Scalable and Adaptive Online Joins

Mohammed Elseidy, Abdallah Elguindy, Aleksandar Vitorovic, and Christoph Koch
{firstname}.{lastname}@epfl.ch
École Polytechnique Fédérale de Lausanne

ABSTRACT

Scalable join processing in a parallel shared-nothing environment requires a partitioning policy that evenly distributes the processing load while minimizing the size of state maintained and number of messages communicated. Previous research proposes static partitioning schemes that require statistics beforehand. In an online or streaming environment in which no statistics about the workload are known, traditional static approaches perform poorly.

This paper presents a novel parallel online dataflow join operator that supports arbitrary join predicates. The proposed operator continuously adjusts itself to the data dynamics through adaptive dataflow routing and state repartitioning. The operator is resilient to data skew, maintains high throughput rates, avoids blocking behavior during state repartitioning, takes an eventual consistency approach for maintaining its local state, and behaves strongly consistently as a black-box dataflow operator. We prove that the operator ensures a constant competitive ratio 3.75 in data distribution optimality and that the cost of processing an input tuple is amortized constant, taking into account adaptivity costs. Our evaluation demonstrates that our operator outperforms the state-of-the-art static partitioning schemes in resource utilization, throughput, and execution time.

1. INTRODUCTION

To evaluate joins with *arbitrary* predicates on very large volumes of data, previous works [35, 28] propose efficient partitioning schemes for *offline* theta-join processing in parallel environments. The goal is to find a scheme that achieves load balancing while minimizing duplicate data storage and network traffic. Offline approaches require that all data is available beforehand and accordingly perform optimization statically before query execution.

However, online and responsive analysis of fresh data is necessary for an increasing number of applications. Businesses, engineers and scientists are pushing data analytics engines earlier in their workflows for rapid decision making. For example, in algorithmic trading, strategy designers run online analytical queries on real-time order book data. Order books consist of frequently changing orders waiting to be executed at a stock exchange. Some orders may stay in the order book relatively long before they are executed or revoked. Orders are executed through a matching engine that matches between buyer and seller trades using sophisticated matching rules. A broad range of applications, including fraud-detection mining algorithms, interactive scientific simulations, and intelligence analysis are characterized as follows: They (i) perform joins on large volumes of data with complex predicates; (ii) require operating in real-time while preserving efficiency and fast response times; (iii) and maintain large state, which potentially depend on the complete history of previously processed tuples [10, 6].

Previous work on stream processing has received considerable attention [3, 5], but is geared towards window-based relational stream models, in which state typically only depends on a recent window of tuples [10]. Although this simplifies the architecture of the stream processing engine, it is ineffective for emerging application demands that require maintaining large historical states. Only recently, has this concern been acknowledged and interest been raised in devising scalable *stateful* operators for stream processing [10].

This motivates our work towards *full-history* theta-join processing in an *online* scalable manner. In this context, the traditional optimize-then-execute strategy is ineffective due to *lack of statistics* such as cardinality information. For pipelined queries, cardinality estimation of intermediate results is challenging because of the possible correlations between predicates [24, 36] and the generality of the join conditions. Moreover, statistics are not known beforehand in streaming scenarios, where data is fed in from remote data sources [14]. Therefore, the online setting requires a versatile dataflow operator that adapts to the data dynamics. Adaptivity ensures low latency, high throughput, and efficient resource utilization throughout the entire execution.

This paper presents a novel design for an *intra-adaptive* dataflow operator for stateful online join processing. The operator supports arbitrary join-predicates and is resilient to data skew. It encapsulates adaptive state partitioning and dataflow routing. The authors of [18] point out the necessity of investigating systematic adaptive techniques as current ones lack theoretical guarantees about their behavior and instead rely on heuristic-based solutions. Therefore, to design a *provably*-efficient operator we need to characterize the optimality measures and the adaptivity costs of the operator. This requires theoretical analysis and addressing several systems design challenges which we discuss while outlining our main contributions.

1. Adapting the partitioning scheme requires state relocation which incurs additional network traffic costs. Our design employs a *locality*-aware migration mechanism that incurs *minimal* state relocation overhead.

2. We present an online algorithm that efficiently decides when to *explore* and *trigger* new partitioning schemes. An aggressively adaptive approach has excessive migration overheads, whereas a conservative approach does not adapt well to data dynamics which results in poor performance and resource utilization. Our presented algorithm balances between maintaining optimal data distribution and adaptation costs. It ensures a constant *competitive ratio* (3.75) in data

distribution optimality and *amortized linear* communication cost (including adaptivity costs).

3. Previous adaptive techniques [34, 26, 31] follow a general *blocking*-approach for state relocation that quiesces input streams until relocation ends. Blocking approaches are not suitable for online operators that maintain large states because they incur lengthy stalls. Our design adopts a *non-blocking* protocol for migrations that seamlessly integrates state relocation with *on-the-fly* processing of new tuples while ensuring *eventual consistency* and result correctness.

4. Statistics are crucial for optimizing the partitioning scheme. The operator must gather them on-the-fly and constantly maintain them up-to-date. Traditionally, adaptive solutions delegate this to a centralized entity [34, 26, 20, 43] which may be a bottleneck if the volume of feedback is high [18]. Our approach for computing global statistics is decentralized requiring no communication or synchronization overhead.

Next we discuss related work; §3 introduces the background and concepts used throughout the rest of the paper and it outlines the problem and the optimization criteria; §4 presents the adaptive data-flow operator and its design in detail; §5 evaluates performance and validates the presented theoretical guarantees; and §6 concludes.

2. RELATED WORK

Parallel Join Processing. In the past decades, much effort has been put into designing distributed and parallel join algorithms to cope with the rapid growth of data sets. Graefe gives an overview of such algorithms in [19]. Schneider *et al.* [33] describe and evaluate several parallel equi-join algorithms that adopt a *symmetric partitioning method* which partitions input on the join attributes, whereas Stamos *et al.* [35] present the *symmetric fragment-and-replicate* method to support parallel theta-joins. This method relies on replicating data to ensure result completeness and on a heuristic model to minimize total communication cost.

MapReduce Joins. MapReduce [12, 1] has emerged as one of the most popular paradigms for parallel computation that facilitates parallel processing of large data and scalability. There has been much work done towards devising efficient join algorithms using this framework. Previous work focuses primarily on equi-join implementations [4, 9, 30, 32, 44] by partitioning the input on the join key, whereas Map-Reduce-Merge [44] supports other join predicates as well. However, the latter requires explicit user knowledge and modifications to the MapReduce model. Recently, Okcan *et al.* [28] proposed techniques that supports theta-join processing without changes to the model. Finally, Zhang *et al.* [45] extend Okcan's work to evaluate multi-way joins.

All of the aforementioned algorithms are offline. They have a blocking behavior that is attributed either to their design or to the nature of the MapReduce framework (the *reduce* phase cannot commence before the *map* phase has completed). In contrast, this paper sets out to build an online operator that supports scalable processing of theta-joins which allows for early results and rich interactivity.

Online Join Algorithms. There has been great interest in designing non-blocking join algorithms. The symmetric hash join SHJ [42] is one of the first along those lines to support equi-joins. It extends the traditional hash join algorithm to support pipelining. However, the SHJ requires that relations fit in memory. XJOIN [40] and DPHJ [25] extend the SHJ with overflow resolution schemes that allow parts of the hash tables to be spilled out to disk for later processing. Similarly, RPJ [37] uses a statistics-based flushing strategy that tries to keep tuples that are more likely to join in memory. Dittrich *et al.* present PMJ [15, 16] which is a sorting-based online join algorithm that supports inequality predicates as well. Mokbel *et al.* present HMJ [27] that combines the advantages of the two state-of-the-art non-blocking algorithms, namely XJOIN and PMJ. Finally, The family of ripple joins [21] generalize block nested loop join, index loop join, and hash join to their online counterparts. Ripple joins automatically adapt their behavior to provide approximate running aggregates defined within confidence intervals. All the previous algorithms are local online join algorithms, and thus, are orthogonal to our data-flow operator. In the presented parallel operator, each machine can freely adopt any flavor of the aforementioned non-blocking algorithms to perform joins locally on its assigned data partition.

Stream Processing Engines. Distributed stream processors such as BOREALIS [3] and STREAM [5] focus on designing efficient operators for continuous queries. They assume that data streams are processed in several sites, each of which holds some of the operators. They are optimized to handle unbounded streams of data by dropping tuples (load shedding) or having window semantics. In contrast, this paper is concerned with the design of a scalable operator, as opposed to a centralized approach. And along the same lines of [10], it targets stateful streaming queries which maintain large states, potentially full historical data. Castro *et al.* [10] introduce a scale-out mechanism for stateful operators, however they are limited to stream models with key attributes.

Adaptive Query Processing. Adaptive query processing AQP techniques cope their behavior, at run-time, to data characteristics. There has been a great deal of work on centralized AQP [8, 14, 22, 17] over the last few years. For parallel environments, [18] presents a detailed survey. The FLUX operator [34] is the closest to our work. FLUX is a *general* adaptive operator that encloses adaptive state partitioning and routing. The operator is *content-sensitive* and suitable for look-up based operators. Although the authors focus on single-input aggregate operators [26], it can support a restricted class of join predicates, e.g. equi-join. FLUX supports equi-joins under skewed data settings but requires explicit user knowledge about partitions before execution. In [20, 41], the authors present techniques to support multi-way non equi-joins. All these approaches are mainly applied to data streaming scenarios with window semantics. On the other hand, this paper presents an adaptive dataflow operator for general joins. It advances the state of the art in online equi-join processing in the presence of data skew. Most importantly, along the lines of [14, 17, 22], the operator runs on long running full-history queries without window semantics, load shedding, and data arrival order restrictions.

Eddies. Eddies [7, 13] are among the first adaptive techniques known for query processing. Eddies act as a tuple router that is placed at the center of a dataflow, intercepting all incoming and outgoing tuples between operators in the flow. Eddies observe the rates of all the operators and accordingly make decisions about the order at which new tuples will visit the operators. In principal, eddies are able to choose different operator orderings for each tuple within the query processing engine to adapt to the current information about the environment and data. Compared to our work, this direction seeks adaptations at an orthogonal hierarchical level, it is concerned with *inter*-operator adaptivity as opposed to our work on *intra*-operator adaptivity. Moreover, the original eddies architecture is centralized and cannot be applied to a distributed setting in a straightforward

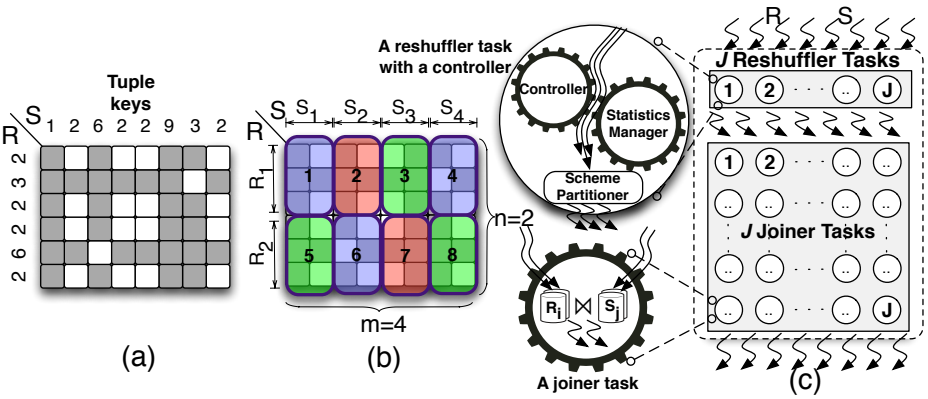


Figure 1: (a) $R \bowtie S$ join-matrix example, gray cells satisfy the $\neq$ predicate. (b) a (2,4)-mapping scheme using $J = 8$ machines. (c) the theta-join operator structure. J *reshuffler* and J *joiner* tasks where each physical machine is assigned one from each. One of the reshuffler tasks is designated the additional role of a *controller*.

manner [18]. However, the work in [38] leverages the eddies design to a distributing setting but assumes window semantics; tolerates loss of information; and neglects adaptations on operators that hold internal state.

3. PRELIMINARIES

This section defines notations and conventions used throughout the rest of this paper. It describes the data partitioning scheme used by the dataflow operator, outlines the operator's structure, and defines the optimization criteria.

3.1 Join Partitioning Scheme

We adopt and extend the join-matrix model [28, 35] to the data streaming scenario. A join between two data streams R and S is modeled as a join-matrix $\mathcal{M}$. For row i and column j, the matrix cell $\mathcal{M}(i,j)$ represents a potential otput result. $\mathcal{M}(i,j)$ is *true* if and only if the tuples r_i and s_j satisfy the join predicate. The result of any join is a subset of the cross-product. Hence, the join-matrix model can represent any join condition. Fig. 1a shows an example of a join-matrix with the predicate $\neq$.

We assume a shared-nothing cluster architecture. J physical machines are dedicated to a *single* join operator. A partitioning scheme maps matrix cells to machines for evaluation such that each cell is assigned to exactly one machine. This ensures result completeness and avoids expensive post processing or duplicate elimination. There are many possible mappings [28], however, we present a grid-layout partitioning scheme which (i) ensures *minimum* join work distribution among *all* machines, (ii) incurs *minimal* storage and communication costs, (iii) and has a symmetric structure that lends itself to adaptivity. We refer the interested reader to [**?**] for bounds, proofs, and comparison with previous partitioning approaches [28]. The scheme can be briefly described as follows: to achieve load balance such that each machine is assigned the same number of cells to evaluate, the join-matrix M is divided into J regions of equal area and each machine is assigned a single region. As illustrated in Fig. 1b, the relations are split into equally sized partitions $R_1, R_2, \ldots, R_n$ and $S_1, S_2, \ldots, S_m$ where $n \cdot m = J$. For every pair $1 \leq i \leq n$ and $1 \leq j \leq m$, there is exactly one machine storing both partitions R_i and S_j. Accordingly, each machine evaluates the corresponding $R_i \bowtie_\theta S_j$ independently. We refer to this as the (n,m)-*mapping scheme*.

3.2 Operator Structure

As illustrated in Fig. 1c, the operator is composed of two sets of tasks. The first set consists of joiner tasks that do the actual join computation whereas the reshufflers set is responsible for distributing and routing the tuples to the appropriate joiner tasks. An incoming tuple to the operator is randomly routed to a reshuffler task. One task among the reshufflers, referred to as the controller, is assigned the additional responsibility of monitoring global data statistics and triggering adaptivity changes. Each of the J machines run one joiner task and one resuffler task.

The reshufflers randomly divide incoming tuples uniformly among partitions. Under an (n,m)-mapping scheme, for an incoming $r(s)$ tuple, it is assigned a *randomly* chosen partition $R_i(S_j)$. This routing policy ensures load balance and resilience to data skew, i.e., *content-insensitivity*. For a large number of input tuples, the numbers in each partition are roughly equal. Thus, all bounds, later discussed, are meant to approximately hold in expectation with high probability.

Exactly m joiners are assigned partition R_i and exactly n joiners are assigned partition S_j. Therefore, whenever a reshuffler receives a new $R(S)$ tuple and decides that it belongs to partition $R_i(S_j)$, the tuple is forwarded to $m(n)$ distinct joiner tasks. Any flavor of non-blocking join algorithm, e.g., [42, 40, 37, 15, 21], can be independently adopted at each joiner task. Local non-blocking join algorithms traditionally operate as follows: when a joiner task receives a new tuple, it is stored for later use and joined with stored tuples of the opposite relation.

3.3 Input-Load Factor

Theta-join processing cost, in the presented model, is determined by the costs of joiners receiving input tuples, computing the join, and outputting the result. Under the presented grid-scheme, the join matrix is divided into congruent rectangular regions. Therefore, the costs are the same for every joiner. Since all joiners operate in parallel, we restrict our attention to analyzing one joiner.

The join computation and its output size on a single joiner are independent of the chosen mapping. This holds because

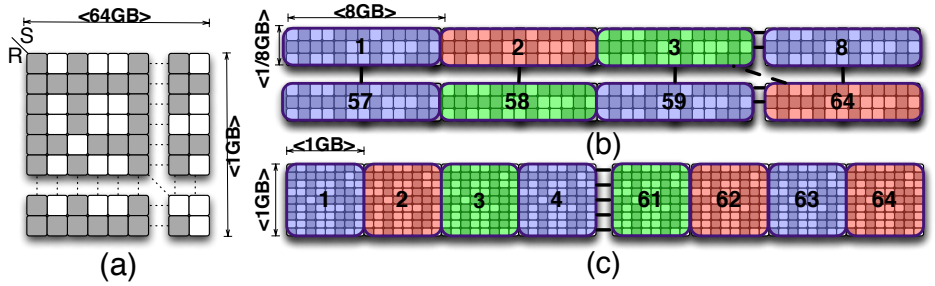


Figure 2: (a) join-matrix with dimensions 1GB and 64GB (b) a $(8,8)$-mapping scheme assigns an ILF of $(8\frac{1}{8})$GB (c) a $(1,64)$-mapping scheme assigns an ILF of 2GB.

both quantities are proportional to the area of a single region, which is $|R| \cdot |S| / J$. This is independent of n and m. However, the input size corresponds to the semi-perimeter of one region and is equal to $size_R \cdot |R| / n + size_S \cdot |S| / m$, where $size_R(size_S)$ is the size of a tuple of $R(S)$. This also represents the storage required by every joiner since each received tuple is eventually stored. We refer to this value as the **input-load factor** (ILF). This is the only performance metric that depends on the chosen mapping. *An optimal mapping covers the entire join matrix with minimum ILF.* Minimizing the ILF maximizes performance and resource utilization. This is extensively validated in our experiments (§5) and is attributed to the following reasons: (i) there is a monotonically increasing overhead for processing input tuples per machine. The overhead includes demarshalling the message; appending the tuple to its corresponding storage and index; probing the indexes of the other relation; sorting the input in case of sort-based online join algorithms [15, 27], etc. Minimizing machine input size results in higher local throughput and better performance. (ii) Minimizing storage size per machine is also necessary, because performance deteriorates when a machine runs out of main memory and begins to spill to disk. Local non-blocking algorithms perform efficiently when they operate within the memory capacity, however they employ overflow resolution strategies that prevent blocking, but persist to experience performance hits and long delayed join evaluation [14]. (iii) Overall, minimizing the ILF results in minimum global duplicate storage and replicated messages ($J \cdot ILF$). This maximizes overall operator performance and increases global resource utilization by minimizing total storage and network traffic and thus preventing congestion. This is essential for cloud infrastructures which typically follow *pay-as-you-go* policies.

Fig. 2 compares between two different mappings for a join-matrix with dimensions 1GB and 64GB for streams R and S respectively. Given 64 machines, an $(8,8)$-mapping results in an $(8\frac{1}{8})$GB ILF and a total of 520GB of replicated storage and messages. Whereas a $(1,64)$-mapping results in a 2GB ILF and a sum of 128GB of replicated data. Since stream sizes are not known in advance, maintaining an optimal (n,m)-mapping throughout execution requires adaptation and mapping changes.

3.4 Grid-Layout Partitioning Scheme

The partitioning scheme used throughout the paper is inspired, but greatly differs from that of [28]. Initially, the number of joiners will be restricted to powers of two. This allows the derivation of bounds (including most notably the input-load factor). Later this assumption will be relaxed. In this subsection, we give some theoretical justification of using this grid-layout scheme with a power of two number of joiners. In the previous work of Okcan *et al.*, the join matrix is divided into square regions with some of the machines left unused. The authors prove that the region semi-perimeter and area are within **twice** and **four** times that of the optimal lower bound, respectively.

THEOREM 3.1. ***(Okcan et al. [28])*** *Under the mapping scheme discussed in [28], the region semi-perimeter is at most* $4 \cdot \sqrt{|R| |S| / J}$ *and the region area is at most* $4RS/J$ *with the optimal lower bounds being respectively* $2 \cdot \sqrt{|R| |S| / J}$ *and* $|R| |S| / J$.

Under the grid-layout mapping scheme, allowing rectangular regions rather than restrictive square regions, the bounds derived can be substantially improved.

THEOREM 3.2. *Under the grid-layout mapping scheme, the region semi-perimeter is at most* 1.07 *times the optimal and the region area is exactly* $|R| |S| / J$ *attaining the optimum lower bound.*

PROOF. The area bound is straightforward. Since there are J regions each with exactly the same area, covering the join matrix, the area is exactly $|R| |S| / J$. It remains to show the semi-perimeter bound. If the ratio of the relation sizes is J or more, the grid-layout mapping is either $(1, J)$ or $(J, 1)$, being exactly optimal. Otherwise, let the ratio R/S be ρ where $1/J < \rho < J$. Since n and m are powers of two, it holds that $\frac{1}{2}\rho \leq n/m = n^2/J \leq 2\rho$. The semi-perimeter is $R/n + S/m = \rho S/n + Sn/J$. The maximum value of the semi-perimeter is $(\frac{1}{\sqrt{2}} + \sqrt{2})S\sqrt{\rho/J}$ and is attained at n being either $\sqrt{2\rho J}$ or $\sqrt{\rho J/2}$. This is at most 1.07 times the optimal lower bound. □

4. INTRA-OPERATOR ADAPTIVITY

We present an intra-adaptive operator that modifies its mapping configurations as data flows in. The goal of adaptive processing is, generally, dynamic recalibration to immediately react to the frequent changes in data and statistics. Adaptive solutions supplement regular execution with a control system that monitors performance, explores alternative configurations and triggers changes. These stages are defined within a cycle called the *Adaptivity Loop*. This section presents the design of an adaptive dataflow theta-join operator that continuously modifies its (n,m)-mapping scheme to reflect the optimal data assignment and routing policy.

Algorithm 1 Controller Algorithm.

Input: Tuple t
Initialize: $|R|, |S|, |\Delta R|, |\Delta S| \leftarrow 0$;
1: **function** UPDATE STATE(t)
2: **if** $t \in R$ **then**
3: $|\Delta R| \leftarrow |\Delta R| + J$ ▷ Scaled Increment.
4: **else**
5: $|\Delta S| \leftarrow |\Delta S| + J$
6: MigrationDecision($|R|, |S|, |\Delta R|, |\Delta S|$)
7: Route t according to the current (n, m)-scheme.
8: **end function**

We follow a discussion flow that adopts a common framework [14] that decomposes the adaptivity loop into three stages: (i) The *monitoring* stage that involves measuring data characteristics like cardinalities. (ii) The *analysis and planning* stage that analyzes the performance of the current (n, m)-mapping scheme and explores alternative layouts. (iii) The *actuation* stage that corresponds to migrating from one scheme to another with careful state relocation.

4.1 Monitoring Statistics

In this stage, the operator continuously gathers and maintains online cardinality information of the incoming data. Traditional adaptive techniques in a distributed environment [34, 26, 20, 43] either rely on a centralized controller that periodically gathers statistics or on exchanging statistics among peers [38, 46]. This may become a bottleneck if the number of participating machines and/or the volume of feedback collected is high [18]. In contrast, we follow a decentralized approach, where reshufflers gather statistics on-the-fly while routing the data to joiners. Since reshufflers receive data that is randomly shuffled from the previous stages, the received local samples can be scaled by J to construct global cardinality estimates (Alg 1 lines 3,5). These estimates can be reinforced with statistical estimation theory tools [23] to provide confidence bounds. The advantages of this design are three-fold: *a*) A centralized entity for gathering statistics is no longer required, removing a source of potential bottlenecks. Additionally, it precludes any exchange communication or synchronization overheads. *b*) This model can be easily extended to monitor other data statistics, e.g., frequency histograms. *c*) The design supports fault tolerance and state reconstruction. When a reshuffler or a controller task fails, any other task can take over.

4.2 Analysis and Planning

Given that global statistics are constructed in Alg. 1, the controller is capable of analyzing the efficiency of the current mapping scheme, and thus, determining the overall performance of the operator. Furthermore, it checks for alternative (n, m)-mapping schemes that minimize the ILF (Alg 1 line 6). If it finds a better one, it triggers the new scheme. This affects the route of new tuples and impacts machine state. Adopting this dynamic strategy reveals three challenges that need careful examination: *a*) Since the controller is additionally a reshuffler task, it has the main duty of routing tuples in parallel to exploring alternative mappings. Thus, it has to balance between the ability to quickly react to new cardinality information against the ability to process new tuples rapidly (the classic *exploration-exploitation dilemma*). *b*) Migrating to a new mapping scheme requires careful state maintenance and transfer between machines. This incurs non-negligible overhead due to data transmission over the network. The associated costs of migration might outweigh the benefits if handled naïvely. *c*) An aggressively adaptive control system suffers from excessive migration overheads while a conservative system does not adapt well to data dynamics. Adaptivity thrashing might incur quadratic migration costs. Thus, the controller should be alert in choosing the moments for triggering migrations.

In this section, we describe a constant-competitive algorithm that decides when to *explore* and *trigger* new schemes such that the total cost of communication, including adaptation, is amortized linear.

4.2.1 1.25-Competitive Online Algorithm

Alg. 2 decides the time points that explore and trigger migration decisions. Right after an optimal migration, the system has $|R|$ and $|S|$ tuples from the respective relations. The algorithm maintains two counts $|\Delta R|$ and $|\Delta S|$, denoting the newly arriving tuples on both relations respectively after the last migration. If either $|\Delta R|$ reaches $|R|$ or $|\Delta S|$ reaches $|S|$, the algorithm explores alternative mapping schemes and performs a migration, if necessary.

The two metrics of interest here are the ILF and the migration overhead. The aim of this section is to demonstrate the following key result.

THEOREM 4.1. *Assume that the number of joiners J is a power of two, the sizes for $|R|$ and $|S|$ are no more than a factor of J apart, and that tuples from R and S have the same size. For a system applying Alg. 2, the following holds:*

1. *The ILF is at most* 1.25 *times that of the optimal mapping at any point in time. $ILF \leq 1.25 \cdot ILF^*$, where ILF^* is the input-load factor under the optimal mapping. Thus, the algorithm is* 1.25*-competitive.*

2. *The total communication overhead of migration is amortized, i.e., the cost of routing a new input tuple, including its migration overhead, is $O(1)$.*

Input-Load Factor. We hereby analyze the behavior of the ILF under the proposed algorithm. Since we assume that $size(r) = size(s)$, it follows that minimizing the ILF is equivalent to minimizing $(|R|/n + |S|/m)$.

LEMMA 4.1. *If J is a power of two and it holds that $1/J \leq |R|/|S| \leq J$, then under an optimal mapping (n, m),*

$$\frac{1}{2}\frac{|S|}{m} \leq \frac{|R|}{n} \leq 2\frac{|S|}{m} \quad \text{and} \quad \frac{1}{2}\frac{|R|}{n} \leq \frac{|S|}{m} \leq 2\frac{|R|}{n}.$$

PROOF. An optimal mapping minimizes $|R|/n + |S|/m$, under the restriction that $n \cdot m = J$. This happens when $|R|/n$ and $|S|/m$ are closest to each other. Since J is a power of two, by assumption, (and also n and m), it follows that under the optimal mapping $|R|/n \leq 2|S|/m$. Assume it were not the case, then $|R|/n > 2|S|/m$. Under the mapping $(2n, m/2)$, both $|R|/n$ and $|S|/m$ are closer, yielding a lower input-load factor, contradicting the optimality of (n, m). Choosing such a mapping is possible, assuming that $1/J \leq |R|/|S| \leq J$. The other inequality is symmetric. □

This lemma is useful in proving all subsequent results. The first important result is that the ILF is within a constant factor from that of the optimal scheme. This is due to the fact that Alg. 2 does not allow the operator to receive many tuples without deciding to recalibrate. The following theorem formalizes this intuition.

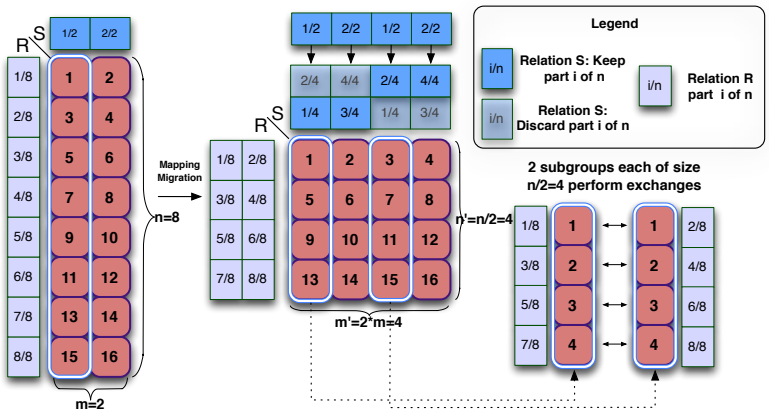


Figure 3: Migration from a $(8, 2)$- to a $(4, 4)$-mapping. Discards are performed on the state of stream S and exchanges are performed on the state of stream R.

Algorithm 2 Migration Decision Algorithm.

Input: $|R|$, $|S|$, $|\Delta R|$, $|\Delta S|$

1: **function** MigrationDecision($|R|$, $|S|$, $|\Delta R|$, $|\Delta S|$)
2: **if** $|\Delta R| \geq |R|$ or $|\Delta S| \geq |S|$ **then**
3: Choose mapping (n, m) minimizing $\frac{|R|}{n} + \frac{|S|}{m}$
4: Decide a migration to (n, m)
5: $|R| \leftarrow |R| + |\Delta R|; |S| \leftarrow |S| + |\Delta S|$
6: $|\Delta R| \leftarrow 0; |\Delta S| \leftarrow 0$
7: **end function**

Lemma 4.2. *If $|\Delta R| \leq |R|$ and $|\Delta S| \leq |S|$ and (n, m) is the optimal mapping for $(|R|, |S|)$ tuples, then the optimal mapping for $(|R| + |\Delta R|, |S| + |\Delta S|)$ is one of (n, m), $(n/2, 2m)$, and $(2n, m/2)$.*

Proof. Without loss of generality, assume that $|\Delta S| \geq |\Delta R|$. It must be that an optimal mapping will not decrease m (since $|S|$ grew relative to $|R|$). Therefore, the optimal is one of (n, m), $(n/2, 2m)$, $(n/4, 4m)$, ..., etc. To prove that the optimum is either (n, m) or $(n/2, 2m)$, it is sufficient to prove the following inequality

$$\frac{|R| + |\Delta R|}{n/2} + \frac{|S| + |\Delta S|}{2m} \leq \frac{|R| + |\Delta R|}{n/4} + \frac{|S| + |\Delta S|}{4m}$$
$$\frac{|S| + |\Delta S|}{m} \leq \frac{8(|R| + |\Delta R|)}{n}$$

which means that the ILF under an $(n/2, 2m)$-mapping is smaller than that under an $(n/4, 4m)$-mapping. This holds because $|S|/m \leq 2|R|/n$ (lemma 4.1), even if $|\Delta S| = |S|$ and $|\Delta R| = 0$. The case $|\Delta R| \geq |\Delta S|$ is symmetric. □

Alg. 2 decides migration once $|\Delta R| = |R|$ or $|\Delta S| = |S|$. Therefore, lemma 4.2 implies that while the system is operating with the mapping (n, m), the optimum is one of (n, m), $(n/2, 2m)$, and $(2n, m/2)$. This implies the following.

Lemma 4.3. *If $|\Delta R| \leq |R|$ and $|\Delta S| \leq |S|$ and (n, m) is the optimal mapping for $(|R|, |S|)$ tuples, then under Alg. 2, the input-load factor ILF never exceeds $1.25 \cdot ILF^*$. In other words, the algorithm is 1.25-competitive.*

Proof. By lemma 4.2, the optimal mapping is either (n, m), $(n/2, 2m)$ or $(2n, m/2)$. If the optimal mapping is (n, m) then $ILF = ILF^*$. Otherwise, the ratio can be bounded as follows. Without loss of generality, assume that the optimum is $(n/2, 2m)$ then

$$\frac{ILF}{ILF^*} \leq \frac{(|R| + |\Delta R|)/n + (|S| + |\Delta S|)/m}{(|R| + |\Delta R|)/(n/2) + (|S| + |\Delta S|)/(2m)}$$

where the constraints $|\Delta R|/n \leq |R|/n$, $|\Delta S|/m \leq |S|/m$ and those in lemma 4.1 must hold. All cardinalities are non-negative. Consider the ratio as a function of the variables $|R|/n$, $|S|/m$, $|\Delta R|/n$ and $|\Delta S|/m$. The maximum value of the ratio of linear functions in a simplex (defined by the linear constraints) is attained at a simplex vertex. By exhaustion, the maximum occurs when $|\Delta R| = 0$, $|\Delta S| = |S|$ and $|S|/m = 2|R|/n$. Substituting gives 1.25. □

Migration Overhead. It remains to show that, under the described algorithm, the migration overhead is amortized. This requires showing that the migration process can be done efficiently and that when a migration is triggered, enough tuples are received to "pay" for this migration cost.

The migration of interest is the change from the (n, m) to $(n/2, 2m)$-mapping (symmetrically, (n, m) to $(2n, m/2)$). Migration can be done naïvely by repartitioning all previous states around the joiners according to the new scheme. This approach unnecessarily congests the network and is expensive. In contrast, we present a *locality*-aware migration mechanism that minimizes state transfer overhead. To illustrate the procedure, we walk through an example. Consider a migration from a $(8, 2)$ to a $(4, 4)$-mapping scheme $(J = 16)$ as depicted in Fig. 3. Before the migration, each joiner stores about an eighth of R and half of S. After the migration, each joiner stores a quarter of R and only one quarter of S. To adapt, joiners can efficiently and deterministically discard a quarter of S (half of what they store). However, tuples of R must be exchanged. In Fig. 3, joiners 1 and 2 store the "first" eighth of R while joiners 3 and 4 store the "second" eighth of R. Joiners 1 and 3 can exchange their tuples and joiners 2 and 4 can do the same in parallel. Joiners 5 and 7, 6 and 8, and so forth operate similarly in parallel. This incurs a total overhead of $|R|/4$ time units which is the bi-directional communication cost of $|R|/8$. This idea can be generalized, yielding bounds on the migration overhead.

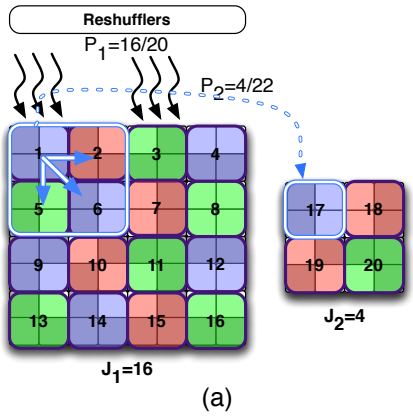


Figure 4: (a) decomposing $J = 20$ machines into independent groups of 16 and 4 machines. Tuple storage within a group is defined by the probability measures.

LEMMA 4.4. *Migration from (n, m) to $(n/2, 2m)$-mapping can be done with a communication cost of $2|R|/n$ time units. Similarly, migrating to $(2n, m/2)$ incurs a cost of $2|S|/m$.*

PROOF. Without loss of generality, consider the migration to $(n/2, 2m)$. No exchange of S state is necessary. On the other hand, tuples of R have to be exchanged among joiners. Before migration each of the J joiners had $|R|/n$ tuples from R, while after the migration, each must have $2|R|/n$. Consider one group of n joiners sharing the same tuples from S (corresponding to a "column" in Fig. 3). These joiners, collectively, contain the entire state of R. They can communicate in parallel with the other $m-1$ groups. Therefore, we analyze the state relocation for one such group and it follows that all groups behave similarly in parallel.

Divide the group into two subgroups of $n/2$ joiners. Number the joiners in each group $1, 2, \ldots, n/2$. Joiner pairs labeled i should exchange their tuples together. It is clear that each pair of joiners labeled i ends up with a distinct set of $2|R|/n$ tuples. Fig. 3 describes this exchange process. Each of the pairs labeled i can communicate completely in parallel. Therefore, the total migration overhead is $2|R|/n$, since each joiner in the pair sends $|R|/n$ tuples to the other. □

LEMMA 4.5. *The cost of routing tuples and data migration is linear. The amortized cost of an input tuple is $O(1)$.*

PROOF. Since all joiners are symmetrical and operate simultaneously in parallel, it suffices to analyze cost at one joiner. Therefore, after receiving $|\Delta R|$ and $|\Delta S|$ tuples, the operator spends *at least* $\max(|\Delta R|/n, |\Delta S|/m)$ units of time processing these tuples at the appropriate joiners. By assigning a sufficient amortized cost per time unit, the received tuples pay for the later migration.

By lemma 4.2, the optimal mapping is (n, m), $(n/2, 2m)$ or $(2n, m/2)$. If the optimal mapping is (n, m), then there is no migration. Without loss of generality, assume that $|\Delta S| \geq |\Delta R|$ and that the optimal mapping is $(n/2, 2m)$. Between migrations, $\max(|\Delta R|/n, |\Delta S|/m)$ time units elapse, each is charged 7 units. One unit is used to pay for routing and 6 are reserved for the next migration. The cost of migration by lemma 4.4 is $2(|R|+|\Delta R|)/n$. The amortized cost reserved for migration is $6\max(|\Delta R|/n, |\Delta S|/m)$. Since a migration was triggered, either $|\Delta R| = |R|$ or $|\Delta S| = |S|$. In either case, it should hold that the reserved cost is at least the migration cost, that is,

$$6\max(|\Delta R|/n, |\Delta S|/m) \geq 2(|R| + |\Delta R|)/n.$$

If $|\Delta R| = R$, then by substituting, the left hand side is $6\max(|\Delta R|/n, |\Delta S|/m) \geq 6|R|/n$ and the right hand side is $2(|R| + |\Delta R|)/n = 4|R|/n$. Therefore, the inequality holds. If $|\Delta S| = S$, then the left hand side is

$$6\max(|\Delta R|/n, |\Delta S|/m) \geq 2|\Delta R|/n + 4|S|/m.$$

Therefore, the left hand side is not smaller than the right hand side, since $2|S|/m \geq |R|/n$ (by lemma 4.1). Thus, the inequality holds in both cases. The cases, when $|\Delta R| \geq |\Delta S|$ or when the optimal is $(2n, m/2)$, are symmetric. □

Lemmas 4.3 and 4.5 directly imply Theorem 4.1.

4.2.2 Generalization and Discussion

In the previous section, the analysis was based upon three assumptions: the cardinality ratio of the larger relation to the smaller relation does not exceed J; the number of joiners is a power of two; and tuples from R and S have the same size. In this section we outline how to relax these assumptions and show that the algorithm remains constant-competitive and the migration overhead persists to be amortized and linear to the number of input tuples.

Relation cardinality ratio. Without loss of generality, assume that $|R| > |S|$. The analysis in the previous section assumed that $|R| \leq J|S|$. This can be relaxed by continuously padding the smaller relation with dummy tuples to maintain the ratio less than J. This requires padding the relation S with at most $|R|/J \leq T/J$ tuples, where T is the total number of tuples $|R| + |S|$. Therefore, the total number of tuples the operator handles, including dummy tuples, is at most $T + T/J = (1 + 1/J)T$ tuples. The ratio of the relation sizes still respects the assumption. Therefore, the analysis in the previous section holds except that the ILF now gets multiplied by a factor of $1 + 1/J$. This factor is at most 1.5 (since $J \geq 2$). This factor *tends to one* as the number of joiners increases. Therefore, the algorithm is still constant-competitive, with the constant being $1.25 \cdot 1.5 = 1.875$. Similarly, adding the dummy tuples multiplies the migration overhead by at most 1.5. Therefore, the communication overhead remains linear.

Number of joiners. Assume that $J \in \mathbb{N}^+$, then J has a unique decomposition into a sum of powers of two. Let $J = J_1 + J_2 + \ldots + J_c$ where each J_i is a power of two. Accordingly, the machines are broken down into c groups, where group i has J_i machines. There can be at most $\lceil \log J \rceil$ of such groups. Finally, each group operates exactly as described in the previous section. Fig. 4a illustrates an example, given a pool of $J = 22$ machines, it is clustered into three groups of sizes 16, 4 and 2 which operate independently. An incoming tuple is sent to all c groups to be joined with all stored tuples. Only one group stores this tuple for joining with future tuples. The group that stores this tuple is determined by computing a pseudo-random hash whose ranges are proportional to the group sizes. The probability that group i is chosen is equal to $P_i = J_i/J$. With high probability, after T tuples have been received, the number of tuples stored in group i is close to $(J_i/J)T$.

It is essential that if a pair of tuples are sent to two machines, each belonging to different groups, that this pair of tuples is received in the same order by both machines. With very high probability (after a small number of tuples has been received), the mappings of two groups will be similar.

More specifically, for two groups with sizes $J_1 < J_2$, it will hold that n_2 (m_2) is divisible by n_1 (m_1). Blocks of machines in the bigger group correspond to a single machine in the smaller group (see figure 4). In each such block, a single machine does the task of forwarding all tuples to machines within that block as well as the machine in the smaller group (see the same figure). This ensures that machines get tuples in the same order at the cost of tuple latency proportional to $\log J$, since tuples have to be propagated serially among $\log J$ groups of machines.

Let the biggest group be L with size J' which is at least half of J. The storage is bounded by that of L (receiving the entire input). The optimal storage is at most half that of L (since J' is at least half of J). Therefore, the competitive ratio of storage is at most doubled (3.75). Since groups operate independently, migrations are performed asynchronously and completely in parallel. Therefore, only tuple routing gets multiplied by a $\log J$ factor, since every tuple is broadcast to at most $\log J$ groups. Therefore, the total routing cost, including migrations, is $O(T \log J)$.

It remains to show that the described distribution of data does not affect the original configuration that all joiners perform an equal amount of join work. Without loss of generality, consider two tuples t_R and t_S where t_R arrives to the system before t_S (the other case is symmetric). We show that the probability a specific joiner j computes $\{t_R\} \bowtie \{t_S\}$ is $1/J$, implying directly that the work gets equally distributed. For joiner j to perform the join, t_R has to be stored on j. The probability of this happening is $(J_g/J) \cdot (1/n_g)$ where J_g is the group size of group g containing joiner j and n_g is the number of rows in the mapping of this group. t_S gets communicated to all groups. The probability that t_S is sent to j is exactly $1/m_g$ where m_g is the number of columns in the mapping of group g. Multiplying both probabilities and noticing that $n_g \cdot m_g = J_g$ gives exactly $1/J$.

Optimality-Communication tradeoff. It is possible to modify Alg. 2 to tradeoff the mapping optimality with the communication overhead. The algorithm checks for the possibility of performing migration whenever either $|\Delta R| = |R|$ or $|\Delta S| = |S|$. By modifying these conditions to be $|\Delta R| = \epsilon |R|$ or $|\Delta S| = \epsilon |S|$, where $0 < \epsilon \leq 1$, we directly get a tradeoff between optimality and communication cost.

THEOREM 4.2. *Under modified Alg. 2 (parameterized by ϵ), the competitive ratio of the ILF becomes $\frac{3+2\epsilon}{3+\epsilon}$ and the amortized communication cost becomes $\frac{8}{\epsilon} = O(\frac{1}{\epsilon})$.*

PROOF. The proof is exactly following the lemmas of subsection 4.2.1 and replacing the conditions $|\Delta R| \leq |R|$ and $|\Delta S| \leq |S|$ by $|\Delta R| \leq \epsilon |R|$ and $|\Delta S| \leq \epsilon |S|$, respectively. The competitive ratio is given by the following expression:

$$\frac{ILF}{ILF^*} \leq \frac{(|R| + |\Delta R|)/n + (|S| + |\Delta S|)/m}{(|R| + |\Delta R|)/(n/2) + (|S| + |\Delta S|)/(2m)}$$

This attains its maximum value $\frac{3+2\epsilon}{3+\epsilon}$ at $|\Delta R| = 0$, $|\Delta S| = \epsilon |S|$ and $|S|/m = 2|R|/n$.

For every input tuple, an amortized cost of $3 + 4/\epsilon$ is given. Between migrations, at least $\max(|\Delta R|/n, |\Delta S|/m)$ are received. Without loss of generality, the migration cost is $2\frac{|R|+|\Delta R|}{n}$. If $|\Delta R| = \epsilon |R|$, substituting shows that the amortized cost exceeds the migration cost. In the case of $|\Delta S| = \epsilon |S|$, substituting and noting that $\frac{S}{m} \geq \frac{R}{2m}$ (by lemma 4.1), it also holds that the total migration cost is less than the amortized cost. The theorem statement immediately follows. □

Notice that by setting $\epsilon = 1$, the proven bounds are recovered as a special case of this theorem.

Elasticity. In the context of online query processing, the query planner may be unable to a-priori determine the number of machines J to be dedicated to a join operator. It is thus desirable to allocate as few joiners as possible to the operator while ensuring that the stored state on each machine is reasonably maintained to prevent disk spills and performance degradation. We hereby present a scheme that allows the join operator to elastically expand using more machines, as needed, while maintaining all the theoretical bounds described (merely constant changes in the communication cost).

For joiners, designate a maximum number M of tuples (ILF) per joiner. At migration checkpoints (following theorem 4.2 when $|\Delta R| = \epsilon |R|$ or $|\Delta S| = \epsilon |S|$), after migration, if each joiner stores a number of tuples exceeding $M/2$, the system expands by splitting every joiner into 4 joiners. Every joiner communicates its tuples to three new joiners as described in Fig. 5. This can be done with a total communication cost equal to twice the number of tuples stored on that joiner prior to expansion.

Under this scheme, it is obvious that the competitive ratio of the ILFis unaffected, since splitting every machine to four machines does not change the ratio of n to m. It remains to show that the amortized cost of communication is not much affected.

THEOREM 4.3. *Under modified Alg. 2 (parameterized by ϵ), the described expansion has an amortized cost of $\frac{8}{\epsilon} = O(1/\epsilon)$.*

PROOF. After receiving $|\Delta R|$ and $|\Delta S|$ tuples, the operator spends at least $\max(|\Delta R|/n, |\Delta S|/m)$ units of time processing these tuples at the appropriate joiners. Each is assigned an amortized cost of $4 + 4/\epsilon \leq 8/\epsilon$. The communication cost due to expansion is at most $2(\frac{|R|+|\Delta R|}{n} + \frac{|S|+|\Delta S|}{m})$. $4\max(|\Delta R|/n, |\Delta S|/m)$ is used to account for $2|\Delta R|/n + |\Delta S|/m$. It remains to notice that $\frac{4}{\epsilon}\max(|\Delta R|/n, |\Delta S|/m) \geq 2(|R|/n + |S|/m)$ since either $|\Delta R| = \epsilon |R|$ or $|\Delta S| = \epsilon |S|$ and since $\frac{1}{2}\frac{|R|}{n} \leq \frac{|S|}{m} \leq 2\frac{|R|}{n}$ (by lemma 4.1). □

Relative tuple sizes. Let the sizes of an R tuple and an S tuple be τ_R and τ_S respectively. An input R tuple can be viewed as the reception of τ_R "unit" tuples. Similarly an S tuple is τ_S unit tuples. The previous analysis holds except that migration decisions can be slightly delayed. For example, if the migration decision is supposed to happen after the reception of 5 unit tuples and a tuple of size 1000 units is received, then the migration decision is delayed by 995 units. Therefore, the ILF is increased by at most an additive factor of $\max(\tau_R, \tau_S)$, i.e., $ILF \leq K \cdot ILF^* + \max(\tau_R, \tau_S)$.

4.3 Actuation

The previous section provides a high-level conceptual description of the algorithm. Migration decision points are specified to guarantee a close-to-optimal ILF and linear amortized adaptivity cost. This section describes the system-level implementation of the migration process.

Previous work on designing adaptive operators [34, 26, 31] follow a general theme for state relocation. The following steps give a brief description of the process: (i) Stall the input to the machines that contain state to be repartitioned. The new input tuples are buffered at the data sources. (ii) Machines wait for all in-flight tuples to arrive and be processed. (iii) Relocate state. (iv) Finally, online

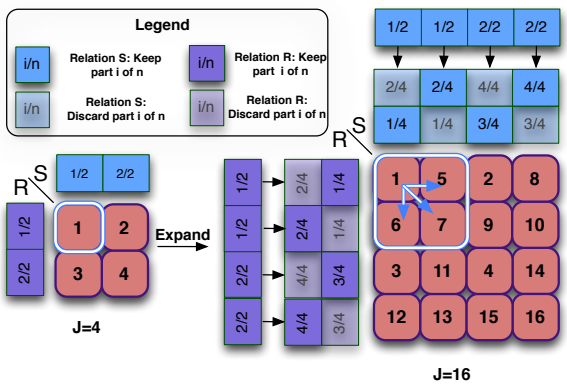


Figure 5: This figure illustrates the expansion of the system. Each machine state is distributed to 4 joiners. Each joiner communicates the appropriate portions of its state to the three new joiners. For example, joiner 1 sends the second half of its S tuples to joiners 5 and 7. It sends the second half of its R tuples to 6 and 7. It also sends the first half of R to 5 and the first half of S to 6.

processing resumes. Buffered tuples are redirected to their new location to be processed. This protocol is not suitable for stateful operators. Its blocking behavior causes lengthy stalls during online processing until state relocation ends.

4.3.1 Eventually Consistent Protocol

It is essential for the operator to continue processing tuples *on-the-fly* while performing adaptations. Achieving this presents new challenges to the correctness of the results. When the operator migrates from one partitioning scheme $\mathcal{M}_i$ to another $\mathcal{M}_{i+1}$ it undergoes a state relocation process. During this, the state of each machine, within the operator, does not represent a state that is consistent with either $\mathcal{M}_i$ or $\mathcal{M}_{i+1}$. Hence, it becomes hard to reason about how new tuples entering the system should be joined. This section presents a non-blocking protocol that allows continuous processing of new tuples during state relocation by reasoning about the state of any tuple circulating the system with the help of *epochs*. This ensures that the system (i) is consistent at all times except during migration, (ii) eventually converges to the consistent target state $\mathcal{M}_{i+1}$, and (iii) produces correct and complete join results in a continuous manner. The operation of the system is divided into *epochs*. Initially, the system is in epoch zero. Whenever the controller decides a mapping change, the system enters a new epoch with incremented index. For example, if the system starts with the mapping $(8, 8)$, later migrates to $(16, 4)$ and finally migrates to $(32, 2)$, the system went through exactly three epochs. All tuples arriving between the first and the second migration decision belong to epoch 1. All tuples arriving after the last mapping-change decision belong to epoch 2. Reshufflers and joiners are not instantaneously aware of the epoch change, but continue to process tuples normally until they receive an epoch change signal along with the new mapping. Whenever a reshuffler routes a tuple to joiners, it tags it with the latest epoch number it is aware of. It is crucial for the correctness of the scheme described shortly to guarantee that all machines are at most one epoch behind the controller. That is, all machines operate on, at most, two different epochs. This is, however, guaranteed theoretically and formalized later in Theorem 4.6.

The migration starts by the controller making the decision. The controller broadcasts to all reshufflers the mapping change signal. When a reshuffler receives this signal, it notifies all joiners and immediately starts sending tuples in accordance to the new mapping. Joiners continuously join incoming tuples and start exchanging migration tuples. Once a joiner has received epoch change signals from *all* reshufflers, it is guaranteed that it will receive no further tuples tagged with the old epoch index. At that point, the joiner proceeds to finalize the migration and notifies the controller once it is done. The controller can only start a new migration once all joiners notify it that they finished the data migration. The subsequent discussion shows how joiners continue processing tuples while guaranteeing consistent state and correct output.

The timestamp of the migration decision at the controller partitions the tuples into several sets. During a migration, τ is the set of all tuples received before the migration decision. μ is the set of all tuples that are sent from one joiner to another (due to migration). The set of new tuples received after the migration decision timestamp are either tagged with the old epoch index, referred to as Δ, or with the new epoch index, referred to as Δ'. Notice that $\mu \subset (\tau \cup \Delta)$. To simplify notation, no distinction is made between tuples of R or S. For example, writing $\Delta \bowtie \Delta'$ refers to $(\Delta_R \bowtie \Delta'_S) \cup (\Delta_S \bowtie \Delta'_R)$, where σ_R (σ_S) refers to the tuples of $R(S)$ in the set σ.

During the migration, joiners have tuples tagged with the old epoch and the new epoch. Those tuples tagged with the new epoch are already on the correct machines since the reshuffler sent them according to the new mapping. Joiners should redistribute the tuples tagged with old labels according to the new mapping. The set of tuples tagged with the old label is exactly $\tau \cup \Delta$. Joiners discard portions and communicate other portions to the other machines. The discarded tuples are referred to as $\textsc{Discard}(\tau \cup \Delta)$. For convenience, $(\tau \cup \Delta) - \textsc{Discard}(\tau \cup \Delta)$ is referred to as $\textsc{Keep}(\tau \cup \Delta)$. The migrated tuples are $\textsc{Migrated}(\tau \cup \Delta)$ which coincides exactly with μ. $\textsc{Keep}(\tau)$ refers to tuples in $\textsc{Keep}(\tau \cup \Delta) \cap \tau$. The same holds for $\textsc{Discard}$, $\textsc{Migrated}$ and the set Δ.

Definition 4.4. *A migration algorithm is said to be correct if right after the completion of a migration, the output of the system is exactly* $(\tau \cup \Delta \cup \Delta') \bowtie (\tau \cup \Delta \cup \Delta')$.

During the migration, the output may be incomplete. Therefore, completeness and consistency are defined only upon the completion of the migration. The complete output is the join of all tuples that arrived to the system before (τ) and after the migration decision $(\Delta \cup \Delta')$. Alg. 3 describes the joiner algorithm. The output of the algorithm is provably correct. For the proof of correctness, an alternative characterization of the correct output is needed.

Lemma 4.6.

$$(\tau \cup \Delta \cup \Delta') \bowtie (\tau \cup \Delta \cup \Delta')$$

is equivalent to the union of (1) $\tau \bowtie \tau$*, (2)* $\Delta \bowtie \Delta$*, (3)* $\tau \bowtie \Delta$*, (4)* $\Delta' \bowtie \mu$*, (5)* $\Delta' \bowtie \textsc{Keep}(\Delta)$*, (6)* $\Delta' \bowtie \textsc{Keep}(\tau)$*, and (7)* $\Delta' \bowtie \Delta'$.

Proof. Since set union distributes over join, the result can be rewritten as,

$$(\tau \bowtie \tau) \cup (\tau \bowtie \Delta) \cup (\tau \bowtie \Delta') \cup (\Delta \bowtie \Delta) \cup (\Delta \bowtie \Delta') \cup (\Delta' \bowtie \Delta').$$

Algorithm 3 Joiner-Epoch Algorithm.

Input: s signal
Initialize: Use HANDLETUPLE$_1$ to handle incoming tuples.
1: **procedure** MAIN(s)
2: **if** First Reshuffler Signal Received **then**
3: SEND τ for migration.
4: **else if** All Reshuffler Signals Received **then**
5: Use HANDLETUPLE$_2$ to handle incoming tuples.
6: **else if** Migration Ended **then**
7: Run FINALIZEMIGRATION.
8: Use HANDLETUPLE$_1$ to handle incoming tuples.

Input: t an incoming tuple
9: **procedure** HANDLETUPLE$_1$(t)
10: **if** $t \in \mu$ **then** OUTPUT
11: $\{t\} \bowtie \Delta'$; $\mu \leftarrow \mu \cup \{t\}$
12: **else if** $t \in \Delta'$ **then**
13: OUTPUT $\{t\} \bowtie (\mu \cup \Delta')$; $\Delta' \leftarrow \Delta' \cup \{t\}$
14: OUTPUT $\{t\} \bowtie$ KEEP$(\tau \cup \Delta)$
15: **else if** $t \in \Delta$ **then**
16: OUTPUT $\{t\} \bowtie (\tau \cup \Delta)$; $\Delta \leftarrow \Delta \cup \{t\}$
17: **if** $t \in$ KEEP(Δ) **then**
18: OUTPUT $\{t\} \bowtie \Delta'$
19: **if** $t \in$ MIGRATED(Δ) **then**
20: SEND $\{t\}$ for migration

Input: t an incoming tuple
21: **procedure** HANDLETUPLE$_2$(t)
22: **if** $t \in \mu$ **then**
23: OUTPUT $\{t\} \bowtie \Delta'$; $\mu \leftarrow \mu \cup \{t\}$
24: **else if** $t \in \Delta'$ **then**
25: OUTPUT $\{t\} \bowtie (\mu \cup \Delta')$; $\Delta' \leftarrow \Delta' \cup \{t\}$
26: OUTPUT$\{t\} \bowtie$ KEEP$(\tau \cup \Delta)$

27: **procedure** FINALIZEMIGRATION
28: SEND`(Ack)` signal to coordinator
29: $\tau \leftarrow$ KEEP$(\tau \cup \Delta) \cup \mu \cup \Delta'$
30: $\Delta \leftarrow \emptyset$; $\Delta' \leftarrow \emptyset$; $\mu \leftarrow \emptyset$

Subsets (1), (2), (3) and (7) appear directly in the expression. It remains to argue that $\Delta' \bowtie (\tau \cup \Delta)$ is equal to $\Delta' \bowtie (\mu \cup$KEEP$(\tau \cup \Delta))$. This follows directly from the correctness of the migration. $\tau \cup \Delta$ is the set of tuples labeled with the old epoch, while $(\mu \cup$KEEP$(\tau \cup \Delta))$ is the same set distributed differently between the machines according to the new mapping. □

Alg. 3 exploits this equivalence to continue processing tuples throughout migration. Informally, parts (1), (2) and (3) are continuously computed in HANDLETUPLE$_1$ whereas, (4), (5), (6) and (7) are continuously computed in both HANDLETUPLE$_1$ and HANDLETUPLE$_2$.

THEOREM 4.5. *Alg. 3 produces the correct and complete output and ensures eventually consistent state for all joiners.*

PROOF. First, it is easy to see that the data migration is performed correctly. τ is sent immediately at the very beginning (line 3). Tuples of Δ are sent as they are received (line 20). Finally, the discards are done once the migration is over (line 29). By lemma 4.6, the result is the union of:

1. $\tau \bowtie \tau$. This is computed prior to the start of migration.

2. $(\Delta \bowtie \Delta) \cup (\tau \bowtie \Delta)$. Δ is initially empty. Tuples are only added to it in line 16. Every added tuple gets joined with all previously added tuples to Δ and to all tuples in τ (also in line 16). It follows that this part of the join is computed. τ never changes until the migration is finalized.

3. $\Delta' \bowtie (\mu \cup$KEEP$(\tau \cup \Delta))$. Whenever a tuple is added to Δ' (in lines 13 and 25), it gets joined with $\mu \cup$KEEP$(\tau \cup \Delta)$ (lines 13, 14, 25 and 26). Whenever a tuple is added to μ (lines 11 and 23), it gets joined with Δ'. Furthermore, tuples added to Δ are joined with Δ' if they are in KEEP(Δ) (line 18). τ never changes until the migration ends.

4. $\Delta' \bowtie \Delta'$. Initially Δ' is empty. Tuples get added to it in lines 13 and 25. Whenever a tuple gets added, it gets joined with all previously added tuples (lines 13 and 25).

Therefore, all parts are computed by the algorithm (completeness). Since the analysis covers all the lines that perform a join, it follows that each of the 4 parts of the result is output exactly once (correctness). Thus, the result of the algorithm is correct right after migration is complete. Tuples tagged with the old epoch (τ and Δ) are migrated correctly. Tuples tagged with the new epoch (Δ') are immediately sent to machines according to the new scheme. Therefore, at the end of migration, the state of all joiners is consistent with the new mapping. □

4.3.2 Theoretical Guarantees Revisited

The guarantees given in Theorem 4.1 assume a blocking operator. During migration, it is required that no tuples are received or processed. However, Alg. 3 continuously processes new tuples while adapting. We set the joiners to process migrated tuples at twice the rate of processing new incoming tuples. We show that, under these settings, the proven guarantees hold. It is clear that the amortized cost is unchanged and remains linear because incoming tuples continue to "pay" for future migration costs. The results for competitiveness, on the other hand, need to be verified.

THEOREM 4.6. *With the non-blocking scheme Alg. 3, the competitive ratio ensured by Theorem 4.1 remains* 1.25[1].

PROOF. We prove that the numbers of tuples, received during migration, $|\Delta R|$ and $|\Delta S|$, are bounded by $|R|$ and $|S|$, respectively. 1.25-competitiveness follows immediately (by lemma 4.3).

Consider a migration decision after the system has received $|R|$ and $|S|$ tuples from R and S. Let the current mapping be (n, m). Lemma 4.2 asserts that the optimal mapping is one of (n, m), $(n/2, 2m)$ and $(2n, m/2)$. This is trivially true for the first migration. Since we prove below that $|\Delta R|$ and $|\Delta S|$ are bounded by $|R|$ and $|S|$, this also holds for all subsequent migrations, inductively. Without loss of generality, let the chosen optimal mapping for a subsequent migration be $(n/2, 2m)$. The migration process lasts for $2|R|/n$ time units (by lemma 4.4). Alg. 3 processes new tuples at half the rate of processing migrated tuples. Thus, during migration, the operator receives at most $1/2 \cdot (n/2)$ new tuples from R and $1/2 \cdot (2m)$ from S per time unit. Hence, it holds that,

$$|\Delta R| \leq \tfrac{2|R|}{n} \cdot \tfrac{n}{4} < |R| \text{ and } |\Delta S| \leq \tfrac{2|R|}{n} \cdot m \leq \tfrac{|S|}{m} \cdot m = |S|$$

where the last inequality holds by lemma 4.1 (with the optimal being $(n/2, 2m)$ instead of (n, m)). □

[1]Notice that Theorem 4.6 is based on the assumptions made in Theorem 4.1. However, it naturally follows, that if any of the assumptions are relaxed the competitive ratio is changed accordingly as described in §4.2.2.

Query	Join	Predicate
E_{Q_5}	(R⋈N⋈S)⋈L	Equi-join
E_{Q_7}	(S⋈N)⋈L	Equi-join
B_{NCI}	L⋈L	Band-join
B_{CI}	L⋈L	Band-join

Table 1: R, N, S, and L correspond to the relations Region, Nation, Supplier, and Lineitem respectively as defined in the TPC-H benchmark.

4.3.3 Towards Fault-tolerance

Although fault tolerance is orthogonal to the scope of this paper, this section outlines how to extend the presented dataflow operator to provide fault-tolerance using existing techniques. For topologies with arbitrary operators, FTOpt's [39] fault-tolerance protocol guarantees *exactly-once* semantics (no lost or duplicate tuples). We can easily extend our operator to follow the protocol such that the entire query plan provides end-to-end fault-tolerance. The protocol is established between any two communicating nodes (producer/consumer pairs) in the query plan by splitting the fault-tolerance responsibilities between them. When a consumer takes responsibility of a received tuple, it sends an acknowledgment to the producer. This frees the producer from replaying acknowledged tuples on failures. The consumer can fulfill its responsibility by checkpointing to stable storage. On the other hand, the producer is responsible for replaying unacknowledged tuples on failure. This protocol supports many-to-many producer/consumer relationships.

At a high level, when a node fails, it first recovers its state from the latest checkpoint. Because some tuples may have been processed successfully on a consumer, but their acknowledgment may not have reached the producer before its failure, the recovered node then communicates with the downstream and upstream nodes to identify which tuples to replay. For every communication pair, the consumer provides information about the last seen tuple, and the producer has to replay only the missing portion of the stream. This protocol can provide fault-tolerance during migration as well. The only additional consideration is that communication pairs may vary due to the different migrations, and hence, this information also needs to be preserved.

5. EVALUATION

Environment. Our experimental platform consists of an Oracle Blade 6000 Chassis with 10 Oracle X6270 M2 blade servers. Each blade has two Intel Xeon X5675 CPUs running at 3GHz, each with 6 cores and 2 hardware threads per core, 72GB of DDR3 RAM, 4 SATA 3 hard disks of 500GB each, and a 1Gbit Ethernet interface. All blades run Solaris 10, which offers Solaris Zones, a native resource management and containment solution. Overall, there are 220 virtual machines available exclusively for our experiments, each with its own CPU hardware thread and dedicated memory resources. There are $10 \cdot 2$ separate hardware threads for running instances of the host operating system.

Datasets. For the evaluation setup, we use the TPC-H benchmark [2]. We employ the TPC-H generator proposed by [11] to generate databases with different degrees of skew under the *Zipf* distribution. The degree of skew is adjusted by choosing a value for the *Zipf* skew parameter z. We experiment on five different skew settings Z_0, Z_1, Z_2, Z_3, Z_4 which correspond to $z = 0$, $z = 0.25, z = 0.5, z = 0.75$ and $z = 1.0$ respectively. We build eight databases with sizes $8, 10, 20, 40, 80, 160, 320$, and 640GB.

Queries. We experiment on four join queries, namely, two equi-joins from the TPC-H benchmark and two synthetic band-joins. The equi-joins, E_{Q_5} and E_{Q_7}, represent the most expensive join operation in queries Q_5 and Q_7 respectively from the benchmark. All intermediate results are materialized before online processing. Moreover, the two band-joins depict two different workload settings. *a*) B_{CI} is a *high*-selectivity join query that represents a computation-intensive workload, and *b*) B_{NCI} is a *low*-selectivity join query that corresponds to a *non*-computation-intensive workload. The output of B_{CI} is three orders of magnitude bigger than its input size, whereas the output of B_{NCI} is an order of magnitude smaller. Both join queries are described below and all query characteristics are summarized in Table 1.

B_{CI}:
```
SELECT *
FROM  LINEITEM L1, LINEITEM L2
WHERE ABS(L1.shipdate - L2.shipdate) <= 1
AND (L1.shipmode='TRUCK' AND L2.shipmode!='TRUCK')
AND L1.Quantity>45
```

B_{NCI}:
```
SELECT *
FROM  LINEITEM L1, LINEITEM L2
WHERE ABS(L1.orderkey - L2.orderkey) <= 1
AND (L1.shipmode='TRUCK' AND L2.shipinstruct='NONE')
AND L1.Quantity>48
```

Operators. To run the testbed, we implement SQUALL[2], a distributed online query processing engine built on STORM[3], Twitter's backend engine for data analytics. The engine is based on Java and runs on JRE v1.7. Throughout the discussion, we use four different dataflow operators: (i) STATICMID, a static operator with a fixed $(\sqrt{J}, \sqrt{J})$-mapping. This scheme assumes that both input streams have the same size and lies in the center of the (n, m)-mapping spectrum. (ii) DYNAMIC, our adaptive operator, initialized with the $(\sqrt{J}, \sqrt{J})$-mapping scheme. (iii) STATICOPT, another static operator with a fixed optimal mapping scheme. This requires knowledge about the input stream sizes before execution, which is practically *unattainable* in an online setting. (iv) SHJ, the parallel symmetric hash-join operator described in [19]. This operator can only be used for equi-join predicates and it is *content-sensitive* as it partitions data on the join key. STATICMID, assumes as a best guess, that the streams are equal in size; hence it has a square grid partitioning scheme, i.e., $(\sqrt{J}, \sqrt{J})$. Comparing against STATICOPT shows that our operator does not perform much worse than an omniscient operator with oracle knowledge about stream sizes, which are unknown beforehand. Joiners perform the local join in memory, but if it runs out of memory it begins spilling to disk. For this purpose, we integrated the operators with the back-end storage engine BERKELEYDB [29]. We first experimentally verify that, in case of overflow to disk, machines suffer from long delayed join evaluation and performance hits. Then, for a more fair comparison, we introduce more memory resources, such that all operations fit in memory if possible. The heap size of each joiner is set to 2GB. As indexes, joiners use balanced binary trees for band joins and hashmaps for equi-joins. Input data rates are set such that joiners are fully utilized.

5.1 Skew Resilience

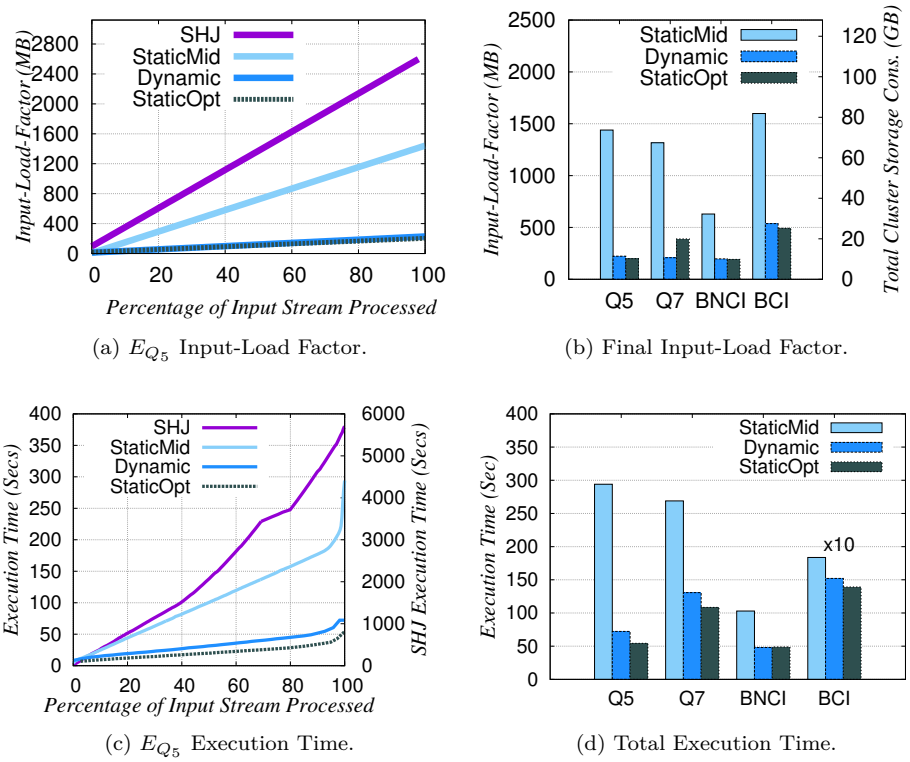
Table 2 shows results for running joins E_{Q_5} and E_{Q_7} with different skew settings of the 10G dataset. It compares the

[2]https://github.com/epfldata/squall/wiki
[3]https://github.com/nathanmarz/storm

	E_{Q_5}			E_{Q_7}		
Zipf	SHJ	DYNAMIC	STATICMID	SHJ	DYNAMIC	STATICMID
$Z=0$	79	168	838*	98	192	210
$Z=1$	79	176	851*	159	183	301
$Z=2$	2742*	158	1425*	191	369	462
$Z=3$	4268*	212	2367*	5462*	334	2610*
$Z=4$	5704*	203	2849*	6385*	415	3502*

Note: [*] Overflow to disk.

Table 2: Runtime in secs.



(a) E_{Q_5} Input-Load Factor.

(b) Final Input-Load Factor.

(c) E_{Q_5} Execution Time.

(d) Total Execution Time.

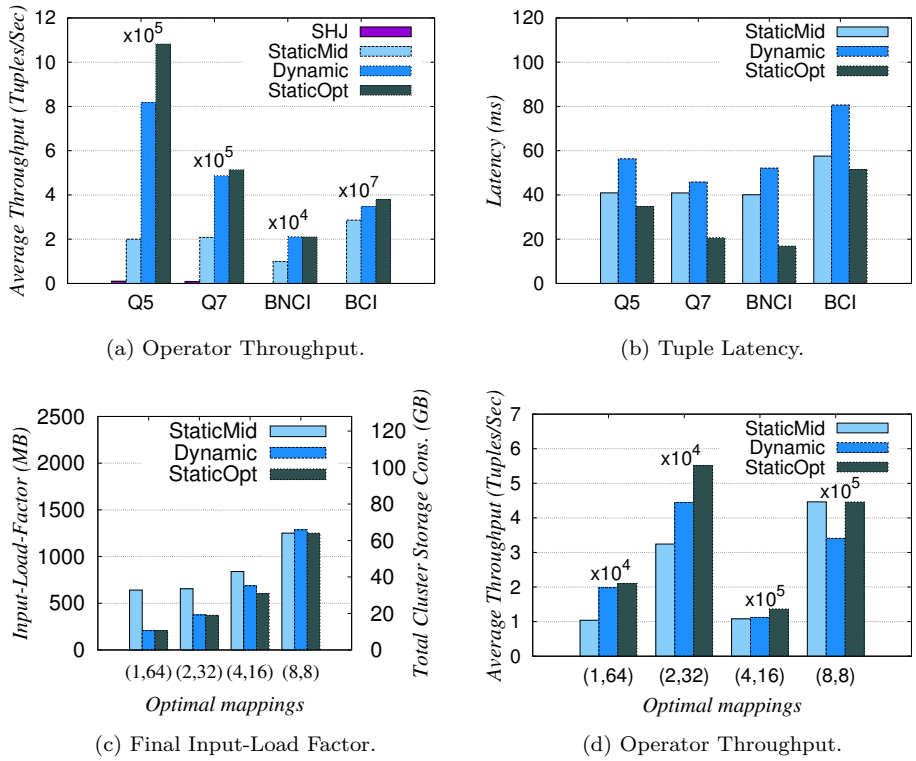
Figure 6

performance of our DYNAMIC operator against the SHJ operator using 16 machines. We report the final execution time. We observe that SHJ performs well under non-skewed settings as it evenly partitions data among machines and does not replicate data. On the other hand, the DYNAMIC operator, distributes workload fairly between machines, but pays for the unnecessary overhead of replicating data. As data gets skewed, SHJ begins to suffer from poor partitioning and unbalanced distribution of data among joiners. Thus, the progress of join execution is dominated by a few overwhelmed workers, while the remaining starve for more data. The busy workers are congested with input data and must overflow to disk, hindering the performance severely. In contrast, the DYNAMIC operator is resilient to data skew and persists to partition data equally among joiners.

5.2 Performance Evaluation

We analyze in detail the performance of static dataflow operators against their adaptive counterpart. We report the results for E_{Q_5} and E_{Q_7} on a Z_4 10G dataset and of B_{NCI} and B_{CI} on a uniform (Z_0) 10G dataset. We start by comparing performance using 16 machines. As illustrated in Table 2, DYNAMIC operates efficiently, whereas STATICMID consistently performs worse. For skewed data, the latter suffers from very high values of ILF, and thus, overflows to disk, hindering the performance drastically. For a fair comparison, we increase the number of machines to 64 such that STATICMID is given enough resources. Under this setting, STATICMID has a fixed $(8,8)$-mapping scheme, whereas the optimal mapping scheme for all joins is $(1,64)$. Our results show that DYNAMIC behaves roughly the same as STATICOPT. This is attributed to the fact that DYNAMIC migrates to the optimal mapping scheme at early stages. For completeness, we also include the results for E_{Q_5} and E_{Q_7} using SHJ. The operator overflows to disk due to high data skew.

Input-Load Factor. As described in §3.3, different mappings incur different values for the input-load factor. Examining the average input-load factor for each operator shows that the growth rate of the ILF is linear over time. Due to the lack of space, we illustrate this behavior for E_{Q_5} only. Fig. 6a plots the maximum size of ILF per machine against the percentage of total input stream processed. SHJ and STATICMID suffer from a larger growth rate than DYNAMIC. Specifically, their rates are 27, 14 and 2MB per 1%



(a) Operator Throughput.

(b) Tuple Latency.

(c) Final Input-Load Factor.

(d) Operator Throughput.

Figure 7

input stream processed, respectively. The graphs depicted in Fig. 6b report on the final average ILF per machine for all the join queries. StaticMid is consistently accompanied with larger ILF values. Its ILF is about 3 to 7 times that of Dynamic. The optimal mapping $(1, 64)$ lies at one end of the mapping spectrum and is far from that of StaticMid. And Shj is up to 13 times that of the other operators.

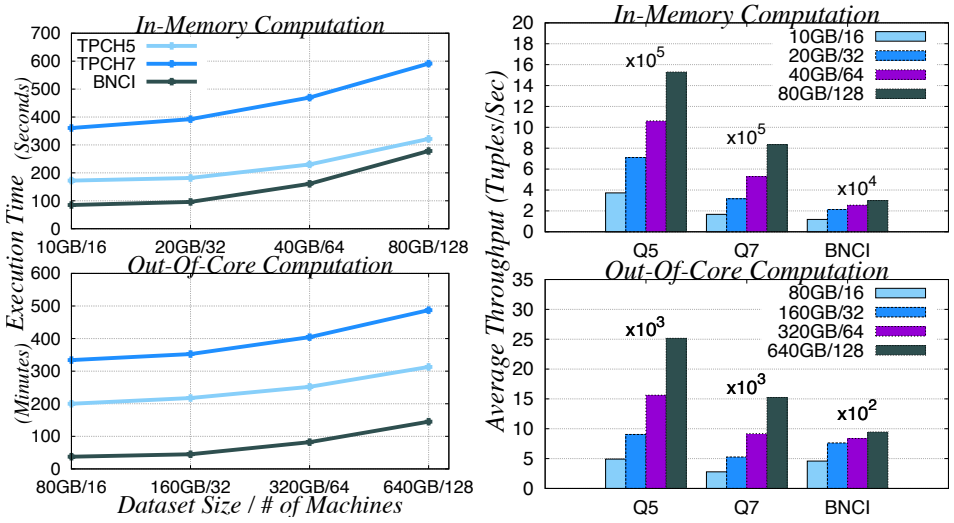
§3.3 also emphasizes the fact that minimizing the ILF maximizes resource utilization and performance. This is due to the fact that higher ILF values also imply (i) unnecessary replicated data stored around the cluster, (ii) more duplicate messages sent congesting the network, and (iii) additional overhead for processing and housekeeping replicated data at each joiner. In what follows, we measure the impact of ILF on overall operator performance.

Resource Utilization. Fig. 6b also shows the total cluster storage consumption (GB), as shown on the right axis. StaticMid's fixed partitioning scheme misuses allocated resources as it unnecessarily consumes more storage and network bandwidth to spread the data. Moreover, it requires four times more machines (64) than Dynamic to operate fully in memory (16 machines used in Table 2). Shj could not fully operate in memory even with 64 machines. Dynamic performs efficiently in terms of resource utilization. This is essential for cloud infrastructures which typically follow *pay-as-you-go* policies.

Execution Time. Fig. 6c shows the execution time to process the input stream for query E_{Q_5}. The other join queries are similar in behavior and we omit them due to the lack of space. Fig. 6d shows the total execution time for all the join queries. We observe that execution time is linear in the percentage of input stream processed. The ILF has a decisive effect on processing time. The rigid assignment $(8, 8)$ of StaticMid yields high ILF values and leads to consistently worse performance. As ILF grows, the amount of data to process, and hence, processing time increases. However, this performance gap is not large when the join operation is computationally intensive, i.e., B_{CI} in Fig. 6d. The execution time for SHJ, shown at the right axis of Fig. 6c, is two orders of magnitude more, illustrating that poor resource utilization may push the operator to disk spills, hindering the performance severely. In all cases, the adaptivity of Dynamic allows it to perform very close to StaticOpt.

Average Throughput and Latency. Fig. 7a shows global operator throughput. For all queries, the throughputs of Dynamic and StaticOpt are close. They are at least twice that of StaticMid, and two orders of magnitude more than that of Shj, except for B_{CI} where the difference is slight. This validates the fact that the ILF has a direct effect on throughput, and that the effect is magnified when overflow occurs. The throughput gap between operators depends on the amount of join computation a machine has to perform (e.g. compare B_{CI} and B_{NCI}). Fig 7b shows average tuple latencies. We define latency as the difference between the time an output tuple t is emitted and the time at which the (more recent) corresponding source input tuple arrives to the operator. The figure shows that the operator latency is not greatly affected by its adaptivity. During state migration, an additional network hop increases the tuple latency. Dynamic achieves average latency close to that of StaticMid while attaining much better throughput.

Different Optimal Mappings. So far, the join queries we experiment on capture the interesting case of an optimal mapping that is far from the $(\sqrt{J}, \sqrt{J})$ scheme. As illustrated in Figs. 7c, 7d, we compare performance under various optimal mapping settings. We achieve this by increasing



(a) Total Execution Time.

(b) Operator Throughput.

the size of the smaller input stream. In all cases, DYNAMIC adjusts itself to the optimal mapping at early stages. Fig. 7c shows how the input-load factor gap between DYNAMIC and STATICMID decreases as the optimal mapping gets closer to the $(\sqrt{J}, \sqrt{J})$-mapping scheme. Similarly, Fig. 7d illustrates how the performance gap decreases between the two operators. This validates the fact that the input-load factor has a decisive effect on performance. In case of the optimal $(\sqrt{J}, \sqrt{J})$-mapping scheme, STATICOPT has the same mapping as STATICMID, whereas DYNAMIC does not deviate from its initial mapping scheme. However, it performs slightly worse because adaptivity comes with a small cost.

5.3 Scalability Results

We evaluate the scalability of DYNAMIC. Specifically, we measure operator execution time and throughput as both the data-size and parallelism configurations grow. We evaluate weak scalability on 10GB/16 joiners, 20GB/32 joiners, and so forth as illustrated in the in-memory computation graphs of Figs. 8a, 8b. Ideally, while increasing the data-size/joiners configuration, the input-load factor and the output size should remain constant per joiner. However, the input-load factor grows, preventing the operator to achieve perfect scalability (same execution time and double average throughput as the data-size/joiners double). For example, for B_{NCI}, under the 20GB/32 configuration, the input stream sizes are 0.68M (million) and 30M tuples, respectively, yielding a $(1, 32)$ optimal mapping scheme with an ILF of $0.68M + 30M/32 = 1.61M \cdot size_{tuple}$ per joiner. However, under the 40GB/64 configuration, the input stream sizes are 1.36M and 60M, respectively, yielding a $(1, 64)$ optimal mapping scheme with an ILF of $1.36M + 60M/64 = 2.29M \cdot size_{tuple}$. In both cases, the output size per joiner is the same (64K tuples). However, the ILF differs by 42% because of the replication of the smaller relation. The ILF for the other two joins does not grow more than 9%. Accordingly, the execution time (Fig. 8a) and the average throughput (Fig. 8b) graphs show that E_{Q5} and E_{Q7} achieve almost perfect scalability. In case of B_{NCI}, a joiner processes more input tuples as data grows. Overall, the operator achieves very good scalability taking into account the increase in ILF.

Secondary storage. Out-of-core computation in Figs. 8a, 8b illustrates performance under weak scalability with secondary storage support. As before, all the queries achieve ideal scalability, taking into account the increase in ILF. This validates the fact that our system can scale with large volumes of data, and that it works well regardless of the local join algorithm. However, compared to the *in-memory* results (Fig. 8a), the performance drops by an order of magnitude. This validates our conclusion that secondary storage is not perfectly suited for high-performance online processing.

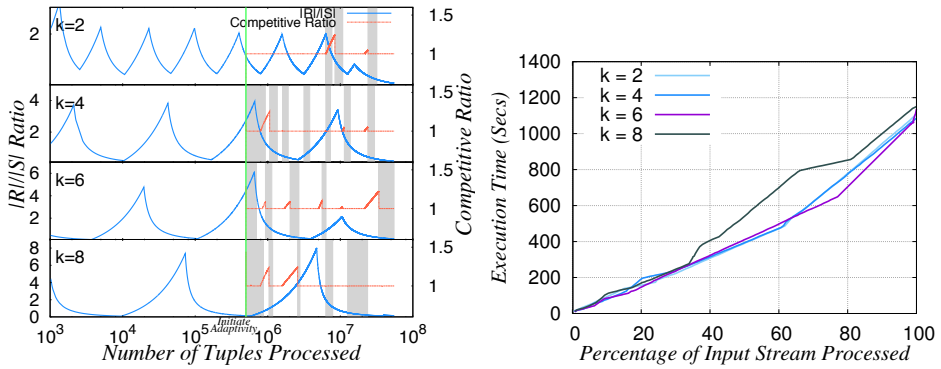
5.4 Data Dynamics

In order to validate the proven theoretical guarantees, we evaluate the performance of DYNAMIC under severe fluctuations in data arrival rates. We simulate a scenario where the cardinality aspect ratios keep on alternating between k and $1/k$ where k is the fluctuation rate. Data from the first relation is streamed into the operator until its cardinality is k times that of the second one. Then, the roles are swapped, by quiescing the first input stream and allowing data to stream in from the second until its cardinality is k times that of the first. This fluctuation continues until the streams are finished. We experiment on an 8G dataset using the *Fluct-Join* query defined below on 64 machines. We run the query under various fluctuation factor, specifically, $k = 2$, $k = 4$, $k = 6$ and $k = 8$. We set the operator to begin adapting after it has received at least 500K tuples, corresponding to less than 1% of the total input.

Fluct-Join

```
SELECT *
FROM  ORDER O, LINEITEM L
WHERE O.orderkey=L.orderkey
AND O.shippriority !='5-LOW' AND O.shippriority !='1-URGENT'
```

Analysis. The first metric of interest is the ILF competitive ratio of DYNAMIC in comparison to an *oracle* that assigns the optimal mapping, and thus optimal ILF*, instantly at all times. Fig. 8c plots both the $|R|/|S|$, on the left axis, and the ILF/ILF* ratio, on the right axis, throughout query execution. In the graph, migration durations are depicted by the shaded regions. We observe that the ratio never exceeds 1.25 at all times which validates the result of Theorem 4.6. Even under severe fluctuations, the operator is well advised in choosing the right moments to adapt. Fig. 8d shows the execution time progress under different fluctuation factors. Although DYNAMIC undergoes many migrations, it persists to progress linearly showing that all



(c) ILF/ILF* Ratio.

(d) Execution Time.

Figure 8: (a)-(b) Weak scalability. (c)-(d) Performance under fluctuations.

migration costs are amortized. This verifies the results of Lemma 4.5 and Theorem 4.1.

5.5 Summary

Experiments show that our adaptive operator outperforms *practical* static schemes in every performance measure without sacrificing low latency. They emphasize the effect of ILF on resource utilization and performance. This validates the optimization goal of minimizing ILF as a direct performance measure. Our operator ensures efficient resource utilization in storage consumption and network bandwidth that is up to 7 times less than non-adaptive theta-join counterparts. Non-adaptivity causes misuse of allocated resources leading to overflows. Even when provided enough resources, the adaptive operator completes the join up to 4 times faster with an average throughput of up to 4 times more. Adaptivity is achieved at the cost of slight increase in tuple latency (by as little as 5ms and at most 20ms). Experiments also show that our operator is scalable. Under severe data fluctuations, the operator adapts to data dynamics with the ILF remaining within the proven bounds from the optimum and with amortized linear migration costs. Additionally, the operator, being *content-insensitive*, is resilient to data skew while *content-sensitive* operators suffer from overflows, hindering performance by up to two orders of magnitude.

6. CONCLUSION AND FUTURE WORK

This paper provides a novel adaptive solution to computing joins with general predicates in an online setting. Unlike previous offline approaches, the adaptive operator presented does not require any prior knowledge about the input data. This is essential when statistics about input data are not known in advance or are difficult to estimate. The operator is highly scalable and continuously processes input streams even during adaptation. Theoretical analysis proves that our algorithm maintains a close-to-optimal state, under an experimentally validated performance measure that captures resource utilization. Furthermore, cost of adaptation is provably minimum. Experiments validate the theoretical guarantees and show that the operator outperforms static approaches; is highly adaptive; and is resilient to data skew. It is also very efficient in resource consumption and maintains high throughput and low tuple latency. Evaluation suggests that there is room for optimization for a special class of joins like equi and band joins. In such low-selectivity joins, the join matrix contains large regions where the join condition never holds. These regions need not be assigned joiners. This motivates designing a *content-sensitive* theta-join operator. Such an operator shares many common features with our operator, but its design poses additional challenges. We leave this for future work.

7. REFERENCES

[1] The Apache Hadoop project. `http://hadoop.apache.org`.

[2] The TPC-H benchmark. `http://www.tpc.org/tpch/`.

[3] D. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the Borealis stream processing engine. In *CIDR*, 2005.

[4] F. Afrati and J. Ullman. Optimizing joins in a MapReduce environment. In *EDBT*, 2010.

[5] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. STREAM: The Stanford data stream management system. Technical report, Stanford InfoLab, 2004.

[6] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. Maskey, E. Ryvkina, M. Stonebraker, R. Tibbetts. Linear road: a stream data management benchmark. In *VLDB*, 2004.

[7] R. Avnur and J. Hellerstein. Eddies: continuously adaptive query processing. In *SIGMOD*, 2000.

[8] S. Babu and P. Bizarro. Adaptive query processing in the looking glass. In *CIDR*, 2005.

[9] S. Blanas, J. Patel, V. Ercegovac, J. Rao, E. Shekita, and Y. Tian. A comparison of join algorithms for log processing in MapReduce. In *SIGMOD*, 2010.

[10] R. Fernandez, M. Migliavacca, E. Kalyvianaki and P. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *SIGMOD*, 2013.

[11] S. Chaudhuri and V. Narasayya. TPC-D data generation with skew.

[12] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI*, 2004.

[13] A. Deshpande and J. Hellerstein Lifting the burden of history from adaptive query processing. In *VLDB*, 2004.

[14] A. Deshpande, Z. Ives, and V. Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1(1), 2007.

[15] J. Dittrich, B. Seeger, D. Taylor, and P. Widmayer. Progressive merge join: a generic and non-blocking sort-based join algorithm. In *VLDB*, 2002.

[16] J. Dittrich, B. Seeger, D. Taylor, and P. Widmayer. On producing join results early. In *PODS*, 2003.

[17] A. Gounaris, N. Paton, A. Fernandes, and R. Sakellariou. Adaptive query processing: A survey. In *British National Conference on Databases*, 2002.

[18] A. Gounaris, E. Tsamoura, and Y. Manolopoulos. Adaptive query processing in distributed settings. *Advanced Query Processing*, 36(1), 2012.

[19] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2), 1993.

[20] X. Gu, P. Yu, and H. Wang. Adaptive load diffusion for multiway windowed stream joins. In *ICDE*, 2007.

[21] P. Haas and J. Hellerstein. Ripple joins for online aggregation. In *SIGMOD*, 1999.

[22] J. Hellerstein, M. Franklin, S. Chandrasekaran, A. Deshpande, K. Hildrum, S. Madden, V. Raman, and M. Shah. Adaptive query processing: Technology in evolution. *IEEE Data Engineering Bulletin*, 23(2), 2000.

[23] J. Hellerstein, P. Haas, and H. Wang. Online aggregation. In *SIGMOD*, 1997.

[24] Y. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. In *SIGMOD*, 1991.

[25] Z. Ives, D. Florescu, M. Friedman, A. Levy, and D. Weld. An adaptive query execution system for data integration. In *SIGMOD*, 1999.

[26] B. Liu, M. Jbantova, and E. Rundensteiner. Optimizing state-intensive non-blocking queries using run-time adaptation. In *ICDE Workshop*, 2007.

[27] M. Mokbel, M. Lu, and W. Aref. Hash-Merge join: A non-blocking join algorithm for producing fast and early join results. In *ICDE*, 2004.

[28] A. Okcan and M. Riedewald. Processing theta-joins using MapReduce. In *SIGMOD*, 2011.

[29] M. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *Annual Technical Conference*. USENIX, 1999.

[30] C. Olston, B. Reed, A. Silberstein, and U. Srivastava. Automatic optimization of parallel dataflow programs. In *Annual Technical Conference*. USENIX, 2008.

[31] N. Paton, J. Buenabad, M. Chen, V. Raman, G. Swart, I. Narang, D. Yellin, and A. Fernandes. Autonomic query parallelization using non-dedicated computers: an evaluation of adaptivity options. *VLDBJ*, 18(1), 2009.

[32] A. Pavlo, E. Paulson, A. Rasin, D. Abadi, D. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD*, 2009.

[33] D. Schneider and D. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *SIGMOD*, 1989.

[34] M. Shah, J. Hellerstein, S. Chandrasekaran, and M. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *ICDE*, 2002.

[35] J. Stamos and H. Young. A symmetric fragment and replicate algorithm for distributed joinsyout. *Transactions on Parallel and Distributed Systems*, 4(12), 1993.

[36] M. Stillger, G. Lohman, V. Markl, and M. Kandil. LEO - DB2's learning optimizer. In *VLDB*, 2001.

[37] Y. Tao, M. L. Yiu, D. Papadias, M. Hadjieleftheriou, and N. Mamoulis. RPJ: producing fast join results on streams through rate-based optimization. In *SIGMOD*, 2005.

[38] F. Tian and D. DeWitt. Tuple routing strategies for distributed eddies. In *VLDB*, 2003.

[39] P. Upadhyaya, Y. Kwon, and M. Balazinska. A latency and fault-tolerance optimizer for online parallel query plans. In *SIGMOD*, 2011.

[40] T. Urhan and M. Franklin. XJoin: A reactively-scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, 23(2), 2000.

[41] S. Wang and E. Rundensteiner. Scalable stream join processing with expensive predicates: workload distribution and adaptation by time-slicing. In *EDBT*, 2009.

[42] A. Wilschut and P. Apers. Dataflow query execution in a parallel main-memory environment. In *Parallel and Distributed Information Systems*, 1991.

[43] Y. Xing, S. Zdonik, and J. Hwang. Dynamic load distribution in the Borealis stream processor. In *ICDE*, 2005.

[44] H. Yang, A. Dasdan, R. Hsiao, and D. Parker. Map-Reduce-Merge: simplified relational data processing on large clusters. In *SIGMOD*, 2007.

[45] X. Zhang, L. Chen, and M. Wang. Efficient multi-way theta-join processing using MapReduce. *VLDBJ*, 5(11), 2012.

[46] Y. Zhou, B. Ooi, and K. Tan. Dynamic load management for distributed continuous query systems. In *ICDE*, 2005.