Squall: Scalable Real-time Analytics

Aleksandar Vitorovic, Mohammed Elseidy, Khayyam Guliyev, Khue Vu Minh, Daniel Espino, Mohammad Dashti, Yannis Klonatos and Christoph Koch

\{firstname\}.\{lastname\}@epfl.ch
École Polytechnique Fédérale de Lausanne

ABSTRACT

Squall is a scalable online query engine that runs complex analytics in a cluster using skew-resilient, adaptive operators. Squall builds on state-of-the-art partitioning schemes and local algorithms, including some of our own. This paper presents the overview of Squall, including some novel join operators. The paper also presents lessons learned over the five years of working on this system, and outlines the plan for the proposed system demonstration.

1. INTRODUCTION

Online processing implies that results are incrementally built as the input arrives. Thus, each input tuple produces output and updates the system state necessary for processing subsequent inputs. Online processing is ubiquitous for many applications such as algorithmic trading, clickstream analysis and business intelligence (e.g., in order to reach a potential customer during the active session).

Existing open-source online systems (e.g., Twitter’s Storm [20], Spark Streaming [33], Flink [1]) focus on distribution primitives (e.g., communication patterns, fault tolerance) and low-level performance optimizations. However, these systems provide only vanilla database operators, such as hash-based equi-joins (and general UDFs), which do not perform well in the case of skew (see §3.1). On the other hand, some join partitioning schemes (e.g., [22]) are skew-resilient, but they are designed for offline processing, and thus, they are unable to adapt to changing data statistics (see §5).

In contrast, Squall is a system that puts together state-of-the-art partitioning schemes, local query operators, and techniques for scalable online query processing. We also build novel 2-way [13, 30] and multi-way schemes (Hybrid-Hypercube, see §3.1). Such a system allows us to leverage the effect of various design choices on the performance, and to seamlessly build efficient novel operators (see §3). Squall operators achieve skew-resilience, adaptivity and scalability.

Squall is an open-source project\(^1\) that has been developed for the last five years (mainly by the authors at EPFL, but also with external contributions). It has been available for several years, and it has attracted a community of users.

2. SYSTEM ARCHITECTURE

\(^1\)Flink provides both offline and online processing, but in this paper we discuss only the online case.

Squall is an online distributed query engine which achieves low latency and high throughput. It supports incremental view maintenance and window (stream) semantics. Squall implements typical stream primitives, such as tumbling and sliding windows, by adding the window expiration logic on top of the full-history engine. Squall uses Storm [20] as a distribution and parallelization platform.

The overall system architecture is shown in Figure 1.

**User interface.** Squall offers multiple interfaces: declarative (SQL), functional (a modern Scala collections API), interactive (Scala) and imperative (Java). Similarly to Hive which provides an SQL interface on top of Hadoop, Squall’s declarative interface offers running SQL over Storm. Squall’s functional interface provides for compositions of data transformations over streams. Squall also provides interactive interface built on top of the Scala REPL that allows a user to interactively construct query plans. For each of these three interfaces, Squall translates the user input to a logical query plan (see Figure 1). Finally, the imperative interface gives the user full control over the physical query plan.

**Logical and Physical query plans.** A logical Squall query plan is a DAG of relational algebra operators. A physical Squall query plan consists of a DAG of physical operators and their requested level of parallelism. An operator is specified by the partitioning scheme and local algorithm. To minimize the number of network hops, and thus maximize the performance, we collocate the connected operators that use the same partitioning scheme. We denote a pipeline of collocated operators as a *component*. Figure 1 shows components as rounded rectangles in the example physical plan. An example of a component is a join followed by a selection.

**Operators.** By combining different partitioning schemes and local join algorithms, Squall offers many join operators. We build novel join operators: adaptive 1-Bucket [13] and Equi-weight-histogram (EWH) join [30]. This paper also presents some novel multi-way joins (a multi-way join runs within a component). Beside joins, Squall offers database operators such as selections, projections and aggregations.

**Query optimizer.** Squall’s optimizer generates a physical plan from the logical plan. The optimizer maximizes throughput and minimizes latency and the number of machines used. It starts from the data sources and adds the operators one after another, pushing selections and projections as close as possible to the data sources. Where possible, the optimizer collocates operators to components to minimize network transfers. Further, it assigns the right parallelism to each component, such that a component is neither overloaded nor mostly idle. We refer to this as universal parallelism to each component.
3. NOVEL JOIN OPERATORS

We devise new join operators by wiring up state-of-the-art partitioning schemes and local join algorithms. So far, we built 2-way joins [13, 30]. This paper presents multi-way joins. These can outperform 2-way joins as they avoid shuffling intermediate data, which can be very large [4, 34]. We devise a novel multi-way join partitioning scheme that further enhances performance. In addition, Squall has efficient local online multi-way joins.

3.1 Partitioning schemes

Next, we describe partitioning schemes for multi-way joins, and their skew resilience and supported join conditions.

Hash-Hypercube scheme [4] models the result space as a hypercube, where each axis corresponds to a join key domain. Each machine covers a unique portion of the hypercube space. Figure 2a illustrates this scheme for query $R(x, y) \bowtie S(y, z) \bowtie T(z, t)$. The scheme assigns input tuples to machines by hashing on their join keys and replicating on join keys from the other relations. For example, each $R$ tuple is replicated to a "row" of machines with coordinates $(\text{hash}(y), \ast)$. Correctness is preserved as each potential output tuple $t_R(x, y) \bowtie t_S(y, z) \bowtie t_T(z, t)$ is assigned to a single machine with coordinates $(\text{hash}(y), \text{hash}(z))$. For uniform distribution, the load of each machine $L$ is $|R|/8 + |S|/8 + |T|/8 \approx 0.26H$ (each relation is of size $H$, so given 64 machines, the dimensions $8 \times 8$ minimize the load). This scheme supports skew-free multi-way equi-joins.

Random-Hypercube scheme [34]. This scheme also models the result space as a hypercube, but each axis corresponds to a relation, as shown in Figure 2b. This scheme randomly distributes the tuples on the axes of the originating relation, and replicates on the other axes. For example, each $R$ tuple is replicated on a "slice" of machines (Figure 2b shows a slice with shading). The load per machine is $3 \cdot H/4 = 0.75H$ (the dimensions are $4 \times 4 \times 4$, and a machine receives 1/4 of each relation). The Random-Hypercube supports multi-way theta-joins and is skew resilient. However, it replicates tuples more than the Hash-Hypercube (it uses a 3-dimensional rather than 2-dimensional hypercube).

2-way join schemes. For 2-way joins, Hash-Hypercube becomes hash partitioning, and Random-Hypercube becomes 1-Bucket scheme [22], which uses random partitioning over a 2-dimensional hypercube (matrix). Random partitioning is skew resilient but replicates tuples over the matrix. For low-selectivity band and inequality 2-way joins, range partitioning allows fast detection of large continuous matrix portions that produce no output. As these portions are not assigned to machines, range partitioning schemes outperform the 1-Bucket scheme. Examples include the M-Bucket scheme [22] and our Equi-Weight Histogram (EWH) scheme [30]. The M-Bucket scheme is prone to output skew. In contrast, the EWH scheme works well for any data distribution. To do so, our EWH scheme provides an efficient parallel scheme for capturing the input and output distribution from the join to a matrix. To evenly partition the work (matrix) among the machines, the EWH scheme employs our join-specialized computational geometry algorithm for rectangle tiling.

Our Hybrid-Hypercube scheme. Consider the same query $(R(x, y) \bowtie S(y, z) \bowtie T(z, t))$ on a non-uniform dataset. For example, assume that $y$ has uniform distribution and that $z$ has zipfian distribution (the skew parameter of 2) both in $S$ and $T$. The Random-Hypercube scheme performs the same independently of skew ($L = 0.75H$, as before). The Hash-Hypercube scheme with the given data distribution is shown in Figure 2c. Due to skew, it performs only slightly better than the Random-Hypercube (the maximum
load per machine is $L = |R|/8 + |S|/(8 \cdot 2) + |T|/2 \approx 0.69H$.

Hash- and Random-Hypercube are designed only for the cases when either all or none of the relations is skew-free. We propose Hybrid-Hypercube, which uses hash partitioning for skew-free join keys, and more costly random partitioning elsewhere. That way, our scheme achieves skew resilience while minimizing tuple replication. Further, in contrast to the Hash-Hypercube, the Hybrid-Hypercube supports non-equi joins and can tolerate overlapping ranges of tuples from these relations on the same set of machines.

"row" of machines. We can consider Random-Hypercube as before. Thus, our scheme subsumes the same performance improvement compared to the hash join. We preserve correctness as we partition the same tuples into different partitions. Thus, our scheme achieves skew resilience while minimizing tuple replication. Further, in contrast to the Hash-Hypercube, the Hybrid-Hypercube supports non-equi joins (using random partitioning therein).

Assume that $R$ and $S$ have an inequality join condition between $R$ and $S$, and $T$ is a (replicated) hash join. We preserve correctness as we partition $R$ and $S$ using the same hash function, so the corresponding partitions from these relations are on the same set of machines. Whereas, each $T$ tuple randomly picks a "column" of machines to be replicated on. Given that there are no skew on $y$ and no functional dependencies between $y$ and $z$ (which is a common case), hash($y$) from $R$ and $S$ simulates random distribution with respect to $T$. Thus, we can consider $(R \bowtie S) \bowtie T$ as a 1-Bucket join. We preserve correctness as follows. $R$ and $S$ tuples "meet" all the tuples from $T$, as each $T$ tuple intersects each row on a single machine.

As a result, the maximum machine load in the Hybrid-Hypercube is $L = (|R| + |S|)/7 + |T|/9 \approx 0.36H$, which is $2.08\times$ and $1.92\times$ better than that of Random-Hypercube and Hash-Hypercube, respectively.

3.2 Important special cases

Star schema typically consists of one big fact table and several small dimension tables. Usually, in a distributed setting, the fact table is partitioned and dimension tables are replicated. Interestingly, both the Hash-Hypercube and Random-Hypercube schemes comply with this partitioning. Namely, due to relative relation sizes, these schemes yield $p \times 1 \cdots 1$ partitioning ($p$ is the number of machines), which implies partitioning on one dimension and replication on other dimensions. The only difference is that the Hash-Hypercube scheme partitions the fact table on join keys, while the Random-Hypercube scheme randomly partitions the fact table.

3.3 Local join algorithms

Online local joins typically work as follows: a new incoming tuple for a relation is joined with the stored tuples from the other relation(s), and stored for use by future tuples [13]. Existing online distributed systems enhance their local joins with indexes (hash or balanced binary tree) to improve performance. However, these local joins are orders of magnitude slower than the state-of-the-art online local join, DBToaster [5]. The gap deepens with the increase in the number of relations in a multi-way join.

In brief, the main idea of DBToaster is to recursively maintain views for an $n$-way join. Instead of maintaining only the final result, DBToaster maintains all the intermediate $(n-1)$-, $(n-2)$-, ..., and 2-way joins. When a new tuple comes, DBToaster updates the intermediate relations, and produces the result by joining the tuple with the appropriate $(n-1)$-way materialized join. The savings come from the fact that DBToaster does not recompute the $(n-1)$-way join for each new tuple, as it would be the case if we use indexes only on the base relations.

Yet, there was no parallel version of DBToaster until now.

3.4 HyLD operator: Hypercube scheme with Local DBToaster

Squall seamlessly parallelizes the state-of-the-art local join (DBToaster) by using separation of concerns. In particular, the hypercube schemes ensure that each machine executes an independent portion of the join, so that each output tuple is produced at exactly one machine. That way, we can run a separate instance of DBToaster on each machine. We denote such an operator as Hypercube scheme with Local DBToaster (HyLD). The HyLD operator combines network efficiency due to a hypercube scheme with CPU efficiency due to using DBToaster locally.

Choosing among hypercube schemes. As shown in §3.1, random partitioning is expensive but skew-resilient, while hash partitioning is cheaper but prone to skew. To decide on the hypercube scheme, we need to know if a join key is skew-free or not. A good initial choice of a hypercube scheme saves us from future adaptations. Fortunately, in many cases, even in an online scenario, we know before-
4. MULTI-WAY JOINS: GENERAL CASE

So far, we illustrated the hypercube schemes on a 3-way join (see §3.1). Next, we discuss how to find an optimal partitioning for a general join for each scheme. For each scheme, the optimal partitioning minimizes the load per machine, and thus, it also minimizes the total amount of replication. We are given \( p \) machines, where \( p = p_1 \cdot p_2 \cdots p_l \). The machines are organized in a hypercube, where each dimension \( j \) is of size \( p_j \).

**Hash-Hypercube.** Given relations \( R_i \), where \( i \in 1..k \), and hypercube dimension sizes \( p_1 \times p_2 \times \cdots \times p_l \), the formula for load per machine is \( \sum i |R_i|/p_i \) [4]. In other words, the load from each relation is partitioned among dimensions that correspond to the join keys from that relation. In general, not each join key has a separate axis (equivalently, each join key corresponds to an axis, but some axes are of size 1, so we omit them from the dimensions).

**Random-Hypercube.** The problem formulation is similar as before, except that the number of hypercube dimensions is the same as the number of relations. More precisely, we want to minimize load per machine, which is equal to \( \sum i |R_i|/p_i \) [34]. As shown in [34], the optimal hypercube is the one that divides its dimensions into segments of equal size, that is, \( |R_i|/p_1 \approx |R_2|/p_2 \approx \cdots \approx |R_k|/p_k \). For example, if we have 64 machines and \( R_1 \) is 4x bigger than \( R_2 \), the optimal partitioning \( (R_1, R_2) \) is 16x4. This partitioning implies the minimal load per machine and minimal communication cost.

**Hybrid-Hypercube.** In the 3-way join example from §3.1, the Hybrid-Hypercube saved one dimension compared to the Random-Hypercube, while still providing for skew resilience. In general, we can save more than one hypercube dimension. For example, if in \( R(x,y) \bowtie S(y,z) \bowtie T(z,t) \bowtie U(t) \) only \( z \) has skew, the Random-Hypercube uses 4 dimensions, while the Hybrid-Hypercube uses only 2 dimensions. To do so, we hash \( R \) and \( S \) on attribute \( y \) to rows, and \( T \) and \( U \) on \( t \) to columns. In general, we can apply dimensionality reduction in multiple places in the query.

To decide on dimensions and their sizes for a general multi-way join, we extend the optimization algorithm for the Hash-Hypercube. First, we rename join keys where skew occurs or where non-equi join is used. An example for the former is \( R(x,y) \bowtie S(y,z) \bowtie T(z,t) \), which becomes \( R(x,y) \bowtie S(y',z) \bowtie T(z,t) \) if \( y' \) is skewed in \( S \). An example for the latter is \( R(x,y) \bowtie S(y,z) \bowtie T(z,t) \) with \( S.z < T.z \), which becomes \( R(x,y) \bowtie S(y,z) \bowtie T(z',t) \). Further, if we have one join key appearing in multiple places, such as \( R(x) \bowtie S(x) \bowtie T(x) \) and only \( T.x \) is skewed, we can use \( x \) variable for \( R \) and \( S \) and only rename \( T.x \) to \( x' \).

Let us consider the rewritten non-equi join \( R(x,y) \bowtie S(y,z) \bowtie T(z',t) \). From the viewpoint of the Hash-Hypercube optimization algorithm, this can be considered as an equi-join with dimensions \( (y, z, z') \). Note that the fact that \( T \) on \( z' \) uses random rather than hash partitioning does not change anything in the formulas for the dimension sizes. This is because we care only about equal distribution of tuples among the rows/columns. It is irrelevant for the formulas whether we achieve this using randomization or a hash function on a uniform dataset. The optimization algorithm typically returns a partitioning with only \( (y, z') \) dimensions, which corresponds to our Hybrid-Hypercube. However, this is a valid partitioning only if there is no functional dependencies between \( y \) and \( z \) in relation \( S \). If this does not hold, we use the best partitioning with \( z \) being one of the dimensions. In this particular case (we have \( (y, z, z') \) dimensions), we do not achieve dimensionality reduction compared to the Random-Hypercube. Thus, we might fall back to Random-Hypercube (that is, use random partitioning for each \( (y, z, z') \) dimension), as it is not sensitive to relation skew at all.

5. SKEW TYPES AND ADAPTIVITY

The data distribution in an online system can change, so Squall offers some adaptivity techniques.

**Skew due to hash imperfections.** One may think that, in the case of uniform data distribution, hashing (both for aggregations and joins) always leads to even load distribution. However, there are two situations when this is not the case. The first one happens if the number of distinct keys is smaller than the operator parallelism. It causes some machines to be completely idle. Second, uneven load distribution becomes very likely when the number of distinct keys \( d \) and the operator parallelism \( p \) are the same, or when \( d \) is a bit bigger than \( p \). For instance, if \( d = 15 \) and \( p = 8 \), the optimal scheme will assign no more than \( 15/8 = 2 \) keys for each machine. However, due to imperfections of hash functions, it is very likely that some machine will have 3 keys, causing it to have severe performance degradations (1.5x higher maximum load per machine than in the optimal case). The situation is even worse when \( d = p \), as it becomes very likely that one machine is assigned 2 keys, while the optimum is 1 key per machine. The node, being assigned two times more work, becomes a bottleneck. This results in a largely suboptimal plan in terms of resource utilization, throughput and latency.

Unfortunately, this happens frequently in practice. For example, many TPC-C queries (e.g. Q4, Q5, Q12) have final aggregations with only up to 25 distinct values. There are also some queries in TPC-H that have small number of distinct keys for joins. For example, Q7 joins two \( NATITION \) tables, which have only 25 distinct values.

On the other hand, we typically know all the distinct values for attributes with low distinctness (e.g. possible values for ship priorities are predefined). Squall uses this information to achieve perfect load balancing. Before the execution starts, Squall creates a mapping from different keys to machines using a round robin partitioning.

**Skew fluctuations.** There is an important difference in adaptivity among hash, range and random partitionings. Hash partitioning uniformly partitions the data, and thus, it always yields bad performance in the presence of skew. For range partitioning, an online operator needs to periodically
adjust to the data distribution changes. Interestingly, an adversary can change the data distribution right after the system adjusts the scheme, causing the scheme to always be highly suboptimal. The random partitioning avoids this problem as it randomly assigns tuples to machines, essentially removing any skew in data distribution.

**Temporal skew.** Having the exact data distribution, including the uniform distribution, might not suffice for skew resilience. For hash partitioning, in the case of sorted tuple arrival and moderate join key frequencies, only one machine will be active at a time, which is equivalent to sequential execution. Sorted tuple arrival is very common in practice, e.g., when joining on monotonically increasing unique identifiers. We denote imbalance in load caused by tuple arrival order as temporal skew. Range partitioning is also prone to temporal skew. We observe similar behavior for random partitioning and sorted tuple arrival. In contrast, random partitioning performs the same independently of tuple arrival order as temporal skew. Range partitioning is also prone to temporal skew. We observe similar behavior for random partitioning and sorted tuple arrival. In contrast, random partitioning performs the same independently of tuple arrival order, as the tuples are randomly distributed among the machines.

Thus, it is insufficient to capture only the data distribution. Rather, we also need to capture the temporal skew, which we do indirectly by monitoring the machine load. To achieve good performance, Squall uses random partitioning schemes in the case of data or temporal skew.

**Join selectivity fluctuations.** Next, we explain how multi-way joins bring an additional adaptivity level compared to the pipeline of 2-way joins. The join selectivity for 2-way joins can vary at run-time, and some intermediate relations may grow very large. A possible response is adaptive join reordering. In that case, we discard some intermediate relations (e.g., \( R \bowtie S \)) and rebuild new state for other intermediate relations (e.g., \( S \bowtie T \)) from scratch. This may have very adverse and hard to predict effects in an online system, including very large latencies for new incoming tuples.

On the other hand, multi-way joins maintain no intermediate relations. Thus, in contrast to a pipeline of 2-way joins, hypercube schemes inherently bring adaptivity to the join selectivity fluctuations.

**Hypercube sizes.** The optimal hypercube dimension sizes minimize replication, and thus, maximize performance. We determine the optimal sizes from the relative relation sizes. Hence, a hypercube scheme needs to adapt to changing relation sizes. Squall implements an adaptive 1-Bucket join operator [13], which periodically adjusts the offline partitioning scheme according to the current relation sizes. This operator minimizes state migration, offers a non-blocking migration algorithm, and provides optimality guarantees on data distribution and communication cost.

**SAR principle.** We introduce the SAR principle, which summarizes this section. To achieve skew-resilience and adaptivity for increasing number of skew types in an online system, partitioning schemes need to increase the input tuple Replication. For 2-way joins, hash partitioning is prone to skew but requires no replication. Whereas, random partitioning is resilient to data and temporal skew, including skew fluctuations, but it requires replication. Multi-way joins bring adaptivity to join selectivity variations, but require higher replication than their counterpart 2-way joins.

**Fault tolerance.** Squall uses the features of Storm to achieve fault tolerance. However, we can sometimes design a better FT strategy by taking into account peculiarities of the employed partitioning schemes. In fact, if the partitioning scheme replicates tuples, a failed node can recover its state from some of its peers rather than from a disk checkpoint. For example, in Figure 2b, if a machine with coordinates \( \{1, 1, 1\} \) fails, we can recover its state from any machine \( \{1, *, *\} \) (for \( R \)), \( \{*, 1, *\} \) (for \( S \)) and \( \{*, *, 1\} \) (for \( T \)). This improves performance, as network accesses are several times faster than disk accesses. When RDMA is used, the performance improvements are even higher.

We can employ the same optimization even if the partitioning scheme only partially replicates the operator state. In that case, we achieve efficient fault tolerance without replicating the entire operator. Rather, we replicate only the parts of the operator state that are not already replicated by the partitioning scheme.

### 6. DEMONSTRATION SETUP

The proposed system demonstration exposes scalability and skew-resilience of Squall in high-data-rate analytics applications.

**Google cluster monitoring data** contains information about jobs (start and end time, status, etc.), tasks (events, resource usage) and machines (assignments, attributes). We put ourselves in the shoes of a large cluster administrator, who gets notified when a potential problem arises. An interesting multi-way join query is *List the machines which often fail tasks belonging to production jobs*. Another interesting query is *Measure the scheduling algorithm quality*. Schedulers assign jobs to machines to maximize "goodness" score [28], which includes the machine's number of preempted or failed tasks, jobs distribution across the cluster etc. The score involves joining multiple relations. We

5This requires that the partitioning scheme reflects the actual data distribution.

6https://gist.github.com/jboner/2841832

7https://github.com/google/cluster-data
observe the scheduling algorithm quality by monitoring (in real-time) the score aggregated over jobs and machines.

**Demo.** As shown in Figures 3 and 4, we allow attendees to specify a query and to try out different partitioning schemes, local joins and the number of machines. To illustrate the effect of temporal skew, we also offer choosing among sorted and non-sorted datasets. With a button click, the attendees will run the specified query plan on an in-house cluster with 220 hardware threads. At run-time, they will be able to continually monitor the query results, performance metrics (throughput, latency, CPU utilization and memory consumption) and operators’ properties such as hypercube dimensions, replication factor and skew. The replication factor is the component’s number of input tuples divided by the total number of tuples produced by the immediate upstream components. We define skew degree as the division between the largest partition size and the average partition size.

**Evaluating partitioning schemes.** We allow attendees to compare hypercube schemes by monitoring the performance as a function of the operator’s replication factor and skew degree. For instance, the Random-Hypercube scheme achieves perfect load-balancing (no partition skew) but it replicates tuple (as we can observe from the replication factor). For each hypercube scheme, we identify scenarios (the number of relations, their sizes and skew degrees) where it performs the best. We also evaluate the effect of temporal skew to the performance of hash join and 1-Bucket join. The results validate the SAR principle and suggest that replicating skew to the performance of hash join and 1-Bucket join. The results validate the SAR principle and suggest that replication is ubiquitous for the worst-case load balancing.

**CPU or network-bound?** We aid attendees to find the bottleneck in online processing. To estimate the CPU share, we run the same query plan with different local joins. The attendees can also observe the correlation among the operator’s memory consumption and throughput. To estimate the network share, we define intermediate network factor as 

\[
\frac{\sum_{\text{comp. task}} \text{input}_t \cdot \text{output}_t}{(\text{query input} + \text{query output})}.
\]

Then, we compare the performance among different query plans (of the same query) as a function of this factor.

7. RELATED WORK

**Offline multi-way join schemes.** The Hash-Hypercube [4] and Random-Hypercube [34] schemes, which we describe in detail in §3.1.4, are originally proposed for offline systems. (We showed that we can use these scheme in online systems as well, by periodically adjusting to the statistics collected so far [13].) Similarly, our Hybrid-Hypercube scheme is also directly applicable for offline processing. Our scheme advances state-of-the-art, as in contrast to the Hash-Hypercube it supports non-equijoin and is skew resilient, while incurring significantly smaller communication cost compared to the Random-Hypercube. The main insight of the Hybrid-Hypercube is to optimize the replication according to the relation skew degree and join conditions. Our Hybrid-Hypercube scheme only needs a skew degree for relations, which we estimate from a sample from each relation.

Chu et al. [10] propose an operator that combines the Hash-Hypercube partitioning scheme with a state-of-the-art offline local operator for cyclic joins. In contrast, we offer different hypercube schemes, and use state-of-the-art online local join operator for acyclic joins. Inspired by [10], in the future we plan to combine local online cyclic joins with our hypercube schemes. YSmart [15] studies partitioning schemes for subqueries consisting of both joins and aggregations. It recognizes subqueries that can be executed without any replication within a single MapReduce job.

BinHC [8] and SharesSkew [3] are partitioning schemes for multi-way joins that treat separately heavy hitters (the join keys with high multiplicity). The main idea is to use some variant of hash partitioning for light hitters and random partitioning for heavy hitters. These operators may achieve smaller load per machine compared to the Hybrid-Hypercube in the offline setting. However, both BinHC and SharesSkew are restricted to equi-joins.

BinHC [8] and SharesSkew [3] might be suboptimal in an online scenario. They require detailed statistics about skew, that is, key frequencies. Although we can adjust the partitioning scheme according the statistics seen so far, the (relative) key frequencies repeatedly change over time. This implies frequent data migrations, which affects the performance. In contrast, the Hybrid-Hypercube requires only information about whether the relation is skewed or not (this information is used to decide on hash or range partitioning). This typically changes less frequently, causing smaller number of migration and better performance compared to the online counterparts of BinHC and SharesSkew.

**Local online join algorithms.** There is a significant body of work on local online 2-way join algorithms [32, 27, 25, 12, 21]. Symmetric hash join [32] requires that data fits in memory. Works [27, 25, 12, 21] address this issue by employing different strategies for spilling to disk. MJoin [29] generalizes XJoin [27] to multi-way joins, and focuses on strategies for spilling to disk. CACQ [19] and STAIRs [11] execute multi-way joins using Eddies architecture [7], that is, they decide on per-tuple basis on an optimal join order. The main difference between DBToaster [5] that we use in Squall and these multi-way joins is as follows. First, these works [29, 19, 11] focus on equi-joins. Second, DBToaster materializes intermediate multi-way joins (2-way to (n-1)-way joins) in order to avoid re-computation. In contrast, STAIRs only partially avoids re-computation, as it materializes intermediate tuples that results from joining of only up to 2 relations. Finally, Squall is an extensible system, as we can combine any of these local join algorithms with our partitioning schemes.

Distributed online joins. BiStream [16] is a 2-way stream join operator that partitions input relations on a disjoint set of machines. It supports both equi- and non-equijoins, and it focuses on scalability and elasticity. However, when using random partitioning (for non-equijoins or for equi-joins with high skew), it has higher communication cost than the 1-Bucket scheme [22].

Distributed online joins: multiple hops. We next describe the line of work that execute multi-way joins using multiple network hops. CTR scheme [14] and PSP scheme [31] optimize tuple routing, providing for adaptive join ordering. These approaches have the following drawbacks. First, the intermediate state can be considerably large, causing high communication overhead, and potentially high latency for producing result tuples. Second, these approaches do not materialize intermediate tuples, and suffer from recomputation. In contrast, our HyLD operator solves both problems. It requires only one network hop to produce the result tuple, and it uses local DBToaster operator that allows reusing the previously computed intermediate results.

Distributed Eddies [26, 35] also provide for adaptive join
ordering. They assume window semantics, tolerates information loss and do not study intra-operator adaptations (such as our Adaptive 1-Bucket scheme [13]). Distributed Eddies do not materialize intermediate results, as for small to moderately-sized windows intermediate results might not be frequently reused (when window expires, its intermediate results also expire). However, reusing intermediate results is especially important for large windows and full-history queries, and we focus on these scenarios.

Distributed online joins: single hop. Next, we present the multi-way join operators that require only one network hop, similarly to our hypercube schemes. ATR scheme [14] uses range partitioning (with some overlapping) on timestamp, so it replicates tuples less than the hypercube schemes. However, ATR executes the entire window on one machine, so it might not scale for large windows and fast incoming rates. Nowadays, online operators with large windows or full-history history semantics are very popular [9, 6]. We can extend Squall with ATR partitioning schemes to support small to moderate-sized window operators.

Flux [24] is an adaptive partitioning scheme, where the number of partitions is much higher than the number of machines. This scheme supports skew but assumes that none of the partitions, which are specified in the initialization, suffers a machine capacity. As explained in [31], this is easily violated in online scenarios. Flux is originally proposed for single-input operators, but it can support some join conditions, such as equi-joins [17]. Liu et al. [17, 18] provide multi-way equi-join operators using Flux (and inheriting its drawbacks). Liu et al. [17, 18] does not consider partitioning schemes with replication, rather they focus on multi-way joins where all the relations use the same join key. This line of work offers moving operator states among the machines, as well as spilling to disk. In addition, it allows changing the join order at run-time, or even run-time changing a pipeline of 2-way joins to a single-hop multi-way join. However, it requires blocking of input streams while migrating state. This causes long stalls for operators with large state, which is unacceptable in online systems. In contrast, our Adaptive 1-Bucket [13] is a non-blocking scheme.

8. REFERENCES