Flow-centric Data Pipelines

Victoria Dib, Andrew J. Mikalsen, Jaroslaw Zola, Oliver Kennedy
{vdib,ajmikals,jzola,okennedy}@buffalo.edu
University at Buffalo, SUNY

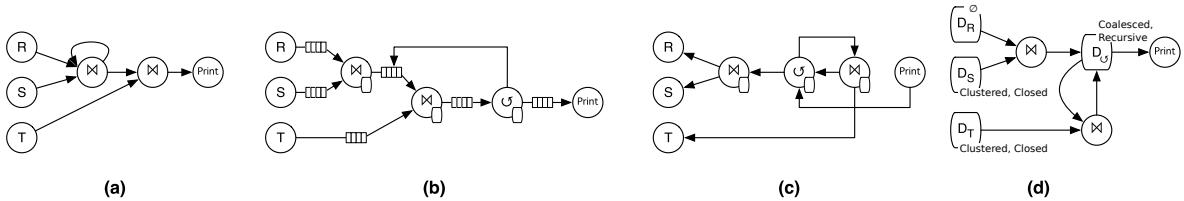

Figure 1: A simple dataflow (a), with three implementations: (b) Push, (c) Pull, (d) Dataflow

State- vs Flow-centric Pipelines. Classical database query evaluation focus on operators, isolated tasks that perform both compute and IO. Operators are chained together through an abstraction of data streams, typically framed as push- or pull-based. Figure 1.a shows the abstract dataflow for a simple recursive query of the form $Q = T \bowtie (Q \cup (R \bowtie S))$. In the push model (b), each operator input has a buffer that its source operators write into. In the pull model (c), each operator actively reaches out into its sources to retrieve tuples. In both approaches, the join and recursion operator must maintain state (rounded rectangles): Indexed copies of one or both input relations for joining or deduplication.

Having operators with internal state creates implementation challenges. First, operators often end up maintaining redundant state. Sharing state between operators is possible [3], but requires operators to be tightly coupled to the schedule, significantly increasing the complexity of both the scheduler and operator implementations. Second, the push- and pull-based models each admit a non-overlapping set of optimizations. For example, the push-based model has finer-grained control over tuple creation and can limit paging [2], while the pull-based model can schedule the execution order of its inputs to avoid materializing both inputs in a join. Finally, advances in data structures must be manually integrated into each operator, individually. For example, Apache Sedona, a geospatial extension for Apache Spark, must manually implement its index structures into the join and filtering operators.

Consider the primitive dataflow graph of the query we are discussing, illustrated in Figure 1 (c). Each edge in this graph represents a single data flow, from a source to a sink. Our main insight is that operator-materialized state is (almost) exactly the data passing over one (or more) of these flows. This insight opens the door to a query evaluation model that decouples state materialization from computation. A flow-centric data pipeline alternates between two types of operators: *stateless* Compute operators pull input tuples from Data operators, and push their output tuples to another Data operator. Our example query is illustrated in Figure 1 (d), with rounded rectangles showing data operators.

Flow-centric Pipelines. Our approach builds on standard query optimization techniques; In the interest of space, we focus primarily on the dataflow aspects of our approach. The first challenge to implementing a flow-centric pipeline is decoupling Compute operator implementations from the abstract data structures backing each Data operator. Unfortunately, the requirements of each operator are distinct, and often non-overlapping; We need a way to ensure that the data structure can (efficiently) provide the capabilities needed by the operator. As a result, we label each input of a Compute operator input with a set of properties, several of which are illustrated in Figure 2. Properties apply constraints to cursors through which the operator accesses its data: over the data itself (order, aggregation), or on access patterns (`seek_to_*`).

Property	Data	Cursor
Coalesced	Pre-aggregated	-
Clustered	Clustered by key	`seek_to_key`
Sorted	Sorted by key	`seek_to_key`
Closed	-	`seek_to_head`

Figure 2: Examples of flow properties

Data operators are responsible for instantiating cursors that satisfy the listed constraints (even if inefficiently). The compiler's role is twofold: (i) Merging flows into combined Data operators (e.g., for two operators that read from the same set of sources), and (ii) Instantiating Data operators that can efficiently implement the required cursor properties. For example, morsels [2] can be used for any operator that requires none of the listed properties, while an on-disk hash table can support the Clustered property.

Implications. The flow model changes several operators. For example most flavors of join become index-nested loops, as the build phase is now a data operator. Aggregation can be completely rewritten as a data operator, and may even be deferred using algebraic tricks [1]. These changes allow re-use of existing organization (e.g., tables already clustered) We are developing an on-disk datalog engine[1] that uses the flow-centric style to give runtime scheduler visibility into memory usage.

Thanks. The authors wish to thank Arlen Cox for his invaluable contributions to this project.

References

[1] Christoph Koch, Yanif Ahmad, Oliver Kennedy, Milos Nikolic, Andres Nötzli, Daniel Lupei, and Amir Shaikhha. 2014. DBToaster: higher-order delta processing for dynamic, frequently fresh views. *VLDB J.* 23, 2 (2014), 253–278.
[2] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD Conference*. ACM, 743–754.
[3] Frank McSherry, Andrea Lattuada, Malte Schwarzkopf, and Timothy Roscoe. 2020. Shared Arrangements: practical inter-query sharing for streaming dataflows. *Proc. VLDB Endow.* 13, 10 (2020), 1793–1806.

[1] https://git.odin.cse.buffalo.edu/Norn/Draupnir