Query Processing over Streaming Data Using Flink

Abstract

While the study of query processing on static databases is well-established, many contemporary applications necessitate the evaluation of queries on databases that are subject to continuous updates. Tasks such as real-time analytics demand frequent refreshment of query results to accommodate high update rates. This introduces a new set of challenges for query processing, given that the validity of a query result can be compromised or become outdated almost instantaneously with any change in the underlying data. This project focuses on the implementation of Cquirrel, a continuous query processing engine built on top of Flink. Specifically, the implementation focuses query Q7 from the TPC-H benchmarks[1]

Index Terms

Query evaluation under updates; incremental view maintenance; acyclic joins; sliding windows

I. Introduction

Query evaluation on a static database is a well studied problem. In many emerging applications, queries are evaluated on a database that is being continuously updated. Examples include online data analytics, stock price prediction, sensor monitoring, network traffic analytic, etc. With massive data being collected everywhere, it is very important to monitor various statistics (often in the form of SPJA queries) in real time, in order to make informed decisions. In these applications, updates to the database are being made at high speeds and the query processing system must maintain the query answer with high throughput and low latency.

A. Problem definition

Let db be the contents of the current database and $Q(db)$ denote the results of evaluating query Q on db. For an update u to db where u can be either the insertion or deletion of a tuple, we write $db + u$ as the new database instance after applying u To facilitate the computation, some data structure on db denoted $\mathcal{D}(db)$, can be maintained. Depending on the application, there are two output modes. (1) Delta enumeration: Given $\mathcal{D}(db)$ and u, the system should output ΔQ, i.e., all differences between $Q(db)$ and $Q(db + u)$. (2) Full enumeration: Given $\mathcal{D}(db)$, all query results in $Q(db)$ can be enumerated with constant delay [1].

B. The demonstration system

This project mainly implement Cquirrel (Continuous query process ing ove r acyclic rel ational schemas) [2], a system for continuous query processing built on top of Flink [3]. Cquirrel specifically targets at queries with multi-way foreign-key joins over an acyclic schema. The output model of my implementation is delta enumeration. The user first registers a query on an initially empty database. An update sequence (containing insertions and deletions) are then fed into Cquirrel in the form of a DataStream of Flink. As updates are being processed, deltas of the query answer are generated as an output DataStream in real time.

II. Related Work

A. Foreign-key Acyclic Schemas

Let $R_1(\bar{x}_1)$, $R_2(\bar{x}_1)$, . . . be the relations in the database, where $\bar{x}_i = \{x_{i1}, x_{i2}, ...\}$ denotes the set of attributes of relation R_i . Let $PK(R)$ be the primary key of R, and $\bar{x}_i = \{x_{i1}, x_{i2}, \cdots\}$ to indicate that $PK(R) = x_{i1}$. At any time instance, all tuples in a relation must have unique values on the primary key. For a composite primary key, we (conceptually) create a combined primary key by concatenating the attributes in the composite key, while the original attributes in the composite key are treated as regular, non-key attributes.

An attribute x_{ik} of a relation R_i is a foreign key referencing the primary key x_{j1} of relation R_j , if the x_{ik} values taken by tuples in R_i must appear in x_{j1} [4]. If a relation has a composite foreign key, we similarly create a combined attribute referencing the composite primary key.

We can model all the primary-foreign key relationships as a directed graph: Each vertex represents a relation, and there is a directed edge from R_i to R_j if R_i has a foreign key referencing the primary key of R_j . This graph is called the foreign key graph. If its foreign key graph is an acyclic directed graph (DAG), the schema is foreign key acyclic. The TPC-H schema, shown in Figure 1, is foreign key acyclic.

[1]My implementation is publicly available at https://github.com/kentxusy/Query-Stream-Processing-flink

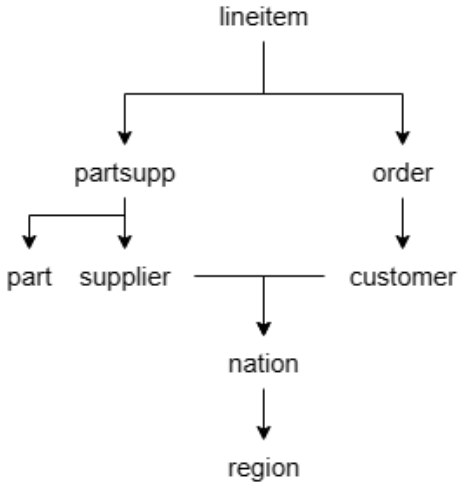

Fig. 1 The foreign-key graph of TPC-H schema.

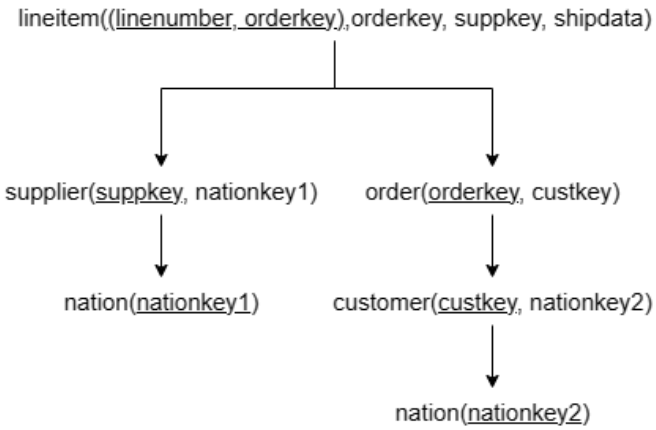

Fig. 2 The foreign-key graph of TPC-H Q7.

B. Foreign-key Acyclic Joins

We start by considering multi-way natural joins of the form

$$\sigma_{\phi_1} R_1(\bar{x}_1) \bowtie \cdots \bowtie \sigma_{\phi_n} R_n(\bar{x}_n), \tag{1}$$

where ϕ_i is a selection condition on $\bar{x}_i$. We require the query to be a foreign key join, i.e., for any attribute x appearing in more than one relation in (1), x must be the primary key of at least one of them. We can similarly define the foreign key graph of a query: Introduce a vertex for each R_i , and build a directed edge from R_i to R_j if $PK(R_j)$ appears as an attribute of R_i . The query is foreign-key acyclic, if this graph is a DAG with one vertex (relation) having in-degree 0, which is called the root of the query. This definition implies that every relation in the multi-way join (1) must have a primary key, which is joined with another relation.

All multi-way joins in the TPC-H queries satisfy these requirements. The multi-way join of TPC-H Q7 used in this project, written as a natural join,is

$$\text{lineitem(orderkey, suppkey, shipdate}, \cdots) \bowtie \text{supplier(}\underline{\text{suppkey}}\text{, nationkey1)} \bowtie \text{nation(}\underline{\text{nationkey1}}\text{)}$$
$$\bowtie \text{orders(}\underline{\text{orderkey}}\text{, custkey)} \bowtie \text{customer(}\underline{\text{custkey}}\text{, nationkey2)} \bowtie \text{nation(}\underline{\text{nationkey2}}\text{)}$$

The foreign key graph of this query is shown in Figure 2, which is not a subgraph of Figure 1. This the query involves multiple logical copies of the same relation(nation).

C. Live Tuples

In the context of acyclic joins, every join result must include a distinct tuple in the root relation. The tuples in the root relations that can join with one tuple in every other relation are referred to as “alive” or “live tuples”. The efficient maintenance of join results is based on keeping track of these live tuples.

For a leaf relation R, all its tuples are alive. For a non-leaf relation R, a tuple t in R is alive if it can join with one tuple in every relation in its descendant relations.

However, using this definition directly to maintain the set of live tuples is expensive. We introduce the notion of alive on a child, which recursively depends on a child tuple being alive: Let R be a non-leaf relation in τ , and R_c a child relation of R. A t tuple in R is alive on R_c if there is an alive tuple t_c in R_c such that $\pi_{PK(R_c)} t = \pi_{PK(Rc)} t_c$.

If whether a tuple in a relation R is alive could be determined by only checking whether it is alive on its child relations, we would be able to close the recursion and arrive at an efficient maintenance algorithm. However this is only a necessary condition. Assertion keys are introduced to fix the problem.

III. Implementation

A. Data Structures

Let $L(R)$ be the set of live tuples in R and $N(R)$ the set of non-live tuples in R. The data structure $D(db)$ maintained by the algorithm consists of the following indexes for each relation $R(\underline{x1}, x2, \ldots)$:

- $I(L(R))$: An index on $L(R)$, using $x1$ as the key.
- $I(N(R))$, for non-leaf R: An index on $N(R)$, using $x1$ as the key.

- For a non-leaf R, a counter $s(t)$ for each tuple $t \in R$ that is equal to the number of children of R on which t is alive. This counter is stored together with the tuple in $I(N(R))$.
- $I(R, Rc)$, for non-leaf R and each child $Rc \in C(R)$: An index on $\pi_{x1,PK(Rc)}R$ using $PK(Rc)$ as the key.

The standard implementation of such an index is a hash table on all the distinct key values of attributes. I use the MapState to implement the data structures mentioned above.

B. Algorithm

The implementation is mainly based on the Cquirrel algorithm [5].

1) Insert: For the process of inserting a tuple t into a relation R. We start with the assumption that t is not already present in R, a fact that can be verified in constant time. Then we must check whether t is alive or not. If R is a leaf relation, t is automatically considered alive. If not, we need to compute s(t). To do this, for each child R_c, we look up $I(L(R_c))$ by key $\pi_{PK(R_c)}t$. t is alive if the result is not empty. During this process, we also maintain the assertion keys and auxiliary keys. After calculating s(t), we check if s(t) equals $|C(R)|$ and all assertion keys are not equal to $\perp$. If these conditions are met, t is alive and we add it to $I(L(R))$. If not, we add it to $I(N(R))$. This process is detailed in Algorithm 1.

If t is not alive, it won't affect the status of other tuples or the join result. If it is alive, it will increase the $s(t_p)$ value for each tuple t_p in R_p that joins with t. These tuples can be found by probing the index $I(R, R_p)$. This could potentially make these tuples alive, triggering further updates in a bottom-up manner. When this process reaches the root, a new join result will be included in ΔQ. This process is further detailed in Algorithm 2.

Algorithm 1: Insert(t, R)

```
Result: D(db + t) and ΔQ
ΔQ ← ∅;
if R is not a leaf then
    s ← 0;
    foreach R_c ∈ C(R) do
        I(R, R_c) ← I(R, R_c) + (π_{PK(R_c)}t →
          π_{PK(R)}, PK(R_c)t);
        if π_{PK(R_c)}t ∈ I(R_c) then
            s(t) ← s(t) + 1;
        end
    end
    if s(t) = |C(R)| then
        foreach R_c ∈ C(R) do
            tc ← look up I(R_c) with key
              π_{PK(R_c)}t;
            foreach assertion key x of R do
                if x ∈ R_c then
                    if π_x t = NULL then
                        π_x t ← π_x t_c;
                    else
                        if π_x t ≠ π_x tc or π_x t = ⊥
                          then
                            π_x t ← ⊥;
                        end
                    end
                end
            end
        end
    end
    if R is a leaf or (s(t) = |C(R)| and all
      assertion keys are not ⊥) then
        Insert-Update(t, R, t);
    end
    I(N(R)) ← I(N(R)) + (π_{PK(R)}t → t);
end
```

Algorithm 2: Insert-Update(t, R, join_result)

```
Result: Update all affected tuples and ΔQ
I(L(R)) ← I(L(R)) + (π_{PK(R)}t → t);
if R is the root of T then
    ΔQ ← ΔQ ∪ {join_result};
end
else
    P ← look up I(Rp, R) with key π_{PK(R)}t;
    foreach tp ∈ P do
        s(t_p) ← s(t_p) + 1;
        if s(t_p) = |C(R_p)| then
            foreach R_c ∈ C(R_p) do
                t_c ← look up I(R_c) with key
                  π_{PK(R_c)}tp;
                foreach assertion key x of Rp do
                    if x ∈ R_c then
                        if π_x t = NULL then
                            π_x t_p ← π_x t_c;
                        else
                            if π_x t_p ≠ π_x t_c or
                              π_x t_p = ⊥ then
                                π_x t_p ← ⊥;
                            end
                        end
                    end
                end
            end
        end
        if π_x t_p ≠ ⊥ for every assertion key x then
            I(N(R_p)) ←
              I(N(R_p)) − (π_{PK(R_p)}t_p → t_p);
            Insert-Update(t_p, R_p, join_result ⋈ t_p);
        end
    end
end
```

2) Delete: Deletions of tuples can be handled in a similar fashion, but slightly simpler. To delete a tuple t from R, we first check if it is alive. If it is non-alive, then we simply delete from $I(N(R))$. Otherwise, it will affect the status of tuples in R_p . More precisely, it will decrement the $s(t_p)$ value for every $t_p \in R_p$ that joins with t. If a t_p is currently alive, this will

make it non-alive, which may trigger further updates recursively in a bottom-up fashion. When a live tuple in the root relation becomes non-alive or deleted, its corresponding join result will be deleted (included in ΔQ). Finally, we also need to update the index I (R_p , R). The deletion process is described in Algorithm 3 and 4.

Algorithm 3: Delete(t, R)

Result: $D(db-t)$and ΔQ
$\Delta Q \leftarrow \emptyset$;
if *t ∈ L(R)* **then**
  Delete-Update(t, R, t);
end
else
  $I(N(R)) \leftarrow I(N(R)) - (\pi_{PK(R)}t \rightarrow t)$;
  if *R is not the root of T* **then**
    $I(R_p, R) \leftarrow I(R_p, R) - (\pi_{PK(R)}t \rightarrow \pi_{PK(R_p),PK(R)t})$;
  end
end

Algorithm 4: Delete-Update(t, R, join_result)

Result: Update all affected tuples and ΔQ
$I(L(R)) \leftarrow I(L(R)) - (\pi_{PK(R)}t \rightarrow t)$;
if *R is the root of T* **then**
  $\Delta Q \leftarrow \Delta Q \cup \{\text{join\_result}\}$;
end
else
  P ← look up $I(Rp, R)$ with key $\pi_{PK(R)}t$;
  foreach $tp \in P$ **do**
    if $tp \in N(Rp)$ **then**
      $s(t_p) \leftarrow s(t_p) - 1$;
    end
    else
      $s(t_p) \leftarrow |C(R)| - 1$;
      $I(N(Rp)) \leftarrow I(N(Rp)) + (\pi_{PK(Rp)}t \rightarrow t)$;
      Delete-Update(tp, Rp, join_result $\cup\, tp$);
    end
  end
end

IV. Experiment

A. Algorithm Implementation

The algorithm is implemented in JAVA1.8 on my laptop with windows server. Q7 is evaluated in this project. The foreign key graph of Q7 is presented before in figure 2 Here are details of the algorithm implementation:

- Define a class for the row in each relation and partial joint relation.
- Use the connect function in flink to process the natural join between relations. CoProcessFunction is extended for each relation to generate partial joint result.
- Implemented the algorithm mentioned before. But the partial joint result is sent to the downstream instead of a single tuple.

B. Experiment Results

1) Accuracy: During the experiment, I equally divided the data into two parts. The first half of the data are inserted into an empty databases. Then for each tuple in the other half of the data, it will be first inserted and then deleted from the database. The final results will be same to the query results only on the first half of the data.

From Figure 3 and Figure 4, we can see that the output are consistent on all (supp_nation, cust_nation, l_year) tuples, which proof the accuracy of the algorithm.

C. Running Time Comparison

Figure 5 shows the running times on the FIFO update sequences for query Q7 from TPC-H benchmark with different parallelism and scale factor.

From the results, we can draw the following observations:

- For the scalability, I varied the data size with the scale factor ranging from 0.5 to 5. From Figure 6, we see that the running time of the algorithm scales almost linearly as the number of updates(both x-axis and y-axis use logarithmic scale). This is to be expected, since the algorithm spends constant time processing each update on FIFO update sequences.
- For the parallelism, I varied the data with the parallelism ranging from 1 to 12. From Figure 5, we see that the running time decreases as the parallelism increase. However, the decreasing is not as expected. This is because the actual number of workers is not same to the number of parallelism I set. From Figure 7, and Figure 8, we see that the actual parallelism for preset parallelism = 4, 8, 12 is 3, 4, 4. So the running time of parallelism = 8 and 12 are very close in Figure 5 and the gap of running time when parallelism = 4 and 8 is small.
- The inconsistency in parallelism is mainly due to the filtering in the processing phase. I filtered the nation by name equal to ”Algeria” or ”Brazil”. In the aggregation phase to gather the final result, there are only four possible combination of (supp_nation, cust_nation) tuples by enumeration. Therefore the parallelism can only reach the maximum value of 4. After removing the filter, we can observe more workers from Figure 9.

supp_nation	cust_nation	l_year	revenue
ALGERIA	BRAZIL	1995	6642433.6288
ALGERIA	BRAZIL	1996	5974885.9088
BRAZIL	ALGERIA	1995	6628129.9886
BRAZIL	ALGERIA	1996	5945800.1644

4 rows in set (36.31 sec)

Fig. 3 The query results when scale = 1 from MySQL.

```
(ALGERIA,BRAZIL,1996,5955928.992800002)
(ALGERIA,BRAZIL,1996,5974885.908800002)
(BRAZIL,ALGERIA,1996,5677723.700799999)
(BRAZIL,ALGERIA,1996,5711140.110399999)
(BRAZIL,ALGERIA,1995,6452189.271399998)
(BRAZIL,ALGERIA,1996,5772950.701399999)
(BRAZIL,ALGERIA,1995,6541104.511399998)
(BRAZIL,ALGERIA,1996,5833108.986199998)
(BRAZIL,ALGERIA,1996,5846377.076199998)
(ALGERIA,BRAZIL,1995,6460571.505699998)
(ALGERIA,BRAZIL,1995,6498842.686899998)
(ALGERIA,BRAZIL,1995,6534372.886899998)
(BRAZIL,ALGERIA,1995,6587153.041399999)
(BRAZIL,ALGERIA,1996,5848197.236199998)
(ALGERIA,BRAZIL,1995,6564805.069299999)
(BRAZIL,ALGERIA,1996,5857694.810199998)
(ALGERIA,BRAZIL,1995,6622188.821799999)
(BRAZIL,ALGERIA,1996,5862431.672199998)
(BRAZIL,ALGERIA,1996,5886756.080199998)
(ALGERIA,BRAZIL,1995,6642433.628799999)
(BRAZIL,ALGERIA,1996,5919652.533999998)
(BRAZIL,ALGERIA,1995,6628129.988599999)
(BRAZIL,ALGERIA,1996,5945800.164399998)
```

Fig. 4 The query results when scale = 1 from algorithm.

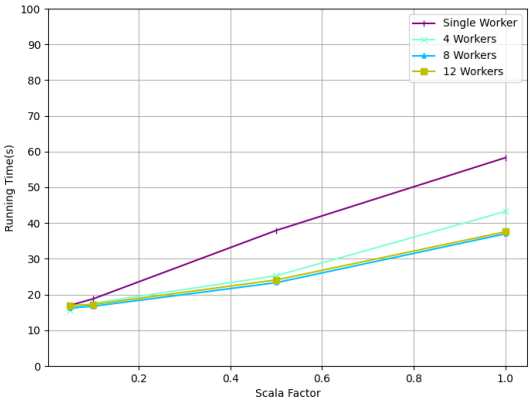

Fig. 5 Processing time on TPC-H Query 7 with different Parallelism and Scale Factor

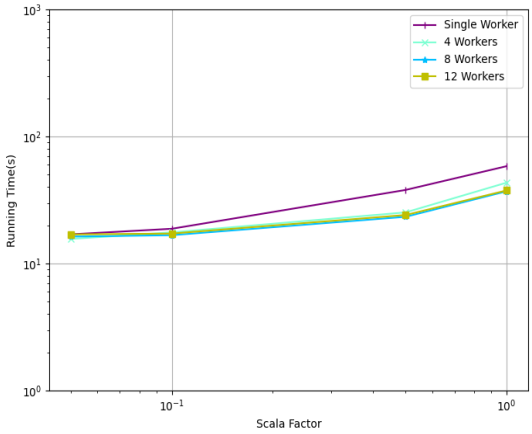

Fig. 6 Processing time on TPC-H Query 7 with different Parallelism and Scale Factor(loglog)

V. Conclusion

This project focuses on the implementation of Cquirrel, a system for continuous query processing built on top of Flink. The experiment prove the accuracy and efficiency of the algorithm on query results from TPC-H benchmarks with different size of parallelism and scalability.

The experimental results show that the running time of the algorithm scales linearly with the number of updates, and increasing parallelism can decrease the running time, but we need to further check the actual parallelism in implementation.

The running time in my implementation is still not fast. Potential avenues for future implementation could include the incorporation of Flame Graph and Web UI to analyze the internal of the algorithm and boost the implementation to achieve a lower running time.

Acknowledgment

I would like to express my sincere gratitude to Prof Yi Ke for his invaluable guidance, profound patience, and constant encouragement throughout the course of this research work. Help from Liu Yaxuan is also greatly appreciated for clearing my doubts.

References

[1] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. 2007. On acyclic conjunctive queries and constant delay enumeration. In International Workshop on Computer Science Logic. Springer, 208–222.

[2] Qichen Wang, Chaoqi Zhang, Danish Alsayed, Ke Yi, Bin Wu, Feifei Li and Chaoqun Zhan. Cquirrel: Continuous Query Processing over Acyclic Relational Schemas. PVLDB, 14(12): 2667 - 2670, 2021. doi:10.14778/3476311.3476315.

```
1> (ALGERIA,BRAZIL,1996,5897417.948799999)
1> (ALGERIA,BRAZIL,1996,5929175.828799999)
3> (BRAZIL,ALGERIA,1996,5794236.653499998)
3> (BRAZIL,ALGERIA,1996,5829161.079399998)
3> (BRAZIL,ALGERIA,1996,5867954.379399998)
3> (ALGERIA,BRAZIL,1995,6454197.823300003)
3> (ALGERIA,BRAZIL,1995,6500742.316100003)
3> (BRAZIL,ALGERIA,1996,5900059.059399998)
1> (ALGERIA,BRAZIL,1996,5974885.908799999)
4> (BRAZIL,ALGERIA,1995,6561782.456)
3> (BRAZIL,ALGERIA,1996,5918796.609399998)
3> (ALGERIA,BRAZIL,1995,6523658.120600003)
3> (BRAZIL,ALGERIA,1996,5926604.001399998)
3> (BRAZIL,ALGERIA,1996,5945800.164399997)
3> (ALGERIA,BRAZIL,1995,6578865.020600003)
4> (BRAZIL,ALGERIA,1995,6584604.971)
3> (ALGERIA,BRAZIL,1995,6593077.751600003)
4> (BRAZIL,ALGERIA,1995,6628129.9886)
3> (ALGERIA,BRAZIL,1995,6642433.628800003)
```

Fig. 7 The query results when parallelism=8 from algorithm.

```
7> (BRAZIL,ALGERIA,1995,6561782.455999996)
5> (ALGERIA,BRAZIL,1995,6360819.2465)
6> (BRAZIL,ALGERIA,1996,5799795.7314)
6> (BRAZIL,ALGERIA,1996,5867954.3794)
6> (BRAZIL,ALGERIA,1996,5900059.0594)
5> (ALGERIA,BRAZIL,1995,6412413.1121000005)
6> (BRAZIL,ALGERIA,1996,5918796.6093999995)
1> (ALGERIA,BRAZIL,1996,6048108.930199998)
6> (BRAZIL,ALGERIA,1996,5937992.772399999)
5> (ALGERIA,BRAZIL,1995,6453301.1321)
5> (ALGERIA,BRAZIL,1995,6469349.7401)
5> (ALGERIA,BRAZIL,1995,6500742.3161)
5> (ALGERIA,BRAZIL,1995,6555949.216100001)
7> (BRAZIL,ALGERIA,1995,6605307.4735999955)
6> (BRAZIL,ALGERIA,1996,5945800.164399999)
5> (ALGERIA,BRAZIL,1995,6605305.093300001)
7> (BRAZIL,ALGERIA,1995,6628129.988599995)
5> (ALGERIA,BRAZIL,1995,6619517.8243)
5> (ALGERIA,BRAZIL,1995,6642433.6288)
```

Fig. 8 The query results when parallelism=8 from algorithm.

```
5> (FRANCE,BRAZIL,1996,4383974.960899999)
7> (SAUDI ARABIA,ALGERIA,1996,3326353.7424000003)
8> (INDONESIA,INDONESIA,1995,4171964.6302000005)
8> (ARGENTINA,FRANCE,1996,4538668.263400003)
8> (ALGERIA,ETHIOPIA,1996,4238449.8714)
6> (ARGENTINA,KENYA,1996,3553697.5054)
2> (GERMANY,SAUDI ARABIA,1995,5268419.354899998)
3> (ALGERIA,IRAN,1995,4038872.8285)
3> (EGYPT,FRANCE,1996,4089660.7288999995)
2> (CANADA,ARGENTINA,1995,4540200.6275)
6> (BRAZIL,ALGERIA,1996,3877253.3412999976)
8> (INDONESIA,CANADA,1996,4177192.9115000013)
7> (ALGERIA,FRANCE,1995,4815401.310700001)
7> (SAUDI ARABIA,SAUDI ARABIA,1995,4068391.2460999996)
5> (ALGERIA,ALGERIA,1996,5054499.443100001)
4> (INDONESIA,ALGERIA,1996,4451558.6837)
4> (INDONESIA,ETHIOPIA,1996,3080071.8465)
4> (ARGENTINA,IRAN,1996,3910962.347600002)
4> (FRANCE,CANADA,1996,2609002.357500001)
4> (INDONESIA,ETHIOPIA,1996,3143598.9101)
1> (EGYPT,INDONESIA,1995,4823183.276800003)
```

Fig. 9 The query results when parallelism=8 and filter removed.

[3] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and Batch Processing in a Single Engine. IEEE Data Engineering Bulletin 38, 4 (2015), 28–38.

[4] SUDARSHAN AbrahamSilberschatzHenryF KorthS. Database System Concepts[J]. CERN European Organization for Nuclear Research - Zenodo,CERN European Organization for Nuclear Research - Zenodo, 2020.

[5] WANG Q, YI K. Maintaining Acyclic Foreign-Key Joins under Updates[C/OL]//Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. 2020. http://dx.doi.org/10.1145/3318464.3380586. DOI:10.1145/3318464.3380586.