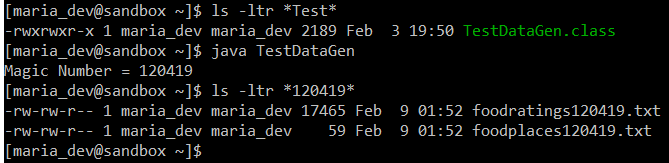
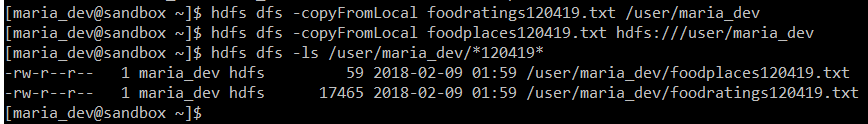
**CS595 - Assignment 5**

* Create new versions of the foodratings and foodplaces files by using TestDataGen (as described in assignment #4) and copy them to HDFS.



Magic Number = 120419



**Command Executed:**

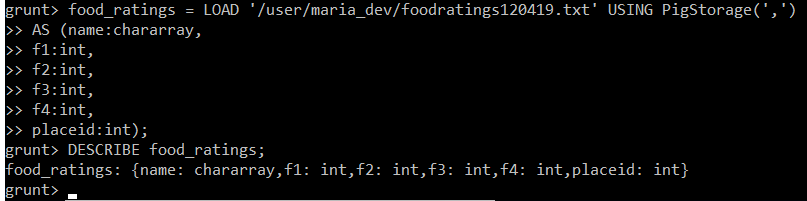
java TestDataGen

hdfs dfs -copyFromLocal foodratings120419.txt /user/maria\_dev

hdfs dfs -copyFromLocal foodplaces120419.txt hdfs:///user/maria\_dev;

1. Write and execute a sequence of pig latin statements that loads the foodratings file as a relation. Call the relation ‘food\_ratings’. The load command should associate a schema with this relation where the first attribute is referred to as ‘name’ and is of type chararray, the next attributes are referred to as ‘f1’ through ‘f4’ and are of type int, and the last field is referred to as ‘placeid’ and is also of type int.

Execute the describe command on this relation.



Provide the magic number, the load command you wrote and the output of the describe command as the result of this exercise.

**Command Executed:**

Magic Number = 120419

food\_ratings = LOAD '/user/maria\_dev/foodratings120419.txt' USING PigStorage(',')

AS (name:chararray,

f1:int,

f2:int,

f3:int,

f4:int,

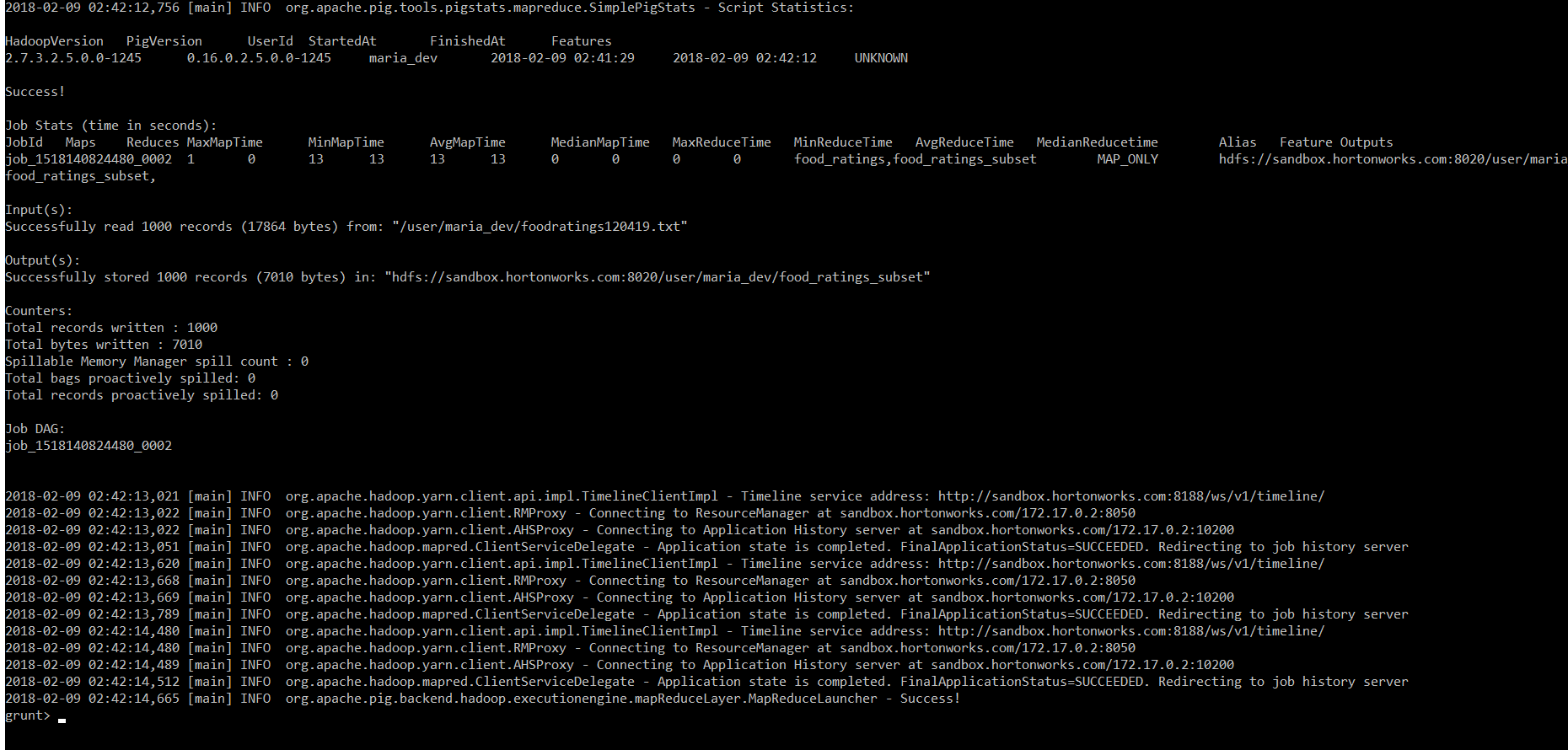
placeid:int);

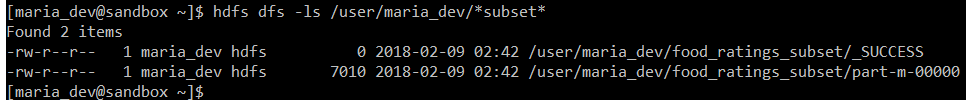
DESCRIBE food\_ratings;

1. Create another relation with two fields of the initial (food\_ratings) relation: ‘name’ and ‘f4’. Call this relation ‘food\_ratings\_subset’.



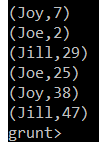
Store this last relation back to HDFS.





Also write 6 records of this relation out to the console.





Submit the pig latin statements you used and the six records printed out to the console as the result of this exercise.

**Command Executed:**

Magic Number = 120419

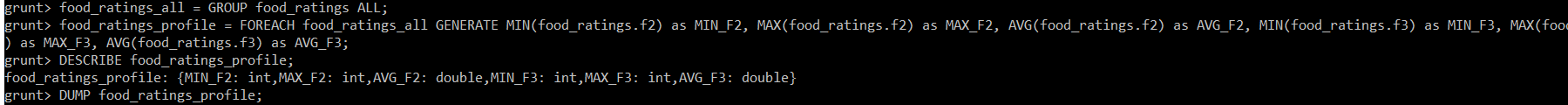
food\_ratings\_subset = FOREACH food\_ratings GENERATE name, f4;

STORE food\_ratings\_subset INTO 'food\_ratings\_subset' USING PigStorage ('|');

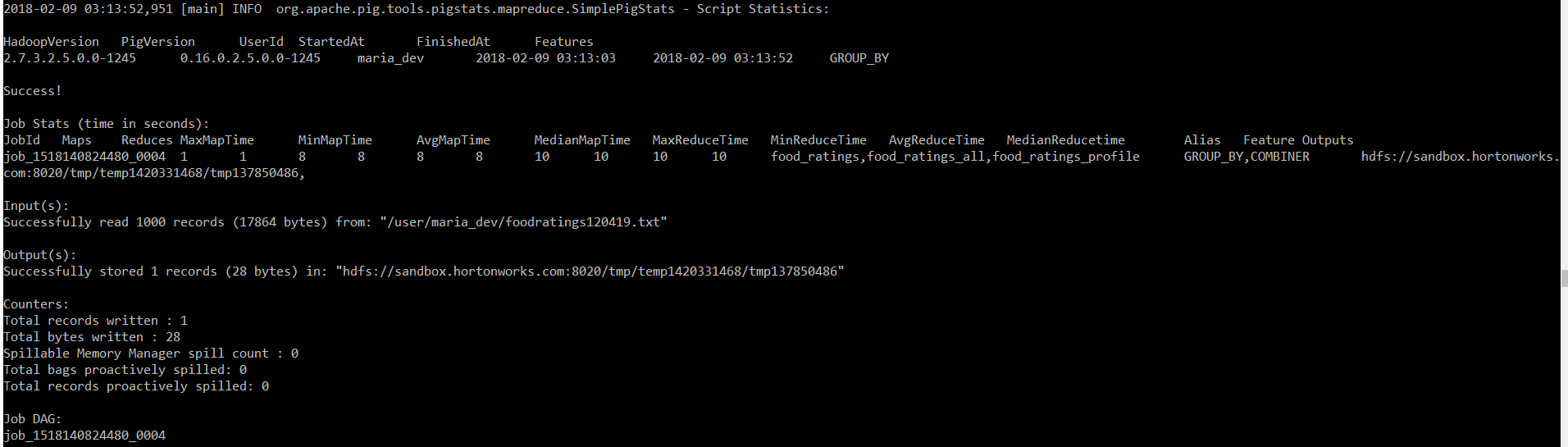
food\_ratings\_subset\_6 = LIMIT food\_ratings\_subset 6;

DUMP food\_ratings\_subset\_6;

1. Create another relation using the initial (food\_ratings) relation. Call this relation ‘food\_ratings\_profile’. The new relation should only have one record. This record should hold the minimum, maximum and average values for the attributes ‘f2’ and ‘f3’. (So this one record will have 6 fields).



Write the record of this relation out to the console.





Submit the pig latin statements you used and the record printed out to the console as the result of this exercise.

**Command Executed:**

Magic Number = 120419

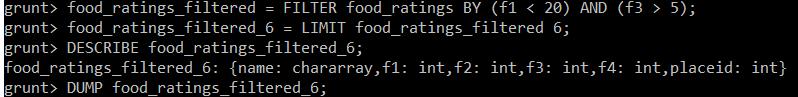
food\_ratings\_all = GROUP food\_ratings ALL;

food\_ratings\_profile = FOREACH food\_ratings\_all GENERATE MIN(food\_ratings.f2) as MIN\_F2, MAX(food\_ratings.f2) as MAX\_F2, AVG(food\_ratings.f2) as AVG\_F2, MIN(food\_ratings.f3) as MIN\_F3, MAX(food\_ratings.f3) as MAX\_F3, AVG(food\_ratings.f3) as AVG\_F3;

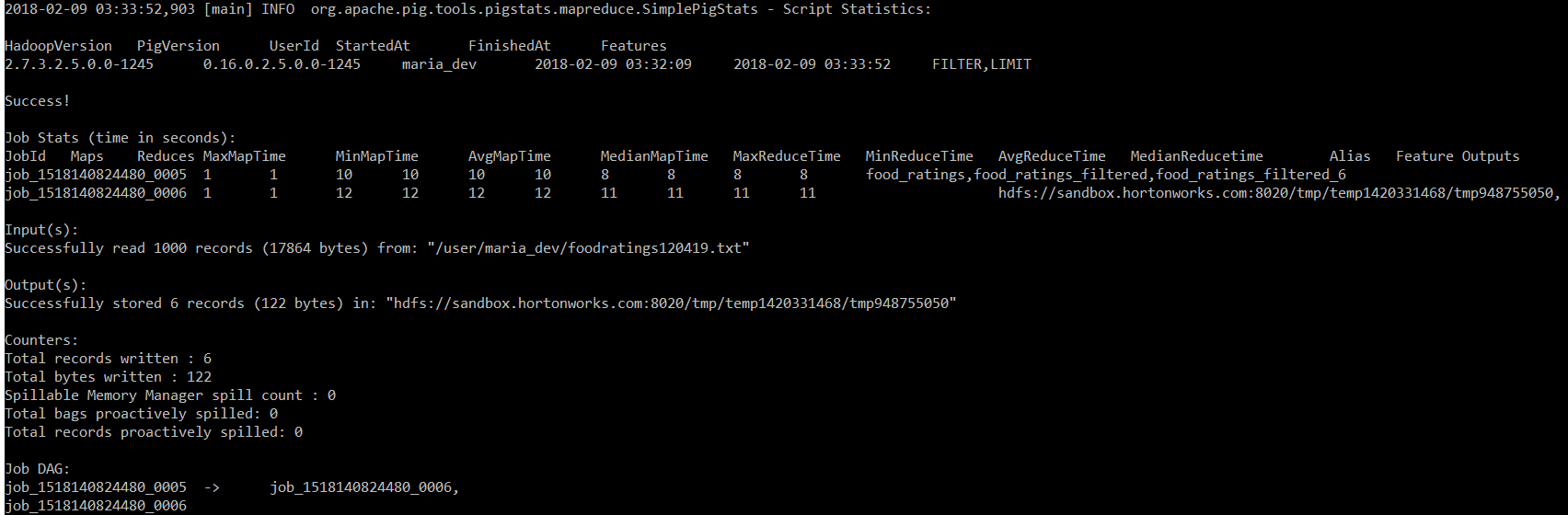
DESCRIBE food\_ratings\_profile;

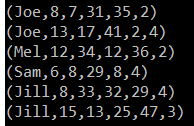
DUMP food\_ratings\_profile;

1. Create yet another relation from the initial (food\_ratings) relation. This new relation should only include tuples (records) where f1 < 20 and f3 > 5. Call this relation ‘food\_ratings\_filtered’.



Write 6 records of this relation out to the console.





Submit the pig latin statements you used and the six records printed out to the console as the result of this exercise.

**Command Executed:**

Magic Number = 120419

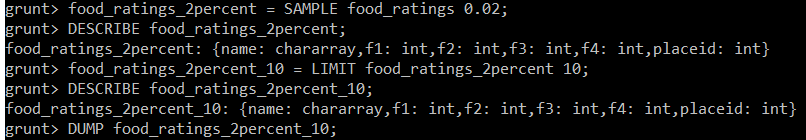
food\_ratings\_filtered = FILTER food\_ratings BY (f1 < 20) AND (f3 > 5);

food\_ratings\_filtered\_6 = LIMIT food\_ratings\_filtered 6;

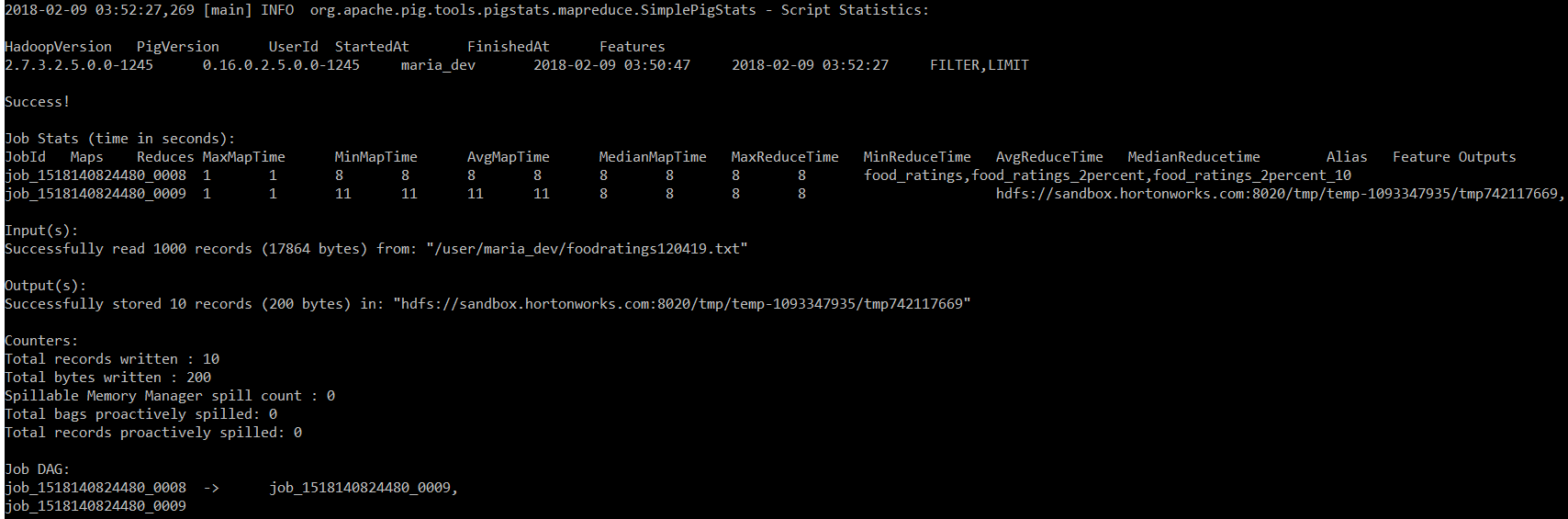
DESCRIBE food\_ratings\_filtered\_6;

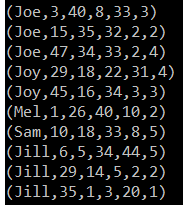
DUMP food\_ratings\_filtered\_6;

1. Using the initial (food\_ratings) relation, write and execute a sequence of pig latin statements that creates another relation, call it ‘food\_ratings\_2percent’, holding a random selection of 2% of the records in the initial relation.



Write 10 of the records out to the console.





Submit the pig latin statements and the records printed out to the console.

**Command Executed:**

Magic Number = 120419

food\_ratings\_2percent = SAMPLE food\_ratings 0.02;

DESCRIBE food\_ratings\_2percent;

food\_ratings\_2percent\_10 = LIMIT food\_ratings\_2percent 10;

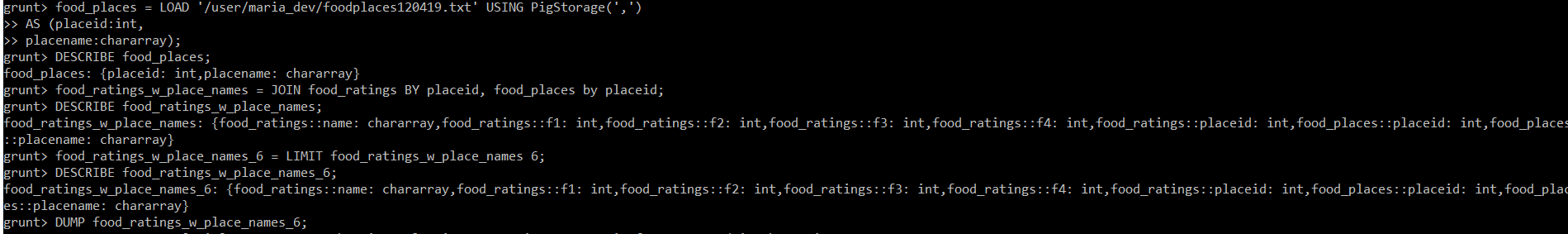
DESCRIBE food\_ratings\_2percent\_10;

DUMP food\_ratings\_2percent\_10;

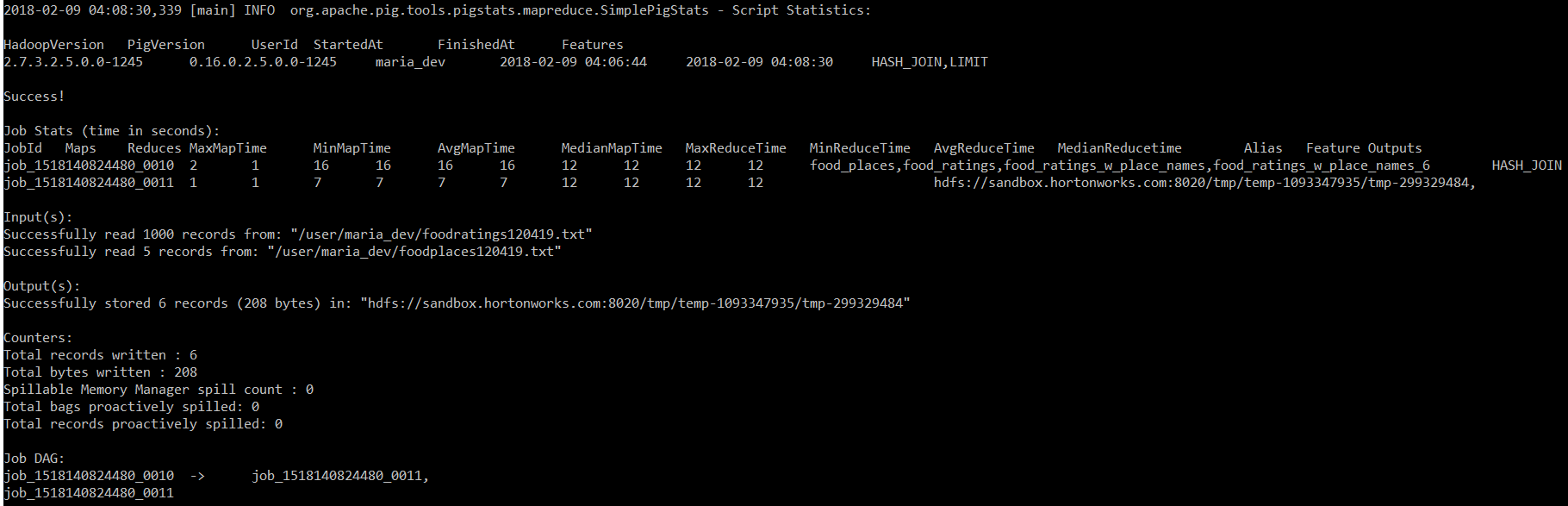
1. Write and execute a sequence of pig latin statements that loads the foodplaces file as a relation. Call the relation ‘food\_places’. The load command should associate a schema with this relation where the first attribute is referred to as ‘placeid’ and is of type int and the second attribute is referred to as ‘placename’ and is of type chararray.

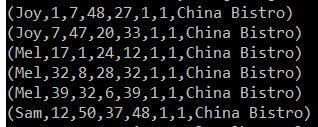
Execute the describe command on this relation.

Now perform a join between the initial place\_ratings relation and the food\_places relation on the placeid attributes to create a new relation called ‘food\_ratings\_w\_place\_names’. This new relation should have all the attributes (columns) of both relations. The new relation will allow us to work with place ratings and place names together.



Write 6 records of this relation out to the console.





Submit the pig latin statements you used and the six records printed out to the console as the result of this exercise.

**Command Executed:**

Magic Number = 120419

food\_places = LOAD '/user/maria\_dev/foodplaces120419.txt' USING PigStorage(',')

AS (placeid:int,

placename:chararray);

DESCRIBE food\_places;

food\_ratings\_w\_place\_names = JOIN food\_ratings BY placeid, food\_places by placeid;

DESCRIBE food\_ratings\_w\_place\_names;

food\_ratings\_w\_place\_names\_6 = LIMIT food\_ratings\_w\_place\_names 6;

DESCRIBE food\_ratings\_w\_place\_names\_6;

DUMP food\_ratings\_w\_place\_names\_6;

***Extra Credit:***

1. Write a half page summary of the following article on the blackboard in section “Articles:”

“Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing”

**Answer:** Following is the summary of “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing” article:

* The article introduces Resilient Distributed Datasets (RDDs), a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner.
* RDDs are motivated by two types of applications that current computing frameworks handle inefficiently: iterative algorithms and interactive data mining tools. In both cases, keeping data in memory can improve performance by an order of magnitude.
* To achieve fault tolerance efficiently, RDDs provide a restricted form of shared memory, based on coarse-grained transformations rather than fine-grained updates to shared state. However, we show that RDDs are expressive enough to capture a wide class of computations, including recent specialized programming models for iterative jobs, such as Pregel, and new applications that these models do not capture.

**Business Case for RDD:**

* Cluster computing frameworks like MapReduce have been widely adopted for large-scale data analytics. These systems let users write parallel computations using a set of high-level operators, without having to worry about work distribution and fault tolerance.
* Although current frameworks provide numerous abstractions for accessing a cluster’s computational resources, they lack abstractions for leveraging distributed memory. This makes them inefficient for an important class of emerging applications: those that reuse intermediate results across multiple computations.
* Data reuse is common in many iterative machine learning and graph algorithms, including PageRank, K-means clustering, and logistic regression.
* Also, in interactive data mining, where a user runs multiple adhoc queries on the same subset of the data. Unfortunately, in most current frameworks, the only way to reuse data between computations (e.g., between two MapReduce jobs) is to write it to an external stable storage system, e.g., a distributed file system. This incurs substantial overheads due to data replication, disk I/O, and serialization, which can dominate application execution times.

**Solution:**

* Resilient Distributed Datasets (RDDs) enable efficient data reuse in a broad range of applications. RDDs are fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators.
* The article then further highlights the advantages, implementation and limitation of RDD implemented in Spark system. The article then compares the performance of RDD against the industry standard big data technologies/applications benchmarks like Logistic Regression, K-Mean clustering etc.

**Conclusion:**

* Resilient Distributed Datasets (RDDs) provide an efficient, general-purpose and fault-tolerant abstraction for sharing data in cluster applications. RDDs can express a wide range of parallel applications, including many specialized programming models that have been proposed for iterative computation, and new applications that these models do not capture.
* Unlike existing storage abstractions for clusters, which require data replication for fault tolerance, RDDs offer an API based on coarse-grained transformations that lets them recover data efficiently using lineage.