The Essence of Online Data Processing

PHILIP DEXTER, YU DAVID LIU, and KENNETH CHIU, State University of New York (SUNY) at Binghamton, USA

Data processing systems are a fundamental component of the modern computing stack. These systems are routinely deployed *online*: they continuously receive the requests of data processing operations, and continuously return the results to end users or client applications. Online data processing systems have unique features beyond conventional data processing, and the optimizations designed for them are complex, especially when data themselves are *structured* and *dynamic*. This paper describes DON Calculus, the first rigorous foundation for online data processing. It captures the essential behavior of both the backend data processing engine and the frontend application, with the focus on two design dimensions essential yet unique to online data processing systems: *incremental operation processing* (IOP) and *temporal locality optimization* (TLO). A novel design insight is that the operations continuously applied to the data can be defined as an *operation stream* flowing through the data structure, and this abstraction unifies diverse designs of IOP and TLO in one calculus. DON Calculus is endowed with a mechanized metatheory centering around a key observable equivalence property: despite the significant non-deterministic executions introduced by IOP and TLO, the observable result of DON Calculus data processing is identical to that of conventional data processing without IOP and TLO. Broadly, DON Calculus is a novel instance in the active pursuit of providing rigorous guarantees to the software system stack. The specification and mechanization of DON Calculus provide a sound base for the designers of future data processing systems to build upon, helping them embrace rigorous semantic engineering without the need of developing from scratch.

CCS Concepts: • **Information systems** → **Database design and models**; • **Theory of computation** → **Operational semantics**.

Additional Key Words and Phrases: Formal Reasoning, Online Data Processing, Incremental Evaluation, Online Data Optimization

ACM Reference Format:
Philip Dexter, Yu David Liu, and Kenneth Chiu . 2022. The Essence of Online Data Processing. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 157 (October 2022), 30 pages. https://doi.org/10.1145/3563320

1 INTRODUCTION

Providing high assurance to each layer of the computing stack is of critical importance in trustworthy computing (e.g., [Appel et al. 2016]). The bedrock of many data-intensive applications — from social networks, to bioinformatics, to artificial intelligence — is the data processing system, such as databases and data analytical engines. Accelerated by the wide adoption of cloud computing, these applications and systems are routinely deployed *online*: a long-running program continuously applies a large number of data processing operations to a large amount of data, and continuously provides its clients with results. One timely example is online graph processing [Cheng et al. 2012;

Authors' address: Philip Dexter, pdexter1@binghamton.edu; Yu David Liu, davidl@binghamton.edu; Kenneth Chiu, kchiu@binghamton.edu, State University of New York (SUNY) at Binghamton, 4400 Vestal Parkway East, Binghamton, New York, USA, 13902.

2475-1421/2022/10-ART157
https://doi.org/10.1145/3563320

Cipar et al. 2012; Dhulipala et al. 2019; Ediger et al. 2012; Han et al. 2014; Ju et al. 2016; Kumar and Huang 2020; Sheng et al. 2018; Shi et al. 2016; Suzumura et al. 2014; Vora et al. 2017].

Building scalable online data processing systems is notoriously challenging. Indeed, a naive online data processing system could behave just as conventional data processing, i.e., processing each operation individually. Such a design however does not scale when operation requests come at a rapid rate, especially when two challenges complicate the design space: (1) *structured data* support: whereas key-value stores or relational data were dominant in the past, more structured data — such as graphs — are increasingly prevalent; (2) *dynamic data* support: for many data-intensive applications, the data themselves are mutable, and continuously evolve as operations are processed. For example, social network analytics converge on the two challenges.

Although online data processing systems are widely deployed, faced with unique challenges, and experimentally supported with diverse and complex solutions, no prior formal foundations exist for this important family of software systems.

DON Calculus. We introduce DON Calculus, a formal foundation to account for the essential behavior of online processing in the presence of dynamic structured data. Our theoretical motivation is to understand the correctness of online data processing systems in the presence of complex optimizations. More practically, we wish to build a "base" formal system — in the artifacts of specification and mechanization — that future rigor-minded data processing system designers can build upon. With these artifacts, their effort in specification and mechanization can focus on the details unique to their system, not from scratch.

The centerpiece of DON Calculus are two essential features at the heart of online data processing systems but beyond conventional data processing:

- *Incremental Operation Processing* (IOP): operations may be deferred for incremental processing, so that the system can balance the need of processing potentially numerous operations that arrive at a rapid rate.
- *Temporal Locality Optimization* (TLO): temporally consecutive operations applied to the data may be manipulated for optimization before or during their processing, such as through batching, reordering, fusing, or reusing (see § 2.1).

IOP and TLO reflect the same philosophy that underlies the design of online data processing systems: instead of viewing the processing of each operation individually, a scalable solution should take a *multitude* of operations into account. Indeed, these two forms of optimizations essential for online data processing go hand in hand: it is often the delay resulted from IOP that enables multiple operations to participate in a TLO.

DON Calculus features an operational semantics that spans the data processing system (the *backend*) and the data processing application (the *frontend*). The backend captures the IOP and TLO behavior, and the frontend is supported with a simple programming model for constructing data-intensive applications. A key insight of DON Calculus is that the spirit of *online* data processing can be embodied by viewing the operations as a *stream*, which we call the *operation stream*; more importantly, the operation stream does not only exist at the frontend-backend boundary, but also "flows through" the data structure itself. This view is aligned with our intuition, and more importantly, it provides a unified abstraction to model the essential features of online data processing: IOP is modeled as operation propagation in the stream, and TLO is modeled as stream rewriting.

Sound Online Data Processing. DON Calculus is a rigorous study on the correctness of building online data processing systems. As we have seen, the essential features of these systems are indeed the *optimizations* designed over conventional data processing. To trust the result produced by an online data processing system, we must ensure the optimizations are *sound*: these systems must produce *deterministic* results as in conventional data processing. Enforcing result determinism however is a non-trivial problem, especially when expressive forms of IOP and TLO are in place. With IOP, significant non-deterministic executions are introduced. With TLO, the operations in the stream are altered. An important goal of DON Calculus is to establish both IOP and TLO are *sound* optimizations. The main property enjoyed by DON Calculus is an *observable equivalence* property: despite significant non-deterministic executions introduced by IOP and TLO (see § 4), all terminating executions of the same program produce the same result as a conventional processing model with neither TLO nor IOP.

DON Calculus is also endowed with a type system for its frontend programming model. The system, a standard type-and-effect system in form, enforces the novel property of *phase distinction in data processing*: while the computation at the frontend can freely issue new operations for backend processing, the backend computation should not issue new operations for processing. If phase distinction were ignored, the non-deterministic executions inherent in operation streams would lead to non-determinism in results. Intuitively, this is analogous to a high-level data race that our type system eliminates.

Mechanization. DON Calculus is mechanized in Coq, in around 7000 LOC. The proofs consist of all properties of our operational semantics as well as the type system presented in the paper. Being the first mechanization for online data processing (i.e., IOP and TLO features), this implementation may serve as a basis for rigorously specifying and reasoning about other online data processing systems, such as those with richer data processing primitives or optimizations. Our mechanization includes the confluence proof *a la* Huet [Huet 1980], which may be a reusable (side) artifact for observable equivalence proofs. The source code is available for inspection [Dexter et al. 2022].

Contributions. We envision DON Calculus can benefit the theory and practice of data processing in two dimensions. The theoretical contribution of DON Calculus is that it enriches the foundation of data processing by focusing on its *online* behavior, and especially, establishing its *soundness* in the presence of common but non-trivial optimizations of TLO and IOP. The practical contribution of DON Calculus is that it may help specify and mechanize existing or future online data processing systems (see an example in § 8.4), so that new features of optimization can be rigorously defined and reasoned about on top of a sound "base," and not from scratch. As DON Calculus and its mechanization represent a significant effort, we hope the artifacts from DON Calculus can improve the productivity of rigorous semantic engineering of future data processing systems, and ultimately, attract more developers of experimental data processing systems to formal methods.

More technically, this paper makes the following contributions:

- a foundation that captures the essence of online data processing with IOP and TLO;
- an operational semantics based on operation streams to uniformly account for IOP and TLO in one system;
- a frontend programming model with a type system to enforce phase distinction;
- a metatheory defining the soundness of online data processing, including observable equivalence and type soundness;
- the mechanized proofs for rigorous semantic engineering of online data processing systems.

2 AN INFORMAL ACCOUNT

In this section, we informally highlight the essential features of DON Calculus through examples.

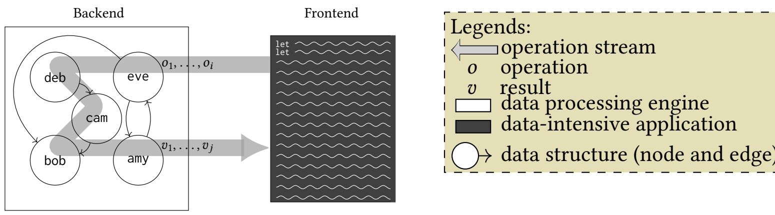


Fig. 1. The Frontend and Backend of Online Data Processing

2.1 The Big Picture: Scope and Expressiveness

The scope of our calculus is illustrated in Fig. 1. The frontend program continuously produces data processing operations such as $o_1, \ldots, o_i$ in the Figure, and delivers them to the backend that maintains a potentially large and evolving data, here a graph. As operations are processed and results become available, the backend delivers the latter back to the frontend, $v_1, \ldots, v_j$. In scope, DON Calculus spans both the backend and the frontend: the backend data processing engine enabled with IOP and TLO, and the frontend programming model for constructing online data processing applications.

A key abstraction of our calculus is the operational stream. For example, Fig. 1 shows an operation stream extends from the frontend to backend (which we call the *top-level operation stream* for convenience), and then continues to flow into nodes `eve`, `deb`, `cam`, `bob`, `amy`, in that order (which we call the *in-data operation stream*). To place this novel view in context, observe that there is a fundamental difference between the view taken by DON Calculus here and *data streaming* (e.g., [Ashcroft and Wadge 1977; Caspi et al. 1987; Meyerovich et al. 2009; Murray et al. 2013, 2011; Spring et al. 2007; Thies et al. 2002; Vaziri et al. 2014; Zaharia et al. 2013, 2016]). In DON Calculus, a stream is formed by operations, to be passed through structured data. In data streaming systems, a stream is formed by data, to be passed through structured operations. A more detailed discussion on this difference can be found in § 9.

An important design goal of DON Calculus is to provide support for *structured data*, an essential feature in state-of-the-art experimental systems. Our core calculus is defined over graph data. Relational tables and key-value stores are simpler representations that can also be supported by DON Calculus (see § 8.5).

Another design goal of DON Calculus is to support *dynamic data*. Not only the "payload" values carried by data may change (e.g., the value contained in the `amy` node may be changed from 0 to 1), but also the structure of the data (e.g., a new edge may be added between `amy` and `cam`). In expressiveness, our calculus goes beyond online processing of immutable data — such as MapReduce datasets [Dean and Ghemawat 2004] or Spark RDDs [Zaharia et al. 2016] — and more on par with data processing systems where data query operations and data update operations are continuously received and processed (e.g., [Vora et al. 2017]).

2.2 Motivating Scenarios and Examples

We now use two motivating scenarios as running examples throughout the paper, demonstrating the expressiveness of the calculus.

2.2.1 Application 1: Graph Databases. Graph databases [Buneman et al. 1996; Papakonstantinou et al. 1995; Venkataramani et al. 2012] are an important family of databases that rely on structured graphs for data storage.

```
1:  // node payload values
2:  let amy, bob, cam, deb, eve, fred =
        n_amy, n_bob, n_cam, n_deb, n_eve, n_fred in
3:  // graph construction
4:  let a = add amy in
5:  let b = add bob in
6:  let c = add cam in
7:  let d = add deb in
8:  let e = add eve in
9:  addRelationship c b;
10: addRelationship d c;
11: addRelationship e b;
12: addRelationship e a;
13: addRelationship a e;
14: // dynamic queries and updates
15: let nb = queryNode b in
16: updatePayload a nb;
17: let nb2 = queryNode b in
18: let f = add fred in
19: addRelationship b f;
20: deleteRelationship b f;
21: ...
```

Fig. 2. The CoreSocial Application in DON Calculus

```
1:  let numSuperSteps = 30 in
2:  // keys of interest
3:  let keys = ... in
4:  let numNodes = length keys in
5:  let fPInit = λ⟨_; _; _⟩. 1/numNodes in
6:  mapVal fPInit keys;
7:  foreach 1..numSuperSteps
8:      let neighborPSums =
9:          [⟨nk; foldVal fPSum 0 keys⟩
10:          | nk in keys,
11:           let fPSum = λ⟨_; payload; adjlist⟩.λsum.if nk in adjlist
                                                     then payload + sum
                                                     else sum] in
12:     foreach ⟨nk; ⟨_; neighborsSum; _⟩⟩ in neighborPSums
13:         let fPG = λ⟨_; _; adjlist⟩. 0.15/numNodes + 0.85 * neighborsSum/(length adjlist) in
14:         mapVal fPG [nk]
```

Fig. 3. The CorePR Application in DON Calculus (Expressions encodable by λ calculus are liberally used, such as loop at Line 7 and list comprehension at Line 9, with a summary of encoded expressions in § 3.1.)

Example 2.1 (CoreSocial in DON Calculus Sugared Syntax). Fig. 2 shows a minimalistic program for maintaining a social network in the form of a graph database. In this sugared syntax, Lines 1-8 are node additions and Lines 9-13 are relationship additions. The remaining lines further consist of a mixture of queries (Lines 15 and 17) and updates (Lines 16, 18, 19, 20). Each data processing operation — highlighted in blue — is analogous to an API function in the graph database. The (logical) graph after the program reaches Line 13 is shown as the backend of Fig. 1.

The programmer syntax assumed by our calculus is conventional: it consists of standard features encodable by λ calculus, together with data processing primitives. As we shall see (§ 3.1), the database operations that appear in this example — such as `add`, `addRelationship`, and `updatePayload` — can be encoded by those primitives. The CoreSocial example attempts to maintain a social graph, where each node in this data structure carries a unique *key*, and also a *payload* value. For example, the `add` expression at Line 4 adds a node whose key is freshly generated, whose payload is $n_{\mathtt{amy}}$. The generated key is returned and bound to name `a`. The functionality of other database operations should be self-explanatory through their names.

Relevant to online data processing is that the database operations — 16 of them in this program — are continuously submitted to the graph database, and the graph continuously evolves.

2.2.2 *Application 2: Iterative Graph Analytics.*

Graph analytics are algorithm-centric data processing applications, often computing graph-theoretic properties. Most involve multiple *iterations* (or *supersteps*), each of which involves non-trivial computations based on graph payload and topological information.

Example 2.2 (CorePR in DON Calculus Sugared Syntax). Fig. 3 presents a 30-superstep PageRank [Brin and Page 1998] algorithm in DON Calculus. Lines 2-4 compute the number of nodes. Lines 5-6 initialize the node payloads. Line 7 iterates over supersteps. Each superstep has two sub-steps. The first sub-step, at Lines 9-11, computes the sum of payload values for each node's in-degree adjacent nodes. The second sub-step, a loop at Lines 12-14, updates each node with a new payload value, by utilizing `fPG`, the core PageRank aggregation function.

As shown here, a graph analytical program may consist of numerous data processing operations — within a superstep and across supersteps — continuously applied to the graph. In this program, the two forms of graph processing operations are shown in blue. The `mapVal-foldVal` pair is standard, except for a small variation. The *selective* map/fold is supported here: the last argument for the `mapVal` or `foldVal` operation is the keys which identify which nodes the operation should be applied to.

2.3 The Frontend-Backend Interaction

In DON Calculus, a simple *asynchronous* semantics is designed for data processing operations: the evaluation of a data processing operation at the frontend does *not* need to block until the backend returns the result. Instead, the evaluation places the operation of concern into the operation stream destined for the backend, which we say the operation is *emitted* from now on. DON Calculus follows the same route of futures [Flanagan and Felleisen 1995, 1999; Halstead 1985]. For example, the emission of `add amy` at Line 4 in Example 2.1 generates a future value, which is subsequently *claimed* at Line 12 *a la* future semantics. Modeling the frontend-backend interaction through asynchronous semantics aligns with the philosophy well-articulated for asynchronous data processing [Bertsekas and Tsitsiklis 1989; Elteir et al. 2010; Wang et al. 2013].

Example 2.3 (Operation Streams). The operations emitted at Line 4-8 form an operation stream are $[\mathsf{add}\ \mathsf{n}_{\mathsf{amy}}, \mathsf{add}\ \mathsf{n}_{\mathsf{bob}}, \mathsf{add}\ \mathsf{n}_{\mathsf{cam}}, \mathsf{add}\ \mathsf{n}_{\mathsf{deb}}, \mathsf{add}\ \mathsf{n}_{\mathsf{eve}}]$.

Similarly, for the CorePR example, each evaluation at Line 3, Line 6, Line 9, and Line 14, results in emitting a data processing operation to the operation stream.

DON Calculus supports *dependent operations*: an operation may have an argument referring to the result of an earlier emitted operation. For example, Line 15 queries the node `b` through the `queryNode` expression. The resulting value is used to update the payload of the node `a` at Line 16 through the `updatePayload` expression. The interaction between asynchrony and dependency naturally calls for the *backend claim*, a feature of DON Calculus.

Example 2.4 (Backend Claim). The execution of Line 15-16 emits both operations into the operation stream. At the *backend*, the argument of the `updatePayload` expression, a future value, can be claimed upon the completion of processing `queryNode`, without any interaction with the frontend.

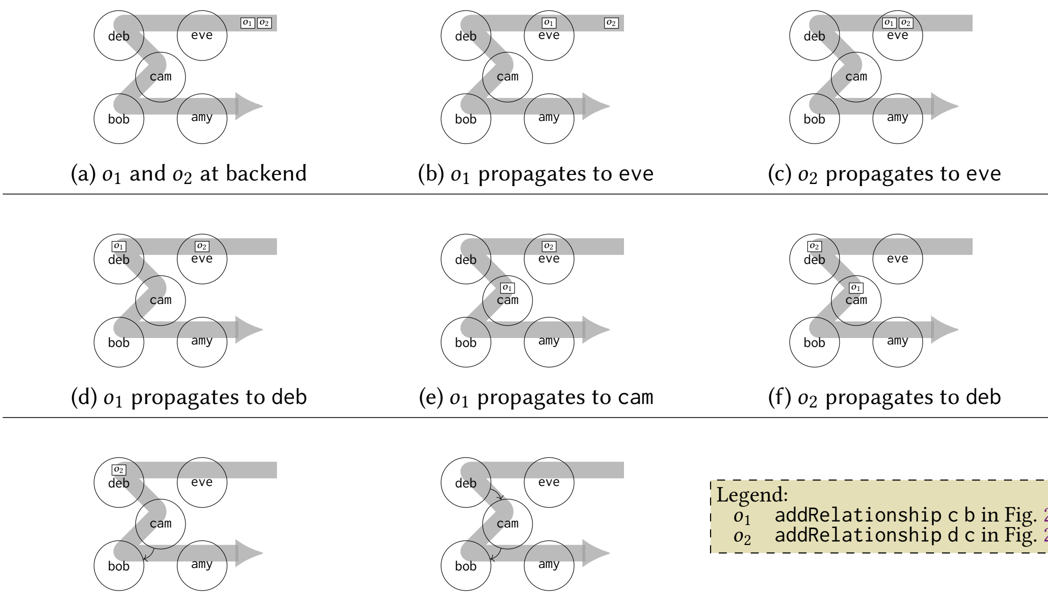
2.4 Incremental Operational Processing (IOP) in Online Data Processing

Taking a per-operation view, data processing can be viewed as a process that reaches the data nodes one by one through data scans or traversals (*propagation*), and along the way, computation is performed when the operation reaches the data node(s) it is intended for (*realization*). The default "baseline" behavior in data processing is *eager processing*, where the processing of an operation must be completed once it is started.

Example 2.5 (Eager Processing). If one were to apply eager processing for executing Lines 9-10 in Fig. 2, and if we use o_1 and o_2 to represent the two operations issued at Line 9 and Line 10 respectively, the backend would process o_1 first, traversing through `eve`, `deb` and `cam`, and finally realizing at the latter. After the completion of o_1, the traversal of the graph may start for o_2, through nodes `eve` and `deb`, and finally realize at `deb`.

In contrast, DON Calculus supports *incremental in-data* processing:

Example 2.6 (In-Data Operation Streams). Fig. 4 illustrates 8 runtime configurations of the backend graph for CoreSocial in a DON Calculus reduction sequence. The first one coincides with the moment when the processing of Line 1-8 in Fig. 2 is completed, and the operations in Line 9-10



(a) o_1 and o_2 at backend (b) o_1 propagates to eve (c) o_2 propagates to eve

(d) o_1 propagates to deb (e) o_1 propagates to cam (f) o_2 propagates to deb

(g) o_1 realizes at cam (h) o_2 realizes at deb

Fig. 4. In-Data Operation Streams for Fig. 2 Lines 9-10.

have been emitted but not processed. These two operations o_1 and o_2 flow through the graph nodes following the traversal of eve, deb, cam, bob, amy, in that order. Intuitively, the in-data stream view entails that the processing of multiple operations may co-exist: for configurations (b)(c)(d)(e)(f), neither o_1 nor o_2 is completed. In addition, the propagation steps for different operations may intermingle, the first 3 transitions in Fig. 4 are propagation steps for o_1, o_2, and o_1, respectively.

In-data operation streams are a novel feature in our calculus. They provide a flexible and natural design for IOP, as the operation can be incrementally applied through the data items (graph nodes here), and be deferred at any arbitrary data node and resumed later. Deferred operation processing is a common optimization in online data processing systems [Cheng et al. 2012; Dexter et al. 2016; Sheng et al. 2018; Vora et al. 2017]; our stream-based design captures the general scenario where the operations may be deferred at an arbitrary step of data scan. A second benefit of in-data operation streams is it enables TLO "on the fly"; see § 4.2 for details.

The behavior exhibited in Example 2.6 is *incremental propagation*: the processing of o_1 can be deferred without the need of "rushing" to its realization. When o_1 is deferred, the runtime can process (i.e., either propagate or realize) another operation, such as the later emitted o_2.

As a general calculus, DON Calculus places no restriction on the "schedule" of operation stream processing: when multiple operations are processed, a *non-deterministic* choice can be made as to which operation should take a step. For example, instead of transitioning from Fig. 4(b) to Fig. 4(c), the program runtime may choose to have o_1 take another propagation step to deb. To ensure result determinism, a non-deterministic propagation is *not* an arbitrary propagation. In particular, the operations in the operation stream form a *chronological order* of emission. It must be preserved unless TLO allows for reordering.

Example 2.7 (Chronological Order Preservation). Let us assume the payload value in amy is initially 1, i.e., $\mathsf{n}_{\mathsf{amy}} = 1$. Operation o_1 is an operation to double the payload of amy while o_2 is an operation

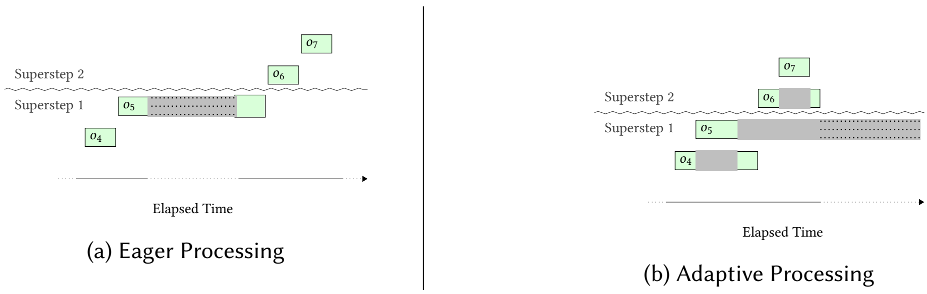


Fig. 5. PageRank with Stragglers (The CorePR program is applied to a graph with two nodes. Notation o_i refers to the i^{th} operation in the operation stream. With that, o_4 and o_5 refer to the two `mapVal` operations emitted at Line 14 in superstep 1, and o_6 and o_7 refer to the two `foldVal` operations emitted at Line 9 in superstep 2. Gray area indicates wait and dotted gray area indicates straggling. We assume the processing of the straggler will eventually complete, normally or through a time-out.)

to add the payload of `amy` by 10. After the two operations are completed, `amy` should have a payload of 12. Should we allow o_2 to "swap" with o_1, the payload of `amy` would be 22.

A data processing system that supports non-deterministic executions *but* deterministic results — which DON Calculus enjoys — is good news for adaptiveness support, which we now illustrate through a so-called "straggler" example, a classic problem in data processing [Ousterhout et al. 2015].

Example 2.8 (Superstep Blending for Straggler Mitigation). Fig. 5 illustrates two timelines of execution of CorePR. Due to system resource fluctuations and transient failures, the processing of operation o_5 may be suspended, becoming a straggler. In Fig. 5a, the slowdown by the straggler delays the beginning of the next superstep. In Fig. 5b however, while the straggler is suspended, operation o_6 in the second superstep may start, interleaving the two supersteps.

Another dimension of IOP support is *incremental load update*, where the *load* refers to the payload expression carried by a data item:

Example 2.9 (Incremental Load Update). Suppose the operation at Line 14 of Fig. 3 is processed at the backend and the node indicated by `nk` is reached whose payload value is 5. The realization step of our calculus will update the node payload with expression `fPG` 5, without evaluating it immediately.

IOP does not change the complexity of operation processing. In our system, an operation incrementally propagates through the in-data operation stream, with a complexity of $O(n)$ where n is the data size. In eager data processing systems, the query/operation processing engine still needs to *scan* or *traverse* the data to process a query, with complexity of $O(n)$. Indeed, eager data processing is formally a special case in DON Calculus (as we will see). In practice, many $O(n)$ algorithms in data processing are experimentally effective, especially in the presence of parallelism. In § 8.6, we discuss the relationship between DON Calculus and sublinear operation processing.

2.5 Temporal Locality Optimization (TLO) in Online Data Processing

TLO is a broad family of optimizations. For the simple case of two temporally consecutive operations o_1 and o_2, where o_1 is submitted to the data processing engine before o_2, four forms of TLO are well-known and captured in DON Calculus:

- *Batching*: processing o_1 and o_2 "in tandem," so that only one data scanning/traversal is needed for processing both, as opposed to two if o_1 and o_2 are processed one by one.
- *Reordering*: processing o_2 first and o_1 later, on the assumption that the reversal does not impact the result. Reordering is useful in use scenarios e.g., when o_2 has a higher priority or a closer deadline.
- *Fusing*: composing o_1 and o_2 into one operation o, on the assumption that processing o can produce the same result as processing both o_1 and o_2. Just like batching, fusing is useful in reducing the amount of data scanning/traversal.
- *Reusing*: applying o_1 and o_2' to the data where o_2' derives from o_2 but reuses the result of o_1 processing to avoid redundant computation. This style of TLO is known as Multi-Query Optimization (MQO) [Park and Segev 1988; Sellis 1988; Sellis and Shapiro 1985].

We now revisit the CoreSocial example to illustrate the common forms of TLO that DON Calculus supports. A novel consequence of in-data operation streams is that they enable on-the-fly TLOs: optimization may happen while multiple operations are incrementally propagated to an arbitrary data node in the in-data operation stream, leading to in-data batching, in-data reordering, in-data fusing, and in-data reusing. In other words, our calculus highlights *where* and *when* TLOs may happen, in addition to *how* they are defined.

Example 2.10 (In-Data Operation Batching). Consider Fig. 4(c). Since neither `addRelationship` operation realizes at `eve`, both may propagate in a "batch" to `deb` in one reduction step.

Example 2.11 (In-Data Operation Reordering). Consider a configuration where 3 operations at Lines 15-17 in Fig. 2 reach node `deb`. The third operation, `queryNode b`, reads from `b` while the second operation writes to `a`. The latter 2 operations can "swap" since they do not operate on the same node.

Example 2.12 (In-Data Operation Fusing). Imagine two operations at Lines 19-20 in Fig. 2 before they reach node `bob`. DON Calculus allows the `addRelationship` and `deleteRelationship` operations to "cancel out" so that further processing of both is avoided.

Example 2.13 (In-Data Operation Reusing). Let us follow up on Example 2.11. After swapping, two `queryNode b` operations are adjacent in the operation stream at node `deb`. DON Calculus allows the second instance to immediately return, referencing the return value of the first instance.

In DON Calculus, TLOs are supported through rewriting rules over the operation stream. Not to lose generality, TLOs are applied dynamically. This is aligned with our "open-world" assumption on the usage scenarios in practice: when the program is compiled, the operations may not be statically known yet. In other words, the program we showed in Fig. 2 may well be a textual *a posteriori* representation of an interactive program, where each line of graph processing operation is submitted through some interactive graphical interface.

2.6 A Type System for Phase Distinction

The primary goal of DON Calculus's type system is to enforce a *phase distinction* of operation emission: the backend should not emit an operation for processing while processing another operation. To see why this restriction is important, let us start with a counterexample.

Example 2.14 (Backend Operation Emission). Consider the following program (that does not typecheck in DON Calculus):

Expressions, Operations, Values	
$e ::= v \mid e\ e \mid x \mid \mathbf{fix}\ e$	*expression*
$\quad \mid\ K \mid e \oplus e \mid e \ominus e$	
$\quad \mid\ N \mid {}^{\pi}e$	
$\quad \mid\ o \mid \Downarrow e$	
$o ::= \mathsf{add}\ e \mid \mathsf{map}\ e\ e \mid \mathsf{fold}\ e\ e\ e$	*operation*
$v ::= f \mid k \mid n \mid \mathit{KLV} \mid \mathit{NV} \mid \ell$	*value*
$f ::= \lambda x : \tau.e$	*function*
ℓ	*future value/label*
τ	*type (see § 5)*

Keys, Nodes, Integers, Names	
$k \in \mathbb{KEY}$	*key*
$K ::= \mathbf{KL}\langle \overrightarrow{e} \rangle$	*key list*
$\mathit{KLV} ::= \mathbf{KL}\langle \overrightarrow{k} \rangle \in \mathbb{KLV}$	*key list value*
$N ::= \mathbf{N}\langle e; e; e \rangle$	*node*
$\mathit{NV} ::= \mathbf{N}\langle k; n; \mathit{KLV} \rangle$	*node value*
n	*integer*
x, y, z, u, w	*name*
$\pi \in \{1, 2, 3\}$	*projection index*

Fig. 6. Abstract Syntax

```
let k = ... // key of interest
let f = λ⟨_; payload; _⟩.payload * 2 in
let g = λ⟨_; payload; _⟩.(mapVal f [k]; payload) in
mapVal g [k];
let h = λ⟨_; payload; _⟩.payload + 1 in
mapVal h [k]
```

If the operation `mapVal f [k]` inside the body of `g` is emitted *before* the operation `mapVal h [k]` is emitted, the node with key `k` will have its payload value multiplied by 2 and then incremented by 1. If the order is reversed, the payload value will be incremented by 1 and then multiplied by 2.

The root problem is that the evaluation order between the backend-emitted `mapVal` and the frontend-emitted one cannot be decided upon, a symptom analogous to a race condition. Our type system disallows backend operation emission through effect types: for every operation that is emitted from the frontend, we guarantee that its processing does not have the effect of operation emission. As a result, the program in Example 2.14 fail to typecheck.

3 SYNTAX AND RUNTIME STRUCTURES

In this section, we provide definitions for DON Calculus, including abstract syntax in § 3.1 and runtime configuration in § 3.2.

Notations. We summarize 3 common structures used in this paper: sequence, set, and mapping. We use notation $[\sigma_1, \sigma_2, \ldots, \sigma_m]$ to represent a sequence of $\sigma_1, \ldots, \sigma_m$ in that order for some $m \geq 0$; we shorthand it as $\overrightarrow{\sigma}^m$, or $\overrightarrow{\sigma}$ when its length does not matter. We further call σ_1 as the *head element* and σ_m as the *last element*. When $m = 0$, we further represent an empty sequence as $[]$. Binary operator $\sigma :: \Sigma$ prepends σ to sequence Σ as the head, and binary operator $\Sigma +\!\!+ \Sigma'$ concatenates Σ and Σ' together. We elide their definitions here. We use notation $\{\sigma_1, \sigma_2, \ldots, \sigma_m\}$ to represent a set with elements $\sigma_1, \ldots, \sigma_m$ for some $m \geq 0$; we shorthand it as $\overline{\sigma}^m$, or $\overline{\sigma}$ when its length does not matter. When $m = 0$, we further represent an empty set as $\{\}$. Common set operators $\in$, $\subseteq$, and $\cap$ apply. We overload the operator $|\star|$ to compute the size of $\star$, where $\star$ may either be a sequence or a set.

When a sequence takes the form of $\overrightarrow{\sigma \mapsto \sigma'}^m$ or when a set takes the form of $\overline{\sigma \mapsto \sigma'}^m$, we call it a *mapping* when $\sigma_1, \ldots, \sigma_m$ are distinct. Given μ as the aforementioned mapping, we further define $\mu(\sigma_i)$ as σ'_i for some $1 \leq i \leq m$; $dom(\mu)$ as $\overline{\sigma}^m$; and $ran(\mu)$ as $\overline{\sigma'}^m$.

We omit some common definitions in λ calculus: $e[v/x]$ for substitution of name x with value v for expression e; $\equiv$ for term equivalence; `Id` for the identity function; and $\circ$ for function composition.

3.1 Syntax

Fig. 6 defines the abstract syntax of DON Calculus. It consists of conventional λ calculus features, such as name, abstraction, application, and fixpoint computation. Encodable features that appeared

Programmer Syntax	Formal Syntax	f
$\langle e; e'; e''\rangle$	$\mathbf{N}\langle e; e'; e''\rangle$	
$[e_1, \ldots, e_n]$	$\mathbf{KL}\langle [e_1, \ldots, e_n]\rangle$	
`addRelationship` $e\ e'$	`map f` $\mathbf{KL}\langle [e]\rangle$	$\lambda x.\mathbf{N}\langle {}^1x; {}^2x; {}^3x \oplus \mathbf{KL}\langle [e']\rangle\rangle$
`deleteRelationship` $e\ e'$	`map f` $\mathbf{KL}\langle [e]\rangle$	$\lambda x.\mathbf{N}\langle {}^1x; {}^2x; {}^3x \ominus \mathbf{KL}\langle [e']\rangle\rangle$
`updatePayload` $e\ e'$	`map f` $\mathbf{KL}\langle [e]\rangle$	$\lambda x.\mathbf{N}\langle {}^1x; {}^2e'; {}^3x\rangle$
`queryNode` e	`fold f` $\mathbf{N}\langle \_; 0; \mathbf{KL}\langle []\rangle\rangle\ \mathbf{KL}\langle [e]\rangle$	$\lambda x.\lambda y.x$
`mapVal` $e\ e'$	`map f` e'	$\lambda x.\mathbf{N}\langle {}^1x; e\ x; {}^3x\rangle$
`foldVal` $e\ e'\ e''$	`fold f` $\mathbf{N}\langle \_; e'; \mathbf{KL}\langle []\rangle\rangle\ e''$	$\lambda x.\lambda y.\mathbf{N}\langle {}^1y; e\ x\ {}^2y; {}^3y\rangle$

Table 1. Data Processing Operations Encodings

in the earlier examples are omitted, including `if − then − else`, list comprehension, `let − in`, the ; expression, and `foreach`.

Values. The values of our language are functions (f), node keys (k), node payloads (n), key list values (KLV), node values (NV), and futures (ℓ).

Both the key list and the node are first-class citizens in our calculus. In the programmer syntax, the former is represented as a sequence and the latter as a triple. To differentiate programming abstractions from meta-level structures, we associate the key list with an explicit constructor **KL** and the node with constructor **N** in the formal syntax, as shown in Table 1. We use ${}^{\pi}e$ to project the first, second, and third component of a node e when $\pi = 1, 2, 3$, respectively.

Key lists in our calculus play two roles: defining the (ordered) adjacency list of a node, and providing as argument for selective mapping and folding. Each data node (N) is a triple: a key, a payload expression, and an adjacency list expression. The last component accounts for the structural information latent in structured data, intuitively, the "out-edges" of the node.

A future value ℓ is generated when an operation is emitted (§ 2.2.1), and as we shall see soon, it also serves as the unique label for identifying the operation and its result in backend data processing. Except for futures, all forms of values are also programs, including keys. To be consistent with real-world practice, we allow programmers to name a key in their program.

```
let nb = queryNode ⇓ b in
updatePayload ⇓ a ⇓ nb;
let nb2 = queryNode ⇓ b in
let f = add ⇓ fred in
addRelationship ⇓ b ⇓ f;
deleteRelationship ⇓ b ⇓ f;
...
```

Fig. 7. Fig. 2 Lines 14-21 in Formal Syntax

Data Processing Operations and Lifecycle Support. Two new expressions handle the operation stream at the frontend: operation emission (o) and result claim ($\Downarrow e$). To highlight the asynchronous nature of operation processing, each program point of result claim in the programmer syntax is annotated with a $\Downarrow$ symbol explicitly. For example, Fig. 7 shows how the Lines 14-21 of Fig. 2 can be explicitly annotated with $\Downarrow$.

DON Calculus supports 3 core operations: `add`, `map`, and `fold`. The first operation has been used in the CoreSocial and CorePR examples. The second and third operations are similar to `mapVal` and `foldVal` in CorePR, except that the mapping function argument of `map` returns a node, and the folding function argument of `fold` is a binary function over nodes. These primitives are sufficient to encode all data processing operations in CoreSocial and CorePR, as shown in Table 1. In § 8, we will further discuss how they can encode other common programming idioms. Finally, `add` is useful to support dynamic data (§ 2.2.1).

For operations, we introduce a convenience function $\odot$ that computes the keys of nodes where the operation is intended for realization:

Definition 3.1 (Operation Target). The function $\odot(o)$ computes the *target* of the operation o, defined as $\overline{k}$ if $o = \mathtt{map}\ f\ \mathbf{KL}\langle \overrightarrow{k}\rangle$ or $o = \mathtt{fold}\ f\ e\ \mathbf{KL}\langle \overrightarrow{k}\rangle$. The operator is undefined for `add`.

$C ::= \langle B; O; R; e\rangle$		*configuration*
$B ::= \overrightarrow{S}$		*backend*
$S ::= \langle N; O\rangle$		*station*
$O ::= \overrightarrow{U}$		*operation stream/streamlet*
$U ::= \overrightarrow{\ell \mapsto o}$		*stream unit*
$R ::= \overline{\ell \overset{KLV}{\longmapsto} v}$		*result store*

Fig. 8. Runtime Definitions

$$\begin{aligned}&[\langle \mathbf{N}\langle k_{\mathsf{eve}}; n_{\mathsf{eve}}; \mathbf{KL}\langle[k_{\mathsf{bob}}, k_{\mathsf{amy}}]\rangle\rangle; [[\ell_2 \mapsto o_2]]\rangle,\\ &\langle \mathbf{N}\langle k_{\mathsf{deb}}; n_{\mathsf{deb}}; \mathbf{KL}\langle[k_{\mathsf{cam}}]\rangle\rangle; [[\ell_1 \mapsto o_1]]\rangle,\\ &\langle \mathbf{N}\langle k_{\mathsf{cam}}; n_{\mathsf{cam}}; \mathbf{KL}\langle[k_{\mathsf{bob}}]\rangle\rangle; []\rangle,\\ &\langle \mathbf{N}\langle k_{\mathsf{bob}}; n_{\mathsf{bob}}; \mathbf{KL}\langle[]\rangle\rangle; []\rangle,\\ &\langle \mathbf{N}\langle k_{\mathsf{amy}}; n_{\mathsf{amy}}; \mathbf{KL}\langle[k_{\mathsf{eve}}]\rangle\rangle; []\rangle]\end{aligned}$$

Fig. 9. A Backend Example of Fig. 4(d) (k_{amy}, k_{bob}, k_{cam}, k_{deb}, k_{eve} are keys of corresponding nodes and ℓ_1, ℓ_2 are labels for o_1, o_2)

Additional Expressions. In addition to being in the value form, a key list or a node may also be in its expression form, K and N respectively, when any of its components is not in the value form. The $\oplus$ and $\ominus$ expressions are binary operators over key lists for their concatenation and subtraction respectively. To support key list subtraction, we define function $\overrightarrow{k} \setminus \overrightarrow{k'}$ as identical to $\overrightarrow{k}$ except that every element that appears in $\overrightarrow{k'}$ is removed.

3.2 The Structure of the Runtime

As shown in Fig. 8, a runtime configuration C consists of 4 components: the (backend) runtime data structure B, the (frontend) expression e, and two structures that bridge them: the (top-level) *operation stream* O and the *result store* R.

We represent the runtime data structure as a sequence of runtime nodes called *stations*, each of which consists of a data node (N) and the operations (O) that have so far propagated to that node. We also call the latter as a *streamlet*. In other words, the in-data operation stream is composed of per-station streamlets. An example of the formal data representation can be found in Fig. 9. This representation reflects the *fine-grained* nature of our support for incremental processing: the operation can propagate to and be deferred at *any node*. Client calculi to DON Calculus can further restrict this most general treatment, e.g., a more implementation-oriented choice where nodes form partitions and streamlets can only be associated with (the first node of) partitions.

Placing the data nodes into a sequence faithfully captures experimental data processing systems. It may be tempting to represent the structured graph data as a linked data structure, i.e., a formal representation of in-memory graphs through C-like pointers or Java-like object references. Unfortunately, experimental *graph* processing systems rarely adopt this form. The root cause is that they routinely process data that exceed the memory capacity, so their runtime representation is strongly influenced by the graph representation in file or storage systems, where ordered access dominates. In addition, this choice of representation does not impact the expressiveness of our calculus: for smaller in-memory data structures implemented as a linked data structure, traversal algorithms (e.g., depth-first or breadth-first) can always place nodes into a sequence. For example, if the graph in Fig. 1 is implemented as a linked data structure, its traversal order — `eve`, `deb`, `cam`, `bob`, `amy` — remains a sequence which DON Calculus can work with.

We formally represent an operation stream/streamlet as a sequence of *stream units* (U), each of which is a sequence of operations. This 2-dimensional representation — instead of a 1-dimensional one — results from batching (§ 2.4), so that each stream unit can be viewed as a "batch." In the stream/streamlet, the operation is indexed by a unique label (ℓ). Each element in the result store takes the form of $\ell \overset{KLV}{\longmapsto} v$, associating result value v with label ℓ. The additional KLV is called a *residual target.* If any key in the target key list of an operation cannot be found during processing, it will be kept as the residual target in the result store.

The following definitions highlight the different access patterns of the operation stream and the result store: whereas order does not matter for the latter, it matters for the former (recall § 2.4):

$$\begin{array}{lr} \mathbb{F} ::= \langle B; O; R; \mathbb{E}\rangle & \textit{frontend context} \\ \mathbb{B} ::= \langle \bullet; O; R; e\rangle & \textit{backend context} \\ \mathbb{T} ::= \mathbb{B}[B \mathbin{+\!\!+} \bullet \mathbin{+\!\!+} B] & \textit{task context} \\ \mathbb{L} ::= \mathbb{T}[[\langle \mathbb{E}; O\rangle]] & \textit{load context} \\ \quad \mid \ \mathbb{T}[[\langle N; O \mathbin{+\!\!+} [\ell \mapsto \texttt{fold}\ f\ \mathbb{E}\ \mathit{KLV}] :: O\rangle]] & \end{array}$$

$$\begin{array}{ll} \mathbb{E} ::= \bullet \mid \mathbb{E}\ e \mid v\ \mathbb{E} \mid {}^{\pi}\mathbb{E} \mid \mathbf{KL}\langle \overrightarrow{k} \mathbin{+\!\!+} [\mathbb{E}] \mathbin{+\!\!+} \overrightarrow{e}\rangle & \textit{expression context} \\ \quad \mid \ \mathbf{N}\langle \mathbb{E}; e; e\rangle \mid \mathbf{N}\langle k; \mathbb{E}; e\rangle \mid \mathbf{N}\langle k; n; \mathbb{E}\rangle & \\ \quad \mid \ \mathbb{E} \oplus e \mid \mathit{KLV} \oplus \mathbb{E} \mid \mathbb{E} \ominus e \mid \mathit{KLV} \ominus \mathbb{E} & \\ \quad \mid \ \texttt{add}\ \mathbb{E} \mid \texttt{map}\ \mathbb{E}\ e \mid \texttt{map}\ f\ \mathbb{E} & \\ \quad \mid \ \texttt{fold}\ \mathbb{E}\ e\ e \mid \texttt{fold}\ f\ \mathbb{E}\ e \mid \texttt{fold}\ f\ v\ \mathbb{E} \mid \Downarrow \mathbb{E} & \end{array}$$

Fig. 10. Evaluation Contexts

Definition 3.2 (Operation Stream Addition and Result Store Addition). The (overloaded) ◀ operator appends a stream unit to the configuration, or appends a stream unit to a non-empty backend, or adds results to the configuration:

$$\langle B; O; R; e\rangle \blacktriangleleft U \triangleq \langle B; O \mathbin{+\!\!+} [U]; R; e\rangle$$
$$\langle N; O\rangle :: B \blacktriangleleft U \triangleq \langle N; O \mathbin{+\!\!+} [U]\rangle :: B$$
$$\langle B; O; R; e\rangle \blacktriangleleft R' \triangleq \langle B; O; R' \cup R; e\rangle$$

The definition above says that any addition to an operation stream — be it a top-level operation stream or a streamlet — must be *appended.* As we shall see in the operational semantics, any *removal* from the operation stream will be from the head. It is through this consistent access pattern that the chronological order of the operations is preserved in our semantics.

4 DON CALCULUS OPERATIONAL SEMANTICS

The main reduction system is presented in § 4.1. The semantics of TLO is an independent system that bridges with the main system via one reduction rule, whose details are in § 4.2.

4.1 Semantics for Online Data Processing

The reduction relation $C \longrightarrow C'$ in Fig. 11 says that configuration C one-step reduces to configuration C'. We use $\longrightarrow^*$ to represent the reflexive and transitive closure of $\longrightarrow$. Evaluation contexts are defined in Fig. 10. To simplify our discussion, we classify $\longrightarrow$ reduction into 4 categories, based on *where* a reduction happens.

1) Frontend Reduction. Rules with the $\mathbb{F}$ evaluation context enable reductions that happen on the frontend. The pair of Emit and Claim rules define the behavior of asynchronous operation processing at the frontend, with the former placing an operation on the top-level operation stream, and the latter reading from the result store. The definition here follows future semantics, where the fresh label in Emit is the future value. We say an operation is emittable if all of its arguments are values, which we represent as metavariable ω:

$$\omega ::= \texttt{add}\ n \mid \texttt{map}\ f\ \mathit{KLV} \mid \texttt{fold}\ f\ v\ \mathit{KLV}$$

Both nodes and key lists as first-class citizens can be constructed at the frontend. The components of a node may be inspected through Node. Key list concatenation and subtraction are defined through KSA and KSS respectively. The rest of the frontend computation is enabled by Beta, in a call-by-value style.

2) In-Data Task Reduction. On the backend, in-data processing may either be enabled by a *task reduction* and a *load reduction*, the first of which we describe now. Rules with the $\mathbb{T}$ evaluation context enable reductions that perform a *task*, i.e., a step on operation processing.

The task that "drives" the data processing at the backend is propagation, an instance of Prop. A step of operation propagation involves two consecutive stations in the runtime data structure. The reduction removes the *head* element (the oldest element) from the streamlet of the first station, and places it to the *last* element (the youngest element) of the streamlet in the second station. It is important to observe that the selection of redex for propagation is non-deterministic according to

$$\textsc{Emit}\ \frac{\ell \text{ fresh}}{\mathbb{F}[\omega] \rightarrow \mathbb{F}[\ell] \blacktriangleleft [\ell \mapsto \omega]} \qquad \textsc{Claim}\ \frac{\mathbb{F}[\Downarrow \ell] = \langle B; O; R; e\rangle \qquad \ell \in dom(R)}{\mathbb{F}[\Downarrow \ell] \rightarrow \mathbb{F}[R(\ell)]} \qquad \textsc{Beta}\ \mathbb{F}[(\lambda x.e)\ v] \rightarrow \mathbb{F}[e[v/x]]$$

$$\textsc{Node}\ \frac{N = \mathbf{N}\langle v_1; v_2; v_3\rangle}{\mathbb{F}[{}^{\pi}N] \rightarrow \mathbb{F}[v_\pi]} \qquad \text{KSA}\ \mathbb{F}[\mathbf{KL}\langle \vec{k}\rangle \oplus \mathbf{KL}\langle \vec{k'}\rangle] \rightarrow \mathbb{F}[\mathbf{KL}\langle \vec{k} +\!\!+ \vec{k'}\rangle]$$

$$\text{KSS}\ \mathbb{F}[\mathbf{KL}\langle \vec{k}\rangle \ominus \mathbf{KL}\langle \vec{k'}\rangle] \rightarrow \mathbb{F}[\mathbf{KL}\langle \vec{k} \setminus \vec{k'}\rangle] \qquad \textsc{Map}\ \frac{\begin{array}{c} k \in \overline{k}_1 \qquad o_i = \mathtt{map}\ f\ \mathbf{KL}\langle \vec{k}_i\rangle \text{ for } i = 1,2 \\ N_1 = \mathbf{N}\langle k; e; e'\rangle \qquad N_2 = \mathbf{N}\langle k; {}^2(f\ N_1); {}^3(f\ N_1)\rangle \qquad \vec{k}_2 = \vec{k}_1 \setminus k \end{array}}{\mathbb{T}[\langle N_1; [\ell \mapsto o_1] :: O\rangle] \rightarrow \mathbb{T}[\langle N_2; [\ell \mapsto o_2] :: O\rangle]}$$

$$\textsc{Fold}\ \frac{\begin{array}{c} k \in \overline{k}_1 \\ o_i = \mathtt{fold}\ f\ e_i\ \mathbf{KL}\langle \vec{k}_i\rangle \text{ for } i = 1,2 \qquad N = \mathbf{N}\langle k; e'; e''\rangle \qquad e_2 = f\ N\ e_1 \qquad \vec{k}_2 = \vec{k}_1 \setminus k \end{array}}{\mathbb{T}[\langle N; [\ell \mapsto o_1] :: O\rangle] \rightarrow \mathbb{T}[\langle N; [\ell \mapsto o_2] :: O\rangle]}$$

$$\textsc{Prop}\ \frac{k \notin \bigcup_{o \in ran(U)} \odot(o) \qquad B_i = [\langle \mathbf{N}\langle k; e; e'\rangle; O_i\rangle] \text{ for } i = 1,2 \qquad O_1 = U :: O_2}{\mathbb{T}[B_1 +\!\!+ B] \rightarrow \mathbb{T}[B_2 +\!\!+ (B \blacktriangleleft U)]}$$

$$\textsc{Complete}\ \frac{B_i = [\langle N; O_i\rangle] \text{ for } i = 1,2 \qquad O_1 = [\ell \mapsto o] :: O_2 \qquad \odot(o) = \{\}}{\mathbb{T}[B_1] \rightarrow \mathbb{T}[B_2] \blacktriangleleft [\circlearrowleft \ell \mapsto o]}$$

$$\textsc{Last}\ \frac{\mathbb{T} = \mathbb{B}[B +\!\!+ \bullet] \qquad N = \mathbf{N}\langle k; e; e'\rangle \qquad k \notin \odot(o)}{\mathbb{T}[\langle N; \ell \mapsto o :: O\rangle] \rightarrow \mathbb{T}[\langle N; O\rangle] \blacktriangleleft [\circlearrowleft \ell \mapsto o]} \qquad \textsc{Opt}\ \frac{B_i = [\langle N; O +\!\!+ O_i +\!\!+ O'\rangle] \text{ for } i = 1,2 \qquad O_1 \rightsquigarrow O_2, R}{\mathbb{T}[B_1] \rightarrow \mathbb{T}[B_2] \blacktriangleleft R}$$

$$\textsc{Load}\ \frac{\mathbb{L}[e] = \langle B; O; R; e''\rangle \qquad \langle []; []; R; e\rangle \rightarrow \langle []; []; R; e'\rangle}{\mathbb{L}[e] \rightarrow \mathbb{L}[e']}$$

$$\textsc{Empty}\ \frac{o \neq \mathsf{add}\ n}{\langle []; [\ell \mapsto o] :: O; R; e\rangle \rightarrow \langle []; O; \{\circlearrowleft \ell \mapsto o\} \cup R; e\rangle} \qquad \textsc{First}\ \frac{o \neq \mathsf{add}\ n}{\langle B; [\ell \mapsto o] :: O; R; e\rangle \rightarrow \langle B \blacktriangleleft [\ell \mapsto o]; O; R; e\rangle}$$

$$\textsc{Add}\ \frac{k \text{ fresh}}{\langle B; [\ell \mapsto \mathsf{add}\ n] :: O; R; e\rangle \rightarrow \langle \mathbf{N}\langle k; n; \mathbf{KL}\langle []\rangle\rangle :: B; O; \{\ell \xmapsto{\mathbf{KL}\langle []\rangle} k\} \cup R; e\rangle}$$

Fig. 11. DON Calculus Operational Semantics

the definition of $\mathbb{T}$. In other words, propagation may happen between any adjacent two stations in the runtime data.

The realizations of `map` and `fold` are defined by Map and Fold, over a single station as the redex. The task reduction for `map` realization happens when the key of the redex is included in the target key list, the second argument of the `map` operation. It further applies the mapping function (the first argument) to the current node, which computes a new node to update the current node. Following the convention in data processing, Map does not allow a `map` operation to update the key of the node: even though the node payload and the data structure topology can be changed in dynamic data, keys as unique identifiers of nodes do not change. In Fold, the folding function is applied to the current node, and the resulting expression becomes the initial expression (the second argument) of the `fold` operation for further propagation. Both Map and Fold demonstrate the incremental nature of load update (recall § 2.4). For example, when being applied, the `map` operation does not immediately evaluate the resulting payload expression or adjacency list expression to a value.

When the target of the `map` (or `fold`) operation contains multiple keys, its processing is "incremental": the processing consists of many PROP steps occasionally interposed by MAP (or FOLD) steps. We will show an example of this incremental process shortly, in Example 4.1.

Finally, COMPLETE and LAST are a pair of rules to "wrap up" the processing of an operation. The former captures the case when a `map` or `fold` operation is successfully realized over every node defined by its target. The latter represents the case when the last node is reached in the data. In both cases, the $\circlearrowleft$ operator computes the result to be placed to the result store:

$$\circlearrowleft \ell \mapsto \omega \triangleq \begin{cases} \ell \xmapsto{KLV} 0 & \text{if } \omega = \texttt{map}\ f\ KLV \\ \ell \xmapsto{KLV} v & \text{if } \omega = \texttt{fold}\ f\ v\ KLV \end{cases}$$

A quick case analysis can reveal that each task reduction only involves at most two consecutive stations in the station sequence (PROP), and often one station only (MAP, FOLD, COMPLETE, LAST, or OPT). In other words, both task reductions exhibit *local* behaviors.

3) In-Data Load Reduction. On the backend, the other form of in-data processing is a *load reduction*, enabled by LOAD. Unlike task reductions that process *operations*, load reductions (lazily) process computations in data. What constitutes a *load* is evident by an inspection on the $\mathbb{L}$ evaluation context, whose fulfilling redex we call a *load expression*: (i) the data node inside a station, or (ii) the initial expression argument of a `fold` operation in the streamlet.

As revealed by LOAD, a load reduction depends on a frontend reduction: the premise of the rule is a reduction over a configuration whose backend and top-level operation stream are both set to $\{\}$, *de facto* only allowing for a frontend reduction. Intuitively, this means we consider every load expression forms its own runtime with a trivial configuration that has no backend data or operation stream. This simplifies our definition because a load reduction can thus depend on a BETA, NODE, or CLAIM reduction, effectively allowing the reductions they represent to happen at the backend of data processing. The last case is especially important, in that it enables a dependent operation to claim its argument in the form of a future, while processing at the backend (recall § 2.2.1).

Before we move on, let us illustrate the behavior of task and load reductions, especially on how a propagation step, a realization step, and a load reduction step interleave with each other, through an example:

Example 4.1 (Incremental Folding). Consider a configuration where the backend consists of two stations, with nodes N_1 and N_2, and a `fold` operation has been propagated to the first station. The operation has a folding function f representing a function which sums up the payloads of all target nodes (this is a simplified version of the COREPR example), and a target key list of $\mathbf{KL}\langle[k_1, k_2]\rangle$. The following is one reduction sequence which ends in the `fold` being completed, where $N_i = \mathbf{N}\langle k_i; i; \mathbf{KL}\langle[]\rangle\rangle$ for $i = 0, 1, 2$ and $N_0' = \langle k_0; 3; \mathbf{KL}\langle[]\rangle\rangle$:

$$\begin{array}{rlrl}
 & \langle[\langle N_1; [[\ell \mapsto \texttt{fold}\ f\ N_0\ \mathbf{KL}\langle[k_1, k_2]\rangle]]\rangle, & \langle N_2; []\rangle]; []; & \{\}; e\rangle \\
(\textsc{Fold}) \rightarrow & \langle[\langle N_1; [[\ell \mapsto \texttt{fold}\ f\ (f\ N_1\ N_0)\ \mathbf{KL}\langle[k_2]\rangle]]\rangle, & \langle N_2; []\rangle]; []; & \{\}; e\rangle \\
(\textsc{Prop}) \rightarrow & \langle[\langle N_1; []\rangle, & \langle N_2; [[\ell \mapsto \texttt{fold}\ f\ (f\ N_1\ N_0)\ \mathbf{KL}\langle[k_2]\rangle]]\rangle]; []; & \{\}; e\rangle \\
(\textsc{Fold}) \rightarrow & \langle[\langle N_1; []\rangle, & \langle N_2; [[\ell \mapsto \texttt{fold}\ f\ (f\ N_2\ (f\ N_1\ N_0))\ \mathbf{KL}\langle[]\rangle]]\rangle]; []; & \{\}; e\rangle \\
(\textsc{Load}) \rightarrow^* & \langle[\langle N_1; []\rangle, & \langle N_2; [[\ell \mapsto \texttt{fold}\ f\ N_0'\ \mathbf{KL}\langle[]\rangle]]\rangle]; []; & \{\}; e\rangle \\
(\textsc{Last}) \rightarrow & \langle[\langle N_1; []\rangle, & \langle N_2; []\rangle]; []; & \{\ell \xmapsto{\mathbf{KL}\langle[]\rangle} N_0'\}; e\rangle
\end{array}$$

4) To-Data Reduction. The three rules that capture the behavior at the boundary of the top-level operation stream and the data are simple. EMPTY considers the bootstrapping case where the data so far contains no nodes. If the operation is a `map` or `fold` operation, a result is immediately returned.

TLO-Batch	$[U, U'] \rightsquigarrow [U +\!\!+ U'], \{\}$	
TLO-Unbatch	$[U_1 +\!\!+ U_2] \rightsquigarrow [U_1, U_2], \{\}$	if $U_i \neq []$ for $i = 1, 2$
TLO-ReorderD	$[[\ell_1 \mapsto o_1], [\ell_2 \mapsto o_2]] \rightsquigarrow [[\ell_2 \mapsto o_2], [\ell_1 \mapsto o_1]], \{\}$	if $\odot(o_1) \cap \odot(o_2) = \{\}$
TLO-ReorderRR	$[[\ell_1 \mapsto o_1], [\ell_2 \mapsto o_2]] \rightsquigarrow [[\ell_2 \mapsto o_2], [\ell_1 \mapsto o_1]], \{\}$	if $o_i = \texttt{fold}\ f_i\ e_i\ \mathit{KLV}_i$ for $i = 1, 2$
TLO-ReorderRW	$[[\ell_1 \mapsto \texttt{map}\ f_1\ \mathbf{KL}\langle \overrightarrow{k}_1 \rangle], [\ell_2 \mapsto \texttt{fold}\ f_2\ e\ \mathit{KLV}_2]] \rightsquigarrow [[\ell_2 \mapsto \texttt{fold}\ (f_2 \overset{\overline{k_1}}{\circ\circ} f_1)\ e\ \mathit{KLV}_2], [\ell_1 \mapsto \texttt{map}\ f_1\ \mathbf{KL}\langle \overrightarrow{k}_1 \rangle]], \{\}$	

Fig. 12. Selected Rules of Temporal Locality Optimization (A complete definition can be found in the supplementary material.)

First removes the head element from the top-level operation stream, and places it as the last element of the streamlet associated with the first node.

According to Add, a new node is created with a freshly generated key. In DON Calculus we adopt a simple design for node addition: they are always placed at the beginning of the data station sequence. This can be seen in Add. It also explains why an add reduction is a to-data reduction not an in-data one.

4.2 Temporal Locality Optimization

The Opt rule bridges the main reduction relation ($\longrightarrow$) with the $\rightsquigarrow$ relation, which defines different forms of temporal locality optimization. With selected rules defined in Fig. 12, the $O \rightsquigarrow O', R$ relation says that operation stream O reduces to operation stream O' in one step, while producing result R.

TLO-Batch and TLO-Unbatch allow units in the in-data operation streams to be batched and unbatched. As the Opt rule can be applied over the streamlet in any station, batching and unbatching may happen in-data at an arbitrary station. The reader may notice that many task reduction rules, such as Map and Fold, are defined with a singleton stream unit (batch). This is because any batched stream unit can be unbatched first via TLO-Unbatch, realized, and then batched again via TLO-Batch for further propagation.

Reordering is supported by three rules. TLO-ReorderD says that two operations with disjoint target key lists can be reordered in the operation stream.

Example 4.2 (Operation Reordering). Imagine we have two operations that target disjoint keys: $\ell \mapsto \texttt{map}\ e\ \mathbf{KL}\langle [k_1, k_2] \rangle$ and $\ell' \mapsto \texttt{map}\ e'\ \mathbf{KL}\langle [k_3, k_4] \rangle$. According to TLO-ReorderD, they may be swapped.

TLO-ReorderRR says that two fold operations can be reordered, as both are "read" in nature. Finally, TLO-ReorderRW shows a map operation and a fold operation may still be reordered even if they have overlapping targets. The insight behind is that a fold can "skip ahead" of a map if the former alters its folding function as applying the mapping function of the latter first. This rule relies on a helper operator for composing a mapping function and a folding function together, where $f \overset{\overline{k}}{\circ\circ} f'$ is defined as $\lambda x.\lambda y.f\ (\mathbf{if}\ {}^1x \in \overline{k}\ \mathbf{then}\ f'\ x\ \mathbf{else}\ x)\ y$.

To speed up the narrative, we defer the specification on fusion and reuse to the supplementary material. Despite the diversity of TLOs — from batching, reordering, fusing, to reusing — the principle here is that they all rewrite on the operation stream before the operations are realized. The insight revealed by DON Calculus is that they may all happen *in data* (see § 2.5) thanks to the fact that Opt can be applied in any streamlet.

$$
\begin{array}{lll}
\tau ::= \mathsf{int} \mid \mathsf{key} \mid \mathsf{future}[\tau] \mid \mathsf{kl} \mid \mathsf{node} \mid \tau \xrightarrow{\varepsilon} \tau & & \textit{type} \\
\varepsilon ::= \mathsf{T} \mid \mathsf{F} & & \textit{emittability} \\
\Gamma ::= \overrightarrow{x : \tau} & & \textit{typing environment}
\end{array}
$$

$$
\text{T-ADD}\ \frac{\Gamma \vdash e : \mathsf{int} \setminus \varepsilon}{\Gamma \vdash\ \mathsf{add}\ e : \mathsf{future}[\mathsf{key}] \setminus \mathsf{T}}
\qquad
\text{T-MAP}\ \frac{\Gamma \vdash e : \mathsf{node} \xrightarrow{\mathsf{F}} \mathsf{node} \setminus \varepsilon \qquad \Gamma \vdash e' : \mathsf{kl} \setminus \varepsilon'}{\Gamma \vdash\ \mathsf{map}\ e\ e' : \mathsf{future}[\mathsf{int}] \setminus \mathsf{T}}
$$

$$
\text{T-FOLD}\ \frac{\Gamma \vdash e : \mathsf{node} \xrightarrow{\mathsf{F}} \mathsf{node} \xrightarrow{\mathsf{F}} \mathsf{node} \setminus \varepsilon \qquad \Gamma \vdash e' : \mathsf{node} \setminus \varepsilon' \qquad \Gamma \vdash e'' : \mathsf{kl} \setminus \varepsilon''}{\Gamma \vdash\ \mathtt{fold}\ e\ e'\ e'' : \mathsf{future}[\mathsf{node}] \setminus \mathsf{T}}
\qquad
\text{T-ABS}\ \frac{\Gamma +\!\!+ [x : \tau] \vdash e : \tau' \setminus \varepsilon}{\Gamma \vdash \lambda x : \tau.e : \tau \xrightarrow{\varepsilon} \tau' \setminus \mathsf{F}}
$$

$$
\text{T-APP}\ \frac{\Gamma \vdash e : \tau \xrightarrow{\varepsilon} \tau' \setminus \varepsilon' \qquad \Gamma \vdash e' : \tau \setminus \varepsilon''}{\Gamma \vdash e\ e' : \tau' \setminus (\varepsilon \vee \varepsilon' \vee \varepsilon'')}
$$

Fig. 13. Selected Rules of the DON Calculus Type System (A complete definition can be found in the supplementary material.)

5 THE TYPE SYSTEM

Fig. 13 defines a type system for DON Calculus, where typing judgement $\Gamma \vdash e : \tau \setminus \varepsilon$ says that given typing environment Γ, expression e has type τ with *emittability* ε. Metavariable ε ranges over booleans, where a true value (T) indicates the expression may emit an operation whereas a false value (F) indiates it must not. Operator $\Gamma\{x\}$ is defined as τ where $x' : \tau$ is the right most occurrence in Γ such that $x = x'$.

Types are either a key type key, a payload type int, a key list type kl, a node type node, a future type $\mathsf{future}[\tau]$ where τ is the type of the result represented by the future, or a function type $\tau \xrightarrow{\varepsilon} \tau$. In the last form, emittability $\xrightarrow{\varepsilon}$ is the effect of the function, which we will explain next. When a function has type $\tau \xrightarrow{\mathsf{T}} \tau$, we informally say that the function is *latently emittable*.

5.1 Phase Distinction

The primary goal of the type system is to enforce phase distinction: whereas the evaluation of an expression at the frontend is unrestricted, the evaluation at the backend cannot lead to an operation emission. We establish phase distinction through a simple type-and-effect system. It is built on the insight that an operation might be emitted at the backend if the functions that serve as the arguments of operations were latently emittable. As a result, the key to enforcing phase distinction is to make sure these arguments are not latently emittable. Note that in our type system, both T-MAP and T-FOLD ensure that their argument functions — be it the mapping function or the folding function — have function types that are not latently emittable. Emittability is disjunctive, as shown in rules such as T-APP. On top of a standard type-and-effect core, the main novelty of our type system is the property it enforces: phase distinction is a previously unreported property, yet critical in establishing result determinism.

To revisit Example 2.14, the program does not type check because expression `mapVal g [k]` would violate phase distinction.

5.2 Runtime Typing

Our type system can be implemented either as a static system or a dynamic system. The former is useful with the "closed world" assumption: the entire processing operations are known before the program starts. The latter is more appropriate with the "open world" (see Sec. 3.1), where the forms of operations and their arguments may not be known until run time. The runtime typing

$$
\begin{array}{rcll}
\hat{\mathbb{F}} & ::= & \langle \beta; []; R; \mathbb{E} \rangle & \textit{eager frontend context} \\
\hat{\mathbb{B}} & ::= & \langle \bullet; []; R; e \rangle & \textit{eager backend context} \\
\hat{\mathbb{T}} & ::= & \hat{\mathbb{B}}[\beta \mathbin{+\!\!+} \bullet \mathbin{+\!\!+} \beta] & \textit{eager task context (redex load-free)} \\
\hat{\mathbb{L}} & ::= & \hat{\mathbb{B}}[\beta \mathbin{+\!\!+} \langle \mathbb{E}; O \rangle :: \beta] & \textit{eager load context} \\
& | & \hat{\mathbb{B}}[\beta \mathbin{+\!\!+} \langle \mathit{NV}; [[\ell \mapsto \texttt{fold}\ f\ \mathbb{E}\ \mathit{KLV}]] \rangle :: \beta] & \\
\beta & ::= & \overrightarrow{\langle \mathit{NV}; [] \rangle} & \textit{dry backend}
\end{array}
$$

Fig. 14. Evaluation Context for Eager Processing

rules are a predictable extension of static typing, with additional rules for configuration typing and value typing. The judgment $\Gamma \vdash_{\mathsf{c}} C : \tau \setminus \varepsilon$ says configuration C has type τ with emittability ε under typing environment Γ. We defer these rules to the supplementary material.

6 META-THEORY

We now state important properties for DON Calculus. We say a backend is *dry* if it follows the form $\overrightarrow{\langle \mathit{NV}; [] \rangle}$, written as β. We say a configuration C is well-typed iff $[] \vdash_{\mathsf{c}} C : \tau \setminus \varepsilon$ for some τ and ε. We define function $init(e, B)$ to compute the initial configuration of frontend program e given initial backend B. Specifically, $init(e, B) \triangleq \langle B; []; \{\}; e \rangle$. The function $init(e, B)$ is only defined if $\langle B; []; \{\}; e \rangle$ is well-typed. According to this definition, a program does not have to start with an empty data structure; it can start with a data structure represented by B.

1) Type Soundness.

Lemma 6.1 (Type Preservation). *If $\Gamma \vdash_{\mathsf{c}} C : \tau \setminus \varepsilon$, and $C \longrightarrow C'$ then $\Gamma \vdash_{\mathsf{c}} C' : \tau \setminus \varepsilon'$ where $\varepsilon = \mathsf{F}$ implies $\varepsilon' = \mathsf{F}$.*

Lemma 6.2 (Progress). *For any C which is well-typed, then either $C = \langle \beta; []; R; v \rangle$ for some β and R or there exists some $C' \neq C$ and $C \longrightarrow C'$.*

In this lemma, note that the configuration $\langle \beta; []; R; v \rangle$ has the first component (the backend) as a *dry* backend, the second component (the top-level operation stream) as empty ($[]$), and the fourth component (the expression) as a value. This configuration is intuitively a terminating configuration.

Theorem 6.3 (Soundness). *For any program e and backend B, if $init(e, B) = C$ then either there exists C' such that $C \longrightarrow^* C'$ where $C' = \langle \beta; []; R; v \rangle$ or C diverges.*

This important theorem establishes type soundness. As expected, it does require the initial configuration to be well-typed, because function $init(e, B)$ has the pre-condition that $\langle B; []; \{\}; e \rangle$ is well-typed.

Corollary 6.4 (Phase Distinction). *For any well-typed configuration C, if $C \longrightarrow C'$, then either (1) the reduction is an instance of Emit, or (2) the reduction is not an instance of Emit, and its derivation does not contain an instance of Emit.*

Recall that Emit is defined with the frontend context $\mathbb{F}$. Case (1) says that operation emission may happen at the frontend. On the backend, recall that the only reduction that may contain a subderivation of Emit would be an instance of Load. Case (2) says that such a derivation is not possible. In other words, operation emission cannot happen on the backend. As shown in Example 2.14, the importance of phase distinction is that it contributes to result determinism, which we elaborate next.

2) Result Determinism (Observable Equivalence). With generality as a design goal, DON Calculus is guided with a design rationale that we should place as few restrictions on the evaluation order as possible, leading to a semantics inherent with non-deterministic executions. One example is the non-deterministic redex selection for propagation which we described in § 4. More generally, a simple case analysis of evaluation contexts in Fig. 10 should make clear that DON Calculus is endowed with non-deterministic redex selection between:

- *a frontend reduction and a backend reduction*: given a configuration, either $\mathbb{F}$ or $\mathbb{B}$ can be used for selecting the redex of the next step of reduction;
- *task reductions over different stations*: according to $\mathbb{T}$, the redex can be an arbitrary station in the runtime data, where the task is an instance of Map, Fold, Complete, and Last, or two adjacent stations, where the task is an instance of Prop;
- *load reductions inside different stations*: according to $\mathbb{L}$, the redex can be any load expression inside an arbitrary station;
- *a task reduction and a load reduction*: either $\mathbb{T}$ and $\mathbb{L}$ can be used for redex selection.

Non-deterministic executions are good news for generality and adaptability (see § 2.4), but they are a challenge to correctness: do different reduction sequences from the same configuration produce the same result? We answer this question now.

Lemma 6.5 (Result Confluence). *For any frontend program e and backend B, if $init(e,B) \longrightarrow^* \langle B_1; O_1; R_1; v_1\rangle$ and $init(e,B) \longrightarrow^* \langle B_2; O_2; R_2; v_2\rangle$ then $\forall \ell \in dom(R_1) \cap dom(R_2).R_1(\ell) = R_2(\ell)$.*

In other words, despite the non-deterministic execution exhibited by the asynchronous processing between the frontend and the backend (see § 2.2.1), despite the non-deterministic choices in propagation and realization in the backend (see § 2.4), despite non-deterministic executions over load expressions resulting from lazy realization (see § 2.4), despite the in-data TLO (see § 2.5), all executions that produce a result for an operation will converge on the same result. Taken all operations into account, we can further establish:

Theorem 6.6 (Determinism). *For any frontend program e and backend B, if $init(e,B) \longrightarrow^* \langle \beta_1; []; R_1; v_1\rangle$ and $init(e,B) \longrightarrow^* \langle \beta_2; []; R_2; v_2\rangle$ then $\beta_1 = \beta_2$ and $dom(R_1) = dom(R_2)$ and $\forall \ell \in dom(R_1).R_1(\ell) = R_2(\ell)$ and $v_1 \equiv v_2$.*

According to this theorem, all terminating executions not only produce the same results for operations, but also lead to the same final data structure, and the same values modulo term equivalence in λ calculus. Here, term equivalence is needed because of the TLO rules such as fusing. It is also important to observe this Theorem can only be established with the support of phase distinction. Without it, both the frontend and the backend could emit operations in a non-deterministic, interleaved manner such that the reduction rules could no longer ensure determinism.

Finally, eager data processing (see § 1) can be modeled by redefining evaluation contexts *without altering any reduction rules.* Intuitively, this means that eager data processing is a restrictive instance of DON Calculus. Rigorously, we represent eager processing as the $\xrightarrow{\mathsf{E}}$ reduction relation, defined as identical as the $\longrightarrow$ we introduced in Fig. 4, except that the $\mathbb{F}$, $\mathbb{B}$, $\mathbb{T}$, $\mathbb{L}$ evaluation contexts are replaced with $\hat{\mathbb{F}}$, $\hat{\mathbb{B}}$, $\hat{\mathbb{T}}$, $\hat{\mathbb{L}}$ evaluation contexts in Fig. 14. We use $\xrightarrow{\mathsf{E}}^*$ to represent the reflexive and transitive closure of $\xrightarrow{\mathsf{E}}$. We say a backend B is *load-free* if any load expression in any station in B is a value. For the eager task context $\hat{\mathbb{T}}$, we further require any element in the domain of its fulfillment function to be load-free. A trivial case analysis will reveal $\xrightarrow{\mathsf{E}}$ is deterministic, conforming to our intuition of one-at-a-time processing.

Corollary 6.7 (DON Calculus With Regard to Eager Processing). *For any frontend program e and backend B, if $init(e,B) \xrightarrow{\mathsf{E}}^* \langle \beta_1; []; R_1; v_1\rangle$ and $init(e,B) \longrightarrow^* \langle \beta_2; []; R_2; v_2\rangle$ then $\beta_1 = \beta_2$ and $dom(R_1) = dom(R_2)$ and $\forall \ell \in dom(R_1).R_1(\ell) = R_2(\ell)$ and $v_1 \equiv v_2$.*

The simple corollary however carries an important message: the general, less restrictive data processing of DON Calculus preserves the computation results of conventional data processing. In a nutshell, IOP and TLO are both *sound* optimizations.

7 COQ MECHANIZATION

DON Calculus has been mechanized in Coq. The proofs include all properties of our meta-theory presented in § 6, spanning around 7,000 LOC. In addition to gaining confidence in the correctness of our calculus, the artifact of Coq mechanization may serve as a first-step reference for computer system researchers to rigorously specify and reason about their own systems of online data processing. Determinism in processing results is a fundamental property that transcends the individual designs of online data processing.

The most challenging part of our mechanization is the confluence proof for determinism (Theorem 6.6). Our proof follows the structure of Huet [Huet 1980], with two main properties to establish: (1) the reduction system is locally confluent; (2) local confluence leads to global confluence. The proof relies on Noetherian (well-founded) induction, following Huet.

8 PRACTICAL EXTENSIONS

In this section, we discuss some encodings and higher-level programming idioms, as well as a number of extensions.

8.1 Custom Data Storage

Data processing routinely requires metadata support for optimization purposes, and/or produce intermediate results stored in data. Encoding in-data storage beyond the integer payload is simple. A node with key k, edges KLV, and custom structured payload $cp \in \mathbb{CP}$, can be encoded as $\mathsf{N}\langle k; I(cp); \mathit{KLV}\rangle$ where $I : \mathbb{CP} \mapsto \mathbb{INT}$ is a bijective "integer encoding" function. I^{-1} can compute the custom payload storage from the node integer payload. Given cp is inductive, I is a standard tree compression function. We will see an example in § 8.4.

8.2 Deletion

Edge deletion is straightforward in DON Calculus; see the encoding of `deleteRelationship` in Table 1. In large-scale data processing systems (e.g., [neo 2010]), node deletion is commonly supported through a conceptual "mark-and-sweep": a boolean "in-use" field in each node indicates whether a node is in use (`true`) or deleted (`false`); processing a deletion operation online only implies resetting the field, and all nodes whose "in-use" field is set as `false` is swept offline. In DON Calculus, this "in-use" field can be supported through custom storage (§ 8.1). The deletion operation itself is a simple `map` function that sets the field to `false`. A user-level "map" function can be encoded as a `map` whose mapping function first checks the "in-use" field is `true`; the same applies to a user-level "fold" function.

8.3 Subgraph Computations

Within graph processing, graph algorithms are often defined over *subgraphs*, a neighborhood of nodes logically connected through edges. The algorithm building blocks of subgraph computation are either *pull*-based (e.g., [Wang et al. 2016]) or *push*-based (e.g., [Roy et al. 2013]), or both (e.g., [Shun and Blelloch 2013; Zhang et al. 2015]). For a directed graph where each edge connects from the *source* node to the *destination* node, a pull-based model iterates over destination nodes, and aggregates over in-edges for each of them, whereas a push-based model iterates over source nodes, and scatters over out-edges for each of them [Grossman et al. 2018]. The essence of both models can be encoded with DON Calculus as follows, where f_{agg} and $f_{\mathtt{dist}}$ are the aggregation and distribution functions respectively, n is the initial value for aggregation, and KLV is the keys of dataset for processing:

$$
\begin{aligned}
\texttt{pull}\ f_{\texttt{agg}}\ n\ \mathit{KLV} \triangleq\ & \texttt{foreach}\ w\ \texttt{in}\ \mathit{KLV}\\
& \quad \texttt{let}\ z = \lambda x.\lambda y.\texttt{if}\ (w\ \texttt{in}\ {}^{3}x)\ \texttt{then}\ f_{\texttt{agg}}\ {}^{2}x\ y\ \texttt{else}\ y\ \texttt{in}\\
& \quad \texttt{let}\ u = \texttt{foldval}\ z\ n\ \mathit{KLV}\ \texttt{in}\\
& \quad \texttt{mapVal}\ (\lambda x.u)\ \mathbf{KL}\langle [w] \rangle
\end{aligned}
$$

$$
\begin{aligned}
\texttt{push}\ f_{\texttt{dist}}\ \mathit{KLV} \triangleq\ & \texttt{foreach}\ w\ \texttt{in}\ \mathit{KLV}\\
& \quad \texttt{let}\ z = \texttt{queryNode}\ w\ \texttt{in}\ \texttt{mapVal}(\lambda x.(f_{\texttt{dist}}\ {}^{2}z\ x))\ {}^{3}z
\end{aligned}
$$

Here, the `pull` encoding iterates over each destination node w, aggregates for all its source nodes, and updates the payload of w. Indeed, the CorePR example in essence is pull-based aggregation: Line 9-11 of Fig. 3 has a similar structure. The `push` encoding iterates over each source node w and updates the payloads of all destination nodes, i.e., ${}^{3}z$ in the definition.

Variants of the pull/push are common in think-like-a-vertex graph processing systems, e.g., [Emoto et al. 2016; Gonzalez et al. 2012; Low et al. 2012; Malewicz et al. 2010]. Take the Gather-Apply-Scatter (GAS) model in Powergraph [Gonzalez et al. 2012] for example. The `pull` encoding is analogous to the combination of "Gather" and "Apply", whereas the `push` encoding is analogous to "Scatter."

8.4 Modeling Existing Systems

DON Calculus lays a foundation for rigorously reasoning about online data processing systems. We now use KickStarter [Vora et al. 2017] as an example to sketch our foundational role in helping specify existing experimental systems.

KickStarter is an online graph processing system where queries results are continuously expected while the queries may be interspersed with graph update operations such as edge addition or deletion. One example query is the single-source widest path (SSWP), where each edge is weighted, and continuous queries may be issued to find out the widest path of a node to a common source node. The key metadata in KickStarter tracks the *value dependency* among nodes: each node maintains a set of nodes whose change may impact the query result to that node. DON Calculus can encode the metadata through custom storage (§ 8.1) in the form of $\langle CV; DS; W \rangle$ with each node, where $CV \in \mathbb{INT}$ keeps the current query result, $DS \in \mathbb{KLV}$ is the *value dependency store*, and $W : \mathbb{KEY} \mapsto \mathbb{INT}$ represents edge weights. Intuitively, when a node of key k has a DS where k' appears, it means that the change of node k' may impact the query result for node k. When a node of key k has a W entry that maps k' to n, it means that the weight for the edge connecting k and k' has the weight of n. (One observation made by KickStarter is that DS is often a singleton set for common graph queries; we keep the list representation for generality.) For the rest of the section, we define convenience functions to retrieve the current query result and the value dependency store associated with each node:

$$
\begin{aligned}
\texttt{getV}\ \mathbf{N}\langle k; n; \mathit{KLV} \rangle &\triangleq {}^{1}(I^{-1}(n))\\
\texttt{getD}\ \mathbf{N}\langle k; n; \mathit{KLV} \rangle &\triangleq {}^{2}(I^{-1}(n))
\end{aligned}
$$

KickStarter judiciously determines the need for recomputing the query result. Not to lose generality, we represent recomputation through a higher-order function `recompute`, which takes a function f that can be applied to a node to produce the recomputed result. Just as SSWP and single-source shortest path (SSSP) may have different ways of recomputation, KickStarter allows programmers to provide (i.e., customize) this function f:

$$\texttt{recompute}\ f\ \mathit{NV} \triangleq \langle k; I(f\ \mathit{NV}); \mathit{KLV} \rangle\ \text{where}\ \mathit{NV} = \mathbf{N}\langle k; n; \mathit{KLV} \rangle$$

Here, f can rely on any information in the node NV (e.g., current query result or dependency store) to recompute. With that, we can encode the query function of KickStarter as follows, where $v_{\texttt{uinit}}$ represents the uninitialized value for CV, i.e., before the first query is conducted:

$$\begin{aligned}\texttt{KSQuery}\ k\ f \triangleq\ & \texttt{let}\ y = \lambda x.\ \texttt{if}\ ((\texttt{getV}\ x) == v_{\texttt{uinit}})\ \texttt{recompute}\ f\ x\ \texttt{else}\ x\ \texttt{in}\\ & \texttt{map}\ y\ \mathbf{KL}\langle[k]\rangle; \texttt{getV}\ (\texttt{queryNode}\ k)\end{aligned}$$

The more interesting case is edge deletion, which we encode as follows. Here, keys k_{s} and k_{d} are the source/destination node of the edge to be deleted, KLV is the scope of keys to be inspected (such as a partition, or the entire graph), and f is the custom recomputation function.

$$\begin{aligned}&\texttt{KSDeleteEdge}\ k_{\mathsf{s}}\ k_{\mathsf{d}}\ \mathit{KLV}\ f \triangleq \texttt{deleteRelationship}\ k_{\mathsf{s}}\ k_{\mathsf{d}}; \texttt{trim}\ \mathbf{KL}\langle[k_{\mathsf{s}}]\rangle\ \mathit{KLV}\ f\\ &\text{where}\ \texttt{trim}\ \mathit{KLV}'\ \mathit{KLV}\ f \triangleq \texttt{foreach}\ (w\ \texttt{in}\ \mathit{KLV}')\\ &\qquad\qquad \texttt{let}\ z = \lambda x.\lambda y.\texttt{if}(w\ \texttt{in}\ (\texttt{getD}\ x))\ y \oplus \{^1 x\}\ \texttt{else}\ y\ \texttt{in}\\ &\qquad\qquad \texttt{let}\ u = \texttt{fold}\ z\ \mathbf{KL}\langle[]\rangle\ \mathit{KLV}\ \texttt{in}\\ &\qquad\qquad \texttt{map}\ (\texttt{recompute}\ f)\ u; \texttt{trim}\ u\ \mathit{KLV}\ f\end{aligned}$$

It says that the edge will be deleted from the graph (the `deleteRelationship` operation), and the dependency store needs to be processed through *trimming*. The `trim` function iteratively inspects and updates the dependency stores of nodes that may be impacted by the edge deletion. At each iteration, the `fold` function collects the nodes keys that may subject to recomputation, performed by `map`.

The take-away message is that, with DON Calculus, the KickStarter developers can focus on defining their unique algorithm details (e.g., f for the recomputation of query results and dependencies) while enjoying the correctness properties defined by DON Calculus. This also means that they can reuse the mechanized proofs of DON Calculus, only strengthening them with properties unique to their algorithm (e.g., approximation monotonicity).

8.5 Key-Value Store and Tabular Data Support

Supporting structured data is a design goal of our DON Calculus (see § 2.1). To be inclusive on general data structures such as graphs, the DON Calculus runtime necessarily includes structures such as adjacency lists. Other common data organizations — key-value stores and tabular/relational data — are topologically simpler than graphs; they can also be supported by DON Calculus, i.e., endowing IOP and TLO to the online processing of these forms of data.

Supporting key-value stores with DON Calculus are trivial: the adjacency list for each data node should always be an empty sequence. The most common operations in key-value stores, mapping and aggregation (reduction), have corresponding primitives in our calculus, `map` and `fold`. From this perspective, DON Calculus describes the behavior of online processing of a *dynamic* key-value store where incremental processing and operation batching/reordering/fusion/reuse are in place.

For tabular/relational data, we first need to support multiple tables. This can be encoded as long as we have a bijective mapping between $\mathbb{TABLEID} \times \mathbb{ROWID}$ and $\mathbb{KEY}$ where $\mathbb{TABLEID}$ is the set of table IDs and $\mathbb{ROWID}$ is the set of row IDs. In other words, the backend data structure (B) can always be logically partitioned into multiple tables. The payload associated with each node in this case would be a tuple, each component being the value of a column. For the common relational operations, column projection can be directly supported by `map`, where the mapping function is the tuple elimination indexed at the column of interest. As the `map` operation propagates through the backend data, *incremental* column projection is supported for free. The SQL-style `GROUP BY` operator can be supported in a similar fashion, except the result is a mapping whose domain constitute the column values of interest identified by the `GROUP BY` operator. As this operator is often used for aggregation. the aggregation function can be performed incrementally similar to the incremental fold example (Example 4.1).

8.6 Sublinear Operation Processing

Indexing and hashing are two examples where processing an operation may become *sublinear* in time complexity: through auxiliary structures (e.g., indexes and hashes), an operation may circumvent the scan and traversal in data.

For *immutable* data, DON Calculus can be trivially extended with indexing and hashing. Since no update is allowed, this is analogous to a subset of DON Calculus without `add` and `map` expressions. Here, a simple query (e.g., a key-value lookup) can be directly answered by the index/hash, while more complex queries (e.g., a folding operation that involves many nodes) continue to follow the same semantics currently defined by DON Calculus. Note that the use scenario of *immutable* data processing is indeed where indexing and hashing are most common (e.g., in Spark).

For *mutable* (i.e., evolving) data processing, extending DON Calculus with indexing and hashing requires one consideration: the result from index-based or hash-based query should be "corrected" by the updates that are under propagation (i.e., the updates that are emitted but not realized). The notion of "correction" is analogous to a TLO optimization that reorders a `map` operation and a `fold` operation; see TLO-ReorderRW. Orthogonal to the DON Calculus support, readers should be aware that indexing/hashing support in *mutable* data processing by itself is often problematic in practical systems (e.g., modern databases [neo 2010]) and hence less commonly used. The general practice is to leave the correctness of using indexing or hashing to the programmer: she can create an index to her very large and mutable graph, but the potentially expensive reindexing in the presence of data change is a programmer task. As a result, no guarantee is provided at the level of the data processing engine that the index-based query returns a correct (i.e., non-stale) result. In this context, the DON Calculus variant we discussed above provides the correctness guarantee *up to* the program. In other words, this variant can ensure a non-deterministic execution can produce the same result as that of eager processing of the program (Corollary 6.7).

8.7 Exception Handling

Recall that in § 3.2, we described the residual target key list associated with each entry in the result store. In a language extension with explicit exception handling support, modeling "key not found" as an exception is a simple extension. The only change is to replace Claim with the following rules:

$$\textsc{ClaimY}\ \frac{\mathbb{F}[\Downarrow \ell] = \langle B; O; R; e\rangle \qquad \ell \xmapsto{\mathbf{KL}\langle[\,]\rangle} v \in R}{\mathbb{F}[\Downarrow \ell] \to \mathbb{F}[v]} \qquad \textsc{ClaimN}\ \frac{\mathbb{F}[\Downarrow \ell] = \langle B; O; R; e\rangle \qquad \ell \xmapsto{\mathit{KLV}} v \in R \qquad \mathit{KLV} \neq \mathbf{KL}\langle[\,]\rangle}{\mathbb{F}[\Downarrow \ell] \to \mathbb{F}[\texttt{exception}(\mathit{KLV})]}$$

where $\texttt{exception}(\mathit{KLV})$ is a value of this extended language. A programmer can further inspect KLV for exception handling.

8.8 More Extensions

In the supplementary material, we further describe the support of additional features, including parallelism, mapping/folding all elements, and alternative design choices for node addition.

8.9 Applicability and Limitations

In summary, DON Calculus is best suited for specifying systems or applications that can be expressed as continuously submitting queries (reads) and updates (writes) to an evolving piece of data. In other words, a beneficiary data processing system/application should (1) have a natural data-centric view, i.e., a piece of dynamic data structure evolves as the program progresses; (2) have operations continuously applied to the data.

One limitation of our calculus is its fundamental incompleteness, i.e., there are always optimizations in existing/future online data processing systems out of scope of our calculus. Nonetheless, we think IOP and TLO are arguably the most common forms of optimization relevant to the *online* requirements of data processing. For optimizations beyond IOP and TLO, the most important family beyond (the main text of) this paper is perhaps parallelism.

Our core DON Calculus assumes data are scanned or traversed when an operation is processed. For extending our calculus with alternative data access such as indexing and hashing, see § 8.6.

8.10 An Implementation

The design of DON Calculus has inspired us to develop PitStop [Eymer et al. 2022], an online processing system for graph databases. PitStop targets the use scenario described in § 2.2.1. It supports IOP features (in the same style as Example 2.6) [Eymer et al. 2019] and a subset of TLO (batching and fusion). The implementation details of this system are out of the scope of this paper, but we wish to describe the relationship between DON Calculus and PitStop. First, DON Calculus provides a foundation to confirm the *correctness* claims made for PitStop, especially determinism. Second, PitStop confirms the *performance* benefits of IOP and TLO in the context of graph databases: it shows that workload fluctuation and longtail — two challenging scenarios of online data processing — can benefit from them. Third, PitStop also implemented features beyond the scope of DON Calculus, e.g., fine-grained parallelism. A parallel variant of DON Calculus can be found in the supplementary material.

9 RELATED WORK

Incrementality. Self-adjusting computation [Acar et al. 2006] enables computations to respond to dynamically changing (input) data automatically. It tracks the control/data dependencies in a computation so that changes to data can be propagated through the computation. DON Calculus explores a use scenario where data respond to a stream of operations, and the propagation appears in the data itself. With i3QL [Mitschke et al. 2014], incremental computations can be specified and maintained in a declarative SQL-like language, embedded in Scala. A foundation for fault-tolerant distributed computing [Haller et al. 2018] describes a formal semantics and lineage-based programming model for distributed data processing. In their model, deferred evaluation is supported at the boundary of distributed nodes to promote opportunities for operation fusion and improve the efficiency of network communications. More broadly, incremental computing systems [Hammer et al. 2014; Harkes et al. 2016; Harkes and Visser 2017; Pugh and Teitelbaum 1989] propagate changes in the *program* dependency graph, and efficiently perform re-computation along the propagation path only when necessary.

Temporal Locality Optimization. In databases, the various forms of TLOs formalized by DON Calculus are well known. Batching is a basic operation supported by numerous systems. QUEL* [Sellis and Shapiro 1985] is an early compiler optimization defined with a number of tactics for inter-query optimization, such as combining two REPLACE operations in a relational query language into one. This is analogous to fusing in the style of the TLO-FuseM rule in DON Calculus.

Database queries can be optimized so that common tasks can be shared [Sellis 1988], and this problem can also be formulated as a sub-expression identification problem [Park and Segev 1988]. These pioneer efforts lead to a large body of research on MQO-style query optimization (e.g., [Le et al. 2012; Ramachandra and Sudarshan 2012; Ren and Wang 2016; Scully and Chlipala 2017; Sousa et al. 2014]). The essence of exploring commonality among queries is embodied by the TLO-Reuse rule in DON Calculus.

Overall, the relationship between existing work and DON Calculus is complementary. Existing work highlights the importance of TLO in data processing design and provides the context for our calculus. DON Calculus provides a language-based foundation where TLOs are specified as a part

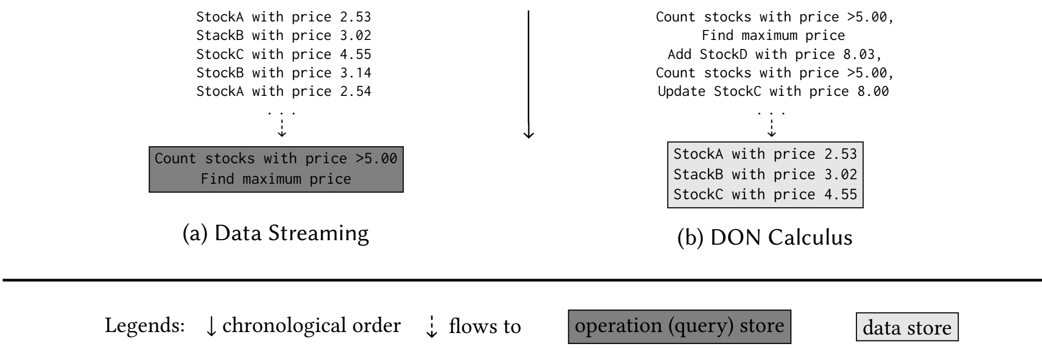


Fig. 15. Data Streams and Operation Streams: Different Scenarios in Stock Data Processing

of the semantics of a data processing engine, and various TLOs are unified in one system. It also elucidates *when* and *where* TLOs may happen (§ 2.5).

Data Streaming. Data streaming systems have a model where a stream of *data* flow through data processing operations (often called stream processors) composed together through framework-defined combinators. This is a well established area, including data flow and data streaming languages [Ashcroft and Wadge 1977; Caspi et al. 1987; Meyerovich et al. 2009; Spring et al. 2007; Thies et al. 2002; Vaziri et al. 2014], data flow processing frameworks [Hirzel et al. 2014; Murray et al. 2013, 2011; Zaharia et al. 2013, 2016], and foundations [Arasu and Widom 2004; Bartenstein and Liu 2014; Cohen et al. 2006; Gurevich et al. 2007; Haller and Miller 2019; Lee and Messerschmitt 1987; Soulé et al. 2010].

As we described in § 2.1, DON Calculus explores a near dual design space. To help understand the fundamental semantic and use scenario difference between existing work and ours, let us refer to an example frequently used in data streaming systems, real-time stock data processing. As shown in Fig. 15(a), a data streaming system is designed for a use scenario where a live stream of *data* may be processed by a pre-deployed query (or queries) — *continuous queries* [Arasu and Widom 2004] — e.g., continuously finding out what the maximum stock price is. DON Calculus is designed for a different use scenario where a live stream of *operations*, as shown in Fig. 15(b), may be applied to a continuously evolving data store. The different use scenarios each direction targets lead to different design needs. For example, TLO is an essential design component in DON Calculus, where we answer e.g., how to reorder operation "`Count stocks with price >5.00`" and operation "`Update StockC with price 8.00`" with both operations still returning the expected results. There appears to be no natural analogy for reordering in a data streaming system. In that setting, more commonly known is data aggregation, such as through a sliding window [Tangwongsan et al. 2015]. In essence, the design space of a data streaming systems addresses how to apply a sequence of data to a program, whereas the design space of DON Calculus addresses how to apply a sequence of programs to an evolving set (or structure) of data.

From an end-user perspective, the choice between operation streams and data streams depends on the application use scenario. In data streaming, new data are emitted continuously, but the queries themselves — such as those at the bottom right of Fig. 15 — are relatively stable; they do not go through rapid changes at run time and are often deployed ahead of time. In contrast, the operations in the operation streams are emitted continuously, and their emission (from a frontend program) is dynamic, not known *a priori*. With operation streams, new data can indeed be added or updated — through `add` and `map` operations in DON Calculus — but the natural use scenario is that these additions/updates of data are mixed with dynamically emitted and diverse queries.

Online Data Processing Systems. The need for scalable online data processing is long sought after. In the naive sense (see § 1), any data processing system — a database or a graph analytic engine — can be viewed "online" if deployed in an interactive setting. In recent years however, the explosive growth in data volume and the complexity of analytical queries/updates together redefine its essence, so that any system that can be justifiably termed "online" must embrace optimizations to support continuous, low-latency, and sometimes real-time processing. In databases, one example is Online Analytical Processing (OLAP) databases. For data processing frameworks that are primarily deployed with immutable datasets, such as MapReduce and Spark, the scalability demands are often met with scale-out solutions, as data parallelism can be effective. The same holds for early graph processing systems (e.g., [Gonzalez et al. 2012; Low et al. 2012; Shun and Blelloch 2013]) where static graphs are assumed. For newer graph processing systems, IOP and TLO both play significant roles. For example, GraPU [Sheng et al. 2018] allows updates to the graph to be buffered and pre-processed, similar to a TLO operation in our top-level operation stream. Kineograph [Cheng et al. 2012] supports a commit protocol for incremental graph updates. DeltaGraph [Dexter et al. 2016] allows for incremental propagation of graph operations, which can be batched and fused within the graph through a Haskell datatype representation of an inductive graph. C-Trees [Dhulipala et al. 2019] are purely functional data structures to enable efficient concurrent processing in the presence of queries and updates. In addition to KickStarter, other examples that target online data processing include LazyBase [Cipar et al. 2012], Chronos [Han et al. 2014], Tornado [Shi et al. 2016], Version Traveler [Ju et al. 2016], GraphBolt [Mariappan and Vora 2019], GraphOne [Kumar and Huang 2020], GraphQ [Wang et al. 2015], and DZig [Mariappan et al. 2021].

Together, the experimental systems in this subsection provide a context that DON Calculus lays a foundation for, answering the crucial question of correctness in the presence of IOP and TLO.

Phase Distinction. Broadly speaking, phase distinction in type system design can be traced to Cardelli [Cardelli 1988], where a phased type system distinguishes compile-time terms and run-time terms. Harper et al. [Harper et al. 1989] defines phase distinction in the context of ML modules. In meta-programming, macro systems, and multi-stage programming, a crucial concern is to ensure the code generated at run time remains type-safe. This leads to a rich set of language and type system designs where some notion of phase distinction is enforced. Several examples include cross-stage safety and persistence in MetaML [Taha and Sheard 1997] and MetaOCaml [Calcagno et al. 2003], process separation in <ML> [Liu et al. 2009], and cross-stage distinction in Scala multi-stage macros [Stucki et al. 2021]. In DON Calculus, the property of phase distinction is specific to data processing, with the phases being the front-end computation and the back-end computation respectively.

10 CONCLUDING REMARKS

Designing online processing systems with optimization support of temporal locality optimization and incremental operation processing is a challenging problem. DON Calculus illuminates the design space of these systems, and complements experimental systems with a correctness-driven approach. The specification and mechanization of DON Calculus can be used as a sound base by future designers of online data processing systems in their pursuit of rigorous semantic engineering.

Data Availability Statement. The Coq mechanization is publicly available [Dexter et al. 2022].

Acknowledgments. This work is sponsored by the US National Science Foundation, award NSF CCF-1815949.

REFERENCES

2010. Neo4j Graph Database. http://www.neo4j.org.

Umut A. Acar, Guy E. Blelloch, and Robert Harper. 2006. Adaptive Functional Programming. *ACM Trans. Program. Lang. Syst.* 28, 6 (Nov. 2006). https://doi.org/10.1145/1186632.1186634

Andrew W. Appel, Lennart Beringer, Adam Chlipala, Benjamin C. Pierce, Zhong Shao, Stephanie Weirich, and Steve Zdancewic. 2016. The DeepSpec Project: The Science of Deep Specification, https://deepspec.org/, 2016-2021.

Arvind Arasu and Jennifer Widom. 2004. A Denotational Semantics for Continuous Queries over Streams and Relations. *SIGMOD Rec.* 33, 3 (sep 2004), 6–11. https://doi.org/10.1145/1031570.1031572

E. A. Ashcroft and W. W. Wadge. 1977. Lucid, a nonprocedural language with iteration. *Commun. ACM* 20, 7 (July 1977), 8 pages. https://doi.org/10.1145/359636.359715

Thomas W. Bartenstein and Yu David Liu. 2014. Rate Types for Stream Programs. In *OOPSLA'14* (Portland, Oregon, USA) *(OOPSLA '14)*. 213–232. https://doi.org/10.1145/2660193.2660225

Dimitri P Bertsekas and John N Tsitsiklis. 1989. *Parallel and distributed computation: numerical methods*. Vol. 23. Prentice hall Englewood Cliffs, NJ.

Sergey Brin and Lawrence Page. 1998. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Computer Networks* 30 (1998), 107–117. https://doi.org/10.1016/S0169-7552(98)00110-X

Peter Buneman, Susan Davidson, Gerd Hillebrand, and Dan Suciu. 1996. A query language and optimization techniques for unstructured data. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*. 505–516. https://doi.org/10.1145/233269.233368

Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. 2003. Implementing Multi-Stage Languages Using ASTs, Gensym, and Reflection. In *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering* (Erfurt, Germany) *(GPCE '03)*. Springer-Verlag, 57–76. https://doi.org/10.1007/978-3-540-39815-8_4

Luca Cardelli. 1988. *Phase Distinctions in Type Theory.* Technical Report.

P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. 1987. LUSTRE: a declarative language for real-time programming. In *POPL '87* (Munich, West Germany). 178–188. https://doi.org/10.1145/41625.41641

Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. 2012. Kineograph: Taking the Pulse of a Fast-changing and Connected World. In *Proceedings of the 7th ACM European Conference on Computer Systems* (Bern, Switzerland) *(EuroSys '12)*. ACM, New York, NY, USA, 85–98. https://doi.org/10.1145/2168836.2168846

James Cipar, Greg Ganger, Kimberly Keeton, Charles B. Morrey, Craig A.N. Soules, and Alistair Veitch. 2012. LazyBase: Trading Freshness for Performance in a Scalable Database. In *Proceedings of the 7th ACM European Conference on Computer Systems* (Bern, Switzerland) *(EuroSys '12)*. Association for Computing Machinery, New York, NY, USA, 169–182. https://doi.org/10.1145/2168836.2168854

Albert Cohen, Marc Duranton, Christine Eisenbeis, Claire Pagetti, Florence Plateau, and Marc Pouzet. 2006. N-Synchronous Kahn Networks: A Relaxed Model of Synchrony for Real-Time Systems. In *POPL'06* (Charleston, South Carolina, USA) *(POPL '06)*. 180–193. https://doi.org/10.1145/1111320.1111054

Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. San Francisco, CA, 137–150. https://doi.org/10.1145/1327452.1327492

Philip Dexter, Yu David Liu, and Kenneth Chiu. 2016. Lazy graph processing in Haskell. In *Proceedings of the 9th International Symposium on Haskell*. ACM, 182–192. https://doi.org/10.1145/3241625.2976014

Philip Dexter, Yu David Liu, and Kenneth Chiu. 2022. *The Essence of Online Data Processing - Coq Mechanization.* https://doi.org/10.5281/zenodo.7051651

Laxman Dhulipala, Guy E Blelloch, and Julian Shun. 2019. Low-latency graph streaming using compressed purely-functional trees. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 918–934. https://doi.org/10.1145/3314221.3314598

David Ediger, Rob McColl, Jason Riedy, and David A Bader. 2012. Stinger: High performance data structure for streaming graphs. In *2012 IEEE Conference on High Performance Extreme Computing*. IEEE, 1–5. https://doi.org/10.1109/HPEC.2012.6408680

Marwa Elteir, Heshan Lin, and Wu-chun Feng. 2010. Enhancing mapreduce via asynchronous data processing. In *2010 IEEE 16th International Conference on Parallel and Distributed Systems*. IEEE, 397–405. https://doi.org/10.1109/ICPADS.2010.116

Kento Emoto, Kiminori Matsuzaki, Zhenjiang Hu, Akimasa Morihata, and Hideya Iwasaki. 2016. Think like a Vertex, Behave like a Function! A Functional DSL for Vertex-Centric Big Graph Processing. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (Nara, Japan) *(ICFP 2016)*. Association for Computing Machinery, New York, NY, USA, 200–213. https://doi.org/10.1145/2951913.2951938

Jeffrey Eymer, Philip Dexter, and Yu David Liu. 2019. Toward Lazy Evaluation in a Graph Database. In *The Second Workshop on Incremental Computing (IC'19)*.

Jeff Eymer, Philip Dexter, Joseph Raskind, and Yu David Liu. 2022. The PitStop System, online at https://github.com/PitStop-Github/PitStop.

Cormac Flanagan and Matthias Felleisen. 1995. The Semantics of Future and Its Use in Program Optimizations. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, Ron K. Cytron and Peter Lee (Eds.). ACM Press, 209–220. https://doi.org/10.1145/199448.199484

Cormac Flanagan and Matthias Felleisen. 1999. The Semantics of Future and an Application. *J. Funct. Program.* 9, 1 (1999), 1–31. http://journals.cambridge.org/action/displayAbstract?aid=44231

Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX, Hollywood, CA, 17–30.

Samuel Grossman, Heiner Litz, and Christos Kozyrakis. 2018. Making Pull-Based Graph Processing Performant. *SIGPLAN Not.* 53, 1 (Feb. 2018), 246–260. https://doi.org/10.1145/3200691.3178506

Yuri Gurevich, Dirk Leinders, and Jan Van Den Bussche. 2007. A Theory of Stream Queries. In *Proceedings of the 11th International Conference on Database Programming Languages* (Vienna, Austria) *(DBPL'07)*. Springer-Verlag, Berlin, Heidelberg, 153–168. https://doi.org/10.1007/978-3-540-75987-4_11

Philipp Haller and Heather Miller. 2019. A reduction semantics for direct-style asynchronous observables. *J. Log. Algebraic Methods Program.* 105 (2019), 75–111. https://doi.org/10.1016/j.jlamp.2019.03.002

Philipp Haller, Heather Miller, and Normen Müller. 2018. A programming model and foundation for lineage-based distributed computation. *JFP* 28 (2018). https://doi.org/10.1017/S0956796818000035

Robert H. Halstead, Jr. 1985. MULTILISP: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.* 7 (10 1985), 501–538. Issue 4. https://doi.org/10.1145/4472.4478

Matthew A. Hammer, Khoo Yit Phang, Michael Hicks, and Jeffrey S. Foster. 2014. Adapton: Composable, Demand-driven Incremental Computation. In *PLDI '14*. https://doi.org/10.1145/2666356.2594324

Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. 2014. Chronos: a graph engine for temporal graph analysis. In *Proceedings of the Ninth European Conference on Computer Systems*. 1–14. https://doi.org/10.1145/2592798.2592799

Daco C Harkes, Danny M Groenewegen, and Eelco Visser. 2016. IceDust: Incremental and Eventual Computation of Derived Values. In *ECOOP '16*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ECOOP.2016.11

Daco C Harkes and Eelco Visser. 2017. IceDust 2: Derived Bidirectional Relations and Calculation Strategy Composition. In *31st European Conference on Object-Oriented Programming*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ECOOP.2017.14

Robert Harper, John C. Mitchell, and Eugenio Moggi. 1989. Higher-Order Modules and the Phase Distinction. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '90)*. 341–354. https://doi.org/10.1145/96709.96744

Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. 2014. A Catalog of Stream Processing Optimizations. *ACM Comput. Surv.* 46, 4, Article 46 (mar 2014), 34 pages. https://doi.org/10.1145/2528412

Gérard Huet. 1980. Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems: Abstract Properties and Applications to Term Rewriting Systems. *J. ACM* 27, 4 (Oct. 1980), 797–821.

Xiaoen Ju, Dan Williams, Hani Jamjoom, and Kang G. Shin. 2016. Version Traveler: Fast and Memory-Efficient Version Switching in Graph Processing Systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 523–536.

Pradeep Kumar and H. Howie Huang. 2020. GraphOne: A Data Store for Real-Time Analytics on Evolving Graphs. *ACM Trans. Storage* 15, 4, Article 29 (Jan. 2020), 40 pages. https://doi.org/10.1145/3364180

W. Le, A. Kementsietsidis, S. Duan, and F. Li. 2012. Scalable Multi-query Optimization for SPARQL. In *2012 IEEE 28th International Conference on Data Engineering*. 666–677. https://doi.org/10.1109/ICDE.2012.37

E. A. Lee and D. G. Messerschmitt. 1987. Synchronous data flow. *Proc. IEEE* 75, 9 (1987), 1235–1245. https://doi.org/10.1109/PROC.1987.13876

Yu David Liu, Christian Skalka, and Scott F. Smith. 2009. Type-specialized staged programming with process separation. *Higher-Order and Symbolic Computation* 24 (2009), 341–385. https://doi.org/10.1145/1596614.1596622

Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. 2012. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment* 5, 8 (2012), 716–727. https://doi.org/10.14778/2212351.2212354

Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-scale Graph Processing. In *SIGMOD '10* (Indianapolis, Indiana, USA). 135–146. https://doi.org/10.1145/1807167.1807184

Mugilan Mariappan, Joanna Che, and Keval Vora. 2021. DZiG: Sparsity-Aware Incremental Processing of Streaming Graphs. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 83–98. https://doi.org/10.1145/3447786.3456230

Mugilan Mariappan and Keval Vora. 2019. GraphBolt: Dependency-Driven Synchronous Processing of Streaming Graphs. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 25, 16 pages. https://doi.org/10.1145/3302424.3303974

Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. 2009. Flapjax: A Programming Language for Ajax Applications. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications* (Orlando, Florida, USA) *(OOPSLA '09)*. ACM, New York, NY, USA, 1–20. https://doi.org/10.1145/1640089.1640091

Ralf Mitschke, Sebastian Erdweg, Mirko Köhler, Mira Mezini, and Guido Salvaneschi. 2014. I3QL: Language-Integrated Live Data Views. In *OOPSLA'14*. 417–432. https://doi.org/10.1145/2660193.2660242

Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 439–455. https://doi.org/10.1145/2517349.2522738

Derek G Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. 2011. CIEL: a universal execution engine for distributed data-flow computing. In *Proc. 8th ACM/USENIX Symposium on Networked Systems Design and Implementation*. 113–126.

Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. 2015. Making Sense of Performance in Data Analytics Frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 293–307.

Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. 1995. Object exchange across heterogeneous information sources. In *Proceedings of the eleventh international conference on data engineering*. IEEE, 251–260. https://doi.org/10.1109/ICDE.1995.380386

J. Park and A. Segev. 1988. Using common subexpressions to optimize multiple queries. In *Proceedings. Fourth International Conference on Data Engineering*. 311–319. https://doi.org/10.1109/ICDE.1988.105474

W. Pugh and T. Teitelbaum. 1989. Incremental Computation via Function Caching. In *POPL '89*. https://doi.org/10.1145/75277.75305

Karthik Ramachandra and S. Sudarshan. 2012. Holistic Optimization by Prefetching Query Results. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. 133–144.

Xuguang Ren and Junhu Wang. 2016. Multi-Query Optimization for Subgraph Isomorphism Search. *Proc. VLDB Endow.* 10, 3 (Nov. 2016), 121–132. https://doi.org/10.14778/3021924.3021929

Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-stream: Edge-centric graph processing using streaming partitions. In *SOSP '13*. ACM, 472–488. https://doi.org/10.1145/2517349.2522740

Ziv Scully and Adam Chlipala. 2017. A Program Optimization for Automatic Database Result Caching. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017)*. 271–284. https://doi.org/10.1145/3009837.3009891

Timos K. Sellis. 1988. Multiple-Query Optimization. *ACM Trans. Database Syst.* 13, 1 (March 1988), 23–52. https://doi.org/10.1145/42201.42203

Timos K. Sellis and Leonard Shapiro. 1985. Optimization of Extended Database Query Languages. In *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data* (Austin, Texas, USA) *(SIGMOD '85)*. Association for Computing Machinery, New York, NY, USA, 424–436. https://doi.org/10.1145/971699.318993

Feng Sheng, Qiang Cao, Haoran Cai, Jie Yao, and Changsheng Xie. 2018. GraPU: Accelerate Streaming Graph Analysis through Preprocessing Buffered Updates. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA) *(SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 301–312. https://doi.org/10.1145/3267809.3267811

Xiaogang Shi, Bin Cui, Yingxia Shao, and Yunhai Tong. 2016. Tornado: A system for real-time iterative analysis over evolving data. In *SIGMOD '16*. ACM, 417–430. https://doi.org/10.1145/2882903.2882950

Julian Shun and Guy E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Shenzhen, China) *(PPoPP '13)*. Association for Computing Machinery, New York, NY, USA, 135–146. https://doi.org/10.1145/2442516.2442530

Robert Soulé, Martin Hirzel, Robert Grimm, Buğra Gedik, Henrique Andrade, Vibhore Kumar, and Kun-Lung Wu. 2010. A Universal Calculus for Stream Processing Languages. In *Proceedings of the 19th European Conference on Programming Languages and Systems* (Paphos, Cyprus) *(ESOP'10)*. Springer-Verlag, Berlin, Heidelberg, 507–528. https://doi.org/10.1007/978-3-642-11957-6_27

Marcelo Sousa, Isil Dillig, Dimitrios Vytiniotis, Thomas Dillig, and Christos Gkantsidis. 2014. Consolidation of Queries with User-defined Functions. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom). 554–564. https://doi.org/10.1145/2666356.2594305

Jesper H. Spring, Jean Privat, Rachid Guerraoui, and Jan Vitek. 2007. Streamflex: High-Throughput Stream Programming in Java. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications* (Montreal, Quebec, Canada) *(OOPSLA '07)*. Association for Computing Machinery, New York, NY, USA, 211–228. https://doi.org/10.1145/1297027.1297043

Nicolas Stucki, Jonathan Immanuel Brachthäuser, and Martin Odersky. 2021. Multi-Stage Programming with Generative and Analytical Macros. In *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Chicago, IL, USA) *(GPCE 2021)*. Association for Computing Machinery, New York, NY, USA, 110–122. https://doi.org/10.1145/3486609.3487203

Toyotaro Suzumura, Shunsuke Nishii, and Masaru Ganse. 2014. Towards Large-Scale Graph Stream Processing Platform. In *Proceedings of the 23rd International Conference on World Wide Web* (Seoul, Korea) *(WWW '14 Companion)*. Association for Computing Machinery, New York, NY, USA, 1321–1326. https://doi.org/10.1145/2567948.2580051

Walid Taha and Tim Sheard. 1997. Multi-Stage Programming with Explicit Annotations. In *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (Amsterdam, The Netherlands) *(PEPM '97)*. 203–217. https://doi.org/10.1145/258993.259019

Kanat Tangwongsan, Martin Hirzel, Scott Schneider, and Kun-Lung Wu. 2015. General Incremental Sliding-Window Aggregation. *Proc. VLDB Endow.* 8, 7 (feb 2015), 702–713. https://doi.org/10.14778/2752939.2752940

William Thies, Michal Karczmarek, and Saman Amarasinghe. 2002. StreamIt: A Language for Streaming Applications. In *Compiler Construction*, R. Nigel Horspool (Ed.), Vol. 2304. Springer Berlin Heidelberg, Berlin, Heidelberg, 179–196.

Mandana Vaziri, Olivier Tardieu, Rodric Rabbah, Philippe Suter, and Martin Hirzel. 2014. Stream Processing with a Spreadsheet. In *ECOOP 2014 – Object-Oriented Programming*, Richard Jones (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 360–384. https://doi.org/10.1007/978-3-662-44202-9_15

Venkateshwaran Venkataramani, Zach Amsden, Nathan Bronson, George Cabrera III, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Jeremy Hoon, Sachin Kulkarni, Nathan Lawrence, Mark Marchukov, Dmitri Petrov, and Lovro Puzar. 2012. TAO: How Facebook Serves the Social Graph. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. ACM, New York, NY, USA, 791–792. https://doi.org/10.1145/2213836.2213957

Keval Vora, Rajiv Gupta, and Guoqing Xu. 2017. KickStarter: Fast and Accurate Computations on Streaming Graphs via Trimmed Approximations. In *ASPLOS '17* (Xi'an, China). 237–251. https://doi.org/10.1145/3037697.3037748

Guozhang Wang, Wenlei Xie, Alan J Demers, and Johannes Gehrke. 2013. Asynchronous Large-Scale Graph Processing Made Easy.. In *CIDR*, Vol. 13. 3–6.

Kai Wang, Guoqing Xu, Zhendong Su, and Yu David Liu. 2015. GraphQ: Graph Query Processing with Abstraction Refinement—Scalable and Programmable Analytics over Very Large Graphs on a Single PC. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. USENIX Association, Santa Clara, CA, 387–401.

Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. Gunrock: A High-Performance Graph Processing Library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Barcelona, Spain) *(PPoPP '16)*. Article 11, 12 pages. https://doi.org/10.1145/2851141.2851145

Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 423–438. https://doi.org/10.1145/2517349.2522737

Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (Oct. 2016), 56–65. https://doi.org/10.1145/2934664

Kaiyuan Zhang, Rong Chen, and Haibo Chen. 2015. NUMA-Aware Graph-Structured Analytics. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Francisco, CA, USA) *(PPoPP 2015)*. 183–193.