A Runtime System for Interruptible Query Processing: When Incremental Computing Meets Fine-Grained Parallelism

JEFF EYMER, SUNY Binghamton, USA
PHILIP DEXTER, SUNY Binghamton, USA
JOSEPH RASKIND, SUNY Binghamton, USA
YU DAVID LIU, SUNY Binghamton, USA

Online data services have stringent performance requirement and must tolerate workload fluctuation. This paper introduces PitStop, a new query language runtime design built on the idea of *interruptible query processing*: the time-consuming task of data inspection for processing each query or update may be interrupted and resumed later at the boundary of fine-grained data partitions. This counter-intuitive idea enables a novel form of *fine-grained concurrency* while preserving *sequential consistency*. We build PitStop through modifying the language runtime of Cypher, the query language of a state-of-the-art graph database, Neo4j. Our evaluation on the Google Cloud shows that PitStop can outperform unmodified Neo4j during workload fluctuation, with reduced latency and increased throughput.

CCS Concepts: • **Information systems → Query optimization**.

Additional Key Words and Phrases: interruptible query processing; query language runtime design; fine-grained parallelism

ACM Reference Format:
Jeff Eymer, Philip Dexter, Joseph Raskind, and Yu David Liu. 2024. A Runtime System for Interruptible Query Processing: When Incremental Computing Meets Fine-Grained Parallelism. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 332 (October 2024), 28 pages. https://doi.org/10.1145/3689772

1 Introduction

Cloud data centers facilitate the deployment of *online* services, many of which are supported with a database query language runtime as the backend. These services must be able to process a large number of incoming requests (queries and updates) at a rapid rate. The performance of these online services often critically depends on that of the query runtime. A particular thorny problem for online services is *workload fluctuation*: the request rate may change over time, and often rapidly so. While provisioning less compute resource than needed is undesirable for maintaining Quality of Service (QoS), provisioning more than needed introduces waste, with severe impact on sustainability [42]. There is a growing interest in addressing this challenging problem [11, 12, 20, 25, 28].

Interruptible Query Processing. We take on this fundamental problem with a solution at the layer of *query runtime* design. We show that by introducing a notion of fine-grained concurrency, the performance of online data processing can be improved in the presence of workload fluctuation. Our concrete proposal is *interruptible query processing*: the processing of each query or update can be *interrupted* upon *partial* data inspection and resumed later. In other words, we break down the

Authors' Contact Information: Jeff Eymer, SUNY Binghamton, Binghamton, USA, jeymer1@binghamton.edu; Philip Dexter, SUNY Binghamton, Binghamton, USA, pdexter1@binghamton.edu; Joseph Raskind, SUNY Binghamton, Binghamton, USA, jraskin3@binghamton.edu; Yu David Liu, SUNY Binghamton, Binghamton, USA, davidl@binghamton.edu.

ACM 2475-1421/2024/10-ART332
https://doi.org/10.1145/3689772

otherwise monolithic process of data inspection needed for answering a query or update into a series of partial inspections, each of which is called a *leg*. This idea may appear counter-intuitive at the first glance: by interrupting the processing, wouldn't latency be increased?

The key insight here is that we must consider all requests as a whole — especially when they arrive at a rapid rate — and the interruption in data inspection unleashes the opportunity of *fine-grained concurrency* across *multiple requests*, i.e., processing a leg of one request can happen concurrently with some leg of another request. We call this new notion *leg concurrency*, and the parallelism enabled by it *leg parallelism*. Semantically speaking, an interesting consequence of leg concurrency is that we can interleave the legs from multiple requests *regardless of whether they are reads or writes* while still preserving the strong guarantee of *sequential consistency* [40].

In design, interruptible query processing is the confluence of two influential ideas in programming languages: incremental computing [4, 31, 33, 34, 47] and fine-grained parallelism [5, 22, 35]. Indeed, interrupting an otherwise monolithic data inspection process can alternatively be viewed as incrementally propagating the query/update request inside the data. The performance benefits of interruptible query processing indeed arise from the synergy when incremental computing meets fine-grained parallelism. Now that the otherwise monolithic process of data inspection is divided into concurrent legs, we inherit the benefits of fine-grained parallelism in load balancing [6, 22], and more generally, scalable performance [44, 51, 54]. Thanks to incremental computing, two classic data processing optimizations — batching [24, 58, 65] and fusion [46, 50, 52, 64] — can be applied with more effectively: they can now be *opportunistically* applied *during* the incremental process of data inspection, which we term *on-the-fly* batching and fusion (see § 3).

The PitStop System. We implement the idea of interruptible query processing by replacing the existing runtime of Cypher [21, 27], a query language for a state-of-the-art graph database, Neo4j [2]. The new system, called PitStop, can be viewed as a new Cypher runtime that processes Cypher queries/updates without any change to its programming model, but with the new dynamic semantics. We choose Cypher/Neo4j as our baseline for two reasons. First, graph databases are an emerging system with broad applications, such as bioinformatics, social networks, machine learning, and artificial intelligence. Second, graph databases set a more challenging "high bar" for evaluation as they can be viewed as a combination of two: as graph processing systems, they must address queries over complex and structured data, and as databases, they must meet stringent performance requirements when deployed *online*, such as low latency and high throughput.

Our results show PitStop can significantly reduce latency and increase throughput of Neo4j, by 1-2 orders of magnitude in some contexts of workload fluctuation. Our comparative baseline of unmodified Neo4j is set up in the realistic scenario where different queries can be processed in parallel, except that it does not have the fine-grained parallelism that PitStop enjoys. We further implemented a second baseline, an optimization on Neo4j where batching and fusion are supported at a "top-level" buffer, i.e., a store that temporarily keeps the arriving requests before they are processed. PitStop still outperforms the latter baseline by a large margin (see § 6). Furthermore, we found PitStop can reduce the *variance* in latency among request processing, useful in mitigating a pernicious problem in online processing: the long-tail latency [37].

Contributions. This paper makes the following contributions:

- a new query language runtime design that supports interruptible query processing, enabled by incremental query/update propagation and fine-grained concurrency
- an evaluation over real-world datasets (Twitter, StackOverflow, and GitHub) in a modern cloud environment that demonstrates the performance benefits of PitStop

- a design-space exploration over different fluctuation settings and different natures of query traces, together with a hybrid design where indexing co-exists with data inspection

2 Background

We briefly review key terms and principles in query language runtimes and processing systems. With an essential role of data processing in the computing stack, a query language—if it must be called a domain-specific language—captures a domain familiar to most readers. That said, some design principles well known to query runtime designers may not be familiar to all. We also use this section as a summary for terms that will be consistently used in the rest of the paper.

Operations: Queries and Updates. In databases, a "read" request (i.e., one that does not mutate the presistent data) is often called a *query* and a "write" request (i.e., one that does) is often called an *update*. To unify the two in one term, we henceforth refer to each request of online data processing as an *operation*. Perhaps confusingly, the historical terms of *query processing* and *query languages* can refer to the support of both reads and writes. In this setting, one may refer to query processing for only "read" requests as *static data processing* [14, 63], and one that supports both as *dynamic data processing*. By this terminology, PitStop is an instance of the latter.

Data Organization. Not to lose generality, we view data(base) as a collection of *data units*, each of which consists of a *key* (i.e., the unique identifier of the unit), and a *payload value* (i.e., the content held by the data unit). When no confusion can arise, we also shorten a payload value as a *value*. The key-value pair representation may be evocative to a specific form of databases, the key-value store, but our treatment is more general: the payload value in our formulation may carry additional information that reflect how data units are related to each other, i.e., structural information in data.

Consider a graph for example. Imagine we wish to represent a simple social network of `Alice` and `Bob` where `Alice` is a friend of `Bob`. This can be captured through two data units, with keys $k_{\texttt{Alice}}$ and $k_{\texttt{Bob}}$ identifying `Alice` and `Bob` respectively. The data unit of `Bob` may carry a payload value of $k_{\texttt{Alice}}$, indicating that `Alice` is his friend. In other words, the payload value can serve as an encoding of the outgoing edges of the (logical) graph. In the rest of the paper, our main interest is on structured data such as graphs. From now on, we interchangeably call each data unit as a *node*. We also use "node k" to refer to a node whose key is k.

Structured Persistent Data: In-Memory Representation and In-Storage Representation. For C programmers, it might be intuitive to implement a graph where graph vertices are memory objects, and graph edges are pointers from one memory object to another. A similar implementation for Java programmers can be carried out, except that object references are used instead of pointers. Indeed, this is how graphs as *non-persistent* data structures are routinely implemented.

For *persistent* graphs however, real-world query runtime systems do not design the in-memory representation of graphs through pointers or object references. The key insight here is that databases must *store* data *in some linear order*. When such data is loaded into memory, it would be prohibitively expensive to reconstruct a pointer/reference-based graph data structure from the linear representation. Worse, when the in-memory pointer/reference-based graph is updated, synchronizing it with the in-storage sequence-based representation would be even more expensive. As a result, real-world query runtimes nearly always keep the in-memory representation and the in-storage representation as similar as possible, if not identical.

The consequence is that, regardless of the *logical* view of the persistent structured data — a list, a tree, a graph —- their in-memory representation in real-world systems is generally a *sequence*, i.e., a collection of objects in total order. Recall our earlier example that the structural information of the graph is encoded into the payload value of nodes. As a result, the in-memory representation of

this graph can simply be a sequence of two nodes: node $k_{\texttt{Alice}}$, followed by node $k_{\texttt{Bob}}$. The graph database PitStop is built upon — Neo4j — has sequence-based in-memory representation.

The Lifecycle of Operation Processing. To process an operation, the query runtime must *inspect* data, sometimes also referred to as *data scan* (a term historically used for unstructured data) or *data traversal* (a term historically used for structured data). For example, if a query intends to search for a data unit which contains a payload value of 17, it will need to scan/traverse through the sequence of data units, inspecting each of them until it finds the unit with 17 as the payload value.

In the rest of the paper, we describe this lifecycle of processing in the following terms: an operation *arrives* at the query runtime, *propagates* through the data as it inspects each node and compares keys, and eventually *realizes* at the node that the operation is intended for. We also use *serving* to refer to the combined lifecycle states from propagating to realizing.

Sequence-based in-memory representation has important implications on data inspection. It implies that when data is scanned/traversed, some linear order—often the order in the in-memory sequence representation—is followed. The design of PitStop reflects the same principle in query runtime design. One notable exception to linear-order scanning/traversal is the use of indexing, a common strategy particularly suited for databases that are query-only (i.e., no updates), or query-dominant (i.e., rare updates). We support indexing as a variant called PitStop-I (see § 7.1).

Sequential Consistency. *Sequential consistency* [40] is a fundamental property that says that if two operations with unique labels ℓ_1 and ℓ_2 respectively arrive at the query runtime and ℓ_1 arrives earlier than ℓ_2, then the final result of query processing—both the query results and the updated database state— must be identical as one from processing ℓ_1 and ℓ_2 in the chronological order of their arrivals, i.e., ℓ_1 must be first completed by the query runtime, then ℓ_2 is processed.

For static data processing, sequential consistency trivially holds. In dynamic data processing, guaranteeing sequential consistency and achieving performance are often at odds. The property is non-trivial when concurrent processing of multiple operations is allowed, or optimizations across multiple operations are enabled.

Provisioning. In computer systems, *underprovisioning (UP)* refers to the system state when the system resources (say, CPU, memory, bandwidth) allocated to an application cannot meet the demand of the application. Conversely, *overprovisioning (OP)* refers to the system state when system resources exceed the demand of the application. In the setting of query systems, we use UP to refer to the system state when operations arrive more rapidly than they can be processed, and OP as the opposite. We also say the query runtimes undergoes a *UP/OP phase* for the duration of time when the underlying system is in the UP/OP state. As workload fluctuation is the primary use scenario PitStop addresses, UP and OP will play a prominent role in our experimental evaluation.

3 PitStop Data Processing

Figure 1a shows a a simple online graph database we use as a running example. The graph consists of 6 nodes, whose traversal follows the order from k_1 to k_6. Here, 3 operations arrive at the query language runtime (the "database engine"), labeled with ℓ_1, ℓ_2, and ℓ_3, with ℓ_1 arriving the earliest. The first two operations are query operations; for example, "$\ell_1 \rightarrow$ query k_6" says that the ℓ_1 operation looks for the node with key k_6. The third operation is an update, where "$\ell_3 \rightarrow$ update k_3 with 200" says the ℓ_3 operation intends to update the payload value of node k_3 with 200.

We start with a pedagogical subsection to describe the concurrent-but-not-parallel behavior of PitStop, followed by another subsection to describe the concurrent-and-parallel behavior. The former is only meant for introducing basic concepts, and the latter is where real-world benefits lie. PitStop under these two variants are illustrated in Fig. 1b and Fig.1c respectively. Each row

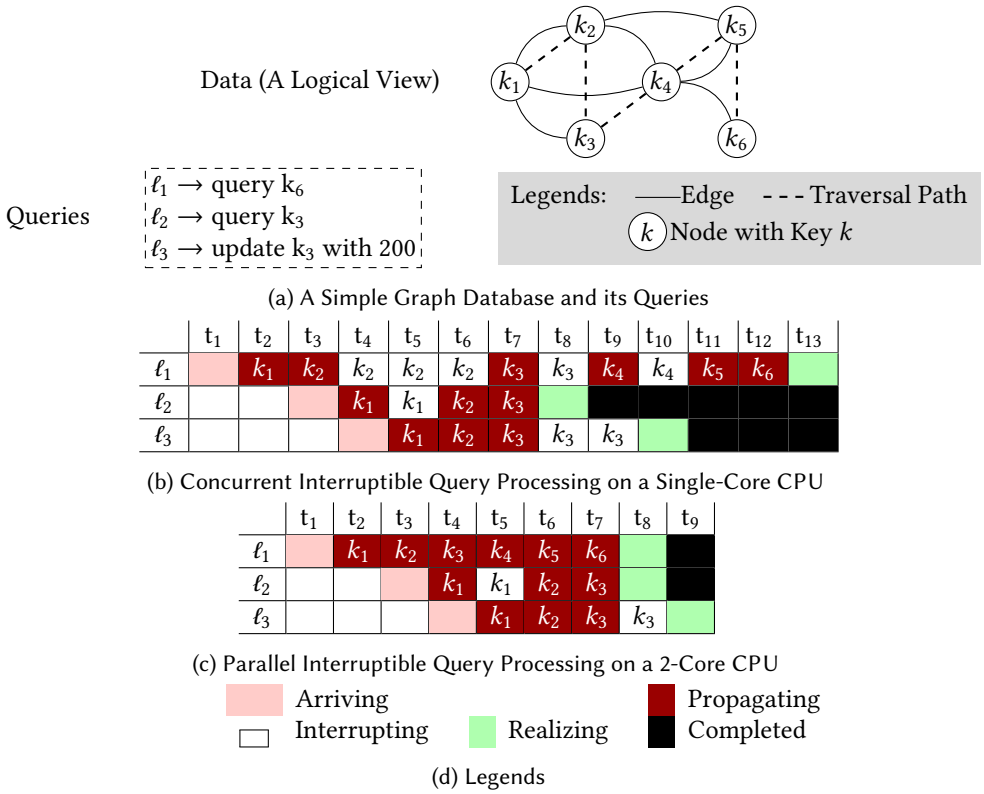


Fig. 1. An Example on Interruptible Query Processing. For (b) and (c), each row represents the processing of an operation; each column represents a computation step, where a column denotes an abstract unit of elapsed time, where $t_1, t_2, \ldots t_{13}$ follows the chronological order; each cell with k_n indicates the query operation currently reaches (pitstop) node with key k_n. To highlight the conceptual view, (b) is analogous to a concurrent (but not parallel) PitStop execution on a single-core CPU. (c) is a parallel execution with 2 cores.

represents an operation under concurrent query processing, and each column represents a unit of elapsed time. While this illustration may remind some readers of pipelines, PitStop is not a pipeline-based system: there is no synchronization across the processing of different operations.

Basic Concepts in Interruptible Query Processing. In standard query runtime design which we henthforth call *monolithic processing* (MP), a *run-to-completion* semantics is adopted for data inspection: once the processing of a request starts, it runs to completion before the hosting thread(s) can process another request. For example, 6 continuous time units would be needed for ℓ_1 processing under MP [1]. Note that run-to-completion semantics does not preclude concurrency, but it only supports a *coarse-grained* form of concurrency: different operations can be concurrently processed by different threads when the consistency model allows it, but from the perspective of *each operation*, its processing must run to completion.

In PitStop, the data inspection process — even for one single operation — is incremental and may be *interrupted*. PitStop partitions data so that inspection can be interrupted at the *last node of each*

[1]Strictly speaking, OS-level context switches could happen during the processing of ℓ_1 under MP, so the 6 time units may or may not be continuous. They however are still taken by the same thread. For simplicity, the rest of discussion in this section does not consider OS/hardware context switches. We revisit this discussion in § 6.11.

data partition, which we call a *pitstop*. In this simple example, we assume each node can be a partition and hence a pitstop in data inspection. For example, the processing of ℓ_1 is interrupted at node k_2 at timestamps t_4, t_5, t_6. The consequence of the interruption is that the current thread can be freed up and then process other operations, improving concurrency. Similarly, an interruption happens at pitstop k_3 at t_8 and pitstop k_4 at t_{10}. We further consider an imaginary "root" pitstop — the position in data before the first partition. We call the inspection of a data partition — i.e., propagating an operation from one pitstop to the next pitstop — as a *leg*. For instance, the computation happening at t_2 and t_3 form the first leg, and that at t_{11} and t_{12} forms the fourth (and last) leg.

To clarify the terminology, we always use the term "pitstop" to refer to a node in data and the term "leg" to refer to a computation that enables the inspection of data nodes sandwiched by a pair of pitstops. Given that the leg of inspection is synonymous to the propagation of an operation from one pitstop to the next, we also informally say "an operation o residing in pitstop k" when o has hitherto propagated to node k (after one or more legs of processing).

Whenever a thread becomes available, it can select an operation (or a batch of them) currently under processing and process it by a leg. To ensure fairness, the selection of an operation (and its processing by a leg) is random (see § 4).

In summary, if we take the lifecycle of an operation as a whole, its processing in PitStop is not a continuous process from arrival to completion, but incremental and interruptible.

Leg Parallelism. Our pedagogical description so far has only focused on concurrency without parallelism. We now describe the more realistic setting where concurrent executions run in parallel.

PitStop, through dividing the otherwise monolithic task of data inspection into smaller leg-sized concurrency units, can improve performance through balancing the workload across cores more effectively. The granularity of parallelism in PitStop is the *leg*. In contrast, when MP is parallelized, its run-to-completion requirement means its parallel unit is the entire serving process, e.g., 6 time units for ℓ_1 processing. To understand why granularity matters, let us now come back to the example. MP under the two-core execution can indeed allow ℓ_1 and ℓ_2 to process in parallel, but ℓ_3 may still have to *involuntarily wait* and start only after either ℓ_1 and ℓ_2 is completed. We call this unfortunate situation the *cascading effect*: the long latency of serving earlier arriving requests may cause the later arriving request to wait "involuntarily" — especially in the UP phase — and further extend the latency of the latter. With PitStop, the processing of ℓ_3 can start as soon as either of ℓ_1 or ℓ_2 reaches a pitstop and frees up its host thread. This is particularly helpful if ℓ_3 can be completed only after inspecting a small portion of data. In other words, the latency of ℓ_3 is less impacted by the involuntary wait and the cascading effect in the UP phase.

Another important — but less obvious — advantage of leg parallelism is its *semantics-independence*: deterministic parallel processing of multiple legs is independent of the read/write nature of the operations performed within individual legs. In PitStop, two threads can run two legs (of different operation batches residing at different pitstops) in parallel, *regardless of whether the operations processed by the two legs are query/query, query/update, or update/update*. For example, while a thread in PitStop processes ℓ_2 for the leg propagating from pitstop k_1 to pitstop k_2, another thread can process ℓ_3 for the leg propagating from pitstop "root" to pitstop k_1. This is in contrast with traditional MP, where the processing of a write operation cannot arbitrarily interleave with that of a read operation. For example, if the monolithic processing of ℓ_3 — an update operation — *were* to be parallelized with that of ℓ_2, the processing of ℓ_2 may return 200, which would be in violation of sequential consistency. In PitStop, sequential consistency is preserved (see § 4).

Leg parallelism differs from data parallelism: it happens between processing (different legs of) different operations, rather than different data partitions for processing the same operation.

On-the-Fly Batching and Fusion. PitStop composes well and further strengthens classic multi-request optimizations in online data processing, especially *batching* [24, 58, 65] and *fusion* [46, 50, 52, 64]. As an optimization, fusion allows multiple operations to be combined into an equivalent operation potentially involving fewer computation steps. Batching — while not reducing the overall workload of the computation — allows multiple operations to share one data inspection process, known for its benefits in more efficient I/O (for accessing persistent data), cache (for in-memory processing), and lock management (for concurrent processing).

Without PitStop, batching and fusion can only be performed when the operations arrive, presumably in a top-level buffer (see § 1). With PitStop, they may happen at any pitstop. At t_6, ℓ_2 and ℓ_3 form a batch and continue to propagate in tandem in one step. At t_7, all 3 operations batch together for one step of processing. Batching is now *on-the-fly*, i.e., at *any pitstop*: a batch formed by ℓ_2 and ℓ_3 at t_6 and a batch of all 3 operations at t_7. On-the-fly batching increases the likelihood of batching: assuming available resources, a newly arriving operation can start its processing immediately and forms a batch with other operations later after partial data inspection. In contrast, classic batching systems have to grapple with a binary decision when an operation arrives: should it start processing immediately and forgo the opportunity of batching, or should it *voluntarily wait* until the next operations come to form a batch?

For fusion, consider the two operations are submitted to the same graph processing engine as in Figure 1, wtih ℓ_4 refers to update k_6 with 500, ℓ_5 refers to update k_6 with 600, and ℓ_4 arriving earlier than ℓ_5. Here, node k_6 is first updated to value 500 and later updated to 600, a pattern common in the real world where updated data are often updated again. Before ℓ_4 reaches its intended node k_6, eliminating ℓ_4 does not affect the state of the database.

Use Scenarios and End-User Benefits. Interruptible query processing is designed to target online server-type environments such as cloud servers (see § 1). Specifically, it is designed to address several thorny problems in achieving scalable performance on such platforms. We now briefly summarize our end-user benefits, and how PitStop leads to them.

The primary benefit is that interruptible query processing is resilient to workload fluctuation. With a combination of incremental query processing and fine-grained leg parallelism, PitStop is adaptive to the system state changes betweeen UP and OP. As we described earlier in this section, classic MP instead may suffer from the *cascading effect*.

Second, PitStop can reduce the occurrence of *long-tail* latency. An operation may take long to process for various reasons: (a) some non-deterministic system events (such as I/Os, resource allocations, or hardware failures) cause delays; (b) an operation may represent a heavier computation, e.g., more complex processing or more data for inspection; and (c) the long processing due to (a) and (b) may excessively hold systems resources away from other operations, making the latter fall into long tail as well. PitStop mitigates the impact of the delays caused by (a) and (b) *on the entire system*, i.e., reducing the occurrence of (c). In doing so, it may increase the latency of (b), but not necessarily that of (a) because the latter is dominated by the non-deterministic event. In § 6.4, we show PitStop can reduce *latency variance* among the operations.

Third, PitStop allows classic online optimization techniques—batching and fusion—to be applied at a finer granularity, enabling *on-the-fly* batching or *on-the-fly* fusion.

Fourth, leg parallelism is semantically independent of the read/write nature of operations. This implies that it can deliver comparable performance for both static and dynamic data processing.

Applicability and Limitations. First, PitStop is primarily designed for online data processing systems where the requesting operations arrive at fluctuating rates, leading to fluctuating workloads between UP and OP. When workloads are perpetually in an UP or OP state, PitStop can remain effective in some scenarios but not all (see § 6.8). Effective or not, PitStop is less interesting for

these non-fluctuating scenarios because an end user can solve the problem by simply configuring different resources (e.g., more/fewer CPU cores). Second, while PitStop does not rely on any existing partitioning strategy of the underlying database (Neo4j in our case), its algorithm logically maintains a partitioned view of the data where pitstops reside between partitions. For data that are fundamentally non-partitionable, PitStop does not apply. Finally, we will discuss some extensions in § 7, for settings of data processing different from the current implementation of PitStop.

4 PitStop Algorithm Specification

```
Global Definitions and Variables
  structure DataUnit
    key : key
    value : value
    deleted : Boolean = false
  end
  data : List<DataUnit>
  structure Operation
    target : key
    name :Enum {add,delete,update,query }
    f : Functor
    bound : int
  end
  structure BatchOps
    ops : List<Operation>
  end
  structure Server
    legs : List<BatchOps>
    results : Map<Operation, Value>
  end
  server : Server
```

Fig. 2. PitStop Structures

Fig. 2 defines relevant data structures. The modifications to standard run-to-completion query processing are highlighted in green.

Data and Operations. Not to lose generality, we first define our core algorithm over semi-structured data, and its variation on graph data will be described in § 5. Here, data is defined as a list of DataUnit's, where each DataUnit contains a `key-value` pair and a flag to mark whether the node is `deleted`.

Each Operation contains the necessary information for its processing, including the name of operation (`name`), the key of the DataUnit where the realization happens (`target`), the functor (`f`), and a field called the `bound` which we will explain later. A functor is a first-class lambda function that defines the logic for realization. For example, operation ℓ_1 in § 3 can be encoded with a functor as $\lambda d.\ (d.\texttt{value})$, and the ℓ_3 operation can be encoded with a functor

$$\lambda d.\ (\text{DataUnit}\{d.\texttt{key}; 200; \texttt{false}\})$$

For `add` and `delete` operations, the functor definition is irrelevant. In this simple specification, a realization can only happen when the target of the operation matches the key of the data unit. More complex predicate-based matching can be encoded through **if..then** within the functor.

Pitstops and Legs. A pitstop is placed for every *LDSIZE* number of DataUnit's. Pitstop 0 indicates the imaginary data unit (see § 3) before the first, used for keeping operations that have just arrived but not served at all yet.

The unique structure of PitStop is the `legs` field of Server, a list whose i^{th} element indicates the batch of operations (BatchOps) currently reside at the pitstop i, where $i \geq 0$. When a batch of operations in the pitstop i are ready for their next leg of processing, each DataUnit between pitstops i and $i+1$ is inspected. Observe that our `legs` field naturally considers on-the-fly batching: the operations residing at the same pitstop are batched together, through structure BatchOps; at different pitstops, the number of operations within each BatchOps may well differ.

In addition, the Server keeps all results from all completed operations in the `results` field as a map from Operation to Value. While this data structure helps us specify our algorithm, realistic systems (e.g., privacy-preserving servers) may choose to only keep them temporarily (such as per session), or not at all.

Algorithm 1: Scheduler

```
   procedure main()
1      while true do
2          upon ready t : Thread do
3              if op arrived then
4                  if op.name = add then
5                      _ , v ← realize(o_i, _)
6                      server.results[o_i → v]
7                  else
8                      op.bound ← size(data)
9                      run t with append(0, op)
10             else
11                 l ← ⌈size(data)/LDSIZE⌉
12                 p ← random ([0, l))
13                 if size(server.legs[p]) > 0 then
14                     run t with propagate(p)
           end
   end
   procedure append(p : Int, op : Operation)
15     if undef server.legs[p] then
16         server.legs[p] ← BatchOps{[]}
17     add op to server.legs[p].ops
   end
```

Algorithm 2: Propagator

```
   procedure propagate(p : int)
18     pd ← LDSIZE * p
19     while pd < min( size(data) , LDSIZE * (p+1))
       do
20         d ← data[pd]
21         lock server.legs[p].ops
22         for [o_1, o_2, ..., o_n] in server.legs[p].ops do
23             if d.deleted = false
               and d.key = o_i.target
               then
24                 u, v ← realize(o_i, d)
25                 if u != _ then
26                     data[pd] ← u
27                 server.results[o_i → v]
28             else if pd+1 >= op.bound then
29                 server.results[o_i → ⊥]
30             else
31                 append(p+1, o_i)
           end
32         server.legs[p].ops ← []
33         unlock server.legs[p].ops
34         pd ← pd+1
   end
```

Algorithm Details. Algorithm 1 describes the overall runtime. Algorithm 2 defines the most unique aspect of PitStop design, on leg-based incremental propagation. Algorithm 3 defines the behavior of common operations for dynamic data processing: addition, deletion, query, and update. Notation $X[y \rightarrow z]$ means a mapping identical to X, except its domain element y is mapped to z. We use $\top$ to refer to a special Value indicating that the operation processing succeeds and $\bot$ to refer to the failure, such as a "key not found" error. In both algorithms, *data* and *server* are global variables declared in Fig. 2.

Upon a thread becoming available, the scheduler checks if a new operation has arrived (Line 3). If so, the newly arrived operation is either immediately processed if it is an add operation (Lines 4-6), or append-ed (Lines 9, 15-17) to the batch of operations meant to reside at pitstop 0. If no new operation has arrived, the thread randomly selects a pitstop and propagates the operation batch corresponding to that pitstop for a leg (Lines 11-14).

The behavior for propagation is shown in Algorithm 2. The algorithm first calculates *pd*, the index of the first element in the data immediately after pitstop p (Line 18). The outer loop steps through the data units one by one (Line 34) until either the next pitstop or the end of data is reached (Line 19). The inner loop steps through each operation in the batch (Line 22). The operation may be realized (Line 24) if the data unit d is not `deleted` and the key matches the operation's target. Upon realization of an operation, the data is updated to reflect the possible change made (Lines 25 and 26), and the result of realization informs the server's `result` update (Line 27). If an operation cannot be realized at a leg, it is append-ed to the batch for the next leg (Line 31).

An important observation here is that *order matters*: the order of Operation within each BatchOps reflects the chronological order of operation arrival, and the order of BatchOps's within the `legs` field also reflects the same chronological order. In other words, there is no "jump-ahead" among operations: the Operation's still arrive at each pitstop in the same order as they arrive at the data processing engine (pitstop 0). Observe that both when an operation first arrives (Lines 9) and when an operation is propagated to the next pitstop (Line 31), we consistently use append, i.e., adding the operation to the tail of the list.

Algorithm 3: Sample Realization Functions

```
    procedure realize(op : Operation, d : DataUnit)
35      switch op.name do
36          case add do
37              k ← keygen()
38              v ← apply op.f to ()
39              d_0 ← DataUnit{k, v, false}
40              [d_1, d_2, ..., d_n] ← data
41              data ← [d_1, d_2, ..., d_n, d_0]
42              return _, ⊤
43          case delete do
44              d.deleted ← true
45              return d, ⊤
46          case update do
47              d ← apply op.f to d
48              return d, ⊤
49          case query do
50              ret ← apply op.f to d
51              return d, ret
        end
    end
```

Finally, if the inspection reaches the end of data and the operation is not realized, a special value $\perp$ is the result (Line 29). In a dynamic setting when data units may be add-ed online, care is required for maintaining data *visibility*. For example, if a query operation with a target k arrives before an add operation that adds a data unit with key k, the query should return $\perp$. The challenge for PitStop is that even though the add operation immediately changes data, the query operation may take time to propagate and hence may incorrectly find the added data unit. To address this, we associate each Operation with a bound field, recording the size of data at the time of its arrival. This bound is used to determine the end of data inspection for the operation (Line 29).

Our algorithm specification is orthogonal to the concrete data representation itself: we do not require data to be organized as a list. All PitStop requires is an order can be derived from the data representation. For instance, if an in-memory graph is represented as a pointer-based linked structure, the structure is still traversable through some (depth-first or breadth-first) order. In practice indeed, linear data organization is common in large data (or graph) processing — such as in Neo4j — because it would otherwise be too costly to reconstruct a pointer graph based on the storage representation (which is linear) and continuously synchronize the two.

We adopt a simple model for thread safety: when a leg propagation happens, its associated BatchOps is locked, as shown at Line 21 and Line 33. This prevents different threads from propagating the same operation batch at the same pitstop more than once. This is an example why batching is a useful optimization: only one lock needs to be managed for the entire batch.

Complexity. Given the size of data as m and the number of operations as n, the time complexity of our PitStop is $O(m \times n)$. Despite our design of leg-size propagation over operation batches, each operation still only inspects each data unit at most once. The space complexity of PitStop is $O(n)$. In other words, the only additional storage needed is to keep an entry for each operation under processing, as in the `legs` field of Server.

Properties. PitStop observes sequential consistency. This is a non-trivial result considering the query processing can be non-deterministically interrupted at pitstops; concurrent/parallel processing of multiple operations is allowed; random legs are selected for propagation; on-the-fly batching and on-the-fly fusion are supported.

To gain an intuition on why sequential consistency holds, consider an example where two operations ℓ_1 and ℓ_2 arrive at the query runtime, and the former arrives earlier than the latter. For simplicity, let us assume both are targeting a node with key k and ℓ_1 is an update operation whereas ℓ_2 is a query operation. Not to lose generality, there are three subcases: (1) neither ℓ_1 nor ℓ_2 has reached node k during their propagation. In this case, neither query nor update is realized; (2) ℓ_1 has reached node k but k_2 has not reached node k. In this case, the update can be realized but not the query. (3) both ℓ_1 and ℓ_2 have been propagated to node k; in this case, observe that Line 22 of

Operation	Equivalent Cypher Query
find user ⟨*property*⟩ ⟨*value*⟩	`MATCH (n : User{⟨property⟩ : ⟨value⟩})` `RETURN n`
findFollowers ⟨*property*⟩ ⟨*value*⟩	`MATCH (: User{⟨property⟩ : ⟨value⟩}) ⟵ (n : User)` `RETURN n`
findFollowees ⟨*property*⟩ ⟨*value*⟩	`MATCH (: User{⟨property⟩ : ⟨value⟩}) ⟶ (n : User)` `RETURN n`
update ⟨*property*⟩ ⟨*value*⟩ ⟨*new property*⟩ ⟨*new value*⟩	`MATCH (n : User{⟨property⟩ : ⟨value⟩})` `SET n.⟨new property⟩ = ⟨new value⟩`

add ⟨*property*⟩ ⟨*value*⟩ ⟨*property to find*⟩ ⟨*value to find*⟩	`MATCH (a : User{⟨property to find⟩ : ⟨value to find⟩})`
	`CREATE (b : User{⟨property⟩ : ⟨value⟩})`
	`CREATE (b) -- [r : FOLLOWS] ⟶ (a)`

delete ⟨*property*⟩ ⟨*value*⟩	`MATCH (n : User{⟨property⟩ : ⟨value⟩})` `DELETE n`

Table 1. Cypher Queries for the Twitter Dataset

Algorithm 2 says ℓ_1 must be realized before ℓ_2. In particular, note that there does *no* exist a fourth subcase where ℓ_2 is propagated to k but ℓ_1 is not, according to the preservation of the chronological order during the propagation process (see *Algorithm Details* discussion earlier). Finally, observe that the analysis here still assumes that the processing of ℓ_1 and ℓ_2 are concurrent: the chornological ordering does not preclude, say, the propagation of ℓ_1 by one leg and that of ℓ_2 by another leg, from happening in parallel.

We defer a formal proof of this property to the supplementary material. The formal insight is that PitStop is inspired by DON calculus [15]. Indeed, PitStop can be viewed as a concrete instance in the spectrum of online data processing algorithms formally defined by their calculus, and our property sequential consistency is (conceptually) a corollary of their property of determinism. A more detailed discussion on this connection can be found in § 8.

In addition, PitStop preserves *atomicity* [26, 29], which in the context of data processing says that the processing of each operation must abide by our intuition of "all or nothing". To gain intuition on why it holds, recall in § 2, the lifecycle of an operation is composed of a sequence of steps in the following order after the operation arrives: propagation, propagation, . . . , propagation, propagation, realization. To establish atomicity, the most relevant observations here are: (1) each propagation step *reads* the *immutable* key from the graph node, with no changes in the *persistent* data. From here on, we refer to such a step as producing a R_I effect (i.e., immutable read); (2) the realization step may either *read* the potential *mutable* payload from the graph data, or *write* to such data. From here on, we refer to the former as producing a R_M effect (i.e., mutable read), and the latter as producing a W effect (i.e., write). If we extend our reduction system where the propagation/realization rules explicitly produce the effects specified above as observables, the processing of an operation leads to a trace in one of the two patterns:

- Case I: a query operation: $R_I, R_I, \ldots, R_I, R_I, R_M$
- Case II: an update operation: $R_I, R_I, \ldots, R_I, R_I, W$

In Case I, all steps are "reads" and no commit to the persistent data is needed. Case II is a *tail-commit*, i.e., only the very last step involves change to persistent data. The "all-or-nothing" semantics for atomicity is preserved depending on whether the last step is committed ("all") or aborted ("nothing").

Dataset	# Nodes	# Edges	Read-Only Trace Operations	Read-Write Trace Operations	
Twitter [39]	41.7 million users	1.47 billion 'follows'	33.33% find user 33.33% find followers 33.33% find followees	40% find user 10% find followers 10% find followees	16.67% add user 16.67% delete user 16.67% update user
Stack Overflow [3]	11.8 million users 47.9 million posts 57 thousand tags	132 million 'posted' 56 million 'tagged'	33.33% find user 33.33% find posts of a user 33.33% find post tags of user	40% find user 10% find posts 10% find tags	16.67% add user 16.67% delete user 16.67% update user
GitHub [1]	4.4 million users 9.7 million repositories	16.9 million 'pulled from' 92.0 million 'pushed to' 4.3 million 'forked from' 11.9 million 'commented on'	25% find pulls 25% find pushes 25% find forks 25% find comments	15% find pulls 15% find pushes 15% find forks 15% find comments	16.67% add user 16.67% delete user 16.67% update user

Table 2. Dataset Description (X% represents the composition ratio of operations in the trace.)

5 Implementation and Experiment Settings

We implemented PitStop by modifying the core data inspection algorithm of Neo4j version 3.5.12 [2], written in Java. PitStop is currently deployed on and experimented with a Google Cloud virtual machine running Ubuntu 16.04, with 96 CPU cores and 86.4 GB of memory.

Graph Data Support. In Neo4j, a graph is represented as a list of Node's and a list of Relationship's. Both forms of data are identified by `Id`'s, analogous to our Key. To facilitate node-to-relationship queries, each Neo4j Node contains fields to refer to *its* Relationship's, (conceptually) the range of entries in the Relationship list. Both may contain a deleted field, as part of Neo4j's built-in support.

Our implementation does not require any metadata change of the Neo4j database, nor any programming model of its query language, Cypher. The specific Cypher queries that we used for the traces can be found in Table. 1. By default, we set *LDSIZE* as 1% of the length of the dataset.

Fusion Implementation. Our implementation also supports fusion similar to the example in § 3. Just as an available thread may randomly select an operation batch for propagation (see Line 14 in the algorithm), it may also randomly select an operation batch for fusion. The behavior of fusion is a linear-time inspection of all operations within the operation batch, and fusion happens if two consecutive `update` operations are applied to the same node, and all `query` or `delete` operations between the two are not applied to the same node.

Try Locking. As an optimization of our specified algorithm, the lock we associate with each operation batch is a try-lock. In other words, if a pitstop is randomly selected according to Algorithm 1 but its associated batch is currently locked, our implementation will go back to the while loop in Algorithm 1 and randomly select another operation.

Datasets. We ported three datasets to Neo4j, including Twitter, StackOverflow, and GitHub. Among all datasets on Neo4j's website, StackOverflow and GitHub are the largest ones. We further manually imported Twitter data to a Neo4j-compatible format, which is larger than the other two. For all experiments — including different trials of the same experiment — we always first initialize the graph to the same initial state.

Operation Traces. While commonly used datasets are widely available, *operation traces* — i.e., the sequence of operations applied to a dataset with properly labeled timestamps — are rarely, if ever, available in data processing. We resort to the standard practice in database evaluation and generate both *read-only* traces and *read-write* traces for our experiments, each consisting of 10,000 operations. The specific operations generated for each dataset, and their composition, can be found in Table 2. For each dataset, our selected operations address common use scenarios. For each generated trace, the order of operations is randomized, conforming to the composition ratio specified in the Table.

We evaluate PitStop with two distributions on the target nodes of the operations. By default, the target nodes of individual operations are uniformly random among data units in the database. Alternatively, we experimented with a trace (§ 6.7) where the target nodes of operations follow the power-law distribution, i.e., some nodes are queried or updated more frequently than others.

	UP phase	OP phase
F1	[0, 10ms]	[90ms, 100ms]
F2	[0, 10ms]	[190ms, 200ms]
F3	[0, 10ms]	[290ms, 300ms]

Fig. 3. Fluctuation Scenario Settings

Workload Fluctuation Settings. We model workload fluctuation through adjusting the arrival rates of operations, alternating them between UP and OP phases. In a UP (or OP) phase, operations arrive at a rapid (or slow) rate. Concrete to our experimental setting, the first 2500 operations (1/4 of total operations) are issued in the UP phase, the next 2500 operations are issued in the OP phase, and so on. We experiment with 3 arrival rate settings, described in Table 3. For example, fluctuation setting F1 says that during the UP phase, the time interval between the two operation arrivals is randomly and uniformly set to be between 0 and 10ms, and during the OP phase, the interval is similarly set to be between 290ms and 300ms. The concrete values of the three settings are determined heuristically as the highest arrival rate where the cascading effect is not observed, intuitively the rate for "break-even" provisioning. This rate on average, across experiments and data sets, is the median between the two ranges of F1.

Experiments and Visualization. Each experiment is the average of 5 trials. While plotting the per-operation latency graphs, we further average out the results of every 100 consecutively arriving operations and represent the average.

Development Effort. PitStop is implemented in about 3500LOC. There are also additional 3000LOC for PitStop-I (in § 7.1), various baselines and comparative variants, and scripts for experiments.

6 Experimental Validation

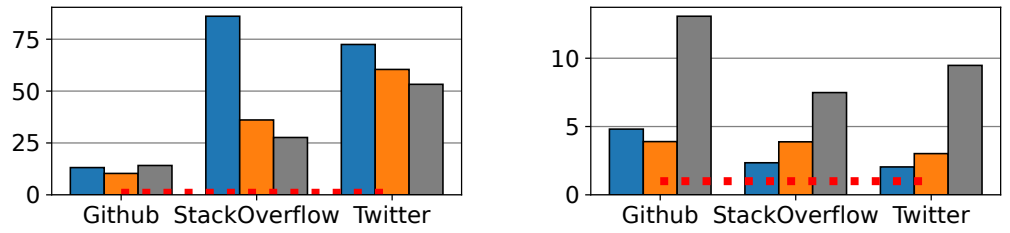
Our evaluation aims at answering the following questions:

- **RQ1:** What is the impact of PitStop on latency (§ 6.2) and throughput optimization (§ 6.3)?
- **RQ2:** What is the impact of PitStop on mitigating latency variance (§ 6.4)?
- **RQ3:** How does PitStop perform under alternative settings (§ 6.5, § 6.6, § 6.7, § 6.8)?
- **RQ4:** How do the design features contribute to the effectiveness of PitStop (§ 6.9)?
- **RQ5:** How does PitStop perform when PitStop is integrated with indexing (§ 7.1)?

6.1 Comparative Baselines

To demonstrate the effectiveness of PitStop, we construct two baselines. The first baseline is MP. This is the default implementation of Neo4j with parallelism support. A second baseline we introduce is *Buffered Monolithic Processing* (BMP), as an optimized form of MP, where batching and fusion may happen at the client-database boundary, the "top-level" buffer. Our BMP implementation is *de facto* collaborative scanning [65], closest to Crescando [24, 58], except that those SQL queries are now graph queries.

As we discussed in § 3, a BMP-like system must grapple with the decision on voluntary wait. We set two parameters for BMP: the targeted batch size before the batch is served B, and the maximum wait time for the batch to be formed, i.e., timeout, WT. We adopt the common rationale and set $B = \lceil \frac{WT}{AT} \rceil$, where AT is the average arrival time. In other words, the targeted batch size is the expected number of operations that would be received in the WT time (the second case). In this paper, we set $WT = 750ms$, and AT is set by experiments of different workload fluctuation: $\frac{5ms+95ms}{2}$ for F1, $\frac{5ms+195ms}{2}$ for F2, and so on.



(a) PitStop over MP

(b) PitStop over BMP

Fig. 4. A Summary of PitStop Latency Speedup (the Y-axis is the latency speedup ratio of processing operations with PitStop against the two baselines, averaged over the 10,000 operations described in § 5. With each group, the 3 bars correspond to the fluctuation scenarios of F1, F2, F3, in that order. The dotted horizontal line is the break-even latency, i.e., 1×. For each bar, the higher the better.)

Parallelism is in place for both baselines. Neither is a naive single-threaded implementation. For read-only traces, the processing of operations are executed in parallel. For read-write traces, non-exclusive read locks and exclusive write locks are in place. For MP, BMP, and PitStop, the same thread pool implementation of PitStop is used for thread management.

6.2 Latency Speedup

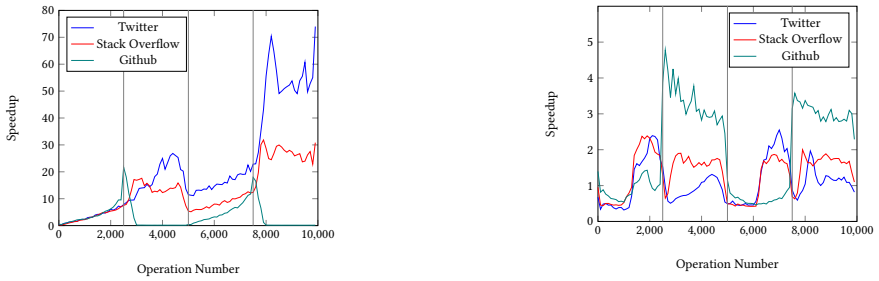
Summary. We first summarize the impact of PitStop on the operation latency, i.e., the "end-to-end" processing time between the arrival of the operation and the completion of its processing. Fig. 4 shows the average latency speedup of PitStop over the two baselines under various fluctuation scenarios. Overall, PitStop outperforms both baselines for all 3 datasets in all 3 fluctuation settings.

As shown in Fig. 4a, the more significant speedup is shown in the comparison with MP, the default parallel implementation of Neo4j. The poor performance by MP is due to the cascading effect of latency (§ 3). On the other hand, PitStop decouples the operation latency from the operation arrival order: a later arriving but computationally less complex operation may well be completed sooner than an earlier arriving operation. Thanks to this key difference in design, PitStop outperforms MP by at least 1 magnitude and occasionally near 2 magnitudes.

Fig. 4b shows that PitStop also outperforms BMP but at a lesser scale. Relative to MP, BMP softens the impact of the cascading effect by batching multiple operations together. However, just like MP, the processing of later-arriving operations cannot be started until a thread/core becomes available upon the *completion* of some earlier-arriving operation batches during the UP phase. A later-arriving but computationally less complex operation may still have the prolonged wait.

A Detailed Per-Operation Analysis. Fig. 5 presents a per-operation view of latency speedup. Recall that (§ 5) we alternate between UP and OP for the operation issue for each 2500 operations. With MP (Fig. 5a), MP quickly suffers from the cascading effect in the first UP phase, and the PitStop latency speedup quickly increases. When the system moves to the OP phase (when operations `2501-5000` arrive), the cascading effect in MP gradually recedes, but how fast this can happen depends on the size of data. For smaller datasets such as Github, the recovery period is shorter. For larger datasets, such as Twitter however, the recovery is so long that the next UP phase (when operations `5001-7500` arrive) starts before the recovery is fully complete.

One subtle observation is that the peak PitStop speedup often happens for an operation *arriving* in the OP phase, not exactly the last operation in the UP phase (such as operation 2500). Earlier, we have explained PitStop's role in mitigating the cascading effect in MP, i.e., "softening the blunt". If



(a) PitStop over MP

(b) PitStop over BMP

Fig. 5. PitStop Per-Operation Latency Impact (Results are under fluctuation setting F1, with the gray vertical lines indicate the phase change at the time of operation issuance. For the Y axis, higher is better.)

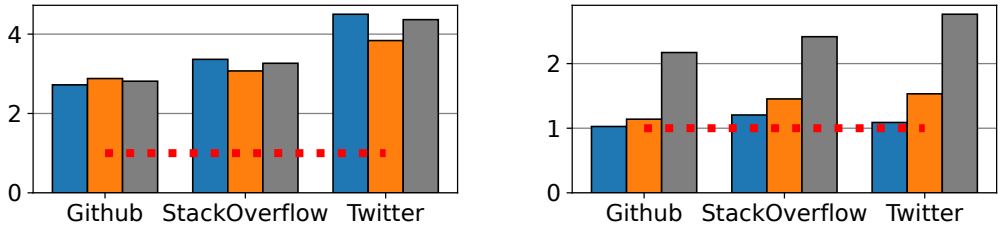
this *were* the only effect of PitStop, recall however, this is not the only effect of PitStop: when the system transitions from the UP phase to the OP phase, PitStop also has the interesting role of "speeding up the recovery". After UP switches to OP, the arrival rate of the subsequent operations will drop sharply. As a result, all operations that have arrived but not completed are more likely to be (randomly) selected by PitStop for propagation. PitStop's role in speeding up the recovery may further drive up the effectiveness of PitStop relative to MP.

The per-operation latency of PitStop relative to BMP is shown in Fig. 5b. At the beginning of each UP phase, BMP is indeed effective: it batches up a number of operations and assigns the processing to an available core/thread. The turning point however is that when all cores/threads become occupied. At this point, each BMP data processing task — even more coarse-grained than an MP data processing task due to batching — may take longer time to complete, and the cascading effect similar to MP occurs. This is why the relative effectiveness of PitStop increases often in the middle of each UP phase. In summary, BMP is friendly at the onset of an UP phase when there are still available cores/resources, but unfriendly when cores become saturated: at this point, batching makes BMP even coarser-grained than MP. In other words, not only the query processing is run-to-completion, and BMP requires multiple operations in the same batch to run to completion as one unit of concurrency. The coarsening here is prone to load imbalancing, leading to inferior performance of BMP as the UP phase continues on.

The Trend Across Fluctuation Scenarios. As we move from F1 to F3, the effectiveness of PitStop over MP generally decreases (Fig. 4a), whereas the effectiveness of PitStop over BMP generally increases (Fig. 4b). The difference between the three fluctuation settings is F1/F2/F3 has the shortest/medium/longest recovery period in each OP phase. As the OP phase becomes longer, MP is more likely to recover from its cascading effect. For BMP however, a longer OP phase implies fewer opportunities for batching. These opposite trends highlight the applicability difference among the 3 systems: MP is friendly for OP, BMP is friendly for UP, and PitStop has the dual impact of "softening the blunt" in UP and "speeding up the recovery" in OP, and thus strikes a balance in mitigating workload fluctuation.

6.3 Throughput Optimization

In online data processing where operations *continuously* arrive and get processed, the meaningful throughput is *rolling throughput*, i.e., the number of processed operations based on a rolling window of time. Fig. 6 shows the peak rolling throughput where the time window is set as 100 seconds.



(a) PitStop Peak Throughput over MP (b) PitStop Peak Throughput over BMP

Fig. 6. A Summary of PitStop Peak Throughput (The Y-axis is the peak throughput of processing operations with PitStop against the two baselines, for the trace of 10,000 operations described in § 5. For each bar, the higher the better. The other conventions are identical to Fig. 4.)

As shown here, PitStop outperforms MP by 1.8×-4.5× and BMP by 1×-2.4×. Unlike the pronounced latency impact, PitStop occasionally does not have a positive impact on throughput. For example, for the two fluctuation settings F1 and F2, PitStop breaks even in throughput with BMP for the dataset of Github. Among the 3 datasets, Github is the smallest. Relatively, the benefit PitStop brings in on latency reduction has limited impact on the throughput; instead, BMP shines for its natural friendliness for increasing peak throughput: every operation in a batch completes at the same time, forming a "peak" for the enclosing time window in which the rolling throughput is calculated. In a way, the real story — and a pleasant surprise — is that PitStop can outperform BMP in peak throughput in most cases.

6.4 Latency Variance Mitigation

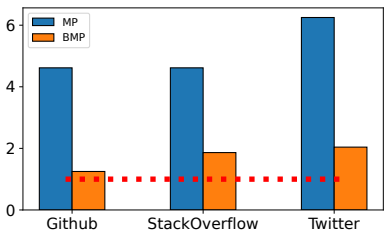


Fig. 7. Average Normalized Standard Deviations of Operation Latency for PitStop (For each dataset, MP and BMP refer to PitStop results normalized against MP and BMP respectively. Results are averaged across 3 fluctuation settings. A value greater than 1 means PitStop has less variance than the comparative baseline.)

The less obvious feature is that PitStop exhibits *less* latency variance in most comparative experiments. As shown in Fig. 7, the standard deviation of PitStop is significantly smaller than that of MP in all scenarios, and the standard deviation of PitStop is smaller than BMP for almost all scenarios. Note that latency variance among operations is inherent: each operation has different amount of work. What this comparison shows is that MP and BMP may further amplify the variance, likely resulting from a combination of the cascading effect and less balanced work load.

6.5 Alternative *LDSIZE* Settings

We experimented PitStop with different partition sizes by altering *LDSIZE*, with results shown in Fig. 8. Overall, *LDSIZE* has limited impact on PitStop performance when there are tens or (low) hundreds of pitstops in data. In this range, tuning *LDSIZE* may have ±10% difference, relatively small to the difference due to core features (MP vs. PitStop or BMP vs. PitStop). The key take-away message is that the effectiveness of PitStop is not hinged on a "fortunate" `LDSIZE` choice: a wide range of values can work, and if a thorough tuning is performed, PitStop may even have a 10% headroom for further speedup. Performance

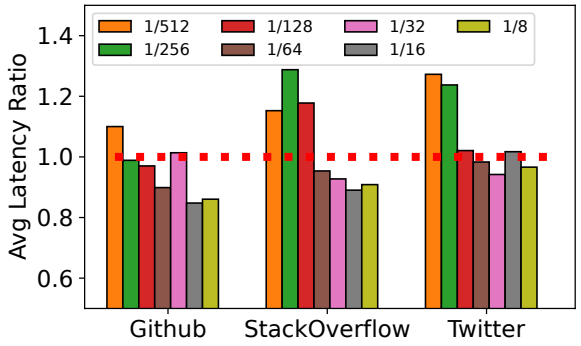


Fig. 8. PitStop Under Different *LDSIZE* (For $1/x$ in the legend, LDSIZE equals to $1/x$ of the data size. Each bar represents the average latency of all operations, normalized against the default PitStop setting, i.e., *LDSIZE* is 1/100 of the graph size. Results are under F1.)

impact becomes more pronounced when *LDSIZE* is set outside the range above. On one end of the spectrum, when *LDSIZE* is set as small as 1/512 of the graph size, latency degradation becomes more visible in Fig. 8. We will revisit another extreme on the other end of the spectrum where *LDSIZE* is set to be as large as the size of the graph, in § 6.9.

6.6 Alternative Trace Compositions

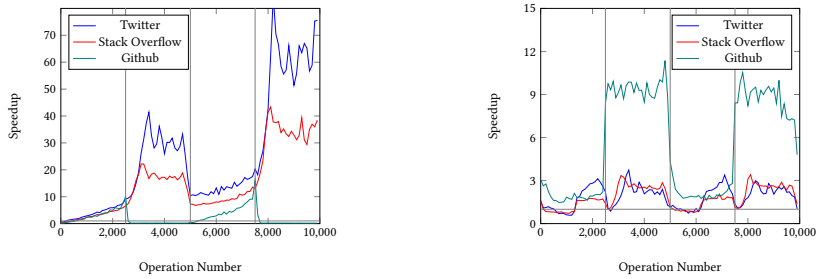
	Twitter	Stack Overflow	Github
PitStop	40.05	54.88	176.70
BMP	2.71	3.19	14.50
MP	0.45	1.93	6.25

Fig. 9. Throughput (ops/sec): the Read-Write Traces

We further conducted the same experiments for read-write traces. The throughputs of MP, BMP, and PitStop for the three datasets can be found in Fig. 9. With PitStop outperforming magnitudes better, the latency graphs become uninteresting: the speedups rapidly increase. What matters here is PitStop is friendly for leg parallelism, independent of whether the operation is `query` or `update`. The throughput data for read-write traces for PitStop and the read-only traces for PitStop are on par. A more advanced locking support for MP and BMP may make them more efficient, but unless such support is able to process read-write traces *equally efficiently* as read-only traces, it is unlikely to outperform PitStop, which enjoys *semantics-independent* leg parallelism.

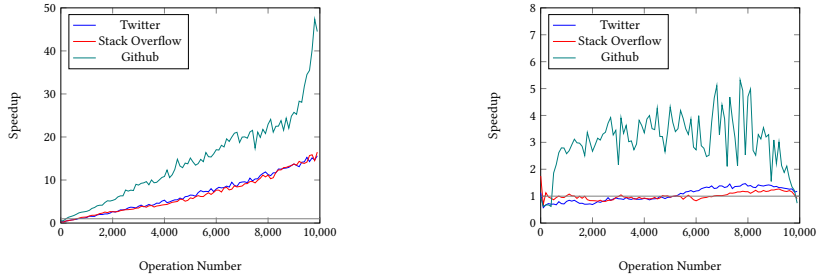
6.7 Alternative Operation Target Distribution

For the power-law traces, the experimental results can be seen in Fig. 10. Compared with the uniformly random trace, the results here follow the same general trend for the PitStop vs. MP comparison. More interestingly, our experiments show PitStop outperforms BMP at a larger margin for the power-law trace: Github for instance has a peak speedup of 3×-5× for uniform trace (see the main paper), but now has a peak speedup of 9×-11× for power-law traces. This highlights the challenges of determining wait time (WT) in the BMP baseline. When the target nodes of the operation trace are uniformly random, a fixed WT appears to make BMP a relatively performant baseline. When target nodes fall into a power-law distribution however, a pre-determined WT can no longer take into the account that the nodes targeted by the operations are "lopsided."



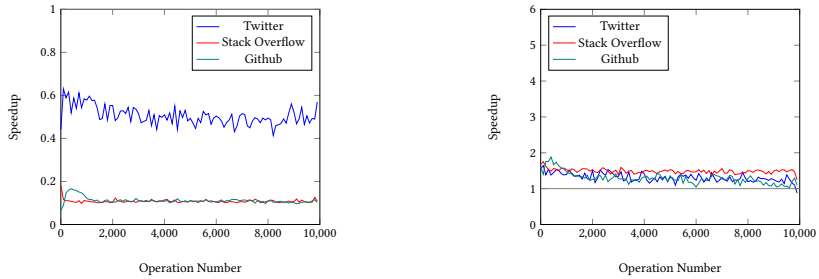
(a) PitStop over MP (b) PitStop over BMP

Fig. 10. PitStop Latency Speedup with Power-Law Traces with Fluctuation Setting F1



(a) over MP (b) over BMP

Fig. 11. PitStop Latency Speedup in Perpetual UP Scenarios (higher is better)



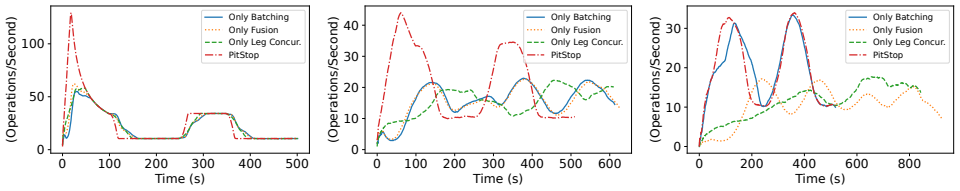
(a) Over MP (b) Over BMP

Fig. 12. PitStop Latency Speedup in Perpetual OP Scenarios (higher is better)

PitStop, however, remains resilient in the presence of the power-law trace thanks to its fine-grained parallelism, and more importantly, its lack of need for heuristically determining *WT* in the first place.

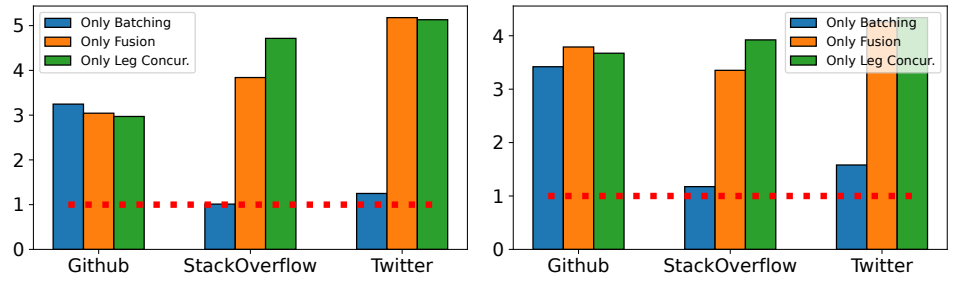
6.8 PitStop in Perpetual UP/OP Scenarios

PitStop is primarily designed for query language runtimes that may need to react to fluctuating workloads. In this section, we study how PitStop under two *extreme* scenarios: the *Perpetual UP*



(a) Github (b) Stack Overflow (c) Twitter

Fig. 13. Rolling Throughput Results of Ablation Studies (In each sub-figure, the X-axis is elapsed time, and the Y-axis is the rolling throughput. All are under F1. For the curve, the higher the better.)



(a) Average Operation Latency (b) Standard Deviation of Operation Latency

Fig. 14. Latency Results of Ablation Studies (a) Average Latency Normalized Against PitStop; (b) Average Standard Deviation of Operation Latency Normalized Against PitStop. (In both cases, the dotted line is the break-even value, i.e., 1×. A bar taller than 1× indicates the ablation variant is less effective than PitStop. All results are averaged across 3 fluctuation scenarios.)

scenario when PitStop *perpetually* receives more operations than it can process, and the *Perpetual OP* scenario when PitStop *perpetually* receives fewer operations than it processes.

The results for the *Perpetual UP* scenario can be found in Fig. 11. In near all time intervals, PitStop significantly outperforms MP: as time goes on, the cascading effect makes MP progressively worse. PitStop is as effective as BMP. Now that there are many more operations than a query language runtime can process, both PitStop and BMP have batching in play, leading to comparable performance.

The results for the *Perpetual OP* scenario can be found in Fig. 12. As shown here, PitStop is not as effective as MP, but more effective than BMP. The fact that MP is better should come as no surprise: now that the underlying system is often idle — i.e., with more computational resources than it needs — the simple execution semantics that each query is processed to completion works just fine. With PitStop however, the bookkeeping for pitstops and the additional maintenance of leg paralellism may introduce overhead. The more interesting observation here is that PitStop can outperform BMP. With BMP, the voluntary wait for forming batches is slowing down the query processing.

Our experiments in the *Perpetual OP* scenario can also be interpreted as an overhead analysis: it isolates the overhead by introducing the extreme scenario where there are only "pains" (of interruption and resumption) but no "gains" (i.e., the same interrupted operation will be resumed again). Recall that in § 1 and § 3, we discussed the applicability of PitStop: it is primarily designed for online data processing with workload fluctuation.

6.9 Ablation Study

We now study how design features in PitStop contribute to its effectiveness through an ablation study with 3 settings. First, the *Only Batching* setting supports the top-level buffer, but there are no pitstops inside the data (and hence no leg concurrency). Whenever a thread becomes available, all requests in the buffer are processed as a batch. This setting is similar to our baseline BMP, except that we do not form batches through bounded batch size and voluntary wait, and there is no fusion. In implementation, it is analogous to PitStop when *LDSIZE* is set as the graph size, and fusion is disabled. Second, the *Only Fusion* setting is also supported by the top-level buffer for arriving requests, and whenever a thread becomes available, it will fuse the requests in the buffer when possible, but no batching. Third, the *Only Leg Concur* setting allows pitstops to be placed inside the data, and leg concurrency is allowed, but no batching or fusion is allowed. All 3 settings support parallelism just as PitStop does. Their relative effectiveness is shown in Fig. 13 on throughput and Fig. 14a on latency.

Overall, PitStop consistently performs better than the ablated variants, and often significantly so. Fig. 13 shows that the peak throughput for PitStop is higher than all ablated variants. In Fig. 14a, 7 of the 9 variants of ablation have a latency slowdown of around 3× or more. It is the synergy of the three features that leads PitStop to a performance that none can achieve individually.

The *Batching Only* variant indeed can occasionally achieve competitive performance. For example, this ablation variant for `StackOverflow` is at 1.01× and `Twitter` at 1.25× in Fig. 14a, only slightly less competitive than PitStop. There are two additional reasons why PitStop remains favorable. First, PitStop remains effective in mitigating latency variance compared against the ablation settings (recall the same trend in § 6.4 when compared against MP and BMP). The comparative results against ablation settings are shown in Fig. 14b. Second, as we aim for a general solution neutral to data sets, PitStop exhibits more consistently superior performance across datasets.

6.10 Memory Consumption

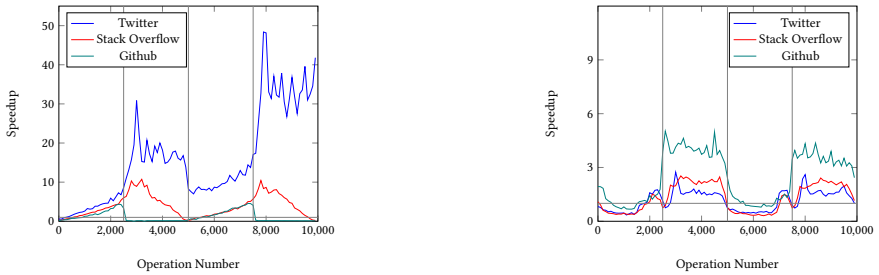
	peak memory consumption (**GB**)
PitStop	20.77 ± 0.15
BMP	20.77 ± 0.13
MP	21.25 ± 0.16

Fig. 15. Memory Consumption (`Twitter` under fluctuation setting `F1`)

PitStop has negligible impact on memory consumption, with the results for `Twitter` shown in Fig. 15. PitStop consumes statistically the same amount of memory as BMP, and consumes slightly less memory than the unmodified Neo4j (MP). This result should not come as a surprise. Recall in § 4 that the space complexity of PitStop is $O(n)$, where n is the number of operations. In other words, the only additional storage is to keep track of *which* pitstop each operation batch has propagated to. For our experiments of 10,000 operations, there are 10,000 entries, in KB-range data. This is dwarfed in 5-6 magnitudes by the memory consumption needed for large-scale data processing, e.g., more than 20GB in our experiments here. In other words, the deciding factor of memory usage becomes the overall efficiency of data processing itself. Finally, note that varying *LDSIZE* has no impact on memory consumption, as the number of operations remains the same.

6.11 A Reflexive Look on Data-Aware Concurrency

It is important to observe that while our comparative baselines (MP and BMP) are in operation, the underlying system already supports (semantics-oblivious) concurrency: the Java Virtual Machine (JVM) may optimize thread management, the OS may perform context switches, and the hardware may resort to Simultaneous Multithreading (SMT). These lower-layer concurrency mechanisms



(a) PitStop-I over MP

(b) PitStop-I over BMP

Fig. 16. PitStop-I Latency Speedup with Fluctuation Setting F1

remain unchanged in the PitStop experiments. In other words, our experiments show that leg parallelism can lead to *additional* performance benefits on top of a JVM/OS/hardware stack that already supports lower-level concurrency.

7 Extension and Discussion

PitStop is a fundamental redesign at the core of the query language runtime itself. In this section, we describe how the core ideas behind interruptible query processing may compose with the diverse set of existing techniques.

7.1 Extension: PitStop Integrated with Indexing

The PitStop algorithm can also be combined with existing index-based data access. For example, with hashing-based indexing, realizing simple key-value lookups can be completed in constant time assuming the index is pre-built. Indeed, indexing is not a panacea for data processing because not all operations can be processed through indexing. For example, queries can be dependent on structural patterns or may require aggregation among a large number of data. For the rest of the discussion, we refer to simple queries that can be processed through index lookup as *indexed operations* and those are not as *non-indexed operations*. We built a variant of PitStop where the PitStop algorithm is integrated with indexing, called PitStop-I.

The design of PitStop-I for read-only data access is trivial, because indexed queries and non-indexed queries can interleave without any concern on data consistency. For dynamic data access, PitStop-I can follow the similar compensation-based solution classic in indexing systems: (1) each indexed operation is first processed through the index, and the result is compensated by the latest non-completed operation targeting the same node (i.e., with the same index key). (2) each non-indexed operation is processed with the default PitStop algorithm.

For our datasets, we treat the "find user" operations in Twitter and StackOverflow, and "find pulls" and "find pushes" operations in GitHub as indexed queries, and the rest of operations as non-indexed queries. When PitStop-I is compared against baselines, MP and BMP are also integrated with indexed access. In other words, for the same indexed operations in PitStop-I, they will also be processed through indexing in MP and BMP.

The performance of PitStop-I is shown in Fig. 16. Observe that the same trend that we showed earlier for PitStop (without indexing) is preserved here: PitStop-I outperforms both MP and BMP significantly when data processing is in the UP phase. Relative to PitStop without indexing, the speed up for PitStop-I is occasionally less significant. This is not surprising because indexed

operations are usually processed magnitudes faster. Our results show that PitStop-I remains an effective solution in this setting due to its significant advantage in handling non-indexed operations.

7.2 Relational Data Processing

PitStop is evaluated on graph data. As our algorithm specification is neutral to the structure of data, we believe the core idea may also be built on top of relational data processing systems. The latter however come with distinct operators, whose support we now sketch.

Column projection can be directly supported by ours. Unlike our specification where each operation is realized at *one* data node, the projection operation can continue to propagate through all records, and continuously and incrementally collects the column data of interest until the last record is reached. When multiple projection operations are issued, each may collect the column of its interest, *at their own pace* through the propagation.

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. This operator is often used for aggregation. With an algorithm such PitStop, the aggregation function can be performed incrementally, so that when the propagation reaches the last record, the final result of aggregation is produced.

Table joins can be supported through nested operations. To support the join of table A and table B with column X, the join operation should propagate within the data of table A following the specification of PitStop. For each record it encounters, the join operation o_1 may issue another operation o_2, to be propagated and realized with the data of table B following the behavior of the PitStop specification. When o_2 completes the table B scan, the propagation of table A can now proceed to the next node. What the core algorithm of PitStop brings in is a natural flavor of *incremental* joins where processing this well-known expensive operator can be broken down into legs, both within table A scanning and table B scanning.

7.3 Programmable Operation Dependencies

In PitStop, data flow dependencies between operations are supported through the *target* of operations. For example, ℓ_2 may depend on ℓ_1 if both operate on the same target key, and ℓ_1 is an `update` and the subsequently issued ℓ_2 is a `query`. The key to ensuring the correct behavior in the presence of operation dependencies is chronological order preservation (see the end of § 3, and § 4). With an extension on the programming model of the operations itself, PitStop is also capable of supporting data flow dependencies where the result of an operation may be named as a variable, which subsequently occurs in another operation. To support this variant, the primary change is that substitution (i.e., replacing the variable name with the value result) may happen during the propagation. It is also possible some operation may never be realized (for example, two operations mutually depend on each other's result), which according to our algorithm specification, such an operation will return $\perp$ at the end of propagation. Thanks to the key property of chronological order preservation preserved by PitStop, the rest of the PitStop does not require additional changes.

7.4 Distributed Data Processing

To support distributed data, PitStop can be applied to each cluster node, where sequential consistency can be enforced *within* the cluster node. Enforcing sequential consistency for PitStop *across* the entire distributed cluster would be expensive, as it is for distributed data processing in general. Indeed, the key to scalable distributed data processing is to relax the consistency model [56]. With this expectation, PitStop can be extended to distributed data processing in a standard manner, where sequential consistency is preserved within each node, and relaxed consistency is achieved across nodes.

8 Related Work

Allowing a potentially long computation to be interrupted is a classic idea across the computing stack. The most well-known example is perhaps OS interrupts, which an OS thread can be interrupted by interrupt handlers so that high-priority system maintenance can be performed in a timely manner. In OS with cooperative scheduling, a thread can also be interrupted and context-switched when its share of time is completed. As recent examples, ITask [19] may interrupt a memory-intensive task upon memory pressure for partial memory reclamation. Shinjuku [36] supports preemptive scheduling between short requests and long requests to reduce tail latency. Intermittent computing [41] allows a computation to be interrupted due to resource constraints, such as insufficient power source, and resume later. SEDA [61] allows a network service divided into a network of stages through a programming model. As interruption and resumption are closely related, checkpointing systems (e.g., [38, 41, 45, 48]) are also technically relevant in this design space. On one hand, interruptible query processing shares the same high-level philosophy, incarnating the philosophy into the design of query runtimes. On the other hand, the specific and unique needs of different systems lead to different designs. In our case, our problem domain is online query processing. Our design goal is to enable scalable performance in the presence of workload fluctuation. Our solution is a form of incremental query propagation within data, coupled with leg parallelism. In other words, the analogy between a query runtime and, say a memory allocator or a battery-powered device, can only go to an extent. For example, checkpointing, a central problem in many systems cited above, is relatively straightforward in PitStop: each operation only needs to remember which pitstop it has reached so far.

Deferring query processing is the central idea of a number of query processing systems. Here, the most relevant guiding technique is lazy evaluation. For example, Sloth [13] is a compiler where a data processing request in a database-backed program is lazily evaluated, so that a traditional call-by-need style of program semantics can guide query execution, leading to reduced database round-trips. Lazy transactions [18] allows the execution of a transaction to be split into a now-phase and a later-phase. In neither system, the deferral happens *during* the data inspection, the key idea of PitStop. In spirit, PitStop is closer to incremental computing [4, 31, 33, 34, 47], in that the PitStop operations are incrementally propagated through data. Lineage-based programming [30] allows deferred computation at the boundary of distributed network nodes. We diverge on our design goals: theirs on fault tolerance in distributed data processing and ours on workload fluctuation in cloud-based data processing. As a result, we do not overlap on other aspects of design and evaluation. KickStarter [60] addresses continuous queries, i.e., the same (query) operation (e.g., returning the node with the largest payload value) is continuously processed over time but the underlying (graph) data continuously change. They have an incremental algorithm to determine whether the query needs to be recomputed based on what graph changes happen. In PitStop, the operations that arrive at the query runtime are arbitrary (Cypher) queries and updates, not (multiple instances of) the same query.

There is a large body of work on query optimization. Batching is a classic optimization in data processing. Cooperative scans [65] allow multiple queries to share a scan cursor in data processing, with a focus on efficient storage access. Crescando [9, 24, 58] is a collaborative data inspection system for in-memory data processing, where query-data joins and update-data joins are supported for operation processing in a batch. These systems do not allow the query processing to be interrupted; indeed, our comparative BMP baseline is the adaptation of Crescando to graph databases. Multi-query optimization (MQO) [52, 53] shares our belief that multiple requests should be considered as a whole for optimization, not in isolation. MQO approaches generally combine multiple queries together through compilation, and often identify common subqueries that can

be shared among queries. As a more recent exxample, SharedDB [23] compiles the database workloads into a global and optimized query plan. Query pipelining allows nested SQL queries [62], programmable queries [8], or multiple queries [32] to be compiled and processed in a pipelined manner, often in a vectorized execution. While pipelining is a good solution when the underlying system is in a perpetual UP state, it is not known as a solutoin for addressing workload fluutation. Overall, PitStop is orthogonal to MQO approaches other than the high-level philosophy: two central ideas of our approach — incremental propagation and fine-grained parallelism — are not part of the MQO design space.

Actors [5, 35] is a foundational model that leads to many influential language designs that support fine-grained concurrency. Java virtual threads, Go routines, and CML [49] are widely known examples in realistic languages. These features provide natural vehicles for implementing fine-grained concurrency, but to achieve scalable performance, additional runtime support is needed, with examples such as work stealing [22], MultiMLton [54], and Scala Akka [6]. The idea of fine-grained parallelism has also influenced the design of many computer systems with scalablity as a concern, such as AWS Lambda, ActOP [44], and PLASMA [51]. In this context, PitStop is a runtime design that harvests the semantic information of query processing for achieving fine-grained concurrency. In data processing, data streaming systems—often enabled by data flow and data streaming languages [7, 10, 43, 55, 57, 57, 59]—are also known for their parallelism support. Conceptually, the design space explored by data streaming systems and PitStop are *duals*. In PitStop, operations flow through (different pitstops of the potentially large) data whereas in data streaming systems, data flow through (often networks of) operations. The duality between operations-flowing-through-data and data-flowing-through-operations was articulated before [15].

DON calculus [15] lays a formal foundation for online data processing. It defines a spectrum of online data processing systems, focusing two core features: (1) incremental operation processing: a (query/update) operation can be incrementally propagated through the data, and (2) temporal locality optimization: multiple operations issued near the same time may be optimized through term rewriting. Most notably, they show that under certain conditions (which they call phase distinction), a determinism property holds which says that the result of online data processing of a sequence of operations — despite the non-deterministic executions due to incremental processing and temporal locality optimization — remains identical to that from sequential processing the sequence of operations. In that light, PitStop can be conceptually viewed as a concrete instance in that spectrum of online data processing algorithms. PitStop supports a more restricted form of incremental operation processing: whereas DON calculus allows an operation to be incrementally propagated to *any* node in the data, PitStop only allows the incremental propagation to pause at pitstop nodes. PitStop supports batching and fusion, which are perhaps the most commonly used "temporal locality" optimizations in practice; DON instead supports general term rewriting as optimizations which may go beyond batching and fusion. The phase distinction condition of their calculus requires that a query cannot generate another query during the query processing process; this condition trivially holds for Cypher. DON calculus does not support parallelism, a centerpiece of our work. Lacking experimental evaluation, DON calculus confirmed that a family of online data processing algorithm can be *correctly* designed, without unknown experimental effectiveness. PitStop identifies that workload fluctuation and long-tail are compelling use scenarios for incremental propagation in query processing.

Broadly, mitigating workload fluctuation is a fundamental problem in computer systems which can be addressed at different layers. In data centers, the most established route is dynamic provisioning [12, 28], often coupled with workload analysis and prediction [20, 25]. In these systems, a more accurate prediction on the workload enables dynamic provisioning of computation resources, often through turning on/off servers. As powering servers on/off usually takes time (in seconds or

minutes), such an approach is most effective in workload fluctuation settings with coarser time intervals, such as those with diurnal patterns. Another classic OS approach is to address workload fluctuation with dynamic resource allocation [11]. As perfect provisioning is not realistic, PitStop complements these approaches for scenarios where UP and OP may happen in practice, and by innovating over a unique aspect of the data processing engine, data inspection. Our experiments show PitStop is capable of addressing fine-grained workload fluctuations over a short duration of time.

9 Conclusion

PitStop is a novel query runtime designed for online data processing in the presence of workload fluctuation. The key idea is to view the otherwise monolithic query processing incrementally, where each step of incremental query propagation is enabled through fine-grained concurrency. PitStop satisfies a trio of competing goals in data processing: achieving scalable performance, processing dynamic data, and maintaining sequential consistency. PitStop address challenging use scenarios such as workload fluctuation and longtail, whose effectiveness is evidenced by experiments over real-world graph datasets on cloud servers.

In the future, we would like to extend the core idea of PitStop in several dimensions. First, we would like to build a distributed data processing system where PitStop runs within each node in the cluster, and relaxed consistency is enforced across nodes (§ 7.4). Another interesting direction is to achieve failure resilience and transparency. With atomicity (§ 3), disregarding a failed operation *per se* is less difficult: if a failure happens during its propagation, no persistent data is modified, and the failed operation can be removed from the pitstop it current resides. A more interesting design space lies in efficient checkpointing to speed up failure recovery. With our discussion in § 8 on the connection between PitStop and checkpointing, our speculation is that PitStop is already a more checkpointing-friendly system than MP: if pitstop information is preserved as checkpoints, data inspection of a recovered operation (after failure) can continue from the saved pitstop, instead from the beginning.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful suggestions and comments. This project is sponsored by the US NSF under CCF-1815949 and CNS-1910532.

Data Availability Statement

PitStop is an open-source project. The source code and experimental data are available [17]. The supplementary material can be found online [16].

References

[1] [n. d.]. Github Archive. http://www.githubarchive.org.
[2] [n. d.]. Neo4j Graph Database. http://www.neo4j.org.
[3] [n. d.]. Stack Exchange Data Dump. https://archive.org/details/stackexchange.
[4] Umut A. Acar, Guy E. Blelloch, and Robert Harper. 2006. Adaptive Functional Programming. *ACM Trans. Program. Lang. Syst.* 28, 6 (Nov. 2006).
[5] Gul Agha. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA.
[6] akka [n. d.]. Scala Akka, https://akka.io/.
[7] E. A. Ashcroft and W. W. Wadge. 1977. Lucid, a nonprocedural language with iteration. *Commun. ACM* 20, 7 (July 1977), 8.
[8] Peter A Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution.. In *Cidr*, Vol. 5. 225–237.
[9] George Candea, Neoklis Polyzotis, and Radek Vingralek. 2009. A Scalable, Predictable Join Operator for Highly Concurrent Data Warehouses. *Proc. VLDB Endow.* 2, 1 (Aug. 2009), 277–288.

[10] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. 1987. LUSTRE: a declarative language for real-time programming. In *POPL '87*. 178–188.

[11] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin M. Vahdat, and Ronald P. Doyle. 2001. Managing Energy and Server Resources in Hosting Centers. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP '01)*. Association for Computing Machinery, New York, NY, USA, 103–116. https://doi.org/10.1145/502034.502045

[12] Gong Chen, Wenbo He, Jie Liu, Suman Nath, Leonidas Rigas, Lin Xiao, and Feng Zhao. 2008. Energy-Aware Server Provisioning and Load Dispatching for Connection-Intensive Internet Services. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI'08)*. USENIX Association, USA, 337–350.

[13] Alvin Cheung, Samuel Madden, and Armando Solar-Lezama. 2014. Sloth: Being Lazy is a Virtue (when Issuing Database Queries). In *SIGMOD '14*. 931–942.

[14] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04*. San Francisco, CA, 137–150.

[15] Philip Dexter, Yu David Liu, and Kenneth Chiu. 2022. The essence of online data processing. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 899–928. https://doi.org/10.1145/3563320

[16] Jeff Eymer, Philip Dexter, Joseph Raskind, and Yu David Liu. [n. d.]. *Pitstop Supplementary Material (https://www.cs.binghamton.edu/~davidl/papers/OOPSLA24Sup.pdf)*. Technical Report.

[17] Jeff Eymer, Phillip Dexter, Joseph Raskind, and Yu David Liu. 2024. A Runtime System for Interruptible Query Processing: When Incremental Computing Meets Fine- Grained Parallelism - Artifact. https://doi.org/10.5281/zenodo.13372050

[18] Jose M Faleiro, Alexander Thomson, and Daniel J Abadi. 2014. Lazy evaluation of transactions in database systems. In *SIGMOD'14*. ACM, 15–26.

[19] Lu Fang, Khanh Nguyen, Guoqing Xu, Brian Demsky, and Shan Lu. 2015. Interruptible Tasks: Treating Memory Pressure as Interrupts for Highly Scalable Data-Parallel Programs. In *SOSP'15*. 394–409.

[20] Wei Fang, ZhiHui Lu, Jie Wu, and ZhenYin Cao. 2012. RPPS: A Novel Resource Prediction and Provisioning Scheme in Cloud Data Center. In *2012 IEEE Ninth International Conference on Services Computing*. 609–616. https://doi.org/10.1109/SCC.2012.47

[21] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs. In *SIGMOD '18*. 1433–1445.

[22] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI '98)*. Association for Computing Machinery, New York, NY, USA, 212–223. https://doi.org/10.1145/277650.277725

[23] Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. 2012. SharedDB: killing one thousand queries with one stone. *Proceedings of the VLDB Endowment* 5, 6 (2012), 526–537.

[24] Georgios Giannikis, Philipp Unterbrunner, Jeremy Meyer, Gustavo Alonso, Dietmar Fauser, and Donald Kossmann. 2010. Crescando. In *SIGMOD'10 (SIGMOD '10)*. 1227–1230.

[25] Daniel Gmach, Jerry Rolia, Ludmila Cherkasova, and Alfons Kemper. 2007. Workload Analysis and Demand Prediction of Enterprise Data Center Applications. In *2007 IEEE 10th International Symposium on Workload Characterization*. 171–180. https://doi.org/10.1109/IISWC.2007.4362193

[26] Jim Gray. 1988. *The transaction concept: virtues and limitations*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 140–150.

[27] Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Martin Schuster, Petra Selmer, and Hannes Voigt. 2019. Updating Graph Databases with Cypher. *Proc. VLDB Endow.* 12, 12 (Aug. 2019), 2242–2254.

[28] Brian Guenter, Navendu Jain, and Charles Williams. 2011. Managing cost, performance, and reliability tradeoffs for energy-aware server provisioning. In *2011 Proceedings IEEE INFOCOM*. 1332–1340. https://doi.org/10.1109/INFCOM.2011.5934917

[29] Theo Haerder and Andreas Reuter. 1983. Principles of transaction-oriented database recovery. *ACM Comput. Surv.* 15, 4 (dec 1983), 287–317. https://doi.org/10.1145/289.291

[30] Philipp Haller, Heather Miller, and Normen Müller. 2018. A programming model and foundation for lineage-based distributed computation. *Journal of Functional Programming* 28 (2018).

[31] Matthew A. Hammer, Khoo Yit Phang, Michael Hicks, and Jeffrey S. Foster. 2014. Adapton: Composable, Demand-driven Incremental Computation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*.

[32] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. 2005. QPipe: A Simultaneously Pipelined Relational Query Engine. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD '05)*. Association for Computing Machinery, New York, NY, USA, 383–394. https://doi.org/10.1145/1066157.

1066201

[33] Daco C Harkes, Danny M Groenewegen, and Eelco Visser. 2016. IceDust: Incremental and Eventual Computation of Derived Values. In *30th European Conference on Object-Oriented Programming*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[34] 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.

[35] Carl Hewitt, Peter Bishop, Irene Greif, Brian Smith, Todd Matson, and Richard Steiger. 1973. Actor Induction and Meta-Evaluation. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '73)*. 153–168.

[36] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for second-Scale Tail Latency *(NSDI'19)*. USENIX Association, USA, 345–359.

[37] Ron Kohavi and Roger Longbotham. 2007. Online Experiments: Lessons Learned. *IEEE Computer* 40 (10 2007), 103–105. https://doi.org/10.1109/MC.2007.328

[38] R. Koo and S. Toueg. 1987. Checkpointing and Rollback-Recovery for Distributed Systems. *IEEE Transactions on Software Engineering* SE-13, 1 (1987), 23–31.

[39] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *WWW '10*. 591–600.

[40] Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.* C-28, 9 (1979), 690–691. https://doi.org/10.1109/TC.1979.1675439

[41] Brandon Lucia, Vignesh Balaji, Alexei Colin, Kiwan Maeng, and Emily Ruppel. 2017. Intermittent Computing: Challenges and Opportunities. In *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA (LIPIcs)*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.), Vol. 71. 8:1–8:14.

[42] Eric Masanet, Arman Shehabi, Nuoa Lei, Sarah Smith, and Jonathan Koomey. 2020. Recalibrating global data center energy-use estimates. *Science* 367, 6481 (2020), 984–986. https://doi.org/10.1126/science.aba3758

[43] 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 (OOPSLA '09)*. ACM, New York, NY, USA, 1–20. https://doi.org/10.1145/1640089.1640091

[44] Andrew Newell, Gabriel Kliot, Ishai Menache, Aditya Gopalan, Soramichi Akiyama, and Mark Silberstein. 2016. Optimizing distributed actor systems for dynamic interactive services. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. Article 38, 15 pages.

[45] orangefs [n. d.]. The OrangeFS Project, http://www.orangefs.org/.

[46] J. Park and A. Segev. 1988. Using common subexpressions to optimize multiple queries. In *Proceedings. Fourth International Conference on Data Engineering*. 311–319.

[47] W. Pugh and T. Teitelbaum. 1989. Incremental Computation via Function Caching. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '89)*.

[48] B. Randell, P. Lee, and P. C. Treleaven. 1978. Reliability Issues in Computing System Design. *ACM Comput. Surv.* 10, 2 (June 1978), 123–165.

[49] John H. Reppy. 2007. *Concurrent Programming in ML* (1st ed.). Cambridge University Press, USA.

[50] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. 2000. Efficient and Extensible Algorithms for Multi Query Optimization. In *SIGMOD'00*. 249–260.

[51] Bo Sang, Pierre-Louis Roman, Patrick Eugster, Hui Lu, Srivatsan Ravi, and Gustavo Petri. 2020. PLASMA: programmable elasticity for stateful cloud computing applications. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*. Article 42, 15 pages.

[52] Timos K Sellis. 1988. Multiple-query optimization. *ACM Transactions on Database Systems (TODS)* 13, 1 (1988), 23–52.

[53] Timos K. Sellis. 1988. Multiple-Query Optimization. *ACM Trans. Database Syst.* 13, 1 (March 1988), 23–52.

[54] K. C. Sivaramakrishnan, Lukasz Ziarek, and Suresh Jagannathan. 2014. MultiMLton: A multicore-aware runtime for standard ML. *Journal of Functional Programming* 24 (2014), 613 – 674.

[55] 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 (OOPSLA '07)*. Association for Computing Machinery, New York, NY, USA, 211–228. https://doi.org/10.1145/1297027.1297043

[56] Andrew S. Tanenbaum and Maarten van Steen. 2006. *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice-Hall, Inc., USA.

[57] William Thies, Michal Karczmarek, and Saman Amarasinghe. 2002. StreamIt: A language for streaming applications. In *International Conference on Compiler Construction*. Springer, 179–196.

[58] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann. 2009. Predictable Performance for Unpredictable Workloads. *Proc. VLDB Endow.* 2, 1 (Aug. 2009), 706–717.
[59] 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.
[60] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2017. KickStarter: Fast and Accurate Computations on Streaming Graphs via Trimmed Approximations. In *ASPLOS '17*. 237–251.
[61] Matt Welsh, David E. Culler, and Eric A. Brewer. 2001. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *SOSP'01*, Keith Marzullo and Mahadev Satyanarayanan (Eds.). ACM, 230–243.
[62] A.N. Wilschut and P.M.G. Apers. 1990. Pipelining in query execution. In *Proceedings. PARBASE-90: International Conference on Databases, Parallel Architectures, and Their Applications*. 562–. https://doi.org/10.1109/PARBSE.1990.77227
[63] 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.
[64] Jingren Zhou and Kenneth A Ross. 2004. Buffering databse operations for enhanced instruction cache performance. In *SIGMOD'04*. ACM, 191–202.
[65] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter Boncz. 2007. Cooperative Scans: Dynamic Bandwidth Sharing in a DBMS. In *VLDB '07*. VLDB Endowment.

Received 2024-04-06; accepted 2024-08-18