@Author: Jurijus Pacalovas

Algorithm Programf:

1 Chapter

```
if block<=7:
     if e4=="0":
         sda3=sda3+"1"
         block=7


     if e4=="1":
         sda3=sda3+"0"
         block=7



 if block==8:
     if e4=="1":
         sda3=sda3+"0"
         block=0


     if e4=="0":
         sda3=sda3+"1"
         block=0
```

The LU decomposition may not always exist or may be very unstable. More specific, (1) if a leading fc x h sub-matrix of A is singular the LU decomposition fails, and (2) the Gaussian elimination can be unstable because of the possibility of arbitrary small pivots. Interchanging the rows and columns during elimination, such that as the next pivot the matrix element with the largest absolute value is chosen, will alleviate this problem. This is called pivoting for stability. In fact the LU decomposition of the permuted system PAQ is computed, where P and Q denote the row and column permutations.

In least squares problems linear system of equations of the type (ATWA)x=(A W)y have to be solved, the so-called normal equations Nx=b with normal matrix N (ATWA) and right hand sides b (A Wy). The normal matrix A WA is (1) square, (2) symmetric and (3) semi positive definite. For these matrices a special variant of the LU decomposition is considered. If A is square and symmetric a symmetric decomposition A = LDL , with D diagid ,..,d), is possible. If A is also positive definite all elements d. 1 n ^ are positive, which follows directly from the definition of positive definiteness. Let A be positive definite, i.e. x Ax > 0 for all non-zero vectors x; let e. be the i'th unit vector and let x=L~ e . Then, x Ax=d > 0. t I t Now let L'=LD1/2 and U'=D"1/*U, then A = L'L'T. This important result is formalized in the following theorem. Theorem (C.2) If A is an n x n symmetric positive definite matrix, it has a unique triangular factorization LL , where L is a lower triangular matrix with positive diagonal entries.

In least squares problems linear system of equations of the type (ATWA)x=(A W)y have to be solved, the so-called normal equations Nx=b with normal matrix N (ATWA) and right hand sides b (A Wy).

The normal matrix A WA is (1) square, (2) symmetric and (3) semi positive definite. For these matrices a special variant of the LU decomposition is considered. If A is square and symmetric a symmetric decomposition A = LDL , with D diagid ,..,d), is possible. If A is also positive definite all elements d. 1 n ^ are positive, which follows directly from the definition of positive definiteness. Let A be positive definite, i.e. x Ax > 0 for all non-zero vectors x; let e. be the i'th unit vector and let x=L~ e . Then, x Ax=d > 0. t I t Now let L'=LD1/2 and U'=D"1/*U, then A = L'L'T. This important result is formalized in the following theorem. Theorem (C.2) If A is an n x n symmetric positive definite matrix, it has a unique triangular factorization LL , where L is a lower triangular matrix with positive diagonal entries. 202 proof: Any textbook, e.g. [George S Liu, 1981, p. 15] a The triangular factor can be computed by Choleski factorization. The linear system of equations Ax=b can be decomposed into the lower and upper triangular systems L g = b and L x = g, which are solved by simple ; forward and back substitution. Multiple right hand sides are solved by repeated forward and back substitution, the factorization has to be computed only once. The importance of Choleski factorization for our purposes lies in the fact that pivoting for stability is not needed. C.2.3 Stability Considerations A system of equations is ill-conditioned if one of its equations is "almost" linearly dependent on the others. In this case small changes in the data can give large variations in the solution. A measure for the condition is the so-called condition number K, defined by KU) = I A I • f A"1! (C.12) where | | is any matrix norm. Let us define the relative errors v v l „ n H P , K-, A|| e = ■"—n—n—"- and e . = v H A II A II where the vector and matrix norms are compatible. Then the relative error in the solution of the system Ax=b is e s KU) [e, + c j + 0(e2) (C. 13) x AD This bound is the best possible , i.e. no algorithm can do better [Golub & Van Loan, 1983]. For instance, in case of ill-conditioned problems, truncation errors during the factorization can also result in large variations in the solution, or can even make the equations singular. The stability of a method or an algorithm says something about its sensitivity for truncation errors. Consider the round off errors during Gaussian elimination, then the computed triangular factors L and U satisfy LU = A + E (C.14) and |E| is of the order (dim(A) (|A|+|L|•|U|) e), where |E| stands for the matrix with absolute values |e. .| and e denotes the truncation error of the number representation in the computer. With Gaussian elimination there is the possibility that the term with |L||U| becomes large, because there is nothing to prevent small pivots and small pivots give large elements in L and U. Obviously the stability can be improved by selecting in each step of the Gaussian elimination as pivot the element with largest absolute value. This amounts to solving the permuted system PAQ z = Pb with solution x=Qz. The decomposition is now PAQ=LU. Fortunately it can been shown that it is safe not to pivot symmetric positive definite systems. The lower triangular factor L of the Choleski factorization A=L L is given by L =L D , where U =DL with D a diagonal cc c G G G ° matrix with positive entries. The equality |L | = [|A|[shows that the Choleski factor must be nicely bounded, therefore the terms |L ||L | and |E| are also c c nicely bounded. It will now be evident that 1) pivoting for stability is not necessary and 2) the stability depends mainly on the condition number K. If the condition number K ~ 10 p and rounding errors e ~ 10 q then the relative error in the solution x is of the order 10p q. C.3 Choleski Factorization We take the following theorem - concerning a two by two partitioned system - as the basis for Choleski factorization. Algorithms for computing the factorization follow from recursive application of this theorem. Theorem (C.3) If A is an n x n symmetric positive definite matrix, with A = f A 11 A I 21 A 1 21 A 22 J , and i f L = L 11 L 21 1. 22 is the lower triangular matrix from the factorization A=LL , then L = L A and L L = A - A A A = A - L LT . 22 21 21 proof Straightforward multiplication LL gives A = L L , A = L L and A = L L + L L .Solving L by simple forward substitution from 22 22 22 21 21 6 21 ' A21= L11L21 * ive S L21= Sl V ^writing A^= L22L22+ L L,, and substituting the result for L^ gives L^L ^ A^- L21L21=

\ f k z^^u k 2y = A - A A"1A T . 22 21 11 21 Corollary (C.4) L can be computed from A by forward substitution with L . Corollary (C.5) The Choleski sub-factor L of the factor L is equal to the Choleski - 2 Z -1 T factor of the reduced matrix A = A - A A A . 22 22 21 11 21 The last corollary shows that any theorem, lemma, corollary or algorithm 1TT" applicable to A is also applicable toA=A-AA A . This opens the way • 22 22 21 11 21 for recursive application of theorem (C.3). First consider a special case of theorem (C.3). Lemma (C.6) If A is an n x n symmetric positive definite matrix, with A = a a a A T .. s , and if L = 1 1 L is th e lower triangula r matrix from the T 1 'T ö Q T factorization A=LL , then I = Va, 1 = T a and L L = A - = A - 11 = A l s s s a s s proof Straightforwar d multiplicatio n LL give s a = I , a = I 1 and A = L LT + 11 T . So I = Va, 1 = | a and L LT = A - 11T = A - — = s s s L ss s s OL = A 204 Recursive application of the previous lemma on the submatrices A , for s s = l,..n, with A =A, suggests an algorithm for Choleski factorization. This algorithm is called the outer product method after the outer products 11 . The algorithm is given in diagram C. 1. The outer product method can be written as a sequence of transformations T. Let A be an n x n symmetric positive definite matrix with factorization A=LL . Define the block partitioned n x n matrices A. = i I 0 11 and L. = i L 0 11 L I 22J , 21 22J _ where the dimension of the first block is ixi, and A = A _- A..L..L..A.. = 22 22 21 11 11 21 A - L L , so A =k, A si, L =1 and L =L. Then 22 21 21' 0 n 0 n (1) A = L.A.LT , i=0,..,n i i i Furthermore let T. „ =L . ,- L. +1 , then i+l i+l i (2) Li+1 = Li Ti+l = Li + Ti+1 " l (C - 1 5) (3) Ai = T i+ l A i+ l T I+ l n n So the Choleski factor L = j|T. or L = £ T. - (n-1) I. 1=1 l i=l A different algorithm for Choleski factorization is suggested by the following lemma: Lemma (C.7) If A is an n x n symmetric positive definite matrix, with T A = A a 11 i a a i i i , and if L = L 0 11 1 I i ii is the lower triangular matrix from the factorization A=LL , then the row 1 of L is 1 =L a and the i i 11 i diagonal element l..=va .-1 1 ï i i i proof The lemma follows from theorem (C.3) by considering (1) that the solution L to the triangular system L L = A can be computed row by row L LT 21 21 and (2) that I = 1/ l i and a = a - 1 1 ii i l i i ST.. where a is the first entry in A = A - U ii * 22 22 The algorithm suggested by the previous lemma is called the bordering method, because the provisional factor is successively bordered with a row of A, from which the corresponding row of L is computed by forward substitution. The Choleski factor is computed row by row. By changing the order of the computations, so that the elements of the Choleski factor are computed column by column, a third algorithm is derived, the so-called inner product method. 205 In diagram C.1 all three possible algorithms for Choleski factorization are given. Note that the indices in the algorithms presented in diagram C. 1 run faster the higher they are in the alphabet. With the outer product method the Choleski factor is computed column by column, but now using outer products rather than inner products and during the computation the other elements are modified. The access to the matrix elements during factorization is shown in diagram C.2 for full matrices and diagram C.3 for envelope matrices. With the bordering and inner product method the elements of the Choleski factor are computed directly from inner products. The precision of the computations can be increased simply by increasing the precision of the inproduct computations without using extra storage. In case of the outer product method the Choleski factor elements are not computed directly: after a new column has been computed the outer product of this column with itself is added to the remaining elements. Therefore if the stability has to be increased not only the precision of the outer product computation must be improved, but also the wordsize of the modified part must be increased. The cost of the factorization, back and forward substitution are evaluated in terms of operation counts or flops when floating point operations are considered. An operation is defined here as an addition, plus a multiplication, plus the necessary array accesses. The operation count for the Choleski factorization of a full n x n matrix is actually — n and the 1 2 6 operation count for back and forward substitution is — n . Therefore multiple right hand sides can be solved easily and cheaply, 0(n), by repeated forward and back substitution once the initial factorization has been obtained. The operation counts decrease dramatically if the matrices are sparse, i.e. if the matrices only contain relatively few non-zeroes.

2 Chapter

GraphX

Algorithm API:

Abstract In pursuit of graph processing performance, the systems community has largely abandoned general-purpose distributed dataflow frameworks in favor of specialized graph processing systems that provide tailored programming abstractions and accelerate the execution of iterative graph algorithms. In this paper we argue that many of the advantages of specialized graph processing systems can be recovered in a modern general-purpose distributed dataflow system. We introduce GraphX, an embedded graph processing framework built on top of Apache Spark, a widely used distributed dataflow system. GraphX presents a familiar composable graph abstraction that is sufficient to express existing graph APIs, yet can be implemented using only a few basic dataflow operators (e.g., join, map, group-by). To achieve performance parity with specialized graph systems, GraphX recasts graph-specific optimizations as distributed join optimizations and materialized view maintenance. By leveraging advances in distributed dataflow frameworks, GraphX brings low-cost fault tolerance to graph processing. We evaluate GraphX on real workloads and demonstrate that GraphX achieves an order of magnitude performance gain over the base dataflow framework and matches the performance of specialized graph processing systems while enabling a wider range of computation. 1 Introduction The growing scale and importance of graph data has driven the development of numerous specialized graph processing systems including Pregel [22], PowerGraph [13], and many others [7, 9, 37]. By exposing specialized abstractions backed by graph-specific optimizations, these systems can naturally express and efficiently execute iterative graph algorithms like PageRank [30] and community detection [18] on graphs with billions of vertices and edges. As a consequence, graph GraphX (2,500) Spark (30,000) GAS Pregel API (34) PageRank (20) Connected Comp. (20) K-core (60) Triangle Count (50) LDA (220) SVD++ (110) Figure 1: GraphX is a thin layer on top of the Spark general-purpose dataflow framework (lines of code). processing systems typically outperform general-purpose distributed dataflow frameworks like Hadoop MapReduce by orders of magnitude [13, 20]. While the restricted focus of these systems enables a wide range of system optimizations, it also comes at a cost. Graphs are only part of the larger analytics process which often combines graphs with unstructured and tabular data. Consequently, analytics pipelines (e.g., Figure 11) are forced to compose multiple systems which increases complexity and leads to unnecessary data movement and duplication.

Furthermore, in pursuit of performance, graph processing systems often abandon fault tolerance in favor of snapshot recovery. Finally, as specialized systems, graph processing frameworks do not generally enjoy the broad support of distributed dataflow frameworks. In contrast, general-purpose distributed dataflow frameworks (e.g., Map-Reduce [10], Spark [39], Dryad [15]) expose rich dataflow operators (e.g., map, reduce, group-by, join), are well suited for analyzing unstructured and tabular data, and are widely adopted. However, directly implementing iterative graph algorithms using dataflow operators can be challenging, often requiring multiple stages of complex joins. Furthermore, the general-purpose join and aggregation strategies defined in distributed dataflow frameworks do not leverage the common patterns and structure in iterative graph algorithms and therefore miss important optimization opportunities. 600 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14) USENIX Association Historically, graph processing systems evolved separately from distributed dataflow frameworks for several reasons. First, the early emphasis on single stage computation and on-disk processing in distributed dataflow frameworks (e.g., MapReduce) limited their applicability to iterative graph algorithms which repeatedly and randomly access subsets of the graph. Second, early distributed dataflow frameworks did not expose fine-grained control over the data partitioning, hindering the application of graph partitioning techniques. However, new in-memory distributed dataflow frameworks (e.g., Spark and Naiad) expose control over data partitioning and in-memory representation, addressing some of these limitations. Given these developments, we believe there is an opportunity to unify advances in graph processing systems with advances in dataflow systems enabling a single system to address the entire analytics pipeline. In this paper we explore the design of graph processing systems on top of general purpose distributed dataflow systems. We argue that by identifying the essential dataflow patterns in graph computation and recasting optimizations in graph processing systems as dataflow optimizations we can recover the advantages of specialized graph processing systems within a general-purpose distributed dataflow framework. To support this argument we introduce GraphX, an efficient graph processing framework embedded within the Spark [39] distributed dataflow system. GraphX presents a familiar, expressive graph API (Section 3). Using the GraphX API we implement a variant of the popular Pregel abstraction as well as a range of common graph operations. Unlike existing graph processing systems, the GraphX API enables the composition of graphs with unstructured and tabular data and permits the same physical data to be viewed both as a graph and as collections without data movement or duplication. For example, using GraphX it is easy to join a social graph with user comments, apply graph algorithms, and expose the results as either collections or graphs to other procedures (e.g., visualization or rollup). Consequently, GraphX enables users to adopt the computational pattern (graph or collection) that is best suited for the current task without sacrificing performance or flexibility. We built GraphX as a library on top of Spark (Figure 1) by encoding graphs as collections and then expressing the GraphX API on top of standard dataflow operators. GraphX requires no modifications to Spark, revealing a general method to embed graph computation within distributed dataflow frameworks and distill graph computation to a specific join–map–group-by dataflow pattern. By reducing graph computation to a specific pattern we identify the critical path for system optimization. However, naively encoding graphs as collections and executing iterative graph computation using generalpurpose dataflow operators can be slow and inefficient. To achieve performance parity with specialized graph processing systems, GraphX introduces a range of optimizations (Section 4) both in how graphs are encoded as collections as well as the execution of the common dataflow operators. Flexible vertex-cut partitioning is used to encode graphs as horizontally partitioned collections and match the state of the art in distributed graph partitioning. GraphX recasts system optimizations developed in the context of graph processing systems as join optimizations (e.g., CSR indexing, join elimination, and join-site specification) and materialized view maintenance (e.g., vertex mirroring and delta updates) and applies these techniques to the Spark dataflow operators. By leveraging logical partitioning and lineage, GraphX achieves low-cost fault tolerance. Finally, by exploiting immutability GraphX reuses indices across graph and collection views and over multiple iterations, reducing memory overhead and improving system performance. We evaluate GraphX on real-world graphs and compare against direct implementations of graph algorithms using the Spark dataflow operators as well as implementations using specialized graph processing systems. We demonstrate that GraphX can achieve performance parity with specialized graph processing systems while preserving the advantages of a general-purpose dataflow framework. In summary, the contributions of this paper are: 1. an integrated graph and collections API which is sufficient to express existing graph abstractions and enable a much wider range of computation. 2. an embedding of vertex-cut partitioned graphs in horizontally partitioned collections and the GraphX API in a small set of general-purpose dataflow operators. 3. distributed join and materialized view optimizations that enable general-purpose distributed dataflow frameworks to execute graph computation at performance parity with specialized graph systems. 4. a large-scale evaluation on real graphs and common benchmarking algorithms comparing GraphX against widely used graph processing systems. 2 Background In this section we review the design trade-offs and limitations of graph processing systems and distributed dataflow frameworks. At a high level, graph processing systems define computation at the granularity of vertices and their neighborhoods and exploit the sparse dependency structure pre-defined by the graph. In contrast, general-purpose distributed dataflow frameworks define computation as dataflow operators at either the granularity of individual items (e.g., filter, map) or across entire collections (i.e., operations like non-broadcast join that require a shuffle). USENIX Association 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI ’14) 601 2.1 The Property Graph Data Model Graph processing systems represent graph structured data as a property graph [33], which associates user-defined properties with each vertex and edge. The properties can include meta-data (e.g., user profiles and time stamps) and program state (e.g., the PageRank of vertices or inferred affinities). Property graphs derived from natural phenomena such as social networks and web graphs often have highly skewed, power-law degree distributions and orders of magnitude more edges than vertices [18]. In contrast to dataflow systems whose operators (e.g., join) can span multiple collections, operations in graph processing systems (e.g., vertex programs) are typically defined with respect to a single property graph with a pre-declared, sparse structure. While this restricted focus facilitates a range of optimizations (Section 2.3), it also complicates the expression of analytics tasks that may span multiple graphs and sub-graphs. 2.2 The Graph-Parallel Abstraction Algorithms ranging from PageRank to latent factor analysis iteratively transform vertex properties based on the properties of adjacent vertices and edges. This common pattern of iterative local transformations forms the basis of the graph-parallel abstraction. In the graph-parallel abstraction [13], a user-defined vertex program is instantiated concurrently for each vertex and interacts with adjacent vertex programs through messages (e.g., Pregel [22]) or shared state (e.g., PowerGraph [13]). Each vertex program can read and modify its vertex property and in some cases [13, 20] adjacent vertex properties. When all vertex programs vote to halt the program terminates. As a concrete example, in Listing 1 we express the PageRank algorithm as a Pregel vertex program. The vertex program for the vertex v begins by summing the messages encoding the weighted PageRank of neighboring vertices. The PageRank is updated using the resulting sum and is then broadcast to its neighbors (weighted by the number of links). Finally, the vertex program assesses whether it has converged (locally) and votes to halt. The extent to which vertex programs run concurrently differs across systems. Most systems (e.g., [7, 13, 22, 34]) adopt the bulk synchronous execution model, in which all vertex programs run concurrently in a sequence of super-steps. Some systems (e.g., [13, 20, 37]) also support an asynchronous execution model that mitigates the effect of stragglers by running vertex programs as resources become available. However, the gains due to an asynchronous programming model are often offset by the additional complexity and so we focus on the bulksynchronous model and rely on system level techniques (e.g., pipelining and speculation) to address stragglers. def PageRank(v: Id, msgs: List[Double]) { // Compute the message sum var msgSum = 0 for (m <- msgs) { msgSum += m } // Update the PageRank PR(v) = 0.15 + 0.85 * msgSum // Broadcast messages with new PR for (j <- OutNbrs(v)) { msg = PR(v) / NumLinks(v) send_msg(to=j, msg) } // Check for termination if (converged(PR(v))) voteToHalt(v) } Listing 1: PageRank in Pregel: computes the sum of the inbound messages, updates the PageRank value for the vertex, and then sends the new weighted PageRank value to neighboring vertices. Finally, if the PageRank did not change the vertex program votes to halt. While the graph-parallel abstraction is well suited for iterative graph algorithms that respect the static neighborhood structure of the graph (e.g., PageRank), it is not well suited to express computation where disconnected vertices interact or where computation changes the graph structure. For example, tasks such as graph construction from raw text or unstructured data, graph coarsening, and analysis that spans multiple graphs are difficult to express in the vertex centric programming model. 2.3 Graph System Optimizations The restrictions imposed by the graph-parallel abstraction along with the sparse graph structure enable a range of important system optimizations. The GAS Decomposition: Gonzalez et al. [13] observed that most vertex programs interact with neighboring vertices by collecting messages in the form of a generalized commutative associative sum and then broadcasting new messages in an inherently parallel loop. They proposed the GAS decomposition which splits vertex programs into three data-parallel stages: Gather, Apply, and Scatter. In Listing 2 we decompose the PageRank vertex program into the Gather, Apply, and Scatter stages. The GAS decomposition leads to a pull-based model of message computation: the system asks the vertex program for value of the message between adjacent vertices rather than the user sending messages directly from the vertex program. As a consequence, the GAS decomposition enables vertex-cut partitioning, improved work balance, serial edge-iteration [34], and reduced data movement. However, the GAS decomposition also prohibits direct communication between vertices that are not adjacent in the graph and therefore hinders the expression of more general communication patterns. 602 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI ’14) USENIX Association def Gather(a: Double, b: Double) = a + b def Apply(v, msgSum) { PR(v) = 0.15 + 0.85 * msgSum if (converged(PR(v))) voteToHalt(v) } def Scatter(v, j) = PR(v) / NumLinks(v) Listing 2: Gather-Apply-Scatter (GAS) PageRank: The gather phase combines inbound messages. The apply phase consumes the final message sum and updates the vertex property. The scatter phase defines the message computation for each edge. Graph Partitioning: Graph processing systems apply a range of graph-partitioning algorithms [16] to minimize communication and balance computation. Gonzalez et al. [13] demonstrated that vertex-cut [12] partitioning performs well on many large natural graphs. Vertex-cut partitioning evenly assigns edges to machines in a way that minimizes the number of times each vertex is cut. Mirror Vertices: Often high-degree vertices will have multiple neighbors on the same remote machine. Rather than sending multiple, typically identical, messages across the network, graph processing systems [13, 20, 24, 32] adopt mirroring techniques in which a single message is sent to the mirror and then forwarded to all the neighbors. Graph processing systems exploit the static graph structure to reuse the mirror data structures. Active Vertices: As graph algorithms proceed, vertex programs within a graph converge at different rates, leading to rapidly shrinking working sets (the collection of active vertex programs). Recent systems [11, 13, 20, 22] track active vertices and eliminate data movement and unnecessary computation for vertices that have converged. In addition, these systems typically maintain efficient densely packed data-structures (e.g., compressed sparse row (CSR)) that enable constant-time access to the local edges adjacent to active vertices. 2.4 Distributed Dataflow Frameworks We use the term distributed dataflow framework to refer to cluster compute frameworks like MapReduce and its generalizations. Although details vary from one framework to another, they typically satisfy the following properties: 1. a data model consisting of typed collections (i.e., a generalization of tables to unstructured data). 2. a coarse-grained data-parallel programming model composed of deterministic operators which transform collections (e.g., map, group-by, and join). 3. a scheduler that breaks each job into a directed acyclic graph (DAG) of tasks, where each task runs on a (horizontal) partition of data. 4. a runtime that can tolerate stragglers and partial cluster failures without restarting. In MapReduce, the programming model exposes only two dataflow operators: map and reduce (a.k.a., group-by). Each job can contain at most two layers in its DAG of tasks. More modern frameworks such as DryadLINQ [15], Pig [29], and Spark expose additional dataflow operators such as fold and join, and can execute tasks with multiple layers of dependencies. Distributed dataflow frameworks have enjoyed broad adoption for a wide variety of data processing tasks, including ETL, SQL query processing, and iterative machine learning. They have also been shown to scale to thousands of nodes operating on petabytes of data. In this work we restrict our attention to Apache Spark, upon which we developed GraphX. Spark has several features that are particularly attractive for GraphX: 1. The Spark storage abstraction called Resilient Distributed Datasets (RDDs) enables applications to keep data in memory, which is essential for iterative graph algorithms. 2. RDDs permit user-defined data partitioning, and the execution engine can exploit this to co-partition RDDs and co-schedule tasks to avoid data movement. This is essential for encoding partitioned graphs. 3. Spark logs the lineage of operations used to build an RDD, enabling automatic reconstruction of lost partitions upon failures. Since the lineage graph is relatively small even for long-running applications, this approach incurs negligible runtime overhead, unlike checkpointing, and can be left on without concern for performance. Furthermore, Spark supports optional in-memory distributed replication to reduce the amount of recomputation on failure. 4. Spark provides a high-level API in Scala that can be easily extended. This aided in creating a coherent API for both collections and graphs. We believe that many of the ideas in GraphX could be applied to other contemporary dataflow systems and in Section 6 we discuss some preliminary work on a GraphLINQ, a graph framework within Naiad. 3 The GraphX Programming Abstraction We now revisit graph computation from the perspective of a general-purpose dataflow framework. We recast the property graph data model as collections and the graphparallel abstraction as a specific pattern of dataflow operators. In the process we reveal the essential structure of graph-parallel computation and identify the key operators required to execute graph algorithms efficiently. USENIX Association 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI ’14) 603 3.1 Property Graphs as Collections The property graph, described in Section 2.1, can be logically represented as a pair of vertex and edge property collections. The vertex collection contains the vertex properties uniquely keyed by the vertex identifier. In the GraphX system, vertex identifiers are 64-bit integers which may be derived externally (e.g., user ids) or by applying a hash function to the vertex property (e.g., page URL). The edge collection contains the edge properties keyed by the source and destination vertex identifiers. By reducing the property graph to a pair of collections we make it possible to compose graphs with other collections in a distributed dataflow framework. Operations like adding additional vertex properties are naturally expressed as joins against the vertex property collection. The process of analyzing the results of graph computation (i.e., the final vertex and edge properties) and comparing properties across graphs becomes as simple as analyzing and joining the corresponding collections. Both of these tasks are routine in the broader scope of graph analytics but are not well served by the graph parallel abstraction. New property graphs can be constructed by composing different vertex and edge property collections. For example, we can construct logically distinct graphs with separate vertex properties (e.g., one storing PageRanks and another storing connected component membership) while sharing the same edge collection. This may appear to be a small accomplishment, but the tight integration of vertices and edges in specialized graph processing systems often hinders even this basic form of reuse. In addition, graph-specific index data structures can be shared across graphs with common vertex and edge collections, reducing storage overhead and improving performance. 3.2 Graph Computation as Dataflow Ops. The normalized representation of a property graph as a pair of vertex and edge property collections allows us to embed graphs in a distributed dataflow framework. In this section we describe how dataflow operators can be composed to express graph computation. Graph-parallel computation, introduced in Section 2.2, is the process of computing aggregate properties of the neighborhood of each vertex (e.g., the sum of the PageRanks of neighboring vertices weighted by the edge values). We can express graph-parallel computation in a distributed dataflow framework as a sequence of join stages and group-by stages punctuated by map operations. In the join stage, vertex and edge properties are joined to form the triplets view1 consisting of each edge and its corresponding source and destination vertex properties. 1 The triplet terminology derives from the classic Resource Description Framework (RDF), discussed in Section 6. CREATE VIEW triplets AS SELECT s.Id, d.Id, s.P, e.P, d.P FROM edges AS e JOIN vertices AS s JOIN vertices AS d ON e.srcId = s.Id AND e.dstId = d.Id Listing 3: Constructing Triplets in SQL: The column P represents the properties in the vertex and edge property collections. The triplets view is best illustrated by the SQL statement in Listing 3, which constructs the triplets view as a three way join keyed by the source and destination vertex ids. In the group-by stage, the triplets are grouped by source or destination vertex to construct the neighborhood of each vertex and compute aggregates. For example, to compute the PageRank of a vertex we would execute: SELECT t.dstId, 0.15+0.85*sum(t.srcP*t.eP) FROM triplets AS t GROUP BY t.dstId By iteratively applying the above query to update the vertex properties until they converge, we can calculate the PageRank of each vertex. These two stages capture the GAS decomposition described in Section 2.3. The group-by stage gathers messages destined to the same vertex, an intervening map operation applies the message sum to update the vertex property, and the join stage scatters the new vertex property to all adjacent vertices. Similarly, we can implement the GAS decomposition of the Pregel abstraction by iteratively composing the join and group-by stages with data-parallel map stages. Each iteration begins by executing the join stage to bind active vertices with their outbound edges. Using the triplets view, messages are computed along each triplet in a map stage and then aggregated at their destination vertex in a groupby stage. Finally, the messages are received by the vertex programs in a map stage over the vertices. The dataflow embedding of the Pregel abstraction demonstrates that graph-parallel computation can be expressed in terms of a simple sequence of join and group-by dataflow operators. Additionally, it stresses the need to efficiently maintain the triplets view in the join stage and compute the neighborhood aggregates in the group-by stage. Consequently, these stages are the focus of performance optimization in graph processing systems. We describe how to implement them efficiently in Section 4. 3.3 GraphX Operators The GraphX programming abstraction extends the Spark dataflow operators by introducing a small set of specialized graph operators, summarized in Listing 4. 604 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI ’14) USENIX Association class Graph[V, E] { // Constructor def Graph(v: Collection[(Id, V)], e: Collection[(Id, Id, E)]) // Collection views def vertices: Collection[(Id, V)] def edges: Collection[(Id, Id, E)] def triplets: Collection[Triplet] // Graph-parallel computation def mrTriplets(f: (Triplet) => M, sum: (M, M) => M): Collection[(Id, M)] // Convenience functions def mapV(f: (Id, V) => V): Graph[V, E] def mapE(f: (Id, Id, E) => E): Graph[V, E] def leftJoinV(v: Collection[(Id, V)], f: (Id, V, V) => V): Graph[V, E] def leftJoinE(e: Collection[(Id, Id, E)], f: (Id, Id, E, E) => E): Graph[V, E] def subgraph(vPred: (Id, V) => Boolean, ePred: (Triplet) => Boolean) : Graph[V, E] def reverse: Graph[V, E] } Listing 4: Graph Operators: transform vertex and edge collections. The Graph constructor logically binds together a pair of vertex and edge property collections into a property graph. It also verifies the integrity constraints: that every vertex occurs only once and that edges do not link missing vertices. Conversely, the vertices and edges operators expose the graph’s vertex and edge property collections. The triplets operator returns the triplets view (Listing 3) of the graph as described in Section 3.2. If a triplets view already exists, the previous triplets are incrementally maintained to avoid a full join (see Section 4.2). The mrTriplets (Map Reduce

Triplets) operator encodes the essential two-stage process of graphparallel computation defined in Section 3.2. Logically, the mrTriplets operator is the composition of the map and group-by dataflow operators on the triplets view. The user-defined map function is applied to each triplet, yielding a value (i.e., a message of type M) which is then aggregated at the destination vertex using the user-defined binary aggregation function as illustrated in the following: SELECT t.dstId, reduceF(mapF(t)) AS msgSum FROM triplets AS t GROUP BY t.dstId The mrTriplets operator produces a collection containing the sum of the inbound messages keyed by the destination vertex identifier. For example, in Figure 2 we use the mrTriplets operator to compute a collection containing the number of older followers for each user in a social network. Because the resulting collection contains a subset of the vertices in the graph it can reuse the same indices as the original vertex collection. Finally, Listing 4 contains several functions that simF D E A C 42 B 23 30 19 75 16 mapF( ) = 1 A B 42 23 Vertex Id Property A 0 B 2 C 1 D 1 E 0 F 3 Source Property Target Property Resulting Vertices Message to vertex B val graph: Graph[User, Double] def mapUDF(t: Triplet[User, Double]) = if (t.src.age > t.dst.age) 1 else 0 def reduceUDF(a: Int, b: Int): Int = a + b val seniors: Collection[(Id, Int)] = graph.mrTriplets(mapUDF, reduceUDF) Figure 2: Example use of mrTriplets: Compute the number of older followers of each vertex. def Pregel(g: Graph[V, E], vprog: (Id, V, M) => V, sendMsg: (Triplet) => M, gather: (M, M) => M): Collection[V] = { // Set all vertices as active g = g.mapV((id, v) => (v, halt=false)) // Loop until convergence while (g.vertices.exists(v => !v.halt)) { // Compute the messages val msgs: Collection[(Id, M)] = // Restrict to edges with active source g.subgraph(ePred=(s,d,sP,eP,dP)=>!sP.halt) // Compute messages .mrTriplets(sendMsg, gather) // Receive messages and run vertex program g = g.leftJoinV(msgs).mapV(vprog) } return g.vertices } Listing 5: GraphX Enhanced Pregel: An implementation of the Pregel abstraction using the GraphX API. ply perform a dataflow operation on the vertex or edge collections. We define these functions only for caller convenience; they are not essential to the abstraction and can easily be defined using standard dataflow operators. For example, mapV is defined as follows: g.mapV(f) ≡ Graph(g.vertices.map(f), g.edges) In Listing 5 we use the GraphX API to implement a GAS decomposition of the Pregel abstraction. We begin by initializing the vertex properties with an additional field to track active vertices (those that have not voted to halt). Then, while there are active vertices, messages are computed using the mrTriplets operator and the vertex program is applied to the resulting message sums. By expressing message computation as an edgeparallel map operation followed by a commutative associative aggregation, we leverage the GAS decomposition USENIX Association 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI ’14) 605 def ConnectedComp(g: Graph[V, E]) = { g = g.mapV(v => v.id) // Initialize vertices def vProg(v: Id, m: Id): Id = { if (v == m) voteToHalt(v) return min(v, m) } def sendMsg(t: Triplet): Id = if (t.src.cc < t.dst.cc) t.src.cc else None // No message required def gatherMsg(a: Id, b: Id): Id = min(a, b) return Pregel(g, vProg, sendMsg, gatherMsg) } Listing 6: Connected Components: For each vertex we compute the lowest reachable vertex id using Pregel. to mitigate the cost of high-degree vertices. Furthermore, by exposing the entire triplet to the message computation we can simplify algorithms like connected components. However, in cases where the entire triplet is not needed (e.g., PageRank which requires only the source property) we rely on UDF bytecode inspection (see Section 4.3.2) to automatically drop unused fields from join. In Listing 6 we use the GraphX variant of Pregel to implement the connected components algorithm. The connected components algorithm computes the lowest reachable vertex id for each vertex. We initialize the vertex property of each vertex to equal its id using mapV and then define the three functions required to use the GraphX Pregel API. The sendMsg function leverages the triplet view of the edge to only send a message to neighboring vertices when their component id should change. The gatherMsg function computes the minimum of the inbound message values and the vertex program (vProg) determines the new component id. Combining Graph and Collection Operators: Often groups of connected vertices are better modeled as a single vertex. In these cases, it can be helpful coarsen the graph by aggregating connected vertices that share a common characteristic (e.g., web domain) to derive a new graph (e.g., the domain graph). We use the GraphX abstraction to implement graph coarsening in Listing 7. The coarsening operation takes an edge predicate and a vertex aggregation function and collapses all edges that satisfy the predicate, merging their respective vertices. The edge predicate is used to first construct the subgraph of edges that are to be collapsed (i.e., modifying the graph structure). Then the graph-parallel connected components algorithm is run on the subgraph. Each connected component corresponds to a super-vertex in the new coarsened graph with the component id being the lowest vertex id in the component. The super-vertices are constructed by aggregating all the vertices with the same component id using a data-parallel aggregation operator. Finally, we update the edges to link together super-vertices and generate the new graph for subsequent graph-parallel computation. def coarsen(g: Graph[V, E], pred: (Id,Id,V,E,V) => Boolean, reduce: (V,V) => V) = { // Restrict graph to contractable edges val subG = g.subgraph(v => True, pred) // Compute connected component id for all V val cc: Collection[(Id,ccId)] = ConnectedComp(subG).vertices // Merge all vertices in same component val superV: Collection[(ccId,V)] = g.vertices.leftJoin(cc) .groupBy(CC_ID, reduce) // Link remaining edges between components val invG = g.subgraph(ePred = t => !pred(t)) val remainingE: Collection[(ccId,ccId,E)] = invG.leftJoin(cc).triplets.map { e => (e.src.cc, e.dst.cc, e.attr) } // Return the final graph Graph(superV, remainingE) } Listing 7: Coarsen: The coarsening operator merges vertices connected by edges that satisfy the edge predicate. The coarsen operator demonstrates the power of a unified abstraction by combining both data-parallel and graph-parallel operators in a single graph-analytics task. 4 The GraphX System GraphX achieves performance parity with specialized graph processing systems by recasting the graph-specific optimizations of Section 2.3 as optimizations on top of a small set of standard dataflow operators in Spark. In this section we describe these optimizations in the context of classic techniques in traditional database systems including indexing, incremental view maintenance, and join optimizations. Along the way, we quantify the effectiveness of each optimization; readers are referred to Section 5 for details on datasets and experimental setup. 4.1 Distributed Graph Representation GraphX represents graphs internally as a pair of vertex and edge collections built on the Spark RDD abstraction. These collections introduce indexing and graph-specific partitioning as a layer on top of RDDs. Figure 3 illustrates the physical representation of the horizontally partitioned vertex and edge collections and their indices. The vertex collection is hash-partitioned by the vertex ids. To support frequent joins across vertex collections, vertices are stored in a local hash index within each partition (Section 4.2). Additionally, a bitmask stores the visibility of each vertex, enabling soft deletions to promote index reuse (Section 4.3.1). 606 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14) USENIX Association Edges 1 2 1 3 edge partition A 1 4 5 4 edge partition B 1 5 edge partition C 1 6 6 5 clustered indices on source vertex 2 3 4 edge partition A edge partition B 6 edge partition C 1 5 Graph Vertices vertex partition A 1 2 3 1 1 1 bitmask vertex partition B 4 5 6 1 1 0 bitmask hash indices on vertex id Routing Table partition A C 1 B 1 A 1,2,3 partition B A 4,5 C 5,6 B Figure 3: Distributed Graph Representation: The graph (left) is represented as a vertex and an edge collection (right). The edges are divided into three edge partitions by applying a partition function (e.g., 2D Partitioning). The vertices are partitioned by vertex id. Copartitioned with the vertices, GraphX maintains a routing table encoding the edge partitions for each vertex. If vertex 6 and adjacent edges (shown with dotted lines) are restricted from the graph (e.g., by subgraph), they are removed from the corresponding collection by updating the bitmasks thereby enabling index reuse. The edge collection is horizontally partitioned by a user-defined partition function. GraphX enables vertexcut partitioning, which minimizes communication in natural graphs such as social networks and web graphs [13]. By default edges are assigned to partitions based on the partitioning of the input collection (e.g., the original placement on HDFS). However, GraphX provides a range of built-in partitioning functions, including a 2D hash partitioner with strong upper bounds [8] on the communication complexity of operators like mrTriplets. This flexibility in edge placement is enabled by the routing table, described in Section 4.2. For efficient lookup of edges by their source and target vertices, the edges within a partition are clustered by source vertex id using a compressed sparse row (CSR) [35] representation and hash-indexed by their target id. Section 4.3.1 discusses how these indices accelerate iterative computation. Index Reuse: GraphX inherits the immutability of Spark and therefore all graph operators logically create new collections rather than destructively modifying existing ones. As a result, derived vertex and edge collections can often share indices to reduce memory overhead and accelerate local graph operations. For example, the hash index on vertices enables fast aggregations, and the resulting aggregates share the index with the original vertices. In addition to reducing memory overhead, shared indices enable faster joins. Vertex collections sharing the same index (e.g., the vertices and the messages from mrTriplets) can be joined by a coordinated scan, similar to a merge join, without requiring any index lookups. In our benchmarks, index reuse reduces the per-iteration runtime of PageRank on the Twitter graph by 59%. The GraphX operators try to maximize index reuse. Operators that do not modify the graph structure (e.g., mapV) automatically preserve indices. To reuse indices for operations that restrict the graph structure (e.g., subgraph), GraphX relies on bitmasks to construct restricted views. In cases where index reuse could lead to decreased efficiency (e.g., when a graph is highly filtered), GraphX uses the reindex operator to build new indices. 4.2 Implementing the Triplets View As described in Section 3.2, a key stage in graph computation is constructing and maintaining the triplets view, which consists of a three-way join between the source and destination vertex properties and the edge properties. Vertex Mirroring: Because the vertex and edge property collections are partitioned independently, the join requires data movement. GraphX performs the three-way join by shipping the vertex properties across the network to the edges, thus setting the edge partitions as the join sites [21]. This approach substantially reduces communication for two reasons. First, real-world graphs commonly have orders of magnitude more edges than vertices. Second, a single vertex may have many edges in the same partition, enabling substantial reuse of the vertex property. Multicast Join: While broadcast join in which all vertices are sent to each edge partition would ensure joins occur on edge partitions, it could still be inefficient since most partitions require only a small subset of the vertices to complete the join. Therefore, GraphX introduces a multicast join in which each vertex property is sent only to the edge partitions that contain adjacent edges. For each vertex GraphX maintains the set of edge partitions with adjacent edges. This join site information is stored in a routing table which is co-partitioned with the vertex collection (Figure 3). The routing table is associated with the edge collection and constructed lazily upon first instantiation of the triplets view. The flexibility in partitioning afforded by the multicast join strategy enables more sophisticated applicationspecific graph partitioning techniques. For example, by adopting a per-city partitioning scheme on the Facebook social network graph Ugander et al. [38] showed a 50.5% reduction in query time. In Section 5.1 we exploit the optimized partitioning of our sample datasets to achieve up to 56% reduction in runtime and 5.8× reduction in communication compared to a 2D hash partitioning. Partial Materialization: Vertex replication is performed eagerly when vertex properties change, but the local joins at the edge partitions are left unmaterialized to USENIX Association 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14) 607 Figure 4: Impact of incrementally maintaining the triplets view: For both PageRank and connected components, as vertices converge, communication decreases due to incremental view maintenance. The initial rise in communication is due to message compression (Section 4.4); many PageRank values are initially the same. avoid duplication. Instead, mirrored vertex properties are stored in hash maps on each edge partition and referenced when constructing triplets. Incremental View Maintenance: Iterative graph algorithms often modify only a subset of the vertex properties in each iteration. We therefore apply incremental view maintenance to the triplets view to avoid unnecessary movement of unchanged data. After each graph operation, we track which vertex properties have changed since the triplets view was last constructed. When the triplets view is next accessed, only the changed vertices are re-routed to their edge-partition join sites and the local mirrored values of the unchanged vertices are reused. This functionality is managed automatically by the graph operators. Figure 4 illustrates the impact of incremental view maintenance for both PageRank and connected components on the Twitter graph. In the case of PageRank, where the number of active vertices decreases slowly because the convergence threshold was set to 0, we see only moderate gains. In contrast, for connected components most vertices are within a short distance of each other and converge quickly, leading to a substantial reduction in communication from incremental view maintenance. Without incremental view maintenance, the triplets view would need to be reconstructed from scratch every iteration, and communication would remain at its peak throughout the computation. 4.3 Optimizations to mrTriplets GraphX incorporates two additional query optimizations for the mrTriplets operator: filtered index scanning and automatic join elimination. 4.3.1 Filtered Index Scanning The first stage of the mrTriplets operator logically involves a scan of the triplets view to apply the user-defined Figure 5: Sequential scan vs index scan: Connected components on the Twitter graph benefits greatly from switching to index scan after the 4th iteration, while PageRank benefits only slightly because the set of active vertices is large even at the 15th iteration. map function to each triplet. However, as iterative graph algorithms converge, their working sets tend to shrink, and the map function skips all but a few triplets. In particular, the map function only needs to operate on triplets containing vertices in the active set, which is defined by an application-specific predicate. Directly scanning all triplets becomes increasingly wasteful as the active set shrinks. 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 mrTriplets we still must sequentially scan 1.5 billion edges and check whether their vertices are in the active set. To address this problem, we introduced an indexed scan for the triplets view. The application expresses the current active set by restricting the graph using the subgraph operator. The vertex predicate is pushed to the edge partitions, where it can be used to filter the triplets using the CSR index on the source vertex id (Section 4.1). We measure the selectivity of the vertex predicate and switch from sequential scan to clustered index scan when the selectivity is less than 0.8. Figure 5 illustrates the benefit of index scans in PageRank and connected components. As with incremental view maintenance, index scans lead to a smaller improvement in runtime for PageRank and a substantial improvement in runtime for connected components. Interestingly, in the initial iterations of connected components, when the majority of the vertices are active, a sequential scan is slightly faster as it does not require the additional index lookup. It is for this reason that we dynamically switch between full and indexed scans based on the fraction of active vertices. 608 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI ’14) USENIX Association 4.3.2 Automatic Join Elimination In some cases, operations on the triplets view may access only one of the vertex properties or none at all. For example, when mrTriplets is used to count the degree of each vertex, the map UDF does not access any vertex properties. Similarly, when computing messages in PageRank only the source vertex properties are used. GraphX uses a JVM bytecode analyzer to inspect userdefined functions at runtime and determine whether the source or target vertex properties are referenced. If only one property is referenced, and if the triplets view has not already been materialized, GraphX automatically rewrites the query plan for generating the triplets view from a threeway join to a two-way join. If none of the vertex properties are referenced, GraphX eliminates the join entirely. This modification is possible because the triplets view follows the lazy semantics of RDDs in Spark. If the user never accesses the triplets view, it is never materialized. A call to mrTriplets is therefore able to rewrite the join needed to generate the relevant part of the triplets view. Figure 6 demonstrates the impact of this physical execution plan rewrite on communication and runtime for PageRank on the Twitter follower graph. We see that join elimination cuts the amount of data transferred in half, leading to a significant reduction in overall runtime. Note that on the first iteration there is no reduction in communication. This is due to compression algorithms that take advantage of all messages having exactly the same initial value. However, compression and decompression still consume CPU time so we still observe nearly a factor of two reduction in overall runtime. 4.4 Additional Optimizations 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: Spark’s default shuffle implementation materializes the temporary data to disk. We modified the shuffle phase to materialize map outputs in memory and remove this temporary data using a timeout. 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: While GraphX uses 64-bit vertex ids, in most cases the ids are much smaller than 264. To exploit this fact, during shuffling, we encode integers Figure 6: Impact of automatic join elimination on communication and runtime: We ran PageRank for 20 iterations on the Twitter dataset with and without join elimination and found that join elimination reduces the amount of communication by almost half and substantially decreases the total execution time. 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 256 require 5 bytes to encode. This technique reduces communication in PageRank by 20%. 5 System Evaluation In this section we demonstrate that, for iterative graph algorithms, GraphX is over an order of magnitude faster than directly using the general-purpose dataflow operators described in Section 3.2 and is comparable to or faster than specialized graph processing systems. We evaluate the performance of GraphX on several graph-analytics tasks, comparing it with the following: 1. Apache Spark 0.9.1: the base distributed dataflow system for GraphX. We compare against Spark to demonstrate the performance gains relative to the baseline distributed dataflow framework. 2. Apache Giraph 1.1: an open source graph computation system based on the Pregel abstraction. 3. GraphLab 2.2 (PowerGraph): the open-source graph computation system based on the GAS decomposition of vertex programs. Because GraphLab USENIX Association 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI ’14) 609 is implemented in C++ and all other systems run on the JVM, given identical optimizations, we would expect GraphLab to have a slight performance advantage. We also compare against GraphLab without sharedmemory parallelism (denoted GraphLab NoSHM). GraphLab communicates between workers on the same machine using shared data structures. In contrast, Giraph, Spark, and GraphX adopt a shared-nothing worker model incurring extra serialization overhead between workers. To isolate this overhead, we disabled shared-memory by forcing GraphLab workers to run in separate processes. It is worth noting that the shared data structures in GraphLab increase the complexity of the system. Indeed, we encountered and fixed a critical bug in one of the GraphLab shared data structures. The resulting patch introduced an additional lock which led to a small increase in thread contention. As a consequence, in some cases (e.g., Figure 7c) disabling shared memory contributed to a small improvement in performance. All experiments were conducted on Amazon EC2 using 16 m2.4xlarge worker nodes. Each node has 8 virtual cores, 68 GB of memory, and two hard disks. The cluster was running 64-bit Linux 3.2.28. We plot the mean and standard deviation for multiple trials of each experiment. 5.1 System Comparison Cross-system benchmarks are often unfair due to the difficulty in tuning each system equitably. We have endeavored to minimize this effect by working closely with experts in each of the systems to achieve optimal configurations. We emphasize that we are not claiming GraphX is fundamentally faster than GraphLab or Giraph; these systems could in theory implement the same optimizations as GraphX. Instead, we aim to show that it is possible to achieve comparable performance to specialized graph processing systems using a general dataflow engine while gaining common dataflow features such as fault tolerance. While we have implemented a wide range of graph algorithms on top of GraphX, we restrict our performance evaluation to PageRank and connected components. These two representative graph algorithms are implemented in most graph processing systems, have wellunderstood behavior, and are simple enough to serve as an effective measure of the system's performance. To ensure a fair comparison, our PageRank implementation is based on Listing 1; it does not exploit delta messages and therefore benefits less from indexed scans and incremental view maintenance. Conversely, the connected components implementation only sends messages when a vertex must change component membership and therefore does benefit from incremental view maintenance. Dataset Edges Vertices twitter-2010 [5, 4] 1,468,365,182 41,652,230 uk-2007-05 [5, 4] 3,738,733,648 105,896,555 Table 1: Graph Datasets. Both graphs have highly skewed power-law degree distributions. For each system, we ran both algorithms on the twitter2010 and uk-2007-05 graphs (Table 1). For Giraph and GraphLab we used the included implementations of these algorithms. For Spark we implemented the algorithms both using idiomatic dataflow operators (Naive Spark, as described in Section 3.2) and using an optimized implementation (Optimized Spark) that eliminates movement of edge data by pre-partitioning the edges to match the partitioning adopted by GraphX. Both GraphLab and Giraph partition the graph according to specialized partitioning algorithms. While GraphX supports arbitrary user defined graph partitioners including those used by GraphLab and Giraph, the default partitioning strategy is to construct a vertex-cut that matches the input edge data layout thereby minimizing edge data movement when constructing the graph. However, as point of comparison we also tested GraphX using a randomized vertex-cut (GraphX Rand). We found (see Figure 7) that for the specific datasets used in our experiments the input partitioning, which was determined by a specialized graph compression format [4], actually resulted in a more communication-efficient vertex-cut partitioning. Figures 7a and 7c show the total runtimes for connected components algorithm. We have excluded Giraph and Optimized Spark from Figure 7c because they were unable to scale to the larger web-graph in the allotted memory of the cluster. While the basic Spark implementation did not crash, it was forced to re-compute blocks from disk and exceeded 8000 seconds per iteration. We attribute the increased memory overhead to the use of edge-cut partitioning and the need to store bi-directed edges and messages for the connected components algorithm. Figures 7b and 7d show the total runtimes for PageRank for 20 iterations on each system. In Figure 7b, GraphLab outperforms GraphX largely due to shared-memory parallelism; GraphLab without shared memory parallelism is much closer in performance to GraphX. In 7d, GraphX outperforms GraphLab because the input partitioning of uk-2007-05 is highly efficient, resulting in a 5.8x reduction in communication per iteration. 5.2 GraphX Performance Scaling: In Figure 8 we evaluate 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. 610 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14) USENIX Association (a) Conn. Comp. Twitter (b) PageRank Twitter (c) Conn. Comp. uk-2007-05∗ (d) PageRank uk-2007-05 Figure 7: System Performance Comparison. (c) Spark did not finish within 8000 seconds, Giraph and Spark + Part. ran out of memory. Figure 8: Strong scaling for PageRank on Twitter (10 Iterations) Figure 9: Effect of partitioning on communication Figure 10: Fault tolerance for PageRank on uk-2007-05 While this is hardly linear scaling, it is actually slightly better than the 3.2x speedup reported by GraphLab [13]. The poor scaling performance of PageRank has been attributed by [13] to high communication overhead relative to computation for the PageRank algorithm. It may seem surprising that GraphX scales slightly better than GraphLab given that Spark does not exploit shared memory parallelism and therefore forces the graph to be partitioned across processors rather than machines. However, Figure 9 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 an approximately 2-fold increase in communication. Returning to the analysis of vertex-cut partitioning conducted by [13], we find that the vertex-cut partitioning adopted by GraphX mitigates the 8-fold increase in communication. Fault tolerance: Existing graph systems only support checkpoint-based fault tolerance, which most users leave disabled due to the performance overhead. GraphX is built on Spark, which provides lineage-based fault tolerance with negligible overhead as well as optional dataset replication (Section 2.4). We benchmarked these fault tolerance options for PageRank on uk-2007-05 by killing a worker in iteration 11 of 20, allowing Spark to recover by using the remaining copies of the lost partitions or recomputing them, and measuring how long the job took in total. For comparison, we also measured the end-to-end time for running until failure and then restarting from scratch on the remaining nodes using a driver script, as would be necessary in existing graph systems. Figure 10 shows that in case of failure, both replication and recomputation are faster than restarting the job from scratch, and moreover they are performed transparently by the dataflow engine. 6 Related Work In Section 2 we described the general characteristics shared across many of the earlier graph processing systems. However, there are some exceptions to many of these characteristics that are worth noting. While most of the work on large-scale distributed graph processing has focused on static graphs, several systems have focused on various forms of stream processing. One USENIX Association 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI ’14) 611 of the earlier examples is Kineograph [9], a distributed graph processing system that constructs incremental snapshots of the graph for offline static graph analysis. In the multicore setting, GraphChi [17] and later X-Stream [34] introduced support for the addition of edges between existing vertices and between computation stages. Although conceptually GraphX could support the incremental introduction of edges (and potentially vertices), the existing data-structures would require additional optimization. Instead, GraphX focuses on efficiently supporting the removal of edges and vertices: essential functionality for offline sub-graph analysis. Most of the optimizations and programming models of earlier graph processing systems focus on a single graph setting. While some of these systems [19, 13, 34] are capable of operating on multiple graphs independently, they do not expose an API or present optimizations for operations spanning graphs (or tables). One notable exception is CombBLAS [7] which treats graphs (and data more generally) as matrices and supports generalized binary algebraic operators. In contrast GraphX preserves the native semantics of graphs and tables and provides a simple API to combine data across these representations. The triplets view in GraphX is related to the classic Resource Description Framework [23]

(RDF) data model which encodes graph structured data as subjectpredicate-object triplets (e.g., NYC-isA-city). Numerous systems [1, 6, 28] have been proposed for storing and executing SPARQL [31] subgraph queries against RDF triplets. Like GraphX, these systems rely heavily on indexing and clustering for performance. Unlike GraphX, these systems are not distributed or do not address iterative graph algorithms. Nonetheless, we believe that the optimizations techniques developed for GraphX may benefit the design of distributed graph query processing. There have been several recent efforts at exploring graph algorithms within dataflow systems. Najork et al. [27], compares implementations of a range of graph algorithms on the DryadLINQ [15] and SQL Server dataflow systems. However, the resulting implementations are fairly complex and specialized, and little is discussed about graph-specific optimizations. Both Ewen et al. [11] and Murray et al. [26] proposed dataflow systems geared towards incremental iterative computation and demonstrated performance gains for specialized implementations of graph algorithms. While this work highlights the importance of incremental updates in graph computation, neither proposed a general method to express graph algorithms or graph specific optimizations beyond incremental dataflows. Nonetheless, we believe that the GraphX system could be ported to run on-top of these dataflow frameworks and would potentially benefit from advances like timely dataflows [26]. At the time of publication, the Microsoft Naiad team had announced initial work on a system called GraphLINQ [25], a graph processing framework on-top of Naiad which shares many similarities to GraphX. Like GraphX, GraphLINQ aims to provides rich graph functionality within a general-purpose dataflow framework. In particular GraphLINQ presents a GraphReduce operator that is semantically similar to the mrTriplets operator in GraphX except that it operates on streams of vertices and edges. The emphasis on stream processing exposes opportunities for classic optimizations in the stream processing literature as well as recent developments like the Naiad timely dataflows [26]. We believe this further supports the advantages of embedding graph processing within more general-purpose data processing systems. Others have explored join optimizations in distributed dataflow frameworks. Blanas et al. [3] show that broadcast joins and semi-joins compare favorably with the standard MapReduce style shuffle joins when joining a large table (e.g., edges) with a smaller table (e.g., vertices). Closely related is the work by Afrati et al. [2] which explores optimizations for multi-way joins in a MapReduce framework. They consider joining a large relation with multiple smaller relations and provide a partitioning and replication strategy similar to classic 2D partitioning [8]. However, in contrast to our work, they do not construct a routing table forcing the system to broadcast the smaller relations (e.g., the vertices) to all partitions of the larger relation (e.g., the edges) that could have matching tuples. Furthermore, they force a particular hash partitioning on the larger relation precluding the opportunity for user defined graph partitioning algorithms (e.g., [16, 38, 36]). 7 Discussion The work on GraphX addressed several key themes in data management systems and system design: Physical Data Independence: GraphX allows the same physical data to be viewed as collections and as graphs without data movement or duplication. As a consequence the user is free to adopt the best view for the immediate task. We demonstrated that operations on collections and graphs can be efficiently implemented using the same physical representation and underlying operators. Our experiments show that this common substrate can match the performance of specialized graph systems. Graph Computation as Joins and Group-By: The design of the

GraphX system reveals the strong connection between distributed graph computation and distributed join optimizations. When viewed through the lens of dataflow operators, graph computation reduces to join and group-by operators. These two operators correspond to the Scatter and Gather stages of the GAS abstraction. Likewise, the optimizations developed for graph processing systems reduce to indexing, distributed join 612 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI ’14) USENIX Association Contemporary Graph Processing Systems < / > < / > < / > XML ETL Slice Compute Analyze Figure 11: Graph Analytics Pipeline: requires multiple collection and graph views of the same data. site selection, multicast joins, partial materialization, and incremental view maintenance. The Narrow Waist: In designing the GraphX abstraction, we sought to develop a thin extension on top of dataflow operators with the goal of identifying the essential data model and core operations needed to support graph computation. We aimed for a portable framework that could be embedded in a range of dataflow frameworks. We believe that the GraphX design can be adopted by other dataflow systems, including MPP databases, to efficiently support a wide range of graph computations. Analytics Pipelines: GraphX provides the ability to stay within a single framework throughout the analytics process, eliminating the need to learn and support multiple systems (e.g., Figure 11) or write data interchange formats and plumbing to move between systems. As a consequence, it is substantially easier to iteratively slice, transform, and compute on large graphs as well as to share data-structures across stages of the pipeline. The gains in performance and scalability for graph computation translate to a tighter analytics feedback loop and therefore a more efficient work flow. Adoption: GraphX was publicly released as part of the 0.9.0 release of the Apache Spark open-source project.2 It has since generated substantial interest in the community and has been used in production at various places.3 Despite its nascent state, there has been considerable opensource contribution to GraphX with contributors providing some of the core graph functionality. We attribute this to its wide applicability and the simple abstraction built on top of an existing, popular dataflow framework. 8 Conclusions and Future Work In this work we introduced GraphX, an efficient graph processing system that enables distributed dataflow frameworks such as Spark to naturally express and efficiently execute iterative graph algorithms. We identified a simple pattern of join–map–group-by dataflow operators that forms the basis of graph-parallel computation. Inspired by this observation, we proposed the GraphX abstraction

Chapeter3:

```
if block<=7:
     if e4=="1":
         sda3=sda3+"1"
         block=7


     if e4=="0":
         sda3=sda3+"0"
         block=7



 if block==8:
     if e4=="1":
         sda3=sda3+"0"
         block=0


     if e4=="0":
         sda3=sda3+"1"
         block=0
```

Chapeter4:

```
if block<=7:

  if e4=="0":

    sda3=sda3+"0"

    e4="0"

    block=7

    e4=""




  if e4=="1":

    sda3=sda3+"1"

    e4="1"

    block=7

    e4=""
```

```
if block==8:
    if e4=="1":
        sda3=sda3+"0"
        e4="0"
        block=0
        e4=""


    if e4=="0":
        sda3=sda3+"1"
        e4="1"
        block=0
        e4=""



if block<=7:
    if e4=="0":
        sda3=sda3+"1"
        e4="1"
        block=7
        e4=""
    if e4=="1":
        sda3=sda3+"0"
        e4="0"
        block=7
        e4=""


if block==8:
    if e4=="1":
        sda3=sda3+"0"
        e4="0"
        block=0
        e4=""


if e4=="0":
        sda3=sda3+"1"
        e4="1"
        block=0
        e4=""
```

20 Level Jurijus Pacalovas GOD 100%