Scaling Out Multi-Way Stream Joins using Optimized, Iterative Probing

Manuel Dossinger
TU Kaiserslautern (TUK)
Kaiserslautern, Germany
dossinger@cs.uni-kl.de

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

Abstract—We propose MultiStream, a novel multi-way join operator that optimizes tuple-routing schemes across materialized relations and intermediate results to compute the join results. It allows trading off materialization effort vs. network utilization, which is essential to tailoring the query evaluation to the quantity and power of available compute resources. This goes far beyond the capabilities of more rigid schemes like the HyperCube technique or the BiStream approach that is limited to simple binary plans only. Around MultiStream, we have implemented CLASH, a high-level abstraction on top of Apache Storm, allowing users to phrase declarative queries, which are then optimized and translated into executable Storm topologies. Experiments on Amazon's EC2, using TPC-H and an additional synthetic benchmark, underpin the versatility of the operator model and CLASH's ability to outperform state-of-the-art competitors.

I. INTRODUCTION

Processing data streams is a classical and ubiquitous problem. It ranges from monitoring 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 streams comprising search queries and ad-clicks in Google [1]. While this would require an equi join, joining blog posts and Twitter tweets for enriched, user-specific content delivery, requires more complex join predicates to validate text-based similarity with respect to user-defined thresholds.

In this paper, we present MultiStream, a novel multi-way join operator for processing theta joins over continuous relations. Algorithms computing such joins need to assure that all potential join partners eventually meet each other such that each result is produced 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 when additionally enabling multi-way joins in a fully integrated way. The problem also opens various ways to optimize performance. In this paper, we provide a full-fledged solution that harnesses the potential of sophisticated tuple-routing schemes for processing multi-way theta joins in a scale-out fashion. This goes far beyond existing works [2]–[4] that provide solutions to processing multi-way stream joins only for equi joins. We focus on multi-way joins using a single operator and further detail on the construction of query plans composed of multiple MultiStream operators and explore the trade-offs of different plans, an important and challenging consideration when dealing with multiple relations. For the special case of only two input relations, our approach resembles BiStream [5].

A. Problem Statement

We consider continuous join queries for join predicates $\theta$ over relations $R_1, \ldots, R_n$, the result is the subset of tuples from the Cartesian product $R_1 \times \ldots \times R_n$ that satisfy $\theta$. The predicate $\theta$ is a conjunction of arbitrary binary predicates over tuples of pairs of relations, i.e., $\theta = \bigwedge_{1 \leq i,j \leq n} \theta_{i,j}$.

Consider the example depicted in Figure 1, where relations $R$, $S$, $T$, and $U$ are joined with $\theta = \theta_{R,S} \wedge \theta_{S,T} \wedge \theta_{T,U}$. Time is evolving from left to right. Two tuples connected by a blue line satisfy the partial join predicate, e.g., $r_2$ and $s_2$ satisfy $\theta_{R,S}$ and, thus, are considered intermediate results of the join so far. Since $s_2$ does not find a join partner in $T$ (yet), they do not yet qualify as a final join result. On the other hand, $r_2$, $s_3$, $t_1$, and $u_1$, simultaneously satisfy all partial predicates, thus, should be joined and included to the result, likewise for $r_1$, $s_1$, $t_1$, and $u_1$. This example underpins that tuples, once observed, have to be stored (e.g., in main memory) for a while as they might be join partners for tuples that are arriving later on. Each relation $R_i$ has a window specification $W_i$ attached. The window specifies which tuples are potential join partners for an upcoming tuple. In this paper, we consider time-based windows [6] in a discrete time domain, including the special case of full-histoiry joins. For each point in time $\tau$, only tuples from relation $R_i$ which arrived later than $\tau - W_i$ can qualify as join partners. For instance, in Figure 1, the green boxes indicate the window of each relation at the time when $s_3$ arrives. Since only tuples inside the boxes may serve as join partners, applying the window constraint for relation $U$ shows that $u_1$ is too old and, thus, $r_2 \circ s_3 \circ t_1 \circ u_1$ does not belong to the overall result.

As the volume of data grows, it becomes necessary to distribute computation across a cluster of nodes, in order to raise available memory and exploit parallelism. Distributed computation of stream joins has been considered before; however, previous work [5], [7] emphasized joins of two relations or equi joins. In contrast to this, we study join queries over multiple relations and how the distribution can and should
be realized concerning different requirements while supporting arbitrary theta joins.

Arrival rates of the input relations and selectivities of the predicates \( \theta_{i,j} \) are assumed to be known and are plugged into a cost model that allows finding feasible query plans. Eventually, we aim to monitor such statistics, e.g., via compact and effective data synopses, and rewire the routing topology if the savings by rewiring outweigh the price for the reorganization process. This is, however, beyond the scope of this paper.

### B. Contributions and Outline

In this paper, we introduce MultiStream, a novel approach to computing multi-way joins by iteratively sending tuples in-between materialized inputs. We describe the problem of constructing query plans comprised of multi-way joins and propose optimization strategies for finding plans with cost-efficient materialization points and routing schemes. And we report on a comprehensive performance evaluation using Amazon EC2 nodes and standard benchmark data.

The remainder of this paper is organized as follows. Section II reviews related work before Section III describes the core architectural considerations in which MultiStream is realized. In Section IV, we introduce the MultiStream operator, show how to full join trees using this operator, and present optimization techniques. Section V describes the setup of the experimental evaluation, the detailed results, and the lessons learned. Finally, Section VI concludes the paper and gives a brief outlook on ongoing work.

## II. Related Work

There is a vast amount of research on processing join queries—for data streams [5], [7], for classical database systems [10], and also for distributed, scale-out scenarios [11].

In the join matrix [7], [11] scheme, relation \( R \) gets replicated across \( N \) machines, each tuple of relation \( S \) is forwarded to one of the \( N \) machines, and vice versa. For multiple relations, it becomes a hyper cube, where relations get replicated to machines for multiple dimensions [12]. The BiStream approach by Lin et al. [5] uses a tuple-routing scheme that avoids replication of data stream tuples. Both models 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 true generalization to \( n \)-way joins, that we discuss in this paper, comes with additional challenges for assuring that all potential join partners will eventually meet. Addressing multiple joins (\( n > 2 \)) in one join operator, or via a plan of multi-way operators, comes with wider flexibility in optimizing tuple routing across machines compared to a na"ive realization through sequences of binary joins.

Vitorovic et al. [13] choose cell sizes in the join matrix model such that their perimeter is minimal and thus achieve low latency and better resource utilization. Jacques-Silva et al. [14] introduce a low-latency join operator and discuss its application in a cloud environment. Zhou et al. [2] propose an approach for minimizing the communication cost between nodes when evaluating a multi-way equi join. Afrati et al. [4] present a multi-round algorithm with bounded communication cost for computing equi joins over multiple relations.

For streaming data in general, Viglas and Naughton [15] propose the use of rate-based optimization rather than classical cost-based optimization and in [16] they introduce the MJoin operator for multi-way join computation on a single node. Golab and Özsu [17] process windowed stream joins on a single machine using multiple nested loop joins, where the join order is determined using the arrival rate of the streams and selectivity of the predicates. With partial key grouping, Nasir et al. [18] introduce a value-based partitioning scheme that is able to reduce the load of individual nodes in a computing cluster if the partitioning key is skewed, and Qiu et al. [19] apply streaming hypercube for heavily skewed data. Wang and Rundensteiner [20] present a way to distribute the work of a single join operation over multiple stages by employing time-slicing of the join operators state.

Joglekar and Ré [21] 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 (cf., [22]), Hammad et al. [23] 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.

## III. System Architecture and Core Concepts

In order to process stream joins using the MultiStream paradigm and its optimizations, we have developed a query processing framework coined CLASH, designed as a high-level abstraction on top of Apache Storm [24], [25]. CLASH optimizes and translates user- or application-provided queries into operator topologies that can be deployed and run in the underlying infrastructure. CLASH is not reinventing the wheel when it comes to tuple routing primitives, like random assignment and full broadcast, by using existing stream processors as routing substrate—benefiting from provenly robust, mature systems with out-of-the-box properties such as automatic distribution and fault tolerance.
CLASH accepts queries provided in a declarative fashion, using either a SQL-like language or a programmatic API. Naturally, the results of such a query could also be computed by using routines provided by contemporary frameworks. In fact, CLASH does exactly this when translating the query into a Storm topology. It is a rather tedious task to craft such queries by hand without high-level routines. CLASH does not only ameliorate usability but can also ensure efficiency at runtime, as bad plans can be avoided that could mistakenly be handcrafted by inexperienced users. While current APIs like Flink-SQL [26] or Storm’s Trident [24] provide convenient means to implement streaming applications, they also force the user to set up the order in which operations are happening, lack join order optimization, or do not even support theta joins.

The core abstraction of CLASH is the store. A store contains data and is potentially partitioned amongst multiple compute tasks. Data contained in a store is the prefix of an input relation or intermediate results. For example, a store for \( \sigma_{a \leq 20}(R) \) contains at some time point \( \tau \) all tuples of relation \( R \) which satisfy the predicate \( a \leq 20 \), observed earlier than \( \tau \). Further, if a window \( W_R \) is specified, the store contains tuples that are observed earlier than \( \tau \) but later than \( \tau - W_R \). As time moves on, new tuples of \( R \) arrive and are logically appended to the prefix and, physically, placed in the store.

A store for relation \( R_i \) might be deployed on \( N_i \) different network-separated tasks. \( N_i \) is called the degree of parallelism of the store. If \( N_i \geq 2 \), tuples of \( R_i \) can be either placed randomly at each task, or partitioned according to a list of attributes. In the latter case, tuples can be sent directly to a single task if the attributes are present in that tuple. If the partitioning attributes are not known or the store is not partitioned, tuples can be sent to a random task of the store or broadcast to all tasks.

CLASH uses a set of features that are describing data characteristics, like tuple arrival rates and selectivities. For a given user query, the query optimizer is called that produces a query plan, consisting of multiple stores that are connected via multi-way join operators. Once the query plan is generated, CLASH translates it to a Storm topology in a fully automated fashion. The sole task left to the user is to provide a few lines of code to register input operators (called spouts in Storm) and a sink for writing join results. CLASH itself creates the necessary components that implement the stores, connects them using the according internal streams, and assigns the predicate evaluation to the correct positions.

Figure 2 shows a Storm topology for joining three relations via Apache Storm. The components for storing prefixes of \( R, S \), and \( T \) have to be registered, for each component, the routing of incoming data has to be registered, and which predicates have to be evaluated where. For example, the green arrows in the figure indicate that tuples of the particular relation (either \( R, S \), or \( T \), or the query result) are sent for storing/outputting to the receiving components, while the blue dashed arrows indicate probing. This means that the \( R \)-store receives tuples from the \( R \)-spout for storing, from the \( S \)-store for probing and evaluating the predicate \( \theta_{R,S} \), and from the \( S \)-store for probing and evaluating the predicates \( \theta_{R,S} \) and \( \theta_{R,T} \).

IV. THE MULTISTREAM OPERATOR

The MultiStream join operator (MSJ) computes the join result over multiple input relations and arbitrary join predicates by creating stores for each input and iteratively probing the prefixes for every incoming tuple. We illustrate MSJ as an n-ary node, where child nodes correspond to input relations, as shown in Figure 3a.

Figure 3b depicts how MSJ is joining three relations \( R, S \), and \( T \). For each relation, a store is registered and distributed over multiple tasks, e.g., \( R \) is partitioned into \( R_1 \) and \( R_2 \). A tuple \( r \) originating from \( R \) is sent to one randomly chosen task of the \( R \)-store for storing it (green arrow). This tuple is also sent to both tasks of the \( S \)-store where the prefixes are probed in order to compute \( (r \bowtie S_1) \cup (r \bowtie S_2) = r \bowtie S \) (blue arrows). If it was sent to only one task, we could not guarantee that all results are found. The partial results \( r \bowtie S_1 \) and \( r \bowtie S_2 \) are both broadcast to both \( T \)-tasks, where the final result is produced. That means, every store has two primary tasks: (1) receiving and storing tuples of the “own” relation and (2) receiving and probing tuples from other relations and sending results, if any, to other stores or sinks. This is similar to iteratively probing hash tables in MJoin [16] for centralized environments.

A. Iterative Probing

The order in which the stores are probed for a specific tuple during join computation does not affect the correctness of the algorithm. However, in MSJ, this so-called probe order is exploited to reduce network traffic. More formally, for relation \( R_i \), we can write the probe order as follows:

\[
\sigma_{R_i} := \{R_{\sigma_i(1)}, R_{\sigma_i(2)}, \ldots, R_{\sigma_i(n)}\}. \tag{1}
\]
It is a permutation of the relations participating in the MSJ, with \( R_{\sigma_i(1)} = R_i \), and specifies the sequence in which a tuple \( r_i \in R_i \) has to be sent to the other stores for iteratively computing the intermediate results \( r_i \bowtie R_{\sigma_i(2)} \bowtie R_{\sigma_i(3)} \bowtie \ldots \). These probe messages are broadcast to every task of a store, such that every potential join partner is met and no join result is missed. For the special case of only two relations, there is only one trivial probe order and no intermediate results, thus, a setup identical to BiStream \([5]\).

We now give an estimation of the issued traffic cost, starting with a single probe order for \( R_i \), \( \sigma_i \). We write \( |R| \) for the number of tuples contained in a relation with respect to the specified window sizes and denote \( f_{i,j} \) as the join selectivity between \( R_i \) and \( R_j \). At first, we only look at the sizes of the produced intermediate results. All tuples from \( R_i \) are broadcast to the \( R_{\sigma_i(2)} \)-store in order to be joined there. A single tuple \( r_i \) finds \( |R_{\sigma_i(2)}| \cdot f_{i,\sigma_i(2)} \cdot \frac{1}{2} =: c_2 \) tuples. While \( |R_{\sigma_i(2)}| \cdot f_{i,\sigma_i(2)} \) is textbook cardinality estimation \([27]\), the factor of \( \frac{1}{2} \) comes from the fact that only tuples \( r_{\sigma_i(2)} \leq r_i \) (i.e., \( r_{\sigma_i(2)} \) arrives earlier than \( r_i \)) are contained in the join result. The reminder of \( R_i \bowtie R_{\sigma_i(2)} \) is computed on the \( R_i \)-store. The generated results are then sent to the \( R_{\sigma_i(3)} \)-store, and here we expect \( c_2 \cdot |R_{\sigma_i(3)}| \cdot f_{i,\sigma_i(3)} \cdot f_{\sigma_i(2),\sigma_i(3)} \cdot \frac{1}{2} \) tuples. Again, this is a combination of the selectivity-based estimation of the join size combined with the restriction on tuples that with \( r_{\sigma_i(3)} \leq r_i \). With these considerations, and the fact that each tuple created at step \( j - 1 \) is broadcast and, thus, copied \( N_j \) times, the general estimation on probe cost for the MSJ operator \( O \) is given as:

\[
P\text{Cost}(O) = \sum_{1 \leq i \leq n} N_{\sigma_i(j)} \cdot \prod_{1 \leq k \leq j - 1} |R_{\sigma_i(k)}| \cdot \prod_{1 \leq k \leq j - 1} \prod_{1 \leq k' \leq j - 1} f_{k,k'} \cdot \frac{1}{j}.
\]

\[
(2)
\]

**B. Probe Order Optimization**

A bad choice of probe orders leads to high traffic between the involved tasks due to an unnecessarily large amount of intermediate results. The entire space of possible probe orders for \( n \) relations consists of \((n-1)!\) possibilities to choose from, one for each starting relation. In order to find viable probe orders in feasible time, we leverage the bottom-up greedy algorithm shown in Figure 4. This algorithm incrementally builds probe order \( \sigma_R \), and tracks the so-far joined relations in variable \( R \). The loop in Lines 2–4 is iterated as long as there are more than two not-yet joined relations \( R' \). In Line 3, the next relation \( R_j \) for the probe order is chosen as the relation which issues the least duplication of tuples of the current join result \(|R \cdot N_j|\) and also produces intermediate join results weighted by the parallelism of all other relation stores \( R_k \) that could be joined afterwards \(|R \bowtie R_j \cdot N_k|\). Note that \( R_k \) is not scheduled yet, instead all possible \( R_k \) are tested in order to avoid a choice for \( R_j \) that causes a bigger intermediate result in the next iteration. This algorithm is invoked once for each starting relation of the MultiStream operator at hand; thus, the overall runtime for \( n \) input streams is in \( O(n^3) \). While this could be a show-stopper for ad-hoc query answering for larger \( n \), it seems to be acceptable for continuous queries for practical numbers of streams to be joined. For instance, for \( n = 7 \), we observe wall-clock times of less than a second on standard CPUs.

**C. Composing MSJs to Trees**

Instead of using a single MSJ operator for joining relations \( R_1, R_2, \ldots, R_n =: R \), we can replace a subset \( R' \subset R \) with another MSJ operator and materialize its result into a newly created \( R' \)-store. This means, in general, we can build a query plan that is a multi-way tree with \( n \) leaves and multiple MSJs as inner nodes. For example, in Figure 5 we see two query plans for a join involving relations \( R_1, R_2, R_3, R_4 \). A plan on the left-hand side using only one MSJ as root (we call this flat) and, on the right-hand side, a bushy plan with relations \( R' = \{R_2, R_3, R_4\} \) being joined in a separate MSJ whose result is materialized. In order to highlight all materialized relations, we visualize input relations and inner joins with squares and the non-materialized root using an ellipse.

In general, each additional store is expected to increase the storage cost, but at the same time, will presumably lower the probe cost. The cost considerations for storage and probing can be naturally extended to a tree \( T \) composed of a root \( O_{\text{root}} \), inner operators \( O \), and source relations \( R_i \). The total storage cost for a join tree is the sum over the size of the prefix of each individual relation and the sizes of the joins of the inner operators, denoted \( |O| \):

\[
S\text{Cost}(T) = \sum_i |R_i| + \sum_{O \in \mathcal{O}} |O|.
\]

As the tuples of \( O_{\text{root}} \) are not materialized, it is not included in \( S\text{Cost} \). The total probe cost is composed of the individual
probing cost of each involved operator:

\[
PCost(T) = \sum_{O \in O \setminus \{O_{root}\}} PCost(O). \tag{4}
\]

D. Optimizing Linear Join Graphs (Chains)

We now present an algorithm for optimization of linear join graphs under constrained storage budget. The goal of this algorithm is to minimize the communication cost by materializing as many intermediate results as possible. A linear join graph consists of relations \(R_1, ..., R_n\) with join predicates \(\theta_{i,i+1}\) (e.g., TPC-H queries Q2 and Q3). We construct a plan using a top-down strategy that starts with a flat tree and incrementally fuses neighboring relations into a new join operator until the system capacity is exceeded. The pseudocode in Figure 7 describes this: First, a flat tree is initialized in Lines 1 and 2, and the initially consumed budget is set. In this algorithm, the order of the children is significant and we label each node with the list of all leaves below.

As long as the budget is not exceeded, the algorithm searches for relations to merge as follows: For all indexes \(i\) and \(j\) it checks using the \(valid\)-function whether relations \(R_i\) up to \(R_j\) can be fused into a new MSJ operator. This is the case if a node \(v\) has two children, one where \(R_i\) is the leftmost and one where \(R_j\) is the rightmost entry of the label, and at least one child with an index below \(i\) or above \(j\). If so, the \(mat\) function can materialize the result of the join of \(R_i\) up to \(R_j\) into a new operator which is put as child of \(v\) and the nodes with \(R_i\) up to \(R_j\) are reassigned to \(v\). For all valid combinations of \(i\) and \(j\), the pair with minimal cost for materializing the join of relations \(R_{i_1}, ..., R_{j_k}\) is selected in Line 4. If the cost for materializing this would exceed the budget, the algorithm terminates—otherwise, it introduces a materialization for these relations and continues. Finally, that tree is returned in Line 8.

For four relations, the algorithm starts with Shape I, as depicted in Figure 6. It then finds the pair \((i, j) \in \{(1, 2), (1, 3), (2, 3), (2, 4), (3, 4)\}\) where materializing the join \(\times_{i \leq k \leq j} R_k\) adds the least SCost. If the pair found is \((1, 2)\), the tree has Shape II, and in the next iteration the valid choices for \((i, j)\) are \((1, 3)\) (which would lead to Shape V) and \((3, 4)\) (leading to Shape III). If the pair found in the first iteration was Shape IV, the valid index pairs for the next iteration are \((1, 2)\) and \((2, 3)\), both leading to a tree of Shape V, but the latter would materialize \(R_2 \times R_3\), so it is a zig-zag tree.

input: relations \(R\), budget \(B\)
1. \(T \leftarrow createFlatTree(R)\)
2. \(b \leftarrow \sum_{R_i \in R} |R_i|\)
3. while \(b < B\) and there are splittable nodes
4. \((i, j) \leftarrow \arg \min_{(i, j), s.t. valid(T, i, j)} \{SCost(mat(T, i, j))\}\)
5. if \(B < SCost(mat(T, i, j))\)
6. break
7. \(T \leftarrow mat(T, i, j)\)
8. return \(T\)

E. Correctness

MSJ is correct, i.e., computes all join results exactly once, if tuples are not too delayed. This is achieved by storing probe tuples \(r_p\) for a small period and allow them to probe with later arriving store tuples \(r_s\) if \(r_s \preceq r_p\). The longer this period is, the more tuples can be delayed, and more storage is required for keeping probe tuples in every store. Correctness under the presence of node failures can be handled by the underlying stream processor. As stream join is a stateful operation, the state has to be recoverable, e.g., by replaying input tuples or by copying the store tuples to remote replicas.

V. EXPERIMENTAL EVALUATION

In this section, we explore the performance of the MSJ operator applied to different queries and datasets. We compare MSJ with a left-deep cascade of binary MSJs resembling the state-of-the-art BiStream approach for multiple relations as briefly sketched in [5]. This is also the way Flink executes joins when specified using their SQL interface. Between these extremes, we also explore the performance of different operator trees created by different optimization strategies. We deploy the generated Apache Storm topologies on an Amazon EC2 cluster consisting of multiple t2.medium instances, each having two virtual cores, 4 GB main memory, and being interconnected with “low to moderate” networking performance. According to iperf measurements, the network provides at least 50 Mbps. Each instance is running up to two workers, and the number of instances is adjusted based on requirements of the plan and degree of parallelism, such that each there is no contention on CPU resources. Storm version 1.2.2 is used running on OpenJDK 11 and Ubuntu Server 18.04.02. Locally, we use a standard nested loop join which is able to produce the

Fig. 6: All structurally different join trees over four streamed relations.

Fig. 7: Top-down strategy for chain queries that incrementally adds materialization.
desired result for any given computable binary predicate. The source code of CLASH, including the MultiStream operator as described in this paper, is publicly available on GitHub\textsuperscript{7}.

**Benchmark Data and Queries:** We use data from the TPC-H benchmark \textsuperscript{28}, which is commonly used also for evaluating stream processors. We use scale factors 1 to 20 and the joins inside queries Q2, Q3, and Q5. For example, Q2 normally selects the supplier that has minimal cost. As we are focusing on join processing and not on further aggregation, we do not compute this minimum, and no other subqueries or aggregations. The TPC-H queries consist of equi joins according to the foreign-key relations between tables, yet, we refrain from exploiting this in the routing. This means that our performance results remain valid for other, non-equi-join predicates. Due to MSJ’s random tuple placement, skewed attributes do not affect the computation.

Further, we use a custom generated dataset that allows the creation of linear join queries with arbitrary selectivities for the individual joins. This way, we can specifically explore the creation of linear join queries with arbitrary selectivities for attributes do not affect the computation. Due to MSJ’s random tuple placement, skewed yet, we refrain from exploiting this in the routing. This means that our performance results remain valid for other, non-equi-join predicates. Here, we show the size of the partial prefixes $P_1, \ldots, P_k$ that are kept in four tasks and belong to the same store (relation).

**B. Effect of Materialized Intermediates on Communication**

Here, we compare the savings for different scenarios. First, for custom-$LLLL$ a five-way, low-selectivity join with equal relation sizes of $10^6$ and a pairwise selectivity of $10^{-8}$. Figure 8a shows the number of probed tuples depending on the capacity of a single store. If the task capacity is enough for a tenth of the incoming tuples of each relation (the leftmost case on the x-axis), over half of the probe tuples can be saved with the left-deep compared to a flat plan. The more task capacity there is, the less need for parallelizing the individual stores, and consequently also less probe overhead. If every task can handle the entire incoming relations, the difference is negligible (the rightmost case on the x-axis).

For medium selectivities of $10^{-6}$ as shown in Figure 8a, flat still shows an advantage; however, the relative saving is not that big anymore. If the selectivities are getting larger to $10^{-5}$, shown in Figure 8b then the flat plan’s performance degrades further. This effect is due to the increased parallelism requirement on the stores of the intermediate relations: the higher the selectivity, the more intermediate results, and the more tasks are required in order to store the prefixes. Therefore, probe tuples have to be broadcast to more stores, increasing the overall communication. This means, a flat plan, consisting of a single MSJ, is very well usable for scenarios where high selectivity joins are involved.

**C. Throughput and Scalability**

For measuring the throughput of different query plans, we feed all tuples into the corresponding Storm topologies, without rate restrictions, and measure the time it takes to compute the entire result. The numbers are average values from ten runs. The throughput is then the total number of read input tuples divided by the time spent. The results for TPC-H query Q2 are shown in Figure 8c. Q2 essentially consists of a chain-style join graph (part—partsupp—supplier—nation—region). The parallelism factor indicates the degree of parallelism per store. This means, for a parallelism factor of $k$, the flat plan consists of $k$ stores, and the left-deep plan of $2k - 1$. We see that with one task per store (the minimum required for running this topology), the flat plan provides the highest throughput, as there is less overhead due to fewer active bolts and a higher probability of two tasks being on the same physical node in the topology. In order to understand how throughput behaves, when a store is parallelized over multiple tasks, we increase the

---

\textsuperscript{7} https://github.com/clash-streaming/clash
For the last test, we used custom-MLM with \(10^6\) tuples in each relation and \(M = 10^{-6}\) and \(L = 10^{-5}\). This four-way linear query has one join with big intermediate results in the middle which have to be computed via probing eventually. In Figure 8h we see the impact of this, most importantly, the throughput is lower than for Q2 by an order of magnitude. Secondly, increasing the available tasks and scaling out the stores does not bring a significant performance boost with any plan. The plans used correspond to the shapes illustrated in Figure 8f. the flat plan matches Shape I and the left-deep plan is an instance of Shape V, as the optimized. The other shapes are presented for completeness with relations ordered for optimal storage cost. This means if join queries have huge intermediate results, they do not have to be materialized in order to get throughput performance gains.

D. Latency

In order to measure the latencies of tuples, we assign each tuple \(t\) the timestamp of the system time using the Java system method `currentTimeMillis()`. The resulting tuple of a join between \(t\) and another previously-stored tuple gets the same timestamp assigned. If this tuple finally arrives at the sink, i.e., if \(t\) finds join partners such that the overall query is satisfied, the sink again reads the current system time and reports the difference between those timestamps in milliseconds. This measurement requires the clocks of the hosts where dispatcher and sink run to be synchronized, which we accomplish by calling the Unix command `ntptime` before every run. In contrast to the throughput measurements, here we feed the input tuples with a low rate into the topology in order to avoid measuring the time how long a tuple is buffered in the in- or output queues of the tasks.

---

**Fig. 8: Experimental evaluation.**
The boxplots in Figure 8j show that intelligent placement of communication boundaries effectively limits the latency encountered for Q2, where the average latency of the optimized is half of the latency of the left-deep or flat plan—the boxes are showing the quartile bounds and the whiskers indicate the 0.025 and 0.975 quantiles. This is due to the, on average, lower number of network hops needed to compute the joins. Figure 8k shows the results for answering query custom-MM with $10^9$ tuples and $M = 10^{-6}$ using the left-deep plan under different parallelization factors. The median latency is reduced when using more store instances in parallel. However, also much more tuples need also significantly more time since the network load increases due to the excess of broadcast tuples.

In order to understand the decrease in average latency, we look at the time needed to compute a local join. As we use nested loop joins in order to remain compatible with the ultimate goal of computing theta joins, the complexity depends on the size of the stored prefix. Thus, with a parallelism factor of 4, only a quarter of the join predicates have to be evaluated in each store instance compared to parallelism factor 1, and the evaluation can be done in each store instance in parallel. Figure 8l shows how the time to locally compute a join when a probe tuple arrives changes over the course of a workload computation. As more tuples arrive, the time also changes. However, with a higher degree of parallelism, the number of join candidates found per probe in a store instance grows more slowly, hence the time of the local join computation influences the end-to-end latency less.

E. Lessons Learned

The development and evaluation of the MultiStream Join operator revealed interesting insights that can be summarized as follows: Smart placement of materialization points can benefit throughput as well as latency for computing stream joins over arbitrary predicates. The avoidance of materialization leads to higher network load, but renders the computation of queries with large intermediate results possible, by moving these intermediate results into the network. Conversely, by materializing fewer results, the long-living state is reduced which again can reduce redundant network traffic. Scaling out distributed theta joins has to be done with care, as scaling out these intermediate results into the network. Conversely, by queries with large intermediate results possible, by moving leads to higher network load, but renders the computation of operator revealed interesting insights that can be summarized slowly, hence the time of the local join computation influences the end-to-end latency less.

VI. CONCLUSION AND OUTLOOK

In this work, we presented the novel multi-way join operator MultiStream. MultiStream answers join queries over multiple relations with optional intermediate result materialization for trading-off network consumption for memory requirements. We described cost-based optimization and showed the effectiveness of query planning. Future work considers online adaption to changing characteristics of the incoming data streams.

REFERENCES