MapReduce simulation in SQL of the word_count problem

Dirk Van Gucht

1 The Word Count problem

- A document is modeled as a $(doc, words)$ pair
  1. doc the document identifier of a document
  2. $words$ is the set (bag) of words in that document
- The input to the word count is a set of documents
- The output is a set of pairs $(word, wordCount)$ where $wordCount$ is the number of occurrences of the word $word$ across all the documents
- We will present a SQL program in MapReduce-style for the word count problem.

- The relation documents is created as follows. Notice that we represent a bag of words with an array

```
CREATE TABLE documents(
   doc text,
   words text[]);
```

- Populate the documents relation as follows[1]

```
INSERT INTO documents VALUES('d1', ARRAY['A','B','C']);
INSERT INTO documents VALUES('d2', ARRAY['B','C','D']);
INSERT INTO documents VALUES('d3', ARRAY['A','E']);
INSERT INTO documents VALUES('d4', ARRAY['B','B','A','D']);
INSERT INTO documents VALUES('d5', ARRAY['E','F']);
```

We get

```
 doc     |   words
---------+-----------
 d1      | {A,B,C}
 d2      | {B,C,D}
 d3      | {A,E}
 d4      | {B,B,A,D}
 d5      | {E,F}
```

[1] Notice that a word may occur multiple times in a document.

- The expected output for the word count problem is the following:

```
 word | word_count
------+------------
 A    |          3
 B    |          4
 C    |          2
 D    |          2
 E    |          2
 F    |          1
```

2 The Word Count problem in SQL

- Before we show the MapReduce simulation, we begin by writing a SQL program for the word count problem
- This program will serve as a blueprint for the MapReduce simulation.

```
WITH
%map:
        doc_word AS (SELECT word, 1 AS one
                     FROM    documents d, UNNEST(d.words)),
%group:
        word_ones AS (SELECT word, array_agg(one) AS ones
                      FROM    doc_word
                      GROUP BY (word),
%reduce:
        occurrences AS (SELECT word, CARDINALITY(ones) AS wordCount
                        FROM word_ones p)

% output:
SELECT  * FROM occurences
```

3 MapReduce programs

- A basic MapReduce program is a pair of functions (`mapper, reducer`).
- The `mapper` function takes as input a $(key, value)$-pair and outputs a set (bag) of $(key, value)$-pairs.
- Note that the output key and $values$ need not be of of the same type as the input key and $value$.
- The `reducer` function takes as input a (key, bag_of_values)-pair and outputs a set (bag) of $(key, value)$-pairs.

4 Semantics of a MapReduce program

- The semantics of a basic MapReduce program consists of a map-, a group-, and a reduce-phase:
  - In the map-phase, the mapper is map-applied[2] to a set of $(key, value)$ pairs that come from the input and the outputs of all these calls are put together in a binary relation $A(key, value)$.
  - In the group-phase, the A relation is grouped[3]on its key-value column[4], and for each key-value, a pair (key, bag_of_values) is produced, where the bag_of_values is the bag of all values in A with that key value.
  - In the reduce-phase, the reducer is map-applied[5] to the (key, bag_of_values) pairs produced in the group-phase. For each such pair, the reducer produces a bag of $(key, value)$ pairs.
  - The output of the program is the bag of all these $(key, value)$ pairs.
- The semantics of a MapReduce program is the composition[6] of basic MapReduce programs.

[2]Typically in a parallelized manner.
[3]Sometimes also called *shuffled*
[4]Typically by hashing on this key-value.
[5]Typically in a parallelized manner.
[6]Typically in a pipe-lined manner.

5 Simulating a (basic) MapReduce program in SQL

- In Section 2, we expressed the `word count` problem in SQL. We deliberately wrote it in a fashion that resembles the map-, group-, and reduce- phases present in the semantics of a basic MapReduce program.
- We can give an even more faithful simulation if we write this SQL program using a mapper function and a reducer function. This can be done with SQL user-defined functions.

- We specify the `mapper` function as follows:

```
CREATE OR REPLACE FUNCTION mapper(doc text, words text[])
       RETURNS TABLE (word text, one integer) AS
$$
     SELECT w.wd, 1 FROM (SELECT UNNEST(words) AS wd) w;
$$ LANGUAGE SQL;
```

- The `mapper` function does the following when applied to a document:

```
SELECT p.word, p.one FROM mapper('d1',array['A','A','B']) p;
```

```
 word | one
------+-----
 A    |   1
 A    |   1
 B    |   1
```

- The `mapper` function when map-applied to the `documents` relation produces the relation `map_output(word, one)`.

```
WITH map_output AS
   (SELECT word, one
    FROM   documents d,  LATERAL(SELECT p.word, p.one
                                 FROM mapper(d.doc,d.words) p) q)

SELECT word, one
FROM map_output;
```

- Notice how we use `LATERAL` clause. This is necessary since in the `FROM` clause we need to apply the `mapper` function to each document `d` in the `documents` relation (i.e., the input pairs)

The map_output relation consist of the following key value pairs:

word	one
A	1
B	1
C	1
B	1
C	1
D	1
A	1
E	1
B	1
B	1
A	1
D	1
E	1
F	1

- Before we specify the `reducer` function, we show how the group-phase prepares the inputs to this function.
- This will be done by taking the `map_output` relation and grouping it on `word`
- This will associate with each `word` the bag of 1-values that it occurs with in this relation.
- These bags of 1-values will be formed using the `ARRAY` aggregation operator.
- We will put the output of the group-phase in the relation `group_output(word,ones)`.

```
WITH group_output AS
        (SELECT p.word, array_agg(p.one)
         FROM   map_output p
         GROUP BY (p.word))

SELECT word, ones FROM group_output;

 word |   ones
------+-----------
 F    | {1}
 A    | {1,1,1}
 E    | {1,1}
 C    | {1,1}
 B    | {1,1,1,1}
 D    | {1,1}
```

- We now specify the `reducer` function. In our case, this function takes as input a (`word`, bag of 1's `ones`) pair and outputs the desired (word, wordCount) pair
- For ease of programming, we let the reducer function return a relation with this `(word, wordCount)` pair.

```
CREATE OR REPLACE FUNCTION reducer(word TEXT, ones INTEGER[])
RETURNS TABLE(word TEXT, word_count INTEGER) AS
$$
    SELECT reducer.word, CARDINALITY(ones);
$$ LANGUAGE SQL;
```

- The reducer function does the following when applied to a `(word,ones)` pair.

```
SELECT word, word_count
FROM   reducer('A','{1,1,1,1}');

 word | word_count
------+------------
 A    |          4
```

- We can now map-apply the `reducer` function to the `(word,ones)` pairs `b` generated in the group-by phase and get the desired output.

```
SELECT t.word, t.word_count
FROM   group_output r, LATERAL(SELECT s.word, s.word_count
                                FROM   reducer(r.word, r.ones) s) t
```

```
 word | word_count
------+------------
 F    |          1
 A    |          3
 E    |          2
 C    |          2
 B    |          4
 D    |          2
```

6 MapReduce simulation in SQL

- Putting everything together we get the following SQL simulation of the `word_count` MapReduce program.

```
WITH
    %mapper phase
    map_output AS
    (SELECT q.word, q.one
     FROM   documents d,  LATERAL(SELECT p.word, p.one
                                  FROM   mapper(d.doc,d.words) p) q),

    %group phase
    group_output AS
    (SELECT p.word, array_agg(p.one)
     FROM   map_output p
     GROUP BY (p.word),

    %reducer phase
    reduce_output AS
    (SELECT t.word, t.word_count
     FROM   group_output r, LATERAL(SELECT s.word, s.word_count
                                    FROM   reducer(r.word, r.ones) s) t)


%output
SELECT word, word_count
FROM   reduce_output;
```

7 MapReduce in distributed setting

- In a distributed setting of compute nodes connected by a network, the `documents` relation is stored in chunks across the local file systems of these nodes.
- A mapper can process the chunk of documents at its compute node and then send its output to other compute nodes. This is typically done by applying a hash-function to a key-value. This hash-function will give the location of another compute node. The (key-value) pair is than sent to the compute node with the key's hash-function value.
- After all the appropriate values for a key have been sent to the appropriate compute nodes, the reducers can go to work locally (at the compute node) on the list of values associated with a key.
- The reducers can transmit their output, or they can keep it locally for further processing by other MapReduce programs.
- A big problem is skew in the data. It is possible that there is an uneven distribution of the values associated with keys. In that case, computation can slow considerably and the benefits of parallel (distributed) computing can be lost.