MapReduce simulation in SQL of the word_count problem

Dirk Van Gucht

1 The word_count problem

- A document is modeled as a $(doc\_id, words)$ pair, where doc_id is the document identifier of a document and $words$ is the set (bag) of words in that document.
- The input to the word_count problem is a set of documents and the output is a set of pairs $(word, word\_count)$ where $word\_count$ is the number of occurrences of the word $word$ across 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_id text,
   words text[]);
```

- Populate the `documents` relation as follows. Notice that a word may occur multiple times in a document.

```
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']);
```

- The contents of documents:

```
SELECT * FROM documents;

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

- The expected output is as follows:

```
 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 d.doc_id,
                                UNNEST(d.words) AS word
                                FROM documents d),
%group:
            word_ones AS (SELECT DISTINCT p.word,
                ARRAY(SELECT 1
                             FROM    doc_word p1
                             WHERE  p1.word=p.word) AS ones
                             FROM doc_word p),
%reduce:
            occurrences AS (SELECT p.word,
                              CARDINALITY(p.ones) AS word_count
                             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.
- Since in the semantics of MapReduce, the mapper and reducer functions are "map"-applied, basic MapReduce programs can be composed to form a MapReduce program.

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[1] to a set of $(key, value)$ pairs 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[2]on its key-value column[3], 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 S with that key value.
  - In the reduce-phase, the reducer is map-applied[4] 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[5] of basic MapReduce programs.

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

5 Simulating a (basic) MapReduce program in SQL

- In Section 2, we specified 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_id 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 q.word, q.one
    FROM   documents d,
                 LATERAL(SELECT p.word, p.one
                         FROM mapper(d.doc_id,d.words) p) q)
SELECT word, one FROM map_output;

 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
```

- Notice how we use `LATERAL` subqueries. This is necessary since in the `FROM` clause we need to apply the `mapper` function to each document `d` in the `documents` relation.

- 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 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)`.

  Notice that we need the `DISTINCT` clause.

```
WITH group_output AS
(SELECT DISTINCT p.word, (SELECT ARRAY(SELECT p1.one
                                        FROM   map_output p1
                                        WHERE  p1.word = p.word)) as ones
 FROM    map_output p)

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` and a bag of 1's `ones`.
- The cardinality of this bag corresponds to the number of occurrences, i.e., the word_count, of the word `word` in the `documents` relation. The `reducer` will output this `(word, word_count)` pair.
- For ease of programming, we let the reducer function return a relation with this `(word, word_count)` 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 to 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_id,d.words) p) q),
    %group phase
    group_output AS
    (SELECT DISTINCT q.word, (SELECT ARRAY(SELECT q1.one
                                           FROM   map_output  q1
                                           WHERE  q1.word = q.word)) as ones
     FROM map_output q),

     %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.