Technical Perspective: A Fresh Look at Stream Computation through DSP Glasses

Dan Olteanu
Department of Informatics, University of Zurich
dan.olteanu@uzh.ch

DBSP (Data Base Stream Processing) is a *simple yet expressive language for stream computation* that draws inspiration from DSP (Digital Signal Processing). In DBSP, stream computation is expressed using circuits of stream operators whose input and output are (possibly infinite) sequences of database updates.

Queries in languages such as SQL and even Datalog with recursion can be compiled into DBSP circuits in a modular way, where logical relational algebra operators such as projection, selection, join, union, and difference are compiled into stream operators that are composed into a circuit implementing the logic of the query. Special stream operators are also used to delay streams, to gather the entire stream into a database, and to generate the sequence of changes between any two subsequent stream elements.

DBSP's stream circuits are a powerful artifact. They are an *intermediate representation* between the high-level declarative queries and the low-level highly efficient code. They can be optimized by merging operators and by transforming them using rewrite rules. A notable optimization is for the recursive Datalog program that computes reachability in a graph. The circuit obtained for this program implements the classical naïve computation of reachability: Each iteration adds the pairs of start and end nodes of increasingly longer paths. Its optimization naturally recovers the classical semi-naïve evaluation: Now each iteration only adds those pairs of start and end nodes of paths that do not have shorter paths between them, so these pairs were not already discovered in earlier iterations.

Most importantly for stream computation, the circuits can be *automatically incrementalized*: For DBSP, there is a simple and elegant procedure that turns any circuit that computes a query over a database into a circuit that incrementally maintains the query under the stream of changes to the database. At its core, this procedure uses the property that a composite query can be incrementalized by incrementalizing each of its subqueries independently. A prime example is the automatic incrementalization of the equality join operator. The DBSP framework recovers the well-known delta rule for a join: Given the join of two relations and changes to both relations, the change to the join result is the union of (i) the join of the changes; (ii) the join of the first relation and of the change to the second relation; and (iii) the join of the second relation and of the change to the first relation. Query incrementalization, which lies at the foundation of *incremental view maintenance*, is proved formally in the context of DBSP using a theorem prover.

A further key ingredient of the DBSP framework is the use of $\mathbb{Z}$-sets to keep track in a *uniform way* of the nature and amount of changes: Changes consist of database tuples annotated with multiplicities. Inserts are expressed using tuples with positive multiplicities, whereas deletes are expressed using tuples with negative multiplicities. Allowing negative multiplicities in the database offers great flexibility when dealing with out-of-order updates. To see this, consider an insert and a delete of the same tuple. Regardless of the order of their arrival, the DBSP framework concludes that the two updates cancel each other. Virtually all commercial database systems, which provide support for incremental view maintenance, follow a non-confluent update semantics as they obtain different outcomes depending on the order of the two updates: the tuple is (not) in the database if the delete (insert) comes first. The latter update semantics can also be recovered by DBSP using a *distinct* operator in DBSP circuits. Using tuple multiplicities for incremental view maintenance goes back to the counting algorithm by Gupta et al from 1993. These days, $\mathbb{Z}$-sets are used critically in research prototypes such as DBToaster, F-IVM, and Crown and in the RelationalAI commercial engine. It is surprising that despite the clean update semantics and rather small implementation overhead of $\mathbb{Z}$-sets, so far their use remains limited.

Overall, DBSP is a fresh look at the long-standing problem of maintaining the result of relational queries under updates to the input database. Thanks to its unifying treatment of both non-recursive and recursive queries, DBSP follows in the footsteps of Differential Dataflow, a much celebrated framework that caters for general stream computation. The intermediate representation remains however the key feature of DBSP: High-level query languages can be compiled into circuits, which are more fine-grained than queries and allow for lower-level optimization and generation of efficiently executable code. It would be exciting to investigate how recent maintenance strategies, which achieve (amortized) constant time per single-tuple update for several classes of non-recursive queries, can be adapted to the DBSP framework. This would require extensions with stream operators that maintain state in the form of auxiliary data structures, with worst-case optimal join algorithms, and with factorized join computation and maintenance.