Distributed Evaluation of Subgraph Queries Using Worst-case Optimal Low-Memory Dataflows

Khaled Ammar†, Frank McSherry‡, Semih Salihoglu†, Manas Joglekar♯
†University of Waterloo, ‡ETH Zürich,♯Google, Inc
khaled.ammar, semih.salihoglu@uwaterloo.ca,
frank.mcsherry@inf.ethz.ch, brahmaneya@gmail.com

ABSTRACT

We study the problem of finding and monitoring fixed-size subgraphs in a continually changing large-scale graph. We present the first approach that (i) performs worst-case optimal computation and communication, (ii) maintains a total memory footprint linear in the number of input edges, and (iii) scales down per-worker computation, communication, and memory requirements linearly as the number of workers increases, even on adversarially skewed inputs.

Our approach is based on worst-case optimal join algorithms, recast as a data-parallel dataflow computation. We describe the general algorithm and modifications that make it robust to skewed data, prove theoretical bounds on its resource requirements in the *massively parallel computing* model, and implement and evaluate it on graphs containing as many as 64 billion edges. The underlying algorithm and ideas generalize from finding and monitoring subgraphs to the more general problem of computing and maintaining relational equi-joins over dynamic relations.

PVLDB Reference Format:
Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. Distributed Evaluation of Subgraph Queries Using Worst-case Optimal Low-Memory Dataflows. *PVLDB*, 11(6): 691-704, 2018.
DOI: https://doi.org/10.14778/3184470.3184473

1. INTRODUCTION

Subgraph queries, i.e., finding instances of a given subgraph in a larger graph, are a fundamental computation performed by many applications and supported by many software systems that process graphs. Example applications include finding triangles and larger clique-like structures for detecting related pages in the World Wide Web [19] and finding *diamonds* for recommendation algorithms in social networks [23]. Example systems include graph databases [41, 56], RDF engines [42, 67], as well as many other specialized graph processing systems [2, 38, 54]. As the scale of real-world graphs and the speed at which they evolve increase, applications need to evaluate subgraph queries both offline and in real-time on highly-parallel and shared-nothing distributed systems.

This paper studies the problem of evaluating subgraph queries on large static and dynamic graphs in a distributed setting, with efficiency and scalability as primary goals. Our approach is to design distributed versions of recent worst-case join algorithms [43, 45, 58]. We show that our algorithms require memory that is linear in the size of the input graphs and are worst-case optimal in terms of computation and communication costs (defined momentarily). We also show optimizations to balance the workload of the machines in the cluster (*workers* hereafter) and make our algorithms provably skew-resilient, i.e., guarantee that the costs per worker decrease linearly as we introduce additional workers. We prove the efficiency of our algorithms theoretically and demonstrate their practicality through extensive evaluation of their implementations in the Timely Dataflow system [40, 55]. Although we focus on subgraph queries, our algorithmic and theoretical contributions apply equally to more general relational equi-joins.

Proceedings of the VLDB Endowment, Vol. 11, No. 6
Copyright 2018 VLDB Endowment 2150-8097/18/02... $ 10.00.
DOI: https://doi.org/10.14778/3184470.3184473

1.1 Joins and Worst-case Optimality

Throughout the paper, we adopt the relational view of subgraph queries (as done by many previous work [2, 42, 58, 65]) in which any subgraph query can be seen as a multiway join on replicas of an `edge` table of the input graph. Given a directed subgraph query Q, we label each vertex in the query with an attribute a_i. Instances of Q in an input graph G is equivalent to the multiway join of tables `edge`(a_i, a_j) for each edge (a_i, a_j) in Q, where each `edge`(a_i, a_j) table contains each edge (u, v) in G. For example, the directed triangle query, in Datalog syntax, is equivalent to:

$$\texttt{tri}(a_1,a_2,a_3) \;:=\; \texttt{edge}(a_1,a_2),\texttt{edge}(a_2,a_3),\texttt{edge}(a_3,a_1)$$

In the serial setting, a join algorithm is *worst-case optimal* for a query Q if its computation cost is not asymptotically larger than the AGM bound of Q [9], which is the maximum possibly output size for the given size of the relations in Q. We refer to this quantity as $MaxOut_Q$. For example, on a graph with IN edges, $MaxOut_Q$ for the triangle query is $\text{IN}^{3/2}$. Analogously, we say a distributed algorithm has worst-case optimal computation and communication costs, if respectively the total computation and communication done across all workers are $O(MaxOut_Q)$, for any parallelism level, i.e., number of workers.

1.2 Existing Approaches

Existing distributed approaches that can be used to evaluate general subgraph queries can be broadly grouped into two classes: (i) edge-at-a-time approaches [16, 20, 27, 42, 52, 54, 65] that correspond to binary join plans in relational terms; and (ii) those that use variants of the Shares [5] or Hypercube [10, 11, 32] algorithm. We also review a recent vertex-at-a-time approach that has been used in the serial setting and on which we base our distributed algorithms.

1.2.1 *Edge-at-a-time Approaches*

Perhaps the most common approach to finding instances of a query subgraph is to treat it as a relational query, and to execute a sequence of binary joins to determine the result. For example, this approach computes:

```
open-tri(a₁,a₂,a₃):=edge(a₁,a₂),edge(a₂,a₃)
tri(a₁,a₂,a₃):=open-tri(a₁,a₂,a₃),edge(a₃,a₁)
```

Recent developments [43, 45] have shown that edge-at-a-time approaches are provably suboptimal. For example on a graph with IN edges, any edge-at-a-time approach will do $O(\text{IN}^2)$ computation in the worst-case and comparable communication in the distributed setting, which is worse than the AGM bound of $\text{IN}^{3/2}$. This is because irrespective of the join order, the worst-case size of the first join is $O(\text{IN}^2)$. Although couched in the language of worst-case bounds, these suboptimalities do manifest on real graph datasets, especially those demonstrating *skew*. For example, the largest graph we consider has a maximum degree of 45 million, and any algorithm that considers the $(4.5 \times 10^7)^2 \perp 2 \times 10^{15}$ candidate pairs of neighbors of the maximum degree vertex will simply not work.

Different systems have several optimizations on top of this basic approach including: (i) picking different join orders; (ii) decomposing the query into several subqueries; and (iii) preprocessing and indexing commonly appearing subqueries [15, 25, 33, 37, 66]. None of these techniques correct the asymptotic sub-optimality.

1.2.2 *The Shares Algorithm*

The second existing technique for distributed evaluation of subgraph queries is to use the *Shares* of *Hypercube* join algorithm from references [5, 10, 11, 32]. Consider a distributed cluster with w workers and a query with n relations and m attributes, i.e., n is the number of edges and m is the number of vertices in the query subgraph. Shares divides the m-dimensional output space equally over the w workers and replicates each tuple t of each relation to every worker that can produce an output that depends on t. Finally, each worker runs any local join algorithm on the inputs it receives.

There are several advantages of Shares. For most queries and parallelism levels w (but not all), Shares' communication cost is less than the AGM bound (and often much less). In addition, in distributed bulk synchronous parallel systems, in which the computation is broken down into a series of *rounds*, Shares requires a very small number of rounds. However, Shares' cumulative memory requirement is $O(w^{1-\epsilon}\text{IN})$ and its per worker memory requirement is $O(\frac{\text{IN}}{w^\epsilon})$. Here IN is the size of the input and $\epsilon \in [0, 1]$ is a query-dependent parameter. This implies a super-linear cumulative memory growth and sub-linear scaling of per-worker memory (and workload) as w increases. For example, for the triangle query, $\epsilon = 1/2$. Often ϵ is much smaller, and scaling becomes an increasingly resource-inefficient way to improve performance.

1.2.3 *Vertex-at-a-time Approaches*

In the serial setting, Ngo et. al. and soon after Veldheuizen recently developed the first worst-case optimal join algorithms called respectively the *NPRR* [45] and *Leapfrog TrieJoin* [58] algorithms. These algorithms were shown to be instances of another algorithm called *Generic Join* [43] (GJ), on which we base our algorithms. In graph terms, these algorithms adopt a *vertex-at-a-time* evaluation technique. Specifically, on a query that involves $a_1, ..., a_m$ vertices, these algorithms first find all of the (a_1) vertices that can end up in the output. Then they find all of the (a_1, a_2) vertices that can end up in the output and so forth until the final output is constructed. When extending a partial subgraph to a new vertex a_i, all of the edges that are incident on a_i are considered and intersected. For example, on the triangle query, these algorithms would first find all (a_1) vertices and then (a_1, a_2) edges that can possibly be part of a triangle. Then the algorithms extend these edges to (a_1, a_2, a_3) triangles by intersecting a_1's incoming and a_2's outgoing edges. Compared to edge-at-a-time approaches, these algorithms will never generate intermediate data larger $MaxOut_Q$. We note that Turbo_{ISO} [24] is a serial algorithm that was developed independently in the context of subgraph matching around the same time as NPRR and Leaprfrog Triejoin. The algorithm is not worst-case-optimal but overall adopts a vertex-at-a-time approach.

1.3 Our Approach and Contributions

Our approach is based on recasting the basic building block of the GJ algorithm as a distributed dataflow computation primitive. We optimize and modify this basic primitive to obtain different algorithms tailored for different settings and achieving different theoretical guarantees. Our contributions are as follows:

1. A distributed algorithm called *BiGJoin* for static graphs that achieves a subset of the theoretical guarantees we seek.
2. A distributed algorithm called *Delta-BiGJoin* for dynamic graphs that achieves the same guarantees as BiGJoin in insertion-only workloads.
3. A distributed algorithm called *BiGJoin-S* for static graphs that achieves all of the theoretical guarantees we seek including workload balance across distributed workers on arbitrary input instances.

We implement BiGJoin and Delta-BiGJoin algorithms in Timely Dataflow and evaluate their performances extensively. Our evaluations include comparisons against an optimized single threaded algorithm, an existing shared-parallel system, and two existing distributed systems specialized for evaluating subgraph queries. We show that our approach can monitor complex sugraphs very efficiently on graphs with up to 64B edges on a cluster of 16 machines using just over eight bytes per edge. Graphs at the scale we process are significantly larger than graphs used in previous work. We note that our algorithms can also be easily used in both existing distributed bulk synchronous parallel systems, such as MapReduce [17] and Spark [64], as well as streaming systems, such as Storm [57] and Apache Flink [14].

We end this section with a note on our theoretical contributions.
Delta-GJ's Optimality (Theorem 3.2): Our Delta-BiGJoin algorithm is a distributed version of a new incremental view maintenance algorithm we develop for join queries called *Delta-GJ*. We prove that under insertion only workloads Delta-GJ is worst-case optimal. When we distribute Delta-GJ in Delta-BiGJoin, we achieve worst-case optimality in terms of communication as well.
BiGJoin-S's Optimality (Theorem 3.4): The challenge in the distributed setting is to achieve optimality in cumulative bounds while requiring low memory, e.g., $O(\frac{\text{IN}}{w})$, and workload per worker. Indeed, a naive ?distributed? algorithm can send all of the input to one worker w^* and use a sequential worst-case join algorithm. This algorithm would achieve all of the optimality guarantees we seek but without balancing the workload in the cluster.

BiGJoin achieves cumulative worst-case optimality and in our real-world data sets and queries achieves good workload-balance and low per-worker memory. However, on adversarial inputs it can lead to a single worker performing most of the work. We address this theoretical shortcoming with BiGJoin-S. Specifically, BiGJoin-S is the first distributed join algorithm that has worst-case communication and computation costs and achieves workload-balance across workers on every query. In addition, BiGJoin-S achieves

```
$P_0$={}
for ($j$ = 1... $m$):
    $P_j$={}
    for ($p \in P_{j-1}$):
        // $\cap$ below is performed starting from smallest $Ext^i_j(p)$
        $ext_p = \cap Ext^i_j(p)$
        $P_j = P_j \cup ext_p$
```

Figure 1: Pseudocode of GJ.

these guarantees with as low as $O(\frac{\text{IN}}{w})$ memory per-worker. In prior work, reference [32] had shown that variants of the Shares algorithm have the same guarantees only for certain queries, e.g., cycles. We provide a detailed comparison of BiGJoin-S with the algorithm in reference [32] in our longer technical report [30].

2. PRELIMINARIES

2.1 Notation

We present our algorithms in the general setting when they process general multiway equi-join queries, also referred to as *full conjunctive queries*. Let Q be a query over n relational tables, R_1, ..., R_n, where each R_i is over a subset of m attributes $a_1, \ldots, a_m$. We let $\text{IN} = \Sigma_i |R_i|$ be the size of the input. We write Q as:

$$Q(a_1, ..., a_m) := R_1(a_{11}, ..., a_{1r_1}), ..., R_n(a_{n1}, ..., a_{nr_n})$$

2.2 Generic Join

We base our work on the GJ algorithm (Figure 1). Given a query Q, GJ consists of the following three high-level steps:

- **Global Attribute Ordering:** GJ first orders the attributes. Here we assume for simplicity the order is $a_1, \ldots, a_m$. We will have a stronger preference on the order, but everything that follows remains correct if the attributes are arbitrarily ordered.
- **Extensions Indices:** Let a *prefix j-tuple* be any fixed values of the first $j < m$ attributes. For each R_i and j-tuple p only some values for attribute a_{j+1} exist in R_i. Let the *extension index* Ext^i_j map each j-tuple p to values of a_{j+1} matching p in R_i:

  $$Ext^i_j : (p = (a_1, .., a_j)) \rightarrow \{a_{j+1}\} \ .$$

  Extension indices need three properties for the theoretical bounds of GJ: for a given p we can retrieve (i) the size $|Ext^i_j(p)|$ in constant time, (ii) the contents of $Ext^i_j(p)$ in time linear in its size, and (iii) check that a value e of attribute a_{j+1} exists in $Ext^i_j(p)$ in constant time. Throughout the text we denote by $Ext^i_j(p \bullet e)$ the operation of checking of value e in $Ext^i_j(p)$. These properties are satisfied by many indices, for example hash tables.
- **Prefix Extension Stages:** GJ iteratively computes intermediate results $P_1 \ldots P_m$, where P_j is the result of Q when each relation is restricted to the first j attributes in the common global order. GJ starts from the singleton relation P_0 with no attributes, determines P_{j+1} from P_j using the extension indices, and ultimately arrives at $P_m = Q$. Specifically, for each *prefix* j-tuple $p \in P_j$, GJ determines the (possibly empty) set of $(j + 1)$-tuples extending p by intersecting the $Ext^i_j(p)$ *extension sets* of each relation R_i containing a_{j+1}. This is done by proposing candidate extensions from the smallest of the sets, and then intersecting each candidate with the extension indices of the remaining relations. Starting from the smallest set, and in general performing this intersection in time proportional to the size of the smallest set ensures worst-case optimal run-time.

An example of GJ is given in our longer technical report [30]. We next re-state a theorem from [43] using our notation:

THEOREM 2.1. *[43] For any query Q comprising relations $R_1 \ldots R_n$ and attributes $a_1 \ldots a_m$, and any ordering of attributes, if Ext^i_j indices satisfy the three properties discussed above, GJ runs in time $O(mnMaxOut_Q)$.*

2.3 Massively Parallel Computation Model

Massively Parallel Computation (MPC) [10, 11, 32] is an abstract model of distributed bulk synchronous parallel systems. Briefly there are w workers in a cluster. The input data is assumed to be equally distributed among the workers arbitrarily. The computation is broken down into a series of *rounds*, where in each round the workers first perform some local computation and then send each other messages. The complexity of algorithms are measured in terms of three parameters: (1) r: the number of rounds; (2) L: the maximum *load* or messages any of the workers receives in any of the rounds; and (3) C: the total communication, i.e., sum of the loads across all rounds.

We extend MPC with a fourth parameter M that measures the memory that an algorithm uses. Let $LocM^t_k$ be the local memory that worker k requires in round t, excluding the output tuples. In our setting, $LocM^t_k$ will be the load L of worker k in round t and the amount of input data worker k has indexed. M is then the $\max_{t=1,...,r} \Sigma_{k=1,...,w} LocM^t_k$. We assume output tuples are written to a storage outside the cluster and do not stay in memories of workers. This is because any correct algorithm incurs this cost.

For simplicity, similar to prior work [4, 10, 32] our unit of communication and memory will be tuples and prefixes, instead of bits, and we assume that tuples and prefixes have a common unit size.

2.4 Timely Dataflow

Timely Dataflow [40] is a distributed data-parallel dataflow system, in which one connects dataflow *operators* describing computation using dataflow edges describing communication. The operators are *data-parallel*, meaning that their input streams may be partitioned by a provided key, and their implementations may be distributed across multiple workers. All operators are distributed across all workers, and each worker is responsible for the execution of some fraction of each operator, which allows our algorithms to share indices (of the underlying relations) between operators.

Timely Dataflow is a dataflow system in the sense that computation occurs in response to the availability of data, rather than through centralized control. The *timely* modifier corresponds to the extension of each operator with information about logical progress through the input streams, roughly corresponding to *punctuation* or *watermarks* in traditional stream processing systems. Importantly for the current paper, operators can delay processing inputs with some timestamps until others have finished, which can be used to synchronize the workers and ensure that the work queues of downstream operators have drained, an important component of ensuring a bounded memory footprint.

3. ALGORITHMS

Our algorithms are based on a common dataflow primitive that extends prefixes P_j to P_{j+1}. We first describe a naive version of the primitive that explains the overall structure (and is closest to the implementation we evaluate). We then develop the BiGJoin and Delta-BiGJoin algorithms using this core primitive. We then modify the primitive and develop BiGJoin-S to achieve workload-balance and skew-resilience.

3.1 Dataflow Primitive

The core dataflow primitive starts from a collection of P_j tuples stored across w workers, and produces the P_{j+1} tuples across the same workers. We first describe a dataflow that closely tracks the GJ algorithm, starting from the full collection P_j and producing the full collection P_{j+1}. We will need to modify this dataflow in several ways to achieve both memory boundedness and workload balance across workers to achieve our theoretical bounds, but this simpler description is instructive and empirically useful.

3.1.1 A synchronous implementation

We first describe the dataflow primitive as a sequence of steps, where workers execute each step to completion and synchronize between each step (corresponding to a round in BSP terms). Naive execution of these steps may produce very large amounts of data between steps and require very large memory in the workers.

- *Initially*: The tuples of P_j are distributed among the w workers arbitrarily. Each prefix p is transformed into a triple $(p, \infty, \perp)$ capturing the prefix, the currently smallest candidate set size, and the index of the relation with that number of candidates.
- *Count minimization*: For each R_i binding attribute a_{j+1}, in order: Workers exchange the triples by the hash of p's attributes bound by R_i, placing each triple at the worker with access to $Ext_j^i(p)$. Each worker, for each triple updates the smallest count and introduces its own index if $|Ext_j^i(p)|$ is smaller than the recorded smallest count. Each triple is then output as input of the count minimization for the next relation. In the end we have a collection of triples $(p, min\text{-}c, min\text{-}i)$ indicating for each prefix the relation with the fewest extensions.
- *Candidate Proposal*: Each worker exchanges triples using a hash of p's attributes bound by $R_{min\text{-}i}$. Each worker now produces for each triple $(p, min\text{-}c, min\text{-}i)$ it has, and each extension e of p in $Ext_j^{min\text{-}i}(p)$, a candidate $(j+1)$-tuple $(p \bullet e)$.
- *Intersection*: For each relation R_i binding attribute a_{j+1}, in order: Workers exchange the candidate $(p \bullet e)$ tuples by the hash of $(p \bullet e)$'s attributes bound by R_i. Each worker consults $Ext_j^i(p)$. If e exists $(p \bullet e)$ is produced as output otherwise it is discarded.

Figure 2 shows the operators of this dataflow primitive. In the figure, the vertical lines annotated with s indicate synchronization points. These steps, executed in sequence would be a synchronous BSP implementation of the computation extending P_j to P_{j+1}, which we could repeat until we arrive at $P_m = Q$. The random access working set of these operators are only the extension indices; all inputs and outputs are processed sequentially. Nonetheless, the sizes of the inputs and intermediate outputs to operators could be quite large requiring large memory/storage, which we address next.

3.1.2 A batching optimization to reduce memory

Notice that the `Proposal` operator is the only operator that may produce more output than it consumes as input and increase the memory usage of the system. We can fix this with a simple batching optimization. Instead of producing all of the proposals for each P_j prefix they have, each `Proposal` operator produces its candidate extensions in batches of B'. The remaining extensions are produced in the subsequent invocations. This may leave some prefixes only partially extended. To keep track of these partial extensions, we store $(p, min\text{-}c, min\text{-}i, rem\text{-}ext)$ quadruples where $rem\text{-}ext$ is the *remaining extensions* metadata. Letting $B = wB'$, this ensures that the dataflow has at most B queued elements at any

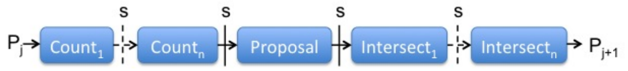


Figure 2: Dataflow Primitive.

time across the workers, as the B proposals created by `Proposal` operators are retired before any more are produced. The `Count` and `Intersect` steps remain unchanged.

3.1.3 A streaming implementation

Except for the `Proposal` operators, the operators described above do not need to synchronize. Specifically, instead of synchronizing, the `Count` and `Intersect` operators can produce outputs as inputs to their next operators as soon as they receive inputs. This leads to a streaming implementation, which can improve performance in practice. When implementing the above batching optimization however, the `Proposal` operators need to synchronize and be notified that they can produce another batch of extensions.

3.2 Joins on Static Relations: BiGJoin

We now describe how to use our dataflow primitive to build a dataflow for evaluating queries on static graphs. First, we order the attributes arbitrarily, and build indices over each relation for each prefix of its attributes in the global order. Next, we assemble the dataflows for extending each P_j to P_{j+1} for each attribute a_j, so that starting from an empty input tuple () we produce streams of prefixes P_j, out to $P_m = Q$. Finally, we introduce the empty tuple to start the computation, producing the stream of records from Q as output. We use the batching optimization described above and when deciding which batch of P_j to P_{j+1} extensions to invoke next, we pick the largest j value such that at least one worker has B' prefixes to propose. We refer to this algorithm as BiGJoin. The next lemma summarizes the costs of BiGJoin:

LEMMA 3.1. *Given a query Q over m attributes and n relations, the communication and computation cost of BiGJoin equals that of computation of GJ and is $O(mnMaxOut_Q)$. Let B' be a batching parameter and let $B = wB'$. The cumulative memory BiGJoin requires is $O(m\text{IN} + mB)$, and the number of rounds of computation BiGJoin takes is $O(\frac{mnMaxOut_Q}{B'})$.*

The proof of this lemma, presented in our longer technical report [30] , is based on the fact that each operation that BiGJoin does on each tuple corresponds to an operation in the serial execution of GJ and small enough batches can keep the memory footprint very low. In essence, BiGJoin inherits its computation and communication optimality from GJ. Moreover, as we will demonstrate in Section 5, in practice BiGJoin also achieves good workload-balance across the workers in the cluster. However, on adversarial inputs BiGJoin cannot guarantee workload balance. We will address this theoretical shortcoming in Section 3.4 to achieve one of our main theoretical results.

3.3 Joins on Dynamic Relations: Delta-BiGJoin

We next show how to use our dataflow primitive to maintain join queries over dynamic relations, which we use to maintain subgraph queries on dynamic graphs. We first describe a new incremental view maintenance (IVM) algorithm for join queries called *Delta-GJ* and then describe its distributed version Delta-BiGJoin.

3.3.1 Delta-GJ

Let Q be a query and consider a setting where for each relation R_i we have a change ΔR_i, corresponding to the addition and deletion of some records in R_i. We begin by reviewing an incremental view maintenance technique based on delta queries from references [13, 22]. Let's assume that tuples in each ΔR_i are labeled such that we can tell the inserted tuples apart from the deleted ones. Let R'_i be $R_i + \Delta R_i$, where the union operation removes a tuple t in R_i if ΔR_i contains a deletion of t. Let Out and Out' be the output of Q before and after the updates, respectively. Then consider the following n delta queries:

$$\begin{aligned} dQ_1 &:= \Delta R_1, R_2, R_3, ..., R_n \\ dQ_2 &:= R'_1, \Delta R_2, R_3, ..., R_n \\ dQ_3 &:= R'_1, R'_2, \Delta R_3, ..., R_n \\ &\quad ... \\ dQ_n &:= R'_1, R'_2, R'_3, ..., \Delta R_n \end{aligned}$$

We assume output tuples that emerge from inserted and deleted tuples are labeled as inserted and deleted, respectively. It can be shown that the union of the n queries above are exactly the changes to the output of Q, i.e., $Q' \setminus Q = dQ_1 + dQ_2 + ... + dQ_n$ [13, 22].

Delta-GJ runs the n delta queries indepenpendently, where each dQ_i is executed using GJ. Note that Delta-GJ's correctness, i.e., that it finds the correct differences to the output of Q, simply follows from the correctness of the delta query technique [13, 22]. However, in order to prove that Delta-GJ is efficient, we need to order the attributes of each dQ_i in a specific order. Specifically, for dQ_i, Delta-GJ picks an attribute ordering that starts with any permutation of R_i's attributes $a_{i1}, a_{i2}, ..., a_{ir_i}$ and an arbitrary order for the remaining $m - r_i$ attributes. The next theorem states that under insertion-only workloads, Delta-GJ is a worst-case optimal IVM algorithm for join queries. The proof is given in our longer technical report [30].

THEOREM 3.2. *Consider a query Q and a series of z updates that only consist of inserting tuples to the input relations of Q. Let $R_i(z)$ denote the relation R_i after the z'th update. Then the total computation cost of Delta-GJ is $O(mn^2 MaxOut_Q)$, where $MaxOut_Q$ is the AGM bound of Q on $R_i(z)$.*

It is harder to characterize the performance of Delta-GJ under workloads with both insertions and deletions because ?problematic? records might require a lot of work and could simply be repeatedly added and removed. A more precise characterization of Delta-GJ under arbitrary workloads is left as future work. We note that an incremental version of the Leapfrog TrieJoin algorithm [59] also achieves worst-case optimality under insertion-only workloads but by maintaining indices that can be super-linear in the size of the inputs. Delta-GJ's indices are linear in the size of the inputs.[1]

3.3.2 Delta-BiGJoin

We next describe how we parallelize Delta-GJ in the distributed setting. We have a separate dataflow for each dQi that is a dQ_i-specific variation of the BiGJoin dataflow from Section 3.2. By ordering the attributes of dQ_i starting with the attributes of R_i, we can seed the computation with the elements of ΔR_i, instead of (), which is expected to be much smaller than the other relations in dQ_i. Importantly, we only need to *maintain* the indices as changes occur, rather than fully rebuilding them. The resulting cost is proportional to the number of changes (for rebuilding indices) and the number of prefixes in the delta queries as we evaluate them.

The next lemma is proved in our longer technical report [30].

[1]In a separate paper, one of the co-authors and his colleagues have used Delta-GJ to support triggers in the context of an active graph database called Graphflow [31]. The Graphflow paper cited a previous partial technical report version of the current paper.

LEMMA 3.3. *Consider a series of z insertion-only updates to the input relations of a query Q. Let $R_i(z)$ denote the relation R_i after the z'th update and $\text{IN}(z)$ be $\sum_i |R_i(z)|$. Then, given a batch size B' and letting $B = wB'$, Delta-BiGJoin's communication and computation cost is $O(mn^2 MaxOut_Q)$. The cumulative memory Delta-BiGJoin uses is $O(mn\text{IN}(z) + mB)$. In MPC terms, the number of rounds of computation Delta-BiGJoin takes is $O(\frac{mn^2 MaxOut_Q}{B'} + zmn^2)$.*

3.4 A Work-balanced Dataflow: BiGJoin-S

As we demonstrate in Section 5, BiGJoin and Delta-BiGJoin perform very well on the real-world queries and datasets we experimented with. However they have an important theoretical shortcoming. Specifically, they do not guarantee that the workloads of the workers are balanced. Indeed, it is easy to construct skewed inputs where most of the work could even be performed by a single worker. We next modify our dataflow primitive to ensure workload balance across workers. We note that the contributions of this section are theoretical. An implementation and evaluation of these techniques are left for future work.

There are three sources of imbalance in our dataflow primitive:

1. *Sizes of extension indices:* Recall that Ext^i_j are distributed randomly. Yet for each prefix p, a single worker stores the entire $Ext^i_j(p)$ (the a_{j+1} extensions of p). In graph terms, this corresponds to a single worker storing the entire adjacency list of a vertex. On skewed inputs, this may generate imbalances in the amount of data indexed at each worker.
2. *Number of Proposals:* After count minimization, each worker gets a set of $(p, min\text{-}c, min\text{-}i, rem\text{-}ext = min\text{-}c)$ quadruples where p is a P_j prefix to extend. Even if each worker has to extend the same number of prefixes, each worker might have to do imbalanced amount of proposals of $(p \bullet e)$ candidate extensions because the counts might be very different.
3. *Number of Index Lookups:* When minimizing the counts of a P_j prefix p, producing the candidate proposals, or intersecting the $(p \bullet e)$ candidate extensions with Ext^i_j, prefixes and candidate extensions are routed to the worker that holds $Ext^i_j(p)$ based on the hash of p's attributes that are bound by R_i. If there are many prefixes whose attribute values that are bound by R_i are the same, there may be an imbalance in the number of prefixes and extensions each worker receives. For example, consider a triangle query where all triangles involve some specific vertex v^*, then every P_2 prefixes could be routed to a single worker to access v^*'s count.

We show how to fix these sources of imbalance without asymptotically affecting the other costs of BiGJoin.

3.4.1 Skew-resilient Extension Indices

We distribute the contents of $Ext^i_j(p)$ across workers, instead of storing at a single worker. Specifically, we store three indices.

- $C^i_j(p)$ *Count Index*: Stores the size of $Ext^i_j(p)$. This index is distributed randomly by the hash of prefixes p.
- $Ext\text{-}Res^i_j((p,k))$ *Extension Resolver Index*: Let $\{e_1, ..., e_c\}$ be the a_{j+1} extensions of p in R_i. We use $Ext\text{-}Res^i_j((p,k))$, for $k = 1 \dots c$, to *resolve* the kth extension, i.e., $Ext\text{-}Res^i_j((p,k))$ $= e_k$. This index is distributed randomly by the hash of (p,k) tuples. Essentially we distribute each element of $Ext^i_j(p)$ randomly across the workers.
- $Ext^i_j((p \bullet e))$: As in the original extension indices, this index is used to lookup the existence of a particular extension e of p and is distributed randomly by the hash of (p, e).

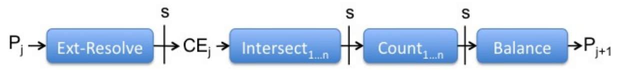


Figure 3: Work-balanced Dataflow Primitive.

Since the contents of $Ext^i_j(p)$ are now randomly distributed across workers, these indices fix the first source of imbalance from above.

3.4.2 *Balance and Extension Resolving Operators*

We modify the dataflow primitive as shown in Figure 3. Compared to BiGJoin's dataflow primitive, BiGJoin-S's dataflow primitive contains modifications of the `Count` and `Intersect` operators, does not contain the `Proposal` operator, and contains two new operators `Balance` and `Extension-Resolve`. In Figure 3, the P_j tuples are $(p, min\text{-}i, start, end)$ quadruples which indicate a range, indicated by $start$ and end of candidate extensions that the worker holding the tuple should make for the prefix j-tuple p. The dataflow however takes (p, k) tuples where $k \in [start, end]$ and produces a set of $(p', min\text{-}i, start, end)$ where $p' \in P_{j+1}$. The (p, k) tuples move along the operators as follows:

- `Extension-Resolve`: Each worker for each $(p, min\text{-}i, start, end)$ tuple and $k \in [start, end]$, *resolves* (p, k) by consulting the $Ext\text{-}Res^{min\text{-}i}_j$ indices and gets back a candidate extension $(p \bullet e)$. As in BiGJoin this happens in batches of B' (p, k) tuples per worker. Recall that the lookup for (p, k) is made to the worker that holds $Ext\text{-}Res^{min\text{-}i}_j$ based on the attributes of p bound by $R_{min\text{-}i}$. Although each (p, k) is distinct, there may be skew in the attribute values of (p, k) that are bound by $R_{min\text{-}i}$. To guard against this, workers first locally aggregate the $Ext\text{-}Res^{min\text{-}i}_j$ requests they will make to the same relation $R_{min\text{-}i}$ with the same lookup key (the attributes of p bound by $R_{min\text{-}i}$) and k value, and send only one request instead. This and a similar aggregation optimization in the `Intersect` and `Count` operators fix the third source of imbalance from above. Upon receiving the answers to their requests, workers have the set of candidate extensions, which we refer to as CE_j.
- `Intersect`: Instead of routing each $(p \bullet e)$ through the Ext^i_j indices one by one as done by BiGJoin's `Intersect` operator, each worker *manages* each $(p \bullet e)$ candidate extension it initially holds. Specifically for each R_i that contains a_{j+1}, in synchronous rounds, each worker does a distributed lookup of $(p \bullet e)$ in Ext^i_j by sending $(p \bullet e)$ to the worker that holds $Ext^i_j(p \bullet e)$ and gets the tuple back with a yes/no label. Similar to the `Extension-Resolve` operator, workers locally aggregate the requests they will make with the same key.

  Recall that the worker that holds $Ext^i_j(p \bullet e)$ is based on the hash of the attributes of $(p \bullet e)$ bound by R_i (possibly including the value e). Although each $(p \bullet e)$ candidate extension is distinct, there may be skew in these projections. To guard against this, before sending their lookup requests, workers first aggregate the projections of all of their CE_j candidate extensions and for each possible projection sends at most one lookup request. After at most n intersections, each $(p \bullet e)$ either becomes a P_{j+1} prefix or is discarded if it does not successfully intersect an R_i.
- `Count`: For each P_{j+1} prefix, workers compute the $(p' = (p \bullet e), min\text{-}c, min\text{-}i)$ triples, after at most n synchronous rounds, by looking up p' in the C^i_{j+1} indices by aggregating lookups with the same key. Similar to the above `Intersect` operator and unlike BiGJoin's `Count` operator, instead of routing the prefixes through each C^i_{j+1} index, the workers manage the triples.
- `Balance`: For each $(p', min\text{-}c, min\text{-}i)$ tuple, there is `min-c` number of proposals and following intersections to make. There may be an imbalance in how much intersection work each worker gets after the count minimization. To balance this intersection work, each worker deterministically distributes its total proposal work among the other workers. Each worker w_ℓ first finds the target intersection work amount T to distribute and gives T/w proposal and intersection work (with a +/- 1 difference) to each other worker $w_{\ell'}$. This is done by sending $(p', min\text{-}i, start, end)$ tuples to $w_{\ell'}$. The $start \leq end \leq min\text{-}c$ indicate the range of extensions among the $min\text{-}c$ total candidate extensions of p' that the receiving worker $w_{\ell'}$ is responsible for. At this point each worker gets a set of $(p', min\text{-}i, start, end) \in P_{j+1}$ tuples. This fixes the second form of skew discussed above.

Similar to BiGJoin, we assemble this workload-balanced dataflow for extending P_j to P_{j+1} for each attribute a_j. We call this algorithm with batching optimization in the `Ext-Resolve` operator BiGJoin-S. When deciding which batch of P_j to P_{j+1} candidate extensions to compute next, BiGJoin-S picks the largest j value such that all of the workers have B' candidate extensions to resolve and propose (instead of at least one as in BiGJoin). The following theorem states that BiGJoin-S achieves workload balance over large enough, but logarithmic size, batch sizes, while asymptotically maintaining the optimality bounds of BiGJoin on any query and arbitrary datasets (so under any amount of skew in inputs). It is the first algorithm to achieve these bounds for arbitrary queries and datasets. A detailed comparison of its costs against a variant of the Shares algorithm is provided in our longer technical report [30]. The proof is technical and provided also in our longer technical report [30].

THEOREM 3.4. *Suppose* $\frac{B'}{w} \geq \max\{w, \log(\text{IN} \times MaxOut_Q)\}$ *and let* $B = wB'$. *Then BiGJoin-S has the following costs:*

- *Cumulative computation and communication cost of* $O(mnMaxOut_Q)$ *and memory cost of* $O(mn\text{IN} + mB)$.
- $O(\frac{mnMaxOut_Q}{B})$ *rounds of computation.*
- *With at least probability* $1 - O(\frac{1}{\text{IN}})$, *each worker performs* $O(B')$ *communication and computation in each round of the algorithm. In MPC terms, the load of BiGJoin-S is* $O(\frac{mn\text{IN}}{w} + mB')$, *so assuming* $B' < \frac{IN}{w}$, *BiGJoin-S has optimal load.*

We note that we can make Delta-BiGJoin also skew-resilient under large enough updates by using BiGJoin-S with delta queries instead of BiGJoin. We need the update sizes, i.e., the size of the ΔR_i, to be large enough to make Delta-BiGJoin-S skew-resilient. For example if each update to the relations contain a single tuple, then the amount of work to maintain the query results could be too small to possibly distribute equally across workers.

4. IMPLEMENTATION

In this section we describe our implementations of BiGJoin and Delta-BiGJoin in Timely Dataflow. Although our implementations are tailored for evaluating subgraph queries, so the input relations are binary relations consisting of the edges of an input graph, the underlying machinery nonetheless is suitable for more general queries. We will demonstrate this in Section 5.4 when using our algorithm to take as input a ternary relation. We start by developing the prefix extension dataflow primitive as a Timely fragment. Our implementations can be found here [29].

4.1 Prefix Extension in Timely Dataflow

Our approach to prefix extension follows the primitive from Section 3.1: we will assemble a dataflow fragment that starts from a stream of prefixes of some number j of attributes, and produces as output the corresponding stream of prefixes resulting from the extension of each input prefix by the relations constraining the attribute a_{j+1} in terms of the first j attributes. As we explain in Section 4.3, for our Delta-BiGJoin implementation, the prefixes are tagged with a timestamp and a signed integer, reflecting the time of change and whether it is an addition or deletion, respectively.

Prefix extension happens through three methods acting on streams, corresponding to the three steps described in Section 3.1: count minimization, candidate proposal, and intersection. Each of these steps is implemented as a sequence of operators, each of which corresponds to one of the relations constraining attribute a_{j+1} . Each operator will consult some indexed form of the relation it represents, and requires the prefixes in its input stream to be shuffled by the corresponding attribute, so that prefixes arrive at the worker that store the appropriate fragment of the index. Importantly, we use the same partitioning for each relation and attribute in that relation, so that any number of uses of the relation in the query require only one physical instance of each index. In the case of graph processing, this means we keep only a forward and reverse index, storing respectively the outgoing and incoming neighbors of each vertex.

4.1.1 Count Minimization

The implementation of this step is straightforward and follows our description in Section 3.1 directly. There is a sequence of operators and each one represents one relation $R_i(a_{j+1}, a_k)$ or $R_i(a_k, a_{j+1})$, where $k \leq j$. The operator takes (p, c, i) triples as input. Let $v^* = \Pi_{a_k} p$. The operator updates the count c if the size of v^*'s outgoing neighbors (if $R_i = R_i(a_{j+1}, a_k)$), or incoming neighbors (if $R_i = R_i(a_k, a_{j+1})$), is less than c. At the end we identify the $(p, min\text{-}c, min\text{-}i)$ triples but then send only p to a stream for R_i (explained next).

4.1.2 Candidate Proposal

This step is implemented by a single operator that divides its stream of input prefixes into one stream for each relation R_i, by the $min\text{-}i$ index identified in the previous stage. Suppose $R_i = R_i(a_{j+1}, a_k)$. Then an input p whose $min\text{-}i$ was i, where $v^* = \Pi_{a_k} p$, will be part of the stream for R_i and be extended to a tuple $(p \bullet \{e_1, ..., e_c\})$ containing the *set of candidate extensions*, which are v^*'s outgoing neighbors. When $R_i = R_i(a_k, a_{j+1})$, we use the incoming neighbors of v^* instead. This deviates from our description where we had flattened this tuple for simplicity of explanation and had c separate $(p \bullet e)$ candidate extensions. These extensions are sent through a single output stream for the next stage.

4.1.3 Intersection

The stream of pairs of prefix and candidate extensions go through a sequence of operators, one for each involved relation R_i, each of which intersects the set of candidate extensions with an appropriate neighbor list of a vertex and removes those extensions that do not intersect. The result is a stream of pairs of prefix and valid extensions, successfully intersected by all relations. The extensions are flattened to a list of prefixes for the next stage except if they are the final outputs, they are output in their compact representation.

4.2 The BiGJoin Dataflow

The dataflow for enumerating subgraphs in a static graph applies a sequence of prefix extension stages, each corresponding to an attribute in the global attribute order. For simplicity, we fix the global order so that the first two attributes are connected by an edge, which allows us to seed the stream of prefixes with length-two prefixes read from the edges themselves. This is equivalent to starting the extensions from P_2 instead of the empty tuple $() \in P_0$. All other attributes are extended using the prefix extension dataflow fragment we described above.

The indices used by the workers are static, and we simply memory-map in a pre-built index. For simplicity we use the whole graph, which means we can easily vary the number of workers without changing the index used. One could alternately partition the graph and provide each worker with its own index, but the graphs on which we evaluate the static computations are rather small. For larger graphs, such as those we consider with DeltaBiGJoin, we build the indices as part of the computation, distributing the data to only the workers that require it.

The execution of the BiGJoin dataflow happens in batches, where we feed some number of prefixes into the dataflow and await their results before introducing more prefixes. This batching allows some control over the peak memory requirements, but does not guarantee that in the course of processing a batch we do not produce unboundedly many intermediate results. To manage back-pressure more precisely one can use the batching techniques described in [34], which allow the *Proposal* stages to wait until the downstream dataflow has drained as our batching optimization from Section 3.1.2 does, but we have not implemented them for our evaluation as our basic input batching worked well in our evaluations.

4.3 The Delta-BiGJoin Dataflow

The dataflow for finding subgraphs in a dynamically changing graph is more complex than for a static graph, along a few dimensions. First, as described in Section 3.3, we will have an independent dataflow for each dQ_i. Each dataflow is responsible for changes to each logical relation R_i in the query, i.e., one for each of the edges in the subgraph query. Second, although these dataflows may execute concurrently, we will logically sequence them so that each dataflow computes the delta query as if executed in sequence (to resolve simultaneous updates correctly). Third, our index implementation will be more complicated, as it must support changes as well as the multi-versioned interaction required by the logical sequencing above.

We have a dataflow for each dQ_i, each of which uses a different global attribute order as described in Section 3.3. Although there are several dataflows with different attribute orders, each operator only requires access to either the *forward* or *reverse* edge index.

Each delta query dataflow dQ_i computes changes in the outputs made to relation R_i with respect to the other relations. Recall that dQ_i uses the ?new? versions $R'_i = R_i + \Delta R_i$ for $i < j$ and the ?old? versions R_i for $i > j$. This has the effect of logically sequencing the update rules, so that they are correct even if there are simultaneous updates to the input relations, something we expect in graph queries where the single underlying edges relation is re-used often. This use of new and old versions of the same index requires our implementation to be multi-versioned, if we want to only have a single copy of each index.

Our index implementation is a multi-version index, which tracks the accumulation of (src, dst) pairs at various times and with various integer weights. It can respond to queries about the outgoing and incoming neighbors for a given key v using updates at a target time. The updates are ?committed? when all tuples in the system have a timestamp greater than it (meaning the update will participate in all future accumulations for v); this information comes from Timely Dataflow's progress tracking infrastructure. The index maintains data in three regions: (i) a compacted index of committed updates, (ii) an uncompacted index of committed updates, and (iii) an ordered list of uncommitted updates. Committed updates are moved to the uncompacted index, which uses a log-structured merge list for each v, which can be compacted on a per-vertex basis to ensure that the amortized work for each vertex lies within the bounds prescribed for the worst-case optimality result. The compact index is formed from initial data during loading and in principle could be periodically re-formed by merging the uncompacted committed data. Practically, on large datasets we were not able to apply enough updates to make such re-compaction worthwhile, within reasonable experimentation timeframes.

The execution of the Delta-BiGJoin dataflow proceeds with the stream of batches of updates to the graph supplied as an input. Each of the tuples moving through a delta query dataflow has both a logical timestamp and a signed integer weight. The former allows us to work with multiple logical times concurrently, and to remain clear on which version of an index the prefix should be matched against. The integer weight allows us to represent both additions and deletions from the underlying relations.

5. EVALUATION

We next evaluate the performance of our Timely Dataflow implementations of our algorithms on a variety of subgraph queries and large-scale static and dynamic input graphs.

We first evaluate a reference computation (triangle finding) on several standard graphs using a few different systems, to establish a baseline for running time (Section 5.2). With each system we quickly discover limits on their capacity; they struggle to load graphs at the larger end of the spectrum. We then study the scaling of our implementation as we vary the number of Timely workers both within a single machine as well as across multiple machines on a 64 billion-edge graph (Section 5.3). We next demonstrate that several optimizations that have been introduced in prior work can also be integrated into our algorithms to improve our algorithms (Section 5.4). Here we also show an experiment in which we use a ternary relation as input, demonstrating our algorithms' application to general relational queries. Finally, we study the effects of our batch size on performance and memory usage (Section 5.5).

We used both BiGJoin and Delta-BiGJoin in our experiments and refer to their Timely Dataflow implementations as BiGJoinT and Delta-BiGJoinT, respectively. Unless specified explicitly, we use a batch size of $100,000$ in all our experiments.

5.1 Experimental Setup

Table 1 reports statistics of the graphs we use for evaluation. The sizes range from the relatively small but popular LiveJournal graph, with 68 million edges, up three orders of magnitude to the relatively large Common Crawl graph, with 64 billion edges. The abbreviations we use for the datasets are given in parentheses in Table 1. We used five queries:

- `triangle`:= e(a_1,a_2),e(a_1,a_3),e(a_2,a_3)
- `4-clique`:= e(a_1,a_2),e(a_1,a_3),e(a_1,a_4),e(a_2,a_3),e(a_2,a_4),e(a_3,a_4)
- `diamond`:= e(a_1,a_2), e(a_2,a_3),e(a_4,a_1), e(a_4,a_3)
- `house`:= e(a_1,a_2),e(a_1,a_3),e(a_1,a_4),e(a_2,a_3),e(a_2,a_4),e(a_3,a_4), e(a_2,a_5),e(a_3,a_5)[2]
- `5-clique`:= e(a_1,a_2),e(a_1,a_3),e(a_1,a_4),e(a_1,a_5),e(a_2,a_3),e(a_2,a_4), e(a_2,a_5),e(a_3,a_4),e(a_3,a_5),e(a_4,a_5)

We note that the Common Crawl dataset has prohibitively large number of instances of each query. For example we estimate that there are more than 2.36×10^{16} diamonds in Common Crawl, and enumerating all of them explicitly would take a prohibitively long time for any correct system. Instead, for the Common Crawl graph we focus on the *incremental maintenance* of these queries, which

[2]This is query $q6$ from the SEED reference [37] and is a 5-clique with two missing edges from one node.

Table 1: Graph datasets used in our experiments.

Name	Vertices	Edges
LiveJournal (LJ) [36]	4.8M	68.9M
Twitter (TW) [35]	42M	1.5B
UK-2007 (UK) [35]	106M	3.7B
Common Crawl (CC) [60]	1.7B	64B

can fortunately be performed without the initial computation of all answers.

For all experiments except one we used a local cluster of up to 16 machines. All machines have 2x Intel E5-2670 @2.6GHz CPU with 16 physical cores in total. Most machines have 256 GB memory, but we occasionally used a machine with 512 GB memory to accommodate single-machine experiments. Each machine has 10 Gigabit network interface. For experiments using EmptyHeaded (see Section 5.2.2), we used an AWS machine similar to our cluster machines (r3.8xlarge) and another machine with 1TB memory (x1.16xlarge) to accommodate EmptyHeaded's memory requirements when running the triangle query on the TW graph.

In all of our experiments we use one CPU core for each Timely worker. For each experiment we explicitly state how the workers are located, i.e., within a single machine, across machines, or both.

5.2 Baseline measurements

We start with measurements of several existing approaches for finding subgraphs in static graphs. Our goal is to assess whether our implementations have relatively good absolute performance when evaluating queries in static graphs. We consider three baselines: (1) a single threaded implementation; (2) the shared-memory parallel EmptyHeaded system; and (3) the distributed Arabesque system. All of these implementations operate only on static graphs. None of these implementations are capable of working with our largest graph, and not all of them can evaluate our smaller graphs either.

5.2.1 COST

COST [39] (configuration that outperforms a single thread) is a metric to evaluate the parallelization overheads of a parallel algorithm or system. Specifically, COST of a parallel algorithm A solving a problem P is the number of cores that the algorithm needs to outperform an optimized single-threaded algorithm solving P. A small COST indicates that the system itself introduces little overhead, and the benefits of scaling are immediately realized.

In order to measure the COST of our algorithms, we implemented an optimized single-threaded triangle enumeration algorithm, that is based on GJ in Rust [49]. We considered using the popular SNAP library [36], but found that our own single-threaded implementation was faster. We used the TW data set. Figure 4 shows the optimized single-threaded, BiGJoinT, and Delta-BiGJoin-T implementations for the triangle query. Delta-BiGJoinT can find all triangles in a static graph by loading each of the edges as updates to the initially empty graph. However, it is expected to be slower than an algorithm that loads the whole graph first and then finds triangles. As seen there, the COST of our two implementations are 2 and 4 cores, respectively.

5.2.2 EmptyHeaded

EmptyHeaded (EH) is a highly-optimized shared-memory parallel system evaluating subgraph queries on static graphs using GJ. EH evaluates queries using a mixture of GJ and binary join plans. The EH optimizer considers *generalized hypertree decompositions* of the query, which join multiple subsets of the relations using GJ which are then joined using binary joins. For the queries we study

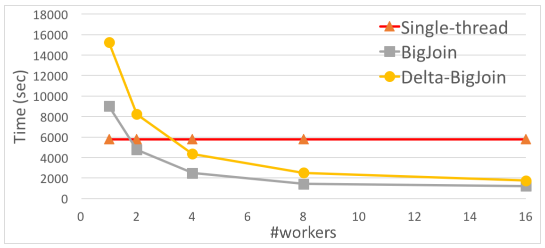


Figure 4: BiGJoin and Delta-BiGJoin counting triangles in the Twitter graph, plotted with the time it takes our single-threaded implementation. Both approaches outperform the single-threaded implementation with small number of cores, and continue to improve from there. The Delta-BiGJoin performance lags slightly behind, as it uses more complex data structures to support updates.

Table 2: Comparison against EmptyHeaded. ?-R? and ?-I? indicate runtime and index time, respectively. EmptyHeaded's absolute performance is better on a single machine. The index building time can be non-trivial.

Query	EH-R	EH-I	BiGJoinT-R	BiGJoinT-I
Triangle-LJ	1.2**s**	150.3**s**	6.5s	1.9s
Diamond-LJ	31.7**s**	150.3**s**	712.3s	1.9s
Triangle-TW	213.8**s**	4155**s**	588s	34.4s

in this paper, EH, similar to BiGJoin, uses a pure GJ plan based simply on attribute ordering. EH is highly optimized for evaluating queries on static graphs, and spends a non-trivial amount of time preparing its indices, which vary their representation in response to structural properties of the underlying data.

To guarantee a fair comparison with EH, we run its experiment using the AMI machine provided by the EH team. We started by using a machine with similar configuration as our cluster machines[3], however EH ran out of memory when running the triangle query on TW. Therefore, we used an x1.16xlarge AWS machine with 64 cores and 976 GB memory. We used TW and LJ and the triangle and diamond queries. Unfortunately, EH ran out of memory on the diamond query on TW.

Table 2 reports two metrics for both systems: (1) the runtime; and (2) the time to index the input data. As shown in the table, our implementations perform worse than EH due to our lack of specific optimizations for static datasets, such as compacting dense extension lists into bit vectors. In exchange, we are able to distribute across multiple machines and respond to changes in input, but this generality comes at a price. We are also evaluating EH's index build time and memory footprint, something EH is explicitly not optimized for, which combined with a lack of distribution limits our ability to evaluate EH on the largest datasets.

5.2.3 Arabesque

Arabesque is a distributed system specialized for finding subgraphs in large graphs. In Arabesque, each distributed worker gets an entire copy of the graph and starts extending a partition of the vertices to form larger and larger subgraphs that are called *embeddings*, equivalent to prefixes in our terminology. In Arabesque, prefixes are extended by considering the neighbors of individual vertices, rather than by intersecting the neighborhoods of multiple vertices as GJ does, and correspond to an edge-at-a-time strategy in our terminology . This puts it at a disadvantage for purely structural

[3]An r3.8xlarge AWS machine with 244 GB memory and 32 cores.

Table 3: Comparison against Arabesque. ?-R? and ?-I? indicate runtime and index time, respectively. BiGJoinT is faster and considers fewer candidate subgraphs than Arabesque.

Query	Arbsq-R	Arbsq-I	BiGJoinT-R	BiGJoinT-I
Triangle	69.0s	1.46B	**3.4s**	**38M**
4-clique	273.7s	18.7B	**21.8s**	**350M**

queries of the sort we examine (though, it more cleanly supports queries like ?subgraphs with average edge density at least 1/2?).

We used Arabesque's most recent version (1.0.1-BETA) which runs on Giraph. On our cluster, Arabesque was only able to load the LJ dataset and ran out of memory on our other datasets. We used the triangle and 4-clique queries. We used 8 machines, each running one Arabesque worker, and each worker using 16 cores. We measured both run-time and intermediate prefixes considered by the system. We used the triangle and 4-clique code provided by the authors of the system but improved the code to not output any intermediate prefixes or final output.[4] We repeated the same experiments with BiGJoinT on the same configuration, so using 8 machines with 16 Timely workers on each.

Table 3 reports the running times as well as the number of intermediate results considered, which partly explain the running times. Arabesque considers roughly 30x more prefixes as BiGJoinT, which manifests as between 10x and 20x higher running times.

5.3 Capacity and Scaling

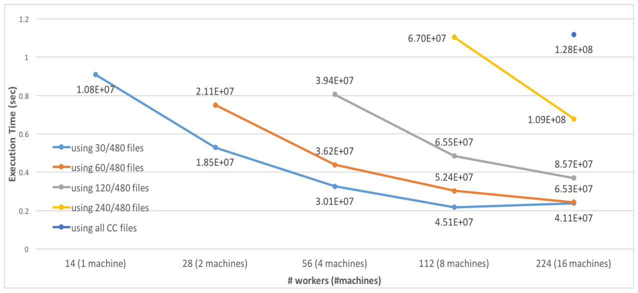
When a graph fits in the memory of a single machine, the naive parallelization strategy of replicating the graph to each machine should work very well in practice. That is why one of our primary goals was to scale to graphs (and datasets) whose collected indices do not fit in the memory of a single machine, which our algorithms achieve by using a working set that is only linear in the input relations. At the same time, very large graphs can contain prohibitively many instances for even the simplest queries. For example, we estimate that there are over 9 trillion triangles and 23 quadrillion (2.3×10^{16}) diamonds in Common Crawl[5]. Our goal is therefore not to evaluate BiGJoin when computing all subgraph instances, but Delta-BiGJoin's throughput and capacity when maintaining these queries under updates.

We use the Common Crawl dataset, which has 64B edges and is roughly 1,000x larger than LJ, 50x larger than TW, and 20x larger than UK. When each node ID requires 4 bytes, the graph requires $\perp$512GB written as a list of edges (u, v), and $\perp$ 256GB as an adjacency list. Since we index edges in both directions, our implementation requires $\perp$512GB.

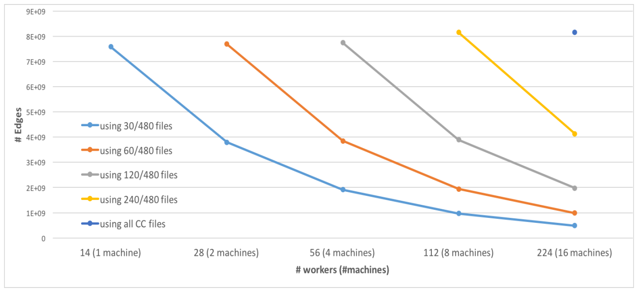
We load up various fractions of the edges in the graph, ranging from one-sixteenth to all edges, and evaluate Delta-BiGJoin on a range of one to sixteen machines. We use 14 workers per machines. So our number of cores/workers range from 14 to 224. Each subset of the graph results in a scaling curve as we increase the number of machines, and we require an increasing number of machines to start the experiments as the size of the subsets grow. For each configuration, we track the number of edges indexed on each machine, the peak memory required, and the throughput of changes (both input and output). We use the triangle query.

[4]We note that this code used *VertexInducedEmbeddings* of Arabesque, which extend prefixes by one vertex but internally by considering each edge separately.

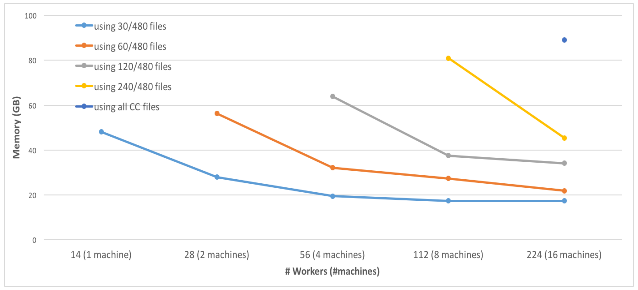
[5]These estimates are based on the number of triangles and diamonds we find per edge in our incremental experiments, which are 143 and over 368K, respectively.



(a) Execution Time; data points are the times to perform a batch of one million updates, averaged across twenty batches. The numbers by each data point report the number of output changes per second (triangles changed). The computation processes roughly 1M updates per-second, reporting between 10M and 100M changed triangles per second.



(b) Maximum Index Size per Machine, in total index tuples per machine. Index size decrease roughly linearly with additional machines at each scale.



(c) Maximum Memory per Machine, in gigabytes per machine. This peak occurs in initial index building rather than steady-state execution. The maximum does increase as we double the workers and input size, but this appears to be due to execution skew in data loading.

Figure 5: Scaling as we increase machines (and workers) and the initial graph input. Each line represents an experiment where we pre-load an indicated fraction of the CC dataset, and then perform 20 rounds of 1M input edge updates for a triangle-finding query.

Figure 5 shows our scaling results on this large graph. For each fixed subset of the graph, additional workers both improve the throughput and reduce the per-machine index size and memory requirements. The plot of maximum index size (across all machines) indicates that as we double the amount of data and number of workers, the maximum size stays roughly fixed at 8 billion, which is roughly equal to the total tuples divided by the number of machines, indicating effective balance despite some vertices with very high degree (the largest out-degree is ⊥45 million). With the exception of the smallest dataset on the largest number of machines, throughput increases and peak memory requirements decrease with further machines; however, as the work gets progressively more thin (one sixteenth of the graph spread across 224 workers) system overheads begin to emerge.

Table 4: Common Crawl experiments. Sixteen machines load 64 billion edges, index them, and track motifs in 20 batches of $10K$ random edge changes. Although the input throughput is much lower than for triangles, the *output* throughput remains relatively high at tens of millions of observed subgraph changes per second.

Query	Average Time / batch	Output Throughput	Max. Mem.
4-clique	226.378 **s**	$46,517,875$ **/s**	108.4 **GB**
Diamond	276.587 **s**	$26,681,430$ **/s**	92.6 **GB**

We also report our throughput and the peak memory required when running the diamond and 4-clique queries when loading the full graph and using 224 workers in Table 4. Here we see substantially lower throughput of input changes. For example, computing the triangles of a batch of 1M edges with 224 workers after loading the entire graph takes about 1.1 seconds (shown as the highest singleton point on the right in Figure 5a). In contrast, computing the 4-cliques on 200K edges in the same set up takes 226 seconds. However, we see a relatively similar throughput for output changes, in tens of millions, in both cases. That is, each input edge changed results in substantially more subgraph matches changed, and it is the volume of output that limits our throughput.

5.4 Generality and Specializations

In this section we show that our algorithms can employ existing optimizations from subgraph queries and multiway joins literature. In doing so, we also achieve two things. First, we compare our work to the recent SEED [37] work, which develops efficient optimizations for evaluating undirected subgraph queries in the distributed setting. Second, by implementing one of the optimizations, we demonstrate that our approach can take as input general relations instead of the binary `edge`(a_i, a_j) relations we used so far. We implement the following three optimizations:

- **Symmetry Breaking:** SEED imposes constraints on vertex IDs to break symmetries. For example, for `4-clique` query, we might constrain that $a_1 < a_2 < a_3 < a_4$. This allows finding each undirected four-clique once instead of 24 times, for each permutation of the vertices in the clique. One can be more efficient by first ordering by *degree*, and then by ID if there are ties. This is commonly accomplished by giving new IDs to the vertices so that they are ordered by degree, and edges point from vertices with lower ID to higher ID. We incorporate this optimization by transforming the input dataset, and supporting inequality constraints (which are just filters applied to intermediate prefixes).
- **Triangle Indexing:** SEED builds index structures over small non-trivial subgraphs, such as triangles. These indices provide more direct access to relevant vertex IDs reflecting multiple constraints already imposed. The ideas are similar to the recent FAQ work [3], which identifies some common subqueries in larger queries (for example, triangles in a four-clique query) and materializes these subqueries. We incorporate this optimization by first finding all the triangles in the graph and then writing these as a ternary relation `tri`(a_i, a_j, a_k). Since we support general relational queries and can index general relations, we can index `tri`(a_i, a_j, a_k) by (a_i, a_j) and provide efficient random access to vertices a_k that complete a triangle with (a_i, a_j). Using the `tri` relation, `4-clique` query simplifies to:

$$\texttt{tri}(a_1, a_2, a_3), \texttt{tri}(a_1, a_2, a_4), \texttt{tri}(a_1, a_3, a_4).$$

This rewriting reduces the complexity of the query, and results in fewer intermediate prefixes explored. We stress that this is

Table 5: Comparison with SEED, against three BiGJoin variants including several optimizations: breaking symmetry by renaming vertices by degree (-SYM) and then re-using pre-computed triangles (-TR). BiGJoin's absolute performance is comparable to optimized approaches, and improves as optimizations are applied.

Query	SEED-O	BiGJoinT	BiGJoinT-SYM	BiGJoinT-SYM-TR
4-clique	60s	54.0s	43.4s	13.3s
house	1013s	370.0s	294.3s	74.1s
5-clique	1206s	2861.1s	2153.2s	315.7s

not precisely the same optimization SEED does. SEED indexes triangles by a_1 so that full neighborhoods of each vertex is available, revealing large cliques at once. Our optimization is closer in spirit to the FAQ work, but demonstrates the utility of supporting general relations in evaluating subgraph queries.

- **Factorization:** The `house` query is amenable to a technique called *factorization* [46], which expresses parts of the query results as Cartesian products. In the `house` query, (a_2, a_3, a_4, a_5) form a clique and the missing edges are (a_1, a_4) and (a_1, a_5). We can first compute the triangle (a_2, a_3, a_4) and then perform two independent extensions to the lists of a_1 and a_5 values. As these two variables do not constrain each other, they can be left as lists rather than flattened into the list of their Cartesian product. The SEED work proposes a similar optimization (named SEED+O) in which large cliques are kept as cliques, rather than explicitly enumerating all bindings to variables. We use this optimization only for the `house` query.

Table 5 compares SEED+O (SEED with clique optimizations) measurements taken from their paper with three variations of our work: (i) vanilla BiGJoinT, (ii) BiGJoinT with symmetry breaking (BiGJoinT-SYM), and (iii) BiGJoinT with symmetry breaking and triangle indexing (BiGJoinT-SYM-TR). All of our `house` measurements also contain the factorization optimization. We used 10 machines with 16 cores, which is a cluster setup similar to the one used in the SEED paper. Table 5 demonstrates two things: (1) Our algorithms have the flexibility to employ several optimizations from prior work to become more efficient; and (2) The results of the 4-clique and 5-clique queries demonstrate that we are initially competitive with SEED using the same resources, and when incorporating some of their optimizations, we can even outperform it. We emphasize that the SEED measurements are reported from [37] rather than reproduced on identical hardware, and that our goal is not to provide evidence that our work outperforms SEED so much as that our work is able to accommodate similar optimizations.

5.5 Sensitivity to Batch Size

We finally evaluate the effects of the batch size on our algorithms. Batch size affects two aspects of our algorithms. First, very small batch sizes can impede parallelism. As an extreme example, consider finding all instances of a subgraph in a graph with a batch size of 1. Then at least initially only one worker in the cluster will do count minimization, candidate proposals, and intersections. Second, with larger batch sizes, we expect the algorithm to use more cluster memory. Therefore we expect that as batch sizes get larger, runtime improves because the algorithm can parallelize better but after we get to a large enough batch size, we expect the algorithm to have a stable runtime but use more memory.

To test this, we used the triangle query and ran Delta-BiGJoin on the UK graph using 16 workers on 1 machine and using batch sizes of 10, 100, 1K, 10K, 100K, 1M, and 10M. We first load the

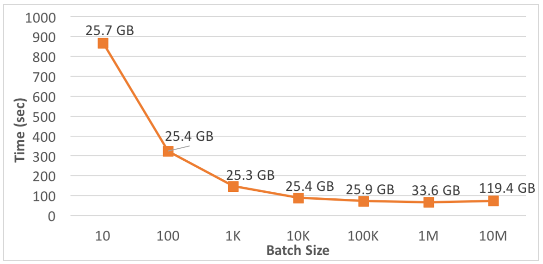


Figure 6: Effects of batch size. Note that the maximum memory usage in small batches is very close to the index size (25.1 GB).

dataset, then ran Delta-BiGJoinT using 10M edges. The results are shown in Figure 6. The numbers on top of the points indicate the maximum memory usage[6]. As shown in the figure, indeed as batch size increases the runtime initially improves and then remains the same around after batch size of 10K. As we expect, larger batch sizes lead to more cluster memory usage. We note that the increase in the memory usage is very small for batch sizes less than or equal to 100K because the intermediate data that the algorithm generates with these batch sizes is insignificant compared to the size of the input graph. Batch size is a useful parameter to balance memory usage and speedup.

6. RELATED WORK

We reviewed related work on worst-case join algorithms in Section 1.2.3. Here, we review related work in distributed join algorithms, incremental view maintenance, graph processing systems, algorithms that evaluate one-time subgraph queries, and streaming or semi-streaming algorithms for subgraph finding.

Distributed Multiway Join Algorithms: We reviewed some of the algorithms based on the Shares algorithm in Section 1.2.2. A more recent algorithm [26] has introduced a new distributed algorithm for queries that involve only two relations based on sorting the relations on their join attributes. This contrasts with the hashing approach of Shares. The algorithm runs for a small number of rounds, requires cumulative memory and communication that is as large as the actual output but does not generalize to more complex joins, e.g., involving three relations.

Reference [6] has introduced a multiround join algorithm called *GYM*, which takes as input a *generalized hypertree decomposition* (GHD) D of Q. The algorithm first computes several intermediate relations based on D in one round using Shares. Then the algorithm runs a distributed version of Yannakakis's algorithm for acyclic queries [62]. Overall the algorithm runs for $O(n)$ rounds and incurs a communication and cumulative memory cost of $O(\text{IN}^w + \text{OUT})$, where $w \geq 1$ is called the *width* of the GHD D and OUT is the actual output size. This amount of communication cost is always $O(MaxOut_Q)$ but w is only 1 for acyclic queries, so for any cyclic query the memory requirements of GYM is superlinear in IN.

Reference [28] introduces another algorithm, which we refer to as *the DBP algorithm.* DBP algorithm takes 3 rounds and takes $O(L \times \text{IN}^{DBP(L)} + \text{OUT})$, where L is a free parameter that indicates load per machine and $DBP(L)$ is called the *degree-based packing bound* of the query for load L. Similar to GYM, for any L, DBP's communication is always $O(MaxOut_Q)$ but (for any L)

[6]We measure memory usage using an operating system tool which reports a snapshot of memory usage every second instead of the average memory usage every second. This explains the small approximation and inaccuracy in the reported memory size.

the algorithm can require a cluster memory that is superlinear in IN as it computes intermediate relations that can be superlinear in IN.

Incremental View Maintenance (IVM): There is a vast body of work on incrementally maintaining views that contain selection, projection, joins, group-by-aggregates, among others. We refer the readers to reference [48] for a survey of these techniques. The overall technique of Delta-BiGJoin falls under the *algebraic technique* of representing updates to tables as delta relations and maintaining views through a set of relational algebraic queries. This approach has been extensively studied by previous work. Prior work on algebraic techniques range from addressing limitations of delta query-based techniques, e.g., when evaluating a top-k query [63], to techniques using higher-delta queries [7], e.g., delta queries of delta queries of a query. When evaluating subgraph queries, these techniques do not yield theoretically optimal results and may require materializing very large intermediate results.

The only IVM algorithm with known theoretical guarantees and the one closest to our work is the algorithm described in reference [59]. This IVM algorithm is based on the Leapfrog TrieJoin (LFTJ) worst-case optimal join algorithm. We refer to this algorithm as LFTJ-Inc. Similar to GJ, LFTJ is based on doing intersections of multiple extension sets in time proportional to the size of the minimum-size set. Unlike our description of GJ, which uses hash-based indices, LFTJ uses tries to index the prefixes of the tuples in each input relation. LFTJ-Inc uses another set of indices called *sensitivity indices* which, for each prefix p, store the set of intervals in the extensions of p such that any update to these intervals *could result* in the output of the query to change. For example, consider a join $R(a_1, a_2) \bowtie S(a_2, a_3)$. A sensitivity index for R could store $(5, [-\infty, 8))$, meaning that if a tuple with $a_1 = 5$ and $a_2 \in [-\infty, 8)$ is added or deleted from R, this update could change the output of the join. Using the sensitivity indices, LFTJ-Inc ?fixes? the necessary intersections to compute the outputs that have changed. Between any two updates, LFTJ-Inc maintains query results in time proportional to what the author calls the *trace edit distance* of running LFTJ on the relations before and after the update. That is, the author analyzes the ?trace? of LFTJ, which is the set of iterator operations that the algorithm does, on inputs before and after the update, and conclude that the work that LFTJ-Inc does to maintain the query result is proportional to the amount of work one would need to ?fix? LFTJ's iterator operators before the update. We note that this is a stronger theoretical guarantee than our Delta-BiGJoin's worst-case optimality under insertion-only workloads. In particular a trace edit distance guarantee implies that LFTJ-Inc is worst-case optimal under insertion-only workloads. However unlike Delta-BiGJoin, which requires indices linear in the input sizes, the sensitivity indices could be as large as the AGM bound of the query (so super-linear in the size of the input) for some queries and thus require a prohibitively large amount of memory.

Systems and Algorithms For One-time Subgraph Queries: Although they significantly differ in their graph storage, algorithms, and optimizations, existing systems that evaluate general subgraph queries are based on the edge-at-a-time strategy, unlike BiGJoin's vertex-at-a-time strategy. We reviewed Arabesque, EmptyHeaded, and SEED in Sections 5.2.3, 5.2.2, and 5.4. We review other work below.

PSgL [50]: PSgL is a subgraph enumeration system that is built on top of Giraph [21]. PSgL picks an order of the vertices (i.e., attributes), say $a_1, ..., a_m$ in Q, called a *traversal order*. It starts with candidate partial matches G_{psi} for a_1, then extends each G_{psi} to all neighbors of a_1 in Q (not just a_2). When matching a_j, the existence of edges (a_i, a_j) edges for $i < j$ will be checked and if they exist a_j will be extended to all neighbors $a_k > a_j$. This is effectively an edge-at-a-time strategy. The paper presents techniques for picking good traversal orders, balancing workload among workers, and breaking internal symmetries in queries over undirected graphs, which can complement our algorithms on undirected graphs as well.

TrinityRDF [65] and Spartex [1] are two distributed RDF engines that can evaluate any SPARQL [51] query. SPARQL queries can express any subgraph query, so both of these systems can evaluate general subgraph queries. The optimizers of both systems use edge-at-a-time strategies although they use different techniques to choose edge extension plans.

There are several other work, such as references [8, 47, 61] that describe data distribution techniques or other optimizations to find subgraphs in a distributed setting, using black box or naive subgraph finding algorithms as subroutines. We do not review these references here. There are also several studies that study evaluation of a single specific query, e.g., the triangle query [18, 23], which we do not review here.

Streaming and Semi-streaming Algorithms: Several works study variants of continuous subgraphs queries in a streaming or semi-streaming setting, i.e., in which the algorithms can use slightly superlinear space. A thorough review of these works is beyond the scope of our work. Example studies include those that focus on triangle finding and variants [12, 53]. Many works in this area focus on approximating the counts of different subgraphs and instead of enumerating, which is the problem we study in this paper.

7. FUTURE WORK

We outline three broad directions for future work. First is studying the extent of workload imbalance in real-world graphs and designing more efficient workload-balanced versions of BiGJoin. Although BiGJoin-S is theoretically skew-resilient, in our preliminary implementation of the algorithm, we observed that its overheads were higher than its benefits. Better understanding the effects of skew, when it hurts BiGJoin and DeltaBiGJoin's performance, and how to effectively guard against it is an interesting future direction. Second, we are interested in studying how to utilize internal symmetries of queries during query evaluation. For example, when evaluating the 4-clique query, some of the delta-queries, e.g., $dQ2$ and $dQ3$, compute the same P_2 and P_3 prefixes due to internal symmetry of the query. An interesting future direction is to automatically exploit such symmetries to share computations across multiple dataflows of delta queries. Finally, from a theoretical perspective, an interesting direction is designing practical algorithms that have stronger guarantees than worst-case optimality. A stronger than worst-case optimality guarantee could be optimality in terms of *certificate complexity*, which is achieved by the recent serial *Minesweeper* algorithm for multiway joins in terms of computation cost [44]. At a high-level, certificate complexity captures the smallest proof size to verify that the output is correct and is a strictly stronger notion than worst-case optimality.

8. ACKNOWLEDGEMENTS

We would like to thank Lori Paniak for assisting with numerous systems issues. We would also like to thank Michael Isard and Chris Ré for early discussions that resulted in the first implementation of BiGJoin.

9. REFERENCES

[1] Ibrahim Abdelaziz, Razen Harbi, Semih Salihoglu, Panos Kalnis, and Nikos Mamoulis. SPARTex: A Vertex-Centric

Framework for RDF Data Analytics (Demonstration). *PVLDB*, 8(12):1880?1883, 2015.

[2] Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. EmptyHeaded: A Relational Engine for Graph Processing. In *SIGMOD*, 2016.

[3] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. Faq: Questions asked frequently. In *PODS*, 2016.

[4] F. N. Afrati, A. D. Sarma, S. Salihoglu, and J. D. Ullman. Upper and Lower Bounds on the Cost of a Map-Reduce Computation. *PVLDB*, 6(4):277?288, 2013.

[5] F. N. Afrati and J. D. Ullman. Optimizing Multiway Joins in a Map-Reduce Environment. *TKDE*, 2011.

[6] Foto N. Afrati, Manas R. Joglekar, Christopher Ré, Semih Salihoglu, and Jeffrey D. Ullman. GYM: A multiround distributed join algorithm. In *ICDT*, 2017.

[7] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views. *PVLDB*, 5(10):968?979, 2012.

[8] Shaikh Arifuzzaman, Maleq Khan, and Madhav Marathe. PATRIC: A Parallel Algorithm for Counting Triangles in Massive Networks. In *CIKM*, 2013.

[9] A. Atserias, M. Grohe, and D. Marx. Size Bounds and Query Plans for Relational Joins. *SIAM Journal on Computing*, 2013.

[10] P. Beame, P. Koutris, and D. Suciu. Communication Steps for Parallel Query Processing. In *PODS*, 2013.

[11] P. Beame, P. Koutris, and D. Suciu. Skew in Parallel Query Processing. In *PODS*, 2014.

[12] Luca Becchetti, Paolo Boldi, Carlos Castillo, and Aristides Gionis. Efficient Semi-streaming Algorithms for Local Triangle Counting in Massive Graphs. In *SIGKDD*, 2008.

[13] Blakeley, Jose A. and Larson, Per-Ake and Tompa, Frank Wm. Efficiently Updating Materialized Views. *SIGMOD Record*, 15(2), June 1986.

[14] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink^{TM}: Stream and Batch Processing in a Single Engine. *IEEE Data Engineering Bulletin*, 38, 2015.

[15] Jiefeng Cheng, Jeffrey Xu Yu, Bolin Ding, Philip S. Yu, and Haixun Wang. Fast Graph Pattern Matching. In *ICDE*, 2008.

[16] Sutanay Choudhury, Lawrence B. Holder, George Chin Jr., Khushbu Agarwal, and John Feo. A Selectivity based approach to Continuous Pattern Detection in Streaming Graphs. In *EDBT*, 2015.

[17] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[18] Danny Dolev, Christoph Lenzen, and Shir Peled. ?Tri, Tri Again?: Finding Triangles and Small Subgraphs in a Distributed Setting. In *DISC*, 2012.

[19] Gary William Flake, Steve Lawrence, C. Lee Giles, and Frans M. Coetzee. Self-Organization and Identification of Web Communities. *Computer*, 35(3), March 2002.

[20] Jun Gao, Chang Zhou, Jiashuai Zhou, and Jeffrey Xu Yu. Continuous Pattern Detection Over Billion-edge Graph Using Distributed Framework. In *ICDE*, 2014.

[21] Apache Incubator Giraph. http://incubator.apache.org/giraph/.

[22] Gupta, Ashish and Mumick, Inderpal Singh and Subrahmanian, V. S. Maintaining Views Incrementally. *SIGMOD Record*, 22(2), 1993.

[23] Gupta, Pankaj and Satuluri, Venu and Grewal, Ajeet and Gurumurthy, Siva and Zhabiuk, Volodymyr and Li, Quannan and Lin, Jimmy. Real-time Twitter Recommendation: Online Motif Detection in Large Dynamic Graphs. *PVLDB*, 7(13):1379?1380, 2014.

[24] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. Turboiso: Towards Ultrafast and Robust Subgraph Isomorphism Search in Large Graph Databases. In *SIGMOD*, 2013.

[25] Huahai He and Ambuj K. Singh. Graphs-at-a-time: Query Language and Access Methods for Graph Databases. In *SIGMOD*, 2008.

[26] Xiao Hu, Yufei Tao, and Ke Yi. Output-optimal Parallel Algorithms for Similarity Joins. In *PODS*, 2017.

[27] Mohammad Husain, James McGlothlin, Mohammad M. Masud, Latifur Khan, and Bhavani M. Thuraisingham. Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing. *TKDE*, 23(9), 2011.

[28] Manas Joglekar and Christopher Ré. It's All a Matter of Degree. *Theory of Computing Systems*, Sep 2017.

[29] Dataflow Join. https://github.com/frankmcsherry/timely-dataflow.

[30] K. Ammar, and F. McSherry, and S. Salihoglu, and M. Joglekar. Distributed Evaluation of Subgraph Queries Using Worst-case Optimal and Low-Memory Dataflows. *CoRR*, abs/1802.03760, 2018.

[31] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. Graphflow: An Active Graph Database. In *SIGMOD*, 2017.

[32] Paraschos Koutris, Paul Beame, and Dan Suciu. Worst-Case Optimal Algorithms for Parallel Query Processing. In *ICDT*, 2016.

[33] Longbin Lai, Lu Qin, Xuemin Lin, and Lijun Chang. Scalable subgraph enumeration in mapreduce: A cost-oriented approach. *The VLDB Journal*, 26(3):421?446, 2017.

[34] Andrea Lattuada, Frank McSherry, and Zaheer Chothia. Faucet: A User-level, Modular Technique for Flow Control in Dataflow Engines. In *BeyondMR*, 2016.

[35] The Laboratory for Web Algorithmics. http://law.dsi.unimi.it/datasets.php.

[36] Jure Leskovec and Andrej Krevl. SNAP: Stanford Network Analysis Project. `http://snap.stanford.edu`, June 2014.

[37] Longbin Lai and Lu Qin and Xuemin Lin and Ying Zhang and Lijun Chang. Scalable distributed subgraph enumeration. *PVLDB*, 10(3):217?228, 2016.

[38] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-scale Graph Processing. In *SIGMOD*, 2010.

[39] Frank McSherry, Michael Isard, and Derek G. Murray. Scalability! But at What Cost? In *HOTOS*, 2015.

[40] Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A Timely Dataflow System. In *SOSP*, 2013.

[41] Neo4j Home Page. http://neo4j.com/.

[42] Thomas Neumann and Gerhard Weikum. The RDF-3X Engine for Scalable Management of RDF Data. *The VLDB Journal*, 19(1):91?113, 2010.

[43] H. Ngo, C. Ré, and A. Rudra. Skew Strikes Back: New Developments in the Theory of Join Algorithms. *SIGMOD*, 2014.

[44] Hung Q. Ngo, Dung T. Nguyen, Christopher Re, and Atri Rudra. Beyond Worst-case Analysis for Joins with Minesweeper. In *PODS*, 2014.

[45] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case Optimal Join Algorithms. In *PODS*, 2012.

[46] Dan Olteanu and Maximilian Schleich. Factorized databases. *SIGMOD Record*, 45(2), September 2016.

[47] Ha-Myung Park, Sung-Hyon Myaeng, and U. Kang. PTE: enumerating trillion triangles on distributed systems. In *SIGKDD*, 2016.

[48] Rada Chirkova and Jun Yang. Materialized Views. *Foundations and Trends in Databases*, 4(4), 2012.

[49] Rust. https://www.rust-lang.org.

[50] Yingxia Shao, Bin Cui, Lei Chen, Lin Ma, Junjie Yao, and Ning Xu. Parallel Subgraph Listing in a Large-scale Graph. In *SIGMOD*, 2014.

[51] SPARQL Specification. http://www.w3.org/TR/rdf-sparql-query.

[52] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. Efficient Subgraph Matching on Billion Node Graphs. *PVLDB*, 5(9):788?799, 2012.

[53] Kanat Tangwongsan, A. Pavan, and Srikanta Tirthapura. Parallel Triangle Counting in Massive Streaming Graphs. In *CIKM*, 2013.

[54] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. Arabesque: A System for Distributed Graph Mining. In *SOSP*, 2015.

[55] Timely Dataflow. https://github.com/frankmcsherry/timely-dataflow.

[56] Titan Home Page. http://thinkaurelius.github.io/titan/.

[57] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. Storm@Twitter. In *SIGMOD*, 2014.

[58] Todd L. Veldhuizen. Leapfrog Triejoin: a worst-case optimal join algorithm. *CoRR*, abs/1210.0481, 2012.

[59] Todd L. Veldhuizen. Incremental Maintenance for Leapfrog Triejoin. *CoRR*, abs/1303.5313, 2013.

[60] Web Data Commons. http://www.webdatacommons.org/hyperlinkgraph.

[61] Bin Wu and YunLong Bai. An Efficient Distributed Subgraph Mining Algorithm in Extreme Large Graphs. In *AICI*, 2010.

[62] Mihalis Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, pages 82?94, 1981.

[63] Ke Yi, Hai Yu, Jun Yang, Gangqiang Xia, and Yuguo Chen. Efficient Maintenance of Materialized Top-k Views. In *ICDE*, 2003.

[64] Zaharia, M. and Chowdhury, M. and Franklin, M. J. and Shenker, S. and Stoica, I. Spark: Cluster Computing with Working Sets. In *HotCloud*, 2010.

[65] Zeng, Kai and Yang, Jiacheng and Wang, Haixun and Shao, Bin and Wang, Zhongyuan. A Distributed Graph Engine for Web Scale RDF Data. *PVLDB*, 6(4):265?276, 2013.

[66] Lei Zou, Lei Chen, and M. Tamer Özsu. Distance-join: Pattern Match Query in a Large Graph Database. *PVLDB*, 2(1):886?897, 2009.

[67] Lei Zou, Jinghui Mo, Lei Chen, M. Tamer Özsu, and Dongyan Zhao. gStore: Answering SPARQL Queries via Subgraph Matching. *PVLDB*, 4(8):482?493, 2011.