RStream: Marrying Relational Algebra with Streaming for Efficient Graph Mining on A Single Machine

Kai Wang, ***UCLA;*** **Zhiqiang Zuo,** ***Nanjing University;*** **John Thorpe,** ***UCLA;***
Tien Quang Nguyen, ***Facebook;*** **Guoqing Harry Xu,** ***UCLA***

https://www.usenix.org/conference/osdi18/presentation/wang

This paper is included in the Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18).

October 8–10, 2018 • Carlsbad, CA, USA

ISBN 978-1-931971-47-8

Open access to the Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.

RStream: Marrying Relational Algebra with Streaming for Efficient Graph Mining on A Single Machine*

Kai Wang† Zhiqiang Zuo‡ John Thorpe† Tien Quang Nguyen§ Guoqing Harry Xu†
UCLA† *State Key Laboratory for Novel Software Technology* *Facebook*§
Nanjing University‡

Abstract

Graph mining is an important category of graph algorithms that aim to discover structural patterns such as cliques and motifs in a graph. While a great deal of work has been done recently on *graph computation* such as PageRank, systems support for scalable graph mining is still limited. Existing mining systems such as Arabesque focus on distributed computing and need large amounts of compute and memory resources.

We built RStream, the *first single-machine, out-of-core* mining system that leverages disk support to store intermediate data. At its core are two innovations: (1) a rich programming model that exposes relational algebra for developers to express a wide variety of mining tasks; and (2) a runtime engine that implements relational algebra efficiently with *tuple streaming*. A comparison between RStream and four state-of-the-art distributed mining/Datalog systems — Arabesque, ScaleMine, DistGraph, and BigDatalog — demonstrates that RStream outperforms all of them, running on a 10-node cluster, *e.g.*, by at least a factor of **1.7**×, and can process large graphs on an inexpensive machine.

1 Introduction

There are two major types of analytical problems over large graphs: *graph computation* and *graph mining*. Graph computation includes a set of problems that can be represented through linear algebra over an adjacency matrix based representation of the graph. As a typical example of graph computation, PageRank [52] can be modeled as iterative sparse matrix and vector multiplications. Due to their importance in information retrieval and machine learning, graph computation problems have been extensively studied in the past decade; practical solutions have been implemented in a wide variety of graph systems [31, 27, 30, 33, 43, 39, 63, 48, 85, 58, 75, 83, 34, 57, 69, 84], most of which follow the "think like a vertex" programming paradigm pioneered by Pregel [46]. These systems have been highly optimized for locality, partitioning, and communication in order to deliver efficiency and scalability for processing very large graphs.

While this programming model makes it easy for developing computation algorithms, it is *not* designed for mining algorithms that aim to discover complex *structural patterns* of a graph rather than perform value computations. Fitting such algorithms into this model requires significant reformulation. For many mining tasks such as frequent subgraph mining (FSM), their patterns are not known *a priori*; hence, it is impossible to express these tasks using a vertex-centric model.

There is a body of work that uses declarative models to solve mining problems. Representative examples are Datalog [2, 40, 73, 62, 61], Arabesque [66], ScaleMine [4], or DistGraph [65]. For instance, due to its support for relational algebra, Datalog provides simple interfaces for developing mining tasks [40, 61]. A Datalog program for Triangle Counting, for example, needs only the following two lines of code, with *R* representing the relation of edges and *U* representing a new relation of triangles:

```
1 U(a,b,c) <- R(a,b), R(b,c), R(a,c)
2 count U(a,b,c)
```

However, Datalog's support for graph mining is rather limited since the declarative nature of its programming model dictates that only mining algorithms whose patterns are known *a priori* can be expressed by Datalog. Arabesque is a Giraph-based graph mining system that presents developers a view of "embeddings". Embeddings are subgraphs that developers can easily check to find structural patterns. Using a *filter-process* programming model, Arabesque provides full support for developing a broad set of mining algorithms. For example, Arabesque enumerates all possible subgraphs and invokes the user-defined `filter` function on each subgraph. The user logic in the function determines whether the given subgraph is an instance of the specified motif (for motif counting) or turns the subgraph into a canonical form to count the number of instances of the form (for FSM).

Specialized systems have been developed for FSM due to its broad applications. Examples are ScaleMine [4] and DistGraph [65], but these systems do not work for other mining algorithms such as Triangle Counting or Cliques.

1.1 Problems with State-of-the-Art Systems

Typical mining workloads are memory-intensive. Even simple mining algorithms can generate an enormous amount of intermediate data, which cannot fit into the main memory of any single machine. Early single-machine techniques such as gSpan [78] and GraMi [29] can analyze only small graphs as they are fundamen-

*Work was done when all authors were with UC Irvine.

tally limited by the size of the main memory of the machine on which they run. Recent mining tools such as Arabesque [66], ScaleMine [4], and DistGraph [65] are distributed systems — they leverage distributed memory resources to store intermediate mining data.

Mining Systems Distributed mining systems have several drawbacks that significantly impact their practicality. First, they commonly suffer from large startup and communication overhead. For small graphs, it is difficult for the startup/communication overhead to get amortized over the processing. For example, when FSM was executed on Arabesque to process a small graph (CiteSeer, with 4K edges) on a 10-node cluster, it took Arabesque 35 seconds to boost the system and load the graph, while executing the algorithm itself only took 3 seconds.

Second, in order to scale to large graphs, mining systems often need enterprise clusters with large amounts of memory. This is because the amount of intermediate data for a typical mining algorithm grows exponentially with the size of the graph. For example, built on top of MPI, a recent mining system DistGraph [65], using 128 IBM BlueGene/Q compute nodes, could only run 3-FSM with support = 25000[1] on a million-edge graph — even on such a small graph, the computation requires a total of 128 $\times$ 256 = 32,768GB memory. Obviously, not all users have access to such enterprise clusters. Even if they do, running a simple mining algorithm on a relatively small graph does not seem to justify very well the cost of blocking hundreds or even thousands of machines for several hours.

When many compute nodes are employed primarily to offer memory, their CPU resources are often underutilized. Unlike the "think-like-a-vertex" computation algorithms that are amenable to the bulk synchronous parallel (BSP) model, mining workloads are not massively parallel by nature — a mining algorithm enumerates subgraphs of increasing sizes to find those that match a pattern; finer-grained partitioning of the input graph to exploit parallelism often does not scale well with increased CPU resources because subgraphs often cross partitions, creating great numbers of dependencies between tasks.

Load balancing in a distributed mining system is another major challenge. Algorithms such as FSM have dynamic working sets. Their search space is often unknown in advance and it is thus hard to partition the graph and distribute the workload appropriately before the execution. When we executed FSM on DistGraph, we observed that some nodes had high memory pressure and ran out of memory in several minutes while the memory usage of some other nodes was below 10%.

[1] 25000 is a very large frequency threshold for FSM — a subgraph is considered frequent only if its frequency exceeds this threshold. The smaller the support is, the more computation is needed.

Dataflow/Datalog Systems The major problem of dataflow systems or Datalog engines is that they do not have a programming model flexible enough for expressing complex graph mining algorithms. For example, for mining frequent subgraphs whose structures have to be dynamically discovered, none of the Datalog systems can directly support it.

A Strawman Approach A possible way to develop a more cost-effective graph mining system is to add simple support for data spilling in an existing system (such as Arabesque or DistGraph) rather than developing a new system from scratch — if intermediate data can be swapped between memory and disk, the amount of compute resources needed may be significantly reduced. In fact, data spilling is already implemented in many existing systems: Arabesque is based on Giraph, which places on disk partitions that do not fit in memory; BigDatalog is based on Spark, which spills data throughout the execution. However, generic data spilling does not work well due to the lack of semantic information of how each data partition is used in the program.

To understand whether semantics-agnostic data spilling is effective, we ran transitive closure computation on BigDatalog over the MiCo graph [29] (with 1.1M edges) using a cluster of 10 nodes each with 32GB memory. Despite Spark's disk support, which spilled a total of 6.006GB of data to disk across all executors, BigDatalog still crashed in 1375 seconds.

1.2 Challenges and Contributions

To address the shortcomings of the existing mining tools, we developed RStream, the *first disk-based, out-of-core* system that supports efficient mining of large graphs. Our key insight is consistent with the recent trend on building single-machine graph computation systems [39, 58, 75, 70, 45, 83, 8, 81] — given the increasing accessibility of high-volume SSDs, a disk-based system can satisfy the large storage requirement of mining algorithms by utilizing disk space available in modern machines; yet it does not suffer from any startup and communication inefficiencies that are inherent in distributed computing.

Building RStream has two major challenges. *The first challenge* is how to provide a programming interface rich enough to support a wide variety of mining algorithms. The design of RStream's programming model is inspired from both Datalog and the gather-apply-scatter (GAS) model used widely in the existing computation systems [30, 39, 58]. On the one hand, the relational operations in Datalog enable the composition of structures of smaller sizes into a structure of a large size, making it straightforward for the developer to program mining algorithms. On the other hand, GAS is a powerful programming model that supports iterative graph processing with a well-defined termination semantics. To enable

easy programming of mining algorithms with and without statically-known structural patterns, we propose a novel programming model (§3), referred to as *GRAS*, which adds relational algebra into GAS. We show, with several examples, that under GRAS, many mining algorithms, including FSM, Triangle and Motif Counting, or Clique, can all be easily developed with less than 80 lines of code.

The second challenge is how to implement relational operators (especially join) efficiently for graphs. Since join is expensive, its efficiency is critical to the system performance. Instead of treating edges and vertices generically as relational tables as in Datalog, we take inspirations from graph computation systems to leverage the domain knowledge in graphs. In particular, we are inspired by recent systems (*e.g.*, X-Stream [58] and GridGraph [85]) that use streaming to reduce I/O costs.

The scatter/gather phase in these systems loads vertices into memory and *streams in* edges/updates to generate updates/new vertex values. The insight behind streaming is that since the number of edges/updates is much larger than the number of vertices for a graph, edge streaming provides efficiency by sequentially accessing edge data from disk (as edges are sequentially read but not stored in memory) and randomly accessing vertex data held in memory. Streaming essentially provides an *efficient, locality-aware join implementation*. RStream leverages this insight (§4) to implement relational operations.

1.3 Summary of Results

We have implemented RStream and made it publicly available at https://github.com/rstream-system. We evaluated it using 6 mining algorithms over 6 real-world graphs. With a rich programming model and an efficient implementation of the model using streaming, RStream, running on a single machine with 32GB memory and 5.2TB disk space, outperformed 4 state-of-the-art distributed mining and Datalog systems — Arabesque, ScaleMine, DistGraph, and BigDatalog by at least a factor of **1.7**×, when they each ran on a 10-node cluster.

These results do not necessarily suggest that RStream has better scalability than a distributed system, which may be able to scale to larger graphs if sufficient memory is provided. However, RStream is indeed a better choice if a user has only a limited amount of computing resources, since its disk requirement is easier to fulfill and yet it can scale to large enough real-world graphs.

2 Background and Overview

Since RStream builds on streaming, we provide a brief discussion of this idea and the related systems. We then use a concrete example to overview RStream's design.

2.1 Background

RStream's tuple streaming idea is inspired by a number of prior works, and in particular, the X-Stream graph computation system [58] that uses edge streaming to reduce I/O. X-Stream partitions a graph into *streaming partitions* based on vertex intervals. Each streaming partition consists of (1) a vertex set, which contains vertices in a logical interval and their values, (2) an edge set, containing edges whose *source vertices* are in its vertex set, as well as (3) an update set, containing updates over the edges whose *destinations* are in its vertex set. X-Stream's design is based on the GAS model. It first conducts the scatter phase, which, for each partition, loads its vertex set into memory and streams in edges from the edge set to generate updates (*i.e.*, propagate the value of the source to the destination for each edge).

The update over each edge is shuffled into the update set of the partition containing the destination of the edge. This enables an important *locality property* — for each vertex in a streaming partition, updates from all of its incoming edges are present in the update set of the same partition. The property leads to an efficient gather-apply phase, because vertex computation can be performed *locally* in each partition without accessing other partitions.

The following gather-apply phase loads the vertex set for each partition into memory, streams in updates from the update set of the partition, and invokes the user vertex function to compute a new value for each vertex. During scatter and gather-apply, edges/updates are streamed in *sequentially* from disk while in-memory vertices are randomly accessed to compute vertex values. This design leads to high performance because the number of edges is much larger than that of vertices.

2.2 RStream Overview

We use X-Stream's partitioning technique as the starting point to build RStream. RStream adds a number of relational (R) phases into the GAS programming/execution model, resulting in a new model referred to as *GRAS* in the paper. To accommodate the relational semantics, RStream's programming interface treats vertex set, edge set, and update set all as relational tables. From this point on, we use *vertex table*, *edge table*, and *update table* to refer to these sets.

Since edges do not carry data, the edge table has a fixed schema of two columns (source and destination) – its numbers of rows and columns never change. Both the vertex and update table may change their schema during computation. For example, the vertex table, initially with two columns (ID and initial value), may grow to have multiple columns (due to joins) where each vertex corresponds to a row with multiple elements; an example can be found shortly in Figure 2. In the update table, one vertex may have multiple corresponding rows since the vertex can receive values from multiple edges. The update table can also change due to joins. Tuples in these tables remain *unsorted* throughout the execution.

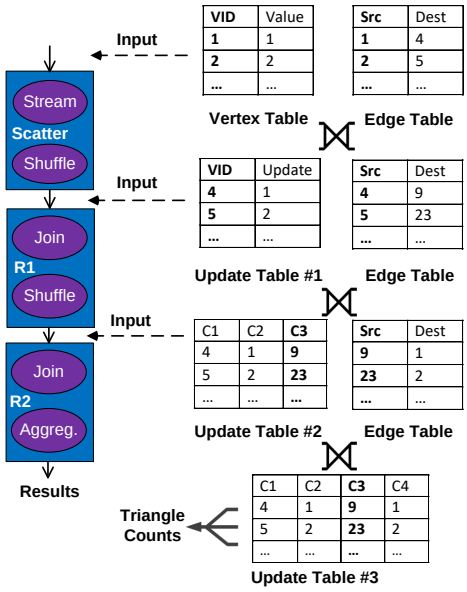


Figure 1: A Triangle Counting example in RStream; highlighted in each table is its key column. For each table, only a small number of relevant tuples are shown.

RStream first conducts scatter to generate the update table. Similarly to X-Stream, the vertex table is loaded into memory in this phase; edges are streamed in and updates are shuffled. The user-defined relational phases are then performed over the update table and the edge table in each streaming partition. What and how many relational phases are needed is programmable. These relational phases produce a new set of update tables, which will be fed as input to the gather-apply phase to compute new tuples for each vertex. The new tuples are saved into the vertex table at the end of an iteration.

Example We use Triangle Counting as an example. Although Triangle Counting is also supported by many computation systems, it is a typical structure mining algorithm that has a simple logic and thus provides a good introductory example. Figure 1 depicts the dataflow of the computation while the RStream code is shown in Figure 2. The execution contains three phases: scatter and two additional relational phases. The scatter phase has the same semantics as in X-Stream — the vertex table is loaded into memory; edges are streamed in and updates are shuffled. The relational phases are user-defined and their implementations are shown in Line 13–49. RStream lets the developer register the dataflow by connecting phases (Line 4–8). Each node on the dataflow graph is a `Phase` object. Class `TCScatter` is a scatter phase with a standard semantics; its definition is omitted for brevity. The developer adds relational phases into the dataflow.

Initially, we let the value of each vertex be its own ID (shown in the vertex table in Figure 1). The scatter

```
1  class TriangleCounting : public Application {
2    void run(Engine e){
3      /*Create a dataflow graph*/
4      TCScatter s;
5      e.set_start(&s);
6      R1 r1; R2 r2;
7      e.insert_phase(r1, s);
8      e.insert_phase(r2, r1);
9      e.run();
10   }
11 };
12
13 class R1 : public RPhase{
14   /*Called from join: only keep such <a, b, c>
         that b < a < c */
15   bool filter(Tuple t1, Tuple t2){
16     if(t1.element(1) > t1.element(0))
17        return FALSE;
18     if(t2.element(0) > t2.element(2))
19        return FALSE;
20     return TRUE;
21   }
22
23   /*Called from join: new key column*/
24   int new_key(){
25     return 2; /* set 'C3' as key*/
26   }
27
28   /*The main entry point*/
29   void execute(StreamingPartition sp){
30     UpdateTable ut = sp.update_table;
31     ut.set_key(0); //set 'VID' as key
32     EdgeTable et = sp.edge_table;
33     /*Join ut with et; et's key is 'Src';
           generated tuples are shuffled on
           new_key*/
34     super::join(sp);
35   }
36 };
37
38 class R2: public RPhase{
39   bool filter(Tuple t1, Tuple t2){
40    if(t2.element(1) != t1.element(0))
41      return FALSE;
42    return TRUE;
43  }
44
45  void execute(StreamingPartition sp){
46    super::join(sp);
47    super::aggregate(sp, COUNT, null);
48  }
49 };
```

Figure 2: Triangle counting in RStream.

phase streams edges in from the edge table. For each edge e, RStream retrieves the tuple from the vertex table corresponding to e's source vertex and produces an update based on it. In the beginning, since each vertex has only one value (*i.e.*, its own ID), the update over each edge e is essentially e's source vertex ID. These updates are shuffled into the update tables (#1 in Figure 1) across the streaming partitions. Specifically, the update for e, which is e's source vertex ID, goes into the update table of the partition that contains e's destination.

The program has two relational phases R1 and R2. R1 essentially joins all such edges (a, b) with (b, c) to produce relation (a, b, c), while R2 joins (a, b, c) with (c, a) to detect triangles. To implement R1, the developer invokes the `join` function defined in class `RPhase`. This function takes a streaming partition (sp) as input and implements a *fixed semantics* of joining sp's update table (ut) with

its own edge table (*et*) on their key columns. The key column for the update table can be set by using `set_key`, while the edge table always uses the source vertex column as its key column.

Joining the two tables also conducts (1) filtering, (2) tuple reshuffling, and (3) updating of *sp*'s update table. Filtering uses the user-defined `filter` function (Line 15–21). Tuples produced by this join form the new update table of each partition. The user can override the function `new_key` to specify the key column of this new table. If the new key is different than the current key of the update table, the generated tuples need to be reshuffled across partitions — each tuple is sent to the partition that contains the key element of the tuple.

For instance, the invocation of `join` in Line 34 joins the update table #1 with the edge table in Figure 1 using the filter defined in Line 15 of Figure 2. Specifically, it joins (a, b) with (b, c) and produces tuples of the form (a, b, c). The `filter` function specifies that we select only rows (a, b, c) with $b < a < c$, to filter out duplicates. Next, since function `new_key` specifies C3 as the new key column, each generated (a, b, c) will be shuffled to the streaming partition whose vertex table contains vertex ID c. This provides a benefit of locality for the next join, which will be performed on column C3 of the update table and Src of the edge table. Finally, the update table of each streaming partition *sp* is updated to the new table containing such (a, b, c) tuples.

The second invocation of `join` in Line 46 joins the update table resulting from R1 (*i.e.*, #2 in Figure 1) and the same edge table with the filtering condition defined in Line 39–43. The goal of this join is to find tuples of the form (a, b, c) and (c, b) to confirm that (a, b, c) indeed forms a triangle. After R2, the new update table (#3) in each partition contains triangles that can be counted using the aggregation function `aggregate` (Line 47). Here we do not need a cycle in the dataflow graph and the algorithm ends after the two joins.

Since the example aims to count the total number of triangles, a gather-apply phase is not needed. However, if one wants to count the number of distinct triangles for each vertex, an additional gather-apply phase would be required to stream in triangle tuples from the update table #3 and gather them based on their key element to compute per-vertex triangle counts. The gather phase essentially implements a group-by operation. More details can be found in §3.

Observation on Expressiveness We make several observations with the example. The first one is the expressiveness of the GRAS model. Joins performed by the relational phases over the update table and the edge table enable us to "grow" existing subgraphs we have found (*i.e.*, stored in the update table) with edges (*i.e.*, stored in the edge table) to form larger subgraphs. This is the key ability enabling Datalog and Arabesque to express mining algorithms. Our GRAS model is *as expressive as Arabesque's filter-process model* – the `filter` function in a relational phase achieves the same functionality as Arabesque's filter while Arabesque's embedding enumeration and processing can be achieved with relational joins between the update and edge tables.

Clearly GRAS is *more expressive than Datalog* – the combination of dataflow cycles and relational joins allows RStream to express algorithms that aim to discover structures whose shapes cannot be described *a priori*, such as subgraph mining.

A surprising side effect of building our programming model on top of GAS is that RStream can also support graph computation algorithms and even the transitive closure computation, which none of the existing mining systems can support. Developing computation algorithms such as PageRank is easy — they need the traditional scatter, gather, and apply, rather than any relational phases.

Observation on Efficiency The locality property of X-Stream is preserved in RStream. Tuple shuffling performed at the end of each join (based on `new_key`) makes it possible for joins to occur locally within each streaming partition *sp*. This is because (1) all the update tuples whose key column contains a vertex ID belonging to *sp* have been shuffled into the *sp*'s update table, and (2) all the edges whose source vertex (*i.e.*, key column) belonging to *sp* are already in *sp*'s edge table. Random accesses may occur only during shuffling; accesses are conducted sequentially in all other phases. Our join is implemented efficiently by tuple streaming (§4) – since the update table is often orders of magnitude larger than the edge table, RStream loads the edge table in memory and streams in tuples from the update table.

Limitation A limitation of RStream is that it currently assumes a static graph and does not deal with graph updates without restarting the computation. Hence, it cannot be used for interactive mining tasks at this moment.

3 Programming Model

This section provides a detailed description of RStream's programming model. Figure 3 shows the data structures and interface functions provided by RStream. An RStream program is made up of a dataflow graph constructed by the developer. The main entry of an RStream application is a subclass of `Application`, which the developer needs to provide to implement a given algorithm.

Adding Structural Info A special function to be implemented in an application is `need_structure`, which, by default, returns FALSE. As shown in Figure 1, each join grows an existing group of vertices with a new edge, generating a new (larger) structure. However, since each tuple currently only contains vertex IDs, the *structural*

information of these vertices (*i.e.*, edges connecting them) is missing. This will not create a problem for applications such as Triangle Counting because the structure of a triangle is known *a priori*. However, for applications like FSM, the shape of a frequent subgraph needs to be discovered dynamically. Missing structural information in tuples would create two challenges for these applications. First, tuples with identical elements may represent different structures. For example, a tuple $\langle 1,2,3,4\rangle$ may come from the joining of $\langle 1,2,3\rangle$ and $\langle 3,4\rangle$ or of $\langle 1,2,3\rangle$ and $\langle 2,4\rangle$; these are clearly two different subgraphs. The lack of structural information causes RStream to recognize them as the same subgraph instance, leading to incorrect aggregation.

Conversely, missing structural information makes it difficult for RStream to find and merge identical (automorphic) subgraphs that are represented by different tuples. For instance, joining $\langle 1,2,4\rangle$ and $\langle 2,3\rangle$ on the two columns #1 and #0 generates the same subgraph instance as joining $\langle 1,2,3\rangle$ and $\langle 2,4\rangle$ on the columns (#1, #0), although the tuples produced look different ($\langle 1,2,4,3\rangle$ and $\langle 1,2,3,4\rangle$). Failing to identify such duplicates would lead not only to mis-aggregation but also inefficiencies.

To develop applications requiring structural information, a RStream developer can override function `need_structure` to make it return TRUE. This informs RStream to append a piece of information regarding each join to each tuple produced by the join. For example, joining $\langle 1,2\rangle$ with $\langle 2,3\rangle$ on the columns (#1, #0) produces a tuple $\langle 1,2,3,(1)\rangle$, where (1) indicates that this tuple comes from expanding a previous tuple with an edge on its 2nd column.

A further join between $\langle 1,2,3,(1)\rangle$ and $\langle 2,4\rangle$ on the columns (#1, #0) generates tuple $\langle 1,2,3,4,(1,1)\rangle$, which indicates that this tuple comes from first expanding the second column with an edge and then the second column with another edge. This piece of information is added (implicitly) at the end of each tuple, encoding the history of joins, which, in turn, represents the edges that connect the vertices in the tuple.

This structural information is needed in the following two scenarios. First, it is used to encode a subgraph represented by a tuple into a coordination-free *canonical form*, which can be used by the function `is_isomorphic` (defined in `Tuple`) during aggregation to find *isomorphic subgraphs*. Two subgraphs (*i.e.*, tuples) are *isomorphic* iff there exists a one-to-one mapping between their vertices and between their edges, *s.t.* (1) each vertex/edge in one subgraph has one matching vertex/edge in another subgraph, and (2) each matching edge connects matching vertices. Tuples are aggregated at the end based on isomorphism-induced equivalence classes.

Second, the structural information is used to identify tuples representing the same subgraph instance (*i.e.*, by `is_automorphic`). Two subgraphs are *automorphic* iff they contain the same edges and vertices. Tuples that represent the same subgraph instance need to be merged during computation for correctness and performance. The implementation of these functions is discussed in §4.

```
/*Data structures*/
template <class T>
class Tuple {
  int num_elements() {...}
  T element(int i){...}
  virtual bool is_automorphic(Tuple t){...}
  virtual bool is_isomorphic(Tuple t){...}
};
class Edge : public Tuple {...};
class Vertex: public Tuple {...};

class Table {
  int get_key(){...}
  void set_key(int i) {...}
};
class UpdateTable : public Table {...};
class EdgeTable : public Table {...};
class VertexTable : public Table {...};
struct StreamingPartition {
  UpdateTable update_table;
  EdgeTable edge_table;
  VertexTable vertex_table;
  virtual void set_init_value(Vertex v);
};

class Application{
  /* Dataflow graph registered here */
  virtual void run();
  /* Whether we need structural info*/
  virtual bool need_structure() {return FALSE;}
};

/*Phases*/
class Phase {
  virtual bool converged(TerminationLogic l);
};
class Scatter : public Phase {
  virtual Tuple generate_update(Edge e){...};
};
class GatherApply : public Phase {
   virtual void apply_update(Vertex v, Tuple
        update);
};

class RPhase : public Phase{
  /* Functions called from join or select*/
  virtual bool filter(Tuple t1, Tuple t2) {
       return TRUE;}
  virtual int new_key();

 /* Called from the engine*/
  virtual void execute(StreamingPartition p);

 /* == A set of relational functions ==*/
 /* Join ut and et of p and updates ut*/
 void join(StreamingPartition p){...}
 /* Join ut and et of p on all columns of ut
      and updates ut*/
 void join_on_all_columns(StreamingPartition p)
      {...}
 /* Select rows from ut of p and updates ut*/
 void select(StreamingPartition p){...}
 /* Aggregate rows from ut of p*/
 void aggregate(StreamingPartition p, int type)
      {...}
};
```

Figure 3: Major data structures and API functions.

RStream tuples are essentially vertex-based representations of subgraphs. Edges are represented as structural

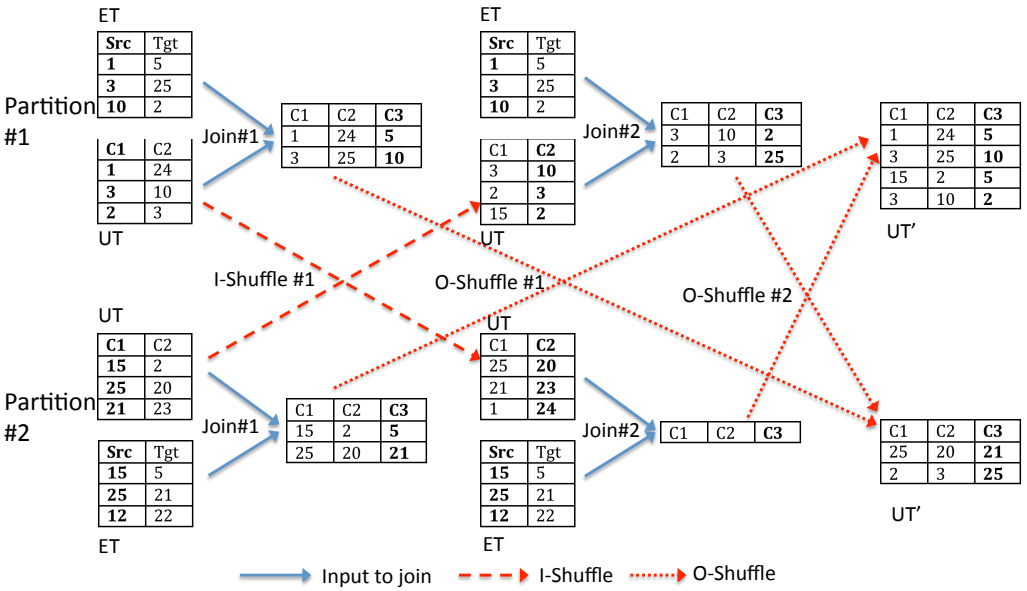


Figure 4: A graphical illustration of `join_on_all_columns`; the streaming partitions #1 and #2 contain vertices [0, 10] and [11, 25], respectively; suppose `new_key` returns 2 (which is column C_3). Structural info is not shown.

information appended at the end of each tuple. Compared to Arabesque where each subgraph (embedding) has an edge-based representation, RStream's representation allows the application to express whether the edge information is needed, providing space efficiency for applications that aim to find statically-known patterns and thus do not need the edge information.

Relational Phases Operations that can be performed in a relational phase include `join`, `select`, `aggregate`, and `join_on_all_columns`. `join` joins the update table with the edge table of each streaming partition on their key columns; `select` selects rows from the update table based on the user-defined filter; and `aggregate` aggregates values from all rows in the update table. The "type" parameter of `aggregate` indicates the type of aggregation such as MAX, MIN, SUM, COUNT, or STRUCTURE_SUM. A special type is STRUCTURE_SUM, which counts the number of subgraphs that belong to the same isomorphism class. If a programmer needs to aggregate over a subset of rows, she can first invoke `select` and then `aggregate`. `join` and `select` change the update table while `aggregate` does not. `join_on_all_columns` will be discussed shortly.

The two callback functions `filter` and `new_key` in class `RPhase` are invoked by `join`, `select`, and `join_on_all_columns` to determine what rows need to be considered and how results should be shuffled, respectively. For either `join` or `select`, changing the key column of the update table (*i.e.*, using `new_key`) will trigger tuple shuffling across streaming partitions.

Note that `RPhase` does not provide a `group-by` function, because group-by can be essentially implemented by a gather-apply phase. During a gather-apply, the vertex table is loaded into memory and tuples from the update table (produced either by a scatter phase or by a relational phase) are streamed in. RStream gathers tuples that have the same key element (*i.e.*, vertex ID) and invokes the user-defined `apply_update` function at Line 41 to compute a new tuple for the vertex. These new tuples are then saved into the vertex table, which is written back to disk at the end of each iteration. In other words, gather-apply produces a new vertex table.

`join_on_all_columns` is the same as `join` except that it joins the update table with the edge table *multiple times*, each time using a different column from the update table as key. The key of the edge table remains unchanged (*i.e.*, source vertex column). The number of joins performed by this function equals the number of columns in the update table. This function is necessary to implement mining algorithms that need to grow a subgraph from all of its vertices, such as Clique or FSM.

Figure 4 illustrates `join_on_all_columns`. Since it changes the key of the update table for each join, RStream shuffles tuples *twice* after a join — the first one, referred to as input shuffle (I-shuffle), shuffles tuples from the update table based on the next key to be used to prepare for the next join; the second one, referred to as output shuffle (O-shuffle), shuffles the result tuples based on the new key defined by `new_key` to prepare for the final output, which will eventually become the new update table (UT′).

Termination Class `Phase` contains an abstract function `converged` that needs to be implemented in user-defined phases. This function defines termination logic for iterative computation algorithms (with back edges on the dataflow graph). Note that RStream invokes this function

```
1  class FSMProgram : public Application {
2    /*FSM needs structural info*/
3    bool need_structure() { return TRUE; }
4
5    void run(Engine e){
6      Scatter cs;
7      e.set_start(cs);
8      FSMPhase fsm;
9      e.insert_phase(fsm, cs);
10     /* This forms a cycle */
11     e.insert_phase(fsm, fsm);
12     e.run();
13   }
14 };
15
16 class AggregateFilter : public RowFilter{
17   AggregationStream aggStream;
18   int threshold;
19
20   bool filter_out_row(Tuple t){
21     int support = get_support(aggStream, t);
22     if(support >= threshold) return FALSE;
23     /*It couldn't be a frequent subgraph.*/
24     return TRUE;
25   }
26 };
27
28 class FSMPhase : public RPhase{
29   static int MAX_ITE = MAX_FSM_SIZE * (
         MAX_FSM_SIZE - 1)/2;
30
31   bool converged(TerminationLogic l) {
32     if(l.get_ite_id() == MAX_ITE) return TRUE;
33     return FALSE;
34   }
35
36   int new_key(){ return LAST_COLUMN;}
37
38   void execute(StreamingPartition sp){
39     UpdateTable ut = sp.update_table;
40     ut.set_key(0);
41     EdgeTable et = sp.edge_table;
42     et.set_key(0);
43     super::join_on_all_columns(sp);
44     super::aggregate(sp, STRUCTURE_SUM);
45     AggregateFilter af;
46     super::select(sp, af);
47   }
48 };
```

Figure 5: An FSM program; structural info is needed.

only for the phases that are sources of dataflow back edges to determine whether further iterations are needed.

Example: FSM on RStream We use one more example — frequent subgraph mining — to demonstrate the power of RStream's programming model, and in particular, the usage of dataflow cycles and the function `join_on_all_columns`. Figure 5 shows the computation logic. It consists of two phases: a (standard) scatter phase and an iterative relational phase `FSMPhase`. The basic idea is that each execution of `FSMPhase` performs `join_on_all_columns` between the update and edge table. Each tuple in the update table represents a new subgraph we have found. This special join attempts to "grow" each subgraph with one edge on each vertex in the subgraph. For example, for a tuple (a,b,c,d), this join will join it with the edge table *four times*, each on a different column. Each join generates five-tuples of the form (a,b,c,d,e), which is keyed at e (*i.e.*, LAST_COLUMN specified in Line 36). Such tuples are shuffled into the partitions to which e belongs.

Given the max size of subgraphs to be considered (*e.g.*, *MAX_FSM_SIZE* = 4), all we need is to execute `FSMPhase` for a fixed number of times; this number equals the maximum number of edges that can be involved in the largest FSM: $MAX\_FSM\_SIZE \times (MAX\_FSM\_SIZE - 1)/2$, as shown in Line 29.

At the end of each `FSMPhase`, we aggregate all tuples in the update table (Line 44) to count the number of each distinct structural pattern. After the aggregation, a `select` is performed to filter out tuples corresponding to infrequent subgraphs (Line 46). This function takes as input a variable of class `AggregateFilter`, which contains a function `filter_out_row` that will be applied to each tuple. This function eliminates tuples that represent structural patterns whose supports are not high enough (Lines 20-25). The intuition here is that if a subgraph is infrequent, then any supergraphs generated based on it must be infrequent — referred to as the Downward Closure Property [7]. These infrequent tuples can be safely ignored in the next iteration. Similarly to Arabesque [66], we use the *minimum image-based support metric* [22] as it can be efficiently computed. This metric defines the frequency of a structural pattern as the *minimum* number of distinct mappings for any vertex in the pattern over all instances of the pattern.

4 RStream Implementation

RStream's implementation has an approximate of 7K lines of C++ code and is available on Github.

4.1 Preprocessing

For graphs that cannot fit into memory, they are first partitioned by a *preprocessing* step. The graph is in the edge-list or adjacency-list format on disk. RStream divides vertices into logical intervals. One interval in RStream defines a partition that contains edges whose *source vertices* fall into the interval. Edges that belong to the same partition do not need to be further sorted. To achieve work balance, we ensure that partitions have similar sizes. Since our join implementation (discussed shortly) needs to load each edge table entirely into memory, the number of streaming partitions is determined automatically to guarantee that the edge table for each streaming partition does not exceed the memory capacity while memory can still be fully utilized.

For graphs that can be fully loaded, RStream generates one single partition and no tuple shuffling will be incurred for joins. However, unlike share-memory graph computation systems that can hold all computations in memory, mining algorithms in RStream can cause update tables to keep increasing — even for very small graphs, their update tables can grow to be several orders of magnitude

larger than the size of the original graph. Hence, RStream requires disk support regardless of the initial graph size.

4.2 Join Implementation

As the update table grows quickly, to implement join, we load the edge table into memory and *stream in* tuples from the update table for each streaming partition. RStream performs *sequential disk accesses* to both the update table and the edge table, and *random memory accesses* to the loaded edge data.

Note that the edge table represents the original graph while the update table contains intermediate data generated during computation. Since the edge table never changes, the amount of memory required by RStream is bounded by the maximum size of a partition in the original graph, *not* the intermediate computation data, which can be much larger than the graph size.

Scatter and gather-apply are implemented in the same way as in X-Stream — for scatter, the vertex table is loaded while edges are streamed in; for gather-apply, the vertex table is loaded while updates are streamed in.

Filtering is performed by invoking the user-defined `filter` function upon the generation of a new tuple. When *join_on_all_columns* is used, different tuples generated may represent identical (automorphic) structures. Similarly to Arabesque, we define *tuple canonicality* by selecting a unique (canonical) tuple from its automorphic set as a representative and remove all other tuples. Details of this step are discussed shortly in §4.3.

Multi-threading RStream uses a producer-consumer paradigm for implementing join. The main thread pushes the IDs of the streaming partitions to be processed into a worklist as tasks, and starts multiple producer and consumer threads. Each producer thread pops a task off the list, loads its edge table, and streams in its update table into the producer's thread-local buffer. The producer thread joins each "old" update tuple with the edge table and produces a "new" update tuple.

We allocate a *reshuffling buffer*, for each streaming partition, to store new update tuples entering this partition. Producers and consumers synchronize using locks to ensure concurrent accesses to reshuffling buffers. Each producer sends each generated tuple to its corresponding reshuffling buffer when the buffer has room, while each consumer flushes a buffer into its corresponding "new" update table on disk when the buffer is full.

Figure 6 illustrates multiple producers and consumers. There are four producer threads and two consumer threads. Eight tasks are pushed onto the task worklist. Each producer takes one task from the list, loads its edge partition, and streams in its update partition. Each producer conducts the computation and generates output updates locally. Reshuffling is synchronized using std::mutex.

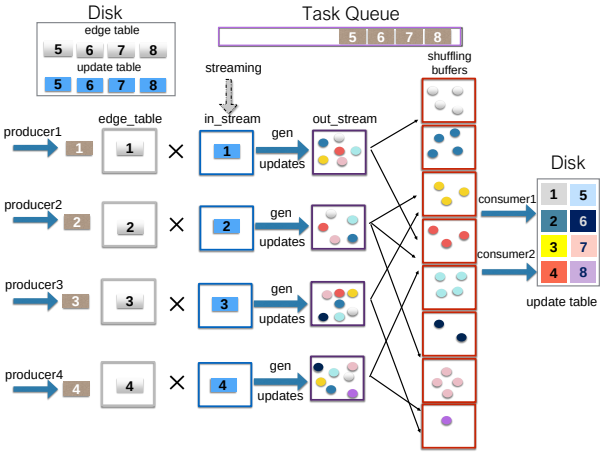


Figure 6: A graphical illustration of multiple producers, multiple consumers and reshuffling buffers.

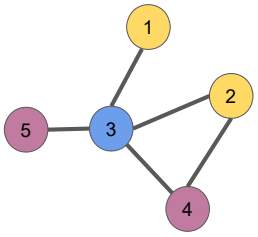
Load (Re)balancing Unlike X-Stream where the size of each streaming partition stays unchanged, in RStream, the size of each partition can grow significantly for two reasons. First, mining algorithms keep looking for graph patterns of increasing sizes, leading to the ever-growing update table. Second, tuple reshuffling at the end of each join can result in unbalanced partitions. These unbalanced partitions, if handled inappropriately, can result in significant inefficiencies (*e.g.*, underutilized CPU).

One possible solution would be to repartition the streaming partitions at the end of each relational phase for load rebalancing. However, repartitioning can incur significant disk I/O, slowing down the computation. Rather than repartition the graph, we use fine-grained tasks by dividing each update table into multiple smaller update chunks. Instead of pushing an entire update partition into the list, we push one chunk at a time. For work balancing, we also order these tasks based on their sizes so that "larger" tasks have a higher priority to be processed.

Enumeration Note that, by joining the update table with the edge table, RStream performs *breadth-first* enumeration of subgraphs. While this approach requires more storage to materialize tuples compared to a depth-first approach, it enables easier parallelization as all tuples of a given size are materialized and available for processing. Further, as a disk-based approach, RStream's breadth-first enumeration increases disk usage rather than memory usage — As shown in Figure 6, the enumeration delivers each newly generated tuple to a shuffling buffer and once the buffer is full, RStream flushes the buffer to disk.

4.3 Redundancy Removal via Automorphism Checks

Since different workers can reach identical (automorphic) tuples during processing, we need to identify and filter out such tuples. RStream adopts the idea of *embedding*



Canonical tuples of size 3		
1	3	2
1	3	4
1	3	5
2	3	4
2	3	5
3	4	5

Figure 7: A graph and its canonical tuples of size 3.

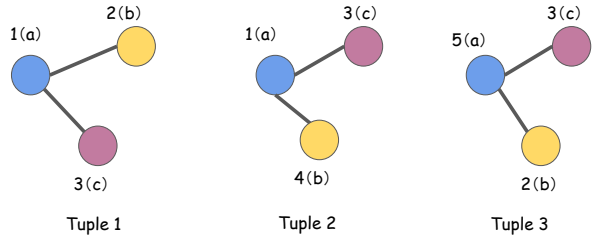


Figure 8: Aggregation example of three isomorphic tuples.

canonicality used in Arabesque [66]. We select exactly one of the automorphic tuples and elect it as "canonical". RStream runs a tuple canonicality check to verify whether a tuple t can be pruned. This algorithm runs on a single tuple without coordination. It starts with an existing canonical tuple t and checks, when t is grown with a new vertex v into a new tuple t', whether t' is also canonical. The basic idea is based on a notion of *uniqueness*: given the set S_m of all tuples automorphic to a tuple m, there exists exactly one canonical tuple t_c in S_m. The goal of this algorithm is, thus, to check whether the newly generated tuple t' is this t_c.

The tuple t' is canonical if and only if its vertices are visited in an order that is consistent with their IDs: a vertex with a smaller ID is visited earlier than one with a larger ID. In other words, RStream characterizes a tuple as the list of its vertices sorted by the order in which they are visited. When we check the canonicality of tuple t' that comes from growing an existing canonical tuple t with a vertex v, we first find the first neighbor v' of v, and then verify that there is no vertex $\in t$ after v' with a larger ID than v. Figure 7 shows a simple graph and its canonical tuples of size 3. Because RStream only processes canonical tuples, *uniqueness* is maintained in our tuple encoding (with structural information). A more detailed description can be found in [67].

4.4 Pattern Aggregation via Isomorphism Checks

For mining algorithms, aggregation needs to be done on tuples to count the number of each distinct shape (*i.e.*, structural pattern) at the end of the computation. Aggregation boils down to isomorphism checks — among all non-automorphic tuples, we count the number of those that belong to each isomorphism class. A challenge here is that isomorphism checks are expensive to compute — it is known to be isomorphism (GI)-complete and the bliss library [3] we use employs an exponential time algorithm.

RStream adopts the aggregation idea from Arabesque by turning each tuple into a *quick pattern* and then into a *canonical pattern* [16, 66]. The canonical pattern of a subgraph, which is different than the canonical tuple described earlier for automorphism checks, encodes the shape of the subgraph with all vertex information removed. Two tuples are isomorphic iff they have the same canonical patterns. The quick pattern of a subgraph is simply a total order of edges in the subgraph with vertex information removed. Two tuples may have different quick patterns even if they are isomorphic.

Given that canonical checks are expensive, we use the same two-step aggregation as in Arabesque — the first step uses quick patterns that can be efficiently computed to perform *coarse-grained* pattern aggregation, while the second step takes as input results from the first step, converts them into canonical patterns, based on which *fine-grained* aggregation is done. The aggregation conducts a two-stage MapReduce computation — the first on quick patterns and the second on canonical forms — across *all* streaming partitions. Although the aggregation idea originates from Arabesque [66], we provide a detailed example in the rest of this section to make this paper more self-contained.

Example The map phase takes quick patterns and canonical forms as input, performs local aggregation, and shuffles them into hash buckets defined by the hash value of these patterns. The reduce phase aggregates key/value pairs in the same bucket. Figure 8 depicts an example with three tuples: $tuple_1 : \langle 1(a), 2(b), 3(c), (0)\rangle$, $tuple_2 : \langle 1(a), 3(c), 4(b), (0)\rangle$, and $tuple_3 : \langle 5(a), 3(c), 2(b), (0)\rangle$. Here numbers represent vertex IDs and characters represent labels for each vertex. Note that mining algorithms often require graphs to have vertices and edges explicitly labeled. These labels represent vertex/edge properties that never change during the computation and they are needed for isomorphism checks. (0) represents the structural information obtained from the past joins.

RStream first turns each tuple into a quick pattern to reduce the number of distinct tuples. A quick pattern is obtained by simply extracting the label information and renaming vertex IDs in a given tuple, with vertex ID always starting at 1 and increasing consecutively. In the previous example, the quick patterns for the three tuples are $qp_1 : \langle 1(a), 2(b), 3(c), (0,0)\rangle$, $qp_2 : \langle 1(a), 2(c), 3(b), (0,0)\rangle$, $qp_3 : \langle 1(a), 2(c), 3(b), (0,0)\rangle$, respectively. In the map phase, RStream emits three quick pattern pairs: (qp_1, 1), (qp_2, 1), (qp_3, 1); the reduce phase further aggregates them into (qp_1, 1), (qp_2, 2) as qp_2 and qp_3 are identical.

Graphs	#Edges	#Vertices	Description
CiteSeer [29]	4,732	3,312	CS pub graph
MiCo [29]	1.1M	100K	Co-authorship graph
Patents [32]	14.0M	2.7M	US Patents graph
LiveJournal [17]	69M	4.8M	Social network
Orkut [1]	117M	3M	Social network
UK-2005 [20]	936M	39.5M	Web graph

Table 1: Real world graphs.

Program	LoC	Description
Triangle Counting (TC)	75	Counting # triangles
Closure	68	Computing transitive closure
N-Clique	36	Identify cliques of size N
N-Motif	36	Counting motifs of size N
Frequent Subgraph Mining (FSM)	40	Identify FSM of size N
Connected Components (CC)	40	Identify connected components

Table 2: Algorithms experimented.

Due to the coarse-grained modeling of quick patterns, tuples that are actually isomorphic may correspond to different quick patterns. As a next step, quick patterns are turned into canonical forms (by bliss) to perform fine-grained aggregation. A canonical form uniquely identifies a class of isomorphic subgraphs. In the example, the two quick patterns correspond to the same canonical form cf_1 : $\langle 1(a),2(b),3(c),(0,0)\rangle$. RStream eventually reports (cf_1, 3) as the final result. Since the number of quick patterns is much smaller than the number of distinct tuples, the cost of isomorphic checks can be significantly reduced.

One possible optimization is to perform *eager aggregation* — tuples are aggregated as they are being streamed into their respective partitions. We have implemented this optimization, but our experimental results showed only a minor improvement (5% in the aggregation phase and less than 2% for the overall execution).

5 Evaluation

Our evaluation focuses on three research questions:

- Q1: How does RStream compare to state-of-the-art graph mining systems? (§5.1)
- Q2: How does RStream compare to state-of-the-art Datalog engines? (§5.2)
- Q3: What is RStream's overall and I/O throughput and how quickly does data grow for mining algorithms? (§5.3)

Experimental Setup We ran our experiments using six algorithms (Table 2) over six real-world graphs (Table 1). CiteSeer, MiCo, and Patents are the graphs that were used by Arabesque and DistGraph in their evaluations. We used them primarily for comparisons with the mining systems. Similarly, Orkut and LiveJournal were used by BigDatalog [61] and we used them to compare RStream with BigDatalog. UK-2005 has almost a billion edges and is much larger than all the graphs used by Arabesque [66].

For mining algorithms, we developed Triangle Counting (TC), Clique, Motif Counting (MC), Transitive Closure Computation (Closure), and Frequent Subgraph Mining (FSM). Closure is a typical Datalog workload, and hence, we used it specifically to compare RStream with Datalog. Connected Components (CC) is a graph computation algorithm. Since RStream can also support computation (with just GAS and no relational phases), we added CC into our algorithm set to help us develop a deep understanding of the behavioral differences between graph computation and graph mining (§5.3).

		CS	MC	PA
TC	**RS**	**0.04**	**15.8**	**6.7**
	AR-10	38.1	43.1	114.9
	AR-5	39.8	44.9	116.4
	AR-1	34.2	40.7	131.5
5-C	**RS**	**0.01**	**115.1**	**35.3**
	AR-10	42.8	132.0	174.5
	AR-5	39.3	171.7	183.0
	AR-1	34.9	404.3	227.9
3-M	**RS**	**0.02**	**43.0**	**89.1**
	AR-10	40.6	51.7	116.0
	AR-5	39.7	52.8	110.5
	AR-1	32.7	47.0	132.9
4-M	**RS**	**1.41**	**93417**	**8849**
	AR-10	41.7	-	-
	AR-5	40.4	-	-
	AR-1	34.2	-	-
3-F 300	**RS**	**0.89**	**402.1**	**517.4**
	AR-10	35.9	-	-
	AR-5	39.3	-	-
	AR-1	33.7	-	-
	SM-10	2.1	69431.7	-
	SM-5	2.6	66604.3	-
	SM-1	3.5	77332.7	-
	DG-10	12.3	-	-
	DG-5	4.1	-	-
	DG-1	5.2	-	-
3-F 500	**RS**	**0.10**	**384.3**	**502.1**
	AR-10	35.7	-	-
	AR-5	39.3	-	-
	AR-1	34.4	-	-
	SM-10	2.0	15867.5	-
	SM-5	2.3	15209.4	-
	SM-1	3.2	21043.3	-
	DG-10	0.4	-	-
	DG-5	0.12	-	-
	DG-1	0.11	-	-
3-F 1K	**RS**	**0.06**	**351.7**	**383.7**
	AR-10	35.6	5790.1	-
	AR-5	39.9	5397.9	-
	AR-1	33.9	5848.2	-
	SM-10	1.2	802.6	-
	SM-5	1.1	790.8	-
	SM-1	1.1	1175.1	-
	DG-10	0.4	-	-
	DG-5	0.12	-	-
	DG-1	0.10	-	-
3-F 5K	**RS**	**0.02**	51.0	**376.4**
	AR-10	41.6	120.8	-
	AR-5	37.7	192.7	-
	AR-1	31.8	610.3	-
	SM-10	1.0	12.1	-
	SM-5	1.1	**11.6**	-
	SM-1	1.3	14.5	-
	DG-10	0.3	-	-
	DG-5	0.05	-	-
	DG-1	0.08	-	-

Table 3: Comparisons between RStream (RS), Arabesque (AR-n), ScaleMine (SM-n), and DistGraph(DG-n) on four mining algorithms — triangle counting (TC), Clique (k-C), Motif Counting (k-M), and FSM (k-F) — over three graphs CiteSeer (CS), MiCo (MC), and Patents (PA); n represents the number of nodes the distributed systems use; k is the size of the structure to be mined; '-' indicates execution failures. For FSM, four different support parameters (300, 500, 1K, and 5K) are used and explicitly shown in each 3-F row. Highlighted rows are the shortest times (in seconds).

Our experiments were conducted on a 10-node cluster, each with 2 Xeon(R) CPU E5-2640 v3 processors, 32GB memory, and 3 SSDs with a total of 5.2TB disk space, running CentOS 6.8. Data was split evenly on the three disks. RStream ran on one single node with 32 threads to fully utilize CPU resources and disk bandwidth, while distributed systems used all the nodes.

5.1 Comparisons with Mining Systems

Systems and Algorithms We compared RStream with three state-of-the-art distributed mining systems: Arabesque [66], ScaleMine [4], and DistGraph [65].

Other distributed mining systems such as G-thinker [77] are not publicly available and hence not considered in our experiments. We ran these three systems with 10 nodes, 5 nodes, and 1 node to have a precise understanding of where RStream stands. In this first set of experiments, all Motif executions were run with a maximum size of 4; Clique was run with a maximum size of 5; and FSM was run with size of 3.

As discussed earlier, to run FSM we used the minimum image-based support metric [22], which defines the frequency of a pattern as the minimum number of distinct mappings for any vertex in the pattern, over all instances of the pattern. We explicitly state the support, denoted S, used in each experiment since this parameter is sensitive to the input graph. Clearly, the smaller S is, the more computation is needed.

In this experiment, we used CiteSeer, MiCo, and Patent as our input graphs. These three graphs came with labels[2] and were also used to evaluate Arabesque, ScaleMine, and DistGraph. Our initial goal was to evaluate RStream with all graphs used in prior works, but other graphs were either unavailable or do not have labels. Although these are relatively small graphs from the perspective of graph computation, running mining algorithms on them can generate orders-of-magnitude more data (see Table 5).

Table 3 reports the running times of the four systems. Note that ScaleMine and DistGraph were designed specifically to mine frequent subgraphs, and hence we could obtain only FSM's performance for these two systems. It is clear that RStream **outperforms all three systems in all cases but 3-FSM with support = 5000**. Arabesque, ScaleMine, and DistGraph failed when the size of a pattern increases. These failures were primarily due to their high memory requirement (for storing intermediate data) that could not be fulfilled by our cluster.

For FSM, on small graphs such as CiteSeer, DistGraph appears to be more efficient than the other two systems. However, DistGraph could not scale to the MiCo graph on our 10-node cluster. ScaleMine successfully processed MiCo, but took a long time, because ScaleMine trades off computation for memory; instead of caching intermediate results in memory, it always re-computes from scratch, which explains why it has better scalability but lower efficiency. None of these three systems could process FSM over the Patents graph even when support = 5000. By contrast, RStream successfully executed FSM over all the graphs under all the configurations.

RStream underperforms ScaleMine in only one case: 3-FSM (S=5000) over MiCo. RStream outperforms Arabesque (on 10 nodes) by an overall (GeoMean) of **60.9**×, ScaleMine by an overall of **12.1**×, and DistGraph by an overall of **7.2**×. As Arabesque was developed in Java, the 60.9× speedup may be partly due to RStream's use of an efficient language (C++). ScaleMine and DistGraph were both C++ applications and, hence, the wins over them provide a closer approximation of the benefit a disk-based system could offer.

[2]Mining algorithms require *labeled graphs* (*i.e.*, vertices and edges have semantic labels).

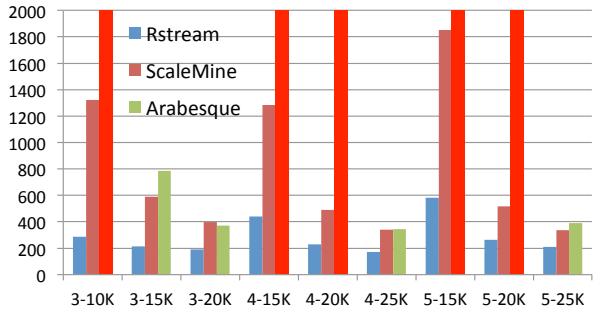


Figure 9: FSM performance comparisons with different pattern sizes and supports over the Patents graph. Tall red bars on the right of each group represent Arabesque failures.

UK Graph To understand RStream's performance on larger graphs, we ran 3-FSM on RStream to process the UK-2005 graph that has almost a billion edge. Note that none of the three distributed systems could process the graph when running 3-FSM with even a 5K support on our 10-node cluster. In all prior works, the only evidence of a mining system successfully processing a billion-edge graph was reported in [65] where DistGraph, using 512–2048 IBM BlueGene/Q machines each with 16 cores and 256GB memory, processed several synthetic graphs with 1B–4B edges in 2000 – 7000 seconds (with varying supports). Here we experimented RStream with four support parameters – 2K, 3K, 4K, and 5K – on one single machine with only 32GB memory. RStream successfully processed all of them, *e.g.*, in 4080.9, 3016.3, 2228.9, and 2146.2 seconds, respectively.

RStream ran out of memory when a relatively small support was used (*i.e.*, $\leq$1000) to compute frequent subgraphs over UK. After spending a great amount of time investigating the problem, we found that the large memory consumption was potentially due to memory leaks in the bliss library rather than RStream, which guarantees that the amount of data to be loaded from each streaming partition never exceeds the memory capacity.

Larger FSMs To evaluate how RStream performs on k-FSMs with larger k, we conducted a set of experiments over the Patents graph with various k and supports. Since DistGraph failed in most cases when we increased k, this set of experiments focused on RStream, ScaleMine, and Arabseque, and the results of the comparisons are reported in Figure 9. Both Arabesque and ScaleMine were executed with 10 nodes. Overall, RStream is **2.46**× and **2.28**× faster than ScaleMine and Arabesque.

Support	Patents		Mico	
	RStream	GraMi	RStream	GraMi
5K	**504.6**	-	**51.0**	-
10K	**286.7**	-	**23.2**	36.5
15K	**213.3**	-	**14.3**	18.7
20K	**190.8**	-	**8.6**	9.2

Table 4: FSM performance comparisons between RStream and GraMi over Patents and Mico; time is measured in seconds.

We have also compared RStream with GraMi [29], which is a specialized graph mining library designed to perform single-machine shared-memory FSM computation, over the Patents and Mico graphs. Table 4 reports the results. Note that, for each support, GraMi reports patterns of all sizes with respect to the support. RStream was executed in a similar way to provide a fair comparison. GraMi ran out of memory for all cases over the Patents graph. On the Mico graph, RStream outperforms GraMi even for large (*e.g.*, 20K) supports.

There are two reasons that could explain RStream's superior efficiency. First, joins performed by RStream grow subgraphs *in batch* while the other systems enumerate and grow embeddings individually. Second, the three systems RStream was compared against are all distributed systems that have a large startup and communication overhead. While the data size quickly grows to be larger than the memory capacity of a single machine, this size is often small in an early stage of the execution. Distributed systems suffer from communication overhead throughout the execution, while RStream does not have heavy I/O in this early stage.

The fact that the three distributed systems failed in many cases does not necessarily indicate that RStream can scale to larger graphs than them. We believe that these systems, if given enough memory, should have performed better than what is reported in Table 3. However, their exceedingly high memory requirement is very difficult to satisfy — the 10-node cluster we used is the only cluster to which we have exclusive access. According to [66], running 4-motif on a 200M-edge graph took Arabesque 6 hours consuming 20 $\times$ 110GB = 2200GB memory. As a reference point, the most memory-optimized cluster (x1.32xlarge) Amazon EC2 offers has only 1952GB memory, which is still not enough to run the algorithm.

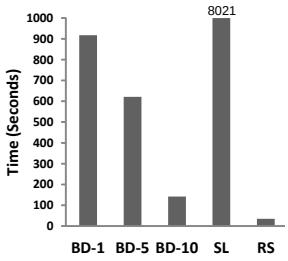
These results do suggest, though, that if a user has only a limited amount of computing resources, RStream should be a better choice than these other systems because RStream's disk requirement is much easier to fulfill and yet it can scale to large enough real-world graphs.

5.2 Comparisons with Datalog Engines

Since our GRAS model is inspired partly by the way Datalog enables easy programming of mining algorithms, we have also compared RStream with the state-of-the-art

		LiveJournal	Orkut
TC	**RS**	**87.0**	**827.4**
	BD-10	94.8	1205.3
	BD-5	109.6	1850.3
	BD-1	567.3	-
	SL	896.1	-
CC	**RS**	**101.1**	**352.0**
	BD-10	179.6	441.2
	BD-5	251.3	420.3
	BD-1	187.8	445.2
	SL	379.8	-

(a)



(b)

Figure 10: (a) Comparisons between RStream (RS), BigDatalog (BD-*n*), and SociaLite (SL) on TC and CC; (b) Closure comparison over CiteSeer.

Datalog engines. We use BigDatalog [61] with Spark joins and SociaLite [40], a shared memory Datalog engine. We used the LiveJournal and Orkut graphs, which were initially used to evaluate BigDatalog [61] to evaluate BigDatalog. We used three algorithms: Triangle Counting (TC), Connected Components (CC), and Closure Computation (Closure). Although CC and Closure are not typical mining algorithms, they are Datalog programs regularly used to evaluate the performance of a Datalog engine. Hence, we included them in this experiment. Note that BigDatalog has been shown to outperform vanilla Spark over these workloads due to several optimizations implemented over Spark joins [61].

Figure 10(a) compares the performance of RStream with that of BigDatalog and SociaLite. For TC and CC, RStream outperforms BigDatalog (with 10 nodes) by a GeoMean of **1.37**$\times$, while SociaLite failed in most cases. For transitive closure, CiteSeer was the only graph that RStream, BigDatalog, and SociaLite could all successfully process. Their performance comparison is shown in Figure 10(b): RStream is **4**$\times$ faster than BigDatalog running on 10 nodes, while it took SociaLite a large amount of time (8021 seconds) to finish closure computation.

These results appear to be different from what was reported in the prior works [61] and [40]. We found that the difference was primarily due to the input graphs — both the works [61] and [40] used synthetic acyclic graphs for transitive closure, while real graphs have both cycles and very high density that synthetic graphs do not have. Neither BigDatalog nor SociaLite could finish closure computation for any graph other than CiteSeer, while RStream successfully computed closure for LiveJournal in 4578 seconds.

5.3 RStream Performance Breakdown

To fully understand RStream's performance, throughput, I/O efficiency, and disk usage, we have conducted a set of experiments using various graphs and algorithms.

Intermediate Data Generation Table 5 reports, for 4-Motif (over the Patents graph) and 4-FSM (over the Patents graph), the number of tuples generated at the end

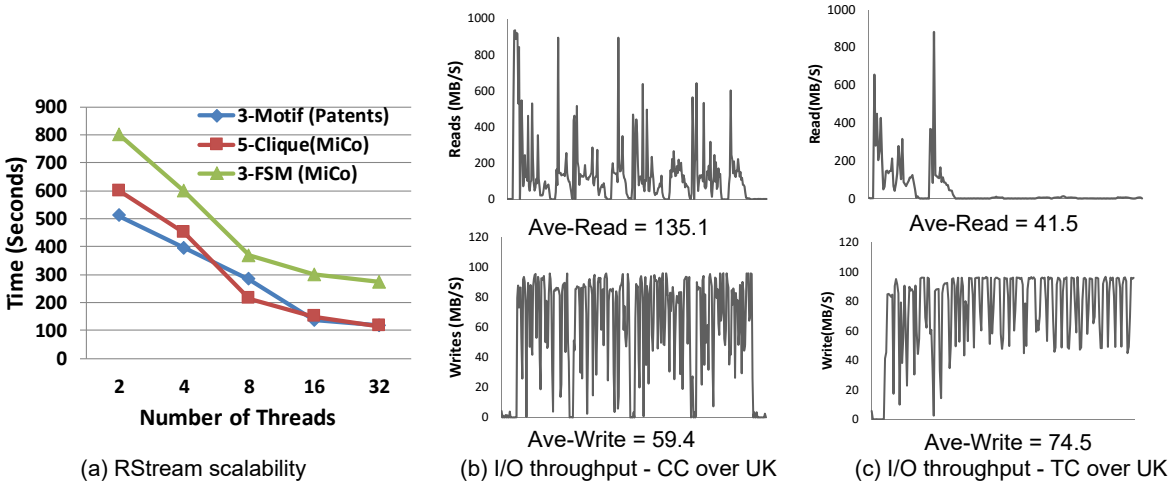


Figure 11: RStream's scalability (a), I/O throughput when running CC over UK (b), and I/O throughput when running TC over UK (c). I/O was measured with `iostat`.

	Phase	#Tuples	TS	#MB
4-Motif MiCo	0	1,080,156	16	16.5
	1	91,151,339	24	2,086.3
	2	29,044,509,725	32	886,378.1
	3	10,016,299,628	40	382,091.5
	Total	3.9×10^{10}	-	1,270,572.4 (1.21TB)
4-FSM, S=10K Patents	0	13,965,409	16	213.1
	1	625	28	0.02
	2	5,861,830	16	89.4
	3	93,313,116	24	2,135.8
	4	13,764	36	0.5
	5	29,462,761	24	674.3
	6	816,909,842	32	24,930.1
	7	101,254	44	4.2
	8	633,673,981	32	19,338.2
	9	57,361,813	40	2,188.2
	10	30,283	52	1.5
	11	509,304	40	19.4
	Total	1.65×10^{9}	-	49,594.72 (48.4GB)

Table 5: The number of tuples (**Tuples**) generated for each phase execution, the size of each tuple (**TS**), and the number of bytes (**#MB**) shuffled for 4-Motif over the Patents graph and 4-FSM, S=10K over the Mico graph.

of each phase, the size of each tuple, as well as the storage consumption of these tuples. The amount of data generated during the execution can easily exceed the memory capacity. For 4-Motif, the total amount of intermediate data generated requires 1.21TB of disk space. This motivates our out-of-core design that leverages large SSDs to store these intermediate subgraphs.

	FSM(300)	FSM(500)	FSM(1000)	3-Motif	4-Motif	5-Clique
CiteSeer	129	110	76	83	1914	26
MiCo	2388	2366	2285	1206	12408	6968
Patents	1234	1151	936	110	2791	275
UK	1367	2379	1461	1001	8914	7231

Table 6: Ratios between the final disk usage and original graph size (in the binary format).

To understand how large the total amount of data generated is, Table 6 further reports, for each graph, the ratio between the amount of storage needed at the end of each execution and the original size of the graph. This growth can be as large as 5 orders of magnitude (4-Motif over the MiCo graph). These ratios also reflect (1) the density of each graph (regardless of the size of the graph), which determines how difficult the graph is to process; and (2) the computation complexity of each algorithm, which determines how difficult the algorithm is to run. The MiCo graph is the one with the highest density, although it is relatively small in size. 4-Motif is the algorithm that needs the most computations as it generates the most intermediate data compared to other algorithms.

Scalability and I/O Figure 11(a) shows RStream's running time for varying numbers of threads. In general, RStream scales with the number of threads. However, RStream's scalability decreases when the number of threads exceeds 8 because the disk bandwidth was almost saturated when 8 threads were used.

To understand how RStream performs for mining and computation algorithms, Figure 11(b) and (c) depict RStream's I/O throughput for a computation program (CC) and a mining program (TC), respectively. For CC, we monitored I/O in a full scatter-gather-apply iteration, while for TC, our measurement covered the full cycle of a join – loading the edge table, streaming in update tuples, performing joining, and writing back to the update table. The file system cache was flushed during monitoring. Note that the high read throughput (*e.g.*, 800+MB/s)

achieved by RStream was primarily due to data stripped across the SSDs.

These two plots reveal the differences of these two types of algorithms: computation algorithms such as CC are dominated by I/O — *e.g.*, disk reads/writes occur throughout the iteration. By contrast, relational joins in the mining algorithms such as TC are more compute-intensive, as most of the reads occur in an early stage of the join and the rest of the time is all spent on the in-memory computation (of joining and aggregation). For TC, writes still scatter all over the window due to the producer-consumer model used in RStream— the number of consumer threads is often small and hence many of the disk writes overlap with the computation.

6 Related Work

RStream is the first single-machine, out-of-core graph mining system. Since graph processing is an extensively studied topic, we focus on work that is closely related.

Distributed Mining Systems Arabesque [66] is a distributed system designed to support mining algorithms. Arabesque presents to the developer an "embedding" view. Arabesque enumerates all possible embeddings with increasing sizes and the developer processes each embedding with a filter-process programming model. RStream is more efficient than Arabesque because we join tuples *in batch* rather than enumerating them individually. ScaleMine [4] is a parallel frequent subgraph mining system that contains two phases. The first phase computes an approximate solution by quickly identifying subgraphs that are frequent with high probability and collecting various statistics. The second phase computes the exact solution by using the results of the approximation to prune the search space and achieve load balancing. DistGraph [65] is an MPI-based distributed mining system that uses a set of optimizations and efficient operations to minimize communication costs. With these optimizations, DistGraph scales to billion-edge graphs with 2048 IBM BlueGene/Q nodes. G-thinker [77] is another distributed mining system that provides an intuitive graph-exploration API and a runtime engine. However, G-thinker does not support FSM and Motif-counting, which are arguably the most important mining algorithms. In addition, G-thinker's implementation is not publicly available.

Specialized Graph Mining Algorithms gSpan [78] is an efficient frequent subgraph mining algorithm designed for mining multiple input graphs. Michihiro *et al.* [38] uses an anti-monotonic definition of support based on the maximal independent set to find edge-disjoint embeddings. GraMi [29] is an effective method for mining large input graph. Ribeiro *et al.* [55] proposes an approach for counting frequencies of motifs [54]. Maximal clique is a well-studied problem. There exist a lot of approaches to this problem, among which work from Bron-Kerbosch [23] has the highest efficiency. Recently, a body of algorithms have been developed to leverage parallel [28, 12, 59, 64], distributed systems (such as Map/Reduce) [35, 19, 41, 44, 71, 6, 36, 82, 18], or GPUs [37].

Single-Machine Graph Computation Systems Single-machine graph computation systems [39, 58, 85, 75, 42, 83, 74, 34, 70, 45, 8] have become popular as they do not need expensive computing resources and free developers from managing clusters and developing/maintaining distributed programs. State-of-the-art single-machine systems include Ligra [63], Galois [51], GraphChi [39], X-Stream [58], GridGraph [85], raphQ [75], MMap [42], FlashGraph [83], TurboGraph [34], Mosaic [45], and Graspan [74].

Ligra [63] provides a shared memory abstraction for vertex algorithms. The abstraction is suitable for graph traversal. Galois [51] is a shared-memory implementation of graph DSLs on a generalized Galois system, which has been shown to outperform their original implementations. GRACE [72], a shared-memory system, processes graphs based on message passing and supports asynchronous execution by using stale messages.

Efforts have been made to improve scalability for systems over semi-external memory and SSDs. GraphChi [39] uses shards and a parallel sliding algorithm to reduce disk I/O for out-of-core graph processing. Bishard Parallel Processor [49] reduces non-sequential I/O by using separate shards to contain incoming and outgoing edges. X-Stream [58] uses an edge-centric approach in order to minimize random disk accesses. GridGraph [85] uses partitioned vertex chunks and edge blocks as well as a dual sliding window algorithm to process graphs residing on disks. Vora *et al.* from [70] reduces disk I/O using dynamic shards.

FlashGraph [83] is a semi-external memory graph engine that stores vertex states in memory and edge-lists on SSDs. It is built on the assumption that all vertices can be held in memory and a high-speed user-space file system for SSD arrays is available to merge I/O requests to page requests. TurboGraph [34] is an out-of-core engine for graph database to process graphs using SSDs. Pearce *et al.* [53] uses an asynchronous approach to execute graph traversal algorithms with semi-external memory. It utilizes in-memory prioritized visitor queues to execute algorithms like breadth-first search and shortest paths.

Distributed Graph Computation Systems Google's Pregel [46] provides a synchronous vertex centric framework for large scale graph processing. Many other distributed systems [46, 43, 30, 26, 57, 27, 84, 80, 60, 69, 48, 76, 24, 68] have been developed on top of the same graph-parallel abstraction.

GraphLab [43] is a framework for distributed asynchronous execution of machine learning and data mining algorithms on graphs. PowerGraph [30] provides efficient distributed graph placement and computation by exploiting the structure of power-law graphs. Cyclops [26] provides a distributed immutable view, granting vertices read-only accesses to their neighbors and allowing unidirectional communication from master vertices to their replicas. Chaos [57] utilizes disk space on multiple machines to scale graph processing. PowerLira [27] is a system that dynamically applies different computation and partitioning strategies for different vertices. Gemini [84] is a distributed system that adapts Ligras hybrid push-pull computation model to a distributed form, facilitating efficient vertex-centric data update and message passing. Cube [80] uses a 3D partitioning strategy to reduce network traffic for certain machine learning and data mining problems. KickStarter [69] and Naiad [48] are systems that deal with streaming graphs.

GraphX [31] is a distributed graph system built on top of Spark, which is a general-purpose dataflow framework. GraphX provides a middle layer that offers a graph abstraction and "think like a vertex" interface for graph computation using low-level dataflow operators such as join and group-by available in Spark. GraphX's design goal is completely opposite to that of RStream— GraphX aims to *hide* the relational representation of data and operations in the underlying dataflow system to provide a user-familiar graph computation interface while RStream aims to *expose* relational representation of data and operations over the underlying graph engine to enable the expression and processing of graph mining algorithms that focus on patterns and structures.

Datalog Engines There exists a great deal of work that aims to improve efficiency and scalability for Datalog engines [13, 40, 73, 56, 61, 47]. These existing graph computation and Datalog systems are orthogonal to our work because none of them could support full graph mining. LogicBlox [13] is a system designed to reduce the complexity of software development for modern applications. It provides a LogiQL language, a unified and declarative language based on Datalog, for developers to express relations and constraints. SociaLite [40] is a Datalog engine designed for social network analysis. SociaLite programs are evaluated by parallel workers that use message passing to communicate.

Myria [73] provides runtime support for Datalog evaluation using a pipelined, parallel, distributed execution engine that evaluates a graph of operators. Datasets are sharded and stored in PostgreSQL instances at worker nodes. Both SociaLite and Myria support monotonic aggregation inside recursion using aggregate semantics based on the lattice-semantics of Ross and Sagiv [56]. BigDatalog [61] is a distributed Datalog engine built on top of Spark. It bases its monotonic aggregate (operational and declarative) semantics on work [47] that uses monotonic *w.r.t.* set-containment semantics. While RStream takes inspiration from Datalog, our experimental results show that RStream is much more efficient than existing Datalog engines on graph mining workloads.

Dataflow Systems Many dataflow systems [79, 11, 9, 21, 25] were developed. Systems such as Spark [79] and Asterix [10] provide relational operations for dataset transformations. While RStream takes inspiration from these systems, it is built specifically for graph mining, and thus has to overcome unique challenges that do not exist in existing systems.

At first glance, RStream's GRAS model is similar to a chain of MapReduce tasks — *e.g.*, the input data first gets shuffled into streaming partitions; relational expressions are next applied and the generated data is re-shuffled before the next relational phase comes. The key difference between these two model lies in the semantics — the GRAS abstraction that we built enables developers to easily develop and reason about mining algorithms by composing structures of smaller sizes into large sizes, while generic MapReduce tasks do not have any semantics. Join is a critical relational operation that has been extensively studied in the database community [5, 50, 15, 14]. While there exist many efficient join implementations such as worst-case optimal join [50], RStream is largely orthogonal to these prior works — future work could improve RStream with a more efficient join implementation.

7 Conclusion

This paper presents RStream, the first single-machine, out-of-core graph mining system. RStream employs a new GRAS programming model that uses a combination of GAS and relational algebra to support a wide variety of mining algorithms. At the low level, RStream leverages tuple streaming to efficiently implement relational operations. Our experimental results demonstrate that RStream can be more efficient than state-of-the-art distributed mining systems. We hope that these promising results will encourage future work that builds disk-based systems to scale expensive mining algorithms.

Acknowledgements

We would like to thank the many anonymous reviewers for their valuable and thorough comments. We are especially grateful to our shepherd Frans Kaashoek for his extensive feedback, helping us improve the paper substantially.

This work is supported by NSF grants CCF-1409829, CNS-1613023, CNS-1703598, and CNS-1763172, as well as by ONR grants N00014-16-1-2149, N00014-16-1-2913, and N00014-18-1-2037.

References

[1] Orkut social network. http://snap.stanford.edu/data/com-Orkut.html.

[2] The LogicBlox Datalog engine. http://www.logicblox.com/, 2016.

[3] Bliss: A tool for computing automorphism groups and canonical labelings of graphs. http://www.tcs.hut.fi/Software/bliss/, 2017.

[4] ABDELHAMID, E., ABDELAZIZ, I., KALNIS, P., KHAYYAT, Z., AND JAMOUR, F. ScaleMine: Scalable parallel frequent subgraph mining in a single large graph. In *SC* (2016), pp. 61:1–61:12.

[5] ABITEBOUL, S., HULL, R., AND VIANU, V., Eds. *Foundations of Databases: The Logical Level*, 1st ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[6] ABOULNAGA, A., XIANG, J., AND GUO, C. Scalable maximum clique computation using mapreduce. In *ICDE* (2013), pp. 74–85.

[7] AGRAWAL, R., AND SRIKANT, R. Mining sequential patterns. In *ICDE*, pp. 3–14.

[8] AI, Z., ZHANG, M., WU, Y., QIAN, X., CHEN, K., AND ZHENG, W. Squeezing out all the value of loaded data: An out-of-core graph processing system with reduced disk i/o. In *USENIX ATC* (2017), pp. 125–137.

[9] ALSUBAIEE, S., ALTOWIM, Y., ALTWAIJRY, H., BEHM, A., BORKAR, V., BU, Y., CAREY, M., GROVER, R., HEILBRON, Z., KIM, Y.-S., LI, C., ONOSE, N., PIRZADEH, P., VERNICA, R., AND WEN, J. ASTERIX: An open source system for "big data" management and analysis (demo). *Proc. VLDB Endow. 5*, 12 (2012), 1898–1901.

[10] ALSUBAIEE, S., BEHM, A., BORKAR, V., HEILBRON, Z., KIM, Y.-S., CAREY, M. J., DRESELER, M., AND LI, C. Storage management in asterixdb. *Proc. VLDB Endow. 7*, 10 (2014), 841–852.

[11] Hadoop: Open-source implementation of MapReduce. http://hadoop.apache.org.

[12] APARICIO, D. O., RIBEIRO, P. M. P., AND D. SILVA, F. M. A. Parallel subgraph counting for multicore architectures. In *IPDPS* (2014), pp. 34–41.

[13] AREF, M., TEN CATE, B., GREEN, T. J., KIMELFELD, B., OLTEANU, D., PASALIC, E., VELDHUIZEN, T. L., AND WASHBURN, G. Design and implementation of the LogicBlox system. In *SIGMOD* (2015), pp. 1371–1382.

[14] ATSERIAS, A., GROHE, M., AND MARX, D. Size bounds and query plans for relational joins. In *FOCS* (2008), pp. 739–748.

[15] AVNUR, R., AND HELLERSTEIN, J. M. Eddies: Continuously adaptive query processing. pp. 261–272.

[16] BABAI, L., AND LUKS, E. M. Canonical labeling of graphs. In *STOC* (1983), pp. 171–183.

[17] BACKSTROM, L., HUTTENLOCHER, D., KLEINBERG, J., AND LAN, X. Group formation in large social networks: Membership, growth, and evolution. In *KDD* (2006), pp. 44–54.

[18] BAHMANI, B., KUMAR, R., AND VASSILVITSKII, S. Densest subgraph in streaming and MapReduce. *Proc. VLDB Endow. 5*, 5 (2012), 454–465.

[19] BHUIYAN, M. A., AND HASAN, M. A. An iterative mapreduce based frequent subgraph mining algorithm. *IEEE Transactions on Knowledge and Data Engineering 27*, 3 (2015), 608–620.

[20] BOLDI, P., AND VIGNA, S. The WebGraph framework I: Compression techniques. In *WWW* (2004), pp. 595–601.

[21] BORKAR, V. R., CAREY, M. J., GROVER, R., ONOSE, N., AND VERNICA, R. Hyracks: A flexible and extensible foundation for data-intensive computing. pp. 1151–1162.

[22] BRINGMANN, B., AND NIJSSEN, S. What is frequent in a single graph? In *Proceedings of the 12th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining (PAKDD '08)* (2008), T. Washio, E. Suzuki, K. M. Ting, and A. Inokuchi, Eds., pp. 858–863.

[23] BRON, C., AND KERBOSCH, J. Algorithm 457: Finding all cliques of an undirected graph. *Commun. ACM 16*, 9 (1973), 575–577.

[24] BU, Y., BORKAR, V., JIA, J., CAREY, M. J., AND CONDIE, T. Pregelix: Big(ger) graph analytics on a dataflow engine. *Proc. VLDB Endow. 8*, 2 (Oct. 2014), 161–172.

[25] CHAIKEN, R., JENKINS, B., LARSON, P.-A., RAMSEY, B., SHAKIB, D., WEAVER, S., AND ZHOU, J. SCOPE: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow. 1*, 2 (2008), 1265–1276.

[26] CHEN, R., DING, X., WANG, P., CHEN, H., ZANG, B., AND GUAN, H. Computation and communication efficient graph processing with distributed immutable view. In *HPDC* (2014), pp. 215–226.

[27] CHEN, R., SHI, J., CHEN, Y., AND CHEN, H. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. In *EuroSys* (2015), pp. 1:1–1:15.

[28] DI FATTA, G., AND BERTHOLD, M. R. Dynamic load balancing for the distributed mining of molecular structures. *IEEE Trans. Parallel Distrib. Syst. 17*, 8 (2006), 773–785.

[29] ELSEIDY, M., ABDELHAMID, E., SKIADOPOULOS, S., AND KALNIS, P. GraMi: Frequent subgraph and pattern mining in a single large graph. *Proc. VLDB Endow. 7*, 7 (2014), 517–528.

[30] GONZALEZ, J. E., LOW, Y., GU, H., BICKSON, D., AND GUESTRIN, C. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI* (2012), pp. 17–30.

[31] GONZALEZ, J. E., XIN, R. S., DAVE, A., CRANKSHAW, D., FRANKLIN, M. J., AND STOICA, I. GraphX: Graph processing in a distributed dataflow framework. In *OSDI* (2014), pp. 599–613.

[32] HALL, B. H., JAFFE, A. B., AND TRAJTENBERG, M. The NBER patent citation data file: Lessons, insights and methodological tools. Tech. Rep. 8498, National Bureau of Economic Research, 2001.

[33] HAN, W., MIAO, Y., LI, K., WU, M., YANG, F., ZHOU, L., PRABHAKARAN, V., CHEN, W., AND CHEN, E. Chronos: A graph engine for temporal graph analysis. In *EuroSys* (2014), pp. 1:1–1:14.

[34] HAN, W.-S., LEE, S., PARK, K., LEE, J.-H., KIM, M.-S., KIM, J., AND YU, H. TurboGraph: A fast parallel graph engine handling billion-scale graphs in a single PC. In *KDD* (2013), pp. 77–85.

[35] HILL, S., SRICHANDAN, B., AND SUNDERRAMAN, R. An iterative mapreduce approach to frequent subgraph mining in biological datasets. In *BCB* (2012), pp. 661–666.

[36] HUANG, Y., BASTANI, F., JIN, R., AND WANG, X. S. Large scale real-time ridesharing with service guarantee on road networks. *Proc. VLDB Endow. 7*, 14 (2014), 2017–2028.

[37] KESSL, R., TALUKDER, N., ANCHURI, P., AND ZAKI, M. J. Parallel graph mining with gpus. In *BIGMINE* (2014), pp. 1–16.

[38] KURAMOCHI, M., AND KARYPIS, G. Finding frequent patterns in a large sparse graph*. *Data Min. Knowl. Discov. 11*, 3 (Nov. 2005), 243–271.

[39] KYROLA, A., BLELLOCH, G., AND GUESTRIN, C. GraphChi: Large-scale graph computation on just a PC. In *OSDI* (2012), pp. 31–46.

[40] LAM, M. S., GUO, S., AND SEO, J. SociaLite: Datalog extensions for efficient social network analysis. In *ICDE* (2013), pp. 278–289.

[41] LIN, W., XIAO, X., AND GHINITA, G. Large-scale frequent subgraph mining in MapReduce. In *ICDE* (2014), pp. 844–855.

[42] LIN, Z., KAHNG, M., SABRIN, K. M., CHAU, D. H. P., LEE, H., , AND KANG, U. MMap: Fast billion-scale graph computation on a pc via memory mapping. In *BigData* (2014), pp. 159–164.

[43] LOW, Y., BICKSON, D., GONZALEZ, J., GUESTRIN, C., KYROLA, A., AND HELLERSTEIN, J. M. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow. 5*, 8 (2012), 716–727.

[44] LU, W., CHEN, G., TUNG, A. K. H., AND ZHAO, F. Efficiently extracting frequent subgraphs using MapReduce. In *Big Data* (2013), pp. 639–647.

[45] MAASS, S., MIN, C., KASHYAP, S., KANG, W., KUMAR, M., AND KIM, T. Mosaic: Processing a trillion-edge graph on a single machine. In *EuroSys* (2017), pp. 527–543.

[46] MALEWICZ, G., AUSTERN, M. H., BIK, A. J. C., DEHNERT, J. C., HORN, I., LEISER, N., CZAJKOWSKI, G., AND INC, G. Pregel: A system for large-scale graph processing. In *SIGMOD* (2010), pp. 135–146.

[47] MAZURAN, M., SERRA, E., AND ZANIOLO, C. Extending the power of datalog recursion. *The VLDB Journal 22*, 4 (Aug. 2013), 471–493.

[48] MURRAY, D. G., MCSHERRY, F., ISAACS, R., ISARD, M., BARHAM, P., AND ABADI, M. Naiad: A timely dataflow system. In *SOSP* (2013), pp. 439–455.

[49] NAJEEBULLAH, K., KHAN, K. U., NAWAZ, W., AND LEE, Y.-K. Bishard parallel processor: A disk-based processing engine for billion-scale graphs. *Journal of Multimedia & Ubiquitous Engineering 9*, 2 (2014), 199–212.

[50] NGO, H. Q., PORAT, E., RÉ, C., AND RUDRA, A. Worst-case optimal join algorithms: [extended abstract]. In *PODS* (2012), pp. 37–48.

[51] NGUYEN, D., LENHARTH, A., AND PINGALI, K. A lightweight infrastructure for graph analytics. In *SOSP* (2013), pp. 456–471.

[52] PAGE, L., BRIN, S., MOTWANI, R., AND WINOGRAD, T. The PageRank citation ranking: Bringing order to the web. Tech. rep., Stanford University, 1998.

[53] PEARCE, R., GOKHALE, M., AND AMATO, N. M. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In *SC* (2010), pp. 1–11.

[54] PRŽULJ, N. Biological network comparison using graphlet degree distribution. *Bioinformatics 23*, 2 (2007), e177–e183.

[55] RIBEIRO, P., AND SILVA, F. G-Tries: A data structure for storing and finding subgraphs. *Data Min. Knowl. Discov. 28*, 2 (2014), 337–377.

[56] ROSS, K. A., AND SAGIV, Y. Monotonic aggregation in deductive databases. In *PODS* (1992), pp. 114–126.

[57] ROY, A., BINDSCHAEDLER, L., MALICEVIC, J., AND ZWAENEPOEL, W. Chaos: Scale-out graph processing from secondary storage. In *SOSP* (2015), pp. 410–424.

[58] ROY, A., MIHAILOVIC, I., AND ZWAENEPOEL, W. X-Stream: Edge-centric graph processing using streaming partitions. In *SOSP* (2013), pp. 472–488.

[59] SHAO, Y., CUI, B., CHEN, L., MA, L., YAO, J., AND XU, N. Parallel subgraph listing in a large-scale graph. In *SIGMOD* (2014), pp. 625–636.

[60] SHI, J., YAO, Y., CHEN, R., CHEN, H., AND LI, F. Fast and concurrent RDF queries with rdma-based distributed graph exploration. In *USENIX ATC* (2016), pp. 317–332.

[61] SHKAPSKY, A., YANG, M., INTERLANDI, M., CHIU, H., CONDIE, T., AND ZANIOLO, C. Big data analytics with datalog queries on spark. In *SIGMOD* (2016), pp. 1135–1149.

[62] SHKAPSKY, A., YANG, M., AND ZANIOLO, C. Optimizing recursive queries with monotonic aggregates in DeALS. In *ICDE* (2015), pp. 867–878.

[63] SHUN, J., AND BLELLOCH, G. E. Ligra: A lightweight graph processing framework for shared memory. In *PPoPP* (2013), pp. 135–146.

[64] SLOTA, G. M., AND MADDURI, K. Parallel color-coding. *Parallel Comput. 47*, C (2015), 51–69.

[65] TALUKDER, N., AND ZAKI, M. J. A distributed approach for graph mining in massive networks. *Data Mining and Knowledge Discovery: Special Issue on ECML/PKDD 2016 Journal Track Papers 30*, 5 (2016), 1024–1052.

[66] TEIXEIRA, C. H. C., FONSECA, A. J., SERAFINI, M., SIGANOS, G., ZAKI, M. J., AND ABOULNAGA, A. Arabesque: A system for distributed graph mining. In *SOSP* (2015), pp. 425–440.

[67] TEIXEIRA, C. H. C., FONSECA, A. J., SERAFINI, M., SIGANOS, G., ZAKI, M. J., AND ABOULNAGA, A. Arabesque: A system for distributed graph mining - extended version. *ArXiv e-prints* (Oct. 2015).

[68] VORA, K., GUPTA, R., AND XU, G. Synergistic analysis of evolving graphs. *ACM Trans. Archit. Code Optim. 13*, 4 (2016), 32:1–32:27.

[69] VORA, K., GUPTA, R., AND XU, G. KickStarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *ASPLOS* (2017).

[70] VORA, K., XU, G., AND GUPTA, R. Load the edges you need: A generic I/O optimization for disk-based graph processing. In *USENIX ATC* (2016), pp. 507–522.

[71] WANG, C., AND PARTHASARATHY, S. Parallel algorithms for mining frequent structural motifs in scientific data. In *ICS* (2004), pp. 31–40.

[72] WANG, G., XIE, W., DEMERS, A., AND GEHRKE, J. Asynchronous large-scale graph processing made easy. In *CIDR* (2013).

[73] WANG, J., BALAZINSKA, M., AND HALPERIN, D. Asynchronous and fault-tolerant recursive datalog evaluation in shared-nothing engines. *PVLDB 8*, 12 (2015), 1542–1553.

[74] WANG, K., HUSSAIN, A., ZUO, Z., XU, G., AND SANI, A. A. Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code. In *ASPLOS* (2017), pp. 389–404.

[75] WANG, K., XU, G., SU, Z., AND LIU, Y. D. GraphQ: Graph query processing with abstraction

refinement—programmable and budget-aware analytical queries over very large graphs on a single PC. In *USENIX ATC* (2015), pp. 387–401.

[76] WU, M., YANG, F., XUE, J., XIAO, W., MIAO, Y., WEI, L., LIN, H., DAI, Y., AND ZHOU, L. GraM: Scaling graph computation to the trillions. In *SoCC* (2015), pp. 408–421.

[77] YAN, D., CHEN, H., CHENG, J., ÖZSU, M. T., ZHANG, Q., AND LUI, J. C. S. G-thinker: Big graph mining made easier and faster. *CoRR abs/1709.03110* (2017).

[78] YAN, X., AND HAN, J. gSpan: Graph-based substructure pattern mining. In *ICDM* (2002), pp. 721–.

[79] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Spark: Cluster computing with working sets. HotCloud, p. 10.

[80] ZHANG, M., WU, Y., CHEN, K., QIAN, X., LI, X., AND ZHENG, W. Exploring the hidden dimension in graph processing. In *OSDI* (2016), pp. 285–300.

[81] ZHANG, M., WU, Y., ZHUO, Y., QIAN, X., HUAN, C., AND CHEN, K. Wonderland: A novel abstraction-based out-of-core graph processing system. In *ASPLOS* (2018), pp. 608–621.

[82] ZHAO, Z., WANG, G., BUTT, A. R., KHAN, M., KUMAR, V. S. A., AND MARATHE, M. V. SAHAD: Subgraph analysis in massive networks using hadoop. In *IPDPS* (2012), pp. 390–401.

[83] ZHENG, D., MHEMBERE, D., BURNS, R., VOGELSTEIN, J., PRIEBE, C. E., AND SZALAY, A. S. FlashGraph: processing billion-node graphs on an array of commodity ssds. In *FAST* (2015), pp. 45–58.

[84] ZHU, X., CHEN, W., ZHENG, W., AND MA, X. Gemini: A computation-centric distributed graph processing system. In *OSDI* (2016), pp. 301–316.

[85] ZHU, X., HAN, W., AND CHEN, W. GridGraph: Large scale graph processing on a single machine using 2-level hierarchical partitioning. In *USENIX ATC* (2015), pp. 375–386.