The GraphX Graph Processing System

Daniel Crankshaw  Ankur Dave  Reynold S. Xin
Joseph E. Gonzalez  Michael J. Franklin  Ion Stoica

UC Berkeley AMPLab (crankshaw, ankurd, rxin, jegonzal, franklin, istoica)@cs.berkeley.edu

ABSTRACT
The growing scale and importance of graph data has driven the development of specialized graph computation engines capable of inferring complex recursive properties of graph-structured data. However, these systems are unable to express many of the inherently data-parallel stages in a typical graph-analytics pipeline. As a consequence, existing graph analytics pipelines resort to multiple stages of data-parallel and graph-parallel systems composed through external storage systems. Instead, the GraphX abstraction unifies graph-parallel and data-parallel computation in a single system and is capable of succinctly expressing the entire graph analytics pipeline. However, implemented naively with relational operators it is impractically slow.

We describe an implementation of the GraphX abstraction that encodes graphs as collections of edges and vertices along with simple auxiliary index structures, and represents graph computations as a sequence of relational joins and aggregations. We incorporate techniques such as incremental view maintenance and index scans in databases and adapt these techniques to exploit common characteristics of graph computation workloads.

We evaluate these techniques individually as well as our overall GraphX implementation and find that it achieves performance comparable to contemporary graph-parallel systems in graph computation while retaining the expressiveness of contemporary data-parallel systems.

1. INTRODUCTION
From social networks to language modeling, graphs capture the structure in data and play a central role in the recent advances in machine learning and data mining. The growing scale and importance of graph data has driven the development of numerous specialized systems for graph analytics (e.g., Pregel [13], PowerGraph [8], and others [3][8][16]). Each system presents a new restricted programming abstraction to compactly express iterative graph algorithms (e.g., PageRank and connected components). By leveraging the restricted abstraction in conjunction with the static graph structure, these systems are able to optimize the data layout and distribute the execution of complex iterative algorithms on graphs with tens of billions of vertices and edges.

By restricting the types of computation they express to iterative vertex-centric algorithms on a single static graph, these graph-parallel systems are able to achieve orders-of-magnitude performance gains over contemporary data-parallel systems such as Hadoop MapReduce. However, these same restrictions make it difficult to express many of the operations found in a typical graph analytics pipeline. These operations include constructing the graph from external sources, modifying the graph structure (e.g., collapsing groups of vertices), and expressing computation that spans multiple graphs (e.g., merging two graphs). For example, while the PowerGraph system can compactly express and execute algorithms like PageRank several orders of magnitude faster than contemporary data-parallel systems, it is not well suited for extracting graphs from a collection of databases, collapsing vertices within the same domain (i.e., constructing a domain graph), or comparing the PageRank across several web graphs. Fundamentally, operations that move information outside of the graph topology or require a more global view are not well suited for graph-parallel systems.

In contrast, data-parallel systems like MapReduce [9] and Spark [19] are well suited for these tasks as they place few constraints on data movement and operate at a global view. By exploiting data-parallelism, these systems are highly scalable and can enable interactive data processing. However, directly implementing iterative graph algorithms in these data-parallel abstractions can be challenging and typically leads to complex joins and excessive data movement due to the failure to exploit the graph structure or take advantage of any of the recent developments [3][4][8] in distributed graph partitioning and representation.

As a consequence, existing graph analytics pipelines (e.g., GraphBuilder [9]) resort to composing graph-parallel graph analytics and data-parallel systems for graph loading through external storage systems such as HDFS. The resulting APIs are tailored to specific tasks and do not enable users to easily and efficiently compose graph-parallel and data-parallel operations on their data.

To address these challenges we previously introduced GraphX, a distributed graph computation framework which unifies graph-parallel and data-parallel computation in a single system. GraphX presents a unified abstraction which allows the same data to be viewed both as a graph and as tables without data movement or duplication. In addition to the standard data-parallel operators (e.g., map, reduce, filter, join, etc.), GraphX introduces a small set of graph-parallel operators including subgraph and mrTriplets, which transform graphs through a highly parallel edge-centric API. These operators are expressive enough to implement the Pregel and PowerGraph abstractions but also simple enough to be cast in relational algebra.

However, we demonstrate that a system implementing the GraphX abstraction using standard relational operators would be too slow compared to specialized graph frameworks. Instead, GraphX must be implemented as a system with graph-specific optimizations. In this paper we describe our implementation and optimization of GraphX, with the goal of achieving competitive performance versus specialized graph frameworks while retaining the generality and expressivity of the GraphX abstraction.

We evaluate each of our optimizations using microbenchmarks to demonstrate their utility and applicability. We also evaluate our
implementation of GraphX as a whole compared to other graph frameworks. We find that, after a variety of graph-specific optimizations, GraphX is performance-competitive with these frameworks and even outperforms them in some cases.

As background, in Section 2 we describe the existing graph computation frameworks and the characteristics of common algorithms implemented on them. Next, in Section 3 we describe the GraphX abstraction. In Section 4 we detail our implementation of this abstraction as a system on top of the Spark cluster computing framework. In Sections 5 and 6 we describe the optimizations we have made to this system and evaluate these optimizations using microbenchmarks. In Section 7 we evaluate the GraphX implementation as an end-to-end system. We discuss related work in Section 8. Finally, we discuss the system in Section 9 and conclude.

2. GRAPH PROCESSING SYSTEMS

In contrast to general data processing systems (e.g., MapReduce, Dryad, and Spark) which compose data-parallel operators to transform collections and are capable of expressing a wide range of computation, graph processing systems apply vertex-centric logic to transform graph data and exploit the graph structure to achieve more efficient distributed execution. In this section we introduce the key ideas behind graph-parallel systems and how they enable substantial performance gains. We then describe how the same restrictions that enable substantial performance gains limit the applicability of these systems to many important tasks in graph analytics.

2.1 Graph-Parallel Computation

The recursive nature of graph data (e.g., my interests are a function of my profile and the interests of my friends) necessitates the ability to calculate recursive properties on a graph separate from their structure. Algorithms ranging from PageRank and connected components to label propagation and collaborative filtering recursively define transformations on vertex and edge properties in terms of functions on the properties of adjacent vertices and edges. For example, the PageRank of each vertex may be computed by iteratively recomputing the PageRank of each vertex as a function of the PageRank of its neighboring vertices. The corresponding algorithms iteratively propagate information along the graph structure by transforming intermediate vertex and edge properties and solving for the fixed-point assignments. This common pattern of iterative local updates forms the basis of graph-parallel computation.

Graph-parallel computation is the analogue of data-parallel computation applied to graph data (i.e., property graphs). Just as data-parallel computation adopts a record-centric view of collections, graph-parallel computation adopts a vertex-centric view of graphs. In contrast to data-parallel computation which derives parallelism by processing independent data on separate resources, graph-parallel computation derives parallelism by partitioning the graph (dependent) data across processing resources and then resolving dependencies (along edges) through iterative computation and communication. More precisely, graph-parallel computation recursively defines the transformations of properties in terms of functions on neighboring properties and achieves parallelism by executing those transformations in parallel.

2.2 Graph-Parallel Systems

The increasing scale and importance of graph-structured data has led to the emergence of a range of graph-parallel systems [11, 13, 10, 8, 5, 15]. Each system is built around a variation of the graph-parallel abstraction [8], which consists of a property graph and a vertex-program that runs on each vertex in the graph and can interact with adjacent vertex-programs through messages (e.g., Pregel [13]) or shared state (e.g., GraphLab [10] and PowerGraph [8]). Each instantiation of the vertex-program can read and modify its vertex property as well as the properties on adjacent edges and in some cases even the properties on adjacent vertices. Most systems (e.g., [11, 13, 8]) adopt the bulk synchronous execution model, in which all vertex-programs run concurrently in a sequence of super-steps operating on the adjacent vertex-program state or on messages from the previous super-step. GraphX adopts this model as well because it ensures deterministic execution, simplifies debugging, and enables fault tolerance.

More recently, Gonzalez et al. [8] observed that many vertex-programs factor along edges both when receiving messages and when computing messages to neighboring vertices. As a consequence they proposed the gather-apply-scatter (GAS) decomposition that breaks the vertex-program into purely edge-parallel and vertex-parallel stages, eliminating the ability to directly iterate over the neighborhood of a vertex. The commutative associative gather function is responsible for accumulating the inbound messages, the apply function operates only on the vertex, and the scatter function computes the message for each edge and can be safely executed in parallel. The GAS decomposition enables vertices to be split across machines, increasing parallelism and addressing the challenge of the high-degree vertices common to many real-world graphs. The GraphX system adopts this more edge-centric perspective, enabling high-degree vertices to be split across machines.

The static graph structure of the graph-parallel abstraction constrains data movement (communication) to the static topology of the graph, enabling the system to optimize the distributed execution. By leveraging advances in graph partitioning and representation, these systems are able to reduce communication and storage overhead. For example, [8] uses a range of vertex-based partitioning heuristics to efficiently split large power-law graphs across the cluster and vertex-replication and pre-aggregation to reduce communication. Given the result of the previous iteration, vertex-programs are independent and can be executed in any order, providing opportunities for better cache efficiency [15] and on-disk computation. As graph algorithms proceed, vertex-programs converge at different rates, leading to active sets (the collection of active vertex-programs) that shrink quickly. For example, when computing PageRank, vertices with no in-links will converge in the first iteration. Recent systems [13, 7, 10, 8] track active vertices and eliminate data movement and additional computation for vertices that have converged. We exploit many of these optimizations in implementing GraphX as well.

The same restrictions that enable graph-parallel systems to out-perform contemporary data-parallel systems when applied to graph computation also limit their applicability to many of the operations found in a typical graph analytics pipeline. For example, while graph-parallel systems can efficiently compute PageRank or label diffusion, they are not well suited to building graphs from multiple data sources, coarsening the graph (e.g., building a domain graph), or comparing properties across multiple graphs. More precisely, the narrow view of computation provided by the graph-parallel abstraction is unable to express operations that build and transform the graph structure or span multiple independent graphs. Instead, these operations require data movement beyond the topology of the graph and a view of computation at the level of graphs rather than individual vertices and edges.

To address the lack of support for these essential operations, existing graph-parallel systems either rely on additional graph ETL support tools (e.g., GraphBuilder [9]) or have special internal functions for specific ETL tasks (e.g., parsing a text file into a property graph). These solutions are limited in the range of operations they support and use external storage systems for sharing data across framework
boundaries, incurring extensive data copying and movement. Finally, these systems do not address the challenge of computation that spans multiple graphs. The GraphX abstraction aims to solve these problems, and in implementing it we are concerned with the performance of both graph algorithms and graph pipelines.

2.3 Graph Algorithms

We benchmark on two ubiquitous graph algorithms, PageRank and connected components, which can both be expressed using the previously-mentioned graph computation frameworks. This section describes the properties of these algorithms.

PageRank is a simple algorithm that computes a ranking of the importance of each vertex in a graph. The basic formula for the PageRank of a vertex is

\[ PR'(v) = (1 - C) + C \sum_{u \in N_v} \frac{PR(u)}{d_u} \]  

where the sum is over all of vertices \( u \) that have an edge directed from \( v \) to \( v \), \( d_u \) is the outdegree of \( u \), and \( 1 - C \) is a reset probability, traditionally set to 0.15. To calculate the final PageRank for each vertex, one sets each vertex to an initial value and then repeatedly updates Eq. \( 1 \) until the ranks converge to within some tolerance.

Connected components is an even simpler algorithm. For each vertex \( v \), the connected components algorithm computes the smallest vertex ID \( u \) such that there is a path from \( v \) to \( u \). Connected components is also an iterative algorithm. Every vertex is initialized to be in its own component. Then, the component of each vertex is updated in every round to be the minimum component of its neighbors. Convergence is reached once there is a round when no vertices change components.

3. THE GraphX LOGICAL ABSTRACTION

This section serves as background for our contribution, the GraphX implementation, in Section 4. Our implementation in turn shaped the abstraction by guiding us towards more efficiently implementable primitives.

The GraphX abstraction unifies the data-parallel and graph-parallel computation through a data model that presents graphs and collections as first class objects and a set of primitive operators that enables their composition. By unifying graphs and collections as first class composable objects, the GraphX data model is capable of spanning the entire graph analytics pipeline. By presenting a set of data-parallel and graph-parallel operators that can be composed in any order, GraphX allows users to leverage the programming model best suited for the current task without having to sacrifice performance or flexibility of future operations. We now describe the its data model and operators and demonstrate their composability and expressiveness through example applications.

3.1 The GraphX Data Model

The GraphX data model consists of immutable collections and property graphs. The immutability constraint simplifies the abstraction and enables data reuse and fault tolerance. Collections in GraphX consist of unordered tuples (i.e., key-value pairs) and represent unstructured data. The key can be null and does not need to be unique, and the value can be an arbitrary object. The unordered collection view of data is essential for processing raw input, evaluating the results of graph computation, and certain graph transformations. For example, when loading data from a file we might begin with a collection of strings (with null keys) and then apply relational operators to obtain a collection of edge properties (keyed by edge), construct a graph and run PageRank, and finally view the resulting PageRank values (keyed by vertex identifier) as a collection for additional analytics.

The property graph \( G(P) = (V, E, P) \) combines structural information, \( V \) and \( E \), with properties \( P = (P_V, P_E) \) describing the vertices and edges. The vertex identifiers \( i \in V \) can be arbitrary, but the GraphX system currently uses 64-bit integers (without consecutive ordering constraints). These identifiers may be derived externally (e.g., user ids) or by applying a hash function to a vertex property (e.g., page URL). Logically the property graph combines the vertex and edge property collections consisting of key-value pairs \( (i, P_V(i)) \) and \( ((i, j), P_E(i, j)) \) respectively. We introduce the property graph as a first class object in the data model to enable graph specific optimizations which span the vertex and edge property collections and to present a more natural graph-oriented API.

3.2 The Operators

Computation in the GraphX abstraction is achieved by composing graph-parallel and data-parallel operators that take graphs and collections as input and produce new graphs and collections. In selecting the core set of operators we try to balance the desire for parsimony with the ability to exploit specific lower-level optimizations. As a consequence these operators form a narrow interface to the underlying system, enabling the GraphX abstraction to be expressive and yet feasible to implement and execute efficiently on a wide range of data-parallel systems. To simplify graph analytics, GraphX exposes a rich API of more complex graph operators (e.g., coarsening, neighborhood aggregation) and even other abstractions (e.g., Pregel) by composing the basic set of primitive operators.

The GraphX system exposes the standard data-parallel operators (Listing 2) found in contemporary data-flow systems. The unary operators filter, map, and reduceByKey each takes a single collection and produces a new collection with the records removed, transformed, or aggregated. The binary operator leftJoin performs a standard left outer equi-join by key. Both the map and filter operators are entirely data-parallel without requiring any data movement or communication. On the other hand, the reduceByKey and leftJoin operators may require substantial data movement depending on how the data is partitioned.

In Listing 2 we describe the set of graph-parallel operators that produce new graphs and collections. These operators join vertex and edge collections, apply transformations on the properties and structure, and move data along edges in the graph. In all cases, these operators express local transformations on the graph (i.e., UDFs have access to at most a single triplet at a time).
The `Graph` operator constructs a property graph from vertex and edge collections. In many applications the vertex collection may contain duplicate vertex properties or may not contain properties for vertices in the edge collection. For example when working with web data, web-links may point to missing pages or pages may have been crawled multiple times. By applying the `merge` UDF to duplicate vertices and substituting the default property for missing vertices, the `Graph` operator ensures that the resulting graph is consistent: without missing or duplicate vertices.

While the `Graph` operator produces a graph-oriented view of collections, the `vertices`, `edges`, and `triplets` produce collection-oriented views of a graph. The `vertices` and `edges` operators deconstruct the property graph into the corresponding vertex and edge collections. The collection views are used when computing aggregates, or when saving graphs to external data stores. The `triplets` operator is logically a three-way join to form a new collection consisting of key-value pairs of the form `(i, j, (Pv(i), Pv(j)))`. This essential graph operator can be concisely cast in terms of relational operators:

```
SELECT s.Id, t.Id, s.P, e.P, t.P
FROM edges AS e
JOIN vertices AS s, vertices AS t
ON e.srcId = s.Id AND e.dstId = t.Id
```

By joining properties along edges, the `triplets` operator enables a wide range of graph computation. For example, the composition of the `triplets` and data-parallel `filter` operators can be used to extract edges that span two domains or connect users with different interests. Furthermore, the `triplets` operator is used to construct the other graph-parallel operators (e.g., `subgraph` and `merge`).

The `mapV` and `mapE` operators transform the vertex and edge properties respectively and return the transformed graph. The `mapV` UDF provided to `mapV` and `mapE` can only return a new attribute value and cannot modify the structure (i.e., change the vertex identifiers for the vertex or edge). As a consequence, the resulting graph is guaranteed to be consistent and can reuse the underlying structural representation.

In many cases it is necessary to merge external vertex properties (e.g., merging user profile data with a social network) stored in a vertex property graph with an existing graph. This can be accomplished in GraphX using the `leftJoin` graph operator. `leftJoin` takes a collection of vertex properties and returns a new graph that incorporates the properties into all matching vertices in the graph. The `leftJoin` preserves the original graph structure and is logically equivalent to a left outer equi-join of the vertices with the input vertex property collection.

Comparing the results of graph computation (e.g., PageRank) on different slices (i.e., subgraphs) of a graph based on vertex and edge properties (e.g., time) often reveals trends in the data. To support this type of analysis we need to be able to efficiently construct subgraphs and compare properties and structural changes across subgraphs. The `subgraph` operator restricts the graph to the vertices and edges that satisfy the respective predicates. To ensure that the graph is consistent, all retained edges must satisfy both the source and target vertex predicate as well as the edge predicate.

The `mrTriplets` operator is logically the composition of the `triplets` graph-parallel operator with the `map` and `reduceByKey` data-parallel operators. More precisely, the `mrTriplets` operator applies the map UDF to each triplet in the output of the `triplets` operator. The map UDF optionally constructs “messages” (of arbitrary type) to send to the source and target vertices (or both). All messages destined for the same vertex are aggregated using the commutative associative reduce UDF and the resulting aggregates are returned as a collection keyed by the destination vertex. This can be expressed in the following SQL query:

```
SELECT t.dstId, r(m(t)) AS sum
FROM triplets AS t GROUPBY t.dstId
WHERE sum IS NOT null
```

The constraint that the map UDF only emits messages to the source or target vertex ensures that data movement remains along edges in the graph, preserving the graph dependency semantics. By expressing message computation as an `edge-parallel` map operation followed by a commutative associative aggregation, we eliminate the effect of high degree vertices on communication and parallel

```java
class Graph[V, E] {
  def Graph(v: Col[(Id, V)], e: Col[(Id, Id, E)]) {
    def vertices: Col[(Id, V)]
    def edges: Col[(Id, Id, E)]
    def triplets: Col[(Id, Id, (V, E, V))]
    def mapV(m: (Id, V) => V): Graph[V, E]
    def mapE(m: Triplet[(Id, Id, E)] => E): Graph[V, E]
    def leftJoin(t: Col[(Id, U)]: Graph[(V, U), E])
    def subgraph(vPred: (Id, V) => Boolean, ePred: Triplet[(Id, Id, E)] => Boolean): Graph[(V, E)]
    def mrTriplets(m: Triplet[(Id, Id, E)] => (M, M),
                  r: (M, M) => R,
                  activeSet: Col[(Id, V)] = None,
                  direction: Direction = None): Col[(Id, M)]
  }
}
```

Listing 2: `Graph` operators: The `mapE` operator takes a Graph over vertex and edge properties of type `V` and `E` and a `map` UDF from triplets to a new edge property and returns the graph with the new edge properties.

```
val graph: Graph[User, Double]
val mapF(t: Triplet[User, Double]): Int = {
  if (t.src.age > t.dst.age) t.srcId, 1
  else (t.src.age < t.dst.age) (t.srcId, 1)
  else None
}
val seniors = graph.mrTriplets(mapUDF, reduceUDF)
```

Figure 1: Example use of `mrTriplets`: The `mrTriplets` operator is used to compute the number of more senior neighbors of each vertex. Note that vertex `E` does not have a more senior neighbor and therefore does not appear in the collection.
def preg(el: Graph[V, E],
    vprog: (V, M) => V,
    sendMsg: Triplet[V, E] => M,
    gather: (M, M) => M):
    g = Graph[V, E] = {
        var live = g.vertices.count
        // Loop until convergence
        while (live > 0) {
            // Compute the messages
            val msgs = g.mrTriplets(sendMsg, gather, Out)
            // Receive the messages and run vertex program
            g = g.leftJoin(msgs).mapV(vprog)
            // Count the vertices that don’t want to halt
            live = g.vertices.filter(v => !v.halt).count
        }
        return g
    }

Listing 3: Enhanced Pregel: We implemented a version of Pregel built around the GAS decomposition to enable degree independence and at the same allow message computation to read the remote vertex attributes.

scalability. The mrTriplets operator is the primary building block of graph-parallel algorithms. For example, in Figure[1] we use the mrTriplets operator to compute the number of more similar neighbors for each user in a social network. In the next section we show how to compose these basic operators to express more complex tasks like graph coarsening and even implement existing graph-parallel abstractions.

When solving recursive properties on a graph, some vertices typically converge early in the computation and are excluded from the active vertex set. As a consequence, executing the mrTriplets function on all edges can be wasteful, especially when only a few vertices are active. While it is possible to implement such logic within message calculation, the system must still invoke the message calculation on all edges. Therefore, mrTriplets has an optional argument activeSet which by default is disabled. However, if an active set is provided with direction of Out, for example, then mrTriplets only runs on edges originating from vertices in the active set. Active vertex sets typically result in only a small fraction of the graph changing, specifically the region of the graph neighboring the active set. We exploit this fact to reduce communication in Section[5].

3.3 Composing Operators

The GraphX operators can express efficient versions of some of the most widely adopted graph-parallel abstractions. We have currently implemented enhanced versions of Pregel and the PowerGraph abstractions. In Listing[3] we construct an enhanced version of Pregel built around the more efficient GAS decomposition. The Pregel abstraction iteratively computes the messages on the active subgraph using the mrTriplets operator and then applies the mapV operator to execute the vertex program UDF. In this example we use change tracking option in mrTriplets to restrict execution to out-edges of vertices that changed in the previous round. In Section[4] we show that allowing mrTriplets to track changes enables several important system optimizations. Unlike the original formulation of Pregel, our version exposes both the source and target vertex properties during message calculation. In Section[5] we demonstrate how through UDF bytecode inspection in the mrTriplets operator we can eliminate extra data movement if only one of the source or target attribute is accessed when computing the message (e.g., PageRank).

In Listing[4] we used our version of Pregel to implement connected components. The connected components algorithm computes for each vertex its lowest reachable vertex id. We first initialize the vertex properties using the vMap operator and then define the three functions required to use the GAS version of Pregel. The sendMsg function leverages the triple view of the edge to only send a message to neighboring vertices when their component id is larger.

4. THE GraphX SYSTEM

The scalability and performance of GraphX is derived from the design decisions and optimizations made in the physical execution layer. The design of the physical representation and execution model is heavily influenced by three characteristics of graph computation. First, in Section[4] we demonstrated that graph computation can be modeled as a series of joins and aggregations. Maintaining the proper indexes can substantially speed up local join and aggregation performance. Second, as outlined in [8], we can minimize communication in real-world graphs by using vertex-cut partitioning, in which edges are partitioned evenly across a cluster and vertices are replicated to machines with adjacent edges. Finally, graph computations are typically iterative and therefore we can afford to construct indexes. Furthermore, as computation proceeds, the active set of vertices – those changing between iterations – often decreases.

In the remainder of this section, we introduce Apache Spark, the open source data-parallel engine on which GraphX was built. We then describe the physical representation of data and the execution strategies adopted by GraphX. Along the way, we quantify the effectiveness of each optimization technique. Readers are referred to Section[7] for details on datasets and experimental setup.

4.1 Apache Spark

GraphX is implemented on top of Spark [19], a widely used data-parallel engine. Similar to Hadoop MapReduce, a Spark cluster consists of a single driver node and multiple worker nodes. The driver node is responsible for task scheduling and dispatching, while the worker nodes are responsible for the actual computation and physical data storage. However, Spark also has several features that differentiate it from traditional MapReduce engines and are important to the design of GraphX.

In-Memory Caching: Spark provides the Resilient Distributed Dataset (RDD) in-memory storage abstraction. RDDs are collections of objects that are partitioned across a cluster. GraphX uses RDDs as the foundation for distributed collections and graphs.

Computation DAGs: In contrast to the two-stage MapReduce topology, Spark supports general computation DAGs by composing multiple data-parallel operators on RDDs, making it more suitable for expressing complex data flows. GraphX uses and extends Spark operators to achieve the unified programming abstraction.
Lineage-Based Fault Tolerance: RDDs and the data-parallel computations on RDDs are fault-tolerant. Spark can automatically reconstruct any data or execute tasks lost during failures.

Programmable Partitioning: RDDs can be co-partitioned and co-located. When joining two RDDs that are co-partitioned and co-located, GraphX can exploit this property to avoid any network communication.

Interactive Shell: Spark allows users to interactively execute Spark commands in a Scala or Python shell. We have extended the Spark shell to support interactive graph analytics.

4.2 Distributed Graph Representation

GraphX represents graphs internally using two Spark distributed collections (RDDs) – an edge collection and a vertex collection. By default, the edges are partitioned according to their configuration in the input collection (e.g., original placement on HDFS). However, they can be re-partitioned by their source and target vertex ids using a user-defined partition function. GraphX provides a range of built-in partitioning functions, including a 2D hash partitioner that provides an upper bound on communication for the mrTriplets operator that is \( O(n \sqrt{p}) \) for \( p \) partitions and \( n \) vertices. For efficient lookup of edges by their source and target vertices, the edges within a partition are clustered by source vertex id, and there is an unclustered index on target vertex id. The clustered index on source vertex id is a compressed sparse row (CSR) representation that maps a vertex id to the block of its out-edges. Section 6.2 discusses how these indexes are used to accelerate iterative computation.

The vertices are hash partitioned by their vertex ids, and on each partition, they are stored in a hash index (i.e., clustered by the hash index). Each vertex partition also contains a bitmask and routing table. The bitmask enables the set intersection and filtering required by the subgraph and join operators. Vertices hidden by the bitmask do not participate in the graph operations. The routing table contains the join sites for each vertex in the partition and is used when broadcasting vertices to construct triplets. The routing table is logically a map from a vertex id to the set of edge partitions that contain adjacent edges and is derived from the edge table by collecting the source and target vertex ids for each edge partitions and aggregating the result by vertex id. The routing table is stored as a compressed bitmap (i.e., for each edge partition, which vertices are present).

4.3 Graph Operator Execution Strategies

The GraphX abstraction consists of both data-parallel and graph-parallel operators. For the data-parallel operators we adopt the standard well-established execution strategies, using indexes when available. Therefore, in this section we focus on execution strategies for the graph-parallel operators outlined in Section 5.4.

The graph-parallel operators defined in Listing 2 are implemented by transforming the vertex and edge RDDs using the Spark API. The execution strategies for each operator are as follows:

**vertices, edges:** Directly extract the vertex and edge RDDs.

**mapV, mapE:** Transform the internal vertex and edge collections, preserving the indexes.

**leftJoin:** Co-partition the input with the vertex attributes, join the vertex attributes with the co-partitioned input using the internal indexes, and produce a new set of vertex attributes. As a consequence only the input is shuffled across the network.

**triplets:** Logically requires a multiway distributed join between the vertex and edge RDDs. However using the routing map, we move the execution site of the multiway join to the edges, allowing the system to shuffle only the vertex data and avoid moving the edges, which are often orders of magnitude larger than the vertices. The triplets are assembled at the edges by placing the vertices in a local hash map and then scanning the edge table.

**subgraph:** (1) Generate the graph’s triplets, (2) filter the triplets using the conjunction of the edge triplet predicate and the vertex predicate on both source and target vertices to produce a restricted edge set, and (3) filter the vertices using the vertex predicate. To avoid allocation and provide fast joins between the subgraph and the original graph, the vertex filter is performed using the bitmask in the internal vertex collection, as described in Section 6.2.

**innerJoin:** (1) Perform an inner join between the input and the internal vertex collection to produce the new vertex properties, and (2) ensure consistency by joining the ids in the input collection with the internal edge collection and dropping invalidated edges.

The distributed join in step 2 is only performed separately when the user requests the edges of the result. It is redundant for operations on the triplet view of the graph, such as triplets and mrTriplets, because the joins in these operations implicitly filter out edges with no corresponding vertex attributes.

Vertices eliminated by the inner join in step 1 can be removed using the bitmask in a similar fashion as for subgraph, enabling fast joins between the resulting vertex set and the original graph. We exploit this in our Enhanced Pregel implementation, as described in Section 5.4.

**mrTriplets:** Apply the map UDF to each triplet and aggregate the resulting messages by target vertex id using the reduce UDF. The activeSet argument is implemented by joining the active set with the triplet view and running the map UDF only on triplets adjacent to active vertices. Vertices not adjacent to an active vertex do not change as a result of the mrTriplets call, which we exploit to reduce communication in 5.4. We implement the filtered edge scan efficiently in 6.2.

5. LOGICAL OPTIMIZATIONS

The logical query plan for the mrTriplets operator consists of a three-way join to bring the source vertex attributes and the target vertex attributes to the edges and generate the triplets view of the graph, followed by an aggregation step to apply the map and reduce UDFs. We use the routing table to ship vertices to edges and set the
edge partition as the join sites, which is equivalent to the idea of vertex-cut partitioning in PowerGraph. In addition, we describe two techniques we have developed that further minimize the communication in the join step. The first applies the concept of incremental view maintenance to communicate only vertices that change values after a graph operation, and the second uses bytecode analysis to automatically rewrite the physical join plan. These techniques enable GraphX to present a simple logical view of triplets with the capability to optimize the communication patterns in the physical execution plan.

5.1 Incremental View Maintenance

We observe that the number of vertices that change in iterative graph computations usually decreases as the computation converges to a fixed-point, particularly if the application has an explicit notion of active vertex sets. This presents an opportunity to further optimize the join in mrTriplets using techniques in incremental view maintenance. Recall that in order to compute the join, GraphX uses the routing table to route vertices to the appropriate join sites in the internal edge RDD. After each graph operation, we update a bit mask to track which vertex properties have changed. When GraphX needs to ship the vertices to materialize (in-memory) the replicated vertex view, it creates the view by shipping only vertices that have changed, and uses values from the previously materialized replicated vertex view for vertices that have not changed. Figure 3 illustrates the impact of incrementally maintaining the replicated vertex view in both PageRank and connected components on the Twitter graph.

It is important to note that incrementally maintaining the replicated vertex view is independent of the concept of active vertex sets. While incremental view maintenance benefits from active sets, it lowers communication even when the application does not have active sets, as some vertex attributes still may not change.

5.2 Automatic Join Elimination

The map UDF in the mrTriplets operator may only access one of the vertices, or none at all, in many algorithms. For example, when mrTriplets is used to count the degree of each vertex, the map UDF does not access any vertex attributes. In the case of PageRank, only the source vertex attributes are referenced.

GraphX implements a JVM bytecode analyzer that inspects the bytecode of the map UDF at runtime for a given mrTriplets query plan and determines whether the source or target vertex attributes are referenced. If only the source attributes are referenced, GraphX automatically rewrites the query plan from a three-way join to a two-way join. If none of the vertex attributes are referenced, GraphX eliminates the join entirely. Figure 4 demonstrates the impact of this physical execution plan rewrite on communication and runtime.

6. PHYSICAL OPTIMIZATIONS

6.1 Structural Index Reuse

Because the collections and graphs are immutable they can share the structural indexes associated within each vertex and edge partition to both reduce memory overhead and accelerate local graph operations. For example, within a vertex partition, we can use the hash index to perform fast aggregations and the resulting aggregates would share the same index as the vertices. This shared index enables very efficient joining of the original vertices and the aggregates by converting the join into coordinated sequential scans (similar to a merge join). In our benchmarks, index reuse brings down the runtime of PageRank on the Twitter graph from 27 seconds per iteration to 16 seconds per iteration. Index reuse has the added benefit of reducing memory allocation, because the indexes are reused in memory from one collection and graph to the next, and only the properties are changed.

Most of the GraphX operators preserve the structural indexes to maximize index reuse. Operators that do not modify the graph structure (e.g., mapV, mapE, leftJoin, and mrTriplets) directly preserve the indexes. To reuse indexes for operations that restrict the graph structure (e.g., subgraph and innerJoin), GraphX relies on the bitmask to construct the restricted graph view. Some of the collections operations (e.g., g.vertices.map) enable more general transformations (e.g., renumbering) that destroy the index but have more restrictive analogues that preserve the index (e.g., g.mapV). Finally, in some cases extensive index reuse could lead to decreased efficiency, such as for graphs that are highly filtered. GraphX therefore provides a reindex operator for graphs which rebuilds the index over the visible vertices.

6.2 Sequential Scan vs Index Scan

Figure 3: Impact of incrementally maintaining the replicated vertex view: For both PageRank and connected components, as vertices converge, communication decreases due to incremental view maintenance. We suspect the initial steep rise in communication is due to compression; many early rank update messages are the same and can be run-length encoded.

Figure 4: Impact of automatic join elimination on communication and runtime: We ran PageRank for 20 iterations on the Twitter dataset with join elimination turned on and off. We observe that automatic join elimination reduces the amount of communication by almost half and substantially decreases the total execution time as well.
Recall that in most operators, GraphX uses the structural indexes and relies on bitmasks to track whether a particular vertex is still visible. While this reduces the cost of computing index structures in iterative computations, it also prevents the physical data set from shrinking in size. For example, in the last iteration of connected components on the Twitter graph, only a few of the vertices are still active. However, to execute the mrTriplets on the triplet view we still need to sequentially scan 1.5 billion edges and verify for each edge whether its vertices are still visible using the bitmask, even if the application provides the active set to GraphX.

To mitigate this problem, we implement an index scan access method on the bitmask and switch from sequential scan on edges to bitmap index scan on vertices when the fraction of active vertices is less than 0.8. This bitmap index scan on vertices exploits the property that edges are clustered by their source vertex id to efficiently join vertices and edges together. Figure 5 illustrates the performance of sequential scan versus index scan in both PageRank and connected components.

### 6.3 Additional Engineering Techniques

While implementing GraphX, we discovered that a number of low level engineering details had significant performance impact. We sketch some of them here.

#### Memory-based Shuffle

GraphX relies on Spark’s shuffle mechanism for join and aggregation communication. Spark’s default implementation materializes the shuffle data to disk, hoping that it will remain in the OS buffer cache when the data is fetched by remote nodes. In practice, we have found that the extra system calls and file system journaling adds significant overhead, and the inability to control when buffer caches are flushed leads to variability in communication-intensive workloads like graph algorithms. We modified the shuffle phase to materialize map outputs in memory and remove this temporary data using a timer.

#### Batching and Columnar Structure

In our join code path, rather than shuffling the vertices one by one, we batch a block of vertices routed to the same target join site and convert the block from row-oriented format to column-oriented format. We then apply the LZF compression algorithm on these blocks to send them. Batching has a negligible impact on CPU time while improving the compression ratio of LZF by 10–40% in our benchmarks.

#### Variable Integer Encoding

Though we use 64-bit integer identifiers for vertices and edges, in most cases the ids are much smaller than $2^{64}$. To exploit this fact, during shuffling, we encode integers using a variable-encoding scheme where for each byte, we use only the first 7 bits to encode the value, and use the highest order bit to indicate whether we need another byte to encode the value. In this case, smaller integers are encoded with fewer bytes. In the worst case, integers greater than $2^{50}$ require 5 bytes to encode. This technique reduces our communication in PageRank by 20%.

### 7. SYSTEM EVALUATION

We evaluate the performance of GraphX on specific graph-parallel computation tasks as well as end-to-end graph analytic pipelines, comparing to the following systems:

1. Apache Spark 0.8.1: the data-parallel cluster compute engine GraphX builds on. We use Spark to demonstrate the performance of graph algorithms implemented naively on data-parallel systems. We chose Spark over Hadoop MapReduce because of Spark’s support for distributed joins and its reported superior performance.

2. Apache Giraph 1.0: an open source implementation of Google’s Pregel. It is a popular graph computation engine in the Hadoop ecosystem initially open-sourced by Yahoo!

3. GraphLab 2.2 (PowerGraph): another open source graph computation engine commonly believed to be one of the fastest available. Note that GraphLab is implemented in C++, while both other systems and GraphX run on the JVM. It is expected that even if all four systems implement identical optimizations, GraphLab would have an “unfair” advantage due to its native runtime.

For graph-parallel algorithms, we demonstrate that GraphX is more than an order of magnitude faster than idiomatic Spark and performs comparably to the specialized systems, while outperforming them in end-to-end pipelines.

All experiments were conducted on Amazon EC2 using 16 m2.4xlarge workers and 64-bit Linux 3.2.28. We plot the mean and standard deviation for 10 trials of each experiment.

### 7.1 Graph-Parallel Performance
We evaluated the performance of GraphX on PageRank and Connected Components, two well-understood graph algorithms that are simple enough to serve as an effective measure of the system’s performance rather than the performance of the user-defined functions.

For each system, we ran both algorithms on the Twitter and LiveJournal social network graphs (see Table 1). We used the implementations of these algorithms included in the Giraph and PowerGraph distributions, and we additionally implemented PageRank using idiomatic Spark dataflow operators.

Figures 6 and 7 show the total runtimes for the connected components algorithm running until convergence. On the Twitter graph, Giraph outperforms GraphX and is as fast as GraphLab despite the latter’s highly optimized C++ implementation. We conjecture that this is due to the difference in partitioning strategies: GraphLab and GraphX use vertex cuts while Giraph uses edge cuts. Vertex cuts split high-degree vertices across partitions, but incur some overhead due to the joins and aggregation needed to coordinate vertex properties across partitions containing adjacent edges. The connected components algorithm does very little communication per iteration (see Figure 5), negating the benefit of vertex cuts but still incurring the overhead. In the case of LiveJournal, Giraph is slower because it uses Hadoop MapReduce for resource scheduling and the overhead of that (approximately 10 seconds) is quite substantial when the graph is small.

Figures 6 and 7 show the total runtimes for PageRank for 20 iterations on each system, including the idiomatic Spark dataflow implementation of PageRank. PageRank on GraphX is much faster than PageRank on Spark, and since GraphX is built on Spark, the difference can be isolated to the fact that GraphX exploits the graph structure using vertex cuts, structural indices, and the other optimizations described in Section 2. The specialized systems also outperform the Spark dataflow implementation for similar reasons.

In Figure 7, we plot the strong scaling performance of GraphX running PageRank on the Twitter follower graph. As we move from 8 to 32 machines (a factor of 4) we see a 3x speedup. However as we move to 64 machines (a factor of 8) we only see a 3.5x speedup. While this is hardly linear scaling, it is actually slightly better than the 3.2x speedup reported by PowerGraph (1). The poor scaling performance of PageRank has been attributed by (1) to high communication overhead relative to computation.

The fact that GraphX is able to scale slightly better than PowerGraph is relatively surprising given that the Spark shared-nothing worker model eliminates the potential for shared memory parallelism and forces the graph to be partitioned across processors and not machines. However, Figure 5 shows the communication of GraphX as a function of the number of partitions. Going from 16 to 128 partitions (a factor of 8) yields only around a 2-fold increase in communication. Returning to the analysis conducted by (1), we find that the vertex-cut partitioning adopted by GraphX mitigates the 8-fold increase in communication due to Spark.

7.2 End-to-End Pipeline Performance

Specialized graph-parallel systems are much faster than data-parallel systems such as Hadoop MapReduce and Apache Spark for iterative graph algorithms, but they are not well suited for many of the operations found in a typical graph analytics pipeline. To illustrate the unification of graph-parallel and data-parallel analytics in GraphX, we evaluate the end-to-end performance of each system in performing a multi-step pipeline that determines the 20 most important articles in the English Wikipedia by PageRank.

This analytics pipeline contains three stages: (1) parsing an XML file containing a snapshot of all English Wikipedia articles and extracting the link graph, (2) computing PageRank on the link graph, and (3) joining the 20 highest-ranked articles with their full text. Existing graph processing systems focus only on stage 2, and we demonstrate that GraphX’s unified approach provides as good or better end-to-end performance than specialized graph-parallel systems even for simple pipelines.

Because Giraph and GraphLab do not support general data-parallel operations such as XML parsing, joins, or top-K, we implemented these operations in their pipelines by transferring data to and from a data-parallel system using files. We used Spark and HDFS for this purpose. The GraphX unified model was capable of expressing the entire pipeline.

Figure 7 shows the performance of each system’s pipeline. Despite GraphLab’s superior performance on the graph-parallel portion of the pipeline, GraphX outperforms it in end-to-end runtime by avoiding the overhead of serialization, replication, and disk I/O at the stage boundaries. The GraphX pipeline was also simpler and easier to write due to the unified programming model.

8. RELATED WORK

This section focuses on related work in databases, graph-parallel engines (also discussed in Section 2), and data-parallel systems.

Query optimizers in databases use many of the same techniques as we have described (e.g., join rewrite and incremental view maintenance). In particular, we borrow many ideas from distributed databases, particularly R* (12), such as distributed join sites. However, without specialized index structures and join optimizations customized for graphs, particularly an awareness of active vertex sets, databases would not be able to achieve performance competitive with modern graph-parallel systems.

Specialized graph-parallel systems also share many similarities with GraphX. Most of these systems (e.g., Giraph) use edge cuts, and computation occurs by giving each vertex a fixed location, denormalizing edges and storing them with their source and destination vertices, and passing messages from vertex to vertex along edges. Thus a neighborhood-local computation requires all-to-all communication for high-degree vertices. This can be still be viewed as a vertex-edge-vertex join in which the edges move by being replicated to all neighboring vertices rather than the vertices moving. How-
ever, the edge set is typically much larger than the vertex set due to the distribution of degrees in natural graphs [8]. This also has the downside of vulnerability to stragglers for high-degree vertices due to a large number of incoming and outgoing messages since vertices cannot be further partitioned. We adopt the approach in [8] of splitting the vertex program into an edge computation and a reduction to vertices, allowing us to avoid moving the edges and handle high-degree vertices.

There has been recent work applying incremental iterative data-parallel systems to graph computation. Both Ewen et al. [7] and Murray et al. [4] proposed systems geared towards incremental iterative data-parallel computation and demonstrated performance gains for specialized implementations of PageRank. While this work demonstrates the importance of incremental updates in graph computation, neither proposed a graph oriented view of the data or graph specific optimizations beyond incremental data-flows.

9. DISCUSSION

**Short Computations:** We found that iterative computations are short even on large graphs, so repartitioning mid-computation has limited potential to improve performance. Instead, the focus should be on reducing communication in other ways, such as join rewrite and incremental view maintenance. In hindsight, this view is also taken by mainstream graph systems and even databases, where it is rare to move data around speculatively. Even in MapReduce, only computation is moved speculatively.

**Stragglers and HPC Nodes:** Because stragglers are often due to data skew rather than hardware discrepancies, they affect HPC nodes as well as commodity nodes, accounting for a significant amount of waste. A successful approach to mitigate stragglers would therefore modify the computation and the framework rather than the hardware.

**Future Work:** The GraphX abstraction is independent of the Spark engine on which it is implemented. In particular, it can be expressed on a database engine supporting relational queries. Porting the abstraction to a relational database could take advantage of the existing schemas to provide higher-level primitives that work well with those schemas. Similarly, porting GraphX to a graph database system like Facebook’s TAO [17] would provide a rich source of new research questions regarding the abstraction design.

10. CONCLUSION

Specialized graph computation engines capable of inferring complex recursive properties of graph-structured data are unable to express many of the inherently data-parallel stages in a typical graph-analytics pipeline. Existing graph analytics pipelines [2] resort to multiple stages of data-parallel and graph-parallel systems composed through external storage systems. This approach to graph analytics is inefficient, difficult to adopt to new workloads, and difficult to maintain. We previously introduced GraphX, a distributed graph processing framework that unifies graph-parallel and data-parallel computation in a single system and is capable of succinctly expressing the entire graph analytics pipeline. It presents graphs and collections as first-class objects with a set of primitive operators enabling their composition. However, implemented naively with relational operators it is impractically slow.

In this work we described an implementation of the GraphX abstraction that encodes graphs as collections of edges and vertices along with simple auxiliary index structures, and represents graph computations as a sequence of relational joins and aggregations. It incorporates techniques such as incremental view maintenance and index scans in databases and adapts these techniques to exploit common characteristics of graph computation workloads. The result is a system that achieves performance comparable to contemporary graph-parallel systems in graph computation while retaining the expressiveness of contemporary data-parallel systems.

We have open sourced GraphX at amplab.github.io/graphx. Though it has not been officially released, a brave industry user has successfully deployed GraphX and achieved a speedup of two orders of magnitude over their pre-existing graph analytics pipelines.

11. REFERENCES


