Key-value stores: The Spark Data and Programming Model

(Explained relative to Relational Algebra and Object-Relational SQL)

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

Dirk Van Gucht[1]

Key-values stores and queries

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

  (key: K, value: V)

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

- A key-value pair (k, v) in $\mathbf{S}(K, V)$ is an element of $dom(K) \times dom(V)$

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

The Spark data and programming model (Data Model)

- Resilient Distributed Datasets (RDDs): collection of elements that can be operated on in parallel
- Elements can be any type. Typically, however, they are key-value pairs.
- There are two ways to further create RDDs
  1. parallelizing an existing (RDD) collection

```
val data = Array(1, 2, 3, 4, 5)
val distData = sc.parallelize(data)
```

  2. referencing a dataset in an external storage system (not further discussed)

Spark data and programming model (Data Model)

- Spark permits the definition of functions to create (key, value) pairs.
- In many ways, just as MapReduce, Spark processes key-value stores
- Below is an example of (key-value) creation in Spark:

Input RDD

String
hello
world
how
are
you

```
.map(word ⇒ Tuple2(word, 1))

.map(word ⇒ (word,1))
```

Output RDD

String	Int
hello	1
world	1
how	1
are	1
you	1

Spark programming model

- Spark supports two types of operations on RDDs:
  1. transformations, which create a new RDD dataset from an existing RDD
  2. actions, which return a value to the driver program after running a computation on the dataset

```
.map(word ⇒ (word,1)).reduceByKey(lambda a,b: a+b)
```

- Transformations and actions are written as functions that use algebraic operations most of which correspond directly to operations in Relational Algebra (join, selection, union, etc) and Object-Relational SQL (GROUP BY, aggregate functions, and UNNEST)

Spark programming model (related database concepts)

- All transformations in Spark are lazy: they do not compute their results immediately:
  database concept: views

- The transformations are only computed when an action requires a result:
  database concept:
  query evaluation on data represented by views

- The Spark programming model permits compilation and optimization:
  database concept:
  query translation and query optimization

Spark programming model (Persistent RDDs)

- By default, each transformed RDD is recomputed each time you run an action on it

  database concept: just like views are lazily evaluated

- However, you may also persist a (transformed) RDD in memory using the persist (or cache) method

- Spark will keep the elements around on the cluster for much faster access the next time you query it

  database concept: just like materialized views

Spark programming model (Transformations)

Spark	SQL/RA
R.map(func)	SELECT func(r) FROM R r
$(R_1, \cdots, R_n)$.mapPartitions(func)	SELECT func(r_1) FROM R_1 r_1 UNION · · · UNION SELECT func(r_n) FROM R_n r_n
R.filter(func)	SELECT r.* FROM R r WHERE func(r)
R.flatMap(func)	SELECT UNNEST(func(r)) FROM R r
R.union(S)	SELECT r FROM R r UNION SELECT s FROM S s
R.intersection(S)	SELECT r FROM R r INTERSECT SELECT sFROM S s

Spark programming model (Transformations)

Spark	SQL/RA
R.distinct()	SELECT DISTINCT r.* FROM R r
$R_{K,V}$.groupByKey()	SELECT K, array_agg(V) FROM $R_{K,V}$ GROUP BY(K)
$R_{K,V}$.reduceByKey(func)	SELECT K, func(array_agg(V)) FROM $R_{K,V}$ GROUP BY(K)
$R_{K,V}$.sortByKey()	SELECT r.* FROM $R_{K,V}$ r ORDER BY(K)
$R_{K,V}$.join($S_{K,W}$)	SELECT r.K, (r.V, s.W) FROM R r NATURAL JOIN S s
R.cartesian(S)	SELECT (r.*, s.*) FROM R r CROSS JOIN S s

Spark programming model (The `co-group` transformation)

$$R_{K,V}.\mathtt{cogroup}(S_{K,W})$$

R

K	V
a	1
a.	2
b.	1
c.	3

S

K	W
a	1
a	3
c	2
d	1
d	4

cogroup
→

K	(RV_values, SW_values)
a	({1,2}, {1,3})
b	({1} , {})
c	({3}, {2})
d	({}, {1,4})

```
WITH Kvalues AS (SELECT r.K FROM R r UNION SELECT s.K FROM S s),
        R_K AS (SELECT k.K,
                        ARRAY(SELECT r.V
                              FROM R r WHERE r.K = k.K) AS RV_values
                FROM Kvalues k),
        S_K AS (SELECT k.K,
                        ARRAY(SELECT s.W
                              FROM S s WHERE s.K = k.K) AS SW_values
                FROM Kvalues k)

SELECT K, (RV_values, SW_values)
FROM R_K NATURAL JOIN S_K
```

Observe that $R \bowtie S$ can be derived by applying flattening operations.

SQL to RA to Spark

- Translate SQL query to RA expression
- Optimize RA expression
- Translate RA expressions on the basis of the following correspondences:

σ_C	→	.filter(func$_C$)
π_C	→	.map(func$_L$)
$\times$	→	.cartesian
$\bowtie$	→	.join
$\cup$	→	.union
$\cap$	→	.intersection
$-$	→	.except or minus

Object-relational SQL queries with aggregate functions can be similarly translated using .groupByKey, reduceByKey(func)

Spark (Actions)

A Spark action triggers the evaluation of a program (including evaluation of the transformations)

`.reduce(func)`	Aggregate the elements of the dataset using a function func (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
`.collect()`	Return all the elements of the dataset as an array at the driver program.
`.count()`	Return the number of elements in the dataset.
`.first()`	Return the first element of the dataset (similar to take(1)).
`.take(n)`	Return an array with the first n elements of the dataset.
`.countByKey()`	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.

Complications during distributed computation due to partitioned data

RDDs R and S are stored as a partitions so that

$$R = R_1 \cup \cdots \cup R_m$$

The unary operations π and σ can be efficiently implemented in a parallel/distributed system

$$\pi_L(R) = \pi_L(R_1) \cup \cdots \cup \pi_L(R_m)$$

$$\sigma_C(R) = \sigma_C(R_1) \cup \cdots \cup \sigma_C(R_m)$$

Complications during distributed computation due to partitioned data

$$
\begin{aligned}
R &= R_1 \cup \cdots \cup R_m \\
S &= S_1 \cup \cdots \cup S_n
\end{aligned}
$$

The binary operations $\cup$, $\cap$, $-$, $\bowtie$ and $\times$ may require extensive data communication and transfer:

$R\,[\cup \mid \cap]\,S$	$=$	$_{i,j}\ R_i\,[\cup \mid \cap]\,S_j$	$R - S$	$=$	$_{i,j}\ R_i - S_j$
$R \bowtie S$	$=$	$_{i,j}\ \mathrm{R}_\mathrm{i} \bowtie \mathrm{S}_\mathrm{j}$	$R \times S$	$=$	$_{i,j}\ R_i \times S_j$

- Notice that we get a quadratic number $m \times n$ of operations to perform!
- Data needs to be shuffled which is expensive.
- These problems get only worse when there are many RDDs that are part of a query such as $(R \bowtie S) - T$.

Spark: Shared Variables

Spark does provide two limited types of shared variables:

1. broadcast variables: Broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks
2. accumulators: Accumulators are variables that are only “added” to through an associative and commutative operation and can therefore be efficiently supported in parallel.
   Compute nodes can add to the accumulator (but not see it). Only driver see accumulator.