2016 16th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing

GoDB: From Batch Processing to Distributed Querying over Property Graphs

Nitin Jamadagni[1] and Yogesh Simmhan[2]
[1] National Institute of Technology, Suratkal, India
[2] Department of Computational and Data Sciences, Indian Institute of Science, Bangalore, India
Email: nitin.jamadagni@gmail.com, simmhan@cds.iisc.ac.in

***Abstract*—Property Graphs with rich attributes over vertices and edges are becoming common. Querying and mining such linked Big Data is important for knowledge discovery and mining. Distributed graph platforms like Pregel focus on *batch execution* on commodity clusters. But exploratory analytics requires platforms that are both *responsive and scalable*. We propose *Graph-oriented Database (GoDB)*, a distributed graph database that supports declarative queries over large property graphs. GoDB builds upon our GoFFish subgraph-centric batch processing platform, leveraging its scalability while using execution heuristics to offer responsiveness. The GoDB declarative query model supports vertex, edge, path and reachability queries, and this is translated to a distributed execution plan on GoFFish. We also propose a novel cost model to choose a query plan that minimizes the execution latency. We evaluate GoDB deployed on the Azure IaaS Cloud, over real-world property graphs and for a diverse workload of 500 queries. These show that the cost model selects the optimal execution plan at least 80% of the time, and helps GoDB weakly scale with the graph size. A comparative study with Titan, a leading open-source graph database, shows that we complete all queries, each in ≤ 1.6 $secs$, while Titan cannot complete up to 42% of some query workloads.**

***Index Terms*—Graph Database, Property Graph, Query Optimization, Distributed Systems, Big Data**

I. Introduction

Fifteen years after Berners-Lee and Hendler proposed the Semantic Web [1], automated acquisition of knowledge graphs is becoming a reality in this Big Data landscape. Google's Freebase [1] and CMU's Never Ending Language Learning (NELL) project [2] leverage commercial and community efforts to capture concepts and relationships about the world. Further, domain-specific *property graphs* with rich metadata, such as citation and social networks, are common-place. Managing, mining, and querying such large graphs, with 10's of millions of vertices and edges and numerous attributes each, poses unique challenges to Big Data platforms.

Recent *distributed graph programming* models such as *Google's Pregel* [3] and *GraphLab* [4] have proposed vertex-centric or asynchronous execution models for graph processing on commodity clusters. However, such frameworks favor batch processing, and are better suited for graph algorithms like finding the centrality or shortest paths within *minutes*. Further, they do not emphasize the storage of and querying over complex properties of a graph's vertices and edges.

[1]Introducing the Knowledge Graph, http://goo.gl/2u3Vur

As such, there is inadequate work on distributed graph databases that offer the scalability of batch systems, and the query capabilities and short response time (*O(secs)*) required for interactive processing. There have been some efforts on research into scalable graph databases [5], [6] and open source graph stores like *Titan* [7]. There has also been focused work on semantic databases and triple stores that support SPARQL, inferencing and reasoning. However, these scale with the number of concurrent queries and less so with the complexity of the query or the number of results, or are specialized to RDF. For e.g., a common knowledge discovery query tries to find the reachability between two concepts (vertices), and this can require enumerating millions of paths between these vertices. Further, declarative queries that combine user-logic like PageRank along with attribute predicates are not easy to represent, if at all.

In this paper, we address this gap by extending our subgraph-centric distributed graph processing platform, *GoFFish* [8], with support for declarative graph querying that is both responsive and scalable. Our approach is similar to tuple-based NoSQL databases like Apache Hive or Pig, that compile a NoSQL query to MapReduce jobs, executed on a batch processing system. We also leverage ideas from relational database query optimization for distributed execution.

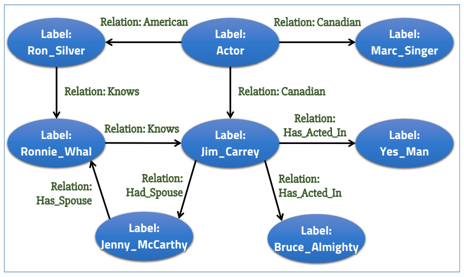
Specifically, we make the following contributions:

1) We define a *query model* for vertex/edge, path and reachability queries over property graphs (§ II), and propose an architecture for its distributed execution using a subgraph-centric Bulk Synchronous Parallel (BSP) model (§ III).
2) We develop a *heuristic cost model* to evaluate the costs of different distributed query execution plans using statistics and indexes, and use this to select the best execution plan at runtime (§ IV).
3) We implement and comparatively evaluate our distributed graph database, *GoDB*, for different query workloads, and highlight its scalability and the responsiveness due to the query plans and cost heuristics (§ V).

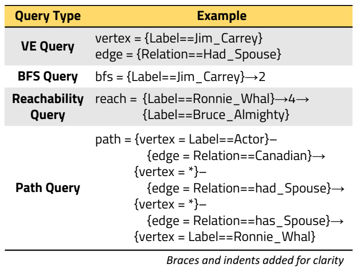
II. Property Graph Query Model

Property graphs (or attribute graphs) are multi-graphs that in addition to the connectivity between vertices using directed or undirected edges, also define name-value pairs associated with their vertices and edges [9]. Each vertex or edge may have one or more property names and their optionally-typed values.

978-1-5090-2453-7/16 $31.00 © 2016 IEEE
DOI 10.1109/CCGrid.2016.105



(a) Subset of the NELL Property Graph



Query Type	Example

VE Query	vertex = {Label==Jim_Carrey}
	edge = {Relation==Had_Spouse}

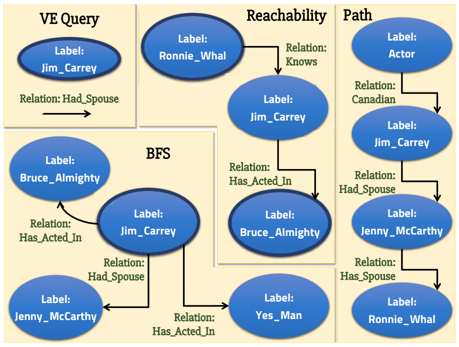
BFS Query	bfs = {Label==Jim_Carrey}→2

Reachability Query	reach = {Label==Ronnie_Whal}→4→
	{Label==Bruce_Almighty}

Path Query	path = {vertex = Label==Actor}– {edge = Relation==Canadian}→ {vertex = *}– {edge = Relation==had_Spouse}→ {vertex = *}– {edge = Relation==has_Spouse}→ {vertex = Label==Ronnie_Whal}

Braces and indents added for clarity

(b) Example query definitions



(c) Example query results

Figure 1. Sample NELL property graph, different query types defined over it and their results.

One of these properties typically refers to a unique identifier for that vertex or edge, while the others may describe features of that vertex or edge.

For example, in the *NELL knowledge graph* fragment [2] shown in Fig. 1a, each vertex has a `Label` associated with it denoting the unique name of that resource, while directed edges between vertices have several properties over them, such as the `Relation` between the vertices and the `Confidence` in the relationship that was learned from a web corpus (not shown). Similarly, in a *Patent Citation graph* [10], the vertices capture properties such as `granting_year` and `country`, while edges represent the citations between vertices.

Given such a property graph model, we define four types of queries that can be executed over it, many of which have been identified previously in the context of graph databases [11]. The grammar for these are given below.

$$
\begin{array}{lll}
\langle\textit{query}\rangle & ::= & \langle\textit{ve-query}\rangle \mid \langle\textit{path-query}\rangle \\
 & \mid & \langle\textit{bfs-query}\rangle \mid \langle\textit{reachability-query}\rangle \\
\langle\textit{ve-query}\rangle & ::= & \text{`}\texttt{vertex=}\text{'}\ \langle\textit{pred}\rangle \mid \text{`}\texttt{edge=}\text{'}\ \langle\textit{pred}\rangle \\
\langle\textit{path-query}\rangle & ::= & \text{`}\texttt{path=}\text{'}\ \langle\textit{ve-path}\rangle\text{+}\ \langle\textit{pred}\rangle \\
\langle\textit{bfs-query}\rangle & ::= & \text{`}\texttt{bfs=}\text{'}\ \langle\textit{pred}\rangle\ \text{`}\rightsquigarrow\text{'}\ \langle\textit{max-depth}\rangle \\
\langle\textit{reachability-query}\rangle & ::= & \text{`}\texttt{reach=}\text{'}\ \langle\textit{pred}\rangle\ \text{`}\rightsquigarrow\text{'}\ \langle\textit{max-depth}\rangle \\
 & & \text{`}\rightsquigarrow\text{'}\ \langle\textit{pred}\rangle \\
\langle\textit{ve-path}\rangle & ::= & \langle\textit{pred}\rangle\ \text{`}\vdash\text{'}\ \langle\textit{pred}\rangle\ \text{`}\rightarrow\text{'} \\
 & \mid & \langle\textit{pred}\rangle\ \text{`}\leftarrow\text{'}\ \langle\textit{pred}\rangle\ \text{`}\dashv\text{'} \\
\langle\textit{binary-pred}\rangle & ::= & \langle\textit{prop-name}\rangle\ \langle\textit{comparator}\rangle\ \langle\textit{prop-value}\rangle \\
 & \mid & \langle\textit{binary-pred}\rangle\ \text{`}\texttt{AND}\text{'}\ \langle\textit{binary-pred}\rangle \\
 & \mid & \langle\textit{binary-pred}\rangle\ \text{`}\texttt{OR}\text{'}\ \langle\textit{binary-pred}\rangle \\
\langle\textit{pred}\rangle & ::= & \text{`}\star\text{'} \mid \langle\textit{binary-pred}\rangle \\
\langle\textit{comparator}\rangle & ::= & \text{`}\texttt{==}\text{'} \mid \text{`}\texttt{!=}\text{'} \mid \text{`}\texttt{>}\text{'} \mid \text{`}\texttt{>=}\text{'} \mid \text{`}\texttt{<}\text{'} \mid \text{`}\texttt{<=}\text{'}
\end{array}
$$

Vertex and Edge (VE) queries are those which match some property values present on vertices or edges in the graph, and return the matching vertices or edges. The query predicates on each vertex or edge can be matched independently, and the result returned is either the identifiers or the complete property list of the vertices or the edges.

Query predicates can be defined as expressions using operators such as equality, greater-than and lesser-than to compare the property values in the graph against a given literal value, and these expressions can be combined using boolean operators like `AND`, `OR` and `NOT`. "$\star$" is a wildcard predicate that does no filtering, and matches all vertices or edges.

Breadth First Search (BFS) queries initially match vertices in the graph using the given vertex predicates, and further initiate a BFS traversal from each of the matching source vertices, if any, up to a specified depth. All vertices and edges in the traversal path are returned as a tree rooted at each matching vertex.

Reachability queries determine if there is a path of length within a given distance (depth) between two vertices, and returns the set of shortest paths that exist. The source and sink vertices themselves can be selected using query predicates defined for each, and if there are multiple vertices that match the source and/or sink predicates, then the shortest reachability paths between each pair of source and sink is returned.

Path queries enumerate paths matching a sequence of predicates, specified successively on vertices and edges that lie on the path. Path queries can also denote the direction of the edges that lie on the path, and allow different directions for different edges within the same path. Using $\star$ as a wildcard predicate for an edge matches all outgoing edges from a previous vertex that has been matched in the path, and similarly, a $\star$ predicate for a vertex matches any vertex that comes after a preceeding edge that has been matched.

Examples. We illustrate these query types and their results in Figs. 1b and 1c. Two sample *VE queries* to locate vertices whose `Label` attribute equals `Jim_Carrey` and edges whose `Relation` attribute equals `Had_Spouse` is shown. Vertices can also be filtered based on a topology feature, such as `out-degree>1000`. A *BFS Query* that identifies vertices matching the predicate `Label==Jim_Carrey` and traverses up to to a depth of 2 is shown. A *reachability query* to enumerate all shortest paths with distance less than 4 between source and destination vertices having predicates `Label==Ronnie_Whal` $\rightsquigarrow$ 4 $\rightsquigarrow$

`Label==Bruce_Almighty` is illustrated. An example *path query* to locate paths from any Canadian actor whose ex-Spouse is married to Ronnie Whal(berg) is also shown.

III. GoDB Architecture for Distributed Querying

We design our *Graph-oriented Database (GoDB)* as an application that runs on our GoFFish distributed graph processing platform [8], with an emphasis on supporting responsive, declarative queries over large graphs.

Briefly, *GoFFish* allows users to develop and run batch analytics over a graph partitioned across hosts on a commodity cluster or Cloud. Its imperative programming model extends Pregel's vertex-centric programming model [3] to a subgraph-centric one, where the user's logic is written for *weakly connected components (subgraphs)* within a partition. A BSP [12] model is used to execute subgraphs' logic concurrently in a *superstep*, and then exchange messages between subgraphs in-between supersteps. The supersteps iterate until all subgraphs decide to halt. Algorithms like BFS, PageRank and shortest path have been scaled to graphs with $O(10^8)$ vertices and edges, in a batch mode, taking 10-100's of seconds to run.

GoFS is the native distributed graph store of GoFFish that is designed for write-once and read-many. When a graph is loaded into GoFS, it is partitioned across machines to balance the vertices in each partition and minimize edge cuts. The subgraphs in each partition are identified, and their *structure*, including local vertices and edges, and boundary edges to remote subgraphs, and the *name-value properties* on their vertices and edges are stored on disk. Within a partition, GoFS stores the structure and properties of the subgraphs separately, and these can be materialized in-memory using a Java API.

Interactive querying over large property graphs by GoDB needs support for concurrent requests and rapid response to each, in 0.1-10 secs. Next, we explore several strategies to effectively leverage the GoFFish batch platform for low-latency transactional processing.

A. GoDB Architecture

GoDB is designed as a persistent GoFFish application (Fig. 2). The GoDB logic uses the GoFFish subgraph-centric programming and storage APIs to load and process partitions of a graphs, and uses a BSP model for messaging and coordination. In addition, GoDB maintains separate metadata on distributed storage, such as indexes and statistics.

The user submits a query through the GoFFish coordinator that is running on the head node, and this is sent as an input message to all GoDB workers in the cluster. In the first superstep, each worker parses the query and constructs a *query plan*, as described later, to execute it on the local subgraphs present on that compute node. Any data and metadata required by this query is loaded and the execution initiated.

Each GoDB worker executes the query plan on its local subgraphs and when it reaches a boundary edge to a remote subgraph, sends a state message to the neighboring GoDB worker holding that subgraph. These are available in the next superstep for further processing of the query. When a GoDB worker has completed processing the query plan as far as possible within a superstep, it ends that superstep.

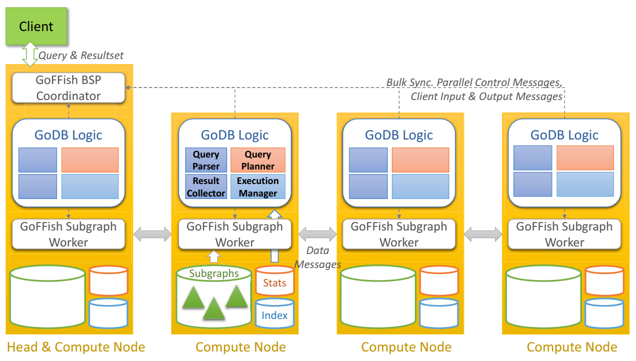


Figure 2. Architecture of GoDB, layered on top of GoFFish

GoFFish does a barrier synchronization across all workers after each superstep, and initiates the next superstep with messages delivered from remote workers. If partial or complete results for a query are available, these are collected and combined, as discussed later. When no new messages are available in a superstep and no further progress can be made for a query by any GoDB worker, the results are returned to the user through the GoFFish coordinator.

B. Graph Pre-fetching

Large property graphs can run to 10-100's of GigaBytes in size on disk. Even with a distributed deployment that utilizes the cumulative bandwidth across many disks, the time to load the complete property graph into memory can run into minutes. This is punitive for queries that need fast responses. GoDB workers mitigate this by retaining subgraphs that it loads from disk in memory so that it can be reused across multiple queries, and the disk I/O cost is amortized across queries. Here, two capabilities of GoFS are used.

One, rather than load the entire property graph into memory when a query operates on it, we only fetch those properties of the vertices or edges that are used in the query predicates. We use GoFS's Java API that allows loading of a subgraph's structure and a specific subset of its vertex and edge properties into memory. Such a *projection* of properties translates to accessing and loading only those files on disk holding those properties. For graphs with tens of properties on each vertex or edge, the disk I/O and memory savings can be substantial.

Secondly, GoFS internally has a *Least Recently Used (LRU) cache* of subgraph structures and properties that were loaded from disk by any application. Even when a GoDB worker releases a subgraph and its properties, the cache ensures that a subsequent query access to this subgraph and its properties can be re-materialized from cache rather than from disk. In future, we will consider caching strategies specific to the needs of query workloads to optimize the memory usage.

C. Property Indexes

Queries filter vertices and edges using value predicates that test if a property of a vertex or an edge matches a given value.

These matches may form the final results for VE queries, or act as *starting vertices* for a path or BFS query, or the *source and sink vertices* for a reachability query. Once the initial vertices are identified for the latter query types, the search space reduces to their edges and neighboring vertices that are then traversed for identifying paths and trees. Explicitly checking if each vertex or edge in a subgraph matches the predicate can be a costly $O(|V|)$ or $O(|E|)$ operation for each subgraph, albeit in parallel across subgraphs.

Just as in relational databases, we allow users to specify properties of vertices and edges that should be *indexed* to improve the query performance. For each partition, the GoDB worker builds an index over the values of the specified properties, and stores it in the local file system. The index is shared by all subgraphs in the partition – since some subgraphs can be tiny ($|V|$ and $|E|$ are $O(100)$), this economizes the index creation and loading time. Given a binary value predicate, the index returns the vertex or edge IDs, and the subgraph ID that have that property and value. These are then used by the query execution manager to scope the subsequent search. Since the index size is typically smaller than the graph size, the index is pre-fetched into memory and reused across queries. We use *Apache Lucene v4.10* for indexing.

D. Distributed Query Execution

Each GoDB worker independently receives a copy of the input query and parses it in the first superstep. Then, depending on the type of query – VE, BFS, Reachability or Path – different query plans are developed and executed.

For *VE queries*, each GoDB worker checks if the property used in the predicate is indexed. If so, it looks up the vertices or edges that match the predicate. If the property is not indexed, the worker iterates through all the vertices or edges of the subgraph and evaluates the predicates on each [2]. Since VE queries do not have dependencies on other elements of the graph, the results from each subgraph is returned back to the client independently, without any additional combination.

The vertex predicate in a *BFS query* is applied on each subgraph, similar to a VE query, to identify the source vertices. For each source vertex in a subgraph, the GoDB worker starts a BFS traversal within the subgraph. The worker builds a partial result tree from each traversal and marks visited vertices to avoid revisits. This proceeds until the given depth for the BFS query or a boundary edge is reached. Note that while the index is usable for initial source vertex match, subsequent matches are done through traversals since BFS does not impose any predicates on non-source vertices.

At the end of the superstep, the partial result tree for each source vertex is sent as a message to the subgraph containing any boundary vertices that were encountered during traversal. In the next superstep, subgraphs that receive partial BFS result trees from their neighbors continue the traversal until the BFS depth is reached. This may repeat, passing the partial results to neighboring subgraphs which are encountered in the traversal for processing in the next superstep, and so on. The supersteps stop when all subgraphs have reached their traversal depths. Here, we do need to combine the partial result trees into complete BFS trees for each source vertex. The subgraphs with the leaves of the tree send their partial results back to subgraphs having the source vertices, which assemble the full BFS tree to be returned back to the client.

Reachability queries extend from BFS queries. As in BFS, there can be multiple source and sink vertices that match the given predicates. We apply the source predicate on each subgraph and start BFS traversals from the matching vertices, with a traversal depth as specified in the query. Besides checking the depth when traversing, we also check if a visited vertex matches the sink vertex predicate, and if so, terminate further traversal on that path. The subgraphs stop their traversal supersteps when they have reached the maximum depth or vertices that match the sink predicate.

Now, only the subgraphs having matching sink vertices return their partial result trees back to the subgraphs with the source vertices. The source subgraph then determines the shortest path from the source to each matching sink vertex, and returns the shortest paths extracted from the tree for each source and sink vertex pair. Note that the reachability query could have instead started a reverse BFS from the sink vertices, and stopped when the source vertices were reached. In the next section, we discuss cost models that evaluate these alternate execution plans and select the optimal one.

A *Path query* has a sequence of predicates on vertices and edges. In each subgraph, we use the index, if present, to locate the vertices matching the initial vertex predicate in the path and start a traversal along each of their edges. We check the next edge predicate, and if it matches the visited edges, continue traversing to the other vertex it connects to, where the next vertex predicate is applied, and so on. Note that a path query can traverse along either the incoming or outgoing edges from each vertex.

When boundary edges are reached in the subgraph, a message is sent to the neighboring subgraph holding the remote vertex, with the location in the path query to continue traversals from. In the next superstep, the downstream subgraph picks up the traversal from the relevant vertices. Each subgraph maintains the partial results of the path fragments it has matched, but does not forwarded it to downstream subgraphs. When a subgraph matches the last predicate in a path, it sends the partial path fragments to the upstream subgraphs for each path. In the next superstep, these upstream subgraphs combine their path fragments with incoming ones to build longer path fragments, and so on till the subgraphs matching the source vertex predicate combines the incoming path fragments into the final results for paths originating from that subgraph.

Rather than initiating the query from the source vertex in the path query, the execution can also start at the sink vertex in the query, or even an intermediate vertex. The execution

[2]While a boolean combination or negation of predicates is not yet supported in GoDB, this is a straight-forward extension where the indexed properties (if any) are first executed, then a scan of the matching vertices or edges is performed for the unindexed predicate properties, and a set union or intersection performed to get the results.

cost of using different initial vertices in the path query can be different, and the next section explores these.

IV. Query Cost Model and Plan Selection

Multiple query plans are possible for distributed execution of path and reachability queries. Specifically, a reachability query can initiate a forward BFS from the source vertices or a backward BFS from the sink vertices, while the path query can initiate a traversal from any of the vertex predicates on its path. The execution cost for these different execution plans depends on several factors: *Which of the predicates are on indexed properties? How many candidate vertices and edges that have to be checked at each level of the traversal? How many partial result messages are transmitted between subgraphs at supersteps?* Next, we present a query cost model for the reachability and path queries that predict and use such information to calculate the estimated cost of each query plan.

A. Preliminaries

Let $\mathbb{V}$ be the set of vertices in a subgraph i and $\mathbb{E}$ be the set of edges. $|\mathbb{V}|$ and $|\mathbb{E}|$ are the number of vertices and edges in the subgraph, respectively. Let $\pi(a)$ be a *vertex predicate* defined on a property a of a vertex and $\widehat{\pi}(b)$ be an *edge predicate* defined on a property b of an edge.

Let $\mathbb{V}_{\pi(a)} \subset \mathbb{V}$ be the set of vertices that match $\pi(a)$, with the number of vertices matching that predicate in the subgraph, given by the *predicate cardinality*, $n_{\pi(a)} = |\mathbb{V}_{\pi(a)}|$. Similarly, let $\mathbb{E}_{\widehat{\pi}(b)} \subset \mathbb{E}$ be the set of edges that match the predicate $\widehat{\pi}(b)$, with $n_{\widehat{\pi}(b)} = |\mathbb{E}_{\widehat{\pi}(b)}|$.

Let the *probability* that a vertex in the subgraph matches the given vertex predicate be $\rho_{\pi(a)}$ and that an edge matches the edge predicate be $\rho_{\widehat{\pi}(b)}$, defined as:

$$\rho_{\pi(a)} = \frac{n_{\pi(a)}}{|\mathbb{V}|} \qquad \rho_{\widehat{\pi}(b)} = \frac{n_{\widehat{\pi}(b)}}{|\mathbb{E}|}$$

Let the *average local input* and *output edge degrees* of vertices that match a predicate $\pi(a)$ be $\lambda^{in}_{\pi(a)}$ and $\lambda^{out}_{\pi(a)}$. *Local edges* are those whose source and sink vertices lie within the same subgraph. We define:

$$\lambda^{in}_{\pi(a)} = \frac{\sum_{V_i \in \mathbb{V}_{\pi(a)}} \left(\textsc{LocalInDegree}(V_i)\right)}{|\mathbb{V}_{\pi(a)}|}$$

and similarly for the average local out degree. Further, let the *average remote input* and *output edge degrees* of vertices that match a predicate $\pi(a)$ be $\mu^{in}_{\pi(a)}$ and $\mu^{out}_{\pi(a)}$. *Remote edges* are those whose source (or sink) vertex lies in one subgraph while the sink (or source) vertex is on different subgraph. We have:

$$\mu^{in}_{\pi(a)} = \frac{\sum_{V_i \in \mathbb{V}_{\pi(a)}} \left(\textsc{RemoteInDegree}(V_i)\right)}{|\mathbb{V}_{\pi(a)}|}$$

and similarly for the average remote out degree.

Let the function $\iota(a)$ indicate if a given property a is indexed, and if so, return the relative benefit for performing a lookup of the index, as opposed to a scan of the subgraph for a non-indexed property:

$$\iota(a) = \begin{cases} 1 & \text{, if property } a \text{ is not indexed} \\ \frac{\text{Query time with index}}{\text{Query time without index}} & \text{, if property } a \text{ is indexed} \end{cases}$$

B. Reachability Query Cost Model

We first develop a cost model for the execution plans of a reachability query within a subgraph. Let $q = \pi(v_{src}) \rightsquigarrow d \rightsquigarrow \pi(v_{snk})$ be a reachability query. Recollect that a reachability query can start by executing the source vertex predicate $\pi(v_{src})$ and perform a forward BFS traversal from the vertices matching it to the vertices matching the sink predicate, or traverse backward from vertices matching the sink vertex predicate $\pi(v_{snk})$ to reach the source vertices, both within depth d. Let these two plans be $\Phi_{v_{src}}(q)$ and $\Phi_{v_{snk}}(q)$.

Here, there are no predicates defined on the BFS paths other than the source and sink vertices. Hence, assuming a uniform distribution of in and out edges across different vertices, the relative cost of these two plans will depend on the number of initial vertices matching the source or sink predicate, and the number of out or in degree edges from them, respectively. We give the relative cost of these two plans as:

$$\mathcal{C}(\Phi_{v_{src}}(q)) = |\mathbb{V}| \cdot \rho_{\pi(v_{src})} \cdot (\lambda^{out}_{\pi(v_{src})} + \mu^{out}_{\pi(v_{src})})$$
$$\mathcal{C}(\Phi_{v_{snk}}(q)) = |\mathbb{V}| \cdot \rho_{\pi(v_{snk})} \cdot (\lambda^{in}_{\pi(v_{snk})} + \mu^{in}_{\pi(v_{snk})})$$

The first plan finds the number of vertices that we expect to match in the subgraph for the source predicate, and consequently the number of *outgoing* local and remote edges from these vertices that are expected. Similarly, for the second plan, we find the number of sink vertices expected to match and the number of *incoming* edges to do a backward traversal.

Since we perform a BFS traversal from these vertices, even a small increase in the number of initial edges we start from can cause an exponential growth in the cost of traversal as the depth increases. So the cost to search for the initial vertex is itself dwarfed by these traversals, and omitted in our model. Our query execution heuristic selects the plan $\Phi_{v_{src}}(q)$ or $\Phi_{v_{snk}}(q)$ which has the lower cost.

C. Path Query Cost Model

Let $q = \pi(v_0) \vdash \widehat{\pi}(e_0) \rightarrow \pi(v_1)$ be a path query of unit length, with predicates defined on two vertices and the one edge connecting them. We have two possible plans for executing this path query, $\Phi_{v_0}(q)$ starting at predicate $\pi(v_0)$ and $\Phi_{v_1}(q)$ starting at $\pi(v_1)$.

We estimate two terms for the path query's cost model: one is the *partial cost*, c, of matching one of the elements, whether a vertex or an edge, and the other is the *predicate cardinality*, n. These terms depend on the direction of traversal: *forward* along the same direction as an edge ($\overrightarrow{c}$ and $\overrightarrow{n}$), or *backward* if in the opposite direction ($\overleftarrow{c}$ and $\overleftarrow{n}$).

Forward Plan. Considering the cost for the forward traversal first, we have the cost for matching the initial vertex with predicate $\pi(v_0)$ and the expected number of matching vertices:

$$\overrightarrow{c_{\pi(v_0)}} = |\mathbb{V}| \cdot \iota(v_0) \qquad \overrightarrow{n_{\pi(v_0)}} = |\mathbb{V}| \cdot \rho_{\pi(v_0)}$$

Since this is the first predicate being executed, it is applied to the entire subgraph which has $|\mathbb{V}|$ vertices, and this can make use of the index, if present. The number of vertices expected to be matched depends on the probability $\rho_{\pi(v_0)}$.

Next, the cost of evaluating the out edges from these initial vertices, and the number of edges expected to be matched is:

$$\begin{aligned}\overrightarrow{c_{\widehat{\pi}(e_0)}} &= \overrightarrow{n_{\pi(v_0)}} \cdot \lambda^{out}_{\pi(v_0)} \\ \overrightarrow{n_{\widehat{\pi}(e_0)}} &= \overrightarrow{n_{\pi(v_0)}} \cdot \lambda^{out}_{\pi(v_0)} \cdot \rho_{\widehat{\pi}(e_0)}\end{aligned}$$

This cost is based on each of the $\lambda^{out}_{\pi(v_0)}$ outgoing local edges from the matching initial vertices being checked – an index, even if present, is on the entire subgraph and will require a set intersection between the edge results from the index and the candidate edges that are reached. The number of edges matched depends on the probability $\rho_{\widehat{\pi}(e_0)}$. Note that the average edge degrees and the probability of a match are themselves independent of the direction of traversal.

Lastly, the cost of applying the predicate $\pi(v_1)$ to the candidate set of vertices reached from the edges matched earlier, and the number of these vertices expected to be matched are given by:

$$\overrightarrow{c_{\pi(v_1)}} = \overrightarrow{n_{\widehat{\pi}(e_0)}} \qquad \overrightarrow{n_{\pi(v_1)}} = \overrightarrow{n_{\widehat{\pi}(e_0)}} \cdot \rho_{\pi(v_1)}$$

Thus, the total cost for the forward plan is given by:

$$\mathcal{C}(\Phi_{v_0}(q)) = \overrightarrow{c_{\pi(v_0)}} + \overrightarrow{c_{\widehat{\pi}(e_o)}} + \overrightarrow{c_{\pi(v_1)}}$$

We can define a recursive forward cost model for an arbitrary path query q_d of length d which has a source vertex predicate $\pi(v_0)$, followed by multiple *path segments* with d edge and vertex predicates, $\widehat{\pi}(e_i)$ and $\pi(v_{i+1}), \forall i \in 0..(d-1)$:

$$\begin{aligned}\overrightarrow{c_{\pi(v_0)}} &= |\mathbb{V}| \cdot \iota(v_0) & \overrightarrow{n_{\pi(v_0)}} &= |\mathbb{V}| \cdot \rho_{\pi(v_0)} \\ \overrightarrow{c_{\widehat{\pi}(e_i)}} &= \overrightarrow{n_{\pi(v_i)}} \cdot \lambda^{out}_{\pi(v_i)} & \overrightarrow{n_{\widehat{\pi}(e_i)}} &= \overrightarrow{n_{\pi(v_i)}} \cdot \lambda^{out}_{\pi(v_i)} \cdot \rho_{\widehat{\pi}(e_i)} \\ \overrightarrow{c_{\pi(v_{i+1})}} &= \overrightarrow{n_{\widehat{\pi}(e_i)}} & \overrightarrow{n_{\pi(v_{i+1})}} &= \overrightarrow{n_{\widehat{\pi}(e_i)}} \cdot \rho_{\pi(v_{i+1})}\end{aligned}$$

The resulting forward query plan cost is:

$$\mathcal{C}(\Phi_{v_0}(q_d)) = \overrightarrow{c_{\pi(v_0)}} + \sum_{i \in 0..(d-1)} \left(\overrightarrow{c_{\widehat{\pi}(e_i)}} + \overrightarrow{c_{\pi(v_{i+1})}}\right)$$

and the estimated number of resultset matches is $\overrightarrow{n_{\pi(v_d)}}$.

Backward Plan. Similarly, we can define a backward traversal cost for query q_d based on starting at the last vertex and using the average *in degrees*. The recursive definition for this is:

$$\begin{aligned}\overleftarrow{c_{\pi(v_d)}} &= |\mathbb{V}| \cdot \iota(v_d) & \overleftarrow{n_{\pi(v_d)}} &= |\mathbb{V}| \cdot \rho_{\pi(v_d)} \\ \overleftarrow{c_{\widehat{\pi}(e_i)}} &= \overleftarrow{n_{\pi(v_{i+1})}} \cdot \lambda^{in}_{\pi(v_{i+1})} \\ \overleftarrow{n_{\widehat{\pi}(e_i)}} &= \overleftarrow{n_{\pi(v_{i+1})}} \cdot \lambda^{in}_{\pi(v_{i+1})} \cdot \rho_{\widehat{\pi}(e_i)} \\ \overleftarrow{c_{\pi(v_i)}} &= \overleftarrow{n_{\widehat{\pi}(e_i)}} & \overleftarrow{n_{\pi(v_i)}} &= \overleftarrow{n_{\widehat{\pi}(e_i)}} \cdot \rho_{\pi(v_i)}\end{aligned}$$

The resulting forward query plan cost is:

$$\mathcal{C}(\Phi_{v_d}(q_d)) = \overleftarrow{c_{\pi(v_d)}} + \sum_{i \in (d-1)..0} \left(\overleftarrow{c_{\widehat{\pi}(e_i)}} + \overleftarrow{c_{\pi(v_i)}}\right)$$

and the estimated number of resultset matches is $\overleftarrow{n_{\pi(v_0)}}$.

Mixed Plan. In practice, users can specify paths that do not strictly go in one direction, with different edges in the path varying in direction. The partial sums of forward and backward traversals of path segments from the above equations can be used to evaluate such cases.

Further, the path query may be executed by starting at some *intermediate* (non-terminal) vertex in the query, thus splitting it into two paths (left and right) that are evaluated separately and then joined using a *cross-product* of the individual results. This will effectively enumerate all the results of the original query. Given a path query q_d with length d, it offers $d+1$ possible query plans, $\Phi_{v_i}(q_d), i \in 0..d$, each starting at one of the vertices in the path. Each of their costs are evaluated by our heuristic to find the plan with the least cost.

When starting at an intermediate vertex, in addition to the cost of evaluating the left and the right paths, we also have an additional *join cost*. This is given by the product of the size of the resultsets from the left and right paths. If vertex predicate $\pi(v_i), 1 \le i < d$, is where a query q_d is split into two, $q_{0..i}$ and $q_{i..d}$, then its join cost is:

$$j_{\pi(v_i)} = \gamma \cdot |q_{0..i}| \cdot |q_{i..d}|$$

Here, γ is an *aggregation coefficient* that is the ratio of the cost to join results from a pair of paths and the cost to compute a query predicate on a vertex or edge. The total query plan cost is the cost of executing the left and right paths, and the join:

$$\begin{aligned}\mathcal{C}(\Phi_{v_i}(q_d)) = \; & \min\big(\mathcal{C}(\Phi_{v_0}(q_{0..i})), \mathcal{C}(\Phi_{v_i}(q_{0..i}))\big) + \\ & \min\big(\mathcal{C}(\Phi_{v_i}(q_{i..d})), \mathcal{C}(\Phi_{v_d}(q_{i..d}))\big) + \\ & j_{\pi(v_i)}\end{aligned}$$

The plan with the lowest cost among all the vertex split points in the path is selected by our heuristic.

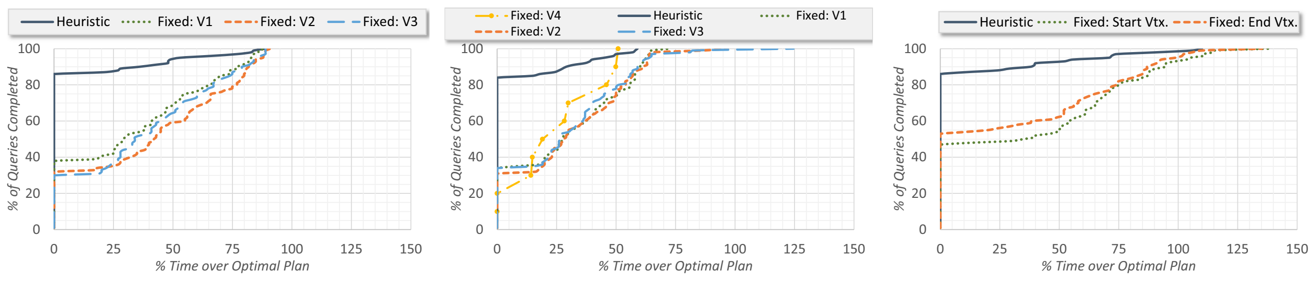
Network Communication Cost. These earlier costs have only considered the execution within a single subgraph. However, in a *distributed execution*, there are network costs for exchanging messages with neighboring subgraphs in a different machine, holding remote vertices. We model the overhead of messages passing across the network as the number of remote edges that we encounter in the traversal. Given a forward path segment, $\pi(v_i) \vdash \widehat{\pi}(e_i) \rightarrow$, the network cost is:

$$\overrightarrow{w_{\widehat{\pi}(e_i)}} = \omega \cdot \overrightarrow{n_{\pi(v_i)}} \cdot \mu^{out}_{\pi(v_i)} \cdot \rho_{\widehat{\pi}(e_i)}$$

which is a product of the estimated number of vertex results from $\pi(v_i)$, its average remote out edge degree, and the probability that the remote edge matches its edge predicate $\widehat{\pi}(e_i)$. ω is a *communication coefficient* that gives the ratio between the cost to transfer state to a remote vertex relative to a query predicate evaluation time. The backward traversal network cost $\overleftarrow{w_{\widehat{\pi}(e_i)}}$ instead uses the remote in edge degree, $\mu^{in}_{\pi(v_{i+1})}$. This network cost is added to each path segment that is evaluated to get the total cost.

D. Implementation

Each GoDB worker upon receiving a reachability or path query, uses the above model to calculate the execution costs for the different execution plans. To this end, each worker locally maintains statistics on the different parameters and coefficients for subgraphs, that are used by the cost model, viz., ρ, λ and μ for each predicate, and γ, ι and ω that are predicate independent. Each worker exchanges their estimated cost for each possible query plan with all other workers at the end of the first superstep. This allows the workers to estimate the global cost for each query plan, summed across all workers,



(a) Selective Path Queries (b) Report Path Queries (c) Reachability Queries

Figure 3. Cumulative distribution function (CDF) of the % of queries that are able to complete (Y Axis) within a % of time greater than the optimal plan for the query (Y Axis). Plan selection based on heuristic cost model and fixed vertices are shown for Selective, Report and Reachability query types.

in the second superstep, and they all pick the same lowest cost plan independently and initiate the execution of this plan.

V. Evaluation

We use several real-world and synthetic graphs to evaluate GoDB. The primary graph used in all experiments is the Patent Citation Graph (CITP) [10], which is formed from citations made from patents granted by the US Patent Office over 37 years. We have $3,764,110$ patents as vertices, with 4 attributes on each, `patent_class`, `grant_year`, `country` and `state`, and $16,511,699$ citation edges.

We also run our experiments on a smaller NELL knowledge graph [2]. But, due to space constraints, we do not present the results for NELL but observe similar behavior as CITP.

For our workload, we use 5 different query types – *Vertex or Edge (VE), Breadth First Search (BFS), Selective Path, Report Path,* and *Reachability*. While vertex and edge queries match individual vertices and edges having the given predicate, BFS matches multiple source vertices and starts a traversal of given depth over it. Selective and Report queries are path queries of a given depth, with predicates defined on each vertex of the path; Selective queries return a few (tens) of paths as resultset while Report queries have a large (thousands) resultset size. Reachability query matches two sets of source and sink vertices based on given predicates and determines if there is a path within a given length between them.

For each query type, we generate 100 synthetic queries that define a random value predicate on the vertex or edge of CITP, as relevant. We also validate that the query returns at least a single result. This gives us a workload of 500 queries.

GoDB uses GoFFish v2.6 and Java v1.7, and is deployed on Microsoft Azure IaaS Cloud VMs running CentOS v7.0. We use D2 VM types in the Southeast Asia data-centre, and these VMs have 2 Intel Xeon 2.2 GHz cores, 7GB RAM, 100 GB of local SSD disk and are connected by Ethernet. The graphs were partitioned using METIS. Each VM as 1 partition per core, and each partition has 1 subgraph.

The queries were submitted using a web service call to the coordinator node. We only measure the time taken for executing the query, and not the time for indexing, or loading the graph and index into memory – since these are amortized across queries – or the time for writing the resultsets to disk or to console. We use sample runs of path queries on the CITP graph using GoDB to determine the cost model's coefficients.

A. *Impact of Query Cost Model Heuristic*

We first compare the benefits of the cost model in helping GoDB select a distributed query execution plan, compared to a fixed query plan. We execute 100 each of the Selective queries of path length 3, Report queries with a mix of path lengths of 3 and 4 (90:10), and Reachability queries with a distance ≤ 3 and ≤ 4 (50:50) using the heuristic query plan selection. Vertex value predicates are defined on *indexed* properties, unless otherwise noted.

In addition, for each query, we also evaluate multiple *fixed query plans* that start at each possible predicate, e.g., at vertex 1, 2, 3 or 4 for path queries with length 4, and from either the source or the sink vertex for reachability queries. For each of the 300 queries, we determine the *optimal plan* as the one that gives the smallest runtime after execution, i.e., it is an "oracle". For the heuristic and the fixed plans, we then determine by what fraction their runtime for a query was greater than the optimal plan. The ideal case is 0% over optimal for all queries.

Figs. 3a, 3b and 3c show the cumulative probability density function for the fraction of queries (on the Y axis) that complete with a fraction of time greater than the optimal time (X Axis) for all the query plans. As we see, for all three query types, our heuristic is able to achieve the optimal execution time for 85% of the queries. Each of the fixed plans, on the other hand, are unable to match the optimal plan for more than $35-59\%$ of the queries, depending on the query type. We see that at the 90^{th} percentile the query is $59-95\%$ slower than the optimal for the fixed query plans. The heuristic plan shows a long tail from the $85-100^{th}$ percentile in the growth of query times, having a worst case of $59-110\%$ above optimal time, depending on the query type. The fixed plans have a worst case of $72-138\%$ over the optimal time. The only exception is the Selective queries with path length 4, of which there were just 10, that have a worst case of 51% over optimal.

These empirically demonstrate that our cost model heuristic is effective for these three types of queries, and offer up to 88% benefit in execution times compared to fixed query plans. The rest of the experiments use the heuristic plan.

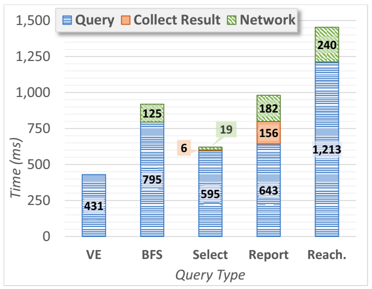


Figure 4. Query, network & result collection times for different query types.

B. Performance of Query Types

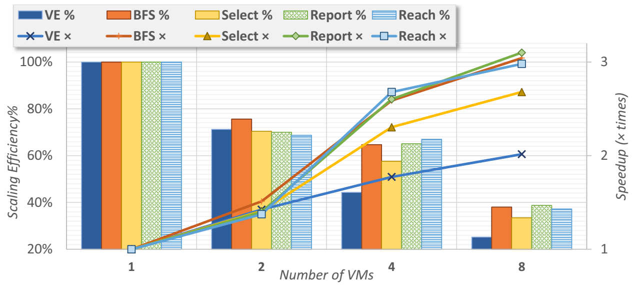
For different query types, we examine the breakup of the time spent on performing the *distributed queries* on partitions, the *communication cost* for messaging between partitions, and *distributed collection* and combination of partial results at the end, if relevant. Fig. 4 shows a stacked plot of these average component times for 100 queries each of the 5 types.

As expected, pure Vertex and Edge queries are embarrassingly parallel and do not require any traversal across partitions or result combinations; hence they spend all their time on query computation. BFS and reachability queries spend time in query computation and on the network, but do not have to combine any partial results. The time spent on the network is relatively smaller than the query computation, accounting for only $14-17\%$ of the total time. Selective queries spend some time on result collection, but given their small resultsets, the time spent on messaging across partitions and the result collection together is less than 4% of the total time. Report queries, on the other hand, have large resultsets, and these cause 19% of the total time to be spent on the network and 16% of the time in combining results.

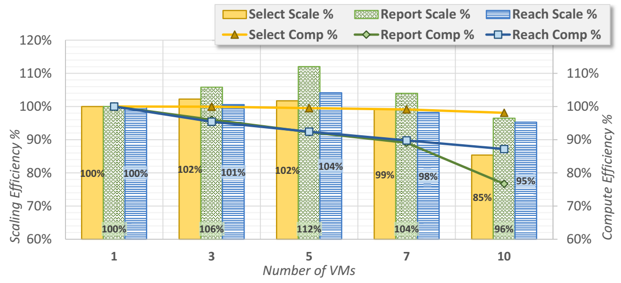
C. Strong and Weak Scaling

We evaluate the scalability of distributed execution on multiple VMs, and examine both the strong- and weak-scaling properties of GoDB. Fig. 5a shows the average time taken to run the query workloads on a GoDB deployment on 1, 2, 4 and 8 VMs for CITP. Note that since a D2 VM with 7GB RAM could not hold the entire CITP graph, the 1 VM case alone used a D11 Azure VM with twice the memory (14GB), but all other configurations the same as a D2.

As we can see, the benefits of adding more VMs are greater initially, giving a speedup of $\sim 1.4\times$ with 2 VMs, relative to 1 VMs, and $1.7-2.6\times$ with 4 VMs, with scaling efficiencies of $\sim 70\%$ and $44-67\%$, respectively. However, this drops off as we increase the number of VMs to 8, where we only get a peak $3.1\times$ speedup and 39% efficiency. We note that the time spent in the network increases to 23% for 8 VMs, while the relative reduction in query time does not keep up. Interestingly, the VE queries show the poorest strong-scaling while we might expect a pleasingly-parallel execution. The reason is the index – the index lookup for VE queries is nearly constant, and a linear drop in the index entries per VM does not lead to a reduction in index lookup or query time.



(a) Strong-scaling of GoDB on CITP graph



(b) Weak-scaling of GoDB on synthetic graph (2M V & 5M E per VM)

Figure 5. Scalability of GoDB.

While GoDB shows poor strong-scaling, we do see that it exhibits good weak-scaling as we increase the graph size with a corresponding increase in the number of VMs. Fig. 5b shows a query workload being executed on a synthetic graph generated using RMAT [13] along with synthetic properties with a Gaussian distribution of their values. We set the number of vertices and edges *per VM* to be 2M and 5M, respectively, and run the experiment on $1-10$ VMs.

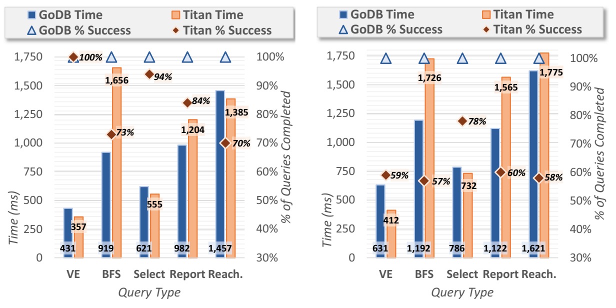
As we can see, the time taken for the query workload mostly remains constant, with a $85-112\%$ scaling efficiency. We also see that the compute efficiency – the % of time spent in compute rather than communication – drops only gradually, thus showing that the penalty for distributed communication is largely offset by the gains of increased compute speeds, thus demonstrating the weak-scaling of GoDB.

D. Comparative Performance

We compare the execution of these CITP graph query workloads on GoDB against *Titan*, a popular open-source distributed graph database [7] that has shown to scale well for large graphs [3]. We deploy Titan on 4 Azure D2 VMs and configure it to use Apache Cassandra as the backend storage. The CITP graph is loaded onto Titan and Elastic Search is used for indexing. The 100 queries from the 5 query types are implemented using the Gremlin language, and executed from Titan's command shell running on the head VM.

Fig. 6a shows the average query time (left Y Axis) taken by each workload on GoDB and Titan, when all query predicates are on indexed properties. In addition, we also plot the fraction of the 100 queries that could complete within a 5 *min time budget* on each platform (right Y Axis). We see that for simple

[3]Educating the planet with Pearson, May 2013, http://goo.gl/ldHxGN



(a) *All* properties indexed (b) *At least one* unindexed property

Figure 6. Runtime of different query types (Left Y Axis) and the % of queries that complete within 5 mins (Right Y Axis), on GoDB and Titan.

Vertex and Edge queries, both GoDB and Titan are able to complete all queries within the time limit, with GoDB being slightly slower than Titan, taking an average of 431 ms to complete each query compared to 357 ms taken by Titan.

The times taken for the Selective queries are also comparable, with GoDB slightly slower at 621 ms compared to 555 ms for Titan. However, here we see that Titan is able to complete only 94% of all Selective queries with this average time, and for the remaining 6 queries, it fails to complete in 5 mins (not included in average time). GoDB's average time includes all 100 queries. This completion % gets worse for Reachability queries where only 70% of queries can complete on Titan within the time limit while all of them complete on GoDB, even as the average time of GoDB is 5% slower. When we get to BFS and Report queries, GoDB is much faster than Titan (80% and 23%), and further, Titan is unable to complete 27% and 30% of the queries on time, respectively.

We further see the advantage of GoDB relative to Titan in cases where *all attributes are not indexed*. Here, we use a similar query workload but ensure that at least one of the query predicates is not on an indexed attribute. In practice, it may be costly to index all properties, so this offers a realistic scenario. Fig. 6b shows the time taken by the query workloads and their completion statistic. For the most part, we see a modest increase in relative average execution times of between $4-47\%$ for both GoDB and Titan. But Titan's ability to complete the queries within the time budget drops even further in this case. In fact, it is not able to complete even a single query type workload completely, and finishes only $57-78\%$ of the queries, depending on the type. GoDB on the other hand completes all queries, and is even able to better Titan on its Reachability query time. So GoDB is uniformly responsive and scalable for both simple and costly queries, while Titan cannot complete a large fraction of the query workload in time.

We can perform a nominal, illustrative comparison against *Horton* [14] in the absence of availability of their platform, graphs, or queries publicly. Our CITP graph is slightly larger than their real graph (Vertices/Edges: $3.77M/16.5M$ vs. $2.91M/13.61M$), and we run similar categories of Selective, Report and Reachability (topological Closure) queries. On 4 quad-core Xeon 2.6 GHz physical machines with 16 GB RAM, they achieve average query runtimes of 60, 900 and 62 secs, while we run on 4 dual-core VMs with 7 GB RAM and achieve average query times of 0.62, 0.98 and 1.46 secs, which are almost *two orders of magnitude faster*. This can partly be explained by our use of a subgraph-centric BSP execution model (as opposed to their vertex-centric BSP) that allows queries to progress farther within each superstep, and reduces the communication cost and supersteps to converge.

VI. Related Work

Scalable graph processing has focused on two key aspects: distributed batch processing of graphs, and graph databases for interactive querying. We review these here.

Google's *Pregel* [3] model of vertex-centric graph processing overcomes the shortcomings of MapReduce to handle iterative batch processing of graphs [15]. Pregel leverages vertex-level parallelism and supports fault-tolerance, but its focus is on batch processing for algorithms such as connected components and PageRank, and not interactive queries. Also, their data model does not explicitly support named/typed attributes necessary for querying. Other like *Giraph++*, *Blogel* [16] and *GPS* have improved the distributed programming model and introduced execution optimizations. Our prior work [8] extends Pregel's vertex-centric model to a subgraph-centric one to reduce costly messaging between vertices. However, none of these support declarative queries over property graphs, or are designed for responsive applications running within seconds.

GraphX [17], built on top of Apache Spark, provides Resident Distributed Graphs which describe transformations on graphs to yield a new graph. While GraphX has some declarative capabilities offered by the underlying Spark, this is used more to duplicate a vertex-centric model. *GraphLab* [4] is another distributed vertex-centric graph platform that, instead of message passing like Pregel, gives vertices direct read and write access to their neighboring vertex's data. GraphLab does not expose any declarative graph query model and logic written explicitly is executed in batch mode.

Trinity [18] is a graph engine built as a distributed in-memory key-value store to support online and offline graph processing. Interactive querying in Trinity is efficient due to in-memory access to graph, similar to GoDB. However, it relies on clusters with high-speed networks and large physical memory. We instead focus on commodity clusters and Clouds.

The *Titan* [7] distributed graph database supports scalable backend storage like Cassandra and HBase, and indexing using ElasticSearch and Lucene. It is designed for concurrent transactions and complex queries, but does not support graph analytics algorithms. Queries are written using their Blueprints API or Gremlin language. While Titan is optimized for thousands of concurrent queries, unlike us, their performance short-comings have been demonstrated here. *GBase* [5] is an analysis platform for large graphs. They use compressed block encoding for efficient graph storage. They identify global queries which require traversal of the whole graph and targeted queries which access only parts of the graph. The queries are converted to matrix-vector multiplications which correspond to SQL join operations and are executed on Hadoop.

Horton+ [14] is the closest to our work in literature. It executes declarative queries on partitioned attributed multi-graphs that are maintained in memory. Their query language is transformed into a DFA of transition predicates using three operators: *select, traverse* and *join*. The *select* operator queries for vertices while the traverse performs a BFS, similar to our VE and BFS queries. Their explicit *join* operator is more powerful than our implicit join of partial path query splits. They also support full closure queries unlike our topological closure using reachability queries. Their operators can traverse over the edges and across multiple partitions, and this is managed by BSP synchronization and message passing.

There are however several distinctions in the execution model. They do not explicitly support indexes and instead hold all primary key-vertex ID mappings in their head-node to scope a vertex predicate search to a specific partition. Our indexes allow a fast lookup for indexed vertices (or edges), in parallel on all GoDB workers. Also, our subgraph-centric model takes much fewer supersteps to converge and is able to query a larger part of graph (i.e., a subgraph) in one superstep. They use a vertex-centric execution model that requires at least as many supersteps as the length of the query path. We also perform a distributed collection of results that has shown to be faster than a centralized collect of results that they use. Their cost model also uses additional statistics on each triple of vertex-edge-vertex, which can be costly to calculate and store compared to a single statistic per predicate that we use.

Other works [19] uses histogram based approaches for determining selectivity of twig queries (parts of structural XPath queries). They perform a prime labelling of the graph nodes and use binary matrix based representation and transformations. XPath based estimations cannot be used in all cases when handling cyclic graphs. [20] is based on a vertex sketch partitioning technique with graph stream data and query workload data. They estimate query selectivity on specific graph queries and are complementary to our cost model.

Rather than develop a distributed graph database from scratch, GoDB instead leverages existing work on scalable distributed batch processing of graphs and extends that to interactive queries. It supports similar query models as the graph databases, with the added ability to perform complex graph queries at scale using it batch processing engine.

VII. Conclusions

We have proposed GoDB, a distributed graph database that extends from an existing scalable distributed graph processing system, GoFFish, going from a batch to an interactive execution model. Our work has explored the query model and distributed execution strategies used to support declarative VE, reachability and path query types. We have also introduced a cost model for evaluating query plans which selects the optimal plan $\geq 80\%$ of the time. GoDB is validated against real world graphs and large query workloads. It shows fast response times for most queries and also exhibits weak scaling. When compared to the Titan graph database, it exhibits comparable query latencies, and out-performs in completing all the queries within a 5 min time budget while Titan fails to execute as many as 40% of queries in this time, for some workloads.

There are several novel research ideas we plan to explore further. We propose to support declarative queries that include complex algorithms, such as using PageRank as an in-place function in the query. Further, we plan to examine the possibility of queries over time-series graphs, where multiple similar graphs collected over time have to be examined. For e.g., the NELL graph is built every month and queries to examine knowledge evolution need to operate across all these graphs. Lastly, we will examine scaling to larger graphs and for many concurrent queries. Here, piggy-backing batches of queries over the same traversal will help increase the throughput [4].

References

[1] T. Berners-Lee, J. Hendler, and O. Lassila, "The semantic web," *Scientific American*, 2001.

[2] T. Mitchell, et al., "Never-ending learning," in *AAAI*, 2015.

[3] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *SIGMOD*, 2010.

[4] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: a framework for machine learning and data mining in the cloud," *PVLDB*, vol. 5, no. 8, 2012.

[5] U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos, "Gbase: an efficient analysis platform for large graphs," *VLDBJ*, vol. 21, no. 5, pp. 637–650, 2012.

[6] M. Sarwat, S. Elnikety, Y. He, and G. Kliot, "Horton: Online query execution engine for large distributed graphs," in *ICDE*. IEEE, 2012, pp. 1289–1292.

[7] "Titan," http://thinkaurelius.github.io/titan/.

[8] Y. Simmhan, A. Kumbhare, C. Wickramaarachchi, S. Nagarkar, S. Ravi, C. Raghavendra, and V. Prasanna, "Goffish: A sub-graph centric framework for large-scale graph analytics," in *Euro-Par*, 2014.

[9] M. A. Rodriguez and J. Shinavier, "Exposing multi-relational networks to single-relational network analysis algorithms," *Journal of Informetrics*, vol. 4, no. 1, pp. 29–41, 2010.

[10] "US Patent Citation Network," Graph at https://snap.stanford.edu/data/cit-Patents.html, Attributes at http://www.nber.org/patents/.

[11] P. T. Wood, "Query languages for graph databases," *ACM SIGMOD Record*, vol. 41, no. 1, pp. 50–60, 2012.

[12] L. G. Valiant, "A bridging model for parallel computation," *CACM*, vol. 33, no. 8, pp. 103–111, 1990.

[13] F. Khorasani, K. Vora, and R. Gupta, "Parmat: A parallel generator for large r-mat graphs," 2015.

[14] M. Sarwat, S. Elnikety, Y. He, and M. F. Mokbel, "Horton+: a distributed system for processing declarative reachability queries over partitioned graphs," *PVLDB*, vol. 6, no. 14, pp. 1918–1929, 2013.

[15] J. Cohen, "Graph twiddling in a mapreduce world," *Computing in Science & Engineering*, vol. 11, no. 4, pp. 29–41, 2009.

[16] D. Yan, J. Cheng, Y. Lu, and W. Ng, "Blogel: A block-centric framework for distributed computation on real-world graphs," *PVLDB*, vol. 7, no. 14, 2014.

[17] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "Graphx: A resilient distributed graph system on spark," in *Workshop on Graph Data Management Experiences and Systems*, 2013.

[18] B. Shao, H. Wang, and Y. Li, "Trinity: A distributed graph engine on a memory cloud," in *SIGMOD*, 2013.

[19] Y. Peng, B. Choi, and J. Xu, "Selectivity estimation of twig queries on cyclic graphs," in *ICDE*. IEEE, 2011, pp. 960–971.

[20] P. Zhao, C. C. Aggarwal, and M. Wang, "gsketch: on query estimation in graph streams," *PVLDB*, vol. 5, no. 3, pp. 193–204, 2011.

[4]This work is supported by grants from NetApp Inc. and Microsoft Azure for Research. We thank Prof. Partha Talukdar from IISc for his inputs on the NELL knowledge graph and queries.