Query Optimization of Distributed Pattern Matching

Jiewen Huang, Kartik Venkatraman, Daniel J. Abadi
Yale University
jiewen.huang@yale.edu, kartik.venkatraman@yale.edu, dna@cs.yale.edu

***Abstract*—Greedy algorithms for subgraph pattern matching operations are often sufficient when the graph data set can be held in memory on a single machine. However, as graph data sets increasingly expand and require external storage and partitioning across a cluster of machines, more sophisticated query optimization techniques become critical to avoid explosions in query latency. In this paper, we introduce several query optimization techniques for distributed graph pattern matching. These techniques include (1) a System-R style dynamic programming-based optimization algorithm that considers both linear and bushy plans, (2) a cycle detection-based algorithm that leverages cycles to reduce intermediate result set sizes, and (3) a computation reusing technique that eliminates redundant query execution and data transfer over the network. Experimental results show that these algorithms can lead to an order of magnitude improvement in query performance.**

I. Introduction

The graph data model is becoming an increasingly popular way to represent data for various applications. Reasons for this include: (1) It can be less complex for a user to shoehorn semi-structured or sparse data into a vertex-edge-vertex data model than a relational data model, (2) some increasingly popular data sets (such as the Twitter, Facebook, and LinkedIn social networks) are most naturally reasoned about using a graph paradigm, and (3) graph operations, such as shortest path calculations, subgraph pattern matching, and PageRank are easily expressed over a graph data model.

Many graph data sets are becoming too large to manage on a single machine, and therefore clusters of machines are being deployed to process, store, manage, and analyze graph data. For instance, as of 2012, Facebook's user graph has 900 million vertices (and the average degree of a vertex is 130) [1]. In Semantic Web community, the Linking Open Data movement has collected 6 billion triples (a triple is equivalent to an edge in a graph) from 300 interconnected data sets [3]. Since many graph algorithms were originally designed with the assumption that the entire graph can be stored in memory on a single machine, these distributed architectures require revisiting these algorithms in a distributed context, as considerations such as network latency and throughput can bottleneck the traditional implementation of these algorithms.

Subgraph pattern matching is a particularly important operation that must be revisited for distributed graph stores. Subgraph matching operations are heavily used in social network data mining operations (e.g. counting triangles for gauging social influence of celebrities [33]), SPARQL queries over the Linked Data graph, and machine learning algorithms that power recommendation engines for e-commerce retail applications and entertainment choices.

This paper is the first (to the best of our knowledge) to explicitly use System-R style dynamic programming techniques [30] in order to optimize distributed subgraph pattern matching. However, we find that these traditional algorithms need to be modified in three ways in order to work well for graph data:

- Although others have noted that even in the traditional relational context, System-R's consideration of only left-deep join trees can lead to a suboptimal optimization strategy [21], the consideration of bushy plans for distributed graph pattern matching queries is particularly important in order to reduce network traffic and sizes of intermediate output. The heuristics for which bushy plans to consider should leverage graph characteristics.
- Cycles appear more frequently in query patterns over a graph model than data represented in other models. They can be potentially leveraged to improve query performance and should be explicitly considered during plan generation.
- In general, distributed subgraph pattern matching is performed by finding components of the subgraph separately, and joining these components together. However, when pure graph patterns are being searched for (without any selection predicates on vertex or edge attributes), the intermediate result sets tend to be extremely large.

In this paper we introduce two query optimization frameworks for subgraph pattern matching. In particular, given a data store represented in the graph data model, and a query that requests all instances of a particular graph pattern within the data store, we provide algorithms that generate a series of query execution plans, estimate the cost of each of these plans, and select the plan with lowest cost for execution. We make no assumptions about how the graph data is partitioned across a cluster, except that all (directed) edges emanating from the same vertex are stored on the same physical machine, and that a deterministic function (e.g. a hash function) can be applied to any edge in order to determine its location. Furthermore, we make no assumptions about the subgraph being matched — our algorithms apply to both unattributed subgraphs (where just the structure of the graph pattern is being matched) and attributed subgraphs (where each vertex and edge in the graph data set may have attributes associated with it, and the subgraph may include predicates on these attributes in order to reduce the scope of the search). Our work makes the following contributions:

- We propose a dynamic programming-based optimization framework that considers bushy plans without encountering query plan space explosion (Section III).

978-1-4799-2555-1/14/$31.00 © 2014 IEEE

- We propose an cycle detection-based optimization framework based on the observation that cycles tend to significantly reduce the size of intermediate result sets (Section IV).
- We introduce a computation reusing technique which eliminates repetitive identical subquery execution and redundant network transfer of intermediate result sets within a query (Section VI).

Our experimental results show that our proposed techniques improve performance of subgraph pattern matching in a distributed graph store by an order of magnitude over commonly used greedy algorithms, and often result in even larger performance gains.

II. Preliminaries

A. Subraph Patten Matching

The subgraph pattern matching problem is to find all subgraphs that match certain patterns in a graph. Although there are many variations on this problem [14], it is traditionally defined in terms of subgraph isomorphism [36], which only considers the graph structure. In practice, many graphs are annotated with semantic information, and this semantic information can be included as part of a pattern to be matched. Semantics are often represented as the types, labels and attributes of vertices and edges. Types and labels can be readily converted to attributes, so we will refer to all semantic additions as "attributed graphs". In this paper we consider both subgraph isomorphism operations and more general matching over attributed graphs. The formal definition that encompasses both types of matching is given below.

Definition 1. *A directed edge is represented as (A,B) or $A \rightarrow B$. Vertex A is called the* **originating vertex** *and vertex B is the called the* **destination vertex**.

Definition 2. *A* ***data graph*** *is a directed graph $G = (V, E, A_V, A_E)$. V is a set of vertices, and E is a set of directed edges. $\forall e \in E$, $e = (v_1, v_2)$ and $v_1, v_2 \in V$. $\forall v \in V$, $A_V(v)$ is a tuple $(A_1 = a_1, A_2 = a_2, ..., A_n = a_n)$, where A_i is an attribute of v and a_i is a constant, $1 \leq i \leq n$. A_E is a similar function defined on E. A* ***subgraph*** *$G' = (V', E')$ is a subgraph of G if and only if $V' \subseteq V$ and $E' \subseteq V' \times V' \subseteq E$.*

Definition 3. *A* ***pattern graph / query graph / query*** *is a directed graph $Q = (V_Q, E_Q, f_{V_Q}, f_{E_Q})$. V_Q is a set of vertices and E_Q is a set of directed edges. $\forall e \in E_Q$, $e = (v_1, v_2)$ and $v_1, v_2 \in V_Q$. $\forall v \in V_Q$, $f_{V_Q}(v)$ is a formula which consists of predicates connected by logical operators $\vee$ (disjunction), $\wedge$ (conjunction) and $\neg$ (negation). A predicate takes attributes of v, constants and functions as arguments and applies a comparison operator, such as $<$, $\leq$, $=$, $\neq$, $>$ and $\geq$. f_{E_Q} is a similar function defined on E_Q.*

Definition 4. *A* ***match*** *of a pattern graph $Q = (V_Q, E_Q, f_{V_Q}, f_{E_Q})$ in a data graph $G = (V, E, A_V, A_E)$ is a subgraph $G' = (V', E')$ of G such that*

(1) There are two bijections, one from V_Q to V' and one from E_Q to E'. In other words, there are one-to-one correspondences from V_Q to V' and from E_Q to E', respectively.

(2) A_V satisfies f_{V_Q} on V' and A_E satisfies f_{E_Q} on E'.

Intuitively, A_V and A_E define the vertex and edge attributes while f_{V_Q} and f_{E_Q} specify the match conditions on attributes. The task of **subgraph pattern matching** is to find all matches.

B. Data Partitioning and Joins

In distributed and parallel database systems, data partitioning has a significant impact on query execution. The costs of distributed joins depend heavily on how data is partitioned across the cluster of machines that store different partitions of the data. For data sets represented in the graph data model, vertices (and any associated attributes of these vertices) are typically partitioned by applying a deterministic function (often a hash function) to the vertex, where the result of this function indicates where the vertex should be stored. For edges, one of the two vertices that the edge connects is designated the **partitioning vertex** (for directed edges, this is typically the originating vertex), and the same partitioning function is applied to that vertex to determine where that edge will be placed. For example, edges A→B and A→C, which share the same originating vertex (A), are guaranteed to be placed in the same partition. On the other hand, edges A→B and C→B (which share the same destination vertex) may not be placed together.

In general, the task of subgraph pattern matching is performed piecewise — searching for **fragments** of the subgraph independently and joining these fragments on shared vertices. These fragments usually start as small as a single edge, and get successively larger with each join. We call all matches that have been found for each subgraph fragment an **intermediate result set**. Each intermediate result set for a subgraph fragment has a vertex in the fragment designated as the partitioning vertex, and the same deterministic function that was used to partition vertices and edges in the raw graph dataset is used to partition the intermediate result sets based on this vertex. The joining of intermediate result sets can be performed using the standard join methods in distributed and parallel database systems: co-located joins, directed joins, distributed hash joins, and broadcast joins.

The co-located join is a local join (it does not require any data transfer over the network). It can be used when the join vertex is the partitioning vertex for all intermediate result sets being joined. The partitioning vertex of the intermediate result set of the join is the join vertex.

If only one of the intermediate result sets being joined is partitioned by the join vertex, then a directed join can be used. In this case, the intermediate result set that is not already partitioned by the join vertex is repartitioned by the join vertex. After the repartitioning, a co-located join is performed.

If none of the intermediate result sets being joined are partitioned by the join vertex, then either a distributed hash join or a broadcast join is used. The distributed hash join repartitions all join inputs by the join vertex, and then does a co-located join. Therefore the partitioning vertex of the intermediate result set produced by this join is the join vertex. The broadcast join replicates the smaller intermediate result set in its entirety to each location holding a partition of the larger intermediate result set. A join is then performed between each partition of the larger intermediate result and the entire smaller

intermediate result set. Therefore the partitioning vertex of the output is the same as the partitioning vertex of the larger input intermediate result set.

III. Dynamic Programming-Based Optimization

Query optimization in traditional database systems can be seen as a search problem, which consists of three parts, namely, search space generation, cost estimation and search. The query optimizer defines a space of query plans, explores the space with a search algorithm and estimates the cost of plans encountered. The problem of subgraph pattern matching can be thought of in the same way, where the subgraph pattern being matched is the "query" that is processed over the raw graph data. As described in the previous section, these queries can be performed via matching fragments of the subgraph, and then joining these fragments together. Therefore, the optimization problem for subgraph pattern matching can be posed in terms of join ordering optimization (where the number of joins is equal to the number of vertices in the query subgraph), and traditional join ordering optimization techniques can be applied.

In general, given n relations (vertices), there are $((2n-2)!)/((n-1)!)$ different but logically equivalent join orders. If each join has several different implementation options (such as the the four join algorithms described Section II-B), then the plan space further explodes. As a result, many of the research prototypes and commercial implementations of traditional relational database systems limit the search space to a restricted subset [8] and bear the risk of missing the optimal plan.

In this section, we show how System-R style dynamic programming can be applied to subgraph pattern matching queries over distributed graph data while keeping the search space managably small.

A. Search Algorithm

Figure 1 (which calls code defined in Figures 2, 3, 5 and 7) provides psuedocode for our `FindPlan` algorithm, which, given an input query (or a query fragment), generates a plan space for processing the query, and uses a dynamic programming search algorithm to find the lowest cost execution plan for that query. Note that different execution plans will result in the output query result set being partitioned in different ways. In general, if a query (or query fragment) has n vertices, there are n different ways that the output could be partitioned. The output partitioning vertex is an important consideration if the input is a query fragment and the output will eventually be joined with a different query fragment. Therefore, instead of producing a single lowest cost plan for the entire query (fragment), it produces a lowest cost plan for each unique output partitioning vertex. The output of `FindPlan` is therefore an array of **triples** containing (1) the output partitioning vertices, (2) the cost of the lowest cost plan, and (3) the lowest cost plan. We now explain the algorithm in more detail.

In short, `FindPlan` enumerates all possible decompositions of the query graph into two or more fragments, and computes the costs of joining each pair of fragments. Since the same fragment is visited by many branches of the recursive algorithm, repetitive computation is avoided by mapping every possible query fragment to an initially empty triples array that is populated the first time `FindPlan` is called for that fragment. Therefore, the first line of the `FindPlan` pseudocode first checks to see whether the triples array associated with this fragment is non-empty. If so, then `FindPlan` has already been called for this fragment and need not be re-executed.

For any query that is repeatedly decomposed into smaller and smaller fragments, eventually the fragments will become as small as a single edge. This is the base case of the recursive `FindPlan` algorithm. If the fragment has one edge, we simply calculate the cost of matching the edge (which is a scan of all edges, E, of the data graph), and return from the function. However, we also add a second base case: if all edges in a query fragment all originate from the same vertex, then an n-way co-located join is clearly the optimal way to process this fragment, and no further composition is necessary.

If the if-statements in the pseudocode corresponding to these base cases all fail, then the query will be decomposed into fragments by `LinearDecomposition` and `BushyDecomposition`. We then recursively call `FindPlan` for each of these fragments. Once the query plans for these fragments have been calculated, the lowest cost plans for joining them back together is calculated by the `GenerateLinearPlans` and `GenerateBushyPlans` functions respectively. After these plans are generated, the non-promising ones are discarded by the `EliminateNonMinCosts` function. For instance, suppose we have three triples, that is t_1=(A, 100, p_1), t_2=(A, 200, p_2) and t_3=(B, 400, p_3). Although t_2 has a lower cost than t_3, it will be removed because its cost is higher than that of t_1 and hence will not be used by any future join. Therefore, the output of `FindPlan` is just [(A, 100, p_1), (B, 400, p_3)].

We now discuss linear and bushy plan generation.

B. Linear Plans

There are two steps to generate a linear plan for a query. First the query is decomposed into two fragments, at least one of which is not decomposable, and then a feasible join method is used to join the fragments. These two steps are done by `LinearDecomposition` and `GenerateLinearPlans` in Figure 2, respectively. As explained above, a fragment with all edges sharing the same originating vertex is considered non-decomposable.

Example 1. *Given query A→B, A→C, B→D, B→A, C→E, edges are grouped into three fragments based on the originating vertices of the edges: Q_1: A→B,A→C. Q_2: B→D,B→A. Q_3: C→E. Three decompositions are generated as follows:*

1:$\{Q_1\}$,$\{Q_2,Q_3\}$ *2:*$\{Q_2\}$,$\{Q_1,Q_3\}$ *3:*$\{Q_3\}$,$\{Q_1,Q_2\}$

In decomposition 1, it (1) joins the two edges in Q_1 , (2) joins the two edges in Q_2, (3) joins the result of (2) with Q_3, and finally joins the results of (1) and (3). Technically speaking, it is not a linear plan. However, the joins in steps (1) and (2) are done via co-located joins, which are so much cheaper than the distributed joins that they are considered atomic (non-decomposable) entities, and the plan is linear with respect to all non-local joins.

```
function FindPlan(Q)
  if (Q.triples != null) // already generated plans for Q
    return
  if (Q has only one edge e = (v1, v2))
    Q.triples ={(v1, scan cost of E (all edges), "match e")}
    return
  if (all edges in Q share the same originating vertex v)
    Q.triples ={(v, co-l join cost, "co-located join of Q")}
    return
  T = ∅ // a set of triples
  LD =LinearDecomposition(Q)
  for each linear decompositon (q1, q2) in LD
    FindPlan(q1)
    FindPlan(q2)
    T = T ∪ GenerateLinearPlans(q1, q2)
  LDAGs = GenerateLayeredDAGs(Q)
  for each layered DAG d in LDAGs
    (q1, q2, ..., qn−1, qn) = BushyDecomposition(d)
    for i from 1 to n
      FindPlan(qi)
    T = T ∪ GenerateBushyPlans(q1, ..., qn)
  Q.triples = EliminateNonMinCosts(T)

// for each vertex, remove the plans with non-minimum costs
function EliminateNonMinCosts(Triples)
  for each unique partitioning vertex v in triples of Triples
    let cv be the min cost of all triples of v
    remove all triples (v, c, p) from Triples such that c > cv
  return Triples
```

Fig. 1. Dynamic programming search algorithm that produces the lowest cost execution plan for each unique output partitioning vertex of a given input query (or query fragment). Functions undefined here are defined in Figures 2, 3, 5 and 7.

After `FindPlan` calls `LinearDecomposition` to generate a linear decomposition, query plans are generated for each pair of fragments returned by `LinearDecomposition` (via recursive calls to `FindPlan` for each fragment). The plan for reassembling each pair of fragments (and the associated costs) is then performed by the `GenerateLinearPlans` function. This function generates every possible pairwise combination of plans — one from each fragment — and calculates the cost of joining these plans together for each different distributed join algorithm that can possibly be applied given the partitioning vertex of each fragment.

Finding linear plans has time complexity $\mathcal{O}(|V|^2 \cdot |E|!)$.

C. Bushy Plans

As mentioned above, the complete search space of query plans is extremely large and the cost of an exhaustive enumeration of all plans (including bushy plans) is prohibitive. However, limiting the search to just linear plans can result in significant network bandwidth bottlenecks for distributed joins. Therefore, in this section, we propose a heuristic that allows us to explore certain promising bushy plans by taking into account the characteristics of graph pattern matching queries in a cost-effective manner. The heuristic consists of the three steps: (1) transform the query graph into a special type of directed acyclic graphs (DAGs); (2) decompose the DAGs into several fragments at vertices with more than one incoming edge; (3) form bushy query plans by joining the fragments with bushy hash joins and broadcast joins.

```
function LinearDecomposition(Q = (V, E))
  D = ∅ // a set of decomposition
  group edges in E on the originating vertices into
    fragments q1, q2, ..., qn
  for each fragment q
    D = D ∪ {(q, Q − q)}
  return D

// Generate linear plans given a decomposition of q1 and q2
function GenerateLinearPlans(q1, q2)
  T = ∅ // a set of triples
  for each triple (v1, c1, p1) in q1
    for each triple (v2, c2, p2) in q2
      for each common vertex between q1 and q2
        for each feasible join method j from Section II-B
          C = c1 + c2+ cost of j
          P = "run p1; run p2; join them using j"
          // v is the partitioning vertex of j
          T = T ∪ {(v, C, P)}
  return T
```

Fig. 2. Pseudocode for producing linear plans.

1) Transformation: The motivation of the transformation is two-fold. Firstly, many query graphs contain cycles, which complicates the fragmentation of the query, so removing cycles by transforming the query into a DAG leads to more straightforward fragmentation. Secondly, since the bushy plan space is enormous, rewriting the query as a DAG helps restrict the search space, so that only a few promising candidates are explored.

The result of a transformation is a set of a special type of DAG, called a **layered DAG**. A graph is a layered DAG if the following two conditions are met: (1) it is a DAG; (2) each vertex v in the graph is assigned a layer ID l and edges emanating from v go to vertices with layer ID $l + 1$.

Figure 3 shows the major functions used in the transformation process, namely, `GenerateLayeredDAGs`, `GenerateLayeredDAG`, `GenerateComponent`, and `Merge`. `GenerateLayeredDAGs` generates, for every vertex in an input query graph, a layered DAG starting with that vertex. The code for generating a layered DAG starting with a given vertex, v, is found in `GenerateLayeredDAG`. It does this by finding all vertices and edges that are reachable from v without ever traversing an edge twice. This set of vertices and edges (a subset of the input query graph) is termed a "component" (and the code for generating a component is given in the `GenerateComponent` function). If the component generated that started from v is not equal to the entire input query graph, then an additional vertex (that was unreachable from v) is selected as a starting point for a second component. This process repeats until all vertices and edges from the original query graph are accounted for in at least one component. At this point, all components that were generated are merged together using the `Merge` function, and the result is a single layered DAG.

The `GenerateComponent` function itself also produces a layered DAG. The input vertex exists by itself in the zeroth layer of the DAG. All vertices reachable from it via a single edge comprise the first layer, and all vertices reachable from those vertices via a single edge comprise the second layer. This process continues, with the $i+1^{th}$ layer comprising of vertices

```
function GenerateLayeredDAGs(Q = (V, E))
    S = ∅ // a set of layered DAGs
    for each vertex v ∈ V
        S = S ∪ {GenerateLayeredDAG(Q, v)}
        unmark all edges in E
    return S

function GenerateLayeredDAG(Q = (V, E), v0)
    S = ∅ // a set of layered DAG components
    v = v0
    while (there are unmarked edges in E)
        S = S ∪ {GenerateComponent(Q, v)}
        v = the originating vertex of an unmarked edge in E
    return Merge(S)

function GenerateComponent(Q = (V, E), v0)
    C = ∅ // layered DAG component
    add v0 to layer 0 of C
    l = 0 // current layer ID
    while (there are unmarked edges in E)
        for each vertex v in layer l
            for each unmarked edge e = (v, v') ∈ E
                add e to layer l of C and mark e
                add v' to layer (l + 1) of C
        if (no edges are added to layer l)
            break while
        else
            l++
    return C

// Merge a set of layered DAG components into one
function Merge(S: a set of layered DAG components)
    while (there are more than one component in S)
        find two components C1, C2 with a vertex v in common
        let i and j be the layer IDs of v in C1 and C2
        for k from 0 to max(maxLayer(C1)),maxLayer(C2))
            merge layer k from C1 and layer (k + j − i) from C2
        let the result of merging be C3
        add C3 to S and remove C1 and C2 from S
    return the only component left in S
```

Fig. 3. Transformation of a query graph into layered directed acyclic graphs.

that are reachable via an edge from any vertex in the i^{th} layer, as long as the edge connecting them had not already been traversed in a previous layer. Along with the vertices, each layer contains the set of edges that were traversed to reach the vertices in the succeeding layer. Although each edge can only exist once in the entire layered DAG, vertices can exist more than once. For example, if the vertices A and B are connected via one edge from A to B and another edge from B to A, the layered DAG that is generated starting from A is A→B→A, where A appears twice (in the zeroth and second layers). To distinguish multiple instances of a vertex in a component, we also use the layer ID as a subscript to the vertex ID to identify a vertex. Therefore, the previously mentioned layered DAG is represented as: $A_0 \rightarrow B_1 \rightarrow A_2$. All instances of a vertex found in the same layer are merged into a single version of that vertex.

As mentioned above, `GenerateLayeredDAG` continues to generate components until all vertices and edges are included in at least one component. At this point the `Merge` function is called to merge all components. Two components are merged by finding a vertex that is shared between them and

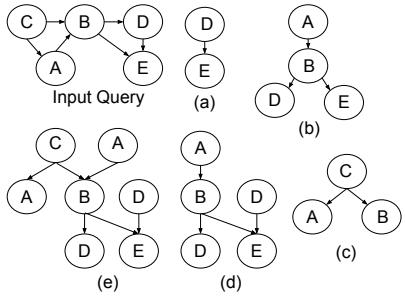

Fig. 4. Graphs for Example 2.

connecting the two components at this vertex. For instance, suppose two components C_1 and C_2 have vertices B_1 (i.e. vertex B in layer 1) and B_3 (i.e. vertex B in layer 3), respectively. Since B_1 and B_3 correspond to the same vertex B in the query graph, two components can be merged on B_1 and B_3. The merging of layers works as follows: merge layer 0 from C_1 and layer 2 from C_2, merge layer 1 from C_1 and layer 3 from C_2, and so forth. If a layer has no corresponding layer to merge, it becomes a layer by itself in the new merged component (e.g. layers 0 and 1 from C_2) and keeps its relative position to other merged layers.

Example 2. *This example shows what happens if* `GenerateLayeredDAG` *is called on the sample input query from Figure 4 with starting vertex D. Only one vertex is reachable from D (E), so the first component, shown in Figure 4(a), consists of just D → E. Assume A is picked as the next starting vertex. This results in the component shown in Figure 4(b) being produced. Note that the D → E edge had been used in Figure 4(a) and hence cannot be added to Figure 4(b). C is the only vertex left with edges emanating from it, so it is chosen as starting vertex for the next component, and the component shown in Figure 4(c) is generated. At this point, all edges are accounted for in one of the three components, and the next step is merging. The components from Figure 4(a) and Figure 4(b) are merged on vertex E (though D could have been used as an alternative) and component shown in Figure 4(d) is generated. Components from Figure 4(c) and Figure 4(d) are merged on vertex B (though A could have also been used), and the final layered DAG shown in Figure 4(e) is generated.*

Note that although `GenerateLayeredDAGs` produces a layered DAG for each vertex in the graph, it is possible some of these layered DAGs are identical. For example, a simple path query A→B→C of length 2 will result in the same exact layered DAGs for all three starting vertices.

The important end result of the transformation step is that all layered DAGs that are produced are logically equivalent to the input query graph. If the DAG is searched for over the raw data set, any results that match the DAG will also match the original query. Therefore the transformation step can be thought of as corresponding to the query rewrite step of traditional database optimizers.

2) Bushy Decomposition: Once layered DAGs have been produced by the transformation step, they can then be decomposed into fragments that can be planned for and executed independently, and then joined together. Vertices with several incoming edges are natural decomposition points, where one fragment is produced for the branch of the DAG corresponding to each incoming edge and one additional fragment for the

```
function BushyDecomposition(G = (V, E))
    current layer = bottom layer
    D = ∅ // a bushy decomposition
    while (current layer ≠ top layer)
        for each vertex v in current layer
            if (v has more than 1 incoming edge)
                if (Decomposable(G, v, D))
                    break while
        current layer = one layer above current layer
    return D // D is modified in Decomposable

// given a vertex v as the decomposition vertex, decide
// whether the input layered DAG is decomposable
function Decomposable(G = (V, E), v, D)
    // add all edges below v as one fragment to D
    D = {ReachableDAG(G, {v}, null)}
    // for each incoming edge of v, add one fragment to D
    for each (v', v) ∈ E
        V_rrv = ReversiblyReachableVertices(G, v')
        D = D ∪ {ReachableDAG(G, V_rrv, v)}
    if (fragments in D are pairwise disjoint in terms of edges)
        return true
    else
        D = ∅ // reset D to ∅ due to the decomposition failure
        return false

// find vertices that are reversibly reachable given a vertex;
// vertex v' is reversibly reachable from v if there is a
// directed path from v' to v
function ReversiblyReachableVertices(G = (V, E), v)
    V_all = ∅ // all reversibly reachable vertices (RRVs) found
    V_new = ∅ // newly found RRVs
    V_cur = {v} // current RRVs
    while (V_cur ≠ ∅)
        V_new = ∅ // reset new RRVs
        for each vertex v_1 ∈ V_cur
            for each edge (v_2, v_1) ∈ E
                V_new = V_new ∪ {v_2}
        V_all = V_all ∪ V_cur, V_cur = V_new
    return V_all

// find all reachable edges given a set of vertices
function ReachableDAG(G = (V, E), S_v, v_s)
    S_e = ∅ // a set of edges
    for each vertex v ∈ S_v
        for each e = (v_1, v_2) in E
            if there exists path from v to e without traversing v_s
                S_e = S_e ∪ {(v_1, v_2)}
    return S_e
```

Fig. 5. Functions for producing a bushy decomposition of the layered DAG generated from Figure 4.

descendants of this vertex.

Figure 5 shows the pseudocode for the above described bushy decomposition. The procedure starts from the bottom layer of the DAG and attempts to find vertices with more than one incoming edge. If such a vertex v is found, the `Decomposable` function is called that checks to see if it is possible to decompose the DAG into disjoint fragments around v, and if so, returns this set of fragments as the decomposition. The `Decomposable` function first calls `ReachableDAG` which creates a fragment consisting of all descendants of v (which is empty if v is in the bottom layer). Then for each incoming edge, it calls the `ReversiblyReachableVertices` function to find all

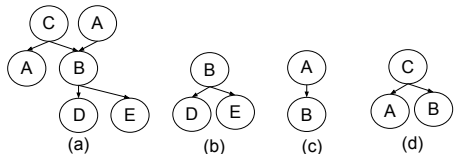

Fig. 6. Graphs for Example 3

vertices that are reversibly reachable from v starting with that edge, and then creates a fragment that contains all descendants of all of these vertices that are reachable without traversing through v (again through a call to `ReachableDAG`). Depending on how the ancestors of v overlap, these fragments either overlap or they are pairwise disjoint. If they overlap, the `Decomposable` function returns false. Otherwise, it returns true, along with the set of disjoint fragments it generated.

Example 3. *Assume the input DAG shown in Figure 6(a). The algorithm starts from the bottom layer and moves up. Vertex B is the first vertex that is encountered that has two incoming edges and therefore we decompose around it. First, we produce the fragment containing all of B's descendants, shown in Figure 6(b). Then, we produce one fragment for each incoming edge. For the edge coming from C, we find all reversibly reachable vertices (which is just C in this case), and then find all descendants from C, which leads to the DAG shown in Figure 6(d). For the edge coming from A, the fragment simpler, containing just one edge, as shown in Figure 6(c).*

3) Generating Bushy Plans: Recall from Figure 1 that after a bushy decomposition is produced, `FindPlan` generates the plans for each fragment in the decomposition and then calls `GenerateBushyPlans` to plan how the decomposed fragments should be combined. The code for `GenerateBushyPlans` is presented in Figure 7.

The function iterates through each combination of taking one triple from each input fragment, and produces a plan for performing the join (on a common vertex v_j) of the intermediate results specified by these triples. A directed join can be used if the partitioning vertex of at least one of the triples is the same as the join vertex. Otherwise a hash join must be used.

Example 4. *Suppose a bushy decomposition consists of three fragments, namely, q_1, q_2 and q_3, which have the following plan triples:*

q_1: triples t_1=(A, 100, p_1) and t_2=(B, 200, p_2).

q_2: triples t_3=(A, 300, p_3) and t_4=(B, 400, p_4).

q_3: triples t_5=(A, 500, p_5) and t_6=(B, 600, p_6).

There are $2^3 = 8$ different ways of choosing one triple from each fragment. One such way is t_1, t_4 and t_5. If A is the join vertex, then a directed join can be used, and the plan would look like the following: "execute p_1; execute p_4; partition p_4 on A; execute p_5; join on A;". The cost of the plan is 100 + 400 + 500 + (cost of the join) + (network cost of partitioning p_4).

Finding bushy plans has time complexity $\mathcal{O}(|V|^2 \cdot |E| \cdot |E|!)$.

```
function GenerateBushyPlans(q_1, ..., q_n)
  T = ∅ // a set of triples
  for each triple (v_1, c_1, p_1) in q_1
    .....
      for each triple (v_n, c_n, p_n) in q_n
        for each common vertex v_j of q_1, ..., q_n
          cost = cost of joining q_1, ..., q_n on v_j locally
          plan = empty
          for i from 1 to n
            cost += c_i
            plan += "execute p_i;"
            if (v_i ≠ v_j)
              cost += network cost of partitioning p_i
              plan += "partition p_i on v_j;"
          plan += "join on v_j;"
          T = T ∪ {(v_j, cost, plan)}
  return T
```

Fig. 7. Pseudocode for planning the join of fragments produced by bushy decomposition

IV. Cycle-Based Optimization

In this section, we present another optimization framework which is based on cycle detection. The dynamic programming-based optimization framework presented in the previous section always groups edges originating from the same vertex together because of its preference to do co-located joins whenever possible. However, if the graph contains cycles, it may be advantageous to match cycles first (even before co-located joins). This is because matching a cycle serves as a type of selection predicate — it restricts the subset of the raw data set that needs to be explored. Therefore, matching cycle patterns first may lead to smaller intermediate result sets and consequently yield better performance. For example, given a query A→B, A→C, B→A, a cycle-based algorithm would match A→B, B→A first, while the algorithms presented in the previous section would match A→B, A→C first. However, if a vertex in the raw data set has 20 outgoing edges, then there are 190 matches to the A→B, A→C join. However, if only 2 of the 10 outgoing edges from that vertex have a return edge in the other direction, then joining A→B, B→A first significantly reduces the intermediate result set. Note that both directed and undirected cycles are able to reduce the size of intermediate result sets.

Example 5. *Pattern A→B, B→C, C→A is a directed cycle. Pattern A→B, A→C, B→C is an undirected cycle.*

Figure 8 shows the pseudocode for the cycle detection-based optimization framework. It first converts the graph into an undirected graph and finds all cycles. After cycles are identified, two approaches are employed to combine them. The first approach is a greedy one. It chooses a cycle as a starting point and then keeps adding overlapping cycles to it greedily. Overlapping cycles are two cycles which share at least one edge (see Example 6). If overlapping cycles are not found, non-overlapping cycles are added instead. Overlapping cycles take precedence over non-overlapping cycles since it is more efficient to add overlapping cycles than non-overlapping ones (because part of the overlapping cycles have already been matched).

The second approach is a bushy approach. Similar to the formation of bushy plans described in the previous section, the algorithm decomposes the graph into several fragments,

```
function CycleDetectionOptimization(Q)
convert directed query graph Q into an undirected graph
find all cycles with a depth-first search on Q
// greedy approach
for every cycle c found
  current_cycle = c
  matched = ∅ // the fragment that has been matched
  while (current_cycle ≠ null)
    add current_cycle to matched and compute the cost
    if (there is a cycle c_o overlapping with matched)
      current_cycle = c_o
    else if (there is a cycle c_l left)
      current_cycle = c_l
    else
      current_cycle = null
  // for edges that do not belong to any cycles
  if (there are any edges left)
    add them to matched and compute the cost
// bushy approach
decompose Q into fragments such that overlapping
  cycles stay in the same fragment
for each decomposition D
  compute the cost of matching each fragment in D
  compute the costs of joining them

  return the plan with the lowest cost in both approaches
```

Fig. 8. Query optimization with cycle detection.

finds matches in each fragment separately, and then joins the intermediate results associated with these fragments. A heuristic is used for decomposition in which overlapping cycles stay in the same component. The rationale for this heuristic is that overlapping cycles tend to produce smaller intermediate output than non-overlapping cycles, so it is beneficial to group the overlapping cycles together. Example 7 illustrates how the cycle detection-based optimization works.

Example 6. *We use Q3, Q4 and Q5 in Figure 11 to illustrate the concept of (non-)overlapping cycles. In Q3, cycles ABC and DE do not overlap because they don't share any edges. In Q4, cycles AB and CBD do not overlap. They share vertex B, but do not share any edges. In Q5, cycles BD and BDE overlap because they share edge BD.*

Example 7. *There are three cycles in Q4 of Figure 11, namely, AB, BCD and BCE. Cycles BCD and BCE overlap by sharing edge BC. For the greedy approach, if it first matches BCD, it next adds edges B→E and C→E because BCE and BCE overlap. Finally, it adds edges A→B and B→A. For the bushy approach, the pattern is decomposed into two fragments. The first one contains cycle AB and the second has cycles BCD and BCE. Matching is done on the two fragments separately and then the results are joined.*

Greedy and bushy cycle-detection optimization have time complexity $\mathcal{O}(|V|^2 \cdot C^2)$ and $\mathcal{O}(|V|^2 \cdot C!)$, respectively, where C is the number of cycles in the query.

V. Cost Estimation

The cost of a join depends on the size and other statistics of its inputs. Given a join $q_1 \bowtie q_2$, we estimate its cost by summing up four parts, namely, (1) the cost of q_1, (2) the

Join Method	Condition	Partitioning Vertex	Network Cost
Collocated Join	A=B	A	0
Distributed Hash Join	All	C	$\|q_1\| + \|q_2\|$
Directed Join	B=C	C	$\|q_1\|$
	A=C	C	$\|q_2\|$
Broadcast Join	All	A	$\|q_2\| \times n$
	All	B	$\|q_1\| \times n$

Fig. 9. Join methods between two subqueries q_1 and q_2 on vertex C. Output of q_1 and q_2 are partitioned on vertices A and B, respectively. $|q_1|$ denotes the size of output of q_1. n is the number of machines involved.

Q1: A→B, C→B	
Step	Action
(1)	Match $V_1 \rightarrow V_2$.Partition on V_2
(2)	Self join (1) on V_2

Q2: B→A, C→B, D→B	
Step	Action
(1)	Match $V_1 \rightarrow V_2$
(2)	One copy of (1) stays put
(3)	Partition second copy of (1) on V_2
(4)	Perform self-join of (3) on V_2
(5)	Join V_1 from (2) with V_2 from (4)

Fig. 10. Query plans with computation reusing.

cost of q_2, (3) the network cost of moving the intermediate results of q_1 and/or q_2 and (4) the cost of performing the join locally after the network transport. (1) and (2) are computed recursively and (4) is a standard cost estimation in the DBMS. Figure 9 gives the network costs of the four different joins used in our system.

Traditional estimation techniques do not work well on graphs with cycles because it is hard to predict how much the presence of a cycle reduces the intermediate output size. To obtain more accurate estimates, we precompute and maintain in the catalog the size of all directed/undirected cycles up to three edges in the dataset, because cycles of size two and three appear frequently in input queries. Standard selectivity estimation techniques are used if there are predicates on attribtues within these cycles that further reduce selectivity.

VI. Computation Reusing

We will introduce the concept of computation reusing with several intuitive examples on structural pattern matching. Given a simple query, A→B, C→B, one of the possible plans is to first execute A→B and C→B separately and then perform a distributed hash join on B. But one should note that matching A→B and C→B are the same exact match (despite the different variable names). They are both simply selecting all edges in the graph where the originating vertex is different from the destination vertex, and can be represented as $V_1 \rightarrow V_2$. Consequently, the two input relations of the hash join on each machine are identical. Therefore, instead of generating the same intermediate result set twice, we can generate it just once, and do a self-join. The resulting query plan is shown as Q1 in Figure 10.

Another example query could be: B→A, C→B, D→B. Similar to above, we can get the set of all edges $V_1 \rightarrow V_2$. In this case, we use 3 copies of the intermediate result. One copy stays put (since it is already partitioned by V_1), and the other two copies are repartitioned by V_2. At this point, the three sets of intermediate results: $V_1 \rightarrow V_2$, $V_2 \rightarrow V_1$, and $V_2 \rightarrow V_1$ can all be joined (locally on each node) on the partitioning vertex of each intermediate result set. The query plan is shown as Q2 in Figure 10. Note that as an optimization, we do a self-join of $V_2 \rightarrow V_1$ instead of making a third copy.

As long as multiple fragments produce the same intermediate result sets **independent** of the data sets, fragment execution is reusable. Computation reusing mostly occurs in structural pattern matching and relatively rarely in semantic pattern matching since the matching of attributes prior to the generation of intermediate results sets makes them less reusable.

VII. Experimental Evaluation

In this section, we measure the performance of our optimization algorithms on four graph data sets of varying nature and size. The code and queries used in this evaluation are available at http://db.cs.yale.edu/icde2014.

A. Experimental Setup

1) Experimental Environment and System Setup: Experiments were conducted on a cluster of machines. Each machine has a single 2.40 GHz Intel Core 2 Duo processor running 64-bit Red Hat Enterprise Linux 5 (kernel version 2.6.18) with 4GB RAM and two 250GB SATA-I hard disks. According to hdparm, the hard disks deliver 74MB/sec for buffered reads. All machines are on the same rack, connected via 1Gbps network to a Cisco Catalyst 3750E-48TD switch. Experiments were run on 10 machines, unless stated otherwise.

Our system was implemented in Java and Python. It utilizes PostgreSQL 9.2.1 on each machine as the query processing unit.

2) Graph Data sets: Our experiments benchmark performance of subgraph pattern matching on four publicly available (real-world) graph data sets: the Amazon product co-purchasing graph, the Youtube video co-watching graph, the Google Web graph, and the Twitter follower graph.

Wherever possible, the sub-graphs that we search for within these data sets are extracted from previous research papers published by other groups that use these same data sets to study subgraph pattern matching. In particular, the Amazon data set was experimented with in [24] and [25], the Youtube video data set was experimented with in [24], and the Web graph data set was experimented with in [25]. The following is a detailed description of the data sets.

Amazon product graph is a co-purchasing directed graph of Amazon products with 548,552 vertices and 1,788,725 edges. It was collected from the "Customers Who Bought This Item Also Bought" section of Amazon's Website in the summer of 2006. An edge from product A to B means some customers bought B after A. Products were assigned zero to multiple categories (such as "Health, Mind and Body" and "Medicine"), which are treated as attributes.

Youtube video graph is a directed graph gathered from Youtube in 2007 with 155,513 vertices and 3,110,120 edges. An edge from video A to B means that viewers who watched A also watched B. Videos were assigned a type (such as "Music" or "Sports"), which were treated as attributes in our experiments.

Web graph is a Web graph in which nodes represent web pages and directed edges represent hyperlinks between them. It has 875,713 vertices and 5,105,039 edges and was released in 2002 by Google as a part of a Google Programming Contest. Vertices do not have attributes in this data set.

Twitter graph is a graph of Twitter follower information in which an edge from A to B means that A follows B on Twitter. It has 1,652,252 vertices and 1,468,365,182 edges and was collected in 2009. We assigned a random date to each edge intended to represent the date that the edge was created, and treated it as an edge attribute.

3) Pattern Queries: Our benchmark consists of "structural" and "semantic" pattern matching, where structural queries search purely for a particular pattern of connected vertices and edges (the subgraph does not use attributes on vertices or edges to reduce the scope of the search), and semantic queries include attributes on graph objects.

There are six structural queries in our benchmark, which consists of two queries from related research [24] and four more complicated ones which are more difficult to optimize. These six queries are shown in Figure 11.

For the semantic queries, we simply add attributes to the vertices in the structural queries. To cover different use cases, we varied the numbers of vertices with attributes. Some queries have attributes on each vertex in the subgraph, while others have only one attributed vertex.

In addition to the above mentioned subgraphs, we include an additional subgraph for the Twitter data set (see Figure 11). Previous work has shown that finding triangles (three vertices connected to each other) is a crucial step in measuring the clustering coefficient of nodes in a social network [33]. For Twitter, three accounts who follow each other are regarded as a triangle. Therefore, our benchmark includes a search for the triangle pattern in the Twitter data set. In particular, the query finds triangles which were established before the year 2009. Unlike the semantic queries mentioned above, the attributes in this query are on the edges of the graph. Therefore, we analyze the results of this query separately from the above queries.

4) Algorithms: We experimented with five query optimization algorithms. A **greedy** algorithm serves as the baseline, which, in each iteration, adds one edge with the lowest cost to the already matched subgraph until all edges are matched. The other four approaches are based on the algorithms presented in this paper. The **DP-linear** (DP is short for dynamic programming) algorithm considers only linear query plans in the dynamic programming framework, while the **DP-bushy** algorithm considers both bushy and linear plans and picks a plan with the lowest cost. The **cycle-greedy** algorithm adds the cycle detection optimization technique to the greedy approach, while **cycle-bushy** algorithm considers all approaches (leveraging the cycle detection optimization when possible) and picks a plan with the lowest cost. We ran each algorithm for each query five times and report the average.

B. Performance of Structural Matching

Figure 12 presents the execution time of each technique on the task of structural pattern matching (i.e. without attributes) for the Amazon products, Youtube videos and Web graph data sets (DP-bushy and/or cycle-greedy bars are omitted if they produce the same plan as DP-linear; cycle-bushy bars are omitted if they produce the same plan as cycle-greedy). Overall, cycle detection optimization algorithms fare the best and the greedy algorithm is outperformed by other algorithms by a large margin for most of the queries.

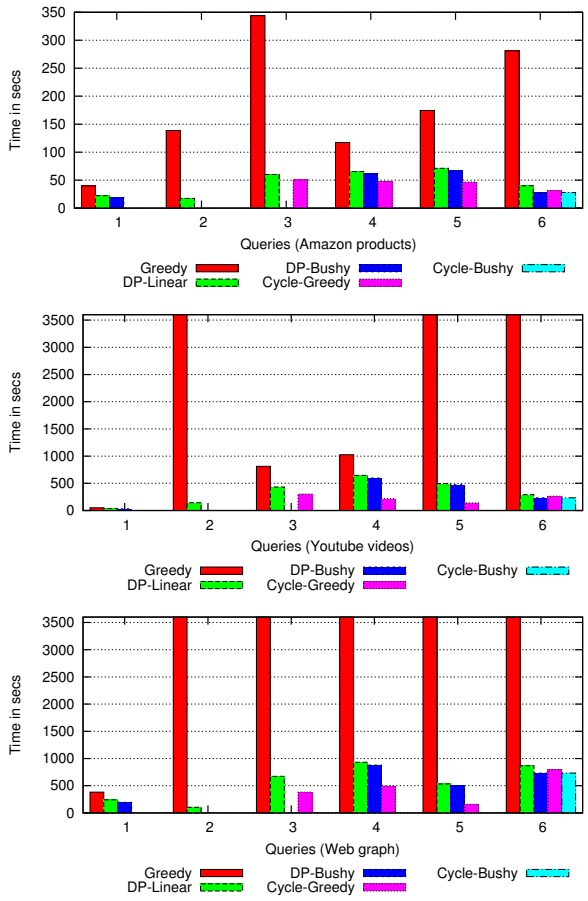

Fig. 12. Structural matching performance on three graph datasets. DP-bushy and/or cycle-greedy bars are omitted if they produce the same plan as DP-linear; cycle-bushy bars are omitted if it produces the same plan as cycle-greedy.

Greedy plans fail to finish within one hour on queries 2 and 5 for the Youtube dataset, and on queries 2, 3, 4, 5 and 6 for the Web graph. Its performance on the Amazon data set is not as bad because of its relatively smaller size.

To further analyze these results, we examine queries 2 and 4 in more detail. Query 2 is relatively simple compared to the other queries but the plan generated by the greedy algorithm fails to finish on two data sets. The reason is that after joining A→C and A→D, it next matches B→C and joins it to the existing intermediate result set. However, this join produces an enormous intermediate result set and dramatically increases the cost of the following join (with B→D). A better plan would involve joining A→C and A→D and then joining B→C and B→D. After these two joins have completed, the two sets of intermediate result sets are joined with each other[1]. By doing so, the step that produces a large intermediate result set is avoided. DP-Linear, DP-Bushy, and both cycle detection algorithms generate this superior plan.

Query 4 is perhaps more interesting than query 2, since the greedy, DP-linear, DP-bushy and cycle-greedy algorithms all produce different plans (cycle-greedy and cycle-bushy produce the same plan for this query). These plans are summarized in Figure 14. The greedy, DP-linear, and DP-bushy algorithms

[1]Recall from Section III-B that co-located joins are treated as atomic units, and therefore this plan is produced via the linear algorithm even though that it looks bushy.

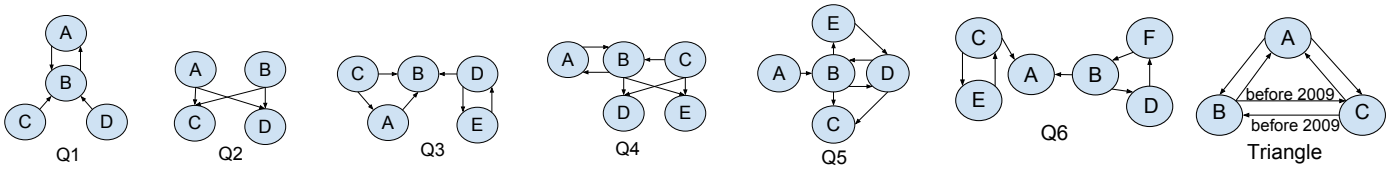

Fig. 11. Structural queries and triangle finding query. "Before year 2009" means that the following relation is established before year 2009. The year restriction is applied to all edges, however, only two are shown.

		DP-Bushy		Cycle-Bushy	
Data set	Queries	Time	Size	Time	Size
Amazon	Q3	59.95	7.8G	50.41	2.4G
	Q4	65.47	3.9G	47.68	0.9G
	Q5	71.23	5.8G	46.61	1.2G

Fig. 13. Execution time vs. intermediate result set size for the DP and Cycle plans. (Since the plans for queries 1, 2 and 6 are the same for both algorithms, they are omitted here.)

Greedy Plan	
Step	Action
(1)	Match A→B
(2)	Match B→A. Partition on A
(3)	Join (1) and (2)
(4)	Match B→D
(5)	Partition (3) on B
(6)	Join (4) and (5)
(7)	Match B→E
(8)	Co-located join (6) and (7)
(9)	Match C→B. Partition on B
(10)	Join (8) and (9)
(11)	Match C→D and broadcast
(12)	Join (10) and (11)
(13)	Match C→E and broadcast
(14)	Join (12) and (13)
DP-Linear Plan	
Step	Action
(1)	Co-located join: B→A,B→D,B→E
(2)	Match A→B. Partition on B
(3)	Join (1) and (2)
(4)	Co-located join: C→B,C→D,C→E. Partition on B
(5)	Join (3) and (4)

DP-Bushy Plan	
Step	Action
(1)	Co-located join: B→A,B→D,B→E
(2)	Match A→B. Partition on B
(3)	Collocated-join: C→B,C→D,C→E Partition on B
(4)	Join (1), (2) and (3)
Cycle-Greedy Plan	
Step	Action
(1)	Match A→B. Partition on B
(2)	Match B→A
(3)	Join (1) and (2)
(4)	Match B→D
(5)	Co-located join (3) and (4)
(6)	Match C→B. Partition on B
(7)	Join (5) and (6). Partition on C
(8)	Match C→D. Partition on C
(9)	Join (7) and (8)
(10)	Match C→D. Partition on C
(11)	Join (9) and (10). Partition on B
(12)	Match B→E. Partition on B
(13)	Join (11) and (12)

Fig. 14. Query plans of query 4 (without computation reusing)

make use of six, four, and three joins, respectively. The greedy algorithm, by adding one edge at a time, misses several opportunities for collocated joins, and therefore five out of the six joins it performs are either directed joins or broadcast joins. Both directed and broadcast joins involve sending an entire table (or a set of intermediate result set) across the network. The DP-Linear and DP-Bushy plans both perform two collocated joins, with DP-Linear performing two more directed joins and DP-Bushy doing one (the DP-Bushy algorithm does one three-way directed join while DP-Linear does two 2-way directed joins). This results in slightly better performance for the DP-Bushy plan.

However, looking only at the number of joins is not sufficient to explain the difference in performance of the plans. The plan generated by the cycle algorithms has six joins — the same number as the greedy plan. However the performance of its plan is significantly better than all alternatives. This is because it discovers several cycles in the query graph and matches these cycles first. The structural limitations that the cycles present serve as a sort of query predicate that limits the size of the intermediate result sets, thereby reducing network traffic and upstream join costs. Therefore, even though more joins are performed to match these cycles, the plan produced results in lower cost and better performance.

To highlight the performance difference between dynamic programing-based and cycle detection-based optimization algorithms, we list the execution time and the size of intermediate result sets for bushy and cycle-bushy plans on the Amazon data set in Figure 13 for the three queries for which these algorithms produce difference plans. While it is clear that the cycle plans reduce the intermediate result set by factors of 3-5X, the performance in improvement is not quite commensurate with this. This is due to the bottleneck shifting from data movement to PostgreSQL join performance. Many read-optimized commercial analytical databases that build on top of PostgreSQL modify PostgreSQL's poorly optimized join algorithms. We would expect to see larger performance improvements for the cycle plans under more optimized settings.

C. Performance of Semantic Matching

Figure 15 presents the performance of each algorithm on semantic pattern matching (i.e. with selections on attributes of vertexes) for the Amazon and Youtube data sets. The particular predicates we used can be found at: http://db.cs.yale.edu/icde2014.

The key difference between this set of experiments and the structural matching experiments is that the relative performance of the dynamic programming-based algorithms and the cycle detection-based algorithms are reversed. This is because the key advantage of the cycle detection algorithms is that cycles serve as a sort of selection predicate, the most effective way for reducing the scope of structural matching (and thereby reducing the size of the intermediate data). However, semantic matching allows for other selection predicates (in these experiments the selection predicates were on the vertices). These selection predicates do much more for reducing the scope of the search than cycle detection. The table below shows the sizes of the intermediate result sets for queries 3 and 4 on the Amazon data set. With the main benefit of the cycle detection algorithms reduced, its disadvantage in terms of the number and types of joins is more evident, and the dynamic-programming based algorithms which optimize for reducing the number of joins and ensuring that they are as local as possible outperform the cycle plans.

Data set	Query	DP-Bushy	Cycle-Bushy
Amazon	Q3	164M	223M
	Q4	53M	89M

D. Performance of Triangle Finding

The table below presents the execution time (in seconds) of each technique on the task of triangle finding in the Twitter

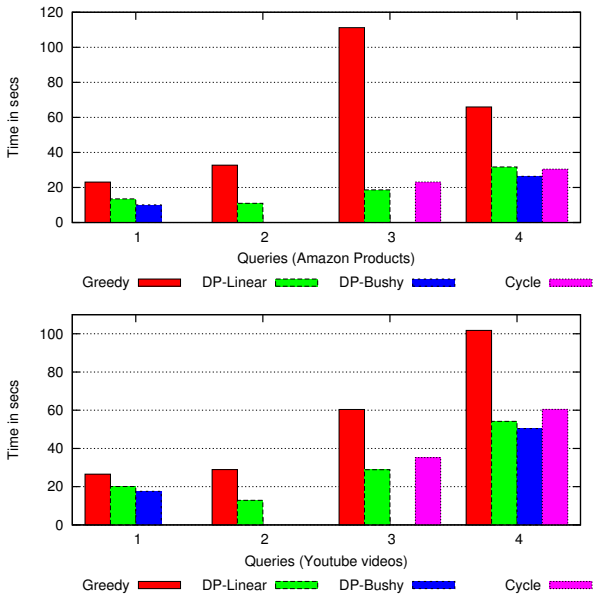

Fig. 15. Semantic matching performance on two graph data sets. DP-bushy and/or cycle bars are omitted if they produce the same plan as DP-linear.

DP-bushy Plan	
Step	Action
(1)	Co-located join : A→B,A→C
(2)	Co-located join : B→A,B→C. Partition on A
(3)	Co-located join : C→A,C→B. Partition on A
(4)	Join (1), (2) and (3)

Cycle-bushy Plan	
Step	Action
(1)	Match A→B
(2)	Match B→A. Partition on A
(3)	Join (1) and (2)
(4)	Match A→C
(5)	Match C→A. Partition on A
(6)	Join (4) and (5)
(7)	Join (3) and (6). Partition on B
(8)	Match B→C
(9)	Match C→B. Partition on B
(10)	Join (8) and (9)
(11)	Join (7) and (10)

Fig. 16. Query plans of query 4 (without computation reusing)

social graph. Since the data set is large, the experiment was run on 50 machines. Cycle-G and Cycle-B stand for Cycle-Greedy and Cycle-Bushy algorithms, respectively. We find that the DP-Bushy plan performs the best. To understand why this is the case, we explore the DP-Bushy and Cycle-Bushy plans in more detail in Figure 16. Both plans decompose the query into three fragments. However, because the fragments from dynamic programming are done via co-located joins (without any network traffic), they take less time than those from cycle detection (note that steps (1), (2), and (3) are identical except for variable naming, and therefore can leverage the computation reusing technique). Meanwhile the advantage of cycle detection in terms of reducing the scope of the search is reduced due to the culture on Twitter of following people back who follow you.

	Greedy	DP-linear	DP-bushy	Cycle-G	Cycle-B
Time	375.35	114.72	93.89	129.72	107.51

E. Effects of Computation Reusing

To study the effects of computation reusing, we compare the query execution time (in seconds) on the Amazon and Twitter data sets with and without the computation reusing technique being leveraged. The results are shown in the following table. All queries benefit from computation reusing, although to different extents. The queries that improve the most are queries 1 and 2 on the Amazon data set and the triangle query on the Twitter data set. They have one or both of the following characteristics: (1) the total execution is short so that the saved computation accounts for a big portion of the execution; (2) the structure of the pattern is highly symmetrical so all/most of the fragments are identical and can be reused. (Query 2 has both of these characteristics.)

Data set	Query	w/o reusing	w/ reusing	ratio
Amazon	Q1	23.51	18.97	1.24
	Q2	21.26	17.28	1.23
	Q3	65.78	59.95	1.09
	Q4	73.14	65.47	1.11
	Q5	80.64	71.23	1.13
	Q6	45.29	39.79	1.14
Twitter	triangle	131.23	93.89	1.40

F. Summary of Experimental Findings

After performing our experimental evaluation, we come to the following conclusions: (1) Cycle detection-based algorithms tend to perform the best in structural matching, (2) dynamic programming-based algorithms tend to perform the best in semantic matching, and (3) computation reusing reduces the execution time for symmetrical queries, and should be used whenever possible.

However, these rules are only heuristics. The relative benefits of cycle detection are highly dependent on the graph data sets being queried. When cycles are rare, the cycle detection algorithm significantly reduces the scope of the search, but when cycles are common (or at least more common than other predicates available to the query) cycle detection is far less helpful. Therefore, in general, both DP and cycle-based plans should be generated and cost-estimated, and the cost-based optimizer should choose the plan with the lowest cost.

VIII. Related Work

There has been an increasing amount of interest in large-scale graph data processing. In this section we survey related work in the following three areas: graph pattern matching, distributed graph processing and query optimization in relational databases.

Graph pattern matching has been studied extensively [9], [10], [13], [16], [34], [35], [39]. Three surveys summarize the recent progress on this topic [12], [14], [31]. Graph pattern matching is particularly well-explored in the context of the Resource Description Framework (RDF) [28], [22] and [5]. The majority of the work in this area assumes that the whole graph dataset can fit into the memory of a single machine, and proposes efficient centralized pattern matching algorithms. Most of these algorithms can be applied to the matching of subqueries on individual worker machines in our system.

Several recent papers studied distributed graph pattern matching. [18] proposes a parallel RDF system in which vertices and edges within multiple hops were guaranteed to be stored on the same machine, in order to minimize network traffic. However, query optimization is not considered. [24] and [25] focus on pattern matching in the context of graph simulation. [32] and [38] are built on top of Trinity [7] and use graph exploration in query optimization for large vertex-labeled undirected graphs and RDF graphs, respectively. The optimization techniques described in this paper can be

added to the above-mentioned systems to further improve their performance.

In the wave of big data, many large-scale graph data management systems were proposed to satisfy the needs of different applications, such as Pregel [26] (and its open-source version Giraph [2]), Neo4j [4], Trinity [7], Pegasus [20], GraphBase [19] and GraphLab [23]. Pregel programs are written in Bulk Synchronous Parallel model [37] and are suitable for iterative algorithms like PageRank [6] and finding shortest paths. Neo4j is a centralized transactional graph database. Pegasus and GraphLab are best suited for graph mining and machine leaning. Mondal et. al. [27] present a horizontally-partitioned system managing dynamic graphs. However, none of these systems introduce sophisticated query optimization algorithms for graph pattern matching operations.

Our work is closely related to traditional database system query optimization, particularly in parallel databases. The use of dynamic programming as a search strategy dates back to the System R project [30]. Chaudhuri gives a survey [8] on various aspects of query optimization in centralized systems. Query optimization in parallel database systems has also been well studied. The Papyrus [11] project employed a formulation [15] minimizing response time subject to constraints on throughput. Lanzelotte et. al. [21] maintained that bushy search space should be included in order not to miss the optimal plan and advocated a simulated annealing search strategy. The XPRS [17] project, aiming at shared memory systems, proposed a two-phase approach. The Gamma project [29] studied several execution strategies but did focus on query optimization. Each of these papers targeted SQL queries over relational data. The key contribution of our work is that we introduce an approach that exploits the particular characteristics of graph data and graph pattern matching queries.

IX. Conclusions

In this paper, we presented optimization techniques for distributed graph pattern matching. We proposed two optimization frameworks that are based on dynamic programming and cycle detection, respectively. These frameworks explore greedy, linear and bushy plans. In addition, we proposed a computation reusing technique that eliminates redundant subquery pattern matching and reduces network traffic. With the proposed techniques, our system is able to outperform greedy query optimization by orders of magnitude.

Acknowledgment

This work was sponsored by the NSF under grant IIS-0845643 and by a Sloan Research Fellowship

References

[1] http://www.statisticbrain.com/facebook-statistics.

[2] Apache Giraph. http://giraph.apache.org/, 2012.

[3] Linking Open Data. http://stats.lod2.eu/, 2012.

[4] Neo4j. http://neo4j.org/, 2012.

[5] M. Arenas, S. Conca, and J. Pérez. Counting beyond a yottabyte, or how sparql 1.1 property paths will prevent adoption of the standard. In *WWW*, 2012.

[6] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. WWW7, pages 107–117, 1998.

[7] B.Shao, H.Wang, and Y.Li. The Trinity graph engine. Technical report, Microsoft Research, 2012.

[8] S. Chaudhuri. An overview of query optimization in relational systems. PODS '98, pages 34–43, 1998.

[9] J. Cheng, J. X. Yu, B. Ding, P. S. Yu, and H. Wang. Fast graph pattern matching. In *ICDE*, pages 913–922, 2008.

[10] J. Cheng, J. X. Yu, and P. S. Yu. Graph pattern matching: A join/semijoin approach. *IEEE TKDE.*, July 2011.

[11] T. Connors, W. Hasan, C. Kolovson, M.-A. Neimat, D. Schneider, and K. Wilkinson. The papyrus integrated data server. PDIS '91.

[12] W. Fan. Graph pattern matching revised for social network analysis. In *ICDT*, pages 8–21, 2012.

[13] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu. Graph pattern matching: From intractable to polynomial time. *PVLDB*, 2010.

[14] B. Gallagher. Matching structure and semantics: A survey on graph-based pattern matching. In *AAAI FS*, pages 45–53, 2006.

[15] S. Ganguly, W. Hasan, and R. Krishnamurthy. Query optimization for parallel execution. *SIGMOD Rec. 1992*.

[16] H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *SIGMOD'08*.

[17] W. Hong and M. Stonebraker. Optimization of parallel query execution plans in xprs. PDIS '91.

[18] J. Huang, D. J. Abadi, and K. Ren. Scalable sparql querying of large rdf graphs. *PVLDB*, 4(11):1123–1134, 2011.

[19] U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos. Gbase: a scalable and general graph management system. In *KDD'11*.

[20] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system. In *ICDM'09*.

[21] R. S. G. Lanzelotte, P. Valduriez, and M. Zaït. On the effectiveness of optimization search strategies for parallel execution spaces. In *VLDB*, pages 493–504, 1993.

[22] K. Losemann and W. Martens. The complexity of evaluating path expressions in sparql. In *PODS*, 2012.

[23] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *UAI*, July 2010.

[24] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo. Capturing topology in graph pattern matching. *PVLDB'11*.

[25] S. Ma, Y. Cao, J. Huai, and T. Wo. Distributed graph pattern matching. In *WWW*, pages 949–958, 2012.

[26] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD'10*.

[27] J. Mondal and A. Deshpande. Managing large dynamic graphs efficiently. SIGMOD '12, pages 145–156, 2012.

[28] T. Neumann and G. Weikum. The rdf-3x engine for scalable management of rdf data. *The VLDB Journal*, 19(1):91–113, 2010.

[29] D. A. Schneider and D. J. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. SIGMOD '89, pages 110–121.

[30] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *SIGMOD'79*, pages 23–34.

[31] D. Shasha, J. T.-L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *PODS'02*.

[32] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *PVLDB'12*.

[33] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. WWW '11, pages 607–614, 2011.

[34] Y. Tian and J. M. Patel. Tale: A tool for approximate large graph matching. In *ICDE*, pages 963–972, 2008.

[35] H. Tong, C. Faloutsos, B. Gallagher, and T. Eliassi-Rad. Fast best-effort pattern matching in large attributed graphs. In *KDD*, 2007.

[36] J. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)*, 23(1):31–42, 1976.

[37] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, Aug. 1990.

[38] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for web scale rdf data. In *VLDB*, 2013.

[39] L. Zou, L. Chen, and M. T. Özsu. Distancejoin: Pattern match query in a large graph database. *PVLDB*, 2(1):886–897, 2009.