Squall: Scalable Real-time Analytics

Aleksandar Vitorovic, Mohammed Elseidy, Khayyam Guliyev, Khue Vu Minh,
Daniel Espino, Mohammad Dashti, Yannis Klonatos and Christoph Koch
{firstname}.{lastname}@epfl.ch
École Polytechnique Fédérale de Lausanne

ABSTRACT

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

1. INTRODUCTION

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

Existing open-source online systems (e.g., Twitter's Storm [4], Spark Streaming [10]) focus on distribution primitives (e.g., communication patterns, fault tolerance) and low-level performance optimizations. However, these systems provide only vanilla database operators, such as hash-based equi-joins (and general UDFs), which do not perform well in the case of skew (see §3.1). On the other hand, some join partitioning schemes (e.g., [6]) are skew-resilient, but they are designed for offline processing, and thus, they are unable to adapt to changing data statistics (see §4).

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 [3, 8] and multi-way schemes (Hybrid-Hypercube, see §3.1). Such a system allows us to leverage the effect of various design choices on the performance, and to seamlessly build efficient novel operators (see §3). Squall operators achieve skew-resilience, adaptivity and scalability.

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

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

Proceedings of the VLDB Endowment, Vol. 9, No. 13

2. SYSTEM ARCHITECTURE

Squall is an online distributed query engine which achieves low latency and high throughput. It supports incremental view maintenance and window (stream) semantics. Squall implements typical stream primitives, such as tumbling and sliding windows, by adding the window expiration logic on top of the full-history engine. Squall uses Storm [4] as a distribution and parallelization platform.

The overall system architecture is shown in Figure 1.

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

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

Operators. By combining different partitioning schemes and local join algorithms, Squall offers many join operators. We build novel join operators: adaptive 1-Bucket [3] and Equi-weight-histogram (EWH) join [8]. This paper also presents some novel multi-way joins (a multi-way join runs within a component). Beside joins, Squall offers database operators such as selections, projections and aggregations.

Query optimizer. Squall's optimizer generates a physical plan from the logical plan. The optimizer maximizes throughput and minimizes 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. Further, it assigns the right parallelism to each component, such that a component is neither overloaded nor mostly idle. We refer to this as universal producer-consumer balance. The optimizer uses heuristics to find an optimal join order and component parallelism.

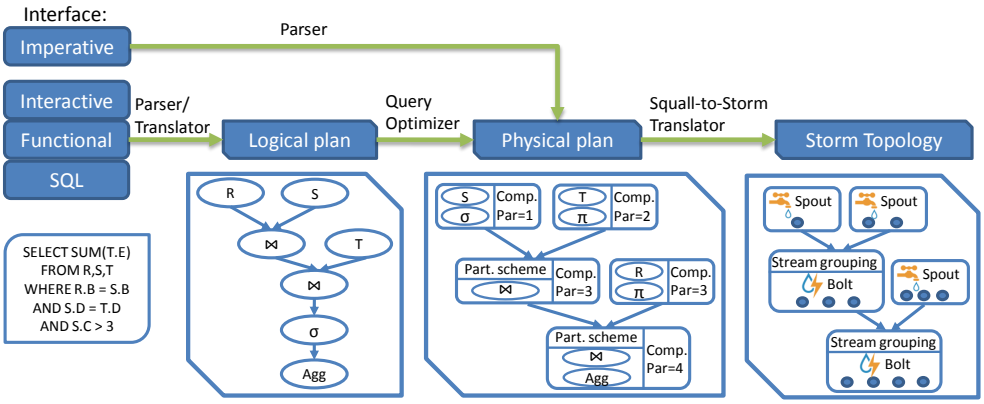


Figure 1: Squall architecture. An example query plan has selections (σ), projections (π), joins ($\bowtie$) and aggregations (Agg).

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 §4). Furthermore, it offers multiple partitioning schemes that achieve different levels of adaptivity for different skew types (e.g., data, temporal and join selectivity skew).

Distribution platform. Squall uses Storm [4] as a distribution platform, but our ideas are more widely applicable. Storm executes a topology, which is a graph of spouts (data sources) and bolts (a bolt consumes streams and produces new ones). An edge in the topology graph is called stream grouping, and it represents partitioning of incoming tuples from a stream among the machines. Squall maps a physical plan to a Storm topology, components to spouts and bolts, and builds partitioning schemes using stream grouping.

3. NOVEL JOIN OPERATORS

We devise new join operators by wiring up state-of-the-art partitioning schemes and local join algorithms. So far, we built 2-way joins [3, 8]. This paper introduces multi-way joins in Squall. These joins can outperform 2-way joins as they avoid shuffling intermediate data [1, 11]. We also devise a novel multi-way join partitioning scheme that further enhances performance. In addition, Squall has efficient local online multi-way joins.

3.1 Partitioning schemes

Next, we describe partitioning schemes for multi-way joins, their skew resilience and supported join conditions. For detailed analysis, please consult our technical report [9].

Hash-Hypercube scheme [1] models the result space as a hypercube, where each axis corresponds to a join key domain. Each machine covers a unique portion of the hypercube space. Figure 2a illustrates this scheme for query $R(x,y) \bowtie S(y,z) \bowtie T(z,t)$. The scheme assigns an input tuple to machines by hashing on the tuple's join keys and replicating on join keys from the other relations. For example, each R tuple is replicated to a "row" of machines with coordinates $(hash(y), *)$. Correctness is preserved as each potential output tuple $t_R(x,y) \bowtie t_S(y,z) \bowtie t_T(z,t)$ is assigned to a single machine with coordinates $(hash(y), hash(z))$. 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. 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.

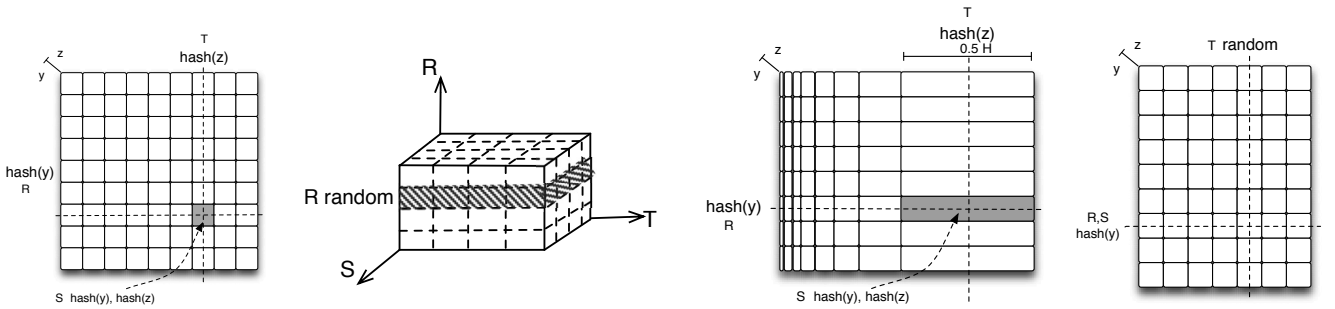
Random-Hypercube scheme [11]. This scheme also models the result space as a hypercube, but each axis corresponds to a relation, as shown in Figure 2b. This scheme randomly distributes the tuples on the axes of the originating relation, and replicates on the other axes. For example, each R tuple is replicated on a "slice" of machines (Figure 2b shows a slice with shading). As the dimensions are $4 \times 4 \times 4$ and a machine receives $1/4$ of each relation, the load per machine is $3 \cdot H/4 = 0.75H$. The Random-Hypercube supports multi-way theta-joins and is skew resilient. However, it replicates tuples more than the Hash-Hypercube (because it uses a 3-dimensional rather than 2-dimensional hypercube).

2-way join schemes. For 2-way joins, Hash-Hypercube becomes hash partitioning, and Random-Hypercube becomes 1-Bucket scheme [6], 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 [6] and our Equi-Weight Histogram (EWH) scheme [8]. The M-Bucket scheme is prone to join output skew. In contrast, the EWH scheme works well for any data distribution. To do so, our EWH scheme provides an efficient parallel scheme for capturing the input and output distribution from the join to a matrix. To evenly partition the work (matrix) among the machines, the EWH scheme employs our join-specialized computational geometry algorithm for rectangle tiling.

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

Hash- and Random-Hypercube are designed only for the cases when either all or none of the relations is skew-free. We propose Hybrid-Hypercube, which uses hash partitioning for skew-free join keys, and more costly random partitioning elsewhere. That way, our scheme achieves skew resilience while minimizing tuple replication. Further, in contrast to the Hash-Hypercube, the Hybrid-Hypercube supports non-equi joins (using random partitioning therein). Thus, our scheme subsumes both the Hash- and Random-Hypercube.

The Hybrid-Hypercube scheme is illustrated in Figure 2d. 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)



(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).

hash join. Whereas, each T tuple randomly picks a "column" of machines to be replicated on. Given that there are no skew on y and no functional dependencies between y and z (which is a common case), $hash(y)$ from R and S simulates random distribution with respect to T. Thus, we can consider $RS \bowtie T$ as a 1-Bucket join.

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× and 1.92× better than that of Random-Hypercube and Hash-Hypercube, respectively.

3.2 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 [3]. Existing local joins use indexes (hash or balanced binary tree) to improve performance. However, these joins are orders of magnitude slower than the state-of-the-art online local join, **DBToaster** [2].

In brief, the main idea of DBToaster is to recursively maintain views for an n-way join. Instead of maintaining only the final result, DBToaster maintains all the intermediate $(n-1)$-, $(n-2)$-, ..., and 2-way joins. When a new tuple comes, DBToaster updates the intermediate relations, and produces the result by joining the tuple with the 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. The savings grow with the increase in the number of relations n.

In contrast to Squall, existing parallel DBToaster [5] do not focus on skew resilience.

3.3 HyLD operator: Hypercube scheme with Local DBToaster

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

Choosing among hypercube schemes. As shown in §3.1, random partitioning is expensive but skew-resilient, while hash partitioning is cheaper but prone to skew. To decide on the hypercube scheme, we need to know if a join key is skew-free or not. A good initial choice of a hypercube scheme saves us from future adaptations. Fortunately, in many cases, even in an online scenario, we know beforehand whether a join key is skew-free. For example, an attribute with the uniqueness property (such as the primary key) cannot have skew. On the other hand, zipfian distributions are typical in many real-life datasets, including Internet packet traces, city sizes and word frequency in natural languages.

4. SKEW TYPES AND ADAPTIVITY

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

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. However, 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 any skew in data distribution.

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

Thus, it is insufficient to capture only the data distribution. Rather, we also need to capture the temporal skew, which we do indirectly by monitoring the machine load[2]. To achieve good performance, Squall uses random partitioning schemes in the case of data or temporal skew.

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

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

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

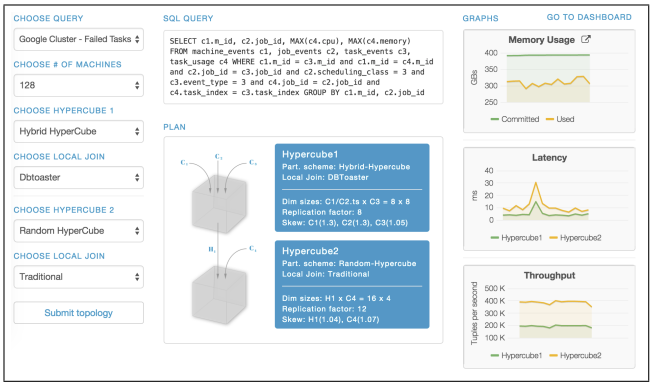


Figure 3: Demonstration: Running a query.

Hypercube sizes. The optimal hypercube dimension sizes minimize replication, and thus, maximize performance. We determine the optimal sizes from the relative base relation sizes. Hence, a hypercube scheme needs to adapt to changing relation sizes. Squall implements an adaptive 1-Bucket join operator [3], which periodically adjusts the offline 1-Bucket partitioning scheme according to the current relation sizes. This operator minimizes state migration, offers a non-blocking migration algorithm, and provides optimality guarantees on data distribution and communication cost.

SAR principle. We introduce the SAR principle, which summarizes this section. To achieve **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 is prone to skew but requires no replication. Whereas, random partitioning is resilient to data and temporal skew and skew fluctuations, but it requires replication. A multi-way join brings adaptivity to join selectivity variations, but it requires higher replication than the corresponding pipeline of 2-way joins.

5. DEMONSTRATION SETUP

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

Google cluster monitoring data[3] contains information about jobs (start and end time, status, etc.), tasks (events, resource usage) and machines (assignments, attributes). We put ourselves in the shoes of a large cluster administrator, who gets notified when a potential problem arises. An interesting multi-way join query is *List the machines which often fail tasks belonging to production jobs*. Another interesting query is *Measure the scheduling algorithm quality*. Schedulers assign jobs to machines to maximize "goodness" score [7], which includes the machine's number of preempted or failed tasks, jobs distribution across the cluster etc. Computing the score involves joining multiple relations. We observe the scheduling algorithm quality by monitoring (in real-time) the score aggregated over jobs and machines.

Demo. As illustrated in Figures 3 and 4, we allow attendees to specify a query and to try different partitioning schemes, local joins and the parallelisms. With a button click, the attendees will run the specified query plan on an in-house cluster with 220 hardware threads. At run-time, they 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. The replication factor is the component's number of input tuples divided by the

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

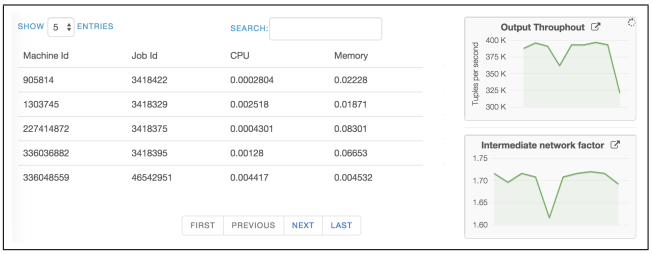


Figure 4: Results and query performance metrics.

total number of tuples produced by the immediate upstream components. We define skew degree as the division between the largest partition size and the average partition size.

Evaluating partitioning schemes. We allow attendees to compare hypercube schemes by monitoring the performance as a function of the operator's replication factor and skew degree. For each hypercube scheme, we identify scenarios (the number of relations, their sizes and skew degrees) where it performs the best. We also evaluate the effect of temporal skew to the performance of hash join and 1-Bucket join. The results validate the SAR principle and suggest that replication is ubiquitous for reliable load balancing.

CPU or network-bound? We aid attendees to find the bottleneck in online processing. To estimate the CPU share, we run the same query plan with different local joins. The attendees can also observe the correlation among the operator's memory consumption and throughput. To estimate the network share, we define intermediate network factor as $(\sum_{comp.\ task\ t} input_t + output_t)/(query\ input + query\ output)$. Then, we compare the performance among different query plans (of the same query) as a function of this factor.

6. ACKNOWLEDGMENTS

This work was supported by ERC grant 279804.

7. REFERENCES

[1] F. Afrati and J. Ullman. Optimizing joins in a MapReduce environment. In *EDBT*, 2010.
[2] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic. Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views. In *VLDB*, 2012.
[3] M. Elseidy, A. Elguindy, A. Vitorovic, and C. Koch. Scalable and adaptive online joins. In *VLDB*, 2014.
[4] N. Marz. STORM: Distributed and fault-tolerant realtime computation. http://storm.apache.org/.
[5] 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.
[6] A. Okcan and M. Riedewald. Processing theta-joins using MapReduce. In *SIGMOD*, 2011.
[7] 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.
[8] A. Vitorovic, M. Elseidy, and C. Koch. Load balancing and skew resilience for parallel joins. In *ICDE*, 2016.
[9] A. Vitorovic et al. Squall: Scalable real-time analytics. Technical Report 217286, EPFL, 2016.
[10] 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.
[11] X. Zhang, L. Chen, and M. Wang. Efficient multi-way theta-join processing using MapReduce. *VLDBJ*, 5(11), 2012.