SOPJ: A Scalable Online Provenance Join for Data Integration

Song Zhu, Giuseppe Fiameni, Giovanni Simonini and Sonia Bergamaschi
Department of Engineering Enzo Ferrari
Università di Modena e Reggio Emilia
Italy
Email: firstname.lastname@unimore.it

***Abstract*—Data integration is a technique used to combine different sources of data together to provide an unified view among them.**

MOMIS[1] is an open-source data integration framework developed by the DBGroup[1]. The goal of our work is to make MOMIS be able to scale-out as the input data sources increase without introducing noticeable performance penalty. In particular, we present a full outer join method capable to efficiently integrate multiple sources at the same time by using data streams and provenance information. To evaluate the scalability of this innovative approach, we developed a join engine employing a distributed data processing framework. Our solution is able to process input data sources in the form of continuous stream, execute the join operation on-the-fly and produce outputs as soon as they are generated. In this way, the join can return partial results before the input streams have been completely received or processed optimizing the entire execution. Encouraging results of adopting the proposed approach on real datasets closes the paper.

***Keywords*—Stream processing; full outer join; distributed process**

I. Introduction

With the advent of new technology, instruments, computing systems and sensors, data is generated at an incredible speed and stored in various forms. In order to create an unified view among collected data sets and be able to extract new insights, a new approach to integrate the various data sources together needs to be developed. MOMIS is an open-source Data Integration framework that aims at addressing this integration challenge but which requires further improvements to scale-out as data sources increase too rapidly. In this paper we present a new scalable join engine that using data streams is able to elaborate a huge amount of data simultaneously without decreasing execution performance. Many frameworks, such as `Spark Stream`[2] and `Apache Storm`[3], already offer similar approaches but none of them can be considered complete or exhaustive enough to this purpose.

A. Background

Performing data aggregation by combining different data sources together is a commonly used operation. In relational DBMS, this operation is called join. Combining the records of two or more data sources, namely tables, it generates output value bags by matching tuples on the basis of pivotal attributes, i.e. keys. In an inner join, if a tuple has no correspondences on the other sources, it will not end up into the final result. Conversely, the outer join preserves all the attributes despite the existence of any crossmatch[2].

Definition 1:

Cartesian Product: Given 2 relationships with schemes R(X) and S(Y), the result of Cartesian Product is a relation with schema R.X U R.Y and set of tuples:

$$r \times s = \{t | t = t_R t_S : t_R \in r \wedge t_S \in s\} \tag{1}$$

Definition 2:

Theta-Join: Given 2 relationships with schemes R(X) and S(Y) and a predicate P built by a boolean expression of join condition between R and S:

$$r \bowtie_P s = \sigma_P(r \times s) \tag{2}$$

Equi-Join are special join where predicates are only equality predicates.

Definition 3:

Outer join: Given two relations R(X) and S(Y). Define j to be the inner join with predicate P.

$$j = r \bowtie_P s \tag{3}$$

Define R' and S' as follows:

$$r' = r - \pi_X(j) \tag{4}$$

$$s' = s - \pi_Y(j) \tag{5}$$

R' and S' are unmatched tuples or dangling tuples of R and S, respectively, with respect to the join. The full outer join with predicate P is defined as:

$$r =\!\bowtie\!=_P s = r \bowtie_P sU(r' \times nY)U(nX \times s') \tag{6}$$

Where nX and nY are null values for attributes X and Y respectively.

[1]http://www.dbgroup.unimore.it/
[2]http://spark.apache.org/streaming
[3]http://storm.apache.org

Authorized licensed use limited to: UNIVERSITA MODENA. Downloaded on April 12,2022 at 09:57:13 UTC from IEEE Xplore. Restrictions apply.

B. Data Integration Join

While integrating different data sources, the full-outer-join is more appropriate than the inner-join to merge data as it preserves all partial results. Therefore we are focused to improve the scalability and efficiency of a full outer equi-join.

To reduce the number of stages during an outer join operation, our approach does not use a binary operator. It is able to accept records originating from many data sources and join them within one single stage. To this purpose, we use the provenance information to combine records of different sources.

C. Our Contribution

Within shared-nothing systems, the most common computing platforms for distributed computing[3], the join operation hardly scales due to the high number of network communications which are needed to aggregate the keys of the different sources[4]. Hence, the bandwidth of such systems is typical bottleneck for inter-node communications.

As a consequence, minimizing the volume and the frequency of data exchanges across the network becomes one of the most important aspects to consider while executing distributed join operations. Afrati F. et al[4] underlined the importance of communication cost reduction in a Map-Reduce context, and the same consideration can be done also in distributed stream process.

In this paper, we present a novel approach based on data streams. To the best of our knowledge, this approach is the first attempt to provide a full outer join operator over multiple sources based on the minimization of data transferred over the network. We choose a full outer multi-way join, since it is very suitable for data integration and ETL applications. Performance tests have been executed using the `Apache Ignite`[5], an open source distributed in-memory platform.

The rest of this paper is organized as follows: related works are presented in section II. We define the problem and present our solution in section III. Results of the empirical evaluation of our approach are presented IV. Finally, Section V concludes the paper.

II. Related work

The work presented in this paper reports about the implementation of a scalable join operator to solve the problem of massive data integration. Our solution is able to consume streams of data, thus optimizing the network traffic, and produce results as long as data sets are joint. In literature The join operation is widely studied.

In the distributed context, the main join paradigms/algorithms are:

- Map Reduce Joins;
- Stream Joins;
- Online Joins.

[4]The keys are typically spread among data sources.
[5]https://apacheignite.readme.io/docs

Map Reduce Joins: Map Reduce is one of most popular paradigms for distributed computation. Thus, there are a lot of work to improve the join performance in this framework [5], [4], [6], [7], [8]. However, map reduce join algorithms have a blocking behavior, since the Map Reduce paradigm uses a barrier to synchronize consecutive Map and Reduce steps affecting the execution performance. Each Map task needs to complete before the successive Reduce can start. As a consequence, data sets of a big source must be entirely loaded into the system before any join operation can start.

Stream Joins: The stream join merges data in the stream paradigm, and the result of join could be used before it is completed, there are many works and implementations on stream joins [9], [10], [11], [12]. This join paradigm is useful to merge sensor data, where the input data is continuous. However, all previews cited works on stream join are windows based, to support continues streams of data. This means, it has the limitation of arrival time. Whereby, if a tuple arrives out of time it will be lost. To employ windowed stream join, we have to sort the input data sources, and the arrival time is critical. Furthermore, the sort operation for a big data source is expensive. Moreover no data loss can happen while integrating data.

A remarkable contribution to stream join algorithms is that proposed by Lin Q, Ooi B C, Wang Z, et al. [13]. They propose join-bi-clique Model based on a bipartite graph. This algorithm allows historical and window-based Stream Join with any predicate, however with respect to our requirements, it allows join only between two relations, namely sources.

Online Joins: For our purpose we require a non-blocking and no data loss join algorithms. An implementation of distributed online join is studied by Elseidy. In [14], authors proposed an scalable and adaptive online join. However, Elseidy's algorithm does not support multiple sources neither full outer join.

III. SOPJ approach

A. Problem definition

Our first goal is to develop a Distributed Streaming Full Outer Join able to handle a large number of different sources whose data volume can be huge.

The proposed solution consists of a scalable join engine providing the following characteristics:

- **Full outer join**: In the data integration context, all tuples of data are required, including the no matching ones.
- **Distributed**: In the last decades, the distributed computing paradigm is the most employed method to meet scalability problems. The join operation is hard to scale in this paradigm, however, we believe that the distributed paradigm is the only solution to process the large amount of data available nowadays.
- **Streaming**: Our approach offers reactive response also at large-scale environment, this is due to our on the fly join process, that processes data when it arrives at a join node. In addition, stream process has a great advantage in chain processes [15].

Authorized licensed use limited to: UNIVERSITA MODENA. Downloaded on April 12,2022 at 09:57:13 UTC from IEEE Xplore. Restrictions apply.

- **Multi-way**:It may also happen that many data sources share the same keys, or can be mapped into the same keys; in this case, a multi-way join with the same key reduces join stages compared to binary join. This characteristic can be interesting for data integration of sensor data, where there are many different datasources, sensors, and it is necessary merge them for timestamp or geolocation attribute.

1) Distributed processing: The main strategy to improve an application performance is to make it use multiple nodes provided its internal tasks can be distributed.

In this context, many frameworks have been developed to simplify the development of such distributed applications [16].

As presented before II, the main join paradigms/algorithms are:

- Map Reduce Joins;
- Online Joins;
- Stream Joins.

For our purpose we adopt online join paradigm since we require a non-blocking join. Our join engine receives stream inputs from data sources and merges arrival tuples on-the-fly and, when the result of a tuple is complete, it can be used immediately in the next operation.

2) Stream processing: The *stream-based applications* is a class of data processing workloads where streams of tuple are pushed to the system and continuously quiered [17].

For multiple steps processes, the stream-based paradigm can bring a fundamental advantage with respect to the batch processing one where a succesive step starts only when all previous ones have been completed. It becomes even more important and necessary when input streams are continuous and never ends [18].

For example, in a two stages join process, as shown in figure 1, there are three relations *product(cod_product, name, description, price)*, *order_product(cod_order, cod_product, quantity)* and *order(cod_order, customer)*, and we have to join *product* with *order_product* on *cod_product* and to join *order* and *order_product* on *cod_order*. We define (*product* join *order_product*) join *order*. This means we have two stages of join. First stage, join between *product* and *order_product*, then the result join *order*. In batch process paradigm, we have to finish the first stage before starting the second stage of join. In a stream process, the second join process can starting before first stage ends [19].

SOPJ is not a window-based join. Thus the arrival time is not critical for our algorithm.

B. SOPJ concept

We implement SOPJ as a variant of hash join taking inspiration form the Doubled Pipelined Hash Join[20].

Hash Join: In an conventional hash join the smaller relation is stored into hash table in the build phase, and the other relation is scanned in the probe phase to get result.

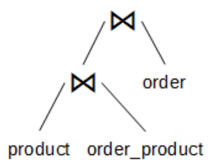


Fig. 1. Multiple join

Doubled Pipelined Hash Join: The Doubled Pipelined Hash Join [20] is still a join with two relations, and both relations send tuple to the join operator as fast as possible. The join operator probes each record into the hash table of the other relation to produce results, and stores the record into the hash table of the current relation. This join algorithm produces output records almost immediately and partially masks datasources network bottleneck consuming more memory as the Doubled Pipelined Join stores both relations.

SOPJ: Our join has to be able to accept more then two relations. In addition SOPJ is a full outer join. Our idea is to store records from all datasources in a single hash table, the join attribute is used as key, in this way, all records with the same values in the join attribute are grouped in an single entry of the hash table. Input message for each input record from datasources has to be composed of three parts:

- join key - the value of join attribute.
- datasource provenance id, i.e. dpID - each datasource has an identifier, it is used to identify datasource during join.
- data - values of the input record.

In the hash table, records from all datasources are stored with the join key as key of hash table as explained above, the value of hash table is a collection of dpID and data, the join engine combines data by using the provenance key of each record in the value. Each input record or message adds or updates a row of the hash map. When a value of the hash table is complete, then there is at least one record for each relation in the join, output records are generated immediately. At the end of record sending process, when all datasources have sent all records, SOPJ evaluates all rows of the hash table with incomplete values, row with a missing join key at least in one datasource, these are dangling tuples of full outer join. SOPJ produces output also for these dangling tuples, reporting null values for the attributes of missing datasources.

Our idea is to use a distributed hash table. In this way, we have the advantage of scalability and fault tolerance with the replication of hash table in more nodes to avoid loss of data in cases of node failure.

In principle the SOPJ algorithm is affected by the same memory issued of the Doubled Pipelined Hash Join as the hash table which needs to be stored can be huge. However since an hash table can be easily distributed across multiple nodes and access operations executed in parallel, the capability of our join implementation to process data depends on the amount

Authorized licensed use limited to: UNIVERSITA MODENA. Downloaded on April 12,2022 at 09:57:13 UTC from IEEE Xplore. Restrictions apply.

TABLE I
OUTPUT GENERATION EXAMPLE: NO OUTPUT GENERATED

”d1”, (”v11”, ”v12”)
”d3”, (”v31”, ”v34”, ”v33”)
”d1”, (”v15”, ”v14”)

TABLE II
OUTPUT GENERATION EXAMPLE: VALUE COMPLETED

”d1”, (”v11”, ”v12”)
”d3”, (”v31”, ”v34”, ”v33”)
”d1”, (”v15”, ”v14”)
”d2”, (”v21”)

of resources available. Performance can scale-out as linearly as the number of nodes.

C. Implementation

The implementation of the SOPJ algorithm is based on the Apache Ignite framework. It was chosen for its capability to handle data streams and cache distributed data. At the end of our evaluation these features resulted fundamental for the implementation of the algorithm.

A distributed system usually processes the join operation in two steps:

1) **Data shipment**: the input data is distributed across all join computational nodes;
2) **Local join step**: Local join step: each node performs a stand-alone join on its local portion of data.

The first step corresponds to the receiving of input record from datasources, and the local join step is the generation of output records. In SOPJ the two steps of the join operation, data shipment and local join, are performed almost in parallel reducing the computation time. Basically SOPJ does not need to wait until the whole data shipment takes before starting to perform local joins. However the algorithm produces results for a join key as soon as the join key has at least one value for each datasource. In addition, if a new value arrives for a join key that already generated output records, new outputs are generated for the new value. Let us consider an output generation example: In this example we consider a single join key ”jk5”, and we join three datasources with provenance id ”d1”, ”d2” and ”d3”. In the hash map, for the datasource ”d1” there are two records with the value of ”jk5” and one record for datasource ”d3” as shown in table I

When a record sent from datasource d2 with join key ”jk5” is received, as shown in table II, output records are generated on the fly, as shown in table III. The output record is a cartesian product of among the values in table II grouped by their provenance id as fellowing equation.

$$Output = d1 \times d2 \times d3 \tag{7}$$

We can do cartesian product since their have the same join key.

If an additional record with join key ”jk5” is received for example from datasource d3, as shown in table IV, additional output are generated, as shown in table V. The new output is generated as fellowing equation.

TABLE III
OUTPUT GENERATION EXAMPLE: FIRST OUTPUTS

jk5	v11	v12	v21	v31	v34	v33
jk5	v15	v14	v21	v31	v34	v33

TABLE IV
OUTPUT GENERATION EXAMPLE: AN NEW INPUT FROM DATASOURCE D3 FOR JOIN KEY JK5

”d1”, (”v11”, ”v12”)
”d3”, (”v31”, ”v34”, ”v33”)
”d1”, (”v15”, ”v14”)
”d2”, (”v21”)
”d3”, (”v35”, ”v36”, ”v38”)

$$NewOutput = d1 \times d2 \times NewRecord\_d3 \tag{8}$$

Where NewRecord_d3 is the only the last received record of d3

IV. TESTS

In this section, we illustrate a performance evaluation of our join algorithm. To evaluate SOPJ we used `Apache spark`’s DataFrame and `Spark SQL`[6]. We know that the two contexts are completely different, one is a batch process and the other is a stream process. However, `Spark` is the most known and appreciated distributed framework by the open source community. Thus, we believe, it is appropriate to use `Spark` as Golden Standard for distributed applications comparison.

A. Hardware

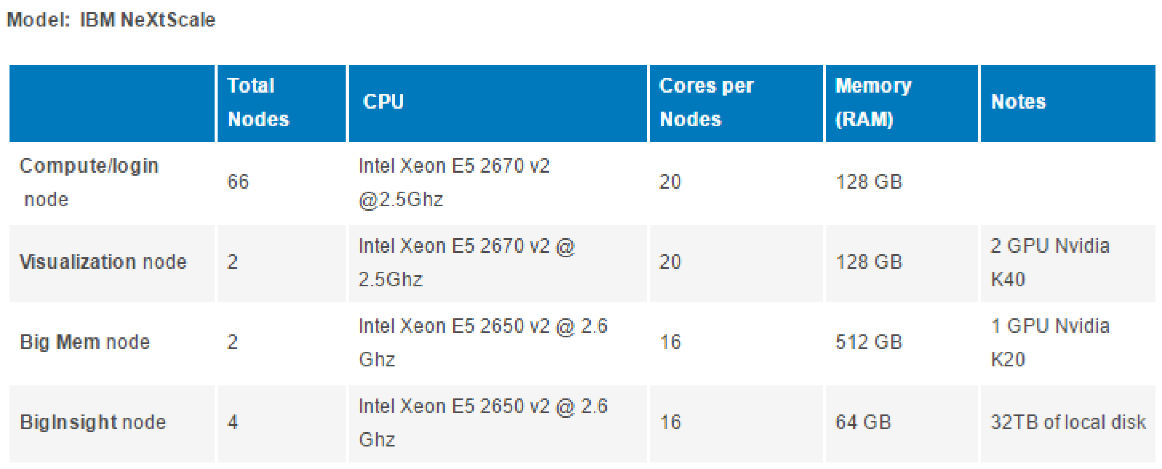
The performance evaluation was done by using **Pico**[7], a computing system made available by **Cineca**[8]. PICO is made of an Intel NeXtScale server, designed to optimize density and performance, driving a large data repository shared among all the HPC systems in CINECA. The storage area is composed of high throughput disks (based on GSS technology) for a total amount of about 4 PB, connected with a large capacity tape library for a total actual amount of 12 PByte (expandable to 16 PByte). The storage area is accessible from all HPC systems in CINECA (under the $DATA name) an is organized on the basis of projects. Each active project on any HPC system has a corresponding entry on $DATA. The storage area is a ”multi-level” memory. Depending on the defined policy, data migrate automatically on the tape, in a transparent way with respect to the user.

[6]http://spark.apache.org/sql/
[7]http://www.hpc.cineca.it/hardware/pico
[8]http://www.cineca.it/en

TABLE V
OUTPUT GENERATION EXAMPLE: ADDITIONAL OUTPUTS FOR NEW INPUT

jk5	v11	v12	v21	v35	v36	v38
jk5	v15	v14	v21	v35	v36	v38

Authorized licensed use limited to: UNIVERSITA MODENA. Downloaded on April 12,2022 at 09:57:13 UTC from IEEE Xplore. Restrictions apply.


Model: IBM NeXtScale

	Total Nodes	CPU	Cores per Nodes	Memory (RAM)	Notes
Compute/login node	66	Intel Xeon E5 2670 v2 @2.5Ghz	20	128 GB	
Visualization node	2	Intel Xeon E5 2670 v2 @ 2.5Ghz	20	128 GB	2 GPU Nvidia K40
Big Mem node	2	Intel Xeon E5 2650 v2 @ 2.6 Ghz	16	512 GB	1 GPU Nvidia K20
BigInsight node	4	Intel Xeon E5 2650 v2 @ 2.6 Ghz	16	64 GB	32TB of local disk

Fig. 2. Cineca's Pico nodes

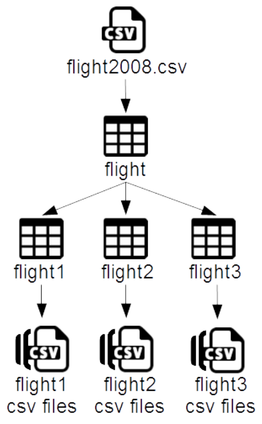


Fig. 3. Input partitions

Pico has 66 compute nodes, as show in Figure 2, and we employ them to create our computation cluster with `Ignite`. In this way, we can easily test SOPJ with different number of computation node.

B. Input data

To test the join algorithm employed the flights[9] dataset of year 2008 with over 7 million of records, we split its columns into 3 data sources, in this way we obtained a 3 way join.

The input data are stored to disc as csv files. The initial dataset is split to 3 datasets (*flight1*, *flight2* and *flight3*) with a common join id and the partitioned columns, as to consider them 3 data sources; then we random split each data source to csv files, each csv file contains 10000 rows, as shown in figure 3.

We implement a distributed job to read each file and send them as streaming message to SOPJ. We implement SOPJ to receive multiple stream input from each datasource, in this way also each datasource can be distributed on more nodes and sends its data on multiple streams.

Tests are made by using the strong scaling paradigm to measure the speedup of the join engine on distributed environment. We used the same configuration, the only difference is the number of computation nodes.

[9]http://stat-computing.org/dataexpo/2009/the-data.html

C. Spark SQL

To compare SOPJ with `Spark SQL`, we created a `Scala` script with `Spark SQL` API.

The script loads *flight1*, *flight2* and *flight3* datasets into 3 `Spark`'s DataFrames. Then, the script queries on Dataframes are performed by using `Spark SQL` full outer join.

```
select
coalesce(f1.id,f2.id,f3.id) as newid,
*
from flight_1 f1
full outer join flight_2 f2
  on f1.id = f2.id
full outer join flight_3 f3
  on coalesce(f1.id, f2.id) = f3.id
```

D. Results

In this experiment, the focus is on the speedup of the algorithm with different number of nodes in the network. Results are shown in table VI and figure 4.

The `Spark`'s chart demonstrates by trial, the high difficulty to scale the join operation on distributed environments. The trend of `Spark` is practically constant with increasing number of nodes. The query plans of `Spark` full outer join executions are always the same with a different number of nodes. Looking at the query plan, `Spark` performs hash partitioning for the data shipment phase, and sort merge join on nodes for local join step.

Authorized licensed use limited to: UNIVERSITA MODENA. Downloaded on April 12,2022 at 09:57:13 UTC from IEEE Xplore. Restrictions apply.

TABLE VI
JOIN PERFORMANCE

Number of nodes	SOPJ [ms]	Spark SQL [ms]
3	116337	74541
5	83652	62485
8	58520	64256
10	54377	64209
12	50643	58141
15	50527	55821

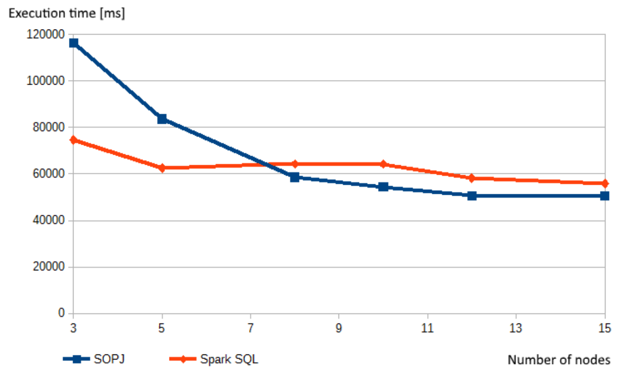


Fig. 4. Join performance

The performance of SOPJ is similar to Spark join performance. However, SOPJ performs worse on few nodes, and scale better with higher number of nodes. Differences of performance among 3, 5 and 8 nodes are significant, while differences of performance with more than 8 nodes are small.

To investigate deeper on performances of SOPJ, we measured the resource usage of join execution. The left part of figure 5 shows the average of CPU usage, and the total TCP traffic of all nodes when 5 nodes are used. The right part of figure 5 shows the same measures when 10 nodes are used. The trends of the same type of charts are similar. All 4 charts are subdivided in 2 parts, the first part executions the system is busy, the CPU usage is almost 100% and also the networks.

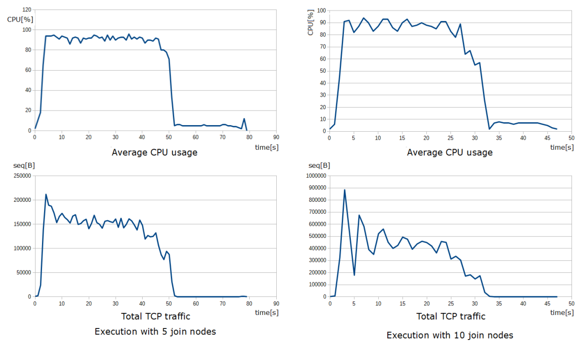


Fig. 5. Resources consumption chart

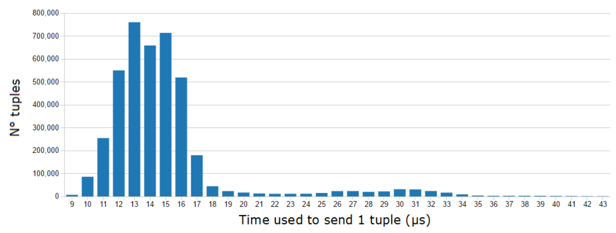


Fig. 6. tuple sending statics

When networks activities stops also the CPU activity drop down. The main networks activities are the data shipment, and during the data shipment phase, the local join step is performed at the same time. If we suppose the local join step consumes same CPU resources in both part of executions. The main activities of CPU in the first part of execution is due to data preparation for data shipment. The local join consumes few CPU resources, since it likes other hash joins, it is a memory intensive operation [21].

The networks activities with 10 nodes execution is more intense and short then with 5 nodes, this is due to more resources (CPU and bandwidth) available for execution, and increase the scalability of join engine.

Therefore, data shipment is the key aspect to consider for the join optimization, hence another issue to be evaluate is the speed to send tuples. In figure 6 the sending statistics of tuples on 1 node by the join execution with 10 nodes is shown. From this chart, we observed the time to send 1 tuple around 14μS, and we can calculate the time for 1 node to send its tuples with the following operation.

$$ST = MeanTime * Ntuples * Nsources/Nnodes \quad (9)$$

For the 10 nodes execution, the time necessary to send all tuples is around 29 seconds, and it coincides with the networks activities shown in bottom right part of figure 5.

V. CONCLUSION AND FUTURE WORK

This paper presents a novel approach (SOPJ) to execute a stream-based multi-way full-outer join over distributed data sources. The measured performance of our implementation is comparable with a batch join using `Apache Spark` provided the local capacity of the node memory is large enough to store the entire data set. Despite some inefficiencies while using small data sets, our approach looks promising and thus deserves further investigations. As next step, we will implement the SOPJ algorithm by using the `Spark Stream API` and compare the performance against `Apache Ignite`. The comparison is worth the implementation effort as the philosophy of manipulating data streams standing behind the two frameworks is completely different.

During our evaluation, we observed that one of the key aspects of the join process optimization is the the data shipment. Reducing the time to transfer the records across the

Authorized licensed use limited to: UNIVERSITA MODENA. Downloaded on April 12,2022 at 09:57:13 UTC from IEEE Xplore. Restrictions apply.

network gives a significant increase to the overall performance. Moreover, our algorithm performs well when the bandwidth available on the system represents a bottleneck as data is processed on the fly optimizing the use of resources.

The data skew is an important issue in distributed join, however it is hard to manage in stream context, since the join key distribution is unknown a priori. In [14], an adaptive algorithm is proposed, nevertheless, this solution needs additional movement of data through the network and an accurate knowledge about resources of the system. We will evaluate in future the trade off about an adaptive algorithm for SOPJ.

The applications for our join engine are numerous. The main context of application is data integration system, like MOMIS system [1], as it is necessary to return tuples from all integrated data sources, including also all dangling tuples. Moreover, when the number of data sources to be integrated increases, the multi-way join feature of our algorithm outperforms the binary join approach of other frameworks such as `Spark`. Another application we are investigating is in the sensor monitor context, within represents (along with data integration) the main focus of our future work.

VI. Acknowledgments

We would also like to show our gratitude to Cineca for supporting us for tests and making available Pico for our test environment.

References

[1] D. Beneventano and S. Bergamaschi, "The momis methodology for integrating heterogeneous data sources," in *Building the Information Society*. Springer, 2004, pp. 19–24.

[2] E. F. Codd, "Extending the database relational model to capture more meaning," *ACM Transactions on Database Systems (TODS)*, vol. 4, pp. 397–434, 1979.

[3] M. Hogan, "Shared-disk vs. shared nothing, comparing architectures for clustered databases," *URL http://www. scaledb. com/pdfs/WP_SDvSN. pdf.(Zitiert auf Seite 33)*, 2012.

[4] F. N. Afrati and J. D. Ullman, "Optimizing joins in a map-reduce environment," 2010, pp. 99–110.

[5] H. C. Yang, A. Dasdan, R. L. Hsiao, and D. S. Parker, "Map-reduce-merge: simplified relational data processing on large clusters," in *ACM SIGMOD International Conference on Management of Data*, 2007, pp. 1029–1040.

[6] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian, "A comparison of join algorithms for log processing in mapreduce," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010, pp. 975–986.

[7] C. Zhang, J. Li, and L. Wu, "Optimizing theta-joins in a mapreduce environment," *International Journal of Database Theory & Application*, vol. 6, 2013.

[8] F. Afrati, N. Stasinopoulos, J. D. Ullman, and A. Vassilakopoulos, "Sharesskew: An algorithm to handle skew for joins in mapreduce," *Computer Science*, 2015.

[9] Y. Zhou, Y. Yan, F. Yu, and A. Zhou, *PMJoin: Optimizing Distributed Multi-way Stream Joins by Stream Partitioning*. Springer Berlin Heidelberg, 2006.

[10] B. S. Lee and T. M. Tran, "Distributed adaptive windowed stream join processing," *International Journal of Distributed Systems and Technologies*, vol. 2, no. 2, pp. 59–81, 2011.

[11] P. Roy, J. Teubner, and R. Gemulla, "Low-latency handshake join," *Proceedings of the VLDB Endowment*, vol. 7, no. 9, pp. 709–720, 2014.

[12] V. Gulisano, Y. Nikolakopoulos, M. Papatriantafilou, and P. Tsigas, "Scalejoin: A deterministic, disjoint-parallel and skew-resilient stream join," in *IEEE International Conference on Big Data*, 2015, pp. 144–153.

[13] Q. Lin, B. C. Ooi, Z. Wang, and C. Yu, "Scalable distributed stream join processing," pp. 811–825, 2015.

[14] M. Elseidy, A. Elguindy, A. Vitorovic, and C. Koch, "Scalable and adaptive online joins," *Proceedings of the VLDB Endowment*, vol. 7, no. 6, pp. 441–452, 2014.

[15] S. D. Viglas, J. F. Naughton, and J. Burger, "Maximizing the output rate of multi-way join queries over streaming information sources," in *Proceedings of the 29th international conference on Very large data bases-Volume 29*. VLDB Endowment, 2003, pp. 285–296.

[16] O.-C. Marcu, A. Costan, G. Antoniu, and M. S. Pérez, "Spark versus flink: Understanding performance in big data analytics frameworks," in *Cluster 2016-The IEEE 2016 International Conference on Cluster Computing*, 2016.

[17] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik, "High-availability algorithms for distributed stream processing," in *21st International Conference on Data Engineering (ICDE'05)*. IEEE, 2005, pp. 779–790.

[18] A. Riabov and Z. Liu, "Planning for stream processing systems," in *Proceedings of the National Conference on Artificial Intelligence*, vol. 20. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2005, p. 1205.

[19] B. Liu and E. A. Rundensteiner, "Revisiting pipelined parallelism in multi-join query processing," in *Proceedings of the 31st international conference on Very large data bases*. VLDB Endowment, 2005, pp. 829–840.

[20] Z. G. Ives, D. Florescu, M. Friedman, A. Levy, and D. S. Weld, "An adaptive query execution system for data integration," in *ACM SIGMOD Record*, vol. 28, no. 2. ACM, 1999, pp. 299–310.

[21] C. Balkesen, J. Teubner, G. Alonso, and M. T. Ozsu, "Main-memory hash joins on modern processor architectures," *IEEE Transactions on Knowledge & Data Engineering*, vol. 27, no. 7, pp. 1754–1766, 2015.

Authorized licensed use limited to: UNIVERSITA MODENA. Downloaded on April 12,2022 at 09:57:13 UTC from IEEE Xplore. Restrictions apply.