Scaling Out Continuous Multi-Way Theta-Joins

Manuel Hoffmann
TU Kaiserslautern
Kaiserslautern, Germany
mhoffmann@cs.uni-kl.de

Sebastian Michel
TU Kaiserslautern
Kaiserslautern, Germany
michel@cs.uni-kl.de

ABSTRACT

In this paper, we propose generic tuple routing schemes that allow the computation of distributed multi-way theta-joins over streaming data. We present an architecture which compiles query plans in form of logical operators into Apache Storm topologies and report on first results of evaluating TPC-H data using Amazon EC2 instances running these topologies.

CCS Concepts

•**Information systems** → **Join algorithms; Stream management;**

Keywords

Data Streams, Theta-Join, Apache Storm

1. INTRODUCTION

Processing data streams is a classical and ubiquitous problem. It ranges from monitoring enterprise-internal system access logs for ad placement or anomaly detection, to providing real-time analytics over social network streams. What many applications ultimately demand is relating information across multiple streams—for instance joining query and ad-click streams in Google, or relating blog posts and Twitter tweets for enriched, user-specific content delivery. In this paper, we propose generic tuple routing schemes that allow processing distributed multi-way theta-joins over streaming data. In contrast to equi-joins, theta-joins are provided with a generic predicate that tells whether or not tuples can be joined. The desired algorithms need to assure that all potential join partners eventually meet each other, once and only once, reflecting anomalies like delayed tuples, skewed distribution of stream loads, node failures, and dropped messages. This is very challenging for generic theta-joins and even more so for natively addressing multi-way joins over such generic predicates. On the other hand, the problem also opens various ways to optimize performance. In this paper, we give a first overview of the potential of sophisticated tuple routing schemes for processing multi-way theta-joins in a scale-out fashion.

BeyondMR'17 May 19, 2017, Chicago, IL, USA

ISBN 978-1-4503-5019-8/17/05…$15.00

DOI: http://dx.doi.org/10.1145/3070607.3070611

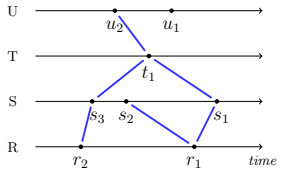


Figure 1: Tuples of multiple input streams arrive.

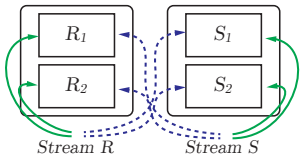


Figure 2: Routing of tuples in BiStream.

1.1 Problem Statement

We consider theta-joins where the join predicate is a conjunction of binary predicates. For brevity, let us introduce the computational model and problem by walking through an example. Consider a join between four relations R, S, T, and U, with $\theta = \theta_{R,S} \wedge \theta_{S,T} \wedge \theta_{T,U}$. Figure 1 depicts how tuples of multiple relations occur interleaved.

Tuples that fulfill the partial predicate, e.g., r_1 and s_2 satisfy $\theta_{R,S}$, are connected together. Such tuples *might* belong to the overall join result. However, since s_2 does not find a join partner in T (yet), they do not. On the other hand, r_1, s_1, t_1, and u_2 simultaneously satisfy all partial predicates, thus, should be joined and included to the result, likewise for r_2, s_3, t_1, and u_2.

Answering queries which not only relate two but many streams has been studied before [12, 8, 3], but with a focus on equi-joins. The restriction to equi-joins allows for simple yet extremely effective partitioning schemes, which are not applicable for the more general case of theta-joins.

In this paper, we present the architecture and general idea behind our approach, sketch optimization possibilities, how to compile routing schemes into Storm topologies, and report on the first results of an experimental evaluation over TPC-H data using Amazon EC2 instances running Apache Storm.

2. RELATED WORK

There is a vast amount of research work that is being dedicated to processing join queries, both in data streaming contexts as well as classical database systems. Considering scale-out architectures, there is work on processing theta-joins in the MapReduce framework [11], and recent work on doing so for data streams [6, 10]. The main idea of such works is to assign relations to nodes such that all possibly matching tuples of the join eventually coincide at the same machine. The simplest case is the so called join matrix [6], where a relation R gets replicated across n machines, and each tuple of relation S is forwarded to one of the n machines,

and vice versa. The BiStream approach [10] on the other hand, proposes tuple routing schemes that avoid replication of data stream tuples. Both works address traditional two-way joins, and briefly sketch to model multi-way joins by fully materializing intermediate results and treat it as input to another stage of the algorithm. The generalization to n-way joins, that is, joins involving n relations or data streams, brings additional challenges for assuring that all potential join partners will eventually meet. On the other hand, addressing multiple joins at once promises wider freedom in optimizing tuple routing across machines. Zhou et al. [12] propose PMJoin and optimizations for processing distributed multi-way stream joins, but are also limited to handling equi-joins. Joglekar and Ré [9] propose using information on the multiplicity of values to optimize multi-way joins, also limited to equi-joins, and not considering distributed computation (although some results are of generic nature). Specifically addressing window stream joins, Hammad et al. [7] present two algorithms for processing multi-way joins in a centralized setting, there is no consideration of how such algorithms could potentially be executed in a distribution fashion. The algorithms are, however, oblivious to the matching predicate, and, thus, not bound to simple equi-joins.

3. PRELIMINARIES

The task of joining multiple streamed relations differs from joining static relations, as in classical DBMSs, by the fact that there is no control about the order of the arriving tuples. While in a DBMS the query execution component can chose a join order of relations R, S and T, e.g., first join R and S, and then join the resulting relation $R \bowtie S$ with T without accessing R or S anymore, in a streaming scenario this is not feasible. Consider a fresh tuple r arriving at relation R *after* $R \bowtie S$ was computed. Then this tuple has to be probed against the entire part of S that was observed so far and if it matches elements of S, $R \bowtie S$ has to be updated.

Further, with passing time more and more tuples arrive in a data stream, eventually exceeding the available memory. This problem can be solved by considering windows over the datastream [5], such that the content of the window can be handled by the system.

3.1 The BiStream Model

To compute a θ-join between *two* relations R and S, the BiStream model of Lin et al. [10] suggests to store tuples from R in partitions $R_1, \ldots, R_i$ and tuples from S in partitions $S_1, \ldots, S_j$ as illustrated in Figure 2. If a tuple r of R arrives, it is sent to a randomly selected partition R_1 or R_2 (indicated by the green, solid arrows). At the same time, it is sent to all partitions of S (indicated by the blue, dashed arrows) where the partial joins $r \bowtie S_1$ and $r \bowtie S_2$ are computed.

With this partitioning and routing scheme in place, it is possible to dynamically scale the number of nodes storing a relation as only the routing has to be changed to consider the new node.

4. APPROACH

We propose a system architecture that is able to compute multi-way theta-joins, illustrated in Figure 3, for the case of three relations. The system consists of several components that can be individually parallelized and deployed on multiple computing nodes: **Data sources**, here R, S, and T, that constantly emit tuples into the system. The **dispatcher**, which is responsible for forwarding the tuples to the correct nodes, depending on the query to be answered. Several

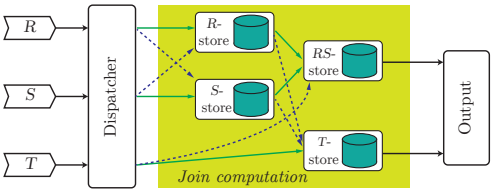
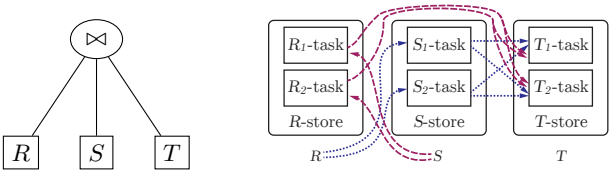


Figure 3: Architecture for theta-join computation handling continuous inputs.



(a) Notation (b) Routing

Figure 4: Non-materialized join operator over three streams.

components, we call "**Stores**" in the join-computation block, interconnected in such a way that tuples sent amongst them can be iteratively joined and eventually a result is produced. Finally, an **output component** that emits resulting tuples.

The task of the stores is twofold: First, they store tuples from the relation or intermediate join results, e.g., the R-store contains tuples from R. Second, they receive tuples from other relations, say $s \in S$, and produce the join result, here, $s \bowtie R$. We also say that s is "probed" against R. The green, solid arrows in the illustration indicate that tuples are sent along these connections, in order to be stored at the receiving store. The blue, dashed arrows indicate that tuples are sent in order to be probed against the receiving store.

We use the same basic idea to scale-out operators that also BiStream [10] uses: When it is necessary to utilize more computing nodes, e.g., due to overloaded nodes in terms of memory usage or CPU-time consumption, further partitions can be added to the stores.

In the example sketched in Figure 3, the intermediate result of $R \bowtie S$ is materialized in the RS-store. Consider now an alternative deployment, where the RS-store is missing. From a static viewpoint, this would save resources, as there are $|R \bowtie S|$ tuples less that have to be stored for the join computation. At the same time, if a tuple t of T arrives, it cannot find potential join partners at a single location. Instead it has to be routed to the S-store, where the partial join $t \bowtie S$ is computed, and the result is sent to the R-store in order to get the final result. This way, the overall number of communication actions is increased.

4.1 Operator Model

The two routing/storing schemes discussed in the previous paragraph represent two different ways of computing a join between three streams. We now generalize these options and introduce an operator model consisting of operators that can be combined into a tree to realize different schemes in a flexible and generic manner. The inner nodes of the tree have 2 to n children and are labeled either as **materializing** (drawn in boxes) or as **non-materializing** (drawn in ellipses). The root is always a non-materializing join, since the system does not reuse final result tuples. The leaf nodes are also drawn in boxes, as they are all materialized.

In order to understand the semantics of a join operator, consider the query $R \bowtie S \bowtie T$. Three cases need to be

handled in order to guarantee the correct join results: *(i)* A tuple of R arrives at the operator, *(ii)* a tuple of S, *(iii)* or a tuple of T. For each of these cases, a local join order is produced, e.g., $R \rightarrow S \rightarrow T$, $S \rightarrow R \rightarrow T$, and $T \rightarrow S \rightarrow R$. Such a local join order dictates the routing of each individual tuple: $R \rightarrow S \rightarrow T$ means, the tuple r is sent to the S-store first, there the partial result $r \bowtie S$ is computed, and all resulting tuples are then sent to the T-store in order to produce the final join result. The order of the joins should be selected in such a way, that the size of the intermediate results is minimized.

In our system, this operator is implemented in form of the routing between the inputs' stores as indicated in Figure 4b. There, the routing of incoming tuples from R is displayed with dotted, blue arrows, and for tuples from S with dashed, violet arrows. The routing for tuples from T is omitted in order to avoid more visual cluttering. The stores for R, S and T all are partitioned into two tasks, thus half of the tuples of R reside in partition R_1, the other half in partition R_2. A newly arriving tuple $s \in S$ is sent to both partitions, R_1 and R_2, where the partial results $s \bowtie R_1$ and $s \bowtie R_2$ are computed. These partial results again are sent to both, T_1 and T_2. After that the following partial results are on hand $s \bowtie R_1 \bowtie T_1$, $s \bowtie R_1 \bowtie T_2$, $s \bowtie R_2 \bowtie T_1$, and $s \bowtie R_2 \bowtie T_2$, which means, that $s \bowtie R \bowtie T$ is computed completely. The process is analogously for tuples arriving from R and S.

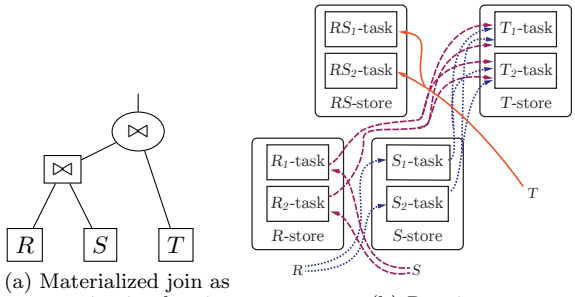
In order to estimate the communication volume produced by this operator, let $\sigma_i = \langle S_i, S_{\sigma_i(2)}, \ldots, S_{\sigma_i(n)} \rangle$ be the chosen local join order if a tuple arrives at S_i and let $N_{\sigma_i(j)}$ be the degree of parallelism for this store. For input stream S_i, $|S_i|$ tuples are sent to the $N_{\sigma_i(2)}$ tasks where the tuples of the second stream in order i are stored. The number of tuples that are generated by this join in total is $|S_i \bowtie S_{\sigma_i(2)}|$. For the next step, all these tuples are sent to the $N_{\sigma_i(3)}$ tasks of the third input where $|S_i \bowtie S_{\sigma_i(2)} \bowtie S_{\sigma_i(3)}|$ tuples are produced and so on. Since these messages occur for all n inputs, the total number of tuples sent is:

$$\sum_{i=1}^{n} \sum_{j=1}^{n-1} |S_{\sigma_i(1)} \bowtie S_{\sigma_i(2)} \bowtie \cdots \bowtie S_{\sigma_i(j)}| \cdot N_{\sigma_i(j+1)} \qquad (1)$$

The sum over the cost for the intermediate join results is known from textbook query optimization in database systems, where the cost of a join is composed of the cost for materializing all intermediate results (the inner sum). But in contrast, in a streaming environment, all input streams can be first and thus not only one but n orders have to be considered (the outer sum). Here, also the downside of parallelization is included explicitly, as the communication of the intermediate results is multiplied by the target's parallelism. Thus, effectively, communication cost grows linearly with the parallelism of the operators.

Operators can materialize their results. This is done by sending the resulting tuples of such an operator to an own store. Such a **materializing join operator** allows the construction of **deeper join trees**, where the inner nodes of the tree function as barriers for the necessary communication.

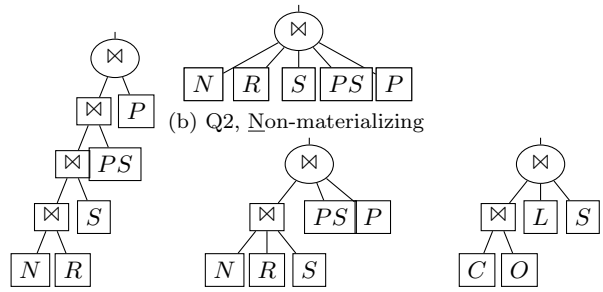
Consider the running example where $R \bowtie S$ is now joined using a materializing join, and its result is joined with T. The resulting operator tree is depicted in Figure 5a, where the box around the join operator above R and S indicates that the resulting tuples are materialized. The routing for tuples arriving at R and S is the same as before, as seen in Figure 5b, with the exception that each of the resulting tuples are also copied to one of the RS-partitions (not shown). When a tuple from T arrives, it does not have to travel first



(a) Materialized join as communication barrier

(b) Routing

Figure 5: Join over three streams with materialization of intermediate results.



(a) Q2, Leftdeep

(b) Q2, Non-materializing

(c) Q2, Mixed plan

(d) Q5, Mixed plan

Figure 6: Different possibilities of constructing query plans for Q2 and Q5.

to R and then to S, depending on the selectivities generating many unused intermediate tuples, but it only has to be sent to the two partitions of RS, where the final result can be computed.

4.2 From Query Plans to Storm Topologies

When a query plan is chosen, it is automatically translated into a Storm topology [1] which ultimately executes the plan. A Storm topology consists of sources (called spouts in Storm terminology), operators (called bolts), and named streams connecting them. For each bolt a "parallelism hint" can be set, which indicates how many instances of a spout or bolt should be deployed in parallel. The named streams have different grouping modes: all grouping, which sends a tuple to each instance of the receiving bolt, and shuffle grouping, which sends tuples to randomly selected instances. Each topology that we generate consists of a dispatcher and a sink bolt as well as a spout for each source stream. Further, there is a bolt for each materialized operator of the query plan.

5. EVALUATION

We implemented the described approach in a system that translates queries given as operator trees into Apache Storm topologies. These topologies are then deployed on Amazon EC2 instances running where Ubuntu Server. OpenJDK-8, and Storm 1.0.2 were used for the implementation/execution. We conduct the experiments on two sets of instances, one consisting of five t2.xlarge instances, each of which has 4 CPUs and 16GB main memory, the other one comprises twenty t2.micro instances, each of which is granted 1 CPU with 1GB main memory. According to Amazon, the micro instances have lower network capabilities compared to the large instances. On the first setup instance, up to eight instances of each bolt are deployed on the same machine,

	Q2	Q3	Q5
N	110030	506985	772154
L	200055	606483	931757
M	120030	–	921757

Figure 7: Number of tuples stored.

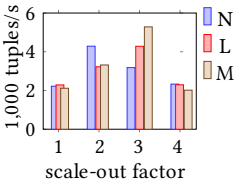


Figure 8: Average throughput for Q5.

while on the second, each bolt-instance is on its own machine.

We use TPC-H data [2] (scale factor 0.1) and the joins given by TPC-H queries Q2, Q3, and Q5. Q2 comprises a join over 5 relations, Q3 over 3, and Q5 over 4 relations. Queries Q2 and Q3 have a linear query graph and while Q5 has a cyclic one. For each query, we fed data from the according base tables into the Storm topology and measure the wallclock time until every tuple was handled. For queries Q2 and Q5 each query we build a leftdeep plan (L), a plan that only consists of a non-materialized join operator over all relations as root (M), and a third plan (mixed, M), which materializes only a part of the intermediate results, illustrated in Figure 6. This mixed plan is to a certain extent a middle ground between the other two plans. As query Q3 only combines three relations, there is no option of building a mixed plan which is not already a leftdeep one. All joins between these relations are in fact equi-joins over foreign-keys, however, we refrain from exploiting this in the routing. Still, every pair of tuples has the chance of meeting each other, which is necessary for computing actual theta-joins and thus our results are also applicable for other join predicates.

The overall **memory occupation** of a topology is actually independent of the degree of parallelism, but it depends heavily on the size of the intermediate results stored by materializing join operators. The lower bound for the required memory is the sum of the source stream sizes, which is achieved when using only a single non-materializing join over all input relations. Table 7 shows that for a query with large intermediate results like Q2, the amount of tuples that need to be stored for a leftdeep query plan is doubled compared with the single non-materialized root variant. For the other queries, the non-materializing and the leftdeep plan only differ by a factor of 1.2. Predicates with a higher selectivity, of course, produce even more intermediate results, thus rendering the materialization of infeasible. However, if still a part of the join predicate has a low selectivity, materializing exactly that part can be an option, as Q2 indicates.

Looking at the **throughput** of these query plans, measured in tuples per second, we found, that for queries Q3 and Q5, the throughput is roughly the same (independent of used plan) with 1600 tuples/s for Q3 and 2200 tuples/s for Q5. For Q2, however, the plans L and N could handle about 2700 tuples per second, while plan M had only a throughput of about 2000.

We expect the throughput to raise when the operators have a higher degree of parallelism as, thus, more parts of the join computation can be done concurrently. When scaling up, the question is, which operator should get which degree of parallelism. A simple proceeding is, to assign every operator the same number and scale this up linearly. The results for this method for query Q5 are shown in Figure 8, where the scale-out factor indicates, how many parallel instances of each store are deployed. Here, all plans gain throughput when the number of instances is doubled or tripled, however at a scale-out factor of four, additional communication slows down the processing. This suggests, that **there is a sweet spot where parallelization can be beneficially applied in order to improve throughput**. Another approach to scaling, by parallelizing these stores more which have higher storage requirements, did neither significantly boost nor degrade performance. Thus, performance is not expected to degrade if a scale-up becomes necessary.

While these results were on the set with five instance, we repeated the same experiment on the second set with forty instances, where processing units were more likely to be placed on different physical machines or even off-rack. Here the throughput went down to about two thousand tuples/second for each query plan, only a slight bump at a scale-out factor of 2 was visible. This indicates, that when network is a limiting factor, a scale out should only be done if necessary due to memory restrictions on single machines and that otherwise the communication overhead overshadows the performance gains that comes with parallelization.

6. CONCLUSION AND OUTLOOK

We have presented an approach that enables the computation of theta-joins over multiple data streams based on an flexible operator model. The model essentially allows to exploit parallelism with n-ary nodes as alternatives to simple leftdeep join processing. It further also allows trading off minimal memory usage vs. result materialization, where needed. We envision that using this model, we can further automate the construction of query plans. If also characteristics of the network are known, a cost model can be established which predicts the performance of plans.

7. REFERENCES

[1] Apache Storm. http://storm.apache.org/, 2016.

[2] The TPC-H Benchmark. http://www.tpc.org/tpch/, 2016.

[3] F. N. Afrati, M. Joglekar, C. Ré, S. Salihoglu, and J. D. Ullman. GYM: A Multiround Join Algorithm in MapReduce. *CoRR*, abs/1410.4156, 2014.

[4] R. Ananthanarayanan, V. Basker, S. Das, A. Gupta, H. Jiang, T. Qiu, A. Reznichenko, D. Ryabkov, M. Singh, and S. Venkataraman. Photon: Fault-tolerant and scalable joining of continuous data streams. SIGMOD, 2013.

[5] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and Issues in Data Stream Systems. SIGMOD, 2002.

[6] M. Elseidy, A. Elguindy, A. Vitorovic, and C. Koch. Scalable and Adaptive Online Joins. *PVLDB*, 7(6), 2014.

[7] M. A. Hammad, W. G. Aref, and A. K. Elmagarmid. Query processing of multi-way stream window joins. *VLDB J.*, 17(3), 2008.

[8] M. Hong, A. J. Demers, J. Gehrke, C. Koch, M. Riedewald, and W. M. White. Massively Multi-Query Join Processing in Publish/Subscribe Systems. SIGMOD, 2007.

[9] M. Joglekar and C. Ré. It's All a Matter of Degree: Using Degree Information to Optimize Multiway Joins. ICDT, 2016.

[10] Q. Lin, B. C. Ooi, Z. Wang, and C. Yu. Scalable Distributed Stream Join Processing. SIGMOD, 2015.

[11] A. Okcan and M. Riedewald. Processing Theta-Joins using MapReduce. SIGMOD, 2011.

[12] Y. Zhou, Y. Yan, F. Yu, and A. Zhou. Pmjoin: Optimizing Distributed Multi-way Stream Joins by Stream Partitioning. DASFAA, 2006.