itachi

Release 0.3.0

Kent Yao

Jul 14, 2022

FUNCTIONS

CHAPTER

ONE

ITACHI

itachi brings useful functions from modern database management systems to Apache Spark :)

For example, you can import the Postgres extensions and write Spark code that looks just like Postgres.

The functions are implemented as native Spark functions, so they're performant.

In general, only those functions that difficult for the Apache Spark Community to maintain in the master branch will be added to this library.

1.1 Installation

Fetch the JAR file from Maven.

> libraryDependencies += "com.github.yaooqinn" %% "itachi" % "0.1.0"

Here's the Maven link where the JAR files are stored.

itachi requires Spark 3+.

1.2 Simple function registration

Access the Postgres / Teradata functions with these commands::

```
org.apache.itachi.registerPostgresFunctions
org.apache.itachi.registerTeradataFunctions
```

1.3 Simple example

Suppose you have the following data table and would like to join the two arrays, with the familiar array_cat function from Postgres.:

```
+------+------+
|  arr1|  arr2|
+------+------+
|[1, 2]|    []|
|[1, 2]|[1, 3]|
+------+------+
```

Concatenate the two arrays::

```
spark
  .sql("select array_cat(arr1, arr2) as both_arrays from some_data")
  .show()

+------------+
| both_arrays|
+------------+
|      [1, 2]|
|[1, 2, 1, 3]|
+------------+
```

itachi lets you write Spark SQL code that looks just like Postgres SQL!

1.4 Spark SQL extensions installation

Config your spark applications with *spark.sql.extensions*, e.g. *spark.sql.extensions=org.apache.spark.sql.extra.PostgreSQLExtensions*

- org.apache.spark.sql.extra.PostgreSQLExtensions
- org.apache.spark.sql.extra.TeradataExtensions

1.5 Databricks Installation

Create an init script in DBFS::

```
dbutils.fs.mkdirs("dbfs:/databricks/scripts/")

dbutils.fs.put("/databricks/scripts/itachi-install.sh","""
#!/bin/bash
wget --quiet -O /mnt/driver-daemon/jars/itachi_2.12-0.1.0.jar https://repo1.maven.org/
↪maven2/com/github/yaooqinn/itachi_2.12/0.1.0/itachi_2.12-0.1.0.jar""", true)
```

Before starting the cluster, set the Spark Config::

```
spark.sql.extensions org.apache.spark.sql.extra.PostgreSQLExtensions
```

Also set the DBFS file path before starting the cluster::

```
dbfs:/databricks/scripts/itachi-install.sh
```

You can now attach a notebook to the cluster using Postgres SQL syntax.

1.6 Spark SQL Compliance

This is a Spark SQL extension supplying add-on or aliased functions to the Apache Spark SQL builtin standard functions.

The functions in this library take precedence over the native Spark functions in the even of a name conflict.

1.7 Contributing

More popular modern dbms system function can be added with your help

1.7.1 Itachi Function List

postgres

age

- **Usage**

```
   age(expr1, expr2) - Subtract arguments, producing a "symbolic" result that uses␣
↪years and months
   age(expr) - Subtract from current_date (at midnight)
```

- **Arguments**

```

```

- **Examples**

```
   > SELECT age(timestamp '1957-06-13');
    43 years 9 months 27 days
   > SELECT age(timestamp '2001-04-10', timestamp '1957-06-13');
    43 years 9 months 27 days
```

- **Class**

```
org.apache.spark.sql.catalyst.expressions.postgresql.Age
```

- **Note**
- **Since** 0.1.0

array_append

- **Usage**

```
 array_append(array, element) - Returns an array of appending an element to the end of␣
↪an array
```

- **Arguments**

```

```

- **Examples**

```
   Examples:
     > SELECT array_append(array(1, 2, 3), 3);
      [1,2,3,3]
     > SELECT array_append(array(1, 2, 3), null);
      [1,2,3,null]
     > SELECT array_append(a, e) FROM VALUES (array(1,2), 3), (array(3, 4), null),␣
↪(null, 5) tbl(a, e);
      [1,2,3]
      [3,4,null]
      [5]
```

- **Class**

```
org.apache.spark.sql.catalyst.expressions.postgresql.ArrayAppend
```

- **Note**
- **Since** 0.1.0

array_cat

- **Usage**

```
array_cat(col1, col2, ..., colN) - Returns the concatenation of col1, col2, ..., colN.
```

- **Arguments**

```

```

- **Examples**

```
   Examples:
     > SELECT array_cat('Spark', 'SQL');
      SparkSQL
     > SELECT array_cat(array(1, 2, 3), array(4, 5), array(6));
      [1,2,3,4,5,6]
```

- **Class**

```
org.apache.spark.sql.catalyst.expressions.Concat
```

- **Note**

  Concat logic for arrays is available since 2.4.0.

- **Since** 1.5.0

array_length

- **Usage**

```
N/A.
```

- **Arguments**

```

```

- **Examples**

```

```

- **Class**

```
org.apache.spark.sql.catalyst.expressions.postgresql.ArrayLength
```

- **Note**
- **Since**

justifyDays

- **Usage**

```
justifyDays(expr) - Adjust interval so 30-day time periods are represented as months
```

- **Arguments**

```

```

- **Examples**

```
  Examples:
    > SELECT justifyDays(interval '1 month -59 day 25 hour');
     -29 days 25 hours
```

- **Class**

```
org.apache.spark.sql.catalyst.expressions.postgresql.JustifyDays
```

- **Note**
- **Since** 0.1.0

justifyHours

- **Usage**

```
justifyHours(expr) - Adjust interval so 30-day time periods are represented as months
```

- **Arguments**

```

```

- **Examples**

```
    Examples:
      > SELECT justifyHours(interval '1 month -59 day 25 hour');
       -29 days 25 hours
```

- **Class**

```
org.apache.spark.sql.catalyst.expressions.postgresql.JustifyDays
```

- **Note**
- **Since** 0.1.0

justifyInterval

- **Usage**

```
justifyInterval(expr) - Adjust interval so 30-day time periods are represented as months
```

- **Arguments**

```

```

- **Examples**

```
    Examples:
      > SELECT justifyInterval(interval '1 month -59 day 25 hour');
       -29 days 25 hours
```

- **Class**

```
org.apache.spark.sql.catalyst.expressions.postgresql.JustifyDays
```

- **Note**
- **Since** 0.1.0

regr_count

- **Usage**

```
   regr_count(expr1, expr2) - Returns the count of all rows in an expression pair. The␣
↪function eliminates expression pairs where either expression in the pair is NULL. If␣
↪no rows remain, the function returns 0.
```

- **Arguments**

```
    expr1        The dependent DOUBLE PRECISION expression
    expr2        The independent DOUBLE PRECISION expression
```

- **Examples**

```
  > SELECT regr_count(1, 2);
   1
  > SELECT regr_count(1, null);
   0
```

- **Class**

```
org.apache.spark.sql.catalyst.expressions.ansi.RegrCount
```

- **Note**
- **Since** 0.2.0

scale

- **Usage**

```
N/A.
```

- **Arguments**

```

```

- **Examples**

```

```

- **Class**

```
Scale
```

- **Note**
- **Since**

split_part

- **Usage**

```
split_part(text, delimiter, field) - Split string on delimiter and return the given␣
↪field (counting from one).
```

- **Arguments**

```

```

- **Examples**

```
    Examples:
      > SELECT split_part('abc~@~def~@~ghi', '~@~', 2);
       def
```

- **Class**

```
org.apache.spark.sql.catalyst.expressions.postgresql.SplitPart
```

- **Note**
- **Since** 0.1.0

stage_attempt_num

- **Usage**

```
stage_attempt_num() - Get stage attemptNumber, How many times the stage that this task␣
↪belongs to has been attempted.
```

- **Arguments**

```

```

- **Examples**

```

```

- **Class**

```
org.apache.spark.sql.catalyst.expressions.debug.StageAttemptNumber
```

- **Note**
- **Since** 0.3.0

stage_id

- **Usage**

```
stage_id() - Get the stage id which the current task belong to
```

- **Arguments**

```

```

- **Examples**

```

```

- **Class**

```
org.apache.spark.sql.catalyst.expressions.debug.StageId
```

- **Note**
- **Since** 0.3.0

stage_id_with_retry

- **Usage**

```
stage_id_with_retry(stageId) - Get task attemptNumber, and will throw␣
↪FetchFailedException in the `stageId` Stage and make it retry.
```

- **Arguments**

```

```

- **Examples**

```

```

- **Class**

```
org.apache.spark.sql.catalyst.expressions.debug.StageIdWithRetry
```

- **Note**
- **Since** 3.3.0

string_to_array

- **Usage**

```
string_to_array(text, delimiter [, replaced]) - splits string into array elements using␣
↪supplied delimiter and optional null string
```

- **Arguments**

```

```

- **Examples**

```
Examples:
  > SELECT string_to_array('xx~^~yy~^~zz~^~', '~^~', 'yy');
   ["xx",null,"zz",""]
```

- **Class**

```
org.apache.spark.sql.catalyst.expressions.postgresql.StringToArray
```

- **Note**
- **Since** 0.1.0

task_attempt_id

- **Usage**

```
task_attempt_id() - Get an ID that is unique to this task attempt within SparkContext
```

- **Arguments**

```

```

- **Examples**

```

```

- **Class**

```
org.apache.spark.sql.catalyst.expressions.debug.TaskAttemptId
```

- **Note**
- **Since** 0.3.0

task_attempt_num

- **Usage**

```
task_attempt_num() - Get task attemptNumber, how many times this task has been attempted
```

- **Arguments**

```

```

- **Examples**

```

```

- **Class**

```
org.apache.spark.sql.catalyst.expressions.debug.TaskAttemptNumber
```

- **Note**
- **Since** 0.3.0

task_metrics_result_size

- **Usage**

```
task_metrics_result_size() - Meaningless
```

- **Arguments**

```

```

- **Examples**

```

```

- **Class**

```
org.apache.spark.sql.catalyst.expressions.debug.TaskMetricsResultSize
```

- **Note**
- **Since** 0.3.0

unnest

- **Usage**

```
unnest(expr) - Separates the elements of array `expr` into multiple rows recursively.
```

- **Arguments**

```

```

- **Examples**

```
 Examples:
   > SELECT unnest(array(10, 20));
    10
    20
   > SELECT unnest(a) FROM VALUES (array(1,2)), (array(3,4)) AS v1(a);
    1
    2
    3
    4
   > SELECT unnest(a) FROM VALUES (array(array(1,2), array(3,4))) AS v1(a);
    1
    2
    3
```

(continues on next page)

(continued from previous page)

```
      4
```

- **Class**

```
org.apache.spark.sql.catalyst.expressions.postgresql.UnNest
```

- **Note**
- **Since** 0.1.0

presto

char2hexint

- **Usage**

```
char2hexint(expr) - Returns the hexadecimal representation of the UTF-16BE encoding of␣
↪the string.
```

- **Arguments**

```

```

- **Examples**

```
  Examples:
    > SELECT char2hexint('Spark SQL');
      00530070006100720006B002000530051004C
```

- **Class**

```
org.apache.spark.sql.catalyst.expressions.teradata.Char2HexInt
```

- **Note**
- **Since** 0.1.0

cosine_similarity

- **Usage**

```
N/A.
```

- **Arguments**

```

```

- **Examples**

```

```

- **Class**

```
CosineSimilarity
```

- **Note**
- **Since**

EDITDISTANCE

- **Usage**

```
EDITDISTANCE(str1, str2) - Returns the Levenshtein distance between the two given␣
↪strings.
```

- **Arguments**

```

```

- **Examples**

```
  Examples:
    > SELECT EDITDISTANCE('kitten', 'sitting');
     3
```

- **Class**

```
org.apache.spark.sql.catalyst.expressions.Levenshtein
```

- **Note**
- **Since** 1.5.0

from_base

- **Usage**

```
from_base(num, from_base, to_base) - Convert `num` from `from_base` to `to_base`.
```

- **Arguments**

```

```

- **Examples**

```
  Examples:
    > SELECT from_base('100', 2, 10);
     4
    > SELECT from_base(-10, 16, -10);
     -16
```

- **Class**

```
org.apache.spark.sql.catalyst.expressions.Conv
```

- **Note**
- **Since** 1.5.0

index

- **Usage**

```
    index(substr, str[, pos]) - Returns the position of the first occurrence of `substr`␣
↪in `str` after position `pos`.
      The given `pos` and return value are 1-based.
```

- **Arguments**

```

```

- **Examples**

```
    Examples:
      > SELECT index('bar', 'foobarbar');
       4
      > SELECT index('bar', 'foobarbar', 5);
       7
      > SELECT POSITION('bar' IN 'foobarbar');
       4
```

- **Class**

```
org.apache.spark.sql.catalyst.expressions.StringLocate
```

- **Note**
- **Since** 1.5.0

infinity

- **Usage**

```
N/A.
```

- **Arguments**

```

```

- **Examples**

```

```

- **Class**

```
Infinity
```

- Note
- Since

is_finite

- Usage

```
N/A.
```

- Arguments

```

```

- Examples

```

```

- Class

```
IsFinite
```

- Note
- Since

is_infinite

- Usage

```
N/A.
```

- Arguments

```

```

- Examples

```

```

- Class

```
IsInfinite
```

- Note
- Since

nan

- **Usage**

```
N/A.
```

- **Arguments**

```

```

- **Examples**

```

```

- **Class**

```
NaN
```

- **Note**
- **Since**

regr_count

- **Usage**

```
    regr_count(expr1, expr2) - Returns the count of all rows in an expression pair. The␣
↪function eliminates expression pairs where either expression in the pair is NULL. If␣
↪no rows remain, the function returns 0.
```

- **Arguments**

```
      expr1         The dependent DOUBLE PRECISION expression
      expr2         The independent DOUBLE PRECISION expression
```

- **Examples**

```
    > SELECT regr_count(1, 2);
     1
    > SELECT regr_count(1, null);
     0
```

- **Class**

```
org.apache.spark.sql.catalyst.expressions.ansi.RegrCount
```

- **Note**
- **Since** 0.2.0

stage_attempt_num

- **Usage**

```
stage_attempt_num() - Get stage attemptNumber, How many times the stage that this task␣
↪belongs to has been attempted.
```

- **Arguments**

```

```

- **Examples**

```

```

- **Class**

```
org.apache.spark.sql.catalyst.expressions.debug.StageAttemptNumber
```

- **Note**
- **Since** 0.3.0

stage_id

- **Usage**

```
stage_id() - Get the stage id which the current task belong to
```

- **Arguments**

```

```

- **Examples**

```

```

- **Class**

```
org.apache.spark.sql.catalyst.expressions.debug.StageId
```

- **Note**
- **Since** 0.3.0

stage_id_with_retry

- **Usage**

```
stage_id_with_retry(stageId) - Get task attemptNumber, and will throw␣
↪FetchFailedException in the `stageId` Stage and make it retry.
```

- **Arguments**

```
```

- **Examples**

```
```

- **Class**

```
org.apache.spark.sql.catalyst.expressions.debug.StageIdWithRetry
```

- **Note**
- **Since** 3.3.0

task_attempt_id

- **Usage**

```
task_attempt_id() - Get an ID that is unique to this task attempt within SparkContext
```

- **Arguments**

```
```

- **Examples**

```
```

- **Class**

```
org.apache.spark.sql.catalyst.expressions.debug.TaskAttemptId
```

- **Note**
- **Since** 0.3.0

task_attempt_num

- **Usage**

```
task_attempt_num() - Get task attemptNumber, how many times this task has been attempted
```

- **Arguments**

```
```

- **Examples**

```
```

- **Class**

```
org.apache.spark.sql.catalyst.expressions.debug.TaskAttemptNumber
```

- **Note**
- **Since** 0.3.0

task_metrics_result_size

- **Usage**

```
task_metrics_result_size() - Meaningless
```

- **Arguments**

```

```

- **Examples**

```

```

- **Class**

```
org.apache.spark.sql.catalyst.expressions.debug.TaskMetricsResultSize
```

- **Note**
- **Since** 0.3.0

to_base

- **Usage**

```
to_base(num, from_base, to_base) - Convert `num` from `from_base` to `to_base`.
```

- **Arguments**

```

```

- **Examples**

```
  Examples:
    > SELECT to_base('100', 2, 10);
     4
    > SELECT to_base(-10, 16, -10);
     -16
```

- **Class**

```
org.apache.spark.sql.catalyst.expressions.Conv
```

- **Note**
- **Since** 1.5.0

try

- **Usage**

```
    try(expr) - Evaluate an expression and handle certain types of runtime exceptions by␣
↪returning NULL.
    In cases where it is preferable that queries produce NULL instead of failing when␣
↪corrupt or invalid data is encountered, the TRY function may be useful, especially␣
↪when ANSI mode is on and the users need null-tolerant on certain columns or outputs.
    AnalysisExceptions will not be handled by this, typically runtime exceptions handled␣
↪by try function are:

      * ArightmeticException - e.g. division by zero, numeric value out of range,
      * NumberFormatException - e.g. invalid casting,
      * IllegalArgumentException - e.g. invalid datetime pattern, missing format␣
↪argument for string formatting,
      * DateTimeException - e.g. invalid datetime values
      * UnsupportedEncodingException - e.g. encode or decode string with invalid charset
```

- **Arguments**

```

```

- **Examples**

```
      Examples:
      > SELECT try(1 / 0);
       NULL
      > SELECT try(date_format(timestamp '2019-10-06', 'yyyy-MM-dd uucc'));
       NULL
      > SELECT try((5e36BD + 0.1) + 5e36BD);
       NULL
      > SELECT try(regexp_extract('1a 2b 14m', '\\d+', 1));
       NULL
      > SELECT try(encode('abc', 'utf-88'));
       NULL
```

- **Class**

```
org.apache.spark.sql.catalyst.expressions.teradata.TryExpression
```

- **Note**
- **Since** 0.1.0