Shared Arrangements: practical inter-query sharing for streaming dataflows

Frank McSherry* Andrea Lattuada Malte Schwarzkopf‡ Timothy Roscoe
*Materialize, Inc. Dept. of Computer Science, ETH Zürich ‡Brown University
mcsherry@materialize.io, {andreal,troscoe}@inf.ethz.ch, malte@cs.brown.edu

ABSTRACT

Current systems for data-parallel, incremental processing and view maintenance over high-rate streams isolate the execution of independent queries. This creates unwanted redundancy and overhead in the presence of concurrent incrementally maintained 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 order-of-magnitude improvements in query response time and resource consumption for incremental, interactive queries against high-throughput streams, while also significantly improving performance in other domains including business analytics, graph processing, and program analysis.

PVLDB Reference Format:
Frank McSherry, Andrea Lattuada, Malte Schwarzkopf, Timothy Roscoe. Shared Arrangements: practical inter-query sharing for streaming dataflows. *PVLDB*, 13(10): 1793-1806, 2020.
DOI: https://doi.org/10.14778/3401960.3401974

1. INTRODUCTION

In this paper, we present *shared arrangements*, a new technique for efficiently sharing indexed, consistent state and computation between the operators of multiple concurrent, data-parallel streaming dataflows. We have implemented shared arrangements in DD, the current implementation of Differential Dataflow [28, 27, 1], but they are broadly applicable to other streaming systems.

Shared arrangements are particularly effective in interactive data analytics against continually-updating data. Consider a setting in which multiple analysts, as well as software like business intelligence dashboards and monitoring systems, interactively submit standing queries to a stream processing system. The queries remain active until they are removed. Ideally, queries would install quickly, provide initial results promptly, and continue to deliver updates with low latency as the underlying data change.

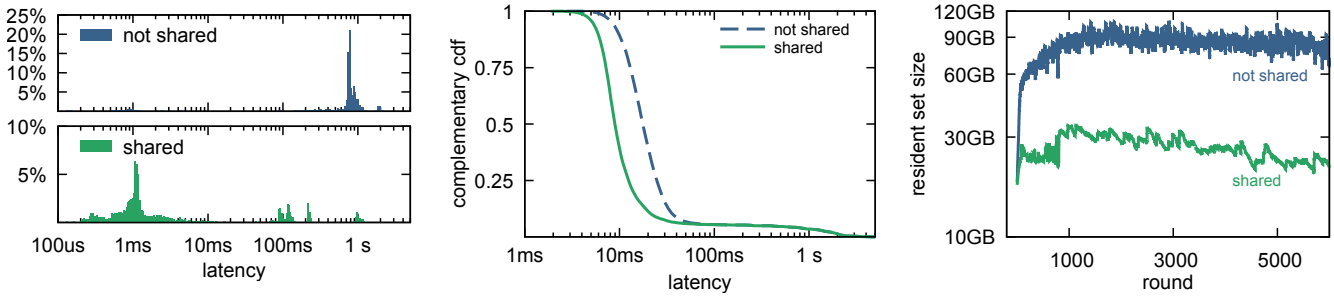
Proceedings of the VLDB Endowment, Vol. 13, No. 10
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3401960.3401974

Data-parallel stream processors like Flink [12], Spark Streaming [40], and Naiad [28] excel at incrementally maintaining the results of such queries, but each maintain queries in independent dataflows with independent computation and operator state. Although these systems support the sharing of common sub-queries, as streams of data, none share the *indexed* representations of relations among unrelated subqueries. However, there are tremendous opportunties for sharing of state, even when the dataflow operators are not the same. For example, we might expect joins of a relation R to use its primary key; even if several distict queries join R against as many other distinct relations, a shared index on R would benefit each query. Existing systems create independent dataflows for distinct queries, or are restricted to redundant, per-query indexed representations of R, wasting memory and computation.

By contrast, classic relational databases have long shared indexes over their tables across unrelated queries. The use of shared indexes reduces query times tremendously, especially for point look-ups, and generally improves the efficiency of queries that access relations by the index keys. While they have many capabilities, relational databases lack streaming dataflow system's support for low-latency, high-throuhput incremental maintenance of materialized query results [19, 8]. Existing shared index implementations share all reads and writes among multiple workers, and are not immediately appropriate for dataflow workloads where the operator state is sharded across independent workers. In this work, we seek to transport the shared index idiom from relational databases to streaming dataflows, applying it across changing maintained queries.

Our main observations are that *(i)* many dataflow operators write the same internal state, representing the accumulated changes of each of their input streams, *(ii)* these dataflow operators often access this state with independent and fundamentally different patterns, and *(iii)* this state can be efficiently shared with single-writer, multiple-reader data structure. Shared arrangements are our design for single-writer, multiple-reader, shared state in dataflow systems.

To illustrate a natural setting for shared arrangements, we run a mix of interactively issued and incrementally maintained TPC-H [6] queries executed as dataflows against a stream of order fulfillment events (i.e., changes to the `lineitem` relation). This is similar to a modern business analytics setting with advertisers, impressions, and advertising channels, and our dynamic query setup mimicks the behavior of human analysts and business analytics dashboards. (TPC-H is originally a static "data-warehousing" benchmark; our streaming setup follows that used by Nikolic et al. [29].) We measure the query installation latency—i.e., the time until a new query returns results—as well as update processing latency and standing memory footprint. Figure 1 reports the performance of DD with shared arrangements ("shared") and without ("not shared"; representative of other data-parallel stream processors). The mea-



(a) Query installation latency.

(b) Update processing latency.

(c) Memory footprint (RSS).

Figure 1: Shared arrangements reduce (a) query installation latency distribution, (b) update processing latency distribution, 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 rows from relations round-robin. Note the $\log_{10}$-scale x-axes in (a) and (b), and the $\log_{10}$-scale y-axis in (c).

surements show orders of magnitude improvements in query installation latency (a weakness of existing dataflow systems), and improved update processing latency and memory use.

Shared arrangements achieve these improvements because they remove the need to maintain dataflow-local indexes for each query. As a concrete example throughout this paper, we consider TPC-H queries 3 and 5. Both queries join `lineitem` with the `order` and `customer` relations by their primary keys. While the queries lack overlapping subqueries that classic multi-query optimization (MQO) would detect, they both perform lookups into `order` and `customer` by their respective primary keys when processing an updated `lineitem` record. Existing stream processors will create and maintain a per-query index for each relation, as these systems are designed to decouple the execution of dataflow operators. Shared arrangements, by contrast, allow Q3 and Q5 to share indexes for these two relations. This can dramatically reduce the time to install the second query and provide initial results, and also increases overall system capacity, as multiple queries share in-memory indexes over the same relations. Finally, these benefits come without restricting update throughput or latency, as they do not change the data-parallel execution model of the stream processor.

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

Our full results in §6 confirm that shared arrangements translate into two benefits: *(i)* queries deploy and produce correct results immediately without rescanning historical data, and *(ii)* the same capacity (stream volume and concurrent queries) can be achieved with fewer cores and less RAM. 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×, compared to systems that do not share indexed state. These benefits hold without degrading performance on other tasks—batch and interactive graph processing, and Datalog-based program analysis—on which DD outperforms other systems.

Shared arrangements can be applied to many modern stream processors, but we implemented them as part of DD. DD has been the publicly available reference implementation of Differential Dataflow for several years [1], and is deployed in variety of industrial settings. For example, VMware Research uses DD to back their reactive DDlog Datalog engine [3], applied to problems in network reconfiguration and program analysis. Shared arrangements have proved key to the system's success.

Some benefits of shared arrangements are attainable in purely windowed streaming settings, which ensure that only bounded historical state must be reviewed for new queries. However, shared arrangements provide similar benefits without these restrictions, and support windowing of data as one of several join idioms. The main limitation of shared arrangements is that their benefits apply only in the cases where actual sharing occurs; while sharing appears common in settings with relational data and queries, bespoke stream processing computations (e.g., with complex and disjoint windowing on relations) may benefit to varying and lesser degrees.

In many ways, shared arrangements are the natural interpretation of an RDBMS index for data-parallel dataflow, and bring its benefits to a domain that has until now lacked them.

2. BACKGROUND AND RELATED WORK

Shared arrangements allow queries to share indexed state. Inter-query state sharing can be framed in terms of *(i) what* can be shared between queries, *(ii)* if this shared state can be *updated*, and *(iii)* the *coordination* required to maintain it. Figure 2 compares sharing in different classes of systems.

Relational databases like PostgreSQL [31] excel at answering queries over schema-defined tables. Indexes help them speed up access to records in these tables, turning sequential scans into point lookups. When the underlying records change, the database updates the index. This model is flexible and shares indexes between different queries, but it requires coordination (e.g., locking [15]). Scaling this coordination out to many parallel processors or servers holding shards of a large database has proven difficult, and scalable systems consequently restrict coordination.

Parallel-processing **"big data" systems** like MapReduce [16], Dryad [24], and Spark [39] rely only on coarse-grained coordination. They avoid indexes and turn query processing into parallel scans of distributed collections. But these collections are immutable: any change to a distributed collection (e.g. 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 allows different queries to share the (static) collection for reading [23]. This design aids scale-out, but makes these systems a poor fit for streaming computations, with frequent fine-grained changes to the collections.

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	DD	**Indexed state**	**Record-level**	**Coarse-grained**

Figure 2: Sharing of indexed in-memory state, record-level update granularity, and scalability through coarse-grained coordination are not all found in current systems. Shared arrangements combine these features in a single system.

Stream-processing systems reintroduce fine-grained mutability, but they lack sharing. Systems like Flink [12], Naiad [28], and Noria [19] keep long-lived, indexed intermediate results in memory for efficient incremental processing, partitioning the computation across workers for scale-out, data-parallel processing. However, stream processors 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 operators could, in principle, share.

By contrast, **shared arrangements** allow for fine-grained updates to shared indexes and preserve the scalability of data-parallel computation. In particular, shared arrangements rely on multi-versioned indices and data-parallel sharding to allow updates to shared state without the costly coordination mechanisms of classic databases. In exchange for scalability and parallelism, shared arrangements give up some abilities. Unlike indexes in relational databases, shared arrangements do not support multiple writers, and are not suitable tools to implement a general transaction processor. Because sharing entangles queries that would otherwise execute in isolation, it can reduce performance and fault isolation between queries compared to redundant, duplicated state.

It is important to contrast shared arrangements to Multi-Query Optimization (MQO) mechanisms that identify overlapping subqueries. MQO shares state and processing between queries with common subexpressions, but shared arrangements also benefit distinct queries that access the same indexes. Both relational and big data systems can identify common sub-expressions via MQO and either cache their results or fuse their computation. For example, CJoin [11] and SharedDB [18] share table scans between concurrent ad-hoc queries over large, unindexed tables in data warehousing contexts, and Nectar [23] does so for DryadLINQ [38] computations. More recently AStream [25] applied the architecture of SharedDB to windowed streaming computation, and can share among queries the resources applied to future windows. TelegraphCQ [13] and DBToaster [8] share state among continuous queries, but sequentially process each query without parallelism or shared indexes. Noria [19] shares computation between queries over streams, but again lacks shared indexes. In all these systems, potential sharing must be identified at query deployment time; none provide new queries with access to indexed historical state. In constrast, shared arrangements (like database indices) allow for post-hoc sharing: new queries can immediately attach to the in-memory arrangements of existing queries, and quickly start producing correct outputs that reflect all prior events.

Philosophically closest to shared arrangements is STREAM [9], a relational stream processor which maintains "synopses" (often indexes) for operators and shares them between operators. In contrast to shared arrangements, STREAM synopses lack features necessary for coarse-grained data-parallel incremental view maintenance: STREAM synopses are not multiversioned and do not support sharding for data-parallelism. STREAM processes records one-at-a-time; shared arrangements expose a stream of shared, indexed batches to optimized implementations of the operators.

Shared arrangements 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 in the context of streaming dataflow systems which provide certain core functionality. We enumerate the requirements in §3.1, and describe how several popular systems meet those requirements in §3.2. Our implementation builds on Timely Dataflow [2], which offers performant implementations of key abstractions required by shared arrangements §3.5. With this context, §3.4 shows how shared arrangements support deployment and continual maintenance of multiple queries against evolving data with the example of TPC-H Q3 and Q5.

3.1 Time-aware Dataflow

We designed shared arrangements for use in streaming dataflow computations that implement incrementally maintained queries on high-rate streams. 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.

Operator State. 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 operators, the state is 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.

Data Parallelism. 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 partition. 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; these index shards, taken collectively, represent the same index a single operator instance would maintain.

Logical Timestamps. Updates flow through a dataflow graph asynchronously. Concurrent updates may race along the multiple paths (and even cycles) between dataflow operators potentially distributed across multiple threads of control, 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. Many systems assign a logical timestamp to messages, either explicitly or implicitly through their scheduling mech-

anisms. At the same time, systems need to inform operators in the dataflow graphs when each logical time has "passed", in the sense that it will not again appear on messages input to the operator. With logical timestamps on messages and timestamp progress statements from the system, operators can maintain clear semantics even with asynchronous, non-deterministic execution.

We use the terminology of Timely Dataflow to describe progress statements and their consequences. Timely Dataflow reports timestamp progress information to each operator input by a *frontier*: a set of logical timestamps. We say a time is *beyond* a frontier when it is greater than or equal to some element of the frontier. A system should guarantee that all future timestamps received at an operator input are beyond the frontier most recently reported by the system, and that these reports should only advance (i.e., elements of a frontier should each be beyond the prior frontier).

3.2 Time-aware Dataflow Systems

Several dataflow systems are time-aware, either implicitly or explicitly. We now give examples to relate the concepts for readers familiar with these systems. Shared arrangements can be implemented in each of these systems, but our implementation will benefit from specific system details, which we explain in §3.5.

Spark Streaming [40] partitions logical time into small batches, and for each batch evaluates an entire dataflow. Spark Streaming therefore implicitly provides logical timestamps, with progress indicated by the scheduling of an operator. Spark Streaming operators do not have long-lived state, but each invocation can read an input corresponding to its prior state and write an output for its updated state, at greater expense than updating in-memory state.

Flink [12] is a streaming dataflow system that timestamps each message, and flows control messages, called *low watermarks*, in-band with data messages. A "watermark" for a timestamp t indicates that all messages that follow have timestamps greater or equal to t. Flink operators can have long-lived state, and can themselves be the result of sharding a larger dataflow operator.

Timely Dataflow is a model for data-parallel dataflow execution, introduced by Naiad [28]. Each Timely Dataflow operator is sharded across all workers, with data exchanged between workers for dataflow edges where the destination operator requires it. In Timely Dataflow, all data carries a logical timestamp, and workers exchange timestamp progress statements out-of-band. Workers independently determine frontiers for each of their hosted operators.

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 multiple concurrent dataflows, serving concurrent continuous queries. Shared arrangements substitute for per-instance operator state in the dataflow, and should appear to an individual operator as if it was a private copy of its state. Across operators, the shared arrangement's semantics are identical to maintaining individual copies of the indexed state in each operator. At the same time, the shared arrangement permits index reuse between operators that proceed at a different pace due to asynchrony in the system.

Operators that provide incremental view maintenance, so that their output continually reflects their accumulated input updates, offer particularly good opportunities for sharing state. This is because each stream of updates has one logical interpretation: as an accumulation of all updates. When multiple such operators want to build the same state, but vary what subset to read based on the time t they are currently processing, they can share arrangements instead. We assume that developers specify their dataflows using existing interfaces, but that they (or an optimizing compiler) explicitly indicate which dataflow state to share among which operators.

Collection trace

```
(data=(id=342, "Company LLC", "USA"),         time=4350, diff=+1)
(data=(id=563, "Firma GmbH", "Deutschland"), time=4355, diff=+1)
(data=(id=225, "Azienda SRL", "Italia"),      time=4360, diff=+1)
(data=(id=225, "Azienda SRL", "Italia"),      time=6200, diff=-1)
(data=(id=225, "Company Ltd", "UK"),          time=6220, diff=+1)
```

Collection at time $t = 4360$

```
(data=(id=342, "Company LLC", "USA"),         diff=+1)
(data=(id=563, "Firma GmbH", "Deutschland"), diff=+1)
(data=(id=225, "Azienda SRL", "Italia"),      diff=+1)
```

Collection at time $t = 6230$

```
(data=(id=342, "Company LLC", "USA"),    diff=+1)
(data=(id=563, "Firma", "Deutschland"), diff=+1)
(data=(id=225, "Company Ltd", "UK"),     diff=+1)
```

Figure 3: Update triples incoming to an operator, a "collection trace", and the resulting collection view at different times.

A shared arrangement exposes different *versions* of the underlying state to different operators, depending on their current time t. The arrangement therefore emulates, atop physically shared state, the separate indexes that operators would otherwise keep. Specifically, shared arrangements maintain state for operators whose state consists of the input collection (i.e., the cumulative streaming input). Following Differential Dataflow [27] terminology, a *collection trace* is the set of update triples *(data, time, diff)* that define a collection at time t by the accumulation of those inputs *(data, diff)* for which *time* $\leq t$ (Figure 3). Each downstream operator selects a different view based on a different time t of accumulation. Formal semantics of differential dataflow operators are presented in [7].

An explicit, new `arrange` operator maintains the multiversioned 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 for a future logical time if the operator has yet to process them.

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 change of key (an `exchange` operator in Differential Dataflow, or a "shuffle" in Flink), an arrangement-unaware operator (e.g., `map`, which may change the key), or an operator that explicitly materializes a new collection.

3.4 Shared Arrangements Example

We illustrate a concrete use of shared arrangements with the example of TPC-H Q3 and Q5. Recall that in our target setting, analysts author and execute SQL queries against heavily normalized datasets. Relations in analytics queries are commonly normalized into "fact" and "dimension" tables, the former containing foreign keys into the latter. While new facts (e.g., ad impressions, or line items in TPC-H) are continually added, the dimension tables are also updated (for example, when a customer or supplier updates their information). The dimension tables are excellent candidates for arrangement by primary keys: we expect many uses of these tables to be joins by primary keys, and each time this happens an arrangement can be shared rather than reconstructed.

TPC-H Q3 retrieves the ten unshipped orders with the highest value. This is a natural query to maintain, as analysts work to unblock a potential backlog of valuable orders. The query derives

from three relations—`lineitem`, `orders`, and `customer`—joined using the primary keys on `orders` and `customer`. A dataflow would start from `lineitem` and join against `orders` and `customer` in sequence. TPC-H Q5 lists the revenue volume done through local suppliers, and derives from three more relations (`supplier`, `nation`, and `region`). Each relation other than `lineitem` is joined using its primary key. A dataflow might start from `lineitem` and join against dimension tables in a sequence that makes a foreign key available for each table before joining it. In both queries, each dimension table is sharded across workers by their primary key.

The two queries do not have overlapping subqueries—each has different filters on order dates, for example—but both join against `orders` and `customer` by their primary keys. Deployed on the same workers, we first apply `arrange` operators to the `orders` and `customer` relations by their primary keys, shuffling updates to these relations by their key and resulting in shareable arrangements. In separate dataflows, Q3 and Q5 both have `join` operators that take as input the corresponding arrangement, rather than the streams of updates that formed them. As each arrangement is pre-sharded by its key, each worker has only to connect its part of each arrangement to its dataflow operators. Each worker must still stream in the `lineitem` data but the time for the query to return results becomes independent of the sizes of `orders` and `customer`.

3.5 System Features Supporting Efficiency

Shared arrangements apply in the general dataflow setting described in §3.1, and can benefit any system with those properties. But additional system properties can make an implementation more performant. We base our implementation on frameworks (Timely and Differential Dataflow) with these properties.

Timestamp batches. Timestamps in Timely Dataflow only need to be *partially ordered*. The partial order of these timestamps allows Timely Dataflow graphs to avoid unintentional concurrency blockers, like serializing the execution of rounds of input (Spark) or rounds of iteration (Flink). By removing these logical concurrency blockers, the system can retire larger groups of logical times at once, and produce larger batches of updates. This benefits DD because the atoms of shared state can increase in granularity, and the coordination between the sharing sites can decrease substantially. Systems that must retire smaller batches of timestamps must coordinate more frequently, which can limit their update rates.

Multiversioned state. Differential Dataflow has native support for *multiversioned* state. This allows it to work concurrently on any updates that are not yet beyond the Timely Dataflow frontier, without imposing a serial execution order on updates. Multiversioned state benefits shared arrangements because it decouples the execution of the operators that share the state. Without multiversioned state, operators that share state must have their executions interleaved for each logical time, which increases coordination.

Co-scheduling. Timely Dataflow allows each worker to host an unbounded number of dataflow operators, which the worker then schedules. This increases the complexity of each worker compared to a system with one thread per dataflow operator, but it increases the efficiency in complex dataflows with more operators than system threads. Co-scheduling benefits shared arrangements because the state shared between operators can be partitioned between worker threads, who do not need mutexes or locks to manage concurrency. Systems that cannot co-schedule operators that share state must use inter-thread or inter-process mechanisms to access shared state, increasing complexity and the cost.

Incremental Updates. Differential dataflow operators are designed to provide incremental view maintenance: their output updates continually reflects their accumulated input updates. This restriction from general-purpose stream processing makes it easier to compose dataflows based on operators with clear sharing semantics. Systems that provide more general interfaces, including Timely Dataflow, push a substantial burden on to the user to identify operators that can share semantically equivalent state.

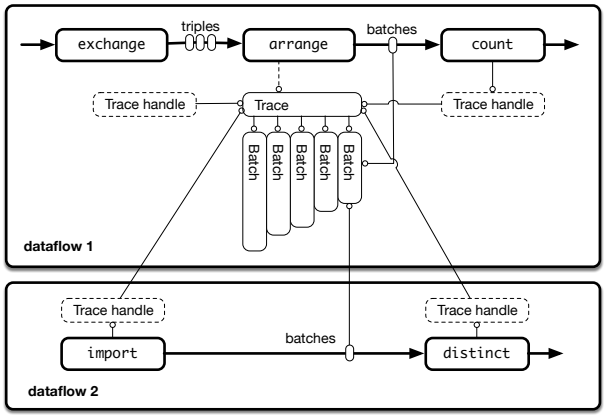


Figure 4: A worker-local overview of arrangement. Here the arrangement is constructed for the `count` operator, but is shared with a `distinct` operator in another dataflow. Each other worker performs the same collective data exchange, followed by local batch creation, trace maintenance, and sharing.

4. IMPLEMENTATION

Our implementation of a shared arrangement consists of three inter-related components:

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

Each shared arrangement has its updates partitioned by the key of its index, across the participating dataflow workers. This same partitioning applies to the trace, the `arrange` operator, and the read handles, each of whose interactions are purely *intra*-worker; each worker maintains and shares its *shard* of the whole arrangement. The only inter-worker interaction is the pre-shuffling of inbound updates which effects the partition.

Figure 4 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, 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 an input to the dataflow or from the output of another dataflow operator. Although update triples arrive

continually, an `arrange` operator learns that updates for a subset of times are complete only as the Timely Dataflow frontier advances.

In our design and implementation 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 beyond the lower frontier and not beyond of the upper frontier. The upper frontier of each batch matches the lower frontier of the next batch, and the growing list of batches reports the developing history of confirmed 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). Background merge computation (performed by the `arrange` operator) ensures that at any time, a trace consists of logarithmically many batches, which ensures 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. Each handle has an associated frontier, and ensures that it provides correct views of the index for any times beyond this frontier. Trace readers advance the frontier of their trace handle when they no longer require certain historical distinctions, which allows the `arrange` operator to compact batches by coalescing updates at older times, and to maintain a bounded memory footprint as a collection evolves.

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 beyond 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.

Batch implementation. Each batch is immutable, but indexed to provide efficient random access. Our default implementation sorts update triples *(data, time, diff)* first by *data* and then by *time*, and stores the fields each in its own column. This balances the performance of read latency, read throughput, and merge throughput. We have other batch implementations for specific domains (e.g., graphs), and new user implementations can be added without changing the surrounding superstructure. Most OLTP index structures are more general than needed for our immutable batches, but many of their data layout ideas could still be imported.

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 processed in steps: for each new batch, we perform work proportional to the batch size on 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 maximum 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 [28], but the mathematics of compaction have not been reported previously. Our extended technical report [26] contains proofs of optimality and correctness.

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 handles the operator will continue to produce batches but cease updating the trace. This optimization is crucial 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 beyond 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 beyond 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, and in extreme cases discard it entirely.

A trace handle has an `import` method that, in a new dataflow, creates an arrangement 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 importing the trace. Computations require no special logic or modes to accommodate attaching to in-progress streams; imported 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 representations lead to different operator designs, where the arrangement-based designs can be substantially more efficient than traditional record-at-a-time operators. This section explains simple and more complex arrangement-aware operator designs.

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, as an aggressive filter may reduce the data volume 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: they can filter an arrangement, or 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. One 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 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 iterative processing of static graphs.

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 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 output, either buffered for transmission or (as in our prototype) sent to destination workers but buffered at each awaiting reduction. Instead, operators respond to input batches by producing "futures", limited batches of computation that can each be executed until sufficiently many outputs are produced, and then suspend. Futures make copies of the shared batch and trace references they use, which avoids blocking 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 (as the least upper bound of two times does not need to be one of the times). Hence, 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. EVALUATION

We evaluate DD on end-to-end workloads to measure the impact of shared arrangements with regards to query installation latency, throughput, and memory use (§6.1). We then use microbenchmarks with DD to characterize our design's performance and the arrangement-aware operator implementations (§6.2). Finally, we evaluate DD on pre-existing benchmarks across multiple domains to check if DD maintains high performance compared to other peer systems with and without using shared arrangements (§6.3).

Implementation. We implemented shared arrangements as part of DD, our stream processor. DD is our reference Rust implementation of Differential Dataflow [27] with shared arrangements. It consists of a total of about 11,700 lines of code, and builds on an open-source implementation of Timely Dataflow [2].

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. Rust's static typing ensure that developers cannot incorrectly mix ordinary update triples and streams of arranged batches.

Setup. We evaluate DD on a four-socket NUMA system with four Intel Xeon E5-4650 v2 CPUs, each with 10 physical cores and 512 GB of aggregate system memory. We compiled DD with `rustc` 1.33.0 and the `jemalloc` [5] allocator. DD 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 DD only if we are executing it on comparable or less powerful hardware than the other systems had access to.

6.1 End-to-end performance impact

We start with an evaluation of shared arrangements in DD, in two domains with interactively issued queries against incrementally updated data sources. We evaluate the previously described

streaming TPC-H setup, which windows the `lineitem` relation, as well as a recent interactive graph analytics benchmark. For the relational queries, we would hope to see shared arrangements reduce the installation latency and memory footprint of new queries when compared to an instance of DD that processes queries independently. For the graph tasks, we would hope that shared arrangements reduce the update and query latencies at each offered update rate, increase the peak update rate, and reduce the memory footprint when compared to an instance of DD that processes queries indepedently. In both cases, if shared arrangements work as designed, they should increase the capacity of DD on fixed resources, reducing the incremental costs of new queries.

6.1.1 TPC-H

The TPC-H [6] 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 meaningful 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 of instantaneous events and do not arrange it.

TPC-H contains 22 "data warehousing" queries, meant to be run against large, static datasets. We consider a modified setup where the eight relations are progressively loaded [29], one record at a time, in a round-robin fashion among the relations (at scale factor 10).[1] To benchmark the impact of shared arrangements, we interactively deploy and retire queries while we load the eight relations. Each query has access to the full current contents of the seven keyed relations that we maintain shared arrangements for. By contrast, fulfillment events are windowed and each query only observes the fulfillment events from when it is deployed until when it is retired, implementing a "streaming" rather than a "historic" query. This evaluates the scenario presented in §1, where analysts interactively submit queries. We report performance for ten active queries.

The 22 TPC-H queries differ, but broadly either derive from the windowed `lineitem` relation and reflect only current fulfillments, or they do not derive from `lineitem` 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 expect queries of the first type to be quick to deploy, as their outputs are initially empty. Queries of the second type should take longer to deploy in either case, as their initial output depends on many records.

Query latency. To evaluate query latency, we measure the time from the start of query deployment until the initial result is ready to be returned. Query latency is significant because it determines whether the system delivers an interactive experience to human users, but also to dashboards that programmatically issue queries.

Figure 1a (shown in §1) reports the distribution of query installation latencies, with and without shared arrangements. With shared arrangements, most queries (those that derive from `lineitem`) deploy and begin updating in milliseconds; the five queries that do not derive from `lineitem` are not windowed and perform non-trivial computation to produce their initial correct answer: they take between 100ms and 1s, depending on the sizes of the relations they use. Without shared arrangements, almost all queries take 1–2 seconds to install as they must create a reindexed copy of their inputs. Q1 and Q6 are exceptions, since they use no relations other than `lineitem`, and thus avoid reindexing any inputs; shared arrangements cannot improve the installation latency of these queries. We conclude that shared arrangements substantially reduce the majority of query installation latencies, often by several orders of magnitude. The improvement to millisecond latency brings responses within interactive timescales, which helps improve productivity of human analysts and intervential latency for dependent software.

Update latency. Once a query is installed, DD continually 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.

Figure 1b presents the distribution of these times, with and without shared arrangements, as a complementary cumulative distribution function (CCDF). The CCDF 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 shared arrangements. Shared arrangements yield lower latencies and increase update throughput.

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 1c 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. Consequently, with shared arrangements, a given amount of system memory should allow for more active queries. In this experiment, ten concurrent queries are installed; workloads with more concurrent queries may have more sharing opportunities and achieve further memory economies.

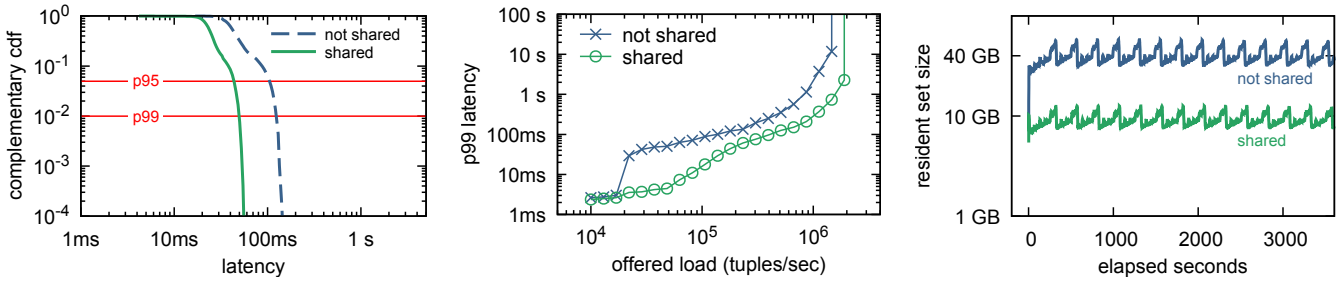
6.1.2 Interactive graph queries

We further evaluate DD with an open-loop experiment issuing queries against an evolving graph. This experiment issues the four queries used by Pacaci et al. [30] to compare 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 [14] and is common in stream processors, allowing them to treat queries as a streaming input. The transformation can be applied to any queries presented as prepared statements. The dataflows depend on two arrangements of the graph edges, by source and by target; they are the only shared state among the queries.

We use a graph with 10M nodes and 64M edges, and update the graph and 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

[1] We focus on shared arrangements here, but DD matches or outperforms DBToaster [29] even when queries run in isolation [26].



(a) Latencies for query mix. **(b)** 99th percentile latency at given offered load. **(c)** Resident set size.

Figure 5: 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).

between the four query types.

System	#	look-up	one-hop	two-hop	4-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
DD, 10^0	32	0.64ms	0.92ms	1.28ms	1.89ms
DD, 10^1	32	0.81ms	1.19ms	1.65ms	2.79ms
DD, 10^2	32	1.26ms	1.79ms	2.92ms	8.01ms
DD, 10^3	32	5.71ms	6.88ms	10.14ms	72.20ms

Figure 6: On comparable 10M node/64M edge graphs, DD is broadly competitive with the average graph query latencies of three systems evaluated by Pacaci et al. [30], and scales to higher throughput using batching. The DD batch size is the number of concurrent queries per measurement.

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

Figure 5a 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.

Update throughput. To test how DD'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 5b 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 unshared, so we would expect a reduction in memory footprint of somewhat below 4×.

Figure 5c 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.

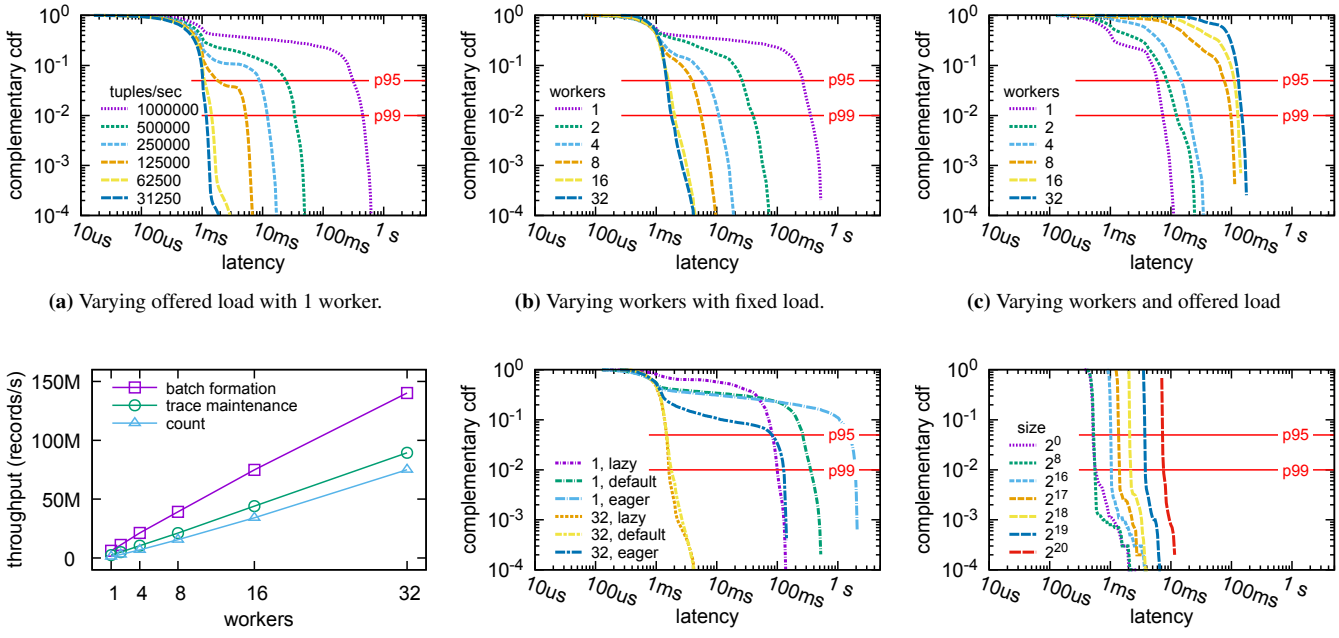
6.1.3 *Comparison with other systems*

Pacaci et al. [30] evaluated relational and graph databases on the same graph queries. DD 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 requires that queries be expressed as prepared statements, a restriction the other systems do not impose.

We ran DD experiments with a random graph comparable to the one used in Pacaci et al.'s comparison. Figure 6 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 DD. While DD does not provide the lowest latency for point look-ups, it does provides low latencies for other queries and increases query throughput with batch size.

6.2 Design evaluation

We now perform microbenchmarks of the `arrange` operator, to evaluate its response to changes in load and resources. 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 statis-



(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 7: 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.

tic 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 7a 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. Latencies drop as load decreases, down to the test harness's limit of one millisecond. This demonstrates that arrangements are suitable for both low-latency and high-throughput.

Strong scaling. More parallel workers should allow faster maintenance of a shared arrangement, as the work to update it parallelizes, unless coordination frequency interferes. Figure 7b 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 7c shows that the latency distributions do exhibit increased tail latency, as the act of data exchange at the arrangement input becomes more complex. However, the latencies do stabilize at 100–200ms as the number of workers and data increase proportionately.

Throughput scaling. An arrangement consists of several subcomponents: batch formation, trace maintenance, and e.g., a maintained `count` operator. To evaluate throughput scaling, we issue batches of 10,000 updates at each worker, repeated as soon as each batch is accepted, rather than from a rate-limited open-loop harness. Figure 7d 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. Eager merging performs the least work overall but can increase tail latency. Lazy merging performs more work overall, but should reduce the tail latency. Ideally, DD's default would pick a good tradeoff between common-case and tail latencies at different scales.

Figure 7e reports the latency distributions for one and 32 workers, each with three different merge amortization coefficients: the most eager, DD'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. The lazier settings are critical for effective strong scaling, where eager work causes multiple workers to seize up, which matches similar observations about garbage collection at scale [20]. DD's default setting achieves good performance at both scales.

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

The varying lines in Figure 7f demonstrate that the join work is indeed proportional to the small collection's size, rather than to the (constant) 10M arranged keys. This behavior is not possible in a record-at-a-time stream processor, which must at least examine each input record. This behavior is possible in DD only because the `join` operator receives as input pre-arranged batches 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 eval. (no SA)	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 eval. (no SA)	56.45s	0.60s	19.85s

Figure 8: DD enables interactive computation of three Datalog queries (32 workers, medians and maximums over 100 queries). Full evaluation is required without shared arrangements.

updates. Query deployment in the TPC-H workload would not be fast without this property.

6.3 Baseline performance on reference tasks

We also evaluate DD against established prior work to demonstrate that DD is competitive with and occasionally better than peer systems. Importantly, these established benchmarks are traditionally evaluated in isolation, and are rarely able to demonstrate the benefits of shared arrangements. Instead, this evaluation is primarily to demonstrate that DD does not *lose* baseline performance as compared to other state-of-the-art systems. Most but not all of the peer systems in this section do maintain private indexed data in operators; this decision alone accounts for some of the gaps.

6.3.1 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 [4], DeALS [37], and several distributed systems have recently emerged, including Myria [35], SociaLite [32], and BigDatalog [33]. At the time of writing, only LogicBlox supports decremental updates to Datalog queries, using a technique called "transaction repair" [34]. DD supports incremental and decremental updates to Datalog computations and interactive top-down queries.

Top-down (interactive) evaluation. Datalog users commonly specify values in a query, such as *reach*(*"david"*, ?), to request nodes reachable from a source node. The "magic set" transformation [10] 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. DD, like some interactive Datalog environments, performs this work against maintained arrangements of 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 (or DD without shared arrangements) require.

Figure 8 reports DD'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. DD's arrangements mostly reduce runtimes from seconds to milliseconds. The slower performance for sg(x,?) on grid-150 reveals that the transformation is not always beneficial, a known problem with the magic set transform.

System	cores	*linux*	*psql*	*httpd*
SociaLite	4	OOM	OOM	4 hrs
Graspan	4	713.8 min	143.8 min	11.3 min
RecStep	20	430s	359s	74s
DD	1	65.8s	32.0s	8.9s

(a) *dataflow* query, DD on laptop hardware.

System	cores	*linux*	*psql*	*httpd*
RecStep	20	430s	359s	74s
DD	2	53.9s	25.5s	7.5s
DD	4	34.8s	16.3s	4.7s
DD	8	24.4s	11.2s	3.2s
DD	16	20.7s	8.7s	2.5s

(b) *dataflow* query, DD on server hardware.

System	cores	*linux (kernel only)*	*psql*	*httpd*
DD (med)	1	1.05ms	143ms	18.1ms
DD (max)	1	7.34ms	1.21s	201ms

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

Figure 9: DD performs well for Graspan [36] *dataflow* query on three graphs. SociaLite and Graspan results from Wang et al. [36]; RecStep results from Fan et al. [17]; OOM: out of memory.

Bottom-up (batch) evaluation. In our extended technical report [26], we compare DD to distributed and shared-memory Datalog engines, using their benchmarks and datasets ("transitive closure" and "same generation" on trees, grids, and random graphs). Our results show that DD generally outperforms the distributed systems and is comparable to the best shared-memory engine (DeALS).

6.3.2 Program Analysis

Graspan [36] is a system built for static analysis of large code bases, 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* [36]. 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 full implementation of Graspan—query parsing, dataflow construction, input parsing and loading, dataflow execution—in 179 lines of code on top of DD.

Graspan is designed to operate out-of-core, and explicitly manages its data on disk. We therefore report DD 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 about 20% faster than the out-of-core execution.

Figure 9a and Figure 10a show the running times reported by Wang et al. compared to those DD achieves. For both queries, we see a substantial improvement (from 24× to 650×). 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 more efficient computation, but one that reuses some relations several (five) times; the benefits of the improved plan may not be realized by systems without shared arrangements. Figure 10a reports the optimized running times as (Opt).

System	cores	*linux*	*psql*	*httpd*
SociaLite	4	OOM	OOM	> 24 hrs
Graspan	4	99.7 min	353.1 min	479.9 min
RecStep	20	61s	162s	162s
DD	1	241.0s	151.2s	185.6s
DD (Opt)	1	121.1s	52.3s	51.8s

(a) *points-to* analysis, DD on laptop. DD (Opt) is an optimized query.

System	cores	*linux*	*psql*	*httpd*
RecStep	20	61s	162s	162s
DD	2	230.0s	134.4s	145.3s
DD	4	142.6s	73.3s	80.2s
DD	8	86.0s	40.9s	44.9s
DD	16	59.8s	24.0s	27.5s
DD (Opt)	2	125.2s	53.1ss	46.0s
DD (Opt)	4	89.8s	30.8s	26.7s
DD (Opt)	8	57.4s	18.0s	15.1s
DD (Opt)	16	43.1s	11.2s	9.1s

(b) *points-to* analysis, DD on server. DD (Opt) is an optimized query.

Figure 10: DD performs well for Graspan [36] program analyses on three graphs. SociaLite and Graspan results from Wang et al. [36]; RecStep results from Fan et al. [17]; OOM: out of memory.

In Figure 9b and Figure 10b we also report the runtimes of DD on these program analysis tasks on server hardware (with the same hardware configuration as previous sections) and compare them to RecStep [17], a state-of-the-art parallel datalog engine. For all queries, DD matches or outperforms RecStep running times even when it is configured to utilize a smaller number of CPU cores.

Top-down evaluation. Both *dataflow* and *points-to* can be transformed to support interactive queries instead of batch computation. Figure 9c 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 an improved developer experience.

6.3.3 Batch graph computation

We evaluate DD on standard batch iterative graph computations on three standard social networks: LiveJournal, Orkut, and Twitter. We report results for the largest of the graphs, Twitter, in Figure 11; results for LiveJournal and Orkut are available in our extended technical report [26]. Following prior work [33] we use the tasks of single-source reachability (reach), single-source shortest paths (sssp), and undirected connectivity (wcc). For the first two problems we start from the first graph vertex with any outgoing edges (each reaches a majority of the graph).

We separately report the times required to form the forward and reverse edge arrangements, with the former generally faster than the latter as the input graphs are sorted by the source as in the forward index. The first two problems require a forward index and undirected connectivity requires indices in both directions, and we split the results accordingly. We include measurements by Shkapsky et al. [33] for several other systems. We also report running times for simple single-threaded implementations that are not required to follow the same algorithms. For example, for undirected connectivity we use the union-find algorithm rather than label propagation, which outperforms all systems except DD at 32 cores. We also include single-threaded implementations that replace the arrays storing per-node state with hash maps, as they might when the graph identifiers have not been pre-processed into a compact range; the graphs remain densely packed and array indexed.

System	cores	index-f	reach	sssp	index-r	wcc
Single thread	1	-	14.89s	14.89s	-	33.99s
w/hash map	1	-	192.01s	192.01s	-	404.19s
BigDatalog	120	-	125s	260s	-	307s
Myria	120	-	102s	1593s	-	1051s
SociaLite	120	-	755s	OOM	-	OOM
GraphX	120	-	3677s	6712s	-	12041s
RaSQL	120	-	45s	81s	-	108s
RecStep	20	-	174s	243s	-	501s
DD	1	162.41s	256.77s	310.63s	312.31s	800.05s
DD	2	99.74s	131.50s	159.93s	164.12s	417.20s
DD	4	49.46s	64.31s	77.27s	81.67s	200.28s
DD	8	27.99s	33.68s	40.24s	43.20s	101.42s
DD	16	18.04s	17.40s	20.99s	24.73s	51.83s
DD	32	12.69s	11.36s	10.97s	14.44s	27.48s

Figure 11: System performance on various tasks on the 42M node, 1.4B edge `twitter` graph. DD does not share any arrangements here, but the sharing infrastructure does not harm performance.

DD is consistently faster than the other systems—Myria [35], BigDatalog [33], SociaLite [32], GraphX [21], RecStep [17], and RaSQL [22]—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, DD becomes competitive between two and four cores. These results are independent of shared arrangements, but indicate that DD's arrangement-aware implementation does not impose any undue cost on computations without sharing.

7. CONCLUSIONS

We described shared arrangements, detailed their design and implementation in DD, and showed how they yield improved performance for interactive analytics against evolving data. Shared arrangements enable interactive, incrementally maintained queries against streams by sharing sharded indexed state between operators within or across dataflows. Multiversioning the shared arrangement is crucial to provide high throughput, and sharding the arrangement achieves parallel speedup. Our implementation in DD 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 workloads. In particular, we showed that shared arrangements improve performance for workloads with concurrent queries, such as a streaming TPC-H workload with interactive analytic queries and concurrent graph queries.

Shared arrangements rely on features shared by time-aware dataflow systems, and the idiom of a single-writer, multiple-reader index should apply to several other popular dataflow systems. We left undiscussed topics like persistence and availability. As a deterministic data processor, DD is well-suited to active-active replication for availability in the case of failures. In addition, the immutable LSM layers backing arrangements are appropriate for persistence, and because of their inherent multiversioning can be persisted asynchronously, off of the critical path.

DD [1] is the reference open-source implementation of Differential Dataflow, used by several research groups and companies.

Acknowledgements. We thank Natacha Crooks, Jon Howell, Michael Isard, and the MIT PDOS group for their valuable feedback, and the many users of DD who exercised and informed its design. This work was partly supported by Google, VMware, and the Swiss National Science Foundation. Andrea Lattuada is supported by a Google PhD fellowship.

8. REFERENCES

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

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

[3] DDlog. https://research.vmware.com/projects/differential-datalog-ddlog.

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

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

[6] The TPC-H decision support benchmark. http://www.tpc.org/tpch/default5.asp.

[7] M. Abadi, F. McSherry, and G. Plotkin. Foundations of differential dataflow. In A. Pitts, editor, *Foundations of Software Science and Computation Structures*, Lecture Notes in Computer Science, pages 71–83. Springer Berlin Heidelberg, 2015.

[8] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic. Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views. *PVLDB*, 5(10):968–979, 2012.

[9] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. *STREAM: The Stanford Data Stream Management System*, pages 317–336. Springer, Berlin/Heidelberg, Germany, 2016.

[10] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *Proceedings of the 5th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS)*, pages 1–15, 1986.

[11] G. Candea, N. Polyzotis, and R. Vingralek. A scalable, predictable join operator for highly concurrent data warehouses. *PVLDB*, 2(1):277–288, 2009.

[12] P. Carbone, S. Ewen, S. Haridi, A. Katsifodimos, V. Markl, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *IEEE Data Engineering*, 38(4), Dec. 2015.

[13] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. Telegraphcq: Continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 668–668, 2003.

[14] J. Chen, D. J. DeWitt, F. Tian, and Y. 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.

[15] E. Darling. Locks taken during indexed view modifications. Brent Ozar Unlimited Blog, https://www.brentozar.com/archive/2018/09/locks-taken-during-indexed-view-modifications/, Sept. 2019.

[16] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, Jan. 2008.

[17] Z. Fan, J. Zhu, Z. Zhang, A. Albarghouthi, P. Koutris, and J. M. Patel. Scaling-up in-memory datalog processing: Observations and techniques. *PVLDB*, 12(6):695–708, 2019.

[18] G. Giannikis, G. Alonso, and D. Kossmann. Shareddb: Killing one thousand queries with one stone. *PVLDB*, 5(6):526–537, 2012.

[19] J. Gjengset, M. Schwarzkopf, J. Behrens, L. T. Araújo, M. Ek, E. Kohler, M. F. Kaashoek, and R. 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, Oct. 2018.

[20] I. Gog, J. Giceva, M. Schwarzkopf, K. Vaswani, D. Vytiniotis, G. Ramalingan, D. Murray, S. Hand, and M. Isard. Broom: Sweeping out garbage collection from big data systems. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems (HotOS)*, 2015.

[21] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 599–613, 2014.

[22] J. Gu, Y. H. Watanabe, W. A. Mazza, A. Shkapsky, M. Yang, L. Ding, and C. Zaniolo. RaSQL: Greater Power and Performance for Big Data Analytics with Recursive-Aggregate-SQL on Spark. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD)*, page 467–484, 2019.

[23] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. 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.

[24] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. 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, Mar. 2007.

[25] J. Karimov, T. Rabl, and V. Markl. AStream: Ad-hoc Shared Stream Processing. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD)*, pages 607–622, 2019.

[26] F. McSherry, A. Lattuada, M. Schwarzkopf, and T. Roscoe. Shared arrangements: Practical inter-query sharing for streaming dataflows (extended technical report). https://arxiv.org/abs/1812.02639.

[27] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard. Differential dataflow. In *Proceedings of the 6th Biennial Conference on Innovative Data Systems Research (CIDR)*, Jan. 2013.

[28] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A Timely Dataflow System. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 439–455, Nov. 2013.

[29] M. Nikolic, M. Dashti, and C. 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.

[30] A. Pacaci, A. Zhou, J. Lin, and M. T. Özsu. Do we need specialized graph databases?: Benchmarking real-time social networking applications. In *Proceedings of the 5th International Workshop on Graph Data-management Experiences & Systems (GRADES)*, pages 12:1–12:7, 2017.

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

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

[33] A. Shkapsky, M. Yang, M. Interlandi, H. Chiu, T. Condie, and C. 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.

[34] T. L. Veldhuizen. Transaction repair: Full serializability without locks. https://arxiv.org/abs/1403.5645, 2014.

[35] J. Wang, T. Baker, M. Balazinska, D. Halperin, B. Haynes, B. Howe, D. Hutchison, S. Jain, R. Maas, P. Mehta, D. Moritz, B. Myers, J. Ortiz, D. Suciu, A. Whitaker, and S. Xu. The myria big data management and analytics system and cloud services. In *Proceedings of the 8th Biennial Conference on Innovative Data Systems Research (CIDR)*, Jan. 2017.

[36] K. Wang, A. Hussain, Z. Zuo, G. Xu, and A. Amiri Sani. Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 389–404, 2017.

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

[38] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. 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)*, Dec. 2008.

[39] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. 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, Apr. 2012.

[40] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 423–438, Nov. 2013.