Key-value stores

The MapReduce Data and Programming Model (Simulated in Object-Relational SQL)

An Introduction to Distributed/Parallel Query Processing Based on Data Partitioning

Presented by Muazzam Siddiqui

Slides mainly adapted from Dirk Van Gucht

Key-values stores and queries

- A key-value store **S** is a binary relation with schema

  (key: K, value: V)

  where K and V are types with domains $dom(K)$ and $dom(V)$ of objects

- Note that the key attribute K is not necessarily a primary key of **S**. It is possible to have different key-value pairs in **S** with the same key value

- A key-value query $q : \mathbf{S}_1 \rightarrow \mathbf{S}_2$ is a mapping that sends a key-value store $\mathbf{S}_1(K_1, V_1)$ to a key-value store $\mathbf{S}_2(K_2, V_2)$

Key-value query languages and data-compute environments

- A key-value query *q* can be programmed in any number of (database) programming language such as (Object-Relational) SQL, PhP, MapReduce, Hive, PigLatin, Scala, Python, Java, Javascript, R, scripting languages, etc
- The database programming language is implemented in a data-compute environment (system) which can centralized or distributed and which has a certain model of data storage, data transfer, and communication: MySQL, PostgreSQL, Hadoop, Spark, MongoDb, Cloudera Impala, Neo4J, Amazon AWS, etc

Key-value query languages and data-compute environments

- Database applications are often implemented and run in different such systems
- Performance can be greatly affected by this choice
- The database-information ecosystem is bewildering complex and varied, and keeps changing

The MapReduce Data and Programming Model

- In this lecture, we will introduce the MapReduce programming language as an early example of a noSQL language to program key-value pair queries

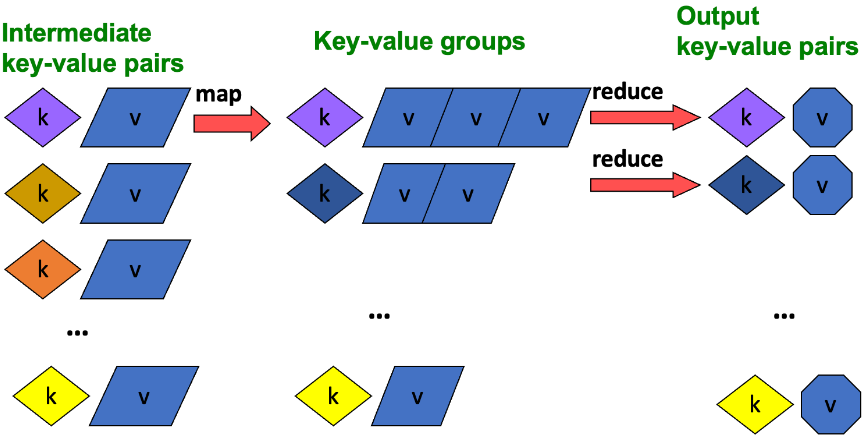
- MapReduce exhibits a variety of database modeling and query language concepts that have impacted newer database models and query languages

The MapReduce Data and Programming Model

- MapReduce as a programming environment is disappearing but is still very useful to discuss important concepts such as key-value stores, data partitioning, and distributed query processing

- We will cover the semantics of MapReduce and use the object-relational database model (such as implemented in PostgreSQL) to simulate it

The MapReduce Data and Programming Model



The Word Count query (Running Example)

- A key-value store Document(doc: text, words: text[]) stores a set of documents

- Each document is represented as a (doc, words) pair:
  1. doc is the document identifier of the document
  2. words is the set (bag) of words contained in the document

The Word Count query (Running Example)

- We wish to implement the word count query that maps the key-value store Document to the key-value store $\mathit{wordCount}(\mathit{word} : \mathit{text}, \mathit{wordcount} : \mathit{int})$
- The input to the word count query is the Document store
- The output of the word count query is the store of (word,wordCount) pairs where wordCount is the number of occurrences of the word word across all the documents in the Document store
- In summary, we want to implement

$$\mathit{WordCount} : \mathit{Document}(\mathit{text}, \mathit{words}) \rightarrow \mathit{wordCount}(\mathit{word}, \mathit{wordcount})$$

The Word Count query (Input)

- The store Document is created as follows:[1]

  CREATE TABLE Document(doc text, words text[]);

- Assume that we populate the Document store as follows:[2]

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

Document

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

[1] Notice that we represent a bag of words with an array.

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

The Word Count query (Output)

Document

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

word count query
→

wordCount

word	wordcount
A	3
B	4
C	2
D	2
E	2
F	1

The Word Count query in object-relational SQL

- First formulation:

```
SELECT      p.word, CARDINALITY(ARRAY_AGG(p.doc)) AS wordCount
FROM        (SELECT d.doc, UNNEST(d.words) AS word
             FROM    Document d) p
GROUP BY    (p.word)
```

- Second formulation: Notice that we don’t need the actual values of the document identifiers “doc". Thus, we can also formulate the word count query as follows:

```
SELECT      p.word, CARDINALITY(ARRAY_AGG(p.one)) AS wordCount
FROM        (SELECT UNNEST(d.words) AS word, 1 as one
             FROM    Document d) p
GROUP BY    (p.word)
```

The Word Count query in object-relational SQL (MapReduce style)

- Before we present the MapReduce simulation, we will decompose its object-relational SQL formulation into 3 phases that aligns with the MapReduce programming model
- This formulation will serve as a blueprint for the MapReduce simulation

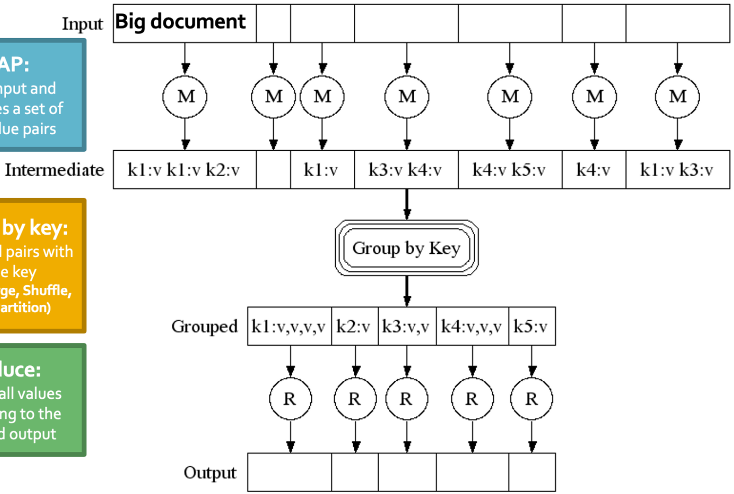
```
WITH
%mapper phase:
        doc_word AS (SELECT UNNEST(d.words) as word, 1 AS one
                     FROM Document d),
%group (shuffle) phase:
        word_ones AS (SELECT p.word, ARRAY_AGG(p.one) AS ones
                      FROM doc_word p
                      GROUP BY (p.word)),
%reducer phase:
        word_count AS (SELECT q.word, CARDINALITY(q.ones) AS wordCount
                       FROM word_ones q)

%output:
SELECT  word, wordCount FROM word_count
```

The MapReduce Data and Programming Model

MAP:
Read input and produces a set of key-value pairs

Group by key:
Collect all pairs with same key
(Hash merge, Shuffle, Sort, Partition)

Reduce:
Collect all values belonging to the key and output



The Word Count query (mapper phase)

%mapper phase:

doc_word AS (SELECT UNNEST(d.words) as word, 1 AS one
FROM Document d),

Document

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

mapper phase
→

doc_word

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

The Word Count query (group (shuffle) phase)

%group (shuffle) phase:

```
word_ones AS (SELECT p.word, ARRAY_AGG(p.one) AS ones
              FROM doc_word p
              GROUP BY (p.word)),
```

doc_word

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

group (shuffle) phase
→
group by word and aggregate the 1’s

word_ones

word	ones
A	{1,1,1}
B	{1,1,1,1}
C	{1,1}
D	{1,1}
E	{1,1}
F	{1}

The Word Count query (reducer phase)

```
%reducer phase:
    word_count AS (SELECT q.word, CARDINALITY(q.ones) AS wordCount
                   FROM word_ones q)
```

word_ones

word	ones
A	{1,1,1}
B	{1,1,1,1}
C	{1,1}
D	{1,1}
E	{1,1}
F	{1}

reducer phase
→
per word, count its document occurrences

word_count

word	wordCount
A	3
B	4
C	2
D	2
E	2
F	1

MapReduce queries (mapper and reducer functions)

- A basic MapReduce query is a pair of functions:

  (mapper, reducer)

- The mapper function takes as input a (*key, value*)-pair and outputs a relation (bag) of (*key, value*)-pairs:

  function mapper (key T_1, value T_2)
  returns table(key T_3, value T_4)

- The reducer function takes as input a (*key, bagOfValues*)-pair and outputs a relation (bag) of (*key, value*)-pairs:

  function reducer (key T_3, values T_4[])
  returns table(key T_5, value T_6)

- Notice how the output types T_3 and T_4 of the mapper function line up with the input types T_3 and T_4[] of the reducer function

MapReduce queries

- A MapReduce query is a sequence (composition) of basic MapReduce queries

$$(mapper_1, reducer_1); (mapper_2, reducer_2); \cdots ; (mapper_n, reducer_n)$$

- The output type of the key-value pairs produced by $reducer_i$ must be the same as the input type for $mapper_{i+1}$, for $i \in [1, n - 1]$

Semantics of a MapReduce query

- The semantics of a basic MapReduce query consists of a mapper-, a group (shuffle-), and a reducer-phase:
  - In the mapper phase, the mapper is map-applied[3] to an input relation of (key,value) pairs and the result is put into an intermediate relation *output_map* also consisting of (key,value) pairs
  - In the group (shuffle) phase, the *output _map* relation is grouped[4] on its key attribute[5] , and for each key k, a pair (k, *bagOfValues*(k)) is produced, where

    $$bagOfValues(k) = \{ v \mid output\_map(k, v) \},$$

    i.e, the bag of all values in *output_map* with key value k
  - In the reducer phase, the reducer is map-applied[6] to the (*key, bagOfValues*) pairs produced in the group (shuffle) phase. For each such pair, the reducer produces a relation of (*key, value*) pairs.
  - The output of the query is the union of all the relations produced in the reduce_phase
- The semantics of a MapReduce query is the composition of its basic MapReduce queries

[3]Typically in a parallelized and/or distributed manner.

[4]Sometimes also called *shuffled*

[5]Typically by hashing on this key-value.

[6]Typically in a parallelized and/or distributed manner.

Simulating a (basic) MapReduce query in SQL

- At the beginning of the lecture, we formulated the word count query in object-relational SQL.
- We deliberately wrote it in a fashion that resembles the mapper-, group (shuffle)-, and reducer phases present in the semantics of a basic MapReduce query
- We can give an even more faithful simulation if we write this SQL query using a mapper function and a reducer function
- These functions can be programmed as Object-Relational SQL user-defined functions

The mapper function

- We formulate the mapper function as follows:

```
CREATE FUNCTION mapper(doc text, words text[])
    RETURNS TABLE (word text, one int) AS
$$
    SELECT UNNEST(words) as word, 1 as one;
$$ LANGUAGE SQL;
```

- The mapper function does the following when applied to a document (doc,words) pair:

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

word	one
A	1
A	1
B	1

The mapper function (map-apply simulation)

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

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

- Notice how we use the LATERAL clause
- This is convenient since we need to map-apply the mapper function to each document d in the Document store

The mapper function (map-apply)

Document

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

map-apply the mapper function
→

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

Simulation of group (shuffle) phase

- Before we specify the reducer function, we show how the group-phase is simulated
- This will be done by taking the map_output relation and grouping it on the key attribute word

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

Simulation of the group (shuffle) phase

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

group (shuffle)
$\rightarrow$

group_output

word	ones
A	{1,1,1}
B	{1,1,1,1}
C	{1,1}
D	{1,1}
E	{1,1}
F	{1}

The reducer function

- We now formulate the reducer function.
- In our case, this function takes as input a

$$(\text{word}, \ \{1, \ldots, 1\})$$

pair and outputs the desired

$$(\text{word}, \text{COUNT}(\{1, \ldots, 1\}))$$

pair (in a relation)

```
CREATE FUNCTION reducer(word text, ones int[])
    RETURNS TABLE(word text, wordCount int) AS
$$
    SELECT word, CARDINALITY(ones) as wordCount;
$$ LANGUAGE SQL;
```

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

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

word	wordCount
A	4

The reducer function (map-apply simulation)

- We can now show the simulation of the reducer phase:
- I.e, map-apply the reducer function to the (word,ones) pairs generated in the group (shuffle) phase and produce the output of the word count MapReduce query

```
SELECT r.word, r.wordCount
FROM group_output q,
        LATERAL(SELECT p.word, p.wordCount
                FROM reducer(q.word, r.ones) p) r
```

- So we have

group_output

word	ones
A	{1,1,1}
B	{1,1,1,1}
C	{1,1}
D	{1,1}
E	{1,1}
F	{1}

map-apply the reducer function
→

word	wordCount
A	3
B	4
C	2
D	2
E	2
F	1

The word count MapReduce query in Object-Relational SQL

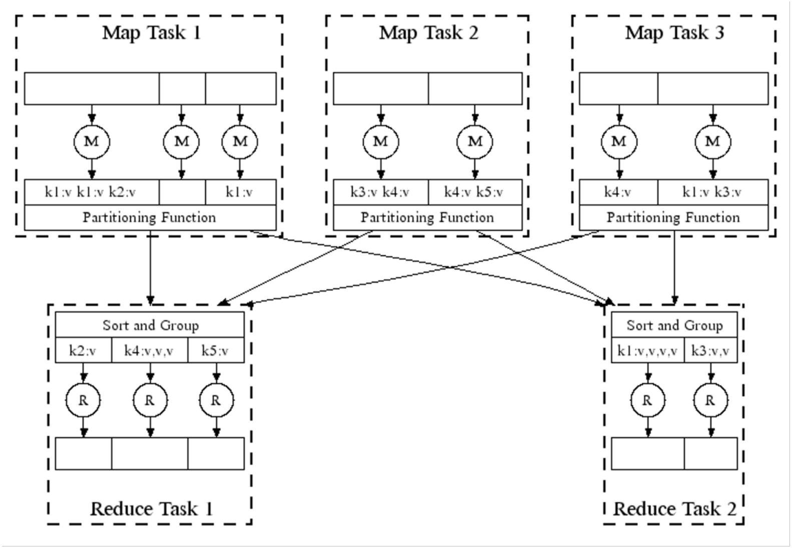
```
CREATE FUNCTION mapper(doc text, words text[])
    RETURNS TABLE (word text, one int) AS
$$
    SELECT UNNEST(words), 1
$$ LANGUAGE SQL;

CREATE FUNCTION reducer(word text, ones int[])
    RETURNS TABLE(word text, wordCount int) AS
$$
    SELECT word, CARDINALITY(ones);
$$ LANGUAGE SQL;
```

MapReduce simulation in Object-Relational SQL

- Putting everything together, we get the following simulation of the word count MapReduce query in Object-Relational SQL:

```
WITH
  %mapper phase
  map_output  AS
  (SELECT q.word, q.one
    FROM     Document 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) as ones
    FROM     map_output p
    GROUP BY (p.word)),

  %reducer phase
  reduce_output AS
   (SELECT r.word, r.wordCount
    FROM    group_output q,
                LATERAL(SELECT p.word, p.wordCount
                            FROM    reducer(q.word, q.ones) p) r)

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

MapReduce in distributed setting



MapReduce in distributed setting

- In a distributed setting of compute nodes connected by a network, the Document key-value store is stored or can be partitioned into chunks that reside at or can be distributed to the local file systems of the compute nodes
- There are multiple copies of the same mapper function at the compute nodes these can be evaluated (i.e., map-applied) independently and in parallel (there is no shared memory nor explicit communication)
- More specifically, an instance of the mapper function processes the chunk of Document at its compute node and provide its output to the group (shuffle) process
- The group (shuffle) phase is typically implemented by applying a hash-function to the keys of the (key,value) pairs emitted by the mappers. Applied to a key, this hash-function will give the location of another compute node
- A (key,value) pair is sent to the compute node with that key's hash-function value. Notice that, because of the properties of hashing functions, all (key,value) pairs with the same key will be sent to the same compute node

MapReduce in distributed setting

- There are multiple copies of the reducer function waiting at the compute nodes where the hashed (key,value) pairs arrive
- After all the appropriate values for a key have been sent and received by 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 queries.
- 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, the computation can be slowed at the reducer compute nodes and the benefits of parallel (distributed) computing can be lost.

The word count MapReduce query in Object-Relational SQL (Chunks/Partitioned Data)

- Assume that the document database Document is partitioned into a collection of pairwise disjoint document databases $\text{Document}_1, \ldots, \text{Document}_n$:

$$\text{Document} = \text{Document}_1 \cup \cdots \cup \text{Document}_n.$$

- We can use the same mapper function and apply it independently and in parallel to each of the Document_i
- The results of these n mapper computations can be accumulated and then grouped (shuffled) and sent to the reducer function.
- Observe that the mapper and reducer functions do not need to be changed.
- The simulation of this process is shown on the next slide

The word count MapReduce query in Object-Relational SQL (Alternative; Partitioned Data)

```
WITH
 %mapper phase– mapper function is evaluated on separate document chunks and accumulated with UNION
  map_output AS
   (SELECT q.word, q.one
    FROM     Document1 d1,
                   LATERAL(SELECT p.word, p.one
                                   FROM    mapper(d1.doc,d1.words) p)
    UNION
      . . .
    UNION
    SELECT q.word, q.one
    FROM     Documentn dn,
                   LATERAL(SELECT p.word, p.one
                                   FROM    mapper(dn.doc,dn.words) p) q)
  %group phase
  group_output AS
   (SELECT p.word, array_agg(p.one) as ones
    FROM     map_output p
    GROUP BY (p.word)),
  %reducer phase
  reduce_output AS
   (SELECT r.word, r.wordCount
    FROM    group_output q,
                 LATERAL(SELECT p.word, p.wordCount
                                FROM reducer(q.word, q.ones) p) r)
%output
SELECT word, wordCount
FROM reduce_output;
```

MapReduce Limitations

- Limitation 1:
  - Notice that a MapReduce query does not have conditional statements nor loop statements
  - This is a serious limitation for problems that require iteration to find solutions
  - Such is the case in many data science and machine learning problems (for example page-rank algorithm, k-means clustering, gradient-descent in deep learning etc.)
- Limitation 2: Performance can be greatly affected by skew in data: during the grouping, different keys may be associated with vastly different numbers of values

MapReduce Limitations

- Limitation 3:
  - Between successive basic MapReduce query, intermediate results are written to files
  - This limits in-memory processing
- Limitation 4:
  - The input to a MapReduce program is a single relation.
  - Notice that therefore to do a binary operation such a join, union, intersection, or set-difference between two relations *R* and *S*, it is necessary store/model the data in both *R* and *S* into a single relation of key-values pairs– awkward data modeling