Shared Arrangements: practical inter-query sharing for streaming dataflows

Paper #314

Abstract

Current systems for data-parallel, incremental processing over high-rate streams isolate the execution of independent queries. This creates unwanted redundancy and overhead in the presence of concurrent queries: each query must independently maintain the same indexed state over the same input streams, and new queries must build this state from scratch before they can begin to emit their first results.

This paper introduces *shared arrangements*: indexed views of maintained state that allow concurrent queries to reuse the same in-memory state without compromising data-parallel performance and scaling. We implement shared arrangements in a modern stream processor and show dramatic improvements in query response time and resource consumption for interactive queries against low-latency streams, while also significantly improving performance in other domains including business analytics, graph processing, and program analysis.

1 Introduction

We present *shared arrangements*, a new technique for sharing consistent state and computation between the operators of multiple concurrent data-parallel streaming dataflows. Shared arrangements improve the efficiency of systems which execute multiple stream processing computations, for example to support interactive queries against event streams. They can also reduce the time required to install new queries, often by several orders of magnitude.

Centralized relational databases naturally use the same indexes to answer a broad range of different queries, and can even support sophisticated multi-query optimization (MQO) [12, 13]. In contrast, data-parallel stream processors like Flink [8], Spark Streaming [32], and Naiad [20] lack such sharing. While they do support common sub-queries and record streams, they are unable to share the *indexed* representations of these streams that operators require to execute efficiently.

Instead, independent operators re-process these shared streams, expending communication, computation, and memory to produce and maintain private indexed representations of their contents. For example, while multiple queries joining against an evolving social graph may reuse the same stream of graph updates, each join operator produces and maintains an independent indexed representation of the social graph. For the same reason, a new query must first read and index the graph before it can start producing results. This setup work can take orders of magnitude longer than answering the query from existing maintained indexes, and queries' separate indexes introduce costly duplication of data and computation.

Shared arrangements address this inefficiency by making it possible for dataflow operators, and consequently queries, to share indexed state. The arrangement abstraction is a maintained index shared between a single writer and multiple readers. Using the shared index, arrangement-aware versions of familiar dataflow operators can process updates more efficiently than traditional record-at-a-time operators would.

The key challenge for shared arrangements is balancing sharing against the need for coordination in the dataflow. In the scenarios we target, logical operator state is spread across multiple, parallel physical operators; sharing such state between multiple queries could require global synchronization. Arrangements solve this challenge by carefully structuring how they share data: they *(i)* hard-partition shared state between workers and move computation to it, and *(ii)* multiversion shared state to allow operators to interact with it at different times and rates.

A shared arrangement is a sharded, multiversioned index of update events, which records the evolution of intermediate results at some point in a dataflow graph. A new `arrange` operator maintains the arrangement as updates arrive, and ensures high throughput, low latency, and a compact memory footprint. Other operators read from the shared arrangement to get consistent views of the state at different logical times.

Shared arrangements can be applied to many modern stream processors, but we used them to implement Differential Dataflow [19] in a new streaming engine, SysX, built on an existing implementation of Timely Dataflow [1]. We show in § 7 that shared arrangements dramatically reduce interactive query latency, by up to three orders of magnitude over approaches that cannot share indexes between computations. For a streaming variant of TPC-H and a changing graph, shared arrangements also reduce update latency by 1.3–3× and reduce the memory footprint of the computation by 2–4×.

Overall, shared arrangements improve update throughput of most parallel dataflows, while reducing memory footprint and allowing additional new streaming queries to be deployed near-instantaneously in many cases (since the required initial state already exists). In most cases, the multiversioned index used by arrangements improves even single-query execution, suggesting that they are simply a better basis for passing tuples between dataflow operators.

2 Background and Related Work

We can characterize the design space for inter-query state sharing in dataflow systems in terms of (1) *what* is shared between queries, (2) how this shared state can be *updated*,

System class	Example	Sharing	Updates	Coordination
RDBMS	Postgres	**Indexed state**	**Record-level**	Fine-grained
Batch processor	Spark	Non-indexed collections	Whole collection	**Coarse-grained**
Stream processor	Flink	None	**Record-level**	**Coarse-grained**
Shared arrangements (SysX)		**Indexed state**	**Record-level**	**Coarse-grained**

Table 1. Sharing of indexed in-memory state, record-level update granularity, and scalability through coarse-grained coordination are mutually exclusive in current systems. Shared arrangements make it possible to combine these features in a single system.

and (3) the degree of *coordination* required to maintain it. Table 1 contrasts relational databases, batch processors, and stream processors with shared arrangements.

Relational databases like PostgreSQL [23] excel at answering a wide range of different queries over tables laid out according to a predefined schema. The use indexes to speed up access to records in the tables, turning sequential scans into point lookups. The database updates the index when the underlying data changes. This model is flexible and naturally shares indexes between different queries but requires coordination. Scaling this coordination out to many parallel processors or machines has proven difficult, and scalable systems consequently attempt to restrict coordination.

Classic parallel processing **"big data" systems** like MapReduce [11], Dryad [18], and Spark [31] rely only on coarse-grained coordination. For scalability in processing large, unstructured data sets, they avoid indexes and turn query processing into parallel scans of distributed collections. These collections are immutable: any modification to a distributed collection (such as a Spark RDD) requires reconstituting that collection as a new one. This captures a collection's lineage and makes all parallelism deterministic, which eases recovery from failures. Immutability also allows different queries to share the (static) collection for reading [17]. These tradeoffs help systems scale out, but are a poor fit for streaming computations, in which fine-grained changes frequently update the collection data.

Stream-processing systems reintroduce fine-grained mutability, and incrementally maintain intermediate state in a long-running computation, but lack sharing. Systems like Flink [8], Naiad [20], and Noria [14] keep long-lived, indexed operator state in memory to enable efficient incremental processing, but associate each piece of state *exclusively* with a single operator, since concurrent accesses to this state from multiple operators would race with state mutations. Consequently, these systems *duplicate* the state that multiple operators could, in principle, share. This allows independent queries and operators to execute in isolation, but increases write processing cost and space footprint.

In contrast with these three classes of system, **shared arrangements** allow for fine-grained updates to shared indexes, while preserving the scalability of data-parallel dataflow computation. In particular, shared arrangements rely on primitives (multiversioned indices and data-parallel sharding) that allow updates to shared state without the costly concurrency control mechanisms of classic databases or distributed shared memory. In exchange, shared arrangements give up some abilities: unlike relational databases, they do not support fine-grained transaction processing, and because sharing entangles different queries that would otherwise have executed in isolation, it gives up the performance isolation and fault isolation between queries that the redundancy in big data systems achieves.

The work most related to shared arrangements is perhaps STREAM [5], a relational stream processor which maintains "synopses" (often indexes) for operators and shares them between operators. Shared arrangements can be seen as data-parallel, multiversioned STREAM synopses. Unlike STREAM, shared arrangements also reveal the synopsis structure (a log of indexed batches) to operators, which are able to take advantage of this representation.

Many other relational and big data systems exhibit qualitatively different types of sharing. Relational engines like CJoin [7] and SharedDB [13] share table scans to more efficiently retire queries over large, unindexed tables in data warehousing contexts. Both relational and big data systems can identify common sub-expressions and either cache their results or fuse their computation; for example, Nectar [17] does so for DryadLINQ [30] computations. Stream processors like TelegraphCQ [9] share state among continuous queries, but sequentially process each query without parallelism or shared indexes. Noria [14] supports parallel dataflow processing, and shares overlapping prefixes of the dataflow graph between queries, but still maintains per-operator indexed state. "Upqueries" from downstream operators within the same dataflow can read from these indexes to derive missing downstream state, but the absence of multiversioning imposes complex restrictions on Noria's upqueries and often requires duplicate indexes (e.g., within join chains).

Shared arrangements, by contrast, allow for operators fundamentally designed around shared indexes. Their ideas are, in principle, compatible with many existing stream processors that provide versioned updates (as e.g., Naiad and Flink do) and support physical co-location of operator shards (as e.g., Naiad and Noria do).

3 Context and Overview

Shared arrangements are designed for use in streaming dataflow computations. Data-parallel stream processing systems express such computations as a dataflow graph whose vertices are *operators*, and whose roots constitute *inputs* to the dataflow. An *update* (e.g., an event in stream) arrives at an input and flows along the graph's edges into operators. Each operator takes the incoming update, processes it, and emits any resulting derived updates.

In processing the update, a dataflow operator may refer to its *state*: long-lived information that the operator maintains across invocations. State allows for efficient incremental processing, such as keeping a running counter. For many common operators, the state can be indexed by a *key* contained in the input update. For example, a `count` operator over tweets grouped by the user who posted them will access its state by user ID. It is these indexes that shared arrangements seek to share between multiple operators.

Dataflow systems achieve parallel processing by *sharding* operators whose state is indexed by key. The system partitions the key space, and creates operators to independently process each of these partitions. In the tweet counting example, the system may partition updates by the user ID, and send each update to an appropriate operator shard, which maintains an index for its subset of user IDs. Each operator shard maintains its own private index, which taken collectively represent the same index a single operator would maintain. Shared arrangements must share these sharded indexes between multiple correspondingly sharded operators.

Shared arrangements apply in this general dataflow setting, but we build on specific prior work for coordination and incrementally-maintained state, as described in the following.

3.1 Timely Dataflow

Updates flow through a dataflow graph asynchronously. Concurrent updates may race along the multiple paths (and even cycles) between dataflow operators, and arrive in different orders than they were produced. For operators to compute correct results in the face of this asynchrony, some coordination mechanism is required. Timely Dataflow [1, 20] is a dataflow model that provides such a coordination mechanism using logical times.

Timely Dataflow is a model for data-parallel dataflow execution, introduced by Naiad [20]. It provides a dataflow abstraction in which nodes house operator logic, and edges transport data from the outputs of operators to the inputs of other operators. All data in Timely Dataflow bear a partially ordered logical timestamp, and operators are obliged to maintain (or advance) these timestamps as they process data. Timely Dataflow graphs may have cycles, where timestamps are augmented with an iteration count to correctly track progress within the loop.

Timely Dataflow schedules work on a static set of *workers*, each a single thread of control. All operators are sharded across all workers, and each worker multiplexes its time between each dataflow and dataflow operator. Workers schedule operator shards in response to the arrival of data, which are routed among workers according to functions the operators specify for each of their input streams (e.g. a function of a key in the record, to ensure all records with the same key arrive at the same worker). Crucially, for our purposes, we can co-locate on the same worker those operator shards that might profitably share the same indexed representation of their input data.

Timely Dataflows provide coordination information to their operators in the form of a *frontier*: a set of logical times such that all future timestamps must be greater than or equal to some element of the frontier. In Timely Dataflow, a frontier only ever advances and the set of times that an operator shard may yet see strictly decreases. This progress information tells operator shards when they have received all records with certain timestamps, at which point it may be appropriate for the operator shard to take some action. An operator may, for example, choose to emit an output update for a timestamp t only once its has received all updates for t ensuring that its result for t no longer changes.

Importantly, frontier changes provide Timely Dataflow operators with coordination information, but they do not require an immediate reaction from the operator. As frontiers advance, their changes can be accumulated and supplied to operators in batches that have coarser granularity than the frontier changes themselves. Operators can then retire entire batches of logical times at once, and do not need to have their execution finely coordinated by the system. This opens the door to *multiversioned* shared arrangements, which encode shared state at multiple different logical times, rather than requiring coordination among the multiple readers.

3.2 Differential Dataflow

Efficient stream processing requires operators to do incremental work for each update. Differential Dataflow [19] builds on the Timely Dataflow model to support efficient incremental update processing over distributed collections.

Every update in a Differential Dataflow is a *triple* of the form *(data, time, diff)*. *data* is an arbitrary data element, such as a record, while *time* is a logical timestamp, and *diff* is an incremental value change associated with the update. A *collection trace* is the set of update triples *(data, time, diff)* that define a collection at any time t by the accumulation of those *(data, diff)* for which $time \leq t$.

The abstractions of triples and collection traces, in combination with Timely Dataflow's progress tracking, are sufficient to implement incremental versions of familiar operators over collections, such as `group`, `map`, `filter`, etc. These operators can maintain state indexed by *key*, a field in *data*. Shared arrangements seek to extend this model by sharing collection traces between different operators, which requires changes to the dataflow model, a new `arrange` operator, and updated operator implementations.

3.3 Shared Arrangements Overview

The high-level objective of shared arrangements is to share indexed operator state, both within a single dataflow and across

```
// Arrange evolving node and graph state.
let (nodes, edges) = timely::dataflow(|scope| {
    // evolving (node, name), maintained.
    let nodes = kafka::load("nodes.stream")
                  .arrange();
    // evolving graph edges, maintained.
    let edges = kafka::load("edges.stream")
                  .arrange();

    return (nodes.trace, edges.trace);
});
```

(a) A Timely Dataflow fragment that captures evolving node state and graph structure. The returned `nodes` and `edges` arrangements can be re-used elsewhere in another dataflow.

```
// Take a collection of (source, query_id) to
// (node, name, query_id) two hops from source.
timely::dataflow(|scope| {
    let nodes_arranged = scope.import(nodes);
    let edges_arranged = scope.import(edges);
    queries
        .join(&edges_arranged) // sharing!
        .map(|(src,qid,mid)| (mid,qid))
        .join(&edges_arranged) // sharing!
        .map(|(mid,qid,dst)| (dst,qid))
        .join(&nodes_arranged) // sharing!
        .inspect(|x| println!("{}", x));
});
```

(b) A Timely Dataflow fragment that re-uses the `nodes` and `edges` arrangements to compute and maintain node state within two edges of query nodes. The `join` operator matches input records `(x,y)` and `(x,z)` with a common `x` into triples `(x,y,z)`. As these relations reuse shared arrangements, the query produces results at once.

Figure 1. Example code for shared arrangements in our prototype. In this example, `nodes` contain pairs of user identifier and the user's full name, and `edges` are directed pairs of user identifiers. The two collections evolve concurrently. A second dataflow computes the identifiers and names of users within two edges of a set of source user identifiers (`queries`), and correctly maintains the results as the inputs change.

multiple concurrent dataflows. We assume that developers specify their dataflows using existing interfaces, but that they (or an optimizing compiler) indicate explicitly which dataflow state to share among which operators, e.g., as Figure 1 shows.

A shared arrangement exposes different *views* of the underlying state to different operators and therefore emulates, atop physically shared state, the separate indexes that operators would otherwise keep. An explicit, new `arrange` operator maintains the state and views, while downstream operators read from their respective views. The contents of these views vary according to current logical timestamp frontier at the different operators: for example, a downstream operator's view may not yet contain updates that the upstream `arrange` operator has already added into the index.

Downstream operators in the same dataflow, and operators in other dataflows operating in the same logical time domain, can share the arrangement as long as they use the same key as the arrangement index. In particular, sharing can extend as far as the next `exchange` operator (which changes the key), an arrangement-unaware operator (e.g., `map`, which may change the key), or an operator that explicitly materializes a new collection. Sharing a key means that these operators exist on the same logical dataflow shard, and the system can exploit this fact to increase efficiency of sharing. Co-locating the operators for shared processing by the same thread makes access to shared physical state cheap, as it requires no locks or network communication. It does come at the cost of a reduced granularity of parallelism, but this tradeoff in our experience is nearly always worthwhile.

4 Shared Arrangements

A shared arrangement consists of three key components:

1. a *trace* of immutable, indexed batches that together make up the multiversioned index;
2. an `arrange` operator, which maintains the trace; and
3. arrangement-aware operators that access the shared index through *handles* to the trace.

Logically, an arrangement is a pair of *(i)* a stream of shared indexed batches of updates and *(ii)* a shared, compactly maintained index of all updates. Physically, an arrangement partitions its input collection among workers, each of which are responsible for creating batches, maintaining an update index, and sharing them to operators hosted by the same worker. Arrangements spend the communication, computation, and memory required to arrange data once, and then reuse the arrangement from within multiple operators.

Figure 2 depicts a dataflow which uses an arrangement for the `count` operator, which must take a stream of *(data, time, diff)* updates and report the changes to accumulated counts for each *data*. This operation can be implemented by first partitioning the stream among workers by *data*, after which each worker maintains an index from *data* to its history, a list of *(time, diff)*. This same indexed representation is what is needed by the `distinct` operator, in a second dataflow, which can re-use the same partitioned and indexed arrangement rather than re-construct the arrangement itself.

4.1 Collection traces

As in Differential Dataflow (§3.2), a *collection trace* is the set of update triples *(data, time, diff)* that define a collection at any time t by the accumulation of those *(data, diff)* for which *time* $\leq t$. A collection trace is initially empty and is only revealed as a computation proceeds, determined either as a dataflow input or from the output of another dataflow operator. Although update triples may arrive continually, it is only when the Timely Dataflow input frontier advances that the `arrange` operator learns that the updates for a subset of times are now fully committed.

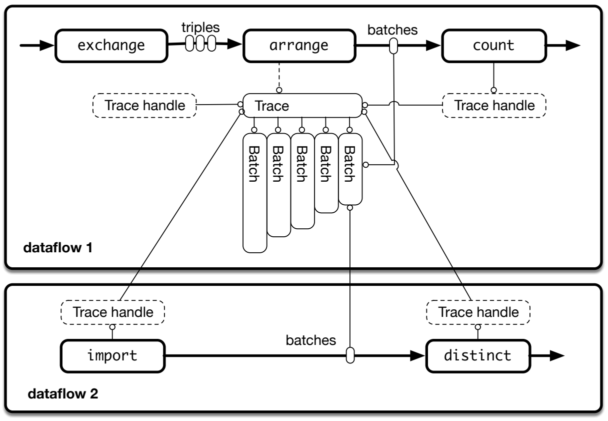


Figure 2. A worker-local overview of arrangement. Here the arrangement is constructed for the `count` operator, but is shared with a `distinct` operator in another dataflow. Each other worker performs the same collective data exchange, followed by local batch creation, trace maintenance, and sharing.

In our design a collection trace is logically equivalent to an append-only list of immutable batches of update triples. Each batch is described by two frontiers of times, *lower* and *upper*, and the batch contains exactly those updates whose times are in advance of the lower frontier and not in advance of the upper frontier. The upper frontier of each batch should match the lower frontier of the next batch, and the growing list of batches reports the developing history of committed updates triples. A batch may be empty, which indicates that no updates exist in the indicated range of times.

To support efficient navigation of the collection trace, each batch is indexed by its *data* to provide random access to the history of each *data* (the set of its (*time*, *diff*) pairs). A trace maintains relatively few (logarithmically many) batches by continually merging existing batches, ensuring that operators can efficiently navigate the union of all batches.

Each reader of a trace holds a *trace handle*, which acts as a cursor that can navigate the multiversioned index *as of any time* in advance of a frontier the trace handle holds. As readers advance through time, by advancing the frontier of their trace handle, the `arrange` operator is able to compact batches by coalescing updates at indistinguishable times, maintaining a memory footprint proportional to the size of the collection.

4.2 The `arrange` operator

The `arrange` operator receives update triples, and must both create new immutable indexed batches of updates as its input frontier advances and compactly maintain the collection trace without violating its obligations to readers of the trace.

At a high level, the `arrange` operator buffers incoming update triples until the input frontier advances, at which point it extracts and indexes all buffered updates not in advance of the newly advanced input frontier. A shared reference to this new immutable batch is both added to the trace and emitted as output from the `arrange` operator. When adding the batch to the trace, the operator may need to perform some maintenance to keep the trace representation compact and easy to navigate.

Amortized trace maintenance: The maintenance work of merging batches in a trace is amortized over the introduced batches, so that no batch causes a spike in computation (and a resulting spike in latency). Informally, the operator performs the same *set* of merges as would a merge sort applied to the full sequence of batches, but only as the batches become available. Additionally, each merge is not immediately completed; instead, for each new batch we perform an amount of work proportional to the batch size to each incomplete merge. A higher constant of proportionality leads to more eager merging, improving the throughput of the computation, whereas a lower constant improves the latency of the computation.

Consolidation: As readers of the trace advance through time, historical times become indistinguishable and updates at such times to the same *data* can be coalesced. The logic to determine which times are indistinguishable is present in Naiad's prototype implementation, but the mathematics of compaction have not been reported previously. In our extended technical report, we present proofs of optimality and correctness [4].

Shared references: Both immutable batches and traces themselves are reference counted. Importantly, the `arrange` operator holds only a "weak" reference to its trace, and if all readers of the trace drop their references the operator will continue to produce batches but cease updating the trace. This optimization is important for competitive performance in computations that use both dynamic and static collections.

4.3 Trace handles

Read access to a collection trace is provided through a *trace handle*. A trace handle provides the ability to `import` a collection into a new dataflow, and to manually navigate a collection, but both only "as of" a restricted set of times. Each trace handle maintains a frontier, and guarantees only that accumulated collections will be correct when accumulated to a time in advance of this frontier. The trace itself tracks outstanding trace handle frontiers, which indirectly inform it about times that are indistinguishable to all readers (and which can therefore be coalesced).

Many operators (including `join` and `group`) only need access to their accumulated input collections for times in advance of their input frontiers. As these frontiers advance, the operators are able to advance the frontier on their trace handles and still function correctly. The `join` operator is even able to drop the trace handle for an input when its *other* input ceases changing. These actions, advancing the frontier and dropping trace handles, provide the `arrange` operator with the opportunity to consolidate the representation of its trace.

A trace handle has a method `import` which creates an arrangement in a new dataflow exactly mirroring that of the trace. The imported collection immediately produces any existing consolidated historical batches, and begins to produce newly minted batches. The historical batches reflect all updates applied to the collection, either with full historical detail or coalesced to a more recent timestamp, depending on whether the handle's frontier has been advanced before it was used to import the trace. Computations do not require special logic or modes to accommodate attaching to in-progress streams; imported existing traces appear indistinguishable to their original streams, other than their unusually large batch sizes and recent timestamps.

5 Arrangement-aware operators

Operators act on collections, which can be represented either as a stream of update triples or as an arrangement. These two representations lead to different operator implementations, where the arrangement-based implementations can be substantially more efficient than traditional record-at-a-time operator implementations. In this section we explain arrangement-aware operator designs, starting with the simplest examples and proceeding to the more complex `join`, `group`, and `iterate` operators.

5.1 Key-preserving stateless operators

Several stateless operators are "key-preserving": they do not transform their input data to the point that it needs to be re-arranged. Example operators are `filter`, `concat`, `negate`, and the iteration helper methods `enter` and `leave`. These operators are implemented as streaming operators for streams of update triples, and as wrappers around arrangements that produce new arrangements. For example, the `filter` operator results in an arrangement that applies a supplied predicate as part of navigating through a wrapped inner arrangement.

This design implies a trade-off. An aggressive filter may reduce the volume of data to the point that it is cheap to maintain a separate index, and relatively ineffective to search in a large index only to discard the majority of results. The user controls which implementation to use by specifying the type: they can filter an arrangement, or first reduce the arrangement to a stream of updates and then filter it.

5.2 Key-altering stateless operators

Some stateless operators are "key-altering", in that the indexed representation of their output has little in common with that of their input. The most obvious example is the `map` operator, which may perform arbitrary record-to-record transformations. These operators always produce outputs represented as streams of update triples.

5.3 Stateful operators

Differential Dataflow's stateful operators are data-parallel: their input *data* have a *(key, val)* structure, and that the computation acts independently on each group of *key* data. This independence is what allows Naiad and similar systems to distribute operator work across otherwise independent workers, which can then process their work without further coordination. At a finer scale, this independence means that each worker can determine the effects of a sequence of updates on a key-by-key basis, resolving all updates to one key before moving to the next, even if this violates timestamp order.

5.3.1 The `join` operator

Our `join` operator takes as inputs batches of updates from each of its arranged inputs. It produces any changes in outputs that result from its advancing inputs, but our implementation has several variations from a traditional streaming hash-join.

Trace capabilities. The join operator is bi-linear, and needs only each input trace in order to respond to updates from the *other* input. As such, the operator can advance the frontiers of each trace handle by the frontier of the other input, and it can drop each trace handle when the other input closes out. This is helpful if one input is static, as in static graph processing.

Alternating seeks. Join can receive input batches of substantial size, especially when importing an existing shared arrangement. Naively implemented, we might require time linear in the input batch sizes. Instead, we perform alternating seeks between the cursors for input batches and traces of the other input: when the cursor keys match we perform work, and if the keys do not match we seek forward for the larger key in the cursor with the smaller key. This pattern ensures that we perform work at most linear in the smaller of the two sizes, seeking rather than scanning through the cursor of the larger trace, even when it is supplied as an input batch.

Amortized work. The `join` operator may be called upon to produce a significant amount of output data that can be reduced only once it crosses an exchange edge for a downstream operator. If each input batch is immediately processed to completion, workers may be overwhelmed by the amount of output, either buffered for transmission or (as in our prototype) sent to destination workers but then buffered at each awaiting reduction. Instead, operators respond to new input batches by producing "futures", limited batches of computation which can each be executed until sufficiently many outputs are produced and are then suspended. These futures make copies of the shared batch and trace references they use, and so do not block state maintenance for other operators.

5.3.2 The `group` operator

The `group` operator takes as input an arranged collection with data of the form *(key, val)* and a reduction function from a key and list of values to a list of output values. At each time the output might change, we reform the input and apply the reduction function, and compare the results to the reformed output to determine if output changes are required.

Perhaps surprisingly, the output may change at times that do not appear in the input (because the least upper bound of two times does not need to be one of the times). Consequently, the `group` operator tracks a list of pairs *(key, time)* of future work that are required even if we see no input updates for the key at that time. For each such *(key, time)* pair, the `group` operator accumulates the input and output for *key* at *time*, applies the reduction function to the input, and subtracts the accumulated output to produce any corrective output updates.

Output arrangements. The `group` operator uses a shared arrangement for its output, to efficiently reconstruct what it has previously produced as output without extensive re-invocation of the supplied user logic (and to avoid potential non-determinism therein). This provides the `group` operator the opportunity to share its output trace, just as the `arrange` operator does. It is common, especially in graph processing, for the results of a `group` to be immediately joined on the same key, and `join` can re-use the same indexed representation that `group` uses internally for its output.

5.4 Iteration

The iteration operator is essentially unchanged from Naiad's Differential Dataflow implementation. We have ensured that arrangements can be brought in to iterative scopes from outer scopes using only an arrangement wrapper, which allows access to shared arrangements in iterative computations.

6 Implementation

We implemented shared arrangements in a new prototype stream processor, SysX. Our Rust implementation of Differential Dataflow [19] with shared arrangements consists of a total of about 11,700 lines of code, and builds on an existing open-source implementation of timely dataflow [1].

To ease development of dataflows with arrangements, we make use of Rust types and modularity wherever possible. For example, streams carrying ordinary update triples and streams carrying arranged batches are different types, making it impossible to compile a dataflow that mixes them incorrectly. Likewise, converting arranged operator output into a standalone collection requires calling a `as_collection()` method to obtain a stream of the right type for consumption by operators that are not arrangement-aware.

We also aim to make it easy to add new arrangement-aware operator implementations. The `arrange` operator is defined in terms of a generic trace type, and our amortized merging trace is defined in terms of a generic batch type. Our batch implementations are defined for generic data types that are orderable (for merging) and hashable (for partitioning). Each of these layers can be replaced without rewriting the surrounding superstructure. For example, we provide two batch implementations that structure the *data* part of the update triple either as `(key, val)` and or just `key`, the latter with a simplified representation and navigation logic.

7 Evaluation

We now experimentally evaluate the benefits of shared arrangements in SysX. We first evaluate SysX on real-world end-to-end workloads to demonstrate the benefits of shared, indexed state (§ 7.1), where we see that it improves latency, throughput, and memory utilization. We then use microbenchmarks to characterize the performance of our design and of the arrangement-aware operator implementations (§ 7.2). Finally, we demonstrate that SysX's shared arrangements offer benefits across several domains, and that SysX maintains high baseline performance compared to other systems even without using shared arrangements (§ 7.3).

Setup. We evaluate SysX on a four-socket NUMA system equipped with four Intel Xeon E5-4650 v2 CPUs, each with 10 physical cores and 512 GB of aggregate system memory. We compiled SysX with `rustc` 1.33.0 and the `jemalloc` [3] allocator. SysX does distribute across multiple machines and supports sharding shared arrangements across them, but our evaluation here is restricted to multiprocessors. When we compare against other systems, we rely on the best, tuned measurements reported by their authors, but compare SysX only if we are executing it on comparable or less powerful hardware than the other systems had access to.

7.1 End-to-end impact of shared arrangements

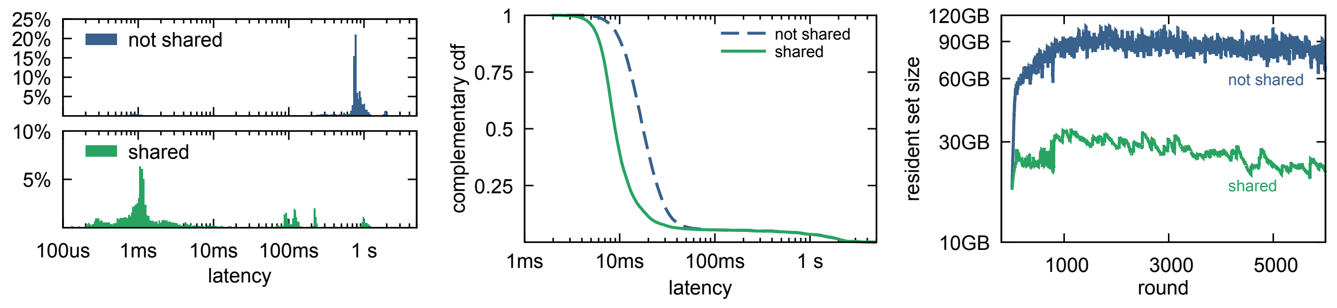
We start with a demonstration of the impact of shared arrangements on the end-to-end experience for familiar analytics tasks. We choose a standard relational analytics task, the established TPC-H data warehousing benchmark, and a recent interactive graph analytics benchmark. For the relational queries, we would expect to see shared arrangements reduce the *latency* of installing a new query compared to non-shared setups that reprocess entire collections to add a query. For the graph tasks, we would hope to see SysX tolerate a higher update *throughput* than existing graph processors, as SysX can parallelize and scale shared indexes.

7.1.1 TPC-H

The TPC-H benchmark schema has eight relations, which describe order fulfillment events, as well as the orders, parts, customers, and suppliers they involve, and the nations and regions in which these entities exist. Of the eight relations, seven have primary keys, and are immediately suitable for arrangement (by their primary key). The eighth relation is `lineitem`, which contains fulfillment events, and we treat this collection as a stream and do not arrange it.

TPC-H contains 22 "data warehousing" queries, meant to be run against large, static datasets. Following work by Nikolic et al. [21], we consider a modified setup where the eight relations are progressively loaded, one record at a time, in a round-robin fashion among the relations (at scale factor 10).[1] To demonstrate the benefits of maintained arrangements, we interactively deploy and retire queries while the eight relations are loaded. Each query has access to the full current contents of the seven keyed relations that we maintain shared

[1]SysX matches or outperforms Nikolic et al.'s DBToaster results [21], even when running queries in isolation on the entire fulfillment stream and without shared arrangements. These results are in our extended technical report [4].



(a) Query latency. **(b)** Time to produce results. **(c)** Memory footprint (RSS).

Figure 3. Shared arrangements reduce query (a) query installation latency, (b) update processing latency, and (c) the memory footprint of concurrent TPC-H queries that randomly arrive and retire. The setup uses 32 workers, runs at TPC-H scale factor 10, and loads relations round-robin. Note the $\log_{10}$-scale x-axes in (a) and (b), and the $\log_{10}$-scale y-axis in (c).

arrangements for. We treat the fulfillment events as a stream, and each query only observes the fulfillment events from the point it is deployed until the point it is retired, implementing a "streaming" rather than a "historic" query.

The 22 TPC-H queries have different characteristics, but broadly they either derive from the `lineitem` relation and reflect only current fulfillments, or they do not and reflect the full accumulated volume of other relations. Without shared arrangements, either type of query requires building new indexed state for the seven non-`lineitem` relations. With shared indexes, we would expect queries of the first type to be very fast to deploy, as their outputs are initially empty. Queries of the second type should take a non-trivial amount of time to deploy in either case, as their initial output requires computing over many records.

Query latency. To evaluate query latency, we measure the time from the start of query deployment until just before the first record is ready to be returned. Query latency is important because it determines whether the system delivers an interactive experience to human users, but also because it determines how quickly applications can derive insights from streams by programmatically issuing queries against them.

Figure 3a reports the distribution of query latencies, with and without shared arrangements. With shared arrangements, most queries (those that derive from `lineitem`) are deployed and begin updating in milliseconds; the five queries that do not derive from `lineitem` perform non-trivial computation to produce their initial correct answer and take between 100ms and 1s. Without shared arrangements, almost all queries take 1–2 seconds to install as they must create a reindexed copy of their inputs. Q01 and Q06 are exceptions, as they use no relations other than `lineitem`, and thus avoid reindexing any inputs. Maintained shared arrangements therefore allows more rapid deployment of new queries, often by multiple orders of magnitude.

Update latency. Once a query is installed, SysX continuously updates its results as new `lineitem` records arrive. To evaluate the update latency achieved, we record the amount of time required to process each round of input data updates after query installation. If shared arrangements had high overheads, or if they slowed down the sharing dataflows, we might see an increase in latency; if sharing reduces the aggregate work required, we should see a reduction.

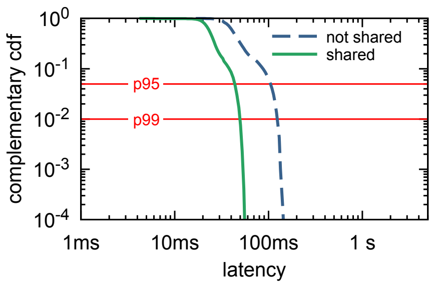
Figure 3b presents the distribution of these times, with and without shared arrangements, as a complementary cumulative distribution function (CDF). This visualization—which we will use repeatedly—shows the "fraction of times with latency greater than" and highlights the tail latencies towards the bottom-right side of the plot. We see a modest but consistent reduction in processing time (about 2×) when using shared arrangements, which eliminate redundant index maintenance work. There is a noticeable tail in both cases, owed to two expensive queries that involve inequality joins (Q11 and Q22) and which respond slowly to changes in their inputs independently of whether their arrangements are shared or not. Shared arrangements therefore offer runtime performance improvements and potentially increased update throughputs.

Memory footprint. Since shared arrangements eliminate duplicate copies of index structures, we would expect them to reduce the dataflow's memory footprint. To evaluate the memory footprint, we record the resident set size (RSS) as the experiment proceeds.

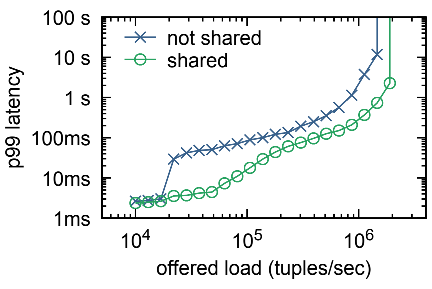
Figure 3c presents the timelines of the RSS with and without shared arrangements, and shows a substantial reduction (2–3×) in memory footprint when shared arrangements are present. Without shared arrangements, the memory footprint also varies substantially (between 60 and 120 GB) as the system creates and destroys indexes for queries that arrive and depart, while shared arrangements remain below 40 GB.

7.1.2 Interactive graph queries

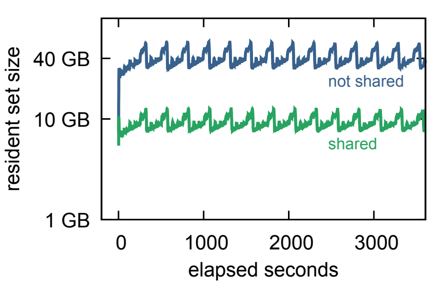
We further evaluate SysX with an open-loop experiment issuing queries against an evolving graph. This experiment issues the four queries used by Pacaci et al. [22] to compare



(a) Latencies for query mix.



(b) 99th percentile latency at given offered load.



(c) Resident set size.

Figure 4. Shared arrangements reduce query latency, increase the load handled, and reduce the memory footprint of interactive graph queries. The setup uses 32 workers, and issues 100k updates/sec and 100k queries/sec against a 10M node/64M edge graph in (a) and (c), while (b) varies the load. Note the $\log_{10}$–$\log_{10}$ scales in (a) and (b), and the $\log_{10}$-scale y-axis in (c).

relational and graph databases: point look-ups, 1-hop look-ups, 2-hop look-ups, and 4-hop shortest path queries (shortest paths of length at most four). In the first three cases, the query argument is a graph node identifier, and in the fourth case it is a pair of identifiers.

We implement each of these queries as Differential Dataflows where the query arguments are independent collections that may be modified to introduce or remove specific query arguments. This query transformation was introduced in NiagaraCQ [10] and is common in stream processors, allowing them to treat queries as a streaming input.

We apply a fixed rate of graph updates (100,000 per second) to a graph with 10M nodes and 64M edges, and also update the query arguments of interest at experiment-specific rates. Each graph update is the addition or removal of a random graph edge, and each query update is the addition or removal of a random query argument (queries are maintained while installed, rather than issued only once). All experiments evenly divide the query updates between the four query types.

Query latency. Initially, we run an experiment with a fixed rate of 100,000 query updates per second, independently of how quickly SysX responds to them. We would hope that SysX responds quickly, and that shared arrangements of the graph structure should help reduce the latency of query updates, as SysX must apply changes to one shared index rather than several independent ones.

Figure 4a reports the latency distributions with and without a shared arrangement of the graph structure, as a complementary CDF. Sharing the graph structure results in a 2–3× reduction in overall latency in the 95th and 99th percentile tail latency (from about 150ms to about 50ms). In both cases, there is a consistent baseline latency, proportional to the number of query classes maintained. Shared arrangements yield latency reductions across all query classes, rather than, e.g., imposing the latency of the slowest query on all sharing dataflows. This validates that queries can proceed at different rates, an important property of our shared arrangement design.

System	cores	look-up	one-hop	two-hop	four-path
Neo4j	32	9.08ms	12.82ms	368ms	21ms
Postgres	32	0.25ms	1.4ms	29ms	2242ms
Virtuoso	32	0.35ms	1.23ms	11.55ms	4.81ms
SysX (batch: 10^0)	32	0.64ms	0.92ms	1.28ms	1.89ms
SysX (batch: 10^1)	32	0.81ms	1.19ms	1.65ms	2.79ms
SysX (batch: 10^2)	32	1.26ms	1.79ms	2.92ms	8.01ms
SysX (batch: 10^3)	32	5.71ms	6.88ms	10.14ms	72.20ms

Table 2. On comparable 10M node/64M edge graphs, SysX is broadly competitive with the average graph query latencies of three systems evaluated by Pacaci et al. [22], and scales to higher throughput using batching. The SysX batch size is the number of concurrent queries per measurement.

Update throughput. To test how SysX's shared arrangements scale with load, we next scale the rates of graph updates and query changes up to two million changes per second each. An ideal result would show that sharing the arranged graph structure consistently reduces the computation required, thus allowing us to scale to a higher load using fixed resources.

Figure 4b reports the 99th percentile latency with and without a shared graph arrangement, as a function of offered load and on a log–log scale. The shared configuration results in reduced latencies at each offered load, and tolerates an increased maximum load at any target latency. At the point of saturating the server resources, shared arrangements tolerate 33% more load than the unshared setup, although this number is much larger for specific latencies (e.g., 5× at a 20ms target). We note that the absolute throughputs achieved in this experiment exceed the best throughput observed by Pacaci et al. (Postgres, at 2,000 updates per second) by several orders of magnitude, further illustrating the benefits of parallel dataflow computation with shared arrangements.

Memory footprint. Finally, we consider the memory footprint of the computation. There are five uses of the graph across the four queries, but also per-query state that is not profitably shared, so we would expect a reduction in memory footprint of somewhat below 4×.

Figure 4c reports the memory footprint for the query mix with and without sharing, for an hour-long execution. The memory footprint oscillates around 10 GB with shared arrangements, and around 40 GB (4× larger) without shared arrangements. This illustrates that sharing state affords memory savings proportional to the number of reuses of a collection.

Comparison with other systems. Pacaci et al. [22] evaluated relational and graph databases on the same graph queries. SysX is a stream processor rather than a database and supports somewhat different features, but its performance ought to be comparable to the databases' for these queries. We stress, however, that our implementation of the queries as Differential Dataflows relies on advanced knowledge of the query classes, a specialization that the other systems do not require.

We ran SysX experiments with a random graph comparable to the one used in Pacaci et al.'s comparison. Table 2 reports the average latency to perform and then await a single query in different systems, as well as the time to perform and await batches of increasing numbers of concurrent queries for SysX. While SysX does not provide the lowest latency for point look-ups, it does provides excellent latencies for other queries and increases query throughput with batch size.

7.2 Design evaluation

To validate that our shared arrangement design meets satisfies the objectives we set out to achieve, we perform microbenchmarks of an arrangement's response to different loads. In all benchmarks, we apply an `arrange` operator to a continually changing collection of 64-bit identifiers (with 64-bit timestamp and signed difference). The inputs are generated randomly at the worker, and exchanged (shuffled) by key prior to entering the arrangement. We are primarily interested in the distribution of response latencies, as slow edge-case behavior of an arrangement would affect this statistic most. We report all latencies as complementary CDFs to get high resolution in the tail of the distribution.

Varying load. As update load varies, our shared arrangement design should trade latency for throughput until equilibrium is reached. Figure 5a reports the latency distributions for a single worker as we vary the number of keys and offered load in an open-loop harness, from 10M keys and 1M updates per second, downward by factors of two. As load decreases, latencies also drop proportionally.

Strong scaling. More parallel workers should allow faster maintenance of a shared arrangement, as the work to update it parallelizes. Figure 5b reports the latency distributions for an increasing numbers of workers under a fixed load of 10M keys and 1M updates per second. As the number of workers increases, latencies decrease, especially in the tail of the distribution: for example, the 99th percentile latency of 500ms with one worker drops to 6ms with eight workers.

Weak scaling. Varying the number of workers while proportionately increasing the number of keys and offered load would ideally result in constant latency. Figure 5c shows that the latency distributions do exhibit increased tail latency, as data exchange at the arrangement input becomes more complex. However, the latencies do stabilize at 100–200ms with larger numbers of workers.

Throughput. An arrangement consists of several subcomponents: batch formation, trace maintenance, and e.g., a maintained `count` operator. Ideally, each of these components would scale with growing update load, but many possible implementation mistakes would inhibit their scaling. To validate it, we issue repeated rounds of batches of 10,000 updates at each worker, rather than from an open-loop harness. Figure 5d reports the peak throughputs as the number of cores (and thus, workers and arrangement shards) grows. All components scale linearly to 32 workers.

Amortized merging. The amortized merging strategy is crucial for shared arrangements to achieve low update latency, but its efficacy depends on setting the right amortization coefficients. Lazy merging better amortizes expensive changes, but might increases common-case latency, while eager merging amortizes less, so expensive changes potentially increase tail latency. Ideally, SysX's default would pick a good tradeoff between common-case and tail latencies at different scales.

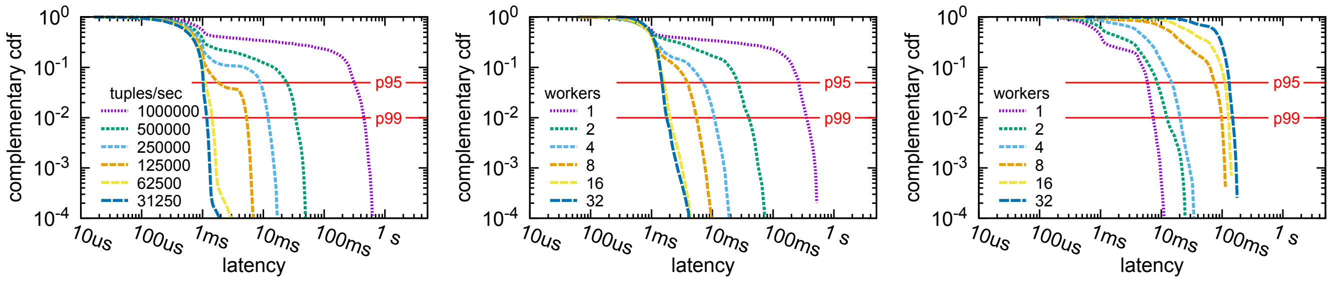
Figure 5e reports the latency distributions for one and 32 workers, each with three different merge amortization coefficients: the most eager, SysX's default, and the most lazy possible. For a single worker, lazier settings have smaller tail latencies, but are more often in that tail. For 32 workers, the lazier settings are significantly better, because eager strategies often cause workers to stall waiting for a long merge at one worker. This significantly impedes strong scaling, and matches similar observations about garbage collection at scale [15]. SysX's default setting achieves good performance at both scales.

Join proportionality. Our arrangement-aware join operator design aims to work proportional only to the size of the smaller of the incoming pre-arranged batch vs. the state joined against (§ 5.3.1). We validate that this is the case by measuring the latency distributions to install, execute, and complete new dataflows that join small collections of varying size against a pre-existing arrangement of 10M keys.

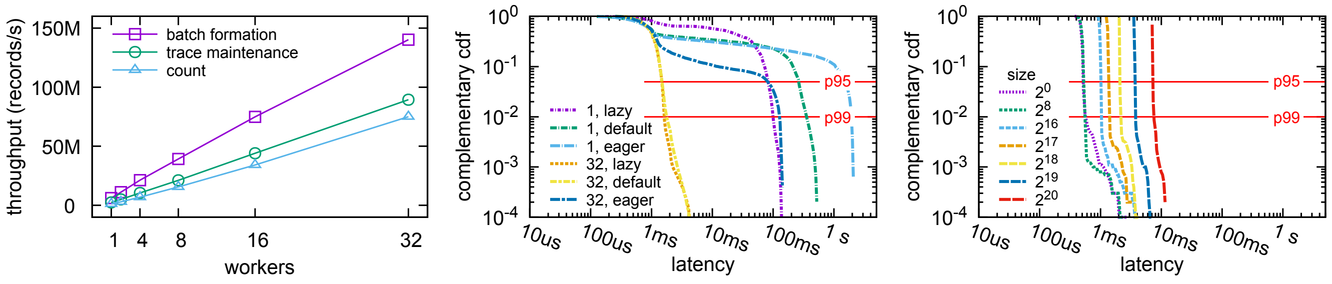
The varying lines in Figure 5f demonstrate that the join work is indeed proportional to the small collection's size, rather than to the (constant) 10M arranged keys. This is possible because the `join` operator receives pre-arranged batches of input, and SysX advances the cursors such that it does work proportional to the smaller input—an optimization that is impossible in a record-at-a-time stream processor.

7.3 Baseline performance on reference tasks

We also evaluate SysX against established prior work to demonstrate that SysX is competitive with and occasionally



(a) Varying offered load with 1 worker. **(b)** Varying workers with fixed load. **(c)** Varying workers and offered load



(d) Task throughput, varying workers. **(e)** Amortized merging levels. **(f)** Join with pre-arranged collection.

Figure 5. Microbenchmarks of our shared arrangement design suggest that our design scales well with growing parallelism ((b)–(d)) and load ((a), (c)–(d)), and that the key ideas of amortized merging ((e)) and proportional work across inputs ((f)) are crucial to achieving low update latencies. (b) and (e) generate a fixed load of 1M input records per second.

better than peer systems. Some of these benefits derive from the use of shared arrangements, but others do not, as many established benchmarks intentionally run in isolation.

7.3.1 Datalog workloads

Datalog is a relational language in which queries are sets of recursively defined productions, which are iterated from a base set of records until no new records are produced. Unlike graph computation, Datalog queries tend to produce and work with substantially more records than they are provided as input. Several shared-memory systems for Datalog exist, including LogicBlox, DLV [2], DeALS [29], and several distributed systems have recently emerged, including Myria [27], SociaLite [24], and BigDatalog [25]. At the time of writing, only LogicBlox supports decremental updates to Datalog queries, using a technique called “transaction repair” [26]. SysX supports incremental and decremental updates to Datalog computations, as well as interactive top-down queries.

Top-down (interactive) evaluation. Datalog users commonly specify certain values in a query, such as *reach*(*“david”*, ?), to request nodes reachable from a specified source node. The “magic set” transformation [6] rewrites such queries as bottom-up computations with a new base relation that seeds the bottom-up derivation with query arguments; the rewritten rules derive facts only with the participation of some seed record. SysX, like some interactive Datalog environments, can perform this work against maintained arrangements of

Query	statistic	tree-11	grid-150	gnp1
tc(x,?)	increm., median	2.56ms	346.28ms	18.29ms
	incremental, max	9.05ms	552.79ms	25.40ms
	full evaluation	0.08s	6.18s	9.45s
tc(?,x)	increm., median	15.63ms	320.83ms	15.58ms
	incremental, max	18.01ms	541.76ms	23.84ms
	full evaluation	0.08s	6.18s	9.45s
sg(x,?)	increm., median	68.34ms	1075.11ms	20.08ms
	incremental, max	95.66ms	2285.11ms	26.56ms
	full evaluation	56.45s	0.60s	19.85s

Table 3. SysX enables interactive computation of three Datalog queries (32 workers, medians and maximums over 100 queries). Full eval. is required without shared arrangements.

the non-seed relations. We would expect this approach to be much faster than full evaluation, which batch processors that re-index the non-seed relations must perform.

Table 3 reports SysX’s median and maximum latencies for 100 random arguments for three interactive queries on three widely-used benchmark graphs, and the times for full evaluation of the related query, using 32 workers. SysX’s arrangements mostly reduce runtimes from seconds to milliseconds. The slower transformed query for sg(x,?) on grid-150 is due to a known problem with the magic set transform.

Bottom-up (batch) evaluation. In our extended technical report [4], we evaluate SysX relative to distributed and shared-memory Datalog engines, using their benchmark queries and

System	cores	*linux*	*psql*	*httpd*
SociaLite	4	OOM	OOM	4 hrs
Graspan	4	713.8 min	143.8 min	11.3 min
SysX	1	76.8s	37.0s	10.9s

(a) *dataflow* analysis.

System	cores	*linux*	*psql*	*httpd*
SociaLite	4	OOM	OOM	> 24 hrs
Graspan	4	99.7 min	353.1 min	479.9 min
SysX	1	423.1s	362.0s	536.3s
SysX (Opt)	1	401.7s	94.3s	91.9s
SysX (Opt+)	1	191.3s	75.9s	77.4s

(b) *points-to* analysis. SysX (Opt) is an optimized query, and SysX (Opt+) is the optimized query with shared arrangements.

System	cores	*linux*	*psql*	*httpd*
SysX (med)	1	1.11ms	185ms	22.0ms
SysX (max)	1	8.13ms	1.48s	218ms

(c) Times to remove each of the first 1,000 null assignments from the interactive top-down *dataflow* analysis.

Table 4. SysX performs well for Graspan [28] program analyses on three graphs. SociaLite and Graspan results reproduced from Wang et al. [28]; OOM = out of memory.

datasets ("transitive closure" and "same generation" on trees, grids, and random graphs). We find that SysX generally outperforms the distributed systems, and is comparable to the best shared-memory engine (DeALS).

7.3.2 Program Analysis

Graspan [28] is a system built for static analysis of large code base, created in part because existing systems were unable to handle non-trivial analyses at the sizes required. Wang et al. benchmarked Graspan for two program analyses, *dataflow* and *points-to* [28]. The *dataflow* query propagates null assignments along program assignment edges, while the more complicated *points-to* analysis develops a mutually recursive graph of value flows, and memory and value aliasing. We developed a complete implementation of Graspan—query parsing, dataflow construction, input parsing and loading, dataflow execution—in 179 lines of code on top of SysX, and benchmark it on these queries.

Graspan is designed to operate out-of-core, and explicitly manages its data on disk. We therefore report SysX measurements from a laptop with only 16 GB of RAM, a limit exceeded by the *points-to* analysis (which peaks around 30 GB). The sequential access in this analysis makes standard OS swapping mechanisms sufficient for out-of-core execution, however. To verify this, we modify the computation to use 32-bit integers, reducing the memory footprint below the RAM size, and find that this optimized version runs only 20% faster than the out-of-core execution.

Tables 4a and Table 4b show the running times reported by Wang et al. compared those SysX achieves. For both queries, we see a substantial improvement (from 14× to 550×). The *points-to* analysis is dominated by the determination of a large relation (value aliasing) that is used only once. This relation can be optimized out, as value aliasing is eventually restricted by dereferences, and this restriction can be performed before forming all value aliases. This optimization results in a much more efficient computation, and one that reuses relations multiple times. Table 4b reports the optimized running times, with sharing (Opt+) and without (Opt), illustrating that shared arrangements offer benefits even within a single query.

Top-down evaluation. Both *dataflow* and *points-to* can be transformed to support interactive queries instead of batch computation. Table 4c reports the median and maximum latencies to remove the first 1,000 null assignments from the completed *dataflow* analysis and correct the set of reached program locations. While there is some variability, the timescales are largely interactive and suggest the potential for a improved developer experience.

7.3.3 Batch graph computation

In our extended technical report [4], we evaluate SysX on standard batch iterative graph computations of reachability, breadth-first distance, and undirected connectivity on standard graphs. Our measurements indicate that SysX is consistently faster than systems like BigDatalog [25], Myria [27], SociaLite [24], and GraphX [16], but is substantially less efficient than purpose-written single-threaded code applied to pre-processed graph data. Such pre-processing is common, as it allows use of efficient static arrays, but it prohibits more general vertex identifiers or graph updates. When we amend our purpose-built code to use a hash table instead of an array, SysX becomes competitive between two and four cores.

These results are independent of shared arrangements, but indicate that SysX's arrangement-aware implementation does not impose any undue cost on computations without sharing.

8 Conclusions

Shared arrangements enable low-latency, interactive queries against data streams by sharing indexed stated between queries. In our design, multiple operators in the same or different parallel dataflows share read access to continuously updated shared arrangement. Multiversioning the shared arrangement is crucial to computing correct results and sharding the arrangement helps achieve parallel speedup.

Our prototype implementation, SysX, installs new queries against a stream in milliseconds, reduces the processing and space cost of multiple dataflows, and achieves high performance on a range of different workloads, whether using shared arrangements or not.

SysX has been released as open-source software, and is in use by several research groups and companies. Early anecdotal reports indicate that shared arrangements have led to important performance improvements for these users.

References

[1] https://github.com/TimelyDataflow/timely-dataflow/.

[2] DLVSYSTEM. http://www.dlvsystem.com.

[3] Jemalloc memory allocator. http://jemalloc.net.

[4] Anonymous. Shared arrangements: Practical inter-query sharing for streaming dataflows (extended technical report). https://github.com/shared-arrangements/tr/raw/master/anonymous-technical-report.pdf.

[5] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. *STREAM: The Stanford Data Stream Management System*, pages 317–336. Springer, Berlin/Heidelberg, Germany, 2016.

[6] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, PODS '86, pages 1–15, New York, NY, USA, 1986. ACM.

[7] George Candea, Neoklis Polyzotis, and Radek Vingralek. A scalable, predictable join operator for highly concurrent data warehouses. *Proceedings of the VLDB Endowment*, 2(1):277–288, August 2009.

[8] Paris Carbone, Stephan Ewen, Seif Haridi, Asterios Katsifodimos, Volker Markl, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *IEEE Data Engineering*, 38(4), December 2015.

[9] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. Telegraphcq: Continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 668–668, 2003.

[10] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. Niagaracq: A scalable continuous query system for internet databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA.*, pages 379–390, 2000.

[11] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, January 2008.

[12] Kayhan Dursun, Carsten Binnig, Ugur Cetintemel, and TIm Kraska. Revisiting reuse in main memory database systems. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1275–1289, 2017.

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

[14] Jon Gjengset, Malte Schwarzkopf, Jonathan Behrens, Lara Timbó Araújo, Martin Ek, Eddie Kohler, M. Frans Kaashoek, and Robert Morris. Noria: dynamic, partially-stateful data-flow for high-performance web applications. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 213–231, October 2018.

[15] Ionel Gog, Jana Giceva, Malte Schwarzkopf, Kapil Vaswani, Dimitrios Vytiniotis, Ganesan Ramalingan, Derek Murray, Steven Hand, and Michael Isard. Broom: Sweeping out garbage collection from big data systems. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, HOTOS'15, pages 2–2, Berkeley, CA, USA, 2015. USENIX Association.

[16] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, pages 599–613, 2014.

[17] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. Nectar: Automatic management of data and computation in datacenters. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 75–88, 2010.

[18] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *Proceedings of the 2nd ACM SIGOPS European Conference on Computer Systems (EuroSys)*, pages 59–72, March 2007.

[19] Frank McSherry, Derek G. Murray, Rebecca Isaacs, and Michael Isard. Differential dataflow. In *Proceedings of the 6th Biennial Conference on Innovative Data Systems Research (CIDR)*, January 2013.

[20] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A Timely Dataflow System. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 439–455, November 2013.

[21] Milos Nikolic, Mohammad Dashti, and Christoph Koch. How to win a hot dog eating contest: Distributed incremental view maintenance with batch updates. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 511–526, 2016.

[22] Anil Pacaci, Alice Zhou, Jimmy Lin, and M. Tamer Özsu. Do we need specialized graph databases?: Benchmarking real-time social networking applications. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems*, GRADES'17, pages 12:1–12:7, New York, NY, USA, 2017. ACM.

[23] PostgreSQL Global Development Group. The PostgreSQL Database Management System. https://www.postgresql.org/, April 2019.

[24] Jiwon Seo, Stephen Guo, and Monica S. Lam. Socialite: An efficient graph query language based on datalog. *IEEE Trans. Knowl. Data Eng.*, 27(7):1824–1837, 2015.

[25] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. Big data analytics with datalog queries on spark. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*, pages 1135–1149, 2016.

[26] Todd L. Veldhuizen. Transaction repair: Full serializability without locks. *CoRR*, abs/1403.5645, 2014.

[27] Jingjing Wang, Tobin Baker, Magdalena Balazinska, Daniel Halperin, Brandon Haynes, Bill Howe, Dylan Hutchison, Shrainik Jain, Ryan Maas, Parmita Mehta, Dominik Moritz, Brandon Myers, Jennifer Ortiz, Dan Suciu, Andrew Whitaker, and Shengliang Xu. The myria big data management and analytics system and cloud services. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*, 2017.

[28] Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 389–404, New York, NY, USA, 2017. ACM.

[29] Mohan Yang, Alexander Shkapsky, and Carlo Zaniolo. Scaling up the performance of more powerful datalog systems on multicore machines. *VLDB Journal*, 26(2):229–248, 2017.

[30] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, December 2008.

[31] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, pages 15–28, April 2012.

[32] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming

computation at scale. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 423–438, November 2013.