CLASH: A High-Level Abstraction for Optimized, Multi-Way Stream Joins over Apache Storm

Manuel Dossinger
TU Kaiserslautern
Kaiserslautern, Germany
dossinger@cs.uni-kl.de

Sebastian Michel
TU Kaiserslautern
Kaiserslautern, Germany
michel@cs.uni-kl.de

Constantin Roudsarabi
TU Kaiserslautern
Kaiserslautern, Germany
c_roudsara13@cs.uni-kl.de

ABSTRACT

We propose the demonstration of CLASH, a high-level abstraction on top of Apache Storm. CLASH is designed around *MultiStream*, a novel join operator designed for native support of distributed, multi-way stream joins. MultiStream allows trading off materialization of intermediate results versus communication load. With this demonstration, we invite the audience to explore the full potential of CLASH: multi-way stream joins, creation of complex join plans and their automated optimization, and ultimately the hassle-free SQL-style user/application interface and the translation of the optimized query plans to deployable Storm topologies that are executed on our local compute cluster.

ACM Reference Format:
Manuel Dossinger, Sebastian Michel, and Constantin Roudsarabi. 2019. CLASH: A High-Level Abstraction for Optimized, Multi-Way Stream Joins over Apache Storm. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3299869.3320217

1 INTRODUCTION

Handling massive data streams requires intelligent protocols and algorithms that allow scaling out data as well as computation to multiple nodes within a compute cluster [7, 10]. Application scenarios that require this are ubiquitous, ranging from monitoring enterprise-internal system access logs for ad placement [2] or anomaly detection [5], to providing real-time analytics over social network streams [1]. A core difficulty lies in joining information across multiple streams—for instance joining query and ad-click streams in Google [2], or relating blog posts and Twitter tweets for enriched, user-specific content delivery.

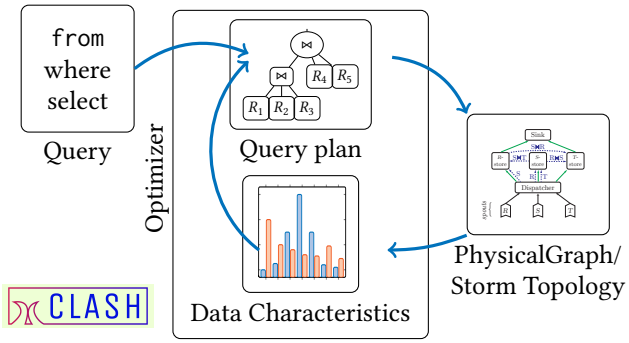


Figure 1: Architectural overview of CLASH.

We present CLASH, a stream processing system that generates optimized Apache Storm topologies for answering join queries with arbitrary predicates—allowing equi and theta, full history and window joins—over multiple relations; Figure 1 gives an overview of the system's architecture. Queries are entered to CLASH via a declarative interface, either text-based using an SQL dialect or via a Java API. The CLASH optimizer obtains a query as well as estimations on data characteristics, e.g., arrival rates and join selectivities, as input and produces a query plan, a logical representation of the materialization steps of intermediate results. The query plan is then transformed into a physical graph (more on this later) which is the basis for translating the query into a deployable Storm topology. During processing, data characteristics are captured in order to periodically re-evaluate the feasibility of the deployed query plan.

2 RELATED WORK

Distributed stream join processing can be accomplished by several different techniques, using a join-matrix model [3] which is then generalized to the join hypercube model [10] for multi-way joins, via the join-biclique model and the BiStream operator by Lin et al. [7], by making tuple-centric routing decisions for distributed eddies [8], or by time-slicing of the join operator's state in multiple stages [11]. For optimizing stream joins, Viglas and Naughton [9] propose the use of rate-based optimization rather than classical cost-based

SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5643-5/19/06…$15.00
https://doi.org/10.1145/3299869.3320217

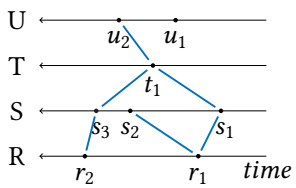


Figure 2: Tuples of multiple input streams arrive.

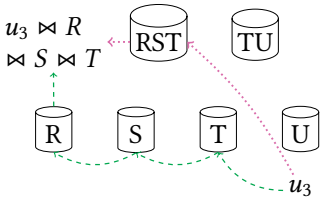


Figure 3: Possible choices of materialized intermediate results.

optimization, due to the continuous nature of data streams. Joglekar and Ré [6] propose using information on the multiplicity of values to optimize multi-way joins. Our approach of performing and optimizing multi-way stream joins is generalizing the BiStream approach [7] by introducing tuple routing schemes across multiple materialized intermediate results or initial relations. These routing paths are optimized separately for each type (i.e., origin) of tuples, while at the same time complex join trees are considered and optimized using dynamic programming.

3 THE ANATOMY OF CLASH

CLASH is designed as a high-level abstraction on top of Apache Storm (or, ultimately, similar infrastructures). The key design rationale behind CLASH emphasizes joins, which are realized through sophisticated tuple placement and routing schemes. Cost models and join-plan optimization ensure effective query evaluation—with support of full-history as well as window-based joins, equality joins and even arbitrary theta joins. CLASH is not reinventing the wheel when it comes to tuple routing primitives, like key-grouping, random assignment, full broadcast, etc., as it is using existing stream processors, like Apache Storm, as routing substrate—benefiting further from provenly robust, mature systems with out-of-the-box properties like fault tolerance. CLASH optimizes and translates user- or application-provided queries into operator topologies that can be deployed and run in the underlying infrastructure.

CLASH allows writing queries in a declarative fashion, like

```
queryBuilder
  .from("R").from("S").from("T")
  .where(...)
  .select(...)
```

Naturally, such a query can be expressed in any contemporary framework—in fact CLASH does exactly this when translating the query into a Storm topology. Manually, however, without high-level routines, it is rather a tedious task to craft such queries. CLASH does not only ameliorate usability, but can also ensure efficiency at runtime, as bad plans can be avoided that could mistakenly be hand crafted by inexperienced users. Current APIs like Flink-SQL or Trident provide convenient means to write queries, however they solely produce linear trees and do not optimize the order of joins operators.

While operators like group-by and in particular also projection and selection are easy to implement, joins are more involved: Consider a four-way join between relations R, S, T, and U, and the tuples of these relations are arriving as shown in Figure 2. Tuples connected by a blue line indicate that they satisfy a partial join predicate. When s_1 is observed, it cannot be joined, as the potential join partners are not yet known to the system. When r_1 is observed, it can be indeed joined with s_1, however no matching partners of T and U are available. Only when t_1 and u_2 arrive, these four tuples can be joined and output as result. This example underpins the two core tasks in distributed stream join processing—on-the-fly partitioning of relations as well as probing tuples against stored tuples of other relations.

3.1 Prefix Placement and Iterative Probing

In CLASH, we store so-called **prefixes** of relations/streams as well as intermediate results. A prefix comprises the tuples of a stream received so far, in case of full-history joins, respectively the most recent tuples according to a window specification (e.g., count- or time-based). Figure 3 shows possible choices for storing inputs and the intermediate results. An arriving tuple of u_3 can either probe the prefixes of T, S, and R sequentially (green dashed arrows), or probe the materialized intermediate result of $R \bowtie S \bowtie T$ (red dotted arrows); in both cases the desired join result is found.

We call the components that contain prefixes **stores**. Each store itself can be distributed among multiple compute nodes, either to enable in-memory processing if the prefix size exceeds available memory, or in order to parallelize computation of the join predicate. Each instance of the store is called **task**. For example, the query optimizer could decide that prefix of stream R should ideally be stored using five tasks, while prefix of stream S is stored using twelve tasks. If more than one task is used to store the prefix of stream, which is the common case when dealing with large streams, a partitioning strategy is responsible for assigning tuples to tasks. In the simplest case, when dealing with theta joins, a random assignment will be used. For equality joins, a hash-based partitioning is preferred. With tuples being placed at individual stores, in order to find matching tuples between two or more streams, tuples of one stream (or intermediate result) need to be sent to stores of other streams (or intermediate results) in order to probe for matching join partners. To illustrate this behavior, Figure 3 shows an example involving four different streams, R, S, T, and U. Let us assume that the join predicate is given as $\theta_{T,U} := T.a = U.b$, then the T-store can be partitioned w.r.t. the value of $T.a$ and tuple u_3 would be sent according to its b-value where the partial join can be

computed. Consider further the predicate $\theta_{S,T} := S.c = T.c$ and a tuple $s_4 \in S$ should be sent to T for probing. This tuple has to be sent to all tasks of the T-store, since in every task there could be potential join partners. on the other hand, the data could be partitioned randomly among the tasks, then both, u_3 and s_4, have to be sent to each T-task, however this way the load is guaranteed to be evenly balanced. We refer the reader to [4] for more details on the tuple routing scheme employed in CLASH.

3.2 Cost-based Join Order Optimization

In order to derive a deployable Storm topology based on a query, there are several opportunities that need to be jointly optimized: First, regarding the choice of (intermediate) prefixes to be stored and how many tasks should be used per store. Second, optimizing how data should be partitioned among the these tasks. Third, finding an efficient probe order according to which tuples are sent across stores in order to test them against stored tuples.

It is clear that for the base (input) relation, there should be one store each, commonly involving several tasks. Materializing intermediate results, on the other hand, might or might not be feasible, depending on data, system, and network characteristics. This means, there is the choice between introducing materialization points implemented by issuing the creation of stores comprising intermediate result tuples or by avoiding materialization and instead computing a multi-way join by intensively sending probe tuples across the network.

In order to highlight the case of intense materialization, consider the left-deep join tree shown Figure 4a. Here, the intermediate result of $R_1 \bowtie R_2 =: R_{12}$ is materialized, the result of $R_{12} \bowtie R_3$ is computed and materialized, and so on. In contrast to the join ordering on relational database systems, where allocated memory can be freed after an operation is completed, we continuously read fresh input. Thus, the intermediate results are never finished and the tuples have to be kept for a long period of time, in case of full-history joins or large windows. If the topmost priority is saving the space required for storing prefixes, then a trivial solution is to only materialize the inputs, as shown in the flat tree in Figure 4b. There, $R_1, \ldots, R_5$ are joined using a single MultiStream operator. If it is affordable to allow a certain amount of storage, a plan like in Figure 4c can be deployed. Consider the case of a join where each pair of relations R_1, R_2, and R_3 produces a big result, but the result of the join $R_1 \bowtie R_2 \bowtie R_3$ is just roughly of the same size as the input relations. Then, it is preferable to materialize only the entire result instead of the intermediate joins. In Figure 4c the role of materialization as boundary for communication can be seen: When tuples of R_4 arrive, they only have to be probed against the prefix of R_5 and the prefix of the materialized result of $R_1 \bowtie R_2 \bowtie R_3$,

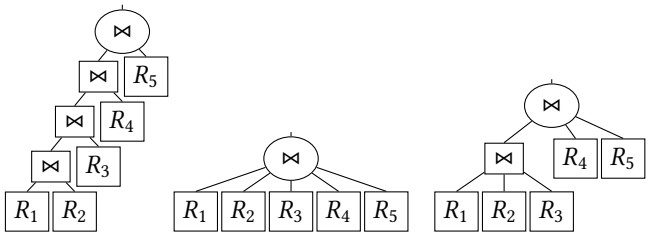


Figure 4: Different possibilities of constructing query plans for five input relations.

but not against R_1, R_2, or R_3 individually. Each of the latter relations is still probed against all other relations.

3.3 Translation to Deployable Topologies

Once the query plan is generated by the optimizer, CLASH can translate it to a Storm topology in a fully automatic fashion. The sole task left to the user is to provide a few lines of code to register spouts (Storm's ingestion method for data) for the inputs and a sink for writing join results to. CLASH itself creates the necessary bolts (i.e., Storm operators) which implement the stores, connects them using the specified grouping methods, and assigns the predicate evaluation to the correct positions. Using a dynamic ruleset, each store bolt knows how to react to each arriving tuple, i.e., it is checked whether the arriving tuple should be included in the local prefix, or if it should be probed against the prefix, and if so, which predicates should be applied. If the predicate is an equality predicate and the prefix is stored in an according way, a hash join is executed, otherwise a nested-loop join is used. The ruleset further specifies, where the results—if any exist—should be sent to and if the data partitioning of the receiving store can be exploited or if the results have to be broadcast to each task.

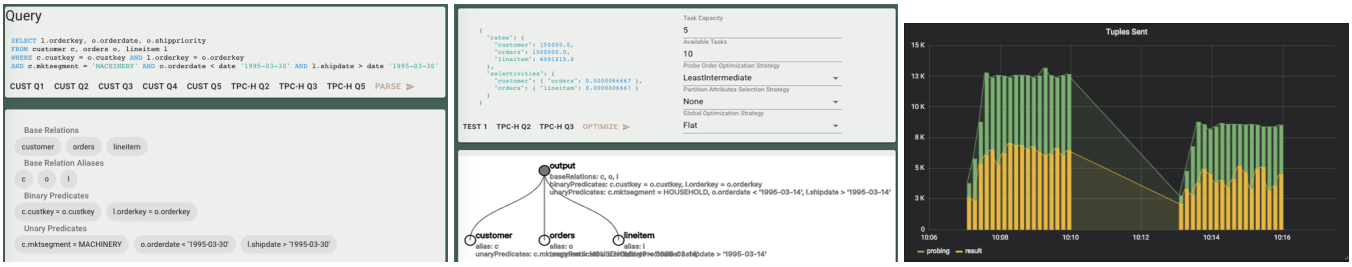
4 THE DEMONSTRATION

We demonstrate CLASH using a web interface for submitting queries and changing parameters for the optimization procedure, and visualizing the performance of the system that runs queries on our cluster using TPC-H or a custom generated data set.[1] A 5-minute video introducing the indented demonstration is available under https://youtu.be/oZxNIwvEQDw.

4.1 Writing Queries

Attendees of our demo can write queries inside an IDE like IntelliJ using the declarative Java API (cf., Figure 6) or alternatively use the web interface where SQL can be used for query formulation. The latter is shown in Figure 5a, where users are shown the internal representation of a CLASH query

[1]The source code of CLASH and the demo will be shared on Github upon acceptance of this demonstration.



(a) (b) (c)

Figure 5: Query input and interpretation (a), optimizer interface and generated query plan (b), and performance observations of two deployed topologies (c).

```
queryBuilder
  .from("T1").from("F").from("T2")
  .where("T1.url = F.original_url"))
  .where("F.url = T2.original_url")))
  .where("T1.user_id = T2.user_id"))
  .select("T1.url", "T1.user", "F.user", "T2.user")
```

Figure 6: Example of using the SQL-style CLASH API.

which serves as input to the optimization procedure. Both cases are easy and concise ways to specify what should be computed without specifying how it should be done.

4.2 Optimization

Figure 5b shows our interface to the optimizer where users can enter assumptions on the arrival rate and join selectivities of inputs or select predefined assumptions, too. Further, one can specify the system configuration the optimizer should use for producing a query plan. This configuration comprises of the number of available tasks and the amount of tuples that can be stored by an individual task. Finally, a strategy for the optimization procedure is selected and the resulting query plan and estimated cost is displayed.

For example, a user might enter data characteristics that require only stores with parallelism 1, and then they increase the arrival rate such that the stores need to be scaled out. At the same time, the communication cost increases. In order to lower the communication cost again, the task capacity might be scaled up. Such observations are relevant, e.g., when virtual machines in a cloud environment have to be commissioned: two EC2 instances have the same cost per hour as one with double amount of memory. Multiple smaller machines can be switched on and off (and therefore payed or not) more fine grained according to the current load; few bigger machines on the other hand save communication and thus bandwidth.

In another scenario, a user would like to know whether an upgrade of machines can enable a more efficient topology to be deployed. To do so, the query optimizer is asked to generate plans for two different numbers of available tasks and compare the expected performance figures.

4.3 Topology Execution

After optimization, a query can be sent to our cluster for execution. The generated Storm topology does not only compute the join but also outputs monitoring information which we visualize using the Grafana tool, as shown in Figure 5c. By submitting differently optimized topologies after each other, the displayed performance metrics change and the audience can understand how the parameters for optimization influence the runtime behavior of the system. Further, the effect of poorly estimated data characteristics on different topologies can be explored, by feeding the query optimizer with wrong or insufficient statistics on purpose. Then some topologies have to deal with higher communication load while other topologies might even be infeasible as huge, unanticipated intermediate results cannot be materialized.

REFERENCES

[1] Foteini Alvanaki and Sebastian Michel. Tracking set correlations at large scale. *SIGMOD, 2014.*

[2] Rajagopal Ananthanarayanan et al. Photon: fault-tolerant and scalable joining of continuous data streams. *SIGMOD, 2013.*

[3] Mohammed Elseidy, Abdallah Elguindy, Aleksandar Vitorovic, and Christoph Koch. Scalable and Adaptive Online Joins. *PVLDB*, 2014.

[4] Manuel Hoffmann and Sebastian Michel. Scaling Out Continuous Multi-Way Theta-Joins. *BeyondMR@SIGMOD*, 2017.

[5] Dimitrije Jankov et al. Real-time High Performance Anomaly Detection over Data Streams: Grand Challenge. *DEBS, 2017.*

[6] Manas Joglekar and Christopher Ré. It's All a Matter of Degree: Using Degree Information to Optimize Multiway Joins. *ICDT, 2016.*

[7] Qian Lin, Beng Chin Ooi, Zhengkui Wang, and Cui Yu. Scalable Distributed Stream Join Processing. *SIGMOD, 2015.*

[8] Feng Tian and David J. DeWitt. 2003. Tuple Routing Strategies for Distributed Eddies. *VLDB, 2003.*

[9] Stratis Viglas and Jeffrey F. Naughton. Rate-based query optimization for streaming information sources. *SIGMOD, 2002.*

[10] Aleksandar Vitorovic et al. Squall: Scalable Real-time Analytics. *PVLDB, 2016.*

[11] Song Wang and Elke A. Rundensteiner. Scalable stream join processing with expensive predicates: workload distribution and adaptation by time-slicing. *EDBT 2009.*