Distributed CQL Made Easy

Robert Soulé[1] Martin Hirzel[2] Robert Grimm[1] Buğra Gedik[2]

[1]*New York University* [2]*IBM Research*

This talk is about making it easy to implement a distributed CQL. CQL is a continuous query language that extends SQL with a notion of windows over infinite streams of data. Programmers like using CQL, because its syntax is already familiar to them from their experience with other database-backed applications. CQL was originally designed to target a single node implementation of the Stanford Stream Data Management System. However, as streaming applications are expected to process larger and larger amounts of data at faster and faster rates, CQL will need to scale to run on clusters of machines.

As database implementers can attest, developing a distributed streaming system from scratch would be a daunting endeavor. This talk argues that they key to simplifying not just the development of distributed CQL, but streaming applications in general, is to treat this as a *language* problem, not a database problem. What we need is an intermediate language (IL) that maps existing streaming languages, like CQL, to existing distributed streaming runtimes. If that intermediate language is designed correctly, and the translation from the IL to the existing runtime is implemented, then the task of developing a distributed CQL becomes simply a matter of providing a straightforward translation from CQL to the IL. Figure 1 illustrates this approach.

A well designed intermediate language does not only simplify the task of mapping existing streaming languages to existing runtimes, but it can also help simplify the task of implementing optimizations. Critical optimizations, such as the data-parallelism that has made MapReduce [8] so popular, can be implemented directly at the level of IL. This means that optimizations need only be implemented once, and can then be shared by different streaming languages.

Our key insight is that the IL must make things explicit that require special machinery in distributed systems to facilitate reasoning about that machinery. First, communication channels must be explicit and one-to-one, i.e., connect the output of exactly one operator with the input of exactly one operator. After all, any form of many-to-many communication in a distributed system requires explicit machinery, such as application-level multicast. Next, all uses of state must be made explicit. After all, keeping state consistent across nodes in a distributed system requires explicit machinery, such as two-phase commit. Finally, the IL cannot assume globally deterministic execution. After all, ordered execution in a distributed system requires, again, additional machinery, such as a sequencer.

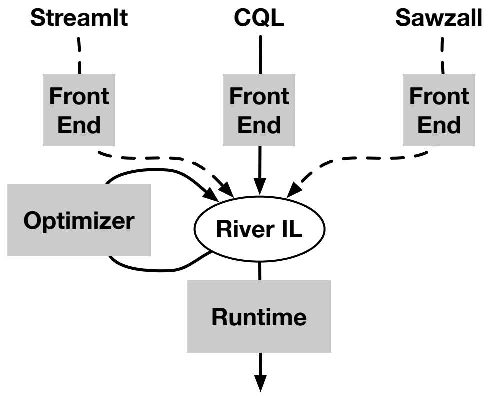
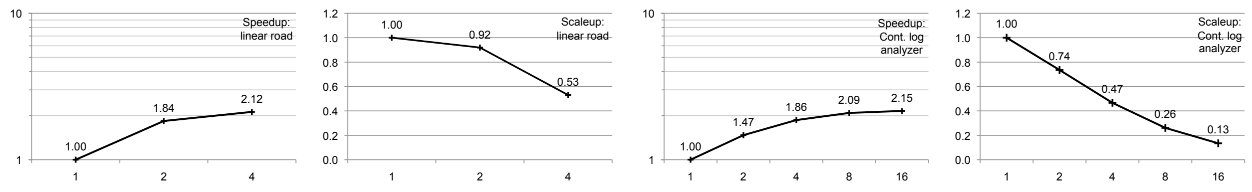


Figure 1: Distributed CQL with River. As the figure indicates, the same approach can be used for other streaming languages, such as StreamIt and Sawzall.

Our intermediate stream processing language, River, is based on our prior work on Brooklet [11], a universal calculus for stream processing languages that makes operators, communication, and state explicit. While both Brooklet and traditional query plan representations capture these *structural* features, River also has *representational* features for data types and expressions. Thus, we can use River to implement the operators themselves. In fact, our implementation of CQL and River share a common expression language, which greatly reduces the overall implementation and translation effort. Moreover, the original description of CQL [7] did not specify a data definition language for declaring data schemas, so we designed our own. Again, to reduce implementation effort, our CQL and River share a common syntax for data types. We feel that our addition of a strong type system results in a more complete language implementation than the original system.

Our implementation of the CQL operators differs from the original description [7] in two respects. First, because the original implementation targeted a single node, operators could store pointers to tuples to avoid unnecessarily copying data. This is obviously not pos-



(a) Linear road speed-up (b) Linear road scale-up (c) Log analyzer speed-up (d) Log analyzer scale-up

Figure 2: Speedup (throughput relative to single processor) and scaleup (speedup divided by number of processors).

sible for a distributed deployment. However, because River makes topology and uses of state explicit, it is easy to determine when stateful operators are co-located on the same host. In on-going work, we are exploring how to apply the same shared state optimization for co-located, stateful operators.

Second, to ensure deterministic execution on a distributed system, the River representation of CQL depends on a sequencer to mark all input data. Unlike the published CQL semantics [5], all operators are incrementalized. This means that each operator consumes a single data item at a time, which contains bags of insertions and deletions. This conforms to the actual implementation of the Stanford STREAM system. The Stanford STREAM system required a central scheduler, which restricts interleavings and would be prohibitively expensive in a distributed system. Therefore, our implementation does all scheduling locally. Unfortunately, this causes non-unary operators to become bottlenecks. Operators execute when they receive input data from any port, and they make no assumptions about which port receives data first. Divergent operators (e.g., splitters) must send something on every path for every timestamp, and convergent operators (e.g., joins, unions) must block until they have received data from every input port.

Our prototype implementation provides a translation of CQL to River, and a translation from River to IBM's streaming middleware, System S [4]. We have used our prototype to run the Linear Road benchmark [6] on 1, 2, and 4 machines, and the results are shown in Figures 2 (a) and (b). Despite the fact that the Linear-Road application shows only limited amounts of task and pipeline parallelism, the first distributed CQL implementation achieves a 2.12x speedup by distributing execution on 4 machines. To test the benefits of data-parallelism, we replicated operators from a web log analyzer query with increasing amounts of parallelism on 1 to 16 machines. The results are shown in Figures 2 (c) and (d). Replication was only able to provide a 2.2x speedup, due to the previously mention synchronization bottleneck.

Of course, our approach can generalize to other streaming languages. In this talk, we concentrate on CQL, rather than other SQL-like languages for stream processing [2, 3, 1, 9], because it has a formally defined semantics [5]. River can also support other styles of streaming languages, and we provide additional translations for the StreamIt [12] and Sawzall [10] languages. This not only allows System S [4] to run programs written in CQL, StreamIt, and Sawzall, but it also allows all three languages to share a common data-parallelism optimization.

In short, this work not only significantly reduces the development effort for implementing a distributed CQL, but also provides a lingua franca for mapping streaming languages in general to distributed streaming runtimes, and a common substrate for implementing optimizations.

References

[1] BEA WebLogic Event Server EPL Reference, July 2007.

[2] D. J. Abadi et al. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, Aug. 2003.

[3] D. J. Abadi et al. The design of the Borealis stream processing engine. In *Proc. 2005 Conference on Innovative Data Systems Research*, pages 277–289, Jan. 2005.

[4] L. Amini et al. SPC: A distributed, scalable platform for data mining. In *Proc. 4th International Workshop on Data Mining Standards, Services, and Platforms*, pages 27–37, Aug. 2006.

[5] A. Arasu et al. A denotational semantics for continuous queries over streams and relations. *ACM SIGMOD Record*, 33(3):6–11, Sept. 2004.

[6] A. Arasu et al. Linear road: A stream data management benchmark. In *Proc. 30th International Conference on Very Large Data Bases*, pages 480–491, Aug./Sep. 2004.

[7] A. Arasu et al. The CQL continuous query language: Semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, June 2006.

[8] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. 6th OSDI*, pages 137–150, Dec. 2004.

[9] N. Jain et al. Towards a streaming sql standard. *Proc. 2008 VLDB Endowment*, 1:1379–1390, August 2008.

[10] R. Pike et al. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming Journal*, 13(4):277–298, 2005.

[11] R. Soulé et al. A universal calculus for stream processing languages. In *Proc. 19th ESOP*, volume 6012 of *LNCS*, pages 507–528, Mar. 2010.

[12] W. Thies et al. StreamIt: A language for streaming applications. In *Proc. 11th International Conference on Compiler Construction*, volume 2304 of *LNCS*, pages 179–196, Apr. 2002.