Optimizing Declarative Graph Queries at Large Scale

Technical Report

Qizhen Zhang, Akash Acharya, Hongzhi Chen[§], Simran Arora, Jiacheng Wu[†], Yucheng Lu[*], Ang Chen[‡], Vincent Liu and Boon Thau Loo

University of Pennsylvania, [§]The Chinese University of Hong Kong, [†]Tsinghua University, [*]Cornell University, [‡]Rice University

ABSTRACT

This paper presents GraphRex, an efficient, robust, scalable, and easy-to-program framework for graph processing on datacenter infrastructure. To users, GraphRex presents a declarative, Datalog-like interface that is natural and expressive. Underneath, it compiles those queries into efficient implementations. A key technical contribution of GraphRex is the identification and optimization of a set of global operators whose efficiency is crucial to the good performance of datacenter-based, large graph analysis. Our experimental results show that GraphRex significantly outperforms existing frameworks—both high- and low-level—in scenarios ranging across a wide variety of graph workloads and network conditions, sometimes by two orders of magnitude.

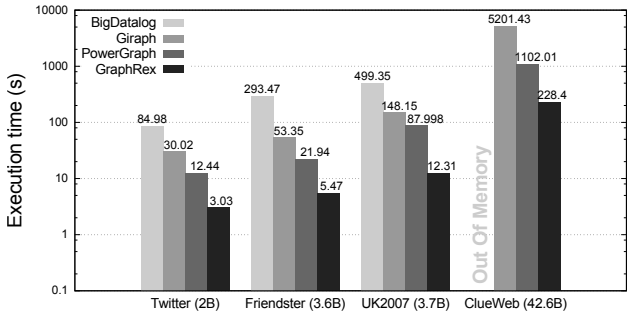


Figure 1: Performance comparison (log scale) of SSSP between declarative systems: BigDatalog and GraphRex, and low-level graph systems: Giraph and PowerGraph on large graphs. All systems are run in a datacenter with 6 TB RAM and 1.6 K cores in aggregate.

1 INTRODUCTION

Over the past decade, there has been a proliferation of graph processing systems, ranging from low-level platforms [21, 35, 44, 46] to more recent declarative designs [57]. While users can deploy these systems in a variety of contexts, the largest instances routinely scale to multiple racks of servers contained in vast datacenters like those of Google, Facebook, and Microsoft [54]. This trend of large-scale distributed data processing is likely to persist as data continues to accumulate.

These massive deployments are in a class of their own: their size and the inherent properties of the datacenter infrastructure present unique challenges for graph processing. To highlight these performance issues on practical workloads, Figure 1 illustrates, for multiple graph processing systems and billion-edge graphs, the running time of a single-source shortest path (SSSP) query on a representative datacenter testbed. We tested four systems: (1) BigDatalog [57], a recent system that provides a declarative interface to Spark; (2) Giraph [21], a platform built on Hadoop that powers Facebook's social graph analytics; (3) PowerGraph [29], a highly optimized custom framework; and as a sneak preview of the space of possible improvement (4) GraphRex, our system for large-scale datacenter-based graph processing. As the results demonstrate, while the three existing systems are capable of scaling to billion-edge workloads, our approach leads to *up to two orders of magnitude* better performance.

The above results barely scratch the surface of optimization opportunities for large-scale graph queries in datacenters. We note two significant opportunities that are underexplored in previous work:

Opportunity #1: The impact of graph workload characteristics. Real-world graphs exhibit particular qualities that incur serious performance degradation if ignored. One example is a *power-law distribution with high skew*, where most vertices are of fairly low degree, but a few vertices have very high edge counts. Even within a single execution, the optimal query plan may then differ depending on which vertex is being processed. Another is a proclivity to produce redundant data, e.g., in the case of label propagation where nodes can often reach each other via many paths. Each of these presents opportunities for optimization.

Opportunity #2: The impact of datacenter architecture. Performance can also depend heavily on the underlying infrastructure. Consider the rack-based architecture of Facebook's most recent datacenter design [15]. Racks of servers are connected through an interconnection network such that a given server's bandwidth to another can *differ by a factor of four* depending on whether the other server is in the same rack or not. Though this type of structure is ubiquitous in today's datacenters due to practical design constraints [15, 31, 59],

existing processing systems (e.g., [21, 29, 57]) have largely ignored these effects, typically assuming uniform connectivity that is not the case in modern datacenters.

The GraphRex system. To exploit these two opportunities, this paper explores a suite of optimization techniques specifically designed to ensure good performance for massive graph queries running in modern datacenters. We have developed GraphRex (Graph Recursive Execution) that significantly outperforms state-of-the-art graph processing systems.

The performance of GraphRex stems, in part, from the high-level language it presents. It compiles Datalog queries into distributed execution plans that can be processed in a massively parallel fashion using distributed semi-naïve evaluation [43]. While prior work has noted that declarative abstractions based on Datalog are natural fits for graph queries [8, 57], these systems fall short on constructing efficient physical plans that (1) scale to large graphs that cannot fit in the memory of one machine, and (2) scale to a large number of machines where the network is a bottleneck. GraphRex goes beyond these systems by combining traditional query processing with *network-layer optimizations*. It aims to achieve the best of both worlds: ease of programming using a declarative interface and high performance on typical datacenter infrastructure. Our key observation is that these two goals exhibit extraordinary synergy.

We note that this synergy comes with a requirement: that the graph processing system be aware of the underlying physical network. In a private cloud datacenter where the operator has full-stack control of the application and infrastructure, visibility is trivial. In a public cloud, the provider would likely expose GraphRex "as a service" in order to abstract away infrastructure concerns from users.

Our contributions. We make the following contributions in the design and implementation of GraphRex:

(i) Datacenter-centric relational operators for large-scale graph processing. We have developed a collection of optimizations that, taken together, specialize relational operators for datacenter-scale graph processing. The scope and effect of these optimizations is broad, but their overarching goal is to reduce data and data transfer, particularly across "expensive" links in the datacenter. These techniques, applied using knowledge of the underlying datacenter topology and semantics of relational operators in GraphRex's declarative language, allow us to significantly outperform existing graph systems.

(ii) Dynamic join reordering. We also observe that graph queries may require changing join reorderings as join selectivity is heavily influenced by graph degrees; and degrees can vary significantly across a graph. Inspired by prior work on pipelined dynamic query reoptimizations [17], we develop a distributed join operator that can dynamically adapt to changing join selectivities as the query execution progresses along different regions of a graph.

(iii) Implementation and evaluation. We have implemented a prototype of GraphRex. Based on evaluations on the CloudLab testbed, we observe that GraphRex has dominant efficiency over existing declarative and low-level systems on a wide range of real-world workloads and micro-benchmarks. GraphRex outperforms BigDatalog by factors of 11–109×, Giraph by factors of 5–26×, and PowerGraph by 3–8×. In addition, we find that GraphRex is more robust to datacenter network practicalities such as cross-traffic and link degradation because our datacenter-centric operators significantly reduce the amount of traffic traversing bottleneck links.

2 BACKGROUND

Today's graph processing processing systems span multiple layers. *Applications* are written in low-level languages like C++ or Java; they run on *frameworks* including GraphX, Giraph; which in turn run in large *datacenter deployments* like those of Google, Amazon, Microsoft, and Facebook. These systems are powerful, efficient, and robust, but difficult to program and tune [12, 57].

2.1 Declarative Graph Processing

GraphRex uses Datalog as a declarative abstraction, drawing inspiration from recent work [51, 57]. Datalog is a particularly attractive choice for writing graph queries because of its natural support for *recursion*—a key construct in a wide variety of graph queries [39, 56].

Datalog *rules* have the form p :- $q_1, q_2, ..., q_n$, which can be read informally as "q_1 and q_2 ... and q_n implies p." p is the *head* of the rule, and $q_1, q_2, ..., q_n$ is a list of *literals* that constitutes the *body* of the rule. Literals are either *predicates* over *fields* (variables and constants), or functions (formally, *function symbols*) applied to fields. The rules can refer to each other in a cyclic fashion to express recursion, which is particularly useful for graph processing. We adhere to the convention that names of predicates, function symbols and constants begin with a lower-case letter, while variable names begin with an upper-case letter. We use predicate, table, and relation interchangeably.

Query 1: Connected Components (CC)

```
cc(A,min<A>) :- e(A,_)
cc(A,min<L>) :- cc(B,L), e(B,A)
```

Our example above shows a classical graph query that computes connected components in a graph. This query takes a set of edges `e` as inputs, with `e(X,Y)` representing an edge from vertex `X` to vertex `Y`, and computes a `cc` tuple for each vertex, where the first field is the vertex and the second is a label for the vertex. The first rule initializes the

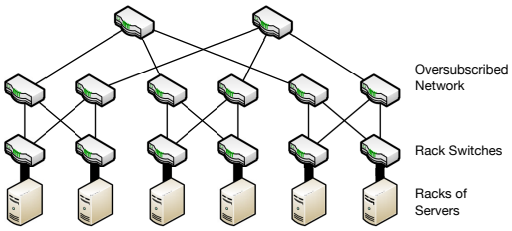


Figure 2: A canonical datacenter network. Racks contain dozens of servers connected by a single switch. Racks then connect via an oversubscribed network.

label of each vertex with its vertex ID. In the second rule, `cc(A,min<L>)` means that the tuples in `cc` are grouped by `A` first, and in each group, the labels `L` are aggregated with `min`. The rule is recursively evaluated so that the smallest label is passed hop by hop until all vertices in the same connected component have the same label. An equivalent program in Spark requires upwards of one hundred lines of code.

Partitioning graph data. Distributed graph processing requires specification of how the graph data and relations are partitioned. *Graph partitioning* maps vertices (or edges) to workers, and is useful when queries have consistent and predictable access patterns over data. In this paper, we assume a default graph partitioning where vertex id is hashed modulo the number of workers, although our optimizations are not restricted to, and indeed are compatible with, more advanced graph partitioning mechanisms. *Relation partitioning* refers to cases where an attribute of a relation is selected as *partition key* and all of its tuples with the same partition key are put in the same location. For example, in Query 1 (CC), `cc` has two attributes so it has two potential partitionings: `cc(@A,B)` and `cc(A,@B)`, where @ denotes the partition key.

2.2 Graph Queries in Datacenters

A crucial component for performance is an understanding of the deployment environment, which in the case of today's largest graph applications, refers to a datacenter. Modern datacenter designs, e.g., those of Google [59], Facebook [15], and Microsoft [31], have coalesced around a few common features, depicted in Figure 2, which are necessitated by practical considerations such as scalability and cost.

At the core of all modern datacenter designs are *racks* of networked servers [24, 42, 59]. The servers come in many form factors, but server racks typically contain a few dozen standard servers connected to a single *rack switch* that serves as a gateway to the rest of the datacenter network [52]. The datacenter-wide network that connects those rack switches is structured as a multi-rooted tree, as shown in Figure 2. The rack switches form the leaves of those trees [40].

The above architecture leads to several defining features in modern datacenter networks. One example: *oversubscription*. While recent architectures have striven to reduce oversubscription [11, 31], fundamentally, cross-rack links are much longer and therefore more expensive (as much as an order of magnitude) [42, 67]. As such, the tree is often thinned immediately above the rack level, i.e., oversubscribed, and it may be oversubscribed even further higher up. This is in contrast to racks' internal networks, which are well connected.

The result is that servers can often overwhelm their rack switch with too much traffic. A 1:y oversubscription ratio indicates that the datacenter's servers can generate $y\times$ more traffic than the inter-rack network can handle.[1] In essence, these networks are wagering that servers either mostly send to others in the same rack, or rarely send traffic concurrently. In this way, network connectivity is not uniform. Instead, datacenter networks are hierarchical, and job placement within the network affects application performance. Ignoring these issues can lead to poor results (see Figure 1).

3 GRAPHREX QUERY INTERFACE

The goal of GraphRex is to provide a high-level interface with the performance of a system tuned for datacenters. To that end, GraphRex presents a Datalog-like interface and leverages an array of optimizations that reduce data and data transfer. We illustrate our variant of Datalog with several graph queries, most of which involve recursion:

Query 2: Number of Vertices (NV)

```
vnum(count<A>) :- e(A,B)
```

Query 3: PageRank (PR)

```
deg(A,count<B>) :- e(A,B)
pr(A, 1.0) :- deg(A,_)
pr(A,0.15+0.85*sum<PR/DEG>)[10] :- pr(B,PR),
     deg(B,DEG), e(B,A)
```

Query 4: Same Generation (SG)

```
sg(A,B) :- e(X,A), e(X,B), A!=B
sg(A,B) :- e(X,A), sg(X,Y), e(Y,B)
```

Query 5: Transitive Closure (TC)

```
tc(A,B) :- e(A,B)
tc(A,B) :- tc(A,C), e(C,B)
```

Query 2 counts the number of vertices in a graph (NV). It takes as input all edge tuples `e(A,B)` and does a count of all unique vertices `A`. Query 3 computes page ranks of all vertices in a graph (PR). Query 4 returns the set of all vertices that are at the same generation starting from a vertex (SG). Query 5 computes standard transitive closure (TC). The Datalog variant we use has similar syntax to traditional Datalog with

[1]Typical rack-level oversubscription ratios can range from 1:2 to 1:10 [15, 59]. Some public clouds strive for 1:1, but these are in the minority [62]. Regardless, as we show in Section 6, other datacenter practicalities can result in effects similar to oversubscription.

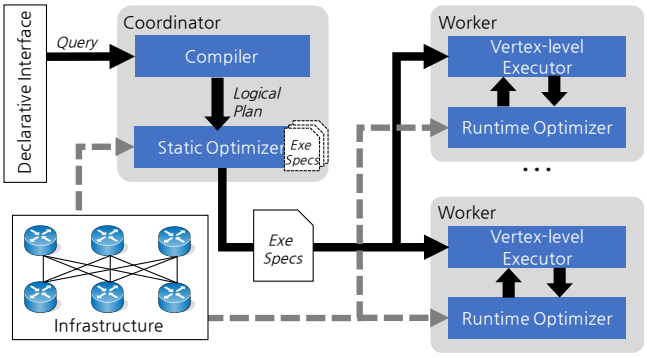


Figure 3: The GraphRex architecture. A compiler generates a logical plan from a Datalog query (4.1). The static optimizer then constructs from the logical plan a datacenter-centric execution specification (4.2) that is optimized (5) before the final translation to and evaluation of the physical plan by workers. Grey lines describe dissemination of infrastructure configurations and black lines communication for query execution.

aggregation, where aggregate constructs are represented as functions with variables in brackets (<>).

One extension we make to Datalog can be seen in PR: a stopping condition denoted as "[..]" in the rule head, for rules that may not converge to a fixpoint using traditional incremental evaluation of aggregates in recursive queries [27, 39, 60, 63]. For example, in PR, instead of stopping the query when no more new tuples are generated, we can impose a bound on the number of iterations, e.g., "[10]".

We also note that some of our queries involve multi-way joins. For example, SG is a "same generation" query that generates all pairs of vertices that are of the same distance from a given vertex (for example, given the root of a tree, SG generates a tuple for each pair of vertices which have the same depth. If the graph has cycles, a vertex can appear in different generations, significantly increasing query complexity). In existing distributed Datalog systems, the syntactic order is the sole determinant for the evaluation strategy of these joins—*they are simply evaluated "from left to right"* [57, 63]. This is because in a distributed environment, there is no global knowledge of relations and no easy way to find the optimal join order. As we will show later, this naïve order is suboptimal in many cases, and GraphRex improves on this by dynamically picking the best join order. Note that PR also has a multi-way join, but there is no need for picking join orders dynamically for this particular case, because the cardinalities of `pr`, `deg` and `e` never change in semi-naïve evaluation.

4 QUERY PLANNING

Figure 3 shows the overall architecture of GraphRex, consisting of a centralized coordinator and set of workers. The coordinator first applies a graph partitioning, so that each worker has a portion of the graph. Then during query execution, the coordinator's Query Compiler translates queries into a **logical plan**.

A Static Optimizer then generates an **execution specification** from that logical plan. Execution specifications are similar to physical plans, but include our datacenter-centric **global operators**. The final translation of these operators to concrete physical operators is left until runtime, and depends on both the placement of workers in the datacenter (which is obtained through an infrastructure configuration) and data characteristics. Each worker's physical plan may differ. We discuss this process in Section 5.

Finally, each worker runs the Distributed Semi-Naïve (GR-DSN) algorithm designed for very fine-grained execution, which is a distributed extension of the semi-naïve algorithm used in Datalog evaluation [43]. In semi-naïve evaluation (SN), tuples generated in each iteration are used as input in the next iteration until no new tuples are generated. The distributed variant relaxes the set operations by allowing for tuple-at-a-time pipelined execution. GR-DSN is designed for graph queries to allow massively parallel execution and tuple-level optimizations. We include its details in Appendix B.

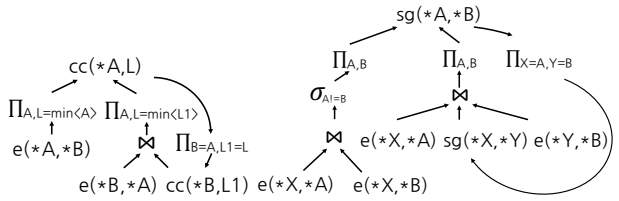
The above process occurs directly at the workers, which receive the execution specification, generate a local vertex-centric operator, and execute it, all with the help of two components: (1) a *Vertex-level Executor* that uses DSN to execute the specification until a fixpoint; and (2) a *Runtime Optimizer* that optimizes each global operator locally.

4.1 Logical Plan

From the query, the first step in processing it is to generate a logical plan. In GraphRex, a logical plan is a directed graph, where nodes represent relations or relational operators, and edges represent dataflow. Figures 4a and 4b show logical plans for Queries 1 (CC) and 4 (SG), respectively.

An important part of logical plan generation in GraphRex is a *Vertex Identification* phase, in which the compiler traverses the plan graph starting from the edge relations and marks attributes whose types are vertices with a `*` symbol. These attributes are candidates for being the partition key. As an example, in Figure 4a, since both attributes in the input edge relation `e(A,B)` represent vertices, they are both marked with the `*` symbol. Likewise, all attributes that have a dependency to either vertex attribute `A` or `B` are also marked.

By the time we generate a physical plan, only one partition attribute will be chosen for every relation. Later, we will denote the selected attribute by prepending with an `@` symbol. At this stage, we can make the decision for two simple cases. First, if a relation `r` only has one vertex attribute, then it is trivially partitioned by that attribute. Second, the edge table `e` is partitioned on the first key by default so that each



(a) Query 1 (CC) **(b) Query 4 (SG)**

Figure 4: Logical plans of CC (a) and SG (b).

vertex maintains the list of outgoing neighbors. This is a convenient placement for many practical graph applications, such as PageRank, SSSP, that only require each vertex to know its outgoing neighbors. All other partitioning decisions are made during the placement of the SHUFF and ROUT operators described in the following section.

4.2 Execution Specification

Traditional query planning proceeds directly from a logical plan to a physical plan. In GraphRex, we add an additional step to help identify opportunities for datacenter-centric optimization. The core of this process is the addition of **Global Operators** to the logical plan to form what we term an *execution specification*. These operators are special in that they govern communication across workers; oversubscription, capacity constraints, and congestion mean that their efficient execution is a primary bottleneck in processing large graphs. We describe each Global Operator below.

4.2.1 Join (JOIN)

Joins are one such operation that frequently incurs communication in graph processing. In Datalog, (natural) joins are expressed as conjunctive queries. GraphRex evaluates them as binary operations; multi-way joins are executed as a sequence of binary joins. Graphically, we represent these as:

In the case of binary joins, we simply insert a JOIN in lieu of the logical operator ⋈. Recursive joins, where one or more of the inputs are recursive predicates, are handled similarly to BigDatalog [57]. Namely, if the recursion is linear, the non-recursive inputs are loaded into a lookup table and streamed. If the recursion is non-linear, we load all but one of the recursive inputs into a lookup table and stream the remaining input. This enables us to reduce non-linear recursion to linear recursion from the viewpoint of a single new tuple. Figure 5 shows an example of a recursive join. Multi-way joins require additional handling, as different join orders can lead to drastically different evaluation costs (Section 5.4). In GraphRex, multi-way joins are implemented as a sequence of binary joins, where the order is chosen at *runtime* and *per-tuple*. Existing distributed Datalog systems arbitrarily evaluate 'left-to-right' [57, 63]. We represent this choice in the execution specification by enumerating all possible decompositions of the multi-way join and routing between them dynamically with the next operator.

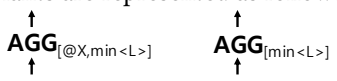
4.2.2 Routing (ROUT)

The ROUT operator enables the dynamic and tuple-level multi-way join ordering mentioned above. ROUTs take a tuple and direct it to one among multiple potential branches in the execution specification. This operator is only used in conjunction with multi-way joins, and is represented as:

For example, Figure 6 shows the specification for Query 4 (SG) where the multi-way join e ⋈ sg ⋈ e in Figure 4b is broken into (e ⋈ sg) ⋈ e and e ⋈ (sg ⋈ e). We generate plans for the two possible orderings and insert a ROUT operator that takes A and B as input to decide which will result in better performance. We discuss how the ROUT operator makes that decision in Section 5.4.

4.2.3 Aggregation (AGG)

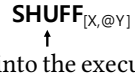
Another important Global Operator is AGG, which aggregates tuples. There are three types of aggregation in GraphRex, two of which are mapped to Global Operators. The one type of aggregation that is *not* mapped is purely local aggregation, which operates on tuples with the same partition key, for instance, in the left branch of Figure 5 (in the projection). This type of aggregation does not need its own Global Operator as its evaluation does not incur communication. The other two variants are represented as follows:



Left to right, (1) also operates at each vertex, but requires shuffling of inputs to compute the relation, and (2) covers global aggregation, where a single value is obtained across the entire graph. For (1), the semantics are similar to a purely local aggregation, but as communication is required, GraphRex will eventually rewrite the specification in order to reduce the data sent across the oversubscribed datacenter interconnect. The right branch of Figure 5 demonstrates this case. For (2), aggregation is instead finalized at the coordinator. For example, Query 2 (NV) computes the number of vertices in the graph using a global aggregator. That value is eventually collected by the coordinator and potentially redistributed to all workers for subsequent use.

4.2.4 Shuffle (SHUFF)

Last, but arguably most important is the SHUFF operator that encompasses all network communication in GraphRex.



SHUFFs are inserted into the execution specification whenever it is necessary to move tuples from one worker to another between relations. Their placement is therefore closely

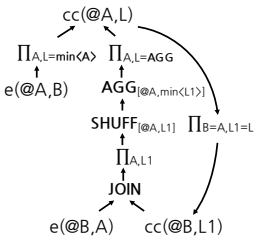


Figure 5: Execution specification of CC

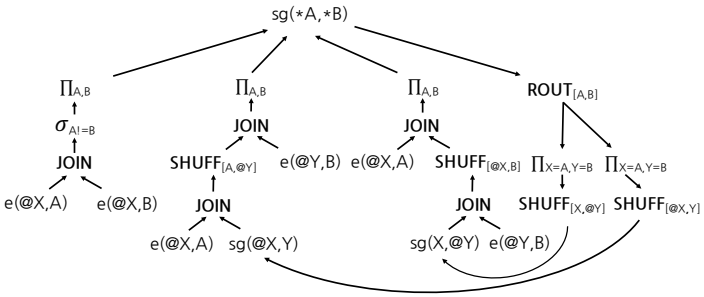


Figure 6: Execution specification of SG

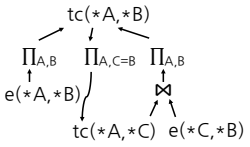
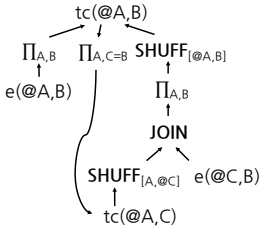
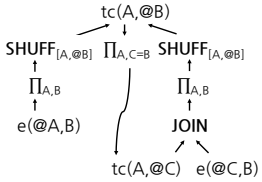


Figure 7: Logical plan of TC (Q5).



(a)



(b)

Figure 8: Two potential partitionings for TC.

integrated with the process of relation partitioning, which instantiates the partition attribute (`@`) from the set of partition candidates (`*`) and inserts SHUFF operators where necessary.

Conceptually, there are two scenarios that require a SHUFF. The first is when the tuples of relation `r` are not generated in the location specified by `r`'s partition key. An example of this is shown in Figure 5. The JOIN operation generates `cc` tuples that have a distinct partition key (denoted by the `@` sign) from the join key B. This results in the insertion of a SHUFF operator after the join. The second scenario is when the input relations to an operator are not partitioned on the same attribute, such as the inputs to the join operator in Figure 8a. In the example, there is a join operator for `tc` and `e` on attribute `C`. If we partition `tc` on its first attribute, as in Figure 8a, a SHUFF is needed to repartition the tuples in `tc` on the second attribute so that the join can be evaluated.

In relation partitioning, the optimizer checks every possible partitioning and selects the one that incurs the minimum number of SHUFFs. The details of partitioning algorithm are shown in Appendix A. As a heuristic, we assume that recursive rules are executed many times. To demonstrate this, Figure 8a shows the execution specification where `tc` is partitioned by the first key. The number of SHUFFs in the plan is $2K$, as there are two SHUFFs in each recursive rule evaluation. In comparison, the other partitioning of `tc` shown in Figure 8b requires fewer SHUFFs, i.e., $K + 1$; there is a single SHUFF for the non-recursive rule as well as one for each recursion. Our evaluation later shows that the latter plan provides a greater than 2× improvement.

5 GLOBAL OPERATOR OPTIMIZATIONS

Translation from the Global Operators described above depends on both context and the structure of the datacenter network. Refining these operators is important as they can incur significant performance costs in a large-scale datacenter deployment. We note that translation of the execution specification's *classic* logical operators into equivalent physical operators follows standard database plan generation, and we omit those details for brevity.

GraphRex introduces an array of synergistic optimizations (see Table 1), some of which can be used in combination, and some of which are intended as complements. Their benefits stem from a variety of reasons, but the overarching principle is to reduce data and data transfer, particularly across "expensive" links in the datacenter. Our results show that these techniques result in orders of magnitude better performance in typical datacenter environments.

5.1 Columnization and Compression

One important optimization in GraphRex applies to SHUFF. In SHUFF, tuples to be shuffled are stored in *message buffers*, which are then exchanged between workers. Rather than directly shuffle those buffers between workers, GraphRex (1) first sorts the data, (2) reorganizes (transposes) the tuples into a column-based structure, and (3) compresses the resulting data using the two techniques described below.

Although columnar databases are well-studied [5–7], their primary benefit in the literature has been in storage requirements. Performance benefits, on the other hand, are traditionally dependent on access patterns [33, 45]. GraphRex instead sends columnar data by default due to its benefits to two techniques—column unrolling and byte-level compression—that are particularly effective on typical graph workloads.

The first technique, column unrolling, is a process where we elide columns of known low cardinality, C, by creating C distinct columnar data stores—one for each unique value. For instance, in an adaptively ordered multi-way join, as described in Section 5.4, each intermediate tuple must carry with it an ID that denotes the join order and its place in that ordering of binary joins. In this and many other queries,

	Optimization	Description	Section
SHUFF	*Columnization & Compression*	Leverages workload characteristics to reduce the amount of data sent across the network on every SHUFF.	5.1
	Hierarchical Network Transfer	Further reduces the amount of data sent over 'expensive' links by applying columnization and compression hierarchically.	5.2
JOIN/ ROUT	*Join Deduplication*	To enforce distributed set semantics in JOINs, when a JOIN feeds into a SHUFF, we push deduplication into the SHUFF evaluation in a datacenter-centric manner.	5.3
	Adaptive Join Ordering	To account for power-law degree counts, we sometimes allow ROUT to dynamically chose a tuple-level join ordering. Only used when duplicates are uncommon.	5.4
AGG	*Hierarchical Global Aggregation*	Applies our datacenter-centric approach to global aggregation.	5.5
	On-path Aggregation	When SHUFF comes before a local AGG, we push the AGG down into the SHUFF to pre-aggregate values, again in a datacenter-centric fashion.	5.6

Table 1: GraphRex's Global Operator optimizations, when they apply, and where they are described.

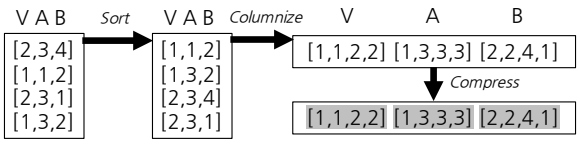


Figure 9: Column-based organization for `r(V,A,B)`, where `V` is the partition key. Shaded is compressed.

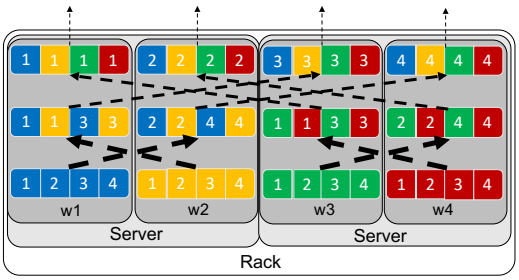


Figure 10: An example hierarchical transfer. Each worker groups their tuples by partition key, and sends the them first within a server, then within a rack, and finally to their destinations. A naïve system would send directly to other racks. Colors track where the tuple was generated; numbers indicate the partition.

column unrolling can all but remove the storage requirement of those columns.

The second technique, byte-level compression, compresses sorted and serialized streams using the Lempel-Ziv-class LZ4 lossless and streaming compression algorithm [4]. This process is shown in Figure 9. Both sorting and columnization significantly increase the similarity of adjacent data in typical graph applications, resulting in higher compression ratios. More optimal algorithms exist, but LZ4 is among the fastest in terms of both compression and decompression speed. To further reduce the overhead of this optimization, we only sort over the partition key (`V` in the example of Figure 9). We also limit compression to large messages, directly sending messages that are under certain size. As typical message sizes are bimodal, any reasonable threshold will provide a similarly effective reduction of overhead (in our infrastructure, a threshold of 128 bytes was robust).

Once the shuffle operation is finished, each worker decompresses, deserializes and transposes the received data to access the tuples. We store the tuples in row form for access and cache efficiency. We also heavily optimize memory copies, buffer reuse, and other aspects of serialization and deserialization, but omit the details for space. Applying columnization and compression together at a worker level brings ~2× overall message reduction for the CC query, however, its effectiveness in typical datacenters can be magnified by the next optimization we propose to SHUFF operator.

5.2 Hierarchical Network Transfer

GraphRex extends the benefits of the previous section by executing Hierarchical Network Transfers as part of SHUFF. This optimization reduces transfers over network, particularly the oversubscribed portions described in Section 2.2.

Figure 10 depicts this process for a rack with two servers and two workers per server. Specifically, transfers occur in three steps: *server-level shuffling*, *rack-level shuffling* and the final *global shuffling*. At each level, workers communicate with other workers in the same group, and split their tuples so that each partition key is assigned to a single worker in the group. At each step, tuples are efficiently decompressed, merge sorted, and re-compressed. The benefit of performing this iterative shuffling and compression is that, with every stage, the working sets of workers become increasingly homogenous and therefore more easily compressed.

To show the effect of this optimization, we present results for Query 1 (CC) on a billion-edge *Twitter* dataset running in a 40-server, 1:5 oversubscription testbed (more results are in Section 6). Table 2 shows the communication/total speedup

Worker	Worker Level	Server Level	Rack Level
$W1$	(1,2),(2,3),(3,4),(4,5)	(1,2),(3,4)	(1,2)
$W2$	(1,2),(2,3),(3,4),(4,5)	(2,3),(4,5)	(2,3)
$W3$	(1,2),(2,3),(3,4),(4,5)	(1,2),(3,4)	(3,4)
$W4$	(1,2),(2,3),(3,4),(4,5)	(2,3),(4,5)	(4,5)

Table 3: An example of Hierarchical Deduplication with a single rack of two servers, with two workers per server. At each successive layer of the hierarchy, workers coordinate to deduplicate join results before incurring increasingly expensive communication.

of two schemes: simple compression (directly on tuples) and SHUFF (column-based hierarchical compression).

They are compared against a baseline that does not implement compression or infrastructure-aware network transfer. Columnization combined with hierarchical network transfer creates more total traffic, but with less going over oversubscribed links and better load balancing (see Section E for an explanation). In this case, server-level shuffling reduces the data by 4.6×, and rack-level shuffling reduces the data by 6.2× in our datacenter testbed running 20 workers per server. Together with our optimizations on memory management and (de)serialization, SHUFF achieves a 9.8× speedup in communication time and 7.2× in total execution time.

	Comm	Total
Only compression	1.02×	1.02×
SHUFF	9.84×	7.2×

Table 2: Communication and total speedup of SHUFF and row-based compression in CC on *Twitter*.

5.3 Join Deduplication

JOINs are among the most expensive operations in large graph applications. One reason for this is the prevalence of high amounts of duplicate data in real-world distributed graph joins. For example, with Query 5 (TC) on a social graph, users may have many common friends and thus many potential paths to any other user.

In order to provide set-semantics for joins, previous systems perform a global deduplication on the generated tuples [57]. GraphRex instead introduces Hierarchical Deduplication, which takes advantage of datacenter-specific communication structures to decrease the cost of deduplication when it observes JOIN followed by a SHUFF. Note that when the results of a JOIN are used directly (without an intermediate SHUFF), local deduplication is sufficient.

To illustrate the process of Hierarchical Deduplication, consider again the deployment environment of Figure 10, where we have four workers in a single rack. Assume also that all four workers generate the same tuples {(1,2), (2,3), (3,4), (4,5)}, where the first attribute in the relation is the partition key. After the tuples are generated, workers insert them into a hash set that stores all tuples they have seen thus far. This results in the local state shown in the second column of Table 3. Workers on the same server then shuffle tuples among themselves, never traversing the network. The same is done at a rack level: servers deduplicate tuples without ever sending across the oversubscribed interconnect. In the end, of the 16 tuples generated in the rack, only 4 are sent to the other rack—a factor of 4 decrease in inter-rack communication. Queries on real-world graphs, e.g., social networks and web graphs, often exhibit even greater duplication because of dense connectivity: in the execution of TC over *Twitter*, for instance, 98.5% of generated `tc` tuples are duplicates.

	Dup %	Comm	Total
Baseline	98.5%	39.9 s	41.1 s
Hierarchical Dedup	42.7%	2.7 s (14.8×)	4.3 s (9.6×)

Table 4: Hierarchical Deduplication in TC on *Twitter*. Dup % indicates duplicate tuples received at workers

Table 4 presents the *Twitter*/TC result on the testbed used in the preceding section. We can see that, for workloads with many duplicates, hierarchical deduplication can efficiently remove most of them. In comparison, push-down techniques at worker level and server level can only reduce the duplication ratio to 96.3% and 90.7% respectively, which shows that deduplication should be performed at greater scale. The high deduplication rate of JOIN results in a 14.8× communication speedup and 9.6× total speedup. Even for workloads with few duplicates, the overhead of this optimization is low.

5.4 Adaptive Join Ordering

In the case of multi-way joins, GraphRex sometimes chooses a more aggressive optimization: Adaptive Join Ordering. To that end, the ROUT operator decides, per-tuple, of how to order the constituent binary joins of a multi-way join. A key challenge here is predicting the performance effects of choosing one order over another. One reason this can be difficult is due to duplicates; different join orders may result in tuples being generated on different workers, impacting the occurrence of duplicates in unpredictable ways for the current and future iterations.

For that reason, Adaptive Join Ordering is a complement to Join Deduplication: when the number of duplicates is high, the latter is effective, otherwise the optimization described here is a better choice. We rely on programmers to differentiate between the two when configuring the query. In practice, this is typically straightforward (and akin to the configuration of combiners in Hadoop/Spark), but profiling and sampling could automate the process in future work.

To illustrate a simple example of how join ordering can result in improved performance, consider the evaluation of

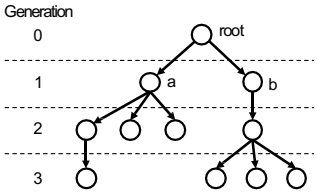


Figure 11: Query 4 (SG) on an example graph.

Query 4 (SG) over the graph in Figure 11. Starting at the root, vertices a and b are in the same generation, so a tuple (a, b) in sg is generated by the first rule. The evaluation of the second rule is decided by how sg is partitioned:

- If the relation is partitioned by the first attribute, then the join is evaluated from left to right ((e ⋈ sg) ⋈ e) where (a, b) is sent to a to join with Γ_a (the adjacency list of a) before the intermediate tuples are shuffled to b to finish the join.
- If partitioned by the second key, then the join ordering is from right to left (e ⋈ (sg ⋈ e)) where Γ_b is sent to a to finish the join, less cost than the first order.

For this iteration, the left-to-right ordering used by existing distributed Datalog systems results in a factor of three increase in intermediate tuples compared to right-to-left. The opposite is true for the third generation. Real-world graphs produce many such structural discrepancies due to their power-law distributions of vertex degree. This distribution can result in substantial performance discrepancies between different join orderings, even within a single relation. Thus, static ordering—any static ordering—can result in poor performance.

Optimization target. The goal of ROUT is as follows. Let T be the bag of tuples generated by GR-DSN query evaluation. T consists of tuples generated in every iteration, so we have $T = \sum_{k=0}^{K} T_k$ where T_k is the bag of tuples generated in iteration k and K is the iteration where a fixpoint is reached. ROUT's optimization objective is:

$$\min |T| = \min \sum_{k=0}^{K} |T_k|$$

Intuitively, more tuples mean increased cost of tuple generation and shuffling. More formally, let T_k^{α} be the bag of intermediate tuples—those that are generated in the intermediate binary joins in order to complete the multi-way join—and T_k^{β} be the bag of output tuples of the head relation (for example, sg in SG), so $T_k = T_k^{\alpha} + T_k^{\beta}$, and we have:

$$\min |T| = \min \sum_{k=0}^{K} (|T_k^{\alpha}| + |T_k^{\beta}|)$$

As mentioned previously, GraphRex makes an assumption that there are no duplicates in generated tuples. Formally, this simplifies optimization in two ways. First, if there are no duplicates, any ordering generates the same T_k^{β} (because of the commutativity and associativity of natural joins) so $|T_k^{\beta}|$ becomes a constant. Second, the ordering of one iteration does not affect another. This independence allows us to optimize each iteration without worrying about later ones. With this assumption, we now have

$$\min |T| = \sum_{k=0}^{K} \min(|T_k^{\alpha}|) + C \tag{1}$$

where C is a constant representing the number of output tuples generated in the evaluation.

Ordering joins. With the above, GraphRex picks a tuple-level optimal ordering using a precomputed index.

For every newly generated tuple that goes through the ROUT operator, GraphRex enumerates all possible join orders, computes the cost (i.e., the number of tuples in T_k^{α} that the order generates) for each order, and selects the order with the minimum cost. Then, GraphRex sets the partition key of this tuple based on the join order, and sends it to the destination for join evaluation. For example, in SG, for every new sg tuple (a, b), there are two possible join orders: ((e ⋈ sg) ⋈ e) and (e ⋈ (sg ⋈ e)). The cost for the first order is the degree of a because (a, b) is sent to a first for the first binary join and then Γ_a is sent to b for the second binary join. Similarly, the cost for the second order is the degree of b. The degrees of all vertices are precomputed as an index, and thus efficiently accessible at runtime.

Generality. For n-way joins, the possible options grow to $\binom{n-1}{i-1}$, where i is the position of the recursive predicate, e.g., e ⋈ sg ⋈ e is a 3-way join with sg in position 2. Note that the recursive predicate in position 0 or n leads to only 1 ordering. GraphRex scales efficiently by preloading necessary information as indexes whose total size grows as $O(n|V|)$. Regardless, typical values of n are small and there are only a small number of possible orders. See Appendix C for details.

	1st	2nd	3rd	4th
% of LR	77.47%	80.64%	87.65%	88.16%
% of RL	22.53%	19.36%	12.35%	11.84%

Table 5: The percentage of tuples using each join order during the first four iterations of SG on *SynTw*. LR is (e ⋈ sg) ⋈ e and RL is e ⋈ (sg ⋈ e).

Table 5 shows the percentages of tuples in the optimal query plan of the first four iterations of SG on *SynTw*, a synthetic graph of Twitter follower behavior (see Section 6 for more information). For most tuples, LR ordering is optimal, but for a non-negligible fraction, it is not. Because of

this variability, Table 6 shows that, compared to static ordering, Adaptive Join Ordering brings 2.7× and 2× speedup to communication and execution time respectively.

	Comm	Total
Static ordering	3.4 s	9.3 s
Adaptive Join Ordering	1.3 s (2.7×)	4.6 s (2×)

Table 6: Comparison of adaptive and static ordering.

5.5 Hierarchical Global Aggregation

As mentioned in Section 4.2, there are three types of aggregations, two of which are translated to Global Operators. This section describes our optimizations for the global AGG, which is used to compute and disseminate a single global value to all workers via the coordinator. A naïve implementation would create a significant bottleneck at the coordinator. A classic alternative is parallel aggregation, in which workers aggregate among themselves in small groups, then aggregate the sub-aggregates, and so on. GraphRex improves this by leveraging knowledge of datacenter network hierarchies.

Figure 12 shows an example of this process. First, each worker applies the aggregate function on its vertices and computes a partial aggregated value, then it sends its partial value to a designated aggregation master in the server. When the server master receives partial values from all workers in the same server, it again applies the aggregate function to update its partial value and then it sends the value to the rack master, which updates its own partial value and finally sends that value to the global aggregation coordinator.

As in previous instances, hierarchical transmission significantly reduces traffic over the oversubscribed network. As the computations and communications of Hierarchical Global Aggregation are distributed at each network hierarchy, the overhead to the aggregation coordinator is also reduced. Table 7 shows the performance of Hierarchical Global Aggregation in the query of counting vertex number (NV) on *Twitter*. The baseline is infrastructure-agnostic, which means the global aggregation is implemented in an AllReduce manner where all workers send their partial aggregated values to the coordinator. Hierarchical Global Aggregation results in 41× speedup in communication and reduces query processing latency from 2.26 s to 0.16 s.

	Comm	Total
Baseline	2.154 s	2.26 s
Hier. Glob. Agg.	0.052 s (41.4×)	0.158 s (14.3×)

Table 7: Evaluation of Query 2 (NV) on *Twitter*.

5.6 On-path Aggregation

Finally, the other AGG operator computes a value for each vertex, but requires a SHUFF first. In this case, GraphRex pushes AGG down into SHUFF so that every worker only sends aggregated tuples. The key insight is that tuples that are shuffled to the same vertex can be pre-aggregated. On-path Aggregation again leverages hierarchical shuffling: at each level in the network, it consolidates the tuples for the same vertices to efficiently and incrementally apply aggregation and reduce the number of shuffled tuples.

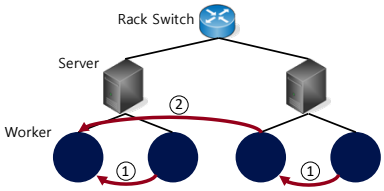


Figure 12: Hierarchical Global Aggregation in a rack. After worker-level aggregation, intermediate aggregates are shuffled (1) at a server-level, and (2) at a rack-level before finishing global aggregation.

Table 8 shows the performance of On-path Aggregation in CC on *Twitter*, where the baseline is aggregation at the destination, which means that all tuples are shuffled through the network first, and then aggregated using (*min*). On-path Aggregation brings 10× speedup in the communication, and the end-to-end query processing latency is reduced by 7.8×.

	Comm	Total
Baseline	119.8 s	124.29 s
On-path Aggregation	11.997 s (10×)	15.97 s (7.8×)

Table 8: Evaluation of Query 1 (CC) on *Twitter*.

6 EVALUATION

In this section, we evaluate the performance of GraphRex with a representative set of real-world graph datasets and queries in order to answer three high-level questions:

- *How competitive is the performance of GraphRex?* We compare GraphRex with BigDatalog [3], which is shown to outperform other distributed declarative graph processing systems (such as Myria [63] and SociaLite [56]), Giraph [1], and PowerGraph [29], two highly-optimized distributed graph processing systems.
- *How robust is GraphRex to datacenter network dynamics?* We emulate typical network events that affect the connectivity between servers, vary network capacity, inject background traffic following typical traffic patterns in datacenters, and test systems under such dynamics.
- *How scalable is GraphRex?* We evaluate how GraphRex scales with additional resources. We also perform a COST [50] analysis to compare it with optimized, single-threaded implementations, and examine scale-up/out performance for large-scale graph processing.

Due to space constraints, we have included more experiments in the Appendix, including more results (App. F) and the analysis of communication patterns in GraphRex (App. E).

6.1 Methodology

Setup. Our CloudLab datacenter testbed consists of two racks and 20 servers per-rack. Each server has two 10-core Intel E5-2660 2.60GHz CPUs with 40 hardware threads, 160 GB of DDR4 memory, and a 10 Gb/s NIC. In aggregate, the testbed has 6.4 TB memory and 1.6 k CPU threads. Mirroring modern datacenter designs [15, 31, 59], our testbed is connected using a 10 Gb/s leaf-spine network [11] with four spine switches by default, resulting in an over-subscription ratio of 1:5.

Queries. We have selected a set of representative queries to evaluate GraphRex. *General Graph Queries* include Connected Components (CC, Q1), PageRank (PR, Q3), Single Source Transitive Closure (TC, Q5), Single Source Shortest Path (SSSP, Q6), and Reachability (REACH, Q7). Among those queries, CC and PR are compute-intensive and TC, SSSP and REACH are more communication-intensive. We also evaluated local and global *Aggregation* queries (CM, Q8) (*sum* and *min* aggregators produced similar results) as well as *Multi-way Join* queries like Same Generation (SG, Q4).

Query 6: SSSP (SSSP)

```
sssp($ID,0) :- e($ID,_,_)
sssp(A,min<C1+C2>) :- sssp(B,C1), e(B,A,C2)
```

Query 7: Reachability (REACH)

```
reach($ID) :- e($ID,_)
reach(A) :- reach(B), e(B,A)
```

Query 8: CountMax (CM)

```
inout(A,count<B>) :- e(A,$ID), e($ID,B)
maxcount(max<CNT>) :- inout(_,CNT)
```

Datasets. As shown in Table 9, we have selected four real-world graph datasets, all of which contain billions of edges. *Twitter* and *Friendster* are social network graphs, and *UK2007* and *ClueWeb* are web graphs.

Graph	# Vertices	# Edges	Data Size
Twitter (TW)	52.6 M	2 B	12 GB
Friendster (FR)	65.6 M	3.6 B	31 GB
UK2007 (UK)	105.9 M	3.7 B	33 GB
ClueWeb (CW)	978.4 M	42.6 B	406 GB

Table 9: Large graphs in the evaluation.

System configurations. We compare against the latest versions of in-comparison systems, and configured them to achieve the best performance in our datacenter. We provisioned them with sufficient cores and memory and optimized other parameters, such as the number of shuffle partitions in BigDatalog, the number of containers in Giraph, and partition strategies in PowerGraph. When possible, we used the query implementations provided by these systems, and implemented the remainder from scratch. Not all systems were able to support all queries easily/efficiently; we omit those as needed. BigDatalog, for instance, has difficulty supporting *PageRank* because it cannot limit the number of iterations. The original paper [57] also omits PR. Similarly, PowerGraph cannot easily support SG, because a) vertex adjacency lists are not readily accessible, and b) it forces message consolidation, which would be very inefficient for SG.

6.2 System Performance

We first evaluate the performance of GraphRex against state-of-the-art systems in terms of query processing times.

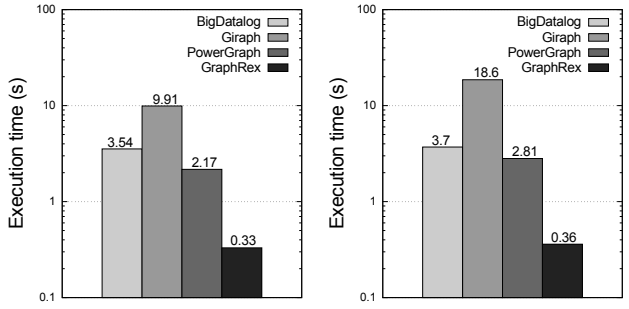
General graph queries. Table 10 compares the overall performance of GraphRex, BigDatalog, PowerGraph, and Giraph across different graphs and queries. CC and PR require more computation than other queries. Even in these cases, the oversubscribed network is enough of a bottleneck that GraphRex outperforms other systems by up to an order of magnitude. Against BigDatalog and CC, this order of magnitude improvement is consistent. PowerGraph and Giraph, due to their specialization to graph processing, perform better than BigDatalog, but they are still significantly slower than GraphRex, if they complete (between 3.2× and 17.3×). We note that the largest graph, *CW*, caused out-of-memory issues on both BigDatalog and Giraph; our deduplication and compression alleviate some issues with working set size.

On more communication-intensive queries, i.e., TC, SSSP and REACH, GraphRex achieves even greater speedups. On on these too, BigDatalog failed to complete on the largest graph, *CW*. For TC, GraphRex outperforms BigDatalog and Giraph by up to two orders of magnitude, and PowerGraph by more than 4× on average. Some of this stems from GraphRex's automatic relation partitioning (Section 4.2.4). BigDatalog, by default, partitions by the first key, which happens to be a poor choice in this case. Manually partitioning by the second key leads to 2× better performance, but this is still much slower than GraphRex as it lacks our other optimizations. For SSSP (results in Figure 1), GraphRex outperforms BigDatalog by 28–54× on the workloads BigDatalog could complete, and outforms PowerGraph and Giraph by an average of more than 5× and 10×. Finally, for REACH, GraphRex achieves up to 45.6× higher performance than BigDatalog and up to 26.2× speedup over PowerGraph and Giraph.

Aggregation queries. Figure 13 shows the results of an aggregation, Query 8 (CM), on *TW* and *FR*. Since we have found similar results on *UK*, and BigDatalog cannot handle *CW*, we have omitted these results. Here, BigDatalog performs better than Giraph, achieving 2.8× and 5× better performance on

		CC				PR				TC				REACH			
		G.R.	B.D.	Giraph	P.G.	G.R.	B.D.	Giraph	P.G.	G.R.	B.D.	Giraph	P.G.	G.R.	B.D.	Giraph	P.G.
TW	Time	10.3s	119.8s	49.1s	35.6s	13.4s	-	68.6s	43.2s	3.1s	336.8s	50.8s	11.8s	2.8s	90s	26.7s	11.5s
	SpdUp		**11.6×**	**4.7×**	**3.4×**		**N/A**	**5.1×**	**3.2×**		**109.4×**	**16.5×**	**3.8×**		**32×**	**9.5×**	**4.1×**
FR	Time	15.3s	278.6s	79.3s	60.5s	18.5s	-	148.7s	60s	5.1s	898.5s	81.8s	20.4s	5.2s	236.1s	49.01s	20.7s
	SpdUp		**18.2×**	**5.2×**	**4.0×**		**N/A**	**8.1×**	**3.2×**		**176×**	**16×**	**4×**		**45.6×**	**9.5×**	**3.99×**
UK	Time	30.9s	452.8s	274.4s	164.6s	9.6s	-	149.9s	73.6s	18.5s	866.3s	192.1s	86.1s	17.6s	361.02s	152.6s	87.1s
	SpdUp		**14.7×**	**8.9×**	**5.3×**		**N/A**	**15.6×**	**7.7×**		**46.9×**	**10.4×**	**4.7×**		**20.5×**	**8.7×**	**4.9×**
CW	Time	472.6s	OOM	8159.5s	1808s	188.7s	-	OOM	668.8s	207.4s	OOM	5395.2s	978.7s	187.1s	OOM	4909.7s	969.2s
	SpdUp		**N/A**	**17.3×**	**3.8×**		**N/A**	**N/A**	**3.5×**		**N/A**	**26×**	**4.7×**		**N/A**	**26.2×**	**5.2×**

Table 10: Execution time and speedup for GraphRex (G.R.) compared to BigDatalog (B.D.), Giraph and PowerGraph (P.G.). We present results for four queries (CC, PR, TC, and REACH) (Figure 1 shows results for SSSP), and four graph datasets (TW, FR, UK, CW). OOM indicates an out-of-memory error. Note that B.D. does not support PR.



(a) *Twitter* **(b) *Friendster***

Figure 13: Aggregation query evaluation with CM.

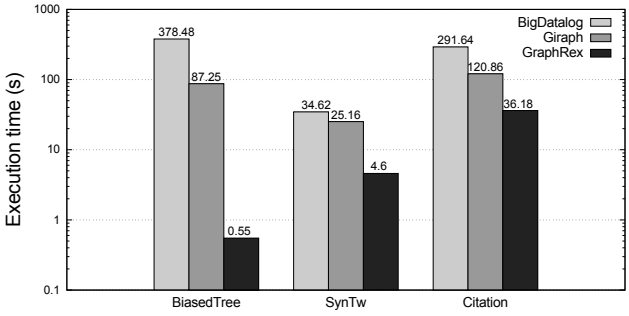


Figure 14: Multi-way join query evaluation with SG.

TW and *FR*, respectively, similar to PowerGraph. GraphRex is almost an order of magnitude faster than all of them as a result of our AGG Global Operator optimizations (Sections 5.5 and 5.6) and their ability to avoid traversal of the oversubscribed network. GraphRex finishes within one second.

Multi-way join queries. Multi-way joins are challenging even on small social network and web graphs. Consider SG as an example: since such graphs are well-connected, all vertices will eventually be at the same generation. This would result in an output size of $|V|^2$, where $|V|$ is the number of vertices; so a small graph with 1 M vertices would result in 1 T sg tuples. Therefore, we have used three alternative datasets to evaluate SG: (1) *BiasedTree*, which amplifies the imbalance in Figure 11 by setting the degree of the high-degree vertices to 10K and increasing the depth of the tree to 10, (2) *SynTw*, a synthesized graph simulating follower behavior in Twitter but without cycles, and (3) *Citation*, which is a real-world graph of paper citation relationships that we collected from public sources. While numbers of edges are relatively small (0.1 M, 35.7 M, and 20.4 M, respectively), the generated tuple sets are large: 1 B, 70 M, 6 B tuples during the evaluation of SG when using the best static join order.

Figure 14 shows our results (PowerGraph is omitted as noted earlier). For fairness, we ensured that Giraph and BigDatalog used the best static join order for the query. Even so, GraphRex significantly outperforms both. Adaptive Join Ordering, by picking the most efficient join ordering for every tuple, reduces the number of generated tuples to 0.2 M, 17 M, and 3 B. The resulting performance improvement is 3.3× in the worst case, with an upper bound of 2–3 orders of magnitude in the extreme case (*BiasedTree*).

Summary: This set of experiments shows that, as a declarative system, GraphRex consistently and significantly outperforms existing systems—both declarative and low-level—particularly on large-scale graph workloads.

6.3 Robustness to Network Dynamics

We next evaluate the robustness of GraphRex to network dynamics, which are common in datacenter networks.

Network degradation. One such class is link degradations, where the link capabilities can experience a sudden drop due to gray failures, faulty connections, or hardware issues [28, 68]. To emulate this, we randomly select a single rack switch uplink and throttle its capacity to $1/10$, $1/50$ and $1/100$ of its original capacity. Note that a degradation of a single server's access link would decrease performance for all systems equally. We deploy five systems and test their performance with CC on *Twitter* (results are similar for other

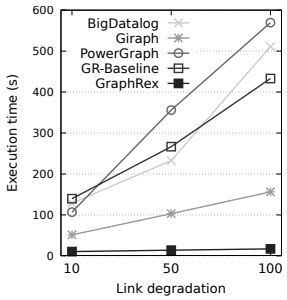


Figure 15: System performance under varying link degradations.

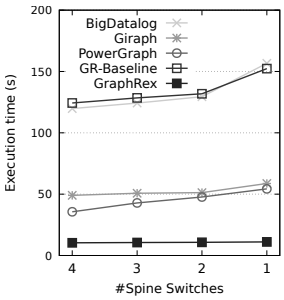


Figure 16: System performance with varying #aggregation switches.

graphs and queries): GraphRex, BigDatalog, Giraph, PowerGraph, and 'GR-Baseline', a version of GraphRex with Global Operator optimizations disabled.

Figure 15 shows performance under different degrees of link degradation. Because GraphRex minimizes traffic sent through bottleneck links, it is by far the most robust to degradations of those links. In fact, a 1/10 degradation shows almost no effect at all (10.61 s vs 10.3 s); even in the 1/100 case, GraphRex finishes in 17.24 s. In comparison, GraphRex-baseline experiences significant delay, taking 140 s in the 1/10 case, and 433 s in the 1/100 case. Among all systems, PowerGraph is most sensitive to network dynamics (16× slower than normal for the 1/100 case. Other systems are also severely impacted.

Over-subscription variation. We next evaluate the effect of over-subscription. We emulate this by adding/removing spine switches from the testbed. Less spine switches means less inter-rack capacity and greater over-subscription. Due to hardware constraints, we only vary the number of switches in the spine layer from 4 to 1.

Figure 16 shows results for CC on *TW*. Results for other graphs are included in Appendix F.1. The over-subscription significantly degrades the performance of other systems: PowerGraph performance drops 52% (36 s to 54 s) between 4 and 1 spine switches, BigDatalog drops 31% (120 s to 157 s), Giraph 20% (49 s to 59 s), and GR-Baseline 23% (124 s to 152 s). For reasons similar to the prior section, GraphRex's performance only changes 7% (10.3s to 11.1s) over the same range.

Background traffic. Finally, since datacenters typically host multiple applications, applications often experience unpredictable "noise" in the network in the form of background traffic. To evaluate GraphRex and the other systems in its presence, we inject background traffic using a commonly used datacenter traffic pattern [13, 14, 37]. Following the existing methodology, we generate traffic flows from random sender/receiver pairs, with flow sizes and flow arrival times governed by the real-world datacenter workloads [14]. Overall, we generated five representative traffic traces, each with an average network utilization of 40%. Details of the

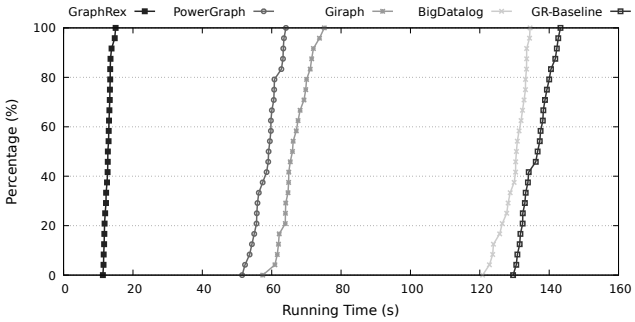


Figure 17: The CDF of performance with random background traffic. Each dot represents a complete run.

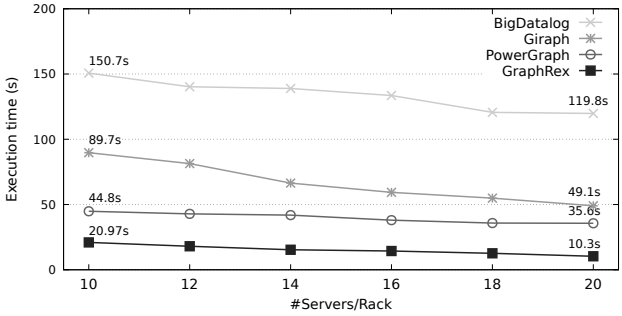


Figure 18: Scalability with the number of servers.

	G.R.	B.D.	Giraph	P.G.
Running Time	19.95s	142.31s	86.09s	42.33s
Two-rack Speedup	1.93×	1.18×	1.76×	1.2×

Table 11: The performance in single rack.

generated traces are included in Appendix D. We ran CC on *TW* in each system with background traffic, and note that other query workloads have similar findings. As Figure 17 shows, the performance variation is significant for other systems, with standard deviations (σ) of 3.6 (P.G.), 4.3 (Giraph), 3.9 (B.D.) and 4.2 (GR-Baseline). GraphRex, on the other hand, achieves $\sigma = 0.96$, which is much more robust, and its performance is significantly better than other systems, with average speedups of 4.6× (over P.G.), 5.2× (over Giraph), 10.1× (over B.D.), and 10.6× (over the baseline).

Summary: The datacenter-centric design in GraphRex increases robustness to network dynamics, even in harsh network conditions with significant link degradation, over-subscription, and random background noise.

6.4 Scalability Analysis

Finally, we evaluate scalability compared to other systems.

Number of servers/racks. We first examine how adding servers to the job affects performance. Specifically, we vary the number of servers per rack in our two-rack testbed from 10 to 20 with a step of 2. Figure 18 shows the result of running

	TW(S)	TW(D)	FR(D)	UK(D)	CW(D)
EmptyHeaded	**57.5s**	N/A	N/A	N/A	N/A
Timely	65.6s	25.96s	44.5s	23.5s	464.9s
GraphRex	99.3s	**13.4s**	**18.5s**	**9.6s**	**188.7s**

Table 12: Scale-up and out performance comparison. S stands for single-machine and D for distributed. EmptyHeaded ran OOM on graphs larger than *Twitter*.

CC on *TW*. For all systems, the running times decrease when more servers are added. However, more servers per rack also leads to higher oversubscription, which poses scalability bottlenecks. As a result, BigDatalog and PowerGraph only achieve around 1.3× speedup when we double the number of servers; Giraph achieves a 1.8× speedup, yet it still has lower performance than PowerGraph. In contrast, GraphRex, in our representative datacenter configuration, scales almost linearly: 2× speedup when server count doubles.

We saw a similar result when scaling up the number of 20-machine racks from one to two, as shown in Table 11. Here too, doubling the number of racks almost doubled GraphRex performance. Giraph also scaled well achieving 1.76× speedup, but the other systems did not. Appendix F.2 includes results for other workloads.

COST and scale-up/out analysis. For reference, we also compared our system to centralized alternatives, though we note that GraphRex is not optimized for single-machine execution. One comparison is a COST analysis [50]. We found that, compared to a COST of hundreds or thousands for many other systems [50], GraphRex only has a COST metric of 8 for CC and 10 for PR, meaning that it requires 10 threads to beat an optimized single-threaded implementation in [48].

We also present results with EmptyHeaded [8], the state-of-the-art single-machine system for Datalog-based graph processing, and Timely Dataflow [49] (Timely, hereafter), a distributed dataflow system optimized for both high throughput and low latency. Table 12 shows results for PageRank, the most accessible benchmark for both systems, on all four big graphs. In single machine experiments, *TW* is the only graph that can be processed on a single server with 160 GB memory. EmptyHeaded has the best performance, showing the efficiency of its single-machine optimizations, while Timely is only 14% slower. Although GraphRex is slower than both EmptyHeaded (1.7×) and Timely (1.5×) in single machine, its performance in distributed execution is much faster than Emptyheaded (4.3×[2]) and also faster than distributed Timely (1.9×). GraphRex outperforms Timely by 2.4×, 2.4× and 2.5× on *FR*, *UK* and *CW*, respectively. EmptyHeaded ran out of memory in the three larger graphs, demonstrating the need for scale-out to process large graphs. Appendix F.3 contains the evaluation of performance stability with Timely under different network conditions.

[2]Only for illustration as they are not using the same hardware resources.

7 RELATED WORK

Many graph processing systems have been proposed, including Pregel [46], Giraph [21], GraphX [30], PowerGraph [29], GPS [55], Pregelix [20], GraphChi [38], and Chaos [53]. GraphRex adopts a Datalog-like interface and computation model in order to explore the space of optimizations for large graph queries running on modern datacenter infrastructure.

Declarative data analytics: SociaLite [39] and Emptyheaded [8] are Datalog systems optimized for a single-machine setting. Hive [2] and SparkSQL [16] are distributed, but only accept SQL queries without recursion. BigDatalog [57] and Datalography [51] explore an intermediate design point (Datalog compiled to SparkSQL and GiraphUC); however, they ignore infrastructure-level optimizations and can be worse than the systems they are built on. GraphRex instead leverages Datalog for graph-specific and datacenter-centric optimizations, and outperforms existing systems significantly.

Performance optimizations. Several existing proposals [10, 22, 25, 32, 34] have explored the network-level optimization of groups of related network traffic flows, e.g., in a shuffle operation. GraphRex is distinguished by its deep level of integration with the Datalog execution model and its optimizations for graph workloads. Recent work has also proposed techniques to either partition input graphs [47, 66] or execute in a subgraph-centric fashion [61, 64] to minimize communication. Our optimizations at the infrastructure and query processing layers are orthogonal to, and could potentially benefit from, optimizations from these layers.

Graph compression and deduplication. Recent work has used data compression on graphs. Blandford et al. [18, 19] propose techniques to compactly represent graphs. Ligra+ [58] further parallelizes these techniques. GBASE [36] and SLASHBURN [41] perform compression for MapReduce to reduce storage. Succinct [9] can directly process compressed data. GraphRex is mostly related to C-Store [5], a column-oriented database, and we have further proposed novel techniques like the compressed transpose data structure.

Prior work has also explored deduplication, e.g., via MapReduce combiners [26, 65] and mechanisms for distributed set semantics [23, 57]. Our system pursues the same goals, but our key contribution is to adapt these techniques to create datacenter-centric optimizations for relational operators.

8 CONCLUSION

GraphRex is a framework that supports declarative graph queries by translating them to low-level datacenter-centric implementations. At its core, GraphRex identifies a set of

global operators (SHUFF, JOIN/ROUT, and AGG) that account for a significant portion of typical graph queries, and then heavily optimizes them based on the underlying datacenter, using techniques such as hierarchical deduplication, aggregation, data compression, and dynamic join orders. With a comprehensive evaluation, we demonstrate that GraphRex works efficiently over large graphs and outperform state-of-the-art systems by orders of magnitude. Generalizing our techniques to not rely on graph-specific properties (e.g., the ability to preload join cardinalities for Adaptive Join Ordering) is left to future work.

REFERENCES

[1] Apache Giraph. http://giraph.apache.org/.

[2] Apache Hive. https://hive.apache.org/.

[3] BigDatalog. https://github.com/ashkapsky/BigDatalog.

[4] Lz4 - extremely fast compression. http://lz4.github.io/lz4/.

[5] D. J. Abadi, S. R. Madden, and M. C. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, 2006.

[6] D. J. Abadi, S. R. Madden, and N. Hachem. Column-stores vs. row-stores: How different are they really? In *Proc. SIGMOD*, 2008.

[7] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. R. Madden. Materialization strategies in a column-oriented DBMS. In *Proc. ICDE*, 2007.

[8] C. R. Aberger, S. Tu, K. Olukotun, and C. Ré. EmptyHeaded: A relational engine for graph processing. In *SIGMOD '16*.

[9] R. Agarwal, A. Khandelwal, and I. Stoica. Succinct: Enabling queries on compressed data. In *Proc. NSDI*, 2015.

[10] F. Ahmad and et al. ShuffleWatcher: Shuffle-aware scheduling in multi-tenant mapreduce clusters. In *USENIX ATC '14*, 2014.

[11] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. SIGCOMM '08. ACM, 2008.

[12] O. Alipourfard and et al. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. NSDI'17, 2017.

[13] M. Alizadeh, T. Edsall, and et al. CONGA: Distributed congestion-aware load balancing for datacenters. In *Proc. SIGCOMM*, 2014.

[14] M. Alizadeh and et al. Data center TCP (DCTCP). In *SIGCOMM*, 2010.

[15] A. Andreyev. Introducing data center fabric, the next-generation facebook data center network. https://goo.gl/rE8wkL, 2014. Facebook.

[16] M. Armbrust and et al. Spark SQL: relational data processing in spark. In *SIGMOD '15*, 2015.

[17] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. 2000.

[18] D. K. Blandford, G. E. Blelloch, and I. A. Kash. Compact representations of separable graphs. In *Proc. SODA*, 2003.

[19] D. K. Blandford, G. E. Blelloch, and I. A. Kash. An experimental analysis of a compact graph representation. In *Proc. ALENEX*, 2004.

[20] Y. Bu and et al. Pregelix: Big(ger) graph analytics on a dataflow engine. *PVLDB*, 2014.

[21] A. Ching and et al. One trillion edges: Graph processing at facebook-scale. *PVLDB*, 2015.

[22] M. Chowdhury and et al. Managing data transfers in computer clusters with Orchestra. SIGCOMM '11. ACM, 2011.

[23] X. Chu, I. F. Ilyas, and P. Koutris. Distributed data deduplication. *PVLDB*, 2016.

[24] Cisco Systems. *Data Center Design Summary*, August 2014. https://www.cisco.com/c/dam/en/us/td/docs/solutions/CVD/Aug2014/DataCenterDesignSummary-AUG14.pdf.

[25] P. Costa and et al. Camdoop: Exploiting in-network aggregation for big data applications. In *NSDI '12*. USENIX, 2012.

[26] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI'04*, San Francisco, CA, 2004.

[27] S. Ganguly, S. Greco, and C. Zaniolo. Extrema predicates in deductive databases. *J. Comput. Syst. Sci.*, 51(2):244–259, 1995.

[28] P. Gill and et al. Understanding network failures in data centers: measurement, analysis, and implications. In *SIGCOMM*, 2011.

[29] J. E. Gonzalez and et al. PowerGraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.

[30] J. E. Gonzalez and et al. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*, 2014.

[31] A. Greenberg and et al. VL2: A scalable and flexible data center network. *SIGCOMM Comput. Commun. Rev.*, 2009.

[32] K. Hasanov and A. L. Lastovetsky. Hierarchical optimization of MPI reduce algorithms. In *PaCT 2015*.

[33] Intel. Optimize data structures and memory access patterns to improve data locality. https://goo.gl/xQ3ZGT, 2012. Intel.

[34] V. Jalaparti and et al. Network-aware scheduling for data-parallel jobs: Plan when you can. SIGCOMM '15, 2015.

[35] M. Kabiljo and et al. A comparison of state-of-the-art graph processing systems. https://code.facebook.com/posts/319004238457019/a-comparison-of-state-of-the-art-graph-processing-systems/.

[36] U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos. GBASE: An efficient analysis platform for large graphs. In *Proc. VLDB*, 2012.

[37] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford. HULA: Scalable load balancing using programmable data planes. In *Proc. SOSR*, 2016.

[38] A. Kyrola, G. E. Blelloch, and C. Guestrin. GraphChi: Large-scale graph computation on just a PC. In *OSDI 2012*, pages 31–46, 2012.

[39] M. S. Lam, S. Guo, and J. Seo. Socialite: Datalog extensions for efficient social network analysis. In *ICDE '13*, 2013.

[40] C. E. Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE Trans. Comput.*, 34:892–901, October 1985.

[41] Y. Lim, U. Kang, and C. Faloutsos. Slashburn: Graph compression and mining beyond caveman communities. *IEEE TKDE*, 2014.

[42] V. Liu and et al. Subways: A case for redundant, inexpensive data center edge links. CoNEXT '15. ACM, 2015.

[43] B. T. Loo and et al. Declarative networking: language, execution and optimization. In *SIGMOD '06*.

[44] Y. Low and et al. GraphLab: A new framework for parallel machine learning. UAI'10. AUAI Press, 2010.

[45] Z. Majo and T. R. Gross. Matching memory access patterns and data placement for NUMA systems. In *Proc. CGO*, 2012.

[46] G. Malewicz and et al. Pregel: a system for large-scale graph processing. In *SIGMOD '10*.

[47] D. W. Margo and et al. A scalable distributed graph partitioner. *PVLDB*, 8(12):1478–1489, 2015.

[48] F. McSherry. Cost. https://github.com/frankmcsherry/COST.

[49] F. McSherry. Timely dataflow. https://github.com/frankmcsherry/timely-dataflow.

[50] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what COST? In *HotOS '15*, 2015.

[51] W. E. Moustafa and et al. Datalography: Scaling datalog graph analytics on graph processing systems. In *IEEE BigData 2016*.

[52] Open Compute Project. *Server/SpecsAndDesigns*, June 2018. http://www.opencompute.org/wiki/Server/SpecsAndDesigns.

[53] A. Roy and et al. Chaos: scale-out graph processing from secondary storage. In *SOSP 2015*, 2015.

[54] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network's (datacenter) network. In *SIGCOMM '15*, 2015.

[55] S. Salihoglu and et al. GPS: a graph processing system. In *SSDBM '13*.

[56] J. Seo, J. Park, J. Shin, and M. S. Lam. Distributed Socialite: A Datalog-based language for large-scale graph analysis. *PVLDB*, 2013.

[57] A. Shkapsky, M. Yang, M. Interlandi, H. Chiu, T. Condie, and C. Zaniolo. Big data analytics with Datalog queries on Spark. In *SIGMOD '16*, 2016.

[58] J. Shun and et al. Smaller and faster: Parallel processing of compressed graphs with Ligra+. In *Proc. Data Compression Conference*, 2015.

[59] A. Singh and et al. Jupiter rising: A decade of clos topologies and centralized control in Google's datacenter network. In *Sigcomm*, 2015.

[60] S. Sudarshan and R. Ramakrishnan. Aggregation and relevance in deductive databases. In *VLDB '91*, pages 501–511, 1991.

[61] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson. From "think like a vertex" to "think like a graph". *PVLDB*, 7(3):193–204, 2013.

[62] W. Vogels. https://twitter.com/werner/status/25137574680.

[63] J. Wang and et al. Asynchronous and fault-tolerant recursive datalog evaluation in shared-nothing engines. *PVLDB*, 2015.

[64] D. Yan and et al. Blogel: A block-centric framework for distributed computation on real-world graphs. *PVLDB*, 2014.

[65] Y. Yu and et al. Distributed aggregation for data-parallel computing: Interfaces and implementations. SOSP '09, 2009.
[66] C. Zhang, F. Wei, Q. Liu, Z. G. Tang, and Z. Li. Graph edge partitioning via neighborhood heuristic. In *SIGKDD '17*, pages 605–614, 2017.
[67] D. Zhuo and et al. RAIL: A case for redundant arrays of inexpensive links in data center networks. In *NSDI '17*, 2017.
[68] D. Zhuo and et al. Understanding and mitigating packet corruption in data center networks. In *SIGCOMM*, 2017.

A RELATION PARTITIONING

Algorithm 1 shows the specific relation partitioning algorithm that is adopted in the Static Optimizer. For each relation `r`, if there is only one attribute being marked as '*', then `r` is partitioned by that attribute, because that is the only vertex attribute that can maintain the tuples of `r`; otherwise the static optimizer enumerates every possible partitioning and selects the one with the minimum number of SHUFFs. We assume the heuristic that recursive rules are executed many times. This assumption is reasonable as practical graph queries often run more than one iteration because of the dense connectivity between vertices in real-world graphs.

B GRAPHREX DSN EVALUATION

The Distributed Semi-Naïve in GraphRex (GR-DSN) pseudocode is shown in Algorithm 2. Here, w_i represents a worker that stores the subgraph V_i, and each vertex v maintains its own vertex id id_v and the edge list Γ_v.

The DSN algorithm works as follows. Initially, w_i initializes each vertex with `init` function (line 12-13). Specifically, w_i creates a local table $\mathtt{r}_v$ for each vertex v and each relation `r` except edge relation. Recall that the logical plan already ensures that all relations are indexable by vertex. In the `init` function (line 1-3), base rules are evaluated, which generates the initial tuple set *NewTuples* in each relation, and the entire tuple set *AllTuples* is initialized to be the same set. w_i then loops to iteratively evaluate recursive rules. In each iteration, w_i checks if new tuples were generated in last iteration (the Δ tuples in semi-naïve evaluation [43]) at vertex v and uses `recur` function to evaluate recursive rules one time, otherwise calls `sleep` to deactivate v (line 14-19). Inside `recur`, the recursive rules are evaluated based on Γ_v and *NewTuples* of last iteration to generate new *NewTuples*

Algorithm 1 Static Relation Partitioning

```
1: v_atts ← get the list of marked attributes of r
2: if size of v_atts = 1 then
3:     Mark the attribute as the partition key
4: else
5:     v ← arg min_v(the number of SHUFF operators in the
   physical plan based on partition key v ∈ v_atts)
6:     Mark v as the partition key
```

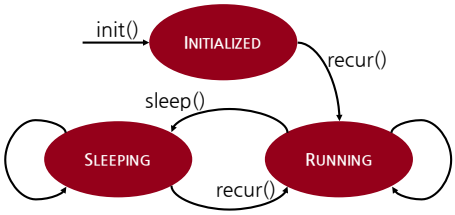


Figure 19: Vertex states in DSN.

(line 5), and then the deduplication is performed to eliminate redundant evaluation (line 6) and the resulting tuples are merged to the entire tuple set (line 7).

In the `eval` function, the corresponding part of execution plan is evaluated; and the executor consults the dynamic optimizer to execute each global operator efficiently. In particular, A SHUFF operator sends around new tuples according to their partition key. If a vertex v receives tuples, the callback function `onrecv` is invoked to handle the tuples. Specifically, the received tuples are merged to $NewTuples_v$ and deduplicated, and also added to $AllTuples_v$ (line 8-11).

Vertex states. A vertex in GraphRex could be in one of three states: *initialized*, *running* and *sleeping*. A vertex enters *initialized* after calling `init` to evaluate the base rules, and transitions to *running* on calling `recur`, where the recursive rules are iteratively evaluated in GR-DSN.

A significant difference from traditional, centralized semi-naïve evaluation is that when a vertex has no new tuples, it

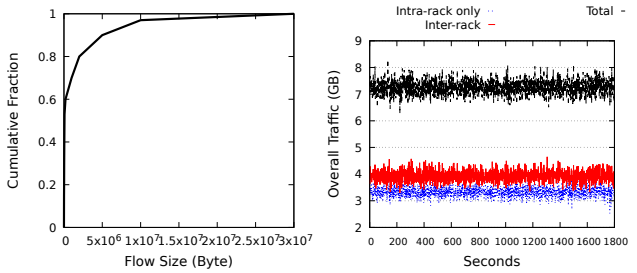
Algorithm 2 Distributed Semi-Naïve in GraphRex

```
1: function INIT(v)
2:     NewTuples_v ← eval(BaseRules, Γ_v)
3:     AllTuples_v ← NewTuples_v

4: function RECUR(v)
5:     NewTuples_v ← eval(RecurRules, Γ_v, NewTuples_v)
6:     NewTuples_v ← NewTuples_v − AllTuples_v
7:     AllTuples_v ← AllTuples_v ⋃ NewTuples_v

8: function ONRECV(v)
9:     NewTuples_v ← NewTuples_v ⋃ v's received tuples
10:    NewTuples_v ← NewTuples_v − AllTuples_v
11:    AllTuples_v ← AllTuples_v ⋃ NewTuples_v

12: for each vertex v ∈ V_i do
13:     init(v)
14: loop until the coordinator signifies to terminate
15:     for each vertex v ∈ V_i do
16:         if the size of NewTuples_v > 0 then
17:             recur(v)
18:         if the size of NewTuples_v = 0 then
19:             sleep(v)
```



(a) Flow size distribution in real data centers. **(b) Summarized pattern of background traffic.**

Figure 20: Background traffic generation.

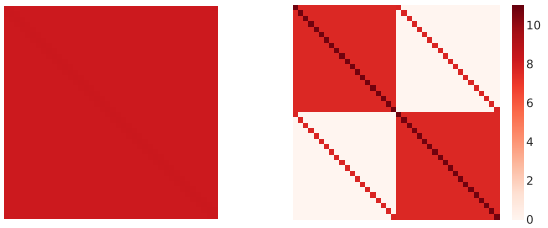
transitions to *sleeping*; if later, new tuples are received, the vertex will be activated again and transition into *running* again. This design ensures that the distributed evaluation converges globally rather than locally at a vertex level. The recursion reaches a fixpoint when: (1) all vertices in the graph are at the *sleeping* state, and (2) no tuples are being shuffled, i.e., no vertex received new tuples. The coordinator sends termination signal to workers when either the specified number of iterations or the fixpoint is reached.

C GENERALIZATION OF JOIN ORDERING

Here we explain how adaptive join ordering is applied to a 4-way join example: `r(X,Y) :- e(X,A), r(A,B), e(B,C), e(C,Y)`. Given a new `r` tuple, there are three possible join orders: (1) (((`e` ⋈ `r`) ⋈ `e`) ⋈ `e`), (2) ((`e` ⋈ (`r` ⋈ `e`)) ⋈ `e`), and (3) (`e` ⋈ ((`r` ⋈ `e`) ⋈ `e`)). The costs (in terms of the numbers of intermediate tuples) of the three orders for `r(v1,v2)` are: (1) $C_1 = InDeg(\texttt{v1}) + InDeg(\texttt{v1}) \times OutDeg(\texttt{v2})$, (2) $C_2 = OutDeg(\texttt{v2}) + InDeg(\texttt{v1}) \times OutDeg(\texttt{v2})$, and (3) $C_3 = OutDeg(\texttt{v2}) + Out^2Deg(\texttt{v2})$, where $InDeg(v)$ is v's indegree, $OutDeg(v)$ is v's outdegree and $Out^2Deg(v)$ is v's two-hop outdegree. Therefore, the global information needed by GraphRex for this query is: the indegrees of all vertices, the outdegrees of all vertices and the two-hop outdegrees of all vertices, which is $O(|V|)$ where V is the set of vertices. This information can be computed offline and loaded by GraphRex as index, so that when GraphRex enumerates the three orders for a tuple, the costs of the orders can be efficiently computed and GraphRex selects the order with minimum cost for this tuple. Similarly, the adaptive join ordering can be extended to other values of n for n-way joins.

D DATACENTER TRAFFIC GENERATOR

We generate background traffic by using the commonly used datacenter flow patterns from DCTCP [14]. Figure 20a shows DCTCP flow size distribution, and Figure 20b shows an example of the random background traffic that we inject into our testbed (the first half hour). The blue line represents the total volume of intra-rack only traffic in each second, and red line represents inter-rack traffic. We note that inter-rack traffic also consumes the bandwidth of intra-rack links. The total traffic volume in every second is represented in black line. We control the overall network utilization as 40%.



(a) The baseline. **(b) GraphRex.**

Figure 21: Heat maps of traffic volume (number of bytes sent between servers, values are log 10 scale) for CC on *Friendster*. GraphRex (b) saves 94.8% traffic compared to the infrastructure-agnostic baseline (a).

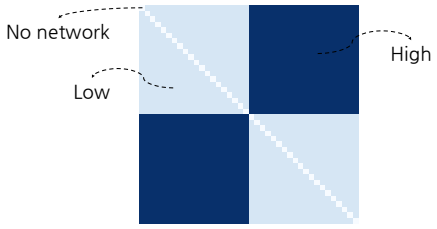


Figure 22: Heat map of cross-server communication.

E COMMUNICATION PATTERN

Figure 22 shows the communication cost distribution in the datacenter, with three layers: (1) communications inside servers require no network traffic (the left diagonal in the server matrix), (2) communications between servers in the same rack require traffic to be sent intra-rack (the light blue areas), and (3) communications between servers in different racks, which incur the highest traffic cost.

Figure 21 compares GraphRex against the infrastructure-agnostic baseline in terms of the communication patterns. Although the baseline has server-level locality, i.e., each worker sends more traffic to the workers in the same server than the workers in other servers, it ignores the network structure and treats all other servers as the same. However, the communication pattern in GraphRex results in two benefits. *Reduced traffic:* GraphRex prefers low-cost communications to reduce high-cost traffic due to its infrastructure-aware design, minimizing the amount of inter-rack traffic by incurring additional intra-rack communication. As a result, in this example, it reduces the traffic cost by 94.8%.

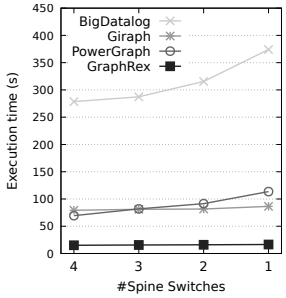


Figure 23: Performance on *Friendster*.

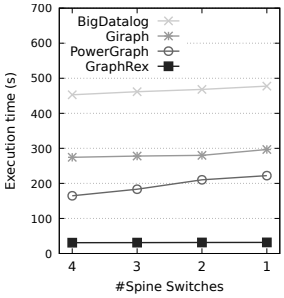


Figure 24: Performance on *UK*.

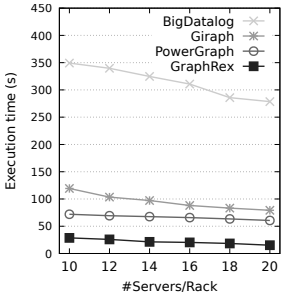


Figure 25: Scalability on *Friendster* with #Servers.

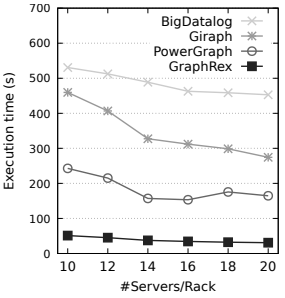


Figure 26: Scalability on *UK* with #Servers.

Fewer connections: In the baseline, every worker directly builds $N-1$ connections with all other workers for shuffling, where N is the number of parallel workers. In GraphRex, each worker establishes $W-1$ connections with other workers in the same server first, where W is the number of workers in the same server; then, at the rack level, it establishes at most $S-1$ connections with other servers in the same rack, where S is the number of servers in the same rack. Finally, it establishes at most $R-1$ connections with other racks, where R is the number of racks in the datacenter. Therefore, the number of connections that each worker builds in GraphRex is $O(W+S+R)$. In the naïve approach, assuming that all racks have the same number of servers and all servers have the same number of workers, we have $O(W \times S \times R)$ instead.

In summary, The infrastructure-centric design minimizes traffic cost by reducing traffic sent over bottleneck links.

F ADDITIONAL RESULTS

We show results on more workloads in this section.

F.1 Spine Switch Count

Figure 23 and 24 compare the performance of GraphRex with other systems when querying CC on *FR* and *UK* when we vary the network capacity by changing the number of spine switches from 4 to 1. The results show similar trend as on *TW* that GraphRex is the most robust system when network performance varies. Among other systems, PowerGraph is fastest when network capacity is not constrained, and it is also the most sensitive system to network changes. Its performance is lower than Giraph on *FR* when network capacity is low. Giraph and BigDatalog are also significantly impacted when the number of switches drops. We omit other queries for space and note that the finding of GraphRex being the most robust system still holds.

F.2 Server-rack Ratio

Figure 25 and 26 present the performance of different systems for CC on *FR* and *UK*, respectively, when the number of servers in the cluster changes. GraphRex achieves the highest speedup when the number of servers in the datacenter doubles: 1.8× on *FR* and 1.7× on *UK*. Although PowerGraph always has the best performance, it does not scale as well as other systems, and on *UK*, its performance does not continue to improve when the number of servers in each rack is higher than 16. Adding more machines improves the performance of Giraph and BigDatalog, but their scalability is not as good as GraphRex which minimizes the impact from network constraints, and scales better with more resources.

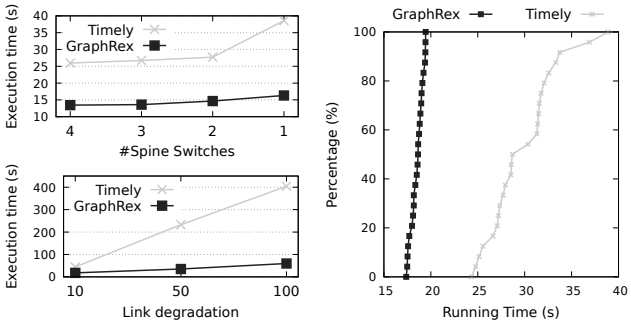


Figure 27: Comparison with Timely.

F.3 Comparison with Timely Dataflow

Figure 27 shows additional experiments comparing GraphRex and Timely in PageRank under different network situations. As we can see, GraphRex is more robust than Timely. Upper-left figure shows that when the number of spine switches drops from 4 to 1, the speedup of GraphRex over Timely increases from 1.9× to 2.4×, lower-left figure shows that link degradation severely impacts the performance of Timely, and the speedup of GraphRex over Timely goes up from 2.5× to 6.8× when the link capacity drops from $1/10$ to $1/100$. The right figure presents the CDF of running times of GraphRex and Timely under random background traffic, showing that GraphRex has better and more stable performance ($\sigma = 0.7$) than Timely ($\sigma = 3.7$) when noise is present.