K-Pg: Shared State in Differential Dataflows

Paper # 86, 12 pages

Abstract

Many of the most popular scalable data-processing frameworks are fundamentally limited in the generality of computations they can express and efficiently execute. These limitations result in a surprising absence of *algorithms* from domains where the scales of data call for them most. In this paper we present the design and implementation of a data-processing framework that is general, scalable, and responsive. Our evaluation indicates that our system's performance is comparable to (and can even exceed) specialized systems across multiple domains, while at the same time significantly generalizing their capabilities.

1 Introduction

The landscape of systems for computing on "big data" is fractured into many specialized systems, each with their own capabilities and limitations. Batch processors [9, 14, 27] support general computations, but are expensive for random access to mutable state. Stream processors [6, 16, 23] support high rates of changes, but only for the equivalent of straight-line programs: directed acyclic graphs. Graph processors [8, 11, 28] support iterative computation, but for static or infrequently updated input data. New specialized systems emerge regularly, recently to handle mutually recursive tasks in Datalog evaluation [22] and program analysis [25], because existing systems are not sufficient for increasingly sophisticated computations.

Our goal in this paper is to develop a broadly capable data processing system. We are motivated by the number of specialized systems that emerge for new tasks, each introducing some novel but narrow functionality, that nonetheless re-use common principles. We have been encouraged by interactions with non-systems builders who have needs that hybrize those of current offerings, but which are not fundamentally intractable. We have taken these experiences and distilled them into a system that covers a large class of computations, including all those mentioned above and more, built from just a small number of re-usable components.

Our system is based on differential dataflow [17], a data-parallel processing paradigm in which users express their computation using high-level operations on collections of records, and which then responds to arbitrary input changes with the corresponding output changes. These operations include relational, data-parallel, and iterative constructs, sufficient for SQL-based relational analytics, MapReduce dataflows, iterative graph computation, and mutually recursive Datalog evaluation. Its execution as a dataflow of data-parallel operators can be effectively distributed across multiple workers, and its incremental semantics support low-latency and high-rate updates. Although differential dataflow has a reference implementation, as part of the Naiad [18] project, several aspects of this implementation (common to most data processors) limit its generality and performance.

This paper describes K-Pg, a complete re-implementation of differential dataflow, based on a fundamentally re-designed dataflow processor. Whereas traditional big data processors create and manage dataflows of independent operators, K-Pg exposes and *shares* indexed state between operators. Such sharing is common in database systems, but largely absent from scalable systems like Hadoop, Spark, Flink, and indeed Naiad. Such sharing not only reduces communication, computation, and memory use, but also leads to higher-throughput operator implementations and more responsive deployment of new dataflows. K-Pg's adherence to a specific data and computational model brings structure and meaning to its shared state, and its system architecture makes such sharing effecient.

Several changes are required to a conventional dataflow processor to realize the performance benefits of shared indexed state. We introduce a new dataflow operator `arrange` that maintains a worker-local shared index with high throughput, low latency, and a compact representation, using minimal coordination; we detail this design in Section 3. Readers of shared indices must take care to restrict how they interpret the index, and to indicate historical detail they no longer require so that the index may be compacted; we detail the arrangements interface in Section 3. Finally, several dataflow operators can be re-written to operate more efficiently on pre-indexed input batches; we describe these implementations in Section 4. These changes represent the bulk of K-Pg by volume, which otherwise runs on an unmodified timely dataflow runtime [1].

We implemented K-Pg in Rust, and experimentally evaluate it on a range of computations with different per-

formance requirements. We evaluate K-Pg on benchmark tasks in relational analytics, graph processing, and Datalog evaluation, where we confirm that its performance is comparable to and in some cases better than other systems in their target domains. We also evaluate K-Pg on tasks supported by relatively fewer and more specialized systems, including interactive graph navigation, program analysis, and goal-driven evaluation, where we find that framing queries as differential dataflow computations can gives us lower latency and higher throughput. Despite the variety of tasks and requirements, these are each implemented as idioms in one sufficiently expressive and performant system.

Contributions. K-Pg is our attempt to provide a computational framework that is general, scalable, and responsive. This paper summarizes the design and implementation of K-Pg, with the specific intended contributions of:

1. A data-parallel dataflow design with indexed state shared between operators in multiple dataflows (§ 2).
2. A multi-versioned shared index design with high throughput, low read and write latency, and compact memory footprint (§ 3).
3. New operator implementations based on streams of shared indexed batches (§ 4),
4. An evaluation of K-Pg that indicates it can be as capable as specialized data processing systems in their target domains, while supporting a more general range of workloads than any, including some that cannot currently run on data-parallel systems (§ 5).

We conclude in Section 7 with a summary and thoughts for future research directions.

2 System design and background

K-Pg is built on an existing timely dataflow execution layer [1], and inherits its distributed execution design. K-Pg also borrows Naiad's differential dataflow design as a timely dataflow of operators that consume and produce collection updates, but it makes fundamental modifications to how this occurs. This section overviews necessary background about timely and differential dataflow, and then describes K-Pg's architectural changes.

2.1 Timely dataflow

Timely dataflow is a framework for data-parallel dataflow execution, introduced by Naiad [18]. 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, but structural requirements preclude cycles along which timestamps do not strictly increase.

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 inputs (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 operator shards that might profitably share the same indexed inputs.

In addition to scheduling operators and transporting data, the timely dataflow workers provide operator shards with bounds on the potential timestamps they may yet see at each of their inputs. This information comes 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. We say that a time is *in advance of* a frontier if it is greater than or equal to some element of the frontier. A frontier only ever advances, and informs operator shards when they have received all records with certain timestamps, at which point it may be appropriate to take some action.

A user program alternately interacts with dataflows, creates new dataflows, and schedules dataflow operators. Any number of dataflows can be run concurrently, but the set must be the same on all workers. Dataflows are automatically retired when they their inputs are closed and they contain no more messages. User code can programmatically construct dataflows, and nothing prevents the sharing of state within a worker (even across dataflows), other than a discipline for doing so.

2.2 Differential dataflow

Differential dataflow computations are initially defined as functional transformations of time-varying *collections*, but are then rendered to timely dataflow computations that propagate only changes to these collections. A user interacts with a differential dataflow computation by supplying timestamped input changes and observing the correspondingly timestamped output changes. Differential dataflow's responsibility is to maintain the functional relationships between the exogenous input collections and all intermediate and output collections, as those input collections change arbitrarily.

A differential dataflow collection is parameterized by three types, *Data*, *Time*, and *Diff*. The *Time* type must be a lattice (supporting less than, equals, least upper bound, and greatest lower bound) and the *Diff* type must be an

```
// create inputs.
let nodes = scope.new_input();
let edges = scope.new_input();

// define computation.
let reach = nodes.iterate(|inner|
    edges.semijoin(inner)
         .map(|(src,dst)| dst)
         .concat(nodes)
         .distinct()
);

// inspect and monitor output.
reach.inspect(/* per-record closure */)
     .probe()
```

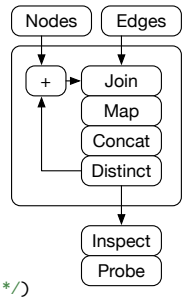


Figure 1: Graph reachability written in differential dataflow, and the resulting dataflow graph.

associative group (with a + operator and a zero element). A differential dataflow collection should be interpreted as a map from *Data* to *Diff* that can take on arbitrary values at each element of *Time*.

A differential dataflow collection is defined either as an input to the computation or by a functional transformation of other collections. Example transformations include standard operators like `map`, `filter`, `join`, and `group` (equivalent to MapReduce's `reduce`). Less standard, differential dataflow provides an `iterate` operator that subjects its input collection to a supplied differential dataflow fragment an unbounded number of times. The `iterate` operator is rendered to a timely dataflow subgraph, rather than a single dataflow operator, and extends all timestamps with a new "round of iteration" coordinate.

Operationally, a differential dataflow computation is rendered to a timely dataflow in which operators consume and produce streams of *(data, time, diff)* update triples. Each such stream defines a collection at all times t not in advance of its frontier, from the accumulation of update triples at times less or equal to t. Each differential dataflow operator's output stream should accumulate to the functional operator logic applied to the correspondingly accumulated input stream, for all times t not in advance of the input frontier. The essential difference from other streaming systems is that these times may be only partially ordered, which can result in more efficient differencing for iterative computations.

Figure 1 presents an example graph reachability computation. Two inputs `nodes` and `edges` are first defined, and then an iterative computation describes an iterative update rule for a set of nodes: joining with edges, re-introduce nodes, and retain the distinct elements. Both `nodes` and `edges` can then be interactively updated, introducing and perhaps changing a set of edges, and updating the set of source nodes to initiate queries. As the changes propagate, the `inspect` operator's closure will be called, and the `probe` reports its input frontier to confirm the completion of timestamps. Each worker maintains a shard of each operator, and is responsible for a fraction of the keyspace of each; the workers collectively exchange data along edges leading in to the `join` and `distinct` operators.

2.3 Design modifications

K-Pg's stateful operators implementations are broken in two parts: an `arrange` operator, which exchanges, batches, and indexes updates, and thinner shell operators which apply operator-specific logic using these shared indices. The `arrange` operator is new to differential dataflow (and would likely be new to other streaming systems), and the shell operators have substantially different (and often much simpler) designs given their input as indexed batches rather than streams of independent updates. These two elements require careful design to support high-throughput updates to shared state. We discuss the `arrange` operator in Section 3 and new operator designs in Section 4.

Our proposed architecture for K-Pg results from a set of design principles; constraints we impose to increase the likelihood that the computation executes robustly, despite (or perhaps because of) the absence of frustrating systems knobs. We will invoke these principles in our design discussions, and in all cases we view violations of any of these principles as problematic.

Principle 1: Decouple logical and physical batching. K-Pg computations consume, manipulate, and produce large volumes of updates at distinct logical times. These updates should be manipulated in large physical batches, with no artificial serialization imposed by the logical times.

Principle 2: Sequential memory traversal. The state managed by K-Pg represents historical data for large collections that may grow beyond the capacity of our fast random access memory. Access to operator state should be at worst one sequential pass (though ideally to a sparse subset).

Principle 3: Bounded memory footprint. K-Pg can produce large volumes of updates, but they may be to a relatively smaller number of distinct records. We should use memory proportional to the number of distinct *(data, time)* pairs in the system.

Principle 4: Operator work proportionality. The volume of data K-Pg operators consume and produce can vary substantially. Operator invocations should perform computation proportional to the number of output updates it might produce.

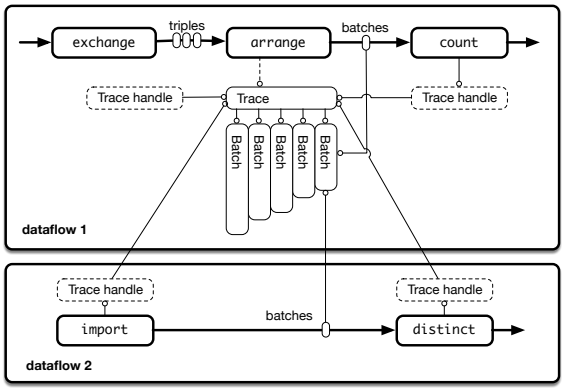


Figure 2: A worker-local overview of arrangement.

3 Arrangements

Our most substantial design departure from standard stream processors, returning somewhat to the design of relational databases, is our use and re-use of arranged streams of indexed data. We introduce a new `arrange` operator, which takes as input a stream of update triples and produces an *arrangement*: a pair of *(i)* a stream of shared indexed batches of updates and *(ii)* a shared, compactly maintained index of all produced updates. Arrangements allow K-Pg to spend the communication, computation, and memory required to arrange data once, and eliminate the cost all subsequent uses of the arrangement. This representation also provides a uniform approach to historical and streaming changes, where the former are simply presented as surprisingly large and promptly delivered update batches.

In this section we work through the details of the design and implementation of arrangements, and explain how they support shared indexed data under high update throughputs. Figure 2 sketches the elements and some uses of an arrangement, which we will further develop in the text.

3.1 Collection traces

Following prior work [17], 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 $time \leq t$.

Our design commits to a collection trace as 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 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.

A sequence of batches with lower and upper frontiers is self-describing, in that it can be understood without additional runtime support from the timely dataflow system. An independent timely dataflow, or other computation, can consume this collection trace representation and correctly understand the collection history.

3.2 The `arrange` operator

The `arrange` operator receives update triples, and is tasked with minting new immutable indexed batches of updates in response to advances in its input frontier and compactly maintaining the collection trace in response to advances in trace handle frontiers.

At a high level, the `arrange` operator buffers incoming updates until the input frontier advances, at which point it extracts and indexes all buffered updates not in advance of the advanced input frontier. A shared reference to the newly minted immutable batch is both added to the trace and emitted as output from the operator. As part of adding the batch to the trace, the operator may need to perform some maintenance to keep the trace representation compact and easy to navigate. Abstractly these tasks are not hard, but there are several details required to satisfy our design principles.

Input buffering. Incoming update triples are buffered in what is effectively a partially evaluated merge sort: a sequence of sorted lists of geometrically increasing size, which are merged when two are within a constant multiple in length. This representation allows us to coalesce updates with the same *(data, time)* fields, and ensures that we maintain a number of updates at most linear in the number of distinct *(data, time)* pairs (the longest list contains only distinct pairs, and all other lists accumulate to at most a constant multiple of its length).

Physical batching. K-Pg maintains a large collection of updates and informs the timely dataflow system only about changes to the lower frontier of update timestamps held. The input frontier can advance in large steps, and in response the `arrange` operator creates one update batch and involves the timely dataflow system only once, independent of the number of distinct logical times processed. This decouples the logical update rate from the physical batching, and is crucial to support high update rates. Limitations of Naiad's notification API prevent it from supporting physical batching of logical timestamps.

Shared references. Immutable batches are wrapped in reference-counted shared references, so that the batch and downstream consumers can reference the same underlying memory. The trace is also shared but it is not immutable. Importantly, the `arrange` operator has only a weak reference to the trace, so that should all readers drop their references, the trace (and its references to

batches) are dropped; the `arrange` operator will continue to produce indexed batches, but it will not (and cannot) add them to the now non-existent trace. This optimization is important for the performance of the `join` operator where one input has ceased changing (and we may cease maintaining the trace for its still-changing input), as seen often in processing static graph data.

Amortized trace maintenance. The `arrange` operator must maintain a compact representation that is easy for readers to navigate, even as we add batches to a trace. Our choice is to merge adjacent batches of comparable size, so that we maintain a number of batches at most logarithmic in the number of distinct `(data,time)` pairs. These merges happen on the same worker thread, which we do not want to block when large batches should be merged. Instead, we initialize but do not complete a merge, and for each introduced batch we apply an amount of effort proportional to its size to each in-progress merge. A large constant of proportionality performs merges eagerly and trades latency away for throughput, whereas a small constant provides lower latency but maintains more open batches for operators to navigate. A constant of at least one ensures that merges complete before their results are required for a new merge. When a merge completes the new batch is installed and references to the merged batches are dropped.

Consolidation. A trace can coalesce timestamps that are indistinguishable to all trace readers and consolidate updates at now indistinguishable times, in a process analogous to MVCC "vacuuming". Each reader's trace handle (described soon) maintains a frontier of times the trace must distinguish. When we initiate a merge we capture the lower bound F of all these times, and we replace each time t with a representative $rep_F(t)$ that compares identically to t for all times in advance of F. Updates with the same representative timestamp are consolidated.

Naiad's reference implementation performs the same consolidation, but does so in place and only in response to reading the state for the key. K-Pg performs consolidation at merges; regular churn ensures that this happens for all keys. Without a regular vacuuming mechanism, Naiad effectively leaks memory for cancellations of records that are not seen again. The mathematics of compaction do not appear to have been recorded previously, so we present the formal definition of $rep_F(t)$ and proofs of its optimality and correctness in the appendix.

Modularity. The `arrange` operator is defined in terms of a generic trace type. 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 distinct batches for *data* structured as *(key, val)* and just *key*, the latter with a simplified representation and navigation logic.

3.3 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 has a frontier-valued capability, and guarantees only that the accumulated collections will be correct when accumulated at times in advance of this frontier. The trace itself tracks outstanding trace handle capabilities, which indirectly inform it about times that are indistinguishable to all readers (and which can 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 downgrade the frontier capability on their trace handles and still function correctly. Some operators are also able to drop their trace handles entirely, notably the `join` operator when its opposite input ceases changing. These actions, downgrading and dropping capabilities, provide the trace with the opportunity to consolidate the trace representation.

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 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 downgraded. Full historical information means that computations do not require special logic or modes to accommodate attaching to incomplete in-progress streams; imported traces appear indistinguishable to the original streams, other than their large batch sizes.

4 Operator implementations

Many of K-Pg's operators act on shared indexed batches of input updates, and this structure and potential volume of data can lead to very different operator implementations from record-at-a-time streaming systems. In this section we explain K-Pg's operator implementations, starting with the simplest examples and proceeding to the more complex `join`, `group`, and `iterate` operators.

4.1 Key-preserving operators

Several stateless operators are "key-preserving", in that 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 can be implemented either as streaming operators for streams of update triples, or as wrappers around arrangements. For example, the `filter` operator only needs to restrict the data presented in batch and trace navigation, based on whatever predicate is supplied to the filter operator.

These implementations contain trade-offs. An aggressive filter may reduce the volume of data to the point that it is relatively cheap to maintain a separate index, and relatively ineffective to search in a large index only to discard the majority of results. A user can filter an arrangement, or first reduce the arrangement to a stream of updates and then filter it.

4.2 Key-altering 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 reduce any arranged representations to streams of update triples.

4.3 Stateful operators

Differential dataflow's stateful operators are data-parallel, meaning that 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 K-Pg and similar systems to distribute operator work across otherwise independent workers, who can then process their work without further coordination. At a smaller 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 a key before moving to the next key, even if this violates the timestamp order.

History replay. Our operators rely, for reasons of the operator work proportionality principle, on a common mechanism that for each *key* takes historical tuples *(val, time, diff)* and allows the operator to advance their accumulation (under the partial order) forward through a compatible total order on times. As we move forward through times we maintain the set of distinguishable updates, using the same logic as a trace does for consolidation, but in a private per-key copy that we can mutate and eventually discard. By consolidating as we go, each step forward takes time proportional to the number of distinguishable updates. Without this layer it is easy for operators to unintentionally "go quadratic" where each update re-scans the full history to re-form its accumulated view, which causes large physical batching to go into a tailspin.

4.3.1 The `join` operator

Our `join` operator takes as inputs batches of updates from each of its arranged inputs. Its job is to produce any changes in outputs that result from its advancing inputs. Our implementation has several variations from a traditional streaming hash-join.

Trace capabilities. When join retires a batch of input changes, the resulting outputs all have timestamps in advance of those in the input batch. This means that times in the opposing trace do not need to be distinguished at times not an advance of the first input frontier. The `join` operator downgrades each of its trace handle capabilities to be the input frontier of the opposing input, and drops a handle when the opposing input frontier becomes empty.

Alternating seeks. Join can receive input batches of substantial size, especially from a maintained arranged collection. To respect operator work proportionality, we perform alternating seeks in the input cursors: when the cursor keys match we perform work, and 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.

Amortized work. Especially in graph computation, the `join` operator may be called upon to produce a significant amount of output data that can be reduced only once is crosses an exchange edge for a downstream operator. If each input batch is immediately processed to completion workers may be overwhelmed with the amount of output data, either buffered for transmission or (as in K-Pg) transmitted to the destination workers but then buffered at each awaiting reduction. Instead, K-Pg responds to new input batches by producing futures, 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 require, and so do not block state maintenance for other operators.

4.3.2 The `group` operator

The `group` operator takes as input a 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. For each key, we must reconsider each time that is the least upper bound of a set of times in updates for that key, which can include times at which no updates occurred. Consequently, the `group` operator tracks a list of *interesting* pairs *(key, time)* of future work that is required even if we see no input updates for the key at that time. To support this work, the operator also informs timely dataflow that it retains the ability to send outputs with timestamps in advance of the lower frontier of interesting times.

For each interesting *(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. The `group` operator relies on the history replay mechanism to amortize the cost of re-

processing a key across the multiple times it may need to reconsider.

Output arrangements. The `group` operator uses a collection trace for its output, to efficiently reconstruct what it has previously produced as output without extensive re-invocation of the supplied user logic (and to evade potential non-determinism therein). This provides the `group` operator the opportunity to share its output, just as the `arrange` operator. It is not uncommon, especially in graph processing, for the results of a `group` to be immediately joined with edges to disemminate the information, and `join` can re-use the same indexed representation that `group` uses internally for its output.

Specializations. The `group` operator is much simpler for totally ordered times, and becomes simpler still when it only needs to implement `count` or `distinct`. We provide several such specialized operators, with type-level restrictions to guarantee they are not mis-used. Users select the operator that best suits their purpose, as long as the types satisfy the imposed constraints.

4.4 Iteration

The iteration operator is largely unchanged from Naiad's differential dataflow implementation. It creates a new subgraph with an extended timestamp type, containing an additional integer for "round of iteration" and which is partially ordered using the product partial order (two products are ordered if both of their coordinates are equivalently ordered). The initial collection, the method's argument, is introduced as a stream of changes at iteration zero, with the body of the iterative computation attached. At the tail of the body, the result is merged with the negation of the initial input collection, and all changes are returned around the loop to the head with the iteration index incremented.

We have made two minor modifications. First, arrangements external to the iteration can be introduced (as can un-arranged collections) with the `enter` operator, whose implementation for arrangements only wraps cursors with logic that introduces a zero coordinate to the timestamp; indices and batches remain shared. Second, we introduced a `Variable` type for recursively defined collections which allows for programmatic construction of mutual recursion, as well as the ability to return intermediate collections other than the result of the loop body. This second feature is important when we want to share collections around a loop iteration, as it allows us to rotate the loop body so that the sharing is within one iteration while still returning the intended result.

5 Evaluation

We now shift to an evaluation of K-Pg, and its implementation of differential dataflow. To compare with prior work we are interested in evaluating scalability, responsiveness, and generality, but we also want to assess the benefits of state sharing and high throughput updates.

We run our experiments on 4-socket NUMA systems equipped with 4 *Intel Xeon E5-4650 v2* cpus each with 10 physical cores and 512GB of aggregate system memory. K-Pg distributes across multiple machines, but our evaluation here is restricted to multiprocessors, which have been sufficient to reproduce computation that require more resources for less expressive frameworks.

We start with microbenchmarks which assess the performance of individual components of our system, primarily the arrangement infrastructure. Our goal is to understand the robustness of the core state management operator. We are specifically interested in the latency profiles at various throughputs, as we vary the characteristics of the data, the offered load, and the number of workers.

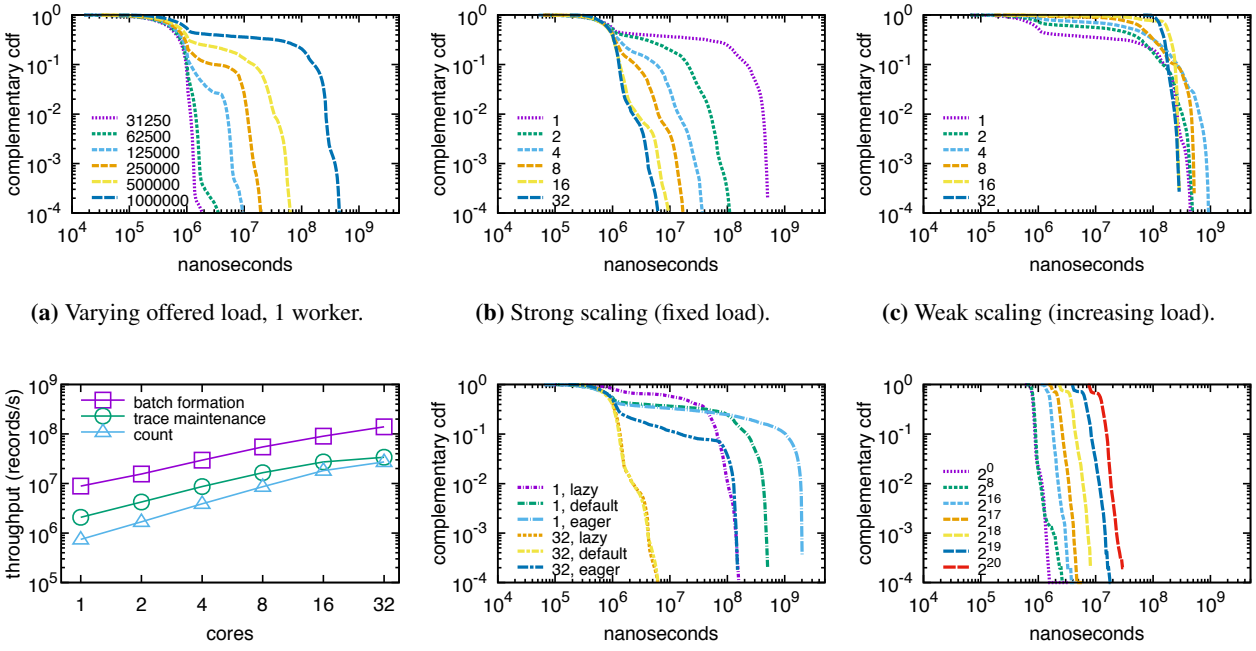
We then move into several analytics tasks, including relational queries, graph computation, and Datalog evaluation. We observe the benefits of decoupling logical updates from physical batches for high throughput relational and graph computations, where we exceed update rates of prior systems. We observe the benefits of state sharing for graph computation and Datalog evaluation, where more opportunities for re-use exist and for which we see substantial reduction in memory footprint and latencies, and an increase in throughput.

Due to the breadth of the evaluation, we have largely restated recent reported measurements from other systems on comparable hardware, rather than attempt to reproduce their results on our own hardware. Our goal is not to demonstrate that K-Pg outperforms any of these systems, only that its performance is comparable, while supporting more general computation than the other systems. Many of the other systems, notably the databases, provide significant additional functionality that K-Pg does not provide.

5.1 Microbenchmarks

We perform several microbenchmarks assessing the `arrange` operator applied to a continually changing collection of 64bit identifiers (with 64bit timestamp and signed difference). We are most interested in the distribution of response latencies as configurations change, and we report all latencies in complementary cdf form ("fraction of times with latency greater than") to get high detail in the tail of the distribution.

Varying load. Figure 3a 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. K-Pg allows the test harness to trade latency for throughput until equilibrium is reached.



(a) Varying offered load, 1 worker.

(b) Strong scaling (fixed load).

(c) Weak scaling (increasing load).

(d) Throughput by task.

(e) Amortized merging levels.

(f) Join dataflow execution.

Figure 3: Microbenchmarks for arrangement and join execution.

Strong scaling. Figure 3b reports the latency distributions for varying numbers of workers under a fixed workload of 10M keys and 1M updates per second. As the number of workers increases the latency-throughput trade-off swings in favor of latency.

Weak scaling. Figure 3c reports the latency distributions for varying numbers of workers as we proportionately increase the offered load. While the shape of the latency distribution changes, drawing in the tail at the expense of median latency, the latencies generally remains stable.

Throughput. Figure 3d breaks down the peak throughputs of sub-components of arrangement: batch formation, trace maintenance, and a `count` operator maintained for the results. The trace maintenance scaling is interrupted at 32 workers by an unpleasant interaction with the Linux memory subsystem that can likely be worked around.

Amortized merging. Figure 3e reports the latency distributions for one and 32 workers each with three different settings for merge amortization: eager, default, and lazy. We see that for a single worker the lazier settings have smaller tail latencies, but are more often in the tail. For 32 workers, the lazier settings are significantly better as workers are less likely to stall and block the entire computation. As with garbage collection in Broom [10], we conclude that large but rare dataflow interruptions are nonetheless significant impediments to strong scaling.

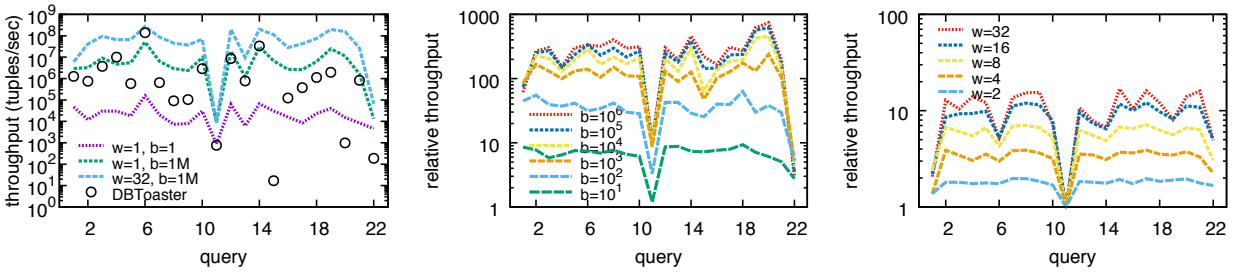
Join proportionality. Figure 3f reports the distributions of latencies to install, execute, and complete new dataflows joining small collections of varying size against a pre-arranged collection of 10M keys. K-Pg has nominal overheads for installing new dataflows, as low as milliseconds, and executes joins in time proportionate to the size of the small collection.

5.2 Analytics workloads

TPC-H is a traditional data analytics benchmark: twenty-two relational queries of varying complexity over relations that describe parts, orders, suppliers, and their inter-relationships. Nikolic et al. [20] study the problem of maintaining the TPC-H queries as they incrementally load the source data, varying logical batching but without decoupling it from physical batching. We implement the same queries in K-Pg, but planned manually.

Absolute performance. Figure 4a reports absolute throughputs on the twenty-two queries for a scale factor 10 input for DBToaster and K-Pg in three configurations: single worker with batch size one, single worker with batch size 1M, and 32 worker with batch size 1M. Physical batching allows K-Pg to increase throughput and distribute work without altering the computation.

Physical batching. Figure 4b reports the relative throughput increases for one worker as we increase the physical batching from one up to 1,000,000. The increases are substantial at first, and continue but diminish



(a) DBToaster and K-Pg. **(b)** Increases with batching. **(c)** Increases with workers.

Figure 4: Absolute and relative query rates for the 22 TPC-H queries.

with larger batch sizes.

Worker scaling. Figure 4c reports the relative throughput increases as we increase the workers from one to sixteen, holding physical batching at 1M. Many queries increase their throughput proportionally, and correspondingly reduce their latencies, though several which involve global aggregation do not. The temporal nature of the queries, producing histories for each aggregate, means these computations resist traditional techniques like pre-aggregation, and instead require techniques like parallel-prefix aggregation to parallelize effectively.

TPC-H contains several opportunities for optimization. Query 15 contains an `argmax`, which we implemented hierarchically with a sequence of `group` operators using progressively more coarse keys; this transformation was manual, but gains five orders of magnitude throughput over full re-evaluation as in Nikolic et al. Queries Q11 and Q22 would benefit from an inequality join operator, one that responds to changes in a threshold by extracting the subset of values between the changes.

Several queries could be made more interactive. For example, Query 21 asks for suppliers that kept orders waiting, restricted to suppliers in Saudi Arabia (the nation is a "substitution parameter" and could have other values). This restriction is hard-wired into the query in most systems, but our implementation maintains results for all countries and joins the results with a query nation at the end; it would be natural to have this be an input and to allow users to ask about other countries interactively.

We could arrange each of the eight base relations and maintain them as they change. This would provide scalable incremental view maintenance over queries that share these underlying indices. Unfortunately most of the TPC-H queries first aggressively filter their inputs, and do not provide opportunities to demonstrate these benefits. A more sophisticated benchmark with broader uses would potentially better highlight the potential of sharing.

5.3 Graph workloads

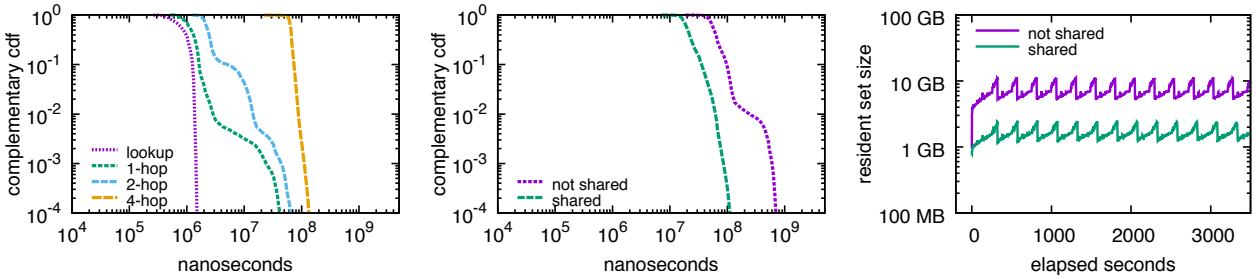
We next evaluate K-Pg on graph workloads, ranging from large computations on static graphs, to interactive queries against evolving graphs.

Batch computation. We evaluated K-Pg on standard computations of reachability, breadth-first distance labeling, and undirected connectivity, on a standard Twitter social network. We compare against single-threaded code, and several general systems applicable to graph processing. There are any number of specialized graph processors, and one should imagine them as perfectly scaling versions of the single-threaded code. Importantly, neither K-Pg nor the general systems require any graph preprocessing, whereas the single-threaded and specialized graph processors do require varying amounts.

Our measurements indicate that K-Pg is consistently faster than systems like BigDatalog, Myria, SociaLite, and GraphX, and substanially slower (roughly 30x) than purpose-written single-threaded code against preprocessed graph data. We did find that when we modified the purpose-written code to use hash maps for node state instead of arrays, to accommodate non-preprocessed node identifiers, K-Pg was immediately competitive at twice the core count. We implemented an optimized single-threaded computation to maintain BFS labels over 1 million graph updates, and K-Pg matched the performance using between two and four cores (depending on the graph properties).

Interactive queries. Pacaci et al [21] evaluate multiple databases (graph and relational) on four interactive graph queries: point look-ups, 1-hop look-ups, 2-hop look-ups, and 4-hop shortest path queries (shortest paths of length at most four). We implement each of these queries in differential dataflow, each query as a dataflow where the query arguments are independent collections that may be modified to introduce or remove specific query instances.

Figure 5 reports latency distributions for the four query classes on an evolving graph of 10 million nodes and 64 million edges, under a load of 200,000 changes



(a) Latencies for homogenous queries. **(b)** Latencies for query mix. **(c)** Resident set size.

Figure 5: Graph query benchmarks. 32 workers, 10M nodes, 32M edges, 200K updates / second.

per second, half graph modifications and half query modifications. The reported latencies are comparable to those reported in [21] for single queries against a static graph (of the same size, but generated differently), except for the shortest path queries that load the workers more. The throughput greatly exceeds that observed in [21], primarily because we are able to compile the repeated query structure to dataflow, and the logical timestamps pre-resolve read and write conflicts.

Figure 5b reports the latency distributions for a mix of the four queries, where the 100,000 queries per second are evenly distributed between the four query types, both with and without sharing the graph structure. There is a noticeable latency penalty due to the increased system load, which would only increase as more query classes are maintained. While all databases do share such state, they do not compile query classes to streaming dataflow. While streaming dataflow systems like Flink and Naiad could implement these queries, they would not benefit from the shared state.

Figure 5c reports the memory footprint for the query mix with and without sharing, for an hour-long execution. We can see that the memory footprint is stable, varying between 1GB and 2GB corresponding to 32 bytes per graph edge update. The variability is due to the geometrically sized merges, which maintains varying numbers of batch layers, containing at most double the number of edges. The memory footprint for the unshared implementations is also stable, but roughly five times higher (as there are five uses of the graph across the four queries).

5.4 Datalog workloads

Datalog is a relational language in which the query results are the fixed point of repeated application of recursively defined rules. Datalog queries are more general than graph computation, and in particular 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, DeALS, and several distributed systems have recently emerged, including Myria, SociaLite, and BigDatalog. At the time of writing only LogicBlox supports decremental updates to Datalog queries, using a technique called "transaction repair" [24].

Bottom-up (batch) evaluation. In the literature, Datalog systems are often evaluated on two simple queries, "transitive closure" (*tc*) and "same generation" (*sg*), which exercise many Datalog engines by the volume of tuples they can produce. We compared K-Pg with the reported numbers of the shared memory and distributed systems above on six reference problems, taken from [22], and found that K-Pg was generally comparable. For four out of six reference problems K-Pg out-performed all of the distributed systems, and was comparable to the best shared memory systems (DeALS) on one core (three out of five problems) but scaled less well with increasing cores (one out of five problems on 64 cores).

Many of the query-level optimizations these systems employ could be translated to K-Pg, but the scaling of the shared memory systems relies in part on racy code permitted by the monotonic semantics of Datalog. K-Pg prevents racy access and imposes synchronization between rounds of derivation, which can limit performance but also allows it to efficiently update such computations when base facts are added or removed.

K-Pg's generality also has some benefits; one transitive closure computation takes 312 seconds single-threaded on DeALS, but K-Pg can determine the strongly connected component structure, summarizing the transitive closure, in just 274ms.

Top-down evaluation. If the user imposes constraints on the target query, for example $tc(\text{"}david\text{"},x)$, one can explore the space of facts from the possible goals back to the facts of the base relations. The "magic set" transformation [5] rewrites such queries as bottom-up computations with a new base relation that "seeds" the bottom-up derivation with query arguments, and the rules are rewritten so that facts are not derived without the participation of some seed record. For example, the magic set trans-

System	cores	*linux*	*psql*	*httpd*
SociaLite	4	OOM	OOM	4 hrs
Graspan	4	713.8 min	143.8 min	11.3 min
K-Pg	1	76.8s	37.0s	10.9s
K-Pg (med)	1	1.11ms	185ms	22.0ms
K-Pg (max)	1	8.13ms	1.48s	218ms

Table 1: System performance for the three graphs for the *dataflow* analysis. Times for the first two systems are reproduced from [25], and exclude pre-processing. Also, the median and maximum times to remove each of the first 1,000 null assignments from the complete analysis.

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

Table 2: System performance for the three graphs for the *points-to* analysis. Times for the first two systems are reproduced from [25], and exclude pre-processing. Further K-Pg implementations correspond to an optimized query (Opt), and the optimized query without sharing (NoS).

formation of the *tc* query produces either a forward or backward reachability query, seeded with the specified value. In K-Pg (and some interactive Datalog environments) this work can be performed against maintained indices of the non-seed relations, in much less time than it would take batch processors to re-index these relations.

We evaluated the six reference problems transformed to top-down queries, and found largely unsurprising results. When the result sets were small the latencies were interactive, when they were large the queries took proportionately longer. Two reference problems, "same generation" on grids and random graphs, re-derive most of their output as part of the transformed query, and were not interactive. This appears to be more a limitation of the query transformation rather than K-Pg. Because the queries are interactively evaluated, we were able to perform expensive queries like "same generation" on graphs whose scale would prevent complete bottom-up evaluation; however, at this point the batch processors applied to the transformed query might also be reasonable solutions.

5.5 Program Analysis

Graspan [25] is a system built to perform static analysis of large code bases, created in part because existing systems could not handle the non-trivial analyses at the sizes required. Graspan out-performs SociaLite by orders of magnitude when the latter successfully completes, which it often does not.

Tables 1 and Table 2 reproduce the running times reported in [25], and reports those of K-Pg for their *dataflow* and *points-to* analyses, respectively. The *dataflow* query is a relatively simple reachability query, where null assignments are propagated along program assignment edges. The more complicated *points-to* analysis develops a mutually recursive graph of value flows, and memory and value aliasing. In both cases we see a substantial improvement, due in some part to our ability to re-use operators from an optimized system rather than implement and optimize an entirely new system.

Disk-based access. Graspan is designed to operate out-of-core, and explicitly manages its data on disk. We report K-Pg measurements on a laptop with only 16GB of RAM; only the *points-to* analysis exceeds this limit (peaking around 30GB), but its sequential access makes the operating system's paging mechanisms sufficient for out-of-core execution. We verify this by modifying the computation to only use 32bit timestamps and differences, which brings the memory footprint to within RAM limits; this optimized version runs only 20% faster.

Optimization. The *dataflow* analysis is a simple reachability computation, which performs well and has little room for optimization. The *points-to* analysis is a more complicated mutually recursive computation; in the formulation of [25] it is dominated by the determination of a large relation (value aliasing) that is used only once. However, this relation can be optimized out (value aliasing is eventually restricted by dereferences, and this restriction can be performed before forming all value aliases). The result is a much more efficient computation, and one that re-uses each relation multiple times. Table 2 reports the optimized running times, with and without sharing, where we can see the positive effect of sharing, and the limiting effect of failing to share state.

Top-down evaluation. Both *dataflow* and *points-to* can be transformed to support interactive queries instead of batch computation. Figure 1 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 can be interactive.

6 Related work

Many of K-Pg's features have appeared before in isolation, but they have not been brought together in one system. This section characterizes prior data processors, highlights their limitations, and points out assumptions made by them that K-Pg leaves behind.

Batch processors. Systems like MapReduce [9], DryadLINQ [26], Spark [27] execute large computations as directed acyclic graphs (DAGs) of independent, finite-duration tasks. The independence of tasks allows these systems to scale, and task re-execution allows them to tolerate failures [19, 27]. However, the independence of these tasks trade efficiency for resilience.

DryadLINQ and Spark have re-usable datasets (respectively, Nectar [12] and Resilient Distributed Datasets), but this re-use avoids only the re-computation of the dataset from dataflow inputs. The dataset must still be re-scanned and re-indexed for each use, whereas in K-Pg the data are maintained in the indexed form that many operators require, supporting the effectively free deployment of new instances of such operators.

Stream processors. Systems like Borealis [4], STREAM [3], and TelegraphCQ [7] maintain continuous queries over streams of data. STREAM in particular maintains "synopses" (often indices) for operators and shares them between operators. K-Pg's shared indices can be seen as distributed, multi-temporal versions of STREAM's synopses. Unlike STREAM, K-Pg reveals the synopsis structure (a log of indexed batches) to its operators, which take advantage of this representation.

Modern stream processors like Flink [6] and MillWheel [2] impose less structure, and more closely resemble continually executing MapReduce dataflows. This flexibility enables complex, non-relational event processing, but their architectures evolved without the goal of supporting shared data. We believe this is a design flaw, and that modern stream processors operate far below capacity because of this decision.

Finally, stream processors lack support for iteration. Unlike batch processors, which can extend their dataflow DAG arbitrarily, stream processors do not continually re-structure their dataflows. One exception, Naiad [18], supports iterative computation through cyclic dataflow graphs and partially-ordered timestamps. In a sense, Naiad shares state across re-invocations of the same operator, but not otherwise.

Databases. Relational (and other) databases provide general functionality for managing data, but have not to date been efficient, scalable solutions for general computation. Databases do however share indices between queries, and K-Pg explicitly draws inspiration from their economy of execution on their target functionality. Databases perform a great many tasks beyond computation, and one should view K-Pg as general, scalable, and responsive *incremental view maintenance*, applicable to the change log of a durable and consistent store.

Our shared state representation is inspired by the design of Log Structured Merge (LSM) trees. One can interpret K-Pg as a dataflow system where state is commonly maintained in a multi-temporal LSM structure, and whose dataflow edges transport LSM layers. There is a healthy amount of recent work on LSM design, and our experience has been that a write-optimized design has similar positive implications for high-throughput computation as it has for storage.

K-Pg is most analogous to a temporal database, in which records have timelines of changes they undergo, and taken together describe the history of an evolving collection of records. Queries describe transformations of these collections, without windowing requirements, and themselves produce as output an evolving collection of results. Our system resembles a streaming temporal database, with the expressivity of its query language raised to that of differential dataflow.

Specialized systems. Relatively few graph processors attack the problem of computations over continually evolving graphs. Chronos [13] is a temporal graph engine, in that it supports queries over a fixed sequence of graphs, rather than a responsive system against live data; it targets coarsely batched snapshots and is evaluated on monotonic graph computations. GraphTau [15] targets continually evolving graph data (and queries against historical data) and uses persistent data structures to share the representation of multiple snapshots. Sharing graph data across computations is not discussed, though their data representation should make this possible.

Datalog has re-emerged as an expressive analytics language, capturing some graph computations among other more general recursive queries. Datalog is by its nature restricted to monotonic queries (once true, facts remain true), but variants support non-monotonic queries using "stratification", which serializes the execution of parts of the query. Single-machine Datalog systems support interactive queries, but the distributed batch processing models effectively preclude an interactive experience. Datalog has traditionally resisted efficient updates in the presence of retractions, with exceptions being differential dataflow and LogicBlox's "transaction repair" [24].

7 Conclusions

We presented K-Pg, a general, scalable, and responsive data processor. K-Pg's design required re-thinking the traditional dataflow architecture, specifically to introduce shared state that supports a high rate of logical updates. This design shift enables new opportunities for operators, which can more efficiently retire batches of indexed updates than tuple-at-a-time processors. Together, these changes enable the efficient implementation of new classes of computation, in which random access to existing indexed state comes at relatively low cost. We evaluated K-Pg on a variety of computations, and find that its performance is comparable with and occasionally better than existing systems on a variety of tasks.

References

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

[2] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, et al. "MillWheel: Fault-tolerant Stream Processing at Internet Scale". In: *Proceedings of the VLDB Endowment* 6.11 (Aug. 2013), pages 1033–1044.

[3] Arvind Arasu, Brian Babcock, Shivnath Babu, et al. "STREAM: The Stanford Data Stream Management System". In: *Data Stream Management: Processing High-Speed Data Streams*. Edited by Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. Berlin/Heidelberg, Germany: Springer, 2016, pages 317–336.

[4] Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Michael Stonebraker. "Fault-tolerance in the Borealis Distributed Stream Processing System". In: *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*. SIGMOD '05. Baltimore, Maryland: ACM, 2005, pages 13–24.

[5] 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. Cambridge, Massachusetts, USA: ACM, 1986, pages 1–15.

[6] Paris Carbone, Stephan Ewen, Seif Haridi, Asterios Katsifodimos, Volker Markl, and Kostas Tzoumas. "Apache Flink: Stream and batch processing in a single engine". In: *IEEE Data Engineering* 38.4 (Dec. 2015).

[7] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, et al. "TelegraphCQ: Continuous Dataflow Processing". In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. San Diego, California, USA, 2003, pages 668–668.

[8] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. "PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs". In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys '15. Bordeaux, France: ACM, 2015, 1:1–1:15.

[9] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Communications of the ACM* 51.1 (Jan. 2008), pages 107–113.

[10] Ionel Gog, Jana Giceva, Malte Schwarzkopf, et al. "Broom: Sweeping out Garbage Collection from Big Data Systems". In: *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*. HOTOS'15. Switzerland: USENIX Association, 2015, pages 2–2.

[11] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs". In: *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Hollywood, California, USA, Oct. 2012, pages 17–30.

[12] 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)*. Vancouver, British Columbia, Canada, 2010, pages 75–88.

[13] Wentao Han, Youshan Miao, Kaiwei Li, et al. "Chronos: A Graph Engine for Temporal Graph Analysis". In: *Proceedings of the Ninth European Conference on Computer Systems*. EuroSys '14. Amsterdam, The Netherlands: ACM, 2014, 1:1–1:14.

[14] 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)*. Lisbon, Portugal, Mar. 2007, pages 59–72.

[15] Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, and Ion Stoica. "Time-evolving Graph Processing at Scale". In: *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*. GRADES '16. Redwood Shores, California: ACM, 2016, 5:1–5:6.

[16] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, et al. "Twitter Heron: Stream Processing at Scale". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. Melbourne, Victoria, Australia, 2015, pages 239–250.

[17] 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)*. Asilomar, California, USA, Jan. 2013.

[18] 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)*. Nemacolin Woodlands, Pennsylvania, USA, Nov. 2013, pages 439–455.

[19] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. "CIEL: A Universal Execution Engine for Distributed Data-flow Computing". In: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. NSDI'11. Boston, MA: USENIX Association, 2011, pages 113–126.

[20] 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)*. San Francisco, California, USA, 2016, pages 511–526.

[21] 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. Chicago, IL, USA: ACM, 2017, 12:1–12:7.

[22] 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)*. San Francisco, California, USA, 2016, pages 1135–1149.

[23] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, et al. "Storm @Twitter". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. Snowbird, Utah, USA, 2014, pages 147–156.

[24] Todd L. Veldhuizen. "Transaction Repair: Full Serializability Without Locks". In: *CoRR* abs/1403.5645 (2014). arXiv: 1403.5645.

[25] 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. Xi'an, China: ACM, 2017, pages 389–404.

[26] 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)*. San Diego, California, USA, Dec. 2008.

[27] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, et al. "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)*. San Jose, California, USA, Apr. 2012, pages 15–28.

[28] Yunhao Zhang, Rong Chen, and Haibo Chen. "Sub-millisecond Stateful Stream Querying over Fast-evolving Linked Data". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP '17. Shanghai, China: ACM, 2017, pages 614–630.