GraphFrames: An Integrated API for Mixing Graph and Relational Queries

Ankur Dave[★], Alekh Jindal[□], Li Erran Li[†], Reynold Xin[◇], Joseph Gonzalez[★], Matei Zaharia[▽ ◇]
[★]*University of California, Berkeley* [▽]*MIT* [†] *Uber Technologies* [◇]*Databricks* [□]*Microsoft*

ABSTRACT

Graph data is prevalent in many domains, but it has usually required specialized engines to analyze. This design is onerous for users and precludes optimization across complete workflows. We present GraphFrames, an integrated system that lets users combine graph algorithms, pattern matching and relational queries, and optimizes work across them. GraphFrames generalize the ideas in previous graph-on-RDBMS systems, such as GraphX and Vertexica, by letting the system materialize multiple *views* of the graph (not just the specific triplet views in these systems) and executing both iterative algorithms and pattern matching using joins. To make applications easy to write, GraphFrames provide a concise, declarative API based on the "data frame" concept in R that can be used for both interactive queries and standalone programs. Under this API, GraphFrames use a graph-aware join optimization algorithm across the whole computation that can select from the available views.

We implement GraphFrames over Spark SQL, enabling parallel execution on Spark and integration with custom code. We find that GraphFrames make it easy to express end-to-end workflows and match or exceed the performance of standalone tools, while enabling optimizations across workflow steps that cannot occur in current systems. In addition, we show that GraphFrames' view abstraction makes it easy to further speed up interactive queries by registering the appropriate view, and that the combination of graph and relational data allows for other optimizations, such as attribute-aware partitioning.

1. INTRODUCTION

Analyzing the graphs of relationships that occur in modern datasets is increasingly important, in domains including commerce, social networks, and medicine. To date, this analysis has been done through specialized systems like Neo4J [17], Titan [16] and GraphLab [11]. These systems offer two main capabilities: *pattern matching* to find subgraphs of interest [17] and *graph algorithms* such as shortest paths and PageRank [12, 11, 7].

While graph analytics is powerful, running it in a separate system is both onerous and inefficient. Most workflows involve building a graph from existing data, likely in a relational format, then running search or graph algorithms on it, and then performing further computations on the result. With isolated graph analysis systems, users have to move data manually and there is no optimization of computation across these phases of the workflow. Several recent systems have started to bridge this gap by running graph algorithms on a relational engine [7, 9], but they have no support for pattern matching and do not optimize across graph and relational queries.

```
gf = GraphFrame(vertices, edges)

triples = gf.pattern("(x:User)->(p:Product)<-(y:User)")

pairs.where(pairs.p.category == "Books")
     .groupBy(pairs.p.name)
     .count()
```

Listing 1: An example of the GraphFrames API. We create a GraphFrame from two tables of vertices and edges, and then we search for all instances *pattern*, namely two users that bought the same product. The result of this search is another table that we can then perform filtering and aggregation on. The system will optimize across these steps, e.g., pushing the filter above the pattern search.

We present GraphFrames, an integrated system that can combine relational processing, pattern matching and graph algorithms and optimize computations across them. GraphFrames generalize the ideas behind GraphX [7] and Vertexica [9] by maintaining arbitrary *views* of a graph (e.g., triplets or triangles) and executing queries using joins across them. They then optimize execution across the relational and graph portions of the computation. A key challenge in achieving this goal is query planning. For this purpose, we extend a graph-aware dynamic programming algorithm by Huang et al. [8] to select among multiple input views and compute a join plan. We also propose an algorithm for suggesting new views based on the query workload.

To make complete graph analytics workflows easy to write, GraphFrames provide a declarative API similar to "data frames" in R, Python and Spark that integrates into procedural languages like Python. Users build up a computation out of relational operators, pattern matching, and calls to algorithms, as shown in Listing 1. The system then optimizes across these steps, selecting join plans and performing algebraic optimizations. Similar to systems like Pig [14] and Spark SQL [4], the API makes it easy to build a computation incrementally while receiving the full benefits of relational optimization. Finally, the GraphFrames API is also designed to be used *interactively*: users can launch a session, define views that will aid their queries, and query data interactively from a Python shell. Unlike current tools, GraphFrames let analysts perform their complete workflow in a single system.

We have implemented GraphFrames over Spark SQL [4], and made them compatible with Spark's existing DataFrame API. We show that GraphFrames match the performance of other distributed

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2016 ACM. ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

graph engines for various tasks, while enabling optimizations across the tasks that would not happen in other systems. Support for multiple views of the graph adds significant benefits: for example, materializing a few simple views can speed up queries by 10× over the algorithm in [8]. Finally, by building on Spark, GraphFrames interoperate easily with custom UDFs (e.g., ETL code) and with Spark's machine learning library and external data sources.

In summary, our contributions are:

- A declarative API that lets users combine relational processing, pattern matching and graph algorithms into complex workflows and optimizes across them.
- An execution strategy that generalizes those in Vertexica and GraphX to support multiple views of the graph.
- A graph-aware query optimization algorithm that selects join plans based on the available views.
- An implementation and evaluation of GraphFrames on Spark.

2. GRAPHFRAME API

The main programming abstraction in GraphFrames' API is a *GraphFrame*. Conceptually, it consists of two relations (tables) representing the vertices and edges of the graph, as well as a set of materialized views of subgraphs. The vertices and edges may have multiple *attributes* that are used in queries. The views are defined using *patterns* to match various shapes of subgraphs, as we shall describe in Section 2.2.2. For example, a user might create a view of all the triangles in the graph, which can then be used to quickly answer other queries involving triangles.

GraphFrames expose a concise language-integrated API that unifies graph analytics and relational queries. We based this API on DataFrames, a common abstraction for data science in Python and R that is also available as a declarative API on Spark SQL [4]. In this section, we first cover some background on DataFrames, and then discuss the additional operations available on GraphFrames. We demonstrate the generality of GraphFrames for analytics by mapping the core primitives in GraphX into GraphFrame operations. Finally, we discuss how GraphFrames integrate with the rest of Apache Spark (e.g., the machine learning library).

All the code examples are shown in Python. We show the GraphFrame API itself in Scala because it explicitly lists data types.

2.1 DataFrame Background

DataFrames are the main programming abstraction for manipulating tables of structured data in R, Python, and Spark. Different variants of DataFrames have slightly different semantics. For the purpose of this paper, we describe Spark's DataFrame implementation, which we build on [4]. Each DataFrame contains data grouped into named columns, and keeps track of its own schema. A DataFrame is equivalent to a table in a relational database, and can be transformed into new DataFrames using various relational operators available in the API.

As an example, the following code snippet computes the number of female employees in each department by performing aggregation and join between two data frames:

```
employees
  .join(dept, employees.deptId == dept.id)
  .where(employees.gender == "female")
  .groupBy(dept.id, dept.name)
  .agg(count("name"))
```

`employees` is a DataFrame, and `employees.deptId` is an expression representing the `deptId` column. Expression objects have many operators that return new expressions, including the usual comparison operators (e.g., == for equality test, > for greater than) and arithmetic ones (+, -, etc). They also support aggregates, such as `count("name")`.

Internally, a DataFrame object represents a *logical plan* to compute a dataset. A DataFrame does not need to be materialized, until the user calls a special "output operation" such as `save`. This enables rich optimization across all operations that were used to build the DataFrame.[1]

In terms of data type support, DataFrame columns support all major SQL data types, including boolean, integer, double, decimal, string, date, and timestamp, as well as complex (i.e., non-atomic) data types: structs, arrays, maps and unions. Complex data types can also be nested together to create more powerful types. In addition, DataFrame also supports user-defined types [4].

2.2 GraphFrame Data Model

A GraphFrame is logically represented as two DataFrames: an edge DataFrame and a vertex DataFrame. That is to say, edges and vertices are represented in separate DataFrames, and each of them can contain attributes that are part of the supported types. Take a social network graph for an example. The vertices can contain attributes including name (string), age (integer), and geographic location (a struct consisting of two floating point values for longitude and latitude), while the edges can contain an attribute about the time a user friended another (timestamp). The GraphFrame model supports user-defined attributes with each vertex and edges, and thus is equivalent to the property graph model used in many graph systems including GraphX and GraphLab. GraphFrame is more general than Pregel/Giraph since GraphFrame supports user-defined attributes on edges.

Similar to DataFrames, a GraphFrame object is internally represented as a logical plan, and as a result the declaration of a GraphFrame object does not necessarily imply the materialization of its data.

Next, we explain how a GraphFrame can be constructed and operations available on them.

```
class GraphFrame {
  // Different views on the graph
  def vertices: DataFrame
  def edges: DataFrame
  def triplets: DataFrame
  // Pattern matching
  def pattern(pattern: String): DataFrame

  // Relational-like operators
  def filter(predicate: Column): GraphFrame
  def select(cols: Column*): GraphFrame
  def joinV(v: DataFrame, predicate: Column): GraphFrame
  def joinE(e: DataFrame, predicate: Column): GraphFrame

  // View creation
  def createView(pattern: String): DataFrame

 // Partition function
 def partitionBy(Column*) GraphFrame
}
```

Listing 2: GraphFrame API in Scala

2.2.1 Graph Construction

A GraphFrame can be constructed using two DataFrames: a vertex DataFrame and an edge DataFrame. A DataFrame is merely a logical view (plan) and can support a wide range of sources that

[1] This aspect of Spark DataFrames is different from R and Python; in those languages, DataFrame contents are materialized eagerly after each operation, which precludes optimization across the whole logical plan [4].

implement a data source API. Some examples of a DataFrame input include:

- a table registered in Spark SQL's system catalog
- a table in an external relational database through JDBC
- JSON, Parquet, Avro, CSV files on disk
- a table in memory in columnar format
- a set of documents in ElasticSearch or Solr
- results from relational transformations on the above

The following code demonstrates constructing a graph using a user table in a live transactional database and the edges table from some JSON based log files in Amazon S3:

```
users = read.jdbc("mysql://...")
likes = read.json("s3://...")
graph = GraphFrame(users, likes)
```

Again, since DataFrames and GraphFrames are logical abstractions, the above code does not imply that `users`, `likes`, or `graph` are materialized.

2.2.2 *Edges, Vertices, Triplets, and Patterns*

A GraphFrame exposes four tabular views of a graph: edges, vertices, triplets, and a pattern view that supports specifying graph patterns using a syntax similar to the Cypher pattern language in Neo4J [17].

The edges view and the vertices view should be self-evident. The triplets view consists of each edge and its corresponding source and destination vertex attributes. It can actually be constructed using the following 3-way join:

```
e.join(v, v.id == e.srcId)
 .join(v, v.id == e.dstId)
```

We provide it directly since the triplets view is used commonly enough. Note that edges, vertices, and triplets views are also the three fundamental views in GraphX, and GraphFrames is at least as expressive as GraphX from the perspective of views.

In addition to the three basic tabular views, a GraphFrame also supports a pattern operator that accepts a graph pattern in a Cypher-like syntax and returns a DataFrame consisting of edges and vertices specified by the pattern. This pattern operator enables easy expression of pattern matching in graphs.

Typical graph patterns consist of two nodes connected by a directed edge relationship, which is represented in the format *()-[]->()*. Nodes are specified using parentheses (), and relationships are specified using square brackets, []. Nodes and relationships are linked using an arrow-like syntax to express edge direction. The same node may be referenced in multiple relationships, allowing relationships to be composed into complex patterns. Additionally, nodes and edges can be constrained using inline type predicates expressed using colons.

For example, the following snippet shows a user `u` who viewed both item `x` and item `y`.

```
(u:Person)-[viewed]->(x: Item), u-[viewed]->(y: Item)
```

The resulting DataFrame from the above pattern should contain 3 structs: `u`, `x`, and `y`.

Note that the pattern operator is a simple and intuitive way to specify pattern matching. Under the hood it is implemented using the join and filter operators available on a GraphFrame. We provide it because it is often more natural to reason about graphs using patterns than using relational joins. The pattern operator can also be combined with other operators, as demonstrated in the next subsection.

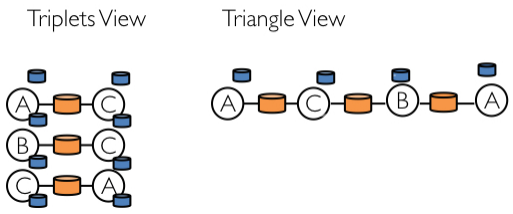


Figure 1: View Reuse

Programmatic Pattern Generation and Pattern Library Patterns such as cliques can be cumbersome to specify for interactive queries. We therefore provide a pattern library to programmatically generate and compose patterns. For example, a star of size K can be specified as `(hub, spokes) = star(nodePred, edgePred, K)`. `nodePred` and `edgePred` filters out nodes and edges. The pattern returns a hub and a list of K-1 spokes. Their names can then be used in further pattern specification, or materialized immediately.

2.2.3 *View Creation*

To enable reuse of computation, GraphFrames support view creation. The system materializes the view internally and uses it to speed up subsequent queries. For example, in Figure 1, if we create a view on triplets, we can reuse it to create a triangle view. The system will avoid rematerializing the triplet view for computing the triangle view. We will discuss how our query planner performs view selection to optimize the computation in the next section.

2.2.4 *Relational Operators*

Since a GraphFrame exposes the four tabular views, it already supports all the relational operators (e.g. project, filter, join) available on these tabular views. Relational operators on these views can already support many graph analysis algorithms. For example, the following snippet computes the out-degree for each vertex:

```
g.edges.groupBy(g.edges.srcId).count()
```

In addition to relational operators on the tabular views, a GraphFrame also includes a few relational-like operators on a graph, namely `select`, `filter`, and `join`. These graph operators apply the corresponding relational operators to the vertices and/or edges, as appropriate. For example, `filter` applies a predicate to the vertex and edge tables, then removes dangling edges to ensure graph integrity. Similarly, `join` allows updating the vertex or edge table by joining in external data.

2.2.5 *Attribute-Based Partitioning*

Similar to GraphX, GraphFrames by default partitions a graph based on the natural partitioning scheme of the edges. In [7], it was shown that natural partitioning can lead to great performance when the input is pre-partitioned.

In addition, GraphX supports partitioning a graph based on arbitrary vertex or edge attributes. This is more general than GraphX or Giraph because they only support partitioning on vertex identifiers. This enables users to partition a graph based on their domain-specific knowledge that can lead to strong data locality and minimize data communications.

Take the Amazon dataset [13] for example. The following snippet partitions the bipartite graph based on product categories:

```
g.partitionBy(g.vertices.productCategory)
```

Intuitively, customers are more likely to buy products in the same category. Partitioning the Amazon graph this way puts products of the same categories and their associated edges closer to each other.

2.2.6 *User-defined Functions*

GraphFrame also supports arbitrary user-defined functions (UDFs) in Scala, Java, and Python. The `udf` function accepts a lambda function as input and creates an expression that can be used in relational and graph projection. For example, given a `model` object for a machine learning model, we could create a UDF predicting some user behavior based on users' `age` and `registrationTime` attributes.

```
model: LogisticRegressionModel = ...
predict = udf(lambda x, y: model.predict(Vector(x, y)))
g.select(
  predict(g.vertices.age, g.vertices.registrationTime))
```

Unlike database systems which often require UDFs to be defined in a separate programming environment that is different from the primary query interfaces, our GraphFrame API supports inline definition of UDFs. We do not need complicated packaging and registration process found in other database systems.

2.3 Generality of GraphFrames

As simple as it is, the GraphFrame abstraction is powerful enough to express many workloads. We demonstrate the expressiveness by mapping all GraphX operators to operators in GraphFrame. Since GraphX can be used to model the programing abstractions in GraphLab, Pregel, and BSP [7], by mapping GraphX operations to GraphFrame, we demonstrate that GraphFrame is at least as expressive as GraphX, GraphLab, Pregel, and BSP.

GraphX's operators can be divided into three buckets: collection views, relational-like operators, and graph-parallel computations. Section 2.2.2 already demonstrated that GraphFrame provides all the three fundamental collection views in GraphX (edges, vertices, and tripets).

All relational-like operators in GraphX can be trivially mapped one-to-one to GraphFrame operators. For example, the `select` operator is a superset of GraphX's `mapV` and `mapE` operators, and `joinV` and `joinE` are the generalized variant of GraphX's `leftJoinV` and `leftJoinE` operators. The `filter` operator is a more general version of GraphX's `subgraph` operator.

In GraphX, graph-parallel computations consist of `aggregateMessages`[2] and its variants. Similar to the Gather phase in the GAS abstraction, `aggregateMessages` encodes a two-stage process of graph-parallel computation. Logically, it is the composition of a projection followed by an aggregation on the triplets view. In [7], it was illustrated using the following SQL query:

```
SELECT t.dstId, reduceF(mapF(t)) AS msgSum
FROM triplets AS t GROUP BY t.dstId
```

This SQL query can indeed be expressed using the following GraphFrame operators:

```
g.triplets
 .select(mapF(g.triplets.attribute[s]).as("mapOutput"))
 .groupBy(g.triplets.dstId)
 .agg(reduceF("mapOutput"))
```

We demonstrated that GraphFrame can support all the operators available in GraphX and consequently can support all operations in GraphLab, Pregel, and BSP. For convenience, we also provide similar APIs as GraphX's Pregel variant in GraphFrame for implementing iterative algorithms. We have also implemented common graph algorithms including connected components, PageRank, triangle counting.

2.4 Spark Integration

Because GraphFrames builds on top of Spark, this brings three benefits. First, GraphFrames can load data from and save data into existing Spark SQL data sources such as HDFS files in Json, Parquet format, HBASE, Cassandra, etc. Second, GraphFrame can use a growing list of machine learning algorithms in MLlib. Third, GraphFrames can call Spark DataFrame API. As an example, the following code reads user-product rating information from HDFS into a DataFrame. We then select the review text and use user ID and product ID pair as the key. We can call the topic model to learn the topics of reviews. With the topics, we can compute similar products, etc as in [13] and do graph pattern matching to uncover user communities who bought similar products.

```
corpus = rating.read.parquet("hdfs:///...")
 .select(pair(user_id, product_id), review_txt)
ldaModel = LDA.train(corpus, k=10000)
topics = ldaModel.topicsMatrix()
```

2.5 Putting It Together

```
# 1. ETL
# users: [id: int, attributes: MapType(user_name)]
# products: [id: int, attributes: MapType(brand,
#     category, price)]
# ratings: [user_id: int, product_id: int,
#     rating: int, review_text: string]
# cobought: [product_1_id: int, product_2_id: int]

vertices = users.union(products)
graph = GraphFrame(vertices, ratings)

# 2. Run ALS to get top 1M inferred recommendations
# predictedRatings: [user_id: int, product_id: int,
#     predicted_rating: int]
predictedRatings = ALS.train(graph, iterations=20)
  .recommendForAll(1e6)

densifiedGraph = GraphFrame(vertices,
  ratings.union(predictedRatings).union(cobought))

# 3. Find groups of users with the same interests
densifiedGraph.pattern("""(u1)-[r1]->(p1);
  (u2)-[r2]->(p2); (p1)-[]->(p2)""")
  .filter("r1.rating > 3 && r2.rating > 3")
  .select("u1.id", "u2.id")
```

Listing 3: GraphFrame End-to-End Example in Python

We show that the ease of developing an end-to-end graph analytics pipeline with an example in Listing 3. The example is an ecommerce application that groups users of similar interests.

The first step is to perform ETL to extract information on users, products, ratings and cobought. They are represented as DataFrames. We then construct a GraphFrame `graph`. The vertices contain both user nodes and product nodes. The edges are between users and products. An edge exists between a user and a product if the user rated the product. This is a bipartite graph.

For the second step, we run collaborative filtering to compute predicted ratings of users, i.e. to uncover latent ratings not present in the dataset. We then create a graph `densifiedGraph` with the same vertex node as `graph` and more edges by adding product-product edges. A product-product edge is added if the two are cobought.

As the final step, we will find pairs of users who have good ratings for at least two products together. We can also find group of users of size K.

This example shows the ease of using the GraphFrames API. We performed ETL, iterative graph algorithms and graph pattern matching in one system. It is much more intuitve than coding the pipeline in SQL. Language integration also makes it easy to plug in UDFs. For example, we can create a UDF to extract product topics and topics user interested.

In the next section, we will high light the opportunities for joint optimization.

[2] `aggregateMessages` was called `mrTriplets` in [7], but renamed in the open source GraphX system.

3. IMPLEMENTATION

We implemented GraphFrames as a library on top of Spark SQL. The library consists of the GraphFrame interface described in Section § 2, a pattern parser, and our view-based query planner. We also made improvements to Spark SQL's Catalyst optimizer to support GraphFrames.

Each GraphFrame is represented as two Spark DataFrames (a vertex DataFrame and a edge DataFrame), a collection of user-defined views. Implementations of each of the GraphFrame methods follow naturally from this representation, and the GraphFrame interface is 250 lines of Scala. The GraphFrame operations delegate to the pattern parser and the query planner.

Our query planner is implemented as a layer on top of Spark Catalyst, taking patterns as input, collecting statistics using Catalyst APIs, and emitting a Catalyst logical plan. At query time, the planner receives the user-specified views from the GraphFrame interface. The planner additionally can suggest views when requested by the user. The query planner also accepts custom attribute-based partitioners which it uses to make more accurate cost estimates and incorporates into the generated plans.

To simplify the query planner, we modified Catalyst to support join elimination when allowed by the foreign key relationship between vertices and edges. This change required adding support for unique and foreign key constraints on DataFrames to Spark SQL. Join elimination enables the query planner to emit one join per referenced vertex, and joins unnecessary to produce the final output will be eliminated. This change required 800 lines of Scala.

Our pattern parser uses the Scala parser combinator library and is implemented in 50 lines of Scala.

Finally, building on top of Spark enables GraphFrames to easily integrate with data sources and call its machine learning libraries.

3.1 Query Optimization

The GraphFrame query planner extends the dynamic programming algorithm of Huang et al. [8] to the distributed setting and adds a view rewrite capability. The user can register arbitrary materialized views and the planner will automatically rewrite the query to reuse a materialized view when appropriate. This is useful because pattern queries could be very expensive to run and reusing computations across several queries can improve the user experience. Our optimizer also offers suggestions for which views to create. See appendix for details on this algorithm.

4. EVALUATION

In this section we demonstrate that GraphFrames benefit greatly in some cases by materializing appropriate views and outperform a mix of systems on analytics pipelines by avoiding communication between systems and optimizing across the entire pipeline.

All experiments were conducted on Amazon EC2 using 8 `r3.2xlarge` worker nodes in November 2015. Each node has 8 virtual cores, 61 GB of memory, and one 160 GB SSD.

4.1 Impact of Views

We first demonstrate that materializing the appropriate views reduces query time, and in some cases can greatly improve the plan selection. We ran the six queries shown in Figure 2a on a web graph dataset released by Google in 2002 [10]. This graph has 875,713 vertices and 5,105,039 edges. It is the largest graph used for these queries in [8]. Before running these queries we registered the views listed in Table 1. We then ran each query with and without view rewrite enabled. The results are reported in Figure 2b.

Queries 1, 2, 3, and 6 do not benefit much from views, because the main cost in these queries comes from generating unavoidably large intermediate result sets. For example, in Query 1 the bidirectional edge between vertices A and B can use the 2-cycle view, but by far the more expensive part of the plan is joining C and D to the view, because this generates all pairs of such vertices.

View	Query	Size in Google graph
2-cycle	`(a)->(b)->(a)`	1,565,976
V	`(c)<-(a)->(b)`	67,833,471
Triangle	`(a)<-(b)->(c)->(a)`	28,198,954
3-cycle	`(a)->(b)->(c)->(a)`	11,669,313

Table 1: Views registered in the system to explore their impact on queries in [8]

However, in Query 4 we observe a large speedup when using views. In Query 4, the order-of-magnitude speedup is because the view equivalence check exposes an opportunity to reuse an intermediate result that the planner would otherwise miss. This is because the reuse requires recognizing that two subgraphs are isomorphic despite having different node labels, a problem that is difficult in general but becomes much easier with the right choice of view. In particular, the Triangle view is applicable both to the BCD triangle and the BCE triangle in Query 4, so the planner can replace the naive 5-way join with a single self-join of the Triangle view with equality constraints on vertices B and C.

Additionally, in Query 5, precomputing views speeds up the main query by a factor of 2 by moving the work of computing the BCD and BED triangles from the main query into the Triangle 2 and 3-cycle views. These views are expensive to create, and since they are common patterns it is reasonable to precompute them.

4.2 End-to-End Pipeline Performance

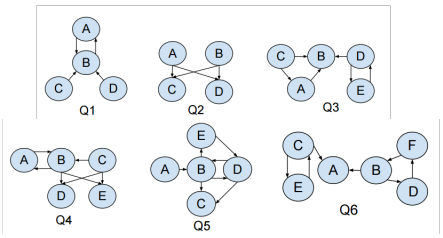
We next evaluate the end-to-end performance of a multi-step pipeline that finds groups of users with the same interests in an Amazon review dataset. We will see that using Spark and GraphFrames for the whole pipeline allows more powerful optimizations and avoids the overhead of moving data between system boundaries.

We ran the pipeline described in Listing 3 on an Amazon review dataset [13] with 82,836,502 reviews and 168,954,245 pairs of related products. Additionally, after finding groups of users with the same interests in step 3, we aggregated the result for each user to find the number of users with the same interests. To simulate running this pipeline without GraphFrames as a comparison point, we ran each stage separately using Spark, saving and loading the working data between stages. In addition to the I/O overhead, this prevented projections from being pushed down into the data scan, increasing the ETL time. Figure 2c shows this comparison.

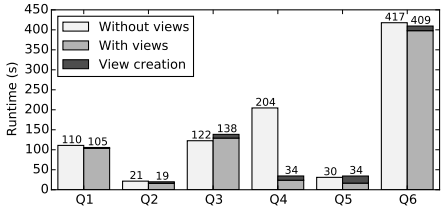
5. RELATED WORK

To our knowledge, GraphFrames is the first system that lets users combine graph algorithms, pattern matching and relational queries in a single API, and optimizes computations across them. GraphFrames builds on previous work in graph analytics using relational databases, query optimization for pattern matching, and declarative APIs for data analytics.

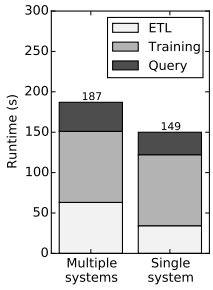
Graph Databases Graph databases such as Neo4j [17] and Titan [16] focus on mostly on graph queries, often using pattern matching languages like Cypher [17] and Gremlin [15]. They have very limited support for graph algorithms such as PageRank and for connecting with relational data outside the graph database. GraphFrames use the pattern matching abstraction from these systems, but can also support other parts of the graph processing workflow, such as building the graph itself out of relational queries on multiple tables, and running analytics algorithms in addition to pattern matching. GraphFrames then optimizes query execution across this entire workflow. GraphFrames' language-integrated API also makes



(a) Pattern queries [8]



(b) Performance of pattern queries with and without views



(c) End-to-end pipeline performance: multiple systems vs. single system

it easy to call user-defined functions (e.g., ETL code) and, in our Spark-based implementation, to call into Spark's built-in libraries, giving users a single environment in which to write end-to-end workflows.

Graph-Parallel Programming Standalone systems including Pregel and GraphLab [12, 11] have been designed to run graph algorithms, but they require separate data export and import and thus make end-to-end workflows complex to build. GraphFrames use similar parallel execution plans to many of these systems (e.g., the Gather-Apply-Scatter pattern) while supporting broader workflows.

Graph Processing over RDBMS GraphX and Vertexica [7, 9] have explored running graph algorithms on relational databases or dataflow engines. Of these, GraphX materializes a triplets view of the graph to speed up the most common join in iterative graph algorithms, while the others use the raw tables in the underlying database. GraphFrames generalize the execution strategy in these systems by letting the user materialize multiple views of arbitrary patterns, which can greatly speed up common types of queries. GraphFrames also provide a much broader API, including pattern matching and relational queries, where these tasks can all be combined, whereas previous systems only focused on graph algorithms.

6. DISCUSSION AND CONCLUSION

Graph analytics applications typically require relational processing, pattern matching and iterative graph algorithms. However, these applications previously had to be implemented in multiple systems, adding both overhead and complexity. In this paper, we aim to unify the three with the GraphFrames abstraction. The GraphFrames API which is concise and declarative, based on the "data frame" concept in R, and enables easy expression and mixing of these three paradigms. GraphFrames optimize the entire computation using graph-aware join optimization and view selection algorithm that generalizes the execution strategies in previous graph-on-RDBMS systems. GraphFrames are implemented over Spark SQL, enabling parallel execution on Spark and easy integration with Spark's external data sources, built-in libraries, and custom ETL code. We showed that GraphFrames make it easy to write complete graph processing pipelines and enable optimizations across them that are not possible in current systems.

We have open sourced our system to allow other researchers to build upon it [1].

Our current optimization algorithm produces a tree of pairwise join operators. As part of future work, we would like to support other options, such as one-shot join algorithms over multiple tables [2] and worse-case optimal join algorithms [6]. It should be possible to integrate these algorithms into our System-R based framework.

Additionally, GraphFrames currently do not provide support for processing dynamic graphs. In the future, we would like to develop efficient incremental graph update and processing support. We plan to leverage the newly available IndexedRDD [3] project to do this over Spark, or a relational database engine as an alternative backend. One interesting addition here will be deciding which graph views we wish to maintain incrementally as the graph changes.

7. REFERENCES

[1] GraphFrames: DataFrame-based graphs. https://github.com/graphframes/graphframes, Apr. 2016.

[2] Afrati, F. N., et al. GYM: a multiround join algorithm in MapReduce. *CoRR abs/1410.4156* (2014).

[3] Apache Spark. Spark IndexedRDD: An efficient updatable key-value store for Apache Spark. https://github.com/amplab/spark-indexedrdd, 2015.

[4] Armbrust, M., et al. Spark SQL: relational data processing in Spark. In *SIGMOD* (2015).

[5] Chirkova, R., et al. A formal perspective on the view selection problem. *VLDB 2002*.

[6] Chu, S., et al. From theory to practice: Efficient join query evaluation in a parallel database system. In *SIGMOD* (2015).

[7] Gonzalez, J., et al. GraphX: Graph processing in a distributed dataflow framework. In *OSDI* (2014).

[8] Huang, J., et al. Query optimization of distributed pattern matching. In *ICDE* (2014).

[9] Jindal, A., et al. Vertexica: Your relational friend for graph analytics! *VLDB* (2014).

[10] Leskovec, J., et al. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[11] Low, Y., et al. GraphLab: A new framework for parallel machine learning. In *UAI* (2010).

[12] Malewicz, G., et al. Pregel: A system for large-scale graph processing. In *SIGMOD* (2010).

[13] McAuley, J., et al. Inferring networks of substitutable and complementary products. In *KDD* (2015).

[14] Olston, C., Reed, B., Srivastava, U., Kumar, R., and Tomkins, A. Pig Latin: A not-so-foreign language for data processing. In *SIGMOD* (2008).

[15] Rodriguez, M. A. The Gremlin graph traversal machine and language. *CoRR abs/1508.03843* (2015).

[16] Titan distributed graph database. http://thinkaurelius.github.io/titan/.

[17] Webber, J. A programmatic introduction to Neo4j. In *SPLASH* (2012).

Algorithm 1: `FindPlanWithViews`

Input : query Q; graph G; views GV_1, .., GV_n; partitioning P
Output : Solutions for running Q

```
if Q.sol != null then
    return null; // already generated plans for Q
if Q has only one edge e = (v1, v2)) then
    Q.sol = ("match e", scan cost of E_i, P(e_i), ce_i);
    Q.sol = Q.sol ∪ MatchViews(Q.sol, views);
    return;
if all edges in Q are co-partitioned w.r.t P then
    Q.sol = ("co-l join of Q",co-l join cost i, P(Q), ce_i);
    Q.sol = Q.sol ∪ MatchViews(Q.sol, views);
    return;
T = φ ;
LD = LinearDecomposition(Q) ;
foreach linear decomposition (q1,q2) in LD do
    FindPlan(q1);
    FindPlan(q2 );
    linearPlans = GenerateLinearPlans(q1,q2);
    T = T ∪ linearPlans;
    T = T ∪ MatchViews(linearPlans, views);
LDAGs = GenerateLayeredDAGs(Q) ;
foreach layered DAG d in LDAGs do
    (q1, q2, ..., q_{n-1}, q_n) = BushyDecomposition(d) ;
    for i from 1 to n do
        FindPlan(q_i );
    bushyPlans = GenerateBushyPlans(q1, ..., q_n);
    T = T ∪ bushyPlans;
    T = T ∪ MatchViews(bushyPlans, views);
Q.sol = EliminateNonMinCosts(T);
```

Algorithm 2: `MatchViews`

Input : query plan solution set $\mathbb{S}$; graph views GV_1, .., GV_n
Output : query plan solution set from matching views

```
V = φ ;
foreach Solution S in 𝕊 do
    foreach Graph View GV_i do
        queryFragment = S.plan.query;
        viewQuery = GV_i.query;
        if viewQuery is equivalent to queryFragment then
            V = V ∪ ("scan", scan cost of GV_i, GV_i.p, CE_i);
return V;
```

APPENDIX

A. QUERY OPTIMIZATION

GraphFrame operators, including both the graph as well as the relational operators, are compiled to relational operators. Thereafter, we optimize the complete pipeline by extending Catalyst. To do this, the GraphFrame query planner extends the dynamic programming algorithm of Huang et al. [8] to the distributed setting and adds the view rewrite capability. The user can register arbitrary materialized views and the planner will automatically rewrite the query to reuse a materialized view when appropriate. This is useful because pattern queries could be very expensive to run and reusing computations across several queries can improve the user experience. GraphFrame API also allows users to get suggestions for the views to create. We also describe how we can further extend the query planning to create the views adaptively.

Additionally, by building on top of Spark SQL, GraphFrames also benefit from whole-stage code generation.

A.1 Query Planner

The dynamic programming algorithm proposed in [8] recursively decomposes a pattern query into fragments, the smallest fragment being a set of co-partitioned edges, and builds a query plan in a bottom-up fashion. The original algorithm considers a single input graph. In this paper, we extend it to views, i.e., the algorithm matches the views in addition to matching the pattern query. The input to the algorithm is the base graph G, a set of graph views $\{GV_1, GV_2, .., GV_n\}$, and the pattern query $Q = (V_q, E_q)$. Each graph view GV_i consists of the view query that was used to create the view and a cost estimator CE_i. The algorithm also takes the partitioning function P as an input, as opposed to a fixed partitioning in the original algorithm. The output is the best query plan (lowest cost) to process the query Q.

The algorithm starts by recursively decomposing the pattern query into smaller fragments and building the query plan for each fragment. At the leaf level, i.e., when the fragment consists of only a single edge, we lookup the edge (along with its associated predicates) in the base graph. At higher levels, we combine the child query plans to produce larger plans. At each level, we also check whether the query fragment matches with the view query of any of the graph views. In case a match is found, we add the view as a candidate solution to the query fragment. This also takes care of combining child query plans from multiple graph views, i.e., we consider all combinations of the graph views and later pick the best one. Algorithm 1 shows the extended algorithm for finding plans using views. Each time a new plan is generated for a query fragment, we match the fragment with the set of graph views, as shown in blue in Algorithm 1. Algorithm 2 shows the pseudocode for view matching. For every query plan solution, we check whether its query fragment is equivalent to a view query[3] and add the view to the solution set in case a match is found. Note that we keep both the solutions, one which uses the view and one which does not, and later pick the best one. Also note that we match the views on the logical query fragments in a bottom-up fashion, i.e., a view matched a lower levels could still be replaced by a larger view (and thus more useful view) at the higher levels.

Combining graph views, however, produces a new (intermediate) graph view and so we need to consider the new (combined) cost estimate when combining it further. To handle this, we keep track of four things in the query plan solution at each level: (i) the query plan, (ii) the estimated cost, (iii) the partitioning, and (iv) the cost estimator. When combining the solutions, we combine their cost estimates as well[4].

Figure 2 illustrates the query planning using views. The system takes the given pattern query and the three graph views, V_1, V_2, and V_3 as inputs. The linear decomposition (recursively) splits the query into two fragments, such that at least one of them is not decomposable (i.e., it is either a single edge or co-partitioned set of edges). The lower left part of Figure 2 shows one such linear decomposition and the corresponding candidate query plan using views. Here we match a view with a query fragment only when it contains the exact same set of edges. The bushy decomposition generates query fragments none of which may be non-decomposable (i.e., each query fragment could be further split into linear or bushy decompositions). The lower right part of Figure 2 shows one such bushy decomposition. We can see that the corresponding candidate query plan is different and could not have been generated by the linear decomposition alone.

[3]Instead of looking for exact match, the algorithm could also be extended to match views which *contain* the query fragment, as in traditional view matching literature.

[4]We can do this more efficiently by pre-computing the estimates for all combinations of the graph views.

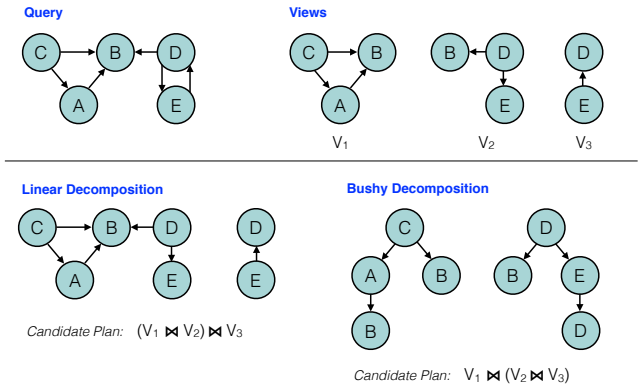


Figure 2: View Matching during Linear and Bushy Decompositions.

S.No.	View
1	$A \leftarrow B, A \leftarrow C, B \leftarrow F$
2	$A \leftarrow B, A \leftarrow C, C \leftarrow E$
3	$A \leftarrow B, A \leftarrow C, D \leftarrow B$
4	$A \leftarrow B, A \leftarrow C, E \leftarrow C$
5	$B \leftarrow A, B \leftarrow D, E \leftarrow B$

Table 2: Top-5 views of size three suggested by the system for the workload in [8]

Our extend view matching algorithm can still leverage all of the optimizations proposed by Huang et al. [8], including considering co-partitioned edges as the smallest fragment since they can be computed locally, performing both linear and bushy decomposition of the queries, and applying cycle-based optimization.

A.2 View Selection via Query Planning

The assumption so far was that the user manually creates the views. The system then generates query plans to process pattern queries using one or more of them. However, in some cases, the user may not be able to select which views to create. The question therefore is whether the system can suggest users the views to create in such scenarios. Notice that the nice thing about recursive query planning is that we are anyways traversing the space of all possible query fragments that are relevant for the given query. We can consider each of these query fragments as candidate views. This means that every time we generate a query plan for a fragment, we add it to a global set of candidate views.

In the end, we can rank the candidate views using different utility functions and present the top ones. One such function could be the ratio of *cost*, i.e., savings due to the view, and *size*, i.e., the spending on the view. Recall that each query plan solution (the candidate view) contains its corresponding cost estimator as well, which could be used to compute these metrics. The system could also collect the candidate views over several queries before presenting the interesting ones to the user. We refer the readers to traditional view selection literature for more details [5].

The key thing to take away from here is that we can generate interesting candidate views as a side-effect of dynamic programming based query planning. This means that the user can start running his pattern queries on the input graph and later create one or more of the suggested views to improve the performance. To illustrate, we ran the view suggestion API for the six query workload from [8]. Table 2 shows the top-5 views of size three produced by the system.

A.3 Adaptive View Creation

We discussed how the system can help users to select views. However, the views are still created manually as an offline process. This is expensive and often the utility of a view is not known a-priori. Let us now see how we can leverage the query planning algorithm to adaptively create the graph views. The key idea is to start by materializing smaller query fragments and progressively combine them to create views of larger query fragments. To do this, we annotate each solution with the list of graph views it processes, i.e., solution s now have five pieces of information: $(plan, cost, partitioning, estimator, \{GV_i\})$. When combining the child query plans, we union the graph views from the children.

When the algorithm runs for the first time there is only a single input graph view which is the base graph itself. We look at all the leaf level query plans, and materializing the one(s) having the maximum utility, i.e., they are the most useful. In each subsequent runs, we consider materializing the query plans which combine existing views, i.e., we essentially *move* the view higher up in the query tree. We still consider materializing new leaf level plans from the base graph. Rather than combining the graph views greedily, a more fancy version can also keep counters on how many times each graph view is used. We can then combine the most frequent as well as most useful graph views.

The above adaptive view creation technique has two major advantages. First, it amortizes the cost of creating the view over several queries. This is because creating views at the lower levels involve fewer joins and hence it is cheaper. The system only spends more resources on creating a view in case it is used more often. Second, this approach starts from more general leaf level views, which could be used across a larger set of queries, and gradually specializes to larger views higher up in the query tree. This is useful in scenarios where a user starts from ad-hoc analysis and later converges to a specific query workload — something which is plausible in pattern matching queries.