AJoin: Ad-hoc Stream Joins at Scale

Jeyhun Karimov
DFKI GmbH
jeyhun.karimov@dfki.de

Tilmann Rabl*
Hasso Plattner Institute, University of Potsdam
tilmann.rabl@hpi.de

Volker Markl
DFKI GmbH, TU Berlin
volker.markl@tu-berlin.de

ABSTRACT

The processing model of state-of-the-art stream processing engines is designed to execute long-running queries one at a time. However, with the advance of cloud technologies and multi-tenant systems, multiple users share the same cloud for stream query processing. This results in many ad-hoc stream queries sharing common stream sources. Many of these queries include joins.

There are two main limitations that hinder performing ad-hoc stream join processing. The first limitation is missed optimization potential both in stream data processing and query optimization layers. The second limitation is the lack of dynamicity in query execution plans.

We present AJoin, a dynamic and incremental ad-hoc stream join framework. AJoin consists of an optimization layer and a stream data processing layer. The optimization layer periodically reoptimizes the query execution plan, performing join reordering and vertical and horizontal scaling at run-time without stopping the execution. The data processing layer implements pipeline-parallel join architecture. This layer enables incremental and consistent query processing supporting all the actions triggered by the optimizer. We implement AJoin on top of Apache Flink, an open-source data processing framework. AJoin outperforms Flink not only at ad-hoc multi-query workloads but also at single-query workloads.

PVLDB Reference Format:
Jeyhun Karimov, Tilmann Rabl, and Volker Markl. AJoin: Ad-hoc Stream Joins at Scale. *PVLDB*, 13(4): 435-448, 2019.
DOI: https://doi.org/10.14778/3372716.3372718

1. INTRODUCTION

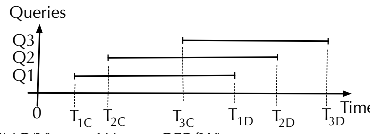
Stream processing engines (SPEs) process continuous queries on real-time data, which are series of events over time. Examples of such data are sensor events, user activity on a website, and financial trades. There are several open-source streaming engines, such as Apache Spark Streaming [4, 53], Apache Storm [48], and Apache Flink [15], backed by big communities.

With the advance of cloud computing [20], such as the Software as a Service model [51], multiple users share public or private clouds for stream query processing. Many of these queries include joins. Stream joins continuously combine rows from two or more bounded streaming sources. In particular, executing multiple ad-hoc queries on common streaming relations needs careful consideration to avoid redundant computation and data copy.

*Work was partially done while author was at TU Berlin.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 13, No. 4
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3372716.3372718



$Q1 = \sigma_{V.lang=ENG}(\mathbf{V}) \bowtie_{V.vID=W.vID} \sigma_{W.geo=GER}(\mathbf{W})$, Wnd=1s

$Q2 = \sigma_{V.geo=US}(\mathbf{V}) \bowtie_{V.vID=W.vID} \sigma_{W.duration<10}(\mathbf{W}) \bowtie_{W.usrID=C.usrID} \sigma_{C.length>5}(\mathbf{C})$, Wnd=2s

$Q3 = \sigma_{W.geo=EU}(\mathbf{W}) \bowtie_{W.usrID=C.usrID} \sigma_{C.emojis>0}(\mathbf{C}) \bowtie_{C.usrID=R.usrID} \sigma_{R.reaction=angry}(\mathbf{R})$, Wnd=3s

Figure 1: Ad-hoc stream join queries. T_{iC} and T_{iD} show creation and deletion times of ith query, respectively.

Motivation. Stream join services are used in many companies, e.g., Facebook [32]. Clients subscribed to such a service create and delete stream join queries in an ad-hoc manner. In order to execute the queries efficiently, a service owner needs to periodically reoptimize the query execution plan (QEP).

Let V={vID, length, geo, lang, time} be a stream of videos (videos displayed at a user wall), W={usrID, vID, duration, geo, time} a video view stream of a user, C={usrID, comment, length, photo, emojis, time} a stream of user comments, and R={usrID, reaction, time} a steam of user reactions, such as like, love, and angry. Figure 1 shows an example use-case scenario for ad-hoc stream join queries. The machine learning module initiates Q1 to feed the model with video preferences of users. The module targets people living in Germany ($\sigma_{\text{W.geo=GER}}$) and watching videos in English ($\sigma_{\text{V.lang=ENG}}$). The editorial team initiates Q2 to discover web brigades or troll armies [10, 45]. The query detects users that comment ($\sigma_{\text{C.length}>5}$) on videos published in US ($\sigma_{\text{V.geo=US}}$) just a few seconds after watching them ($\sigma_{\text{W.duration}<10}$). The quality assurance team initiates Q3 to analyze the users' reactions to promoted videos. Specifically, the team analyzes videos that are watched in Europe ($\sigma_{\text{W.geo=EU}}$), receive `angry` reactions ($\sigma_{\text{R.reaction=angry}}$), and at least one emoji in comments ($\sigma_{\text{C.emojis}>0}$). We use the queries, shown in Figure 1, throughout the paper.

As we can see from the example above, these stream queries are executed within a finite time duration. Depending on ad-hoc query creation and deletion time and selection predicates, (W⋈V) or (W⋈C) can be shared between Q1 and Q2 or between Q2 and Q3, respectively. Different sharing strategies can also require reordering the join relations.

With many concurrent join queries, data copy, computation, and resource usage will be a bottleneck. So, scan sharing for common data sources and object reuse are necessary. Also, the data and query throughput can fluctuate at run-time. To support such dynamic workloads, SPEs need to support scale out and in, and scale up and down, and join reordering at run-time, without stopping the execution. Note that the state-of-the-art streaming systems are optimized for maximizing the data throughput. However, in a multi-user cloud environment it is also important to maximize query throughput (frequency of created and deleted queries).

Sharing Limitations in Ad-hoc SPEs. Ad-hoc query sharing has been studied both for batch and stream data processing systems. Unlike ad-hoc batch query processing systems, in ad-hoc SPEs query sharing happens between queries running on fundamentally different subsets of the data sets, determined by the creation and deletion times of each query. Below, we analyze the main limitations of modern ad-hoc SPEs.

`Missed optimization potential:` To the best of our knowledge, there is no ad-hoc SPE providing ad-hoc stream QEP optimization. Modern ad-hoc SPEs embed rule-based query sharing techniques, such as query indexing [17], in the data processing layer [35]. However, appending a query index payload to each tuple causes redundant memory and computation usage. As the number of running queries increases, each tuple carries more payload.

Modern ad-hoc SPEs materialize intermediate join results eagerly. Especially with low selectivity joins, the eager materialization results in high transfer costs of intermediate results between subsequent operators.

Also, the join operator structure in modern SPEs performs several costly computations, such as buffering stream tuples in a window, triggering the processing of a window, computing matching tuples, and creating a new set of tuples based on matching tuples. With more queries and n-way ($n \geq 3$) joins, the join operation will be a bottleneck in the QEP.

`Dynamicity:` Modern ad-hoc SPEs consider ad-hoc query processing only with a static QEP and with queries with common join predicates. In stream workloads with fluctuating data and query throughput, this is inefficient.

AJoin. We propose AJoin, a scalable SPE that supports ad-hoc equi-join query processing. AJoin also supports selection operators. We overcome the limitations stated above by combining incremental and dynamic ad-hoc stream query processing in our solution:

`Efficient distributed join architecture:` Because the join operator in modern SPEs is computationally expensive, AJoin shares the workload of the join operator with a source and sink operator. The join architecture is not only data-parallel but also pipeline-parallel. Tuples are indexed in the source operator. The join operator utilizes indexes for an efficient join operation. AJoin incrementally computes multiple join queries. It performs a scan, data, and computation sharing between multiple join queries with different predicates. Our solution adopts late materialization for intermediate join results. This technique enables the system to compress the intermediate results and pass them to downstream operators efficiently.

`Dynamic query processing:` AJoin supports dynamicity at the optimization and data processing layer: dynamicity at the optimization layer means that the optimization layer performs regular reoptimization, such as join reordering and horizontal and vertical scaling; dynamicity at the data processing layer means that the layer is able to perform all the actions triggered by the optimizer at run-time, without stopping the QEP.

Contributions and Paper Organization. The main contributions of the paper are as follows: (1) We present the first optimizer to process ad-hoc streaming queries in an incremental manner; (2) We develop distributed pipeline-parallel stream join architecture. This architecture also supports dynamicity (modify QEP on-the-fly in a consistent way); (3) We perform an extensive experimental evaluation with state-of-the-art streaming engines.

The rest of the paper is organized as follows. We present related work in Section 2. Section 3 gives the system overview. Section 4 presents the AJoin optimizer. We provide implementation details in Section 5 and run-time operations in Section 6. Experimental results are shown in Section 7. We conclude in Section 8.

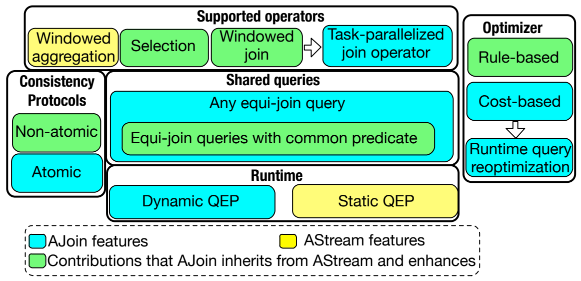


Figure 2: Comparison between AJoin and AStream

2. RELATED WORK

Shared query processing. SharedDB is based on batch data processing model and handles OLTP, OLAP, and mixed workloads [22, 23]. Giceva et al. adopt SharedDB ideas and implement shared query processing on multicores [25]. CJoin [12, 13] and DataPath [6] focus on ad-hoc query processing in data warehouses. Braun et al. propose a hybrid (OLTP and OLAP) computation system, which integrates key-value-based event processing and SQL-based analytical processing on the same distributed store [11]. BatchDB implements hybrid workloads sharing for interactive applications [42]. SharedHive is a shared query processing solution built on top of MapReduce framework [18]. The works mentioned above are designed for batch data processing environments. Although we also embrace some ideas from shared join operators, we focus on stream data processing environments with ad-hoc queries.

To increase data throughput, MJoin proposes a multi-way join operator that operates over more than two inputs [52]. While the bucket data structure in AJoin also mimics the behavior of multi-way joins, the join operator of AJoin supports binary input streams. To increase data throughput, AJoin reoptimizes the QEP periodically. FluxQuery is a centralized main-memory execution engine based on the idea of continual circular clock scans and adjusted for interactive query execution [19]. Similarly, MQJoin supports efficient ad-hoc execution of main-memory joins [41]. Hammad et al. propose streaming ad-hoc joins [26]. The solution adopts a centralized router, extended from Eddies [7]. Also, the work adopts a selection pull-up approach, which might result in high bookkeeping cost of resulting joined tuples and intensive CPU and memory consumption. The above works are designed for a single-node environment. However, AJoin is designed for distributed environments. AJoin does not utilize any centralized computing structure. Dynamicity and progressive optimization, are more essential in distributed environments. Also, AJoin exploits pipeline-parallelism. In a single-node environment, however, task-fusion is more beneficial [55].

AJoin vs AStream. AJoin was inspired by AStream [35], the first shared ad-hoc SPE. AStream adds an additional attribute to each tuple that represents a bitset of potentially interested queries in that tuple. This attribute is called `query-set`. For example, a query-set `0011` means that the tuple matches selection predicates of the third and fourth queries. AStream also adopts changelogs that is a special data structure consisting of *i*) query deletion and creations meta-data and *ii*) a changelog-set, a bitset encoding the associated query deletions and creations. By utilizing query-sets and changelog-sets, AStream ensures consistent query creation and deletion.

Figure 2 shows an architectural comparison between AJoin and AStream. AJoin inherits query-sets and changelogs from AStream. Also, AJoin enhances the rule-based optimizer of AStream. Instead of encoding all queries in a query-set, AJoin arranges queries with similar selection predicates into the same groups.

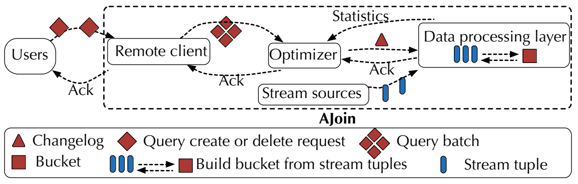


Figure 3: AJoin architecture

This enables AJoin to lower the cost of sharing and the query-set payload. AJoin features a cost-based query optimizer that performs progressive query optimization periodically at run-time. AStream performs data and computation sharing aggressively, which might lead to suboptimal QEP. AJoin, however, shares data and computation if the sharing is beneficial. Similar to AStream, AJoin supports selection and windowed join operators. However, the selection operator executes on a group of queries (determined at run-time), rather than a full set of queries. Also, AJoin utilizes an efficient and pipeline-parallelized join architecture. Different from AStream, AJoin supports sharing of any equi-join query. AJoin features a dynamic data processing layer that is able to perform QEP changes at run-time. To be able to perform the run-time changes, AJoin inherits the non-atomic consistency protocol from AStream and provides an atomic consistency protocol.

Adaptive query processing. Progressive query optimization, POP, uses cardinality boundaries in which a selected plan is optimal [43]. Our optimizer uses a similar idea, cost sharing, but we target streaming scenarios. Li et al. propose adaptive join ordering during query execution [39]. The solution adds an extra operator, a local predicate on the driving table to exclude the already processed rows if the driving table is changed. We perform join reordering without extra operators.

Gedik et al. propose an elastic scaling framework for data streams [21]. Cardellini et al. propose a similar idea on top of Apache Storm [16]. Both works use state migration as a separate phase to redistribute the state among nodes. AJoin, on the other hand, features a smooth repartitioning scheme, without stopping the topology. Heinze et al. propose an operator placement technique for elastic stream processing [28, 29]. AJoin does not perform operator placement optimization for all streaming operators but only for join and selection operators (e.g., grouping queries and executing them in specific operators).

Query optimization. Trummer et al. solve join ordering problem via a mixed integer programming model [49]. Although this approach is acceptable in a single query environment, with ad-hoc queries we need an optimization framework that can optimize incrementally. Unlike dynamic programming approaches [44, 46], current numerical optimization frameworks lack this feature. The IK/KBZ family of algorithms can construct the optimal join plan in polynomial time [38, 31]. The iterative dynamic programming approach combines benefits from both dynamic programming and greedy algorithms [37]. To perform incremental query optimization, we adopt ideas from this technique and enhance them for our scenario.

3. SYSTEM OVERVIEW AND EXAMPLE

In this section, we provide a high-level overview of AJoin. Figure 3 shows the architecture of AJoin. The remote client listens to users requests, such as query creation or deletion requests. It batches user requests in a `query batch` and sends this batch to the optimizer. Apart from query batches, the optimizer periodically receives statistics from the data processing layer. It periodically reoptimizes the QEP based on statistics and received query batches. As part of the reoptimization, the optimizer triggers actions, such as scale up and down, scale out and in, query pipelining, and join reordering. Similarly, the data processing layer performs all the actions at run-time, without stopping the QEP. AJoin supports equi-joins with event-time windows and selection operators.

Data Model. There are three main data structures in AJoin: a stream tuple, a bucket, and a changelog. Source operators of AJoin pull stream tuples from external sources. Then, the tuples are transformed into the internal data structure of AJoin, which is a `bucket`. Below, we discuss the data structures bucket and changelog in detail.

A `bucket` is the main data structure throughout the QEP. It contains a set of index entries and stream tuples corresponding to the index entries. Each bucket includes a bucket ID. Stream tuples in a bucket can be indexed w.r.t. different attributes. Buckets are read-only. All AJoin operators, except for the source operator, receive buckets from upstream operators and output new read-only buckets. This way, the buckets can be easily shared between multiple concurrent stream queries.

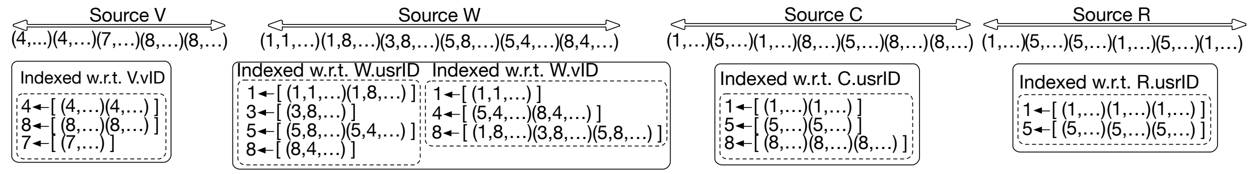
Figure 4a shows stream tuples generated from sources V, W, C, and R and generated buckets from the respective stream sources. The bucket generated from the stream source V is indexed w.r.t. V.vID attribute because the downstream join operator uses the predicate V.vID=W.vID. However, the bucket generated from the stream source W is indexed w.r.t. two attributes: W.vID and W.usrID. The reason is that *i*) Q1 and Q2 requires indexing w.r.t. the attribute W.vID and *ii*) Q3 requires indexing w.r.t. the attribute W.usrID. Unless stated otherwise, we assume that the join ordering of Q2 is $(\mathrm{V} \bowtie_{\mathrm{V.vID=W.vID}} \mathrm{W}) \bowtie_{\mathrm{W.usrID=C.usrID}} \mathrm{C}$.

A `changelog` is a special marker dispatched from the optimizer. It contains metadata about QEP changes, such as horizontal or vertical scaling, query deletion, and query creation. A changelog propagates through the QEP. Operators receiving the changelog update their execution accordingly.

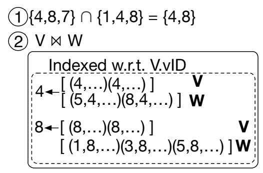
Join Operation. In modern SPEs, such as Spark [4], Flink [15], and Storm [48], the computation distribution of a join operation is rather skewed among different stream operators: source, join, and sink operators. For example, the source operator is responsible for pulling stream tuples from external stream sources. The join operator buffers stream tuples in a window, finds matching tuples, and builds resulting tuples by assembling the matching tuples. The join operator also implements all the functionalities of a windowing operator. The sink operator pushes the resulting tuples to external output channels. Because most of the computation is performed in the join operator, it can easily become a bottleneck. With more concurrent n-way join queries ($n \geq 3$), the join operator is more likely to be a limiting factor.

To overcome this issue, we perform two main optimizations. First, we perform `pipeline parallelization` sharing the load of the join operator between the source and sink operators. The source operator combines the input data acquired in the last t time slots and builds a bucket. With this, we transmit the windowing operation from the join operator to the source operator. Also, buckets contain indexed tuples, which are used at the downstream join operator to perform the join efficiently. Afterwards, the partitioner distributes buckets based on a given partitioning function. Then, the join operator performs a set intersection between the index entries of input buckets. Note that for all downstream operators of the source operator, the unit of data is a bucket instead of a stream tuple. Finally, the sink operator performs full materialization, i.e., it converts buckets into stream tuples, and outputs join results.

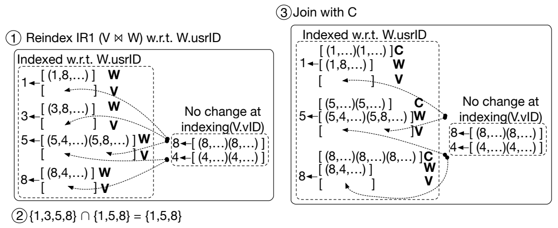
Second, we perform `late materialization` of intermediate join results. After computing the matching tuples (via intersecting index entries), the join operator avoids performing the cross-product among them. Figure 4b shows the join operation for Q1. Index entries from the two input buckets are joined (①). Then, tuples with the matched indexes are retained in the resulting bucket (②). The late materialization technique can also be used for n-way joins. For example, Figure 4e shows the



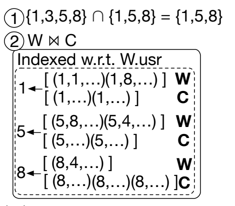
(a) Tuples (after applying filter) from stream sources V, W, C, and R (top row) and constructed buckets from the respective sources (bottom row)



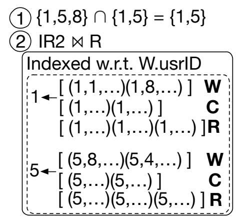
(b) $V \bowtie_{V.vID=W.vID} W$, result of Q1 and intermediate result for Q2 (IR1)


① Reindex IR1 (V ⋈ W) w.r.t. W.usrID
Indexed w.r.t. W.usrID
No change at indexing(V.vID)
③Join with C
②{1,3,5,8} ∩ {1,5,8} = {1,5,8}

(c) Result of Q2



(d) $W \bowtie_{W.usrID=C.usrID} C$, intermediate result for Q3 (IR2)



(e) Result of Q3

Figure 4: Executing Q1, Q2, and Q3 in AJoin between time T_{4C} and T_{1D}. For simplicity, the attributes that are not used by the queries are indicated as '...'.

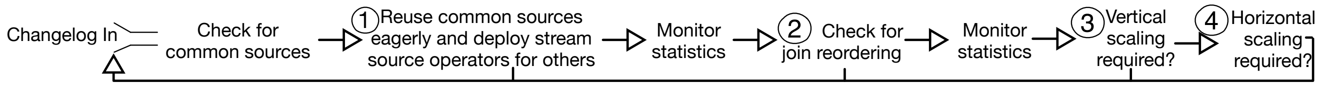

Changelog In
Check for common sources
① Reuse common sources eagerly and deploy stream source operators for others
Monitor statistics
② Check for join reordering
Monitor statistics
③ Vertical scaling required?
④ Horizontal scaling required?

Figure 5: Optimization process

resulting bucket of Q3. The bucket keeps indexes of matched tuples from stream sources W, C, and R.

All join predicates in Q3 use the same join attribute (usrID). In this case, the late materialization can be easily leveraged with built-in indexes (Figures 4d and 4e). However, if join attributes are different (e.g. in Q2), then repartitioning is required after the first join. AJoin benefits from late materialization also in this scenario. To compute Q3, AJoin computes the result of the upstream join operator (Figure 4b). Then, the resulting bucket ($V \bowtie_{V.vID=W.vID} W$) is reindexed w.r.t. W.usrID (Figure 4c, ①). Note that reindexing is related to the tuples belonging to W because only these tuples contain attribute usrID. Instead of materializing the intermediate result fully and iterating through it ($V \bowtie_{V.vID=W.vID} W$) and reindexing, AJoin avoids full materialization and only iterates over the tuples belonging to W: (1) every tuple $tp \in$ W is reindexed w.r.t. W.usrID; (2) a list of its matched tuples from V is retrieved (get list with index ID=*tp*.vID); (3) the pointer of the resulting list is appended to *tp*. When *tp* is eliminated in the downstream join operator, all its matched tuples from V are also automatically eliminated. For example, tuples with usrID=3 in Figure 4c ①, are eliminated when joining with C (Figure 4d). In this case, the pointers are also eliminated without iterating through them.

4. OPTIMIZER

In this section, we discuss the query optimization process in AJoin. Figure 5 shows the optimization phases of AJoin. After a changelog ingestion, the optimizer eagerly shares the newly created query with the running queries (①). For example, Q2 is deployed at time T_{2C} (Figure 1). Then, the optimizer searches common subqueries among running queries (Q1 in this case), without considering the selection predicate and the cost. In this case, the optimizer deploys Q2 as (V⋈W)⋈C to reuse the existing stream sources and the join operator. In the following phases, the optimizer performs a cost based analysis and reoptimizes the QEP, when necessary. If the optimizer cannot find common subqueries, it will check for common sources to benefit from scan sharing. The optimizer restarts the optimization process, if a new changelog has arrived.

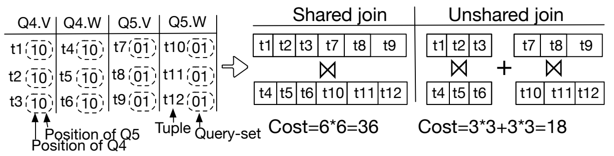


Figure 6: Cost of shared and separate join execution for Q4 and Q5. `Q.S` means the stream `S` of the query `Q`.

Below, we explain each phase of the optimization separately and describe when the optimizer decides to trigger each of them.

Query Grouping. Consider Q4 and Q5 in Figure 7a. These queries do not share data because of their selection predicates. Figure 6 shows an example scenario for performing shared (Q4 and Q5 together) and separate join (Q4 and Q5 separately). Previous solutions, such as AStream [35], share data and computation aggressively. However, this might lead to suboptimal QEP. For example, in Figure 6, the cost of shared query execution is higher than executing queries separately. The reason is that both Q4.V, Q5.V and Q4.W, Q5.W do not share enough data tuples to benefit from shared execution. Throughout the paper, we denote the stream source `S` of query `Q` as `Q.S` and stream partition p_i of query `Q` as `Q.`p_i.

To avoid the drawback of aggressive sharing, we arrange queries in groups. Queries that are likely to filter (or not filter) a given set of stream tuples are arranged in one `query group`. For example, after successful grouping, Q4 and Q5 in Figure 6 would reside in different groups. Let `t1`, `t2`, `t3`, and `t4` be tuples with query-sets `(100100)`, `(101100)`, `(100100)`, and `(100000)`, respectively, and Q1-Q6 be queries with selection operators. Q1 and Q4 share 3 tuples `(t1, t2, t3)` out of 4. Also, Q2, Q3, Q5, and Q6 do not share 3 tuples `(t1, t3, t4)` out of 4. Finding the optimal query groups is an NP-Hard problem, as it can be reduced to the euclidean sum-of-squares clustering problem [2].

`Crd` is a function that calculates the cardinality of possibly intersecting sets. We use set union operation to calculate the cardinality. For example, for 3 sets (A, B, C) `Crd` function is shown in Equation 1. Equation 2 shows the cost function. b_{i1}

and b_{i2} are boolean variables showing if indexing is required on stream S1 and S2, respectively. AJoin performs indexing when stream S is the leaf node of the QEP (source operator) or when repartitioning is performed. b_m is also a boolean variable indicating if full materialization is required. AJoin performs full materialization only at the sink operator.

Figure 7a shows our approach to calculate query groups. First, we compare the cost of sharing stream sources between two queries and executing them separately. If the cost of the former is less than the latter, we put the two queries into the same query group. Once we find query groups consisting of two queries, we eagerly check other queries, which are not part of any group, to include into the group. The only condition (to be accepted to the group) is that the cost of executing the new query and the queries inside the group in a shared manner must be less than executing them separately (e.g., Figure 6). Query grouping is performed periodically during the query execution. When join reordering is triggered, it utilizes recent query groups.

$$|A|+|B|+|C|-|A\cap B|-|B\cap C|-|A\cap C|+|A\cap B\cap C| \tag{1}$$

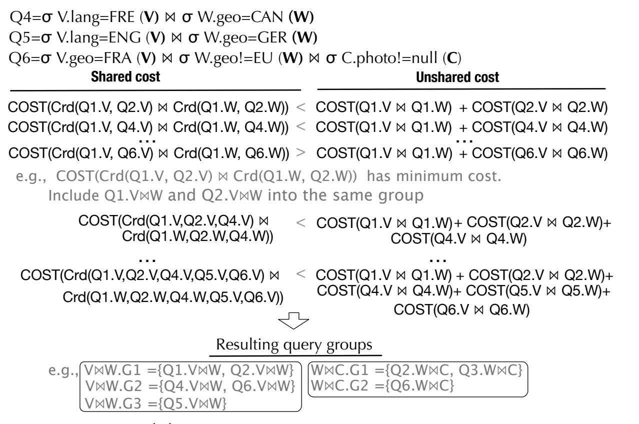
$$\begin{aligned}\mathrm{COST}(S1\bowtie S2) = {} & \underbrace{b_{i1}*\mathrm{Crd}(S1)}_{\text{Indexing S1}}+\underbrace{b_{i2}*\mathrm{Crd}(S2)}_{\text{Indexing S2}}+ \\ & \underbrace{\mathrm{Min}(\mathrm{DistKey}_{S1},\mathrm{DistKey}_{S2})}_{\text{Index set intersection}}+\underbrace{b_m*\mathrm{Crd}(S1\bowtie S2)}_{\text{Full materialization}}\end{aligned} \tag{2}$$

Join Reordering. After discovering query groups, the optimizer performs iterative QEP optimization. We enhance an iterative dynamic programming technique [37] and adapt it to ad-hoc stream query workloads. Our approach combines dynamic programming with iterative heuristics. In each iteration, the optimizer *i*) calculates the shared cost of subqueries and *ii*) selects a subplan based on the cost. The shared cost is the cardinality of a particular subquery divided by the number of QEPs, sharing the subquery.

Figure 7b shows an example scenario for iterative QEP optimization. Assume that Q4-Q6, which are shown in Figure 7a, are also added to the existing queries (Q1-Q3). In the first iteration, the optimizer calculates the shared cost of 2-way joins. For example, Q1.V⋈W can be shared between Q1 and Q2 because Q1 and Q2 are in the same group (Figure 7a). Also, the cost of Q1.V⋈W differs when exploiting all sharing opportunities (MaxShared) and executing it separately (MinShared). After the first iteration, the optimizer selects subplans with minimum costs. Then, the optimizer substitutes the selected subqueries with T1 and T2. If the cost is shared with other QEPs (e.g., Q1.V⋈W is shared between Q1 and Q2), then the optimizer assigns the shared cost to all other related queries.

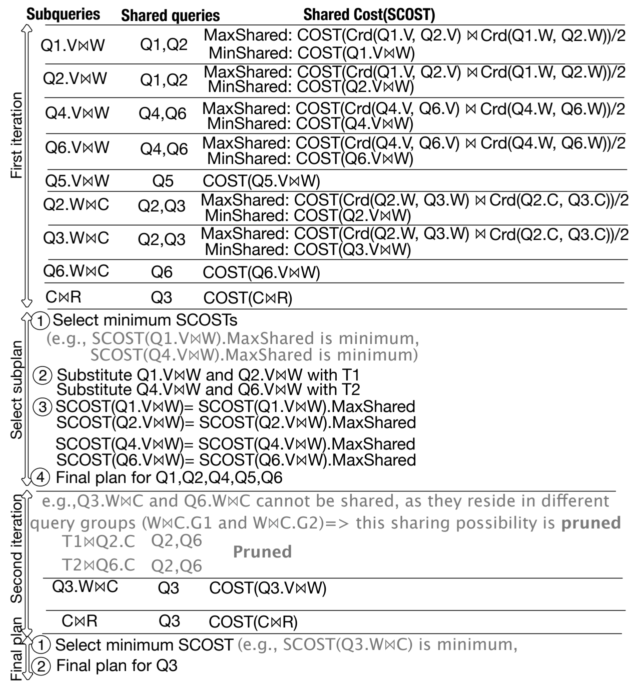
The second iteration is similar to the first one. Note that T1⋈Q2.C cannot be shared with Q6 because Q6.V⋈W and Q2.V⋈W reside in different query groups. So, the optimizer prunes this possibility. Also, Q3.W⋈C is no longer shared with Q2 because in the first iteration the optimizer assigned (V⋈W)⋈C to Q2.

Computing the optimal QEP for multiple queries is an NP-Hard problem [24, 31]. For ad-hoc queries, this is particularly challenging, since queries are created and deleted in an ad-hoc manner. The optimizer must therefore support incremental computation. Assume that Q4 in Figure 7 is deleted, and Q7= σ_{sp1}(W)⋈σ_{sp2}(C) is created, where sp1 and sp2 are selection predicates. At compile-time, the optimizer shares Q7 aggressively (without considering the selection predicates) with existing queries. In this case, the optimizer shares Q7 with Q3.W⋈C. After collecting statistics, the optimizer tries to locate Q7 in one of W⋈C groups (e.g., Figure 7a). If including Q7 is not beneficial to any query group (shared execution is more costly than executing queries in the group and the added query separately), the optimizer creates a new group for Q7. Assume that Q7 is placed in W⋈C.G2 (Figure 7a). In this case, only the execution of Q4 and Q6 might be affected.



Q4=σ V.lang=FRE (V) ⋈ σ W.geo=CAN (W)
Q5=σ V.lang=ENG (V) ⋈ σ W.geo=GER (W)
Q6=σ V.geo=FRA (V) ⋈ σ W.geo!=EU (W) ⋈ σ C.photo!=null (C)
Shared cost
Unshared cost
COST(Crd(Q1.V, Q2.V) ⋈ Crd(Q1.W, Q2.W)) < COST(Q1.V ⋈ Q1.W) + COST(Q2.V ⋈ Q2.W)
COST(Crd(Q1.V, Q4.V) ⋈ Crd(Q1.W, Q4.W)) < COST(Q1.V ⋈ Q1.W) + COST(Q4.V ⋈ Q4.W)
...
COST(Crd(Q1.V, Q6.V) ⋈ Crd(Q1.W, Q6.W)) > COST(Q1.V ⋈ Q1.W) + COST(Q6.V ⋈ Q6.W)
e.g., COST(Crd(Q1.V, Q2.V) ⋈ Crd(Q1.W, Q2.W)) has minimum cost.
Include Q1.V⋈W and Q2.V⋈W into the same group
COST(Crd(Q1.V,Q2.V,Q4.V) ⋈ Crd(Q1.W,Q2.W,Q4.W)) < COST(Q1.V ⋈ Q1.W)+ COST(Q2.V ⋈ Q2.W)+ COST(Q4.V ⋈ Q4.W)
...
COST(Crd(Q1.V,Q2.V,Q4.V,Q5.V,Q6.V) ⋈ Crd(Q1.W,Q2.W,Q4.W,Q5.V,Q6.V)) < COST(Q1.V ⋈ Q1.W) + COST(Q2.V ⋈ Q2.W)+ COST(Q4.V ⋈ Q4.W)+ COST(Q5.V ⋈ Q5.W)+ COST(Q6.V ⋈ Q6.W)
Resulting query groups
e.g., V⋈W.G1 ={Q1.V⋈W, Q2.V⋈W}
V⋈W.G2 ={Q4.V⋈W, Q6.V⋈W}
V⋈W.G3 ={Q5.V⋈W}
W⋈C.G1 ={Q2.W⋈C, Q3.W⋈C}
W⋈C.G2 ={Q6.W⋈C}

(a) Calculation of query groups



Subqueries | Shared queries | Shared Cost(SCOST)
First iteration
Q1.V⋈W | Q1,Q2 | MaxShared: COST(Crd(Q1.V, Q2.V) ⋈ Crd(Q1.W, Q2.W))/2 MinShared: COST(Q1.V⋈W)
Q2.V⋈W | Q1,Q2 | MaxShared: COST(Crd(Q1.V, Q2.V) ⋈ Crd(Q1.W, Q2.W))/2 MinShared: COST(Q2.V⋈W)
Q4.V⋈W | Q4,Q6 | MaxShared: COST(Crd(Q4.V, Q6.V) ⋈ Crd(Q4.W, Q6.W))/2 MinShared: COST(Q4.V⋈W)
Q6.V⋈W | Q4,Q6 | MaxShared: COST(Crd(Q4.V, Q6.V) ⋈ Crd(Q4.W, Q6.W))/2 MinShared: COST(Q6.V⋈W)
Q5.V⋈W | Q5 | COST(Q5.V⋈W)
Q2.W⋈C | Q2,Q3 | MaxShared: COST(Crd(Q2.W, Q3.W) ⋈ Crd(Q2.C, Q3.C))/2 MinShared: COST(Q2.V⋈W)
Q3.W⋈C | Q2,Q3 | MaxShared: COST(Crd(Q2.W, Q3.W) ⋈ Crd(Q2.C, Q3.C))/2 MinShared: COST(Q3.V⋈W)
Q6.W⋈C | Q6 | COST(Q6.V⋈W)
C⋈R | Q3 | COST(C⋈R)
Select subplan
1 Select minimum SCOSTs
(e.g., SCOST(Q1.V⋈W).MaxShared is minimum, SCOST(Q4.V⋈W).MaxShared is minimum)
2 Substitute Q1.V⋈W and Q2.V⋈W with T1
Substitute Q4.V⋈W and Q6.V⋈W with T2
3 SCOST(Q1.V⋈W)= SCOST(Q1.V⋈W).MaxShared
SCOST(Q2.V⋈W)= SCOST(Q2.V⋈W).MaxShared
SCOST(Q4.V⋈W)= SCOST(Q4.V⋈W).MaxShared
SCOST(Q6.V⋈W)= SCOST(Q6.V⋈W).MaxShared
4 Final plan for Q1,Q2,Q4,Q5,Q6
Second iteration
e.g.,Q3.W⋈C and Q6.W⋈C cannot be shared, as they reside in different query groups (W⋈C.G1 and W⋈C.G2)=> this sharing possibility is pruned
T1⋈Q2.C Q2,Q6
T2⋈Q6.C Q2,Q6
Pruned
Q3.W⋈C | Q3 | COST(Q3.V⋈W)
C⋈R | Q3 | COST(C⋈R)
Final plan
1 Select minimum SCOST (e.g., SCOST(Q3.W⋈C) is minimum,
2 Final plan for Q3

(b) Join reordering

Figure 7: Optimization example. The optimization is performed between time T_{3C} and T_{1D}. Assume that Q4-Q6 (Figure 7a) are also being executed at the time of optimization. In the figure, `Crd` refers to the cardinality function, and `COST` refers to the cost function in Equation 2.

In other words, the optimizer does not need to recompute the whole plan, but part of the QEP. Also, the optimizer does not recompute query groups from scratch but reuses existing ones.

The cost of incremental computation is high and may result in an suboptimal plan. Therefore, we use a threshold when to trigger a full optimization. If the number of created and deleted queries exceeds 50% of all queries in the system, the optimizer computes a new plan (including the query groups) holistically instead of incrementally. We have determined this threshold experimentally, as it gives a good compromise between dynamicity and optimization cost. Computing the threshold in a deterministic way, on the other hand, is out of the scope of this paper. The decision to reorder joins (② in Figure 5) is triggered by the cost-based optimizer using techniques explained above.

There are two main requirements behind our cost computation. The first requirement is that the cost function should include the computation semantics of our pipeline-parallelized join operator. As we can see from Equation 2, `COST` consists of the cost of the source operator (indexing S1 and S2), the cost of join operator (index set intersection), and the cost of sink operator (full materialization). The second requirement is that the cost computation should include sharing information. We achieve this requirement by dividing `COST` by the number of shared queries (Figure 7b, `MaxShared`). We select this cost computation semantics because it complies with our requirements, and it is simple.

Vertical and Horizontal Scaling. AJoin uses consistent hashing for assigning tuples to partitions. The partitioning function `PF` maps each tuple with key `k` to a circular hash space of key-groups: `PF(k)=(Hash(k) mod |P|)`, where `|P|` is the number of parallel partitions. At compile-time, partitions are distributed evenly among nodes.

The optimizer performs vertical scaling (③ in Figure 5), if the latency of tuples residing in specific partitions is high, and there are resources available on nodes, in which overloaded partitions are located. The optimizer checks for scaling up first, because scaling up is less costly than scaling out. Note that when scaling up, the partitioning function and the partition range assigned to each node remains the same. Instead, the number of threads operating on specific partitions are increased. When new operators are deployed, and existing operators exhibit low resource-utilization, the optimizer decides to scale down the existing operators.

The optimizer checks for horizontal scaling (④ in Figure 5) when new and potentially non-shared queries are created. Also, the optimizer decides to scale out if CPU or memory is a bottleneck. When the optimizer detects a latency skew, and there are no available resources to scale up, it triggers scaling out. In this case, the optimizer distributes the partition range, which is overloaded, among new nodes added to the cluster. Therefore, at runtime, the partition range might not be distributed evenly among all nodes.

5. IMPLEMENTATION DETAILS

Bucketing. Bucketing is performed in the source operator. The source operator is the first operator in the AJoin QEP. Each index entry inside a bucket points to a list of tuples with the common key. If there are multiple indexes, pointers are used to reference stream tuples. The main intuition is that buckets are read-only; so, sharing the stream tuples between multiple concurrent queries (with different indexes) is safe. Each source operator instance assigns a unique ID to the generated bucket; however, bucket IDs are not unique between different partitions. The bucket ID is an integer indicating the generation time of the bucket.

Join. Let L_{in} and L_{out} be lists inside a join operator storing buckets from inner and outer stream sources, respectively. When the join operator receives buckets, b_{in} from the inner and b_{out} from the outer stream source, it *i*) joins all the buckets inside L_{out} with b_{in}, all the buckets inside L_{in} with b_{out}, and combines the two results in one output bucket, *ii*) emits the output bucket, and *iii*) removes unnecessary buckets from L_{in} and L_{out}.

The join operator handles join queries with different join predicates and window constraints. The operator receives query changelogs from upstream operators and updates its query metadata. Figure 8 shows an example scenario for incremental ad-hoc join query computation. At time T1 Q1 is initiated. At time T2 the join operator receives the query changelog indicating the creation of Q2. Also, first buckets from both streams are joined and emitted. Since the joined buckets are no longer needed, they are deleted. Q1 and Q2 have the same join predicates but different window length. Therefore, $3\bowtie 3$ is shared between Q1 and Q2, but $2\bowtie 3$ and $3\bowtie 2$ are associated with only Q2. Since buckets support multiple indexes, the join operator can share join queries with different join predicates. The rest of the example follows a similar pattern.

The join operation between two buckets is performed as follows. Firstly, queries with similar stream sources and join predicates

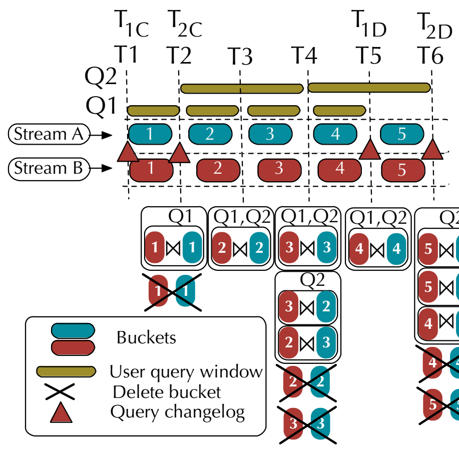


Figure 8: Ad-hoc join example. The join operation is performed between T_{1C} and T_{2D} time interval.

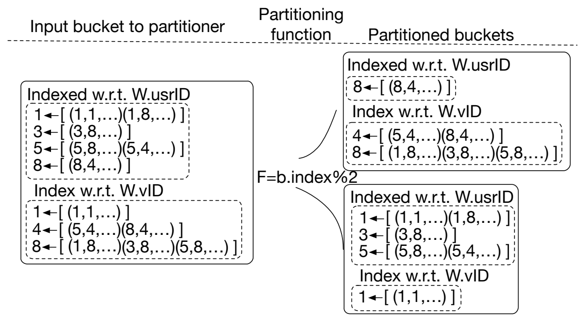


Figure 9: Example partitioning of the bucket described in Figure 4e

are grouped. We perform scan sharing for the queries in the same group. The join operation is a set intersection of indexes, as we use a grace join [36] for streaming scenarios. AJoin supports out-of-order stream tuples if they reside within the same bucket.

Partitioning. The partitioner is an operator that partitions buckets among downstream operator instances. This operator accepts and outputs buckets. Given an input bucket, the partitioner traverses over existing indexes of the bucket. It maps each index entry and corresponding stream tuples to one output bucket. In this way, the partitioner traverses only indexes instead of all stream tuples.

The partitioning strategy of AJoin with multiple queries is similar to one with a single query. If queries have the same join predicate, the partitioner avoids copying data completely. That is, each index entry and its corresponding tuples are mapped to only one downstream operator instance. If queries possess different join predicates, AJoin is able to avoid data copy partially. For example, in Figure 9 the input bucket is partitioned into two downstream operator instances. Note that tuples that are partitioned to the same node w.r.t. both partitioning attributes (e.g. (1,1,. . .),(8,4,. . .)) are serialized and deserialized only once, without data copy.

Materialization. The sink operator performs full materialization. Basically, it traverses all indexes in a bucket, performs the cross-product of tuples with the same key, constructs new tuples, and pushes them to output channels.

Exactly-Once Semantics. AJoin guarantees exact-once semantics, meaning every stream tuple is only processed once, even under failures. AJoin inherits built-in exactly-once semantics of Apache Flink [14]. Whether the unit of data is a stream tuple or a bucket, under the hood the fault tolerance semantics is the same.

Optimizer. We implement the AJoin optimizer as part of the Flink's optimizer. Flink v1.7.2 lacks a run-time optimizer. There-

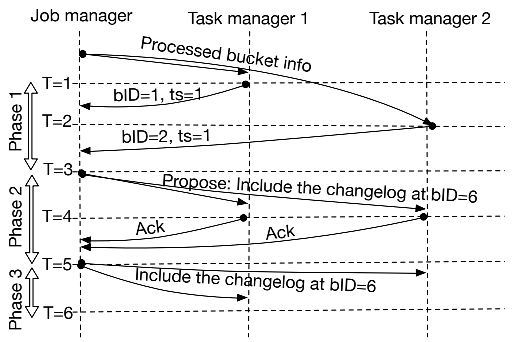


Figure 10: 3-phase atomic protocol

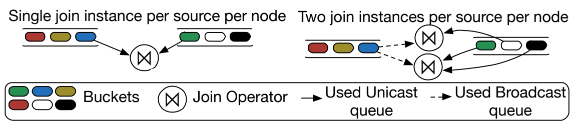

Single join instance per source per node
Two join instances per source per node
Buckets
Join Operator
Used Unicast queue
Used Broadcast queue

Figure 11: Scale up operation

fore, the AJoin optimizer can be easily integrated into Flink's optimizer. We also integrate the AJoin optimizer with Flink's compile-time optimization. The compile-time optimization process consists of three main phases. In the first phase, AJoin performs logical query optimization. Then, Flink's optimizer receives the resulting plan, applies internal optimizations, and generates the physical QEP. Afterwards, the AJoin optimizer analyzes the resulting physical QEP. For n-way join queries, the AJoin optimizer inspects if each node contains at least one operator instance of all join operators in the query. For example, the physical QEP of (A⋈B)⋈C should contain at least one instance of the upstream (A⋈B) and the downstream join operators ((...)⋈C) in each node. Also, the optimizer checks if all join operator instances are evenly distributed among the cluster nodes. It is acceptable if some nodes have free (idle) task slots. The free task slots provide flexibility for scaling up during the run-time. If there are join operators that share the same join partitioning attribute, the optimizer schedules them in the same task slot and notifies Flink to share the task slot between the two join operator instances. For example, in a query like $(A\bowtie_{A.a=B.b}B)\bowtie_{B.b=C.c}C$, the instances of the upstream join operator $(A\bowtie_{A.a=B.b}B)$ share the same task slot with the instances of the downstream join operator $((...)\bowtie_{B.b=C.c}C)$. The reason is to ensure the data locality, as the resulting stream of the upstream join operator is already partitioned w.r.t. attribute B.b. The optimizer performs the necessary changes in the physical QEP generated by Flink (second phase) to perform the optimizations listed above.

6. RUN-TIME QEP CHANGES

It is widely acknowledged that streaming workloads are unpredictable [8]. Supporting ad-hoc queries for streaming scenarios leads to more dynamic workloads. Therefore, AJoin supports several run-time operations updating the QEP on-the-fly.

Consistency Protocols. AJoin features two consistency protocols: atomic and non-atomic. The atomic protocol is a three-phase protocol. Figure 10 shows an example scenario for this protocol. In the first phase, the job manager requests `bID`, the current bucket ID, and `ts`, the current time in the task manager, from all task managers. In the second phase, the job manager proposes the task managers to ingest the changelog after the bucket with `bID=6`. If the job manager receives `ack` from all task managers, it sends a confirmation message to the task managers to ingest the changelog. In the non-atomic protocol, on the other hand, the job manager sends the changelog without any coordination with task managers.

Vertical Scaling. AJoin features two buffering queues between operators: a `broadcast queue` and a `unicast queue`. Let S be a set of subscribers to a queue. In the broadcast queue, the head element of the queue is removed if all subscribers in S pull the element. Any subscriber $s_i \in S$ can pull elements up to the last element inside the queue. Afterwards, the subscriber thread is put to sleep mode and awakened once a new element is pushed into the broadcast queue. In a unicast queue, on the other hand, the head element of the queue is removed if one subscriber pulls it. The consequent subscriber pulls the next element in the queue.

The join operation is distributive over union (A⋈(B∪C)= A⋈B ∪ A⋈C). We use this feature and the two queues to scale up and down efficiently. Each join operator subscribes to two upstream queues: one broadcast and one unicast queue. When a new join operator is initiated in the same worker node (scale up), it also subscribes to the same input channels. For example, in Figure 11, there are two queues. If we increase the number of join instances, then both instances would get the same buckets from the broadcast queue but different buckets from the unicast queue. As a result, the same bucket is joined with different buckets in parallel.

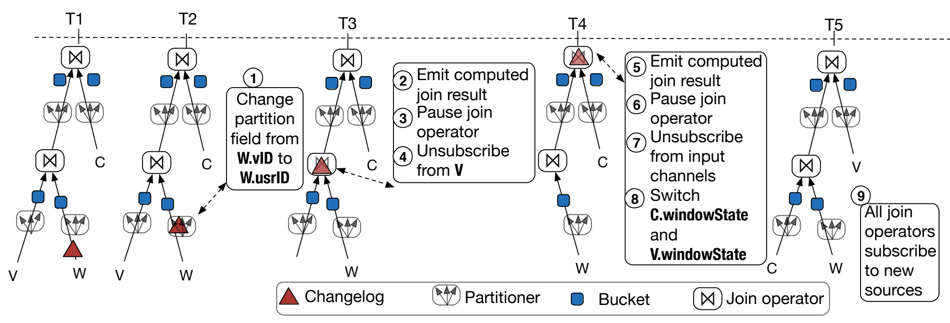
We use the non-atomic protocol for the vertical scaling. Let S1 and S2 be the two joined streams(S1⋈S2) and P=$\{p_1,p_2,...,p_n\}$ be parallel partitions in which the join operation is performed. Vertical scaling in AJoin is performed on a partition of a stream (i.e., a vertical scaling affects only one partition). So, we show that the scaled partition produces correct results. Assume that `k` new task managers are created at partition p_i, which output join results to p_i^1, p_i^2, ..., p_i^k. Since $\bigcup_{j=1}^{k} p_i^j$= S1.p_i⋈S2.p_i (distributivity over union), the result of vertical scaling is correct. Since there is no synchronization among partitions, and since each vertically scaled partition is guaranteed to produce correct results, vertical scaling is performed in an asynchronous manner.

Horizontal Scaling. AJoin scales horizontally in two cases: when a new query is created (or deleted), and when an existing set of queries needs to scale out (or scale in). We refer to the first case as query pipelining. We assume that created or deleted queries share a subquery with running queries. Otherwise, the scaling is straightforward - adding new resources and starting a new job.

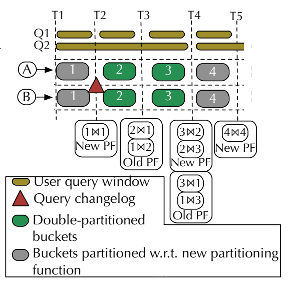
Query pipelining consists of three main steps. Let the existing query topology be `E` and the pipelined query topology be `P`. In the first step, the job manager sends a changelog to the task managers of `E`. Upon receiving the changelog, the task managers switch sink operators of `E` to the pause state and ack to the job manager. In the second step, the job manager arranges the input and output channels of the operators deployed inside the task managers, such that the input channels of `P` are piped to the output channels of `E`. In the third step, the job manager resumes the paused operators. If the changelog contains deleted queries, the deletion of the queries is performed similarly. The job manager pauses upstream operators of deleted stream topologies. Then, the job manager pipelines a sink operator to the paused operators. Lastly, the job manager resumes the paused operators.

Query pipelining is performed via the non-atomic protocol. Thus, all the partitions of the pipelined query are not guaranteed to start (or stop) processing at the same time. However, modern SPEs [54], [48], [15] also connect to data sources, such as Apache Kafka [1], in an asynchronous manner. Also, when a stream query in the modern SPEs is stopped, there is no guarantee that all sink operators stop at the same time.

Scaling out and in can be generalized to changing the partitioning function and computation resources. We explain the partitioning strategy in Section 4. Assume that AJoin scales out by N new nodes, and each node is assigned to execute P′ partitions. Then, the new partitioning function becomes `PF′(k)=(Hash(k) mod (|P|+|P′*N|))`. Also, each new node is assigned a partition range. The partition range is determined via further splitting the overloaded partitions. For example, if a partition with hashed key


T1
T2
T3
T4
T5
V
W
C
1 Change partition field from W.vID to W.usrID
2 Emit computed join result
3 Pause join operator
4 Unsubscribe from V
5 Emit computed join result
6 Pause join operator
7 Unsubscribe from input channels
8 Switch C.windowState and V.windowState
9 All join operators subscribe to new sources
Changelog
Partitioner
Bucket
Join operator

(a) Join reordering



(b) The operation that shows the change in the partitioning function (PF)

Figure 12: Run-time QEP changes: Join reordering (left) and partitioning function change (right)

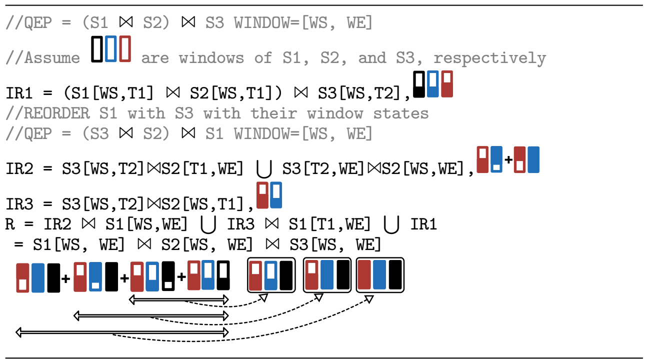
range [0,10] is overloaded, and one new partition is initiated in the new node, then the hashed key ranges of the two partitions become [0,5] and (5,10]. The similar approach applies for scaling in.

The change of the partitioning function is completed in three steps. Assume that the partitioning function of a join operator is modified. There are multiple queries using the join operator with different window configurations. In the first step, the job manager retrieves the biggest window size, say `BW`. In the second step, the job manager sends a partition-change changelog via the atomic protocol. Once the partitioner receives this marker, it starts double partitioning, meaning partitioned buckets contain data both w.r.t the old and new partitioning function. The partitioner performs double-partitioning at most `BW` time, then partitions only w.r.t. the recent partitioning function. In the third step, new task managers are launched (scale out) or stopped (scale in).

Figure 12b shows an example scenario for a partitioning function change. First buckets from the stream A and B have single partitioning info. At time T2 the partition-change changelog arrives at the join operator. So, the tuples arriving before T2 no longer have the new or latest partitioning schema. At time T3, the second and first buckets are joined w.r.t. the old partitioned data. At time T4, the third and second buckets are joined w.r.t. the new partitioned data; however, the third and first buckets are joined w.r.t. the old partitioned data. Starting from T4, the partitioner stops double-partitioning and switches to the new partitioning function.

We use the atomic protocol when changing the partitioning function. Changing the partitioning function possibly affects all partitions. In order to guarantee the correctness of results, there are two main requirements: *i*) all partition operators must change the partitioning function at the same time and *ii*) downstream operators must ensure the consistency between the data partitioned w.r.t. new and old partitioning functions. To achieve the first requirement, we use the atomic 3-phase protocol. To achieve the second requirement, we use a custom join strategy in which we avoid to join old-partitioned and new-partitioned data. Instead, we perform double-partitioning and ensure that any joined two tuples are partitioned w.r.t. the same partitioning function. We apply the similar technique, mentioned above, when query groups are changed.

Join reordering. Suppose at time T_{1D}, the optimizer triggers to change the QEP of Q2 from $(V \bowtie_{V.vID=W.vID} W) \bowtie_{W.usrID=C.usrID} C$ to $V \bowtie_{V.vID=W.vID} (W \bowtie_{W.usrID=C.usrID} C)$. Figure 12a shows the main idea behind reordering joins. At time T1, the job manager pushes the changelog marker via the non-atomic protocol. The marker passes through the partitioner at time T2. The marker informs the partitioner to partition based on W.usrID, instead of W.vID. At time T3, the changelog marker arrives at the first join operator. Having received the changelog, the join operator emits the join result, if any, and acks to the job manager. The job manager then *i*) pauses the join operator and *ii*) unsubscribes it from stream V. At time T4, the marker



```
//QEP = (S1 ⋈ S2) ⋈ S3 WINDOW=[WS, WE]
//Assume ▯▯▯ are windows of S1, S2, and S3, respectively
IR1 = (S1[WS,T1] ⋈ S2[WS,T1]) ⋈ S3[WS,T2],
//REORDER S1 with S3 with their window states
//QEP = (S3 ⋈ S2) ⋈ S1 WINDOW=[WS, WE]
IR2 = S3[WS,T2]⋈S2[T1,WE] ⋃ S3[T2,WE]⋈S2[WS,WE],
IR3 = S3[WS,T2]⋈S2[WS,T1],
R = IR2 ⋈ S1[WS,WE] ⋃ IR3 ⋈ S1[T1,WE] ⋃ IR1
 = S1[WS, WE] ⋈ S2[WS, WE] ⋈ S3[WS, WE]
```

Figure 13: Formal definition of join reordering. The black, blue, and red boxes represent the windows of S1, S2, and S3. Filled boxes mean that the respective portion of the boxes are joined.

arrives at the second join operator. Similarly, the second join operator emits the join result, if any. It informs the job manager about the successful emission of results. The job manager pauses the operator and unsubscribes it from input channels. Afterward, the second join operator switches its state with the upstream join operator. Finally, the job manager subscribes both join operators to modified input channels and resumes computation.

We use the non-atomic protocol for join reordering. Join reordering is performed in all partitions, independently. Assume that S1, S2, and S3 are streams, W denotes window length, WS and WE are window start and end timestamps, and T1 and T2 are timestamps in which the changelog arrives to the first and to the second window. Figure 13 shows the formal definition of the join reordering. When the changelog arrives at the first join operator, the intermediate join result (IR1 in Figure 13) is computed and emitted. At this point, AJoin switches the window states of S1 and S3. Then, unjoined parts of S3 and S2 are joined (IR2 in Figure 13). Although IR3 is included in IR1, IR3 is joined with S1[T1,WE] in the final phase; therefore, the result is not a duplication. Finally, AJoin combines all intermediate results to the final output (R in Figure 13), which is correct and does not include any duplicated data.

7. EXPERIMENTS

Experimental design. Our benchmark framework consists of a distributed driver and four systems under test (SUT): AJoin, AStream, Spark Streaming v2.4.4, and Apache Flink v1.7.2. The driver maintains two queues: one for stream tuples and one for user requests (query creation or deletion). The tuple queue receives data from tuple generators inside the driver. The driver generates tuples at maximum sustainable throughput [34]. A SUT pulls records from the data queue with the highest throughput it can process. So, the longer the tuple stays in the queue, the higher its event-time latency. The working principle of the user request queue is similar to the tuple queue.

If a SUT exhibits backpressure, it automatically reduces the pull rate. Contrary to data tuples, user requests are periodically pushed to the client module of the SUT. The SUT acks to the driver, after receiving the user request. If the `ack timeout` is exceeded or every subsequent ack duration keeps increasing, then the SUT cannot sustain the given query throughput. Similarly, if there is an infinite backpressure, then the SUT cannot sustain the given workload. In these cases, the driver terminates the experiment and tests the SUT again with a lower query and data throughput.

Metrics. `Query deployment latency` is the duration between a query create or delete request and the actual query create or delete time at a SUT. `Overall data throughput` is the sum of data throughputs of all running queries. `Query similarity` is the similarity between the generated query and the pattern query.

Data generation. Equation 3 shows the calculation of the query similarity. To evaluate the similarity between a query Q (e.g. A$\bowtie_{A.a=B.a}$B) and the pattern query PQ (e.g. A$\bowtie_{A.a=B.b}$B), we *i*) find the number of the common sources (ComS) between Q and PQ (A and B), *ii*) find the number of the common sources with common join attributes (ComSJA) between Q and PQ (only A.a), and *iii*) divide the multiplication of the two with square of all sources (AllS) in PQ (i.e., $\frac{2*1}{2^2}$).

$$\text{Similarity(ComS,ComSJA,AllS)}=\frac{\text{ComS} * \text{ComSJA}}{(\text{AllS})^2} \qquad (3)$$

To generate query Q with n% similarity with PQ, we apply the following approach. Assume that n is 75% and PQ is A$\bowtie_{A.a=B.b1}$B$\bowtie_{B.b2=C.c}$C.

`(Step 1.)` We randomly select ComS, which is between 1 and AllS (e.g., ComS=2).

`(Step 2.)` Given ComS=2 and AllS=3, we calculate ComSJA from Equation 3. If a stream is a source stream, it is partitioned by the join attribute of the downstream join operator. If a stream is an intermediate result, two join operators affect the sharing possibility of this stream: the upstream join operators (how the stream was partitioned) and the downstream join operator (how the stream will be partitioned). Therefore, each stream can be affected by maximum of two partitioning attributes. If ComSJA number of join attributes cannot be used with ComS number of sources (e.g., 1 stream source can be affected by maximum of 2 join attributes), then we increase ComS by one and repeat this step.

`(Step 3.)` We select ComS number of random stream sources from PQ, such that these stream sources are joined with each other with a join predicate and not via cross-product (e.g., A$\times$C is not acceptable). Similarly, we select random ComSJA number of join attributes from the selected sources.

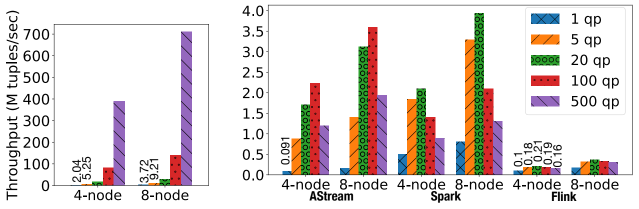
Figure 14 shows a query template for join query generation. For each stream source, we add a selection predicate. After filtering, we join stream sources based on randomly selected join attributes. All attributes of the data tuples can be used as a join attribute. The window length of the generated query is either 1, 2, or 3 seconds. We perform window duration, selection predicate, and join predicate assignment in a uniform manner.

Each data tuple features 6 attributes. Each attribute of a tuple is generated as a random uniform variable between 0 and ATR_MAX. We set different seed value for data generation per each stream source. We use `Java Random` class for uniform data generation. Throughout our experiments, we set ATR_MAX to be 500. The data generation speed for all stream sources is equal.

Workload. The first workload scenario (SC1) is applicable when a user activity is higher on specific time periods. Also, in this workload scenario, users execute long-running queries. The second workload scenario (SC2) is relevant for fluctuating

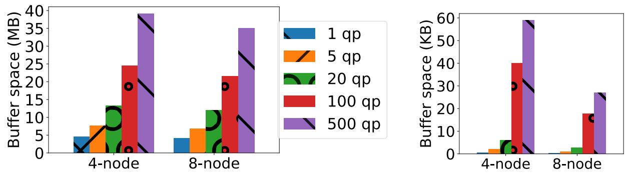
```
SELECT *
FROM S1,S2,...,Sn WINDOW=[1|2|3|] sec
WHERE S1.[JA1] = S2.[JA2] AND S2.[JA3] = S3.[JA4] ... AND
      Sn-1.[JAj-1] = Sn.[JAj] AND
      S1.[SA1][=|>|<|>=|<=] [FV1] AND S2.[SA2] [=|>|<|>=|<=] [FV2]
      ... AND Sn.[SAn] [=|>|<|>=|<=] [FVn]
```

Figure 14: Query template used in our experiments. `Sn[i]` means ith attribute of stream n. `JA`$_i$ (join attribute) and `SA`$_i$ (selection attribute) $(0 \leq$ `JA`$_i$,`SA`$_i$$<|6|)$ are random variables (e.g., `Sn[SA`$_i$`]` is `SA`$_i$th attribute of stream `Sn`). `FV`$_i$ (filtering value) is a randomly assigned value used to filter streams.



(a) AJoin **(b) AStream, Spark, and Flink**

Figure 15: Overall data throughput of AJoin, AStream, Spark, and Flink with 1, 5, 20, 100, and 500 parallel queries on 4- and 8-node cluster configurations. qp at the legend means query parallelism.



(a) Tuple buffer **(b) Index buffer**

Figure 16: Buffer space used for tuples and indexes inside a 1-second bucket

workloads. Modern SPEs cannot execute ad-hoc stream queries. Therefore, there is no industrial workload for ad-hoc stream query processing. Therefore, we use the workload used in AStream [35], which is similar to cloud workloads [9, 3, 50, 47, 40, 30]. Nevertheless, the design of AJoin is generic and not specific to the workloads explained above.

Setup. We conduct experiments in 4- and 8-node cluster configurations. Each node features 16-core Intel Xeon CPU (E5620 2.40GHz) and 48 GB main memory. We configure the batch interval of queries (in the client module) to be 1 second and ack timeout is 15 seconds, as these configurations are the most suitable for our workloads. The latency threshold for scaling up and out is 5 seconds. The threshold is derived from the latency-aware elastic scaling strategy for SPEs [27]. We measure the sustainable performance of the SUTs [34, 33] to detect if the latency spike is due to backpressure or unsustainable workload. If the latency value is higher than a given threshold because the system cannot sustain the workload, then AJoin scales up or out. Each created query in AJoin features this threshold value. For simplicity, we set the same threshold value for all queries. However, the overall methodology remains the same with different threshold values for each query. Because of the space constraints, we will include the related experimental results in the technical report of this paper.

7.1 Scalability

Figure 15 shows the impact of scalability on the performance of the SUTs. All the queries are submitted to the SUTs at compile-time. The queries are 2-way joins and have 50% query similarity.

For this experiment, we remove selection predicates from input queries to measure the performance of pure join operation. We can observe that the performance of all SUTs increases with more resources. Also, with more parallel queries, the overall data throughput of AJoin increases dramatically. The reason is that sharing opportunities increase with more parallel queries.

The throughput of AStream is significantly lower than AJoin. The reason is that AStream performs scan, data, and computation sharing if the input queries have common join predicate. Queries with different join predicates are deployed as separate stream jobs. The computation sharing in AStream is not always beneficial (e.g., Figure 6). Because AJoin supports cost-based optimization, in addition to rule-based optimization, it groups queries in query groups and shares the data and computation if the sharing is beneficial. AStream supports static QEP. Each query eagerly utilizes all available resources. Also, AStream utilizes nested-loop joins.

We execute Spark with hash join implementation with the Catalyst optimizer [5]. With multiple queries, submitted at compile-time, the optimizer shares common subqueries, such as joins with the same join predicate. The sharing is possible because there is no selection predicate. For queries with selection predicates, Spark cannot share the computation and data. For joins with different join predicates, Spark deploys a new QEP. Also, Spark does not utilize late materialization. The hashing phase in Spark is blocking. It uses blocking stage-oriented architecture.

AJoin performs better than AStream, Spark, and Flink even with single query setups. The reason is the join implementation of AJoin. AJoin uses not only data parallelism (like AStream, Spark, and Flink) but also pipeline parallelism for the join operation. The join operator in AStream, Spark, and Flink remains idle and buffers input tuples until the window is triggered. AStream and Flink perform nested-loop join after the window is triggered. AJoin performs windowing in the source operator. While the tuples are buffered, they are indexed on-the-fly. Therefore the load of join operator is lower in AJoin, as it performs the set-intersection operation. After the join is performed, AStream, Spark, and Flink create many new data objects. These new objects cause extensive heap memory usage. AJoin reuses existing objects, keeps them un-joined (late materialization), and performs full materialization at the sink operator. Because the data tuples are indexed, AJoin avoids to iterate all the data elements while joining them, but only indexes. Also, at the partitioning phase, AJoin iterates over indexes to partition a set of tuples with the same index at once, rather than iterating over each data tuple. Different from Flink, AJoin performs incremental join computation. Quantifying the impact of each component (e.g., indexing, grace join usage, late materialization, object reuse, task-parallelism, etc) stated above, is nontrivial because these components function as an atomic unit. If we detach one component (e.g., indexing), then the whole join implementation would fail to execute. However, there is a significant improvement in throughput from 0.1 M t/s in Flink to 2.04 M t/s in AJoin.

Figure 16 shows the space used to buffer tuples and indexes in AJoin. With more queries, AJoin buffers more tuples and indexes. However, AJoin shares tuples among different queries and avoids new object creation and copy. The buffer size increases more for indexes than for tuples. The reason is that each tuple might be reused by different indexes. In this figure, the key space is between 0 and 500. When we increase the key space in the orders of millions, the index buffer space also increases significantly. Although this increase did not cause significant overhead in our setup (48GB memory per node), with low-memory setups and with very large key space, index usage causes significant overhead for AJoin.

Figure 17 shows the effect of distinct keys and the selectivity of selection predicates on the performance of the SUTs. Given that the data throughput is constant, with less distinct keys

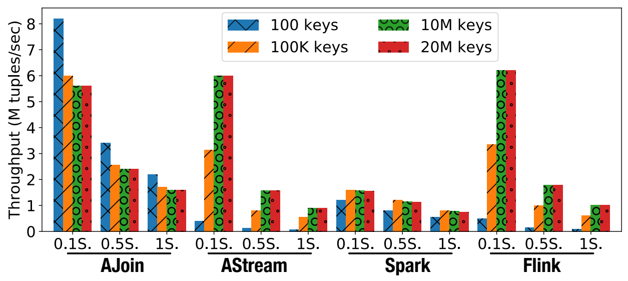


Figure 17: The effect of the number of distinct keys in stream sources and the selectivity of selection operators on the performance of AJoin, AStream, Spark, and Flink. Values on the x-axis show the selectivity of selection operators.

Flink and AStream output more tuples as a result of the cross product. This leads to an increase in data, computation, copy, serialization, and deserialization cost. With more distinct keys the performance of AJoin decreases, because AJoin cannot benefit from the late materialization. At the same time, the performance of Flink and AStream increases, because it performs fewer cross products and data copy. As the number of distinct keys increases, the throughput of Spark first increases then decreases slowly. The reason is that Spark utilizes hash join. With more keys, maintaining the hash table in memory becomes costly.

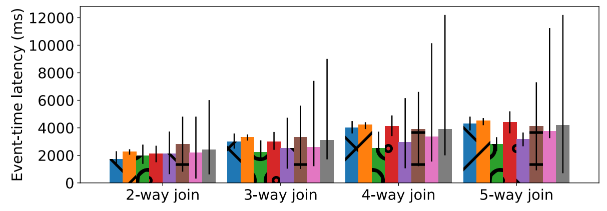
As the selectivity of the selection operator increases, the performance of all SUTs decreases. The decrease is steep in Flink and AStream. The reason is that the performance of the low-selective selection operator dominates the overall throughput. When the selectivity increases, data copy and inefficient join implementation become the bottleneck for the whole QEP.

The effect of the selectivity on Spark is more stable than other systems. In other words, as the selection operator filters more tuples, the overall performance of Spark does not exhibit an abrupt increase. Although Spark utilizes a hash join implementation, it adopts a stage-oriented mini-batch processing model. For example, hashing and filtering are separate stages of the job, which operate on the whole RDD. The subsequent stage cannot be started if all the parent stages are not finished. AJoin, AStream, and Flink, however, perform a tuple-at-a-time processing model. Therefore, the throughput performance of these systems is mainly dominated by the performance of filtering operators, especially with low-selectivity selection operators. Also, Spark's hash join implementation includes a blocking phase (hashing). Flink and AStream, on the other hand, perform a nested-loop join, which performs better with less data (after the filtering phase).

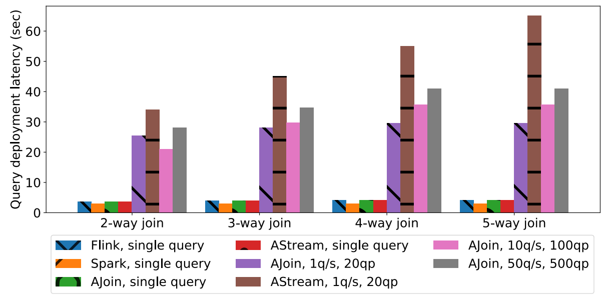
7.2 Dynamicity

Latency. In this section, we create and delete queries in an ad-hoc manner. Figure 18a shows the event-time latency of stream tuples for SC1. Since Flink cannot sustain ad-hoc query workloads, we show its event-time with a single query. During our experiments, we select the selectivity of selection operators to be approximately `0.5`. Although event-time latency of Flink is comparable with AJoin, the data throughput is significantly less than AJoin (Figure 15). The error bars in the figure denote the maximum and minimum latency of tuples during the experiment. In SPEs the latency of tuples might fluctuate due to backpressure, buffer size, garbage collection, etc. [34]. Therefore, we measure the average latency of the tuples.

The event-time latency increases with 3-, 4-, and 5-way join queries. The reason is that a streaming tuple traverses through more operators in the QEP. As the query throughput increases, so does the gap between latency boundaries. The reason is that AJoin performs run-time optimizations, which result in high



(a) Average event-time latency of stream tuples with min and max boundaries for SC1



(b) Deployment latency for SC1. iq/s $j$qp indicates that i queries per second were created until the query parallelism is j.

Figure 18: Average event-time and deployment latency for SC1

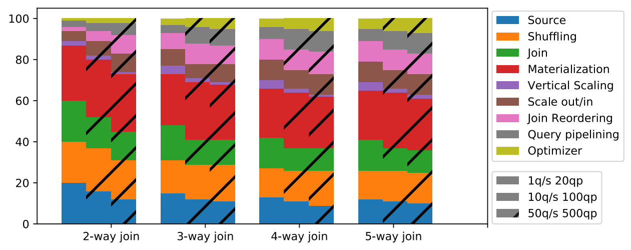


Figure 19: Breakdown of AJoin components in terms of percentage for SC1

latencies for some tuples. However, these high latencies can be regarded as outliers, because of much lower average latency.

The overall picture for event-time latency is similar for SC2. The only difference is that the average latency is lower and latency fluctuations are wider than SC1. The reason is that in SC2, the average number of running queries are less than SC1, which results in lower average event-time latency. The query throughput is higher in SC2, which results in more fluctuations in event-time latency.

Figure 18b shows the deployment latency for SC1 in AJoin. The experiment is executed in a 4-node cluster. The query similarity again is set to 50%. The query deployment latency for 1qs 20qp (create one query per second until there are 20 parallel queries) is higher than 10qs 100qp with 2-way joins. The reason is that query batch time is one second, meaning user requests submitted in the last second are batched and sent to the SUT. However, with 3- and 4-way joins, the overhead of on-the-fly QEP changes also contributes to query deployment latency.

Breakdown. Figure 19 shows a breakdown of the overhead by the AJoin components. We initialize AJoin with a 2-node cluster configuration and enable it to utilize up to 25 nodes. The overhead is based on the event-time latency of stream tuples. In this experiment, we ingest a special tuple to the QEP every second. Every component shown in Figure 19 logs its latency contribution to the tuple.

Note that the overhead of source, join, and materialization components are similar. This leads to a higher data throughput in the QEP. As the query throughput increases, the proportional overhead of horizontal scaling increases. The reason is that the optimizer eagerly shares the biggest subquery of a created query and eagerly deploys the remaining part of the query. Although the 3-phase protocol avoids stopping the QEP, it also has an impact on the overall latency. With 3-way and 4-way joins, the cost of query pipelining and join reordering also increase. With more join operators in a query, subquery sharing opportunities are high. So, the optimizer frequently pipelines the part of the newly created query to the existing query. Also, we can see that materialization is one of the major components causing latency. The reason is that tuples have to be fully materialized, copied, serialized, and sent to different physical output channels. We notice that similar overhead of source, join, and materialization leads to a higher data throughput (e.g., the throughput of 2-way is higher than others). The reason is that when n (n-way join) increases, new stream sources, join operators, and sink operators are deployed. Therefore, the overall overhead for these operators remains stable. The overhead of the optimizer also increases as n (n-way join) gets higher and as query throughput increases. The reason is that the sharing opportunities increase with more queries and with 3- and more n-way joins.

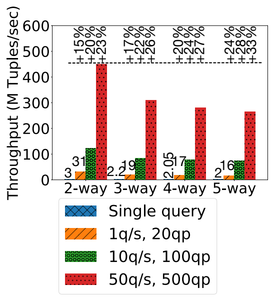
Throughput. Figure 20 shows the effect of n-way joins, query groups, and query similarity to the performance of the SUTs. We show the performance improvement of AJoin when submitting queries at compile-time above the dashed lines in the figure. As n increases in n-way joins, the throughput of AJoin drops (Figure 20a). The performance drop is sharp from 2-way join to 3-way join. The reason is that 3- and more n-way joins benefit from the late materialization more. Also, the performance difference between ad-hoc and compile-time query processing increases as the query throughput and n increase.

Figure 20b shows the throughput of AStream, Spark, and Flink with n-way join queries. Because of the efficient join implementation, Spark performs better than other SUTs with single query execution. The performance of Flink and AStream decreases with more join operators. In some 4- and 5-way join experiments, Flink and AStream were stuck and remained unresponsive. The reason is that each join operator creates new objects in memory, which leads to intensive, CPU, network usage and garbage collection stalls. While Spark also performs data copy, its Catalyst optimizer efficiently utilizes on-heap and off-heap memory to reduce the effect of data copy on the performance.

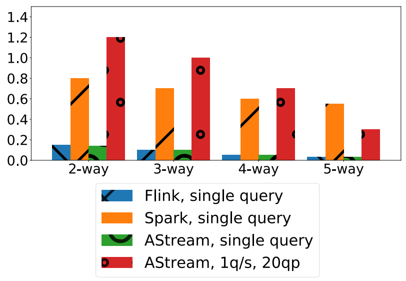
Figure 20c shows the effect of the number of query groups on the performance of AJoin. With more query groups the throughput of AJoin decreases. However, the decreasing speed slows down gradually. Although there are less sharing opportunities with more query groups, updating the QEP becomes cheaper (as a result of incremental computation). The incremental computation also leads to a decrease in the overhead of executing queries ad-hoc.

Figure 20d shows the effect of query similarity on the performance of the SUTs. Both AStream and AJoin perform better with more similar queries. However, the performance increase is higher in AJoin. AStream lacks all the run-time optimization techniques AJoin features. As a result, AStream shares queries only with the same structure (e.g., 2-way joins can be shared only with 2-way joins) and the same join predicates. The effect of executing queries in an ad-hoc manner decreases as the query similarity increases. The overall picture in SC2 is similar with SC1.

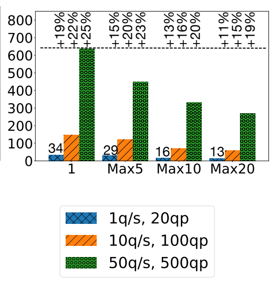
Impact of each component. Figure 21 shows the impact of AJoin's optimization components on the performance. In this experiment, we disable one optimization component (e.g., join reordering) and measure the performance drop. When the number of join operations in a query increases, the impact of join reordering and query pipelining also increase. Also, with more query throughput, the optimizer shares input queries aggressively. Therefore, the impact of the query pipelining increases with higher query throughput. As the number of query groups increases, the



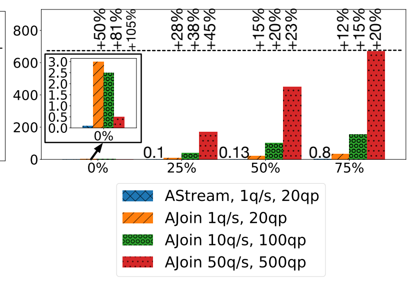
(a) Throughput of AJoin with n-way joins



(b) Throughput of AStream, Spark, and Flink with n-way joins



(c) Throughput of AJoin with different query groups



(d) Throughput of AJoin and AStream with different query similarities

Figure 20: Throughput measurements for AJoin, AStream, Spark, and Flink. +P% above the dashed lines denotes that the throughput increases by P% when queries are submitted at compile-time.

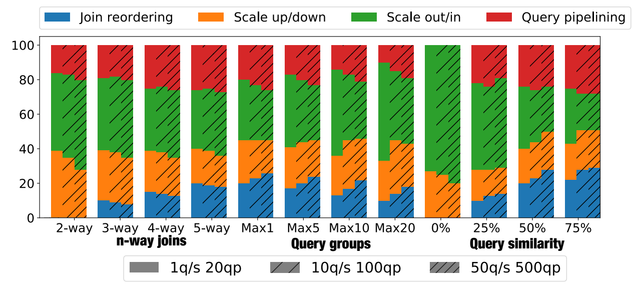


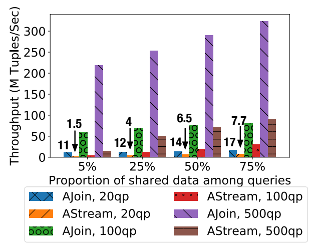
Figure 21: Impact of AJoin components in terms of percentage

impact of the join reordering optimization decreases because of the drop in sharing opportunities. This also leads to the extensive use of scaling out and in. When all queries are dissimilar, the join reordering and query pipelining have zero impact on overall execution. With more similar queries, the effect of other components, especially the join reordering component, increases.

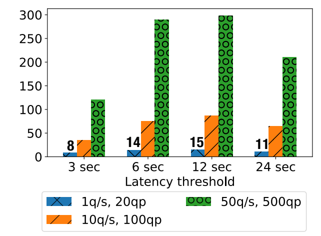
The overall picture is similar in SC2. The most noticeable difference is that the impact of scaling out and in is less, and the impact of join reordering is more. The execution time and the query throughput in SC1 are higher than SC2. In SC2, queries are not only created but also deleted with lower throughput. This leads to a higher impact on join reordering.

Cost of sharing. Figure 22a shows the performance of AStream and AJoin with four input streams: 5%, 25%, 50%, and 75% shared. For example, 50% shared data source means that tuples are shared among 50% of all queries. We omit experiments with 0% shared data source, as in this scenario all the data tuples are filtered and no join operation is performed. We perform this experiment with a workload suitable for AStream (i.e., all join queries have the same join predicate and the same number of join operators) and disable the dynamicity property (except query grouping) of AJoin. This setup enables us to measure the cost of sharing and query-set payload of AStream and AJoin. As the proportion of shared data decreases, the performance gap between AStream and AJoin increases. The reason is that AJoin performs query grouping that leads to an improved performance (Figure 6). The impact of the query grouping is more evident when the proportion of shared data is small.

Impact of the latency threshold value. Figure 22b shows the throughput of AJoin with different latency threshold values. The latency threshold value, which is 5 seconds in our experiments, needs to be configured carefully. When it is too low (3 seconds in Figure 22b), we experience an overhead for frequent optimizations. When it is too high (24 seconds in Figure 22b), there is a loss in optimization potential.



(a) Impact of data sharing and query-set payload to the throughput of AJoin and AStream



(b) Impact of the latency threshold value to the throughput of AJoin

Figure 22: Cost of data sharing and the impact of the latency threshold value with 3-way join queries

8. CONCLUSION

In this paper we present AJoin, an ad-hoc stream join processing engine. We develop AJoin based on two main concepts: (1) Efficient distributed join architecture: AJoin features pipeline-parallel join architecture. This architecture utilizes late materialization, which significantly reduces the amount of intermediate results between subsequent join operators; (2) Dynamic query processing: AJoin features an optimizer, which reoptimizes ad-hoc stream queries periodically at run-time, without stopping the QEP. Also, the data processing layer supports dynamicity, such as vertical and horizontal scaling and join reordering;

We benchmark AJoin, AStream, Spark, and Flink. When all the queries are submitted at compile-time, AJoin outperforms Flink by orders of magnitude. With single query workloads, AJoin also outperforms AStream, Spark, and Flink. With more join operators in a query (3-, 4-, 5−way joins) the performance gap between AJoin and the other systems even increases. With ad-hoc stream query workloads, Flink and Spark cannot sustain the workload, and AStream's performance is less than AJoin's. In the future, we envision to further distribute concepts of AJoin into an Internet of Things data processing system that we are currently developing at TU Berlin

Acknowledgments. We thank Phil Bernstein, Walter Cai, Alireza Rezaei Mahdiraji, and all the anonymous reviewers for their very valuable feedback. This work has been supported by the German Ministry for Education and Research as Berlin Big Data Center BBDC 2 (01IS18025A) and the German Federal Ministry for Economic Affairs and Energy, Project "ExDra" (01MD19002B).

9. REFERENCES

[1] Apache Kafka. https://kafka.apache.org/, 2019. [Online; accessed 17-August-2019].

[2] D. Aloise, A. Deshpande, P. Hansen, and P. Popat. Np-hardness of euclidean sum-of-squares clustering. *Machine learning*, 75(2):245–248, 2009.

[3] C. Anglano, M. Canonico, and M. Guazzone. Fc2q: exploiting fuzzy control in server consolidation for cloud applications with sla constraints. *Concurrency and Computation: Practice and Experience*, 27(17):4491–4514, 2015.

[4] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia. Structured streaming: A declarative api for real-time applications in apache spark. In *Proceedings of the 2018 International Conference on Management of Data*, pages 601–613. ACM, 2018.

[5] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1383–1394. ACM, 2015.

[6] S. Arumugam, A. Dobra, C. M. Jermaine, N. Pansare, and L. Perez. The datapath system: a data-centric analytic processing engine for large data warehouses. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 519–530. ACM, 2010.

[7] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *ACM SIGMOD record*, volume 29, pages 261–272. ACM, 2000.

[8] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16. ACM, 2002.

[9] A. Beitch, B. Liu, T. Yung, R. Griffith, A. Fox, D. A. Patterson, et al. Rain: A workload generation toolkit for cloud computing applications. *University of California, Tech. Rep. UCB/EECS-2010-14*, 2010.

[10] S. Bradshaw and P. Howard. Troops, trolls and troublemakers: A global inventory of organized social media manipulation. 2017.

[11] L. Braun, T. Etter, G. Gasparis, M. Kaufmann, D. Kossmann, D. Widmer, A. Avitzur, A. Iliopoulos, E. Levy, and N. Liang. Analytics in motion: High performance event-processing and real-time analytics in the same database. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 251–264. ACM, 2015.

[12] G. Candea, N. Polyzotis, and R. Vingralek. A scalable, predictable join operator for highly concurrent data warehouses. *PVLDB*, 2(1):277–288, 2009.

[13] G. Candea, N. Polyzotis, and R. Vingralek. Predictable performance and high query concurrency for data analytics. *The VLDB Journal*, 20(2):227–248, 2011.

[14] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas. State management in apache flink®: consistent stateful distributed stream processing. *PVLDB*, 10(12):1718–1729, 2017.

[15] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[16] V. Cardellini, M. Nardelli, and D. Luzi. Elastic stateful stream processing in storm. In *High Performance Computing & Simulation (HPCS), 2016 International Conference on*, pages 583–590. IEEE, 2016.

[17] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer. Path sharing and predicate evaluation for high-performance xml filtering. *ACM Transactions on Database Systems (TODS)*, 28(4):467–516, 2003.

[18] T. Dokeroglu, S. Ozal, M. A. Bayir, M. S. Cinar, and A. Cosar. Improving the performance of hadoop hive by sharing scan and computation tasks. *Journal of Cloud Computing*, 3(1):12, 2014.

[19] R. Ebenstein, N. Kamat, and A. Nandi. Fluxquery: An execution framework for highly interactive query workloads. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1333–1345. ACM, 2016.

[20] A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, and I. Stoica. Above the clouds: A berkeley view of cloud computing. *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, 28(13):2009, 2009.

[21] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu. Elastic scaling for data stream processing. *IEEE Transactions on Parallel & Distributed Systems*, (1):1–1, 2014.

[22] G. Giannikis, G. Alonso, and D. Kossmann. SharedDB: killing one thousand queries with one stone. *PVLDB*, 5(6):526–537, 2012.

[23] G. Giannikis, D. Makreshanski, G. Alonso, and D. Kossmann. Workload optimization using shareddb. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1045–1048. ACM, 2013.

[24] G. Giannikis, D. Makreshanski, G. Alonso, and D. Kossmann. Shared workload optimization. *PVLDB*, 7(6):429–440, 2014.

[25] J. Giceva, G. Alonso, T. Roscoe, and T. Harris. Deployment of query plans on multicores. *PVLDB*, 8(3):233–244, 2014.

[26] M. A. Hammad, M. J. Franklin, W. G. Aref, and A. K. Elmagarmid. Scheduling for shared window joins over data streams. *PVLDB*, 29:297–308, 2003.

[27] T. Heinze, Z. Jerzak, G. Hackenbroich, and C. Fetzer. Latency-aware elastic scaling for distributed data stream processing systems. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, pages 13–22. ACM, 2014.

[28] T. Heinze, Y. Ji, L. Roediger, V. Pappalardo, A. Meister, Z. Jerzak, and C. Fetzer. Fugu: Elastic data stream processing with latency constraints. *IEEE Data Eng. Bull.*, 38(4):73–81, 2015.

[29] T. Heinze, L. Roediger, A. Meister, Y. Ji, Z. Jerzak, and C. Fetzer. Online parameter optimization for elastic data stream processing. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 276–287. ACM, 2015.

[30] N. R. Herbst, S. Kounev, et al. Modeling variations in load intensity over time. In *Proceedings of the third international workshop on Large scale testing*, pages 1–4. ACM, 2014.

[31] T. Ibaraki and T. Kameda. On the optimal nesting order for computing n-relational joins. *ACM Transactions on Database Systems (TODS)*, 9(3):482–502, 1984.

[32] G. Jacques-Silva, R. Lei, L. Cheng, G. J. Chen, K. Ching, T. Hu, Y. Mei, K. Wilfong, R. Shetty, S. Yilmaz, et al. Providing streaming joins as a service at facebook. *PVLDB*, 11(12):1809–1821, 2018.

[33] J. Karimov. *Stream Benchmarks*, pages 1–6. Springer International Publishing, Cham, 2018.

[34] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl. Benchmarking distributed stream data processing systems. In *IEEE 34th International Conference on Data Engineering (ICDE)*, pages 1507–1518. IEEE, 2018.

[35] J. Karimov, T. Rabl, and V. Markl. Astream: Ad-hoc shared stream processing. In *SIGMOD 2019*. ACM, 2019.

[36] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. Application of hash to data base machine and its architecture. *New Generation Computing*, 1(1):63–74, 1983.

[37] D. Kossmann and K. Stocker. Iterative dynamic programming: a new class of query optimization algorithms. *ACM Transactions on Database Systems (TODS)*, 25(1):43–82, 2000.

[38] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. *PVLDB*, 86:128–137, 1986.

[39] Q. Li, M. Shao, V. Markl, K. Beyer, L. Colby, and G. Lohman. Adaptively reordering joins during query execution. In *IEEE 23rd International Conference on Data Engineering, 2007. ICDE 2007.*, pages 26–35. IEEE, 2007.

[40] L. Lu, X. Zhu, R. Griffith, P. Padala, A. Parikh, P. Shah, and E. Smirni. Application-driven dynamic vertical scaling of virtual machines in resource pools. In *2014 IEEE Network Operations and Management Symposium (NOMS)*, pages 1–9. IEEE, 2014.

[41] D. Makreshanski, G. Giannikis, G. Alonso, and D. Kossmann. Mqjoin: efficient shared execution of main-memory joins. *PVLDB*, 9(6):480–491, 2016.

[42] D. Makreshanski, J. Giceva, C. Barthels, and G. Alonso. Batchdb: Efficient isolated execution of hybrid oltp+ olap workloads for interactive applications. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 37–50. ACM, 2017.

[43] V. Markl, V. Raman, D. Simmen, G. Lohman, H. Pirahesh, and M. Cilimdzic. Robust query processing through progressive optimization. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 659–670. ACM, 2004.

[44] G. Moerkotte and T. Neumann. Dynamic programming strikes back. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 539–552. ACM, 2008.

[45] W. Phillips. Meet the trolls. *Index on Censorship*, 40(2):68–76, 2011.

[46] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 23–34. ACM, 1979.

[47] B. Suleiman, S. Sakr, R. Jeffery, and A. Liu. On understanding the economics and elasticity challenges of deploying business applications on public cloud infrastructure. *Journal of Internet Services and Applications*, 3(2):173–193, 2012.

[48] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, et al. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147–156. ACM, 2014.

[49] I. Trummer and C. Koch. Solving the join ordering problem via mixed integer linear programming. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1025–1040. ACM, 2017.

[50] A. Turner, A. Fox, J. Payne, and H. S. Kim. C-mart: Benchmarking the cloud. *IEEE Transactions on Parallel and Distributed Systems*, 24(6):1256–1266, 2012.

[51] M. Turner, D. Budgen, and P. Brereton. Turning software into a service. *Computer*, 36(10):38–44, 2003.

[52] S. D. Viglas, J. F. Naughton, and J. Burger. Maximizing the output rate of multi-way join queries over streaming information sources. *PVLDB*, 29:285–296, 2003.

[53] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 423–438. ACM, 2013.

[54] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *Presented as part of the*, 2012.

[55] S. Zeuch, B. Del Monte, J. Karimov, C. Lutz, M. Renz, J. Traub, S. Breß, T. Rabl, and V. Markl. Analyzing efficient stream processing on modern hardware. *PVLDB*, 12(5):516–530, 2019.