Flow-centric Query Evaluation Pipelines [Talk]

Victoria Dib Andrew J. Mikalsen Krishna Sivakumar Jaroslaw Zola Oliver Kennedy

University at Buffalo
{ vdib, ajmikals, sivakum3, jzola, okennedy }@buffalo.edu

Compute-centric Pipelines

Most database engines evaluate queries by compiling them down to an execution pipeline of composable operators that each encapsulate part of the query's compute and/or IO. Operators are chained together through an abstraction of relational 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. However, both models fundamentally rely on a streaming abstraction in which operators are necessarily stateful. For example, an in-memory hash join operator will fully materialize one of its input relations (the 'build' relation) in the pull model, and both of its input relations in the push model.

Preserving the abstraction of isolated operators requires forcing each operator to manage state internally. Unfortunately, in a typical out-of-core database system, state management consumes limited shared resources like memory and IO. Globally managing how such resources are shared requires poking holes through the simple stream-based operator abstraction. In this talk, we will outline our work-in-progress "dataflow-oriented" pipeline abstraction. Closely related to the push based model, dataflow pipelines enable a range of global optimization opportunities for query evaluation pipelines.

Reader/Writer Co-Optimization A 2-pass hash operator immediately partitions tuples it receives to disk. However, in the push- based model, a source operator must first encode the tuple into a standard format (that may differ from its on-disk representation), and must enqueue the tuple for subsequent processing (limiting opportunities to leverage temporal locality). Eliminating this extra work by inlining the write step into the source operator, requires breaking the clasiscal operator abstraction.

Reader/Reader Co-Optimization An in-memory hash operator constructs a hash table, while an aggregation operator might uses a hash table to store aggregate values for each group. Even when both operators read from the same source, the abstraction forces them to construct independent data structures. Under some circumstances sharing is possible [McSherry et al.(2020)], but requires engineering the operators for coordination. In the case of a hash join and an aggregation operator, sharing presents a further challenge: The ideal data structure for the joint needs of both operators (e.g., a range tree) is suboptimal for each operator individually.

IO Scheduling Optimization Two operators that read the same tuple can benefit from temporal locality at the cache level if co-scheduled [Leis et al.(2014)], but the scheduler lacks direct access to this information without breaking the operator abstraction.

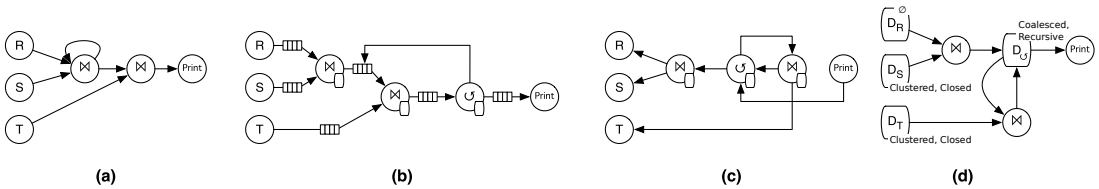

Figure 1: A simple dataflow (a), with three implementations: (b) Push, (c) Pull, (d) Dataflow

Dataflow-centric Pipelines

In summary, a wide range of optimization opportunities require punching holes in the the classical stream-oriented operator abstraction. These optimizations are possible[McSherry et al.(2020), Leis et al.(2014)], but by violating the core premise of the push- or pull-based operator abstraction, introduce significant complexity, and limit opportunities for composability. An abstraction that decouples relational state from compute-centric operators would simultaneously provide greater opportunities for inlining, cross-operator state management, and co-scheduling IO and compute as distinct resources.

For this abstraction to be viable, (i) operators need to be aware of how tuples they produce will be consumed, and (ii) the scheduler needs to manage different blocking patterns. 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.

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

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_*`).

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 efficiently implement the required properties. For example, morsels [Leis et al.(2014)] can be used for any operator that requires only baseline streaming access to data, while an on-disk hash table supports 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 [Koch et al.(2014)]. 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

[Koch et al.(2014)] 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.

[Leis et al.(2014)] 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.

[McSherry et al.(2020)] 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`