K-Pg: Shared State in Differential Dataflows

Anonymous Author(s)

Abstract

Many of the most popular scalable data-processing frameworks are fundamentally limited in the generality of computations they can express and efficiently execute. In particular, we observe that systems' abstractions limit their ability to *share* and *reuse* indexed state within and across computations. These limitations result in an inability to express and efficiently implement *algorithms* in domains where the scales of data call for them most.

In this paper, we present the design and implementation of K-Pg, a data-processing framework that provides high-throughput, low-latency incremental view maintenance for a general class of iterative data-parallel computations. This class includes SQL, stratified Datalog with negation and non-monotonic aggregates, and much of graph processing. Our evaluation indicates that K-Pg's performance is either comparable to, or exceeds, that of specialized systems in 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 [10, 16, 29] support general computations, but are expensive for random access to mutable state. Stream processors [7, 18, 25] support fine-grained changes at high rates, but lack support for iterative subcomputations. Graph processors [9, 13, 30] 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 [24] and program analysis [27], because existing systems are reportedly insufficient for increasingly sophisticated computations.

These systems differ mainly in how they express and exploit the potential for sharing and re-use of both indexed data and computation (Table 1). Relational databases **holistically** share maintained indices and views across potentially unrelated computations, and apply fine-grained updates to them as the underlying data change. Stream processors share maintained state **temporally** across a sequence of input changes to a fixed computation. Graph processors share vertex or edge state **iteratively** across repeated rounds of the computation, as they propagate changes or explore the graph.

Sharing	holistic	temporal	iterative
Spark	no	no	no
RDBMS	yes	no	some
Flink	no	yes	some
Naiad	no	some	yes
K-Pg (this work)	yes	yes	yes

Table 1: Representative data processing frameworks.

Each type of sharing avoids a potentially unneccessary *cost* for a computation, in the form of repeated work, and when a system lacks a sharing type it becomes inefficient and ill-suited for a class of computations. Worse, some computations require multiple forms of sharing—consider, for example, streaming graph processors or multi-user Datalog environments. A system that supports all three sharing types could efficiently support existing workloads, integrate specialized systems, and remove the barriers for domains which no system currently supports.

This paper develops a holistic, temporal, and iterative data-parallel dataflow system, K-Pg, capable of efficiently computing and incrementally maintaining a broad class of iterative data-parallel computations. K-Pg outperforms specialized systems in high-throughput incremental view maintenance, interactive graph queries against dynamic graph data, and goal-driven Datalog evaluation. K-Pg remains viable for traditional batch analytics, graph processing, and Datalog evaluation, though it does not out-perform the top specialized systems in these domains. Finally, we have found K-Pg to be excellent for prototyping in new domains; a recent system Graspan [27] for program analysis requires an order of magnitude less code when re-implemented on K-Pg, and doing so improves its performance by factors ranging from 14x to 550x on their workloads.

1.1 System overview

K-Pg is based on differential dataflow [19], a data-parallel processing paradigm in which users express computations by high-level operations on collections of records and then interactively update the inputs to their computations, and receive the corresponding changes to the outputs of their computations. 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. Differential dataflow's 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 (now dormant) prototype implementation, as part of the Naiad [20] project, several aspects of this implementation limit its performance for general computations. Specifically, while the prototype provides iterative sharing, its operator design did not reveal opportunities for holistic sharing and details of Naiad's system interface prevented high-throughput temporal sharing, and so the prototype implementation evolved without these constraints. Our work starts from our attempts to use this prototype in new problem domains, and details the redesigns we found necessary to introduce holistic and temporal sharing.

K-Pg is a complete re-implementation of the differential dataflow model, based on a fundamentally redesigned dataflow processor. Whereas big data processors are traditionally "shared nothing", and create and manage dataflows of independent operators, K-Pg exposes and *shares* indexed representations of streamed operator inputs, among operators and even across dataflows. Importantly, while this sharing may happen across operators and dataflows it does not cross worker (thread) boundaries, which allows us to retain the scalability of the shared-nothing worker design.

A conventional shared-nothing dataflow processor requires several changes to realize the performance benefits of shared indexed state. We decompose traditional operators like `join`, `group`, and `distinct` into the combination of a new general operator `arrange` and thinner task-specific shell operators. The `arrange` operator produces worker-sharded indexed batches of data, and maintains worker-local multiversioned indices aligned to these batches. It does this with high throughput, low latency, and a compact representation, all with minimal worker coordination. Operators are modified to share access to this local index, and become simpler and more performant as a result, but must take care when interpreting the contents of their shared indices.

We implemented K-Pg in roughly 6,000 lines of Rust [1], atop an unmodified timely dataflow runtime [2]. 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 specialized 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 offers lower latency and higher throughput than other bespoke approaches. Despite the variety of tasks and requirements, these are each implemented as idioms in one sufficiently expressive and performant system.

1.2 Limitations and Tradeoffs

K-Pg provides sharing opportunities to computations that are structured to benefit from them, but the support for sharing does come with some costs. Holistic sharing moves us to a multiversion index design, as operators may make progress at different rates, which can be more expensive than a single-user index. Temporal and iterative sharing leads to operator implementations that consider the histories of records, even for computation that would completely overwrite values.

K-Pg is intentionally implemented as an optional layer atop a timely dataflow platform, so that users can pick and choose dataflow operators that they may benefit from, without being prevented from selecting or implementing more efficient dataflow operators when appropriate.

1.3 Contributions

This paper summarizes the design and implementation of K-Pg, with the specific intended contributions of:

(1) A data-parallel dataflow design with indexed operator inputs shared across multiple dataflows (§ 3).

(2) A multi-versioned shared index design with high read and write throughput, low read and write latency, and compact memory footprint (§ 4).

(3) New operator implementations which take advantage of input streams of shared indexed batches as opposed to input streams of tuples or records (§ 5),

(4) An evaluation of K-Pg that indicates it can be as capable as many 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 (§ 6).

We discuss the most relevant related work, on which K-Pg draws, in Section 7 and conclude in Section 8 with a summary and thoughts for future research directions.

2 Motivation

Differential dataflow [19] is a framework in which users first write a query using relational and data-parallel operators, including fixed-point iteration, and then interactively update the inputs to their query and receive the corresponding updates to the outputs of the query. If all inputs are changed once, from their initial empty state, differential dataflow appears as a batch processor. If the inputs continue to be changed, differential dataflow appears as incremental view maintenance for the expressed computation. Differential dataflow's incremental semantics allow a system to perform an amount of work proportional only to the volume of changes across internal collections. This volume can often be surprisingly small, even for sophisticated recursive queries, which enables both low latency and high throughput updates.
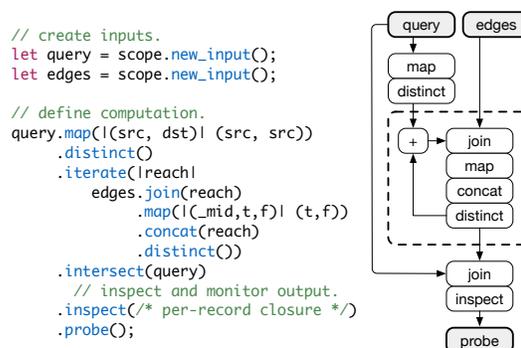
A fully featured implementation of differential dataflow enables computational idioms beyond the reach of traditional batch and stream processing. With iteration one can faithfully express a class of *algorithms* outside the reach of traditional relational and big data frameworks. Through interactive manipulation of input collections we can pose queries and direct the execution of such computations. The re-use of shared indices provide nearly zero overhead when new computations require random access to pre-existing collections. In many ways differential dataflow more closely resembles a declarative *programming* language than a traditional query language, and it has the potential to substantially enlarge the volume of computer science applicable to large-scale data processing.

2.1 A motivating example

Consider the problem of determining for a directed graph and a collection of query pairs (src, dst), for which of the queries can one reach from src to dst along directed graph edges. Figure 1 contains a stylized differential dataflow program to accomplish this, but its behavior showcases many subtleties which we now investigate.

The example in Figure 1 starts by transforming the query pairs into reachability statements $(to, from)$, initially the sources of the queries. This set is then iteratively developed, by repeatedly joining it with the set of edges to create new reachability statements, folding in old statements, and maintaining the set of distinct results. This iterative process produces all nodes reachable from each source, and we intersect this with the query set to return only those pairs posed as queries.

Importantly, Figure 1 is only a declarative description of the computation. It provides a differential dataflow system with requirements, but it also provides flexibility in how the computation should be effected.


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

// define computation.
query.map(|(src, dst)| (src, src))
    .distinct()
    .iterate(|reach|
        edges.join(reach)
            .map(|(_mid,t,f)| (t,f))
            .concat(reach)
            .distinct())
    .intersect(query)
    // inspect and monitor output.
    .inspect(/* per-record closure */)
    .probe();
```

Figure 1: A graph reachability example, where inputs `query` and `edges` each contain (src, dst) **pairs, and whose outputs are those pairs in `query` that can be reached along paths in `edges`. The two inputs can be interactively updated, resulting in interactive queries that are incrementally maintained as the graph changes.**

Interaction Both the `query` and `edges` input collections can be interactively updated, which yields the output changes required to correctly update the output. When we add or remove an edge, the output produces additions for newly reachable query pairs, or subtractions for query pairs that can no longer reach from one to the other. Alternately, when we add or remove a query, we may initiate a new reachability computation if the source is previously unseen; if the source is already present in the query set the change will be suppressed at the `distinct` operator, as the distinct set of sources has not changed; the result will instead just be read out of the `intersect` operator.

As far as we know, differential dataflow alone supports this type of interactive computation. Conventional graph processors do not support low-latency interactive updates to computations, and the graph processors that do (only [17], to our knowledge) do not blend interactive queries in the same model.

Economy The use of `edges` in the `join` operation requires the collection of edges indexed by src, in order to quickly read out matching dst values. This *arrangement* of the edges collection is not uncommon in graph processing, and if the collection is already available in this form the time to install the computation and begin servicing changes to `query` is approximately zero (milliseconds, in our experiments). Of course, work must take place as new queries arrive and begin to explore the graph, but this work is proportional only to the work required (nodes explored) rather than the size of the perhaps substantially larger `edges` collection.

Most non-RBDMS systems do not share indexed data and would require fresh copies of graph data, and often

even require a fresh set of computers for each computation. Systems like Spark, Flink, and Naiad, for example, would need to re-ingest, index, and maintain the whole graph for each use, limiting the number of concurrent computations they can maintain.

Throughput As the query is specified declaratively, rather than as the result of a sequence of imperative tasks, we have the ability to concurrently process updates at many distinct times. This substantially improves the effective system throughput, without compromising its fidelity to the source timestamps. We can service on the order of millions of distinct changes to the `query` and `edges` collections per second, producing at all times consistent and correct outputs, even with the query and graph changes interleaved.

Other than stream processors, typical techniques for incremental computation (e.g. Incremental View Maintenance) move serially through a sequence of updates. Naiad pipelines distinct logical times, but still introduces coordination traffic for each and can only scale its throughput by coarsening logical times, and removing the distinctions between changes in the input.

Our declaratively programmed reachability "query" is now an interactive computation that can both answer reachability queries between pairs of nodes and install pairs of nodes to monitor as the underlying graph continually changes. If the graph is already arranged, the additional memory overhead is proportional only to the number of nodes reachable from the query sources currently maintained. As edges and queries change, we maintain a high-throughput view of the correct results without serializing the system execution. These benefits largely derive from the declarative description of our interactive computation, rather than as an imperative sequence of queries and transactions against common relations.

With light modification, we could additionally track the distances between the query nodes, or recover the paths themselves. With further effort one can implement even smarter algorithms for shortest path queries, namely bi-directional search, which brings query times for even new sources down to milliseconds and increases throughput substantially (as the computation "changes less").

3 System design and background

K-Pg is built on an existing timely dataflow execution layer [2], and inherits its distributed execution design. K-Pg also borrows aspects of 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.

3.1 Timely dataflow

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

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

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. In timely dataflow a frontier only ever advances, and the set of times in advance of the frontier strictly decreases. This progress information tells operator shards when they have received all records with certain timestamps, at which point it may be appropriate for the operator shard to take some action.

User code can programmatically construct dataflows, interact with the inputs to these dataflows, and invoke the worker to schedule 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. Timely dataflow does not prevent the sharing of state between operators and dataflows within a worker, but it does not itself provide meaning or structure to this shared state other than the guarantees its frontiers provide.

3.2 Differential dataflow

Differential dataflow computations are initially defined as functional transformations of time-varying *collections*. A differential dataflow collection is parameterized by
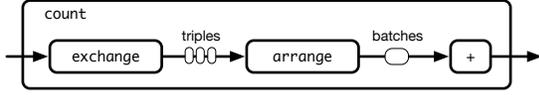
Figure 2: The `count` operator, decomposed into data exchange, arrangement, and a shell operator that reports the accumulated count from indexed batches.

three types, *Data*, *Time*, and *Diff*. The *Time* type must be a lattice (supporting the operations less than, equals, least upper bound, and greatest lower bound) and the *Diff* type must be a commutative group (with a + operator and a zero element), often the integers. A differential dataflow collection can be interpreted as a function from *Data* to *Diff* that can vary arbitrary at each *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`, `concat`, `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. Other operators exist, and the set can be further expanded as new implementations land, but the set above is sufficient for all applications we discuss in this paper.

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 timely dataflow frontier, from the pointwise accumulation of update triples at times less or equal to *t*.

$$Collection@t[data] = \sum_{\substack{(data, time, diff) \\ time \leq t}} diff$$

Each differential dataflow operator's output stream must 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 by the ≤ operator, which can result in more efficient differencing for iterative computations (among others), but with more subtle state management and operator implementations.

3.3 Design modifications

K-Pg departs from Naiad's differential dataflow design by breaking stateful operator implementations (for example: `join`, `distinct`, `count`) into two parts: an `arrange` operator, which exchanges, batches, and indexes updates,

and thinner shell operators which each apply operator-specific logic using these shared indices. The decomposition is depicted in Figure 2 for the `count` operator, and is similar for other stateful operators. This departure may appear superficial, but it fundamentally changes the implementation and performance characteristics of the system.

Arrangements act as high-throughput multi-version indices that can safely share read access among operators. The `arrange` operator is new to K-Pg, as is the associated inter-operator and inter-dataflow sharing of indexed state in a distributed dataflow system. The shell operators in K-Pg have substantially different implementations (often much simpler) when given their input as streams of indexed batches rather than streams of independent tuples, but some new care must be taken to maintain high throughput in all cases. We discuss the `arrange` operator in Section 4 and new operator designs in Section 5.

A second important departure of K-Pg lies in a set of design principles we impose to increase the likelihood that an unanticipated computation executes robustly, despite (or perhaps because of) the absence of frustrating systems knobs. These principles intend to ensure that arrangements and operators in K-Pg work "as expected". Each take only the minimal time and space required, up to constants, and transition smoothly between low-latency and high-throughput operation. We will invoke these principles in our design discussions, and in all cases we view violations of these principles as problematic. Naiad's differential dataflow prototype violates each of these principles.

**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. Systems that impose per-time coordination overhead must either limit their throughput, or compromise their fidelity to the source timestamped data.

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 indexed operator state should be at worst one sequential pass for each batch (though ideally to a sparse subset). Random access patterns reduce the effectiveness of batching, and eventually limit a system by its random rather than sequential throughput to its storage.

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,*
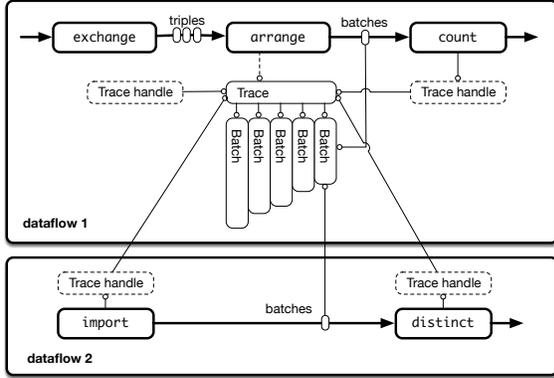
Figure 3: A worker-local overview of arrangement. Here the arrangement is constructed for the `count` operator, but is shared with a `distinct` operator in another dataflow.

time) pairs in each collection. Systems that eagerly materialize data and only later accumulate the results down (for example, Spark-style batch processors) may spill out of memory or even overwhelm local temporary storage.

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. Disproportionate computation limits our ability to scale our system with low latency, as coordinating workers take time proportional to the maximum of the participants.

4 Arrangements

Our most substantial design departure from standard dataflow 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 update batches. Arrangements allow K-Pg to spend the communication, computation, and memory required to arrange data once.

In this section we work through the design and implementation of arrangements, and explain how they support shared indexed data under high update throughputs. Figure 3 sketches the elements and some uses of an arrangement, which we will further develop in the text.

4.1 Collection traces

Following prior work [19], 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, either as a dataflow input or as the output of a dataflow operator whose inputs have advanced. Because times may be partially ordered, there is not necessarily a fixed order (e.g. by time) on the triples in a collection trace; instead a timely dataflow frontier indicates which times may still be observed.

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 the batch contains exactly those updates whose times are in advance of the lower frontier and not in advance of the upper frontier. The upper frontier of each batch should match the lower frontier of the next batch, and the growing list of batches reports the developing history of committed updates triples. A batch may be empty, which indicates that no updates exist in the indicated range of times. 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 evolving collection history.

To support operators, each batch should be indexed by *data*, so that it can provide random access to the history of each *data* (the set of its *(time, diff)* pairs). A trace will attempt to maintain relatively few batches (by merging existing batches) so that operators can efficiently navigate the union of all batches. Each reader of a trace holds a *trace handle*, which provides access to a cursor that can navigate the multiversioned index *as of* any time in advance of a frontier the trace handle holds. The set of trace reader frontiers indirectly reveal which updates can no longer be distinguished by any readers, and which updates can be coalesced.

4.2 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 without violating its obligations to readers of the trace.

At a high level, the `arrange` operator buffers incoming update triples until the input frontier advances, at which point it extracts and indexes all buffered updates not in advance of the newly advanced input frontier. A shared reference to this 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 several details are important if we aim to satisfy our design principles.

Input buffering. Incoming update triples are buffered in what is effectively a partially evaluated merge sort of *(data, time, diff)* triples: 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. Although the input frontier can advance in large steps, the `arrange` operator creates only one update batch and informs the timely dataflow system about changes in its output frontier 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, as it supports the appending of batches (and internally, their compaction). Importantly, while the `arrange` operator has a shared reference to the trace it does not keep the trace alive (it is a "weak" reference) so that should all readers drop their references to the trace, the trace (and its references to batches) are also dropped. In this case, the `arrange` operator will continue to produce indexed batches, but it will not (and cannot) merge them into the now non-existent trace. This optimization substantially improves performance in cases where a trace index can be dropped, but the stream of changes is still live, for example a join against static data which we see often in the processing of static graphs.

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. When a merge completes the new batch is installed and references to the merged batches are dropped.

A large constant of proportionality performs merges eagerly and trades latency away for improved throughput, whereas a small constant provides lower latency but impairs throughput as K-Pg maintains more open batches for operators to navigate. A charging argument shows that a constant of two ensures that merges complete before their results are required for a new merge, and K-Pg enforces this choice.

Consolidation. A trace can coalesce timestamps that are indistinguishable to all trace readers and consolidate updates at now indistinguishable times, analogous to MVCC "vacuuming". Each reader's trace handle maintains a frontier of times and restricts its reader to times in advance of this frontier. 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.

This compaction logic is borrowed from Naiad's prototype, but the mathematics of compaction have not been reported previously. Appendix A presents the definition of $rep_F(t)$ and proofs of its optimality and correctness.

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.

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 times in advance of this frontier. The trace itself tracks outstanding trace handle frontiers, which indirectly inform it about times that are indistinguishable to all readers (and which can be coalesced).

Many operators (including `join` and `group`) only need access to their accumulated input collections for times in advance of their input frontiers. As these frontiers advance, the operators are able to advance the frontier on their trace handles and still function correctly. Some operators are also able to drop their trace handles entirely, notably the `join` operator when its opposite

input ceases changing. These actions, advancing the frontier and dropping trace handles, provide the trace with the opportunity to consolidate its 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 before it was used to import the trace. 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 surprisingly large batch sizes and recent timestamps.

5 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.

5.1 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.

5.2 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.

5.3 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 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. 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. The join operator is bi-linear, and only needs each input traces in order to respond to updates from its *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 especially helpful when either input is static, as in static graph processing.

Alternating seeks. Join can receive input batches of substantial size, especially when importing an already maintained arranged collection. 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 the keys do not match we seek forward for the larger key in the cursor with the smaller key. This pattern ensures that we perform work at most linear in the smaller of the two sizes, seeking rather than scanning through the cursor of the larger trace, even when it is supplied as an input batch.

Amortized work. The `join` operator may be called upon to produce a significant amount of output data that can be reduced only once it crosses an exchange edge for a downstream operator. If each input batch is immediately processed to completion workers may be overwhelmed 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", limited batches of computation which can each be executed until sufficiently many outputs are produced and are then suspended. These futures make copies of the shared batch and trace references they require, and so do not block state maintenance for other operators.

5.3.2 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. At each time the output might change, we reform the input and apply the reduction function, and compare the results to the reformed output to determine if output changes are required.

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

Output arrangements. The `group` operator uses a 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 trace, just as the `arrange` operator does. It is not uncommon, 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.

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.

5.4 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.

6 Evaluation

We now evaluate our main hypotheses: that through multiple types of sharing, K-Pg can (*i*) outperform specialized systems in their own domain, (*ii*) remain viable in mature domains, and (*iii*) enable new solutions in open domains. We demonstrate this across multiple application areas, and conclude that specialized systems could be expressed as layers atop K-Pg.

We evaluate K-Pg on a four-socket NUMA system equipped with four Intel Xeon E5-4650 v2 CPUs, each with 10 physical cores and 512 GB of aggregate system memory. 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. 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.

We stress that we compare K-Pg to systems of many different classes, some of which provide significant additional functionality that K-Pg does not provide. Most prominently, K-Pg is not a transaction processor. Rather, K-Pg accepts changes from a source-of-truth system, and should be viewed as a high-performance analytics replacement.

6.1 Relational analytics

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. We manually implemented the same queries in K-Pg.

Nikolic et al. [22] study the problem of maintaining the TPC-H queries as they incrementally load the source data, and the effect of *logical* batching on throughput (which has the potentially undesirable effect of changing the result of the computation). A fair comparison is complicated by variation in the set of operators with

incremental implementations: DBToaster is fast on relational queries, but must occasionally fall back to full re-evaluation for queries with complex aggregation.

Our experiments show that K-Pg can maintain often substantially higher throughputs via physical batching and worker scaling.

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

Physical batching. Figure 4b reports K-Pg's 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 with larger batch sizes. Latency increases with the batch sizes, nominally at first and then more significantly.

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, with a corresponding reduction in latency, though several queries which involve global aggregation do not. The temporal nature of the queries, *histories* for each aggregate, means they resist traditional techniques like pre-aggregation, and instead require techniques like parallel-prefix aggregation to parallelize effectively.

TPC-H highlights several opportunities for further 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 performed by [22]. 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.

In Appendix B we report TPC-H measurements with increased logical batching. Table 5 reports the processing rates for logical batches of 100,000 records, along with the rates of [22]; K-Pg offers a more uniformly high throughput which scales out to multiple workers, though for some "easier" queries (q6, q14) K-Pg lags behind [22]. Table 6 reports elapsed times to process each query as a single logical batch, compared with evaluations from recent work [11] for Postgres, Spark, HyPer, and Flare; for these measurements K-Pg is faster than the first two systems, and not as fast as the latter two.

6.2 Graph workloads

We next evaluate K-Pg on graph workloads, ranging from large computations on static graphs, to interactive queries against evolving graphs. Our experiments show that K-Pg is much faster than existing general-purpose graph processors, not as fast as specialized static graph processors, and offers substantially higher throughput than interactive graph databases.

Batch graph computation. In Appendix C we evaluate K-Pg on standard computations of reachability, breadth-first distance labeling, and undirected connectivity, on three standard social networks: LiveJournal, Orkut, and Twitter, in tables 7, 8, and 9 respectively. Our measurements indicate that K-Pg is consistently faster than systems like BigDatalog, Myria, SociaLite, and GraphX, but is substanially less efficient than purpose-written single-threaded code on pre-processed graph data. We did find that when we modified the purpose-written code to use hash maps rather than arrays for vertex state, as might be required for more general vertex identifiers, K-Pg was immediately competitive at between two and four cores. Our conclusion is that K-Pg will (grossly) underperform specialized graph processors executing on pre-processed data, but that this gap closes if one must account for edge and vertex pre-processing, or any form of graph changes.

Interactive graph queries. Pacaci et al [23] 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 as differential dataflows where the query arguments are independent collections that may be modified to introduce or remove specific query instances. This gives K-Pg the benefit of treating the queries as stored procedures, an advantage over systems that do not do so.

Figure 5 reports latency distributions for the four query classes on an evolving graph of 10 million nodes and 64 million edges, under an open-loop load of 200,000 changes per second, half graph modifications and half modifications to the query collections. These latencies are comparable to those reported in [23] for single queries against a static graph (reproduced in Table 10, with measurements for K-Pg), except for the shortest path queries due to the higher worker load. The throughput exceeds that reported in [23] by two orders of magnitude, primarily because K-Pg compiles the repeated query structure to dataflow in which 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

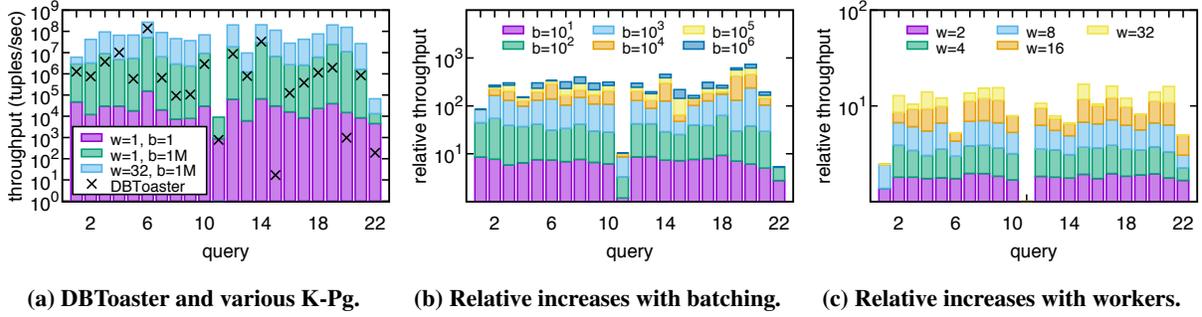| (a) DBToaster and various K-Pg. | (b) Relative increases with batching. | (c) Relative increases with workers. |

Figure 4: Absolute and relative query rates for the 22 TPC-H queries. K-Pg can achieve higher throughputs due to physical batching and scaling out, while producing the same output as single-thread DBToaster.

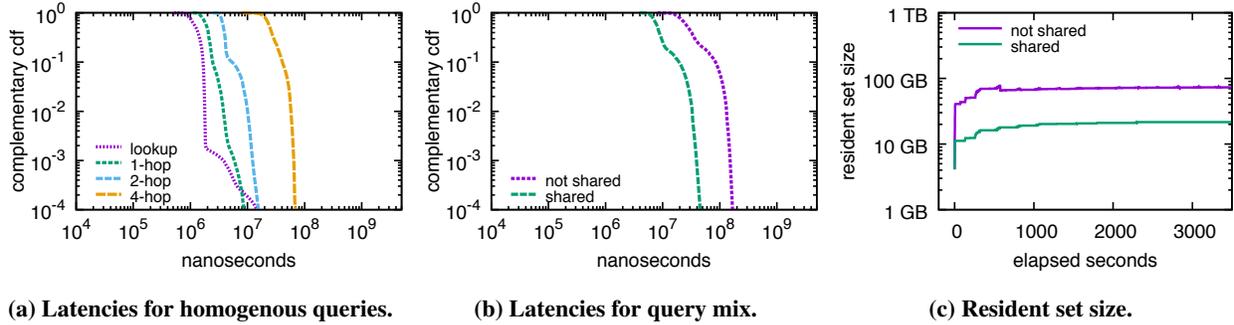| (a) Latencies for homogenous queries. | (b) Latencies for query mix. | (c) Resident set size. |

Figure 5: Interactive graph query benchmarks. 32 workers, 10M nodes, 32M edges, 200K updates / second. Query latencies are low even under 100k queries per-second load. Sharing reduces both latency and memory.

consistent latency penalty due to the increased system load, which would only increase as more query classes are maintained.

Figure 5c reports the memory footprint for the query mix with and without sharing, for an hour-long execution. The memory footprint stabilizes at 20GB for the shared implementation, and roughly four times that for the not shared implementation. There are five uses of the graph across the four queries, but also per-query state that is not profitably shared. The absolute numbers are perhaps higher than they need to be, due in part to our discouraging jemalloc from aggressively releasing memory and to our use of 64 bit graph identifiers, timestamps, and differences. A user program can modify any of these.

6.3 Datalog workloads

Datalog is a relational language in which the query results are the fixed point of repeated application of recursively defined productions. 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, 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" [26].

Our experiments show that K-Pg is generally faster than distributed Datalog engines and matches the best shared memory engine. At the same time, K-Pg natively supports incremental and decremental updates to Datalog computations, and interactive top-down queries.

Bottom-up (batch) evaluation. In Appendix D we evaluate K-Pg relative to distributed and shared-memory Datalog engines, using their benchmark queries and datasets ("transitive closure" and "same generation" on trees, grids, and random graphs). Table 11 reports that K-Pg generally outperforms the distributed systems, and is comparable to the best shared-memory engine (DeALS).

K-Pg's generality brings some benefits to this domain: single-threaded, DeALS takes 312 seconds to determine the transitive closure of a random graph, but K-Pg can determine the strongly connected component structure of the same graph, summarizing the transitive closure, in just 274ms. Our strongly connected components implementation uses doubly nested non-monotonic iteration, and is not expressible in Datalog.

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

Table 2: Interactive and full computation of three queries, on K-Pg with 32 workers. The interactive latencies are medians and maximums of 100 queries.

Top-down (interactive) evaluation. If the user imposes constraints on a target query, for example $tc(\textit{"david"}, ?)$, one can explore the space of facts from the possible goals back to the facts of the base relations. The "magic set" transformation [6] rewrites such queries as bottom-up computations with a new base relation that seeds the bottom-up derivation with query arguments; the rewritten rules derive facts only with the participation of some seed record. 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.

Table 2 reports median and maximum latencies for 100 random arguments for three interactive queries on three of the benchmark graphs from above, and the times for full evaluation of the related query in both cases using 32 workers. In many cases the times reduce from seconds to milliseconds. In some cases the transformed queries are slower, most prominently for sg(x,?) on grid-150, which is a known problem with the automatic transformation.

6.4 Program Analysis

Graspan [27] is a system built to perform static analysis of large code bases, created in part because existing systems reportedly 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 3 and Table 4 reproduce the running times reported in [27], and reports those of K-Pg for their two program analyses, *dataflow* and *points-to*, respectively. The *dataflow* query is propagates null assignments 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 (from 14x to 550x), which we attribute to our re-use of operators from an optimized system. A complete implementation of Graspan—query parsing, dataflow construction, input parsing and loading, dataflow execution—is 179 lines of code on top of K-Pg.

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 3: System performance for the three graphs for the *dataflow* analysis. The first two lines are reproduced from [27]. 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 4: System performance for the three graphs for the *points-to* analysis. The first two lines are reproduced from [27]. Further K-Pg implementations correspond to an optimized query (Opt), and the optimized query without sharing (NoS).

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 actually 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 use 32bit timestamps and differences, which brings the memory footprint to within RAM limits; this optimized version runs only 20% faster.

Optimization. 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 (value aliasing is eventually restricted by dereferences, and this restriction can be performed before forming all value aliases), which results in a much more efficient computation, one that re-uses relations multiple times. Table 4 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. Table 3 reports the median and maximum latencies to remove the first 1,000 null assignments from the completed *dataflow* analysis and correct the set of reached program locations. While there is some variability the timescales are largely interactive, and suggest the potential for a improved developer experience.

(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.
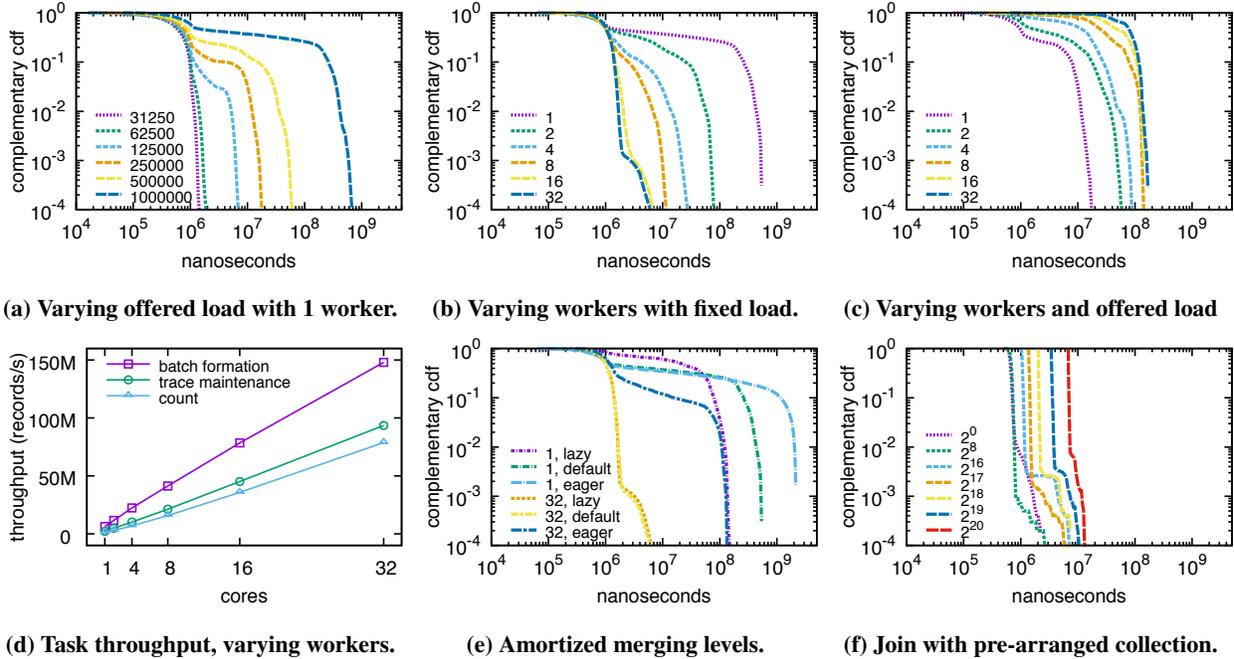
(f) Join with pre-arranged collection.

Figure 6: Microbenchmarks for arrangement and join execution.

6.5 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 primarily 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 6a 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 automatically trade latency for throughput until equilibrium is reached.

Strong scaling. Figure 6b 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 tradeoff swings in favor of latency.

Weak scaling. Figure 6c reports the latency distributions for varying numbers of workers as we proportionately increase the number of keys and offered load. While the latency distributions do increase, as data exchange becomes more complicated, they do stabilize.

Throughput. Figure 6d breaks down the peak throughputs of sub-components of arrangement: batch formation,

trace maintenance, and a maintained `count` operator. To allow the throughput to vary, this experiment uses repeated rounds of batches of 10,000 updates for each worker rather than the open-loop harness. Scaling is linear out to 32 workers.

Amortized merging. Figure 6e reports the latency distributions for one and 32 workers each with three different merge amortization coefficients: eager, default, and lazy. For a single worker the lazier settings have smaller tail latencies, but are more often in that 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 [12], we conclude that rare but large dataflow interruptions are nonetheless significant impediments to strong scaling.

Join proportionality. Figure 6f 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.

7 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 [10], Dryad-LINQ [28], Spark [29] execute large computations as directed acyclic graphs (DAGs) of independent, finite-duration tasks. The independence of tasks allows these systems to scale, and re-execution allows them to tolerate failures [21, 29]. However, independence trades away efficiency for resilience.

DryadLINQ and Spark re-use datasets (respectively, Nectar [14] 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 [5], STREAM [4], and TelegraphCQ [8] 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 [7] and Mill-Wheel [3] 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 sharing data between operators and across dataflows. We believe 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 [20], supports iterative computation through cyclic dataflow graphs and partially-ordered timestamps. In a sense, Naiad shares state across iterative 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.

Specialized systems. Relatively few graph processors attack the problem of computations over continually evolving graphs. Chronos [15] 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 [17] 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" [26], a technique specialized to relational equijoins.

8 Conclusions

We presented K-Pg, a system that supports holistic, temporal, and iterative sharing. K-Pg's design required rethinking 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 often better than existing systems on a variety of tasks.

References

[1] https://www.rust-lang.org.

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

[3] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: Fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, August 2013.

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

[5] 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, pages 13–24, New York, NY, USA, 2005. ACM.

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

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

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

[9] 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, pages 1:1–1:15, New York, NY, USA, 2015. ACM.

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

[11] Gregory Essertel, Ruby Tahboub, James Decker, Kevin Brown, Kunle Olukotun, and Tiark Rompf. Flare: Optimizing apache spark with native compilation for scale-up architectures and medium-size data. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'18. USENIX Association, 2018.

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

[13] 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)*, pages 17–30, Hollywood, California, USA, October 2012.

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

[15] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. Chronos: A graph engine for temporal graph analysis. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 1:1–1:14, New York, NY, USA, 2014. ACM.

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

[17] 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, pages 5:1–5:6, New York, NY, USA, 2016. ACM.

[18] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 239–250, 2015.

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

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

[21] 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, pages 113–126, Berkeley, CA, USA, 2011. USENIX Association.

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

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

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

[25] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. Storm @twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 147–156, 2014.

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

[27] Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and*

Operating Systems, ASPLOS '17, pages 389–404, New York, NY, USA, 2017. ACM.

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

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

[30] 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, pages 614–630, New York, NY, USA, 2017. ACM.

A Compaction Theorems

Let F be an antichain of partially ordered times (a "frontier"). Writing $f \geq F$ to mean f is greater than some element of F, we will say that two times t_1 and t_2 are "indistinguishable as of F", written $t_1 \equiv_F t_2$, when

$$t_1 \equiv_F t_2 \text{ when } \forall_{f \geq F}(t_1 \leq f \text{ iff } t_2 \leq f)$$

As performed in the Naiad prototype, we can determine a representative for a time t relative to a set F using the least upper bound ($\wedge$) and greatest lower bound ($\vee$) operations of the lattice of times, taking the greatest lower bound of the set of least upper bounds of t_i and elements of F:

$$rep_F(t) := \vee_{f \in F}(t \wedge f)$$

The rep_F function finds a representative that is both correct (t and $rep_F(t)$ compare identically to times greater than F) and optimal (two times comparing identically to all times greater than F map to the same representative).

The formal properties of rep_F rely on properties of the $\wedge$ and $\vee$ operators, that they are respectively upper and lower bounds of their arguments, and their optimality:

$$b \leq a \text{ and } c \leq a \;\rightarrow\; (b \wedge c) \leq a \qquad (lub)$$
$$a \leq b \text{ and } a \leq c \;\rightarrow\; a \leq (b \vee c) \qquad (glb)$$

In particular, we will repeatedly use that if $t_1 \leq (t_2 \vee f)$ for all $f \in F$, then $t_1 \leq rep_F(t_2)$.

THEOREM A.1 (CORRECTNESS). *For any lattice element t and set F of lattice elements, $t \equiv_F rep_F(t)$.*

PROOF. We prove both directions of the implication in $\equiv_F$ separately, for all $f \geq F$. First assume $t \leq f$. By assumption, f is greater than some element f' of F, and so $t \wedge f' \leq f$ by the (lub) property. As a lower bound, $rep_F(t) \leq t \wedge f'$ for each $f' \in F$, and by transitivity $rep_F(t) \leq f$. Second assume $rep_F(t) \leq f$. Because $t \leq (t \wedge f')$ for all $f' \in F$, then $t \leq rep_F(t)$ by the (glb) property and by transitivity $t \leq f$. □

query	DBToaster	K-Pg (w=1)	K-Pg (w=32)
TPC-H 01	4,372,480	9,341,713	31,283,993
TPC-H 02	691,260	4,388,761	29,651,632
TPC-H 03	4,580,003	11,049,606	37,263,673
TPC-H 04	9,752,380	9,046,854	30,886,269
TPC-H 05	509,490	5,802,513	27,952,246
TPC-H 06	101,327,791	33,090,863	65,335,474
TPC-H 07	646,018	7,551,628	30,962,626
TPC-H 08	221,020	4,949,412	28,230,062
TPC-H 09	76,296	2,932,421	18,119,469
TPC-H 10	5,964,290	9,708,371	25,037,510
TPC-H 11	591,716	1,720,655	1,749,464
TPC-H 12	7,469,474	11,258,702	33,975,983
TPC-H 13	474,765	1,446,223	16,792,703
TPC-H 14	53,436,252	21,908,762	38,843,085
TPC-H 15	964	5,057,397	23,122,916
TPC-H 16	58,721	4,435,818	23,495,608
TPC-H 17	131,964	5,218,907	25,888,103
TPC-H 18	971,313	5,854,293	29,574,347
TPC-H 19	8,776,165	22,696,357	36,393,109
TPC-H 20	1,871,407	16,089,949	46,456,453
TPC-H 21	407,540	1,968,771	10,928,516
TPC-H 22	815,903	1,843,397	15,233,935

Table 5: Streaming update rates (in tuples per second) for the 22 TPC-H queries at scale factor 10, with logical batching of 100,000 elements at the same time. Elapsed times for DBToaster are for one thread, and are reproduced from [22]. For K-Pg we report both one worker and 32 worker rates.

At the same time, rep_F is optimal in that two equivalent elements will be mapped to the same representative.

THEOREM A.2 (OPTIMALITY). *For any lattice elements t_1 and t_2 and set F of lattice elements, if $t_1 \equiv_F t_2$ then $rep_F(t_1) = rep_F(t_2)$.*

PROOF. For all $f \in F$ we have both that $t_1 \leq t_1 \wedge f$ and $f \leq t_1 \wedge f$, the latter implying that $t_1 \wedge f \geq F$. By our assumption, t_2 agrees with t_1 on times greater than F, making $t_2 \leq t_1 \wedge f$ for all $f \in F$. By correctness, $rep_F(t_2)$ agrees with t_2 on times greater than F, which includes $t_1 \wedge f$ for $f \in F$ and so $rep_F(t_2) \leq t_1 \wedge f$ for all $f \in F$. Because $rep_F(t_2)$ is less or equal to each term in the greatest lower bound definition of $rep_F(t_1)$, it is less or equal to $rep_F(t_1)$ itself. The symmetric argument proves that $rep_F(t_1) \leq rep_F(t_2)$, which implies that the two are equal (by antisymmetry). □

B Relational computations

Table 5 reports throughput in tuples per second for [22] and K-Pg on the scale factor 10 TPC-H workload with logical batches of 100,000 elements. K-Pg has a generally higher and more consistent rate, though is less efficient on lighter queries (q04, q06, and q14); K-Pg performs no pre-aggregation, which could improve its rates for logically batched queries. For 32 workers, almost all rates are above 10 million updates per second, which correspond to latencies below 10ms between reports.

query	Postgres	SparkSQL	HyPer	Flare	K-Pg
TPC-H 01	241,404	18,219	603	530	7,789
TPC-H 02	6,649	23,741	59	139	2,426
TPC-H 03	33,721	47,816	1,126	532	5,948
TPC-H 04	7,936	22,630	842	521	8,550
TPC-H 05	30,043	51,731	941	748	14,001
TPC-H 06	23,358	3,383	232	198	1,185
TPC-H 07	32,501	31,770	943	830	12,029
TPC-H 08	29,759	63,823	616	1,525	19,667
TPC-H 09	64,224	88,861	1,984	3,124	27,873
TPC-H 10	33,145	42,216	967	1,436	4,559
TPC-H 11	7,093	3,857	131	56	1,534
TPC-H 12	37,880	17,233	501	656	4,458
TPC-H 13	31,242	28,489	3,625	3,727	3,893
TPC-H 14	22,058	7,403	330	278	1,695
TPC-H 15	23,133	14,542	253	302	1,591
TPC-H 16	13,232	23,371	1,399	620	2,238
TPC-H 17	155,449	70,944	563	2,343	17,750
TPC-H 18	90,949	53,932	3,703	823	9,426
TPC-H 19	29,452	13,085	1,980	909	2,444
TPC-H 20	65,541	31,226	434	870	4,658
TPC-H 21	299,178	128,910	1,626	1,962	29,363
TPC-H 22	11,703	10,030	180	177	2,819

Table 6: Elapsed milliseconds for the 22 TPC-H queries at scale factor 10, each using a single core. Elapsed times for the four other systems are reproduced from [11]. K-Pg used as a batch processor is not as fast as the best systems (HyPer and Flare), but is faster than other popular frameworks.

Table 6 reports elapsed times for K-Pg applied to the scale-factor 10 TPC-H workload. We also reproduce several measurements from [11] for other systems. All are executed with a single thread. K-Pg used as a batch processor is not as fast as the best systems (HyPer and Flare), but is faster than other popular frameworks.

C Graph computations

We now report on the performance of K-Pg used as a batch graph processor, where the inputs are static collections of edges defining a directed graph. Following [24] 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 the process 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 graphs are made available sorted by the source as in the forward index. The reported times for the three problems are then any further time required after the index construction, where the first two problems require a forward index and undirected connectivity requires indices in both directions.

We report times for three graphs: `livejournal` in Table 7, `orkut` in Table 8, and `twitter` in Table 9. We also reproduce measurements reported in [24] for several other systems. We include running times for

System	cores	index-f	reach	sssp	index-r	wcc
Single thread	1	-	0.40s	0.40s	-	0.29s
w/hash map	1	-	4.38s	4.38s	-	8.90s
BigDatalog	120	-	17s	53s	-	27s
Myria	120	-	5s	70s	-	39s
SociaLite	120	-	52s	172s	-	54s
GraphX	120	-	36s	311s	-	59s
K-Pg	1	4.39s	8.50s	13.14s	7.56s	23.97s
K-Pg	2	2.49s	4.33s	6.71s	4.01s	12.95s
K-Pg	4	1.39s	2.31s	3.58s	2.06s	6.29s
K-Pg	8	0.74s	1.20s	1.79s	1.03s	3.41s
K-Pg	16	0.54s	0.62s	0.90s	0.58s	1.71s
K-Pg	32	0.55s	0.51s	0.59s	0.41s	0.90s

Table 7: System performance on various tasks on the 4.8M node, 68M edge `livejournal` graph.

System	cores	index-f	reach	sssp	index-r	wcc
Single thread	1	-	0.46s	0.46s	-	0.52s
w/hash map	1	-	11.56s	11.56s	-	19.00s
BigDatalog	120	-	20s	39s	-	33s
Myria	120	-	6s	44s	-	57s
SociaLite	120	-	67s	106s	-	78s
GraphX	120	-	48s	67s	-	53s
K-Pg	1	14.02s	20.33s	24.65s	21.27s	47.79s
K-Pg	2	7.92s	10.29s	13.06s	11.49s	25.02s
K-Pg	4	4.25s	5.34s	6.21s	5.73s	12.38s
K-Pg	8	2.37s	2.68s	3.34s	3.03s	6.29s
K-Pg	16	1.43s	1.47s	1.60s	1.69s	3.30s
K-Pg	32	1.22s	1.11s	0.87s	1.05s	1.75s

Table 8: System performance on various tasks on the 3M node, 117M edge `orkut` graph.

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
K-Pg	1	162.41s	256.77s	310.63s	312.31s	800.05s
K-Pg	2	99.74s	131.50s	159.93s	164.12s	417.20s
K-Pg	4	49.46s	64.31s	77.27s	81.67s	200.28s
K-Pg	8	27.99s	33.68s	40.24s	43.20s	101.42s
K-Pg	16	18.04s	17.40s	20.99s	24.73s	51.83s
K-Pg	32	12.69s	11.36s	10.97s	14.44s	27.48s

Table 9: System performance on various tasks on the 42M node, 1.4B edge `twitter` graph.

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 of the measurements reported here. We also include the same times when the single-threaded implementations replace their array-indexed data structures with hash maps, as they might when the graph identifiers have not be pre-processed to be in a compact range.

The measurements indicate that K-Pg is at least as capable as these peer systems at graph processing, often with just a single core. K-Pg lags well behind graph processors that can rely on dense integer identifiers and direct array access, but appears likely to be competetive with graph processors that cannot rely on dense integer identifiers.

We next compare to measurements made of several graph databases, on relatively simpler read queries. Pacaci

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

Table 10: Average query latencies on a graph with 10M nodes and 64M undirected edges. Latencies for the first three systems are reproduced from [23]. K-Pg measurements are taken on a random graph with the same numbers of nodes and edges and are the average of 1,000 measurements. The batch size is the number of concurrent queries per measurement.

System	cores	tc(t)	tc(g)	tc(r)	sg(t)	sg(g)	sg(r)
BigDatalog	120	49s	25s	7s	53s	34s	72s
Spark	120	244s	OOM	63s	OOM	1955s	430s
Myria	120	91s	22s	50s	822s	5s	436s
SociaLite	120	DNF	465s	654s	OOM	17s	OOM
LogicBlox	64	NR	24420s	913s	58732s	326s	3363s
DLV	1	NR	13127s	9272s	OOM	105s	48039s
DeALS	1	NR	148s	321s	1309s	7.6s	2926s
DeALS	64	NR	5s	12s	48s	0.35s	79s
K-Pg	1	98.26s	132.23s	210.25s	1210.78s	4.43s	482.91s
K-Pg	2	53.42s	68.13s	111.98s	652.74s	2.76s	253.80s
K-Pg	4	27.85s	34.42s	57.69s	325.24s	1.63s	125.00s
K-Pg	8	15.37s	17.97s	30.90s	173.96s	1.06s	66.10s
K-Pg	16	9.63s	9.74s	16.66s	93.47s	0.69s	35.44s
K-Pg	32	7.18s	6.18s	9.45s	56.45s	0.60s	19.85s

Table 11: System performance on various Datalog problems and graphs. Times for the first four systems are reproduced from [24]. NR indicates the measurements were not reported, DNF indicates a run that lasted more than 24 hours, and OOM indicates the system terminated due to lack of memory.

et al [23] evaluated several graph database solutions with a mix of four queries: look-ups, one hop neighborhood, two-hop neighborhood, and "shortest path if not longer than four hops". We build query dataflows in K-Pg for these four queries and evaluate the latencies when we introduce new query seeds to each of the four types. In this set-up K-Pg treats the queries as stored procedures, which makes this an unfair comparison for those databases that do not do this.

Table 10 measures the average latency to perform and then await a single query in several systems, as well as the time to perform and await batches of increasing numbers of queries. While not providing the lowest latency for point look-ups, K-Pg provides excellent latencies for other queries, and supports increasing batches without much degradation.

D Datalog computations

Table 11 reports elapsed seconds first for distributed systems, then for single-machine systems, and then for K-Pg at varying numbers of workers. The workloads are "transitive closure" (*tc*) and "same generation" (*sg*) on supplied graphs that are trees (*t*), grids (*g*), and random graphs (*r*).

K-Pg is generally competitive with the best of the specialized Datalog systems (here: DeALS), and generally out-performs the distributed data processors. BigDatalog competes well on transitive closure due to an optimization for linear queries where it broadcasts its input dataset to all workers; this works well with small inputs, as here, but is not generally a robust strategy.