

KARPOV.COURSES >>> КОНСПЕКТ



> Конспект > 5 урок > Применение R, Python, Geospatial в расчетах на GreenPlum

> Оглавление

- > Оглавление
- > Аналитическая экосистема GreenPlum
- > Аналитические функции
 - Агрегатные функции
 - Оконные функции
- > Процедурные языки
 - Пример на PL/Python:
- > Преобразование данных
- > Машинное обучение
 - Пример с построением линейной регрессии
- > Работа с текстом
 - GreenPlum DataBase Text Search
 - GreenPlum Text
- > Работа с географией

> Аналитическая экосистема GreenPlum

Для расширения возможностей GreenPlum, к нему можно подключить различные модули. Аналитическая экосистема GreenPlum состоит из:

- **Analytical Functions** - аналитические функции
- **Procedural Languages** - процедурные языки
- **Data Transformation** - преобразования данных
- **Machine & Deep Learning** - машинное обучение
- **Graph** - работа с графами
- **Time Series** - работа с временными сериями
- **Geospatial** - работа с географией
- **Text processing** - обработка текстов

Далее рассмотрим каждый из этих пунктов подробнее.

> Аналитические функции

Агрегатные функции

Позволяют за одно чтение данных с диска произвести всевозможные вычисления и выдать нужный результат, что сильно ускоряет время действия запроса.

- **COUNT** - подсчитывает количество входных строк
- **SUM** - сумма всех непустых входных значений
- **MIN/MAX** - минимум/максимум среди всех непустых входных значений
- **AVG** - арифметическое среднее всех непустых входных значений

Расширенный набор агрегатных функций:

- **MEDIAN** - вычисляет медиану среди всех непустых входных значений
- **PERCENTILE_COUNT/PERCENTILE_DISC** - вычисляет процентиль для непрерывного/дискретного распределения
- **SUM(array[])** - позволяет производить суммирование списков (суммирование ведется со соответствующим индексам)

```
WITH matrix AS (  
    SELECT ARRAY[1, 2], [3, 4] AS value  
    UNION ALL  
    SELECT ARRAY[0, 1], [1, 0] AS value  
)  
SELECT  
    SUM(value)  
FROM matrix  
;
```

```
-----+  
sum      |  
-----+  
{1,3},{4,4}|
```

- **UNNEST(array[])** - позволяет из одной строчки со списком создать несколько строк, в каждой из которой будет запись из переданного списка

Оконные функции

Позволяют выполнять вычисления над набором строк таблицы, который некоторым образом соотносится с текущей строкой.

Синтаксис определения рамки в упрощенном виде задается как:

OVER ([PARTITION BY ...], [ORDER BY ...]), где указание секции **PARTITION BY** группирует строки исходной таблицы в разделы, а порядок строк в разделе задается секцией **ORDER BY**.

- **FIRST_VALUE/LAST_VALUE** - позволяют в рамках окна найти соответственно первое и последнее значение
- **LAG/LEAD** - позволяют работать со строками, которые находятся на некотором расстоянии от текущей позиции в рассматриваемом окне

- **CUME_DIST** - позволяет создать функцию распределения

```
WITH table AS (
    SELECT 1 AS id, 10 AS val
    UNION ALL
    SELECT 2 AS id, 20 AS val
    UNION ALL
    SELECT 3 AS id, 30 AS val
    UNION ALL
    SELECT 4 AS id, 50 AS val
    UNION ALL
    SELECT 5 AS id, 70 AS val
)
SELECT
    id AS "ID",
    val AS "VAL",
    CUME_DIST() OVER (ORDER BY val) AS "Cume_Dist"
FROM table
ORDER BY val
;
```

```
-----+
ID|VAL|Cume_Dist|
---+---+-----+
1| 10|      0.2|
2| 20|      0.4|
3| 30|      0.6|
4| 50|      0.8|
5| 70|      1.0|
```

- **RANK/DENSE_RANK** - ранг текущей строки с пропусками / без пропусков

```
SELECT
    dep_id,
    last_name,
    salary,
    RANK() OVER (
        PARTITION BY dep_id
        ORDER BY salary
    ) AS "RANK",
    DENSE_RANK() OVER (
        PARTITION BY dep_id
        ORDER BY salary
    ) AS "DRANK"
FROM employees
WHERE dep_id = 60
ORDER BY "RANK", last_name
;
```

```

-----+-----+-----+-----+-----+
dep_id | last_name|salary|RANK|DRANK|
-----+-----+-----+-----+-----+
      60|Lorentz  | 4200| 1| 1|
      60|Austin   | 4800| 2| 2|
      60|Pataballa| 4800| 2| 2| -- RANK=DRANK из-за равенства зарплат
      60|Ernst    | 6000| 4| 3| -- RANK пропускает 3, а DENSE_RANK нет
      60|Hunold   | 9000| 5| 4| -- RANK учитывает общее кол-во элементов

```

- **NTILE** - разделяет упорядоченный набор данных на несколько групп (бакетов) и определяет, в какую группу попадет каждая строка

```

SELECT
    last_name,
    salary,
    NTILE(4) OVER (
        ORDER BY salary DESC
    ) AS quartile
FROM employees
WHERE department_id = 100
ORDER BY last_name, salary, quartile
;

```

```

-----+-----+-----+
last_name|salary|quartile|
-----+-----+-----+
Chen     | 8200|      2|
Faviet   | 9000|      1|
Greenberg| 12000|     1|
Popp     | 6900|      4|
Sciarra  | 7700|      3|
Urman    | 7800|      2|

```

- **ROW_NUMBER** - номер текущей строки в её разделе
- **PERCENT_RANK** - относительный ранг текущей строки

> Процедурные языки

К GreenPlum можно подключить различные процедурные языки. По умолчанию есть **PL/pgSQL**.

Дополнительно можно подключить:

- **PL/Python**

- PL/Perl

В проприетарных версиях можно подключить:

- PL/Java
- PL/R

Исполнение каждого из процедурных языков можно контейнеризировать (PL/Container).

С точки зрения скорости PL/pgSQL самый быстрый. Остальные процедурные языки нужно использовать аккуратно, так как это вносит дополнительные траты ресурсов машин на сегментах.

Пример на PL/Python:

Рассмотрим синтаксис. Нам требуется объявить функцию, которая возвращает указанный тип. Внутри мы пишем тело функции на python.

```
CREATE FUNCTION funcname (argument-list)
    RETURNS return-type
AS $$
    -- pl/python function-body
$$ LANGUAGE plpythonu;
```

Пример простой функции, которая из двух переданных чисел возвращает максимальное из них:

```
CREATE FUNCTION pymax (a integer, b integer)
    RETURNS integer
AS $$
    if a > b:
        return a
    return b
$$ LANGUAGE plpythonu;

SELECT pymax(1, 5)
```

Можно использовать как plpython 2, так и 3 (они не совместимы друг с другом). Важно помнить, что у python 2 поддержка уже закончилась, лучше использовать 3.

> Преобразование данных

В GreenPlum поддержан очень широкий набор различных функций для преобразования данных из одних форматов в другие, например из:






- JSON(b) <-> SQL array
- Xml <-> SQL array
- Python list <-> SQL array
- Создание json на лету

Поверх json (или бинарного json) можно создавать индексы, что заметно ускоряет работу с нестандартизированными документами.

В качестве примера работы с xml, можно выделить функции: `table_to_xml` / `schema_to_xml` / `database_to_xml`, которые позволяют перевести таблицу/схему/бд в xml формат.

Для работы с json можно выделить следующие операции:

Json operations

|  Operator |  Type |  Description |  Example |  Result |
|--|--|---|---|--|
| <code>=></code> | int | Get the JSON array element (indexed from zero). | <code>'[{"a":"foo"}, {"b":"bar"}, {"c":"baz"}]::json->2</code> | <code>{"c":"baz"}</code> |
| <code>=</code> | text | Get the JSON object field by key. | <code>'{"a": {"b":"foo"}}::json->a'</code> | <code>{"b":"foo"}</code> |
| <code>->></code> | int | Get the JSON array element as text. | <code>'[1,2,3]::json->>2</code> | <code>3</code> |
| <code>->></code> | text | Get the JSON object field as text. | <code>'{"a":1,"b":2}::json->>b'</code> | <code>2</code> |
| <code>#></code> | text[] | Get the JSON object at specified path. | <code>'{"a": {"b": {"c": "foo"}}}::json#>'a,b'</code> | <code>{"c": "foo"}</code> |
| <code>#>></code> | text[] | Get the JSON object at specified path as text. | <code>'{"a":[1,2,3],"b": [4,5,6]}::json#>>'a,2'</code> | <code>3</code> |

Работа с json поддерживается на уровне СУБД.

> Машинное обучение

В GreenPlum внутри sql запросов использовать методы машинного обучения. Для этого используется библиотека **Apache MADlib**, которая содержит:

- методы DataScience
- преобразование данных
- описательная и индуктивная статистика
- обучение "с учителем" или "на примерах"
- глубокое обучение с Keras & TensorFlow
- модули для работы с графами (необходим специальный формат таблиц, в которых мы храним вершины и ребра. Есть возможность поиска кратчайшего пути, поиска связных компонент и. т. п.)
- решение систем линейных уравнений
- функции обработки текстов
- анализ временных рядов ARIMA

Подробнее про работу с временными рядами:

Обработка и хранение данных временных рядов достаточно удобно, так как мы можем использовать:

- append-optimized таблицы
- колоночное хранение (будут считываться только нужные для анализа колонки)
- партиционирование (по дате и времени события)
- сжатие
- параллельная обработка на разных сегментах

Пример с построением линейной регрессии

Создадим таблицу, в которой есть идентификаторы, значения того, что нужно предсказать и x1, x2.

```
CREATE TABLE regr_example (  
  id int,  
  y int,  
  x1 int,
```



```

        x2 int
    );
    INSERT INTO regr_example VALUES
        (1, 5, 2, 3),
        (2, 10, 7, 2),
        (3, 6, 4, 1),
        (4, 8, 3, 4);

```

Потом применяем к ней функцию `linregr_train` из библиотеки MADlib:

```

SELECT madlib.linregr_train (
    'regr_example',          -- source table
    'regr_example_model',   -- output model table
    'y',                    -- dependent variable
    'ARRAY[1, x1, x2]'      -- independent variables
);

```

В результате выводится краткий отчет с основными коэффициентами регрессии, необходимыми для анализа полученной модели:

```

SELECT * FROM regr_example_model;

-[ RECORD 1 ]-----+-----+-----+-----+-----+-----+-----+-----+
coef                | {0.111111111111127,1.14814814814815,1.01851851851852} |
r2                  | 0.968612680477111 |
std_err             | {1.49587911309236,0.207043331249903,0.346449758034495} |
t_stats             | {0.0742781352708591,5.54544858420156,2.93987366103776} |
p_values            | {0.952799748147436,0.113579771006374,0.208730790695278} |
condition_no        | 22.650203241881 |
num_rows_processed  | 4 |
num_missing_rows_skipped | 0 |
variance_covariance | {{2.23765432098598,-0.257201646090342,-0.437242798353582}, |
                    | {-0.257201646090342,0.042866941015057,0.0342935528120456}, |
                    | {-0.437242798353582,0.0342935528120457,0.12002743484216}} |

```

С помощью полученной модели можно строить предсказания. Для этого нужно использовать функцию `linregr_predict`

```

SELECT
    regr_example.*,

```

```

madlib.linregr_predict ( ARRAY[1, x1, x2], m.coef ) AS predict,
y - madlib.linregr_predict ( ARRAY[1, x1, x2], m.coef ) AS residual
FROM regr_example, regr_example_model m;

```

| id | y | x1 | x2 | predict | residual |
|----|----|----|----|------------------|--------------------|
| 1 | 5 | 2 | 3 | 5.46296296296297 | -0.462962962962971 |
| 3 | 6 | 4 | 1 | 5.72222222222224 | 0.277777777777762 |
| 2 | 10 | 7 | 2 | 10.1851851851852 | -0.185185185185201 |
| 4 | 8 | 3 | 4 | 7.62962962962964 | 0.370370370370364 |

> Работа с текстом

В GreenPlum есть 2 варианта работы с текстом. Можно использовать [GreenPlum DataBase Text Search](#) или [GreenPlum Text](#).

GreenPlum DataBase Text Search

Он представляет из себя PostgreSQL Text Search , который адаптирован в MPP системе.

Функционал:

- парсинг и хранение документов
- GIST (Generalized Search Tree) и GIN (Generalized Inverted index) индексы по токенам
- работает внутри БД

Пример использования:

```

SELECT title
FROM pgweb
WHERE to_tsvector('english', body) @@ to_tsquery('english', 'friend');

```

Здесь `to_tsvector` токенизирует текст, а `to_tsquery` - ищем конкретный токен. Синтаксис, который нужно использовать в `to_tsquery` , можно изучить в [документации](#)

То, как происходит токенизация настраивается на уровне СУБД с помощью специальных параметров.

GreenPlum Text

Это проприетарный текстовый поиск на базе **Apache Solr**.

Кластер Apache Solr нужно поднимать самостоятельно. Он поднимается рядом с GreenPlum, что позволяет масштабировать его отдельно.

Функционал:

- Index & Search в Solr - индексы могут храниться как в GreenPlum, так и кластере Apache Solr
- данные как в GreenPlum так и вне
- faceting, NER (Named entity recognition) , POS (Part-of-speech)
- proprietary
- большое количество функции, их больше, чем в GreenPlum DataBase Text Search, но это проприетарно

> Работа с географией

Для работы с геоданными в GreenPlum удобно использовать библиотеку **PostGIS**.

Она включается в себя:

- геометрические типы
- вычислительные и аналитические функции
- GIST индексы
- инструменты импорта/экспорта
- GDAL - Geospatial Data Abstraction Library
- параллельную обработку

Примеры работы с PostGIS

Для проверки того, что одна гео-сущность входит в другую можно использовать функцию `ST_Contains`.

Создадим временную табличку и добавим у ней гео-колонку типа полигон: `CREATE temporary TABLE gtest (ID int4, NAME varchar(20));`
`SELECT AddGeometryColumn('', 'gtest', 'geom', -1, 'POLYGON', 2);`

```
CREATE temporary TABLE gtest ( ID int4, NAME varchar(20) );
SELECT AddGeometryColumn('', 'gtest', 'geom', -1, 'POLYGON', 2);
```

Заполним эту таблицу полигоном, который является квадратом со стороной равной 10:

```
INSERT INTO gtest (ID, NAME, GEOM)
VALUES (
    1,
    'First Geometry',
    ST_GeomFromText('POLYGON((0 0, 0 10, 10 10, 10 0, 0 0))', -1)
);
```

```
SELECT id, name, ST_AsText(geom) AS geom FROM gtest;
-----+-----+
id | name          | geom
-----+-----+
 1 | First Geometry | POLYGON ((0 0, 0 10, 10 10, 10 0, 0 0)) |
```

Попробуем найти в получившейся таблице такую геометрию, которая включала бы в себя нужную нам линию, получим наш квадрат:

```
SELECT id, geom
FROM gtest
WHERE ST_Contains(geom, 'LINESTRING(2 3,4 5,6 5,7 8)');
```

```
--+-----+
id|geom
--+-----+
1|POLYGON ((0 0, 0 10, 10 10, 10 0, 0 0))|
```

Geohash

Для удобной работы с координатами можно использовать **geohash**. Геохэши позволяют из любой точки создать набор символов, который с некоторой точностью задает нашу точку.

Их преимущество заключается в том, что:

- можно явно задавать точность
- каждый геохэш более высокого порядка входит в геохэш более низкого порядка, поэтому легко производить операции включения/исключения
- использование геохешей экономит место, так как их хранить проще, чем координаты

Например, зададим точку и посмотрим, какой геохэш будет ей соответствовать:

```
SELECT ST_GeoHash(ST_SetSRID(ST_MakePoint(-126,48),4326));

-----+
st_geohash      |
-----+
c0w3hf1s70w3hf1s70w3|
```

Здесь **ST_SetSRID** - устанавливает SRID для геометрии в определенное целочисленное значение. SRID (spatial referencing system identifier) - уникальный идентификатор, однозначно определяющий систему координат, например, используемый нами код 4326 соответствует географической системе координат WGS84.

Уменьшить длину геохеша, например, до 5. Видим, что полученное значение входит в полученный ранее более точный геохэш:

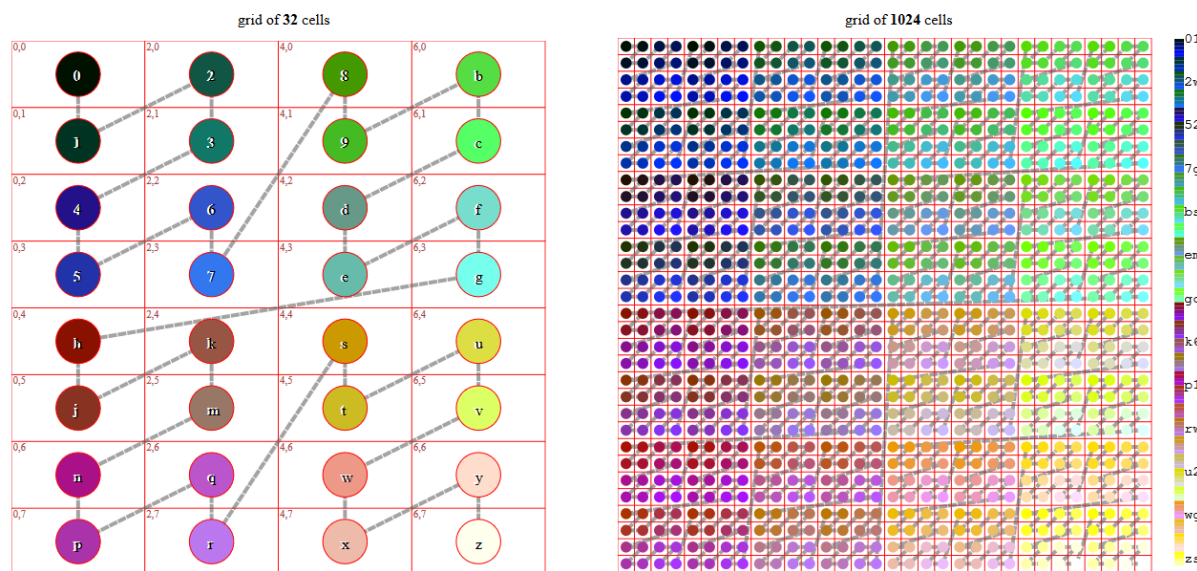
```
SELECT ST_GeoHash(ST_SetSRID(ST_MakePoint(-126,48),4326),5);

-----+
st_geohash|
-----+
c0w3h    |
```

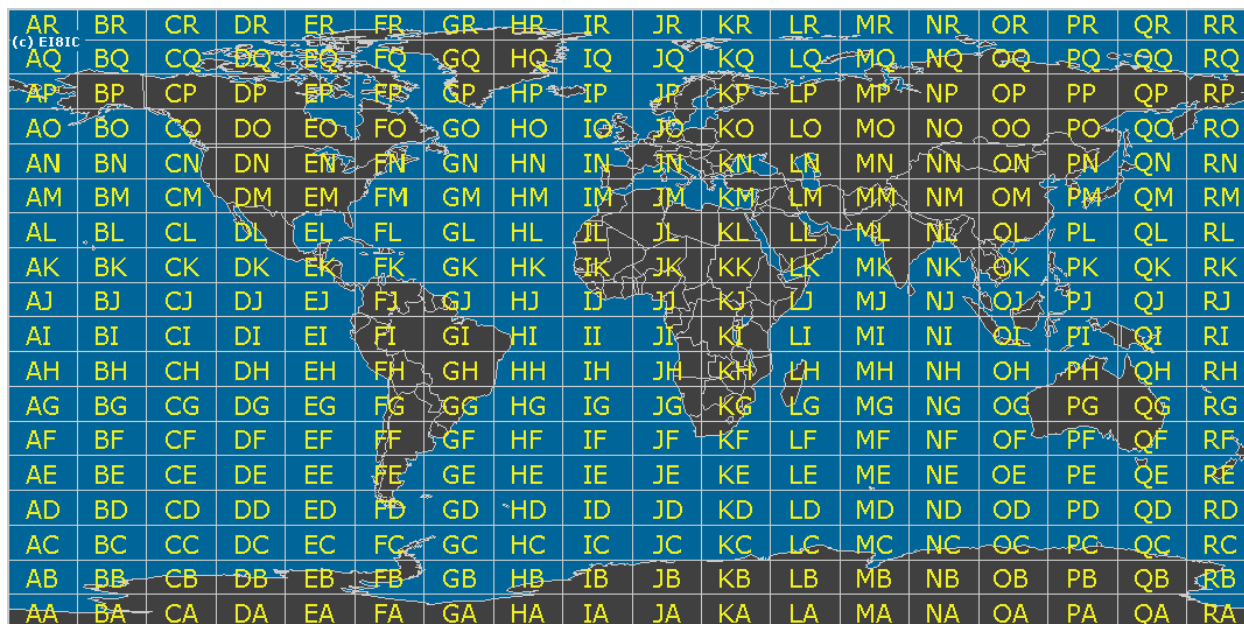
Подобным образом можно явно задавать точность геохеша. Например, геохэш в 6 символов дает погрешность \approx 600 м.

Чтобы лучше понять, как устроены геохэши разберем алгоритм их создания.

Разобьем карту на 32 квадрата, далее нумеруем эти квадраты от 1 до z. Каждый отдельный квадрат мы повторно разбиваем на 32 блока и снова нумеруем:



Таким образом, например, можно разбить карту мира на 64 блока, тогда каждый из этих блоков будет представлять из себя geohash уровня 2:



> **Дополнительные материалы**

Ссылки на документации:

- [аналитические функции](#)
- [про PL/Container](#)
- [преобразование json](#)
- [преобразование xml](#)
- [MADlib](#)
- [PostGIS](#)
- [сравнение GreenPlum DataBase Text Search и GreenPlum Text](#)