NoSQL Systems : Assignment 3

Krutik Patel
IMT2021024

Suyash Chavan
IMT2021048

Sai Madhavan G
IMT2021101

I. Problem 1 - PIG

A. *Part A : Top 10 frequently occurring predicates*

*1) **Relevant files**:* The files from our submission relevant to this subpart are:

- `parta/problem1_a.txt`
- `parta/output/1a/`
- `parta/src/1a.pig`
- `parta/logs/1a.log`

*2) **Program Logic**:* The program logic to find Top 10 frequently occurring predicates is as follows:

a) *LOAD Data:* Load the data from the TSV file `yago_full_clean.tsv` into a relation named `lines`, using the `PigStorage` function to specify the delimiter as a space (`' '`) and define the schema with three fields: `subject`, `predicate`, and `object`, all of type `chararray`.

b) *GROUP Data:* Group the `lines` relation by the `predicate` field, creating groups of tuples with the same `predicate` value. This step aggregates the data based on the `predicate`, resulting in a relation named `predicate_groups`.

c) *COUNT Predicate Groups:* For each group of lines with the same `predicate`, count the number of lines in the group using the `COUNT` function. Rename the `predicate` field as `predicate` and the count result as `count`. Store the results in a relation named `predicate_counts`.

d) *ORDER Predicate Counts:* Order the `predicate_counts` relation by the `count` field in descending order (`DESC`), and if there are ties, then by the `predicate` field in ascending order (`ASC`). This step sorts the counts in descending order and within each count, sorts the predicates alphabetically.

e) *LIMIT Top 10:* Select the top 10 records from the `ordered_predicate_counts` relation. This step limits the output to only the top 10 predicates with the highest counts.

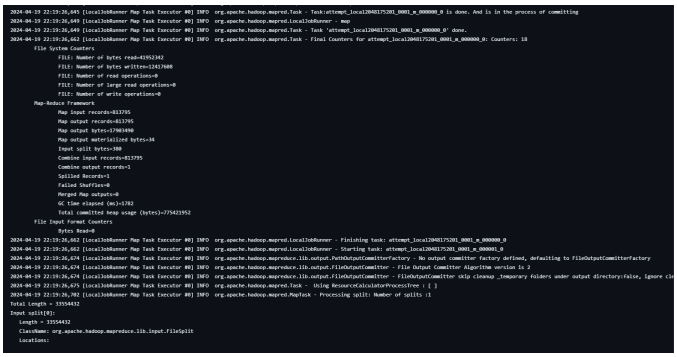


Fig. 1. The beginning of execution for problem 1.a

f) *STORE Results:* Store the top 10 records in a directory named `1a`.

*3) **Pseudocode**:* The pseudocode to find Top 10 frequently occurring predicates is as follows:

Algorithm 1 Top 10 frequently occurring predicates

1. Load Data: Load data from TSV file into relation *lines* using `PigStorage(' ')`
2. Group Data: Group *lines* by *predicate* field, creating *predicate_groups*
3. Count Predicate Groups: Count lines in each group, store results in *predicate_counts*
4. Order Predicate Counts: Order *predicate_counts* by count DESC, predicate ASC, store in *ordered_predicate_counts*
5. Select Top 10: Select top 10 records from *ordered_predicate_counts*, store in *top10*
6. Store Results: Store *top10* records in directory `1a`

*4) **Observations**:* Refer figures 1 and 2

a) Metrics:

- **Number of Maps:** 4
- **Number of Reducers:** 1
- **Time taken:** 1 minute 33 seconds
- **Total Records written:** 10
- **Features Used:** `GROUP_BY, ORDER_BY, LIMIT`

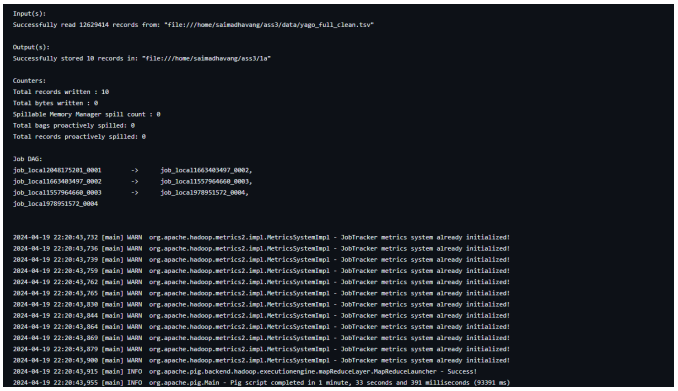


Fig. 2. The end of execution for problem 1.a

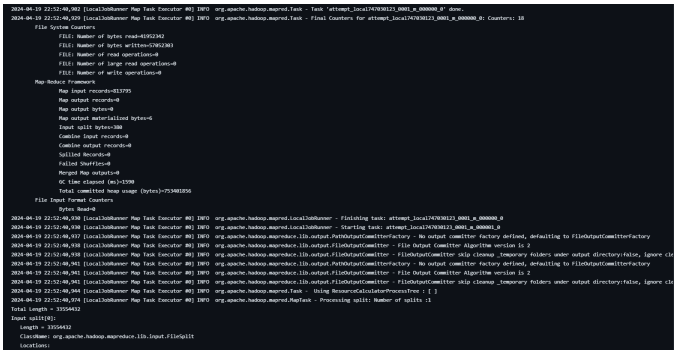


Fig. 3. The beginning of execution for problem 1.b

B. *Part B : Object values of the hasGivenName predicate*

*1) **Relevant files**:* The files from our submission relevant to this subpart are:

- `parta/problem1_b.txt`
- `parta/output/1b/`
- `parta/output/1b`$_{d}istinct$`/parta/src/1b.pig`
- `parta/src/1b`$_{d}istinct.pig$`parta/logs/1b.log`
- `parta/logs/1b`$_{d}istinct.log$

*2) **Program Logic**:* The program logic to find Object values of the hasGivenName predicate is as follows:

a) *LOAD Data:* Load the data from the TSV file `yago_full_clean.tsv` into a relation named `records`, specifying the delimiter as a space (`' '`) and defining the schema with three fields: `subject`, `predicate`, and `object`, all of type `chararray`.

b) *FILTER by Predicate:* Filter the `records` relation to get all subjects with the `<livesIn>` predicate. Store the filtered records in a relation named `lives_in_records`.

c) *GROUP by Subject:* Group the `lives_in_records` relation by the `subject` field, creating groups of tuples with the same `subject`. Store the result in a relation named `lives_in_groups`.

d) *COUNT LivesIn Records:* For each group in `lives_in_groups`, count the number of records using the `COUNT` function. Rename the `subject` field as `li_subject` and the count result as `count`. Store the results in a relation named `subject_li_count`.

e) *FILTER Valid Subjects:* Filter the `subject_li_count` relation to keep only those subjects with a count greater than 1. Store the filtered records in a relation named `valid_subjects`.

f) *FILTER by hasGivenName Predicate:* Filter the `records` relation to get all records with the `<hasGivenName>` predicate. Store the filtered records in a relation named `given_name_records`.

g) *JOIN Valid Subjects with Given Names:* Join the `valid_subjects` relation with the `given_name_records` relation based on the `subject` field. Store the joined records in a relation named `joined_records`.

h) *GENERATE Objects:* For each record in `joined_records`, extract the `object` field. Store the extracted objects in a relation named `objects`.

i) *STORE Results:* Store the `objects` relation into the directory `./output/1b`.

*3) **Pseudocode**:* The pseudocode to find Object values of the hasGivenName predicate is as follows:

Algorithm 2 Object values of the hasGivenName predicate

1. Load Data: Load data from TSV file into relation *records* using `PigStorage(' ')`
2. Filter by Predicate: Filter *records* to get subjects with the `<livesIn>` predicate, store in *lives_in_records*
3. Group by Subject: Group *lives_in_records* by `subject`, store in *lives_in_groups*
4. Count LivesIn Records: Count records for each `subject` in *lives_in_groups*, store in *subject_li_count*
5. Filter Valid Subjects: Filter *subject_li_count* for subjects with count ¿ 1, store in *valid_subjects*
6. Filter by hasGivenName Predicate: Filter *records* to get records with the `<hasGivenName>` predicate, store in *given_name_records*
7. Join Valid Subjects with Given Names: Join *valid_subjects* with *given_name_records* on `subject`, store in *joined_records*
8. Extract Objects: Extract `object` field from *joined_records*, store in *objects*
9. Store Results: Store *objects* in directory `./output/1b`

*4) **Observations**:* Refer figures 3, 4, 5 and 6

a) Metrics:

- **Number of Maps:** 2
- **Number of Reducers:** 1
- **Time taken:** 1 minute 28 seconds
- **Total Records written:** 10728
- **Features Used:** `GROUP_BY, HASH_JOIN, FILTER`

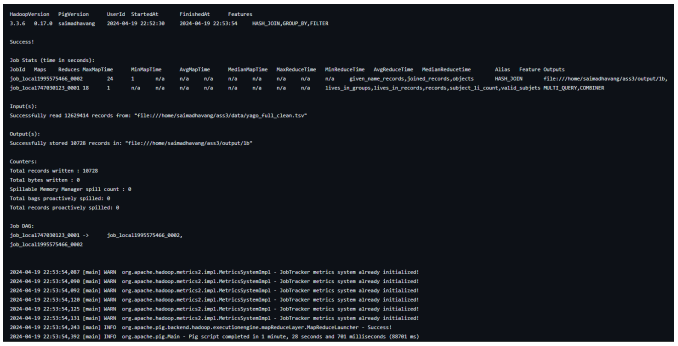
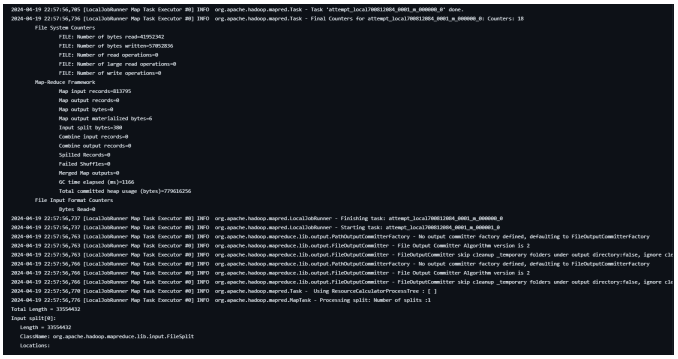


Fig. 4. The end of execution for problem 1.b



II. Problem 2 - PIG

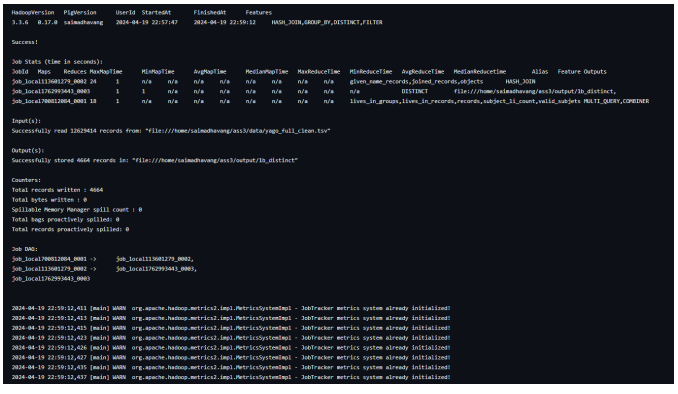
*1) **Relevant files**:* The files from our submission relevant to this subpart are:

- `parta/problem2.txtparta/src/SPCOUNT.java`
- `parta/src/2.pig`
- `parta/jars/SPCount.jar`
- `parta/outputs/2`
- `parta/logs/2.log`

*2) **Program Logic**:*

a) UDF:

- We inherit our custom UDF class from the `EvalFunc` class and the `Algebraic` interface.



- We need to use the `EvalFunc` as we need to output a single value for each input tuple through the `count` method.
- We need to use the `Algebraic` interface as we need to apply our function through a bag of tuples containing all records.
- We implement the `Initial`, Intermed and `Final` classes to inform `Algebraic` methods what to do in the map, combine and reduce phases respectively.
- In Initial, we simply call the `count` method.
- The count method takes 3 inputs: bag of tuples, subject and predicate. It then iterates through each of tuple in the bag while counting the tuples for which the subject and predicate matches. It returns this result of this counting.
- In `Intermed` and `Final`, we call the `sum` method which returns the sum of the inputs.
- We compile this into the `SPCount.jar` file which we can use from pig.

b) Pig:

a) *Load Data:* Load data from the TSV file `yago_full_clean.tsv` into a relation named `records`, specifying the delimiter as a space and defining the schema with three fields: `subject`, `predicate`, and `object`, all of type `chararray`.
b) *Register Jar:* Register the JAR file `SPCount.jar` containing the custom Pig UDF (User-Defined Function).
c) *Group All Records:* Group all records together to process them collectively.
d) *Compute SP Count:* For each group of all records, compute the count of occurrences of the specific subject-predicate pair using the custom UDF `SPCOUNT` with arguments `'<Alice_Roberts>'` and `'<isCitizenOf>'`.
e) *Store Results:* Store the computed SP count into the directory `./output/2`.

*3) **Pseudocode:** :* The pseudocode to find count of unique objects associated with subject-predicate combination is as follows:

Algorithm 3 find count of unique objects associated with subject-predicate combination

1. Load Data: Load data from TSV file into relation *records* with schema (subject:chararray, predicate:chararray, object:chararray)
2. Register Jar: Register custom Pig UDF contained in *SPCount.jar*
3. Group All Records: Group all records together
4. Compute SP Count: For each group, compute count of occurrences of subject-predicate pair $' < Alice\_Roberts >'$ and $' < isCitizenOf >'$ using *SPCOUNT* UDF
5. Store Results: Store computed SP count in directory *./output/2*

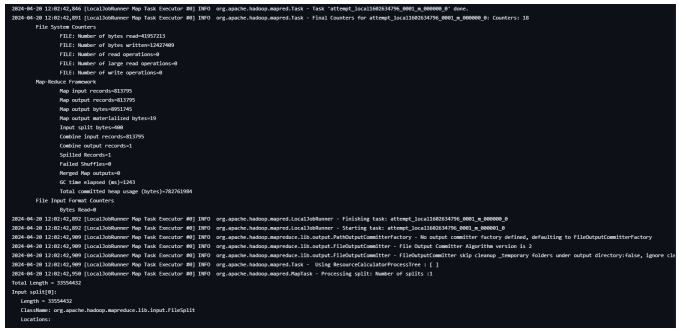


Fig. 7. The beginning of execution for problem 2

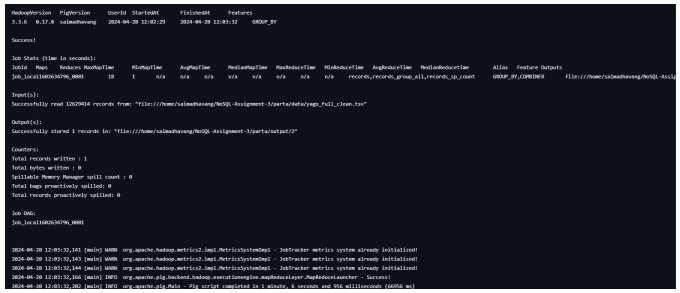


Fig. 8. The beginning of execution for problem 2

*4) **Observations**:* The following were the observed metrics when the input was 'Alice_Roberts ' and 'isCitizenOf ' (Fig. 7 and 8):

- **Number of Maps:** 18
- **Number of Reducers:** 1
- **Time taken:** 1 minute 6 seconds
- **Features Used:** `GROUP_BY, COMBINER`

III. Problem 3 - HIVE

A. *Part A : Top 3 frequently occurring predicates*

*1) **Relevant files**:* The files from our submission relevant to this subpart are:

- `partb/problem1/problem2_a.txt`
- `partb/problem1/output/2a/`
- `partb/problem1/src/2a.hiveql`
- `partb/problem1/logs/2a.log`

*2) **Program Logic**:* The program logic to find Top 3 frequently occurring predicates is as follows:

a) *Create Table:* Create a table named `records` with three columns: `subject`, `predicate`, and `object`, specifying the delimiter as a space and defining the location as `'../tables'`.

b) *Load Data:* Load data from the TSV file `yago_full_clean.tsv` into the `records` table.

c) *Group by Predicate:* Create a view named `predicate_groups` by grouping the `records` table by the `predicate` column and counting the occurrences of each predicate.

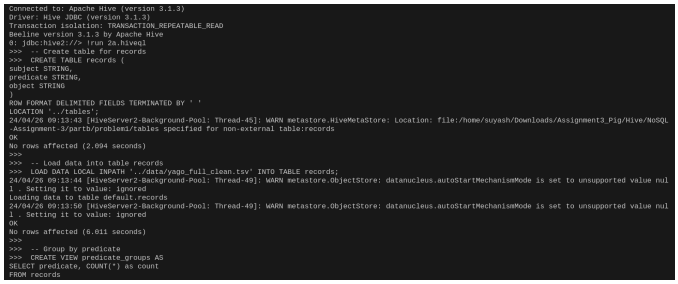


Fig. 9. The beginning of execution for problem 3.a

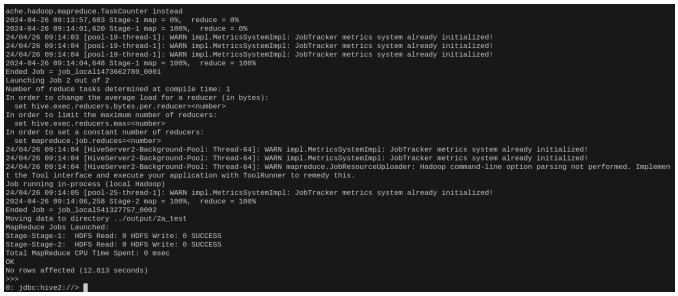


Fig. 10. The end of execution for problem 3.a

d) *Order by Count:* Create a view named `ordered_predicate_counts` by ordering the `predicate_groups` view by count in descending order.

e) *Select Top 3 Records:* Create a view named `top3` by selecting the top 3 records from the `ordered_predicate_counts` view.

f) *Store Results:* Store the top 3 records from the `top3` view into the directory `'../output/2a'` with space delimiter.

*3) **Pseudocode**:* The pseudocode to find Top 3 frequently occurring predicates is as follows:

Algorithm 4 find Top 3 frequently occurring predicates

1. Create Table: Create a table named `records` with columns `subject`, `predicate`, and `object`, delimited by space, and located at `'../tables'`.
2. Load Data: Load data from `yago_full_clean.tsv` into the `records` table.
3. Group by Predicate: Group the `records` table by `predicate`, counting occurrences for each predicate, and store in `predicate_groups` view.
4. Order by Count: Order `predicate_groups` by count in descending order, storing in `ordered_predicate_counts` view.
5. Select Top 3 Records: Select top 3 records from `ordered_predicate_counts`, storing in `top3` view.
6. Store Results: Store the top 3 records from `top3` into directory `'../output/2a'` with space delimiter.

*4) **Observations**:* Refer figures 9 and 10

a) Metrics:

- **Number of Maps:** 1
- **Number of Reducers:** 1
- **Time taken:** 12.81 seconds (Faster than PIG)
- **Features Used:** `GROUP_BY`, `VIEW`, `ORDER_BY`, `LIMIT`

B. Part B : Object values of the hasGivenName predicate

*1) **Relevant files**:* The files from our submission relevant to this subpart are:

- `partb/problem1/problem2_b.txt`
- `partb/problem1/output/2b/`
- `partb/problem1/src/2b.hiveql`
- `partb/problem1/logs/2b.log`

*2) **Program Logic**:* The program logic to find Object values of the hasGivenName predicate is as follows:

a) **Load Data:** Load data from the TSV file into a Hive table named `records2`. Define the schema with columns `subject`, `predicate`, and `object`. Specify the delimiter as space and the location as `'../tables'`.

b) **Load Data into Table:** Load data from the local path `'../data/yago_full_clean.tsv'` into the `records2` table.

c) **Get Subjects with Multiple 'livesIn' Predicates:**

1) Create a view `lives_in_records` to select all records from `records2` where the predicate is $' < livesIn >'$.
2) Create a view `lives_in_groups` to count the occurrences of each subject in `lives_in_records`.
3) Create a view `valid_subjects` to filter `lives_in_groups` to keep only subjects with more than one $' < livesIn >'$ predicate.

d) **Join with 'hasGivenName' Predicate:**

1) Create a view `given_name_records` to select all records from `records2` where the predicate is '¡hasGivenName¿'.
2) Create a view `joined_records` by joining `valid_subjects` with `given_name_records` on the subject field.

e) **Store Result:** Store the result from `joined_records` into the HDFS directory `'../output/2b'`. Fields should be terminated by space.

*3) **Pseudocode**:* The pseudocode to find Object values of the hasGivenName predicate is as follows:

Algorithm 5 find Object values of the hasGivenName predicate

1. Load Data: Load data from the TSV file into a Hive table named `records2`, with columns `subject`, `predicate`, and `object`, delimited by space, and located at `'../tables'`.

2. Load Data into Table: Load data from the local path `'../data/yago_full_clean.tsv'` into the `records2` table.

3. Get Subjects with Multiple $'livesIn'$ Predicates:

Create a view `lives_in_records` to select all records from `records2` where the predicate is $' < livesIn >'$.

Create a view `lives_in_groups` to count the occurrences of each subject in `lives_in_records`.

Create a view `valid_subjects` to filter `lives_in_groups` to keep only subjects with more than one $' < livesIn >'$ predicate.

4. Join with $'hasGivenName'$ Predicate:

Create a view `given_name_records` to select all records from `records2` where the predicate is $' < hasGivenName >'$.

Create a view `joined_records` by joining `valid_subjects` with `given_name_records` on the subject field.

5. tore Result: Store the result from `joined_records` into the HDFS directory `'../output/2b'`, with fields terminated by space.

*4) **Observations**:* Refer figures 11 and 12

a) Metrics:

- **Number of Maps:** 3
- **Number of Reducers:** 1
- **Time taken:** 39.55 seconds (Faster than PIG)
- **Features Used:** `GROUP_BY`, `JOIN`, `VIEW`

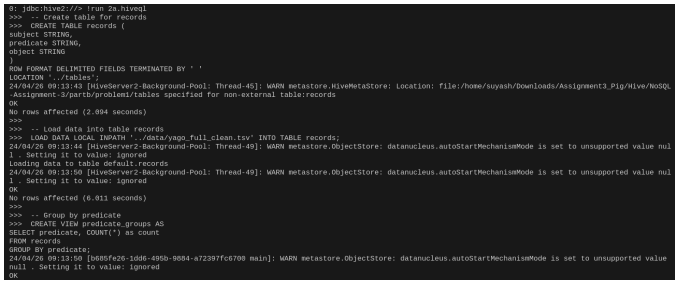


Fig. 11. The beginning of execution for problem 3.b

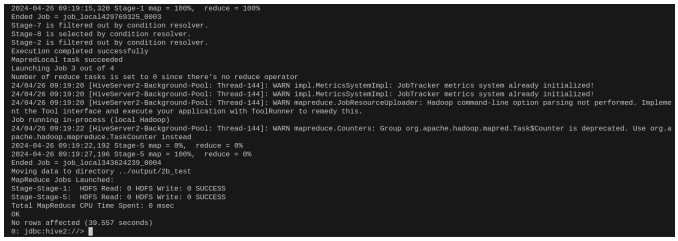


Fig. 12. The end of execution for problem 3.b

Fig. 13. Create query for partitioning and bucketing

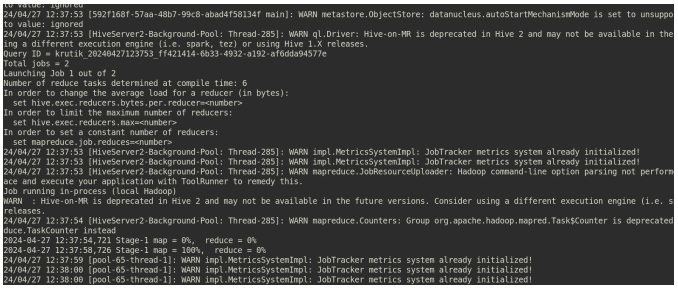


Fig. 14. Insert query for partitioning and bucketing

Fig. 15. Create query for partitioning and no bucketing

IV. Problem 4 - HIVE

A. *Database Creation:*

*1) **Partitioning and Bucketing : refer 13, 14***

: Program logic :

a) **Create Table:** Create a Hive table named `yago_part_buck` with columns `subject` and `object`. Partition the table by the `predicate` column. Cluster the table by the `subject` column into 6 buckets.

b) **Specify Row Format:** Define the row format for the table. Fields are delimited by '\t' (tab) and lines are terminated by '\n' (newline).

c) **Specify Location:** Specify the location for storing the table data as 'yago_part'.

Pseudocode :

Algorithm 6 Creating Table `yago_part_buck`

1. Create Table `yago_part_buck`:
Specify columns: `subject` (string), `object` (string)
Partition by: `predicate` (string)
Cluster by: `subject` (string) into 4 buckets
Row format: delimited by '\t', lines terminated by '\n'
Location: 'yago_part'

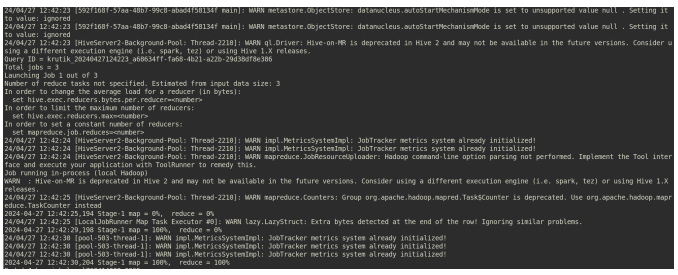


Fig. 16. Insert query for partitioning and no bucketing

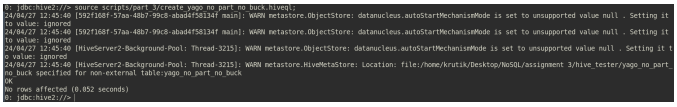


Fig. 17. Create query for no partitioning and no bucketing

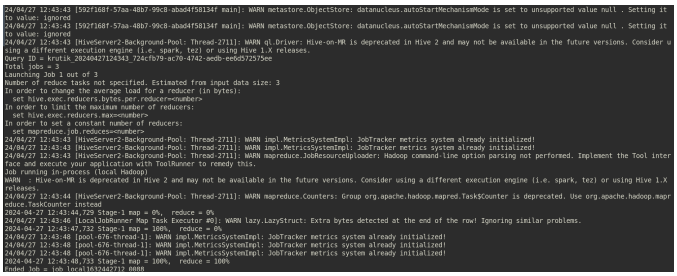


Fig. 18. Insert query for no partitioning and no bucketing

*2) **Partitioning but not Bucketing : refer 15, 16***

: Program logic :

a) **Create Table:** Define the table `yago_part_no_buck` with columns `subject` and `object`, both of type string.

b) **Partitioning:** Partition the table by the column `predicate`, which is also of type string.

c) **Row Format:** Specify the row format as delimited by '\t', indicating tab separation, and terminated by '\n', indicating newline.

d) **Location:** Set the location of the table data to 'yago_part_no_buck'.

Pseudocode :

Algorithm 7 Creating Table `yago_part_no_buck`

1. Create Table `yago_part_no_buck`:
Specify columns: `subject` (string), `object` (string)
Partition by: `predicate` (string)
Row format: delimited by '\t', lines terminated by '\n'
Location: 'yago_part_no_buck'

*3) **Neither Partitioning nor Bucketing : refer 17, 18***

: Program logic :

1. **Create Table:** Define the table `yago_no_part_no_buck` with columns `subject`, `predicate`, and `object`, all of type string.
2. **Row Format:** Specify the row format as delimited by space ('␣'), indicating space separation, and terminated by '\n', indicating newline.
3. **Location:** Set the location of the table data to 'assignment/'.

Pseudocode :

B. *Query Logic :*

*1) **Partitioning and Bucketing : refer 19***

: Program logic :

1. Insert Overwrite Local Directory 'output/yago_part_buck':

Algorithm 8 Creating Table yago_no_part_no_buck

1. Create Table `yago_no_part_no_buck`:

Specify columns: `subject` (string), `predicate` (string), `object` (string)

Row format: delimited by space (' '), lines terminated by '\n'

Location: 'assignment/'

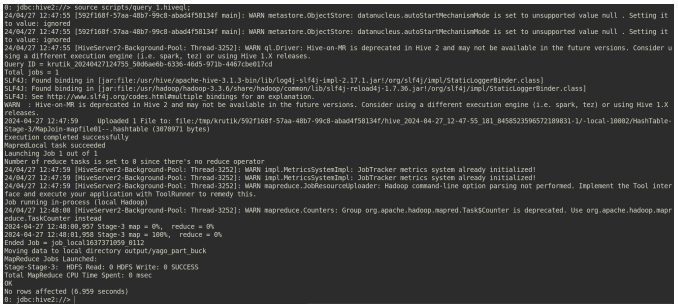


Fig. 19. Query with partitioning and bucketing

2. Specify row format as delimited by '\t', fields terminated by '\n', and stored as textfile.
3. Select all columns from `yago_part_buck y1` joined with `yago_part_buck y2` on the condition `y1.subject=y2.subject`.
4. Apply a condition to filter the rows where `y1.predicate` is $' < hasGivenName >'$ and `y2.predicate` is $' < livesIn >'$.

Observations :

a) Metrics:

- **Output Tuple Count:** 67, 028
- **Time taken:** 6.959 seconds
- **Features Used:** `PARTITION, DISTRIBUTE, JOIN`

*2) **Partitioning but not Bucketing : refer 20***

: Program logic :

1. Insert Overwrite Local Directory 'output/yago_part_no_buck':
2. Specify row format as delimited by '\t', fields terminated by '\n', and stored as textfile.
3. Select all columns from `yago_part_no_buck y1` joined with `yago_part_no_buck y2` on the condition `y1.subject=y2.subject`.
4. Apply a condition to filter the rows where `y1.predicate` is $' < hasGivenName >'$ and `y2.predicate` is $' < livesIn >'$.

Observations :

a) Metrics:

- **Output Tuple Count:** 67, 028
- **Time taken:** 5.458 seconds
- **Features Used:** `PARTITION, JOIN`

*3) **Neither Partitioning nor Bucketing : refer 21***

: Program logic :

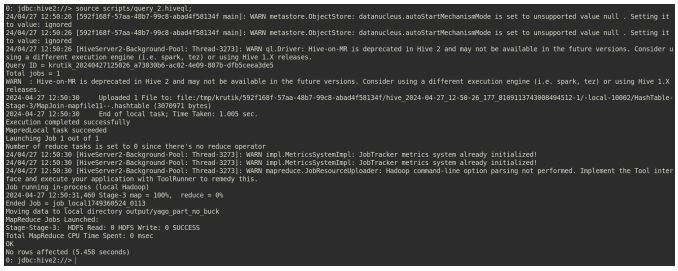


Fig. 20. Query with partitioning and no bucketing

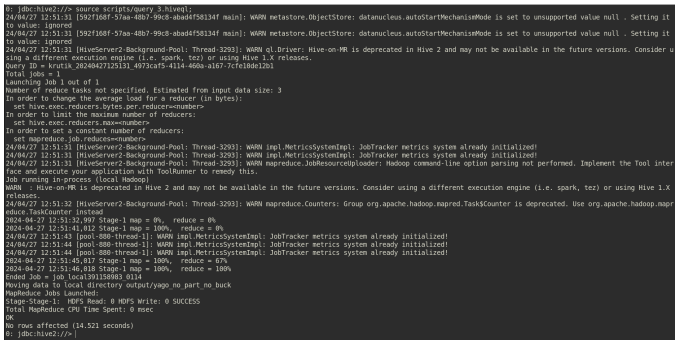


Fig. 21. Query with no partitioning and no bucketing

1. Insert Overwrite Local Directory 'output/yago_no_part_no_buck':
2. Specify row format as delimited by '\t', fields terminated by '\n', and stored as textfile.
3. Select all columns from `yago_no_part_no_buck y1` joined with `yago_no_part_no_buck y2` on the condition `y1.subject=y2.subject`.
4. Apply a condition to filter the rows where `y1.predicate` is $' < hasGivenName >'$ and `y2.predicate` is $' < livesIn >'$.

Observations :

a) Metrics:

- **Output Tuple Count:** 67, 028
- **Time taken:** 14.521 seconds
- **Features Used:** `JOIN`