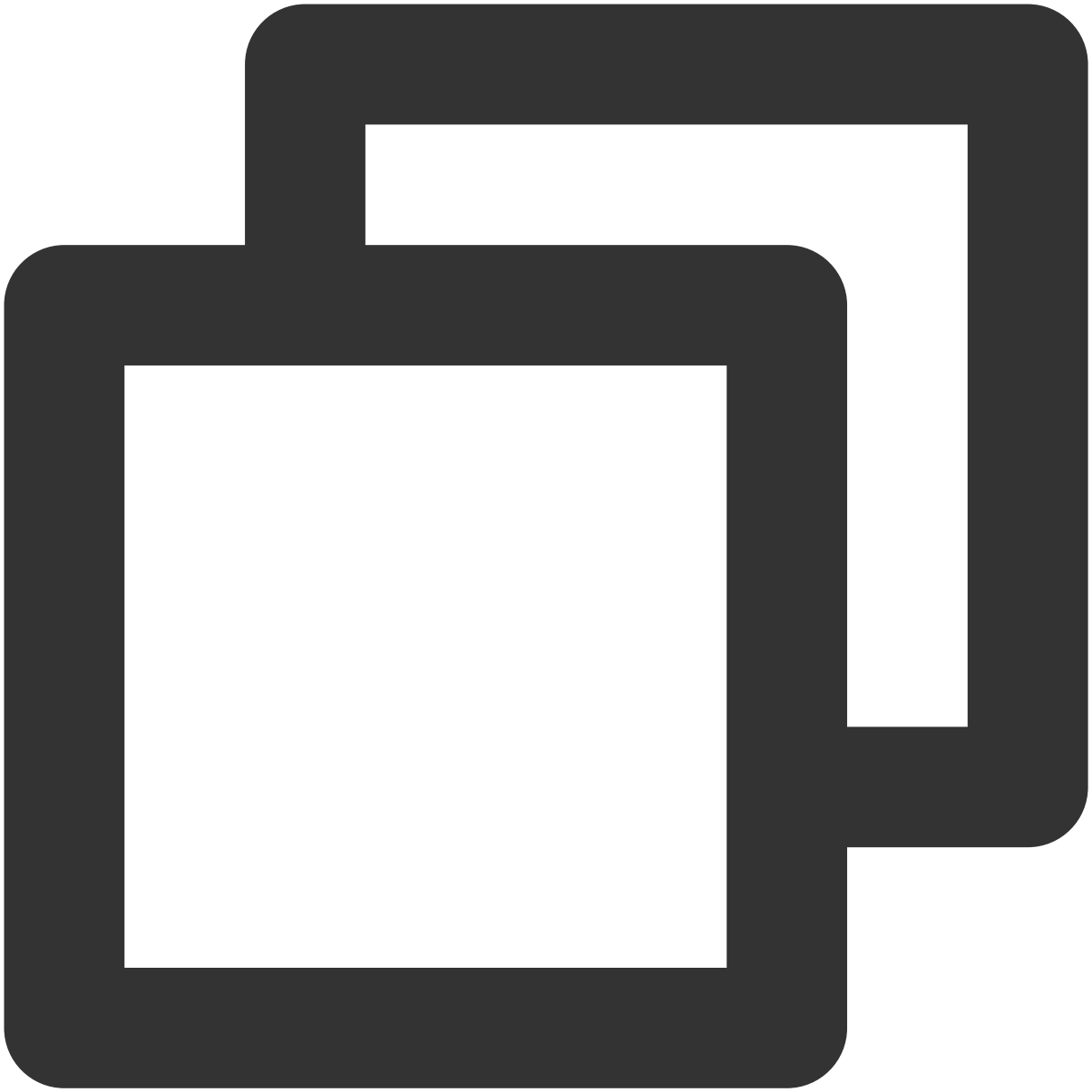
```
FIRST_VALUE(<expr> T[, <isIgnoreNull> boolean])
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the first value of expr in a set of rows. If isIgnoreNull is true, only non-null values are returned.

Return type: T

Example:



```
> SELECT first_value(col) FROM (VALUES (10), (5), (20)) AS tab(col);
 10
> SELECT first_value(col) FROM (VALUES (NULL), (5), (20)) AS tab(col);
 NULL
> SELECT first_value(col, true) FROM (VALUES (NULL), (5), (20)) AS tab(col);
 5
```

FIRST

Function statement:

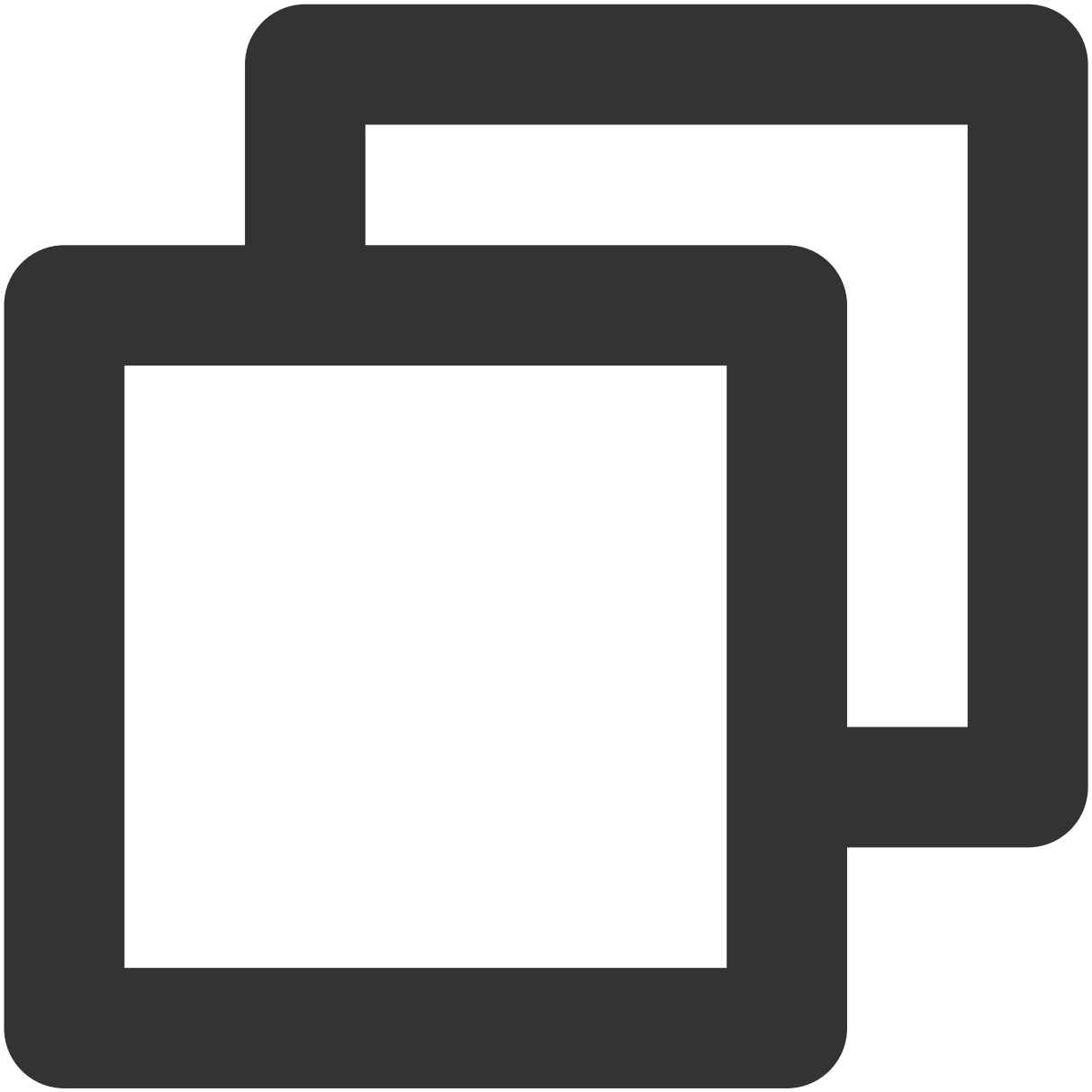
```
FIRST(<expr> T[, <isIgnoreNull> boolean])
```

Supported engine: SparkSQL

Usage instructions: Return the first value of expr in a set of rows. If isIgnoreNull is true, only non-null values are returned.

Return type: T

Example:



```
> SELECT first(col) FROM (VALUES (10), (5), (20)) AS tab(col);
 10
> SELECT first(col) FROM (VALUES (NULL), (5), (20)) AS tab(col);
 NULL
> SELECT first(col, true) FROM (VALUES (NULL), (5), (20)) AS tab(col);
 5
```

KURTOSIS

Function statement:

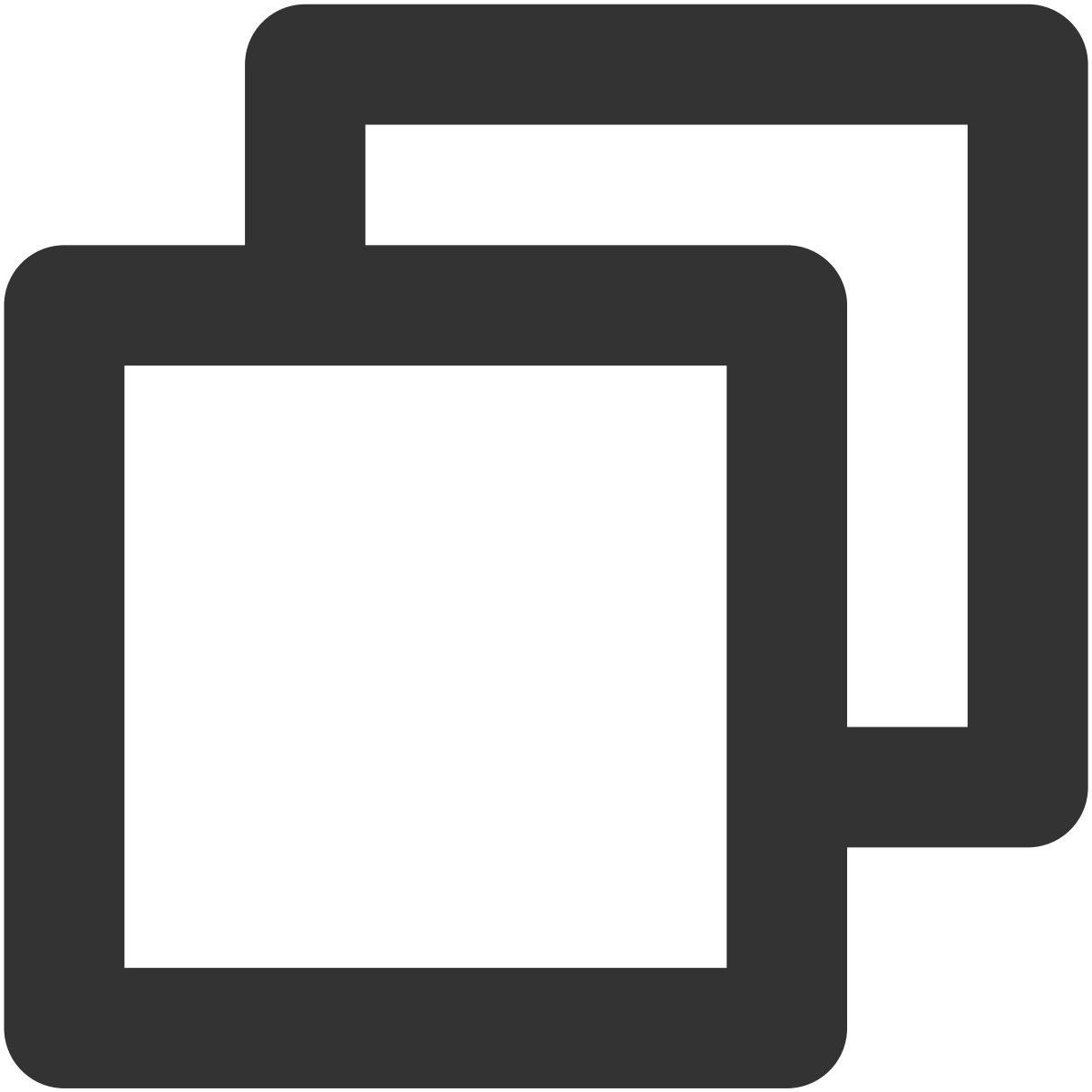
```
KURTOSIS(<expr> integer|double|decimal)
```

Supported engine: SparkSQL

Usage instructions: Return the kurtosis value calculated based on the group's values.

Return type: double

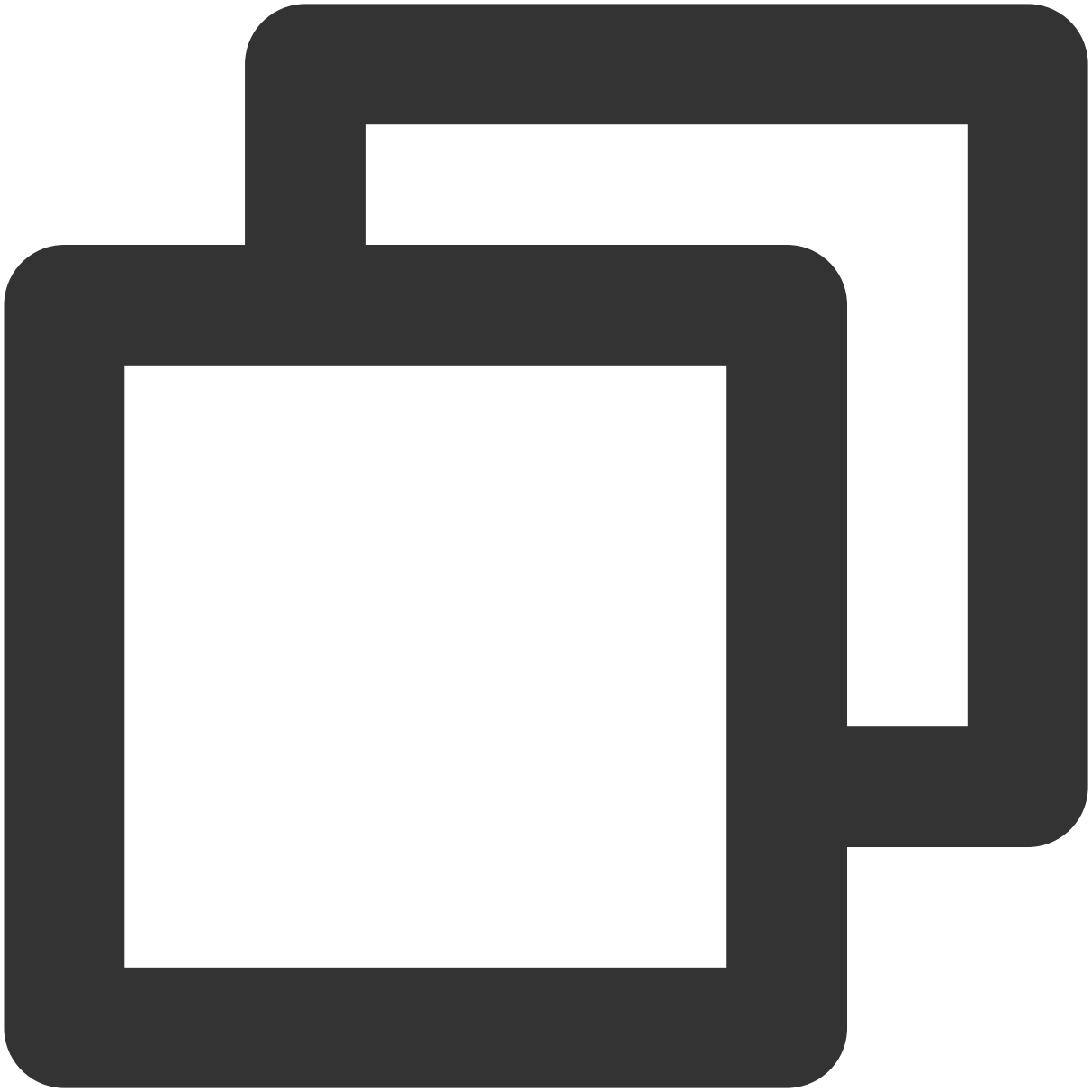
Example:



```
> SELECT kurtosis(col) FROM (VALUES (-10), (-20), (100), (1000)) AS tab(col);
 -0.7014368047529627
> SELECT kurtosis(col) FROM (VALUES (1), (10), (100), (10), (1)) as tab(col);
 0.19432323191699075s
```

LAST_VALUE

Function statement:



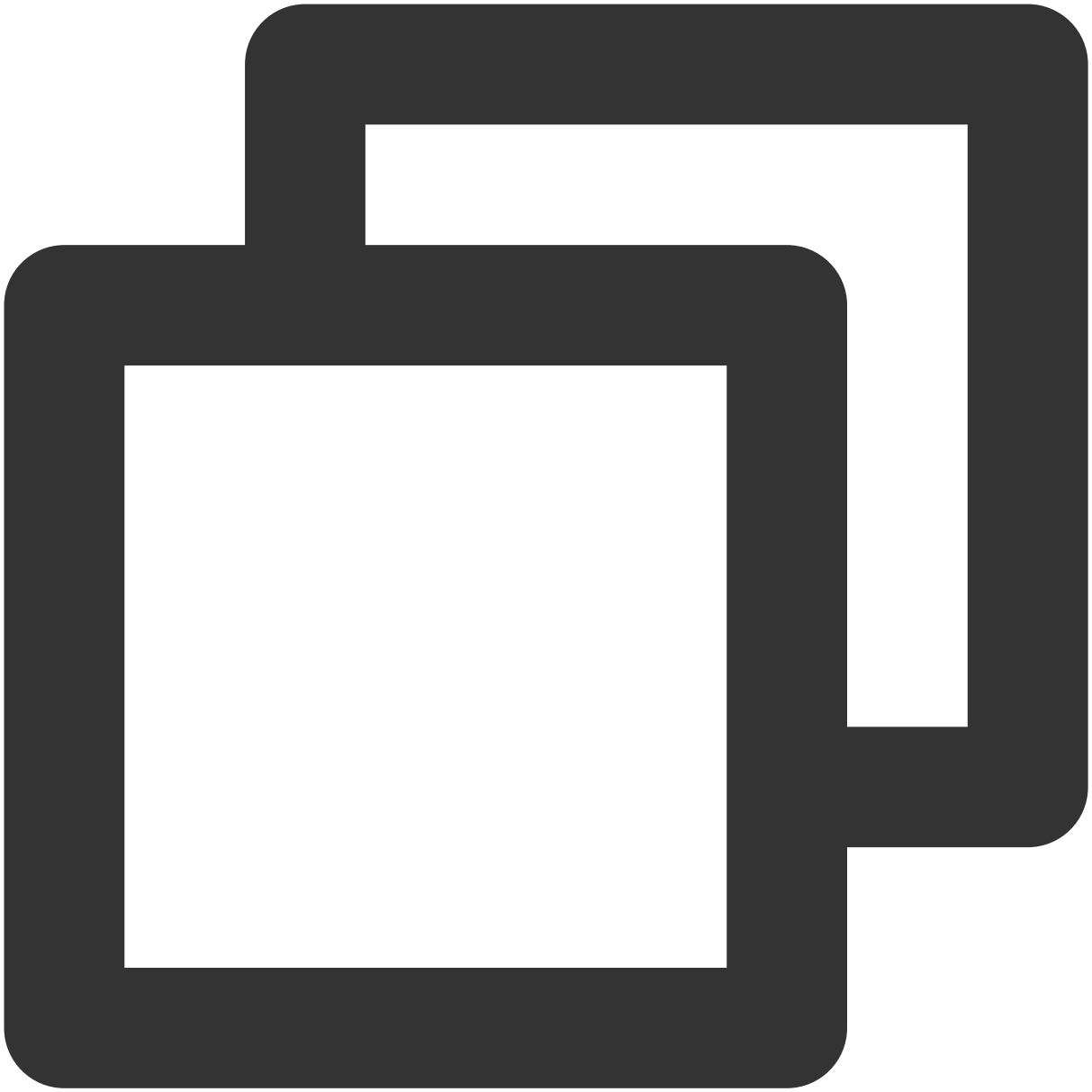
```
LAST_VALUE(<expr> T[, <isIgnoreNull> boolean])
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the last value of expr in a set of rows. If isIgnoreNull is true, only non-null values are returned.

Return type: T

Example:



```
> SELECT last_value(col) FROM (VALUES (10), (5), (20)) AS tab(col);
 20
> SELECT last_value(col) FROM (VALUES (10), (5), (NULL)) AS tab(col);
 NULL
> SELECT last_value(col, true) FROM (VALUES (10), (5), (NULL)) AS tab(col);
 5
```

LAST

Function statement:

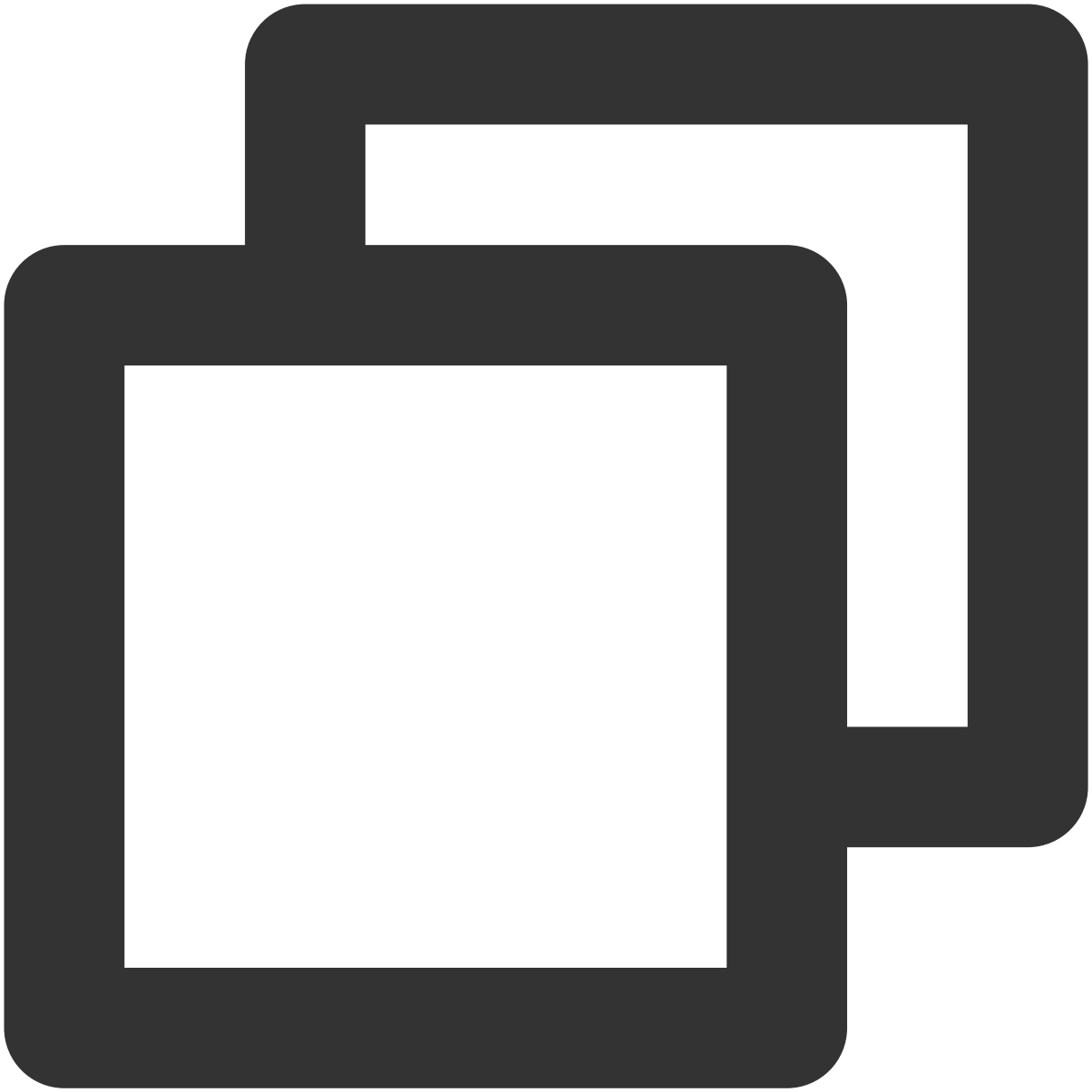
```
LAST(<expr> T[, <isIgnoreNull> boolean])
```

Supported engine: SparkSQL

Usage instructions: Return the last value of expr in a set of rows. If isIgnoreNull is true, only non-null values are returned.

Return type: T

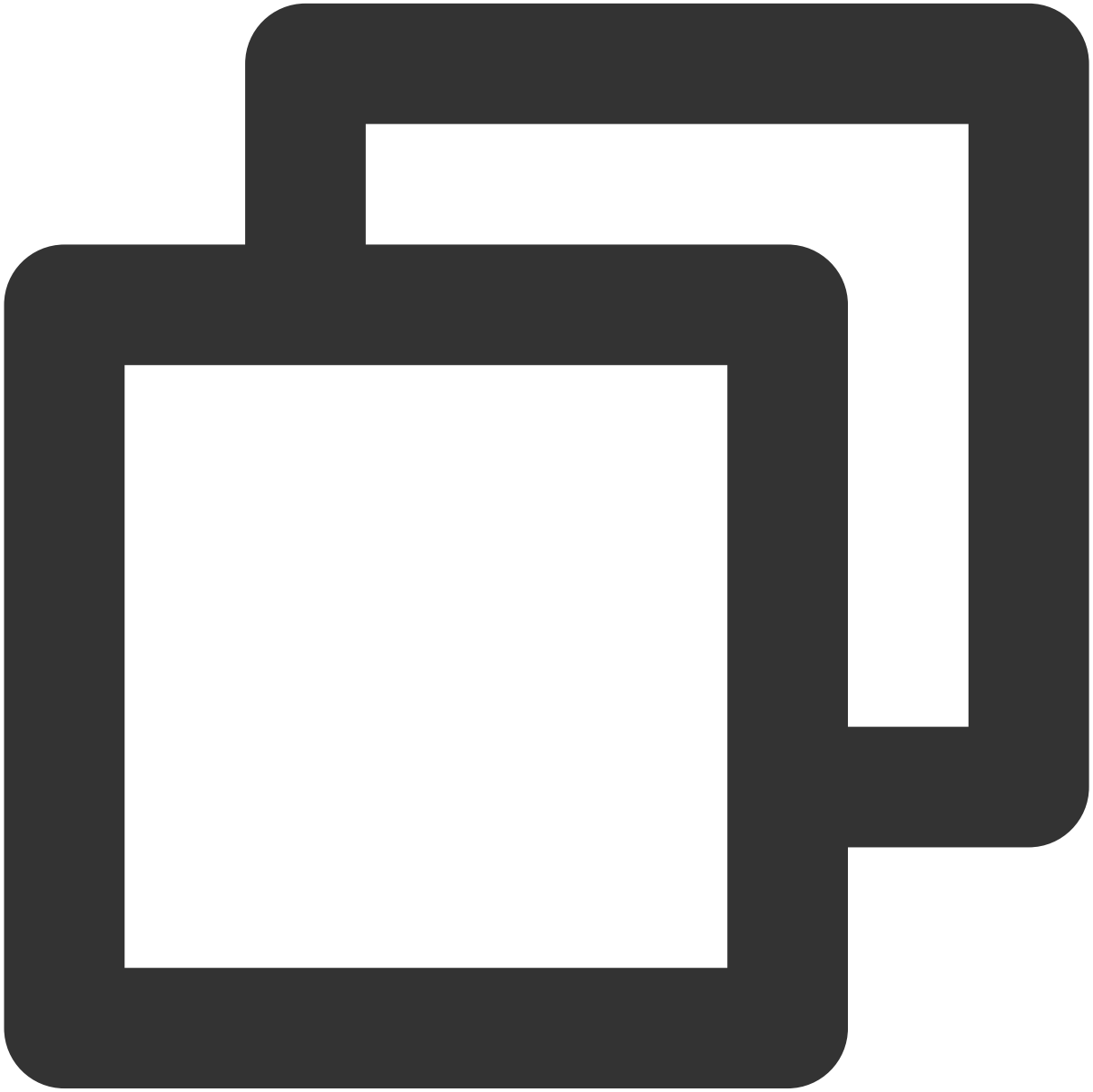
Example:



```
> SELECT last(col) FROM (VALUES (10), (5), (20)) AS tab(col);
 20
> SELECT last(col) FROM (VALUES (10), (5), (NULL)) AS tab(col);
 NULL
> SELECT last(col, true) FROM (VALUES (10), (5), (NULL)) AS tab(col);
 5
```

MEAN

Function statement:



```
MEAN(<expr> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the average value calculated based on the group's values.

Return type: double

Example:

```
> SELECT mean(col) FROM (VALUES (1), (2), (3)) AS tab(col);
 2.0
> SELECT mean(col) FROM (VALUES (1), (2), (NULL)) AS tab(col);
 1.5
```

PERCENTILE

Function statement:

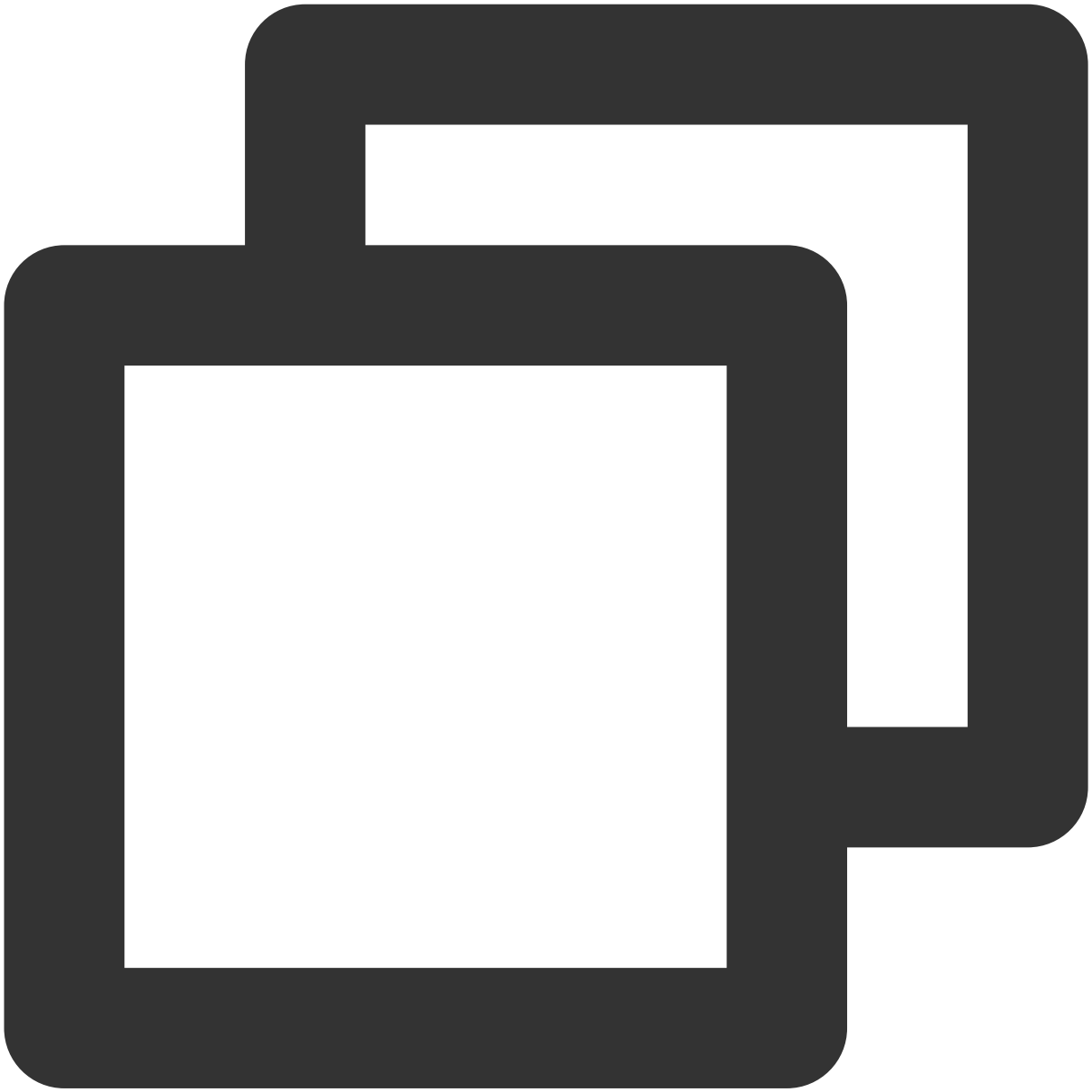
```
PERCENTILE(<col> ANY, <percentage> integer|double|decimal|array<double> [, <frequen
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the exact percentile value of the numeric column col at the given percentage. The percentage value must be between 0.0 and 1.0. frequency should be a positive integer.

Return type: double

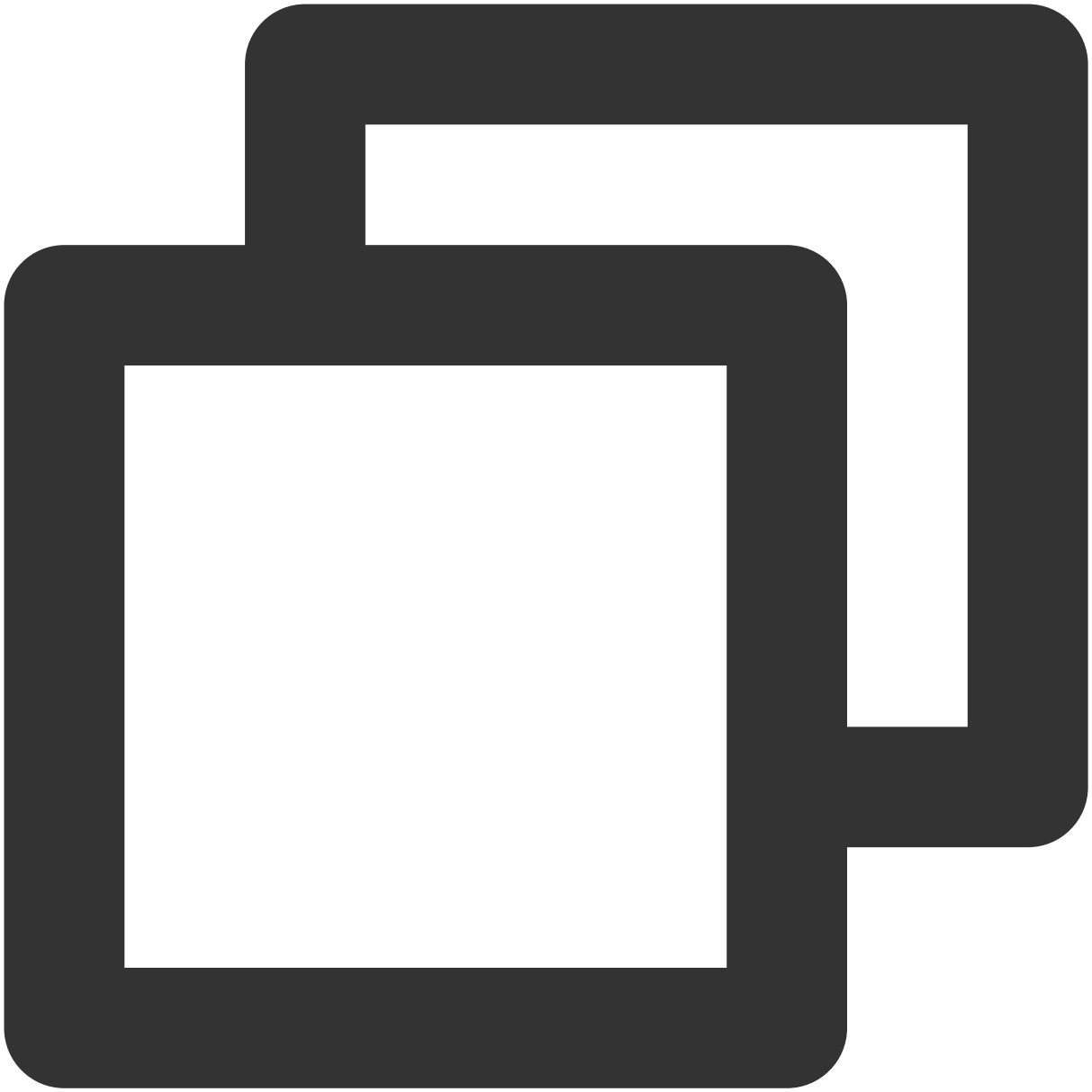
Example:



```
> SELECT percentile(col, 0.3) FROM (VALUES (0), (10)) AS tab(col);
 3.0
> SELECT percentile(col, array(0.25, 0.75)) FROM (VALUES (0), (10)) AS tab(col);
 [2.5,7.5]
```

SKEWNESS

Function statement:



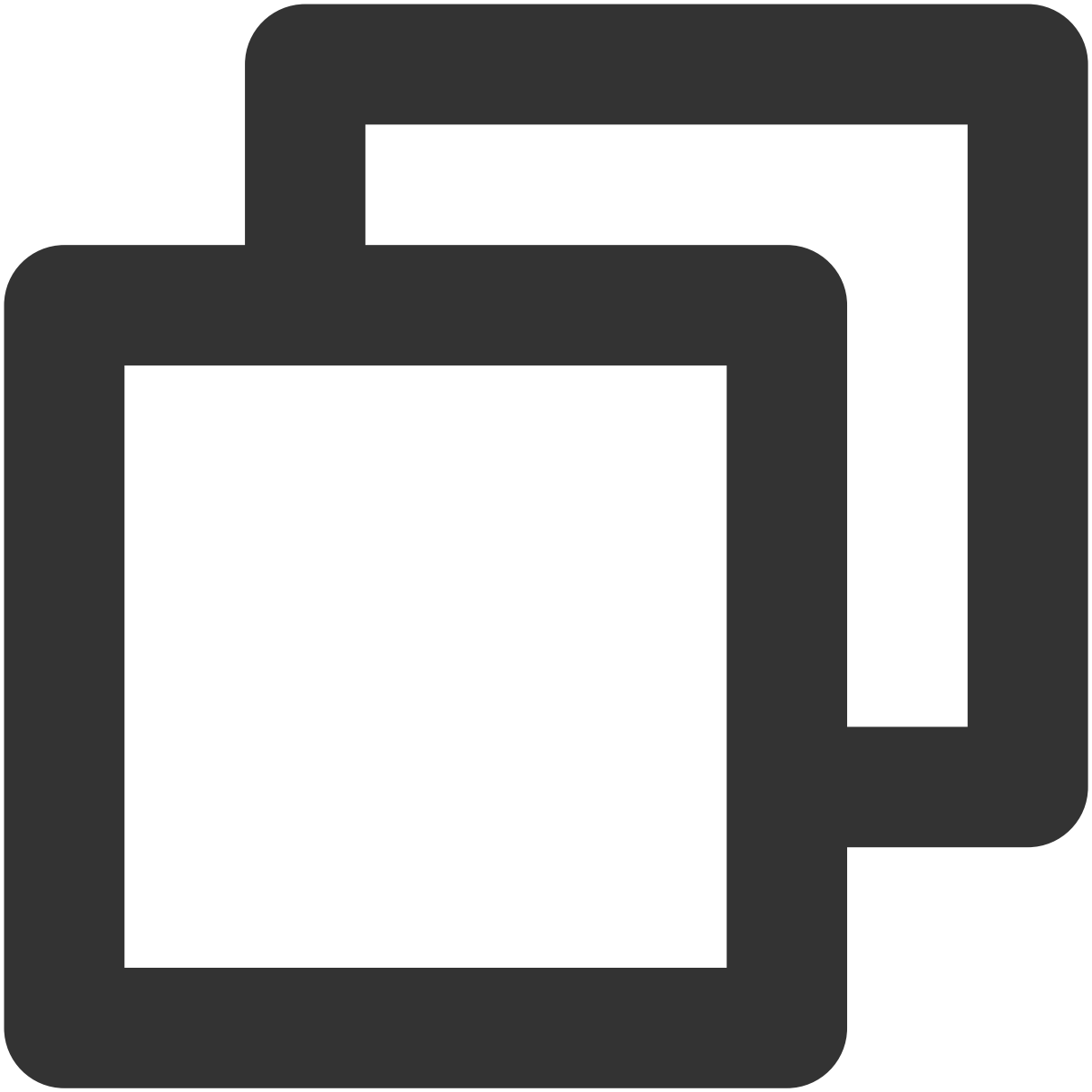
```
SKEWNESS(<col> integer|double|decimal)
```

Supported engine: SparkSQL

Usage instructions: Return the skewness value calculated based on the group's values.

Return type: double

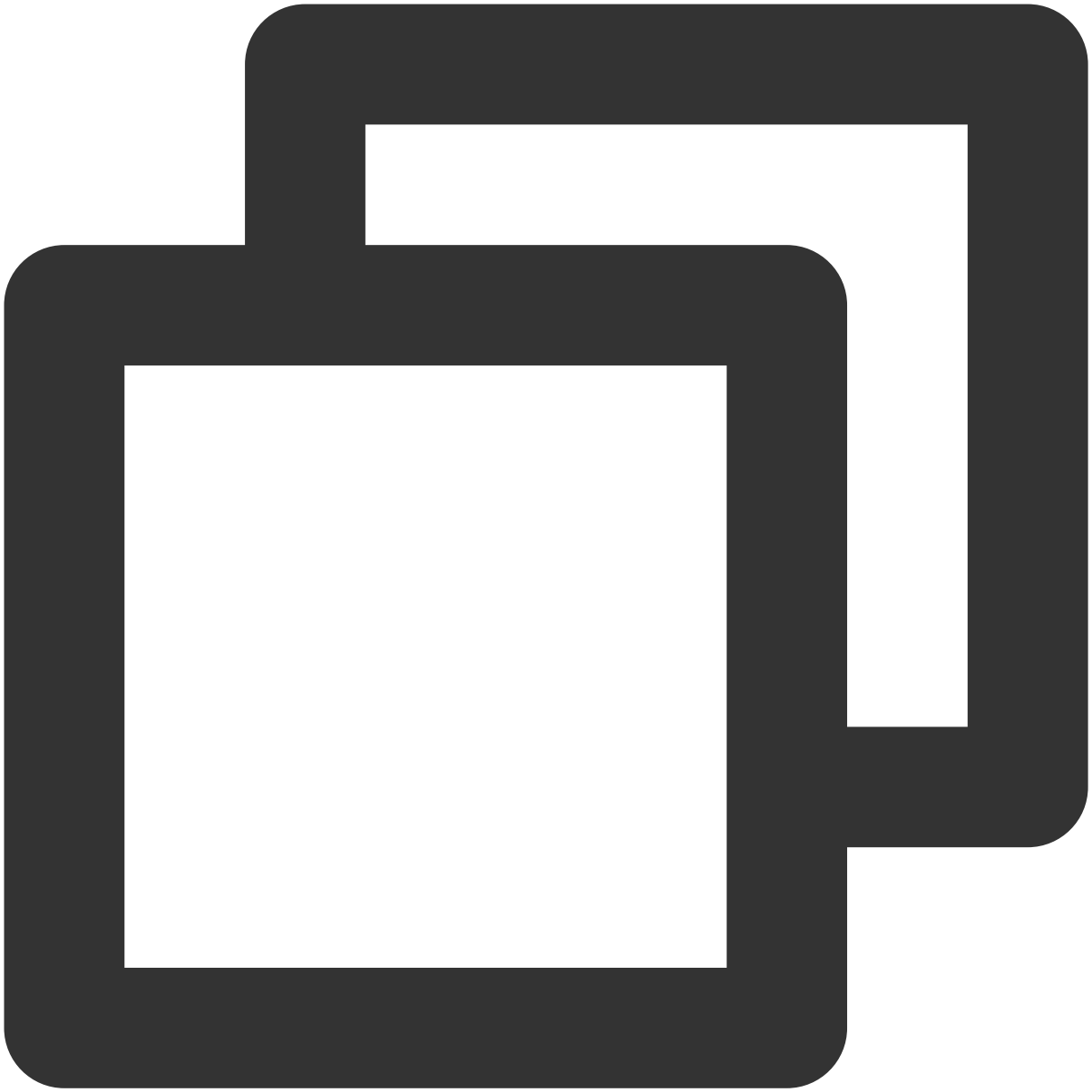
Example:



```
> SELECT skewness(col) FROM (VALUES (-10), (-20), (100), (1000)) AS tab(col);
 1.1135657469022011
> SELECT skewness(col) FROM (VALUES (-1000), (-100), (10), (20)) AS tab(col);
 -1.1135657469022011
```

PERCENTILE_APPROX

Function statement:



```
PERCENTILE_APPROX(<col> integer|double|decimal, <percentage> double|array<double>[,
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the approximate percentile of the numeric column col, which is the minimum value in the ordered col values (sorted from minimum to maximum) such that no more than the given percentage of col values are less than or equal to this value. The percentage value must be between 0.0 and 1.0. The accuracy parameter (default value: 10,000) is a positive numeric literal that controls the approximation accuracy at the cost of memory. The higher the accuracy value, the better the precision. 1.0/accuracy is the approximate relative error. When percentage is an

array, each value in the percentage array must be between 0.0 and 1.0. In this case, an array of approximate percentiles for the column col corresponding to the given percentage array is returned.

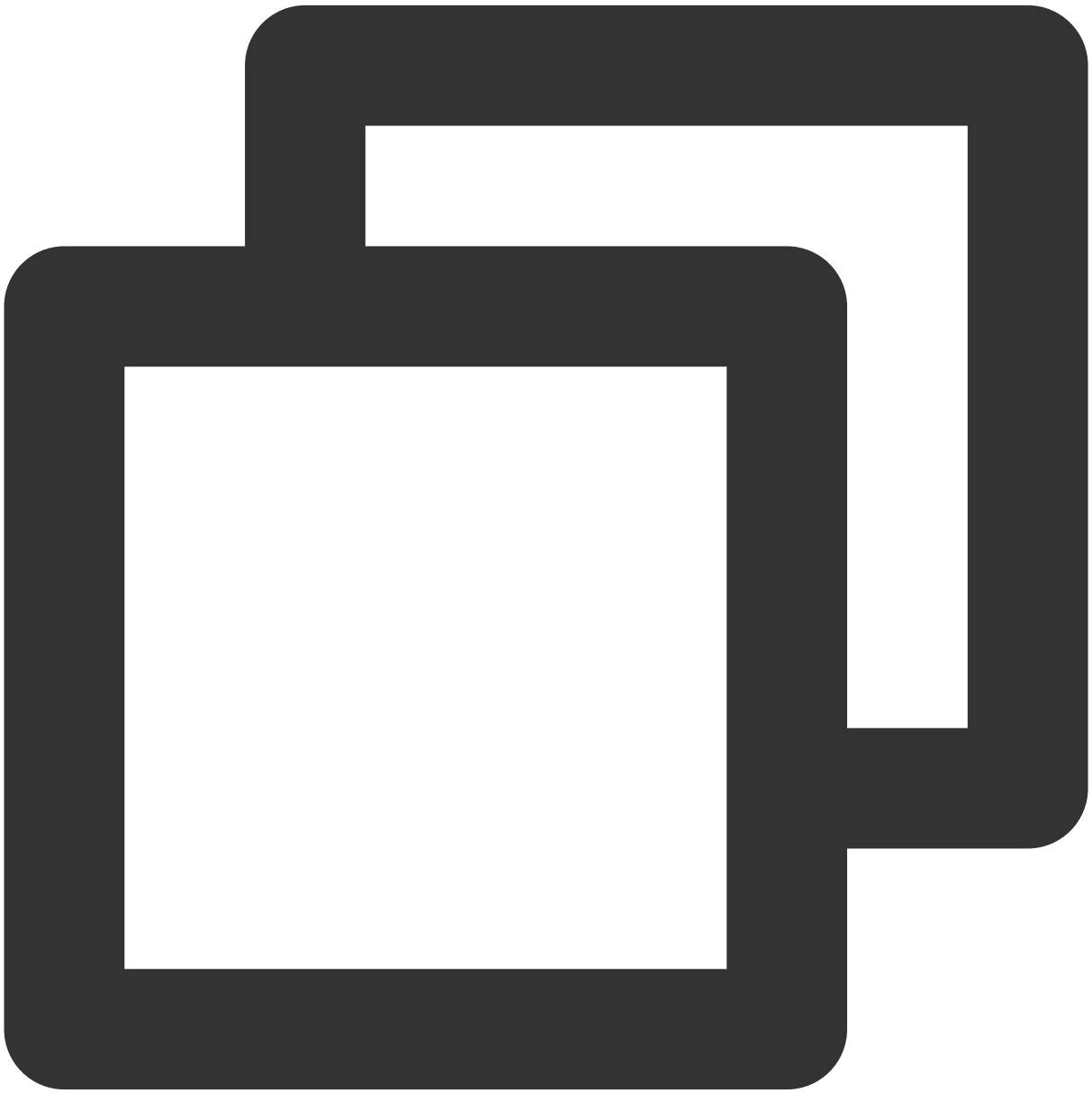
Return type: integer |array<integer>

Example:

```
> SELECT percentile_approx(col, array(0.5, 0.4, 0.1), 100) FROM (VALUES (0), (1), (
 [1,1,0]
> SELECT percentile_approx(col, 0.5, 100) FROM (VALUES (0), (6), (7), (9), (10)) AS
 7
```

APPROX_PERCENTILE

Function statement:



```
APPROX_PERCENTILE(<col> integer|double|decimal, <percentage> double|array<double>[,
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the approximate percentile of the numeric column col, which is the minimum value in the ordered col values (sorted from minimum to maximum) such that no more than the given percentage of col values are less than or equal to this value. The percentage value must be between 0.0 and 1.0. The accuracy parameter (default

value: 10,000) is a positive numeric literal that controls the approximation accuracy at the cost of memory. The higher the accuracy value, the better the precision. 1.0/accuracy is the approximate relative error. When percentage is an array, each value in the percentage array must be between 0.0 and 1.0. In this case, an array of approximate percentiles for the column col corresponding to the given percentage array is returned.

Return type: integer |array<integer>

Example:

```
> SELECT APPROX_PERCENTILE(col, array(0.5, 0.4, 0.1), 100) FROM (VALUES (0), (1), (
 [1,1,0]
> SELECT APPROX_PERCENTILE(col, 0.5, 100) FROM (VALUES (0), (6), (7), (9), (10)) AS
```

```
7
```

MAX

Function statement:

```
MAX(<col> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the maximum value of col.

Return type: same as col

Example:

```
> SELECT max(col) FROM (VALUES (10), (50), (20)) AS tab(col);
 50
```

MAX_BY

Function statement:

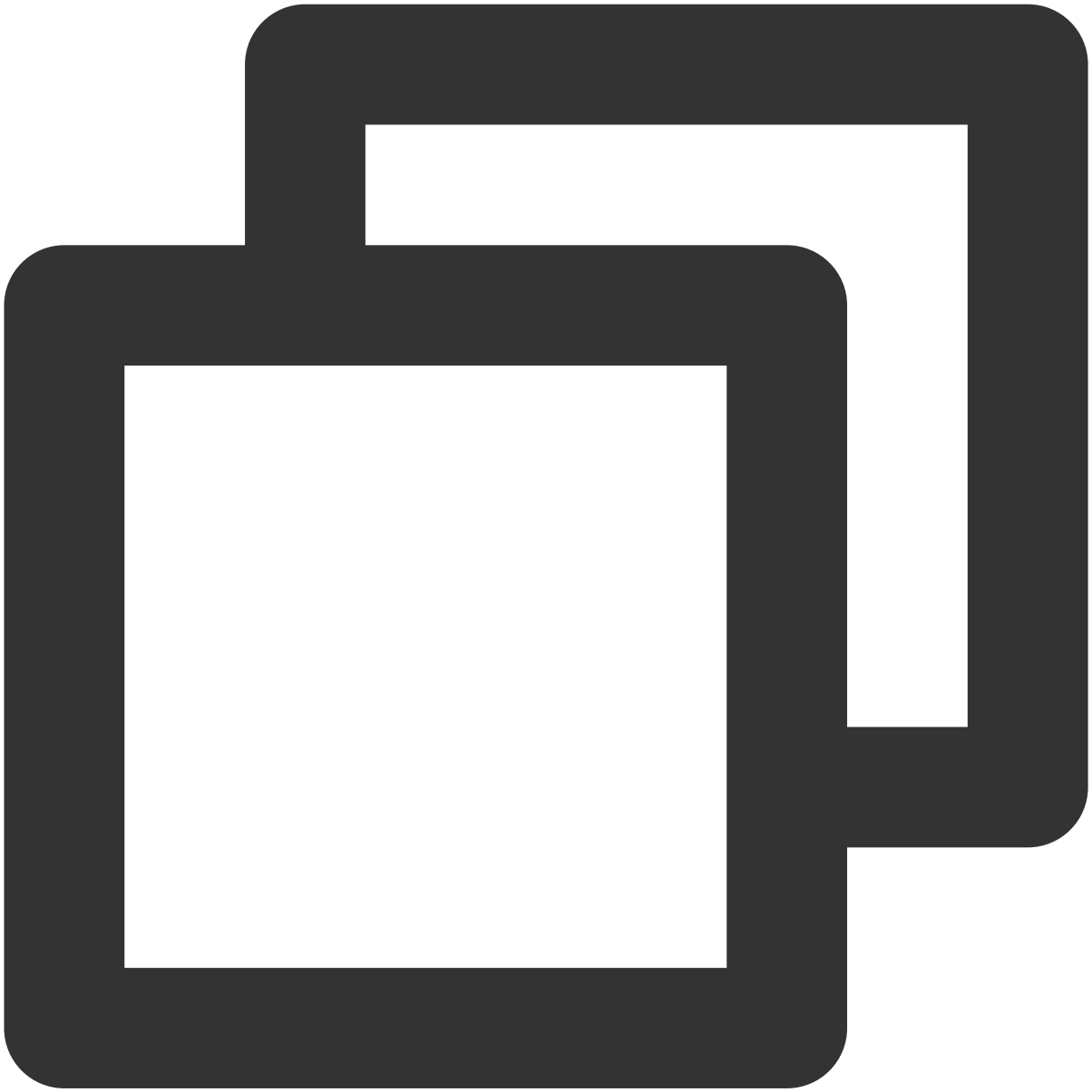
```
MAX_BY(<x> T, <y> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the x value associated with the maximum y value.

Return type: T

Example:



```
> SELECT max_by(x, y) FROM (VALUES (('a', 10)), (('b', 50)), (('c', 20))) AS tab(x,
 b
```

MIN

Function statement:

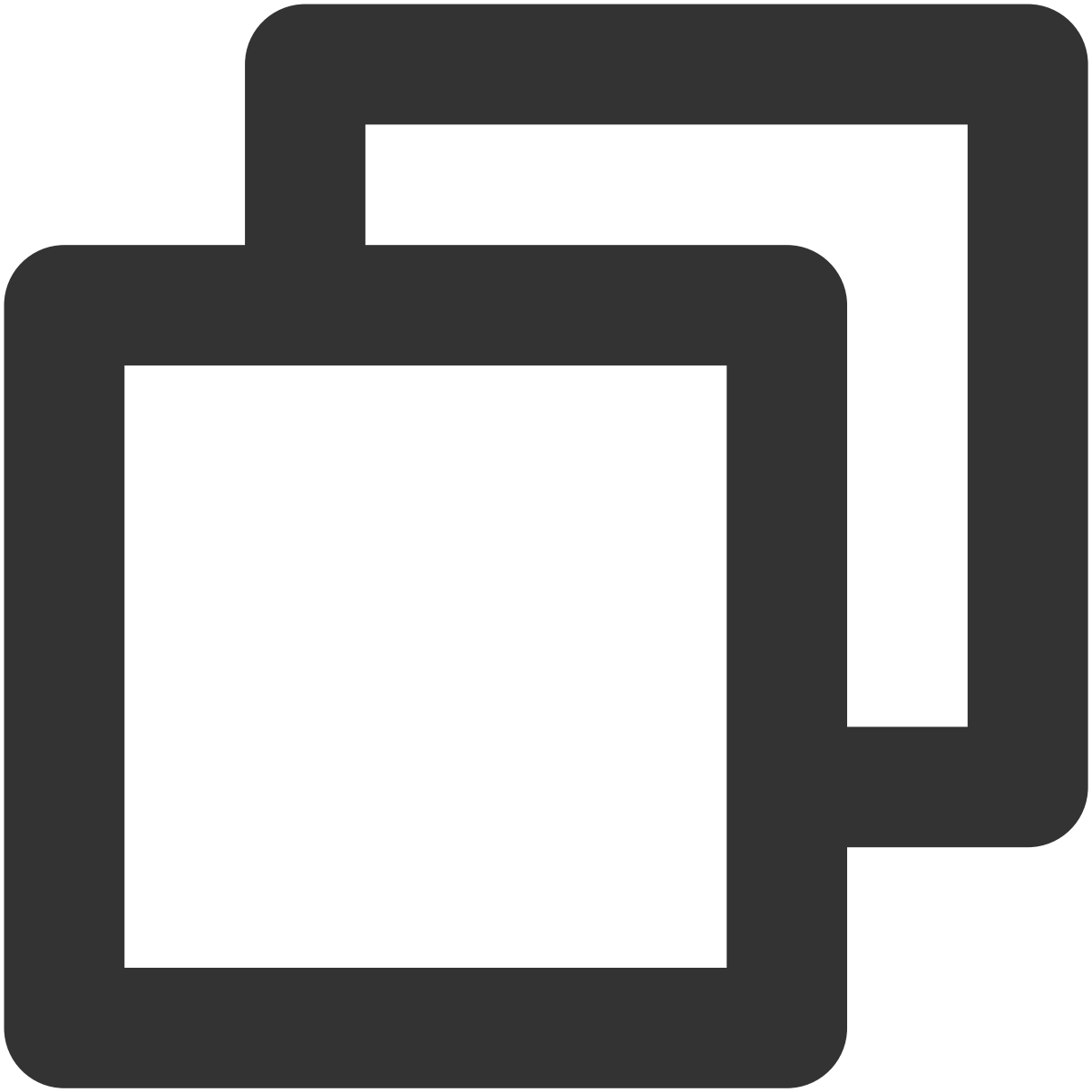
```
MIN(<col> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the minimum value of col.

Return type: same as col

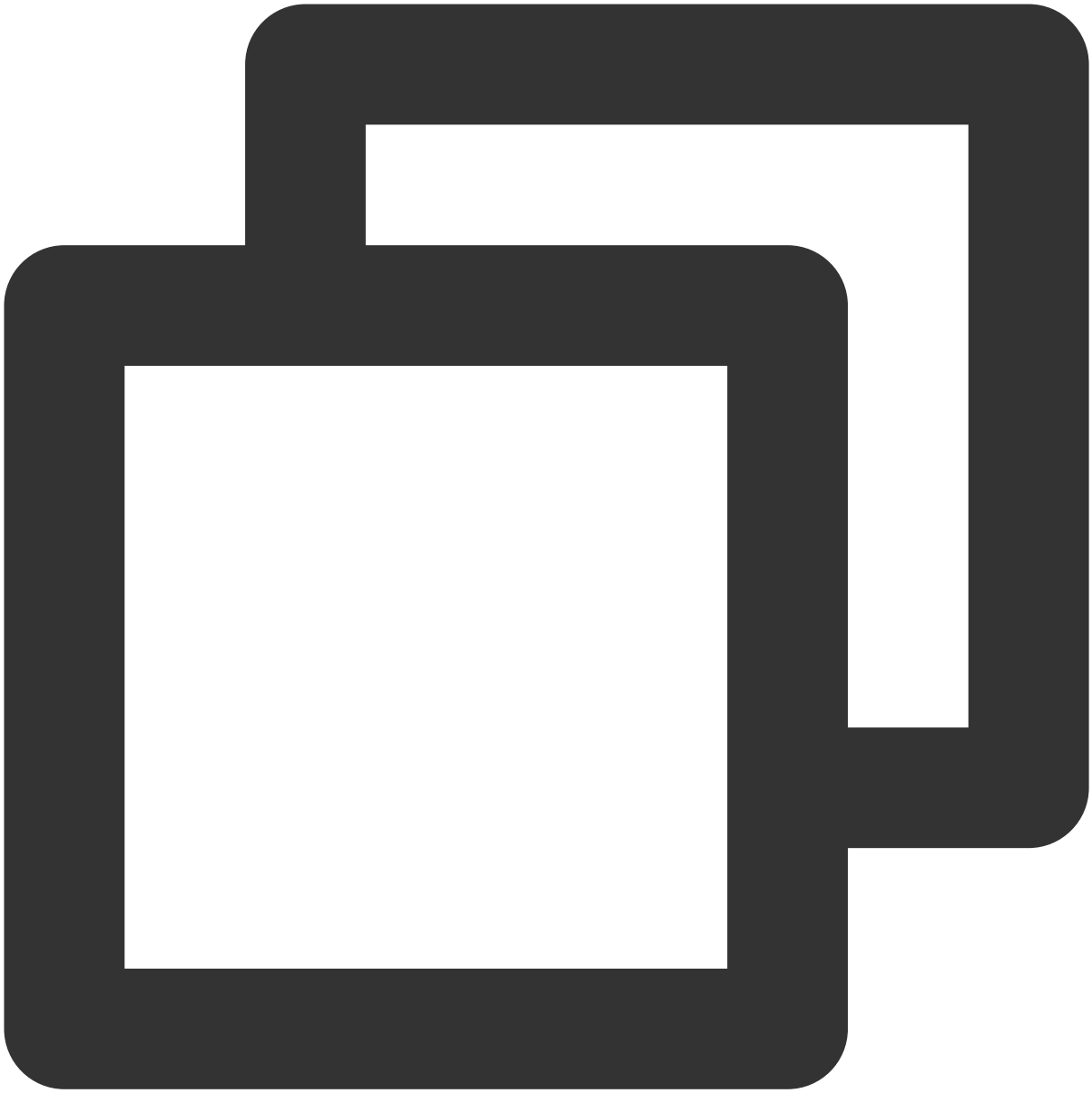
Example:



```
> SELECT min(col) FROM (VALUES (10), (50), (20)) AS tab(col);
 10
```

MIN_BY

Function statement:



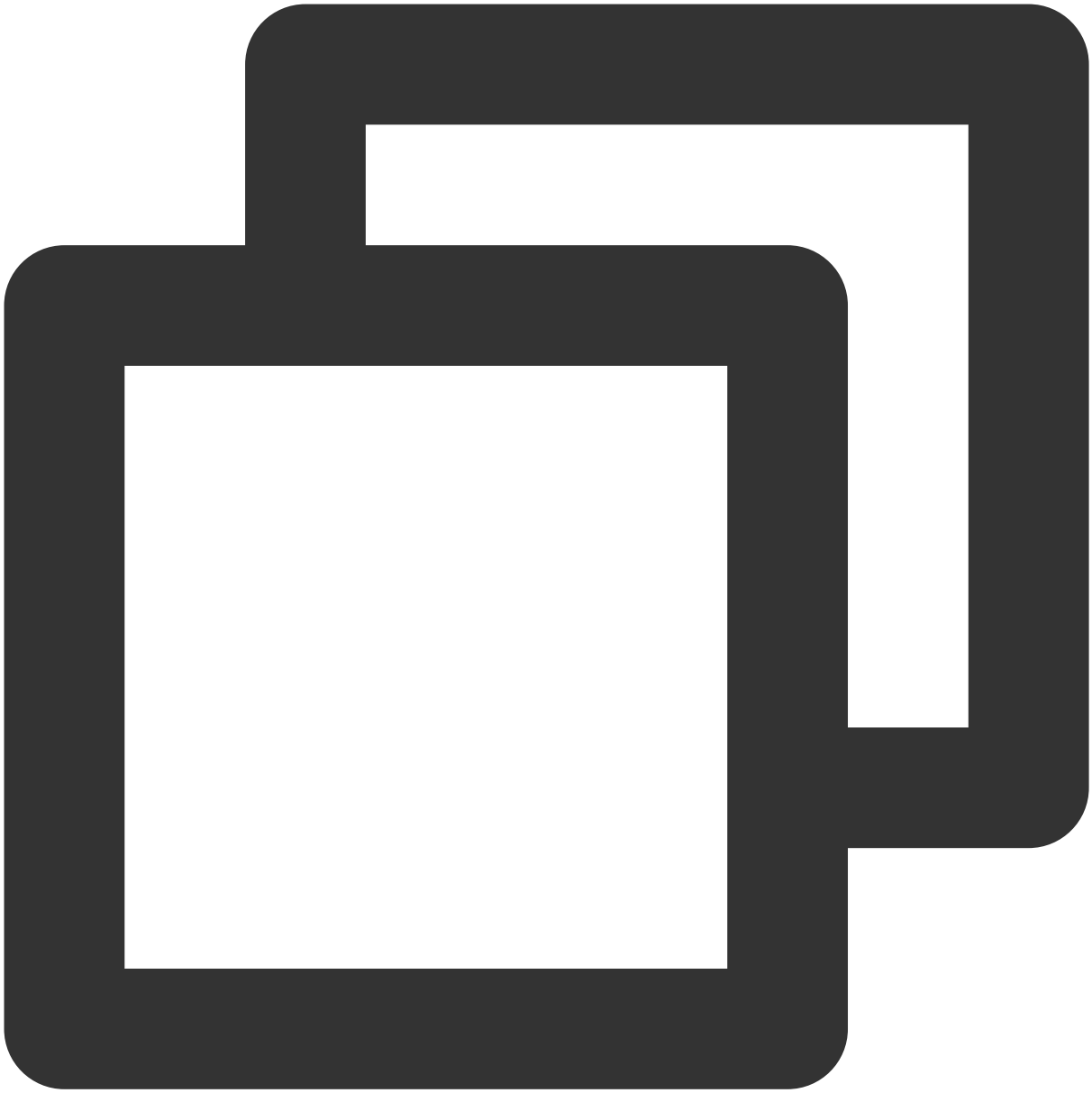
```
MIN_BY(<x> T, <y> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the x value associated with the minimum y value.

Return type: T

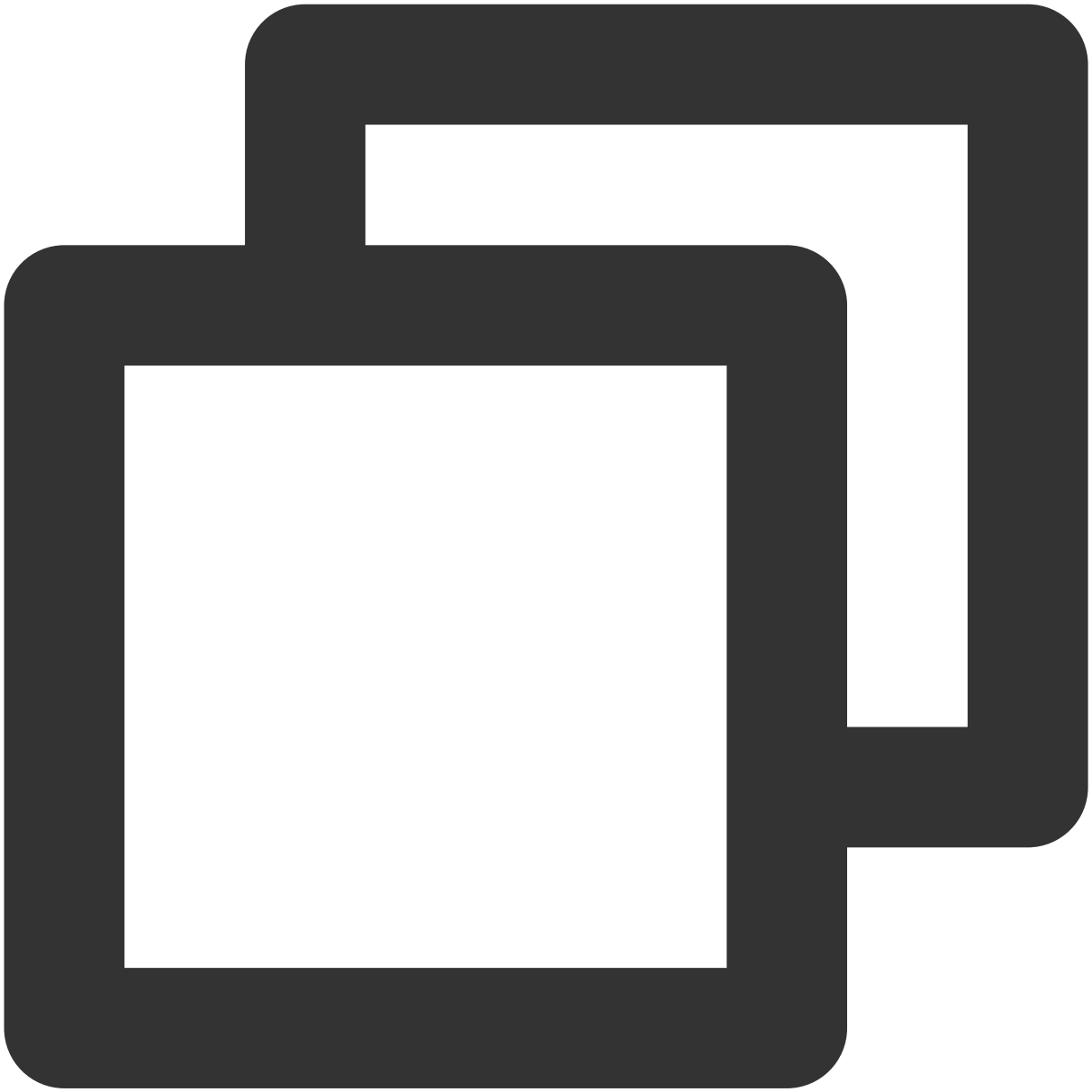
Example:



```
> SELECT min_by(x, y) FROM (VALUES (('a', 10)), (('b', 50)), (('c', 20))) AS tab(x,
 a
```

STD

Function statement:



```
STD(<col> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

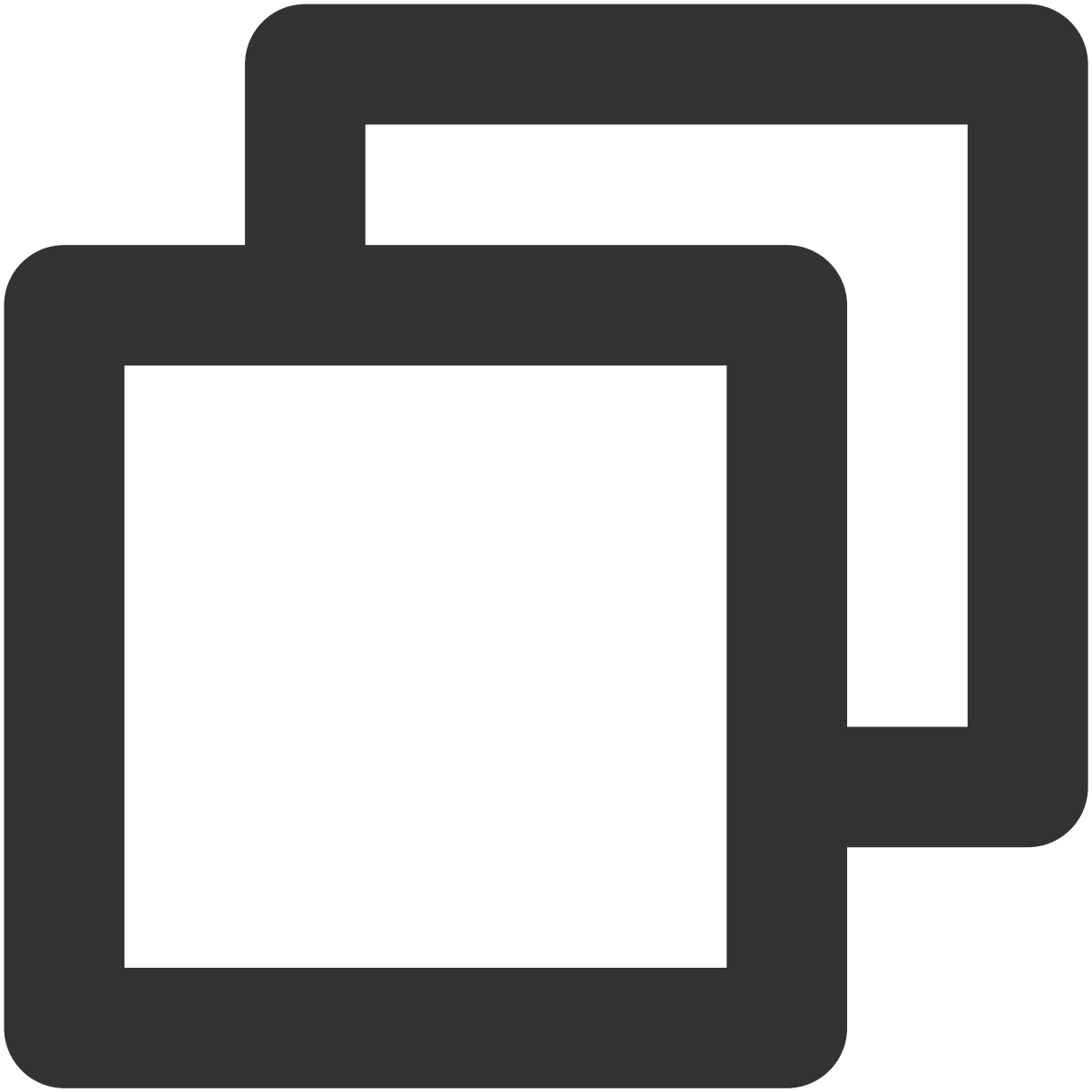
Usage Instructions

SparkSQL: Return the sample standard deviation calculated based on the group's values.

Presto: Return the population standard deviation calculated based on the group's values.

Return type: double

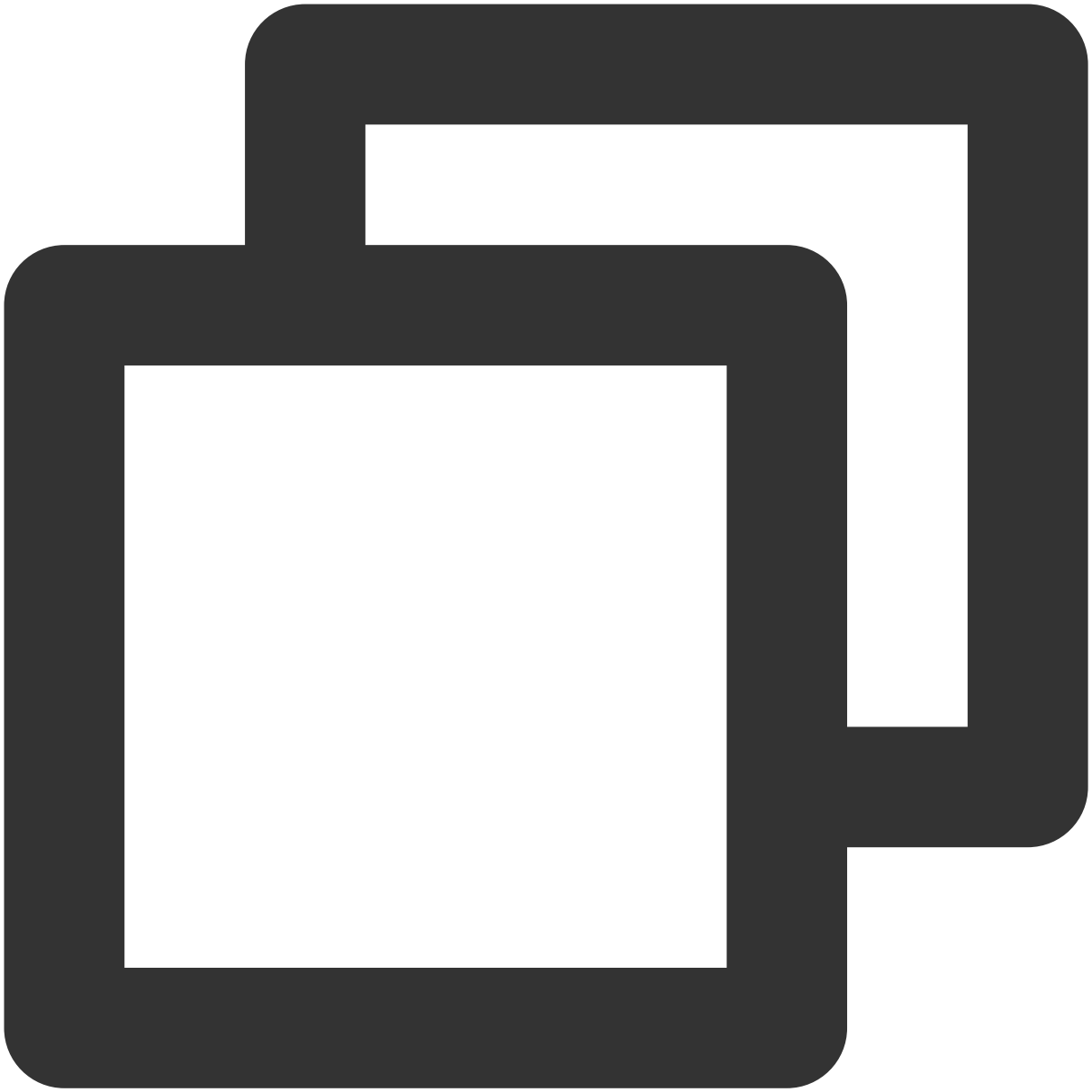
Example:



```
> SELECT std(col) FROM (VALUES (1), (2), (3)) AS tab(col);
 1.0
```

STDDEV

Function statement:



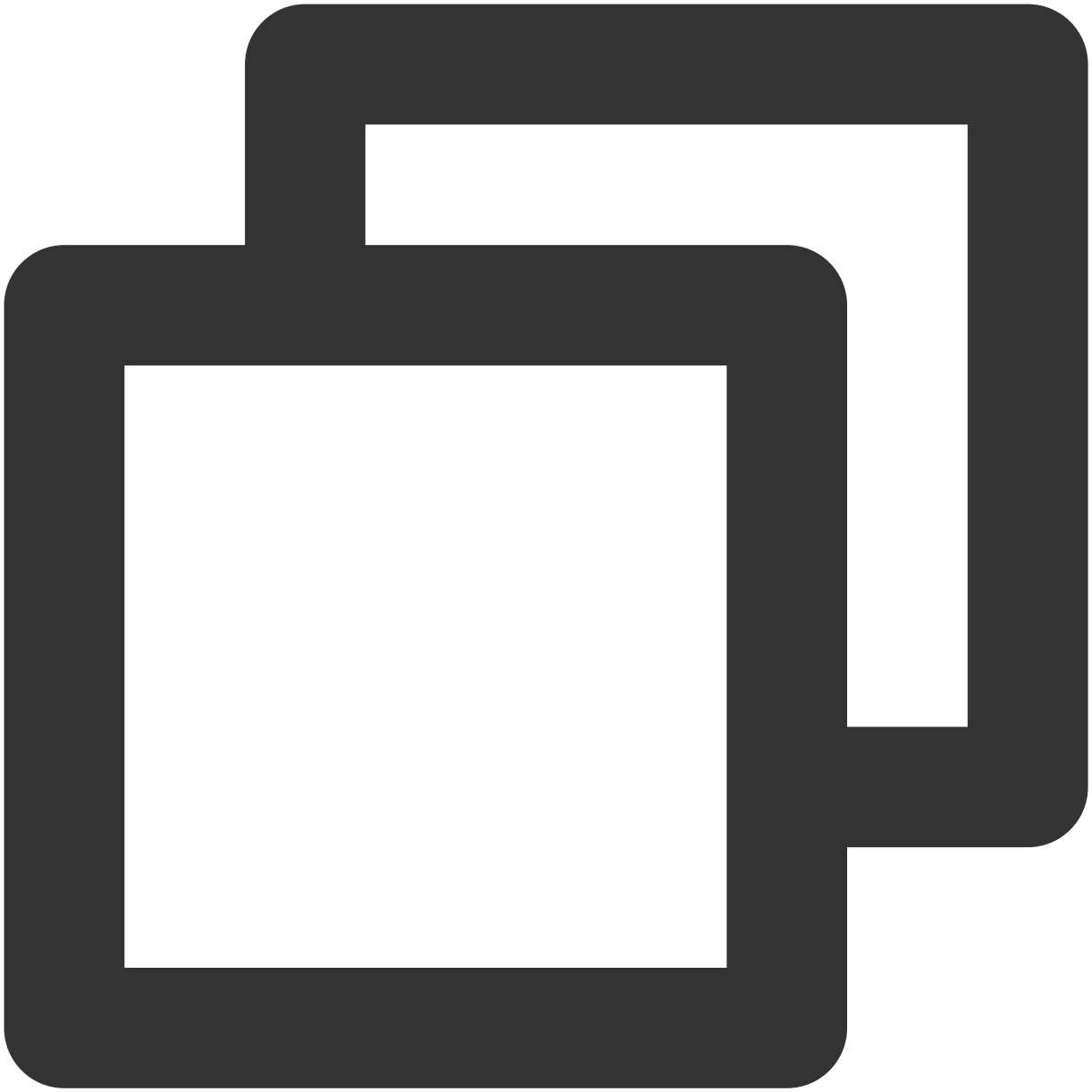
```
STDDEV(<col> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the sample standard deviation calculated based on the group's values.

Return type: double

Example:



```
> SELECT stddev(col) FROM (VALUES (1), (2), (3)) AS tab(col);
 1.0
```

STDDEV_POP

Function statement:

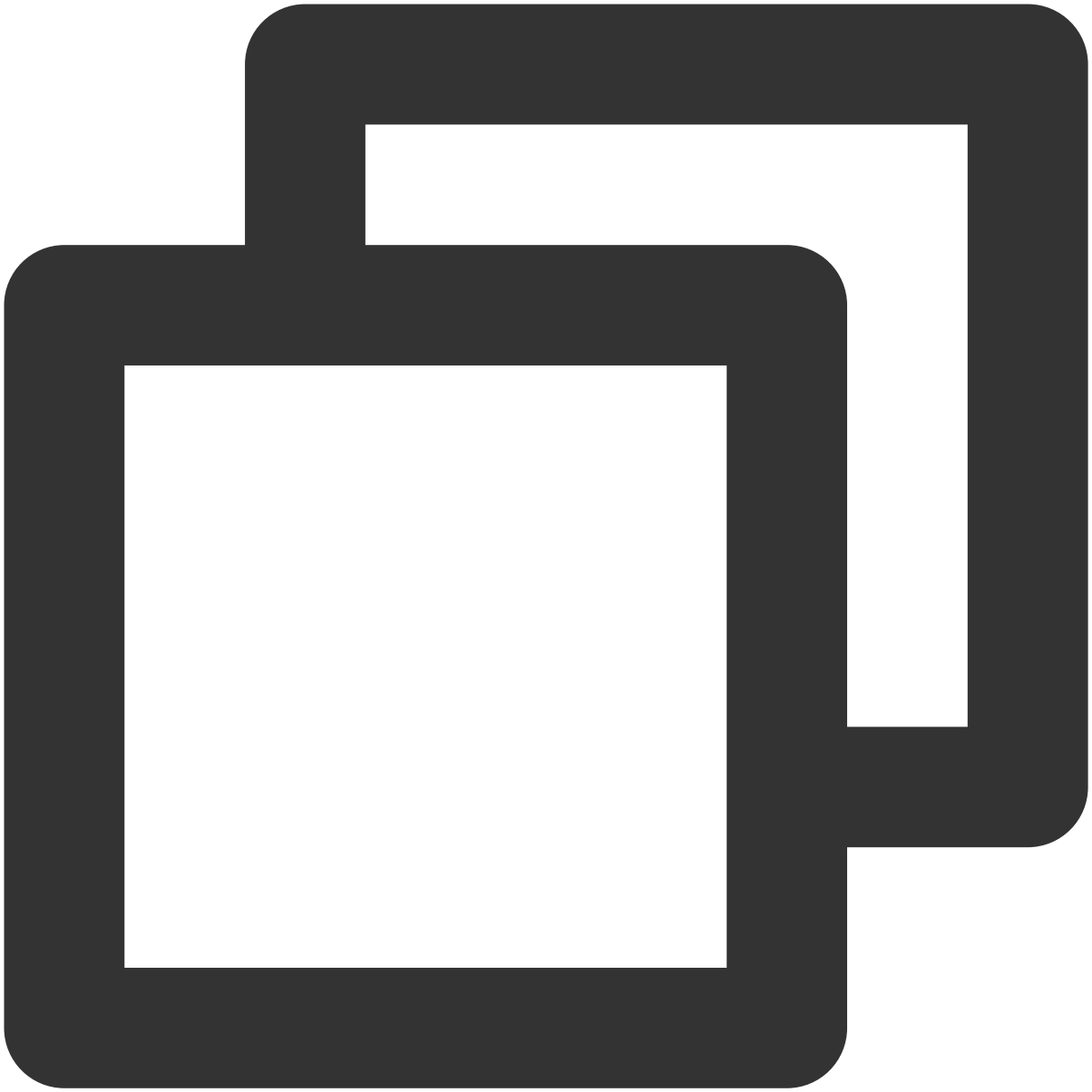
```
STDDEV_POP(<col> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the population standard deviation calculated based on the group's values.

Return type: double

Example:



```
> SELECT stddev_pop(col) FROM (VALUES (1), (2), (3)) AS tab(col);
 0.816496580927726
```

STDDEV_SAMP

Function statement:

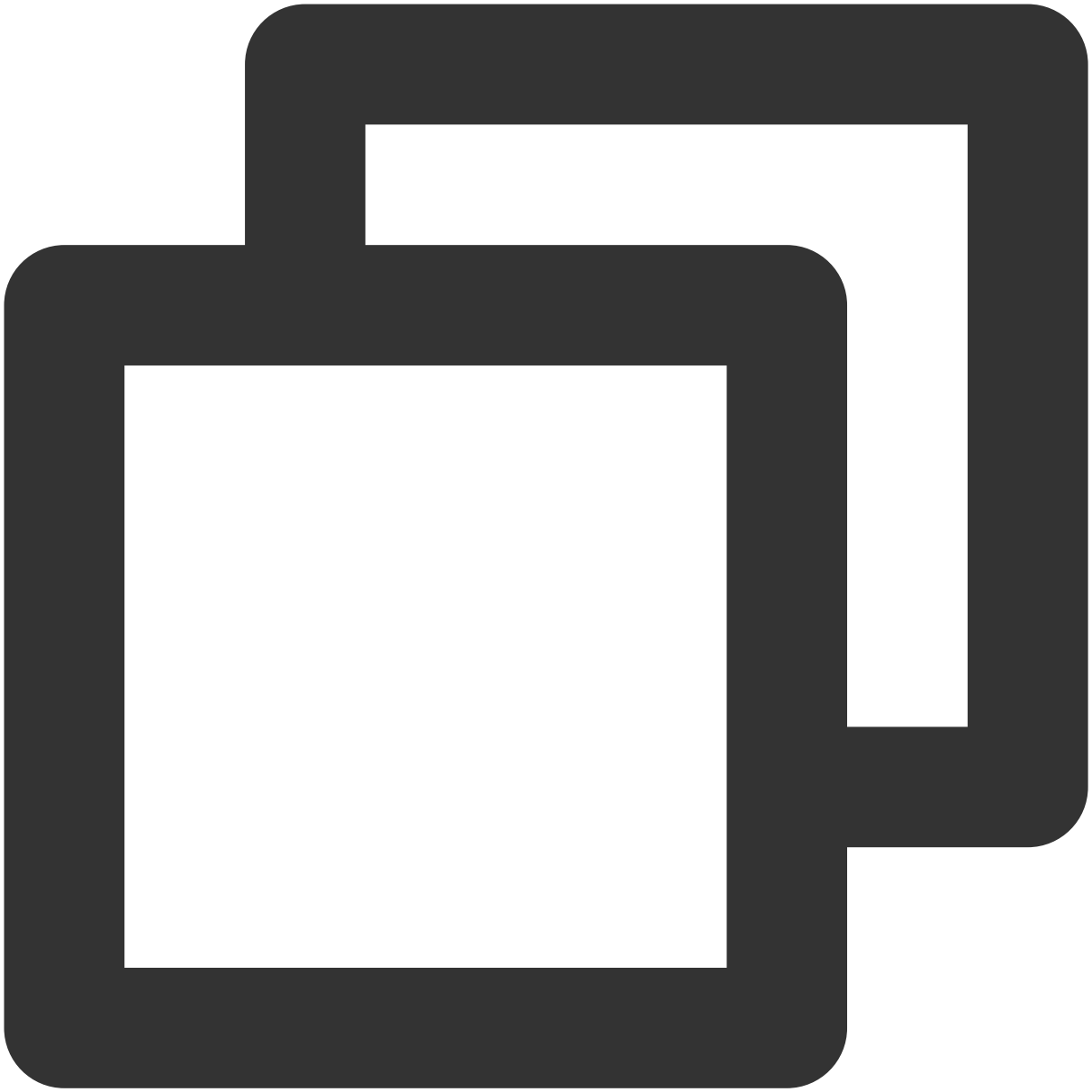
```
STDDEV_SAMP(<col> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the sample standard deviation calculated based on the group's values.

Return type: double

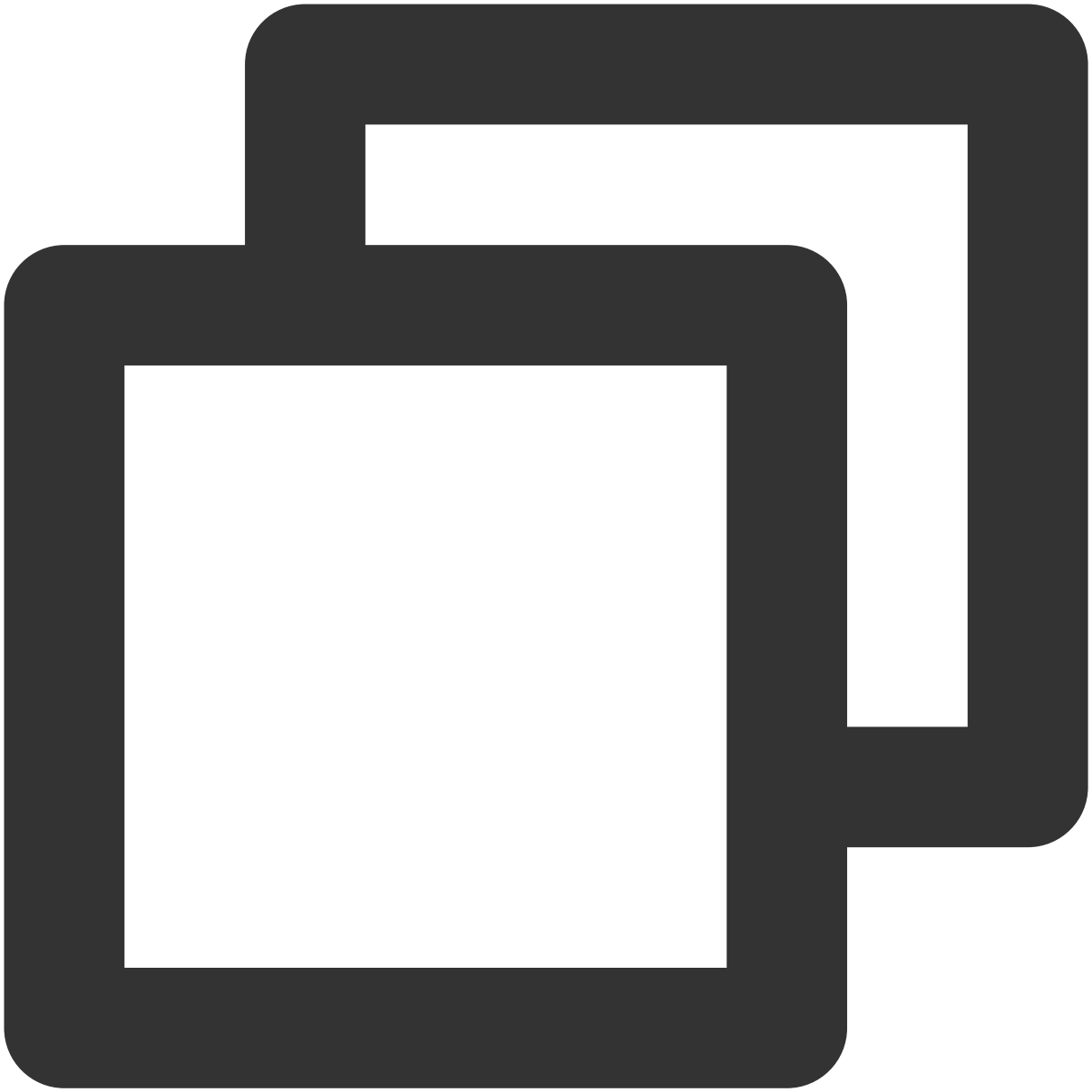
Example:



```
> SELECT stddev_samp(col) FROM (VALUES (1), (2), (3)) AS tab(col);
 1.0
```

SUM

Function statement:



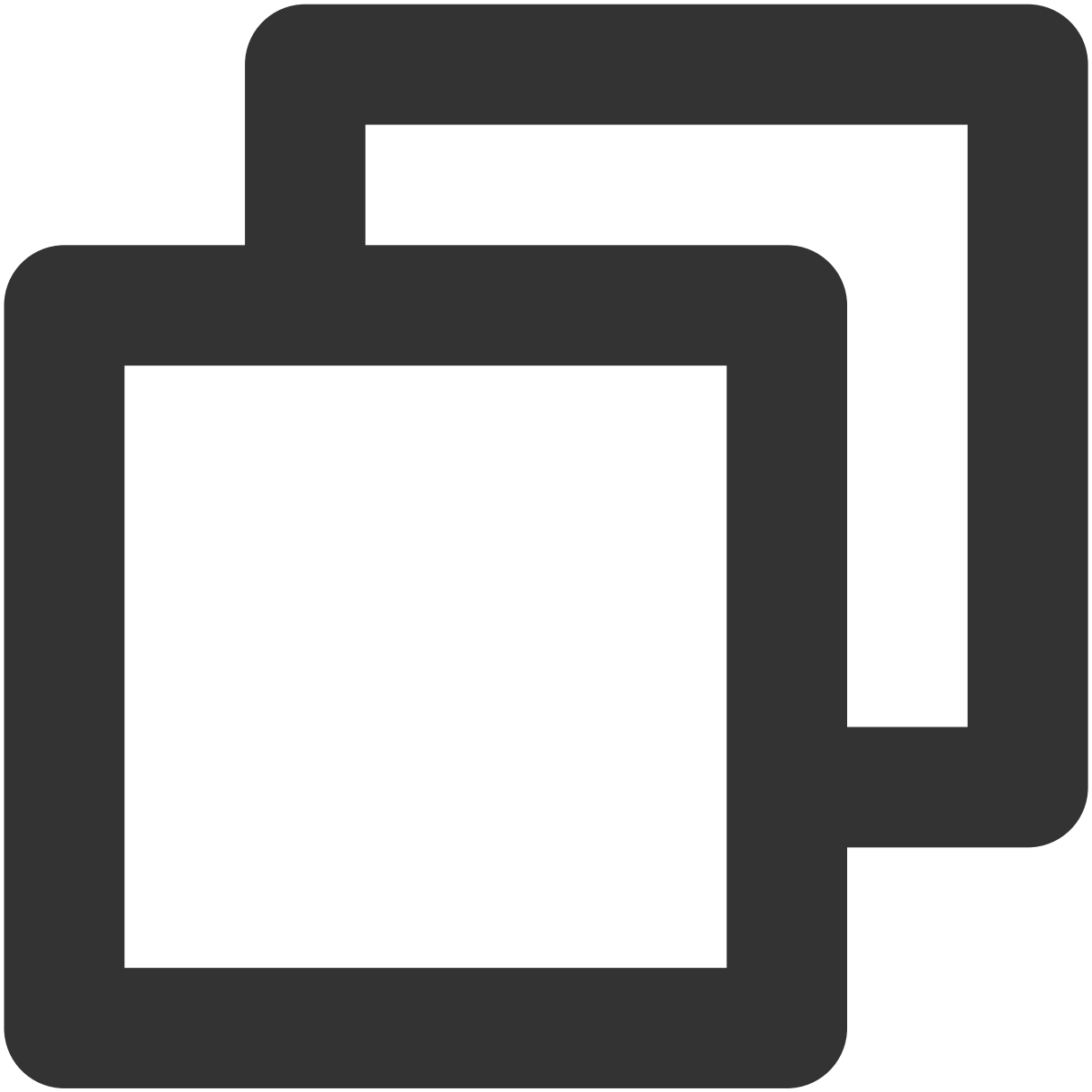
```
SUM(<col> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the sum calculated based on the group's values.

Return type: same as col

Example:



```
> SELECT sum(col) FROM (VALUES (5), (10), (15)) AS tab(col);
 30
> SELECT sum(col) FROM (VALUES (NULL), (10), (15)) AS tab(col);
 25
> SELECT sum(col) FROM (VALUES (NULL), (NULL)) AS tab(col);
 NULL
```

VARIANCE

Function statement:

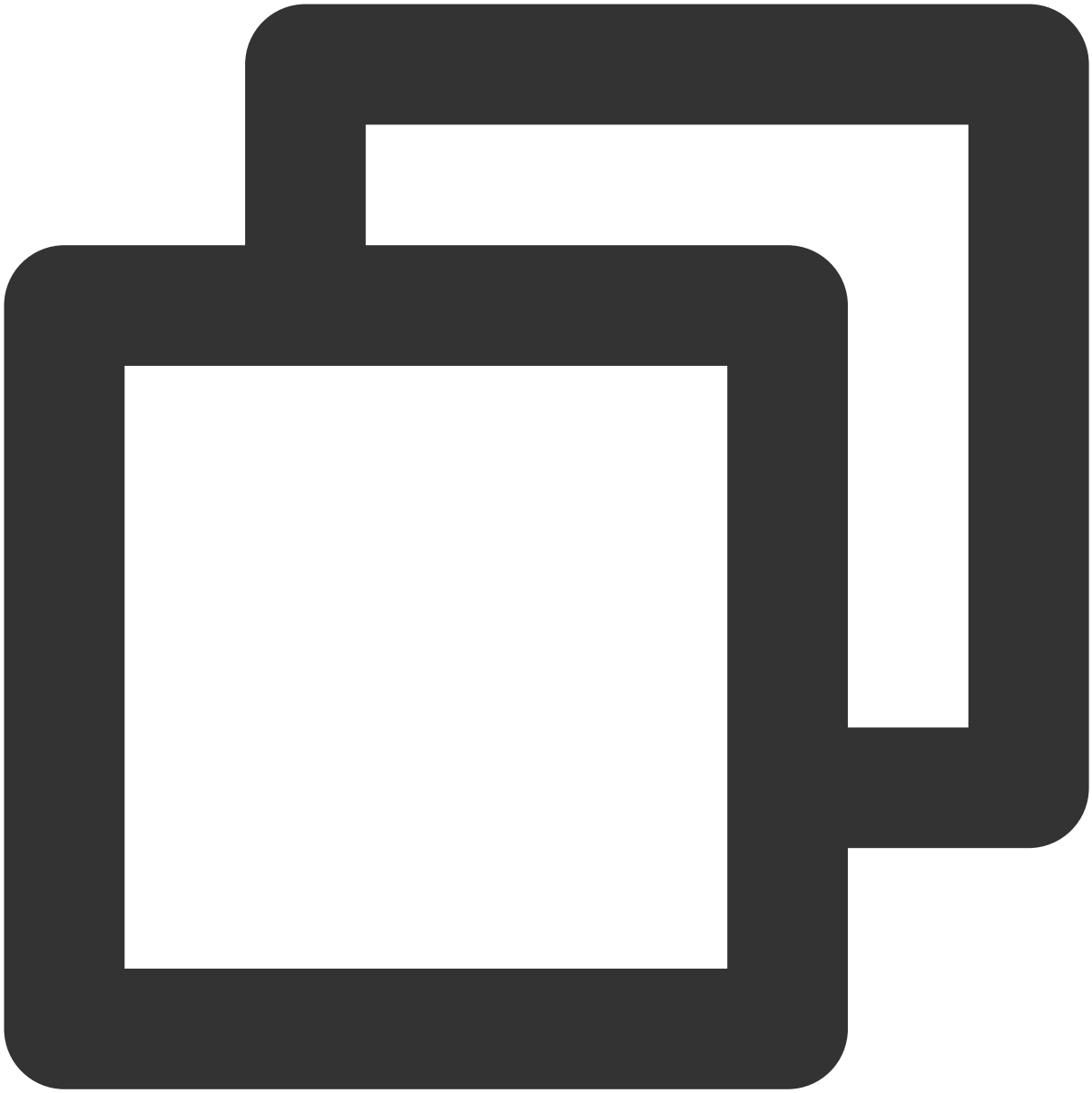
```
VARIANCE(<col> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the sample variance calculated based on the group's values.

Return type: double

Example:



```
> SELECT VARIANCE(col) FROM (VALUES (1), (2), (3)) AS tab(col);
 1.0
```

VAR_POP

Function statement:

```
VAR_POP(<col> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the population variance calculated based on the group's values.

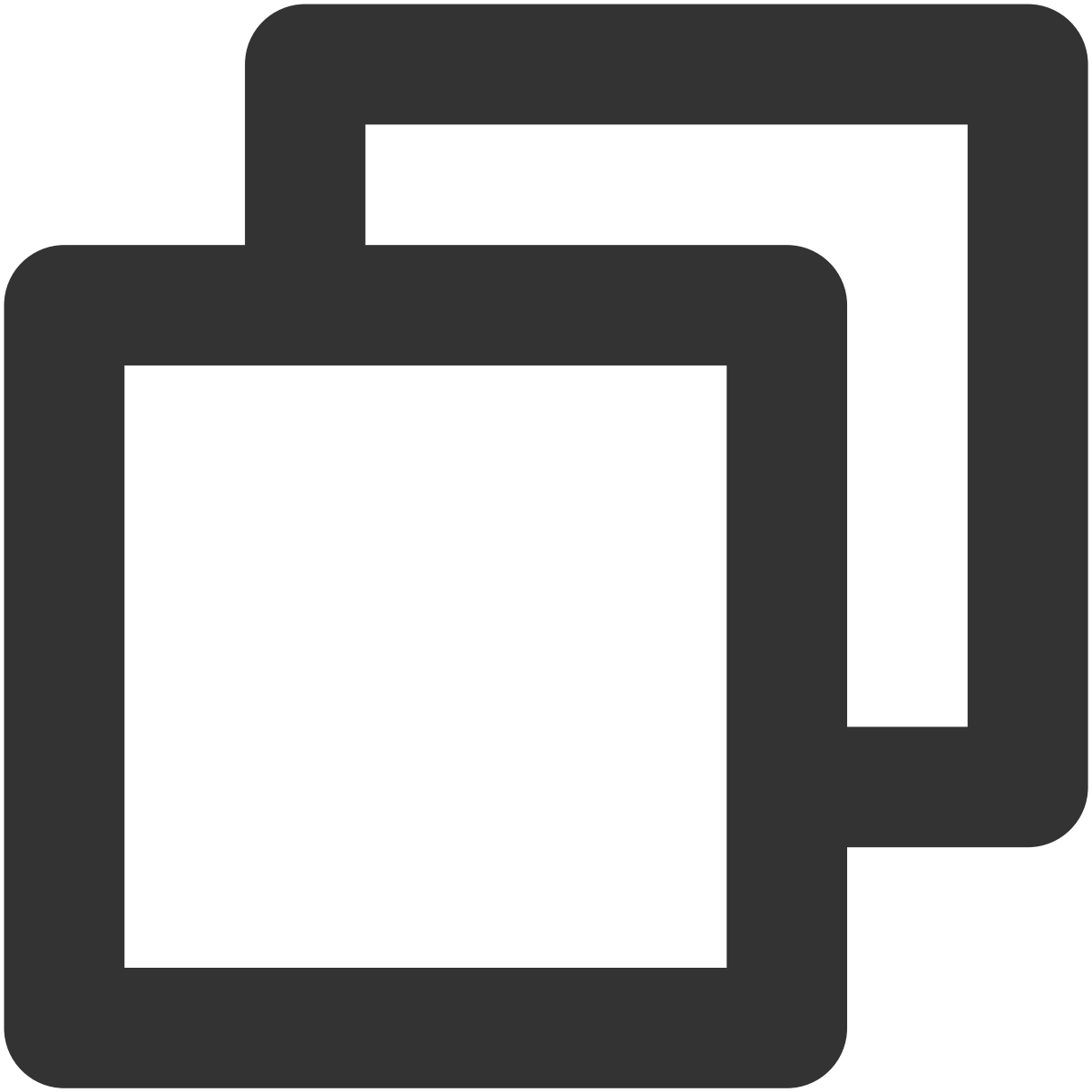
Return type: double

Example:

```
> SELECT var_pop(col) FROM (VALUES (1), (2), (3)) AS tab(col);
 0.6666666666666666
```

VAR_SAMP

Function statement:



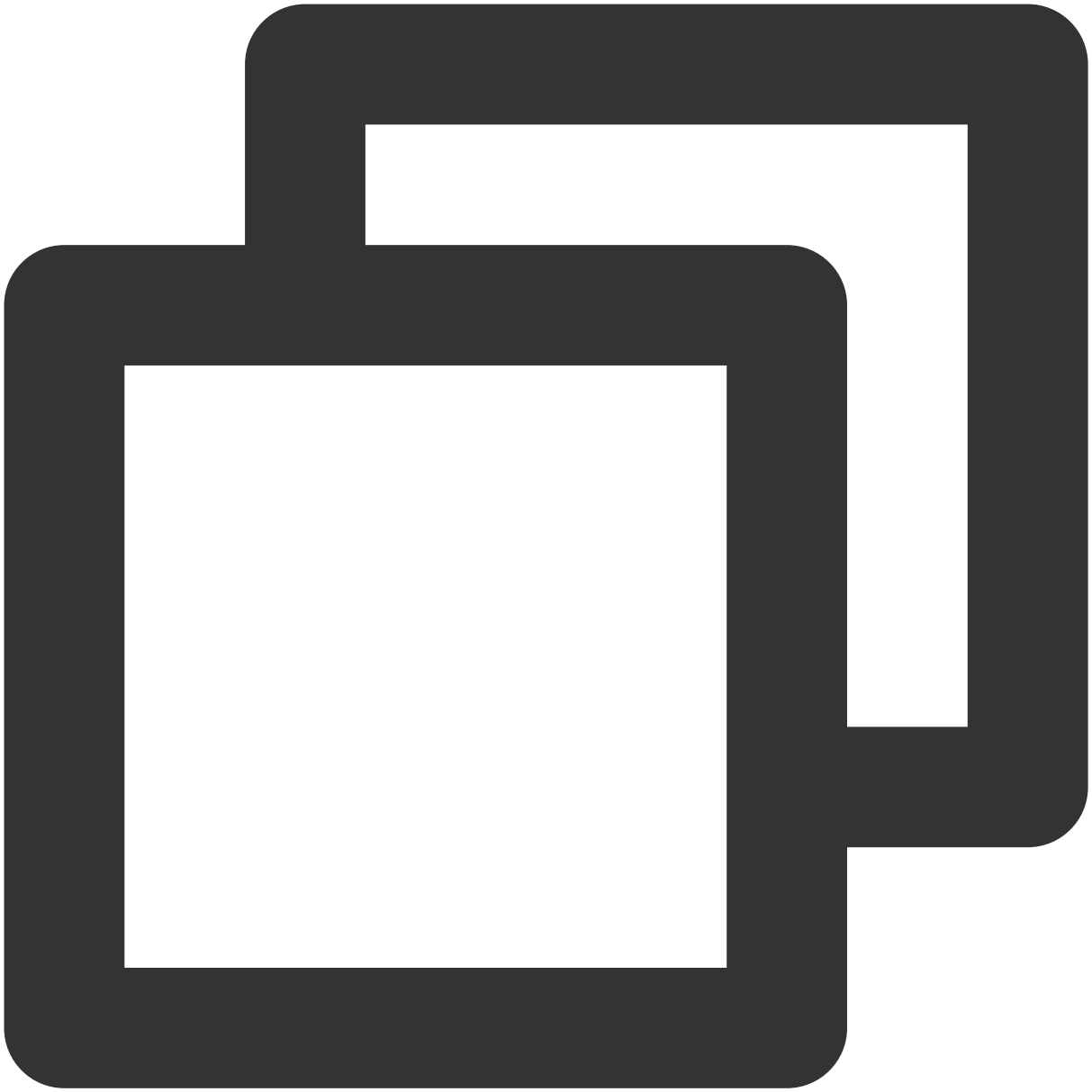
```
VAR_SAMP(<col> integer|double|decimal)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the sample variance calculated based on the group's values.

Return type: double

Example:



```
> SELECT var_samp(col) FROM (VALUES (1), (2), (3)) AS tab(col);
 1.0
```

HISTOGRAM_NUMERIC

Function statement:

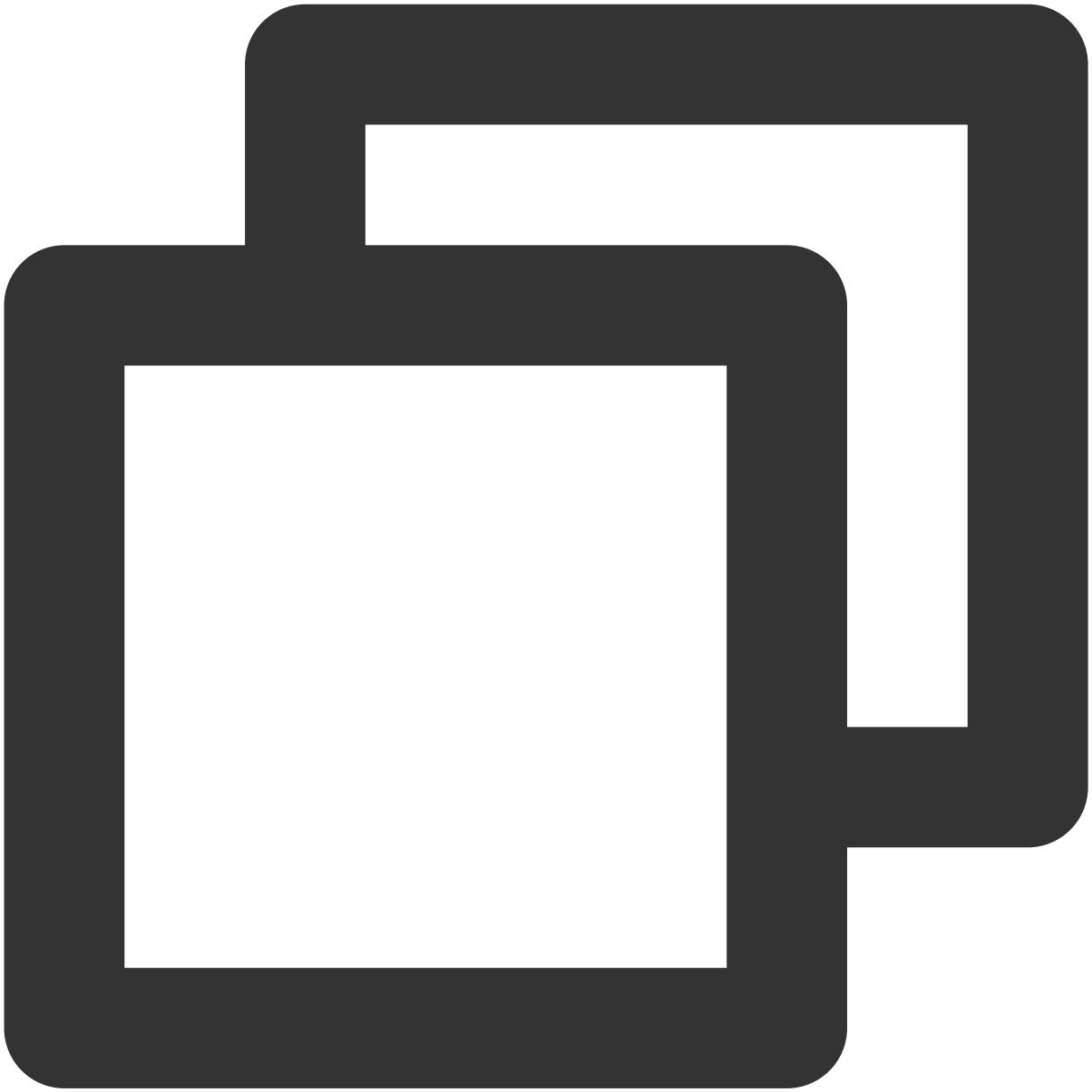
```
HISTOGRAM_NUMERIC(<col> integer, <nb> integer)
```

Supported engine: Presto

Usage instructions: Compute a histogram of a numeric column in the group using nb bins with non-uniform intervals. The output is an array of nb (x, y) coordinates, representing the center and height of the bins.

Return type: array<struct {'x','y'}>

Example:



```
> SELECT histogram_numeric(col, 5) FROM (VALUES (0), (1), (2), (10)) AS tab(col);
 [{"x":0,"y":1.0},{"x":1,"y":1.0},{"x":2,"y":1.0},{"x":10,"y":1.0}]
```

COLLECT_LIST

Function statement:

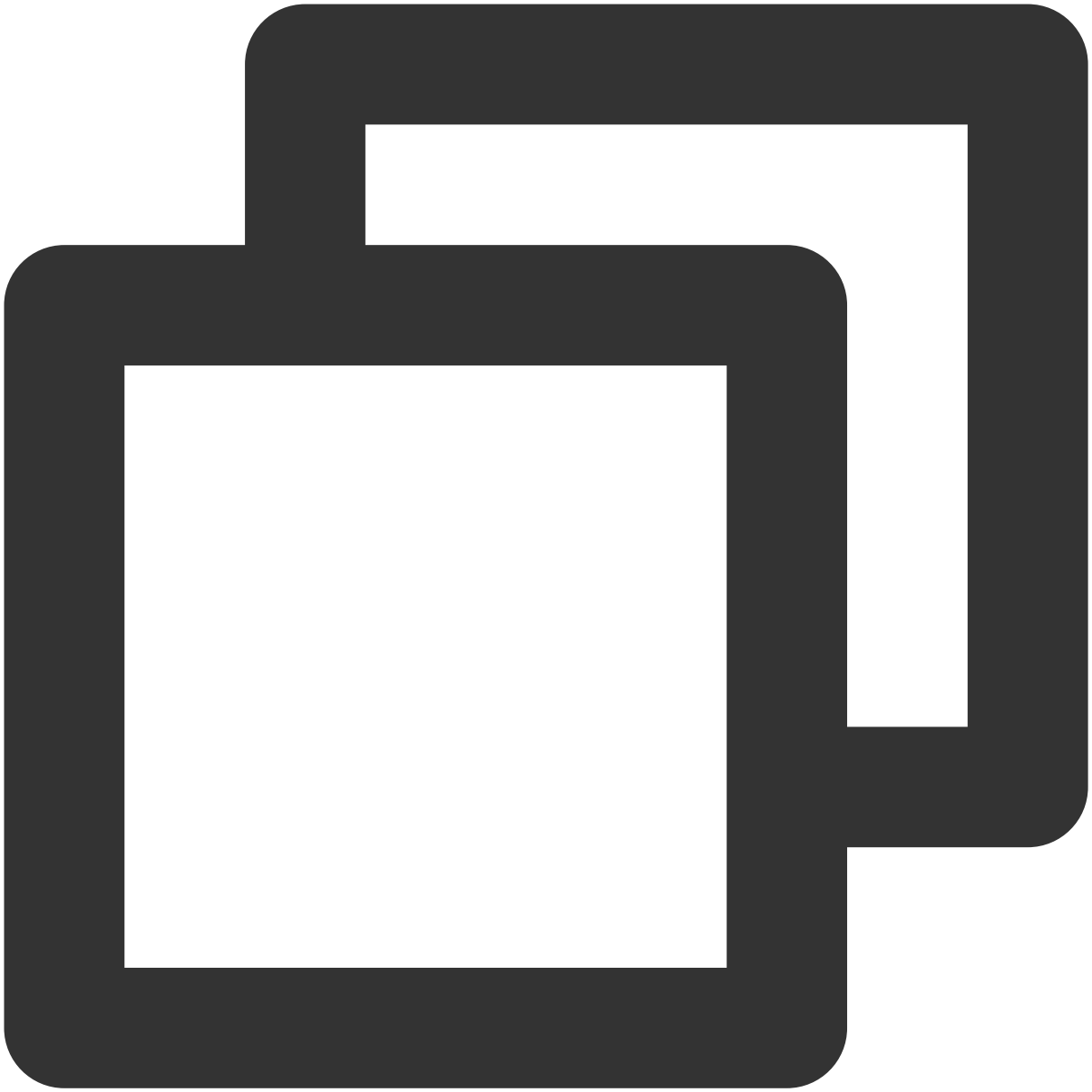
```
COLLECT_LIST(<col> T)
```

Supported engines: SparkSQL and Presto

Usage instructions: Collect and return a list of non-unique elements.

Return type: array<T>

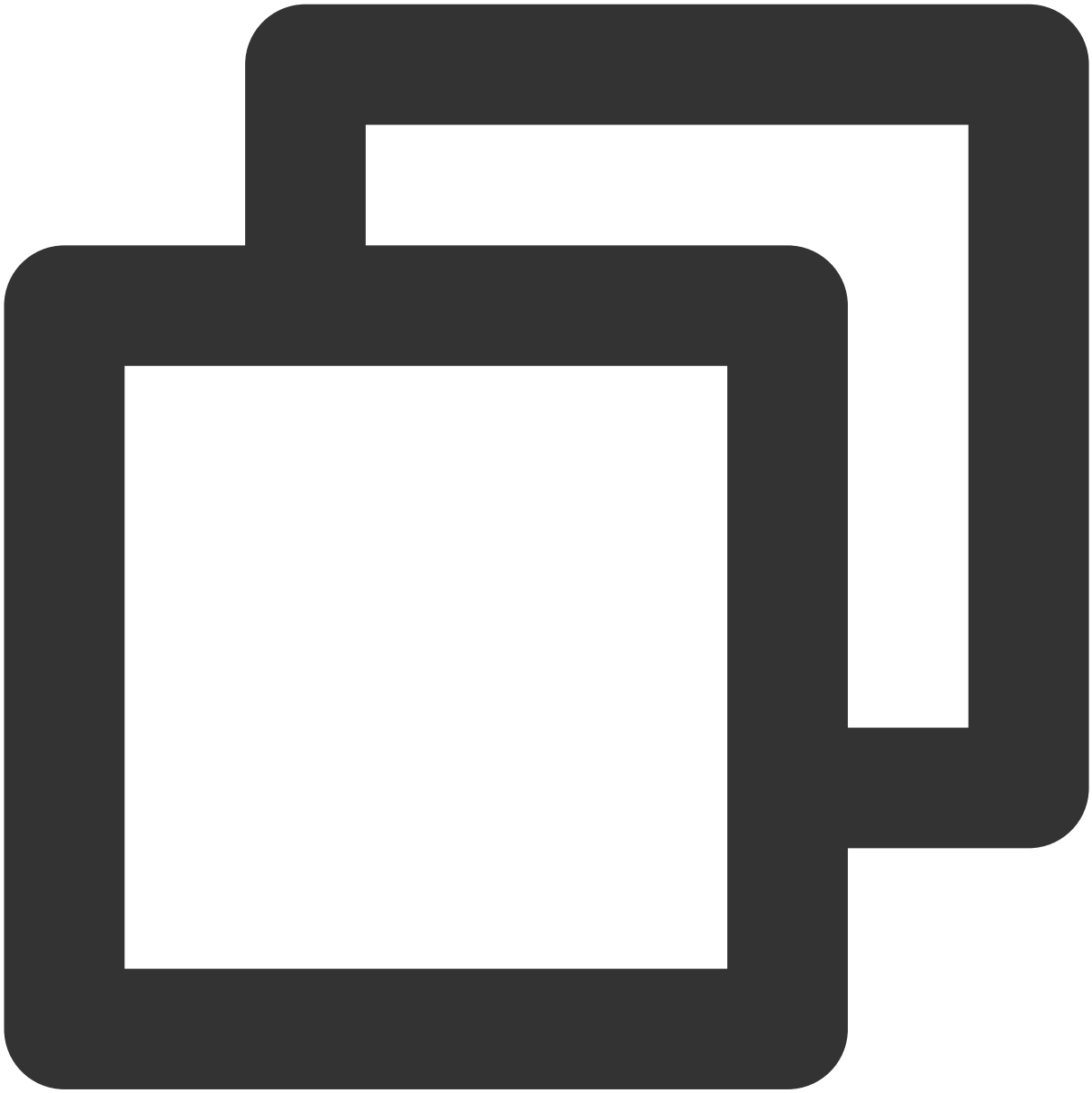
Example:



```
> SELECT collect_list(col) FROM (VALUES (1), (2), (1)) AS tab(col);
 [1,2,1]
```

COLLECT_SET

Function statement:



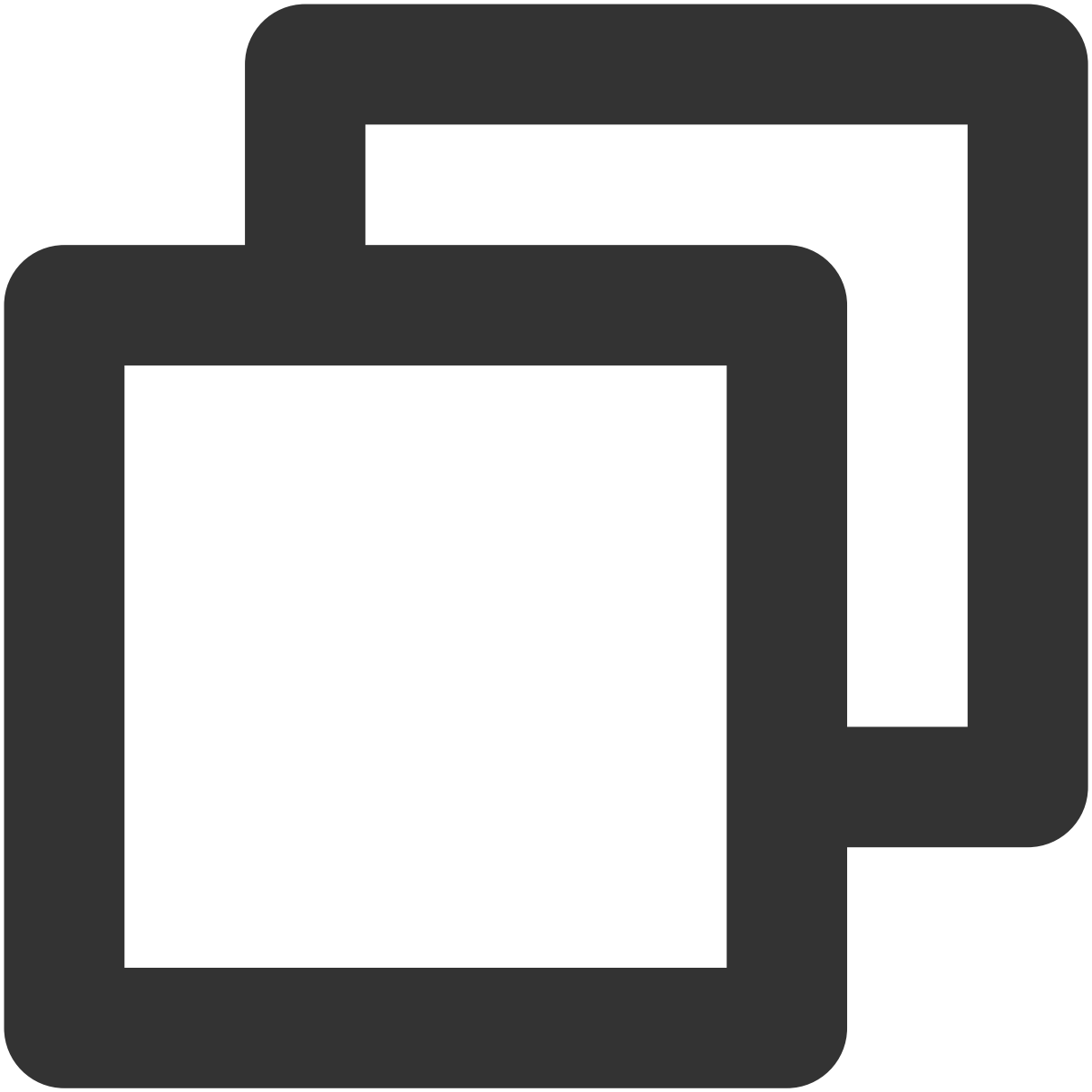
```
COLLECT_SET(<col> T)
```

Supported engines: SparkSQL and Presto

Usage instructions: Collect and return a set of unique elements.

Return type: array<T>

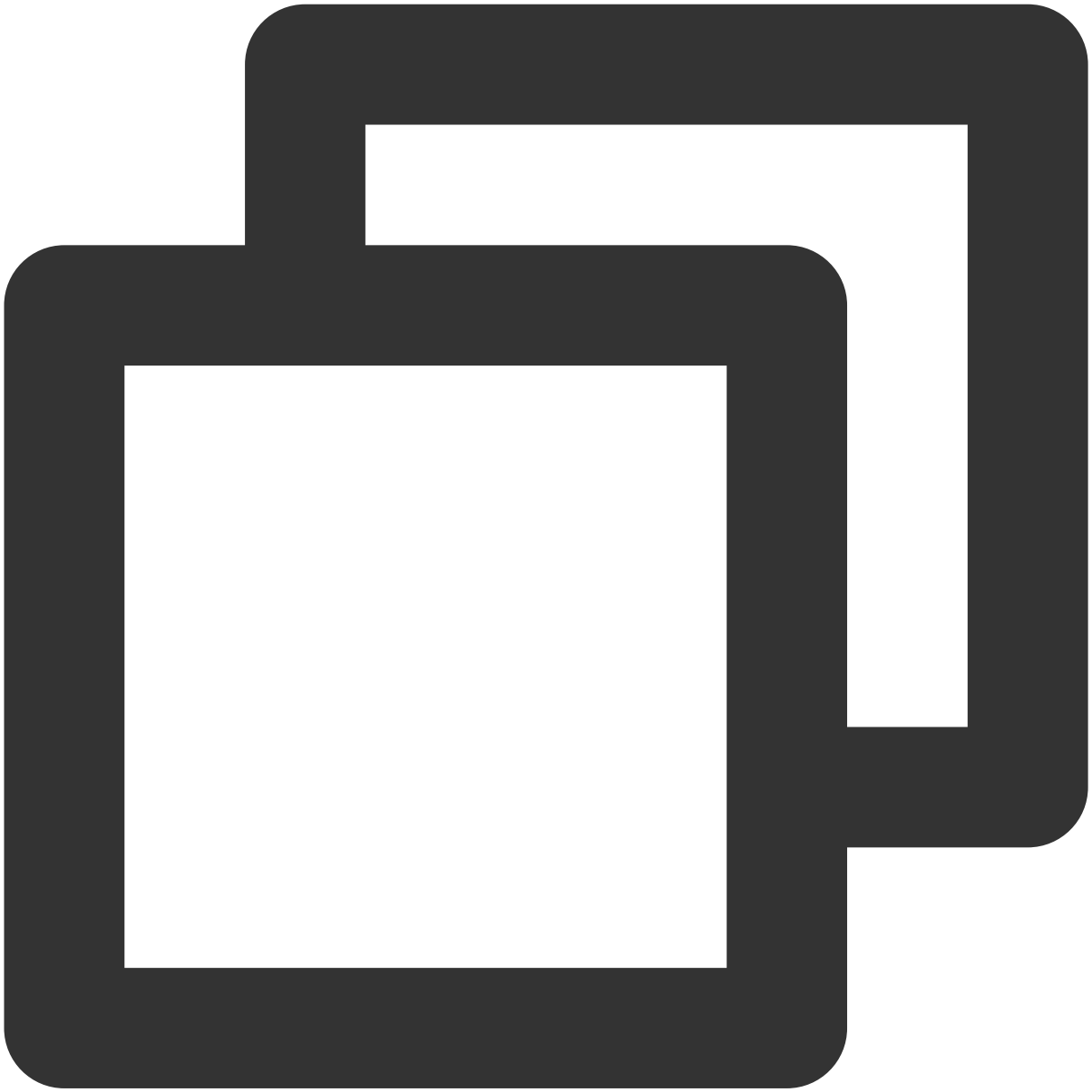
Example:



```
> SELECT collect_set(col) FROM (VALUES (1), (2), (1)) AS tab(col);
 [1,2]
```

COUNT_MIN_SKETCH

Function statement:



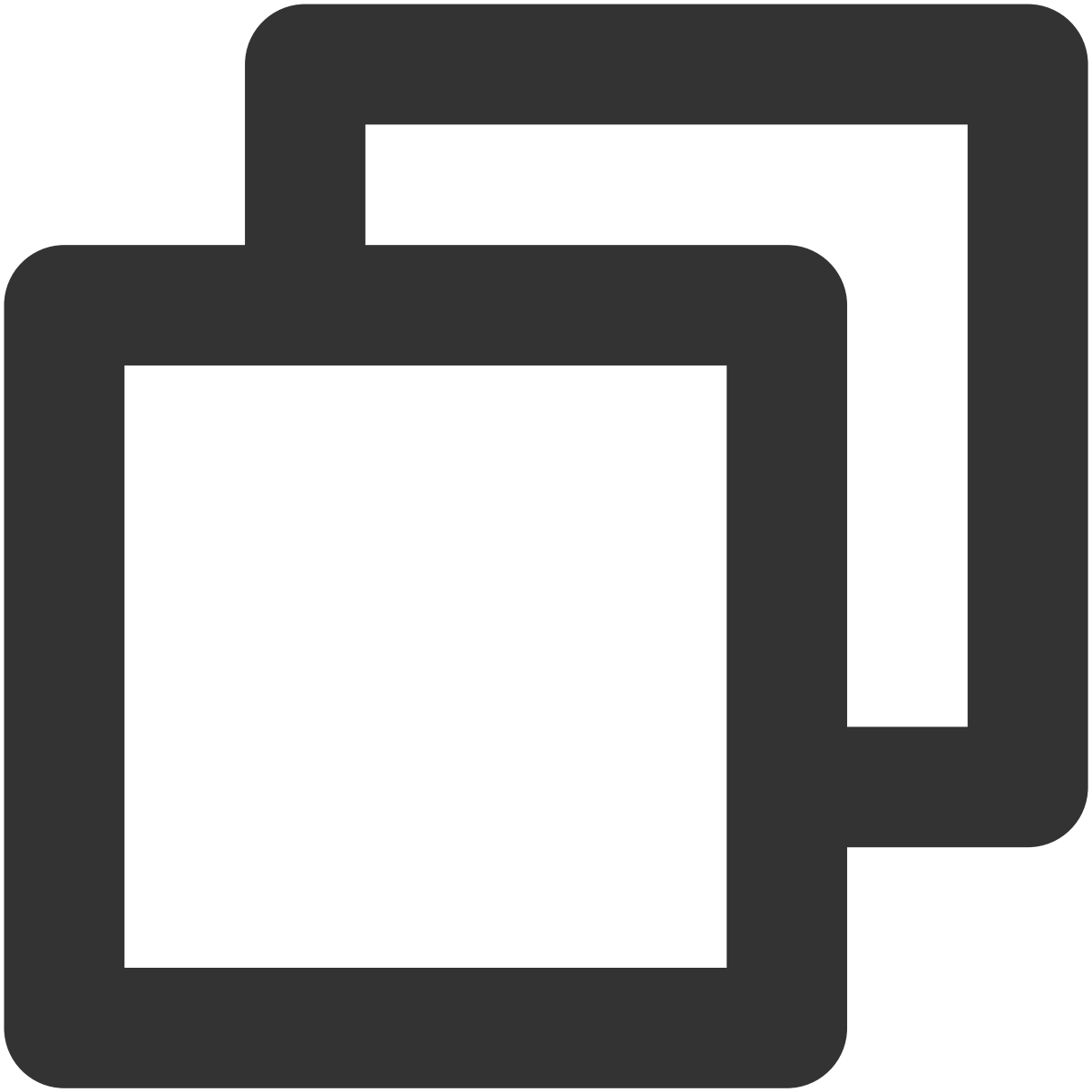
```
COUNT_MIN_SKETCH(<col> T, <eps> double, <confidence> double, <seed> integer)
```

Supported engine: SparkSQL

Usage instructions: Return the Count-Min Sketch of a column with the given esp, confidence, and seed. The result is binary and can be deserialized into CountMinSketch before use.

Return type: binary

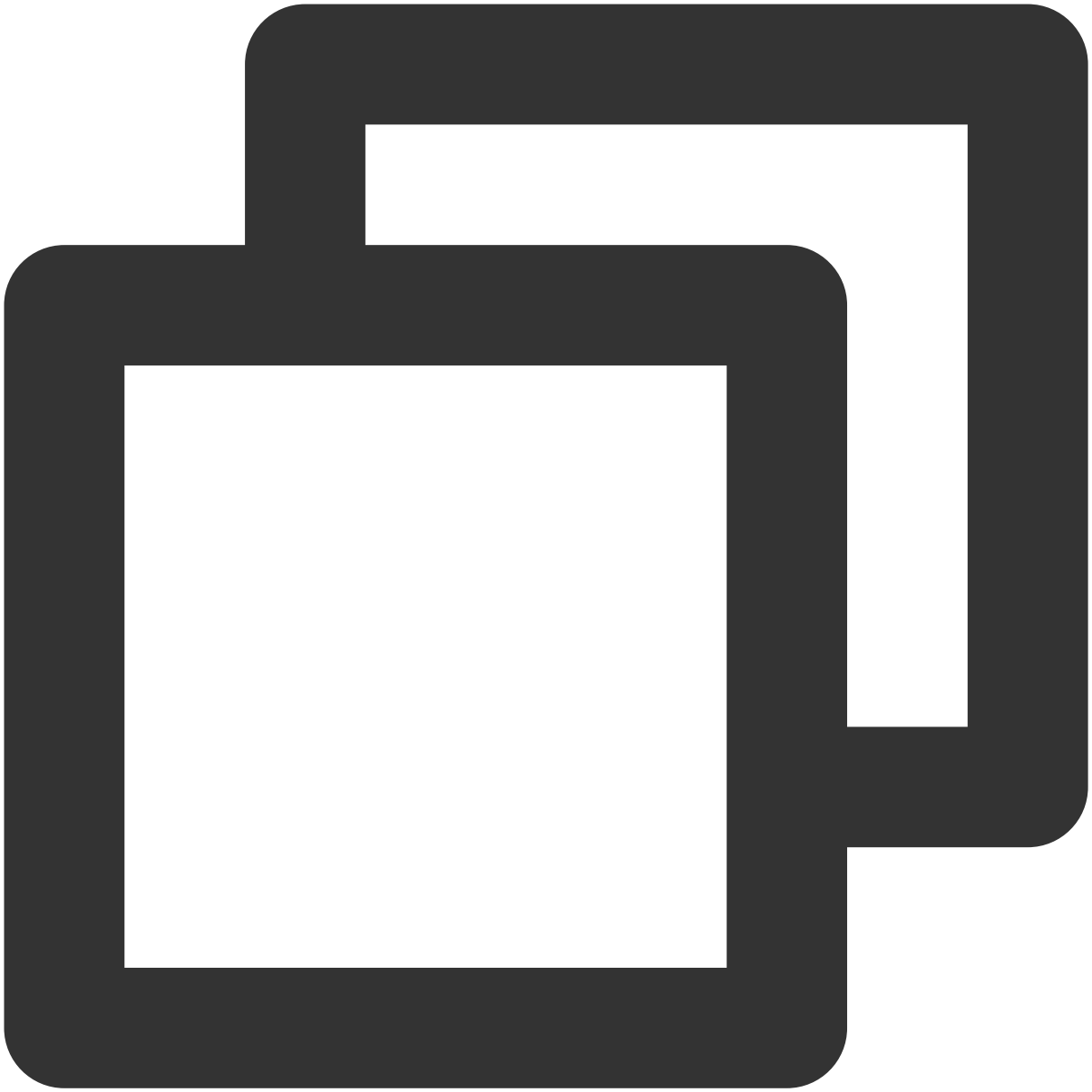
Example:



```
> SELECT hex(count_min_sketch(col, 0.5d, 0.5d, 1)) FROM (VALUES (1), (2), (1)) AS t
00000001000000000000000300000001000000040000000005D8D6AB90000000000000000000000000000
```

EVERY

Function statement:



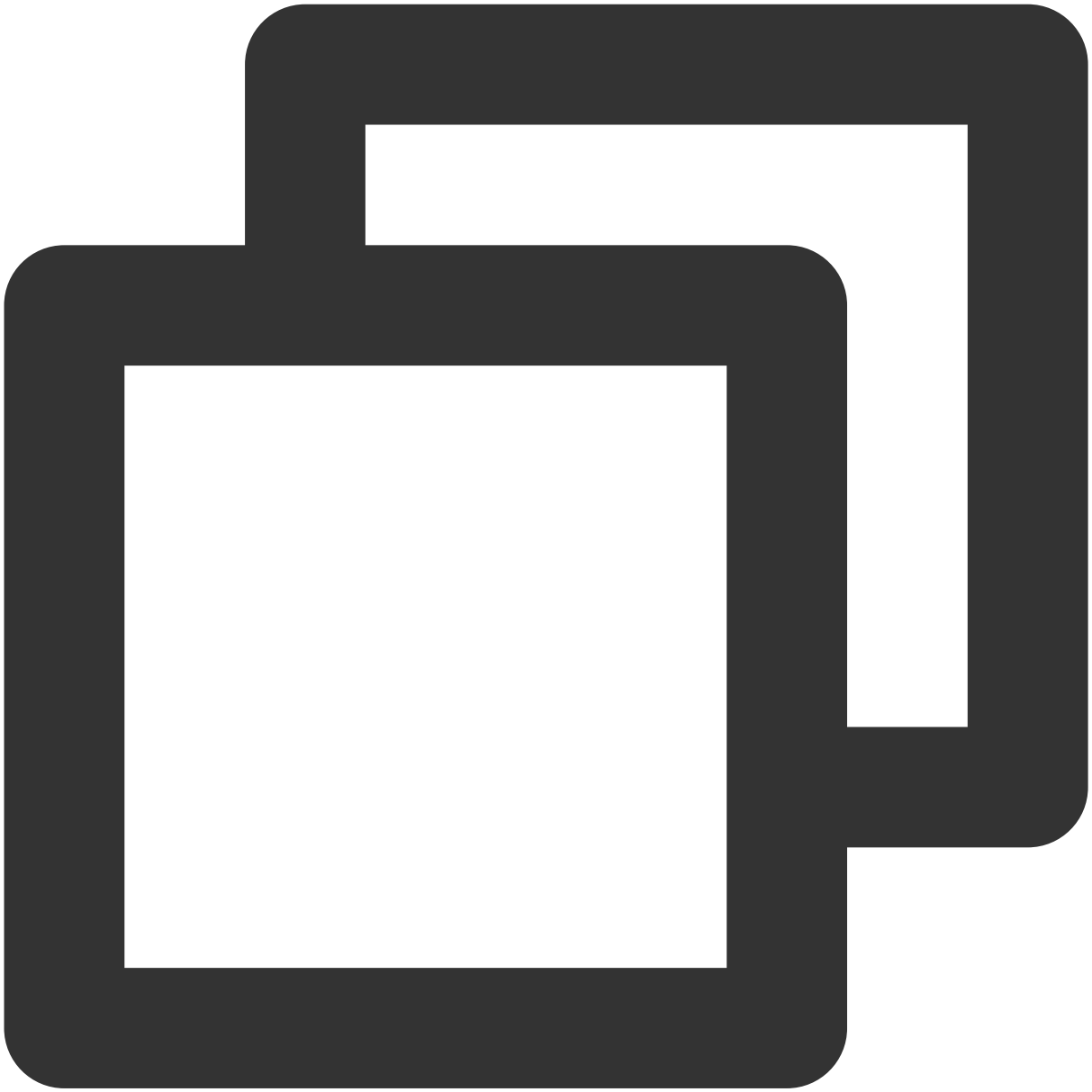
```
EVERY(<col> boolean)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return true if all values of col are true.

Return type: boolean

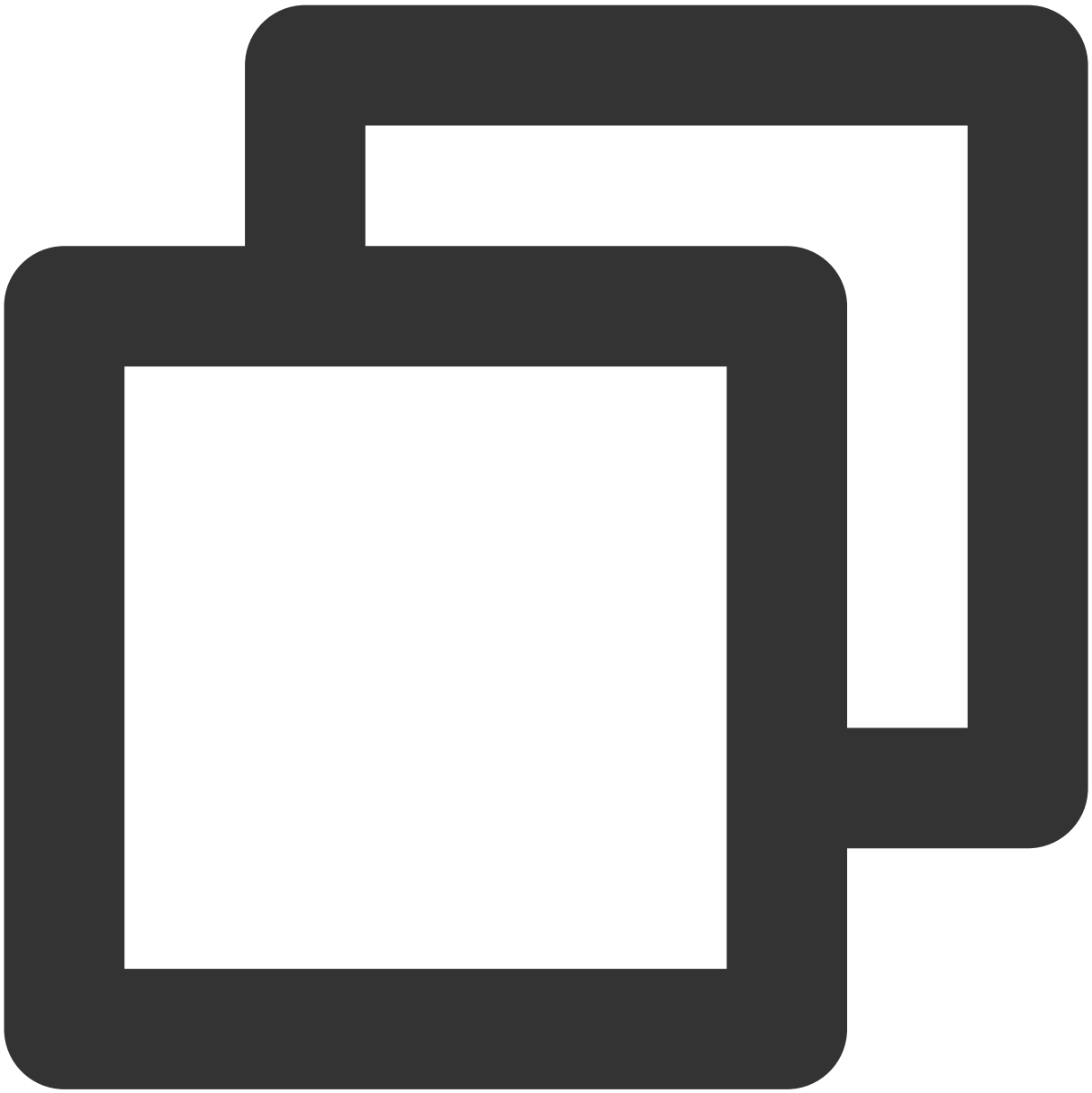
Example:



```
> SELECT every(col) FROM (VALUES (true), (true), (true)) AS tab(col);
 true
> SELECT every(col) FROM (VALUES (NULL), (true), (true)) AS tab(col);
 true
> SELECT every(col) FROM (VALUES (true), (false), (true)) AS tab(col);
 false
```

BOOL_AND

Function statement:



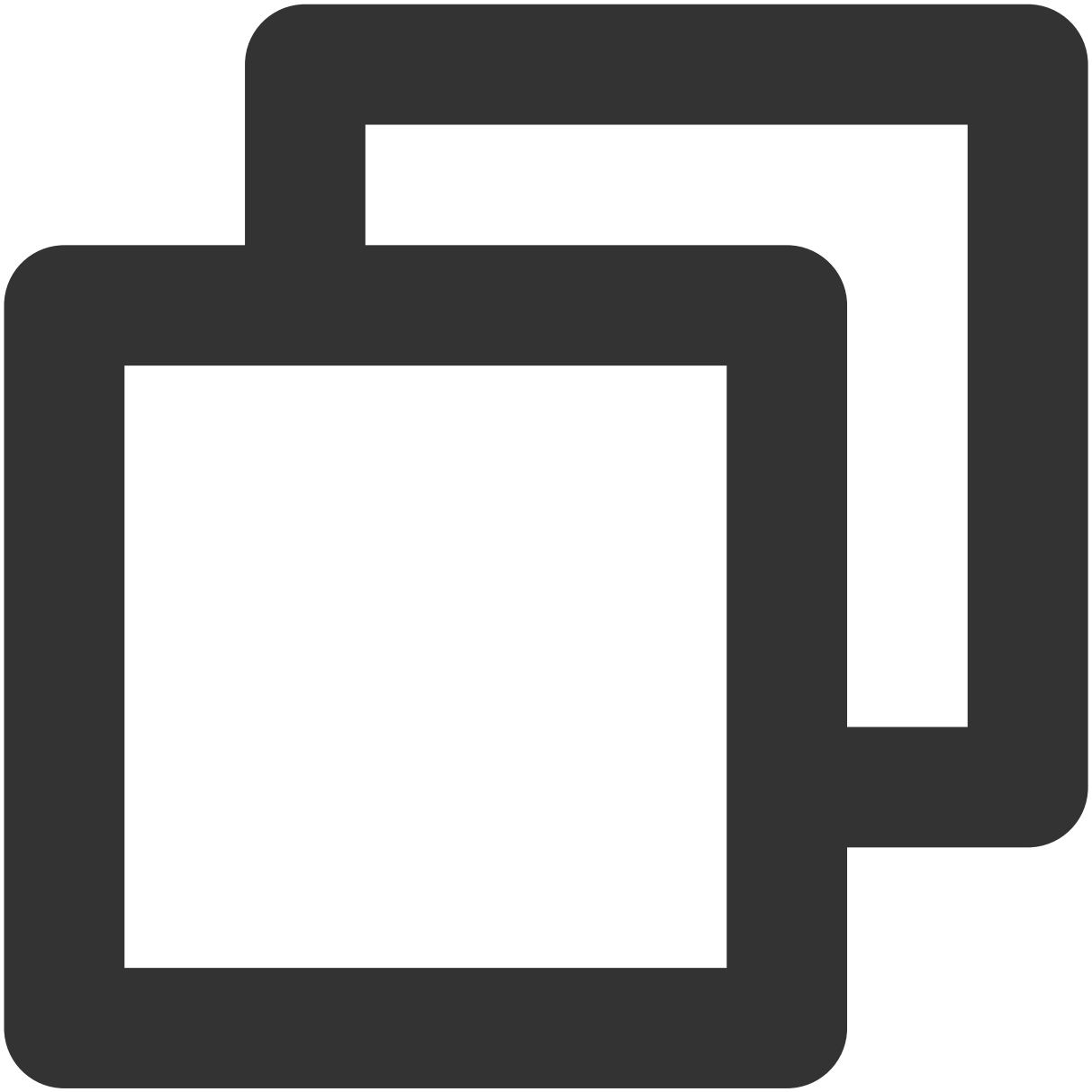
```
BOOL_AND(<col> boolean)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return true if all values of col are true.

Return type: boolean

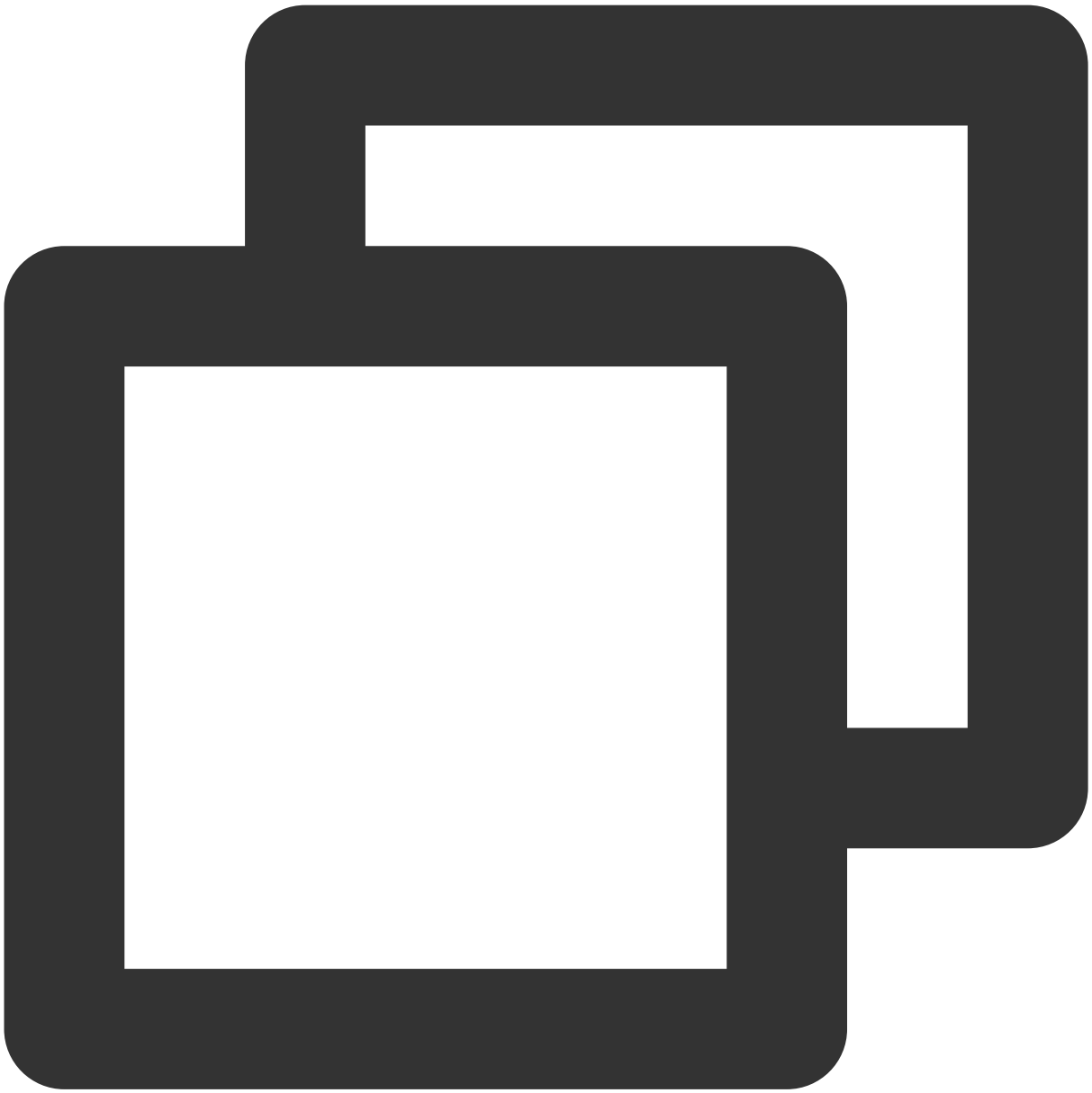
Example:



```
> SELECT bool_and(col) FROM (VALUES (true), (true), (true)) AS tab(col);
 true
> SELECT bool_and(col) FROM (VALUES (NULL), (true), (true)) AS tab(col);
 true
> SELECT bool_and(col) FROM (VALUES (true), (false), (true)) AS tab(col);
 false
```

AND

Function statement:



```
<expr1> AND <expr2>
```

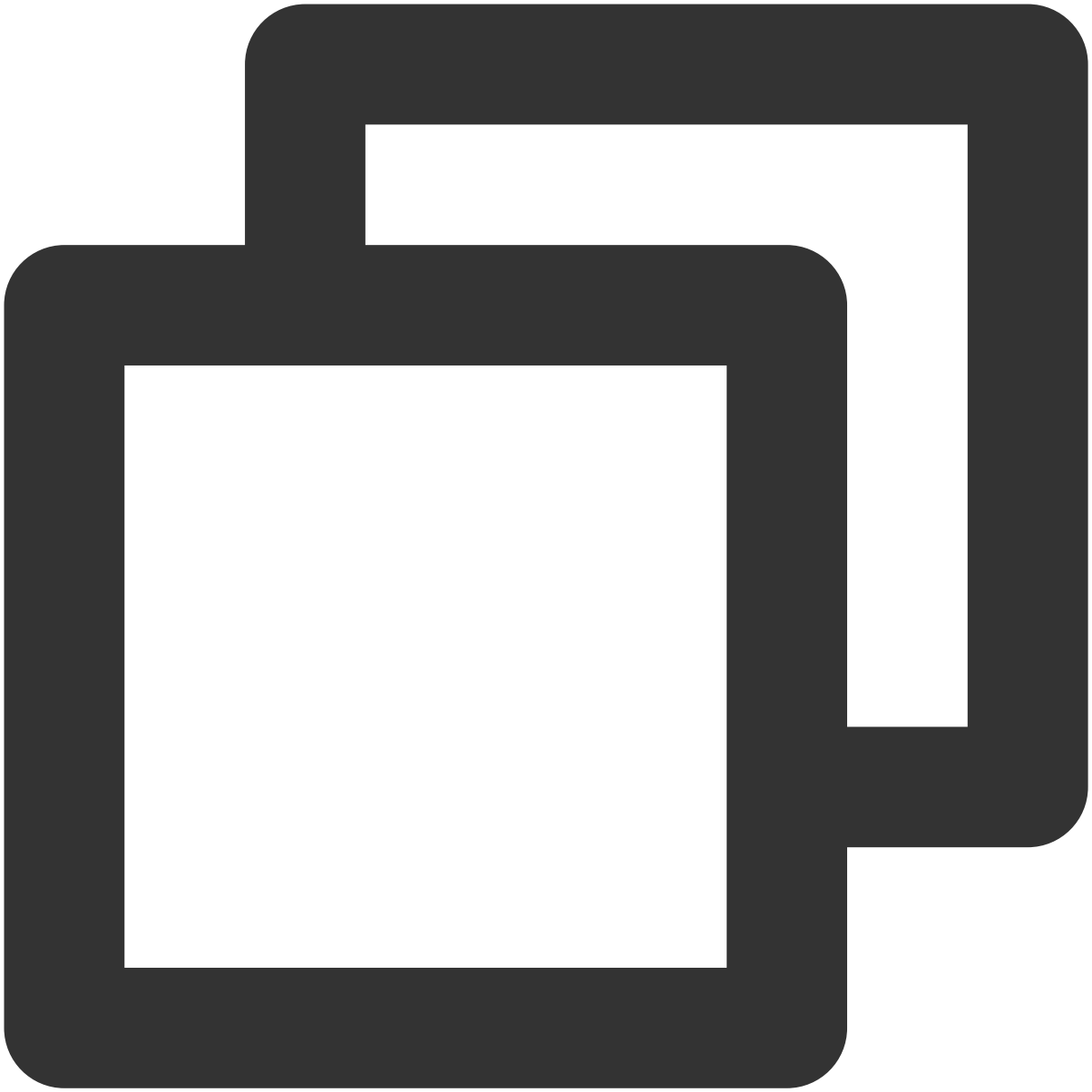
Supported engines: SparkSQL and Presto

Usage Instructions

Logical AND

Return type: boolean

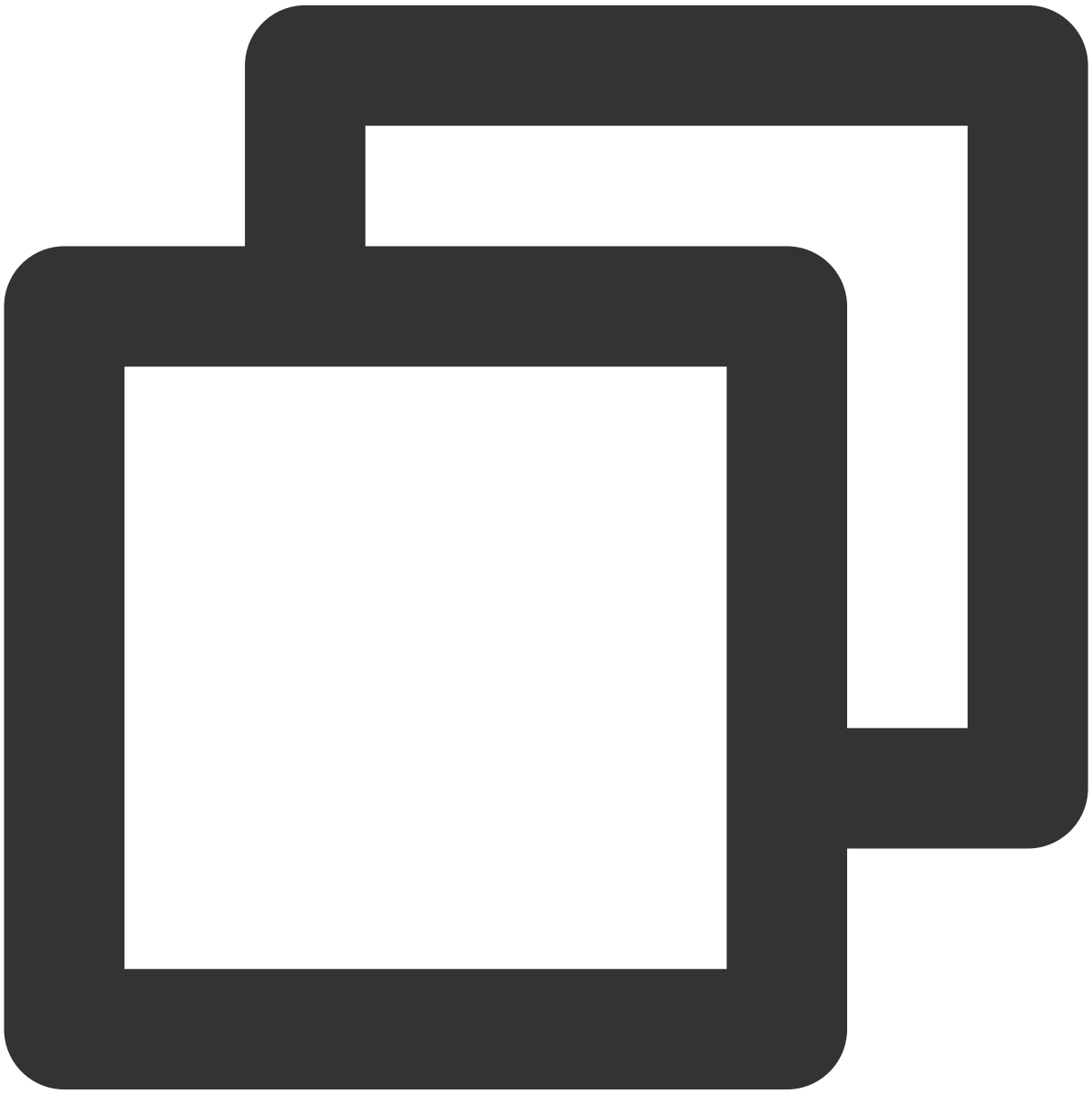
Example:



```
> SELECT true and true;
 true
> SELECT true and false;
 false
> SELECT true and NULL;
 NULL
> SELECT false and NULL;
 false
```

OR

Function statement:



```
<expr1> OR <expr2>
```

Supported engines: SparkSQL and Presto

Usage instructions: Logical OR

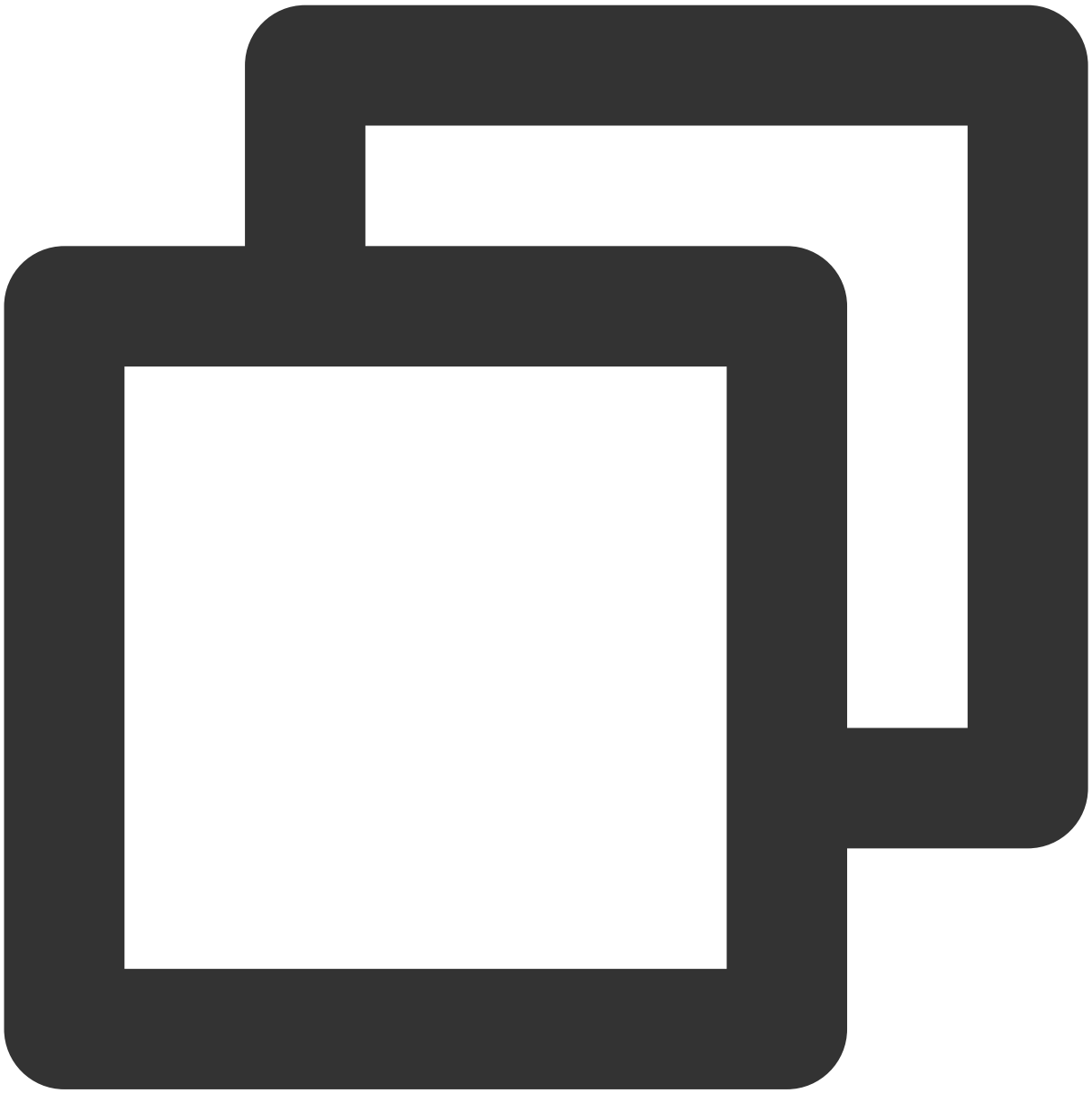
Return type: boolean

Example:

```
> SELECT true or false;
 true
> SELECT false or false;
 false
> SELECT true or NULL;
 true
> SELECT false or NULL;
 NULL
```

ANY

Function statement:



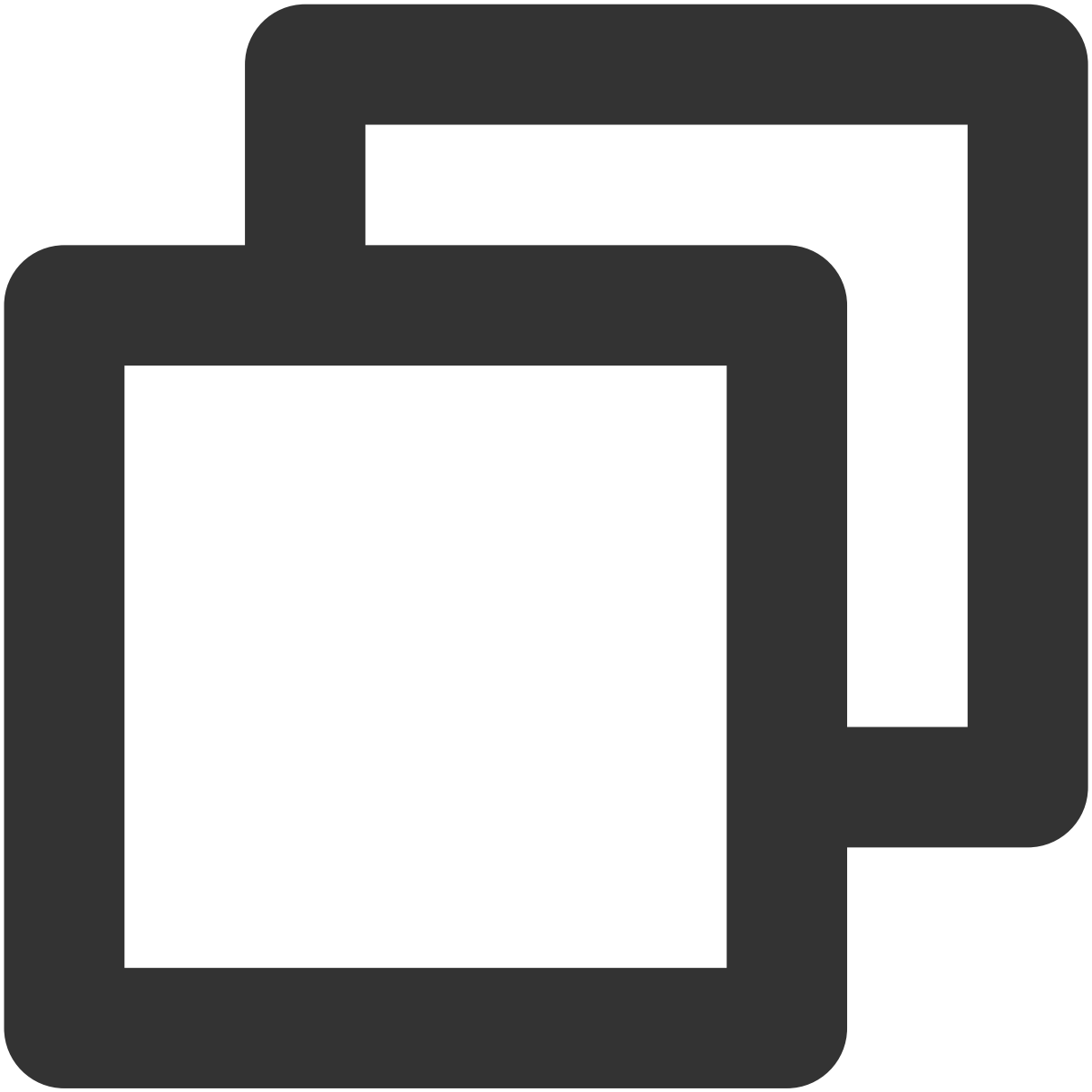
```
ANY(<col> boolean)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return true if at least one value of col is true.

Return type: boolean

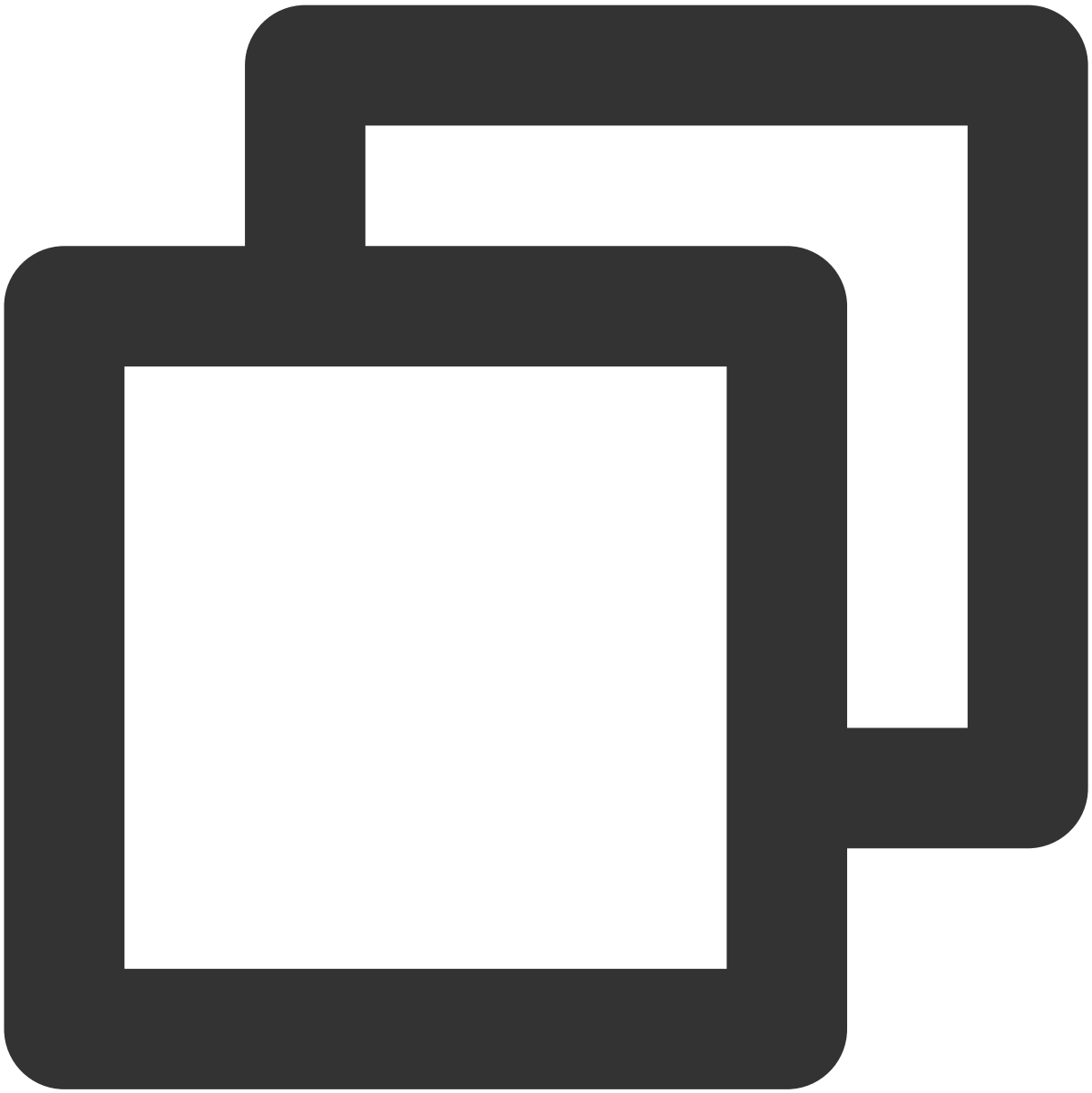
Example:



```
> SELECT any(col) FROM (VALUES (true), (false), (false)) AS tab(col);
 true
> SELECT any(col) FROM (VALUES (NULL), (true), (false)) AS tab(col);
 true
> SELECT any(col) FROM (VALUES (false), (false), (NULL)) AS tab(col);
 false
```

SOME

Function statement:



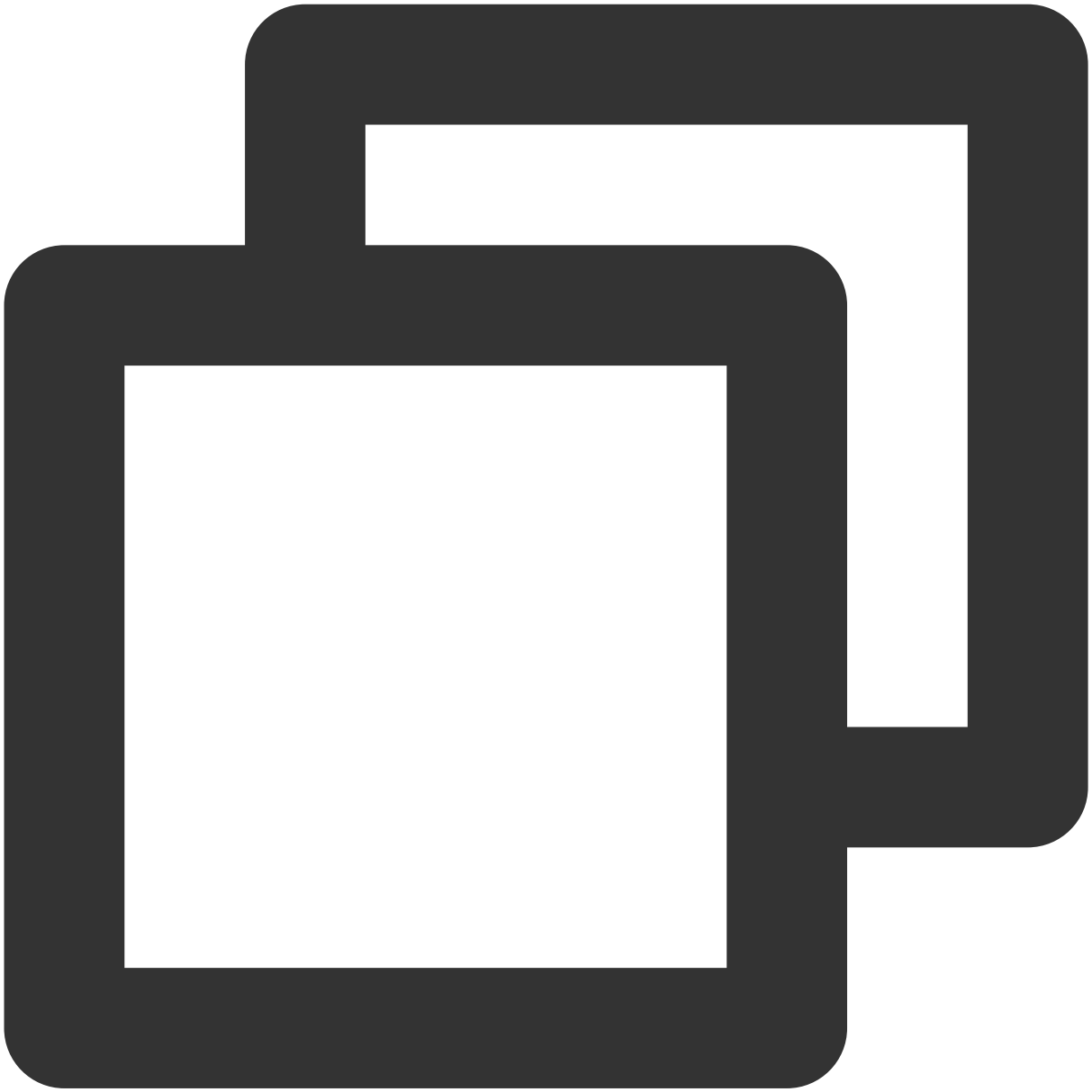
```
SOME(<col> boolean)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return true if at least one value of col is true.

Return type: boolean

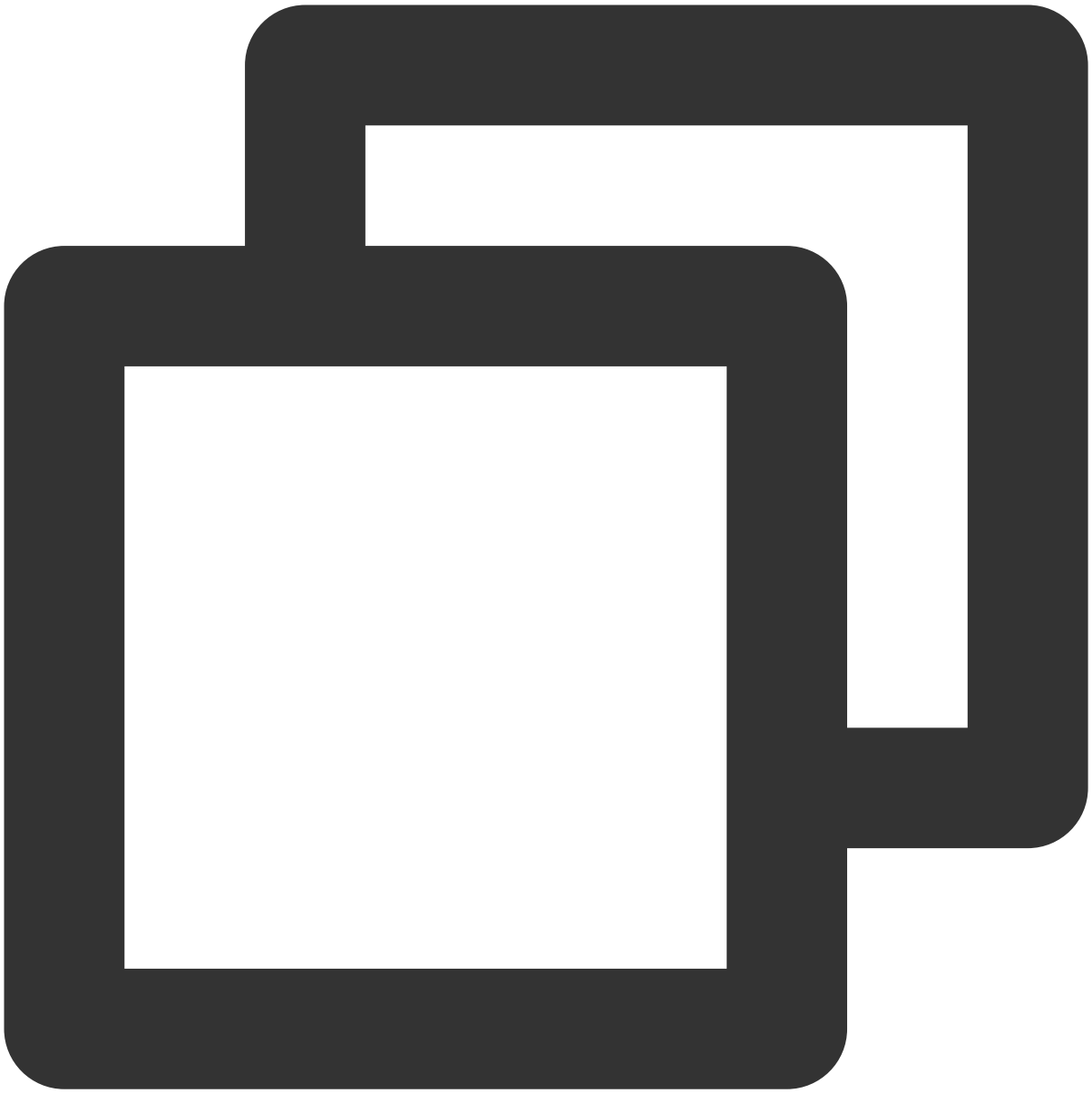
Example:



```
> SELECT some(col) FROM (VALUES (true), (false), (false)) AS tab(col);
 true
> SELECT some(col) FROM (VALUES (NULL), (true), (false)) AS tab(col);
 true
> SELECT some(col) FROM (VALUES (false), (false), (NULL)) AS tab(col);
 false
```

BOOL_OR

Function statement:



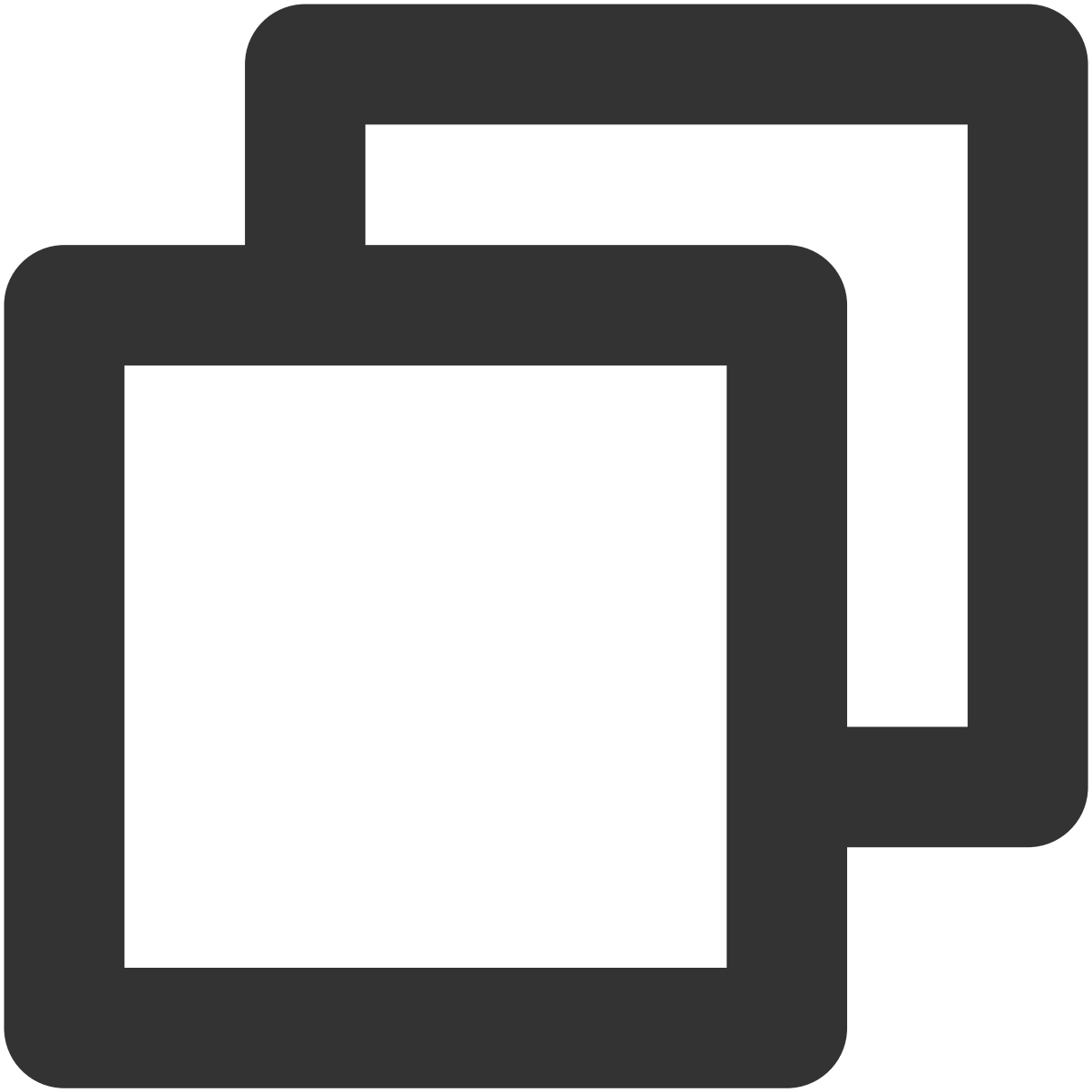
```
BOOL_OR(<col> boolean)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return true if at least one value of col is true.

Return type: boolean

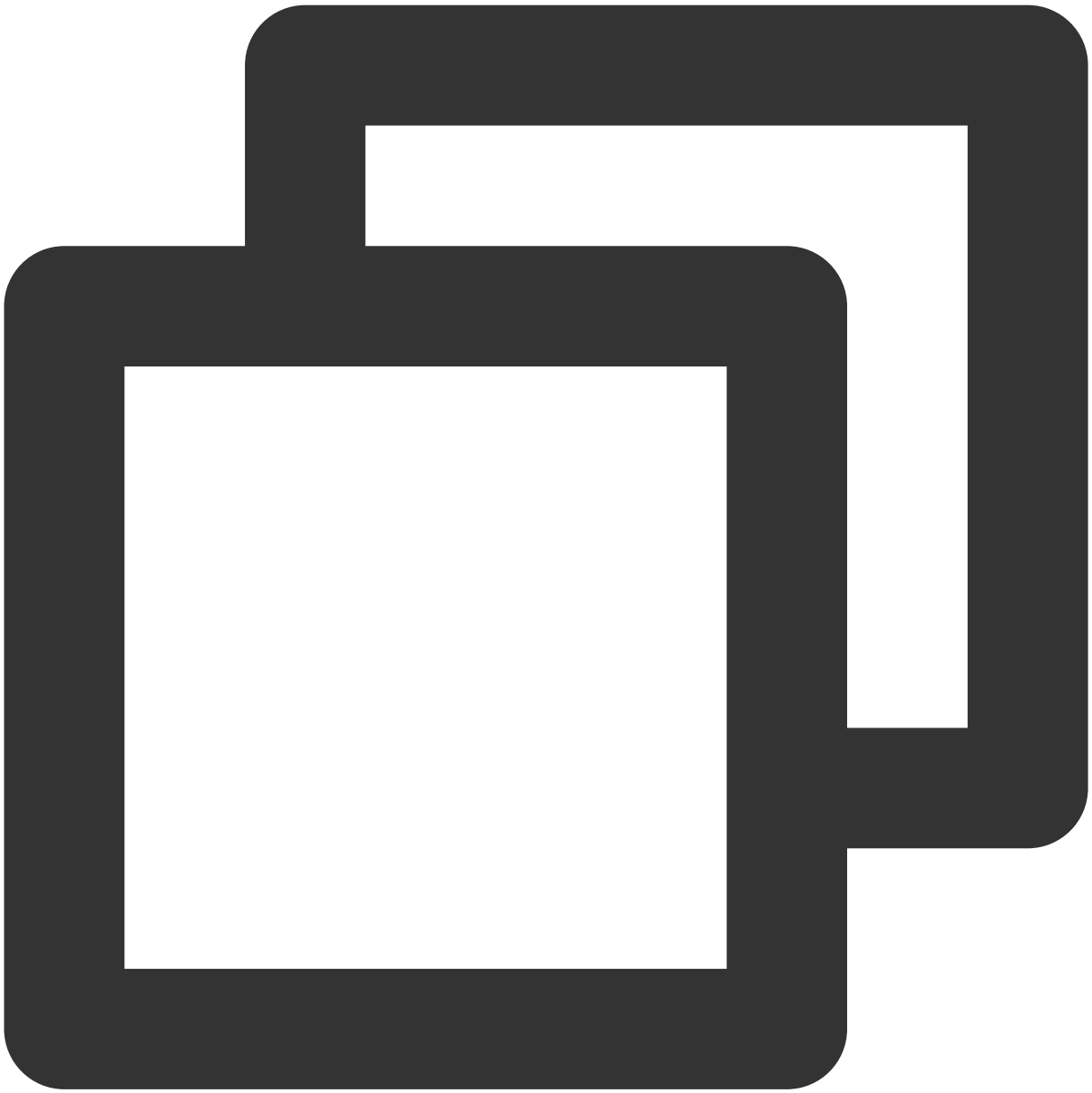
Example:



```
> SELECT BOOL_OR(col) FROM (VALUES (true), (false), (false)) AS tab(col);
 true
> SELECT BOOL_OR(col) FROM (VALUES (NULL), (true), (false)) AS tab(col);
 true
> SELECT BOOL_OR(col) FROM (VALUES (false), (false), (NULL)) AS tab(col);
 false
```

BIT_AND

Function statement:



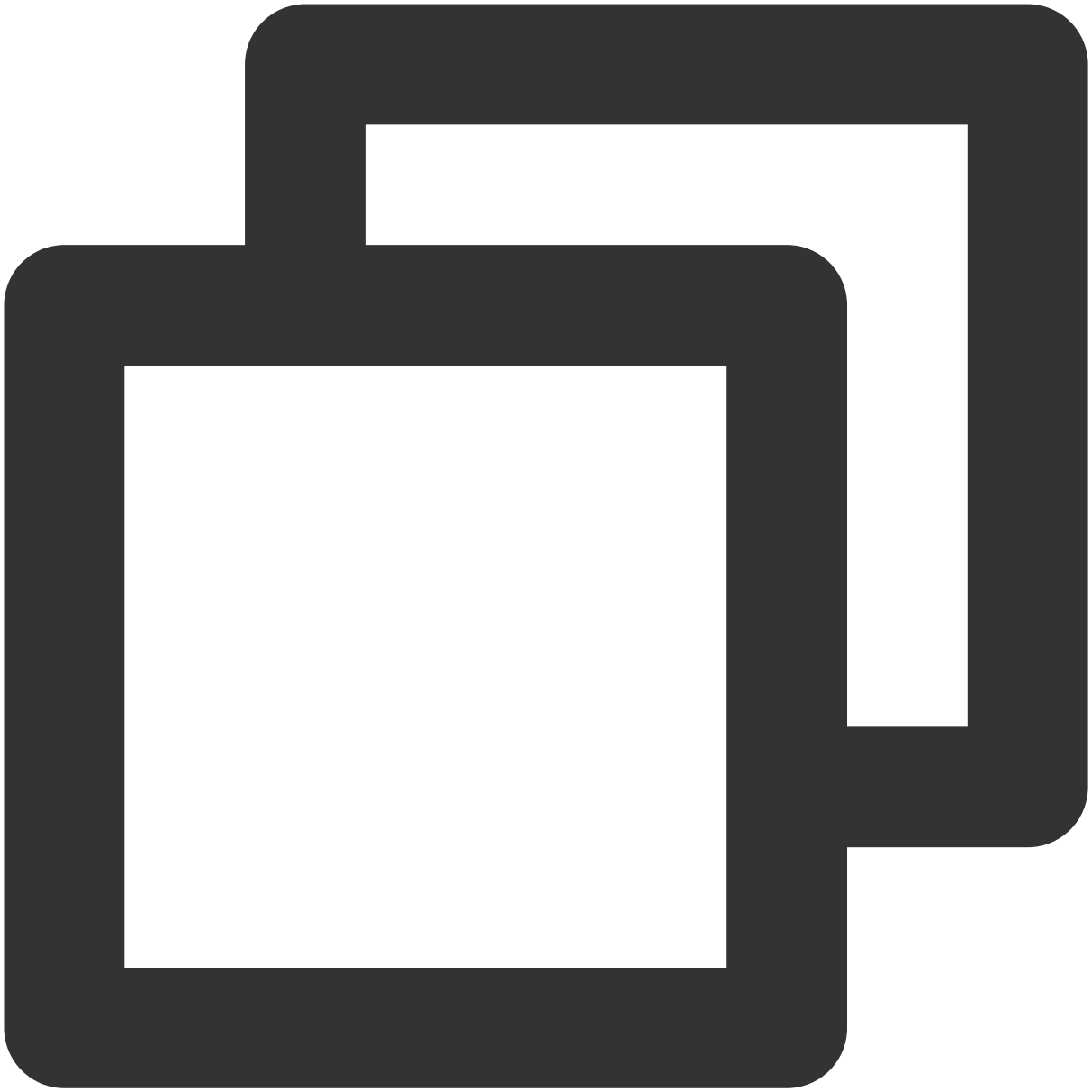
```
BIT_AND(<col> integer|bigint)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the bitwise AND of all non-null input values, or null if there are none.

Return type: same as col

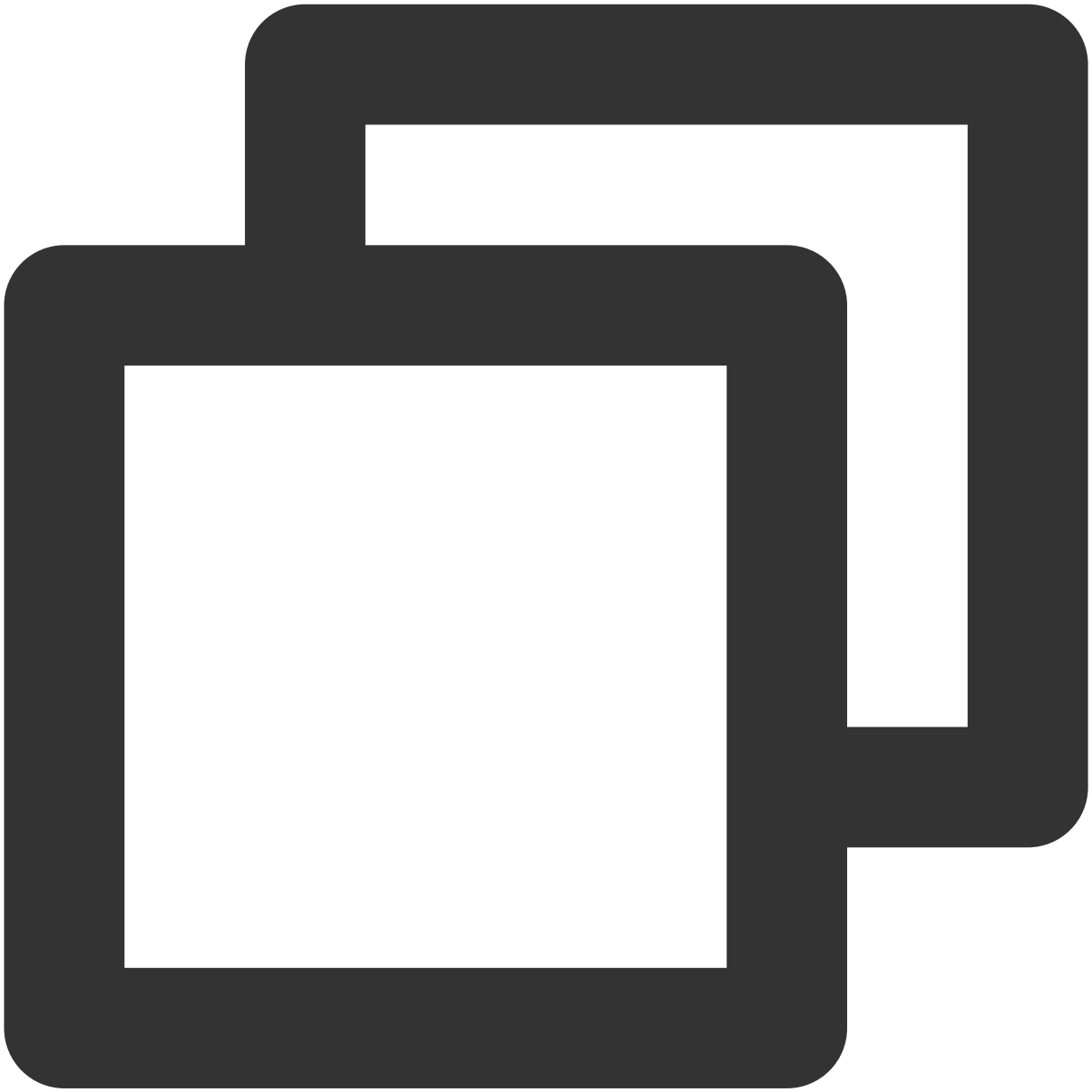
Example:



```
> SELECT bit_and(col) FROM (VALUES (3), (5)) AS tab(col);
 1
```

BIT_OR

Function statement:



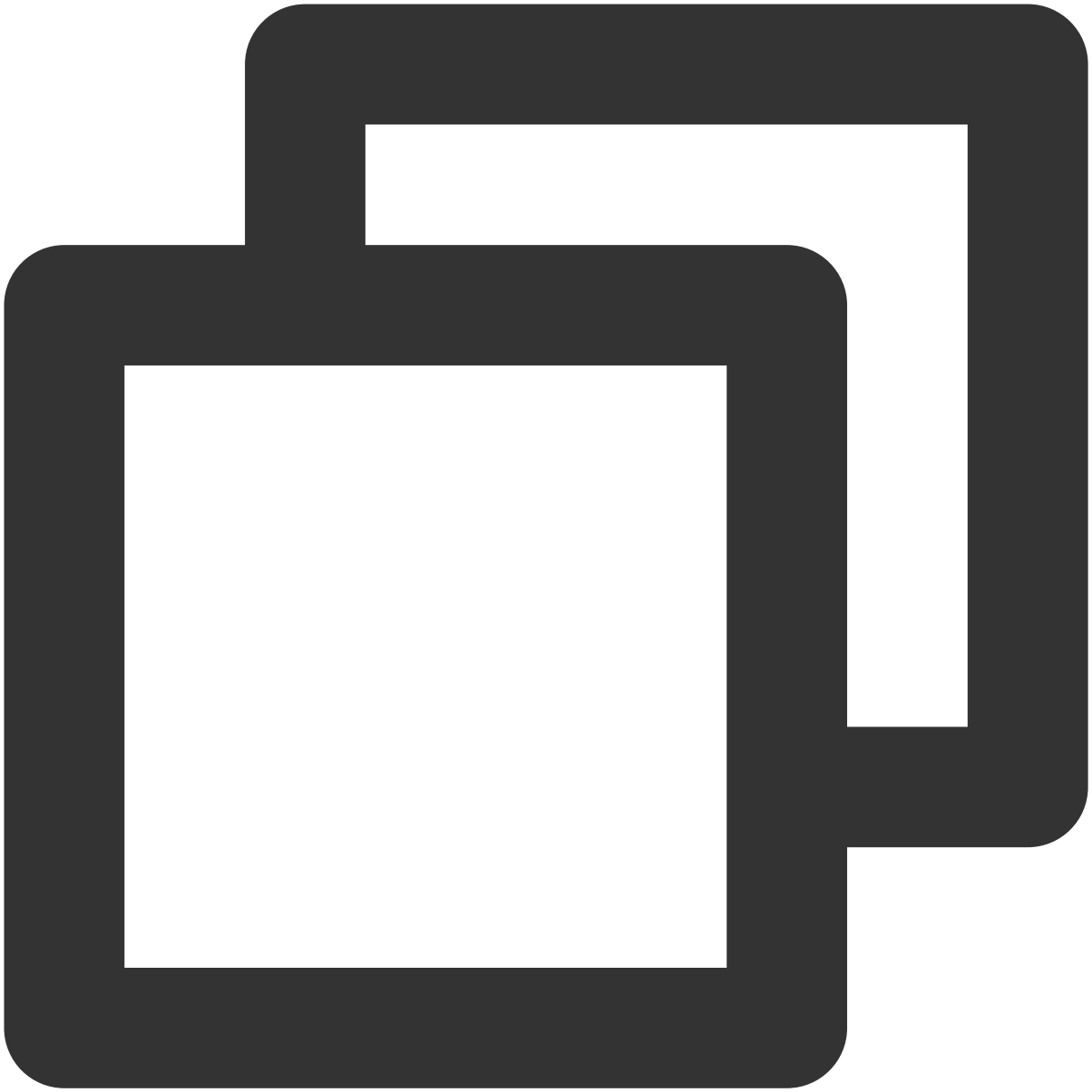
```
BIT_OR(<col> integer|bigint)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the bitwise OR of all non-null input values, or null if there are none.

Return type: same as col

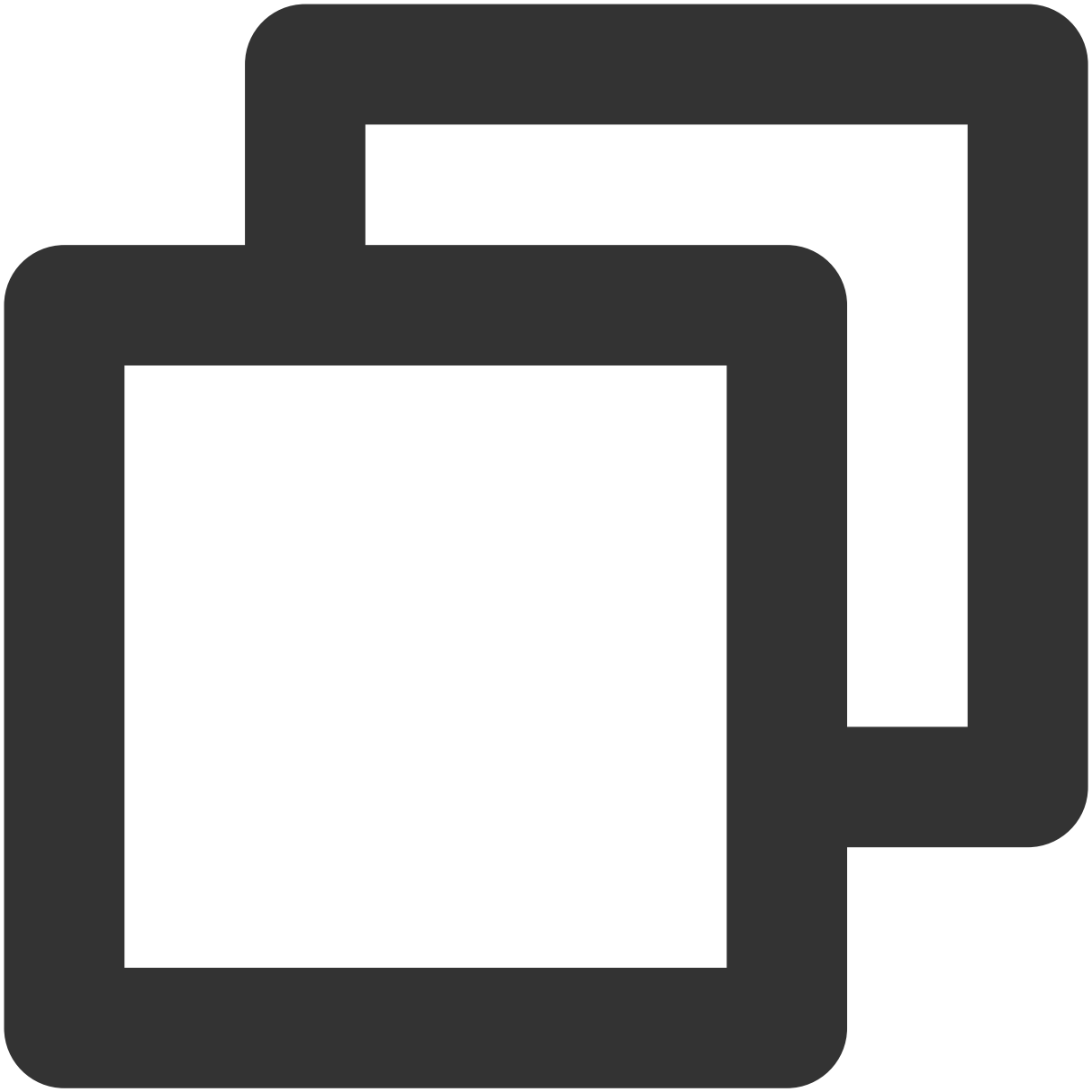
Example:



```
> SELECT bit_or(col) FROM (VALUES (3), (5)) AS tab(col);
 7
```

BIT_XOR

Function statement:



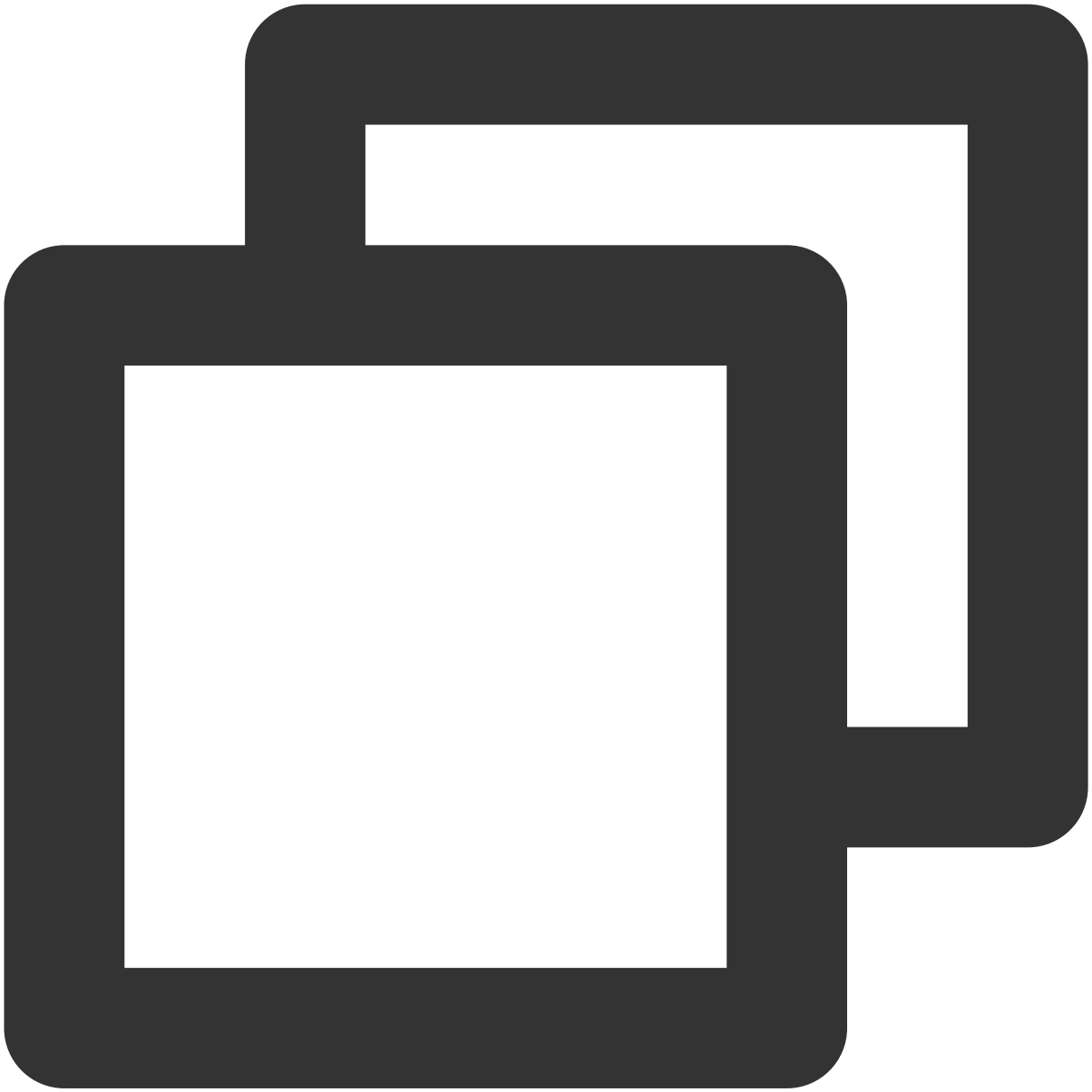
```
BIT_XOR(<col> integer|bigint)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the bitwise XOR of all non-null input values, or null if there are none.

Return type: same as col

Example:



```
> SELECT bit_xor(col) FROM (VALUES (3), (5)) AS tab(col);
 6
```

ARG_MIN

Function statement:

```
ARG_MIN(<col1>, <col2> | expr(col2))
```

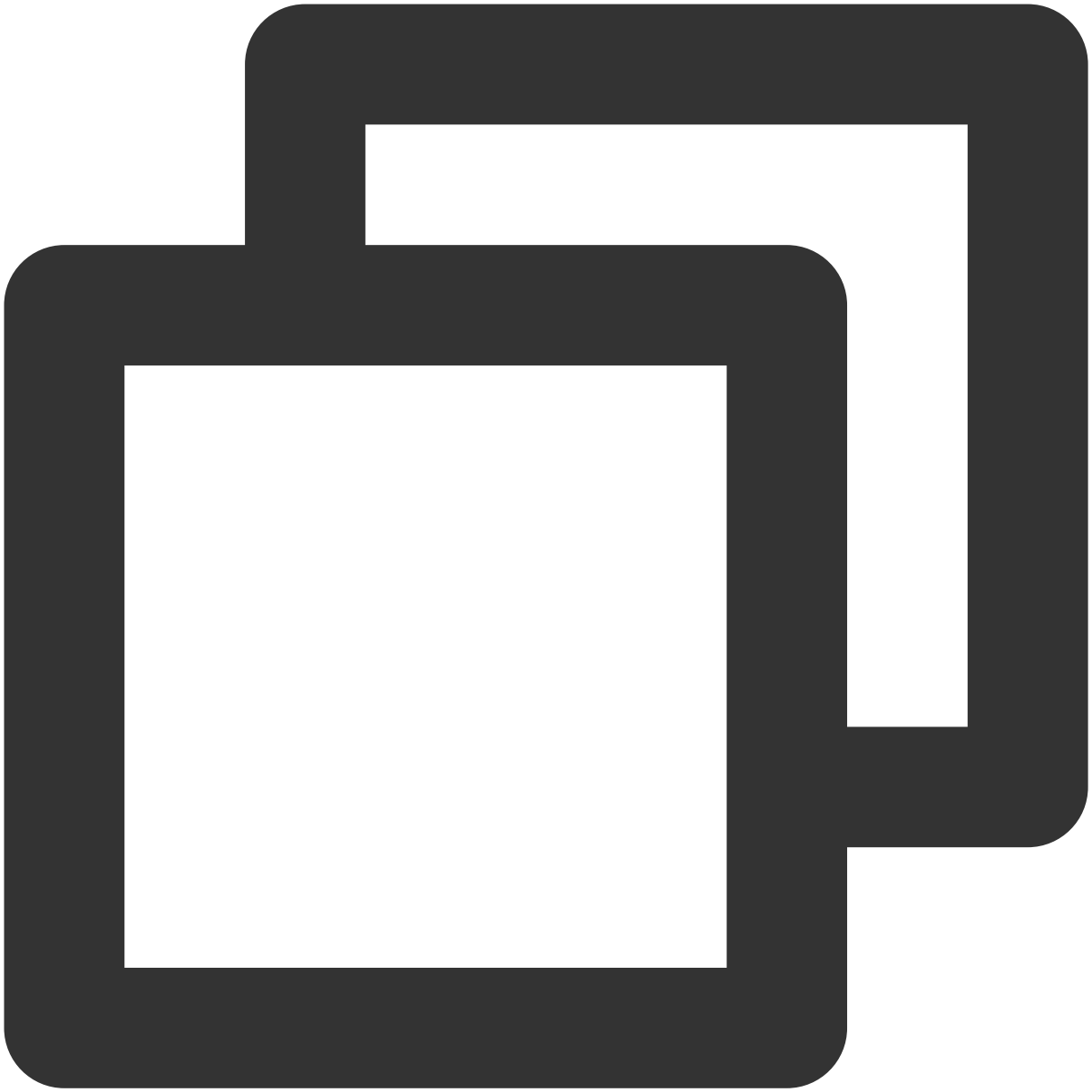
Supported engine: SparkSQ

Usage instructions: Return the row with the minimum value of the specified column col1 based on the specified calculation expression.

col1: For sortable column names, if a constant is provided, any value may be returned. If there are multiple minimum values, one of them is returned randomly.

Return type: same as col2 or expr(col2) type

Example:



```
> SELECT arg_min(dt, uid) from values (1, 'm1'), (2, 't2'), (3, 'z3') as tab(dt, ui
m1
> SELECT arg_min(dt, upper(uid)) from values (1, 'm1', 1), (2, 't2', 1), (3, 'z3',
M1
Z3
```

ARG_MAX

Function statement:

```
ARG_MAX(<col1>, <col2> | expr(col2))
```

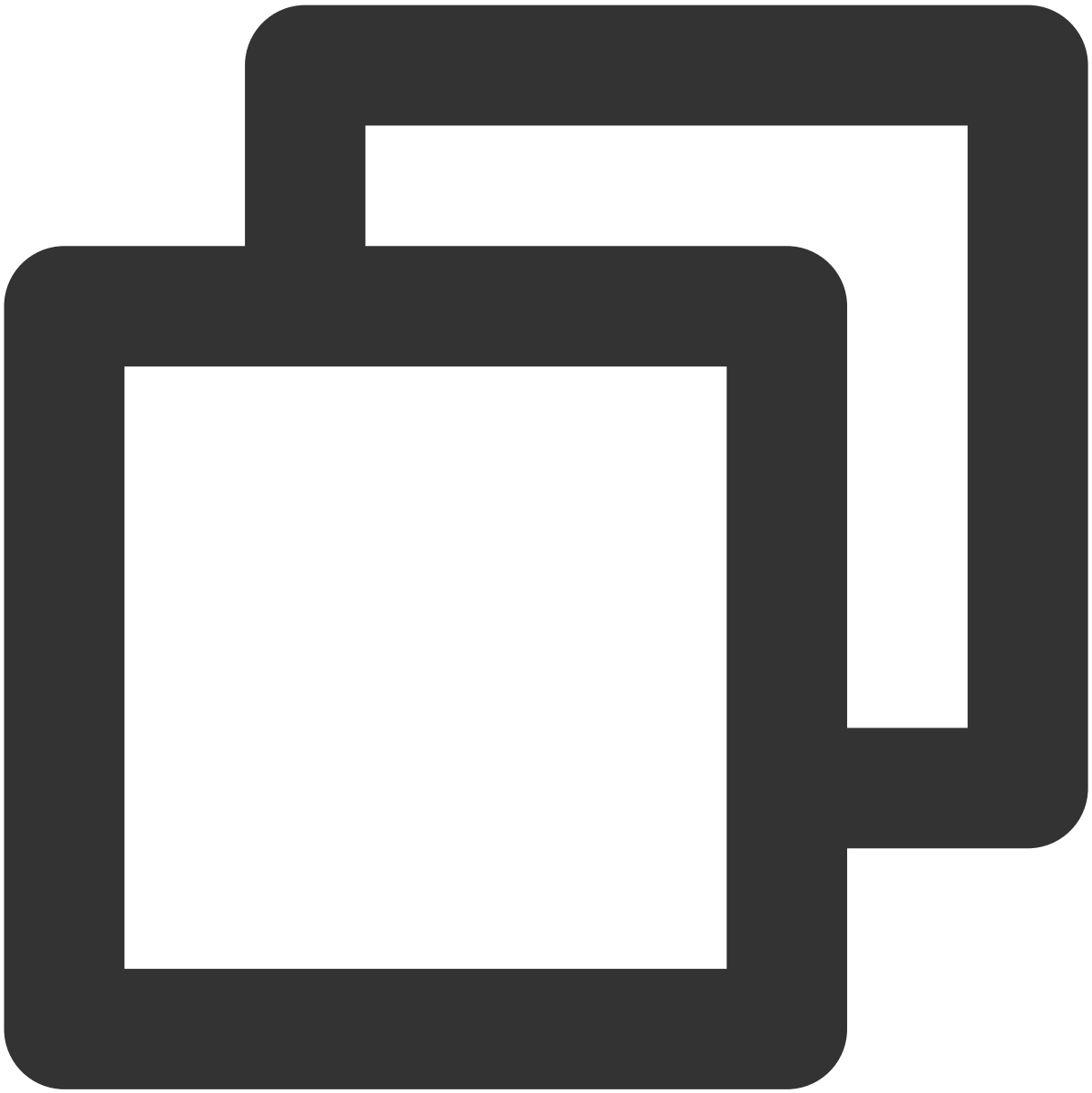
Supported engine: SparkSQL

Usage instructions: Return the row with the maximum value of the specified column col1 based on the specified calculation expression.

col1: For sortable column names, if a constant is provided, any value may be returned. If there are multiple maximum values, one of them is returned randomly.

Return type: same as col2 or expr(col2) type

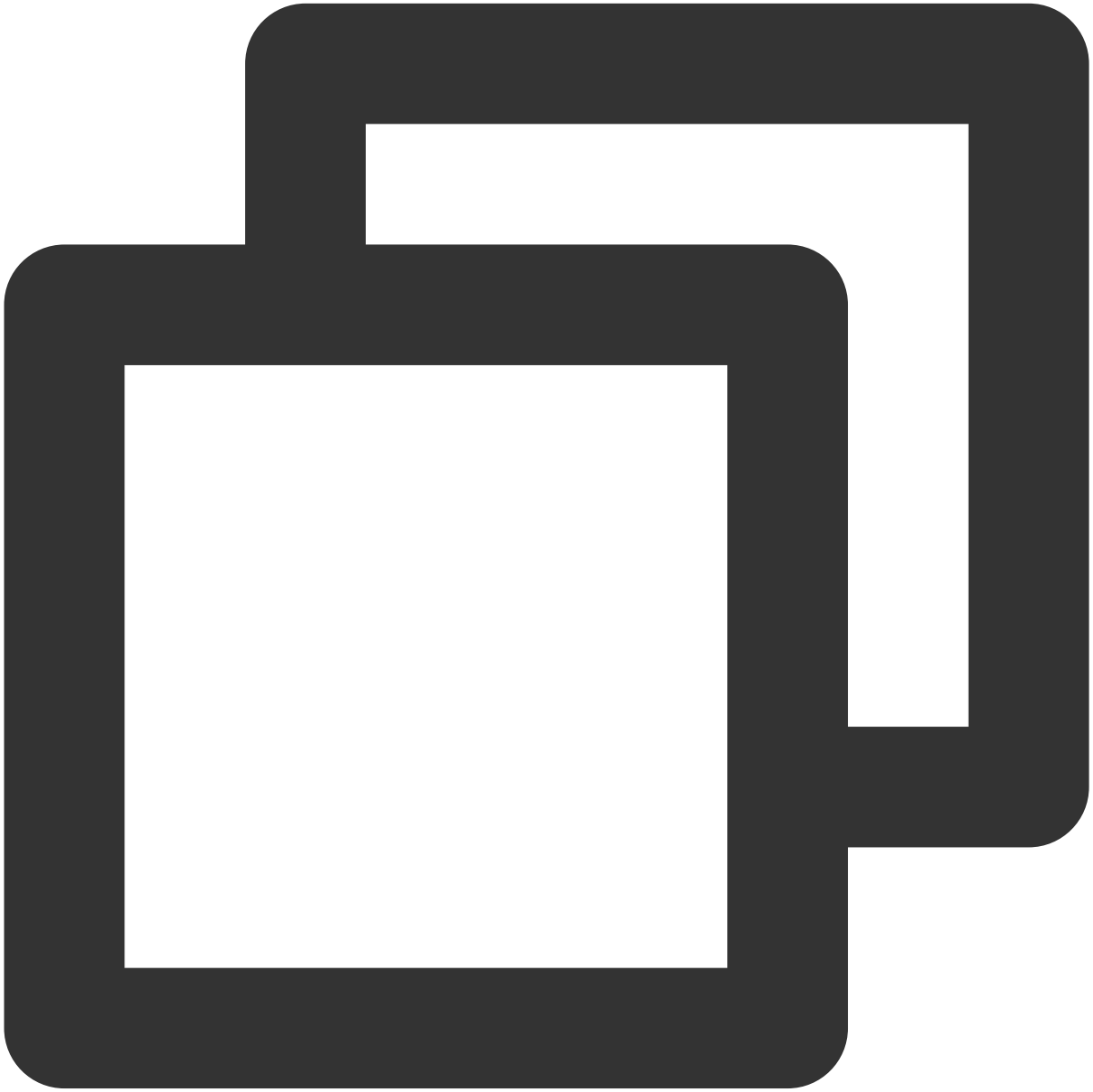
Example:



```
> SELECT arg_max(dt, uid) from values (1, 'm1'), (2, 't2'), (3, 'z3') as tab(dt, ui
z3
> SELECT arg_max(dt, upper(uid)) from values (1, 'm1', 1), (2, 't2', 1), (3, 'z3',
T2
Z3
```

MAP_UNION_SUM

Function statement:



```
MAP_UNION_SUM(map<k, v> input)
```

Supported engine: SparkSQL

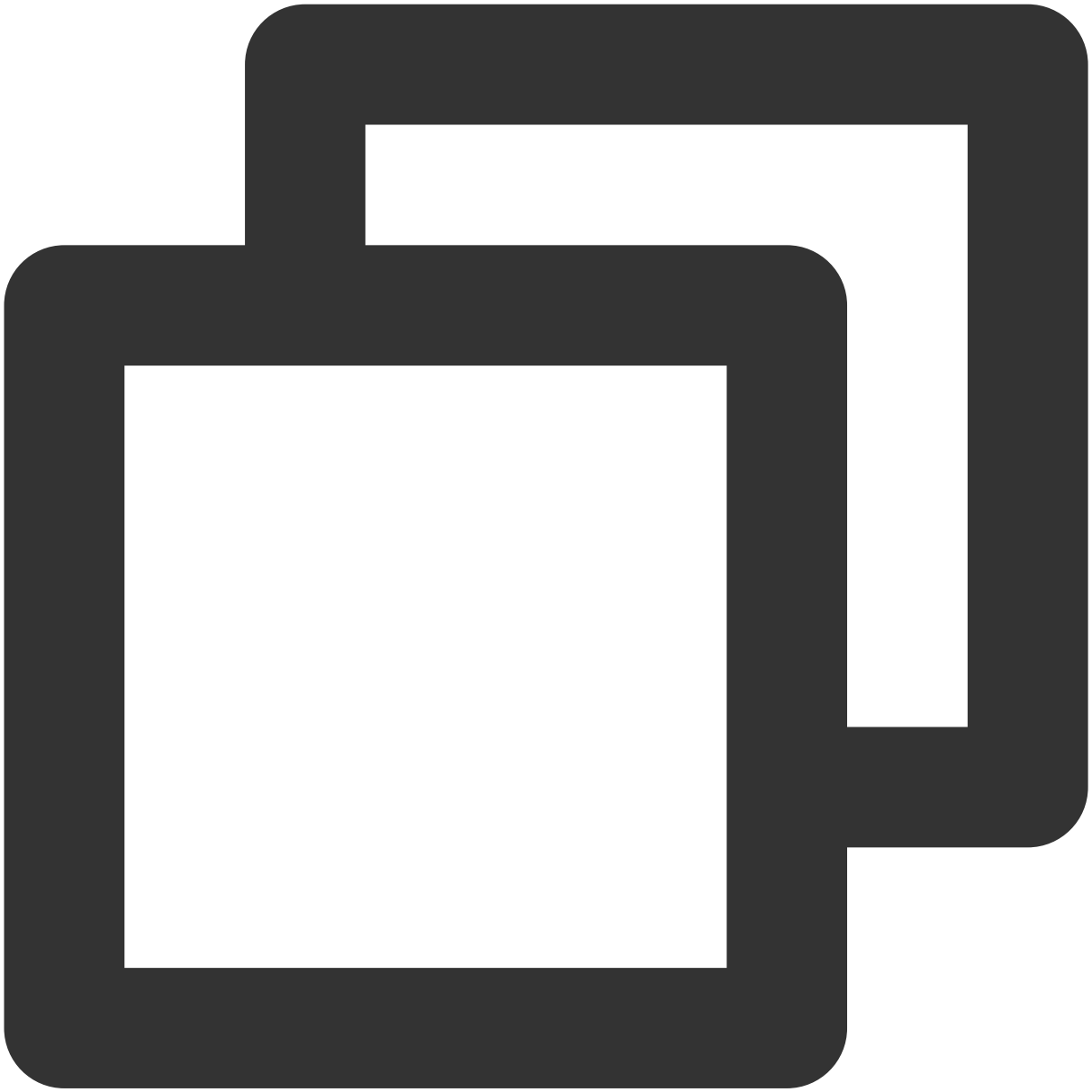
Usage instructions: Specify the column and sum all the values in the map by key.

Note 1: Aggregated data is allowed to be null. If all are null, the output is null.

Note 2: The value of the key is allowed to be null, in which case it will be converted to the number 0.

Return type: same as input type

Example:



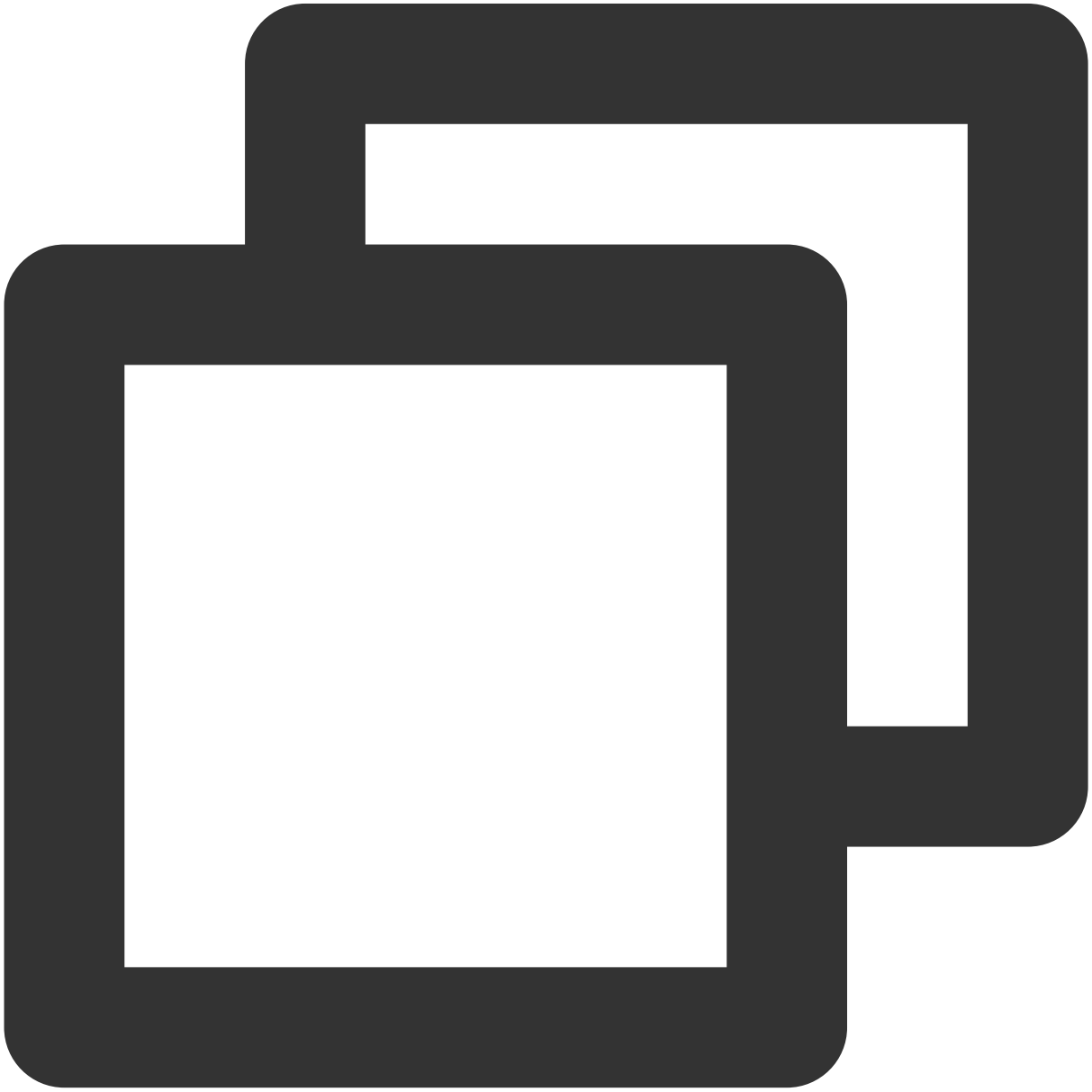
```
> SELECT map_union_sum(ids) from values (map(1, 1)), (map(1,2)), (map(2,1)) as tab(
{1:3,2:1}
> SELECT idx, map_union_sum(ids) from values (map(1, 1), 1), (map(1,2), 1), (map(2,
1 {1:3,2:1}
```

Window Functions

Last updated：2024-08-07 17:33:25

row_number

Function statement:



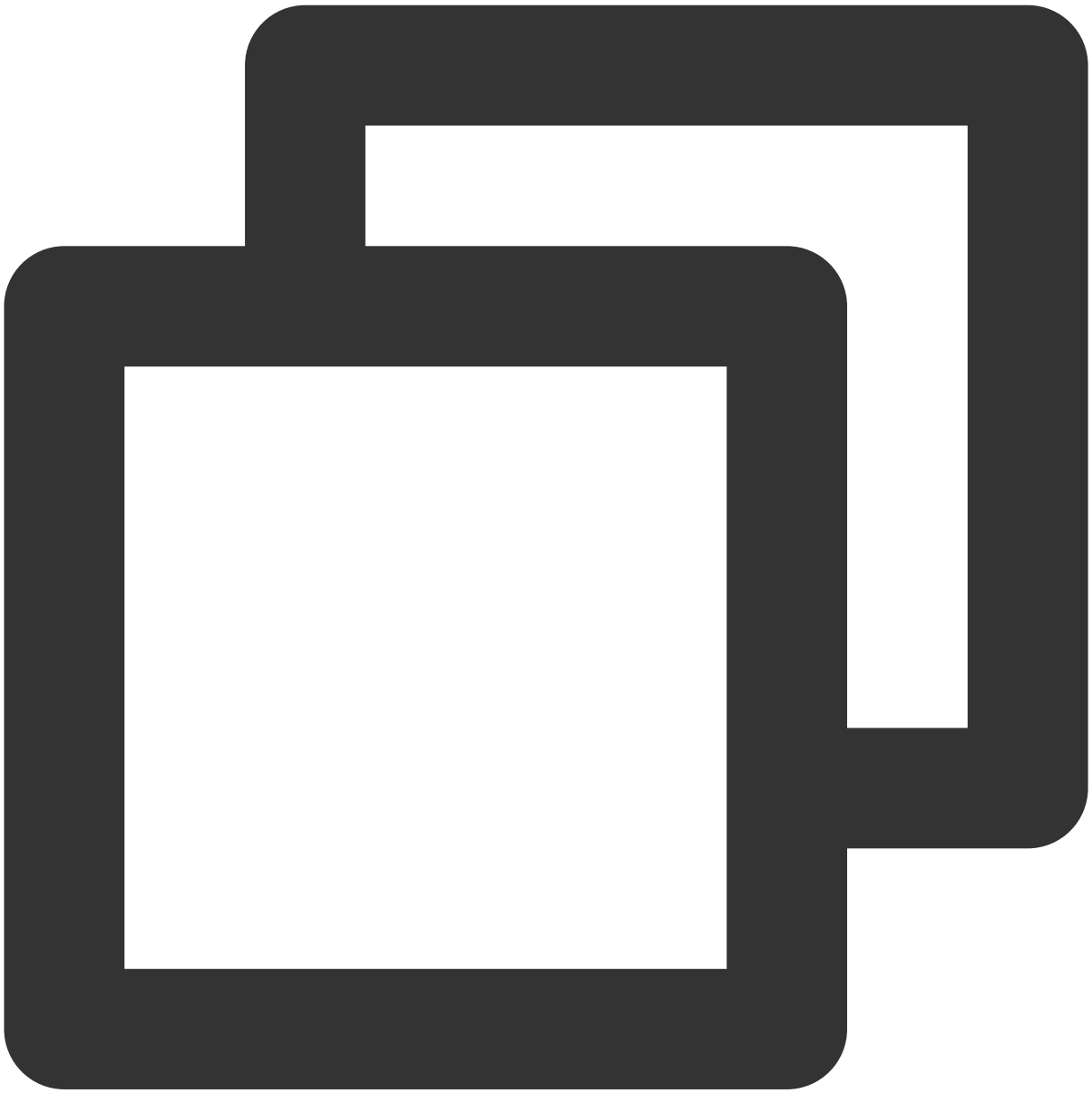
```
row_number()
```

Supported engines: SparkSQL and Presto

Usage instructions: Assign a unique consecutive number to each row.

Return type: int

Example:

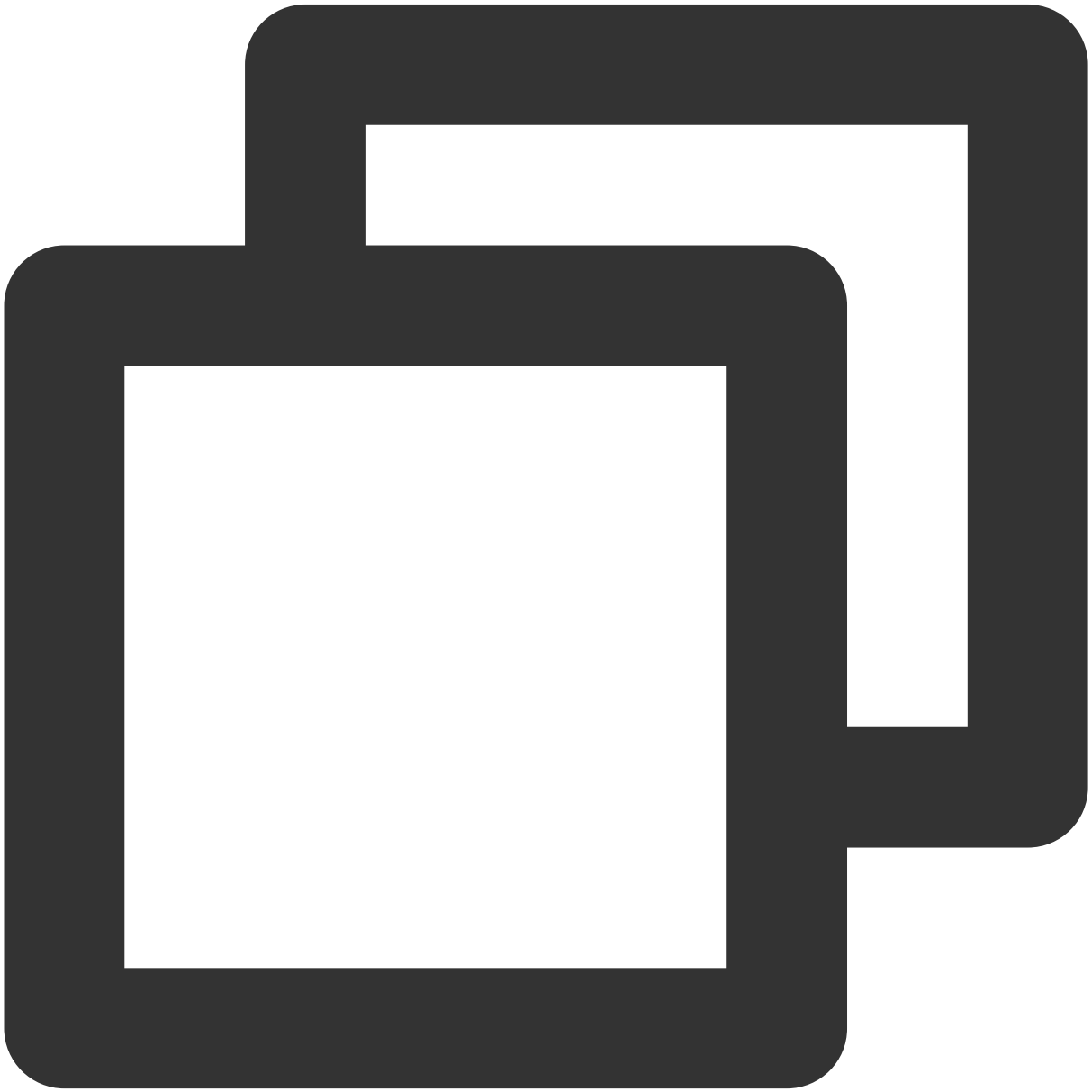


```
SELECT a, b, row_number() OVER (PARTITION BY a ORDER BY b) FROM VALUES ('A1', 2), (
A1      1       1
A1      1       2
A1      2       3
```

```
A2      3       1
```

rank

Function statement:



```
rank()
```

Supported engines: SparkSQL and Presto

Usage instructions: Compute the rank of a value within a group of values. If there are ties, leaves gaps in the ranking sequence.

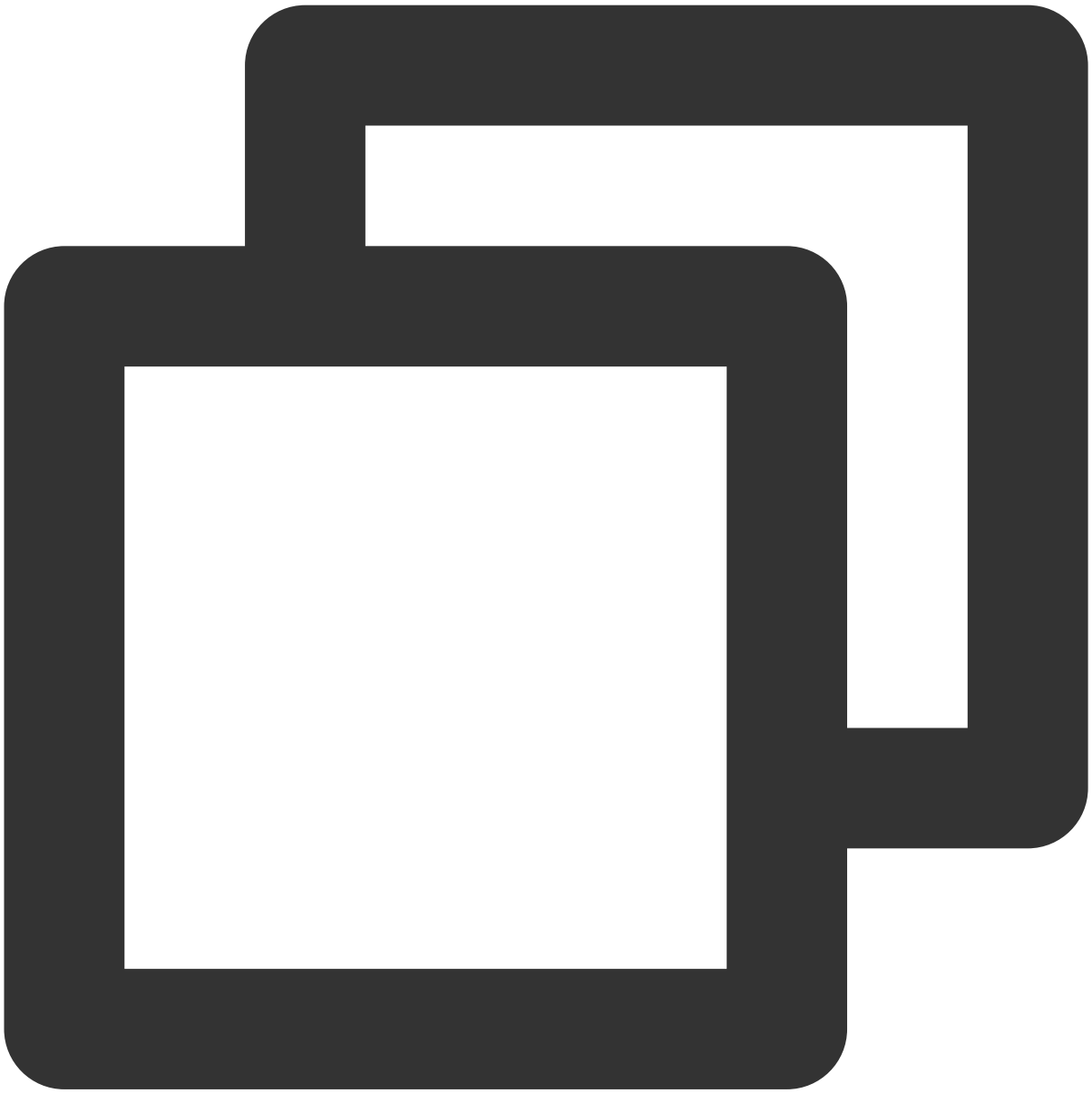
Return type: int

Example:

```
SELECT a, b, rank() OVER (PARTITION BY a ORDER BY b) FROM VALUES ('A1', 2), ('A1',
A1      1       1
A1      1       1
A1      2       3
A2      3       1
```

dense_rank

Function statement:



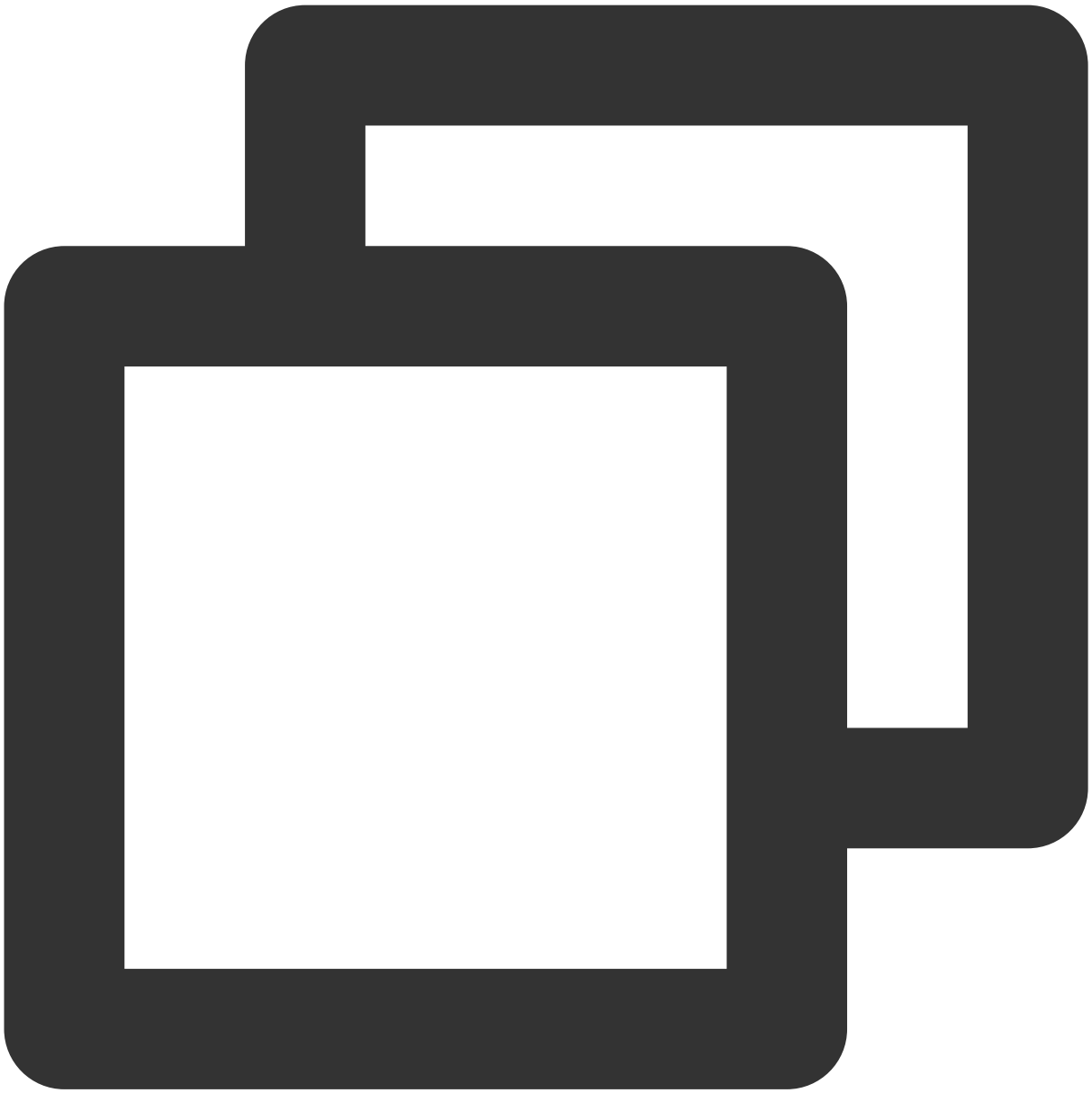
```
dense_rank()
```

Supported engines: SparkSQL and Presto

Usage instructions: Compute the rank of a value within a group of values. Unlike the rank function, dense_rank does not leave gaps in the ranking sequence in the case of ties.

Return type: int

Example:



```
SELECT a, b, dense_rank() OVER (PARTITION BY a ORDER BY b) FROM VALUES ('A1', 2), (
A1      1       1
A1      1       1
A1      2       2
A2      3       1
```

percent_rank()

Function statement:

```
percent_rank()
```

Supported engines: SparkSQL and Presto

Usage instructions: Compute the percentile rank of a value within a group of values. The return value is a decimal between 0 and 1.

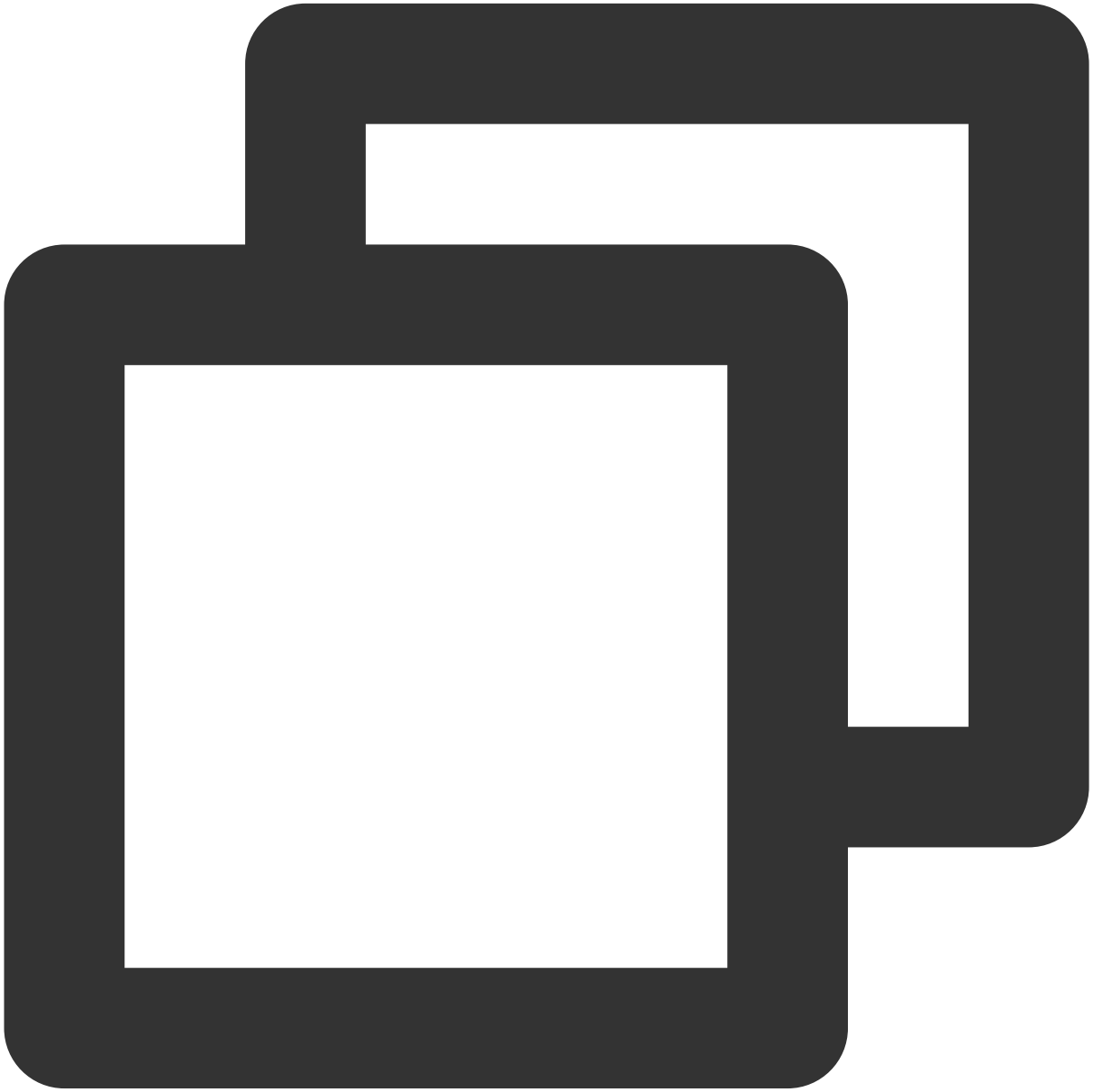
Return type: double

Example:

```
SELECT a, b, percent_rank() OVER (PARTITION BY a ORDER BY b) FROM VALUES ('A1', 2),
A1      1       0.0
A1      1       0.0
A1      2       1.0
A2      3       0.0
```

cume_dist

Function statement:



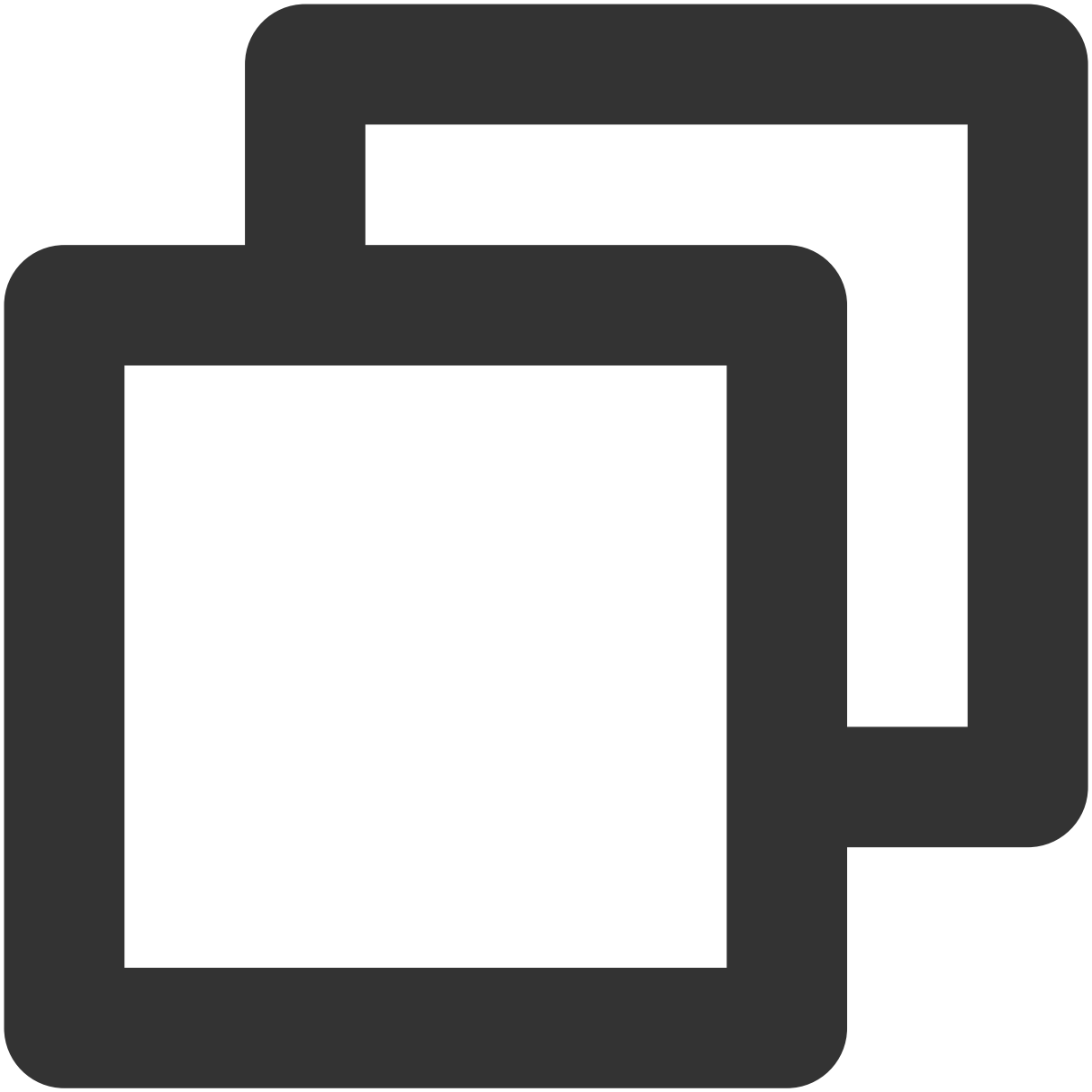
```
cume_dist()
```

Supported engines: SparkSQL and Presto

Usage instructions: Compute the position of a value relative to all values within a partition.

Return type: double

Example:



```
SELECT a, b, cume_dist() OVER (PARTITION BY a ORDER BY b)  FROM VALUES ('A1', 2), (
A1      1       0.6666666666666666
A1      1       0.6666666666666666
A1      2       1.0
A2      3       1.0
```

first_value

Function statement:

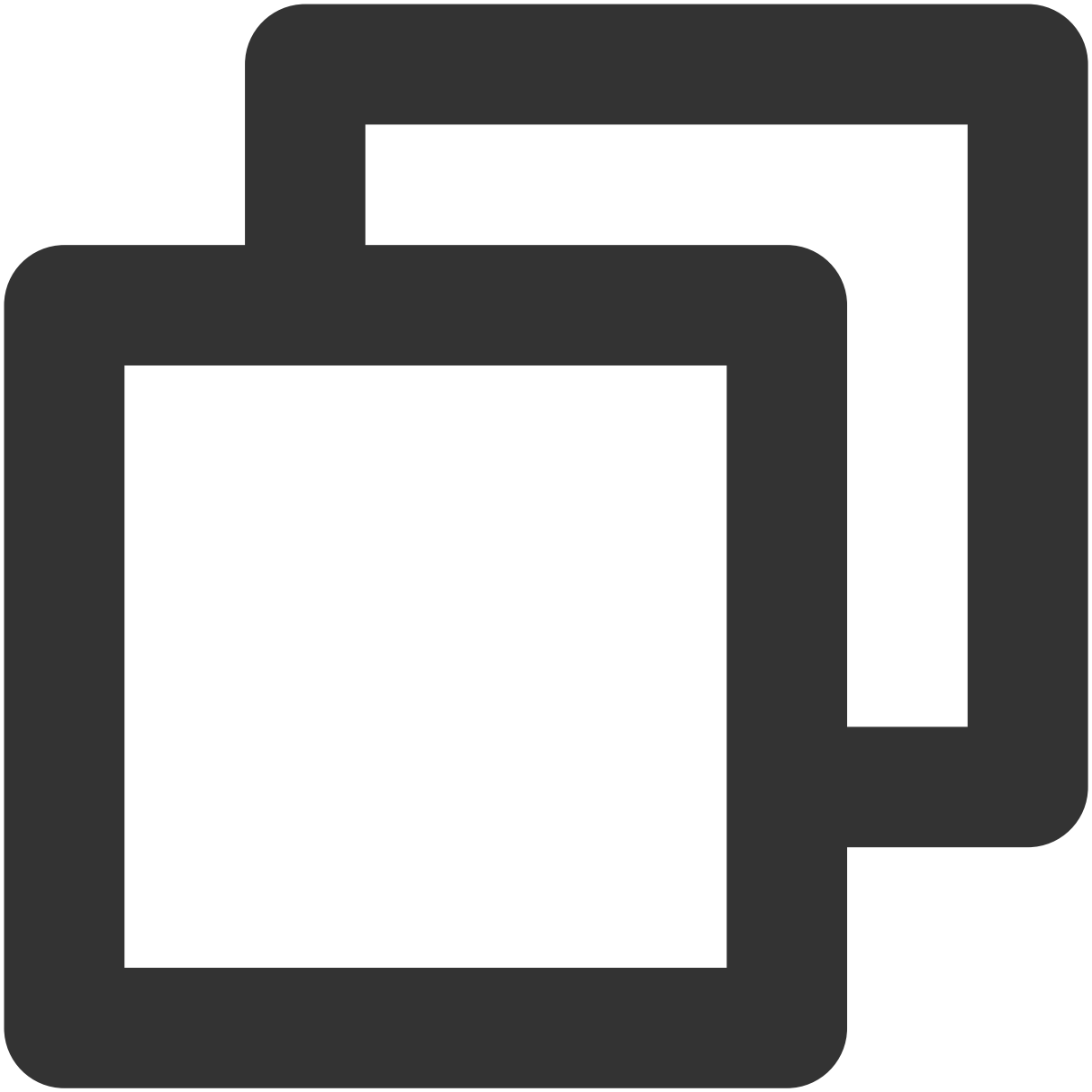
```
first_value(col)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the value of the first row of a column in a partition.

Return type: data type of the col column

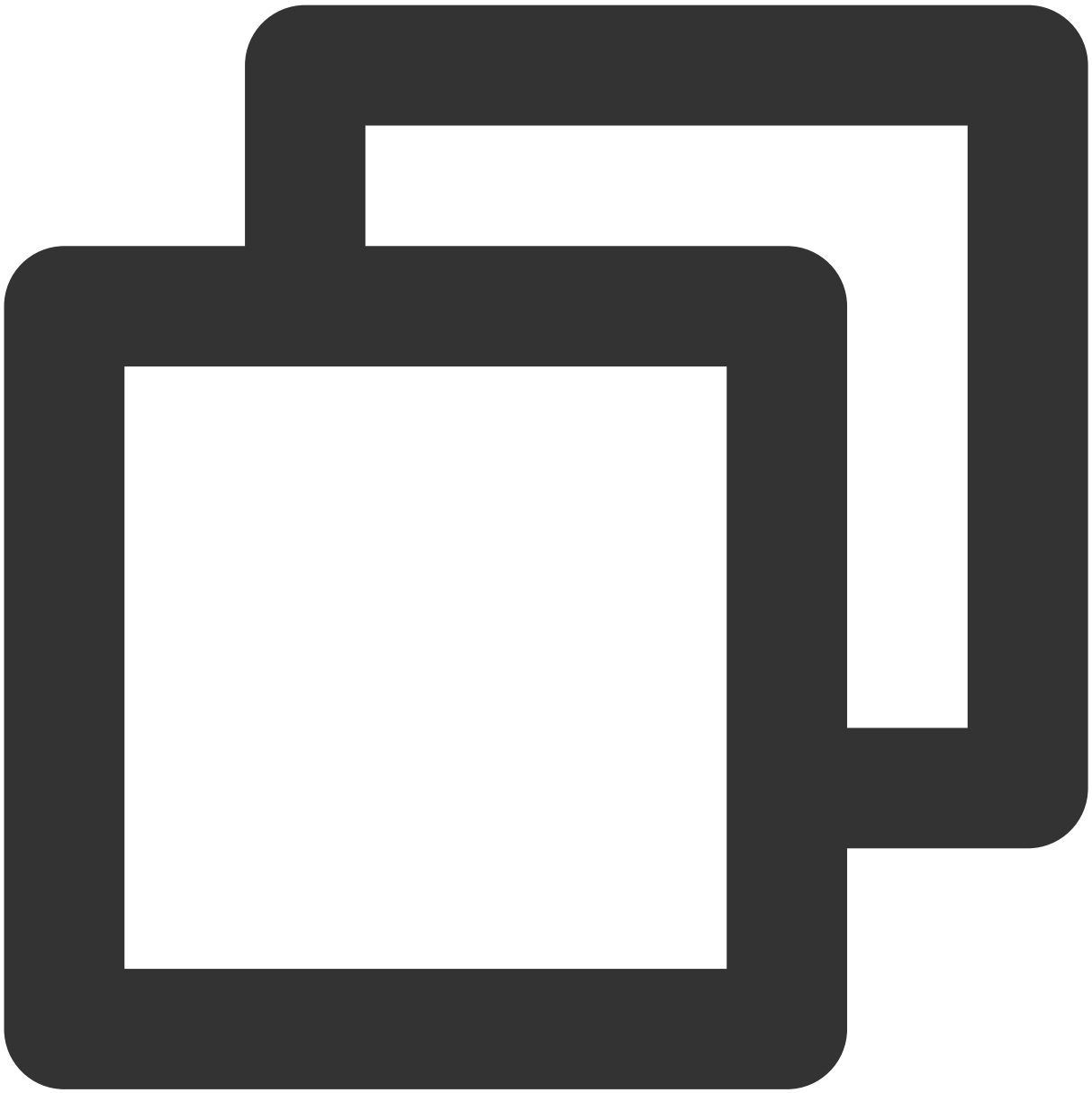
Example:



```
SELECT a, b, first_value(b) OVER (PARTITION BY a ORDER BY b) FROM VALUES ('A1', 2),
A1      1       1
A1      1       1
A1      2       1
A2      3       3
```

last_value

Function statement:



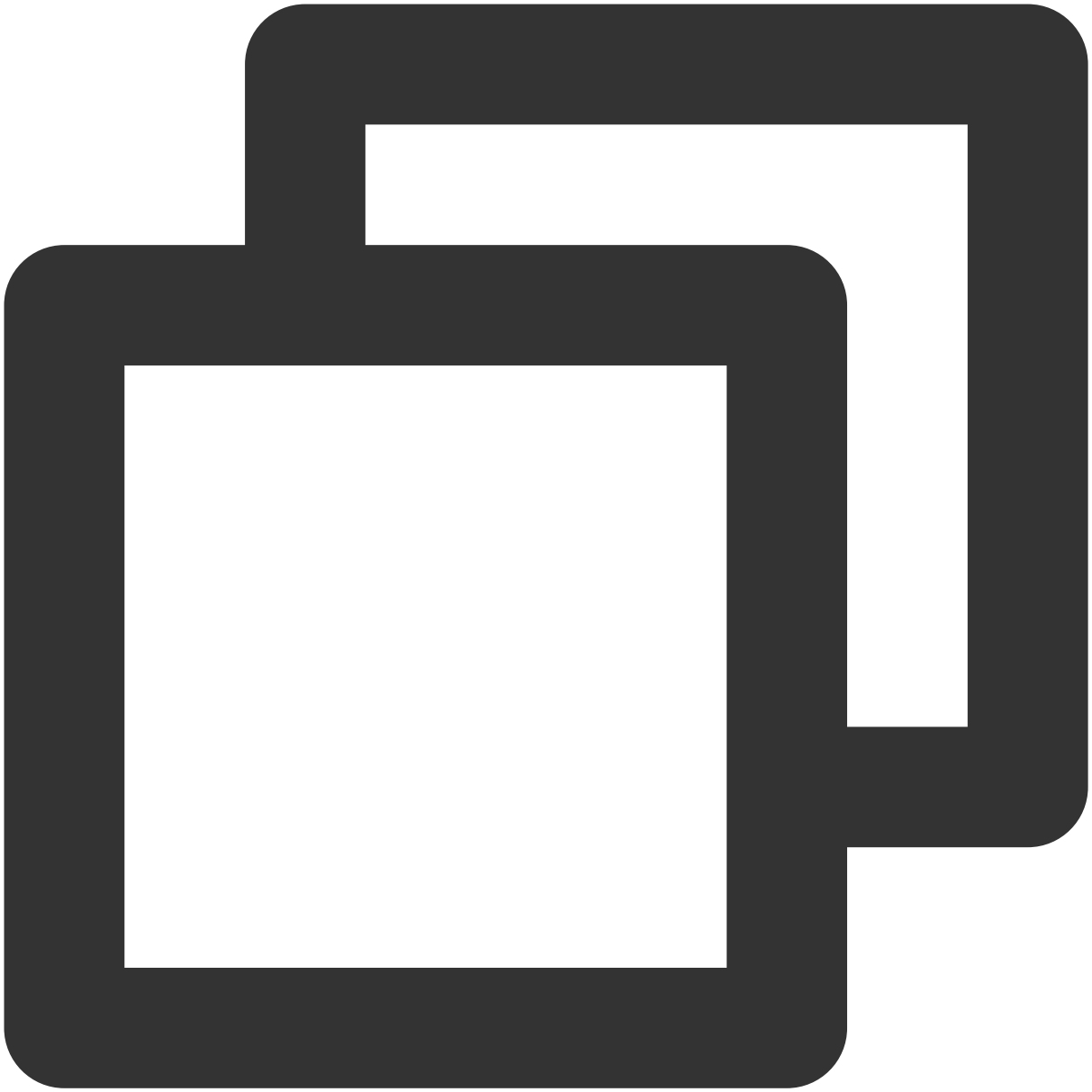
```
last_value(col)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the value of the last row of a column in a partition.

Return type: int

Example:



```
SELECT a, b, last_value(b) OVER (PARTITION BY a ORDER BY b) FROM VALUES ('A1', 2),
A1      1       1
A1      1       1
A1      2       2
A2      3       3
```

lag

Function statement:

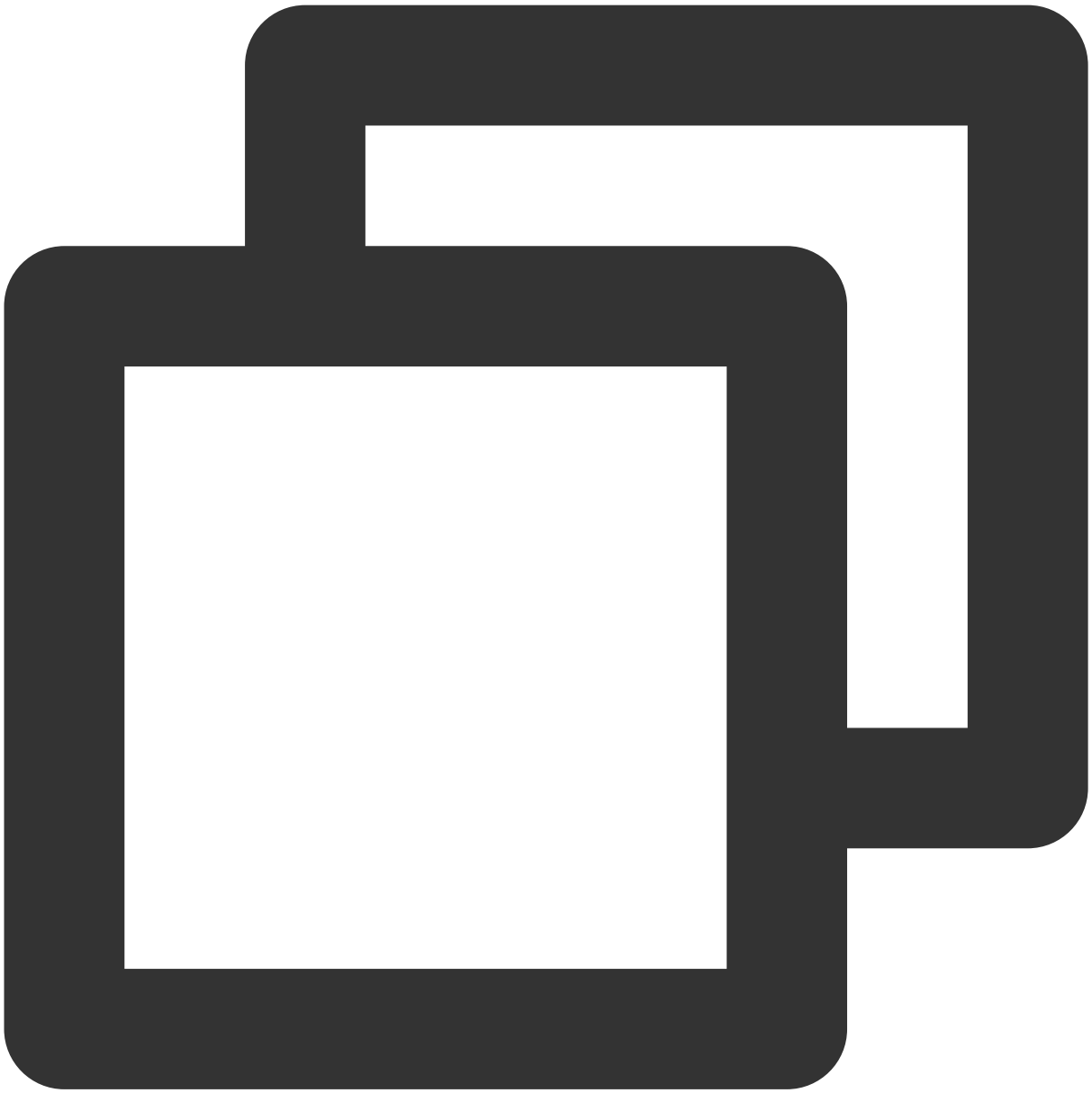
```
lag(col[, n [, default]])
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the value of the nth row above the current row in the window. The default value for n is 1, and the default value for default is null. If the value of the nth row is null, null is returned. If such an offset row does not exist (e.g., the first row of the window has no row above it when the offset is 1), default is returned. The first parameter is the column name, the second parameter is the nth row before, and the third parameter is the default value.

Return type: data type of the col column

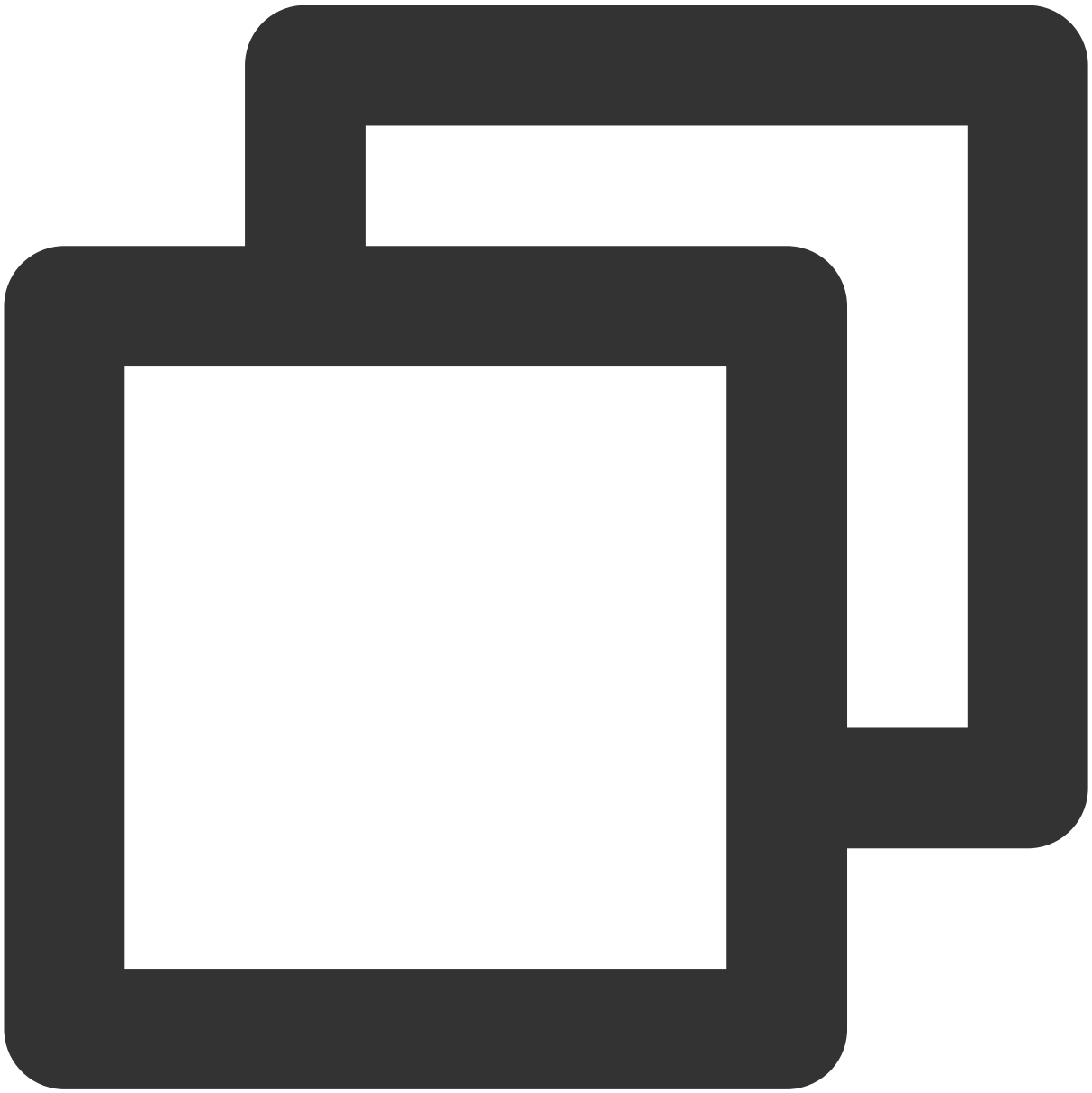
Example:



```
SELECT a, b, lag(b) OVER (PARTITION BY a ORDER BY b) FROM VALUES ('A1', 2), ('A1',
A1      1       NULL
A1      1       1
A1      2       1
A2      3       NULL
```

lead

Function statement:



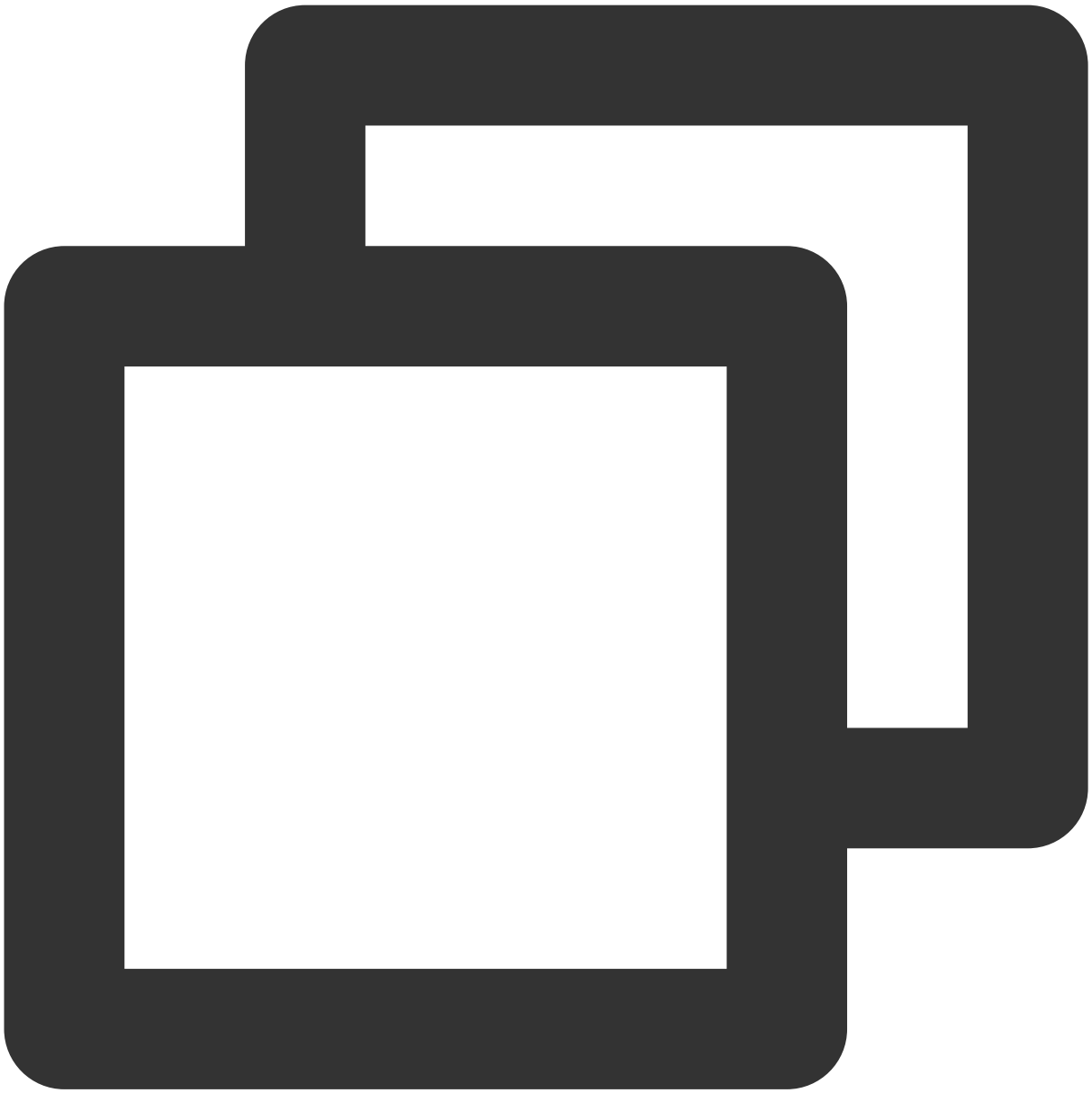
```
lead(col[, n[, default]])
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the value of the nth row after the current row in the window. The default value for n is 1, and the default value for default is null. If the value of the nth row is null, null is returned. If such an offset row does not exist (e.g., the last row of the window has no row below it when the offset is 1), default is returned. The first parameter is the column name, the second parameter is the nth row before, and the third parameter is the default value.

Return type: data type of the col column

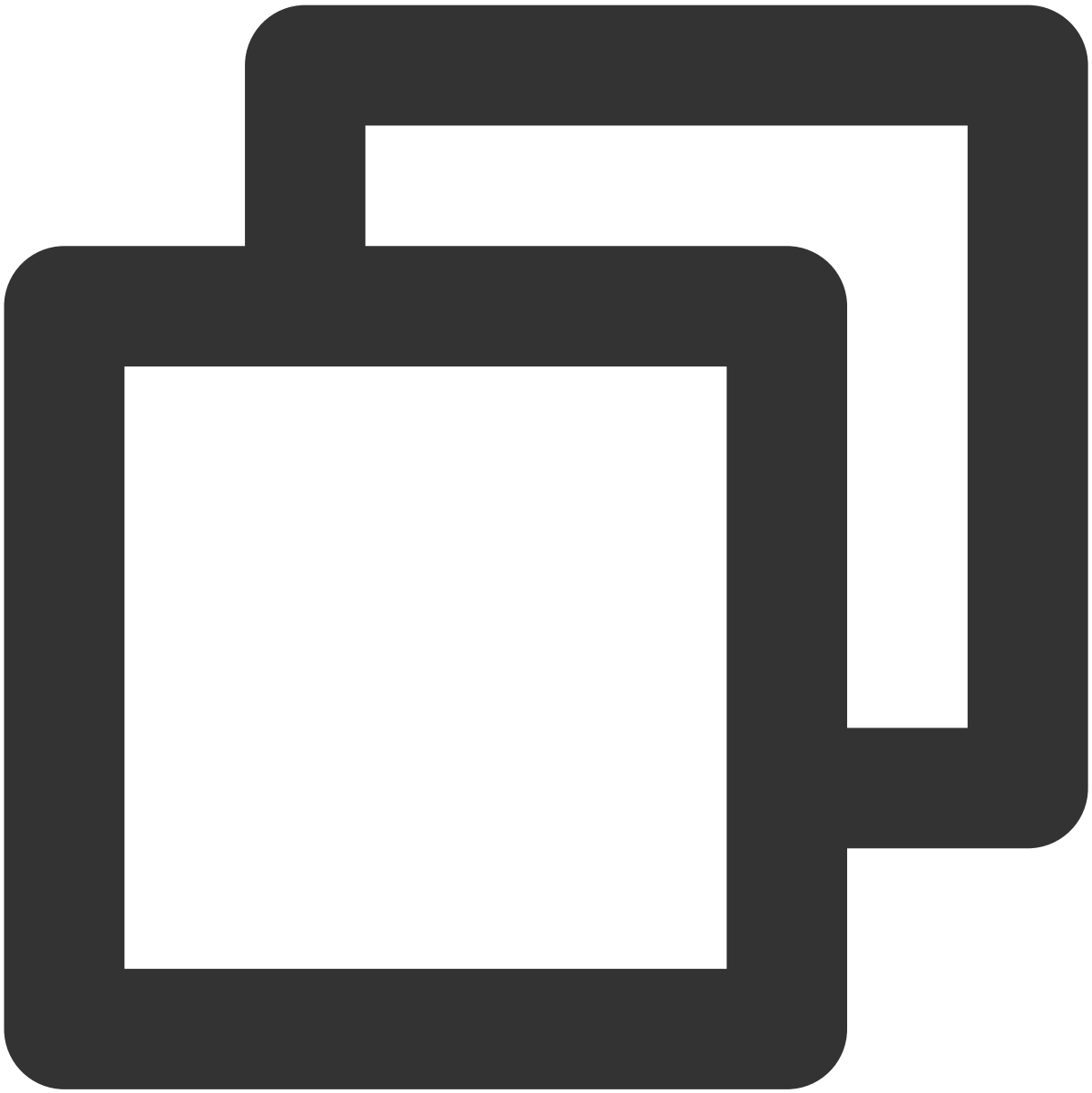
Example:



```
SELECT a, b, lead(b) OVER (PARTITION BY a ORDER BY b) FROM VALUES ('A1', 2), ('A1',
A1      1       1
A1      1       2
A1      2       NULL
A2      3       NULL
```

nth_value

Function statement:



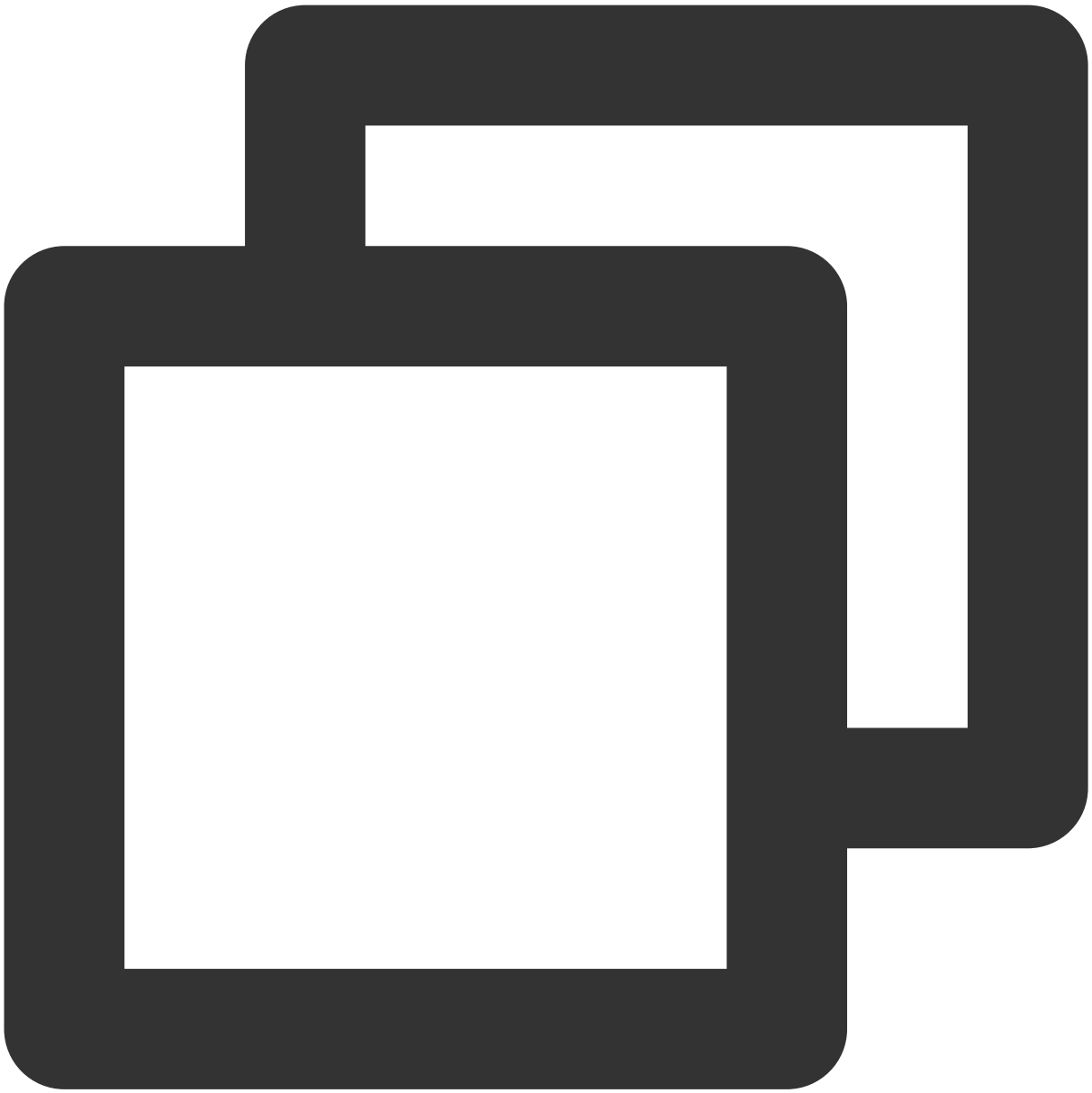
```
nth_value(col[, n])
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the value of the nth row from the start of the window. n starts from 1. If ignoreNulls=true, null will be skipped when searching for the nth row. Otherwise, every row is counted in n. If the nth row does not exist (e.g., when n is 10 but the window size is less than 10), null is returned. The first parameter is the column name, and the second parameter is the nth row.

Return type: data type of the col column

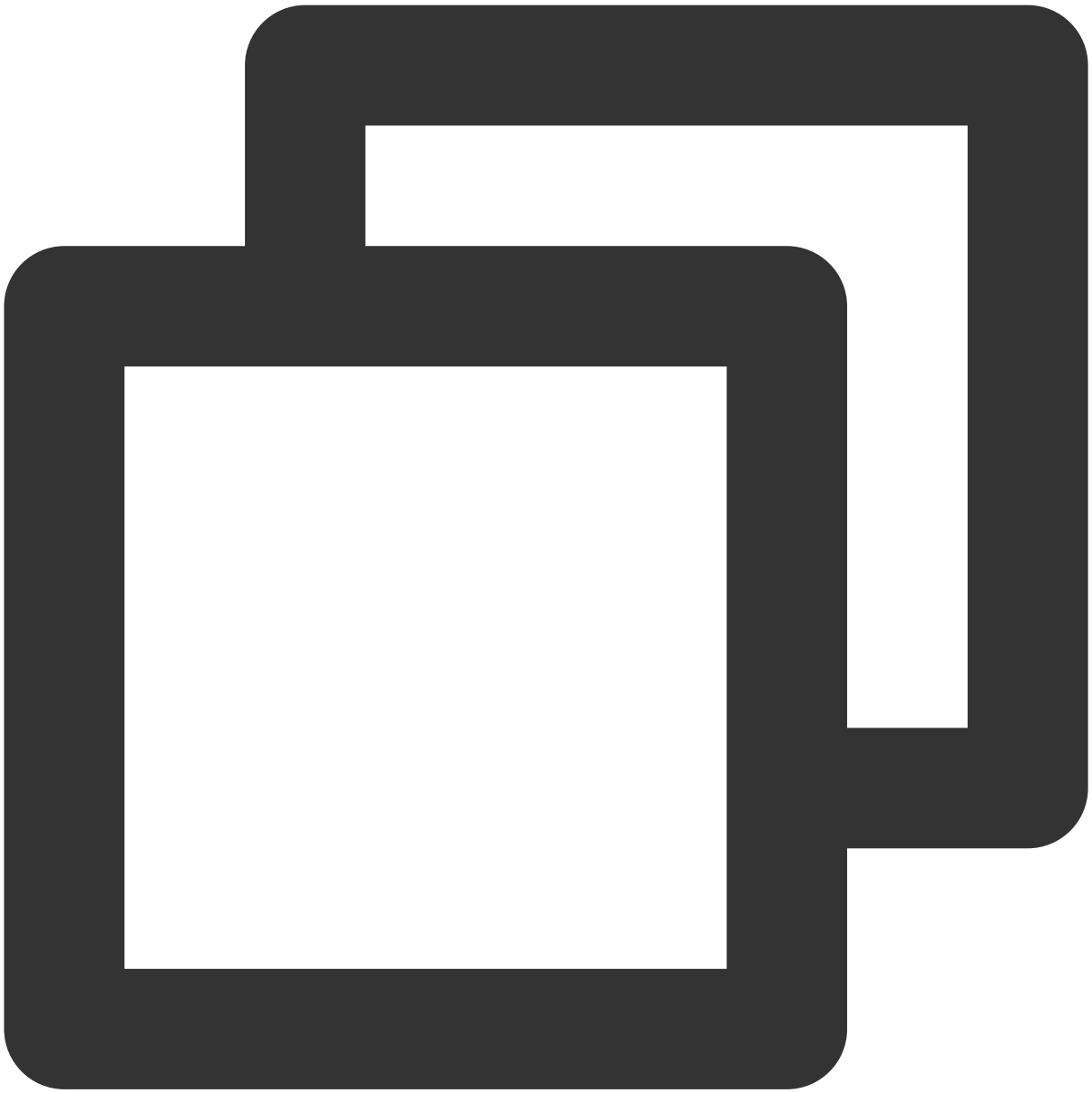
Example:



```
SELECT a, b, nth_value(b, 2) OVER (PARTITION BY a ORDER BY b) FROM VALUES ('A1', 2)
A1      1       1
A1      1       1
A1      2       1
A2      3       NULL
```

ntile

Function statement:



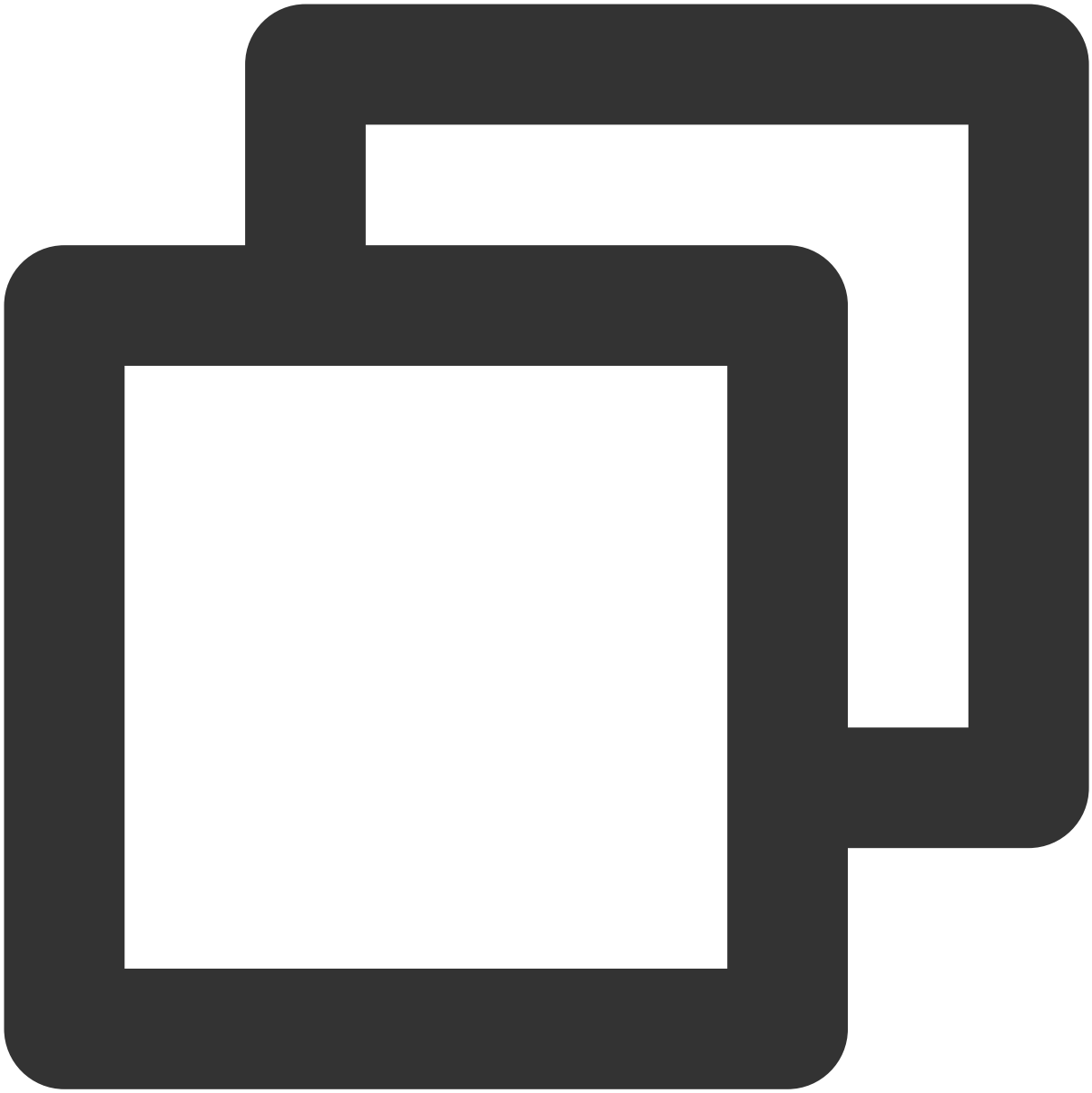
```
ntile(n)
```

Supported engines: SparkSQL and Presto

Usage instructions: Divide the rows in the window partition into n buckets and return the bucket number of the row, ranging from 1 to n.

Return type: int

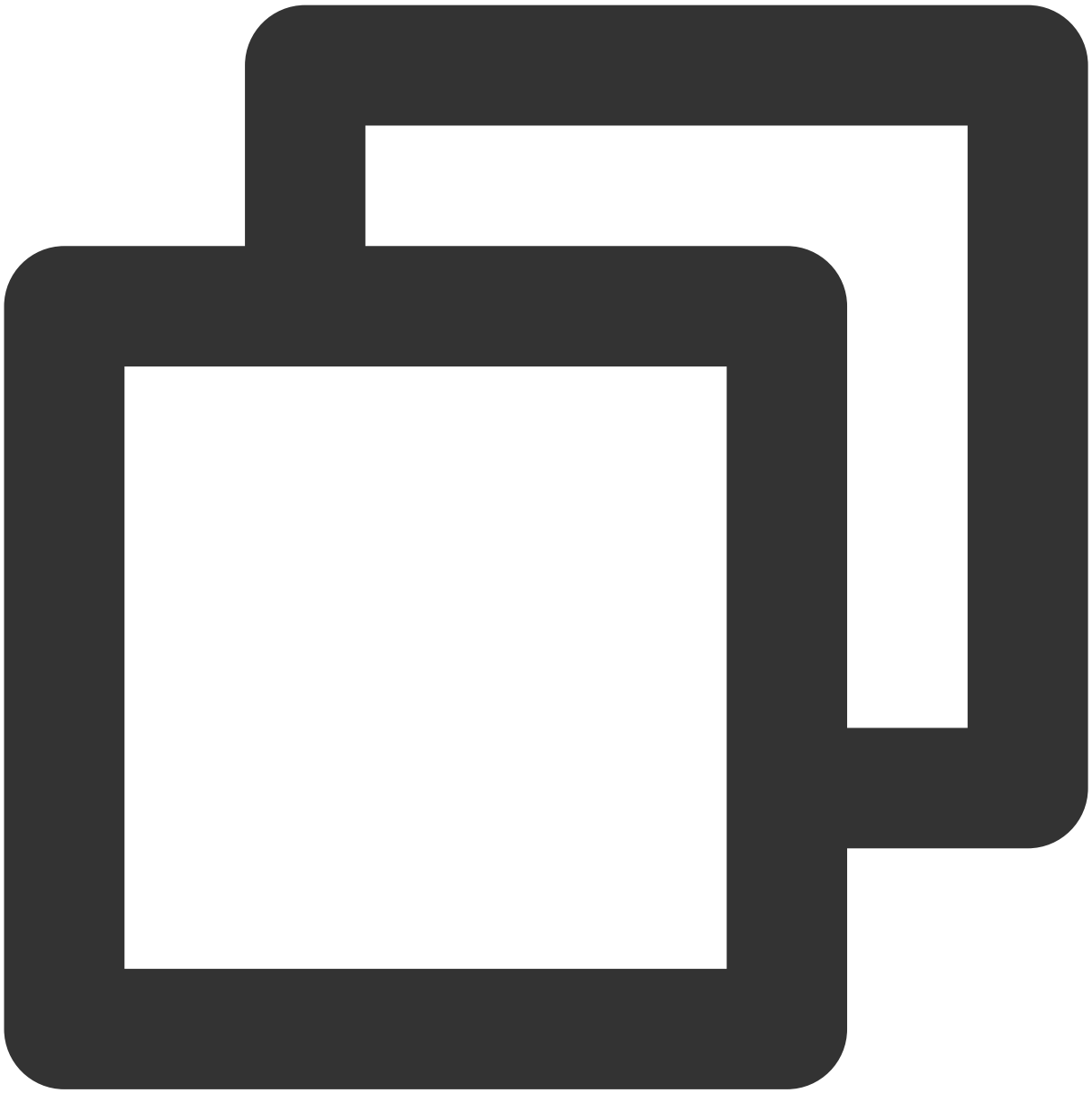
Example:



```
SELECT a, b, ntile(2) OVER (PARTITION BY a ORDER BY b) FROM VALUES ('A1', 2), ('A1'
A1      1       1
A1      1       1
A1      2       2
A2      3       1
```

CLUSTER_SAMPLE

Function statement:



```
CLUSTER_SAMPLE(<int> N[, <int> M]) over (PARTITION BY col1 ORDER by col2)
```

Supported engine: SparkSQL

Usage instructions: Sample within the window according to the specified proportion or quantity.

N: Required, int type. When only N is provided, it indicates sampling N data entries. The sampling result is close to N but not guaranteed to be exactly N.

M: Optional, int type. When M is specified, it indicates sampling the total number of entries in the M/N *window. The sampling result is close to total entries of M/N* .

Return type: boolean, where true indicates the row is sampled, and false indicates it is not sampled.

Example:

```
> SELECT a, b, cluster_sample(2) OVER (PARTITION BY a ORDER BY b) FROM VALUES ('A1'
A1 2 true
A1 1 true
A2 3 true
A1 1 false
> SELECT a, b from (select a, b, cluster_sample(2) OVER (PARTITION BY a ORDER BY b)
A1 2
A1 1
```

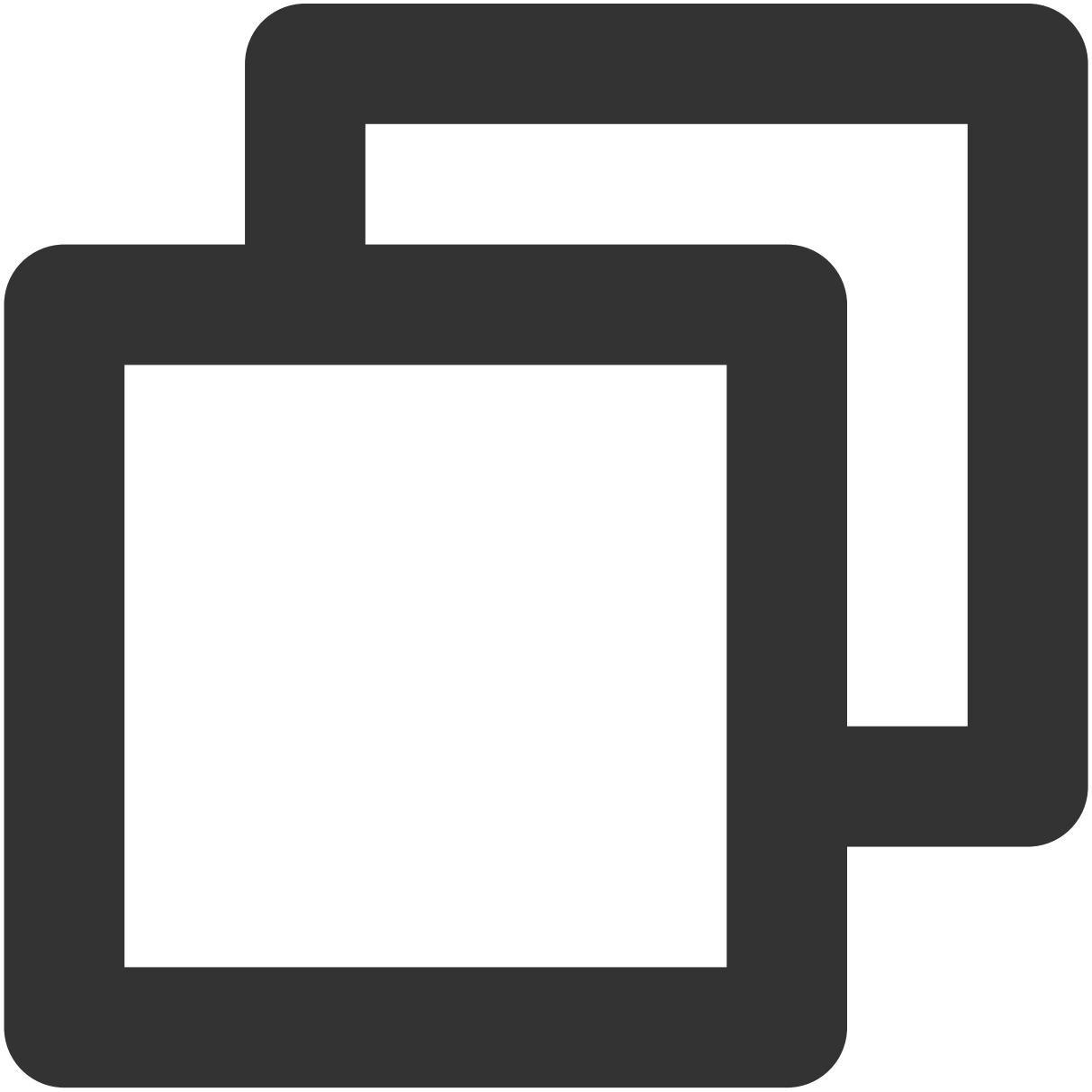
```
A2 3
```

Other Functions

Last updated：2024-08-07 17:33:39

FIELD

Function statement:



```
FIELD(<val> T, <val1> T, <val2> T, ...)
```

Supported engine: Presto

Usage instructions: Return the index of val in the list of val1, val2, etc. If it is not found, 0 is returned.

All primitive types are supported, and parameters are compared using str.equals(x). If val is NULL, the return value is 0.

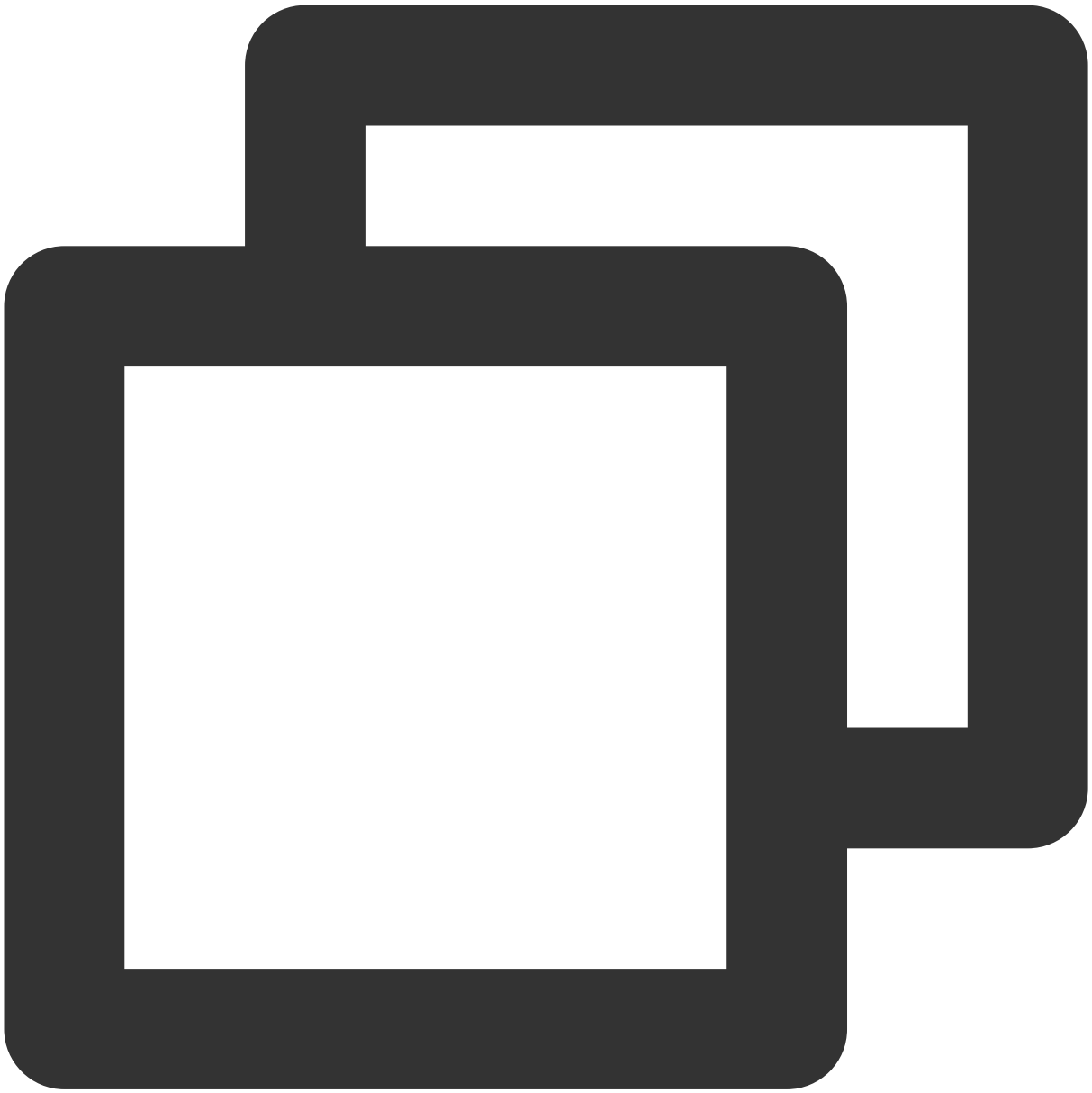
Return type: integer

Example:

```
select field('world', 'say', 'hello', 'world');
 3
```

COALESCE

Function statement:



```
COALESCE(<expr1> T, <expr2> T)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the first non-null parameter, if any. Otherwise, null is returned.

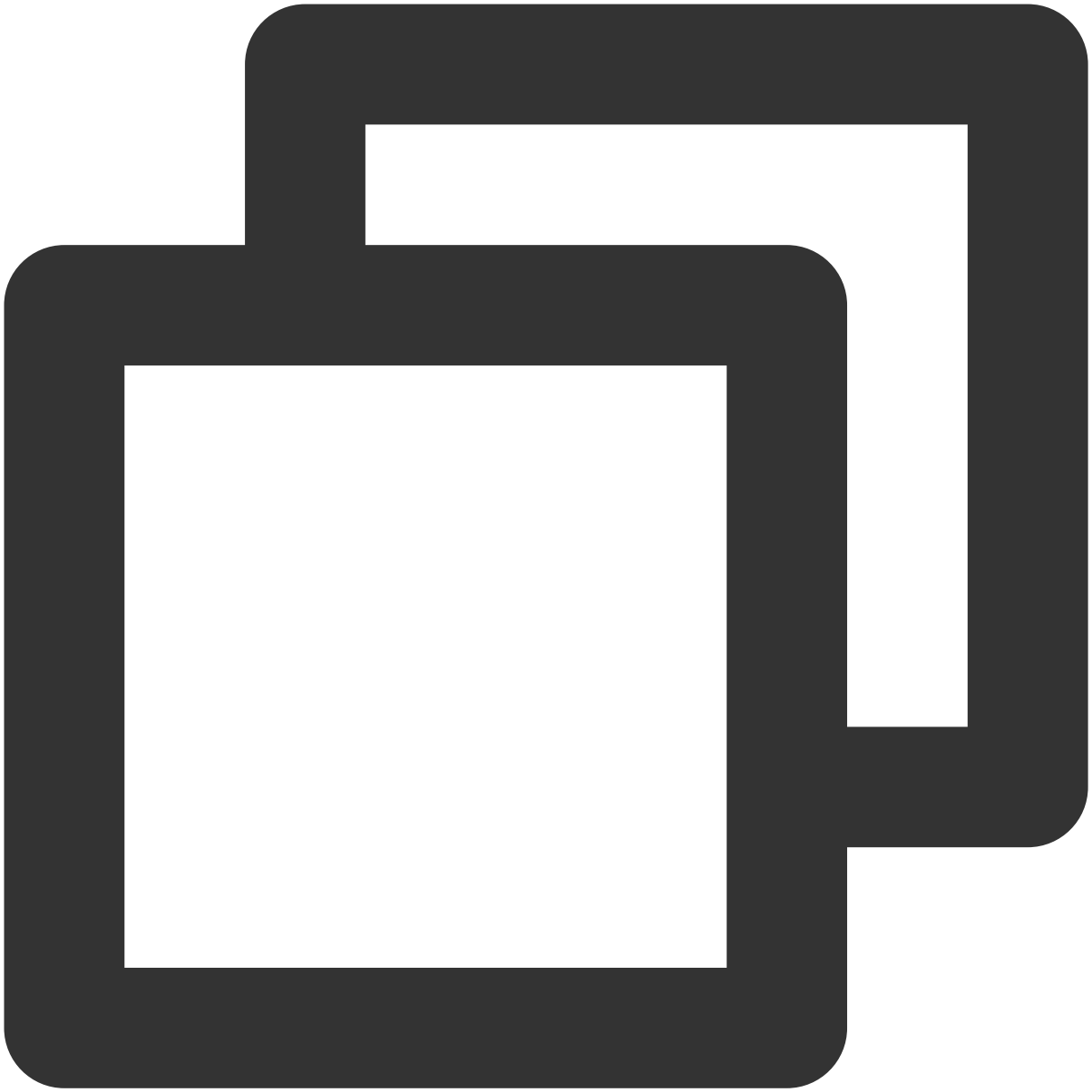
Return type: integer

Example:

```
> SELECT coalesce(NULL, 1, NULL);
 1
```

EXPLODE

Function statement:



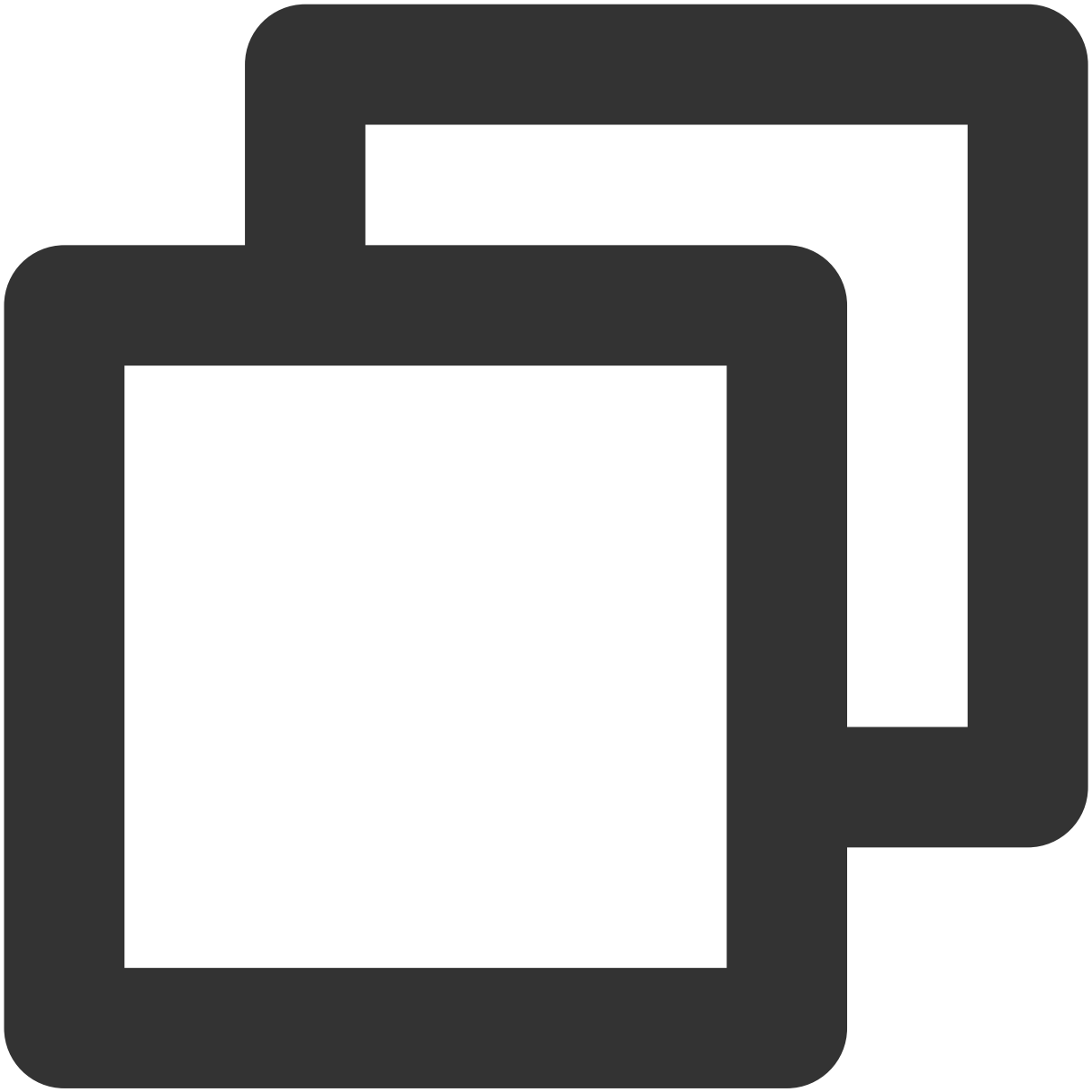
```
EXPLODE(<expr> array<T>|map<K, V>)
```

Supported engine: SparkSQL

Usage instructions: Separate the elements of an array-type expr into multiple rows, or separate a map-type expr into multiple rows and columns. Use the default column name col for array elements, or key and value for mapping elements.

Return type: row(col T) | row(key K, value V)

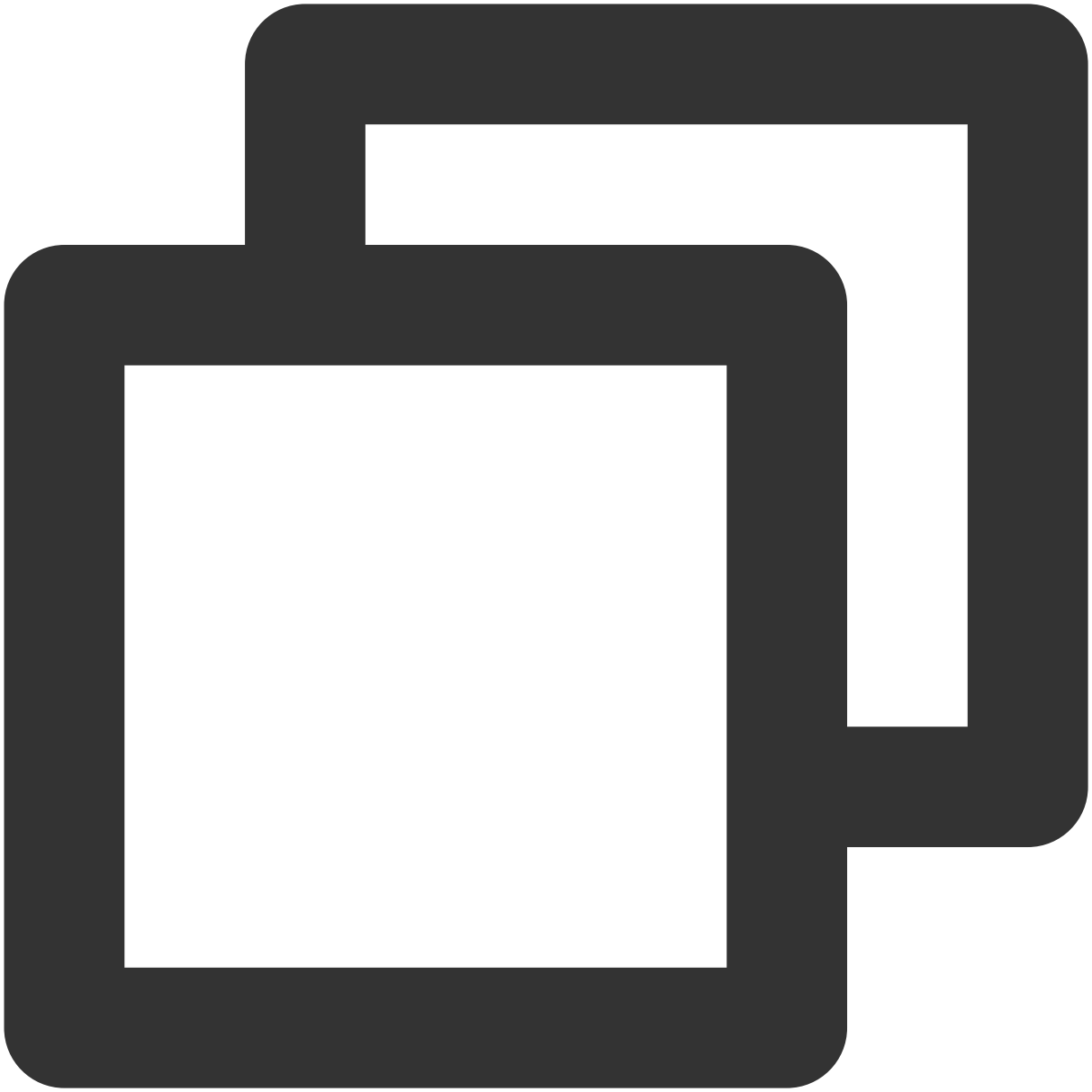
Example:



```
SELECT explode(array(10, 20));
 10
 20
```

EXPLODE_OUTER

Function statement:



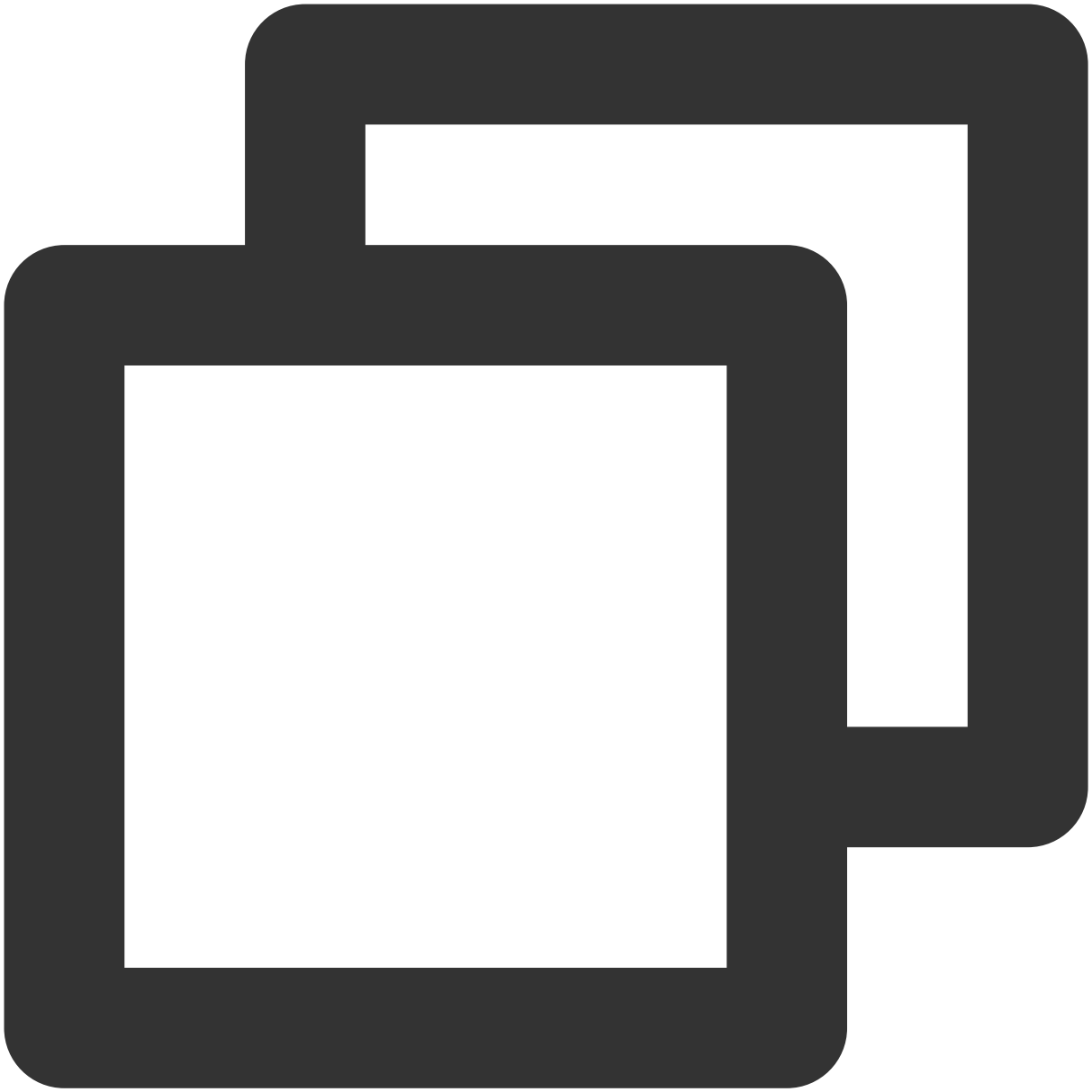
```
EXPLODE_OUTER(<expr> array<T>|map<K, V>)
```

Supported engine: SparkSQL

Usage instructions: Separate the elements of an array-type expr into multiple rows, or separate a map-type expr into multiple rows and columns. Use the default column name col for array elements, or key and value for mapping elements.

Return type: row(col T) | row(key K, value V)

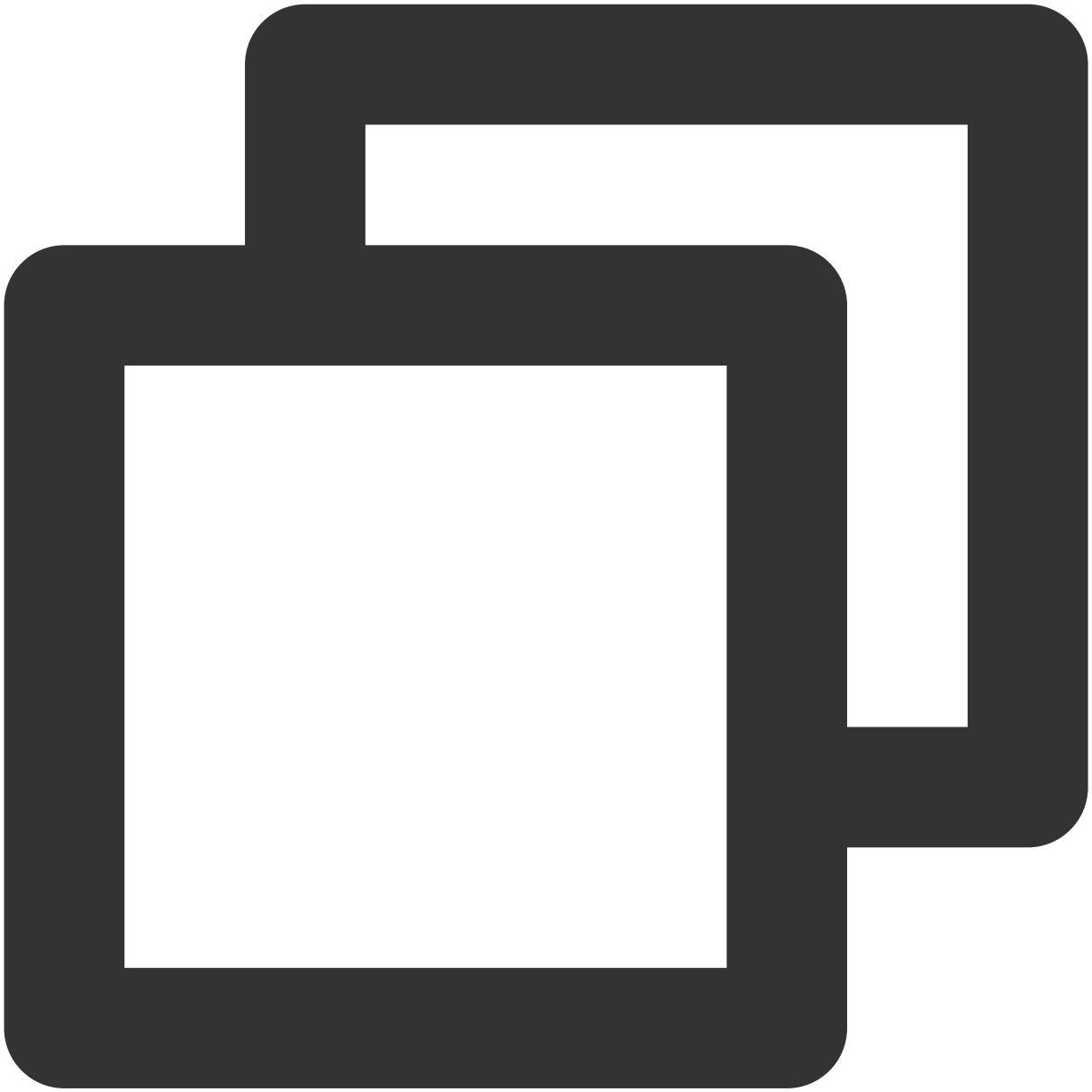
Example:



```
SELECT explode_outer(array(10, 20));
 10
 20
```

GREATEST

Function statement:



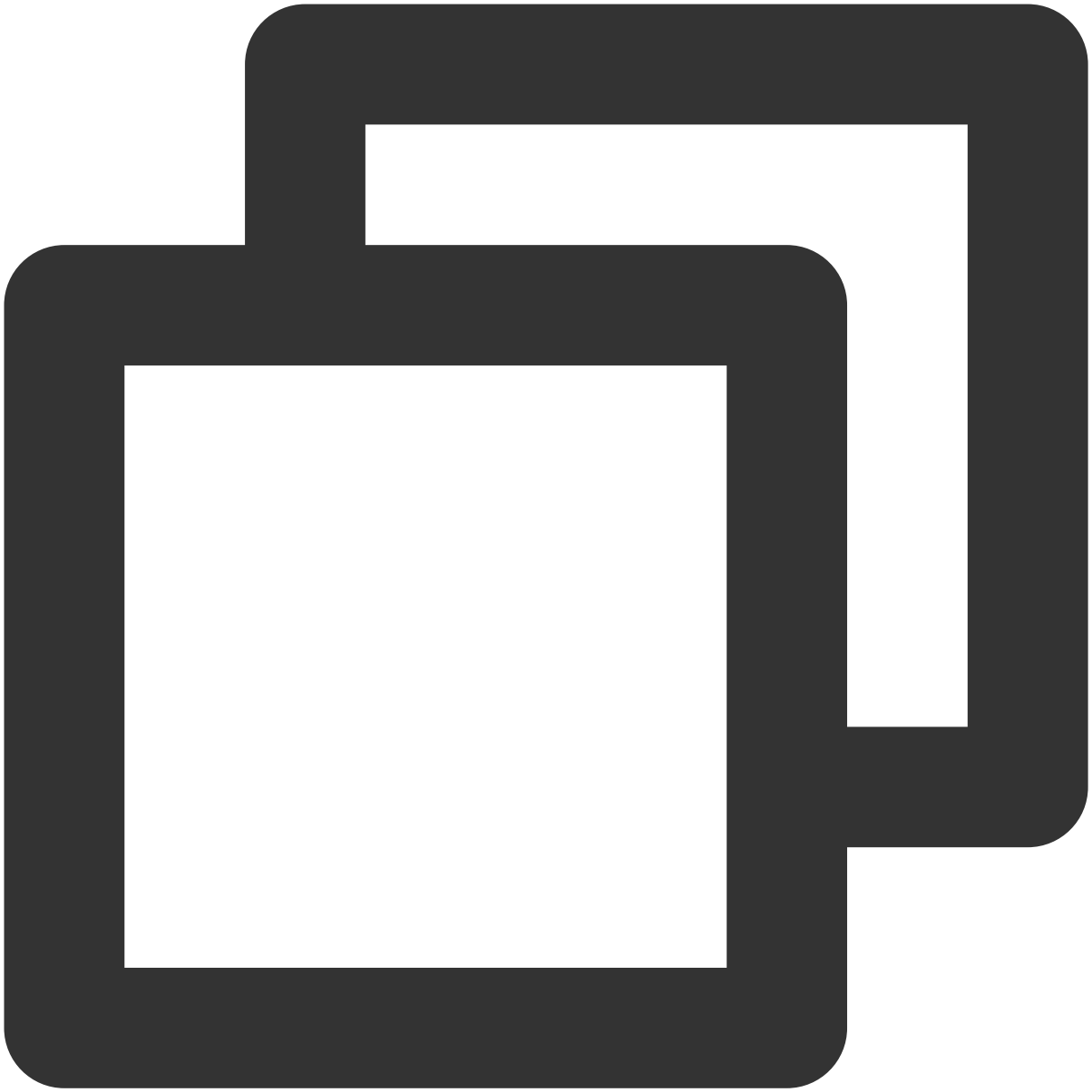
```
GREATEST(<expr1> T, <expr2> T, ...>)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the maximum value among all parameters, ignoring null values.

Return type: T

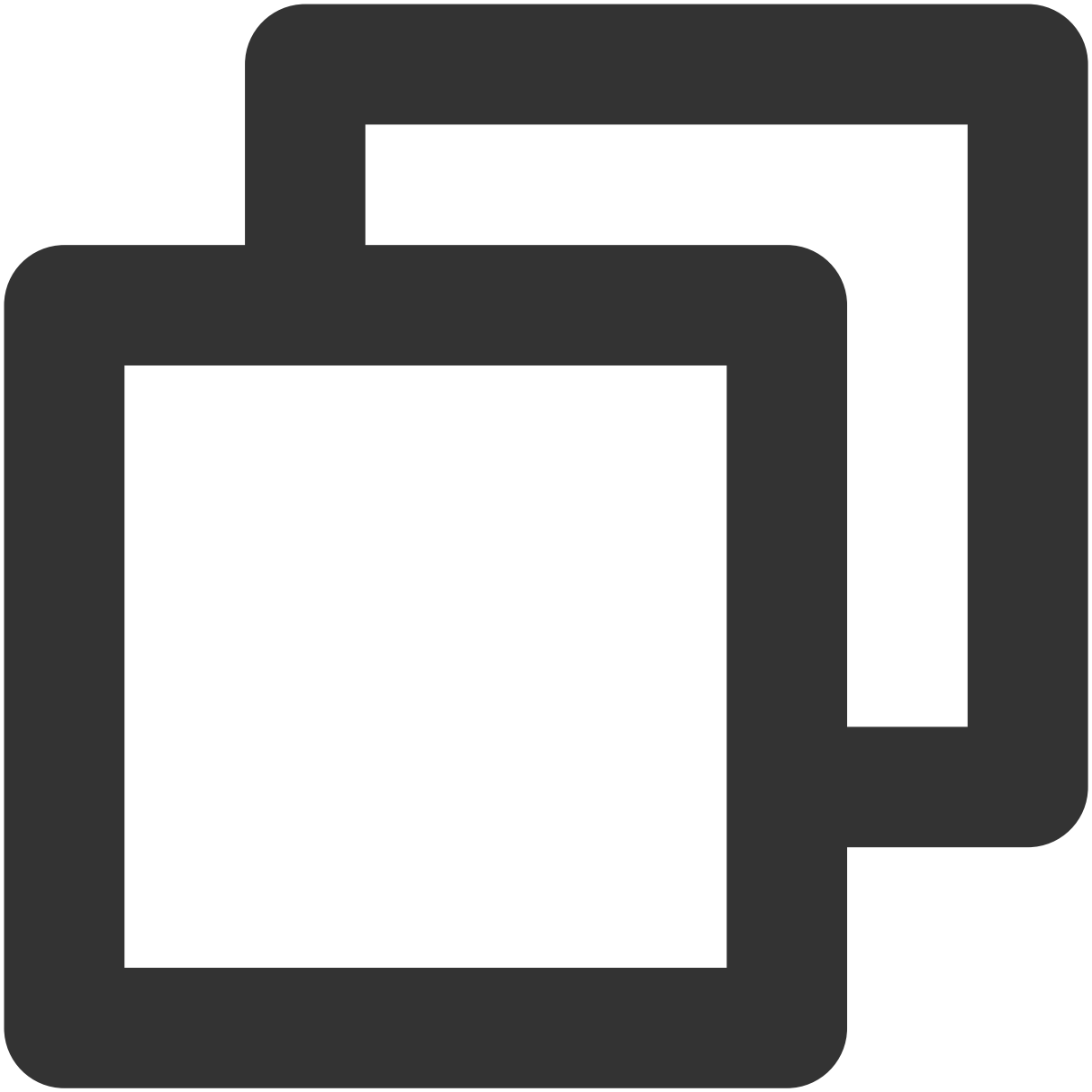
Example:



```
> SELECT greatest(10, 9, 2, 4, 3);
 10
```

IF

Function statement:



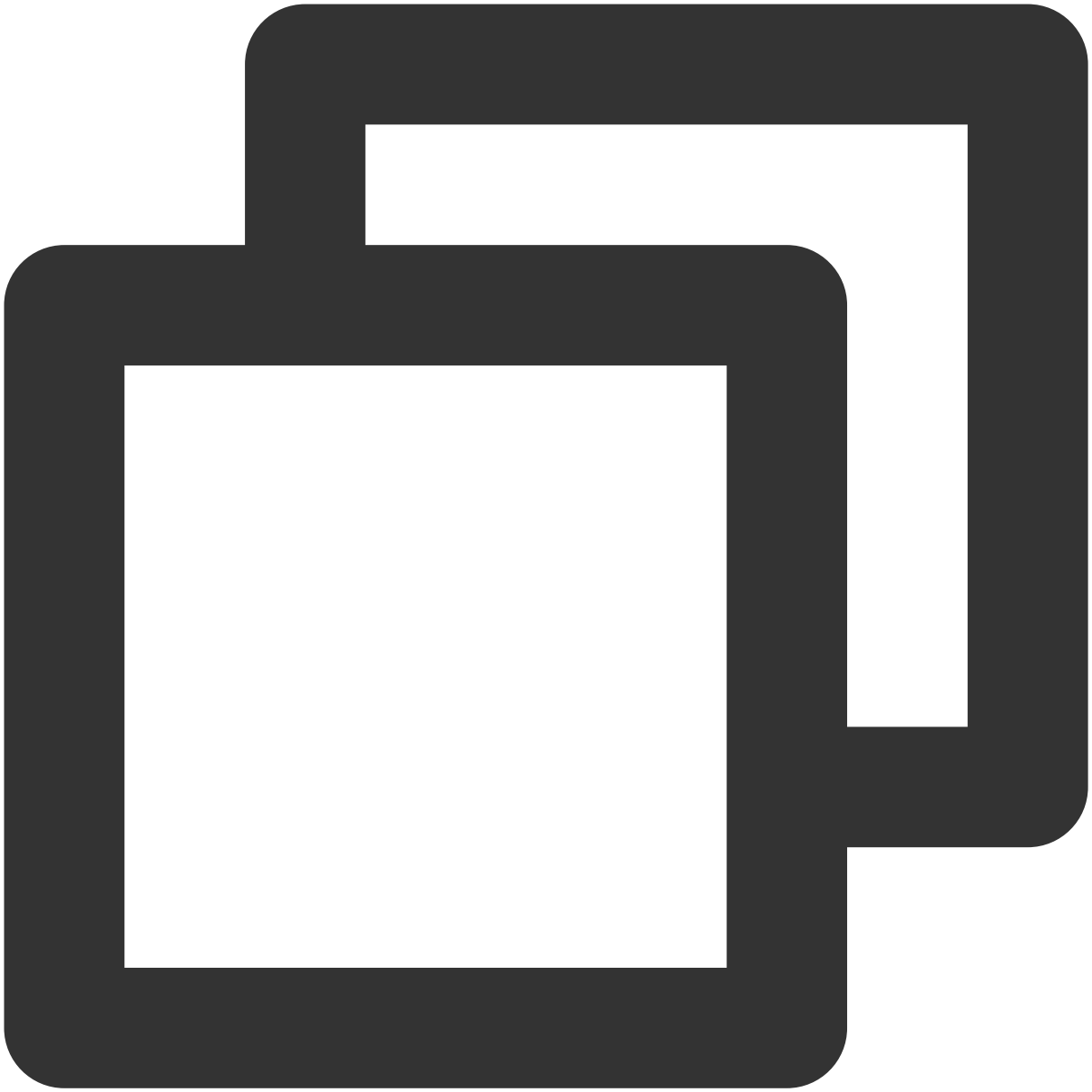
```
IF(<expr1> boolean, <expr2> T, <expr3> U)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return expr2 if expr1 evaluates to true; otherwise, return expr3.

Return type: T|U

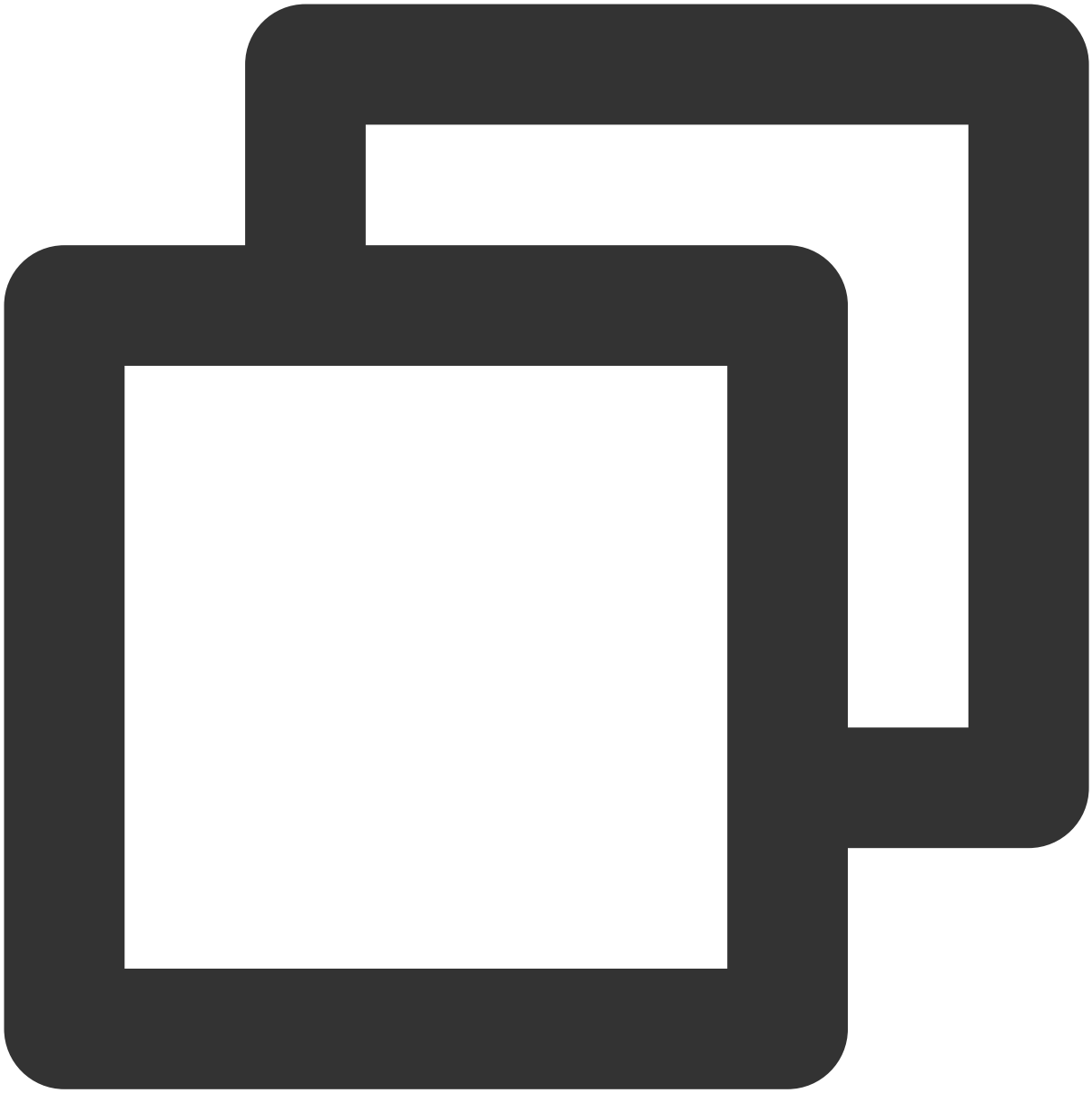
Example:



```
> SELECT if(1 < 2, 'a', 'b');
 a
```

INLINE

Function statement:



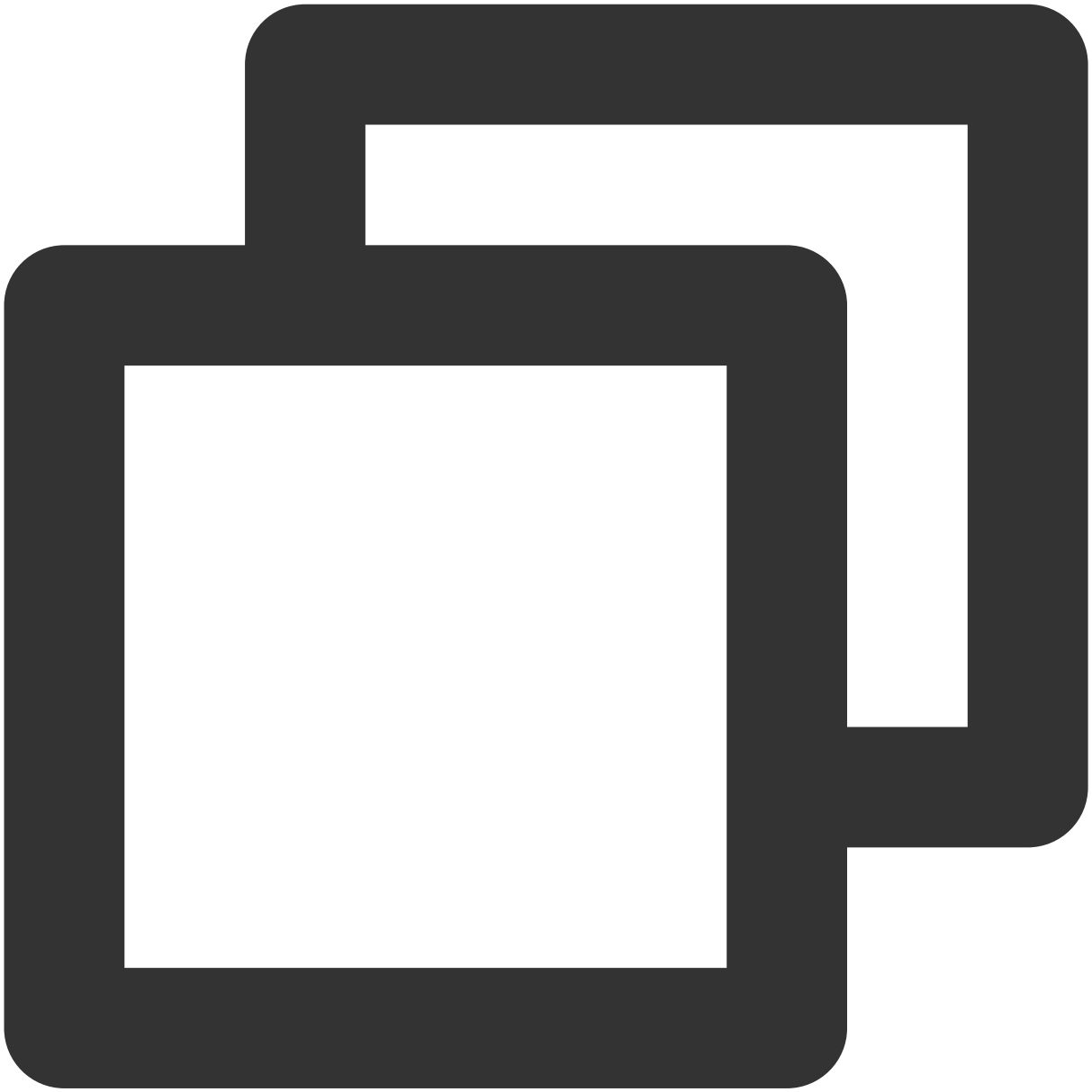
```
INLINE(a array<struct<f1:T1,...,fn:Tn>>)
```

Supported engine: SparkSQL

Usage instructions: Convert an array of structs into a table. By default, col1, col2, etc. are used as column names.

Return type: row(T1, ..., Tn)

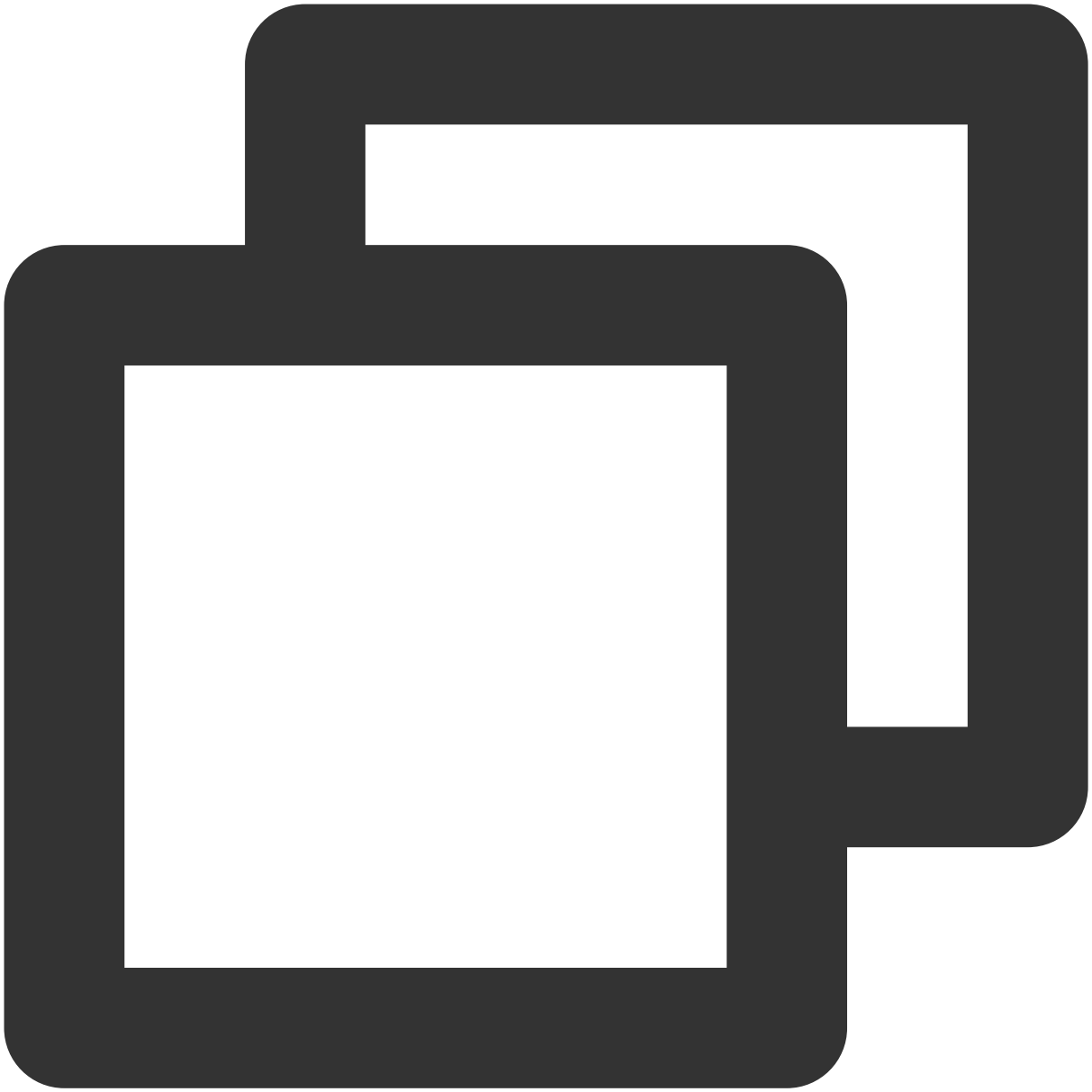
Example:



```
> SELECT inline(array(struct(1, 'a'), struct(2, 'b')));
 1  a
 2  b
```

INLINE_OUTER

Function statement:



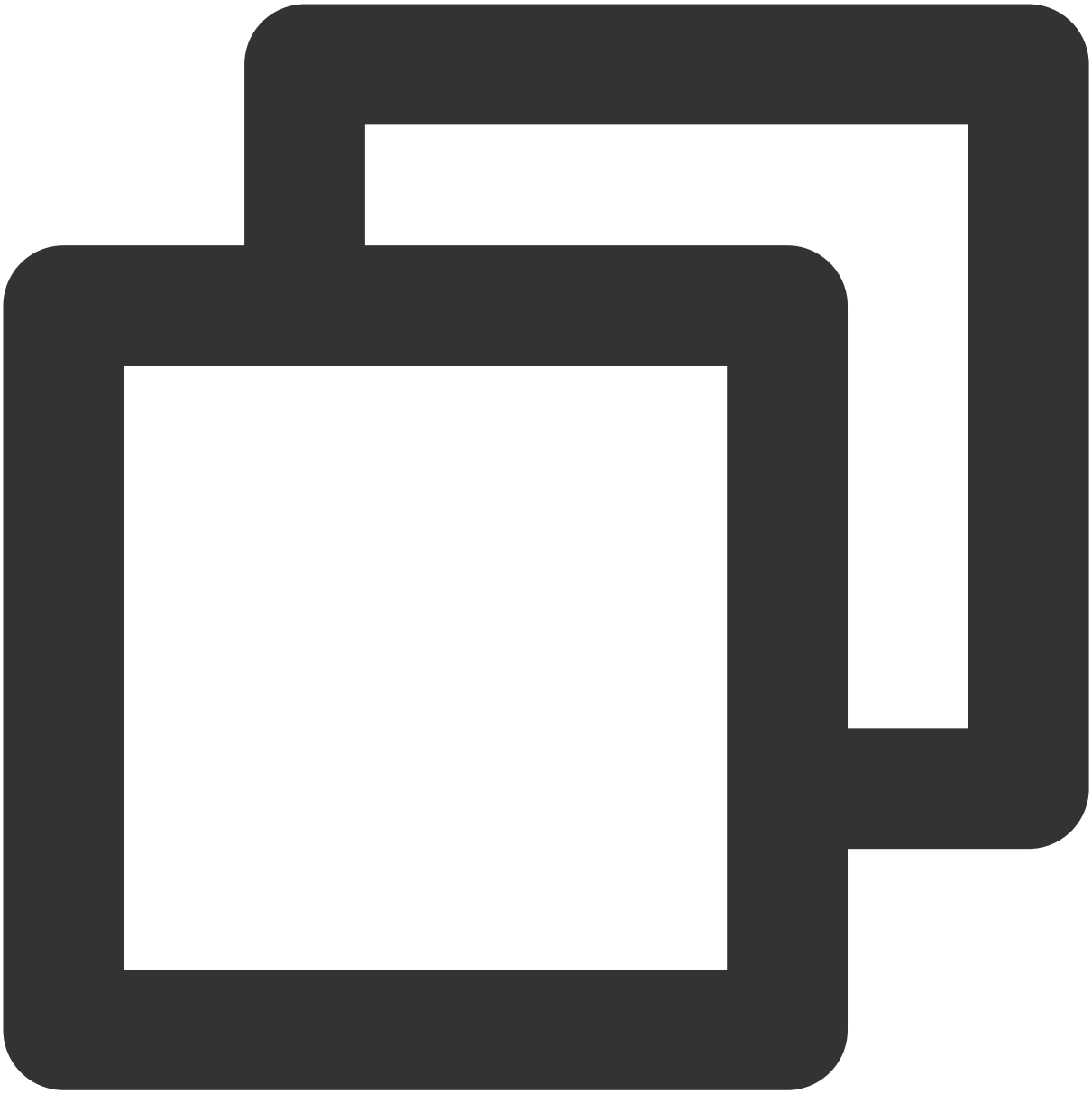
```
INLINE_OUTER(a array<struct<f1:T1,...,fn:Tn>>)
```

Supported engine: SparkSQL

Usage instructions: Convert an array of structs into a table. By default, col1, col2, etc. are used as column names.

Return type: row(T1, ..., Tn)

Example:



```
> SELECT inline(array(struct(1, 'a'), struct(2, 'b')));
 1  a
 2  b
```

IN

Function statement:

```
<expr1> IN(<expr2> T, <expr3> T, ...)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return true if expr1 is equal to any of exprn.

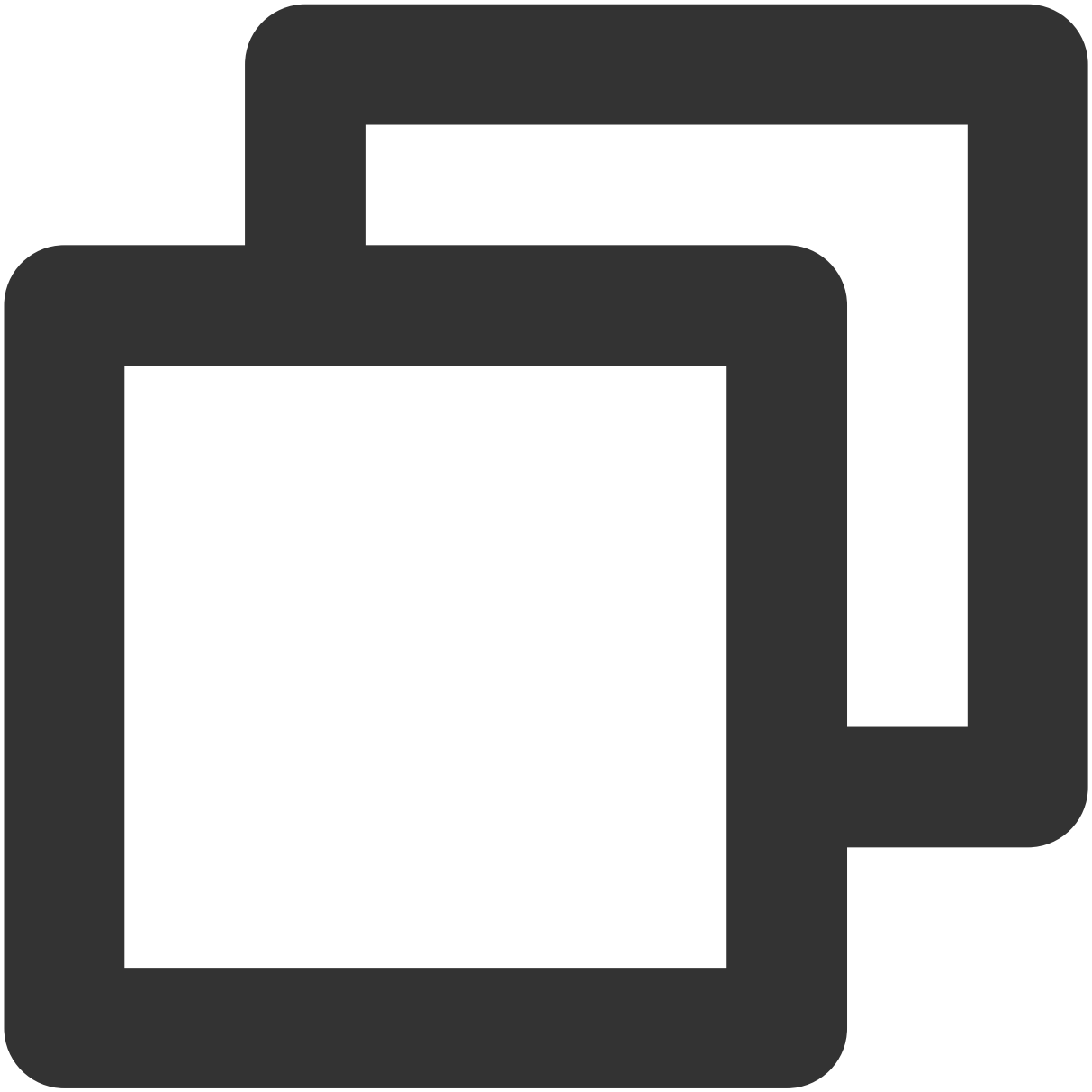
Return type: boolean

Example:

```
> SELECT 1 in(1, 2, 3);
 true
> SELECT 1 in(2, 3, 4);
 false
```

ISNAN

Function statement:



```
ISNAN(<expr> T)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return true if expr is NaN; otherwise, return false.

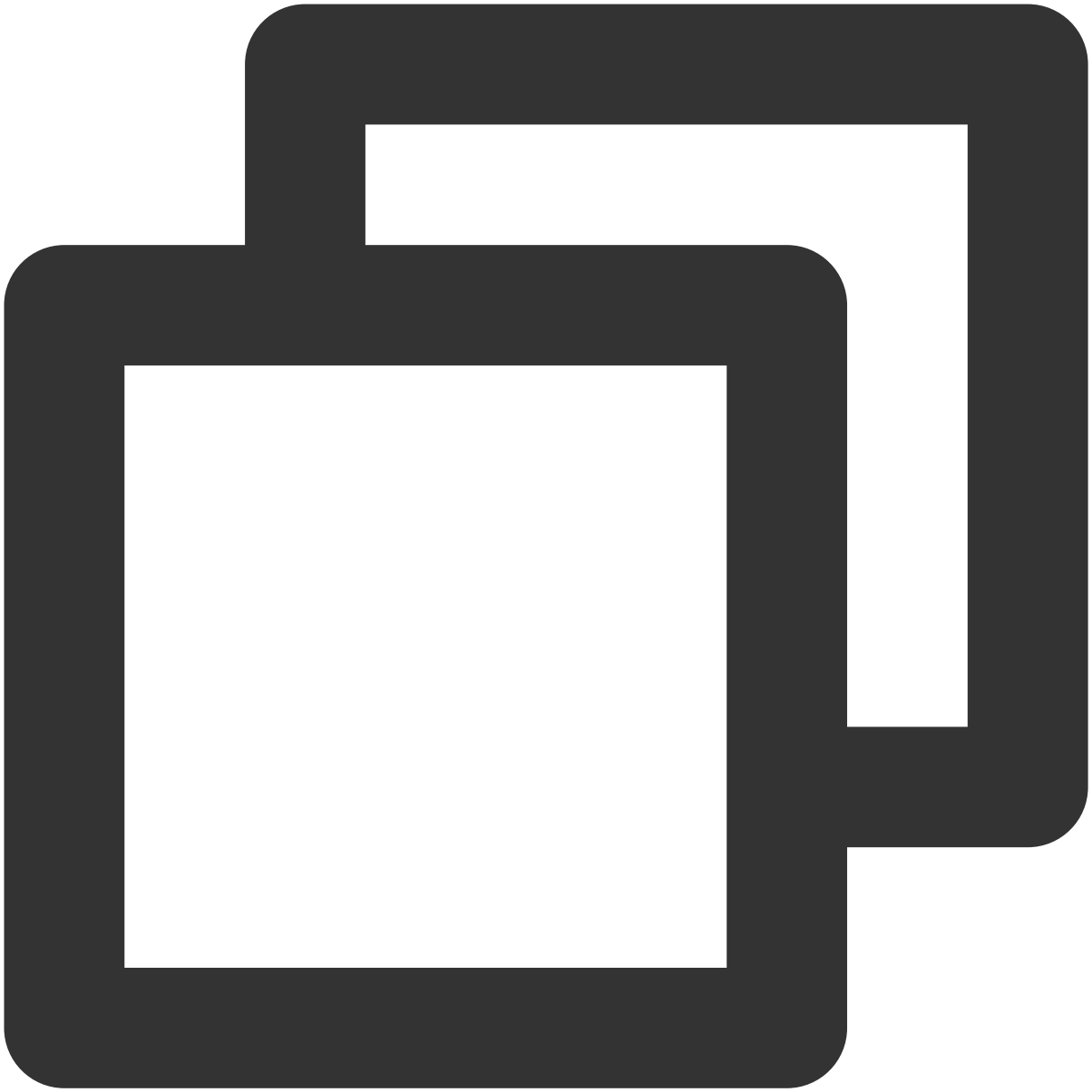
Return type: boolean

Example:

```
> SELECT isnan(cast('NaN' as double));
 true
```

IFNULL

Function statement:



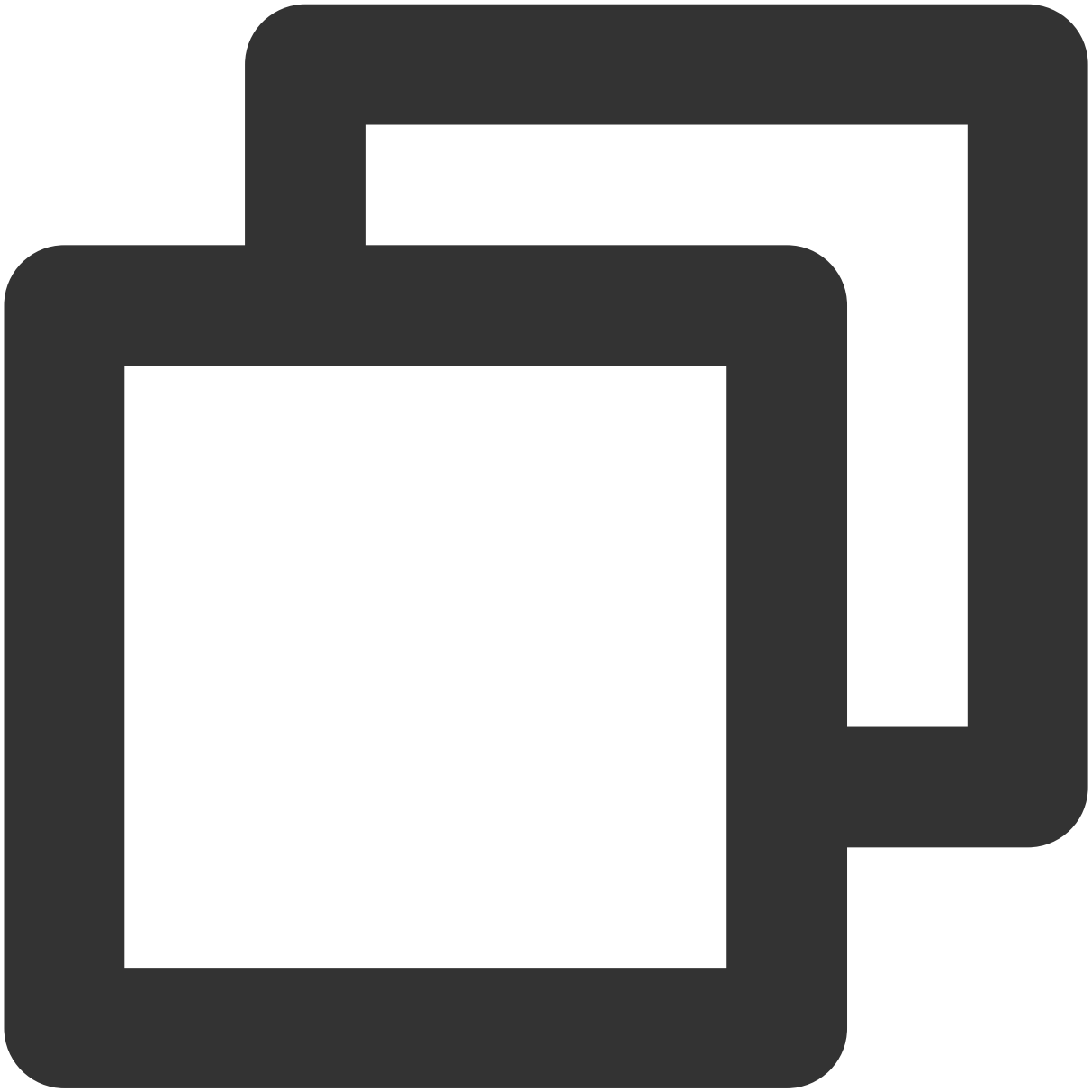
```
IFNULL(<expr1> T, <expr2> U)
```

Supported engine: SparkSQL

Usage instructions: Return expr2 if expr1 is null; otherwise, return expr1.

Return type: T|U

Example:



```
> SELECT ifnull(NULL, array('2'));
 ["2"]
```

ISNULL

Function statement:

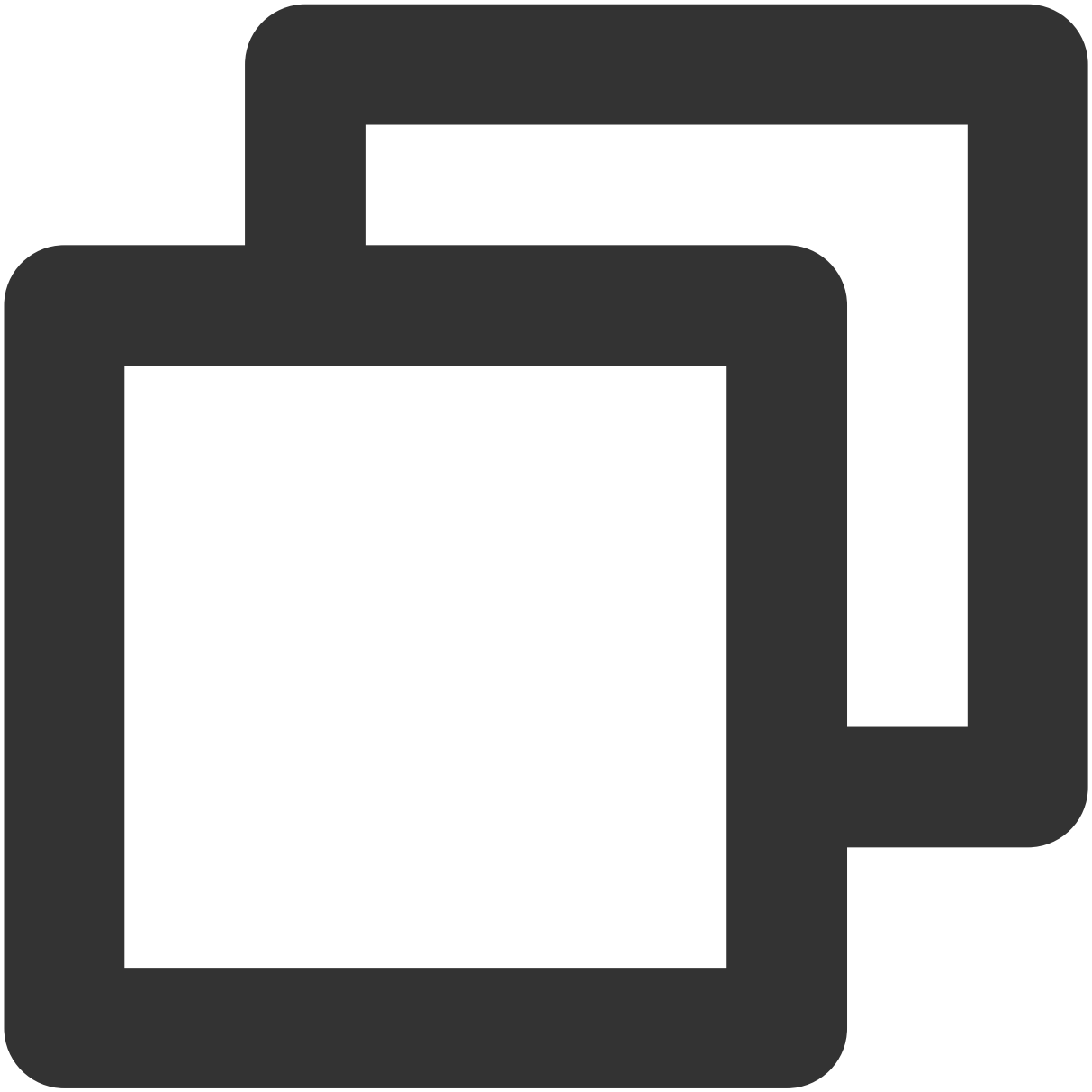
```
ISNULL(<expr> T)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return true if expr is null; otherwise, return false.

Return type: boolean

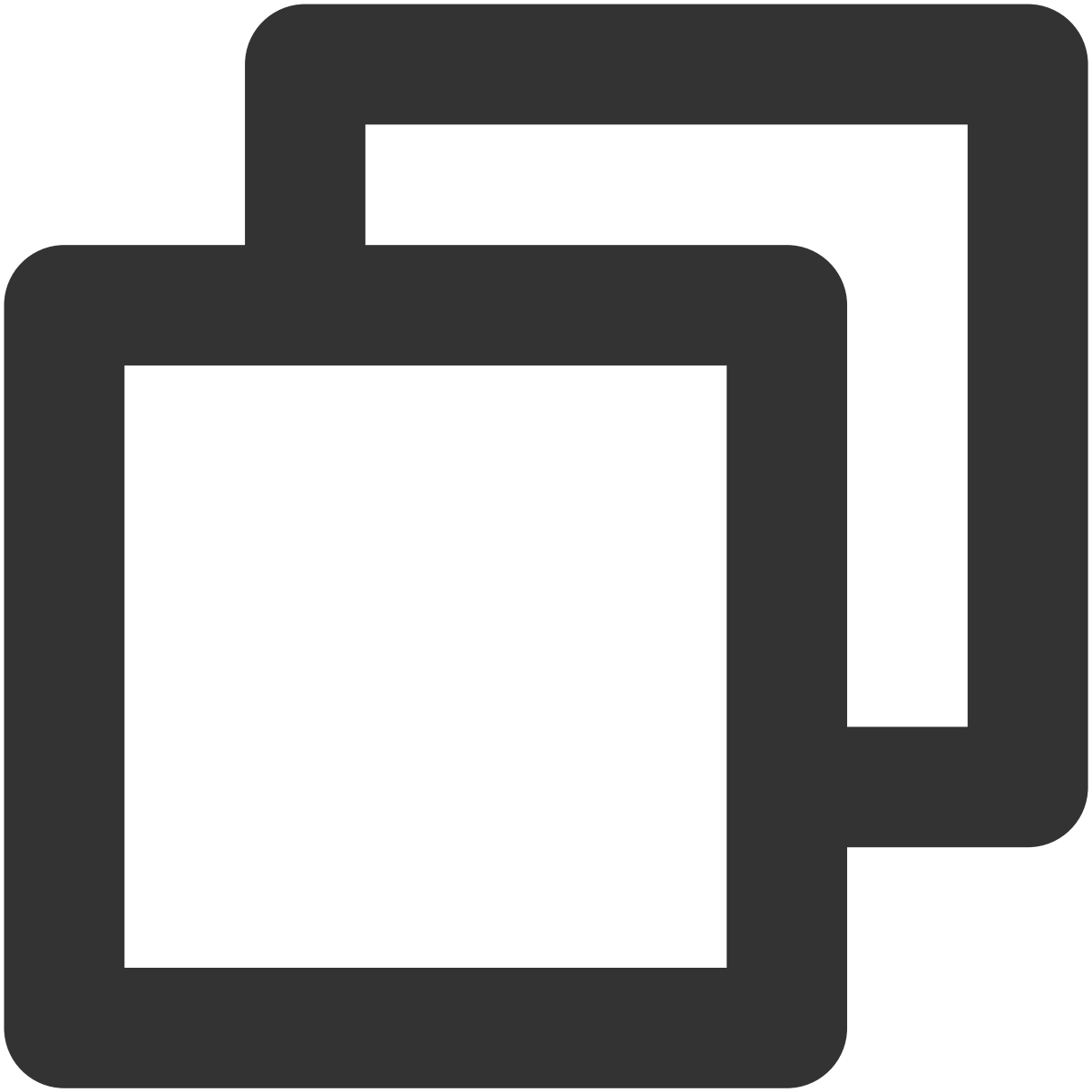
Example:



```
> SELECT isnull(1);
 false
```

ISNOTNULL

Function statement:



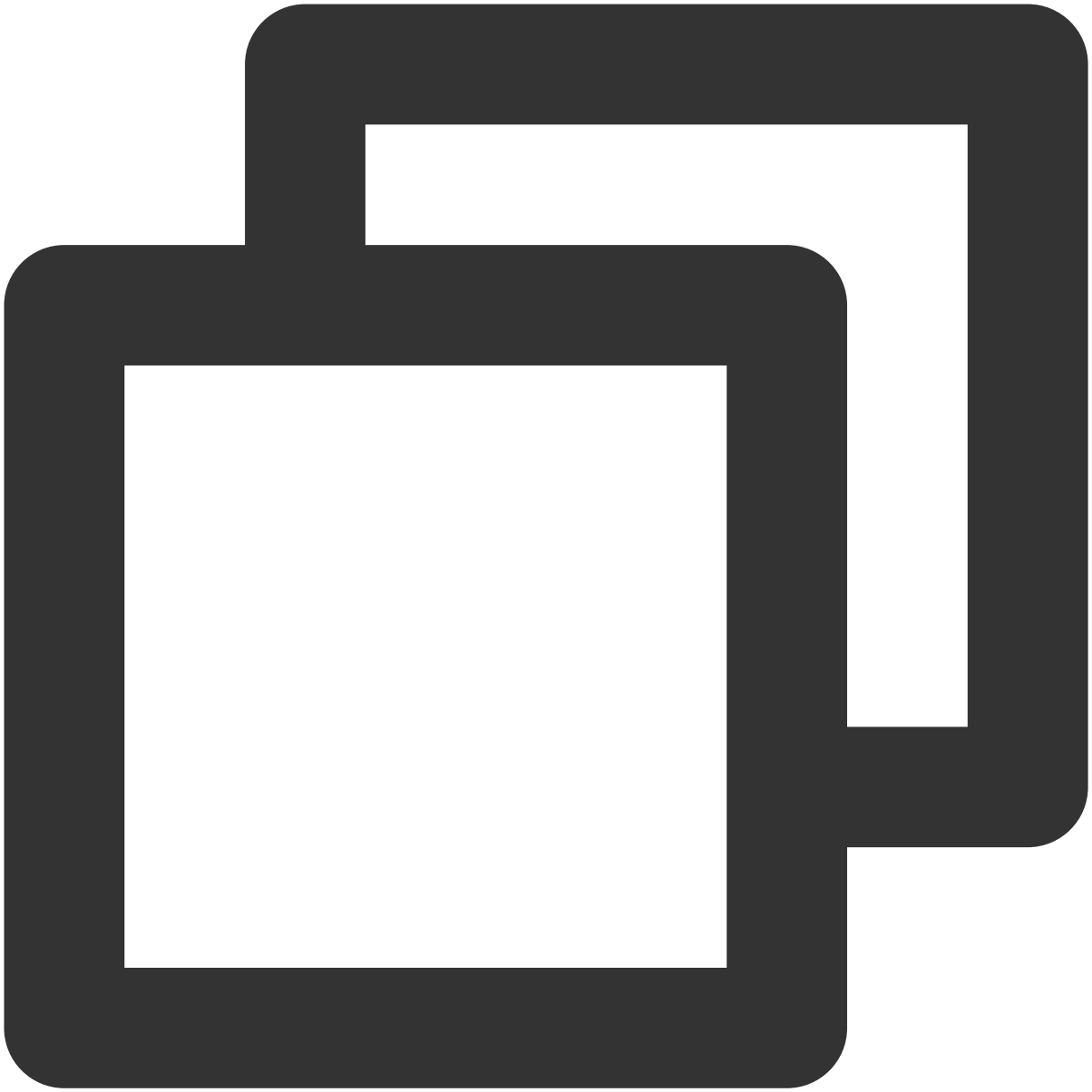
```
ISNOTNULL(<expr> T)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return true if expr is not null; otherwise, return false.

Return type: boolean

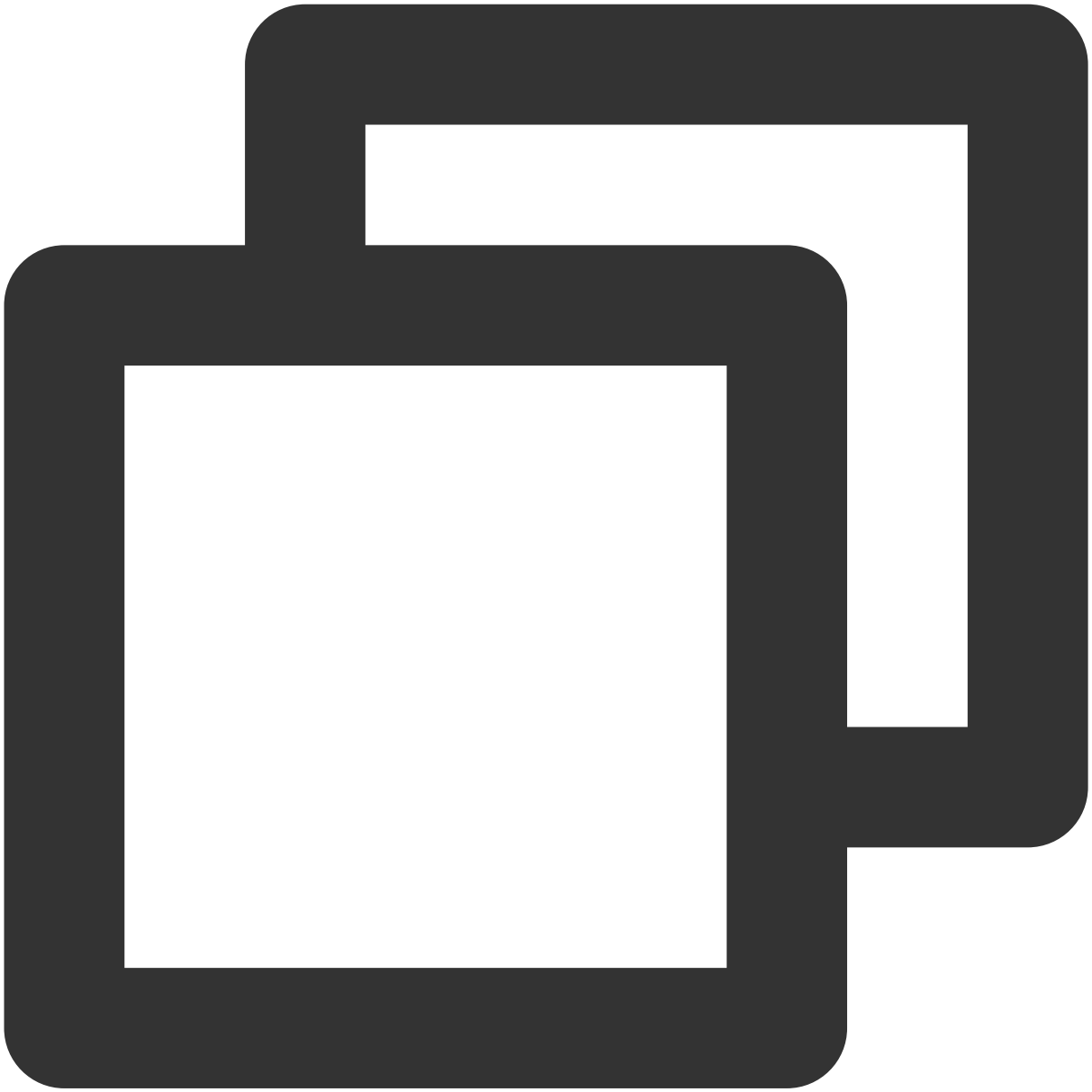
Example:



```
> SELECT isnotnull(1);
 true
```

LEAST

Function statement:



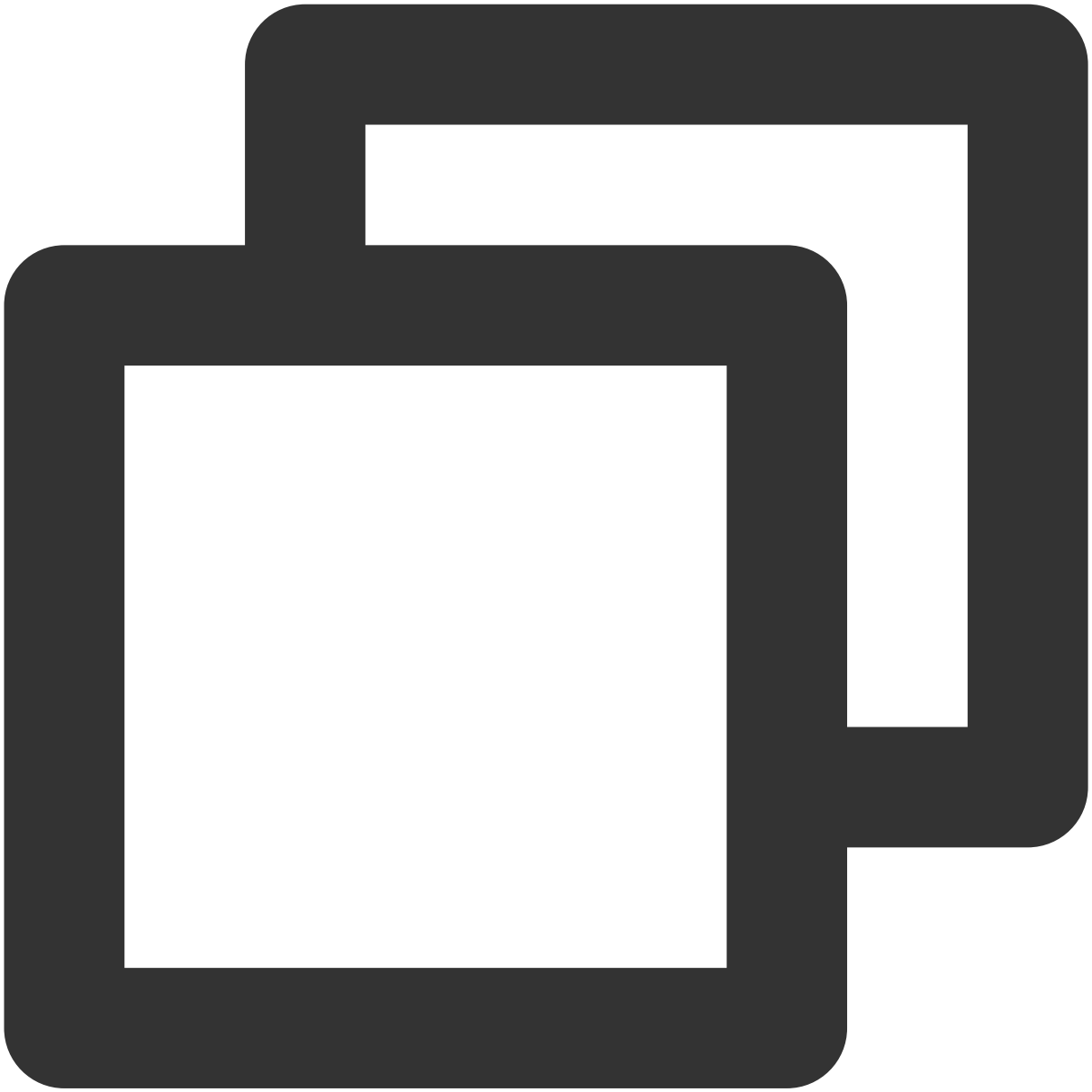
```
LEAST(<expr1> T, <expr2> T, ...)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the minimum value among all parameters, ignoring null values.

Return type: T

Example:



```
> SELECT least(10, 9, 2, 4, 3);
 2
```

NANVL

Function statement:

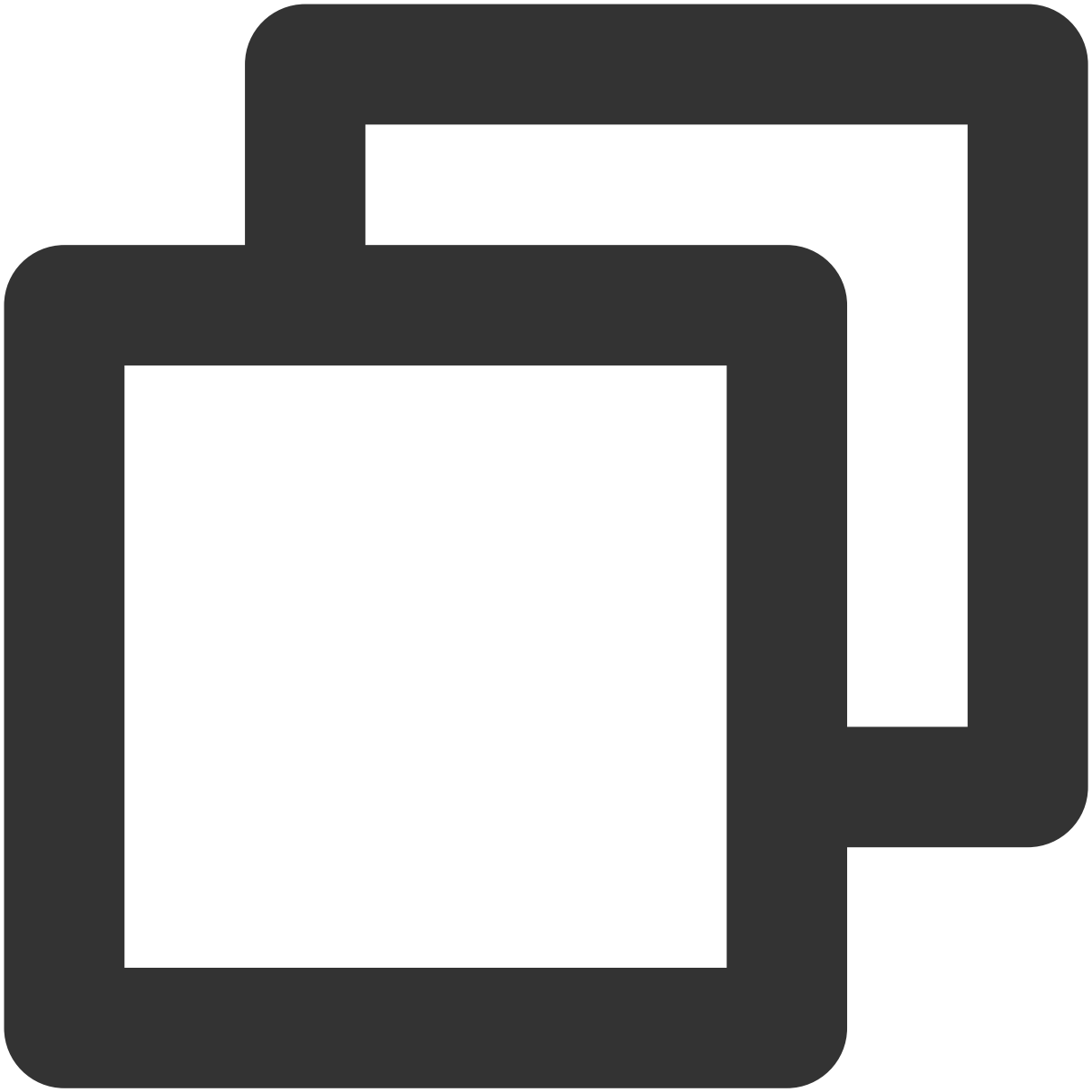
```
NANVL(<expr1> T, <expr2> U)
```

Supported engines: SparkSQL and Presto

Usage instructions: nanvl(expr1, expr2) returns expr1 if expr1 is not NaN; otherwise, it returns expr2.

Return type: T|U

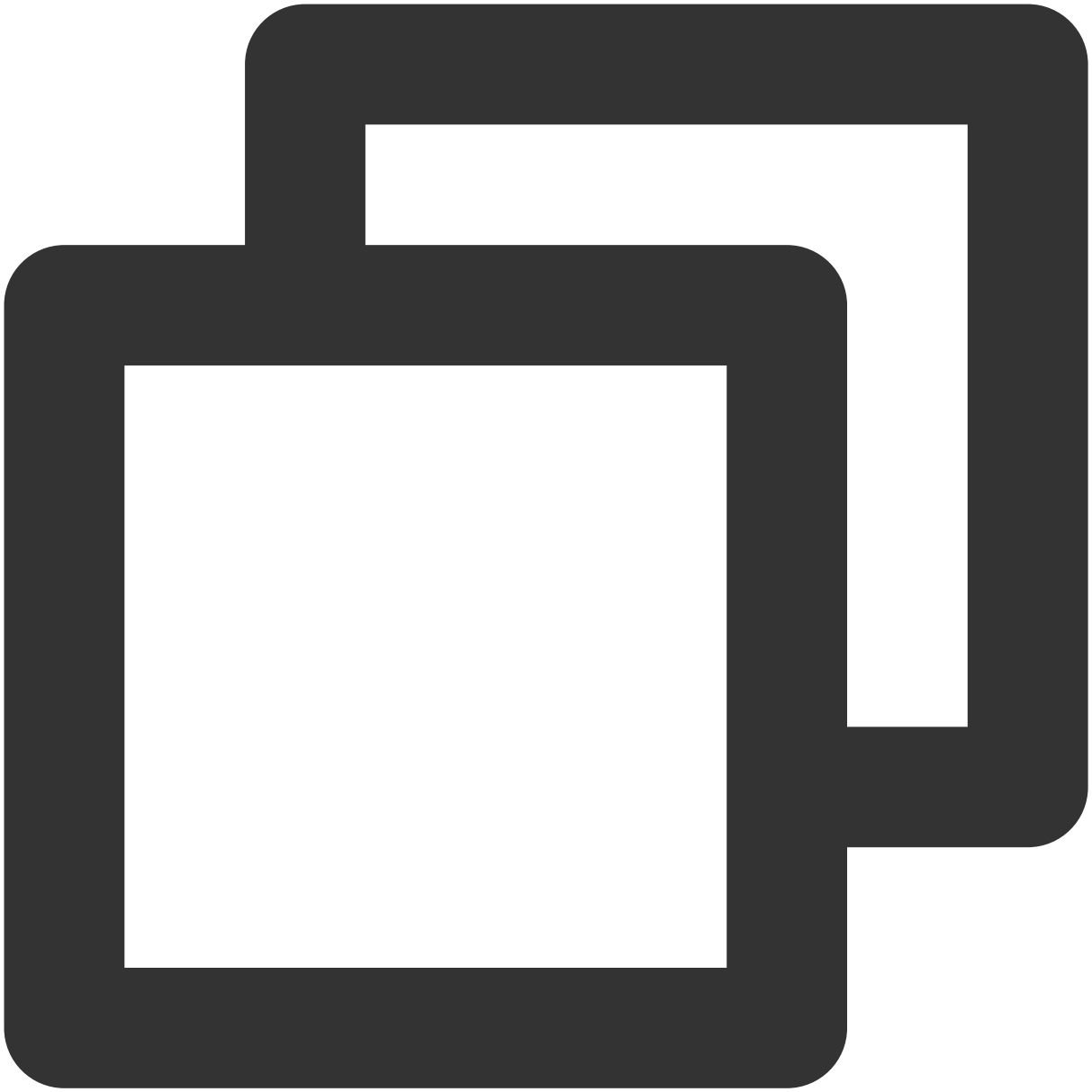
Example:



```
> SELECT nanvl(cast('NaN' as double), 123);
 123.0
```

NULLIF

Function statement:



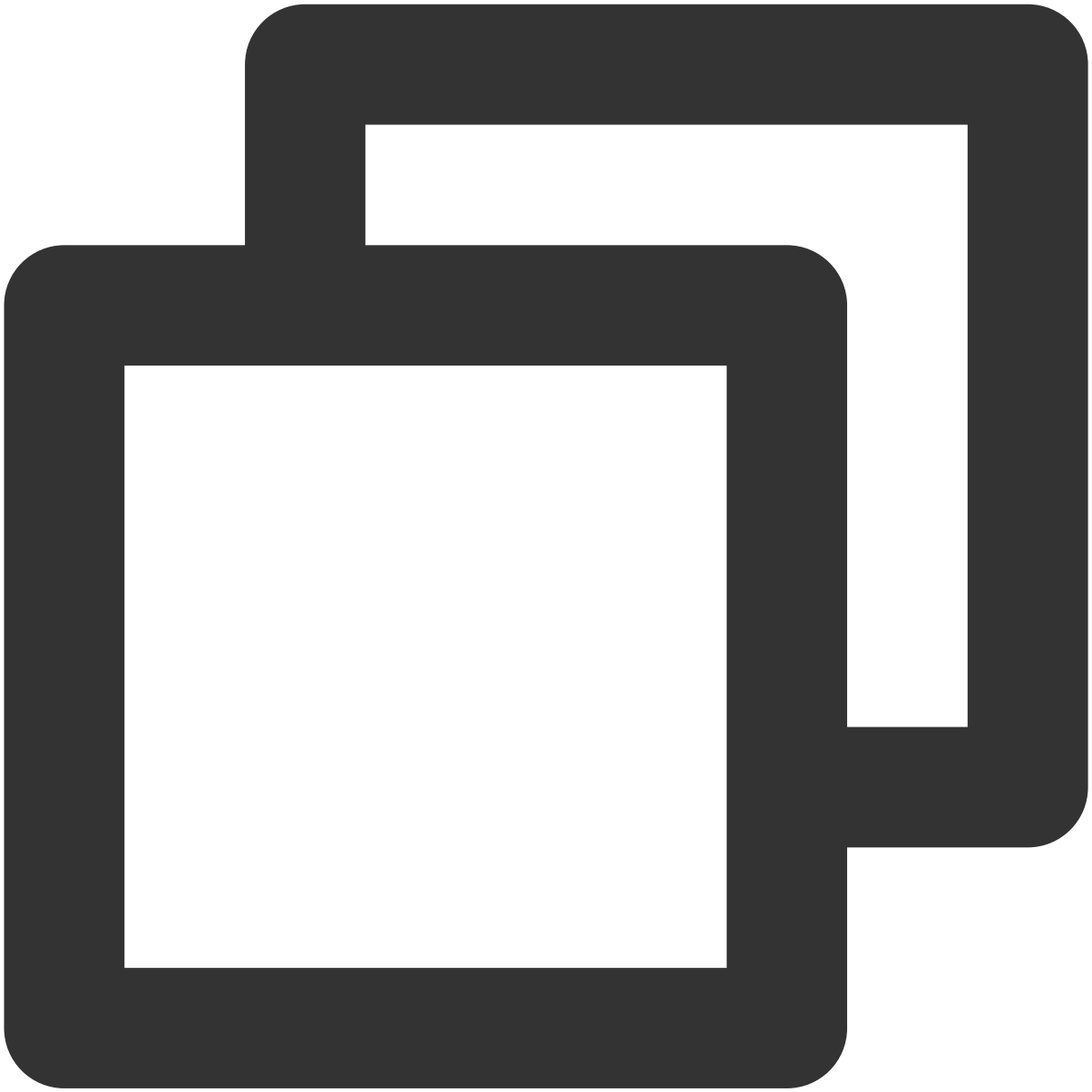
```
NULLIF(<expr1> T, <expr2> U)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return null if expr1 is equal to expr2; otherwise, return expr1.

Return type: T

Example:



```
> SELECT nullif(2, 2);
 NULL
```

NVL

Function statement:

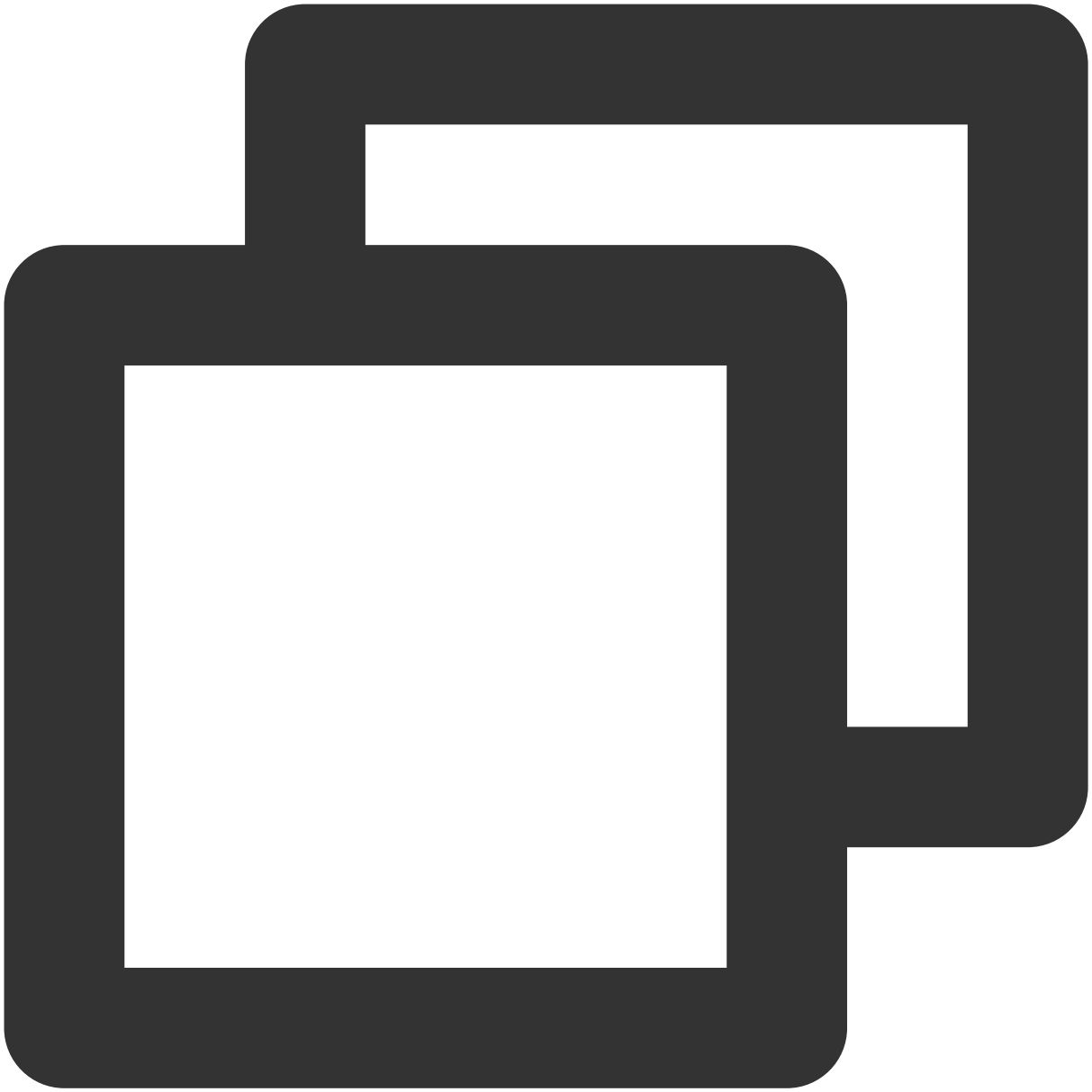
```
NVL(<expr1> T, <expr2> U)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return expr2 if expr1 is null; otherwise, return expr1.

Return type: T|U

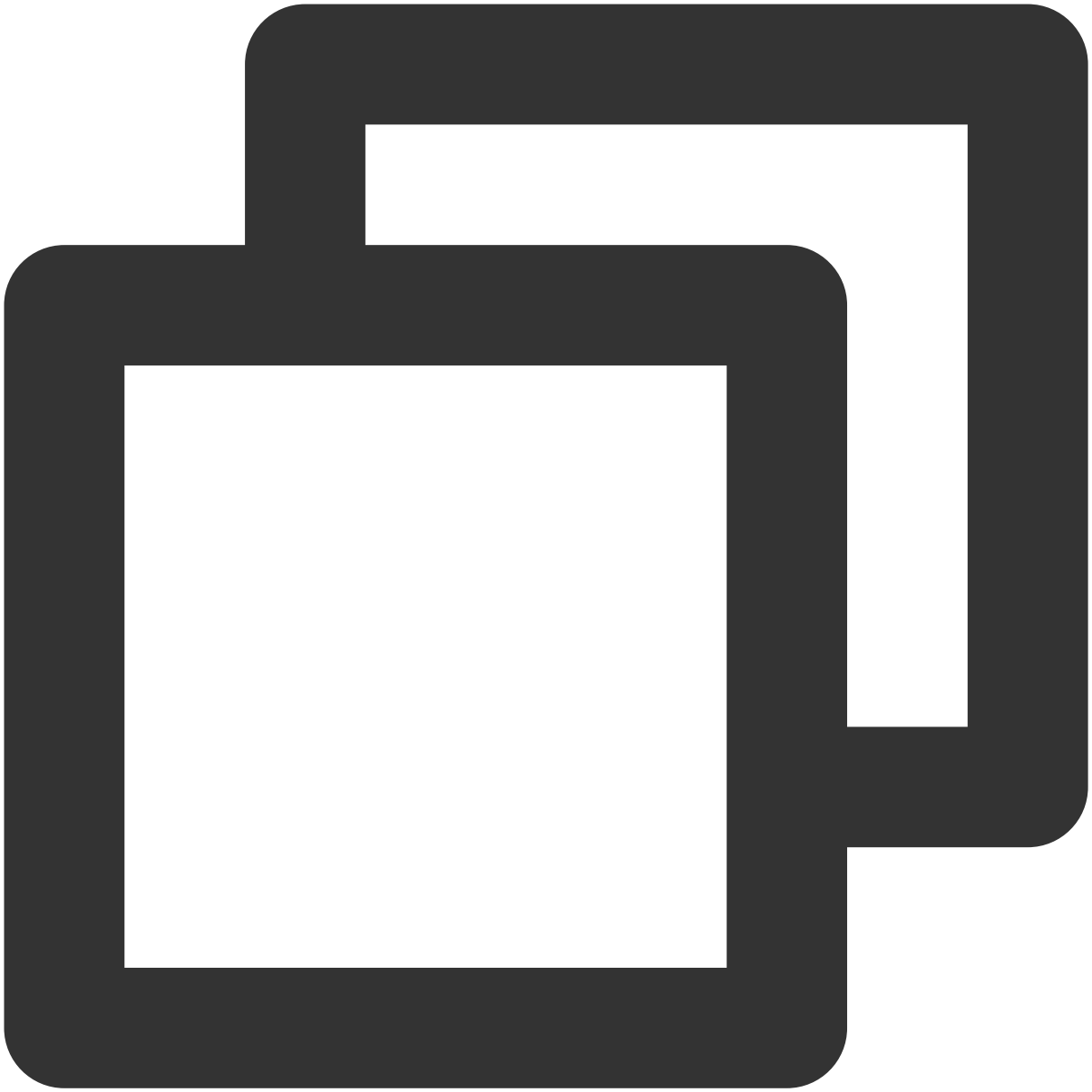
Example:



```
> SELECT nvl(NULL, array('2'));
 ["2"]
```

NVL2

Function statement:



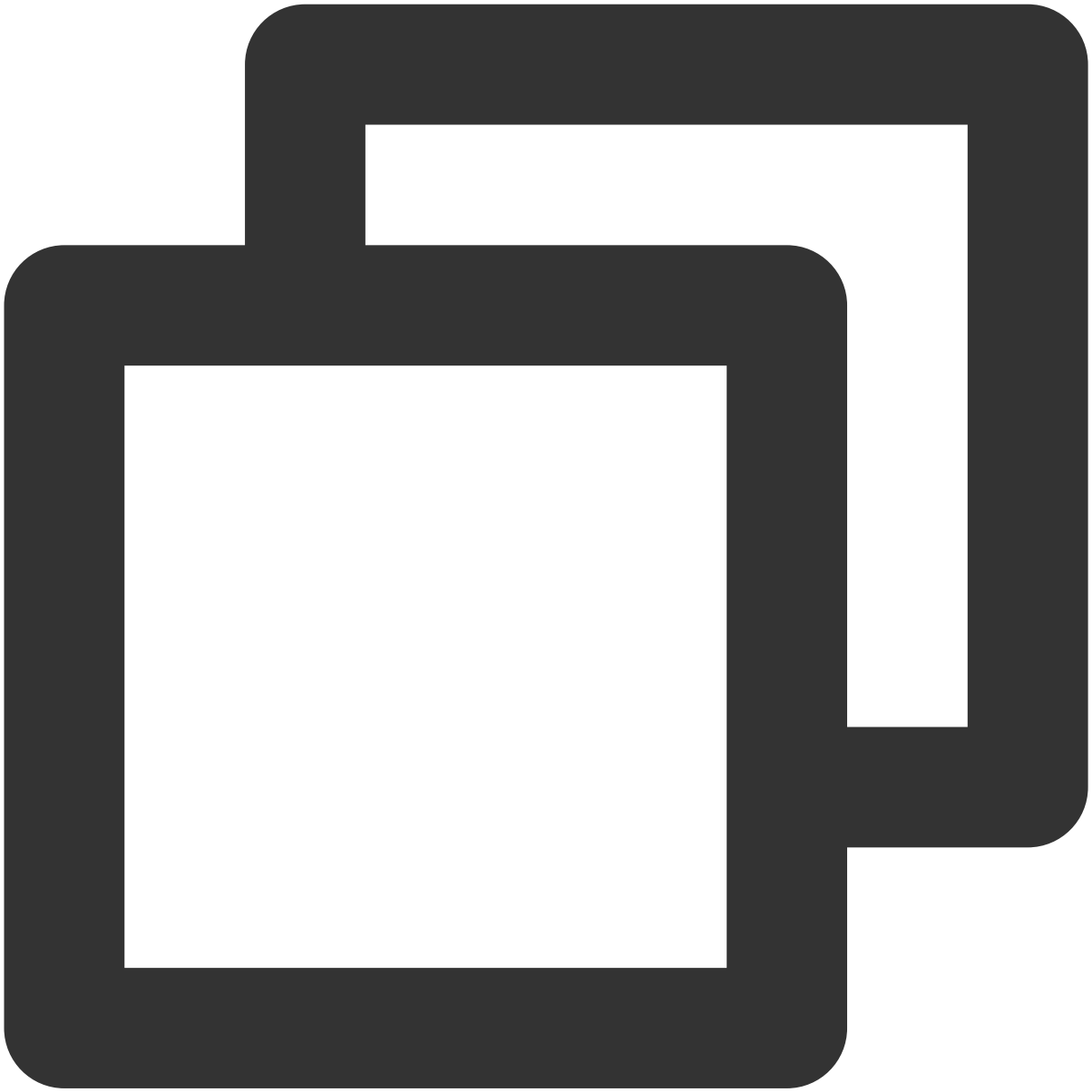
```
NVL2(<expr1> T1, <expr2> T2, <expr3> T3)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return expr2 if expr1 is not null; otherwise, return expr3.

Return type: T2|T3

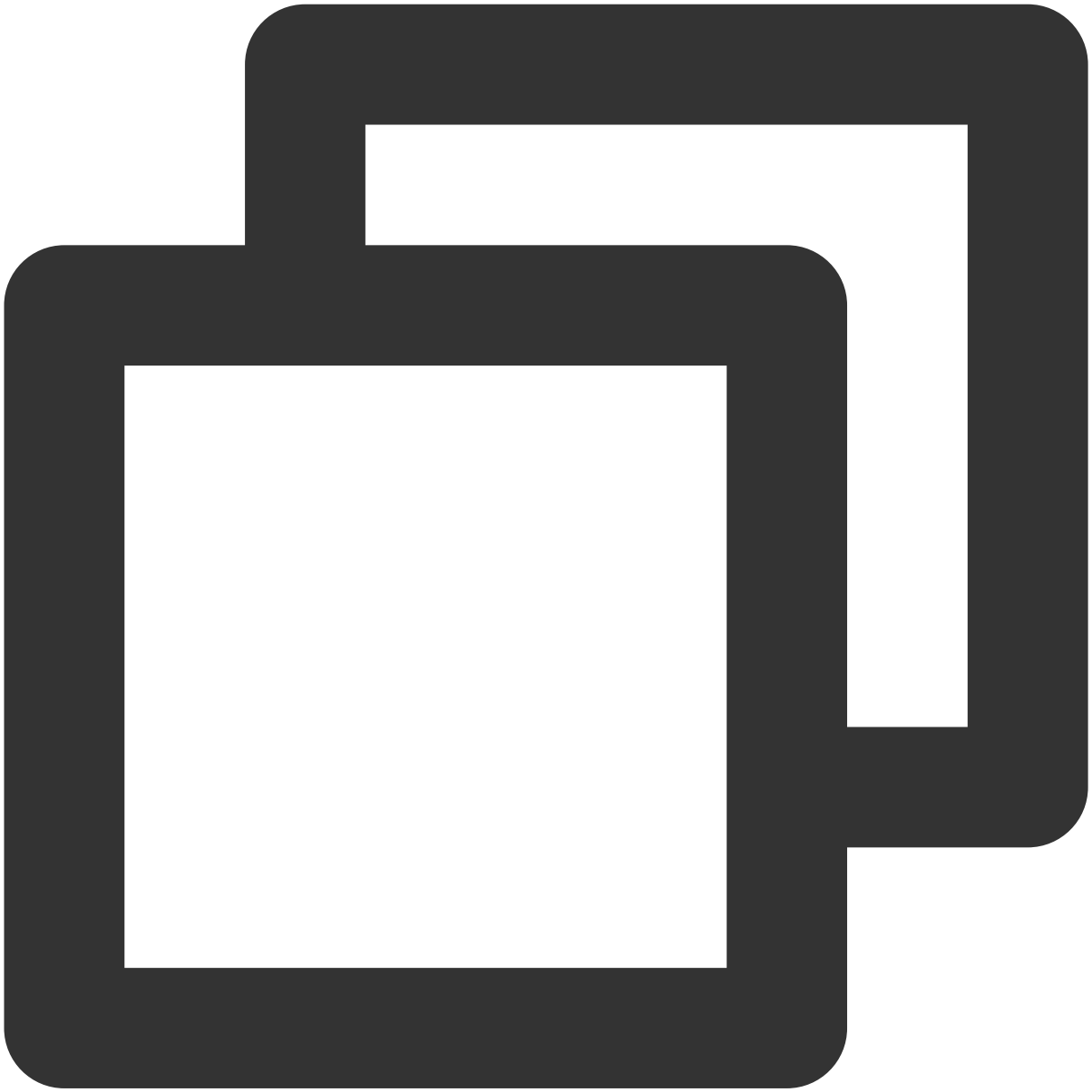
Example:



```
> SELECT nvl2(NULL, 2, 1);
 1
```

POSEXPLODE

Function statement:



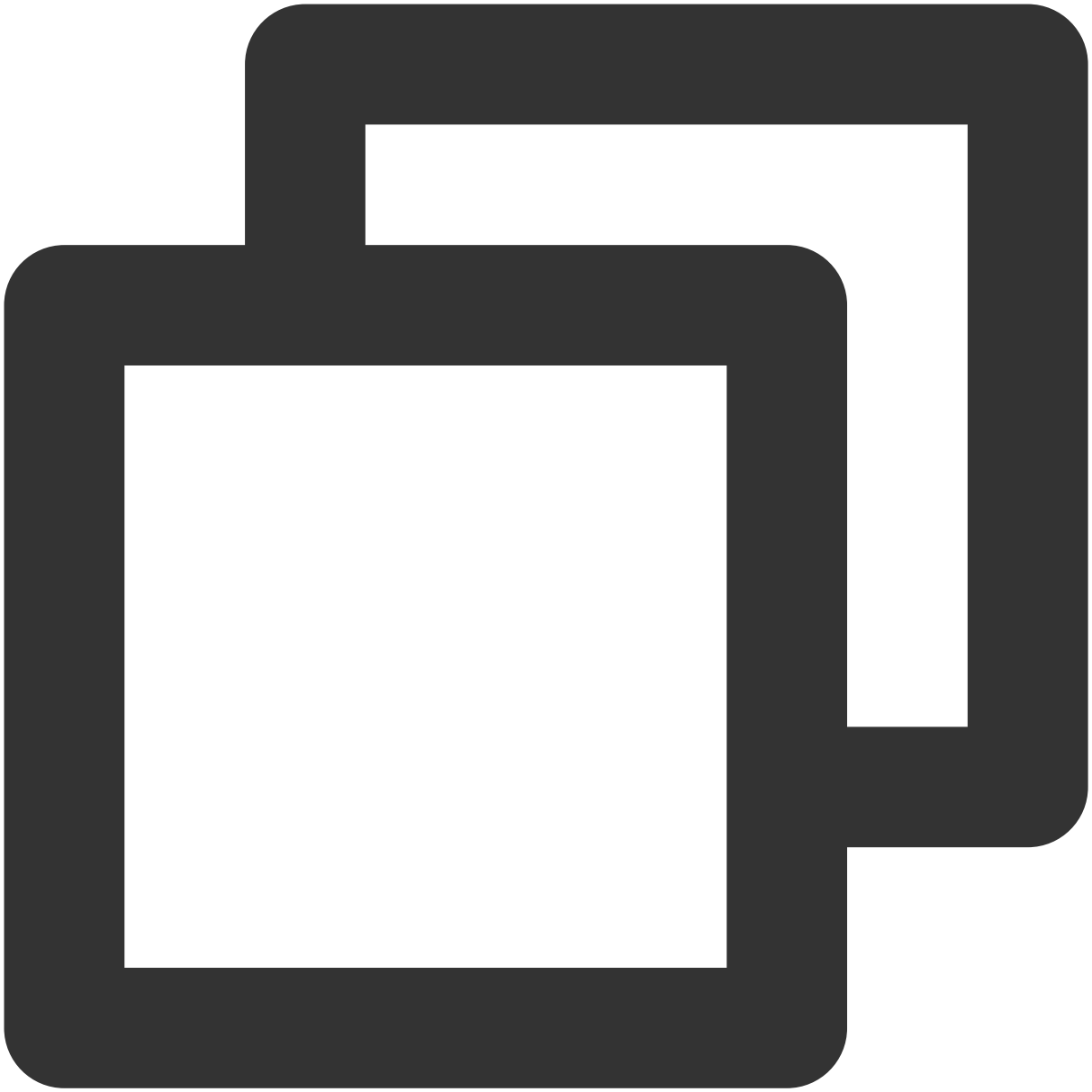
```
POSEXPLODE(<expr> array<T>|map<K, V>)
```

Supported engine: SparkSQL

Usage instructions: Separate the elements of an array-type expr into multiple rows, or separate a map-type expr into multiple rows and columns. Use the column name pos to represent position, with the default column name col for array elements, or key and value for map elements.

Return type: row(pos integer, col T)|row(row integer, key K, value V)

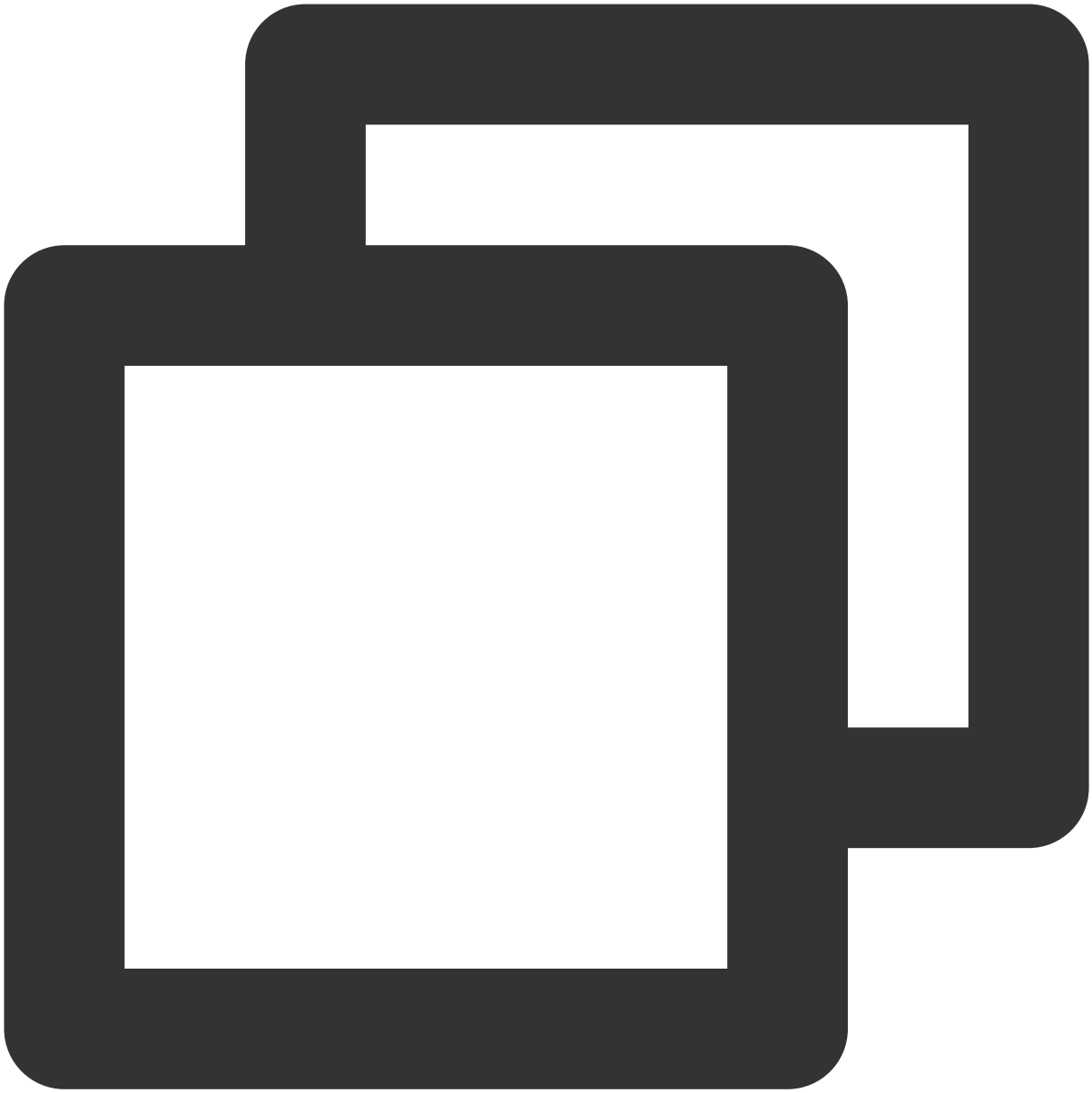
Example:



```
> SELECT posexplode(array(10,20));
 0  10
 1  20
```

POSEXPLODE_OUTER

Function statement:



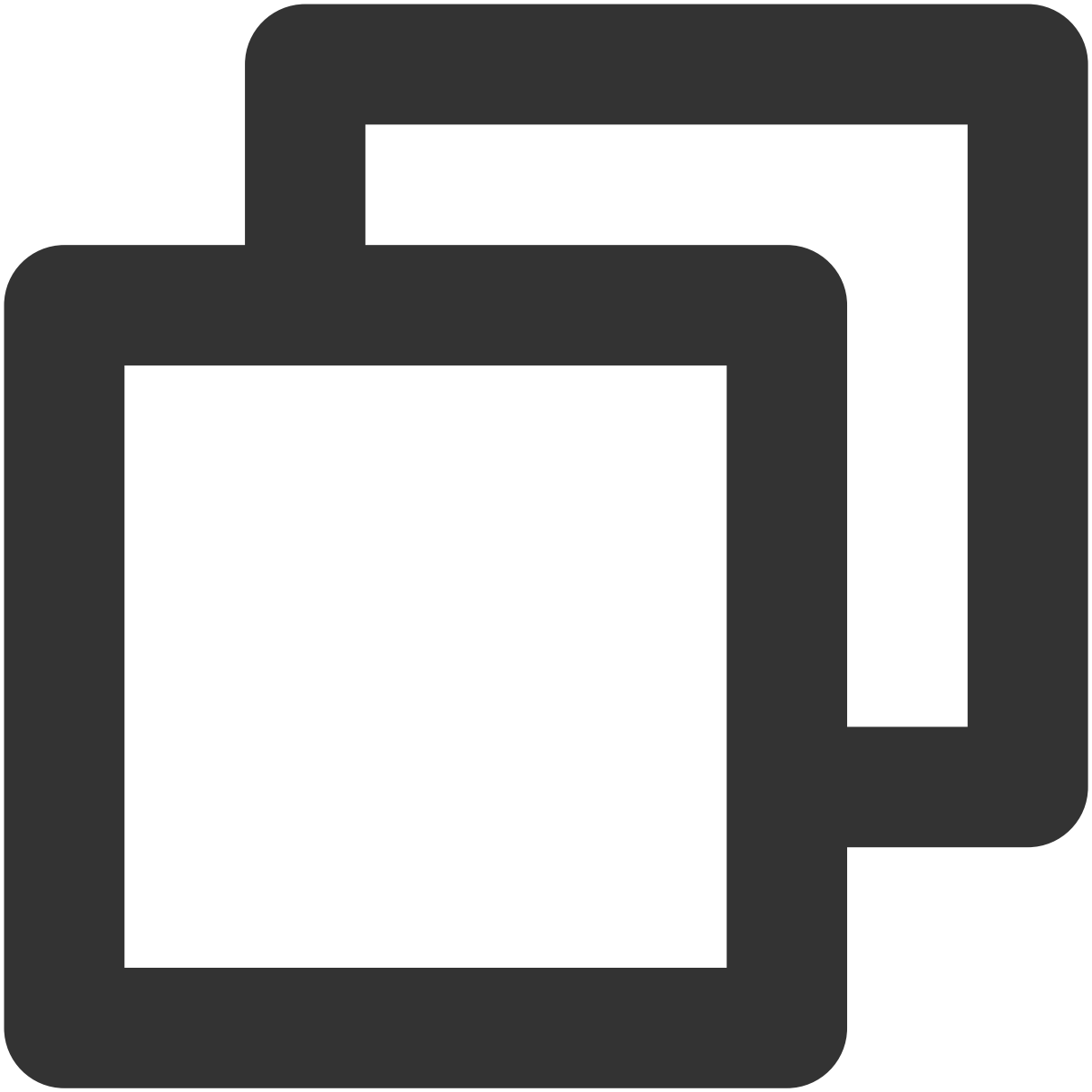
```
POSEXPLODE_OUTER(<expr> array<T>|map<K, V>)
```

Supported engine: SparkSQL

Usage instructions: Separate the elements of an array-type expr into multiple rows, or separate a map-type expr into multiple rows and columns. Use the column name pos to represent position, with the default column name col for array elements, or key and value for map elements.

Return type: row(pos integer, col T)|row(row integer, key K, value V)

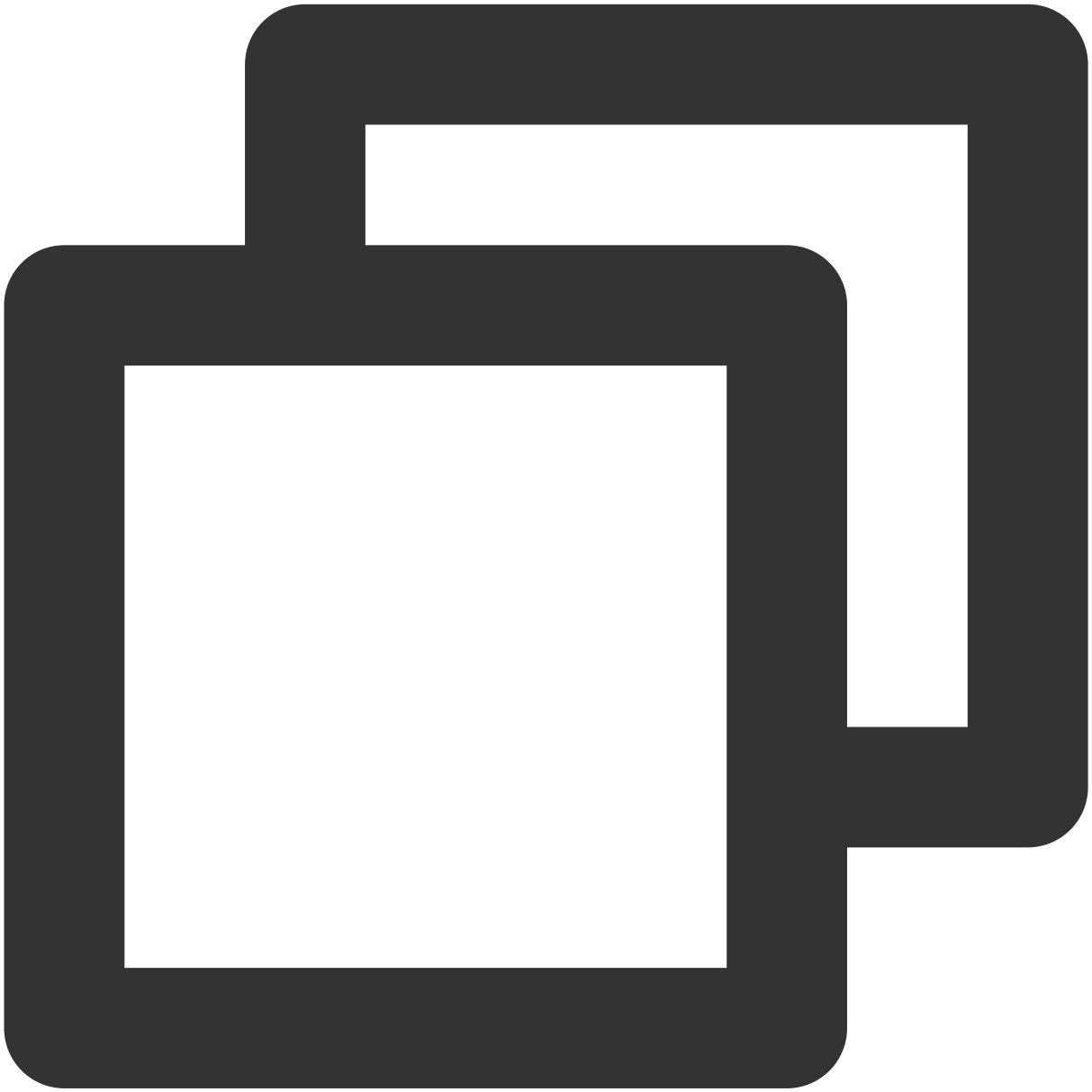
Example:



```
> SELECT posexplode_outer(array(10,20));
 0  10
 1  20
```

STACK

Function statement:



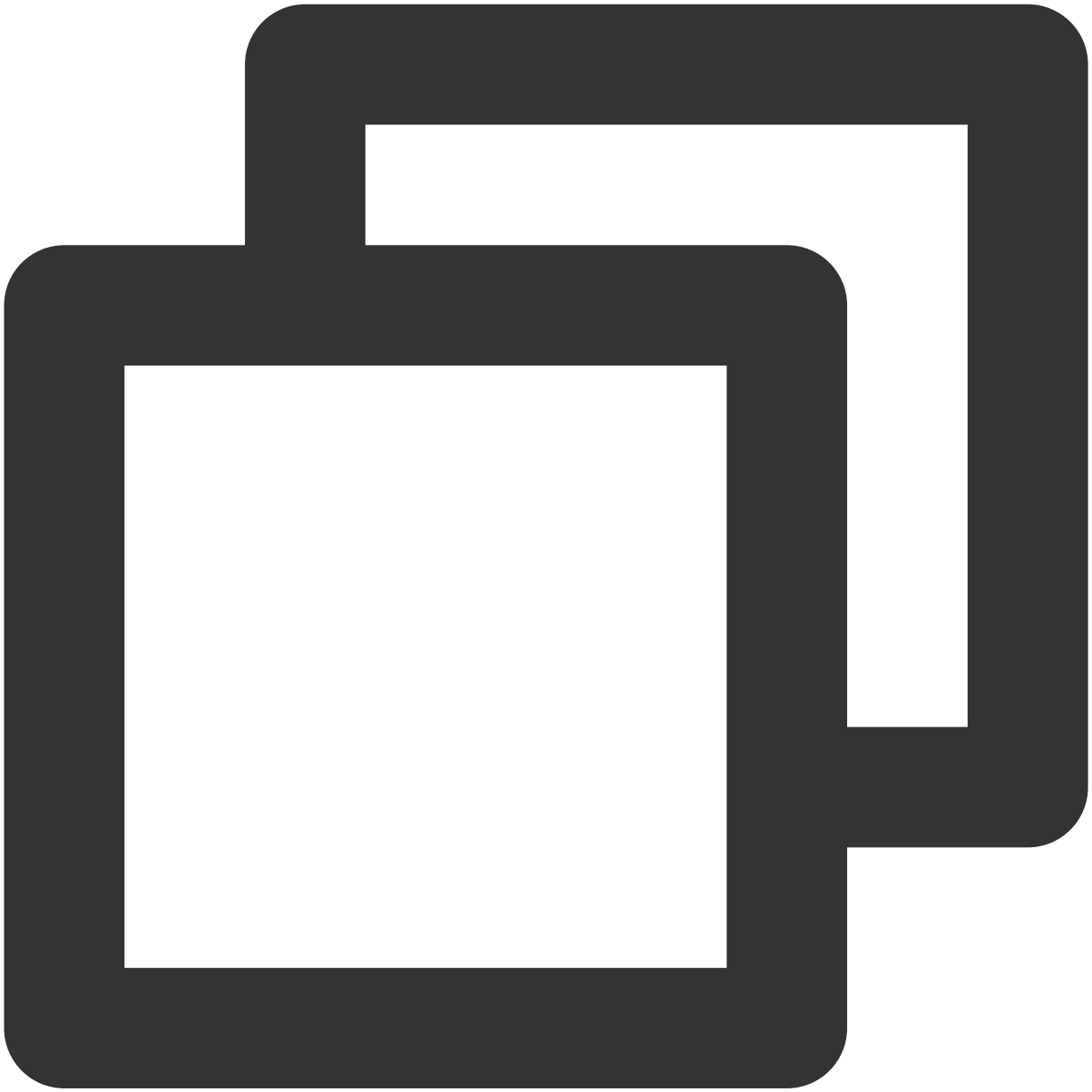
```
STACK(<n> integer, <expr0> T0, ..., <expr1> T1)
```

Supported engine: SparkSQL

Usage instructions: stack(n, expr1, ..., exprk) - Separate expr1, ..., exprk into n rows. By default, col0, col1, etc. are used as column names.

Return type: row(col0 T0, ..., coln Tn)

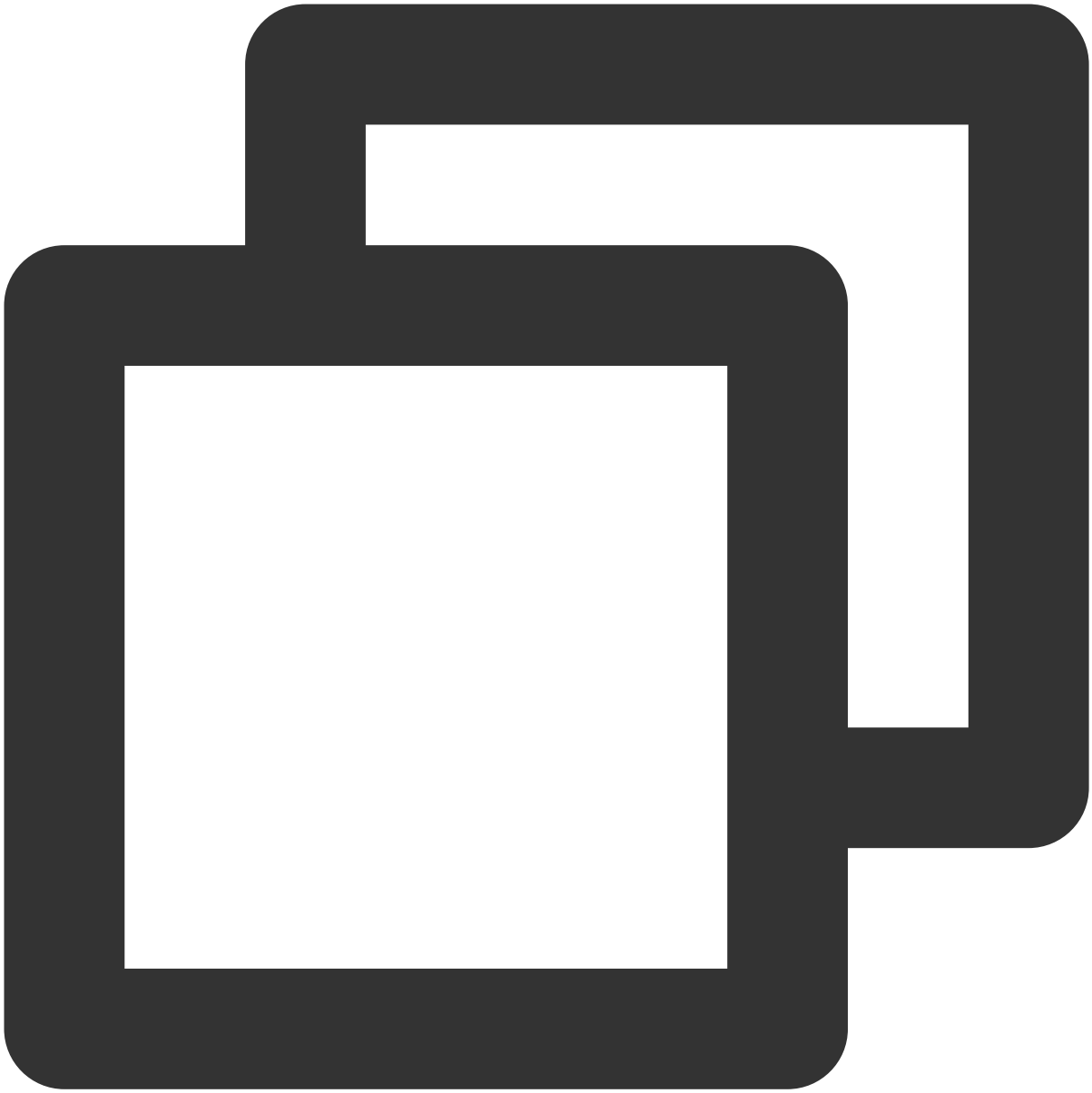
Example:



```
> SELECT stack(2, 1, 2, 3);
 1  2
 3  NULL
```

ASSERT_TRUE

Function statement:



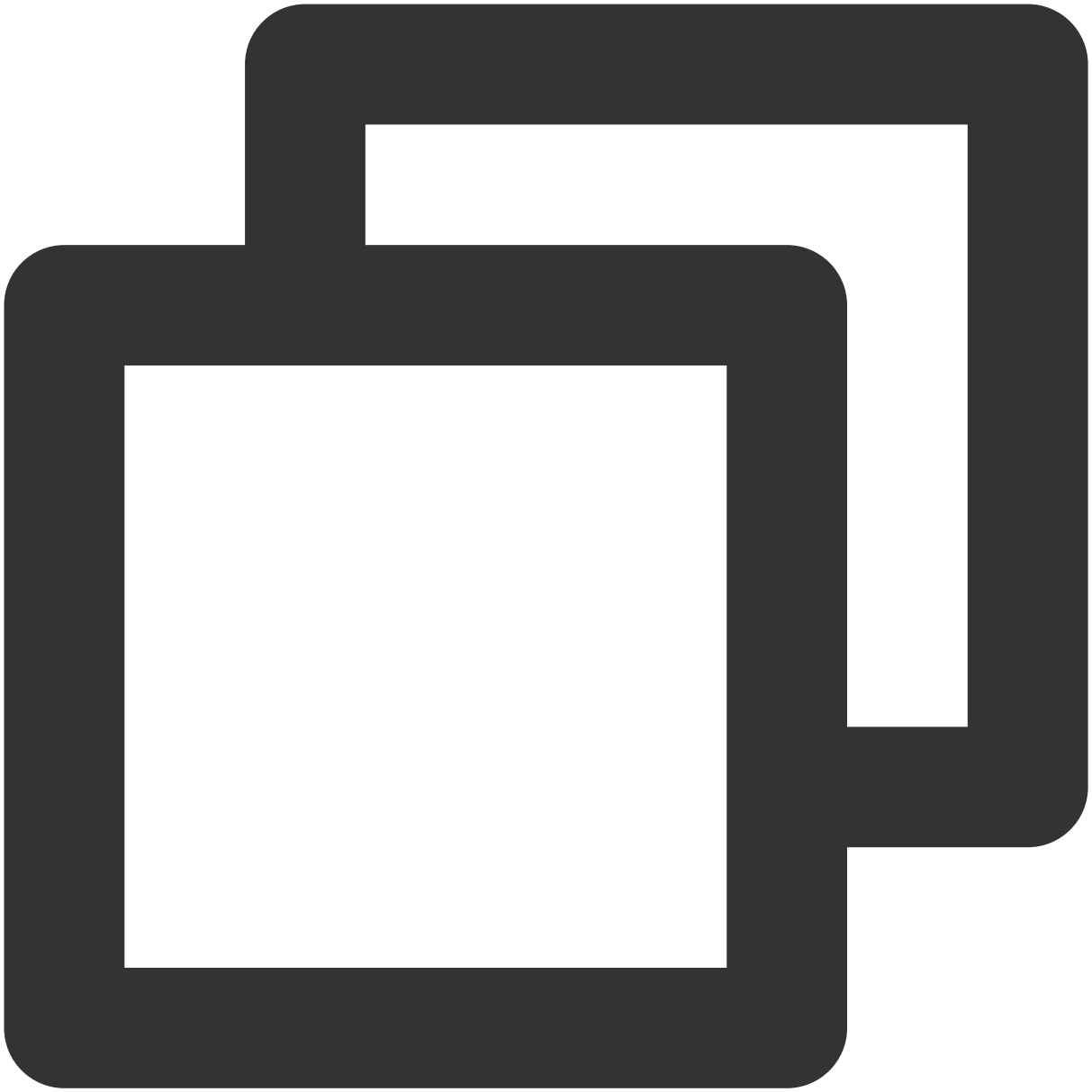
```
ASSERT_TRUE(<expr> boolean)
```

Supported engines: SparkSQL and Presto

Usage instructions: Throw an exception if expr is not true.

Return type: boolean

Example:



```
> SELECT assert_true(0 < 1);
 NULL
```

RAISE_ERROR

Function statement:

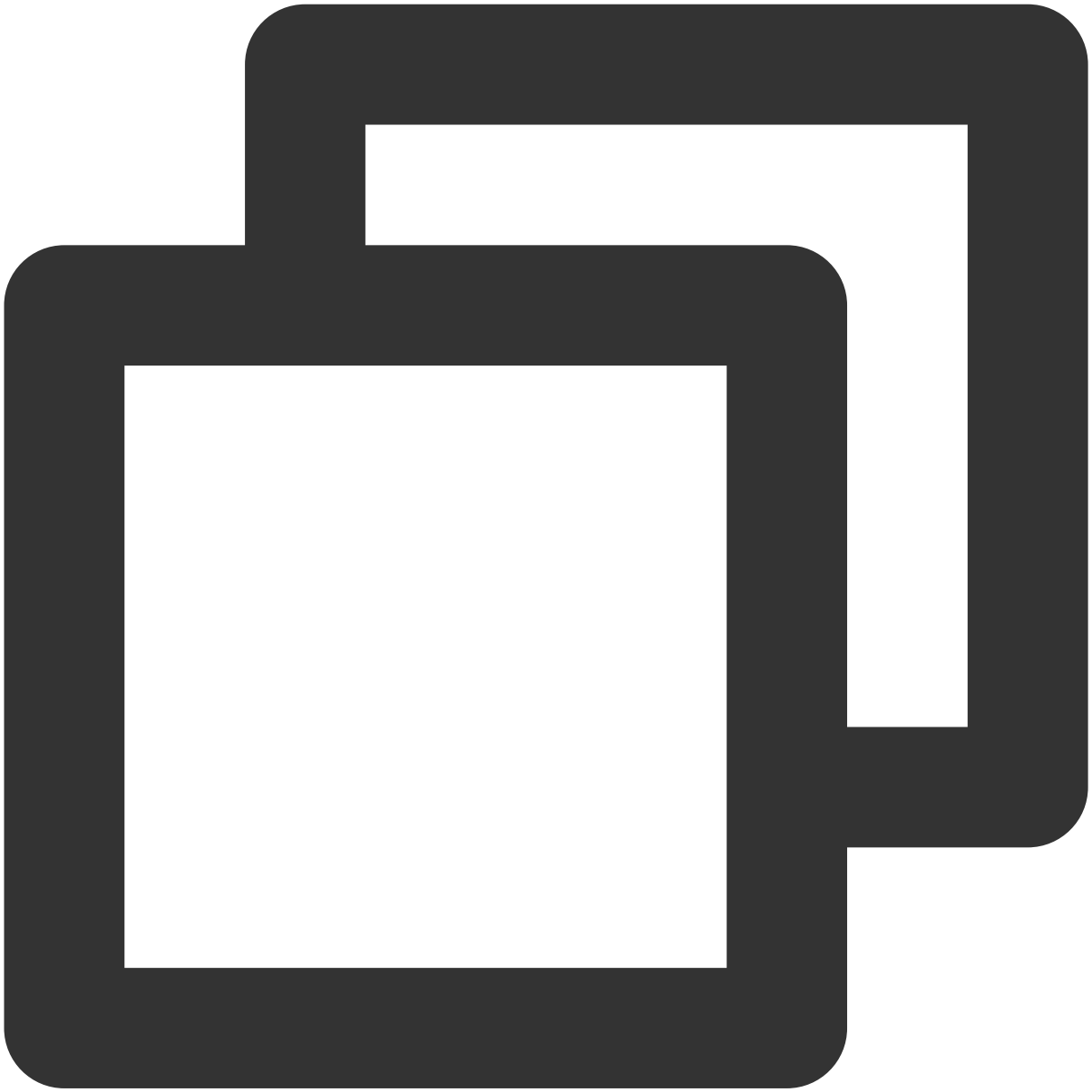
```
RAISE_ERROR(<error> string)
```

Supported engines: SparkSQL and Presto

Usage instructions: Throw an exception of expr.

Return type: string

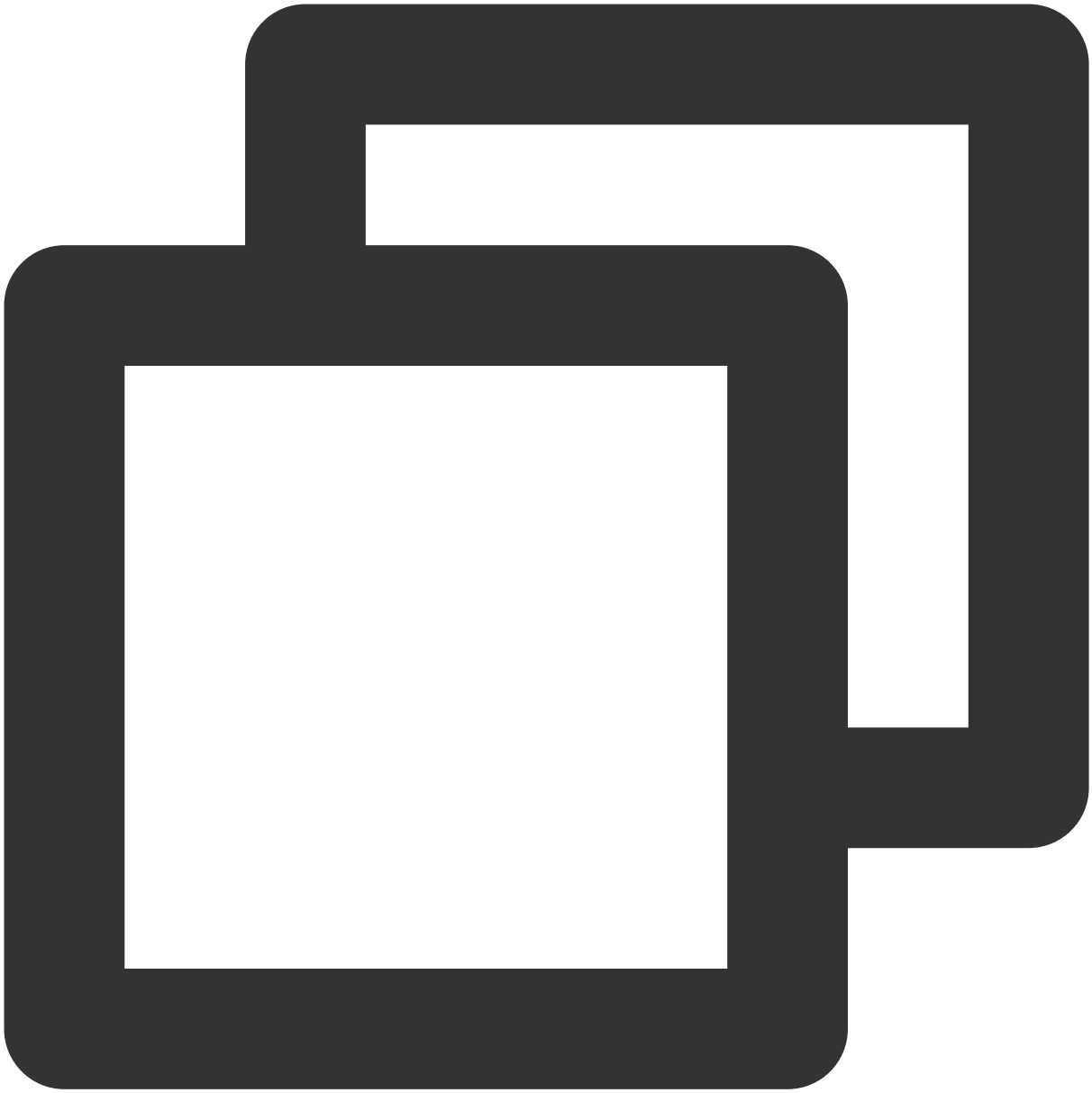
Example:



```
> SELECT raise_error('custom error message');
 java.lang.RuntimeException
 custom error message
```

SPARK_PARTITION_ID

Function statement:



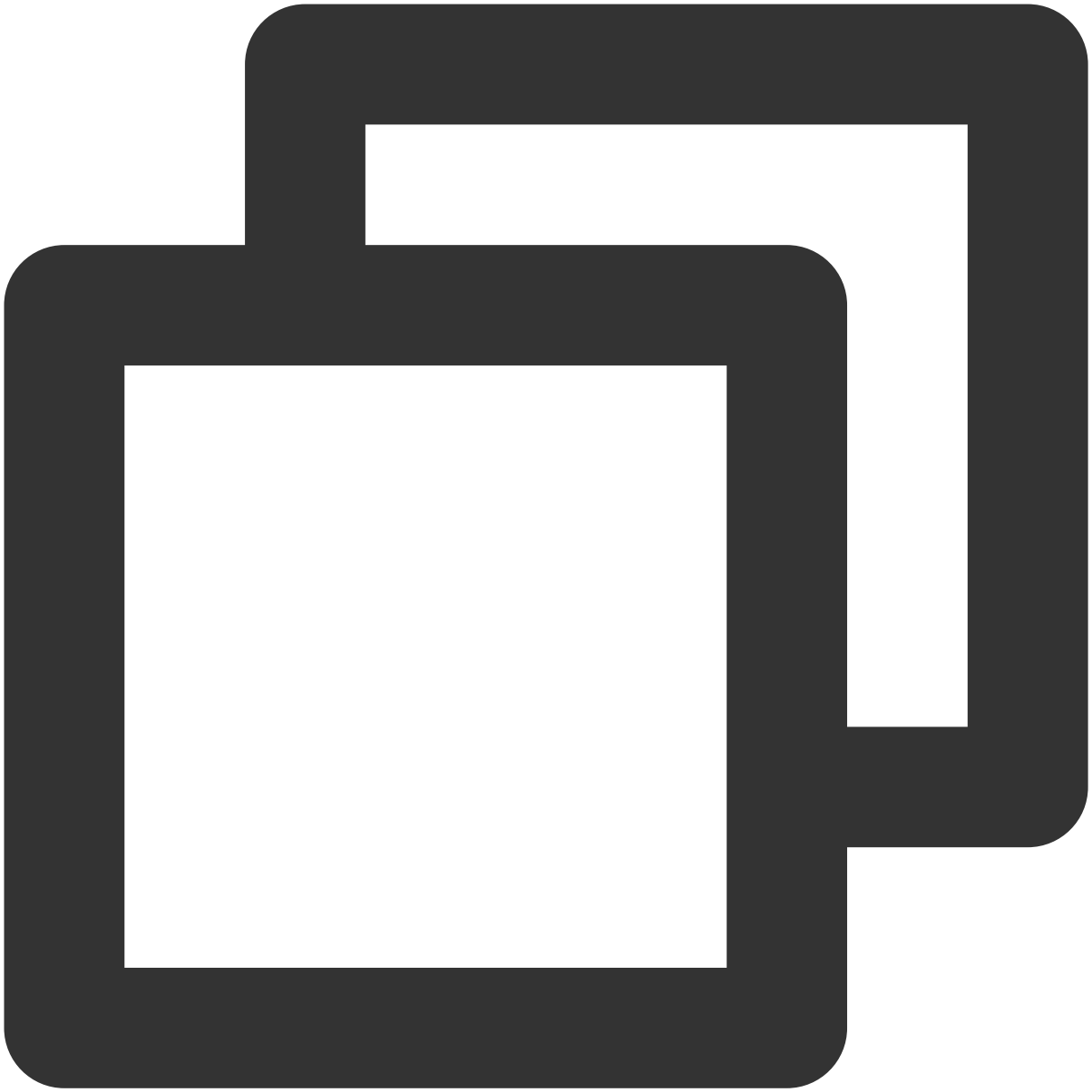
```
SPARK_PARTITION_ID()
```

Supported engine: SparkSQL

Usage instructions: Return the current partition ID.

Return type: integer

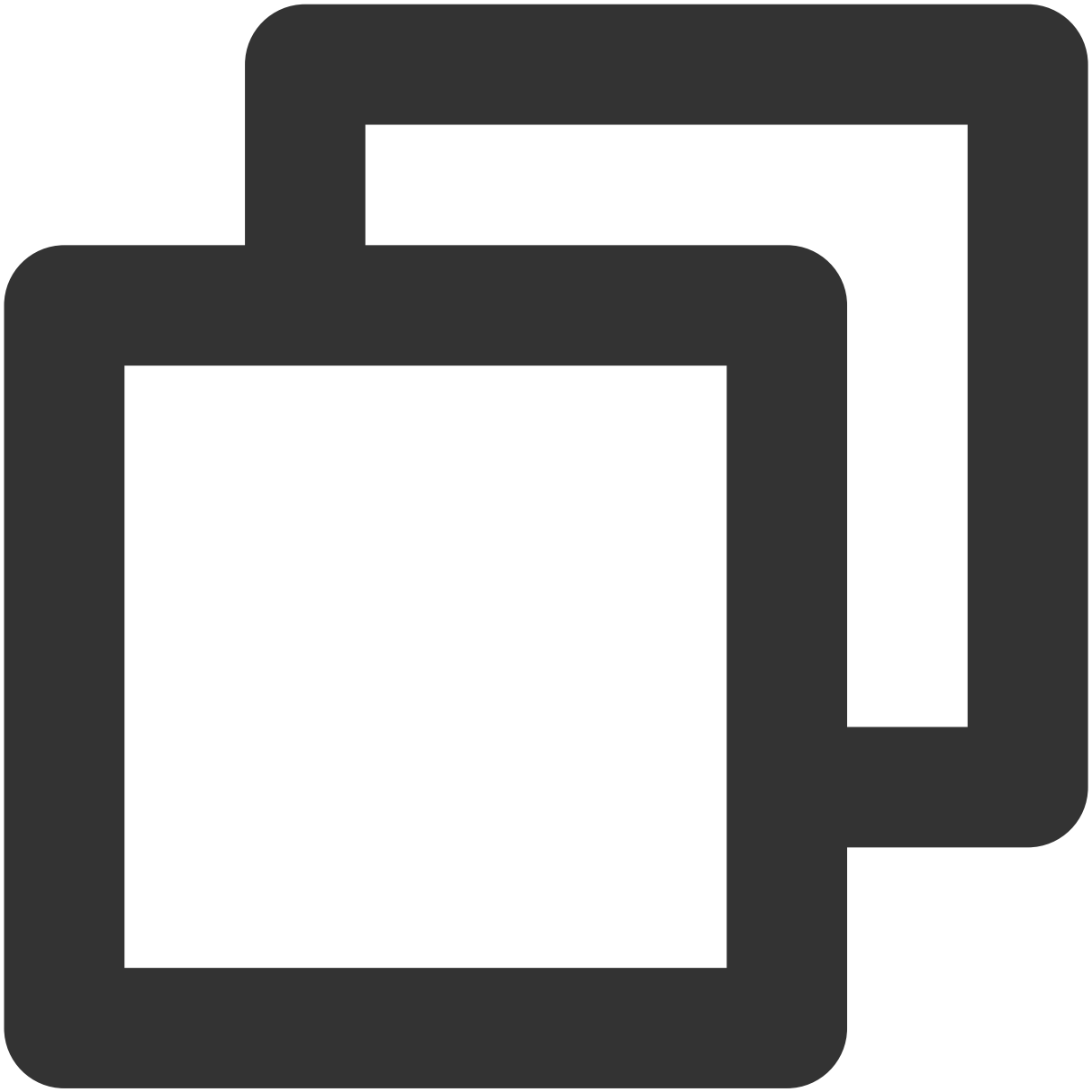
Example:



```
> SELECT spark_partition_id();
 0
```

INPUT_FILE_NAME

Function statement:



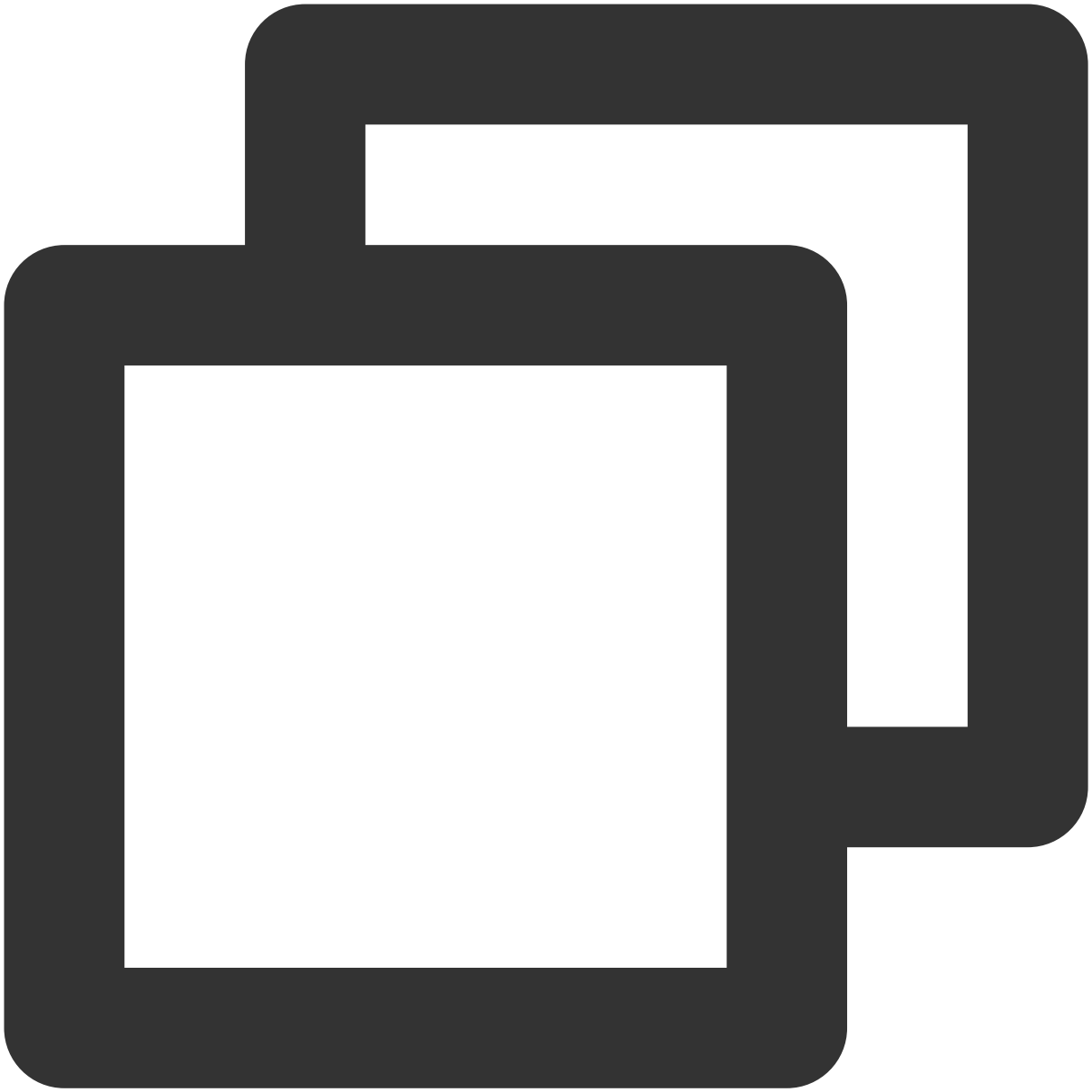
```
INPUT_FILE_NAME()
```

Supported engine: SparkSQL

Usage instructions: Return the name of the file being read. If the file name is unavailable, return a null string.

Return type: string

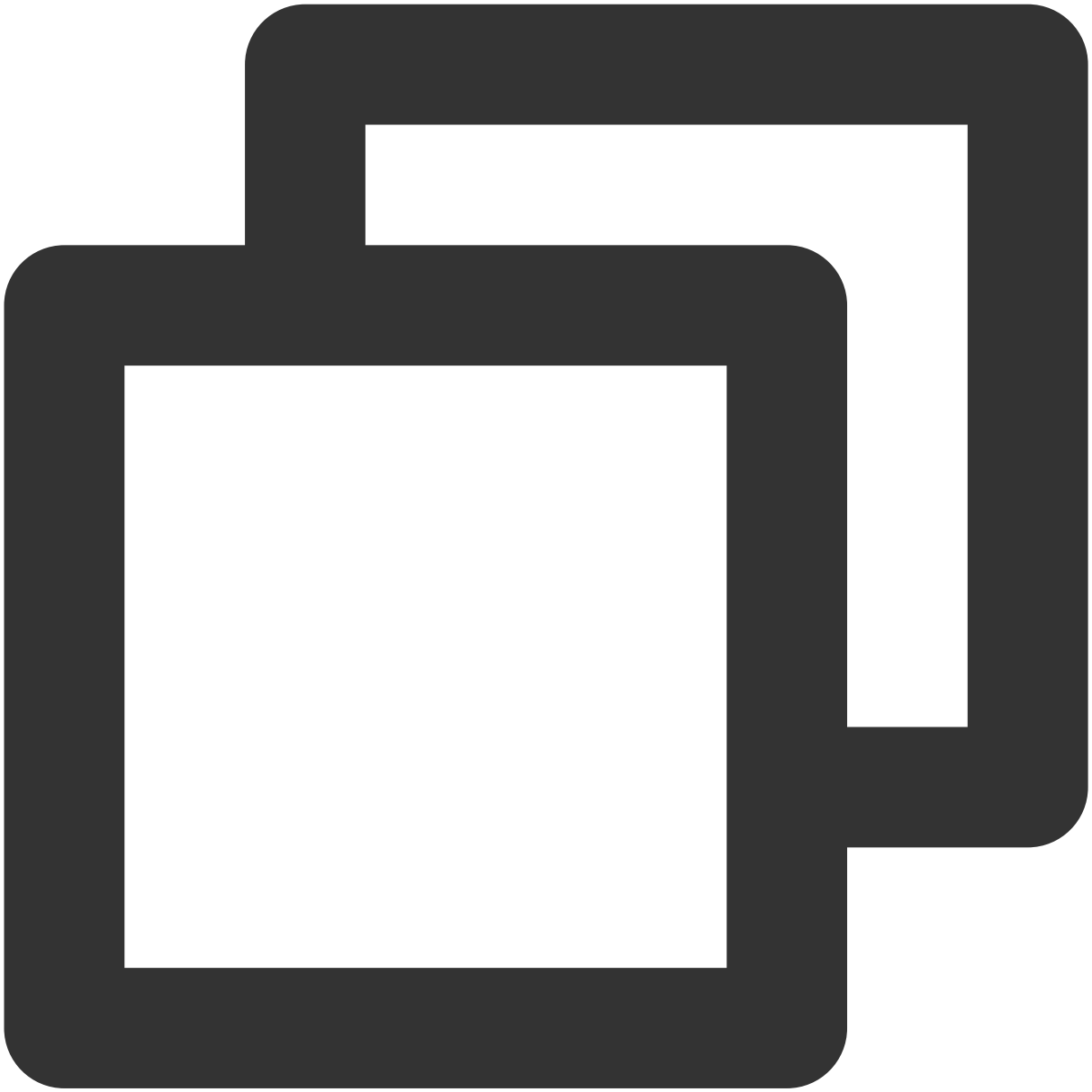
Example:



```
> SELECT input_file_name();
```

INPUT_FILE_BLOCK_START

Function statement:



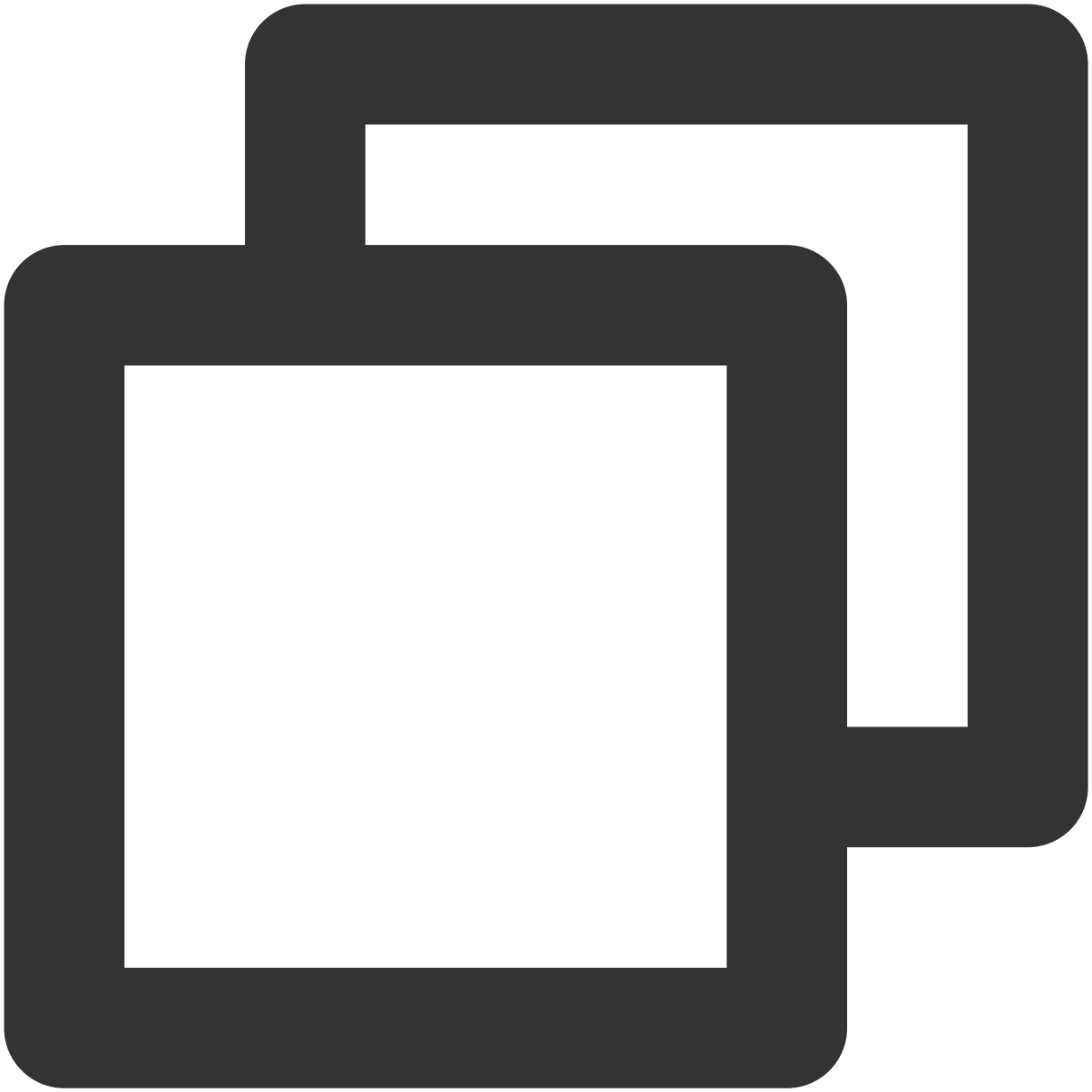
```
INPUT_FILE_BLOCK_START()
```

Supported engine: SparkSQL

Usage instructions: Return the starting offset of the block being read. If the offset is unavailable, return -1.

Return type: integer

Example:



```
> SELECT input_file_block_start();
 -1
```

INPUT_FILE_BLOCK_LENGTH

Function statement:

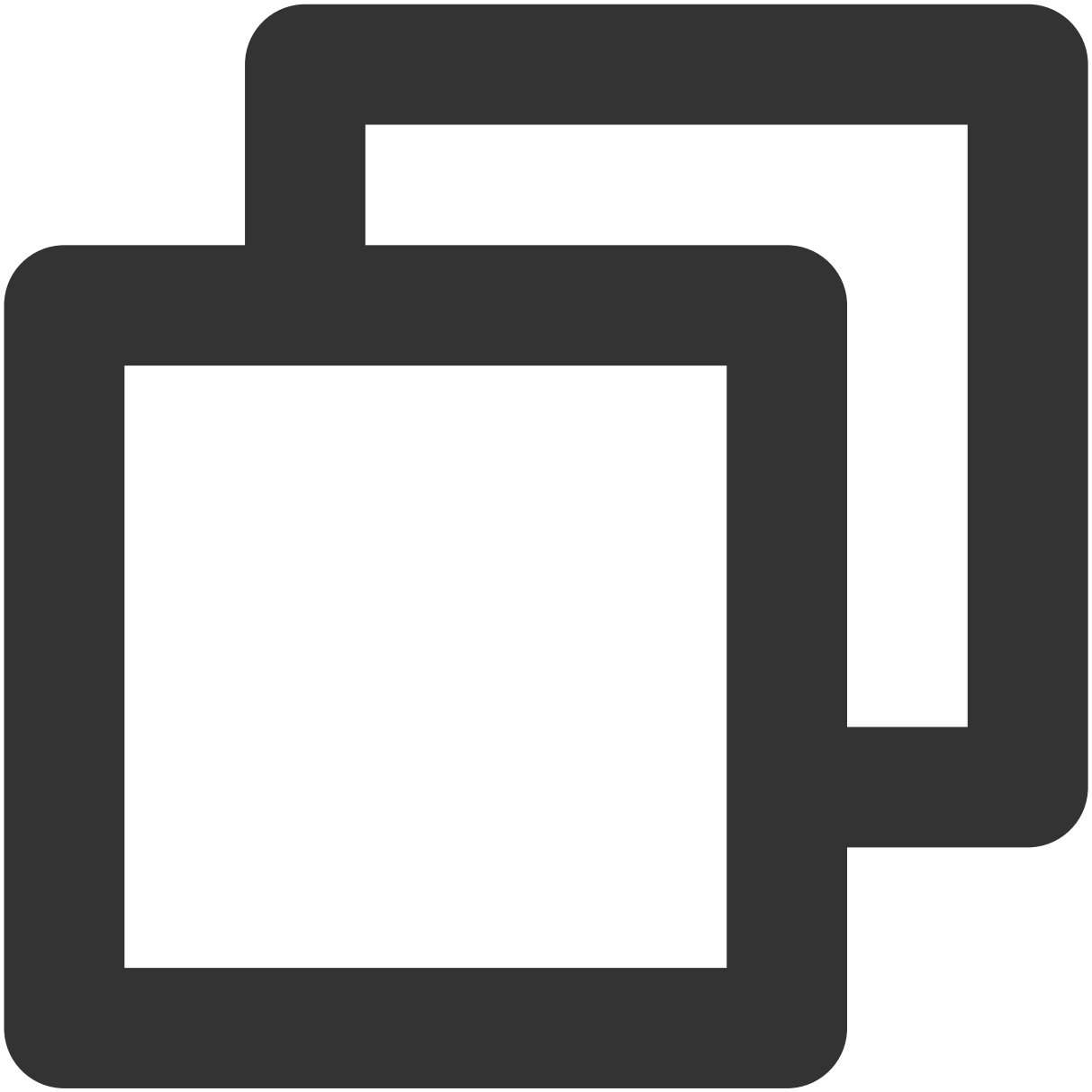
```
INPUT_FILE_BLOCK_LENGTH()
```

Supported engine: SparkSQL

Usage instructions: Return the length of the block being read. If the length is unavailable, return -1.

Return type: integer

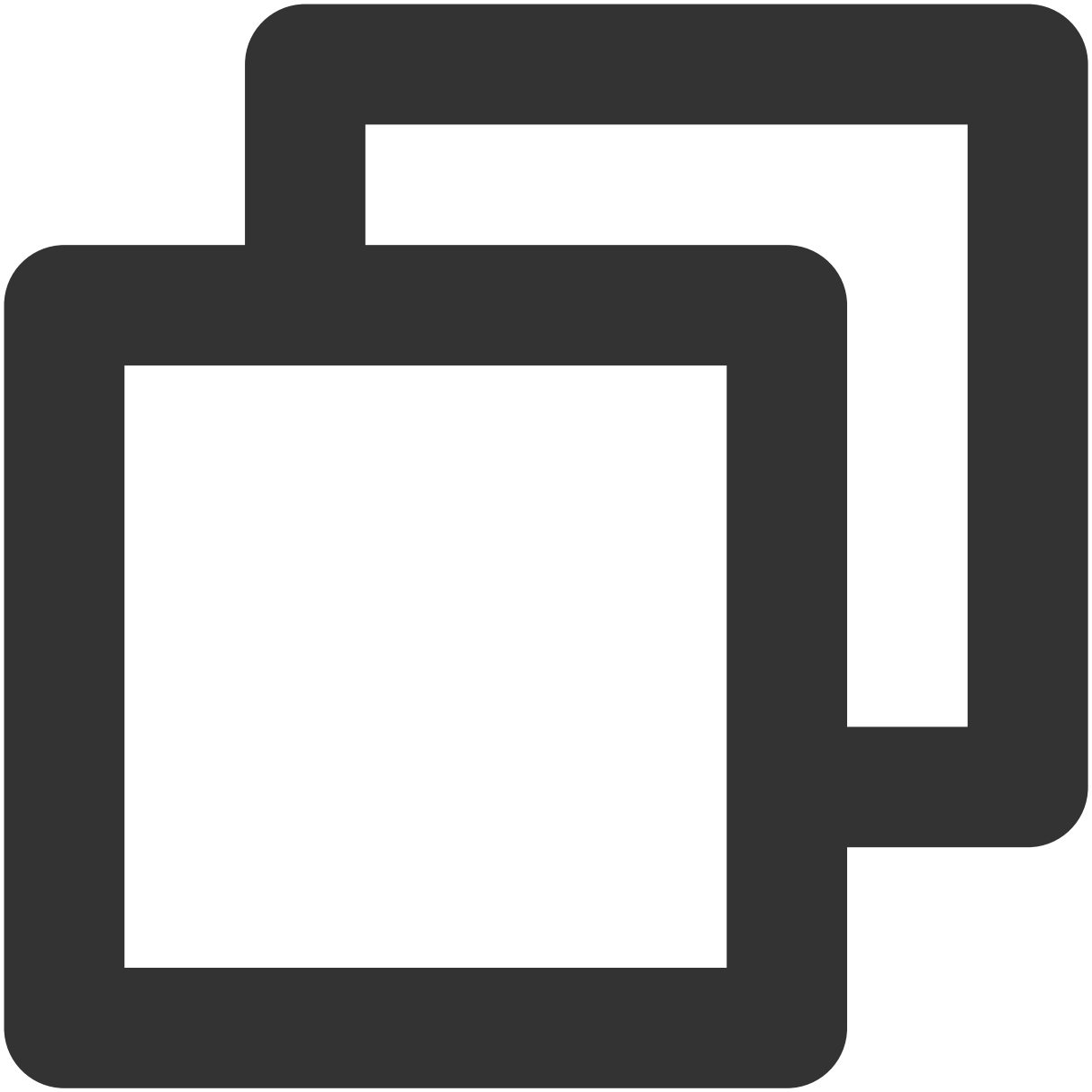
Example:



```
> SELECT input_file_block_length();
 -1
```

MONOTONICALLY_INCREASING_ID

Function statement:



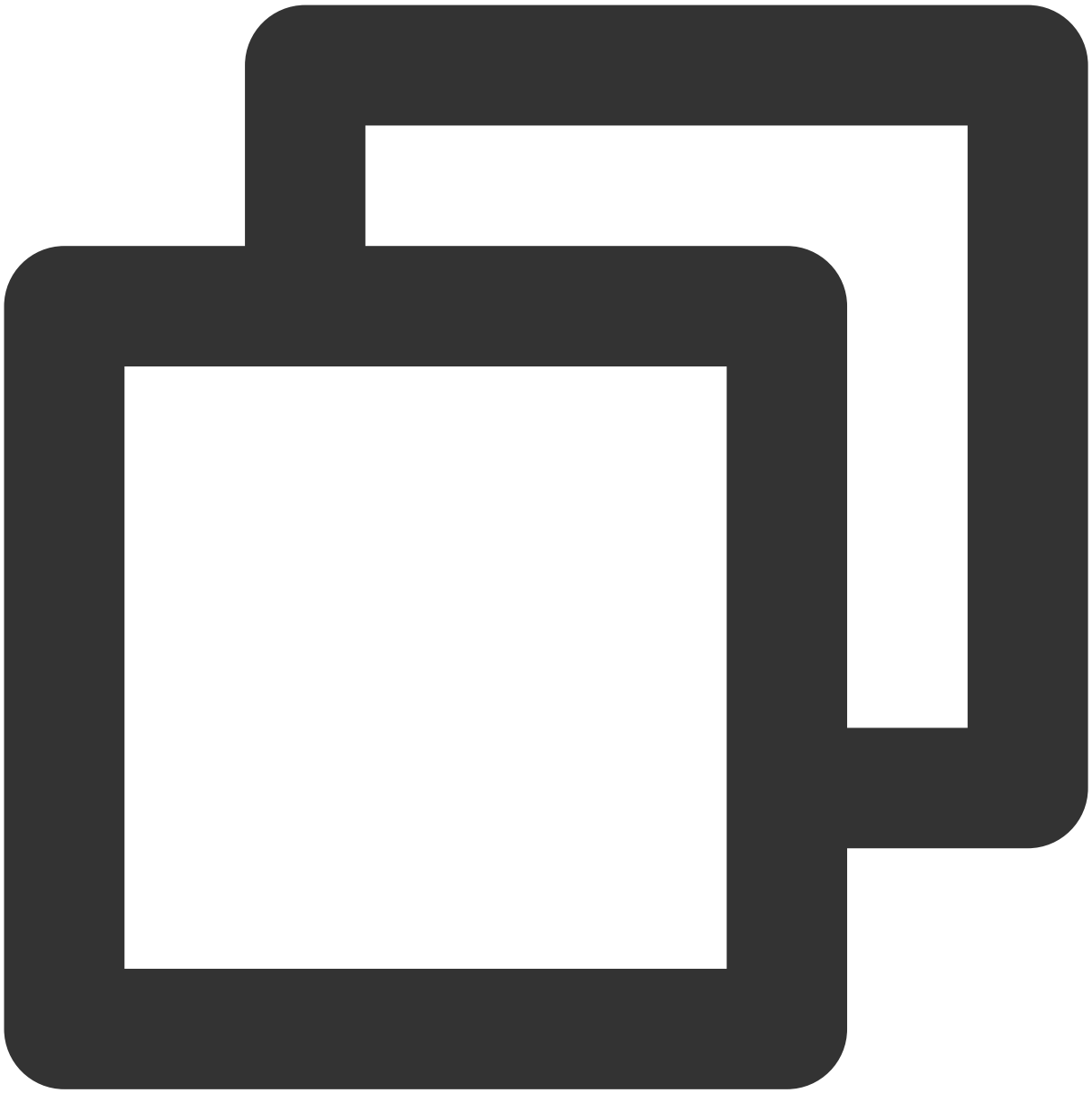
```
MONOTONICALLY_INCREASING_ID()
```

Supported engine: SparkSQL

Usage instructions: Return a monotonically increasing 64-bit integer. The generated ID is guaranteed to be monotonically increasing and unique, but not consecutive. The current implementation puts the partition ID in the high 31 bits and the record number within each partition in the low 33 bits. Assume the data frame has fewer than 1 billion partitions and fewer than 8 billion records per partition. This function is non-deterministic because its result depends on the partition ID.

Return type: bigint

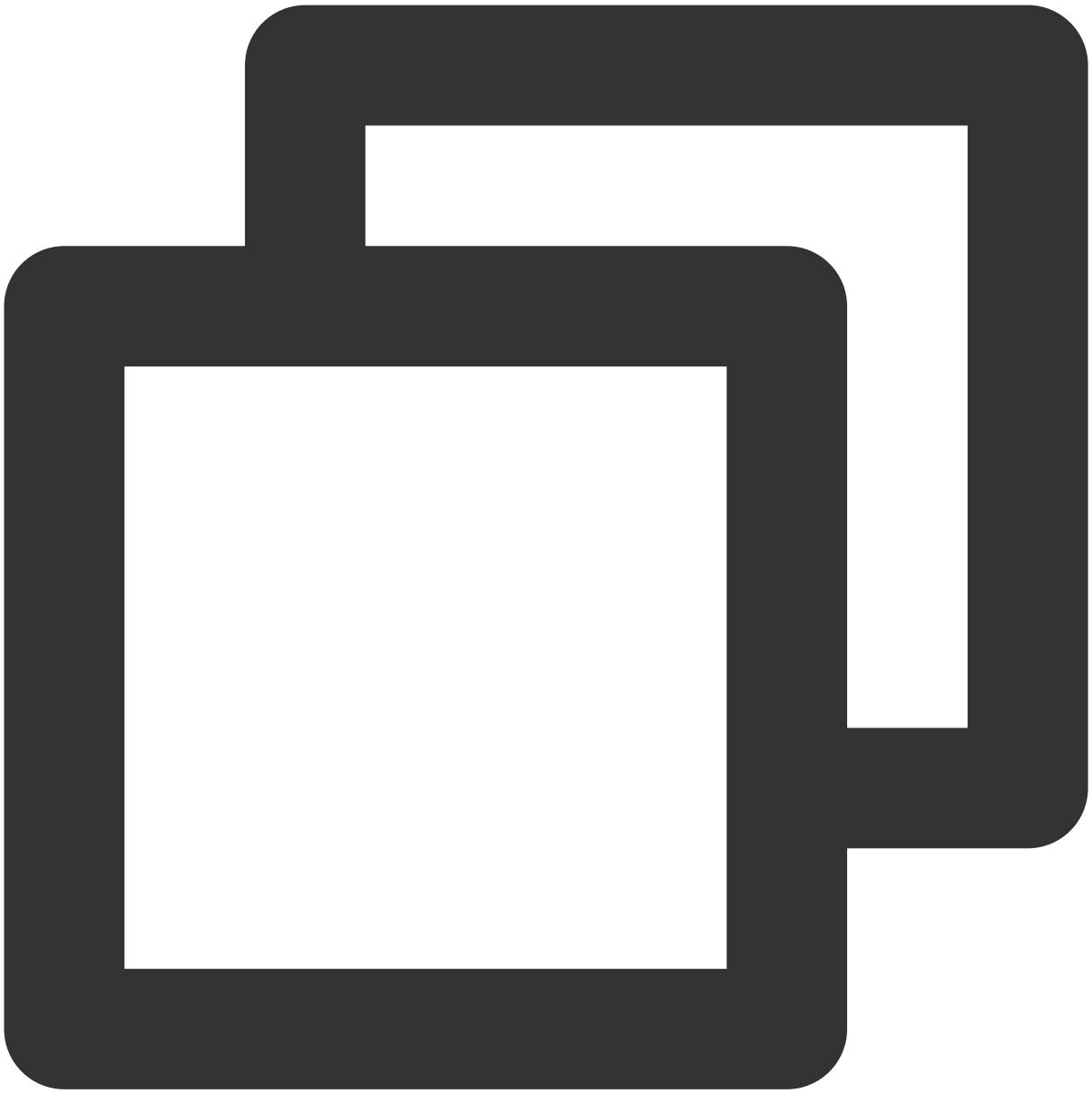
Example:



```
> SELECT monotonically_increasing_id();
 0
```

CURRENT_DATABASE

Function statement:



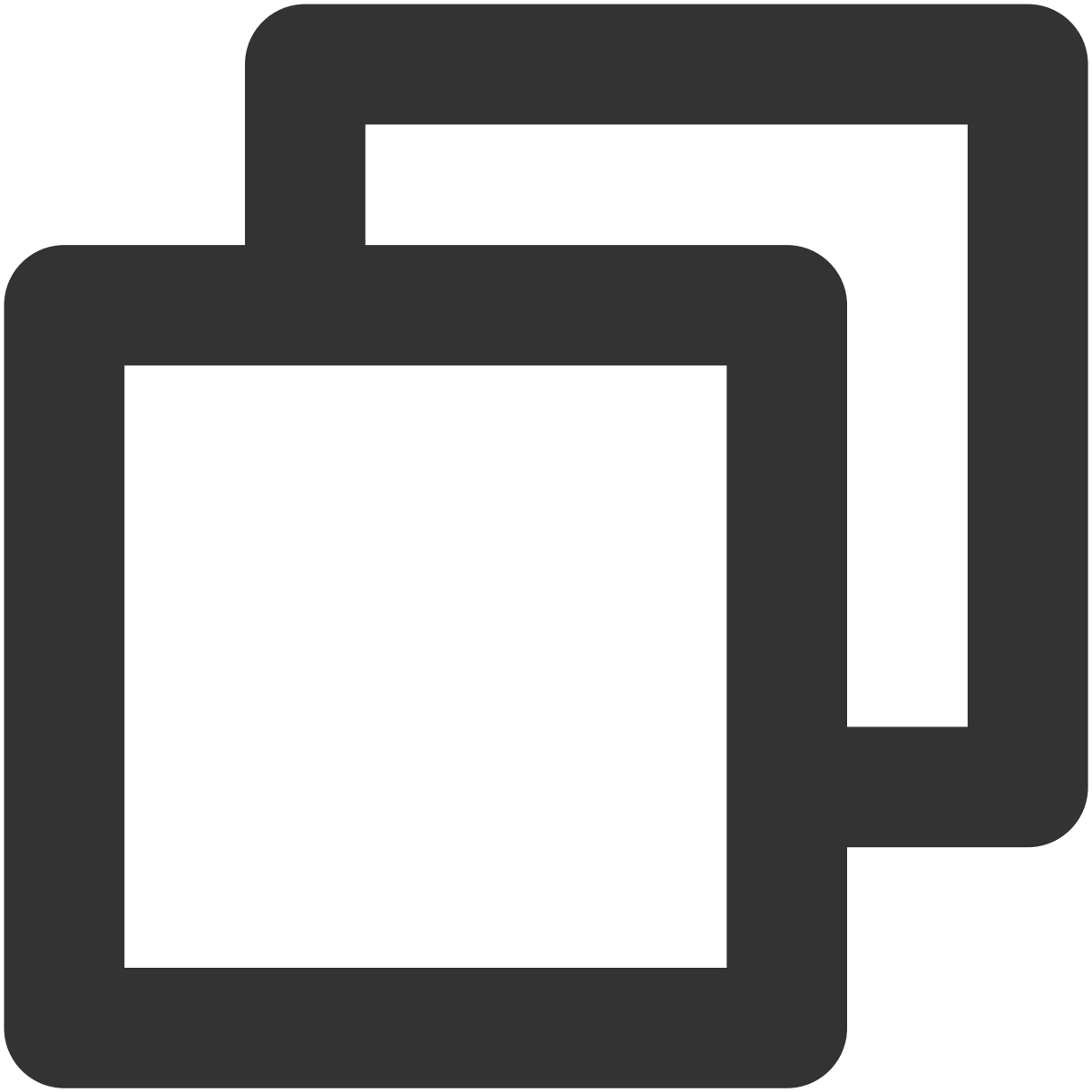
```
CURRENT_DATABASE()
```

Supported engine: SparkSQL

Usage instructions: Return the current database.

Return type: string

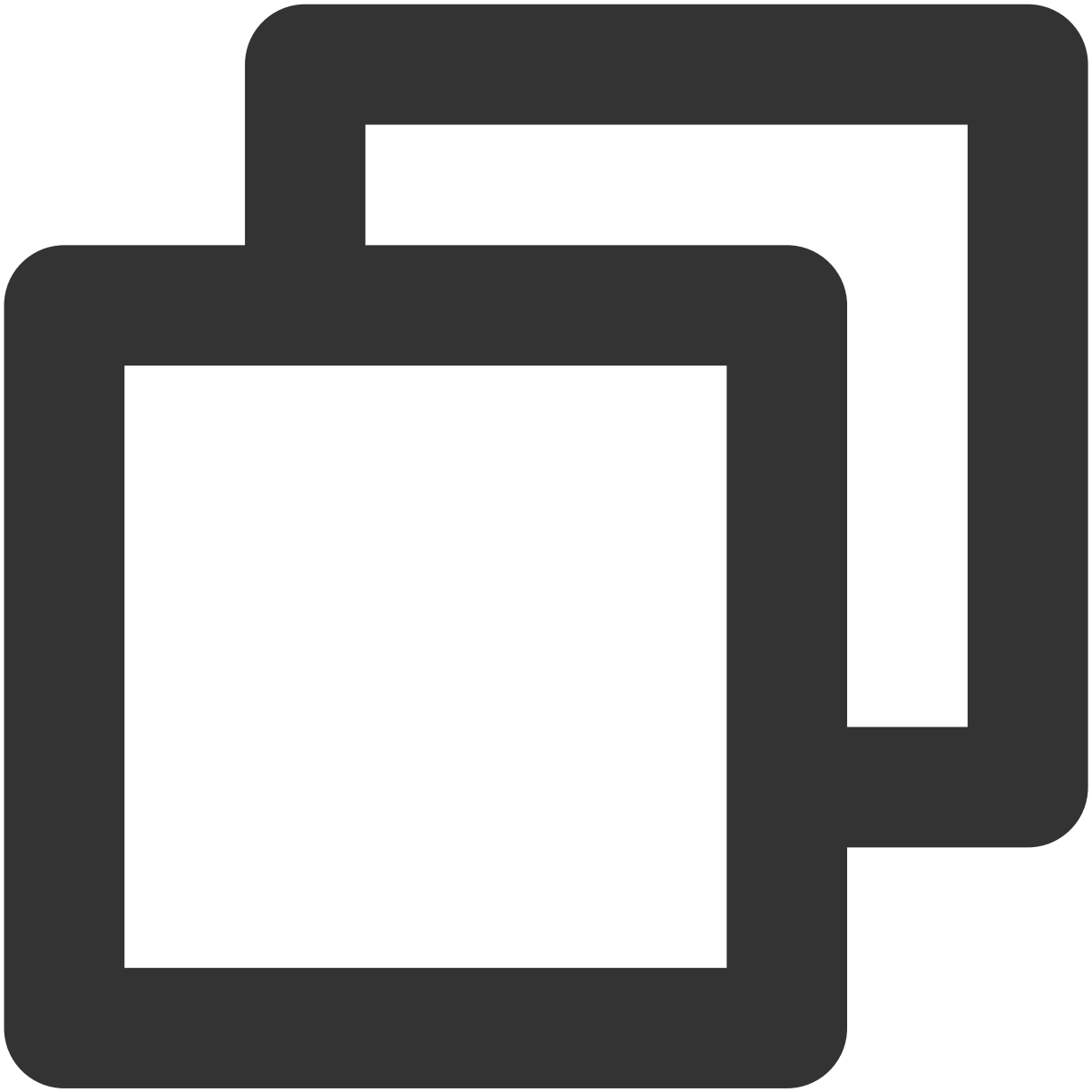
Example:



```
> SELECT current_database();
 default
```

CURRENT_CATALOG

Function statement:



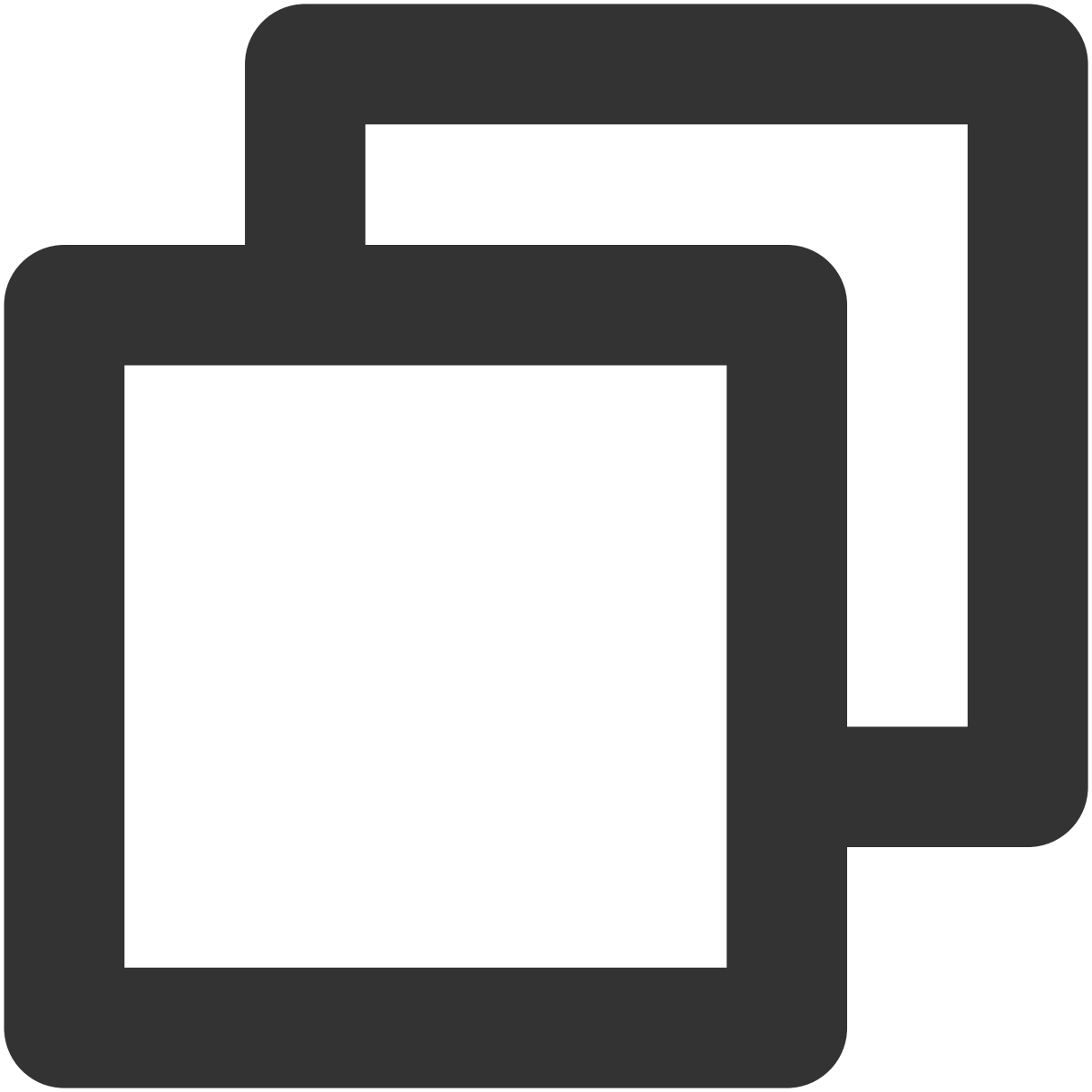
```
CURRENT_CATALOG()
```

Supported engine: SparkSQL

Usage instructions: Return the current catalog.

Return type: string

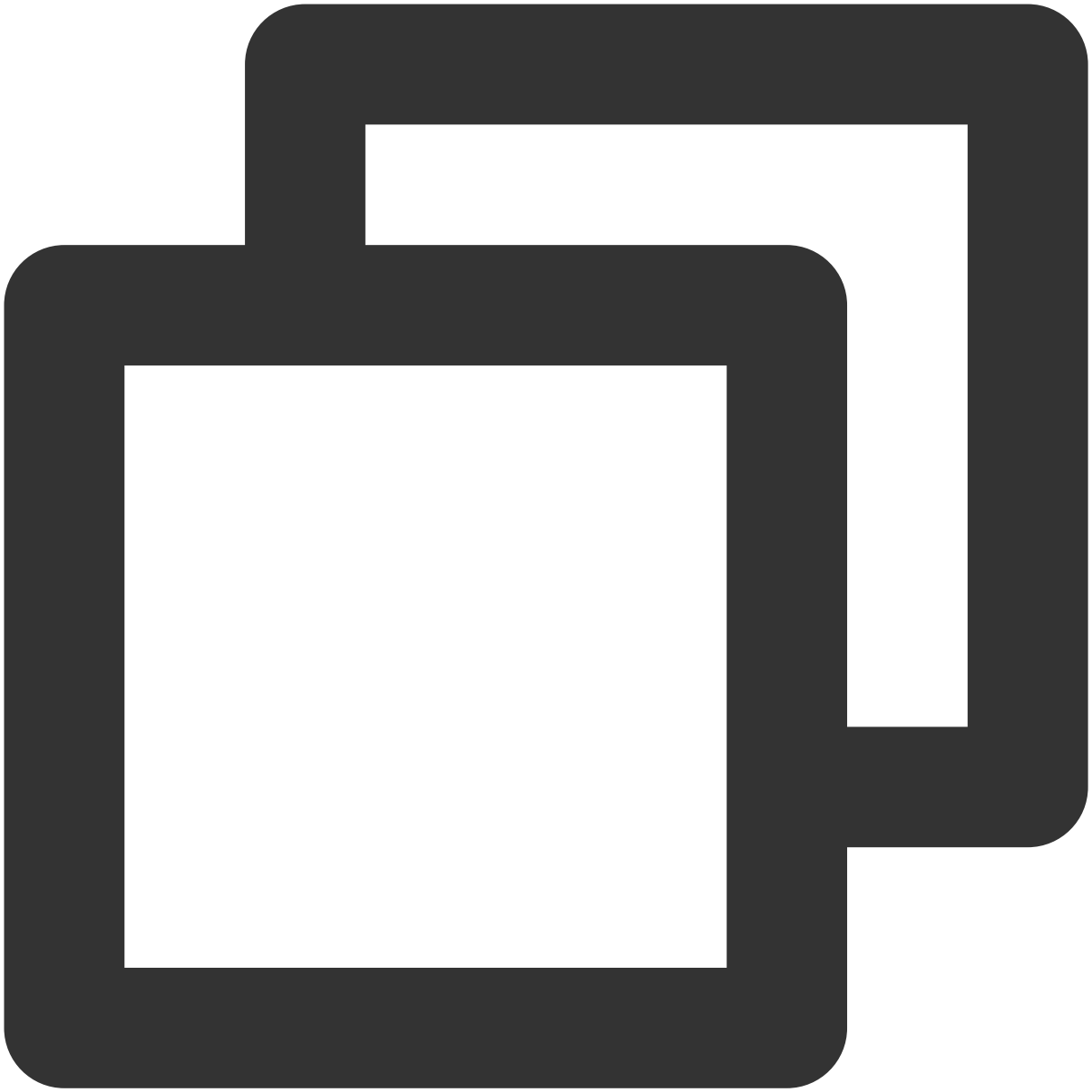
Example:



```
> SELECT current_catalog();
 spark_catalog
```

CURRENT_USER

Function statement:



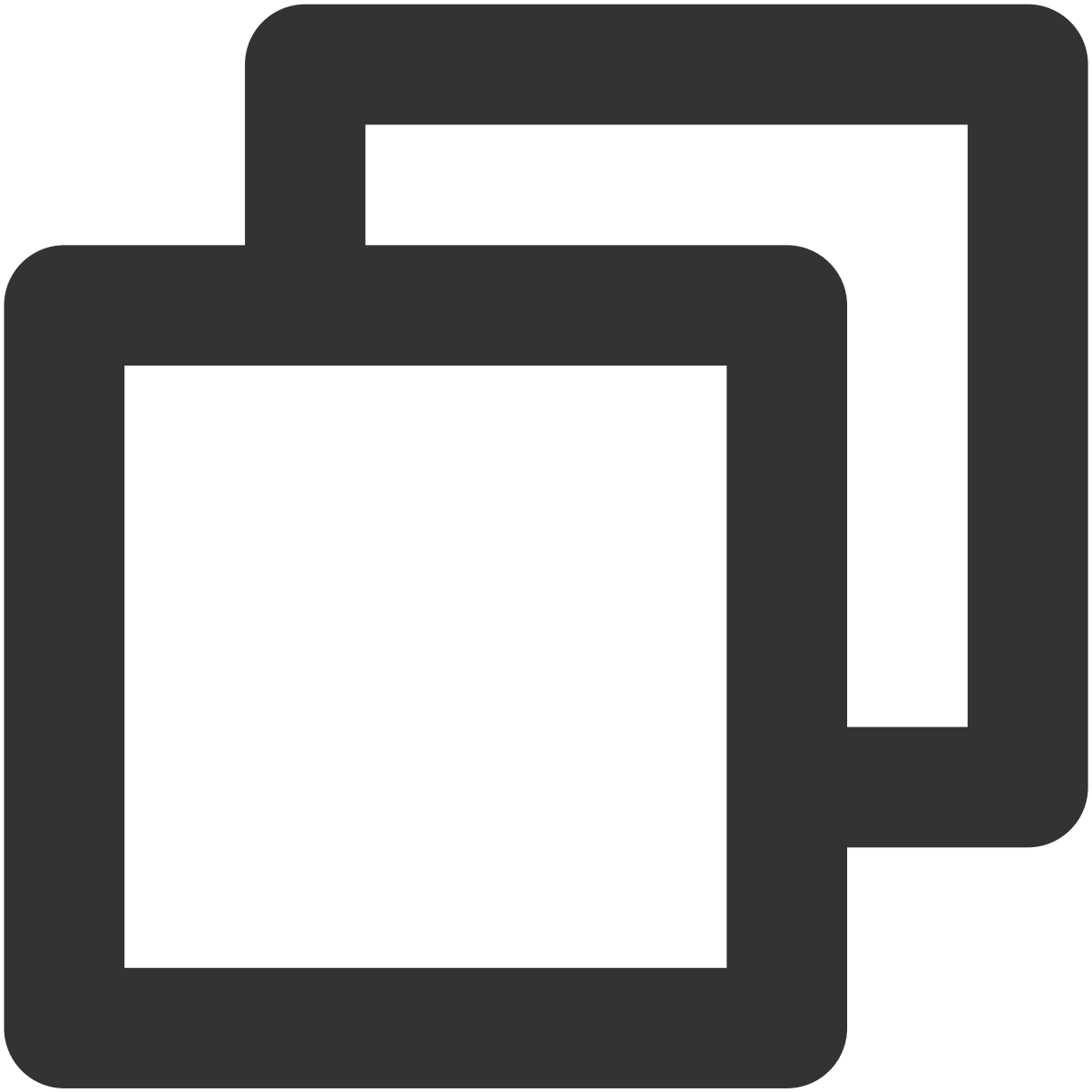
```
CURRENT_USER()
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the current user.

Return type: string

Example:



```
> SELECT current_user();
```

REFLECT

Function statement:

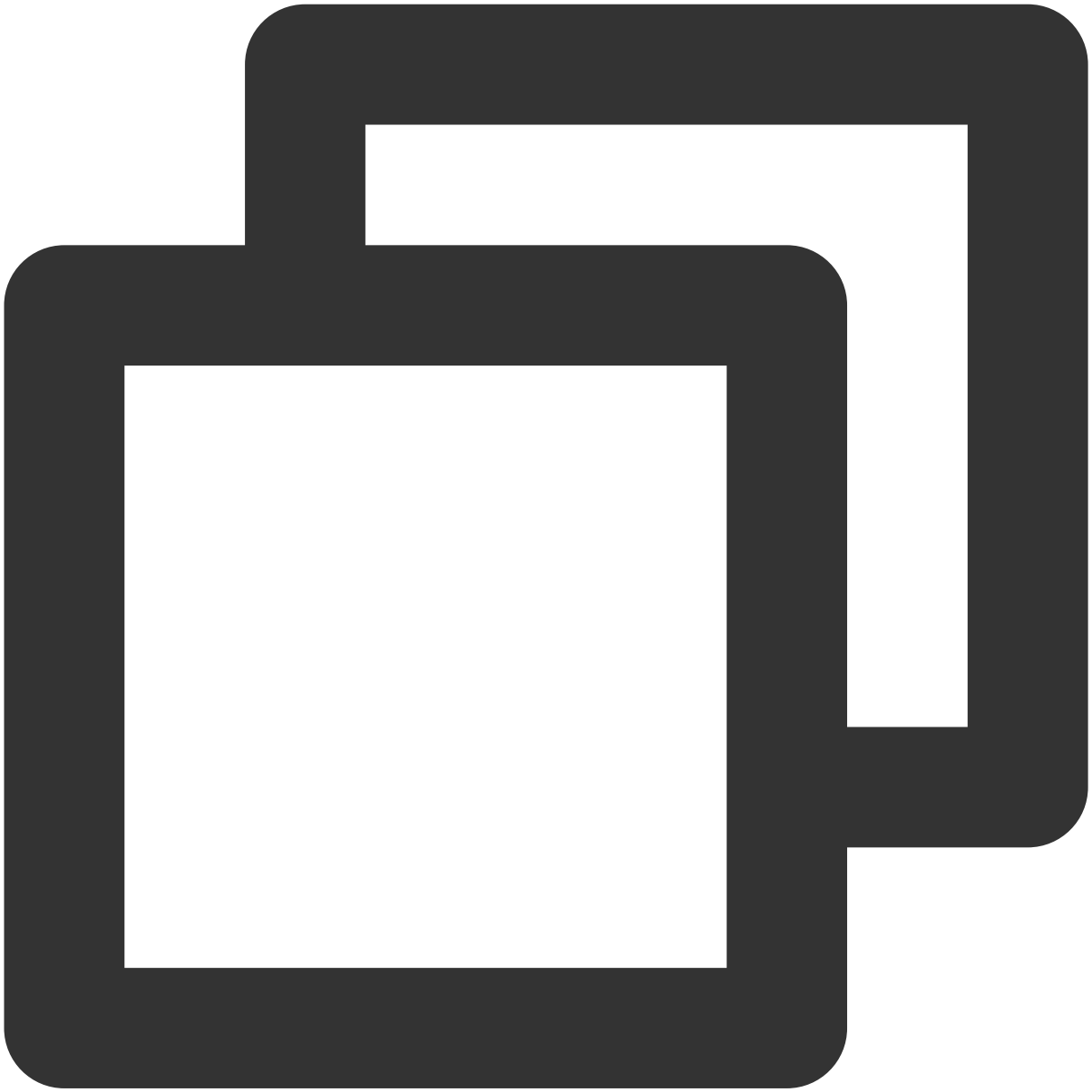
```
REFLECT(<class> string, <method> string[, <arg1> T1[, <arg2> T2, ...]])
```

Supported engines: SparkSQL and Presto

Usage instructions: Call a method with reflection.

Return type: string

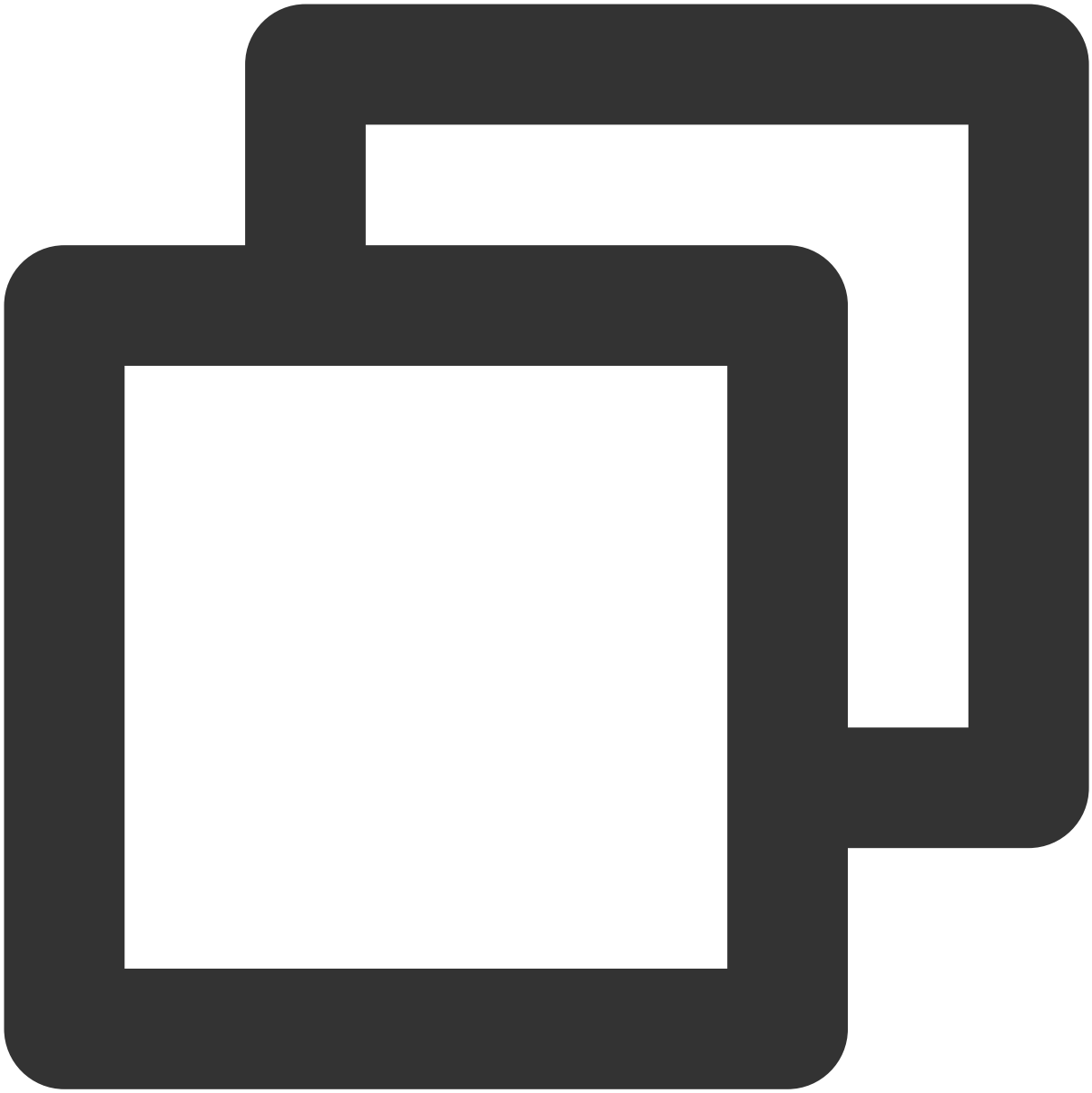
Example:



```
> select reflect('java.lang.Math', 'abs', -1);
 1
```

JAVA_METHOD

Function statement:



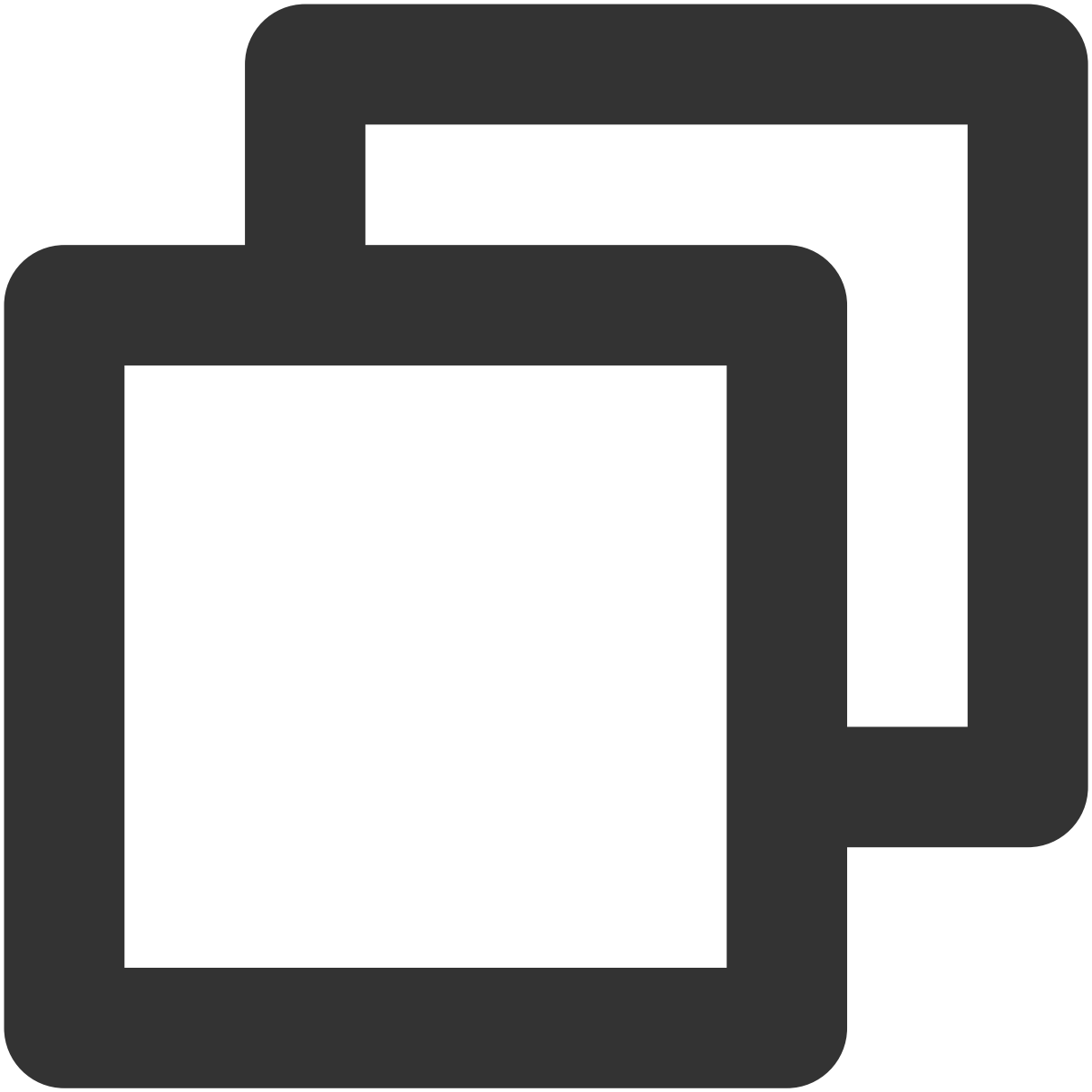
```
JAVA_METHOD(<class> string, <method> string[, <arg1> T1[, <arg2> T2, ...]])
```

Supported engines: SparkSQL and Presto

Usage instructions: Call a method with reflection.

Return type: string

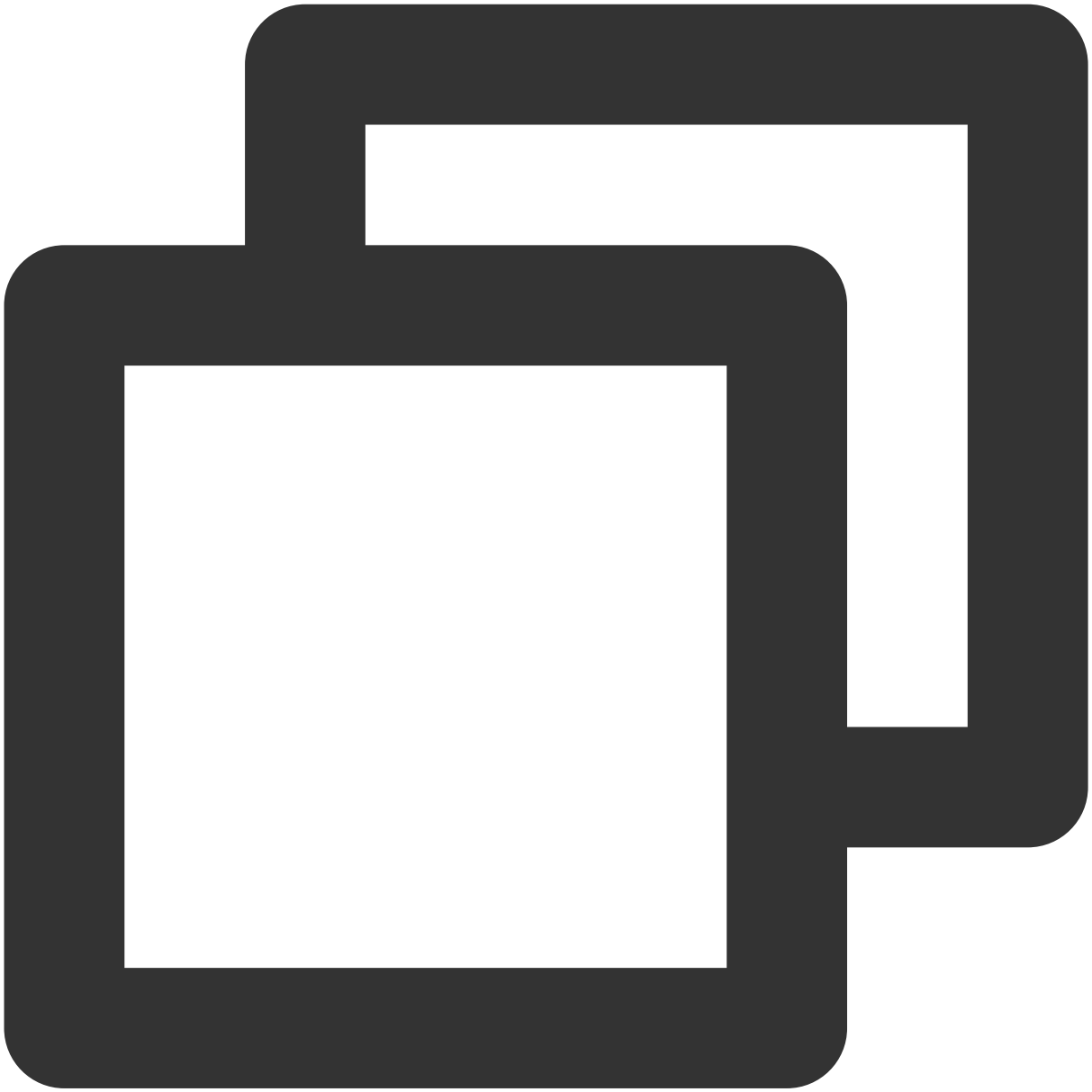
Example:



```
> select JAVA_METHOD('java.lang.Math', 'abs', -1);
 1
```

VERSION

Function statement:



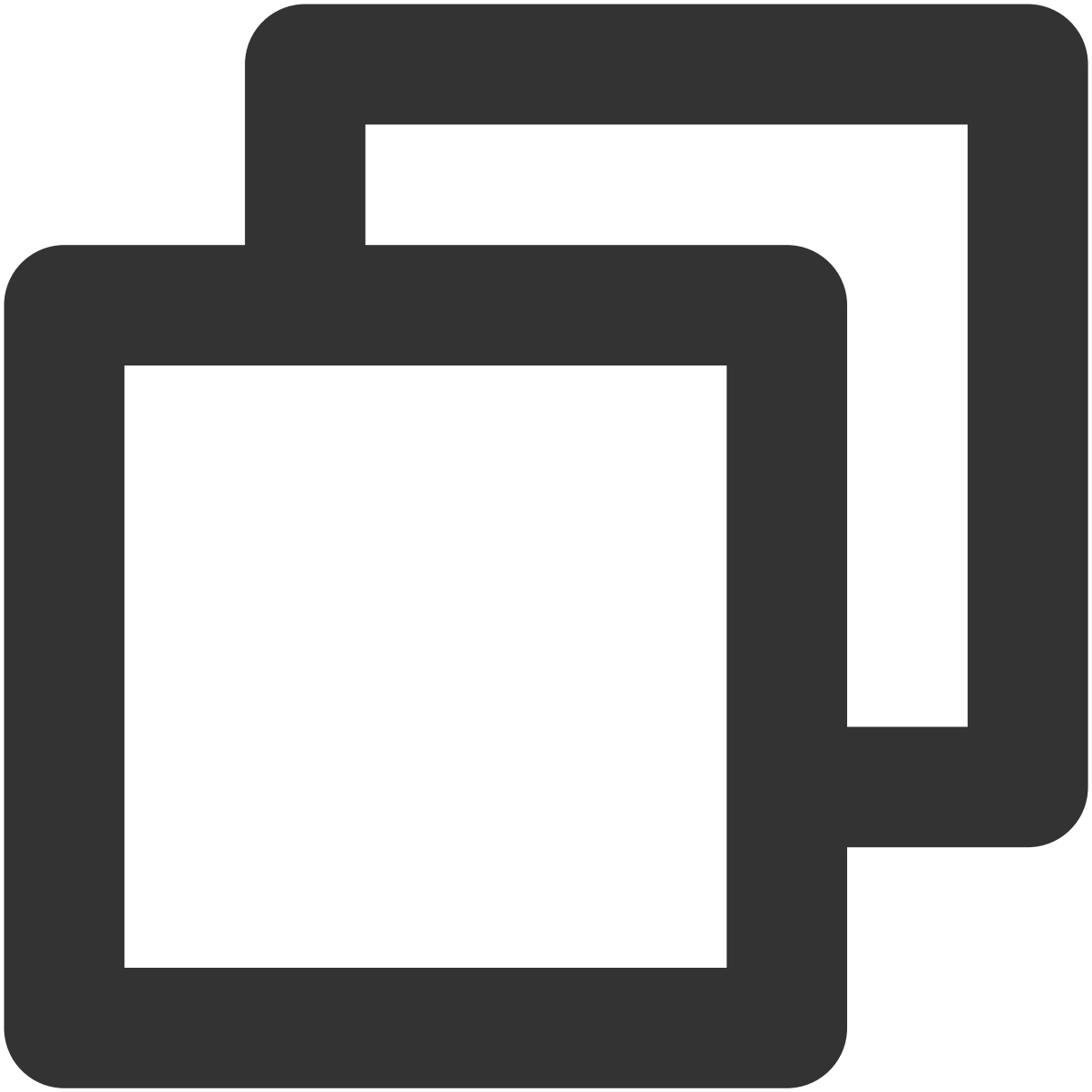
```
VERSION()
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the engine version.

Return type: string

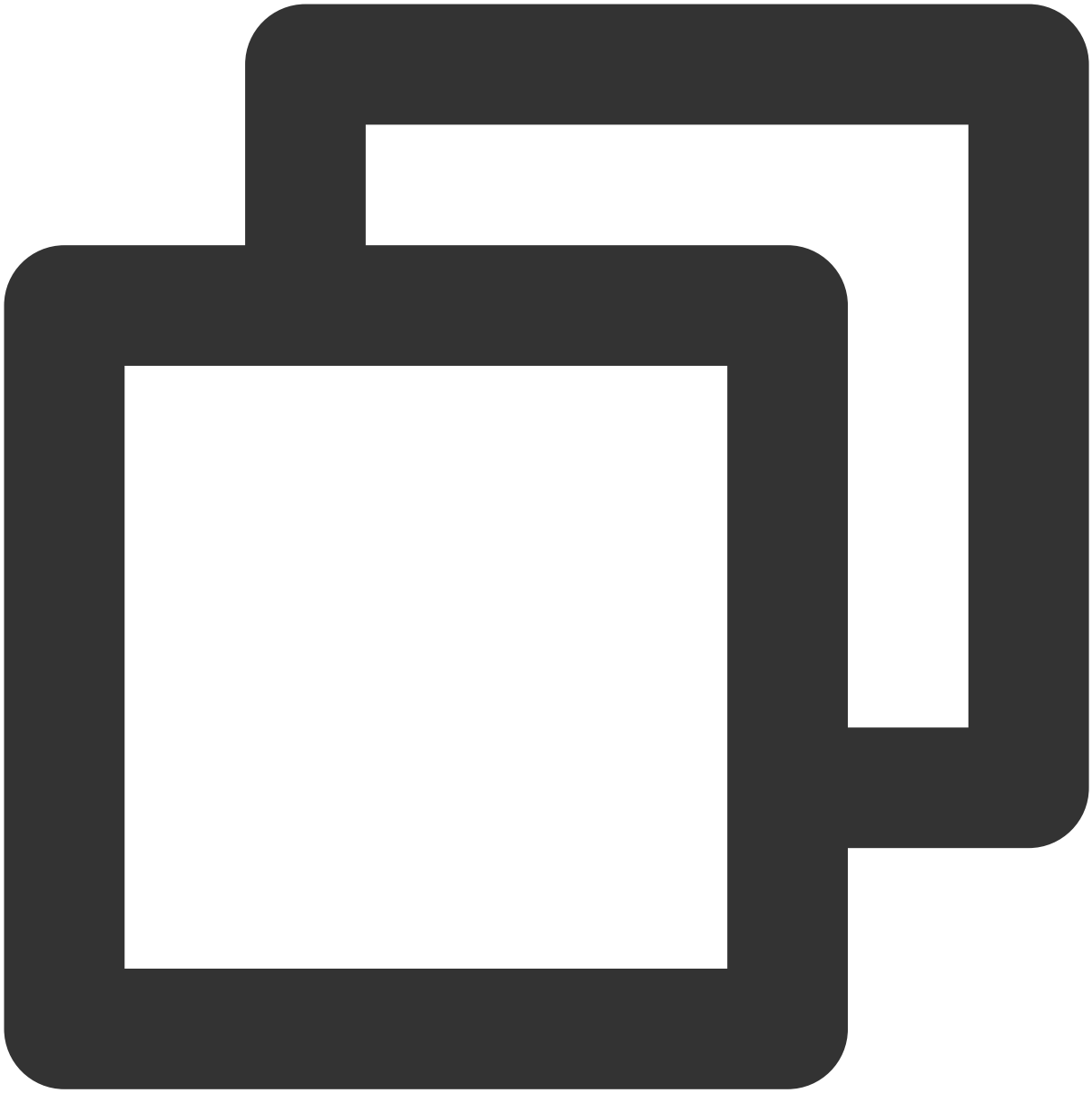
Example:



```
> select VERSION()
 3.0.0 rce61711a5fa54ab34fc74d86d521ecaeea6b072a
```

TYPEOF

Function statement:



```
TYPEOF(<expr> T)
```

Supported engines: SparkSQL and Presto

Usage instructions: Return the data type of expr.

Return type: string

Example:

```
> SELECT typeof(1);
 int
> SELECT typeof(array(1));
 array<int>
```

CAST

Function statement:

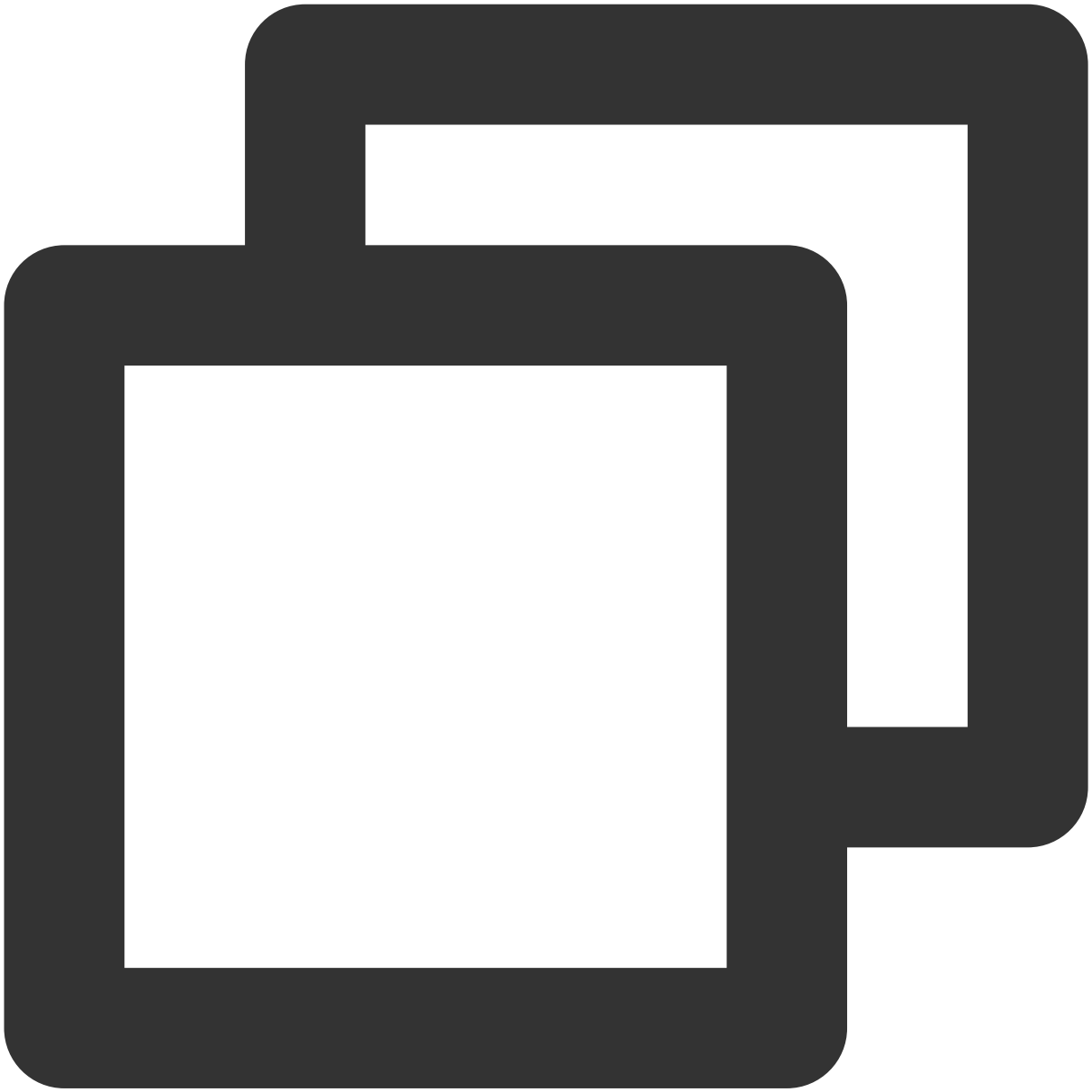
```
CAST(<expr> AS <type>)
```

Supported engines: SparkSQL and Presto

Usage instructions: Convert expr to the type.

Return type: <type>

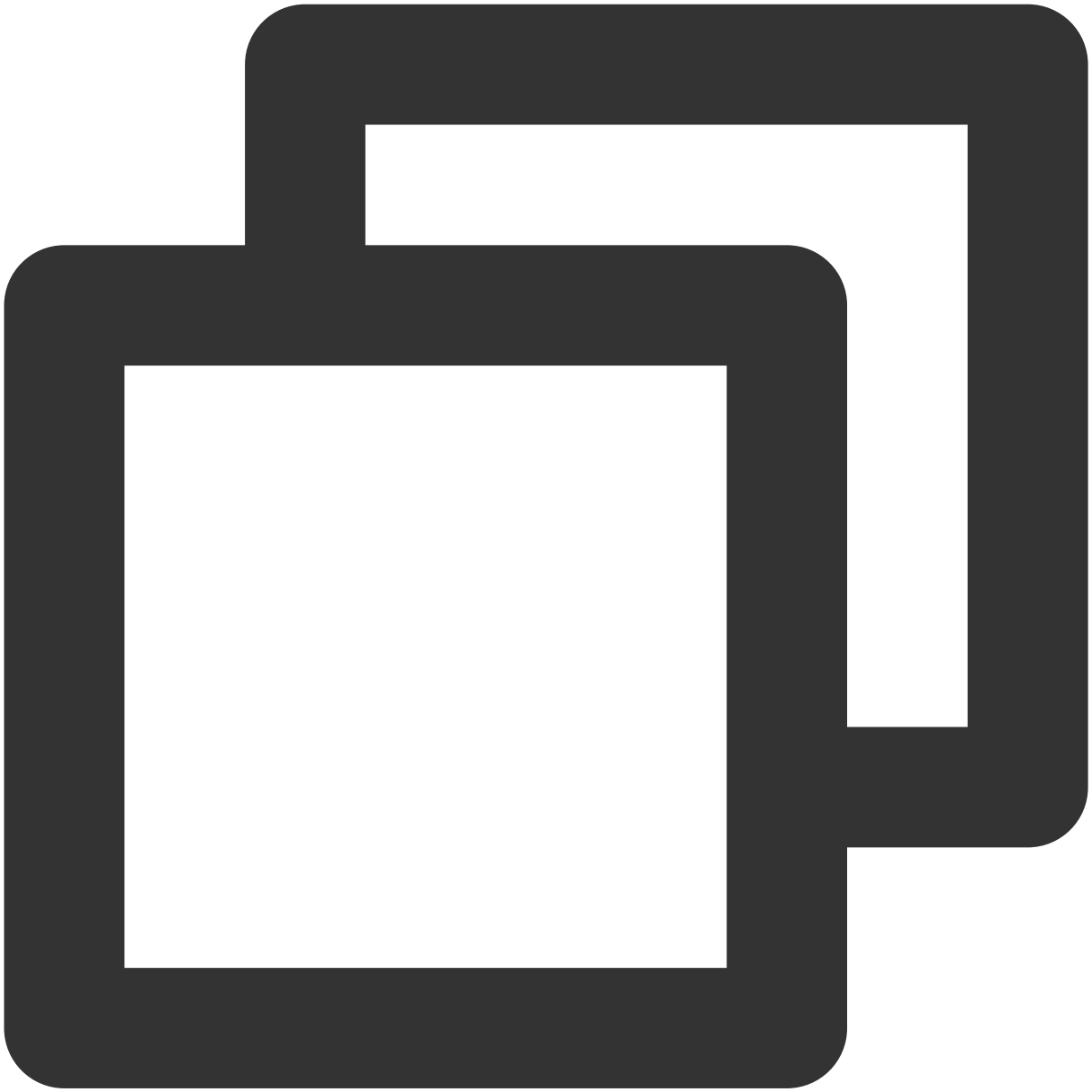
Example:



```
> SELECT cast('10' as int);
 10
```

BOOLEAN

Function statement:



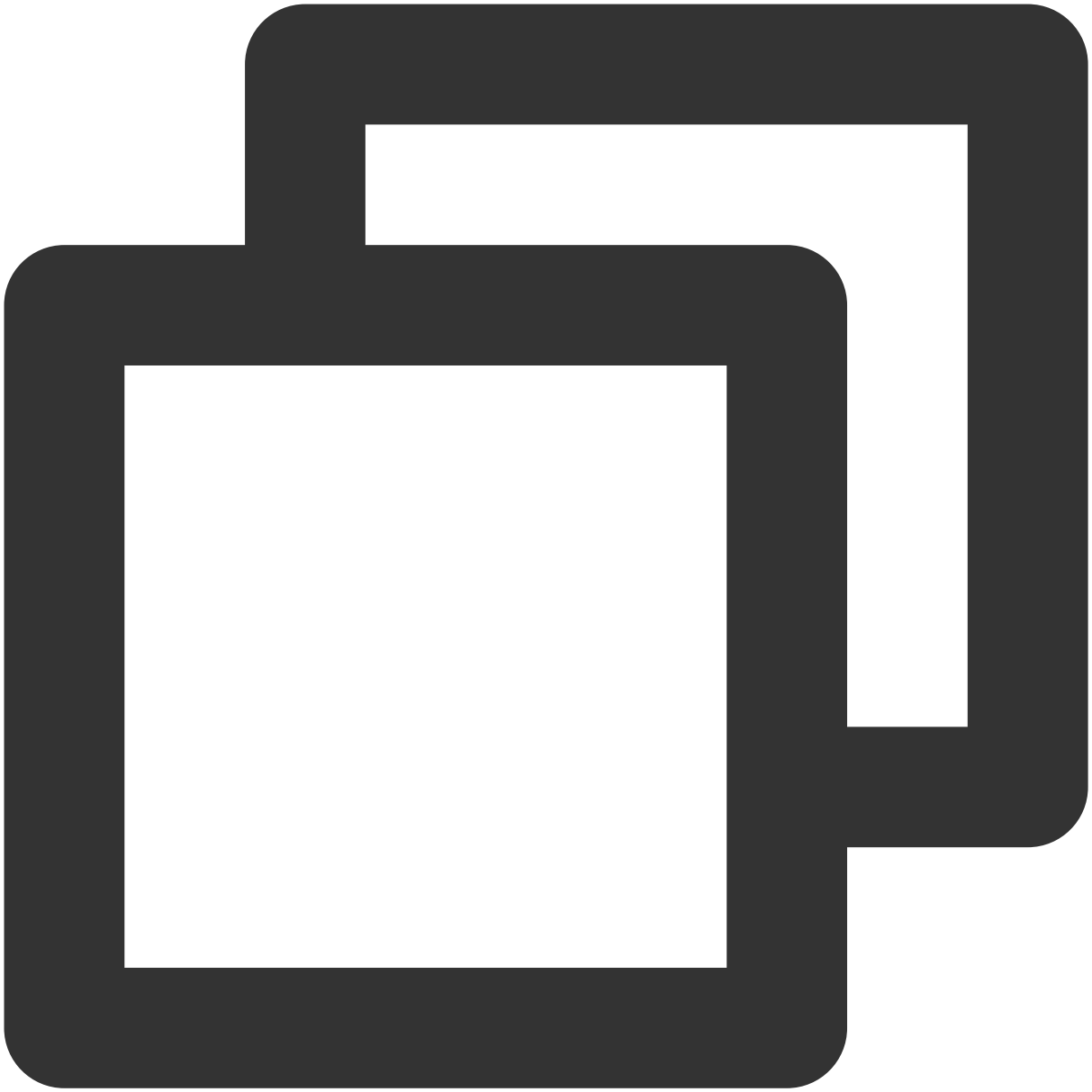
```
BOOLEAN(<expr> T)
```

Supported engines: SparkSQL and Presto

Usage instructions: Convert expr to the boolean type.

Return type: boolean

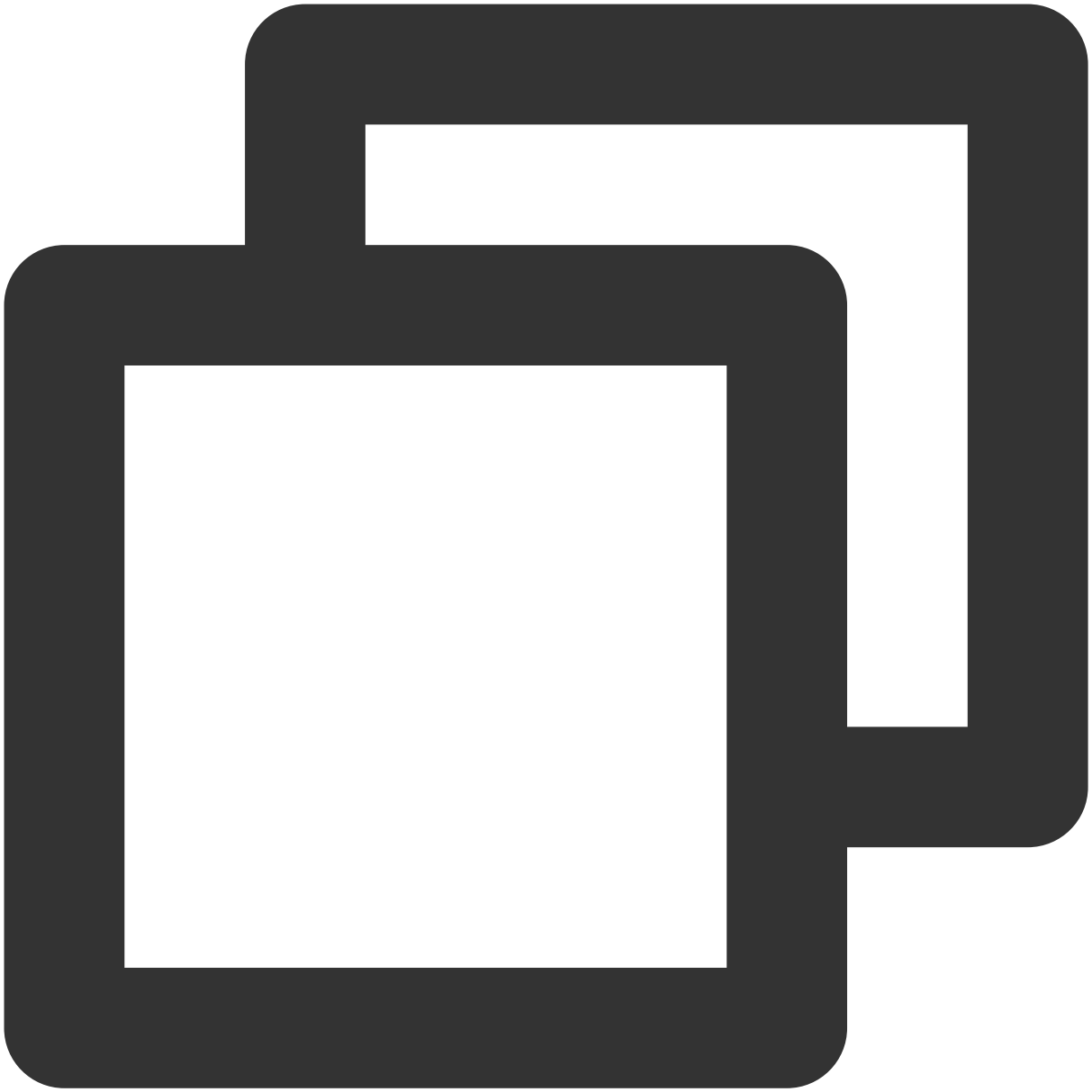
Example:



```
> SELECT boolean(1);
 true
```

BIGINT

Function statement:



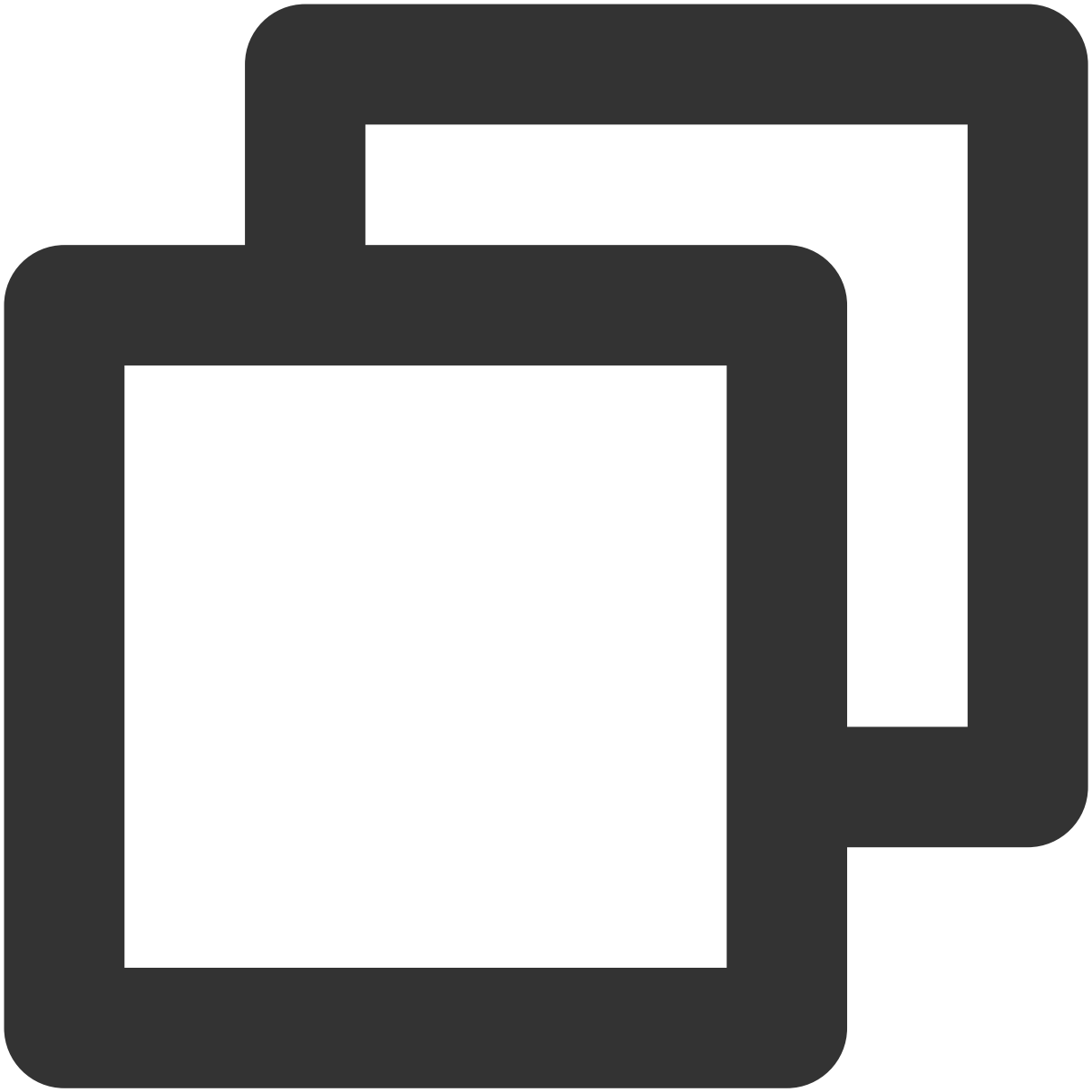
```
BIGINT(<expr> T)
```

Supported engines: SparkSQL and Presto

Usage instructions: Force the type to be converted to bigint.

Return type: bigint

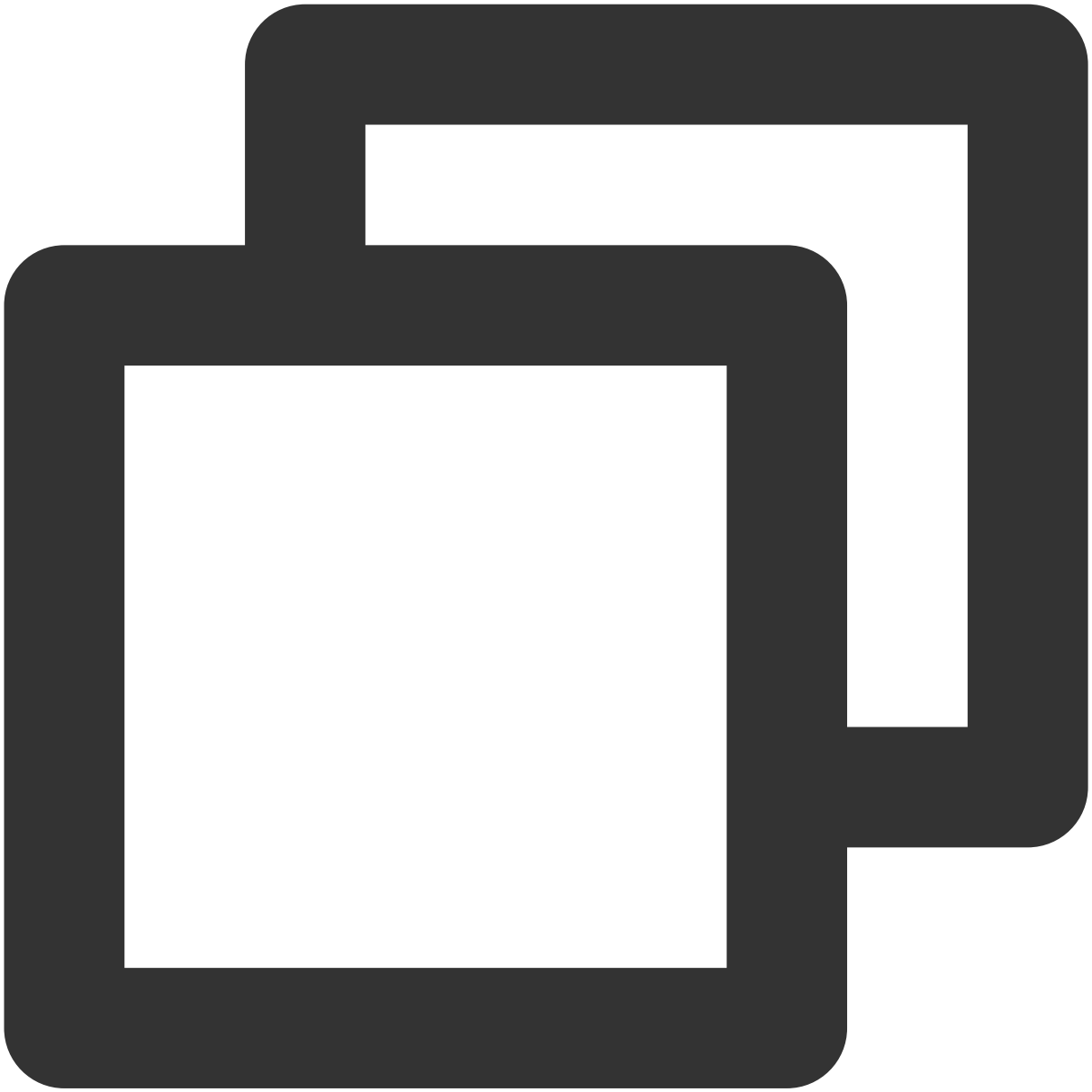
Example:



```
> select bigint(0);
 0
```

BINARY

Function statement:



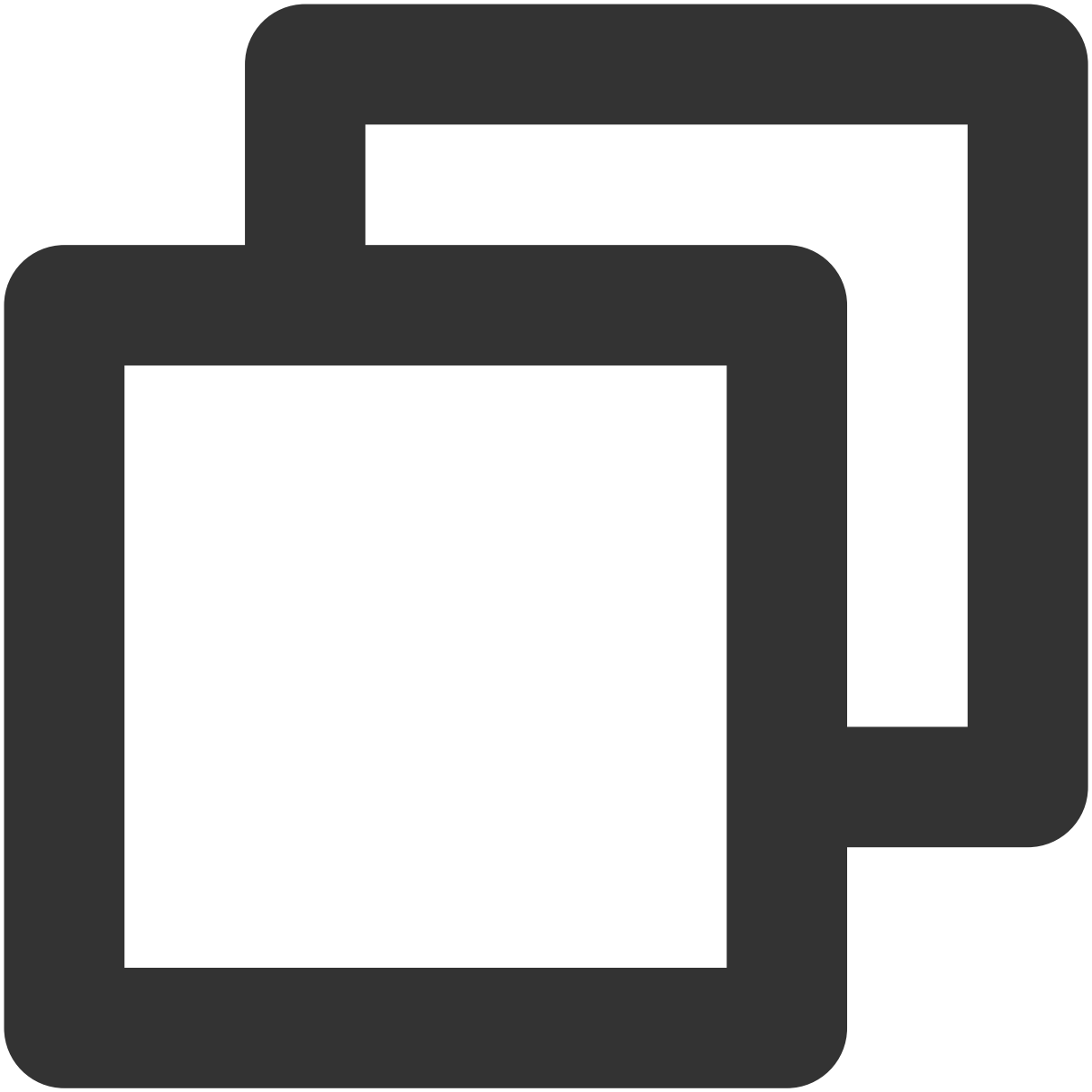
```
BINARY(<expr> T)
```

Supported engines: SparkSQL and Presto

Usage instructions: Force the type to be converted to BINARY.

Return type: binary

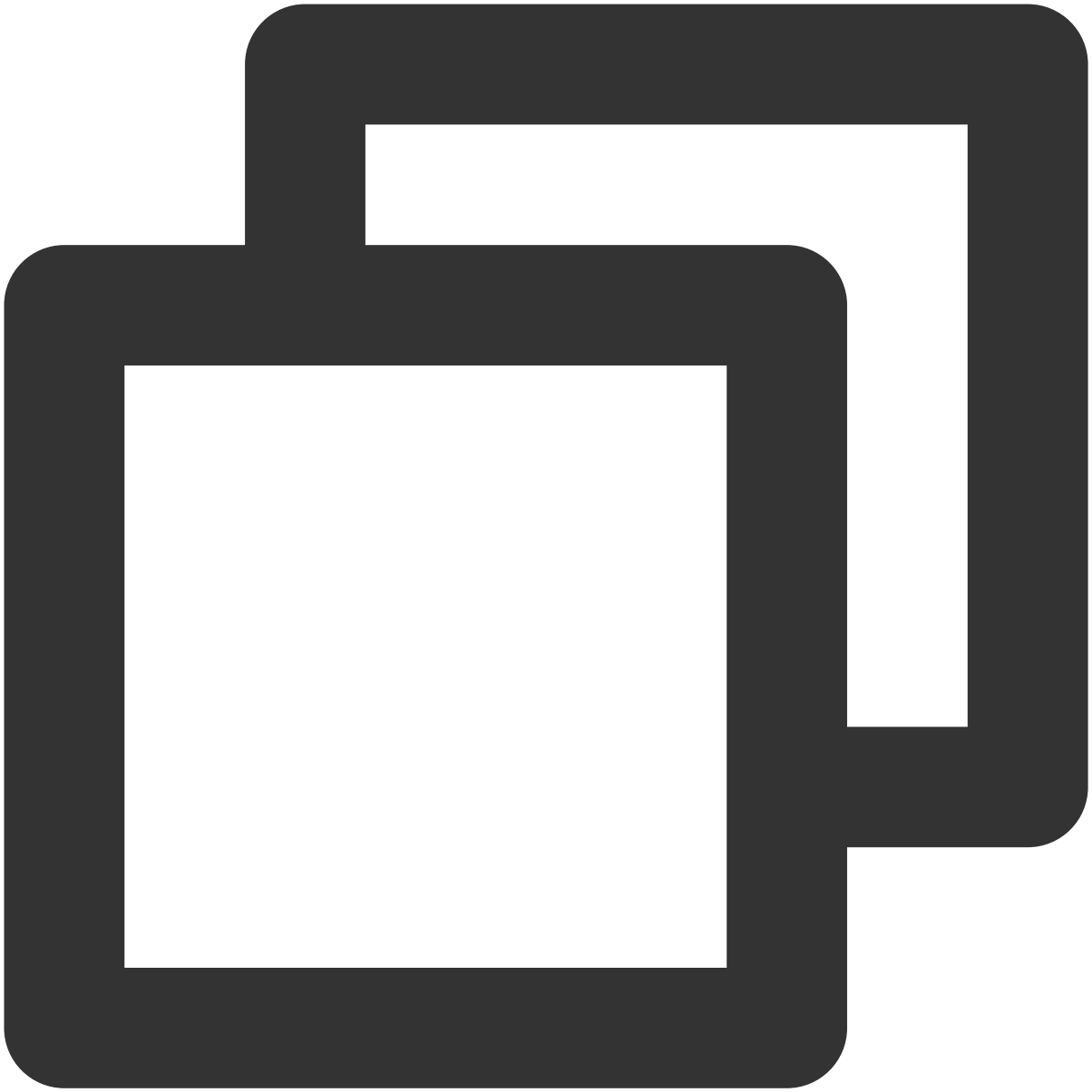
Example:



```
> select binary(65);
 A
```

DOUBLE

Function statement:



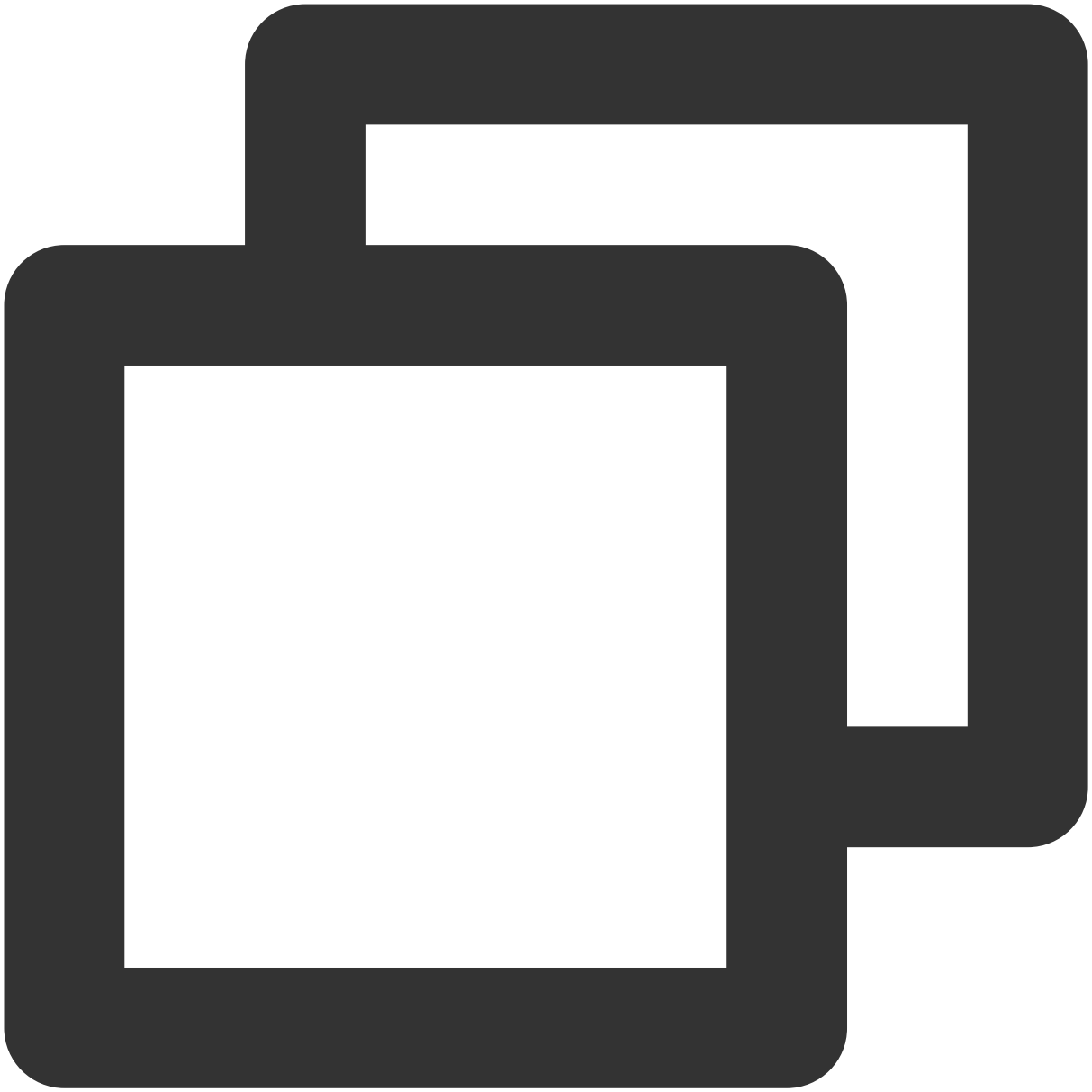
```
DOUBLE(<expr> T)
```

Supported engines: SparkSQL and Presto

Usage instructions: Force the type to be converted to double.

Return type: double

Example:



```
select double(1);
 1.0
```

FLOAT

Function statement:

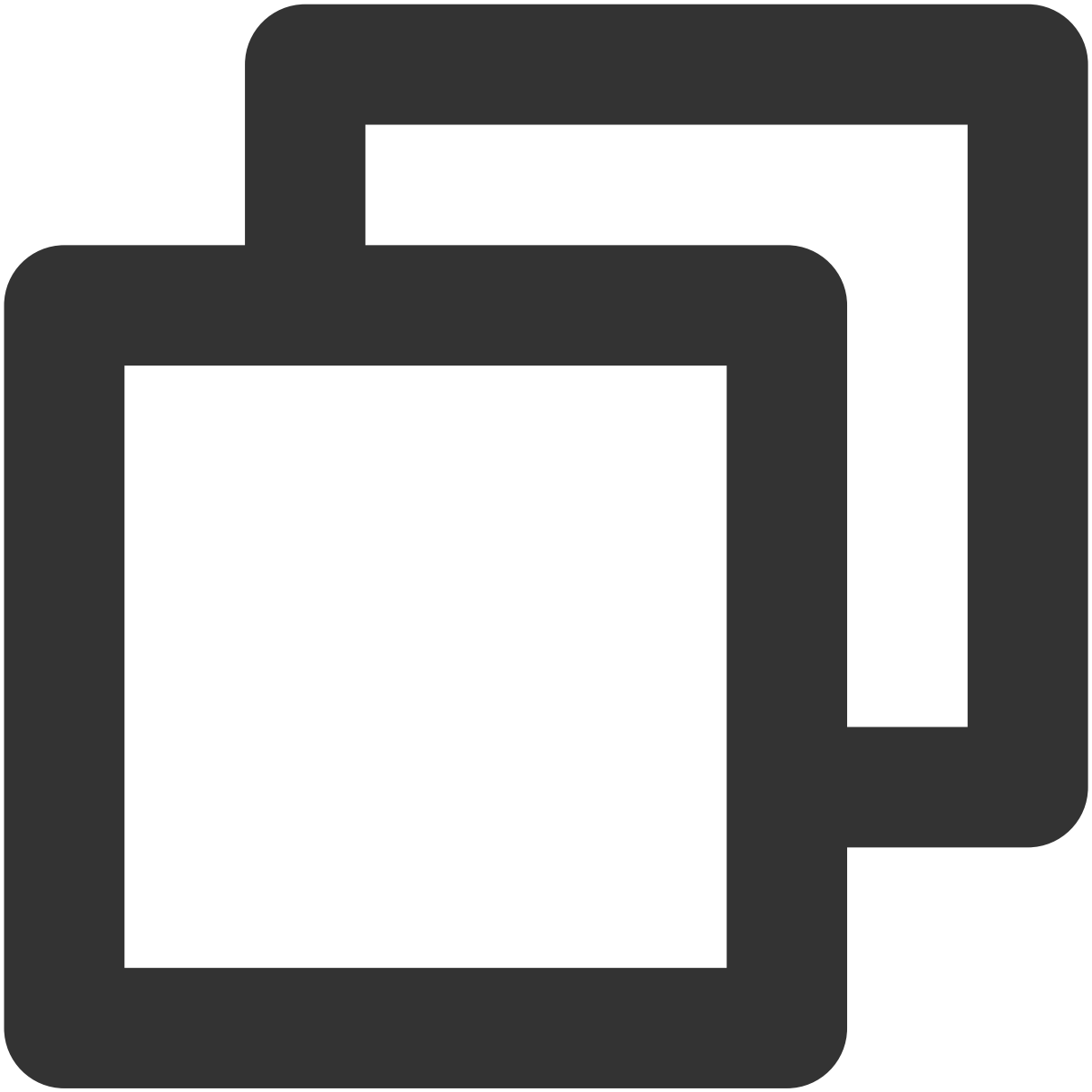
```
FLOAT(<expr> T)
```

Supported engines: SparkSQL and Presto

Usage instructions: Force the type to be converted to float.

Return type: float

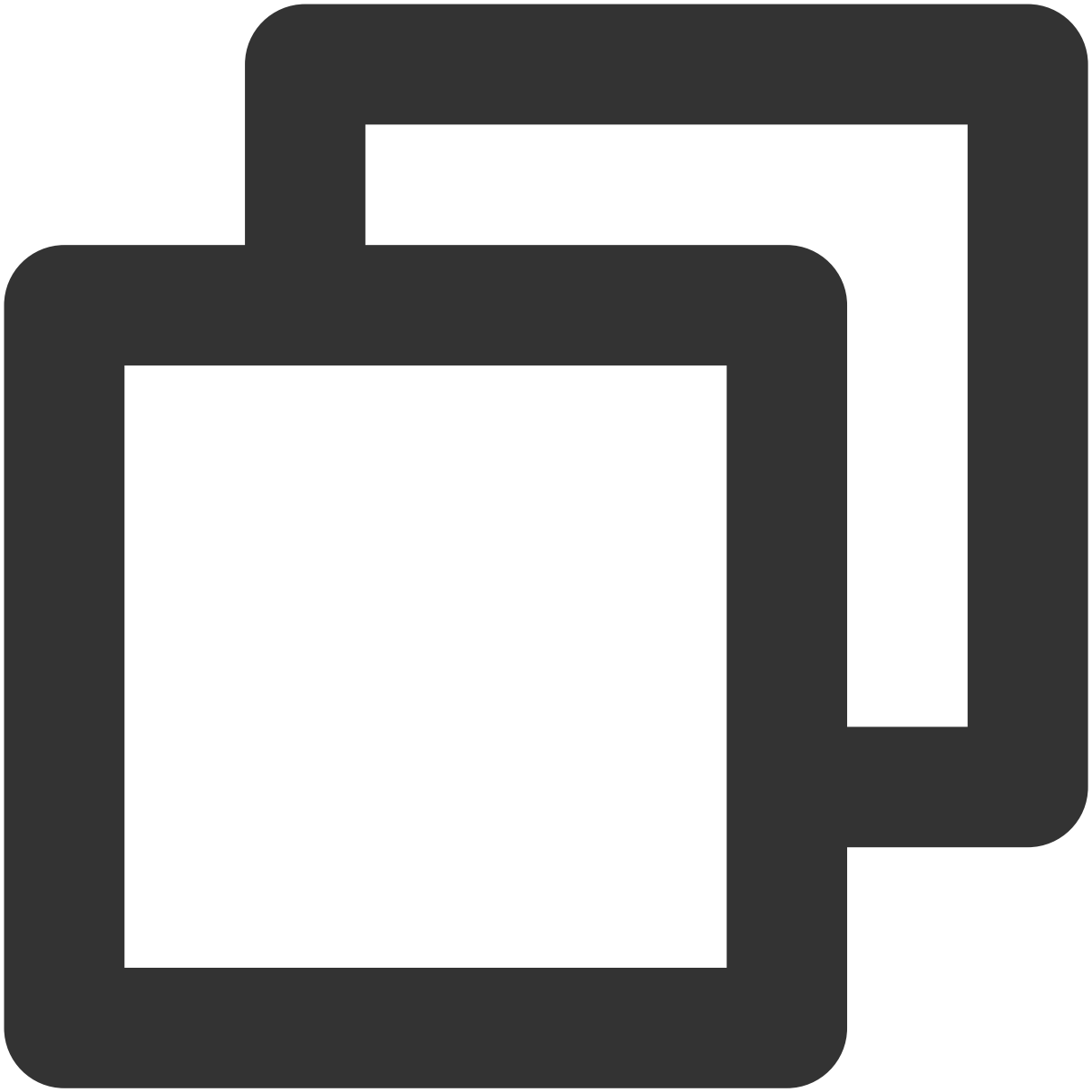
Example:



```
> select float(1);
 1.0
```

INT

Function statement:



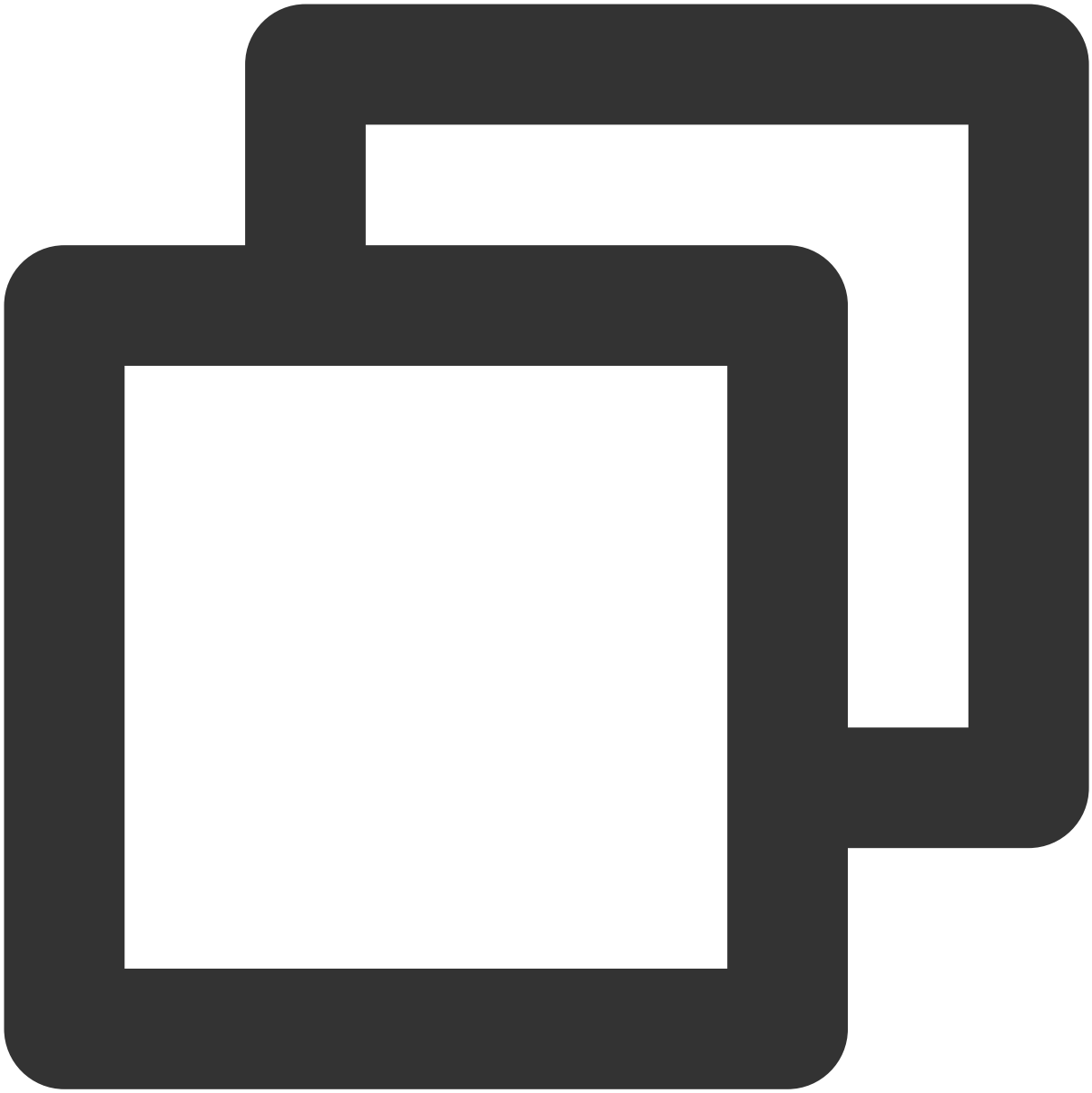
```
INT(<expr> T)
```

Supported engines: SparkSQL and Presto

Usage instructions: Force the type to be converted to integer.

Return type: integer

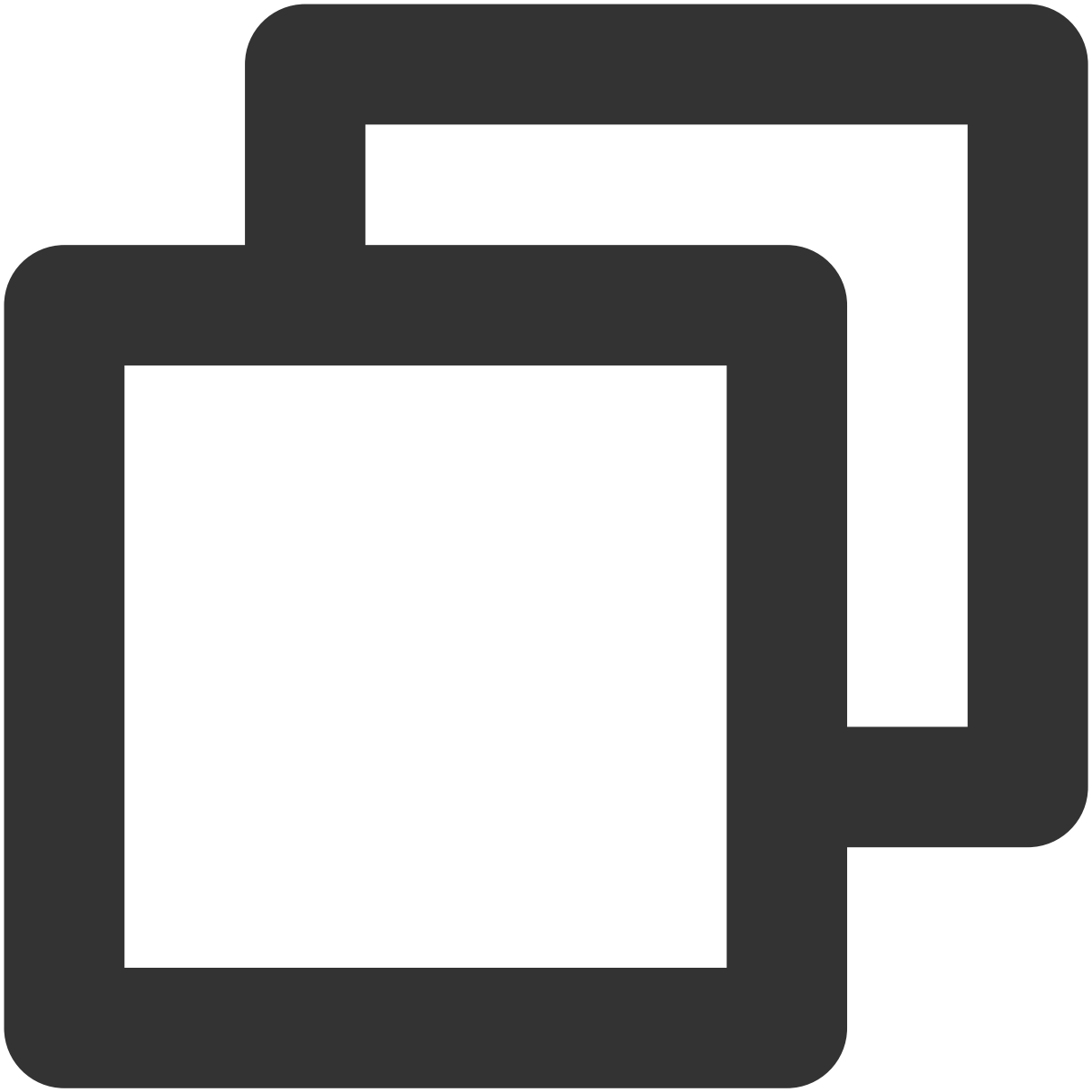
Example:



```
> select int(1.0);
 1
```

SMALLINT

Function statement:



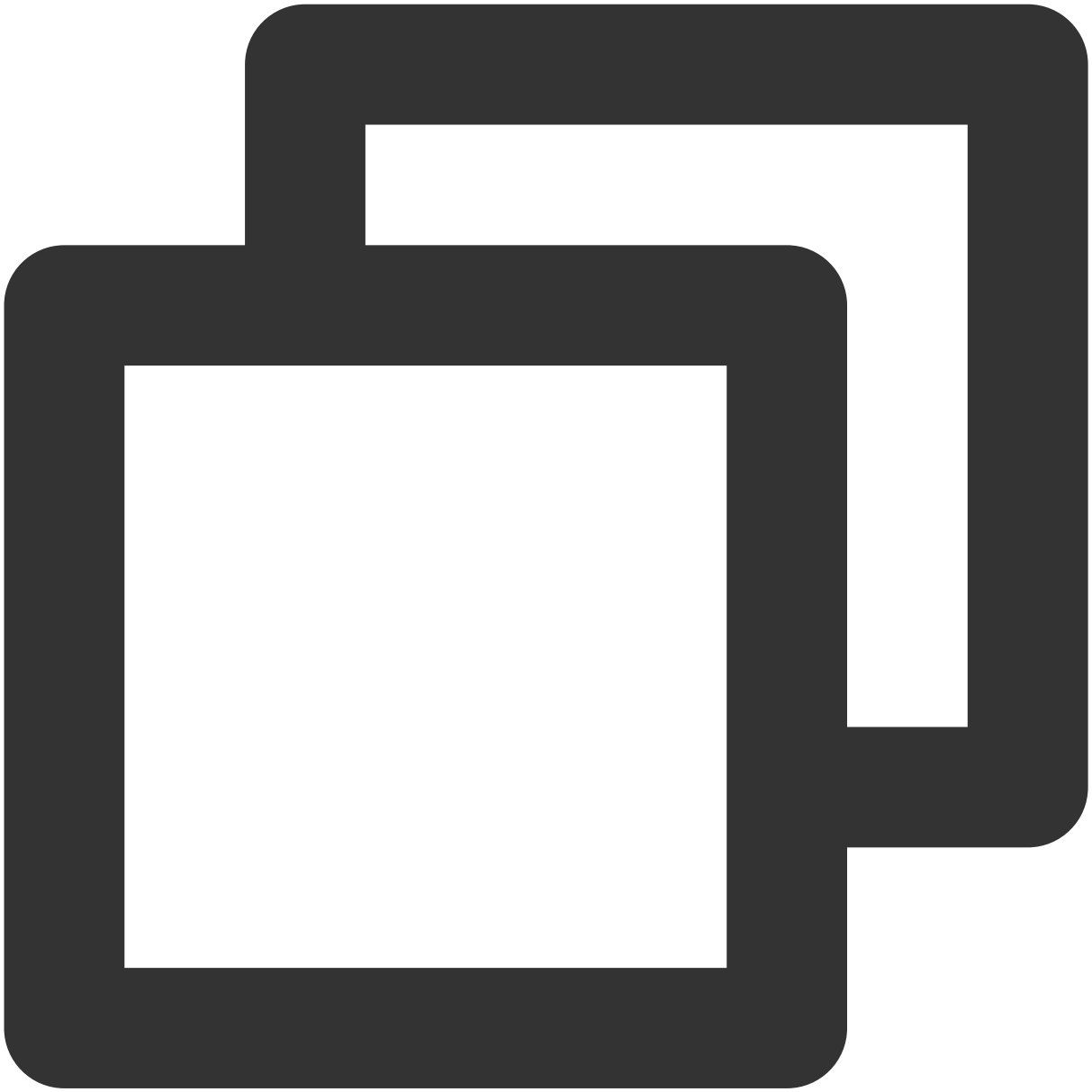
```
SMALLINT(<expr> T)
```

Supported engines: SparkSQL and Presto

Usage instructions: Force the type to be converted to smallint.

Return type: smallint

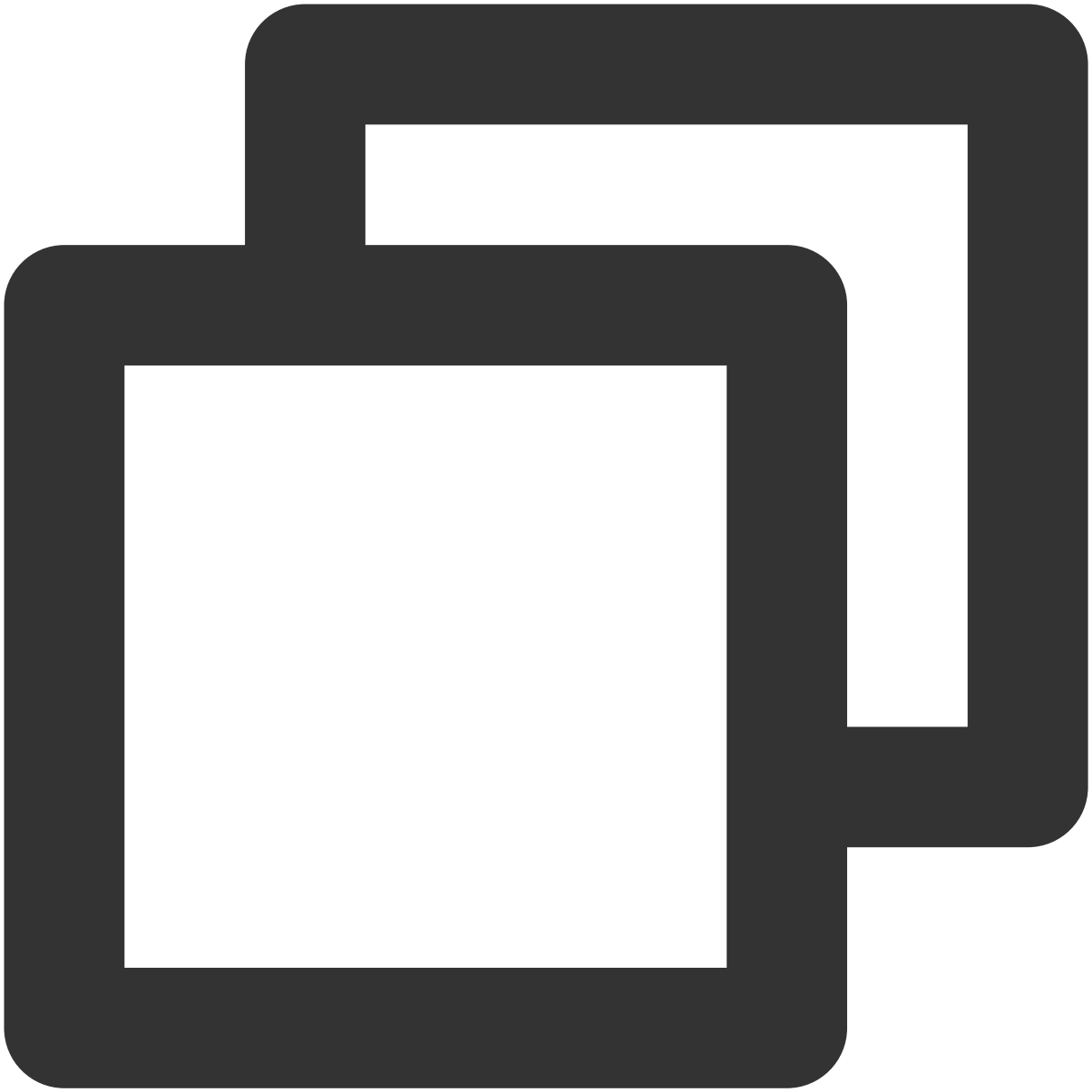
Example:



```
select typeof(smallint(1));
 smallint
```

STRING

Function statement:



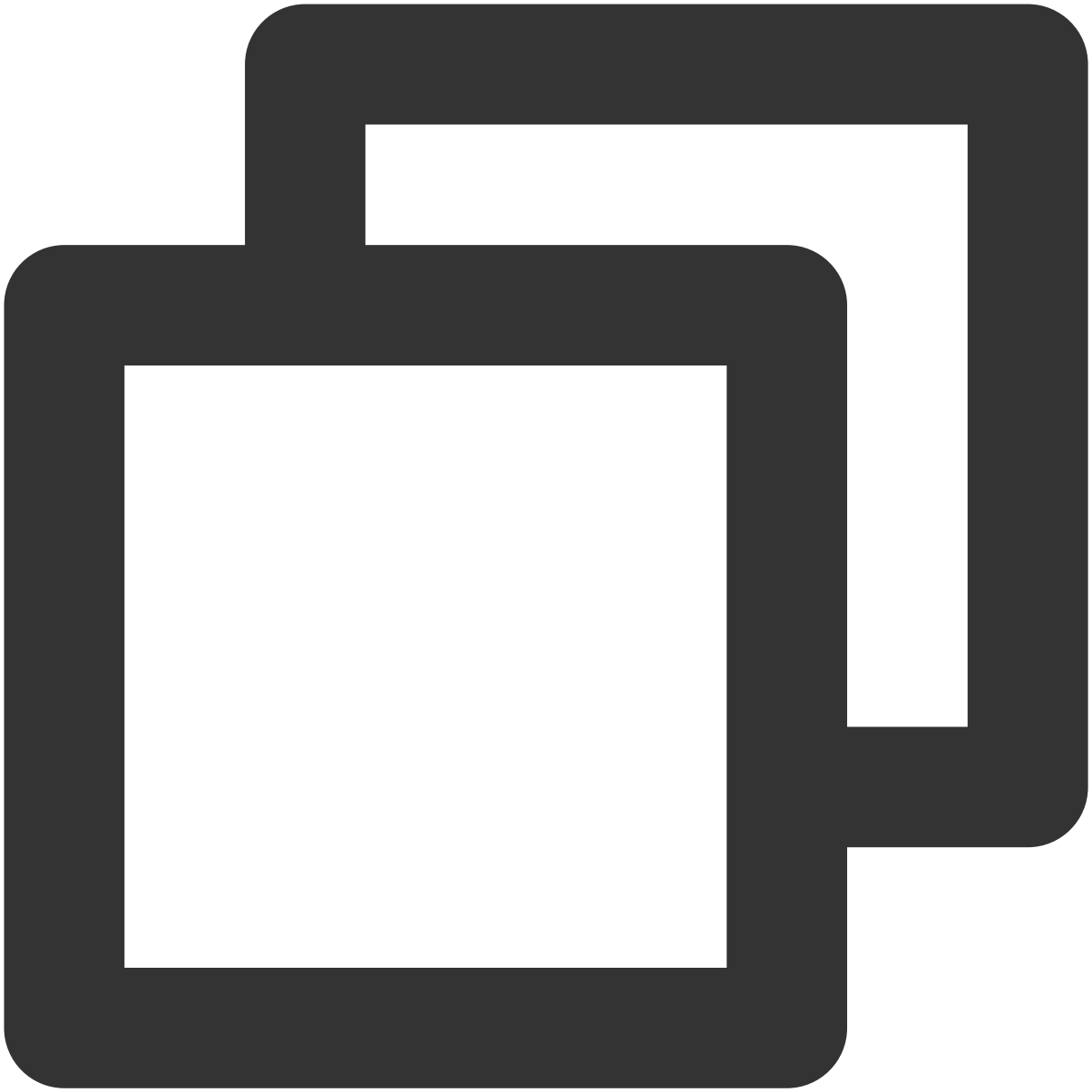
```
STRING(<expr> T)
```

Supported engines: SparkSQL and Presto

Usage instructions: Force the type to be converted to string.

Return type: string

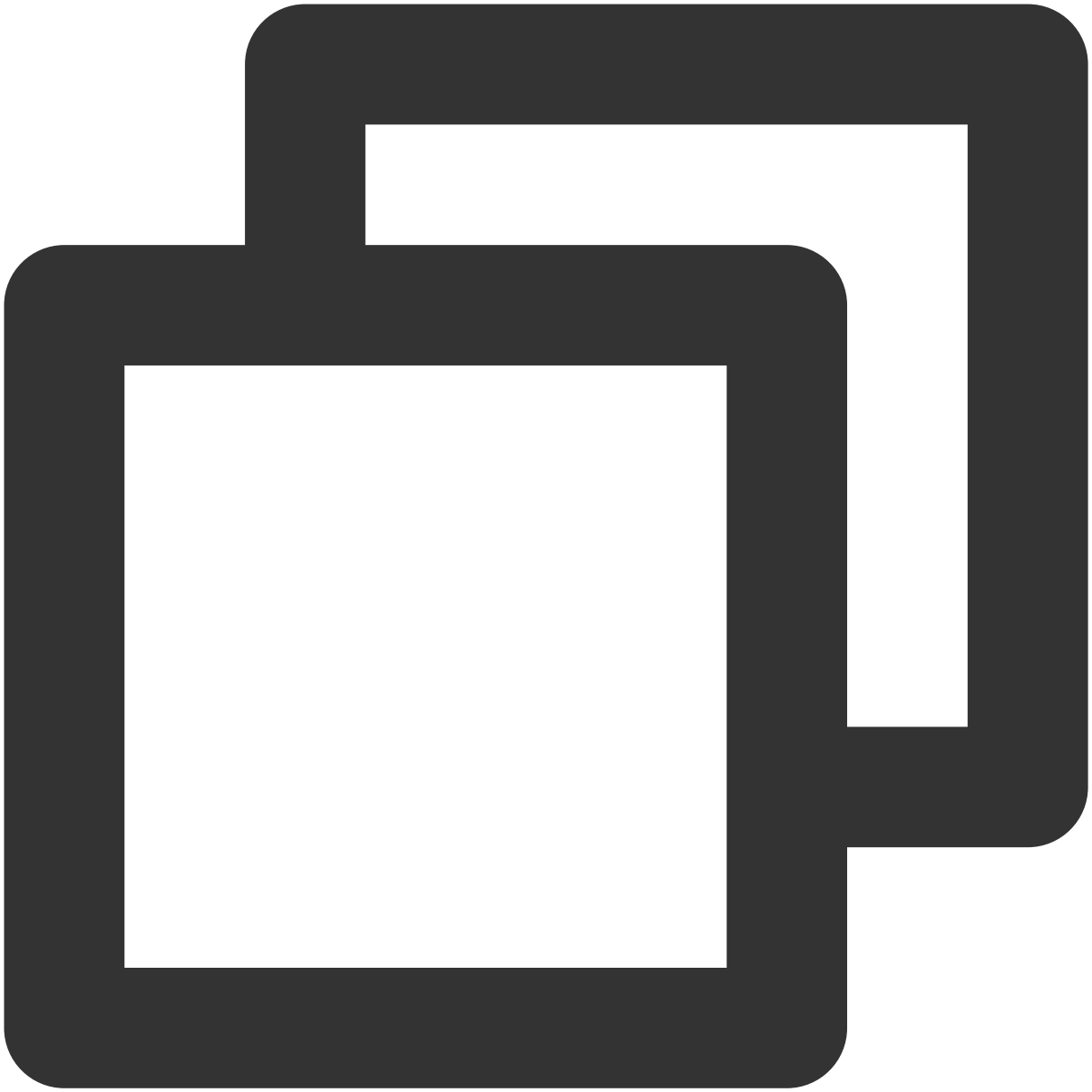
Example:



```
> select typeof(string(1));
 string
```

TINYINT

Function statement:



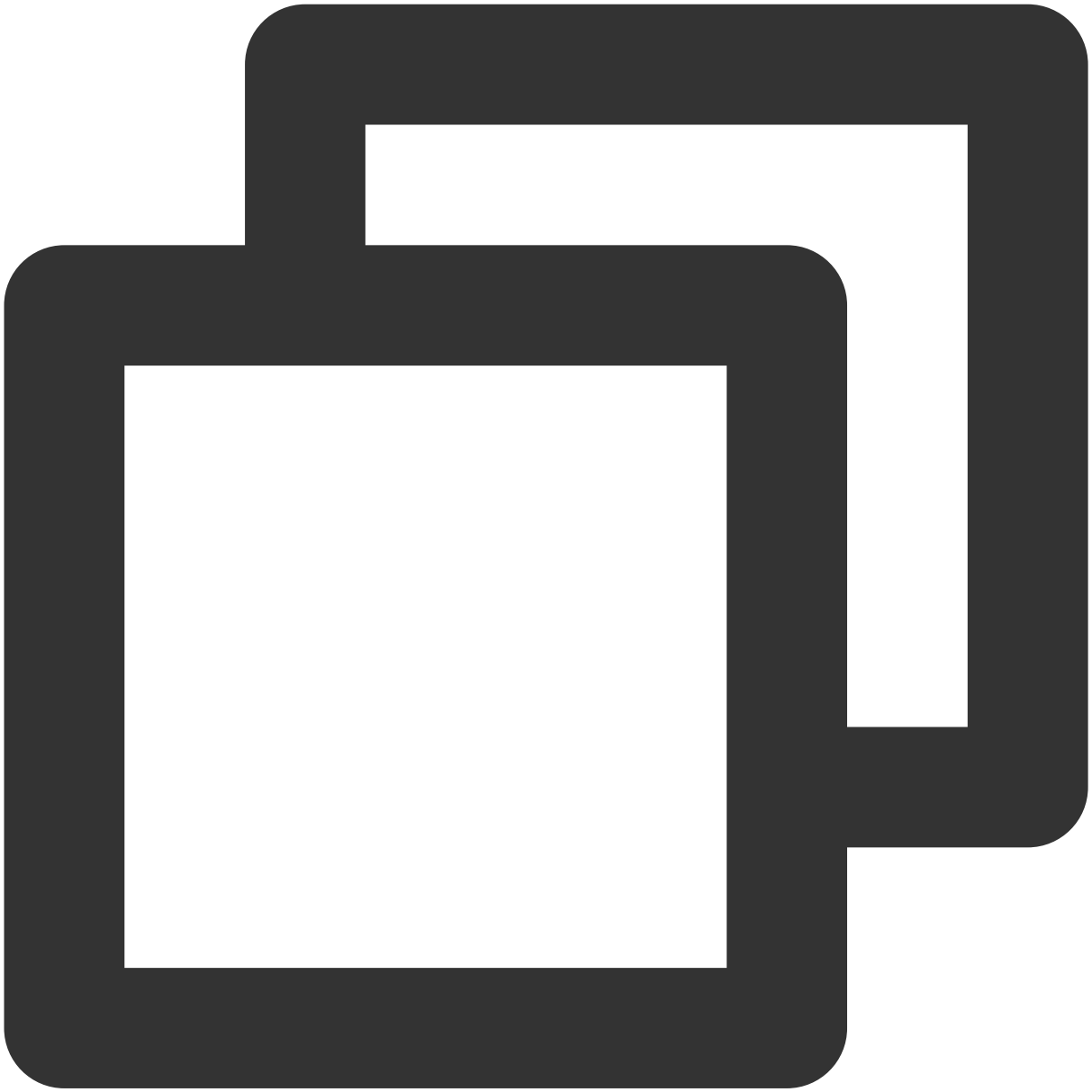
```
TINYINT(<expr> T)
```

Supported engines: SparkSQL and Presto

Usage instructions: Force the type to be converted to tinyint.

Return type: tinyint

Example:



```
> select typeof(tinyint(1));
 tinyint
```

DECIMAL

Function statement:

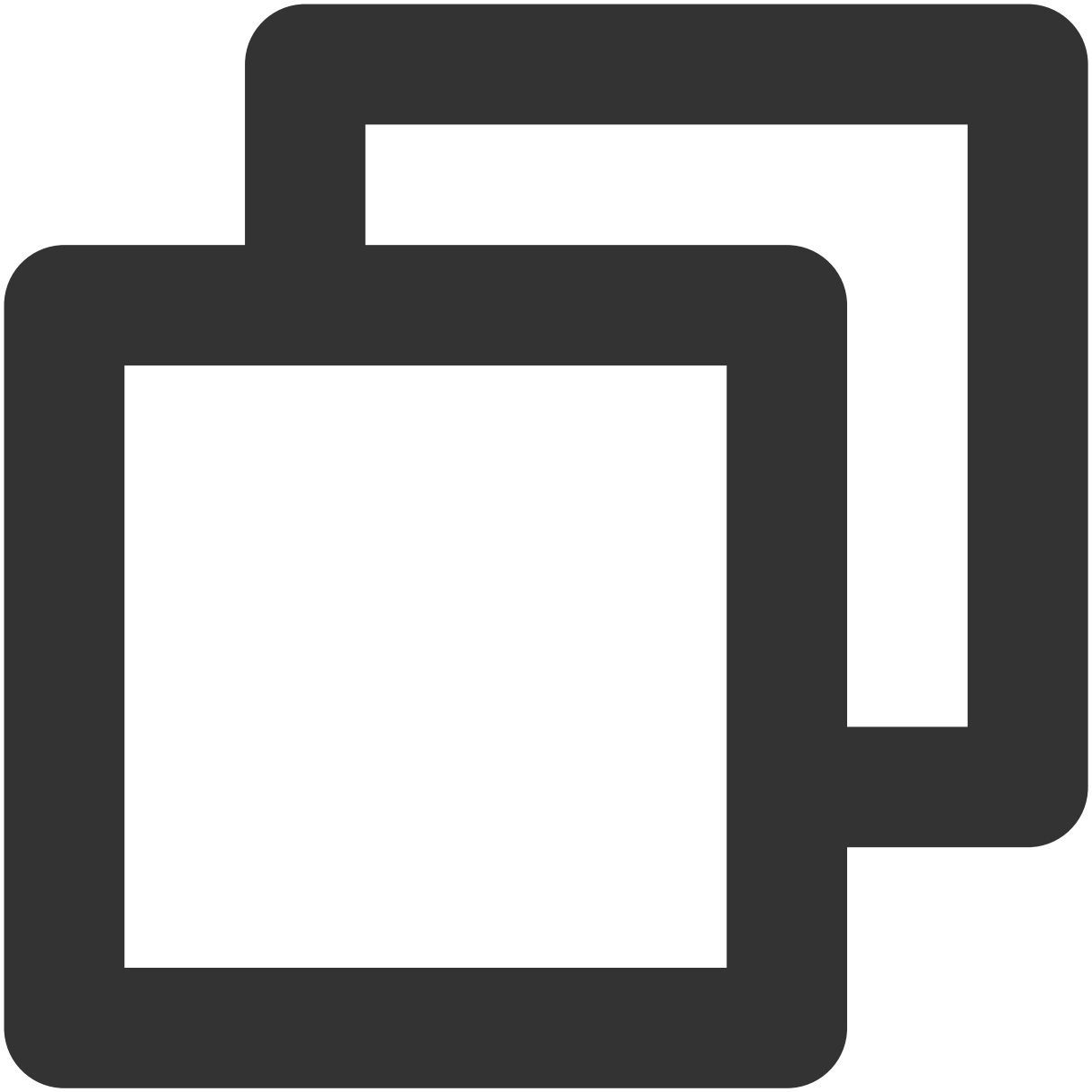
```
DECIMAL(<expr> T)
```

Supported engines: SparkSQL and Presto

Usage instructions: Force the type to be converted to decimal.

Return type: decimal

Example:



```
> select typeof(decimal(1));
 decimal(10, 0)
```

GET_IDCARD_BIRTHDAY

Function statement:

```
GET_IDCARD_BIRTHDAY(<string> idcardno)
```

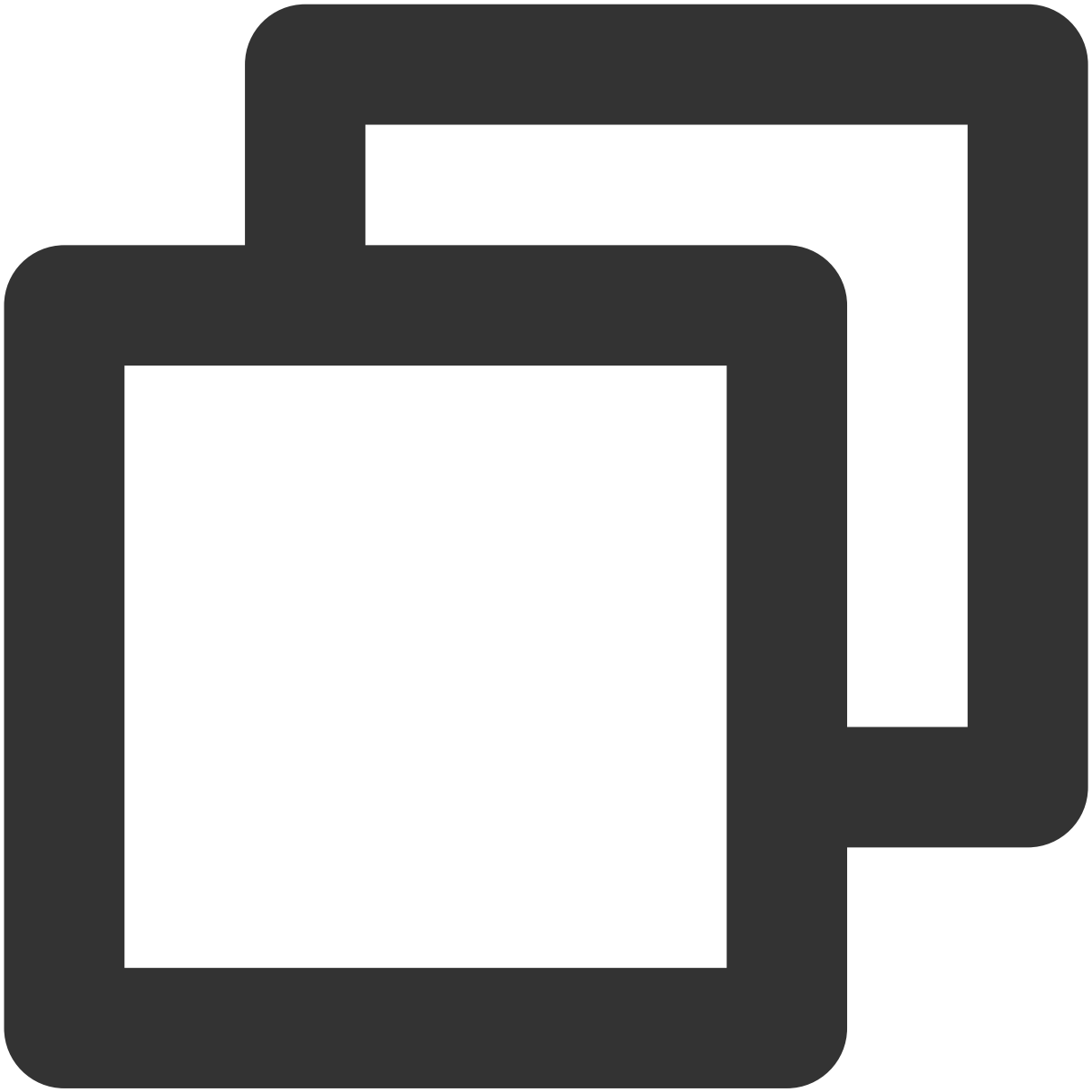
Supported engine: SparkSQL

Usage instructions: Obtain the birth date from the ID card number.

idcardno: Required, string type. Must be a 15-digit or 18-digit ID card number, and its validity will be verified. Other non-null inputs are not allowed. Return null if the input is null.

Return type: date

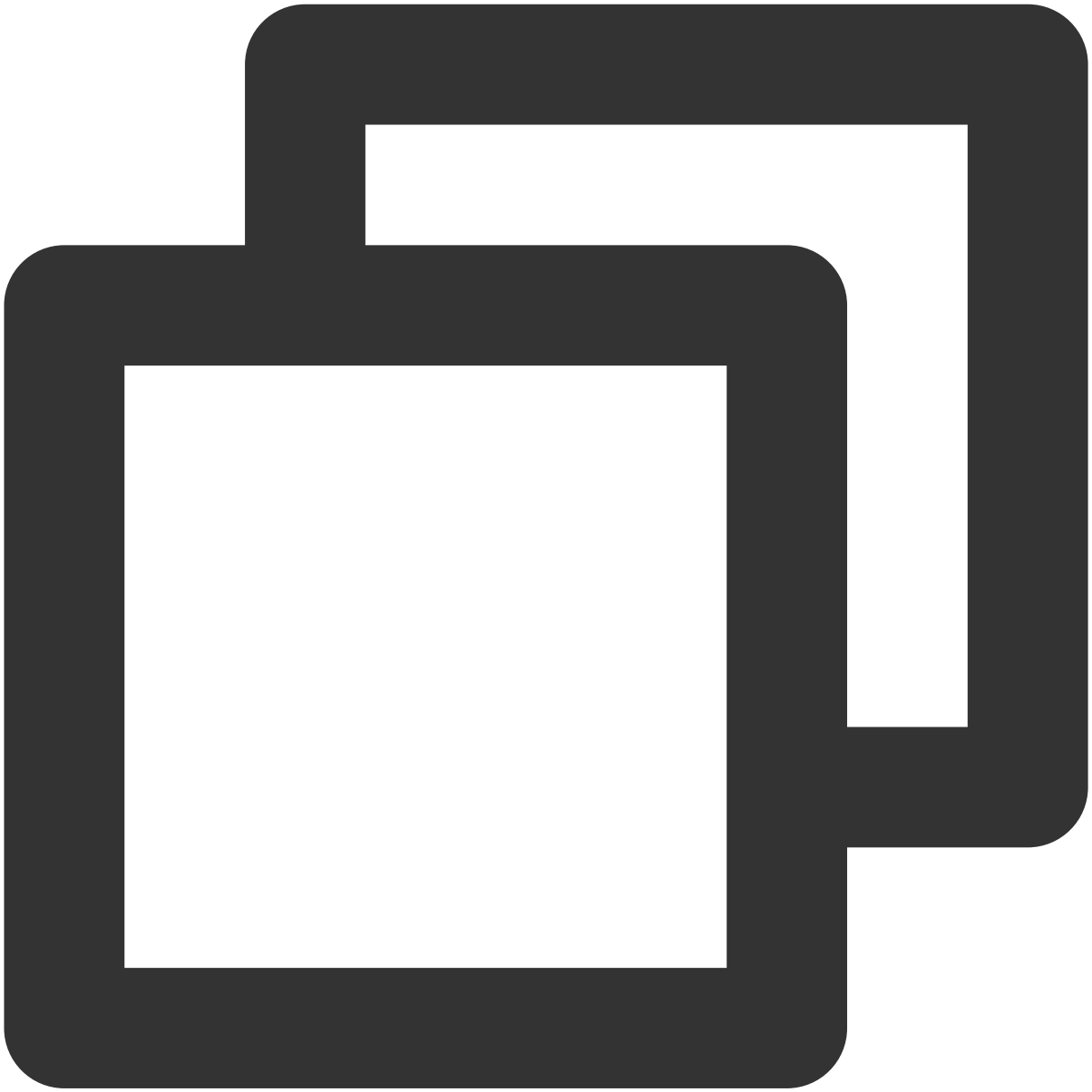
Example (The following test ID numbers are random samples and do not represent valid ID numbers.)



```
> SELECT get_idcard_birthday('421081199001011222');
1990-01-01
```

GET_IDCARD_SEX

Function statement:



```
GET_IDCARD_SEX(<string> idcardno)
```

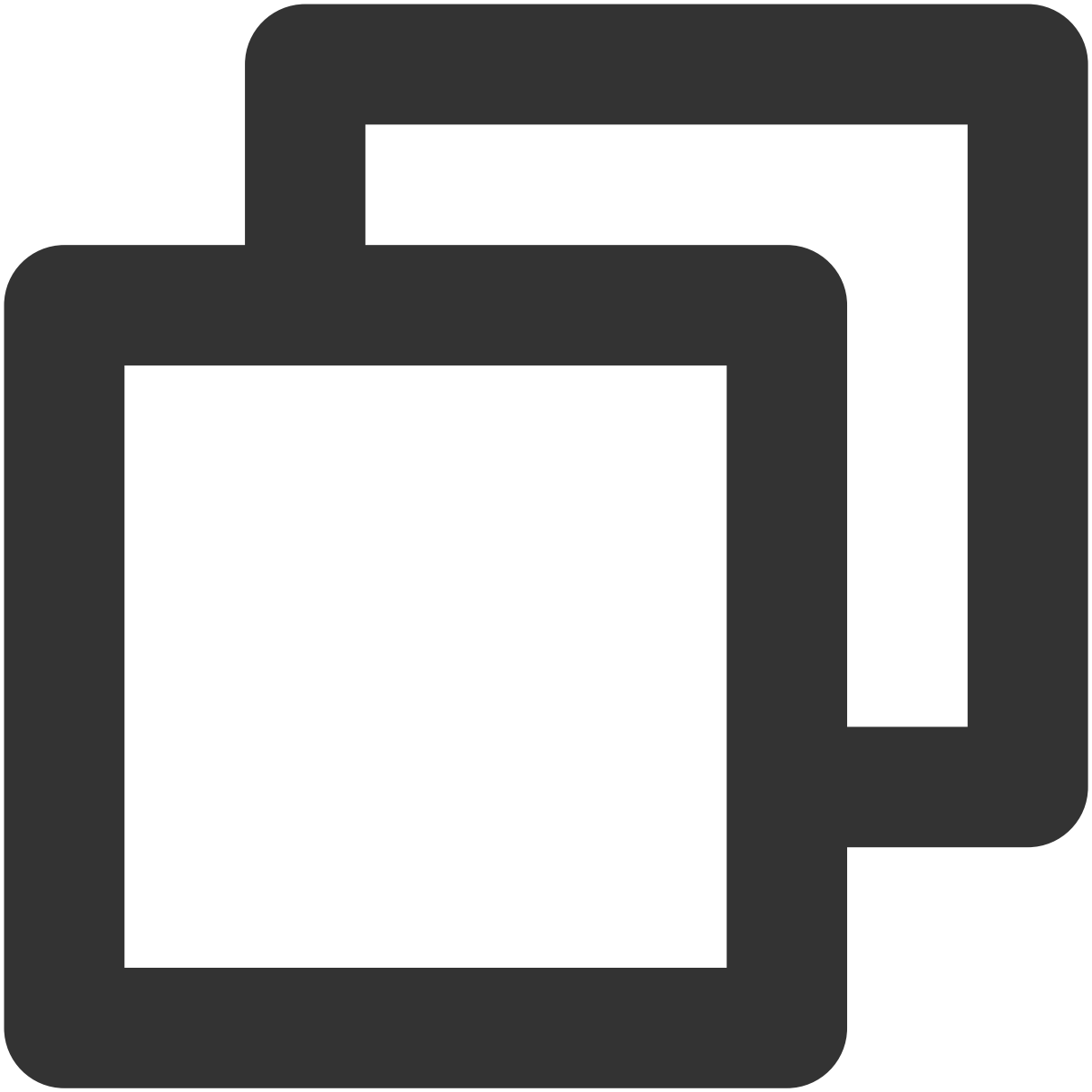
Supported engine: SparkSQL

Usage instructions: Obtain the gender from the ID card number.

idcardno: Required, string type. Must be a 15-digit or 18-digit ID card number, and its validity will be verified. Other non-null inputs are not allowed. Return null if the input is null.

Return type: string

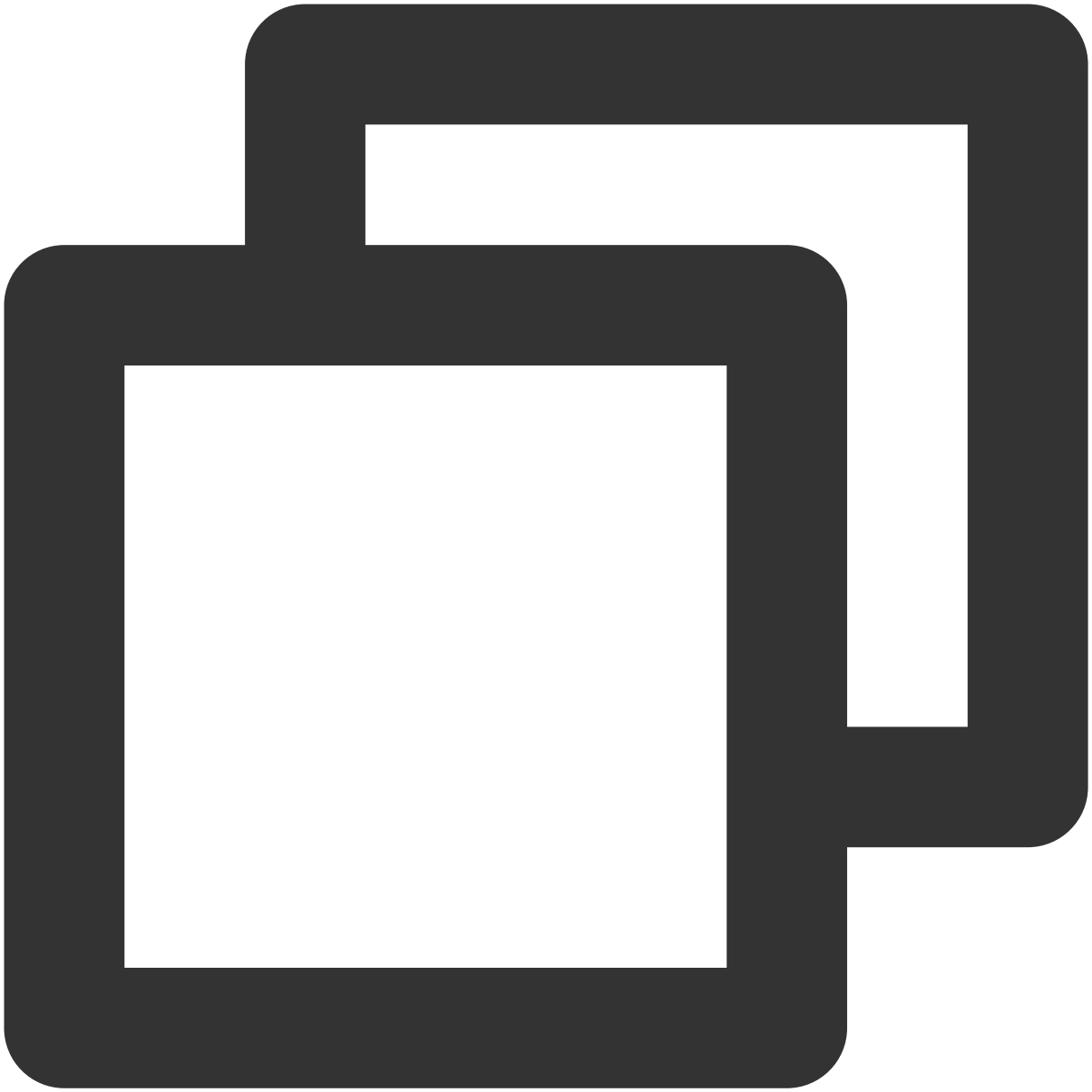
Example: (The following test ID numbers are random samples and do not represent valid ID numbers.)



```
> SELECT get_idcard_birthday('421081199001011222');
```

GET_IDCARD_AGE

Function statement:



```
GET_IDCARD_AGE(<string> idcardno)
```

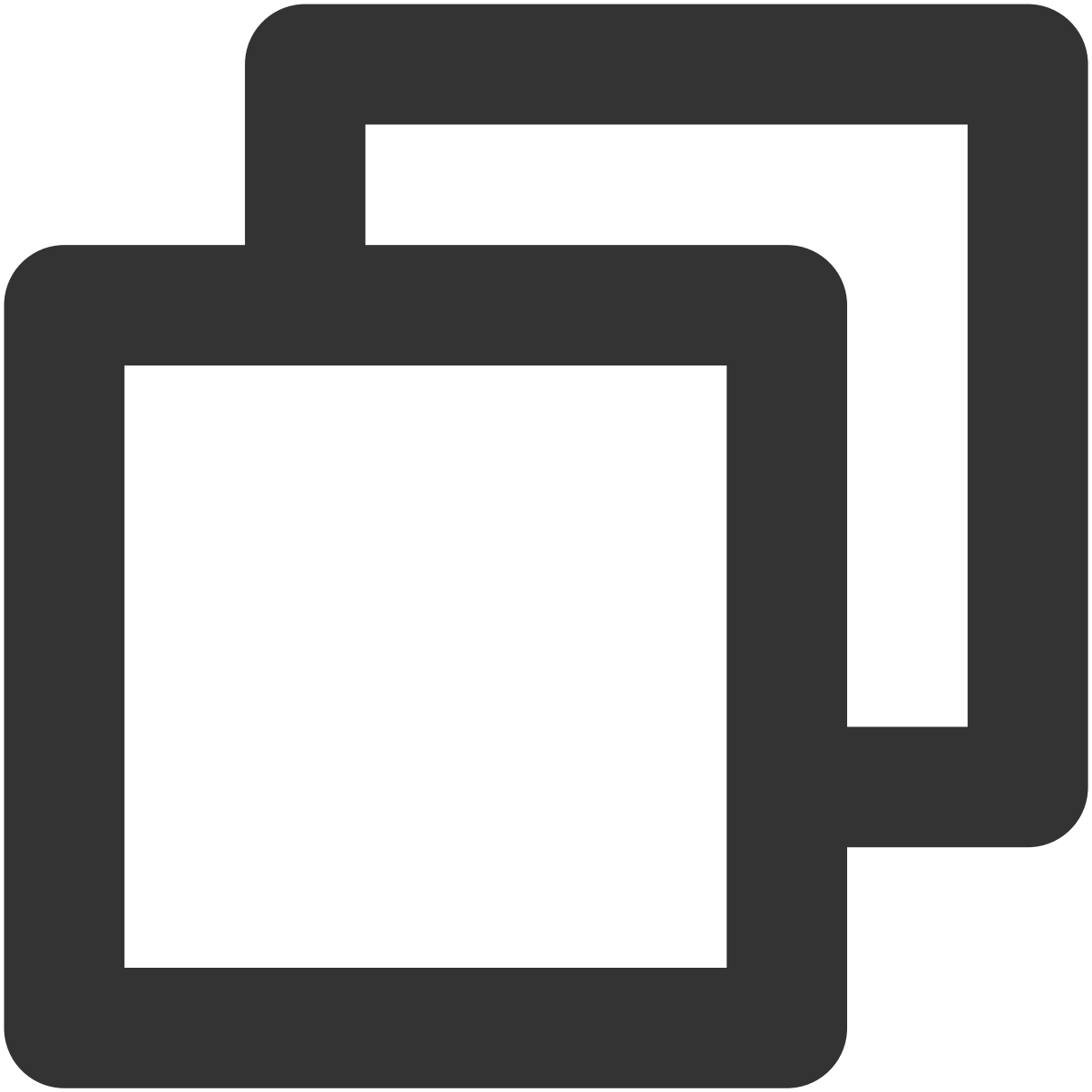
Supported engine: SparkSQL

Usage instructions: Obtain the age from the ID card number.

idcardno: Required, string type. Must be a 15-digit or 18-digit ID card number, and its validity will be verified. Other non-null inputs are not allowed. Return null if the input is null.

Return type: int

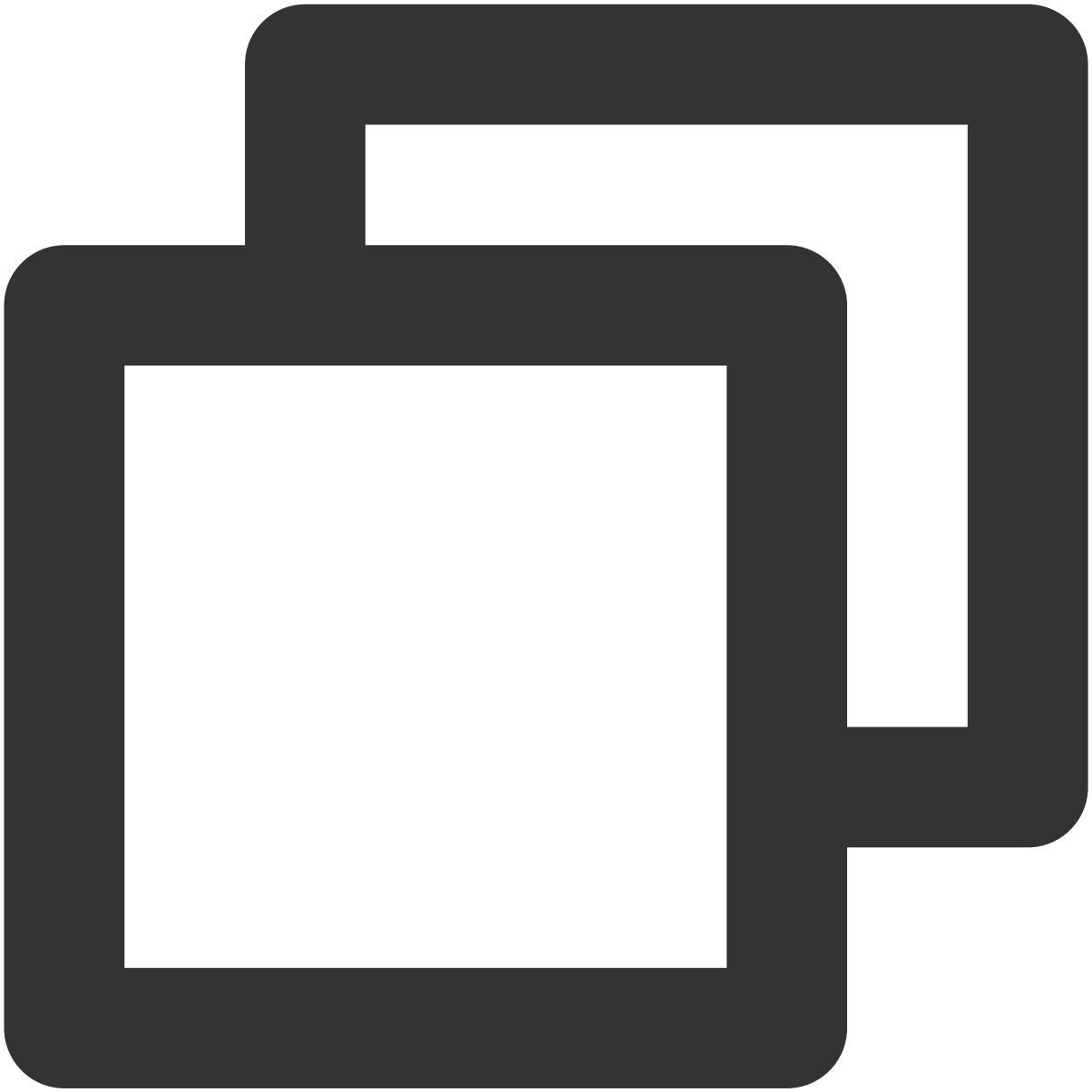
Example (The following test ID numbers are random samples and do not represent valid ID numbers.)



```
> SELECT get_idcard_age('421081197001021233');
53
```

MAX_PT

Function statement:



```
MAX_PT(<const string> tableFullName)
```

Supported engine: SparkSQL

Usage instructions: Retrieve the maximum value in the specified partitioned table.

tableFullName: Required. It is of string type. Must be a constant value, otherwise an error will occur. tableFullName consists of three parts: catalog.db.table. If catalog and db are omitted, the current session's configuration will be used by default, but it is recommended to specify the full name.

An error will occur if the input table is not a partitioned table.

This function only returns the maximum partition that **has data**. Since data presence involves the metadata center, **for external tables, ensure that you have executed the ANALYZE statement** to report the partition's statistics to the metadata.

Partition values are sorted in dictionary order to get the maximum value. When there are multiple levels of partitions, only the **first-level partition** is sorted.

Return type: string

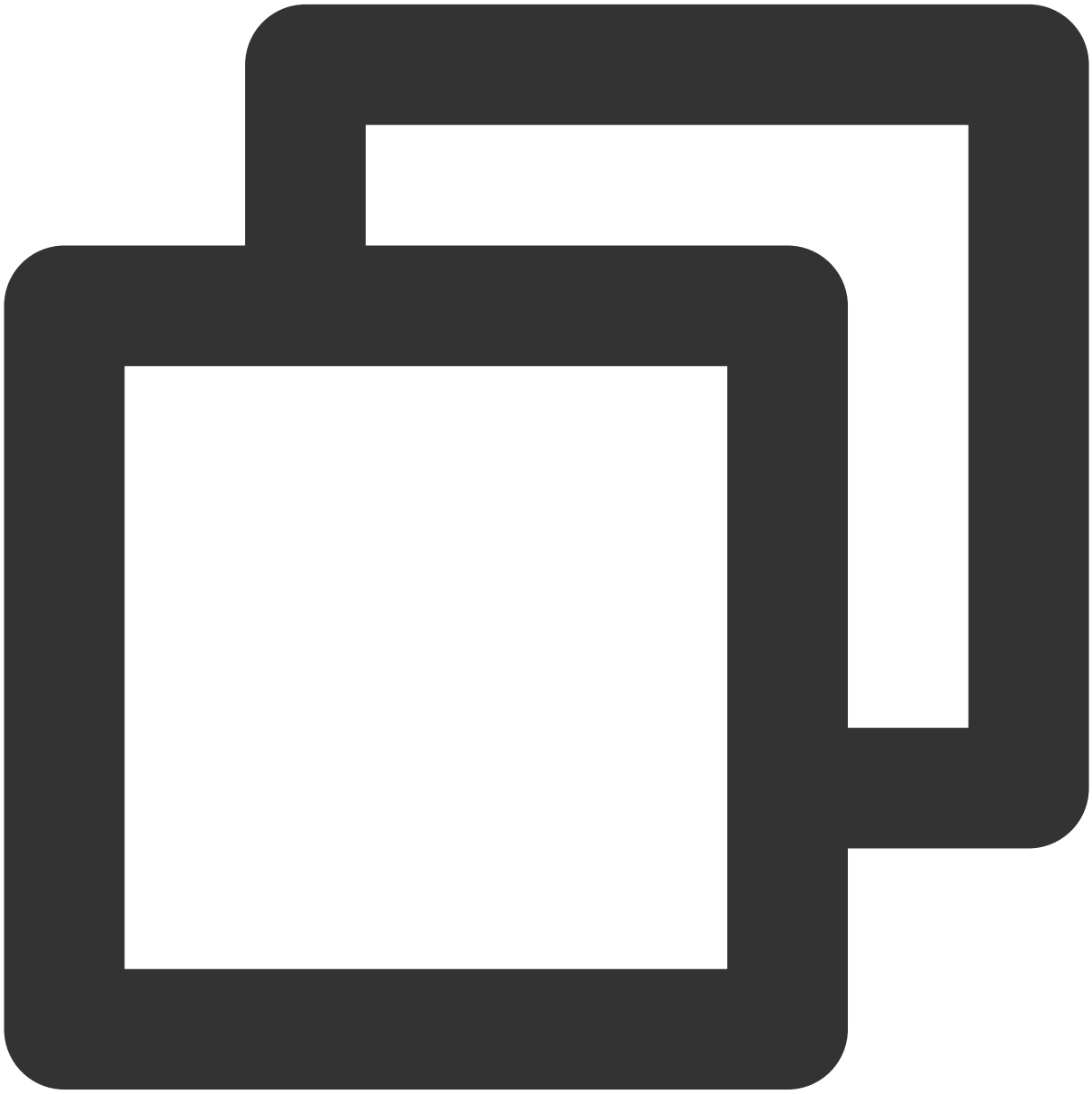
Example:

```
> SELECT max_pt('test.tableName');
20231024
```

```
> select * from test.tableName where dt=max_pt('test.tableName');
Equivalent to select * from test.tableName where dt='20231014';
```

TRANS_ARRAY

Function statement:



```
TRANS_ARRAY(<int> numKeys, <string> separator, <key1>, <key2>, ..., <col1>, <col2>,
```

Supported engine: SparkSQL

Usage instructions: Split and transpose the specified columns cols into multiple rows, while supporting the specification of certain columns keys to be used as keys for the transposition.

numKeys: int type, required. Indicates the number of key columns used for transposition. Must be greater than or equal to 0.

separator: String type, required. When cols is a string, it needs to be split based on the separator; when cols is an array, this parameter can be filled arbitrarily.

keys: Any type, the number is determined by numKeys.

cols: String type or array type. All specified columns, except keys, are considered cols, which are the columns to be split and transposed.** All types of cols must be the same, meaning they should either all be strings or all be arrays. Alternatively, cols can also support a combination of strings and string arrays. If there are no cols, only one row will be output.

Note 1: The sum of the quantities of keys and cols must be greater than 0.

Note 2: When the lengths of the split cols are not equal, the number of rows after the final transpose is based on the longest column, with other columns padded with NULL.

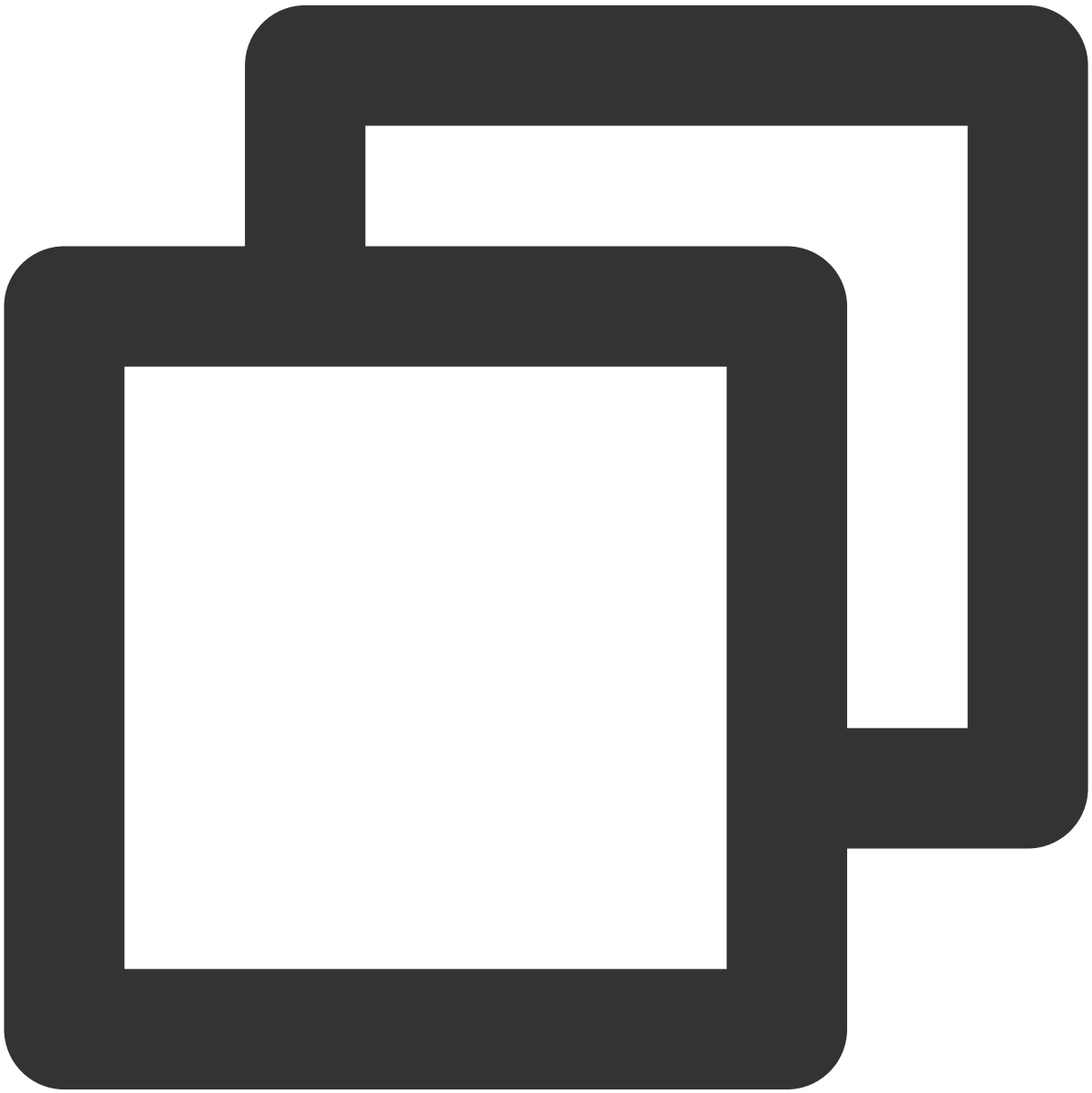
Return type: any type The type of keys remains unchanged, and the type of cols is the element type of the string or array.

Example:

```
> SELECT trans_array(1, ',', key, trans1, trans2) as (key, col1, col2) from values
1 2 4
1 3 5
> SELECT trans_array(0, ',', key, trans1, trans2) as (key, col1, col2) from values
1 2 4
NULL 3 NULL
> SELECT trans_array(3, ',', key, trans1, trans2) as (key, col1, col2) from values
12,3 [4,5]
```

TRANS_COLS

Function statement:



```
TRANS_COLS(<int> numKeys, <key1>, <key2>, ..., <col1>, <col2>, ...)
```

Supported engine: SparkSQL

Usage instructions: Transpose the specified multiple columns cols into multiple rows, while also supporting the specification of certain columns keys as the keys for the transposition.

numKeys: int type, required. Indicates the number of keys columns used for transposition. Must be greater than or equal to 0.

keys: Any type, the number is determined by numKeys.

cols: Any type, all specified columns excluding keys are considered as cols, i.e., the columns to be transposed. **All cols must be of the same type**. When there are no cols, only one row will be output.

Note: The sum of the quantities of keys and cols must be greater than 0.

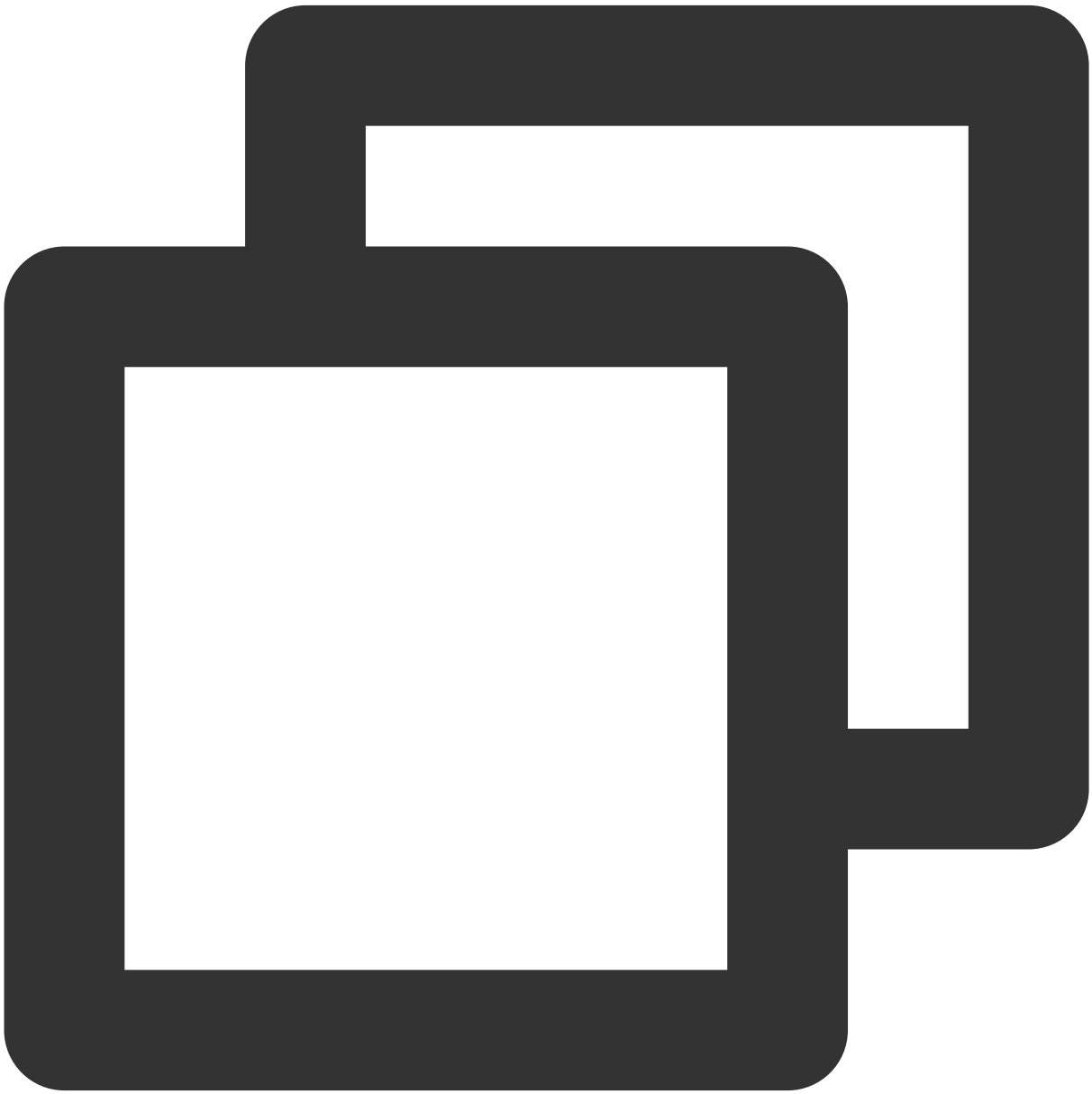
Return type: Any type The output will include a column idx, indicating the index of the row among all transposed rows. The type of keys remains unchanged, and the type of cols remains unchanged.

Example:

```
> SELECT trans_cols(1,sid,ip1,ip2) as (idx, sid, ip) from values ('s1','0.0.0.0','1
1 s1 0.0.0.0
2 s1 1.1.1.1
```

SAMPLE

Function statement:



```
SAMPLE(<int/long> totalParts[, <int/long> pointParts][, <col1>, <col2>, ...])
```

Supported engine: SparkSQL

Usage instructions: Sampling function. Divide the data globally or by the hash value of specified columns into totalParts parts and select the specified pointParts part to return.

totalParts: int/long type, required. Indicates the total number of partitions needed.

pointParts: int/long type, optional. Indicates which part of the data to return. The default value is 1 and must be less than or equal to totalParts.

cols: Any type, optional. You can specify any number of columns as keys for dividing the data. The data will be partitioned based on the hash values of these columns.

Note 1: When cols are not specified, or when all the specified cols are NULL, to avoid data skew, **random values will be used instead of hash values** for calculation. When cols are specified, as long as at least one column is not NULL, it will affect the hash value calculation and partitioning result.

Note 2: Since the seeds for the hash value and random value are fixed, if the data order remains consistent, the sampled data within a single partition will also be consistent upon repeated executions. In scenes involving small file merges, the sampling order may change. It is recommended to specify columns for sampling in such cases.

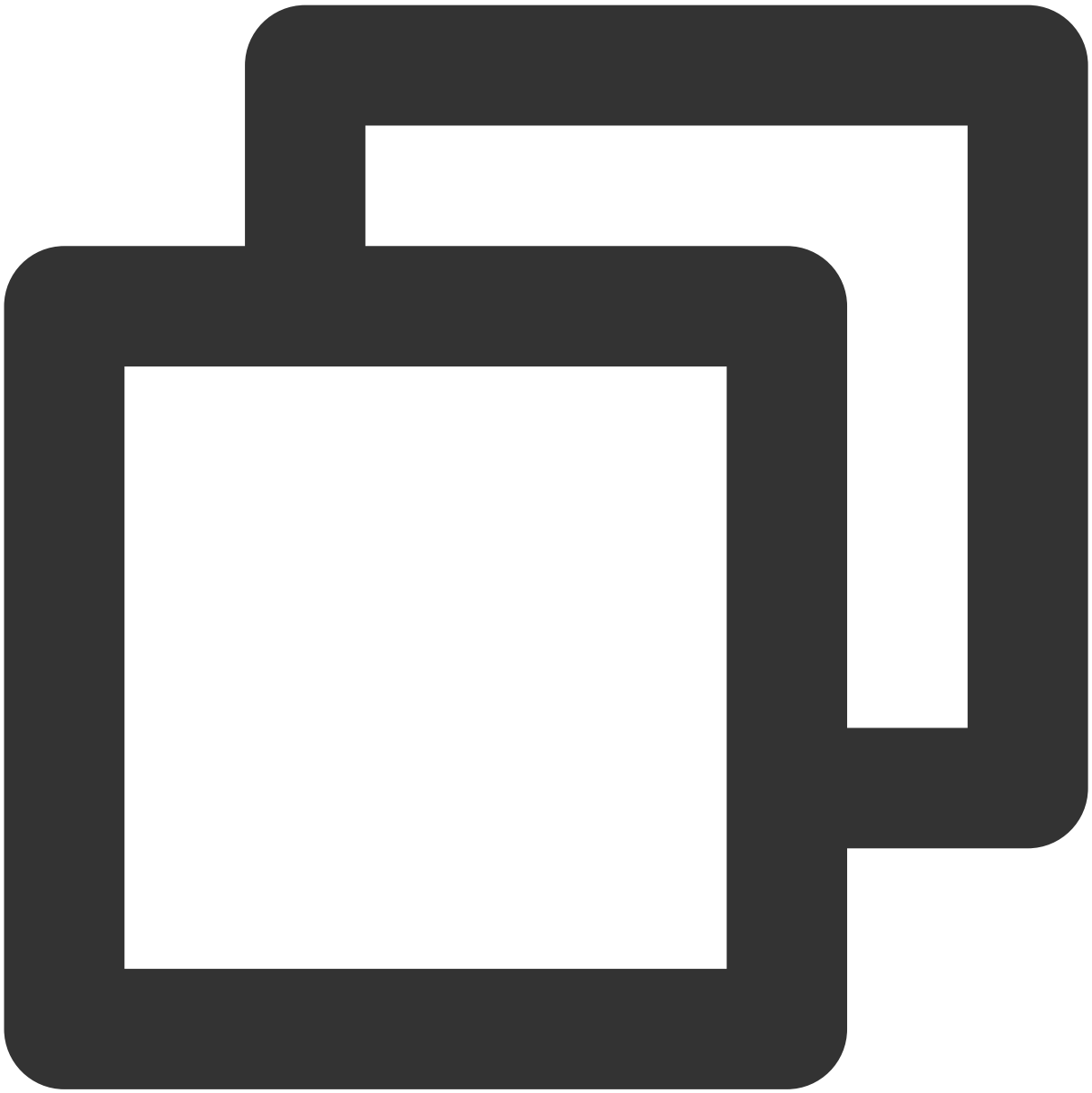
Return type: boolean, where true indicates the data has been sampled, and false indicates the data has not been sampled.

Example:

```
> select * from values (1, 'm1'), (2, 't2'), (3, 'z3') as tab(dt, uid) where sample
2 t2
→ select * from values (1, 'm1'), (2, 't2'), (3, 'z3') as tab(dt, uid) where sample
2 t2
3 z3
```

TO_CHAR

Function statement:



```
TO_CHAR(<boolean|int|long|double|decimal> data [, <string> format])
TO_CHAR(<date|timestamp> data, string format)
```

Supported engine: SparkSQL

Usage instructions 1: Format numbers according to the specified template. The format can be omitted, which directly converts the number to a string. Otherwise, the format is defined as follows:

'0' or '9': Placeholder for digits. At the beginning of the string, if 0 is used as a placeholder, it replaces missing digits with 0. If 9 is used, it replaces missing digits with spaces. At the end of the string, if there are no digits, 0 is used as a

replacement.

'.' or 'D': Placeholder for the decimal point, which can appear only once.

',' or 'G': Comma, used as a placeholder for hundreds, millions, etc., and must be surrounded by digits 0-9.

'$': USD sign, which can appear only once.

'S' or 'MI': Placeholder for negative and positive signs, which can only appear at most once at the beginning or end of the string.

'PR': It is only allowed to appear once at the end of the string. When the input data is negative, it converts the number to positive and adds angle brackets.

Usage instructions 2: Format the date or time according to the specified template. The format cannot be omitted and is defined as follows:

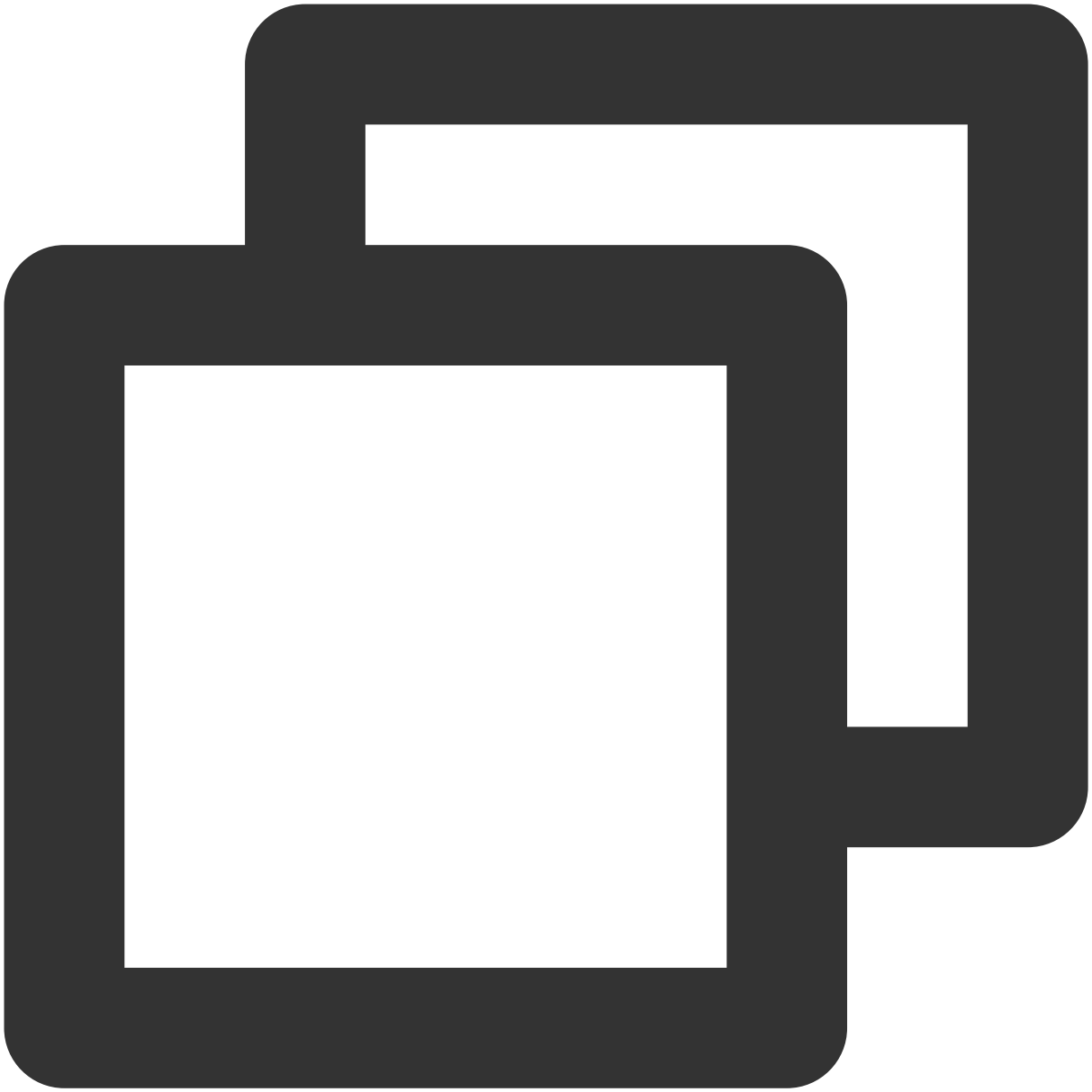
A template composed of any characters from yyyyMMdd HH:mm:ss.SSS.

Detailed reference: https://spark.apache.org/docs/latest/sql-ref-datetime-pattern.html

Usage instructions 3: TO_CHAR supports implicit conversion of the type of the first parameter. When the first parameter is of string type, the function will implicitly convert it to a timestamp and apply the date or time formatting logic based on the template.

Return type: string

Example:



```
> select to_char(124.23);
124.23
> select to_char(124.23, '00999.9999');
<space><space>124.2300
> select to_char(4124.23, '9,999.99');
4,124.23
> select to_char(-124.23, '999.99S');
124.23-
> select to_char(-4124.23, '$9,999.99PR');
<$4,124.23>
> select to_char(date '2016-12-31 12:34:56', 'yyyyMMddHHmmss');
```

```
20161231123456
```

Presto Built-in Functions

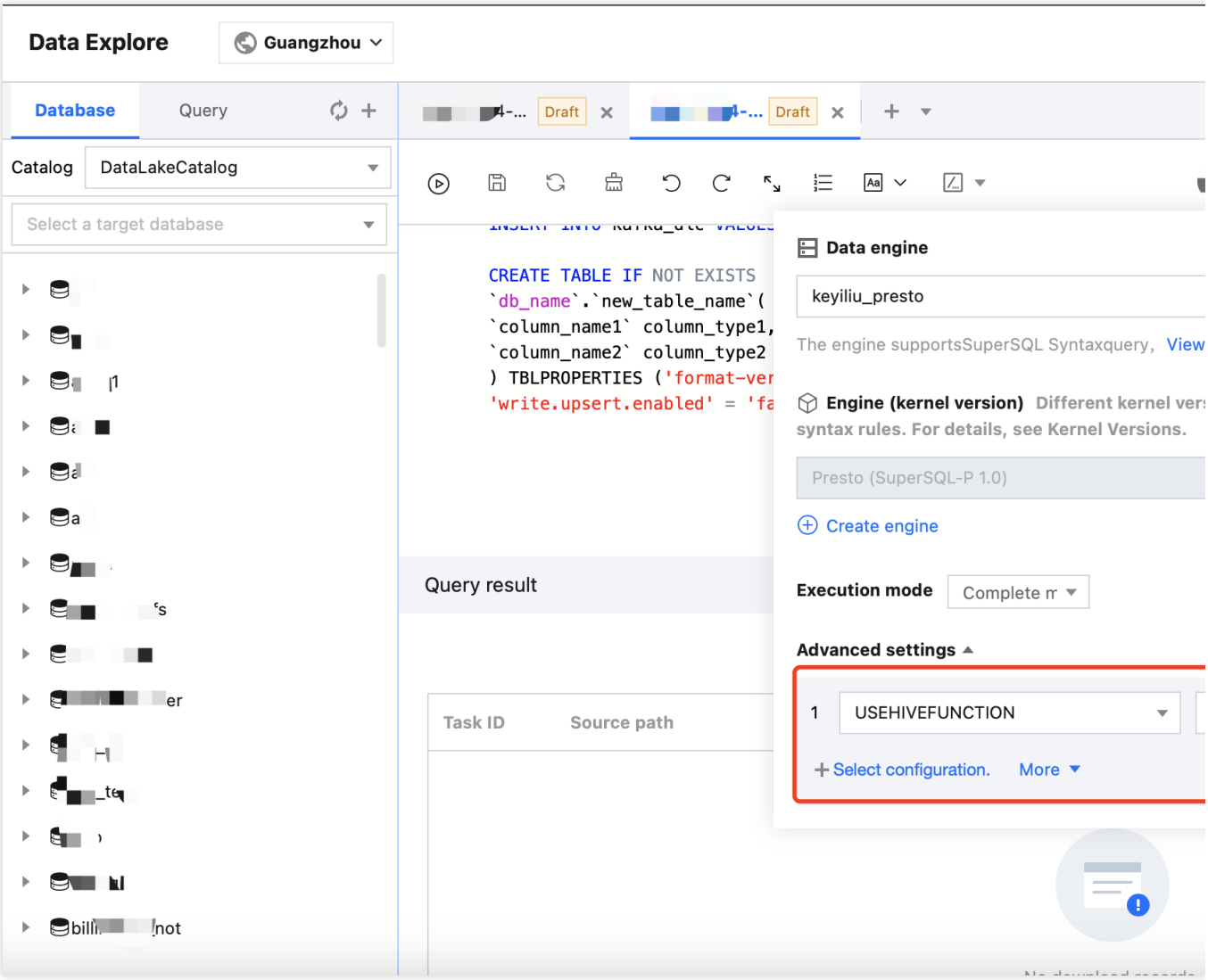
Last updated：2024-08-07 17:34:36

In addition to Unified Functions, DLC supports Presto built-in functions.

How to Enable Presto Built-in Function Applications

Method 1: Configuring Functions for the Data Engine in Data Exploration

1. Log in to the DLC Console and select the service region.
2. Enter Data Exploration and select the data engine. When the engine kernel is Presto, you can select the parameter **USEHIVEFUNCTION** in the advanced configuration. Set this parameter to **false** to use Presto built-in functions for SQL tasks with this data engine.

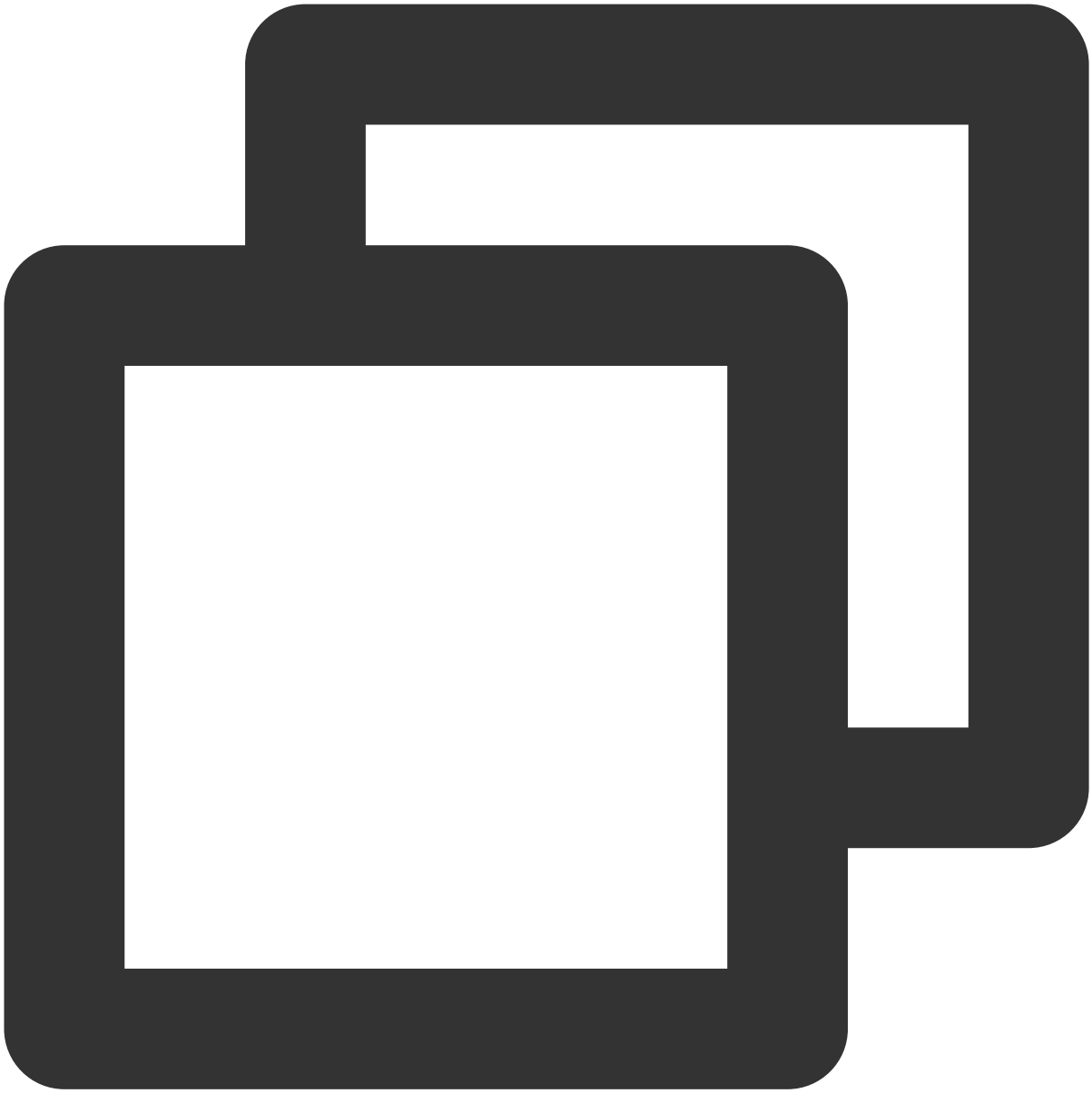


Caution

In the current query session, all query tasks using this data engine can use Presto built-in functions.

Method 2: Adding Parameters in the SQL Statement

If you want a specific SQL task to call Presto built-in functions, you can add configuration information in the SQL task. Example:



```
SELECT/*+OPTIONS('useHiveFunction'='false')*/prestofunc(xx)
```

Method 3: Adding Configuration Parameters When Using API

Set kv in the config of the task struct with useHiveFunction=false.

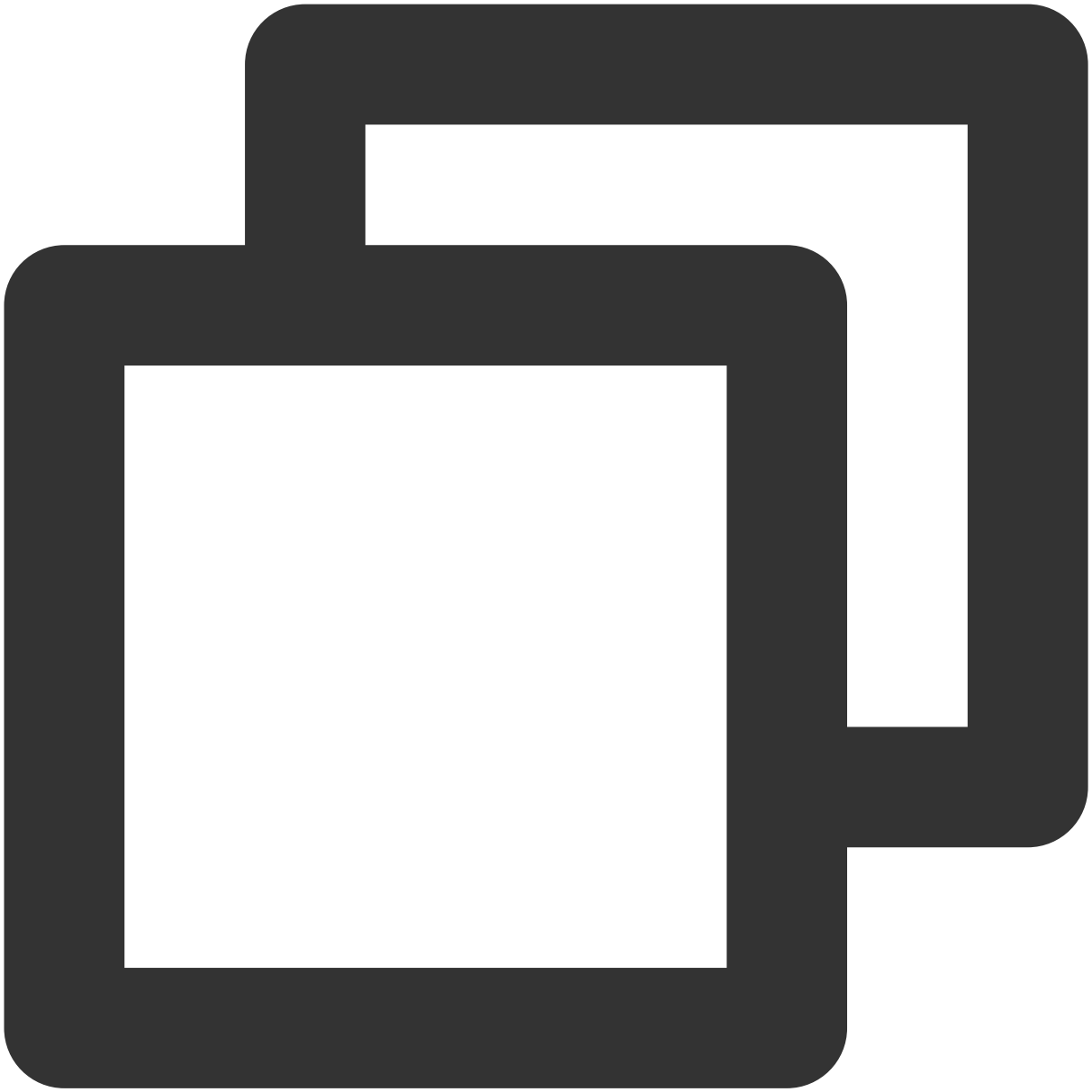
SQL Task

SQL Query Task

Referenced by the following interfaces: CreateTask。

Name	Type	Required	Description
SQL	String	Yes	Base64-enc Example val
Config	Array of KVPair	No	Task configu

Example:



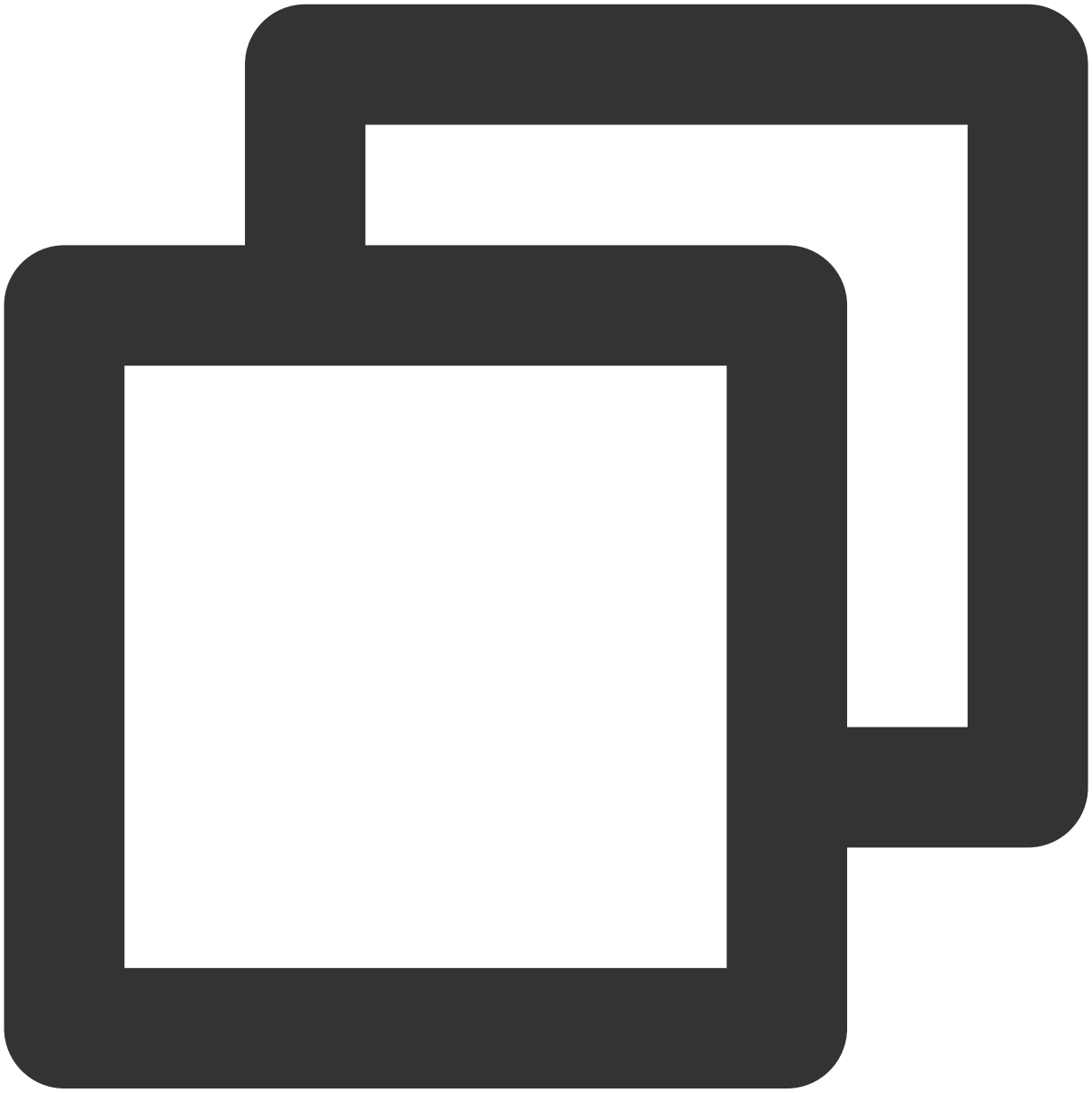
```
Stringstatement="SELECTdate_add('week',3,TIMESTAMP'2001-08-2203:04:05.321')";
TasksInfotask=newTasksInfo();
task.setTaskType("SQLTask");
task.setSQL(Base64.getEncoder().encodeToString(statement.getBytes()));
//Add the following parameter configuration.
KVPairpair=newKVPair();
pair.setKey("useHiveFunction");
pair.setValue("false");
task.setConfig(newKVPair[]{pair});

CreateTasksRequestrequest=newCreateTasksRequest();
```

```
request.setDatabaseName("");
request.setDataEngineName(PRESTO_ENGINE);
request.setTasks(task);
```

See the API documentation Task Creation.

Method 4: Adding Parameters During Task Creation Using JDBC

Add the `&presto.USEHIVEFUNCTION=true` parameter in the JDBCURL path or use `info.setProperty("presto.USEHIVEFUNCTION","false");`.



```
Connectionconnection=
```

```
DriverManager.getConnection("jdbc:dlc:dlc.tencentcloudapi.com?task_type=SQLTask&xOm
Or
info.setProperty("presto.USEHIVEFUNCTION","false");
info.setPropertv("user","");
info.setProperty("password","");
```

List of Supported Presto Built-in Functions

Mathematical Functions

Function Name	Return Value	Function Feature Description
abs(x)	Same as x	Return the absolute value of x.
cbrt(x)	double	Return the cube root of x.
ceil(x) | ceiling(x)	Same as x	Return the smallest integer greater than or equal to x. For example: ceil(2.2); -->3
cosine_similarity(x, y)	double	Return the cosine similarity of vectors x and y. For example: SELECT cosine_similarity(MAP(ARRAY['a'], ARRAY[1.0]), MAP(ARRAY['a'], ARRAY[2.0])); -->1.0
degrees(x)	double	Convert radians to degrees.
e()	double	Return the constant e.
exp(x)	double	Return e raised to the power of x
floor(x)	Same as x	Return the largest integer less than or equal to x. For example: floor(2.2); -->2
ln(x)	double	Return the natural logarithm of x.
log2(x)	double	Return the base-2 logarithm of x.
log10(x)	double	Return the base-10 logarithm of x.
mod(n, m)	Same as n	Return the remainder of n divided by m.
pi()	double	Return the constant Pi.
pow(x, p) | power(x, p)	double	Return x raised to the power of p.
radians(x)	double	Convert angle x (in degrees) to radians.

rand() | random()	double	Return a random value in the range 0.0 <= x < 1.0.
random(n)	Same as n	Return a random number between 0 and n (excluding n).
secure_rand() | secure_random()	double	Return a random encrypted value in the range 0.0 <= x < 1.0.
secure_random(lower, upper)	Same as lower	Return a random encrypted value in the range lower <= x < upper, where lower < upper.
round(x)	Same as x	Return the nearest integer to which x is rounded.
round(x, d)	Same as x	Return x rounded to d decimal places.
sign(x)	Same as x	Return the sign of x. If x > 0, return 1; if x < 0, return -1; if x = 0, return 0.
sqrt(x)	double	Return the square root of x.
truncate(x)	bigint	Return the number with the decimal part truncated. For example: truncate(4.9) --> 4
truncate(x, n)	double	Return x truncated to n decimal places. If n < 0, truncate to the left of the decimal point. For example: truncate(12.333, -1) --> 10.0; truncate(12.333, 0) --> 12.0; truncate(12.333, 1) --> 12.3;

Binary Functions

Function Name	Return Value	Function Feature Description
length(b)	bigint	Return the binary length of b in bytes.
concat(b1, ..., bN)	varbinary	Return the binary value by concatenating b1, ..., bN.
substr(b, start)	varbinary	Return the binary value extracted from b starting at position start. If start > 0, the extraction starts from the beginning; if start < 0, the extraction starts from the end.
substr(b, start, length)	varbinary	Return the binary value of length length extracted from b starting at position start. If start > 0, the extraction starts from the beginning; if start < 0, the extraction starts from the end.
to_base64(b)	varchar	Convert binary data b to a base64 string.
from_base64(string)	varbinary	Convert a base64 encoded string to binary data.

to_base64url(b)	varchar	Use URL-safe characters to convert binary data b to a base64 string.
from_base64url(string)	varbinary	Use URL-safe characters to convert a base64 encoded string to binary data.
from_base32(string)	varbinary	Convert a base32 encoded string to binary data.
to_base32(b)	varchar	Convert binary data b to a base32 string.
to_hex(b)	varchar	Convert binary data b to a hexadecimal string.
from_hex(string)	varbinary	Convert a hexadecimal encoded string to binary data.
lpad(b, size, padb)	varbinary	Return binary data with padb concatenated to the left of b until the length reaches size.
rpad(b, size, padb)	varbinary	Return binary data with padb concatenated to the right of b until the length reaches size.
crc32(b)	bigint	Calculate the cyclic redundancy check value of the expression using the CRC32 algorithm.
md5(b)	varbinary	Calculate the MD5 value of binary b.
sha1(b)	varbinary	Calculate the SHA-1 value of binary b.
sha256(b)	varbinary	Calculate the SHA-256 value of binary b.
sha512(b)	varbinary	Calculate the SHA-512 value of binary b.
xxhash64(b)	varbinary	Calculate the XXHash64 value of binary b.

Bitwise Functions

Function Name	Return Value	Function Feature Description
bit_count(x, bits)	bigint	Return the position of the bit in x.
bitwise_and(x, y)	bigint	Return the bitwise AND of x and y.
bitwise_not(x)	bigint	Return the bitwise NOT of x.
bitwise_or(x, y)	bigint	Return the bitwise OR of x and y.
bitwise_xor(x, y)	bigint	Return the bitwise XOR of x and y.
bitwise_left_shift(value, shift)	Same as input	Return the value of value left-shifted by shift bits.

		For example: bitwise_left_shift(TINYINT '7', 2) --> 28
bitwise_right_shift(value, shift)	Same as input	Return the value of value right-shifted by shift bits. For example: bitwise_right_shift(TINYINT '7', 2) -->1

String Functions

Function Name	Return Value	Function Feature Description
chr(n)	varchar	Return the Unicode character for n.
codepoint(string)	integer	Return the Unicode value of the string. For example: codepoint('0') -->48
concat(string1, ..., stringN)	varchar	Return the concatenated string of string1, ..., stringN.
length(string)	bigint	Return the length of the string.
hamming_distance(string1, string2)	bigint	Return the number of differing characters at corresponding positions in two strings of the same length. For example: hamming_distance(abc, art) -->2
levenshtein_distance(string1, string2)	bigint	Return the minimum number of edits (insertions, deletions, substitutions) required to convert one string into another. For example: levenshtein_distance('ab', 'abcde') -->3
lower(string)	varchar	Return the string converted to lowercase.
upper(string)	varchar	Return the string converted to uppercase.
ltrim(string)	varchar	Delete leading spaces from the string.
rtrim(string)	varchar	Return the string with trailing spaces deleted.
trim(string)	varchar	Return the string with all spaces deleted.
replace(string, search)	varchar	Delete all instances of the search character from the string.
replace(string, search, replace)	varchar	Replace the search string with the replace string. For example: replace('abcavc', ‘a', ‘c') -->'cbccvc'
reverse(string)	varchar	Return the string with its characters in reverse order. For example: reverse('abc') -->'cba'
lpad(string, size, padstring)	varchar	Concatenate the left side of the string with a

		padstring until the string reaches the specified size. If size is less than the length of the string, the string is truncated to the specified size.
rpad(string, size, padstring)	varchar	Concatenate the right side of the string with a padstring until the string reaches the specified size. If size is less than the length of the string, the string is truncated to the specified size.
split(string, delimiter)	array	Return an array of substrings split by the delimiter.
split(string, delimiter, limit)	array	Return an array of substrings split by the delimiter with a limit on the number of splits. If limit > 0, the last element contains the rest of the string. For example: select split('ab-cd-efg', '-', 2) --> ['ab','cd-efg']
split_part(string, delimiter, index)	varchar	Return the substring at the specified index from the array split by delimiter. The index starts at 1. If index > array length, return null. For example: select split_part('ab-cd-efg', '-', 2) -->'cd'
split_to_map(string, entryDelimiter, keyValueDelimiter)	map<varchar, varchar>	Return a map obtained by splitting the string using entryDelimiter and keyValueDelimiter. For example: select split_to_map('a:1,b:2', ',', ':') --> {'a':'1','b':'2'}
split_to_map(string, entryDelimiter, keyValueDelimiter, function(K, V1, V2, R))	map<varchar, varchar>	Return a map obtained by splitting the string using entryDelimiter and keyValueDelimiter. If there are duplicate keys, the value for each key is determined by the specified function. For example: split_to_map('a:1,b:2;a:3',',' , ':', (k, v1, v2) -> v1)-->{'a':'1','b':'2'}
substr(string, start)	varchar	Return the substring from the specified start position in the string. The position starts at 1. If start <0, counting starts from the end of the string.
substr(string, start, length)	varchar	Return the substring of the specified length starting from the start position in the string. The position starts at 1. If start <0, counting starts from the end of the string.
position(substring IN string)	bigint	Return the position of the first occurrence of substring in the string. The position starts at 1. If the position is not found, 0 is returned.

strpos(string, substring)	bigint	Return the position of the first occurrence of substring in the string from left to right. The position starts at 1. If the position is not found, 0 is returned. For example: strpos('abcdefg', 'a') -->1
strpos(string, substring, instance)	bigint	Return the position of the instance-th occurrence of substring in the string from left to right. If instance >0, the position starts at 1. If the position is not found, 0 is returned. For example: strpos('abcaefg', 'ab',2) -->0
strrpos(string, substring)	bigint	Return the position of the first occurrence of substring in the string from right to left. The position starts at 1. If the position is not found, 0 is returned. For example: strpos('abcdefg', 'a') -->7
strrpos(string, substring, instance)	bigint	Return the position of the instance-th occurrence of substring in the string from right to left. If instance >0, the position starts at 1. If the position is not found, 0 is returned. For example: strpos('abcaefg', 'a',2) -->0
to_utf8(string)	varbinary	Convert the string to a UTF-8 encoded binary number.
from_utf8(binary)	varchar	Convert the binary number to a UTF-8 encoded string, replacing invalid characters with the Unicode character U+FFFD.
from_utf8(binary, replace)	varchar	Convert the binary number to a UTF-8 encoded string, replacing invalid characters with the replace.

Date and Time Functions

Function Name	Return Value	Function Feature Description
current_date	date	Return the current date.
current_time	time	Return the current time with time zone.
current_timestamp	timestamp	Return the current timestamp with time zone.
current_timezone()	varchar	Return the current time zone.
date(x)	date	Return the current date.

last_day_of_month(x)	date	Return the last day of the current month.
from_unixtime(unixtime)	timestamp	Return the Unix timestamp. For example: from_unixtime(1475996660) -->2016-10-09 15:04:20.000
to_unixtime(timestamp)	double	Return the Unix timestamp of the timestamp.
from_unixtime(unixtime, string)	timestamp	Return the timestamp from Unixtime, with the string specifying the time zone. For example: from_unixtime(1617256617,'Asia/Shanghai') -->2021-04-01 13:56:57.000 Asia/Shanghai
from_iso8601_timestamp(string)	timestamp	Convert the string to a timestamp in ISO 8601 format.
from_iso8601_date(string)	date	Convert the string to a date in ISO 8601 format.
localtime	time	Return the current time.
localtimestamp	timestamp	Return the current timestamp.
now()	timestamp	Return the current timestamp with time zone.
to_iso8601(x)	varchar	Return the ISO 8601 formatted string of x, where x can be a date, timestamp, or timestamp with time zone.
date_trunc(unit, x)	Same as input	Return the time x truncated to the specified unit, which can be second, minute, hour, day, week, month, quarter, or year. For example: date_trunc('hour', TIMESTAMP '2020-03-17 02:09:30') --> '2020-03-17 02:00:00'
date_add(unit, value, timestamp)	Same as input	Return a new timestamp calculated by adding or subtracting the specified interval. A value >0 means adding, while a value <0 means subtracting. The unit can be millisecond, second, minute, hour, day, week, month, quarter, or year. For example: date_add('hour', 2, cast ('2023-07-20 20:22:22.022' as timestamp) -->'2023-07-20 22:22:22.022'
date_diff(unit, timestamp1, timestamp2)	bigint	Return the time difference between two timestamps in the specified unit. The unit can be millisecond, second, minute, hour, day, week, month, quarter, or year. For example: date_diff('hour', cast ('2023-07-01 22:22:22' as timestamp) , cast ('2023-07-03 22:22:22' as timestamp) -->48

extract(field FROM x)	bigint	Return the time extracted from x based on the field. The field can be year, quarter, month, week, day, dow, doy, yow, hour, minute, second, timezone_hour, timezone_minute, day_of_week, day_of_year, or day_of_month. For example: extract(year from current_date) 2023
day(x)	bigint	Return the day of the month for x.
day_of_month(x)	bigint	Return the day of the month for x.
day_of_week(x)	bigint	Return the day of the week for x, ranging from 1 to 7.
day_of_year(x)	bigint	Return the day of the year for x.
dow(x)	bigint	Return the day of the week for x. Same as day_of_week(x)
doy(x)	bigint	Return the day of the year for x. Same as day_of_year(x)
hour(x)	bigint	Return the hour of the day for x, ranging from 0 to 23.
millisecond(x)	bigint	Return the millisecond within the second for x.
minute(x)	bigint	Return the minute of the hour for x.
month(x)	bigint	Return the month of the year for x.
quarter(x)	bigint	Return the quarter of the year for x.
second(x)	bigint	Return the second of the minute for x.
week(x)	bigint	Return the week of the year for x.
week_of_year(x)	bigint	Return the week of the year for x.
year(x)	bigint	Return the year for x.

Array Functions

Function Name	Return Value	Function Feature Description
array_distinct(x)	array	Delete duplicate occurrences of the value x from the array.
array_intersect(x, y)	array	Return the distinct elements in the intersection of x and

		y.
array_union(x, y)	array	Return the distinct elements in the union of x and y.
array_except(x, y)	array	Return the distinct elements that are in x but not in y. (Difference)
array_join(x, delimiter, null_replacement)	varchar	Concatenate the elements of the array x using the delimiter, replacing null values with null_replacement. null_replacement is an optional character.
array_max(x)	x	Return the maximum value in the input array.
array_min(x)	x	Return the minimum value in the input array.
array_position(x, element)	bigint	Return the position (index) of the first occurrence of element in the array x (return 0 if not found).
array_remove(x, element)	array	Delete all elements in the array x that are equal to element.
array_sort(x)	array	Return the sorted array x. The elements of x must be sortable. Null elements will be placed at the end of the returned array.
array_sort(array(T), function(T, T, int))	array(T)	Return the array sorted according to the comparison function. For example: array_sort(ARRAY [3, 2, 5, 1, 2], (x, y) -> IF(x < y, 1, IF(x = y, 0, -1))) -->[5, 3, 2, 2, 1]
arrays_overlap(x, y)	Boolean value	Check if arrays x and y have any common non-null elements.
cardinality(x)	bigint	Return the cardinality (size) of the array.
concat(array1, array2, ..., arrayN)	array	Concatenate array1, array2, ... and arrayN.
contains(x, element)	Boolean value	Check if the array x contains the element
element_at(array(E), index)	E	Return the element at the index position in the array. If index > 0, counting starts from the left; if index < 0, counting starts from the right.
filter(array(T), funciton(T, boolean))	array(T)	Return a new array consisting of elements for which the function returns true. For example: filter(array[5, -6, NULL, 7], x -> x > 0); --> [5, 7]

repeat(element, count)	array	Count the number of times the element occurs.
reverse(x)	array	Return the array x with elements in reverse order.
sequence(start, stop, n)	array(bigint)	Generate a sequence of integers from start to stop with step size n, which is optional and defaults to 1. If start is less than or equal to stop, increment by 1; otherwise, increment by -1.
shuffle(x)	array	Shuffle the elements of the array.
slice(x, start, length)	array	Return a slice of the array x starting from start position with a length of length.
transform(array(T), function(T, U))	array(U)	Return a new array where each element T in the array is transformed to U by the function. For example: transform(array [5, 6], x -> x + 1); -- [6, 7]

JSON Functions

Function Name	Return Value	Function Feature Description
is_json_scalar(json)	boolean	Check if a JSON is a JSON number, JSON string, true, false, or null. For example: is_json_scalar('[1, 2, 3]') -->false
json_array_contains(json, value)	boolean	Check if a value exists in a JSON (a string containing a JSON array). For example: json_array_contains('[1, 2, 3]', 2) -->true
json_array_length(json)	bigint	Return the length of a JSON array (a string containing a JSON array). For example: json_array_length('[1, 2, 3]') --> 3
json_extract(json, json_path)	json	Evaluate a JSONPath-like expression json_path on a JSON (a string containing a JSON array) and return the result as JSON. For example: json_extract('{"log":{"file": {"path":"/etc/nginx/logs/access.log"},"offset":19991212}}', '$.log.file.path')--> "/etc/nginx/logs/access.log"
json_extract_scalar(json, json_path)	varchar	Similar to json_extract(), but the result is returned as a string instead of a JSON string. The value referenced by json_path must be a boolean, number, or string. For example: json_extract_scalar('{"log":{"file": {"path":"/etc/nginx/logs/access.log"},"offset":19991212}}', '$.log.file.path') -->/etc/nginx/logs/access.log

json_format(json)	varchar	Return JSON text serialized from the input JSON value. This is the inverse function of json_parse(). For example: json_format(JSON '[1, 2, 3]') -->[1,2,3]
json_parse(string)	json	Return a JSON value deserialized from the input JSON text. This is the inverse function of json_format(). For example: json_parse('[1, 2, 3]') -->[1,2,3]
json_size(json, json_path)	bigint	Similar to json_extract(), but the size of the value is returned. For objects or arrays, the size is the number of members. For scalar values, the size is zero. For example: json_size('{"x": [1, 2, 3]}', '$.x') -->3

Aggregation Functions

Function Name	Return Value	Function Feature Description
arbitrary(x)	Same as input	Return any non-null value of x if it exists.
array_agg(x)	array[x]	

Comparison of Hive Functions

Last updated：2024-08-07 17:34:50

DLC has comprehensive support for Hive functions, allowing you to easily upgrade from Hive to DLC and take advantage of more powerful data lake features.
There are some slight differences between DLC unified functions and Hive functions. The specific function comparison table is as follows:
For details on Hive functions, see the Apache Hive official website LanguageManual UDF.

Mathematical Functions

For Hive mathematical functions, see the Apache Hive official website MathematicalFunctions.
For DLC mathematical functions, see the Tencent Cloud official website documentation Mathematical Functions.

Hive 3.1 Function Name	Function Feature Description	DLC Function Name	Difference Description	Usage Reference
round	Rounding to the nearest integer	round	No differences	select round(1.23);
round	Rounding to a specified number of decimal places	round	No differences	select round(1.234,2);
bround	Rounding in HALF_EVEN mode	bround	In the DLC function, bround requires two parameters, with the second parameter specifying the number of decimal places to retain. Setting the second parameter to 0 has the same effect as	select bround(1.237,0);

			bround(double a) in Hive.	
bround	Rounding in HALF_EVEN mode, retaining the specified number of decimal places	bround	No differences	select bround(1.237,2);
floor	Rounding down	floor	No differences	select floor(1.23);
ceil, ceiling	Rounding up	ceil, ceiling	No differences	select ceil(1.93); select ceiling(1.13);
rand	Return a random number between 0 and 1. If a seed is specified, it returns a stable sequence of random numbers.	rand	No differences	select rand(); select rand(5);
exp	e raised to the power of a	exp	No differences	select exp(1); select exp(2.4);
ln	Natural logarithm function	ln	No differences	select ln(2.4);
log10	Base-10 logarithm function	log10	No differences	select log10(2.4);
log2	Base-2 logarithm function	log2	No differences	select log2(2.4);
log	Logarithm function	log	No differences	select log(2,4);
pow, power	Power function	pow, power	No differences	select pow(2,4);
sqrt	Square root function	sqrt	No differences	select sqrt(4);
bin	Binary function	bin	No differences	select bin(4);
hex	Hexadecimal function	hex	No differences	select hex(10);
unhex	Hexadecimal to string conversion	unhex	The result needs to be decoded in DLC Spark functions.	select decode(unhex('3141'), 'UTF-8');

conv	Base conversion	conv	No differences	select conv(2,10,2);
abs	Absolute value	abs	No differences	select abs(-1);
pmod	Modulus	pmod	No differences	select pmod(5,3);
sin	Trigonometric function sin, with a in radians	sin	No differences	select sin(3.14);
asin	Trigonometric function arc sin, with a in radians	asin	No differences	select asin(0.5);
cos	Trigonometric function cos, with a in radians	cos	No differences	select cos(3.14);
acos	Trigonometric function arc cos, with a in radians	acos	No differences	select acos(1);
tan	Trigonometric function tan, with a in radians	tan	No differences	select tan(3.14);
atan	Trigonometric function arc tan, with a in radians	atan	No differences	select atan(1);
degrees	Radians to degrees	degrees	No differences	select degrees(3.14);
radians	Degrees to radians	radians	No differences	select radians(180);
positive	Return a.	positive	No differences	select positive(-1);
negative	Return -a.	negative	No differences	select negative(1);
sign	If a is positive, it returns 1.0; if a is negative, it returns -1.0. If a is 0, then it returns 0.0.	sign	No differences	select sign(1.12);
e	Return the natural constant e.	e	No differences	select e();
pi	Return the value of pi.	pi	No differences	select pi();
factorial	Factorial	factorial	No differences	select factorial(5);

cbrt	Cube root	cbrt	No differences	select cbrt(27);
shiftleft	Left shift	shiftleft	No differences	select shiftleft(3,1);
shiftright	Right shift	shiftright	No differences	select shiftright(3,1);
shiftrightunsigned	Unsigned right shift	shiftrightunsigned	No differences	select shiftrightunsigned(3,1);
greatest	Return the maximum value.	greatest	When there are null values among the parameters, the DLC function ignores null values and returns the maximum value excluding nulls, whereas the Hive function returns null.	select greatest(1,2,3.3);
least	Return the minimum value.	least	When there are null values among the parameters, the DLC function ignores null values and returns the minimum value excluding nulls, whereas the Hive function returns null.	select least(1,2,3.3);
width_bucket	Bucket value. Create num_buckets + 1 buckets of equal size	width_bucket	No differences	select width_bucket(10,1,20,2)

	based on min_value/max_value, and return the bucket number where the current value resides.			

Collection Functions

DLC collection functions are identical to Hive collection functions.

For Hive collection functions, see the Apache Hive official website CollectionFunctions.

For DLC aggregate functions, see the Tencent Cloud official documentation CollectionFunctions.

Hive 3.1 Function Name	Function Feature Description	DLC Function Name	Difference Description	Usage Reference
size	Return the size of a map/array.	size	No differences	select size(str_to_map('a:1,b:2'));
map_keys	Return the list of keys in a map.	map_keys	No differences	select map_keys(str_to_map('a:1,b:2'));
map_values	Return the list of values in a map.	map_values	No differences	select map_values(str_to_map('a:1,b:2'));
array_contains	Check if the array contains value.	array_contains	No differences	select array_contains(split('a,b',','),'a');
sort_array	Sort the elements in the array in ascending order.	sort_array	No differences	select sort_array(split('1,3,2',','));

Type Conversion Functions

Hive 3.1 Function	Function Description	DLC Function	Difference Description	Usage Reference

Name		Name		
binary	Convert the input parameter to a binary array.	binary	Hive 3.1 only supports input parameters of type string and binary, while DLC supports input parameters of type string, int, long, and binary. In Hive 3.2, the output parameter is converted to a string for display, whereas DLC's output parameter is a binary array and is not converted to a string.	select binary('testString') select binary(1) (not supported in HIVE) select binary(inputCol) from inputTable (not supported in HIVE)
cast	Force the input parameter to be converted to the specified type. An error is thrown if the given expression or value cannot be forcefully converted to the specified type. For example, a string-to-long conversion error: Cast function cannot convert value of type VARCHAR(65536) to type LONG.	cast	In Hive 3.1, failed force conversions return NULL, whereas in DLC, an error is reported.	select cast('10' as int)select cast(inputCol as int) from inputTable

Time Functions

Hive 3.1 Function Name	Function Feature Description	DLC Corresponding Function Name	Difference Description	Usage Referen
from_unixtime	Convert a numeric Unix time (representing the number of seconds since	from_unixtime	No differences	select from_unixtime(select from_uni 'yyyyMMdd HH

	1970-01-01 00:00:00 UTC) to a string in the specified format. The default output time format is yyyy-MM-dd HH:mm:ss, and the timezone is the system-defined timezone.			
unix_timestamp	Specify a time, or default to the current time, and return the number of seconds from 1970-01-01 00:00:00 UTC to the specified time.	unix_timestamp	No differences	select unix_tim select unix_timestamp 12 00:00:00') select unix_timestamp 12', 'yyyy-MM-d
to_date	Specify a time and return the date of that time. For example, the date for 2023-04-12 13:14:20 is 2023-04-12.	to_date	The to_date function in HIVE can only accept input in the specified format yyyy-MM-dd HH:mm:ss. For any other formats, NULL is returned. DLC can accept an additional parameter to specify the data format, making the input time format more flexible.	select to_date(' 19:41:23') select to_date(' 19:41', 'yyyy-M HH:mm') (not s HIVE) select to_date(' 19:41', 'yyyyMM HH:mm') (not s HIVE)
year	Specify a time and return the year of that time. For example, the year for 2023-04-12 13:14:20 is 2023.	year	HIVE does not support input in yyyy or yyyy-MM formats but supports other time or date formats. DLC supports input in yyyy or yyyy-MM formats.	select year('202 19:41:25') select year('202 supported in HI
quarter	Specify a time and return the quarter	quarter	HIVE does not support input in yyyy	select quarter(' 19:41')

	of that time. For example, the quarter for 2023-04-12 13:14:20 is 2.		or yyyy-MM formats but supports other time or date formats. DLC supports input in yyyy or yyyy-MM formats.	select quarter(' (not supported
month	Specify a time and return the month of that time. For example, the month for 2023-04-12 13:14:20 is 4.	month	HIVE does not support input in yyyy or yyyy-MM formats but supports other time or date formats. DLC supports input in yyyy or yyyy-MM formats. When the month exceeds 12 in the time format, HIVE calculates the difference with the last month and takes the remainder, whereas DLC returns NULL.	select month('2 19:41') select month('2 (not supported
day/dayofmonth	Specify a time and return the day of that time. For example, the day for 2023-04-12 13:14:20 is the 12th day of the month.	day/dayofmonth	HIVE does not support input in yyyy or yyyy-MM formats but supports other time or date formats. DLC supports input in yyyy or yyyy-MM formats. When the day exceeds the number of days in the month in the time format, HIVE takes the remainder of the number of days in the month, whereas DLC returns NULL.	select day('202 supported in HI select dayofmo 04-12 19:41') select day('202 19:41')
hour	Specify a time and return the hour of that time.	hour	HIVE does not support input in yyyy or yyyy-MM formats	select hour('20 19:41') select hour('20 supported in HI

	For example, the hour for 2023-04-12 13:14:20 is the 13th hour.		but supports other time or date formats. DLC supports input in yyyy or yyyy-MM formats. When the hour exceeds 23 in the time format, HIVE takes the remainder of 23, whereas DLC returns NULL.	
minute	Specify a time and return the minute of that time. For example, the minute for 2023-04-12 13:14:20 is the 14th minute.	minute	HIVE does not support input in yyyy or yyyy-MM formats but supports other time or date formats. DLC supports input in yyyy or yyyy-MM formats. When the minute exceeds 59 in the time format, HIVE takes the remainder of 60, whereas DLC returns NULL.	select minute('2 00:41') select minute('2 (not supported
second	Specify a time and return the second of that time. For example, the second for 2023-04-12 13:14:20 is the 20th second.	second	HIVE does not support input in yyyy or yyyy-MM formats but supports other time or date formats. DLC supports input in yyyy or yyyy-MM formats. When the second exceeds 59 in the time format, HIVE takes the remainder of 60, whereas DLC returns NULL.	select second(' 00:41:24') select second(' (not supported
weekofyear	Specify a time and return the week of the year for that time.	weekofyear	HIVE does not support input in yyyy or yyyy-MM formats	select weekofye 04-12 00:41:25 select weekofye 04') (not suppo

	For example, 2023-04-12 13:14:20 is the 15th week of 2023. Note that the first Monday of the year is considered the start of the first week. If the date is before the first Monday of the year, it is considered the 52nd week of the previous year. For example, if 2023-01-01 is a Sunday, weekofyear('2023-01-01') returns 52 (the 52nd week of the previous year), while weekofyear('2023-01-02') returns 1 (the first week of the current year).		but supports other time or date formats. DLC supports input in yyyy or yyyy-MM formats.	HIVE)
extract	extract(field from source) extracts the specified field from the input source time or interval.	extract	HIVE and DLC versions currently do not support adjusting precision. When the DATE or TIMESTAMP identifier is explicitly specified, if the date or time format does not match, DLC throws an error, whereas HIVE returns 0. When HIVE calculates INTERNAL intervals, if the field is not present in the	select extract(D DATE '2023-04 select extract(s TIMESTAMP '2 00:41:25') select extract(m interval '23-1' Y MONTH) select extract(m interval '23' MC select extract(y interval '23' MC supported in DL select extract(h interval '23 13:2 DAY TO SECC

			source, it automatically calculates the remainder, while DLC throws an error. For example, when the year field is extracted from an interval of 24 months, HIVE returns 2 years, while DLC throws an error stating the field does not exist.	
datediff	Calculate the number of days between startDate and endDate. If startDate is later than endDate, a negative number is returned. If any input is NULL, NULL is returned.	datediff	HIVE does not support input in yyyy or yyyy-MM formats but supports other time or date formats. DLC supports input in yyyy or yyyy-MM formats.	select datediff(' '2022-10') (not HIVE) select datediff(' 13', '2022-04-1
from_utc_timestamp	Given a timestamp in UTC, convert it to the time in the specified time zone timeZone. If any input is NULL, NULL is returned.	from_utc_timestamp	HIVE does not support input in yyyy or yyyy-MM formats but supports other time or date formats. DLC supports input in yyyy or yyyy-MM formats. HIVE returns the time as a string in the format of yyyy-MM-dd HH:mm:ss. DLC returns the UTC standard representation, for example, for the Shanghai time zone, it is yyyy-MM-ddTHH:mm:ss+08:00.	select from_utc_times 04', 'Asia/Seou select from_utc_times 04-12 15:00:00 'Asia/Shanghai

to_utc_timestamp	Given a timestamp in the time zone timeZone, convert it to the UTC standard time timestamp. If any input is NULL, NULL is returned.	to_utc_timestamp	HIVE does not support input in yyyy or yyyy-MM formats but supports other time or date formats. DLC supports input in yyyy or yyyy-MM formats. HIVE returns the time as a string in the format of yyyy-MM-dd HH:mm:ss. DLC returns the UTC standard representation, for example, for the Shanghai time zone, it is yyyy-MM-ddTHH:mm:ss+08:00.	select to_utc_timestar 04-12 15:00:00 'Asia/Shanghai select to_utc_timestar 04', 'Asia/Shan supported in HI
date_add	Date addition: Given a startDate date, return the date after adding numDays days. numDays can be negative. If any input is NULL, NULL is returned.	date_add	HIVE does not support input in yyyy or yyyy-MM formats but supports other time or date formats. DLC supports input in yyyy or yyyy-MM formats.	select date_add 12 15:00:00', 1 select date_add -1) (not support
date_sub	Date subtraction: Given a startDate date, return the date after subtracting numDays days. numDays can be negative. If any input is NULL, NULL is returned.	date_sub	HIVE does not support input in yyyy or yyyy-MM formats but supports other time or date formats. DLC supports input in yyyy or yyyy-MM formats.	select date_sub 12 15:00:00', 1 select date_sub -1) (not support
current_date	Return the current date.	current_date	No differences	select current_

current_timestamp	Return the current time.	current_timestamp	No differences	select current_t
add_months	Calculate the date after adding a specified number of months to a given date (or time). If the input is NULL, NULL is returned.	add_months	HIVE does not support input in yyyy or yyyy-MM formats but supports other time or date formats. DLC supports input in yyyy or yyyy-MM formats.	select add_mor 04-12 15:00:00 select add_mor 04', -1) (not sup HIVE)
last_day	Calculate the last day of the month for a given date. If the input is NULL, NULL is returned.	last_day	HIVE does not support input in yyyy or yyyy-MM formats but supports other time or date formats. DLC supports input in yyyy or yyyy-MM formats.	select last_day 12 15:00:00') select last_day (not supported
next_day	Calculate the date of the first day_of_week after the specified date, where day_of_week can be any day from Monday to Sunday. If any input is NULL, NULL is returned.	next_day	HIVE does not support input in yyyy or yyyy-MM formats but supports other time or date formats. DLC supports input in yyyy or yyyy-MM formats.	select next_day 12 15:00:00') select next_day (not supported
trunc	Calculate the first day of the specified date truncated to the given format (e.g., year, quarter, month, week, etc.), such as the first day of the quarter, month, or week.	trunc	HIVE does not support input in yyyy or yyyy-MM formats but supports other time or date formats. DLC supports input in yyyy or yyyy-MM formats. HIVE does not support truncation to WEEK.	select trunc('20 15:00:00', 'MOI select trunc('20 'YEAR') (not su HIVE)

-	Calculate the time of the first day of the specified date truncated to the given format (e.g., year, quarter, month, week, etc.), such as midnight on the first day of the quarter. Unlike the trunc method, date_trunc returns the time.	date_trunc	HIVE does not support this function, see the trunc function.	select date_tru '2023-04-12 15 select date_trunc('SE '2023-04-12 15
months_between	Calculate the number of months between date1 and date2. If date1 < date2, a negative number is returned. Note that for the calculation, each month is estimated as 31 days (i.e., the denominator is 31). The numerator is the actual time difference between the two dates, precise to the millisecond. The final result is a decimal. If any input is NULL, NULL is returned.	months_between	HIVE does not support input in yyyy or yyyy-MM formats but supports other time or date formats. DLC supports input in yyyy or yyyy-MM formats.	select months_betwee 12 15:00:00', '2 15:00:00') select months_betwee 04', '2023-04-1 (not supported
date_format	According to the specified formatting template fmt (see	date_format	When the date or time format is incorrect, or the fmt template format is incorrect,	select date_for 04-12 15:00:00 select date_for 04-12', 'yyyy-M

	Datetime Patterns for Formatting and Parsing, format the entered date. If any input is NULL, NULL is returned.		HIVE returns NULL, whereas DLC throws an error.	HH:mm')

Conditional Functions

Hive 3.1 Function Name	Function Feature Description	DLC Function Name	Difference Description	Usage Reference
if	Return valueTrue if the condition is true; otherwise, return valueFalseOrNull.	if	When the condition is NULL, HIVE treats the condition as false, whereas DLC throws an error.	select if(1<2, 12, 'false')
isnull	Return true if a is NULL; otherwise, return false.	isnull	No differences	select isnull('false')
isnotnull	Return true if a is not NULL; otherwise, return false.	isnotnull	No differences	select isnotnull('false')
nvl	If the value is NULL, return the second parameter.	nvl	No differences	select nvl(NULL, 'false') select nvl(1, 'false')
coalesce	Return the first non-NULL parameter.	coalesce	No differences	select coalesce(NULL, 'false')
case/when	Match evaluation	case/when	When the types of the match selection are different, HIVE only supports implicit conversion between integers and strings, whereas DLC also supports implicit	select CASE 'c' WHEN 'a' THEN 1 WHEN 'b' THEN 2 ELSE 0 END; select CASE WHEN 1 > 0 THEN 1 WHEN 2 > 0 THEN 2.0 ELSE 1.2 END;

			conversion for boolean types, etc.	
nullif	Return null if the parameters a = b; otherwise, return a.	nullif	HIVE does not support the first parameter being NULL, whereas DLC supports any parameter being NULL.	select nullif(1, 'test'); select nullif(NULL, 'test'); (not supported in HIVE)
assert_true	Throw an exception if the condition is not true; otherwise, return null.	assert_true	No differences	select assert_true(1>0)

String Functions

Hive 3.1 Function Name	Function Feature Description	DLC Corresponding Function Name	Difference Description	Usage Reference
ascii	Return the numerical value of the first string in str.	ascii	No differences	select ascii('222');
base64	Convert the parameter from binary to a base64 string.	base64	No differences	select base64('tencent');
character_length	Return the number of UTF-8 characters contained in str.	character_length	No differences	select char_length(binary('tencent'));
chr	Return a binary character equivalent to A.	chr	No differences	select chr(65);
concat	Return a string or byte sequence that concatenates	concat	No differences	select concat('Spark', 'SQL');

	the parameters passed as strings or bytes in order.			
context_ngrams	Given a contextual N-grams, return the top k contextual N-grams from a set of tokenized sentences.	context_ngrams	No differences	-
concat_ws	Return a string separated by sep.	concat_ws	No differences	select concat_ws(' ', 'tencent', 'dlc');
decode	Decode the first parameter into a string using the provided character sets (one of US-ASCII, ISO-8859-1, UTF-8, UTF-16BE, UTF-16LE, and UTF-16). If any parameter is NULL, the result is also NULL.	decode	No differences	select decode(encode('abc', 'utf-8'), 'utf-8');
elt				

Overview of Standard Spark Statement

Last updated：2024-08-07 17:35:07

The standard Spark engine kernel is self-developed based on Spark version 3.2 and is compatible with native Spark statement and behavior. For detailed statement, see the Spark statement documentation.

DDL Statement

Statement Related to Database

Purpose	Statement
Create a new database.	CREATE DATABASE
Show all databases defined in the metadata.	SHOW DATABASES
View database attributes.	DESCRIBE DATABASE
Change database attributes.	ALTER DATABASE SET DBPROPERTIES
Delete a database.	DROP DATABASE

Statement Related to Data Table

Purpose	Statement
Create a new data table.	CREATE TABLE
Query data table creation information.	SHOW CREATE TABLE
Query table attributes.	SHOW TBLPROPERTIES
Query all tables in the database.	SHOW TABLES
View column information and metadata information on data table.	DESCRIBE TABLE
Add columns to the data table.	ALTER TABLE ADD COLUMNS
Add new columns to the data table.	ALTER TABLE ADD COLUMN AFTER/FIRST
Chang field name.	ALTER TABLE ... RENAME COLUMN

Delete a specific column from the data table.	ALTER TABLE DROP COLUMN
Add partition information to the data table.	ALTER TABLE ADD PARTATION
List table partitions.	SHOW PARTITIONS
Delete partition information from the data table.	ALTER TABLE DROP PARTITION
Add partition fields to the Iceberg table.	ALTER TABLE ADD PARTITION FIELD
Delete partition fields from the Iceberg table.	ALTER TABLE DROP PARTITION FIELD
Change data table attributes.	ALTER TABLE SET TBLPROPERTIES
Change the storage location of the data table.	ALTER TABLE SET LOCATION
Modify the sorting method of the table data.	ALTER TABLE ... WRITE ORDERED BY
Modify the allocation policy of the partitioned table.	ALTER TABLE ... WRITE DISTRIBUTED BY PARTITION
Update partition information.	MSCK REPAIR TABLE
Delete the metadata table.	DROP TABLE
Show the logical or physical execution plan of SQL.	EXPLAIN
Call the table stored procedure.	CALL STATEMENT

View-Related Statement

Purpose	Statement
Create a view from the select results.	CREATE VIEW AS
Query views in the database.	SHOW VIEWS
View the column information of a view.	DESCRIBE VIEW
Show the view creation statement.	SHOW CREATE VIEW

View column information of the view.	SHOW COLUMNS IN VIEW
Rename the view.	ALTER VIEW RENAME TO
Modify view attributes.	ALTER VIEW SET TBLPROPERTIES
Delete the view.	DROP VIEW

Function-Related Statement

Purpose	
Create a function.	CREATE FUNCTION
View function creation statement.	SHOW FUNCTION
Delete function.	DROP FUNCTION

DML Statements

Purpose	
Insert a row of data.	INSERT STATEMENT
Replace a row of data.	INSERT OVERWRITE
Row-level data update operations, which can be used to replace INSERT OVERWRITE operations.	MERGE INTO
Query Iceberg table metadata.	TABLE METADATA
Insert query results into a data table.	INSERT INTO
Delete data from an Iceberg table.	DELETE STATEMENT
Update specific rows.	UPDATE

DQL Statement

Purpose	
Query data.	SELECT STATEMENT

Overview of Standard Presto Statement

Last updated：2024-08-07 17:35:31

The standard Presto engine kernel is based on Presto 0.242 and is self-developed. It is compatible with native Presto statement and behavior, suitable for interactive query analysis. For detailed statement, see the Presto statement documentation.

Purpose	Statement	Supported	Notes
Modify function definition.	ALTER FUNCTION	No	
Rename Schema.	ALTER SCHEMA	Yes	
Modify data table.	ALTER TABLE	Yes	Only supports Iceberg tables. When executing, you must specify the target table's Iceberg catalog using a three-part name, or execute USE iceberg.dbname first.
Collect statistics on data table.	ANALYZE	Yes	Only supports Hive tables.
Commit current transaction.	COMMIT	No	
Create function.	CREATE FUNCTION	No	
Create role.	CREATE ROLE	No	
Create SCHEMA.	CREATE SCHEMA	No	
Create data table.	CREATE TABLE	No	
Create table using SELECT.	CREATE TABLE AS	No	
Create view.	CREATE VIEW	No	
Delete PREPARE.	DEALLOCATE PREPARE	No	
Delete data.	DELETE	No	
Show column information.	DESCRIBE	Yes	

Show PREPARE input information.	DESCRIBE INPUT	No	
Show PREPARE output information.	DESCRIBE OUTPUT	No	
Delete function.	DROP FUNCTION	No	
Drop role.	DROP ROLE	No	
Delete SCHEMA.	DROP SCHEMA	No	
Delete data table.	DROP TABLE	No	
Delete view.	DROP VIEW	No	
Execute PREPARE.	EXECUTE	Yes	
Show logical or physical execution plan of SQL.	EXPLAIN	Yes	
Execute SQL and display execution plan.	EXPLAIN ANALYZE	Yes	
Grant privileges.	GRANT	No	
Grant role to specified object.	GRANT ROLE	No	
Insert data.	INSERT	Yes	If the table is an Iceberg table, you must specify the target table's Iceberg catalog using a three-part name, or execute USE iceberg.dbname first.
Create PREPARE.	PREPARE	Yes	
Reset SESSION to default value.	RESET SESSION	Yes	
Revoke authorization.	REVOKE	No	
Revoke granted role.	REVOKE ROLES	No	
Rollback transaction.	ROLLBACK	No	
Query data.	SELECT	Yes	
Set role.	SET ROLE	No	

Set value for specified SESSION.	SET SESSION	Yes	
Show CATALOG list.	SHOW CATALOGS	Yes	
Show column information of a table.	SHOW COLUMNS	Yes	
Show function information.	SHOW CREATE FUNCTION	No	
Show table creation information.	SHOW CREATE TABLE	Yes	
Show view creation information.	SHOW CREATE VIEW	No	
Show function list.	SHOW FUNCTIONS	Yes	
Show permissions of specified user.	SHOW GRANTS	No	
Show the list of granted roles.	SHOW ROLE GRANTS	No	
Show role list.	SHOW ROLES	No	
Show SCHEMA list.	SHOW SCHEMAS	Yes	
Show SESSION list.	SHOW SESSION	Yes	
Show table statistics.	SHOW STATS	Yes	
Show table list.	SHOW TABLES	Yes	
Start transaction.	START TRANSACTION	No	
Delete all contents of a table.	TRUNCATE	No	
Update table contents.	UPDATE	No	
Set default SCHEMA or database.	USE	Yes	

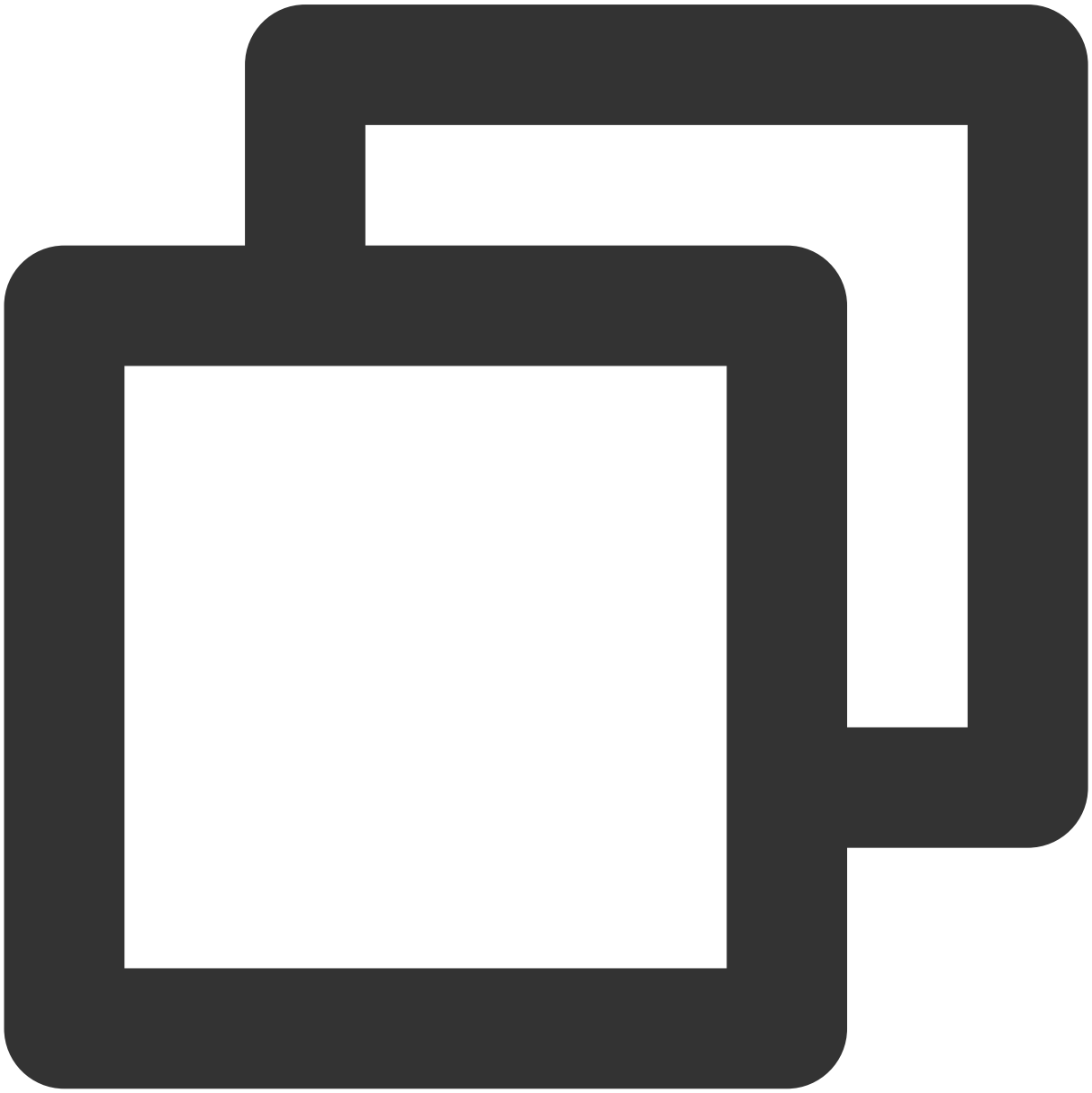
Define inline table.	VALUES	Yes	

Reserved Words

Last updated：2024-08-07 17:35:44

Must-Knows

If any of these identifiers are used as reserved words for database names, table names, column names, function names, or view names, they need to be enclosed with default escape characters:



```
Identifier: ` backticks

For example: hour -> `hour`

create table `hour` (
id string,
`asc` int
)

hour and asc are keywords. If you need to create columns with these names, you must
```

Key Reserved Words

A

ALL
ALTER
AND
ANY
AS
AUTHORIZATION

B

BETWEEN
BOTH
BY

C

CALL
CASE
CAST
CHECK
CLUSTER
COLLATE
COLUMN
CONSTRAINT
CREATE
CROSS

CUBE
CURRENT_DATE
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_USER
CURSOR

D

DEALLOCATE
DEFAULT
DELETE
DESCRIBE
DISTINCT
DISTRIBUTE
DROP

E

ELSE
END
ESCAPE
EXCEPT
EXECUTE
EXISTS
EXPLAIN
EXTRACT

F

FETCH
FILTER
FOR
FOREIGN
FROM
FULL
FALSE

G

GRANT
GROUP

GROUPING

H

HAVING

I

IN
INNER
INSERT
INTERSECT
INTERVAL
INTO
IS

J

JOIN

L

LATERAL
LEADING
LEFT
LIKE
LIMIT
LOCALTIME
LOCALTIMESTAMP

M

MERGE
MINUS

N

NATURAL
NEW
NEXT
NORMALIZE
NOT
NULL

O

OFFSET
ON
ONLY
OR
ORDER
OUTER
OVER
OVERLAPS

P

PARTITION
PATTERN
PERCENTILE_CONT
PERCENTILE_DISC
PERMUTE
PREPARE
PRIMARY

R

RANGE
RECURSIVE
REFERENCES
RIGHT
ROLLUP
ROW
ROWS

S

SELECT
SEMI
SESSION_USER
SET
SOME

T

TABLE
THEN

TIME
TO
TRAILING
TRUE

U

UESCAPE
UNION
UNIQUE
UNKNOWN
UNNEST
UPDATE
USER
USING

V

VALUES

W

WHEN
WHERE
WINDOW
WITH
WITHIN