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).

Skew occurs frequently in real-life datasets. For instance, certain types of skewed distributions (such as zipfian distribution) appear in Internet packet traces, city sizes, word frequency in natural languages and advertisement clickstreams [17]. Existing open-source online systems (e.g., Twitter's Storm [49], Spark Streaming [73], Flink [14][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). Regarding non-equi joins, Storm do not provide them. Whereas, Spark Streaming and Flink execute non-equi joins very inefficiently (a Cartesian product followed by a selection). On the other hand, existing partitioning schemes that support both equi-joins and non-equi joins (e.g., [54]) have the following drawbacks. First, they work efficiently only for a narrow set of data distribution properties. Second, these schemes are designed for offline processing, and thus, they are unable to adapt to changing data statistics (see §5). Squall addresses there problems.

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 [32, 66] 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[2] 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

Squall is an online distributed query engine which achieves low latency and high throughput. It supports full-history (incremental view maintenance) and window (stream) semantics. Squall uses Storm [49] as a distribution and parallelization platform.

The overall system architecture is shown in Figure 1. Next, we give an overview of various Squall concepts.

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 (Read-Eval-Print Loop) that allows a user to interactively and run 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. A user can run a query plan specified by any Squall interfaces either locally or on a cluster, making it easy to learn and test Squall.

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. A physical operator is specified by the partitioning scheme and local algorithm. To minimize the number of network hops, and thus to maximize the performance, we co-locate the connected operators that employ the same partitioning scheme. We denote a

[1]Flink provides both offline and online processing, but in this paper we discuss only the online case.

[2]https://github.com/epfldata/squall/

Proceedings of the VLDB Endowment, Vol. 9, No. 10

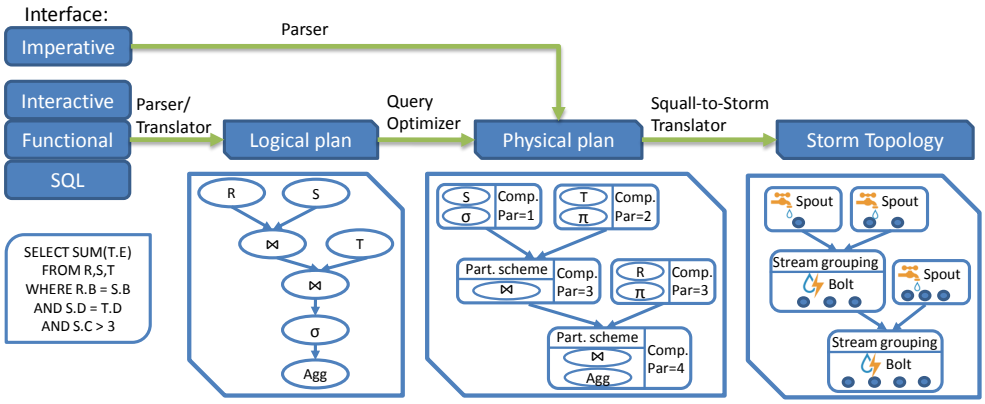


Figure 1: Squall architecture. An example query plan has selections (σ), projections (π), joins ($\bowtie$) and aggregations (Agg).

pipeline of co-located operators as a *component*. Component is an execution unit in a distributed environment, and it can be scaled out to many machines. Figure 1 shows components as rounded rectangles in the example physical plan. An example of a component is a data source (R, S, T in our example) followed by a selection. All the components are continuously processing tuples, and continuously sending the output to the downstream components, if any.

Operators. By combining different partitioning schemes and local join algorithms, Squall offers many join operators. We build novel join operators: Adaptive 1-Bucket [32] and Equi-weight-histogram (EWH) join [66]. This paper also presents some novel multi-way joins (a multi-way join runs within a single component, rather than using a pipeline of 2-way joins). Beside joins, Squall offers database operators such as selections, projections and aggregations (we currently support sum, count and average aggregates). Squall provides both full-history and window semantics for its operators. It implements typical stream primitives, such as tumbling and sliding windows, by adding the window expiration logic on top of the full-history engine.

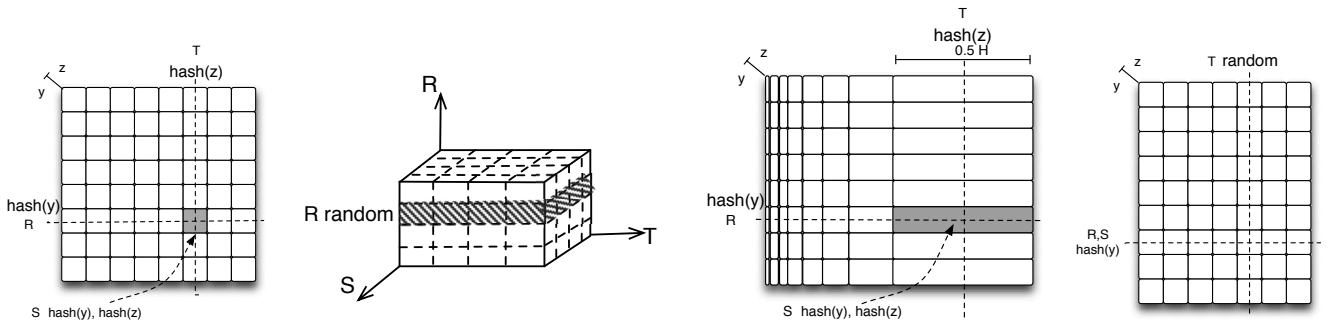
Query optimizer. Squall's optimizer generates a physical plan from the logical plan. The optimizer maximizes throughput and minimizes both latency and the number of machines used. It starts from the data sources and adds the operators one after another, pushing selections and projections as close as possible to the data sources. Where possible, the optimizer co-locates operators to components to minimize network transfers. It also performs common subexpression elimination. That is, if only expressions are used downstream a component in the query plan, the component sends only expressions (rather than the all the corresponding fields). To do so, each component decides on its output scheme based on the fields/expressions that are needed downstream in the query plan. Furthermore, the optimizer assigns the right parallelism to each component, such that a component is neither overloaded nor mostly idle. We refer to this as universal producer-consumer balance. The optimizer uses heuristics to find an optimal join order and component parallelism.

Online processing aspects. An online system must adapt to changing data statistics. Squall collects statistics and adjusts the operator's partitioning scheme at run-time (see §5). Furthermore, it offers multiple partitioning schemes that achieve different levels of adaptivity for different skew types (e.g., data, temporal and join selectivity skew) and degrees of skew fluctuations.

Distribution platform. Squall uses Storm [49] as a distribution platform, but our contributions and ideas are more widely applicable. That is, all the proposed ideas are orthogonal to the underlying system (Storm), and are applicable to other systems as well (Flink, Spark Streaming etc.). In other words, although Squall uses Storm, we could also use Spark Streaming. For our purpose, the two systems are interchangeable, even though they come from different backgrounds, Storm having been developed for realtime processing using certain stream processing abstractions, and Spark Streaming having been developed by modifying Spark, very pronouncedly a batch processing system, to perform online processing. We note that Storm is sometimes called a data stream processor, but we think of it more as an online/realtime analytics system with a very convenient programming abstraction and excellent scalability, since it does not of itself enforce small state or handle overload situations (by load shedding). Along these lines, Akidau *et al.* [11] explain that micro-batch or streaming systems should be equivalent from the user perspective, that is, a user can use either kind of systems to run the same application. The authors state that micro-batch and streaming systems should differ only in the achieved tradeoff between latency, throughput and resource utilization.

In Storm, real-time computation is performed through topologies. Storm executes a topology, which is a graph of spouts (data sources) and bolts (which perform computation). A spout generates a stream(s), where a stream is a sequence of tuples. A bolt consumes streams and produces new ones. Each topology graph node (spout or bolt) executes on one or more machines, according to the requested parallelism. An edge in the topology graph is called stream grouping, and it represents partitioning of incoming tuples from a stream among the machines of a bolt. Squall maps a physical query plan to a Storm topology, each component to a Storm spout or bolt, and builds partitioning schemes using Storm's stream grouping.

Squall is a main-memory system. It also offers connectivity to BerkeleyDB [56], which spills tuples to disk when main memory is insufficient. However, throughput and latency are orders of magnitude better when only main-memory is used. Squall assumes a shared-nothing architecture.



(a) Hash-Hypercube. (b) Random-Hypercube. (c) Hash-Hypercube with skew. (d) Hybrid-Hypercube.

Figure 2: Partitioning schemes for $R(x, y) \bowtie S(y, z) \bowtie T(z, t)$. Uniform data (a), data-independent (b), skewed data (c, d).

3. NOVEL JOIN OPERATORS

We devise new join operators in Squall by wiring up state-of-the-art partitioning schemes and local join algorithms. We presented the partitioning schemes for 2-way joins of Squall in our previous work [32, 66]. This paper introduces multi-way joins in Squall (a multi-way join uses a single communication step, that is, it runs within a single component). These joins can outperform the corresponding pipelines of 2-way joins as they avoid shuffling intermediate data, which can be very large [8, 74, 26]. Multi-way joins are especially beneficial when the output of intermediate stages is big compared to the size of the base relations and/or final output. Even if this is not the case, a multi-way join may outperform the corresponding pipeline of 2-way joins due to the following. An optimal query plan consisting of 2-way joins is very sensitive to the join selectivity of intermediate relations. As a query optimizer typically lacks accurate join selectivity information, it might produce a suboptimal pipeline of 2-way joins. In contract, multi-way joins are inherently resilient to inaccurate statistics [40]. In an online system, the join selectivity might vary substantially. As we explain in §5, we could periodically adjust the join order, but the cost might be unacceptably high due to recomputing large intermediate relations. In contrast, multi-way joins inherently bring adaptivity to join selectivity fluctuations.

We also devise a novel multi-way join partitioning scheme that further enhances performance by taking into account skew degrees of different relation attributes (see §3.1). In particular, our scheme constructs composite partitioning, consisting of different partitioning schemes according to the skew degree in different relation attributes. In addition, Squall has efficient local algorithms for online multi-way joins (DBToaster, see §3.3).

3.1 Partitioning schemes

Next, we describe partitioning schemes for multi-way joins, their skew resilience and supported join conditions. We first present the schemes briefly, along with some examples. A detailed analysis of the multi-way join schemes is in §4.

Hash-Hypercube scheme [8] 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 a query with a join condition $R.y = S.y$ AND $S.z = T.z$. In the further text, we refer to this query as $R(x, y) \bowtie S(y, z) \bowtie T(z, t)$. The Hash-Hypercube scheme is a generalization of hash partitioning to multi-way joins. This scheme assigns an input tuple to machines by hashing on the tuple's join keys and by replicating on the join keys from the other relations. For example, R tuples are hashed on y and replicated on z (each R tuple is replicated to a "row" of machines with coordinates $(y, z) = (hash(y), *)$). Similarly, T tuples are replicated on y and hashed on z (each T tuple is replicated to a "column" of machines with coordinates $(y, z) = (*, hash(z))$). Whereas, S tuples are partitioned using coordinates $(y, z) = (hash(y), hash(z))$. The scheme achieves correctness 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 $(hash(y), hash(z))$.

The operator's performance depends on the slowest machine, that is, the machine with the highest load (number of received input tuples). Thus, the optimization criterion is to choose the dimension sizes, such that we minimize the load per machine. In Figure 2a, given 64 machines and that each relation is of size H and assuming uniform distribution, the dimensions $y \times z = 8 \times 8$ minimize the load. (The dimension choosing algorithm is presented in §4.) Thus, the load of each machine L is $|R|/8 + |S|/(8 \cdot 8) + |T|/8 \approx 0.26H$. The Hash-Hypercube scheme supports skew-free multi-way equi-joins.

Random-Hypercube scheme [74] also models the result space as a hypercube, but each axis corresponds to a relation, as shown in Figure 2b. The Random-Hypercube scheme is a generalization of the 1-Bucket scheme [54], which uses random partitioning over a matrix (2-dimensional hypercube). The Random-Hypercube scheme randomly distributes the input 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 one such slice with diagonally engraved lines). In Figure 2b, given 64 machines and given that each relation is of size H, the dimensions $R \times S \times T = 4 \times 4 \times 4$ minimize the load. (The algorithm for deciding on dimensions is presented in §4.) As each machine receives $1/4$ of each relation, the load per machine is $3 \cdot H/4 = 0.75H$, where the relations are of the same size H. The Random-Hypercube scheme supports multi-way theta-joins and is skew resilient. However, it replicates tuples more than the Hash-Hypercube scheme (because it uses a 3-dimensional rather than 2-dimensional hypercube). The Random-Hypercube scheme is skew-resilient and it achieves perfect load balancing, but at the expense of the excessive tuple replication.

2-way join schemes. For 2-way joins, Hash-Hypercube becomes hash partitioning, and Random-Hypercube becomes 1-Bucket scheme [54], 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 [54] and our Equi-Weight Histogram (EWH) scheme [66]. The M-Bucket scheme is prone to join product skew [67]. 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 and work well only for the cases when skew exists either in all the relations or in none of them. We propose the Hybrid-Hypercube, which uses hash partitioning for skew-free join keys, and random partitioning elsewhere. Random partitioning implies replication, so it is more costly than hash partitioning. That way, our scheme achieves skew resilience while minimizing tuple replication. In the case of equi-joins and skew-free attributes, the Hybrid-Hypercube produces the same partitioning as the Hash-Hypercube. Similarly, in the case of skew on all the join keys, the Hybrid-Hypercube is equivalent to the Random-Hypercube scheme. Thus, our scheme subsumes both the Hash- and Random-Hypercube schemes. Furthermore, in contrast to the Hash-Hypercube, the Hybrid-Hypercube supports non-equi joins (using random partitioning therein). For instance, our scheme works without any change if we have an inequality join condition between S and T [3], bringing the same performance improvement compared to the Random-Hypercube as before.

The Hybrid-Hypercube scheme is illustrated in Figure 2d, and it works as follows. R and S tuples are hashed on y and replicated in the selected "row" of machines. We can consider $R \bowtie S$ as 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, $hash(y)$ from R and S simulates random distribution with respect to T. Thus, we can consider $RS \bowtie T$ as a 1-Bucket join. $RS \bowtie T$ does not indicate the order of execution, but simply the different partitioning schemes employed. We use RS rather than $R \bowtie S$ notation due to the following. As R and S use the same partitioning on y, the replication in 1-Bucket join is the same as if we had a relation of size $R + S$. 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. It is interesting to compare these schemes with respect to total load among all the machines. The Hash-Hypercube total load is $R \cdot 8 + S + T \cdot 8 = 17H$, the total load for the Random-Hypercube is $R \cdot 16 + S \cdot 16 + T \cdot 16 = 48H$, and the one for the Hybrid-Hypercube is $R \cdot 7 + S \cdot 7 + T \cdot 9 = 23H$. Our scheme with slightly higher replication than the Hash-Hypercube (due to using random partitioning on the attributes with skew) achieves the best maximum load per machine among all the three hypercube schemes. This illustrates the tradeoff between replication and skew resilience, which we talk about in a greater detail in §5.

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 $p \times 1 \cdots \times 1$ 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.

Join among multiple relations on the same key appears often in practice. An example is TPC-H [5] Q9, which joins *Lineitem*, *PartSupp* and *Part* on *Partkey*. This allows execution of a multi-way join within the same component, without any replication. Interestingly, the Hash-Hypercube scheme yields the same partitioning, as it uses the join keys as the hypercube axes.

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 [34, 32]. Existing local joins build indexes on the fly (hash indexes for equi-joins, and balanced binary tree indexes for band and inequality joins) to improve performance. For example, let us consider a join condition $R.A = S.A \;\; AND \;\; 2 \cdot R.B < S.C$. In this case, we need to build hash-indexes $R.A$ and $S.A$ and balanced binary tree indexes $R.B$ and $S.C$. Upon tuple arrival, we store the tuple, update all of its indexes, and lookup indexes on the opposite relation in order to produce result tuples.

However, these joins are orders of magnitude slower than the state-of-the-art online local join, **DBToaster** [9]. 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. For

[3] We only need to change the local join implementation to reflect the change in the join condition.

instance, given 4 relations R, S, T, V, DBToaster materializes and maintains $\binom{4}{2}$ 2-way intermediate relations ($R \bowtie S$, $R \bowtie T$, $R \bowtie V$, $S \bowtie T$, $S \bowtie V$ and $T \bowtie V$), $\binom{4}{3}$ 3-way intermediate relations ($R \bowtie S \bowtie T$, $R \bowtie S \bowtie V$, $R \bowtie T \bowtie V$ and $S \bowtie T \bowtie V$) and final result $R \bowtie S \bowtie T \bowtie V$. When a new tuple comes, DBToaster updates the intermediate relations, and produces the (delta) result by joining the incoming tuple with the corresponding $(n-1)$-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. This is why the savings grow with the increase in the number of relations n.

When parallelizing DBToaster, it is challenging to preserve correctness of the result (exactly-once semantics) as tuples (in the Incremental View Maintenance terminology, updates to relations) may arrive in different order to different machines. Existing parallel DBToaster [53] relies on a synchronous system (Spark/Spark Streaming) to circumvent the problem. As we will explain in §8.1, Spark Streaming performs synchronization at the end of each micro-batch, so it cannot achieve low latency of Storm/Squall (tens of milliseconds). Furthermore, in contrast to Squall, existing parallel DBToaster [53] does not focus on skew resilience.

Regarding the system optimizations for local joins, we employ collections of primitive rather than wrapper types using Trove library [6]. Similarly, we store complex data types (such as Strings) as byte arrays. Both of these optimizations bring significant savings in memory consumption. As explained in [52], memory savings can translate to performance improvements.

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. That is, Squall requires no changes in the partitioning scheme and local join when putting them together in a parallel join operator. 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 DBToaster instance 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 and CPU efficiency due to using DBToaster locally.

As we saw in §3.1, the Hybrid-Hypercube subsumes the other two hypercube schemes. Hence, we need to choose the right partitioning type for each Hybrid-Hypercube dimension. As already shown, random partitioning is expensive (because high replication leads to increased work on all the machines) 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. We are interested in the join keys' distribution after applying selection operators over the base relations. In addition, if a relation has only a few distinct join keys, hash partitioning assigns work only to a few machines, leaving the other machines idle. In this case, we consider the relation as skewed, and use random partitioning therein.

Although DBToaster is an online local join operator, our hypercube schemes are applicable both for the offline and online scenarios. We start with the offline scenario.

Choosing among hypercube schemes: offline case. There is a threshold in attribute skew after which random partitioning brings better performance compared to hash partitioning. In offline systems, we can employ sampling and estimate the frequency of the most popular key in the dataset. Sampling incurs negligible overheads compared to the query execution time [71, 30, 55]. To find the optimal partitioning for a hypercube scheme, we run the optimization algorithm twice. In the first run, we simply compute the load after marking the attribute skewed (which enforces using random partitioning). In the second run, we run the optimization algorithm marking the attribute uniform (which opts for hash partitioning). When computing the maximum load for hash partitioning, we take into account the top key frequency, as all the tuples with the same key go to the same machine. In particular, we estimate the maximum load per machine as $(L - L_{mf})/p + L_{mf}$, where L and L_{mf} are the load for all the keys and for the most frequent key, respectively, and p is the number of machines[4]. Finally, we choose the partitioning (hash or random) with the smaller maximum load per machine. Alternatively, we could find out the threshold analytically. In that case, we mark the attribute as skewed or non-skewed using the information from the sample, and we run the optimization algorithm only once.

Choosing among hypercube schemes: online case. A good initial choice of a hypercube scheme saves us from future adaptations. Fortunately, in many cases, even in an online scenario, we know beforehand whether a join key is skew-free. Sometimes we can infer this from the scheme. For example, an attribute with the uniqueness property (such as the primary key) cannot have skew[5]. On the other hand, zipfian skew distributions are typical in many real-life datasets, such as Internet packet traces, city sizes, word frequency in natural languages and advertisement clickstreams [17]. Another example is dealing with chain stores, where we know ahead of time that some stores (e.g., these ones in bigger cities) sell more items than other stores. Similarly, we may know ahead of time that some products are very popular (they are sold much more frequently than other products).

4. MULTI-WAY JOINS: GENERAL CASE

Until now, we illustrated the Hash-Hypercube, Random-Hypercube and Hybrid-Hypercube schemes on a specific 3-way join (see §3.1). Next, we discuss the optimization algorithm for each scheme, which finds an optimal partitioning for a general join. For each scheme, the optimal partitioning produces a partitioning that minimizes the load per machine, and thus, it also minimizes the total amount of replication. We are given p machines, and the produced partitioning is a hypercube where each dimension j is of size p_j, so that $p = p_1 \cdot p_2 \cdot \cdots \cdot p_l$.

Hash-Hypercube. Given relations R_i from the query, where $i \in 1..k$, the formula for load per machine is $L = \sum_i |R_i| / \prod_{j:j \in R_i} p_j$ [8], where hypercube dimension sizes are $p_1 \times p_2 \times \cdots \times p_l$. Given the relative relation sizes (e.g., $|R_i| : |R_i| : \cdots : |R_k| = s_1 : s_2 \cdots : s_k$), the optimization algorithm

[4]We can obtain more precise estimation by using more information from the sample about data distribution, e.g., by using J most popular keys.

[5]This holds for hash partitioning, which is a natural choice in this scenario. If we use range partitioning, we could have skew, depending on the data distribution and range bounds.

chooses the dimension sizes for the Hash-Hypercube so that it minimizes the load per machine. This algorithm is known as the HyperCube algorithm [8, 19].

The formula for load L reflects the fact that 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 hypercube dimensions). In contrast, previous work on the optimization algorithm [8, 18] takes as input all the attributes appearing in the query, which includes both join keys and the attributes from the SELECT clause (GROUP BY, aggregation attributes etc.). Indeed, we discover that using join keys is sufficient, as we will explain shortly. This observation is important as it reduces the input to the optimization algorithm, improving its performance. Using only join keys as the algorithm input also allows us to more easily reason about the optimization algorithms for the Random-Hypercube and Hybrid-Hypercube, as we will see later.

We next explain why it suffices to use only the join keys (rather than all the relation's attributes appearing in the query) as the input in the optimization algorithm. Let us relation R which has attributes x_1, x_2, ... and x_n (these are join keys), and y_1, y_2, ... and y_n (these are non-join attributes). For a fixed number of partitions p for relation R, the load per machine is the same for hypercube schemes with different dimensions from that relation. For instance, the load is the same for a hypercube scheme that uses only x_1 from R where $p = p_{x_1} = 12$, and for a scheme that has x_1, y_2 dimensions from R where $p = p_{x_1} \cdot p_{y_2} = 3 \cdot 4$. In other words, the load per machine due to relation R depends only on the number of partitions p for that relation, and not on the number of hypercube dimensions. On the other hand, only the joins keys increase the number of partitions (and reduce the load) for other relations (the ones that share the same join key). Thus, for each hypercube partitioning that contains non-join attributes, there is one which uses only join keys as dimensions, which is at least as good as the partitioning with non-join attributes. In other words, the algorithm always chooses the join keys as the hypercube dimensions, as this allows partitioning two (or more) relations with a single attribute (join key).

There are different versions of the Hash-Hypercube optimization algorithm. The original one [8] is computationally expensive as it solves a system of non-linear equations in order to find optimal dimension sizes. Beame *et al.* [18] address the efficiency problem by translating the non-linear to a linear system of equations by using some mathematical transformations. Namely, the authors express the dimension sizes in an exponential form, and then take a logarithm over the obtained mathematical expressions. The full details are out of the scope of this paper. We refer an interested reader to [18]. Unfortunately, as explained in [26], both works [8, 18] do not handle the case when dimension sizes (obtained from solving the equations) are not integers. For instance, if we have 7 machines in total and 3 dimensions of the same size, each dimension is of size $7^{1/3} = 1.91$. If we round down this value, we fall back to sequential execution (using only 1 machine), completely wasting the remaining 6 machines. Chu *et al.* [26] propose an algorithm that always proposes integer dimension sizes. To do so, the authors use breadth-first search to explore different configurations whose total number of machines is less or equal than the given number of machines. Then, the algorithm chooses a configuration with the smallest load per machine.

To our knowledge, we are the first to introduce the terms Hash-Hypercube and Random-Hypercube, and to discover and analyze the common structure between the two schemes in a principled way.

Random-Hypercube. The problem formulation is similar as before, except that the dimensions correspond to the relations themselves, rather than to the join keys. The load per machine is equal to $\sum_i |R_i|/p_i$ [74], as each relation randomly chooses a position on its own dimension, and replicates among the other dimensions. As shown in [74], the optimal hypercube is the one that divides its dimensions into segments of equal size, that is, $|R_1|/p_1 \approx |R_2|/p_2 \approx \cdots \approx |R_k|/p_k$. In other words, in the optimal partitioning, the dimension sizes are in the same proportion as the relation sizes. For example, if we have 64 machines and R_1 is $4\times$ bigger than R_2, the optimal partitioning is $\{R_1 \times R_2\} = \{16 \times 4\}$. This 16×4 partitioning implies the minimal load per machine and minimal communication cost among all the possible Random-Hypercube partitionings for the given proportion among the relations sizes.

A part of our contribution is discovering a technique for translating the Random-Hypercube partitioning problem to that of the Hash-Hypercube. That is, we express the join $R_1 \bowtie R_2 \bowtie \cdots \bowtie R_k$ as $R_1(x_1), R_2(x_2) \ldots R_k(x_k)$, where x_i are quasi-attributes that we use as the dimensions in the Hash-Hypercube optimization algorithm. As no attribute appearing in more than one relation, and each relation has exactly one attribute, the resulting partitioning scheme is the same as the one produced by the Random-Hypercube algorithm [74] for the given number of machines[6]. After we compute the dimension sizes using the Hash-Hypercube optimization algorithm, we use random rather than hash partitioning on each dimension.

Hybrid-Hypercube. To decide on dimensions and their sizes for a general multi-way join, we extend the optimization algorithm for the Hash-Hypercube. Let us first more closely look at query $R(x, y) \bowtie S(y, z) \bowtie T(z, t)$ from §3.1. The corresponding Hybrid-Hypercube partitioning scheme is shown in Figure 2d. We obtain this partitioning by using join key renaming[7] and by assigning each join key name to a separate hypercube dimension. In particular, given that there is skew on $S.z$ and $T.z$, we rename them to z' and z'', respectively. To address skew at join execution time, we use random partitioning on the renamed attributes z' and z''. We have to use different attribute names (z' and z''), otherwise the optimization would use the same dimension for $S.z$ and $T.z$, and as we are using random partitioning on both attributes, we would miss many result tuples. As we use separate dimensions for $S.z$ and $T.z$, and on each of them we employ random partitioning, this implies that we perform $S \bowtie T$ using the 1-Bucket scheme. On the other hand, we join R and S using hash partitioning, given that they share a common skew-free attribute y.

As we already discussed, we need to provide only join keys (rather than all the attributes from the query) as the

[6]Work [74] has an additional optimization criterion of finding the optimal operator parallelism. In our work, we assume that the number of machines is given ahead of time.

[7]The renaming is used only in the optimization algorithm and the partitioning scheme. The local joins are unchanged.

input for the optimization algorithm. In our example, and after renaming, the input for the optimization algorithm is $R(y), S(y, z'), T(z'')$. Interestingly, the fact that renamed attributes z' and z'' use random rather than hash partitioning changes nothing in the formulas for the dimension sizes from the optimization algorithm. This is because we care only about equal distribution of tuples among the rows and columns. It is irrelevant for the formulas whether we achieve this using a hash function on a uniform dataset or by randomization. Thus, from the viewpoint of the Hash-Hypercube optimization algorithm, we can consider a renamed equi-join $R(x, y) \bowtie S(y, z') \bowtie T(z'', t)$ as an equi-join with hypercube dimensions (y, z', z'').

The Hybrid-Hypercube partitioning from Figure 2d has 2 rather than 3 dimensions (y, z', z''). The reason is the following. As we already discussed, an optimal partitioning includes only join keys, that is, the attributes that appear in multiple relations. Given that z' only appears in relation S, and that this relation is already partitioned by y attribute (which is a join key appearing also in R relation), the optimization algorithms sets the dimension size of z' to one, effectively removing it from the hypercube dimensions. On the other hand, although z'' is also appearing only in a single relation, it is the only attribute that partitions the relation T. Thus, attribute z'' remains in the final (y, z'') partitioning, which corresponds to our Hybrid-Hypercube from Figure 2d. Each tuple from R or S is hashed on y and replicated on z''. Whereas, we randomize T on z'' and replicate it on y. In other words, we perform replicated hash join between R and S, and a 1-Bucket $RS \bowtie T$ join. By doing so, the Hybrid-Hypercube saves one hypercube dimension compared to the Random-Hypercube (which directly translates to smaller amount of replication and thus better performance), while still providing for skew resilience.

Continuing this example, for certain relative relation sizes, a partitioning may become a 1-dimensional one. For instance, if T is really small compared to R and S, the optimal partitioning (with respect to the minimal load per machine) is (y), which implies broadcasting relation T. A nice property of our Hybrid-Hypercube is that it automatically handles all these cases. A user needs to provide only the relation sizes and whether each join key is skew-free or not.

Let us now consider a query $R(x, z) \bowtie S(y, z) \bowtie T(z, t)$ in which only $T.z$ is skewed. In this case, we rename only $T.z$ to z' and use random partitioning therein. This allows us to share z attribute among R and S relations, lowering the amount of replication required. From the perspective of the 1-Bucket join $S \bowtie T$, we simulate random distribution on $S.z$ using $hash(S.z)$, as $S.z$ is a skew-free attribute. In general, we rename attributes and create new hypercube dimensions only when necessary (that is, in the presence of skew), allowing sharing of attributes among different relations whenever possible.

The Hybrid-Hypercube can save more than one hypercube dimension compared to the Random-Hypercube scheme. 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 (each corresponding to one relation), while the Hybrid-Hypercube uses only 2 dimensions (one on y attribute, and another on t). In particular, the Hybrid-Hypercube hashes R and S on attribute y to "rows" of the 2-dimensional hypercube (matrix), and T and U on t to "columns". In other words, we perform replicated hash join for $R \bowtie S$ and $T \bowtie U$, and a 1-Bucket join $RS \bowtie TU$. In order to partition the data equally using the 1-Bucket join, hashing on $S.y$ needs to produce a similar effect as random partitioning on $S.z$ (the same should hold for $T.t$ and $T.z$ attributes, respectively). This holds, as there is no skew on $S.y$ nor on $T.t$. Thus, we can apply dimensionality reduction in multiple places in the query. In general, with the increase in the number of relations (dimensions), the potential of our hypercube scheme for saving dimensions (and reducing replication) grows. Similarly, increasing the number of relations in a pipeline of 2-way joins implies network transferring of more intermediate relations, while the corresponding hypercube scheme transfers no intermediate relations at all. On the other hand, given a fixed number of machines, increasing the dimensionality of any hypercube scheme (including ours) leads to higher replication. This is due to the fact that more dimensions have to share the same total number of machines.

Next, we analyze the Hybrid-Hypercube optimization algorithm for queries with non-equi joins. Let us consider a query $R.x = S.x$ and $S.x < T.y$. From the perspective of the optimization algorithm, we can consider this query as an equi-join $R(x), S(x), T(y)$ and dimensions (x, y) [8]. We do not require any renaming, and we use hash partitioning for both x and y. Hash partitioning on $S.x$ allows us to reuse the same dimension for $R.x$ attribute. From the perspective of 1-Bucket join $S \bowtie T$, we simulate random distribution on $S.x$ using $hash(S.x)$, given that $S.x$ is a skew-free attribute. Similarly, we simulate random distribution on $T.y$ using $hash(T.y)$, given that $T.y$ is a skew-free attribute[9]. That way, we perform a replicated hash join $R \bowtie S$ and an 1-Bucket join $RS \bowtie T$. In other words, the resulting partitioning scheme replicates R and S over a "row" of machines in the matrix (2-dimensional hypercube), and it replicates T over a "column" of machines.

Continuing this example, let us assume that there is skew on $T.y$. The dimensions (x, y) and their sizes are the same as before. The only difference is that we need to employ random (rather than hash) partitioning on $T.y$. On the other hand, if there is skew only on $S.x$ we need to rename this attribute to x', and the optimization algorithm produces a hypercube with (x, x', y) dimensions, using hash, random and hash partitioning, respectively. In that case, attributes $R.x$ and $S.x$ correspond to different dimensions, and we employ random partitioning over the renamed attribute $S.x$ in order to handle skew.

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 equi-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 GROUP BY/join 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

[8]These changes are only for the optimization algorithm. The local joins are unchanged.

[9]We could as well use random partitioning on $T.y$ in order to more closely mimic 1-Bucket partitioning for $S \bowtie T$. In that case, we do not pay any extra replication cost, as there are no other y attributes in the query.

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 $p = 8$, an optimal scheme will assign no more than $\lceil 15/8 \rceil = 2$ keys to each machine. However, due to imperfections of hash functions, it is very likely that some machine is assigned 3 keys, leading to 1.5× higher maximum load per machine than in an optimal case. This causes severe performance degradations. The performance gap deepens for $d = p$, as it becomes very likely that one machine is assigned 2 keys (keeping another machine completely idle), while an optimal assigns exactly 1 key per machine. The machine which is assigned two times more work becomes a bottleneck. This results in a largely suboptimal query plan in terms of resource utilization, throughput and latency.

Unfortunately, suboptimal assignments due to a small number of distinct keys d happen frequently in practice. For example, many queries from the TPC-H benchmark [5] (e.g. Q4, Q5, Q12) have final aggregations with only up to 25 distinct values. In particular, Q4, Q12 and Q5 have 5, 7 and 25 distinct values, respectively.

On the other hand, we typically know all the distinct values for attributes with a small domain (e.g. possible values for ship priorities in TPC-H are predefined). Squall uses this information to optimally assign distinct values and to achieve perfect load balancing[10]. Before the execution starts, Squall creates a mapping from the predefined keys to the machines using a round robin partitioning.

Temporal skew. There is another type of skew called temporal skew, where it does not suffice for skew resilience to have the exact data distribution (even in the case of uniform distribution). Temporal skew occurs when a specific tuple arrival order causes load imbalance. In contrast to skew due to hash imperfections, temporal skew occurs only in online systems. Different partitioning schemes have different properties with respect to temporal skew. Partitioning schemes are commonly classified [60] to content-sensitive schemes (e.g., joins with hash or range partitioning) and content-insensitive schemes (e.g., 1-Bucket scheme [54], which uses random partitioning). Content-sensitive schemes are prone to temporal skew. In particular, for hash partitioning, in the case of sorted tuple arrival and moderate join key frequencies, only one machine will be active at a time. This is equivalent to a sequential execution. We denote imbalance in load caused by tuple arrival order as temporal skew. Range partitioning is also prone to temporal skew. In the case of range partitioning and sorted or nearly sorted tuple arrival (e.g., a timestamp is the join key), only a few machines at a time perform some work. In the context of hypercube schemes, each scheme that uses hash partitioning on at least one dimension (with size greater than 1) is considered content-sensitive. On the other hand, content-insensitive schemes use random partitioning and they are resilient to temporal skew. Namely, these schemes perform 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 can do indirectly by monitoring the machine load[11]. To achieve good performance, we recommend using random partitioning schemes in the case of data or temporal skew (or both).

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 (e.g., when a different key becomes the one with highest frequency, or when the skew degree changes). If changes are occurring frequently, the operator spends a large amount of time on state relocations over the network. Even worse, an adversary can change the data distribution right after the system adjusts the scheme, thus causing the scheme to always be highly suboptimal. The random partitioning avoids this problem as it randomly assigns tuples to machines, essentially removing skew in the data distribution.

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 order in an optimal query plan consisting of 2-way joins is very sensitive to the join selectivity of intermediate relations. In other words, a small change in the join selectivity may cause another join order to become an optimal one. In online systems, the join selectivity for 2-way joins can vary at run-time. Furthermore, some intermediate relations may grow very large [8, 74, 26].

A possible response is adaptive join reordering [33]. 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. For this reason, existing online systems typically do not perform join reordering at run-time. Squall also do not reorder join at run-time, but it offers resilience to join selectivity fluctuations through multi-way joins.

In contrast to a pipeline of 2-way joins, a multi-way joins consists of a single join operator, so there is no need for join reordering. Furthermore, a multi-way join does not need to change which intermediate relations are materialized (e.g., $R \bowtie S$ to $S \bowtie T$ in the example above), nor to send the intermediate results over the network. Thus, in contrast to a pipeline of 2-way joins, hypercube schemes inherently bring adaptivity to the join selectivity fluctuations.

SAR principle. We introduce the SAR principle, which summarizes this section. To achieve **S**kew-resilience and **A**daptivity for more skew types in an online system, partitioning schemes need to increase the input tuple **R**eplication. Namely, for 2-way joins, hash partitioning (e.g., [34]) is prone to skew but requires no replication (hash partitioning is limited to equi-joins). Whereas, with small amount of replication, range partitioning provides resilience to redistribution skew (e.g., M-Bucket scheme [54]), or to both redistribution and join product skew (e.g., our equi-weight histogram scheme [66]). Unfortunately, range partitioning is prone to temporal skew and skew fluctuations. Random partitioning (e.g., 1-Bucket scheme [54]) is resilient to data and temporal skew and skew fluctuations, but it requires a

[10]The optimal assignment for uniform distribution is as follows. If the number of different values is divisible by the number of machines, all the machines should be responsible for the same number of values. Otherwise, the number of values should not differ by more than one between any two machines.

[11]This requires that the partitioning scheme reflects the actual data distribution.

higher amount of replication compared to the one from a range partitioning scheme. A multi-way join brings adaptivity to join selectivity fluctuations. A Random-Hypercube multi-way join is resilient to all the skew types. However, it requires higher replication than in the 1-Bucket scheme [54] due to the following. Both in 1-Bucket and multi-way joins, in order to produce the join result without requiring communication among joiner machines, a potential output tuple and all its corresponding input tuples are assigned to a single machine. Given more relations in the join, a single tuple needs to join with more tuples from other relations, effectively increasing replication. (On the other hand, pipeline of 2-way joins may incur higher total network cost compared to a multi-way join due to transferring large intermediate results over the network.)

Related work. There is a lot of work on adapting to changing input rates [61, 37, 39]. However, these works focus on a single-machine scenario, and optimizing the local join algorithm accordingly. In contrast, we introduce skew fluctuations and temporal skew, which concerns changing data distribution, and influences the choice of optimal partitioning scheme. Flux [60] introduces transient skew which is essentially a short-term temporal skew. The authors of [60] propose processing tuples out of order from buffers. This does not address a general case of temporal skew because all the tuples in the buffer can have a single destination. Furthermore, Flux does not discuss the behavior of different partitioning schemes with respect to transient skew. In contrast, we reveal that only content-insensitive schemes can address temporal skew.

Regarding the SAR principle, we are the first to formalize it. The trade-off between skew-resilience and replication was known from before, both in the context of offline (e.g., 1-Bucket scheme [54]) and online processing (e.g., [33]). In contrast to the previous work, we observe the connection between adaptivity on one side, and skew-resilience and replication on the other side. In addition, we provide classifications of different partitioning schemes according to their properties regarding skew-resilience (for different types of skew), adaptivity and replication.

Hypercube sizes. The optimal hypercube dimension sizes minimize replication, and thus, maximize performance. We determine the optimal sizes from the relative base relation sizes, as explained in §4. In an online system, the relative sizes may change at run-time. In that case, a hypercube scheme needs to adapt to these changes. Squall implements an adaptive 1-Bucket join operator [32].

Fault tolerance. Squall uses Storm features 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[12]. 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.

[12]https://gist.github.com/jboner/2841832

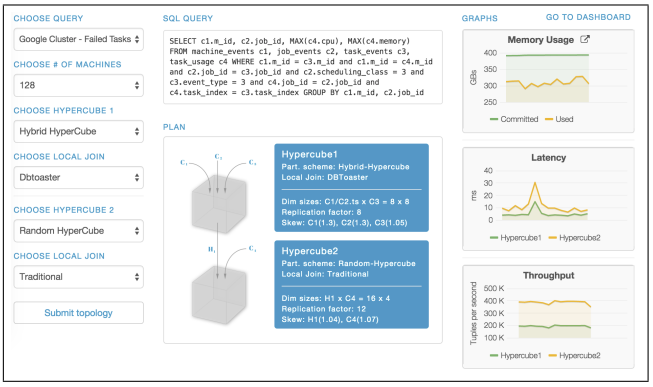


Figure 3: Demonstration: Running a query.

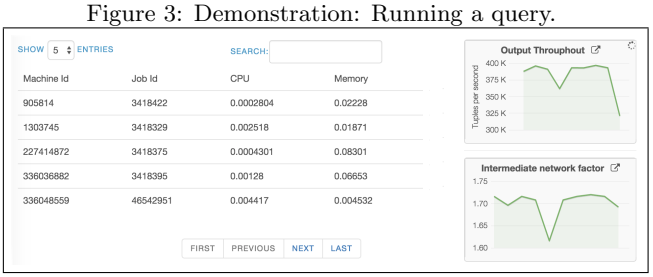


Figure 4: Results and query performance metrics.

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

Google cluster monitoring data[13] 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 finding machines that are not production-ready, that is, *List the machines which often fail tasks belonging to production jobs*. This is a 3-way join between jobs, tasks and machines relations. Another interesting query is *Measure the scheduling algorithm quality*. A motivation for this query is in the fact that the scheduling algorithm might perform badly for a particular (rare) event order, and this can manifest only in production. Schedulers assign jobs to machines to maximize "goodness" score [64], which includes the machine's number of preempted or failed tasks, (production) jobs distribution across the cluster, presence of application dependencies, cluster failure domains etc. For instance, it is particularly important to assign production jobs to machines with high "goodness" score. Computing the score involves joining multiple relations. We can observe the scheduling algorithm quality by monitoring (in real-time) the score aggregated over jobs and machines.

Demonstration. As shown in Figures 3 and 4, we allow attendees to specify a query and to try out different partitioning schemes (Hash-Hypercube, Random-Hypercube, Hybrid-Hypercube), local joins (traditional joins, DBToaster) and the parallelisms (number of machines). Attendees can verify

[13]https://github.com/google/cluster-data

scalability by changing the number of machines for a topology. With a button click, the attendees run the specified query plan on an in-house cluster with 220 hardware threads. At run-time, they can 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 degree. The replication factor is the component's number of input tuples divided by the total number of tuples produced by the immediate upstream components. The replication factor is an online counterpart of the MapReduce replication rate defined in [59] as the proportion between the output and input size of the mappers in terms of number of tuples. 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 tuples (as we 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. The results validate the SAR principle and suggest that replication is ubiquitous for reliable load balancing.

CPU-bound 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 (DBToaster, traditional joins). The attendees can also see the correlation among the operator's memory consumption and throughput. To estimate the network share, we run the query plan with the same local joins but with different partitioning schemes. For instance, we replace a Hash-Hypercube with a Random-Hypercube scheme. We quantify the difference among the query plans (of the same query) using *intermediate network factor* which we define as the sum of all the component tasks' input and output divided by the sum of the query input and query output, that is, $(\sum_{comp.\ task\ t} input_t + output_t)/(query\ input + query\ output)$. The intermediate network factor represents the amount of intermediate network shuffling. Then, we compare the performance among different query plans (of the same query) as a function of this factor. The attendees can also verify on real-world queries and datasets that query plans with multi-way joins frequently outperform the ones with a pipeline of 2-way joins due to network savings.

There is an alternative way to find out if Squall query plans are CPU-bound or network-bound. We run a query plan and starting from data source reading, we add a single element (computation or network). We illustrate this process in Figure 5 on the example of $Customer \bowtie Orders$ from the TPC-H [5] dataset. For some data points, we run the query with a no-op selection (no tuples are filtered out) in order to estimate the computation cost of selections. The full join has no selections. From the first three bars, the cost of a selection over an integer field is only 1.6% of the entire execution. Whereas, the cost of a selection over a date field is about 16%. This is because the creation of a Date instance (from an input String) is much more expensive than the creation of an integer. From the last two bars, we extract the cost of network transferring and join computation. The network transferring takes 60% of the entire execution. Whereas, the join computation takes only

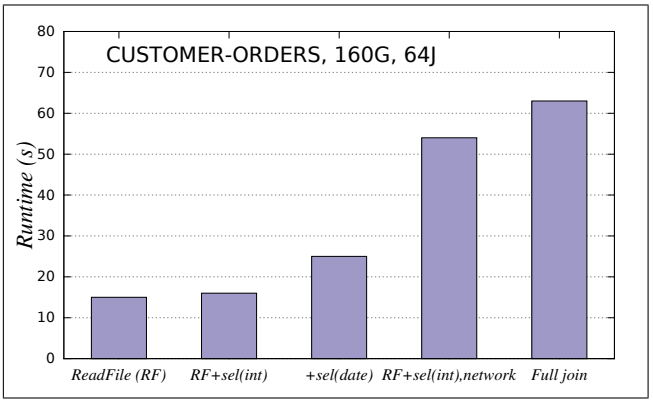


Figure 5: Finding bottleneck in a Squall query plan. *sel* stands for a no-op selection (it passes through all the tuples).

14% of the full join execution. Thus, Squall/Storm is clearly network-bound. The input throughput of the full join is 4.19 million tuples per second for the entire cluster, and 65.000 tuples per second per join task (hardware thread). Hence, our operators are fairly efficient.

7. EVALUATION

We evaluate different hypercube schemes (Hash-Hypercube, Random-Hypercube and our Hybrid-Hypercube) for multi-way joins. We also run the corresponding pipelines of 2-way joins, where each 2-way join uses hash partitioning in the case of skew-free equi-joins, otherwise it uses the 1-Bucket partitioning. Furthermore, we compare the performance among multi-way joins with the same hypercube scheme but different local joins (DBToaster and traditional local joins).

Environment. We perform our experiments on an Oracle Blade 6000 server with 10 Oracle X6270 M2 blades. Each blade has two 3Ghz 6-core Intel Xeon X5675 CPUs. Each blade runs Ubuntu 12.04 and has 72GB of DDR3 RAM and a 1Gbit Ethernet interface. Later on, by a machine assigned to an operator, we mean a core with an exclusively assigned portion of the blade main memory. In this paper, we run the experiments using SQUALL based on STORM[14] version 0.9.4. Squall runs in Java JRE v1.7.

7.1 Datasets

We show the performance of our multi-way join operators both on TPC-H and on real-world datasets. The first dataset is the Hyperlink Graph of the Web from August 2012 Common Crawl Corpus [2]. In the further text, we call this dataset *WebGraph*. The WebGraph dataset has one relation with {FromUrl, ToUrl} pairs, and it is available for different domain aggregation levels. We experiment on the "Host" and "Pay-Level-Domain" aggregation levels.

Another dataset that we use is *CrawlContent*, which has crawled content from a large number of web pages [1]. We can analyze the crawled content using different tools, such as Readability test or Sentiment analysis tools. In the further text, *CrawlContent* refers to a relation with the schema $\{Url, Score\}$, where *Score* stands for the output of any text analysis tools. As the text analysis tools are out of the scope of this work, and the *Score* is not a join key (it is used only in

[14]http://storm.apache.org/

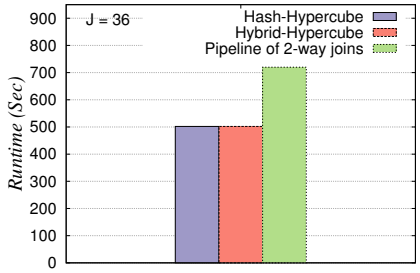


Figure 6: Performance for 3-reachability query. We use 36 joiner machines.

some aggregations), the query performance does not depend on the $Score$ values. Thus, we synthesize them.

Finally, we use publicly available Google cluster monitoring dataset[15], which we describe in §6.

7.2 Multi-way vs 2-way joins

Multi-way joins may outperform the corresponding pipeline of 2-way join, even if the corresponding pipeline is the optimal one.

3-Reachability Query. We illustrate this for a 3-step reachability query over the WebGraph dataset. The SQL of this query is shown below:

3-Reachability

```
SELECT W1.FromUrl, COUNT(*)
FROM  WebGraph as W1, WebGraph as W2, WebGraph as W3
WHERE W1.ToUrl = W2.FromUrl AND W2.ToUrl = W3.FromUrl
GROUP BY W1.FromUrl
```

This query frequently occurs in practice, as it helps to understand the structure of the web. We could run the same query (with W3.ToUrl in the SELECT and GROUP BY clause) over the clickstream data, and use the query result to suggest a better list of hyperlinks for each website. In particular, a user may considering adding a direct link from W1.FromUrl to W3.ToUrl, if the corresponding count aggregate value is high.

Hypercube properties. As the query contains only equi-joins, and the dataset is uniform, the Hash-Hypercube and Hybrid-Hypercube schemes produce the same partitioning. Given 36 joiners, the optimal partitioning is a 2-dimensional hypercube (matrix) $W1.ToUrl \times W2.ToUrl = 6 \times 6$, as W1, W2 and W3 are of the same size. This partitioning implies that W1 is hashed $W1.ToUrl$ and replicated on $W2.ToUrl$, W3 is hashed on $W2.ToUrl$ and replicated on $W1.ToUrl$, and W2 is hashed on both $W1.ToUrl$ and $W2.ToUrl$. Thus, the replication factor is $6 + 6 + 1 = 13$, and total network transfer due to reshuffling data is $13 \times 10.2M = 132.6M$ tuples. We run the query on 0.5% sample of the "Host" WebGraph (the full "Host" dataset has 2,043 million arcs, so the sample has 10.2 million arcs), so that the pipeline of 2-way joins can also finish (otherwise, it runs out of memory due to large intermediate results). The total network transfer in the pipeline of 2-way joins is $3 \times 10.2M + 130M = 160.6M$ tuples ($130M$ is the intermediate output of the first join).

[15]https://github.com/google/cluster-data/blob/master/ClusterData2011_2.md

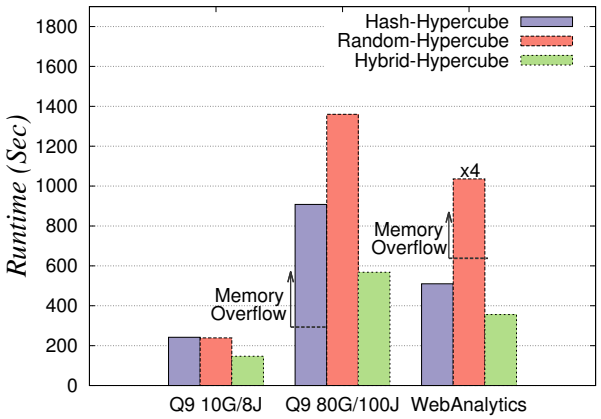


Figure 7: Comparison of different hypercube schemes.

Performance results. As Figure 6 shows, our multi-way join outperforms the corresponding pipeline of 2-way joins by 1.43×. This is because it transfers less tuples over the network compared to the corresponding pipeline ($132.6M$ tuples compared to $160.6M$ tuples). In both cases, we use DBToaster as the local join operator. The speedup comes from the fact that the intermediate results are quite large with respect to the input relations. Thus, the shuffling cost of the pipeline of 2-way joins surpasses the replication cost of a hypercube.

7.3 Hybrid-Hypercube versus Hash-Hypercube and Random-Hypercube

Next, we show two queries where our schemes outperforms the-state-of-the-art multi-way join partitioning schemes. All the multi-way join operators use DBToaster locally.

TPCH9-Partial Query. The first query is a subquery $Lineitem \bowtie PartSupp \bowtie Part$ from the TPC-H [5] Q9. We refer to this query as *TPCH9-Partial*. TPCH9-Partial is an example of a query where multiple relations share the same join key ($Partkey$). Wu *et al.* [70] showed that the Hash-Hypercube scheme outperforms the corresponding pipeline of 2-way joins for TPCH9-Partial on an uniform TPC-H dataset. Indeed, the Hash-Hypercube and the Hybrid-Hypercube produce the same partitioning (the hypercube is 1-dimensional with $Partkey$ as the key).

Hypercube properties. However, for a skewed TPC-H dataset, the Hybrid-Hypercube outperforms both the Hash-Hypercube and Random-Hypercube schemes. We experiment with different configurations (J is the number of machines): $10G/8J$ and $80G/100J$ TPC-H datasets with zipfian distribution and skew factor of 2. The Hash-Hypercube scheme partitions all the relations on $Partkey$, as all the three relations use this attribute as a join key. The Random-Hypercube scheme uses relations as hypercube dimensions and it produces partitioning $Part \times PartSupp \times Lineitem$ with the dimensions $\{1 \times 1 \times 8\}$ for the $10G/8J$ configuration (broadcasting two smallest relations) and $\{1 \times 4 \times 25\}$ for the $80G/100J$ configuration. Due to skew in $Lineitem.Partkey$, the Hybrid-Hypercube schemes uses random partitioning on $Partkey$ and hash partitioning on $Suppkey$. In particular, our Hybrid-Hypercube scheme produces $Partkey \times Suppkey$ partitioning with the dimensions $\{1 \times 8\}$ for the $10G/8J$ configuration and $\{1 \times 100\}$ for the $80G/100J$ configuration.

Table 1: Maximum and average load per machine for different hypercube schemes. M stands for millions of tuples.

Query	Size	Machine Load	Hypercube type		
			Hash	Random	Hybrid
TPCH9-Partial	10G	Maximum	38.5M	15.6M	22.8M
		Average	8.5M	15.6M	8.6M
TPCH9-Partial	80G	Maximum	N/A	35M	78.9M
		Average	N/A	35M	6.3M
WebAnalytics	Pay-Level-Domain	Maximum	2.26M	N/A	2.07M
		Average	2.18M	N/A	2M

Table 2: Replication factor for different hypercube schemes.

Query	Size	Hypercube Replication factor		
		Hash	Random	Hybrid
TPCH9-Partial	10G	1	1.83	1.01
TPCH9-Partial	80G	N/A	6.19	1.11

Performance results. Figure 7 shows the performance results. For query TPCH9-Partial and the $80G/100J$ configuration, the Hash-Hypercube does not complete the processing due to high memory requirements caused by high skew. However, we extrapolate its completion time using the information about the number of tuples processed before running out of memory. The Hybrid-Hypercube outperforms the Random-Hypercube by a factor of 2.39× and the Hybrid-Hypercube by 1.6×. This is due to the fact that our scheme uses hash partitioning whenever possible (on $Suppkey$) and random partitioning only when necessary due to high skew (for $Partkey$).

WebAnalytics Query. The second query that shows the advantages of our Hybrid-Hypercube scheme is over the Pay-Level-Domain WebGraph and CrawlContent datasets. It reports hyperlink paths from the WebGraph dataset that have length of two and that go through 'blogspot.com' (which has the highest in-degree in the dataset), and joins the result with the CrawlContent relation that has URL and a web page content score. The SQL for this query is shown below:

WebAnalytics

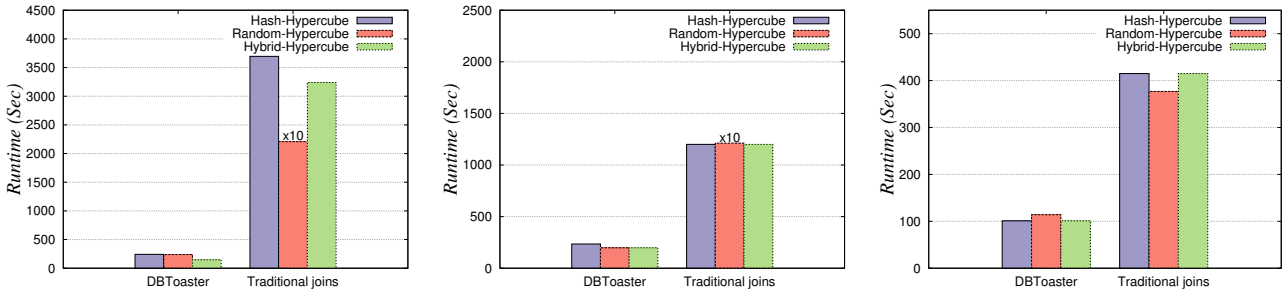
```
SELECT W1.fromUrl, Score, COUNT(*)
FROM  WebGraph as W1, WebGraph as W2, CrawlContent as C
WHERE W1.ToUrl = 'blogspot.com' AND W2.FromUrl = 'blogspot.com'
AND W1.ToUrl = W2.FromUrl AND W1.FromUrl = C.Url
GROUP BY W1.fromUrl, Score
```

Hypercube properties. The size of WebGraph relation is 623 million arcs. After applying selections, the size of W1 and W2 is 1.03 and 3.9 million arcs, respectively. The CrawlContent relation has 43 million tuples (this is the number of distinct Urls from the Pay-Level-Domain WebGraph dataset). We compare the performance using 40 machines for each hypercube scheme. The Hash-Hypercube scheme employs a 2-dimensional hypercube with dimensions $W1.FromUrl(C.Url) \times W2.FromUrl(W1.ToUrl) = \{20 \times 2\}$. Relation $W1$ is partitioned among the machines using its FromUrl and ToUrl attributes. Relation $W2$ is hashed on $W2.FromUrl$ and replicated on $W1.FromUrl$ attribute. Whereas, relation C is hashed on $C.Url$ and replicated on $W2.FromUrl$ attribute. The Random-Hypercube scheme creates a 3-dimensional hypercube $W1 \times W2 \times C = \{1 \times 2 \times 20\}$. This schemes uses replication on all the dimensions, and relation $W1$ is replicated on all the machines. The Hybrid-Hypercube scheme creates a 2-dimensional hypercube with dimensions $W1.FromUrl(C.Url) \times W2.FromUrl = \{20 \times 2\}$. This scheme opts for random partitioning on $W2.FromUrl$ (this is optimal because WebGraph is highly skewed, as there is only one distinct value of this join key) and hash partitioning on $W1.FromUrl$ attribute (this is optimal because there is no skew on $W1.FromUrl$ and this attribute is the primary key in CrawlContent, so it is skew-free). In other words, the Hybrid-Hypercube scheme performs a replicated hash join $W1 \bowtie C$ and a 1-Bucket join $W1C \bowtie W2$. We use DBToaster as the local join operator for all hypercube schemes.

Performance results. Figure 7 shows the performance results for the WebAnalytics query. As this query takes more than an hour to execute, we show the runtime for producing the first 6.5 million output tuples (this gives us comparable running times to the ones from the TPCH9-Partial query). The Hybrid-Hypercube achieves 1.43× speedup compared to the Hash-Hypercube, and 11.64× speedup compared to the Random-Hypercube (we extrapolate its running time). This is due to the fact that, among the hypercube schemes, only our Hybrid-Hypercube scheme is able to employ different partitionings for different attributes. Furthermore, our scheme does so in an optimal manner.

Relationship between maximum load per machine and performance. To better understand the performance differences and skew resiliency among different hypercube schemes, we also extract the maximum and average load per machine in terms of number of input tuples received. Table 1 shows these numbers. From these numbers we can also extract skew degree, which we define in §6 as the division between the maximum and average load per machine. Due to the fact that the Hash-Hypercube does not address skew, it has very high maximum load compared to the average load per machine. This scheme does not finish for the TPCH9-Partial $80G$ configuration, and this is why we cannot obtain its maximum and average load for this configuration. In contrast, the Hybrid-Hypercube addresses skew and thus it has smaller maximum load per machine than the Hash-Hypercube scheme. This explains why the Hybrid-Hypercube outperforms the Hash-Hypercube scheme, as Figure 7 shows. For the WebAnalytics query, it is interesting that a relatively small difference in the load (1.09×) among the Hash-Hypercube and Hybrid-Hypercube schemes leads to a considerable difference (1.43×) in the performance. This is due to the fact that this query is CPU-intensive (each incoming tuple incurs considerable computation).

The Random-Hypercube always achieves perfect load balancing due to randomization of all the input tuples. This is



(a) TPCH9-Partial 10G/8J (b) TPC-H Q3 10G/8J (c) Google TaskCount 8J

Figure 8: Multi-way joins with different local joins (traditional vs DBToaster).

why the maximum and average load per machine are always the same for this scheme, but average load is rather high. In contrast, the Hybrid-Hypercube scheme replicates tuples only when necessary, and thus it has smaller average load per machine than the Random-Hypercube scheme. On the other hand, for TPCH9-Partial, the Hybrid-Hypercube has higher maximum load per machine than the Random-Hypercube, as *Suppkey* does not have completely uniform distribution (the skew is not high enough to justify using randomization on that attribute) [16]. Still, the Hybrid-Hypercube outperforms the Random-Hypercube, as Figure 7 shows. Thus, when choosing an optimal scheme, we need to consider not only the actual maximum load per machine but also the total communication cost (which is the average load multiplied by the number of machines), as network may be a bottleneck (i.e., network might be unable to sustain high enough throughput among all the communicating machines).

A closer look at the replication factor. Table 2 shows the replication factor for different hypercubes in TPCH9-Partial. We define the replication factor in §6 as the ratio between the total number of tuples the the component receives and the number of tuples that the immediate upstream components (in this case, data sources) produce. A small replication factor implies low network traffic as well as small amount of local join processing. Table 2 illustrates that not only the Hybrid-Hypercube has lower replication factor than the Random-Hypercube (and thus better performance), but its replication factor also scales considerably better. In addition, the Hybrid-Hypercube has slightly higher replication factor than the Hash-Hypercube. However, this is exactly the reason why it is skew-resilient, and consequently, why it achieves better performance than the Hash-Hypercube scheme.

7.4 DBToaster versus traditional local joins

Next, we compare multi-way joins with traditional local joins and DBToaster local joins.

TPC-H Queries. We run TPCH9-Partial with the $10G/8J$ configuration and TPC-H Q3 with the $10G/8J$ configuration on the TPC-H dataset with the zipfian distribution and the skew factor of 2. In all the TPC-H queries, we disregard LIMIT and ORDER BY clauses, as Squall does not support these constructs yet. The query plans with traditional joins cannot finish due to high computation cost (joiners cannot keep pace even with a minimal number of data sources), so we extrapolate their running time. The performance numbers from Figures 8a and 8b show that DBToaster brings an order of magnitude improvement compared to the traditional local joins.

Google TaskCount Query. We run a query that provides the count of failed tasks per machine id and platform over the publicly available Google cluster monitoring dataset:

Google TaskCount

```
SELECT MACHINE_EVENTS.machineID, MACHINE_EVENTS.platform, COUNT(*)
FROM JOB_EVENTS, TASK_EVENTS, MACHINE_EVENTS
WHERE TASK_EVENTS.eventType = FAIL
AND JOB_EVENTS.jobID = TASK_EVENTS.jobID
AND MACHINE_EVENTS.machineID = TASK_EVENTS.machineID
GROUP BY MACHINE_EVENTS.machineID, MACHINE_EVENTS.platform
```

Hypercube properties. We run the Google TaskCount query using 8 machines. The Hash-Hypercube creates $machineId \times jobId = \{1 \times 8\}$ partitioning, that is, it hashes Job_Events and $Task\_Events$ and replicates the smallest relation ($Machine\_Events$). Whereas, the Random-Hypercube produces $Machine\_Events \times Job\_Events \times Task\_Events = \{1 \times 1 \times 8\}$ partitioning, that is, it replicates the smallest two relations ($Machine\_Events$ and Job_Events). As the query consists of only equi-joins, and there is no significant skew, the Hybrid-Hypercube generates the same partitioning as the Hash-Hypercube scheme (recall that the Hybrid-Hypercube subsumes both the Hash-Hypercube and Random-Hypercube schemes). The difference between the three hypercube schemes is rather small, as the total size of $Machine\_Events$ and Job_Events is only 14.5% of the relation $Task\_Events$ size.

Local joins. On the other hand, using different local joins makes a big difference. As Figure 8c shows, the hypercube schemes with DBToaster outperforms the schemes with traditional local joins by a factor of $3-4\times$.

7.5 Summary

Multi-way joins avoid shuffling the intermediate results, but typically have higher replication of the base relation tuples compared to the corresponding pipeline of 2-way joins. However, for certain queries (such as 3-Reachability), multi-way joins reduce the total communication costs. This translates to achieving better performance compared to the corresponding pipelines of 2-way joins, validating the fact that

[16] Our current implementation of the Hybrid-Hypercube optimization algorithm assumes uniform distribution for the attributes marked as non-skewed. Thus, the computed maximum and average load per machine are the same.

communication cost plays an important role in distributed processing. Multi-way joins are also more amenable for online processing due to their inherent adaptivity to join selectivity fluctuations. Namely, due to their hypercube structure, multi-way joins completely avoid the need for join reordering. Our Hybrid-Hypercube outperforms an existing scheme by up to an order of magnitude (e.g., WebAnalytics in Figure 7). Our scheme achieves up to 1.6× performance improvement compared to the best existing hypercube scheme (TPCH9-Partial on 80G dataset using 100 machines in Figure 7). This is due to the fact that our scheme achieves skew resilience, while reducing replication of the base relation tuples. The maximum and average load per machine are good performance predictors for different hypercube schemes. In general, if some of the relations are relatively small, the performance difference between the hypercube scheme drops. Finally, using DBToaster locally brings an additional speedup of up to an order of magnitude compared to the case when traditional local joins are used.

8. RELATED WORK

Offline multi-way join schemes. The Hash-Hypercube [8] and Random-Hypercube [74] schemes, which we describe in detail in §3.1 and 4, are originally proposed for offline systems. (As we show in §5, we can use these schemes in online systems as well, by periodically adjusting to the statistics collected so far.) Similarly, our Hybrid-Hypercube scheme is also directly applicable for offline processing. The Hybrid-Hypercube advances state-of-the-art, as in contrast to the Hash-Hypercube it supports non-equi joins and it 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 join keys' skew degree and join conditions. We estimate the skew degree information from a sample from each relation.

Chu *et al.* [26] 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 [26], in the future we plan to combine local online cyclic joins with our hypercube schemes. YSmart [41] 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 [19] and SharesSkew [7] are partitioning schemes for multi-way joins that separate relation's tuples into heavy hitters (the join keys with high multiplicity) and light hitters (the remaining join keys). The main idea is to use some variant of hash partitioning for light hitters and random partitioning for heavy hitters. These operators reduce replication as much as possible and they may achieve smaller load per machine compared to the Hybrid-Hypercube in the offline setting. This is due to the fact that Hybrid-Hypercube always decide on partitioning according to the attribute distribution on the relation as a whole, while BinHC [19] and SharesSkew [7] partition each relation into two parts (heavy and light hitters). Thus, an optimal partitioning scheme for multi-way joins should support efficient execution of both equi-joins and non-equi joins (which our Hybrid-Hypercube does), as well as per-key partitioning for equi-joins (as BinHC [19] does). We left the design and implementation of such an operator for future work.

However, both BinHC [19] and SharesSkew [7] are restricted to equi-joins. In addition, these approaches might be suboptimal in an online scenario. In particular, 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 can repeatedly change over time, even right after the scheme adjustment (we denote this pattern as skew fluctuations, and explain it in detail in § 5). This implies frequent data migrations, which affects the performance. In contrast, the Hybrid-Hypercube requires only information about whether the relation's attribute is skew-free or not (this information is used to decide on hash or range partitioning). It does not require information about the exact degree of skew, nor about which keys are highly skewed. The skew degree on the relation as a whole typically changes less frequently than the skew degree among the particular keys, causing smaller number of migration and better performance of online Hybrid-Hypercube 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 [69, 63, 61, 31, 51]. Symmetric hash join [69] requires that data fits in memory. Works [63, 61, 31, 51] address this issue by employing different strategies for spilling to disk. MJoin [65] generalizes XJoin [63] to (local) multi-way joins, and focuses on strategies for spilling to disk. CACQ [48] and STAIRs [29] execute multi-way joins using Eddies architecture [16], that is, they decide on per-tuple basis on an optimal join order. The main difference between DBToaster [9] that we use in Squall and these multi-way joins is as follows. First, these works [65, 48, 29] focus on equi-joins. Whereas, DBToaster also supports complex non-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 [44] and Photon [13] offer online join processing in a distributed setting. Photon [13] is designed for click-stream analytics in Google, and it supports only equi-joins. BiStream [44] is a 2-way stream join operator that partitions each input relation on a separate set of machines. It focuses on scalability and elasticity, and it supports both equi- and non-equi joins. Upon receiving an incoming tuple, BiStream always store it on exactly one machine, and produces the output by sending the tuple to all the machines that (may) contain joinable tuples from the opposite relation. BiStream uses hash partitioning (it sends an input tuple to two machines, one for storing the originating relation, and another for joining with the opposite relation) and random partitioning (an input tuple is randomly assigned to a machine of the originating relation, and sent to all the machines of the opposite relation for join processing). BiStream also proposes ContRand partitioning, which hashes an input tuple to a subgroup of machines. Within a subgroup, ContRand uses random partitioning. As BiStream always store a tuple on exactly one machine, it has smaller memory requirements than the 1-

Bucket scheme [54]. However, when using random partitioning (for non-equi joins or for equi-joins with high skew), BiStream has higher communication cost that the 1-Bucket scheme [54]. We illustrate this on the following example. To simplify the analysis, we compare the two schemes assuming that the relations are of equal sizes. In that case, each relation in the BiStream scheme uses $p/2$ machines. Whereas, the 1-Bucket scheme is a $\sqrt{p} \times \sqrt{p}$ matrix. Thus, BiStream sends each tuple to $p/2$ machines, while the 1-Bucket sends a tuple only to $\sqrt{p}$ machines. In other words, the 1-Bucket scheme implies smaller communication and storage cost than the BiStream scheme.

Distributed online joins: multiple hops. We next describe work that executes multi-way joins using multiple network hops. CTR scheme [35] and PSP scheme [68] optimize tuple routing, providing for adaptive join ordering. PSP [68] partitions the state among the machines according to their timestamp. CTR scheme [35] and PSP scheme [68] support both equi- and non-equi joins, These approaches attempt to address the problem of join selectivity fluctuations by adaptive join ordering. However, CTR and PSP schemes have the following drawbacks. First, these approaches do not materialize intermediate results, and suffer from recomputation. Second, the intermediate results are sent over the network and can be considerably large, causing high communication overhead, and potentially high latency for producing result tuples. In contrast, our HyLD operator solves both problems. It uses local DBToaster operator that allows reusing the previously computed intermediate results, and it requires only one network hop to produce the result tuple.

Distributed Eddies [62, 75] are based on SteMs Eddies [58], and they provide for per-tuple routing and thus, they also provide for adaptive join ordering. However, Distributed Eddies [62, 75] suffer from the same drawbacks as the CTR scheme [35] and PSP scheme [68] (multiple network hops and no intermediate relation reusing). Distributed Eddies assume window semantics, tolerate information loss and do not study intra-operator adaptations (as our Adaptive 1-Bucket scheme [32] does). Furthermore, they do not materialize intermediate results, which leads to recomputation every time a new tuple comes. For small windows, intermediate results might not be frequently reused (when window expires, its intermediate results also expire). However, reusing intermediate results is especially important for moderately-sized to large windows, and for full-history queries, which are nowadays very popular [22, 15]. Thus, we focus on large-state and full-history operators.

Distributed online joins: single hop. Next, we present multi-way join operators that require only one network hop for producing output, similarly to our hypercube schemes. ATR scheme [35] support non-equi joins and it 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. As we already discussed, online operators with large windows or full-history semantics are very popular nowadays [22, 15]. We can extend Squall with ATR partitioning schemes to support small to moderate-sized window operators.

Flux [60] 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, surpasses a machine capacity. As explained in [68], 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 [45]. Liu *et al.* [45, 46] provide multi-way equi-join operators using Flux, inheriting its drawbacks. Liu *et al.* [45, 46] do 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 offer 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 changing a pipeline of 2-way joins to a single-hop multi-way join at run-time. 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 [32] is a non-blocking scheme.

8.1 Existing work on online systems

This section provides a brief overview of the most important (and most widely used) existing online systems. For more details about different online systems, we refer an interested reader to an excellent survey [47].

MapReduce systems cannot provide online processing. The challenge of real-time data processing has recently moved to the forefront of interest among users of analytics and data warehousing systems as well as the large-scale Web applications / NoSQL crowd. On the other hand, map-reduce style batch processing systems [28, 4, 36] are not amenable for low-latency processing due to the following. A MapReduce job consists of a map and a reduce stage. A job does not produce any output before all the input is processed, that is, a reduce function is invoked only after the map function processes all the input data. If the computation consists of multiple MapReduce jobs, only one job is executing at a time, and the next stage blocks until the current one completely delivers its intermediate result[17]. Thus, latencies are very high in these systems, and we need to use different systems to achieve low latencies.

Cohabitation of offline and online systems. Large Web applications companies, which play a key role in the NoSQL movement and the development of map-reduce style batch processing systems [28, 4, 36], use batch processing systems in conjunction with large-scale realtime frontend systems. An architecture that concurrently runs fault-tolerant batch processing and low-latency online processing for the same application is denoted in literature as Lambda architecture [50]. In this architecture, once the exact results from the batch processing are in place, they overwrite the corresponding eventually consistent results from the online processing pipeline. Twitter's Summingbird [20] offers a user the same declarative interface for offline and online processing. The system uses Scalding (Cascading's Scala API) [3] as the backend for offline processing, and Storm [49] as the backend for online processing. Summingbird also allows running the same application in both backends at the same time (hybrid mode). Google DataFlow [11] provides similar functionalities using FlumeJava framework [23] and MapReduce for offline processing, and MillWheel [10] for online processing. Google DataFlow focuses on time series data processing for unbounded streams, allowing a user to choose a tradeoff between latency, correctness and resource costs.

[17]If there is no data dependencies among jobs, they can execute in parallel.

Micro-batch systems. There have been proposals that attempted to introduce *onlineness* in Hadoop, the most famous examples being the Hadoop Online Prototype (HOP) [27] and Scalla [43, 42]. We note that the paper [57] on Nova also claims batched incremental processing of workflows on Hadoop, but provides little detail on the systems aspects of it. HOP pipelines in small batches the map output to reducers, and it performs multi-pass merge on the reducers. However, it was shown in Scalla [43] that HOP is not amenable for high-performance online processing, because sort-merge, inherited from Hadoop, has unacceptable blocking cost. Rather, Scalla [43, 42] uses hash partitioning, which performs better. This system also maximizes performance by carefully partitioning tuples among memory and disk in the case of memory overflow. Both HOP [27] and Scalla [43, 42] focus on general micro-batch MapReduce processing, rather than on database operators. In contrast, Squall focuses on database operators. It uses hash partitioning in the case of skew-free datasets, but we design and implement other partitioning schemes as well (depending on the join conditions and skew degree).

There are attempts to bring online processing to other batch engines. Spark is an in-memory MapReduce system where the computation is specified as transformations over resilient distributed datasets (RDDs). RDD abstraction ensures that, in the case of a machine failure, other machines divide among themselves the work that was assigned to the failed machine. Spark Streaming [73, 72] is based on Spark and it simulates online processing by performing MapReduce-style computation in small batches (micro-batching). As explained in Trill [25], Spark Streaming unfortunately uses the same batch size for physical batching (which helps in achieving high performance) and semantic batching (which is due to specific window semantics). In contrast, these two types of batching are independent in Squall, and query results do not depend on the physical batch sizes. In contrast to Squall, Spark Streaming has no skew-resilient joins[18] nor multi-way joins.

All these systems [27, 43, 42, 73, 72] modify an existing batch system to perform micro-batching. Micro-batch systems achieve better latencies than batch systems. However, micro-batching systems still suffer from high synchronization penalties between machines. This is due to the fact that the system needs to synchronize after each micro-batch, and new incoming tuples are blocked until the whole micro-batch is processed. If a computation contains multiple stages (MapReduce jobs), the synchronization overheads grow as the system synchronizes after each micro-batch on each stage. This is equivalent to a coarse-grained lock-step. Thus, the slowest machine of an operator limits the entire operator execution. The performance degradations occur even in the absence of skew, as one machine may be slower due to non-deterministic reasons (small glitches in network, or small differences in performance among the same hardware). Synchronization raises latencies to the order of seconds and fundamentally hinders scalability.

[18]Spark-skewjoin library (https://github.com/tresata/spark-skewjoin) extends Spark with the support for skew resilience for equi-joins. However, their partitioning scheme is very similar to the F-Skew scheme [21], which efficiently handles only certain types of skew (the key frequencies need to be low in at least one relation).

Ground-up online systems. Next, we describe systems that are designed specifically for online processing. These systems are implemented from scratch, rather than by modifying an existing offline system. Ground-up online systems represent the computation as a DAG of pipelined operators (rather than a series of map and reduce stages), where each operator produces output on a per-tuple basis.

Flink is an Apache project that emerged from a research project called Stratosphere [12]. This system is designed for online processing, that is, the input is unbounded stream, and the input tuples are continuously pipelined through a computation graph. However, Flink can also support offline processing by treating its input in a special way (bounded streams). Flink provides functional interface, where computation is specified through operations over parallel collections. This system offers two join partitioning schemes (repartition and broadcast) and local join operators (hybrid-hash and sort-merge). Flink is equipped with a cost-based optimizer that chooses an optimal scheme and local operator, according to the data and memory sizes. However, Flink currently does not provide skew-resilient nor multi-way joins. On the other hand, Flink has better support for UDF operators (including UDF joins) compared to Squall. In particular, Flink may reorder UDFs (and operators in general) to achieve better performance, while preserving the original program semantics. Furthermore, in contrast to Squall, Flink can run iterative analytics.

MillWheel [10] is another system for online processing. It focuses on efficient fault-tolerance techniques such as replay with duplicate elimination using Bloom filters. This work is orthogonal to Squall, as we could use MillWheel's techniques for achieving fault-tolerance techniques in Squall.

Twitter Storm [49] has a very convenient, dataflow-like, programming abstraction and excellent scalability. It allows users to write arbitrary programs by specifying the computation DAG and the code within each DAG node. Storm offers persistent storage and it supports at-least once, at-most once and exactly-once semantics. To provide exactly-once semantics, Storm uses a persistent storage. Storm's Trident library offers database operators such as aggregations, joins, selections and projections. However, Storm supports only equi-joins on skew-free datasets, as well as multi-way joins with the same join key among all the relations. In contrast, Squall supports complex join operators, including 2-way and multi-way joins, both over skew-free and skewed datasets.

Heron [38] is a next-generation online processing engine developed at Twitter. Heron and Storm are built with the same goal in mind, and Heron is API-compatible with Storm. In fact, Heron is built from scratch with the goal of addressing various performance bottlenecks in Storm. The main performance inefficiency in Storm is the presence of multiple levels of indirection: a worker (JVM process) has multiple executors (threads), and each executor is assigned multiple component tasks [38]. This design causes Storm to spend significant amount of time in multiplexing and demultiplexing each tuple through tasks, executors and workers. That is, each received or sent tuple in Storm goes through multiple queues and threads. In particular, a Storm worker has one thread for receiving tuples and one for sending them further down. Whereas, a Storm executor has a thread for user logic, and a thread for sending tuples to the worker. Thus, each input tuple has to go through 4 threads [38]. In addition, multiple levels of indirection result

in conflicting scheduling goals and thus, in scheduling inefficiencies. By adopting a simpler design and by implementing tuple transferring more efficiently, Heron achieves an order-of-magnitude performance improvements[19] compared to Storm. Heron also achieves better scalability than Storm due to limiting the maximum number of connections for heartbeats (Zookeeper) and for tuple routing (Stream Manager) via hierarchical structuring of communicating nodes.

Trill [25] is a high-performance library for online processing, and Quill [24] is its parallel version. Trill and Quill achieve high throughput mainly due to using column-store optimizations. This line of work is orthogonal to Squall, as we could employ their optimizations in our system.

Overall, existing open-source online systems focus on distribution primitives (e.g., communication patterns, fault tolerance) and low-level performance optimizations. In contrast to existing online systems, Squall focuses on supporting complex joins and on skew resilience.

9. ACKNOWLEDGMENTS

This work was supported by ERC grant 279804.

10. REFERENCES

[1] Common Crawl Corpus. `http://commoncrawl.org/`.
[2] Extracted Hyperlink Graph from August 2012 Common Crawl Corpus. `http://webdatacommons.org/hyperlinkgraph/index.html`.
[3] Scalding: A scala api for cascading. `https://github.com/twitter/scalding`.
[4] The Apache Hadoop project. `http://hadoop.apache.org`.
[5] The TPC-H benchmark. `http://www.tpc.org/tpch/`.
[6] Trove: High Performance Collections for Java. `http://trove.starlight-systems.com/`.
[7] F. Afrati, N. Stasinopoulos, J. D. Ullman, and A. Vassilakopoulos. SharesSkew: An algorithm to handle skew for joins in mapreduce. http://arxiv.org/abs/1512.03921.
[8] F. Afrati and J. Ullman. Optimizing joins in a MapReduce environment. In *EDBT*, 2010.
[9] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic. Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views. In *VLDB*, 2012.
[10] T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. Millwheel: Fault-tolerant stream processing at internet scale. In *Very Large Data Bases*, pages 734–746, 2013.
[11] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8(12):1792–1803, 2015.
[12] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke. The stratosphere platform for big data analytics. *The VLDB Journal*, 23(6):939–964, Dec. 2014.
[13] R. Ananthanarayanan, V. Basker, S. Das, A. Gupta, H. Jiang, T. Qiu, A. Reznichenko, D. Ryabkov, M. Singh, and S. Venkataraman. Photon: Fault-tolerant and scalable joining of continuous data streams. In *SIGMOD*, 2013.
[14] Apache Flink: Scalable batch and stream data processing. https://flink.apache.org/.
[15] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear road: a stream data management benchmark. In *VLDB*, 2004.
[16] R. Avnur and J. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD*, 2000.
[17] S. Banerjee and K. Ramanathan. Collaborative filtering on skewed datasets. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, pages 1135–1136, New York, NY, USA, 2008. ACM.
[18] P. Beame, P. Koutris, and D. Suciu. Skew in parallel query processing. `http://arxiv.org/pdf/1401.1872`.
[19] P. Beame, P. Koutris, and D. Suciu. Skew in parallel query processing. In *PODS*, 2014.
[20] O. Boykin, S. Ritchie, I. O'Connell, and J. Lin. Summingbird: A framework for integrating batch and online mapreduce computations. *Proceedings of the VLDB Endowment*, 7(13):1441–1451, 2014.
[21] N. Bruno, Y. Kwon, and M.-C. Wu. Advanced join strategies for large-scale distributed computation. In *VLDB*, 2014.
[22] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *SIGMOD*, 2013.
[23] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. Henry, R. Bradshaw, and Nathan. FlumeJava: Easy, Efficient Data-Parallel Pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 363–375, 2 Penn Plaza, Suite 701 New York, NY 10121-0701, 2010.
[24] B. Chandramouli, R. C. Fernandez, J. Goldstein, A. Eldawy, and A. Quamar. The Quill Distributed Analytics Library and Platform. Technical Report MSR-TR-2016-25, Microsoft Research, 2016.
[25] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. C. Platt, J. F. Terwilliger, and J. Wernsing. Trill: A high-performance incremental query processor for diverse analytics. *Proceedings of the VLDB Endowment*, 8(4):401–412, 2014.
[26] S. Chu, M. Balazinska, and D. Suciu. From theory to practice: Efficient join query evaluation in a parallel database system. In *SIGMOD*, 2015.
[27] T. Condie, N. Conway, P. Alvaro, J. Hellerstein, J. Gerth, J. Talbot, K. Elmeleegy, and R. Sears. Online aggregation and continuous query support in MapReduce. In *SIGMOD*, 2010.
[28] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI*, 2004.
[29] A. Deshpande and J. M. Hellerstein. Lifting the burden of history from adaptive query processing. In *VLDB*, 2004.
[30] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri. Practical skew handling in parallel joins. In *VLDB*, 1992.
[31] J. Dittrich, B. Seeger, D. Taylor, and P. Widmayer. Progressive merge join: a generic and non-blocking sort-based join algorithm. In *VLDB*, 2002.
[32] M. Elseidy, A. Elguindy, A. Vitorovic, and C. Koch. Scalable and adaptive online joins. In *VLDB*, 2014.
[33] A. Gounaris, E. Tsamoura, and Y. Manolopoulos. Adaptive query processing in distributed settings. *Advanced Query Processing*, 36(1), 2012.

[19] http://www.infoq.com/news/2015/06/twitter-storm-heron

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

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

[36] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, 2007.

[37] J. Kang, J. Naughton, and S. Viglas. Evaluating window joins over unbounded streams. In *ICDE*, 2003.

[38] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter heron: Stream processing at scale. In *SIGMOD*, 2015.

[39] R. Lawrence. Early hash join: a configurable algorithm for the efficient and early production of join results. In *VLDB*, 2005.

[40] R. Lawrence. Using slice join for efficient evaluation of multi-way joins. *Data Knowl. Eng.*, 67(1):118–139, Oct. 2008.

[41] R. Lee, T. Luo, Y. Huai, F. Wang, Y. He, and X. Zhang. Ysmart: Yet another sql-to-mapreduce translator. In *Distributed Computing Systems (ICDCS)*, pages 25–36. IEEE, 2011.

[42] B. Li, Y. Diao, and P. Shenoy. Supporting scalable analytics with latency constraints. *Proceedings of the VLDB Endowment*, 8(11):1166–1177, 2015.

[43] B. Li, E. Mazur, Y. Diao, A. McGregor, and P. J. Shenoy. A platform for scalable one-pass analytics using mapreduce. In *SIGMOD Conference*, pages 985–996, 2011.

[44] Q. Lin, B. C. Ooi, Z. Wang, and C. Yu. Scalable distributed stream join processing. In *SIGMOD*, 2015.

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

[46] B. Liu, Y. Zhu, M. Jbantova, B. Momberger, and E. A. Rundensteiner. A dynamically adaptive distributed system for processing complex continuous queries. In *VLDB*, pages 1338–1341, 2005.

[47] X. Liu, N. Iftikhar, and X. Xie. Survey of real-time processing systems for big data. In *Proceedings of the 18th International Database Engineering & Applications Symposium*, pages 356–361. ACM, 2014.

[48] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *SIGMOD*, 2002.

[49] N. Marz. STORM: Distributed and fault-tolerant realtime computation. https://github.com/nathanmarz/storm.

[50] N. Marz and J. Warren. *Big Data: Principles and best practices of scalable realtime data systems*. Manning Publications Co., 2015.

[51] 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.

[52] M. A. U. Nasir, G. D. F. Morales, D. García-Soriano, N. Kourtellis, and M. Serafini. The power of both choices: Practical load balancing for distributed stream processing engines. In *2015 IEEE 31st International Conference on Data Engineering*, pages 137–148. IEEE, 2015.

[53] M. Nikolic, M. Dashti, and C. Koch. How to Win a Hot Dog Eating Contest: Distributed Incremental View Maintenance with Batch Updates. In *SIGMOD*, 2016.

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

[55] F. Olken. Random sampling from databases, 1993. PhD Thesis, UC Berkeley.

[56] M. A. Olson, K. Bostic, and M. I. Seltzer. Berkeley DB. In *USENIX Annual Technical Conference, FREENIX Track*, pages 183–191, 1999.

[57] C. Olston, G. Chiou, L. Chitnis, F. Liu, Y. Han, M. Larsson, A. Neumann, V. Rao, V. Sankarasubramanian, S. Seth, C. Tian, T. ZiCornell, and X. Wang. Nova: continuous Pig/Hadoop workflows. In *SIGMOD*, 2011.

[58] V. Raman, A. Deshpande, and J. M. Hellerstein. Using state modules for adaptive query processing. In *ICDE*, pages 353–364, 2003.

[59] A. D. Sarma, F. N. Afrati, S. Salihoglu, and J. D. Ullman. Upper and lower bounds on the cost of a map-reduce computation. *Proc. VLDB Endow.*, 6(4):277–288, 2013.

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

[61] 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.

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

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

[64] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *European Conference on Computer Systems*. ACM, 2015.

[65] S. D. Viglas, J. F. Naughton, and J. Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *VLDB*, 2003.

[66] A. Vitorovic, M. Elseidy, and C. Koch. Load balancing and skew resilience for parallel joins. In *ICDE*, 2016.

[67] C. Walton, A. Dale, and R. Jenevein. A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins. In *VLDB*, 1991.

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

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

[70] S. Wu, F. Li, S. Mehrotra, and B. C. Ooi. Query optimization for massively parallel data processing. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 12. ACM, 2011.

[71] Y. Xu, P. Kostamaa, X. Zhou, and L. Chen. Handling data skew in parallel joins in shared-nothing systems. In *SIGMOD*, 2008.

[72] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. ACM.

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

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

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